aws

Entwicklerhandbuch

AWS Lambda

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

AWS Lambda: Entwicklerhandbuch

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

Die Marken und Handelsmarken von Amazon dürfen nicht in einer Weise in Verbindung mit nicht von Amazon stammenden Produkten oder Services verwendet werden, die geeignet ist, Kunden irrezuführen oder Amazon in irgendeiner Weise herabzusetzen oder zu diskreditieren. Alle anderen Marken, die nicht im Besitz von Amazon sind, gehören den jeweiligen Besitzern, die möglicherweise mit Amazon verbunden sind oder von Amazon gesponsert werden.

Table of Contents

Was ist AWS Lambda? 1
Funktionsweise von Lambda 1
Schlüssel-Features 2
Ähnliche Informationen 3
Funktionsweise 3
Lambda-Funktionen und -Funktionshandler 4
Lambda-Ausführungsumgebung und -Laufzeiten 5
Ereignisse und Auslöser 5
Lambda-Berechtigungen und -Rollen 6
Ausführen von Code 9
Erstellen ereignisgesteuerter Architekturen 26
Entwerfen einer Anwendung 39
Erstellen Sie Ihre erste Funktion 47
Voraussetzungen 47
Erstellen der Funktion 49
Aufrufen der -Funktion 56
Bereinigen 60
Nächste Schritte 61
Beispiel-Apps und -Muster 63
Dateiverarbeitung 63
Datenbankintegration 63
Geplante Aufgaben 64
Workflows mit langer Laufzeit 64
Weitere Ressourcen 64
Datei-Bearbeitungs-App 65
Erstellen der Quellcodedateien 66
Bereitstellen der App 69
Testen der Anwendung 82
Nächste Schritte 89
App für planmäßige Wartung 90
Voraussetzungen 91
Die Beispiel-App-Dateien werden heruntergeladen 92
DynamoDB-Beispieltabelle erstellen und auffüllen 102
Die App für geplante Wartungsarbeiten erstellen 105

Testen der App ... 110
Nächste Schritte ... 111
Erstellung eines Auftragsabwicklungssystems mit Lambda Durable-Funktionen ... 112
Voraussetzungen ... 112
Erstellen Sie die Quellcodedateien ... 112
Bereitstellen der Anwendung ... 113
Testen der Anwendung ... 114
Nächste Schritte ... 114
Entwicklungstools ... 116
Lokale Entwicklungstools ... 116
Infrastructure as Code (IaC)-Tools ... 117
GitHub-Actions-Tools ... 118
Powertools for AWS Lambda ... 118
Tools für die Workflow- und Ereignisverwaltung ... 119
Lokale Entwicklung ... 119
Hauptvorteile der lokalen Entwicklung ... 119
Voraussetzungen ... 119
Authentifizierung und Zugriffskontrolle ... 120
Wechsel von der Konsole zur lokalen Entwicklung ... 123
Lokales Arbeiten mit Funktionen ... 124
Konvertieren Ihrer Funktion in eine AWS SAM-Vorlage und Verwenden von IaC-Tools ... 127
Nächste Schritte ... 127
GitHub Actions ... 128
Beispiel-ArbeitsFlows ... 128
Weitere Ressourcen ... 129
Infrastructure as Code (IaC) ... 129
IaC-Tools für Lambda ... 129
Verwenden von AWS SAM und Infrastructure Composer ... 131
Verwenden AWS CDK ... 144
Powertools for AWS Lambda ... 154
Hauptvorteile von Powertools for AWS ... 154
Integration von Powertools mit AWS ... 154
Nächste Schritte ... 127
Workflows und Ereignisse ... 155
Orchestrierung, bei der der Code an erster Stelle steht, mit dauerhaften Funktionen ... 156
Orchestrierung von Workflows mit Step Functions ... 158

Verwaltung von Ereignissen mit und Scheduler EventBridge EventBridge 159
Langlebige Lambda-Funktionen 161
Wichtigste Vorteile 161
Funktionsweise 162
Wann sollten langlebige Funktionen verwendet werden 163
Nächste Schritte 164
Grundkonzepte 164
Dauerhafte Ausführung 165
DurableContext 165
Schritte 167
Warte, Staaten 167
Andere Funktionen aufrufen 168
Dauerhafte Funktionskonfiguration 169
Schaffung dauerhafter Funktionen 171
Voraussetzungen 172
Erstellen Sie die dauerhafte Funktion 174
Rufen Sie die dauerhafte Funktion auf 181
Bereinigen 183
Nächste Schritte 184
Verwenden AWS CLI 184
Infrastruktur als Code verwenden 191
Beispiele 198
Kurzlebige fehlertolerante Prozesse 198
Prozesse mit langer Laufzeit 204
Funktionsübergreifende verkettete Aufrufe 211
Fortgeschrittene Muster 215
Nächste Schritte 224
Sicherheit und Berechtigungen 224
Berechtigungen für die Ausführungsrolle 224
Staatliche Verschlüsselung 226
CloudTrail Protokollierung 227
Kontoübergreifende Überlegungen 228
Vererbte Lambda-Sicherheitsfunktionen 228
SDK für langlebige Ausführung 228
DurableContext 228
Was macht das SDK 229

Wie funktioniert Checkpointing ... 230
Verhalten bei der Wiedergabe ... 231
Verfügbare dauerhafte Operationen ... 232
Wie lange Betriebsabläufe gemessen werden ... 239
Unterstützte Laufzeiten ... 242
Dauerhafte Funktionen aufrufen ... 246
Methoden zum Aufrufen ... 246
Qualifizierte Anforderung ARNs ... 247
Den Ausführungslebenszyklus verstehen ... 249
Aus dem Anwendungscode aufrufen ... 249
Verkettete Aufrufe ... 251
Zuweisung von Ereignisquellen ... 252
Erneute Versuche ... 260
Idempotenz ... 267
Testen langlebiger Funktionen ... 273
Lokale Tests ... 273
Testen in der Cloud ... 276
Was muss getestet werden? ... 278
Teststrategie ... 278
Fehler beim Debuggen ... 279
Überwachung dauerhafter Funktionen ... 279
CloudWatch Metriken ... 280
EventBridge Ereignisse ... 282
AWS X-Ray Rückverfolgung ... 284
Best Practices ... 285
Schreiben Sie deterministischen Code ... 285
Design für Idempotenz ... 289
Verwalten Sie den Status effizient ... 291
Entwerfen Sie effektive Schritte ... 293
Verwenden Sie Warteoperationen effizient ... 293
Weitere Überlegungen ... 294
Weitere Ressourcen ... 295
Von Lambda verwaltete Instanzen ... 296
Die wichtigsten Funktionen ... 296
Wann sollten Lambda Managed Instances verwendet werden ... 296
Funktionsweise ... 297

Modell der Parallelität ... 297
Tenance und Isolierung ... 298
Verwaltete Instanzen verstehen ... 298
Preisgestaltung ... 298
Wie sich Lambda Managed Instances vom Compute-Typ Lambda (Standard) unterscheiden ... 299
Nächste Schritte ... 300
Erste Schritte ... 300
Eine Lambda Managed Instance-Funktion erstellen (Konsole) ... 300
Eine Lambda Managed Instance-Funktion (AWS CLI) erstellen ... 302
Schlüsselkonzepte ... 308
Kapazitätsanbieter ... 309
Skalierung ... 313
Sicherheit ... 317
Rolle des Operators ... 319
Ausführungsumgebung ... 322
Veröffentlichung von Versionen ... 324
Laufzeiten ... 325
Unterstützte Laufzeiten ... 325
Laufzeitspezifische Überlegungen ... 325
Nächste Schritte ... 326
Java-Laufzeit ... 326
Laufzeit von Node.js ... 331
Python-Laufzeit ... 337
.NET-Laufzeit ... 339
Netzwerk ... 344
Verbindungsoptionen ... 344
Öffentliches Subnetz mit einem Internet-Gateway ... 344
VPC-Endpunkte ... 346
Privates Subnetz mit NAT-Gateway ... 346
Auswahl einer Konnektivitätsoption ... 347
Nächste Schritte ... 348
Überwachen ... 348
Verfügbare Metriken ... 348
Metrische Häufigkeit und Retention ... 350
Metriken anzeigen in CloudWatch ... 350
Verwenden von Metriken zur Leistungsoptimierung ... 350

Nächste Schritte ... 351
Kontingente ... 351
Lambda-API-Anforderungskontingente ... 351
Ressourcenkontingente für Lambda Managed Instances ... 351
Kontingente für die Zuordnung von Ereignisquellen ... 352
Beantragen einer Kontingenterhöhung ... 353
Nächste Schritte ... 353
Best Practices ... 353
Konfiguration des Kapazitätsanbieters ... 353
Auswahl des Instance-Typs ... 354
Konfiguration der Funktionen ... 354
Skalierungskonfiguration ... 355
Sicherheit ... 355
Kostenoptimierung ... 356
Überwachung und Beobachtbarkeit ... 356
Laufzeitspezifische Überlegungen ... 357
Nächste Schritte ... 357
Fehlerbehebung ... 357
Probleme bei der Drosselung und Skalierung ... 357
Probleme mit der Parallelität ... 358
Leistungsprobleme ... 361
Probleme mit dem Kapazitätsanbieter ... 362
Probleme bei der Überwachung und Beobachtbarkeit ... 363
Weitere Hilfe ... 365
Nächste Schritte ... 366
Lambda-Laufzeiten ... 367
Unterstützte Laufzeiten ... 368
Neue Laufzeit-Versionen ... 371
Richtlinie für den Laufzeitablauf ... 372
Modell der geteilten Verantwortung ... 373
Unterstützung der Verwendung nach der Einstellung ... 374
Empfangen von Benachrichtigungen über veraltete Laufzeitversionen ... 376
Veraltete Laufzeitumgebungen ... 378
Aktualisierungen der Laufzeitversion ... 382
Abwärtskompatibilität ... 383
Modi der Laufzeitaktualisierung ... 385

Zweiphasiges Rollout der Laufzeitversion ... 386
Konfigurieren der Laufzeitverwaltung ... 387
Rollback der Laufzeitversion ... 388
Aktualisierungen der Laufzeitversion ... 390
Modell der übergreifenden Verantwortlichkeit ... 392
Berechtigungen ... 394
Daten über Funktionen zur Laufzeit abrufen ... 395
Auflisten von Funktionsversionen, die eine bestimmte Laufzeit verwenden ... 395
Identifizieren der am häufigsten und zuletzt aufgerufenen Funktionen ... 397
Änderungen an der Laufzeitumgebung ... 402
Sprachspezifische Umgebungsvariablen ... 402
Wrapper-Skripte ... 402
Laufzeit-API ... 406
Nächster Aufruf ... 407
Aufrufantwort ... 408
Initialisierungsfehler ... 409
Aufruffehler ... 410
Reine OS-Laufzeiten ... 413
Erstellen einer benutzerdefinierten Laufzeit ... 414
Tutorial für eine benutzerdefinierte Laufzeit ... 420
Open-Source-Repositorys ... 428
Laufzeit-Schnittstellenclients ... 428
Ereignisbibliotheken ... 429
Container-Basisimages ... 430
Entwicklungstools ... 430
Beispielprojekte ... 431
Konfigurieren von -Funktionen ... 432
ZIP-Dateiarchive ... 435
Erstellen der Funktion ... 435
Verwenden des Konsolencode-Editors ... 437
Aktualisieren des Funktionscodes ... 437
Ändern der Laufzeit ... 438
Ändern der Architektur ... 439
Verwenden der Lambda-API ... 439
Herunterladen des Funktionscodes ... 439
CloudFormation ... 440

Verschlüsselung ... 441
Container-Images ... 448
Voraussetzungen ... 450
Verwenden eines AWS Basis-Images ... 450
Es wird ein AWS reines Betriebssystem-Basis-Image verwendet ... 451
Verwenden eines AWS Nicht-Base-Images ... 452
Laufzeitschnittstellen-Clients ... 453
Amazon-ECR-Berechtigungen ... 453
Lebenszyklus der Funktion ... 456
Arbeitsspeicher ... 458
Wann sollte der Speicher erhöht werden ... 458
Verwenden der Konsole ... 459
Mit dem AWS CLI ... 459
Benutzen AWS SAM ... 460
Akzeptieren von Empfehlungen für den Funktionsspeicher (Konsole) ... 460
Flüchtiger Speicher ... 462
Anwendungsfälle ... 462
Verwenden der Konsole ... 463
Mit dem AWS CLI ... 463
Verwenden AWS SAM ... 464
Befehlssätze (ARM/x86) ... 465
Vorteile der Verwendung von arm64-Architektur ... 465
Anforderungen für die Migration zur arm64-Architektur ... 466
Funktionscode-Kompatibilität mit arm64-Architektur ... 466
Migration zur arm64-Architektur ... 467
Konfigurieren der Befehlssatz-Architektur ... 468
Zeitüberschreitung ... 469
Wann sollte das Timeout erhöht werden ... 469
Verwenden der Konsole ... 470
Mit dem AWS CLI ... 470
Verwenden AWS SAM ... 471
Konfigurieren Sie langlebige Funktionen ... 471
Ermöglichen Sie eine dauerhafte Ausführung ... 471
IAM-Berechtigungen für dauerhafte Funktionen ... 472
Bewährte Methoden für die Konfiguration ... 474
Umgebungsvariablen ... 476

Umgebungsvariablen erstellen ... 476
Beispielszenario für Umgebungsvariablen ... 480
Abrufen von Umgebungsvariablen ... 482
Definierte Laufzeitumgebungsvariablen ... 483
Sichern von Umgebungsvariablen ... 486
Funktionen an eine VPC anfügen ... 490
Erforderliche IAM-Berechtigungen ... 491
Hinzufügen von Lambda-Funktionen zu einer Amazon VPC in Ihrem AWS-Konto ... 492
Internetzugang bei Verbindung mit einer VPC ... 496
IPv6 Unterstützung ... 496
Bewährte Methoden für die Verwendung von Lambda mit Amazon VPCs ... 497
Grundlegendes zu elastischen Hyperplane-Netzwerkschnittstellen () ENIs ... 499
Verwenden von IAM-Bedingungsschlüsseln für VPC-Einstellungen ... 500
VPC-Tutorials ... 505
Funktionen an Ressourcen in einem anderen Konto anhängen ... 507
Voraussetzungen ... 507
Erstellen Sie eine Amazon VPC im Konto Ihrer Funktion ... 508
Gewährung von VPC-Berechtigungen für die Ausführungsrolle Ihrer Funktion ... 509
Erstellen Sie eine VPC-Peering-Verbindungsanfrage ... 509
Bereiten Sie das Konto Ihrer Ressource vor ... 510
Aktualisieren Sie die VPC-Konfiguration im Konto Ihrer Funktion ... 512
Testen der -Funktion ... 513
Internetzugriff für VPC-Funktionen ... 514
Eingehende Netzwerke ... 540
Überlegungen für Lambda-Schnittstellenendpunkte ... 540
Erstellen eines Schnittstellenendpunkts für Lambda ... 541
Erstellen einer Richtlinie des Schnittstellenendpunkts für Lambda ... 543
Dateisystem ... 545
Unterstützte Regionen ... 545
Ausführungsrolle und Benutzerberechtigungen ... 545
Konfigurieren eines Dateisystems und eines Zugriffspunkts ... 546
Herstellen einer Verbindung mit einem Dateisystem (Konsole) ... 547
Aliasnamen ... 549
Verwenden von Aliassen ... 551
Gewichtete Aliase ... 552
Versionen ... 558

Erstellen von Funktionsversionen ... 559
Verwenden von Versionen ... 560
Gewähren von Berechtigungen ... 561
Tags (Markierungen) ... 563
Erforderliche Berechtigungen zum Arbeiten mit Tags ... 563
Verwendung von Tags mit der Konsole ... 564
Verwenden von Tags mit AWS CLI ... 565
Antwort-Streaming ... 568
Bandbreitenbegrenzung für Antwort-Streaming ... 569
VPC-Kompatibilität mit Antwort-Streaming ... 569
Schreiben von Funktionen ... 570
Aufrufen von -Funktionen ... 572
Tutorial: Erstellen einer Funktion zum Streamen von Antworten mit einer Funktions-URL ... 574
Aufrufen von -Funktionen ... 579
Eine Funktion synchron aufrufen ... 581
Asynchroner Aufruf ... 585
Fehlerbehandlung ... 586
Konfiguration ... 587
Beibehalten von Aufzeichnungen ... 589
Aufruf ... 601
Grenzwerte für synchrone Aufrufe ... 601
Asynchroner Aufruf für Workflows mit langer Laufzeit ... 601
Verwaltung der Ausführung APIs ... 602
Zuweisung von Ereignisquellen ... 604
Zuordnungen von Ereignisquellen und Auslösern ... 605
Batching-Verhalten ... 605
Modus bereitgestellter Kapazität ... 609
API für die Ereignisquellenzuordnung ... 611
Tags für die Zuordnung von Ereignisquellen ... 612
Ereignisfilterung ... 617
Grundlagen der Ereignisfilterung verstehen ... 618
Umgang mit Datensätzen, die die Filterkriterien nicht erfüllen ... 620
Filterregelsyntax ... 621
Anhängen von Filterkriterien an eine Ereignisquellenzuordnung (Konsole) ... 623
Anhängen von Filterkriterien an eine Ereignisquellenzuordnung (AWS CLI) ... 624
Anhängen von Filterkriterien an eine Ereignisquellenzuordnung (AWS SAM) ... 625

Verschlüsselung der Filterkriterien ... 626
Verwenden von Filtern mit unterschiedlichen AWS-Services ... 632
Testen in der Konsole ... 634
Aufrufen von Funktionen mit Testereignissen ... 634
Private Testereignisse erstellen ... 635
Freigabefähige Testereignisse erstellen ... 635
Löschen von freigabefähigen Test-Ereignisschemas ... 637
Funktionszustände ... 638
Funktionsstatus während der Aktualisierung ... 639
Wiederholversuche ... 641
Erkennung rekursiver Schleifen ... 643
Grundlegendes zur Erkennung rekursiver Schleifen ... 643
Unterstützt AWS-Services und SDKs ... 645
Benachrichtigungen zu rekursiven Schleifen ... 648
Reagieren auf Benachrichtigungen im Zusammenhang mit der Erkennung rekursiver Schleifen ... 649
Zulassen, dass eine Lambda-Funktion in einer rekursiven Schleife ausgeführt wird ... 650
Unterstützte Regionen für die Erkennung rekursiver Lambda-Schleifen ... 652
Funktion URLs ... 653
Erstellen einer Funktions-URL (Konsole) ... 654
Erstellen einer Funktions-URL (AWS CLI) ... 656
Hinzufügen einer Funktions-URL zu einer CloudFormation Vorlage ... 657
Cross-Origin Resource Sharing (CORS) ... 659
Drosselungsfunktion URLs ... 660
Funktion wird deaktiviert URLs ... 660
Funktion löschen URLs ... 661
Zugriffskontrolle ... 661
Aufrufen von Funktions-URLs ... 673
Überwachen von Funktions-URLs ... 686
Funktion im URLs Vergleich zu Amazon API Gateway ... 687
Tutorial: Erstellen eines Webhook-Endpunkts ... 694
Funktionsskalierung ... 708
Verstehen und Visualisieren der Gleichzeitigkeit ... 708
Berechnen der Gleichzeitigkeit einer Funktion ... 713
Verständnis der reservierten Gleichzeitigkeit und der bereitgestellten Gleichzeitigkeit ... 715
Reservierte Gleichzeitigkeit ... 715

Provisioned Concurrency ... 718
So weist Lambda bereitgestellte Gleichzeitigkeit zu ... 723
Vergleich zwischen reservierter und bereitgestellter Gleichzeitigkeit ... 724
Gleichzeitigkeit und Anforderungen pro Sekunde verstehen ... 725
Gleichzeitigkeitskontingente ... 727
Konfigurieren reservierter Gleichzeitigkeit ... 730
Konfigurieren reservierter Gleichzeitigkeit ... 731
Genaue Schätzung reservierter Gleichzeitigkeit für eine Funktion ... 733
Konfigurieren von Provisioned Concurrency ... 735
Konfigurieren von Provisioned Concurrency ... 736
Genaue Schätzung der erforderlichen bereitgestellten Gleichzeitigkeit für eine Funktion ... 739
Optimierung des Funktionscodes bei Verwendung der bereitgestellten Gleichzeitigkeit ... 740
Verwendung von Umgebungsvariablen zur Anzeige und Steuerung des bereitgestellten Gleichzeitigkeitsverhaltens ... 741
Verstehen des Protokollierungs- und Abrechnungsverhaltens bei bereitgestellter Gleichzeitigkeit ... 741
Auto Scaling von Anwendungen zur Automatisierung der Verwaltung von Gleichzeitigkeit bei der Bereitstellung ... 742
Skalierungsverhalten ... 747
Nebenläufigkeitsskalierungsrate ... 747
Überwachen der Gleichzeitigkeit ... 749
Allgemeine Metriken für Gleichzeitigkeit ... 749
Metriken für Provisioned Concurrency ... 749
Die Metrik `ClaimedAccountConcurrency` ... 752
Erstellen mit Node.js ... 756
SDK-Versionen, die zur Laufzeit enthalten sind ... 758
Verwenden von Keepalive ... 758
CA-Zertifikat wird geladen ... 758
Experimentelle Funktionen von Node.js ... 759
Handler ... 761
Einrichten Ihres Projekts ... 761
Beispielfunktion ... 762
CommonJS- und ES-Module ... 764
Initialisierung von Node.js ... 766
Namenskonventionen für Handler ... 766
Eingabeereignisobjekt ... 767

Gültige Handler-Muster ... 767
Verwenden Sie das SDK für JavaScript ... 769
Zugriff auf Umgebungsvariablen ... 771
Verwenden des globalen Zustands ... 771
Best Practices ... 771
Bereitstellen von ZIP-Dateiarchiven ... 774
Laufzeitabhängigkeiten in Node.js ... 774
ZIP-Bereitstellungspakets ohne Abhängigkeiten erstellen ... 775
ZIP-Bereitstellungspakets mit Abhängigkeiten erstellen ... 775
Erstellen einer Node.js-Ebene für Ihre Abhängigkeiten ... 777
Suchpfad für Abhängigkeiten und integrierte Laufzeit-Bibliotheken ... 778
Erstellen und Aktualisieren von Node.js-Lambda-Funktionen mithilfe von ZIP-Dateien ... 779
Bereitstellen von Container-Images ... 787
AWS Basis-Images für Node.js ... 788
Verwenden eines AWS Basis-Images ... 789
Verwenden Sie ein Nicht-Basis-Image AWS ... 795
Layer ... 806
Verpacken Ihres Ebeneninhalts ... 806
Erstellen Sie die Ebene in Lambda ... 811
Fügen Sie die Ebene zu Ihrer Funktion hinzu ... 812
Beispiel-App ... 813
Kontext ... 814
Protokollierung ... 816
Erstellen einer Funktion, die Protokolle zurückgibt ... 816
Verwenden von Lambda-Optionen für die erweiterte Protokollierung mit Node.js ... 818
Anzeigen von Protokollen in der Lambda-Konsole ... 825
Protokolle in der Konsole anzeigen CloudWatch ... 826
Logs mit dem Befehl () anzeigen AWS Command Line InterfaceAWS CLI ... 826
Löschen von Protokollen ... 829
Nachverfolgung ... 830
Verwenden von ADOT zur Instrumentierung Ihrer Node.js-Funktionen ... 831
Verwenden des X-Ray-SDK zum Instrumentieren Ihrer Node.js-Funktionen ... 831
Aktivieren der Nachverfolgung mit der Lambda-Konsole ... 832
Aktivieren der Nachverfolgung mit der Lambda-API ... 833
Die Ablaufverfolgung wird aktiviert mit CloudFormation ... 833
Interpretieren einer X-Ray-Nachverfolgung ... 834

Laufzeitabhängigkeiten in einer Ebene speichern (X-Ray-SDK) ... 837
Bauen mit TypeScript ... 839
Entwicklungsumgebung ... 840
Typdefinitionen für Lambda ... 841
Handler ... 843
Einrichten Ihres Projekts ... 843
Beispielfunktion ... 844
CommonJS- und ES-Module ... 847
Initialisierung von Node.js ... 847
Namenskonventionen für Handler ... 848
Eingabeereignisobjekt ... 848
Gültige Handler-Muster ... 849
Verwenden Sie das SDK für JavaScript ... 852
Zugriff auf Umgebungsvariablen ... 853
Verwenden des globalen Zustands ... 853
Best Practices ... 853
Bereitstellen von ZIP-Dateiarchiven ... 856
Benutzen AWS SAM ... 856
Mit dem AWS CDK ... 858
Verwenden von und AWS CLI esbuild ... 861
Bereitstellen von Container-Images ... 864
Erstellen und Paketieren von TypeScript-Funktionscode mithilfe eines Node.js-Basis-Image ... 864
Kontext ... 872
Protokollierung ... 875
Tools und Bibliotheken ... 875
Verwendung von Powertools für AWS Lambda (TypeScript) und für die strukturierte Protokollierung AWS SAM ... 876
Verwenden Sie Powertools für AWS Lambda (TypeScript) und AWS CDK für die strukturierte Protokollierung ... 879
Anzeigen von Protokollen in der Lambda-Konsole ... 883
Logs in der CloudWatch Konsole anzeigen ... 883
Nachverfolgung ... 884
Powertools für AWS Lambda (TypeScript) und für die Nachverfolgung verwenden AWS SAM ... 885
Verwenden von Powertools for AWS Lambda (TypeScript) und AWS CDK for tracing ... 887

Interpretieren einer X-Ray-Nachverfolgung ... 891
Erstellen mit Python ... 892
SDK-Versionen, die zur Laufzeit enthalten sind ... 894
Deaktivierte Python-Funktionen ... 894
Reaktionsformat ... 895
Ordnungsgemäßes Herunterfahren von Erweiterungen ... 895
Handler ... 896
Beispiel für Python-Lambda-Funktionscode ... 896
Namenskonventionen für Handler ... 898
Verwenden des Lambda-Ereignisobjekts ... 899
Zugreifen auf und Verwenden des Lambda-Kontextobjekts ... 900
Gültige Handler-Signaturen für Python-Handler ... 901
Rückgabe eines Wertes ... 901
Verwenden Sie das AWS SDK für Python (Boto3) in Ihrem Handler ... 902
Zugriff auf Umgebungsvariablen ... 903
Bewährte Codemethoden für Python-Lambda-Funktionen ... 904
Bereitstellen von ZIP-Dateiarchiven ... 906
Laufzeitabhängigkeiten in Python ... 906
ZIP-Bereitstellungspakets ohne Abhängigkeiten erstellen ... 907
ZIP-Bereitstellungspakets mit Abhängigkeiten erstellen ... 908
Suchpfad für Abhängigkeiten und integrierte Laufzeit-Bibliotheken ... 911
__pycache__-Ordner verwenden ... 912
ZIP-Bereitstellungspakete mit nativen Bibliotheken erstellen ... 912
Python-Lambda-Funktionen mithilfe von ZIP-Dateien erstellen und aktualisieren ... 914
Bereitstellen von Container-Images ... 922
AWS Basisbilder für Python ... 923
Verwenden eines AWS Basis-Images ... 925
Verwenden Sie ein Image, das nicht zur Basisversion gehört AWS ... 931
Layer ... 941
Verpacken Ihres Ebeneninhalts ... 941
Erstellen Sie die Ebene in Lambda ... 811
Fügen Sie die Ebene zu Ihrer Funktion hinzu ... 947
Beispiel-App ... 948
Kontext ... 949
Protokollierung ... 951
Ausdrucken in das Protokoll ... 951

Verwendung einer Protokollierungsbibliothek ... 952
Verwenden von Lambda-Optionen für die erweiterte Protokollierung mit Python ... 954
Anzeigen von Protokollen in der Lambda-Konsole ... 959
Logs in CloudWatch der Konsole anzeigen ... 959
Logs anzeigen mit AWS CLI ... 960
Löschen von Protokollen ... 963
Tools und Bibliotheken ... 963
Verwendung von Powertools für AWS Lambda (Python) und AWS SAM für strukturiertes Logging ... 964
Verwendung von Powertools für AWS Lambda (Python) und AWS CDK für strukturiertes Logging ... 968
Testen ... 975
Testen Ihrer Serverless-Anwendungen ... 976
Nachverfolgung ... 978
Verwenden von Powertools für AWS Lambda (Python) und AWS SAM für die Nachverfolgung ... 979
Verwenden von Powertools für AWS Lambda (Python) und AWS CDK für die Nachverfolgung ... 982
Verwenden von ADOT zum Instrumentieren Ihrer Python-Funktionen ... 987
Verwenden des X-Ray-SDK zum Instrumentieren Ihrer Python-Funktionen ... 987
Aktivieren der Nachverfolgung mit der Lambda-Konsole ... 988
Aktivieren der Nachverfolgung mit der Lambda-API ... 988
Aktivieren von Nachverfolgungen mit CloudFormation ... 989
Interpretieren einer X-Ray-Nachverfolgung ... 990
Laufzeitabhängigkeiten in einer Ebene speichern (X-Ray-SDK) ... 993
Erstellen mit Ruby ... 994
SDK-Versionen, die zur Laufzeit enthalten sind ... 996
Aktivieren von Yet Another Ruby JIT (YJIT) ... 996
Handler ... 997
Ruby-Handler-Grundlagen ... 997
Bewährte Codemethoden für Ruby-Lambda-Funktionen ... 998
Bereitstellen von ZIP-Dateiarchiven ... 1001
Abhängigkeiten in Ruby ... 1001
ZIP-Bereitstellungspakets ohne Abhängigkeiten erstellen ... 1002
ZIP-Bereitstellungspakets mit Abhängigkeiten erstellen ... 1002
Erstellen einer Ruby-Ebene für Ihre Abhängigkeiten ... 1004

Erstellen von ZIP-Bereitstellungspaketen mit nativen Bibliotheken 1004
Erstellen und Aktualisieren von Ruby-Lambda-Funktionen mithilfe von ZIP-Dateien 1007
Bereitstellen von Container-Images 1014
AWS Basis-Images für Ruby 1015
Verwenden eines AWS Basis-Images 1015
Verwenden Sie ein Nicht-Basis-Image AWS 1022
Ebenen 1032
Verpacken Ihres Ebeneninhalts 1032
Erstellen Sie die Ebene in Lambda 811
Verwenden von Gems aus Ebenen in einer Funktion 1039
Fügen Sie die Ebene zu Ihrer Funktion hinzu 1040
Beispiel-App 1041
Context 1042
Protokollierung 1043
Erstellen einer Funktion, die Protokolle zurückgibt 1043
Anzeigen von Protokollen in der Lambda-Konsole 1045
Protokolle in der Konsole anzeigen CloudWatch 1045
Logs mit dem Befehl () anzeigen AWS Command Line InterfaceAWS CLI 1045
Löschen von Protokollen 1049
Arbeiten mit der Ruby-Logger-Bibliothek 1049
Nachverfolgung 1051
Aktivieren der aktiven Ablaufverfolgung mit der Lambda-API 1057
Aktivieren von aktiver Ablaufverfolgung mit CloudFormation 1057
Speichern von Laufzeitabhängigkeiten in einer Ebene 1058
Erstellen mit Java 1060
Handler 1064
Einrichten Ihres Java-Handler-Projekts 1064
Beispiel für Java-Lambda-Funktionscode 1065
Gültige Klassendefinitionen für Java-Handler 1070
Namenskonventionen für Handler 1072
Definieren Sie das Eingabeereignisobjekt und greifen Sie darauf zu 1072
Zugreifen auf und Verwenden des Lambda-Kontextobjekts 1073
Verwenden des AWS SDK für Java v2 in Ihrem Handler 1074
Zugriff auf Umgebungsvariablen 1076
Verwenden des globalen Zustands 1076
Bewährte Codemethoden für Java-Lambda-Funktionen 1076

Bereitstellen von ZIP-Dateiarchiven 1079
Voraussetzungen 1080
Tools und Bibliotheken 1080
Erstellen eines Bereitstellungspakets mit Gradle 1081
Verwenden von Ebenen für Abhängigkeiten 1082
Erstellen eines Bereitstellungspakets mit Maven 1083
Hochladen eines Bereitstellungspakets mit der Lambda-Konsole 1085
Hochladen eines Bereitstellungspakets mit dem AWS CLI 1087
Hochladen eines Bereitstellungspakets mit AWS SAM 1088
Bereitstellen von Container-Images 1091
AWS Basis-Images für Java 1092
Verwenden eines AWS Basis-Images 1093
Verwenden Sie ein Nicht-Basis-Image AWS 1102
Layer 1114
Verpacken Ihres Ebeneninhalts 1114
Erstellen Sie die Ebene in Lambda 811
Fügen Sie die Ebene zu Ihrer Funktion hinzu 1118
Benutzerdefinierte Serialisierung 1119
Wann sollte die benutzerdefinierte Serialisierung verwendet werden 1119
Benutzerdefinierte Serialisierung implementieren 1120
Testen der benutzerdefinierten Serialisierung 1121
Benutzerdefiniertes Startverhalten 1122
Die `JAVA_TOOL_OPTIONS`-Umgebungsvariable verstehen 1122
Log4j-Patch für Log4Shell 1125
Ahead-of-Time (AOT) und CDS-Caches 1125
Context 1127
Kontext in Beispielanwendungen 1129
Protokollierung 1131
Erstellen einer Funktion, die Protokolle zurückgibt 1131
Verwenden von Lambda-Optionen für die erweiterte Protokollierung mit Java 1133
Implementierung der erweiterten Protokollierung mit Log4J2 und J SLF4 1137
Tools und Bibliotheken 1140
Verwendung von Powertools für AWS Lambda (Java) und für strukturiertes Logging AWS SAM 1141
Anzeigen von Protokollen in der Lambda-Konsole 1145
Protokolle in der CloudWatch Konsole anzeigen 1146

Logs mit dem Befehl () anzeigen AWS Command Line InterfaceAWS CLI 1146
Löschen von Protokollen 1149
Beispielprotokolliercode 1149
Nachverfolgung 1151
Verwendung von Powertools für AWS Lambda (Java) und für die Nachverfolgung AWS SAM 1152
Verwendung von Powertools für AWS Lambda (Java) und AWS CDK für die Ablaufverfolgung 1154
Verwenden von ADOT zur Instrumentierung Ihrer Java-Funktionen 1166
Instrumentieren Sie mit dem X-Ray-SDK Ihre Java-Funktionen 1167
Aktivieren der Nachverfolgung mit der Lambda-Konsole 1167
Aktivieren der Nachverfolgung mit der Lambda-API 1168
Die Ablaufverfolgung wird aktiviert mit CloudFormation 1168
Interpretieren einer X-Ray-Nachverfolgung 1169
Laufzeitabhängigkeiten in einer Ebene speichern (X-Ray-SDK) 1172
X-Ray-Nachverfolgung in Beispielanwendungen (X-Ray-SDK) 1173
Beispiel-Apps 1175
Erstellen mit Go 1177
Unterstützte Go-Laufzeiten 1177
Tools und Bibliotheken 1178
Handler 1179
Einrichten Ihres Go-Handler-Projekts 1179
Beispiel für Go Lambda-Funktionscode 1180
Namenskonventionen für Handler 1183
Definieren Sie das Eingabeereignisobjekt und greifen Sie darauf zu 1183
Zugreifen auf und Verwenden des Lambda-Kontextobjekts 1184
Gültige Handler-Signaturen für Go-Handler 1185
Verwenden Sie die AWS SDK für Go v2 in Ihrem Handler 1186
Zugriff auf Umgebungsvariablen 1188
Verwenden des globalen Zustands 1188
Bewährte Codemethoden für Go-Lambda-Funktionen 1188
Kontext 1190
Unterstützte Variablen, Methoden und Eigenschaften im Kontextmenüobjekt 1190
Zugreifen auf Aufrufkontextinformationen 1191
Verwendung des Kontexts bei Initialisierungen und Aufrufen von AWS-SDK-Clients 1193
Bereitstellen von ZIP-Dateiarchiven 1194

Erstellen einer ZIP-Datei unter macOS und Linux 1194
Erstellen einer ZIP-Datei unter Windows 1196
Go Lambda-Funktionen mithilfe von ZIP-Dateien erstellen und aktualisieren 1199
Bereitstellen von Container-Images 1206
AWS Basis-Images für die Bereitstellung von Go-Funktionen 1206
Laufzeitschnittstellen-Clients von Go 1207
Verwenden eines AWS reinen Betriebssystem-Basisimages 1207
Verwenden Sie ein Nicht-Basis-Image AWS 1215
Ebenen 1224
Protokollierung 1225
Erstellen einer Funktion, die Protokolle zurückgibt 1225
Anzeigen von Protokollen in der Lambda-Konsole 1227
Protokolle in der Konsole anzeigen CloudWatch 1227
Logs mit dem Befehl () anzeigen AWS Command Line InterfaceAWS CLI 1228
Löschen von Protokollen 1231
Nachverfolgung 1232
Verwenden von ADOT zur Instrumentierung Ihrer Go-Funktionen 1233
Instrumentierung Ihrer Go-Funktionen mithilfe von X-Ray-SDK 1233
Aktivieren der Nachverfolgung mit der Lambda-Konsole 1234
Aktivieren der Nachverfolgung mit der Lambda-API 1234
Aktivieren von Nachverfolgungen mit CloudFormation 1235
Interpretieren einer X-Ray-Nachverfolgung 1235
Erstellen mit C# 1239
Entwicklungsumgebung 1239
Installieren der .NET-Projektvorlagen 1239
Installation und Aktualisierung der CLI-Tools 1240
Handler 1241
Einrichten Ihres C#-Handler-Projekts 1241
Beispiel für den C#-Lambda-Funktionscode 1243
Handler für Klassenbibliotheken 1246
Ausführbare Assembly-Handler 1247
Gültige Handler-Signaturen für C#-Funktionen 1248
Namenskonventionen für Handler 1249
Serialisierung in C#-Lambda-Funktionen 1249
Dateibasierte Funktionen 1252
Zugreifen auf und Verwenden des Lambda-Kontextobjekts 1253

Verwenden Sie die SDK für .NET Version 3 in Ihrem Handler ... 1254
Zugriff auf Umgebungsvariablen ... 1255
Verwenden des globalen Zustands ... 1255
Vereinfachen Sie den Funktionscode mit dem Lambda Annotations Framework ... 1256
Bewährte Codemethoden für C#-Lambda-Funktionen ... 1257
Bereitstellungspaket ... 1260
NET Lambda Global CLI ... 1261
AWS SAM ... 1267
AWS CDK ... 1271
ASP.NET ... 1275
Ebenen ... 1280
Bereitstellen von Container-Images ... 1281
AWS Basis-Images für.NET ... 1282
Verwenden eines AWS Basis-Images ... 1282
Verwenden eines Nicht-Basis-Images AWS ... 1285
Native AOT-Kompilierung ... 1290
Lambda-Laufzeit ... 1290
Voraussetzungen ... 1291
Erste Schritte ... 1292
Serialisierung ... 1295
Trimmen ... 1295
Fehlerbehebung ... 1296
Context ... 1297
Protokollierung ... 1300
Erstellen einer Funktion, die Protokolle zurückgibt ... 1300
Verwenden von Lambda-Optionen für die erweiterte Protokollierung mit .NET ... 1301
Tools und Bibliotheken ... 1309
Verwendung von Powertools für AWS Lambda (.NET) und für strukturiertes Logging AWS SAM ... 1309
Anzeigen von Protokollen in der Lambda-Konsole ... 1313
Protokolle in der CloudWatch Konsole anzeigen ... 1313
Logs mit dem Befehl () anzeigen AWS Command Line InterfaceAWS CLI ... 1313
Löschen von Protokollen ... 1317
Nachverfolgung ... 1318
Verwenden von Powertools für AWS Lambda (.NET) und AWS SAM für die Nachverfolgung ... 1319

Instrumentierung Ihrer .NET-Funktionen mithilfe von X-Ray-SDK ... 1322
Aktivieren der Nachverfolgung mit der Lambda-Konsole ... 1323
Aktivieren der Nachverfolgung mit der Lambda-API ... 1324
Aktivieren von Nachverfolgungen mit CloudFormation ... 1324
Interpretieren einer X-Ray-Nachverfolgung ... 1325
Testen ... 1329
Testen Ihrer Serverless-Anwendungen ... 1330
Bauen mit PowerShell ... 1334
Entwicklungsumgebung ... 1336
Bereitstellungspaket ... 1337
Erstellen einer Lambda-Funktion ... 1337
Handler ... 1340
Zurückgeben von Daten ... 1341
Kontext ... 1342
Protokollierung ... 1343
Erstellen einer Funktion, die Protokolle zurückgibt ... 1343
Anzeigen von Protokollen in der Lambda-Konsole ... 1345
Protokolle in der Konsole anzeigen CloudWatch ... 1345
Logs mit dem Befehl () anzeigen AWS Command Line InterfaceAWS CLI ... 1346
Löschen von Protokollen ... 1349
Erstellen mit Rust ... 1350
Handler ... 1352
Einrichten Ihres Rust-Handler-Projekts ... 1352
Beispiel für Rust-Lambda-Funktionscode ... 1353
Gültige Klassendefinitionen für Rust-Handler ... 1356
Namenskonventionen für Handler ... 1357
Definieren Sie das Eingabeereignisobjekt und greifen Sie darauf zu ... 1358
Zugreifen auf und Verwenden des Lambda-Kontextobjekts ... 1359
Verwenden Sie den AWS SDK für Rust in Ihrem Handler ... 1360
Zugriff auf Umgebungsvariablen ... 1360
Geteilten Zustand verwenden ... 1361
Bewährte Codemethoden für Rust-Lambda-Funktionen ... 1361
Kontext ... 1364
Zugreifen auf Aufrufkontextinformationen ... 1364
HTTP-Ereignisse ... 1366
Bereitstellen von ZIP-Dateiarchiven ... 1369

Voraussetzungen ... 1369
Erstellen der Funktion ... 1369
Bereitstellen der Funktion ... 1370
Aufrufen der Funktion ... 1372
Ebenen ... 1373
Protokollierung ... 1374
Erstellen einer Funktion, die Protokolle schreibt ... 1374
Implementierung der erweiterten Protokollierung mit der Tracing Crate ... 1374
Best Practices ... 1377
Funktionscode ... 1377
Konfiguration der Funktionen ... 1379
Skalierbarkeit von Funktionen ... 1380
Metriken und Alarme ... 1381
Arbeiten mit Streams ... 1382
Bewährte Methoden für die Gewährleistung der Sicherheit ... 1383
Serverless-Funktionen testen ... 1385
Gezielte Geschäftsergebnisse ... 1386
Was muss getestet werden? ... 1386
So wird Serverless getestet ... 1387
Testmethoden ... 1388
Testen in der Cloud ... 1389
Testen mit Mocks ... 1392
Lokales Testen mit AWS SAM CLI ... 1393
Testen mit Emulation ... 1394
Bewährte Methoden ... 1395
Priorisieren Sie Tests in der Cloud ... 1395
Strukturieren Sie Ihren Code zum Testen ... 1395
Entwicklungs-Feedback-Schleifen beschleunigen ... 1396
Fokus auf Integrationstests ... 1396
Erstellen Sie isolierte Testumgebungen ... 1397
Mocks für isolierte Geschäftslogik verwenden ... 1398
Emulatoren sparsam verwenden ... 1399
Herausforderungen beim Testen vor Ort ... 1399
Beispiel: Die Lambda-Funktion erstellt einen S3-Bucket ... 1399
Beispiel: Verwenden Sie eine Lambda-Funktion, um Nachrichten aus einer Amazon-SQS-Warteschlange zu verarbeiten. ... 1400

Häufig gestellte Fragen ... 1401
Nächste Schritte und Ressourcen ... 1402
Lambda SnapStart ... 1404
Anwendungsfälle ... 1405
Unterstützte Funktionen und Einschränkungen ... 1405
Unterstützte Regionen ... 1406
Erwägungen zur Kompatibilität ... 1406
Preisgestaltung ... 1407
Aktiviert SnapStart ... 1409
Aktivierung SnapStart (Konsole) ... 1409
SnapStart Aktiviert (AWS CLI) ... 1410
Aktivierung SnapStart (API) ... 1412
Funktionszustände ... 1413
Aktualisieren eines Snapshots ... 1414
SnapStart Verwenden mit AWS SDKs ... 1414
Verwenden Sie SnapStart mit CloudFormationAWS SAM, und AWS CDK ... 1414
Löschen von Snapshots ... 1414
Handhabung der Eindeutigkeit ... 1416
Vermeiden Sie das Speichern von Zuständen ... 1416
Verwenden von CSPRNGs ... 1418
Scanning-Tool (Java) ... 1421
Laufzeit-Hooks ... 1422
Java ... 1422
Python ... 1426
.NET ... 1428
Überwachen ... 1431
CloudWatch-Protokolle ... 1431
AWS X-Ray ... 1432
Telemetrie-API ... 1433
API-Gateway- und Funktions-URL-Metriken ... 1433
Sicherheitsmodell ... 1434
Bewährte Methoden ... 1435
Leistungsoptimierung ... 1435
Bewährte Praktiken für das Netzwerk ... 1439
Fehlerbehebung ... 1441
SnapStartNotReadyException ... 1441

SnapStartTimeoutException 1441
500 Interner Dienstfehler 1442
401 Nicht autorisiert 1442
UnknownHostException (Java) 1442
Fehler bei der Erstellung von Snapshots 1443
Latenzzeit bei der Snapshot-Erstellung 1443
Isolierung von Mietern 1445
Wann sollte der Mandantenisolationsmodus verwendet werden 1446
Unterstützte Funktionen und Einschränkungen 1446
Unterstützt AWS-Regionen 1446
Überlegungen 1446
Preisgestaltung 1447
Isolationsmodus 1447
Aktivierung der Mandantenisolierung 1449
Konsole 1449
AWS CLI 1449
API 1450
CloudFormation 1450
Aufrufen von Funktionen mit Mandantenisolierung 1452
AWS CLI 1452
API 1452
API Gateway 1453
SDK 1454
Zugriff im Code 1457
Zugriff 1457
Nutzungsmuster 1458
Überwachen 1459
Überwachen 1461
Grundlegendes zur Protokollierung für den isolierten Mandantenmodus 1461
Fehlerbehebung 1463
InvalidParameterValueException 1463
TooManyRequestsException 1463
Integration anderer Services 1464
Erstellen eines Auslösers 1464
Liste der Services 1465
Apache Kafka 1468

MSK ... 1470
Selbstverwaltetes Apache Kafka ... 1521
Skalierung des Ereignis-Pollers ... 1541
Abfrage und Stream-Positionen ... 1549
Verbrauchergruppen-ID ... 1550
Ereignisfilterung ... 1551
Schemaregister mit Ereignisquellen ... 1557
Niedrige Latenz für Apache-Kafka ... 1587
Versuchen Sie es erneut mit den Konfigurationen ... 1589
Fehlgeschlagene Aufrufe beibehalten ... 1595
Kafka-Ziel bei einem Ausfall ... 1603
Fehlerbehebung ... 1608
API Gateway ... 1612
Auswählen eines API-Typs ... 1613
Hinzufügen eines Endpunkts zur Lambda-Funktion ... 1614
Proxy-Integration ... 1615
Ereignisformat ... 1615
Reaktionsformat ... 1615
Berechtigungen ... 1616
Beispielanwendung ... 1619
Der Event-Handler von Powertools für Lambda AWS ... 1619
Tutorial ... 1620
Fehler ... 1638
API Gateway im Vergleich zur Funktion URLs ... 1639
Infrastructure Composer ... 1644
Exportieren einer Lambda-Funktion in Infrastructure Composer ... 1645
Sonstige Ressourcen ... 1647
CloudFormation ... 1648
Amazon DocumentDB ... 1652
Beispiel für Amazon-DocumentDB-Ereignis ... 1653
Voraussetzungen und Berechtigungen ... 1654
Konfigurieren Sie die Netzwerksicherheit ... 1656
Erstellen einer Zuordnung von Ereignisquellen von Amazon DocumentDB (Konsole) ... 1659
Erstellen einer Zuordnung von Ereignisquellen von Amazon DocumentDB (SDK oder CLI) 1661
Startpositionen für Abfragen und Streams ... 1664
Überwachen Ihrer Amazon-DocumentDB-Ereignisquelle ... 1664

Tutorial ... 1665
DynamoDB ... 1693
Abrufen und Stapeln von Streams ... 1694
Startpositionen für Abfragen und Streams ... 1695
Gleichzeitige Leser ... 1695
Beispielereignis ... 1695
Zuordnung erstellen ... 1697
Fehler bei Batchelementen ... 1700
Fehlerbehandlung ... 1718
Zustandsbehaftete Verarbeitung ... 1725
Parameter ... 1730
Ereignisfilterung ... 1733
Tutorial ... 1742
EC2 ... 1759
Erteilen von Berechtigungen für EventBridge (CloudWatch Events) ... 1760
Elastic Load Balancing (Application Load Balancer) ... 1761
Event-Handler von Powertools für Lambda AWS ... 1763
Mit einem EventBridge-Scheduler aufrufen ... 1765
Einrichten der Ausführungsrolle ... 1765
Erstellen eines Zeitplans ... 1765
Zugehörige Ressourcen ... 1770
IoT ... 1771
Kinesis Data Streams ... 1773
Abfragen und Stapeln von Streams ... 1773
Beispielereignis ... 1776
Zuordnung erstellen ... 1777
Fehler bei Batchelementen ... 1782
Fehlerbehandlung ... 1801
Zustandsbehaftete Verarbeitung ... 1808
Parameter ... 1812
Ereignisfilterung ... 1815
Tutorial ... 1819
Kubernetes ... 1837
AWS-Controller für Kubernetes (ACK) ... 1837
Crossplane ... 1838
MQ ... 1840

Verstehen der Lambda-Verbrauchergruppe für Amazon MQ 1842
Konfigurieren der Ereignisquelle 1846
Parameter 1853
Ereignisfilterung 1854
Fehlerbehebung 1861
RDS 1862
Konfiguration Ihrer Funktion für die Verwendung mit RDS-Ressourcen 1862
Herstellen einer Verbindung mit einer Amazon-RDS-Datenbank in einer Lambda-Funktion 1869
Verarbeitung von Ereignisbenachrichtigungen von Amazon RDS 1888
Vollständiges Lambda- und Amazon-RDS-Tutorial 1889
Amazon RDS im Vergleich zu DynamoDB 1889
S3 1894
Tutorial: Verwenden eines S3-Auslösers 1896
Tutorial: Verwenden eines Amazon-S3-Auslösers zum Erstellen von Miniaturbildern 1922
Secrets Manager 1951
Auswahl des Ansatzes 1951
Wann Sie den Secrets Manager verwenden sollten 1952
Verwenden der Lambda-Erweiterung AWS Parameter und Geheimnisse 1952
Verwenden Sie das Parameter-Hilfsprogramm von Powertools für AWS Lambda 1964
SQS 1970
Verständnis des Polling- und Batching-Verhaltens für Amazon-SQS-Zuordnungen von Ereignisquellen 1971
Verwenden des Bereitstellungsmodus mit Amazon SQS SQS-Ereignisquellenzuordnungen 1972
Konfiguration des Bereitstellungsmodus für die Zuordnung von Amazon SQS SQS-Ereignisquellen 1974
Beispiel für ein Standard-Warteschlangen-Nachrichtenereignis 1975
Beispiel für ein FIFO-Warteschlangen-Nachrichtenereignis 1977
Zuordnung erstellen 1978
Skalierungsverhalten 1983
Fehlerbehandlung 1986
Parameters 2002
Ereignisfilterung 2003
Tutorial 2008
SQS-Kontoübergreifendes Tutorial 2026
Step Functions 2034

Wann sollte Step Functions verwendet werden? ... 2034
Wann sollte Step Functions nicht verwendet werden? ... 2041
S3-Stapel ... 2043
Aufrufen von Lambda-Funktionen aus Amazon-S3-Batchvorägngen ... 2044
SNS ... 2046
Idempotenz-Dienstprogramm von Powertools for AWS Lambda ... 2046
Hinzufügen eines Amazon-SNS-Themenauslösers für eine Lambda-Funktion mithilfe der Konsole ... 2047
Manuelles Hinzufügen eines Amazon-SNS-Themen-Auslösers für eine Lambda-Funktion . 2047
Beispiel für eine SNS-Eventform ... 2048
Tutorial ... 2049
Lambda-Berechtigungen ... 2070
Ausführungsrolle (Berechtigungen für Funktionen zum Zugriff auf andere Ressourcen) ... 2072
Erstellen einer Ausführungsrolle in der IAM-Konsole ... 2072
Rollen erstellen und verwalten mit dem AWS CLI ... 2073
Gewähren Sie den Zugriff auf Ihre Lambda-Ausführungsrolle mit den geringsten Berechtigungen ... 2075
Aktualisieren einer Ausführungsrolle ... 2075
AWS verwaltete Richtlinien ... 2077
ARN der Quellfunktion ... 2081
Zugriffsberechtigungen (Berechtigungen für andere Entitäten, auf Ihre Funktionen zuzugreifen) ... 2086
Identitätsbasierte Richtlinien ... 2086
Ressourcenbasierte Richtlinien ... 2093
Attributbasierte Zugriffskontrolle ... 2102
Ressourcen und Bedingungen ... 2110
Sicherheit, Governance und Compliance ... 2118
Datenschutz ... 2119
Verschlüsselung während der Übertragung ... 2120
Verschlüsselung im Ruhezustand ... 2120
Verwenden von servicegebundenen Rollen ... 2126
Dienstbezogene Rollenberechtigungen für Lambda ... 2126
Eine serviceverknüpfte Rolle für Lambda erstellen ... 2127
Bearbeiten einer serviceverknüpften Rolle für Lambda ... 2127
Löschen einer serviceverknüpften Rolle für Lambda ... 2127
Unterstützte Regionen für dienstverknüpfte Lambda-Rollen ... 2128

Identitäts- und Zugriffsverwaltung ... 2130
Zielgruppe ... 2130
Authentifizierung mit Identitäten ... 2130
Verwalten des Zugriffs mit Richtlinien ... 2132
Funktionsweise von AWS Lambda mit IAM ... 2134
Beispiele für identitätsbasierte Richtlinien ... 2140
AWS verwaltete Richtlinien ... 2143
Fehlersuche ... 2153
Governance ... 2155
Proaktive Kontrollen mit Guard ... 2158
Proaktive Kontrollen mit AWS Config ... 2162
Detektive Kontrollen mit AWS Config ... 2170
Codesignatur ... 2175
Scannen von Code ... 2178
Beobachtbarkeit ... 2183
Compliance-Validierung ... 2192
Ausfallsicherheit ... 2192
Sicherheit der Infrastruktur ... 2193
Sicherung von Workloads mit öffentlichen Endpunkten ... 2194
Authentifizierung und Autorisierung ... 2194
API-Endpunkte schützen ... 2195
Codesignatur ... 2196
Signaturvalidierung ... 2197
Konfiguration erstellen ... 2198
Berechtigungen ... 2200
Code-Signaturkonfigurations-Tags ... 2201
Überwachungs- und Debugging-Funktionen ... 2206
Preisgestaltung ... 2206
Funktionsmetriken ... 2207
Anzeigen von Funktions-Metriken ... 2207
Metriktypen ... 2208
Funktionsprotokolle ... 2217
Auswahl eines Serviceziels für das Senden von Protokollen ... 2217
Konfigurieren von Protokollzielen ... 2218
Konfigurieren erweiterter Protokollierungsoptionen für Lambda-Funktionen ... 2219
Protokollformate ... 2219

Filterung auf Protokollebene ... 2226
Protokollieren mit CloudWatch Logs ... 2232
Protokollieren mit Firehose ... 2252
Protokoll mit Amazon S3 ... 2254
CloudTrail-Protokolle ... 2260
Lambda-Datenereignisse in CloudTrail ... 2261
Lambda-Verwaltungsereignisse in CloudTrail ... 2263
Verwenden von CloudTrail zur Problembehandlung deaktivierter Lambda-Ereignisquellen 2265
Beispiele für Lambda-Ereignisse ... 2266
AWS X-Ray ... 2269
Grundlegendes zu X-Ray-Ablaufverfolgungen ... 2270
Standardmäßiges Nachverfolgungsverhalten in Lambda ... 2275
Berechtigungen für die Ausführungsrolle ... 2276
Aktivieren der `Active`-Nachverfolgung mit der Lambda-API ... 2276
Aktivieren der Ablaufverfolgung mit `Active`CloudFormation ... 2277
Einblicke zu Funktionen ... 2278
Funktionsweise ... 2278
Preisgestaltung ... 2279
Unterstützte Laufzeiten ... 2279
Lambda Insights in der Konsole aktivieren ... 2279
Programmgesteuertes Aktivieren von Lambda Insights ... 2280
Verwenden des Lambda-Insights-Dashboards ... 2280
Erkennen von Funktionsanomalien ... 2282
Fehlerbehebung bei einer Funktion ... 2284
Als nächstes ... 2286
Anzeigen von Anwendungs-Metriken ... 2287
Application Signals ... 2290
So lässt sich Application Signals in Lambda integrieren ... 2290
Preisgestaltung ... 2291
Unterstützte Laufzeiten ... 2291
Aktivieren von Application Signals in der Lambda-Konsole ... 2292
Verwenden des Application Signals-Dashboards ... 2292
Debuggen mit VS Code ... 2294
Unterstützte Laufzeiten ... 2294
Sicherheit und Remote-Debugging ... 2294
Voraussetzungen ... 2295

Remote-Debugging von Lambda-Funktionen ... 2296
Deaktivieren von Remote-Debugging ... 2296
Zusätzliche Informationen ... 2297
Lambda-Ebenen ... 2298
Verwenden von Ebenen ... 2300
Ebenen und Ebenenversionen ... 2300
Verpacken von Ebenen ... 2302
Ebenenpfade für jede Lambda-Laufzeit ... 2302
Erstellen und Löschen von Ebenen ... 2306
Erstellen einer Ebene ... 2306
Löschen einer Ebenen-Version ... 2307
Hinzufügen von Ebenen ... 2309
Suche nach Ebeneninformationen ... 2311
Ebenen mit CloudFormation ... 2313
Ebenen mit AWS SAM ... 2314
Lambda-Erweiterungen ... 2315
Ausführungsumgebung ... 2316
Auswirkungen auf Leistung und Ressourcen ... 2317
Berechtigungen ... 2318
Konfigurieren von Erweiterungen ... 2319
Konfigurieren von Erweiterungen (ZIP-Dateiarchiv) ... 2319
Verwenden von Erweiterungen in Container-Images ... 2319
Nächste Schritte ... 2320
Partner für Erweiterungen ... 2321
AWSVon verwaltete Erweiterungen ... 2322
Erweiterungs-API ... 2323
Lebenszyklus der Lambda-Ausführungsumgebung ... 2324
Erweiterungs-API-Referenz ... 2334
Telemetrie-API ... 2341
Erstellen von Erweiterungen mithilfe der Telemetrie-API ... 2343
Registrieren Ihrer Erweiterung ... 2344
Erstellen eines Telemetrie-Listeners ... 2345
Festlegen eines Zielprotokolls ... 2346
Konfiguration der Speichernutzung und Pufferung ... 2347
Senden einer Abonnementanfrage an die Telemetrie-API ... 2349
Eingehende Telemetrie-API-Nachrichten ... 2350

API-Referenz ... 2354
Referenz zum Event-Schema ... 2358
Konvertieren von Ereignissen in OTel-Spans ... 2380
Logs API ... 2387
Fehlerbehebung ... 2401
Konfiguration ... 2401
Arbeitsspeicherkonfigurationen ... 2402
CPU-gebundene Konfigurationen ... 2402
Zeitüberschreitungen ... 2402
Speicherlecks zwischen Aufrufen ... 2403
Asynchrone Ergebnisse wurden bei einem späteren Aufruf zurückgegeben ... 2406
Bereitstellung ... 2411
Allgemein: Berechtigung wird verweigert/Kann solche Datei nicht laden ... 2412
Allgemein: Beim Aufrufen von UpdateFunctionCode ... 2412
Amazon S3: Fehlercode PermanentRedirect. ... 2413
Allgemein: Kann nicht gefunden werden, kann nicht geladen werden, Klasse nicht gefunden, keine solche Datei oder kein Verzeichnis ... 2413
Allgemein: Undefinierter Methodenhandler ... 2414
Allgemein: Speicherlimit für Lambda-Code überschritten ... 2414
Lambda: Ebenenkonvertierung ist fehlgeschlagen. ... 2415
Lambda: oder InvalidParameterValueException RequestEntityTooLargeException ... 2416
Lambda: InvalidParameterValueException ... 2416
Lambda: Kontingente für Gleichzeitigkeit und Speicher ... 2417
Lambda: Ungültige Aliaskonfiguration für bereitgestellte Gleichzeitigkeit ... 2417
Aufruf ... 2418
Lambda: Funktion bricht während der Init-Phase ab (Sandbox.Timedout) ... 2419
IAM: Lambda: InvokeFunction nicht autorisiert ... 2420
Lambda: Es konnte kein gültiger Bootstrap gefunden werden (Runtime). InvalidEntrypoint) 2421
Lambda: Vorgang kann nicht ausgeführt werden ResourceConflictException ... 2421
Lambda: Funktion bleibt im Status Pending ... 2421
Lambda: Eine Funktion verwendet alle Parallelität ... 2422
Allgemein: Funktion kann nicht mit anderen Konten oder Diensten aufgerufen werden ... 2422
Allgemein: Funktionsaufruf bleibt im Looping ... 2422
Lambda: Alias-Routing mit bereitgestellter Parallelität ... 2422
Lambda: Kaltstarts mit bereitgestellter Parallelität ... 2423
Lambda: Kaltstarts mit neuen Versionen ... 2424

Lambda: Unerwartetes Beenden von Node.js während der Laufzeit (Runtime). NodejsExit) 2424
EFS: Die Funktion konnte das EFS-Dateisystem nicht mounten 2425
EFS: Die Funktion konnte keine Verbindung zum EFS-Dateisystem herstellen 2425
EFS: Die Funktion konnte das EFS-Dateisystem aufgrund eines Timeouts nicht mounten . 2425
Lambda: Lambda hat einen IO-Prozess erkannt, der zu lange dauerte 2426
Container: Fehler CodeArtifactUserException 2426
Container: InvalidEntrypoint Fehler 2426
Ausführung 2427
Lambda: Remote-Debugging mit Visual Studio Code 2428
Lambda: Die Ausführung dauert zu lange 2428
Lamda: Unerwartete Ereignis-Nutzdaten 2428
Lambda: Unerwartet große Nutzdaten 2429
Lambda: Fehler beim Kodieren und Dekodieren von JSON 2430
Lambda: Protokolle oder Ablaufverfolgungen erscheinen nicht 2431
Lambda: Nicht alle Protokolle meiner Funktion werden angezeigt 2431
Lambda: Die Funktion kehrt zurück, bevor die Ausführung beendet ist 2432
Lamda: Ausführen einer unbeabsichtigten Funktionsversion oder eines Alias 2433
Lambda: Erkennen von Endlosschleifen 2433
Allgemein: Nichtverfügbarkeit nachgelagerter Dienste 2434
AWS SDK: Versionen und Updates 2435
Python: Bibliotheken werden falsch geladen 2436
Java: Ihre Funktion braucht länger, um Ereignisse zu verarbeiten, nachdem sie von Java 11 auf Java 17 aktualisiert wurde 2437
Kafka: Probleme bei der Fehlerbehandlung und beim erneuten Versuch der Konfiguration 2437
Ereignisquellen-Zuweisung 2438
Drosselung erkennen und verwalten 2438
Fehler in der Verarbeitungsfunktion 2440
Identifizierung und Behandlung von Gegendruck 2442
Netzwerk 2443
VPC: Funktion verliert Internetzugriff oder läuft ab 2443
VPC: TCP- oder UDP-Verbindung bricht zeitweise ab 2444
VPC: Die Funktion benötigt Zugriff, AWS-Services ohne das Internet zu nutzen 2444
VPC: Grenzwert für Elastic-Network-Schnittstellen erreicht 2445
EC2: Elastische Netzwerkschnittstelle mit dem Typ „Lambda“ 2445
DNS: Verbindung zu Hosts mit UNKNOWNHOSTEXCEPTION schlägt fehl 2445

Beispielanwendungen 2447
Arbeitet mit AWS SDKs 2450
Codebeispiele 2452
Grundlagen 2454
Hallo Lambda 2456
Kennenlernen der Grundlagen 2465
Aktionen 2602
Szenarien 2738
Bestätigen Sie automatisch bekannte Benutzer mit einer Lambda-Funktion 2739
Migrieren Sie automatisch bekannte Benutzer mit einer Lambda-Funktion 2780
Erstellen einer REST-API zur Verfolgung von COVID-19-Daten 2804
Leihbibliothek-REST-API erstellen 2805
Erstellen einer Messenger-Anwendung 2806
Erstellen einer Serverless-Anwendung zur Verwaltung von Fotos 2807
Erstellen einer Websocket-Chat-Anwendung 2811
Erstellen einer Anwendung zum Analysieren von Kundenfeedback 2812
Aufrufen einer Lambda-Funktion von einem Browser aus 2818
Transformieren von Daten mit S3 Object Lambda 2819
Verwenden von API Gateway zum Aufrufen einer Lambda-Funktion 2820
Verwenden von Step Functions, um Lambda-Funktionen aufzurufen 2822
Verwendung geplanter Ereignisse zum Aufrufen einer Lambda-Funktion 2823
Verwenden der Neptune-API, um Grafikdaten abzufragen 2825
Schreiben von benutzerdefinierten Aktivitätsdaten mit einer Lambda-Funktion nach der Amazon Cognito-Benutzerauthentifizierung 2825
Serverless-Beispiele 2848
Herstellen einer Verbindung mit einer Amazon-RDS-Datenbank in einer Lambda-Funktion 2849
Aufrufen einer Lambda-Funktion über einen Kinesis-Auslöser 2868
Aufrufen einer Lambda-Funktion über einen DynamoDB-Auslöser 2879
Aufrufen einer Lambda-Funktion über einen Amazon DocumentDB-Auslöser 2888
Aufrufen einer Lambda-Funktion über einen Amazon-MSK-Auslöser 2900
Aufrufen einer Lambda-Funktion über einen Amazon-S3-Auslöser 2910
Eine Lambda-Funktion über einen Amazon-SNS-Trigger aufrufen 2922
Aufrufen einer Lambda-Funktion über einen Amazon-SQS-Auslöser 2931
Melden von Batch-Elementfehlern für Lambda-Funktionen mit einem Kinesis-Auslöser 2940
Melden von Batch-Elementfehlern für Lambda-Funktionen mit einem DynamoDB-Auslöser 2954

Melden von Batch-Elementfehlern für Lambda-Funktionen mit einem Amazon-SQS-Auslöser 2965
AWS Beiträge der Gemeinschaft 2975
Erstellen und Testen einer Serverless-Anwendung 2975
Lambda-Kontingente 2978
Datenverarbeitung und Speicherung 2980
Funktionskonfiguration, -bereitstellung und -ausführung 2981
Lambda-API-Anforderungen 2984
Sonstige -Services 2985
Dokumentverlauf 2987
Frühere Updates 3019
..... mmmxxviii

Was ist AWS Lambda?

AWS Lambda ist ein Rechendienst, der Code ausführt, ohne dass Server verwaltet werden müssen. Ihr Code wird ausgeführt und automatisch nach oben und unten skaliert, mit pay-per-use Preisgestaltung. Um zu beginnen, sehen Sie sich Erstellen Sie Ihre erste Funktion an.

Sie können Lambda für Folgendes verwenden:

- Dateiverarbeitung: Dateien werden automatisch verarbeitet, wenn sie auf Amazon Simple Storage Service hochgeladen werden. Einzelheiten finden Sie in den Beispielen zur Dateiverarbeitung.
- Workflows mit langer Laufzeit: Verwenden Sie langlebige Lambda-Funktionen, um statusbehaftete, mehrstufige Workflows zu erstellen, die bis zu einem Jahr lang ausgeführt werden können. Perfekt für Auftragsabwicklung, Genehmigungsworkflows, human-in-the-loop Prozesse und komplexe Daten-Pipelines, die ihren Fortschritt speichern müssen.
- Beispiele für Datenbankoperationen und -integration: Reagieren Sie auf Datenbankänderungen und automatisieren Sie Datenworkflows. Einzelheiten finden Sie in den Datenbankbeispielen.
- Geplante und regelmäßige Aufgaben: Führen Sie automatisierte Vorgänge nach einem regelmäßigen Zeitplan aus mit EventBridge. Einzelheiten finden Sie in den Beispielen für geplante Aufgaben.
- Stream-Verarbeitung: Verarbeiten Sie Echtzeit-Datenströme für Analytik und Überwachung. Einzelheiten finden Sie unter Kinesis Data Streams.
- Webanwendungen: Erstellen Sie skalierbare Webanwendungen, die sich automatisch an die Nachfrage anpassen.
- Mobile Backends: Erstellen Sie sichere API-Backends für Mobil- und Webanwendungen.
- IoT-Backends: Verarbeiten Sie Web-, Mobil-, IoT- und Drittanbieter-API-Anfragen. Weitere Details finden Sie unter IoT.

Preisinformationen finden Sie unter AWS Lambda – Preise.

Funktionsweise von Lambda

Wenn Sie Lambda verwenden, sind Sie nur für Ihren Code verantwortlich. Lambda führt Ihren Code auf einer hochverfügbaren Datenverarbeitungsinfrastruktur aus und verwaltet alle Datenverarbeitungsressourcen, einschließlich Server- und Betriebssystemwartung, Kapazitätsbereitstellung, automatische Skalierung und Protokollierung.

Da Lambda ein ereignisgesteuerter Serverless-Datenverarbeitungsdienst ist, verwendet er ein anderes Programmierparadigma als herkömmliche Webanwendungen. Das folgende Modell veranschaulicht die Funktionsweise von Lambda:

1. Sie schreiben und organisieren Ihren Code in Lambda-Funktionen, die die grundlegenden Bausteine für die Erstellung einer Lambda-Anwendung darstellen.
2. Sie kontrollieren Sicherheit und Zugriff über Lambda-Berechtigungen und verwalten mithilfe von Ausführungsrollen, mit welchen AWS Diensten Ihre Funktionen interagieren können und welche Ressourcenrichtlinien mit Ihrem Code interagieren können.
3. Ereignisquellen und AWS Dienste lösen Ihre Lambda-Funktionen aus, indem sie Ereignisdaten im JSON-Format übergeben, die von Ihren Funktionen verarbeitet werden (dazu gehören auch Zuordnungen von Ereignisquellen).
4. Lambda führt Ihren Code mit sprachspezifischen Laufzeiten (wie Node.js und Python) in Ausführungsumgebungen aus, die Ihre Laufzeit, Ebenen und Erweiterungen bündeln.

> **Tip**
>
> Weitere Informationen zum Erstellen von Serverless-Lösungen finden Sie im Serverless-Benutzerhandbuch.

Schlüssel-Features

Konfigurieren, Steuern und Bereitstellen sicherer Anwendungen:

- Umgebungsvariablen: Modifizieren Sie das Verhalten von Anwendungen, ohne dass neuer Code bereitgestellt werden muss.
- Versionen: Testen Sie neue Funktionen auf sichere Weise und sorgen Sie gleichzeitig für stabile Produktionsumgebungen.
- Lambda-Ebenen: Optimieren Sie die Wiederverwendung und Wartung von Code, indem Sie gemeinsame Komponenten über mehrere Funktionen hinweg gemeinsam nutzen.
- Codesignatur: Sorgen Sie für die Einhaltung der Sicherheitsbestimmungen, indem Sie sicherstellen, dass nur genehmigter Code die Produktionssysteme erreicht.

Zuverlässiges Skalieren Arbeiten:

- Parallelitäts- und Skalierungskontrollen steuern präzise die Reaktionsfähigkeit von Anwendungen und die Ressourcennutzung während Verkehrsspitzen.
- Lambda SnapStart: Reduzieren Sie die Kaltstartzeiten erheblich. Lambda SnapStart kann eine Startleistung von nur unter einer Sekunde bieten, in der Regel ohne Änderungen an Ihrem Funktionscode.
- Antwort-Streaming: Optimieren Sie die Funktionsleistung, indem Sie große Nutzdaten schrittweise für die Echtzeitverarbeitung bereitstellen.
- Container-Images: Verpacken Sie Funktionen mit komplexen Abhängigkeiten mithilfe von Container-Workflows.

Nahtloses Verbinden und Integrieren:

- VPC-Netzwerke sichern sensible Ressourcen und interne Dienste.
- Die Dateisystem-Integration nutzt persistente Daten gemeinsam und verwaltet zustandsbehaftete Vorgänge über Funktionsaufrufe hinweg.
- Funktion URLserstellen Sie öffentlich zugängliche Endgeräte APIs und Endgeräte ohne zusätzliche Dienste.
- Lambda-Erweiterungen: Erweitern Sie die Funktionen durch Überwachungs-, Sicherheits- und Betriebstools.

Ähnliche Informationen

- Weitere Informationen zur Funktionsweise von Lambda finden Sie unter Funktionsweise von Lambda.
- Informationen zum Einstieg in Lambda finden Sie unter Erstellen Sie Ihre erste Lambda-Funktion.
- Eine Liste von Beispielanwendungen finden Sie unter Erste Schritte mit Beispielanwendungen und -mustern.

Funktionsweise von Lambda

Lambda-Funktionen sind die grundlegenden Bausteine, die Sie zum Erstellen von Lambda-Anwendungen verwenden. Beim Schreiben von Funktionen ist es wichtig, die Kernkonzepte und -komponenten zu verstehen, aus denen das Lambda-Programmiermodell besteht. In diesem Abschnitt

werden die grundlegenden Elemente vorgestellt, die Sie benötigen, um mit der Entwicklung von Serverless-Anwendungen mit Lambda zu beginnen.

- Lambda-Funktionen und -Funktionshandler- Eine Lambda-Funktion ist ein kleiner Codeblock, der als Reaktion auf Ereignisse ausgeführt wird. Die Funktionen können standardmäßig (bis zu 15 Minuten) oder dauerhaft (bis zu einem Jahr) ausgeführt werden. Funktionen sind die grundlegenden Bausteine, die Sie zum Erstellen von Anwendungen verwenden. Funktionshandler sind der Einstiegspunkt für Ereignisobjekte, die Ihr Lambda-Funktionscode verarbeitet.
- Lambda-Ausführungsumgebung und -Laufzeiten: Lambda-Ausführungsumgebungen verwalten die Ressourcen, die zum Ausführen Ihrer Funktion erforderlich sind. Für langlebige Funktionen umfasst die Ausführungsumgebung automatische Statusverwaltungs- und Checkpoint-Funktionen. Runtimes sind die sprachspezifischen Umgebungen, in denen Ihre Funktionen ausgeführt werden.
- Ereignisse und Auslöser- Andere AWS-Services können Ihre Funktionen als Reaktion auf bestimmte Ereignisse aufrufen. Bei dauerhaften Funktionen können Ereignisse auch die Wiederaufnahme unterbrochener Workflows auslösen.
- Lambda-Berechtigungen und -Rollen- Steuern Sie, wer auf Ihre Funktionen zugreifen kann und mit welchen anderen Funktionen AWS-Services Sie interagieren können. Dauerhafte Funktionen erfordern zusätzliche Berechtigungen für die Statusverwaltung und die erweiterte Ausführung.

> **Tip**
>
> Wenn Sie die Entwicklung von Serverless-Komponenten zuerst besser verstehen möchten, lesen Sie die Informationen zum Unterschied zwischen herkömmlicher Entwicklung und Serverless-Entwicklung im AWS -Serverless-Entwicklerhandbuch.

Lambda-Funktionen und -Funktionshandler

In Lambda sind Funktionen die grundlegenden Bausteine, mit denen Sie Anwendungen erstellen. Eine Lambda-Funktion ist ein Codeteil, der als Reaktion auf Ereignisse ausgeführt wird, z. B. wenn ein Benutzer auf eine Schaltfläche auf einer Website klickt oder eine Datei in einen Bucket in Amazon Simple Storage Service (Amazon S3) hochgeladen wird. Mit dauerhaften Funktionen kann Ihr Code die Ausführung zwischen den einzelnen Schritten unterbrechen und so den Status automatisch beibehalten. Dadurch eignen sie sich ideal für langwierige Workflows wie Auftragsverarbeitung oder Inhaltsmoderation. Sie können sich eine Funktion als eine Art eigenständiges Programm mit den folgenden Eigenschaften vorstellen.

Ein Lambda-Funktionshandler ist die Methode in Ihrem Funktionscode, die Ereignisse verarbeitet. Wenn eine Funktion als Reaktion auf ein Ereignis ausgeführt wird, führt Lambda den Funktionshandler aus. Daten zum Ereignis, das die Ausführung der Funktion verursacht hat, werden direkt an den Handler übergeben. Während der Code in einer Lambda-Funktion mehrere Methoden oder Funktionen enthalten kann, können Lambda-Funktionen nur einen Handler enthalten.

Um eine Lambda-Funktion zu erstellen, bündeln Sie den Funktionscode und die zugehörigen Abhängigkeiten in einem Bereitstellungspaket. Lambda unterstützt zwei Arten von Bereitstellungspaketen: ZIP-Dateiarchive und Container-Images.

- Eine Funktion hat eine bestimmte Aufgabe oder einen bestimmten Zweck.
- Sie wird nur ausgeführt, wenn sie als Reaktion auf bestimmte Ereignisse benötigt wird.
- Sie wird automatisch beendet, wenn sie fertig ist.

Lambda-Ausführungsumgebung und -Laufzeiten

Lambda-Funktionen werden in einer sicheren, isolierten Ausführungsumgebung ausgeführt, die Lambda für Sie verwaltet. Für langlebige Funktionen umfasst die Ausführungsumgebung zusätzliche Komponenten für die Statusverwaltung und die Workflow-Koordination. Die Ausführungsumgebung verwaltet die Prozesse und Ressourcen, die für die Ausführung Ihrer Funktion erforderlich sind. Wenn eine Funktion zum ersten Mal aufgerufen wird, erstellt Lambda eine neue Ausführungsumgebung, in der die Funktion ausgeführt werden kann. Nachdem die Ausführung der Funktion abgeschlossen wurde, beendet Lambda die Ausführungsumgebung nicht sofort. Wenn die Funktion erneut aufgerufen wird, kann Lambda die vorhandene Ausführungsumgebung wiederverwenden.

Die Lambda-Ausführungsumgebung enthält außerdem eine Laufzeit, eine sprachspezifische Umgebung, die Ereignisinformationen und Antworten zwischen Lambda und Ihrer Funktion weiterleitet. Lambda bietet eine Reihe von verwalteten Laufzeiten für die gängigsten Programmiersprachen, Sie können aber auch eigene erstellen.

Bei verwalteten Laufzeiten wendet Lambda automatisch Sicherheitsupdates und Patches auf Funktionen an, die die Laufzeit verwenden.

Ereignisse und Auslöser

Sie können eine Lambda-Funktion auch direkt aufrufen, AWS CLIindem Sie die Lambda-Konsole oder eines der AWS Software Development Kits () verwenden. SDKs In einer Produktionsanwendung ist es üblicher, dass Ihre Funktion als Reaktion auf ein bestimmtes Ereignis von einer anderen AWS-

Service aufgerufen wird. Sie möchten beispielsweise, dass eine Funktion immer dann ausgeführt wird, wenn ein Element zu einer Amazon-DynamoDB-Tabelle hinzugefügt wird.

Damit Ihre Funktion auf Ereignisse reagiert, richten Sie einen Auslöser ein. Ein Auslöser verbindet Ihre Funktion mit einer Ereignisquelle und Ihre Funktion kann mehrere Auslöser haben. Wenn ein Ereignis eintritt, empfängt Lambda Ereignisdaten als JSON-Dokument und konvertiert es in ein Objekt, das Ihr Code verarbeiten kann. Sie können das folgende JSON-Format für Ihr Ereignis definieren und die Lambda-Laufzeit konvertiert dieses JSON-Dokument in ein Objekt, bevor es an den Handler Ihrer Funktion übergeben wird.

Example Benutzerdefiniertes Lambda-Ereignis

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

Lambda verwendet wie Stream- und Warteschlangenservices wie Amazon Kinesis oder Amazon SQS eine Zuordnung von Ereignisquellen anstelle eines Standardauslösers. Bei Zuordnungen von Ereignisquellen wird die Quelle nach neuen Daten abgefragt, Datensätze gebündelt und anschließend Ihre Funktion mit den gebündelten Ereignissen aufgerufen. Weitere Informationen finden Sie unter Wie unterscheiden sich Zuordnungen von Ereignisquellen von direkten Auslösern.

Um zu verstehen, wie ein Auslöser funktioniert, beginnen Sie mit dem Tutorial Verwenden eines Amazon-S3-Auslösers. Einen allgemeinen Überblick über die Verwendung von Auslösern und Anweisungen zum Erstellen eines Auslösers mithilfe der Lambda-Konsole finden Sie unter Integration anderer Services.

Lambda-Berechtigungen und -Rollen

Bei Lambda gibt es zwei Haupttypen von Berechtigungen, die Sie konfigurieren müssen:

- Berechtigungen, die Ihre Funktion benötigt, um auf andere zuzugreifen AWS-Services
- Berechtigungen, die andere Benutzer für den Zugriff auf Ihre Funktion AWS-Services benötigen

In den folgenden Abschnitten werden diese beiden Berechtigungstypen beschrieben und bewährte Methoden für die Anwendung von geringsten Berechtigungen erörtert.

Berechtigungen für Funktionen, um auf andere AWS Ressourcen zuzugreifen

Lambda-Funktionen müssen häufig auf andere AWS Ressourcen zugreifen und Aktionen mit ihnen ausführen. Eine Funktion könnte beispielsweise Elemente aus einer DynamoDB-Tabelle lesen, ein Objekt in einem S3-Bucket speichern oder in eine Amazon-SQS-Warteschlange schreiben. Um Funktionen die Berechtigungen zu geben, die sie für die Ausführung dieser Aktionen benötigen, verwenden Sie eine Ausführungsrolle.

Eine Lambda-Ausführungsrolle ist eine besondere Rolle AWS Identity and Access Management (IAM), eine Identität, die Sie in Ihrem Konto erstellen und der in einer Richtlinie bestimmte Berechtigungen zugeordnet sind.

Jede Lambda-Funktion muss eine Ausführungsrolle haben und eine einzelne Rolle kann von mehr als einer Funktion verwendet werden. Wenn eine Funktion aufgerufen wird, übernimmt Lambda die Ausführungsrolle der Funktion und erhält die Berechtigung, die in der Richtlinie der Rolle definierten Aktionen auszuführen.

Wenn Sie eine Funktion in der Lambda-Konsole erstellen, erstellt Lambda automatisch eine Ausführungsrolle für Ihre Funktion. Die Rollenrichtlinie gibt Ihrer Funktion grundlegende Berechtigungen zum Schreiben von Protokollausgaben in Amazon CloudWatch Logs. Um Ihrer Funktion die Erlaubnis zu erteilen, Aktionen mit anderen AWS Ressourcen durchzuführen, müssen Sie die Rolle bearbeiten, um die zusätzlichen Berechtigungen hinzuzufügen. Der einfachste Weg, Berechtigungen hinzuzufügen, ist die Verwendung einer AWS verwalteten Richtlinie. Verwaltete Richtlinien werden von vielen gängigen Anwendungsfällen erstellt AWS und verwaltet und bieten Berechtigungen für diese. Wenn Ihre Funktion beispielsweise CRUD-Operationen an einer DynamoDB-Tabelle ausführt, können Sie die AmazonDynamoDBFullAccess-Richtlinie zu Ihrer Rolle hinzufügen.

Berechtigungen für andere Benutzer und Ressourcen für den Zugriff auf Ihre Funktion

Um anderen AWS-Service Zugriffsberechtigungen auf Ihre Lambda-Funktion zu gewähren, verwenden Sie eine ressourcenbasierte Richtlinie. In IAM werden ressourcenbasierte Richtlinien

einer Ressource zugeordnet (in diesem Fall Ihrer Lambda-Funktion). Sie definieren, wer auf die Ressource zugreifen kann und welche Aktionen sie ausführen können.

Damit eine andere AWS-Service Funktion Ihre Funktion über einen Trigger aufrufen kann, muss die ressourcenbasierte Richtlinie Ihrer Funktion diesem Dienst die Erlaubnis erteilen, die Aktion zu verwenden. `lambda:InvokeFunction` Wenn Sie den Auslöser mit der Konsole erstellen, fügt Lambda diese Berechtigung automatisch für Sie hinzu.

Um anderen AWS Benutzern die Erlaubnis zu erteilen, auf Ihre Funktion zuzugreifen, können Sie dies in der ressourcenbasierten Richtlinie Ihrer Funktion genauso definieren wie für eine andere OR-Ressource. AWS-Service Sie können auch eine identitätsbasierte Richtlinie verwenden, die dem Benutzer zugeordnet ist.

Bewährte Methoden für Lambda-Berechtigungen

Wenn Sie Berechtigungen mithilfe von IAM-Richtlinien festlegen, empfiehlt es sich aus Sicherheitsgründen nur die zum Ausführen einer Aufgabe erforderlichen Berechtigungen zu erteilen. Dies wird als Prinzip der geringsten Berechtigungen bezeichnet. Um mit der Gewährung von Berechtigungen für Ihre Funktion zu beginnen, können Sie sich für die Verwendung einer AWS verwalteten Richtlinie entscheiden. Verwaltete Richtlinien können die schnellste und einfachste Methode sein, um Berechtigungen für die Ausführung einer Aufgabe zu erteilen. Sie können jedoch auch andere Berechtigungen beinhalten, die Sie nicht benötigen. Wenn Sie von der frühen Entwicklungsphase über die Testphase bis hin zur Produktion voranschreiten, empfehlen wir Ihnen, die Berechtigungen auf die erforderlichen zu reduzieren, indem Sie Ihre eigenen vom Kunden verwalteten Richtlinien definieren.

Das gleiche Prinzip gilt für die Erteilung von Berechtigungen für den Zugriff auf Ihre Funktion mithilfe einer ressourcenbasierten Richtlinie. Wenn Sie beispielsweise Amazon S3 die Erlaubnis geben möchten, Ihre Funktion aufzurufen, empfiehlt es sich, den Zugriff auf einzelne Buckets oder Buckets im Besonderen zu beschränken AWS-Konten, anstatt dem S3-Service pauschale Berechtigungen zu erteilen.

Ausführen von Code mit Lambda

Wenn Sie eine Lambda-Funktion schreiben, erstellen Sie Code, der in einer eindeutigen Serverless-Umgebung ausgeführt wird. Um zu verstehen, wie Lambda Ihren Code tatsächlich ausführt, müssen Sie zwei wichtige Aspekte kennen: das Programmiermodell, das definiert, wie Ihr Code mit Lambda interagiert, und den Lebenszyklus der Ausführungsumgebung, der bestimmt, wie Lambda die Laufzeitumgebung Ihres Codes verwaltet.

Das Lambda-Programmiermodell

Das Programmiermodell dient als allgemeines Regelwerk dafür, wie Lambda mit Ihrem Code arbeitet, unabhängig davon, ob Sie in Python, Java oder einer anderen unterstützten Sprache schreiben. Das Programmiermodell umfasst Ihre Laufzeit und Ihren Handler.

Für Standardfunktionen:

1. Lambda empfängt ein Ereignis.
2. Lambda verwendet die Laufzeit, um das Ereignis in einem Format vorzubereiten, das Ihr Code verwenden kann.
3. Die Laufzeit sendet das formatierte Ereignis an Ihren Handler.
4. Ihr Handler verarbeitet das Ereignis mit dem Code, den Sie geschrieben haben.

Für langlebige Funktionen:

1. Lambda empfängt ein Ereignis
2. Die Runtime bereitet sowohl das Ereignis als auch DurableContext
3. Ihr Handler kann:
   - Prozessschritte mit automatischem Checkpointing
   - Unterbrechen Sie die Ausführung, ohne Ressourcen zu verbrauchen
   - Fahren Sie mit dem letzten erfolgreichen Checkpoint fort
   - Behalten Sie den Status zwischen den Schritten bei

Wesentlich für dieses Modell ist der Handler, an den Lambda Ereignisse sendet, die von Ihrem Code verarbeitet werden sollen. Betrachten Sie ihn als Einstiegspunkt für Ihren Code. Wenn Lambda ein Ereignis empfängt, leitet es dieses Ereignis und einige Kontextinformationen an Ihren Handler

weiter. Der Handler führt dann Ihren Code aus, um diese Ereignisse zu verarbeiten – beispielsweise kann er eine Datei lesen, wenn sie in Amazon S3 hochgeladen wird, ein Bild analysieren oder eine Datenbank aktualisieren. Sobald Ihr Code die Verarbeitung eines Ereignisses abgeschlossen hat, ist der Handler bereit, das nächste Ereignis zu verarbeiten.

Das Lambda-Ausführungsmodell

Während das Programmiermodell definiert, wie Lambda mit Ihrem Code interagiert, ist die Ausführungsumgebung der Ort, an dem Lambda Ihre Funktion tatsächlich ausführt — es handelt sich um einen sicheren, isolierten Rechenraum, der speziell für Ihre Funktion erstellt wurde.

Jede Umgebung folgt einem Lebenszyklus, der zwischen Standardfunktionen und dauerhaften Funktionen variiert:

Standardfunktionen (bis zu 15 Minuten):

1. Initialisierung: Einrichtung der Umgebung und Laden von Code
2. Aufruf: Einmalige Ausführung des Funktionscodes
3. Herunterfahren: Bereinigung der Umgebung

Dauerhafte Funktionen (bis zu 1 Jahr):

1. Initialisierung: Einrichtung der Umgebung und des dauerhaften Zustands
2. Aufruf: Mehrere Schritte mit automatischem Checkpointing
3. Wartestatus: Unterbrechen Sie die Ausführung ohne Ressourcenverbrauch
4. Fortfahren: Vom letzten Checkpoint aus neu starten
5. Herunterfahren: Bereinigung des dauerhaften Zustands

Diese Umgebung übernimmt wichtige Aspekte der Ausführung Ihrer Funktion. Sie stellt Ihrer Funktion Speicher und ein `/tmp`-Verzeichnis für die temporäre Speicherung zur Verfügung. Für langlebige Funktionen verwaltet es außerdem:

- Automatische Beibehaltung des Zustands zwischen den Schritten
- Checkpoint-Speicherung und Wiederherstellung
- Warten Sie die staatliche Koordination
- Verfolgung des Fortschritts bei lang andauernden Ausführungen

Grundlegendes zum Lambda-Programmiermodell

Lambda bietet zwei Programmiermodelle: Standardfunktionen, die bis zu 15 Minuten laufen, und Durable Functions, die bis zu einem Jahr laufen können. Beide haben zwar gemeinsame Kernkonzepte, Durable Functions bieten jedoch zusätzliche Funktionen für langwierige, zustandsbehaftete Workflows.

Lambda stellt ein Programmiermodell bereit, das allen Laufzeitumgebungen gemeinsam ist. Das Programmiermodell definiert die Schnittstelle zwischen Ihrem Code und dem Lambda-System. Sie teilen Lambda den Einstiegspunkt für Ihre Funktion mit, indem Sie einen Handler in der Funktionskonfiguration definieren. Die Laufzeit übergibt Objekte an den Handler, die das Aufrufereignis und den Kontext, wie z. B. den Funktionsnamen und die Anforderungs-ID, enthalten.

Bei Durable Functions erhält der Handler außerdem ein DurableContext Objekt, das Folgendes bereitstellt:

- Überprüfen Sie die Funktionen mithilfe von step ()
- Warten Sie die Statusverwaltung mit wait () und waitForCallback ()
- Automatische Statuspersistenz zwischen Aufrufen

Wenn der Handler die Verarbeitung des ersten Ereignisses beendet hat, sendet die Laufzeit ihm ein anderes. Bei dauerhaften Funktionen kann der Handler die Ausführung zwischen den Schritten unterbrechen, und Lambda speichert den Status automatisch und stellt ihn wieder her, wenn die Funktion wieder aufgenommen wird. Die Klasse der Funktion bleibt im Speicher, sodass Clients und Variablen, die außerhalb der Handler-Methode in Initialisierungscodedeklariert sind, wiederverwendet werden können. Um die Verarbeitungszeit bei nachfolgenden Ereignissen zu verkürzen, erstellen Sie während der Initialisierung wiederverwendbare Ressourcen wie AWS SDK-Clients. Nach der Initialisierung kann jede Instance Ihrer Funktion Tausende von Anfragen verarbeiten.

Ihre Funktion hat auch Zugriff auf den lokalen Speicher im `/tmp`-Verzeichnis, einen temporären Zwischenspeicher, der für mehrere Aufrufe verwendet werden kann. Weitere Informationen finden Sie unter Ausführungsumgebung.

Wenn die AWS X-Ray -Ablaufverfolgung aktiviert ist, zeichnet die Laufzeitumgebung separate Untersegmente für die Initialisierung und Ausführung auf.

Die Runtime erfasst die Protokollausgabe Ihrer Funktion und sendet sie an Amazon CloudWatch Logs. Neben der Protokollierung der Ausgabe Ihrer Funktion protokolliert die Laufzeitumgebung auch Einträge, wenn der Funktionsaufruf gestartet und beendet wird. Dazu gehört ein Berichtsprotokoll

mit der Anforderungskennung, der fakturierten Dauer, der Initialisierungsdauer und weiteren Details. Wenn Ihre Funktion einen Fehler ausgibt, gibt die Laufzeit diesen Fehler an den Aufrufer zurück.

> **Note**
>
> Die Protokollierung unterliegt den CloudWatch Log-Kontingenten. Protokolldaten können durch Ablehnung verloren gehen, oder, in einigen Fällen, wenn eine Instance Ihrer Funktion gestoppt wird.

Hauptunterschiede bei Durable Functions:

- Der Status wird zwischen den Schritten automatisch beibehalten
- Funktionen können die Ausführung unterbrechen, ohne Ressourcen zu verbrauchen
- Bei einem Fehler werden die Schritte automatisch wiederholt
- Der Fortschritt wird anhand von Checkpoints verfolgt

Lambda skaliert Ihre Funktion, indem zusätzliche Instances davon ausgeführt werden, wenn der Bedarf steigt und indem Instances beendet werden, wenn der Bedarf sinkt. Dieses Modell führt zu Variationen in der Anwendungsarchitektur, wie zum Beispiel:

- Sofern nicht anders angegeben, werden eingehende Anforderungen nicht in der richtigen Reihenfolge oder gleichzeitig verarbeitet.
- Verlassen Sie sich nicht darauf, dass die Instances Ihrer Funktion langlebig sind, sondern speichern Sie den Zustand Ihrer Anwendung an anderer Stelle.
- Verwenden Sie lokale Speicher und Objekte auf Klassenebene, um die Leistung zu steigern, aber halten Sie die Größe Ihres Bereitstellungspakets und die Datenmenge, die Sie in die Ausführungsumgebung übertragen, möglichst gering.

In den folgenden Kapiteln finden Sie eine praktische Einführung in das Programmiermodell in Ihrer bevorzugten Programmiersprache.

- Erstellen von Lambda-Funktionen mit Node.js
- Erstellen von Lambda-Funktionen mit Python
- Erstellen von Lambda-Funktionen mit Ruby
- Erstellen von Lambda-Funktionen mit Java

- Erstellen von Lambda-Funktionen mit Go
- Erstellen von Lambda-Funktionen mit C#
- Aufbau von Lambda-Funktionen mit PowerShell

Verständnis des Lebenszyklus der Lambda-Ausführungsumgebung

Lambda-Ausführungsumgebungen unterstützen sowohl Standardfunktionen (bis zu 15 Minuten) als auch dauerhafte Funktionen (bis zu einem Jahr). Beide haben zwar denselben grundlegenden Lebenszyklus, Durable Functions bieten jedoch zusätzliche Funktionen zur Statusverwaltung für Workflows mit langer Laufzeit.

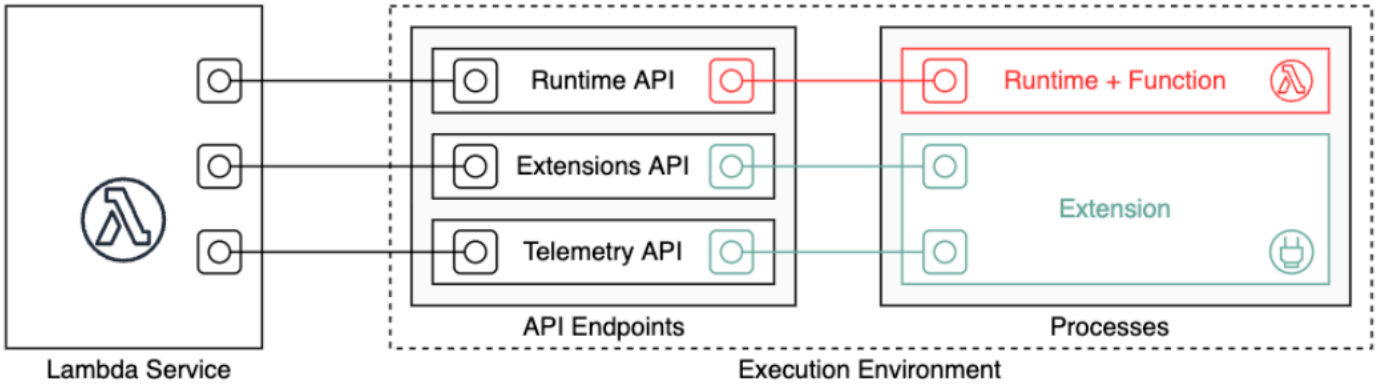
Lambda ruft Ihre Funktion in einer Ausführungsumgebung auf, die eine sichere und isolierte Laufzeitumgebung bereitstellt. Die Ausführungsumgebung verwaltet die Ressourcen, die zum Ausführen Ihrer Funktion erforderlich sind. Die Ausführungsumgebung bietet auch Lebenszyklusunterstützung für die Laufzeit der Funktion und alle externen Erweiterungen die mit Ihrer Funktion verknüpft sind.

Für Durable Functions umfasst die Ausführungsumgebung zusätzliche Komponenten für:

- Persistenz des Zustands zwischen den Schritten
- Verwaltung von Kontrollpunkten
- Warten Sie auf die staatliche Koordination
- Verfolgung der Fortschritte

> **Ausführungsumgebung für Lambda Managed Instances**
>
> Wenn Sie Lambda Managed Instances verwenden, weist die Ausführungsumgebung wichtige Unterschiede zu Lambda-Funktionen (Standard) auf. Verwaltete Instances unterstützen gleichzeitige Aufrufe, verwenden ein anderes Lebenszyklusmodell und werden auf einer kundeneigenen Infrastruktur ausgeführt. Ausführliche Informationen zur Ausführungsumgebung von Managed Instances finden Sie unter. Grundlegendes zur Ausführungsumgebung von Lambda Managed Instances

Die Laufzeitumgebung der Funktion kommuniziert Lambda mit der Laufzeit-API. Erweiterungen kommunizieren mit Lambda über die Erweiterungs-API. Erweiterungen können mithilfe der Telemetrie-API auch Protokollnachrichten und andere Telemetriedaten von der Funktion empfangen.



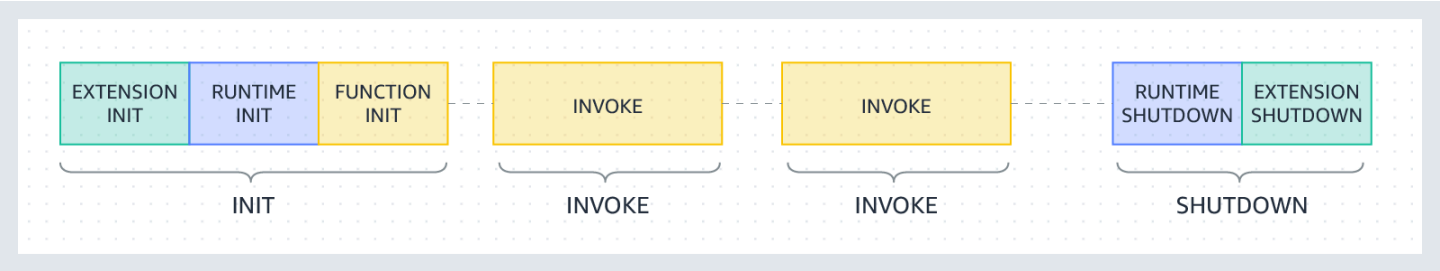
Wenn Sie Ihre Lambda-Funktion erstellen, geben Sie Konfigurationsinformationen an, z. B. den verfügbaren Arbeitsspeicher und die maximal zulässige Ausführungszeit für Ihre Funktion. Lambda verwendet diese Informationen, um die Ausführungsumgebung einzurichten.

Die Laufzeit der Funktion und jede externe Erweiterung sind Prozesse, die innerhalb der Ausführungsumgebung ausgeführt werden. Berechtigungen, Ressourcen, Anmeldeinformationen und Umgebungsvariablen werden von der Funktion und den Erweiterungen gemeinsam genutzt.

Themen

- Lebenszyklus der Lambda-Ausführungsumgebung
- Kaltstarts und Latenzzeiten
- Reduzierung von Kaltstarts mit bereitgestellter Parallelität
- Optimierung der statischen Initialisierung

Lebenszyklus der Lambda-Ausführungsumgebung



Jede Phase beginnt mit einem Ereignis, das Lambda an die Laufzeit und an alle registrierten Erweiterungen sendet. Die Laufzeit und jede Erweiterung zeigen den Abschluss durch Senden einer `Next`-API-Anfrage an. Lambda friert die Ausführungsumgebung ein, wenn die Laufzeit und jede Erweiterung abgeschlossen sind und keine ausstehenden Ereignisse vorhanden sind.

Zu den Lebenszyklusphasen von Durable Functions gehören:

- Init: Standardinitialisierung plus Einrichtung eines dauerhaften Zustands
- Invoke: Kann die Ausführung mehrerer Schritte mit automatischem Checkpointing beinhalten
- Warte: Die Funktion kann die Ausführung unterbrechen, ohne Ressourcen zu verbrauchen
- Fortfahren: Die Funktion wird vom letzten Checkpoint aus neu gestartet
- Herunterfahren: Wiederherstellung des dauerhaften Zustands und der Ressourcen

Topics

- Init-Phase
- Fehler in der Initialisierungsphase
- Wiederherstellungsphase (SnapStart nur Lambda)
- Invoke-Phase
- Fehler während der Aufrufphase
- Shutdown-Phase

Init-Phase

Lambda führt in der `Init`-Phase drei Aufgaben aus:

- Alle Erweiterungen starten (`Extension init`)
- Laufzeit-Bootstrap (`Runtime init`)
- Ausführen des statischen Codes der Funktion (`Function init`)
- Alle Runtime-Hooks vor dem Checkpoint ausführen (nur Lambda) SnapStart

Die `Init`-Phase endet, wenn die Laufzeit und alle Erweiterungen signalisieren, dass sie bereit sind, indem sie eine `Next`-API-Anforderung senden. Die `Init`-Phase ist auf 10 Sekunden begrenzt. Wenn alle drei Aufgaben nicht innerhalb von 10 Sekunden abgeschlossen werden, versucht Lambda die `Init`-Phase zum Zeitpunkt des ersten Funktionsaufrufs mit dem konfigurierten Funktionstimeout erneut.

Wenn Lambda SnapStart aktiviert ist, findet die `Init`-Phase statt, wenn Sie eine Funktionsversion veröffentlichen. Lambda speichert einen Snapshot des Arbeitsspeichers und des Festplattenzustands der initialisierten Ausführungsumgebung, speichert den verschlüsselten Snapshot und speichert

ihn im Cache für den Zugriff mit geringer Latenz. Wenn Sie einen Laufzeit-Hook vor dem Prüfpunkt haben, wird der Code am Ende der `Init`-Phase ausgeführt.

> **Note**
>
> Das 10-Sekunden-Timeout gilt nicht für Funktionen, die bereitgestellte Parallelität oder Lambda Managed SnapStart Instances verwenden. Für die Funktionen Provisioned Concurrency und Managed Instances kann Ihr Initialisierungscode bis zu 15 Minuten lang ausgeführt werden. SnapStart Das Zeitlimit beträgt 130 Sekunden oder das konfigurierte Funktions-Timeout (maximal 900 Sekunden), je nachdem, welcher Wert höher ist.

Wenn Sie die bereitgestellte Gleichzeitigkeit verwenden, initialisiert Lambda die Ausführungsumgebung, wenn Sie die PC-Einstellungen für eine Funktion konfigurieren. Lambda stellt außerdem sicher, dass initialisierte Ausführungsumgebungen vor Aufrufen immer verfügbar sind. Möglicherweise treten Lücken zwischen den Aufruf- und Initialisierungsphasen Ihrer Funktion auf. Abhängig von der Laufzeit- und Speicherkonfiguration Ihrer Funktion können beim ersten Aufruf in einer initialisierten Ausführungsumgebung auch variable Latenzzeiten auftreten.

Bei Funktionen, die On-Demand-Gleichzeitigkeit nutzen, initialisiert Lambda gelegentlich Ausführungsumgebungen vor Aufrufanfragen. In diesem Fall stellen Sie möglicherweise auch eine Zeitlücke zwischen der Initialisierungs- und der Aufrufphase Ihrer Funktion fest. Wir empfehlen Ihnen, sich nicht von diesem Verhalten abhängig zu machen.

Fehler in der Initialisierungsphase

Wenn eine Funktion in der `Init`-Phase abstürzt oder ein Timeout auftritt, gibt Lambda Fehlerinformationen im `INIT_REPORT`-Protokoll aus.

Example – INIT_REPORT-Protokoll für Timeout

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

Example – INIT_REPORT-Protokoll für den Erweiterungsfehler

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

Wenn die `Init` Phase erfolgreich ist, gibt Lambda das `INIT_REPORT` Protokoll nicht aus, es sei denn SnapStart, die bereitgestellte Parallelität ist aktiviert. SnapStart und bereitgestellte

Parallelitätsfunktionen werden immer ausgegeben. `INIT_REPORT` Weitere Informationen finden Sie unter Überwachung für Lambda SnapStart.

Wiederherstellungsphase (SnapStart nur Lambda)

Wenn Sie eine SnapStartFunktion zum ersten Mal aufrufen und die Funktion skaliert wird, nimmt Lambda neue Ausführungsumgebungen aus dem persistenten Snapshot wieder auf, anstatt die Funktion von Grund auf neu zu initialisieren. Wenn Sie über einen Laufzeit-Hook nach der Wiederherstellung verfügen, wird der Code am Ende der `Restore`-Phase ausgeführt. Die Dauer von Laufzeit-Hooks nach der Wiederherstellung wird Ihnen in Rechnung gestellt. Die Laufzeit muss geladen werden und Laufzeit-Hooks nach der Wiederherstellung müssen innerhalb des Timeout-Limits (10 Sekunden) abgeschlossen werden. Andernfalls erhalten Sie eine. SnapStartTimeoutException Wenn die `Restore`-Phase abgeschlossen ist, ruft Lambda den Funktionshandler (Invoke-Phase) auf.

Fehler in der Wiederherstellphase

Wenn die `Restore`-Phase fehlschlägt, gibt Lambda Fehlerinformationen im `RESTORE_REPORT`-Protokoll aus.

Example – RESTORE_REPORT-Protokoll für Timeout

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

Example – RESTORE_REPORT-Protokoll für einen Laufzeit-Hook-Fehler

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

Weitere Informationen zum `RESTORE_REPORT`-Protokoll finden Sie unter Überwachung für Lambda SnapStart.

Invoke-Phase

Wenn eine Lambda-Funktion als Antwort auf eine `Next`-API-Anforderung aufgerufen wird, sendet Lambda ein `Invoke`-Ereignis an die Laufzeit und an jede Erweiterung.

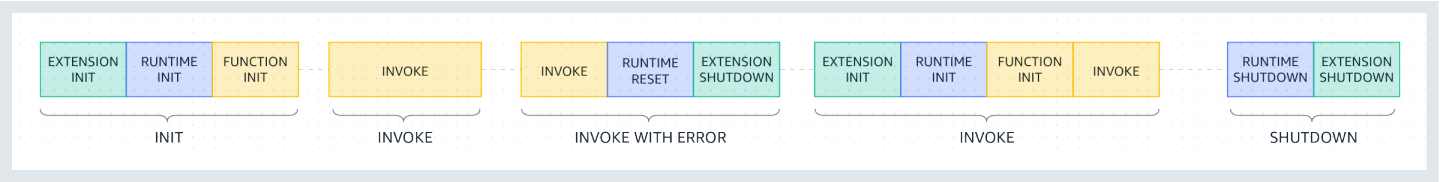
Die Timeout-Einstellung der Funktion begrenzt die Dauer der gesamten `Invoke`-Phase. Wenn Sie beispielsweise die Zeitüberschreitung für die Funktion auf 360 Sekunden festlegen, müssen die Funktion und alle Erweiterungen innerhalb von 360 Sekunden abgeschlossen werden.

Beachten Sie, dass es keine unabhängige Post-Invoke-Phase gibt. Die Dauer ist die Summe der gesamten Aufrufzeit (Laufzeit + Erweiterungen) und wird erst berechnet, wenn die Funktion und alle Erweiterungen vollständig ausgeführt wurden.

Die Invoke-Phase endet nach der Laufzeit und alle Erweiterungen signalisieren durch Senden einer `Next`-API-Anforderung dass sie abgeschlossen sind.

Fehler während der Aufrufphase

Wenn die Lambda-Funktion während der `Invoke`-Phase abstürzt oder das Zeitlimit überschreitet, setzt Lambda die Ausführungsumgebung zurück. Das folgende Diagramm veranschaulicht das Verhalten der Lambda-Ausführungsumgebung bei einem Aufruffehler:



In folgendem Diagramm:

- Ist die erste Phase die INIT-Phase, die ohne Fehler läuft.
- Ist die zweite Phase die INVOKE-Phase, die ohne Fehler läuft.
- Angenommen, Ihre Funktion stößt irgendwann auf einen Aufruffehler (häufige Ursachen sind Funktions-Timeouts, Laufzeitfehler, Speichererschöpfung, VPC-Konnektivitätsprobleme, Berechtigungsfehler, Parallelitätsbeschränkungen und verschiedene Konfigurationsprobleme). Eine vollständige Liste möglicher Aufruffehler finden Sie unter the section called “Aufruf”. Die dritte Phase mit der Bezeichnung INVOKE MIT FEHLER veranschaulicht dieses Szenario. Wenn dies geschieht, führt der Lambda-Service einen Neustart durch. Der Reset verhält sich wie ein `Shutdown`-Ereignis. Zuerst beendet Lambda die Laufzeit und sendet dann ein `Shutdown`-Ereignis an jede registrierte externe Erweiterung. Das Ereignis enthält den Grund für das Abschalten. Wenn diese Umgebung für einen neuen Aufruf verwendet wird, initialisiert Lambda die Erweiterung und die Laufzeit als Teil des nächsten Aufrufers erneut.

  Beachten Sie, dass der Lambda-Reset den Inhalt des `/tmp`-Verzeichnisses vor der nächsten Init-Phase nicht löscht. Dieses Verhalten stimmt mit der regulären Shutdown-Phase überein.

  > **Note**
  >
  > AWS implementiert derzeit Änderungen am Lambda-Service. Aufgrund dieser Änderungen können geringfügige Unterschiede in Struktur und Inhalt der Systemprotokollmeldungen

und Trace-Segmente auftreten, die von verschiedenen Lambda-Funktionen in Ihrem AWS-Konto.
Wenn die Systemprotokollkonfiguration Ihrer Funktion auf Klartext eingestellt ist, wirkt sich diese Änderung auf die Protokollnachrichten aus, die in CloudWatch Logs erfasst werden, wenn bei Ihrer Funktion ein Aufruffehler auftritt. Die folgenden Beispiele zeigen Protokollausgaben sowohl in alten als auch in neuen Formaten.
Diese Änderungen werden in den kommenden Wochen umgesetzt, und alle Funktionen AWS-Regionen außer China und den GovCloud Regionen werden auf die Verwendung der Protokollnachrichten und Trace-Segmente im neuen Format umgestellt.

Example CloudWatch Protokolliert die Protokollausgabe (Laufzeit- oder Erweiterungsabsturz) — alter Stil

```
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 Protokolliert die Protokollausgabe (Funktions-Timeout) im alten Stil

```
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
```

Das neue Format für CloudWatch Protokolle enthält ein zusätzliches `status` Feld in der `REPORT` Zeile. Im Falle eines Laufzeit- oder Erweiterungsabsturzes enthält die `REPORT`-Zeile auch ein Feld `ErrorType`.

Example CloudWatch Protokolliert die Protokollausgabe (Laufzeit- oder Erweiterungsabsturz) — neuer Stil

```
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 Protokolliert die Protokollausgabe (Funktions-Timeout) — neuer Stil

```
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
```

- Die vierte Phase stellt die INVOKE-Phase unmittelbar nach einem Aufruffehler dar. Hier initialisiert Lambda die Umgebung erneut, indem es die INIT-Phase erneut ausführt. Dies wird als unterdrückte Initialisierung bezeichnet. Wenn unterdrückte Inits auftreten, meldet Lambda nicht explizit eine zusätzliche INIT-Phase in Logs. CloudWatch Stattdessen stellen Sie möglicherweise fest, dass die Dauer in der REPORT-Zeile eine zusätzliche INIT-Dauer plus die INVOKE-Dauer enthält. Nehmen wir zum Beispiel an, Sie sehen die folgenden Logs in: 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
```

In diesem Beispiel beträgt der Unterschied zwischen den Zeitstempeln BERICHT und START 2,5 Sekunden. Dies entspricht nicht der angegebenen Dauer von 3 022,91 Millsekunden, da es die zusätzliche INIT (unterdrückte Initialisierung), die Lambda ausgeführt hat, nicht berücksichtigt. In diesem Beispiel können Sie ableiten, dass die tatsächliche INVOKE-Phase 2,5 Sekunden gedauert hat.

Für mehr Einblick in dieses Verhalten können Sie die Zugriff auf Echtzeit-Telemetriedaten für Erweiterungen über die Telemetrie-API verwenden. Die Telemetrie-API gibt `INIT_START`, `INIT_RUNTIME_DONE` und `INIT_REPORT`-Ereignisse mit `phase=invoke` immer dann aus, wenn unterdrückte Initialisierungen zwischen den Aufrufphasen auftreten.

- Die fünfte Phase stellt die SHUTDOWN-Phase dar, die fehlerfrei abläuft.

Shutdown-Phase

Wenn Lambda dabei ist, die Laufzeit herunterzufahren, sendet es ein `Shutdown`-Ereignis an jede registrierte externe Erweiterung. Erweiterungen können diese Zeit für abschließende Bereinigungsaufgaben verwenden. Das `Shutdown`-Ereignis ist eine Antwort auf eine `Next`-API-Anforderung.

Duration Limit (Begrenzung der Dauer): Die maximale Dauer der `Shutdown`-Phase hängt von der Konfiguration der registrierten Erweiterungen ab:

- 0 ms – Eine Funktion ohne registrierte Erweiterungen
- 500 ms – Eine Funktion mit einer registrierten internen Erweiterung
- 2.000 ms – Eine Funktion mit einer oder mehreren registrierten externen Erweiterungen

Wenn die Laufzeit oder eine Erweiterung nicht innerhalb des Limits auf das `Shutdown`-Ereignis reagiert, beendet Lambda den Prozess mit einem `SIGKILL`-Signal.

Nachdem die Funktion und alle Erweiterungen abgeschlossen sind, behält Lambda die Ausführungsumgebung für einige Zeit im Vorgriff auf einen anderen Funktionsaufruf bei. Lambda beendet jedoch alle paar Stunden die Ausführungsumgebungen, um Laufzeitaktualisierungen und -wartungen zu ermöglichen – selbst bei Funktionen, die kontinuierlich aufgerufen werden. Sie sollten nicht davon ausgehen, dass die Ausführungsumgebung auf unbestimmte Zeit bestehen bleibt. Weitere Informationen finden Sie unter Implementieren der Zustandslosigkeit in Funktionen.

Wenn die Funktion erneut aufgerufen wird, taut Lambda die Umgebung zur Wiederverwendung auf. Die Wiederverwendung der Ausführungsumgebung hat folgende Auswirkungen:

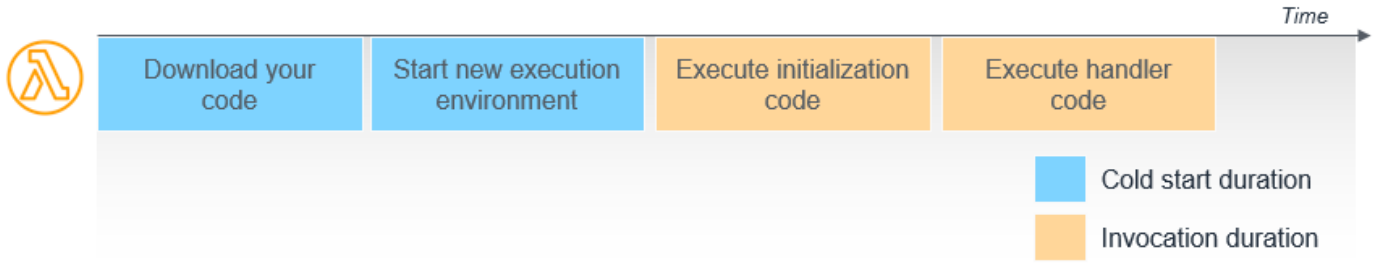
- Objekte, die außerhalb der Handler-Methode der Funktion deklariert sind, bleiben initialisiert und bieten eine zusätzliche Optimierung, wenn die Funktion erneut aufgerufen wird. Beispiel: Falls Ihre Lambda-Funktion eine Datenbankverbindung herstellt, statt die Verbindung neu herzustellen, wird die ursprüngliche Verbindung bei nachfolgenden Aufrufen verwendet. Wir empfehlen, dass Sie Ihrem Code eine Logik hinzufügen, um zu prüfen, ob eine Verbindung besteht, bevor Sie eine neue erstellen.
- Jede Ausführungsumgebung stellt Speicherplatz von 512 MB bis 10 240 MB, in 1-MB-Schritten, im Verzeichnis `/tmp` zur Verfügung. Der Inhalt des Verzeichnisses bleibt bestehen, wenn die Ausführungsumgebung eingefroren ist, und bietet einen temporären Zwischenspeicher, der für

mehrere Aufrufe verwendet werden kann. Sie können zusätzlichen Code hinzufügen, um zu prüfen, ob der Cache die gespeicherten Daten enthält. Weitere Informationen zu Beschränkungen der Bereitstellungsgröße finden Sie unter Lambda-Kontingente.

- Hintergrundprozesse oder Callbacks, die von Ihrer Lambda-Funktion initiiert wurden und nicht abgeschlossen wurden, wenn die Funktion beendet wurde, werden wieder aufgenommen, wenn Lambda die Ausführungsumgebung wiederverwendet. Sie sollten sicherstellen, dass alle Hintergrundprozesse oder Callbacks in Ihrem Code abgeschlossen sind, bevor der Code beendet wird.

Kaltstarts und Latenzzeiten

Wenn Lambda eine Anfrage zur Ausführung einer Funktion über die Lambda-API erhält, bereitet der Dienst zunächst eine Ausführungsumgebung vor. Während dieser Initialisierungsphase lädt der Dienst Ihren Code herunter, startet die Umgebung und führt jeglichen Initialisierungscode außerhalb des Haupt-Handlers aus. Schließlich führt Lambda den Handler-Code aus.



In diesem Diagramm werden die ersten beiden Schritte, das Herunterladen des Codes und das Einrichten der Umgebung, häufig als „Kaltstart“ bezeichnet. Diese Zeit wird Ihnen in Rechnung gestellt und verlängert die Gesamtdauer Ihres Aufrufs.

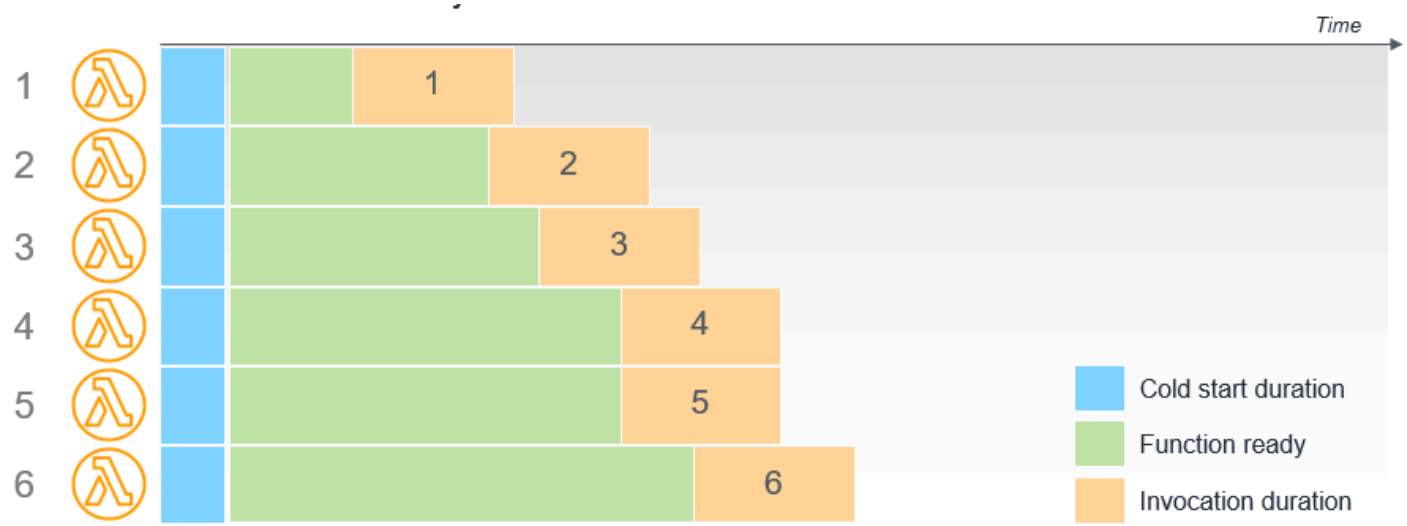
Nachdem der Aufruf abgeschlossen ist, wird die Ausführungsumgebung eingefroren. Um die Ressourcenverwaltung und die Leistung zu verbessern, behält Lambda die Ausführungsumgebung für einen bestimmten Zeitraum bei. Wenn während dieser Zeit eine andere Anfrage für dieselbe Funktion eingeht, kann Lambda die Umgebung wiederverwenden. Diese zweite Anfrage wird in der Regel schneller abgeschlossen, da die Ausführungsumgebung bereits vollständig eingerichtet ist. Dies wird als „Warmstart“ bezeichnet.

Kaltstarts treten in der Regel bei weniger als 1 % der Aufrufe auf. Die Dauer eines Kaltstarts variiert zwischen unter 100 ms und über 1 Sekunde. Im Allgemeinen kommen Kaltstarts in Entwicklungs- und Testfunktionen häufiger vor als in Produktions-Workloads. Das liegt daran, dass Entwicklungs- und Testfunktionen in der Regel seltener aufgerufen werden.

Reduzierung von Kaltstarts mit bereitgestellter Parallelität

Wenn Sie vorhersehbare Startzeiten von Funktionen für Ihren Workload benötigen, ist bereitgestellte Parallelität die empfohlene Lösung, um eine möglichst geringe Latenz sicherzustellen. Mit diesem Feature werden Ausführungsumgebungen vorinitialisiert, was die Anzahl der Kaltstarts reduziert.

Für eine Funktion mit einer bereitgestellten Parallelität von 6 werden beispielsweise 6 Ausführungsumgebungen vorbereitet.



Optimierung der statischen Initialisierung

Die statische Initialisierung erfolgt, bevor der Handlercode in einer Funktion ausgeführt wird. Dies ist der von Ihnen bereitgestellte Initialisierungscode, der sich außerhalb des Haupt-Handlers befindet. Dieser Code wird häufig verwendet, um Bibliotheken und Abhängigkeiten zu importieren, Konfigurationen einzurichten und Verbindungen zu anderen Diensten zu initialisieren.

Das folgende Python-Beispiel zeigt den Import und die Konfiguration von Modulen sowie die Erstellung des Amazon-S3-Clients während der Initialisierungsphase, bevor die `lambda_handler`-Funktion während des Aufrufs ausgeführt wird.

```
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...
```

Der größte Faktor für die Latenz vor der Funktionsausführung ist der Initialisierungscode. Dieser Code wird ausgeführt, wenn eine neue Ausführungsumgebung zum ersten Mal erstellt wird. Der Initialisierungscode wird nicht erneut ausgeführt, wenn ein Aufruf eine warme Ausführungsumgebung verwendet. Faktoren, die die Latenz des Initialisierungscodes beeinflussen, sind unter anderem:

- Die Größe des Funktionspakets in Bezug auf importierte Bibliotheken und Abhängigkeiten sowie Lambda-Schichten.
- Die Menge an Code und die Initialisierungsarbeit.
- Die Leistung von Bibliotheken und anderen Diensten beim Aufbau von Verbindungen und anderen Ressourcen.

Es gibt eine Reihe von Maßnahmen, mit denen Entwickler die Latenzzeit bei der statischen Initialisierung optimieren können. Wenn eine Funktion viele Objekte und Verbindungen hat, können Sie eine einzelne Funktion möglicherweise in mehrere spezialisierte Funktionen umstrukturieren. Diese sind einzeln kleiner und haben jeweils weniger Initialisierungscode.

Es ist wichtig, dass Funktionen nur die Bibliotheken und Abhängigkeiten importieren, die sie benötigen. Wenn Sie beispielsweise nur Amazon DynamoDB im AWS SDK verwenden, können Sie anstelle des gesamten SDK einen einzelnen Service verlangen. Vergleichen Sie die folgenden drei Beispiele:

```
// 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)
```

Die statische Initialisierung ist auch oft der beste Ort, um Datenbankverbindungen zu öffnen, damit eine Funktion Verbindungen über mehrere Aufrufe derselben Ausführungsumgebung

wiederverwenden kann. Möglicherweise verfügen Sie jedoch über eine große Anzahl von Objekten, die nur in bestimmten Ausführungspfaden in Ihrer Funktion verwendet werden. In diesem Fall können Sie Variablen im globalen Bereich langsam laden, um die Dauer der statischen Initialisierung zu reduzieren.

Vermeiden Sie globale Variablen für kontextspezifische Informationen. Wenn Ihre Funktion über eine globale Variable verfügt, die nur für die Lebensdauer eines einzelnen Aufrufs verwendet und für den nächsten Aufruf zurückgesetzt wird, verwenden Sie einen Variablenbereich, der für den Handler lokal ist. Dadurch wird nicht nur verhindert, dass globale Variablen bei Aufrufen verloren gehen, sondern verbessert auch die Leistung der statischen Initialisierung.

Erstellen ereignisgesteuerter Architekturen mit Lambda

Ein Ereignis ist alles, was das Ausführen einer Lambda-Funktion auslöst. Ereignisse können eine Lambda-Funktion auf zwei Arten auslösen: durch direkten Aufruf (Push) und durch Zuordnungen von Ereignisquellen (Pull).

Viele AWS Dienste können Ihre Lambda-Funktionen direkt aufrufen. Diese Services leiten Ereignisse an Ihre Lambda-Funktion weiter. Ereignisse, die Funktionen auslösen, können fast alles sein, von einer HTTP-Anfrage über API Gateway, einem durch eine EventBridge Regel verwalteten Zeitplan, einem AWS IoT Ereignis oder einem Amazon S3 S3-Ereignis. Bei der Zuordnung von Ereignisquellen ruft Lambda aktiv Ereignisse aus einer Warteschlange oder einem Stream ab (Pull). Sie konfigurieren Lambda so, dass Lambda nach Ereignissen von einem unterstützten Service sucht, und Lambda kümmert sich um die Abfrage und den Aufruf Ihrer Funktion.

Wenn Ereignisse an Ihre Funktion übergeben werden, sind sie im JSON-Format strukturiert. Die JSON-Struktur variiert je nach Service, von dem sie generiert wird, und dem Ereignistyp. Während Standardaufrufe von Lambda-Funktionen bis zu 15 Minuten (oder bis zu einem Jahr bei dauerhaften Funktionen) dauern können, eignet sich Lambda am besten für kurze Aufrufe, die eine Sekunde oder weniger dauern. Dies gilt insbesondere für ereignisgesteuerte Architekturen, bei denen jede Lambda-Funktion als Mikroservice behandelt wird, der für die Ausführung eines engen Satzes spezifischer Befehle verantwortlich ist.

> **Note**
>
> Ereignisgesteuerte Architekturen kommunizieren systemübergreifend mithilfe von Netzwerken, was zu variabler Latenz führt. Für Workloads, die eine sehr geringe Latenz erfordern, wie z. B. Handelssysteme in Echtzeit, ist dieses Design möglicherweise nicht die beste Wahl. Für hoch skalierbare und verfügbare Workloads oder solche mit unvorhersehbaren Datenverkehrsmustern können ereignisgesteuerte Architekturen jedoch eine effektive Möglichkeit bieten, diese Anforderungen zu erfüllen.

Themen

- Vorteile einer ereignisgesteuerten Architektur
- Kompromisse ereignisgesteuerter Architekturen
- Anti-Pattern in Lambda-basierten ereignisgesteuerten Anwendungen

Vorteile einer ereignisgesteuerten Architektur

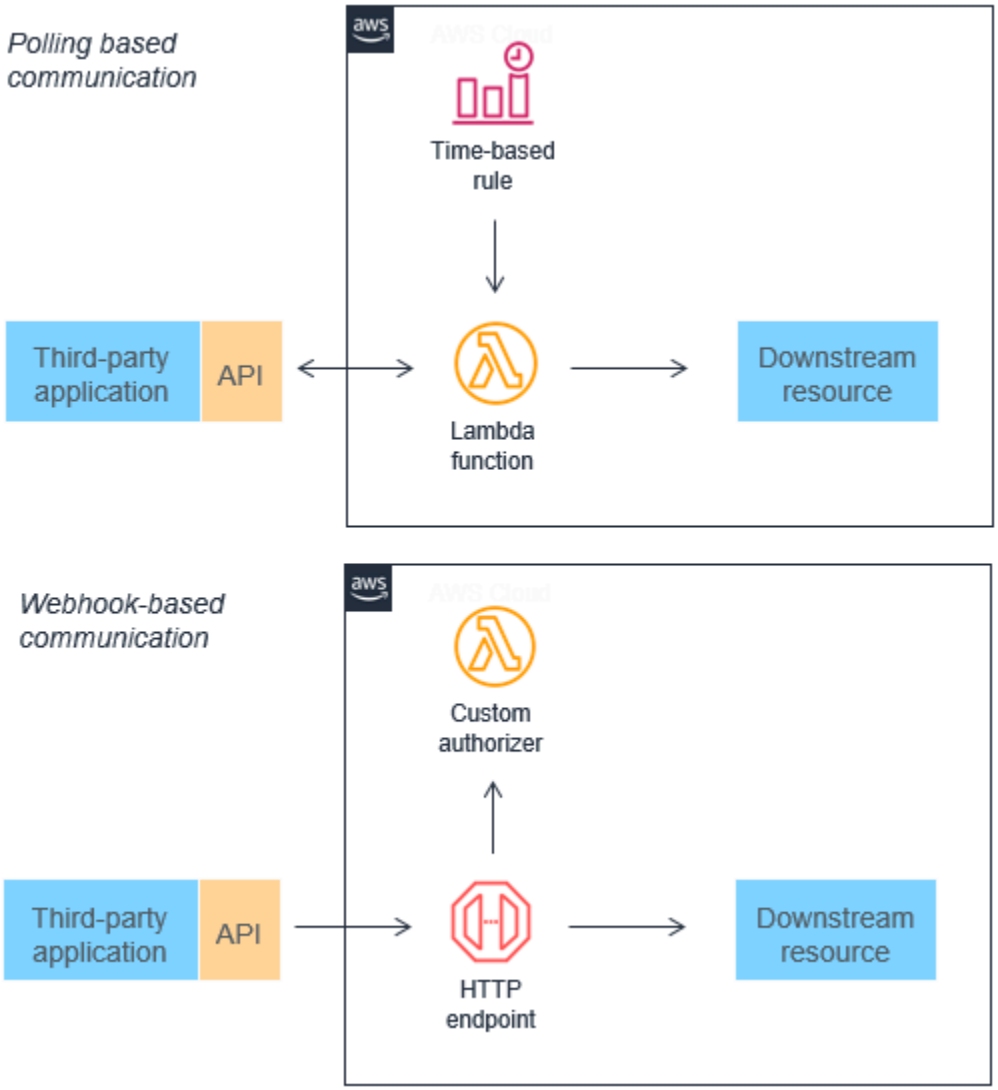
Lambda unterstützt zwei Aufrufmethoden in ereignisgesteuerten Architekturen:

1. Direkter Aufruf (Push-Methode): AWS Dienste lösen Lambda-Funktionen direkt aus. Beispiel:
   - Amazon S3 löst eine Funktion aus, wenn eine Datei hochgeladen wird.
   - API Gateway löst eine Funktion aus, wenn es eine HTTP-Anfrage empfängt.
2. Zuordnung von Ereignisquellen (Pull-Methode): Lambda ruft Ereignisse ab und ruft Funktionen auf. Beispiel:
   - Lambda ruft Nachrichten aus einer Amazon-SQS-Warteschlange ab und ruft eine Funktion auf.
   - Lambda liest Datensätze aus einem DynamoDB-Stream und ruft eine Funktion auf.

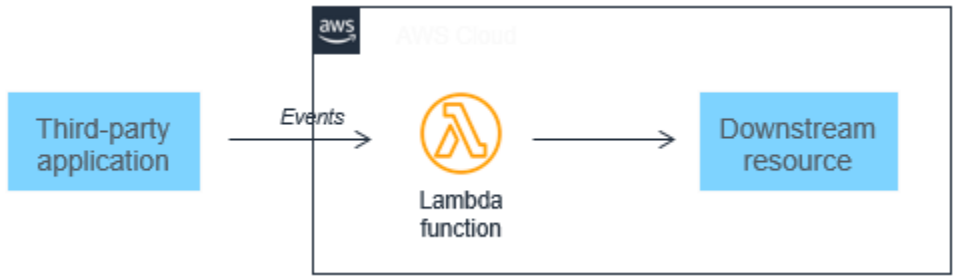
Beide Methoden tragen zu den Vorteilen von ereignisgesteuerten Architekturen bei, wie unten beschrieben.

Polling und Webhooks durch Ereignisse ersetzen

Viele traditionelle Architekturen verwenden Polling- und Webhook-Mechanismen, um den Status zwischen verschiedenen Komponenten zu kommunizieren. Das Abrufen von Aktualisierungen kann sehr ineffizient sein, da es eine Verzögerung zwischen der Verfügbarkeit neuer Daten und der Synchronisierung mit nachgelagerten Diensten gibt. Webhooks werden nicht immer von anderen Microservices unterstützt, in die Sie sich integrieren möchten. Sie erfordern möglicherweise auch benutzerdefinierte Autorisierungs- und Authentifizierungskonfigurationen. In beiden Fällen ist es schwierig, diese Integrationsmethoden ohne zusätzliche Arbeit der Entwicklungsteams bedarfsgerecht zu skalieren.



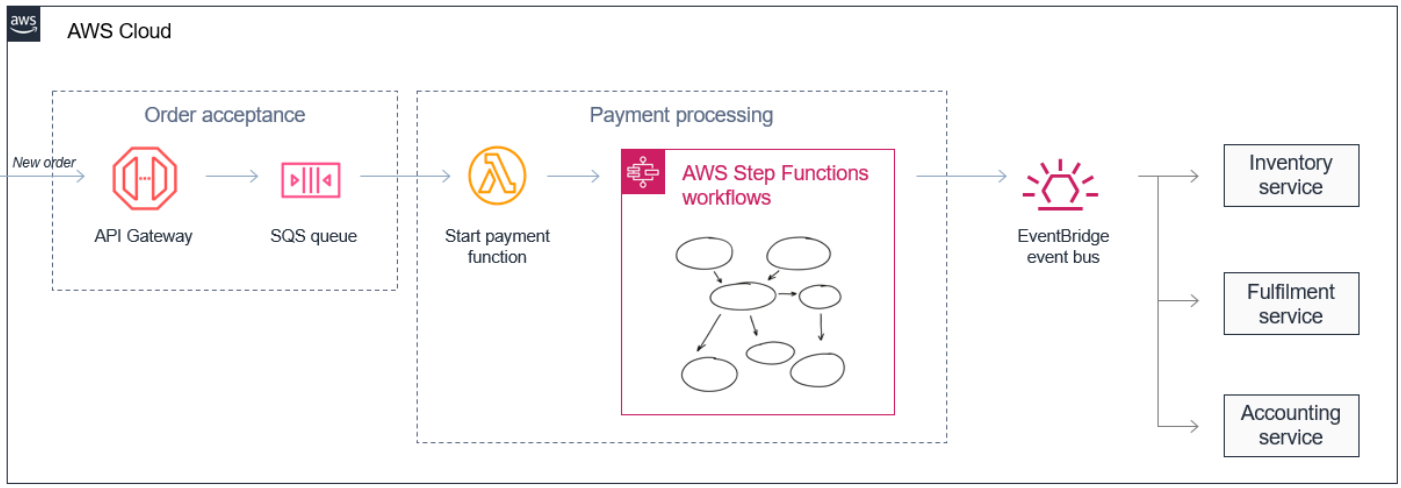
Diese beiden Mechanismen können durch Ereignisse ersetzt werden, die gefiltert, weitergeleitet und an nachgelagerte Microservices weitergeleitet werden können. Dieser Ansatz kann zu einem geringeren Bandbreitenverbrauch, einer geringeren CPU-Auslastung und potenziell niedrigeren Kosten führen. Diese Architekturen können auch die Komplexität verringern, da jede Funktionseinheit kleiner ist und oft weniger Code enthalten ist.



Ereignisgesteuerte Architekturen können auch das Entwerfen von near-real-time Systemen erleichtern und Unternehmen dabei helfen, sich von der stapelbasierten Verarbeitung abzuwenden. Ereignisse werden zu dem Zeitpunkt erzeugt, an dem sich der Zustand der Anwendung ändert. Der benutzerdefinierte Code eines Microservices sollte daher so konzipiert sein, dass er die Verarbeitung eines einzelnen Ereignisses verarbeiten kann. Da die Skalierung durch den Lambda-Dienst erfolgt, kann diese Architektur einen erheblichen Anstieg des Datenverkehrs ohne Änderung des benutzerdefinierten Codes bewältigen. Mit der Zunahme von Ereignissen wächst auch die Datenverarbeitungsschicht, die die Ereignisse verarbeitet.

Reduzierung der Komplexität

Microservices ermöglichen es Entwicklern und Architekten, komplexe Workflows zu vereinfachen. Ein E-Commerce-Monolith kann beispielsweise in Auftragsannahme- und Zahlungsprozesse mit separaten Inventar-, Fulfillment- und Buchhaltungsdienstleistungen unterteilt werden. Was in einem Monolith komplex zu verwalten und zu orchestrieren sein könnte, wird zu einer Reihe von entkoppelten Diensten, die asynchron mit Ereignissen kommunizieren.



Dieser Ansatz ermöglicht auch die Zusammenstellung von Diensten, die Daten mit unterschiedlichen Geschwindigkeiten verarbeiten. In diesem Fall kann ein Microservice für die Auftragsannahme große Mengen eingehender Bestellungen speichern, indem er die Nachrichten in einer Amazon SQS SQS-Warteschlange zwischenspeichert.

Ein Zahlungsabwicklungsservice, der aufgrund der Komplexität der Zahlungsabwicklung in der Regel langsamer ist, kann einen stetigen Strom von Nachrichten aus der Amazon SQS-Warteschlange entgegennehmen. Er kann komplexe Wiederholungsversuche und Fehlerbehandlungslogik mithilfe von AWS Step Functions aktiven Zahlungsabläufen für Hunderttausende von Bestellungen orchestrieren und diese koordinieren.

Verbesserung der Skalierbarkeit und Erweiterbarkeit

Microservices generieren Ereignisse, die in der Regel in Messaging-Diensten wie Amazon SNS und Amazon SQS veröffentlicht werden. Diese verhalten sich wie ein elastischer Puffer zwischen Microservices und helfen bei der Skalierung, wenn der Datenverkehr zunimmt. Dienste wie Amazon EventBridge können dann Nachrichten je nach Inhalt des Ereignisses filtern und weiterleiten, wie in den Regeln definiert. Infolgedessen sind ereignisbasierte Anwendungen skalierbarer und bieten eine größere Redundanz als monolithische Anwendungen.

Dieses System ist außerdem in hohem Maße erweiterbar, so dass andere Teams die Features erweitern und neue Features hinzufügen können, ohne dass dies Auswirkungen auf die Microservices für die Auftrags- und Zahlungsabwicklung hat. Durch die Veröffentlichung von Ereignissen mithilfe EventBridge dieser Anwendung lässt sie sich in bestehende Systeme wie den Inventar-Microservice integrieren, ermöglicht aber auch die Integration jeder future Anwendung als Event-Consumer. Die Produzenten von Ereignissen haben keine Kenntnis über die Konsumenten von Ereignissen, was zur Vereinfachung der Logik des Mikrodienstes beitragen kann.

Kompromisse ereignisgesteuerter Architekturen

Variable Latenzzeit

Im Gegensatz zu monolithischen Anwendungen, die möglicherweise alles verarbeiten, was sich im selben Speicherbereich auf einem einzigen Gerät befindet, kommunizieren ereignisgesteuerte Anwendungen netzwerkübergreifend. Dieses Design führt zu einer variablen Latenzzeit. Es ist zwar möglich, Anwendungen so zu entwickeln, dass die Latenzzeit minimiert wird, aber monolithische Anwendungen können fast immer auf Kosten der Skalierbarkeit und Verfügbarkeit für eine geringere Latenzzeit optimiert werden.

Workloads, die eine konsistente Leistung mit geringer Latenz erfordern, wie z. B. Hochfrequenz-Handelsanwendungen in Banken oder die Submillisekunden-Automatisierung von Robotern in Lagerhäusern, sind keine guten Kandidaten für eine ereignisgesteuerte Architektur.

Letztendliche Datenkonsistenz

Ein Ereignis stellt eine Zustandsänderung dar und da zu einem bestimmten Zeitpunkt viele Ereignisse durch verschiedene Dienste in einer Architektur fließen, sind solche Workloads oft irgendwann konsistent. Dies erschwert die Verarbeitung von Transaktionen, den Umgang mit Duplikaten oder die Ermittlung des genauen Gesamtzustands eines Systems.

Einige Workloads enthalten eine Kombination von Anforderungen, die eventuell konsistent (z. B. Gesamtaufträge in der aktuellen Stunde) oder stark konsistent (z. B. aktueller Bestand) sind. Für

Workloads, die eine hohe Datenkonsistenz erfordern, gibt es Architekturmuster, die dies unterstützen. Beispiel:

- DynamoDB kann strikt konsistente Lesevorgänge bereitstellen, manchmal mit einer höheren Latenz, wodurch ein höherer Durchsatz als im Standardmodus verbraucht wird. DynamoDB kann auch Transaktionen unterstützen, um die Datenkonsistenz aufrechtzuerhalten.
- Sie können Amazon RDS für Features verwenden, die ACID-Eigenschaften benötigen, obwohl relationale Datenbanken weniger skalierbar sind als ein NoSQL-Datenbanken wie DynamoDB. Amazon-RDS-Proxy kann dabei helfen, das Verbindungspooling und die Skalierung von kurzlebigen Verbrauchern wie Lambda-Funktionen zu verwalten.

Ereignisbasierte Architekturen sind in der Regel auf einzelne Ereignisse und nicht auf große Datenmengen ausgelegt. Im Allgemeinen sind Workflows so konzipiert, dass sie die Schritte eines einzelnen Ereignisses oder Ausführungsablaufs verwalten, anstatt mehrere Ereignisse gleichzeitig zu bearbeiten. In Serverless-Systemen wird die Echtzeit-Ereignisverarbeitung der Batch-Verarbeitung vorgezogen: Batches sollten durch viele kleinere inkrementelle Aktualisierungen ersetzt werden. Dies kann zwar die Verfügbarkeit und Skalierbarkeit von Workloads verbessern, macht es aber auch schwieriger für Ereignisse, andere Ereignisse zu erkennen.

Rückgabe von Werten an Anrufer

In vielen Fällen sind ereignisbasierte Anwendungen asynchron. Das bedeutet, dass Anruferdienste nicht auf Anfragen von anderen Diensten warten, bevor sie mit anderen Aufgaben fortfahren. Dies ist eine grundlegende Eigenschaft ereignisgesteuerter Architekturen, die Skalierbarkeit und Flexibilität ermöglicht. Dies bedeutet, dass die Übergabe von Rückgabewerten oder des Ergebnisses eines Workflows komplexer ist als bei synchronen Ausführungsabläufen.

Die meisten Lambda-Aufrufe in Produktionssystemen sind asynchron und reagieren auf Ereignisse von Diensten wie Amazon S3 oder Amazon SQS. In diesen Fällen ist der Erfolg oder Misserfolg der Verarbeitung eines Ereignisses oft wichtiger als die Rückgabe eines Wertes. Funktionen wie Warteschlangen (DLQs) in Lambda sorgen dafür, dass Sie fehlgeschlagene Ereignisse identifizieren und erneut versuchen können, ohne den Anrufer benachrichtigen zu müssen.

Dienst- und funktionsübergreifendes Debugging

Die Fehlersuche in ereignisgesteuerten Systemen unterscheidet sich ebenfalls von einer monolithischen Anwendung. Da verschiedene Systeme und Services Ereignisse weitergeben, ist es nicht möglich, den genauen Zustand mehrerer Services beim Auftreten von Fehlern zu erfassen und

zu reproduzieren. Da jeder Dienst- und Funktionsaufruf über separate Protokolldateien verfügt, kann es komplizierter sein, festzustellen, was mit einem bestimmten Ereignis passiert ist, das einen Fehler verursacht hat.

Es gibt drei wichtige Voraussetzungen für den Aufbau eines erfolgreichen Debugging-Ansatzes in ereignisgesteuerten Systemen. Erstens ist ein robustes Protokollierungssystem von entscheidender Bedeutung. Dieses wird AWS dienstübergreifend bereitgestellt und von Amazon CloudWatch in Lambda-Funktionen eingebettet. Zweitens muss in diesen Systemen sichergestellt werden, dass jedes Ereignis eine Transaktionskennung hat, die bei jedem Schritt während einer Transaktion protokolliert wird, um die Suche nach Protokollen zu erleichtern.

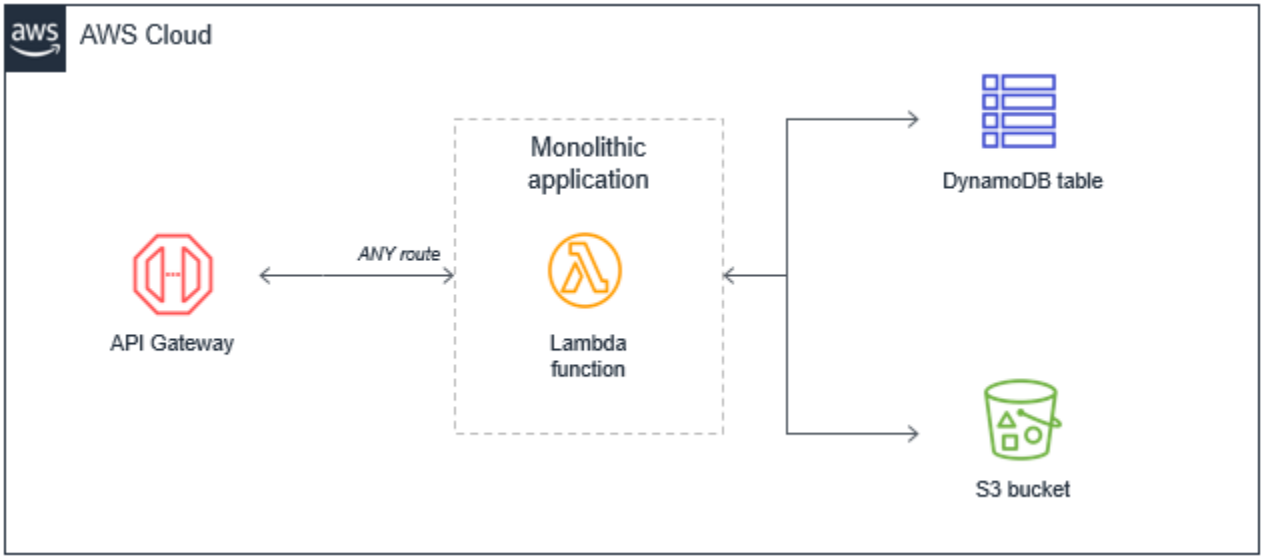
Schließlich ist es sehr empfehlenswert, das Parsing und die Analyse von Protokollen mit Hilfe eines Debugging- und Überwachungsdienstes wie AWS X-Ray zu automatisieren. Dies kann Protokolle über mehrere Lambda-Aufrufe und -Dienste hinweg verbrauchen, wodurch es viel einfacher wird, die Ursache von Problemen zu ermitteln. Eine ausführliche Beschreibung der Verwendung von X-Ray zur Fehlerbehebung finden Sie unter Fehlerbehebungs-Walkthrough.

Anti-Pattern in Lambda-basierten ereignisgesteuerten Anwendungen

Vermeiden Sie beim Aufbau ereignisgesteuerter Architekturen mit Lambda die folgenden gängigen Anti-Patterns. Diese Muster funktionieren, können aber die Kosten und die Komplexität erhöhen.

Der Lambda-Monolith

In vielen Anwendungen, die von herkömmlichen Servern migriert wurden, wie z. B. EC2 Amazon-Instances oder Elastic Beanstalk Beanstalk-Anwendungen, müssen Entwickler vorhandenen Code „nach Belieben verschieben“. Dies führt häufig zu einer einzigen Lambda-Funktion, die die gesamte Anwendungslogik enthält, die für alle Ereignisse ausgelöst wird. Bei einer einfachen Webanwendung würde eine monolithische Lambda-Funktion alle API-Gateway-Routen verwalten und mit allen erforderlichen nachgelagerten Ressourcen integrieren.

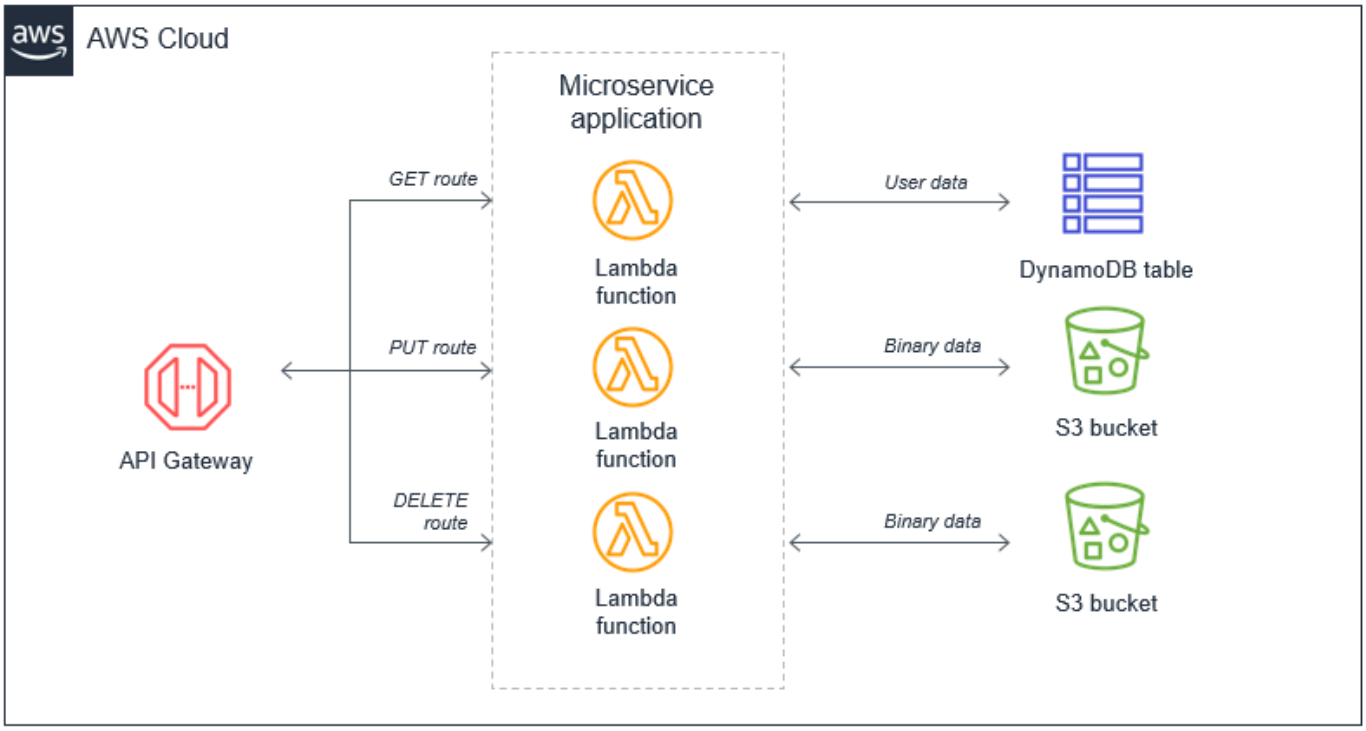


Dieser Ansatz hat mehrere Nachteile:

- Paketgröße — Die Lambda-Funktion ist möglicherweise viel größer, da sie den gesamten möglichen Code für alle Pfade enthält, wodurch die Ausführung des Lambda-Dienstes langsamer wird.
- Es ist schwierig, die geringste Berechtigung durchzusetzen: Die Ausführungsrolle der Funktion muss Berechtigungen für alle Ressourcen gewähren, die für alle Pfade benötigt werden, sodass die Berechtigungen sehr weit gefasst sind. Dies ist ein Sicherheitsproblem. Viele Pfade im funktionalen Monolithen benötigen nicht alle erteilten Berechtigungen.
- Schwieriger zu aktualisieren: In einem Produktionssystem sind Upgrades für eine einzelne Funktion riskanter und können dazu führen, dass die gesamte Anwendung nicht mehr funktioniert. Das Upgrade eines einzelnen Pfads in der Lambda-Funktion ist ein Upgrade der gesamten Funktion.
- Schwieriger zu warten: Es ist schwieriger, mehrere Entwickler an dem Dienst arbeiten zu lassen, da es sich um ein monolithisches Code-Repository handelt. Es erhöht auch die kognitive Belastung der Entwickler und erschwert es, eine angemessene Testabdeckung für Code zu schaffen.
- Schwieriger, Code wiederzuverwenden: Es kann schwieriger sein, wiederverwendbare Bibliotheken von Monolithen zu trennen, was die Wiederverwendung von Code erschwert. Je mehr Projekte Sie entwickeln und unterstützen, desto schwieriger wird es, den Code zu unterstützen und die Geschwindigkeit Ihres Teams zu erhöhen.
- Schwieriger zu testen: Mit zunehmender Anzahl von Codezeilen wird es immer schwieriger, für alle möglichen Kombinationen von Eingaben und Einstiegspunkten in der Codebasis einen Modultest

durchzuführen. Es ist generell einfacher, Modultests für kleinere Dienste mit weniger Code zu implementieren.

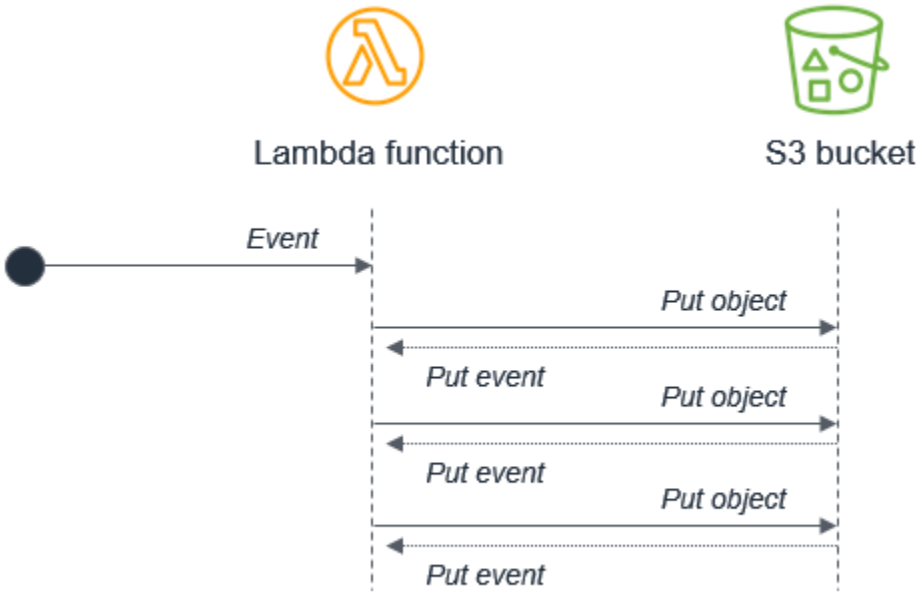
Die bevorzugte Alternative besteht darin, die monolithische Lambda-Funktion in einzelne Microservices aufzuteilen und eine einzelne Lambda-Funktion einer einzigen, klar definierten Aufgabe zuzuordnen. In dieser einfachen Webanwendung mit einigen API-Endpunkten kann die resultierende Microservice-basierte Architektur auf den API-Gateway-Routen basieren.



Rekursive Muster, die außer Kontrolle geratene Lambda-Funktionen verursachen

AWS Dienste generieren Ereignisse, die Lambda-Funktionen aufrufen, und Lambda-Funktionen können Nachrichten an Dienste senden. AWS Im Allgemeinen sollte sich der Dienst oder die Ressource, die eine Lambda-Funktion aufruft, von dem Dienst oder der Ressource unterscheiden, an den die Funktion ausgibt. Wenn dies nicht verwaltet wird, kann dies zu Endlosschleifen führen.

Zum Beispiel schreibt eine Lambda-Funktion ein Objekt in ein Amazon S3-Objekt, das wiederum dieselbe Lambda-Funktion über ein Put-Ereignis aufruft. Durch den Aufruf wird ein zweites Objekt in den Bucket geschrieben, das dieselbe Lambda-Funktion aufruft:



Während das Potenzial für Endlosschleifen in den meisten Programmiersprachen vorhanden ist, hat dieses Anti-Pattern das Potenzial, mehr Ressourcen in Serverless-Anwendungen zu verbrauchen. Sowohl Lambda als auch Amazon S3 skalieren automatisch auf der Grundlage des Datenverkehrs, sodass die Schleife dazu führen kann, dass Lambda skaliert, um die gesamte verfügbare Parallelität zu nutzen, und Amazon S3 schreibt weiterhin Objekte und generiert mehr Ereignisse für Lambda.

In diesem Beispiel wird S3 verwendet, aber das Risiko von rekursiven Schleifen besteht auch bei Amazon SNS, Amazon SQS, DynamoDB und anderen Diensten. Sie können die rekursive Schleifenerkennung verwenden, um dieses Anti-Pattern zu finden und zu vermeiden.

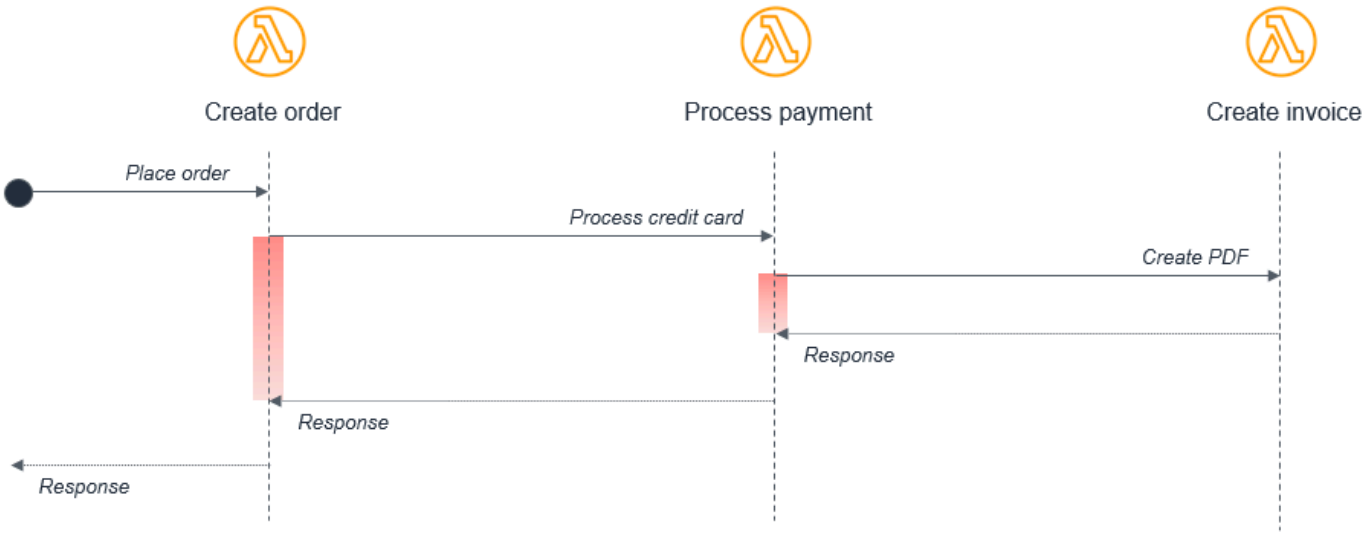
Lambda-Funktionen, die Lambda-Funktionen aufrufen

Funktionen ermöglichen die Kapselung und die Wiederverwendung von Code. Die meisten Programmiersprachen unterstützen das Konzept des synchronen Aufrufs von Funktionen innerhalb einer Codebasis. Wenn die Funktion einen Fehler zurückgibt, gibt die Funktion eine Antwort zurück.

> **Note**
>
> Während Lambda-Funktionen, die direkt andere Lambda-Funktionen aufrufen, aus Kosten- und Komplexitätsgründen im Allgemeinen ein Anti-Pattern darstellen, gilt dies nicht für langlebige Funktionen, die speziell dafür konzipiert sind, mehrstufige Workflows zu orchestrieren, indem sie andere Funktionen aufrufen.

Wenn dies auf einem herkömmlichen Server oder einer virtuellen Instance geschieht, wechselt der Scheduler des Betriebssystems zu einer anderen verfügbaren Arbeit. Ob die CPU zu 0 % oder zu 100 % läuft, hat keinen Einfluss auf die Gesamtkosten der Anwendung, da Sie für die Fixkosten des Besitzes und des Betriebs eines Servers zahlen.

Dieses Modell eignet sich häufig nicht gut für die Serverless-Entwicklung. Nehmen wir zum Beispiel eine einfache E-Commerce-Anwendung, die aus drei Lambda-Funktionen besteht, die eine Bestellung verarbeiten:



In diesem Fall ruft die Funktion Bestellung erstellen die Funktion Zahlung bearbeiten auf, die wiederum die Funktion Rechnung erstellen aufruft. Dieser synchrone Ablauf kann zwar innerhalb einer einzelnen Anwendung auf einem Server funktionieren, führt aber in einer verteilten serverlosen Architektur zu mehreren vermeidbaren Problemen:

- Kosten: Bei Lambda zahlen Sie für die Dauer eines Aufrufs. In diesem Beispiel laufen während der Ausführung der Funktion Rechnung erstellen zwei weitere Funktionen in einem Wartezustand, der im Diagramm in Rot dargestellt ist.
- Fehlerbehandlung: Bei verschachtelten Aufrufen kann die Fehlerbehandlung viel komplexer werden. Ein Fehler in Rechnung erstellen kann beispielsweise dazu führen, dass die Funktion „Zahlung verarbeiten“ die Belastung rückgängig macht, oder es wird stattdessen der Vorgang „Rechnung erstellen“ erneut versucht.
- Enge Verknüpfung: Die Bearbeitung einer Zahlung dauert in der Regel länger als die Erstellung einer Rechnung. Bei diesem Modell wird die Verfügbarkeit des gesamten Workflows durch die langsamste Funktion eingeschränkt.

- Skalierung: Die Gleichzeitigkeit aller drei Funktionen muss gleich sein. In einem stark frequentierten System wird dadurch mehr Gleichzeitigkeit verwendet, als andernfalls erforderlich wäre.

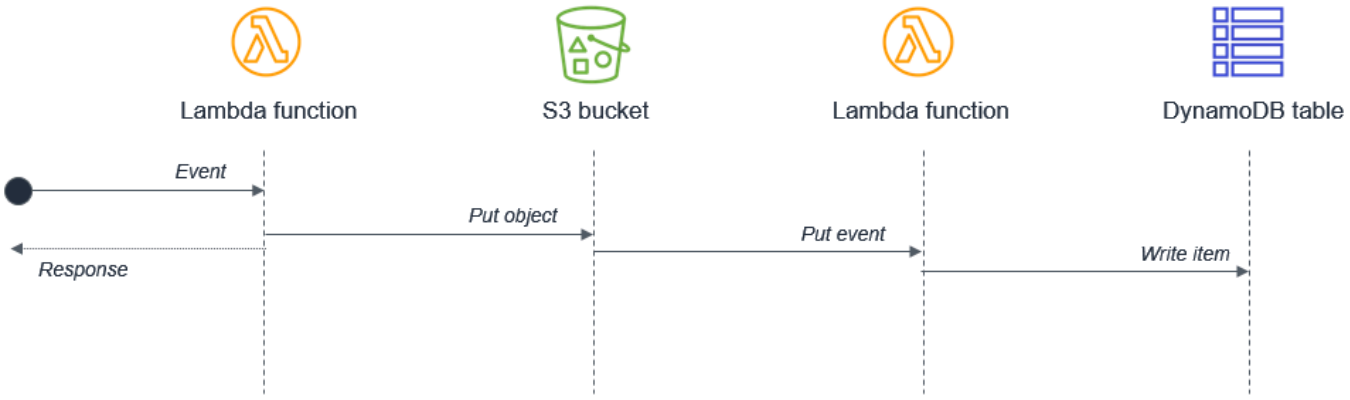
Bei Serverless-Anwendungen gibt es zwei gängige Ansätze, um dieses Muster zu vermeiden. Verwenden Sie zunächst eine Amazon-SQS-Warteschlange zwischen Lambda-Funktionen. Wenn ein nachgeschalteter Prozess langsamer ist als ein vorgeschalteter Prozess, hält die Warteschlange die Nachrichten dauerhaft fest und entkoppelt die beiden Funktionen. In diesem Beispiel würde die Funktion Bestellung erstellen eine Nachricht in einer Amazon SQS SQS-Warteschlange veröffentlichen, und die Funktion Zahlung verarbeiten verarbeitet Nachrichten aus der Warteschlange.

Der zweite Ansatz besteht darin, AWS Step Functions zu verwenden. Bei komplexen Prozessen mit mehreren Arten von Fehlern und Wiederholungslogik können Schrittfunktionen dazu beitragen, den Umfang des benutzerdefinierten Codes zu reduzieren, der für die Orchestrierung des Workflows erforderlich ist. Infolgedessen orchestriert Step Functions die Arbeit und behandelt Fehler und Wiederholungen zuverlässig und die Lambda-Funktionen enthalten nur Geschäftslogik.

Synchrones Warten innerhalb einer einzigen Lambda-Funktion

Stellen Sie sicher, dass potenziell gleichzeitige Aktivitäten nicht synchron innerhalb einer einzelnen Lambda-Funktion geplant werden. Eine Lambda-Funktion könnte beispielsweise in einen S3-Bucket und dann in eine DynamoDB-Tabelle schreiben:

In diesem Design sind die Wartezeiten zusammengesetzt, weil die Aktivitäten sequentiell sind. In Fällen, in denen die zweite Aufgabe von der Fertigstellung der ersten Aufgabe abhängt, können Sie die Gesamtwartezeit und die Ausführungskosten verringern, indem Sie zwei separate Lambda-Funktionen verwenden:



In diesem Design reagiert die erste Lambda-Funktion sofort, nachdem das Objekt in den Amazon-S3-Bucket gestellt wurde. Der S3-Dienst ruft die zweite Lambda-Funktion auf, die dann Daten in die DynamoDB-Tabelle schreibt. Dieser Ansatz minimiert die Gesamtwartezeit bei der Ausführung von Lambda-Funktionen.

Entwerfen von Lambda-Anwendungen

Eine gut konzipierte ereignisgesteuerte Anwendung verwendet eine Kombination aus AWS Diensten und benutzerdefiniertem Code, um Anfragen und Daten zu verarbeiten und zu verwalten. Dieses Kapitel konzentriert sich auf Lambda-spezifische Themen im Anwendungsdesign. Bei der Entwicklung von Anwendungen für stark frequentierte Produktionssysteme gibt es viele wichtige Überlegungen für Serverless-Architekten.

Viele der bewährten Verfahren, die für die Softwareentwicklung und verteilte Systeme gelten, gelten auch für die Entwicklung Serverless-Anwendungen. Das übergeordnete Ziel besteht darin, Workloads zu entwickeln, die:

- zuverlässig sind: Sie bieten Ihren Endnutzern ein hohes Maß an Verfügbarkeit. AWS -Serverless-Dienste sind zuverlässig, weil sie auch für Ausfälle ausgelegt sind.
- langlebig sind: Sie bieten Speicheroptionen, die den Anforderungen an die Haltbarkeit Ihrer Workloads gerecht werden.
- sicher sind: Befolgen Sie bewährte Verfahren und verwenden Sie die bereitgestellten Tools, um den Zugriff auf Workloads zu sichern und den Explosionsradius zu begrenzen.
- leistungsstark sind: effiziente Nutzung von Rechenressourcen und Erfüllung der Leistungsanforderungen Ihrer Endbenutzer.
- kosteneffizient sind: Entwicklung von Architekturen zur Vermeidung unnötiger Kosten, die ohne Mehrausgaben skaliert und auch ohne nennenswerten Mehraufwand außer Betrieb genommen werden können.

Die folgenden Designprinzipien können Ihnen dabei helfen, Workloads zu erstellen, die diese Ziele erfüllen. Nicht jedes Prinzip lässt sich auf jede Architektur anwenden, aber sie sollten Ihnen bei allgemeinen Architekturentscheidungen als Leitfaden dienen.

Themen

- Verwendung von Diensten anstelle von benutzerdefiniertem Code
- Informationen zu Lambda-Abstraktionsebenen
- Implementieren der Zustandslosigkeit in Funktionen
- Kopplung minimieren
- Entwickeln für On-Demand-Daten statt Batches
- Erwägen AWS Step Functions Sie die Orchestrierung

- Implementieren von Idempotenz
- Verwenden Sie mehrere Konten für die Verwaltung von Kontingenten AWS

Verwendung von Diensten anstelle von benutzerdefiniertem Code

Serverlose Anwendungen bestehen normalerweise aus mehreren AWS Diensten, die mit benutzerdefiniertem Code integriert sind, der in Lambda-Funktionen ausgeführt wird. Lambda kann zwar in die meisten AWS Dienste integriert werden, aber die Dienste, die in serverlosen Anwendungen am häufigsten verwendet werden, sind:

Kategorie	AWS Dienst
Datenverarbeitung	AWS Lambda
Datenspeicher	Amazon S3 Amazon DynamoDB Amazon RDS
API	Amazon API Gateway
Integration von Anwendungen	Amazon EventBridge Amazon SNS Amazon SQS
Orchestrierung	AWS Step Functions
Streaming-Daten und Analysen	Amazon Data Firehose

> **Note**
>
> Viele Serverless-Dienste bieten Replikation und Support für mehrere Regionen, einschließlich DynamoDB und Amazon S3. Lambda-Funktionen können als Teil einer Bereitstellungspipeline in mehreren Regionen bereitgestellt werden und API Gateway

kann so konfiguriert werden, dass es diese Konfiguration unterstützt. Sehen Sie sich diese Beispielarchitektur an, die zeigt, wie dies erreicht werden kann.

Es gibt viele etablierte, gängige Muster in verteilten Architekturen, die Sie selbst erstellen oder mithilfe von AWS Services implementieren können. Für die meisten Kunden ist es wirtschaftlich nicht sinnvoll, Zeit in die Entwicklung dieser Muster zu investieren. Wenn Ihre Anwendung eines dieser Muster benötigt, verwenden Sie den entsprechenden AWS Dienst:

Muster	AWS Dienst
Warteschlange	Amazon SQS
Ereignisbus	Amazon EventBridge
Veröffentlichen/Abonnieren (Fan-Out)	Amazon SNS
Orchestrierung	AWS Step Functions
API	Amazon API Gateway
Ereignis-Streams	Amazon Kinesis

Diese Dienste sind für die Integration mit Lambda konzipiert und Sie können Infrastructure as Code (IaC) verwenden, um Ressourcen in den Diensten zu erstellen und zu verwerfen. Sie können jeden dieser Dienste über das AWS SDK nutzen, ohne dass Sie Anwendungen installieren oder Server konfigurieren müssen. Der Umgang mit diesen Diensten über den Code in Ihren Lambda-Funktionen ist ein wichtiger Schritt zur Erstellung gut konzipierter Serverless-Anwendungen.

Informationen zu Lambda-Abstraktionsebenen

Der Lambda-Dienst beschränkt Ihren Zugriff auf die zugrunde liegenden Betriebssysteme, Hypervisoren und Hardware, auf denen Ihre Lambda-Funktionen ausgeführt werden. Der Dienst wird ständig verbessert und die Infrastruktur verändert, um neue Features hinzuzufügen, die Kosten zu senken und den Dienst leistungsfähiger zu machen. Ihr Code sollte kein Wissen darüber voraussetzen, wie Lambda aufgebaut ist und keine Hardware-Affinität voraussetzen.

In ähnlicher Weise werden die Integrationen von Lambda mit anderen Diensten von verwaltet AWS, sodass Ihnen nur eine geringe Anzahl von Konfigurationsoptionen zur Verfügung steht.

Bei der Interaktion zwischen API Gateway und Lambda gibt es beispielsweise kein Konzept für den Load Balancer, da dieser vollständig von den Diensten verwaltet wird. Sie haben auch keine direkte Kontrolle darüber, welche Verfügbarkeitszonen die Dienste beim Aufrufen von Funktionen zu einem bestimmten Zeitpunkt verwenden oder wie Lambda entscheidet, wann die Anzahl der Ausführungsumgebungen nach oben oder unten skaliert wird.

Diese Abstraktion ermöglicht es Ihnen, sich auf die Integrationsaspekte Ihrer Anwendung, den Datenfluss und die Geschäftslogik zu konzentrieren, bei denen Ihr Workload einen Mehrwert für Ihre Endbenutzer bietet. Wenn Sie den Diensten die Verwaltung der zugrundeliegenden Mechanismen überlassen, können Sie Anwendungen schneller entwickeln und müssen weniger eigenen Code pflegen.

Implementieren der Zustandslosigkeit in Funktionen

Bei Standard-Lambda-Funktionen sollten Sie davon ausgehen, dass die Umgebung nur für einen einzigen Aufruf existiert. Die Funktion sollte beim ersten Start alle erforderlichen Zustände initialisieren. Beispielsweise kann Ihre Funktion das Abrufen von Daten aus einer DynamoDB-Tabelle erfordern. Sie sollte alle dauerhaften Datenänderungen vor dem Beenden in einem dauerhaften Speicher wie Amazon S3, DynamoDB oder Amazon SQS speichern. Sie sollte sich nicht auf bestehende Datenstrukturen oder temporäre Dateien oder auf interne Zustände stützen, die bei mehreren Aufrufen verwaltet würden.

Bei der Verwendung von Durable Functions wird der Status zwischen Aufrufen automatisch beibehalten, sodass der Status nicht manuell im externen Speicher gespeichert werden muss. Für alle Daten, die nicht explizit über den verwaltet werden, sollten Sie jedoch weiterhin die Prinzipien der Zustandslosigkeit beachten. DurableContext

Um Datenbankverbindungen und Bibliotheken zu initialisieren oder den Status zu laden, können Sie die statische Initialisierung nutzen. Da Ausführungsumgebungen nach Möglichkeit wiederverwendet werden, um die Leistung zu verbessern, können Sie die für die Initialisierung dieser Ressourcen benötigte Zeit über mehrere Aufrufe hinweg amortisieren. Sie sollten jedoch keine Variablen oder Daten, die in der Funktion verwendet werden, in diesem globalen Bereich speichern.

Kopplung minimieren

Die meisten Architekturen sollten viele, kürzere Funktionen den wenigen, größeren vorziehen. Der Zweck jeder Funktion sollte darin bestehen, das an die Funktion übergebene Ereignis zu verarbeiten, ohne Kenntnisse oder Erwartungen in Bezug auf den gesamten Arbeitsablauf oder das

Transaktionsvolumen. Dadurch ist die Funktion unabhängig von der Quelle des Ereignisses und nur minimal mit anderen Diensten verbunden.

Alle Konstanten mit globalem Geltungsbereich, die sich nur selten ändern, sollten als Umgebungsvariablen implementiert werden, um Aktualisierungen ohne Bereitstellungen zu ermöglichen. Alle Geheimnisse oder sensiblen Informationen sollten im AWS Systems Manager Parameter Store oder AWS Secrets Manager gespeichert und von der Funktion geladen werden. Da diese Ressourcen kontospezifisch sind, können Sie auf diese Weise Build-Pipelines für mehrere Konten erstellen. Die Pipelines laden die entsprechenden Geheimnisse für jede Umgebung, ohne dass diese für Entwickler sichtbar sind oder Codeänderungen erforderlich sind.

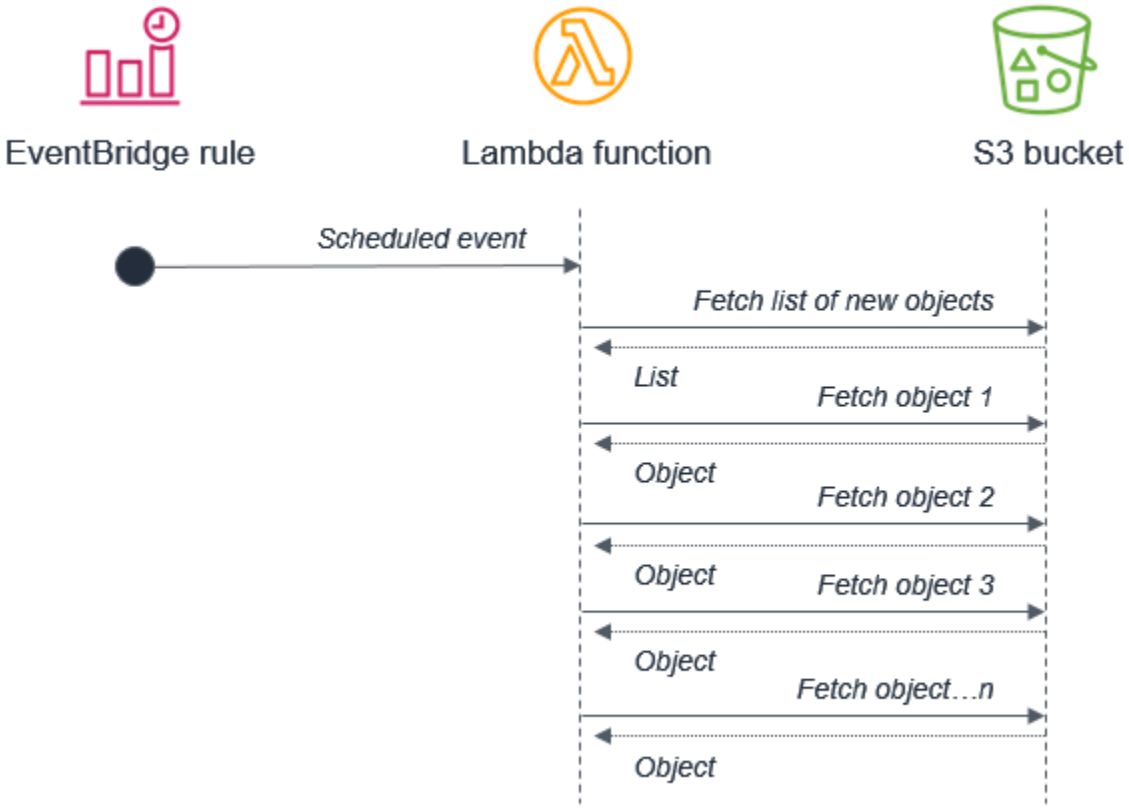
Entwickeln für On-Demand-Daten statt Batches

Viele herkömmliche Systeme sind so konzipiert, dass sie periodisch laufen und Batch von Transaktionen verarbeiten, die sich im Laufe der Zeit angesammelt haben. So kann beispielsweise eine Bankanwendung stündlich laufen, um die Transaktionen von Geldautomaten in einem zentralen Hauptbuch zu verarbeiten. In Lambda-basierten Anwendungen sollte die benutzerdefinierte Verarbeitung durch jedes Ereignis ausgelöst werden, so dass der Dienst bei Bedarf die Gleichzeitigkeit erhöhen kann, um die Verarbeitung von Transaktionen nahezu in Echtzeit zu ermöglichen.

Während Standard-Lambda-Funktionen auf eine Ausführungszeit von 15 Minuten begrenzt sind, können Durable Functions bis zu einem Jahr lang ausgeführt werden, sodass sie für länger andauernde Verarbeitungsanforderungen geeignet sind. Dennoch sollten Sie, wenn möglich, die ereignisgesteuerte Verarbeitung der Stapelverarbeitung vorziehen.

Sie können Cron-Aufgaben zwar in serverlosen Anwendungen ausführen, indem Sie geplante Ausdrücke für Regeln in Amazon verwenden EventBridge, diese sollten jedoch sparsam oder als letztes Mittel verwendet werden. Bei jeder geplanten Aufgabe, die einen Batch verarbeitet, besteht die Möglichkeit, dass das Transaktionsvolumen über das hinausgeht, was innerhalb des 15-minütigen Lambda-Dauerbegrenzung verarbeitet werden kann. Wenn Sie aufgrund der Einschränkungen externer Systeme gezwungen sind, einen Scheduler zu verwenden, sollten Sie in der Regel für den kürzesten sinnvollen wiederkehrenden Zeitraum planen.

So ist es beispielsweise keine optimale Vorgehensweise, einen Batch-Prozess zu verwenden, der eine Lambda-Funktion auslöst, um eine Liste neuer Amazon-S3-Objekte abzurufen. Dies liegt daran, dass der Dienst zwischen den Batches möglicherweise mehr neue Objekte erhält, als innerhalb einer 15-minütigen Lambda-Funktion verarbeitet werden können.



Stattdessen sollte Amazon S3 die Lambda-Funktion jedes Mal aufrufen, wenn ein neues Objekt in den Bucket gestellt wird. Dieser Ansatz ist deutlich skalierbarer und funktioniert nahezu in Echtzeit.

Erwägen AWS Step Functions Sie die Orchestrierung

Workflows, die Verzweigungslogik, verschiedene Arten von Fehlermodellen und Wiederholungslogik beinhalten, verwenden in der Regel einen Orchestrator, um den Status der Gesamtausführung zu verfolgen. Vermeiden Sie die Verwendung von Lambda-Funktionen für diesen Zweck, da dies zu einer engen Verknüpfung und einer komplexen Codeverarbeitung führt.

Mit AWS Step Functions verwenden Sie Zustandsmaschinen, um die Orchestrierung zu verwalten. Dadurch werden die Fehlerbehandlung, das Routing und die Verzweigungslogik aus Ihrem Code extrahiert und durch Zustandsautomaten ersetzt, die mit JSON deklariert werden. Abgesehen davon,

dass es Workflows robuster und beobachtbarer macht, ermöglicht es Ihnen auch, Workflows zu versionieren und den Zustandsautomaten zu einer kodifizierten Ressource zu machen, die Sie zu einem Code-Repository hinzufügen können.

Es ist üblich, dass einfachere Workflows in Lambda-Funktionen mit der Zeit komplexer werden. Beim Betrieb einer produktiven Serverless-Anwendung ist es wichtig zu erkennen, wann dies geschieht, damit Sie diese Logik in einen Zustandsautomaten migrieren können.

Implementieren von Idempotenz

AWS Serverlose Dienste, einschließlich Lambda, sind fehlertolerant und für den Umgang mit Ausfällen konzipiert. Wenn beispielsweise ein Dienst eine Lambda-Funktion aufruft und es zu einer Dienstunterbrechung kommt, ruft Lambda Ihre Funktion in einer anderen Verfügbarkeitszone auf. Wenn Ihre Funktion einen Fehler auslöst, wiederholt Lambda den Aufruf.

Da ein und dasselbe Ereignis mehr als einmal eintreten kann, sollten die Funktionen idempotent sein. Dies bedeutet, dass der mehrfache Empfang desselben Ereignisses das Ergebnis nicht über den ersten Empfang hinaus verändert.

Sie können Idempotenz in Lambda-Funktionen implementieren, indem Sie eine DynamoDB-Tabelle verwenden, um kürzlich verarbeitete Bezeichner zu verfolgen und festzustellen, ob die Transaktion bereits zuvor bearbeitet wurde. Die DynamoDB-Tabelle implementiert in der Regel einen TTL-Wert (Time To Live) für ablaufende Elemente, um den verwendeten Speicherplatz zu begrenzen.

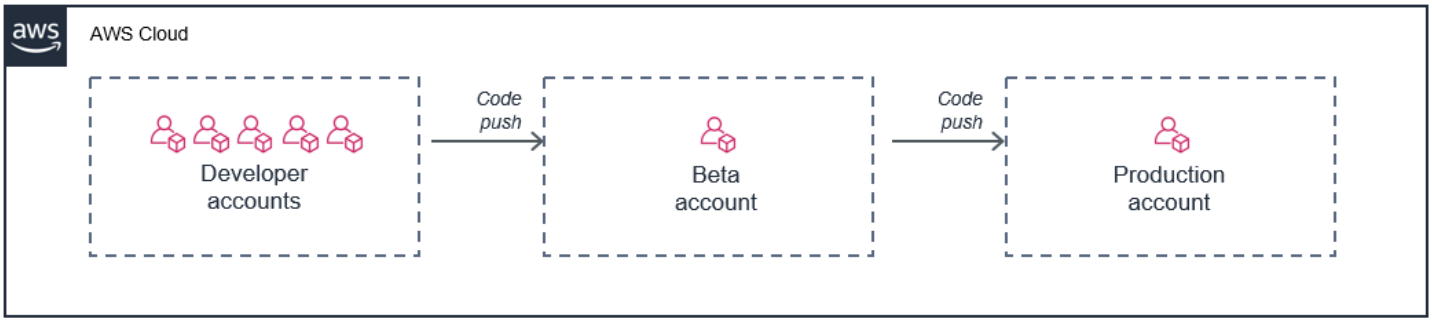
Verwenden Sie mehrere Konten für die Verwaltung von Kontingenten AWS

Viele Servicekontingente AWS werden auf Kontoebene festgelegt. Das bedeutet, dass Sie bei zunehmendem Workload schnell an Ihre Grenzen stoßen können.

Eine effektive Möglichkeit, dieses Problem zu lösen, besteht darin, mehrere AWS Konten zu verwenden und jede Arbeitslast einem eigenen Konto zuzuweisen. Dadurch wird verhindert, dass Kontingente mit anderen Workloads oder Ressourcen außerhalb der Produktion geteilt werden.

Mithilfe von AWS -Organisationen können Sie außerdem die Abrechnung, Compliance und Sicherheit dieser Konten zentral verwalten. Sie können Richtlinien an Gruppen von Konten anhängen, um benutzerdefinierte Skripts und manuelle Prozesse zu vermeiden.

Ein gängiger Ansatz besteht darin, jedem Entwickler ein AWS Konto zur Verfügung zu stellen und dann separate Konten für die Betabereitstellungsphase und die Produktion zu verwenden:



In diesem Modell hat jeder Entwickler seine eigenen Beschränkungen für das Konto, sodass sich deren Nutzung nicht auf Ihre Produktionsumgebung auswirkt. Dieser Ansatz ermöglicht es Entwicklern auch, Lambda-Funktionen lokal auf ihren Entwicklungsmaschinen anhand von Live-Cloud-Ressourcen in ihren individuellen Konten zu testen.

Erstellen Sie Ihre erste Lambda-Funktion

für Ihre ersten Schritte mit Lambda verwenden Sie die Lambda-Konsole, um eine Funktion zu erstellen. In wenigen Minuten können Sie eine Funktion erstellen und bereitstellen und sie in der Konsole testen.

Bei der Durchführung des Tutorials lernen Sie einige grundlegende Lambda-Konzepte kennen, z. B. wie Sie Argumente an Ihre Funktion mithilfe des Lambda-Ereignisobjekts übergeben. Sie erfahren auch, wie Sie Protokollausgaben von Ihrer Funktion zurückgeben und wie Sie die Aufruf-Logs Ihrer Funktion in Amazon CloudWatch Logs einsehen können.

Der Einfachheit halber erstellen Sie Ihre Funktion entweder mit der Python- oder Node.js-Laufzeit. Mit diesen interpretierten Sprachen können Sie Funktionscode direkt im integrierten Code-Editor der Konsole bearbeiten. Bei kompilierten Sprachen wie Java und C# müssen Sie ein Bereitstellungspaket auf Ihrem lokalen Build-Rechner erstellen und es in Lambda hochladen. Weitere Informationen zum Bereitstellen von Funktionen in Lambda mithilfe anderer Laufzeiten finden Sie unter den Links im Abschnitt the section called “Nächste Schritte”.

> **Tip**
>
> Weitere Informationen zum Erstellen von Serverless-Lösungen finden Sie im Serverless-Benutzerhandbuch.

Voraussetzungen

Melden Sie sich an für ein AWS-Konto

Wenn Sie noch keine haben AWS-Konto, führen Sie die folgenden Schritte aus, um eine zu erstellen.

Um sich für eine anzumelden AWS-Konto

1. Öffnen Sie https://portal.aws.amazon.com/billing/die Anmeldung.
2. Folgen Sie den Online-Anweisungen.

   Während der Anmeldung erhalten Sie einen Telefonanruf oder eine Textnachricht und müssen einen Verifizierungscode über die Telefontasten eingeben.

Wenn Sie sich für eine anmelden AWS-Konto, Root-Benutzer des AWS-Kontoswird eine erstellt. Der Root-Benutzer hat Zugriff auf alle AWS-Services und Ressourcen des Kontos. Als bewährte Sicherheitsmethode weisen Sie einem Benutzer Administratorzugriff zu und verwenden Sie nur den Root-Benutzer, um Aufgaben auszuführen, die Root-Benutzerzugriff erfordern.

AWS sendet Ihnen nach Abschluss des Anmeldevorgangs eine Bestätigungs-E-Mail. Sie können Ihre aktuellen Kontoaktivitäten jederzeit einsehen und Ihr Konto verwalten, indem Sie zu https://aws.amazon.com/gehen und Mein Konto auswählen.

Erstellen eines Benutzers mit Administratorzugriff

Nachdem Sie sich für einen angemeldet haben AWS-Konto, sichern Sie Ihren Root-Benutzer des AWS-Kontos AWS IAM Identity Center, aktivieren und erstellen Sie einen Administratorbenutzer, sodass Sie den Root-Benutzer nicht für alltägliche Aufgaben verwenden.

Sichern Sie Ihre Root-Benutzer des AWS-Kontos

1. Melden Sie sich AWS-Managementkonsoleals Kontoinhaber an, indem Sie Root-Benutzer auswählen und Ihre AWS-Konto E-Mail-Adresse eingeben. Geben Sie auf der nächsten Seite Ihr Passwort ein.

   Hilfe bei der Anmeldung mit dem Root-Benutzer finden Sie unter Anmelden als Root-Benutzer im AWS-Anmeldung Benutzerhandbuch zu.
2. Aktivieren Sie die Multi-Faktor-Authentifizierung (MFA) für den Root-Benutzer.

   Anweisungen finden Sie unter Aktivieren eines virtuellen MFA-Geräts für Ihren AWS-Konto Root-Benutzer (Konsole) im IAM-Benutzerhandbuch.

Erstellen eines Benutzers mit Administratorzugriff

1. Aktivieren Sie das IAM Identity Center.

   Anweisungen finden Sie unter Aktivieren AWS IAM Identity Center im AWS IAM Identity Center Benutzerhandbuch.
2. Gewähren Sie einem Administratorbenutzer im IAM Identity Center Benutzerzugriff.

Ein Tutorial zur Verwendung von IAM-Identity-Center-Verzeichnis als Identitätsquelle finden Sie IAM-Identity-Center-Verzeichnis im Benutzerhandbuch unter Benutzerzugriff mit der Standardeinstellung konfigurieren.AWS IAM Identity Center

Anmelden als Administratorbenutzer

- Um sich mit Ihrem IAM-Identity-Center-Benutzer anzumelden, verwenden Sie die Anmelde-URL, die an Ihre E-Mail-Adresse gesendet wurde, als Sie den IAM-Identity-Center-Benutzer erstellt haben.

  Hilfe bei der Anmeldung mit einem IAM Identity Center-Benutzer finden Sie im AWS-Anmeldung Benutzerhandbuch unter Anmeldung beim AWS Access-Portal.

Weiteren Benutzern Zugriff zuweisen

1. Erstellen Sie im IAM-Identity-Center einen Berechtigungssatz, der den bewährten Vorgehensweisen für die Anwendung von geringsten Berechtigungen folgt.

   Anweisungen hierzu finden Sie unter Berechtigungssatz erstellen im AWS IAM Identity Center Benutzerhandbuch.
2. Weisen Sie Benutzer einer Gruppe zu und weisen Sie der Gruppe dann Single Sign-On-Zugriff zu.

   Eine genaue Anleitung finden Sie unter Gruppen hinzufügen im AWS IAM Identity Center Benutzerhandbuch.

Erstellen einer Lambda-Funktion mit der Konsole

In diesem Beispiel besitzt Ihre Funktion ein JSON-Objekt, das zwei ganzzahlige Werte mit der Bezeichnung `"length"` und `"width"` enthält. Die Funktion multipliziert diese Werte, um eine Fläche zu berechnen, und gibt diese als JSON-Zeichenfolge zurück.

Ihre Funktion druckt auch den berechneten Bereich zusammen mit dem Namen der zugehörigen CloudWatch Protokollgruppe. Später im Tutorial werden Sie lernen, CloudWatch Logs zu verwenden, um Aufzeichnungen über den Aufruf Ihrer Funktionen einzusehen.

So erstellen Sie eine Hello-World-Lambda-Funktion mit der Konsole

1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.
2. Wählen Sie Funktion erstellen.
3. Wählen Sie Verfassen von Grund auf aus.
4. Geben Sie im Bereich Grundlegende Informationen als Funktionsname **`myLambdaFunction`** ein.
5. Wählen Sie für Runtime entweder Node.js 24 oder Python 3.14.
6. Belassen Sie die Architektur auf x86_64 und wählen Sie Funktion erstellen.

Zusätzlich zu einer einfachen Funktion, die die Nachricht `Hello from Lambda!` zurückgibt, erstellt Lambda auch eine Ausführungsrolle für Ihre Funktion. Eine Ausführungsrolle ist eine AWS Identity and Access Management (IAM-) Rolle, die einer Lambda-Funktion Zugriffsberechtigungen AWS-Services und Ressourcen gewährt. Für Ihre Funktion gewährt die Rolle, die Lambda erstellt, grundlegende Berechtigungen zum Schreiben in CloudWatch Logs.

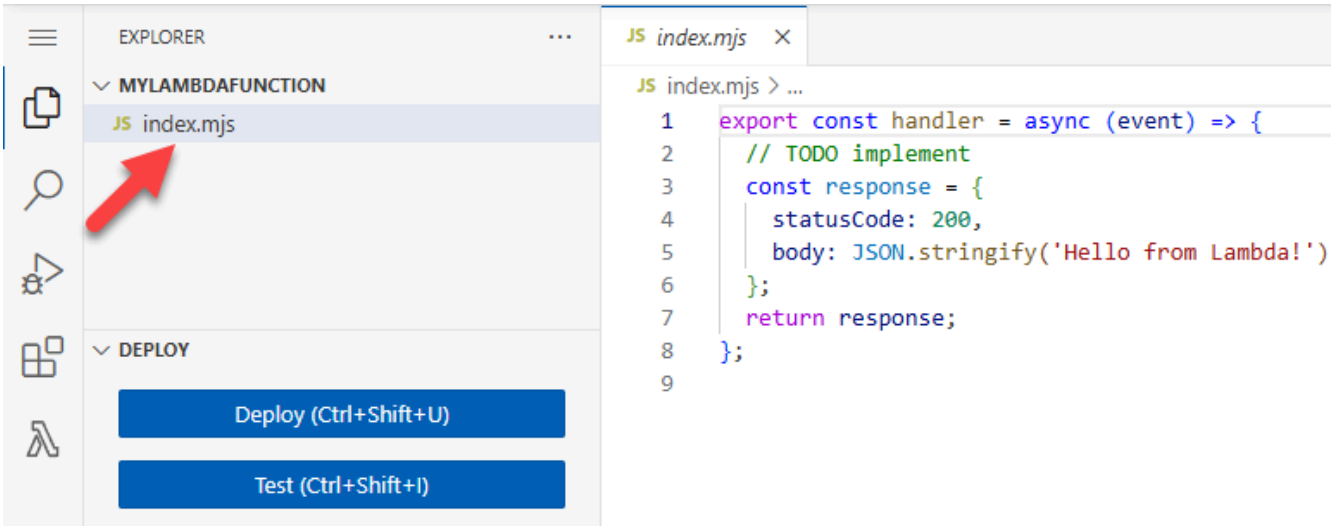
Verwenden Sie den integrierten Code-Editor der Konsole, um den von Lambda erstellten Hello-World-Code durch Ihren eigenen Funktionscode zu ersetzen.

Node.js

So ändern Sie den Code in der Konsole

1. Wählen Sie die Registerkarte Code.

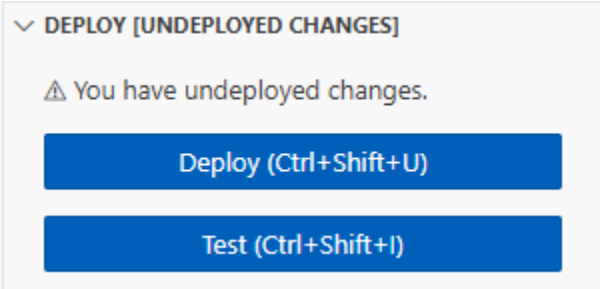
   Im integrierten Code-Editor der Konsole sollten Sie den von Lambda erstellten Funktionscode sehen. Wenn die Registerkarte index.mjs im Code-Editor nicht angezeigt wird, wählen Sie index.mjs im Datei-Explorer aus, wie im folgenden Diagramm gezeigt.



2. Fügen Sie den folgenden Code in die Registerkarte index.mjs ein und ersetzen Sie den von Lambda erstellten Code.

```
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. Wählen Sie im Abschnitt BEREITSTELLEN die Option Bereitstellen aus, um den Code Ihrer Funktion zu aktualisieren:



Den Funktionscode verstehen

Bevor Sie zum nächsten Schritt übergehen, sollten Sie sich einen Moment Zeit nehmen, um den Funktionscode zu betrachten und einige wichtige Lambda-Konzepte zu verstehen.

- Der Lambda-Handler:

  Ihre Lambda-Funktion enthält eine Node.js-Funktion mit dem Namen `handler`. Eine Lambda-Funktion in Node.js kann mehr als eine Node.js-Funktion enthalten, aber die Handler-Funktion ist immer der Einstiegspunkt in Ihren Code. Wenn Ihre Funktion aufgerufen wird, führt Lambda diese Methode aus.

  Wenn Sie Ihre Hello-World-Funktion über die Konsole erstellt haben, hat Lambda den Namen der Handler-Methode für Ihre Funktion automatisch auf `handler` festgelegt. Achten Sie darauf, den Namen dieser Node.js-Funktion nicht zu bearbeiten. Andernfalls kann Lambda Ihren Code nicht ausführen, wenn Sie Ihre Funktion aufrufen.

  Weitere Informationen zum Lambda-Handler in Node.js finden Sie unter the section called “Handler”.
- Das Lambda-Ereignisobjekt:

  Die Funktion `handler` besitzt zwei Argumente, `event` und `context`. Ein Ereignis in Lambda ist ein JSON-formatiertes Dokument, das Daten enthält, die von Ihrer Funktion verarbeitet werden sollen.

  Wenn Ihre Funktion von einer anderen aufgerufen wird AWS-Service, enthält das Ereignisobjekt Informationen über das Ereignis, das den Aufruf verursacht hat. Wenn Ihre Funktion zum Beispiel aufgerufen wird, wenn ein Objekt in einen Amazon Simple Storage Service (Amazon S3) Bucket hochgeladen wird, enthält das Ereignis den Namen des Buckets und den Objektschlüssel.

In diesem Beispiel erstellen Sie ein Ereignis in der Konsole, indem Sie ein JSON-formatiertes Dokument mit zwei Schlüssel-Wert-Paaren eingeben.

- Das Lambda-Kontextobjekt:

  Das zweite Argument, das Ihre Funktion besitzt, ist `context`. Lambda übergibt das Kontextobjekt automatisch an Ihre Funktion. Das Kontextobjekt enthält Informationen über den Aufruf der Funktion und die Ausführungsumgebung.

  Mit dem Kontextobjekt können Sie zu Überwachungszwecken Informationen über den Aufruf Ihrer Funktion ausgeben. In diesem Beispiel verwendet Ihre Funktion den `logGroupName` Parameter, um den Namen ihrer CloudWatch Protokollgruppe auszugeben.

  Weitere Informationen zum Lambda-Kontextobjekt in Node.js finden Sie unter the section called “Kontext”.

- Protokollierung in Lambda:

  Mit Node.js können Sie Konsolenmethoden wie `console.log` und `console.error` verwenden, um Informationen an das Protokoll Ihrer Funktion zu senden. Der Beispielcode verwendet `console.log` Anweisungen, um die berechnete Fläche und den Namen der CloudWatch Logs-Gruppe der Funktion auszugeben. Sie können auch jede Protokollierungsbibliothek verwenden, die in `stdout` oder `stderr` schreibt.

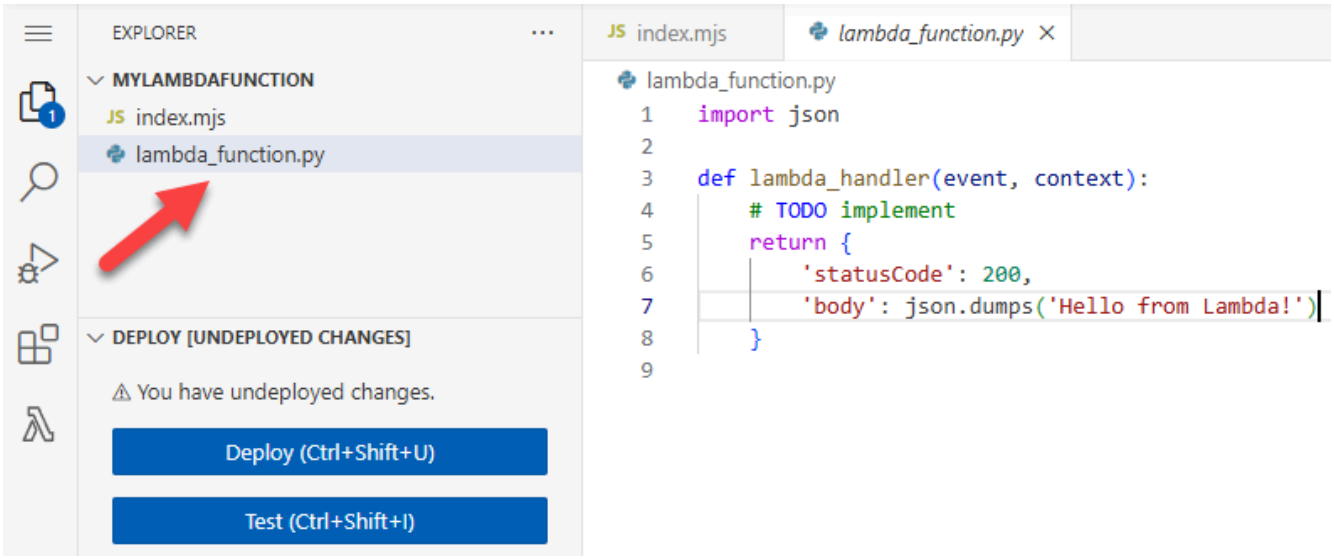
  Weitere Informationen hierzu finden Sie unter the section called “Protokollierung”. Informationen zum Protokollieren in anderen Laufzeiten finden Sie auf den Seiten „Erstellen mit“ für die Laufzeiten, an denen Sie interessiert sind.

Python

So ändern Sie den Code in der Konsole

1. Wählen Sie die Registerkarte Code.

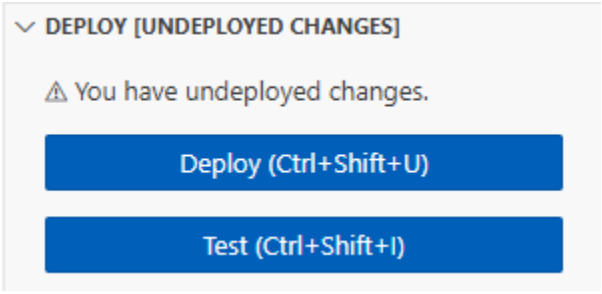
   Im integrierten Code-Editor der Konsole sollten Sie den von Lambda erstellten Funktionscode sehen. Wenn die Registerkarte lambda_function.py im Code-Editor nicht angezeigt wird, wählen Sie lambda_function.py im Datei-Explorer aus, wie im folgenden Diagramm dargestellt.



2. Fügen Sie den folgenden Code in die Registerkarte lambda_function.py ein und ersetzen Sie den von Lambda erstellten Code.

```
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. Wählen Sie im Abschnitt BEREITSTELLEN die Option Bereitstellen aus, um den Code Ihrer Funktion zu aktualisieren:



Den Funktionscode verstehen

Bevor Sie zum nächsten Schritt übergehen, sollten Sie sich einen Moment Zeit nehmen, um den Funktionscode zu betrachten und einige wichtige Lambda-Konzepte zu verstehen.

- Der Lambda-Handler:

  Ihre Lambda-Funktion enthält eine Python-Funktion mit dem Namen `lambda_handler`. Eine Lambda-Funktion in Python kann mehr als eine Python-Funktion enthalten, aber die Handler-Funktion ist immer der Einstiegspunkt in Ihren Code. Wenn Ihre Funktion aufgerufen wird, führt Lambda diese Methode aus.

  Wenn Sie Ihre Hello-World-Funktion über die Konsole erstellt haben, hat Lambda den Namen der Handler-Methode für Ihre Funktion automatisch auf `lambda_handler` festgelegt. Achten Sie darauf, den Namen dieser Python-Funktion nicht zu bearbeiten. Andernfalls kann Lambda Ihren Code nicht ausführen, wenn Sie Ihre Funktion aufrufen.

  Weitere Informationen zum Lambda-Handler in Python finden Sie unter the section called “Handler”.

- Das Lambda-Ereignisobjekt:

  Die Funktion `lambda_handler` besitzt zwei Argumente, `event` und `context`. Ein Ereignis in Lambda ist ein JSON-formatiertes Dokument, das Daten enthält, die von Ihrer Funktion verarbeitet werden sollen.

  Wenn Ihre Funktion von einer anderen aufgerufen wird AWS-Service, enthält das Ereignisobjekt Informationen über das Ereignis, das den Aufruf verursacht hat. Wenn Ihre Funktion zum Beispiel aufgerufen wird, wenn ein Objekt in einen Amazon Simple Storage Service (Amazon

S3) Bucket hochgeladen wird, enthält das Ereignis den Namen des Buckets und den Objektschlüssel.

In diesem Beispiel erstellen Sie ein Ereignis in der Konsole, indem Sie ein JSON-formatiertes Dokument mit zwei Schlüssel-Wert-Paaren eingeben.

- Das Lambda-Kontextobjekt:

  Das zweite Argument, das Ihre Funktion besitzt, ist `context`. Lambda übergibt das Kontextobjekt automatisch an Ihre Funktion. Das Kontextobjekt enthält Informationen über den Aufruf der Funktion und die Ausführungsumgebung.

  Mit dem Kontextobjekt können Sie zu Überwachungszwecken Informationen über den Aufruf Ihrer Funktion ausgeben. In diesem Beispiel verwendet Ihre Funktion den `log_group_name` Parameter, um den Namen ihrer CloudWatch Protokollgruppe auszugeben.

  Weitere Informationen zum Lambda-Kontextobjekt in Python finden Sie unter the section called “Kontext”.

- Protokollierung in Lambda:

  Mit Python können Sie entweder eine `print`-Anweisung oder eine Python-Protokollbibliothek verwenden, um Informationen an das Protokoll Ihrer Funktion zu senden. Um den Unterschied in der Erfassung zu veranschaulichen, werden im Beispielcode beide Methoden verwendet. In einer Produktionsanwendung empfehlen wir die Verwendung einer Protokollierungsbibliothek.

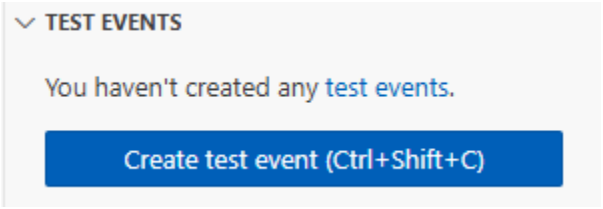
  Weitere Informationen hierzu finden Sie unter the section called “Protokollierung”. Informationen zum Protokollieren in anderen Laufzeiten finden Sie auf den Seiten „Erstellen mit“ für die Laufzeiten, an denen Sie interessiert sind.

Aufrufen der Lambda-Funktion mithilfe des Konsolencode-Editors

Um Ihre Funktion mit dem Code-Editor der Lambda-Konsole aufzurufen, erstellen Sie ein Testereignis, das Sie an Ihre Funktion senden. Beim Ereignis handelt es sich um ein JSON-formatiertes Dokument, das zwei Schlüssel-Wert-Paare mit den Schlüsseln `"length"` und `"width"`. enthält.

So erstellen Sie das Testereignis

1. Wählen Sie im Abschnitt TESTEREIGNISSE des Konsolen-Code-Editors die Option Testereignis erstellen aus.



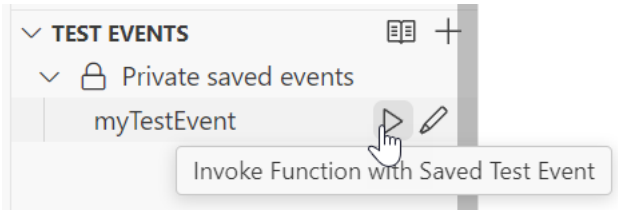
2. Geben Sie als Event name (Ereignisname) die Zeichenfolge **`myTestEvent`** ein.
3. Ersetzen Sie im Abschnitt Event JSON das Standard-JSON durch Folgendes:

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

4. Wählen Sie Speichern.

So testen Sie Ihre Funktion und zeigen Aufrufe an

Wählen Sie im Abschnitt TESTEREIGNISSE des Konsolencode-Editors das Ausführungssymbol neben Ihrem Testereignis:



Wenn Ihre Funktion beendet ist, werden die Antwort- und Funktionsprotokolle auf der Registerkarte OUTPUT angezeigt. Sie sollten ähnliche Ergebnisse wie nachfolgend zu sehen:

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

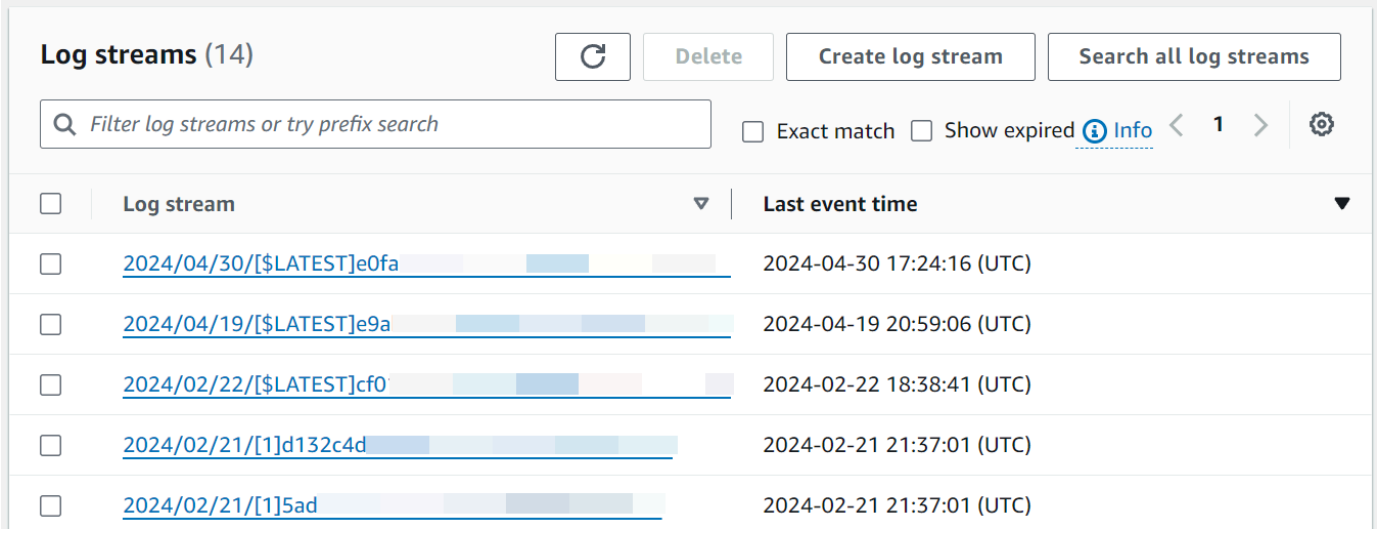
```
Status: Succeeded
Test Event Name: myTestEvent

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
```

Wenn Sie Ihre Funktion außerhalb der Lambda-Konsole aufrufen, müssen Sie CloudWatch Logs verwenden, um die Ausführungsergebnisse Ihrer Funktion einzusehen.

Um die Aufrufdatensätze Ihrer Funktion in Logs anzuzeigen CloudWatch

1. Öffnen Sie die Seite Protokollgruppen der CloudWatch Konsole.
2. Wählen Sie die Protokollgruppe für Ihre Funktion (`/aws/lambda/myLambdaFunction`) aus. Dies ist der Name der Protokollgruppe, den Ihre Funktion an die Konsole ausgegeben hat.

3. Scrollen Sie nach unten und wählen Sie den Protokollstream für die Funktionsaufrufen aus, die Sie sich ansehen möchten.



Die Ausgabe sollte folgendermaßen oder ähnlich aussehen:

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
```

Bereinigen

Wenn Sie mit der Beispielfunktion fertig sind, löschen Sie sie. Sie können auch die Protokollgruppe löschen, in der die Protokolle der Funktion gespeichert sind, sowie die von der Konsole erstellte Ausführungsrolle.

So löschen Sie die Lambda-Funktion:

1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.
2. Wählen Sie die Funktion aus, die Sie erstellt haben.
3. Wählen Sie Aktionen, Löschen aus.
4. Geben Sie **`confirm`** in das Texteingabefeld ein und wählen Sie Delete (Löschen) aus.

So löschen Sie die Protokollgruppe

1. Öffnen Sie die Seite Log Groups (Protokollgruppen) der CloudWatch-Konsole.
2. Wählen Sie die Protokollgruppe der Funktion (`/aws/lambda/myLambdaFunction`).
3. Wählen Sie Actions (Aktionen), Delete log group(s) (Protokollgruppe(n) löschen) aus.
4. Wählen Sie im Dialogfeld Delete log group(s) (Protokollgruppe(n) löschen) die Option Delete (Löschen) aus.

So löschen Sie die Ausführungsrolle

1. Öffnen Sie die Seite Rollen der AWS Identity and Access Management (IAM-) Konsole.
2. Wählen Sie die Ausführungsrolle der Funktion aus (zum Beispiel `myLambdaFunction-role-`*`31exxmpl`*).
3. Wählen Sie Löschen aus.
4. Geben Sie im Dialogfenster Delete role (Rolle löschen) den Namen der Rolle ein und wählen Sie anschließend Delete (Löschen) aus.

Zusätzliche Ressourcen und nächste Schritte

Nachdem Sie mit der Konsole eine einfache Lambda-Funktion erstellt und getestet haben, führen Sie die folgenden Schritte aus:

- Erfahren Sie, wie Sie Ihrer Funktion Abhängigkeiten hinzufügen und diesen mithilfe eines ZIP-Bereitstellungspakets bereitstellen. Wählen Sie aus den folgenden Links Ihre bevorzugte Sprache aus.

  Node.js

  the section called “Bereitstellen von ZIP-Dateiarchiven”

  Typescript

  the section called “Bereitstellen von ZIP-Dateiarchiven”

  Python

  the section called “Bereitstellen von ZIP-Dateiarchiven”

  Ruby

  the section called “Bereitstellen von ZIP-Dateiarchiven”

  Java

  the section called “Bereitstellen von ZIP-Dateiarchiven”

  Go

  the section called “Bereitstellen von ZIP-Dateiarchiven”

  C#

  the section called “Bereitstellungspaket”
- Informationen zum Aufrufen einer Lambda-Funktion mit einer anderen AWS-Service finden Sie unter. Tutorial: Verwenden eines Amazon-S3-Auslösers zum Aufrufen einer Lambda-Funktion
- Wählen Sie eines der folgenden Tutorials für komplexere Beispiele für die Verwendung von Lambda mit anderen AWS-Services.
  - Tutorial: Verwenden von Lambda mit API Gateway: Erstellen Sie eine Amazon-API-Gateway-REST-API, die eine Lambda-Funktion aufruft.

- Verwenden einer Lambda-Funktion für den Zugriff auf eine Amazon-RDS-Datenbank: Verwenden Sie eine Lambda-Funktion, um Daten über den RDS-Proxy in eine Datenbank des Amazon Relational Database Service (Amazon RDS) zu schreiben.
- Verwenden eines Amazon-S3-Auslösers zum Erstellen von Miniaturbildern: Verwenden Sie eine Lambda-Funktion, um bei jedem Hochladen einer Image-Datei in einen Amazon-S3-Bucket ein Miniaturbild zu erstellen.

Erste Schritte mit Beispielanwendungen und -mustern

Die folgenden Ressourcen können verwendet werden, um schnell Serverless-Apps zu erstellen und bereitzustellen, die einige gängige Lambda-Anwendungsfälle implementieren. Für jede der Beispiel-Apps bieten wir Anweisungen, um Ressourcen entweder manuell mithilfe von zu erstellen und zu konfigurieren oder AWS Serverless Application Model um die Ressourcen mithilfe von IaC bereitzustellen. AWS-Managementkonsole Folgen Sie den Anweisungen auf der Konsole, um mehr über die Konfiguration der einzelnen AWS Ressourcen für jede App zu erfahren, oder verwenden Sie sie, AWS SAM um Ressourcen schnell bereitzustellen, wie Sie es in einer Produktionsumgebung tun würden.

Dateiverarbeitung

- PDF-Verschlüsselungsanwendung: Erstellen Sie eine Serverless-Anwendung, die PDF-Dateien beim Hochladen in einen Bucket in Amazon Simple Storage Service verschlüsselt und in einem anderen Bucket speichert. Dies ist nützlich, um vertrauliche Dokumente beim Hochladen zu schützen.

- Bildanalyseanwendung: Erstellen Sie eine Serverless-Anwendung, die mithilfe von Amazon Rekognition Text aus Bildern extrahiert. Dies ist nützlich für die Dokumentverarbeitung, Inhaltsmoderation und automatisierte Bildanalyse.

Datenbankintegration

- Queue-to-Database Anwendung: Erstellen Sie eine serverlose Anwendung, die Warteschlangennachrichten in eine Amazon RDS-Datenbank schreibt. Dies ist nützlich für die Verarbeitung von Benutzerregistrierungen und die Bearbeitung von Bestellungen.

- Datenbankereignis-Handler: Erstellen Sie eine Serverless-Anwendung, die auf Änderungen in Amazon-DynamoDB-Tabellen reagiert. Dies ist nützlich für Audit-Protokollierung, Datenreplikation und automatisierte Workflows.

Geplante Aufgaben

- Anwendung zur Datenbankwartung: Erstellen Sie eine Serverless-Anwendung, die mithilfe eines Cron-Zeitplans automatisch Einträge, die älter als 12 Monate sind, aus einer Amazon-DynamoDB-Tabelle löscht. Dies ist für die automatisierte Datenbankwartung und das Datenlebenszyklusmanagement nützlich.
- Erstellen Sie eine EventBridge geplante Regel für Lambda-Funktionen: Verwenden Sie geplante Ausdrücke für Regeln EventBridge , um eine Lambda-Funktion nach einem Zeitplan auszulösen. Dieses Format verwendet die Cron-Syntax und kann mit einer Granularität von einer Minute festgelegt werden.

Workflows mit langer Laufzeit

- Anwendung zur Auftragsabwicklung: Erstellen Sie eine serverlose Anwendung mit Durable Functions, die komplexe Auftragsabwicklung, einschließlich Zahlungsabwicklung, Inventarprüfungen und Versandkoordination, abwickelt. Dieses Beispiel zeigt, wie Workflows erstellt werden, die über einen längeren Zeitraum ausgeführt werden können und gleichzeitig den Status beibehalten.

Weitere Ressourcen

Die folgenden Ressourcen enthalten weitere Informationen zu Lambda und zur Entwicklung von Serverless-Anwendungen:

- Serverless Land: Eine Bibliothek mit ready-to-use Mustern für die Erstellung serverloser Apps. Es hilft Entwicklern, Anwendungen mithilfe von AWS Diensten wie Lambda, API Gateway und EventBridge schneller zu erstellen. Die Website bietet vorgefertigte Lösungen und bewährte Methoden, die die Entwicklung von Serverless-Systemen erleichtern.
- Lambda-Beispielanwendungen: Anwendungen, die im GitHub Repository für dieses Handbuch verfügbar sind. Diese Beispiele demonstrieren die Verwendung verschiedener Sprachen und AWS Dienste. Jede Beispielanwendung enthält Skripts für die einfache Bereitstellung und Bereinigung und unterstützende Ressourcen.
- Codebeispiele für die Verwendung von Lambda AWS SDKs: Beispiele, die Ihnen zeigen, wie Sie Lambda mit AWS Software Development Kits verwenden ()SDKs. Diese Beispiele beinhalten Grundlagen, Aktionen, Szenarien und Beiträge aus der AWS Community. Beispiele umfassen

grundlegende Operationen, einzelne Servicefunktionen und spezifische Aufgaben, bei denen mehrere Funktionen oder AWS Dienste verwendet werden.

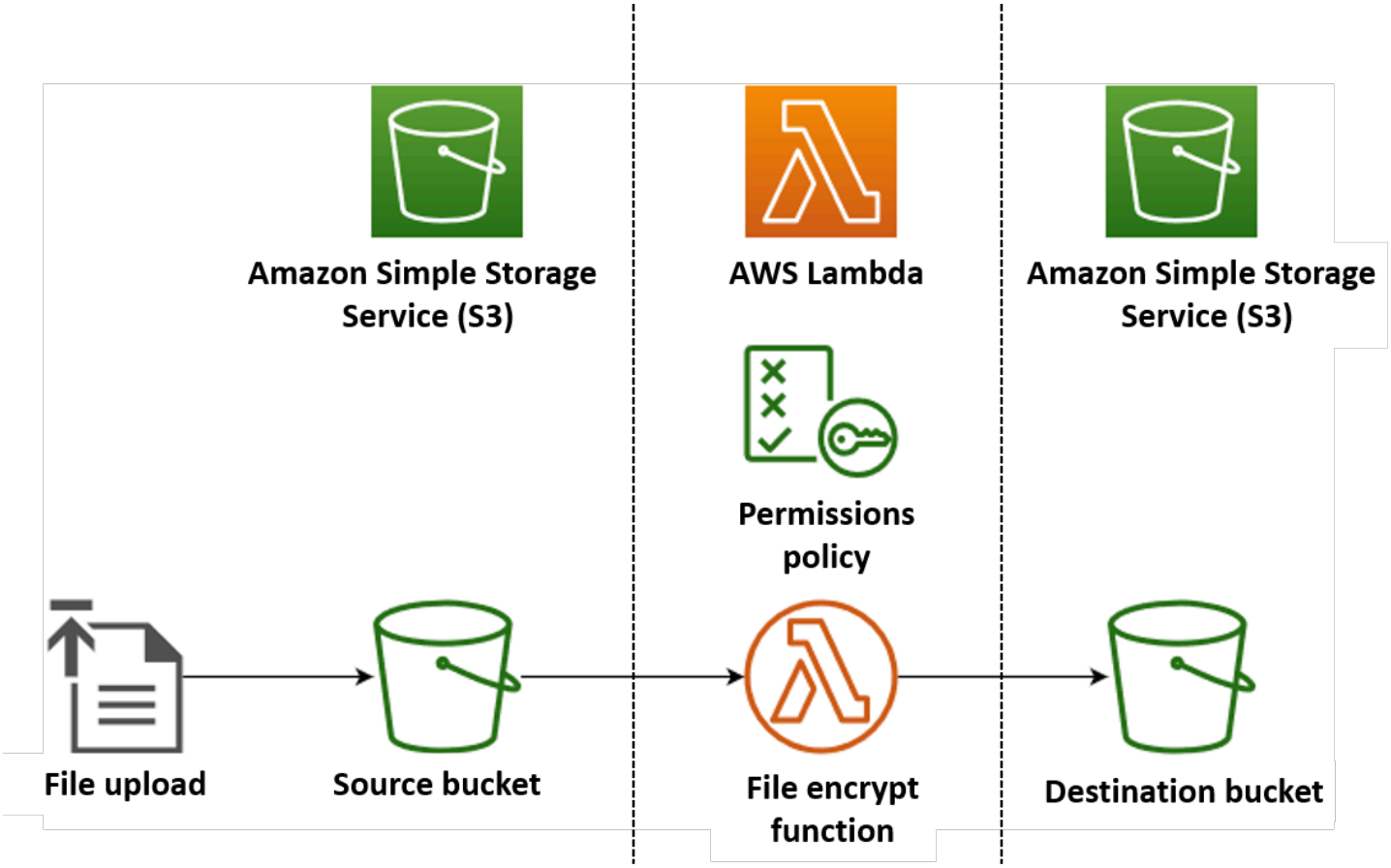
Erstellen Sie eine Serverless-Dateiverarbeitungs-App

Einer der häufigsten Anwendungsfälle für Lambda ist die Ausführung von Dateiverarbeitungsaufgaben. Sie können beispielsweise eine Lambda-Funktion verwenden, um automatisch PDF-Dateien aus HTML-Dateien oder Bildern zu erstellen oder um Miniaturansichten zu erstellen, wenn ein Benutzer ein Bild hochlädt.

In diesem Beispiel erstellen Sie eine App, die PDF-Dateien automatisch verschlüsselt, wenn sie in einen Amazon Simple Storage Service (Amazon S3)-Bucket uploaden. Um diese Anwendung zu implementieren, erstellen Sie die folgenden Ressourcen:

- Ein S3-Bucket, in das Benutzer PDF-Dateien hochladen können
- Eine Lambda-Funktion in Python, die die hochgeladene Datei liest und eine verschlüsselte, passwortgeschützte Version davon erstellt
- Ein zweiter S3-Bucket für Lambda zum Speichern der verschlüsselten Datei

Sie erstellen auch eine AWS Identity and Access Management (IAM-) Richtlinie, um Ihrer Lambda-Funktion die Erlaubnis zu erteilen, Lese- und Schreibvorgänge auf Ihren S3-Buckets durchzuführen.



> **Tip**
>
> Wenn Sie ganz neu bei Lambda sind, empfehlen wir Ihnen, vor der Erstellung dieser Beispiel-App mit dem Tutorial Erstellen Sie Ihre erste Funktion zu beginnen.

Sie können Ihre App manuell bereitstellen, indem Sie Ressourcen mit dem AWS-Managementkonsole oder dem AWS Command Line Interface () erstellen und konfigurieren.AWS CLI Sie können die App auch mithilfe von AWS Serverless Application Model (AWS SAM) bereitstellen. AWS SAM ist ein Tool für Infrastruktur als Code (IaC). Mit IaC erstellen Sie Ressourcen nicht manuell, sondern definieren sie im Code und stellen sie dann automatisch bereit.

Wenn Sie mehr über die Verwendung von Lambda mit IaC erfahren möchten, bevor Sie diese Beispiel-App bereitstellen, finden Sie unter the section called “Infrastructure as Code (IaC)”.

Erstellen der Quellcodedateien für die Lambda-Funktion

Erstellen Sie die folgenden Dateien in Ihrem Projektverzeichnis:

- `lambda_function.py` - Python-Funktionscode für die Lambda-Funktion, die die Dateiverschlüsselung durchführt
- `requirements.txt` - eine Manifestdatei, die die Abhängigkeiten definiert, die Ihr Python-Funktionscode benötigt

Erweitern Sie die folgenden Abschnitte, um den Code anzusehen und mehr über die Rolle der einzelnen Dateien zu erfahren. Um die Dateien auf Ihrem lokalen Computer zu erstellen, kopieren Sie entweder den folgenden Code und fügen Sie ihn ein, oder laden Sie die Dateien aus dem aws-lambda-developer-guide GitHub Repository herunter.

Python-Funktionscode

Kopieren Sie den folgenden Code und fügen Sie ihn in eine Datei mit dem Namen `lambda_function.py` ein.

```
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**
>
> In diesem Beispielcode ist ein Passwort für die verschlüsselte Datei (`my-secret-password`) fest im Funktionscode codiert. Schließen Sie in einer Produktionsanwendung keine sensiblen Informationen wie Passwörter in Ihren Funktionscode ein. Erstellen Sie stattdessen ein AWS Secrets Manager Geheimnis und verwenden Sie dann die Lambda-Erweiterung AWS Parameters and Secrets, um Ihre Anmeldeinformationen in Ihrer Lambda-Funktion abzurufen.

Der Python-Funktionscode enthält drei Funktionen: die Handler-Funktion, die Lambda ausführt, wenn Ihre Funktion aufgerufen wird und zwei separate Funktionen mit dem Namen `add_encrypted_suffix` und `encrypt_pdf`, die der Handler aufruft, um die PDF-Verschlüsselung durchzuführen.

Wenn Ihre Funktion von Amazon S3 aufgerufen wird, übergibt Lambda ein JSON-formatiertes Ereignisargument an die Funktion, das Details zu dem Ereignis enthält, das den Aufruf verursacht hat. In diesem Fall umfassen die Informationen den Namen des S3-Buckets und die Objektschlüssel

für die hochgeladenen Dateien. Weitere Informationen zum Format des Ereignisobjekts für Amazon S3 finden Sie unter the section called “S3”.

Ihre Funktion verwendet dann die AWS SDK für Python (Boto3) , um die im Ereignisobjekt angegebenen PDF-Dateien in ihr lokales temporäres Speicherverzeichnis herunterzuladen, bevor sie sie mithilfe der Bibliothek verschlüsselt. `pypdf`

Schließlich verwendet die Funktion das Boto3-SDK, um die verschlüsselte Datei in Ihrem S3-Ziel-Bucket zu speichern.

`requirements.txt`-Manifestdatei

Kopieren Sie den folgenden Code und fügen Sie ihn in eine Datei mit dem Namen `requirements.txt` ein.

```
boto3
pypdf
```

In diesem Beispiel hat Ihr Funktionscode nur zwei Abhängigkeiten, die nicht Teil der Standard-Python-Bibliothek sind: das SDK für Python (Boto3) und das `pypdf`-Paket, das die Funktion zur Durchführung der PDF-Verschlüsselung verwendet.

> **Note**
>
> Eine Version des SDK für Python (Boto3) ist Teil der Lambda-Laufzeit, sodass Ihr Code ausgeführt werden kann, ohne Boto3 zum Bereitstellungspaket Ihrer Funktion hinzuzufügen. Um jedoch die volle Kontrolle über die Abhängigkeiten Ihrer Funktion zu behalten und mögliche Probleme mit Versionsfehlern zu vermeiden, empfiehlt es sich für Python, alle Funktionsabhängigkeiten in das Bereitstellungspaket Ihrer Funktion aufzunehmen. Weitere Informationen hierzu finden Sie unter the section called “Laufzeitabhängigkeiten in Python”.

Bereitstellen der App

Sie können die Ressourcen für diese Beispiel-App entweder manuell oder mithilfe AWS SAM von erstellen und bereitstellen. In einer Produktionsumgebung empfehlen wir, ein IaC-Tool zu verwenden, AWS SAM um ganze serverlose Anwendungen schnell und wiederholbar bereitzustellen, ohne manuelle Prozesse zu verwenden.

Stellen Sie die Ressourcen manuell bereit

So stellen Sie Ihre App manuell bereit:

- Erstellen Sie Quell- und Ziel-Amazon-S3-Buckets
- Erstellen Sie eine Lambda-Funktion, die eine PDF-Datei verschlüsselt und die verschlüsselte Version in einem S3-Bucket speichert
- Konfigurieren Sie einen Lambda-Auslöser, der Ihre Funktion aufruft, wenn Objekte in Ihren Quell-Bucket hochgeladen werden

Stellen Sie zunächst sicher, dass Python auf Ihrer Build-Maschine installiert ist.

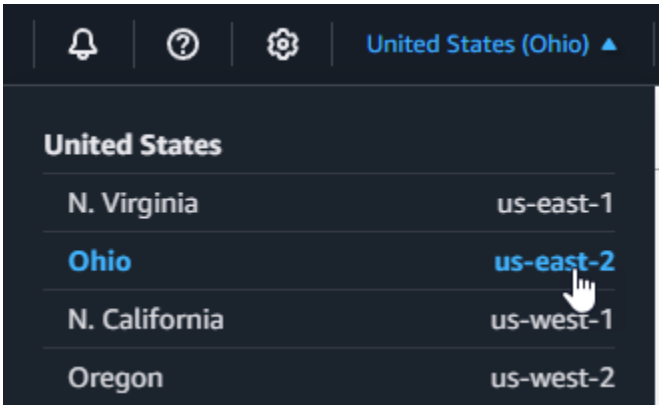
Erstellen Sie zwei S3-Buckets

Erstellen Sie zuerst zwei S3-Buckets. Der erste Bucket ist der Quell-Bucket, in den Sie Ihre PDF-Dateien hochladen. Der zweite Bucket wird von Lambda zum Speichern der verschlüsselten Datei verwendet, wenn Sie Ihre Funktion aufrufen.

Console

So erstellen Sie die S3-Buckets (Konsole)

1. Öffnen Sie die Seite Allzweck-Buckets in der Amazon-S3-Konsole.
2. Wählen Sie die Option aus, die Ihrem AWS-Region geografischen Standort am nächsten liegt. Sie können Ihre Region mithilfe der Dropdown-Liste am oberen Bildschirmrand ändern.



3. Wählen Sie Create Bucket (Bucket erstellen) aus.
4. Führen Sie unter Allgemeine Konfiguration die folgenden Schritte aus:

   a. Stellen Sie sicher, dass für Bucket-Typ die Option Allzweck ausgewählt ist.

b. Geben Sie für den Bucket-Namen einen global eindeutigen Namen ein, der den Regeln für die Bucket-Benennung von Amazon S3 entspricht. Bucket-Namen dürfen nur aus Kleinbuchstaben, Zahlen, Punkten (.) und Bindestrichen (-) bestehen.

5. Belassen Sie alle anderen Optionen auf ihren Standardwerten und wählen Sie Bucket erstellen aus.
6. Wiederholen Sie die Schritte 1 bis 4, um Ihren Ziel-Bucket zu erstellen. Geben Sie unter Bucket-Name den Namen **`amzn-s3-demo-bucket-encrypted`** ein. **`amzn-s3-demo-bucket`** ist hierbei der Name des Quell-Buckets, den Sie soeben erstellt haben.

AWS CLI

Stellen Sie zunächst sicher, dass die AWS CLI auf Ihrer Build-Maschine installiert ist.

So erstellen Sie die Amazon-S3-Buckets (AWS CLI)

1. Führen Sie den folgenden CLI-Befehl aus, um Ihren Quell-Bucket zu erstellen. Der gewählte Name für Ihren Bucket muss global eindeutig sein und den Regeln für die Benennung von Buckets von Amazon S3 entsprechen. Namen dürfen nur Kleinbuchstaben, Zahlen, Punkte (.) und Bindestriche (-) enthalten. Wählen Sie für `region` und `LocationConstraint` die AWS-Region aus, die Ihrem geografischen Standort am nächsten ist.

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

   Später im Tutorial müssen Sie Ihre Lambda-Funktion im selben Verzeichnis AWS-Region wie in Ihrem Quell-Bucket erstellen. Notieren Sie sich also die Region, die Sie ausgewählt haben.

2. Führen Sie den folgenden Befehl aus, um Ihren Ziel-Bucket zu erstellen. Verwenden Sie für den Bucket-Namen **`amzn-s3-demo-bucket-encrypted`**. **`amzn-s3-demo-bucket`** ist hierbei der Name des Quell-Buckets, den Sie in Schritt 1 erstellt haben. Wählen Sie für `region` und dasselbe aus`LocationConstraint`, das AWS-Region Sie zur Erstellung Ihres Quell-Buckets verwendet haben.

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

Erstellen einer Ausführungsrolle

Eine Ausführungsrolle ist eine IAM-Rolle, die einer Lambda-Funktion Zugriff AWS-Services und Ressourcen gewährt. Fügen Sie die AWS -verwaltete Richtlinie `AmazonS3FullAccess` an, um Ihrer Funktion Lese- und Schreibzugriff auf Amazon S3 zu erteilen.

Console

So erstellen Sie eine Ausführungsrolle und fügen die verwaltete **`AmazonS3FullAccess`**-Richtlinie an (Konsole)

1. Öffnen Sie die Seite Roles (Rollen) in der IAM-Konsole.
2. Wählen Sie Rolle erstellen aus.
3. Wählen Sie unter Vertrauenswürdiger Entitätstyp die Option AWS -Service und dann für Anwendungsfall die Option Lambda aus.
4. Wählen Sie Weiter aus.
5. Fügen Sie die verwaltete `AmazonS3FullAccess`-Richtlinie wie folgt hinzu:
   a. Geben Sie unter Berechtigungsrichtlinien die Zeichenfolge **`AmazonS3FullAccess`** in das Suchfeld ein.
   b. Aktivieren Sie das Kontrollkästchen neben der Richtlinie.
   c. Wählen Sie Weiter aus.
6. Geben Sie unter Rollendetails für Rollenname den Namen **`LambdaS3Role`** ein.
7. Wählen Sie Rolle erstellen aus.

AWS CLI

So erstellen Sie eine Ausführungsrolle und fügen die **`AmazonS3FullAccess`**-verwaltete Richtlinie an (AWS CLI)

1. Speichern Sie den folgenden JSON-Code in einer Datei mit dem Namen `trust-policy.json`: Diese Vertrauensrichtlinie ermöglicht es Lambda, die Berechtigungen der Rolle zu verwenden, indem dem Dienstprinzipal die `lambda.amazonaws.com` Erlaubnis erteilt wird, die `AssumeRole` Aktion AWS -Security-Token-Service (AWS STS) aufzurufen.

```
{
```

```
  "Version":"2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "lambda.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
```

2. Führen Sie in dem Verzeichnis, in dem Sie das JSON-Vertrauensrichtliniendokument gespeichert haben, den folgenden CLI-Befehl aus, um die Ausführungsrolle zu erstellen:

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

3. Führen Sie den folgenden CLI-Befehl aus, um die `AmazonS3FullAccess`-verwaltete Richtlinie anzuhängen.

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

Erstellen des Bereitstellungspakets für die Funktion

Um Ihre Funktion zu erstellen, erstellen Sie ein Bereitstellungspaket, das Ihren Funktionscode und die zugehörigen Abhängigkeiten enthält. Bei der hier verwendeten Anwendung verwendet Ihr Funktionscode eine separate Bibliothek für die PDF-Verschlüsselung.

Erstellen des Bereitstellungspakets

1. Navigieren Sie zu dem Projektverzeichnis, das die `lambda_function.py` `requirements.txt` Dateien enthält, die Sie GitHub zuvor erstellt oder heruntergeladen haben, und erstellen Sie ein neues Verzeichnis mit dem Namen`package`.
2. Installieren Sie die in der `requirements.txt`-Datei angegebenen Abhängigkeiten in Ihrem `package`-Verzeichnis, indem Sie den folgenden Befehl ausführen.

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

3. Erstellen Sie eine .zip-Datei, die Ihren Anwendungscode und seine Abhängigkeiten enthält. Führen Sie unter Linux oder MacOS die folgenden Befehle über Ihre Befehlszeilenschnittstelle aus:

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

   Verwenden Sie unter Windows Ihr bevorzugtes ZIP-Tool, um die `lambda_function.zip`-Datei zu erstellen. Achten Sie darauf, dass sich die Datei `lambda_function.py` und die Ordner, die Ihre Abhängigkeiten enthalten, im Stammverzeichnis der ZIP-Datei befinden.

Sie können Ihr Bereitstellungspaket auch in einer virtuellen Python-Umgebung erstellen. Siehe Arbeiten mit ZIP-Dateiarchiven und Python-Lambda-Funktionen

So erstellen Sie die Lambda-Funktion:

Sie verwenden jetzt das Bereitstellungspaket, das Sie im vorherigen Schritt erstellt haben, um Ihre Lambda-Funktion bereitzustellen.

Console

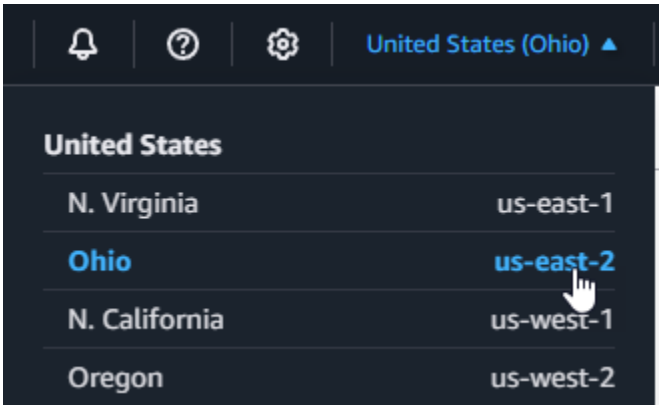
So erstellen Sie die Funktion (Konsole)

Wenn Sie Ihre Lambda-Funktion über die Konsole erstellen möchten, erstellen Sie zunächst eine Basisfunktion, die etwas Hello-World-Code enthält. Anschließend ersetzen Sie diesen Code durch Ihren eigenen Funktionscode, indem Sie die im vorherigen Schritt erstellte ZIP- oder Datei hochladen.

Um sicherzustellen, dass Ihre Funktion beim Verschlüsseln großer PDF-Dateien nicht zu einem Timeout kommt, konfigurieren Sie die Speicher- und Timeout-Einstellungen der Funktion. Sie legen auch das Protokollformat der Funktion auf JSON fest. Die Konfiguration von Protokollen im JSON-Format ist erforderlich, wenn Sie das mitgelieferte Testskript verwenden, damit es den Aufrufstatus der Funktion aus den CloudWatch Protokollen lesen kann, um den erfolgreichen Aufruf zu bestätigen.

1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.

2. Stellen Sie sicher, dass Sie in demselben Modus arbeiten, in dem AWS-Region Sie Ihren S3-Bucket erstellt haben. Sie können Ihre Region mithilfe der Dropdown-Liste am oberen Bildschirmrand ändern.



3. Wählen Sie Funktion erstellen.
4. Wählen Sie Author from scratch aus.
5. Führen Sie unter Basic information (Grundlegende Informationen) die folgenden Schritte aus:
   a. Geben Sie für Function name (Funktionsname) **`EncryptPDF`** ein.
   b. Wählen Sie für Laufzeit die Option Python 3.12 aus.
   c. Wählen Sie für Architektur x86_64 aus.
6. Fügen Sie die Ausführungsrolle, die Sie im vorherigen Schritt erstellt haben, wie folgt hinzu:
   a. Erweitern Sie den Abschnitt Ändern der standardmäßigen Ausführungsrolle.
   b. Wählen Sie Eine vorhandene Rolle verwenden aus.
   c. Wählen Sie unter Bestehende Rolle Ihre Rolle (`LambdaS3Role`) aus.
7. Wählen Sie Funktion erstellen.

So laden Sie den Funktionscode hoch (Konsole)

1. Wählen Sie im Bereich Codequelle die Option Hochladen von aus.
2. Wählen Sie die ZIP-Datei aus.
3. Klicken Sie auf Upload.
4. Wählen Sie in der Dateiauswahl Ihre ZIP-Datei und anschließend Öffnen aus.
5. Wählen Sie Speichern.

Um den Funktionsspeicher und das Timeout (Konsole) zu konfigurieren

1. Wählen Sie die Registerkarte Konfiguration für Ihre Funktion.
2. Wählen Sie im Bereich Allgemeine Konfiguration die Option Bearbeiten aus.
3. Stellen Sie Speicher auf 256 MB und Timeout auf 15 Sekunden ein.
4. Wählen Sie Speichern.

So konfigurieren Sie das Protokollformat (Konsole)

1. Wählen Sie die Registerkarte Konfiguration für Ihre Funktion.
2. Wählen Sie Überwachungs- und Betriebstools aus.
3. Wählen Sie im Bereich Protokollierungskonfiguration die Option Bearbeiten aus.
4. Wählen Sie für die Konfiguration der Protokollierung JSON aus.
5. Wählen Sie Speichern.

AWS CLI

So erstellen Sie die Funktion (AWS CLI)

- Führen Sie den folgenden Befehl in dem Verzeichnis aus, das Ihre `lambda_function.zip`-Datei enthält. Ersetzen Sie `region` für den Parameter `us-east-2` durch die Region, in der Sie Ihre S3-Buckets erstellt haben.

```
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
```

Konfigurieren von Amazon-S3-Auslöser zum Aufrufen der Funktion

Damit Ihre Lambda-Funktion ausgeführt wird, wenn Sie eine Datei in Ihren Quell-Bucket hochladen, müssen Sie einen Auslöser für Ihre Funktion konfigurieren. Sie können den Amazon-S3-Auslöser entweder über die Konsole oder mithilfe der AWS CLI konfigurieren.

> **Important**
>
> Mit diesem Verfahren wird der S3-Bucket so konfiguriert, dass die Funktion jedes Mal aufgerufen wird, wenn ein Objekt im Bucket erstellt wird. Konfigurieren Sie dieses Verhalten nur für den Quell-Bucket. Wenn Ihre Lambda-Funktion Objekte in dem Bucket erstellt, der sie aufruft, kann es passieren, dass Ihre Funktion kontinuierlich in einer Schleife aufgerufen wird. Dies kann dazu führen, dass Ihnen AWS-Konto unerwartete Gebühren in Rechnung gestellt werden.

Console

So konfigurieren Sie den Amazon-S3-Auslöser (Konsole)

1. Öffnen Sie die Seite Funktionen der Lambda-Konsole und wählen Sie Ihre Funktion (`EncryptPDF`) aus.
2. Wählen Sie Add trigger.
3. Wählen Sie S3 aus.
4. Wählen Sie unter Bucket Ihren Quell-Bucket aus.
5. Wählen Sie unter Ereignistypen die Option Alle Objekterstellungsereignisse aus.
6. Aktivieren Sie unter Rekursiver Aufruf das Kontrollkästchen, um zu bestätigen, dass die Verwendung desselben S3-Buckets für die Ein- und Ausgabe nicht empfohlen wird. Weitere Informationen zu rekursiven Aufrufmustern in Lambda finden Sie bei Serverless Land unter Recursive patterns that cause run-away Lambda functions.
7. Wählen Sie Hinzufügen aus.

   Wenn Sie einen Auslöser über die Lambda-Konsole erstellen, erstellt Lambda automatisch eine ressourcenbasierte Richtlinie, um dem von Ihnen ausgewählten Dienst die Berechtigung zum Aufrufen Ihrer Funktion zu erteilen.

AWS CLI

So konfigurieren Sie den Amazon-S3-Auslöser (AWS CLI)

1. Fügen Sie Ihrer Funktion eine ressourcenbasierte Richtlinie hinzu, die es Ihrem Amazon-S3-Quell-Bucket ermöglicht, Ihre Funktion aufzurufen, wenn Sie eine Datei hinzufügen. Eine ressourcenbasierte Grundsatzerklärung erteilt anderen Personen die AWS-Services

Erlaubnis, Ihre Funktion aufzurufen. Führen Sie den folgenden CLI-Befehl aus, um Amazon S3 die Berechtigung zum Aufrufen Ihrer Funktion zu erteilen. Achten Sie darauf, den `source-account` Parameter durch Ihre eigene AWS-Konto ID zu ersetzen und Ihren eigenen Quell-Bucket-Namen zu verwenden.

```
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
```

Die mit diesem Befehl definierte Richtlinie ermöglicht es Amazon S3 nur, Ihre Funktion aufzurufen, wenn eine Aktion in Ihrem Quell-Bucket ausgeführt wird.

> **Note**
>
> Namen von S3-Buckets sind zwar global eindeutig, trotzdem empfiehlt es sich, bei der Verwendung ressourcenbasierter Richtlinien anzugeben, dass der Bucket zu Ihrem Konto gehören muss. Denn wenn Sie einen Bucket löschen, kann ein anderer AWS-Konto Bucket einen Bucket mit demselben Amazon-Ressourcennamen (ARN) erstellen.

2. Speichern Sie den folgenden JSON-Code in einer Datei mit dem Namen `notification.json`: Wenn dieser JSON-Code auf Ihren Quell-Bucket angewendet wird, wird der Bucket so konfiguriert, dass er jedes Mal, wenn ein neues Objekt hinzugefügt wird, eine Benachrichtigung an Ihre Lambda-Funktion sendet. Ersetzen Sie die AWS-Konto Nummer und AWS-Region in der Lambda-Funktion ARN durch Ihre eigene Kontonummer und Region.

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

3. Führen Sie den folgenden CLI-Befehl aus, um die Benachrichtigungseinstellungen in der von Ihnen erstellten JSON-Datei auf Ihren Quell-Bucket anzuwenden. Ersetzen Sie dabei `amzn-s3-demo-bucket` durch den Namen Ihres Quell-Buckets.

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

Weitere Informationen zu dem `put-bucket-notification-configuration` Befehl und der `notification-configuration` Option finden Sie put-bucket-notification-configurationin der AWS CLI-Befehlsreferenz.

Stellen Sie die Ressourcen bereit mit AWS SAM

Stellen Sie zunächst sicher, dass Docker und die neueste Version von AWS SAM CLI auf Ihrer Build-Maschine installiert sind.

1. Kopieren Sie im Projektverzeichnis den folgenden Code und fügen Sie ihn in eine Datei namens `template.yaml` ein. Ersetzen Sie die Platzhalter-Bucket-Namen:

   - Ersetzen Sie für den Quell-Bucket `amzn-s3-demo-bucket` durch einen beliebigen Namen, der den Benennungsregeln für S3-Buckets entspricht.
   - Ersetzen Sie für den Zielbereich `amzn-s3-demo-bucket-encrypted` durch `<source-bucket-name>-encrypted`, wobei `<source-bucket>` der Name ist, den Sie für Ihren Quellbereich gewählt haben.

```
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
```

Die AWS SAM Vorlage definiert die Ressourcen, die Sie für Ihre App erstellen. In diesem Beispiel definiert die Vorlage eine Lambda-Funktion unter Verwendung des `AWS::Serverless::Function`-Typs und zwei S3-Buckets unter Verwendung des `AWS::S3::Bucket`-Typs. Die in der Vorlage angegebenen Bucket-Namen sind Platzhalter. Bevor Sie die App mithilfe bereitstellen AWS SAM, müssen Sie die Vorlage bearbeiten, um die Buckets mit global eindeutigen Namen umzubenennen, die den S3-Bucket-Namensregeln entsprechen. Dieser Schritt wird unter the section called “Stellen Sie die Ressourcen bereit mit AWS SAM” näher erläutert.

Die Definition der Lambda-Funktionsressource konfiguriert einen Trigger für die Funktion mithilfe der `S3Event`-Event-Eigenschaft. Dieser Trigger bewirkt, dass Ihre Funktion immer dann aufgerufen wird, wenn ein Objekt in Ihrem Quell-Bucket erstellt wird.

Die Funktionsdefinition spezifiziert auch eine AWS Identity and Access Management (IAM-) Richtlinie, die der Ausführungsrolle der Funktion zugeordnet werden soll. Die AWS -verwaltete Richtlinie `AmazonS3FullAccess` gibt Ihrer Funktion die Berechtigungen, die sie zum Lesen und Schreiben von Objekten in Amazon S3 benötigt.

2. Führen Sie den folgenden Befehl in dem Verzeichnis aus, in dem Sie Ihre `template.yaml`-, `lambda_function.py`- und `requirements.txt`-Dateien gespeichert haben.

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

Dieser Befehl sammelt die Build-Artefakte für Ihre Anwendung und platziert sie im richtigen Format und am richtigen Ort, um sie bereitzustellen. Wenn Sie die Option `--use-container` angeben, wird Ihre Funktion innerhalb eines Lambda-ähnlichen Docker-Containers erstellt. Wir verwenden es hier, sodass Sie Python 3.12 nicht auf Ihrem lokalen Computer installiert haben müssen, damit der Build funktioniert.

Sucht während des Erstellungsvorgangs nach dem Lambda-Funktionscode an der Stelle, die Sie mit der `CodeUri` Eigenschaft in der Vorlage angegeben haben. AWS SAM In diesem Fall haben wir das aktuelle Verzeichnis als Standort (`./`) angegeben.

Wenn eine `requirements.txt` Datei vorhanden ist, AWS SAM verwendet sie, um die angegebenen Abhängigkeiten zu sammeln. AWS SAM Erstellt standardmäßig ein ZIP-Bereitstellungspaket mit Ihrem Funktionscode und Ihren Abhängigkeiten. Mithilfe der PackageTypeEigenschaft können Sie Ihre Funktion auch als Container-Image bereitstellen.

3. Führen Sie den folgenden Befehl aus, um Ihre Anwendung bereitzustellen und die in Ihrer AWS SAM Vorlage angegebenen Lambda- und Amazon S3 S3-Ressourcen zu erstellen.

```
sam deploy --guided
```

Wenn Sie die `--guided` Markierung verwenden, AWS SAM werden Ihnen Eingabeaufforderungen angezeigt, die Sie durch den Bereitstellungsprozess führen. Akzeptieren Sie für diese Bereitstellung die Standardoptionen, indem Sie die Eingabetaste drücken.

AWS SAM Erstellt während des Bereitstellungsprozesses die folgenden Ressourcen in Ihrem AWS-Konto:

- Ein CloudFormation Stack mit dem Namen `sam-app`
- Eine Lambda-Funktion mit dem Namen `EncryptPDF`
- Zwei S3-Buckets mit den Namen, die Sie bei der Bearbeitung der `template.yaml` AWS SAM Vorlagendatei ausgewählt haben
- Eine IAM-Ausführungsrolle für Ihre Funktion mit dem Namensformat `sam-app-EncryptPDFFunctionRole-2qGaapHFW0Q8`

Wenn Sie mit der Erstellung Ihrer Ressourcen AWS SAM fertig sind, sollte die folgende Meldung angezeigt werden:

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

Testen der Anwendung

Um Ihre App zu testen, laden Sie eine PDF-Datei in Ihren Quell-Bucket hoch und bestätigen, dass Lambda eine verschlüsselte Version der Datei in Ihrem Ziel-Bucket erstellt. In diesem Beispiel können Sie dies entweder manuell mit der Konsole oder dem AWS CLI oder mithilfe des bereitgestellten Testskripts testen.

Für Produktionsanwendungen können Sie herkömmliche Testmethoden und -techniken wie Modultests verwenden, um die korrekte Funktionsweise Ihres Lambda-Funktionscodes zu überprüfen. Es empfiehlt sich auch, Tests wie die im bereitgestellten Testskript enthaltenen durchzuführen, bei denen Integrationstests mit echten, cloudbasierten Ressourcen durchgeführt werden. Integrationstests in der Cloud bestätigen, dass Ihre Infrastruktur korrekt bereitgestellt wurde und dass Ereignisse erwartungsgemäß zwischen verschiedenen Diensten ablaufen. Weitere Informationen hierzu finden Sie unter Serverless-Funktionen testen.

Manuelles Testen der App

Sie können Ihre Funktion manuell testen, indem Sie Ihrem Amazon-S3-Quell-Bucket eine PDF-Datei hinzufügen. Wenn Sie Ihre Datei dem Quell-Bucket hinzufügen, sollte Ihre Lambda-Funktion automatisch aufgerufen werden und eine verschlüsselte Version der Datei in Ihrem Ziel-Bucket speichern.

Console

So testen Sie Ihre Anwendung, indem Sie eine Datei hochladen (Konsole)

1. Um eine PDF-Datei in Ihren S3-Bucket hochzuladen, gehen Sie wie folgt vor:
   a. Öffnen Sie die Seite Buckets der Amazon-S3-Konsole und wählen Sie Ihren Quell-Bucket aus.
   b. Klicken Sie auf Upload.
   c. Wählen Sie Dateien hinzufügen und anschließend über die Dateiauswahl die PDF-Datei aus, die Sie hochladen möchten.
   d. Wählen Sie Öffnen und anschließend Hochladen aus.

2. Überprüfen Sie, ob Lambda eine verschlüsselte Version Ihrer PDF-Datei in Ihrem Ziel-Bucket gespeichert hat, indem Sie Folgendes tun:

   a. Navigieren Sie wieder zur Seite Buckets der Amazon-S3-Konsole und wählen Sie Ihren Ziel-Bucket aus.

   b. Im Objektbereich sollten Sie nun eine Datei mit dem Namen `filename_encrypted.pdf`-Format sehen (wobei `filename.pdf` der Name der Datei ist, die Sie in Ihren Quell-Bucket hochgeladen haben). Um Ihre verschlüsselte PDF-Datei herunterzuladen, wählen Sie die Datei aus und wählen Sie dann Herunterladen aus.

   c. Bestätigen Sie, dass Sie die heruntergeladene Datei mit dem Passwort öffnen können, mit dem Ihre Lambda-Funktion sie geschützt hat (`my-secret-password`).

AWS CLI

So testen Sie Ihre Anwendung, indem Sie eine Datei hochladen (AWS CLI)

1. Führen Sie in dem Verzeichnis, das die hochzuladende PDF-Datei enthält, den folgenden CLI-Befehl aus. Ersetzen Sie dabei den Parameter `--bucket` durch den Namen Ihres Quell-Buckets. Verwenden Sie für die `--key`- und `--body`-Parameter den Dateinamen Ihres Testbilds.

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

2. Vergewissern Sie sich, dass Ihre Funktion eine verschlüsselte Version Ihrer Datei erstellt und in Ihrem S3-Ziel-Bucket gespeichert hat. Führen Sie den folgenden CLI-Befehl aus und ersetzen Sie dabei `amzn-s3-demo-bucket-encrypted` durch den Namen Ihres eigenen Ziel-Buckets:

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

   Wenn Ihre Funktion erfolgreich ausgeführt wird, erhalten Sie eine Ausgabe wie die folgende. Ihr Ziel-Bucket sollte eine Datei mit dem Namensformat *<your_test_file>*`_encrypted.pdf` enthalten, wobei `<your_test_file>` der Name der von Ihnen hochgeladenen Datei ist.

```
{
    "Contents": [
```

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

3. Führen Sie den folgenden CLI-Befehl aus, um die Datei herunterzuladen, die Lambda in Ihrem Ziel-Bucket gespeichert hat. Ersetzen Sie den `--bucket`-Parameter durch den Namen Ihres Ziel-Buckets. Verwenden Sie für den Parameter `--key` den Dateinamen *<your_test_file>*`_encrypted.pdf`, wobei <your_test_file> der Name der von Ihnen hochgeladenen Testdatei ist.

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

   Dieser Befehl lädt die Datei in Ihr aktuelles Verzeichnis herunter und speichert sie als `my_encrypted_file.pdf`.

4. Bestätigen Sie, dass Sie die heruntergeladene Datei mit dem Passwort öffnen können, mit dem Ihre Lambda-Funktion sie geschützt hat (`my-secret-password`).

Testen Sie die App mit dem automatisierten Skript

Erstellen Sie die folgenden Dateien in Ihrem Projektverzeichnis:

- `test_pdf_encrypt.py` - ein Testskript, mit dem Sie Ihre Anwendung automatisch testen können
- `pytest.ini` - eine Konfigurationsdatei für das Testskript

Erweitern Sie die folgenden Abschnitte, um den Code anzusehen und mehr über die Rolle der einzelnen Dateien zu erfahren.

Automatisierte Testskripte

Kopieren Sie den folgenden Code und fügen Sie ihn in eine Datei mit dem Namen `test_pdf_encrypt.py` ein. Ersetzen Sie unbedingt die Platzhalter-Bucket-Namen:

- Ersetzen Sie in der `test_source_bucket_available`-Funktion `amzn-s3-demo-bucket` durch den Namen Ihres Quell-Eimers.
- Ersetzen Sie in der Funktion `test_encrypted_file_in_bucket` `amzn-s3-demo-bucket-encrypted` durch *`source-bucket`*`-encrypted`, wobei `source-bucket>` der Name Ihres Quell-Eimers ist.
- Ersetzen Sie in der `cleanup`-Funktion `amzn-s3-demo-bucket` durch den Namen Ihres Quell-Buckets und ersetzen Sie `amzn-s3-demo-bucket-encrypted` durch den Namen Ihres Ziel-Buckets.

```
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
```

Das automatisierte Testskript führt drei Testfunktionen aus, um den korrekten Betrieb Ihrer App zu bestätigen:

- Der Test `test_source_bucket_available` bestätigt, dass Ihr Quell-Bucket erfolgreich erstellt wurde, indem eine Test-PDF-Datei in den Bucket hochgeladen wird.
- Der Test `test_lambda_invoked` fragt den neuesten CloudWatch Log-Log-Stream für Ihre Funktion ab, um zu bestätigen, dass Ihre Lambda-Funktion beim Hochladen der Testdatei ausgeführt wurde und erfolgreich gemeldet wurde.
- Der Test `test_encrypted_file_in_bucket` bestätigt, dass Ihr Ziel-Bucket die verschlüsselte `test_encrypted.pdf`-Datei enthält.

Nachdem all diese Tests ausgeführt wurden, führt das Skript einen zusätzlichen Bereinigungsschritt aus, um die `test.pdf`- und `test_encrypted.pdf`-Dateien sowohl aus Ihrem Quell- als auch aus Ihrem Ziel-Bucket zu löschen.

Wie bei der AWS SAM Vorlage sind die in dieser Datei angegebenen Bucket-Namen Platzhalter. Bevor Sie den Test ausführen, müssen Sie diese Datei mit den echten Bucket-Namen Ihrer App bearbeiten. Dieser Schritt wird unter the section called “Testen Sie die App mit dem automatisierten Skript” näher erläutert

Testskript-Konfigurationsdatei

Kopieren Sie den folgenden Code und fügen Sie ihn in eine Datei mit dem Namen `pytest.ini` ein.

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

Dies ist erforderlich, um die Reihenfolge anzugeben, in der die Tests im `test_pdf_encrypt.py`-Skript ausgeführt werden.

Um die Tests durchzuführen, gehen Sie wie folgt vor:

1. Stellen Sie sicher, dass das `pytest`-Modul in Ihrer lokalen Umgebung installiert ist. Sie können `pytest` installieren, indem Sie den folgenden Befehl ausführen:

   ```
   pip install pytest
   ```
2. Speichern Sie eine PDF-Datei mit dem Namen `test.pdf` in dem Verzeichnis, das die `test_pdf_encrypt.py`- und `pytest.ini`-Dateien enthält.
3. Öffnen Sie ein Terminal- oder Shell-Programm und führen Sie den folgenden Befehl in dem Verzeichnis aus, das die Testdateien enthält.

   ```
   pytest -s -v
   ```

   Wenn der Test abgeschlossen ist, sollten Sie eine Ausgabe wie die folgende sehen:

   ```
   ============================================================ 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
 ===============================================================
```

Nächste Schritte

Nachdem Sie diese Beispiel-App erstellt haben, können Sie den bereitgestellten Code als Grundlage verwenden, um andere Arten von Dateiverarbeitungsanwendungen zu erstellen. Ändern Sie den Code in der `lambda_function.py`-Datei, um die Dateiverarbeitungslogik für Ihren Anwendungsfall zu implementieren.

Viele typische Anwendungsfälle für die Dateiverarbeitung beinhalten Bildverarbeitung. Bei der Verwendung von Python enthalten die beliebtesten Bildverarbeitungsbibliotheken wie Pillow typischerweise C- oder C++-Komponenten. Um sicherzustellen, dass das Bereitstellungspaket Ihrer Funktion mit der Lambda-Ausführungsumgebung kompatibel ist, ist es wichtig, die richtige Binärdatei für die Quelldistribution zu verwenden.

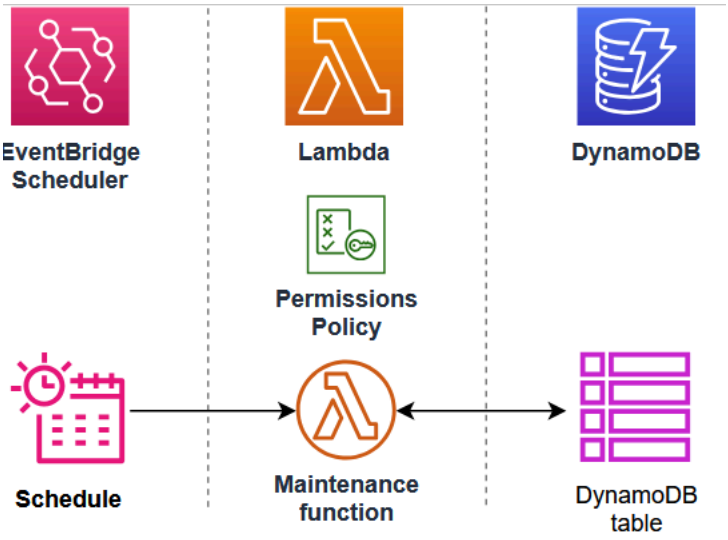
Bei der Bereitstellung Ihrer Ressourcen mit müssen Sie einige zusätzliche Schritte unternehmen AWS SAM, um die richtige Quelldistribution in Ihr Bereitstellungspaket aufzunehmen. Da AWS SAM keine Abhängigkeiten für eine andere Plattform als Ihre Build-Maschine installiert werden, funktioniert die Angabe der richtigen Quelldistribution (`.whl`Datei) in Ihrer `requirements.txt` Datei nicht, wenn Ihre Build-Maschine ein Betriebssystem oder eine Architektur verwendet, die sich von der Lambda-Ausführungsumgebung unterscheidet. Stattdessen sollten Sie eine der folgenden Möglichkeiten nutzen:

- Verwenden Sie die `--use-container`-Option beim Ausführen von `sam build`. Wenn Sie diese Option angeben, wird ein Container-Basis-Image AWS SAM heruntergeladen, das mit der Lambda-Ausführungsumgebung kompatibel ist, und erstellt das Bereitstellungspaket Ihrer Funktion mithilfe dieses Images in einem Docker-Container. Weitere Informationen finden Sie unter Eine Lambda-Funktion in einem bereitgestellten Container erstellen.
- Erstellen Sie das ZIP-Bereitstellungspaket Ihrer Funktion mithilfe der richtigen Binärdatei für die Quelldistribution selbst und speichern Sie die ZIP-Datei in dem Verzeichnis, das Sie in der `CodeUri` Vorlage angeben. AWS SAM Weitere Informationen zum Erstellen von .zip-Bereitstellungspaketen für Python mithilfe von Binärverteilungen finden Sie unter the section called “ZIP-Bereitstellungspakets mit Abhängigkeiten erstellen” und the section called “ZIP-Bereitstellungspakete mit nativen Bibliotheken erstellen”.

Erstellen Sie eine App, um eine geplante Datenbankwartung durchzuführen

Sie können AWS Lambda damit geplante Prozesse wie automatische Systemsicherungen, Dateikonvertierungen und Wartungsaufgaben ersetzen. In diesem Beispiel erstellen Sie eine Serverless-Anwendung, die regelmäßig geplante Wartungsarbeiten an einer DynamoDB-Tabelle durchführt, indem alte Einträge gelöscht werden. Die App verwendet EventBridge Scheduler, um eine Lambda-Funktion nach einem Cron-Zeitplan aufzurufen. Wenn die Funktion aufgerufen wird, fragt sie die Tabelle nach Elementen ab, die älter als ein Jahr sind und löscht sie. Die Funktion protokolliert jedes gelöschte Element in Logs. CloudWatch

Um dieses Beispiel zu implementieren, erstellen Sie zunächst eine DynamoDB-Tabelle und füllen Sie sie mit einigen Testdaten auf, damit Ihre Funktion sie abfragen kann. Erstellen Sie dann eine Python-Lambda-Funktion mit einem EventBridge Scheduler-Trigger und einer IAM-Ausführungsrolle, die der Funktion die Berechtigung erteilt, Elemente aus Ihrer Tabelle zu lesen und zu löschen.



> **Tip**
>
> Wenn Sie neu bei Lambda sind, empfehlen wir Ihnen, vor der Erstellung dieser Beispielanwendung das Tutorial Erstellen Sie Ihre erste Funktion zu absolvieren.

Sie können Ihre App manuell bereitstellen, indem Sie Ressourcen mit dem erstellen und konfigurieren. AWS-Managementkonsole Sie können die App auch mithilfe von AWS Serverless Application Model (AWS SAM) bereitstellen. AWS SAM ist ein Tool für Infrastruktur als Code (IaC). Mit IaC erstellen Sie Ressourcen nicht manuell, sondern definieren sie im Code und stellen sie dann automatisch bereit.

Wenn Sie mehr über die Verwendung von Lambda mit IaC erfahren möchten, bevor Sie diese Beispiel-App bereitstellen, finden Sie unter the section called “Infrastructure as Code (IaC)”.

Voraussetzungen

Bevor Sie die Beispiel-App erstellen können, stellen Sie sicher, dass Sie die erforderlichen Befehlszeilentools und Programme installiert haben.

- Python

  Um die DynamoDB-Tabelle zu füllen, die Sie zum Testen Ihrer App erstellen, verwendet dieses Beispiel ein Python-Skript und eine CSV-Datei, um Daten in die Tabelle zu schreiben. Stellen Sie sicher, dass Python Version 3.8 oder höher auf Ihrem Computer installiert ist.
- AWS SAM CLI

Wenn Sie die DynamoDB-Tabelle erstellen und die Beispiel-App mithilfe bereitstellen möchten AWS SAM, müssen Sie die AWS SAM CLI installieren. Folgen Sie den Installationsanweisungen im AWS SAM Benutzerhandbuch.

- AWS CLI

  Um das bereitgestellte Python-Skript zum Auffüllen Ihrer Testtabelle zu verwenden, müssen Sie das AWS CLI installiert und konfiguriert haben. Das liegt daran, dass das Skript die verwendet AWS SDK für Python (Boto3), die Zugriff auf Ihre AWS Identity and Access Management (IAM-) Anmeldeinformationen benötigt. Sie benötigen außerdem die AWS CLI Installation, um Ressourcen mithilfe AWS SAM von bereitzustellen. Installieren Sie die Befehlszeilenschnittstelle, indem Sie die Installationsanweisungen im AWS Command Line Interface -Benutzerhandbuch befolgen.

- Docker

  Um die App mithilfe von bereitstellen zu können AWS SAM, muss Docker auch auf Ihrem Build-Computer installiert sein. Befolgen Sie die Anweisungen in Installieren der Docker-Engine auf der Docker-Dokumentations-Website.

Die Beispiel-App-Dateien werden heruntergeladen

Um die Beispieldatenbank und die Anwendung für die geplante Wartung zu erstellen, müssen Sie die folgenden Dateien in Ihrem Projektverzeichnis erstellen:

Beispieldatenbankdateien

- `template.yaml`- eine AWS SAM Vorlage, mit der Sie die DynamoDB-Tabelle erstellen können
- `sample_data.csv` - eine CSV-Datei mit Beispieldaten, die Sie in Ihre Tabelle laden können
- `load_sample_data.py` - ein Python-Skript, das die Daten in der CSV-Datei in die Tabelle schreibt

Anwendungsdateien für die planmäßige Wartung

- `lambda_function.py` - den Python-Funktionscode für die Lambda-Funktion, die die Datenbankwartung durchführt
- `requirements.txt` - eine Manifestdatei, die die Abhängigkeiten definiert, die Ihr Python-Funktionscode benötigt

- `template.yaml`- eine AWS SAM Vorlage, die Sie für die Bereitstellung der App verwenden können

Datei testen

- `test_app.py` - ein Python-Skript, das die Tabelle scannt und den erfolgreichen Betrieb Ihrer Funktion bestätigt, indem es alle Datensätze ausgibt, die älter als ein Jahr sind

Erweitern Sie die folgenden Abschnitte, um den Code anzusehen und mehr über die Rolle der einzelnen Dateien beim Erstellen und Testen Ihrer App zu erfahren. Um die Dateien auf Ihrem lokalen Computer zu erstellen, kopieren Sie den folgenden Code und fügen Sie ihn ein.

AWS SAM Vorlage (Beispiel für eine DynamoDB-Tabelle)

Kopieren Sie den folgenden Code und fügen Sie ihn in eine Datei mit dem Namen `template.yaml` ein.

```
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 Vorlagen verwenden eine standardmäßige Benennungskonvention von. `template.yaml` In diesem Beispiel haben Sie zwei Vorlagendateien - eine zur Erstellung der Beispieldatenbank und eine weitere zur Erstellung der Anwendung selbst. Speichern Sie sie in separaten Unterverzeichnissen im Projektordner.

Diese AWS SAM Vorlage definiert die DynamoDB-Tabellenressource, die Sie zum Testen Ihrer App erstellen. Die Tabelle verwendet einen Primärschlüssel von `Order_number` mit einem Sortierschlüssel von `Date`. Damit Ihre Lambda-Funktion Elemente direkt nach Datum finden kann, definieren wir auch einen globalen Sekundärindex mit dem Namen `Date-index`.

Weitere Informationen zum Erstellen und Konfigurieren einer DynamoDB-Tabelle mithilfe der `AWS::DynamoDB::Table`-Ressource finden Sie unter AWS::DynamoDB::Table im AWS CloudFormation Benutzerhandbuch.

Beispiel für Datenbankdatendatei

Kopieren Sie den folgenden Code und fügen Sie ihn in eine Datei mit dem Namen `sample_data.csv` ein.

```
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
```

Diese Datei enthält einige Beispieltestdaten, mit denen Sie Ihre DynamoDB-Tabelle im CSV-Standardformat (durch Komma getrennte Werte) füllen können.

Python-Skript zum Laden von Beispieldaten

Kopieren Sie den folgenden Code und fügen Sie ihn in eine Datei mit dem Namen `load_sample_data.py` ein.

```
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.")
```

Dieses Python-Skript verwendet zunächst die AWS SDK für Python (Boto3) , um eine Verbindung zu Ihrer DynamoDB-Tabelle herzustellen. Anschließend wird jede Zeile in der CSV-Datei mit den Beispieldaten durchlaufen, ein Element aus dieser Zeile erstellt und das Element mithilfe des boto3-SDK in die DynamoDB-Tabelle geschrieben.

Python-Funktionscode

Kopieren Sie den folgenden Code und fügen Sie ihn in eine Datei mit dem Namen `lambda_function.py` ein.

```
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'
    }
```

Der Python-Funktionscode enthält die Handler-Funktion (`lambda_handler`), die Lambda ausführt, wenn Ihre Funktion aufgerufen wird.

Wenn die Funktion vom EventBridge Scheduler aufgerufen wird, verwendet sie die, um eine Verbindung AWS SDK für Python (Boto3) zu der DynamoDB-Tabelle herzustellen, für die die geplante Wartungsaufgabe ausgeführt werden soll. Anschließend wird mit Hilfe der Python `datetime`-Bibliothek das Datum vor einem Jahr berechnet, bevor die Tabelle nach Einträgen durchsucht wird, die älter als dieses Datum sind und diese gelöscht werden.

Beachten Sie, dass Antworten von DynamoDB-Abfrage- und Scanvorgängen auf eine maximale Größe von 1 MB begrenzt sind. Wenn die Antwort größer als 1 MB ist, paginiert DynamoDB die Daten und gibt ein `LastEvaluatedKey`-Element in der Antwort zurück. Um sicherzustellen, dass unsere Funktion alle Datensätze in der Tabelle verarbeitet, prüfen wir, ob dieser Schlüssel vorhanden ist und führen weiterhin Tabellenscans von der zuletzt ausgewerteten Position aus, bis die gesamte Tabelle gescannt wurde.

`requirements.txt`-Manifestdatei

Kopieren Sie den folgenden Code und fügen Sie ihn in eine Datei mit dem Namen `requirements.txt` ein.

```
boto3
```

In diesem Beispiel hat Ihr Funktionscode nur eine Abhängigkeit, die nicht Teil der Standard-Python-Bibliothek ist: das SDK für Python (Boto3), das die Funktion zum Scannen und Löschen von Elementen aus der DynamoDB-Tabelle verwendet.

> **Note**
>
> Eine Version des SDK für Python (Boto3) ist Teil der Lambda-Laufzeit, sodass Ihr Code ausgeführt werden kann, ohne Boto3 zum Bereitstellungspaket Ihrer Funktion hinzuzufügen. Um jedoch die volle Kontrolle über die Abhängigkeiten Ihrer Funktion zu behalten und mögliche Probleme mit Versionsfehlern zu vermeiden, empfiehlt es sich für Python, alle Funktionsabhängigkeiten in das Bereitstellungspaket Ihrer Funktion aufzunehmen. Weitere Informationen hierzu finden Sie unter the section called “Laufzeitabhängigkeiten in Python”.

AWS SAM Vorlage (App für geplante Wartungsarbeiten)

Kopieren Sie den folgenden Code und fügen Sie ihn in eine Datei mit dem Namen `template.yaml` ein.

```
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 Vorlagen verwenden eine Standardbenennungskonvention von. `template.yaml` In diesem Beispiel haben Sie zwei Vorlagendateien - eine zur Erstellung der Beispieldatenbank und eine weitere zur Erstellung der Anwendung selbst. Speichern Sie sie in separaten Unterverzeichnissen im Projektordner.

Diese AWS SAM Vorlage definiert die Ressourcen für Ihre App. Wir definieren die Lambda-Funktion mithilfe der `AWS::Serverless::Function`-Ressource. Der EventBridge Scheduler-Zeitplan und der Trigger zum Aufrufen der Lambda-Funktion werden mithilfe der `Events` Eigenschaft dieser Ressource unter Verwendung eines Typs von erstellt. `ScheduleV2` Weitere Informationen zum Definieren von EventBridge Scheduler-Zeitplänen in AWS SAM Vorlagen finden Sie unter ScheduleV2 im Entwicklerhandbuch.AWS Serverless Application Model

Zusätzlich zur Lambda-Funktion und dem EventBridge Scheduler-Zeitplan definieren wir auch eine CloudWatch Protokollgruppe für Ihre Funktion, an die Datensätze gelöschter Elemente gesendet werden sollen.

Skript testen

Kopieren Sie den folgenden Code und fügen Sie ihn in eine Datei mit dem Namen `test_app.py` ein.

```
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)}")
```

Dieses Testskript verwendet die AWS SDK für Python (Boto3) , um eine Verbindung zu Ihrer DynamoDB-Tabelle herzustellen und nach Elementen zu suchen, die älter als ein Jahr sind. Um zu überprüfen, ob die Lambda-Funktion erfolgreich ausgeführt wurde, druckt die Funktion am Ende des Tests die Anzahl der Datensätze, die älter als ein Jahr sind, noch in der Tabelle aus. Wenn die Lambda-Funktion erfolgreich war, sollte die Anzahl der alten Datensätze in der Tabelle Null sein.

DynamoDB-Beispieltabelle erstellen und auffüllen

Um Ihre App für geplante Wartungsarbeiten zu testen, erstellen Sie zunächst eine DynamoDB-Tabelle und füllen sie mit einigen Beispieldaten. Sie können die Tabelle entweder manuell mithilfe von AWS-Managementkonsole oder mithilfe von AWS SAM erstellen. Wir empfehlen Ihnen, die Tabelle mithilfe einiger AWS CLI Befehle schnell AWS SAM zu erstellen und zu konfigurieren.

Console

Erstellen einer DynamoDB-Tabelle

1. Öffnen Sie die Seite Tables (Tabellen) in der DynamoDB-Konsole.
2. Wählen Sie Create table (Tabelle erstellen) aus.
3. Erstellen Sie die Tabelle folgendermaßen:
    a. Geben Sie unter Tabellendetails für Tabellenname den Wert **`MyOrderTable`** ein.
    b. Geben Sie für Partitionsschlüssel **`Order_number`** ein und behalten Sie den Datentyp als Zeichenfolge bei.

c. Geben Sie für Sortierschlüssel **`Date`** ein und behalten Sie den Datentyp als Zeichenfolge bei.

d. Lassen Sie die Tabelleneinstellungen auf Standardeinstellungen eingestellt und wählen Sie Tabelle erstellen.

4. Wenn Ihre Tabelle mit der Erstellung fertig ist und ihr Status als Aktiv angezeigt wird, erstellen Sie einen globalen sekundären Index (GSI), indem Sie wie folgt vorgehen. Ihre App verwendet diesen globalen Index, um direkt anhand des Datums nach Elementen zu suchen, um zu bestimmen, welche gelöscht werden sollen.

   a. Wählen Sie MyOrderTableaus der Liste der Tabellen aus.

   b. Wählen Sie die Registerkarte Indizes.

   c. Wählen Sie unter Globale Sekundärindizes die Option Index erstellen aus.

   d. Geben Sie unter Indexdetails **`Date`** für den Partitionsschlüssel ein und lassen Sie den Datentyp auf String eingestellt.

   e. Geben Sie für Indexname **`Date-index`** ein.

   f. Verwenden Sie für alle anderen Parameter die Standardwerte, scrollen Sie ans Seitenende und wählen Sie Index erstellen aus.

AWS SAM

Erstellen einer DynamoDB-Tabelle

1. Navigieren Sie zu dem Ordner, in dem Sie die `template.yaml`-Datei für die DynamoDB-Tabelle gespeichert haben. Beachten Sie, dass in diesem Beispiel zwei `template.yaml`-Dateien verwendet werden. Vergewissern Sie sich, dass sie in separaten Unterordnern gespeichert sind und dass Sie sich im richtigen Ordner befinden, der die Vorlage zur Erstellung Ihrer DynamoDB-Tabelle enthält.
2. Führen Sie den folgenden Befehl aus.

   ```
   sam build
   ```

   Dieser Befehl sammelt die Build-Artefakte für die Ressourcen, die Sie bereitstellen möchten und platziert sie im richtigen Format und am richtigen Ort, um sie bereitzustellen.
3. Führen Sie den folgenden Befehl aus, um die in der `template.yaml`-Datei angegebene DynamoDB-Ressource zu erstellen.

```
sam deploy --guided
```

Wenn Sie die `--guided` Flagge verwenden, AWS SAM werden Ihnen Eingabeaufforderungen angezeigt, die Sie durch den Bereitstellungsprozess führen. Geben Sie für diese Bereitstellung ein `Stack name` von **`cron-app-test-db`** ein und akzeptieren Sie die Standardeinstellungen für alle anderen Optionen mit der Eingabetaste.

Wenn AWS SAM die Erstellung der DynamoDB-Ressource abgeschlossen ist, sollte die folgende Meldung angezeigt werden.

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

4. Sie können außerdem überprüfen, ob die DynamoDB-Tabelle erstellt wurde, indem Sie die Tabellenseite der DynamoDB-Konsole öffnen. Sie sollten eine Tabelle mit dem Namen `MyOrderTable` sehen.

Nachdem Sie Ihre Tabelle erstellt haben, fügen Sie als Nächstes einige Beispieldaten hinzu, um Ihre App zu testen. Die CSV-Datei `sample_data.csv`, die Sie zuvor heruntergeladen haben, enthält eine Reihe von Beispieleinträgen, die aus Bestellnummern, Daten sowie Kunden- und Bestellinformationen bestehen. Verwenden Sie das bereitgestellte Python-Skript `load_sample_data.py`, um diese Daten zu Ihrer Tabelle hinzuzufügen.

So fügen Sie der Tabelle die Beispieldaten hinzu

1. Wechseln Sie in das Verzeichnis, das die `sample_data.csv`- und `load_sample_data.py`-Dateien enthält. Wenn sich diese Dateien in separaten Verzeichnissen befinden, verschieben Sie sie so, dass sie am selben Ort gespeichert werden.
2. Erstellen Sie eine virtuelle Python-Umgebung zur Ausführung des Skripts, indem Sie den folgenden Befehl ausführen. Wir empfehlen Ihnen, eine virtuelle Umgebung zu verwenden, da Sie im nächsten Schritt das AWS SDK für Python (Boto3) installieren müssen.

   ```
   python -m venv venv
   ```

3. Aktivieren Sie die virtuelle Umgebung, indem Sie den folgenden Befehl ausführen.

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

4. Installieren Sie das SDK für Python (Boto3) in Ihrer virtuellen Umgebung, indem Sie den folgenden Befehl ausführen. Das Skript verwendet diese Bibliothek, um eine Verbindung zu Ihrer DynamoDB-Tabelle herzustellen und die Elemente hinzuzufügen.

```
pip install boto3
```

5. Führen Sie das Skript aus, um die Tabelle zu füllen, indem Sie den folgenden Befehl ausführen.

```
python load_sample_data.py
```

   Wenn das Skript erfolgreich läuft, sollte es jedes Element beim Laden auf der Konsole ausgeben und `Data loading completed` melden.

6. Deaktivieren Sie die virtuelle Umgebung, indem Sie den folgenden Befehl ausführen.

```
deactivate
```

7. Sie können überprüfen, ob die Daten in Ihre DynamoDB-Tabelle geladen wurden, indem Sie Folgendes tun:

   a. Öffnen Sie die Seite Elemente untersuchen der DynamoDB-Konsole und wählen Sie Ihre Tabelle (`MyOrderTable`).

   b. Im Bereich Zurückgegebene Elemente sollten Sie die 25 Elemente aus der CSV-Datei sehen, die das Skript der Tabelle hinzugefügt hat.

Die App für geplante Wartungsarbeiten erstellen

Sie können die Ressourcen für diese Beispiel-App Schritt für Schritt mithilfe von AWS-Managementkonsole oder mithilfe von erstellen und bereitstellen. AWS SAM In einer Produktionsumgebung empfehlen wir die Verwendung eines Infrustracture-as-Code (IaC-) Tools, mit dem Sie wiederholt serverlose Anwendungen bereitstellen können, ohne manuelle Prozesse AWS SAM zu verwenden.

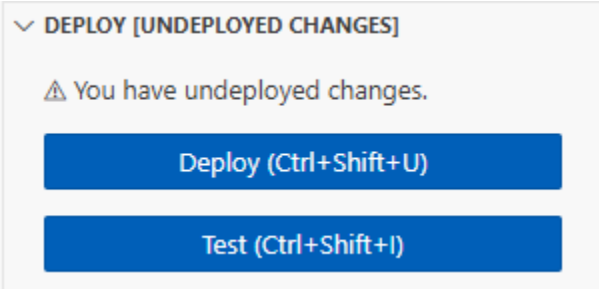
Folgen Sie in diesem Beispiel den Anweisungen auf der Konsole, um zu erfahren, wie Sie jede AWS Ressource separat konfigurieren, oder folgen Sie den AWS SAM Anweisungen, um die App mithilfe von Befehlen schnell bereitzustellen. AWS CLI

Console

Um die Funktion mit dem zu erstellen AWS-Managementkonsole

Erstellen Sie zunächst eine Funktion, die den grundlegenden Startercode enthält. Anschließend ersetzen Sie diesen Code durch Ihren eigenen Funktionscode, indem Sie den Code entweder kopieren und direkt im Lambda-Code-Editor einfügen oder Ihren Code als `.zip`-Paket hochladen. Für diese Aufgabe empfehlen wir, den Code zu kopieren und einzufügen.

1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.
2. Wählen Sie Funktion erstellen.
3. Wählen Sie Author from scratch aus.
4. Führen Sie unter Basic information (Grundlegende Informationen) die folgenden Schritte aus:
   a. Geben Sie für Function name (Funktionsname) **`ScheduledDBMaintenance`** ein.
   b. Für Laufzeit wählen Sie die neueste Python-Version.
   c. Wählen Sie für Architektur x86_64 aus.
5. Wählen Sie Funktion erstellen.
6. Nachdem Ihre Funktion erstellt wurde, können Sie Ihre Funktion mit dem bereitgestellten Funktionscode konfigurieren.
   a. Ersetzen Sie im Bereich Codequelle den Hello World-Code, den Lambda erstellt hat, durch den Python-Funktionscode aus der `lambda_function.py`-Datei, die Sie zuvor gespeichert haben.
   b. Wählen Sie im Abschnitt BEREITSTELLEN die Option Bereitstellen aus, um den Code Ihrer Funktion zu aktualisieren:



Um den Funktionsspeicher und das Timeout (Konsole) zu konfigurieren

1. Wählen Sie die Registerkarte Konfiguration für Ihre Funktion.

2. Wählen Sie im Bereich Allgemeine Konfiguration die Option Bearbeiten aus.
3. Stellen Sie Speicher auf 256 MB und Timeout auf 15 Sekunden ein. Wenn Sie eine große Tabelle mit vielen Datensätzen verarbeiten, z. B. in einer Produktionsumgebung, sollten Sie in Erwägung ziehen, Timeout auf einen höheren Wert festzulegen. Dadurch hat Ihre Funktion mehr Zeit zum Scannen und Bereinigen der Datenbank.
4. Wählen Sie Speichern.

So konfigurieren Sie das Protokollformat (Konsole)

Sie können Lambda-Funktionen so konfigurieren, dass sie Protokolle entweder im unstrukturierten Text- oder im JSON-Format ausgeben. Wir empfehlen, das JSON-Format für Protokolle zu verwenden, um die Protokolldaten einfacher durchsuchen und filtern zu können. Weitere Informationen über die Optionen für die Konfiguration des Lambda-Protokolls finden Sie unter the section called “Konfigurieren erweiterter Protokollierungsoptionen für Lambda-Funktionen”.

1. Wählen Sie die Registerkarte Konfiguration für Ihre Funktion.
2. Wählen Sie Überwachungs- und Betriebstools aus.
3. Wählen Sie im Bereich Protokollierungskonfiguration die Option Bearbeiten aus.
4. Wählen Sie für die Konfiguration der Protokollierung JSON aus.
5. Wählen Sie Speichern.

So richten Sie IAM-Berechtigungen ein

Um Ihrer Funktion die erforderlichen Berechtigungen zum Lesen und Löschen von DynamoDB-Elementen zu erteilen, müssen Sie der Ausführungsrolle Ihrer Funktion eine Richtlinie hinzufügen, die die erforderlichen Berechtigungen definiert.

1. Öffnen Sie die Registerkarte Konfiguration und wählen Sie dann in der linken Navigationsleiste Berechtigungen aus.
2. Wählen Sie den Rollennamen unter Ausführungsrolle.
3. Wählen Sie in der IAM-Konsole Berechtigungen hinzufügen und dann Inline-Richtlinie erstellen aus.
4. Verwenden Sie den JSON-Editor und geben Sie die folgenden Richtlinien ein:

```
{
```

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

5. Geben Sie der Richtlinie **DynamoDBCleanupPolicy** einen Namen und erstellen Sie sie dann.

Um den EventBridge Scheduler als Trigger einzurichten (Konsole)

1. Öffnen Sie die EventBridge -Konsole.
2. Wählen Sie im linken Navigationsbereich im Bereich Scheduler die Option Schedulers aus.
3. Wählen Sie Zeitplan erstellen aus.
4. Konfigurieren Sie den Zeitplan, indem Sie wie folgt vorgehen:
   a. Geben Sie unter Zeitplanname einen Namen für Ihren Zeitplan ein (z. B. **DynamoDBCleanupSchedule**).
   b. Wählen Sie unter Zeitplanmuster die Option Wiederkehrender Zeitplan aus.
   c. Belassen Sie für Zeitplantyp die Standardeinstellung Cron-basierter Zeitplan und geben Sie dann die folgenden Zeitplandetails ein:
      - Minuten: **0**
      - Stunden: **3**
      - Tag des Monats: **1**
      - Monat: *****
      - Wochentag: **?**
      - Jahr: *****

Bei der Auswertung wird dieser Cron-Ausdruck am ersten Tag jedes Monats um 03:00 Uhr ausgeführt.

d. Wählen Sie für Flexibles Zeitfenster die Option Aus.

5. Wählen Sie Weiter aus.
6. Konfigurieren Sie den Auslöser für Ihre Lambda-Funktion folgendermaßen:

   a. Lassen Sie im Bereich Zieldetails die Einstellung Ziel-API auf Vorgefertigte Ziele eingestellt und wählen Sie dann AWS Lambda aufrufen aus.

   b. Wählen Sie unter Aufrufen Ihre Lambda-Funktion (`ScheduledDBMaintenance`) aus der Dropdown-Liste aus.

   c. Lassen Sie das Feld Payload leer und wählen Sie Weiter.

   d. Scrollen Sie nach unten zu Berechtigungen und wählen Sie Neue Rolle für diesen Zeitplan erstellen aus. Wenn Sie mit der Konsole einen neuen EventBridge Scheduler-Zeitplan erstellen, erstellt EventBridge Scheduler eine neue Richtlinie mit den erforderlichen Berechtigungen, die der Zeitplan benötigt, um Ihre Funktion aufzurufen. Weitere Informationen zur Verwaltung Ihrer Zeitplanberechtigungen finden Sie unter Cron-basierte Zeitpläne. im Scheduler-Benutzerhandbuch. EventBridge

   e. Wählen Sie Weiter aus.

7. Überprüfen Sie Ihre Einstellungen und wählen Sie Zeitplan erstellen, um die Erstellung des Zeitplans und des Lambda-Triggers abzuschließen.

AWS SAM

So stellen Sie die App bereit mit AWS SAM

1. Navigieren Sie zu dem Ordner, in dem Sie die `template.yaml`-Datei für die App gespeichert haben. Beachten Sie, dass in diesem Beispiel zwei `template.yaml`-Dateien verwendet werden. Vergewissern Sie sich, dass sie in separaten Unterordnern gespeichert sind und dass Sie sich im richtigen Ordner befinden, der die Vorlage zur Erstellung Ihrer App enthält.
2. Kopieren Sie die `lambda_function.py`- und `requirements.txt`-Dateien, die Sie zuvor heruntergeladen haben, in denselben Ordner. Der in der AWS SAM Vorlage angegebene Codespeicherort ist `./`, d. h. der aktuelle Standort. AWS SAM sucht in diesem Ordner nach dem Lambda-Funktionscode, wenn Sie versuchen, die App bereitzustellen.

3. Führen Sie den folgenden Befehl aus.

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

Dieser Befehl sammelt die Build-Artefakte für die Ressourcen, die Sie bereitstellen möchten und platziert sie im richtigen Format und am richtigen Ort, um sie bereitzustellen. Wenn Sie die Option `--use-container` angeben, wird Ihre Funktion innerhalb eines Lambda-ähnlichen Docker-Containers erstellt. Wir verwenden es hier, sodass Sie Python 3.12 nicht auf Ihrem lokalen Computer installiert haben müssen, damit der Build funktioniert.

4. Führen Sie den folgenden Befehl aus, um die in der `template.yaml` Datei angegebenen Lambda- und EventBridge Scheduler-Ressourcen zu erstellen.

```
sam deploy --guided
```

Wenn Sie die `--guided` Markierung verwenden, AWS SAM werden Ihnen Eingabeaufforderungen angezeigt, die Sie durch den Bereitstellungsprozess führen. Geben Sie für diese Bereitstellung ein `Stack name` von **`cron-maintenance-app`** ein und akzeptieren Sie die Standardeinstellungen für alle anderen Optionen mit der Eingabetaste.

Wenn AWS SAM die Erstellung der Lambda- und EventBridge Scheduler-Ressourcen abgeschlossen ist, sollte die folgende Meldung angezeigt werden.

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

5. Sie können außerdem überprüfen, ob die Lambda-Funktion erstellt wurde, indem Sie die Seite Funktionen der Lambda-Konsole öffnen. Sie sollten eine Funktion mit dem Namen `ScheduledDBMaintenance` sehen.

Testen der App

Um zu testen, ob Ihr Zeitplan Ihre Funktion korrekt auslöst und ob Ihre Funktion die Datensätze in der Datenbank korrekt bereinigt, können Sie Ihren Zeitplan vorübergehend so ändern, dass er einmal zu einer bestimmten Zeit ausgeführt wird. Sie können dann `sam deploy` erneut ausführen, um Ihren Wiederholungsplan so zurückzusetzen, dass er einmal pro Monat ausgeführt wird.

Um die Anwendung mit dem auszuführen AWS-Managementkonsole

1. Navigieren Sie zurück zur EventBridge Scheduler-Konsolenseite.

2. Wählen Sie Ihren Zeitplan aus und klicken Sie dann auf Bearbeiten.
3. Wählen Sie im Abschnitt Zeitplanmuster unter Wiederholung die Option Einmaliger Zeitplan aus.
4. Stellen Sie Ihre Aufrufzeit auf ein paar Minuten ein, überprüfen Sie Ihre Einstellungen und wählen Sie dann Speichern.

Nachdem der Zeitplan ausgeführt und sein Ziel aufgerufen wurde, führen Sie das `test_app.py`-Skript aus, um zu überprüfen, ob Ihre Funktion erfolgreich alle alten Datensätze aus der DynamoDB-Tabelle entfernt hat.

Um zu überprüfen, ob alte Datensätze mit einem Python-Skript gelöscht wurden

1. Navigieren Sie in Ihrer Befehlszeile zu dem Ordner, in dem Sie `test_app.py` gespeichert haben.
2. Führen Sie das Skript aus.

   ```
   python test_app.py
   ```

   Bei Erfolg erhalten Sie die folgende Ausgabe.

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

Nächste Schritte

Sie können den EventBridge Scheduler-Zeitplan jetzt an Ihre speziellen Anwendungsanforderungen anpassen. EventBridge Scheduler unterstützt die folgenden Zeitplanausdrücke: Cron, Rate und One-Time Schedules.

Weitere Informationen zu EventBridge Scheduler-Zeitplanausdrücken finden Sie unter Zeitplantypen im EventBridge Scheduler-Benutzerhandbuch. Zugriffsverwaltung im IAM-Benutzerhandbuch

Erstellung eines Auftragsabwicklungssystems mit Lambda Durable-Funktionen

> **Note**
>
> ERFORDERLICH: Fügen Sie ein Architekturdiagramm hinzu, das das API Gateway, den Durable Function-Workflow und die unterstützenden Dienste zeigt (DynamoDB,) EventBridge

Voraussetzungen

- AWS CLI installiert und konfiguriert
- BEDARF: Spezifische Anforderungen an langlebige Funktionen

Erstellen Sie die Quellcodedateien

Erstellen Sie die folgenden Dateien in Ihrem Projektverzeichnis:

- `lambda_function.py`- der Funktionscode
- `requirements.txt`- Abhängigkeiten manifestieren sich

Funktionscode

```
# 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
```

Anforderungsdatei

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

Bereitstellen der Anwendung

Erstellen Sie eine DynamoDB-Tabelle für Bestellungen

1. Öffnen Sie die DynamoDB-Konsole unter https://console.aws.amazon.com/dynamodb/
2. Wählen Sie Tabelle erstellen
3. Geben Sie als Tabellenname ein **`Orders`**
4. Geben Sie für Partitionsschlüssel Folgendes ein **`orderId`**
5. Belassen Sie andere Einstellungen als Standard
6. Wählen Sie Tabelle erstellen

So erstellen Sie die Lambda-Funktion:

1. Öffnen Sie die Lambda-Konsole unter https://console.aws.amazon.com/lambda/
2. Wählen Sie Create function (Funktion erstellen).
3. Wählen Sie Autor von Grund auf
4. Geben Sie unter Funktionsname **`ProcessOrder`** ein.
5. Wählen Sie für Runtime Ihre bevorzugte Laufzeit
6. ERFORDERLICH: Fügen Sie eine für Durable Functions spezifische Konfiguration hinzu

7. Wählen Sie Create function (Funktion erstellen).

Erstellen Sie den API-Gateway-Endpunkt

1. Öffnen Sie die API Gateway Gateway-Konsole unter https://console.aws.amazon.com/apigateway/
2. Wählen Sie Create API
3. Wählen Sie HTTP-API
4. Wählen Sie Build
5. Fügen Sie eine Integration mit Ihrer Lambda-Funktion hinzu
6. Konfigurieren Sie Routen für die Auftragsabwicklung
7. Bereitstellen der API

Testen der Anwendung

Reichen Sie eine Testbestellung ein:

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

ERFORDERLICH: Fügen Sie spezifische Überwachungsanweisungen für langlebige Funktionen hinzu

Nächste Schritte

Geschäftslogik hinzufügen

Implementieren Sie die Inventarverwaltung:

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

```
    pass
```

Preisberechnungen hinzufügen:

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

Verbessern Sie die Fehlerbehandlung

Vergütungslogik hinzufügen:

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

Stornierungen von Bestellungen bearbeiten:

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

Integrieren Sie externe Systeme

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

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

Verbessern Sie die Überwachung

- Erstellen Sie CloudWatch Dashboards
- Richten Sie Kennzahlen für die Bearbeitungszeiten von Bestellungen ein
- Konfigurieren Sie Benachrichtigungen für verspätete Bestellungen

Entwicklungstools für Lambda

Sie haben Zugriff auf eine Vielzahl von Tools, die die Produktivität und Benutzerfreundlichkeit während des gesamten Entwicklungszyklus steigern. Dieser Abschnitt enthält Informationen zu Tools, die vielen Lambda-Kunden beim Entwerfen, Entwickeln und Verwalten ihrer Anwendungen helfen. Von der lokalen Entwicklung in Ihrer IDE bis hin zur Bereitstellung und Verwaltung komplexer Serverless-Anwendungen helfen Ihnen diese Tools dabei, Ihren Workflow zu optimieren, die Codequalität zu verbessern und die Entwicklung robuster Lambda-basierter Lösungen zu beschleunigen.

- Lokale Entwicklung: Schreiben und testen Sie Lambda-Funktionen schneller in Ihrer bevorzugten Entwicklungsumgebung. Das AWS-Toolkit für VS Code ermöglicht die lokale Funktionsentwicklung, Fehlerbehebung und Tests mit direkten Bereitstellungsfunktionen für Lambda.
- Infrastructure as Code (IaC): Stellen Sie Serverless-Anwendungen vom lokalen Testen bis zur Produktion konsistent bereit und verwalten Sie sie. AWS SAM, AWS CDK und CloudFormation ermöglichen es Ihnen, Ihre Serverless-Infrastruktur mithilfe von Code für konsistente, versionskontrollierte Bereitstellungen zu definieren und zu verwalten.
- GitHub Actions: Automatisieren Sie Lambda-Bereitstellungen direkt aus Ihrem Code-Repository. Mit GitHub Actions können Sie Workflows einrichten, die Ihre Lambda-Funktionen automatisch bereitstellen, sobald Sie Code- oder Konfigurationsänderungen übertragen, wodurch Ihre CI/CD-Pipeline vereinfacht wird.
- Powertools for AWS Lambda: Erstellen Sie produktionsreife Serverless-Anwendungen mit weniger benutzerdefiniertem Code. Powertools for AWS Lambda (auch als Powertools for AWS bezeichnet) ist ein Open-Source-Entwickler-Toolkit, das die Implementierung von Best Practices für Serverless-Anwendungen wie Beobachtbarkeit, Parameterabruf und Idempotenz in Python, TypeScript, Java und .NET vereinfacht.
- Workflows und Ereignisse: Koordinieren Sie Lambda-Funktionen mit AWS-Services, APIs und externen Systemen. AWS Step Functions und Amazon EventBridge bieten Orchestrierungs- und Ereignisverwaltungsfunktionen zur Koordinierung komplexer Lambda-basierter Anwendungsworkflows.

Lokale Entwicklungstools

Lokale Entwicklungsumgebungen ermöglichen es Ihnen, offline zu arbeiten und erweiterte IDE-Funktionen zu nutzen, während Sie Ihre Lambda-Funktionen schnell iterieren. Diese Tools

helfen Ihnen beim Debuggen komplexer Funktionen und bei der Entwicklung in Umgebungen mit eingeschränkter Konnektivität. Sie unterstützen auch die Zusammenarbeit im Team und die Integration mit Versionskontrollsystemen.

Weitere Informationen zum lokalen Entwickeln von Lambda-Funktionen finden Sie unter Lokales Entwickeln von Lambda-Funktionen mit VS Code. Auf dieser Seite wird beschrieben, wie Sie die Entwicklung von Lambda-Funktionen von der AWS-Konsole zu Visual Studio Code verlagern können, das eine umfangreiche Entwicklungsumgebung mit Features wie Debugging und Code-Vervollständigung bietet. Für die Umstellung müssen Sie das AWS Toolkit for Visual Studio Code und die Anmeldeinformationen einrichten. Anschließend können Sie die erweiterten Funktionen in VS Code nutzen und gleichzeitig weiterhin direkt auf AWS bereitstellen.

Die lokale Entwicklung für Lambda bietet mehrere wichtige Funktionen:

- Verwenden der Visual-Studio-Code-Integration mit der Lambda-Konsole
- Konfigurieren lokaler Lambda-Entwicklungsumgebungen
- Lokales Debuggen und Testen von Funktionen
- Anwenden von Best Practices für lokale Funktionsverwaltung

Weitere Informationen finden Sie unter Lokales Entwickeln von Lambda-Funktionen mit VS Code.

Infrastructure as Code (IaC)-Tools

Mit den Tools für Infrastructure as Code (IaC) können Sie Ihre Serverless-Architektur mithilfe von Code definieren und verwalten. Dieser Ansatz trägt dazu bei, die Konsistenz über verschiedene Umgebungen hinweg aufrechtzuerhalten, ermöglicht Ihnen die Kontrolle über Ihre Infrastrukturversionen und erleichtert DevOps-Praktiken. IaC ist besonders wertvoll für die Automatisierung von Bereitstellungen, die Gewährleistung konsistenter Umgebungen und die Verwaltung von Bereitstellungen in mehreren Regionen.

Zu den wichtigsten IaC-Tools und -Konzepten für Lambda gehören Frameworks für die Erstellung von Vorlagen, das Bereitstellungsmanagement und Best Practices für Serverless-Infrastrukturen:

- Kernprinzipien von IaC für die Lambda-Entwicklung
- CloudFormation-, AWS SAM- und AWS CDK-Funktionen
- Auswahlkriterien und Vergleich von Tools
- Best Practices für die Implementierung von Lambda IaC

Ganz gleich, ob Sie selbstständig an einem kleinen Projekt arbeiten oder als Teil eines großen Teams unternehmensweite Serverless-Anwendungen verwalten – diese Entwicklungs- und Bereitstellungstools helfen Ihnen dabei, Ihre Lambda-Funktionen effektiver zu schreiben, bereitzustellen und zu verwalten.

Weitere Informationen finden Sie unter Verwenden von Lambda mit Infrastructure as Code (IaC).

GitHub-Actions-Tools

GitHub Actions bietet automatisierte Bereitstellungsfunktionen für Ihre Lambda-Funktionen direkt aus Ihrem Code-Repository. Durch das Erstellen von Workflow-Dateien in Ihrem Repository können Sie Lambda-Funktionen automatisch bereitstellen, sobald Code- oder Konfigurationsänderungen übertragen werden, und so Ihre Pipeline für kontinuierliche Integration und kontinuierliche Bereitstellung (CI/CD) optimieren. Die Aktion „Lambda-Funktion bereitstellen“ bietet eine deklarative YAML-Schnittstelle, die den Bereitstellungsprozess vereinfacht, AWS-Anmeldeinformationen über OpenID Connect (OIDC) verarbeitet und verschiedene Bereitstellungsszenarien unterstützt, darunter Code-Updates, Konfigurationsänderungen und Testlauf-Validierungen. Diese Integration ermöglicht es Teams, einen konsistenten und automatisierten Bereitstellungsprozess aufrechtzuerhalten und gleichzeitig ihre bestehenden GitHub-Workflows zu nutzen.

Weitere Informationen finden Sie unter Verwenden von GitHub Actions zur Bereitstellung von Lambda-Funktionen.

Powertools for AWS Lambda

Powertools for AWS ist ein Open-Source-Entwickler-Toolkit, mit dem Sie Serverless-Best-Practices mit minimalem benutzerdefiniertem Code implementieren können. Es ist für Python, TypeScript/Node.js, Java und .NET verfügbar und bietet Hilfsfunktionen, Decoratoren und Middleware, die gängige Lambda-Entwicklungsaufgaben optimieren. Das Toolkit umfasst integrierte Funktionen für die Beobachtbarkeit wie strukturierte Protokollierung, Tracing und Metrikenerfassung, darunter Dienstprogramme für die Parameterabfrage, die Verwaltung geheimer Schlüssel und Idempotenzmuster. Diese Tools entsprechen den Best Practices für eine gut konzipierte AWS-Architektur und helfen Entwicklern dabei, produktionsreife Serverless-Anwendungen effizienter zu erstellen. Durch die Reduzierung von Boilerplate-Code und die Standardisierung gängiger Muster ermöglicht Powertools for AWS Teams, sich auf die Geschäftslogik zu konzentrieren und gleichzeitig die konsistente Umsetzung von Serverless-Best-Practices in ihren Anwendungen sicherzustellen.

Weitere Informationen finden Sie unter Powertools for AWS Lambda.

Tools für die Workflow- und Ereignisverwaltung

Lambda-Anwendungen können zur Orchestrierung komplexer Workflows und zur Verarbeitung verschiedener Ereignisse eingesetzt werden. AWS bietet spezielle Tools, die Ihnen bei der Verwaltung dieser Aspekte der Serverless-Entwicklung helfen. Erfahren Sie mehr über AWS Step Functions für die Workflow-Orchestrierung und Amazon EventBridge für die Ereignisverwaltung. Außerdem lernen Sie, wie Sie diese in Ihre Lambda-Funktionen integrieren können. Diese Tools können die Skalierbarkeit und Zuverlässigkeit Ihrer Serverless-Anwendungen erheblich verbessern, indem sie eine robuste Zustandsverwaltung und ereignisgesteuerte Architekturen bieten. Durch die Nutzung dieser Services können Sie anspruchsvollere und widerstandsfähigere Lambda-basierte Lösungen entwickeln, die komplexe Geschäftsprozesse bewältigen und auf eine Vielzahl von System- und Anwendungsereignissen reagieren können.

Weitere Informationen finden Sie unter Verwaltung von Lambda-Workflows und -Ereignissen.

Lokales Entwickeln von Lambda-Funktionen mit VS Code

Sie können Ihre Lambda-Funktionen von der Lambda-Konsole in Visual Studio Code verschieben. Dieser Service bietet eine vollständige Entwicklungsumgebung und ermöglicht Ihnen die Verwendung anderer lokaler Entwicklungsoptionen wie AWS SAM und AWS CDK.

Hauptvorteile der lokalen Entwicklung

Während die Lambda-Konsole eine schnelle Möglichkeit zum Bearbeiten und Testen von Funktionen bietet, verfügt die lokale Entwicklung über erweiterte Funktionen:

- Erweiterte IDE-Features: Tools für Debugging, Codevervollständigung und Faktorwechsel
- Offline-Entwicklung: lokales Arbeiten und Testen von Änderungen vor der Cloud-Bereitstellung
- Integration von Infrastructure as Code: nahtlose Verwendung mit AWS SAM, AWS CDK und Infrastructure Composer
- Abhängigkeitsmanagement: vollständige Kontrolle über Funktionsabhängigkeiten

Voraussetzungen

Vor der lokalen Entwicklung von Lambda-Funktionen in VS Code benötigen Sie Folgendes:

- VS Code: Eine Anleitung zum Installieren finden Sie unter VS Code herunterladen.

- AWS Toolkit for Visual Studio Code: Eine Anleitung zum Installieren finden Sie unter Einrichten von AWS Toolkit for Visual Studio Code. Eine Übersicht finden Sie unter AWS Toolkit for Visual Studio Code.
- AWS-Anmeldeinformationen: Informationen zur Konfiguration von Anmeldeinformationen finden Sie unter Einrichten Ihrer AWS-Anmeldeinformationen.
- AWS SAM CLI: Eine Anleitung zum Installieren finden Sie unter Installieren der AWS SAM CLI.
- Docker installiert (optional, aber für lokale Tests erforderlich): Eine Anleitung zum Installieren finden Sie unter Docker erhalten.

> **Note**
>
> Wenn Sie bereits ein AWS-Konto und ein Profil lokal konfiguriert haben, stellen Sie sicher, dass die verwaltete Richtlinie „AdministratorAccess“ zu Ihrem konfigurierten AWS-Profil hinzugefügt wurde.

Authentifizierung und Zugriffskontrolle

Für das lokale Entwickeln von Lambda-Funktionen benötigen Sie AWS-Anmeldeinformationen, um in Ihrem Namen sicher auf AWS-Ressourcen zuzugreifen und diese zu verwalten, genau wie in der Cloud. Das AWS-Toolkit for VS Code unterstützt die folgenden Authentifizierungsmethoden:

Das AWS-Toolkit for VS Code unterstützt die folgenden Authentifizierungsmethoden:

- Langfristige Anmeldeinformationen für IAM-Benutzer
- Temporäre Anmeldeinformationen von angenommenen Rollen
- Identitätsverbund
- Anmeldeinformationen für den Root-Benutzer des AWS-Kontos (nicht empfohlen)

Dieser Abschnitt führt Sie durch den Prozess zum Abrufen und Konfigurieren dieser Anmeldeinformationen mithilfe der langfristigen Anmeldeinformationen für IAM-Benutzer.

Erhalt von IAM-Anmeldeinformationen

Sie haben bereits einen IAM-Benutzer mit Zugriffsschlüsseln? Halten Sie die Zugriffsschlüssel-ID und den geheimen Zugriffsschlüssel für den nächsten Abschnitt bereit. Wenn nicht, erstellen Sie diese Elemente wie folgt:

> **Note**
>
> Sie müssen die Zugriffsschlüssel-ID und den geheimen Zugriffsschlüssel zusammen verwenden, um Ihre Anforderungen zu authentifizieren.

So erstellen Sie einen IAM-Benutzer und Zugriffsschlüssel:

1. Öffnen Sie die IAM-Konsole unter https://console.aws.amazon.com/iam/.
2. Klicken Sie im Navigationsbereich auf Users (Benutzer).
3. Wählen Sie Create user (Benutzer erstellen) aus.
4. Geben Sie unter Benutzername einen Namen ein und wählen Sie Weiter.
5. Wählen Sie unter Berechtigungen festlegen die Option Richtlinien direkt anhängen.
6. Wählen Sie AdministratorAccess und dann Weiter.
7. Wählen Sie Create user (Benutzer erstellen) aus.
8. Wählen Sie im Banner mit der Erfolgsmeldung die Option Benutzer anzeigen.
9. Wählen Sie Zugriffsschlüssel erstellen aus.
10. Wählen Sie für Anwendungsfall die Option Lokaler Code.
11. Aktivieren Sie das Kontrollkästchen für die Bestätigung und wählen Sie Weiter.
12. (Optional) Geben Sie einen Wert für das Beschreibungs-Tag ein.
13. Wählen Sie Zugriffsschlüssel erstellen aus.
14. Kopieren Sie sofort Ihren Zugriffsschlüssel und den geheimen Zugriffsschlüssel. Sie können nach dem Verlassen dieser Seite nicht mehr auf den geheimen Zugriffsschlüssel zugreifen.

> **Important**
>
> Geben Sie Ihren geheimen Schlüssel niemals weiter und übertragen Sie ihn nicht in die Quellcodeverwaltung. Bewahren Sie diese Schlüssel sicher auf und löschen Sie sie, wenn sie nicht mehr benötigt werden.

> **Note**
>
> Weitere Informationen finden Sie unter Erstellen eines IAM-Benutzers in Ihrem AWS-Konto und Verwalten von Zugriffsschlüsseln für IAM-Benutzer im IAM-Benutzerhandbuch.

Konfigurieren von AWS-Anmeldeinformationen mit dem AWS-Toolkit

In der folgenden Tabelle ist der Prozess zur Einrichtung der Anmeldedaten zusammengefasst, den Sie im folgenden Verfahren durchführen werden.

Vorgehensweise	Warum?
Öffnen Sie das Anmeldefenster.	Starten der Authentifizierung
Suchen Sie in der Befehlspalette nach AWS „Neue Verbindung hinzufügen“.	Greifen Sie auf die Anmeldebenutzeroberfläche zu.
Wählen Sie die IAM-Anmeldeinformationen aus.	Verwenden Sie Ihren Zugriffsschlüssel für den programmatischen Zugriff.
Geben Sie den Profilnamen, den Zugriffss chlüssel und den geheimen Schlüssel ein.	Geben Sie die Anmeldeinformationen für die Verbindung ein.
Siehe AWS-Explorer-Update.	Bestätigen Sie, dass Sie verbunden sind.

Führen Sie die folgenden Schritte aus, um sich bei Ihrem AWS-Konto zu authentifizieren:

1. Öffnen Sie das Anmeldefenster in VS Code:

a. Wählen Sie zum Starten des Authentifizierungsprozesses das AWS-Symbol im linken Navigationsbereich oder öffnen Sie die Befehlspalette (Cmd+Shift+P auf Mac oder Strg+Shift+P auf Windows/Linux) und wählen Sie AWS Neue Verbindung hinzufügen aus.

2. Wählen Sie im Anmeldebereich IAM-Anmeldeinformationen und anschließend Weiter.

> **Note**
>
> Sie müssen AWS IDE-Erweiterungen für VS Code den Zugriff auf Ihre Daten erlauben, um fortzufahren.

3. Geben Sie Ihren Profilnamen, Ihre Zugriffsschlüssel-ID und Ihren geheimen Zugriffsschlüssel ein und wählen Sie dann Weiter aus.
4. Überprüfen Sie die Verbindung, indem Sie im AWS Explorer in VS Code nach Ihren AWS-Services und -Ressourcen suchen.

Informationen zum Einrichten der Authentifizierung mit langfristigen Anmeldeinformationen finden Sie unter Verwenden langfristiger Anmeldeinformationen zur Authentifizierung von AWS-SDKs und -Tools.

Informationen zum Konfigurieren der Authentifizierung finden Sie unter AWS-IAM-Anmeldeinformationen im AWS Toolkit for Visual Studio Code-Benutzerhandbuch.

Wechsel von der Konsole zur lokalen Entwicklung

> **Note**
>
> Wenn Sie Änderungen in der Konsole vorgenommen haben, stellen Sie sicher, dass keine nicht bereitgestellten Änderungen vorhanden sind, bevor Sie zur lokalen Entwicklung übergehen.

So verschieben Sie eine Lambda-Funktion von der Lambda-Konsole nach VS Code:

1. Öffnen Sie die Lambda-Konsole.
2. Wählen Sie den Namen Ihrer Funktion aus.
3. Wählen Sie die Registerkarte Codequelle aus.
4. Wählen Sie In Visual Studio Code öffnen aus.

> **Note**
>
> Die Schaltfläche In Visual Studio Code öffnen ist nur in der AWS-Toolkit-Version 3.69.0 und neuer verfügbar. Wenn Sie eine ältere Version des AWS-Toolkits installiert haben, wird möglicherweise die Meldung „`Cannot open the handler`“ in VS Code angezeigt. Um dieses Problem zu beheben, aktualisieren Sie Ihr AWS-Toolkit auf die neueste Version.

5. Wenn Sie dazu aufgefordert werden, erlauben Sie dem Browser, VS Code zu öffnen.

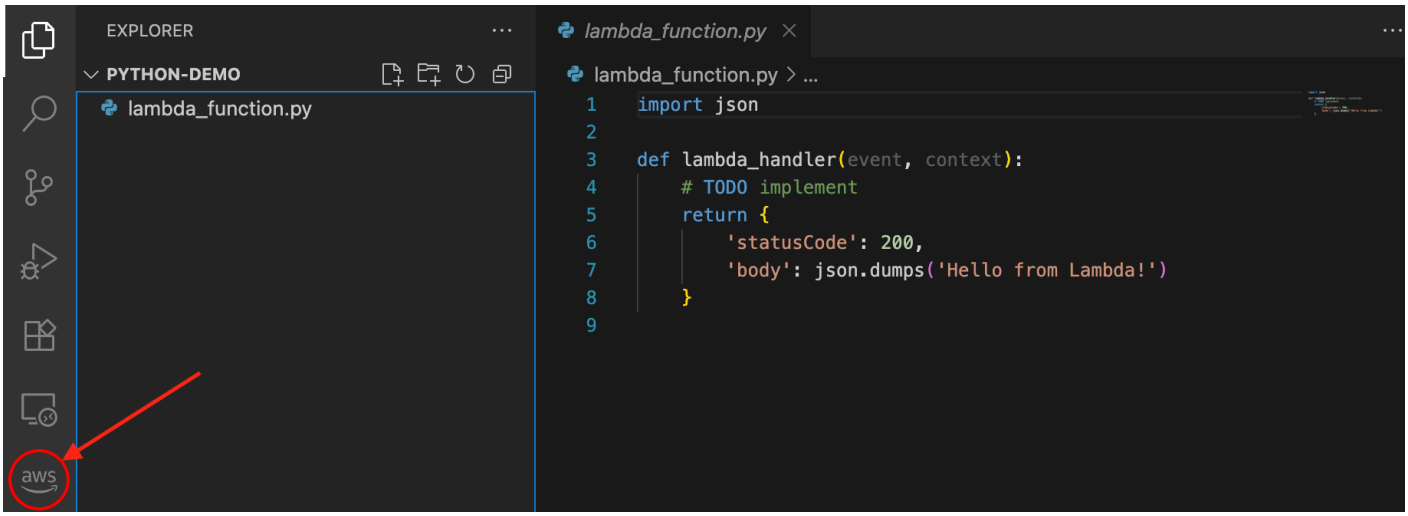
Wenn Sie Ihre Funktion in VS Code öffnen, erstellt Lambda ein lokales Projekt mit Ihrem Funktionscode an einem temporären Speicherort, der für schnelle Tests und Bereitstellungen vorgesehen ist. Dazu gehören der Funktionscode, Abhängigkeiten und eine grundlegende Projektstruktur, die Sie für die lokale Entwicklung verwenden können.

Einzelheiten zur Verwendung von AWS in VS Code finden Sie im AWS Toolkit for Visual Studio Code-Benutzerhandbuch.

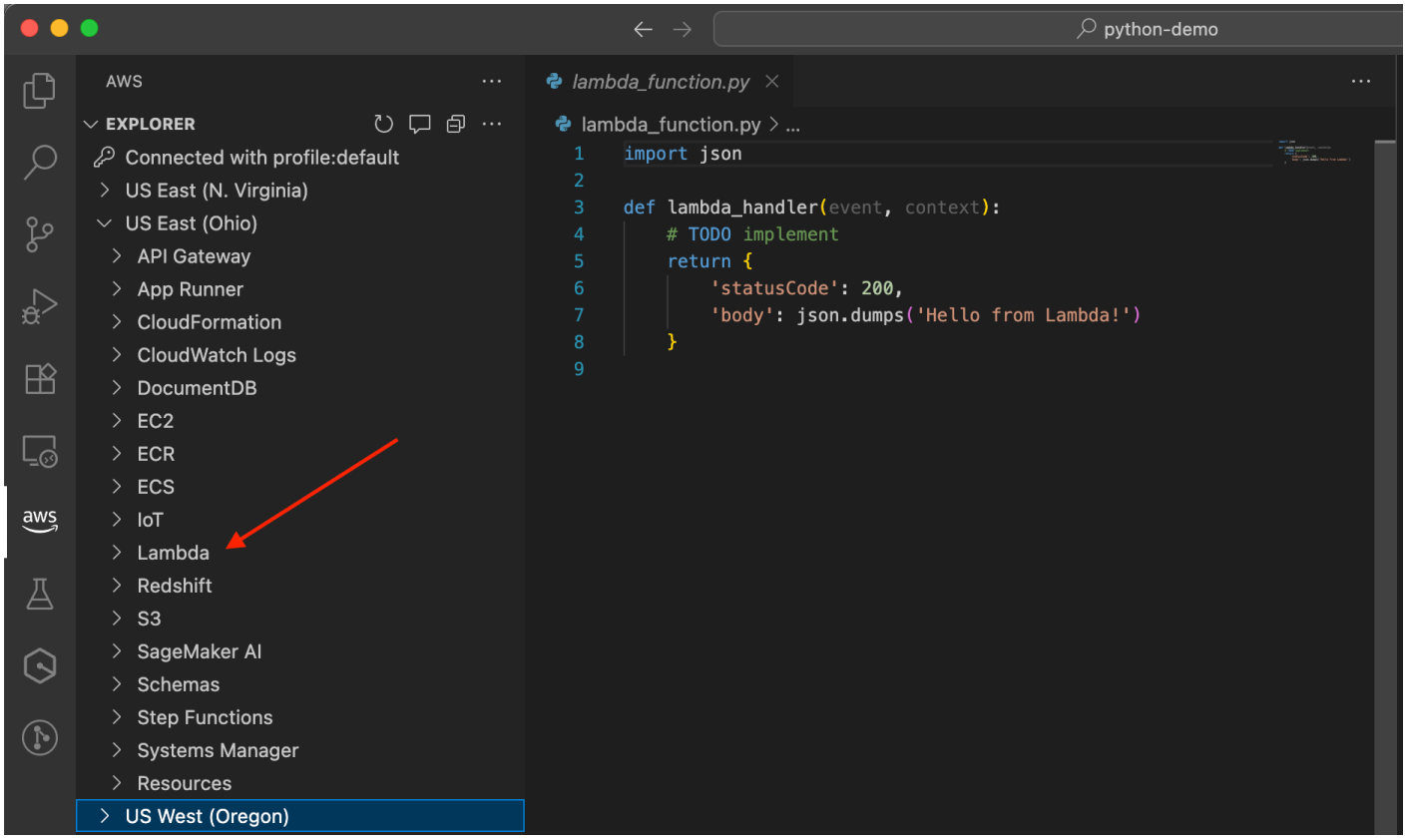
Lokales Arbeiten mit Funktionen

Gehen Sie nach dem Öffnen der Funktion in VS Code wie folgt vor, um auf Ihre Funktionen zuzugreifen und diese zu verwalten:

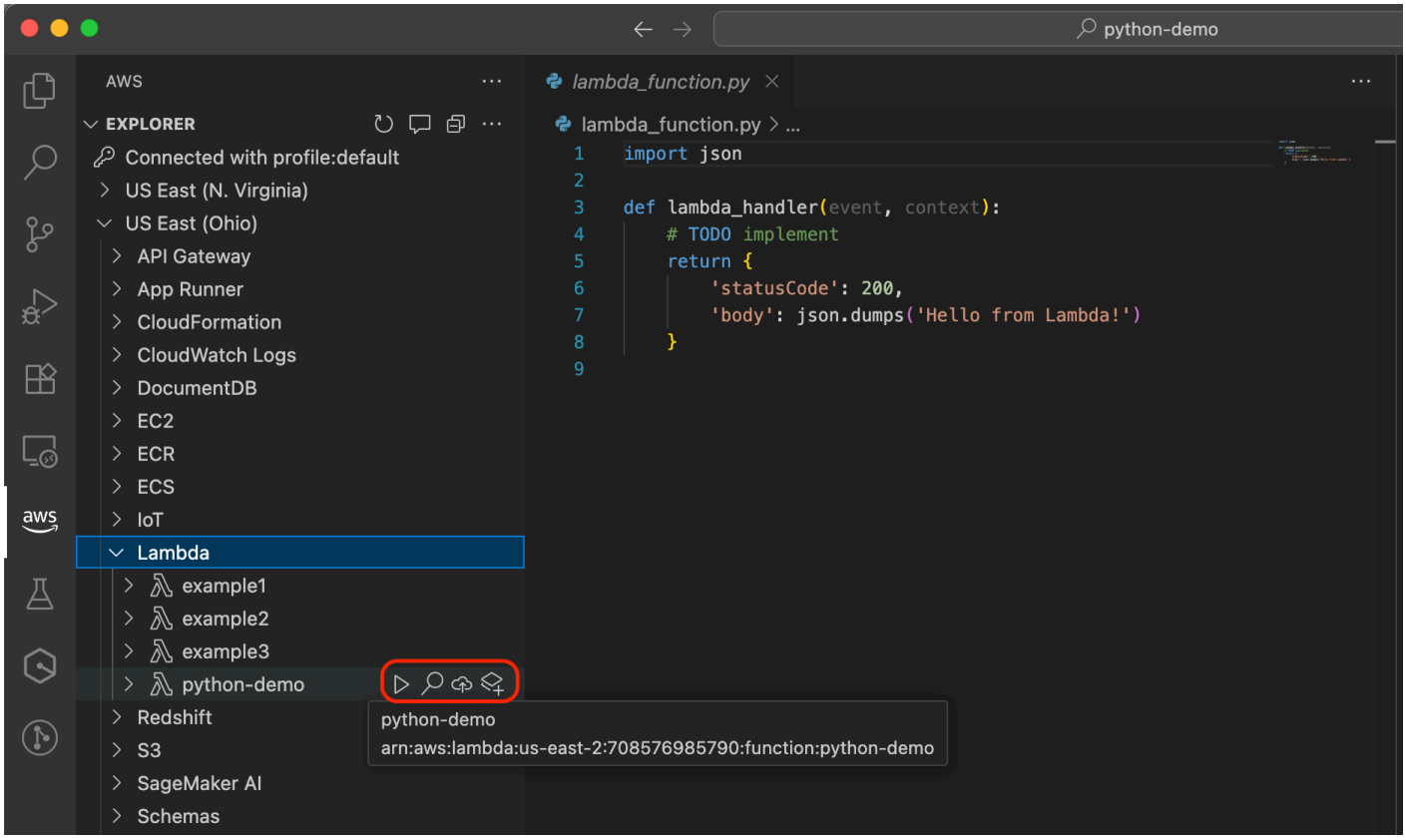
1. Wählen Sie das AWS-Symbol in der Seitenleiste, um den AWS Explorer zu öffnen:



2. Wählen Sie im AWS Explorer die Region mit Ihrer Lambda-Funktion aus:



3. Erweitern Sie unter der ausgewählten Region den Abschnitt „Lambda“, um Ihre Funktionen anzuzeigen und zu verwalten:



Wenn Ihre Funktion in VS Code geöffnet ist, können Sie Folgendes:

- Bearbeiten Sie den Funktionscode mit vollständiger Sprachunterstützung und Codevervollständigung.
- Verwenden Sie die LocalStack-Integration in VS Code, um Lambda-Funktionen zu testen, die während der Ausführung API-Aufrufe an andere AWS-Services richten, z. B. das Lesen aus DynamoDB-Tabellen oder das Schreiben in Amazon-S3-Buckets. LocalStack ist ein Cloud-Service-Emulator, der eine vollständige lokale Entwicklungsumgebung zum Testen von Service-Integrationen bereitstellt. Sie können auch AWS SAM CLI verwenden, um Ihre Funktion in einem lokalen Container zu testen. Wenn Ihre Funktion API-Aufrufe an andere AWS-Services sendet, erreichen diese Aufrufe echte AWS-Ressourcen, keine emulierten.
- Debuggen Sie Ihre Funktion mit Haltepunkten und Variablenprüfung. Weitere Informationen finden Sie im AWS Toolkit for Visual Studio Code-Benutzerhandbuch unter Ausführen und Debuggen von Lambda-Funktionen direkt aus dem Code.
- Stellen Sie Ihre aktualisierte Funktion über das Cloud-Symbol wieder in AWS bereit.
- Installieren und verwalten Sie Abhängigkeiten für Ihre Funktion.

Weitere Informationen finden Sie unter Arbeiten mit AWS-Lambda-Funktionen im AWS Toolkit for Visual Studio Code-Benutzerhandbuch.

Konvertieren Ihrer Funktion in eine AWS SAM-Vorlage und Verwenden von IaC-Tools

In VS Code können Sie Ihre Lambda-Funktion in eine AWS SAM-Vorlage konvertieren, indem Sie neben der Lambda-Funktion auf das Symbol In AWS SAM-Anwendung konvertieren klicken. Sie werden aufgefordert, einen AWS SAM-Projektspeicherort auszuwählen. Nach der Auswahl wird Ihre Lambda-Funktion in eine `template.yaml`-Datei konvertiert, die in Ihrem neuen AWS SAM-Projekt gespeichert wird.

Wenn Ihre Funktion in eine AWS SAM-Vorlage umgewandelt wurde, können Sie Folgendes:

- Versionierung Ihrer Infrastruktur steuern
- Bereitstellungen automatisieren
- Funktionen aus der Ferne debuggen
- Ihrer Anwendung zusätzliche AWS-Ressourcen hinzufügen
- Während Ihres gesamten Entwicklungszyklus für konsistente Umgebungen sorgen
- Infrastructure Composer, um Ihre AWS SAM-Vorlage visuell zu bearbeiten

Weitere Informationen zur Verwendung von IaC-Tools finden Sie in den folgenden Handbüchern:

- Das AWS Serverless Application Model-Entwicklerhandbuch
- Das AWS Cloud Development Kit (AWS CDK)-Entwicklerhandbuch
- Das Infrastructure-Composer-Entwicklerhandbuch
- Das AWS CloudFormation-Benutzerhandbuch

Diese Tools bieten zusätzliche Funktionen zum Definieren, Testen und Bereitstellen Ihrer Serverless-Anwendungen.

Nächste Schritte

Weitere Informationen zum Arbeiten mit Lambda-Funktionen in VS Code finden Sie in den folgenden Ressourcen:

- Arbeiten mit AWS-Lambda-Funktionen im AWS-Toolkit für VS-Code-Benutzerhandbuch
- Arbeiten mit Serverless-Funktionen im AWS-Toolkit für VS-Code-Benutzerhandbuch
- Infrastructure as Code im Lambda-Benutzerhandbuch

Verwenden von GitHub Actions zur Bereitstellung von Lambda-Funktionen

Sie können GitHub Actions verwenden, um Lambda-Funktionen automatisch bereitzustellen, wenn Sie Code- oder Konfigurationsänderungen in Ihr Repository übertragen. Die Aktion Lambda-Funktion bereitstellen bietet eine deklarative, einfache YAML-Schnittstelle, die die Komplexität manueller Bereitstellungsschritte beseitigt.

Beispiel-ArbeitsFlows

Um die automatisierte Bereitstellung von Lambda-Funktionen zu konfigurieren, erstellen Sie eine Workflow-Datei im `.github/workflows/`-Verzeichnis Ihres Repositorys:

Example GitHub-Actions-Workflow für die Lambda-Bereitstellung

```
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
```

Dieser Workflow wird ausgeführt, wenn Sie Änderungen an den `main`-Branch übertragen. Er überprüft Ihr Repository, konfiguriert die AWS-Anmeldeinformationen mithilfe von OpenID Connect (OIDC) und stellt Ihre Funktion mithilfe des Codes im `./dist`-Verzeichnis bereit.

Weitere Beispiele, darunter die Aktualisierung der Funktionskonfiguration, die Bereitstellung über S3-Buckets und die Testlauf-Validierung, finden Sie in der README-Datei zur Bereitstellung von Lambda-Funktionen.

Weitere Ressourcen

- Konfigurieren von AWS-Anmeldeinformationen für GitHub Actions
- Konfigurieren von OpenID Connect in AWS

Verwenden von Lambda mit Infrastructure as Code (IaC)

Lambda-Funktionen werden selten isoliert ausgeführt. Stattdessen sind sie oft Teil einer Serverless-Anwendung, zu der auch andere Ressourcen wie Datenbanken, Warteschlangen und Speicher gehören. Mit infrastructure as code (IaC) können Sie Ihre Bereitstellungsprozesse automatisieren, um ganze Serverless-Anwendungen mit vielen separaten AWS-Ressourcen schnell und wiederholbar bereitzustellen und zu aktualisieren. Dieser Ansatz beschleunigt Ihren Entwicklungszyklus, erleichtert die Konfigurationsverwaltung und stellt sicher, dass Ihre Ressourcen jedes Mal auf die gleiche Weise bereitgestellt werden.

IaC-Tools für Lambda

CloudFormation

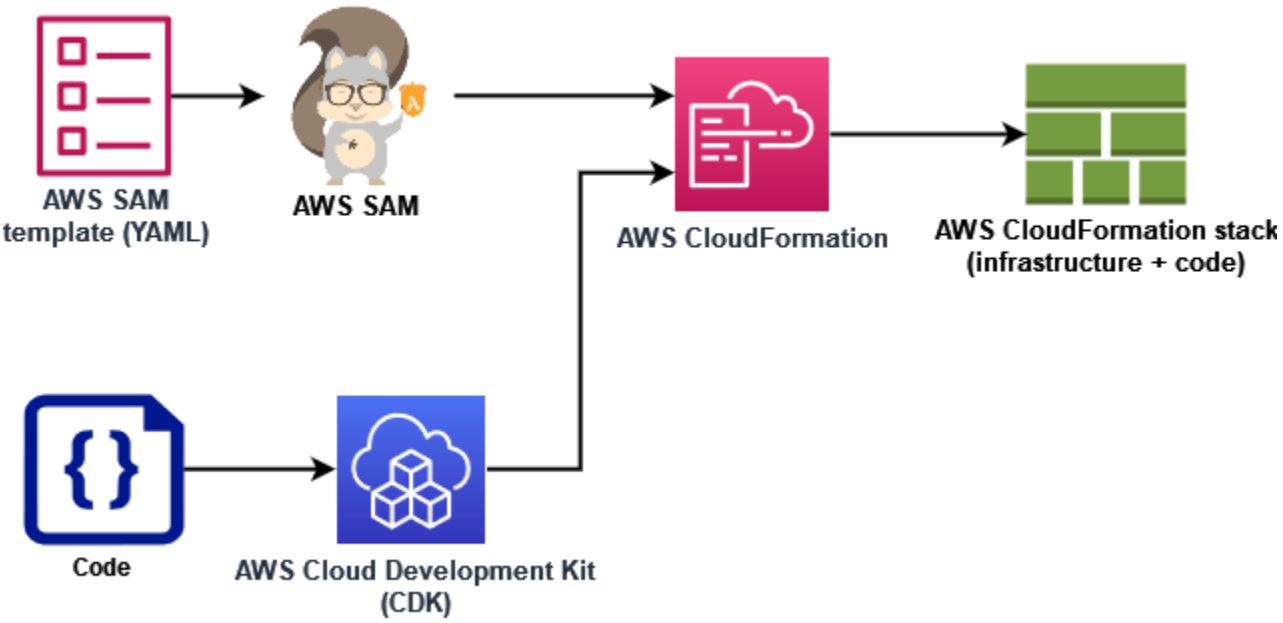
> CloudFormation ist der grundlegende IaC-Dienst von AWS. Sie können YAML- oder JSON-Vorlagen verwenden, um Ihre gesamte AWS-Infrastruktur, einschließlich Lambda-Funktionen, zu modellieren und bereitzustellen. CloudFormation bewältigt die Komplexität beim Erstellen, Aktualisieren und Löschen Ihrer AWS-Ressourcen.

AWS Serverless Application Model (AWS SAM)

AWS SAM ist ein Open-Source-Framework, das auf CloudFormation aufbaut. Es bietet eine vereinfachte Syntax für die Definition Serverless-Anwendungen. Verwenden Sie AWS SAM-Vorlagen, um Lambda-Funktionen, APIs, Datenbanken und Ereignisquellen schnell mit nur wenigen YAML-Zeilen bereitzustellen.

AWS Cloud Development Kit (AWS CDK)

Das CDK ist ein IaC-Ansatz, bei dem der Code an erster Stelle steht. Sie können Ihre Lambda-basierte Architektur mit TypeScript, JavaScript, Python, Java, C#/.Net oder Go definieren. Wählen Sie Ihre bevorzugte Sprache und verwenden Sie Programmierelemente wie Parameter, Konditionale, Schleifen, Komposition und Vererbung, um das gewünschte Ergebnis Ihrer Infrastruktur zu definieren. Das CDK generiert dann die zugrunde liegenden CloudFormation-Vorlagen für die Bereitstellung. Ein Beispiel für die Verwendung von Lambda mit CDK finden Sie unter Bereitstellung von Lambda-Funktionen mit AWS CDK.



AWS bietet außerdem einen Service namens AWS Infrastructure Composer zur Entwicklung von IaC-Vorlagen mithilfe einer einfachen grafischen Oberfläche. Mit Infrastructure Composer entwerfen Sie eine Anwendungsarchitektur, indem Sie AWS-Services auf einer visuellen Zeichenfläche ziehen, gruppieren und verbinden. Infrastructure Composer erstellt dann eine AWS SAM-Vorlage oder eine CloudFormation-Vorlage aus Ihrem Entwurf, die Sie zur Bereitstellung Ihrer Anwendung verwenden können.

Im folgenden Abschnitt the section called “Verwenden von AWS SAM und Infrastructure Composer” verwenden Sie Infrastructure Composer, um eine Vorlage für eine Serverless-Anwendung zu entwickeln, die auf einer vorhandenen Lambda-Funktion basiert.

Verwenden von Lambda-Funktionen in AWS SAM und Infrastructure Composer

In diesem Tutorial können Sie mit der Verwendung von IaC mit Lambda beginnen, indem Sie eine AWS SAM-Vorlage aus einer vorhandenen Lambda-Funktion erstellen und dann eine Serverless-Anwendung in Infrastructure Composer erstellen, indem Sie weitere AWS-Ressourcen hinzufügen.

Während Sie dieses Tutorial durchführen, lernen Sie einige grundlegende Konzepte kennen, z. B. wie AWS-Ressourcen in AWS SAM angegeben werden. Zudem erfahren Sie, wie Sie mit Infrastructure Composer eine Serverless-Anwendung erstellen, die Sie mit AWS SAM oder CloudFormation bereitstellen können.

Führen Sie für dieses Tutorial die folgenden Schritte aus:

- Erstellen Sie eine Beispiel-Lambda-Funktion.
- Verwenden Sie die Lambda-Konsole, um die AWS SAM-Vorlage für die Funktion anzuzeigen.
- Exportieren Sie die Konfiguration Ihrer Funktion nach AWS Infrastructure Composer und entwerfen Sie eine einfache Serverless-Anwendung, die auf der Konfiguration Ihrer Funktion basiert.
- Speichern Sie eine aktualisierte AWS SAM-Vorlage, die Sie als Grundlage für die Bereitstellung Ihrer Serverless-Anwendung verwenden können.

Voraussetzungen

In diesem Tutorial verwenden Sie das Feature für die lokale Synchronisierung von Infrastructure Composer, um Ihre Vorlagen- und Codedateien auf Ihrem lokalen Build-Computer zu speichern. Um diese Feature nutzen zu können, benötigen Sie einen Browser, der die File System Access-API unterstützt. Damit können Webanwendungen Dateien in Ihrem lokalen Dateisystem lesen, schreiben und speichern. Wir empfehlen, entweder Google Chrome oder Microsoft Edge zu verwenden. Weitere Informationen zur File System Access-API finden Sie unter Was ist die File System Access-API?.

Erstellen einer Lambda-Funktion

In diesem ersten Schritt erstellen Sie eine Lambda-Funktion, mit der Sie den Rest des Tutorials abschließen können. Der Einfachheit halber verwenden Sie die Lambda-Konsole, um mithilfe der Python 3.11-Laufzeit eine grundlegende „Hello World“-Funktion zu erstellen.

So erstellen Sie eine „Hello World“-Lambda-Funktion mit der Konsole

1. Öffnen Sie die Lambda-Konsole.
2. Wählen Sie Create function (Funktion erstellen).
3. Lassen Sie Ohne Vorgabe erstellen ausgewählt. Geben Sie im Feld Grundlegende Informationen für Funktionsname **`LambdaIaCDemo`** ein.
4. Wählen Sie für Laufzeit die Option Python 3.11 aus.
5. Wählen Sie Funktion erstellen.

Sehen Sie sich die AWS SAM-Vorlage für Ihre Funktion an.

Bevor Sie die Konfiguration Ihrer Funktion nach Infrastructure Composer exportieren, verwenden Sie die Lambda-Konsole, um die aktuelle Konfiguration Ihrer Funktion als AWS SAM-Vorlage anzuzeigen. Wenn Sie die Schritte in diesem Abschnitt befolgen, erfahren Sie mehr über den Aufbau einer AWS SAM-Vorlage und darüber, wie Sie Ressourcen wie Lambda-Funktionen definieren, um mit der Spezifizierung einer Serverless-Anwendung zu beginnen.

So sehen Sie sich die AWS SAM-Vorlage für Ihre Funktion an

1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.
2. Wählen Sie die Funktion aus, die Sie gerade erstellt haben (`LambdaIaCDemo`).
3. Wählen Sie im Bereich Funktionsübersicht die Option Vorlage.

   Anstelle des Diagramms, das die Konfiguration Ihrer Funktion darstellt, wird eine AWS SAM-Vorlage für Ihre Funktion angezeigt. Die Vorlage sollte jetzt wie folgt aussehen:

```
# 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:*
```

Nehmen wir uns einen Moment Zeit, um uns die YAML-Vorlage für Ihre Funktion anzusehen und einige wichtige Konzepte zu verstehen.

Die Vorlage beginnt mit der Deklaration `Transform: AWS::Serverless-2016-10-31`. Diese Deklaration ist erforderlich, weil im Hintergrund AWS SAM-Vorlagen über CloudFormation bereitgestellt werden. Mithilfe der `Transform`-Anweisung wird die Vorlage als AWS SAM-Vorlagendatei identifiziert.

Nach der `Transform`-Deklaration folgt der `Resources`-Abschnitt. Hier werden die AWS-Ressourcen definiert, die Sie mit Ihrer AWS SAM-Vorlage bereitstellen möchten. AWS SAM-Vorlagen können eine Kombination aus AWS SAM-Ressourcen und CloudFormation-Ressourcen enthalten. Das liegt daran, dass AWS SAM-Vorlagen bei der Bereitstellung zu CloudFormation-Vorlagen erweitert werden, sodass jeder AWS SAM-Vorlage jede gültige CloudFormation-Syntax hinzugefügt werden kann.

Im Moment ist im Abschnitt `Resources` der Vorlage nur eine Ressource definiert, Ihre Lambda-Funktion `LambdaIaCDemo`. Um einer AWS SAM-Vorlage eine Lambda-Funktion hinzuzufügen, verwenden Sie den `AWS::Serverless::Function`-Ressourcentyp. Die `Properties` der Ressource einer Lambda-Funktion definieren die Laufzeit, den Funktionshandler und andere Konfigurationsoptionen der Funktion. Der Pfad zum Quellcode Ihrer Funktion, den AWS SAM für die Bereitstellung der Funktion verwenden soll, ist hier ebenfalls definiert. Weitere Informationen zu Lambda-Funktionsressourcen in AWS SAM finden Sie unter AWS::Serverless::Function im AWS SAM-Entwicklerhandbuch.

Neben den Funktionseigenschaften und Konfigurationen ist in der Vorlage auch eine AWS Identity and Access Management-Richtlinie (IAM) für Ihre Funktion angegeben. Diese Richtlinie gibt Ihrer Funktion die Berechtigung zum Schreiben von Protokollen in Amazon CloudWatch Logs. Wenn Sie eine Funktion in der Lambda-Konsole erstellen, hängt Lambda diese Richtlinie automatisch an Ihre Funktion an. Weitere Informationen zur Angabe einer IAM-Richtlinie für eine Funktion in einer AWS SAM-Vorlage finden Sie in der *`policies`*-Eigenschaft auf der Seite AWS::Serverless::Function im AWS SAM-Entwicklerleitfaden.

Weitere Informationen zur Struktur von AWS SAM-Vorlagen finden Sie unter AWS SAM-Vorlagenaufbau.

Verwenden von AWS Infrastructure Composer zum Entwerfen einer Serverless-Anwendung

Um mit der Erstellung einer einfachen Serverless-Anwendung zu beginnen, die die AWS SAM-Vorlage Ihrer Funktion als Ausgangspunkt verwendet, exportieren Sie Ihre Funktionskonfiguration nach Infrastructure Composer und aktivieren den lokalen Synchronisierungsmodus von

Infrastructure Composer. Die Funktion für die lokale Synchronisierung speichert den Code Ihrer Funktion und Ihre AWS SAM-Vorlage automatisch auf Ihrem lokalen Build-Computer und sorgt dafür, dass Ihre gespeicherte Vorlage synchronisiert wird, wenn Sie weitere AWS-Ressourcen in Infrastructure Composer hinzufügen.

So exportieren Sie Ihre Funktion nach Infrastructure Composer

1. Wählen Sie im Bereich Funktionsübersicht die Option Nach Application Composer exportieren aus.

   Um die Konfiguration und den Code Ihrer Funktion nach Infrastructure Composer zu exportieren, erstellt Lambda einen Amazon-S3-Bucket in Ihrem Konto, um diese Daten vorübergehend zu speichern.
2. Wählen Sie im Dialogfeld Projekt bestätigen und erstellen aus, um den Standardnamen für diesen Bucket zu akzeptieren und die Konfiguration und den Code Ihrer Funktion nach Infrastructure Composer zu exportieren.
3. (Optional) Um einen anderen Namen für den von Lambda erstellten Amazon-S3-Bucket auszuwählen, geben Sie einen neuen Namen ein und wählen Sie Projekt bestätigen und erstellen aus. Die Amazon-S3-Bucket-Namen müssen global eindeutig sein und den Regeln für die Benennung von Buckets entsprechen.

   Wenn Sie Projekt bestätigen und erstellen auswählen, wird die Infrastructure-Composer-Konsole geöffnet. Auf der Zeichenfläche sehen Sie Ihre Lambda-Funktion.
4. Wählen Sie in der Menü-Dropdown-Liste die Option Lokale Synchronisierung aktivieren aus.
5. Wählen Sie in dem sich öffnenden Dialogfeld die Option Ordner auswählen aus und wählen Sie einen Ordner auf Ihrem lokalen Build-Computer aus.
6. Wählen Sie Aktivieren aus, um die lokale Synchronisierung zu aktivieren.

Zum Exportieren Ihrer Funktion nach Infrastructure Composer benötigen Sie die Berechtigung zur Verwendung von bestimmten API-Aktionen. Wenn Sie Ihre Funktion nicht exportieren können, beachten Sie the section called “Erforderliche Berechtigungen” und stellen Sie sicher, dass Sie über die erforderlichen Berechtigungen verfügen.

> **Note**
>
> Für den Bucket, den Lambda erstellt, wenn Sie eine Funktion nach Infrastructure Composer exportieren, gelten die Amazon-S3-Standardpreise. Die Objekte, die Lambda in den Bucket

einfügt, werden nach 10 Tagen automatisch gelöscht, aber Lambda löscht den Bucket selbst nicht.
Folgen Sie den Anweisungen unter Löschen eines Buckets, nachdem Sie Ihre Funktion nach Infrastructure Composer exportiert haben, um zu vermeiden, dass für Ihr AWS-Konto zusätzliche Kosten anfallen. Weitere Informationen zum Amazon-S3-Bucket-Namen, den Lamda erstellt, finden Sie unter the section called “Infrastructure Composer”.

So entwerfen Sie Ihre Serverless-Anwendung in Infrastructure Composer

Nach der Aktivierung der lokalen Synchronisierung werden die Änderungen, die Sie in Infrastructure Composer vornehmen, in der AWS SAM-Vorlage widergespiegelt, die auf Ihrer lokalen Build-Maschine gespeichert ist. Sie können jetzt zusätzliche AWS-Ressourcen per Drag-and-Drop auf die Infrastructure-Composer-Zeichenfläche ziehen, um Ihre Anwendung zu erstellen. In diesem Beispiel fügen Sie eine einfache Amazon-SQS-Warteschlange als Trigger für Ihre Lambda-Funktion und eine DynamoDB-Tabelle für die Funktion hinzu, in die Daten geschrieben werden sollen.

1. Fügen Sie Ihrer Lambda-Funktion einen Amazon-SQS-Trigger hinzu, indem Sie wie folgt vorgehen:

   a. Geben Sie im Suchfeld in der Ressourcenpalette **`SQS`** ein.
   b. Ziehen Sie die Ressource SQS-Warteschlange auf Ihre Zeichenfläche und positionieren Sie sie links neben Ihrer Lambda-Funktion.
   c. Wählen Sie Details und geben Sie **`LambdaIaCQueue`** als Logische ID ein.
   d. Wählen Sie Speichern.
   e. Verbinden Sie Ihre Amazon-SQS- und Lambda-Ressourcen, indem Sie auf der SQS-Warteschlangenkarte auf den Port Abonnement klicken und ihn auf den linken Port auf der Lambda-Funktionskarte ziehen. Das Erscheinen einer Linie zwischen den beiden Ressourcen weist auf eine erfolgreiche Verbindung hin. Infrastructure Composer zeigt außerdem unten auf der Zeichenfläche eine Meldung an, die darauf hinweist, dass die beiden Ressourcen erfolgreich verbunden wurden.
2. Fügen Sie eine Amazon-DynamoDB-Tabelle für Ihre Lambda-Funktion hinzu, in die Daten geschrieben werden sollen, indem Sie so vorgehen:

   a. Geben Sie im Suchfeld in der Ressourcenpalette **`DynamoDB`** ein.
   b. Ziehen Sie die Ressource DynamoDB-Tabelle auf Ihre Zeichenfläche und positionieren Sie sie rechts neben Ihrer Lambda-Funktion.

c. Wählen Sie Details und geben Sie **`LambdaIaCTable`** als Logische ID ein.

d. Wählen Sie Speichern.

e. Verbinden Sie die DynamoDB-Tabelle mit Ihrer Lambda-Funktion, indem Sie auf den rechten Port der Lambda-Funktionskarte klicken und ihn auf den linken Port der DynamoDB-Karte ziehen.

Nachdem Sie diese zusätzlichen Ressourcen hinzugefügt haben, werfen wir einen Blick auf die aktualisierte AWS SAM-Vorlage, die Infrastructure Composer erstellt hat.

So zeigen Sie Ihre aktualisierte AWS SAM-Vorlage an

- Wählen Sie auf der Infrastructure-Composer-Zeichenfläche Vorlage, um von der Zeichenflächenansicht zur Vorlagenansicht zu wechseln.

Ihre AWS SAM-Vorlage sollte jetzt die folgenden zusätzlichen Ressourcen und Eigenschaften enthalten:

- Eine Amazon-SQS-Warteschlange mit der ID `LambdaIaCQueue`

```
LambdaIaCQueue:
    Type: AWS::SQS::Queue
    Properties:
      MessageRetentionPeriod: 345600
```

  Wenn Sie eine Amazon-SQS-Warteschlange mit Infrastructure Composer hinzufügen, setzt Infrastructure Composer die Eigenschaft `MessageRetentionPeriod`. Sie können die `FifoQueue`-Eigenschaft auch festlegen, indem Sie auf der SQS-Warteschlangenkarte Details auswählen und die Fifo-Warteschlange aktivieren oder deaktivieren.

  Um weitere Eigenschaften für Ihre Warteschlange festzulegen, können Sie die Vorlage manuell bearbeiten, um sie hinzuzufügen. Weitere Informationen zur `AWS::SQS::Queue`-Ressource und ihre verfügbaren Eigenschaften finden Sie unter AWS::SQS::Queue im CloudFormation-Benutzerhandbuch.

- Eine `Events`-Eigenschaft in Ihrer Lambda-Funktionsdefinition, die die Amazon-SQS-Warteschlange als Trigger für die Funktion angibt

```
Events:
```

```
  LambdaIaCQueue:
    Type: SQS
    Properties:
      Queue: !GetAtt LambdaIaCQueue.Arn
      BatchSize: 1
```

Die `Events`-Eigenschaft besteht aus einem Ereignistyp und einer Reihe von Eigenschaften, die vom Typ abhängen. Weitere Informationen zu den verschiedenen AWS-Services, die Sie zum Auslösen einer Lambda-Funktion konfigurieren können, und zu den Eigenschaften, die Sie festlegen können, finden Sie unter EventSource im AWS SAM-Entwicklerleitfaden.

- Eine DynamoDB-Tabelle mit dem Bezeichner `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
```

Wenn Sie mit Infrastructure Composer eine DynamoDB-Tabelle hinzufügen, können Sie die Schlüssel Ihrer Tabelle festlegen, indem Sie auf der DynamoDB-Tabellenkarte Details auswählen und die Schlüsselwerte bearbeiten. Infrastructure Composer legt auch Standardwerte für eine Reihe anderer Eigenschaften fest, darunter `BillingMode` und `StreamViewType`.

Weitere Informationen zu diesen und anderen Eigenschaften, die Sie Ihrer AWS SAM-Vorlage hinzufügen können, finden Sie unter AWS::DynamoDB::Table im CloudFormation-Benutzerhandbuch.

- Eine neue IAM-Richtlinie, die Ihrer Funktion die Berechtigung gewährt, CRUD-Operationen für die von Ihnen hinzugefügte DynamoDB-Tabelle auszuführen.

```
Policies:
...
  - DynamoDBCrudPolicy:
```

```
      TableName: !Ref LambdaIaCTable
```

Die endgültige vollständige AWS SAM-Vorlage sollte jetzt so aussehen:

```
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
```

Stellen Sie Ihre Serverless-Anwendung mithilfe von AWS SAM (optional) bereit

Wenn Sie AWS SAM verwenden möchten, um eine Serverless-Anwendung mithilfe der Vorlage bereitzustellen, die Sie gerade in Infrastructure Composer erstellt haben, müssen Sie zuerst die AWS SAM-CLI installieren. Befolgen Sie hierzu die Anweisungen unter Installieren der AWS SAM-CLI.

Bevor Sie Ihre Anwendung bereitstellen, müssen Sie auch den Funktionscode aktualisieren, den Infrastructure Composer zusammen mit Ihrer Vorlage gespeichert hat. Derzeit enthält die von Infrastructure Composer gespeicherte `lambda_function.py`-Datei nur den grundlegenden „Hello World“-Code, den Lambda bei der Erstellung der Funktion bereitgestellt hat.

Um Ihren Funktionscode zu aktualisieren, kopieren Sie den folgenden Code und fügen Sie ihn in die `lambda_function.py`-Datei ein, die Infrastructure Composer auf Ihrer lokalen Build-Maschine

gespeichert hat. Sie haben das Verzeichnis, in dem Infrastructure Composer diese Datei speichern soll, beim Aktivieren des Modus „Lokale Synchronisation“ angegeben.

Dieser Code akzeptiert ein Schlüssel-Wert-Paar in einer Nachricht von der Amazon-SQS-Warteschlange, die Sie in Infrastructure Composer erstellt haben. Wenn sowohl der Schlüssel als auch der Wert Zeichenfolgen sind, verwendet der Code diese Werte, um ein Element in die in Ihrer Vorlage definierte DynamoDB-Tabelle zu schreiben.

Aktualisierter Python-Funktionscode

```
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
```

So stellen Sie Ihre Serverless-Anwendung bereit

Gehen Sie wie folgt vor, um Ihre Anwendung mithilfe der AWS SAM-CLI bereitzustellen. Damit Ihre Funktion korrekt erstellt und bereitgestellt werden kann, muss die Python-Version 3.11 auf Ihrem Build-Computer und in Ihrem PATH installiert sein.

1. Führen Sie den folgenden Befehl in dem Verzeichnis aus, in dem Infrastructure Composer Ihre Dateien `template.yaml` und `lambda_function.py` gespeichert hat.

   ```
   sam build
   ```

   Dieser Befehl sammelt die Build-Artefakte für Ihre Anwendung und platziert sie im richtigen Format und am richtigen Ort, um sie bereitzustellen.
2. Führen Sie den folgenden Befehl aus, um Ihre Anwendung bereitzustellen und die in Ihrer AWS SAM-Vorlage angegebenen Lambda-, Amazon-SQS- und DynamoDB-Ressourcen zu erstellen.

   ```
   sam deploy --guided
   ```

   Wenn Sie die `--guided`-Markierung verwenden, werden Ihnen in AWS SAM Eingabeaufforderungen angezeigt, die Sie durch den Bereitstellungsprozess führen. Akzeptieren Sie für diese Bereitstellung die Standardoptionen, indem Sie die Eingabetaste drücken.

AWS SAM erstellt während des Bereitstellungsprozesses die folgenden Ressourcen in Ihrem AWS-Konto:

- Ein CloudFormation-Stack mit dem Namen `sam-app`
- Eine Lambda-Funktion mit dem Namensformat `sam-app-LambdaIaCDemo-`*`99VXPpYQVv1M`*
- Eine Amazon-SQS-Warteschlange mit dem Namensformat `sam-app-LambdaIaCQueue-`*`xL87VeKsGiIo`*

- Eine DynamoDB-Tabelle mit dem Namensformat `sam-app-LambdaIaCTable-`*`CN0S66C0VLNV`*

AWS SAM erstellt auch die erforderlichen IAM-Rollen und -Richtlinien, sodass Ihre Lambda-Funktion Nachrichten aus der Amazon-SQS-Warteschlange lesen und CRUD-Operationen in der DynamoDB-Tabelle ausführen kann.

Testen der bereitgestellten Anwendung (optional)

Um zu überprüfen, ob Ihre Serverless-Anwendung korrekt bereitgestellt wurde, senden Sie eine Nachricht an Ihre Amazon-SQS-Warteschlange, die ein Schlüssel-Wert-Paar enthält, und überprüfen Sie, ob Lambda mit diesen Werten ein Element in Ihre DynamoDB-Tabelle schreibt.

So testen Sie Ihre Serverless-Anwendung

1. Öffnen Sie die Seite Warteschlangen der Amazon-SQS-Konsole und wählen Sie die Warteschlange aus, die AWS SAM aus Ihrer Vorlage erstellt hat. Der Name weist das Format `sam-app-LambdaIaCQueue-`*`xL87VeKsGiIo`* auf.
2. Wählen Sie Nachrichten senden und empfangen aus und fügen Sie den folgenden JSON in den Nachrichtentext im Abschnitt Nachricht senden ein.

   ```
   {
       "myKey": "myValue"
   }
   ```

3. Klicken Sie auf Send Message (Nachricht senden).

   Wenn Sie Ihre Nachricht an die Warteschlange senden, ruft Lambda Ihre Funktion über Ihre Zuordnung von Ereignisquellen auf, die in Ihrer AWS SAM-Vorlage definiert ist. Um zu bestätigen, dass Lambda Ihre Funktion wie erwartet aufgerufen hat, vergewissern Sie sich, dass Ihrer DynamoDB-Tabelle ein Element hinzugefügt wurde.
4. Öffnen Sie die Seite Tabellen der DynamoDB-Konsole und wählen Sie Ihre Tabelle aus. Der Name weist das Format `sam-app-LambdaIaCTable-`*`CN0S66C0VLNV`* auf.
5. Wählen Sie Tabellenelemente erkunden aus. Im Bereich Zurückgegebene Elemente sollten Sie ein Element mit der ID `myKey` und dem Wert `myValue` sehen.

Bereitstellung von Lambda-Funktionen mit AWS CDK

Das AWS Cloud Development Kit (AWS CDK) ist ein Infrastructure-as-Code-Framework (IaC), mit dem Sie die AWS Cloud-Infrastruktur mithilfe einer Programmiersprache Ihrer Wahl definieren können. Um Ihre eigene Cloud-Infrastruktur zu definieren, schreiben Sie zunächst eine App (in einer der unterstützten Sprachen des CDK), die einen oder mehrere Stacks enthält. Anschließend synthetisieren Sie es zu einer CloudFormation Vorlage und stellen Ihre Ressourcen auf Ihrem bereit. AWS-Konto Folgen Sie den Schritten in diesem Thema, um eine Lambda-Funktion bereitzustellen, die ein Ereignis von einem Amazon-API-Gateway-Endpunkt zurückgibt.

Die im CDK enthaltene AWS Construct Library bietet Module, mit denen Sie die bereitgestellten Ressourcen modellieren können. AWS-Services Für die gängigen Dienste stellt die Bibliothek kuratierte Konstrukte bereit, die intelligente Standards und bewährte Methoden bereitstellen. Sie können das Modul aws_lambda verwenden, um Ihre Funktion und die unterstützenden Ressourcen mit nur wenigen Codezeilen zu definieren.

Voraussetzungen

Bevor Sie mit diesem Tutorial beginnen, installieren Sie das, AWS CDK indem Sie den folgenden Befehl ausführen.

```
npm install -g aws-cdk
```

Schritt 1: Richten Sie Ihr AWS CDK Projekt ein

Erstellen Sie ein Verzeichnis für Ihre neue AWS CDK App und initialisieren Sie das Projekt.

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
```

Nach dem Start des Projekts aktivieren Sie die virtuelle Umgebung des Projekts und installieren die Basisabhängigkeiten für AWS CDK.

```
source .venv/bin/activate
python -m pip install -r requirements.txt
```

Java

```
mkdir hello-lambda
cd hello-lambda
cdk init --language java
```

Importieren Sie dieses Maven-Projekt in Ihre integrierte Java-Entwicklungsumgebung (IDE). Verwenden Sie in Eclipse beispielsweise Datei, Import, Maven, Vorhandene Maven-Projekte.

C#

```
mkdir hello-lambda
cd hello-lambda
cdk init --language csharp
```

> **Note**
>
> Die AWS CDK Anwendungsvorlage verwendet den Namen des Projektverzeichnisses, um Namen für Quelldateien und Klassen zu generieren. In diesem Beispiel trägt das Verzeichnis den Namen `hello-lambda`. Wenn Sie einen anderen Projektverzeichnisnamen verwenden, stimmt Ihre App nicht mit diesen Anweisungen überein.

AWS CDK v2 enthält stabile Konstrukte für alle AWS-Services in einem einzigen Paket, das aufgerufen wird`aws-cdk-lib`. Dieses Paket wird als Abhängigkeit installiert, wenn Sie das Projekt initialisieren. Wenn Sie mit bestimmten Programmiersprachen arbeiten, wird das Paket installiert, wenn Sie das Projekt zum ersten Mal erstellen.

Schritt 2: Definieren Sie den Stapel AWS CDK

Ein CDK-Stack ist eine Sammlung von einem oder mehreren Konstrukten, die Ressourcen definieren AWS . Jeder CDK-Stack steht für einen CloudFormation Stack in Ihrer CDK-App.

Um Ihren CDK-Stack zu definieren, folgen Sie den Anweisungen für Ihre bevorzugte Programmiersprache. Dieser Stack definiert das Folgende:

- Der logische Name der Funktion: `MyFunction`
- Der Speicherort des Funktionscodes, der in der `code`-Eigenschaft angegeben ist. Weitere Informationen finden Sie unter Handler-Code in der AWS Cloud Development Kit (AWS CDK) -API-Referenz.
- Der logische Name der REST-API: `HelloApi`
- Der logische Name des API-Gateway-Endpunkts: `ApiGwEndpoint`

Beachten Sie, dass alle CDK-Stacks in diesem Tutorial die Laufzeit-Node.js für die Lambda-Funktion verwenden. Sie können verschiedene Programmiersprachen für den CDK-Stack und die Lambda-Funktion verwenden, um die Stärken der einzelnen Sprachen zu nutzen. Sie können ihn beispielsweise TypeScript für den CDK-Stack verwenden, um die Vorteile der statischen Typisierung für Ihren Infrastrukturcode zu nutzen. Sie können die Lambda-Funktion verwenden JavaScript , um die Flexibilität und schnelle Entwicklung einer dynamisch typisierten Sprache zu nutzen.

JavaScript

Öffnen Sie die Datei `lib/hello-lambda-stack.js` und ersetzen Sie den Inhalt durch Folgendes.

```
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

Öffnen Sie die Datei `lib/hello-lambda-stack.ts` und ersetzen Sie den Inhalt durch Folgendes.

```
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

Öffnen Sie die Datei `/hello-lambda/hello_lambda/hello_lambda_stack.py` und ersetzen Sie den Inhalt durch Folgendes.

```
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

Öffnen Sie die Datei `/hello-lambda/src/main/java/com/myorg/HelloLambdaStack.java` und ersetzen Sie den Inhalt durch Folgendes.

```
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#

Öffnen Sie die Datei `src/HelloLambda/HelloLambdaStack.cs` und ersetzen Sie den Inhalt durch Folgendes.

```
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
            });
        }
    }
}
```

Schritt 3: Den Lambda-Funktionscode erstellen

1. Erstellen Sie im Stammverzeichnis Ihres Projekts (`hello-lambda`) das `/lib/lambda-handler`-Verzeichnis für den Lambda-Funktionscode. Dieses Verzeichnis ist in der `code` Eigenschaft Ihres AWS CDK Stacks angegeben.
2. Erstellen Sie in dem Verzeichnis `/lib/lambda-handler` eine neue Datei namens `index.js`. Fügen Sie folgenden Code in die Datei ein. Die Funktion extrahiert bestimmte Eigenschaften aus der API-Anfrage und gibt sie als JSON-Antwort zurück.

```
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,
```

```
  };
};
```

Schritt 4: Stellen Sie den AWS CDK Stack bereit

1. Führen Sie im Stammverzeichnis Ihres Projekts den Befehl cdk synth aus:

```
cdk synth
```

Dieser Befehl synthetisiert eine AWS CloudFormation Vorlage aus Ihrem CDK-Stack. Die Vorlage ist eine ca. 400 Zeilen lange YAML-Datei, ähnlich der folgenden.

> **Note**
>
> Wenn Sie die folgende Fehlermeldung erhalten, vergewissern Sie sich, dass Sie sich im Stammverzeichnis Ihres Projektverzeichnisses befinden.
>
> ```
> --app is required either in command-line, in cdk.json or in ~/.cdk.json
> ```

Example CloudFormation Vorlage

```
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. Führen Sie den Befehl cdk deploy aus:

```
cdk deploy
```

Warten Sie, bis Ihre Ressourcen erstellt wurden. Die endgültige Ausgabe enthält die URL für Ihren API-Gateway-Endpunkt. Beispiel:

```
Outputs:
HelloLambdaStack.ApiGwEndpoint77F417B1 = https://abcd1234.execute-api.us-
east-1.amazonaws.com/prod/
```

Schritt 5: Testen der Funktion

Um die Lambda-Funktion aufzurufen, kopieren Sie den API Gateway-Endpunkt und fügen Sie ihn in einen Webbrowser ein oder führen Sie einen `curl`-Befehl aus:

```
curl -s https://abcd1234.execute-api.us-east-1.amazonaws.com/prod/
```

Die Antwort ist eine JSON-Darstellung ausgewählter Eigenschaften aus dem ursprünglichen Ereignisobjekt, das Informationen über die Anfrage an den API-Gateway-Endpunkt enthält. Beispiel:

```
{
```

```
  "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",
     ...
```

Schritt 6: Bereinigen Ihrer Ressourcen

Der API-Gateway-Endpunkt ist öffentlich zugänglich. Um unerwartete Gebühren zu vermeiden, führen Sie den Befehl cdk destroy aus, um den Stack und alle zugehörigen Ressourcen zu löschen.

```
cdk destroy
```

Nächste Schritte

Informationen zum Schreiben von AWS CDK Apps in der Sprache Ihrer Wahl finden Sie im Folgenden:

TypeScript

- Arbeiten mit AWS CDK dem TypeScript

JavaScript

- Mit der Tinte AWS CDK arbeiten JavaScript

Python

- Arbeiten mit dem AWS CDK in Python

Java

- Arbeiten mit dem AWS CDK in Java

C#

Arbeiten mit dem AWS CDK in C#

Go

Arbeiten mit dem AWS CDK in Go

Powertools for AWS Lambda

Powertools for AWS Lambda (auch als Powertools for AWS bezeichnet) bietet Dienstprogrammfunktionen, Dekoratoren und Middleware, die allgemeine Lambda-Aufgaben wie strukturierte Protokollierung, Nachverfolgung, Metrikerfassung und Eingabevalidierung übernehmen. Mit Powertools for AWS Lambda können Sie Serverless-Best-Practices implementieren und die Entwicklung mehrerer Lambda-Funktionen beschleunigen. Dadurch werden allgemeine Entwicklungsaufgaben in Ihren Lambda-Funktionen vereinfacht.

Hauptvorteile von Powertools for AWS

Die Lambda-Entwicklung ist zwar ohne Powertools for AWS möglich, ihre Verwendung bietet jedoch mehrere Vorteile:

- Integrierte Beobachtbarkeit: Strukturierte Protokollierung, Nachverfolgung und benutzerdefinierte Metriken
- Verwaltung von Secrets: Abrufen von Parametern, Umgang mit Secrets und Idempotenz
- Progressive Verbesserung: Auswahl der Dienstprogramme, die Ihren Anforderungen am besten entsprechen
- Beschleunigte Entwicklung: Analyse, Validierung und Batchverarbeitung von Ereignissen
- Bewährte Methoden: Implementierung von gut strukturierten AWS-Serverless-Mustern

Integration von Powertools mit AWS

Powertools for AWS unterstützt Sie beim Erstellen produktionsreifer Serverless-Anwendungen mit weniger benutzerdefiniertem Code. Powertools for AWS ist in Python, TypeScript/Node.js, .NET und Java verfügbar und kann über Lambda-Ebenen oder mithilfe des Sprachpaketmanagers eingebunden werden. Jede Sprachimplementierung bietet wichtige Features wie strukturierte Protokollierung, Nachverfolgung, Erfassung von Metriken und Ereignisbehandlung, wobei die für jede Programmiersprache typischen Idiome beibehalten werden. Diese Implementierungen werden

durch spezielle Komponenten für die AWS-Serviceintegration ergänzt, die den Abruf von Parametern, die Batchverarbeitung und die API-Handhabung unterstützen, sowie durch bewährte Verfahren wie Korrelations-ID-Weitergabe, Fehlerbehandlung und Idempotenzmuster. Zusammen ermöglichen diese Features Entwicklern die Erstellung robuster, wartungsfreundlicher Serverless-Anwendungen bei gleichzeitig reduziertem Aufwand für benutzerdefinierten Code.

- Powertools for AWS Lambda (Python)
- Powertools for AWS Lambda (TypeScript)
- Powertools for AWS Lambda (Java)
- Powertools for AWS Lambda (.NET)

Nächste Schritte

Weitere Informationen zum Arbeiten mit Powertools for AWS finden Sie in den folgenden Ressourcen:

- Workshop „Powertools for AWS Lambda“
- Serverless patterns that use Powertools for AWS
- AWS well-architected serverless lens
- Building Serverless APIs with Powertools for AWS Lambda

Verwaltung von Lambda-Workflows und -Ereignissen

Wenn Sie serverlose Anwendungen mit Lambda erstellen, benötigen Sie häufig Möglichkeiten, die Funktionsausführung zu orchestrieren und Ereignisse zu behandeln. AWS bietet mehrere Ansätze zur Koordination von Lambda-Funktionen:

- Langlebige Lambda-Funktionen für die Code-First-Workflow-Orchestrierung innerhalb von Lambda
- AWS Step Functions für die visuelle Workflow-Orchestrierung über mehrere Dienste hinweg
- Amazon EventBridge Scheduler und Amazon EventBridge für ereignisgesteuerte Architekturen und Planung

Sie können diese Ansätze auch zusammen integrieren. Beispielsweise können Sie den EventBridge Scheduler verwenden, um dauerhafte Funktionen oder Step Functions Functions-Workflows auszulösen, wenn bestimmte Ereignisse eintreten, oder Workflows so konfigurieren, dass

Ereignisse an definierten Ausführungspunkten im EventBridge Scheduler veröffentlicht werden. Die folgenden Themen in diesem Abschnitt enthalten weitere Informationen dazu, wie Sie diese Orchestrierungsoptionen verwenden können.

Orchestrierung, bei der der Code an erster Stelle steht, mit dauerhaften Funktionen

Dauerhafte Lambda-Funktionen bieten einen Code-First-Ansatz für die Workflow-Orchestrierung, sodass Sie statusbehaftete Workflows mit langer Laufzeit direkt in Ihren Lambda-Funktionen erstellen können. Im Gegensatz zu externen Orchestrierungsdiensten behalten langlebige Funktionen Ihre Workflow-Logik im Code, was es einfacher macht, sie zusammen mit Ihrer Geschäftslogik zu versionieren, zu testen und zu warten.

Dauerhafte Funktionen sind ideal, wenn Sie Folgendes benötigen:

- Workflow-Definition, bei der der Code an erster Stelle steht: Definieren Sie Workflows mit vertrauten Programmiersprachen und nicht mit JSON oder visuellen Designern
- Prozesse mit langer Laufzeit: Führen Sie Workflows aus, die bis zu einem Jahr lang ausgeführt werden können und damit weit über die 15-Minuten-Grenze der Standard-Lambda-Funktionen hinausgehen
- Vereinfachte Entwicklung: Behalten Sie die Workflow-Logik und die Geschäftslogik in derselben Codebasis, um die Wartung und das Testen zu vereinfachen
- Kostengünstiges Warten: Unterbrechen Sie die Ausführung während Wartezuständen, ohne Rechenressourcen zu verbrauchen
- Integrierte Statusverwaltung: Automatisches Checkpointing und State-Persistenz ohne externe Speicherkonfiguration

Wählen Sie zwischen dauerhaften Funktionen und Step Functions

Sowohl langlebige Funktionen als auch Step Functions bieten Funktionen zur Workflow-Orchestrierung, dienen jedoch unterschiedlichen Anwendungsfällen:

Überlegungen	Langlebige Funktionen	Step Functions
Definition des Arbeitsablaufs	Code zuerst (JavaScript, Python, Java)	JSON-basierter Sprach- oder Grafikdesigner von Amazon States

Überlegungen	Langlebige Funktionen	Step Functions
Entwicklungsansatz	Eine einzige Codebasis mit Geschäftslogik	Separate Workflow-Definition und Funktionscode
Service-Integration	Durch Lambda-Funktionscode und AWS SDKs	Native Integrationen mit vielen Diensten AWS
Dauer der Ausführung	Bis zu 1 Jahr	Bis zu 1 Jahr (Standard), 5 Minuten (Express)
Parallele Verarbeitung	Promise.all () und codebasierte Muster	Paralleler Zustand und verteilte Karte
Fehlerbehandlung	Try-Catch-Blöcke und benutzerdefinierte Wiederholungslogik	Integrierte Wiederholungs- und Fangstatus
Visuelle Überwachung	CloudWatch Logs und benutzerdefinierte Dashboards	Visuelles Ausführungsdiagramm und detaillierter Verlauf
Am besten geeignet für	Entwicklerorientierte Workflows, komplexe Geschäftslogik, schnelles Prototyping	Orchestrierung mehrerer Dienste, visuelle Workflows, Unternehmensführung

Verwenden Sie langlebige Funktionen, wenn:

- Ihr Team bevorzugt Entwicklungsansätze, bei denen der Code an erster Stelle steht
- Die Workflow-Logik ist eng mit der Geschäftslogik verknüpft
- Sie benötigen schnelles Prototyping und Iteration
- Ihre Workflows beinhalten hauptsächlich Lambda-Funktionen und einfache Serviceanrufe.

Verwenden Sie Step Functions, wenn:

- Sie benötigen ein visuelles Workflow-Design und eine visuelle Überwachung
- Ihre Workflows orchestrieren mehrere AWS Dienste umfassend

- Sie benötigen Funktionen für Unternehmensführung und Compliance
- Nicht technisch versierte Akteure müssen die Workflow-Logik verstehen

Weitere Informationen zu dauerhaften Funktionen finden Sie unter Dauerhafte Funktionen für Lambda.

Orchestrierung von Workflows mit Step Functions

AWS Step Functions ist ein Workflow-Orchestrierungsservice, mit dem Sie mehrere Lambda-Funktionen und andere AWS Dienste in strukturierten Workflows koordinieren können. Diese Workflows können den Status beibehalten, Fehler mit ausgeklügelten Wiederholungsmechanismen behandeln und Daten in großem Umfang verarbeiten.

Step Functions bietet zwei Arten von Workflows für unterschiedliche Orchestrierungsanforderungen:

Standard-Workflows

: Ideal für lang laufende, überprüfbare Workflows, die eine Semantik erfordern, die nur einmal ausgeführt wird. Standard-Workflows können bis zu einem Jahr lang ausgeführt werden, bieten einen detaillierten Ausführungsverlauf und unterstützen visuelles Debugging. Sie eignen sich für Prozesse wie Auftragsabwicklung, Datenverarbeitungspipelines oder mehrstufige Analytikaufgaben.

Express-Workflows

: Konzipiert für high-event-rate kurzzeitige Workloads mit Ausführungssemantik. at-least-once Express-Workflows können bis zu fünf Minuten lang ausgeführt werden und eignen sich ideal für die Verarbeitung großer Ereignismengen, Streamingdatentransformationen oder IoT-Datenerfassungsszenarien. Sie bieten einen höheren Durchsatz und potenziell geringere Kosten im Vergleich zu Standard-Workflows.

> **Note**
>
> Weitere Informationen zu den Workflow-Typen von Step Functions finden Sie unter Auswahl des Workflow-Typs in Step Functions.

Innerhalb dieser Workflows bietet Step Functions zwei Arten von Kartenstatus für die parallele Verarbeitung:

Inline Map

Verarbeitet Elemente aus einem JSON-Array innerhalb des Ausführungsverlaufs des übergeordneten Workflows. Inline Map unterstützt bis zu 40 gleichzeitige Iterationen und eignet sich für kleinere Datensätze oder wenn Sie die gesamte Verarbeitung innerhalb einer einzigen Ausführung durchführen müssen. Weitere Informationen finden Sie unter Verwenden des Kartenstatus im Inline-Modus.

Distributed Map

Ermöglicht die Verarbeitung großer paralleler Workloads durch Iteration über Datensätze, die 256 KiB überschreiten oder mehr als 40 gleichzeitige Iterationen erfordern. Mit Unterstützung für bis zu 10 000 parallele untergeordnete Workflow-Ausführungen eignet sich Distributed Map hervorragend für die Verarbeitung von semistrukturierten Daten, die in Amazon S3 gespeichert sind, wie z. B. JSON- oder CSV-Dateien, und ist somit ideal für die Stapelverarbeitung und ETL-Vorgänge. Weitere Informationen finden Sie unter Verwenden des Kartenstatus im Distributed-Modus.

Durch die Kombination dieser Workflow-Typen und Kartenstatus bietet Step Functions ein flexibles und leistungsstarkes Toolset für die Orchestrierung komplexer Serverless-Anwendungen, von kleinen Vorgängen bis hin zu großen Datenverarbeitungspipelines.

Informationen zu den ersten Schritten mit Lambda mit Step Functions finden Sie unter Orchestrieren von Lambda-Funktionen mit Step Functions.

Verwaltung von Ereignissen mit und Scheduler EventBridge EventBridge

Amazon EventBridge ist ein Event-Bus-Service, der Sie beim Aufbau ereignisgesteuerter Architekturen unterstützt. Es leitet Ereignisse zwischen AWS Diensten, integrierten Anwendungen und SaaS-Anwendungen (Software as a Service) weiter. EventBridge Scheduler ist ein serverloser Scheduler, mit dem Sie Aufgaben von einem zentralen Dienst aus erstellen, ausführen und verwalten können. So können Sie Lambda-Funktionen nach einem Zeitplan mithilfe von Cron- und Rate-Ausdrücken aufrufen oder einmalige Aufrufe konfigurieren.

Amazon EventBridge und EventBridge Scheduler helfen Ihnen beim Aufbau ereignisgesteuerter Architekturen mit Lambda. EventBridge leitet Ereignisse zwischen AWS Diensten, integrierten Anwendungen und SaaS-Anwendungen weiter, während EventBridge Scheduler spezielle Planungsfunktionen für den wiederkehrenden oder einmaligen Aufruf von Lambda-Funktionen bietet.

Diese Services bieten mehrere wichtige Funktionen für die Arbeit mit Lambda-Funktionen:

- Erstellen Sie Regeln, die Ereignisse abgleichen und an Lambda-Funktionen weiterleiten, indem Sie EventBridge
- Richten Sie mit Scheduler wiederkehrende Funktionsaufrufe mithilfe von Cron- und Rate-Ausdrücken ein EventBridge
- Konfigurieren Sie einmalige Funktionsaufrufe zu bestimmten Daten und Uhrzeiten.
- Definieren Sie flexible Zeitfenster und Wiederholungsrichtlinien für geplante Aufrufe.

Weitere Informationen finden Sie unter Aufrufen einer Lambda-Funktion nach einem Zeitplan.

Langlebige Lambda-Funktionen

Mit den dauerhaften Lambda-Funktionen können Sie belastbare mehrstufige Anwendungen und KI-Workflows erstellen, die bis zu einem Jahr lang ausgeführt werden können und gleichzeitig trotz Unterbrechungen einen zuverlässigen Fortschritt gewährleisten. Wenn eine dauerhafte Funktion ausgeführt wird, wird dieser gesamte Lebenszyklus als dauerhafte Ausführung bezeichnet. Dabei werden Prüfpunkte verwendet, um den Fortschritt zu verfolgen und Fehler automatisch durch Wiederholung wiederherzustellen, wobei von Anfang an erneut ausgeführt wird, während abgeschlossene Arbeiten übersprungen werden.

In jeder Funktion verwenden Sie dauerhafte Betriebsabläufe als grundlegende Bausteine. Schritte führen Geschäftslogik mit integrierten Wiederholungsversuchen und Fortschrittsverfolgung aus, während Wartezeiten die Ausführung unterbrechen, ohne dass Rechenkosten anfallen. Dadurch eignen sie sich ideal für lang andauernde Prozesse wie human-in-the-loop Workflows oder das Abfragen externer Abhängigkeiten. Ganz gleich, ob Sie Bestellungen bearbeiten, Microservices koordinieren oder KI-Anwendungen orchestrieren — langlebige Funktionen behalten den Status automatisch bei und stellen sich nach Ausfällen wieder her, während Sie Code in vertrauten Programmiersprachen schreiben.

Wichtigste Vorteile

Schreiben Sie auf natürliche Weise belastbaren Code: Mit vertrauten Programmierkonstrukten schreiben Sie Code, der Fehler automatisch behandelt. Integriertes Checkpointing, transparente Wiederholungsversuche und automatische Wiederherstellung sorgen dafür, dass Ihre Geschäftslogik sauber und zielgerichtet bleibt.

Zahlen Sie nur für das, was Sie tatsächlich nutzen: Während Wartezeiten wird Ihre Funktion unterbrochen, ohne dass Rechenkosten anfallen. Bei Workflows mit langer Laufzeit, die Stunden oder Tage warten, zahlen Sie nur für die tatsächliche Bearbeitungszeit, nicht für ungenutzte Wartezeiten.

Einfache Bedienung: Das serverlose Modell von Lambda ermöglicht automatische Skalierung scale-to-zero, auch ohne Verwaltung der Infrastruktur. Zuverlässige Funktionen kümmern sich automatisch um die Statusverwaltung, die Wiederholungslogik und die Wiederherstellung nach einem Ausfall, wodurch der Betriebsaufwand reduziert wird.

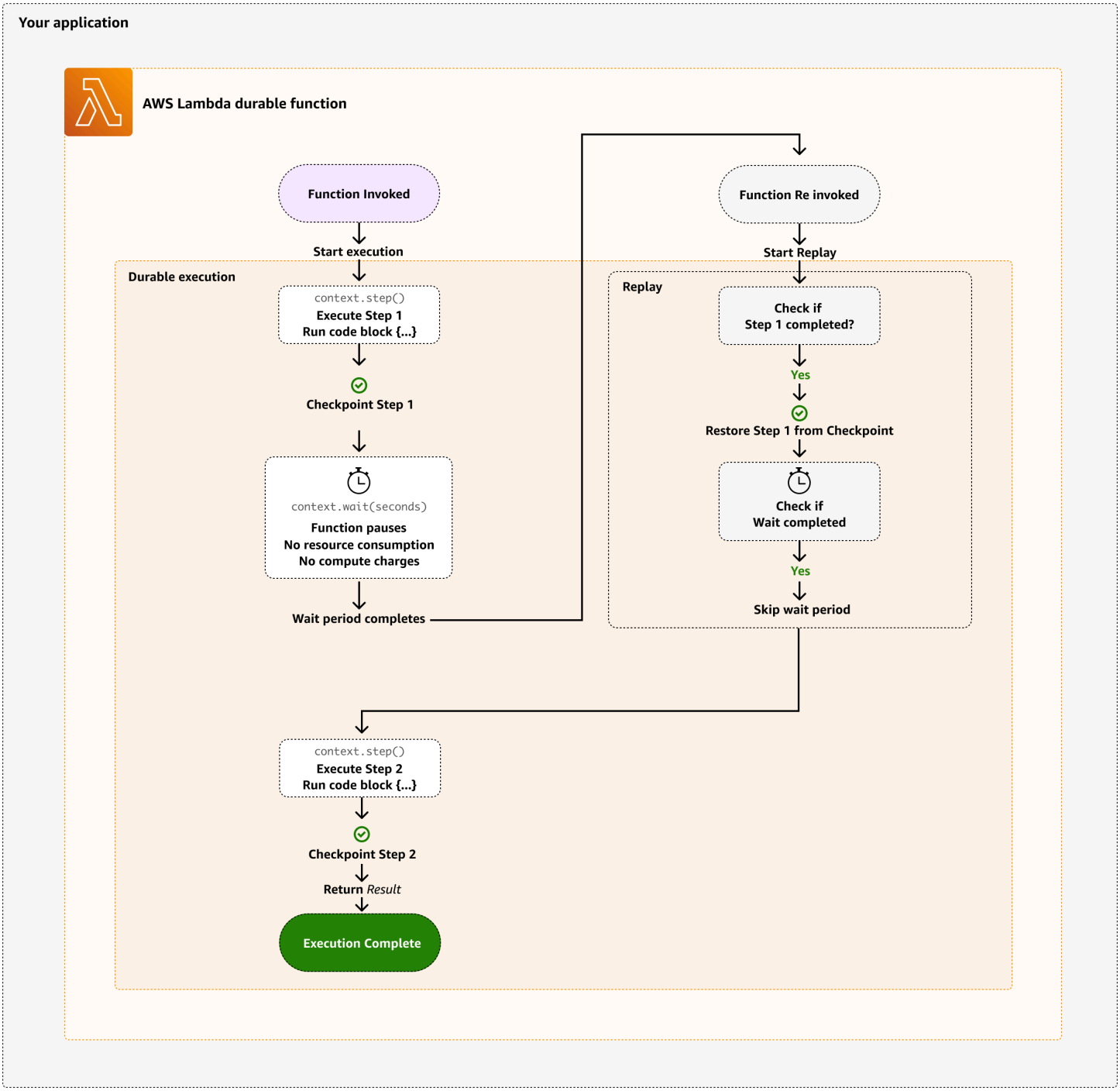
Funktionsweise

Unter der Haube sind langlebige Funktionen reguläre Lambda-Funktionen, die einen checkpoint/replay Mechanismus verwenden, um den Fortschritt zu verfolgen und lang andauernde Operationen über benutzerdefinierte Aufhängungspunkte zu unterstützen, die allgemein als langlebige Ausführung bezeichnet werden. Wenn eine dauerhafte Funktion nach einem Wartezeitpunkt oder einer Unterbrechung wie Wiederholungsversuchen wieder aufgenommen wird, führt das System eine Wiederholung durch. Während der Wiedergabe wird Ihr Code von Anfang an ausgeführt, überspringt jedoch abgeschlossene Prüfpunkte und verwendet gespeicherte Ergebnisse, anstatt abgeschlossene Operationen erneut auszuführen. Dieser Wiedergabemechanismus gewährleistet Konsistenz und ermöglicht gleichzeitig Ausführungen mit langer Laufzeit.

Wenn Ihre Funktion nach einer Pause oder Unterbrechung wieder aufgenommen wird, führt das System eine Wiederholung durch. Während der Wiedergabe wird Ihr Code von Anfang an ausgeführt, überspringt jedoch abgeschlossene Prüfpunkte und verwendet gespeicherte Ergebnisse, anstatt abgeschlossene Operationen erneut auszuführen. Dieser Wiedergabemechanismus gewährleistet Konsistenz und ermöglicht gleichzeitig Ausführungen mit langer Laufzeit.

Um diesen checkpoint-and-replay Mechanismus in Ihren Anwendungen zu nutzen, bietet Lambda ein langlebiges Ausführungs-SDK. Das SDK reduziert die Komplexität der Verwaltung von Checkpoints und der Wiedergabe und macht einfache Primitive verfügbar, die als dauerhafte Operationen bezeichnet werden und die Sie in Ihrem Code verwenden. Das SDK ist für JavaScript TypeScript, und Python verfügbar und lässt sich nahtlos in Ihren bestehenden Lambda-Entwicklungsworkflow integrieren.

Mit dem SDK umschließen Sie Ihren Lambda-Event-Handler, der dann DurableContext neben Ihrem Event einen bereitstellt. Dieser Kontext ermöglicht Ihnen den Zugriff auf dauerhafte Operationen wie Steps und Waits. Sie schreiben Ihre Funktionslogik als normalen sequentiellen Code, aber anstatt Dienste direkt aufzurufen, unterteilen Sie diese Aufrufe in Schritte, um automatische Checkpoints und Wiederholungen zu ermöglichen. Wenn Sie die Ausführung unterbrechen müssen, fügen Sie Wartezeiten hinzu, die Ihre Funktion unterbrechen, ohne dass Gebühren anfallen. Das SDK kümmert sich um die gesamte Komplexität der Statusverwaltung und Wiedergabe hinter den Kulissen, sodass Ihr Code sauber und lesbar bleibt.



Wann sollten langlebige Funktionen verwendet werden

Kurzlebige Koordination: Koordinieren Sie Zahlungen, Inventar und Versand für mehrere Dienste mit automatischem Rollback bei Ausfällen. Bearbeiten Sie Bestellungen durch Validierung, Zahlungsautorisierung, Inventarzuweisung und Erfüllung mit garantierter Fertigstellung.

Zuverlässige Zahlungsabwicklung: Sorgen Sie für stabile Zahlungsabläufe, die den Transaktionsstatus auch bei Ausfällen aufrechterhalten und Wiederholungsversuche automatisch abwickeln. Koordinieren Sie mehrstufige Autorisierung, Betrugskontrollen und Abrechnung zwischen den Zahlungsanbietern mit vollständiger Überprüfbarkeit aller Schritte.

Erstellen Sie zuverlässige KI-Workflows: Erstellen Sie mehrstufige KI-Workflows, die Modellanrufe verketten, menschliches Feedback einbeziehen und bei Ausfällen lang andauernde Aufgaben deterministisch abwickeln. Fahren Sie nach einer Unterbrechung automatisch fort und zahlen Sie nur für die aktive Ausführungszeit.

Orchestrieren Sie die komplexe Auftragsabwicklung: Koordinieren Sie die Auftragsabwicklung in allen Bestands-, Zahlungs-, Versand- und Benachrichtigungssystemen mit integrierter Ausfallsicherheit. Behandeln Sie Teilausfälle automatisch, behalten Sie den Auftragsstatus trotz Unterbrechungen bei und warten Sie effizient auf externe Ereignisse, ohne Rechenressourcen zu verbrauchen.

Automatisieren Sie mehrstufige Geschäftsabläufe: Erstellen Sie zuverlässige Workflows für das Onboarding von Mitarbeitern, die Genehmigung von Krediten und die Einhaltung gesetzlicher Vorschriften, die sich über Tage oder Wochen erstrecken. Behalten Sie den Workflow-Status bei Genehmigungen durch Mitarbeiter, Systemintegrationen und geplanten Aufgaben bei und sorgen Sie gleichzeitig für einen vollständigen Überblick über den Prozessstatus und die Historie.

Nächste Schritte

- Beginnen Sie mit langlebigen Funktionen
- Erkunden Sie das SDK für langlebige Ausführung
- Überwachen und debuggen Sie langlebige Funktionen
- Überprüfen Sie die Sicherheit und die Berechtigungen
- Folgen Sie den bewährten Methoden

Grundkonzepte

Lambda bietet eine dauerhafte Ausführung SDKs für JavaScript TypeScript, und Python. Diese SDKs bilden die Grundlage für die Erstellung dauerhafter Funktionen und bieten die Grundelemente, die Sie benötigen, um den Fortschritt zu überprüfen, Wiederholungsversuche zu verarbeiten und den Ausführungsablauf zu verwalten. Eine vollständige SDK-Dokumentation und Beispiele finden Sie unter JavaScript/TypeScript SDK und Python SDK unter GitHub.

Dauerhafte Ausführung

Eine dauerhafte Ausführung stellt den gesamten Lebenszyklus einer dauerhaften Lambda-Funktion dar und verwendet einen Checkpoint- und Replay-Mechanismus, um den Fortschritt der Geschäftslogik zu verfolgen, die Ausführung zu unterbrechen und die Wiederherstellung nach Ausfällen zu gewährleisten. Wenn Funktionen nach einer Unterbrechung oder Unterbrechung wieder aufgenommen werden, werden zuvor abgeschlossene Checkpoints erneut abgespielt und die Ausführung der Funktion wird fortgesetzt.

Der Lebenszyklus kann mehrere Aufrufe einer Lambda-Funktion beinhalten, um die Ausführung abzuschließen, insbesondere nach Unterbrechungen oder nach einer Wiederherstellung nach einem Ausfall. Dieser Ansatz ermöglicht es, dass Ihre Funktion über längere Zeiträume (bis zu einem Jahr) ausgeführt werden kann und gleichzeitig trotz Unterbrechungen ein zuverlässiger Fortschritt gewährleistet wird.

Wie funktioniert Replay

Lambda führt ein laufendes Protokoll aller dauerhaften Operationen (Schritte, Wartezeiten und andere Operationen), während Ihre Funktion ausgeführt wird. Wenn Ihre Funktion angehalten werden muss oder auf eine Unterbrechung stößt, speichert Lambda dieses Checkpoint-Log und stoppt die Ausführung. Wenn es Zeit ist, fortzufahren, ruft Lambda Ihre Funktion erneut von Anfang an auf und spielt das Checkpoint-Protokoll ab, wobei abgeschlossene Operationen durch gespeicherte Werte ersetzt werden. Das bedeutet, dass Ihr Code erneut ausgeführt wird, aber zuvor abgeschlossene Schritte werden nicht erneut ausgeführt. Ihre gespeicherten Ergebnisse werden stattdessen verwendet.

Dieser Wiedergabemechanismus ist für das Verständnis langlebiger Funktionen von grundlegender Bedeutung. Ihr Code muss während der Wiedergabe deterministisch sein, was bedeutet, dass er bei denselben Eingaben dieselben Ergebnisse liefert. Vermeiden Sie Operationen mit Nebeneffekten (wie das Generieren von Zufallszahlen oder das Abrufen der aktuellen Uhrzeit) außerhalb von Schritten, da diese bei der Wiedergabe unterschiedliche Werte erzeugen und zu nicht deterministischem Verhalten führen können.

DurableContext

DurableContextist das Kontextobjekt, das Ihre dauerhafte Funktion empfängt. Es bietet Methoden für dauerhafte Operationen wie Steps und Waits, mit denen Checkpoints erstellt und der Ausführungsablauf gesteuert wird.

Ihre dauerhafte Funktion erhält `DurableContext` anstelle des Standard-Lambda-Kontextes einen:

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
```

Das Python-SDK für dauerhafte Funktionen verwendet synchrone Methoden und unterstützt `await` diese nicht. Das TypeScript SDK verwendet`async/await`.

Schritte

Steps führt Geschäftslogik mit integrierten Wiederholungsversuchen und automatischem Checkpointing aus. Jeder Schritt speichert sein Ergebnis, sodass Ihre Funktion nach Unterbrechungen mit jedem abgeschlossenen Schritt wieder aufgenommen werden kann.

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))
```

Warte, Staaten

Wartezustände sind geplante Pausen, in denen Ihre Funktion nicht mehr läuft (und den Ladevorgang beendet), bis es Zeit ist, fortzufahren. Verwenden Sie sie, um auf Zeiträume, externe Rückrufe oder bestimmte Bedingungen zu warten.

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)
)
```

Wenn Ihre Funktion auf eine Wartezeit stößt oder angehalten werden muss, speichert Lambda das Checkpoint-Log und stoppt die Ausführung. Wenn es Zeit ist, fortzufahren, ruft Lambda Ihre Funktion erneut auf und spielt das Checkpoint-Protokoll erneut ab, wobei abgeschlossene Operationen durch gespeicherte Werte ersetzt werden.

Für komplexere Workflows verfügen langlebige Lambda-Funktionen auch über erweiterte Operationen wie `parallel()` die gleichzeitige Ausführung, die Verarbeitung von Arrays, `map()` für verschachtelte Operationen und `runInChildContext()` für Abfragen. `waitForCondition()` In den Beispielen finden Sie ausführliche Beispiele und Anleitungen zur Verwendung der einzelnen Operationen.

Andere Funktionen aufrufen

Invoke ermöglicht es einer dauerhaften Funktion, andere Lambda-Funktionen aufzurufen und auf deren Ergebnisse zu warten. Die aufrufende Funktion wird angehalten, während die aufgerufene Funktion ausgeführt wird, wodurch ein Checkpoint erstellt wird, der das Ergebnis beibehält. Auf diese Weise können Sie modulare Workflows erstellen, in denen spezielle Funktionen bestimmte Aufgaben erledigen.

Wird verwendet`context.invoke()`, um andere Funktionen aus Ihrer dauerhaften Funktion heraus aufzurufen. Der Aufruf hat einen Checkpoint. Wenn Ihre Funktion also unterbrochen wird, nachdem die aufgerufene Funktion abgeschlossen ist, wird sie mit dem gespeicherten Ergebnis fortgesetzt, ohne die Funktion erneut aufzurufen.

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'
)
```

Die aufgerufene Funktion kann entweder eine dauerhafte oder eine Standard-Lambda-Funktion sein. Wenn Sie eine dauerhafte Funktion aufrufen, wartet die aufrufende Funktion, bis die vollständige Ausführung der dauerhaften Funktion abgeschlossen ist. Dieses Muster ist in Microservices-Architekturen üblich, bei denen jede Funktion eine bestimmte Domäne verwaltet, sodass Sie komplexe Workflows aus speziellen, wiederverwendbaren Funktionen zusammenstellen können.

> **Note**
>
> Kontoübergreifende Aufrufe werden nicht unterstützt. Die aufgerufene Funktion muss sich in demselben AWS Konto wie die aufrufende Funktion befinden.

Dauerhafte Funktionskonfiguration

Dauerhafte Funktionen verfügen über spezifische Konfigurationseinstellungen, die das Ausführungsverhalten und die Datenspeicherung steuern. Diese Einstellungen unterscheiden sich von der standardmäßigen Lambda-Funktionskonfiguration und gelten für den gesamten Durable Execution-Lebenszyklus.

Das DurableConfigObjekt definiert die Konfiguration für dauerhafte Funktionen:

```
{
  "ExecutionTimeout": Integer,
  "RetentionPeriodInDays": Integer
}
```

Ausführungstimeout

Das Ausführungs-Timeout steuert, wie lange eine dauerhafte Ausführung von Anfang bis Ende laufen kann. Dies unterscheidet sich vom Lambda-Funktions-Timeout, das steuert, wie lange ein einzelner Funktionsaufruf ausgeführt werden kann.

Eine dauerhafte Ausführung kann mehrere Lambda-Funktionsaufrufen umfassen, während sie Checkpoints, Wartezeiten und Wiederholungen durchläuft. Das Ausführungs-Timeout gilt für die gesamte verstrichene Zeit der dauerhaften Ausführung, nicht für einzelne Funktionsaufrufen.

Den Unterschied verstehen

Das Timeout der Lambda-Funktion (maximal 15 Minuten) begrenzt jeden einzelnen Aufruf Ihrer Funktion. Das dauerhafte Ausführungstimeout (maximal 1 Jahr) begrenzt die Gesamtzeit vom Start der Ausführung bis zum Abschluss, zum Fehlschlagen oder zum Timeout. Während dieses Zeitraums kann Ihre Funktion mehrmals aufgerufen werden, da sie Schritte verarbeitet, wartet und sich nach Ausfällen wiederherstellt.

Wenn Sie beispielsweise ein dauerhaftes Ausführungstimeout von 24 Stunden und ein Lambda-Funktions-Timeout von 5 Minuten festlegen:

- Jeder Funktionsaufruf muss innerhalb von 5 Minuten abgeschlossen sein
- Die gesamte dauerhafte Ausführung kann bis zu 24 Stunden dauern
- Ihre Funktion kann während dieser 24 Stunden viele Male aufgerufen werden
- Wartevorgänge werden nicht auf den Timeout der Lambda-Funktion angerechnet, sondern auf den Ausführungs-Timeout.

Sie können das Ausführungstimeout konfigurieren, wenn Sie eine dauerhafte Funktion mit der Lambda-Konsole erstellen, AWS CLI, oder. AWS SAM Wählen Sie in der Lambda-Konsole Ihre Funktion und dann Konfiguration, Dauerhafte Ausführung aus. Legen Sie den Wert für das Ausführungs-Timeout in Sekunden fest (Standard: 86400 Sekunden/24 Stunden, Minimum: 60 Sekunden, Maximum: 31536000 Sekunden/1 Jahr).

> **Note**
>
> Das Ausführungs-Timeout und das Lambda-Funktions-Timeout sind unterschiedliche Einstellungen. Das Timeout der Lambda-Funktion steuert, wie lange jeder einzelne Aufruf ausgeführt werden kann (maximal 15 Minuten). Das Ausführungs-Timeout steuert die gesamte verstrichene Zeit für die gesamte dauerhafte Ausführung (maximal 1 Jahr).

Aufbewahrungszeitraum

Die Aufbewahrungsfrist steuert, wie lange Lambda den Ausführungsverlauf und die Checkpoint-Daten nach Abschluss einer dauerhaften Ausführung aufbewahrt. Zu diesen Daten gehören die Schrittergebnisse, der Ausführungsstatus und das vollständige Checkpoint-Protokoll.

Nach Ablauf der Aufbewahrungsfrist löscht Lambda den Ausführungsverlauf und die Checkpoint-Daten. Sie können keine Ausführungsdetails mehr abrufen oder die Ausführung erneut abspielen. Die Aufbewahrungsfrist beginnt, wenn die Ausführung einen Terminalstatus erreicht hat (SUCCEED, FAILED, STOPPED oder TIMED_OUT).

Sie können den Aufbewahrungszeitraum konfigurieren, wenn Sie eine dauerhafte Funktion mit der Lambda-Konsole erstellen, AWS CLI, oder AWS SAM. Wählen Sie in der Lambda-Konsole Ihre Funktion und dann Konfiguration, Dauerhafte Ausführung aus. Legen Sie den Wert für den Aufbewahrungszeitraum in Tagen fest (Standard: 14 Tage, Minimum: 1 Tag, Maximum: 90 Tage).

Wählen Sie einen Aufbewahrungszeitraum, der auf Ihren Compliance-Anforderungen, Debugging-Anforderungen und Kostenerwägungen basiert. Längere Aufbewahrungszeiträume bieten mehr Zeit für Debugging und Audits, erhöhen jedoch die Speicherkosten.

Dauerhafte Lambda-Funktionen erstellen

Um mit dauerhaften Lambda-Funktionen zu beginnen, verwenden Sie die Lambda-Konsole, um eine dauerhafte Funktion zu erstellen. In wenigen Minuten können Sie eine dauerhafte Funktion erstellen und bereitstellen, die anhand von Schritten und Wartezeiten die Ausführung anhand von Checkpoints demonstriert.

Während der Durchführung des Tutorials lernen Sie grundlegende Konzepte für dauerhafte Funktionen kennen, z. B. wie Sie das `DurableContext` Objekt verwenden, Checkpoints mit Schritten erstellen und die Ausführung mit Wartezeiten unterbrechen. Außerdem erfahren Sie, wie die Wiedergabe funktioniert, wenn Ihre Funktion nach einer Wartezeit wieder aufgenommen wird.

Der Einfachheit halber erstellen Sie Ihre Funktion entweder mit der Python- oder Node.js-Laufzeit. Mit diesen interpretierten Sprachen können Sie Funktionscode direkt im integrierten Code-Editor der Konsole bearbeiten.

> **Tip**
>
> Weitere Informationen zum Erstellen von Serverless-Lösungen finden Sie im Serverless-Benutzerhandbuch.

Voraussetzungen

Melde dich an für ein AWS-Konto

Wenn Sie noch keine haben AWS-Konto, führen Sie die folgenden Schritte aus, um eine zu erstellen.

Um sich für eine anzumelden AWS-Konto

1. Öffnen Sie https://portal.aws.amazon.com/billing/die Anmeldung.
2. Folgen Sie den Online-Anweisungen.

   Während der Anmeldung erhalten Sie einen Telefonanruf oder eine Textnachricht und müssen einen Verifizierungscode über die Telefontasten eingeben.

   Wenn Sie sich für eine anmelden AWS-Konto, Root-Benutzer des AWS-Kontoswird eine erstellt. Der Root-Benutzer hat Zugriff auf alle AWS-Services und Ressourcen des Kontos. Als bewährte Sicherheitsmethode weisen Sie einem Benutzer Administratorzugriff zu und verwenden Sie nur den Root-Benutzer, um Aufgaben auszuführen, die Root-Benutzerzugriff erfordern.

AWS sendet Ihnen nach Abschluss des Anmeldevorgangs eine Bestätigungs-E-Mail. Sie können Ihre aktuellen Kontoaktivitäten jederzeit einsehen und Ihr Konto verwalten, indem Sie zu https://aws.amazon.com/gehen und Mein Konto auswählen.

Erstellen eines Benutzers mit Administratorzugriff

Nachdem Sie sich für einen angemeldet haben AWS-Konto, sichern Sie Ihren Root-Benutzer des AWS-Kontos AWS IAM Identity Center, aktivieren und erstellen Sie einen Administratorbenutzer, sodass Sie den Root-Benutzer nicht für alltägliche Aufgaben verwenden.

Sichern Sie Ihre Root-Benutzer des AWS-Kontos

1. Melden Sie sich AWS-Managementkonsoleals Kontoinhaber an, indem Sie Root-Benutzer auswählen und Ihre AWS-Konto E-Mail-Adresse eingeben. Geben Sie auf der nächsten Seite Ihr Passwort ein.

   Hilfe bei der Anmeldung mit dem Root-Benutzer finden Sie unter Anmelden als Root-Benutzer im AWS-Anmeldung Benutzerhandbuch zu.

2. Aktivieren Sie die Multi-Faktor-Authentifizierung (MFA) für den Root-Benutzer.

   Anweisungen finden Sie unter Aktivieren eines virtuellen MFA-Geräts für Ihren AWS-Konto Root-Benutzer (Konsole) im IAM-Benutzerhandbuch.

Erstellen eines Benutzers mit Administratorzugriff

1. Aktivieren Sie das IAM Identity Center.

   Anweisungen finden Sie unter Aktivieren AWS IAM Identity Center im AWS IAM Identity Center Benutzerhandbuch.

2. Gewähren Sie einem Administratorbenutzer im IAM Identity Center Benutzerzugriff.

   Ein Tutorial zur Verwendung von IAM-Identity-Center-Verzeichnis als Identitätsquelle finden Sie IAM-Identity-Center-Verzeichnis im Benutzerhandbuch unter Benutzerzugriff mit der Standardeinstellung konfigurieren.AWS IAM Identity Center

Anmelden als Administratorbenutzer

- Um sich mit Ihrem IAM-Identity-Center-Benutzer anzumelden, verwenden Sie die Anmelde-URL, die an Ihre E-Mail-Adresse gesendet wurde, als Sie den IAM-Identity-Center-Benutzer erstellt haben.

  Hilfe bei der Anmeldung mit einem IAM Identity Center-Benutzer finden Sie im AWS-Anmeldung Benutzerhandbuch unter Anmeldung beim AWS Access-Portal.

Weiteren Benutzern Zugriff zuweisen

1. Erstellen Sie im IAM-Identity-Center einen Berechtigungssatz, der den bewährten Vorgehensweisen für die Anwendung von geringsten Berechtigungen folgt.

Anweisungen hierzu finden Sie unter Berechtigungssatz erstellen im AWS IAM Identity Center Benutzerhandbuch.

2. Weisen Sie Benutzer einer Gruppe zu und weisen Sie der Gruppe dann Single Sign-On-Zugriff zu.

   Eine genaue Anleitung finden Sie unter Gruppen hinzufügen im AWS IAM Identity Center Benutzerhandbuch.

Erstellen Sie mit der Konsole eine dauerhafte Lambda-Funktion

In diesem Beispiel verarbeitet Ihre langlebige Funktion eine Bestellung in mehreren Schritten mit automatischem Checkpoint. Die Funktion verwendet ein JSON-Objekt, das eine Bestell-ID enthält, validiert die Bestellung, verarbeitet die Zahlung und bestätigt die Bestellung. Jeder Schritt wird automatisch überprüft. Wenn die Funktion unterbrochen wird, wird sie mit dem letzten abgeschlossenen Schritt fortgesetzt.

Ihre Funktion demonstriert auch eine Warteoperation, bei der die Ausführung für einen kurzen Zeitraum angehalten wird, um das Warten auf eine externe Bestätigung zu simulieren.

Um eine dauerhafte Funktion mit der Konsole zu erstellen

1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.
2. Wählen Sie Funktion erstellen.
3. Wählen Sie Verfassen von Grund auf aus.
4. Geben Sie im Bereich Grundlegende Informationen als Funktionsname **`myDurableFunction`** ein.
5. Wählen Sie für Runtime entweder Node.js 24 oder Python 3.14.
6. Wählen Sie Dauerhafte Ausführung aktivieren aus.

Lambda erstellt Ihre dauerhafte Funktion mit einer Ausführungsrolle, die Berechtigungen für Checkpoint-Operationen (`lambda:CheckpointDurableExecutionsundlambda:GetDurableExecutionState`) beinhaltet.

> **Note**
>
> Lambda-Laufzeiten enthalten das Durable Execution SDK, sodass Sie langlebige Funktionen ohne Paketierungsabhängigkeiten testen können. Wir empfehlen jedoch, das SDK in Ihr Bereitstellungspaket für die Produktion aufzunehmen. Dadurch wird die Versionskonsistenz gewährleistet und potenzielle Runtime-Updates vermieden, die Ihre Funktion beeinträchtigen könnten.

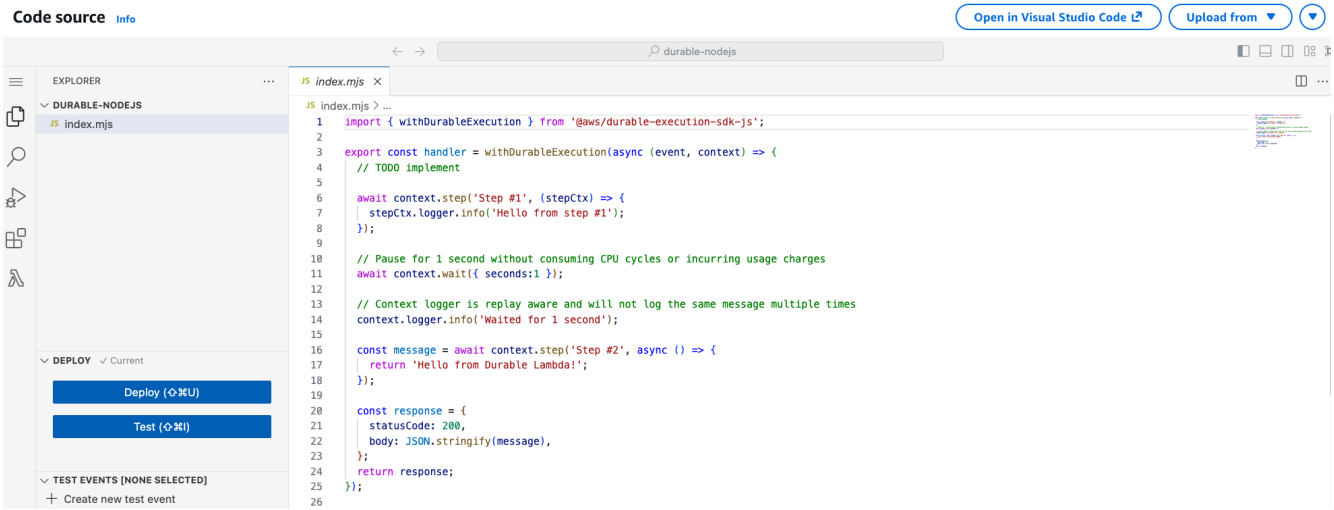
Verwenden Sie den integrierten Code-Editor der Konsole, um Ihren dauerhaften Funktionscode hinzuzufügen.

Node.js

So ändern Sie den Code in der Konsole

1. Wählen Sie die Registerkarte Code.

   Im integrierten Code-Editor der Konsole sollten Sie den von Lambda erstellten Funktionscode sehen. Wenn die Registerkarte index.mjs im Code-Editor nicht angezeigt wird, wählen Sie index.mjs im Datei-Explorer aus, wie im folgenden Diagramm gezeigt.

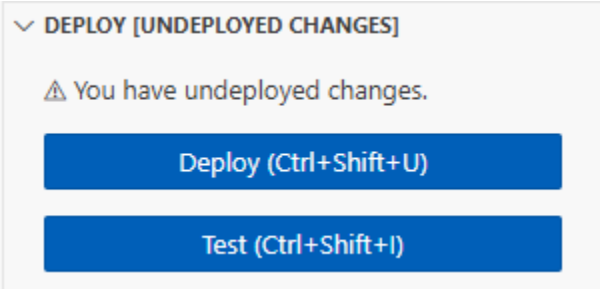


2. Fügen Sie den folgenden Code in die Registerkarte index.mjs ein und ersetzen Sie den von Lambda erstellten Code.

```
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. Wählen Sie im Abschnitt BEREITSTELLEN die Option Bereitstellen aus, um den Code Ihrer Funktion zu aktualisieren:



Verstehen Sie Ihren dauerhaften Funktionscode

Bevor Sie mit dem nächsten Schritt fortfahren, sollten wir uns den Funktionscode ansehen und die wichtigsten Konzepte für langlebige Funktionen verstehen.

- Der `withDurableExecution` Wrapper:

  Ihre langlebige Funktion ist umhüllt von. `withDurableExecution` Dieser Wrapper ermöglicht eine dauerhafte Ausführung, indem er das `DurableContext` Objekt bereitstellt und Checkpoint-Operationen verwaltet.
- Das `DurableContext` Objekt:

  Anstelle des Standard-Lambda-Kontextes erhält Ihre Funktion eine`DurableContext`. Dieses Objekt bietet Methoden für dauerhafte Operationen wie `step()` und `wait()` die Erstellung von Checkpoints.
- Schritte und Checkpoints:

  Jeder `context.step()` Aufruf erstellt vor und nach der Ausführung einen Checkpoint. Wenn Ihre Funktion unterbrochen wird, wird sie ab dem letzten abgeschlossenen Checkpoint wieder aufgenommen. Die Funktion führt abgeschlossene Schritte nicht erneut aus. Stattdessen werden ihre gespeicherten Ergebnisse verwendet.
- Operationen warten:

  Der `context.wait()` Aufruf unterbricht die Ausführung, ohne Rechenressourcen zu verbrauchen. Wenn das Warten abgeschlossen ist, ruft Lambda Ihre Funktion erneut auf und spielt das Checkpoint-Protokoll erneut ab, wobei abgeschlossene Schritte durch gespeicherte Werte ersetzt werden.
- Wiedergabemechanismus:

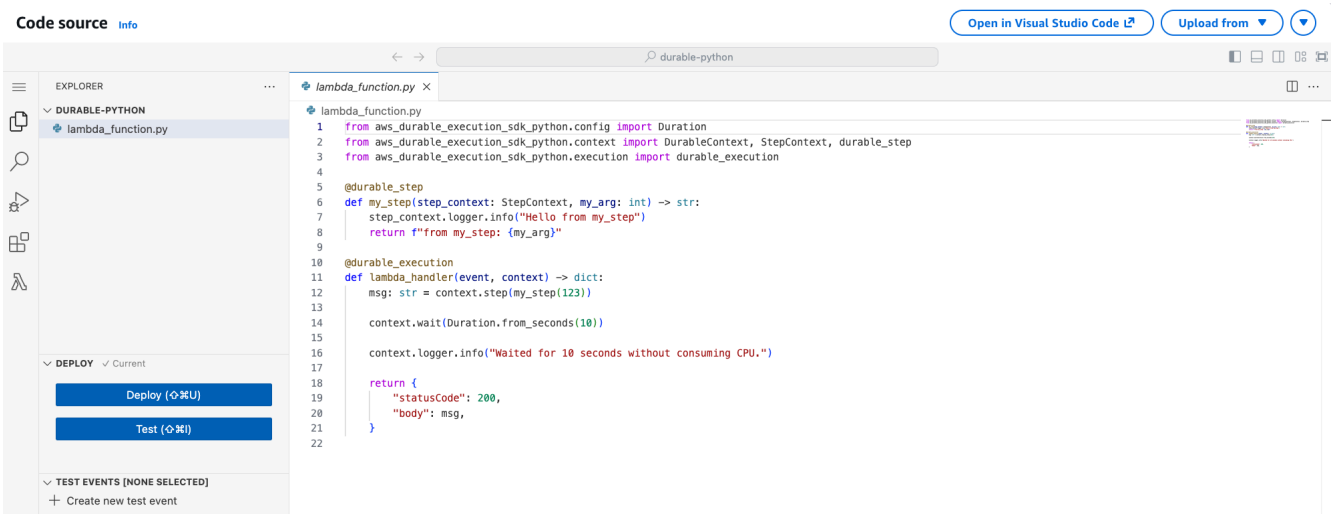
Wenn Ihre Funktion nach einer Wartezeit oder Unterbrechung wieder aufgenommen wird, führt Lambda Ihren Code von Anfang an aus. Abgeschlossene Schritte werden jedoch nicht erneut ausgeführt. Lambda gibt ihre Ergebnisse aus dem Checkpoint-Log wieder. Aus diesem Grund muss Ihr Code deterministisch sein.

Python

So ändern Sie den Code in der Konsole

1. Wählen Sie die Registerkarte Code.

   Im integrierten Code-Editor der Konsole sollten Sie den von Lambda erstellten Funktionscode sehen. Wenn die Registerkarte lambda_function.py im Code-Editor nicht angezeigt wird, wählen Sie lambda_function.py im Datei-Explorer aus, wie im folgenden Diagramm dargestellt.

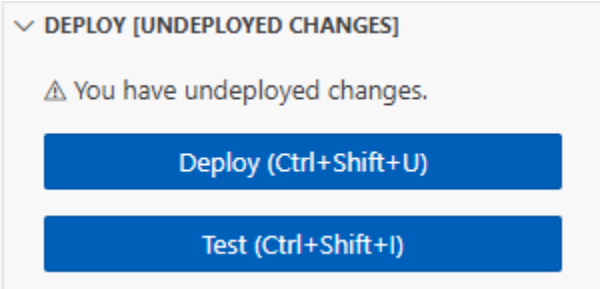


2. Fügen Sie den folgenden Code in die Registerkarte lambda_function.py ein und ersetzen Sie den von Lambda erstellten Code.

```
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. Wählen Sie im Abschnitt BEREITSTELLEN die Option Bereitstellen aus, um den Code Ihrer Funktion zu aktualisieren:



Verstehen Sie Ihren dauerhaften Funktionscode

Bevor Sie mit dem nächsten Schritt fortfahren, sollten wir uns den Funktionscode ansehen und die wichtigsten Konzepte für langlebige Funktionen verstehen.

- Der `@durable_execution` Dekorateur:

  Ihre Handler-Funktion ist mit `@durable_execution` dekoriert. Dieser Decorator ermöglicht eine dauerhafte Ausführung, indem er das `DurableContext` Objekt bereitstellt und Checkpoint-Operationen verwaltet.
- Der `@durable_step` Dekorateur:

  Jede Schrittfunktion ist mit `@durable_step` dekoriert. Dieser Dekorateur kennzeichnet die Funktion als dauerhaften Schritt, der Checkpoints erzeugt.
- Das `DurableContext` Objekt:

  Anstelle des Standard-Lambda-Kontextes erhält Ihre Funktion eine`DurableContext`. Dieses Objekt bietet Methoden für dauerhafte Operationen wie `step()` und `wait()` die Erstellung von Checkpoints.
- Schritte und Checkpoints:

  Jeder `context.step()` Aufruf erstellt vor und nach der Ausführung einen Checkpoint. Wenn Ihre Funktion unterbrochen wird, wird sie ab dem letzten abgeschlossenen Checkpoint wieder aufgenommen. Die Funktion führt abgeschlossene Schritte nicht erneut aus. Stattdessen werden ihre gespeicherten Ergebnisse verwendet.
- Operationen warten:

  Der `context.wait()` Aufruf unterbricht die Ausführung, ohne Rechenressourcen zu verbrauchen. Wenn das Warten abgeschlossen ist, ruft Lambda Ihre Funktion erneut auf und

spielt das Checkpoint-Protokoll erneut ab, wobei abgeschlossene Schritte durch gespeicherte Werte ersetzt werden.

- Das Python-SDK ist synchron:

  Beachten Sie, dass das Python-SDK nicht verwendet`await`. Alle dauerhaften Operationen sind synchrone Methodenaufrufen.

Rufen Sie die dauerhafte Funktion mit dem Code-Editor der Konsole auf

Dauerhafte Funktionen erfordern einen qualifizierten ARN für den Aufruf. Bevor Sie Ihre dauerhafte Funktion aufrufen können, müssen Sie eine Version veröffentlichen.

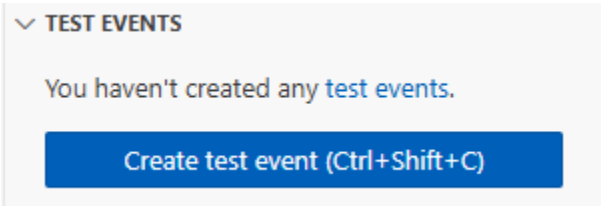
Um eine Version Ihrer Funktion zu veröffentlichen

1. Wählen Sie den Tab Versionen.
2. Wählen Sie Publish new version (Neue Version veröffentlichen) aus.
3. Geben Sie als Versionsbeschreibung **`Initial version`** (optional) ein.
4. Wählen Sie Publish.
5. Lambda erstellt Version 1 Ihrer Funktion. Beachten Sie, dass die Funktion ARN jetzt `:1` am Ende steht, was darauf hinweist, dass es sich um Version 1 handelt.

Erstellen Sie nun ein Testereignis, das an Ihre Funktion gesendet werden soll. Das Ereignis ist ein Dokument im JSON-Format, das eine Bestell-ID enthält.

So erstellen Sie das Testereignis

1. Wählen Sie im Abschnitt TESTEREIGNISSE des Konsolen-Code-Editors die Option Testereignis erstellen aus.



2. Geben Sie als Event name (Ereignisname) die Zeichenfolge **`myTestEvent`** ein.
3. Ersetzen Sie im Abschnitt Event JSON das Standard-JSON durch Folgendes:

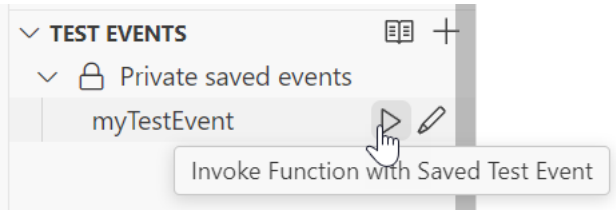
```
{
```

```
  "orderId": "order-12345"
}
```

4. Wählen Sie Speichern.

Um Ihre dauerhafte Funktion zu testen und die Ausführung anzusehen

Wählen Sie im Abschnitt TESTEREIGNISSE des Konsolencode-Editors das Ausführungssymbol neben Ihrem Testereignis:



Ihre dauerhafte Funktion wird ausgeführt. Da sie eine Wartezeit von 10 Sekunden beinhaltet, wird der erste Aufruf schnell abgeschlossen, und die Funktion wird nach Ablauf der Wartezeit wieder aufgenommen. Sie können den Ausführungsfortschritt auf der Registerkarte Dauerhafte Ausführungen einsehen.

Um die Ausführung Ihrer dauerhaften Funktion einzusehen

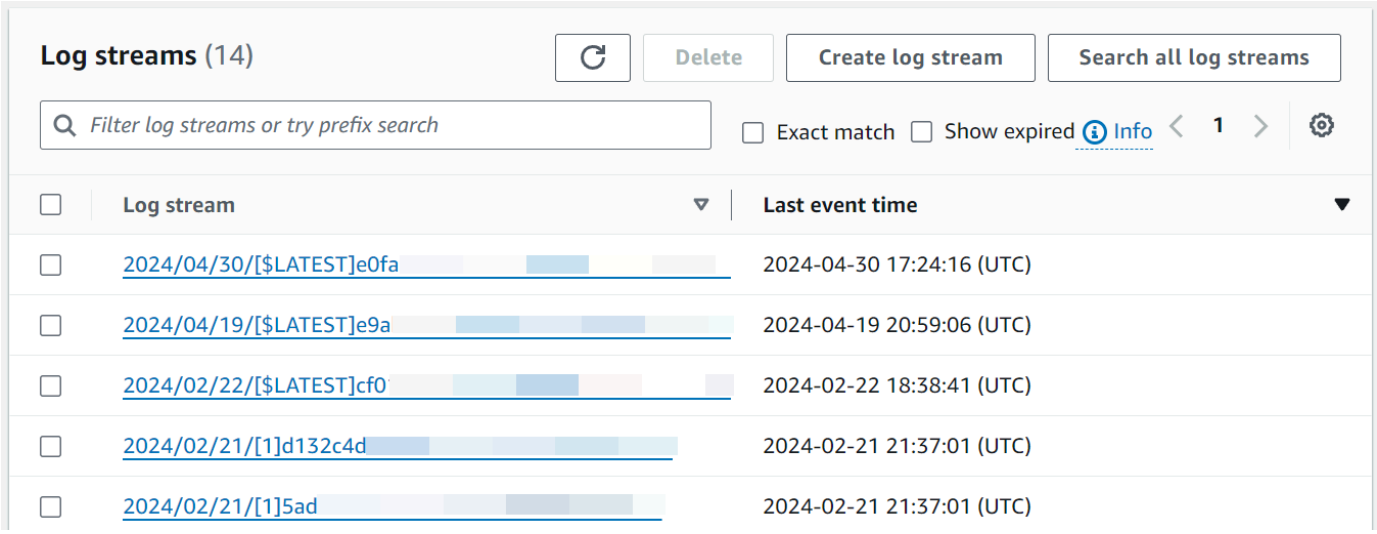
1. Wählen Sie die Registerkarte Dauerhafte Ausführungen.
2. Suchen Sie in der Liste nach Ihrer Hinrichtung. Die Ausführung zeigt den aktuellen Status an (Wird ausgeführt, Erfolgreich oder Fehlgeschlagen).
3. Wählen Sie die Ausführungs-ID, um Details anzuzeigen, darunter:
   - Ausführungszeitplan, der anzeigt, wann jeder Schritt abgeschlossen ist
   - Verlauf des Checkpoints
   - Wartezeiten
   - Ergebnisse des Schritts

Sie können die Protokolle Ihrer Funktion auch unter Logs einsehen, um die Konsolenausgabe der einzelnen Schritte zu sehen. CloudWatch

Um die Aufrufaufzeichnungen Ihrer Funktion in CloudWatch Logs einzusehen

1. Öffnen Sie die Seite Protokollgruppen der CloudWatch Konsole.

2. Wählen Sie die Protokollgruppe für Ihre Funktion (`/aws/lambda/myDurableFunction`) aus.
3. Scrollen Sie nach unten und wählen Sie den Protokollstream für die Funktionsaufrufen aus, die Sie sich ansehen möchten.



   Sie sollten Protokolleinträge für jeden Aufruf Ihrer Funktion sehen, einschließlich der ersten Ausführung und der Wiederholung nach dem Warten.

Bereinigen

Wenn Sie mit der dauerhaften Beispielfunktion fertig sind, löschen Sie sie. Sie können auch die Protokollgruppe löschen, in der die Protokolle der Funktion gespeichert sind, sowie die von der Konsole erstellte Ausführungsrolle.

So löschen Sie die Lambda-Funktion:

1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.
2. Wählen Sie die Funktion aus, die Sie erstellt haben.
3. Wählen Sie Aktionen, Löschen aus.
4. Geben Sie **`confirm`** in das Texteingabefeld ein und wählen Sie Delete (Löschen) aus.

So löschen Sie die Protokollgruppe

1. Öffnen Sie die Seite Log Groups (Protokollgruppen) der CloudWatch-Konsole.
2. Wählen Sie die Protokollgruppe der Funktion (`/aws/lambda/myDurableFunction`).
3. Wählen Sie Actions (Aktionen), Delete log group(s) (Protokollgruppe(n) löschen) aus.

4. Wählen Sie im Dialogfeld Delete log group(s) (Protokollgruppe(n) löschen) die Option Delete (Löschen) aus.

So löschen Sie die Ausführungsrolle

1. Öffnen Sie die Seite Rollen der AWS Identity and Access Management (IAM-) Konsole.
2. Wählen Sie die Ausführungsrolle der Funktion aus (zum Beispiel `myDurableFunction-role-31exxmpl`).
3. Wählen Sie Löschen aus.
4. Geben Sie im Dialogfenster Delete role (Rolle löschen) den Namen der Rolle ein und wählen Sie anschließend Delete (Löschen) aus.

Zusätzliche Ressourcen und nächste Schritte

Nachdem Sie mit der Konsole eine einfache, dauerhafte Funktion erstellt und getestet haben, gehen Sie wie folgt vor:

- Erfahren Sie mehr über gängige Anwendungsfälle für langlebige Funktionen, darunter verteilte Transaktionen, Auftragsabwicklung und Workflows zur Überprüfung durch Mitarbeiter. Sehen Sie sich Beispiele an.
- Erfahren Sie, wie Sie die Ausführung dauerhafter Funktionen anhand von CloudWatch Metriken und Ausführungshistorie überwachen können. Siehe Überwachung und Debuggen.
- Erfahren Sie, wie Sie langlebige Funktionen synchron und asynchron aufrufen und Ausführungen mit langer Laufzeit verwalten. Siehe Dauerhafte Funktionen aufrufen.
- Folgen Sie den bewährten Methoden zum Schreiben von deterministischem Code, zur Verwaltung von Checkpoint-Größen und zur Kostenoptimierung. Siehe Bewährte Methoden.
- Erfahren Sie, wie Sie langlebige Funktionen lokal und in der Cloud testen können. Siehe Testen langlebiger Funktionen.

Stellen Sie langlebige Lambda-Funktionen bereit und rufen Sie sie auf mit dem AWS CLI

Verwenden Sie die AWS CLI , um dauerhafte Lambda-Funktionen mit imperativen Befehlen zu erstellen und bereitzustellen. Mit diesem Ansatz haben Sie die direkte Kontrolle über jeden Schritt des Bereitstellungsprozesses.

Voraussetzungen

- Installieren und Konfigurieren der AWS CLI. Anweisungen finden Sie unter Installation von AWS CLI.
- Erstellen Sie ein Bereitstellungspaket mit Ihrem Funktionscode und dem Durable Execution SDK.
- Erstellen Sie eine IAM-Ausführungsrolle mit Checkpoint-Berechtigungen.

Erstellen der Ausführungsrolle

Erstellen Sie eine IAM-Rolle mit Berechtigungen für grundlegende Lambda-Ausführungs- und Checkpoint-Operationen.

So erstellen Sie eine Ausführungsrolle

1. Erstellen Sie ein Dokument mit Vertrauensrichtlinien, das es Lambda ermöglicht, die Rolle zu übernehmen. Speichern Sie das unter: `trust-policy.json`

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "lambda.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
```

2. Erstellen Sie die Rolle:

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

3. Hängen Sie die grundlegende Ausführungsrichtlinie an:

```
aws iam attach-role-policy \
  --role-name durable-function-role \
```

```
  --policy-arn arn:aws:iam::aws:policy/service-role/AWSLambdaBasicExecutionRole
```

4. Erstellen Sie eine Richtlinie für Checkpoint-Operationen. Speichern Sie das unter: `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. Erstellen Sie die Checkpoint-Richtlinie und hängen Sie sie an:

```
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
```

Erstellen Sie die dauerhafte Funktion

Erstellen Sie Ihre dauerhafte Funktion mit dem `--durable-config` Parameter.

Um eine dauerhafte Funktion zu erstellen

1. Packen Sie Ihren Funktionscode mit Abhängigkeiten in eine ZIP-Datei:

```
zip -r function.zip index.mjs node_modules/
```

2. Erstellen Sie die Funktion mit aktivierter dauerhafter Ausführung:

```
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**
>
> Sie können die dauerhafte Ausführung nur aktivieren, wenn Sie die Funktion erstellen. Sie können es nicht für bestehende Funktionen aktivieren.

Veröffentlichen Sie eine Version

Dauerhafte Funktionen benötigen einen qualifizierten ARN (mit Version oder Alias) für den Aufruf. Veröffentlichen Sie eine Version Ihrer Funktion:

```
aws lambda publish-version \
  --function-name myDurableFunction \
  --description "Initial version"
```

Der Befehl gibt die Version ARN zurück. Notieren Sie sich die Versionsnummer (z. B. `:1`) am Ende des ARN.

Erstellen Sie optional einen Alias, der auf die Version verweist:

```
aws lambda create-alias \
  --function-name myDurableFunction \
  --name prod \
  --function-version 1
```

Rufen Sie die dauerhafte Funktion auf

Rufen Sie Ihre dauerhafte Funktion mit dem qualifizierten ARN (Version oder Alias) auf.

Synchroner Aufruf

Verwenden Sie für Ausführungen, die innerhalb von 15 Minuten abgeschlossen sind, den synchronen Aufruf:

```
aws lambda invoke \
  --function-name myDurableFunction:1 \
  --payload '{"orderId": "order-12345"}' \
  --cli-binary-format raw-in-base64-out \
  response.json
```

Oder mit einem Alias:

```
aws lambda invoke \
  --function-name myDurableFunction:prod \
  --payload '{"orderId": "order-12345"}' \
  --cli-binary-format raw-in-base64-out \
  response.json
```

Asynchroner Aufruf

Verwenden Sie für lang andauernde Ausführungen den asynchronen Aufruf:

```
aws lambda invoke \
  --function-name myDurableFunction:prod \
  --invocation-type Event \
  --payload '{"orderId": "order-12345"}' \
  --cli-binary-format raw-in-base64-out \
  response.json
```

Bei asynchronem Aufruf kehrt Lambda sofort zurück. Die Funktion wird weiterhin im Hintergrund ausgeführt.

Verwaltet dauerhafte Ausführungen

Verwenden Sie die folgenden Befehle, um die Ausführung dauerhafter Funktionen zu verwalten und zu überwachen.

Ausführungen auflisten

Listet alle Ausführungen für eine dauerhafte Funktion auf:

```
aws lambda list-durable-executions \
```

```
  --function-name myDurableFunction:prod
```

Details zur Ausführung abrufen

Rufen Sie Details zu einer bestimmten Ausführung ab:

```
aws lambda get-durable-execution \
  --function-name myDurableFunction:prod \
  --execution-id exec-abc123
```

Ausführungshistorie abrufen

Den Checkpoint-Verlauf für eine Ausführung anzeigen:

```
aws lambda get-durable-execution-history \
  --function-name myDurableFunction:prod \
  --execution-id exec-abc123
```

Stoppen Sie eine Ausführung

Stoppen Sie eine laufende dauerhafte Ausführung:

```
aws lambda stop-durable-execution \
  --function-name myDurableFunction:prod \
  --execution-id exec-abc123
```

Funktionscode aktualisieren

Aktualisieren Sie Ihren dauerhaften Funktionscode und veröffentlichen Sie eine neue Version:

Um eine neue Version zu aktualisieren und zu veröffentlichen

1. Aktualisieren Sie den Funktionscode:

   ```
   aws lambda update-function-code \
     --function-name myDurableFunction \
     --zip-file fileb://function.zip
   ```

2. Warten Sie, bis das Update abgeschlossen ist:

   ```
   aws lambda wait function-updated \
   ```

```
  --function-name myDurableFunction
```

3. Veröffentlichen Sie eine neue Version:

```
aws lambda publish-version \
  --function-name myDurableFunction \
  --description "Updated order processing logic"
```

4. Aktualisieren Sie den Alias so, dass er auf die neue Version verweist:

```
aws lambda update-alias \
  --function-name myDurableFunction \
  --name prod \
  --function-version 2
```

> **Important**
>
> Laufende Ausführungen verwenden weiterhin die Version, mit der sie begonnen haben. Neue Aufrufe verwenden die aktualisierte Alias-Version.

Funktionsprotokolle anzeigen

Sehen Sie sich die Logs Ihrer dauerhaften Funktion unter Logs an CloudWatch :

```
aws logs tail /aws/lambda/myDurableFunction --follow
```

Filtern Sie die Protokolle nach einer bestimmten Ausführung:

```
aws logs filter-log-events \
  --log-group-name /aws/lambda/myDurableFunction \
  --filter-pattern "exec-abc123"
```

Bereinigen von Ressourcen

Löschen Sie Ihre dauerhafte Funktion und die zugehörigen Ressourcen:

```
# 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
```

Nächste Schritte

Nach der Bereitstellung Ihrer dauerhaften Funktion mit dem AWS CLI:

- Überwachen Sie Ausführungen mit den Befehlen `list-durable-executions` und `get-durable-execution`
- Checkpoint-Operationen in AWS CloudTrail Datenereignissen anzeigen
- Richten Sie CloudWatch Alarme für Ausführungsfehler oder lang andauernde Ausführungen ein
- Automatisieren Sie Bereitstellungen mithilfe von Shell-Skripten oder Pipelines CI/CD

Weitere Informationen zu AWS CLI Befehlen für Lambda finden Sie in der AWS CLI Befehlsreferenz.

Stellen Sie langlebige Lambda-Funktionen mit Infrastructure as Code bereit

Sie können dauerhafte Lambda-Funktionen mithilfe von Infrastructure as Code (IaC) -Tools wie AWS CloudFormation AWS CDK, oder bereitstellen. AWS Serverless Application Model Mit diesen Tools können Sie Ihre Funktion, Ausführungsrolle und Berechtigungen im Code definieren, sodass Bereitstellungen wiederholbar und versionskontrolliert werden können.

Für alle drei Tools müssen Sie:

- Ermöglichen Sie die dauerhafte Ausführung der Funktion

- Erteilen Sie der Ausführungsrolle Checkpoint-Berechtigungen
- Veröffentlichen Sie eine Version oder erstellen Sie einen Alias (dauerhafte Funktionen erfordern qualifizierte ARNs Funktionen)

AWS CloudFormation

Wird verwendet CloudFormation , um Ihre dauerhafte Funktion in einer Vorlage zu definieren. Im folgenden Beispiel wird eine dauerhafte Funktion mit den erforderlichen Berechtigungen erstellt.

```
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
```

Um die Vorlage bereitzustellen

```
aws cloudformation deploy \
  --template-file template.yaml \
  --stack-name my-durable-function-stack \
  --capabilities CAPABILITY_IAM
```

AWS CDK

AWS CDK ermöglicht es Ihnen, die Infrastruktur mithilfe von Programmiersprachen zu definieren. Die folgenden Beispiele zeigen, wie eine dauerhafte Funktion mit TypeScript Python erstellt wird.

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'
        )
```

Um den CDK-Stack bereitzustellen

```
cdk deploy
```

AWS Serverless Application Model

AWS SAM vereinfacht CloudFormation Vorlagen für serverlose Anwendungen. Die folgende Vorlage erstellt eine dauerhafte Funktion mit 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
```

Um die SAM-Vorlage bereitzustellen

```
sam build
sam deploy --guided
```

Gängige Konfigurationsmuster

Unabhängig davon, welches IaC-Tool Sie verwenden, folgen Sie diesen Mustern für dauerhafte Funktionen:

Dauerhafte Ausführung aktivieren

Legen Sie die Eigenschaft `DurableExecution.Enabled` auf `true` fest. Diese Eigenschaft ist nur verfügbar, wenn die Funktion erstellt wird. Sie können die dauerhafte Ausführung für bestehende Funktionen nicht aktivieren.

Erteilen Sie Checkpoint-Berechtigungen

Fügen Sie der Ausführungsrolle `lambda:CheckpointDurableExecutions` und `lambda:GetDurableExecutionState` hinzu. Beschränken Sie diese Berechtigungen auf die spezifische Funktion ARN.

Verwenden Sie qualifiziert ARNs

Erstellen Sie eine Version oder einen Alias für Ihre Funktion. Dauerhafte Funktionen erfordern qualifizierte Funktionen ARNs (mit Version oder Alias) für den Aufruf. Verwenden Sie `AutoPublishAlias` in und AWS SAM oder erstellen Sie explizite Versionen in CloudFormation und AWS CDK.

Abhängigkeiten von Paketen

Nehmen Sie das Durable Execution SDK in Ihr Bereitstellungspaket auf. Installieren Sie Node.js`@aws/durable-execution-sdk-js`. Installieren Sie Python`aws-durable-execution-sdk-python`.

Nächste Schritte

Nach der Bereitstellung Ihrer dauerhaften Funktion:

- Testen Sie Ihre Funktion mit dem qualifizierten ARN (Version oder Alias)
- Überwachen Sie den Ausführungsfortschritt in der Lambda-Konsole auf der Registerkarte Dauerhafte Ausführungen
- Zeigen Sie Checkpoint-Operationen in Datenereignissen an AWS CloudTrail
- Überprüfen Sie die CloudWatch Protokolle auf Funktionsausgabe und Wiedergabeverhalten

Weitere Informationen zur Bereitstellung von Lambda-Funktionen mit IaC-Tools finden Sie unter:

- CloudFormation AWS::Lambda::Function Referenz
- AWS CDK Dokumentation zum Lambda-Modul
- AWS SAM Entwicklerhandbuch

Beispiele und Anwendungsfälle

Mit den dauerhaften Lambda-Funktionen können Sie fehlertolerante, mehrstufige Anwendungen mit dauerhaften Vorgängen wie Steps und Waits erstellen. Mit automatischem Checkpointing und einem Checkpoint-Replay-Modell, bei dem die Ausführung nach einem Ausfall wieder von vorne beginnt, abgeschlossene Checkpoints jedoch überspringt, können Ihre Funktionen nach Ausfällen wiederhergestellt und die Ausführung wieder aufgenommen werden, ohne dass der Fortschritt verloren geht.

Kurzlebige fehlertolerante Prozesse

Verwenden Sie langlebige Funktionen, um zuverlässige Abläufe aufzubauen, die in der Regel innerhalb von Minuten abgeschlossen werden. Diese Prozesse sind zwar kürzer als langwierige Workflows, profitieren aber dennoch von automatischem Checkpoint und Fehlertoleranz in verteilten Systemen. Zuverlässige Funktionen stellen sicher, dass Ihre mehrstufigen Prozesse auch dann erfolgreich abgeschlossen werden, wenn einzelne Serviceanrufe fehlschlagen, ohne dass eine komplexe Fehlerbehandlung oder ein Code für die Statusverwaltung erforderlich sind.

Zu den gängigen Szenarien gehören Hotelbuchungssysteme, Restaurantreservierungsplattformen, Anfragen für Mitfahrgelegenheiten, der Kauf von Veranstaltungstickets und SaaS-Abonnement-Upgrades. Diese Szenarien weisen gemeinsame Merkmale auf: mehrere Serviceanrufe, die zusammen abgeschlossen werden müssen, die Notwendigkeit, bei vorübergehenden Ausfällen automatische Wiederholungsversuche durchzuführen, und die Anforderung, einen konsistenten Status über verteilte Systeme hinweg aufrechtzuerhalten.

Verteilte Transaktionen über Microservices hinweg

Koordinieren Sie Zahlungen, Inventar und Versand für mehrere Dienste mit automatischem Rollback bei Ausfällen. Jeder Servicevorgang ist in einem Schritt zusammengefasst, sodass sichergestellt wird, dass die Transaktion jederzeit wiederhergestellt werden kann, falls ein Service ausfällt.

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}
```

Schlägt ein Schritt fehl, wiederholt die Funktion den Vorgang automatisch vom letzten erfolgreichen Checkpoint aus. Die Inventarreservierung bleibt auch dann bestehen, wenn die Zahlungsabwicklung vorübergehend fehlschlägt. Wenn die Funktion es erneut versucht, überspringt sie den abgeschlossenen Inventarisierungsschritt und fährt direkt mit der Zahlungsabwicklung fort. Dadurch werden doppelte Reservierungen vermieden und ein einheitlicher Status in Ihrem gesamten verteilten System gewährleistet.

Auftragsabwicklung mit mehreren Schritten

Bearbeiten Sie Bestellungen durch Validierung, Zahlungsautorisierung, Inventarzuweisung und Versand mit automatischer Wiederholung und Wiederherstellung. Jeder Schritt wird überprüft, sodass sichergestellt wird, dass die Bestellung auch dann bearbeitet wird, wenn einzelne Schritte fehlschlagen und es erneut versucht wird.

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']
    }
```

Dieses Muster stellt sicher, dass Bestellungen niemals in Zwischenzuständen stecken bleiben. Schlägt die Validierung fehl, wird die Bestellung vor der Zahlungsautorisierung abgelehnt. Wenn die Zahlungsautorisierung fehlschlägt, wird kein Inventar zugewiesen. Jeder Schritt baut auf dem vorherigen auf und bietet automatische Wiederholungen und Wiederherstellungen.

Hinweis

Die bedingte Prüfung `if (!validation.itemsValid)` befindet sich außerhalb eines Schritts und wird während der Wiedergabe erneut ausgeführt. Das ist sicher, weil es deterministisch ist — es liefert immer dasselbe Ergebnis bei demselben Überprüfungsobjekt.

Prozesse mit langer Laufzeit

Verwenden Sie langlebige Funktionen für Prozesse, die sich über Stunden, Tage oder Wochen erstrecken. Wartevorgänge unterbrechen die Ausführung, ohne dass Rechenkosten anfallen, wodurch Prozesse mit langer Laufzeit kostengünstig werden. Während Wartezeiten wird Ihre Funktion nicht mehr ausgeführt und Lambda recycelt die Ausführungsumgebung. Wenn es Zeit ist, fortzufahren, ruft Lambda Ihre Funktion erneut auf und spielt sie ab dem letzten Checkpoint erneut ab.

Dieses Ausführungsmodell macht langlebige Funktionen ideal für Prozesse, die über einen längeren Zeitraum unterbrochen werden müssen, unabhängig davon, ob sie auf menschliche Entscheidungen, externe Systemreaktionen, geplante Verarbeitungsfenster oder zeitbedingte Verzögerungen warten. Sie zahlen nur für aktive Rechenzeit, nicht für Wartezeiten.

Zu den gängigen Szenarien gehören Prozesse zur Genehmigung von Dokumenten, geplante Stapelverarbeitung, mehrtägige Onboarding-Prozesse, Abonnement-Testprozesse und Systeme für verzögerte Benachrichtigungen. Diese Szenarien haben gemeinsame Merkmale: längere Wartezeiten, gemessen in Stunden oder Tagen, die Notwendigkeit, den Ausführungsstatus während dieser Wartezeiten beizubehalten, und kostensensible Anforderungen, bei denen es unerschwinglich ist, für ungenutzte Rechenzeit zu bezahlen.

Human-in-the-loop Zulassungen

Unterbrechen Sie die Ausführung für Dokumentenprüfungen, Genehmigungen oder Entscheidungen und behalten Sie dabei den Ausführungsstatus bei. Die Funktion wartet auf externe Rückrufe, ohne Ressourcen zu verbrauchen, und wird automatisch fortgesetzt, wenn die Genehmigung eingeht.

Dieses Muster ist wichtig für Prozesse, die menschliches Urteilsvermögen oder externe Validierung erfordern. Die Funktion wird am Rückrufpunkt unterbrochen, sodass während des Wartens keine Rechenkosten anfallen. Wenn jemand seine Entscheidung über die API übermittelt, ruft Lambda Ihre Funktion erneut auf und spielt sie vom Checkpoint aus erneut ab, wobei das Genehmigungsergebnis angezeigt wird.

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
    }
```

Wenn der Rückruf eingeht und Ihre Funktion wieder aufgenommen wird, wird er von Anfang an wiedergegeben. Der Schritt „Dokument vorbereiten“ gibt sofort das Ergebnis der Prüfung zurück. Der waitForCallback Vorgang kehrt außerdem sofort mit dem gespeicherten Genehmigungsergebnis zurück, anstatt erneut zu warten. Die Ausführung wird dann mit den Finalisierungs- oder Archivierungsschritten fortgesetzt.

Mehrstufige Daten-Pipelines

Verarbeiten Sie große Datensätze durch Extraktions-, Transformations- und Ladephasen mit Checkpoints zwischen den Phasen. Jede Phase kann Stunden dauern, bis sie abgeschlossen ist, und Checkpoints stellen sicher, dass die Pipeline bei einer Unterbrechung aus jeder Phase wieder aufgenommen werden kann.

Dieses Muster ist ideal für ETL-Workflows, Datenmigrationen oder Batch-Verarbeitungsaufträge, bei denen Sie Daten stufenweise mit Wiederherstellungspunkten zwischen ihnen verarbeiten müssen. Wenn eine Phase fehlschlägt, wird die Pipeline mit der letzten abgeschlossenen Phase fortgesetzt, anstatt von vorne neu zu starten. Sie können Warteoperationen auch verwenden, um zwischen

den Phasen eine Pause einzulegen. Dabei können Sie Ratenbegrenzungen einhalten, warten, bis nachgelagerte Systeme bereit sind, oder die Verarbeitung außerhalb der Spitzenzeiten planen.

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']
    }
```

Jede Phase ist in einem Schritt zusammengefasst, wodurch ein Checkpoint entsteht, über den die Pipeline bei einer Unterbrechung aus jeder Phase wieder aufgenommen werden kann. Bei der 5-minütigen Wartezeit zwischen Extrahieren und Transformieren werden die Geschwindigkeitsbegrenzungen des Quellsystems eingehalten, ohne Rechenressourcen zu verbrauchen, während bei der Wartezeit bis 2 Uhr der teure Ladevorgang außerhalb der Spitzenzeiten geplant wird.

Hinweis

Der `new Date()` Aufruf und die `calculateMsUntilHour()` Funktion befinden sich außerhalb von Schritten und werden während der Wiedergabe erneut ausgeführt. Für zeitbasierte Operationen, die bei allen Wiederholungen konsistent sein müssen, berechnen Sie den Zeitstempel innerhalb eines Schritts oder verwenden Sie ihn nur für Wartezeiten (die mit Checkpoints versehen sind).

Funktionsübergreifende verkettete Aufrufe

Rufen Sie andere Lambda-Funktionen aus einer dauerhaften Funktion heraus auf, indem Sie. `context.invoke()` Die aufrufende Funktion wird angehalten, während auf den Abschluss der aufgerufenen Funktion gewartet wird, wodurch ein Checkpoint erstellt wird, der das Ergebnis beibehält. Wenn die aufrufende Funktion nach Abschluss der aufgerufenen Funktion unterbrochen wird, wird sie mit dem gespeicherten Ergebnis fortgesetzt, ohne die Funktion erneut aufzurufen.

Verwenden Sie dieses Muster, wenn Sie über spezielle Funktionen verfügen, die bestimmte Bereiche (Kundenvalidierung, Zahlungsabwicklung, Bestandsverwaltung) behandeln und diese in einem Workflow koordinieren müssen. Jede Funktion behält ihre eigene Logik bei und kann von mehreren Orchestrator-Funktionen aufgerufen werden, wodurch Codeduplikationen vermieden werden.

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']
    }
```

Jeder Aufruf erstellt einen Checkpoint in der Orchestrator-Funktion. Wenn der Orchestrator nach Abschluss der Kundenvalidierung unterbrochen wird, wird er von diesem Checkpoint aus mit den gespeicherten Kundendaten wieder aufgenommen und der Validierungsaufruf übersprungen.

Dies verhindert doppelte Aufrufe von nachgelagerten Diensten und gewährleistet eine konsistente Ausführung auch bei Unterbrechungen.

Bei den aufgerufenen Funktionen kann es sich entweder um dauerhafte Funktionen oder um Standard-Lambda-Funktionen handeln. Wenn Sie eine dauerhafte Funktion aufrufen, kann sie über einen eigenen mehrstufigen Workflow mit Wartezeiten und Checkpoints verfügen. Der Orchestrator wartet einfach darauf, dass die komplette Durable-Ausführung abgeschlossen ist, und erhält das Endergebnis.

> **Note**
>
> Kontoübergreifende Aufrufe werden nicht unterstützt. Alle aufgerufenen Funktionen müssen sich im selben AWS Konto wie die aufrufende Funktion befinden.

Fortgeschrittene Muster

Verwenden Sie langlebige Funktionen, um komplexe mehrstufige Anwendungen zu erstellen, die mehrere dauerhafte Operationen, parallel Ausführung, Array-Verarbeitung, bedingte Logik und Polling kombinieren. Mit diesen Mustern können Sie anspruchsvolle Anwendungen erstellen, die viele Aufgaben koordinieren und gleichzeitig Fehlertoleranz und automatische Wiederherstellung gewährleisten.

Fortgeschrittene Muster gehen über einfache sequentielle Schritte hinaus. Sie können Operationen gleichzeitig mit diesen Primitiven ausführen`parallel()`, Arrays mit verarbeiten`map()`, auf externe Bedingungen warten und diese Primitive kombinieren`waitForCondition()`, um zuverlässige Anwendungen zu erstellen. Jeder dauerhafte Vorgang erstellt seine eigenen Prüfpunkte, sodass Ihre Anwendung bei einer Unterbrechung jederzeit wiederhergestellt werden kann.

Onboarding-Prozesse für Benutzer

Führen Sie Benutzer durch die Registrierung, E-Mail-Überprüfung, Profileinrichtung und Erstkonfiguration mit Wiederholungsversuchen. In diesem Beispiel werden aufeinanderfolgende Schritte, Rückrufe und bedingte Logik kombiniert, um einen vollständigen Onboarding-Prozess zu erstellen.

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()
    }
```

Der Prozess kombiniert aufeinanderfolgende Schritte mit Checkpoints für die Kontoerstellung und das Senden von E-Mails und wartet dann bis zu 48 Stunden lang auf die E-Mail-Bestätigung, ohne Ressourcen zu verbrauchen. Die bedingte Logik verarbeitet unterschiedliche Pfade, je nachdem, ob die Überprüfung abgeschlossen ist oder ob das Zeitlimit überschritten wird. Aufgaben zur Profileinrichtung werden gleichzeitig mit parallel Vorgängen ausgeführt, um die Gesamtausführungszeit zu reduzieren. Jeder Schritt wird bei vorübergehenden Ausfällen automatisch wiederholt, um sicherzustellen, dass das Onboarding zuverlässig abgeschlossen wird.

Stapelverarbeitung mit Checkpoints

Verarbeiten Sie Millionen von Datensätzen mit automatischer Wiederherstellung vom letzten erfolgreichen Checkpoint nach Ausfällen. Dieses Beispiel zeigt, wie langlebige Funktionen `map()` Operationen mit Chunking und Ratenbegrenzung kombinieren, um umfangreiche Datenverarbeitungen zu bewältigen.

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
```

Datensätze werden in überschaubare Stapel aufgeteilt, um eine Überlastung des Speichers oder nachgelagerte Dienste zu vermeiden. Anschließend werden mehrere Batches gleichzeitig verarbeitet, wobei die Parallelität kontrolliert wird. `maxConcurrency` Jeder Stapel hat seinen eigenen Prüfpunkt, sodass bei Fehlern nur der fehlgeschlagene Stapel wiederholt wird, anstatt alle Datensätze erneut zu verarbeiten. Dieses Muster ist ideal für ETL-Jobs, Datenmigrationen oder Massenvorgänge, bei denen die Verarbeitung Stunden dauern kann.

Nächste Schritte

- Machen Sie sich mit grundlegenden Konzepten vertraut DurableContext, lernen Sie Schritte kennen und warten Sie
- Informieren Sie sich über bewährte Methoden zum Schreiben von deterministischem Code und zur Leistungsoptimierung
- Erfahren Sie, wie Sie langlebige Funktionen lokal und in der Cloud testen

Sicherheit und Berechtigungen für langlebige Lambda-Funktionen

Dauerhafte Lambda-Funktionen erfordern spezielle IAM-Berechtigungen, um Checkpoint-Operationen zu verwalten. Folgen Sie dem Prinzip der geringsten Rechte, indem Sie nur die Berechtigungen gewähren, die Ihre Funktion benötigt.

Berechtigungen für die Ausführungsrolle

Die Ausführungsrolle Ihrer dauerhaften Funktion benötigt Berechtigungen, um Checkpoints zu erstellen und den Ausführungsstatus abzurufen. Die folgende Richtlinie zeigt die mindestens erforderlichen Berechtigungen:

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

```
        {
            "Effect": "Allow",
            "Action": [
                "lambda:CheckpointDurableExecution",
                "lambda:GetDurableExecutionState"
            ],
            "Resource": "arn:aws:lambda:region:account-id:function:function-name:*"
        }
    ]
}
```

Wenn Sie mithilfe der Konsole eine dauerhafte Funktion erstellen, fügt Lambda diese Berechtigungen automatisch der Ausführungsrolle hinzu. Wenn Sie die Funktion mit AWS CLI oder erstellen AWS CloudFormation, fügen Sie diese Berechtigungen Ihrer Ausführungsrolle hinzu.

Prinzip der geringsten Rechte

Beschränken Sie das `Resource` Element auf eine bestimmte Funktion, ARNs anstatt Platzhalter zu verwenden. Dies beschränkt die Ausführungsrolle auf Checkpoint-Operationen nur für die Funktionen, die sie benötigen.

Beispiel: Eingeschränkte Berechtigungen für mehrere Funktionen

```
{
    "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:*"
            ]
        }
    ]
}
```

Alternativ können Sie die AWS verwaltete Richtlinie verwenden, `AWSLambdaBasicDurableExecutionRolePolicy` die die erforderlichen dauerhaften

Ausführungsberechtigungen zusammen mit den grundlegenden Lambda-Ausführungsberechtigungen für Amazon CloudWatch Logs umfasst.

Staatliche Verschlüsselung

Dauerhafte Lambda-Funktionen ermöglichen automatisch und kostenlos die Verschlüsselung im Ruhezustand mithilfe AWS eigener Schlüssel. Jede Funktionsausführung behält einen isolierten Status bei, auf den andere Ausführungen nicht zugreifen können. Vom Kunden verwaltete Schlüssel (CMK) werden nicht unterstützt.

Zu den Checkpoint-Daten gehören:

- Schrittergebnisse und Rückgabewerte
- Ausführungsfortschritt und Zeitplan
- Informationen zum Wartestatus

Alle Daten werden während der Übertragung mit TLS verschlüsselt, wenn Lambda Checkpoint-Daten liest oder schreibt.

Benutzerdefinierte Verschlüsselung mit benutzerdefinierten Serialisierern und Deserialisierern

Für kritische Sicherheitsanforderungen können Sie mithilfe von benutzerdefinierten Serialisierern und Deserialisierern () mithilfe eines dauerhaften SDK Ihren eigenen Verschlüsselungs- und Entschlüsselungsmechanismus implementieren. SerDer Dieser Ansatz gibt Ihnen die volle Kontrolle über die Verschlüsselungsschlüssel und Algorithmen, die zum Schutz von Checkpoint-Daten verwendet werden.

> **Important**
>
> Wenn Sie eine benutzerdefinierte Verschlüsselung verwenden, verlieren Sie die Sichtbarkeit der Betriebsergebnisse in der Lambda-Konsole und in den API-Antworten. Checkpoint-Daten erscheinen im Ausführungsverlauf verschlüsselt und können ohne Entschlüsselung nicht überprüft werden.

Die Anforderungen `kms:Encrypt` und `kms:Decrypt` Berechtigungen der Ausführungsrolle Ihrer Funktion für den AWS KMS Schlüssel, der in der benutzerdefinierten SerDer Implementierung verwendet wird.

CloudTrail Protokollierung

Lambda protokolliert Checkpoint-Operationen als Datenereignisse. AWS CloudTrail Sie können CloudTrail damit überprüfen, wann Checkpoints erstellt werden, Änderungen des Ausführungsstatus verfolgen und den Zugriff auf dauerhafte Ausführungsdaten überwachen.

Checkpoint-Operationen werden in CloudTrail Protokollen mit den folgenden Ereignisnamen angezeigt:

- `CheckpointDurableExecution`- Wird protokolliert, wenn ein Schritt abgeschlossen ist und ein Checkpoint erstellt wird
- `GetDurableExecutionState`- Wird protokolliert, wenn Lambda den Ausführungsstatus während der Wiedergabe abruft

Um die Protokollierung von Datenereignissen für dauerhafte Funktionen zu aktivieren, konfigurieren Sie einen CloudTrail Trail zum Protokollieren von Lambda-Datenereignissen. Weitere Informationen finden Sie im CloudTrail Benutzerhandbuch unter Protokollieren von Datenereignissen.

Beispiel: CloudTrail Protokolleintrag für den Checkpoint-Betrieb

```
{
    "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"
}
```

Kontoübergreifende Überlegungen

Wenn Sie dauerhafte Funktionen AWS kontenübergreifend aufrufen, benötigt das aufrufende Konto eine `lambda:InvokeFunction` Genehmigung, aber Checkpoint-Operationen verwenden immer die Ausführungsrolle im Konto der Funktion. Das aufrufende Konto kann nicht direkt auf Checkpoint-Daten oder den Ausführungsstatus zugreifen.

Diese Isolierung stellt sicher, dass Checkpoint-Daten innerhalb des Kontos der Funktion sicher bleiben, auch wenn sie von externen Konten aus aufgerufen werden.

Vererbte Lambda-Sicherheitsfunktionen

Dauerhafte Funktionen übernehmen alle Sicherheits-, Governance- und Compliance-Funktionen von Lambda, einschließlich VPC-Konnektivität, Verschlüsselung mit Umgebungsvariablen, Warteschlangen mit unerlaubten Buchstaben, reservierte Parallelität, Funktions- URLs, Codesignatur- und Compliance-Zertifizierungen (SOC, PCI DSS, HIPAA usw.).

Ausführliche Informationen zu den Sicherheitsfunktionen von Lambda finden Sie unter Sicherheit AWS Lambda im Lambda Developer Guide. Die einzigen zusätzlichen Sicherheitsaspekte für dauerhafte Funktionen sind die in diesem Handbuch dokumentierten Checkpoint-Berechtigungen.

SDK für langlebige Ausführung

Das Durable Execution SDK ist die Grundlage für die Entwicklung langlebiger Funktionen. Es bietet die Grundelemente, die Sie benötigen, um den Fortschritt zu überprüfen, Wiederholungsversuche zu verarbeiten und den Ausführungsablauf zu verwalten. Das SDK abstrahiert die Komplexität der Verwaltung und Wiedergabe von Checkpoints und ermöglicht es Ihnen, sequentiellen Code zu schreiben, der automatisch fehlertolerant wird.

Das SDK ist für JavaScript TypeScript, und Python verfügbar. Eine vollständige API-Dokumentation und Beispiele finden Sie im JavaScript/TypeScript SDK und im Python-SDK unter GitHub.

DurableContext

Das SDK stellt Ihrer Funktion ein `DurableContext` Objekt zur Verfügung, das alle dauerhaften Operationen verfügbar macht. Dieser Kontext ersetzt den Standard-Lambda-Kontext und bietet Methoden für die Erstellung von Checkpoints, die Verwaltung des Ausführungsflusses und die Koordination mit externen Systemen.

Um das SDK zu verwenden, umschließen Sie Ihren Lambda-Handler mit dem Durable Execution Wrapper:

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
```

Der Wrapper fängt Ihren Funktionsaufruf ab, lädt alle vorhandenen Checkpoint-Logs und stellt das zur Verfügung, das Replay und Checkpointing verwaltet. `DurableContext`

Was macht das SDK

Das SDK erfüllt drei wichtige Aufgaben, die eine dauerhafte Ausführung ermöglichen:

Checkpoint-Verwaltung: Das SDK erstellt automatisch Checkpoints, wenn Ihre Funktion dauerhafte Operationen ausführt. Jeder Checkpoint zeichnet den Operationstyp, die Eingaben und die Ergebnisse auf. Wenn Ihre Funktion einen Schritt abgeschlossen hat, behält das SDK den Checkpoint bei, bevor der Vorgang fortgesetzt wird. Dadurch wird sichergestellt, dass Ihre Funktion

nach jeder abgeschlossenen Operation wieder aufgenommen werden kann, falls sie unterbrochen wird.

Koordination der Wiedergabe: Wenn Ihre Funktion nach einer Pause oder Unterbrechung wieder aufgenommen wird, führt das SDK eine Wiederholung durch. Es führt Ihren Code von Anfang an aus, überspringt jedoch abgeschlossene Operationen und verwendet gespeicherte Checkpoint-Ergebnisse, anstatt sie erneut auszuführen. Das SDK stellt sicher, dass die Wiedergabe deterministisch ist. Bei denselben Eingaben und demselben Checkpoint-Log erzeugt Ihre Funktion dieselben Ergebnisse.

Zustandsisolierung: Das SDK behält den Ausführungsstatus getrennt von Ihrer Geschäftslogik bei. Jede dauerhafte Ausführung hat ihr eigenes Checkpoint-Protokoll, auf das andere Ausführungen nicht zugreifen können. Das SDK verschlüsselt Checkpoint-Daten im Ruhezustand und stellt sicher, dass der Status bei allen Wiederholungen konsistent bleibt.

Wie funktioniert Checkpointing

Wenn Sie einen dauerhaften Vorgang aufrufen, folgt das SDK dieser Reihenfolge:

1. Nach einem vorhandenen Checkpoint suchen: Das SDK prüft, ob dieser Vorgang bereits bei einem früheren Aufruf abgeschlossen wurde. Wenn ein Checkpoint vorhanden ist, gibt das SDK das gespeicherte Ergebnis zurück, ohne den Vorgang erneut auszuführen.
2. Führen Sie den Vorgang aus: Wenn kein Checkpoint vorhanden ist, führt das SDK Ihren Operationscode aus. Für Schritte bedeutet das, dass Sie Ihre Funktion aufrufen. Für Wartezeiten bedeutet dies, die Wiederaufnahme zu planen.
3. Checkpoint erstellen: Nach Abschluss des Vorgangs serialisiert das SDK das Ergebnis und erstellt einen Checkpoint. Der Checkpoint umfasst den Operationstyp, den Namen, die Eingaben, das Ergebnis und den Zeitstempel.
4. Checkpoint beibehalten: Das SDK ruft die Lambda-Checkpoint-API auf, um den Checkpoint beizubehalten. Dadurch wird sichergestellt, dass der Checkpoint dauerhaft ist, bevor die Ausführung fortgesetzt wird.
5. Ergebnis zurückgeben: Das SDK gibt das Vorgangsergebnis an Ihren Code zurück, der mit der nächsten Operation fortgeführt wird.

Diese Reihenfolge stellt sicher, dass das Ergebnis eines Vorgangs sicher gespeichert wird, sobald er abgeschlossen ist. Wenn Ihre Funktion zu irgendeinem Zeitpunkt unterbrochen wird, kann das SDK die Wiedergabe bis zum letzten abgeschlossenen Checkpoint wiederholen.

Verhalten bei der Wiedergabe

Wenn Ihre Funktion nach einer Pause oder Unterbrechung wieder aufgenommen wird, führt das SDK eine Wiederholung durch:

1. Checkpoint-Log laden: Das SDK ruft das Checkpoint-Log für diese Ausführung von Lambda ab.
2. Von Anfang an ausführen: Das SDK ruft Ihre Handler-Funktion von Anfang an auf, nicht von der Stelle aus, an der sie angehalten wurde.
3. Abgeschlossene dauerhafte Operationen überspringen: Da Ihr Code dauerhafte Operationen aufruft, überprüft das SDK jeden Vorgang anhand des Checkpoint-Logs. Bei abgeschlossenen dauerhaften Vorgängen gibt das SDK das gespeicherte Ergebnis zurück, ohne den Operationscode auszuführen.

   > **Note**
   >
   > Wenn das Ergebnis eines untergeordneten Kontextes größer als die maximale Checkpoint-Größe (256 KB) war, wird der Code des Kontextes während der Wiedergabe erneut ausgeführt. Auf diese Weise können Sie aus den dauerhaften Vorgängen, die innerhalb des Kontextes ausgeführt wurden, umfangreiche Ergebnisse erstellen, die dann im Checkpoint-Log nachgeschlagen werden. Daher ist es unerlässlich, deterministischen Code nur im Kontext selbst auszuführen. Bei der Verwendung untergeordneter Kontexte mit großen Ergebnissen ist es eine bewährte Methode, lang andauernde oder nicht deterministische Arbeiten innerhalb von Schritten auszuführen und nur Aufgaben mit kurzer Laufzeit auszuführen, die die Ergebnisse im Kontext selbst kombinieren.

4. Am Unterbrechungspunkt fortsetzen: Wenn das SDK einen Vorgang ohne Checkpoint erreicht, wird es normal ausgeführt und erstellt neue Checkpoints, sobald dauerhafte Operationen abgeschlossen sind.

Für diesen Wiedergabemechanismus muss Ihr Code deterministisch sein. Bei denselben Eingaben und demselben Checkpoint-Log muss Ihre Funktion dieselbe Reihenfolge von dauerhaften Operationsaufrufen ausführen. Das SDK erzwingt dies, indem es bei der Wiedergabe überprüft, ob die Namen und Typen der Operationen mit dem Checkpoint-Protokoll übereinstimmen.

Verfügbare dauerhafte Operationen

Das `DurableContext` bietet Operationen für verschiedene Koordinationsmuster. Bei jeder dauerhaften Operation werden automatisch Checkpoints erstellt, sodass Ihre Funktion an jedem beliebigen Punkt wieder aufgenommen werden kann.

Schritte

Führt Geschäftslogik mit automatischem Checkpoint und Wiederholungsversuch aus. Verwenden Sie Schritte für Operationen, die externe Dienste aufrufen, Berechnungen durchführen oder Logik ausführen, für die ein Checkpoint erforderlich ist. Das SDK erstellt vor und nach dem Schritt einen Checkpoint und speichert das Ergebnis zur Wiederholung.

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'
)
```

Die Schritte unterstützen konfigurierbare Wiederholungsstrategien, Ausführungssemantik (at-most-once oder at-least-once) und benutzerdefinierte Serialisierung.

Waits (Warteereignis)

Hält die Ausführung für eine angegebene Dauer an, ohne Rechenressourcen zu verbrauchen. Das SDK erstellt einen Checkpoint, beendet den Funktionsaufruf und plant die Wiederaufnahme. Wenn die Wartezeit abgeschlossen ist, ruft Lambda Ihre Funktion erneut auf und das SDK wiederholt die Wiedergabe bis zum Wartepunkt, bevor es fortfährt.

TypeScript

```
// Wait 1 hour without charges
await context.wait({ seconds: 3600 });
```

Python

```
# Wait 1 hour without charges
context.wait(3600)
```

Rückrufe

Callbacks ermöglichen es Ihrer Funktion, eine Pause einzulegen und darauf zu warten, dass externe Systeme Eingaben bereitstellen. Wenn Sie einen Callback erstellen, generiert das SDK eine eindeutige Callback-ID und erstellt einen Checkpoint. Ihre Funktion wird dann unterbrochen (beendet den Aufruf), ohne dass Rechengebühren anfallen. Externe Systeme senden Rückrufergebnisse mit dem `SendDurableExecutionCallbackSuccess` oder `SendDurableExecutionCallbackFailure` APIs Lambda. Wenn ein Callback gesendet wird, ruft Lambda Ihre Funktion erneut auf, das SDK wiederholt die Wiedergabe bis zum Callback-Point und Ihre Funktion fährt mit dem Callback-Ergebnis fort.

Das SDK bietet zwei Methoden für die Arbeit mit Callbacks:

createCallback: Erstellt einen Callback und gibt sowohl eine Promise- als auch eine Callback-ID zurück. Sie senden die Callback-ID an ein externes System, das das Ergebnis mithilfe der Lambda-API übermittelt.

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: Vereinfacht die Bearbeitung von Rückrufen, indem die Erstellung und Übermittlung von Rückrufen in einem Vorgang kombiniert werden. Das SDK erstellt den Callback, führt Ihre Absenderfunktion mit der Callback-ID aus und wartet auf das Ergebnis.

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)
```

```
)
```

Konfigurieren Sie Timeouts, um zu verhindern, dass Funktionen auf unbestimmte Zeit warten. Wenn bei einem Callback eine Zeitüberschreitung auftritt, löst das SDK eine aus `CallbackError` und Ihre Funktion kann den Timeout-Fall behandeln. Verwenden Sie Heartbeat-Timeouts für lang andauernde Rückrufe, um zu erkennen, wann externe Systeme nicht mehr reagieren.

Verwenden Sie Callbacks für human-in-the-loop Workflows, die Integration externer Systeme, Webhook-Antworten oder jedes Szenario, in dem die Ausführung für externe Eingaben unterbrochen werden muss.

Parallele Ausführung

Führt mehrere Operationen gleichzeitig mit optionaler Parallelitätssteuerung aus. Das SDK verwaltet die parallel Ausführung, erstellt Prüfpunkte für jeden Vorgang und behandelt Fehler gemäß Ihrer Abschlussrichtlinie.

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')
)
```

Wird verwendet`parallel`, um unabhängige Operationen gleichzeitig auszuführen.

Zuordnung

Führt gleichzeitig einen Vorgang für jedes Element in einem Array mit optionaler Parallelitätssteuerung aus. Das SDK verwaltet die gleichzeitige Ausführung, erstellt Prüfpunkte für jeden Vorgang und behandelt Fehler gemäß Ihrer Abschlussrichtlinie.

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'
    )
)
```

Wird verwendet`map`, um Arrays mit Parallelitätssteuerung zu verarbeiten.

Kontexte für Kinder

Erzeugt einen isolierten Ausführungskontext für Gruppierungsvorgänge. Untergeordnete Kontexte haben ihr eigenes Checkpoint-Log und können mehrere Schritte, Wartezeiten und andere Operationen enthalten. Das SDK behandelt den gesamten untergeordneten Kontext für Wiederholungen und Wiederherstellungen als eine Einheit.

Verwenden Sie untergeordnete Kontexte, um komplexe Workflows zu organisieren, Unterworkflows zu implementieren oder Vorgänge zu isolieren, die zusammen wiederholt werden sollten.

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'
)
```

Der Wiederholungsmechanismus erfordert, dass dauerhafte Operationen in einer deterministischen Reihenfolge ablaufen. Wenn Sie mehrere untergeordnete Kontexte verwenden, können Sie mehrere Arbeitsabläufe gleichzeitig ausführen lassen, und der Determinismus gilt für jeden Kontext separat. Auf diese Weise können Sie Hochleistungsfunktionen erstellen, die mehrere CPU-Kerne effizient nutzen.

Stellen Sie sich zum Beispiel vor, wir starten zwei untergeordnete Kontexte, A und B. Beim ersten Aufruf wurden die Schritte innerhalb der Kontexte in dieser Reihenfolge ausgeführt, wobei die A-Schritte gleichzeitig mit den B-Schritten ausgeführt wurden: A1, B1, B2, A2, A3. Bei der Wiedergabe ist das Timing viel schneller, da die Ergebnisse aus dem Checkpoint-Protokoll abgerufen werden und die Schritte zufällig in einer anderen Reihenfolge ausgeführt werden: B1, A1, A2, B2, A3. Da die A-Schritte in der richtigen Reihenfolge (A1, A2, A3) und die B-Schritte in der richtigen Reihenfolge (B1, B2) angetroffen wurden, wurde die Notwendigkeit des Determinismus korrekt erfüllt.

Bedingte Wartezeiten

Führt Abfragen nach einer Bedingung durch, bei der zwischen den Versuchen automatisch ein Checkpoint angezeigt wird. Das SDK führt Ihre Prüffunktion aus, erstellt einen Checkpoint mit dem Ergebnis, wartet entsprechend Ihrer Strategie und wiederholt den Vorgang, bis die Bedingung erfüllt ist.

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}
    )
)
```

Wird verwendet, `waitForCondition` um externe Systeme abzufragen, darauf zu warten, dass Ressourcen bereit sind, oder um Wiederholungsversuche mit Backoff zu implementieren.

Funktionsaufruf

Ruft eine weitere Lambda-Funktion auf und wartet auf ihr Ergebnis. Das SDK erstellt einen Checkpoint, ruft die Zielfunktion auf und nimmt Ihre Funktion wieder auf, wenn der Aufruf abgeschlossen ist. Dies ermöglicht die Funktionskomposition und die Workflow-Zerlegung.

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'
)
```

Wie lange Betriebsabläufe gemessen werden

Jeder dauerhafte Vorgang, über den Sie aufrufen, `DurableContext` erstellt Kontrollpunkte, um den Ausführungsfortschritt zu verfolgen und Statusdaten zu speichern. Für diese Operationen fallen je nach Nutzung Gebühren an, und die Checkpoints können Daten enthalten, die zu Ihren Schreib- und Aufbewahrungskosten beitragen. Zu den gespeicherten Daten gehören Aufrufereignisse, Payloads, die von Schritten zurückgegeben wurden, und Daten, die beim Ausführen von Rückrufen weitergegeben werden. Wenn Sie wissen, wie langlebige Operationen gemessen werden, können Sie die Ausführungskosten abschätzen und Ihre Arbeitsabläufe optimieren. Einzelheiten zur Preisgestaltung finden Sie auf der Seite mit den Lambda-Preisen.

Die Größe der Nutzlast bezieht sich auf die Größe der serialisierten Daten, die bei einem dauerhaften Betrieb fortgeführt werden. Die Daten werden in Byte gemessen und die Größe kann je nach dem für den Vorgang verwendeten Serializer variieren. Die Nutzlast eines Vorgangs kann das Ergebnis selbst sein, wenn der Vorgang erfolgreich abgeschlossen wurde, oder das serialisierte Fehlerobjekt, falls der Vorgang fehlschlägt.

Grundlegende Operationen

Grundoperationen sind die grundlegenden Bausteine für dauerhafte Funktionen:

Operation	Zeitlicher Ablauf der Checkpoints	Anzahl der Operationen	Die Daten wurden beibehalten
Ausführung	Gestartet	1	Größe der eingegebenen Nutzlast
Ausführung	Abgeschlossen () Succeeded/Failed/Stopped	0	Größe der Ausgangsnutzlast
Schritt	Retry/Succeeded/Failed	1 +1 N Wiederholungen	Bei jedem Versuch wurde die Nutzlastgröße zurückgegeben
Wait	Gestartet	1	–
WaitForCondition	Jeder Abfrageversuch	1 + N Umfragen	Bei jedem Abfrageversuch wurde die Größe der Nutzlast zurückgegeben
Wiederholter Versuch auf Aufrufebene	Gestartet	1	Nutzlast für das Fehlerobjekt

Callback-Operationen

Callback-Operationen ermöglichen es Ihrer Funktion, eine Pause einzulegen und darauf zu warten, dass externe Systeme Eingaben bereitstellen. Diese Operationen erstellen Checkpoints, wenn der Callback erstellt und abgeschlossen ist:

Operation	Zeitpunkt der Checkpoints	Anzahl der Operationen	Die Daten wurden beibehalten
CreateCallback	Gestartet	1	–

Operation	Zeitpunkt der Checkpoints	Anzahl der Operationen	Die Daten wurden beibehalten
Abschluss des Rückrufs per API-Aufruf	Completed	0	Payload des Rückrufs
WaitForCallback	Gestartet	3 + N Wiederholungen (Kontext + Rückruf + Schritt)	Payloads, die durch Versuche des Absenders zurückgesendet wurden, plus zwei Kopien der Payload für den Rückruf

Zusammengesetzte Operationen

Zusammengesetzte Operationen kombinieren mehrere dauerhafte Operationen, um komplexe Koordinationsmuster wie parallel Ausführung, Array-Verarbeitung und verschachtelte Kontexte zu handhaben:

Operation	Zeitplan für Checkpoints	Anzahl der Operationen	Die Daten wurden beibehalten
Parallel	Gestartet	1 + N Zweige (1 übergeordneter Kontext + N untergeor	Bis zu zwei Kopien der zurückgegebenen Nutzdatengröße aus jedem Zweig plus den Status jedes Zweigs

Operation	Zeitplan für Checkpoints	Anzahl der Operationen	Die Daten wurden beibehalten
		dnete Kontexte)	
Zuordnung	Gestartet	1 + N Zweige (1 übergeordneter Kontext + N untergeordnete Kontexte)	Bis zu zwei Kopien der zurückgegebenen Payload-Größe aus jeder Iteration plus der Status jeder Iteration
Versprich Helfer	Completed	1	Die Nutzlastgröße wurde aus dem Versprechen zurückgegeben
RunInChildContext	Erfolgreich/Fehlgeschlagen	1	Die Nutzdatengröße wurde aus dem untergeordneten Kontext zurückgegeben

Bei Kontexten, die z. B. aus zusammengesetzten Operationen stammen `runInChildContext` oder intern von zusammengesetzten Operationen verwendet werden, werden Ergebnisse, die kleiner als 256 KB sind, direkt überprüft. Größere Ergebnisse werden nicht gespeichert, sondern während der Wiedergabe rekonstruiert, indem die Operationen des Kontextes erneut verarbeitet werden.

Unterstützte Laufzeiten für langlebige Funktionen

Dauerhafte Funktionen sind für Node.js- und Python-Laufzeiten verfügbar. Sie können dauerhafte Funktionen mithilfe verwalteter Laufzeiten in der Lambda-Konsole erstellen oder sie mithilfe von Container-Images bereitstellen, um zusätzliche Flexibilität bei der Laufzeitversion zu erzielen.

Von Lambda verwaltete Laufzeiten

Die folgenden verwalteten Laufzeiten unterstützen dauerhafte Funktionen, wenn Sie Funktionen in der Lambda-Konsole erstellen oder AWS CLI mit dem `--durable-config '{"ExecutionTimeout": 10, "RetentionPeriodInDays":1}'` Parameter verwenden. Vollständige Informationen zu Lambda-Laufzeiten finden Sie unter Lambda-Laufzeiten.

Sprache	Laufzeit
Node.js	nodejs2.x
Node.js	nodejs24.x
Python	Python 3.13
Python	Python 3,14

> **Note**
>
> Zu den Lambda-Laufzeiten gehört das Durable Execution SDK für Tests und Entwicklung. Wir empfehlen jedoch, das SDK in Ihr Bereitstellungspaket für die Produktion aufzunehmen. Dadurch wird die Versionskonsistenz gewährleistet und potenzielle Laufzeitaktualisierungen vermieden, die Ihr Funktionsverhalten beeinträchtigen könnten.

Node.js

Installieren Sie das SDK in Ihrem Projekt Node.js:

```
npm install @aws/durable-execution-sdk-js
```

Das SDK unterstützt JavaScript und TypeScript. Für TypeScript Projekte enthält das SDK Typdefinitionen.

Python

Installieren Sie das SDK in Ihrem Python-Projekt:

```
pip install aws-durable-execution-sdk-python
```

Das Python-SDK verwendet synchrone Methoden und benötigt `async/await` keine.

Container-Images

Sie können dauerhafte Funktionen mit Container-Images verwenden, um zusätzliche Laufzeitversionen oder benutzerdefinierte Laufzeitkonfigurationen zu unterstützen. Mit Container-Images können Sie Laufzeitversionen verwenden, die nicht als verwaltete Laufzeiten verfügbar sind, oder Ihre Laufzeitumgebung anpassen.

So erstellen Sie eine dauerhafte Funktion mithilfe eines Container-Images:

1. Erstellen Sie ein Dockerfile basierend auf einem Lambda-Basisimage
2. Installieren Sie das Durable Execution SDK in Ihrem Container
3. Erstellen Sie das Container-Image und übertragen Sie es an Amazon Elastic Container Registry
4. Erstellen Sie die Lambda-Funktion aus dem Container-Image mit aktivierter dauerhafter Ausführung

Beispiel für einen Python-Container

Erstellen Sie ein Dockerfile für 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" ]
```

Erstelle eine Datei: `requirements.txt`

```
aws-durable-execution-sdk-python
```

Erstellen Sie das Image und übertragen Sie es:

```
# 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
```

Erstellen Sie die Funktion mit aktivierter dauerhafter Ausführung:

```
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}'
```

Weitere Informationen zur Verwendung von Container-Images mit Lambda finden Sie unter Creating Lambda container images im Lambda Developer Guide.

Überlegungen zur Laufzeit

SDK-Versionsverwaltung: Nehmen Sie das Durable Execution SDK in Ihr Bereitstellungspaket oder Container-Image auf. Dadurch wird sichergestellt, dass Ihre Funktion eine bestimmte SDK-Version verwendet und nicht durch Runtime-Updates beeinträchtigt wird. Stecken Sie SDK-Versionen in Ihren PC `package.jsonrequirements.txt`, um zu kontrollieren, wann Sie ein Upgrade durchführen.

AWS Runtime-Updates: aktualisiert verwaltete Laufzeiten um Sicherheitspatches und Bugfixes. Diese Updates können neue SDK-Versionen enthalten. Um unerwartetes Verhalten zu vermeiden, nehmen Sie das SDK in Ihr Bereitstellungspaket auf und testen Sie es gründlich, bevor Sie es in der Produktion einsetzen.

Größe des Container-Images: Container-Images haben eine maximale unkomprimierte Größe von 10 GB. Das Durable Execution SDK fügt Ihrem Bild eine minimale Größe hinzu. Optimieren Sie Ihren Container, indem Sie mehrstufige Builds verwenden und unnötige Abhängigkeiten entfernen.

Kaltstartleistung: Container-Images haben möglicherweise längere Kaltstartzeiten als verwaltete Laufzeiten. Das langlebige Ausführungs-SDK hat nur minimale Auswirkungen auf die Kaltstartleistung. Verwenden Sie die bereitgestellte Parallelität, wenn die Kaltstart-Latenz für Ihre Anwendung entscheidend ist.

Dauerhafte Lambda-Funktionen aufrufen

Dauerhafte Lambda-Funktionen unterstützen dieselben Aufrufmethoden wie Standard-Lambda-Funktionen. Sie können langlebige Funktionen synchron, asynchron oder über Zuordnungen von Ereignisquellen aufrufen. Der Aufrufvorgang ist identisch mit Standardfunktionen, langlebige Funktionen bieten jedoch zusätzliche Funktionen für Ausführungen mit langer Laufzeit und automatische Statusverwaltung.

Methoden zum Aufrufen

Synchroner Aufruf: Rufen Sie eine dauerhafte Funktion auf und warten Sie auf die Antwort. Synchrone Aufrufe werden vom Lambda auf 15 Minuten (oder weniger, abhängig von der konfigurierten Funktion und dem Ausführungstimeout) begrenzt. Verwenden Sie synchrone Aufrufe, wenn Sie sofortige Ergebnisse benötigen oder wenn Sie Dienste integrieren, die eine Antwort erwarten. APIs Sie können Wait-Operationen für effiziente Berechnungen verwenden, ohne den Aufrufer zu unterbrechen — der Aufruf wartet, bis die gesamte Durable-Ausführung abgeschlossen ist. Verwenden Sie für idempotente Ausführungsstarts den Parameter für den Ausführungsnamen, wie unter Idempotenz beschrieben.

```
aws lambda invoke \
  --function-name my-durable-function:1 \
  --cli-binary-format raw-in-base64-out \
  --payload '{"orderId": "12345"}' \
  response.json
```

Asynchroner Aufruf: Ein Ereignis zur Verarbeitung in eine Warteschlange stellen, ohne auf eine Antwort zu warten. Lambda stellt das Ereignis in eine Warteschlange und kehrt sofort zurück. Asynchrone Aufrufe unterstützen Ausführungsdauern von bis zu einem Jahr. Verwenden Sie den asynchronen Aufruf für fire-and-forget Szenarien oder wenn die Verarbeitung im Hintergrund erfolgen kann. Verwenden Sie für idempotente Ausführungsstarts den Parameter für den Ausführungsnamen, wie unter Idempotenz beschrieben.

```
aws lambda invoke \
  --function-name my-durable-function:1 \
  --invocation-type Event \
  --cli-binary-format raw-in-base64-out \
  --payload '{"orderId": "12345"}' \
  response.json
```

Zuordnungen von Ereignisquellen: Konfigurieren Sie Lambda so, dass Ihre dauerhafte Funktion automatisch aufgerufen wird, wenn Datensätze von stream- oder warteschlangenbasierten Diensten wie Amazon SQS, Kinesis oder DynamoDB verfügbar sind. Zuordnungen von Ereignisquellen fragen die Ereignisquelle ab und rufen Ihre Funktion mit Datensatzstapeln auf. Einzelheiten zur Verwendung von Ereignisquellenzuordnungen mit dauerhaften Funktionen, einschließlich Beschränkungen für die Ausführungsdauer, finden Sie unter Zuordnungen von Ereignisquellen mit dauerhaften Funktionen.

Vollständige Informationen zu den einzelnen Aufrufmethoden finden Sie unter synchroner Aufruf und asynchroner Aufruf.

> **Note**
>
> Dauerhafte Funktionen unterstützen Dead-Letter-Queues (DLQs) für die Fehlerbehandlung, unterstützen aber keine Lambda-Ziele. Konfigurieren Sie eine DLQ, um Datensätze von fehlgeschlagenen Aufrufen zu erfassen.

Qualifizierte Anforderung ARNs

Dauerhafte Funktionen benötigen qualifizierte Bezeichner für den Aufruf. Sie müssen dauerhafte Funktionen mit einer Versionsnummer, einem Alias oder aufrufen. `$LATEST` Sie können entweder

einen vollqualifizierten ARN oder einen Funktionsnamen mit version/alias Suffix verwenden. Sie können keinen unqualifizierten Bezeichner (ohne Versions- oder Aliassuffix) verwenden.

Gültige Aufrufe:

```
# 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
```

Ungültige Aufrufe:

```
# Unqualified ARN (not allowed)
arn:aws:lambda:us-east-1:123456789012:function:my-durable-function

# Unqualified function name (not allowed)
my-durable-function
```

Diese Anforderung stellt sicher, dass langlebige Ausführungen während ihres gesamten Lebenszyklus konsistent bleiben. Wenn eine dauerhafte Ausführung gestartet wird, wird sie an die spezifische Funktionsversion gebunden. Wenn Ihre Funktion Stunden oder Tage später unterbrochen und wieder aufgenommen wird, ruft Lambda dieselbe Version auf, mit der die Ausführung gestartet wurde, wodurch die Codekonsistenz im gesamten Workflow gewährleistet wird.

Bewährte Methode

Verwenden Sie nummerierte Versionen oder Aliase eher für langlebige Produktionsfunktionen als. `$LATEST` Nummerierte Versionen sind unveränderlich und gewährleisten eine deterministische

Wiedergabe. Optional bieten Aliase eine stabile Referenz, die Sie aktualisieren können, um auf neue Versionen zu verweisen, ohne den Aufrufcode zu ändern. Wenn Sie einen Alias aktualisieren, verwenden neue Ausführungen die neue Version, während laufende Ausführungen mit ihrer ursprünglichen Version fortgesetzt werden. Sie können sie `$LATEST` für die Prototypenerstellung oder zur Verkürzung der Bereitstellungszeiten während der Entwicklung verwenden, da Sie wissen, dass Ausführungen möglicherweise nicht korrekt wiedergegeben werden (oder sogar fehlschlagen), wenn sich der zugrunde liegende Code während laufender Ausführungen ändert.

Den Ausführungslebenszyklus verstehen

Wenn Sie eine dauerhafte Funktion aufrufen, erstellt Lambda eine dauerhafte Ausführung, die sich über mehrere Funktionsaufrufe erstrecken kann:

1. Erster Aufruf: Ihre Aufrufanforderung erstellt eine neue dauerhafte Ausführung. Lambda weist eine eindeutige Ausführungs-ID zu und beginnt mit der Verarbeitung.
2. Ausführung und Checkpointing: Während Ihre Funktion dauerhafte Operationen ausführt, erstellt das SDK Checkpoints, die den Fortschritt verfolgen.
3. Aussetzung (falls erforderlich): Wenn Ihre Funktion dauerhafte Wartezeiten wie `wait` oder oder oder `waitForCallback` automatische Schrittwiederholungen verwendet, unterbricht Lambda die Ausführung und beendet die Berechnung der Rechenzeit.
4. Wiederaufnahme: Wenn es Zeit ist, fortzufahren (auch nach erneuten Versuchen), ruft Lambda Ihre Funktion erneut auf. Das SDK spielt das Checkpoint-Protokoll erneut ab und setzt dort fort, wo die Ausführung unterbrochen wurde.
5. Abschluss: Wenn Ihre Funktion ein Endergebnis zurückgibt oder einen unbehandelten Fehler ausgibt, wird die dauerhafte Ausführung abgeschlossen.

Bei synchronen Aufrufen wartet der Aufrufer, bis die gesamte dauerhafte Ausführung abgeschlossen ist, einschließlich aller Warteoperationen. Wenn die Ausführung das Aufruf-Zeitlimit (15 Minuten oder weniger) überschreitet, wird das Timeout für den Aufruf überschritten. Bei asynchronen Aufrufen kehrt Lambda sofort zurück und die Ausführung wird unabhängig fortgesetzt. Verwenden Sie die dauerhafte Ausführung, um den Ausführungsstatus APIs zu verfolgen und die Endergebnisse abzurufen.

Aus dem Anwendungscode aufrufen

Verwenden Sie die AWS SDKs , um dauerhafte Funktionen aus Ihrem Anwendungscode aufzurufen. Der Aufrufvorgang ist identisch mit den Standardfunktionen:

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'})
)
```

Verkettete Aufrufe

Dauerhafte Funktionen können mithilfe der Operation von andere dauerhafte und nicht dauerhafte Funktionen aufrufen. `invoke` `DurableContext` Dadurch entsteht ein verketteter Aufruf, bei dem die aufrufende Funktion wartet (unterbricht), bis die aufgerufene Funktion abgeschlossen ist:

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)}
```

Verkettete Aufrufe erzeugen einen Checkpoint in der aufrufenden Funktion. Wenn die aufrufende Funktion unterbrochen wird, wird sie vom Checkpoint aus mit dem Ergebnis der aufgerufenen Funktion fortgesetzt, ohne die Funktion erneut aufzurufen.

> **Note**
>
> Kontenübergreifende, verkettete Aufrufe werden nicht unterstützt. Die aufgerufene Funktion muss sich in demselben AWS Konto wie die aufrufende Funktion befinden.

Zuordnungen von Ereignisquellen mit dauerhaften Funktionen

Dauerhafte Funktionen funktionieren mit allen Lambda-Ereignisquellenzuordnungen. Konfigurieren Sie Ereignisquellenzuordnungen für dauerhafte Funktionen auf die gleiche Weise, wie Sie sie für Standardfunktionen konfigurieren. Ereignisquellenzuordnungen fragen automatisch Ereignisquellen wie Amazon SQS, Kinesis und DynamoDB Streams ab und rufen Ihre Funktion mit Datensatzstapeln auf.

Zuordnungen von Ereignisquellen sind nützlich für langlebige Funktionen, die Streams oder Warteschlangen mit komplexen, mehrstufigen Workflows verarbeiten. Sie können beispielsweise eine dauerhafte Funktion erstellen, die Amazon SQS SQS-Nachrichten mit Wiederholungen, externen API-Aufrufen und menschlichen Genehmigungen verarbeitet.

Wie Zuordnungen von Ereignisquellen dauerhafte Funktionen aufrufen

Bei Zuordnungen von Ereignisquellen werden dauerhafte Funktionen synchron aufgerufen. Sie warten, bis die vollständige Ausführung abgeschlossen ist, bevor der nächste Stapel verarbeitet oder Datensätze als verarbeitet markiert werden. Wenn die gesamte Dauer der dauerhaften Ausführung 15 Minuten überschreitet, wird das Zeitlimit überschritten und die Ausführung schlägt fehl. Die Ereignisquellenzuordnung empfängt eine Timeout-Ausnahme und behandelt sie entsprechend ihrer Wiederholungskonfiguration.

Ausführungslimit von 15 Minuten

Wenn dauerhafte Funktionen durch Zuordnungen von Ereignisquellen aufgerufen werden, darf die Gesamtdauer der dauerhaften Ausführung 15 Minuten nicht überschreiten. Diese Grenze gilt für die gesamte dauerhafte Ausführung von Anfang bis Ende, nicht nur für einzelne Funktionsaufrufen.

Dieses 15-Minuten-Limit ist unabhängig vom Lambda-Funktions-Timeout (ebenfalls maximal 15 Minuten). Das Funktions-Timeout steuert, wie lange jeder einzelne Aufruf ausgeführt werden kann, während das dauerhafte Ausführungs-Timeout die gesamte verstrichene Zeit vom Start bis zum Abschluss der Ausführung bestimmt.

Beispielszenarien:

- Gültig: Eine dauerhafte Funktion verarbeitet eine Amazon SQS SQS-Nachricht in drei Schritten, die jeweils 2 Minuten dauern, und wartet dann 5 Minuten, bevor ein letzter Schritt abgeschlossen wird. Gesamtausführungszeit: 11 Minuten. Das funktioniert, weil die Gesamtsumme unter 15 Minuten liegt.
- Ungültig: Eine dauerhafte Funktion verarbeitet eine Amazon SQS SQS-Nachricht, schließt die erste Verarbeitung in 2 Minuten ab und wartet dann 20 Minuten auf einen externen Rückruf, bevor der Vorgang abgeschlossen wird. Gesamtausführungszeit: 22 Minuten. Dies überschreitet das 15-Minuten-Limit und schlägt fehl.
- Ungültig: Eine dauerhafte Funktion verarbeitet einen Kinesis-Datensatz mit mehreren Wartevorgängen, die zwischen den Schritten insgesamt 30 Minuten dauern. Obwohl jeder einzelne Aufruf schnell abgeschlossen wird, beträgt die Gesamtausführungszeit mehr als 15 Minuten.

> **Important**
>
> Konfigurieren Sie Ihr dauerhaftes Ausführungstimeout auf 15 Minuten oder weniger, wenn Sie Ereignisquellenzuordnungen verwenden. Andernfalls schlägt die Erstellung der Ereignisquellenzuordnung fehl. Wenn Ihr Workflow längere Ausführungszeiten erfordert, verwenden Sie das unten beschriebene Muster für Zwischenfunktionen.

Zuordnungen von Ereignisquellen konfigurieren

Konfigurieren Sie Ereignisquellenzuordnungen für dauerhafte Funktionen mithilfe der Lambda-Konsole,, AWS CLI oder. AWS SDKs Alle Standardeigenschaften für die Zuordnung von Ereignisquellen gelten für dauerhafte Funktionen:

```
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
```

Denken Sie daran, bei der Konfiguration von Ereignisquellenzuordnungen für dauerhafte Funktionen einen qualifizierten ARN (mit Versionsnummer oder Alias) zu verwenden.

Fehlerbehandlung bei Zuordnungen von Ereignisquellen

Zuordnungen von Ereignisquellen bieten eine integrierte Fehlerbehandlung, die mit dauerhaften Funktionen funktioniert:

- Wiederholungsverhalten: Wenn der erste Aufruf fehlschlägt, versucht die Zuordnung der Ereignisquelle entsprechend ihrer Wiederholungskonfiguration erneut. Konfigurieren Sie die maximale Anzahl an Wiederholungsversuchen und die Wiederholungsintervalle entsprechend Ihren Anforderungen.
- Warteschlangen für unzustellbare Briefe: Konfigurieren Sie eine Warteschlange für unzustellbare Nachrichten, um Datensätze zu erfassen, die nach allen Wiederholungen fehlschlagen. Dies verhindert den Verlust von Nachrichten und ermöglicht die manuelle Überprüfung fehlgeschlagener Datensätze.
- Teilweise Batchfehler: Verwenden Sie für Amazon SQS und Kinesis die Meldung teilweiser Batch-Fehler, um Datensätze einzeln zu verarbeiten und nur fehlgeschlagene Datensätze erneut zu versuchen.
- Bei Fehler halbieren: Aktivieren Sie für Kinesis- und DynamoDB Streams die Option Halbieren bei Fehler, um fehlgeschlagene Batches aufzuteilen und problematische Datensätze zu isolieren.

> **Note**
>
> Dauerhafte Funktionen unterstützen Dead-Letter-Queues (DLQs) für die Fehlerbehandlung, unterstützen aber keine Lambda-Ziele. Konfigurieren Sie eine DLQ, um Datensätze von fehlgeschlagenen Aufrufen zu erfassen.

Vollständige Informationen zur Fehlerbehandlung bei der Zuordnung von Ereignisquellen finden Sie unter Zuordnungen von Ereignisquellen.

Verwendung einer Zwischenfunktion für Workflows mit langer Laufzeit

Wenn Ihr Workflow mehr als 15 Minuten in Anspruch nimmt, verwenden Sie eine Standard-Lambda-Zwischenfunktion zwischen der Zuordnung der Ereignisquelle und Ihrer dauerhaften Funktion. Die Zwischenfunktion empfängt Ereignisse aus der Ereignisquellenzuordnung und ruft die dauerhafte Funktion asynchron auf, wodurch das Ausführungslimit von 15 Minuten aufgehoben wird.

Dieses Muster entkoppelt das synchrone Aufrufmodell der Ereignisquellenzuordnung vom Modell der dauerhaften Funktion mit langer Laufzeit. Bei der Zuordnung der Ereignisquellen wird die Zwischenfunktion aufgerufen, die nach dem Start der dauerhaften Ausführung schnell zurückkehrt. Die dauerhafte Funktion wird dann so lange wie nötig unabhängig ausgeführt (bis zu 1 Jahr).

Architektur

Das Muster der zwischengeschalteten Funktionen besteht aus drei Komponenten:

1. Zuordnung der Ereignisquelle: Ruft die Ereignisquelle ab (Amazon SQS, Kinesis, DynamoDB Streams) und ruft die Zwischenfunktion synchron mit Datensatzstapeln auf.
2. Zwischenfunktion: Eine Standard-Lambda-Funktion, die Ereignisse aus der Ereignisquellenzuordnung empfängt, die Daten bei Bedarf validiert und transformiert und die dauerhafte Funktion asynchron aufruft. Diese Funktion wird schnell abgeschlossen (normalerweise in weniger als 1 Sekunde) und gibt die Steuerung an die Ereignisquellenzuordnung zurück.
3. Dauerhafte Funktion: Verarbeitet das Ereignis mit einer komplexen, mehrstufigen Logik, die über längere Zeiträume ausgeführt werden kann. Wird asynchron aufgerufen, sodass es nicht durch das 15-Minuten-Limit eingeschränkt ist.

Implementierung

Die Zwischenfunktion empfängt das gesamte Ereignis aus der Ereignisquellenzuordnung und ruft die dauerhafte Funktion asynchron auf. Verwenden Sie den Parameter für den Ausführungsnamen, um sicherzustellen, dass die idempotente Ausführung gestartet wird, und verhindern Sie so eine doppelte Verarbeitung, falls die Ereignisquellenzuordnung erneut versucht:

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}
```

Verwenden Sie Powertools für Idempotenz in der Zwischenfunktion selbst, um doppelte Aufrufe der dauerhaften Funktion AWS Lambda zu verhindern, falls die Zuordnung der Ereignisquelle die Zwischenfunktion erneut versucht.

Die dauerhafte Funktion empfängt die Nutzdaten mit dem Namen der Ausführung und verarbeitet alle Datensätze mit lang andauernder Logik:

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())}
```

Wesentliche Überlegungen

Mit diesem Muster wird das Ausführungslimit von 15 Minuten aufgehoben, indem die Zuordnung der Ereignisquelle von der dauerhaften Ausführung entkoppelt wird. Die Zwischenfunktion kehrt unmittelbar nach dem Start der dauerhaften Ausführung zurück, sodass die Verarbeitung der Ereignisquellenzuordnung fortgesetzt werden kann. Die dauerhafte Funktion wird dann so lange wie nötig unabhängig ausgeführt.

Die Zwischenfunktion ist erfolgreich, wenn sie die dauerhafte Funktion aufruft, nicht wenn die dauerhafte Ausführung abgeschlossen ist. Wenn die dauerhafte Ausführung später fehlschlägt, versucht die Zuordnung der Ereignisquelle nicht erneut, da der Stapel bereits erfolgreich verarbeitet wurde. Implementieren Sie die Fehlerbehandlung in der dauerhaften Funktion und konfigurieren Sie Warteschlangen für fehlerhafte Ausführungen.

Verwenden Sie den Parameter für den Ausführungsnamen, um sicherzustellen, dass die idempotente Ausführung gestartet wird. Wenn die Zuordnung der Ereignisquelle erneut versucht, die Zwischenfunktion zu verwenden, startet die dauerhafte Funktion keine doppelte Ausführung, da der Ausführungsname bereits vorhanden ist.

Unterstützte Ereignisquellen

Dauerhafte Funktionen unterstützen alle Lambda-Ereignisquellen, die Ereignisquellenzuordnungen verwenden:

- Amazon SQS SQS-Warteschlangen (Standard und FIFO)
- Kinesis-Streams
- DynamoDB Streams
- Amazon Managed Streaming for Apache Kafka (Amazon MSK)
- Selbstverwaltetes Apache Kafka
- Amazon MQ (ActiveMQ und RabbitMQ)
- Amazon DocumentDB DocumentDB-Change-Streams

Für alle Arten von Ereignisquellen gilt beim Aufrufen dauerhafter Funktionen eine maximale Ausführungsdauer von 15 Minuten.

Wiederholungen für langlebige Lambda-Funktionen

Langlebige Funktionen bieten automatische Wiederholungsfunktionen, die Ihre Anwendungen widerstandsfähig gegen vorübergehende Ausfälle machen. Das SDK verarbeitet Wiederholungen auf zwei Ebenen: schrittweise Wiederholungen bei Ausfällen der Geschäftslogik und Backend-Wiederholungen bei Infrastrukturausfällen.

Schrittweise Wiederholungen

Wenn innerhalb eines Schritts eine nicht abgefangene Ausnahme auftritt, wiederholt das SDK den Schritt automatisch auf der Grundlage der konfigurierten Wiederholungsstrategie. Wiederholungen von Schritten sind Operationen, bei denen das SDK die Ausführung unterbrechen und später fortsetzen kann, ohne dass der Fortschritt verloren geht.

Verhalten bei schrittweisen Wiederholungen

In der folgenden Tabelle wird beschrieben, wie das SDK Ausnahmen in Schritten behandelt:

Szenario	Was passiert	Messung der Auswirkungen
Ausnahme im Gleichschritt mit den verbleibenden Wiederholungsversuchen	Das SDK erstellt einen Checkpoint für den erneuten Versuch und unterbricht die Funktion. Beim nächsten Aufruf wiederholt sich der Schritt mit der konfigurierten Backoff-Verzögerung.	1 Vorgang + Fehler, Payload-Größe
Ausnahme im Schritt ohne verbleibende Wiederholungsversuche	Der Schritt schlägt fehl und löst eine Ausnahme aus. Wenn Ihr Handler-Code diese Ausnahme nicht catch, schlägt die gesamte Ausführung fehl.	1 Vorgang + Fehler, Größe der Nutzlast

Wenn ein Schritt erneut versucht werden muss, überprüft das SDK den Wiederholungsstatus und beendet den Lambda-Aufruf, wenn keine andere Arbeit ausgeführt wird. Auf diese Weise kann das SDK Backoff-Verzögerungen implementieren, ohne Rechenressourcen zu verbrauchen. Die Funktion wird nach Ablauf der Backoff-Periode automatisch wieder aufgenommen.

Konfiguration von Strategien zur schrittweisen Wiederholung

Konfigurieren Sie Wiederholungsstrategien, um zu steuern, wie Schritte mit Fehlern umgehen. Sie können maximale Versuche, Backoff-Intervalle und Bedingungen für Wiederholungsversuche angeben.

Exponentieller Backoff mit maximaler Anzahl von Versuchen:

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)
)
```

Backoff mit festem Intervall:

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)
)
```

Bedingter Wiederholungsversuch (nur bei bestimmten Fehlern wiederholen):

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)
)
```

Wiederholungen deaktivieren:

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}
    )
)
```

Wenn die Wiederholungsstrategie zurückkehrt`shouldRetry: false`, schlägt der Schritt ohne erneute Versuche sofort fehl. Verwenden Sie diese Option für Operationen, die nicht wiederholt werden sollten, z. B. Idempotenzprüfungen oder Operationen mit Nebenwirkungen, die nicht sicher wiederholt werden können.

Ausnahmen außerhalb von Schritten

Wenn in Ihrem Handlercode, aber außerhalb eines Schritts eine nicht abgefangene Ausnahme auftritt, markiert das SDK die Ausführung als fehlgeschlagen. Dadurch wird sichergestellt, dass Fehler in Ihrer Anwendungslogik ordnungsgemäß erfasst und gemeldet werden.

Szenario	Was passiert	Messung der Auswirkungen
Ausnahme im Handler-Code außerhalb eines beliebigen Schritts	Das SDK markiert die Ausführung als FEHLGESCH LAGEN und gibt den Fehler	Fehler bei der Größe der Payload

Szenario	Was passiert	Messung der Auswirkungen
	zurück. Die Ausnahme wird nicht automatisch wiederholt.	

Um die automatische Wiederholung von fehleranfälligem Code zu aktivieren, schließen Sie den Vorgang in einem Schritt mit einer Wiederholungsstrategie ab. Schritte bieten automatische Wiederholungsversuche mit konfigurierbarem Backoff, während Code außerhalb von Schritten sofort fehlschlägt.

Wiederholungen im Backend

Backend-Wiederholungen treten auf, wenn Lambda auf Infrastrukturausfälle oder Laufzeitfehler stößt oder wenn das SDK nicht mit dem Durable Execution Service kommunizieren kann. Lambda versucht diese Fehler automatisch erneut, um sicherzustellen, dass Ihre dauerhaften Funktionen nach vorübergehenden Infrastrukturproblemen wiederhergestellt werden können.

Szenarien für Wiederholungsversuche im Backend

Lambda versucht Ihre Funktion automatisch erneut, wenn die folgenden Szenarien auftreten:

- Interne Dienstfehler — Wenn Lambda oder der Durable Execution Service einen 5xx-Fehler zurückgibt, was auf ein vorübergehendes Serviceproblem hinweist.
- Drosselung — Wenn Ihre Funktion aufgrund von Parallelitätsbeschränkungen oder Servicekontingenten gedrosselt wird.
- Timeouts — Wenn das SDK den Durable Execution Service innerhalb des Timeout-Zeitraums nicht erreichen kann.
- Sandbox-Initialisierungsfehler — Wenn Lambda die Ausführungsumgebung nicht initialisieren kann.
- Laufzeitfehler — Wenn die Lambda-Laufzeit auf Fehler außerhalb Ihres Funktionscodes stößt, z. B. out-of-memory Fehler oder Prozessabstürze.
- Ungültige Checkpoint-Token-Fehler — Wenn das Checkpoint-Token nicht mehr gültig ist, typischerweise aufgrund von Statusänderungen auf der Serviceseite.

In der folgenden Tabelle wird beschrieben, wie das SDK mit diesen Szenarien umgeht:

Szenario	Was passiert	Messung der Auswirkungen
Laufzeitfehler außerhalb des Durable-Handlers (OOM, Timeout, Absturz)	Lambda versucht den Aufruf automatisch erneut. Das SDK wiederholt die Wiedergabe ab dem letzten Checkpoint, wobei abgeschlossene Schritte übersprungen werden.	Fehler: Payload-Größe + 1 Vorgang pro Wiederholung
Dienstfehler (5xx) oder Timeout beim Aufrufen/`CheckpointDurableExecution` `GetDurableExecutionState` APIs	Lambda versucht den Aufruf automatisch erneut. Das SDK wiederholt die Wiedergabe ab dem letzten Checkpoint.	Fehler: Payload-Größe + 1 Vorgang pro Wiederholung
Drosselung (429) oder ungültiges Checkpoint-Token beim Aufrufen von/`CheckpointDurableExecution` `GetDurableExecutionState` APIs	Lambda wiederholt den Aufruf automatisch mit exponentiellem Backoff. Das SDK wiederholt die Wiedergabe ab dem letzten Checkpoint.	Fehler: Payload-Größe + 1 Vorgang pro Wiederholung
Client-Fehler (4xx, außer 429 und ungültigem Token) wenn/`CheckpointDurableExecution` `GetDurableExecutionState` APIs	Das SDK markiert die Ausführung als FEHLGESCHLAGEN. Es erfolgt kein automatischer Wiederholungsversuch, da der Fehler auf ein permanentes Problem hinweist.	Fehler bei der Größe der Payload

Backend-Wiederholungen verwenden exponentielles Backoff und werden fortgesetzt, bis die Funktion erfolgreich ist oder das Ausführungstimeout erreicht ist. Während der Wiedergabe überspringt das

SDK abgeschlossene Prüfpunkte und setzt die Ausführung ab dem letzten erfolgreichen Vorgang fort, um sicherzustellen, dass Ihre Funktion abgeschlossene Arbeit nicht erneut ausführt.

Versuchen Sie es erneut mit bewährten Methoden

Beachten Sie bei der Konfiguration von Wiederholungsstrategien die folgenden bewährten Methoden:

- Konfigurieren Sie explizite Wiederholungsstrategien — Verlassen Sie sich nicht auf das standardmäßige Wiederholungsverhalten in der Produktion. Konfigurieren Sie Strategien für explizite Wiederholungen mit einer für Ihren Anwendungsfall angemessenen maximalen Anzahl von Versuchen und Backoff-Intervallen.
- Verwenden Sie bedingte Wiederholungen — Implementieren Sie `shouldRetry` Logik, um nur vorübergehende Fehler (Ratenbegrenzungen, Timeouts) zu wiederholen und bei dauerhaften Fehlern (Validierungsfehler, nicht gefunden) schnell fehlzuschlagen.
- Stellen Sie eine angemessene maximale Anzahl von Versuchen ein — Balance zwischen Belastbarkeit und Ausführungszeit. Zu viele Wiederholungen können die Fehlererkennung verzögern, während zu wenige Versuche zu unnötigen Fehlern führen können.
- Exponentielles Backoff verwenden — Exponentielles Backoff reduziert die Belastung nachgeschalteter Dienste und erhöht die Wahrscheinlichkeit einer Wiederherstellung nach vorübergehenden Ausfällen.
- Fehleranfälligen Code schrittweise zusammenfassen — Code außerhalb von Schritten kann nicht automatisch wiederholt werden. Integrieren Sie externe API-Aufrufe, Datenbankabfragen und andere fehleranfällige Operationen schrittweise mit Wiederholungsstrategien.
- Überwachen Sie Wiederholungsmetriken — Verfolgen Sie schrittweise Wiederholungsvorgänge und Ausführungsfehler in Amazon CloudWatch , um Muster zu identifizieren und Wiederholungsstrategien zu optimieren.

Idempotenz

Dauerhafte Funktionen bieten eine eingebaute Idempotenz für Ausführungsstarts anhand von Ausführungsnamen. Wenn Sie einen Ausführungsnamen angeben, verwendet Lambda ihn, um doppelte Ausführungen zu verhindern und sichere Wiederholungen von Aufrufanforderungen zu ermöglichen. Schritte haben standardmäßig eine at-least-once Ausführungssemantik — während der Wiedergabe gibt das SDK Checkpoint-Ergebnisse zurück, ohne abgeschlossene Schritte erneut auszuführen, aber Ihre Geschäftslogik muss idempotent sein, um mögliche Wiederholungen vor Abschluss zu verarbeiten.

> **Note**
>
> Lambda Event Source Mappings (ESM) unterstützen Idempotenz beim Start nicht. Daher startet jeder Aufruf (einschließlich Wiederholungen) eine neue dauerhafte Ausführung. Um eine idempotente Ausführung mit Ereignisquellenzuordnungen sicherzustellen, implementieren Sie entweder Idempotenzlogik in Ihrem Funktionscode, z. B. mit Powertools for, AWS Lambda oder verwenden Sie eine reguläre Lambda-Funktion als Proxy (Dispatcher), um eine dauerhafte Funktion mit einem Idempotenzschlüssel (Ausführungsnamenparameter) aufzurufen.

Namen der Ausführung

Sie können einen Ausführungsnamen angeben, wenn Sie eine dauerhafte Funktion aufrufen. Der Ausführungsname fungiert als Idempotenzschlüssel, sodass Sie Aufrufanforderungen sicher wiederholen können, ohne dass doppelte Ausführungen erforderlich sind. Wenn Sie keinen Namen angeben, generiert Lambda automatisch eine eindeutige Ausführungs-ID.

Ausführungsnamen müssen innerhalb Ihres Kontos und Ihrer Region eindeutig sein. Wenn Sie eine Funktion mit einem bereits vorhandenen Ausführungsnamen aufrufen, hängt das Verhalten von Lambda vom Status der vorhandenen Ausführung ab und davon, ob die Nutzlast übereinstimmt.

Verhalten bei Idempotenz

In der folgenden Tabelle wird beschrieben, wie Lambda Aufrufanforderungen verarbeitet, je nachdem, ob Sie einen Ausführungsnamen angeben, welchen Ausführungsstatus es gibt und ob die Nutzlast übereinstimmt:

Szenario	Name angegeben?	Bestehender Ausführungsstatus	Nutzlast identisch?	Behavior
1	Nein	–	–	Neue Ausführung gestartet: Lambda generiert eine eindeutige Ausführungs-ID und startet eine neue Ausführung

Szenario	Name angegeben?	Bestehender Ausführungsstatus	Nutzlast identisch?	Behavior
2	Ja	Hat nie existiert oder die Aufbewahrung ist abgelaufen	–	Neue Ausführung gestartet: Lambda startet eine neue Ausführung mit dem angegebenen Namen
3	Ja	In Ausführung	Ja	Idempotenter Start: Lambda gibt die vorhandenen Ausführungsinformationen zurück, ohne ein Duplikat zu starten. Bei synchronen Aufrufen dient dies als erneutes Anhängen an die laufende Ausführung
4	Ja	In Ausführung	Nein	Fehler: Lambda gibt einen `DurableExecutionAlreadyExists` Fehler zurück, weil eine Ausführung mit diesem Namen bereits mit einer anderen Nutzlast ausgeführt wird

Szenario	Name angegeben?	Bestehender Ausführungsstatus	Nutzlast identisch?	Behavior
5	Ja	Geschlossen (erfolgreich, fehlgeschlagen, gestoppt oder Zeitlimit überschritten)	Ja	Idempotenter Start: Lambda gibt die vorhandenen Ausführungsinformationen zurück, ohne eine neue Ausführung zu starten. Das Ergebnis der geschlossenen Ausführung wird zurückgegeben
6	Ja	Geschlossen (erfolgreich, fehlgeschlagen, gestoppt oder Zeitlimit überschritten)	Nein	Fehler: Lambda gibt einen `DurableExecutionAlreadyExists` Fehler zurück, weil eine Ausführung mit diesem Namen bereits mit einer anderen Nutzlast abgeschlossen wurde

Hinweis

Die Szenarien 3 und 5 zeigen ein idempotentes Verhalten, bei dem Lambda doppelte Aufrufanforderungen sicher verarbeitet, indem es vorhandene Ausführungsinformationen zurückgibt, anstatt Duplikate zu erstellen.

Schritt Idempotenz

Schritte haben standardmäßig eine at-least-once Ausführungssemantik. Wenn Ihre Funktion nach einem Warten, einem Rückruf oder einem Fehler erneut ausgeführt wird, überprüft das SDK jeden

Schritt anhand des Checkpoint-Logs. Für Schritte, die bereits abgeschlossen wurden, gibt das SDK das Checkpoint-Ergebnis zurück, ohne die Schrittlogik erneut auszuführen. Wenn ein Schritt jedoch fehlschlägt oder die Funktion unterbrochen wird, bevor der Schritt abgeschlossen ist, kann der Schritt mehrmals ausgeführt werden.

Ihre in Schritten unterteilte Geschäftslogik muss idempotent sein, um potenzielle Wiederholungsversuche zu bewältigen. Verwenden Sie Idempotenzschlüssel, um sicherzustellen, dass Operationen wie Zahlungen oder Datenbankschreibvorgänge nur einmal ausgeführt werden, auch wenn der Schritt erneut versucht wird.

Beispiel: Schrittweise Verwendung von Idempotenzschlüsseln

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}
```

Sie können Schritte so konfigurieren, dass sie die at-most-once Ausführungssemantik verwenden, indem Sie den Ausführungsmodus auf setzen. `AT_MOST_ONCE_PER_RETRY` Dadurch wird sichergestellt, dass der Schritt pro Wiederholungsversuch höchstens einmal ausgeführt wird, aber möglicherweise überhaupt nicht ausgeführt wird, wenn die Funktion unterbrochen wird, bevor der Schritt abgeschlossen ist.

Das SDK erzwingt eine deterministische Wiedergabe, indem es während der Wiedergabe überprüft, ob die Namen und die Reihenfolge der Schritte mit dem Checkpoint-Protokoll übereinstimmen. Wenn Ihr Code versucht, Schritte in einer anderen Reihenfolge oder mit anderen Namen auszuführen, gibt das SDK eine aus. `NonDeterministicExecutionError`

So funktioniert die Wiedergabe mit abgeschlossenen Schritten:

1. Erster Aufruf: Die Funktion führt Schritt A aus, erstellt einen Checkpoint und wartet dann
2. Zweiter Aufruf (nach dem Warten): Die Funktion wird von Anfang an wiederholt, Schritt A gibt sofort ein Checkpoint-Ergebnis zurück, ohne es erneut auszuführen, und fährt dann mit Schritt B fort

3. Dritter Aufruf (nach erneuter Wartezeit): Die Funktion wird von Anfang an wiederholt, die Schritte A und B geben sofort Checkpoint-Ergebnisse zurück und fährt dann mit Schritt C fort

Dieser Wiederholungsmechanismus stellt sicher, dass abgeschlossene Schritte nicht erneut ausgeführt werden, aber Ihre Geschäftslogik muss immer noch idempotent sein, um Wiederholungen vor Abschluss zu verarbeiten.

Testen langlebiger Lambda-Funktionen

AWS bietet spezielle Tests SDKs für langlebige Funktionen, mit denen Sie Ausführungen sowohl lokal als auch in der Cloud ausführen und überprüfen können. Installieren Sie das Test-SDK für Ihre Sprache:

TypeScript

```
npm install --save-dev @aws/aws-durable-execution-sdk-js-testing
```

Eine vollständige Dokumentation und Beispiele finden Sie im TypeScript Test-SDK unter GitHub.

Python

```
pip install aws-durable-execution-sdk-python-testing
```

Eine vollständige Dokumentation und Beispiele finden Sie im Python-Test-SDK unter GitHub.

Das Test-SDK bietet zwei Testmodi: lokale Tests für schnelle Komponententests und Cloud-Tests für Integrationstests mit bereitgestellten Funktionen.

Lokale Tests

Beim lokalen Testen werden Ihre dauerhaften Funktionen in Ihrer Entwicklungsumgebung ausgeführt, ohne dass bereitgestellte Ressourcen erforderlich sind. Der Test-Runner führt Ihren Funktionscode direkt aus und erfasst alle Operationen zur Überprüfung.

Verwenden Sie lokale Tests für Komponententests, testgetriebene Entwicklung und CI/CD Pipelines. Tests werden lokal ohne Netzwerklatenz oder zusätzliche Kosten ausgeführt.

Beispieltest:

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
```

Der Test-Runner erfasst den Ausführungsstatus einschließlich des Endergebnisses, der Ergebnisse einzelner Schritte, Warteoperationen, Rückrufe und aller Fehler. Sie können Operationen anhand des Namens überprüfen oder alle Operationen wiederholen, um das Ausführungsverhalten zu überprüfen.

Die Ausführung wird gespeichert

Das Test-SDK verwendet Ausführungsspeicher, um Testausführungsdaten beizubehalten. Standardmäßig verwenden Tests einen In-Memory-Speicher, der schnell ist und keine Bereinigung erfordert. Zum Debuggen oder Analysieren des Ausführungsverlaufs können Sie einen Dateisystemspeicher verwenden, der Ausführungen als JSON-Dateien speichert.

In-Memory-Speicher (Standard):

Der In-Memory-Speicher speichert die Ausführungsdaten während der Testläufe im Speicher. Daten gehen verloren, wenn die Tests abgeschlossen sind, und eignen sich daher ideal für Standard-Komponententests und CI/CD Pipelines, bei denen Sie Ausführungen nach Abschluss der Tests nicht überprüfen müssen.

Dateisystemspeicher:

Der Dateisystemspeicher speichert Ausführungsdaten als JSON-Dateien auf der Festplatte. Jede Ausführung wird in einer separaten Datei gespeichert, sodass der Ausführungsverlauf nach Abschluss der Tests leicht überprüft werden kann. Verwenden Sie den Dateisystemspeicher, um komplexe Testfehler zu debuggen oder Ausführungsmuster im Laufe der Zeit zu analysieren.

Konfigurieren Sie den Speicher mithilfe von Umgebungsvariablen:

```
# Use filesystem store
```

```
export AWS_DEX_STORE_TYPE=filesystem
export AWS_DEX_STORE_PATH=./test-executions

# Run tests
pytest tests/
```

Ausführungsdateien werden mit bereinigten Namen gespeichert und enthalten den vollständigen Ausführungsstatus, einschließlich Operationen, Prüfpunkten und Ergebnissen. Der Dateisystemspeicher erstellt automatisch das Speicherverzeichnis, falls es nicht existiert.

Testen in der Cloud

Cloud-Tests rufen bereitgestellte dauerhafte Funktionen auf AWS und rufen deren Ausführungsverlauf mithilfe der Lambda-API ab. Verwenden Sie Cloud-Tests, um das Verhalten in produktionsähnlichen Umgebungen mit echten Diensten und Konfigurationen zu überprüfen. AWS

Für Cloud-Tests sind eine bereitgestellte Funktion und AWS Anmeldeinformationen mit Berechtigungen zum Aufrufen von Funktionen und zum Lesen des Ausführungsverlaufs erforderlich:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "lambda:InvokeFunction",
                "lambda:GetDurableExecution",
                "lambda:GetDurableExecutionHistory"
            ],
            "Resource": "arn:aws:lambda:region:account-id:function:function-name"
        }
    ]
}
```

Beispiel für einen Cloud-Test:

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"
```

Cloud-Tests rufen die tatsächlich bereitgestellte Funktion auf und rufen den Ausführungsverlauf von ab AWS. Auf diese Weise können Sie die Integration mit anderen AWS Diensten überprüfen, Leistungsmerkmale validieren und Tests mit produktionsähnlichen Daten und Konfigurationen durchführen.

Was muss getestet werden?

Testen Sie langlebige Funktionen, indem Sie die Ausführungsergebnisse, das Betriebsverhalten und die Fehlerbehandlung überprüfen. Konzentrieren Sie sich eher auf die Richtigkeit der Geschäftslogik als auf Implementierungsdetails.

Überprüfen Sie die Ausführungsergebnisse: Prüfen Sie, ob die Funktionen die erwarteten Werte für die angegebenen Eingaben zurückgeben. Testen Sie sowohl erfolgreiche Ausführungen als auch Fehlerfälle, um sicherzustellen, dass Funktionen ungültige Eingaben angemessen behandeln.

Überprüfen Sie die Ausführung des Vorgangs: Stellen Sie sicher, dass Schritte, Wartezeiten und Callbacks wie erwartet ausgeführt werden. Überprüfen Sie die Schrittergebnisse, um sicherzustellen, dass Zwischenoperationen korrekte Werte ergeben. Stellen Sie sicher, dass Wartevorgänge mit geeigneten Timeouts konfiguriert sind und dass Callbacks mit den richtigen Einstellungen erstellt wurden.

Testen Sie die Fehlerbehandlung: Stellen Sie sicher, dass Funktionen korrekt fehlschlagen, und geben Sie beschreibende Fehlermeldungen ein, wenn eine ungültige Eingabe erfolgt. Testen Sie das Wiederholungsverhalten, indem Sie vorübergehende Ausfälle simulieren und sicherstellen, dass der Vorgang ordnungsgemäß wiederholt wird. Stellen Sie sicher, dass permanente Fehler nicht zu unnötigen Wiederholungen führen.

Workflows validieren: Stellen Sie bei Workflows mit mehreren Schritten sicher, dass die Vorgänge in der richtigen Reihenfolge ausgeführt werden. Testen Sie die bedingte Verzweigung, um sicherzustellen, dass die verschiedenen Ausführungspfade korrekt funktionieren. Überprüfen Sie, ob parallel Operationen gleichzeitig ausgeführt werden und zu erwarteten Ergebnissen führen.

Die SDK-Dokumentationsrepositorien enthalten umfangreiche Beispiele für Testmuster, darunter mehrstufige Workflows, Fehlerszenarien, Timeout-Behandlung und Abfragemuster.

Teststrategie

Verwenden Sie lokale Tests für Komponententests während der Entwicklung und in CI/CD Pipelines. Lokale Tests werden schnell ausgeführt, erfordern keine AWS Anmeldeinformationen und bieten sofortiges Feedback zu Codeänderungen. Schreiben Sie lokale Tests, um Geschäftslogik, Fehlerbehandlung und Betriebsverhalten zu überprüfen.

Verwenden Sie Cloud-Tests für Integrationstests, bevor Sie sie in der Produktion einsetzen. Cloud-Tests verifizieren das Verhalten anhand realer AWS Dienste und Konfigurationen, validieren

Leistungsmerkmale und testen end-to-end Workflows. Führen Sie Cloud-Tests in Staging-Umgebungen durch, um Integrationsprobleme zu erkennen, bevor sie in die Produktion gelangen.

Simulieren Sie externe Abhängigkeiten in lokalen Tests, um die Funktionslogik zu isolieren und Tests schnell zu halten. Verwenden Sie Cloud-Tests, um die tatsächliche Integration mit externen Diensten wie Datenbanken und anderen AWS Diensten zu überprüfen. APIs

Schreiben Sie gezielte Tests, die ein bestimmtes Verhalten verifizieren. Verwenden Sie aussagekräftige Testnamen, die erklären, was getestet wird. Gruppieren Sie verwandte Tests und verwenden Sie Testvorrichtungen für den allgemeinen Setup-Code. Halten Sie Tests einfach und vermeiden Sie komplexe Testlogik, die schwer zu verstehen ist.

Fehler beim Debuggen

Wenn Tests fehlschlagen, überprüfen Sie das Ausführungsergebnis, um zu verstehen, was schief gelaufen ist. Überprüfen Sie den Ausführungsstatus, um festzustellen, ob die Funktion erfolgreich war, fehlgeschlagen ist oder ob das Zeitlimit überschritten wurde. Lesen Sie die Fehlermeldungen, um die Fehlerursache zu verstehen.

Untersuchen Sie die einzelnen Betriebsergebnisse, um herauszufinden, wo das Verhalten von den Erwartungen abwich. Überprüfen Sie die Ergebnisse der einzelnen Schritte, um zu sehen, welche Werte erzielt wurden. Überprüfen Sie die Reihenfolge der Operationen, um zu bestätigen, dass die Operationen in der erwarteten Reihenfolge ausgeführt wurden. Zählen Sie die Operationen, um sicherzustellen, dass die richtige Anzahl von Schritten, Wartezeiten und Callbacks erstellt wurde.

Zu den häufigsten Problemen gehören nicht deterministischer Code, der bei der Wiedergabe zu unterschiedlichen Ergebnissen führt, gemeinsamer Status durch globale Variablen, der während der Wiedergabe unterbrochen wird, und fehlende Operationen aufgrund bedingter Logikfehler. Verwenden Sie Standard-Debugger und Protokollierung, um den Funktionscode Schritt für Schritt durchzugehen und den Ausführungsablauf zu verfolgen.

Sehen Sie sich bei Cloud-Tests den Ausführungsverlauf in den CloudWatch Protokollen an, um detaillierte Betriebsprotokolle einzusehen. Verwenden Sie Tracing, um den Ausführungsablauf zwischen Diensten zu verfolgen und Engpässe zu identifizieren.

Überwachung langlebiger Funktionen

Sie können Ihre dauerhaften Funktionen mithilfe von CloudWatch Metriken, CloudWatch Protokollen und Tracing überwachen. Da langlebige Funktionen über längere Zeiträume ausgeführt werden

können und sich über mehrere Funktionsaufrufen erstrecken, erfordert ihre Überwachung das Verständnis ihrer jeweiligen Ausführungsmuster, einschließlich Checkpoints, Zustandsübergängen und Wiedergabeverhalten.

CloudWatch Metriken

Lambda veröffentlicht automatisch Metriken ohne zusätzliche Kosten. CloudWatch Dauerhafte Funktionen bieten zusätzliche Metriken, die über die Standard-Lambda-Metriken hinausgehen, und unterstützen Sie bei der Überwachung von Workflows mit langer Laufzeit, der Statusverwaltung und der Ressourcennutzung.

Zuverlässige Ausführungsmetriken

Lambda gibt die folgenden Metriken für dauerhafte Ausführungen aus:

Metrik	Description
`ApproximateRunningDurableExecutions`	Anzahl der dauerhaften Ausführungen im Status RUNNING
`ApproximateRunningDurableExecutionsUtilization`	Prozentsatz des derzeit in Anspruch genommenen maximalen Kontingents für laufende dauerhafte Ausführungen Ihres Kontos
`DurableExecutionDuration`	Verstrichene Zeit in Millisekunden, während der eine dauerhafte Ausführung im Status RUNNING verharrte
`DurableExecutionStarted`	Anzahl der begonnenen dauerhaften Ausführungen
`DurableExecutionStopped`	Anzahl der dauerhaften Ausführungen, die die API nicht mehr verwenden StopDurableExecution
`DurableExecutionSucceeded`	Anzahl der dauerhaften Ausführungen, die erfolgreich abgeschlossen wurden
`DurableExecutionFailed`	Anzahl der dauerhaften Ausführungen, die mit einem Fehler abgeschlossen wurden
`DurableExecutionTimedOut`	Anzahl der dauerhaften Ausführungen, die ihr konfiguriertes Ausführungstimeout überschritten haben

Metrik	Description
`DurableExecutionOperations`	Gesamtzahl der Operationen, die innerhalb einer dauerhaften Ausführung ausgeführt wurden (max. 3.000)
`DurableExecutionStorageWrittenBytes`	Kumulierte Datenmenge in Byte, die bei einer dauerhaften Ausführung beibehalten wurde (max.: 100 MB)

CloudWatch Metriken

Lambda gibt Standardmetriken für Aufruf, Leistung und Parallelität für langlebige Funktionen aus. Da sich eine dauerhafte Ausführung über mehrere Funktionsaufrufen erstrecken kann, während sie Checkpoints und Wiederholungen durchläuft, verhalten sich diese Metriken anders als bei Standardfunktionen:

- Aufrufe: Zählt jeden Funktionsaufruf, einschließlich Wiederholungen. Eine einzige dauerhafte Ausführung kann mehrere Aufruf-Datenpunkte generieren.
- Dauer: Misst jeden Funktionsaufruf separat. Wird `DurableExecutionDuration` für die Gesamtzeit verwendet, die eine einzelne dauerhafte Ausführung benötigt hat.
- Fehler: Verfolgt Fehler beim Aufrufen von Funktionen. Wird `DurableExecutionFailed` für Fehler auf Ausführungsebene verwendet.

Eine vollständige Liste der Standard-Lambda-Metriken finden Sie unter Typen von Metriken für Lambda-Funktionen.

Alarme erstellen CloudWatch

Erstellen Sie CloudWatch Alarme, um Sie zu benachrichtigen, wenn Messwerte Schwellenwerte überschreiten. Zu den häufigsten Alarmen gehören:

- `ApproximateRunningDurableExecutionsUtilization`überschreitet 80% Ihres Kontingents
- `DurableExecutionFailed`steigt über einen Schwellenwert
- `DurableExecutionTimedOut`weist darauf hin, dass bei Hinrichtungen ein Timeout eintritt
- `DurableExecutionStorageWrittenBytes`nähert sich den Speichergrenzen

Weitere Informationen finden Sie unter CloudWatch Alarme verwenden. .

EventBridge Ereignisse

Lambda veröffentlicht dauerhafte Ereignisse zur Änderung des Ausführungsstatus unter. EventBridge Sie können diese Ereignisse verwenden, um Workflows auszulösen, Benachrichtigungen zu senden oder Änderungen des Ausführungslebenszyklus in Ihren dauerhaften Funktionen nachzuverfolgen.

Ereignisse zur Änderung des Status dauerhafter Ausführung

Lambda gibt jedes Mal ein Ereignis aus, EventBridge wenn sich der Status einer dauerhaften Ausführung ändert. Diese Ereignisse weisen die folgenden Merkmale auf:

- Quelle: `aws.lambda`
- Art des Details: `Durable Execution Status Change`

Statusänderungsereignisse werden für die folgenden Ausführungsstatus veröffentlicht:

- `RUNNING`- Die Ausführung wurde gestartet
- `SUCCEEDED`- Die Ausführung wurde erfolgreich abgeschlossen
- `STOPPED`- Die Ausführung wurde mithilfe der StopDurableExecution API gestoppt
- `FAILED`- Die Ausführung ist mit einem Fehler fehlgeschlagen
- `TIMED_OUT`- Die Ausführung hat das konfigurierte Timeout überschritten

Das folgende Beispiel zeigt ein dauerhaftes Ereignis zur Änderung des Ausführungsstatus:

```
{
  "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"
  }
}
```

Bei Terminalzuständen (`SUCCEEDED`,`STOPPED`,`FAILED`,`TIMED_OUT`) umfasst das Ereignis ein `endTimestamp` Feld, das angibt, wann die Ausführung abgeschlossen wurde.

EventBridge Regeln erstellen

Erstellen Sie Regeln, um dauerhafte Ereignisse zur Änderung des Ausführungsstatus an Ziele wie Amazon Simple Notification Service, Amazon Simple Queue Service oder andere Lambda-Funktionen weiterzuleiten.

Im folgenden Beispiel wird eine Regel erstellt, die allen Änderungen des Status der dauerhaften Ausführung entspricht:

```
{
  "source": ["aws.lambda"],
  "detail-type": ["Durable Execution Status Change"]
}
```

Im folgenden Beispiel wird eine Regel erstellt, die nur auf fehlgeschlagene Ausführungen zutrifft:

```
{
  "source": ["aws.lambda"],
  "detail-type": ["Durable Execution Status Change"],
  "detail": {
    "status": ["FAILED"]
  }
}
```

Im folgenden Beispiel wird eine Regel erstellt, die Statusänderungen für eine bestimmte Funktion abgleicht:

```
{
  "source": ["aws.lambda"],
  "detail-type": ["Durable Execution Status Change"],
  "detail": {
    "functionArn": [{
      "prefix": "arn:aws:lambda:us-east-1:123456789012:function:my-function"
    }]
  }
}
```

Weitere Informationen zum Erstellen von Regeln finden Sie in den EventBridge Amazon-Tutorials im EventBridge Benutzerhandbuch.

AWS X-Ray Rückverfolgung

Sie können die Röntgenverfolgung für Ihre dauerhaften Funktionen aktivieren. Lambda übergibt den X-Ray-Trace-Header an die dauerhafte Ausführung, sodass Sie Anfragen in Ihrem gesamten Workflow verfolgen können.

Um X-Ray; Tracing mit der Lambda-Konsole zu aktivieren, wählen Sie Ihre Funktion aus, wählen Sie dann Configuration, Monitoring and Operations Tools und aktivieren Sie Active Tracing unter X-Ray.

Um X-Ray Tracing zu aktivieren, verwenden Sie: AWS CLI

```
aws lambda update-function-configuration \
    --function-name my-durable-function \
    --tracing-config Mode=Active
```

Um die AWS X-Ray Ablaufverfolgung zu aktivieren, verwenden Sie: AWS SAM

```
Resources:
  MyDurableFunction:
    Type: AWS::Serverless::Function
    Properties:
      Tracing: Active
      DurableConfig:
        ExecutionTimeout: 3600
```

Weitere Informationen zu X-Ray finden Sie im AWS X-Ray Developer Guide.

Bewährte Methoden für langlebige Lambda-Funktionen

Dauerhafte Funktionen verwenden ein auf Wiedergabe basierendes Ausführungsmodell, das andere Muster erfordert als herkömmliche Lambda-Funktionen. Folgen Sie diesen bewährten Methoden, um zuverlässige und kostengünstige Workflows zu erstellen.

Schreiben Sie deterministischen Code

Während der Wiedergabe wird Ihre Funktion von Anfang an ausgeführt und muss demselben Ausführungspfad folgen wie die ursprüngliche Ausführung. Code außerhalb dauerhafter Operationen muss deterministisch sein und bei denselben Eingaben dieselben Ergebnisse liefern.

Unterteilen Sie nichtdeterministische Operationen schrittweise:

- Generierung von Zufallszahlen und UUIDs
- Aktuelle Uhrzeit oder Zeitstempel
- Externe API-Aufrufe und Datenbankabfragen
- Dateisystemoperationen

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}
```

Wichtig

Verwenden Sie keine globalen Variablen oder Schließungen, um den Status zwischen Schritten zu teilen. Übergeben Sie Daten durch Rückgabewerte. Der globale Status wird während der Wiedergabe unterbrochen, weil Schritte zwischengespeicherte Ergebnisse zurückgeben, globale Variablen jedoch zurückgesetzt werden.

Vermeiden Sie Mutationen bei Verschlüssen: Variablen, die in Closures erfasst wurden, können während der Wiedergabe Mutationen verlieren. Schritte geben zwischengespeicherte Ergebnisse zurück, aber Variablenaktualisierungen außerhalb des Schritts werden nicht wiederholt.

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}
```

Design für Idempotenz

Operationen können aufgrund von Wiederholungen oder Wiederholungen mehrfach ausgeführt werden. Operationen, die nicht idempotent sind, haben doppelte Nebenwirkungen, z. B. doppelte Gebühren für Kunden oder das Senden mehrerer E-Mails.

Verwenden Sie Idempotenz-Token: Generieren Sie Token innerhalb von Schritten und fügen Sie sie externen API-Aufrufen hinzu, um doppelte Operationen zu vermeiden.

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}
```

at-most-onceSemantik verwenden: Für kritische Operationen, die sich niemals duplizieren dürfen (Finanztransaktionen, Inventarabzüge), konfigurieren Sie den Ausführungsmodus. at-most-once

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')
)
```

Datenbankidempotenz: Verwenden Sie check-before-write Muster, bedingte Aktualisierungen oder Upsert-Operationen, um doppelte Datensätze zu verhindern.

Verwalten Sie den Status effizient

Jeder Checkpoint speichert den Status im persistenten Speicher. Große Zustandsobjekte erhöhen die Kosten, verlangsamen Checkpoints und beeinträchtigen die Leistung. Speichern Sie nur wichtige Daten zur Workflow-Koordination.

Halten Sie den Status minimal:

- Speichern IDs und Verweise, nicht vollständige Objekte
- Rufen Sie nach Bedarf detaillierte Daten in Schritten ab
- Verwenden Sie Amazon S3 oder DynamoDB für große Datenmengen, übergeben Sie Referenzen im Status
- Vermeiden Sie es, große Nutzlasten zwischen den Schritten weiterzugeben

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}
```

Entwerfen Sie effektive Schritte

Stufen sind die grundlegende Arbeitseinheit für langlebige Funktionen. Gut durchdachte Schritte erleichtern das Verständnis, das Debuggen und die Wartung von Workflows.

Prinzipien der schrittweisen Gestaltung:

- Verwende aussagekräftige Namen — Namen wie, `validate-order` anstatt Logs und Fehler leichter verständlich zu `step1` machen
- Namen statisch halten — Verwenden Sie keine dynamischen Namen mit Zeitstempeln oder Zufallswerten. Schrittnamen müssen für die Wiedergabe deterministisch sein
- Ausgewogene Granularität — Teilen Sie komplexe Vorgänge in gezielte Schritte auf, vermeiden Sie jedoch übermäßig kleine Schritte, die den Aufwand an den Checkpoints erhöhen
- Gruppenbezogene Operationen — Operationen, die gemeinsam erfolgreich sein oder fehlschlagen sollten, gehören in denselben Schritt

Verwenden Sie Warteoperationen effizient

Wartevorgänge unterbrechen die Ausführung, ohne Ressourcen zu verbrauchen oder Kosten zu verursachen. Verwenden Sie sie, anstatt Lambda am Laufen zu halten.

Zeitbasierte Wartezeiten: Verwenden Sie diese Option `context.wait()` für Verzögerungen anstelle von oder. `setTimeout` `sleep`

Externe Rückrufe: `context.waitForCallback()` Wird verwendet, wenn auf externe Systeme gewartet wird. Stellen Sie immer Timeouts ein, um unbestimmte Wartezeiten zu vermeiden.

Polling: Verwenden Sie diese Option `context.waitForCondition()` mit exponentiellem Backoff, um externe Dienste abzufragen, ohne sie zu überfordern.

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)
)
```

Weitere Überlegungen

Fehlerbehandlung: Versuchen Sie es erneut mit vorübergehenden Fehlern wie Netzwerk-Timeouts und Ratenbegrenzungen. Versuchen Sie es nicht erneut mit dauerhaften Fehlern wie ungültigen Eingabe- oder Authentifizierungsfehlern. Konfigurieren Sie Wiederholungsstrategien mit einer angemessenen maximalen Anzahl von Versuchen und Backoff-Raten. Ausführliche Beispiele finden Sie unter Fehlerbehandlung und Wiederholungen.

Leistung: Minimiere die Größe von Checkpoints, indem du Referenzen statt vollständiger Payloads speicherst. Verwenden Sie `context.parallel()` und`context.map()`, um unabhängige Operationen gleichzeitig auszuführen. Batch-bezogene Operationen zur Reduzierung des Checkpoint-Overheads.

Versionierung: Rufen Sie Funktionen mit Versionsnummern oder Aliasnamen auf, um Ausführungen an bestimmte Codeversionen zu binden. Stellen Sie sicher, dass neue Codeversionen den Status

älterer Versionen verarbeiten können. Benennen Sie Schritte nicht um und ändern Sie ihr Verhalten nicht so, dass die Wiedergabe unterbrochen wird.

Serialisierung: Verwenden Sie JSON-kompatible Typen für Operationseingaben und -ergebnisse. Konvertiert Datumsangaben in ISO-Zeichenketten und benutzerdefinierte Objekte in einfache Objekte, bevor Sie sie an dauerhafte Operationen übergeben.

Überwachung: Aktivieren Sie die strukturierte Protokollierung mit Ausführungs IDs - und Schrittnamen. Richten Sie CloudWatch Alarme für Fehlerraten und Ausführungsdauer ein. Verwenden Sie Tracing, um Engpässe zu identifizieren. Eine ausführliche Anleitung finden Sie unter Überwachung und Debuggen.

Testen: Testen Sie Happy Path, Fehlerbehandlung und Wiedergabeverhalten. Testen Sie Timeout-Szenarien für Rückrufe und Wartezeiten. Verwenden Sie lokale Tests, um die Iterationszeit zu reduzieren. Eine ausführliche Anleitung finden Sie unter Dauerhafte Funktionen testen.

Häufige Fehler, die Sie vermeiden sollten: Verschachteln Sie `context.step()` Anrufe nicht, sondern verwenden Sie stattdessen untergeordnete Kontexte. Fassen Sie nicht deterministische Operationen schrittweise zusammen. Legen Sie immer Timeouts für Rückrufe fest. Gleichen Sie die Granularität der Schritte mit dem Checkpoint-Overhead ab. Speichern Sie Referenzen statt großer Objekte im Status.

Weitere Ressourcen

- Python SDK-Dokumentation — Vollständige API-Referenz, Testmuster und erweiterte Beispiele
- TypeScript SDK-Dokumentation — Vollständige API-Referenz, Testmuster und Beispiele für Fortgeschrittene

Von Lambda verwaltete Instanzen

Mit Lambda Managed Instances können Sie Lambda-Funktionen auf Ihren EC2 Amazon-Instances der aktuellen Generation ausführen, einschließlich Graviton4, netzwerkoptimierten Instances und anderen speziellen Rechenoptionen, ohne Instance-Lebenszyklen, Betriebssystem- und Sprachlaufzeit-Patching, Routing, Lastenausgleich oder Skalierungsrichtlinien verwalten zu müssen. Mit Lambda Managed Instances profitieren Sie von EC2 Preisvorteilen, einschließlich EC2 Savings Plans und Reserved Instances.

Eine Liste der unterstützten Instance-Typen finden Sie auf der Seite mit den AWS Lambda Preisen und wählen Sie Ihre AWS-Region aus.

Die wichtigsten Funktionen

Lambda Managed Instances bietet die folgenden Funktionen:

- Wählen Sie geeignete Instances aus — Wählen Sie geeignete Instances auf der Grundlage von Leistungs- und Kostenanforderungen aus, einschließlich Zugriff auf aktuelle Versionen CPUs wie Graviton4, konfigurierbare Speicher-CPU-Verhältnisse und Netzwerke mit hoher Bandbreite.
- Automatische Bereitstellung — stellt AWS automatisch geeignete Instanzen bereit und aktiviert Umgebungen zur Funktionsausführung.
- Dynamische Skalierung — Instanzen werden dynamisch auf der Grundlage Ihrer Funktionsdatenverkehrsmuster skaliert.
- Vollständig verwaltetes Erlebnis — AWS kümmert sich um Infrastrukturmanagement, Skalierung, Patching und Routing mit denselben umfassenden Integrationen zur Ereignisquelle, mit der Sie vertraut sind.

Wann sollten Lambda Managed Instances verwendet werden

Ziehen Sie Lambda Managed Instances für die folgenden Anwendungsfälle in Betracht:

- Vorhersehbare Workloads mit hohem Volumen — Am besten geeignet für Steady-State-Workloads ohne unerwartete Datenverkehrsspitzen. Lambda Managed Instances lassen sich standardmäßig so skalieren, dass sie eine Verdoppelung des Datenverkehrs innerhalb von fünf Minuten bewältigen können.

- Leistungskritische Anwendungen — Zugriff auf aktuelle CPUs, unterschiedliche Speicher-CPU-Verhältnisse und hoher Netzwerkdurchsatz
- Regulatorische Anforderungen — Granulare Governance-Anforderungen mit Kontrolle über VPC und Instance-Platzierung
- Vielfältige Anwendungen — ereignisgesteuerte Anwendungen, media/data Verarbeitung, Webanwendungen und ältere Workloads, die auf serverlose Systeme migriert werden

Funktionsweise

Lambda Managed Instances verwendet Kapazitätsanbieter als Grundlage für die Ausführung Ihrer Funktionen:

1. Erstellen Sie einen Kapazitätsanbieter — Definieren Sie, wo Ihre Funktionen ausgeführt werden, indem Sie die VPC-Konfiguration und optional die Instanzanforderungen und die Skalierungskonfiguration angeben
2. Erstellen Sie Ihre Funktion — Erstellen Sie Lambda-Funktionen wie gewohnt und hängen Sie sie an einen Kapazitätsanbieter an
3. Veröffentlichen Sie eine Funktionsversion — Funktionsversionen werden nach der Veröffentlichung auf Kapazitätsanbieter-Instanzen aktiv

Wenn Sie eine Funktionsversion mit einem Kapazitätsanbieter veröffentlichen, startet Lambda Managed Instances in Ihrem Konto. Aus Gründen der AZ-Resilienz werden standardmäßig drei Instances und drei Ausführungsumgebungen gestartet, bevor Ihre Funktionsversion als AKTIV markiert wird. Wenn Sie eine Funktion an einen vorhandenen Kapazitätsanbieter anhängen, der bereits andere Funktionen ausführt, kann es sein, dass Lambda keine neuen Instances hochfährt, wenn die verfügbaren Instances bereits über Kapazitäten für die Ausführungsumgebungen der neuen Funktion verfügen.

Modell der Parallelität

Lambda Managed Instances unterstützen mehrere gleichzeitige Aufrufe, wobei eine Ausführungsumgebung mehrere Aufrufe gleichzeitig verarbeiten kann. Dies unterscheidet sich vom Berechnungstyp Lambda (Standard), der ein einzelnes Parallelitätsmodell bereitstellt, bei dem eine Ausführungsumgebung maximal einen Aufruf gleichzeitig ausführen kann. Multi-Concurrency ermöglicht eine bessere Nutzung Ihrer zugrunde liegenden EC2 Instances und ist besonders

vorteilhaft für IO-intensive Anwendungen wie Webservices oder Batch-Jobs. Diese Änderung des Ausführungsmodells bedeutet, dass Thread-Sicherheit, Statusverwaltung und Kontextisolation je nach Laufzeit unterschiedlich gehandhabt werden müssen.

Tenance und Isolierung

Der Lambda-Compute-Typ (Standard) ist mandantenfähig und nutzt die Firecracker MicroVM-Technologie, um die Ausführungsumgebungen zu isolieren, die auf gemeinsam genutzten Lambda-Flotten ausgeführt werden. Lambda Managed Instances werden in Ihrem Konto ausgeführt und bieten die neuesten EC2 Hardware- und Preisoptionen. Managed Instances verwenden Container, die auf EC2 Nitro-Instances laufen, um Isolation zu gewährleisten, und nicht Firecracker. Kapazitätsanbieter dienen als Sicherheitsgrenze für Lambda-Funktionen. Funktionen werden in Containern innerhalb von Instanzen ausgeführt.

Verwaltete Instanzen verstehen

Die Funktionen von Lambda Managed Instances werden auf EC2 verwalteten Instanzen in Ihrem Konto ausgeführt. Diese Instanzen werden vollständig von Lambda verwaltet, was bedeutet, dass Sie im Vergleich zu EC2 Standardinstanzen eingeschränkte Berechtigungen für sie haben. Sie können Lambda Managed Instances in Ihrem Konto wie folgt identifizieren:

- Das Vorhandensein des `Operator` Felds in EC2 `DescribeInstances` der Ausgabe
- Das `aws:lambda:capacity-provider` Tag auf der Instanz

Sie können auf diesen Instances keine EC2 Standardoperationen direkt ausführen, z. B. sie manuell beenden. Um verwaltete Instanzen zu löschen, löschen Sie den zugehörigen Kapazitätsanbieter. Lambda beendet dann die Instances im Rahmen des Löschvorgangs für den Kapazitätsanbieter.

Preisgestaltung

Lambda Managed Instances verwendet eine Preisgestaltung auf EC2 Basis der Preise mit einer Verwaltungsgebühr von 15% zusätzlich zu den EC2 Instanzkosten. Dieses Preismodell unterstützt EC2 Savings Plans, Reserved Instances und alle anderen Preisnachlässe, die für Ihre EC2 Nutzung gelten. Weitere Informationen finden Sie auf der Preisseite: https://aws.amazon.com/lambda/Preise/

Wichtig: EC2 Preisnachlässe gelten nur für die zugrunde liegende EC2 Rechenleistung, nicht für die Verwaltungsgebühr.

Wie sich Lambda Managed Instances vom Compute-Typ Lambda (Standard) unterscheiden

Lambda Managed Instances ändert die Art und Weise, wie Lambda Anfragen verarbeitet, im Vergleich zu Lambda (Standard).

Die wichtigsten Unterschiede:

	Lambda (Standard)	Von Lambda verwaltete Instanzen
Parallelitätsmodell	Einzelnes Parallelitätsmodell, bei dem eine Ausführungsumgebung maximal einen Aufruf gleichzeitig unterstützen kann	Gleichzeitige Aufrufe, bei denen eine Ausführungsumgebung mehrere Aufrufe gleichzeitig verarbeiten kann, wodurch der Durchsatz insbesondere bei IO-intensiven Anwendungen erhöht wird
Mietverhältnis und Isolierung	Mehrmandantenfähig, wobei die Firecracker MicroVM-Technologie zur Isolierung zwischen Ausführungsumgebungen verwendet wird, die auf gemeinsam genutzten Lambda-Flotten ausgeführt werden	Führen Sie es in Ihrem Konto aus und verwenden Sie Nitro, um die Isolierung zu gewährleisten. EC2 Kapazitätsanbieter dienen als Sicherheitsgrenze, wobei Funktionen in Containern innerhalb von Instanzen ausgeführt werden
Preismodell	Preisgestaltung pro Anfrage und Dauer	Instanzbasierte Preisgestaltung mit EC2 Preismodellen, einschließlich On-Demand-Instances und Reserved Instances, und Sparoptionen wie Compute Savings Plans
Skalierungsverhalten	Skaliert, wenn es keine freie Ausführungsumgebung für die Verarbeitung eines eingehenden	Skaliert asynchron nur basierend auf der CPU-Ressourcenauslastung, ohne Kaltstarts. Skaliert auf ein Minimum an Ausführun

	Lambda (Standard)	Von Lambda verwaltete Instanzen
	Aufrufs gibt (Kaltstart). Skaliert ohne Verkehr auf Null	gsumgebungen, die ohne Datenverkehr konfiguriert sind
Am besten geeignet für	Funktionen mit hohem Datenverkehr, die einige Kaltstartzeiten verkraften können, oder Anwendungen ohne Dauerlast, die von einer Skalierung auf Null profitieren	Hochvolumiger vorhersehbarer Datenverkehr funktioniert, wenn Sie die Flexibilität, Preispläne und Hardwareoptionen von EC2

Nächste Schritte

- Erfahren Sie mehr über Kapazitätsanbieter für Lambda Managed Instances
- Skalierung für Lambda Managed Instances verstehen
- Lesen Sie laufzeitspezifische Anleitungen für Java, Node.js und Python
- Konfigurieren Sie die VPC-Konnektivität für Ihre Kapazitätsanbieter
- Verstehen Sie Sicherheit und Berechtigungen für Lambda Managed Instances

Erste Schritte mit Lambda Managed Instances

Eine Lambda Managed Instance-Funktion erstellen (Konsole)

Sie können die Lambda-Konsole verwenden, um eine Managed Instance-Funktion zu erstellen, die auf EC2 Amazon-Instances ausgeführt wird, die von einem Kapazitätsanbieter verwaltet werden.

Wichtig: Bevor Sie eine Managed Instance-Funktion erstellen, müssen Sie zunächst einen Kapazitätsanbieter erstellen. Für diese Funktionen ist ein Kapazitätsanbieter erforderlich, der die EC2 Amazon-Infrastruktur definiert, auf der Ihre Funktionen ausgeführt werden.

So erstellen Sie eine Lambda Managed Instance-Funktion (Konsole)

1. Öffnen Sie die Lambda-Konsole.
2. Wählen Sie im linken Navigationsbereich Capacity Providers aus.
3. Wählen Sie Kapazitätsanbieter erstellen aus.

4. Geben Sie im Abschnitt Einstellungen für den Kapazitätsanbieter einen Namen für Ihren Kapazitätsanbieter ein.
5. Wählen Sie VPC und Berechtigungen für Ihren Kapazitätsanbieter aus. Sie können entweder ein vorhandenes verwenden oder ein neues erstellen. Informationen zum Erstellen der erforderlichen Operatorrolle finden Sie unter Lambda-Operatorrolle für Lambda Managed Instances.
6. Erweiterten Sie Advanced settings (Erweiterte Einstellungen).
7. Definieren Sie Ihre Instance-Anforderungen, indem Sie die Prozessorarchitektur und die Instance-Typen auswählen.
8. Geben Sie unter Auto Scaling die maximale Anzahl von EC2 v CPUs für Ihren Kapazitätsanbieter an. Sie können auch den Modus Manuelle Instanzskalierung wählen, um Ihren eigenen Skalierungswert für eine präzise Steuerung festzulegen.
9. Wählen Sie Create Capacity Provider aus, um einen neuen zu erstellen.
10. Wählen Sie Create function.
11. Wählen Sie Verfassen von Grund auf aus.
12. Geben Sie im Bereich Grundinformationen einen Funktionsnamen ein.
13. Wählen Sie für Runtime eine der unterstützten Runtimes aus.
14. Wählen Sie die Architektur für Ihre Funktion (dieselbe, die Sie für den Kapazitätsanbieter ausgewählt haben). Standardmäßig x86_64.
15. Stellen Sie unter Berechtigungen sicher, dass Sie über die Berechtigungen für die gewählte Ausführungsrolle verfügen. Andernfalls können Sie eine neue Rolle erstellen.
16. Wählen Sie unter Zusätzliche Konfigurationen den Compute-Typ als Lambda Managed Instances aus.
17. Der Kapazitätsanbieter-ARN des Kapazitätsanbieters, den Sie in den vorherigen Schritten erstellt haben, sollte vorab ausgewählt sein.
18. Wählen Sie Speichergröße und Verhältnis Arbeitsspeicher (GiB) pro vCPU aus.
19. Wählen Sie Funktion erstellen.

Ihre Lambda Managed Instance-Funktion wird erstellt und stellt Kapazität auf Ihrem angegebenen Kapazitätsanbieter bereit. Die Erstellung von Funktionen dauert in der Regel mehrere Minuten. Sobald der Vorgang abgeschlossen ist, können Sie Ihren Funktionscode bearbeiten und Ihren ersten Test ausführen.

Eine Lambda Managed Instance-Funktion (AWS CLI) erstellen

Voraussetzungen

Stellen Sie vor dem Beginn sicher, dass Sie über das Folgende verfügen:

- AWS CLI — Installieren und konfigurieren Sie die AWS CLI. Weitere Informationen finden Sie unter Installation oder Aktualisierung der neuesten Version der AWS CLI.
- IAM-Berechtigungen — Ihr IAM-Benutzer oder Ihre IAM-Rolle muss über Berechtigungen verfügen, um Lambda-Funktionen und Kapazitätsanbieter zu erstellen und IAM-Rollen zu übergeben. Beachten Sie, dass Sie auch angeben müssen, `iam:CreateServiceLinkedRole` ob Sie zum ersten Mal einen Kapazitätsanbieter für das Konto erstellen oder ob die Service Linked Role (SLR) gelöscht wurde.

Schritt 1: Erstellen Sie die erforderlichen IAM-Rollen

Lambda Managed Instances erfordern zwei IAM-Rollen: eine Ausführungsrolle für Ihre Funktion und eine Operator-Rolle für den Kapazitätsanbieter. Die Operator-Rolle ermöglicht es Lambda, EC2 Amazon-Instances in Ihrem Namen zu starten, zu beenden und zu überwachen. Die Funktionsausführungsrolle gewährt der Funktion Berechtigungen für den Zugriff auf andere AWS Dienste und Ressourcen.

So erstellen Sie die Lambda-Ausführungsrolle

1. Erstellen Sie ein Dokument mit Vertrauensrichtlinien, das es Lambda ermöglicht, die Rolle zu übernehmen:

```
cat > lambda-trust-policy.json << 'EOF'
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "lambda.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
```

```
EOF
```

2. Erstellen Sie die Ausführungsrolle:

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

3. Hängen Sie die grundlegende Ausführungsrichtlinie an:

```
aws iam attach-role-policy \
  --role-name MyLambdaExecutionRole \
  --policy-arn arn:aws:iam::aws:policy/service-role/AWSLambdaBasicExecutionRole
```

Um die Rolle des Kapazitätsanbieters als Betreiber zu erstellen

1. Erstellen Sie ein Dokument mit Vertrauensrichtlinien, das es Lambda ermöglicht, die Operatorrolle zu übernehmen:

```
cat > operator-trust-policy.json << 'EOF'
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "lambda.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
EOF
```

2. Erstellen Sie die Operatorrolle:

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

3. Fügen Sie die erforderliche EC2 Berechtigungsrichtlinie bei:

```
aws iam attach-role-policy \
  --role-name MyCapacityProviderOperatorRole \
  --policy-arn arn:aws:iam::aws:policy/AWSLambdaManagedEC2ResourceOperator
```

Schritt 2: VPC-Ressourcen einrichten

Lambda Managed Instances werden in Ihrer VPC ausgeführt und benötigen ein Subnetz und eine Sicherheitsgruppe.

Um VPC-Ressourcen zu erstellen

1. Erstellen Sie eine VPC:

```
VPC_ID=$(aws ec2 create-vpc \
  --cidr-block 10.0.0.0/16 \
  --query 'Vpc.VpcId' \
  --output text)
```

2. Erstellen Sie ein Subnetz:

```
SUBNET_ID=$(aws ec2 create-subnet \
  --vpc-id $VPC_ID \
  --cidr-block 10.0.1.0/24 \
  --query 'Subnet.SubnetId' \
  --output text)
```

3. Erstellen Sie eine Sicherheitsgruppe:

```
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)
```

Hinweis: Ihre Lambda Managed Instance-Funktionen erfordern eine VPC-Konfiguration, um auf Ressourcen außerhalb der VPC zuzugreifen und Telemetriedaten an Logs und X-Ray zu CloudWatch übertragen. Einzelheiten zur Konfiguration finden Sie unter Networking for Lambda Managed Instances.

Schritt 3: Erstellen eines Kapazitätsanbieters

Ein Kapazitätsanbieter verwaltet die EC2 Instances, auf denen Ihre Lambda-Funktionen ausgeführt werden.

Um einen Kapazitätsanbieter zu erstellen

```
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
```

Dieser Befehl erstellt einen Kapazitätsanbieter mit der folgenden Konfiguration:

- VPC-Konfiguration — Gibt das Subnetz und die Sicherheitsgruppe für die Instances an EC2
- Berechtigungen — Definiert die IAM-Rolle, die Lambda zur Verwaltung von Instanzen verwendet EC2
- Instanzanforderungen — Spezifiziert die x86_64-Architektur
- Skalierungskonfiguration — Legt ein Maximum von 30 V CPUs für den Kapazitätsanbieter fest

Schritt 4: Erstellen Sie eine Lambda-Funktion mit Inline-Code

Um eine Funktion mit Inline-Code zu erstellen

1. Erstellen Sie zunächst eine einfache Python-Funktion und verpacken Sie sie inline:

```
# 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. Erstellen Sie die Lambda-Funktion mithilfe der Inline-ZIP-Datei:

```
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}
```

Die Funktion wird erstellt mit:

- Laufzeit — Python 3.13
- Handler — Die `lambda_handler` Funktion in `lambda_function.py`
- Speicher — 2048 MB
- Kurzlebiger Speicher — 512 MB
- Kapazitätsanbieter — Links zu dem von Ihnen erstellten Kapazitätsanbieter

Schritt 5: Veröffentlichen Sie eine Funktionsversion

Um Ihre Funktion auf Lambda Managed Instances auszuführen, müssen Sie eine Version veröffentlichen.

Um eine Funktionsversion zu veröffentlichen

```
aws lambda publish-version \
  --function-name my-managed-instance-function
```

Dieser Befehl veröffentlicht Version 1 Ihrer Funktion und stellt sie für den Kapazitätsanbieter bereit.

Schritt 6: Rufen Sie Ihre Funktion auf

Nach der Veröffentlichung können Sie Ihre Funktion aufrufen.

Um Ihre Funktion aufzurufen

```
aws lambda invoke \
  --function-name my-managed-instance-function:1 \
  --payload '{"name": "World"}' \
  response.json

# View the response
cat response.json
```

Die Funktion wird auf den von Ihrem Kapazitätsanbieter verwalteten EC2 Instanzen ausgeführt und gibt eine Antwort zurück.

Bereinigen

Um Gebühren zu vermeiden, löschen Sie die Ressourcen, die Sie erstellt haben:

1. Löschen Sie die Funktion:

   ```
   aws lambda delete-function --function-name my-managed-instance-function
   ```

2. Löschen Sie den Kapazitätsanbieter:

   ```
   aws lambda delete-capacity-provider --capacity-provider-name my-capacity-provider
   ```

3. Löschen Sie die VPC-Ressourcen:

```
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. Löschen Sie die IAM-Rollen:

```
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
```

Schlüsselkonzepte

Lambda Managed Instances führt mehrere Kernkonzepte ein, die sich von herkömmlichen Lambda-Funktionen unterscheiden. Das Verständnis dieser Konzepte ist für die effektive Bereitstellung und Verwaltung Ihrer Funktionen in der EC2 Infrastruktur unerlässlich.

Kapazitätsanbieter bilden die Grundlage für Lambda Managed Instances. Ein Kapazitätsanbieter definiert die Recheninfrastruktur, in der Ihre Funktionen ausgeführt werden, einschließlich VPC-Konfiguration, Instanzanforderungen und Skalierungsrichtlinien. Kapazitätsanbieter dienen auch als Sicherheitsgrenze für Ihre Funktionen, was bedeutet, dass allen Funktionen, die demselben Kapazitätsanbieter zugewiesen sind, gegenseitig vertraut werden muss.

Das Skalierungsverhalten unterscheidet sich erheblich von herkömmlichen Lambda-Funktionen. Anstatt bei eingehenden Aufrufen nach Bedarf zu skalieren, skalieren Managed Instances asynchron auf der Grundlage der CPU-Ressourcennutzung. Dieser Ansatz macht Kaltstarts überflüssig, erfordert jedoch eine Planung für das Verkehrswachstum. Wenn sich Ihr Traffic innerhalb von 5 Minuten mehr als verdoppelt, kann es zu Drosselungen kommen, da Lambda die Kapazität entsprechend der Nachfrage erhöht.

Sicherheit und Berechtigungen müssen sorgfältig geprüft werden. Sie benötigen Operator-Rollenberechtigungen, damit Lambda die EC2 Ressourcen Ihrer Kapazitätsanbieter verwalten kann. Darüber hinaus benötigen Benutzer die `lambda:PassCapacityProvider` Erlaubnis,

Kapazitätsanbietern Funktionen zuzuweisen. Diese dienen als Sicherheitsschleuse, um zu kontrollieren, welche Funktionen auf einer bestimmten Infrastruktur ausgeführt werden können.

Die gleichzeitige Ausführung mehrerer Instanzen ist ein wesentliches Merkmal von Managed Instances. Jede Ausführungsumgebung kann mehrere Aufrufe gleichzeitig verarbeiten, wodurch die Ressourcennutzung für IO-intensive Anwendungen maximiert wird. Dies unterscheidet sich von herkömmlichem Lambda, bei dem jede Umgebung jeweils eine Anfrage verarbeitet. Dieses Ausführungsmodell erfordert, je nach Laufzeit, Aufmerksamkeit auf Thread-Sicherheit, Statusverwaltung und Kontextisolation.

Die folgenden Abschnitte enthalten detaillierte Informationen zu den einzelnen Kernkonzepten.

Kapazitätsanbieter

Ein Kapazitätsanbieter ist die Grundlage für den Betrieb von Lambda Managed Instances. Es dient als Sicherheitsgrenze für Ihre Funktionen und definiert die Rechenressourcen, die Lambda in Ihrem Namen bereitstellt und verwaltet.

Wenn Sie einen Kapazitätsanbieter erstellen, geben Sie Folgendes an:

- VPC-Konfiguration — Die Subnetze und Sicherheitsgruppen, in denen Instances ausgeführt werden
- Berechtigungen — IAM-Rollen für Lambda zur Verwaltung von Ressourcen EC2
- Instanzanforderungen (optional) — Einstellungen für Architektur und Instanztyp
- Skalierungskonfiguration (optional) — So skaliert Lambda Ihre Instances

Kapazitätsanbieter als Sicherheitsgrenze verstehen

Kapazitätsanbieter dienen als Sicherheitsgrenze für Lambda-Funktionen innerhalb Ihrer VPC und ersetzen die Firecracker-basierte Isolierung. Funktionen werden in Containern innerhalb von Instances ausgeführt, aber Container bieten im Gegensatz zu Firecracker Micro keine starke Sicherheitsisolierung zwischen Funktionen. VMs

Wichtige Sicherheitskonzepte:

- Kapazitätsanbieter: Die Sicherheitsgrenze, die Vertrauensstufen für Lambda-Funktionen definiert
- Container-Isolierung: Container sind KEIN Sicherheitsanbieter — verlassen Sie sich nicht auf sie, wenn es um die Sicherheit zwischen nicht vertrauenswürdigen Workloads geht

- Vertrauenstrennung: Trennen Sie Workloads, denen nicht gegenseitig vertraut wird, indem Sie unterschiedliche Kapazitätsanbieter verwenden

Erstellen eines Kapazitätsanbieters

Sie können einen Kapazitätsanbieter mithilfe der AWS CLI, der AWS Management Console oder erstellen AWS SDKs.

Verwenden von 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
```

Erforderliche Parameter

CapacityProviderName

- Ein eindeutiger Name für Ihren Kapazitätsanbieter
- Muss innerhalb Ihres AWS Kontos eindeutig sein

VpcConfig

- SubnetIds(erforderlich): Mindestens ein Subnetz, maximal 16. Verwenden Sie aus Gründen der Ausfallsicherheit Subnetze in mehreren Availability Zones
- SecurityGroupIds(optional): Sicherheitsgruppen für Ihre Instances. Standardmäßig wird die VPC-Standardsicherheitsgruppe verwendet, falls nicht angegeben

PermissionsConfig

- CapacityProviderOperatorRoleArn(erforderlich): IAM-Rolle, die es Lambda ermöglicht, EC2 Ressourcen in Ihrem Kapazitätsanbieter zu verwalten

Optionale Parameter

InstanceRequirements

Geben Sie die Architektur und die Instanztypen für Ihren Kapazitätsanbieter an:

- Architekturen: Wählen Sie `x86_64` oder`arm64`. Die Standardeinstellung ist `x86_64`
- AllowedInstanceTypes: Geben Sie die zulässigen Instanztypen an. Beispiel: `m5.8xlarge`
- ExcludedInstanceTypes: Geben Sie ausgeschlossene Instanztypen mithilfe von Platzhaltern an. Sie können nur einen von AllowedInstanceTypes oder angeben ExcludedInstanceTypes

Standardmäßig wählt Lambda die besten Instance-Typen für Ihren Workload aus. Wir empfehlen, Lambda Managed Instances die Instanztypen für Sie auswählen zu lassen, da eine Einschränkung der Anzahl möglicher Instanztypen zu einer geringeren Verfügbarkeit führen kann.

CapacityProviderScalingConfig

Konfigurieren Sie, wie Lambda Ihre Instances skaliert:

- ScalingMode: Auf `Auto` für automatische Skalierung oder `Manual` manuelle Steuerung einstellen. Die Standardeinstellung ist `Auto`
- Max. VCpu Anzahl: Maximale Anzahl von v CPUs für den Kapazitätsanbieter. Die Standardeinstellung ist 400.
- ScalingPolicies: Definieren Sie Skalierungsrichtlinien für die Zielverfolgung für die CPU- und Speicherauslastung

KmsKeyArn

Geben Sie einen AWS KMS Schlüssel für die EBS-Verschlüsselung an. Standardmäßig wird der AWS verwaltete Schlüssel verwendet, falls nicht angegeben.

Tags

Fügen Sie Tags hinzu, um Ihre Kapazitätsanbieter zu organisieren und zu verwalten.

Verwaltung von Kapazitätsanbietern

Aktualisieren eines Kapazitätsanbieters

Sie können bestimmte Eigenschaften eines Kapazitätsanbieters mithilfe der `UpdateCapacityProvider` API aktualisieren.

```
aws lambda update-capacity-provider \
  --capacity-provider-name my-capacity-provider \
  --capacity-provider-scaling-config ScalingMode=Auto
```

Löschen eines Kapazitätsanbieters

Mithilfe der `DeleteCapacityProvider` API können Sie einen Kapazitätsanbieter löschen, wenn er nicht mehr benötigt wird.

```
aws lambda delete-capacity-provider \
  --capacity-provider-name my-capacity-provider
```

Hinweis: Sie können keinen Kapazitätsanbieter löschen, an den Funktionsversionen angehängt sind.

Details zum Kapazitätsanbieter anzeigen

Rufen Sie mithilfe der `GetCapacityProvider` API Informationen über einen Kapazitätsanbieter ab.

```
aws lambda get-capacity-provider \
  --capacity-provider-name my-capacity-provider
```

Staaten des Kapazitätsanbieters

Ein Kapazitätsanbieter kann sich in einem der folgenden Staaten befinden:

- Ausstehend: Der Kapazitätsanbieter wird gerade erstellt
- Aktiv: Der Kapazitätsanbieter ist einsatzbereit
- Fehlgeschlagen: Die Erstellung des Kapazitätsanbieters ist fehlgeschlagen
- Löschen: Der Kapazitätsanbieter wird gelöscht

Kontingente

- Maximale Kapazitätsanbieter pro Konto: 1.000

- Maximale Funktionsversionen pro Kapazitätsanbieter: 100 (kann nicht erhöht werden)

Best Practices

1. Nach Vertrauensstufe trennen: Erstellen Sie verschiedene Kapazitätsanbieter für Workloads mit unterschiedlichen Sicherheitsanforderungen
2. Verwenden Sie aussagekräftige Namen: Benennen Sie Kapazitätsanbieter, um deren Verwendungszweck und Vertrauensstufe eindeutig anzugeben (z. B.,,`production-trusted`) `dev-sandbox`
3. Verwenden Sie mehrere Availability Zones: Geben Sie Subnetze über mehrere an, um eine hohe Verfügbarkeit AZs zu gewährleisten
4. Lassen Sie Lambda Instance-Typen wählen: Sofern Sie keine spezifischen Hardwareanforderungen haben, lassen Sie Lambda die besten Instance-Typen für optimale Verfügbarkeit auswählen
5. Nutzung überwachen: Wird verwendet AWS CloudTrail , um die Zuweisungen und Zugriffsmuster von Kapazitätsanbietern zu überwachen

Nächste Schritte

- Erfahren Sie mehr über die Skalierung von Lambda Managed Instances
- Sicherheit und Berechtigungen für Lambda Managed Instances verstehen
- Konfigurieren Sie die VPC-Konnektivität für Ihre Kapazitätsanbieter
- Lesen Sie laufzeitspezifische Anleitungen für Java, Node.js und Python

Skalierung von verwalteten Lambda-Instanzen

Lambda Managed Instances skaliert nicht, wenn Aufrufe eintreffen, und unterstützt keine Kaltstarts. Stattdessen wird es asynchron mithilfe von Signalen zum Ressourcenverbrauch skaliert. Managed Instances wird derzeit auf der Grundlage der CPU-Ressourcenauslastung und der Auslastung mehrerer Parallelitäten skaliert.

Die wichtigsten Unterschiede:

- Lambda (Standard): Skaliert, wenn es keine freie Ausführungsumgebung für die Verarbeitung eines eingehenden Aufrufs gibt (Kaltstart)

- Lambda Managed Instances: Skaliert asynchron auf der Grundlage der CPU-Ressourcenauslastung und der Mehrparallelität von Ausführungsumgebungen

Wenn sich Ihr Traffic innerhalb von 5 Minuten mehr als verdoppelt, kann es zu Drosselungen kommen, wenn Lambda Instances und Ausführungsumgebungen entsprechend der Nachfrage skaliert.

Der Skalierungslebenszyklus

Lambda Managed Instances verwendet eine verteilte Architektur, um die Skalierung zu verwalten:

Komponenten:

- Verwaltete Instanzen — Werden in Ihrem Konto in den von Ihnen bereitgestellten Subnetzen ausgeführt
- Router and Scaler — Gemeinsam genutzte Lambda-Komponenten, die Aufrufe weiterleiten und die Skalierung verwalten
- Lambda Agent — Wird auf jeder verwalteten Instanz ausgeführt, um den Lebenszyklus der Ausführungsumgebung zu verwalten und den Ressourcenverbrauch zu überwachen

So funktioniert es:

1. Wenn Sie eine Funktionsversion mit einem Kapazitätsanbieter veröffentlichen, startet Lambda Managed Instances in Ihrem Konto. Aus Gründen der AZ-Resilienz werden standardmäßig drei und drei Ausführungsumgebungen gestartet, bevor Ihre Funktionsversion als AKTIV markiert wird.
2. Jede verwaltete Instanz kann Ausführungsumgebungen für mehrere Funktionen ausführen, die demselben Kapazitätsanbieter zugeordnet sind.
3. Wenn der Datenverkehr in Ihre Anwendung fließt, verbrauchen Ausführungsumgebungen Ressourcen. Der Lambda-Agent benachrichtigt den Scaler, der entscheidet, ob neue Ausführungsumgebungen oder verwaltete Instances skaliert werden sollen.
4. Wenn der Router versucht, einen Aufruf an eine Ausführungsumgebung mit hohem Ressourcenverbrauch zu senden, benachrichtigt ihn der Lambda-Agent auf dieser Instanz, es auf einer anderen erneut zu versuchen.
5. Wenn der Traffic abnimmt, benachrichtigt der Lambda Agent den Scaler, der entscheidet, die Ausführungsumgebungen herunterzufahren und in verwalteten Instanzen zu skalieren.

Anpassung des Skalierungsverhaltens

Sie können das Skalierungsverhalten von Managed Instances mithilfe von vier Steuerelementen anpassen:

Steuerungen auf Funktionsebene

1. Funktionsspeicher und v CPUs

Wählen Sie die Speichergröße und die vCPU-Zuweisung für Ihre Funktion. Die kleinste unterstützte Funktionsgröße ist 2 GB und 1 vCPU.

Überlegungen:

- Wählen Sie eine Speicher- und vCPU-Einstellung, die mehrere gleichzeitige Ausführungen Ihrer Funktion unterstützt
- Sie können keine Funktion mit weniger als 1 vCPU konfigurieren, da Funktionen, die auf verwalteten Instances ausgeführt werden, mehrere gleichzeitige Workloads unterstützen sollten
- Sie können nicht weniger als 2 GB wählen, da dies dem Verhältnis von 2 zu 1 Arbeitsspeicher zu vCPU von c Instances entspricht, die das niedrigste Verhältnis aufweisen.
- Für Python-Anwendungen müssen Sie möglicherweise ein höheres Verhältnis von Speicher zu V wählenCPUs, z. B. 4 zu 1 oder 8 zu 1, da Python Multiparallelität handhabt
- Wenn Sie CPU-intensive Operationen ausführen oder wenig I/O ausführen, sollten Sie mehr als eine vCPU wählen

2. Maximale Parallelität

Legen Sie die maximale Parallelität pro Ausführungsumgebung fest.

Standardverhalten: Lambda wählt sinnvolle Standardwerte, die den Ressourcenverbrauch und den Durchsatz in Einklang bringen und für eine Vielzahl von Anwendungen geeignet sind.

Richtlinien für Anpassungen:

- Erhöhen Sie die Parallelität: Wenn Ihre Funktionsaufrufen nur sehr wenig CPU verbrauchen, können Sie die maximale Parallelität auf maximal 64 pro vCPU erhöhen
- Gleichzeitigkeit verringern: Wenn Ihre Anwendung viel Arbeitsspeicher und sehr wenig CPU beansprucht, können Sie die maximale Parallelität reduzieren

Wichtig: Da Lambda Managed Instances für Anwendungen mit mehreren gleichzeitigen Vorgängen konzipiert sind, kann es in Ausführungsumgebungen mit sehr geringer Parallelität zu Drosselungen bei der Skalierung kommen.

Kontrollen auf Ebene des Kapazitätsanbieters

3. Nutzung der Zielressourcen

Wählen Sie Ihr eigenes Ziel für den CPU-Nutzungsverbrauch.

Standardverhalten: Lambda behält genügend Headroom bei, sodass sich Ihr Traffic ohne Drosselungen innerhalb von 5 Minuten verdoppeln kann.

Optimierungsoptionen:

- Wenn Ihre Arbeitslast sehr konstant ist oder wenn Ihre Anwendung nicht empfindlich auf Drosselungen reagiert, können Sie das Ziel auf ein hohes Niveau setzen, um eine höhere Auslastung und niedrigere Kosten zu erreichen
- Wenn Sie genügend Spielraum für hohe Datenmengen haben möchten, können Sie die Ressourcenziele auf ein niedriges Niveau setzen, was wiederum mehr Kapazität erfordert

4. Auswahl des Instance-Typs

Legen Sie zulässige oder ausgeschlossene Instance-Typen fest.

Standardverhalten: Lambda wählt die besten Instance-Typen für Ihren Workload aus. Wir empfehlen, Lambda Managed Instances die Instanztypen für Sie auswählen zu lassen, da eine Einschränkung der Anzahl möglicher Instanztypen zu einer geringeren Verfügbarkeit führen kann.

Benutzerdefinierte Konfiguration:

- Spezifische Hardwareanforderungen: Legen Sie für zulässige Instanztypen eine Liste kompatibler Instanzen fest. Wenn Sie beispielsweise eine Anwendung haben, die eine hohe Netzwerkbandbreite benötigt, können Sie mehrere n-Instance-Typen auswählen
- Kostenoptimierung: Für Test- oder Entwicklungsumgebungen können Sie kleinere Instance-Typen wie m7a.large-Instance-Typen wählen

Nächste Schritte

- Erfahren Sie mehr über Kapazitätsanbieter für Lambda Managed Instances

- Lesen Sie die laufzeitspezifischen Anleitungen zum Umgang mit Mehrfachparallelität
- Konfigurieren Sie die VPC-Konnektivität für Ihre Kapazitätsanbieter
- Überwachen Sie Skalierungskennzahlen, um das Skalierungsverhalten zu optimieren

Sicherheit und Berechtigungen

Lambda Managed Instances verwenden Kapazitätsanbieter als Vertrauensgrenzen. Funktionen werden in Containern innerhalb dieser Instanzen ausgeführt, aber Container bieten keine Sicherheitsisolierung zwischen Workloads. Alle Funktionen, die demselben Kapazitätsanbieter zugewiesen sind, müssen gegenseitig vertrauenswürdig sein.

Wichtige Sicherheitskonzepte

- Kapazitätsanbieter: Die Sicherheitsgrenze, die Vertrauensstufen für Lambda-Funktionen definiert
- Container-Isolierung: Container stellen keine Sicherheitsgrenze dar. Verlassen Sie sich nicht auf sie, wenn es um die Sicherheit zwischen nicht vertrauenswürdigen Workloads geht
- Vertrauenstrennung: Trennen Sie Workloads, denen nicht gegenseitig vertraut wird, indem Sie unterschiedliche Kapazitätsanbieter verwenden

Erforderliche Berechtigungen

PassCapacityProvider Aktion

Benutzer benötigen die `lambda:PassCapacityProvider` Erlaubnis, Kapazitätsanbietern Funktionen zuzuweisen. Diese Genehmigung dient als Sicherheitsschleuse und stellt sicher, dass nur autorisierte Benutzer Funktionen bestimmten Kapazitätsanbietern zuweisen können.

Kontoadministratoren kontrollieren über die `lambda:PassCapacityProvider` IAM-Aktion, welche Funktionen bestimmte Kapazitätsanbieter nutzen können. Diese Aktion ist erforderlich, wenn:

- Funktionen erstellen, die Lambda Managed Instances verwenden
- Aktualisierung von Funktionskonfigurationen zur Verwendung eines Kapazitätsanbieters
- Bereitstellung von Funktionen über die Infrastruktur als Code

Beispiel für eine IAM-Richtlinie

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": "lambda:PassCapacityProvider",
      "Resource": "arn:aws:lambda:*:*:capacity-provider:trusted-workloads-*"
    }
  ]
}
```

Serviceverknüpfte Rolle

AWS Lambda verwendet die `AWSServiceRoleForLambda` serviceverknüpfte Rolle, um die ec2-Ressourcen von Lambda Managed Instances bei Ihren Kapazitätsanbietern zu verwalten.

Bewährte Methoden

1. Nach Vertrauensstufe trennen: Erstellen Sie verschiedene Kapazitätsanbieter für Workloads mit unterschiedlichen Sicherheitsanforderungen
2. Verwenden Sie aussagekräftige Namen: Benennen Sie Kapazitätsanbieter, um deren Verwendungszweck und Vertrauensstufe eindeutig anzugeben (z. B.,,`production-trusted`) `dev-sandbox`
3. Least Privilege anwenden: Erteilen Sie `PassCapacityProvider` Berechtigungen nur für die erforderlichen Kapazitätsanbieter
4. Nutzung überwachen: Wird verwendet AWS CloudTrail , um die Zuweisungen und Zugriffsmuster von Kapazitätsanbietern zu überwachen

Nächste Schritte

- Erfahren Sie mehr über Kapazitätsanbieter für Lambda Managed Instances
- Skalierung für Lambda Managed Instances verstehen
- Konfigurieren Sie die VPC-Konnektivität für Ihre Kapazitätsanbieter
- Lesen Sie laufzeitspezifische Anleitungen für Java, Node.js und Python

Lambda-Operatorrolle für Lambda Managed Instances

Wenn Sie Lambda Managed Instances verwenden, benötigt Lambda Berechtigungen zur Verwaltung der Rechenkapazität in Ihrem Konto. Die Operatorrolle stellt diese Berechtigungen über IAM-Richtlinien bereit, die es Lambda ermöglichen, EC2 Instanzen im Kapazitätsanbieter zu verwalten.

Lambda übernimmt bei der Ausführung dieser Verwaltungsvorgänge die Operatorrolle, ähnlich wie Lambda eine Ausführungsrolle übernimmt, wenn Ihre Funktion ausgeführt wird.

Eine Operatorrolle erstellen

Sie können eine Operatorrolle in der IAM-Konsole oder mit der AWS CLI erstellen. Die Rolle muss Folgendes beinhalten:

- Berechtigungsrichtlinie — Erteilt Berechtigungen zur Verwaltung von Kapazitätsanbietern und zugehörigen Ressourcen
- Vertrauensrichtlinie — Ermöglicht dem Lambda-Dienst (`lambda.amazonaws.com`), die Rolle zu übernehmen

Berechtigungsrichtlinie

Die Operatorrolle benötigt Berechtigungen zur Verwaltung von Kapazitätsanbietern und den zugrunde liegenden Rechenressourcen. Für die Rolle sind mindestens die Berechtigungen erforderlich, die in der AWSLambdaEC2ResourceOperatorverwalteten Richtlinie enthalten sind. Derzeit gilt Folgendes:

```
{
  "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"
        }
      }
```

```
    }
  ]
}
```

Vertrauensrichtlinie

Die Vertrauensrichtlinie ermöglicht es Lambda, die Betreiberrolle zu übernehmen:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "lambda.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
```

Serviceverknüpfte Rolle für verwaltete Lambda-Instances

Um den Lebenszyklus von Lambda Managed Instances verantwortungsbewusst zu verwalten, benötigt Lambda dauerhaften Zugriff, um verwaltete Instanzen in Ihrem Konto zu beenden. Lambda verwendet eine dienstgebundene Rolle (SLR) für AWS Identity and Access Management (IAM), um diese Operationen auszuführen.

Automatische Erstellung: Die serviceverknüpfte Rolle wird automatisch erstellt, wenn Sie zum ersten Mal einen Kapazitätsanbieter erstellen. Der Benutzer, der den ersten Kapazitätsanbieter erstellt, muss über die `iam:CreateServiceLinkedRole` Berechtigung für den `lambda.amazonaws.com` Prinzipal verfügen.

Berechtigungen: Die serviceverknüpfte Rolle gewährt Lambda die folgenden Berechtigungen für verwaltete Instanzen:

- `ec2:TerminateInstances`— Um Instanzen am Ende ihres Lebenszyklus zu beenden
- `ec2:DescribeInstances`— Um verwaltete Instanzen aufzuzählen

Löschung: Sie können diese serviceverknüpfte Rolle erst löschen, nachdem Sie alle Kapazitätsanbieter für Lambda Managed Instances in Ihrem Konto gelöscht haben.

Weitere Informationen zu serviceverknüpften Rollen finden Sie unter the section called “Verwenden von servicegebundenen Rollen”.

Grundlegendes zur Ausführungsumgebung von Lambda Managed Instances

Lambda Managed Instances bieten ein alternatives Bereitstellungsmodell, bei dem Ihr Funktionscode auf kundeneigenen EC2 Amazon-Instances ausgeführt wird, während Lambda die betrieblichen Aspekte verwaltet. Die Ausführungsumgebung für Managed Instances weist mehrere wichtige Unterschiede zu Lambda-Funktionen (Standard) auf, insbesondere darin, wie sie gleichzeitige Aufrufe verarbeitet und Container-Lebenszyklen verwaltet.

Hinweis: Informationen zur Lambda-Ausführungsumgebung (Standard) finden Sie unter Grundlegendes zum Lebenszyklus der Lambda-Ausführungsumgebung.

Lebenszyklus der Ausführungsumgebung

Der Lebenszyklus einer Funktionsausführungsumgebung für Lambda Managed Instances unterscheidet sich in mehreren wesentlichen Punkten von Lambda (Standard):

Init-Phase

Während der Init-Phase führt Lambda die folgenden Schritte aus:

- Initialisieren und registrieren Sie alle Erweiterungen
- Bootstrap für den Laufzeit-Einstiegspunkt. Runtime erzeugt die konfigurierte Anzahl von Runtime-Workern (die Implementierung hängt von der Laufzeit ab)
- Führen Sie den Initialisierungscode der Funktion aus (Code außerhalb des Handlers)
- Warten Sie, bis mindestens ein Runtime-Worker die Bereitschaft signalisiert, indem er aufruft `/runtime/invocation/next`

Die Init-Phase gilt als abgeschlossen, wenn die Erweiterungen initialisiert wurden und mindestens ein Runtime-Worker aufgerufen hat. `/runtime/invocation/next` Die Funktion ist dann bereit, Aufrufe zu verarbeiten.

> **Note**
>
> Für Funktionen von Lambda Managed Instances kann die Initialisierung bis zu 15 Minuten dauern. Das Timeout ist das Maximum von 130 Sekunden oder das konfigurierte Funktions-Timeout (bis zu 900 Sekunden).

Invoke-Phase

Die Aufrufphase für Funktionen von Lambda Managed Instances weist mehrere einzigartige Merkmale auf:

Kontinuierlicher Betrieb. Im Gegensatz zu Lambda (Standard) bleibt die Ausführungsumgebung kontinuierlich aktiv und verarbeitet eingehende Aufrufe, ohne dass sie zwischen den Aufrufen einfrieren.

Parallele Verarbeitung. In derselben Ausführungsumgebung können mehrere Aufrufe gleichzeitig ausgeführt werden, wobei jeder Aufruf von einem anderen Runtime-Worker verarbeitet wird.

Unabhängige Timeouts. Das konfigurierte Timeout der Funktion gilt für jeden einzelnen Aufruf. Wenn bei einem Aufruf eine Zeitüberschreitung eintritt, markiert Lambda diesen bestimmten Aufruf als fehlgeschlagen, unterbricht jedoch keine anderen laufenden Aufrufe und beendet die Ausführungsumgebung nicht.

Behandlung von Gegendruck. Wenn alle Runtime-Worker mit der Verarbeitung von Aufrufen beschäftigt sind, werden neue Aufrufanforderungen zurückgewiesen, bis ein Worker verfügbar ist.

Fehlerbehandlung und Wiederherstellung

Die Fehlerbehandlung in Umgebungen zur Funktionsausführung von Lambda Managed Instances unterscheidet sich von Lambda (Standard):

Timeouts aufrufen. Wenn bei einem einzelnen Aufruf das Timeout überschritten wird, gibt Lambda einen Timeout-Fehler für diesen Aufruf zurück. Lambda Managed Instances erzwingt das Timeout jedoch nicht — Ihr Code wird weiter ausgeführt. Als Funktionsentwickler sind Sie dafür verantwortlich, das Timeout zu erkennen und zu behandeln. Das Kontextobjekt gibt die verbleibende Zeit für den Aufruf an, wobei ein Nullwert oder ein negativer Wert auf ein Timeout hinweist. Andere gleichzeitige Aufrufe in der Ausführungsumgebung setzen die Verarbeitung normal fort.

Fehler beim Runtime-Worker. Wenn ein Runtime-Worker-Prozess abstürzt, arbeitet die Ausführungsumgebung mit den verbleibenden fehlerfreien Workern weiter.

Die Erweiterung stürzt ab. Wenn ein Erweiterungsprozess während der Initialisierung oder des Betriebs abstürzt, wird die gesamte Ausführungsumgebung als fehlerhaft markiert und beendet. Lambda erstellt eine neue Ausführungsumgebung, um sie zu ersetzen.

Kein Reset/Reparieren. Im Gegensatz zu Lambda (Standard) versuchen Managed Instances nicht, die Ausführungsumgebung nach Fehlern zurückzusetzen und neu zu initialisieren. Stattdessen werden fehlerhafte Container beendet und durch neue ersetzt.

$LATEST.PUBLISHED Version in Lambda Managed Instances

Die Funktionen von Lambda Managed Instances unterstützen denselben nummerierten Versionierungsworkflow wie Lambda (Standard). Wenn Sie es vorziehen, keine nummerierten Versionen zu verwalten, führt Lambda Managed Instances einen neuen Versionstyp ein:`$LATEST.PUBLISHED`. Mit dieser Version können Sie nach Bedarf eine neueste veröffentlichte Version mit aktualisiertem Code oder aktualisierter Konfiguration erstellen oder erneut veröffentlichen, ohne nummerierte Versionen verwalten zu müssen.

Hauptunterschied zu $LATEST: Wenn Sie eine Lambda Managed Instances-Funktion mit einem unqualifizierten ARN aufrufen, ruft Lambda implizit die Version und nicht die unveröffentlichte `$LATEST.PUBLISHED` $LATEST-Version auf.

Der folgende AWS CLI-Befehl erstellt oder veröffentlicht die `$LATEST.PUBLISHED` Version erneut.

```
aws lambda publish-version --function-name my-function --publish-to LATEST_PUBLISHED
```

Die Ausgabe sollte folgendermaßen aussehen:

```
{
  "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**
>
> Wenn Sie die Lambda-Konsole verwenden AWS CloudFormation , um eine Lambda Managed Instances-Funktion zu erstellen, erstellt Lambda die Version automatisch. `$LATEST.PUBLISHED`

Laufzeiten von Lambda Managed Instances

Lambda verarbeitet Anfragen anders, wenn Lambda Managed Instances verwendet werden. Anstatt Anfragen nacheinander in jeder Ausführungsumgebung zu bearbeiten, verarbeiten Lambda Managed Instances mehrere Anfragen gleichzeitig in jeder Ausführungsumgebung. Diese Änderung des Ausführungsmodells bedeutet, dass Funktionen, die Lambda Managed Instances verwenden, Thread-Sicherheit, Statusverwaltung und Kontextisolation berücksichtigen müssen — Aspekte, die im Lambda-Modell (Standard) mit nur einer Parallelität nicht auftreten. Darüber hinaus variiert die Implementierung mehrerer Parallelitäten je nach Laufzeit.

Unterstützte Laufzeiten

Die folgenden Laufzeiten unterstützen Lambda Managed Instances:

- Java: Java 21 und höher.
- Python: Python 3.13 und höher.
- Node.js: Node.js 22 und höher.
- .NET: .NET 8 und höher.

Laufzeitspezifische Überlegungen

Jede Laufzeit implementiert Multiparallelität auf unterschiedliche Weise. Sie müssen verstehen, wie Multiparallelität in der von Ihnen gewählten Programmiersprache implementiert wird, um die entsprechenden Best Practices für Parallelität anwenden zu können.

Java

Verwendet einen einzigen Prozess mit Betriebssystem-Threads für Parallelität. Mehrere Threads führen die Handler-Methode gleichzeitig aus, was eine threadsichere Behandlung von Status- und gemeinsam genutzten Ressourcen erfordert.

Python

Verwendet mehrere Python-Prozesse, bei denen jede gleichzeitige Anforderung in einem separaten Prozess ausgeführt wird. Dies schützt vor den meisten Parallelitätsproblemen, obwohl bei gemeinsam genutzten Ressourcen wie dem `/tmp` Verzeichnis Vorsicht geboten ist.

Node.js

Verwendet Worker-Threads mit asynchroner Ausführung. Gleichzeitige Anfragen werden über Worker-Threads verteilt, und jeder Worker-Thread kann gleichzeitige Anfragen auch asynchron verarbeiten, was eine sichere Handhabung von Status- und gemeinsam genutzten Ressourcen erfordert.

.NET

Verwendet .NET-Aufgaben mit asynchroner Verarbeitung mehrerer gleichzeitiger Anfragen. Erfordert den sicheren Umgang mit staatlichen und gemeinsam genutzten Ressourcen.

Nächste Schritte

Ausführliche Informationen zu den einzelnen Laufzeitumgebungen finden Sie in den folgenden Themen:

- Java-Laufzeit für Lambda Managed Instances
- Laufzeit von Node.js für verwaltete Lambda-Instanzen
- Python-Laufzeit für verwaltete Lambda-Instanzen
- .NET-Laufzeit für verwaltete Lambda-Instanzen

Java-Laufzeit für Lambda Managed Instances

Für Java-Laufzeiten verwenden Lambda Managed Instances Betriebssystem-Threads für Parallelität. Lambda lädt Ihr Handler-Objekt während der Initialisierung einmal pro Ausführungsumgebung und erstellt dann mehrere Threads. Diese Threads werden parallel ausgeführt und erfordern eine threadsichere Behandlung von Status- und gemeinsam genutzten Ressourcen. Jeder Thread verwendet dasselbe Handler-Objekt und alle statischen Felder.

Konfiguration der Parallelität

Die maximale Anzahl gleichzeitiger Anfragen, die Lambda an jede Ausführungsumgebung sendet, wird durch die `PerExecutionEnvironmentMaxConcurrency` Einstellung in der

Funktionskonfiguration gesteuert. Dies ist eine optionale Einstellung, und der Standardwert variiert je nach Laufzeit. Für Java-Laufzeiten ist die Standardeinstellung 32 gleichzeitige Anfragen pro vCPU, oder Sie können Ihren eigenen Wert konfigurieren. Dieser Wert bestimmt auch die Anzahl der Threads, die von der Java-Laufzeit verwendet werden. Lambda passt die Anzahl der gleichzeitigen Anfragen automatisch bis zum konfigurierten Maximum an, basierend auf der Kapazität jeder Ausführungsumgebung, um diese Anfragen aufzunehmen.

Erstellung von Funktionen für Mehrfachparallelität

Sie sollten bei der Verwendung von Lambda Managed Instances dieselben Thread-Sicherheitspraktiken anwenden wie in jeder anderen Multithread-Umgebung. Da das Handler-Objekt von allen Runtime-Worker-Threads gemeinsam genutzt wird, muss jeder veränderbare Status threadsicher sein. Dazu gehören Sammlungen, Datenbankverbindungen und alle statischen Objekte, die während der Anforderungsverarbeitung geändert werden.

AWS SDK-Clients sind Thread-sicher und erfordern keine besondere Behandlung.

Beispiel: Datenbank-Verbindungspools

Der folgende Code verwendet ein statisches Datenbankverbindungsobjekt, das von Threads gemeinsam genutzt wird. Abhängig von der verwendeten Verbindungsbibliothek ist dies möglicherweise nicht threadsicher.

```
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();
    }
}
```

Ein threadsicherer Ansatz besteht darin, einen Verbindungspool zu verwenden. Im folgenden Beispiel ruft der Funktionshandler eine Verbindung aus dem Pool ab. Die Verbindung wird nur im Kontext einer einzelnen Anfrage verwendet.

```
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();
    }
}
```

Beispiel: Sammlungen

Standard-Java-Sammlungen sind nicht threadsicher:

```
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";
    }
}
```

Verwenden Sie stattdessen Thread-sichere Sammlungen:

```
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";
    }
}
```

Gemeinsames /tmp-Verzeichnis

Das `/tmp` Verzeichnis wird von allen gleichzeitigen Anfragen in der Ausführungsumgebung gemeinsam genutzt. Gleichzeitige Schreibvorgänge in dieselbe Datei können zu Datenbeschädigungen führen, z. B. wenn ein anderer Prozess die Datei überschreibt. Um dieses Problem zu lösen, implementieren Sie entweder Dateisperren für gemeinsam genutzte Dateien oder verwenden Sie eindeutige Dateinamen pro Thread oder pro Anfrage, um Konflikte zu vermeiden. Denken Sie daran, nicht benötigte Dateien zu bereinigen, um zu vermeiden, dass der verfügbare Speicherplatz aufgebraucht wird.

Protokollierung

Das Verschachteln von Protokollen (Protokolleinträge verschiedener Anfragen werden in Protokollen verschachtelt) ist bei Systemen mit mehreren gleichzeitigen Vorgängen normal.

Funktionen, die Lambda Managed Instances verwenden, verwenden immer das strukturierte JSON-Protokollformat, das mit erweiterten Protokollierungssteuerungen eingeführt wurde. Dieses Format beinhaltet die`requestId`, sodass Protokolleinträge mit einer einzigen Anfrage korreliert werden können. Wenn Sie das `LambdaLogger` Objekt aus `context.getLogger()` verwenden, `requestId` ist es automatisch in jedem Protokolleintrag enthalten. Weitere Informationen finden Sie unterthe section called “Verwenden von Lambda-Optionen für die erweiterte Protokollierung mit Java”.

Kontext anfordern

Das `context` Objekt ist an den Anforderungsthread gebunden. `context.getAwsRequestId()`Die Verwendung ermöglicht einen threadsicheren Zugriff auf die Anforderungs-ID für die aktuelle Anfrage.

Wird verwendet`context.getXrayTraceId()`, um auf die X-Ray-Trace-ID zuzugreifen. Dies ermöglicht einen threadsicheren Zugriff auf die Trace-ID für die aktuelle Anfrage. Lambda unterstützt die `_X_AMZN_TRACE_ID` Umgebungsvariable bei Lambda Managed Instances nicht. Die X-Ray-Trace-ID wird bei Verwendung des AWS SDK automatisch weitergegeben.

Wird verwendet`com.amazonaws.services.lambda.runtime.Context.getRemainingTimeInMillis()`, um Timeouts zu erkennen. Weitere Informationen finden Sie unter the section called “Fehlerbehandlung und Wiederherstellung”.

Wenn Sie in Ihrem Programm virtuelle Threads verwenden oder während der Initialisierung Threads erstellen, müssen Sie den erforderlichen Anforderungskontext an diese Threads übergeben.

Initialisierung und Herunterfahren

Die Funktionsinitialisierung erfolgt einmal pro Ausführungsumgebung. Objekte, die während der Initialisierung erstellt wurden, werden von allen Threads gemeinsam genutzt.

Bei Lambda-Funktionen mit Erweiterungen gibt die Ausführungsumgebung beim Herunterfahren ein SIGTERM-Signal aus. Dieses Signal wird von Erweiterungen verwendet, um Bereinigungsaufgaben wie das Leeren von Puffern auszulösen. Sie können SIGTERM-Ereignisse abonnieren, um Aufgaben

zur Funktionsbereinigung auszulösen, z. B. das Schließen von Datenbankverbindungen. Weitere Informationen zum Lebenszyklus der Ausführungsumgebung finden Sie unter the section called “Ausführungsumgebung”.

Versionen von Abhängigkeiten

Lambda Managed Instances erfordert die folgenden Mindestpaketversionen:

- AWS SDK for Java 2.0: Version 2.34.0 oder höher
- AWS X-Ray SDK for Java: Version 2.20.0 oder höher
- AWS Distribution für OpenTelemetry — Instrumentation für Java: Version 2.20.0 oder höher
- Powertools für AWS Lambda (Java): Version 2.8.0 oder höher

Powertools für AWS Lambda (Java)

Powertools for AWS Lambda (Java) ist mit Lambda Managed Instances kompatibel und bietet Dienstprogramme für Logging, Tracing, Metriken und mehr. Weitere Informationen finden Sie unter Powertools for AWS Lambda (Java).

Nächste Schritte

- Überprüfen Sie die Laufzeit von Node.js für Lambda Managed Instances
- Überprüfen Sie die Python-Laufzeit für Lambda Managed Instances
- .NET-Laufzeit für Lambda Managed Instances überprüfen
- Erfahren Sie mehr über die Skalierung von Lambda Managed Instances

Laufzeit von Node.js für verwaltete Lambda-Instanzen

Für Laufzeiten von Node.js verwendet Lambda Managed Instances Worker-Threads mit`async/await`-basierter Ausführung, um gleichzeitige Anfragen zu verarbeiten. Die Funktionsinitialisierung erfolgt einmal pro Worker-Thread. Gleichzeitige Aufrufe werden in zwei Dimensionen verarbeitet: Worker-Threads sorgen für Parallelität in VCPUs, und asynchrone Ausführung sorgt für Parallelität innerhalb jedes Threads. Jede gleichzeitige Anforderung, die von demselben Worker-Thread bearbeitet wird, verwendet dasselbe Handler-Objekt und denselben globalen Status, sodass eine sichere Behandlung bei mehreren gleichzeitigen Anfragen erforderlich ist.

Maximale Parallelität

Die maximale Anzahl gleichzeitiger Anfragen, die Lambda an jede Ausführungsumgebung sendet, wird durch die `PerExecutionEnvironmentMaxConcurrency` Einstellung in der Funktionskonfiguration gesteuert. Dies ist eine optionale Einstellung, und der Standardwert variiert je nach Laufzeit. Für Laufzeiten von Node.js sind standardmäßig 64 gleichzeitige Anfragen pro vCPU festgelegt, oder Sie können Ihren eigenen Wert konfigurieren. Lambda passt die Anzahl der gleichzeitigen Anfragen automatisch bis zum konfigurierten Maximum an, basierend auf der Kapazität jeder Ausführungsumgebung, um diese Anfragen aufzunehmen.

Für Node.js wird die Anzahl der gleichzeitigen Anfragen, die jede Ausführungsumgebung verarbeiten kann, durch die Anzahl der Worker-Threads und die Kapazität jedes Worker-Threads bestimmt, gleichzeitige Anfragen asynchron zu verarbeiten. Die Standardanzahl von Worker-Threads wird durch die Anzahl der CPUs verfügbaren V bestimmt. Sie können die Anzahl der Worker-Threads auch konfigurieren, indem Sie die `AWS_LAMBDA_NODEJS_WORKER_COUNT` Umgebungsvariable festlegen. Wir empfehlen die Verwendung von asynchronen Funktionshandlern, da dies die Verarbeitung mehrerer Anfragen pro Worker-Thread ermöglicht. Wenn Ihr Funktionshandler synchron ist, kann jeder Worker-Thread jeweils nur eine einzelne Anfrage verarbeiten.

Funktionen für Mehrfachparallelität erstellen

Mit einem asynchronen Funktionshandler verarbeitet jeder Runtime-Worker mehrere Anfragen gleichzeitig. Globale Objekte werden von mehreren gleichzeitigen Anfragen gemeinsam genutzt. Vermeiden Sie bei veränderlichen Objekten die Verwendung von global state or use. `AsyncLocalStorage`

AWS SDK-Clients sind asynchron-sicher und erfordern keine besondere Behandlung.

Beispiel: Globaler Status

Der folgende Code verwendet ein globales Objekt, das innerhalb des Funktionshandlers mutiert ist. Das ist nicht asynchron-sicher.

```
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 };
};
```

Durch die Initialisierung des `state` Objekts innerhalb des Funktionshandlers wird ein gemeinsamer globaler Status vermieden.

```
export const handler = async (event, context) => {
    let state = {
        currentUser: event.userId,
        requestData: event.data
    };

    await processData(state.requestData);

    return { user: state.currentUser };
};
```

Beispiel: Datenbankverbindungen

Der folgende Code verwendet ein gemeinsames Client-Objekt, das von mehreren Aufrufen gemeinsam genutzt wird. Je nach verwendeter Verbindungsbibliothek ist dies möglicherweise nicht sicher für Parallelität.

```
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])
  };
};
```

Ein Ansatz, der die Parallelität sicherstellt, besteht darin, einen Verbindungspool zu verwenden. Der Pool verwendet für jede gleichzeitige Datenbankabfrage eine separate Verbindung.

```
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 Callback-basierte Handler

Wenn Sie Node.js 22 verwenden, können Sie keinen Callback-basierten Funktionshandler mit Lambda Managed Instances verwenden. Callback-basierte Handler werden nur für Lambda-

Funktionen (Standard) unterstützt. Für Laufzeiten von Node.js 24 und höher sind Callback-basierte Funktionshandler sowohl für Lambda (Standard) als auch für Lambda Managed Instances veraltet.

Verwenden Sie stattdessen einen `async` Funktionshandler, wenn Sie Lambda Managed Instances verwenden. Weitere Informationen finden Sie unter Definieren des Lambda-Funktionshandlers in Node.js.

Gemeinsames /tmp-Verzeichnis

Das `/tmp` Verzeichnis wird von allen gleichzeitigen Anfragen in der Ausführungsumgebung gemeinsam genutzt. Gleichzeitige Schreibvorgänge in dieselbe Datei können zu Datenbeschädigungen führen, z. B. wenn ein anderer Prozess die Datei überschreibt. Um dieses Problem zu lösen, implementieren Sie entweder Dateisperren für gemeinsam genutzte Dateien oder verwenden Sie eindeutige Dateinamen pro Anfrage, um Konflikte zu vermeiden. Denken Sie daran, nicht benötigte Dateien zu bereinigen, um zu vermeiden, dass der verfügbare Speicherplatz aufgebraucht wird.

Protokollierung

Das Verschachteln von Protokollen (Protokolleinträge verschiedener Anfragen, die in Protokollen verschachtelt werden) ist bei Systemen mit mehreren gleichzeitigen Vorgängen normal. Funktionen, die Lambda Managed Instances verwenden, verwenden immer das strukturierte JSON-Protokollformat, das mit erweiterten Protokollierungssteuerungen eingeführt wurde. Dieses Format beinhaltet die`requestId`, sodass Protokolleinträge einer einzelnen Anfrage zugeordnet werden können. Wenn Sie den `console` Logger verwenden, `requestId` ist er automatisch in jedem Protokolleintrag enthalten. Weitere Informationen finden Sie unterthe section called “Verwenden von Lambda-Optionen für die erweiterte Protokollierung mit Node.js”.

Beliebte Protokollierungsbibliotheken von Drittanbietern, wie Winston, unterstützen in der Regel die Verwendung der Konsole für die Protokollausgabe.

Kontext anfordern

`context.awsRequestId`Die Verwendung ermöglicht einen asynchronen Zugriff auf die Anforderungs-ID für die aktuelle Anfrage.

Wird verwendet`context.xRayTraceId`, um auf die X-Ray-Trace-ID zuzugreifen. Dies ermöglicht einen parallelen Zugriff auf die Trace-ID für die aktuelle Anfrage. Lambda unterstützt die `_X_AMZN_TRACE_ID` Umgebungsvariable bei Lambda Managed Instances nicht. Die X-Ray-Trace-ID wird bei Verwendung des AWS SDK automatisch weitergegeben.

Wird verwendet`context.getRemainingTimeInMillis()`, um Timeouts zu erkennen. Weitere Informationen finden Sie unter the section called “Fehlerbehandlung und Wiederherstellung”.

Initialisierung und Herunterfahren

Die Funktionsinitialisierung erfolgt einmal pro Worker-Thread. Möglicherweise werden Protokolleinträge wiederholt, wenn Ihre Funktion während der Initialisierung Protokolle ausgibt.

Bei Lambda-Funktionen mit Erweiterungen gibt die Ausführungsumgebung beim Herunterfahren ein SIGTERM-Signal aus. Dieses Signal wird von Erweiterungen verwendet, um Bereinigungsaufgaben wie das Leeren von Puffern auszulösen. Lambda-Funktionen (Standard) mit Erweiterungen können das SIGTERM-Signal auch abonnieren, indem `process.on()` Dies wird für Funktionen, die Lambda Managed Instances verwenden, nicht unterstützt, da es `process.on()` nicht mit Worker-Threads verwendet werden kann. Weitere Informationen zum Lebenszyklus der Ausführungsumgebung finden Sie unter the section called “Ausführungsumgebung”.

Versionen von Abhängigkeiten

Lambda Managed Instances erfordert die folgenden Mindestpaketversionen:

- AWS SDK für JavaScript v3: Version 3.933.0 oder höher
- AWS X-Ray SDK für Node.js: Version 3.12.0 oder höher
- AWS Distribution für OpenTelemetry — Instrumentierung für JavaScript: Version 0.8.0 oder höher
- Powertools für AWS Lambda (TypeScript): Version 2.29.0 oder höher

Elektrowerkzeuge für AWS Lambda () TypeScript

Powertools for AWS Lambda (TypeScript) ist mit Lambda Managed Instances kompatibel und bietet Dienstprogramme für Logging, Tracing, Metriken und mehr. Weitere Informationen finden Sie unter Powertools for AWS Lambda () TypeScript.

Nächste Schritte

- Überprüfen Sie die Java-Laufzeit für Lambda Managed Instances
- Überprüfen Sie die Python-Laufzeit für Lambda Managed Instances
- .NET-Laufzeit für Lambda Managed Instances überprüfen
- Erfahren Sie mehr über die Skalierung von Lambda Managed Instances

Python-Laufzeit für verwaltete Lambda-Instanzen

Die Lambda-Laufzeit verwendet mehrere Python-Prozesse, um gleichzeitige Anfragen zu verarbeiten. Jede gleichzeitige Anforderung wird in einem separaten Prozess mit eigenem Speicherplatz und eigener Initialisierung ausgeführt. Jeder Prozess verarbeitet synchron jeweils eine Anfrage. Prozesse teilen sich den Speicher nicht direkt, sodass globale Variablen, Caches auf Modulebene und Singleton-Objekte zwischen gleichzeitigen Anfragen isoliert werden.

Konfiguration der Parallelität

Die maximale Anzahl gleichzeitiger Anfragen, die Lambda an jede Ausführungsumgebung sendet, wird durch die `PerExecutionEnvironmentMaxConcurrency` Einstellung in der Funktionskonfiguration gesteuert. Dies ist eine optionale Einstellung, und der Standardwert variiert je nach Laufzeit. Für Python-Laufzeiten ist die Standardeinstellung 16 gleichzeitige Anfragen pro vCPU, oder Sie können Ihren eigenen Wert konfigurieren. Dieser Wert bestimmt auch die Anzahl der Prozesse, die von der Python-Laufzeit verwendet werden. Lambda passt die Anzahl der gleichzeitigen Anfragen automatisch bis zum konfigurierten Maximum an, basierend auf der Kapazität jeder Ausführungsumgebung, um diese Anfragen aufzunehmen.

> **Important**
>
> Die Verwendung prozessbasierter Parallelität bedeutet, dass jeder Runtime-Worker-Prozess seine eigene Initialisierung durchführt. Die gesamte Speicherbelegung entspricht dem Arbeitsspeicher pro Prozess multipliziert mit der Anzahl gleichzeitiger Prozesse. Wenn Sie große Bibliotheken oder Datensätze laden und über eine hohe Parallelität verfügen, haben Sie einen großen Speicherbedarf. Je nach Arbeitslast müssen Sie möglicherweise Ihr CPU-to-memory Verhältnis anpassen oder eine niedrigere Parallelitätseinstellung verwenden, um eine Überschreitung des verfügbaren Speichers zu vermeiden. Sie können die `MemoryUtilization` Metrik verwenden CloudWatch , um den Speicherverbrauch zu verfolgen.

Erstellung von Funktionen für mehrere Parallelitäten

Aufgrund des prozessbasierten Multi-Concurrency-Modells greifen Lambda Managed Instances-Funktionen, die Python-Laufzeiten verwenden, bei mehreren Aufrufen nicht gleichzeitig auf speicherinterne Ressourcen zu. Sie müssen keine Codierungspraktiken anwenden, um die Sicherheit der Parallelität im Speicher zu gewährleisten.

Gemeinsames /tmp-Verzeichnis

Das `/tmp` Verzeichnis wird von allen gleichzeitigen Anfragen in der Ausführungsumgebung gemeinsam genutzt. Gleichzeitige Schreibvorgänge in dieselbe Datei können zu Datenbeschädigungen führen, z. B. wenn ein anderer Prozess die Datei überschreibt. Um dieses Problem zu lösen, implementieren Sie entweder Dateisperren für gemeinsam genutzte Dateien oder verwenden Sie eindeutige Dateinamen pro Prozess oder pro Anfrage, um Konflikte zu vermeiden. Denken Sie daran, nicht benötigte Dateien zu bereinigen, um zu vermeiden, dass der verfügbare Speicherplatz aufgebraucht wird.

Protokollierung

Das Verschachteln von Protokollen (Protokolleinträge verschiedener Anfragen, die in Protokollen verschachtelt werden) ist bei Systemen mit mehreren gleichzeitigen Vorgängen normal.

Funktionen, die Lambda Managed Instances verwenden, verwenden immer das strukturierte JSON-Protokollformat, das mit erweiterten Protokollierungssteuerungen eingeführt wurde. Dieses Format beinhaltet die`requestId`, sodass Protokolleinträge mit einer einzigen Anfrage korreliert werden können. Wenn Sie das `logging` Modul aus der Python-Standardbibliothek in Lambda verwenden, `requestId` ist das automatisch in jedem Protokolleintrag enthalten. Weitere Informationen finden Sie unter Erweiterte Lambda-Protokollierungssteuerelemente mit Python verwenden.

Kontext anfordern

Wird verwendet`context.aws_request_id`, um auf die Anforderungs-ID für die aktuelle Anfrage zuzugreifen.

Bei Python-Laufzeiten können Sie die `_X_AMZN_TRACE_ID` Umgebungsvariable verwenden, um mit Lambda Managed Instances auf die X-Ray-Trace-ID zuzugreifen. Die X-Ray-Trace-ID wird bei Verwendung des AWS SDK automatisch weitergegeben.

Wird verwendet`context.get_remaining_time_in_millis()`, um Timeouts zu erkennen. Weitere Informationen finden Sie unter the section called “Fehlerbehandlung und Wiederherstellung”.

Initialisierung und Herunterfahren

Die Initialisierung der Funktion erfolgt einmal pro Prozess. Möglicherweise werden Protokolleinträge wiederholt, wenn Ihre Funktion während der Initialisierung Protokolle ausgibt.

Bei Lambda-Funktionen mit Erweiterungen gibt die Ausführungsumgebung beim Herunterfahren ein SIGTERM-Signal aus. Dieses Signal wird von Erweiterungen verwendet, um Bereinigungsaufgaben

wie das Leeren von Puffern auszulösen. Sie können SIGTERM-Ereignisse abonnieren, um Aufgaben zur Funktionsbereinigung auszulösen, z. B. das Schließen von Datenbankverbindungen. Weitere Informationen zum Lebenszyklus der Ausführungsumgebung finden Sie unter the section called “Ausführungsumgebung”.

Versionen von Abhängigkeiten

Lambda Managed Instances erfordert die folgenden Mindestpaketversionen:

- Powertools für AWS Lambda (Python): Version 3.23.0 oder höher

Powertools für AWS Lambda (Python)

Powertools for AWS Lambda (Python) ist mit Lambda Managed Instances kompatibel und bietet Dienstprogramme für Logging, Tracing, Metriken und mehr. Weitere Informationen finden Sie unter Powertools for AWS Lambda (Python).

Nächste Schritte

- Überprüfen Sie die Java-Laufzeit für Lambda Managed Instances
- Überprüfen Sie die Laufzeit von Node.js für Lambda Managed Instances
- .NET-Laufzeit für Lambda Managed Instances überprüfen
- Erfahren Sie mehr über die Skalierung von Lambda Managed Instances

.NET-Laufzeit für verwaltete Lambda-Instanzen

Für .NET-Laufzeiten verwenden Lambda Managed Instances einen einzigen .NET-Prozess pro Ausführungsumgebung. Mithilfe von.NET-Aufgaben werden mehrere Anfragen gleichzeitig verarbeitet.

Konfiguration der Parallelität

Die maximale Anzahl gleichzeitiger Anfragen, die Lambda an jede Ausführungsumgebung sendet, wird durch die `PerExecutionEnvironmentMaxConcurrency` Einstellung in der Funktionskonfiguration gesteuert. Dies ist eine optionale Einstellung, und der Standardwert variiert je nach Laufzeit. Für .NET-Laufzeiten ist die Standardeinstellung 32 gleichzeitige Anfragen pro vCPU, oder Sie können Ihren eigenen Wert konfigurieren. Lambda passt die Anzahl der gleichzeitigen

Anfragen automatisch bis zum konfigurierten Maximum an, basierend auf der Kapazität jeder Ausführungsumgebung, um diese Anfragen aufzunehmen.

Erstellung von Funktionen für Mehrfachparallelität

Sie sollten bei der Verwendung von Lambda Managed Instances dieselben Sicherheitspraktiken für Parallelität anwenden wie in jeder anderen Umgebung mit mehreren gleichzeitigen Vorgängen. Da das Handler-Objekt von allen Tasks gemeinsam genutzt wird, muss jeder veränderbare Status Thread-sicher sein. Dazu gehören Sammlungen, Datenbankverbindungen und alle statischen Objekte, die während der Anforderungsverarbeitung geändert werden.

AWS SDK-Clients sind Thread-sicher und erfordern keine besondere Behandlung.

Beispiel: Datenbank-Verbindungspools

Der folgende Code verwendet ein statisches Datenbankverbindungsobjekt, das von mehreren gleichzeitigen Anfragen gemeinsam genutzt wird. Das `SqlConnection` Objekt ist nicht threadsicher.

```
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();

        ...
    }
}
```

Um dieses Problem zu lösen, verwenden Sie für jede Anfrage eine separate Verbindung, die aus einem Verbindungspool stammt. ADO.NET-Anbieter unterstützen beispielsweise

`Microsoft.Data.SqlClient` automatisch das Verbindungspooling, wenn das Verbindungsobjekt geöffnet wird.

```
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();

        ...
    }
}
```

Beispiel: Sammlungen

Standard-.NET-Sammlungen sind nicht threadsicher:

```
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";
    }
}
```

Verwenden Sie aus Sicherheitsgründen Sammlungen aus dem `System.Collections.Concurrent` Namespace:

```
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";
    }
}
```

Gemeinsames /tmp-Verzeichnis

Das `/tmp` Verzeichnis wird von allen gleichzeitigen Anfragen in der Ausführungsumgebung gemeinsam genutzt. Gleichzeitige Schreibvorgänge in dieselbe Datei können zu Datenbeschädigungen führen, z. B. wenn eine andere Anforderung die Datei überschreibt. Um dieses Problem zu lösen, implementieren Sie entweder Dateisperren für gemeinsam genutzte Dateien oder verwenden Sie eindeutige Dateinamen pro Anfrage, um Konflikte zu vermeiden. Denken Sie daran, nicht benötigte Dateien zu bereinigen, um zu vermeiden, dass der verfügbare Speicherplatz aufgebraucht wird.

Protokollierung

Das Verschachteln von Protokollen (Protokolleinträge verschiedener Anfragen, die in Protokollen verschachtelt werden) ist bei Systemen mit mehreren gleichzeitigen Vorgängen normal. Funktionen, die Lambda Managed Instances verwenden, verwenden immer das strukturierte JSON-Protokollformat, das mit erweiterten Protokollierungssteuerungen eingeführt wurde. Dieses Format beinhaltet die`requestId`, sodass Protokolleinträge mit einer einzigen Anfrage korreliert werden können. Wenn Sie das `context.Logger` Objekt zum Generieren von Protokollen verwenden, `requestId` wird es automatisch in jeden Protokolleintrag aufgenommen. Weitere Informationen finden Sie unterthe section called “Verwenden von Lambda-Optionen für die erweiterte Protokollierung mit .NET”.

Kontext anfordern

Verwenden Sie die `context.AwsRequestId` Eigenschaft, um auf die Anforderungs-ID für die aktuelle Anfrage zuzugreifen.

Verwenden Sie die `context.TraceId` Eigenschaft, um auf die X-Ray-Trace-ID zuzugreifen. Dies ermöglicht einen parallelen Zugriff auf die Trace-ID für die aktuelle Anfrage. Lambda unterstützt die `_X_AMZN_TRACE_ID` Umgebungsvariable bei Lambda Managed Instances nicht. Die X-Ray-Trace-ID wird bei Verwendung des AWS SDK automatisch weitergegeben.

Wird verwendet`ILambdaContext.RemainingTime`, um Timeouts zu erkennen. Weitere Informationen finden Sie unter the section called “Fehlerbehandlung und Wiederherstellung”.

Initialisierung und Herunterfahren

Die Funktionsinitialisierung erfolgt einmal pro Ausführungsumgebung. Objekte, die während der Initialisierung erstellt wurden, werden von allen Anfragen gemeinsam genutzt.

Bei Lambda-Funktionen mit Erweiterungen gibt die Ausführungsumgebung beim Herunterfahren ein SIGTERM-Signal aus. Dieses Signal wird von Erweiterungen verwendet, um Bereinigungsaufgaben wie das Leeren von Puffern auszulösen. Sie können SIGTERM-Ereignisse abonnieren, um Aufgaben zur Funktionsbereinigung auszulösen, z. B. das Schließen von Datenbankverbindungen. Weitere Informationen zum Lebenszyklus der Ausführungsumgebung finden Sie unter the section called “Ausführungsumgebung”.

Versionen von Abhängigkeiten

Für Lambda Managed Instances sind die folgenden Mindestpaketversionen erforderlich:

- Amazon.Lambda.Core: Version 2.7.1 oder höher
- Amazon.Lambda. RuntimeSupport: Version 1.14.1 oder höher
- OpenTelemetry. Instrumentierung. AWSLambda: Version 1.14.0 oder höher
- AWSXRayRecorder.Core: Version 2.16.0 oder höher
- AWSSDK.Core: Version 4.0.0.32 oder höher

Powertools für AWS Lambda (.NET)

Powertools for AWS Lambda (.NET) und AWS Distro for OpenTelemetry - Instrumentation for unterstützen DotNet derzeit keine Lambda Managed Instances.

Nächste Schritte

- Überprüfen Sie die Java-Laufzeit für Lambda Managed Instances
- Überprüfen Sie die Laufzeit von Node.js für Lambda Managed Instances
- Überprüfen Sie die Python-Laufzeit für Lambda Managed Instances
- Erfahren Sie mehr über die Skalierung von Lambda Managed Instances

Netzwerke für Lambda Managed Instances

Wenn Sie Funktionen von Lambda Managed Instances ausführen, müssen Sie die Netzwerkkonnektivität konfigurieren, damit Ihre Funktionen auf Ressourcen außerhalb der VPC zugreifen können. Dazu gehören AWS Dienste wie Amazon S3 und DynamoDB. Die Konnektivität wird auch für die Übertragung von Telemetriedaten an CloudWatch Logs und X-Ray benötigt.

Verbindungsoptionen

Es gibt drei Hauptansätze für die Konfiguration der VPC-Konnektivität, die jeweils unterschiedliche Kompromisse in Bezug auf Kosten, Sicherheit und Komplexität beinhalten.

Öffentliches Subnetz mit einem Internet-Gateway

Diese Option verwendet ein öffentliches Subnetz mit direktem Internetzugang über ein Internet-Gateway. Sie können zwischen IPv4 und IPv6 Konfigurationen wählen.

IPv4 mit Internet-Gateway

Um die IPv4 Konnektivität mit einem Internet-Gateway zu konfigurieren

1. Erstellen oder verwenden Sie ein vorhandenes öffentliches Subnetz mit einem IPv4 CIDR-Block.
2. Hinzufügen eines Internet-Gateways zu Ihrer VPC.
3. Aktualisieren Sie die Routing-Tabelle, um den `0.0.0.0/0` Verkehr an das Internet-Gateway weiterzuleiten.
4. Stellen Sie sicher, dass den Ressourcen öffentliche IPv4 Adressen oder Elastic IP-Adressen zugewiesen sind.
5. Konfigurieren Sie Sicherheitsgruppen so, dass ausgehender Datenverkehr auf den erforderlichen Ports zugelassen wird.

Diese Konfiguration bietet bidirektionale Konnektivität und ermöglicht sowohl ausgehende Verbindungen von Ihren Funktionen als auch eingehende Verbindungen aus dem Internet.

IPv6 mit Internet-Gateway

Um die IPv6 Konnektivität mit einem Internet-Gateway zu konfigurieren

1. Aktivieren Sie es IPv6 auf Ihrer VPC.
2. Erstellen oder verwenden Sie ein vorhandenes öffentliches Subnetz, dem ein IPv6 CIDR-Block zugewiesen wurde.
3. Schließen Sie ein Internet-Gateway an Ihre VPC an (dasselbe Internet-Gateway kann IPv4 sowohl als auch verarbeiten IPv6).
4. Aktualisieren Sie die Routentabelle, um den `::/0` Verkehr an das Internet-Gateway weiterzuleiten.
5. Vergewissern Sie sich, dass die AWS Dienste, die Sie für den Zugriff auf den Support IPv6 in Ihrer Region benötigen, vorhanden sind.
6. Konfigurieren Sie Sicherheitsgruppen so, dass ausgehender Datenverkehr an den erforderlichen Ports zugelassen wird.

Diese Konfiguration bietet bidirektionale Konnektivität mithilfe von IPv6 Adressierung.

IPv6 mit Internet-Gateway nur für ausgehenden Datenverkehr

So konfigurieren Sie die IPv6 Konnektivität mit einem Internet-Gateway nur für ausgehenden Datenverkehr

1. Aktivieren Sie es IPv6 auf Ihrer VPC.
2. Erstellen oder verwenden Sie ein vorhandenes öffentliches Subnetz, dem ein IPv6 CIDR-Block zugewiesen wurde.
3. Schließen Sie ein Internet-Gateway nur für ausgehenden Datenverkehr an Ihre VPC an.
4. Aktualisieren Sie die Routing-Tabelle, um den Datenverkehr an das Internet-Gateway weiterzuleiten, das `::/0` nur für ausgehenden Verkehr bestimmt ist.
5. Vergewissern Sie sich, dass die AWS Dienste, die Sie für den Zugriff auf den Support IPv6 in Ihrer Region benötigen, vorhanden sind.
6. Konfigurieren Sie Sicherheitsgruppen so, dass ausgehender Datenverkehr an den erforderlichen Ports zugelassen wird.

Diese Konfiguration bietet nur ausgehende Konnektivität, wodurch eingehende Verbindungen aus dem Internet verhindert werden, während Ihre Funktionen ausgehende Verbindungen initiieren können.

VPC-Endpunkte

VPC-Endpunkte ermöglichen es Ihnen, Ihre VPC privat mit unterstützten AWS Diensten zu verbinden, ohne dass ein Internet-Gateway, ein NAT-Gerät, eine VPN-Verbindung oder eine AWS Direct Connect-Verbindung erforderlich ist. Der Verkehr zwischen Ihrer VPC und dem AWS Service verlässt das Amazon-Netzwerk nicht.

So konfigurieren Sie VPC-Endpoints

1. Öffnen Sie die Amazon VPC-Konsole unter console.aws.amazon.com/vpc/.
2. Wählen Sie im Navigationsbereich Endpunkte aus.
3. Wählen Sie Endpunkt erstellen aus.
4. Wählen Sie bei Service category (Servicekategorie) die Option AWS services (-Services) aus.
5. Wählen Sie unter Service-Name den Service-Endpunkt aus, den Sie benötigen (z. B. `com.amazonaws.region.s3` für Amazon S3).
6. Wählen Sie im Feld VPC Ihre VPC aus.
7. Wählen Sie für Subnetze die Subnetze aus, in denen Sie Endpunkt-Netzwerkschnittstellen erstellen möchten. Wählen Sie für eine hohe Verfügbarkeit Subnetze in mehreren Availability Zones aus.
8. Wählen Sie für Sicherheitsgruppen die Sicherheitsgruppen aus, die den Security groups (Endpunkt-Netzwerkschnittstellen) zugeordnet werden sollen. Die Sicherheitsgruppen müssen eingehenden Datenverkehr von der Sicherheitsgruppe Ihrer Funktion an den erforderlichen Ports zulassen.
9. Wählen Sie Endpunkt erstellen aus.

Wiederholen Sie diese Schritte für jeden AWS Dienst, auf den Ihre Funktionen zugreifen müssen.

Privates Subnetz mit NAT-Gateway

Diese Option verwendet ein NAT-Gateway, um den Internetzugang für Ressourcen in privaten Subnetzen bereitzustellen und gleichzeitig die Ressourcen privat zu halten.

Um ein privates Subnetz mit einem NAT-Gateway zu konfigurieren

1. Erstellen Sie ein öffentliches Subnetz (falls noch keines vorhanden ist) mit einem CIDR-Block.
2. Hinzufügen eines Internet-Gateways zu Ihrer VPC.
3. Erstellen Sie ein NAT-Gateway im öffentlichen Subnetz und weisen Sie eine Elastic IP-Adresse zu.
4. Aktualisieren Sie die Routing-Tabelle des öffentlichen Subnetzes, um eine Route hinzuzufügen: `0.0.0.0/0` → Internet-Gateway.
5. Erstellen oder verwenden Sie ein vorhandenes privates Subnetz mit einem CIDR-Block.
6. Aktualisieren Sie die Routentabelle des privaten Subnetzes, um eine Route hinzuzufügen: `0.0.0.0/0` → NAT-Gateway.
7. Konfigurieren Sie Sicherheitsgruppen so, dass ausgehender Verkehr an den erforderlichen Ports zugelassen wird.

Stellen Sie für hohe Verfügbarkeit ein NAT-Gateway in jeder Availability Zone bereit und konfigurieren Sie Routing-Tabellen pro Availability Zone, um das lokale NAT-Gateway zu verwenden. Dadurch werden AZ-übergreifende Datenübertragungsgebühren vermieden und die Ausfallsicherheit verbessert.

Auswahl einer Konnektivitätsoption

Berücksichtigen Sie bei der Auswahl einer Konnektivitätsoption die folgenden Faktoren:

Öffentliches Subnetz mit Internet-Gateway

- Einfachste Konfiguration mit niedrigsten Kosten
- Geeignet für Entwicklungs- und Testumgebungen
- Ressourcen können eingehende Verbindungen aus dem Internet empfangen (aus Sicherheitsgründen)
- Unterstützt sowohl als IPv4 IPv6

VPC-Endpunkte

- Höchste Sicherheit, der Verkehr bleibt im AWS Netzwerk
- Geringere Latenz im Vergleich zum Internet-Routing
- Empfohlen für Produktionsumgebungen mit strengen Sicherheitsanforderungen
- Höhere Kosten pro Endpunkt, pro Availability Zone und pro verarbeitetem GB
- Für hohe Verfügbarkeit ist in jeder Availability Zone ein Endpunkt erforderlich

Privates Subnetz mit NAT-Gateway

- Ressourcen bleiben privat, wenn kein eingehender Internetzugang besteht
- Standardmuster für Unternehmensarchitekturen
- Unterstützt den gesamten IPv4 Internetverkehr
- Moderate Kosten mit Stunden- und Datenverarbeitungsgebühren für das NAT-Gateway
- Unterstützt IPv4 nur

Nächste Schritte

- Erfahren Sie mehr über Kapazitätsanbieter für Lambda Managed Instances
- Skalierung für Lambda Managed Instances verstehen
- Lesen Sie laufzeitspezifische Anleitungen für Java, Node.js und Python
- Sicherheit und Berechtigungen für Lambda Managed Instances verstehen

Überwachung von Lambda-verwalteten Instanzen

Sie können Lambda Managed Instances mithilfe von CloudWatch Metriken überwachen. Lambda veröffentlicht automatisch Kennzahlen, mit denen CloudWatch Sie die Ressourcennutzung überwachen, Kosten verfolgen und die Leistung optimieren können.

Verfügbare Metriken

Lambda Managed Instances bietet Metriken auf zwei Ebenen: auf Ebene des Kapazitätsanbieters und auf Ebene der Ausführungsumgebung.

Metriken auf Ebene des Kapazitätsanbieters

Kennzahlen auf Ebene des Kapazitätsanbieters bieten Einblick in die allgemeine Ressourcenauslastung Ihrer Instances. Diese Metriken verwenden die folgenden Dimensionen:

- CapacityProviderName- Der Name Ihres Kapazitätsanbieters
- InstanceType- Der EC2 Instanztyp

Kennzahlen zur Ressourcennutzung:

- CPUUtilization- Der Prozentsatz der CPU-Auslastung aller Instanzen im Kapazitätsanbieter
- MemoryUtilization- Der Prozentsatz der Speicherauslastung für alle Instanzen im Kapazitätsanbieter
- NetworkOut- Netzwerkverkehr, der über das Kunden-ENI gesendet wird (in Byte)
- NetworkIn- Über das Kunden-ENI empfangener Netzwerkverkehr (in Byte)
- DiskReadBytes- Lesen Sie den Datenverkehr aus dem lokalen Speicher über Instanzen hinweg (in Byte)
- DiskWriteBytes- Schreiben Sie den Datenverkehr instanzübergreifend in den lokalen Speicher (in Byte)

Kapazitätskennzahlen:

- v CPUAvailable — Die Menge an vCPUs, die auf Instances für die Zuweisung verfügbar ist (in Anzahl)
- MemoryAvailable— Die Menge an Arbeitsspeicher, die auf Instances für die Zuweisung verfügbar ist (in Byte)
- v CPUAllocated — Die Menge an vCPUs, die Instanzen für Ausführungsumgebungen zugewiesen sind (in Anzahl)
- MemoryAllocated— Die Menge an Speicher, die Instanzen für Ausführungsumgebungen zugewiesen wurde (in Byte)

Metriken auf Ausführungsumgebungsebene

Metriken auf der Ebene der Ausführungsumgebung bieten Einblick in die Ressourcennutzung und Parallelität einzelner Funktionen. Diese Metriken verwenden die folgenden Dimensionen:

- CapacityProviderName- Der Name Ihres Kapazitätsanbieters
- FunctionName- Der Name Ihrer Lambda-Funktion

Verfügbare Metriken der Ausführungsumgebung:

- ExecutionEnvironmentConcurrency- Die maximale Parallelität über einen Probenzeitraum von 5 Minuten
- ExecutionEnvironmentConcurrencyLimit— Das maximale Parallelitätslimit pro Ausführungsumgebung

- ExecutionEnvironmentCPUUtilization— Der Prozentsatz der CPU-Auslastung für die Ausführungsumgebungen der Funktion
- ExecutionEnvironmentMemoryUtilization- Der Prozentsatz der Speicherauslastung für die Ausführungsumgebungen der Funktion

Metrische Häufigkeit und Retention

Die Metriken für Lambda Managed Instances werden in Intervallen von 5 Minuten veröffentlicht und 15 Monate lang aufbewahrt.

Metriken anzeigen in CloudWatch

So zeigen Sie Metriken für Lambda Managed Instances in der Konsole an CloudWatch

1. Öffnen Sie die CloudWatch Konsole unter console.aws.amazon.com/cloudwatch/.
2. Wählen Sie im Navigationsbereich Metriken aus.
3. Wählen Sie auf der Registerkarte Alle Metriken die Option AWS/Lambda aus.
4. Wählen Sie die metrische Dimension aus, die Sie anzeigen möchten:
   - Filtern Sie für Kennzahlen auf Ebene des Kapazitätsanbieters nach CapacityProviderNameund InstanceType
   - Für Metriken auf Ausführungsumgebungsebene filtern Sie nach CapacityProviderNameund FunctionName
5. Wählen Sie die Metriken aus, die Sie überwachen möchten.

Verwenden von Metriken zur Leistungsoptimierung

Überwachen Sie die CPU- und Speicherauslastung, um festzustellen, ob Ihre Funktionen richtig dimensioniert sind. Eine hohe Auslastung kann darauf hindeuten, dass größere Instance-Typen oder eine größere Zuweisung von Funktionsspeicher erforderlich sind. Verfolgen Sie Parallelitätskennzahlen, um das Skalierungsverhalten zu verstehen und mögliche Drosselungen zu identifizieren.

Überwachen Sie die Kapazitätskennzahlen, um sicherzustellen, dass ausreichend Ressourcen für Ihre Workloads verfügbar sind. Anhand von V CPUAvailable und MemoryAvailableMetriken können Sie die verbleibende Kapazität Ihrer Instances ermitteln.

Nächste Schritte

- Erfahren Sie mehr über die Skalierung von Lambda Managed Instances
- Lesen Sie laufzeitspezifische Anleitungen für Java, Node.js und Python
- Konfigurieren Sie die VPC-Konnektivität für Ihre Kapazitätsanbieter
- Sicherheit und Berechtigungen für Lambda Managed Instances verstehen

Kontingente für Lambda Managed Instances

Auf dieser Seite werden die Dienstkontingente für AWS Lambda Managed Instances beschrieben. Diese Kontingente sind von AWS Lambda-Kontingenten (Standard) getrennt. Einige Kontingente können auf Anfrage erhöht werden.

Lambda-API-Anforderungskontingente

Diese Kontingente steuern die Geschwindigkeit, mit der Sie API-Aufrufe tätigen können, um Kapazitätsanbieter für Lambda Managed Instances zu verwalten. Die API-Lese- und Schreibratenbegrenzungen gelten für alle Kapazitätsanbieteroperationen zusammen, einschließlich der Erstellung, Aktualisierung, Beschreibung und Löschung von Kapazitätsanbietern.

Ressource	Kontingent
Die maximale kombinierte Rate (Anfragen pro Sekunde) für alle Lesevorgänge von Kapazitätsanbietern APIs	15 Anforderungen pro Sekunde. Kann nicht erhöht werden.
Die maximale kombinierte Rate (Anfragen pro Sekunde) für alle Schreibvorgänge des Kapazitätsanbieters APIs	1 Anfrage pro Sekunde. Kann nicht erhöht werden.

Ressourcenkontingente für Lambda Managed Instances

Diese Kontingente definieren die Grenzwerte für die Kernressourcen von Lambda Managed Instances in Ihrem AWS Konto. Sie regeln die Anzahl der Kapazitätsanbieter, die Sie erstellen können, und die Anzahl der Funktionsversionen, die jedem Kapazitätsanbieter zugeordnet werden können.

Ressource	Kontingent
Kapazitätsanbieter	1.000. Die maximale Anzahl von Kapazitätsanbietern, die in einem Konto erstellt wurden.
Funktionsversionen pro Kapazitätsanbieter	100. Die maximale Anzahl von Funktionsversionen pro Kapazitätsanbieter. Kann nicht erhöht werden.

Kontingente für die Zuordnung von Ereignisquellen

Diese Kontingente steuern den Durchsatz und die Konfigurationslimits für die Verarbeitung von Ereignissen verschiedener AWS Dienste auf Lambda Managed Instances. Die Durchsatzgrenzen sorgen für eine vorhersehbare Leistung, während die Grenzwerte für die Anzahl der Zuordnungen zur Aufrechterhaltung der Servicestabilität beitragen. Zuordnungen von Ereignisquellen auf Lambda Managed Instances unterstützen Amazon SQS, DynamoDB Streams, Amazon Kinesis Data Streams, Amazon MSK und selbstverwaltetes Apache Kafka als Ereignisquellen.

Ressource	Kontingent
Standarddurchsatz für die Zuordnung von SQS-Ereignisquellen auf Lambda Managed Instances	5 MB pro Sekunde. Kann nicht erhöht werden.
Standarddurchsatz für die Zuordnung von Kafka-Ereignisquellen auf Lambda Managed Instances	1 MB pro Sekunde. Kann nicht erhöht werden.
Standard-Kafka-Ereignisquellenzuordnungen auf Lambda Managed Instances	100 Zuordnungen von Ereignisquellen. Kann nicht erhöht werden.
Durchsatz der Kinesis-Ereignisquellenzuordnung auf Lambda Managed Instances	25 MB pro Sekunde. Kann erhöht werden.
Durchsatz für die Zuordnung von DynamoDB-Ereignisquellen auf Lambda Managed Instances	10 MB pro Sekunde. Kann erhöht werden.

Ressource	Kontingent
Rufen Sie den Anforderungsdurchsatz für asynchrone Aufrufe auf Lambda Managed Instances auf	5 MB pro Sekunde. Kann erhöht werden.

Beantragen einer Kontingenterhöhung

Für Kontingente, die erhöht werden können, können Sie eine Erhöhung über die Service Quotas Quotas-Konsole beantragen.

So fordern Sie eine Kontingenterhöhung an

1. Öffnen Sie die Service Quotas Quotas-Konsole unter console.aws.amazon.com/servicequotas/.
2. Wählen Sie im Navigationsbereich AWS -Services.
3. Wählen Sie AWS Lambda.
4. Wählen Sie das Kontingent aus, das Sie erhöhen möchten.
5. Wählen Sie Kontingenterhöhung anfordern.
6. Geben Sie den neuen Kontingentwert ein und begründen Sie die Erhöhung.
7. Wählen Sie Anfrage aus.

Nächste Schritte

- Erfahren Sie mehr über Kapazitätsanbieter für Lambda Managed Instances
- Skalierung für Lambda Managed Instances verstehen
- Lesen Sie laufzeitspezifische Anleitungen für Java, Node.js und Python
- Konfigurieren Sie die VPC-Konnektivität für Ihre Kapazitätsanbieter

Bewährte Methoden für Lambda Managed Instances

Konfiguration des Kapazitätsanbieters

Separate Kapazitätsanbieter nach Vertrauensstufe. Erstellen Sie verschiedene Kapazitätsanbieter für Workloads mit unterschiedlichen Sicherheitsanforderungen. Alle Funktionen, die demselben

Kapazitätsanbieter zugewiesen sind, müssen gegenseitig vertrauenswürdig sein, da Kapazitätsanbieter als Sicherheitsgrenze dienen.

Verwenden Sie aussagekräftige Namen. Benennen Sie Kapazitätsanbieter, um deren Verwendungszweck und Vertrauensstufe eindeutig anzugeben (z. B.`production-trusted`,`dev-sandbox`). Dies hilft den Teams, den Zweck und die Sicherheitslage der einzelnen Kapazitätsanbieter zu verstehen.

Verwenden Sie mehrere Availability Zones. Geben Sie bei der Erstellung von Kapazitätsanbietern Subnetze für mehrere Availability Zones an. Lambda startet aus Gründen der AZ-Resilienz standardmäßig drei Instances und gewährleistet so eine hohe Verfügbarkeit Ihrer Funktionen.

Auswahl des Instance-Typs

Lassen Sie Lambda die Instanztypen auswählen. Standardmäßig wählt Lambda die besten Instance-Typen für Ihren Workload aus. Wir empfehlen, Lambda Managed Instances die Instanztypen für Sie auswählen zu lassen, da eine Einschränkung der Anzahl möglicher Instanztypen zu einer geringeren Verfügbarkeit führen kann.

Geben Sie Instanztypen für bestimmte Anforderungen an. Wenn Sie bestimmte Hardwareanforderungen haben, legen Sie für zulässige Instanztypen eine Liste kompatibler Instanzen fest. Beispiel:

- Wählen Sie für Anwendungen, die eine hohe Netzwerkbandbreite erfordern, mehrere n-Instance-Typen aus
- Wählen Sie für Test- oder Entwicklungsumgebungen mit Kostenbeschränkungen kleinere Instance-Typen wie m7a.large

Konfiguration der Funktionen

Wählen Sie die entsprechenden Speicher- und vCPU-Einstellungen. Wählen Sie Speicher- und vCPU-Konfigurationen aus, die mehrere gleichzeitige Ausführungen Ihrer Funktion unterstützen. Die unterstützte Mindestfunktionsgröße beträgt 2 GB und 1 vCPU.

- Wählen Sie für Python-Anwendungen ein höheres Verhältnis von Speicher zu V CPUs (z. B. 4 zu 1 oder 8 zu 1), da Python Multiparallelität so handhabt
- Wählen Sie für CPU-intensive Operationen oder Funktionen mit wenig I/O mehr als eine vCPU

- Für IO-intensive Anwendungen wie Webservices oder Batch-Jobs bietet die Mehrfachparallelität den größten Vorteil

Konfigurieren Sie die maximale Parallelität entsprechend. Lambda wählt vernünftige Standardwerte für maximale Parallelität, die den Ressourcenverbrauch und den Durchsatz in Einklang bringen. Passen Sie diese Einstellung an die Ressourcennutzung Ihrer Funktion an:

- Erhöhen Sie die maximale Parallelität (bis zu 64 pro vCPU), wenn Ihre Funktionsaufrufe nur sehr wenig CPU verbrauchen
- Verringern Sie die maximale Parallelität, wenn Ihre Anwendung viel Speicher und sehr wenig CPU verbraucht

Beachten Sie, dass in Ausführungsumgebungen mit sehr geringer Parallelität Drosselungen und Skalierungsschwierigkeiten auftreten können.

Skalierungskonfiguration

Stellen Sie die entsprechende Zielressourcennutzung ein. Standardmäßig bietet Lambda genügend Spielraum, sodass sich Ihr Traffic ohne Drosselungen innerhalb von 5 Minuten verdoppeln kann. Passen Sie dies an Ihre Workload-Merkmale an:

- Bei sehr gleichbleibenden Workloads oder Anwendungen, die nicht empfindlich auf Drosselungen reagieren, sollten Sie das Ziel auf einen hohen Wert setzen, um eine höhere Auslastung und niedrigere Kosten zu erreichen
- Setzen Sie sich bei Workloads mit potenziellen Datenverkehrsspitzen niedrige Ressourcenziele, um zusätzlichen Spielraum zu haben

Planen Sie das Verkehrswachstum ein. Wenn sich Ihr Traffic innerhalb von 5 Minuten mehr als verdoppelt, kann es zu Drosselungen kommen, wenn Lambda Instances und Ausführungsumgebungen skaliert. Entwerfen Sie Ihre Anwendung so, dass sie potenziellen Drosselungen in Zeiten schneller Skalierung begegnet.

Sicherheit

Wenden Sie die geringsten Rechte für Berechtigungen an. PassCapacityProvider Erteilen `lambda:PassCapacityProvider` Sie Berechtigungen nur für die erforderlichen

Kapazitätsanbieter. Verwenden Sie Berechtigungen auf Ressourcenebene, um einzuschränken, welche Kapazitätsanbieter Benutzer Funktionen zuweisen können.

Überwachen Sie die Nutzung von Kapazitätsanbietern. Wird verwendet AWS CloudTrail , um die Zuweisungen und Zugriffsmuster von Kapazitätsanbietern zu überwachen. Dies hilft bei der Identifizierung unberechtigter Zugriffsversuche und gewährleistet die Einhaltung von Sicherheitsrichtlinien.

Trennen Sie nicht vertrauenswürdige Workloads. Verlassen Sie sich bei der Sicherheitsisolierung zwischen nicht vertrauenswürdigen Workloads nicht auf Container. Verwenden Sie verschiedene Kapazitätsanbieter, um Workloads zu trennen, denen nicht gegenseitig vertraut wird.

Kostenoptimierung

Nutzen Sie EC2 Preisoptionen. Nutzen Sie EC2 Savings Plans und Reserved Instances, um die Kosten zu senken. Diese Preisoptionen gelten für die zugrunde liegende EC2 Rechenleistung (die Verwaltungsgebühr von 15% wird nicht rabattiert).

Optimieren Sie für stationäre Workloads. Lambda Managed Instances eignen sich am besten für Steady-State-Funktionen mit vorhersehbarem hohem Verkehrsaufkommen. Bei stark frequentierten Datenverkehrsmustern kann Lambda (Standard) kostengünstiger sein.

Überwachen Sie die Ressourcennutzung. Verfolgen Sie CloudWatch Metriken, um die CPU- und Speicherauslastung zu verstehen. Passen Sie die Funktionsspeicherzuweisung und die Auswahl des Instanztyps an die tatsächlichen Nutzungsmuster an, um die Kosten zu optimieren.

Überwachung und Beobachtbarkeit

Überwachen Sie die Kennzahlen der Kapazitätsanbieter. Verfolgen Sie Kennzahlen auf Ebene der Kapazitätsanbieter CPUUtilization MemoryUtilization, einschließlich,CPUAvailable, und, MemoryAvailable um sicherzustellen, dass genügend Ressourcen für Ihre Workloads verfügbar sind.

Überwachen Sie die Kennzahlen der Ausführungsumgebung. Verfolgen Sie Metriken auf der Ebene der Ausführungsumgebung, einschließlich ExecutionEnvironmentConcurrency und ExecutionEnvironmentConcurrencyLimit um das Skalierungsverhalten zu verstehen und mögliche Drosselungen zu identifizieren.

Richten Sie Alarme ein. CloudWatch Erstellen Sie CloudWatch Alarme für wichtige Kennzahlen, um Probleme proaktiv zu identifizieren:

- Hohe CPU- oder Speicherauslastung
- Niedrige verfügbare Kapazität
- Wir nähern uns den Grenzwerten für Parallelität

Laufzeitspezifische Überlegungen

Folgen Sie laufzeitspezifischen Best Practices. Jede Laufzeit behandelt Multiparallelität unterschiedlich. Ausführliche Empfehlungen finden Sie in den laufzeitspezifischen Leitfäden:

- Java: Verwenden Sie Thread-sichere Sammlungen und für anforderungsspezifische `AtomicInteger` Statusmeldungen `ThreadLocal`
- Node.js: Wird InvokeStore für alle anforderungsspezifischen Zustände verwendet und globale Variablen vermieden
- Python: Verwenden Sie eindeutige Dateinamen in der `/tmp` Anfrage IDs und ziehen Sie die prozessbasierte Speicherisolierung in Betracht

Testen Sie, ob Thread-Sicherheits- und Parallelitätsprobleme bestehen. Testen Sie Ihre Funktionen vor der Bereitstellung in der Produktion gründlich auf Thread-Sicherheitsprobleme, Race-Bedingungen und die korrekte Zustandsisolierung bei gleichzeitiger Last.

Nächste Schritte

- Erfahren Sie mehr über Kapazitätsanbieter für Lambda Managed Instances
- Skalierung für Lambda Managed Instances verstehen
- Lesen Sie laufzeitspezifische Anleitungen für Java, Node.js und Python
- Konfigurieren Sie die VPC-Konnektivität für Ihre Kapazitätsanbieter
- Überwachen Sie Lambda Managed Instances mit Metriken CloudWatch

Fehlerbehebung bei verwalteten Lambda-Instanzen

Probleme bei der Drosselung und Skalierung

Hohe Fehlerraten bei der Skalierung

Problem: Es treten Drosselungsfehler (HTTP 429) auf, wenn der Traffic schnell zunimmt.

Ursache: Lambda Managed Instances werden asynchron auf der Grundlage der CPU-Ressourcenauslastung und der Mehrparallelität skaliert. Wenn sich Ihr Traffic innerhalb von 5 Minuten mehr als verdoppelt, kann es zu Drosselungen kommen, wenn Lambda Instances und Ausführungsumgebungen entsprechend der Nachfrage skaliert.

Lösung:

- Passen Sie die Zielressourcenauslastung an: Wenn Ihr Workload vorhersehbare Verkehrsmuster aufweist, legen Sie eine niedrigere Zielressourcenauslastung fest, um zusätzlichen Spielraum für Datenverkehrsspitzen zu haben.
- Kapazität vor der Erwärmung: Erhöhen Sie den Datenverkehr bei einem geplanten Anstieg des Datenverkehrs über einen längeren Zeitraum schrittweise, damit die Skalierung Schritt halten kann.
- Überwachen Sie die Skalierungsmetriken: Verfolgen Sie die Messwerte für Drosselungsfehler, um den Grund für Drosselungen und Probleme mit der Kapazitätsskalierung zu verstehen.
- Überprüfen Sie die Funktionskonfiguration: Stellen Sie sicher, dass Ihre Funktionsspeicher- und vCPU-Einstellungen mehrere gleichzeitige Ausführungen unterstützen. Erhöhen Sie bei Bedarf die Funktionsspeicher- oder vCPU-Zuweisung.

Langsames Herunterskalieren

Problem: Es dauert lange, bis Instances herunterskaliert werden, nachdem der Traffic zurückgegangen ist.

Ursache: Lambda Managed Instances werden schrittweise herunterskaliert, um die Verfügbarkeit aufrechtzuerhalten und schnelle Kapazitätsänderungen zu vermeiden, die sich auf die Leistung auswirken könnten.

Lösung:

Dieses Verhalten wird erwartet. Lambda skaliert Instanzen konservativ herunter, um Stabilität zu gewährleisten. Überwachen Sie Ihre CloudWatch Metriken, um die Anzahl der laufenden Instanzen zu verfolgen.

Probleme mit der Parallelität

Bei Ausführungsumgebungen mit geringer Parallelität kommt es zu Drosselungen

Problem: Ihre Funktionen werden trotz verfügbarer Kapazität gedrosselt.

Ursache: Ausführungsumgebungen mit sehr niedriger maximaler Parallelität können Schwierigkeiten haben, effektiv zu skalieren. Lambda Managed Instances sind für Anwendungen mit mehreren gleichzeitigen Vorgängen konzipiert.

Lösung:

- Erhöhen Sie die maximale Parallelität: Wenn Ihre Funktionsaufrufen nur sehr wenig CPU verbrauchen, erhöhen Sie die Einstellung für maximale Parallelität auf 64 pro vCPU.
- Funktionscode optimieren: Überprüfen Sie Ihren Funktionscode, um den CPU-Verbrauch pro Aufruf zu reduzieren und so eine höhere Parallelität zu ermöglichen.
- Passen Sie den Funktionsspeicher und die vCPU an: Stellen Sie sicher, dass Ihre Funktion über ausreichende Ressourcen verfügt, um mehrere gleichzeitige Aufrufe zu verarbeiten.

Probleme mit der Thread-Sicherheit (Java-Laufzeit)

Problem: Ihre Java-Funktion erzeugt falsche Ergebnisse oder es treten unter Last Rennbedingungen auf.

Ursache: Mehrere Threads führen die Handler-Methode gleichzeitig aus, und der gemeinsame Status ist nicht threadsicher.

Lösung:

- Verwenden Sie `AtomicInteger` or `AtomicLong` für Zähler anstelle von primitiven Typen
- Ersetze durch `HashMap` `ConcurrentHashMap`
- Zum `Collections.synchronizedList()` Einwickeln verwenden `ArrayList`
- Wird `ThreadLocal` für einen anforderungsspezifischen Status verwendet
- Greifen Sie IDs vom Lambda Context-Objekt aus auf die Ablaufverfolgung zu, nicht über Umgebungsvariablen

Eine ausführliche Anleitung finden Sie in der Dokumentation Java Runtime for Lambda Managed Instances.

Probleme mit der Statusisolierung (Laufzeit von Node.js)

Problem: Ihre Funktion Node.js gibt Daten aus verschiedenen Anfragen zurück oder es kommt zu Datenbeschädigungen.

Ursache: Globale Variablen werden von mehreren gleichzeitigen Aufrufen desselben Worker-Threads gemeinsam genutzt. Wenn asynchrone Operationen die Kontrolle übernehmen, können andere Aufrufe den gemeinsamen Status ändern.

Lösung:

- Installation und Verwendung `@aws/lambda-invoke-store` für alle anforderungsspezifischen Zustände
- Ersetzen Sie globale Variablen durch und `InvokeStore.set()` `InvokeStore.get()`
- Verwenden Sie eindeutige Dateinamen in `/tmp` mit Anfrage IDs
- Access Trace IDs mit Umgebungsvariablen `InvokeStore.getXRayTraceId()` anstelle von Umgebungsvariablen

Eine ausführliche Anleitung finden Sie in der Dokumentation Node.js Runtime for Lambda Managed Instances.

Dateikonflikte (Python-Laufzeit)

Problem: Ihre Python-Funktion liest falsche Daten aus Dateien in`/tmp`.

Ursache: Das `/tmp` Verzeichnis wird von mehreren Prozessen gemeinsam genutzt. Gleichzeitige Schreibvorgänge in dieselbe Datei können zu Datenbeschädigungen führen.

Lösung:

- Verwenden Sie bei Anfrage IDs eindeutige Dateinamen: `/tmp/request_{context.request_id}.txt`
- Verwenden Sie das Sperren von Dateien mit `fcntl.flock()` für gemeinsam genutzte Dateien
- Bereinigen Sie temporäre Dateien mit „`os.remove()`Nach dem Gebrauch“

Eine ausführliche Anleitung finden Sie in der Dokumentation zur Python-Laufzeit für Lambda Managed Instances.

Leistungsprobleme

Hohe Speicherauslastung

Problem: Bei Ihren Funktionen kommt es zu hoher Speicherauslastung oder zu out-of-memory Fehlern.

Ursache: Jede gleichzeitige Anforderung in Python wird in einem separaten Prozess mit eigenem Speicherplatz ausgeführt. Die gesamte Speichernutzung entspricht dem Speicher pro Prozess multipliziert mit gleichzeitigen Prozessen.

Lösung:

- Überwachen Sie die Metrik in `MemoryUtilization` CloudWatch
- Reduzieren Sie die `MaxConcurrency` Einstellung, wenn sich die Speichernutzung dem Speicherlimit der Funktion nähert
- Erhöhen Sie die Speicherzuweisung für Funktionen, um eine höhere Parallelität zu unterstützen
- Optimieren Sie die Speichernutzung, indem Sie Daten bei Bedarf statt während der Initialisierung laden

Inkonsistente Leistung

Problem: Die Funktionsleistung variiert zwischen Aufrufen erheblich.

Ursache: Lambda kann je nach Verfügbarkeit unterschiedliche Instanztypen auswählen, oder Funktionen werden möglicherweise auf Instanzen mit unterschiedlicher Ressourcenverfügbarkeit ausgeführt.

Lösung:

- Zulässige Instanztypen angeben: Wenn Sie bestimmte Leistungsanforderungen haben, konfigurieren Sie zulässige Instance-Typen in Ihrem Kapazitätsanbieter, um die Instance-Typen einzuschränken, die Lambda auswählen kann.
- Überwachen Sie Metriken auf Instanzebene: Verfolgen Sie `CPUUtilization` und verfolgen Sie sie `MemoryUtilization` auf Ebene des Kapazitätsanbieters, um Ressourcenbeschränkungen zu identifizieren.
- Überprüfen Sie `vCPUAvailable` die Kapazitätskennzahlen: Stellen Sie sicher`MemoryAvailable`, dass auf Ihren Instances ausreichend Ressourcen verfügbar sind.

Probleme mit dem Kapazitätsanbieter

Die Funktionsversion kann nicht AKTIV werden

Problem: Ihre Funktionsversion befindet sich nach der Veröffentlichung weiterhin im Status „Ausstehend“.

Ursache: Lambda startet Managed Instances und startet Ausführungsumgebungen. Dieser Vorgang benötigt Zeit, insbesondere bei der ersten Funktionsversion eines neuen Kapazitätsanbieters.

Lösung:

Warten Sie, bis Lambda den Initialisierungsprozess abgeschlossen hat. Lambda startet standardmäßig drei Instances aus Gründen der AZ-Resilienz und startet drei Ausführungsumgebungen, bevor Ihre Funktionsversion als AKTIV markiert wird. Dies dauert in der Regel mehrere Minuten.

Der Kapazitätsanbieter kann nicht gelöscht werden

Problem: Beim Versuch, einen Kapazitätsanbieter zu löschen, wird eine Fehlermeldung angezeigt.

Ursache: Sie können keinen Kapazitätsanbieter löschen, an den Funktionsversionen angehängt sind.

Lösung:

1. Identifizieren Sie mit der `ListFunctionVersionsByCapacityProvider` API alle Funktionsversionen, die den Kapazitätsanbieter verwenden.
2. Löschen oder aktualisieren Sie diese Funktionsversionen, um die Zuordnung des Kapazitätsanbieters zu entfernen.
3. Versuchen Sie erneut, den Kapazitätsanbieter zu löschen.

Generische Fehlermeldungen bei der Veröffentlichung von Funktionen

Problem: Beim Veröffentlichen von Funktionen treten generische Fehlermeldungen wie „Beim Publizieren ist ein interner Fehler aufgetreten“ auf.

Lösung:

- Überprüfen Sie die IAM-Berechtigungen: Stellen Sie sicher, dass Sie über die entsprechenden `lambda:PassCapacityProvider` Berechtigungen für den Kapazitätsanbieter verfügen, den Sie verwenden möchten.

- Überprüfen Sie die Konfiguration des Kapazitätsanbieters: Stellen Sie mithilfe der `GetCapacityProvider` API sicher, dass sich Ihr Kapazitätsanbieter im Status AKTIV befindet.
- Überprüfen Sie die VPC-Konfiguration: Stellen Sie sicher, dass die in Ihrem Kapazitätsanbieter angegebenen Subnetze und Sicherheitsgruppen korrekt konfiguriert und zugänglich sind.
- AWS CloudTrail Protokolle überprüfen: In den CloudTrail Protokollen finden Sie detaillierte Fehlerinformationen zu dem fehlgeschlagenen Vorgang.

Probleme bei der Überwachung und Beobachtbarkeit

Fehlende Metriken CloudWatch

Problem: Sie sehen keine erwarteten Kennzahlen CloudWatch für Ihren Kapazitätsanbieter oder Ihre Funktionen.

Ursache: Metriken werden in Intervallen von 5 Minuten veröffentlicht. Bei neuen Kapazitätsanbietern oder Funktionen sind Metriken möglicherweise nicht sofort verfügbar.

Lösung:

Warten Sie nach der Veröffentlichung einer Funktionsversion mindestens 5-10 Minuten, bevor Sie erwarten, dass Metriken in angezeigt werden CloudWatch. Vergewissern Sie sich, dass Sie sich den richtigen Namespace (`AWS/Lambda`) und die richtigen Dimensionen (`CapacityProviderNameFunctionName`, oder`InstanceType`) ansehen.

Protokolle können nicht gefunden CloudWatch werden

Problem: Ihre Funktion wird erfolgreich ausgeführt, aber Sie können keine CloudWatch Protokolle in Logs finden.

Ursache: Lambda Managed Instances werden in Ihrer VPC ausgeführt und benötigen Netzwerkkonnektivität, um Protokolle an Logs zu CloudWatch senden. Ohne die richtige VPC-Konnektivitätskonfiguration können Ihre Funktionen den Endpunkt des CloudWatch Logs-Dienstes nicht erreichen.

Lösung:

Konfigurieren Sie die VPC-Konnektivität, damit Ihre Funktionen CloudWatch Protokolle an Logs senden können. Sie haben drei Möglichkeiten:

Option 1: VPC-Endpunkt für CloudWatch Logs (für die Produktion empfohlen)

1. Öffnen Sie die Amazon VPC-Konsole unter console.aws.amazon.com/vpc/.
2. Wählen Sie im Navigationsbereich Endpunkte aus.
3. Wählen Sie Endpunkt erstellen aus.
4. Wählen Sie bei Service category (Servicekategorie) die Option AWS services (-Services) aus.
5. Wählen Sie als Servicename die Option (durch Ihre Region ersetzen) aus. `com.amazonaws.region.logs` `region` AWS
6. Wählen Sie für VPC die von Ihrem Kapazitätsanbieter verwendete VPC aus.
7. Wählen Sie für Subnetze die Subnetze aus, in denen Sie Endpunkt-Netzwerkschnittstellen erstellen möchten. Wählen Sie für eine hohe Verfügbarkeit Subnetze in mehreren Availability Zones aus.
8. Wählen Sie für Sicherheitsgruppen Sicherheitsgruppen aus, die eingehenden HTTPS-Verkehr (Port 443) aus der Sicherheitsgruppe Ihrer Funktion zulassen.
9. Aktivieren Sie Private DNS für den Endpunkt.
10. Wählen Sie Endpunkt erstellen aus.

Option 2: Öffentliches Subnetz mit Internet-Gateway

Wenn Ihr Kapazitätsanbieter öffentliche Subnetze verwendet, stellen Sie Folgendes sicher:

1. Ein Internet-Gateway ist an Ihre VPC angeschlossen
2. Die Routentabelle leitet den `0.0.0.0/0` Verkehr an das Internet-Gateway weiter
3. Sicherheitsgruppen lassen ausgehenden HTTPS-Verkehr auf Port 443 zu

Option 3: Privates Subnetz mit NAT-Gateway

Wenn Ihr Kapazitätsanbieter private Subnetze verwendet, stellen Sie Folgendes sicher:

1. Ein NAT-Gateway ist in einem öffentlichen Subnetz vorhanden
2. Die Routing-Tabelle des privaten Subnetzes leitet den `0.0.0.0/0` Verkehr an das NAT-Gateway weiter
3. Die öffentliche Subnetz-Routentabelle leitet den `0.0.0.0/0` Verkehr an ein Internet-Gateway weiter

4. Sicherheitsgruppen ermöglichen ausgehenden HTTPS-Verkehr auf Port 443

Eine ausführliche Anleitung zu VPC-Konnektivitätsoptionen finden Sie unter VPC-Konnektivität für Lambda Managed Instances.

Schwierigkeiten beim Korrelieren von Protokollen aus gleichzeitigen Anfragen

Problem: Die Protokolle verschiedener Anfragen sind ineinander verschachtelt, was es schwierig macht, einzelne Anfragen nachzuverfolgen.

Ursache: Das Verschachteln von Protokollen wird erwartet und ist in Systemen mit mehreren gleichzeitigen Vorgängen Standard.

Lösung:

- Strukturierte Protokollierung im JSON-Format verwenden: Geben Sie die Anforderungs-ID in alle Protokollanweisungen ein
- Java: Verwenden Sie Log4j mit, um die `ThreadContext` Anfrage-ID automatisch einzubeziehen
- Node.js: `console.log()` Mit JSON-Formatierung verwenden und einschließen `InvokeStore.getRequestId()`
- Python: Verwenden Sie das Standard-Logging-Modul mit JSON-Formatierung und fügen Sie hinzu `context.request_id`

Eine ausführliche Anleitung finden Sie auf den laufzeitspezifischen Dokumentationsseiten.

Weitere Hilfe

Wenn Sie nach dem Ausprobieren dieser Lösungen weiterhin Probleme haben:

1. CloudWatch Kennzahlen überprüfen: Überprüfen Sie die Kennzahlen zum Kapazitätsanbieter und zur Ausführungsumgebung, um Ressourcenbeschränkungen oder Skalierungsprobleme zu identifizieren.
2. AWS CloudTrail Protokolle überprüfen: In den CloudTrail Protokollen finden Sie detaillierte Informationen zu API-Aufrufen und Fehlern.
3. AWS Support kontaktieren: Wenn Sie das Problem nicht lösen können, wenden Sie sich mit Einzelheiten zur Konfiguration Ihres Kapazitätsanbieters, zur Funktionskonfiguration und zu den spezifischen Fehlermeldungen, auf die Sie stoßen, an den AWS Support.

Nächste Schritte

- Erfahren Sie mehr über Kapazitätsanbieter für Lambda Managed Instances
- Skalierung für Lambda Managed Instances verstehen
- Lesen Sie laufzeitspezifische Anleitungen für Java, Node.js und Python
- Überwachen Sie Lambda Managed Instances mit Metriken CloudWatch
- Informieren Sie sich über bewährte Methoden für Lambda Managed Instances

Lambda-Laufzeiten

Lambda unterstützt mehrere Sprachen durch die Verwendung von Laufzeiten. Eine Laufzeit bietet eine sprachspezifische Umgebung, die Aufrufereignisse, Kontextinformationen und Antworten zwischen Lambda und der Funktion weiterleitet. Sie können von Lambda bereitgestellte Laufzeiten verwenden oder Ihre eigenen erstellen.

Lambda ist unabhängig von Ihrer Wahl der Laufzeit. Für einfache Funktionen bieten interpretierte Sprachen wie Python und Node.js die schnellste Leistung. Bei Funktionen mit komplexeren Berechnungen lassen sich kompilierte Sprachen wie Java oft langsamer initialisieren, werden aber im Lambda-Handler schnell ausgeführt. Die Wahl der Laufzeit hängt auch von der Präferenz der Entwickler und der Vertrautheit mit der Sprache ab.

Jede größere Programmiersprachenversion verfügt über eine separate Laufzeit mit einer eindeutigen Laufzeitkennung, z. B. `nodejs24.x` oder `python3.14`. Um eine Funktion für die Verwendung einer neuen Hauptsprachversion zu konfigurieren, müssen Sie die Laufzeitkennung ändern. Da die Abwärtskompatibilität zwischen Hauptversionen AWS Lambda nicht garantiert werden kann, handelt es sich um einen kundenorientierten Vorgang.

Bei einer Funktion, die als Container-Image definiert ist, wählen Sie beim Erstellen des Container-Images eine Laufzeit und die Linux-Distribution aus. Um die Laufzeit zu ändern, erstellen Sie ein neues Container-Image.

Wenn Sie ein ZIP-Dateiarchiv für das Bereitstellungspaket verwenden, wählen Sie beim Erstellen der Funktion eine Laufzeit aus. Um die Laufzeit zu ändern, können Sie die Konfiguration Ihrer Funktion aktualisieren. Die Laufzeit ist mit einer der Amazon Linux-Distributionen gepaart. Die zugrunde liegende Ausführungsumgebung bietet zusätzliche Bibliotheken und Umgebungsvariablen, auf die Sie über Ihren Funktionscode zugreifen können.

Lambda ruft Ihre Funktion in einer Ausführungsumgebung auf. Die Ausführungsumgebung bietet eine sichere und isolierte Laufzeitumgebung, die die zum Ausführen Ihrer Funktion erforderlichen Ressourcen verwaltet. Lambda verwendet die Ausführungsumgebung eines vorherigen Aufrufs wieder, sofern vorhanden, oder kann eine neue Ausführungsumgebung erstellen.

Um andere Sprachen in Lambda zu verwenden, wie Go oder Rust, verwenden Sie eine reine OS-Laufzeit. Die Lambda-Ausführungsumgebung bietet eine Laufzeitschnittstelle zum Abrufen von Aufrufereignissen und Senden von Antworten. Sie können andere Sprachen bereitstellen, indem Sie zusammen mit Ihrem Funktionscode oder in einem Layer eine benutzerdefinierte Laufzeit implementieren.

Unterstützte Laufzeiten

In der folgenden Tabelle sind die unterstützten Lambda-Laufzeiten und ihre voraussichtlichen Ablauftermine aufgeführt. Wenn eine Laufzeit als veraltet gilt, können Sie für einen begrenzten Zeitraum weiterhin Funktionen erstellen und aktualisieren. Weitere Informationen finden Sie unter the section called “Unterstützung der Verwendung nach der Einstellung”. Die Tabelle enthält die aktuell angenommenen Ablaufdaten für die Laufzeit, basierend auf unserer the section called “Richtlinie für den Laufzeitablauf”. Diese Daten dienen zu Planungszwecken und können sich ändern.

> **Important**
>
> Das Ende der Lebensdauer von Amazon Linux 2 ist für den 30. Juni 2026 geplant. Lambda-Laufzeiten und Container-Basis-Images für Java 8 (AL2), Java 11, Java 17, Python 3.10, Python 3.11 und provided.al2 erhalten bis zu den in der Tabelle unten aufgeführten Verfallsdaten weiterhin Patches für kritische und ausgewählte wichtige Amazon Linux 2-Sicherheitsprobleme sowie Sprach-Runtime-Patches.
> Wir empfehlen Kunden, so bald wie möglich ein Upgrade auf eine Amazon Linux 2023-basierte Runtime durchzuführen. Kunden, die auf Java 21 oder Java 25 aktualisieren, können AWS Transform Custom verwenden, um sie bei diesen Upgrades zu unterstützen. Für Kunden, die ihre Java-Version nicht aktualisieren können, planen wir, Amazon Linux 2023-basierte Laufzeiten für Java 8, Java 11 und Java 17 vor Ende des zweiten Quartals 2026 zu veröffentlichen.

Name	ID	Betriebssystem	Datum der Veraltung	Blockfunktion erstellen	Blockfunktion aktualisieren
Node.js 24	`nodejs24.x`	Amazon Linux 2023	30. April 2028	1. Juni 2028	1. Juli 2028
Node.js 22	`nodejs22.x`	Amazon Linux 2023	30. April 2027	1. Juni 2027	1. Juli 2027
Node.js 20	`nodejs20.x`	Amazon Linux 2023	30. April 2026	31. August 2026	30. September 2026

Name	ID	Betriebssystem	Datum der Veraltung	Blockfunktion erstellen	Blockfunktion aktualisieren
Python 3.14	`python3.14`	Amazon Linux 2023	30. Juni 2029	31. Juli 2029	31. August 2029
Python 3.13	`python3.13`	Amazon Linux 2023	30. Juni 2029	31. Juli 2029	31. August 2029
Python 3.12	`python3.12`	Amazon Linux 2023	31. Oktober 2028	30. November 2028	10. Januar 2029
Python 3.11	`python3.11`	Amazon Linux 2	30. Juni 2027	31. Juli 2027	31. August 2027
Python 3.10	`python3.10`	Amazon Linux 2	31. Oktober 2026	30. November 2026	15. Januar 2027
Java 25	`java25`	Amazon Linux 2023	30. Juni 2029	31. Juli 2029	31. August 2029
Java 21	`java21`	Amazon Linux 2023	30. Juni 2029	31. Juli 2029	31. August 2029
Java 17	`java17`	Amazon Linux 2	30. Juni 2027	31. Juli 2027	31. August 2027
Java 11	`java11`	Amazon Linux 2	30. Juni 2027	31. Juli 2027	31. August 2027
Java 8	`java8.al2`	Amazon Linux 2	30. Juni 2027	31. Juli 2027	31. August 2027
.NET 10	`dotnet10`	Amazon Linux 2023	14. November 2028	14. Dezember 2028	15. Januar 2029

Name	ID	Betriebssystem	Datum der Veraltung	Blockfunktion erstellen	Blockfunktion aktualisieren
.NET 9 (nur Container)	`dotnet9`	Amazon Linux 2023	10. November 2026	Nicht geplant	Nicht geplant
.NET 8	`dotnet8`	Amazon Linux 2023	10. November 2026	10. Dezember 2026	11. Januar 2027
Rubin 3.4	`ruby3.4`	Amazon Linux 2023	31. März 2028	30. April 2028	31. Mai 2028
Ruby 3.3	`ruby3.3`	Amazon Linux 2023	31. März 2027	30. April 2027	31. Mai 2027
Ruby 3.2	`ruby3.2`	Amazon Linux 2	31. März 2026	31. August 2026	30. September 2026
Reine OS-Laufzeit	`provided.al2023`	Amazon Linux 2023	30. Juni 2029	31. Juli 2029	31. August 2029
Reine OS-Laufzeit	`provided.al2`	Amazon Linux 2	31. Juli 2026	31. August 2026	30. September 2026

> **Note**
>
> Für neue Regionen wird Lambda keine Laufzeiten unterstützen, die innerhalb der nächsten sechs Monate veraltet sein werden.

Lambda hält verwaltete Laufzeiten und ihre entsprechenden Container-Basis-Images mit Patches und Unterstützung für kleinere Versionen auf dem neuesten Stand. Weitere Informationen finden Sie unter Lambda-Laufzeitaktualisierungen.

Um programmgesteuert mit anderen AWS-Services und Ressourcen aus Ihrer Lambda-Funktion zu interagieren, können Sie eine von verwenden. AWS SDKs Die Laufzeiten Node.js, Python und Ruby enthalten eine Version des AWS SDK. Um jedoch die volle Kontrolle über Ihre Abhängigkeiten zu behalten und die Abwärtskompatibilität bei automatischen Laufzeitaktualisierungen zu maximieren, empfehlen wir Ihnen, die von Ihrem Code verwendeten SDK-Module (zusammen mit allen Abhängigkeiten) immer in das Bereitstellungspaket Ihrer Funktion oder in eine Lambda-Ebene aufzunehmen.

Wir empfehlen, die in der Laufzeit enthaltene SDK-Version nur zu verwenden, wenn Sie keine zusätzlichen Pakete in Ihre Bereitstellung aufnehmen können. Zum Beispiel, wenn Sie Ihre Funktion mit dem Code-Editor der Lambda-Konsole oder mit Inline-Funktionscode in einer CloudFormation Vorlage erstellen.

Lambda aktualisiert regelmäßig die Versionen der in den Laufzeiten Node.js, Python und Ruby AWS SDKs enthaltenen. Informationen darüber, welche Version des AWS -SDKs in der von Ihnen verwendeten Laufzeit enthalten ist, finden Sie in den folgenden Abschnitten:

- In der Laufzeit enthaltene SDK-Versionen (Node.js)
- In der Laufzeit enthaltene SDK-Versionen (Python)
- In der Laufzeit enthaltene SDK-Versionen (Ruby)

Lambda unterstützt weiterhin die Programmiersprache Go, auch nachdem die Go 1.x-Laufzeit veraltet ist. Weitere Informationen finden Sie im Compute-Blog unter AWS Lambda Funktionen von der GO1.x-Laufzeit zur benutzerdefinierten Laufzeit auf Amazon Linux 2 migrieren.AWS

Alle unterstützten Lambda-Laufzeiten unterstützen sowohl x86_64- als auch arm64-Architekturen.

Neue Laufzeit-Versionen

Lambda stellt verwaltete Laufzeiten für neue Sprachversionen nur dann zur Verfügung, wenn die Version die LTS-Phase (Long-Term Support) des Veröffentlichungszyklus der Sprache erreicht. Zum Beispiel für den Node.js-Veröffentlichungszyklus, wenn die Version die Active LTS-Phase erreicht.

Bevor die Version die LTS-Phase erreicht, befindet sie sich noch in der Entwicklung und kann noch grundlegenden Änderungen unterliegen. Lambda wendet Laufzeit-Updates standardmäßig automatisch an. Wenn Sie also Änderungen an einer Laufzeitversion vornehmen, funktionieren Ihre Funktionen möglicherweise nicht mehr wie erwartet.

Lambda bietet keine verwalteten Laufzeiten für Sprachversionen, die nicht für die LTS-Veröffentlichung geplant sind.

Im Folgenden ist der geplante Startmonat für die kommenden Lambda-Laufzeiten aufgeführt. Diese Termine sind nur Richtwerte und können sich ändern.

- Ruby 3.5 — März 2026
- Java 8, 11 und 17 auf AL2 023 - Q2 2026
- Node.js 26 — November 2026
- Python 3.15 - November 2026

Richtlinie für den Laufzeitablauf

Lambda-Laufzeiten für ZIP-Dateiarchive werden um eine Kombination aus Betriebssystem, Programmiersprache und Softwarebibliotheken aufgebaut, die Wartungen und Sicherheitsupdates erfordern. Die Standard-Einstellungsrichtlinie von Lambda sieht vor, dass eine Laufzeit als veraltet eingestuft wird, wenn für eine wichtige Komponente der Laufzeit die langfristige Unterstützung (LTS) durch die Community ausläuft und keine Sicherheitsupdates mehr verfügbar sind. In den meisten Fällen handelt es sich dabei um die Sprachlaufzeit. In einigen Fällen kann eine Laufzeit jedoch als veraltet gelten, weil das Betriebssystem das LTS-Ende erreicht.

Wenn eine Runtime als veraltet gilt, AWS darf sie keine Sicherheitspatches oder Updates mehr auf diese Runtime anwenden, und Funktionen, die diese Runtime verwenden, haben keinen Anspruch mehr auf technischen Support. Solche veralteten Laufzeiten werden „wie sie sind“ ohne jegliche Garantie bereitgestellt und können Bugs, Fehler, Defekte oder andere Schwachstellen enthalten.

Weitere Informationen zur Verwaltung von Runtime-Upgrades und veralteten Versionen finden Sie in den folgenden Abschnitten und unter AWS Lambda Runtime-Upgrades verwalten im Compute-Blog.AWS

> **Important**
>
> Lambda verzögert gelegentlich die Einstellung einer Lambda-Laufzeit für einen begrenzten Zeitraum über das Ende der Unterstützung der Sprachversion hinaus, die von der Laufzeit unterstützt wird. Während dieses Zeitraums wendet Lambda nur Sicherheits-Patches auf das Laufzeit-Betriebssystem an. Lambda wendet keine Sicherheits-Patches auf Laufzeiten von Programmiersprachen an, nachdem diese das Ende der Unterstützung erreicht haben.

Modell der geteilten Verantwortung

Lambda ist für das Kuratieren und Veröffentlichen von Sicherheitsupdates für alle unterstützten verwalteten Laufzeiten und Container-Basisimages verantwortlich. Standardmäßig wendet Lambda diese Aktualisierungen automatisch auf Funktionen mit verwalteten Laufzeiten an. Wenn die Standardeinstellung für automatische Laufzeit-Updates geändert wurde, finden Sie weitere Informationen im Laufzeitmanagementkontrollen Modell der geteilten Verantwortung. Bei Funktionen, die mit Container-Images bereitgestellt werden, sind Sie dafür verantwortlich, das Container-Image Ihrer Funktion aus dem neuesten Basis-Image neu zu erstellen und das Container-Image erneut bereitzustellen.

Wenn eine Laufzeit veraltet ist, erlischt die Verantwortung von Lambda für die Aktualisierung der verwalteten Laufzeit- und Container-Basis-Images. Sie sind dafür verantwortlich, Ihre Funktionen so zu aktualisieren, dass sie ein unterstütztes Laufzeit- oder Basis-Image verwenden.

In allen Fällen sind Sie dafür verantwortlich, Aktualisierungen an Ihrem Funktionscode, einschließlich seiner Abhängigkeiten, vorzunehmen. Ihre Verantwortlichkeiten im Rahmen des Modells der gemeinsamen Verantwortung sind in der folgenden Tabelle zusammengefasst.

Laufzeit-Lebenszyklusphase	Verantwortungen von Lambda	Ihre Aufgaben
Unterstützte verwaltete Laufzeit	Stellen Sie regelmäßige Laufzeit-Updates mit Sicherheitspatches und anderen Updates bereit. Wenden Sie Laufzeit-Updates standardmäßig automatisch an (Informationen zu nicht standardmäßigen Verhaltensweisen finden Sie unter the section called “Modi der Laufzeitaktualisierung”).	Aktualisieren Sie Ihren Funktionscode, einschließlich der Abhängigkeiten, um Schwachstellen zu schließen.
Unterstütztes Container-Image	Regelmäßige Aktualisierung des Container-Basis-Images mit Sicherheitspatches und anderen Updates.	Aktualisieren Sie Ihren Funktionscode, einschließlich der Abhängigkeiten, um Schwachstellen zu schließen.

Laufzeit-Lebenszyklusphase	Verantwortungen von Lambda	Ihre Aufgaben
		Erstellen Sie Ihr Container-Image regelmäßig neu und stellen Sie es erneut bereit, indem Sie das neueste Basis-Image verwenden.
Verwaltete Laufzeit nähert sich dem Verfall	Informieren Sie Kunden per Dokumentation, E-Mail und, Health Dashboard bevor Runtime veraltet ist. Trusted Advisor Die Verantwortung für Laufzeit-Updates endet mit dem Zeitpunkt, an dem sie veraltet sind.	Überwachen Sie die Lambda-Dokumentation Health Dashboard, E-Mails oder Informationen Trusted Advisor zu veralteten Laufzeiten. Führen Sie ein Upgrade von Funktionen auf eine unterstüt zte Laufzeit durch, bevor die vorherige Laufzeit veraltet ist.
Container-Image nähert sich der Veralterung	Benachrichtigungen über veraltete Versionen sind für Funktionen, die Container -Images verwenden, nicht verfügbar. Die Verantwortung für Container-Basis-Images-Upda tes endet mit dem Zeitpunkt, an dem sie veraltet sind.	Beachten Sie die Zeitpläne für veraltete Versionen und aktualisieren Sie Funktionen auf ein unterstütztes Basis-Image, bevor das vorherige Image veraltet ist.

Unterstützung der Verwendung nach der Einstellung

Wenn eine Runtime veraltet ist, AWS dürfen keine Sicherheitspatches oder Updates mehr auf diese Runtime angewendet werden, und Funktionen, die diese Runtime verwenden, haben keinen Anspruch mehr auf technischen Support. Sie können Ihre Funktionen zwar unbegrenzt weiter aufrufen, es wird jedoch AWS dringend empfohlen, zu einer unterstützten Runtime zu migrieren. Veraltete Laufzeiten werden „wie sie sind“ ohne jegliche Garantie bereitgestellt und können Bugs,

Fehler, Defekte oder andere Schwachstellen enthalten. Bei Funktionen, die eine veraltete Laufzeit verwenden, kann es zu Leistungseinbußen oder anderen Problemen kommen, wie z. B. dem Ablauf eines Zertifikats, was dazu führen kann, dass sie nicht mehr richtig funktionieren.

Sie können eine Funktion so aktualisieren, dass sie jederzeit eine neuere unterstützte Laufzeit verwendet, nachdem eine Laufzeit veraltet ist. Wir empfehlen, Ihre Funktion mit der neuen Laufzeit zu testen, bevor Sie Änderungen in Produktionsumgebungen vornehmen. Sie können nicht zur veralteten Laufzeit zurückkehren, nachdem Funktionsaktualisierungen blockiert wurden. Wir empfehlen die Verwendung von Funktionsversionen und Aliasnamen, um eine sichere Bereitstellung mit Rollback zu ermöglichen.

Die folgende Zeitleiste beschreibt, was geschieht, wenn eine Laufzeit veraltet ist:

Laufzeit-Lebenszyklusphase	Wann	Was
Frist für Hinweis zur Einstellung	Mindestens 180 Tage vor Ablauf der Einstellung	• AWS sendet Benachrichtigungen per E-Mail und dann Health Dashboard an Konten, deren Version über Funktionen verfügt, die diese Runtime verwenden. `$LATEST` • Betroffene Funktionen sind auch auf der Registerkarte mit Health Dashboard – Geplante Änderungen und in der Prüfung für AWS Trusted Advisor – Veraltete Laufzeiten aufgeführt.
Ablehnung	Datum der Veraltung	• AWS wendet möglicherweise keine Sicherheitsupdates oder andere Updates mehr an. • Funktionen sind nicht mehr für den technischen Support berechtigt. • Sie können mit der veralteten Laufzeit in der Lambda-Konsole keine Funktionen mehr erstellen oder aktualisieren. Sie können weiterhin Funktionen über AWS CLI AWS SAM, oder erstellen und aktualisieren CloudFormation.

Laufzeit-Lebenszyklusphase	Wann	Was
Blockfunktion erstellen	Mindestens 30 Tage nach der Einstellung	• Lambda beginnt, die Erstellung neuer Funktionen zu blockieren. • Sie können den Code und die Konfiguration vorhandener Funktionen weiterhin über AWS CLI AWS SAM, oder aktualisieren CloudFormation.
Blockfunktion aktualisieren	Mindestens 60 Tage nach der Einstellung	• Lambda beginnt, die Aktualisierung des Codes und der Konfiguration für bestehende Funktionen zu blockieren. • Sie können die Konfiguration der Funktion immer noch auf eine unterstützte Laufzeit aktualisieren. Ein Rollback zur veralteten Laufzeit kann jedoch blockiert werden.

> **Note**
>
> Bei einigen Laufzeiten verschiebt AWS sich das block-function-update Datum block-function-create und die Datumsangaben über die üblichen 30 und 60 Tage nach der Deprecation hinaus. AWS hat diese Änderung als Reaktion auf Kundenfeedback vorgenommen, damit Sie mehr Zeit haben, Ihre Funktionen zu aktualisieren. In den Tabellen in the section called “Unterstützte Laufzeiten” und the section called “Veraltete Laufzeitumgebungen” finden Sie die Daten für Ihre Laufzeit. Lambda beginnt nicht vor den in diesen Tabellen angegebenen Terminen mit dem Blockieren von Funktionserstellungen oder -aktualisierungen.

Empfangen von Benachrichtigungen über veraltete Laufzeitversionen

Wenn sich eine Laufzeit ihrem Verfallsdatum nähert, sendet Lambda Ihnen eine E-Mail-Benachrichtigung, falls Funktionen in Ihrer Umgebung diese Laufzeit AWS-Konto verwenden. Benachrichtigungen werden auch in und in angezeigt. Health Dashboard AWS Trusted Advisor

- Empfangen von E-Mail-Benachrichtigungen:

  Lambda sendet Ihnen mindestens 180 Tage, bevor eine Laufzeit veraltet ist, eine E-Mail-Warnung. In dieser E-Mail sind die $LATEST-Versionen aller Funktionen aufgeführt, die die Laufzeit verwenden. Eine vollständige Liste der betroffenen Funktionsversionen finden Sie unter Trusted Advisor oderthe section called “Daten über Funktionen zur Laufzeit abrufen”.

  Lambda sendet eine E-Mail-Benachrichtigung an Ihren AWS-Konto primären Kontokontakt. Informationen zum Anzeigen oder Aktualisieren der E-Mail-Adressen in Ihrem Konto finden Sie in der allgemeinen AWS -Referenz unter Updating contact information.

- Empfangen von Benachrichtigungen über: Health Dashboard

  Health Dashboard Zeigt mindestens 180 Tage, bevor eine Laufzeit veraltet ist, eine Benachrichtigung an. Benachrichtigungen werden auf der Seite Ihr Kontostatus unter Andere Benachrichtigungen angezeigt. Auf der Registerkarte Betroffene Ressourcen in der Benachrichtigung sind die $LATEST-Versionen aller Funktionen aufgeführt, die die Laufzeit verwenden.

  > **Note**
  >
  > Eine vollständige up-to-date Liste der betroffenen Funktionsversionen finden Sie unter Trusted Advisor oder. the section called “Daten über Funktionen zur Laufzeit abrufen”

  Health Dashboard Benachrichtigungen laufen 90 Tage ab, nachdem die betroffene Runtime veraltet ist.

- Verwenden AWS Trusted Advisor

  Trusted Advisor zeigt mindestens 180 Tage, bevor eine Laufzeit veraltet ist, eine Benachrichtigung an. Benachrichtigungen werden auf der Seite Sicherheit angezeigt. Eine Liste der betroffenen Funktionen wird unter AWS Lambda -Funktionen, die veraltete Laufzeiten verwenden angezeigt. Diese Liste von Funktionen zeigt sowohl die $LATEST als auch veröffentlichte Versionen und wird automatisch aktualisiert, um den aktuellen Status Ihrer Funktionen widerzuspiegeln.

  Sie können wöchentliche E-Mail-Benachrichtigungen auf der Trusted Advisor Seite „Einstellungen“ der Trusted Advisor Konsole aktivieren.

Veraltete Laufzeitumgebungen

Die folgenden Laufzeiten haben das Ende der Unterstützung erreicht:

Name	ID	Betriebssystem	Datum der Veraltung	Blockfunktion erstellen	Blockfunktion aktualisieren
Python 3.9	`python3.9`	Amazon Linux 2	15. Dezember 2025	31. August 2026	30. September 2026
Node.js 18	`nodejs18.x`	Amazon Linux 2	1. Sept. 2025	31. August 2026	30. September 2026
.NET 6	`dotnet6`	Amazon Linux 2	20. Dezember 2	31. August 2026	30. September 2026
Python 3.8	`python3.8`	Amazon Linux 2	14. Oktober 202	31. August 2026	30. September 2026
Node.js 16	`nodejs16.x`	Amazon Linux 2	12. Juni 2024	31. August 2026	30. September 2026
.NET 7 (nur Container)	`dotnet7`	Amazon Linux 2	14. Mai 2024	–	–
Java 8	`java8`	Amazon Linux	8. Januar 2024	8. Februar 2024	30. September 2026
Go 1.x	`go1.x`	Amazon Linux	8. Januar 2024	8. Februar 2024	30. September 2026

Name	ID	Betriebssystem	Datum der Veraltung	Blockfunktion erstellen	Blockfunktion aktualisieren
Reine OS-Laufzeit	`provided`	Amazon Linux	8. Januar 2024	8. Februar 202	30. September 2026
Ruby 2.7	`ruby2.7`	Amazon Linux 2	7. Dezember 2(	9. Januar 2024	30. September 2026
Node.js 14	`nodejs14.x`	Amazon Linux 2	4. Dezember 2(	9. Januar 2024	30. September 2026
Python 3.7	`python3.7`	Amazon Linux	4. Dezember 2(	9. Januar 2024	30. September 2026
.NET Core 3.1	`dotnetcore3.1`	Amazon Linux 2	3. Apr 2023	3. Apr 2023	3. Mai 2023
Node.js 12	`nodejs12.x`	Amazon Linux 2	31. März 2023	31. März 2023	30. Apr 2023
Python 3.6	`python3.6`	Amazon Linux	18. Juli 2022	18. Juli 2022	29. August 2022
.NET 5 (nur Container)	`dotnet5.0`	Amazon Linux 2	10. Mai 2022	–	–
.NET Core 2.1	`dotnetcore2.1`	Amazon Linux	5. Januar 2022	5. Januar 2022	13. Apr 2022
Node.js 10	`nodejs10.x`	Amazon Linux 2	30. Juli 2021	30. Juli 2021	14. Februar 2022
Ruby 2.5	`ruby2.5`	Amazon Linux	30. Juli 2021	30. Juli 2021	31. März 2022

Name	ID	Betriebssystem	Datum der Veraltung	Blockfunktion erstellen	Blockfunktion aktualisieren
Python 2.7	`python2.7`	Amazon Linux	15. Juli 2021	15. Juli 2021	30. Mai 2022
Node.js 8.10	`nodejs8.10`	Amazon Linux	6. März 2020	4. Februar 2020	6. März 2020
Node.js 4.3	`nodejs4.3`	Amazon Linux	5. März 2020	3. Februar 2020	5. März 2020
Node.js 4.3 edge	`nodejs4.3-edge`	Amazon Linux	5. März 2020	31. März 2019	30. Apr 2019
Node.js 6.10	`nodejs6.10`	Amazon Linux	12. August 201	12. Juli 2019	12. August 2019
.NET Core 1.0	`dotnetcore1.0`	Amazon Linux	27. Juni 2019	30. Juni 2019	30. Juli 2019
.NET Core 2.0	`dotnetcore2.0`	Amazon Linux	30. Mai 2019	30. Apr 2019	30. Mai 2019
Node.js 0.10	`nodejs`	Amazon Linux	30. August 2016	30. September 2016	31. Oktober 2016

In fast allen Fällen ist das end-of-life Datum einer Sprachversion oder eines Betriebssystems weit im Voraus bekannt. Die folgenden Links enthalten end-of-life Zeitpläne für jede Sprache, die Lambda als verwaltete Laufzeit unterstützt.

Richtlinien für die Unterstützung von Sprache und Framework-Bedingungen

- Node.js – github.com
- Python – devguide.python.org
- Ruby – www.ruby-lang.org
- Java — www.oracle.com und Corretto FAQs

- Go – golang.org
- .NET – dotnet.microsoft.com

Verstehen, wie Lambda Laufzeit-Versionsupdates verwaltet

Lambda hält jede verwaltete Laufzeit mit Sicherheitsupdates, Fehlerbehebungen, neuen Funktionen, Leistungsverbesserungen und Unterstützung für Nebenversionen auf dem neuesten Stand. Diese Laufzeitaktualisierungen werden als Laufzeitversionen veröffentlicht. Lambda wendet Laufzeitaktualisierungen auf Funktionen an, indem es die Funktion von einer früheren Laufzeitversion auf eine neue Laufzeitversion migriert.

Für Funktionen, die verwaltete Laufzeiten verwenden, wendet Lambda Laufzeitaktualisierungen standardmäßig automatisch an. Mit automatischen Laufzeitaktualisierungen übernimmt Lambda den operativen Aufwand für das Patchen der Laufzeitversionen. Für die meisten Kunden sind automatische Aktualisierungen die richtige Wahl. Sie können dieses Standardverhalten ändern, indem Sie die Einstellungen für die Laufzeitverwaltung konfigurieren.

Lambda veröffentlicht auch jede neue Laufzeitversion als Container-Image. Um Laufzeitversionen für Container-basierte Funktionen zu aktualisieren, müssen Sie ein neues Container-Image aus dem aktualisierten Basis-Image erstellen und Ihre Funktion erneut bereitstellen.

Jeder Laufzeitversion ist eine Versionsnummer und ein ARN (Amazon Resource Name) zugeordnet. Laufzeitversionsnummern verwenden ein von Lambda definiertes Nummerierungsschema, unabhängig von den Versionsnummern, die die Programmiersprache verwendet. Laufzeit-Versionsnummern sind nicht immer fortlaufend. Auf Version 42 könnte beispielsweise Version 45 folgen. Der Laufzeitversions-ARN ist eine eindeutige Kennung für jede Laufzeitversion. Sie können den ARN der aktuellen Laufzeitversion Ihrer Funktion in der Lambda-Konsole oder in der `INIT_START`-Zeile der Funktionsprotokolle anzeigen.

Laufzeitversionen sollten nicht mit Laufzeitkennungen verwechselt werden. Jede Laufzeit hat eine eindeutige Laufzeitkennung, z. B. `python3.14` oder `nodejs24.x`. Diese entsprechen den jeweiligen Hauptversionen der Programmiersprachen. Laufzeitversionen beschreiben die Patch-Version einer einzelnen Laufzeit.

> **Note**
>
> Der ARN für dieselbe Runtime-Versionsnummer kann je nach AWS-Regionen CPU-Architektur variieren.

Themen

- Abwärtskompatibilität

- Modi der Laufzeitaktualisierung
- Zweiphasiges Rollout der Laufzeitversion
- Konfigurieren der Einstellungen für die Lambda-Laufzeitverwaltung
- Rollback einer Lambda-Laufzeitversion
- Identifizieren von Änderungen der Lambda-Laufzeitversion
- Verstehen des Modells der geteilten Verantwortung für die Verwaltung der Lambda-Laufzeit
- Kontrolle der Lambda-Laufzeitaktualisierungsberechtigungen für Anwendungen mit hoher Konformität

Abwärtskompatibilität

Lambda ist bestrebt, Laufzeitaktualisierungen bereitzustellen, die mit vorhandenen Funktionen abwärtskompatibel sind. Wie beim Software-Patching gibt es jedoch seltene Fälle, in denen sich eine Laufzeitaktualisierung negativ auf eine vorhandene Funktion auswirken kann. Beispielsweise können Sicherheits-Patches ein zugrunde liegendes Problem mit einer vorhandenen Funktion aufdecken, das vom vorherigen, unsicheren Verhalten abhängt.

Beim Erstellen und Bereitstellen Ihrer Funktion ist es wichtig zu verstehen, wie Sie Ihre Abhängigkeiten verwalten, um mögliche Inkompatibilitäten mit einem zukünftigen Laufzeit-Update zu vermeiden. Nehmen wir zum Beispiel an, dass Ihre Funktion von Paket A abhängig ist, das wiederum von Paket B abhängt. Beide Pakete sind in der Lambda-Laufzeit enthalten (sie könnten beispielsweise Teile des SDK oder seiner Abhängigkeiten oder Teile der Laufzeitsystembibliotheken sein).

Betrachten Sie folgende Szenarien:

Bereitstellung	Patching-kompatibel	Grund
• Paket A: Ab Laufzeit verwenden • Paket B: Ab Laufzeit verwenden	Ja	Zukünftige Laufzeit-Updates für die Pakete A und B sind abwärtskompatibel.
• Paket A: Im Bereitste llungspaket	Ja	Ihre Bereitstellung hat Vorrang, sodass zukünftig e Laufzeit-Updates für

Bereitstellung	Patching-kompatibel	Grund
• Paket B: Im Bereitstellungspaket		die Pakete A und B keine Auswirkungen haben.
• Paket A: Im Bereitstellungspaket • Paket B: Ab Laufzeit verwenden	Ja*	Zukünftige Laufzeit-Updates für das Paket B sind abwärtskompatibel. * Wenn A und B eng miteinander verbunden sind, können Kompatibilitätsprobleme auftreten. Beispielsweise sollten die `botocore` Pakete `boto3` und im AWS SDK für Python zusammen bereitgestellt werden.
• Paket A: Ab Laufzeit verwenden • Paket B: Im Bereitstellungspaket	Nein	Zukünftige Laufzeit-Updates für Paket A erfordern möglicherweise eine aktualisierte Version von Paket B. Die bereitgestellte Version von Paket B hat jedoch Vorrang und ist möglicherweise nicht mit der aktualisierten Version von Paket A aufwärtskompatibel.

Befolgen Sie die folgenden bewährten Methoden, um die Kompatibilität mit zukünftigen Laufzeit-Updates aufrechtzuerhalten:

- Wenn möglich, packen Sie alle Abhängigkeiten: Nehmen Sie alle erforderlichen Bibliotheken, einschließlich des AWS SDK und seiner Abhängigkeiten, in Ihr Bereitstellungspaket auf. Dadurch wird ein stabiler, kompatibler Satz von Komponenten gewährleistet.
- Verwenden Sie runtime-provided SDKs sparsam: Verlassen Sie sich nur dann auf das Runtime-bereitgestellte SDK, wenn Sie keine zusätzlichen Pakete hinzufügen können (z. B. wenn

Sie den Lambda-Konsolen-Code-Editor oder Inline-Code in einer Vorlage verwenden). AWS CloudFormation

- Vermeiden Sie das Überschreiben von Systembibliotheken: Stellen Sie keine benutzerdefinierten Betriebssystembibliotheken bereit, die mit zukünftigen Laufzeit-Updates in Konflikt geraten könnten.

Modi der Laufzeitaktualisierung

Lambda ist bestrebt, Laufzeitaktualisierungen bereitzustellen, die mit vorhandenen Funktionen abwärtskompatibel sind. Wie beim Software-Patching gibt es jedoch seltene Fälle, in denen sich eine Laufzeitaktualisierung negativ auf eine vorhandene Funktion auswirken kann. Beispielsweise können Sicherheits-Patches ein zugrunde liegendes Problem mit einer vorhandenen Funktion aufdecken, das vom vorherigen, unsicheren Verhalten abhängt. Lambda-Laufzeitverwaltungskontrollen tragen dazu bei, das Risiko von Auswirkungen auf Ihre Workloads im seltenen Fall einer Inkompatibilität einer Laufzeitversion zu verringern. Für jede Funktionsversion ($LATEST oder veröffentlichte Version) können Sie einen der folgenden Laufzeitaktualisierungsmodi wählen:

- Automatisch (Standard) – Automatische Aktualisierung auf die neueste und sicherste Laufzeitversion mit Zweiphasiges Rollout der Laufzeitversion. Diesen Modus empfehlen wir den meisten Kunden, damit Sie immer von Laufzeitaktualisierungen profitieren.
- Funktion aktualisieren – Aktualisieren Sie die Laufzeit Ihrer Funktion auf die neueste und sicherste Laufzeitversion. Wenn Sie Ihre Funktion aktualisieren, aktualisiert Lambda die Laufzeit Ihrer Funktion auf die neueste und sicherste Laufzeitversion. Dieser Ansatz synchronisiert Laufzeitaktualisierungen mit Funktionsbereitstellungen, sodass Sie die Kontrolle darüber haben, wann Lambda Laufzeitaktualisierungen anwendet. In diesem Modus können Sie seltene Inkompatibilitäten bei Laufzeitaktualisierungen frühzeitig erkennen und beheben. Wenn Sie diesen Modus verwenden, müssen Sie Ihre Funktionen regelmäßig aktualisieren, um deren Laufzeit auf dem neuesten Stand zu halten.
- Manuell – Aktualisieren Sie Ihre Laufzeitversion manuell. Sie geben in Ihrer Funktionskonfiguration eine Laufzeitversion an. Die Funktion verwendet diese Laufzeitversion unbegrenzt. In dem seltenen Fall, dass eine neue Laufzeitversion mit einer vorhandenen Funktion nicht kompatibel ist, können Sie diesen Modus verwenden, um Ihre Funktion auf eine frühere Laufzeitversion zurückzusetzen. Wir raten davon ab, den Modus Manual (Manuell) zu verwenden, um Laufzeitkonsistenz über Bereitstellungen hinweg zu erreichen. Weitere Informationen finden Sie unter Rollback einer Lambda-Laufzeitversion.

Die Verantwortung für das Anwenden von Laufzeitaktualisierungen auf Ihre Funktionen hängt davon ab, welchen Laufzeitaktualisierungsmodus Sie auswählen. Weitere Informationen finden Sie unter Verstehen des Modells der geteilten Verantwortung für die Verwaltung der Lambda-Laufzeit.

Zweiphasiges Rollout der Laufzeitversion

Lambda führt neue Laufzeitversionen in der folgenden Reihenfolge ein:

1. In der ersten Phase wendet Lambda die neue Laufzeitversion an, sobald Sie eine Funktion erstellen oder aktualisieren. Eine Funktion wird aktualisiert, wenn Sie die oder API-Operationen aufrufen. UpdateFunctionCodeUpdateFunctionConfiguration
2. In der zweiten Phase aktualisiert Lambda alle Funktionen, die den Auto (Automatischen) Laufzeitaktualisierungsmodus verwenden und die noch nicht auf die neue Laufzeitversion aktualisiert wurden.

Die Gesamtdauer des Rollout-Prozesses hängt von mehreren Faktoren ab, einschließlich des Schweregrads von Sicherheits-Patches, die in der Laufzeitaktualisierung enthalten sind.

Wenn Sie Ihre Funktionen aktiv entwickeln und bereitstellen, werden Sie höchstwahrscheinlich in der ersten Phase neue Laufzeitversionen abrufen. Dadurch werden Laufzeitaktualisierungen mit Funktionsaktualisierungen synchronisiert. In dem seltenen Fall, dass die neueste Laufzeitversion Ihre Anwendung negativ beeinflusst, können Sie mit diesem Ansatz, umgehend Korrekturmaßnahmen ergreifen. Funktionen, die sich nicht in der aktiven Entwicklung befinden, erhalten auch in der zweiten Phase den operativen Nutzen von automatischen Laufzeitaktualisierungen.

Dieser Ansatz wirkt sich nicht auf Funktionen aus, die auf Function update (Funktionsaktualisierung) oder Manual (Manuell) festgelegt sind. Funktionen, die den Modus Function update (Funktionsaktualisierung) verwenden, erhalten die neuesten Laufzeitaktualisierungen nur, wenn Sie sie erstellen oder aktualisieren. Funktionen, die den Modus Manual (Manuell) verwenden, erhalten keine Laufzeitaktualisierungen.

Lambda veröffentlicht neue Laufzeitversionen schrittweise und fortlaufend in allen AWS-Regionen. Wenn Ihre Funktionen auf die Modi Auto (Automatisch) oder Function update (Funktionsaktualisierung) festgelegt sind, ist es möglich, dass Funktionen, die zur gleichen Zeit in verschiedenen Regionen oder zu verschiedenen Zeiten in derselben Region eingesetzt werden, unterschiedliche Laufzeitversionen abrufen. Kunden, die eine garantierte Konsistenz der Laufzeitversionen in ihren Umgebungen benötigen, sollten Container-Images verwenden, um ihre Lambda-Funktionen bereitzustellen. Der Modus Manuell ist als vorübergehende Gegenmaßnahme

gedacht, um im seltenen Fall eines Laufzeitzeitproblems ein Laufzeit-Rollback zu ermöglichen, dass eine Laufzeitversion nicht mit Ihrer Funktion kompatibel ist.

Konfigurieren der Einstellungen für die Lambda-Laufzeitverwaltung

Sie können die Laufzeitverwaltungseinstellungen mithilfe der Lambda-Konsole oder der AWS Command Line Interface (AWS CLI) konfigurieren.

> **Note**
>
> Sie können die Laufzeitverwaltungseinstellungen für jede Funktionsversion separat konfigurieren.

So konfigurieren Sie, wie Lambda Ihre Laufzeitversion aktualisiert (Konsole)

1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.
2. Wählen Sie den Namen einer Funktion aus.
3. Wählen Sie auf der Registerkarte Code unter Runtime settings (Laufzeiteinstellungen) die Option Edit runtime management configuration (Laufzeitverwaltungskonfiguration bearbeiten) aus.
4. Wählen Sie unter Runtime management configuration (Laufzeitverwaltungskonfiguration) eine der folgenden Optionen aus:

   - Um Ihre Funktion automatisch auf die neueste Laufzeitversion aktualisieren zu lassen, wählen Sie Auto (Automatisch).
   - Um Ihre Funktion auf die neueste Laufzeitversion zu aktualisieren, wenn Sie die Funktion ändern, wählen Sie Function update (Funktionsaktualisierung).
   - Damit Ihre Funktion nur dann auf die neueste Laufzeitversion aktualisiert wird, wenn Sie den Laufzeitversions-ARN ändern, wählen Sie Manual (Manuell). Sie finden den Laufzeitversions-ARN unter Runtime management configuration (Laufzeitverwaltungskonfiguration). Sie finden den ARN auch in der `INIT_START`-Zeile Ihrer Funktionsprotokolle.

   Weitere Informationen zu diesen Optionen finden Sie unter Laufzeitaktualisierungsmodi.
5. Wählen Sie Speichern.

So konfigurieren Sie, wie Lambda Ihre Laufzeitversion aktualisiert (AWS CLI)

Um das Laufzeitmanagement für eine Funktion zu konfigurieren, führen Sie den AWS CLI-Befehl put-runtime-management-config aus. Wenn Sie den `Manual`-Modus verwenden, müssen Sie auch den ARN der Laufzeitversion angeben.

```
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
```

Die Ausgabe sollte folgendermaßen oder ähnlich aussehen:

```
{
  "UpdateRuntimeOn": "Manual",
  "FunctionArn": "arn:aws:lambda:us-east-2:111122223333:function:my-function",
  "RuntimeVersionArn": "arn:aws:lambda:us-
east-2::runtime:8eeff65f6809a3ce81507fe733fe09b835899b99481ba22fd75b5a7338290ec1"
}
```

Rollback einer Lambda-Laufzeitversion

In dem seltenen Ereignis, dass eine neue Laufzeitversion nicht mit Ihrer vorhandenen Funktion kompatibel ist, können Sie deren Laufzeitversion auf eine frühere Version zurücksetzen. Dadurch bleibt Ihre Anwendung funktionsfähig und die Unterbrechung wird minimiert. Gleichzeitig wird Zeit bereitgestellt, um die Inkompatibilität zu beheben, bevor Sie zur neuesten Laufzeitversion zurückkehren.

Lambda legt keine zeitliche Begrenzung fest, wie lange Sie eine bestimmte Laufzeitversion verwenden können. Wir empfehlen jedoch dringend, so schnell wie möglich auf die neueste Laufzeitversion zu aktualisieren, um von den neuesten Sicherheits-Patches, Leistungsverbesserungen und Funktionen zu profitieren. Lambda bietet die Option, auf eine frühere Laufzeitversion zurückzusetzen, nur als vorübergehende Abhilfe für den seltenen Fall eines Kompatibilitätsproblems bei Laufzeitaktualisierungen. Bei Funktionen, die über einen längeren Zeitraum eine frühere Laufzeitversion verwenden, kann es zu Leistungseinbußen oder Problemen kommen, wie z. B. dem Ablauf eines Zertifikats, was dazu führen kann, dass sie nicht mehr richtig funktionieren.

Sie können eine Laufzeitversion auf die folgenden Arten zurücksetzen:

- Verwenden des Manual (Manuellen) Laufzeitaktualisierungsmodus

- Verwenden veröffentlichter Funktionsversionen

Weitere Informationen finden Sie unter Einführung von AWS Lambda-Laufzeitverwaltungs-Kontrollen im AWS-Computing-Blog.

Rollback einer Laufzeitversion mit dem Manual (Manuellen) Laufzeitaktualisierungsmodus

Wenn Sie den Auto (Automatischen) Aktualisierungsmodus für die Laufzeitversion oder die `$LATEST`-Laufzeitversion verwenden, können Sie Ihre Laufzeitversion im Manual (Manuellen) Modus zurücksetzen. Ändern Sie für die Funktionsversion, die Sie zurücksetzen möchten, den Aktualisierungsmodus der Laufzeitversion zu Manual (Manuell) und geben Sie den ARN der vorherigen Laufzeitversion an. Weitere Informationen zum Ermitteln des ARN der vorherigen Laufzeitversion finden Sie unter Identifizieren von Änderungen der Lambda-Laufzeitversion.

> **Note**
>
> Wenn die `$LATEST`-Version Ihrer Funktion für den Modus Manual (Manuell) konfiguriert ist, können Sie die von Ihrer Funktion verwendete CPU-Architektur oder Laufzeitversion nicht ändern. Um diese Änderungen vorzunehmen, müssen Sie in den Modus Auto (Automatisch) oder Function update (Funktionsaktualisierung) wechseln.

Rollback einer Laufzeitversion mit veröffentlichten Funktionsversionen

Veröffentlichte Funktionsversionen sind eine unveränderliche Momentaufnahme des `$LATEST`-Funktionscodes und der Konfiguration zum Zeitpunkt ihrer Erstellung. Im Modus Auto (Automatisch) aktualisiert Lambda während der zweiten Phase des Rollouts der Laufzeitversion automatisch die Laufzeitversion der veröffentlichten Funktionsversionen. Im Modus Function update (Funktionsaktualisierung) aktualisiert Lambda die Laufzeitversion veröffentlichter Funktionsversionen nicht.

Veröffentlichte Funktionsversionen, die den Modus Function update (Funktionsaktualisierung) verwenden, erstellen daher einen statischen Snapshot des Funktionscodes, der Konfiguration und der Laufzeitversion. Wenn Sie den Modus Function update (Funktionsaktualisierung) mit Funktionsversionen verwenden, können Sie Laufzeitaktualisierungen mit Ihren Bereitstellungen synchronisieren. Sie können auch das Rollback von Code-, Konfigurations- und Laufzeitversionen

koordinieren, indem Sie den Datenverkehr auf eine zuvor veröffentlichte Funktionsversion umleiten. Sie können diesen Ansatz in Ihre kontinuierliche Integration und kontinuierliche Bereitstellung (CI/CD) integrieren, um im seltenen Fall einer Inkompatibilität von Laufzeitaktualisierungen ein vollautomatisches Rollback durchzuführen. Wenn Sie diesen Ansatz verwenden, müssen Sie Ihre Funktion regelmäßig aktualisieren und neue Funktionsversionen veröffentlichen, um die neuesten Laufzeitaktualisierungen zu erhalten. Weitere Informationen finden Sie unter Verstehen des Modells der geteilten Verantwortung für die Verwaltung der Lambda-Laufzeit.

Identifizieren von Änderungen der Lambda-Laufzeitversion

Die Laufzeit-Versionsnummer und der ARN werden in der `INIT_START`-Protokollzeile protokolliert, die Lambda jedes Mal, wenn es eine neue Ausführungsumgebung erstellt, an CloudWatch Logs ausgibt. Da die Ausführungsumgebung für alle Funktionsaufrufe dieselbeLaufzeit verwendet, gibt Lambda die `INIT_START`-Protokollzeile nur aus, wenn Lambda die Init-Phase ausführt. Lambda gibt diese Protokollzeile nicht für jeden Funktionsaufruf aus. Lambda gibt die Protokollzeile an CloudWatch Logs aus, diese ist jedoch in der Konsole nicht sichtbar.

> **Note**
>
> Laufzeit-Versionsnummern sind nicht immer fortlaufend. Auf Version 42 könnte beispielsweise Version 45 folgen.

Example Beispiel für die INIT_START-Protokollzeile

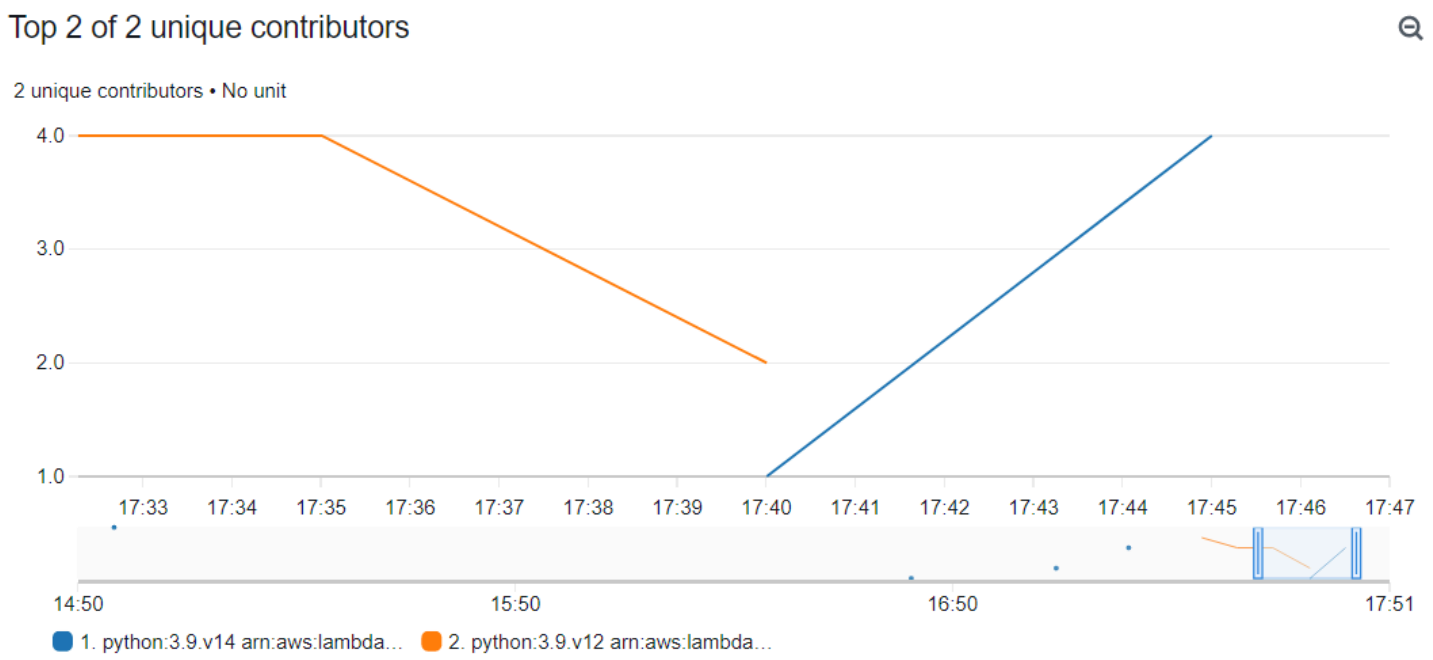
```
INIT_START Runtime Version: python:3.13.v14    Runtime Version ARN: arn:aws:lambda:eu-
south-1::runtime:7b620fc2e66107a1046b140b9d320295811af3ad5d4c6a011fad1fa65127e9e6I
```

Anstatt direkt mit den Protokollen zu arbeiten, können Sie Amazon CloudWatch Contributor Insights verwenden, um Übergänge zwischen Laufzeitversionen zu identifizieren. Die folgende Regel zählt die verschiedenen Laufzeitversionen aus jeder `INIT_START`-Protokollzeile. Um die Regel zu verwenden, ersetzen Sie den Beispiel-Protokollgruppennamen `/aws/lambda/*` durch das entsprechende Präfix für Ihre Funktion oder Funktionsgruppe.

```
{
  "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"
  }
}
```

Der folgende Bericht zu CloudWatch Contributor Insights zeigt ein Beispiel für einen Laufzeit-Versionswechsel, wie er in der vorherigen Regel erfasst wurde. Die orange Linie zeigt die Initialisierung der Ausführungsumgebung für die frühere Laufzeitversion (python:3.13.v12), während die blaue Linie die Initialisierung der Ausführungsumgebung für die neue Laufzeitversion (python:3.13.v14) anzeigt.



Verstehen des Modells der geteilten Verantwortung für die Verwaltung der Lambda-Laufzeit

Lambda ist für das Kuratieren und Veröffentlichen von Sicherheitsupdates für alle unterstützten verwalteten Laufzeiten und Container-Images verantwortlich. Die Verantwortung für die Aktualisierung vorhandener Funktionen zur Verwendung der neuesten Laufzeitversion hängt davon ab, welchen Laufzeitaktualisierungsmodus Sie verwenden.

Lambda ist dafür verantwortlich, Laufzeitaktualisierungen auf alle Funktionen anzuwenden, die für den Laufzeitaktualisierungsmodus Auto (Automatisch) konfiguriert sind.

Für Funktionen, die mit dem Laufzeitaktualisierungsmodus Function update (Funktionsaktualisierung) konfiguriert sind, sind Sie für die regelmäßige Aktualisierung Ihrer Funktion verantwortlich. Lambda ist dafür verantwortlich, Laufzeitaktualisierungen anzuwenden, wenn Sie diese Aktualisierungen vornehmen. Wenn Sie Ihre Funktion nicht aktualisieren, aktualisiert Lambda die Laufzeit nicht. Wenn Sie Ihre Funktion nicht regelmäßig aktualisieren, empfehlen wir dringend, diese für automatische Laufzeitaktualisierungen zu konfigurieren, damit sie weiterhin Sicherheitsupdates erhält.

Bei Funktionen, die für den Laufzeitaktualisierungsmodus Manual (Manuell) konfiguriert sind, sind Sie dafür verantwortlich, Ihre Funktion zu aktualisieren, damit sie die neueste Laufzeitversion verwendet. Wir empfehlen dringend, diesen Modus nur zu verwenden, um im seltenen Ereignis einer Inkompatibilität der Laufzeitaktualisierung die Laufzeitversion vorübergehend zurückzusetzen. Wir

empfehlen Ihnen außerdem, so schnell wie möglich in den Modus Auto (Automatisch) zu wechseln, um die Zeit zu minimieren, in der Ihre Funktionen nicht gepatcht werden.

Wenn Sie Container-Images zum Bereitstellen Ihrer Funktionen verwenden, ist Lambda für das Veröffentlichen aktualisierter Basis-Images verantwortlich. In diesem Fall sind Sie dafür verantwortlich, das Container-Image Ihrer Funktion aus dem neuesten Basis-Image neu zu erstellen und das Container-Image erneut bereitzustellen.

Dies ist in der folgenden Tabelle zusammengefasst:

Bereitstellungsmodus	Verantwortung von Lambda	Verantwortung des Kunden
Verwaltete Laufzeit, Modus Auto (Automatisch)	Veröffentlichen Sie neue Laufzeitversionen mit den neuesten Patches. Wenden Sie Laufzeit-Patches auf vorhandene Funktionen an.	Führen Sie im seltenen Fall eines Kompatibilitätsproblems bei Laufzeitaktualisierungen ein Rollback auf eine frühere Laufzeitversion durch. Folgen Sie den bewährten Methoden für Abwärtskompatibilität.
Verwaltete Laufzeit, Modus Function update (Funktionsaktualisierung)	Veröffentlichen Sie neue Laufzeitversionen mit den neuesten Patches.	Aktualisieren Sie die Funktionen regelmäßig, um die neueste Laufzeitversion zu erhalten. Schalten Sie eine Funktion in den Modus Auto (Automatisch), wenn Sie die Funktion nicht regelmäßig aktualisieren. Führen Sie im seltenen Fall eines Kompatibilitätsproblems bei Laufzeitaktualisierungen ein Rollback auf eine frühere Laufzeitversion durch. Folgen Sie den bewährten Methoden für Abwärtskompatibilität.
Verwaltete Laufzeit, Modus Manual (Manuell)	Veröffentlichen Sie neue Laufzeitversionen mit den neuesten Patches.	Verwenden Sie diesen Modus nur für ein vorübergehendes Laufzeit-Rollback im seltenen Fall eines Kompatibilitätsproblems bei der Aktualisierung.

Bereitstellungsmodus	Verantwortung von Lambda	Verantwortung des Kunden
		Schalten Sie die Funktionen so schnell wie möglich in den Modus Auto (Automatisch) oder Function update (Funktionsaktualisierung) und auf die neueste Laufzeitversion.
Container-Image	Veröffentlichen Sie neue Container-Images mit den neuesten Patches.	Stellen Sie die Funktionen regelmäßig neu bereit, indem Sie das neueste Container-Basis-Image verwenden, um die neuesten Patches abzurufen.

Weitere Informationen zur geteilten Verantwortung mit AWS finden Sie unter Modell der geteilten Verantwortung.

Kontrolle der Lambda-Laufzeitaktualisierungsberechtigungen für Anwendungen mit hoher Konformität

Um Patching-Anforderungen zu erfüllen, verlassen sich Lambda-Kunden in der Regel auf automatische Laufzeitaktualisierungen. Wenn Ihre Anwendung strengen Anforderungen an die Aktualität von Patches unterliegt, sollten Sie die Verwendung früherer Laufzeitversionen einschränken. Sie können die Laufzeitverwaltungskontrollen von Lambda einschränken, indem Sie AWS Identity and Access Management (IAM) verwenden, um Benutzern in Ihrem AWS-Konto den Zugriff auf die PutRuntimeManagementConfig-API-Operation zu verweigern. Dieser Vorgang wird verwendet, um den Laufzeitaktualisierungsmodus für eine Funktion auszuwählen. Wenn Sie den Zugriff auf diesen Vorgang verweigern, werden alle Funktionen standardmäßig auf den Modus Auto (Automatisch) umgestellt. Sie können diese Einschränkung unternehmensweit anwenden, indem Sie Service-Kontrollrichtlinien (SCP) verwenden. Wenn Sie eine Funktion auf eine frühere Laufzeitversion zurücksetzen müssen, können Sie von Fall zu Fall eine Richtlinienausnahme gewähren.

Rufen Sie Daten über Lambda-Funktionen ab, die eine veraltete Laufzeit verwenden

Wenn sich eine Lambda-Laufzeit dem Ende nähert, werden Sie von Lambda per E-Mail benachrichtigt und im Laufe der Zeit benachrichtigt. Health Dashboard Trusted Advisor In diesen E-Mails und Benachrichtigungen werden die $LATEST-Versionen von Funktionen aufgelistet, die die Laufzeit verwenden. Um all Ihre Funktionsversionen aufzulisten, die eine bestimmte Laufzeit verwenden, können Sie die AWS Command Line Interface (AWS CLI) oder eine der folgenden verwenden. AWS SDKs

Wenn Sie über eine große Anzahl von Funktionen verfügen, die eine Laufzeit verwenden, die bald nicht mehr unterstützt wird, können Sie das AWS CLI oder auch verwenden, AWS SDKs um Aktualisierungen Ihrer am häufigsten aufgerufenen Funktionen zu priorisieren.

In den folgenden Abschnitten erfahren Sie, wie Sie Funktionen verwenden AWS CLI und Daten über Funktionen sammeln AWS SDKs , die eine bestimmte Laufzeit verwenden.

Auflisten von Funktionsversionen, die eine bestimmte Laufzeit verwenden

Führen Sie den folgenden Befehl aus, AWS CLI um alle Ihre Funktionsversionen aufzulisten, die eine bestimmte Laufzeit verwenden. `RUNTIME_IDENTIFIER`Ersetzen Sie ihn durch den Namen der Laufzeit, die veraltet ist, und wählen Sie Ihren eigenen aus. AWS-Region Um nur die $LATEST-Funktionsversionen aufzulisten, lassen Sie `--function-version ALL` im Befehl weg.

```
aws lambda list-functions --function-version ALL --region us-east-1 --output text --
query "Functions[?Runtime=='RUNTIME_IDENTIFIER'].FunctionArn"
```

> **Tip**
>
> Der Beispielbefehl listet Funktionen in der `us-east-1` Region für eine bestimmte Region auf. AWS-Konto Sie müssen diesen Befehl für jede Region wiederholen, in der Ihr Konto Funktionen hat, und für jede Ihrer Regionen. AWS-Konten

Sie können auch Funktionen auflisten, die eine bestimmte Laufzeit verwenden, indem Sie eine der folgenden Optionen verwenden AWS SDKs. Der folgende Beispielcode verwendet V3 AWS SDK für JavaScript und die AWS SDK für Python (Boto3) , um eine Liste der Funktionen ARNs für Funktionen

zurückzugeben, die eine bestimmte Laufzeit verwenden. Der Beispielcode gibt auch die CloudWatch Protokollgruppe für jede der aufgelisteten Funktionen zurück. Sie können diese Protokollgruppe verwenden, um das Datum des letzten Aufrufs der Funktion zu ermitteln. Weitere Informationen finden Sie im folgenden Abschnitt the section called “Identifizieren der am häufigsten und zuletzt aufgerufenen Funktionen”.

Node.js

Example JavaScript Code zum Auflisten von Funktionen, die eine bestimmte Laufzeit verwenden

```
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 Python-Code zur Auflistung von Funktionen mit einer bestimmten Laufzeit

```
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')
```

Weitere Informationen zur Verwendung eines AWS SDK zum Auflisten Ihrer Funktionen mithilfe der ListFunctionsAktion finden Sie in der SDK-Dokumentation für Ihre bevorzugte Programmiersprache.

Sie können auch die Funktion AWS Config Erweiterte Abfragen verwenden, um all Ihre Funktionen aufzulisten, die eine betroffene Laufzeit verwenden. Diese Abfrage gibt nur die Versionen der Funktion $LATEST zurück, aber Sie können Abfragen aggregieren, um Funktionen für alle Regionen und mehrere Regionen AWS-Konten mit einem einzigen Befehl aufzulisten. Weitere Informationen finden Sie unter Abfragen des aktuellen Konfigurationsstatus von AWS Auto Scaling Ressourcen im AWS Config Entwicklerhandbuch.

Identifizieren der am häufigsten und zuletzt aufgerufenen Funktionen

Wenn Ihre Version Funktionen AWS-Konto enthält, die eine Laufzeit verwenden, die bald veraltet sein wird, sollten Sie der Aktualisierung von Funktionen, die häufig aufgerufen werden, oder Funktionen, die kürzlich aufgerufen wurden, Priorität einräumen.

Wenn Sie nur über wenige Funktionen verfügen, können Sie die CloudWatch Logs-Konsole verwenden, um diese Informationen zu sammeln, indem Sie sich die Log-Streams Ihrer Funktionen ansehen. Weitere Informationen finden Sie unter An Logs gesendete CloudWatch Protokolldaten anzeigen.

Um die Anzahl der letzten Funktionsaufrufen zu sehen, können Sie auch die in der Lambda-Konsole angezeigten CloudWatch Metrikinformationen verwenden. Gehen Sie wie folgt vor, um diese Informationen anzuzeigen:

1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.
2. Wählen Sie die Funktion aus, für die Sie eine Aufrufstatistik anzeigen möchten.
3. Wählen Sie den Tab Überwachung.
4. Legen Sie den Zeitraum fest, für den Sie die Statistiken anzeigen möchten, indem Sie den Datumsbereich auswählen. Die letzten Aufrufe werden im Bereich Aufrufe angezeigt.

Bei Konten mit einer größeren Anzahl von Funktionen kann es effizienter sein, diese Daten programmgesteuert mithilfe der AWS CLI oder einer der API-Aktionen „ AWS SDKs using the“ und zu sammeln. DescribeLogStreamsGetMetricStatistics

Die folgenden Beispiele enthalten Codefragmente, die V3 AWS SDK für JavaScript und die verwenden, AWS SDK für Python (Boto3) um das Datum des letzten Aufrufs für eine bestimmte Funktion zu identifizieren und die Anzahl der Aufrufe für eine bestimmte Funktion in den letzten 14 Tagen zu bestimmen.

Node.js

Example JavaScript Code, um die Uhrzeit des letzten Aufrufs für eine Funktion zu ermitteln

```
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 Python-Code, um den Zeitpunkt des letzten Aufrufs für eine Funktion zu ermitteln

```
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**
>
> Sie können den Namen der Protokollgruppe Ihrer Funktion mithilfe der ListFunctionsAPI-Operation ermitteln. Der Code in the section called “Auflisten von Funktionsversionen, die eine bestimmte Laufzeit verwenden” enthält ein Beispiel für diese Vorgehensweise.

Node.js

Example JavaScript Code, um die Anzahl der Aufrufe in den letzten 14 Tagen zu ermitteln

```
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 Python-Code, um die Anzahl der Aufrufe in den letzten 14 Tagen zu ermitteln

```
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}')
```

Ändern der Laufzeitumgebung

Sie können interne Erweiterungen verwenden, um den Laufzeitprozess zu ändern. Interne Erweiterungen sind keine separaten Prozesse – sie werden als Teil des Laufzeitprozesses ausgeführt.

Lambda bietet sprachspezifische Umgebungsvariablen, die Sie festlegen können, um Optionen und Werkzeuge zur Laufzeit hinzuzufügen. Lambda bietet auch Wrapper-Skripts, die Lambda erlauben, den Laufzeit-Startup an Ihr Skript zu delegieren. Sie können ein Wrapper-Skript erstellen, um das Laufzeit-Startup-Verhalten anzupassen.

Sprachspezifische Umgebungsvariablen

Lambda unterstützt Konfigurationsmöglichkeiten, mit denen Code während der Funktionsinitialisierung über die folgenden sprachspezifischen Umgebungsvariablen vorgeladen werden kann:

- `JAVA_TOOL_OPTIONS` – Unter Java unterstützt Lambda diese Umgebungsvariable, um zusätzliche Befehlszeilenvariablen in Lambda zu setzen. Mit dieser Umgebungsvariablen können Sie die Initialisierung von Werkzeugen angeben, insbesondere das Starten von nativen oder Java-Agenten mit den Optionen `agentlib` oder `javaagent`. Weitere Informationen finden Sie unter `JAVA_TOOL_OPTIONS`-Umgebungsvariablen.
- `NODE_OPTIONS` – Verfügbar in Node.js-Laufzeiten.
- `DOTNET_STARTUP_HOOKS` – Auf .NET Core 3.1 und höher gibt diese Umgebungsvariable einen Pfad zu einer Assembly (DLL) an, die Lambda verwenden kann.

Die Verwendung sprachspezifischer Umgebungsvariablen ist die bevorzugte Methode zum Festlegen von Startup-Eigenschaften.

Wrapper-Skripte

Sie können ein Wrapper-Skript erstellen, um das Laufzeit-Startup-Verhalten Ihrer Lambda-Funktion anzupassen. Mit einem Wrapper-Skript können Sie Konfigurationsparameter festlegen, die nicht über sprachspezifische Umgebungsvariablen festgelegt werden können.

> **Note**
>
> Aufrufe können fehlschlagen, wenn das Wrapper-Skript den Laufzeitprozess nicht erfolgreich startet.

Wrapper-Skripts werden auf allen nativen Lambda-Laufzeiten unterstützt. Wrapper-Skripts werden auf Reine OS-Laufzeiten (der `provided`-Laufzeitfamilie) nicht unterstützt.

Wenn Sie ein Wrapper-Skript für Ihre Funktion verwenden, startet Lambda die Laufzeit mit Ihrem Skript. Lambda sendet den Pfad an Ihr Skript zum Interpreter und alle ursprünglichen Argumente für den Startup der Standardlaufzeit. Ihr Skript kann das Startup-Verhalten des Programms erweitern oder transformieren. Beispielsweise kann das Skript Argumente injizieren und ändern, Umgebungsvariablen festlegen oder Metriken, Fehler und andere Diagnoseinformationen erfassen.

Sie geben das Skript an, indem Sie den Wert der `AWS_LAMBDA_EXEC_WRAPPER`-Umgebungsvariablen als Dateisystempfad einer ausführbaren Binärdatei oder eines Skripts festlegen.

Beispiel: Erstellen und Verwenden eines Wrapper-Skripts als Lambda-Ebene

Im folgenden Beispiel erstellen Sie ein Wrapper-Skript, um den Python-Interpreter mit der `-X importtime`-Option zu starten. Wenn Sie die Funktion ausführen, generiert Lambda einen Protokolleintrag, der die Dauer der Importzeit für jeden Import anzeigt.

So erstellen und verwenden Sie ein Wrapper-Skript als Ebene

1. Erstellen Sie ein Verzeichnis für die Ebene:

   ```
   mkdir -p python-wrapper-layer/bin
   cd python-wrapper-layer/bin
   ```

2. Fügen Sie im Verzeichnis `bin` den folgenden Code in eine neue Datei namens `importtime_wrapper` ein. Dies ist das Wrapper-Skript.

   ```
   #!/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. Geben Sie dem Skript die Berechtigung zur Ausführung:

```
chmod +x importtime_wrapper
```

4. Erstellen Sie eine ZIP-Datei für die Ebene:

```
cd ..
zip -r ../python-wrapper-layer.zip .
```

5. Vergewissern Sie sich, dass Ihre .zip-Datei die folgende Verzeichnisstruktur aufweist:

```
python-wrapper-layer.zip
# bin
    # importtime_wrapper
```

6. Erstellen Sie eine Ebene mit dem .zip-Paket.
7. Erstellen Sie eine Funktion mit der Lambda-Konsole.
   a. Öffnen Sie die Lambda-Konsole.
   b. Wählen Sie Create function (Funktion erstellen).
   c. Geben Sie einen Function name (Funktionsnamen) ein.
   d. Wählen Sie für Laufzeit die neueste unterstützte Python-Laufzeit aus.
   e. Wählen Sie Create function (Funktion erstellen).
8. Fügen Sie die Ebene zu Ihrer Funktion hinzu.
   a. Wählen Sie Ihre Funktion aus, und wählen Sie dann die Registerkarte Code, falls sie nicht bereits ausgewählt ist.
   b. Blättern Sie nach unten bis zum Abschnitt Ebenen und wählen Sie dann Eine Ebene hinzufügen.
   c. Wählen Sie unter Ebenenquelle die Option Benutzerdefinierte Ebenen und wählen Sie dann Ihre Ebene aus der Dropdown-Liste Benutzerdefinierte Ebenen aus.

d. Wählen Sie für Version die Option 1.

e. Wählen Sie Hinzufügen aus.

9. Fügen Sie die Wrapper-Umgebungsvariable hinzu.

   a. Wählen Sie den Tab Configuration (Konfiguration) und dann Environment variables (Umgebungsvariablen) aus.

   b. Wählen Sie unter Environment variables (Umgebungsvariablen) die Option Edit (Bearbeiten).

   c. Wählen Sie Umgebungsvariablen hinzufügen aus.

   d. Geben Sie für Key (Schlüssel) `AWS_LAMBDA_EXEC_WRAPPER` ein.

   e. Geben Sie als Wert `/opt/bin/importtime_wrapper` (`/opt/`+ die Ordnerstruktur Ihrer ZIP-Ebene) ein.

   f. Wählen Sie Speichern.

10. Testen Sie das Wrapper-Skript.

    a. Wählen Sie die Registerkarte Test.

    b. Wählen Sie unter Testereignis die Option Test aus. Sie brauchen kein Test-Ereignis zu erstellen – das Standard-Ereignis ist ausreichend.

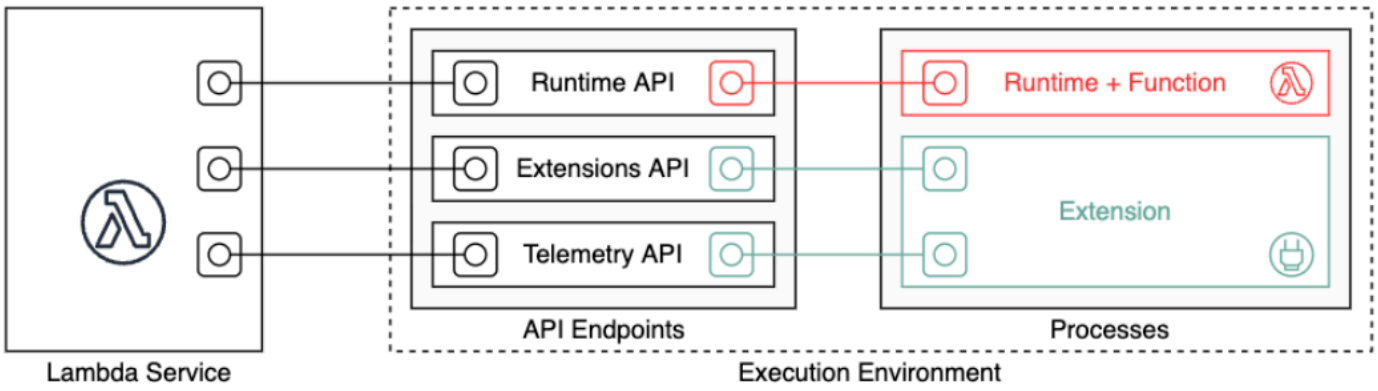
    c. Scrollen Sie nach unten zu Protokollausgabe. Da Ihr Wrapper-Skript den Python-Interpreter mit der `-X importtime`-Option gestartet hat, zeigen die Protokolle die für jeden Import erforderliche Zeit an. Zum Beispiel:

```
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
```

Verwenden der Lambda-Laufzeit-API für benutzerdefinierte Laufzeiten

AWS Lambda bietet eine HTTP-API für benutzerdefinierte Laufzeiten, um Aufrufereignisse von Lambda zu empfangen und Antwortdaten innerhalb der Lambda-Ausführungsumgebung zurückzusenden. Dieser Abschnitt enthält die API-Referenz für die Lambda-Laufzeiten-API.

> **Lambda Managed Instances unterstützen gleichzeitige Anfragen**
>
> Lambda Managed Instances verwenden dieselbe Runtime-API wie Lambda-Funktionen (Standard). Der Hauptunterschied besteht darin, dass verwaltete Instances gleichzeitige `/response` Anfragen `/next` und Anfragen bis zum konfigurierten Limit akzeptieren können. `AWS_LAMBDA_MAX_CONCURRENCY` Dadurch können mehrere Aufrufe gleichzeitig in einer einzigen Ausführungsumgebung verarbeitet werden. Weitere Informationen zu verwalteten Instanzen finden Sie unter. Grundlegendes zur Ausführungsumgebung von Lambda Managed Instances



Die OpenAPI-Spezifikation für die Laufzeit-API-Version 2018-06-01 ist unter runtime-api.zip verfügbar.

Um eine API-Anforderungs-URL zu erstellen, rufen Laufzeitumgebungen den API-Endpunkt aus der `AWS_LAMBDA_RUNTIME_API`-Umgebungsvariablen ab, fügen die API-Version und dann den gewünschten Ressourcenpfad hinzu.

Example Anforderung

```
curl "http://${AWS_LAMBDA_RUNTIME_API}/2018-06-01/runtime/invocation/next"
```

API-Methoden

- Nächster Aufruf
- Aufrufantwort
- Initialisierungsfehler
- Aufruffehler

Nächster Aufruf

Pfad – `/runtime/invocation/next`

Methode – GET

Die Laufzeit sendet diese Meldung an Lambda, um ein Aufrufereignis anzufordern. Der Antworttext enthält die Nutzlast aus dem Aufruf. Dabei handelt es sich um ein JSON-Dokument, das Ereignisdaten aus dem Funktionsauslöser enthält. Die Antwort-Header enthalten zusätzliche Daten zum Aufruf.

Antwort-Header

- `Lambda-Runtime-Aws-Request-Id` – Die Anforderungs-ID, mit der die Anforderung identifiziert wird, die den Aufruf der Funktion ausgelöst hat.

  Beispiel, `8476a536-e9f4-11e8-9739-2dfe598c3fcd`.
- `Lambda-Runtime-Deadline-Ms` – Das Datum, an dem eine Zeitüberschreitung für die Funktion eintritt (in Unix-Millisekunden).

  Beispiel, `1542409706888`.
- `Lambda-Runtime-Invoked-Function-Arn` – Der ARN der/des Lambda-Funktion, -Version oder -Alias, die/der im Aufruf angegeben ist.

  Beispiel, `arn:aws:lambda:us-east-2:123456789012:function:custom-runtime`.
- `Lambda-Runtime-Trace-Id` – Der AWS X-Ray -Nachverfolgungs-Header.

Beispiel, `Root=1-5bef4de7-ad49b0e87f6ef6c87fc2e700;Parent=9a9197af755a6419;Sampled=1`.

- `Lambda-Runtime-Client-Context`— Für Aufrufe aus dem AWS Mobile SDK: Daten über die Client-Anwendung und das Gerät.
- `Lambda-Runtime-Cognito-Identity`— Für Aufrufe aus dem AWS Mobile SDK Daten über den Amazon Cognito Cognito-Identitätsanbieter.

Legen Sie kein Timeout für die `GET`-Anfrage fest, da sich die Antwort verzögern könnte. Zwischen dem Bootstrap der Laufzeit durch Lambda und der Rückgabe eines Ereignisses durch die Laufzeit ist der Laufzeitprozess möglicherweise für mehrere Sekunden eingefroren.

Die Anforderungs-ID verfolgt den Aufruf innerhalb von Lambda nach. Sie verwenden sie, um den Aufruf anzugeben, wenn Sie die Antwort senden.

Der Nachverfolgungs-Header enthält die Nachverfolgungs-ID, die ID des übergeordneten Segments und die Erfassungsentscheidung Wenn die Anforderung erfasst wurde, wurde die Anforderung von Lambda oder einem Upstream-Service erfasst. Die Laufzeit sollte die `_X_AMZN_TRACE_ID` auf den Wert des Headers festlegen. Das X-Ray-SDK liest dies, um die Anfrage abzurufen IDs und festzustellen, ob sie verfolgt werden soll.

Aufrufantwort

Pfad – `/runtime/invocation/`*`AwsRequestId`*`/response`

Methode – POST

Nachdem die Funktion bis zum Abschluss ausgeführt wurde, schickt die Laufzeitumgebung eine Aufrufantwort an Lambda. Im Fall synchroner Aufrufe schickt Lambda die Antwort anschließend an den Client zurück.

Example Erfolg für Anforderung

```
REQUEST_ID=156cb537-e2d4-11e8-9b34-d36013741fb9
curl "http://${AWS_LAMBDA_RUNTIME_API}/2018-06-01/runtime/invocation/$REQUEST_ID/
response"  -d "SUCCESS"
```

Initialisierungsfehler

Wenn die Funktion einen Fehler zurückgibt oder die Laufzeit während der Initialisierung auf einen Fehler stößt, verwendet die Laufzeit diese Methode, um den Fehler an Lambda zu melden.

Pfad – `/runtime/init/error`

Methode – POST

Header

`Lambda-Runtime-Function-Error-Type` – Der Fehlertyp, auf den die Laufzeit gestoßen ist. Erforderlich: Nein.

Dieser Header besteht aus einem Zeichenfolgen-Wert. Lambda akzeptiert jede Zeichenfolge, aber wir empfehlen ein Format von <category.reason>. Beispiel:

- Laufzeit. NoSuchHandler
- Laufzeit. APIKeyNotFound
- Laufzeit. ConfigInvalid
- Laufzeit. UnknownReason

Body-Parameter

`ErrorRequest` – Informationen über den Fehler. Erforderlich: Nein

Dieses Feld ist ein JSON-Objekt mit der folgenden Struktur:

```
{
      errorMessage: string (text description of the error),
      errorType: string,
      stackTrace: array of strings
}
```

Beachten Sie, dass Lambda jeden Wert für `errorType` akzeptiert.

Das folgende Beispiel zeigt eine Lambda-Funktionsfehlermeldung, in der die Funktion die im Aufruf bereitgestellten Ereignisdaten nicht analysieren konnte.

Example Funktionsfehler

```
{
      "errorMessage" : "Error parsing event data.",
      "errorType" : "InvalidEventDataException",
      "stackTrace": [ ]
}
```

Antworttextparameter:

- `StatusResponse` – Zeichenfolge. Statusinformationen, gesendet mit 202 Antwortcodes.
- `ErrorResponse`— Zusätzliche Fehlerinformationen, die zusammen mit den Fehlerantwortcodes gesendet werden. ErrorResponse enthält einen Fehlertyp und eine Fehlermeldung.

Antwortcodes

- 202 – Akzeptiert
- 403 – Verboten
- 500 – Container-Fehler. Nicht wiederherstellbarer Zustand. Die Laufzeit sollte umgehend beendet werden.

Example Initialisierungsfehleranforderung

```
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"
```

Aufruffehler

Wenn die Funktion einen Fehler zurückgibt oder die Laufzeit auf einen Fehler stößt, verwendet die Laufzeitumgebung diese Methode, um den Fehler an Lambda zu melden.

Pfad – `/runtime/invocation/`*`AwsRequestId`*`/error`

Methode – POST

Header

`Lambda-Runtime-Function-Error-Type` – Der Fehlertyp, auf den die Laufzeit gestoßen ist. Erforderlich: Nein.

Dieser Header besteht aus einem Zeichenfolgen-Wert. Lambda akzeptiert jede Zeichenfolge, aber wir empfehlen ein Format von <category.reason>. Beispiel:

- Laufzeit. NoSuchHandler
- Laufzeit. APIKeyNotFound
- Laufzeit. ConfigInvalid
- Laufzeit. UnknownReason

Body-Parameter

`ErrorRequest` – Informationen über den Fehler. Erforderlich: Nein

Dieses Feld ist ein JSON-Objekt mit der folgenden Struktur:

```
{
      errorMessage: string (text description of the error),
      errorType: string,
      stackTrace: array of strings
}
```

Beachten Sie, dass Lambda jeden Wert für `errorType` akzeptiert.

Das folgende Beispiel zeigt eine Lambda-Funktionsfehlermeldung, in der die Funktion die im Aufruf bereitgestellten Ereignisdaten nicht analysieren konnte.

Example Funktionsfehler

```
{
      "errorMessage" : "Error parsing event data.",
      "errorType" : "InvalidEventDataException",
      "stackTrace": [ ]
}
```

Antworttextparameter:

- `StatusResponse` – Zeichenfolge. Statusinformationen, gesendet mit 202 Antwortcodes.

- `ErrorResponse`— Zusätzliche Fehlerinformationen, die zusammen mit den Fehlerantwortcodes gesendet werden. ErrorResponse enthält einen Fehlertyp und eine Fehlermeldung.

Antwortcodes

- 202 – Akzeptiert
- 400 – Ungültige Anfrage
- 403 – Verboten
- 500 – Container-Fehler. Nicht wiederherstellbarer Zustand. Die Laufzeit sollte umgehend beendet werden.

Example Fehler für Anforderung

```
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"
```

Wann sollten die reinen OS-Laufzeiten verwendet werden

Lambda bietet verwaltete Laufzeiten für Java, Python, Node.js, .NET und Ruby. Um Lambda-Funktionen in einer Programmiersprache zu erstellen, die nicht als verwaltete Laufzeit verfügbar ist, verwenden Sie eine reine OS-Laufzeit (die `provided`-Laufzeitfamilie). Es gibt drei Hauptanwendungsfälle für reine OS-Laufzeiten:

- Native Kompilierung ahead-of-time (AOT): Sprachen wie Go, Rust, Swift und C++ werden nativ zu einer ausführbaren Binärdatei kompiliert, für die keine spezielle Sprachlaufzeit erforderlich ist. Diese Sprachen benötigen nur eine Betriebssystemumgebung, in der die kompilierte Binärdatei ausgeführt werden kann. Sie können auch Laufzeiten verwenden, die nur auf Lambda OS basieren, um Binärdateien bereitzustellen, die mit.NET Native AOT und Java GraalVM Native Image kompiliert wurden.

  Sie müssen einen Laufzeitschnitten-Client in Ihre Binärdatei aufnehmen. Der Laufzeitenschnittstellen-Client ruft die Verwenden der Lambda-Laufzeit-API für benutzerdefinierte Laufzeiten auf, um Funktionsaufrufe abzurufen, und ruft dann den Funktionshandler auf. Lambda bietet Runtime-Schnittstellenclients für Rust, Go, .NET Native AOT, Swift (experimentell) und C++ (experimentell).

  Sie müssen Ihre Binärdatei für eine Linux-Umgebung und für dieselbe Befehlssatzarchitektur kompilieren, die Sie für die Funktion verwenden möchten (x86_64 oder arm64).
- Laufzeiten von Drittanbietern: Sie können Lambda-Funktionen mit off-the-shelf Laufzeiten wie Bref für PHP ausführen.
- Benutzerdefinierte Laufzeiten: Sie können Ihre eigene Laufzeit für eine Sprache oder Sprachversion erstellen, für die Lambda keine verwaltete Laufzeit bereitstellt, z. B. Node.js 19. Weitere Informationen finden Sie unter Erstellen einer benutzerdefinierten Laufzeit für AWS Lambda. Dies ist der am wenigsten verbreitete Anwendungsfall für reine OS-Laufzeiten.

Lambda unterstützt die folgenden reinen OS-Laufzeiten.

Name	ID	Betriebssystem	Datum der Veraltung	Blockfunktion erstellen	Blockfunktion aktualisieren
Reine OS-Laufzeit	`provided.al2023`	Amazon Linux 2023	30. Juni 2029	31. Juli 2029	31. August 2029

Name	ID	Betriebssystem	Datum der Veraltung	Blockfunktion erstellen	Blockfunktion aktualisieren
Reine OS-Laufzeit	`provided.al2`	Amazon Linux 2	31. Juli 2026	31. August 2026	30. September 2026

Die Laufzeit von Amazon Linux 2023 (`provided.al2023`) bietet mehrere Vorteile gegenüber Amazon Linux 2, darunter einen geringeren Bereitstellungsaufwand und aktualisierte Versionen von Bibliotheken wie `glibc`.

Die `provided.al2023`-Laufzeit verwendet `dnf` als Paketmanager anstelle von `yum`, was der Standard-Paketmanager in Amazon Linux 2 ist. Weitere Informationen zu den Unterschieden zwischen `provided.al2023` und `provided.al2` finden Sie unter Einführung in die Amazon Linux 2023 Runtime for AWS Lambda im AWS Compute-Blog.

Erstellen einer benutzerdefinierten Laufzeit für AWS Lambda

Sie können eine AWS Lambda Laufzeit in jeder Programmiersprache implementieren. Eine Laufzeit ist ein Programm, das die Handler-Methode einer Lambda-Funktion ausführt, wenn die Funktion aufgerufen wird. Die Laufzeit kann im Bereitstellungspaket Ihrer Funktion enthalten sein oder in einem Layer verteilt werden. Wenn Sie die Lambda-Funktion erstellen, wählen Sie eine reine OS-Laufzeit (die `provided`-Laufzeitfamilie) aus.

> **Note**
>
> Das Erstellen einer benutzerdefinierten Laufzeit ist ein Anwendungsfall für Fortgeschrittene. Informationen zur Kompilierung in eine native Binärdatei oder zur Verwendung einer off-the-shelf Runtime eines Drittanbieters finden Sie unterWann sollten die reinen OS-Laufzeiten verwendet werden.

Eine exemplarische Vorgehensweise für die Bereitstellung einer benutzerdefinierten Laufzeit finden Sie unter Tutorial: Erstellen einer benutzerdefinierten Laufzeit.

Themen

- Voraussetzungen

- Implementieren von Antwort-Streaming in einer benutzerdefinierten Laufzeitumgebung
- Erstellung benutzerdefinierter Laufzeiten für Lambda Managed Instances

Voraussetzungen

Benutzerdefinierte Laufzeiten müssen bestimmte Initialisierungs- und Verarbeitungsaufgaben abschließen. Eine Laufzeit führt den Setup-Code der Funktion aus, liest den Namen des Handlers aus einer Umgebungsvariablen und liest Aufrufereignisse von der Lambda-Laufzeit-API. Die Laufzeit übergibt die Ereignisdaten an den Funktions-Handler und sendet die Antwort aus dem Handler zurück an Lambda.

Initialisierungsaufgaben

Die Initialisierungsaufgaben werden einmal pro Instance der Funktion ausgeführt, um die Umgebung auf die Verarbeitung von Aufrufen vorzubereiten.

- Einstellungen abrufen – Liest Umgebungsvariablen, um Details zu Funktion und Umgebung abzurufen.
  - `_HANDLER` – Der Speicherort für den Handler aus der Konfiguration der Funktion. Das Standardformat ist *`file`*`.`*`method`*, wobei `file` der Name der Datei ohne Erweiterung und `method` der Name einer Methode oder Funktion ist, die in der Datei definiert ist.
  - `LAMBDA_TASK_ROOT` – Das Verzeichnis, das den Funktionscode enthält.
  - `AWS_LAMBDA_RUNTIME_API` – Host und Port der Laufzeit-API.

  Unter Definierte Laufzeitumgebungsvariablen finden Sie eine vollständige Liste der verfügbaren Variablen.
- Funktion initialisieren – Lädt die Handler-Datei und führt den globalen oder statischen Code aus, den sie enthält. Funktionen sollten statische Ressourcen wie SDK-Clients und Datenbankverbindungen einmal erstellen und für mehrere Aufrufe wiederverwenden.
- Fehler verarbeiten – Wenn ein Fehler auftritt, werden die Initialisierungsfehler-API aufgerufen die Ausführung sofort beendet.

Die Initialisierung zählt zur fakturierten Ausführungszeit und zum Timeout. Wenn eine Ausführung die Initialisierung einer neuen Instance Ihrer Funktion auslöst, sehen Sie die Initialisierungszeit in den Protokollen und in der AWS X-Ray -Nachverfolgung.

Example log

```
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
```

Verarbeitungsaufgaben

Während der Ausführung verwendet eine Laufzeit die Lambda-Laufzeitschnittstelle, um eingehende Ereignisse zu verwalten und Fehler zu melden. Nach Abschluss der Initialisierungsaufgaben verarbeitet die Laufzeit eingehende Ereignisse in einer Schleife. Führen Sie in Ihrem Laufzeitcode die folgenden Schritte in der angegebenen Reihenfolge aus.

- Ereignis abrufen – Ruft die API für den nächsten Aufruf auf, um das nächste Ereignis abzurufen. Der Antworttext enthält die Ereignisdaten. De Antwort-Header enthalten die Anforderungs-ID und andere Informationen.
- Nachverfolgungs-Header propagieren – Ruft den X-Ray-Nachverfolgungs-Header aus dem `Lambda-Runtime-Trace-Id`-Header in der API-Antwort ab. Legen Sie die `_X_AMZN_TRACE_ID`-Umgebungsvariable lokal mit demselben Wert fest. Das X-Ray-SDK verwendet diesen Wert, um Ablaufverfolgungsdaten zwischen Services miteinander zu verbinden.
- Context-Objekt erstellen – Erstellt ein Objekt mit Kontextinformationen aus Umgebungsvariablen und Headern in den API-Antworten.
- Funktions-Handler aufrufen – Übergibt Ereignis und Context-Objekt an den Handler.
- Antwort verarbeiten – Ruft die API für die Aufrufantwort ab, um die Antwort aus dem Handler zu veröffentlichen.
- Fehler verarbeiten – Ruft die API für Aufruffehler auf, wenn ein Fehler auftritt.
- Bereinigen Gibt nicht verwendete Ressourcen frei, sendet Daten an andere Services oder führt zusätzliche Aufgaben aus, bevor das nächste Ereignis abgerufen wird.

Eintrittspunkt

Der Eintrittspunkt einer benutzerdefinierten Laufzeit ist eine ausführbare Datei mit dem Namen `bootstrap`. Bei der Bootstrap-Datei kann es sich um die Laufzeit handeln oder es wird eine andere Datei aufgerufen, die die Laufzeit erstellt. Wenn das Stammverzeichnis Ihres Bereitstellungspakets keine Datei mit dem Namen `bootstrap` enthält, sucht Lambda in den Funktionsschichten nach der Datei. Wenn die `bootstrap`-Datei nicht existiert oder nicht ausgeführt werden kann, gibt die Funktion beim Aufruf einen `Runtime.InvalidEntrypoint`-Fehler zurück.

Hier ist eine `bootstrap` Beispieldatei, die eine gebündelte Version von Node.js verwendet, um eine JavaScript Runtime in einer separaten Datei mit dem Namen auszuführen. `runtime.js`

Example bootstrap

```
#!/bin/sh
    cd $LAMBDA_TASK_ROOT
    ./node-v11.1.0-linux-x64/bin/node runtime.js
```

Implementieren von Antwort-Streaming in einer benutzerdefinierten Laufzeitumgebung

Für Antwort-Streaming-Funktionen haben die Endpunkte `response` und `error` ein leicht geändertes Verhalten, das es der Laufzeit ermöglicht, Teilantworten an den Client zu streamen und Nutzlasten in Paketen zurückzugeben. Weitere Informationen zu dem spezifischen Verhalten finden Sie unter:

- `/runtime/invocation/AwsRequestId/response` – Gibt den `Content-Type`-Header von der Laufzeit zum Senden an den Client weiter. Lambda gibt die Nutzlasten der Antwort in Blöcken über HTTP/1.1-Blocktransfer-Codierungsschema zurück. Der Antwort-Stream darf maximal 20 MiB groß sein. Um die Antwort an Lambda zu streamen, muss die Laufzeit:
  - Den `Lambda-Runtime-Function-Response-Mode`-HTTP-Header auf `streaming` festlegen.
  - Legen Sie den `Transfer-Encoding`-Header auf `chunked` fest.
  - Die Antwort in Übereinstimmung mit der HTTP/1.1-Blocktransfer-Codierungsspezifikation schreiben.
  - Schließt die zugrunde liegende Verbindung, nachdem sie die Antwort erfolgreich geschrieben hat.
- `/runtime/invocation/AwsRequestId/error` – Die Laufzeit kann diesen Endpunkt verwenden, um Funktions- oder Laufzeitfehler an Lambda zu melden, das auch den `Transfer-Encoding`-Header akzeptiert. Dieser Endpunkt kann nur aufgerufen werden, bevor die Laufzeit mit dem Senden einer Aufrufantwort beginnt.
- Melden von Fehlern in der Mitte des Prozesses mit Hilfe von Fehler-Trailern in `/runtime/invocation/AwsRequestId/response` – Um Fehler zu melden, die auftreten, nachdem die Laufzeit mit dem Schreiben der Antwort auf den Aufruf begonnen hat, kann die Laufzeit optional HTTP-Trailing-Header namens `Lambda-Runtime-Function-Error-Type` und `Lambda-Runtime-Function-Error-Body` anfügen. Lambda behandelt dies als erfolgreiche Antwort und leitet die Fehler-Metadaten, die die Laufzeit bereitstellt, an den Client weiter.

> **Note**
>
> Um nachgestellte `Trailer`-Header anzufügen, muss die Laufzeit den Header-Wert am Anfang der HTTP-Anfrage festlegen. Dies ist eine Voraussetzung der HTTP/1.1-Blocktransfer-Codierungs-Spezifikation.

- `Lambda-Runtime-Function-Error-Type` – Der Fehlertyp, auf den die Laufzeit gestoßen ist. Dieser Header besteht aus einem Zeichenfolgen-Wert. Lambda akzeptiert jede Zeichenfolge, wir empfehlen jedoch ein Format von*<category.reason>*. Beispiel, `Runtime.APIKeyNotFound`.
- `Lambda-Runtime-Function-Error-Body` – Base64-kodierte Informationen über den Fehler.

Erstellung benutzerdefinierter Laufzeiten für Lambda Managed Instances

Lambda Managed Instances verwenden dieselbe Runtime-API wie Lambda-Funktionen (Standard). Es gibt jedoch wesentliche Unterschiede darin, wie benutzerdefinierte Laufzeiten implementiert werden müssen, um das Modell der gleichzeitigen Ausführung von Managed Instances zu unterstützen.

Gleichzeitige Bearbeitung von Anfragen

Der Hauptunterschied bei der Erstellung benutzerdefinierter Laufzeiten für verwaltete Instanzen ist die Unterstützung gleichzeitiger Aufrufe. Im Gegensatz zu Lambda-Funktionen (Standard), bei denen die Laufzeit jeweils einen Aufruf verarbeitet, können Managed Instances mehrere Aufrufe gleichzeitig innerhalb einer einzigen Ausführungsumgebung verarbeiten.

Ihre benutzerdefinierte Laufzeit muss:

- Support gleichzeitiger **`/next`** Anfragen — Die Runtime kann mehrere gleichzeitige Aufrufe an die nächste Aufruf-API tätigen, bis zu dem in der `AWS_LAMBDA_MAX_CONCURRENCY` Umgebungsvariablen angegebenen Grenzwert.
- Gleichzeitige **`/response`** Anfragen verarbeiten — Bei mehreren Aufrufen kann die Aufruf-Antwort-API gleichzeitig aufgerufen werden.
- Implementieren Sie eine threadsichere Bearbeitung von Anfragen — Stellen Sie sicher, dass sich gleichzeitige Aufrufe nicht gegenseitig behindern, indem Sie gemeinsam genutzte Ressourcen und den Status ordnungsgemäß verwalten.

- Verwenden Sie eine eindeutige Anfrage IDs — Verfolgen Sie jeden Aufruf separat, indem Sie den `Lambda-Runtime-Aws-Request-Id` Header verwenden, um Antworten den entsprechenden Anfragen zuzuordnen.

Implementierungsmuster

Ein typisches Implementierungsmuster für Laufzeiten von Managed Instances beinhaltet die Erstellung von Worker-Threads oder Prozessen zur Verarbeitung gleichzeitiger Aufrufe:

1. Das Parallelitätslimit auslesen — Lesen Sie bei der Initialisierung die `AWS_LAMBDA_MAX_CONCURRENCY` Umgebungsvariable, um zu bestimmen, wie viele gleichzeitige Aufrufe unterstützt werden sollen.
2. Worker-Pool erstellen — Initialisieren Sie einen Worker-Pool (Threads, Prozesse oder asynchrone Aufgaben), der dem Parallelitätslimit entspricht.
3. Worker-Verarbeitungsschleife — Jeder Worker unabhängig:
   - Ruft `/runtime/invocation/next` auf, um ein Aufrufereignis abzurufen
   - Ruft den Funktionshandler mit den Ereignisdaten auf
   - Sendet die Antwort an `/runtime/invocation/AwsRequestId/response`
   - Wiederholt die Schleife

Weitere Überlegungen

- Protokollierungsformat — Verwaltete Instances unterstützen nur das JSON-Protokollformat. Stellen Sie sicher, dass Ihre Laufzeit die `AWS_LAMBDA_LOG_FORMAT` Umgebungsvariable respektiert und nur das JSON-Format verwendet. Weitere Informationen finden Sie unter Konfiguration der JSON- und Klartext-Protokollformate.
- Gemeinsam genutzte Ressourcen — Seien Sie vorsichtig, wenn Sie gemeinsam genutzte Ressourcen wie das `/tmp` Verzeichnis mit gleichzeitigen Aufrufen verwenden. Implementieren Sie geeignete Schließmechanismen, um Rennbedingungen zu vermeiden.

Weitere Informationen zu Ausführungsumgebungen für Lambda Managed Instances finden Sie unterGrundlegendes zur Ausführungsumgebung von Lambda Managed Instances.

Tutorial: Erstellen einer benutzerdefinierten Laufzeit

In diesem Tutorial erstellen Sie eine Lambda-Funktion mit einer benutzerdefinierten Laufzeit. Sie beginnen, indem Sie die Laufzeit in das Bereitstellungspaket der Funktion einfügen. Anschließend migrieren Sie sie zu einer Ebene, die Sie unabhängig von der Funktion verwalten. Schließlich geben Sie die Laufzeitebene frei, indem Sie ihre ressourcenbasierte Berechtigungsrichtlinie aktualisieren.

Voraussetzungen

In diesem Tutorial wird davon ausgegangen, dass Sie über Kenntnisse zu den grundlegenden Lambda-Operationen und der Lambda-Konsole verfügen. Sofern noch nicht geschehen, befolgen Sie die Anweisungen unter Erstellen einer Lambda-Funktion mit der Konsole, um Ihre erste Lambda-Funktion zu erstellen.

Um die folgenden Schritte durchzuführen, benötigen Sie die AWS CLI Version 2. Befehle und die erwartete Ausgabe werden in separaten Blöcken aufgeführt:

```
aws --version
```

Die Ausgabe sollte folgendermaßen aussehen:

```
aws-cli/2.13.27 Python/3.11.6 Linux/4.14.328-248.540.amzn2.x86_64 exe/x86_64.amzn.2
```

Bei langen Befehlen wird ein Escape-Zeichen (\) wird verwendet, um einen Befehl über mehrere Zeilen zu teilen.

Verwenden Sie auf Linux und macOS Ihren bevorzugten Shell- und Paket-Manager.

> **Note**
>
> In Windows werden einige Bash-CLI-Befehle, die Sie häufig mit Lambda verwenden (z. B. `zip`), von den integrierten Terminals des Betriebssystems nicht unterstützt. Um eine in Windows integrierte Version von Ubuntu und Bash zu erhalten, installieren Sie das Windows-Subsystem für Linux. Die CLI-Beispielbefehle in diesem Handbuch verwenden die Linux-Formatierung. Befehle, die Inline-JSON-Dokumente enthalten, müssen neu formatiert werden, wenn Sie die Windows-CLI verwenden.

Sie benötigen eine IAM-Rolle, um eine Lambda Funktion zu erstellen. Die Rolle benötigt die Berechtigung zum Senden von Protokollen an CloudWatch Logs und zum Zugriff auf die AWS-Services, die Ihre Funktion verwendet. Wenn Sie nicht über eine Rolle für die Funktionserstellung verfügen, erstellen Sie jetzt eine.

So erstellen Sie eine Ausführungsrolle

1. Öffnen Sie die Seite Roles (Rollen) in der IAM-Konsole.
2. Wählen Sie Create role aus.
3. Erstellen Sie eine Rolle mit den folgenden Eigenschaften.
   - Trusted entity (Vertrauenswürdige Entität) – Lambda.
   - Permissions (Berechtigungen) – AWSLambdaBasicExecutionRole.
   - Role name (Name der Rolle – **`lambda-role`**.

   Die Richtlinie AWSLambdaBasicExecutionRole verfügt über die Berechtigungen, die die Funktion zum Schreiben von Protokollen in CloudWatch Logs benötigt.

Erstellen einer -Funktion

Erstellen Sie eine Lambda-Funktion mit einer benutzerdefinierten Laufzeit. Dieses Beispiel enthält zwei Dateien: eine `bootstrap`-Laufzeitdatei und einen Funktions-Handler. Beide sind in Bash implementiert.

1. Erstellen Sie ein Verzeichnis für das Projekt und wechseln Sie dann zu diesem Verzeichnis.

   ```
   mkdir runtime-tutorial
   cd runtime-tutorial
   ```

2. Erstellen Sie eine neue Datei mit dem Namen `bootstrap`. Dies ist die benutzerdefinierte Laufzeit.

   Example bootstrap

   ```
   #!/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
```

Die Laufzeit lädt ein Funktionsskript aus dem Bereitstellungspaket. Sie verwendet zwei Variablen, um das Skript zu finden. `LAMBDA_TASK_ROOT` gibt an, wohin das Paket extrahiert wurde, und `_HANDLER` enthält den Namen des Skripts.

Nachdem die Laufzeit das Funktionsskript geladen hat, verwendet sie die Laufzeit-API, um ein Aufrufereignis aus Lambda abzurufen, übergibt das Ereignis an den Handler und sendet die Antwort zurück an Lambda. Um die Anforderungs-ID abzurufen, speichert die Laufzeit die Header aus der API-Antwort in einer temporären Datei und liest den `Lambda-Runtime-Aws-Request-Id`-Header aus der Datei.

> **Note**
>
> Laufzeit besitzen zusätzliche Verantwortlichkeiten, darunter Fehlerbehandlung und Bereitstellung von Kontextinformationen für den Handler. Details hierzu finden Sie unter Voraussetzungen.

3. Erstellen Sie ein Skript für die Funktion. Das folgende Beispielskript definiert eine Handler-Funktion, die Ereignisdaten annimmt, protokolliert sie in `stderr` und gibt sie zurück.

Example function.sh

```
function handler () {
  EVENT_DATA=$1
  echo "$EVENT_DATA" 1>&2;
  RESPONSE="Echoing request: '$EVENT_DATA'"

  echo $RESPONSE
}
```

Das `runtime-tutorial`-Verzeichnis sollte jetzt wie folgt aussehen:

```
runtime-tutorial
# bootstrap
# function.sh
```

4. Konvertieren Sie die Dateien zu ausführbaren Dateien und fügen Sie sie einem ZIP-Dateiarchiv hinzu. Dies ist das Bereitstellungspaket.

```
chmod 755 function.sh bootstrap
zip function.zip function.sh bootstrap
```

5. Erstellen Sie eine Funktion mit dem Namen `bash-runtime`. Geben Sie für `--role` den ARN Ihrer Lambda-Ausführungsrolle ein.

```
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. Die Funktion aufrufen.

```
aws lambda invoke --function-name bash-runtime --payload '{"text":"Hello"}'
 response.txt --cli-binary-format raw-in-base64-out
```

Die cli-binary-format-Option ist erforderlich, wenn Sie AWS CLI Version 2 verwenden. Um dies zur Standardeinstellung zu machen, führen Sie `aws configure set cli-binary-format raw-in-base64-out` aus. Weitere Informationen finden Sie unter Von AWS CLI

unterstützte globale Befehlszeilenoptionen im AWS Command Line Interface-Benutzerhandbuch für Version 2.

Das Ergebnis sollte ungefähr wie folgt aussehen:

```
{
    "StatusCode": 200,
    "ExecutedVersion": "$LATEST"
}
```

7. Überprüfen Sie die Antwort.

```
cat response.txt
```

Das Ergebnis sollte ungefähr wie folgt aussehen:

```
Echoing request: '{"text":"Hello"}'
```

Erstellen einer Ebene

Um Laufzeitcode und Funktionscode voneinander zu trennen, erstellen Sie eine Ebene, die nur die Laufzeit enthält. Mit Ebenen können Sie die Abhängigkeiten Ihrer Funktion unabhängig entwickeln. Sie können darüber hinaus die Speichernutzung reduzieren, wenn Sie dieselbe Ebene für mehrere Funktionen verwenden. Weitere Informationen finden Sie unter Verwaltung von Lambda-Abhängigkeiten mit Ebenen.

1. Erstellen Sie eine ZIP-Datei, die die `bootstrap`-Datei enthält.

```
zip runtime.zip bootstrap
```

2. Erstellen Sie eine Ebene mit dem publish-layer-version-Befehl.

```
aws lambda publish-layer-version --layer-name bash-runtime --zip-file fileb://
runtime.zip
```

Hierdurch wird die erste Version der Ebene erstellt.

Aktualisieren der Funktion

Um die Laufzeitebene in der Funktion zu verwenden, konfigurieren Sie die Funktion für die Verwendung der Ebene und entfernen den Laufzeitcode aus der Funktion.

1. Aktualisieren Sie die Konfiguration der Funktion, um die Ebene einzubeziehen.

   ```
   aws lambda update-function-configuration --function-name bash-runtime \
   --layers arn:aws:lambda:us-east-1:123456789012:layer:bash-runtime:1
   ```

   Dadurch wird die Laufzeit zur Funktion im `/opt`-Verzeichnis hinzugefügt. Um sicherzustellen, dass Lambda die Laufzeit in der Ebene verwendet, müssen Sie die `boostrap`-Datei aus dem Bereitstellungspaket der Funktion entfernen, wie in den nächsten beiden Schritten gezeigt.

2. Erstellen Sie eine ZIP-Datei, die den Funktionscode enthält.

   ```
   zip function-only.zip function.sh
   ```

3. Aktualisieren Sie den Funktionscode so, dass er nur das Handler-Skript enthält.

   ```
   aws lambda update-function-code --function-name bash-runtime --zip-file fileb://
   function-only.zip
   ```

4. Rufen Sie die Funktion auf, um zu überprüfen, ob sie mit der Laufzeitebene funktioniert.

   ```
   aws lambda invoke --function-name bash-runtime --payload '{"text":"Hello"}'
    response.txt --cli-binary-format raw-in-base64-out
   ```

   Die cli-binary-format-Option ist erforderlich, wenn Sie AWS CLI Version 2 verwenden. Um dies zur Standardeinstellung zu machen, führen Sie `aws configure set cli-binary-format raw-in-base64-out` aus. Weitere Informationen finden Sie unter Von AWS CLI unterstützte globale Befehlszeilenoptionen im AWS Command Line Interface-Benutzerhandbuch für Version 2.

   Das Ergebnis sollte ungefähr wie folgt aussehen:

   ```
   {
       "StatusCode": 200,
       "ExecutedVersion": "$LATEST"
   }
   ```

5. Überprüfen Sie die Antwort.

```
cat response.txt
```

Das Ergebnis sollte ungefähr wie folgt aussehen:

```
Echoing request: '{"text":"Hello"}'
```

Aktualisieren der Laufzeit

1. Um Informationen zur Ausführungsumgebung zu protokollieren, aktualisieren Sie das Laufzeitskript, sodass es Umgebungsvariablen ausgibt.

Example bootstrap

```
#!/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. Erstellen Sie eine ZIP-Datei mit der neuen Version der `bootstrap`-Datei.

```
zip runtime.zip bootstrap
```

3. Erstellt eine neue Version der `bash-runtime`-Ebene.

```
aws lambda publish-layer-version --layer-name bash-runtime --zip-file fileb://
runtime.zip
```

4. Konfigurieren Sie die Funktion für die Verwendung der neuen Version der Ebene.

```
aws lambda update-function-configuration --function-name bash-runtime \
--layers arn:aws:lambda:us-east-1:123456789012:layer:bash-runtime:2
```

Freigeben der Ebene

Um eine Ebene mit einem anderen AWS-Konto gemeinsam zu nutzen, fügen Sie der ressourcenbasierten Richtlinie der Ebene eine kontenübergreifende Berechtigungsanweisung hinzu. Führen Sie den Befehl add-layer-version-permission aus und geben Sie die Konto-ID als `principal` an. In jeder Anweisung können Sie einem einzelnen Konto, allen Konten oder einer Organisation in AWS Organizations eine Berechtigung erteilen.

Im folgenden Beispiel wird dem Konto 111122223333 Zugriff auf Version 2 der `bash-runtime`-Ebene gewährt.

```
aws lambda add-layer-version-permission \
  --layer-name bash-runtime \
  --version-number 2 \
  --statement-id xaccount \
  --action lambda:GetLayerVersion \
  --principal 111122223333 \
  --output text
```

Die Ausgabe sollte folgendermaßen oder ähnlich aussehen:

```
{"Sid":"xaccount","Effect":"Allow","Principal":
{"AWS":"arn:aws:iam::111122223333:root"},"Action":"lambda:GetLayerVersion","Resource":"arn:aws:
east-1:123456789012:layer:bash-runtime:2"}
```

Berechtigungen gelten nur für eine Version mit einer Ebene. Wiederholen Sie den Vorgang bei jeder Erstellung einer neuen Ebenenversion.

Bereinigen

Löschen Sie alle Versionen der Ebene.

```
aws lambda delete-layer-version --layer-name bash-runtime --version-number 1
aws lambda delete-layer-version --layer-name bash-runtime --version-number 2
```

Da die Funktion einen Verweis auf Version 2 der Ebene enthält, ist diese nach wie vor in Lambda vorhanden. Die Funktion funktioniert weiter. Es können jedoch keine weiteren Funktionen für die Verwendung der gelöschten Version konfiguriert werden. Wenn Sie die Liste der Ebenen in der Funktion ändern, müssen Sie eine neue Version angeben oder die gelöschte Ebene auslassen.

Löschen Sie die Funktion mit dem Befehl delete-function.

```
aws lambda delete-function --function-name bash-runtime
```

Open-Source-Repositorys

AWS Lambda bietet eine Vielzahl von Open-Source-Tools, -Bibliotheken und -Komponenten, mit denen Sie Ihre serverlosen Anwendungen erstellen, anpassen und optimieren können. Zu diesen Ressourcen gehören Runtime-Interface-Clients, Ereignisbibliotheken, Container-Basis-Images, Entwicklungstools und Beispielprojekte, die von verwaltet werden AWS und auf GitHub dieser verfügbar sind. Durch die Nutzung dieser Open-Source-Repositorys können Sie die Funktionen von Lambda erweitern, benutzerdefinierte Laufzeiten erstellen, Ereignisse aus verschiedenen AWS Diensten verarbeiten und tiefere Einblicke in die Leistung Ihrer Funktion gewinnen. Diese Seite bietet einen Überblick über die wichtigsten Open-Source-Projekte, die die Lambda-Entwicklung unterstützen.

Laufzeit-Schnittstellenclients

Lambda Runtime Interface Clients (RICs) sind Open-Source-Bibliotheken, die die Runtime-API implementieren und die Interaktion zwischen Ihrem Funktionscode und dem Lambda-Service

verwalten. Diese Clients übernehmen den Empfang von Aufrufereignissen, die Weitergabe von Kontextinformationen und die Meldung von Fehlern.

Die von den verwalteten Laufzeiten und Container-Basisimages von Lambda verwendeten Laufzeit-Schnittstellenclients werden als Open Source veröffentlicht. Wenn Sie benutzerdefinierte Laufzeiten erstellen oder bestehende erweitern, können Sie diese Open-Source-Bibliotheken verwenden, um Ihre Implementierung zu vereinfachen. Die folgenden GitHub Open-Source-Repositorys enthalten den Quellcode für Lambdas: RICs

- Node.js-Laufzeit-Schnittstellenclient
- Python-Laufzeit-Schnittstellenclient
- Java-Laufzeit-Schnittstellenclient
- Ruby-Laufzeit-Schnittstellenclient
- .NET-Laufzeit-Schnittstellenclient
- Rust-Runtime-Schnittstellenclient
- Go-Laufzeit-Schnittstellenclient
- Swift Runtime Interface-Client (experimentell)
- C++-Laufzeitschnittstellenclient (experimentell)
- Lambda-Basis-Images

Weitere Informationen zur Verwendung dieser Clients zum Erstellen benutzerdefinierter Laufzeiten finden Sie unter the section called “Erstellen einer benutzerdefinierten Laufzeit”.

Ereignisbibliotheken

Lambda-Ereignisbibliotheken bieten Typdefinitionen und Hilfsprogramme für die Verarbeitung von Ereignissen aus verschiedenen AWS Diensten. Diese Bibliotheken unterstützen Sie beim Parsen und Verarbeiten von Ereignisdaten auf typsichere Weise, wodurch die Arbeit mit Ereignissen aus Services wie Amazon S3, Amazon DynamoDB und Amazon API Gateway vereinfacht wird.

AWS Stellt für kompilierte Sprachen die folgenden Ereignisbibliotheken bereit:

- Java-Ereignisbibliothek
- .NET-Ereignisbibliotheken
- Go-Ereignisbibliothek
- Rust-Ereignisbibliothek

Für interpretierte Sprachen wie Node.js, Python und Ruby können Ereignisse direkt als JSON-Objekte geparst werden, ohne dass eine separate Bibliothek erforderlich ist. Entwickler, die Node.js und Python verwenden, können jedoch Powertools for nutzen AWS Lambda, das integrierte Schemas für AWS Ereignisse bereitstellt, die Typhinweise, Datenvalidierung und Funktionen bieten, die denen von kompilierten Sprachbibliotheken ähneln.

- Powertools für TypeScript
- Powertools für Python

Container-Basisimages

AWS stellt Open-Source-Container-Basis-Images bereit, die Sie als Ausgangspunkt für die Erstellung von Container-Images für Ihre Lambda-Funktionen verwenden können. Diese Basisimages enthalten den Laufzeit-Schnittstellenclient und andere Komponenten, die zum Ausführen Ihrer Funktionen in der Lambda-Ausführungsumgebung erforderlich sind.

Weitere Informationen zu den verfügbaren Basisimages und deren Verwendung finden Sie im Repository für AWS -Basisimages sowie unter the section called “Container-Images”.

Entwicklungstools

AWS bietet zusätzliche Open-Source-Entwicklungstools, mit denen Sie Ihre Lambda-Funktionen erstellen und optimieren können:

Powertools für AWS Lambda

Powertools for AWS Lambda vereinfacht die serverlose Entwicklung mit wichtigen Hilfsprogrammen zur Vermeidung doppelter Verarbeitung, Stapelverarbeitung für die Verarbeitung mehrerer Datensätze und Kafka-Benutzerbibliothek. Diese Funktionen helfen Ihnen, die Komplexität des Codes und den Betriebsaufwand zu minimieren.

Sie können auch die integrierte Ereignisschemavalidierung, die strukturierte Protokollierung und Ablaufverfolgung sowie die Integration von Parameterspeichern nutzen, die darauf ausgelegt sind, die Erstellung produktionsreifer Lambda-Funktionen zu beschleunigen und gleichzeitig gut konzipierte Best Practices zu befolgen AWS .

GitHub Repositorien:

- Python

- TypeScript
- Java
- .NET

Java-Entwicklungstools

- Java Profiler (experimentell) – Ein Tool zur Profilerstellung für Java-Lambda-Funktionen.
- Java-Bibliotheken — Ein Repository, das eine umfassende Sammlung von Java-Bibliotheken und -Tools für die Lambda-Entwicklung enthält, einschließlich wichtiger Projekte wie JUnit Testdienstprogramme und Profilierungstools.
- Serverless-Java-Container – Eine Bibliothek, mit der Sie vorhandene Java-Anwendungen mit minimalen Änderungen in Lambda ausführen können.

.NET-Entwicklungstools

AWS Lambda Das.NET-Repository stellt .NET-Bibliotheken und Tools für die Lambda-Entwicklung bereit, einschließlich wichtiger Projekte wie AWS Lambda Tools für die .NET CLI und .NET Core-Server zum Hosten von.NET Core-Anwendungen.

Beispielprojekte

Erkunden Sie eine umfassende Sammlung von Beispielprojekten und -anwendungen für Lambda in den Serverless-Land-Repositorys. Diese Beispiele veranschaulichen verschiedene Anwendungsfälle, Integrationsmuster und bewährte Verfahren für Lambda, die Ihnen den Einstieg in Ihre Serverless-Anwendungen erleichtern.

AWS Lambda Funktionen konfigurieren

Erfahren Sie, wie Sie die wichtigsten Funktionen und Optionen für Ihre Lambda-Funktion mithilfe der Lambda-API oder der Konsole konfigurieren.

[.zip-Dateiarchive]

Erstellen Sie ein Paket zur Bereitstellung von Lambda-Funktionen, wenn Sie Abhängigkeiten, benutzerdefinierte Laufzeitebenen oder Dateien außerhalb Ihres Funktionscodes einbeziehen möchten. Das Bereitstellungspaket ist ein ZIP-Dateiarchiv, das Ihren Funktionscode und Ihre Abhängigkeiten enthält.

[Container-Images]

Verwenden Sie Container-Images, um Ihren Funktionscode und Ihre Abhängigkeiten zu verpacken, wenn Sie mehr Kontrolle über den Erstellungsprozess benötigen oder wenn Ihre Funktion benutzerdefinierte Laufzeitkonfigurationen erfordert. Mit Tools wie Docker CLI können Sie Lambda-Funktionen als Container-Images erstellen, testen und bereitstellen.

[Arbeitsspeicher]

Hier erfahren Sie, wie und wann das Funktionsgedächtnis erweitert wird.

[Flüchtiger Speicher]

Erfahren Sie, wie und wann Sie die temporäre Speicherkapazität Ihrer Funktion erhöhen können.

[Timeout (Zeitüberschreitung)]

Erfahren Sie, wie und wann Sie den Timeout-Wert Ihrer Funktion erhöhen können.

[Dauerhafte Konfiguration]

Ermöglichen Sie die Ausführung Ihrer Lambda-Funktionen für längere Zeiträume — bis zu einem Jahr — und behalten Sie dabei den Status zwischen den Aufrufen bei. Konfigurieren Sie Ausführungs-Timeouts, die Beibehaltung des Status und das Versionsverwaltungsverhalten für Workflows mit langer Laufzeit und Status. Zuverlässige Funktionen ermöglichen es Ihnen, komplexe, mehrstufige Prozesse zu erstellen, die unterbrochen und wieder aufgenommen werden können und Unterbrechungen automatisch beheben können.

Umgebungsvariablen

Sie können Ihren Funktionscode portabel machen und Geheimnisse aus Ihrem Code heraushalten, indem Sie sie mithilfe von Umgebungsvariablen in der Konfiguration Ihrer Funktion speichern.

Ausgehende Netzwerke

Sie können Ihre Lambda-Funktion mit AWS Ressourcen in einer Amazon VPC verwenden. Wenn Sie Ihre Funktion mit einer VPC verbinden, können Sie auf Ressourcen in einem privaten Subnetz wie relationale Datenbanken und Caches zugreifen.

Eingehende Netzwerke

Sie können einen Schnittstellen-VPC-Endpunkt verwenden, um Ihre Lambda-Funktionen aufzurufen, ohne das öffentliche Internet zu nutzen.

Dateisystem

Sie können Ihre Lambda-Funktion verwenden, um ein Amazon EFS in ein lokales Verzeichnis einzubinden. Ein Dateisystem ermöglicht es Ihrem Funktionscode, sicher und mit hoher Gleichzeitigkeit auf gemeinsam genutzte Ressourcen zuzugreifen und diese zu ändern.

Aliasnamen

Sie können Ihre Clients so konfigurieren, dass sie eine bestimmte Lambda-Funktionsversion aufrufen, indem Sie einen Alias verwenden, anstatt den Client zu aktualisieren.

Versionen

Durch die Veröffentlichung einer Version Ihrer Funktion können Sie Ihren Code und Ihre Konfiguration als separate Ressource speichern, die nicht geändert werden kann.

Tags

Verwenden Sie Tags, um die attributebasierte Zugriffskontrolle (ABAC) zu aktivieren, Ihre Lambda-Funktionen zu organisieren und Berichte über Ihre Funktionen mithilfe der Dienste AWS Cost Explorer oder AWS Billing and Cost Management zu filtern und zu generieren.

Antwort-Streaming

Sie können Ihre Lambda-Funktion so konfigurieren, dass Antwortnutzlasten zurück URLs an Clients gestreamt werden. Antwort-Streaming kann für latenzempfindliche Anwendungen von Vorteil sein, da es die Leistung in der Zeit bis zum ersten Byte (TTFB) verbessert. Dies liegt daran, dass Sie Teilantworten an den Client zurücksenden können, sobald sie verfügbar sind.

Darüber hinaus können Sie Response-Streaming verwenden, um Funktionen zu erstellen, die größere Nutzlasten zurückgeben.

Bereitstellen von Lambda-Funktionen als ZIP-Dateiarchive

Wenn Sie eine Lambda-Funktion erstellen, verpacken Sie Ihren Funktionscode in einem Bereitstellungspaket. Lambda unterstützt zwei Arten von Bereitstellungspaketen: Container-Images und .zip-Archive. Der Workflow zum Erstellen einer Funktion hängt vom Typ des Bereitstellungspakets ab. Zur Konfiguration einer Funktion, die als Container-Image definiert ist, siehe the section called “Container-Images”.

Sie können die Lambda-Konsole und die Lambda-API verwenden, um eine mit einem ZIP-Dateiarchiv definierte Funktion zu erstellen. Sie können auch eine aktualisierte ZIP-Datei uploaden, um den Funktionscode zu ändern.

> **Note**
>
> Sie können den Bereitstellungspakettyp (.zip oder Container-Image) für eine vorhandene Funktion nicht ändern. Sie können zum Beispiel eine Container-Image-Funktion nicht so umwandeln, dass sie ein .zip-Archiv verwendet. Sie müssen eine neue Funktion erstellen.

Themen

- Erstellen der Funktion
- Verwenden des Konsolencode-Editors
- Aktualisieren des Funktionscodes
- Ändern der Laufzeit
- Ändern der Architektur
- Verwenden der Lambda-API
- Herunterladen des Funktionscodes
- CloudFormation
- Verschlüsselung von Lambda-.zip-Bereitstellungspaketen

Erstellen der Funktion

Wenn Sie eine Funktion erstellen, die mit einem ZIP-Dateiarchiv definiert ist, wählen Sie eine Codevorlage, die Sprachversion und die Ausführungsrolle für die Funktion aus. Sie fügen Ihren Funktionscode hinzu, nachdem Lambda die Funktion erstellt hat.

So erstellen Sie die Funktion

1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.
2. Wählen Sie Funktion erstellen.
3. Wählen Sie Von Grund auf neu erstellen oder Vorlage verwenden, um Ihre Funktion zu erstellen.
4. Führen Sie unter Basic information (Grundlegende Informationen) die folgenden Schritte aus:
   a. Geben Sie für Funktionsname den Funktionsnamen ein. Funktionsnamen sind auf eine Länge von 64 Zeichen beschränkt.
   b. Wählen Sie für Laufzeit die Sprachversion aus, die für Ihre Funktion verwendet werden soll.
   c. (Optional) Für-Architekturwählen Sie die Befehlssatz-Architektur aus, die für Ihre Funktion verwendet werden soll. Die Standardarchitektur ist x86_64. Stellen Sie beim Erstellen des Bereitstellungspakets für Ihre Funktion sicher, dass es damit kompatibel istBefehlssatz-Architekturaus.
5. (Optional) Erweitern Sie unter Berechtigungen die Option Standardausführungsrolle ändern. Sie können eine neue Ausführungsrolle erstellen oder eine vorhandene Rolle verwenden.
6. (Optional) Erweitern Sie den Abschnitt Advanced Settings (Erweiterte Einstellungen). Sie können eine Codesignatur-Konfiguration für die Funktion wählen. Sie können auch eine (Amazon VPC) für die Funktion konfigurieren, auf die zugegriffen werden soll.
7. Wählen Sie Funktion erstellen.

Lambda erstellt die neue Funktion. Sie können nun die Konsole verwenden, um den Funktionscode hinzuzufügen und andere Funktionsparameter und -funktionen zu konfigurieren. Anweisungen zur Codebereitstellung finden Sie auf der Handler-Seite für die Laufzeit, die Ihre Funktion verwendet.

Node.js

Bereitstellen von Node.js Lambda-Funktionen mit ZIP-Dateiarchiven

Python

Arbeiten mit ZIP-Dateiarchiven und Python-Lambda-Funktionen

Ruby

Bereitstellen von Ruby-Lambda-Funktionen mit ZIP-Dateiarchiven

Java

Bereitstellen von Java-Lambda-Funktionen mit ZIP- oder JAR-Dateiarchiven

Go

- Bereitstellen von Lambda-Go-Funktionen mit ZIP-Dateiarchiven

C#

- Erstellen und Bereitstellen von C#-Lambda-Funktionen mit ZIP-Dateiarchiven

PowerShell

- PowerShell-Lambda-Funktionen mit .zip-Dateiarchiven bereitstellen

Verwenden des Konsolencode-Editors

Die Konsole erstellt eine Lambda-Funktion mit einer einzigen Quelldatei. Bei Skriptsprachen können Sie diese Datei bearbeiten und mit dem integrierten Code-Editor weitere Dateien hinzufügen. Klicken Sie auf Save (Speichern), um die Änderungen zu speichern. Um Ihren Code auszuführen, wählen Sie Test.

Wenn Sie Ihren Funktionscode speichern, erstellt die Lambda-Konsole ein Bereitstellungspaket für das ZIP-Dateiarchiv. Wenn Sie Ihren Funktionscode außerhalb der Konsole (mit einer IDE) entwickeln, müssen Sie ein Bereitstellungspaket erstellen, um Ihren Code in die Lambda-Funktion hochzuladen.

Aktualisieren des Funktionscodes

Bei Skriptsprachen (Node.js, Python und Ruby) können Sie Ihren Funktionscode im eingebetteten Code-Editor bearbeiten. Wenn der Code größer als 3 MB ist, oder wenn Sie Bibliotheken hinzufügen müssen, oder für Sprachen, die der Editor nicht unterstützt (Java, Go, C#), müssen Sie Ihren Funktionscode als .zip-Archiv uploaden. Wenn das ZIP-Dateiarchiv kleiner als 50 MB ist, können Sie das ZIP-Dateiarchiv von Ihrem lokalen Computer uploaden. Wenn die Datei größer als 50 MB ist, müssen Sie die Datei aus einem Amazon S3-Bucket in die Funktion uploaden.

So können Sie den Funktionscode als .zip-Archiv uploaden

1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.
2. Wählen Sie die zu aktualisierende Funktion aus und wählen Sie die Code-Registerkarte.
3. Wählen Sie unter Codequelle die Option Upload von aus.
4. Wählen Sie .zip-Datei und dann Hochladen.

- Wählen Sie in der Dateiauswahl die neue Image-Version aus, wählen Sie Öffnen und dann Speichern.

5. (Alternative zu Schritt 4) Wählen Sie den Amazon-S3-Standort.

   - Geben Sie in das Textfeld die S3-Link-URL des ZIP-Dateiarchivs ein und wählen Sie dann Speichern.

Ändern der Laufzeit

Wenn Sie die Funktionskonfiguration aktualisieren, um eine neue Laufzeit zu verwenden, müssen Sie möglicherweise den Funktionscode aktualisieren, damit dieser mit der neuen Laufzeit kompatibel ist. Wenn Sie die Funktionskonfiguration aktualisieren, um eine andere Laufzeit zu verwenden,mussstellen Sie neuen Funktionscode bereit, der mit Laufzeit und Architektur kompatibel ist. Anweisungen zum Erstellen eines Bereitstellungspakets für den Funktionscode finden Sie auf der Handler-Seite für die Laufzeit, die die Funktion verwendet.

Die Basis-Images Node.js 20, Python 3.12, Java 21, .NET 8, Ruby 3.3 und höher basieren auf dem minimalen Container-Image von Amazon Linux 2023. Frühere Basis-Images verwenden Amazon Linux 2. AL2023 bietet mehrere Vorteile gegenüber Amazon Linux 2, darunter einen geringeren Bereitstellungsaufwand und aktualisierte Versionen von Bibliotheken wie`glibc`. Weitere Informationen finden Sie unter Introducing the Amazon Linux 2023 runtime for AWS Lambda im AWS -Compute-Blog.

So ändern Sie die Laufzeit

1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.
2. Wählen Sie die zu aktualisierende Funktion aus und wählen Sie die Code-Registerkarte.
3. Blättern Sie nach unten bis zum Abschnitt Laufzeiteinstellungen, der sich unter dem Code-Editor befindet.
4. Wählen Sie Bearbeiten aus.

   a. Wählen Sie für Runtime (Laufzeit) die Laufzeit-ID aus.

   b. Geben Sie im Feld Handler den Dateinamen und en Handler für die Funktion ein.

   c. Für-Architekturwählen Sie die Befehlssatz-Architektur aus, die für Ihre Funktion verwendet werden soll.

5. Wählen Sie Speichern.

Ändern der Architektur

Bevor Sie die Befehlssatzarchitektur ändern können, müssen Sie sicherstellen, dass der Code Ihrer Funktion mit der Zielarchitektur kompatibel ist.

Wenn Sie Node.js, Python oder Ruby verwenden und Ihren Funktionscode im eingebettetenEditorkann der vorhandene Code ohne Änderung ausgeführt werden.

Wenn Sie jedoch Ihren Funktionscode mithilfe eines Bereitstellungspakets im ZIP-Dateiarchiv angeben, müssen Sie ein neues ZIP-Dateiarchiv vorbereiten, das korrekt für die Ziel-Laufzeit- und Anleitungssatz-Architektur kompiliert und erstellt wird. Anweisungen dazu finden Sie auf der Handler-Seite für Ihre Funktions-Laufzeit.

So ändern Sie die Befehlssatz-Architektur

1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.
2. Wählen Sie die zu aktualisierende Funktion aus und wählen Sie die Code-Registerkarte.
3. Wählen Sie unter Laufzeiteinstellungen die Option Bearbeiten.
4. Für-Architekturwählen Sie die Befehlssatz-Architektur aus, die für Ihre Funktion verwendet werden soll.
5. Wählen Sie Speichern.

Verwenden der Lambda-API

Um eine Funktion zu erstellen und zu konfigurieren, die ein ZIP-Dateiarchiv verwendet, verwenden Sie die folgenden API-Operationen:

- CreateFunction
- UpdateFunctionCode
- UpdateFunctionConfiguration

Herunterladen des Funktionscodes

Sie können die aktuelle unveröffentlichte Version (`$LATEST`) Ihres Funktionscodes (.zip) über die Lambda-Konsole herunterladen. Um dies zu tun, müssen Sie zunächst über die folgenden IAM-Berechtigungen verfügen:

- `iam:GetPolicy`
- `iam:GetPolicyVersion`
- `iam:GetRole`
- `iam:GetRolePolicy`
- `iam:ListAttachedRolePolicies`
- `iam:ListRolePolicies`
- `iam:ListRoles`

So laden Sie den Funktionscode (.zip) herunter

1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.
2. Wählen Sie die Funktion aus, für die Sie den Funktionscode (.zip) herunterladen möchten.
3. Wählen Sie in der Funktionsübersicht die Schaltfläche Herunterladen und anschließend Funktionscode herunterladen (.zip).
    - Wählen Sie alternativ AWS SAM Datei herunterladen, um eine SAM-Vorlage zu generieren und herunterzuladen, die auf der Konfiguration Ihrer Funktion basiert. Sie können auch Beide herunterladen wählen, um sowohl die ZIP-Vorlage als auch die SAM-Vorlage herunterzuladen.

CloudFormation

Sie können CloudFormation damit eine Lambda-Funktion erstellen, die ein ZIP-Dateiarchiv verwendet. In Ihrer CloudFormation Vorlage spezifiziert die `AWS::Lambda::Function` Ressource die Lambda-Funktion. Eine Beschreibung der Eigenschaften in der `AWS::Lambda::Function` Ressource finden Sie AWS::Lambda::Functionim AWS CloudFormation Benutzerhandbuch.

Legen Sie in der `AWS::Lambda::Function`-Ressource die folgenden Eigenschaften fest, um eine Funktion zu erstellen, die als Zip-Datei-Archiv definiert ist:

- AWS::Lambda::Function
    - PackageType — Eingestellt auf`Zip`.
    - Code – Geben Sie den Namen des Amazon-S3-Buckets und den ZIP-Dateinamen in die Textfelder `S3Bucket` und `S3Key`. Für Node.js oder Python können Sie Inline-Quellcode Ihrer Lambda-Funktion bereitstellen.

- Laufzeit – Legt den Laufzeitwert fest.
- Architektur — Stellen Sie den Architekturwert auf ein, `arm64` um den AWS Graviton2-Prozessor zu verwenden. Der Standardwert ist`x86_64`aus.

Verschlüsselung von Lambda-.zip-Bereitstellungspaketen

Lambda bietet im Ruhezustand immer serverseitige Verschlüsselung für .zip-Bereitstellungspakete und Funktionskonfigurationsdetails mit einem AWS KMS key an. Standardmäßig verwendet Lambda einen AWS-eigener Schlüssel. Wenn dieses Standardverhalten zu Ihrem Arbeitsablauf passt, müssen Sie nichts weiter einrichten. AWS berechnet Ihnen keine Gebühren für die Verwendung dieses Schlüssels.

Wenn Sie möchten, können Sie stattdessen einen vom AWS KMS Kunden verwalteten Schlüssel bereitstellen. Sie können dies tun, um die Drehung des KMS-Schlüssels zu steuern oder die Anforderungen Ihrer Organisation für die Verwaltung von KMS-Schlüsseln zu erfüllen. Wenn Sie einen vom Kunden verwalteten Schlüssel verwenden, können nur Benutzer Ihres Kontos mit Zugriff auf den KMS-Schlüssel den Code oder die Konfiguration der Funktion anzeigen oder verwalten.

Für vom Kunden verwaltete Schlüssel fallen AWS KMS Standardgebühren an. Weitere Informationen finden Sie unter AWS Key Management Service Preise.

Erstellen eines kundenseitig verwalteten Schlüssels

Sie können einen symmetrischen, vom Kunden verwalteten Schlüssel erstellen, indem Sie den AWS-Managementkonsole, oder den verwenden. AWS KMS APIs

Einen symmetrischen kundenverwalteten Schlüssel erstellen

Befolgen Sie die Schritte zur Erstellung symmetrischer Verschlüsselung Erstellung symmetrischer KMS-Schlüssel mit symmetrischer Verschlüsselung im AWS Key Management Service - Entwicklerhandbuch.

Berechtigungen

Schlüsselrichtlinie

Schlüsselrichtlinien steuern den Zugriff auf den vom Kunden verwalteten Schlüssel. Jeder vom Kunden verwaltete Schlüssel muss über genau eine Schlüsselrichtlinie verfügen, die aussagt, wer

den Schlüssel wie verwenden kann. Weitere Informationen finden Sie unter So ändern Sie eine Schlüsselrichtlinie im AWS Key Management Service -Entwicklerhandbuch.

Wenn Sie einen vom Kunden verwalteten Schlüssel verwenden, um ein .zip-Bereitstellungspaket zu verschlüsseln, fügt Lambda dem Schlüssel keinen Zuschuss hinzu. Stattdessen muss Ihre AWS KMS Schlüsselrichtlinie es Lambda ermöglichen, die folgenden AWS KMS API-Operationen in Ihrem Namen aufzurufen:

- km: GenerateDataKey
- kms:Decrypt

Die folgende Beispielschlüsselrichtlinie ermöglicht es allen Lambda-Funktionen im Konto 111122223333, die erforderlichen AWS KMS Operationen für den angegebenen vom Kunden verwalteten Schlüssel aufzurufen:

Example AWS KMS wichtige Richtlinie

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:*"
                }
            }
        }
    ]
```

```
}
```

Weitere Informationen zur Fehlerbehebung beim Schlüsselzugriff finden Sie im AWS Key Management Service -Entwicklerhandbuch.

Prinzipalberechtigungen

Wenn Sie einen vom Kunden verwalteten Schlüssel verwenden, um ein .zip-Bereitstellungspaket zu verschlüsseln, können nur Prinzipale mit Zugriff auf diesen Schlüssel auf das .zip-Bereitstellungspaket zugreifen. Principals, die keinen Zugriff auf den vom Kunden verwalteten Schlüssel haben, können beispielsweise das .zip-Paket nicht über die vorsignierte S3-URL herunterladen, die in der Antwort enthalten ist. GetFunction Im Abschnitt `Code` der Antwort wird ein `AccessDeniedException` zurückgegeben.

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"
    },
 ...
```

Weitere Informationen zu Berechtigungen für AWS KMS Schlüssel finden Sie unter Authentifizierung und Zugriffskontrolle für. AWS KMS

Verwenden Sie einen vom Kunden verwalteten Schlüssel für Ihr .zip-Bereitstellungspaket

Verwenden Sie die folgenden API-Parameter, um vom Kunden verwaltete Schlüssel für .zip-Bereitstellungspakete zu konfigurieren:

- Source KMSKey Arn: Verschlüsselt das Quell-ZIP-Bereitstellungspaket (die Datei, die Sie hochladen).
- KMSKeyArn: Verschlüsselt Umgebungsvariablen und Lambda-Snapshots SnapStart.

Wenn `SourceKMSKeyArn` und `KMSKeyArn` beide angegeben sind, verwendet Lambda den Schlüssel `KMSKeyArn`, um die entpackte Version des Pakets zu verschlüsseln, das Lambda zum Aufrufen der Funktion verwendet. Wenn `SourceKMSKeyArn` angegeben ist, aber `KMSKeyArn` nicht, verwendet Lambda ein Von AWS verwalteter Schlüssel, um die entpackte Version des Pakets zu verschlüsseln.

Lambda console

So fügen Sie beim Erstellen einer Funktion die kundenverwaltete Schlüsselverschlüsselung hinzu

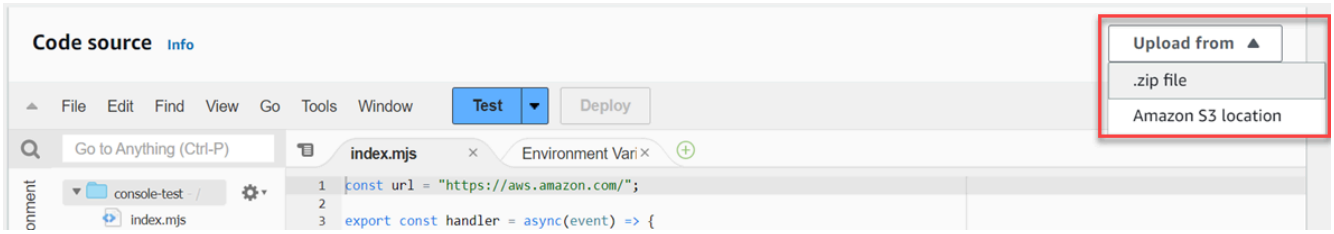
1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.
2. Wählen Sie Funktion erstellen.
3. Klicken Sie auf Ohne Vorgabe erstellen oder Container-Image.
4. Führen Sie unter Basic information (Grundlegende Informationen) die folgenden Schritte aus:
   a. Geben Sie für Funktionsname den Funktionsnamen ein.
   b. Wählen Sie für Laufzeit die Sprachversion aus, die für Ihre Funktion verwendet werden soll.
5. Erweitern Sie Erweiterte Einstellungen und wählen Sie dann Verschlüsselung mit einem vom AWS KMS Kunden verwalteten Schlüssel aktivieren aus.
6. Wählen Sie einen kundenverwalteten Schlüssel.
7. Wählen Sie Funktion erstellen.

Um die vom Kunden verwaltete Schlüsselverschlüsselung zu entfernen oder einen anderen Schlüssel zu verwenden, müssen Sie das .zip-Bereitstellungspaket erneut hochladen.

So fügen Sie einer bestehenden Funktion eine vom Kunden verwaltete Schlüsselverschlüsselung hinzu

1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.
2. Wählen Sie den Namen einer Funktion aus.
3. Wählen Sie im Bereich Codequelle die Option Hochladen von aus.

4. Wählen Sie eine .zip-Datei oder einen Amazon-S3-Speicherort.



5. Laden Sie die Datei hoch oder geben Sie den Amazon-S3-Speicherort ein.
6. Wählen Sie Verschlüsselung mit einem vom AWS KMS Kunden verwalteten Schlüssel aktivieren aus.
7. Wählen Sie einen kundenverwalteten Schlüssel.
8. Wählen Sie Speichern.

AWS CLI

So fügen Sie beim Erstellen einer Funktion die kundenverwaltete Schlüsselverschlüsselung hinzu

Im folgenden Beispiel wird create-function erstellt:

- `--code`: Gibt den lokalen Pfad zum ZIP-Bereitstellungspaket (`ZipFile`) und den vom Kunden verwalteten Schlüssel zur Verschlüsselung an (`SourceKMSKeyArn`).
- `--kms-key-arn`: Gibt den vom Kunden verwalteten Schlüssel zur Verschlüsselung der Umgebungsvariablen und der entpackten Version des Bereitstellungspakets an.

```
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
```

Im folgenden Beispiel wird create-function erstellt:

- `--code`: Gibt den Speicherort der .zip-Datei in einem Amazon S3 S3-Bucket (`S3Bucket`,`S3Key`,`S3ObjectVersion`) und den vom Kunden verwalteten Schlüssel zur Verschlüsselung an (`SourceKMSKeyArn`).

- `--kms-key-arn`: Gibt den vom Kunden verwalteten Schlüssel zur Verschlüsselung der Umgebungsvariablen und der entpackten Version des Bereitstellungspakets an.

```
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
```

So fügen Sie einer bestehenden Funktion eine vom Kunden verwaltete Schlüsselverschlüsselung hinzu

Im folgenden Beispiel: update-function-code

- `--zip-file`: Gibt den lokalen Pfad zum .zip-Bereitstellungspaket an.
- `--source-kms-key-arn`: Gibt den vom Kunden verwalteten Schlüssel zum Verschlüsseln der komprimierten Version des Bereitstellungspakets an. Lambda verwendet einen AWS eigenen Schlüssel, um das entpackte Paket für Funktionsaufrufe zu verschlüsseln. Wenn Sie einen vom Kunden verwalteten Schlüssel verwenden möchten, um die entpackte Version des Pakets zu verschlüsseln, führen Sie den Befehl mit der Option aus. update-function-configuration`--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
```

Im folgenden Beispiel: update-function-code

- `--s3-bucket`: Gibt den Speicherort der .zip-Datei in einem Amazon-S3-Bucket an.
- `--s3-key`: Gibt den Amazon-S3-Schlüssel des Bereitstellungspakets an.
- `--s3-object-version`: Für versionierte Objekte, die Version des zu verwendenden Bereitstellungspaketobjekts.

- `--source-kms-key-arn`: Gibt den vom Kunden verwalteten Schlüssel zum Verschlüsseln der komprimierten Version des Bereitstellungspakets an. Lambda verwendet einen AWS eigenen Schlüssel, um das entpackte Paket für Funktionsaufrufe zu verschlüsseln. Wenn Sie einen vom Kunden verwalteten Schlüssel verwenden möchten, um die entpackte Version des Pakets zu verschlüsseln, führen Sie den Befehl mit der Option aus. update-function-configuration`--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
```

So entfernen Sie die vom Kunden verwaltete Schlüsselverschlüsselung aus einer bestehenden Funktion

Im folgenden update-function-codeBeispiel wird der lokale Pfad zum ZIP-Bereitstellungspaket `--zip-file` angegeben. Wenn Sie diesen Befehl ohne die `--source-kms-key-arn` Option ausführen, verwendet Lambda einen AWS eigenen Schlüssel, um die komprimierte Version des Bereitstellungspakets zu verschlüsseln.

```
aws lambda update-function-code \
  --function-name myFunction \
  --zip-file fileb://myFunction.zip
```

Erstellen einer Lambda-Funktion mit einem Container-Image

Der Code Ihrer AWS Lambda Funktion besteht aus Skripten oder kompilierten Programmen und deren Abhängigkeiten. Sie verwenden ein Bereitstellungspaket, um Ihren Funktionscode in Lambda bereitzustellen. Lambda unterstützt zwei Arten von Bereitstellungspaketen: Container-Images und ZIP-Dateiarchiven.

Es gibt drei Möglichkeiten, ein Container-Image für eine Lambda-Funktion zu erstellen:

- Verwenden eines AWS Basis-Images für Lambda

  Die AWS -Basis-Images sind mit einer Sprachlaufzeit, einem Laufzeitschnittstellen-Client zur Verwaltung der Interaktion zwischen Lambda und Ihrem Funktionscode und einem Laufzeitschnittstellen-Emulator für lokale Tests vorinstalliert.

- Es wird ein AWS reines Betriebssystem-Basis-Image verwendet

  AWS Basis-Images nur für Betriebssysteme enthalten eine Amazon Linux-Distribution und den Runtime-Interface-Emulator. Diese Images werden häufig verwendet, um Container-Images für kompilierte Sprachen wie Go und Rust sowie für eine Sprache oder Sprachversion zu erstellen, für die Lambda kein Basis-Image bereitstellt, wie Node.js 19. Sie können reine OS-Basis-Images auch verwenden, um eine benutzerdefinierte Laufzeit zu implementieren. Um das Image mit Lambda kompatibel zu machen, müssen Sie den Laufzeitschnittstellen-Client für Ihre Sprache in das Image aufnehmen.

- Verwenden eines AWS Nicht-Base-Images

  Sie können auch ein alternatives Basis-Image aus einer anderen Container-Registry verwenden. Sie können auch ein von Ihrer Organisation erstelltes benutzerdefiniertes Image verwenden. Um das Image mit Lambda kompatibel zu machen, müssen Sie den Laufzeitschnittstellen-Client für Ihre Sprache in das Image aufnehmen.

> **Tip**
>
> Um die Zeit zu reduzieren, die benötigt wird, bis Lambda-Container-Funktionen aktiv werden, siehe die Docker-Dokumentation unter Verwenden mehrstufiger Builds. Um effiziente Container-Images zu erstellen, folgen Sie den Bewährte Methoden für das Schreiben von Dockerfiles.

Um eine Lambda-Funktion aus einem Container-Image zu erstellen, erstellen Sie Ihr Image lokal und laden es in ein Amazon Elastic Container Registry (Amazon ECR)-Repository hoch. Wenn Sie ein von einem AWS Marketplace-Verkäufer bereitgestelltes Container-Image verwenden, müssen Sie das Image zuerst in Ihr privates Amazon-ECR-Repository klonen. Geben Sie dann den Repository-URI an, wenn Sie die Funktion erstellen. Das Amazon ECR-Repository muss sich im selben Format AWS-Region wie die Lambda-Funktion befinden. Sie können eine Funktion mit einem Bild in einem anderen AWS Konto erstellen, sofern sich das Bild in derselben Region wie die Lambda-Funktion befindet. Weitere Informationen finden Sie unter Kontoübergreifende Berechtigungen von Amazon ECR.

> **Note**
>
> Lambda unterstützt Amazon-ECR-FIPS-Endpunkte für Container-Images nicht. Wenn Ihr Repository-URI ein `ecr-fips` enthält, verwenden Sie einen FIPS-Endpunkt. Beispiel: `111122223333.dkr.ecr-fips.us-east-1.amazonaws.com`.

Auf dieser Seite werden die Basis-Image-Typen und Anforderungen für die Erstellung von Lambda-kompatiblen Container-Images erläutert.

> **Note**
>
> Sie können den Bereitstellungspakettyp (.zip oder Container-Image) für eine vorhandene Funktion nicht ändern. Sie können zum Beispiel eine Container-Image-Funktion nicht so umwandeln, dass sie ein .zip-Archiv verwendet. Sie müssen eine neue Funktion erstellen.

Themen

- Voraussetzungen
- Verwenden eines AWS Basis-Images für Lambda
- Es wird ein AWS reines Betriebssystem-Basis-Image verwendet
- Verwenden eines AWS Nicht-Base-Images
- Laufzeitschnittstellen-Clients
- Amazon-ECR-Berechtigungen
- Lebenszyklus der Funktion

Voraussetzungen

Installieren Sie die AWS CLI -Version 2 und die Docker-CLI. Beachten Sie außerdem die folgenden Anforderungen:

- Das Container-Image muss die Verwenden der Lambda-Laufzeit-API für benutzerdefinierte Laufzeiten implementieren. Die AWS Open-Source-Laufzeitschnittstellen-Clients implementieren die API. Sie können Ihrem bevorzugten Basis-Image einen Laufzeitschnittstellen-Client hinzufügen, damit es mit Lambda kompatibel ist.
- Das Container-Image muss auf einem schreibgeschützten Dateisystem laufen können. Ihr Funktionscode kann auf ein beschreibbares `/tmp`-Verzeichnis mit einem Speicherplatz zwischen 512 MB bis 10 240 MB, in 1-MB-Schritten, zugreifen.
- Der Lambda-Standardbenutzer muss in der Lage sein, alle Dateien zu lesen, die zum Ausführen Ihres Funktionscodes erforderlich sind. Lambda folgt den bewährten Methoden für die Sicherheit, indem es einen Standard-Linux-Benutzer mit den geringsten Berechtigungen definiert. Das bedeutet, dass Sie in Ihrem Dockerfile keinen BENUTZER angeben müssen. Stellen Sie sicher, dass Ihr Anwendungscode nicht auf Dateien angewiesen ist, von denen andere Linux-Benutzer nicht ausgeführt werden können.
- Lambda unterstützt nur Linux-basierte Container-Images.
- Lambda bietet Multi-Architektur-Basis-Images. Das Image, das Sie für Ihre Funktion erstellen, muss jedoch nur auf eine der Architekturen abzielen. Lambda unterstützt keine Funktionen, die Container-Images mit mehreren Architekturen verwenden.

Verwenden eines AWS Basis-Images für Lambda

Sie können eines der AWS -Basis-Images für Lambda verwenden, um das Container-Image für Ihren Funktionscode zu erstellen. Die Basis-Images sind mit einer Sprachlaufzeit und anderen Komponenten vorgeladen, die zum Ausführen eines Container-Images in Lambda erforderlich sind. Sie fügen Ihren Funktionscode und Ihre Abhängigkeiten dem Basis-Image hinzu und verpacken es dann als Container-Image.

AWS stellt regelmäßig Updates für die AWS Basis-Images für Lambda bereit. Wenn Ihr Dockerfile den Image-Namen in der FROM-Eigenschaft enthält, ruft Ihr Docker-Client die aktuelle Version des Images aus dem Amazon-ECR-Repository ab.. Um das aktualisierte Basis-Image verwenden zu können, müssen Sie Ihr Container-Image neu erstellen und den Funktionscode aktualisieren.

Die Basis-Images Node.js 20, Python 3.12, Java 21, .NET 8, Ruby 3.3 und höher basieren auf dem minimalen Container-Image von Amazon Linux 2023. Frühere Basis-Images verwenden Amazon Linux 2. AL2023 bietet mehrere Vorteile gegenüber Amazon Linux 2, darunter einen geringeren Bereitstellungsaufwand und aktualisierte Versionen von Bibliotheken wie`glibc`.

AL2023-basierte Images verwenden `microdnf` (symbolisiert als`dnf`) als Paketmanager anstelle von`yum`, dem Standard-Paketmanager in Amazon Linux 2. `microdnf`ist eine eigenständige Implementierung von. `dnf` Eine Liste der Pakete, die in AL2 023-basierten Images enthalten sind, finden Sie in den Spalten Minimal Container unter Comparing packages installed on Amazon Linux 2023 Container Images. Weitere Informationen zu den Unterschieden zwischen AL2 023 und Amazon Linux 2 finden Sie unter Einführung in die Amazon Linux 2023 Runtime for AWS Lambda im AWS Compute-Blog.

> **Note**
>
> Um AL2 023-basierte Images lokal auszuführen, auch mit AWS Serverless Application Model (AWS SAM), müssen Sie Docker-Version 20.10.10 oder höher verwenden.

Um ein Container-Image mit einem AWS Basis-Image zu erstellen, wählen Sie die Anweisungen für Ihre bevorzugte Sprache aus:

- Node.js
- TypeScript(verwendet ein Node.js -Basisimage)
- Python
- Java
- Go
- .NET
- Ruby

Es wird ein AWS reines Betriebssystem-Basis-Image verwendet

AWS Basis-Images nur für Betriebssysteme enthalten eine Amazon Linux-Distribution und den Runtime-Interface-Emulator. Diese Images werden häufig verwendet, um Container-Images für kompilierte Sprachen wie Go und Rust sowie für eine Sprache oder Sprachversion zu erstellen, für die Lambda kein Basis-Image bereitstellt, wie Node.js 19. Sie können reine OS-Basis-Images auch

verwenden, um eine benutzerdefinierte Laufzeit zu implementieren. Um das Image mit Lambda kompatibel zu machen, müssen Sie den Laufzeitschnittstellen-Client für Ihre Sprache in das Image aufnehmen.

Tags (Markierungen)	Laufzeit	Betriebssystem	Dockerfile	Ablehnung
al2023	Reine OS-Laufzeit	Amazon Linux 2023	Dockerfile für reine Betriebssystem-Runtime on GitHub	30. Juni 2029
al2	Reine OS-Laufzeit	Amazon Linux 2	Dockerfile nur für Betriebssystem, Runtime aktiviert GitHub	31. Juli 2026

Öffentliche Galerie von Amazon Elastic Container Registry: gallery.ecr. aws/lambda/provided

Verwenden eines AWS Nicht-Base-Images

Lambda unterstützt jedes Image, das einem der folgenden Image-Manifestformate entspricht:

- Docker Image Manifest V2 Schema 2 (mit Docker-Version 1.10 und neuer)
- Open Container Initiative (OCI)-Spezifikationen (v1.0.0 und höher)

Lambda unterstützt eine maximale unkomprimierte Image-Größe von 10 GB, einschließlich aller Ebenen.

> **Note**
>
> - Um das Image mit Lambda kompatibel zu machen, müssen Sie den Laufzeitschnittstellen-Client für Ihre Sprache in das Image aufnehmen.
> - Für eine optimale Leistung sollte die Größe Ihres Bildmanifests unter 25 400 Byte liegen. Um die Größe des Bildmanifests zu reduzieren, minimieren Sie die Anzahl der Ebenen in Ihrem Bild und reduzieren Sie Anmerkungen.

Laufzeitschnittstellen-Clients

Wenn Sie ein reines OS-Basis-Image oder ein alternatives Basis-Image verwenden, müssen Sie den Laufzeitschnittstellen-Client in das Image einbinden. Der Runtime-Schnittstellenclient muss den erweiternVerwenden der Lambda-Laufzeit-API für benutzerdefinierte Laufzeiten, der die Interaktion zwischen Lambda und Ihrem Funktionscode verwaltet. AWS stellt Open-Source-Runtime-Interface-Clients für die folgenden Sprachen bereit:

- Node.js
- Python
- Java
- .NET
- Go
- Ruby
- Rust —

Wenn Sie eine Sprache verwenden, für die kein Runtime-Interface-Client AWS zur Verfügung steht, müssen Sie Ihren eigenen erstellen.

Amazon-ECR-Berechtigungen

Bevor Sie eine Lambda-Funktion aus einem Container-Image erstellen, müssen Sie das Image lokal erstellen und es in ein Amazon-ECR-Repository hochladen. Geben Sie beim Erstellen der Funktion den URI des Amazon-ECR-Repositorys an.

Vergewissern Sie sich, dass die Berechtigungen für den Benutzer oder die Rolle, der/die die Funktion erstellt, `GetRepositoryPolicy` und `SetRepositoryPolicy` umfassen.

Verwenden Sie beispielsweise die IAM-Konsole, um eine Rolle mit der folgenden Richtlinie zu erstellen:

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"
    }
  ]
}
```

Richtlinien für das Amazon-ECR-Repository

Für eine Funktion im selben Konto wie das Container-Image in Amazon ECR können Sie Ihrem Amazon-ECR-Repository die Berechtigungen `ecr:BatchGetImage` und `ecr:GetDownloadUrlForLayer` hinzufügen. Das folgende Beispiel zeigt die Mindestrichtlinie:

```
{
        "Sid": "LambdaECRImageRetrievalPolicy",
        "Effect": "Allow",
        "Principal": {
          "Service": "lambda.amazonaws.com"
        },
        "Action": [
          "ecr:BatchGetImage",
          "ecr:GetDownloadUrlForLayer"
        ]
    }
```

Weitere Informationen zu Amazon-ECR-Repository-Berechtigungen finden Sie unter Private Repository-Richtlinien im Benutzerhandbuch zu Amazon Elastic Container Registry.

Wenn das Amazon-ECR-Repository diese Berechtigungen nicht enthält, versucht Lambda, sie automatisch hinzuzufügen. Lambda kann Berechtigungen nur hinzufügen, wenn der Lambda aufrufende Prinzipal über `ecr:getRepositoryPolicy`- und `ecr:setRepositoryPolicy`-Berechtigungen verfügt.

Um Ihre Repository-Berechtigungen für Amazon ECR anzuzeigen oder zu bearbeiten, befolgen Sie die Anweisungen unter Festlegung einer privaten Repository-Richtlinienanweisung im Benutzerhandbuch zu Amazon Elastic Container Registry.

Kontoübergreifende Berechtigungen von Amazon ECR

Ein anderes Konto in derselben Region kann eine Funktion erstellen, die ein Container-Image verwendet, das Ihrem Konto gehört. Im folgenden Beispiel benötigt die Berechtigungsrichtlinie Ihres Amazon-ECR-Repository die folgenden Anweisungen, um den Zugriff auf Kontonummer 123456789012 zu gewähren.

- CrossAccountPermission— Ermöglicht dem Konto 123456789012 das Erstellen und Aktualisieren von Lambda-Funktionen, die Bilder aus diesem ECR-Repository verwenden.
- Lambda ECRImage CrossAccountRetrievalPolicy — Lambda setzt den Status einer Funktion irgendwann auf inaktiv, wenn sie über einen längeren Zeitraum nicht aufgerufen wird. Diese Anweisung ist erforderlich, damit Lambda das Container-Image zur Optimierung und Zwischenspeicherung im Namen der Funktion abrufen kann, die 123456789012 besitzt.

Example– Ihrem Repository kontoübergreifende Berechtigungen hinzufügen

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"
    }
  ]
}
```

Um Zugriff auf mehrere Konten zu gewähren, fügen Sie das Konto der IDs Principal-Liste in der `CrossAccountPermission` Policy und der Condition-Bewertungsliste in der hinzu. `LambdaECRImageCrossAccountRetrievalPolicy`

Wenn Sie mit mehreren Konten in einer AWS Organisation arbeiten, empfehlen wir, dass Sie jede Konto-ID in der ECR-Berechtigungsrichtlinie auflisten. Dieser Ansatz entspricht der bewährten AWS Sicherheitsmethode, enge Berechtigungen in IAM-Richtlinien festzulegen.

Zusätzlich zu den Lambda-Berechtigungen muss der Benutzer oder die Rolle, der/die die Funktion erstellt, auch über `BatchGetImage`- und `GetDownloadUrlForLayer`-Berechtigungen verfügen.

Lebenszyklus der Funktion

Nachdem Sie ein neues oder aktualisiertes Container-Image hochgeladen haben, optimiert Lambda das Image, bevor die Funktion Aufrufe verarbeiten kann. Der Optimierungsprozess kann einige Sekunden dauern. Die Funktion verbleibt im Zustand `Pending`, bis der Prozess abgeschlossen ist, dann geht der Zustand in `Active` über. Sie können die Funktion erst dann aufrufen, wenn sie den `Active`-Zustand erreicht hat.

Wenn eine Funktion mehrere Wochen lang nicht aufgerufen wird, gewinnt Lambda seine optimierte Version zurück und die Funktion wechselt in den `Inactive`-Zustand. Um die Funktion wieder zu aktivieren, müssen Sie sie aufrufen. Lambda lehnt den ersten Aufruf ab und die Funktion tritt in den `Pending`-Zustand, bis Lambda das Image neu optimiert. Die Funktion kehrt dann zum `Active`-Zustand zurück.

Lambda ruft regelmäßig das zugehörige Container-Image aus dem Amazon ECR Repository ab. Wenn das entsprechende Container-Image in Amazon ECR nicht mehr vorhanden ist oder Berechtigungen widerrufen werden, wechselt die Funktion in den `Failed`-Zustand und Lambda gibt einen Fehler für alle Funktionsaufrufe zurück.

Sie können die Lambda-API verwenden, um Informationen über den Zustand einer Funktion abzurufen. Weitere Informationen finden Sie unter Lambda-Funktionszustände.

Konfigurieren Sie den Speicher der Lambda-Funktion

Lambda weist die Rechenleistung proportional zur Menge des konfigurierten Arbeitsspeichers zu. Arbeitsspeicher ist die Menge an Arbeitsspeicher, die Ihrer Lambda-Funktion zur Laufzeit zur Verfügung steht. Sie können den Arbeitsspeicher und die CPU-Leistung, die Ihrer Funktion zugewiesen sind, mit der Einstellung Arbeitsspeicher erhöhen oder verringern. Sie können Speicher zwischen 128 MB und 10 240 MB in 1-MB-Schritten konfigurieren. Bei 1.769 MB weist eine Funktion das Äquivalent einer vCPU auf (eine vCPU-Sekunde Guthaben pro Sekunde).

Auf dieser Seite wird beschrieben, wie und wann die Speichereinstellung für eine Lambda-Funktion aktualisiert werden muss.

Sections

- Ermitteln der geeigneten Speichereinstellung für eine Lambda-Funktion
- Konfigurieren des Funktionsspeichers (Konsole)
- Konfiguration des Funktionsspeichers (AWS CLI)
- Konfiguration des Funktionsspeichers (AWS SAM)
- Akzeptieren von Empfehlungen für den Funktionsspeicher (Konsole)

Ermitteln der geeigneten Speichereinstellung für eine Lambda-Funktion

Der Speicher ist der wichtigste Hebel zur Steuerung der Ausführung einer Funktion. Die Standardeinstellung, 128 MB, ist die niedrigste mögliche Einstellung. Wir empfehlen, nur 128 MB für einfache Lambda-Funktionen zu verwenden, z. B. solche, die Ereignisse transformieren und an andere AWS -Dienste weiterleiten. Eine höhere Speicherzuweisung kann die Leistung von Funktionen verbessern, die importierte Bibliotheken, Lambda-Ebenen, Amazon Simple Storage Service (Amazon S3) oder Amazon Elastic File System (Amazon EFS) verwenden. Durch die Erweiterung des Arbeitsspeichers erhöht sich proportional die Anzahl der CPUs, wodurch die verfügbare Rechenleistung insgesamt steigt. Wenn eine Funktion an die CPU, das Netzwerk oder den Speicher gebunden ist, kann eine Erhöhung der Speichereinstellung die Leistung erheblich verbessern.

Um die richtige Speicherkonfiguration zu finden, überwachen Sie Ihre Funktionen mit Amazon CloudWatch und stellen Sie Alarme ein, wenn sich der Speicherverbrauch den konfigurierten Höchstwerten nähert. Dies kann helfen, speichergebundene Funktionen zu identifizieren. Bei CPU-gebundenen und IO-gebundenen Funktionen kann die Überwachung der Dauer auch zu

Erkenntnissen führen. In diesen Fällen kann eine Erhöhung des Speichers dazu beitragen, die Rechen- oder Netzwerkengpässe zu beheben.

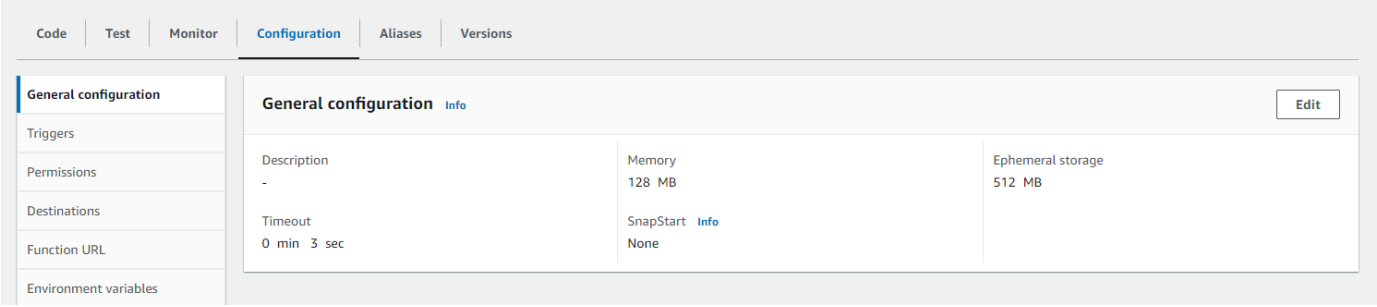
Sie können auch die Verwendung des Open-Source-Tools AWS Lambda Power Tuning in Betracht ziehen. Dieses Tool verwendet AWS Step Functions , um mehrere gleichzeitige Versionen einer Lambda-Funktion an unterschiedlichen Speicherzuweisungen auszuführen und die Leistung zu messen. Die Eingabefunktion wird in Ihrem AWS Konto ausgeführt und führt Live-HTTP-Aufrufe und SDK-Interaktionen durch, um die wahrscheinliche Leistung in einem Live-Produktionsszenario zu messen. Sie können auch einen CI/CD Prozess implementieren, um dieses Tool zur automatischen Messung der Leistung neuer Funktionen zu verwenden, die Sie bereitstellen.

Konfigurieren des Funktionsspeichers (Konsole)

Sie können den Speicher Ihrer Funktion in der Lambda-Konsole konfigurieren.

So aktualisieren Sie den Speicher einer Funktion

1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.
2. Wählen Sie eine Funktion aus.
3. Wählen Sie Configuration (Konfiguration) und dann General configuration (Allgemeine Konfiguration).



4. Wählen Sie unter General configuration (allgemeine Konfiguration) Bearbeiten.
5. Legen Sie für Arbeitsspeicher einen Wert von 128 MB bis 10.240 MB fest.
6. Wählen Sie Speichern.

Konfiguration des Funktionsspeichers (AWS CLI)

Sie können den update-function-configurationBefehl verwenden, um den Speicher Ihrer Funktion zu konfigurieren.

Example

```
aws lambda update-function-configuration \
  --function-name my-function \
  --memory-size 1024
```

Konfiguration des Funktionsspeichers (AWS SAM)

Sie können den AWS Serverless Application Model verwenden, um den Speicher für Ihre Funktion zu konfigurieren. Aktualisieren Sie die MemorySizeEigenschaft in Ihrer `template.yaml` Datei und führen Sie dann sam deploy aus.

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...
```

Akzeptieren von Empfehlungen für den Funktionsspeicher (Konsole)

Wenn Sie Administratorrechte in AWS Identity and Access Management (IAM) haben, können Sie sich dafür entscheiden, Empfehlungen für Lambda-Funktionsspeichereinstellungen von zu erhalten. AWS Compute Optimizer Anweisungen dazu, wie Sie Speicherempfehlungen für Ihr Konto oder Ihr Unternehmen aktivieren, finden Sie unter Aktivieren Ihres Kontos im AWS Compute Optimizer - Benutzerhandbuch.

> **Note**
>
> Compute Optimizer unterstützt nur Funktionen, die die x86_64-Architektur verwenden.

Wenn Sie sich angemeldet haben und Ihre Lambda-Funktion die Anforderungen von Compute Optimizer erfüllt, können Sie Empfehlungen zum Funktionsspeicher von Compute Optimizer in der Lambda-Konsole unter Allgemeine Konfiguration anzeigen und akzeptieren.

Flüchtiger Speicher für Lambda-Funktionen konfigurieren

Lambda bietet flüchtigen Speicher für Funktionen im `/tmp`-Verzeichnis. Dieser Speicher ist temporär und für jede Ausführungsumgebung einzigartig. Sie können die Menge des flüchtigen Speichers, der Ihrer Funktion zugewiesen ist, mithilfe der Einstellung Flüchtiger Speicher steuern. Sie können den flüchtigen Speicher zwischen 512 MB und 10.240 MB in 1-MB-Schritten konfigurieren. Alle in `/tmp` gespeicherten Daten werden im Ruhezustand mit einem von AWS-verwalteten Schlüssel verschlüsselt.

Auf dieser Seite werden allgemeine Anwendungsfälle und die Aktualisierung des kurzlebigen Speichers für eine Lambda-Funktion beschrieben.

Sections

- Häufige Anwendungsfälle für flüchtiger Speicher
- Flüchtigen Speicher konfigurieren (Konsole)
- Konfigurierbarer flüchtiger Speicher (AWS CLI)
- Konfigurierbarer flüchtiger Speicher (AWS SAM)

Häufige Anwendungsfälle für flüchtiger Speicher

Im Folgenden sind einige häufig vorkommende Anwendungsfälle aufgeführt, die von einem erhöhten flüchtigen Speicher profitieren:

- Extract-transform-load (ETL-) Jobs: Erhöhen Sie den temporären Speicher, wenn Ihr Code Zwischenberechnungen durchführt oder andere Ressourcen herunterlädt, um die Verarbeitung abzuschließen. Mehr temporärer Speicherplatz ermöglicht die Ausführung komplexerer ETL-Jobs in Lambda-Funktionen.
- Inferenz durch Machine Learning(ML): Viele Inferenzaufgaben basieren auf großen Referenzdatendateien, einschließlich Bibliotheken und Modellen. Mit mehr ephemerem Speicher können Sie größere Modelle vom Amazon Simple Storage Service (Amazon S3) auf `/tmp` herunterladen und in Ihrer Verarbeitung verwenden.
- Datenverarbeitung: Bei Workloads, die Objekte von Amazon S3 als Reaktion auf S3-Ereignisse herunterladen, ermöglicht mehr `/tmp`-Speicherplatz die Verarbeitung größerer Objekte ohne In-Memory-Verarbeitung. Workloads, bei denen Medien erstellt PDFs oder verarbeitet werden, profitieren ebenfalls von kurzlebiger Speicherung.

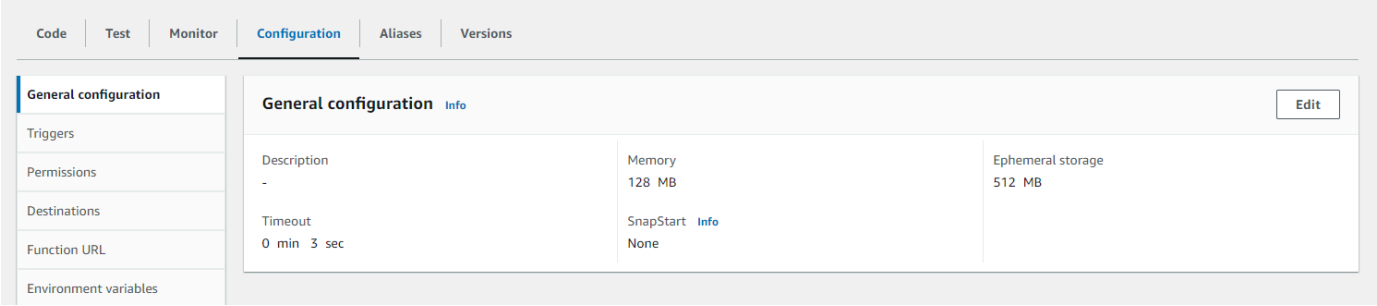
- Grafikverarbeitung: Die Bildverarbeitung ist ein häufiger Anwendungsfall für Lambda-basierte Anwendungen. Bei Workloads, die große TIFF-Dateien oder Satellitenbilder verarbeiten, erleichtert ein kurzlebigerer Speicher die Verwendung von Bibliotheken und die Durchführung der Berechnung in Lambda.

Flüchtigen Speicher konfigurieren (Konsole)

Flüchtiger Speicher können Sie in der Lambda-Konsole konfigurieren.

So ändern Sie den flüchtigen Speicher für eine Funktion

1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.
2. Wählen Sie eine Funktion aus.
3. Wählen Sie Configuration (Konfiguration) und dann General configuration (Allgemeine Konfiguration).



4. Wählen Sie unter General configuration (allgemeine Konfiguration) Bearbeiten.
5. Legen Sie für den flüchtigen Speicher einen Wert zwischen 512 MB und 10.240 MB fest, in 1-MB-Schritten.
6. Wählen Sie Speichern.

Konfigurierbarer flüchtiger Speicher (AWS CLI)

Sie können den update-function-configurationBefehl verwenden, um kurzlebigen Speicher zu konfigurieren.

Example

```
aws lambda update-function-configuration \
  --function-name my-function \
```

```
  --ephemeral-storage '{"Size": 1024}'
```

Konfigurierbarer flüchtiger Speicher (AWS SAM)

Sie können den AWS Serverless Application Model verwenden, um den flüchtigen Speicher für Ihre Funktion zu konfigurieren. Aktualisieren Sie die EphemeralStorageEigenschaft in Ihrer `template.yaml` Datei und führen Sie dann sam deploy aus.

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...
```

Auswahl und Konfiguration einer Befehlssatzarchitektur für Ihre Lambda-Funktion

Die-Befehlssatz-Architektureiner Lambda-Funktion bestimmt den Typ des Computerprozessors, den Lambda zum Ausführen der Funktion verwendet. Lambda bietet eine Auswahl an Befehlssatz-Architekturen:

- arm64 — 64-Bit-ARM-Architektur für den AWS Graviton2-Prozessor.
- x86_64 — 64-Bit-x86-Architektur für x86-basierte Prozessoren.

> **Note**
>
> Die arm64-Architektur ist in den meisten Versionen verfügbar. AWS-Regionen Weitere Informationen finden Sie unter AWS Lambda – Preise. Wählen Sie in der Tabelle mit den Speicherpreisen die Registerkarte Arm-Preis aus und öffnen Sie dann die Dropdownliste Region, um zu sehen, welche Produkte arm64 mit Lambda AWS-Regionen unterstützen. Ein Beispiel für die Erstellung einer Funktion mit der Arm64-Architektur finden Sie unter AWS Lambda Funktionen, die vom Graviton2-Prozessor unterstützt werden. AWS

Themen

- Vorteile der Verwendung von arm64-Architektur
- Anforderungen für die Migration zur arm64-Architektur
- Funktionscode-Kompatibilität mit arm64-Architektur
- Migration zur arm64-Architektur
- Konfigurieren der Befehlssatz-Architektur

Vorteile der Verwendung von arm64-Architektur

Lambda-Funktionen, die die Arm64-Architektur (AWS Graviton2-Prozessor) verwenden, können einen deutlich besseren Preis und eine deutlich bessere Leistung erzielen als die entsprechende Funktion, die auf der x86_64-Architektur ausgeführt wird. Erwägen Sie, arm64 für rechenintensive Anwendungen wie Hochleistungsrechnen, Videocodierung und Simulations-Workloads zu verwenden.

Die Graviton2-CPU verwendet den Neoverse N1-Kern und unterstützt Armv8.2 (einschließlich CRC- und Krypto-Erweiterungen) sowie mehrere andere architektonische Erweiterungen.

Graviton2 reduziert die Lesezeit des Speichers, indem es einen größeren L2-Cache pro vCPU bereitstellt, was die Latenzleistung von Web- und Mobile-Backends, Microservices und Datenverarbeitungssystemen verbessert. Graviton2 bietet auch eine verbesserte Verschlüsselungsleistung und unterstützt Befehlssätze, die die Latenz von CPU-basierten Inferenzen für Machine Learning verbessern.

Weitere Informationen zu Graviton2 finden Sie unter Graviton Processor. AWSAWS

Anforderungen für die Migration zur arm64-Architektur

Wenn Sie eine Lambda-Funktion für die Migration zur arm64-Architektur auswählen, stellen Sie sicher, dass Ihre Funktion die folgenden Anforderungen erfüllt, um eine reibungslose Migration zu gewährleisten:

- Das Bereitstellungspaket enthält nur Open-Source-Komponenten und Quellcode, die Sie steuern, sodass Sie alle notwendigen Updates für die Migration vornehmen können.
- Wenn der Funktionscode Abhängigkeiten von Drittanbietern enthält, bietet jede Bibliothek oder jedes Paket eine arm64-Version.

Funktionscode-Kompatibilität mit arm64-Architektur

Ihr Lambda-Funktionscode muss mit der Befehlssatz-Architektur der Funktion kompatibel sein. Bevor Sie eine Funktion zur arm64-Architektur migrieren, beachten Sie die folgenden Punkte zum aktuellen Funktionscode:

- Wenn Sie Ihren Funktionscode mit dem eingebetteten Code-Editor hinzugefügt haben, läuft Ihr Code wahrscheinlich auf beiden Architekturen ohne Änderung.
- Nach dem Upload Ihres Funktionscodes, müssen Sie neuen Code uploaden, der mit Ihrer Zielarchitektur kompatibel ist.
- Wenn Ihre Funktion Layer verwendet, müssen Sieüberprüfe jede Ebeneum sicherzustellen, dass es mit der neuen Architektur kompatibel ist. Wenn ein Layer nicht kompatibel ist, bearbeiten Sie die Funktion, um die aktuelle Layer-Version durch eine kompatible Layer-Version zu ersetzen.
- Wenn Ihre Funktion Lambda-Erweiterungen verwendet, müssen Sie jede Erweiterung überprüfen, um sicherzustellen, dass sie mit der neuen Architektur kompatibel ist.

- Wenn Ihre Funktion einen Container-Image-Bereitstellungspakettyp verwendet, müssen Sie ein neues Container-Image erstellen, das mit der Architektur der Funktion kompatibel ist.

Migration zur arm64-Architektur

Um eine Lambda-Funktion auf die arm64-Architektur zu migrieren, empfehlen wir die folgenden Schritte:

1. Erstellen Sie die Liste der Abhängigkeiten für Ihre Anwendung oder Ihren Workload. Häufige Abhängigkeiten sind unter anderem:
   - Alle Bibliotheken und Pakete, die die Funktion verwendet.
   - Die Tools, mit denen Sie die Funktion erstellen, bereitstellen und testen, wie Compiler, Test-Suites, Pipelines für Continuous Integration und Continuous Delivery (CI/CD), Bereitstellungstools und Skripte.
   - Die Lambda-Erweiterungen und Tools von Drittanbietern, mit denen Sie die Funktion in der Produktion überwachen.
2. Überprüfen Sie für jede der Abhängigkeiten die Version und prüfen Sie dann, ob arm64-Versionen verfügbar sind.
3. Erstellen Sie eine Umgebung, um Ihre Anwendung zu migrieren.
4. Bootstrap die Anwendung.
5. Testen und debuggen Sie die Anwendung.
6. Testen Sie die Leistung der arm64-Funktion. Vergleichen Sie die Leistung mit der x86_64-Version.
7. Aktualisieren Sie Ihre Infrastruktur-Pipeline, um arm64 Lambda-Funktionen zu unterstützen.
8. Stufen Sie Ihre Bereitstellung in die Produktion ein.

   Verwenden Sie zum Beispiel Alias-Routing-Konfiguration um den Datenverkehr zwischen den Versionen x86 und arm64 der Funktion aufzuteilen und die Leistung und Latenz zu vergleichen.

Weitere Informationen zum Erstellen einer Codeumgebung für die arm64-Architektur, einschließlich sprachspezifischer Informationen für Java, Go, .NET und Python, finden Sie im Repository Erste Schritte mit AWS Graviton. GitHub

Konfigurieren der Befehlssatz-Architektur

Sie können die Befehlssatzarchitektur für neue und bestehende Lambda-Funktionen mithilfe der Lambda-Konsole, AWS SDKs, AWS Command Line Interface (AWS CLI) oder konfigurieren. CloudFormation Führen Sie die folgenden Schritte aus, um die Befehlssatzarchitektur für eine vorhandene Lambda-Funktion von der Konsole aus zu ändern.

1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.
2. Klicken Sie auf den Namen der Funktion, für die Sie die Befehlssatzarchitektur konfigurieren möchten.
3. Wählen Sie auf der Registerkarte Code für den Abschnitt Laufzeiteinstellungen die Option Bearbeiten aus.
4. Unter Architecture (Architektur) wählen Sie die Befehlssatzarchitektur aus, die Ihre Funktion verwenden soll.
5. Wählen Sie Speichern.

Zeitüberschreitung der Lambda-Funktion konfigurieren

Lambda führt Ihren Code für eine festgelegte Zeitspanne aus, bevor ein Timeout erfolgt. Beim Timeout handelt es sich um die maximale Zeitspanne in Sekunden, die eine Lambda-Funktion ausgeführt werden kann. Der Standardwert für diese Einstellung ist 3 Sekunden. Sie können ihn jedoch in Schritten von 1 Sekunde bis zu einem Höchstwert von 900 Sekunden (15 Minuten) anpassen.

Auf dieser Seite wird beschrieben, wie und wann die Timeout-Einstellung für eine Lambda-Funktion aktualisiert werden muss.

Sections

- Ermitteln des geeigneten Timeout-Werts für eine Lambda-Funktion
- Konfigurieren des Timeouts (Konsole)
- Timeout konfigurieren (AWS CLI)
- Timeout konfigurieren (AWS SAM)

Ermitteln des geeigneten Timeout-Werts für eine Lambda-Funktion

Wenn der Timeout-Wert nahe an der durchschnittlichen Dauer einer Funktion liegt, besteht ein höheres Risiko, dass die Funktion unerwartet beendet wird. Die Dauer einer Funktion kann je nach Umfang der Datenübertragung und -verarbeitung sowie der Latenz aller Dienste, mit denen die Funktion interagiert, variieren. Zu den häufigsten Ursachen für Timeouts gehören:

- Downloads von Amazon Simple Storage Service (Amazon S3) sind größer oder dauern länger als der Durchschnitt.
- Eine Funktion sendet eine Anfrage an einen anderen Service, dessen Beantwortung länger dauert.
- Die einer Funktion zur Verfügung gestellten Parameter erfordern eine höhere Rechenkomplexität in der Funktion, wodurch der Aufruf länger dauert.

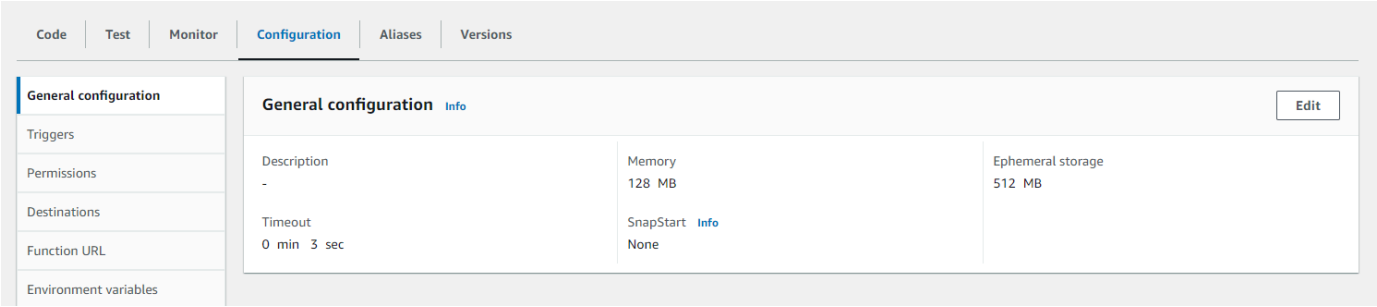
Stellen Sie beim Testen Ihrer Anwendung sicher, dass Ihre Tests die Größe und Menge der Daten sowie realistische Parameterwerte genau wiedergeben. Tests verwenden aus Gründen der Benutzerfreundlichkeit häufig kleine Stichproben. Sie sollten jedoch Datensätze verwenden, die sich an der Obergrenze dessen befinden, was für Ihren Workload vernünftigerweise zu erwarten ist.

Konfigurieren des Timeouts (Konsole)

Sie können das Funktions-Timeout in der Lambda-Konsole konfigurieren.

So ändern Sie das Timeout für eine Funktion

1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.
2. Wählen Sie eine Funktion aus.
3. Wählen Sie Configuration (Konfiguration) und dann General configuration (Allgemeine Konfiguration).



4. Wählen Sie unter General configuration (allgemeine Konfiguration) Bearbeiten.
5. Für Timeout stellen Sie einen Wert zwischen 1 und 900 Sekunden (15 Minuten) ein.
6. Wählen Sie Speichern.

Timeout konfigurieren (AWS CLI)

Sie können den update-function-configurationBefehl verwenden, um den Timeout-Wert in Sekunden zu konfigurieren. Der folgende Beispielbefehl erhöht das Funktions-Timeout auf 120 Sekunden (2 Minuten).

Example

```
aws lambda update-function-configuration \
  --function-name my-function \
  --timeout 120
```

Timeout konfigurieren (AWS SAM)

Sie können den AWS Serverless Application Model verwenden, um den Timeout-Wert in Sekunden zu konfigurieren. Aktualisieren Sie die Eigenschaft Timeout in Ihrer `template.yaml`-Datei und führen Sie dann sam deploy aus.

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...
```

Dauerhafte Lambda-Funktionen konfigurieren

Um eine dauerhafte Ausführung Ihrer Lambda-Funktion zu ermöglichen, müssen Sie spezifische Einstellungen konfigurieren, die steuern, wie lange Ihre Funktion ausgeführt werden kann, wie lange Statusdaten aufbewahrt werden und welche Berechtigungen erforderlich sind.

Ermöglichen Sie eine dauerhafte Ausführung

Um eine dauerhafte Ausführung für Ihre Lambda-Funktion zu ermöglichen, konfigurieren Sie die `DurableConfig` in Ihrer Funktionskonfiguration. Diese Einstellung steuert das Ausführungstimeout, die Beibehaltung des Status und das Versionsverhalten.

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
```

Konfigurationsparameter:

- `ExecutionTimeout`- Maximale Ausführungszeit in Sekunden (bis zu 31.536.000 für ein Jahr)
- `RetentionPeriodInDays`- Wie lange sollen der Ausführungsstatus und der Ausführungsverlauf beibehalten werden (1—365 Tage)
- `AllowInvokeLatest`- Ob der Aufruf der $LATEST-Version für eine dauerhafte Ausführung zulässig sein soll

IAM-Berechtigungen für dauerhafte Funktionen

Dauerhafte Funktionen erfordern zusätzliche IAM-Berechtigungen, die über die standardmäßigen Lambda-Ausführungsrollen hinausgehen. Die Ausführungsrolle Ihrer Funktion muss Berechtigungen für die Statusverwaltung und die dauerhafte Ausführung beinhalten. APIs

Erforderliche Mindestberechtigungen:

```
{
  "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 Beispiel für eine Ausführungsrolle:

```
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: '*'
```

Bewährte Methoden für die Konfiguration

Folgen Sie diesen bewährten Methoden, wenn Sie langlebige Funktionen für den Produktionseinsatz konfigurieren:

- Legen Sie geeignete Ausführungszeitlimits fest — Konfigurieren Sie sie auf der `ExecutionTimeout` Grundlage der maximal zu erwartenden Dauer Ihres Workflows. Legen Sie keine unnötig langen Timeouts fest, da sich diese auf die Kosten und die Ressourcenzuweisung auswirken.
- Sorgen Sie für ein ausgewogenes Verhältnis zwischen Aufbewahrung und Speicherkosten — Legen Sie diese auf der `RetentionPeriodInDays` Grundlage Ihrer Debugging- und Prüfanforderungen fest. Längere Aufbewahrungsfristen erhöhen die Speicherkosten.
- Versionierung in der Produktion verwenden — Wählen Sie `AllowInvokeLatest` diese Option `false` in Produktionsumgebungen aus und verwenden Sie spezifische Funktionsversionen oder Aliase für dauerhafte Ausführungen.
- Statusgröße überwachen — Große Zustandsobjekte erhöhen die Speicherkosten und können die Leistung beeinträchtigen. Halten Sie den Status minimal und verwenden Sie externen Speicher für große Datenmengen.
- Konfigurieren Sie die entsprechende Protokollierung — Aktivieren Sie die detaillierte Protokollierung zur Fehlerbehebung bei lang andauernden Workflows, achten Sie dabei jedoch auf das Protokollvolumen und die Kosten.

Beispiel für eine Produktionskonfiguration:

```
{
  "ExecutionTimeout": 86400,
  "RetentionPeriodInDays": 7,
  "AllowInvokeLatest": false
}
```

Arbeiten mit Lambda-Umgebungsvariablen

Sie können Umgebungsvariablen verwenden, um das Verhalten Ihrer Funktion anzupassen, ohne Code zu aktualisieren. Eine Umgebungsvariable ist ein Zeichenfolgenpaar, das in der versionsspezifischen Konfiguration einer Funktion gespeichert ist. Die Lambda-Laufzeit stellt Umgebungsvariablen für den Code zur Verfügung und legt zusätzliche Umgebungsvariablen mit Informationen über die Funktion und die Aufrufanforderung fest.

> **Note**
>
> Um die Sicherheit zu erhöhen, empfehlen wir, AWS Secrets Manager anstelle von Umgebungsvariablen Datenbankanmeldedaten und andere vertrauliche Informationen wie API-Schlüssel oder Autorisierungstoken zu speichern. Weitere Informationen finden Sie unter Verwenden von Secrets-Manager-Secrets in Lambda-Funktionen.

Umgebungsvariablen werden vor dem Funktionsaufruf nicht ausgewertet. Jeder von Ihnen definierte Wert wird als Literalzeichenfolge betrachtet und nicht erweitert. Führen Sie die Variablenauswertung in Ihrem Funktionscode durch.

Erstellen von Lambda-Umgebungsvariablen

Sie können Umgebungsvariablen in Lambda mithilfe der Lambda-Konsole, der AWS Command Line Interface (AWS CLI), AWS Serverless Application Model (AWS SAM) oder mithilfe eines AWS SDK konfigurieren.

Console

Sie definieren Umgebungsvariablen für die unveröffentlichte Version Ihrer Funktion. Wenn Sie eine Version veröffentlichen, werden die Umgebungsvariablen für diese Version zusammen mit anderen versionsspezifischen Konfigurationseinstellungen gesperrt.

Sie erstellen eine Umgebungsvariable für Ihre Funktion, indem Sie einen Schlüssel und einen Wert definieren. Ihre Funktion verwendet den Namen des Schlüssels, um den Wert der Umgebungsvariablen abzurufen.

So legen Sie Umgebungsvariablen in der Lambda-Konsole fest

1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.

2. Wählen Sie eine Funktion aus.
3. Wählen Sie den Tab Configuration (Konfiguration) und dann Environment variables (Umgebungsvariablen) aus.
4. Wählen Sie unter Environment variables (Umgebungsvariablen) die Option Edit (Bearbeiten).
5. Wählen Sie Umgebungsvariablen hinzufügen aus.
6. Geben Sie einen Schlüssel und einen Wert ein.

   Voraussetzungen

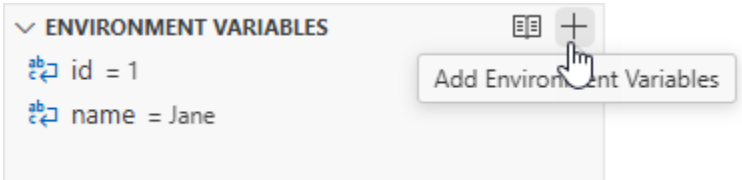
   - Schlüssel beginnen mit einem Buchstaben und bestehen aus mindestens zwei Zeichen.
   - Schlüssel enthalten nur Buchstaben, Zahlen und den Unterstrich (_).
   - Schlüssel sind nicht Lambda vorbehalten.
   - Die Gesamtgröße aller Umgebungsvariablen überschreitet nicht 4 KB.
7. Wählen Sie Speichern.

Generieren einer Liste von Umgebungsvariablen im Konsolen-Code-Editor

Sie können im Lambda-Code-Editor eine Liste von Umgebungsvariablen generieren. So können Sie beim Programmieren schnell auf Ihre Umgebungsvariablen verweisen.

1. Wählen Sie die Registerkarte Code.
2. Scrollen Sie im Code-Editor nach unten zum Abschnitt UMGEBUNGSVARIABLEN. Bestehende Umgebungsvariablen sind hier aufgelistet:

3. Um neue Umgebungsvariablen zu erstellen, wählen Sie das Pluszeichen (+):



Umgebungsvariablen bleiben verschlüsselt, wenn sie im Code-Editor der Konsole aufgeführt werden. Wenn Sie Verschlüsselungshilfen für die Verschlüsselung während der Übertragung aktiviert haben, bleiben diese Einstellungen unverändert. Weitere Informationen finden Sie unter Sicherung von Lambda-Umgebungsvariablen.

Die Liste der Umgebungsvariablen ist schreibgeschützt und nur auf der Lambda-Konsole verfügbar. Diese Datei ist nicht enthalten, wenn Sie das ZIP-Dateiarchiv der Funktion herunterladen, und Sie können keine Umgebungsvariablen hinzufügen, indem Sie diese Datei hochladen.

AWS CLI

Im folgenden Beispiel werden zwei Umgebungsvariablen für eine Funktion mit dem Namen festgeleg `my-function`.

```
aws lambda update-function-configuration \
  --function-name my-function \
  --environment "Variables={BUCKET=amzn-s3-demo-bucket,KEY=file.txt}"
```

Wenn Sie Umgebungsvariablen mit dem `update-function-configuration`-Befehl anwenden, wird der gesamte Inhalt der `Variables`-Struktur ersetzt. Um vorhandene Umgebungsvariablen beibehalten, wenn Sie eine neue hinzufügen, schließen Sie alle vorhandenen Werte in Ihre Anforderung ein.

Mit dem `get-function-configuration`-Befehl können Sie die aktuelle Konfiguration abrufen.

```
aws lambda get-function-configuration \
  --function-name my-function
```

Die Ausgabe sollte folgendermaßen aussehen:

```
{
    "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",
    ...
}
```

Sie können die Revisions-ID aus der Ausgabe von `get-function-configuration` als Parameter an `update-function-configuration` übergeben. Dadurch wird sichergestellt, dass sich die Werte zwischen dem Lesen und dem Aktualisieren der Konfiguration nicht ändern.

Um den Verschlüsselungsschlüssel einer Funktion zu konfigurieren, legen Sie die `KMSKeyARN`-Option fest.

```
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

Sie können das AWS Serverless Application Model verwenden, um Umgebungsvariablen für Ihre Funktion zu konfigurieren. Aktualisieren Sie die Eigenschaften Umgebung und Variablen in Ihrer `template.yaml`-Datei und führen Sie dann sam deploy aus.

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

Verwenden Sie die folgenden API-Operationen, um Umgebungsvariablen mithilfe eines AWS SDK zu verwalten.

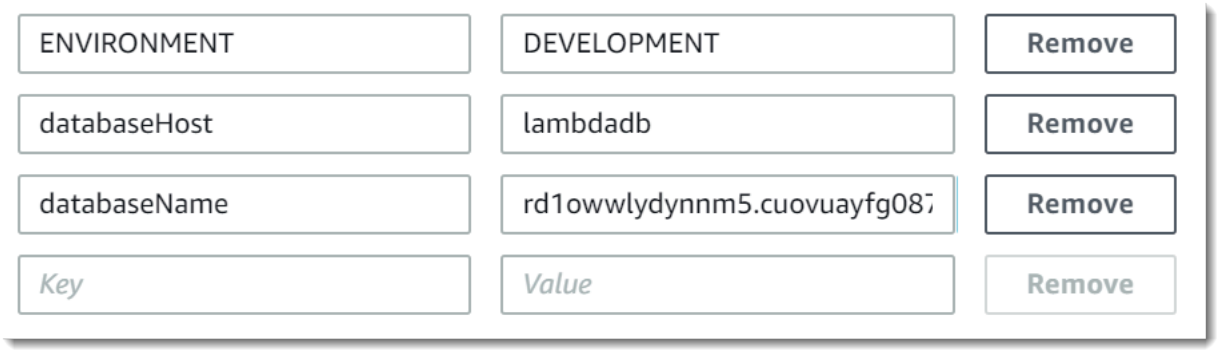
- UpdateFunctionConfiguration
- GetFunctionConfiguration
- CreateFunction

Weitere Informationen finden Sie in der AWS -SDK-Dokumentation für Ihre bevorzugte Programmiersprache.

Beispielszenario für Umgebungsvariablen

Sie können Umgebungsvariablen verwenden, um das Funktionsverhalten in Ihrer Testumgebung und Produktionsumgebung anzupassen. Zum Beispiel können Sie zwei Funktionen mit demselben Code aber mit unterschiedlichen Konfigurationen erstellen. Die eine Funktion stellt eine Verbindung mit einer Testdatenbank und die andere mit einer Produktionsdatenbank her. In diesem Fall verwenden Sie Umgebungsvariablen, um den Hostnamen und andere Verbindungsdetails für die Datenbank an die Funktion zu übergeben.

Das folgende Beispiel zeigt, wie der Datenbankhost und der Datenbankname als Umgebungsvariablen definiert werden.



Wenn Ihre Testumgebung mehr Debugging-Informationen generieren soll als die Produktionsumgebung, können Sie eine Umgebungsvariable festlegen, um Ihre Testumgebung so zu konfigurieren, dass eine ausführlichere Protokollierung oder eine detailliertere Ablaufverfolgung verwendet wird.

Beispielsweise könnten Sie in Ihrer Testumgebung eine Umgebungsvariable mit dem Schlüssel `LOG_LEVEL` und einem Wert festlegen, der die Protokollstufe „Debug“ oder „Trace“ angibt. Im Code Ihrer Lambda-Funktion können Sie dann diese Umgebungsvariable verwenden, um die Protokollstufe festzulegen.

Die folgenden Codebeispiele in Python und Node.js veranschaulichen, wie Sie dies erreichen können. In diesen Beispielen wird davon ausgegangen, dass Ihre Umgebungsvariable den Wert `DEBUG` in Python oder `debug` in Node.js hat.

Python

Example Python-Code zum Festlegen der Protokollstufe

```
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 Node.js-Code zum Festlegen der Protokollstufe

In diesem Beispiel wird die `winston`-Protokollierungsbibliothek verwendet. Verwenden Sie npm, um diese Bibliothek dem Bereitstellungspaket Ihrer Funktion hinzuzufügen. Weitere Informationen finden Sie unter the section called “ZIP-Bereitstellungspakets mit Abhängigkeiten erstellen”.

```
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');

};
```

Abrufen von Lambda-Umgebungsvariablen

Umgebungsvariablen in Ihrem Funktionscode können Sie mit der Standardmethode für Ihre Programmiersprache abrufen.

Node.js

```
let region = process.env.AWS_REGION
```

Python

```
import os
  region = os.environ['AWS_REGION']
```

> **Note**
>
> In einigen Fällen müssen Sie möglicherweise das folgende Format verwenden:
>
> ```
> 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 speichert Umgebungsvariablen sicher, indem sie im Ruhezustand verschlüsselt werden. Sie können Lambda so konfigurieren, dass es einen anderen Verschlüsselungsschlüssel verwendet, Umgebungsvariablenwerte auf der Clientseite verschlüsselt oder Umgebungsvariablen in einer CloudFormation Vorlage mit festlegen. AWS Secrets Manager

Definierte Laufzeitumgebungsvariablen

Lambda Laufzeiten setzen mehrere Umgebungsvariablen während der Initialisierung. Die meisten Umgebungsvariablen stellen Informationen über die Funktion oder Laufzeit bereit. Die Schlüssel für

diese Umgebungsvariablen sind reserviert und können nicht in der Funktionskonfiguration gesetzt werden.

Reservierte Umgebungsvariablen

- `_HANDLER` – Der für die Funktion konfigurierte Handler-Speicherort.
- `_X_AMZN_TRACE_ID` – Die X-Ray-Tracing-Header. Diese Umgebungsvariable ändert sich bei jedem Aufruf.
  - Diese Umgebungsvariable ist nicht für reine OS-Laufzeiten (die `provided`-Laufzeitfamilie) definiert. Sie können `_X_AMZN_TRACE_ID` für benutzerdefinierte Laufzeiten einstellen, indem Sie den Antwort-Header `Lambda-Runtime-Trace-Id` der Nächster Aufruf verwenden.
  - Für die Java-Laufzeit-Versionen 17 und höher wird diese Umgebungsvariable nicht verwendet. Stattdessen speichert Lambda Ablaufverfolgungsinformationen in der Systemeigenschaft `com.amazonaws.xray.traceHeader`.
- `AWS_DEFAULT_REGION`— Die Standardeinstellung AWS-Region , in der die Lambda-Funktion ausgeführt wird.
- `AWS_REGION`— Der AWS-Region Ort, an dem die Lambda-Funktion ausgeführt wird. Wenn eingegeben, überschreibt dieser Wert die `AWS_DEFAULT_REGION`.
  - Weitere Informationen zur Verwendung der AWS-Region Umgebungsvariablen mit AWS SDKs finden Sie unter AWS Region im Referenzhandbuch für Tools AWS SDKs und Tools.
- `AWS_EXECUTION_ENV` – Die Laufzeit-Kennung mit dem Präfix `AWS_Lambda_` z. B. `AWS_Lambda_java8`. Diese Umgebungsvariable ist nicht für reine OS-Laufzeiten (die `provided`-Laufzeitfamilie) definiert.
- `AWS_LAMBDA_FUNCTION_NAME` – Der Name der Funktion.
- `AWS_LAMBDA_FUNCTION_MEMORY_SIZE` – Die Menge des für die Funktion verfügbaren Speichers in MB.
- `AWS_LAMBDA_FUNCTION_VERSION` – Die Version der Funktion, die gerade ausgeführt wird.
- `AWS_LAMBDA_INITIALIZATION_TYPE`— Der Initialisierungstyp der Funktion, `on-demand` `provisioned-concurrency`oder`snap-start`, oder`lambda-managed-instances`. Weitere Informationen finden Sie unter Konfiguration der bereitgestellten Parallelität,, Verbesserung der Startleistung mit Lambda SnapStart oder. Von Lambda verwaltete Instanzen
- `AWS_LAMBDA_LOG_GROUP_NAME`, `AWS_LAMBDA_LOG_STREAM_NAME` — Der Name der Amazon CloudWatch Logs-Gruppe und des Streams für die Funktion. Die `AWS_LAMBDA_LOG_STREAM_NAME` Umgebungsvariablen `AWS_LAMBDA_LOG_GROUP_NAME` und sind in SnapStart Lambda-Funktionen nicht verfügbar.

- `AWS_ACCESS_KEY`, `AWS_ACCESS_KEY_ID`, `AWS_SECRET_ACCESS_KEY`, `AWS_SESSION_TOKEN` – Die Zugriffsschlüssel, die über die Ausführungsrolle der Funktion bezogen wurden.
- `AWS_LAMBDA_RUNTIME_API` – (Benutzerdefinierte Laufzeit) Host und Port der Laufzeit-API.
- `LAMBDA_TASK_ROOT` – Der Pfad zu Ihrem Lambda-Funktionscode.
- `LAMBDA_RUNTIME_DIR` – Der Pfad zu Laufzeitbibliotheken.
- `AWS_LAMBDA_MAX_CONCURRENCY`— (Nur Lambda Managed Instances) Die maximale Anzahl gleichzeitiger Aufrufe, die Lambda an eine Ausführungsumgebung sendet.

Die folgenden zusätzlichen Umgebungsvariablen sind nicht vorbehalten und können in Ihrer Funktionskonfiguration erweitert werden.

Nicht reservierte Umgebungsvariablen

- `LANG` – Das Gebietsschema der Laufzeit (`en_US.UTF-8`).
- `PATH` – Der Ausführungspfad (`/usr/local/bin:/usr/bin/:/bin:/opt/bin`).
- `LD_LIBRARY_PATH` – Der Pfad der Systembibliothek (`/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) Der Node.js-Bibliothekspfad (`/opt/nodejs/node12/node_modules/:/opt/nodejs/node_modules:$LAMBDA_RUNTIME_DIR/node_modules`).
- `NODE_OPTIONS` – (Node.js) Für Node.js-Laufzeiten können Sie `NODE_OPTIONS` verwenden, um experimentelle Features wieder zu aktivieren, die Lambda standardmäßig deaktiviert.
- `PYTHONPATH` – (Python) Der Python-Bibliothekspfad (`$LAMBDA_RUNTIME_DIR`).
- `GEM_PATH` – (Ruby) Der Pfad der Ruby-Bibliothek (`$LAMBDA_TASK_ROOT/vendor/bundle/ruby/3.3.0:/opt/ruby/gems/3.3.0`).
- `AWS_XRAY_CONTEXT_MISSING` – Für die X-Ray-Ablaufverfolgung legt Lambda dies auf `LOG_ERROR` fest, um Laufzeitfehler aus dem X-Ray-SDK zu vermeiden.
- `AWS_XRAY_DAEMON_ADDRESS` – Die IP-Adresse und der Port des X-Ray-Daemons.
- `AWS_LAMBDA_DOTNET_PREJIT` – (.NET) Legen Sie für die .NET-Laufzeitumgebung diese Variable so fest, dass sie .NET-spezifische Laufzeitoptimierungen aktiviert oder deaktiviert. Zu den Werten gehören `always`, `never` und `provisioned-concurrency`. Weitere Informationen finden Sie unter Konfigurieren von Provisioned Concurrency für eine Funktion.
- `TZ` – Die Zeitzone der Umgebung (`:UTC`). Die Ausführungsumgebung verwendet NTP, um die Systemuhr zu synchronisieren.

Die angezeigten Beispielwerte spiegeln die neuesten Laufzeiten wider. Das Vorhandensein bestimmter Variablen oder deren Werte kann je nach früheren Laufzeiten unterschiedlich sein.

Sicherung von Lambda-Umgebungsvariablen

Sie können Ihre Umgebungsvariablen sichern, können Sie eine serverseitige Verschlüsselung nutzen, um Ihre Data-at-Rest zu schützen und eine clientseitige Verschlüsselung, um Ihre Daten während der Übertragung zu schützen.

> **Note**
>
> Um die Datenbanksicherheit zu erhöhen, empfehlen wir, AWS Secrets Manager anstelle von Umgebungsvariablen zum Speichern von Datenbankanmeldedaten zu verwenden. Weitere Informationen finden Sie unter Verwenden von Secrets-Manager-Secrets in Lambda-Funktionen.

Sicherheit im Ruhezustand

Lambda bietet im Ruhezustand immer serverseitiger Verschlüsselung mit einem AWS KMS key an. Standardmäßig verwendet Lambda einen Von AWS verwalteter Schlüssel. Wenn dieses Standardverhalten Ihrem Workflow entspricht, müssen Sie nichts anderes einrichten. Lambda erstellt das Von AWS verwalteter Schlüssel in Ihrem Konto und verwaltet die Berechtigungen für Sie. AWS berechnet Ihnen keine Gebühren für die Verwendung dieses Schlüssels.

Wenn Sie möchten, können Sie stattdessen einen vom AWS KMS Kunden verwalteten Schlüssel bereitstellen. Sie können dies tun, um die Drehung des KMS-Schlüssels zu steuern oder die Anforderungen Ihrer Organisation für die Verwaltung von KMS-Schlüsseln zu erfüllen. Wenn Sie einen vom Kunden verwalteten Schlüssel verwenden, können nur Benutzer in Ihrem Konto mit Zugriff auf den KMS-Schlüssel Umgebungsvariablen für die Funktion anzeigen oder verwalten.

Für vom Kunden verwaltete Schlüssel fallen AWS KMS Standardgebühren an. Weitere Informationen finden Sie unter AWS Key Management Service Preise.

Sicherheit während der Übertragung

Für zusätzliche Sicherheit können Sie Helfer für die Verschlüsselung bei der Übertragung aktivieren, wodurch sichergestellt wird, dass Ihre Umgebungsvariablen clientseitig verschlüsselt sind, um sie während der Übertragung zu schützen.

So konfigurieren Sie die Verschlüsselung für Ihre Umgebungsvariablen

1. Verwenden Sie AWS Key Management Service (AWS KMS), um alle vom Kunden verwalteten Schlüssel für Lambda zu erstellen, die für die serverseitige und clientseitige Verschlüsselung verwendet werden sollen. Weitere Informationen finden Sie unter Erstellen von Schlüsseln im AWS Key Management Service -Entwicklerhandbuch.
2. Navigieren Sie mithilfe der Lambda-Konsole zur Seite Umgebungsvariablen bearbeiten.

   a. Öffnen Sie die Seite Funktionen der Lambda-Konsole.
   b. Wählen Sie eine Funktion aus.
   c. Wählen Sie Konfiguration und dann Umgebungsvariablen aus der linken Navigationsleiste.
   d. Wählen Sie im Abschnitt Umgebungsvariablen Bearbeiten aus.
   e. Erweitern Sie Encryption configuration (Verschlüsselungskonfiguration).
3. (Optional) Aktivieren Sie Konsolenverschlüsselungshelfer, um die clientseitige Verschlüsselung zu verwenden, um Ihre Daten während der Übertragung zu schützen.

   a. Wählen Sie unter Verschlüsselung während der Übertragung die Option Helfer für die Verschlüsselung während der Übertragung aktivieren aus.
   b. Wählen Sie für jede Umgebungsvariable, für die Sie Konsolenverschlüsselungshilfen aktivieren möchten, neben der Umgebungsvariablen die Option Encrypt (Verschlüsseln) aus.
   c. Wählen AWS KMS key Sie unter Verschlüsselung bei der Übertragung einen vom Kunden verwalteten Schlüssel aus, den Sie zu Beginn dieses Verfahrens erstellt haben.
   d. Wählen Sie Ausführungsrollenrichtlinie und kopieren Sie die Richtlinie. Diese Richtlinie gewährt der Ausführungsrolle Ihrer Funktion die Berechtigung zum Entschlüsseln der Umgebungsvariablen.

      Speichern Sie diese Richtlinie zur Verwendung im letzten Schritt dieses Verfahrens.
   e. Fügen Sie Ihrer Funktion Code hinzu, der die Umgebungsvariablen entschlüsselt. Wählen Sie Geheimnisausschnitt entschlüsseln, um ein Beispiel anzuzeigen.
4. (Optional) Geben Sie Ihren kundenverwalteten Schlüssel zur Verschlüsselung im Ruhezustand an.

   a. Wählen Sie Kundenmasterschlüssel verwenden aus.
   b. Wählen Sie einen vom Kunden verwalteten Schlüssel aus, den Sie zu Beginn dieses Verfahrens erstellt haben.
5. Wählen Sie Speichern.

6. Richten Sie Berechtigungen ein.

   Wenn Sie einen vom Kunden verwalteten Schlüssel mit serverseitiger Verschlüsselung verwenden, erteilen Sie allen Benutzern oder Rollen Berechtigungen zum Anzeigen oder Verwalten von Umgebungsvariablen für die Funktion. Weitere Informationen finden Sie unter Verwalten von Berechtigungen für den serverseitigen Verschlüsselungs-KMS-Schlüssel.

   Wenn Sie die clientseitige Verschlüsselung für die Sicherheit während der Übertragung aktivieren, benötigt Ihre Funktion die Berechtigung zum Aufrufen der `kms:Decrypt`-API-Operation. Fügen Sie die Richtlinie, die Sie zuvor in diesem Verfahren gespeichert haben, zur Ausführungsrolle der Funktion hinzu.

Verwalten von Berechtigungen für den serverseitigen Verschlüsselungs-KMS-Schlüssel

Für Ihren Benutzer oder die Ausführungsrolle der Funktion sind keine AWS KMS Berechtigungen erforderlich, um den Standard-Verschlüsselungsschlüssel zu verwenden. Um einen vom kundenverwalteten Schlüssel verwenden zu können, benötigen Sie eine Berechtigungen zur Verwendung des Schlüssels. Lambda erstellt anhand Ihrer Berechtigungen eine Berechtigungserteilung für den Schlüssel. So wird es Lambda ermöglicht, diesen für die Verschlüsselung zu verwenden.

- `kms:ListAliases` – Zur Ansicht von Schlüssel in der Lambda-Konsole.
- `kms:CreateGrant`, `kms:Encrypt` – Zum Konfigurieren eines vom Kunden verwalteten Schlüssels für eine Funktion.
- `kms:Decrypt` – Zum Anzeigen und Verwalten von Umgebungsvariablen, die mit einem vom Kunden verwalteten Schlüssel verschlüsselt sind.

Sie können diese Berechtigungen über Ihre AWS-Konto oder über die ressourcenbasierte Berechtigungsrichtlinie eines Schlüssels erhalten. `ListAliases`wird durch die verwalteten Richtlinien für Lambda bereitgestellt. Schlüsselrichtlinien gewähren Benutzern in der Gruppe Key users (Schlüsselbenutzer) die verbleibenden Berechtigungen.

Benutzer ohne `Decrypt`-Berechtigungen können Funktionen weiterhin verwalten, aber sie können keine Umgebungsvariablen anzeigen oder in der Lambda-Konsole verwalten. Um zu verhindern, dass ein Benutzer Umgebungsvariablen anzeigen kann, fügen Sie den Berechtigungen des

Benutzers eine Anweisung hinzu, die den Zugriff auf den Standardschlüssel, einen vom Kunden verwalteten Schlüssel oder alle Schlüssel verweigert.

Example IAM-Richtlinie – Verweigern des Zugriffs nach Schlüssel-ARN

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"
        }
    ]
}
```

Details zum Verwalten von Schlüsselberechtigungen finden Sie unter Key policies in AWS KMS im AWS Key Management Service -Entwicklerhandbuch.

Lambda-Funktionen Zugriff auf Ressourcen in einer Amazon VPC gewähren

Mit Amazon Virtual Private Cloud (Amazon VPC) können Sie private Netzwerke in Ihren AWS-Konto Host-Ressourcen wie Amazon Elastic Compute Cloud (Amazon EC2) -Instances, Amazon Relational Database Service (Amazon RDS) -Instances und ElastiCache Amazon-Instances erstellen. Sie können Ihrer Lambda-Funktion Zugriff auf Ressourcen geben, die in einer Amazon VPC gehostet werden, indem Sie Ihre Funktion über die privaten Subnetze, die die Ressourcen enthalten, mit der VPC verbinden. Folgen Sie den Anweisungen in den folgenden Abschnitten, um eine Lambda-Funktion über die Lambda-Konsole, die AWS Command Line Interface (AWS CLI) oder an eine Amazon VPC anzuhängen. AWS SAM

> **Note**
>
> Jede Lambda-Funktion läuft in einer VPC, die dem Lambda-Dienst gehört und von diesem verwaltet wird. Diese VPCs werden automatisch von Lambda verwaltet und sind für Kunden nicht sichtbar. Die Konfiguration Ihrer Funktion für den Zugriff auf andere AWS Ressourcen in einer Amazon VPC hat keine Auswirkungen auf die von Lambda verwaltete VPC, in der Ihre Funktion ausgeführt wird.

Sections

- Erforderliche IAM-Berechtigungen
- Hinzufügen von Lambda-Funktionen zu einer Amazon VPC in Ihrem AWS-Konto
- Internetzugang bei Verbindung mit einer VPC
- IPv6 Unterstützung
- Bewährte Methoden für die Verwendung von Lambda mit Amazon VPCs
- Grundlegendes zu elastischen Hyperplane-Netzwerkschnittstellen () ENIs
- Verwenden von IAM-Bedingungsschlüsseln für VPC-Einstellungen
- VPC-Tutorials

Erforderliche IAM-Berechtigungen

Um eine Lambda-Funktion an eine Amazon-VPC in Ihrer anzuhängen AWS-Konto, benötigt Lambda Berechtigungen zum Erstellen und Verwalten der Netzwerkschnittstellen, die es verwendet, um Ihrer Funktion Zugriff auf die Ressourcen in der VPC zu gewähren.

Die von Lambda erstellten Netzwerkschnittstellen werden als Hyperplane Elastic Network Interfaces oder Hyperplane bezeichnet. ENIs Weitere Informationen zu diesen Netzwerkschnittstellen finden Sie unter the section called “Grundlegendes zu elastischen Hyperplane-Netzwerkschnittstellen () ENIs”.

Sie können Ihrer Funktion die erforderlichen Berechtigungen erteilen, indem Sie die AWS verwaltete Richtlinie AWSLambdaVPCAccessExecutionRolean die Ausführungsrolle Ihrer Funktion anhängen. Wenn Sie eine neue Funktion in der Lambda-Konsole erstellen und sie an eine VPC anhängen, fügt Lambda diese Berechtigungsrichtlinie automatisch für Sie hinzu.

Wenn Sie es vorziehen, Ihre eigene IAM-Berechtigungsrichtlinie zu erstellen, stellen Sie sicher, dass Sie alle der folgenden Berechtigungen hinzufügen und sie auf allen Ressourcen erlauben (`"Resource": "*"`):

- ec2: CreateNetworkInterface
- ec2: DescribeNetworkInterfaces
- ec2: DescribeSubnets
- ec2: DeleteNetworkInterface
- ec2: AssignPrivateIpAddresses
- ec2: UnassignPrivateIpAddresses

Beachten Sie, dass die Rolle Ihrer Funktion diese Berechtigungen nur benötigt, um die Netzwerkschnittstellen zu erstellen, nicht aber, um Ihre Funktion aufzurufen. Sie können Ihre Funktion weiterhin erfolgreich aufrufen, wenn sie an eine Amazon VPC angefügt ist, selbst wenn Sie diese Berechtigungen aus der Ausführungsrolle Ihrer Funktion entfernen.

Um Ihre Funktion an eine VPC anzuhängen, muss Lambda auch Netzwerkressourcen mithilfe Ihrer IAM-Benutzerrolle verifizieren. Stellen Sie sicher, dass Ihre Benutzerrolle über die folgenden IAM-Berechtigungen verfügt:

- ec2: DescribeSecurityGroups
- ec2: DescribeSubnets

- ec2: DescribeVpcs
- ec2: GetSecurityGroupsForVpc

> **Note**
>
> Die EC2 Amazon-Berechtigungen, die Sie der Ausführungsrolle Ihrer Funktion gewähren, werden vom Lambda-Service verwendet, um Ihre Funktion an eine VPC anzuhängen. Sie gewähren diese Berechtigungen jedoch auch implizit für den Code Ihrer Funktion. Das bedeutet, dass Ihr Funktionscode diese EC2 Amazon-API-Aufrufe ausführen kann. Ratschläge zur Einhaltung bewährter Sicherheitsverfahren finden Sie unter the section called “Bewährte Methoden für die Gewährleistung der Sicherheit”.

Hinzufügen von Lambda-Funktionen zu einer Amazon VPC in Ihrem AWS-Konto

Verbinden Sie Ihre Funktion mit einer Amazon VPC in Ihrem, AWS-Konto indem Sie die Lambda-Konsole verwenden, oder. AWS CLI AWS SAM Wenn Sie das AWS CLI oder verwenden oder AWS SAM eine bestehende Funktion mithilfe der Lambda-Konsole an eine VPC anhängen, stellen Sie sicher, dass die Ausführungsrolle Ihrer Funktion über die erforderlichen Berechtigungen verfügt, die im vorherigen Abschnitt aufgeführt sind.

Lambda-Funktionen können keine direkte Verbindung mit einer VPC mit Dedicated-Instance-Tenancy herstellen. Zum Herstellen einer Verbindung mit Ressourcen in einer dedizierten VPC verbinden Sie sie mit einer zweiten VPC mit Standard-Tenancy.

Lambda console

So hängen Sie eine Funktion an eine Amazon VPC an, wenn Sie sie erstellen

1. Öffnen Sie die Seite Functions (Funktionen) der Lambda-Konsole und wählen Sie Create function (Funktion erstellen) aus.
2. Geben Sie unter Basic information (Grundlegende Informationen) bei Function name (Funktionsname) einen Namen für Ihre Funktion ein.
3. Konfigurieren Sie die VPC-Einstellungen für die Funktion, indem Sie wie folgt vorgehen:

    a. Erweiterten Sie Advanced settings (Erweiterte Einstellungen).

b. Wählen Sie VPC aktivieren und wählen Sie dann den VPC, dem Sie die Funktion zuordnen möchten.

c. (Optional) Um ausgehenden IPv6 Datenverkehr zuzulassen, wählen Sie Traffic für Dual-Stack-Subnetze zulassen IPv6 aus.

d. Wählen Sie die Teilnetze und Sicherheitsgruppen aus, für die die Netzwerkschnittstelle erstellt werden soll. Wenn Sie IPv6 Datenverkehr für Dual-Stack-Subnetze zulassen ausgewählt haben, müssen alle ausgewählten Subnetze über einen CIDR-Block und einen IPv4 CIDR-Block verfügen. IPv6

> **Note**
>
> Verbinden Sie Ihre Funktion für den Zugriff auf private Ressourcen mit privaten Subnetzen. Wenn Ihre Funktion Internetzugang benötigt, siehe the section called "Internetzugriff für VPC-Funktionen". Durch die Verbindung einer Funktion mit einem öffentlichen Subnetz erhält sie weder Internetzugang noch eine öffentliche IP-Adresse.

4. Wählen Sie Create function (Funktion erstellen).

So hängen Sie eine vorhandene Funktion an eine Amazon VPC an

1. Öffnen Sie die Funktionsseite der Lambda-Konsole und wählen Sie Ihre Funktion aus.
2. Wählen Sie die Registerkarte Konfiguration und anschließend VPC aus.
3. Wählen Sie Bearbeiten aus.
4. Wählen Sie unter VPC die Amazon VPC aus, an die Sie Ihre Funktion anhängen möchten.
5. (Optional) Um ausgehenden IPv6 Datenverkehr zuzulassen, wählen Sie Datenverkehr für Dual-Stack-Subnetze zulassen aus. IPv6
6. Wählen Sie die Teilnetze und Sicherheitsgruppen aus, für die die Netzwerkschnittstelle erstellt werden soll. Wenn Sie IPv6 Datenverkehr für Dual-Stack-Subnetze zulassen ausgewählt haben, müssen alle ausgewählten Subnetze über einen CIDR-Block und einen IPv4 CIDR-Block verfügen. IPv6

> **Note**
>
> Verbinden Sie Ihre Funktion für den Zugriff auf private Ressourcen mit privaten Subnetzen. Wenn Ihre Funktion Internetzugang benötigt, siehe the section called

"Internetzugriff für VPC-Funktionen". Durch die Verbindung einer Funktion mit einem öffentlichen Subnetz erhält sie weder Internetzugang noch eine öffentliche IP-Adresse.

7. Wählen Sie Speichern.

AWS CLI

So hängen Sie eine Funktion an eine Amazon VPC an, wenn Sie sie erstellen

- Um eine Lambda-Funktion zu erstellen und sie mit einer VPC zu verbinden, führen Sie den folgenden CLI `create-function`-Befehl aus.

```
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
```

Geben Sie Ihre eigenen Subnetze und Sicherheitsgruppen an und setzen Sie `Ipv6AllowedForDualStack` je nach Anwendungsfall auf `true` oder `false`.

So hängen Sie eine vorhandene Funktion an eine Amazon VPC an

- Um eine vorhandene Funktion an eine VPC anzuhängen, führen Sie den folgenden CLI `update-function-configuration`-Befehl aus.

```
aws lambda update-function-configuration --function-name my-function \
--vpc-config Ipv6AllowedForDualStack=true,
 SubnetIds=subnet-071f712345678e7c8,subnet-07fd123456788a036,SecurityGroupIds=sg-085912
```

So trennen Sie Ihre Funktion von einer VPC

- Um Ihre Funktion von einer VPC zu trennen, führen Sie den folgenden `update-function-configuration`-CLI-Befehl mit einer leeren Liste der VPC-Subnetze und Sicherheitsgruppen aus.

```
aws lambda update-function-configuration --function-name my-function \
```

```
--vpc-config SubnetIds=[],SecurityGroupIds=[]
```

AWS SAM

So verbinden Sie Ihre Funktion mit einer VPC

- Um eine Lambda-Funktion an eine Amazon VPC anzuhängen, fügen Sie die `VpcConfig`-Eigenschaft zu Ihrer Funktionsdefinition hinzu, wie in der folgenden Beispielvorlage gezeigt. Weitere Informationen zu dieser Eigenschaft finden Sie unter AWS: :Lambda: :Function VpcConfig im CloudFormation Benutzerhandbuch (die AWS SAM `VpcConfig` Eigenschaft wird direkt an die `VpcConfig` Eigenschaft einer CloudFormation `AWS::Lambda::Function` Ressource übergeben).

```
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
```

Weitere Informationen zur Konfiguration Ihrer VPC in AWS SAM finden Sie unter AWS::EC2: :VPC im CloudFormation Benutzerhandbuch.

Internetzugang bei Verbindung mit einer VPC

Standardmäßig haben Lambda-Funktionen Zugriff auf das öffentliche Internet. Wenn Sie Ihre Funktion an eine VPC anhängen, kann sie nur auf Ressourcen zugreifen, die in dieser VPC verfügbar sind. Um Ihrer Funktion Zugriff auf das Internet zu gewähren, müssen Sie die VPC auch für den Internetzugang konfigurieren. Weitere Informationen hierzu finden Sie unter the section called “Internetzugriff für VPC-Funktionen”.

IPv6 Unterstützung

Ihre Funktion kann über eine Verbindung zu Ressourcen in Dual-Stack-VPC-Subnetzen eine Verbindung herstellen. IPv6 Diese Option ist standardmäßig deaktiviert. Um ausgehenden IPv6 Datenverkehr zuzulassen, verwenden Sie die Konsole oder die `--vpc-config Ipv6AllowedForDualStack=true` Option mit der Create-Funktion oder dem Befehl. update-function-configuration

> **Note**
>
> Um ausgehenden IPv6 Verkehr in einer VPC zuzulassen, müssen alle Subnetze, die mit der Funktion verbunden sind, Dual-Stack-Subnetze sein. Lambda unterstützt keine ausgehenden IPv6 Verbindungen für IPv6 reine Subnetze in einer VPC oder ausgehende IPv6 Verbindungen für Funktionen, die nicht mit einer VPC verbunden sind.

Sie können Ihren Funktionscode aktualisieren, um explizit eine Verbindung zu Subnetzressourcen herzustellen. IPv6 Das folgende Python-Beispiel öffnet einen Socket und stellt eine Verbindung zu einem IPv6 Server her.

Example — Connect zum IPv6 Server herstellen

```
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()
```

Bewährte Methoden für die Verwendung von Lambda mit Amazon VPCs

Um sicherzustellen, dass Ihre Lambda-VPC-Konfiguration den Best-Practice-Richtlinien entspricht, befolgen Sie die Hinweise in den folgenden Abschnitten.

Bewährte Methoden für die Gewährleistung der Sicherheit

Um Ihre Lambda-Funktion an eine VPC anzuhängen, müssen Sie der Ausführungsrolle Ihrer Funktion eine Reihe von EC2 Amazon-Berechtigungen erteilen. Diese Berechtigungen sind erforderlich, um die Netzwerkschnittstellen zu erstellen, die Ihre Funktion für den Zugriff auf die Ressourcen in der VPC verwendet. Diese Berechtigungen werden jedoch auch implizit für den Code Ihrer Funktion

gewährt. Das bedeutet, dass Ihr Funktionscode berechtigt ist, diese EC2 Amazon-API-Aufrufe zu tätigen.

Um dem Prinzip des Zugriffs mit den geringsten Rechten zu folgen, fügen Sie der Ausführungsrolle Ihrer Funktion eine Verweigerungsrichtlinie wie das folgende Beispiel hinzu. Diese Richtlinie verhindert, dass Ihr Funktionscode Amazon aufruft EC2 APIs, ermöglicht es dem Lambda-Service dennoch, VPC-Ressourcen in Ihrem Namen zu verwalten. Die Richtlinie verwendet den `lambda:SourceFunctionArn`-Bedingungsschlüssel, der nur für API-Aufrufe gilt, die während der Ausführung von Ihrem Funktionscode getätigt werden. Weitere Informationen finden Sie unter the section called “ARN der Quellfunktion”.

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 stellt Sicherheitsgruppen und Netzwerk-Zugriffskontrolllisten (ACLs) bereit, um die Sicherheit in Ihrer VPC zu erhöhen. Sicherheitsgruppen kontrollieren den ein- und ausgehenden Datenverkehr für Ihre Ressourcen, und das Netzwerk ACLs steuert den ein- und ausgehenden Datenverkehr für Ihre Subnetze. Sicherheitsgruppen bieten ausreichend Zugriffskontrolle für die meisten Subnetze. Sie können das Netzwerk verwenden ACLs , wenn Sie eine zusätzliche Sicherheitsebene für Ihre VPC wünschen. Allgemeine Richtlinien zu bewährten Sicherheitsmethoden bei der Nutzung von Amazon VPCs finden Sie unter Bewährte Sicherheitsmethoden für Ihre VPC im Amazon Virtual Private Cloud Cloud-Benutzerhandbuch.

Bewährte Methoden zur Leistungssteigerung

Wenn Sie Ihre Funktion an eine VPC anhängen, prüft Lambda, ob es eine verfügbare Netzwerkressource (Hyperplane ENI) gibt, mit der es eine Verbindung herstellen kann. Hyperplane ENIs sind einer bestimmten Kombination von Sicherheitsgruppen und VPC-Subnetzen zugeordnet. Wenn Sie bereits eine Funktion an eine VPC angeschlossen haben, bedeutet die Angabe derselben Subnetze und Sicherheitsgruppen beim Anschließen einer weiteren Funktion, dass Lambda die Netzwerkressourcen gemeinsam nutzen kann und keine neue Hyperplane ENI erstellt werden muss. Weitere Informationen zu Hyperplane ENIs und ihrem Lebenszyklus finden Sie unter. the section called “Grundlegendes zu elastischen Hyperplane-Netzwerkschnittstellen () ENIs”

Grundlegendes zu elastischen Hyperplane-Netzwerkschnittstellen () ENIs

Eine Hyperplane ENI ist eine verwaltete Ressource, die als Netzwerkschnittstelle zwischen Ihrer Lambda-Funktion und den Ressourcen fungiert, mit denen Sie Ihre Funktion verbinden möchten. Der Lambda-Service erstellt und verwaltet diese ENIs automatisch, wenn Sie Ihre Funktion an eine VPC anhängen.

Hyperplane ENIs sind für Sie nicht direkt sichtbar, und Sie müssen sie nicht konfigurieren oder verwalten. Wenn Sie jedoch wissen, wie sie funktionieren, können Sie besser verstehen, wie sich Ihre Funktion verhält, wenn Sie sie an eine VPC anhängen.

Wenn Sie zum ersten Mal eine Funktion mithilfe einer bestimmten Kombination aus Subnetz und Sicherheitsgruppe an eine VPC anfügen, erstellt Lambda eine Hyperplane-ENI. Andere Funktionen in Ihrem Konto, die dieselbe Kombination aus Subnetz und Sicherheitsgruppe verwenden, können diese ENI ebenfalls verwenden. Wo immer möglich, verwendet Lambda Bestehendes, ENIs um die Ressourcennutzung zu optimieren und die Entstehung neuer Ressourcen zu minimieren. ENIs Jede Hyperplane ENI unterstützt bis zu 65.000 Verbindungen/Ports. Wenn die Anzahl der Verbindungen diesen Grenzwert überschreitet, skaliert Lambda die Anzahl der Verbindungen ENIs automatisch auf der Grundlage des Netzwerkverkehrs und der Parallelitätsanforderungen.

Bei neuen Funktionen bleibt Ihre Funktion, während Lambda eine Hyperplane-ENI erstellt, im Status Ausstehend und Sie können sie nicht aufrufen. Ihre Funktion wechselt erst in den Status Aktiv, wenn die Hyperplane ENI bereit ist, was mehrere Minuten dauern kann. Bei vorhandenen Funktionen können Sie keine zusätzlichen Operationen durchführen, die auf die Funktion abzielen, wie z. B. das Erstellen von Versionen oder das Aktualisieren des Funktionscodes, aber Sie können weiterhin frühere Versionen der Funktion aufrufen.

Im Rahmen der Verwaltung des ENI-Lebenszyklus kann Lambda Daten löschen und neu erstellen, um den Netzwerkverkehr ENIs zu verteilen ENIs oder um Probleme zu beheben, die bei ENI-Integritätsprüfungen festgestellt wurden. Wenn eine Lambda-Funktion 14 Tage lang inaktiv bleibt, fordert Lambda außerdem alle ungenutzten Hyperplanen zurück ENIs und setzt den Funktionsstatus auf. `Inactive` Der nächste Aufrufversuch schlägt fehl, und die Funktion geht wieder in den Zustand „Ausstehend“ über, bis Lambda die Erstellung oder Zuweisung einer Hyperplane-ENI abgeschlossen hat. Wir empfehlen, dass Ihr Design nicht auf der Persistenz von basiert. ENIs

Wenn Sie eine Funktion aktualisieren, um ihre VPC-Konfiguration zu entfernen, benötigt Lambda bis zu 20 Minuten, um die angehängte Hyperplane-ENI zu löschen. Lambda löscht die ENI nur, wenn keine andere Funktion (oder veröffentlichte Funktionsversion) diese Hyperplane ENI verwendet.

Lambda verwendet Berechtigungen in der Ausführungsrolle der Funktion, um die Hyperplane ENI zu löschen. Wenn Sie die Ausführungsrolle löschen, bevor Lambda die Hyperplane ENI löscht, kann Lambda die Hyperplane ENI nicht löschen. Sie können den Löschvorgang manuell durchführen.

Verwenden von IAM-Bedingungsschlüsseln für VPC-Einstellungen

Sie können Lambda-spezifische Bedingungsschlüssel für VPC-Einstellungen verwenden, um zusätzliche Berechtigungssteuerungen für Ihre Lambda-Funktionen bereitzustellen. Sie können beispielsweise festlegen, dass alle Funktionen in Ihrer Organisation mit einer VPC sein müssen. Sie können auch die Subnetze und Sicherheitsgruppen angeben, die die Benutzer der Funktion verwenden können bzw. nicht verwenden können.

Lambda unterstützt die folgenden IAM-Bedingungsschlüssel:

- lambda: VpcIds — Erlaube oder verbiete eine oder mehrere. VPCs
- lambda: SubnetIds — Erlaubt oder verweigert ein oder mehrere Subnetze.
- lambda: SecurityGroupIds — Erlaubt oder verweigert eine oder mehrere Sicherheitsgruppen.

Die Lambda-API funktioniert CreateFunctionund UpdateFunctionConfigurationunterstützt diese Bedingungsschlüssel. Weitere Informationen zur Verwendung von Bedingungsschlüsseln in IAM-Richtlinien finden Sie unter IAM-JSON-Richtlinienelemente: Bedingung im IAM-Benutzerhandbuch.

> **Tip**
>
> Wenn Ihre Funktion bereits eine VPC-Konfiguration aus einer früheren API-Anforderung enthält, können Sie eine `UpdateFunctionConfiguration`-Anforderung ohne die VPC-Konfiguration senden.

Beispielrichtlinien mit Bedingungsschlüsseln für VPC-Einstellungen

In den folgenden Beispielen wird gezeigt, wie Bedingungsschlüssel für VPC-Einstellungen verwendet werden. Nachdem Sie eine Richtlinienanweisung mit den gewünschten Einschränkungen erstellt haben, fügen Sie die Richtlinienanweisung für den -Zielbenutzer oder die Zielrolle an.

Stellen Sie sicher, dass Benutzer nur VPC-verbundene Funktionen bereitstellen

Um sicherzustellen, dass alle Benutzer nur VPC-verbundene Funktionen bereitstellen, können Sie Erstellungs- und Aktualisierungsoperationen von Funktionen verweigern, die keine gültige VPC-ID enthalten.

Beachten Sie, dass die VPC-ID kein Eingabeparameter für die `CreateFunction`- oder `UpdateFunctionConfiguration` -Anforderung ist. Lambda ruft den VPC-ID-Wert basierend auf den Subnetz- und Sicherheitsgruppenparametern ab.

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Sid": "EnforceVPCFunction",
      "Action": [
          "lambda:CreateFunction",
          "lambda:UpdateFunctionConfiguration"
       ],
      "Effect": "Deny",
```

```
      "Resource": "*",
      "Condition": {
        "Null": {
           "lambda:VpcIds": "true"
        }
      }
    }
  ]
}
```

Verweigern Sie Benutzern den Zugriff auf bestimmte VPCs Subnetze oder Sicherheitsgruppen

Um Benutzern den Zugriff auf bestimmte Bereiche zu verweigern VPCs, verwenden Sie diese Option, `StringEquals` um den Wert der `lambda:VpcIds` Bedingung zu überprüfen. Im folgenden Beispiel wird Benutzern der Zugriff auf `vpc-1` und `vpc-2` verweigert.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "EnforceOutOfVPC",
            "Action": [
                "lambda:CreateFunction",
                "lambda:UpdateFunctionConfiguration"
            ],
            "Effect": "Deny",
            "Resource": "*",
            "Condition": {
                "StringEquals": {
                    "lambda:VpcIds": [
                        "vpc-1",
                        "vpc-2"
                    ]
                }
            }
        }
    ]
}
```

Um Benutzern den Zugriff auf bestimmte Subnetze zu verweigern, verwenden Sie `StringEquals`, um den Wert der `lambda:SubnetIds` Bedingung zu überprüfen. Im folgenden Beispiel wird Benutzern der Zugriff auf `subnet-1` und `subnet-2` verweigert.

```
{
      "Sid": "EnforceOutOfSubnet",
      "Action": [
          "lambda:CreateFunction",
          "lambda:UpdateFunctionConfiguration"
       ],
      "Effect": "Deny",
      "Resource": "*",
      "Condition": {
        "ForAnyValue:StringEquals": {
            "lambda:SubnetIds": ["subnet-1", "subnet-2"]
        }
      }
    }
```

Um Benutzern den Zugriff auf bestimmte Sicherheitsgruppen zu verweigern, verwenden Sie `StringEquals`, um den Wert der `lambda:SecurityGroupIds`-Bedingung zu überprüfen. Im folgenden Beispiel wird Benutzern der Zugriff auf `sg-1` und `sg-2` verweigert.

```
{
      "Sid": "EnforceOutOfSecurityGroups",
      "Action": [
          "lambda:CreateFunction",
          "lambda:UpdateFunctionConfiguration"
       ],
      "Effect": "Deny",
      "Resource": "*",
      "Condition": {
        "ForAnyValue:StringEquals": {
            "lambda:SecurityGroupIds": ["sg-1", "sg-2"]
        }
      }
    }
  ]
}
```

Benutzern das Erstellen und Aktualisieren von Funktionen mit bestimmten VPC-Einstellungen ermöglichen

Um Benutzern den Zugriff auf bestimmte Bedingungen zu ermöglichen VPCs, verwenden Sie diese Option, `StringEquals` um den Wert der `lambda:VpcIds` Bedingung zu überprüfen. Im folgenden Beispiel können Benutzer auf `vpc-1` und `vpc-2` zugreifen.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "EnforceStayInSpecificVpc",
            "Action": [
                "lambda:CreateFunction",
                "lambda:UpdateFunctionConfiguration"
            ],
            "Effect": "Allow",
            "Resource": "*",
            "Condition": {
                "StringEquals": {
                    "lambda:VpcIds": [
                        "vpc-1",
                        "vpc-2"
                    ]
                }
            }
        }
    ]
}
```

Um Benutzern den Zugriff auf bestimmte Subnetze zu ermöglichen, verwenden Sie `StringEquals`, um den Wert der `lambda:SubnetIds`-Bedingung zu überprüfen. Im folgenden Beispiel können Benutzer auf `subnet-1` und `subnet-2` zugreifen.

```
{
      "Sid": "EnforceStayInSpecificSubnets",
      "Action": [
          "lambda:CreateFunction",
```

```
            "lambda:UpdateFunctionConfiguration"
        ],
        "Effect": "Allow",
        "Resource": "*",
        "Condition": {
          "ForAllValues:StringEquals": {
              "lambda:SubnetIds": ["subnet-1", "subnet-2"]
          }
        }
    }
```

Um Benutzern den Zugriff auf bestimmte Sicherheitsgruppen zu ermöglichen, verwenden Sie `StringEquals`, um den Wert der `lambda:SecurityGroupIds`-Bedingung zu überprüfen. Im folgenden Beispiel können Benutzer auf `sg-1` und `sg-2` zugreifen.

```
{
      "Sid": "EnforceStayInSpecificSecurityGroup",
      "Action": [
          "lambda:CreateFunction",
          "lambda:UpdateFunctionConfiguration"
       ],
      "Effect": "Allow",
      "Resource": "*",
      "Condition": {
        "ForAllValues:StringEquals": {
            "lambda:SecurityGroupIds": ["sg-1", "sg-2"]
        }
      }
    }
  ]
}
```

VPC-Tutorials

In den folgenden Tutorials verbinden Sie eine Lambda-Funktion mit Ressourcen in Ihrer VPC.

- Tutorial: Verwenden einer Lambda-Funktion für den Zugriff auf Amazon RDS in einer Amazon VPC
- Tutorial: Konfiguration einer Lambda-Funktion für den Zugriff auf Amazon ElastiCache in einer Amazon VPC

Lambda-Funktionen Zugriff auf eine Ressource in einer Amazon VPC in einem anderen Konto gewähren

Sie können Ihrer AWS Lambda Funktion Zugriff auf eine Ressource in einer Amazon VPC in Amazon Virtual Private Cloud gewähren, die von einem anderen Konto verwaltet wird, ohne dass eine der VPC dem Internet ausgesetzt wird. Dieses Zugriffsmuster ermöglicht es Ihnen, Daten mit anderen Organisationen gemeinsam zu nutzen. AWS Mithilfe dieses Zugriffsmusters können Sie Daten VPCs mit einem höheren Sicherheits- und Leistungsniveau austauschen als über das Internet. Konfigurieren Sie Ihre Lambda-Funktion so, dass sie eine Amazon VPC-Peering-Verbindung für den Zugriff auf diese Ressourcen verwendet.

> **Warning**
>
> Wenn Sie den Zugriff zwischen Konten zulassen oder überprüfen Sie VPCs, ob Ihr Tarif die Sicherheitsanforderungen der jeweiligen Organisationen erfüllt, die diese Konten verwalten. Wenn Sie die Anweisungen in diesem Dokument befolgen, wirkt sich dies auf die Sicherheitslage Ihrer Ressourcen aus.

In diesem Tutorial verbinden Sie zwei Konten über eine Peering-Verbindung mit IPv4. Sie konfigurieren eine Lambda-Funktion, die noch nicht mit einer Amazon VPC verbunden ist. Sie konfigurieren die DNS-Auflösung, um Ihre Funktion mit Ressourcen zu verbinden, die keine statischen IPs Daten bereitstellen. Informationen zur Anpassung dieser Anweisungen an andere Peering-Szenarien finden Sie im VPC Peering Guide.

Voraussetzungen

Um einer Lambda-Funktion Zugriff auf eine Ressource in einem anderen Konto zu gewähren, benötigen Sie:

- Eine Lambda-Funktion, die so konfiguriert ist, dass sie sich mit Ihrer Ressource authentifiziert und dann aus ihr liest.
- Eine Ressource in einem anderen Konto, z. B. einem Amazon-RDS-Cluster, ist über Amazon VPC verfügbar.
- Anmeldeinformationen für das Konto Ihrer Lambda-Funktion und das Konto Ihrer Ressource. Wenn Sie nicht berechtigt sind, das Konto Ihrer Ressource zu verwenden, wenden Sie sich an einen autorisierten Benutzer, um dieses Konto einzurichten.

- Berechtigung zum Erstellen und Aktualisieren einer VPC (und zur Unterstützung von Amazon-VPC-Ressourcen), die mit Ihrer Lambda-Funktion verknüpft werden soll.
- Berechtigung zum Aktualisieren der Ausführungsrolle und der VPC-Konfiguration für Ihre Lambda-Funktion.
- Berechtigung zum Erstellen einer VPC-Peering-Verbindung im Konto Ihrer Lambda-Funktion.
- Berechtigung zum Annehmen einer VPC-Peering-Verbindung im Konto Ihrer Ressource.
- Erlaubnis, die Konfiguration der VPC Ihrer Ressource (und der Unterstützung von Amazon-VPC-Ressourcen) zu aktualisieren.
- Berechtigung zum Aufrufen Ihrer Lambda-Funktion.

Erstellen Sie eine Amazon VPC im Konto Ihrer Funktion

Erstellen Sie eine Amazon VPC, Subnetze, Routing-Tabellen und eine Sicherheitsgruppe im Konto Ihrer Lambda-Funktion.

So erstellen Sie eine VPC, Subnetze und weitere VPC-Ressourcen mit der Konsole

1. Öffnen Sie die Amazon-VPC-Konsole unter https://console.aws.amazon.com/vpc/.
2. Wählen Sie auf dem VPC-Dashboard VPC erstellen aus.
3. Geben Sie für den IPv4 CIDR-Block einen privaten CIDR-Block an. Ihr CIDR-Block darf sich nicht mit Blöcken überlappen, die in der VPC Ihrer Ressource verwendet werden. Wählen Sie keinen Block aus, den die VPC Ihrer Ressourcen verwendet, um sie Ressourcen IPs zuzuweisen, oder einen Block, der bereits in den Routentabellen in Ihrer Ressourcen-VPC definiert ist. Weitere Informationen zur Definition geeigneter CIDR-Blöcke finden Sie unter VPC CIDR-Blöcke.
4. Wählen Sie Anpassen AZs aus.
5. Wählen Sie dasselbe AZs wie Ihre Ressource aus.
6. Wählen Sie für Number of public subnets (Anzahl der öffentlichen Subnetze) 0 aus.
7. Wählen Sie für VPC endpoints (VPC-Endpunkte) None (Keine) aus.
8. Wählen Sie VPC erstellen aus.

Gewährung von VPC-Berechtigungen für die Ausführungsrolle Ihrer Funktion

Hängen Sie es AWSLambdaVPCAccessExecutionRolean die Ausführungsrolle Ihrer Funktion an, damit sie eine Verbindung herstellen kann VPCs.

So gewähren Sie der Ausführungsrolle Ihrer Funktion VPC-Berechtigungen

1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.
2. Wählen Sie den Namen Ihrer Funktion aus.
3. Wählen Sie Konfiguration.
4. Wählen Sie Berechtigungen.
5. Wählen Sie unter Rollenname die Ausführungsrolle.
6. Wählen Sie im Abschnitt Berechtigungsrichtlinien die Option Berechtigungen hinzufügen.
7. Wählen Sie in der Dropdown-Liste Richtlinien anhängen.
8. Geben Sie in das Suchfeld `AWSLambdaVPCAccessExecutionRole` ein.
9. Aktivieren Sie links neben dem Richtliniennamen das Kontrollkästchen.
10. Wählen Sie Add permissions (Berechtigungen hinzufügen) aus.

Um Ihre Funktion an Ihre Amazon VPC anzuhängen

1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.
2. Wählen Sie den Namen Ihrer Funktion aus.
3. Wählen Sie die Registerkarte Konfiguration und anschließend VPC aus.
4. Wählen Sie Bearbeiten aus.
5. Wählen Sie unter VPC Ihre VPC aus
6. Wählen Sie unter Subnetze Ihre Subnetze aus.
7. Wählen Sie unter Sicherheitsgruppen die Standardsicherheitsgruppe für Ihre VPC.
8. Wählen Sie Speichern.

Erstellen Sie eine VPC-Peering-Verbindungsanfrage

Erstellen Sie eine VPC-Peering-Verbindungsanforderung von der VPC Ihrer Funktion (der VPC des Anforderers) zur VPC Ihrer Ressource (der Akzepter-VPC).

So erstellen Sie eine VPC-Peering-Verbindung von der VPC Ihrer Funktion

1. Öffnen Sie die https://console.aws.amazon.com/vpc/.
2. Wählen Sie im Navigationsbereich Peering connections (Peering-Verbindungen) aus.
3. Wählen Sie Create peering connection (Peering-Verbindung erstellen).
4. Wählen Sie für VPC-ID (Requester) die VPC Ihrer Funktion aus.
5. Geben Sie unter Konto-ID die ID des Kontos Ihrer Ressource ein.
6. Geben Sie für VPC ID (Accepter) die VPC Ihrer Ressource ein.

Bereiten Sie das Konto Ihrer Ressource vor

Um Ihre Peering-Verbindung herzustellen und die VPC Ihrer Ressource für die Verwendung der Verbindung vorzubereiten, melden Sie sich beim Konto Ihrer Ressource mit einer Rolle an, die die in den Voraussetzungen aufgeführten Berechtigungen besitzt. Die Schritte zur Anmeldung können je nachdem, wie das Konto gesichert ist, unterschiedlich sein. Weitere Informationen zur Anmeldung bei einem AWS Konto finden Sie im AWS Anmelde-Benutzerhandbuch. Führen Sie im Konto Ihrer Ressource die folgenden Verfahren aus.

So akzeptieren Sie die VPC-Peering-Verbindungsanfrage

1. Öffnen Sie die https://console.aws.amazon.com/vpc/.
2. Wählen Sie im Navigationsbereich Peering connections (Peering-Verbindungen) aus.
3. Wählen Sie erst die ausstehende VPC-Peering-Verbindung (der Status lautet pending-acceptance) aus.
4. Wählen Sie Aktionen.
5. Wählen Sie in der Dropdown-Liste die Option Anfrage annehmen aus.
6. Wenn Sie zur Bestätigung aufgefordert werden, wählen Sie Akzeptieren aus.
7. Wählen Sie Meine Routing-Tabellen jetzt ändern, um der Haupt-Routing-Tabelle für Ihre VPC eine Route hinzuzufügen, sodass Sie Traffic über die Peering-Verbindung senden und empfangen können.

Untersuchen Sie die Routentabellen für die VPC der Ressource. Die von Amazon VPC generierte Route stellt möglicherweise keine Konnektivität her, je nachdem, wie die VPC Ihrer Ressource eingerichtet ist. Suchen Sie nach Konflikten zwischen der neuen Route und der vorhandenen Konfiguration für die VPC. Weitere Informationen zur Fehlerbehebung finden Sie

unter Problembehandlung bei einer VPC-Peering-Verbindung im Amazon Virtual Private Cloud VPC Peering Guide.

So aktualisieren Sie die Sicherheitsgruppe für Ihre Ressource

1. Öffnen Sie die https://console.aws.amazon.com/vpc/.
2. Wählen Sie im Navigationsbereich Sicherheitsgruppen aus.
3. Wählen Sie die Sicherheitsgruppe für Ihre Ressource aus.
4. Wählen Sie Aktionen.
5. Wählen Sie in der Dropdownliste die Option Regeln für eingehenden Datenverkehr bearbeiten aus.
6. Wählen Sie Regel hinzufügen aus.
7. Geben Sie im Feld Quelle die Konto-ID und die Sicherheitsgruppen-ID Ihrer Funktion ein, getrennt durch einen Schrägstrich (z. B. 111122223333/sg-1a2b3c4d).
8. Wählen Sie Edit outbound rules (Regeln für ausgehenden Datenverkehr bearbeiten).
9. Prüfen Sie, ob der ausgehende Verkehr eingeschränkt ist. Die VPC-Standardeinstellungen gestatten allen ausgehenden Datenverkehr. Wenn der ausgehende Datenverkehr eingeschränkt ist, fahren Sie mit dem nächsten Schritt fort.
10. Wählen Sie Regel hinzufügen aus.
11. Geben Sie im Feld Ziel die Konto-ID und die Sicherheitsgruppen-ID Ihrer Funktion ein, getrennt durch einen Schrägstrich (z. B. 111122223333/sg-1a2b3c4d).
12. Wählen Sie Regeln speichern aus.

Aktivieren einer DNS-Auflösung für Ihre Peering-Verbindung

1. Öffnen Sie die https://console.aws.amazon.com/vpc/.
2. Wählen Sie im Navigationsbereich Peering connections (Peering-Verbindungen) aus.
3. Wählen Sie Ihre Peering-Verbindung aus.
4. Wählen Sie Aktionen.
5. Wählen Sie DNS-Einstellungen ändern aus.
6. Wählen Sie unter Akzepter-DNS-Auflösung die Option Erlauben Sie dem anfragenden VPC, DNS von akzeptierenden VPC-Hosts in private IP aufzulösen aus.
7. Wählen Sie Änderungen speichern aus.

Aktualisieren Sie die VPC-Konfiguration im Konto Ihrer Funktion

Melden Sie sich beim Konto Ihrer Funktion an und aktualisieren Sie die VPC-Konfiguration.

So fügen Sie Ihrer VPC-Peering-Verbindung eine Route hinzu

1. Öffnen Sie die https://console.aws.amazon.com/vpc/.
2. Wählen Sie im Navigationsbereich Route Tables (Routing-Tabellen) aus.
3. Aktivieren Sie das Kontrollkästchen neben dem Namen der Routentabelle für das Teilnetz, das Sie mit Ihrer Funktion verknüpft haben.
4. Wählen Sie Aktionen.
5. Wählen Sie Routen bearbeiten aus.
6. Wählen Sie Route hinzufügen aus.
7. Geben Sie unter Ziel den CIDR-Block für die VPC Ihrer Ressource ein.
8. Wählen Sie als Ziel Ihre VPC-Peering-Verbindung aus.
9. Wählen Sie Änderungen speichern aus.

Weitere Informationen zu den Überlegungen, auf die Sie bei der Aktualisierung Ihrer Routing-Tabellen stoßen können, finden Sie unter Aktualisieren der Routing-Tabellen für eine VPC-Peering-Verbindung.

Um die Sicherheitsgruppe für Ihre Lambda-Funktion zu aktualisieren

1. Öffnen Sie die https://console.aws.amazon.com/vpc/.
2. Wählen Sie im Navigationsbereich Sicherheitsgruppen aus.
3. Wählen Sie Aktionen.
4. Wählen Sie Edit inbound rules (Regeln für eingehenden Datenverkehr bearbeiten) aus.
5. Wählen Sie Regel hinzufügen aus.
6. Geben Sie im Feld Quelle die Konto-ID und die Sicherheitsgruppen-ID Ihrer Ressource ein, getrennt durch einen Schrägstrich (z. B. 111122223333/sg-1a2b3c4d).
7. Wählen Sie Regeln speichern aus.

Aktivieren einer DNS-Auflösung für Ihre Peering-Verbindung

1. Öffnen Sie die https://console.aws.amazon.com/vpc/.

2. Wählen Sie im Navigationsbereich Peering connections (Peering-Verbindungen) aus.
3. Wählen Sie Ihre Peering-Verbindung aus.
4. Wählen Sie Aktionen.
5. Wählen Sie DNS-Einstellungen ändern aus.
6. Wählen Sie unter DNS-Auflösung des Antragstellers die Option Zulassen, dass der akzeptierende VPC die DNS von VPC-Hosts des Antragstellers in eine private IP auflöst aus.
7. Wählen Sie Änderungen speichern aus.

Testen der -Funktion

So erstellen Sie ein Testereignis und überprüfen die Ausgabe Ihrer Funktion

1. Wählen Sie im Bereich Codequelle die Option Testen aus.
2. Wählen Sie Neues Ereignis erstellen.
3. Ersetzen Sie im Bereich Event JSON die Standardwerte durch eine Eingabe, die für Ihre Lambda-Funktion geeignet ist.
4. Wählen Sie aufrufen aus.
5. Vergewissern Sie sich auf der Registerkarte Ausführungsergebnisse, dass die Antwort Ihre erwartete Ausgabe enthält.

Darüber hinaus können Sie die Protokolle Ihrer Funktion überprüfen, um sicherzustellen, dass die Protokolle Ihren Erwartungen entsprechen.

Um die Aufrufaufzeichnungen Ihrer Funktion in CloudWatch Logs anzuzeigen

1. Wählen Sie den Tab Überwachung.
2. Wählen Sie CloudWatch Protokolle anzeigen aus.
3. Wählen Sie auf der Registerkarte Protokollstreams den Protokollstream für den Aufruf Ihrer Funktion aus.
4. Vergewissern Sie sich, dass Ihre Protokolle Ihren Erwartungen entsprechen.

Aktivieren Sie den Internetzugang für VPC-verbundene Lambda-Funktionen

Standardmäßig werden Lambda-Funktionen in einer von Lambda verwalteten VPC mit Internetzugang ausgeführt. Um auf Ressourcen in einer VPC in Ihrem Konto zuzugreifen, können Sie einer Funktion eine VPC-Konfiguration hinzufügen. Dadurch wird die Funktion auf Ressourcen innerhalb dieser VPC beschränkt, es sei denn, die VPC hat Internetzugang. Auf dieser Seite wird erklärt, wie Sie den Internetzugang für mit VPN verbundene Lambda-Funktionen bereitstellen.

Ich habe noch keine VPC

Erstellen Sie die VPC

Der VPC-Workflow erstellen erstellt alle VPC-Ressourcen, die für eine Lambda-Funktion für den Zugriff auf das öffentliche Internet von einem privaten Subnetz aus erforderlich sind, einschließlich Subnetzen, NAT-Gateways, Internet-Gateways und Routing-Tabelleneinträgen.

So erstellen Sie die VPC

1. Öffnen Sie die Amazon-VPC-Konsole unter https://console.aws.amazon.com/vpc/.
2. Wählen Sie auf dem VPC-Dashboard VPC erstellen aus.
3. Wählen Sie unter Zu erstellende Ressourcen die Option VPC und mehr aus.
4. Konfigurieren Sie die VPC
    a. Geben Sie unter Name tag auto-generation (Automatische Generierung des Namens-Tags) einen Namen für die VPC ein.
    b. Behalten Sie für den IPv4-CIDR-Block entweder den Standardvorschlag bei oder geben Sie den für Ihre Anwendung oder Ihr Netzwerk erforderlichen CIDR-Block ein.
    c. Wenn die Anwendung über IPv6-Adressen kommuniziert, wählen Sie IPv6-CIDR-Block und Von Amazon bereitgestellter IPv6-CIDR-Block aus.
5. Subnetze konfigurieren
    a. Wählen Sie für Anzahl der Availability Zones 2 aus. Wir empfehlen mindestens zwei AZs für hohe Verfügbarkeit.
    b. Wählen Sie für Number of public subnets (Anzahl der öffentlichen Subnetze) 2 aus.
    c. Wählen Sie für Number of private subnets (Anzahl der privaten Subnetze) 2 aus.

d. Sie können die standardmäßigen CIDR-Blöcke für die Subnetze beibehalten oder alternativ CIDR-Blöcke des Subnetzes anpassen erweitern und einen CIDR-Block eingeben. Weitere Informationen finden Sie unter CIDR-Blöcke für das Subnetz.

6. Wählen Sie für NAT-Gateways 1 pro AZ aus, um die Resilienz zu verbessern.
7. Wählen Sie für ein Internet-Gateway nur für ausgehenden Datenverkehr die Option Ja aus, wenn Sie sich dafür entschieden haben, einen IPv6-CIDR-Block einzubeziehen.
8. Behalten Sie für VPC-Endpunkte den Standardwert (S3-Gateway) bei. Für diese Option fallen keine Kosten an. Weitere Informationen finden Sie unter Arten von VPC-Endpunkten für Amazon S3.
9. Behalten Sie für die DNS-Optionen die Standardeinstellungen bei.
10. Wählen Sie VPC erstellen aus.

Konfigurieren der Lambda-Funktion

So konfigurieren Sie eine VPC bei der Funktionserstellung

1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.
2. Wählen Sie Create function (Funktion erstellen).
3. Geben Sie unter Basic information (Grundlegende Informationen) bei Function name (Funktionsname) einen Namen für Ihre Funktion ein.
4. Erweiterten Sie Advanced settings (Erweiterte Einstellungen).
5. Wählen Sie VPC aktivieren und wählen Sie dann eine VPC aus.
6. (Optional) Um ausgehenden IPv6-Verkehr zuzulassen, wählen Sie Allow IPv6 traffic for dual-stack subnets (IPv6-Verkehr für Dual-Stack-Subnetze zulassen) aus.
7. Wählen Sie für Subnetze alle privaten Subnetze aus. Die privaten Subnetze können über die NAT-Gateway auf das Internet zugreifen. Durch die Verbindung einer Funktion mit einem öffentlichen Subnetz erhält sie keinen Internetzugang.

> **Note**
>
> Wenn Sie Allow IPv6 traffic for dual-stack subnets (IPv6-Verkehr für Dual-Stack-Subnetze zulassen) ausgewählt haben, müssen alle ausgewählten Subnetze einen IPv4-CIDR-Block und einen IPv6-CIDR-Block besitzen.

8. Wählen Sie für Sicherheitsgruppen eine Sicherheitsgruppe aus, die ausgehenden Datenverkehr zulässt.
9. Wählen Sie Funktion erstellen.

Lambda erstellt automatisch eine Ausführungsrolle mit der AWS-verwalteten Richtlinie AWSLambdaVPCAccessExecutionRole. Die Berechtigungen in dieser Richtlinie sind nur zum Erstellen von Elastic Network-Schnittstellen für die VPC-Konfiguration und nicht zum Aufrufen Ihrer Funktion erforderlich. Um Berechtigungen mit den geringsten Rechten anzuwenden, können Sie die Richtlinie AWSLambdaVPCAccessExecutionRole aus Ihrer Ausführungsrolle entfernen, nachdem Sie die Funktion und die VPC-Konfiguration erstellt haben. Weitere Informationen finden Sie unter Erforderliche IAM-Berechtigungen.

So konfigurieren Sie eine VPC für eine vorhandene Funktion

Um einer vorhandenen Funktion eine VPC-Konfiguration hinzuzufügen, muss die Ausführungsrolle der Funktion über die Berechtigung verfügen, elastische Netzwerkschnittstellen zu erstellen und zu verwalten. Die verwaltete Richtlinie AWSLambdaVPCAccessExecutionRole AWS enthält die erforderlichen Berechtigungen. Um Berechtigungen mit den geringsten Rechten anzuwenden, können Sie die Richtlinie AWSLambdaVPCAccessExecutionRole aus Ihrer Ausführungsrolle entfernen, nachdem Sie die VPC-Konfiguration erstellt haben.

1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.
2. Wählen Sie eine Funktion aus.
3. Wählen Sie die Registerkarte Konfiguration und anschließend VPC aus.
4. Wählen Sie unter VPC die Option Edit (Bearbeiten) aus.
5. Wählen Sie die VPC aus.
6. (Optional) Um ausgehenden IPv6-Verkehr zuzulassen, wählen Sie Allow IPv6 traffic for dual-stack subnets (IPv6-Verkehr für Dual-Stack-Subnetze zulassen) aus.
7. Wählen Sie für Subnetze alle privaten Subnetze aus. Die privaten Subnetze können über die NAT-Gateway auf das Internet zugreifen. Durch die Verbindung einer Funktion mit einem öffentlichen Subnetz erhält sie keinen Internetzugang.

> **Note**
>
> Wenn Sie Allow IPv6 traffic for dual-stack subnets (IPv6-Verkehr für Dual-Stack-Subnetze zulassen) ausgewählt haben, müssen alle ausgewählten Subnetze einen IPv4-CIDR-Block und einen IPv6-CIDR-Block besitzen.

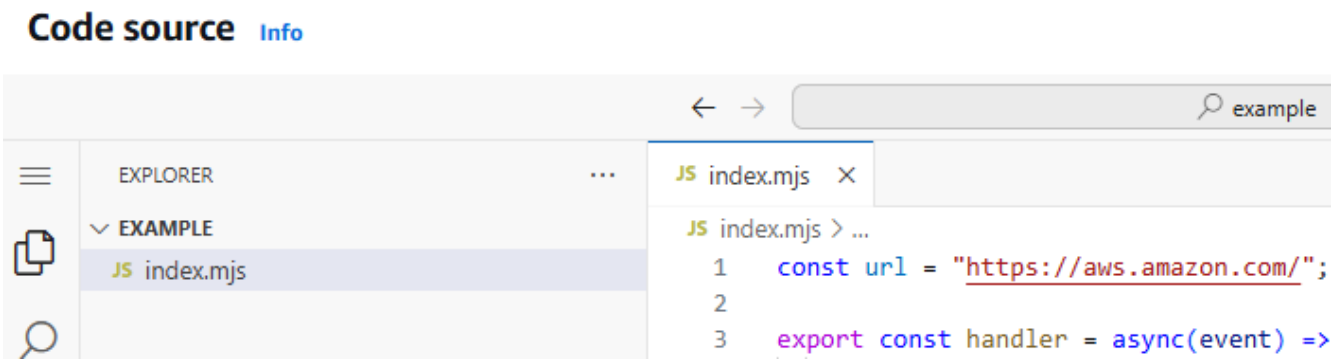
8. Wählen Sie für Sicherheitsgruppen eine Sicherheitsgruppe aus, die ausgehenden Datenverkehr zulässt.
9. Wählen Sie Speichern.

Testen der Funktion

Verwenden Sie den folgenden Beispielcode, um zu bestätigen, dass Ihre VPC-verbundene Funktion das öffentliche Internet erreichen kann. Bei Erfolg gibt der Code einen `200`-Statuscode zurück. Ist dies nicht der Fall, wird die Funktion abgebrochen.

Node.js

1. Fügen Sie im Bereich Codequelle der Lambda-Konsole den folgenden Code in die Datei index.mjs ein. Die Funktion stellt eine HTTP-GET-Anfrage an einen öffentlichen Endpunkt und gibt den HTTP-Antwortcode zurück, um zu prüfen, ob die Funktion Zugang zum öffentlichen Internet hat.

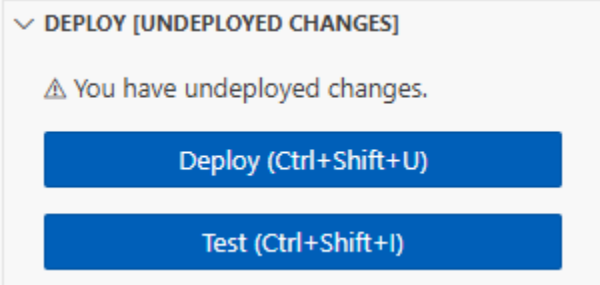


Example – HTTP-Anforderung mit async/await

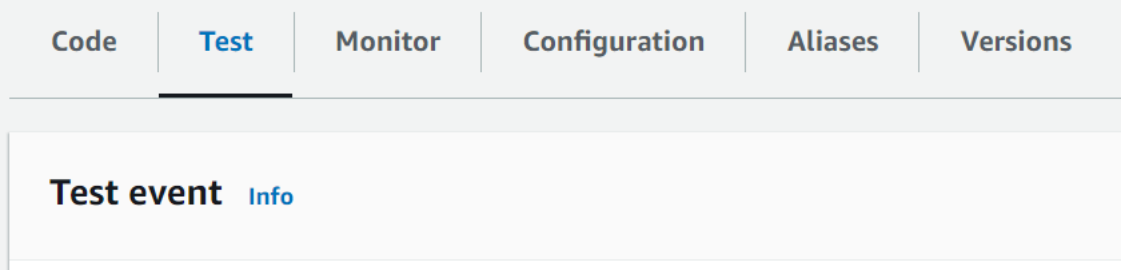
```
const url = "https://aws.amazon.com/";

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;
    }
};
```

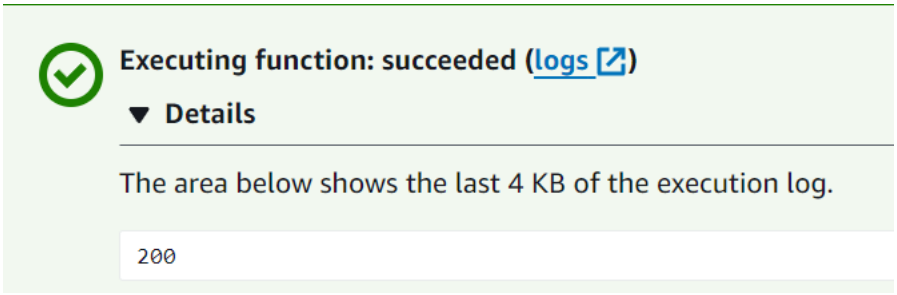
2. Wählen Sie im Abschnitt BEREITSTELLEN die Option Bereitstellen aus, um den Code Ihrer Funktion zu aktualisieren:



3. Wählen Sie die Registerkarte Test.



4. Wählen Sie Test aus.
5. Die Funktion gibt einen `200`-Statuscode zurück. Das bedeutet, dass die Funktion über einen ausgehenden Internetzugang verfügt.

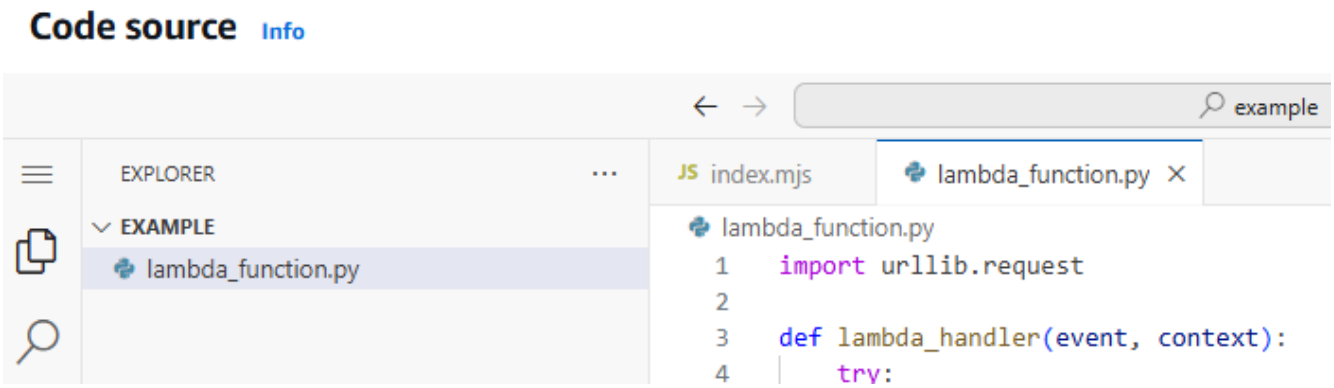


Wenn die Funktion das öffentliche Internet nicht erreichen kann, erhalten Sie eine Fehlermeldung wie die folgende:

```
{
  "errorMessage": "2024-04-11T17:22:20.857Z abe12jlc-640a-8157-0249-9be825c2y110
 Task timed out after 3.01 seconds"
}
```

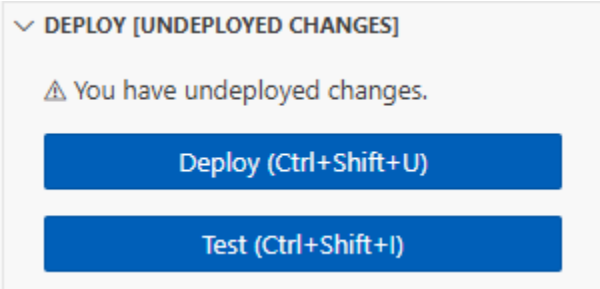
Python

1. Fügen Sie im Bereich Codequelle der Lambda-Konsole den folgenden Code in die Datei lambda_function.py ein. Die Funktion stellt eine HTTP-GET-Anfrage an einen öffentlichen Endpunkt und gibt den HTTP-Antwortcode zurück, um zu prüfen, ob die Funktion Zugang zum öffentlichen Internet hat.

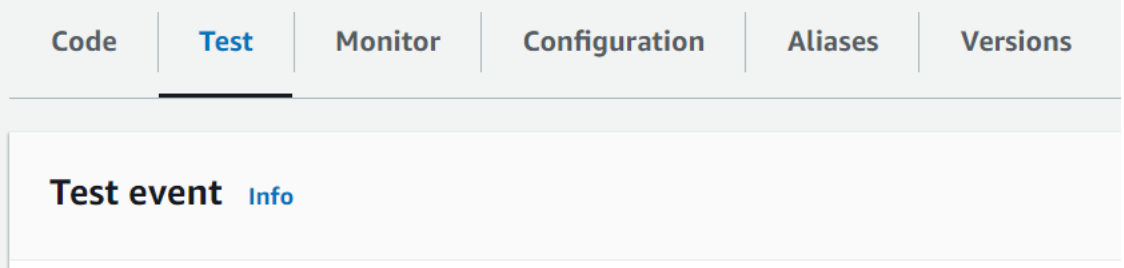



```
import urllib.request

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
```

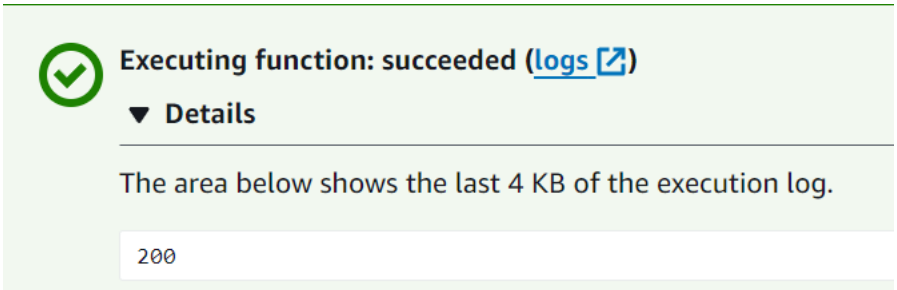
2. Wählen Sie im Abschnitt BEREITSTELLEN die Option Bereitstellen aus, um den Code Ihrer Funktion zu aktualisieren:



3. Wählen Sie die Registerkarte Test.



4. Wählen Sie Test aus.
5. Die Funktion gibt einen 200-Statuscode zurück. Das bedeutet, dass die Funktion über einen ausgehenden Internetzugang verfügt.



Wenn die Funktion das öffentliche Internet nicht erreichen kann, erhalten Sie eine Fehlermeldung wie die folgende:

```
{
  "errorMessage": "2024-04-11T17:22:20.857Z abe12jlc-640a-8157-0249-9be825c2y110
 Task timed out after 3.01 seconds"
}
```

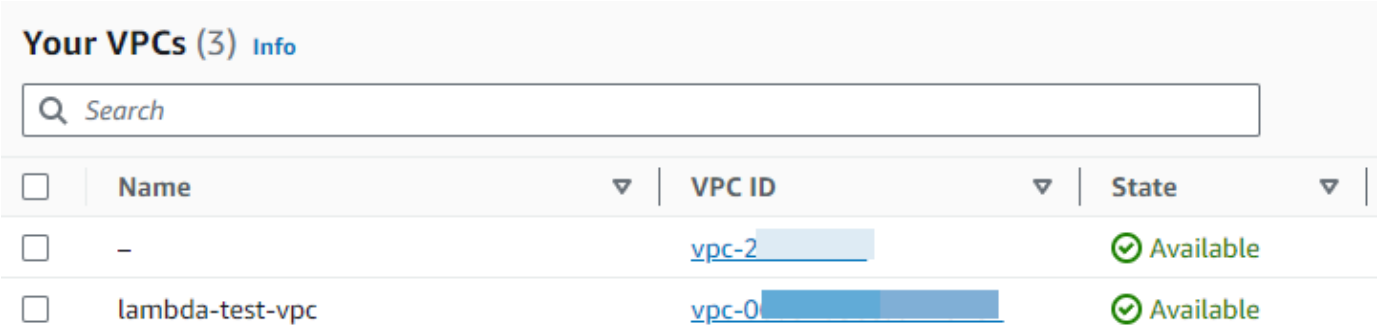
Ich habe bereits eine VPC

Wenn Sie bereits über eine VPC verfügen, aber den öffentlichen Internetzugang für eine Lambda-Funktion konfigurieren müssen, gehen Sie wie folgt vor. Bei diesem Verfahren wird davon

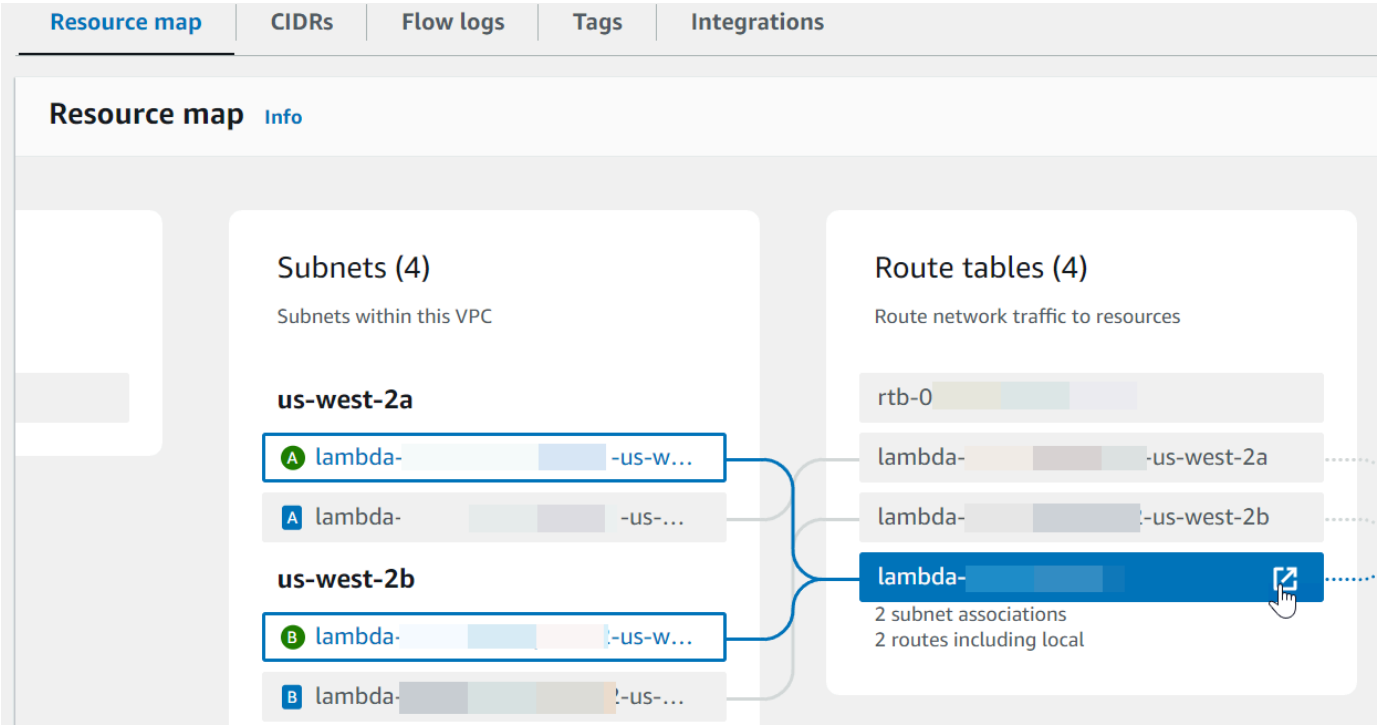
ausgegangen, dass Ihre VPC mindestens zwei Subnetze hat. Wenn Sie nicht über zwei Subnetze verfügen, finden Sie weitere Informationen unter Erstellen eines Subnetzes im Amazon-VPC-Benutzerhandbuch.

Überprüfen der Routing-Tabellen

1. Öffnen Sie die Amazon-VPC-Konsole unter https://console.aws.amazon.com/vpc/.
2. Wählen Sie die VPC ID aus.



3. Scrollen Sie nach unten zum Abschnitt Ressourcenkarte. Beachten Sie die Zuordnungen in den Routing-Tabellen. Öffnen Sie jede Routing-Tabelle, die einem Subnetz zugeordnet ist.

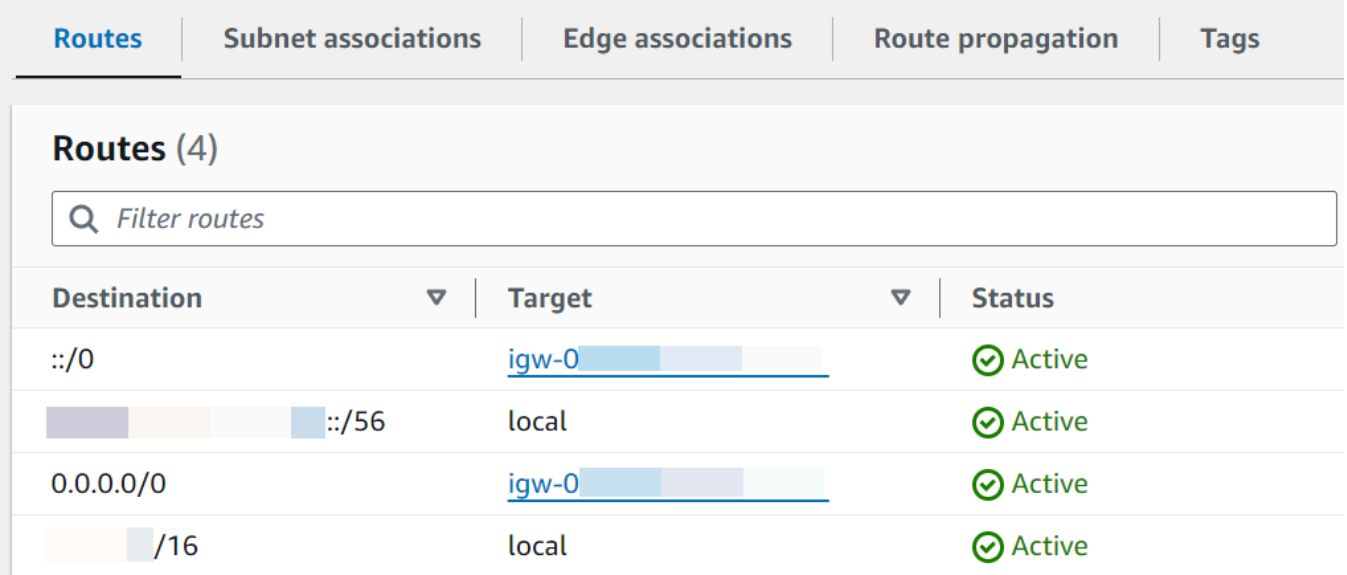


4. Scrollen Sie nach unten zur Registerkarte Routen. Überprüfen Sie die Routen, um festzustellen, ob Ihre VPC über die beiden folgenden Routing-Tabellen verfügt. Jede dieser Anforderungen muss durch eine separate Routing-Tabelle erfüllt werden.

- Internetdatenverkehr (`0.0.0.0/0` für IPv4, `::/0` für IPv6) wird an ein Internet-Gateway (`igw-xxxxxxxxxx`) weitergeleitet. Das bedeutet, dass das der Routing-Tabelle zugeordnete Subnetz ein öffentliches Subnetz ist.

  > **Note**
  >
  > Wenn Ihr Subnetz keinen IPv6-CIDR-Block hat, wird Ihnen nur die IPv4-Route (`0.0.0.0/0`) angezeigt.

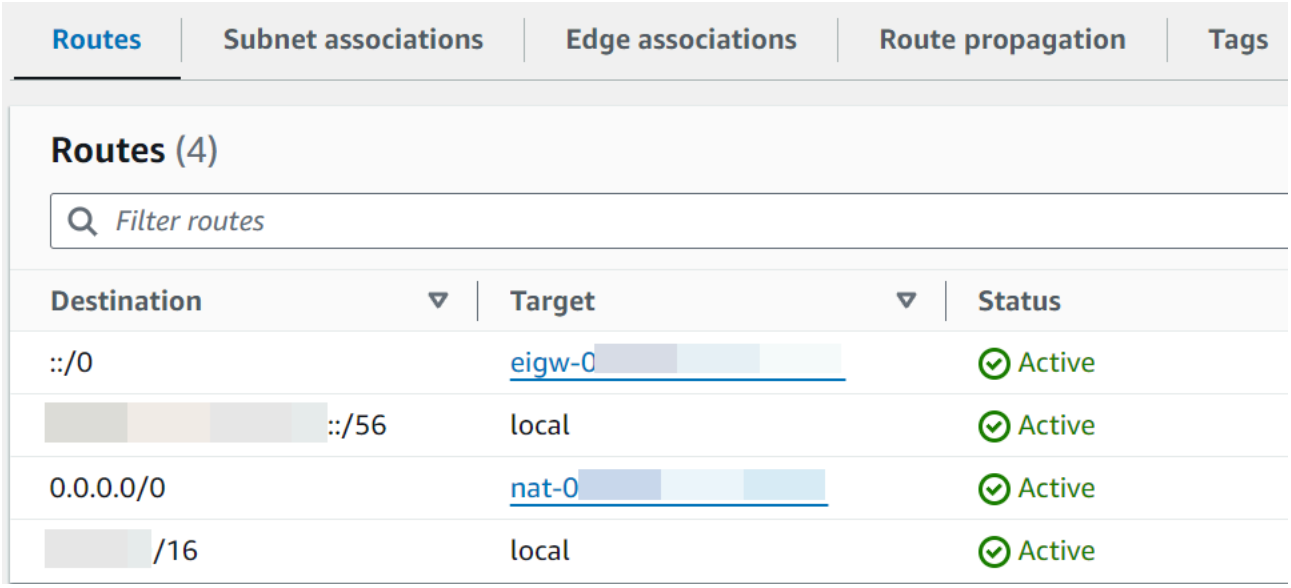
  Example öffentliche Subnetz-Routing-Tabelle



- Internetgebundener Datenverkehr für IPv4 (`0.0.0.0/0`) wird an ein NAT-Gateway (`nat-xxxxxxxxxx`) weitergeleitet, das einem öffentlichen Subnetz zugeordnet ist. Dies bedeutet, dass das Subnetz ein privates Subnetz ist, das über das NAT-Gateway auf das Internet zugreifen kann.

  > **Note**
  >
  > Wenn Ihr Subnetz über einen IPv6-CIDR-Block verfügt, muss die Routing-Tabelle auch IPv6-Datenverkehr (`::/0`) an ein Internet-Gateway für ausgehenden Verkehr (`eigw-xxxxxxxxxx`) routet. Wenn Ihr Subnetz keinen IPv6-CIDR-Block hat, wird Ihnen nur die IPv4-Route (`0.0.0.0/0`) angezeigt.

Example Routing-Tabelle für private Subnetze



5. Wiederholen Sie den vorherigen Schritt, bis Sie alle Routing-Tabellen überprüft haben, die einem Subnetz in Ihrer VPC zugeordnet sind, und bestätigt haben, dass Sie über eine Routing-Tabelle mit einem Internet-Gateway und eine Routing-Tabelle mit einem NAT-Gateway verfügen.

   Wenn Sie nicht über zwei Routing-Tabellen verfügen, eine mit einer Route zu einem Internet-Gateway und eine mit einer Route zu einem NAT-Gateway, gehen Sie wie folgt vor, um die fehlenden Ressourcen und Routing-Tabelleneinträge zu erstellen.

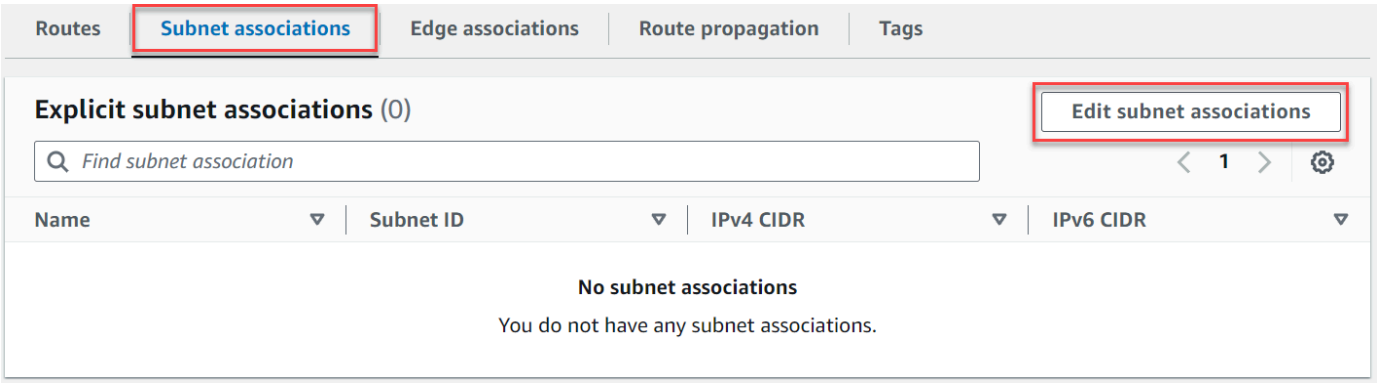
Erstellen einer Routing-Tabelle

Gehen Sie wie folgt vor, um eine Routing-Tabelle zu erstellen und diese mit einem Subnetz zu verknüpfen.

Erstellen einer benutzerdefinierten Routing-Tabelle mithilfe der Amazon-VPC-Konsole

1. Öffnen Sie die Amazon-VPC-Konsole unter https://console.aws.amazon.com/vpc/.
2. Wählen Sie im Navigationsbereich Route Tables (Routing-Tabellen) aus.
3. Klicken Sie auf Create Route Table (Routing-Tabelle erstellen).
4. (Optional) Geben Sie bei Name einen Namen für Ihre Routing-Tabelle ein.
5. Wählen Sie unter VPC Ihre VPC aus.

6. (Optional) Sie fügen ein Tag hinzu, indem Sie Add new tag (Neuen Tag hinzufügen) auswählen und den Tag-Schlüssel und -Wert eingeben.
7. Klicken Sie auf Create Route Table (Routing-Tabelle erstellen).
8. Wählen Sie auf der Registerkarte Subnet associations (Subnetzzuordnungen) die Option Edit subnet associations (Subnetzzuordnungen bearbeiten) aus.



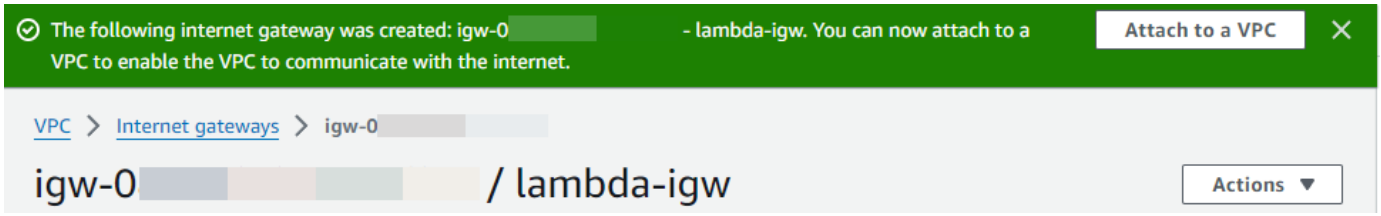
9. Aktivieren Sie das Kontrollkästchen für das Subnetz, um es der Routing-Tabelle zuzuordnen.
10. Klicken Sie auf Save associations (Zuordnungen speichern).

Ein Internet-Gateway erstellen

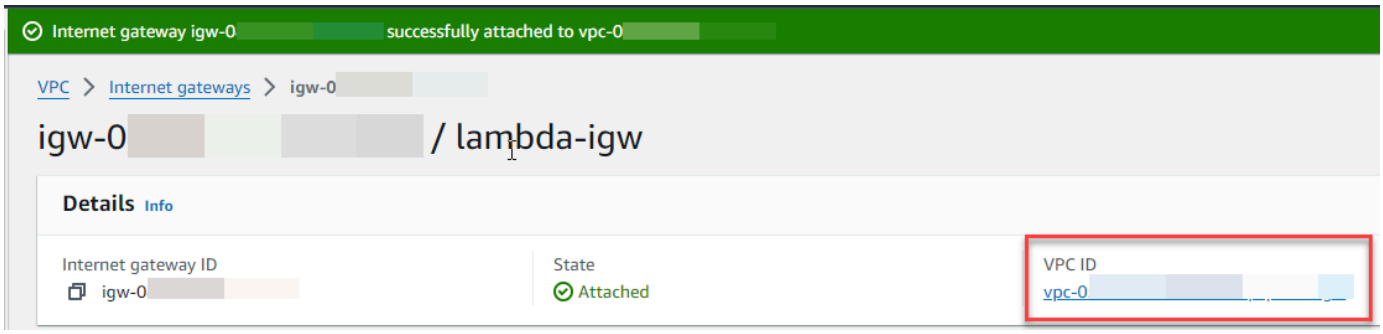
Gehen Sie wie folgt vor, um ein Internet-Gateway zu erstellen, es an Ihre VPC anzuhängen und es der Routing-Tabelle Ihres öffentlichen Subnetzes hinzuzufügen.

So erstellen Sie ein Internet-Gateway

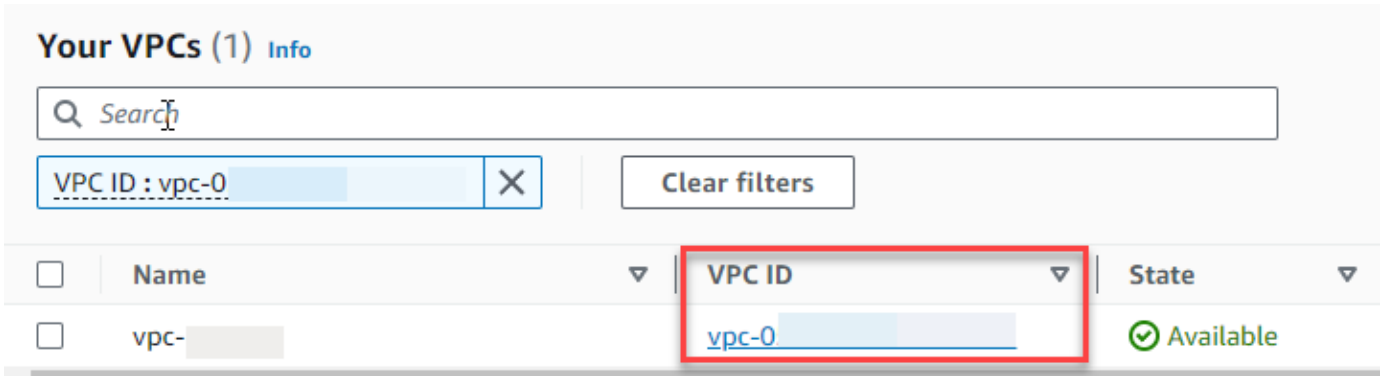
1. Öffnen Sie die Amazon VPC-Konsole unter https://console.aws.amazon.com/vpc/.
2. Wählen Sie im Navigationsbereich Internet Gateways (Internet-Gateways) aus.
3. Wählen Sie Create internet gateway (Internet-Gateway erstellen) aus.
4. (Optional) Geben Sie einen Namen für Ihr Internet-Gateway ein.
5. (Optional) Um ein Tag hinzuzufügen, wählen Sie Add new tag (Neuen Tag hinzufügen) aus und geben Sie den Schlüssel und den Wert für den Tag ein.
6. Wählen Sie Create internet gateway (Internet-Gateway erstellen) aus.
7. Wählen Sie im Banner oben auf dem Bildschirm die Option An eine VPC anhängen, wählen Sie eine verfügbare VPC aus und wählen Sie dann Internet-Gateway anhängen.



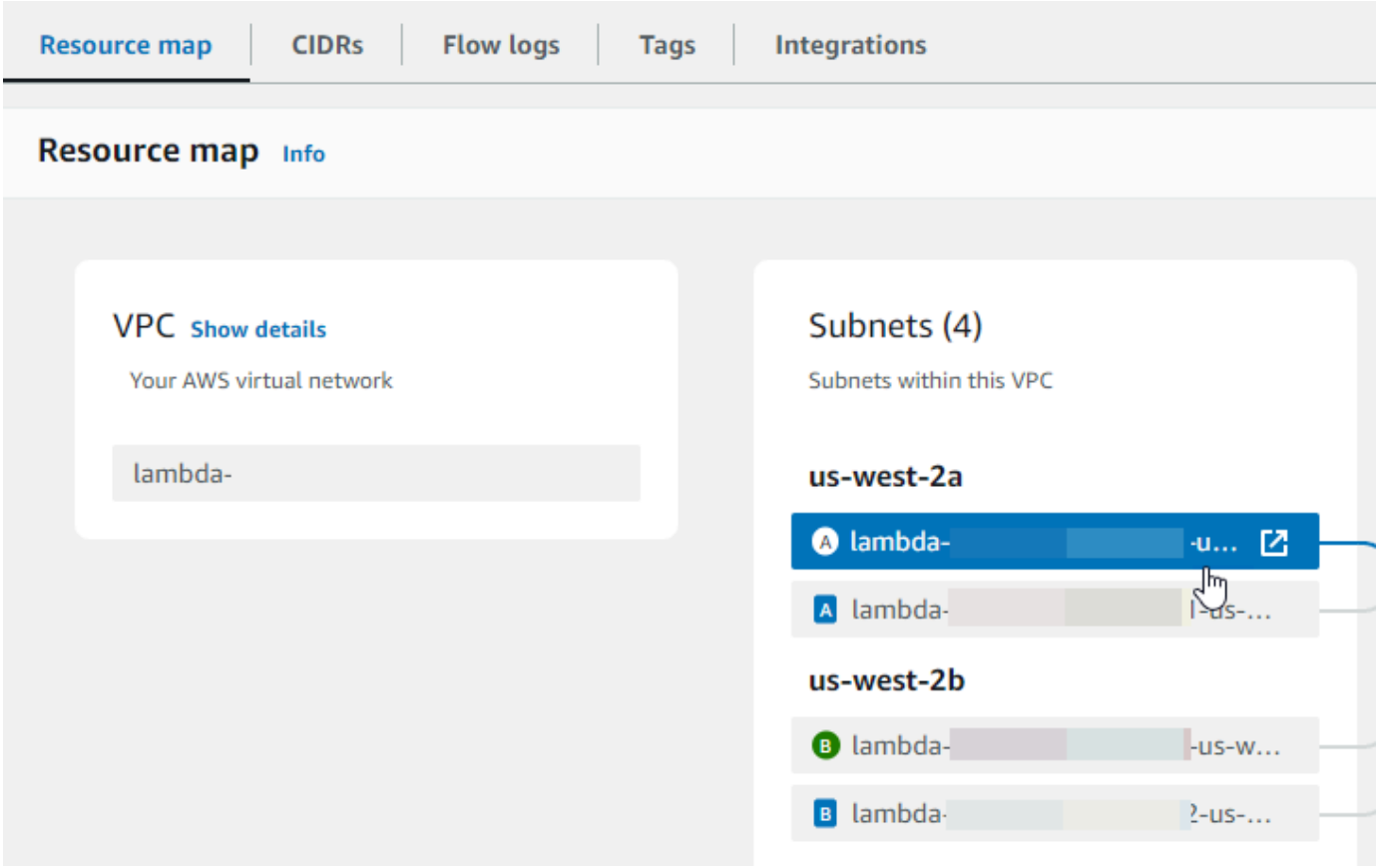
8. Wählen Sie die VPC ID aus.



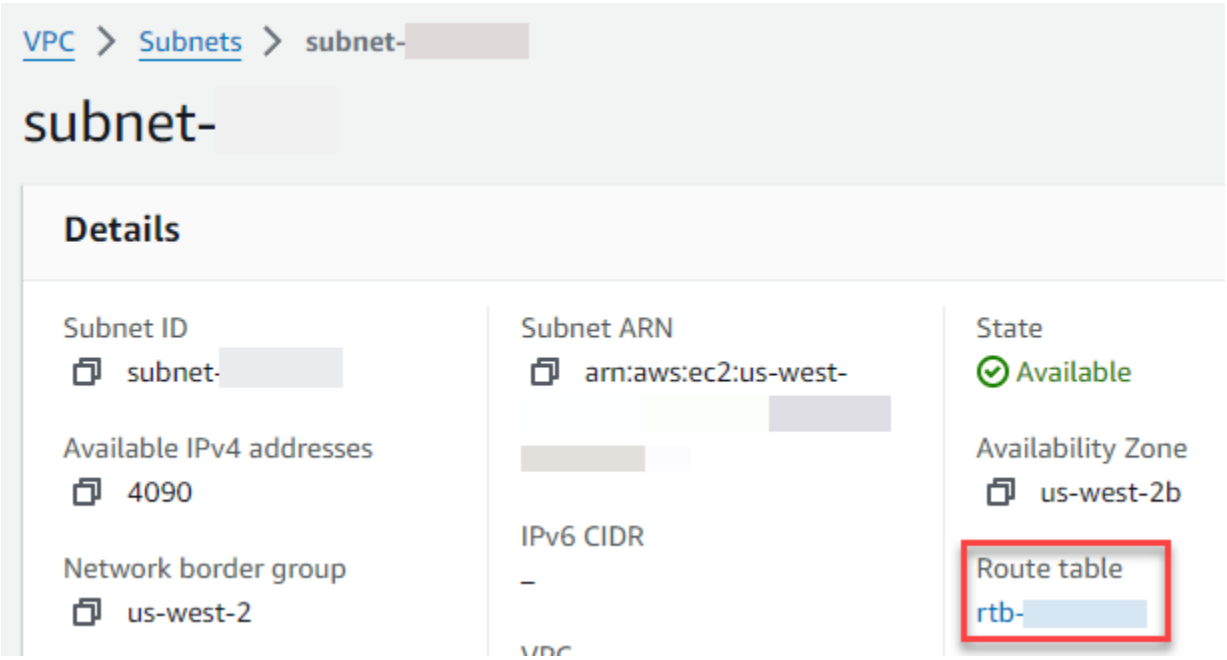
9. Wählen Sie die VPC ID, um die VPC-Detailseite zu öffnen.



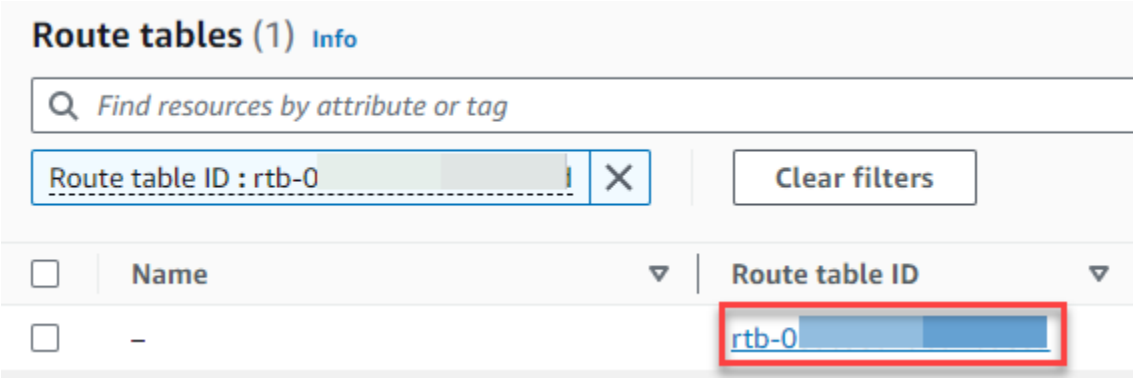
10. Blättern Sie nach unten zum Abschnitt Ressourcenzuordnung und wählen Sie dann ein Subnetz aus. Die Subnetzdetails werden auf der neuen Registerkarte angezeigt.



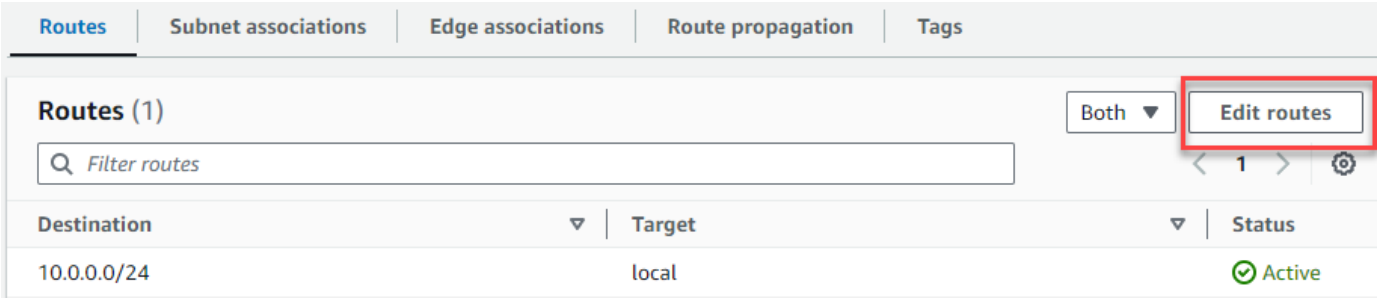
11. Wählen Sie den Link unter Routing-Tabelle aus.



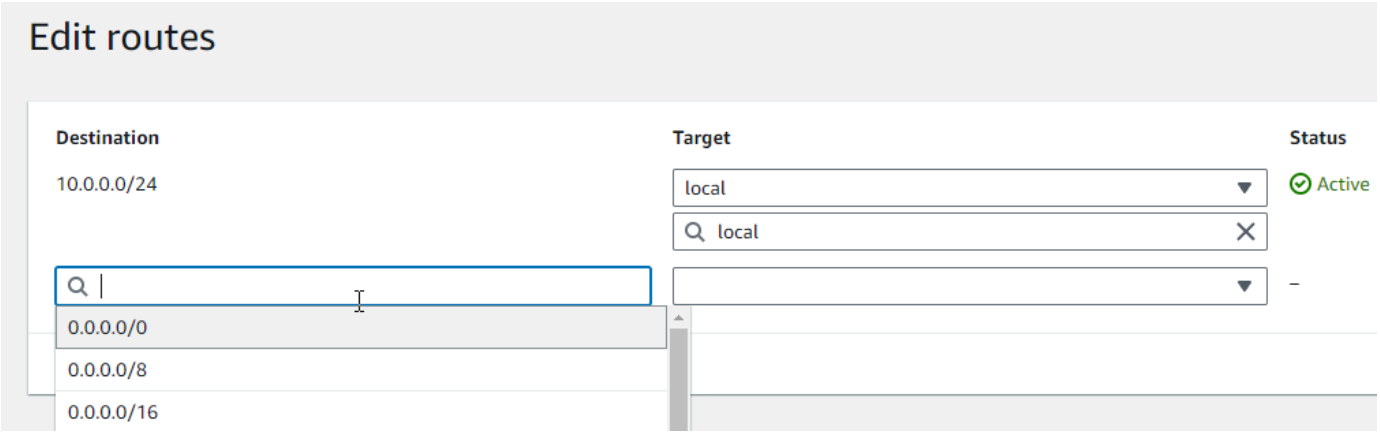
12. Wählen Sie die Routing-Tabellen-ID, um die Detailseite der Routing-Tabelle zu öffnen.



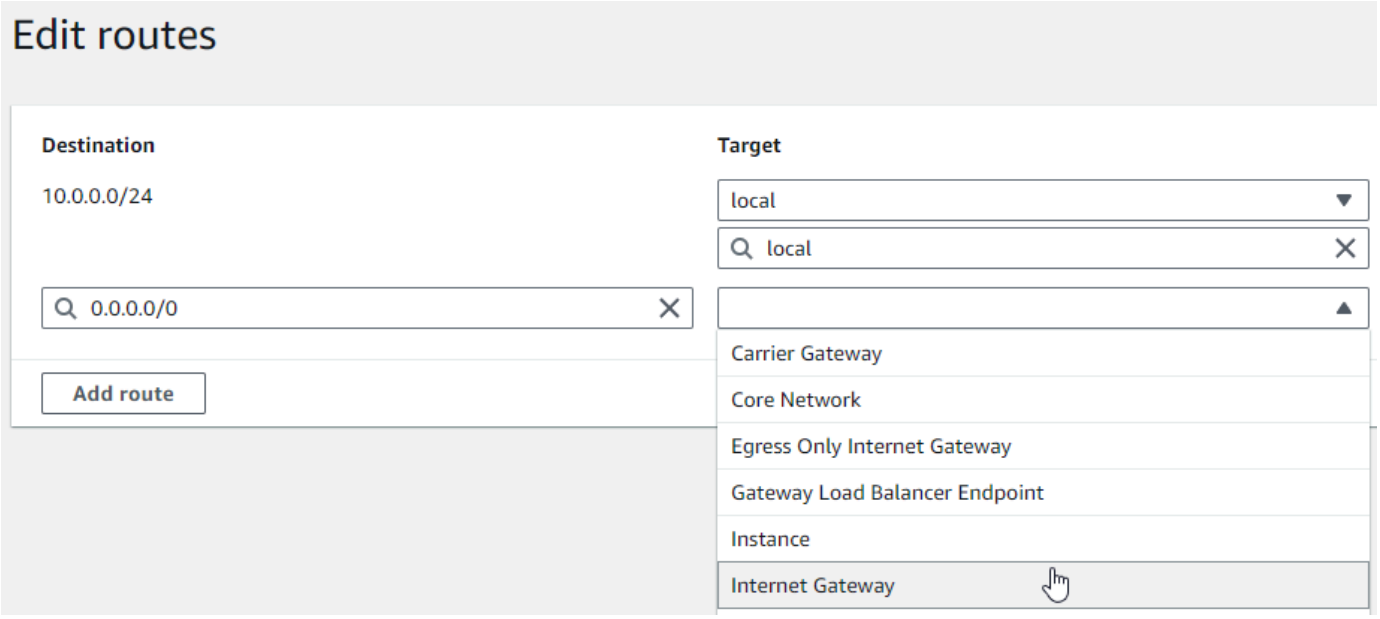
13. Wählen Sie unter Routes (Routen) Routen bearbeiten aus.



14. Wählen Sie Route hinzufügen und geben Sie sie dann `0.0.0.0/0` in das Feld Ziel ein.



15. Wählen Sie für Target Internet-Gateway und dann das Internet-Gateway aus, das Sie zuvor erstellt haben. Wenn Ihr Subnetz einen IPv6-CIDR-Block hat, müssen Sie auch eine Route für `::/0` zum gleichen Internet-Gateway hinzufügen.



16. Wählen Sie Änderungen speichern aus.

Erstellen eines NAT-Gateways

Gehen Sie wie folgt vor, um ein NAT-Gateway zu erstellen, es einem öffentlichen Subnetz zuzuordnen und es dann der Routing-Tabelle Ihres privaten Subnetzes hinzuzufügen.

Um ein NAT-Gateway zu erstellen und es einem öffentlichen Subnetz zuzuordnen

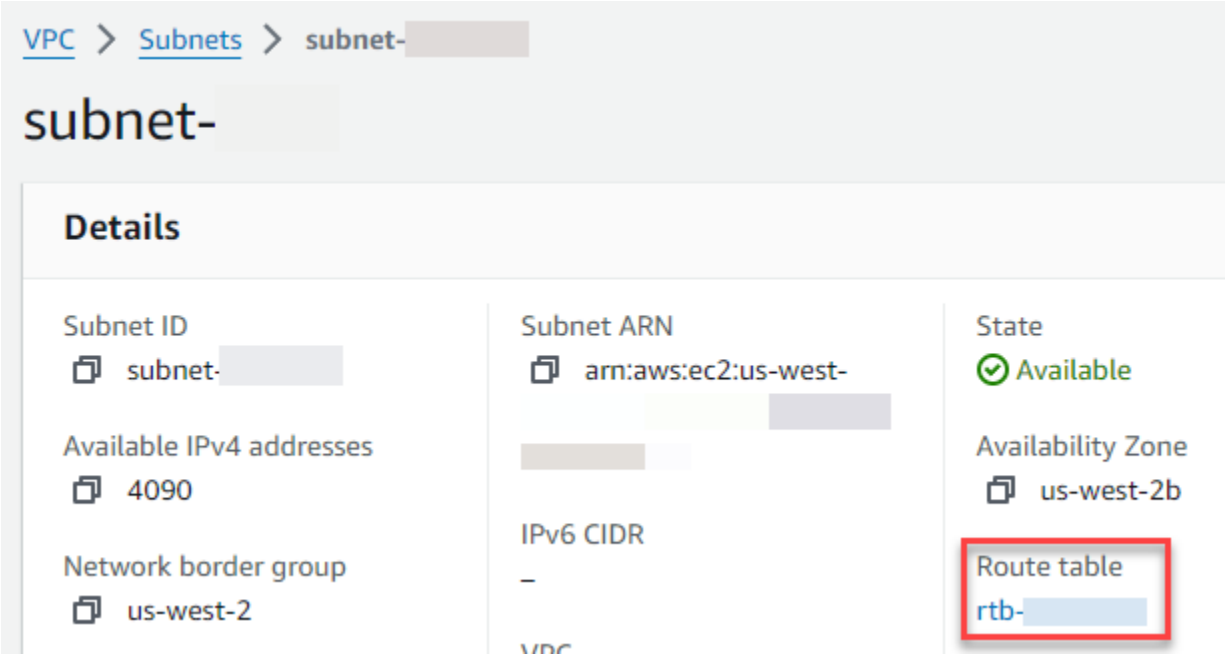
1. Klicken Sie im Navigationsbereich auf NAT-Gateways.
2. Wählen Sie NAT-Gateway erstellen aus.
3. (Optional) Geben Sie einen Namen für Ihr NAT-Gateway ein.
4. Wählen Sie unter Subnetz ein öffentliches Subnetz in Ihrer VPC. (Ein öffentliches Subnetz ist ein Subnetz, das eine direkte Route zu einem Internet-Gateway in seiner Routing-Tabelle hat).

> **Note**
>
> NAT-Gateways sind einem öffentlichen Subnetz zugeordnet, der Eintrag in der Routing-Tabelle befindet sich jedoch im privaten Subnetz.

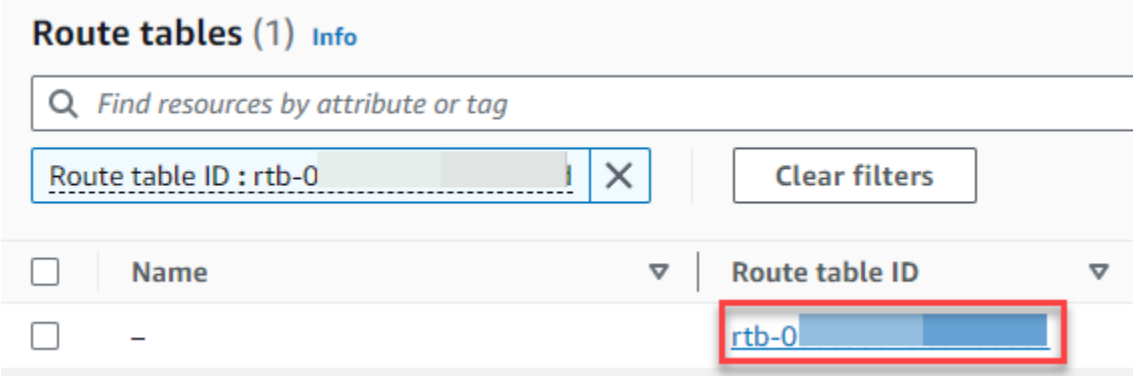
5. Wählen Sie als Elastische IP-Zuweisungs-ID eine elastische IP-Adresse oder Elastische IP zuweisen aus.
6. Wählen Sie NAT-Gateway erstellen aus.

So fügen Sie eine Route zum NAT-Gateway in der Routentabelle des privaten Subnetzes hinzu

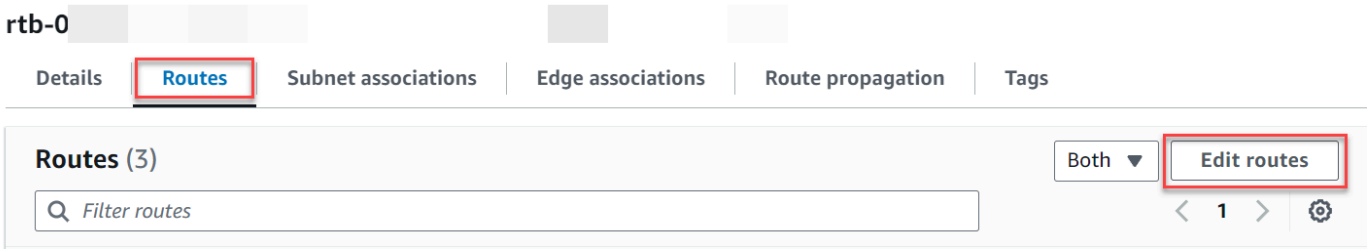
1. Wählen Sie im Navigationsbereich Subnetze aus.
2. Wählen Sie ein privates Subnetz in Ihrer VPC. (Ein privates Subnetz ist ein Subnetz, das keine Route zu einem Internet-Gateway in seiner Routing-Tabelle hat).
3. Wählen Sie den Link unter Routing-Tabelle aus.



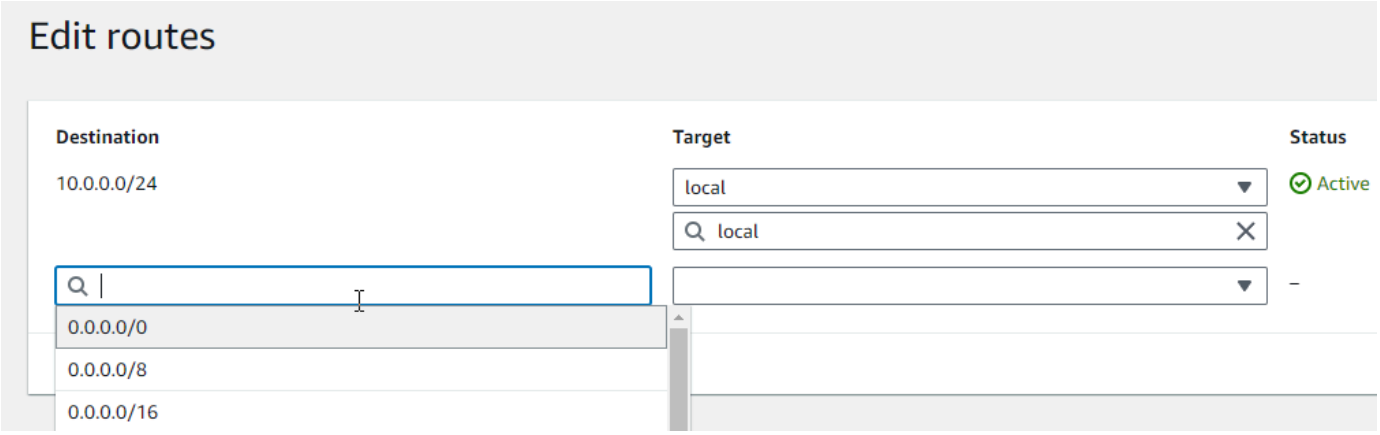
4. Wählen Sie die Routing-Tabellen-ID, um die Detailseite der Routing-Tabelle zu öffnen.



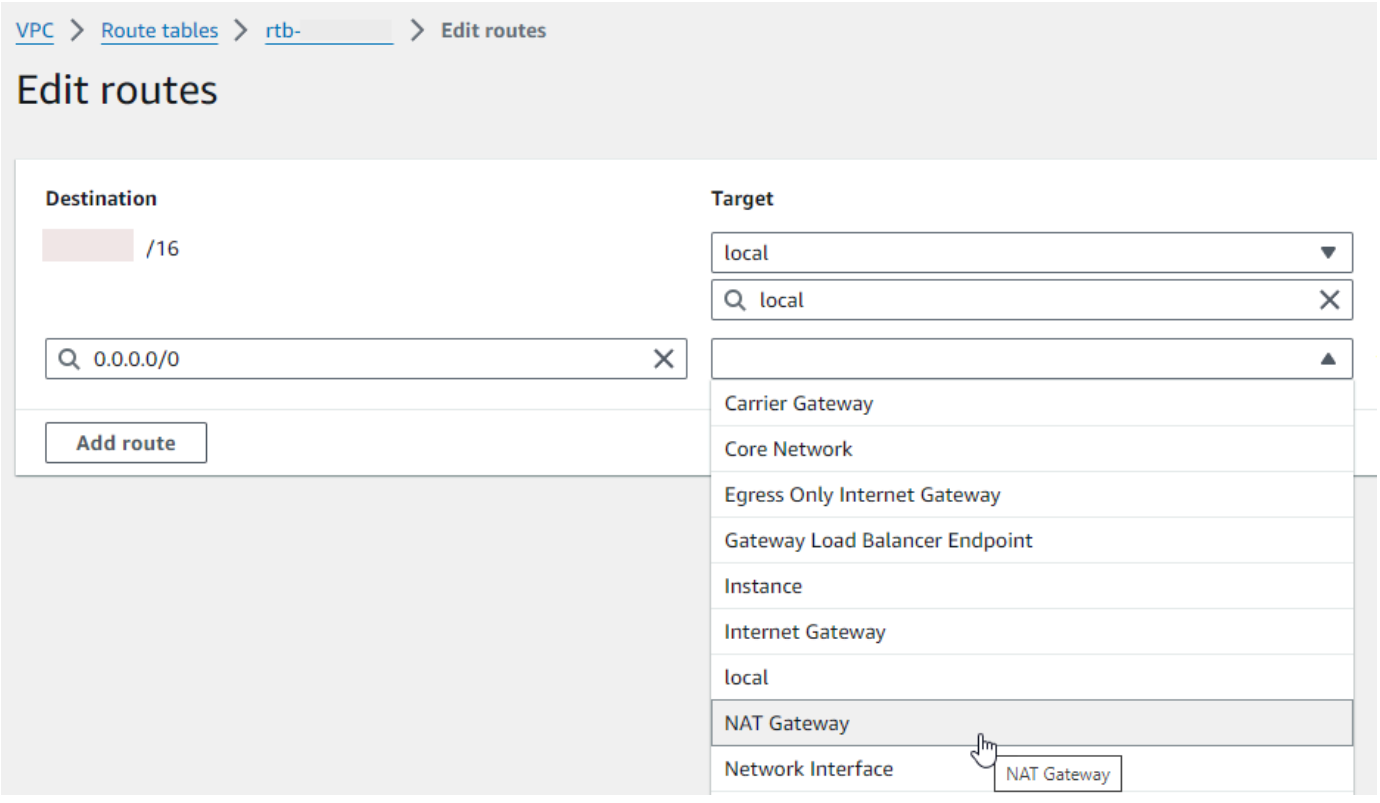
5. Scrollen Sie nach unten und wählen Sie die Registerkarte Routes (Routen) auf Edit routes (Routen bearbeiten).



6. Wählen Sie Route hinzufügen und geben Sie sie dann `0.0.0.0/0` in das Feld Ziel ein.



7. Wählen Sie für Target NAT-Gateway und dann das NAT-Gateway aus, das Sie zuvor erstellt haben.



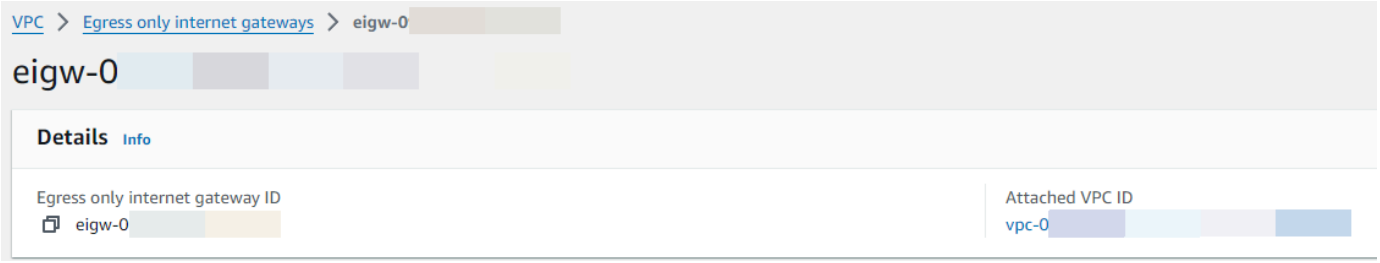
8. Wählen Sie Änderungen speichern aus.

Erstellen eines Internet-Gateways nur für ausgehenden Verkehr (nur IPv6)

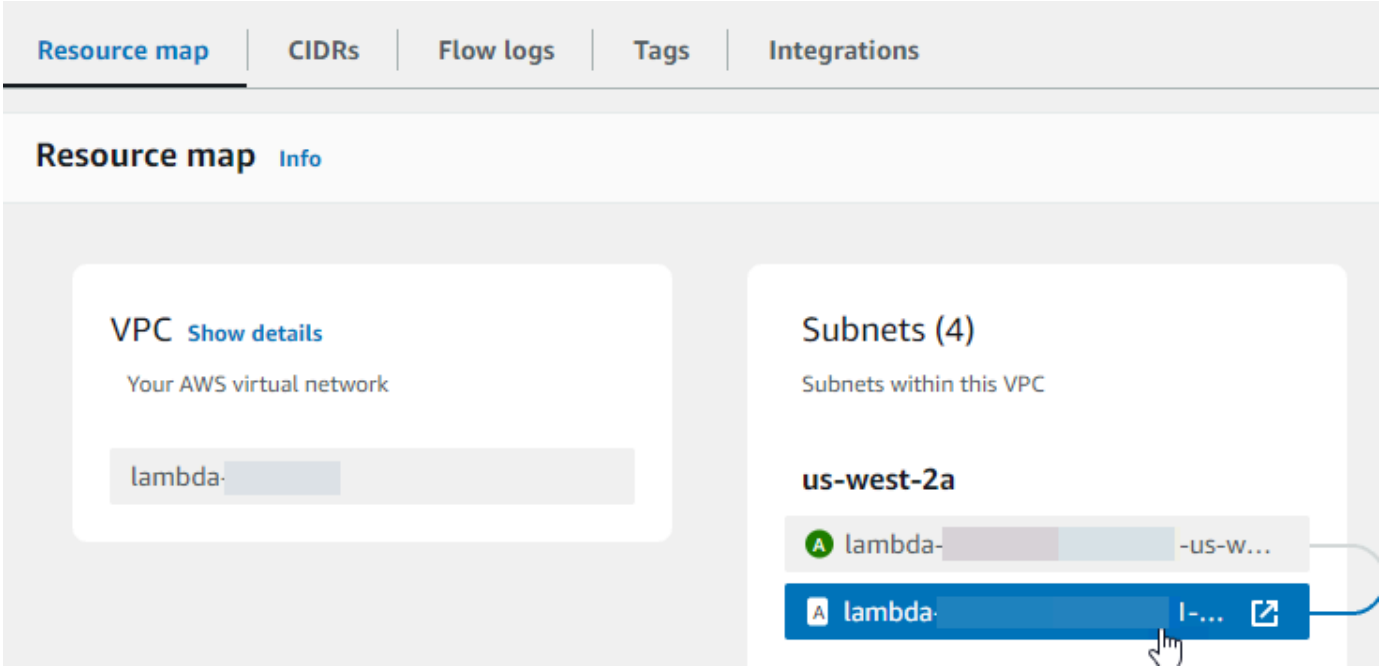
Führen Sie die folgenden Schritte aus, um ein reines Egress-Internet-Gateway zu erstellen und es zur Routing-Tabelle Ihres privaten Subnetzes hinzuzufügen.

So erstellen Sie ein Internet-Gateway nur für ausgehenden Verkehr

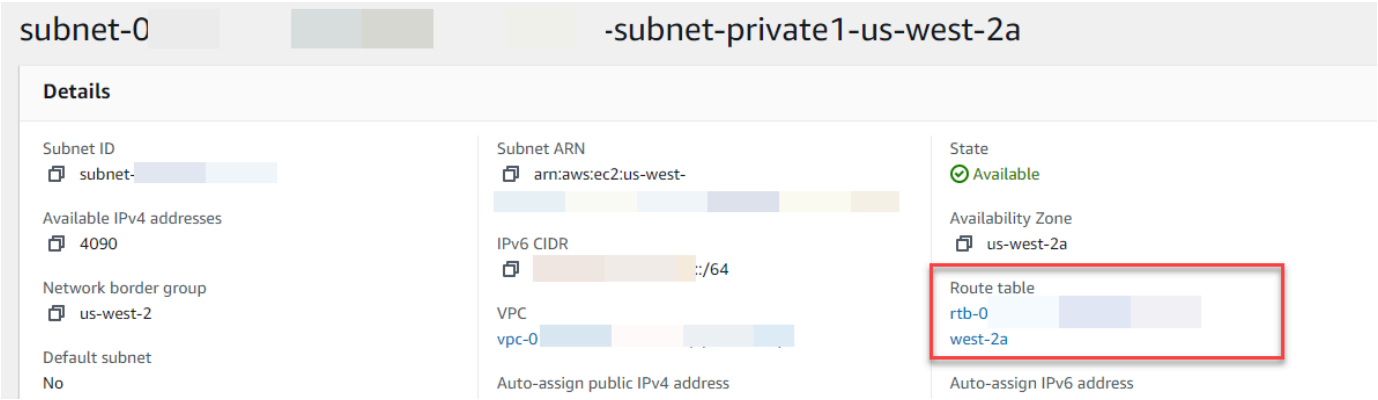
1. Wählen Sie im Navigationsbereich Internet-Gateways für ausgehenden Datenverkehr.
2. Klicken Sie auf Internet-Gateway für ausgehenden Datenverkehr erstellen.
3. (Optional) Geben Sie einen Name ein.
4. Wählen Sie die VPC aus, in der Sie das Internet-Gateway für ausgehenden Verkehr erstellen möchten.
5. Klicken Sie auf Internet-Gateway für ausgehenden Datenverkehr erstellen.
6. Wählen Sie den Link unter Angehängte VPC-ID aus.



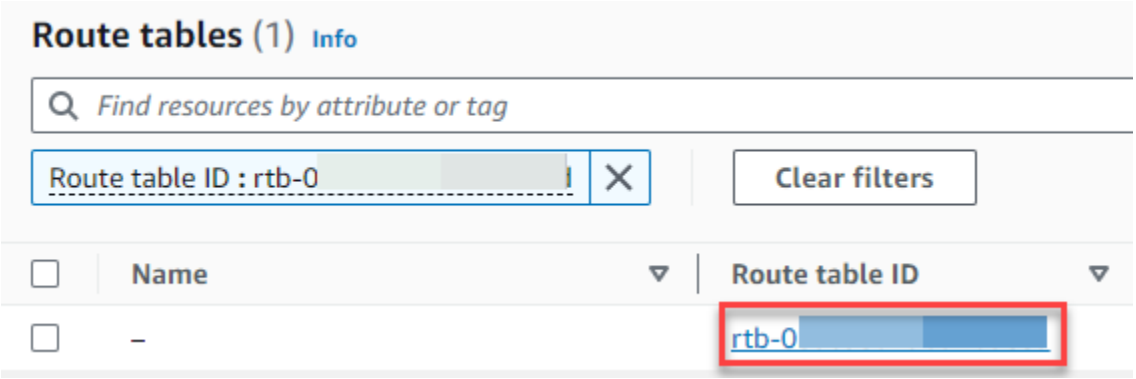
7. Wählen Sie den Link unter VPC ID, um die VPC-Detailseite zu öffnen.
8. Blättern Sie nach unten zum Abschnitt Ressourcenzuordnung und wählen Sie dann ein privates Subnetz aus. (Ein privates Subnetz ist ein Subnetz, das keine Route zu einem Internet-Gateway in seiner Routing-Tabelle hat). Die Subnetzdetails werden auf der neuen Registerkarte angezeigt.



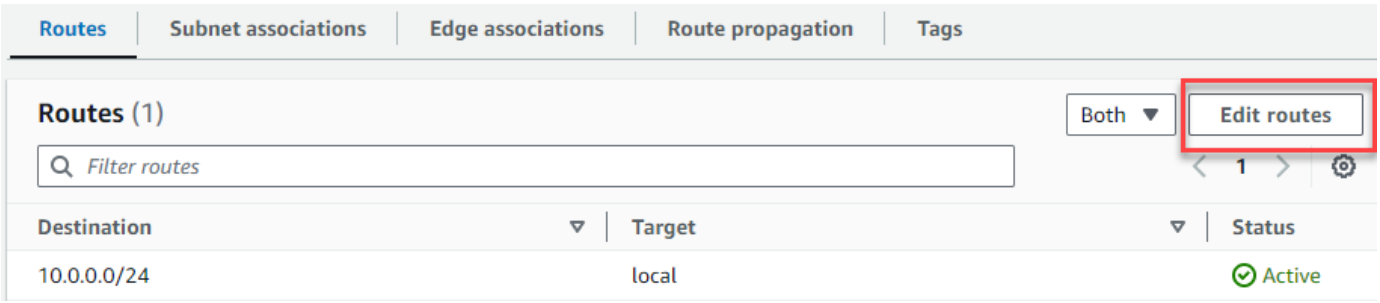
9. Wählen Sie den Link unter Routing-Tabelle aus.



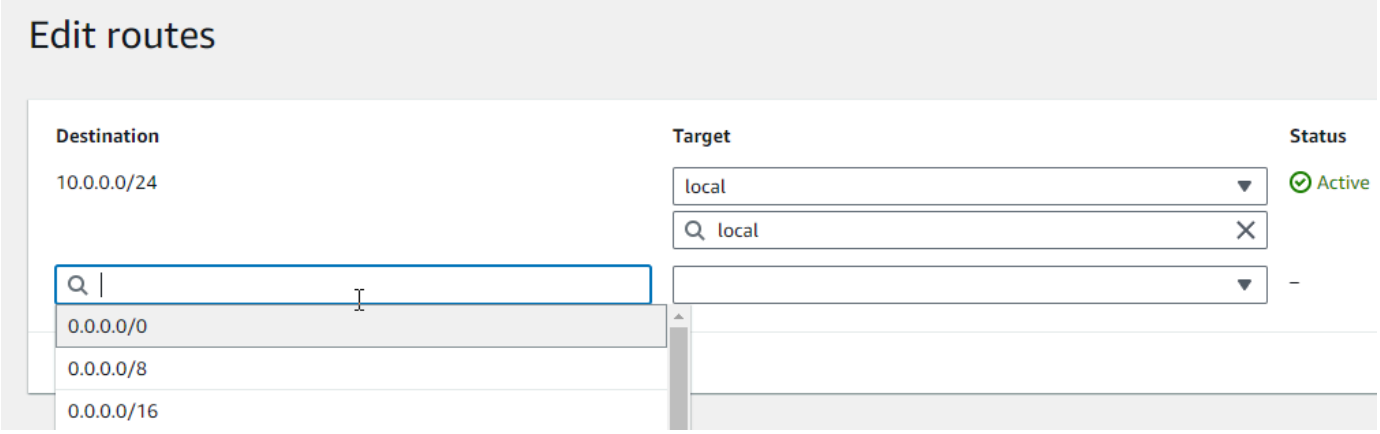
10. Wählen Sie die Routing-Tabellen-ID, um die Detailseite der Routing-Tabelle zu öffnen.



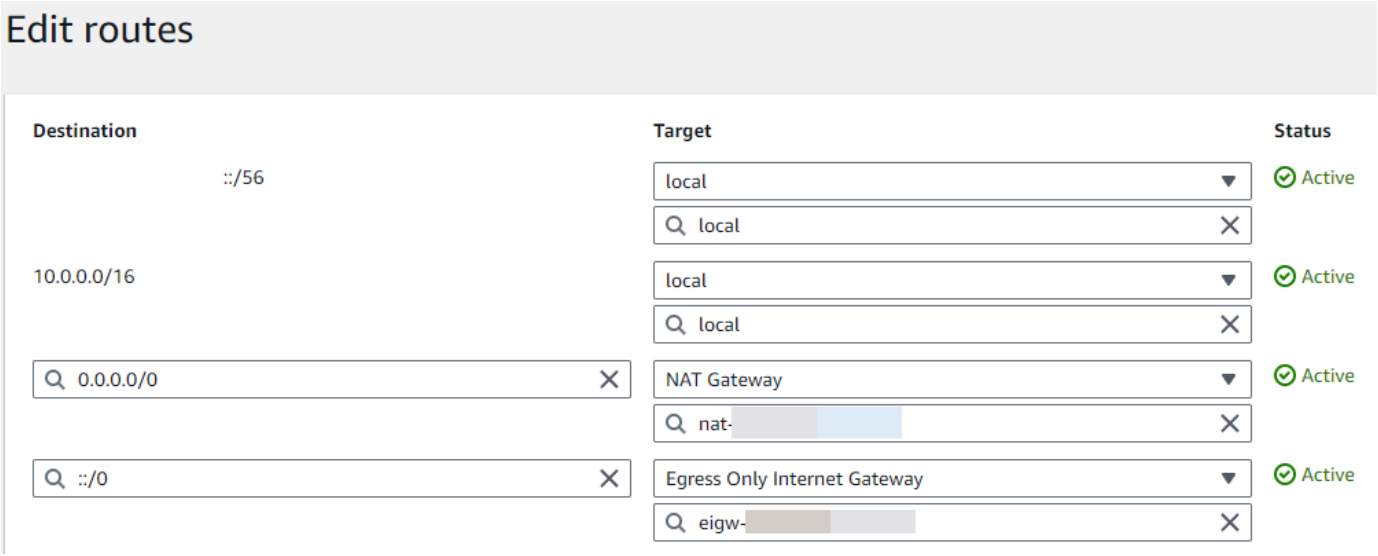
11. Wählen Sie unter Routes (Routen) Routen bearbeiten aus.



12. Wählen Sie Route hinzufügen und geben Sie sie dann `::/0` in das Feld Ziel ein.



13. Wählen Sie für Target Nur Internet-Gateway für den Ausgang und dann das Internet-Gateway aus, das Sie zuvor erstellt haben.



14. Wählen Sie Änderungen speichern aus.

Konfigurieren der Lambda-Funktion

So konfigurieren Sie eine VPC bei der Funktionserstellung

1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.
2. Wählen Sie Create function (Funktion erstellen).
3. Geben Sie unter Basic information (Grundlegende Informationen) bei Function name (Funktionsname) einen Namen für Ihre Funktion ein.
4. Erweiterten Sie Advanced settings (Erweiterte Einstellungen).
5. Wählen Sie VPC aktivieren und wählen Sie dann eine VPC aus.
6. (Optional) Um ausgehenden IPv6-Verkehr zuzulassen, wählen Sie Allow IPv6 traffic for dual-stack subnets (IPv6-Verkehr für Dual-Stack-Subnetze zulassen) aus.
7. Wählen Sie für Subnetze alle privaten Subnetze aus. Die privaten Subnetze können über die NAT-Gateway auf das Internet zugreifen. Durch die Verbindung einer Funktion mit einem öffentlichen Subnetz erhält sie keinen Internetzugang.

   > **Note**
   >
   > Wenn Sie Allow IPv6 traffic for dual-stack subnets (IPv6-Verkehr für Dual-Stack-Subnetze zulassen) ausgewählt haben, müssen alle ausgewählten Subnetze einen IPv4-CIDR-Block und einen IPv6-CIDR-Block besitzen.

8. Wählen Sie für Sicherheitsgruppen eine Sicherheitsgruppe aus, die ausgehenden Datenverkehr zulässt.
9. Wählen Sie Funktion erstellen.

Lambda erstellt automatisch eine Ausführungsrolle mit der AWS-verwalteten Richtlinie AWSLambdaVPCAccessExecutionRole. Die Berechtigungen in dieser Richtlinie sind nur zum Erstellen von Elastic Network-Schnittstellen für die VPC-Konfiguration und nicht zum Aufrufen Ihrer Funktion erforderlich. Um Berechtigungen mit den geringsten Rechten anzuwenden, können Sie die Richtlinie AWSLambdaVPCAccessExecutionRole aus Ihrer Ausführungsrolle entfernen, nachdem Sie die Funktion und die VPC-Konfiguration erstellt haben. Weitere Informationen finden Sie unter Erforderliche IAM-Berechtigungen.

So konfigurieren Sie eine VPC für eine vorhandene Funktion

Um einer vorhandenen Funktion eine VPC-Konfiguration hinzuzufügen, muss die Ausführungsrolle der Funktion über die Berechtigung verfügen, elastische Netzwerkschnittstellen zu erstellen und zu verwalten. Die verwaltete Richtlinie AWSLambdaVPCAccessExecutionRole AWS enthält die erforderlichen Berechtigungen. Um Berechtigungen mit den geringsten Rechten anzuwenden, können Sie die Richtlinie AWSLambdaVPCAccessExecutionRole aus Ihrer Ausführungsrolle entfernen, nachdem Sie die VPC-Konfiguration erstellt haben.

1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.
2. Wählen Sie eine Funktion aus.
3. Wählen Sie die Registerkarte Konfiguration und anschließend VPC aus.
4. Wählen Sie unter VPC die Option Edit (Bearbeiten) aus.
5. Wählen Sie die VPC aus.
6. (Optional) Um ausgehenden IPv6-Verkehr zuzulassen, wählen Sie Allow IPv6 traffic for dual-stack subnets (IPv6-Verkehr für Dual-Stack-Subnetze zulassen) aus.
7. Wählen Sie für Subnetze alle privaten Subnetze aus. Die privaten Subnetze können über die NAT-Gateway auf das Internet zugreifen. Durch die Verbindung einer Funktion mit einem öffentlichen Subnetz erhält sie keinen Internetzugang.

   > **Note**
   >
   > Wenn Sie Allow IPv6 traffic for dual-stack subnets (IPv6-Verkehr für Dual-Stack-Subnetze zulassen) ausgewählt haben, müssen alle ausgewählten Subnetze einen IPv4-CIDR-Block und einen IPv6-CIDR-Block besitzen.

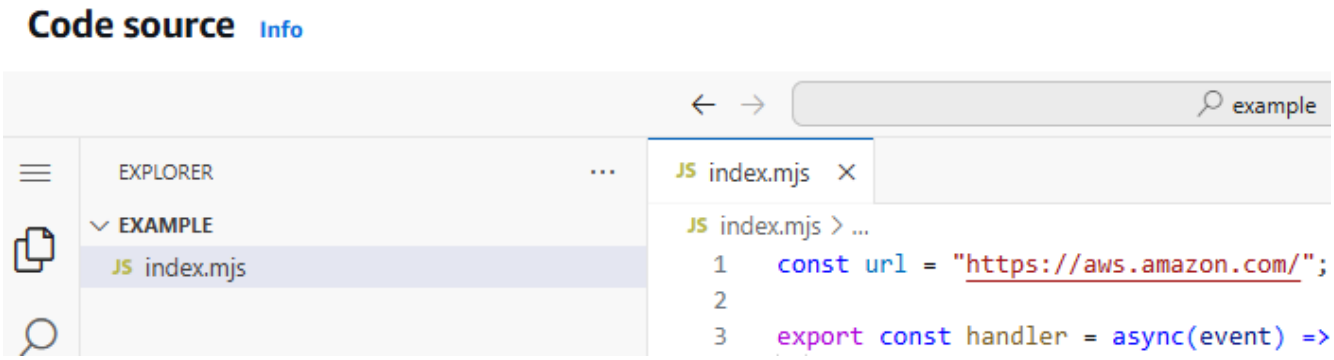
8. Wählen Sie für Sicherheitsgruppen eine Sicherheitsgruppe aus, die ausgehenden Datenverkehr zulässt.
9. Wählen Sie Speichern.

Testen der Funktion

Verwenden Sie den folgenden Beispielcode, um zu bestätigen, dass Ihre VPC-verbundene Funktion das öffentliche Internet erreichen kann. Bei Erfolg gibt der Code einen `200`-Statuscode zurück. Ist dies nicht der Fall, wird die Funktion abgebrochen.

Node.js

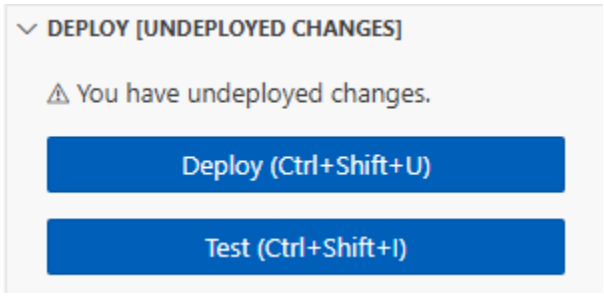
1. Fügen Sie im Bereich Codequelle der Lambda-Konsole den folgenden Code in die Datei index.mjs ein. Die Funktion stellt eine HTTP-GET-Anfrage an einen öffentlichen Endpunkt und gibt den HTTP-Antwortcode zurück, um zu prüfen, ob die Funktion Zugang zum öffentlichen Internet hat.



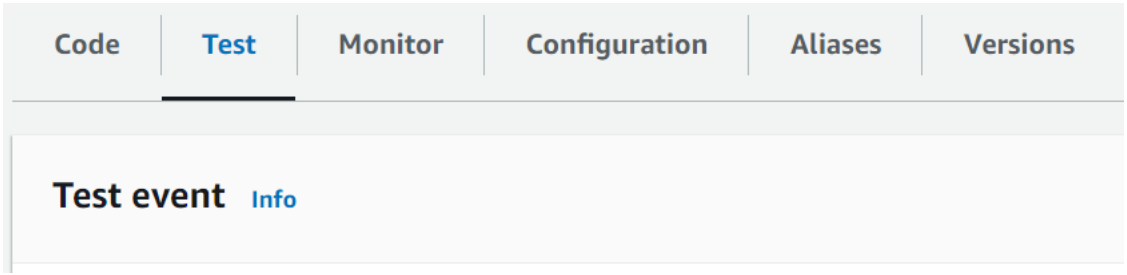
Example – HTTP-Anforderung mit async/await

```
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;
    }
};
```

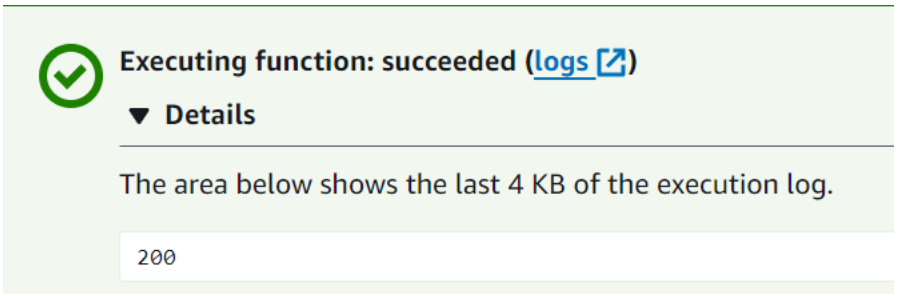
2. Wählen Sie im Abschnitt BEREITSTELLEN die Option Bereitstellen aus, um den Code Ihrer Funktion zu aktualisieren:



3. Wählen Sie die Registerkarte Test.



4. Wählen Sie Test aus.
5. Die Funktion gibt einen 200-Statuscode zurück. Das bedeutet, dass die Funktion über einen ausgehenden Internetzugang verfügt.

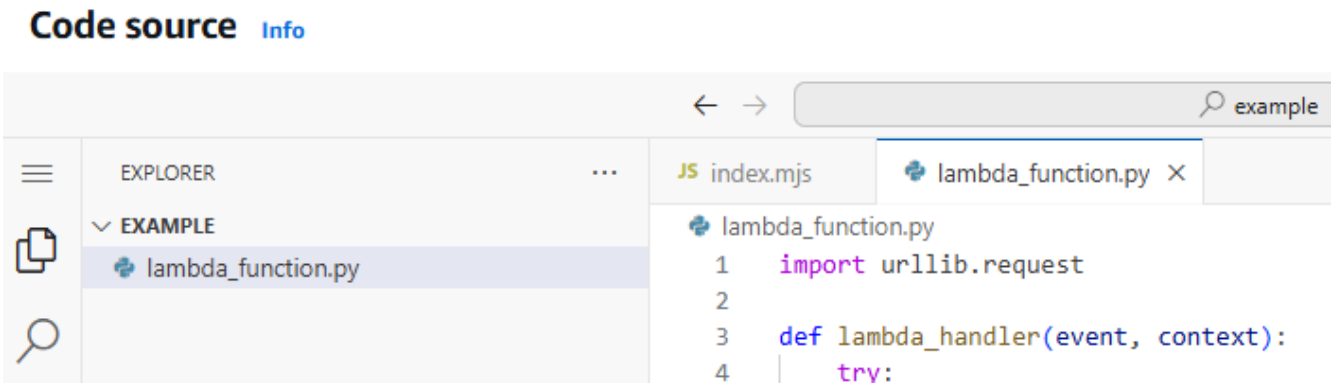


Wenn die Funktion das öffentliche Internet nicht erreichen kann, erhalten Sie eine Fehlermeldung wie die folgende:

```
{
  "errorMessage": "2024-04-11T17:22:20.857Z abe12jlc-640a-8157-0249-9be825c2y110
 Task timed out after 3.01 seconds"
}
```

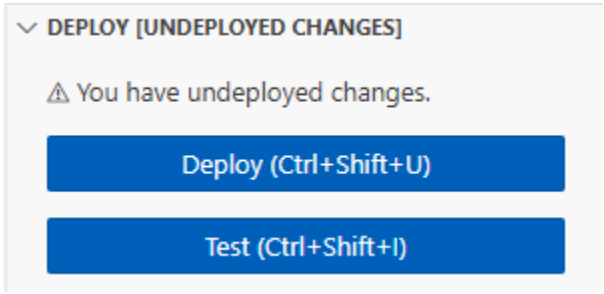
Python

1. Fügen Sie im Bereich Codequelle der Lambda-Konsole den folgenden Code in die Datei lambda_function.py ein. Die Funktion stellt eine HTTP-GET-Anfrage an einen öffentlichen Endpunkt und gibt den HTTP-Antwortcode zurück, um zu prüfen, ob die Funktion Zugang zum öffentlichen Internet hat.

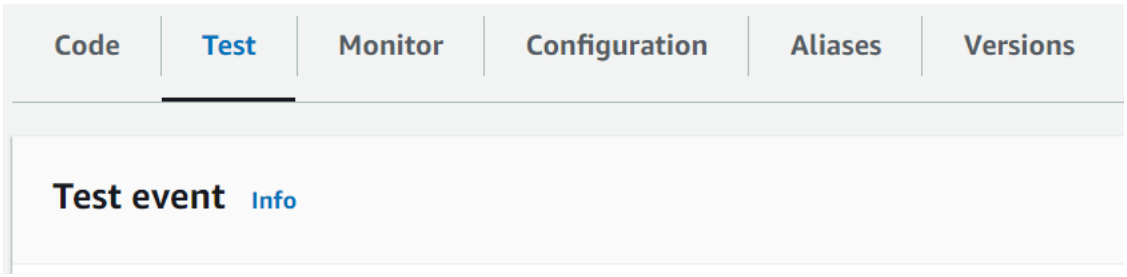



```
import urllib.request

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. Wählen Sie im Abschnitt BEREITSTELLEN die Option Bereitstellen aus, um den Code Ihrer Funktion zu aktualisieren:

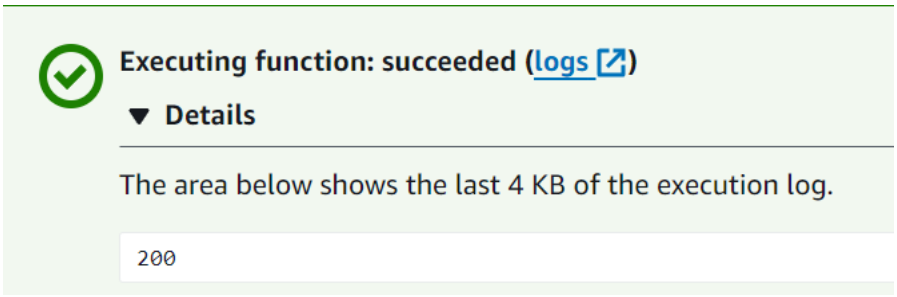


3. Wählen Sie die Registerkarte Test.



4. Wählen Sie Test aus.

5. Die Funktion gibt einen 200-Statuscode zurück. Das bedeutet, dass die Funktion über einen ausgehenden Internetzugang verfügt.



Wenn die Funktion das öffentliche Internet nicht erreichen kann, erhalten Sie eine Fehlermeldung wie die folgende:

```
{
  "errorMessage": "2024-04-11T17:22:20.857Z abe12jlc-640a-8157-0249-9be825c2y110
 Task timed out after 3.01 seconds"
}
```

Verbinden von VPC-Endpunkten von eingehenden Schnittstellen für Lambda

Wenn Sie Amazon Virtual Private Cloud (Amazon VPC) zum Hosten Ihrer AWS-Ressourcen verwenden, können Sie eine Verbindung zwischen Ihrer VPC und Lambda herstellen. Sie können mit dieser Verbindung Ihre Lambda-Funktion aufrufen, ohne das öffentliche Internet überqueren.

Um eine private Verbindung zwischen Ihrer VPC und Lambda herzustellen, erstellen Sie einen Schnittstellen-VPC-Endpunkt. Schnittstellenendpunkte werden von AWS PrivateLink betrieben, sodass Sie privat ohne Internet-Gateway, NAT-Gerät, VPN-Verbindung oder AWS Direct Connect-Verbindung auf Lambda-APIs zugreifen können. Die Instances in Ihrer VPC benötigen für die Kommunikation mit Lambda-APIs keine öffentlichen IP-Adressen. Datenverkehr zwischen Ihrer VPC und Lambda verlässt das AWS-Netzwerk nicht.

Jeder Schnittstellenendpunkt wird durch eine oder mehrere Elastic Network-Schnittstellen in Ihren Subnetzen dargestellt. Eine Netzwerkschnittstelle stellt eine private IP-Adresse bereit, die als Einstiegspunkt für den Datenverkehr zu Lambda dient.

Abschnitte

- Überlegungen für Lambda-Schnittstellenendpunkte
- Erstellen eines Schnittstellenendpunkts für Lambda
- Erstellen einer Richtlinie des Schnittstellenendpunkts für Lambda

Überlegungen für Lambda-Schnittstellenendpunkte

Bevor Sie einen Schnittstellenendpunkt für Lambda einrichten, lesen Sie unbedingt die Eigenschaften und Einschränkungen des Schnittstellenendpunkts im Amazon-VPC-Benutzerhandbuch.

Sie können alle Lambda-API-Operationen von Ihrer VPC aus aufrufen. Sie können die Lambda-Funktion beispielsweise aufrufen, indem Sie die `Invoke`-API aus Ihrer VPC aufrufen. Die vollständige Liste der Lambda-APIs finden Sie unter Aktionen in der Lambda-API-Referenz.

`use1-az3` ist eine Region mit begrenzter Kapazität für Lambda-VPC-Funktionen. Sie sollten in dieser Availability Zone keine Subnetze für Ihre Lambda-Funktionen verwenden, da dies im Falle eines Ausfalls zu einer verringerten zonalen Redundanz führen kann.

Keep-Alive für persistente Verbindungen

Lambda löscht Leerlaufverbindungen im Laufe der Zeit. Daher müssen Sie eine Keep-Alive-Direktive verwenden, um dauerhafte Verbindungen zu pflegen. Der Versuch, eine Leerlaufverbindung beim Aufruf einer Funktion wiederzuverwenden, führt zu einem Verbindungsfehler. Um Ihre persistente Verbindung aufrechtzuerhalten, verwenden Sie die Keep-Alive-Direktive, die Ihrer Laufzeit zugeordnet ist. Ein Beispiel finden Sie unter Wiederverwenden von Verbindungen mit Keep-Alive in Node.js im AWS SDK für JavaScript-Entwicklerhandbuch.

Überlegungen zur Abrechnung

Es fallen keine zusätzlichen Kosten für den Zugriff auf eine Lambda-Funktion über einen Schnittstellen-Endpunkt an. Weitere Informationen über Lambda-Preise finden Sie unter AWS Lambda-Preise.

Die Standardpreise für AWS PrivateLink gelten für Schnittstellenendpunkte für Lambda. IhremAWS -Konto wird jede Stunde in Rechnung gestellt, die ein Schnittstellenendpunkt in jeder Availability Zone bereitgestellt wird und für Daten, die über den Schnittstellenendpunkt verarbeitet werden. Weitere Informationen zu Preisen für Schnittstellenendpunkte finden Sie unter AWS PrivateLink-Preise.

Überlegungen zu VPC Peering

Sie können andere VPCs mit Schnittstellenendpunkten über VPC-Peering mit der VPC verbinden. VPC-Peering ist eine Netzwerkverbindung zwischen zwei VPCs. Sie können eine VPC-Peering-Verbindung zwischen Ihren eigenen beiden VPCs oder mit einer VPC in einem anderen AWS-Konto herstellen. Die VPCs können sich auch in zwei verschiedenen AWS-Regionen befinden.

Der Datenverkehr zwischen über Peering verbundenen VPCs bleibt im AWS-Netzwerk und wird nicht über das öffentliche Internet übertragen. Sobald VPCs per Peering verbunden sind, können Ressourcen wie Amazon-Elastic-Compute-Cloud-(Amazon-EC2)-Instances, Amazon-Relational-Database-Service-(Amazon-RDS)-Instances oder VPC-fähige Lambda-Funktionen in beiden VPCs über Schnittstellenendpunkte, die in einem der VPCs erstellt wurden, auf die Lambda-API zugreifen.

Erstellen eines Schnittstellenendpunkts für Lambda

Sie können einen Schnittstellenendpunkt für Lambda entweder über die Amazon-VPC-Konsole oder die AWS Command Line Interface (AWS CLI) erstellen. Weitere Informationen finden Sie unter Erstellung eines Schnittstellenendpunkts im Amazon VPC Benutzerhandbuch.

So erstellen Sie einen Schnittstellenendpunkt für Lambda (Konsole)

1. Öffnen Sie die Seite Endpunkte der Amazon-VPC-Konsole.
2. Klicken Sie auf Create Endpoint.
3. Stellen Sie sicher, dass für Service-Kategorie die Option AWS-Services ausgewählt ist.
4. Wählen Sie als Service Name (Servicename) com.amazonaws.***region***.lambda aus. Stellen Sie sicher, dass als Typ die Option Schnittstelle gewählt ist.
5. Auswählen von VPC und Subnetzen
6. Um ein privates DNS für den Schnittstellenendpunkt zu aktivieren, aktivieren Sie das Kontrollkästchen für Name eines DNS aktivieren. Wir empfehlen die Aktivierung privater DNS-Namen für Ihren VPC-Endpunkt für AWS-Services. Dadurch wird sichergestellt, dass Anfragen, die die Endpunkte des öffentlichen Dienstes verwenden, z. B. Anfragen, die über ein AWS-SDK gestellt wurden, an Ihren VPC-Endpunkt weitergeleitet werden.
7. Wählen Sie unter Security group (Sicherheitsgruppe) eine oder mehrere Sicherheitsgruppen aus.
8. Wählen Sie Create endpoint.

Um die private DNS-Option verwenden zu können, müssen Sie `enableDnsHostnames` und `enableDnsSupportattributes` Ihrer VPC festlegen. Weitere Informationen finden Sie unter Anzeigen und Aktualisieren der DNS-Unterstützung für Ihre VPC im Amazon-VPC-Benutzerhandbuch. Wenn Sie einen privaten DNS für den Schnittstellenendpunkt aktivieren, können Sie mittels seines standardmäßigen DNS-Namen für die Region, beispielsweise , API-Anforderungen an Lambda senden., `lambda.us-east-1.amazonaws.com`. Weitere Service-Endpunkte finden Sie unter Service-Endpunkte und Kontingente im Allgemeine AWS-Referenz.

Weitere Informationen finden Sie unter Zugriff auf einen Service über einen Schnittstellenendpunkt im Amazon VPC Benutzerhandbuch.

Informationen zum Erstellen und Konfigurieren eines Endpunkts mit CloudFormation finden Sie in der AWS::EC2::VPCEndpoint-Ressource im AWS CloudFormation-Benutzerhandbuch.

So erstellen Sie einen Schnittstellen-Endpunkt für Lambda (AWS CLI)

Verwenden Sie den create-vpc-endpoint-Befehl und geben Sie die VPC-ID an, den VPC-Endpunkttyp (Schnittstelle), den Servicenamen, die Subnetze, die den Endpunkt verwenden sollen, sowie die Sicherheitsgruppen, die den Endpunktnetzwerkschnittstellen zugeordnet werden sollen. Zum Beispiel:

```
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
```

Erstellen einer Richtlinie des Schnittstellenendpunkts für Lambda

Wenn Sie steuern möchten, wer Ihren Schnittstellenendpunkt verwenden kann und auf welche Lambda-Funktionen der Benutzer zugreifen kann, können Sie eine Endpunktrichtlinie an Ihren Endpunkt anfügen. Die Richtlinie gibt die folgenden Informationen an:

- Prinzipal, der die Aktionen ausführen kann
- Die Aktionen, die der Prinzipal ausführen kann.
- Die Ressourcen, auf denen der Prinzipal Aktionen ausführen kann.

Weitere Informationen finden Sie unter Steuerung des Zugriffs auf Services mit VPC-Endpunkten im Amazon VPC Benutzerhandbuch.

Beispiel: Schnittstellenendpunktrichtlinie für Lambda-Aktionen

Im Folgenden finden Sie ein Beispiel für eine Endpunktrichtlinie für Lambda. Bei Anfügung an einen Endpunkt ermöglicht diese Richtlinie dem Benutzer `MyUser` die Funktion `my-function` aufzurufen.

> **Note**
>
> Sie müssen sowohl den qualifizierten als auch den nicht qualifizierten Funktions-ARN in die Ressource aufnehmen.

```
{
   "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:*"
            ]
      }
   ]
}
```

Konfigurieren des Dateisystemzugriffs für Lambda-Funktionen

Sie können eine Funktion konfigurieren, um eine Bindungsbereitstellung für ein Amazon-Elastic-File-System-(Amazon-EFS)-Dateisystem in einem lokalen Verzeichnis zu erstellen. Mit Amazon EFS kann Ihr Funktionscode auf freigegebene Ressourcen sicher und mit hoher Nebenläufigkeit zugreifen und diese ändern.

Unterstützte Regionen

Amazon EFS für Lambda ist in allen kommerziellen Regionen außer Asien-Pazifik (Neuseeland), Asien-Pazifik (Taipeh), Asien-Pazifik (Malaysia), Mexiko (Zentral), Asien-Pazifik (Thailand) und Kanada West (Calgary) verfügbar.

Abschnitte

- Ausführungsrolle und Benutzerberechtigungen
- Konfigurieren eines Dateisystems und eines Zugriffspunkts
- Herstellen einer Verbindung mit einem Dateisystem (Konsole)

Ausführungsrolle und Benutzerberechtigungen

Wenn das Dateisystem keine benutzerkonfigurierte AWS Identity and Access Management (IAM)-Richtlinie hat, verwendet EFS eine Standardrichtlinie, die jedem Client, der über ein Dateisystem-Mounting-Ziel eine Verbindung zum Dateisystem herstellen kann, vollen Zugriff gewährt. Wenn das Dateisystem eine benutzerkonfigurierte IAM-Richtlinie hat, muss die Ausführungsrolle Ihrer Funktion die richtigen `elasticfilesystem`-Berechtigungen besitzen.

Berechtigungen für die Ausführungsrolle

- ElasticFileSystem:ClientMount
- ElasticFileSystem:ClientWrite (nicht für schreibgeschützte Verbindungen erforderlich)

Diese Berechtigungen sind in der verwalteten Richtlinie AmazonElasticFileSystemClientReadWriteAccess enthalten. Darüber hinaus muss Ihre Ausführungsrolle die Berechtigungen, die für die Verbindung mit der VPC des Dateisystems erforderlich sind, besitzen.

Wenn Sie ein Dateisystem konfigurieren, werden Ihre Berechtigungen von Lambda verwendet, um Bindungsbereitstellung-Ziele zu überprüfen. Um eine Funktion für die Herstellung von Verbindungen mit einer VPC zu konfigurieren, benötigt Ihr Benutzer die folgenden Berechtigungen:

Benutzerberechtigungen

- ElasticFileSystem:DescribeMountTargets

Konfigurieren eines Dateisystems und eines Zugriffspunkts

Erstellen Sie ein Dateisystem in Amazon EFS mit einem Bindungsbereitstellung-Ziel in jeder Availability Zone, zu der Ihre Funktion eine Verbindung herstellt. Verwenden Sie für Leistung und Ausfallsicherheit mindestens zwei Availability Zones. In einer einfachen Konfiguration könnten Sie beispielsweise eine VPC mit zwei privaten Subnetzen in separaten Availability Zones haben. Die Funktion verbindet sich mit beiden Subnetzen und ein Mount-Ziel ist in jedem verfügbar. Stellen Sie sicher, dass NFS-Datenverkehr (Port 2049) von den Sicherheitsgruppen erlaubt ist, die von den Funktionen und Mount-Zielen verwendet werden.

> **Note**
>
> Wenn Sie ein Dateisystem erstellen, wählen Sie einen Leistungsmodus, der später nicht geändert werden kann. Der Allzweckmodus hat eine geringere Latenz, und der Max I/O-Modus unterstützt einen höheren maximalen Durchsatz und IOPS. Hilfe bei der Auswahl finden Sie unter Amazon-EFS-Leistung im Amazon-Elastic-File-System-Benutzerhandbuch.

Ein Zugriffspunkt verbindet jede Instance der Funktion mit dem richtigen Mount-Ziel für die Availability Zone, mit der sie eine Verbindung herstellt. Um eine optimale Leistung zu erzielen, erstellen Sie einen Zugriffspunkt mit einem Nicht-Stammpfad und beschränken Sie die Anzahl der Dateien, die Sie in jedem Verzeichnis erstellen. Im folgenden Beispiel wird ein Verzeichnis mit dem Namen `my-function` auf dem Dateisystem erstellt und die Besitzer-ID auf 1001 mit Standardverzeichnisberechtigungen (755) festgelegt.

Example Konfiguration von Zugriffspunkten

- Name (Name – `files`
- Benutzer-ID – `1001`
- Gruppen-ID – `1001`

- Pfad – `/my-function`
- Berechtigungen – `755`
- Benutzer-ID des Eigentümers – `1001`
- Gruppen-Benutzer-ID – `1001`

Wenn eine Funktion den Zugriffspunkt verwendet, erhält sie die Benutzer-ID 1001 und hat vollen Zugriff auf das Verzeichnis.

Weitere Informationen finden Sie in den folgenden Themen im Amazon-Elastic-File-System-Benutzerhandbuch:

- Erstellen von Ressourcen für Amazon EFS
- Arbeiten mit Benutzern, Gruppen und Berechtigungen

Herstellen einer Verbindung mit einem Dateisystem (Konsole)

Eine Funktion stellt eine Verbindung mit einem Dateisystem über das lokale Netzwerk in einer VPC her. Die Subnetze, mit denen Ihre Funktion eine Verbindung herstellt, können dieselben Subnetze sein, die Mounting-Punkte für Ihr Dateisystem enthalten, oder Subnetze in derselben Availability Zone, die NFS-Datenverkehr (Port 2049) an das Dateisystem weiterleiten kann.

> **Note**
>
> Wenn Ihre Funktion noch nicht mit einer VPC verbunden ist, finden Sie weitere Informationen unter Lambda-Funktionen Zugriff auf Ressourcen in einer Amazon VPC gewähren.

So konfigurieren Sie den Zugriff auf Dateisysteme

1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.
2. Wählen Sie eine Funktion aus.
3. Wählen Sie Konfiguration und dann Dateisysteme.
4. Wählen Sie unter Dateisystem die Option Dateisystem hinzufügen.
5. Konfigurieren Sie die folgenden Eigenschaften:
   - EFS-Dateisystem – Der Zugriffspunkt für ein Dateisystem in derselben VPC.

- Lokaler Bindungsbereitstellungspfad – Der Speicherort, an dem das Dateisystem für die Lambda-Funktion gemountet ist, beginnend mit `/mnt/`.

> **Preisgestaltung**
>
> Amazon EFS erhebt Gebühren für Speicher und Durchsatz, mit Raten, die je nach Speicherklasse variieren. Details dazu finden Sie unter Amazon-EFS-Preise.
> Lambda erhebt Gebühren für die Datenübertragung zwischen VPCs. Dies gilt nur, wenn die VPC Ihrer Funktion auf eine andere VPC mit einem Dateisystem gepeered wird. Die Preise sind die gleichen wie für die Amazon-EC2-Datenübertragung zwischen VPCs in derselben Region. Einzelheiten finden Sie unter Lambda – Preise.

Erstellung eines Alias für eine Lambda-Funktion

Sie können für Ihre Lambda-Funktion Aliasse erstellen. Ein Lambda-Alias ist ein Zeiger auf eine Version einer Funktion, der aktualisiert werden kann. Die Benutzer der Funktion können über den Amazon-Ressourcennamen (ARN) des Alias auf die Funktionsversion zugreifen. Wenn Sie eine neue Version bereitstellen, können Sie den Alias aktualisieren, um ihn auf die neue Version umzustellen oder den Datenverkehr zwischen zwei Versionen aufzuteilen.

Console

So erstellen Sie einen Alias mit der Konsole

1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.
2. Wählen Sie eine Funktion aus.
3. Wählen Sie Aliasse und dann Alias erstellen aus.
4. Führen Sie auf der Seite Alias erstellen die folgenden Schritte aus:
   a. Machen Sie für den Alias eine Angabe unter Name.
   b. (Optional) Geben Sie eine Beschreibung für den Alias ein.
   c. Wählen Sie unter Version, eine Funktionsversion aus, auf die der Alias verweisen soll.
   d. (Optional) Um das Routing für den Alias zu konfigurieren, erweitern Sie den Eintrag Gewichteter Alias. Weitere Informationen finden Sie unter Implementieren Sie Lambda Canary-Bereitstellungen mithilfe eines gewichteten Alias.
   e. Wählen Sie Save (Speichern) aus.

AWS CLI

Verwenden Sie den Befehl create-alias, um einen Alias mit der AWS Command Line Interface (AWS CLI) zu erstellen.

```
aws lambda create-alias \
  --function-name my-function \
  --name alias-name \
  --function-version version-number \
  --description " "
```

Verwenden Sie den Befehl update-alias, um einen Alias so zu ändern, dass er auf eine neue Version der Funktion verweist.

```
aws lambda update-alias \
  --function-name my-function \
  --name alias-name \
  --function-version version-number
```

Verwenden Sie den Befehl delete-alias, um einen Alias zu löschen.

```
aws lambda delete-alias \
  --function-name my-function \
  --name alias-name
```

Die obigen AWS CLI-Befehle entsprechen den folgenden Lambda-API-Aktionen.

- CreateAlias
- UpdateAlias
- DeleteAlias

Verwendung von Lambda-Aliasen in Ereignisquellen und Berechtigungsrichtlinien

Jeder Alias hat einen eindeutigen ARN. Ein Alias kann nur auf eine Funktionsversion verweisen, nicht auf einen anderen Alias. Sie können einen Alias aktualisieren, um auf eine neue Version der Funktion zu verweisen.

Ereignisquellen wie Amazon Simple Storage Service (Amazon S3) rufen Ihre Lambda-Funktion auf. Diese Ereignisquellen pflegen eine Zuweisung, die die Funktion identifiziert, die beim Auftreten von Ereignissen aufgerufen werden soll. Wenn Sie in der Mappingkonfiguration einen Lambda-Funktionsalias angeben, müssen Sie das Mapping nicht aktualisieren, wenn sich die Funktionsversion ändert. Weitere Informationen finden Sie unter Wie Lambda Datensätze aus Stream- und warteschlangenbasierten Ereignisquellen verarbeitet.

In einer Ressourcenrichtlinie können Sie Berechtigungen für Ereignisquellen zur Verwendung Ihrer Lambda-Funktion erteilen. Wenn Sie in der Richtlinie einen Alias-ARN angeben, müssen Sie die Richtlinie nicht aktualisieren, wenn sich die Funktionsversion ändert.

Ressourcenrichtlinien

Sie können eine ressourcenbasierte Richtlinie verwenden, um einem Dienst, einer Ressource oder einem Konto Zugriff auf Ihre Funktion zu gewähren. Der Umfang dieser Berechtigung hängt davon ab, ob Sie sie auf einen Alias, eine Version oder die gesamte Funktion anwenden. Wenn Sie beispielsweise einen Aliasnamen verwenden (z. B. `helloworld:PROD`), gewährt Ihnen die Berechtigung, die Funktion `helloworld` mithilfe des ARN des Alias (`helloworld:PROD`) aufzurufen.

Wenn Sie versuchen, die Funktion ohne Alias oder eine bestimmte Version aufzurufen, erhalten Sie einen Berechtigungsfehler. Dieser Berechtigungsfehler tritt weiterhin auf, selbst wenn Sie versuchen, die dem Alias zugeordnete Funktionsversion direkt aufzurufen.

Der folgende AWS CLI-Befehl erteilt beispielsweise Amazon-S3-Berechtigungen zum Aufrufen des PROD-Alias der Funktion `helloworld`, wenn Amazon S3 im Auftrag von `amzn-s3-demo-bucket` handelt.

```
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
```

Weitere Hinweise zum Verwenden von Ressourcennamen in Richtlinien finden Sie unter Feinabstimmung der Abschnitte „Ressourcen“ und „Bedingungen“ der Richtlinien.

Implementieren Sie Lambda Canary-Bereitstellungen mithilfe eines gewichteten Alias

Sie können einen gewichteten Alias verwenden, um den Datenverkehr auf zwei verschiedene Versionen derselben Funktion aufzuteilen. Mit diesem Ansatz können Sie neue Versionen Ihrer Funktionen mit einem kleinen Prozentsatz des Datenverkehrs testen und bei Bedarf schnell wieder zurücksetzen. Dies wird als Canary-Bereitstellung bezeichnet. Kanarische Bereitstellungen unterscheiden sich von blue/green Bereitstellungen dadurch, dass die neue Version nur einem Teil der Anfragen ausgesetzt wird, anstatt den gesamten Datenverkehr auf einmal umzuleiten.

Sie können ein Alias auf maximal zwei Lambda-Funktionsversionen verweisen lassen. Die Versionen müssen folgende Kriterien erfüllen:

- Beide Versionen müssen die gleiche Ausführungsrolle haben.
- Beide Versionen müssen die gleiche Warteschlangenkonfiguration für unzustellbare Nachrichten oder keine Warteschlangenkonfiguration für unzustellbare Nachrichten haben.
- Beide Versionen müssen veröffentlicht werden. Der Alias darf nicht auf verweise `$LATEST`.

> **Note**
>
> Lambda verwendet ein einfaches probabilistisches Modell, um den Datenverkehr zwischen den beiden Funktionsversionen zu verteilen. Bei niedrigem Datenverkehr sehen Sie möglicherweise eine hohe Abweichung zwischen dem konfigurierten und dem tatsächlichen Prozentsatz des Datenverkehrs für jede Version. Wenn Ihre Funktion bereitgestellte Parallelität verwendet, können Sie Überlaufaufrufe durch Konfigurieren einer höheren Anzahl von bereitgestellten Parallelitätsinstances während des aktiven Alias-Routings vermeiden.

Einen gewichteten Alias erstellen

Console

So konfigurieren Sie das Routing für einen Alias über die Konsole

> **Note**
>
> Stellen Sie sicher, dass die Funktion über mindestens zwei veröffentlichte Versionen verfügt. Um weitere Versionen zu erstellen, folgen Sie den Anweisungen unter Erstellen von Funktionsversionen.

1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.
2. Wählen Sie eine Funktion aus.
3. Wählen Sie Aliasse und dann Alias erstellen aus.
4. Führen Sie auf der Seite Alias erstellen die folgenden Schritte aus:
   a. Machen Sie für den Alias eine Angabe unter Name.
   b. (Optional) Geben Sie eine Beschreibung für den Alias ein.
   c. Wählen Sie unter Version die erste Funktionsversion aus, auf die der Alias verweisen soll.
   d. Erweitern Sie Gewichteter Alias.
   e. Wählen Sie unter Zusätzliche Version die zweite Funktionsversion aus, auf die der Alias verweisen soll.
   f. Geben Sie unter Gewicht (%) einen Gewichtungswert für die Funktion ein. Die Gewichtung ist der Prozentsatz des Datenverkehrs, der beim Aufruf des Alias dieser Version zugewiesen wird. Die erste Version erhält den Datenverkehr mit der verbleibenden Gewichtung. Wenn Sie beispielsweise 10 Prozent für die Additional version (Zusätzliche Version) angeben, werden der ersten Version automatisch 90 Prozent zugewiesen.
   g. Wählen Sie Speichern.

AWS CLI

Verwenden Sie die AWS CLI Befehle create-alias und update-alias, um die Datenverkehrsgewichte zwischen zwei Funktionsversionen zu konfigurieren. Wenn Sie den Alias erstellen oder aktualisieren, geben Sie die Datenverkehrsgewichtung im `routing-config`-Parameter an.

Im folgenden Beispiel wird ein Lambda-Funktionsalias namens routing-alias erstellt, der auf Version 1 der Funktion verweist. Version 2 der Funktion erhält 3 Prozent des Datenverkehrs. Die verbleibenden 97 Prozent des Datenverkehrs werden zu Version 1 weitergeleitet.

```
aws lambda create-alias \
  --name routing-alias \
  --function-name my-function \
  --function-version 1  \
  --routing-config AdditionalVersionWeights={"2"=0.03}
```

Verwenden Sie den `update-alias`-Befehl, um den Prozentsatz des eingehenden Datenverkehrs an Version 2 zu erhöhen. Im folgenden Beispiel erhöhen Sie den Datenverkehr auf 5 Prozent.

```
aws lambda update-alias \
  --name routing-alias \
  --function-name my-function \
  --routing-config AdditionalVersionWeights={"2"=0.05}
```

Um den gesamten Datenverkehr an Version 2 weiterzuleiten, ändern Sie den Befehl `update-alias`, um die Eigenschaft `function-version` so zu ändern, dass der Alias auf Version 2 verweist. Der Befehl setzt auch die Weiterleitungskonfiguration zurück.

```
aws lambda update-alias \
  --name routing-alias \
  --function-name my-function  \
  --function-version 2 \
  --routing-config AdditionalVersionWeights={}
```

Die AWS CLI Befehle in den vorherigen Schritten entsprechen den folgenden Lambda-API-Operationen:

- CreateAlias

- UpdateAlias

Feststellen, welche Version aufgerufen wurde

Wenn Sie Datenverkehrsgewichtungen zwischen zwei Funktionsversionen konfigurieren, gibt es zwei Möglichkeiten, die aufgerufene Lambda-Funktionsversion zu bestimmen:

- CloudWatch Logs — Lambda gibt automatisch einen `START` Protokolleintrag aus, der die aufgerufene Versions-ID für jeden Funktionsaufruf enthält. Beispiel:

  ```
  START RequestId: 1dh194d3759ed-4v8b-a7b4-1e541f60235f Version: 2
  ```

  Für Aliasaufrufe verwendet Lambda die Dimension `ExecutedVersion` zum Filtern der Metrikdaten nach aufgerufener Version. Weitere Informationen finden Sie unter Anzeigen von Metriken für Lambda-Funktionen.
- Antwortnutzlast (synchrone Aufrufe) – Antworten auf synchrone Funktionsaufrufe enthalten einen `x-amz-executed-version`-Header, der angibt, welche Funktionsversion aufgerufen wurde.

Erstellen Sie eine fortlaufende Bereitstellung mit gewichteten Aliasnamen

Verwenden Sie AWS CodeDeploy und AWS Serverless Application Model (AWS SAM), um eine fortlaufende Bereitstellung zu erstellen, die automatisch Änderungen an Ihrem Funktionscode erkennt, eine neue Version Ihrer Funktion bereitstellt und die Menge des Datenverkehrs, der zur neuen Version fließt, schrittweise erhöht. Der Datenverkehr und die Erhöhungsrate sind Parameter, die Sie konfigurieren können.

AWS SAM Führt in einer fortlaufenden Bereitstellung die folgenden Aufgaben aus:

- Konfiguriert Ihre Lambda-Funktion und erstellt einen Alias. Die gewichtete Alias-Weiterleitungskonfiguration ist die zugrunde liegende Funktion, die die fortlaufende Bereitstellung implementiert.
- Erstellt eine CodeDeploy Anwendung und eine Bereitstellungsgruppe. Die Bereitstellungsgruppe verwaltet die fortlaufende Bereitstellung und das Rollback, falls erforderlich.
- Erkennt, wenn Sie eine neue Version Ihrer Lambda-Funktion erstellen.
- Löst aus CodeDeploy , um die Bereitstellung der neuen Version zu starten.

Beispiel für AWS SAM eine Vorlage

Das folgende Beispiel zeigt eine AWS SAM -Vorlage für eine einfache fortlaufende Bereitstellung.

```
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
```

Diese Vorlage definiert eine Lambda-Funktion mit der Bezeichnung `myDateTimeFunction` und den folgenden Eigenschaften.

AutoPublishAlias

Die `AutoPublishAlias`-Eigenschaft erstellt einen Alias mit der Bezeichnung `live`. Darüber hinaus erkennt das AWS SAM -Framework automatisch, wenn Sie neuen Code für die Funktion speichern. Das Framework veröffentlicht dann eine neue Funktionsversion und aktualisiert den `live`-Alias so, dass er auf die neue Version verweist.

DeploymentPreference

Die `DeploymentPreference` Eigenschaft bestimmt die Geschwindigkeit, mit der die CodeDeploy Anwendung den Datenverkehr von der ursprünglichen Version der Lambda-Funktion auf die neue Version verlagert. Der Wert `Linear10PercentEvery2Minutes` verschiebt alle zwei Minuten weitere zehn Prozent des Datenverkehrs zur neuen Version.

Eine Liste der vordefinierten Bereitstellungskonfigurationen finden Sie unter Bereitstellungskonfigurationen.

Weitere Informationen zum Erstellen fortlaufender Bereitstellungen mit CodeDeploy und AWS SAM finden Sie im Folgenden:

- Tutorial: Stellen Sie eine aktualisierte Lambda-Funktion bereit mit CodeDeploy und AWS Serverless Application Model
- Schrittweise Bereitstellung serverloser Anwendungen mit AWS SAM

Versionen der Lambda-Funktion verwalten

Sie können Versionen verwenden, um die Bereitstellung Ihrer Funktionen zu verwalten. Beispielsweise können Sie eine neue Version einer Funktion für Beta-Tests veröffentlichen, ohne dass Benutzer der stabilen Produktionsversion betroffen sind. Lambda erstellt jedes Mal, wenn Sie die Funktion veröffentlichen, eine neue Version Ihrer Funktion. Die neue Version ist eine Kopie der unveröffentlichten Version der Funktion. Die unveröffentlichte Version heißt `$LATEST`.

Wichtig ist, dass Sie bei jeder Bereitstellung Ihres Funktionscodes den aktuellen Code in `$LATEST` überschreiben. Um die aktuelle Iteration von `$LATEST` zu speichern, erstellen Sie eine neue Funktionsversion. Wenn `$LATEST` mit einer zuvor veröffentlichten Version identisch ist, können Sie erst dann eine neue Version erstellen, wenn Sie Änderungen an `$LATEST` bereitgestellt haben. Zu diesen Änderungen können die Aktualisierung des Codes oder die Änderung der Funktionskonfigurationseinstellungen gehören.

Nachdem Sie eine Funktionsversion veröffentlicht haben, sind Code, Laufzeit, Architektur, Speicher, Ebenen und die meisten anderen Konfigurationseinstellungen unveränderlich. Das bedeutet, dass Sie diese Einstellungen nicht ändern können, ohne eine neue Version von `$LATEST` zu veröffentlichen. Sie können die folgenden Elemente für eine veröffentlichte Funktionsversion konfigurieren:

- Auslöser
- Ziele
- Bereitgestellte Gleichzeitigkeit
- Asynchroner Aufruf
- Datenbankverbindungen und Proxys

> **Note**
>
> Bei Verwendung von Steuerelementen zur Laufzeitverwaltung im Modus Auto (Automatisch) wird die Laufzeitversion, die von der Funktionsversion verwendet wird, automatisch aktualisiert. Bei Verwendung des Modus Function update (Funktionsaktualisierung) oder Manual (Manuell) wird die Laufzeitversion nicht aktualisiert. Weitere Informationen finden Sie unter the section called “Aktualisierungen der Laufzeitversion”.

Sections

- Erstellen von Funktionsversionen
- Verwenden von Versionen
- Gewähren von Berechtigungen

Erstellen von Funktionsversionen

Sie können den Funktionscode und die Einstellungen nur in der nicht veröffentlichten Version einer Funktion ändern. Wenn Sie eine Version veröffentlichen, sperrt Lambda den Code und die meisten Einstellungen, um eine konsistente Erfahrung für Benutzer dieser Version zu gewährleisten.

Sie können eine Funktionsversion mit der Lambda-Konsole erstellen.

So erstellen Sie eine neue Funktionsversion

1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.
2. Wählen Sie eine Funktion aus und dann den Tab Versionen.
3. Wählen Sie auf der Versionskonfigurationsseite die Option Neue Version veröffentlichen aus.
4. (Optional) Geben Sie eine Versionsbeschreibung ein.
5. Wählen Sie Publish.

Alternativ können Sie eine Version einer Funktion mithilfe der PublishVersionAPI-Operation veröffentlichen.

Der folgende AWS CLI Befehl veröffentlicht eine neue Version einer Funktion. In der Antwort werden die Konfigurationsinformationen der neuen Version zurückgegeben, einschließlich der Versionsnummer und des Funktion-ARN mit dem Versionssuffix.

```
aws lambda publish-version --function-name my-function
```

Die Ausgabe sollte folgendermaßen aussehen:

```
{
  "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 weist monoton steigende Sequenznummern für die Versionsverwaltung zu. Lambda verwendet niemals Versionsnummern wieder, auch nicht nachdem Sie eine Funktion gelöscht und neu erstellt haben.

Verwenden von Versionen

Sie können Ihre Lambda-Funktion entweder mit einem qualifizierten ARN oder einem nicht qualifizierten ARN referenzieren.

- Qualifizierter ARN – Hierbei handelt es sich um den ARN der Funktion mit einem Versionssuffix. Das folgende Beispiel bezieht sich auf Version 42 der Funktion `helloworld`.

  ```
  arn:aws:lambda:aws-region:acct-id:function:helloworld:42
  ```

- Unqualifizierter ARN – Hierbei handelt es sich um den ARN der Funktion ohne ein Versionssuffix.

  ```
  arn:aws:lambda:aws-region:acct-id:function:helloworld
  ```

Sie können in allen relevanten API-Operationen einen qualifizierten oder einen nicht qualifizierten ARN verwenden. Sie können jedoch keinen unqualifizierten ARN verwenden, um einen Alias zu erstellen.

Wenn Sie sich entscheiden, Funktionsversionen nicht zu veröffentlichen, können Sie entweder den qualifizierten oder den nicht qualifizierten ARN in Ihrer Ereignisquellen-Zuweisung verwenden, um die Funktion aufzurufen. Wenn Sie eine Funktion mit einem unqualifizierten ARN aufrufen, ruft Lambda implizit `$LATEST` auf.

Der qualifizierte ARN für jede Lambda-Funktionsversion ist eindeutig. Nachdem Sie eine Version veröffentlicht haben, können Sie den ARN oder den Funktionscode nicht ändern.

Lambda veröffentlicht eine neue Funktionsversion nur dann, wenn der Code noch nie veröffentlicht wurde oder wenn sich der Code im Vergleich zur zuletzt veröffentlichten Version geändert hat. Wenn keine Änderung vorliegt, bleibt die Funktionsversion die zuletzt veröffentlichte Version.

Wenn Sie eine Version veröffentlichen, erstellt Lambda einen unveränderlichen Snapshot des Codes und der Konfiguration Ihrer Funktion. Nicht alle Konfigurationsänderungen lösen die Veröffentlichung einer neuen Version aus. Die folgenden Konfigurationsänderungen qualifizieren eine Funktion für die Versionsveröffentlichung:

- Funktionscode
- Umgebungsvariablen
- Laufzeit
- Handler
- Layer
- Arbeitsspeichergröße
- Zeitüberschreitung
- VPC-Konfiguration
- Konfiguration der Warteschlange für unzustellbare Nachrichten
- IAM role (IAM-Rolle)
- Description
- Architektur (x86_64 oder arm64)
- Größe des flüchtigen Speichers
- Art Package
- Protokollierungskonfiguration
- Dateisystemkonfiguration
- SnapStart
- Nachverfolgungskonfiguration

Betriebseinstellungen wie reservierte Parallelität lösen nicht die Veröffentlichung einer neuen Version aus, wenn sie geändert werden.

Gewähren von Berechtigungen

Sie können eine ressourcenbasierte Richtlinie oder eine identitätsbasierte Richtlinie verwenden, um Zugriff auf Ihre Funktion zu gewähren. Der Umfang der Berechtigung hängt davon ab, ob Sie die Richtlinie auf eine Funktion oder auf eine Version einer Funktion anwenden. Weitere Hinweise zu Funktionsressourcennamen in Richtlinien finden Sie unter Feinabstimmung der Abschnitte „Ressourcen“ und „Bedingungen“ der Richtlinien.

Sie können die Verwaltung von Ereignisquellen und AWS Identity and Access Management (IAM-) Richtlinien vereinfachen, indem Sie Funktionsaliase verwenden. Weitere Informationen finden Sie unter Erstellung eines Alias für eine Lambda-Funktion.

Verwenden von Tags für Lambda-Funktionen

Sie können Tag-Funktionen markieren, um Ihre Ressourcen zu organisieren und zu verwalten. Tags sind frei formulierte Schlüssel-Wert-Paare, die mit Ihren Ressourcen verknüpft sind und von AWS-Services unterstützt werden. Weitere Informationen zu Anwendungsfällen für Tags finden Sie unter Allgemeine Tagging-Strategien im Tagging AWS Resources and Tag Editor Guide.

Tags gelten für die Funktionsebene, nicht für Versionen oder Aliase. Tags sind nicht Teil der versionsspezifischen Konfiguration, die einen Snapshot davon AWS Lambda erstellt, wann Sie eine Version veröffentlichen. Sie können die Lambda-API verwenden, um Tags anzuzeigen und zu aktualisieren. Sie können Tags auch anzeigen und aktualisieren, während Sie eine bestimmte Funktion in der Lambda-Konsole verwalten.

Sections

- Erforderliche Berechtigungen zum Arbeiten mit Tags
- Verwendung von Tags mit der Lambda-Konsole
- Verwenden von Tags mit AWS CLI

Erforderliche Berechtigungen zum Arbeiten mit Tags

Um einer AWS Identity and Access Management -(IAM)-Identität (Benutzer, Gruppe oder Rolle) das Lesen oder Setzen von Tags auf einer Ressource zu ermöglichen, gewähren Sie ihr die entsprechenden Berechtigungen:

- lambda: ListTags —Wenn eine Ressource Tags enthält, gewähren Sie diese Berechtigung jedem, der sie aufrufen muss. `ListTags` Für Funktionen mit Tags ist diese Berechtigung auch für `GetFunction` erforderlich.
- lambda: TagResource —Erteilen Sie diese Berechtigung jedem, der bei create ein Tag aufrufen `TagResource` oder ausführen muss.

Optional können Sie erwägen, auch die lambda: UntagResource -Berechtigung zu erteilen, um `UntagResource` Aufrufe der Ressource zuzulassen.

Weitere Informationen finden Sie unter Identitätsbasierte IAM-Richtlinien für Lambda.

Verwendung von Tags mit der Lambda-Konsole

Sie können die Lambda-Konsole verwenden, um Funktionen mit Tags zu erstellen, Tags zu vorhandenen Funktionen hinzuzufügen und Funktionen nach von Ihnen hinzugefügten Tags zu filtern.

Hinzufügen von Tags beim Erstellen einer Funktion

1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.
2. Wählen Sie Funktion erstellen.
3. Klicken Sie auf Ohne Vorgabe erstellen oder Container-Image.
4. Richten Sie unter Grundinformationen Ihre Funktion ein. Weitere Hinweise zum Konfigurieren von Funktionen finden Sie unter Konfigurieren von -Funktionen.
5. Erweitern Sie Advanced settings (Erweiterte Einstellungen) und wählen Sie dann Enable tags (Tags aktivieren) aus.
6. Wählen Sie Add new tag (Neues Tag hinzufügen) und geben Sie dann einen Key (Schlüssel) und einen optionalen Value (Wert) ein. Wiederholen Sie diesen Schritt, um weitere Tags hinzuzufügen.
7. Wählen Sie Funktion erstellen.

Hinzufügen von Tags zu einer bestehenden Funktion

1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.
2. Wählen Sie den Namen einer Funktion aus.
3. Wählen Sie Configuraton (Konfiguration) und dann Tags aus.
4. Wählen Sie unter Tags die Option Manage tags (Tags verwalten) aus.
5. Wählen Sie Add new tag (Neues Tag hinzufügen) und geben Sie dann einen Key (Schlüssel) und einen optionalen Value (Wert) ein. Wiederholen Sie diesen Schritt, um weitere Tags hinzuzufügen.
6. Wählen Sie Speichern.

So filtern Sie Funktionen mit Tags

1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.
2. Wählen Sie die Suchleiste aus, um eine Liste mit Funktionseigenschaften und Tag-Schlüsseln anzuzeigen.

3. Wählen Sie einen Tag-Schlüssel, um eine Liste der Werte zu sehen, die in der aktuellen AWS Region verwendet werden.
4. Wählen Sie Verwenden: „Tag-Name“, um alle Funktionen zu sehen, die mit dieser Taste gekennzeichnet sind oder wählen Sie einen Operator, um weiter nach Werten zu filtern.
5. Wählen Sie Ihren Tag-Wert aus, um nach einer Kombination aus Tag-Schlüssel und -Wert zu filtern.

Die Suchleiste unterstützt auch die Suche nach Tag-Schlüsseln. Geben Sie `tag` ein, um nur eine Liste der Tag-Schlüssel anzuzeigen, oder geben Sie den Namen eines Schlüssels ein, um ihn in der Liste zu suchen.

Verwenden von Tags mit AWS CLI

Mit der Lambda-API können Sie Tags zu vorhandenen Lambda-Ressourcen, einschließlich Funktionen, hinzufügen und entfernen. Sie können beim Erstellen einer Funktion auch Tags hinzufügen, sodass Sie eine Ressource während ihres gesamten Lebenszyklus mit Tags versehen können.

Aktualisieren von Tags mit dem Lambda-Tag APIs

Sie können Tags für unterstützte Lambda-Ressourcen über die TagResourceund UntagResourceAPI-Operationen hinzufügen und entfernen.

Sie können diese Operationen mit der Taste AWS CLI aufrufen. Um Tags einer vorhandenen Ressource hinzuzufügen, verwenden Sie den folgenden `tag-resource`-Befehl. In diesem Beispiel werden zwei Tags hinzugefügt, eines mit dem Schlüssel *`Department`* und eines mit dem Schlüssel*`CostCenter`*.

```
aws lambda tag-resource \
--resource arn:aws:lambda:us-east-2:123456789012:resource-type:my-resource \
--tags Department=Marketing,CostCenter=1234ABCD
```

Mit dem Befehl `untag-resource` können Sie Tags entfernen. In diesem Beispiel wird das Tag mit dem Schlüssel entfernt*`Department`*.

```
aws lambda untag-resource --resource arn:aws:lambda:us-east-1:123456789012:resource-
type:resource-identifier \
--tag-keys Department
```

Hinzufügen von Tags beim Erstellen einer Funktion

Verwenden Sie die CreateFunctionAPI-Operation, um eine neue Lambda-Funktion mit Tags zu erstellen. Geben Sie den Parameter `Tags` an. Sie können diesen Vorgang mit dem `create-function`-CLI-Befehl und der Option --tags aufrufen. Vergewissern Sie sich vor der Verwendung des Tags-Parameters mit `CreateFunction`, dass Ihre Rolle neben den üblichen Berechtigungen, die für diesen Vorgang erforderlich sind, auch die Berechtigung hat, Ressourcen zu taggen. Weitere Informationen zu den Berechtigungen für das Tagging finden Sie unter the section called "Erforderliche Berechtigungen zum Arbeiten mit Tags". In diesem Beispiel werden zwei Tags hinzugefügt, eines mit dem Schlüssel *`Department`* und eines mit dem Schlüssel*`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
```

Anzeigen von Tags in einer Funktion

Um die Tags anzuzeigen, die auf eine bestimmte Lambda-Ressource angewendet werden, verwenden Sie die `ListTags`-API-Operation. Weitere Informationen finden Sie unter ListTags.

Sie können diesen Vorgang mit dem `list-tags` AWS CLI Befehl aufrufen, indem Sie einen ARN (Amazon Resource Name) angeben.

```
aws lambda list-tags --resource arn:aws:lambda:us-east-1:123456789012:resource-
type:resource-identifier
```

Sie können die Tags anzeigen, die mit der GetFunctionAPI-Operation auf eine bestimmte Ressource angewendet werden. Eine vergleichbare Funktion ist nicht für andere Ressourcentypen verfügbar.

Sie können diesen Vorgang mit dem `get-function`-CLI-Befehl aufrufen:

```
aws lambda get-function --function-name my-function
```

Filtern von Ressourcen nach Tag

Sie können den AWS Resource Groups Tagging API GetResourcesAPI-Vorgang verwenden, um Ihre Ressourcen nach Tags zu filtern. Die `GetResources`-Operation empfängt bis zu 10 Filter, wobei

jeder Filter einen Tag-Schlüssel und bis zu 10 Tag-Werte enthält. Sie stellen `GetResources` mit einem `ResourceType` zur Verfügung, um nach bestimmten Ressourcentypen zu filtern.

Sie können diesen Vorgang mit dem `get-resources` AWS CLI Befehl aufrufen. Beispiele für die Verwendung von `get-resources` finden Sie unter get-resources in der AWS -CLI-Befehlsreferenz.

Antwortstreaming für Lambda-Funktionen

Lambda-Funktionen können Antwortnutzlasten nativ über die Lambda-Funktion URLs oder mithilfe der InvokeWithResponseStreamAPI (über das AWS SDK oder direkte API-Aufrufe) an Clients zurückstreamen. Ihre Lambda-Funktion kann auch Antwort-Payloads über die Amazon API Gateway Gateway-Proxyintegration streamen, die die InvokeWithResponseStreamAPI verwendet, um Ihre Funktion aufzurufen. Antwort-Streaming kann für latenzempfindliche Anwendungen von Vorteil sein, da es die Leistung in der Zeit bis zum ersten Byte (TTFB) verbessert. Dies liegt daran, dass Sie Teilantworten an den Client zurücksenden können, sobald sie verfügbar sind. Darüber hinaus können Antwort-Streaming-Funktionen Nutzdaten bis zu 200 MB zurückgeben, verglichen mit dem Maximum von 6 MB für gepufferte Antworten. Das Streameng einer Antwort bedeutet auch, dass Ihre Funktion nicht die gesamte Antwort im Speicher unterbringen muss. Bei sehr großen Antworten kann dies die Menge des Speichers reduzieren, die Sie für Ihre Funktion konfigurieren müssen.

> **Note**
>
> Lambda-Response-Streaming ist noch nicht in allen AWS-Regionen Bereichen verfügbar. Informationen zur Verfügbarkeit von AWS Funktionen nach Regionen finden Sie in den Funktionen des Builder Centers nach Regionen.

Die Geschwindigkeit, mit der Lambda Ihre Antworten streamt, hängt von der Größe der Antwort ab. Die Streaming-Rate für die ersten 6 MB der Antwort Ihrer Funktion ist nicht begrenzt. Bei Antworten, die größer als 6 MB sind, gilt für den Rest der Antwort eine Bandbreitenbegrenzung. Weitere Informationen zur Streaming-Bandbreite finden Sie unter Bandbreitenbegrenzung für Antwort-Streaming.

Streaming-Antworten sind mit Kosten verbunden, und gestreamte Antworten werden nicht unterbrochen oder gestoppt, wenn die Verbindung zum aufrufenden Client unterbrochen wird. Kunden wird die volle Funktionsdauer in Rechnung gestellt, weshalb Kunden bei der Konfiguration langer Funktions-Timeouts Vorsicht walten lassen sollten.

Lambda unterstützt Antwortstreaming auf verwalteten Laufzeiten von Node.js. Für andere Sprachen, einschließlich Python, können Sie eine benutzerdefinierte Laufzeit mit einer benutzerdefinierten Runtime-API-Integration verwenden, um Antworten zu streamen, oder den Lambda Web Adapter verwenden.

> **Note**
>
> Wenn Sie Ihre Funktion über die Lambda-Konsole testen, werden die Antworten immer gepuffert angezeigt.

Themen

- Bandbreitenbegrenzung für Antwort-Streaming
- VPC-Kompatibilität mit Antwort-Streaming
- Schreiben von Antwort-Streaming-fähigen Lambda-Funktionen
- Aufrufen einer für Response-Streaming aktivierten Funktion mithilfe der Lambda-Funktion URLs
- Tutorial: Erstellen einer Lambda-Funktion zum Streamen von Antworten mit einer Funktions-URL

Bandbreitenbegrenzung für Antwort-Streaming

Für die ersten 6 MB der Nutzlast Ihrer Funktion gibt es keine Bandbreitenbegrenzung. Nach diesem ersten Burst streamt Lambda Ihre Antwort mit einer maximalen Rate von 2. MBps Wenn Ihre Funktionsantworten nie größer als 6 MB sind, gilt diese Bandbreitenbegrenzung nicht.

> **Note**
>
> Bandbreitenbegrenzungen gelten nur für die Antwortnutzlast Ihrer Funktion und nicht für den Netzwerkzugriff durch Ihre Funktion.

Die Rate der unbegrenzten Bandbreite hängt von einer Reihe von Faktoren ab, einschließlich der Verarbeitungsgeschwindigkeit Ihrer Funktion. Normalerweise können Sie MBps für die ersten 6 MB der Antwort Ihrer Funktion mit einer Rate von mehr als 2 rechnen. Wenn Ihre Funktion eine Antwort an ein Ziel außerhalb von streamt AWS, hängt die Streaming-Rate auch von der Geschwindigkeit der externen Internetverbindung ab.

VPC-Kompatibilität mit Antwort-Streaming

Bei der Verwendung von Lambda-Funktionen in einer VPC-Umgebung gibt es wichtige Überlegungen zum Antwort-Streaming:

- Die Lambda-Funktion unterstützt URLs kein Antwort-Streaming innerhalb einer VPC-Umgebung.

- Sie können Antwort-Streaming innerhalb einer VPC verwenden, indem Sie Ihre Lambda-Funktion über das AWS SDK mithilfe der API aufrufen. `InvokeWithResponseStream` Das erfordert die Einrichtung der entsprechenden VPC-Endpunkte für Lambda.
- Für VPC-Umgebungen müssen Sie einen VPC-Schnittstellen-Endpunkt für Lambda erstellen, um die Kommunikation zwischen Ihren Ressourcen in der VPC und dem Lambda-Service zu ermöglichen.

Eine typische Architektur für das Antwort-Streaming in einer VPC könnte Folgendes beinhalten:

```
Client in VPC -> Interface VPC endpoint for Lambda -> Lambda function -> Response
 streaming back through the same path
```

Schreiben von Antwort-Streaming-fähigen Lambda-Funktionen

Das Schreiben des Handlers für Antwort-Streaming-Funktionen unterscheidet sich von typischen Handler-Mustern. Wenn Sie Streaming-Funktionen schreiben, sollten Sie Folgendes beachten:

- Umschließen Sie Ihre Funktion mit dem `awslambda.streamifyResponse()`-Decorator. Das globale `awslambda`-Objekt wird von der Lambda-Laufzeitumgebung Node.js bereitgestellt.
- Beenden Sie den Stream ordnungsgemäß, um sicherzustellen, dass die Datenverarbeitung abgeschlossen ist.

Konfigurieren einer Handler-Funktion zum Streamen von Antworten

Um der Laufzeit zu signalisieren, dass Lambda die Antworten Ihrer Funktion streamen soll, müssen Sie Ihre Funktion mit dem `streamifyResponse()`-Decorator wrappen. Dadurch wird die Laufzeit angewiesen, den richtigen Logikpfad für Streaming-Antworten zu verwenden, und ermöglicht es der Funktion, Antworten zu streamen.

Der `streamifyResponse()`-Decorator akzeptiert eine Funktion, die die folgenden Parameter akzeptiert:

- `event` – Stellt Informationen über das Aufrufereignis der Funktions-URL bereit, z. B. die HTTP-Methode, Abfrageparameter und den Anforderungskörper.
- `responseStream` – Stellt einen schreibbaren Stream bereit.
- `context` – Stellt Methoden und Eigenschaften mit Informationen zum Aufruf, zur Funktion und zur Ausführungsumgebung bereit.

Das `responseStream`-Objekt ist ein Node.js `writableStream`. Wie bei jedem solchen Stream sollten Sie die `pipeline()`-Methode verwenden.

> **Note**
>
> Das globale `awslambda`-Objekt wird automatisch von der Lambda-Laufzeit Node.js bereitgestellt, und es ist kein Import erforderlich.

Example Handler mit aktiviertem Antwort-Streaming

```
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);
});
```

Obwohl `responseStream` die `write()`-Methode bietet, in den Stream zu schreiben, empfehlen wir Ihnen, wo immer möglich, `pipeline()` zu verwenden. Die Verwendung von `pipeline()` stellt sicher, dass der schreibbare Stream nicht von einem schneller lesbaren Stream überfordert wird.

Den Stream beenden

Stellen Sie sicher, dass Sie den Stream ordnungsgemäß beenden, bevor der Handler zurückkehrt. Die `pipeline()`-Methode verarbeitet dies automatisch.

Für andere Anwendungsfälle rufen Sie die `responseStream.end()`-Methode auf, um einen Stream ordnungsgemäß zu beenden. Diese Methode signalisiert, dass keine Daten mehr in den Stream geschrieben werden sollen. Diese Methode ist nicht erforderlich, wenn Sie mit `pipeline()` oder `pipe()` in den Stream schreiben.

Ab Node.js 24 wartet Lambda nicht mehr darauf, dass ungelöste Versprechen abgeschlossen werden, nachdem Ihr Handler zurückgekehrt ist oder der Antwortstream beendet wurde. Wenn Ihre Funktion von zusätzlichen asynchronen Vorgängen wie Timern oder Abrufen abhängt, sollten Sie sie in Ihrem Handler verwenden. `await`

Example Beispiel für das Beenden eines Streams mit Pipeline ()

```
import { pipeline } from 'node:stream/promises';

export const handler = awslambda.streamifyResponse(async (event, responseStream,
 _context) => {
  await pipeline(requestStream, responseStream);
});
```

Example Beispiel für das Beenden eines Streams ohne Pipeline ()

```
export const handler = awslambda.streamifyResponse(async (event, responseStream,
 _context) => {
  responseStream.write("Hello ");
  responseStream.write("world ");
  responseStream.write("from ");
  responseStream.write("Lambda!");
  responseStream.end();
});
```

Aufrufen einer für Response-Streaming aktivierten Funktion mithilfe der Lambda-Funktion URLs

> **Note**
>
> Ihre Lambda-Funktion kann jetzt Antwort-Payloads über die Amazon API Gateway Gateway-Proxyintegration streamen.

Sie können Funktionen mit aktiviertem Antwort-Streaming aufrufen, indem Sie den Aufrufmodus der URL Ihrer Funktion ändern. Der Aufrufmodus bestimmt, welche API-Operation Lambda verwendet, um Ihre Funktion aufzurufen. Die verfügbaren Aufrufmodi sind:

- `BUFFERED` – Dies ist die Standardoption. Lambda ruft Ihre Funktion mithilfe der `Invoke`-API-Operation auf. Die Aufrufergebnisse sind verfügbar, wenn die Nutzlast abgeschlossen ist. Die maximale Nutzlastgröße beträgt 6 MB.
- `RESPONSE_STREAM` – Ermöglicht es Ihrer Funktion, die Ergebnisse der Nutzlasten zu streamen, sobald sie verfügbar sind. Lambda ruft Ihre Funktion mithilfe der `InvokeWithResponseStream`-API-Operation auf. Die maximale Größe der Antwort-Payload beträgt 200 MB.

Sie können Ihre Funktion auch ohne Antwort-Streaming aufrufen, indem Sie die `Invoke`-API-Operation direkt aufrufen. Lambda streamt jedoch alle Antwort-Nutzlasten für Aufrufe, die über die URL der Funktion erfolgen, bis Sie den Aufrufmodus auf `BUFFERED` ändern.

Console

Um den Aufrufmodus einer Funktions-URL (Konsole) festzulegen

1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.
2. Wählen Sie den Namen der Funktion, für die Sie den Aufrufmodus festlegen möchten.
3. Wählen Sie die Registerkarte Konfiguration und dann Funktions-URL.
4. Wählen Sie Bearbeiten und dann Zusätzliche Einstellungen.
5. Wählen Sie unter Aufrufmodus den gewünschten Aufrufmodus aus.
6. Wählen Sie Speichern.

AWS CLI

Um den Aufrufmodus einer Funktions-URL (AWS CLI) festzulegen

```
aws lambda update-function-url-config \
  --function-name my-function \
  --invoke-mode RESPONSE_STREAM
```

CloudFormation

Um den Aufrufmodus einer Funktions-URL (CloudFormation) festzulegen

```
MyFunctionUrl:
  Type: AWS::Lambda::Url
  Properties:
    AuthType: AWS_IAM
    InvokeMode: RESPONSE_STREAM
```

Weitere Informationen zur Konfiguration der Funktion finden Sie URLs unter Lambda-Funktion URLs.

Tutorial: Erstellen einer Lambda-Funktion zum Streamen von Antworten mit einer Funktions-URL

In diesem Tutorial erstellen Sie eine Lambda-Funktion, die als ZIP-Dateiarchiv mit einem Funktions-URL-Endpunkt definiert ist, der einen Antwort-Stream zurückgibt. Weitere Hinweise zur Konfiguration der Funktion URLs finden Sie unterFunktion URLs.

Voraussetzungen

In diesem Tutorial wird davon ausgegangen, dass Sie über Kenntnisse zu den grundlegenden Lambda-Operationen und der Lambda-Konsole verfügen. Sofern noch nicht geschehen, befolgen Sie die Anweisungen unter Erstellen einer Lambda-Funktion mit der Konsole, um Ihre erste Lambda-Funktion zu erstellen.

Um die folgenden Schritte durchzuführen, benötigen Sie die AWS CLI Version 2. Befehle und die erwartete Ausgabe werden in separaten Blöcken aufgeführt:

```
aws --version
```

Die Ausgabe sollte folgendermaßen aussehen:

```
aws-cli/2.13.27 Python/3.11.6 Linux/4.14.328-248.540.amzn2.x86_64 exe/x86_64.amzn.2
```

Bei langen Befehlen wird ein Escape-Zeichen (\) wird verwendet, um einen Befehl über mehrere Zeilen zu teilen.

Verwenden Sie auf Linux und macOS Ihren bevorzugten Shell- und Paket-Manager.

> **Note**
>
> In Windows werden einige Bash-CLI-Befehle, die Sie häufig mit Lambda verwenden (z. B. `zip`), von den integrierten Terminals des Betriebssystems nicht unterstützt. Um eine in Windows integrierte Version von Ubuntu und Bash zu erhalten, installieren Sie das Windows-Subsystem für Linux. Die CLI-Beispielbefehle in diesem Handbuch verwenden die Linux-Formatierung. Befehle, die Inline-JSON-Dokumente enthalten, müssen neu formatiert werden, wenn Sie die Windows-CLI verwenden.

Erstellen einer Ausführungsrolle

Erstellen Sie die Ausführungsrolle die Ihrer Lambda-Funktion die Berechtigung für den Zugriff auf AWS -Ressourcen erteilt.

So erstellen Sie eine Ausführungsrolle

1. Öffnen Sie die Seite Rollen der AWS Identity and Access Management (IAM)-Konsole.
2. Wählen Sie Rolle erstellen aus.
3. Erstellen Sie eine Rolle mit den folgenden Eigenschaften:
   - Vertrauenswürdiger Entitätstyp – AWS -Service
   - Anwendungsfall – Lambda
   - Berechtigungen – AWSLambdaBasicExecutionRole
   - Role name (Name der Rolle – **`response-streaming-role`**

Die AWSLambdaBasicExecutionRoleRichtlinie verfügt über die Berechtigungen, die die Funktion benötigt, um Protokolle in Amazon CloudWatch Logs zu schreiben. Nachdem Sie die Rolle erstellt haben, notieren Sie sich den Amazon-Ressourcennamen (ARN). Sie benötigen ihn im nächsten Schritt.

Erstellen Sie eine Antwort-Streaming-Funktion (AWS CLI)

Erstellen Sie eine Antwort-Streaming-Lambda-Funktion mit einem Funktions-URL-Endpunkt unter Verwendung der AWS Command Line Interface (AWS CLI).

Um eine Funktion zu erstellen, die Antworten streamen kann

1. Kopieren Sie das folgende Codebeispiel in eine Datei mit dem Namen `index.js`. Diese Funktion streamt drei Antworten im Abstand von 1 Sekunde.

```
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. Erstellen Sie ein Bereitstellungspaket.

```
zip function.zip index.js
```

3. Erstellen Sie eine Lambda-Funktion mit dem Befehl `create-function`. Ersetzen Sie den Wert von `--role` durch den Rollen-ARN aus dem vorherigen Schritt. Mit diesem Befehl wird das Funktions-Timeout auf 10 Sekunden festgelegt, sodass die Funktion drei Antworten streamen kann.

```
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
```

Um eine Funktions-URL zu erstellen

1. Fügen Sie Ihrer Funktion eine ressourcenbasierte Richtlinie hinzu, die die Berechtigungen `lambda:InvokeFunctionUrl` und `lambda:InvokeFunction` gewährt. Jede Anweisung

muss in einem separaten Befehl hinzugefügt werden. Ersetzen Sie den Wert von `--principal` durch Ihre AWS-Konto ID.

```
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. Erstellen Sie einen URL-Endpunkt für die Funktion mit dem Befehl `create-function-url-config`.

```
aws lambda create-function-url-config \
  --function-name my-streaming-function \
  --auth-type AWS_IAM \
  --invoke-mode RESPONSE_STREAM
```

> **Note**
>
> Wenn Sie eine Fehlermeldung über `--invoke-mode` erhalten, müssen Sie möglicherweise auf eine neuere Version der AWS CLI aktualisieren.

Testen des Funktions-URL-Endpunkts

Testen Sie Ihre Integration, indem Sie Ihre Funktion aufrufen. Sie können die URL Ihrer Funktion in einem Browser öffnen oder curl verwenden.

```
curl --request GET "https://abcdefghijklm7nop7qrs740abcd.lambda-url.us-east-1.on.aws/"
 --user "AKIAIOSFODNN7EXAMPLE" --aws-sigv4 "aws:amz:us-east-1:lambda" --no-buffer
```

Unsere Funktions-URL nutzt den Authentifizierungstyp `IAM_AUTH`. Dies bedeutet, dass Sie Anfragen sowohl mit Ihrem AWS -Zugriffsschlüssel als auch mit Ihrem geheimen Schlüssel signieren müssen. Ersetzen Sie ihn im vorherigen Befehl `AKIAIOSFODNN7EXAMPLE` durch die AWS Zugriffsschlüssel-ID. Geben Sie Ihren AWS geheimen Schlüssel ein, wenn Sie dazu aufgefordert werden. Wenn Sie Ihren AWS geheimen Schlüssel nicht haben, können Sie stattdessen temporäre AWS Anmeldeinformationen verwenden.

Das Ergebnis sollte ungefähr wie folgt aussehen:

```
Streaming with Helper
Hello 0
Hello 1
Hello 2
```

Bereinigen Ihrer Ressourcen

Sie können jetzt die Ressourcen, die Sie für dieses Tutorial erstellt haben, löschen, es sei denn, Sie möchten sie behalten. Durch das Löschen von AWS Ressourcen, die Sie nicht mehr verwenden, vermeiden Sie unnötige Kosten für Ihre AWS-Konto.

So löschen Sie die Ausführungsrolle

1. Öffnen Sie die Seite Roles in der IAM-Konsole.
2. Wählen Sie die von Ihnen erstellte Ausführungsrolle aus.
3. Wählen Sie Löschen aus.
4. Geben Sie den Namen der Rolle in das Texteingabefeld ein und wählen Sie Delete (Löschen) aus.

So löschen Sie die Lambda-Funktion:

1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.
2. Wählen Sie die Funktion aus, die Sie erstellt haben.
3. Wählen Sie Aktionen, Löschen aus.
4. Geben Sie **`confirm`** in das Texteingabefeld ein und wählen Sie Delete (Löschen) aus.

Grundlegendes zu Methoden zum Aufrufen von Lambda-Funktionen

Nach dem Bereitstellen Ihrer Lambda-Funktion können Sie sie auf verschiedene Arten aufrufen:

- Die Lambda-Konsole – Verwenden Sie die Lambda-Konsole, um schnell ein Testereignis zu erstellen, mit dem Sie Ihre Funktion aufrufen.
- Das AWS SDK — Verwenden Sie das AWS SDK, um Ihre Funktion programmgesteuert aufzurufen.
- Die Invoke-API – Verwenden Sie die Lambda-Invoke-API, um Ihre Funktion direkt aufzurufen.
- Das AWS Command Line Interface (AWS CLI) — Verwenden Sie den `aws lambda invoke` AWS CLI Befehl, um Ihre Funktion direkt von der Befehlszeile aus aufzurufen.
- Ein URL-HTTP (S) -Endpunkt für eine Funktion — Verwenden Sie URLs die Funktion, um einen dedizierten HTTP (S) -Endpunkt zu erstellen, mit dem Sie Ihre Funktion aufrufen können.

All diese Methoden sind direkte Möglichkeiten, Ihre Funktion aufzurufen. In Lambda besteht ein häufiger Anwendungsfall darin, Ihre Funktion auf der Grundlage eines Ereignisses aufzurufen, das an anderer Stelle in Ihrer Anwendung auftritt. Einige Dienste können bei jedem neuen Ereignis eine Lambda-Funktion aufrufen. Dies wird als Trigger bezeichnet. Für stream- und warteschlangenbasierte Dienste ruft Lambda die Funktion mit Batches von Datensätzen auf. Dies wird als Zuordnung von Ereignisquellen bezeichnet.

Wenn Sie eine Funktion aufrufen, können Sie bestimmen, ob sie synchron oder asynchron aufgerufen wird. Bei einem synchronen Aufruf warten Sie, bis die Funktion das Ereignis verarbeitet und eine Antwort zurückgegeben hat. Bei einem asynchronen Aufruf stellt Lambda das Ereignis für die Verarbeitung in eine Warteschlangen und gibt umgehend eine Antwort zurück. Der `InvocationType`-Anforderungsparameter in der Invoke-API bestimmt, wie Lambda Ihre Funktion aufruft. Der Wert von `RequestResponse` steht für einen synchronen Aufruf und der Wert von `Event` steht für einen asynchronen Aufruf.

Verwenden Sie die öffentlichen Dual-Stack-Endpunkte von Lambda IPv6, um Ihre Funktion erneut aufzurufen. Dual-Stack-Endpunkte unterstützen sowohl als auch. IPv4 IPv6 Lambda-Dual-Stack-Endpunkte verwenden die folgende Syntax:

```
protocol://lambda.us-east-1.api.aws
```

Sie können die Lambda-Funktion auch verwenden URLs, um Funktionen aufzurufen. IPv6 Funktions-URL-Endpunkte haben das folgende Format:

```
https://url-id.lambda-url.us-east-1.on.aws
```

Wenn der Funktionsaufruf zu einem Fehler führt, sehen Sie sich bei synchronen Aufrufen die Fehlermeldung in der Antwort an und wiederholen Sie den Aufruf manuell. Bei asynchronen Aufrufen werden Wiederholungsversuche automatisch von Lambda verarbeitet und Aufrufdatensätze an ein Ziel gesendet.

Eine Lambda-Funktion synchron aufrufen

Wenn Sie eine Funktion synchron aufrufen, führt Lambda die Funktion aus und wartet auf eine Antwort. Wenn der Funktionsaufruf beendet ist, gibt Lambda die Antwort aus dem Funktionscode mit zusätzlichen Daten, wie z. B. der Version der aufgerufenen Funktion, zurück. Wenn Sie eine Funktion mithilfe der AWS CLI synchron ausführen möchten, verwenden Sie den Befehl `invoke`.

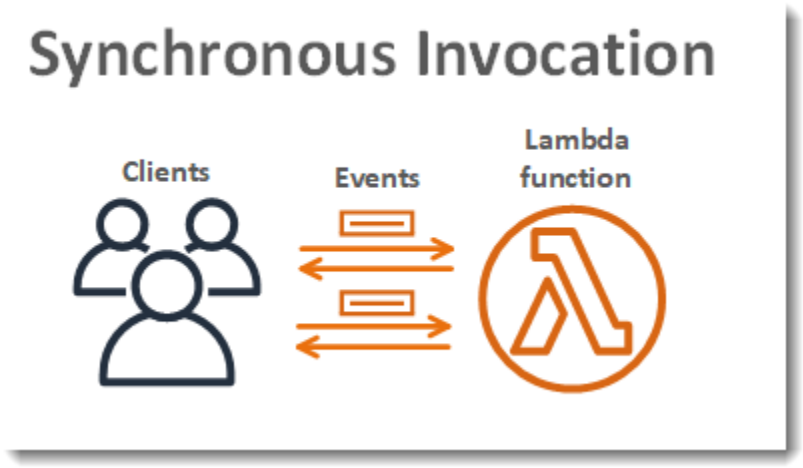
```
aws lambda invoke --function-name my-function \
    --cli-binary-format raw-in-base64-out \
    --payload '{ "key": "value" }' response.json
```

Die cli-binary-format-Option ist erforderlich, wenn Sie AWS CLI Version 2 verwenden. Um dies zur Standardeinstellung zu machen, führen Sie `aws configure set cli-binary-format raw-in-base64-out` aus. Weitere Informationen finden Sie unter Von AWS CLI unterstützte globale Befehlszeilenoptionen im AWS Command Line Interface-Benutzerhandbuch für Version 2.

Die Ausgabe sollte folgendermaßen aussehen:

```
{
    "ExecutedVersion": "$LATEST",
    "StatusCode": 200
}
```

Das folgende Diagramm zeigt Clients, die eine Lambda-Funktion synchron aufrufen. Lambda sendet die Ereignisse direkt an die Funktion und sendet die Antwort der Funktion zurück an den Aufrufer.



`payload` ist eine Zeichenfolge, die ein Ereignis im JSON-Format enthält. Der Name der Datei, in die die AWS CLI die Antwort von der Funktion schreibt, lautet `response.json`. Wenn die Funktion ein

Objekt oder einen Fehler zurückgibt, ist der Antworttext das Objekt oder der Fehler im JSON-Format. Wenn die Funktion fehlerfrei beendet wird, lautet der Antworttext `null`.

> **Note**
>
> Lambda wartet nicht, bis externe Erweiterungen abgeschlossen sind, bevor die Antwort gesendet wird. Externe Erweiterungen werden als unabhängige Prozesse in der Ausführungsumgebung ausgeführt und laufen auch nach Abschluss des Funktionsaufrufs weiter. Weitere Informationen finden Sie unter Lambda-Funktionen mit Lambda-Erweiterungen erweitern.

Die Ausgabe des Befehls, die im Terminal angezeigt wird, enthält Informationen aus Headern in der Antwort von Lambda. Dies umfasst die Version, von der das Ereignis verarbeitet wurde (nützlich bei der Verwendung von Aliassen) und den von Lambda zurückgegebenen Statuscode. Wenn Lambda die Funktion ausführen konnte, lautet der Statuscode 200, auch wenn die Funktion einen Fehler zurückgegeben hat.

> **Note**
>
> Bei Funktionen mit einem langen Timeout wird Ihr Client während eines synchronen Aufrufs möglicherweise getrennt, während er auf eine Antwort wartet. Konfigurieren Sie HTTP-Client, SDK, Firewall, Proxy oder Betriebssystem so, dass lange Verbindungen mit Timeout- oder Keepalive-Einstellungen möglich sind.

Wenn Lambda die Funktion nicht ausführen kann, wird der Fehler in der Ausgabe angezeigt.

```
aws lambda invoke --function-name my-function \
    --cli-binary-format raw-in-base64-out \
    --payload value response.json
```

Die Ausgabe sollte folgendermaßen aussehen:

```
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]
```

Die AWS CLI ist ein Open-Source-Tool, mit dem Sie über Befehle in Ihrer Befehlszeilen-Shell mit den AWS-Services interagieren können. Zur Durchführung der Schritte in diesem Abschnitt benötigen Sie AWS CLI Version 2.

Sie können die AWS CLI verwenden, um Protokolle für einen Aufruf mit der `--log-type`-Befehlsoption abzurufen. Die Antwort enthält das Feld `LogResult`, das bis zu 4 KB base64-verschlüsselte Protokolle aus dem Aufruf enthält.

Example eine Log-ID abrufen

Das folgende Beispiel zeigt, wie eine Protokoll-ID aus dem `LogResult`-Feld für eine Funktion namens `my-function` abgerufen wird.

```
aws lambda invoke --function-name my-function out --log-type Tail
```

Die Ausgabe sollte folgendermaßen aussehen:

```
{
    "StatusCode": 200,
    "LogResult":
 "U1RBUlQgUmVxdWVzdElkOiA4N2QwNDRiOC1mMTU0LTExZTgtOGNkYS0yOTc0YzVlNGZiMjEgVmVyc2lvb...",
    "ExecutedVersion": "$LATEST"
}
```

Example entschlüsseln der Protokolle

Verwenden Sie in derselben Eingabeaufforderung das `base64`-Dienstprogramm, um die Protokolle zu entschlüsseln. Das folgende Beispiel zeigt, wie Base64-codierte Logs für abgerufen werde `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
```

Die cli-binary-format-Option ist erforderlich, wenn Sie AWS CLI Version 2 verwenden. Um dies zur Standardeinstellung zu machen, führen Sie `aws configure set cli-binary-format raw-in-base64-out` aus. Weitere Informationen finden Sie unter Von AWS CLI unterstützte globale Befehlszeilenoptionen im AWS Command Line Interface-Benutzerhandbuch für Version 2.

Die Ausgabe sollte folgendermaßen aussehen:

```
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
```

Das `base64`-Dienstprogramm ist unter Linux, macOS und Ubuntu auf Windowsverfügbar. macOS-Benutzer müssen möglicherweise `base64 -D` verwenden.

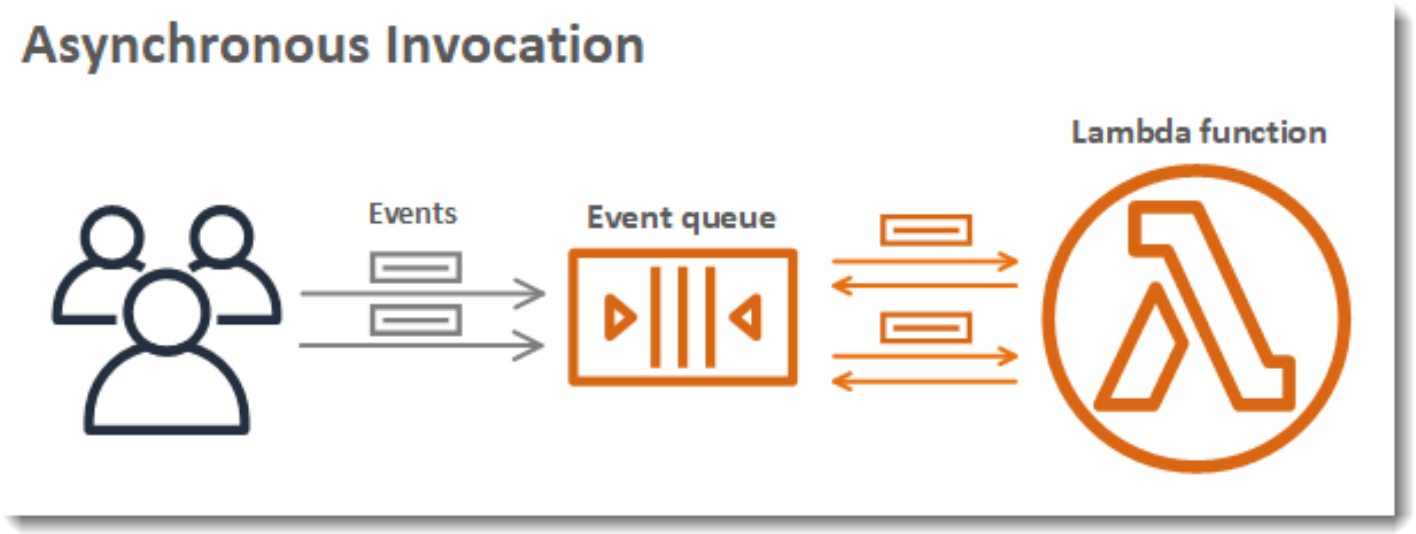
Weitere Informationen über die `Invoke`-API, einschließlich einer vollständigen Liste von Parametern, Kopfzeilen und Fehlern, finden Sie unter Aufrufen.

Wenn Sie eine Funktion direkt aufrufen, können Sie die Antwort auf Fehler überprüfen und es erneut versuchen. Die AWS CLI- und das AWS SDK unternehmen zudem automatisch einen Wiederholversuch bei Client-Zeitbeschränkungen, Drosselung und Service-Fehlern. Weitere Informationen finden Sie unter Grundlegendes zum Wiederholungsverhalten in Lambda.

Asynchrones Aufrufen einer Lambda-Funktion

Mehrere AWS-Services wie Amazon Simple Storage Service (Amazon S3) und Amazon Simple Notification Service (Amazon SNS) rufen Funktionen asynchron auf, um Ereignisse zu verarbeiten. Sie können eine Lambda-Funktion auch asynchron mit dem AWS Command Line Interface (AWS CLI) oder einem der AWS-SDKs aufrufen. Wenn Sie eine Funktion asynchron aufrufen, warten Sie nicht auf eine Antwort aus dem Funktionscode. Sie übergeben das Ereignis an Lambda und Lambda erledigt den Rest. Sie können konfigurieren, wie Lambda Fehler behandelt, und Sie können Aufruf-Datensätze an eine nachgelagerte Ressource wie Amazon Simple Queue Service (Amazon SQS) oder Amazon EventBridge (EventBridge) senden, um Komponenten Ihrer Anwendung miteinander zu verketten.

Das folgende Diagramm zeigt Clients, die eine Lambda-Funktion asynchron aufrufen. Lambda stellt die Ereignisse in eine Warteschlange, bevor sie an die Funktion gesendet werden.



Lambda platziert das Ereignis bei asynchronem Aufruf in eine Warteschlange und gibt eine Erfolgsantwort ohne zusätzliche Informationen zurück. Ein separater Prozess liest Ereignisse aus der Warteschlange und senden sie zu Ihrer Funktion.

Um eine Lambda-Funktion asynchron mit dem AWS Command Line Interface (AWS CLI) oder einem der AWS-SDKs aufzurufen, setzen Sie den InvocationType-Parameter auf `Event`. Das folgende Beispiel zeigt einen AWS CLI-Befehl zum Aufrufen einer Funktion.

```
aws lambda invoke \
  --function-name my-function  \
  --invocation-type Event \
```

```
  --cli-binary-format raw-in-base64-out \
  --payload '{ "key": "value" }' response.json
```

Die Ausgabe sollte folgendermaßen aussehen:

```
{
    "StatusCode": 202
}
```

Die cli-binary-format-Option ist erforderlich, wenn Sie AWS CLI Version 2 verwenden. Um dies zur Standardeinstellung zu machen, führen Sie `aws configure set cli-binary-format raw-in-base64-out` aus. Weitere Informationen finden Sie unter Von AWS CLI unterstützte globale Befehlszeilenoptionen im AWS Command Line Interface-Benutzerhandbuch für Version 2.

Die Ausgabedatei (`response.json`) enthält keine Informationen, wird bei der Ausführung dieses Befehls aber dennoch erstellt. Wenn Lambda das Ereignis nicht zur Warteschlange hinzufügen kann, erscheint in der Befehlsausgabe eine Fehlermeldung.

Wie Lambda mit Fehlern und Wiederholungsversuchen mit asynchronem Aufruf umgeht

Lambda verwaltet Ihre asynchrone Ereigniswarteschlange der Funktion und unternimmt bei Fehlern Wiederholungsversuche. Wenn die Funktion einen Fehler zurückgibt, versucht Lambda standardmäßig zwei weitere Male, sie auszuführen. Dabei wird eine Minute lang zwischen den ersten zwei Versuchen und zwei Minuten lang zwischen dem zweiten und dem dritten Versuch gewartet. Zu Funktionsfehlern gehören Fehler, die vom Code der Funktion zurückgegeben werden, sowie Fehler, die von der Laufzeit des Fehlers zurückgegeben werden, wie z. B. Timeouts.

Wenn für die Funktion nicht genügend Gleichzeitigkeit für die Verarbeitung aller Ereignisse verfügbar ist, werden weitere Anforderungen gedrosselt. Bei Drosselungsfehlern (429) und Systemfehlern (500-Serie) sendet Lambda das Ereignis zur Warteschlange zurück und versucht standardmäßig bis zu 6 Stunden lang, die Funktion erneut auszuführen. Das Wiederholungsintervall erhöht sich exponentiell von 1 Sekunde nach dem ersten Versuch auf maximal 5 Minuten. Lambda stellt viele Einträge in die Warteschlange, erhöht das Wiederholungsintervall und reduziert die Rate, mit der es Ereignisse aus der Warteschlange liest.

Selbst wenn Ihre Funktion keinen Fehler zurückgibt, ist es möglich, dass sie dasselbe Ereignis mehrmals von Lambda erhält, da die Warteschlange selbst „letztendliche Konsistenz“ ist. Wenn die

Funktion mit eingehenden Ereignissen nicht Schritt halten kann, kann es auch vorkommen, dass Ereignisse aus der Warteschlange gelöscht werden, ohne zur Funktion gesendet zu werden. Stellen Sie sicher, dass Ihr Funktionscode doppelte Ereignisse ordnungsgemäß verarbeitet und dass Sie über genügend Gleichzeitigkeit zur Verarbeitung aller Aufrufe verfügen.

Wenn die Warteschlange sehr lang ist, können neue Ereignisse veraltet sein, bevor Lambda sie an Ihre Funktion senden kann. Wenn ein Ereignis abläuft oder bei ihm alle Verarbeitungsversuche fehlschlagen, wird es von Lambda verworfen. Sie können die Fehlerbehandlung für eine Funktion konfigurieren, um die Anzahl der Wiederholungsversuche zu reduzieren, die Lambda durchführt, oder um unverarbeitete Ereignisse schneller zu verwerfen. Konfigurieren Sie eine Warteschlange für unzustellbare Nachrichten, um verworfene Ereignisse zu erfassen. Zur Erfassung von Aufzeichnungen fehlgeschlagener Aufrufe (z. B. Zeitüberschreitungen oder Laufzeitfehler) zu erfassen, erstellen Sie Fehlerziel.

Konfigurieren von Fehlerbehandlungseinstellungen für asynchrone Lambda-Aufrufe

Verwenden Sie die folgenden Einstellungen, um zu konfigurieren, wie Lambda Fehler und Wiederholungen bei asynchronen Funktionsaufrufen behandelt:

- MaximumEventAgeInSeconds: Die maximale Zeitspanne in Sekunden, die Lambda ein Ereignis in der Warteschlange für asynchrone Ereignisse behält, bevor es verworfen wird.
- MaximumRetryAttempts: Die maximale Anzahl der Wiederholungen von Lambda-Ereignissen, wenn die Funktion einen Fehler zurückgibt.

Verwenden Sie die Lambda-Konsole oder AWS CLI, um Einstellungen für die Fehlerbehandlung für eine Funktion, eine Version oder einen Alias zu konfigurieren.

Console

So konfigurieren Sie die Fehlerbehandlung

1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.
2. Wählen Sie eine Funktion aus.
3. Wählen Sie „Konfiguration“ und dann „Asynchroner Aufruf“ aus.
4. Wählen Sie unter Asynchronous invocation (Asynchroner Aufruf) die Option Edit (Bearbeiten).
5. Konfigurieren Sie die folgenden Einstellungen.

- Maximales Alter des Ereignisses – Die maximale Zeitspanne, die Lambda ein Ereignis in der Warteschlange für asynchrone Ereignisse behält (bis zu 6 Stunden).
- Wiederholversuche – Die Anzahl der Wiederholungen, die Lambda es erneut versucht, wenn die Funktion einen Fehler zurückgibt (zwischen 0 und 2).

6. Wählen Sie Speichern.

AWS CLI

Um den asynchronen Aufruf mit dem AWS CLI zu konfigurieren, verwenden Sie den Befehl put-function-event-invoke-config. Im folgenden Beispiel wird eine Funktion mit einem maximalen Ereignisalter von 1 Stunde und ohne Wiederholversuche konfiguriert.

```
aws lambda put-function-event-invoke-config \
  --function-name error \
  --maximum-event-age-in-seconds 3600 \
  --maximum-retry-attempts 0
```

Der Befehl `put-function-event-invoke-config` überschreibt alle vorhandenen Konfigurationen für die Funktion, die Version oder den Alias. Um eine Option zu konfigurieren, ohne andere zurückzusetzen, verwenden Sie update-function-event-invoke-config. Im folgenden Beispiel wird Lambda so konfiguriert, dass ein Datensatz an eine SQS-Standardwarteschlange namens `destination` gesendet wird, wenn ein Ereignis nicht verarbeitet werden kann.

```
aws lambda update-function-event-invoke-config \
  --function-name my-function \
  --destination-config '{"OnFailure":{"Destination": "arn:aws:sqs:us-
east-1:123456789012:destination"}}'
```

Die Ausgabe sollte folgendermaßen aussehen:

```
{
    "LastModified": 1573686021.479,
    "FunctionArn": "arn:aws:lambda:us-east-1:123456789012:function:my-function:
$LATEST",
    "MaximumRetryAttempts": 0,
    "MaximumEventAgeInSeconds": 3600,
    "DestinationConfig": {
        "OnSuccess": {},
```

```
        "OnFailure": {}
    }
}
```

Wenn ein Aufrufereignis das maximale Alter überschreitet oder alle Wiederholversuche fehlschlagen, wird es von Lambda verworfen. Um eine Kopie der verworfenen Ereignisse beizubehalten, konfigurieren Sie ein Ziel für fehlgeschlagene Ereignisse.

Erfassung von Datensätzen asynchroner Lambda-Aufrufe

Lambda kann Aufzeichnungen von asynchronen Aufrufen an einen der folgenden Server senden. AWS-Services

- Amazon SQS – Eine standardmäßige SQS-Warteschlange
- Amazon SNS – Ein SNS-Standardthema
- Amazon S3 – Ein Amazon S3-Bucket (nur bei einem Fehler)
- AWS Lambda – Eine Lambda-Funktion
- Amazon EventBridge — Ein EventBridge Eventbus

Der Aufrufdatensatz enthält Details zur Anforderung und Antwort im JSON-Format. Sie können separate Ziele für Ereignisse konfigurieren, die erfolgreich verarbeitet werden, und für Ereignisse, bei denen alle Verarbeitungsversuche fehlschlagen. Alternativ können Sie eine Amazon-SQS-Standardwarteschlange oder ein Amazon-SNS-Standardthema als Warteschlange für unzustellbare Nachrichten für verworfene Ereignisse konfigurieren. Für Warteschlangen für unzustellbare Nachrichten sendet Lambda nur den Inhalt des Ereignisses ohne Details zur Antwort.

Wenn Lambda einen Datensatz nicht an ein von Ihnen konfiguriertes Ziel senden kann, sendet es eine `DestinationDeliveryFailures` Metrik an Amazon CloudWatch. Dies kann passieren, wenn Ihre Konfiguration einen nicht unterstützten Zieltyp enthält, z. B. eine Amazon-SQS-FIFO-Warteschlange oder ein Amazon-SNS-FIFO-Thema. Zustellungsfehler können auch aufgrund von Berechtigungsfehlern und Größenbeschränkungen auftreten. Weitere Informationen zu Lambda-Aufrufmetriken finden Sie unter the section called “Aufrufmetriken”.

> **Note**
>
> Um zu verhindern, dass eine Funktion ausgelöst wird, können Sie die reservierte Parallelität der Funktion auf Null setzen. Wenn Sie die reservierte Parallelität für eine asynchron aufgerufene Funktion auf Null setzen, sendet Lambda sofort und ohne

Wiederholungsversuche alle Ereignisse ohne erneute Versuche an die konfigurierte Warteschlange für unzustellbare Nachrichten oder an das Ereignisziel bei Fehlern. Um Ereignisse zu verarbeiten, die gesendet wurden, während die reservierte Parallelität auf Null gesetzt war, müssen Sie die Ereignisse aus der Warteschlange für unzustellbare Nachrichten oder dem Ereignisziel bei Fehlern verarbeiten.

Hinzufügen eines Ziels

Um Datensätze der asynchronen Aufrufe beizubehalten, fügen Sie Ihrer Funktion ein Ziel hinzu. Sie können wählen, ob erfolgreiche Aufrufe oder fehlgeschlagene Aufrufe an ein Ziel gesendet werden sollen. Jede Funktion kann mehrere Ziele haben, sodass Sie separate Ziele für erfolgreiche und fehlgeschlagene Ereignisse konfigurieren können. Jeder Datensatz, der an das Ziel gesendet wird, ist ein JSON-Dokument mit Details zum Aufruf. Wie Einstellungen zur Fehlerbehandlung können Sie Ziele für eine Funktion, eine Funktionsversion oder einen Alias konfigurieren.

Tip

Sie können auch Aufzeichnungen über fehlgeschlagene Aufrufe für die folgenden Typen von Ereignisquellenzuordnungen speichern: Amazon Kinesis, Amazon DynamoDB und Apache Kafka (Amazon MSK und selbstverwalteter Apache Kafka).

In der folgenden Tabelle werden die unterstützten Ziele für Datensätze der asynchronen Aufrufen aufgelistet. Damit Lambda erfolgreich Datensätze an das von Ihnen ausgewählte Ziel senden kann, stellen Sie sicher, dass die Ausführungsrolle Ihrer Funktion auch die entsprechenden Berechtigungen enthält. In der Tabelle wird auch beschrieben, wie jeder Zieltyp den JSON-Aufrufdatensatz empfängt.

Zieltyp	Erforderliche Berechtigung	Zielspezifisches JSON-Format
Amazon-SQS-Warteschlange	sqs: SendMessage	Lambda übergibt den Aufrufdatensatz als `Message` an das Ziel.
Amazon-SNS-Thema	sns:Publish	Lambda übergibt den Aufrufdatensatz als `Message` an das Ziel.

Zieltyp	Erforderliche Berechtigung	Zielspezifisches JSON-Format
Ein Amazon S3-Bucket (nur bei einem Fehler)	s3: PutObject s3: ListBucket	• Lambda speichert den Aufrufdatensatz als JSON-Objekt im Ziel-Bucket. • Der Name des S3-Objekts wird nach folgender Konvention benannt: `aws/lambda/async/<function-name>/YYYY/MM/DD/YYYY-MM-DDTHH.MM.SS-<Random UUID>`
Lambda-Funktion	Lambda: InvokeFunction	Lambda übergibt den Aufrufdatensatz als Nutzlast in der Funktion.

Zieltyp	Erforderliche Berechtigung	Zielspezifisches JSON-Format
EventBridge	Ereignisse: PutEvents	• Lambda übergibt den Aufrufdatensatz wie `detail` im PutEvents Aufruf. • Der Wert für das `source`-Ereignisfeld ist `aws.lambda` . • Der Wert für das `detail-type` -Ereignisfeld ist entweder „Lambda Function Invocation Result – Success“ (Ergebnis des Lambda-Funktionsaufrufs: Erfolgreich) oder „Lambda Function Invocation Result – Failure“ (Ergebnis des Lambda-Funktionsaufrufs: Fehler). • Das `resource` Ereignisfeld enthält die Funktion und das Ziel Amazon Resource Names (ARNs). • Weitere Veranstaltungsbereiche finden Sie unter EventBridge Amazon-Veranstaltungen.

> **Note**
>
> Wenn Sie für Amazon S3 S3-Ziele die Verschlüsselung für den Bucket mithilfe eines KMS-Schlüssels aktiviert haben, benötigt Ihre Funktion auch die kms: GenerateDataKey -Berechtigung.

> **Important**
>
> Wenn Sie Amazon SNS als Ziel verwenden, beachten Sie, dass Amazon SNS eine maximale Nachrichtengröße von 256 KB hat. Wenn sich Ihre Nutzdaten für asynchrone Aufrufe 1 MB nähern, kann der Aufrufdatensatz (der die ursprünglichen Nutzdaten plus zusätzliche Metadaten enthält) das Amazon-SNS-Limit überschreiten und zu Übermittlungsfehlern führen. Erwägen Sie die Verwendung von Amazon-SQS- oder Amazon-S3-Zielen für größere Nutzdaten.

In den folgenden Schritten wird beschrieben, wie Sie mit der Lambda-Konsole und der AWS CLI ein Ziel für eine Funktion konfigurieren.

Console

1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.
2. Wählen Sie eine Funktion aus.
3. Wählen Sie unter Function overview (Funktionsübersicht) die Option Add destination (Ziel hinzufügen).
4. Wählen Sie unter Source (Quelle) die Option Asynchronous invocation (Asynchroner Aufruf).
5. Wählen Sie unter Condition (Bedingung) eine der folgenden Optionen aus:
   - Bei Fehler – Es wird ein Datensatz gesendet, wenn das Ereignis alle Verarbeitungsversuche ausschöpft oder das maximale Alter überschreitet.
   - Bei Erfolg – Es wird ein Datensatz gesendet, wenn die Funktion einen asynchronen Aufruf erfolgreich verarbeitet.
6. Wählen Sie unter Destination type (Zieltyp) den Ressourcentyp aus, der den Aufrufdatensatz empfängt.
7. Wählen Sie unter Destination (Ziel) eine Ressource aus.
8. Wählen Sie Speichern.

AWS CLI

Um ein Ziel mit dem zu konfigurieren AWS CLI, führen Sie den Befehl update-function-event-invoke-config aus. Im folgenden Beispiel wird Lambda so konfiguriert, dass ein Datensatz an eine

SQS-Standardwarteschlange namens `destination` gesendet wird, wenn ein Ereignis nicht verarbeitet werden kann.

```
aws lambda update-function-event-invoke-config \
  --function-name my-function \
  --destination-config '{"OnFailure":{"Destination": "arn:aws:sqs:us-
east-1:123456789012:destination"}}'
```

Bewährte Methoden für die Sicherheit in Amazon-S3-Zielen

Das Löschen eines S3-Buckets, der als Ziel konfiguriert ist, ohne das Ziel aus der Konfiguration Ihrer Funktion zu entfernen, kann ein Sicherheitsrisiko darstellen. Wenn ein anderer Benutzer den Namen Ihres Ziel-Buckets kennt, kann er den Bucket in seinem AWS-Konto neu erstellen. Aufzeichnungen über fehlgeschlagene Aufrufe werden an den entsprechenden Bucket gesendet, wodurch möglicherweise Daten aus Ihrer Funktion verfügbar gemacht werden.

> **Warning**
>
> Um sicherzustellen, dass Aufrufdatensätze Ihrer Funktion nicht an einen S3-Bucket in einem anderen gesendet werden können AWS-Konto, fügen Sie der Ausführungsrolle Ihrer Funktion eine Bedingung hinzu, die die `s3:PutObject` Berechtigungen auf Buckets in Ihrem Konto beschränkt.

-Das folgende Beispiel zeigt eine IAM-Richtlinie, die die `s3:PutObject`-Berechtigungen Ihrer Funktion auf Buckets in Ihrem Konto beschränkt. Diese Richtlinie gibt Lambda auch die `s3:ListBucket`-Erlaubnis, einen S3-Bucket als Ziel zu verwenden.

```
{
    "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"
                }
            }
        }
    ]
}
```

Um der Ausführungsrolle Ihrer Funktion mithilfe von AWS-Managementkonsole oder eine Berechtigungsrichtlinie hinzuzufügen AWS CLI, lesen Sie die Anweisungen in den folgenden Verfahren:

Console

So fügen Sie der Ausführungsrolle einer Funktion (Konsole) eine Berechtigungsrichtlinie hinzu

1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.
2. Wählen Sie die Lambda-Funktion aus, deren Ausführungsrolle Sie ändern möchten.
3. Klicken Sie in der Registerkarte Konfiguration auf die Option Berechtigungen.
4. Wählen Sie auf der Registerkarte Ausführungsrolle den Rollennamen Ihrer Funktion aus, um die IAM-Konsolenseite der Rolle zu öffnen.
5. Fügen Sie der Rolle wie folgt eine Richtlinie mit Berechtigungen hinzu:
    a. Wählen Sie im Bereich Berechtigungsrichtlinien die Optionen Berechtigungen hinzufügen und dann Inline-Richtlinie erstellen aus.
    b. Wählen Sie im Richtlinien-Editor JSON aus.
    c. Fügen Sie die Richtlinie, die Sie hinzufügen möchten, in den Editor ein (indem Sie die vorhandene JSON-Datei ersetzt) und wählen Sie dann Weiter aus.
    d. Geben Sie unter Richtliniendetails für den Richtliniennamen ein.
    e. Wählen Sie Richtlinie erstellen aus.

AWS CLI

So fügen Sie der Ausführungsrolle einer Funktion (CLI) eine Berechtigungsrichtlinie hinzu

1. Erstellen Sie ein JSON-Richtliniendokument mit den erforderlichen Berechtigungen und speichern Sie es in einem lokalen Verzeichnis.
2. Verwenden Sie den `put-role-policy` IAM-CLI-Befehl, um die Berechtigungen zur Ausführungsrolle Ihrer Funktion hinzuzufügen. Führen Sie den folgenden Befehl in dem Verzeichnis aus, in dem Sie Ihr JSON-Richtliniendokument gespeichert haben und ersetzen Sie den Rollennamen, den Richtliniennamen und das Richtliniendokument durch Ihre eigenen Werte.

```
aws iam put-role-policy \
--role-name my_lambda_role \
--policy-name LambdaS3DestinationPolicy \
--policy-document file://my_policy.json
```

Beispiel für Aufrufdatensatz

Wenn ein Aufruf mit der Bedingung übereinstimmt, sendet Lambda ein JSON-Dokument mit Details zum Aufruf an das Ziel. Das folgende Beispiel zeigt einen Aufrufdatensatz für ein Ereignis, bei dem drei Verarbeitungsversuche aufgrund eines Funktionsfehlers fehlgeschlagen sind.

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"
    }
}
```

Der Aufrufdatensatz enthält Details zum Ereignis, zur Antwort und zu dem Grund, warum der Datensatz gesendet wurde.

Zurückverfolgen von Anforderungen zu Zielen

Sie können AWS X-Ray verwenden, um eine verbundene Ansicht jeder Anforderung zu sehen, während sie in die Warteschlange gestellt, von einer Lambda-Funktion verarbeitet und an den Zielservice übergeben wird. Wenn Sie X-Ray-Tracing für eine Funktion oder einen Service aktivieren, der eine Funktion aufruft, fügt Lambda der Anforderung einen X-Ray-Header hinzu und übergibt den Header an den Zielservice. Traces von Upstream-Services werden automatisch mit Traces von Downstream-Lambda-Funktionen und Zieldiensten verknüpft, sodass eine end-to-end Ansicht der gesamten Anwendung entsteht. Weitere Informationen zum Tracing finden Sie unter Visualisieren Sie Lambda-Funktionsaufrufe mit AWS X-Ray.

Hinzufügen einer Warteschlange für unzustellbare Nachrichten

Als eine Alternativ zu einem Zielort bei Ausfall können Sie Ihre Funktion mit einer Warteschlangen für unzustellbare Nachrichten konfigurieren, um verworfene Ereignisse zur weiteren Verarbeitung zu speichern. Eine Warteschlange für unzustellbare Nachrichten fungiert genauso wie ein Ziel bei Ausfall, da sie verwendet wird, wenn alle Verarbeitungsversuche eines Ereignisses fehlschlagen oder ein Ereignis abläuft, ohne verarbeitet zu werden. Sie können jedoch nur auf Funktionsebene eine Warteschlange für unzustellbare Nachrichten hinzufügen oder entfernen. Funktionsversionen verwenden dieselben Einstellungen für Warteschlangen für unzustellbare Nachrichten wie die unveröffentlichte Version ($LATEST). Ausfallziele unterstützen auch zusätzliche Ziele und enthalten Details zur Reaktion der Funktion im Aufrufdatensatz.

Um Ereignisse in einer Warteschlange für unzustellbare Nachrichten erneut zu verarbeiten, können Sie sie als Ereignisquelle für Ihre Lambda-Funktion festlegen. Alternativ können Sie die Ereignisse manuell abrufen.

Sie können eine Amazon-SQS-Standardwarteschlange oder ein Amazon-SNS-Standardthema für Ihre Warteschlange für unzustellbare Nachrichten auswählen. FIFO-Warteschlangen und Amazon-SNS-FIFO-Themen werden nicht unterstützt.

- Amazon-SQS-Warteschlange – Eine Warteschlange nimmt fehlgeschlagene Ereignisse auf, bis sie abgerufen werden. Wählen Sie eine Amazon SQS SQS-Standardwarteschlange, wenn Sie erwarten, dass eine einzelne Entität, z. B. eine Lambda-Funktion oder ein CloudWatch Alarm, das fehlgeschlagene Ereignis verarbeitet. Weitere Informationen finden Sie unter Verwenden von Lambda mit Amazon SQS.
- Amazon-SNS-Thema – Ein Thema leitet fehlgeschlagene Ereignisse an eine oder mehrere Ziele. Wählen Sie ein Amazon-SNS-Standardthema aus, wenn Sie erwarten, dass mehrere Entitäten auf ein fehlgeschlagenes Ereignis reagieren. Sie können beispielsweise ein Thema so konfigurieren, dass Ereignisse an eine E-Mail-Adresse, eine Lambda-Funktion oder and/or einen HTTP-Endpunkt gesendet werden. Weitere Informationen finden Sie unter Aufrufen von Lambda-Funktionen mit Amazon-SNS-Benachrichtigungen.

Um Ereignisse an eine Warteschlange oder an ein Thema zu senden, benötigt Ihre Funktion zusätzliche Berechtigungen. Fügen Sie eine Richtlinie mit den erforderlichen Berechtigungen zur Ausführungsrolle Ihrer Funktion hinzu. Wenn die Zielwarteschlange oder das Zielthema mit einem vom Kunden verwalteten AWS KMS Schlüssel verschlüsselt ist, stellen Sie sicher, dass sowohl die Ausführungsrolle Ihrer Funktion als auch die ressourcenbasierte Richtlinie des Schlüssels die entsprechenden Berechtigungen enthalten.

Nachdem das Ziel erstellt und die Ausführungsrolle Ihrer Funktion aktualisiert wurde, fügen Sie die Warteschlange für unzustellbare Nachrichten zu Ihrer Funktion hinzu. Sie können mehrere Funktionen zum Senden von Ereignissen an dasselbe Ziel konfigurieren.

Console

1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.
2. Wählen Sie eine Funktion aus.
3. Wählen Sie „Konfiguration“ und dann „Asynchroner Aufruf“ aus.
4. Wählen Sie unter Asynchronous invocation (Asynchroner Aufruf) die Option Edit (Bearbeiten).

5. Stellen Sie den Warteschlangenservice für unzustellbare Nachrichten auf Amazon SQS oder Amazon SNS ein.
6. Wählen Sie die Ziel-Warteschlange oder das Ziel-Thema aus.
7. Wählen Sie Speichern.

AWS CLI

Verwenden Sie den Befehl, um eine Warteschlange mit unerlaubten Briefen zu AWS CLI konfigurieren. update-function-configuration

```
aws lambda update-function-configuration \
  --function-name my-function \
  --dead-letter-config TargetArn=arn:aws:sns:us-east-1:123456789012:my-topic
```

Lambda sendet das Ereignis mit zusätzlichen Informationen in den Attributen an die Warteschlange für unzustellbare Nachrichten. Sie können anhand dieser Informationen den von der Funktion zurückgegebenen Fehler erkennen oder das Ereignis mit Protokollen oder einer AWS X-Ray - Ablaufverfolgung korrelieren.

Nachrichtenattribute der Warteschlange für unzustellbare Nachrichten

- RequestID (Zeichenfolge) – Die ID der Aufrufanforderung. Die Anfrage IDs wird in den Funktionsprotokollen angezeigt. Sie können das X-Ray-SDK auch verwenden, um die Anforderungs-ID für ein Attribut in der Ablaufverfolgung aufzuzeichnen. Anschließend können Sie anhand der Anforderungs-ID in der X-Ray-Konsole nach Spuren suchen.
- ErrorCode(Nummer) — Der HTTP-Statuscode.
- ErrorMessage(String) — Die ersten 1 KB der Fehlermeldung.

Wenn Lambda keine Nachricht an die Warteschlange für unzustellbare Briefe senden kann, löscht es das Ereignis und gibt die Metrik aus. DeadLetterErrors Dies kann bei mangelnden Berechtigungen passieren oder wenn die Gesamtgröße der Nachricht den Grenzwert der Zielwarteschlange oder des Ziel-Themas überschreitet. Nehmen wir zum Beispiel an, dass eine Amazon-SNS-Benachrichtigung mit einem Text von fast 1 MB eine Funktion auslöst, die zu einem Fehler führt. In diesem Fall können die Ereignisdaten, die Amazon SNS hinzufügt, in Kombination mit den Attributen, die Lambda hinzufügt, dazu führen, dass die Nachricht die maximale Größe überschreitet, die in der Warteschlange für unzustellbare Nachrichten zulässig ist.

Wenn Sie Amazon SQS als Ereignisquelle verwenden, konfigurieren Sie eine Warteschlange für unzustellbare Nachrichten für die Amazon-SQS-Warteschlange selbst und nicht für die Lambda-Funktion. Weitere Informationen finden Sie unter Verwenden von Lambda mit Amazon SQS.

Dauerhafte Lambda-Funktionen aufrufen

Dauerhafte Lambda-Funktionen können mit den gleichen Methoden wie standardmäßige Lambda-Funktionen aufgerufen werden, allerdings mit wichtigen Überlegungen für lang andauernde Ausführungen. In diesem Abschnitt werden Aufrufmuster, Ausführungsmanagement und bewährte Methoden für langlebige Funktionen behandelt.

Grenzwerte für synchrone Aufrufe

Synchrone Aufrufe dauerhafter Lambda-Funktionen sind auf 15 Minuten begrenzt, genau wie bei Standard-Lambda-Funktionen. Wenn Ihre dauerhafte Funktion länger als 15 Minuten ausgeführt werden muss, muss sie asynchron aufgerufen werden.

Wann sollte der synchrone Aufruf verwendet werden? Verwenden Sie diesen Befehl für dauerhafte Funktionen, die innerhalb von 15 Minuten abgeschlossen werden, und wenn Sie sofortige Ergebnisse benötigen, z. B. schnelle Genehmigungsworkflows oder kurze Datenverarbeitungsaufgaben.

Asynchroner Aufruf für Workflows mit langer Laufzeit

Verwenden Sie für dauerhafte Funktionen, die länger als 15 Minuten ausgeführt werden können, den asynchronen Aufruf. Dadurch kann die Funktion weiter ausgeführt werden, während Ihr Client sofort eine Bestätigung erhält.

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'})
)
```

Verwaltung der Ausführung APIs

Lambda ermöglicht APIs die Verwaltung und Überwachung dauerhafter Funktionsausführungen, einschließlich der Auflistung von Ausführungen, dem Abrufen des Ausführungsstatus und dem Stoppen laufender Ausführungen.

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'
    })
)
```

Wie Lambda Datensätze aus Stream- und warteschlangenbasierten Ereignisquellen verarbeitet

Ein Zuordnung von Ereignisquellen ist eine Lambda-Ressource, die Elemente aus Stream- und Warteschlangen-basierten Diensten liest und eine Funktion mit Stapeln von Datensätzen aufruft. Innerhalb einer Zuordnung von Ereignisquellen fragen Ressourcen, sogenannte Event-Poller, aktiv nach neuen Nachrichten ab und rufen Funktionen auf. Standardmäßig skaliert Lambda Ereignisabfragen automatisch, aber für bestimmte Ereignisquellentypen können Sie den Bereitstellungsmodus verwenden, um die Mindest- und Höchstanzahl von Event-Pollern zu steuern, die für Ihre Zuordnung von Ereignisquellen vorgesehen sind.

Die folgenden Dienste verwenden Zuordnungen von Ereignisquellen, um Lambda-Funktionen aufzurufen:

- Amazon DocumentDB (mit MongoDB-Kompatibilität) (Amazon DocumentDB)
- Amazon-DynamoDB
- Amazon Kinesis
- Amazon MQ
- Amazon Managed Streaming for Apache Kafka (Amazon MSK)
- Selbstverwaltetes Apache Kafka
- Amazon-Simple-Queue-Service (Amazon SQS)

> **Warning**
>
> Zuordnung von Lambda-Ereignisquellen verarbeiten jedes Ereignis mindestens einmal und es kann zu einer doppelten Verarbeitung von Datensätzen kommen. Um mögliche Probleme im Zusammenhang mit doppelten Ereignissen zu vermeiden, empfehlen wir Ihnen dringend, Ihren Funktionscode idempotent zu machen. Weitere Informationen finden Sie im Knowledge Center unter Wie mache ich meine Lambda-Funktion idempotent?. AWS

Wie unterscheiden sich Zuordnungen von Ereignisquellen von direkten Auslösern

Einige AWS-Services können Lambda-Funktionen mithilfe von Triggern direkt aufrufen. Diese Dienste leiten Ereignisse an Lambda weiter und die Funktion wird sofort aufgerufen, wenn das angegebene Ereignis eintritt. Trigger eignen sich für diskrete Ereignisse und die Verarbeitung in Echtzeit. Wenn Sie mit der Lambda-Konsole einen Trigger erstellen, interagiert die Konsole mit dem entsprechenden AWS Dienst, um die Ereignisbenachrichtigung für diesen Dienst zu konfigurieren. Der Auslöser wird tatsächlich von dem Dienst gespeichert und verwaltet, der die Ereignisse generiert, nicht von Lambda. Hier sind einige Beispiele für Dienste, die Auslöser zum Aufrufen von Lambda-Funktionen verwenden:

- Amazon Simple Storage Service (Amazon S3): Ruft eine Funktion auf, wenn ein Objekt in einem Bucket erstellt, gelöscht oder geändert wird. Weitere Informationen finden Sie unter Tutorial: Verwenden eines Amazon-S3-Auslösers zum Aufrufen einer Lambda-Funktion.
- Amazon Simple Notification Service (Amazon SNS): Ruft eine Funktion auf, wenn eine Nachricht in einem SNS-Thema veröffentlicht wird. Weitere Informationen finden Sie unter Tutorial: Verwendung AWS Lambda mit Amazon Simple Notification Service.
- Amazon API Gateway: Ruft eine Funktion auf, wenn eine API-Anfrage an einen bestimmten Endpunkt gestellt wird. Weitere Informationen finden Sie unter Aufrufen einer Lambda-Funktion über einen Amazon API Gateway-Endpunkt.

Zuordnungen von Ereignisquellen sind Lambda-Ressourcen, die innerhalb des Lambda-Service erstellt und verwaltet werden. Zuordnungen von Ereignisquellen sind für die Verarbeitung umfangreicher Streaming-Daten oder Nachrichten aus Warteschlangen konzipiert. Die stapelweise Verarbeitung von Datensätzen aus einem Stream oder einer Warteschlange ist effizienter als die Verarbeitung einzelner Datensätze.

Batching-Verhalten

Standardmäßig batcht eine Ereignisquellenzuordnung Datensätze in einer einzigen Nutzlast, die Lambda an Ihre Funktion sendet. Um das Batching-Verhalten zu optimieren, können Sie ein Batching-Fenster (MaximumBatchingWindowInSeconds) und eine Batch-Größe (BatchSize) konfigurieren. Ein Batch-Fenster ist die maximale Zeitspanne zur Erfassung von Datensätzen in einer einzigen Nutzlast. Eine Batch-Größe ist die maximale Anzahl von Datensätzen in einem einzigen Batch. Lambda ruft Ihre Funktion auf, wenn eines der folgenden drei Kriterien erfüllt ist:

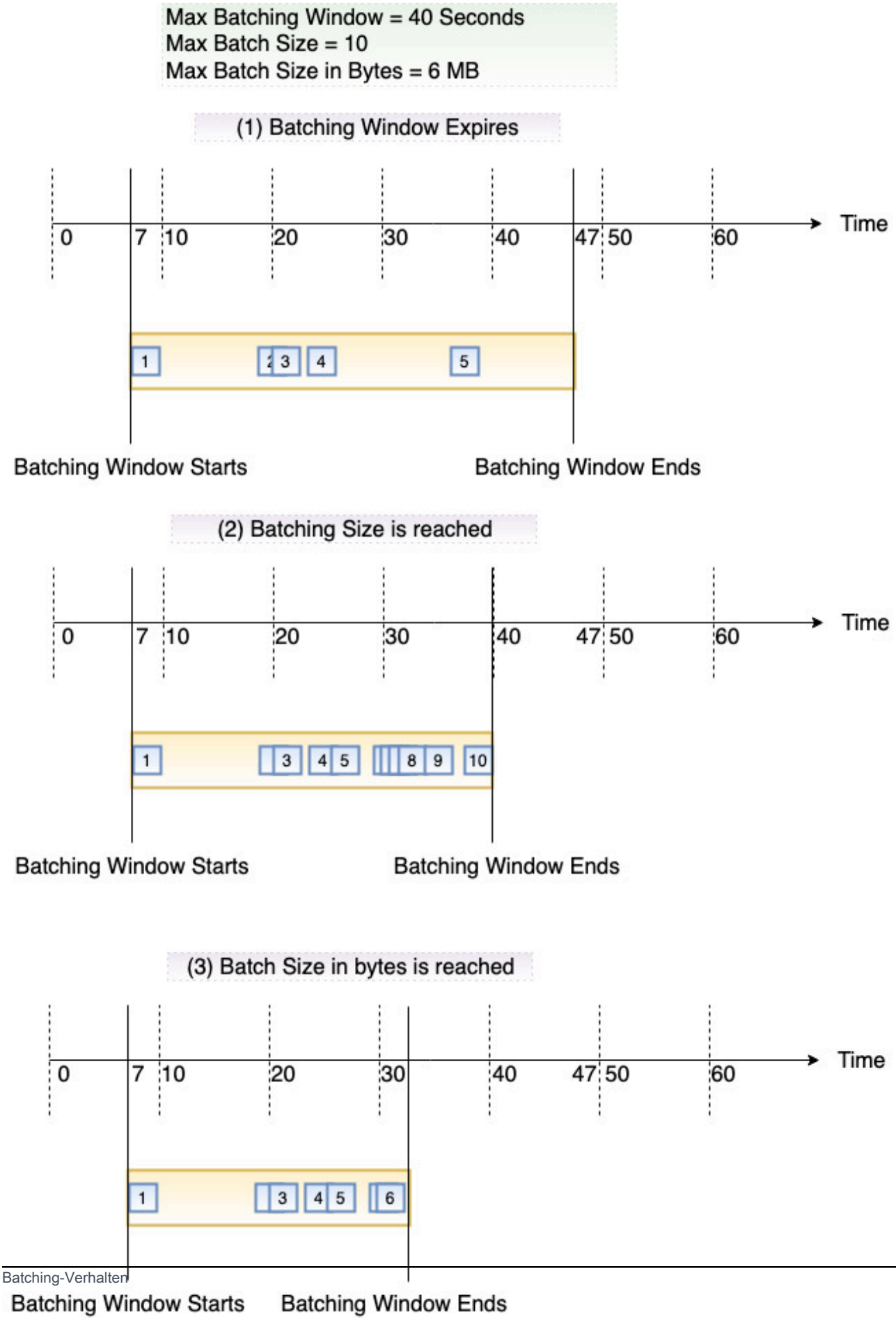
- Das Batching-Fenster erreicht seinen Maximalwert. Das standardmäßige Batch-Fensterverhalten unterscheidet sich je nach Ereignisquelle.
  - Für Kinesis-, DynamoDB- und Amazon SQS SQS-Ereignisquellen: Das Standard-Batch-Fenster beträgt 0 Sekunden. Das bedeutet, dass Lambda Ihre Funktion aufruft, sobald Datensätze verfügbar sind. Um ein Batching-Fenster festzulegen, konfigurieren Sie `MaximumBatchingWindowInSeconds`. Sie können diesen Parameter auf einen beliebigen Wert zwischen 0 und 300 Sekunden in Schritten von 1 Sekunde einstellen. Wenn Sie ein Stapelverarbeitungsfenster konfigurieren, beginnt das nächste Fenster, sobald der vorherige Funktionsaufruf abgeschlossen ist.
  - Für Amazon-MSK-, selbstverwaltete Apache-Kafka-, Amazon-MQ- und Amazon-DocumentDB-Ereignisquellen: Das standardmäßige Batching-Fenster beträgt 500 ms. Sie können `MaximumBatchingWindowInSeconds` auf einen beliebigen Wert von 0 Sekunden bis 300 Sekunden in Sekundenschritten einstellen. Wenn Sie im Bereitstellungsmodus für Kafka-Zuordnungen von Ereignisquellen ein Batching-Fenster konfigurieren, beginnt das nächste Fenster, sobald der vorherige Batch abgeschlossen ist. Wenn Sie ein Batching-Fenster konfigurieren, beginnt das nächste Fenster, sobald der vorherige Funktionsaufruf abgeschlossen ist. Um die Latenz bei der Verwendung von Kafka-Zuordnungen von Ereignisquellen im Bereitstellungsmodus zu minimieren, setzen Sie `MaximumBatchingWindowInSeconds` auf 0. Diese Einstellung stellt sicher, dass Lambda unmittelbar nach Abschluss des aktuellen Funktionsaufrufs mit der Verarbeitung des nächsten Batches beginnt. Weitere Informationen zur Verarbeitung mit niedriger Latenz finden Sie unter Niedrige Latenz für Apache-Kafka.
  - Für die Ereignisquellen Amazon MQ und Amazon DocumentDB: Das standardmäßige Batching-Fenster beträgt 500 ms. Sie können `MaximumBatchingWindowInSeconds` auf einen beliebigen Wert von 0 Sekunden bis 300 Sekunden in Sekundenschritten einstellen. Ein Batch-Fenster beginnt, sobald der erste Datensatz eintrifft.

    > **Note**
    >
    > Da Sie `MaximumBatchingWindowInSeconds` nur in Sekundenschritten ändern können, können Sie nicht zu dem Standard-Batch-Fenster von 500 ms zurückkehren, nachdem Sie es geändert haben. Um das Standard-Batch-Fenster wiederherzustellen, müssen Sie eine neue Ereignisquellenzuordnung erstellen.

- Die Batch-Größe wird erreicht. Die minimale Batch-Größe beträgt 1. Die Standard- und die maximale Batch-Größe hängen von der Ereignisquelle ab. Weitere Informationen zu diesen Werten finden Sie unter BatchSize-Spezifikation für die `CreateEventSourceMapping`-API-Operation.

- Die Nutzlastgröße erreicht 6 MB. Sie können dieses Limit nicht ändern.

Das folgende Diagramm verdeutlicht diese Bedingungen. Angenommen, ein Batch-Fenster beginnt bei `t = 7` Sekunden. Im ersten Szenario erreicht das Batch-Fenster sein Maximum von 40 Sekunden bei `t = 47` Sekunden nach dem Erfassen von 5 Datensätzen. Im zweiten Szenario erreicht die Batch-Größe 10, bevor das Batch-Fenster abläuft, sodass das Batch-Fenster früh endet. Im dritten Szenario wird die maximale Nutzlastgröße erreicht, bevor das Batch-Fenster abläuft, sodass das Batch-Fenster frühzeitig endet.


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

Wir empfehlen Ihnen, mit verschiedenen Stapel- und Datensatzgrößen zu testen, damit die Abfragefrequenz jeder Ereignisquelle darauf abgestimmt ist, wie schnell Ihre Funktion ihre Aufgabe erledigen kann. Der CreateEventSourceMapping`BatchSize`Parameter steuert die maximale Anzahl von Datensätzen, die bei jedem Aufruf an Ihre Funktion gesendet werden können. Eine höhere Stapelgröße kann den mit dem Aufruf-Overhead über eine größere Datensatzgruppe hinweg oft effizienter verarbeiten und Ihren Durchsatz erhöhen.

Lambda wartet mit dem Senden des nächsten zu verarbeitenden Stapels nicht, bis ggf. konfigurierte Erweiterungen abgeschlossen sind. Anders ausgedrückt: Ihre Erweiterungen werden möglicherweise weiter ausgeführt, während Lambda den nächsten Stapel von Datensätzen verarbeitet. Dies kann zu Drosselungsproblemen führen, wenn Sie gegen eine Einstellung oder gegen einen Grenzwert im Zusammenhang mit der Parallelität Ihres Kontos verstoßen. Um zu erkennen, ob möglicherweise ein Problem vorliegt, müssen Sie Ihre Funktionen überwachen sowie überprüfen, ob für Ihre Zuordnung von Ereignisquellen unerwartet hohe Parallelitätsmetriken vorliegen. Aufgrund der kurzen Zeit zwischen den Aufrufen kann Lambda kurzzeitig eine höhere Gleichzeitigkeitsnutzung als die Anzahl der Shards melden. Dies kann sogar für Lambda-Funktionen ohne Erweiterungen gelten.

Wenn Ihre Funktion einen Fehler zurückgibt, verarbeitet die Ereignisquellenzuordnung standardmäßig das gesamte Batch erneut, bis die Funktion erfolgreich ist oder die Elemente im Batch ablaufen. Um eine ordnungsgemäße Verarbeitung zu gewährleisten, unterbricht die Ereignisquellenzuordnung die Verarbeitung für den betroffenen Shard, bis der Fehler behoben ist. Für Stream-Quellen (DynamoDB und Kinesis) können Sie die maximale Anzahl von Wiederholungsversuchen von Lambda konfigurieren, wenn Ihre Funktion einen Fehler zurückgibt. Service-Fehler oder Drosselungen, bei denen der Stapel Ihre Funktion nicht erreicht, zählen nicht zu den Wiederholungsversuchen. Sie können die Zuordnung von Ereignisquellen auch so konfigurieren, dass ein Aufrufdatensatz an ein Ziel gesendet wird, wenn ein Ereignisbatch verworfen wird.

Modus bereitgestellter Kapazität

Lambda-Ereignisquellenzuordnungen verwenden Ereignisabfragen, um Ihre Ereignisquelle nach neuen Nachrichten abzufragen. Standardmäßig verwaltet Lambda die automatische Skalierung dieser Poller auf der Grundlage des Nachrichtenvolumens. Wenn der Nachrichtenverkehr zunimmt, erhöht Lambda automatisch die Anzahl der Event-Poller, um die Last zu bewältigen und reduziert sie, wenn der Verkehr abnimmt.

Im Bereitstellungsmodus können Sie den Durchsatz Ihrer Ereignisquellenzuordnung feinabstimmen, indem Sie Mindest- und Höchstgrenzen für dedizierte Abfrageressourcen definieren, die bereit sind, die erwarteten Datenverkehrsmuster zu verarbeiten. Diese Ressourcen werden dreimal schneller

automatisch skaliert, um plötzliche Spitzen im Ereignisverkehr zu bewältigen, und bieten eine 16-mal höhere Kapazität zur Verarbeitung von Millionen von Ereignissen. Auf diese Weise können Sie reaktionsschnelle ereignisgesteuerte Workloads mit strengen Leistungsanforderungen erstellen.

In Lambda ist ein Event Poller eine Recheneinheit mit Durchsatzmöglichkeiten, die je nach Art der Ereignisquelle variieren. Bei Amazon MSK und selbstverwaltetem Apache Kafka kann jeder Event Poller bis zu 5% MB/sec des Durchsatzes oder bis zu 5 gleichzeitige Aufrufe verarbeiten. Wenn Ihre Ereignisquelle beispielsweise eine durchschnittliche Nutzlast von 1 MB erzeugt und die durchschnittliche Dauer Ihrer Funktion 1 Sekunde beträgt, kann ein einziger Kafka-Event-Poller 5 MB/sec Durchsatz- und 5 gleichzeitige Lambda-Aufrufe unterstützen (vorausgesetzt, es gibt keine Payload-Transformation). Für Amazon SQS kann jeder Event Poller bis zu 1 MB/sec Durchsatz oder bis zu 10 gleichzeitige Aufrufe verarbeiten. Bei der Verwendung des Bereitstellungsmodus fallen zusätzliche Kosten an, die von Ihrer Nutzung des Event-Pollers abhängen. Details zu den Preisen finden Sie unter AWS Lambda -Preise.

Der Bereitstellungsmodus ist für Amazon MSK, selbstverwaltete Apache Kafka- und Amazon SQS SQS-Ereignisquellen verfügbar. Während Sie mit den Gleichzeitigkeitseinstellungen die Skalierung Ihrer Funktion steuern können, haben Sie im Bereitstellungsmodus die Kontrolle über den Durchsatz Ihrer Zuordnung von Ereignisquellen. Um eine maximale Leistung zu gewährleisten, müssen Sie möglicherweise beide Einstellungen unabhängig voneinander anpassen.

Der Bereitstellungsmodus ist ideal für Echtzeitanwendungen, die eine konstante Latenz bei der Ereignisverarbeitung erfordern, z. B. Finanzdienstleister, die Marktdatenfeeds verarbeiten, E-Commerce-Plattformen, die personalisierte Empfehlungen in Echtzeit bereitstellen, und Spieleunternehmen, die Live-Spielerinteraktionen verwalten.

Jeder Event Poller unterstützt unterschiedliche Durchsatzkapazitäten:

- Für Amazon MSK und selbstverwalteten Apache Kafka: bis zu 5% Durchsatz oder bis zu 5 MB/sec gleichzeitige Aufrufe
- Für Amazon SQS: bis zu 1 MB/sec Durchsatz oder bis zu 10 gleichzeitige Aufrufe oder bis zu 10 SQS-Polling-API-Aufrufe pro Sekunde.

Für Amazon SQS SQS-Ereignisquellenzuordnungen können Sie die Mindestanzahl von Pollern zwischen 2 und 200 mit einer Standardeinstellung von 2 und die maximale Anzahl zwischen 2 und 2.000 mit einer Standardeinstellung von 200 festlegen. Lambda skaliert die Anzahl der Event-Poller zwischen Ihrem konfigurierten Minimum und Maximum und fügt schnell bis zu 1.000 Parallelität

pro Minute hinzu, um eine konsistente Verarbeitung Ihrer Ereignisse mit niedriger Latenz zu gewährleisten.

Für Kafka-Ereignisquellenzuordnungen können Sie die Mindestanzahl von Pollern zwischen 1 und 200 mit dem Standardwert 1 und die Höchstanzahl zwischen 1 und 2.000 mit einem Standardwert von 200 festlegen. Lambda skaliert die Anzahl der Event-Poller auf der Grundlage Ihres Event-Backlogs in Ihrem Thema zwischen Ihrem konfigurierten Minimum und Maximum, um eine Verarbeitung Ihrer Ereignisse mit geringer Latenz zu ermöglichen.

Beachten Sie, dass für Amazon SQS SQS-Ereignisquellen die Einstellung für maximale Parallelität nicht im Bereitstellungsmodus verwendet werden kann. Wenn Sie den Bereitstellungsmodus verwenden, steuern Sie die Parallelität über die Einstellung „Maximale Anzahl von Event-Pollers“.

Weitere Informationen zur Konfiguration des Bereitstellungsmodus finden Sie in folgenden Abschnitten:

- Konfiguration des Bereitstellungsmodus für Amazon MSK-Ereignisquellenzuordnungen
- Konfiguration des Bereitstellungsmodus für selbstverwaltete Apache-Kafka-Ereignisquellen-Zuordnungen
- Verwenden des Bereitstellungsmodus mit Amazon SQS SQS-Ereignisquellenzuordnungen

Um die Latenz im Bereitstellungsmodus zu minimieren, setzen Sie den Wert auf 0. `MaximumBatchingWindowInSeconds` Diese Einstellung stellt sicher, dass Lambda unmittelbar nach Abschluss des aktuellen Funktionsaufrufs mit der Verarbeitung des nächsten Batches beginnt. Weitere Informationen zur Verarbeitung mit niedriger Latenz finden Sie unter Niedrige Latenz für Apache-Kafka.

Nachdem Sie den Bereitstellungsmodus konfiguriert haben, können Sie die Verwendung von Event-Pollern für Ihren Workload beobachten, indem Sie die `ProvisionedPollers`-Metrik überwachen. Weitere Informationen finden Sie unter the section called “Metriken zur Zuordnung von Ereignisquellen”.

API für die Ereignisquellenzuordnung

Um eine Ereignisquelle mit der AWS Command Line Interface (AWS CLI) oder einem AWS -SDK zu verwalten, können Sie die folgenden API-Operationen verwenden:

- CreateEventSourceMapping

- ListEventSourceMappings
- GetEventSourceMapping
- UpdateEventSourceMapping
- DeleteEventSourceMapping

Verwendung von Tags für Zuordnungen von Ereignisquellen

Sie können Zuordnungen von Ereignisquellen taggen, um Ihre Ressourcen zu organisieren und zu verwalten. Tags sind frei formulierte Schlüssel-Wert-Paare, die mit Ihren Ressourcen verknüpft sind und von AWS-Services unterstützt werden. Weitere Informationen zu Anwendungsfällen für Tags finden Sie unter Allgemeine Tagging-Strategien im Tagging AWS Resources and Tag Editor Guide.

Zuordnungen von Ereignisquellen sind mit Funktionen verknüpft, die ihre eigenen Tags haben können. Zuordnungen von Ereignisquellen erben nicht automatisch Tags von Funktionen. Sie können die AWS Lambda API verwenden, um Tags anzuzeigen und zu aktualisieren. Sie können auch Tags anzeigen und aktualisieren, während Sie eine bestimmte Ereignisquellenzuordnung in der Lambda-Konsole verwalten, einschließlich solcher, die den Bereitstellungsmodus für Amazon SQS verwenden.

Erforderliche Berechtigungen zum Arbeiten mit Tags

Um einer AWS Identity and Access Management -(IAM)-Identität (Benutzer, Gruppe oder Rolle) das Lesen oder Setzen von Tags auf einer Ressource zu ermöglichen, gewähren Sie ihr die entsprechenden Berechtigungen:

- lambda: ListTags —Wenn eine Ressource Tags enthält, gewähren Sie diese Berechtigung jedem, der sie aufrufen muss. `ListTags` Für Funktionen mit Tags ist diese Berechtigung auch für `GetFunction` erforderlich.
- lambda: TagResource —Erteilen Sie diese Berechtigung jedem, der bei create ein Tag aufrufen `TagResource` oder ausführen muss.

Optional können Sie erwägen, auch die lambda: UntagResource -Berechtigung zu erteilen, um `UntagResource` Aufrufe der Ressource zuzulassen.

Weitere Informationen finden Sie unter Identitätsbasierte IAM-Richtlinien für Lambda.

Verwendung von Tags mit der Lambda-Konsole

Sie können die Lambda-Konsole verwenden, um Ereignisquellenzuordnungen mit Tags zu erstellen, Tags zu vorhandenen Ereignisquellenzuordnungen hinzuzufügen und Ereignisquellenzuordnungen nach Tag zu filtern, einschließlich der im Bereitstellungsmodus für Amazon SQS konfigurierten.

Wenn Sie einen Auslöser für unterstützte Stream- und Warteschlangen-basierte Dienste über die Lambda-Konsole hinzufügen, erstellt Lambda automatisch eine Zuordnung von Ereignisquellen. Weitere Informationen zu diesen Ereignisquellen finden Sie unter the section called “Zuweisung von Ereignisquellen”. Sie benötigen die folgenden Voraussetzungen, um eine Zuordnung von Ereignisquellen in der Konsole zu erstellen:

- Eine -Funktion
- Eine Ereignisquelle von einem betroffenen Dienst.

Sie können die Tags als Teil derselben Benutzeroberfläche hinzufügen, die Sie zum Erstellen oder Aktualisieren von Triggern verwenden.

So fügen Sie ein Tag hinzu, wenn Sie eine Zuordnung von Ereignisquellen erstellen

1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.
2. Wählen Sie den Namen Ihrer Funktion aus.
3. Wählen Sie unter Function overview (Funktionsübersicht) die Option Add trigger (Trigger hinzufügen).
4. Wählen Sie unter Auslöserkonfiguration in der Dropdownliste den Namen des Dienstes aus, aus dem Ihre Ereignisquelle stammt.
5. Geben Sie die Kernkonfiguration für Ihre Ereignisquelle an. Weitere Informationen zur Konfiguration Ihrer Ereignisquelle finden Sie im Abschnitt für den entsprechenden Dienst unter Integration anderer Services.
6. Wählen Sie unter Konfiguration der Zuordnung von Ereignisquellen die Option Zusätzliche Einstellungen aus.
7. Wählen Sie unter Tags die Option Neuen Tag hinzufügen aus
8. Geben Sie im Feld Schlüssel Ihren Tag-Schlüssel ein. Informationen zu Tagging-Einschränkungen finden Sie unter Beschränkungen und Anforderungen für die Benennung von Tags im Tagging Resources and Tag Editor Guide. AWS
9. Wählen Sie Hinzufügen aus.

So fügen Sie Tags zu einer bestehenden Zuordnung von Ereignisquellen hinzu

1. Öffnen Sie die Zuordnungen von Ereignisquellen in der Lambda-Konsole.
2. Wählen Sie aus der Ressourcenliste die UUID für die Zuordnung von Ereignisquellen aus, die Ihrem Funktions- und Ereignisquellen-ARN entspricht.
3. Wählen Sie in der Registerkartenliste unter dem Konfigurationsbereich Allgemein die Option Tags.
4. Wählen Sie Tags verwalten aus.
5. Wählen Sie Neues Tag hinzufügen aus.
6. Geben Sie im Feld Schlüssel Ihren Tag-Schlüssel ein. Informationen zu Einschränkungen beim Taggen finden Sie unter Beschränkungen und Anforderungen für die Benennung von Tags im Leitfaden für AWS Tag-Ressourcen und Tag-Editor.
7. Wählen Sie Speichern.

So filtern Sie Zuordnungen von Ereignisquellen nach Tag

1. Öffnen Sie die Zuordnungen von Ereignisquellen in der Lambda-Konsole.
2. Wählen Sie das Suchfeld.
3. Wählen Sie in der Dropdown-Liste Ihren Tagschlüssel unter der Tag-Unterüberschrift aus.
4. Wählen Sie Verwenden: „Tag-Name“, um alle Zuordnungen von Ereignisquellen zu sehen, die mit dieser Taste gekennzeichnet sind oder wählen Sie einen Operator, um weiter nach Werten zu filtern.
5. Wählen Sie Ihren Tag-Wert aus, um nach einer Kombination aus Tag-Schlüssel und -Wert zu filtern.

Das Suchfeld unterstützt auch die Suche nach Tag-Schlüsseln. Geben Sie den Namen eines Schlüssels ein, um ihn in der Liste zu finden.

Verwenden von Tags mit dem AWS CLI

Mit der Lambda-API können Sie Tags zu vorhandenen Lambda-Ressourcen, einschließlich Zuordnungen von Ereignisquellen, hinzufügen und entfernen. Sie können beim Erstellen einer Zuordnung von Ereignisquellen auch Tags hinzufügen, sodass Sie eine Ressource während ihres gesamten Lebenszyklus mit Tags versehen können.

Aktualisieren von Tags mit dem Lambda-Tag APIs

Sie können Tags für unterstützte Lambda-Ressourcen über die TagResourceund UntagResourceAPI-Operationen hinzufügen und entfernen.

Sie können diese Operationen mit der Taste AWS CLI aufrufen. Um Tags einer vorhandenen Ressource hinzuzufügen, verwenden Sie den folgenden `tag-resource`-Befehl. In diesem Beispiel werden zwei Tags hinzugefügt, eines mit dem Schlüssel *`Department`* und eines mit dem Schlüssel*`CostCenter`*.

```
aws lambda tag-resource \
--resource arn:aws:lambda:us-east-2:123456789012:resource-type:my-resource \
--tags Department=Marketing,CostCenter=1234ABCD
```

Mit dem Befehl `untag-resource` können Sie Tags entfernen. In diesem Beispiel wird das Tag mit dem Schlüssel entfernt*`Department`*.

```
aws lambda untag-resource --resource arn:aws:lambda:us-east-1:123456789012:resource-
type:resource-identifier \
--tag-keys Department
```

Hinzufügen von Tags beim Erstellen einer Zuordnung von Ereignisquellen

Verwenden Sie die CreateEventSourceMappingAPI-Operation, um eine neue Lambda-Ereignisquellenzuordnung mit Tags zu erstellen. Geben Sie den Parameter `Tags` an. Sie können diesen Vorgang mit dem `create-event-source-mapping` AWS CLI Befehl und der `--tags` Option aufrufen. Weitere Informationen zum CLI-Befehl finden Sie create-event-source-mappingin der AWS CLI Befehlsreferenz.

Vergewissern Sie sich vor der Verwendung des `Tags`-Parameters mit `CreateEventSourceMapping`, dass Ihre Rolle neben den üblichen Berechtigungen, die für diesen Vorgang erforderlich sind, auch die Berechtigung hat, Ressourcen zu taggen. Weitere Informationen zu den Berechtigungen für das Tagging finden Sie unter the section called “Erforderliche Berechtigungen zum Arbeiten mit Tags”.

Tags mit dem Lambda-Tag anzeigen APIs

Um die Tags anzuzeigen, die auf eine bestimmte Lambda-Ressource angewendet werden, verwenden Sie die `ListTags`-API-Operation. Weitere Informationen finden Sie unter ListTags.

Sie können diesen Vorgang mit dem `list-tags` AWS CLI Befehl aufrufen, indem Sie einen ARN (Amazon Resource Name) angeben.

```
aws lambda list-tags --resource arn:aws:lambda:us-east-1:123456789012:resource-type:resource-identifier
```

Filtern von Ressourcen nach Tag

Sie können den AWS Resource Groups Tagging API GetResourcesAPI-Vorgang verwenden, um Ihre Ressourcen nach Tags zu filtern. Die `GetResources`-Operation empfängt bis zu 10 Filter, wobei jeder Filter einen Tag-Schlüssel und bis zu 10 Tag-Werte enthält. Sie stellen `GetResources` mit einem `ResourceType` zur Verfügung, um nach bestimmten Ressourcentypen zu filtern.

Sie können diesen Vorgang mit dem `get-resources` AWS CLI Befehl aufrufen. Beispiele für die Verwendung von `get-resources` finden Sie unter get-resources in der AWS -CLI-Befehlsreferenz.

Steuern Sie, welche Ereignisse Lambda an Ihre Funktion sendet

Sie können die Ereignisfilterung verwenden, um zu steuern, welche Datensätze aus einem Stream oder einer Warteschlange Lambda an Ihre Funktion sendet. Sie können zum Beispiel einen Filter hinzufügen, damit Ihre Funktion nur Amazon-SQS-Nachrichten verarbeitet, die bestimmte Datenparameter enthalten. Die Ereignisfilterung funktioniert nur mit der bestimmten Zuordnung von Ereignisquellen. Sie können den Zuordnungen von Ereignisquellen Filter für Folgendes hinzufügen: AWS-Services

- Amazon DynamoDB
- Amazon Kinesis Data Streams
- Amazon MQ
- Amazon Managed Streaming for Apache Kafka (Amazon MSK)
- Selbstverwaltetes Apache Kafka
- Amazon-Simple-Queue-Service (Amazon SQS)

Spezifische Informationen zum Filtern mit bestimmten Ereignisquellen finden Sie unter the section called “Verwenden von Filtern mit unterschiedlichen AWS-Services”. Lambda unterstützt keine Ereignisfilterung für Amazon DocumentDB.

Standardmäßig können Sie bis zu fünf verschiedene Filter für eine einzelne Zuordnung von Ereignisquellen definieren. Ihre Filter gehören logisch ORed zusammen. Wenn ein Datensatz aus Ihrer Ereignisquelle einen oder mehrere Ihrer Filter erfüllt, nimmt Lambda den Datensatz in das nächste Ereignis auf, das es an Ihre Funktion sendet. Wenn keiner Ihrer Filter erfüllt ist, verwirft Lambda den Datensatz.

> **Note**
>
> Wenn Sie mehr als fünf Filter für eine Ereignisquelle definieren müssen, können Sie eine Kontingenterhöhung auf bis zu 10 Filter für jede Ereignisquelle beantragen. Wenn Sie versuchen, mehr Filter hinzuzufügen, als Ihr aktuelles Kontingent erlaubt, wird Lambda einen Fehler zurückgeben, wenn Sie versuchen, die Ereignisquelle zu erstellen.

Themen

- Grundlagen der Ereignisfilterung verstehen

- Umgang mit Datensätzen, die die Filterkriterien nicht erfüllen
- Filterregelsyntax
- Anhängen von Filterkriterien an eine Ereignisquellenzuordnung (Konsole)
- Anhängen von Filterkriterien an eine Ereignisquellenzuordnung (AWS CLI)
- Anhängen von Filterkriterien an eine Ereignisquellenzuordnung (AWS SAM)
- Verschlüsselung der Filterkriterien
- Verwenden von Filtern mit unterschiedlichen AWS-Services

Grundlagen der Ereignisfilterung verstehen

Ein Filterkriterienobjekt (`FilterCriteria`) ist eine Struktur, die aus einer Liste von Filtern (`Filters`) besteht. Jeder Filter ist eine Struktur, die ein Ereignisfiltermuster (`Pattern`) definiert. Ein Muster ist eine Zeichenkette, die eine JSON-Filterregel darstellt. Die Struktur eines `FilterCriteria`-Objekts ist wie folgt.

```
{
   "Filters": [
        {
            "Pattern": "{ \"Metadata1\": [ rule1 ], \"data\": { \"Data1\":
 [ rule2 ] }}"
        }
    ]
}
```

Zur Verdeutlichung sehen Sie hier den Wert des Filter-`Pattern` in reinem JSON.

```
{
    "Metadata1": [ rule1 ],
    "data": {
        "Data1": [ rule2 ]
    }
}
```

Ihr Filtermuster kann Metadateneigenschaften, Dateneigenschaften oder beides enthalten. Die verfügbaren Metadaten-Parameter und das Format der Daten-Parameter variieren je nach dem AWS-Service , der als Ereignisquelle fungiert. Nehmen wir zum Beispiel an, dass Ihre Zuordnung von Ereignisquellen den folgenden Datensatz von einer Amazon-SQS-Warteschlange empfängt:

```
{
    "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"
}
```

- Metadateneigenschaften sind die Felder, die Informationen über das Ereignis enthalten, das den Datensatz erstellt hat. Im Beispiel für den Amazon-SQS-Datensatz umfassen die Metadateneigenschaften Felder wie `messageID`, `eventSourceArn` und `awsRegion`.
- Dateneigenschaften sind die Felder des Datensatzes, die die Daten aus Ihrem Stream oder Ihrer Warteschlange enthalten. Im Amazon-SQS-Ereignisbeispiel ist der Schlüssel für das Datenfeld `body`, und die Dateneigenschaften sind die Felder `City`, `State` und `Temperature`.

Verschiedene Arten von Ereignisquellen verwenden unterschiedliche Schlüsselwerte für ihre Datenfelder. Um nach Dateneigenschaften zu filtern, müssen sie den richtigen Schlüssel in Ihrem Filtermuster verwenden. Eine Liste der Datenfilterschlüssel und Beispiele für Filtermuster für die einzelnen unterstützten AWS-Service Schlüssel finden Sie Verwenden von Filtern mit unterschiedlichen AWS-Services unter.

Die Ereignisfilterung kann mehrstufige JSON-Filterung verarbeiten. Betrachten Sie zum Beispiel das folgende Fragment eines Datensatzes aus einem DynamoDB-Stream:

```
"dynamodb": {
    "Keys": {
        "ID": {
            "S": "ABCD"
        }
        "Number": {
```

```
            "N": "1234"
    },
    ...
}
```

Angenommen, Sie möchten nur die Datensätze verarbeiten, bei denen die `Number` des Sortierschlüssels 4567 ist. In diesem Fall würde Ihr `FilterCriteria`-Objekt wie folgt aussehen:

```
{
    "Filters": [
        {
            "Pattern": "{ \"dynamodb\": { \"Keys\": { \"Number\": { \"N\":
 [ "4567" ] } } } }"
        }
    ]
}
```

Zur Verdeutlichung sehen Sie hier den Wert des Filter-`Pattern` in reinem JSON.

```
{
    "dynamodb": {
        "Keys": {
            "Number": {
                "N": [ "4567" ]
                }
            }
        }
}
```

Umgang mit Datensätzen, die die Filterkriterien nicht erfüllen

Die Art und Weise, wie Lambda mit Datensätzen umgeht, die Ihren Filterkriterien nicht entsprechen, hängt von der Ereignisquelle ab.

- Bei Amazon SQS entfernt Lambda die Nachricht automatisch aus der Warteschlange, wenn die Nachricht Ihren Filterkriterien nicht entspricht. Sie müssen diese Nachrichten in Amazon SQS nicht manuell löschen.
- Bei Kinesis und DynamoDB schreitet der Stream-Iterator nach der Auswertung eines Datensatzes durch Ihre Filterkriterien über diesen Datensatz hinaus. Wenn der Datensatz Ihre Filterkriterien nicht erfüllt, müssen Sie den Datensatz nicht manuell aus Ihrer Ereignisquelle löschen. Nach Ablauf

der Aufbewahrungsfrist löschen Kinesis und DynamoDB diese alten Datensätze automatisch. Wenn Sie möchten, dass Datensätze früher gelöscht werden, lesen Sie Ändern des Zeitraums der Datenaufbewahrung.

- Bei Amazon MSK-, selbstverwalteten Apache-Kafka- und Amazon MQ-Nachrichten verwirft Lambda Nachrichten, die nicht allen im Filter enthaltenen Feldern entsprechen. Bei Amazon MSK und selbstverwaltetem Apache Kafka schreibt Lambda Offsets für übereinstimmende und nicht übereinstimmende Nachrichten fest, nachdem die Funktion erfolgreich aufgerufen wurde. Bei Amazon MQ bestätigt Lambda übereinstimmende Nachrichten nach erfolgreichem Aufruf der Funktion und bestätigt nicht übereinstimmende Nachrichten beim Filtern.

Filterregelsyntax

Für Filterregeln unterstützt Lambda die EventBridge Amazon-Regeln und verwendet dieselbe Syntax wie EventBridge. Weitere Informationen finden Sie unter Amazon EventBridge Event Patterns im EventBridge Amazon-Benutzerhandbuch.

Im Folgenden finden Sie eine Zusammenfassung aller Vergleichsoperatoren, die für die Lambda-Ereignisfilterung verfügbar sind.

Vergleichsoperator	Beispiel	Regelsyntax
Null	UserID is null	"UserID": [null]
Leer	LastName ist leer	"LastName": [""]
Gleichheitszeichen	Name is "Alice"	"Name": ["Alice"]
Gleich (Groß-/Kleinschreibung ignorieren)	Name is "Alice"	„Name“: [{"equals-ignore-case„: „Alice“}]
Bedingung 2	Location is "New York" and Day is "Monday"	"Location": ["New York"], "Day": ["Monday"]
Oder	PaymentType ist „Kredit“ oder „Lastschrift“	"PaymentType„: [„Kredit“, „Lastschrift"]

Vergleichsoperator	Beispiel	Regelsyntax
Oder (mehrere Felder)	Location is "New York", or Day is "Monday".	"$or": [{ "Location": ["New York"] }, { "Day": ["Monday"] }]
Nicht	Weather is anything but "Raining"	"Weather": [{ "anything-but": ["Raining"] }]
Numeric (equals)	Price is 100	"Price": [{ "numeric": ["=", 100] }]
Numeric (range)	Price is more than 10, and less than or equal to 20	"Price": [{ "numeric": [">", 10, "<=", 20] }]
Vorhanden	ProductName existiert	"ProductName„: [{„existiert“: wahr}]
Nicht vorhanden	ProductName existiert nicht	"ProductName„: [{„existiert“: falsch}]
Beginnt mit	Region is in the US	"Region": [{"prefix": "us-" }]
Endet mit	FileName endet mit der Erweiterung.png.	"FileName„: [{„Suffix“: „.png“}]

> **Note**
>
> Wie bei EventBridge Zeichenketten verwendet Lambda den exakten character-by-character Abgleich ohne Umschaltung der Groß- und Kleinschreibung oder eine andere Normalisierung von Zeichenketten. Bei Zahlen verwendet Lambda auch eine Zeichenfolgendarstellung. 300, 300,0 und 3,0e2 werden z. B. nicht gleich behandelt.

Beachten Sie, dass der Exists-Operator nur für Blattknoten in Ihrer JSON-Ereignisquelle funktioniert. Es passt nicht zu Zwischenknoten. Bei der folgenden JSON-Datei `{ "person": { "address": [ { "exists": true } ] } }` würde das Filtermuster beispielsweise keine Übereinstimmung finden, da es sich um einen Zwischenknoten `"address"` handelt.

```
{
  "person": {
    "name": "John Doe",
    "age": 30,
    "address": {
      "street": "123 Main St",
      "city": "Anytown",
      "country": "USA"
    }
  }
}
```

Anhängen von Filterkriterien an eine Ereignisquellenzuordnung (Konsole)

Führen Sie diese Schritte aus, um eine neue Ereignisquellenzuordnung mit Filterkriterien über die Lambda-Konsole zu erstellen.

Eine neue Ereignisquellenzuordnung mit Filterkriterien erstellen (Konsole)

1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.
2. Wählen Sie den Namen einer Funktion aus, für die eine Ereignisquellenzuordnung erstellt werden soll.
3. Wählen Sie unter Function overview (Funktionsübersicht) die Option Add trigger (Trigger hinzufügen).
4. Wählen Sie bei Trigger configuration (Auslöserkonfiguration) einen Auslösertyp aus, der Ereignisfilterung unterstützt. Eine Liste der unterstützten Services finden Sie in der Liste am Anfang dieser Seite.
5. Erweitern Sie Additional settings (Zusätzliche Einstellungen).
6. Wählen Sie unter Filter criteria (Filterkriterien) die Option Add (Hinzufügen) aus und definieren Sie anschließend Ihre Filter. Sie können z. B. Folgendes eingeben.

   ```
   { "Metadata" : [ 1, 2 ] }
   ```

   Damit wird Lambda angewiesen, nur Datensätze zu verarbeiten, in denen das Feld `Metadata` gleich 1 oder 2 ist. Sie können weiterhin Hinzufügen auswählen, um weitere Filter bis zur maximal zulässigen Anzahl hinzuzufügen.
7. Wenn Sie fertig mit dem Hinzufügen Ihrer Filter sind, klicken Sie auf Speichern.

Wenn Sie Filterkriterien über die Konsole eingeben, geben Sie nur das Filtermuster ein und müssen weder den `Pattern`-Schlüssel noch Escape-Anführungszeichen angeben. In Schritt 6 der Anweisungen oben entspricht `{ "Metadata" : [ 1, 2 ] }` den folgenden `FilterCriteria`.

```
{
   "Filters": [
      {
          "Pattern": "{ \"Metadata\" : [ 1, 2 ] }"
      }
   ]
}
```

Nachdem Sie Ihre Ereignisquellenzuordnung in der Konsole erstellt haben, sehen Sie die formatierten `FilterCriteria` in den Auslöserdetails. Weitere Beispiele zum Erstellen von Ereignisfiltern mithilfe der -Konsole finden Sie unter Verwenden von Filtern mit unterschiedlichen AWS-Services.

Anhängen von Filterkriterien an eine Ereignisquellenzuordnung (AWS CLI)

Angenommen, Sie möchten, dass eine Ereignisquellenzuordnung folgende hat `FilterCriteria`:

```
{
   "Filters": [
      {
          "Pattern": "{ \"Metadata\" : [ 1, 2 ] }"
      }
   ]
}
```

Führen Sie den folgenden Befehl aus, um mithilfe der AWS Command Line Interface (AWS CLI) eine neue Zuordnung der Ereignisquelle mit diesen Filterkriterien zu erstellen.

```
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 ]}"}]}'
```

Dieser create-event-source-mappingBefehl erstellt eine neue Amazon SQS SQS-Ereignisquellenzuordnung für eine Funktion `my-function` mit dem angegebenen `FilterCriteria` Wert.

Führen Sie den folgenden Befehl aus, um diese Filterkriterien zu einer vorhandenen Zuordnung von Ereignisquellen hinzuzufügen.

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"Metadata\" : [ 1, 2 ]}"}]}'
```

Beachten Sie, dass Sie zum Aktualisieren einer Ereignisquellenzuordnung ihre UUID benötigen. Sie können die UUID aus einem Anruf abrufen. list-event-source-mappings Lambda gibt auch die UUID in der CLI-Antwort zurück. create-event-source-mapping

Um Filterkriterien aus einer Ereignisquelle zu entfernen, können Sie den folgenden update-event-source-mappingBefehl mit einem leeren Objekt ausführen. `FilterCriteria`

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
    --filter-criteria "{}"
```

Weitere Beispiele für das Erstellen von Ereignisfiltern mithilfe von finden Sie unterVerwenden von Filtern mit unterschiedlichen AWS-Services. AWS CLI

Anhängen von Filterkriterien an eine Ereignisquellenzuordnung (AWS SAM)

Angenommen, Sie möchten eine Ereignisquelle so konfigurieren AWS SAM , dass sie die folgenden Filterkriterien verwendet:

```
{
   "Filters": [
      {
          "Pattern": "{ \"Metadata\" : [ 1, 2 ] }"
      }
   ]
}
```

Um diese Filterkriterien zu Ihrer Zuordnung von Ereignisquellen hinzuzufügen, fügen Sie das folgende Snippet in die YAML-Vorlage für Ihre Ereignisquelle ein.

```
FilterCriteria:
  Filters:
    - Pattern: '{"Metadata": [1, 2]}'
```

Weitere Informationen zum Erstellen und Konfigurieren einer AWS SAM Vorlage für eine Ereignisquellenzuordnung finden Sie im EventSourceAbschnitt des AWS SAM Entwicklerhandbuchs. Weitere Beispiele für die Erstellung von Ereignisfiltern mithilfe von AWS SAM Vorlagen finden Sie unterVerwenden von Filtern mit unterschiedlichen AWS-Services.

Verschlüsselung der Filterkriterien

Standardmäßig verschlüsselt Lambda Ihr Filterkriterienobjekt nicht. In Anwendungsfällen, in denen Sie vertrauliche Informationen in Ihr Filterkriterienobjekt aufnehmen können, können Sie diese mit Ihrem eigenen KMS-Schlüssel verschlüsseln.

Nachdem Sie Ihr Filterkriterienobjekt verschlüsselt haben, können Sie seine Klartextversion mithilfe eines GetEventSourceMappingAPI-Aufrufs anzeigen. Sie müssen über `kms:Decrypt`-Berechtigungen verfügen, um die Filterkriterien im Klartext anzeigen zu können.

> **Note**
>
> Wenn Ihr Filterkriterienobjekt verschlüsselt ist, redigiert Lambda den Wert des `FilterCriteria` Felds in der ListEventSourceMappingsAntwort auf Aufrufe. Stattdessen wird dieses Feld als `null` angezeigt. Verwenden Sie die API`FilterCriteria`, um den wahren Wert von zu ermitteln. GetEventSourceMapping
> Um den entschlüsselten Wert von `FilterCriteria` in der Konsole anzuzeigen, stellen Sie sicher, dass Ihre IAM-Rolle Berechtigungen für enthält. GetEventSourceMapping

Sie können Ihren eigenen KMS-Schlüssel über die Konsole, API/CLI oder CloudFormation angeben.

Um Filterkriterien mit einem kundeneigenen KMS-Schlüssel zu verschlüsseln (Konsole)

1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.
2. Wählen Sie Add trigger. Wenn Sie bereits über einen vorhandenen Trigger verfügen, wählen Sie die Registerkarte Konfiguration und dann Trigger aus. Wählen Sie den vorhandenen Auslöser aus und klicken Sie auf Edit (Bearbeiten).
3. Aktivieren Sie das Kontrollkästchen neben Mit dem vom Kunden verwalteten KMS-Schlüssel verschlüsseln.
4. Wählen Sie für Wählen Sie einen vom Kunden verwalteten KMS-Verschlüsselungsschlüssel einen vorhandenen aktivierten Schlüssel aus oder erstellen Sie einen neuen Schlüssel. Je nach Vorgang benötigen Sie einige oder alle der folgenden Berechtigungen: `kms:DescribeKey`,

`kms:GenerateDataKey` und `kms:Decrypt`. Verwenden Sie die KMS-Schlüsselrichtlinie, um diese Berechtigungen zu gewähren.

Wenn Sie Ihren eigenen KMS-Schlüssel verwenden, müssen die folgenden API-Vorgänge in der Schlüsselrichtlinie zugelassen sein:

- `kms:Decrypt` – Muss dem regionalen Lambda-Serviceprinzipal gewährt werden (`lambda.AWS_region.amazonaws.com`). Dies ermöglicht Lambda die Entschlüsselung von Daten mit diesem KMS-Schlüssel.
  - Um ein dienstübergreifendes Problem mit verwirrten Stellvertretern zu vermeiden, verwendet die Schlüsselrichtlinie den `aws:SourceArn` globalen Bedingungsschlüssel. Der richtige Wert des `aws:SourceArn`-Schlüssels ist der ARN Ihrer Ressource für die Zuordnung von Ereignisquellen. Sie können ihn also erst dann zu Ihrer Richtlinie hinzufügen, wenn Sie den ARN kennen. Lambda leitet auch die `aws:lambda:FunctionArn`- und `aws:lambda:EventSourceArn`-Schlüssel und ihre jeweiligen Werte im Verschlüsselungskontext weiter, wenn eine Entschlüsselungsanforderung an KMS gestellt wird. Diese Werte müssen den in der Schlüsselrichtlinie angegebenen Bedingungen entsprechen, damit die Entschlüsselungsanforderung erfolgreich ist. EventSourceArn Für selbstverwaltete Kafka-Ereignisse müssen Sie keine Quellen angeben, da sie keine haben. EventSourceArn
- `kms:Decrypt`— Muss auch dem Prinzipal gewährt werden, der beabsichtigt, den Schlüssel zur Anzeige der Klartext-Filterkriterien in GetEventSourceMappingunseren API-Aufrufen zu verwenden. DeleteEventSourceMapping
- `kms:DescribeKey` – Stellt die vom Kunden verwalteten Schlüsseldetails bereit, um dem angegebenen Prinzipal die Verwendung des Schlüssels zu ermöglichen.
- `kms:GenerateDataKey` – Ermöglicht Lambda die Generierung eines Datenschlüssels zur Verschlüsselung der Filterkriterien im Namen des angegebenen Prinzipals (Umschlagverschlüsselung).

Sie können AWS CloudTrail damit AWS KMS Anfragen verfolgen, die Lambda in Ihrem Namen stellt. CloudTrail Beispielereignisse finden Sie unter???.

Wir empfehlen außerdem, den `kms:ViaService`-Bedingungsschlüssel zu verwenden, um die Verwendung des KMS-Schlüssels nur auf Anfragen von Lambda zu beschränken. Der Wert dieses Schlüssels ist der regionale Lambda-Serviceprinzipal (`lambda.AWS_region.amazonaws.com`). Im Folgenden finden Sie ein Beispiel für eine Schlüsselrichtlinie, die alle relevanten Berechtigungen gewährt:

Example AWS KMS wichtige Richtlinie

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"
                }
            }
        }
    ]
}
```

Um Ihren eigenen KMS-Schlüssel zum Verschlüsseln von Filterkriterien zu verwenden, können Sie auch den folgenden CreateEventSourceMapping AWS CLI Befehl verwenden. Geben Sie den KMS-Schlüssel ARN mit dem `--kms-key-arn`-Flag an.

```
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
```

Wenn Sie bereits über eine Ereignisquellenzuordnung verfügen, verwenden Sie stattdessen den UpdateEventSourceMapping AWS CLI Befehl. Geben Sie den KMS-Schlüssel ARN mit dem `--kms-key-arn`-Flag an.

```
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
```

Dieser Vorgang überschreibt alle zuvor angegebenen KMS-Schlüssel. Wenn Sie das `--kms-key-arn`-Flag zusammen mit einem leeren Argument angeben, verwendet Lambda Ihren KMS-Schlüssel nicht mehr zum Verschlüsseln von Filterkriterien. Stattdessen verwendet Lambda standardmäßig wieder einen Amazon-eigenen Schlüssel.

Um Ihren eigenen KMS-Schlüssel in einer CloudFormation Vorlage anzugeben, verwenden Sie die `KMSKeyArn` Eigenschaft des `AWS::Lambda::EventSourceMapping` Ressourcentyps. Sie können beispielsweise das folgende Snippet in die YAML-Vorlage für Ihre Ereignisquelle einfügen.

```
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"
    ...
```

Um Ihre verschlüsselten Filterkriterien in einem GetEventSourceMappingoder DeleteEventSourceMappingAPI-Aufruf im Klartext anzeigen zu können, benötigen Sie entsprechende `kms:Decrypt` Berechtigungen.

Ab dem 6. August 2024 wird das `FilterCriteria` Feld nicht mehr in AWS CloudTrail Protokollen von CreateEventSourceMapping, und DeleteEventSourceMappingAPI-Aufrufen angezeigt UpdateEventSourceMapping, wenn Ihre Funktion keine Ereignisfilterung verwendet. Wenn Ihre Funktion die Ereignisfilterung verwendet, wird das `FilterCriteria`-Feld als leer (`{}`) angezeigt. Sie können Ihre Filterkriterien in der Antwort auf GetEventSourceMappingAPI-Aufrufe weiterhin im Klartext anzeigen, wenn Sie über `kms:Decrypt` Berechtigungen für den richtigen KMS-Schlüssel verfügen.

Beispiel für einen CloudTrail Protokolleintrag für Anrufe Create/Update/DeleteEventSourceMapping

Im folgenden AWS CloudTrail Beispiel wird ein Protokolleintrag für einen CreateEventSourceMapping Anruf als leer (`{}`) `FilterCriteria` angezeigt, da die Funktion eine Ereignisfilterung verwendet. Dies ist auch dann der Fall, wenn `FilterCriteria`-Objekt gültige Filterkriterien enthält, die Ihre

Funktion aktiv verwendet. Wenn die Funktion keine Ereignisfilterung verwendet, CloudTrail wird das `FilterCriteria` Feld in Protokolleinträgen überhaupt nicht angezeigt.


```
{
    "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"
}
```


Verwenden von Filtern mit unterschiedlichen AWS-Services

Verschiedene Arten von Ereignisquellen verwenden unterschiedliche Schlüsselwerte für ihre Datenfelder. Um nach Dateneigenschaften zu filtern, müssen sie den richtigen Schlüssel in Ihrem Filtermuster verwenden. In der folgenden Tabelle sind die Filterschlüssel für jeden unterstützten Schlüssel aufgeführt AWS-Service.

AWS-Service	Filterschlüssel
DynamoDB	`dynamodb`
Kinesis	`data`

AWS-Service	Filterschlüssel
Amazon MQ	`data`
Amazon MSK	`value`
Selbstverwaltetes Apache Kafka	`value`
Amazon SQS	`body`

In den folgenden Abschnitten finden Sie Beispiele für Filtermuster für verschiedene Arten von Ereignisquellen. Sie enthalten auch Definitionen der unterstützten Formate für eingehende Daten und Filtermuster für jeden unterstützten Service.

- the section called “Ereignisfilterung”
- the section called “Ereignisfilterung”
- the section called “Ereignisfilterung”
- the section called “Ereignisfilterung”
- the section called “Ereignisfilterung”

Testen von Lambda-Funktionen mit Hilfe der Konsole

Sie können Ihre Lambda-Funktion in der Konsole testen, indem Sie Ihre Funktion mit einem Testereignis aufrufen. Ein Testereignis ist eine JSON-Eingabe für Ihre Funktion. Wenn Ihre Funktion keine Eingabe erfordert, kann das Ereignis ein leeres Dokument (`{}`) sein.

Wenn Sie einen Test in der Konsole ausführen, ruft Lambda Ihre Funktion synchron mit dem Testereignis auf. Die Funktionslaufzeit konvertiert das Ereignis-JSON in ein Objekt und übergibt es zur Verarbeitung an die Handler-Methode Ihres Codes.

> **Erstellen Sie ein Testereignis**
>
> Bevor Sie in der Konsole testen können, müssen Sie ein privates oder gemeinsam nutzbares Testereignis erstellen.

Aufrufen von Funktionen mit Testereignissen

So testen Sie eine Funktion

1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.
2. Klicken Sie auf den Namen der Funktion, die Sie testen möchten.
3. Wählen Sie die Registerkarte Test.
4. Wählen Sie unter Test event (Testereignis) die Option Create new event (Neues Ereignis erstellen) oder Edit saved event (Gespeichertes Ereignis bearbeiten) und dann das gespeicherte Ereignis aus, das Sie verwenden möchten.
5. Optional — wählen Sie ein Template (Vorlage) für den Event-JSON.
6. Wählen Sie Test aus.
7. Erweitern Sie unter Execution result (Ausführungsergebnis) die Option Details, um die Testergebnisse anzuzeigen.

Um Ihre Funktion aufzurufen, ohne Ihr Testereignis zu speichern, wählen Sie Test (Testen) vor dem Speichern aus. Dadurch wird ein nicht gespeichertes Testereignis erstellt, das Lambda nur für die Dauer der Sitzung beibehält.

Für die Laufzeiten Node.js, Python und Ruby können Sie auf der Registerkarte Code auch auf Ihre gespeicherten und nicht gespeicherten Testereignisse zugreifen. Verwenden Sie den Abschnitt TESTEREIGNISSE, um Tests zu erstellen, zu bearbeiten und auszuführen.

Private Testereignisse erstellen

Private Testereignisse stehen nur dem Ereignisersteller zur Verfügung und benötigen keine zusätzlichen Berechtigungen zur Verwendung. Sie können bis zu 10 Testereignisse pro Funktion erstellen und speichern.

So erstellen Sie ein privates Testereignis

1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.
2. Klicken Sie auf den Namen der Funktion, die Sie testen möchten.
3. Wählen Sie die Registerkarte Test.
4. Erledigen Sie unter Testereignis Folgendes:
    a. Wählen Sie eine Vorlage.
    b. Geben Sie einen Namen für den Test an.
    c. Geben Sie im Texteingabefeld das JSON-Testereignis ein.
    d. Unter Ereignisfreigabeeinstellungen wählen Sie Privat aus.
5. Wählen Sie Änderungen speichern aus.

Für die Laufzeiten von Node.js, Python und Ruby können Sie auf der Registerkarte Code auch Testereignisse erstellen. Verwenden Sie den Abschnitt TESTEREIGNISSE, um Tests zu erstellen, zu bearbeiten und auszuführen.

Freigabefähige Testereignisse erstellen

Gemeinsam nutzbare Testereignisse sind Testereignisse, die Sie mit anderen Benutzern desselben AWS Kontos teilen können. Sie können die freigabefähigen Testereignisse anderer Benutzer bearbeiten und Ihre Funktion damit aufrufen.

Lambda speichert gemeinsam nutzbare Testereignisse als Schemas in einer Amazon EventBridge (CloudWatch Events) -Schemaregistry mit dem Namen. `lambda-testevent-schemas` Da Lambda diese Registry verwendet, um gemeinsam genutzte Testereignisse zu speichern und aufzurufen,

empfehlen wir Ihnen nicht, diese Registry zu bearbeiten oder eine Registry mit dem Namen `lambda-testevent-schemas` zu erstellen.

Um gemeinsam nutzbare Testereignisse zu sehen, zu teilen und zu bearbeiten, benötigen Sie Berechtigungen für alle der folgenden API-Operationen für die EventBridge Schemaregistrierung (CloudWatch Ereignisse):

- `schemas.CreateRegistry`
- `schemas.CreateSchema`
- `schemas.DeleteSchema`
- `schemas.DeleteSchemaVersion`
- `schemas.DescribeRegistry`
- `schemas.DescribeSchema`
- `schemas.GetDiscoveredSchema`
- `schemas.ListSchemaVersions`
- `schemas.UpdateSchema`

Beachten Sie, dass das Speichern von Änderungen an einem freigabefähigen Testereignis dieses Ereignis überschreibt.

Wenn Sie keine freigabefähigen Testereignisse erstellen, bearbeiten oder anzeigen können, überprüfen Sie, ob Ihr Konto über die erforderlichen Berechtigungen für diese Operationen verfügt. Wenn Sie über die erforderlichen Berechtigungen verfügen, aber immer noch nicht auf gemeinsam nutzbare Testereignisse zugreifen können, suchen Sie nach ressourcenbasierten Richtlinien, die den Zugriff auf die Registrierung EventBridge (CloudWatch Ereignisse) einschränken könnten.

So erstellen Sie ein freigabefähiges Testereignis

1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.
2. Klicken Sie auf den Namen der Funktion, die Sie testen möchten.
3. Wählen Sie die Registerkarte Test.
4. Erledigen Sie unter Testereignis Folgendes:
    a. Wählen Sie eine Vorlage.
    b. Geben Sie einen Namen für den Test an.
    c. Geben Sie im Texteingabefeld das JSON-Testereignis ein.

d. Wählen Sie unter Event sharing settings (Ereignisfreigabeeinstellungen) Shareable (Freigabefähig) aus.

5. Wählen Sie Änderungen speichern aus.

> Verwenden Sie gemeinsam nutzbare Testereignisse mit. AWS Serverless Application Model
>
> Sie können es verwenden AWS SAM , um gemeinsam nutzbare Testereignisse aufzurufen. Siehe `sam remote test-event` im Entwicklerhandbuch für AWS Serverless Application Model

Löschen von freigabefähigen Test-Ereignisschemas

Wenn Sie freigabefähige Testereignisse löschen, entfernt Lambda sie aus der `lambda-testevent-schemas`-Registry. Wenn Sie das letzte freigabefähige Testereignis aus der Registry entfernen, löscht Lambda die Registry.

Wenn Sie die Funktion löschen, löscht Lambda keine verknüpften freigabefähigen Testereignisschemas. Sie müssen diese Ressourcen manuell über die Konsole EventBridge (CloudWatch Ereignisse) bereinigen.

Lambda-Funktionszustände

Lambda enthält in der Funktionskonfiguration für alle Funktionen ein Statusfeld, das anzeigt, wann Ihre Funktion zum Aufruf bereit ist. `State` gibt Auskunft über den aktuellen Status der Funktion, einschließlich der Frage, ob Sie die Funktion erfolgreich aufrufen können. Funktionszustände ändern nicht das Verhalten von Funktionsaufrufen oder wie Ihre Funktion den Code ausführt.

> **Note**
>
> Die Funktionszustandsdefinitionen unterscheiden sich geringfügig für SnapStartFunktionen. Weitere Informationen finden Sie unter Lambda SnapStart und Funktionszustände.

In vielen Fällen ist eine DynamoDB-Tabelle eine ideale Methode zum Bewahren des Status zwischen Aufrufen, da sie Datenzugriff mit geringer Latenz ermöglicht und mit dem Lambda-Service skaliert werden kann. Sie können Daten auch in Amazon EFS for Lambda speichern, wenn Sie diesen Service verwenden. Dadurch wird ein Zugriff mit geringer Latenz auf den Dateisystemspeicher ermöglicht.

Zu den Funktionszustände gehören:

- `Pending` – Nachdem Lambda die Funktion erstellt hat, setzt es den Status auf „Ausstehend“. Im Status „Ausstehend“ versucht Lambda, Ressourcen für die Funktion zu erstellen oder zu konfigurieren, beispielsweise VPC- oder EFS-Ressourcen. Lambda ruft während des ausstehenden Status keine Funktion auf. Alle Aufrufe oder andere API-Aktionen, die mit der Funktion arbeiten, schlagen fehl.
- `Active` – Ihre Funktion wechselt in den aktiven Status, nachdem Lambda die Ressourcenkonfiguration und -Bereitstellung abgeschlossen hat. Funktionen können nur erfolgreich aufgerufen werden, wenn sie aktiv sind.
- `Failed` – Gibt an, dass bei der Ressourcenkonfiguration oder bei der Bereitstellung ein Fehler aufgetreten ist. Wenn die Funktionserstellung fehlschlägt, setzt Lambda den Funktionsstatus auf Fehlgeschlagen, und Sie müssen die Funktion löschen und neu erstellen.
- `Inactive` – Eine Funktion wird inaktiv, wenn sie lange genug im Leerlauf war, damit Lambda die externen Ressourcen zurückfordern kann, die für sie konfiguriert wurden. Wenn Sie versuchen, eine Funktion aufzurufen, die inaktiv ist, schlägt der Aufruf fehl und Lambda setzt die Funktion auf den Status „Ausstehend“, bis die Funktionsressourcen neu erstellt werden. Wenn Lambda die Ressourcen nicht neu erstellen kann, kehrt die Funktion in den inaktiven Zustand zurück.

Möglicherweise müssen Sie alle Fehler beheben und Ihre Funktion erneut bereitstellen, um sie wieder in den aktiven Zustand zu versetzen.

Wenn Sie SDK-basierte Automatisierungsworkflows verwenden oder den Lambda-Service APIs direkt aufrufen, stellen Sie sicher, dass Sie vor dem Aufruf den Status einer Funktion überprüfen, um sicherzustellen, dass sie aktiv ist. Sie können dies mit der Lambda-API-Aktion GetFunctionoder durch die Konfiguration eines Kellners mithilfe des AWS SDK for Java 2.0 tun.

```
aws lambda get-function --function-name my-function --query 'Configuration.[State,
 LastUpdateStatus]'
```

Die Ausgabe sollte folgendermaßen aussehen:

```
[
 "Active",
 "Successful"
]
```

Die folgenden Operationen schlagen fehl, während die Funktionserstellung aussteht:

- Invoke
- UpdateFunctionCode
- UpdateFunctionConfiguration
- PublishVersion

Funktionsstatus während der Aktualisierung

Lambda hat zwei Vorgänge zum Aktualisieren von Funktionen:

- UpdateFunctionCode: Aktualisiert das Bereitstellungspaket der Funktion
- UpdateFunctionConfiguration: Aktualisiert die Konfiguration der Funktion

Lambda verwendet das LastUpdateStatusAttribut, um den Fortschritt dieser Aktualisierungsvorgänge zu verfolgen. Während ein Update läuft (wenn `"LastUpdateStatus": "InProgress"`):

- Der Status der Funktion bleibt `Active`.

- Bei Aufrufen werden weiterhin der vorherige Code und die vorherige Konfiguration der Funktion verwendet, bis die Aktualisierung abgeschlossen ist.
- Der folgende Vorgang schlägt fehl:
  - UpdateFunctionCode
  - UpdateFunctionConfiguration
  - PublishVersion
  - TagResource

Wenn ein Update fehlschlägt (wann`"LastUpdateStatus": "Failed"`):

- Der Status der Funktion bleibt `Active`.
- Aufrufe verwenden weiterhin den vorherigen Code und die vorherige Konfiguration der Funktion.

Example GetFunctionConfiguration Antwort

Das folgende Beispiel ist das Ergebnis einer GetFunctionConfigurationAnfrage für eine Funktion, die gerade aktualisiert wird.

```
{
    "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",
    ...
}
```

Grundlegendes zum Wiederholungsverhalten in Lambda

Wenn Sie eine Funktion direkt aufrufen, bestimmen Sie die Strategie für den Umgang mit Fehlern im Zusammenhang mit dem Funktionscode. Lambda wiederholt diese Art von Fehlern nicht automatisch in Ihrem Namen. Um den Vorgang zu wiederholen, können Sie die Funktion manuell erneut aufrufen, das fehlgeschlagene Ereignis zum Debuggen an eine Warteschlange senden oder den Fehler ignorieren. Der Code Ihrer Funktion wurde möglicherweise vollständig, teilweise oder überhaupt nicht ausgeführt. Wenn Sie es erneut versuchen, stellen Sie sicher, dass der Code Ihrer Funktion das gleiche Ereignis mehrmals verarbeiten kann, ohne doppelte Transaktionen oder andere unerwünschte Nebenwirkungen zu verursachen.

Wenn Sie eine Funktion indirekt aufrufen, müssen Sie sich über das Wiederholungsverhalten des Aufrufers und alle Services im Klaren sein, mit denen die Anforderung während des Vorgangs konfrontiert wird. Dies umfasst die folgenden Szenarien.

- Asynchroner Aufruf – Lambda startet bei Funktionsfehlern zwei Wiederholungsversuche. Wenn die Kapazität der Funktion nicht für die Verarbeitung aller eingehenden Anforderungen ausreicht, verbleiben die an die Funktion zu sendenden Ereignisse möglicherweise stundenlang in der Warteschlange. Sie können für die Funktion eine Warteschlange für unzustellbare Nachrichten konfigurieren, um nicht erfolgreich verarbeitete Ereignisse aufzufangen. Weitere Informationen finden Sie unter the section called “Warteschlangen für unzustellbare Briefe”.
- Ereignisquellen-Mappings – Ereignisquellen-Mappings, die aus Streams lesen, führen für den gesamten Batch von Elementen einen Wiederholungsversuch durch. Wiederholte Fehlermeldungen blockieren die Verarbeitung des betroffenen Shards, bis der Fehler behoben ist oder die Elemente ablaufen. Zum Erkennen blockierter Shards können Sie die Metrik Iterator Age überwachen.

  Für Ereignisquellen-Zuweisungen, die aus einer Warteschlange lesen, bestimmen Sie die Zeitdauer zwischen Wiederholversuchen und dem Ziel für fehlgeschlagene Ereignisse, indem Sie für die Quellwarteschlange die Zeitbeschränkung für die Sichtbarkeit und die Redrive-Richtlinie konfigurieren. Weitere Informationen finden Sie unter Wie Lambda Datensätze aus Stream- und warteschlangenbasierten Ereignisquellen verarbeitet und in der servicespezifischen Dokumentation unter Lambda mit Ereignissen aus anderen Diensten aufrufen AWS.
- AWS-Services – AWS-Services können Ihre Funktion synchron oder asynchron aufrufen. Beim synchronen Aufruf entscheidet der Dienst, ob er es erneut versucht. Zum Beispiel wiederholen Amazon-S3-Batch-Operationen den Vorgang, wenn die Lambda-Funktion einen `TemporaryFailure`-Antwortcode zurückgibt. Dienste, die ein Proxy von einem Upstream-

Benutzer oder -Kunden anfordert, haben möglicherweise eine Wiederholungsstrategie oder leiten die Fehlerantwort möglicherweise an den Anforderer weiter. Zum Beispiel gibt API Gateway die Fehlerantwort immer an den Anforderer zurück.

Bei asynchronen Aufrufen ist die Wiederholungslogik identisch mit der Wiederholungslogik, unabhängig von der Aufrufquelle. Standardmäßig wiederholt Lambda einen fehlgeschlagenen asynchronen Aufruf bis zu zweimal. Weitere Informationen finden Sie unter Wie Lambda mit Fehlern und Wiederholungsversuchen mit asynchronem Aufruf umgeht.

- Andere Konten und Clients – Wenn Sie Zugriff auf andere Konten gewähren, können Sie mithilfe von ressourcenbasierten Richtlinien einschränken, welche Services und Ressourcen von ihnen zum Aufruf Ihrer Funktion konfiguriert werden können. Zum Schutz Ihrer Funktion vor einer Überlastung sollten Sie Ihrer Funktion mit Amazon API Gateway eine API-Ebene vorlagern.

Um Sie bei der Handhabung von Fehlern in Lambda-Anwendungen zu unterstützen, ist Lambda in Services wie Amazon CloudWatch und integriert AWS X-Ray. Sie können mithilfe einer Kombination aus Protokollen, Metriken, Alarmen und Ablaufverfolgung Probleme in Funktionscode, API und anderen Ressourcen, die für Ihre Anwendung erforderlich sind, schnell erkennen und aufdecken. Weitere Informationen finden Sie unter Überwachung, Debbugging und Fehlerbehebung bei Lambda-Funktionen.

Verwenden Sie die rekursive Lambda-Schleifenerkennung, um Endlosschleifen zu verhindern

Wenn Sie als Ziel für die Ausgabe einer Lambda-Funktion den gleichen Service oder die gleiche Ressource konfigurieren, durch den bzw. durch die die Funktion aufgerufen wird, kann eine unendliche rekursive Schleife entstehen. Ein Beispiel wäre etwa eine Lambda-Funktion, die eine Nachricht in eine Amazon Simple Queue Service (Amazon SQS)-Warteschlange schreibt, die wiederum die gleiche Funktion aufruft. Dieser Aufruf veranlasst die Funktion, eine weitere Nachricht in die Warteschlange zu schreiben, die wiederum erneut die Funktion aufruft.

Unbeabsichtigte rekursive Schleifen können dazu führen, dass Ihnen unerwartete Gebühren in Rechnung gestellt werden. AWS-Konto Außerdem können Schleifen dazu führen, dass Lambda skaliert wird und die gesamte verfügbare Parallelität Ihres Kontos nutzt. Um die Auswirkungen unbeabsichtigter Schleifen zu verringern, erkennt Lambda bestimmte Arten von rekursiven Schleifen kurz nach ihrem Auftreten. Wenn Lambda eine rekursive Schleife erkennt, wird der Aufruf Ihrer Funktion standardmäßig gestoppt und Sie werden benachrichtigt. Wenn Ihr Entwurf bewusst rekursive Muster verwendet, können Sie die Standardkonfiguration einer Funktion so ändern, dass sie rekursiv aufgerufen werden kann. Weitere Informationen finden Sie unter the section called “Zulassen, dass eine Lambda-Funktion in einer rekursiven Schleife ausgeführt wird”.

Sections

- Grundlegendes zur Erkennung rekursiver Schleifen
- Unterstützt AWS-Services und SDKs
- Benachrichtigungen zu rekursiven Schleifen
- Reagieren auf Benachrichtigungen im Zusammenhang mit der Erkennung rekursiver Schleifen
- Zulassen, dass eine Lambda-Funktion in einer rekursiven Schleife ausgeführt wird
- Unterstützte Regionen für die Erkennung rekursiver Lambda-Schleifen

Grundlegendes zur Erkennung rekursiver Schleifen

Zur Erkennung rekursiver Schleifen in Lambda werden Ereignisse nachverfolgt. Lambda ist ein ereignisgesteuerter Compute-Service, der Ihren Funktionscode ausführt, wenn bestimmte Ereignisse auftreten. Ein Beispiel wäre etwa das Hinzufügen eines Elements zu einer Amazon-SQS-Warteschlange oder zu einem Amazon Simple Notification Service (Amazon SNS)-Thema. Ereignisse werden von Lambda als JSON-Objekte mit Informationen zur Änderung des Systemstatus

an Ihre Funktion übergeben. Die Ausführung Ihrer Funktion infolge eines Ereignisses wird als Aufruf bezeichnet.

Zur Erkennung rekursiver Schleifen verwendet Lambda AWS X-Ray-Ablaufverfolgungs-Header. Wenn AWS-Services , die die Erkennung rekursiver Schleifen unterstützen, Ereignisse an Lambda senden, werden diese Ereignisse automatisch mit Metadaten versehen. Wenn Ihre Lambda-Funktion eines dieser Ereignisse AWS-Service mithilfe einer unterstützten Version eines AWS SDK in ein anderes Ereignis schreibt, aktualisiert sie diese Metadaten. In den aktualisierten Metadaten ist angegeben, wie oft das Ereignis die Funktion aufgerufen hat.

> **Note**
>
> Dieses Feature funktioniert ohne Aktivierung der aktiven X-Ray-Ablaufverfolgung. Die Erkennung rekursiver Schleifen ist standardmäßig für alle AWS -Kunden aktiviert. Die Nutzung des Features ist kostenlos.

Bei einer Kette von Anforderungen handelt es sich um eine Sequenz von Lambda-Aufrufen, die durch das gleiche auslösende Ereignis verursacht werden. Ein Beispiel: Angenommen, eine Amazon-SQS-Warteschlange ruft Ihre Lambda-Funktion auf. Anschließend sendet Ihre Lambda-Funktion das verarbeitete Ereignis an die gleiche Amazon-SQS-Warteschlange zurück und diese ruft wiederum erneut Ihre Funktion auf. In diesem Beispiel gehört jeder Aufruf Ihrer Funktion zur gleichen Kette von Anforderungen.

Wenn Ihre Funktion ungefähr 16-mal in der gleichen Kette von Anforderungen aufgerufen wird, beendet Lambda automatisch den nächsten Funktionsaufruf in dieser Anforderungskette und benachrichtigt Sie. Wenn Ihre Funktion mit mehreren Auslösern konfiguriert ist, sind Aufrufe von anderen Auslösern nicht betroffen.

> **Note**
>
> Selbst wenn die `maxReceiveCount`-Einstellung in der Redrive-Richtlinie der Quellwarteschlange höher als 16 ist, verhindert der Lambda-Rekursionsschutz nicht, dass Amazon SQS die Nachricht erneut versucht, nachdem eine rekursive Schleife erkannt und beendet wurde. Wenn Lambda eine rekursive Schleife erkennt und nachfolgende Aufrufe abbricht, gibt es ein `RecursiveInvocationException` an die Zuordnung von Ereignisquellen zurück. Dadurch wird der `receiveCount`-Wert in der Nachricht erhöht. Lambda versucht weiterhin, die Nachricht zu wiederholen, und blockiert weiterhin

Funktionsaufrufe, bis Amazon SQS feststellt, dass das `maxReceiveCount` überschritten wurde, und die Nachricht an die konfigurierte Warteschlange für unzustellbare Nachrichten sendet.

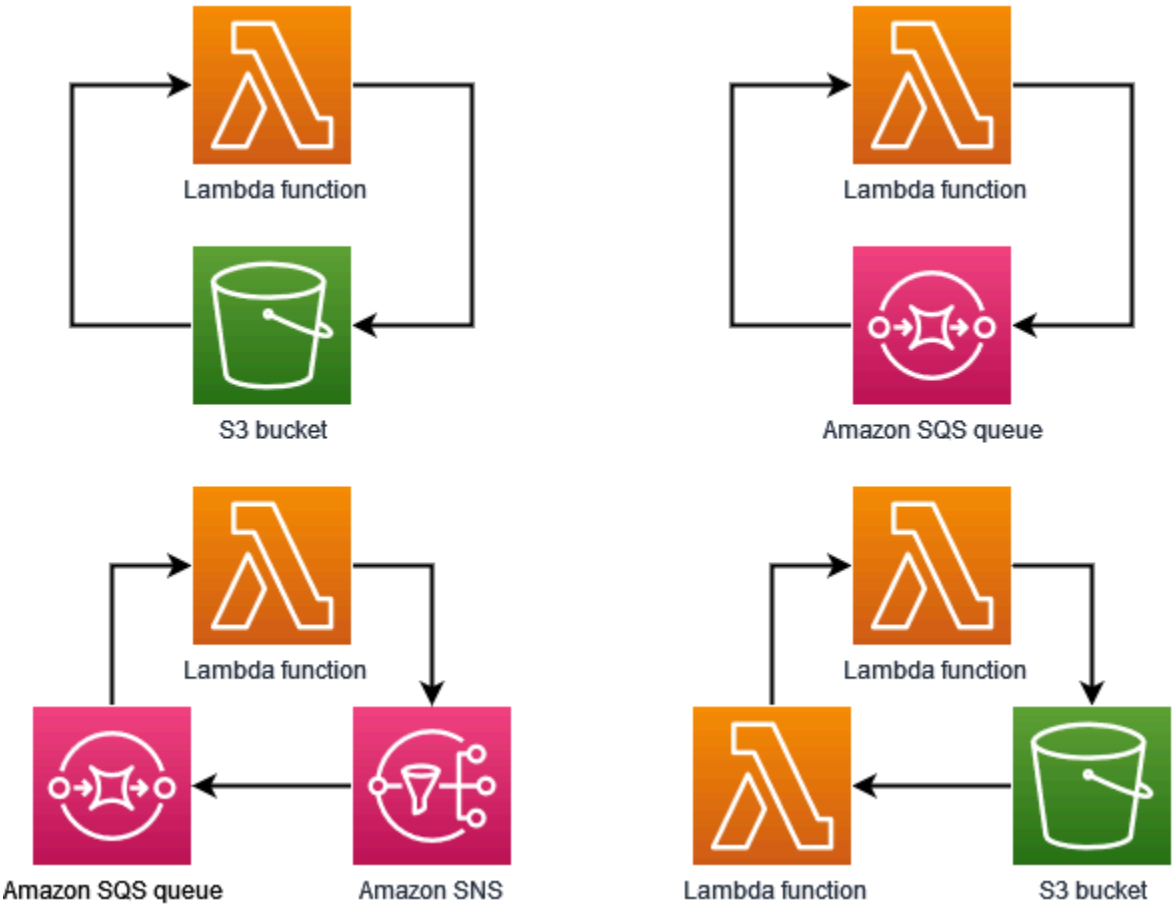
Wenn Sie für Ihre Funktion ein Ziel bei Ausfall oder eine Warteschlange für unzustellbare Nachrichten konfiguriert haben, sendet Lambda das Ereignis aus dem beendeten Aufruf auch an Ihr Ziel oder an die Warteschlange für unzustellbare Nachrichten. Wenn Sie für Ihre Funktion ein Ziel oder eine Warteschlange für unzustellbare Nachrichten konfigurieren, achten Sie darauf, keinen Ereignisauslöser und keine Zuordnung von Ereignisquellen zu verwenden, die Ihre Funktion auch verwendet. Wenn Sie Ereignisse an die Ressource senden, die auch Ihre Funktion aufruft, können Sie eine weitere rekursive Schleife erstellen und diese Schleife wird auch beendet. Wenn Sie die Rekursionsschleifenerkennung deaktivieren, wird diese Schleife nicht beendet.

Unterstützt AWS-Services und SDKs

Lambda kann nur rekursive Schleifen erkennen, die bestimmte unterstützte enthalten. AWS-Services Damit rekursive Schleifen erkannt werden können, muss Ihre Funktion auch eine der unterstützten Schleifen verwenden. AWS SDKs

Unterstützt AWS-Services

Lambda erkennt derzeit rekursive Schleifen zwischen Funktionen, Amazon SQS, Amazon S3 und Amazon SNS. Außerdem erkennt Lambda Schleifen, die nur aus Lambda-Funktionen bestehen, die sich synchron oder asynchron gegenseitig aufrufen. Die folgenden Diagramme zeigen einige Beispiele für Schleifen, die von Lambda erkannt werden:



Wenn ein anderes AWS-Service System wie Amazon DynamoDB Teil der Schleife ist, kann Lambda es derzeit nicht erkennen und stoppen.

Da Lambda derzeit nur rekursive Schleifen erkennt, an denen Amazon SQS, Amazon S3 und Amazon SNS beteiligt sind, ist es immer noch möglich, dass Schleifen, an denen andere beteiligt sind, zu einer unbeabsichtigten Nutzung Ihrer Lambda-Funktionen führen AWS-Services können.

Um zu verhindern, dass Ihnen unerwartete Gebühren in Rechnung gestellt werden AWS-Konto, empfehlen wir Ihnen, CloudWatchAmazon-Alarme so zu konfigurieren, dass Sie auf ungewöhnliche Nutzungsmuster aufmerksam gemacht werden. Sie können beispielsweise so konfigurieren, dass Sie über CloudWatch Spitzenwerte bei der Parallelität oder bei Aufrufen von Lambda-Funktionen benachrichtigt werden. Des Weiteren können Sie einen Abrechnungsalarm konfigurieren, um benachrichtigt zu werden, wenn die Ausgaben für Ihr Konto einen von Ihnen angegebenen Schwellenwert übersteigen. Eine weitere Möglichkeit ist die Verwendung der AWS Cost Anomaly Detection, um über ungewöhnliche Abrechnungsmuster informiert zu werden.

Unterstützt AWS SDKs

Damit Lambda rekursive Schleifen erkennt, muss Ihre Funktion eine der folgenden SDK-Mindestversionen verwenden:

Laufzeit	Minimale erforderliche AWS SDK-Version
Node.js	2.1147.0 (SDK-Version 2) 3.105.0 (SDK-Version 3)
Python	1.24.46 (boto3) 1.27.46 (botocore)
Java 8 und Java 11	2.17.135
Java 17	2,20,81
Java 21	2,21,24
.NET	3,7,293,0
Ruby	3,134,0
PHP	3,232,0
Go	V2 SDK 1.57.0

Einige Lambda-Laufzeiten wie Python und Node.js enthalten eine Version des AWS SDK. Wenn die in der Laufzeit Ihrer Funktion enthaltene SDK-Version kleiner ist als die erforderliche Mindestversion, können Sie dem Bereitstellungspaket Ihrer Funktion eine unterstützte Version des SDK hinzufügen. Sie können auch eine Lambda-Ebene verwenden, um Ihrer Funktion eine unterstützte SDK-Version hinzuzufügen. Eine Liste der in jeder Lambda-Laufzeit SDKs enthaltenen Funktionen finden Sie unterLambda-Laufzeiten.

Benachrichtigungen zu rekursiven Schleifen

Wenn Lambda eine rekursive Schleife beendet, erhalten Sie Benachrichtigungen über das Health Dashboard und per E-Mail. Sie können auch CloudWatch Metriken verwenden, um die Anzahl der rekursiven Aufrufe zu überwachen, die Lambda gestoppt hat.

Health Dashboard Benachrichtigungen

Wenn Lambda einen rekursiven Aufruf stoppt, wird auf der Seite „Ihr Kontostatus“ unter „Offene Probleme“ und „Aktuelle Probleme“ eine Benachrichtigung Health Dashboard angezeigt. Beachten Sie, dass es nach der Beendigung eines rekursiven Aufrufs durch Lambda bis zu 3,5 Stunden dauern kann, bis diese Benachrichtigung angezeigt wird. Weitere Informationen zum Anzeigen von Kontoereignissen in der Health Dashboard finden Sie unter Erste Schritte mit Ihrem AWS Health Dashboard — Ihr Kontostatus im AWS Health-Benutzerhandbuch.

E-Mail-Benachrichtigungen

Wenn Lambda zum ersten Mal einen rekursiven Aufruf Ihrer Funktion beendet, erhalten Sie eine E-Mail-Benachrichtigung. Für jede Funktion in Ihrem AWS-Konto wird alle 24 Stunden maximal eine E-Mail gesendet. Nachdem Lambda eine E-Mail-Benachrichtigung gesendet hat, erhalten Sie für einen Zeitraum von 24 Stunden keine E-Mails mehr zu dieser Funktion. Das gilt auch, wenn Lambda weitere rekursive Aufrufe der Funktion beendet. Beachten Sie, dass es nach der Beendigung eines rekursiven Aufrufs durch Lambda bis zu 3,5 Stunden dauern kann, bis Sie diese E-Mail-Benachrichtigung erhalten.

Lambda sendet rekursive Loop-E-Mail-Benachrichtigungen an Ihren primären AWS-Konto Kundenkontakt und an den alternativen Betriebskontakt. Informationen zum Anzeigen oder Aktualisieren der E-Mail-Adressen in Ihrem Konto finden Sie in der allgemeinen AWS -Referenz unter Updating contact information.

CloudWatch Amazon-Metriken

Die CloudWatch Metrik `RecursiveInvocationsDropped` zeichnet die Anzahl der Funktionsaufrufen auf, die Lambda gestoppt hat, weil Ihre Funktion in einer einzigen Anforderungskette mehr als etwa 16 Mal aufgerufen wurde. Lambda gibt diese Metrik aus, sobald ein rekursiver Aufruf beendet wurde. Um diese Metrik anzuzeigen, folgen Sie den Anweisungen unter Metriken anzeigen auf der CloudWatch Konsole und wählen Sie die Metrik aus. `RecursiveInvocationsDropped`

Reagieren auf Benachrichtigungen im Zusammenhang mit der Erkennung rekursiver Schleifen

Wenn Ihre Funktion etwa mehr als 16-mal durch das gleiche auslösende Ereignis aufgerufen wird, beendet Lambda den nächsten Funktionsaufruf für dieses Ereignis, um die rekursive Schleife zu unterbrechen. Gehen Sie wie folgt vor, um zu verhindern, dass eine rekursive Schleife, die von Lambda unterbrochen wurde, erneut auftritt:

- Reduzieren Sie die verfügbare Parallelität Ihrer Funktion auf Null. Dadurch werden alle zukünftigen Aufrufe gedrosselt.
- Entfernen oder deaktivieren Sie den Auslöser oder die Zuordnung von Ereignisquellen, der bzw. die Ihre Funktion aufruft.
- Identifizieren und beheben Sie Codefehler, die Ereignisse in die AWS Ressource zurückschreiben, die Ihre Funktion aufruft. Eine Fehlerquelle entsteht häufig, wenn die Ereignisquelle und das Ziel einer Funktion mithilfe von Variablen definiert werden. Achten Sie darauf, nicht für beide Variablen den gleichen Wert zu verwenden.

Wenn die Ereignisquelle für Ihre Lambda-Funktion eine Amazon-SQS-Warteschlange ist, empfiehlt es sich außerdem gegebenenfalls, für die Quellwarteschlange eine Warteschlange für unzustellbare Nachrichten zu konfigurieren.

> **Note**
>
> Achten Sie darauf, die Warteschlange für unzustellbare Nachrichten für die Quellwarteschlange und nicht für die Lambda-Funktion zu konfigurieren. Die Warteschlange für unzustellbare Nachrichten, die Sie für eine Funktion konfigurieren, wird für die Warteschlange asynchroner Aufrufe der Funktion und nicht für Ereignisquellen-Warteschlangen verwendet.

Wenn es sich bei der Ereignisquelle um ein Amazon-SNS-Thema handelt, empfiehlt es sich gegebenenfalls, für Ihre Funktion ein Ziel bei Ausfall hinzuzufügen.

So reduzieren Sie die verfügbare Parallelität Ihrer Funktion auf Null (Konsole)

1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.
2. Wählen Sie den Namen Ihrer Funktion aus.

3. Wählen Sie Drosseln aus.
4. Wählen Sie im Dialogfeld Funktion drosseln die Option Bestätigen aus.

So entfernen Sie einen Auslöser oder eine Zuordnung von Ereignisquellen für Ihre Funktion (Konsole)

1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.
2. Wählen Sie den Namen Ihrer Funktion aus.
3. Wählen Sie die Registerkarte Konfiguration und anschließend Auslöser aus.
4. Wählen Sie unter Auslöser den Auslöser oder die Zuordnung von Ereignisquellen aus, den bzw. die Sie löschen möchten, und wählen Sie anschließend Löschen aus.
5. Wählen Sie im Dialogfeld Auslöser löschen die Option Löschen aus.

So deaktivieren Sie eine Zuordnung von Ereignisquellen für Ihre Funktion (AWS CLI)

1. Führen Sie den Befehl AWS Command Line Interface (AWS CLI) aus, um die UUID für die Ereignisquellenzuordnung zu finden, die Sie deaktivieren möchten. list-event-source-mappings

   ```
   aws lambda list-event-source-mappings
   ```

2. Führen Sie den folgenden AWS CLI update-event-source-mappingBefehl aus, um die Zuordnung der Ereignisquellen zu deaktivieren.

   ```
   aws lambda update-event-source-mapping --function-name MyFunction \
   --uuid a1b2c3d4-5678-90ab-cdef-EXAMPLE11111 --no-enabled
   ```

Zulassen, dass eine Lambda-Funktion in einer rekursiven Schleife ausgeführt wird

Wenn Ihr Entwurf absichtlich eine rekursive Schleife verwendet, können Sie eine Lambda-Funktion so konfigurieren, dass sie rekursiv aufgerufen werden kann. Wir empfehlen, dass Sie rekursive Schleifen in Ihrem Entwurf vermeiden. Implementierungsfehler können dazu führen, dass rekursive Aufrufe Ihre gesamte verfügbare Parallelität nutzen und dass Ihrem AWS-Konto Konto unerwartete Gebühren in Rechnung gestellt werden.

> **Important**
>
> Wenn Sie rekursive Schleifen verwenden, sollten Sie diese mit Vorsicht behandeln. Implementieren Sie bewährte Leitplanken, um das Risiko von Implementierungsfehlern zu minimieren. Weitere Informationen zu bewährten Methoden für die Verwendung rekursiver Muster finden Sie bei Serverless Land unter Recursive patterns that cause run-away Lambda functions.

Mithilfe der Lambda-Konsole, der AWS Command Line Interface (AWS CLI) und der API können Sie Funktionen so konfigurieren, dass sie rekursive Schleifen zulassen. PutFunctionRecursionConfig Sie können auch die Einstellung für die Erkennung rekursiver Schleifen einer Funktion in und konfigurieren. AWS SAM CloudFormation

Standardmäßig erkennt und beendet Lambda rekursive Schleifen. Sofern Ihr Entwurf nicht bewusst eine rekursive Schleife verwendet, empfehlen wir Ihnen, die Standardkonfiguration Ihrer Funktionen nicht zu ändern.

Beachten Sie, dass die CloudWatch Metrik `RecursiveInvocationsDropped` nicht ausgegeben wird, wenn Sie eine Funktion so konfigurieren, dass sie rekursive Schleifen zulässt.

Console

So können Sie eine Funktion in einer rekursiven Schleife ausführen (Konsole)

1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.
2. Wählen Sie den Namen Ihrer Funktion, um die Funktionsdetailseite zu öffnen.
3. Wählen Sie die Registerkarte Konfiguration und anschließend Gleichzeitigkeit und Rekursionserkennung.
4. Wählen Sie neben Rekursive Schleifenerkennung die Option Bearbeiten aus.
5. Wählen Sie Rekursive Schleifen zulassen aus.
6. Wählen Sie Speichern.

AWS CLI

Sie können die PutFunctionRecursionConfigAPI verwenden, um zu ermöglichen, dass Ihre Funktion in einer rekursiven Schleife aufgerufen wird. Geben Sie `Allow` als Parameter für die

rekursive Schleife an. Sie können diese API beispielsweise mit dem folgenden Befehl aufrufen: `put-function-recursion-config` AWS CLI

```
aws lambda put-function-recursion-config --function-name yourFunctionName --
recursive-loop Allow
```

Sie können die Konfiguration Ihrer Funktion auf die Standardeinstellung zurücksetzen, sodass Lambda rekursive Schleifen beendet, wenn es sie erkennt. Bearbeiten Sie die Konfiguration Ihrer Funktion mit der Lambda-Konsole oder dem AWS CLI.

Console

So konfigurieren Sie eine Funktion so, dass rekursive Schleifen abgebrochen werden (Konsole)

1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.
2. Wählen Sie den Namen Ihrer Funktion, um die Funktionsdetailseite zu öffnen.
3. Wählen Sie die Registerkarte Konfiguration und anschließend Gleichzeitigkeit und Rekursionserkennung.
4. Wählen Sie neben Rekursive Schleifenerkennung die Option Bearbeiten aus.
5. Wählen Sie Rekursive Schleifen beenden aus.
6. Wählen Sie Speichern.

AWS CLI

Sie können die PutFunctionRecursionConfigAPI verwenden, um Ihre Funktion so zu konfigurieren, dass Lambda rekursive Schleifen beendet, wenn es sie erkennt. Geben Sie `Terminate` als Parameter für die rekursive Schleife an. Sie können diese API beispielsweise mit dem folgenden Befehl aufrufen: `put-function-recursion-config` AWS CLI

```
aws lambda put-function-recursion-config --function-name yourFunctionName --
recursive-loop Terminate
```

Unterstützte Regionen für die Erkennung rekursiver Lambda-Schleifen

Die Lambda-Rekursivschleifenerkennung wird in allen Handelsregionen außer Mexiko (Zentral) und Asien-Pazifik (Neuseeland) unterstützt.

Lambda-Funktion erstellen und verwalten URLs

Eine Funktions-URL ist ein dedizierter HTTPS-Endpunkt für Ihre Lambda-Funktion. Sie können eine Funktions-URL über die Lambda-Konsole oder die Lambda-API erstellen und konfigurieren.

> **Tip**
>
> Lambda bietet zwei Möglichkeiten, Ihre Funktion über einen HTTP-Endpunkt aufzurufen: function URLs und Amazon API Gateway. Wenn Sie sich nicht sicher sind, welche Methode für Ihren Anwendungsfall am besten geeignet ist, siehe the section called “Funktion im URLs Vergleich zu Amazon API Gateway”.

Wenn Sie eine Funktions-URL erstellen, generiert Lambda automatisch einen eindeutigen URL-Endpunkt für Sie. Sobald Sie eine Funktions-URL erstellt haben, ändert sich ihr URL-Endpunkt nie mehr. Funktions-URL-Endpunkte haben das folgende Format:

```
https://<url-id>.lambda-url.<region>.on.aws
```

> **Note**
>
> Funktionen URLs werden in den folgenden Ländern nicht unterstützt AWS-Regionen: Asien-Pazifik (Hyderabad) (`ap-south-2`), Asien-Pazifik (Melbourne) (`ap-southeast-4`), Asien-Pazifik (Malaysia`ap-southeast-5`) (), Asien-Pazifik (Neuseeland) (`ap-southeast-6`), Asien-Pazifik (Thailand) (`ap-southeast-7`), Asien-Pazifik (Taipeh`ap-east-2`) (), Kanada West (Calgary`ca-west-1`) (), Europa (Spanien) (`eu-south-2`), Europa (Zürich) (`eu-central-2`), Israel (Tel Aviv) (`il-central-1`) und Naher Osten (VAE) (`me-central-1`).

URLs Die Funktionen sind Dual-Stack-fähig, unterstützen IPv4 und IPv6. Nachdem Sie eine Funktions-URL für Ihre Funktion konfiguriert haben, können Sie Ihre Funktion über ihren HTTP(S)-Endpunkt über einen Webbrowser, curl, Postman oder einen beliebigen HTTP-Client aufrufen.

> **Note**
>
> Auf Ihre Funktions-URL können Sie nur über das öffentliche Internet zugreifen. Lambda-Funktionen unterstützen zwar AWS PrivateLink, Funktionen jedoch URLs nicht.

Die Lambda-Funktion URLs verwendet ressourcenbasierte Richtlinien für Sicherheit und Zugriffskontrolle. Die Funktion unterstützt URLs auch CORS-Konfigurationsoptionen (Cross-Origin Resource Sharing).

Sie können die Funktion URLs auf einen beliebigen Funktionsalias oder auf die `$LATEST` unveröffentlichte Funktionsversion anwenden. Sie können keiner anderen Funktionsversion eine Funktions-URL hinzufügen.

Im folgenden Abschnitt wird gezeigt, wie Sie eine Funktions-URL mithilfe der Lambda-Konsole AWS CLI, und der Vorlage erstellen und CloudFormation verwalten

Themen

- Erstellen einer Funktions-URL (Konsole)
- Erstellen einer Funktions-URL (AWS CLI)
- Hinzufügen einer Funktions-URL zu einer CloudFormation Vorlage
- Cross-Origin Resource Sharing (CORS)
- Drosselungsfunktion URLs
- Funktion wird deaktiviert URLs
- Funktion löschen URLs
- Steuern Sie den Zugriff auf die Lambda-Funktion URLs
- Aufrufen von Lambda-Funktions-URLs
- Überwachen der Lambda-Funktions-URLs
- Wählen Sie eine Methode, um Ihre Lambda-Funktion über eine HTTP-Anfrage aufzurufen
- Tutorial: Erstellen eines Webhook-Endpunkts mithilfe einer Lambda-Funktions-URL

Erstellen einer Funktions-URL (Konsole)

Gehen Sie wie folgt vor, um eine Funktions-URL mit der Konsole zu erstellen.

So erstellen Sie eine Funktions-URL für eine vorhandene Funktion

1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.
2. Wählen Sie den Namen der Funktion, für die Sie die Funktions-URL erstellen möchten.
3. Wählen Sie die Registerkarte Konfiguration und dann Funktions-URL.
4. Wählen Sie Funktions-URL erstellen.

5. Wählen Sie als Authentifizierungstyp AWS_IAModer NONE. Weitere Informationen zur Authentifizierung der Funktions-URL finden Sie unter Zugriffskontrolle.
6. (Optional) Wählen Sie Configure cross-origin resource sharing (CORS) (Cross-Origin Resource Sharing (CORS) konfigurieren) aus und konfigurieren Sie dann die CORS-Einstellungen für Ihre Funktions-URL. Weitere Informationen über CORS finden Sie unter Cross-Origin Resource Sharing (CORS).
7. Wählen Sie Speichern.

Dies erstellt eine Funktions-URL für die `$LATEST` unveröffentlichte Version Ihrer Funktion. Die Funktions-URL wird im Abschnitt Function overview (Funktionsübersicht) der Konsole angezeigt.

So erstellen Sie eine Funktions-URL für einen vorhandenen Alias

1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.
2. Wählen Sie den Namen der Funktion mit dem Alias, für die Sie die Funktions-URL erstellen möchten.
3. Wählen Sie die Registerkarte Aliase und wählen Sie dann den Namen der Funktion mit dem Alias, für die Sie die Funktions-URL erstellen möchten.
4. Wählen Sie die Registerkarte Konfiguration und dann Funktions-URL.
5. Wählen Sie Funktions-URL erstellen.
6. Wählen Sie als Authentifizierungstyp AWS_IAModer NONE. Weitere Informationen zur Authentifizierung der Funktions-URL finden Sie unter Zugriffskontrolle.
7. (Optional) Wählen Sie Configure cross-origin resource sharing (CORS) (Cross-Origin Resource Sharing (CORS) konfigurieren) aus und konfigurieren Sie dann die CORS-Einstellungen für Ihre Funktions-URL. Weitere Informationen über CORS finden Sie unter Cross-Origin Resource Sharing (CORS).
8. Wählen Sie Speichern.

Dies erstellt eine Funktions-URL für Ihren Funktionsalias. Die Funktions-URL wird im Abschnitt Funktionsübersicht der Konsole für Ihren Alias angezeigt.

So erstellen Sie eine neue Funktion mit einem Funktions-URL

So erstellen Sie eine neue Funktion mit einer Funktions-URL (Konsole)

1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.

2. Wählen Sie Funktion erstellen.
3. Führen Sie unter Basic information (Grundlegende Informationen) die folgenden Schritte aus:
   a. Geben Sie unter Funktionsname einen Namen für Ihre Funktion ein, z. B. **`my-function`**.
   b. Wählen Sie für Runtime die Laufzeitsprache aus, die Sie bevorzugen, z. B. Node.js 24.
   c. Wählen Sie für Architecture (Architektur) entweder x86_64 oder arm64 aus.
   d. Erweitern Sie Permissions (Berechtigungen) und wählen Sie dann aus, ob eine neue Ausführungsrolle erstellt oder eine vorhandene Rolle verwendet werden soll.
4. Erweitern Sie Advanced settings (Erweiterte Einstellungen) und wählen Sie dann Funktions-URL aus.
5. Wählen Sie für Auth type AWS_IAModer NONE. Weitere Informationen zur Authentifizierung der Funktions-URL finden Sie unter Zugriffskontrolle.
6. (Optional) Wählen Sie Cross-Origin Resource Sharing (CORS) konfigurieren aus. Wenn Sie diese Option während der Funktionserstellung auswählen, lässt Ihre Funktions-URL standardmäßig sämtliche Anfragen zu. Sie können die CORS-Einstellungen für Ihre Funktions-URL bearbeiten, nachdem Sie die Funktion erstellt haben. Weitere Informationen über CORS finden Sie unter Cross-Origin Resource Sharing (CORS).
7. Wählen Sie Funktion erstellen.

Dies erstellt eine neue Funktions mit einer Funktions-URL für die `$LATEST` unveröffentlichte Version Ihrer Funktion. Die Funktions-URL wird im Abschnitt Function overview (Funktionsübersicht) der Konsole angezeigt.

Erstellen einer Funktions-URL (AWS CLI)

Führen Sie den folgenden Befehl aus, um eine Funktions-URL für eine bestehende Lambda-Funktion mithilfe von AWS Command Line Interface (AWS CLI) zu erstellen:

```
aws lambda create-function-url-config \
    --function-name my-function \
    --qualifier prod \ // optional
    --auth-type AWS_IAM
    --cors-config {AllowOrigins="https://example.com"} // optional
```

Dies fügt eine Funktions-URL zum **`prod`**-Qualifizierer für die Funktion**`my-function`** hinzu. Weitere Informationen zu diesen Konfigurationsparametern finden Sie unter CreateFunctionUrlConfig in der API-Referenz.

> **Note**
>
> Um eine Funktions-URL über die zu erstellen AWS CLI, muss die Funktion bereits existieren.

Hinzufügen einer Funktions-URL zu einer CloudFormation Vorlage

Verwenden Sie die folgende Syntax, um Ihrer CloudFormation Vorlage eine `AWS::Lambda::Url` Ressource hinzuzufügen:

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

- (Erforderlich) `AuthType` – Definiert den Typ der Authentifizierung für Ihre Funktions-URL. Die möglichen Werte sind `AWS_IAM` oder `NONE`. Um den Zugriff nur auf authentifizierte Benutzer zu

beschränken, setzen Sie den Wert auf `AWS_IAM`. Um die IAM-Authentifizierung zu umgehen und jedem Benutzer zu erlauben, Anfragen an Ihre Funktion zu stellen, setzten Sie dies auf `NONE`.

- (Optional)`Cors` – Definiert die CORS-Einstellungen für Ihre Funktions-URL. Verwenden Sie `Cors` die folgende Syntax CloudFormation, um Ihrer `AWS::Lambda::Url` Ressource in etwas hinzuzufügen.

  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
  ```

- (Optional) `Qualifier` – Der Aliasname.
- (Erforderlich) `TargetFunctionArn` – Der Name oder der Amazon-Ressourcenname (ARN) der Lambda-Funktion. Gültige Namensformate sind unter anderem:
  - Funktionsname – `my-function`.
  - Funktions-ARN – `arn:aws:lambda:us-west-2:123456789012:function:my-function`.
  - Partiellet ARN – `123456789012:function:my-function`.

Cross-Origin Resource Sharing (CORS)

Um zu definieren, wie verschiedene Ursprünge auf Ihre Funktions-URL zugreifen können, verwenden Sie Cross-Origin Resource Sharing (CORS). Wir empfehlen, CORS zu konfigurieren, wenn Sie beabsichtigen, Ihre Funktions-URL von einer anderen Domain aus aufzurufen. Lambda unterstützt die folgenden CORS-Header für Funktionen. URLs

CORS-Header	CORS-Konfigurationseigenschaft	Beispielwerte

Access-Control-Allow-Origin	`AllowOrigins`	`*` (alle Ursprünge zulassen)
		`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` (Standard), `300`

Wenn Sie CORS für eine Funktions-URL mithilfe der Lambda-Konsole oder der konfigurieren AWS CLI, fügt Lambda über die Funktions-URL automatisch die CORS-Header zu allen Antworten hinzu. Alternativ können Sie Ihrer Funktionsantwort manuell CORS-Header hinzufügen. Wenn es widersprüchliche Header gibt, ist das erwartete Verhalten abhängig vom Typ der Anfrage:

- Bei Preflight-Anfragen wie OPTIONS-Anfragen haben die konfigurierten CORS-Header auf der Funktions-URL Vorrang. Lambda gibt nur diese CORS-Header in der Antwort zurück.

- Bei Nicht-Preflight-Anforderungen wie GET- oder POST-Anforderungen gibt Lambda sowohl die konfigurierten CORS-Header der Funktions-URL als auch die von der Funktion zurückgegebenen CORS-Header zurück. Dies kann zu doppelten CORS-Header in der Antwort führen. Möglicherweise wird ein Fehler ähnlich dem folgenden angezeigt: `The 'Access-Control-Allow-Origin' header contains multiple values '*, *', but only one is allowed.`

Im Allgemeinen empfehlen wir, alle CORS-Einstellungen auf der Funktions-URL zu konfigurieren, anstatt CORS-Header manuell in der Funktionsantwort zu senden.

Drosselungsfunktion URLs

Die Drosselungsrate begrenzt die Geschwindigkeit, mit der Ihre Funktion Anfragen verarbeitet. Dies ist in vielen Situationen nützlich, z. B. um zu verhindern, dass Ihre Funktion nachgelagerte Ressourcen überlastet, oder beim Verarbeiten eines plötzlichen Anstiegs der Anfragen.

Sie können die Rate der Anfragen, die Ihre Lambda-Funktion über eine Funktions-URL verarbeitet, drosseln, indem Sie die reservierte Gleichzeitigkeit konfigurieren. Reservierte Gleichzeitigkeit begrenzt die Anzahl der maximalen gleichzeitigen Aufrufe für Ihre Funktion. Die maximale Anforderungsrate Ihrer Funktion pro Sekunde (RPS) entspricht dem Zehnfachen der konfigurierten reservierten Gleichzeitigkeit. Wenn Sie Ihre Funktion beispielsweise mit einer reservierten Gleichzeitigkeit von 100 konfigurieren, beträgt der maximale RPS 1 000.

Immer wenn Ihre Funktionsparallelität die reservierte Gleichzeitigkeit überschreitet, gibt Ihre Funktions-URL einen HTTP-429-Statuscode zurück. Wenn Ihre Funktion eine Anforderung erhält, die das 10-fache RPS-Maximum basierend auf Ihrer konfigurierten reservierten Gleichzeitigkeit überschreitet, erhalten Sie auch einen HTTP-429-Fehler. Weitere Hinweise zur reservierten Gleichzeitigkeit finden Sie unter Konfigurieren reservierter Gleichzeitigkeit für eine Funktion.

Funktion wird deaktiviert URLs

Im Notfall sollten Sie den gesamten Datenverkehr zu Ihrer Funktions-URL ablehnen. Um eine Funktion zu deaktivieren, setzen Sie die reservierte Gleichzeitigkeit auf Null. Dies drosselt alle Anfragen an Ihre Funktions-URL, was zu HTTP-429-Statusantworten führt. Um Ihre Funktions-URL zu reaktivieren, löschen Sie die Konfiguration für die reservierte Gleichzeitigkeit oder legen Sie die Konfiguration auf einen Betrag größer als null fest.

Funktion löschen URLs

Wenn Sie eine Funktions-URL löschen, können Sie sie nicht wiederherstellen. Das Erstellen einer neuen Funktions-URL führt zu einer anderen URL-Adresse.

> **Note**
>
> Wenn Sie Ihre Funktions-URL mit Authentifizierungstyp `NONE` löschen, löscht Lambda nicht automatisch die zugehörige ressourcenbasierte Richtlinie. Wenn Sie diese Richtlinie löschen möchten, müssen Sie dies manuell tun.

1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.
2. Wählen Sie den Namen einer Funktion aus.
3. Wählen Sie die Registerkarte Konfiguration und dann Funktions-URL.
4. Wählen Sie Löschen aus.
5. Geben Sie den Ausdruck delete (löschen) in das Feld ein, um den Löschvorgang zu bestätigen.
6. Wählen Sie Löschen aus.

> **Note**
>
> Wenn Sie eine Funktion löschen, die eine Funktions-URL hat, löscht Lambda asynchron die Funktions-URL. Wenn Sie sofort eine neue Funktion mit demselben Namen im selben Konto erstellen, ist es möglich, dass die ursprüngliche Funktions-URL auf die neue Funktion abgebildet wird, anstatt gelöscht zu werden.

Steuern Sie den Zugriff auf die Lambda-Funktion URLs

> **Note**
>
> Ab Oktober 2025 sind für neue Funktionen URLs sowohl Genehmigungen als auch `lambda:InvokeFunctionUrl` `lambda:InvokeFunction` Genehmigungen erforderlich.

Sie können den Zugriff auf Ihre Lambda-Funktion URLs mithilfe des AuthTypeParameters in Kombination mit ressourcenbasierten Richtlinien steuern, die an Ihre spezifische Funktion angehängt sind. Die Konfiguration dieser beiden Komponenten bestimmt, wer andere administrative Aktionen für Ihre Funktions-URL aufrufen oder ausführen kann.

Der `AuthType`-Parameter bestimmt, wie Lambda Anfragen an Ihre Funktions-URL authentifiziert oder autorisiert. Wenn Sie Ihre Funktions-URL konfigurieren, müssen Sie eine der folgenden `AuthType`-Optionen angeben:

- `AWS_IAM`— Lambda verwendet AWS Identity and Access Management (IAM), um Anfragen auf der Grundlage der Identitätsrichtlinie des IAM-Prinzipals und der ressourcenbasierten Richtlinie der Funktion zu authentifizieren und zu autorisieren. Wählen Sie diese Option, wenn Sie möchten, dass nur authentifizierte -Benutzer und -Rollen Ihre Funktion über die Funktions-URL aufrufen können.

- `NONE` – Lambda führt keine Authentifizierung durch, bevor Sie Ihre Funktion aufrufen. Die ressourcenbasierte Richtlinie Ihrer Funktion ist jedoch immer in Kraft und muss öffentlichen Zugriff gewähren, bevor Ihre Funktions-URL Anfragen erhalten kann. Wählen Sie diese Option, um öffentlichen, nicht authentifizierten Zugriff auf Ihre Funktions-URL zu ermöglichen.

Für zusätzliche Einblicke in die Sicherheit können Sie eine umfassende Analyse des externen AWS Identity and Access Management Access Analyzer Zugriffs auf Ihre Funktions-URL nutzen. IAM Access Analyzer überwacht auch auf neue oder aktualisierte Berechtigungen für Ihre Lambda-Funktionen, um Berechtigungen zu identifizieren, die öffentlichen und kontoübergreifenden Zugriff gewähren. Sie können IAM Access Analyzer kostenlos nutzen. Weitere Informationen zu den ersten Schritten mit IAM Access Analyzer finden Sie unter Verwenden des AWS IAM Access Analyzer.

Diese Seite enthält Beispiele für ressourcenbasierte Richtlinien für beide Authentifizierungstypen und wie diese Richtlinien mithilfe des AddPermissionAPI-Vorgangs oder der Lambda-Konsole erstellt werden. Informationen darüber, wie Sie Ihre Funktions-URL aufrufen können, nachdem Sie Berechtigungen eingerichtet haben, finden Sie unter Aufrufen von Lambda-Funktions-URLs.

Themen

- Verwenden des Auth-Typs AWS_IAM
- Verwenden des Authentifizierungstyps NONE
- Governance und Zugriffskontrolle

Verwenden des Auth-Typs **AWS_IAM**

Wenn Sie den Authentifizierungstyp `AWS_IAM` auswählen, müssen Benutzer, die Ihre Lambda-Funktions-URL aufrufen, die Berechtigungen `lambda:InvokeFunctionUrl` und `lambda:InvokeFunction` haben. Je nachdem, wer die Aufrufanforderung stellt, müssen Sie diese Berechtigung möglicherweise mithilfe einer ressourcenbasierten Richtlinie erteilen.

Wenn sich der Principal, der die Anfrage stellt, in derselben URL befindet AWS-Konto wie die Funktions-URL, muss der Principal entweder über `lambda:InvokeFunctionUrl` `lambda:InvokeFunction` Berechtigungen in seiner identitätsbasierten Richtlinie verfügen oder ihm in der ressourcenbasierten Richtlinie der Funktion Berechtigungen erteilt worden sein. Mit anderen Worten, eine ressourcenbasierte Richtlinie ist optional, wenn der Benutzer bereits die Berechtigungen `lambda:InvokeFunctionUrl` und `lambda:InvokeFunction` in seiner identitätsbasierten Richtlinie hat. Die Richtlinienbewertung folgt den Regeln, die in der Auswertungslogik für Richtlinien beschrieben sind.

Wenn der Prinzipal, der die Anfrage stellt, in einem anderen Konto ist, muss der Prinzipal sowohl eine identitätsbasierte Richtlinie haben, die ihm die Berechtigungen `lambda:InvokeFunctionUrl` und `lambda:InvokeFunction` gewährt, als auch Berechtigungen, die ihm in einer ressourcenbasierten Richtlinie für die Funktion gewährt werden, die er aufrufen möchte. Die Richtlinienbewertung folgt den im Abschnitt Festlegen, ob eine kontoübergreifende Anforderung zulässig ist erläuterten Regeln.

Die folgende ressourcenbasierte Richtlinie ermöglicht es der `example` Rolle in, die der Funktion AWS-Konto `444455556666` zugeordnete Funktions-URL aufzurufen. `my-function` Der Schlüssel lambda: InvokedViaFunctionUrl context beschränkt die `lambda:InvokeFunction` Aktion auf URL-Aufrufe von Funktionen. Das bedeutet, dass der Prinzipal die Funktions-URL verwenden muss, um die Funktion aufzurufen. Wenn Sie `lambda:InvokedViaFunctionUrl` nicht einfügen, kann der Prinzipal Ihre Funktion zusätzlich zur Funktions-URL auch über andere Aufrufmethoden aufrufen.

Example – Kontoübergreifende ressourcenbasierte Richtlinie

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"
        }
      }
    }
  ]
}
```

Sie können diese ressourcenbasierte Richtlinie über die Konsole erstellen, indem Sie die folgenden Schritte ausführen:

So erteilen Sie einem anderen Konto (Konsole) URL-Aufrufberechtigungen

1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.
2. Wählen Sie den Namen der Funktion aus, für die Sie URL-Aufrufberechtigungen gewähren möchten.
3. Wählen Sie die Registerkarte Konfiguration und dann Berechtigungen aus.
4. Wählen Sie unter Resource-based policy (Ressourcenbasierte Richtlinie) die Option Add permissions (Berechtigungen hinzufügen) aus.
5. Wählen Sie Funktions-URL aus.
6. Wählen Sie für Auth type die Option. AWS_IAM

7. Geben Sie eine Anweisungs-ID für Ihre Richtlinienanweisung ein.
8. Geben Sie unter Prinzipal den Amazon-Ressourcennamen (ARN) des Benutzers oder der Rolle ein, dem bzw. der Sie Berechtigungen gewähren möchten. Beispiel: **444455556666**.
9. Wählen Sie Speichern.

Alternativ können Sie diese Richtlinie mit den folgenden add-permission AWS Command Line Interface (AWS CLI) -Befehlen erstellen. Wenn Sie die verwenden AWS CLI, müssen Sie die `lambda:InvokeFunction` Anweisungen `lambda:InvokeFunctionUrl` und separat hinzufügen. Beispiel:

```
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
```

Verwenden des Authentifizierungstyps **NONE**

> **Important**
>
> Wenn Ihre Funktions-URL-Auth-Typ `NONE` ist und Sie eine ressourcenbasierte Richtlinie haben, die öffentlichen Zugriff gewährt, kann jeder nicht authentifizierte Benutzer mit Ihrer Funktions-URL Ihre Funktion aufrufen.

In einigen Fällen möchten Sie möglicherweise, dass Ihre Funktions-URL öffentlich ist. So können Sie beispielsweise Anfragen direkt von einem Webbrowser aus bereitstellen. Um den öffentlichen Zugriff auf Ihre Funktions-URL zu ermöglichen, wählen Sie den Auth-Typ `NONE` aus.

Wenn Sie den Auth-Typ `NONE` auswählen, verwendet Lambda IAM nicht, um Anfragen an Ihre Funktions-URL zu authentifizieren. Ihre Funktion muss jedoch über eine ressourcenbasierte Richtlinie verfügen, die `lambda:InvokeFunctionUrl` und `lambda:InvokeFunction` erlaubt. Wenn Sie

NONE mithilfe der Konsole oder AWS Serverless Application Model (AWS SAM) eine Funktions-URL mit dem Authentifizierungstyp erstellen, erstellt Lambda automatisch die ressourcenbasierte Richtlinie für Sie. Wenn Sie die AWS CLI AWS CloudFormation, oder die Lambda-API direkt verwenden, müssen Sie die Richtlinie selbst hinzufügen.

Wir empfehlen, dass Sie den InvokedViaFunctionUrl Kontextschlüssel lambda: in Ihre ressourcenbasierten Richtlinien aufnehmen, wenn Sie den Authentifizierungstyp verwenden. NONE Dieser Kontextschlüssel stellt sicher, dass die Funktion nur über die Funktions-URL und nicht über andere Aufrufmethoden aufgerufen werden kann.

Beachten Sie die folgenden Informationen über diese Richtlinie:

- Alle Entitäten können `lambda:InvokeFunctionUrl` und `lambda:InvokeFunction` aufrufen. Das bedeutet, dass jeder, der Ihre Funktions-URL hat, Ihre Funktion aufrufen kann.
- Der Wert des `lambda:FunctionUrlAuthType`-Bedingungsschlüssels ist NONE. Das bedeutet, dass diese Richtlinienanweisung den Zugriff nur erlaubt, wenn der Authentifizierungstyp Ihrer Funktions-URL ebenfalls NONE ist.
- Die Bedingung `lambda:InvokedViaFunctionUrl` stellt sicher, dass die Funktion nur über die Funktions-URL und nicht über andere Aufrufmethoden aufgerufen werden kann.

Example – Ressourcenbasierte Standardrichtlinie für den Authentifizierungstyp 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"
        }
      }
    }
  ]
}
```

Erstellen Sie die ressourcenbasierte Richtlinie mithilfe des AWS CLI

Sofern Sie nicht die Konsole verwenden oder AWS SAM eine Funktions-URL mit Authentifizierungstyp erstellen`NONE`, müssen Sie die ressourcenbasierte Richtlinie selbst hinzufügen. Verwenden Sie die folgenden Befehle zur Erstellung von Anweisungen für die Berechtigungen `lambda:InvokeFunctionUrl` und `lambda:InvokeFunction`. Jede Anweisung muss in einem separaten Befehl hinzugefügt werden.

```
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**
>
> Wenn Sie Ihre Funktions-URL mit Authentifizierungstyp `NONE` löschen, löscht Lambda nicht automatisch die zugehörige ressourcenbasierte Richtlinie. Wenn Sie diese Richtlinie löschen möchten, müssen Sie dies manuell tun.

Wenn die ressourcenbasierte Richtlinie einer Funktion die Berechtigungen `lambda:invokeFunctionUrl` und `lambda:InvokeFunction` nicht gewährt, dann erhalten Benutzer einen 403-Forbidden-Fehlercode beim Versuch, Ihre Funktions-URL aufzurufen. Das geschieht auch dann, wenn die Funktions-URL den Authentifizierungstyp `NONE` verwendet.

Governance und Zugriffskontrolle

Neben den Zugriffsberechtigungen für den Aufruf von Funktions-URLs können Sie auch den Zugriff auf Aktionen steuern, die zur Konfiguration der Funktion URLs verwendet werden. Lambda unterstützt die folgenden IAM-Richtlinienaktionen für die Funktion: URLs

- `lambda:InvokeFunctionUrl` – Ruft eine Lambda-Funktion mit der Funktions-URL auf.
- `lambda:CreateFunctionUrlConfig` – Erstellt eine Funktions-URL und legt ihren `AuthType` fest.
- `lambda:UpdateFunctionUrlConfig` – Aktualisiert eine Funktions-URL-Konfiguration und ihren `AuthType`.
- `lambda:GetFunctionUrlConfig` – Zeigt die Details einer Funktions-URL an.
- `lambda:ListFunctionUrlConfigs` – Listet die Funktions-URL-Konfigurationen auf.
- `lambda:DeleteFunctionUrlConfig` – Löscht eine Funktions-URL.

Um anderen AWS Entitäten den Zugriff auf Funktions-URLs zu gewähren oder zu verweigern, nehmen Sie diese Aktionen in die IAM-Richtlinien auf. Die folgende Richtlinie gewährt der `example` Rolle in beispielsweise AWS-Konto `444455556666` Berechtigungen zur Aktualisierung der Funktions-URL für die Funktion **`my-function`** im Konto`123456789012`.

Example Kontoübergreifende Funktions-URL-Richtlinie

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"
        }
    ]
}
```

Bedingungsschlüssel

Verwenden Sie Bedingungskontexttasten URLs, um eine detaillierte Zugriffskontrolle für Ihre Funktion zu erhalten. Lambda unterstützt die folgenden Kontexttasten für Funktionen URLs:

- `lambda:FunctionUrlAuthType` – Definiert einen Aufzählungswert, der den Authentifizierungstyp beschreibt, den Ihre Funktions-URL verwendet. Der Wert kann entweder `AWS_IAM` oder `NONE` sein.
- `lambda:InvokedViaFunctionUrl` – Beschränkt die Aktion `lambda:InvokeFunction` auf Aufrufe, die über die Funktions-URL getätigt werden. Das stellt sicher, dass die Funktion nur über die Funktions-URL und nicht über andere Aufrufmethoden aufgerufen werden kann. Beispiele für ressourcenbasierte Richtlinien, die den Kontextschlüssel `lambda:InvokedViaFunctionUrl` verwenden, finden Sie in den Beispielen unter Verwenden des Auth-Typs `AWS_IAM` und Verwenden des Authentifizierungstyps `NONE`.

Sie können diese Kontextschlüssel in Richtlinien verwenden, die mit Ihrer Funktion verknüpft sind. Beispielsweise möchten Sie möglicherweise einschränken, wer Konfigurationsänderungen an Ihrer Funktion URLs vornehmen kann. Um alle `UpdateFunctionUrlConfig`-Anfragen an jede Funktion mit dem URL-Auth-Typ `NONE` zu verweigern, können Sie die folgende Richtlinie definieren:

Example Funktions-URL-Richtlinie mit expliziter Zugriffsverweigerung

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"
                }
            }
        }
    ]
}
```

Um der `example` Rolle AWS-Konto `444455556666` Berechtigungen zum Erstellen `CreateFunctionUrlConfig` und `UpdateFunctionUrlConfig` Anfordern von Funktionen mit URL-Authentifizierungstyp zu gewähren`AWS_IAM`, können Sie die folgende Richtlinie definieren:

Example Funktions-URL-Richtlinie mit expliziter Zulassung

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"
                }
            }
        }
    ]
}
```

Sie können diesen Bedingungsschlüssel auch in einer Service-Kontrollrichtlinie (SCP) verwenden. Wird verwendet SCPs , um Berechtigungen für eine gesamte Organisation in AWS Organizations zu verwalten. Verwenden Sie beispielsweise die folgende Dienststeuerungsrichtlinie, um Benutzern URLs das Erstellen oder Aktualisieren von Funktionen zu verbieten, die `AWS_IAM` etwas anderes als den Authentifizierungstyp verwenden:

Example Funktions-URL-SCP mit expliziter Zugriffsverweigerung

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"
                }
            }
        }
    ]
```

```
}
```

Aufrufen von Lambda-Funktions-URLs

Eine Funktions-URL ist ein dedizierter HTTPS-Endpunkt für Ihre Lambda-Funktion. Sie können eine Funktions-URL über die Lambda-Konsole oder die Lambda-API erstellen und konfigurieren.

> **Tip**
>
> Lambda bietet zwei Möglichkeiten, Ihre Funktion über einen HTTP-Endpunkt aufzurufen: Funktions-URLs und Amazon API Gateway. Wenn Sie sich nicht sicher sind, welche Methode für Ihren Anwendungsfall am besten geeignet ist, siehe the section called “Funktion im URLs Vergleich zu Amazon API Gateway”.

Wenn Sie eine Funktions-URL erstellen, generiert Lambda automatisch einen eindeutigen URL-Endpunkt für Sie. Sobald Sie eine Funktions-URL erstellt haben, ändert sich ihr URL-Endpunkt nie mehr. Funktions-URL-Endpunkte haben das folgende Format:

```
https://<url-id>.lambda-url.<region>.on.aws
```

> **Note**
>
> Funktions-URLs werden in den folgenden AWS-Regionen nicht unterstützt: Asien-Pazifik (Hyderabad) (`ap-south-2`), Asien-Pazifik (Melbourne) (`ap-southeast-4`), Asien-Pazifik (Malaysia) (`ap-southeast-5`), Asien-Pazifik (Neuseeland) (`ap-southeast-6`), Asien-Pazifik (Thailand) (`ap-southeast-7`), Asien-Pazifik (Taipeh) (`ap-east-2`), Kanada West (Calgary) (`ca-west-1`), Europa (Spanien) (`eu-south-2`), Europa (Zürich) (`eu-central-2`), Israel (Tel Aviv) (`il-central-1`) und Naher Osten (VAE) (`me-central-1`).

Funktions-URLs sind Dual-Stack-fähig und unterstützen IPv4 und IPv6. Nachdem Sie Ihre Funktions-URL konfiguriert haben, können Sie Ihre Funktion über ihren HTTP(S)-Endpunkt über einen Webbrowser, cURL, Postman oder einen beliebigen HTTP-Client aufrufen. Um eine Funktions-URL aufzurufen, müssen Sie `lambda:InvokeFunctionUrl`- und `lambda:InvokeFunction`-Berechtigungen haben. Weitere Informationen finden Sie unter Zugriffskontrolle.

Themen

- Grundlagen des Aufrufs von Funktions-URLs

- Anfordern und Beantworten von Nutzlasten

Grundlagen des Aufrufs von Funktions-URLs

Wenn Ihre Funktions-URL den Auth-Typ `AWS_IAM` hat, müssen Sie jede HTTP-Anfrage mit der AWS Signature Version 4 (SigV4) signieren. Tools wie Postman bieten integrierte Möglichkeiten, Ihre Anforderungen mit SigV4 zu signieren.

Wenn Sie kein Tool verwenden, um HTTP-Anfragen an Ihre Funktions-URL zu signieren, müssen Sie jede Anfrage manuell mit SigV4 signieren. Wenn Ihre Funktions-URL eine Anfrage erhält, berechnet Lambda auch die SigV4-Signatur. Nur wenn die Signaturen übereinstimmen, verarbeitet Lambda die Anfrage. Anweisungen zum manuellen Signieren Ihrer Anforderungen mit SigV4 finden Sie unter Signieren von AWS-Anforderungen mit Signature Version 4 im Allgemeine Amazon Web Services-Referenz.

Wenn Ihre Funktions-URL den Authentifizierungstyp `NONE` verwendet, müssen Sie Ihre Anfragen nicht mit SigV4 signieren. Sie können Ihre Funktion per Webbrowser, cURL, Postman oder einem beliebigen HTTP-Client aufrufen.

Um einfache `GET`-Anfragen an Ihre Funktion zu testen, verwenden Sie einen Webbrowser. Wenn Ihre Funktions-URL zum Beispiel `https://abcdefg.lambda-url.us-east-1.on.aws` ist und sie einen Zeichenfolgeparameter `message` aufnimmt, könnte Ihre Anforderungs-URL folgendermaßen aussehen:

```
https://abcdefg.lambda-url.us-east-1.on.aws/?message=HelloWorld
```

Um andere HTTP-Anfragen zu testen, z. B. eine `POST`-Anfrage, können Sie ein Tool wie cURL verwenden. Wenn Sie z. B. einige JSON-Daten in eine `POST`-Anfrage an Ihre Funktions-URL aufnehmen möchten, können Sie den folgenden cURL-Befehl verwenden:

```
curl -v 'https://abcdefg.lambda-url.us-east-1.on.aws/?message=HelloWorld' \
-H 'content-type: application/json' \
-d '{ "example": "test" }'
```

Anfordern und Beantworten von Nutzlasten

Wenn ein Client Ihre Funktions-URL aufruft, ordnet Lambda die Anforderung einem Ereignisobjekt zu, bevor es sie an Ihre Funktion übergibt. Die Antwort Ihrer Funktion wird dann einer HTTP-Antwort zugeordnet, die Lambda über die Funktions-URL an den Client zurücksendet.

Die Formate für Anforderungs- und Antwortereignisse folgen dem gleichen Schema wie die Amazon-API-Gateway-Nutzlastformatversion 2.0.

Anforderungsnutzlastformat

Eine Anforderungsnutzlast hat die folgende Struktur:

```
{
  "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
}
```

Parameter	Beschreibung	Beispiel
`version`	Die Nutzlastformatversion für dieses Ereignis. Lambda-Funktions-URLs unterstützen derzeit die Nutzlastformatversion 2.0.	`2.0`
`routeKey`	Funktions-URLs verwenden diesen Parameter nicht. Lambda setzt dies auf `$default` als Platzhalter.	`$default`
`rawPath`	Der Anforderungspfad. Wenn die Anforderungs-URL beispielsweise `https://{url-id}.lambda-url.{region}.on.aws/example/test/demo` ist, dann ist der Rohpfadwert `/example/test/demo`.	`/example/test/demo`

Parameter	Beschreibung	Beispiel
`rawQueryString`	Die Rohzeichenfolge, die die Abfragezeichenfolge-Parameter der Anforderung enthält. Zu den unterstützten Zeichen gehören `a-z`, `A-Z`, `0-9`, `.`, `_`, `-`, `%`, `&`, `=`, und `+`.	`"?parameter1=value1&parameter2=value2"`
`cookies`	Ein Array, das alle Cookies enthält, die im Rahmen der Anforderung gesendet wurden.	`["Cookie_1=Value_1", "Cookie_2=Value_2"]`
`headers`	Die Liste der Anforderungs-Header, die als Schlüssel-Wert-Paare dargestellt werden.	`{"header1": "value1", "header2": "value2"}`
`queryStringParameters`	Die Abfrageparameter für die Anforderung. Wenn die Anforderungs-URL beispielsweise `https://{url-id}.lambda-url.{region}.on.aws/example?name=Jane` ist, dann ist der `queryStringParameters` -Wert ein JSON-Objekt mit einem Schlüssel von `name` und einem Wert von `Jane`.	`{"name": "Jane"}`

Parameter	Beschreibung	Beispiel
`requestContext`	Ein Objekt, das zusätzliche Informationen über die Anforderung enthält, z. B. `requestId` , den Zeitpunkt der Anfrage und die Identität des Anrufers, falls über AWS Identity and Access Management (IAM) autorisiert.	
`requestContext.accountId`	Die AWS-Konto-ID des Funktionsbesitzers.	`"123456789012"`
`requestContext.apiId`	Die ID der Funktions-URL.	`"33anwqw8fj"`
`requestContext.authentication`	Funktions-URLs verwenden diesen Parameter nicht. Lambda setzt dies auf `null`.	`null`
`requestContext.authorizer`	Ein Objekt, das Informationen über die Aufruferidentität enthält, wenn die Funktions-URL den Authentifizierungstyp `AWS_IAM` verwendet. Ansonsten setzt Lambda dies auf `null`.	
`requestContext.authorizer.iam.accessKey`	Der Zugriffsschlüssel der Anruferidentität.	`"AKIAIOSFODNN7EXAMPLE"`
`requestContext.authorizer.iam.accountId`	Die AWS-Konto-Identitäts-ID des Anrufers.	`"111122223333"`

Parameter	Beschreibung	Beispiel
`requestContext.authorizer.iam.callerId`	Die ID (Benutzer-ID) des Aufrufers.	`"AIDACKCEVSQ6C2EXAMPLE"`
`requestContext.authorizer.iam.cognitoIdentity`	Funktions-URLs verwenden diesen Parameter nicht. Lambda setzt dies auf `null` oder schließt dies aus der JSON aus.	`null`
`requestContext.authorizer.iam.principalOrgId`	Die Prinzipal-Organisations-ID, die mit der Anruferidentität verknüpft ist.	`"AIDACKCEVSQORGEXAMPLE"`
`requestContext.authorizer.iam.userArn`	Der Benutzer-ARN (Amazon-Ressourcenname) der Anruferidentität.	`"arn:aws:iam::111122223333:user/example-user"`
`requestContext.authorizer.iam.userId`	Die Benutzer-ID des Anrufers.	`"AIDACOSFODNN7EXAMPLE2"`
`requestContext.domainName`	Der Domain-Name der Funktions-URL.	`"<url-id>.lambda-url.us-west-2.on.aws"`
`requestContext.domainPrefix`	Das Domain-Präfix der Funktions-URL.	`"<url-id>"`
`requestContext.http`	Ein Objekt, das Details zur HTTP-Anforderung enthält.	

Parameter	Beschreibung	Beispiel
`requestContext.http.method`	Die in der Anforderung verwendete HTTP-Methode. Gültige Werte sind unter anderem `GET`, `POST`, `PUT`, `HEAD`, `OPTIONS`, `PATCH` und `DELETE`.	`GET`
`requestContext.http.path`	Der Anforderungspfad. Zum Beispiel, wenn die Anforderungs-URL `https://{url-id}.lambda-url.{region}.on.aws/example/test/demo` ist, dann ist der Pfadwert `/example/test/demo` .	`/example/test/demo`
`requestContext.http.protocol`	Das Protokoll der Anforderung.	`HTTP/1.1`
`requestContext.http.sourceIp`	Die Quell-IP-Adresse der TCP-Verbindung, von der die Anforderung stammt.	`123.123.123.123`
`requestContext.http.userAgent`	Der Headerwert der Benutzer-Agent-Anforderung.	`Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) Gecko/20100101 Firefox/42.0`
`requestContext.requestId`	Die ID der aktuellen Aufrufanforderung. Sie können diese ID verwenden, um Aufrufprotokolle zu verfolgen, die sich auf Ihre Funktion beziehen.	`e1506fd5-9e7b-434f-bd42-4f8fa224b599`

Parameter	Beschreibung	Beispiel
`requestContext.routeKey`	Funktions-URLs verwenden diesen Parameter nicht. Lambda setzt dies auf `$default` als Platzhalter.	`$default`
`requestContext.stage`	Funktions-URLs verwenden diesen Parameter nicht. Lambda setzt dies auf `$default` als Platzhalter.	`$default`
`requestContext.time`	Der Zeitstempel der Anfrage.	`"07/Sep/2021:22:50:22 +0000"`
`requestContext.timeEpoch`	Die Uhrzeit der Anfrage in Unix-Epochen-Zeit.	`"1631055022677"`
`body`	Der Text der Anforderung. Wenn der Inhaltstyp der Anforderung binär ist, ist der Text base64-kodiert.	`{"key1": "value1", "key2": "value2"}`
`pathParameters`	Funktions-URLs verwenden diesen Parameter nicht. Lambda setzt dies auf `null` oder schließt dies aus der JSON aus.	`null`
`isBase64Encoded`	`TRUE`, wenn der Text eine binäre Nutzlast hat und base64-kodiert ist. Ansonsten `FALSE`.	`FALSE`

Parameter	Beschreibung	Beispiel
`stageVariables`	Funktions-URLs verwenden diesen Parameter nicht. Lambda setzt dies auf `null` oder schließt dies aus der JSON aus.	`null`

Antwortnutzlastformat

Wenn Ihre Funktion eine Antwort zurückgibt, analysiert Lambda die Antwort und konvertiert sie in eine HTTP-Antwort. Funktionsantwortnutzlasten haben das folgende Format:

```
{
   "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 leitet das Antwortformat für Sie ab. Wenn Ihre Funktion gültiges JSON zurückgibt und keinen `statusCode` zurückgibt, geht Lambda von Folgendem aus:

- `statusCode` is `200`.

> **Note**
>
> Die gültigen `statusCode` liegen im Bereich von 100 bis 599.

- `content-type` is `application/json`.
- `body` ist die Antwort der Funktion.
- `isBase64Encoded` is `false`.

Die folgenden Beispiele zeigen, wie die Ausgabe Ihrer Lambda-Funktion der Antwortnutzlast und die Antwortnutzlast der endgültigen HTTP-Antwort zugeordnet wird. Wenn der Client Ihre Funktions-URL aufruft, wird die HTTP-Antwort angezeigt.

Beispielausgabe für eine Zeichenfolgeantwort

Lambda-Funktionsausgabe	Interpretierte Antwortausgabe	HTTP-Antwort (was der Client sieht)
`"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!"`

Beispielausgabe für eine JSON-Antwort

Lambda-Funktionsausgabe	Interpretierte Antwortausgabe	HTTP-Antwort (was der Client sieht)
`{ "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!" }`

Lambda-Funktionsausgabe	Interpretierte Antwortausgabe	HTTP-Antwort (was der Client sieht)
	`}`	

Beispielausgabe für eine benutzerdefinierte Antwort

Lambda-Funktionsausgabe	Interpretierte Antwortausgabe	HTTP-Antwort (was der Client sieht)
`{ "statusCode": 201, "headers": { "Content-Type": "application/json", "My-Custom-Header": "Custom Value" }, "body": JSON.stringify({ "message": "Hello, world!" }), "isBase64Encoded": false }`	`{ "statusCode": 201, "headers": { "Content-Type": "application/json", "My-Custom-Header": "Custom Value" }, "body": JSON.stringify({ "message": "Hello, world!" }), "isBase64Encoded": false }`	`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!" }`

Cookies

Um Cookies von Ihrer Funktion zurückzugeben, fügen Sie `set-cookie`-Header nicht manuell hinzu. Fügen Sie stattdessen die Cookies in Ihr Antwortnutzlastobjekt ein. Lambda interpretiert dies automatisch und fügt sie als `set-cookie`-Header in Ihre HTTP-Antwort ein, wie im folgenden Beispiel gezeigt.

Lambda-Funktionsausgabe	HTTP-Antwort (was der Client sieht)
`{`	`HTTP/2 201`

Lambda-Funktionsausgabe	HTTP-Antwort (was der Client sieht)

```
   "statusCode": 201,
    "headers": {
        "Content-Type": "application/
json",
        "My-Custom-Header": "Custom
 Value"
    },
    "body": JSON.stringify({
        "message": "Hello, world!"
    }),
    "cookies": [
        "Cookie_1=Value1; Expires=21
 Oct 2021 07:48 GMT",
        "Cookie_2=Value2; Max-Age=7
8000"
    ],
    "isBase64Encoded": false
}
```

```
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

{
  "message": "Hello, world!"
}
```

Überwachen der Lambda-Funktions-URLs

Sie können AWS CloudTrail und Amazon CloudWatch zur Überwachung Ihrer Funktions-URLs verwenden.

Themen

- Überwachen von Funktions-URLs mit CloudTrail
- CloudWatch-Metriken für Funktions-URLs

Überwachen von Funktions-URLs mit CloudTrail

Bei Funktions-URLs unterstützt Lambda automatisch die Protokollierung der folgenden API-Operationen als Ereignisse in CloudTrail-Protokolldateien:

- CreateFunctionUrlConfig
- UpdateFunctionUrlConfig
- DeleteFunctionUrlConfig
- GetFunctionUrlConfig
- ListFunctionUrlConfigs

Jeder Protokolleintrag enthält Informationen über die Anruferidentität, wann die Anfrage gestellt wurde, und andere Details. Sie können alle Ereignisse innerhalb der letzten 90 Tage sehen, indem Sie Ihren CloudTrail-Ereignisverlauf ansehen. Um Datensätze nach 90 Tagen beizubehalten, können Sie einen Trail erstellen.

Standardmäßig protokolliert CloudTrail `InvokeFunctionUrl`-Anfragen, die für Datenereignisse gehalten werden, nicht. Sie können die Datenereignisprotokollierung jedoch in CloudTrail aktivieren. Weitere Informationen finden Sie unter Protokollieren von Datenereignissen für Trails im AWS CloudTrail-Benutzerhandbuch.

CloudWatch-Metriken für Funktions-URLs

Lambda sendet aggregierte Metriken über Funktions-URL-Anfragen an CloudWatch. Mit diesen Metriken können Sie Ihre Funktions-URLs überwachen, Dashboards erstellen und Alarme in der CloudWatch-Konsole konfigurieren.

Funktions-URLs unterstützen die folgenden Aufrufmetriken. Wir empfehlen, diese Metriken mit der Sum-Statistik anzuzeigen.

- `UrlRequestCount` – Die Anzahl der Anforderungen an diese Funktions-URL.
- `Url4xxCount` – Die Anzahl der Anforderungen, bei denen ein 4XX-HTTP-Statuscode zurückgegeben wurde. Codes der 4XX-Serie weisen auf clientseitige Fehler hin, z. B. schlechte Anforderungen.
- `Url5xxCount` – Die Anzahl der Anforderungen, bei denen ein 5XX-HTTP-Statuscode zurückgegeben wurde. Codes der 5XX-Serie weisen auf serverseitige Fehler wie Funktionsfehler und Timeouts hin.

Funktions-URLs unterstützen auch die folgende Leistungsmetrik. Wir empfehlen, diese Metriken mit der `Average`- oder `Max`-Statistik anzuzeigen.

- `UrlRequestLatency` – Die Zeit zwischen dem Erhalt der Funktions-URL eine Anforderung und wann die Funktions-URL eine Antwort zurückgibt.

Jede dieser Aufrufs- und Leistungsmetriken unterstützt die folgenden Dimensionen:

- `FunctionName` – Zeigt Aggregatmetriken für Funktions-URLs an, die der `$LATEST` unveröffentlichten Version oder einem der Aliase der Funktion zugeordnet sind. Beispiel, `hello-world-function`.
- `Resource` – Zeigt Metriken für eine bestimmte Funktions-URL an. Dies wird durch einen Funktionsnamen zusammen mit der `$LATEST` unveröffentlichten Version oder einem der Aliase der Funktion definiert. Beispiel, `hello-world-function:$LATEST`.
- `ExecutedVersion` – Zeigt Metriken für eine bestimmte Funktions-URL basierend auf der ausgeführten Version an. Sie können diese Dimension hauptsächlich verwenden, um die Funktions-URL zu verfolgen, die der `$LATEST`unveröffentlichten Version zugeordnet ist.

Wählen Sie eine Methode, um Ihre Lambda-Funktion über eine HTTP-Anfrage aufzurufen

Viele häufige Anwendungsfälle für Lambda beinhalten den Aufruf Ihrer Funktion über eine HTTP-Anfrage. Beispielsweise möchten Sie möglicherweise, dass eine Webanwendung Ihre Funktion über eine Browseranforderung aufruft. Lambda-Funktionen können auch verwendet werden, um

vollständige REST zu erstellen APIs, Benutzerinteraktionen von mobilen Apps aus abzuwickeln, Daten von externen Diensten über HTTP-Aufrufe zu verarbeiten oder benutzerdefinierte Webhooks zu erstellen.

In den folgenden Abschnitten wird erläutert, welche Möglichkeiten Sie haben, Lambda über HTTP aufzurufen und es werden Informationen bereitgestellt, die Ihnen helfen, die richtige Entscheidung für Ihren speziellen Anwendungsfall zu treffen.

Welche Optionen stehen Ihnen bei der Auswahl einer HTTP-Aufrufmethode zur Verfügung?

Lambda bietet zwei Hauptmethoden zum Aufrufen einer Funktion mithilfe einer HTTP-Anfrage: Funktion URLs und API Gateway. Die wichtigsten Unterschiede zwischen diesen beiden Optionen sind die folgenden:

- Die Lambda-Funktion URLs bietet einen einfachen, direkten HTTP-Endpunkt für eine Lambda-Funktion. Sie sind auf Einfachheit und Kosteneffizienz optimiert und bieten den schnellsten Weg, um eine Lambda-Funktion über HTTP bereitzustellen.
- API Gateway ist ein fortschrittlicherer Dienst zum Erstellen von Funktionen mit vollem Funktionsumfang APIs. API Gateway ist für den Aufbau und die Verwaltung von Produktionen APIs in großem Maßstab optimiert und bietet umfassende Tools für Sicherheit, Überwachung und Verkehrsmanagement.

Empfehlungen, falls Sie Ihre Anforderungen bereits kennen

Wenn Sie sich über Ihre Anforderungen bereits im Klaren sind, finden Sie hier unsere grundlegenden Empfehlungen:

Wir empfehlen Funktionen URLs für einfache Anwendungen oder für die Prototypenentwicklung, bei denen Sie nur grundlegende Authentifizierungsmethoden und request/response deren Handhabung benötigen und bei denen Sie Kosten und Komplexität auf ein Minimum reduzieren möchten.

API Gateway ist eine bessere Wahl für Produktionsanwendungen in großem Maßstab oder für Fälle, in denen Sie erweiterte Funktionen wie OpenAPI-Description, eine Auswahl an Authentifizierungsoptionen, benutzerdefinierte Domainnamen oder umfangreiche request/response Funktionen wie Drosselung, Caching und Transformation benötigen. request/response

Was ist bei der Auswahl einer Methode zum Aufrufen Ihrer Lambda-Funktion zu beachten?

Bei der Auswahl zwischen Funktion URLs und API Gateway müssen Sie die folgenden Faktoren berücksichtigen:

- Ihre Authentifizierungsanforderungen, z. B. ob Sie Amazon Cognito für die Benutzerauthentifizierung benötigen OAuth
- Ihre Skalierungsanforderungen und die Komplexität der API, die Sie implementieren möchten
- Ob Sie erweiterte Funktionen wie die Validierung und Formatierung von Anfragen benötigen request/response
- Ihre Überwachungsanforderungen
- Was sind Ihre Ziele?

Wenn Sie diese Faktoren verstehen, können Sie die Option auswählen, die Ihren Anforderungen an Sicherheit, Komplexität und Kosten am besten gerecht wird.

Die folgenden Informationen fassen die wichtigsten Unterschiede zwischen den beiden Optionen zusammen.

Authentifizierung

- Die Funktion URLs bietet grundlegende Authentifizierungsoptionen über AWS Identity and Access Management (IAM). Sie können Ihre Endpunkte so konfigurieren, dass sie entweder öffentlich sind (keine Authentifizierung) oder dass sie eine IAM-Authentifizierung erfordern. Mit der IAM-Authentifizierung können Sie AWS Standardanmeldedaten oder IAM-Rollen verwenden, um den Zugriff zu steuern. Dieser Ansatz ist zwar einfach einzurichten, bietet aber im Vergleich zu anderen Authentifizierungsmethoden nur begrenzte Optionen.
- API Gateway bietet Zugriff auf eine umfassendere Palette von Authentifizierungsoptionen. Neben der IAM-Authentifizierung können Sie Lambda-Autorisierer (benutzerdefinierte Authentifizierungslogik), Amazon Cognito Cognito-Benutzerpools und .0-Flows verwenden. OAuth2 Diese Flexibilität ermöglicht es Ihnen, komplexe Authentifizierungsschemata zu implementieren, darunter Authentifizierungsanbieter von Drittanbietern, tokenbasierte Authentifizierung und Multi-Faktor-Authentifizierung.

Anfrage-/Antwort-Anforderungen

- Die Funktion URLs bietet eine grundlegende Behandlung von HTTP-Anfragen und -Antworten. Sie unterstützen Standard-HTTP-Methoden und bieten integrierte CORS-Unterstützung (Cross-Origin Resource Sharing). Sie können zwar JSON-Nutzdaten und Abfrageparameter auf natürliche Weise verarbeiten, bieten jedoch keine Funktionen zur Transformation oder Validierung von Anfragen. Die Bearbeitung von Antworten ist ähnlich einfach – der Client erhält die Antwort von Ihrer Lambda-Funktion genau so, wie Lambda sie zurückgibt.
- API Gateway bietet ausgefeilte Funktionen zur Bearbeitung von Anfragen und Antworten. Sie können Anforderungsvalidatoren definieren, Anfragen und Antworten mithilfe von Zuordnungsvorlagen transformieren, request/response Header einrichten und das Zwischenspeichern von Antworten implementieren. API Gateway unterstützt auch binäre Nutzdaten und benutzerdefinierte Domainnamen und kann Antworten ändern, bevor sie den Client erreichen. Sie können Modelle für die request/response Validierung und Transformation mithilfe des JSON-Schemas einrichten.

Skalierung

- Die Funktion URLs skaliert direkt mit den Parallelitätsgrenzen Ihrer Lambda-Funktion und bewältigt Verkehrsspitzen, indem Sie Ihre Funktion bis zum konfigurierten Maximallimit skalieren. Sobald dieses Limit erreicht ist, reagiert Lambda auf weitere Anfragen mit HTTP 429-Antworten. Es gibt keinen integrierten Warteschlangenmechanismus, sodass der Umgang mit der Skalierung vollständig von der Konfiguration Ihrer Lambda-Funktion abhängt. Standardmäßig haben Lambda-Funktionen ein Limit von 1.000 gleichzeitigen Ausführungen pro. AWS-Region
- API Gateway bietet zusätzlich zur Lambda-eigenen Skalierung zusätzliche Skalierungsfunktionen. Es umfasst integrierte Funktionen zur Warteschleife und Drosselung von Anfragen, sodass Sie Datenverkehrsspitzen besser bewältigen können. API Gateway kann standardmäßig bis zu 10.000 Anfragen pro Sekunde pro Region verarbeiten, mit einer Burst-Kapazität von 5.000 Anfragen pro Sekunde. Es bietet auch Tools zum Drosseln von Anfragen auf verschiedenen Ebenen (API, Phase oder Methode), um Ihr Backend zu schützen.

Überwachen

- Function URLs bietet eine grundlegende Überwachung anhand von CloudWatch Amazon-Metriken, einschließlich Anzahl der Anfragen, Latenz und Fehlerraten. Sie erhalten Zugriff auf standardmäßige Lambda-Metriken und -Protokolle, die die Rohanfragen anzeigen, die in Ihre

Funktion eingehen. Dies bietet zwar wichtige betriebliche Transparenz, die Metriken konzentrieren sich jedoch hauptsächlich auf die Funktionsausführung.

- API Gateway bietet umfassende Überwachungsfunktionen, einschließlich detaillierter Metriken, Protokollierungs- und Verfolgungsoptionen. Sie können API-Aufrufe, Latenz, Fehlerraten und hit/miss Cache-Raten über überwachen CloudWatch. API Gateway lässt sich auch AWS X-Ray für verteiltes Tracing integrieren und bietet anpassbare Protokollierungsformate.

Cost (Kosten)

- Die Funktionen URLs folgen dem Standard-Lambda-Preismodell — Sie zahlen nur für Funktionsaufrufe und Rechenzeit. Für den URL-Endpunkt selbst fallen keine zusätzlichen Gebühren an. Dies macht es zu einer kostengünstigen Wahl für einfache Anwendungen APIs oder Anwendungen mit geringem Datenverkehr, wenn Sie die zusätzlichen Funktionen von API Gateway nicht benötigen.
- API Gateway bietet ein kostenloses Kontingent, das eine Million API-Aufrufe für REST APIs und eine Million API-Aufrufe für HTTP umfasst APIs. Danach berechnet API Gateway Gebühren für API-Aufrufe, Datenübertragung und Caching (falls aktiviert). Weitere Informationen zu den Kosten für Ihren eigenen Anwendungsfall finden Sie auf der Seite mit den Preisen für API Gateway.

Weitere Funktionen

- URLsDie Funktionen sind auf Einfachheit und direkte Lambda-Integration ausgelegt. Sie unterstützen sowohl HTTP- als auch HTTPS-Endpunkte, bieten integrierte CORS-Unterstützung und bieten Dual-Stack- (IPv4 und) Endpunkte. IPv6 Ihnen fehlen zwar erweiterte Features, sie zeichnen sich jedoch in Szenarien aus, in denen Sie Lambda-Funktionen schnell und unkompliziert über HTTP verfügbar machen müssen.
- API Gateway umfasst zahlreiche zusätzliche Funktionen wie API-Versionierung, Stagemanagement, API-Schlüssel für Nutzungspläne, API-Dokumentation über Swagger/OpenAPI WebSocket APIs, private Funktionen APIs innerhalb einer VPC und WAF-Integration für zusätzliche Sicherheit. Es unterstützt auch kanarische Implementierungen, Scheinintegrationen zum Testen und die Integration mit anderen Systemen AWS-Services als Lambda.

Wählen Sie eine Methode, um Ihre Lambda-Funktion aufzurufen

Nachdem Sie sich mit den Kriterien für die Auswahl zwischen Lambda-Funktion URLs und API Gateway und den wichtigsten Unterschieden zwischen ihnen vertraut gemacht haben, können Sie

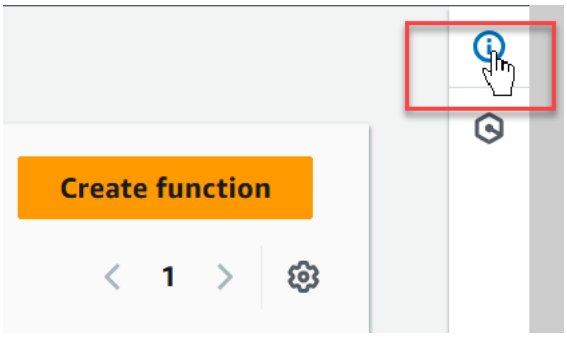
die Option auswählen, die Ihren Anforderungen am besten entspricht, und die folgenden Ressourcen verwenden, um Ihnen den Einstieg in die Verwendung zu erleichtern.

Function URLs

Nutzen Sie die folgenden Ressourcen, um URLs mit Function zu beginnen

- Folgen Sie dem Tutorial Erstellen einer Lambda-Funktion mit einer Funktions-URL
- Weitere Informationen zur Funktion finden Sie URLs im the section called “Funktion URLs” Kapitel dieses Handbuchs
- Probieren Sie das in der Konsole enthaltene Tutorial Erstellen Sie eine einfache Web-App aus, indem Sie wie folgt vorgehen:

1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.
2. Öffnen Sie das Hilfe-Panel, indem Sie das Symbol in der oberen rechten Ecke des Bildschirms wählen.



3. Wählen Sie Tutorials aus.
4. Wählen Sie unter Einfache Web-App erstellen die Option Tutorial starten aus.

API Gateway

Beginnen Sie mit Lambda und API Gateway mit den folgenden Ressourcen

- Folgen Sie dem Tutorial Verwenden von Lambda mit API Gateway, um eine REST-API zu erstellen, die in eine Backend-Lambda-Funktion integriert ist.
- In den folgenden Abschnitten des Amazon API Gateway Developer Guide erfahren Sie mehr über die verschiedenen Arten von APIs, die von API Gateway angeboten werden:
  - API-Gateway-REST APIs
  - API Gateway HTTP APIs

  - API Gateway WebSocket APIs
- Probieren Sie eines oder mehrere der Beispiele im Abschnitt Tutorials und Workshops im Amazon API Gateway Developer Guide aus.

Tutorial: Erstellen eines Webhook-Endpunkts mithilfe einer Lambda-Funktions-URL

In diesem Tutorial erstellen Sie eine Lambda-Funktions-URL, um einen Webhook-Endpunkt zu implementieren. Ein Webhook ist eine einfache, ereignisgesteuerte Kommunikation, die automatisch Daten zwischen Anwendungen über HTTP sendet. Sie können Webhooks verwenden, um sofortige Benachrichtigungen über Ereignisse in einem anderen System zu erhalten, z. B. wenn sich ein neuer Kunde auf einer Website registriert, eine Zahlung verarbeitet wird oder eine Datei hochgeladen wird.

Mit Lambda können Webhooks entweder mithilfe der Lambda-Funktion URLs oder des API Gateway implementiert werden. Function ist eine gute Wahl für einfache Webhooks, für die keine Funktionen wie erweiterte Autorisierung oder Anforderungsvalidierung erforderlich URLs sind.

> **Tip**
>
> Wenn Sie sich nicht sicher sind, welche Lösung für Ihren speziellen Anwendungsfall am besten geeignet ist, lesen Sie bitte the section called “Funktion im URLs Vergleich zu Amazon API Gateway”.

Voraussetzungen

Um dieses Tutorial durchzuführen, muss entweder Python (Version 3.8 oder höher) oder Node.js (Version 18 oder höher) auf Ihrem lokalen Rechner installiert sein.

Um den Endpunkt mit einer HTTP-Anfrage zu testen, verwendet das Tutorial curl, ein Befehlszeilentool, mit dem Sie Daten über verschiedene Netzwerkprotokolle übertragen können. In der curl-Dokumentation erfahren Sie, wie Sie das Tool installieren, falls Sie das noch nicht getan haben.

So erstellen Sie die Lambda-Funktion:

Erstellen Sie zunächst die Lambda-Funktion, die ausgeführt wird, wenn eine HTTP-Anfrage an Ihren Webhook-Endpunkt gesendet wird. In diesem Beispiel sendet die sendende Anwendung bei jeder Zahlung eine Aktualisierung und gibt im Hauptteil der HTTP-Anfrage an, ob die Zahlung erfolgreich war. Die Lambda-Funktion analysiert die Anfrage und ergreift entsprechend dem Status der Zahlung Maßnahmen. In diesem Beispiel gibt der Code lediglich die Bestellnummer für die Zahlung aus, aber in einer realen Anwendung würden Sie die Bestellung möglicherweise einer Datenbank hinzufügen oder eine Benachrichtigung senden.

Die Funktion implementiert auch die gängigste Authentifizierungsmethode für Webhooks, die hashbasierte Nachrichtenauthentifizierung (HMAC). Bei dieser Methode teilen sich sowohl die sendende als auch die empfangende Anwendung einen geheimen Schlüssel. Die sendende Anwendung verwendet einen Hash-Algorithmus, um anhand dieses Schlüssels und des Nachrichteninhalts eine eindeutige Signatur zu generieren, und fügt die Signatur als HTTP-Header in die Webhook-Anfrage ein. Die empfangende Anwendung wiederholt diesen Schritt, generiert die Signatur unter Verwendung des geheimen Schlüssels und vergleicht den resultierenden Wert mit der im Anfrage-Header gesendeten Signatur. Stimmt das Ergebnis überein, wird die Anfrage als legitim angesehen.

Erstellen Sie die Funktion mithilfe der Lambda-Konsole mit der Python- oder Node.js-Laufzeit.

Python

So erstellen Sie die Lambda-Funktion:

1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.
2. Erstellen Sie eine einfache „Hello World“-Funktion:
   a. Wählen Sie Funktion erstellen.
   b. Wählen Sie Verfassen von Grund auf aus.
   c. Geben Sie für Function name (Funktionsname) **`myLambdaWebhook`** ein.
   d. Wählen Sie für Runtime python3.14 aus.
   e. Wählen Sie Funktion erstellen.
3. Ersetzen Sie im Bereich Codequelle den vorhandenen Code durch Folgendes:

```
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. Wählen Sie im Abschnitt BEREITSTELLEN die Option Bereitstellen aus, um den Code Ihrer Funktion zu aktualisieren.

Node.js

So erstellen Sie die Lambda-Funktion:

1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.
2. Erstellen Sie eine einfache „Hello World“-Funktion:
   a. Wählen Sie Funktion erstellen.
   b. Wählen Sie Verfassen von Grund auf aus.
   c. Geben Sie für Function name (Funktionsname) **`myLambdaWebhook`** ein.
   d. Wählen Sie für Runtime nodejs24.x aus.
   e. Wählen Sie Funktion erstellen.
3. Ersetzen Sie im Bereich Codequelle den vorhandenen Code durch Folgendes:

```
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. Wählen Sie im Abschnitt BEREITSTELLEN die Option Bereitstellen aus, um den Code Ihrer Funktion zu aktualisieren.

Erstellen des geheimen Schlüssels

Damit die Lambda-Funktion die Webhook-Anfrage authentifizieren kann, verwendet sie einen geheimen Schlüssel, den sie mit der aufrufenden Anwendung teilt. In diesem Beispiel wird der Schlüssel in einer Umgebungsvariablen gespeichert. Schließen Sie in einer Produktionsanwendung

keine sensiblen Informationen wie Passwörter in Ihren Funktionscode ein. Erstellen Sie stattdessen ein AWS Secrets Manager Geheimnis und verwenden Sie dann die Lambda-Erweiterung AWS Parameters and Secrets, um Ihre Anmeldeinformationen in Ihrer Lambda-Funktion abzurufen.

Erstellen und Speichern des geheimen Webhook-Schlüssels

1. Generieren Sie eine lange, zufällige Zeichenfolge mit einem kryptografisch sicheren Zufallszahlengenerator. Sie können die folgenden Codeausschnitte in Python oder Node.js verwenden, um einen 32-stelligen geheimen Schlüssel zu generieren und auszudrucken, oder Ihre eigene bevorzugte Methode verwenden.

   Python

   Example Code zum Generieren eines geheimen Schlüssels

   ```
   import secrets
   webhook_secret = secrets.token_urlsafe(32)
   print(webhook_secret)
   ```

   Node.js

   Example Code zum Generieren eines geheimen Schlüssels (ES-Modulformat)

   ```
   import crypto from 'crypto';
   let webhookSecret = crypto.randomBytes(32).toString('base64');
   console.log(webhookSecret)
   ```

2. Speichern Sie Ihre generierte Zeichenfolge als Umgebungsvariable für Ihre Funktion:

   a. Wählen Sie auf der Registerkarte Konfiguration für Ihre Funktion die Option Umgebungsvariablen aus.
   b. Wählen Sie Bearbeiten aus.
   c. Wählen Sie Umgebungsvariablen hinzufügen aus.
   d. Geben Sie für Schlüssel **`WEBHOOK_SECRET`** ein und dann für Wert den geheimen Schlüssel, den Sie im vorherigen Schritt generiert haben.
   e. Wählen Sie Speichern.

Sie werden diesen geheimen Schlüssel später im Tutorial erneut verwenden müssen, um Ihre Funktion zu testen. Notieren Sie ihn sich daher jetzt.

Erstellen des Funktions-URL-Endpunkts

Erstellen Sie einen Endpunkt für Ihren Webhook mithilfe einer Lambda-Funktions-URL. Da Sie den Authentifizierungstyp `NONE` verwenden, um einen Endpunkt mit öffentlichem Zugriff zu erstellen, kann jeder, der über die URL verfügt, Ihre Funktion aufrufen. Weitere Informationen zur Steuerung des Zugriffs auf Funktionen finden Sie URLs unter. the section called “Zugriffskontrolle” Wenn Sie erweiterte Authentifizierungsoptionen für Ihren Webhook benötigen, sollten Sie die Verwendung von API Gateway in Betracht ziehen.

Erstellen des Funktions-URL-Endpunkts

1. Wählen Sie auf der Registerkarte Konfiguration Ihrer Funktion die Option Funktions-URL aus.
2. Wählen Sie Funktions-URL erstellen.
3. Wählen Sie als Authentifizierungstyp die Option KEINEN aus.
4. Wählen Sie Speichern.

Der Endpunkt für die Funktions-URL, die Sie gerade erstellt haben, wird im Bereich Funktions-URL angezeigt. Kopieren Sie den Endpunkt, um ihn später im Tutorial zu verwenden.

Testen der Funktion in der Konsole

Bevor Sie eine HTTP-Anfrage verwenden, um Ihre Funktion über den URL-Endpunkt aufzurufen, testen Sie sie in der Konsole, um sicherzustellen, dass Ihr Code wie erwartet funktioniert.

Um die Funktion in der Konsole zu überprüfen, berechnen Sie zunächst eine Webhook-Signatur unter Verwendung des zuvor im Tutorial generierten geheimen Schlüssels mit den folgenden Test-JSON-Nutzdaten:

```
{
    "type": "payment.success",
    "orderId": "1234",
    "amount": "99.99"
}
```

Verwenden Sie eines der folgenden Python- oder Node.js-Codebeispiele, um die Webhook-Signatur mit Ihrem eigenen geheimen Schlüssel zu berechnen.

Python

Berechnen der Webhook-Signatur

1. Speichern Sie den folgenden Code als Datei mit dem Namen `calculate_signature.py`. Ersetzen Sie den geheimen Webhook-Schlüssel im Code durch Ihren eigenen Wert.

```
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. Berechnen Sie die Signatur, indem Sie den folgenden Befehl aus dem Verzeichnis ausführen, in dem Sie den Code gespeichert haben. Kopieren Sie die Signatur, die der Code ausgibt.

```
python calculate_signature.py
```

Node.js

Berechnen der Webhook-Signatur

1. Speichern Sie den folgenden Code als Datei mit dem Namen `calculate_signature.mjs`. Ersetzen Sie den geheimen Webhook-Schlüssel im Code durch Ihren eigenen Wert.

```
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. Berechnen Sie die Signatur, indem Sie den folgenden Befehl aus dem Verzeichnis ausführen, in dem Sie den Code gespeichert haben. Kopieren Sie die Signatur, die der Code ausgibt.

```
node calculate_signature.mjs
```

Sie können Ihren Funktionscode jetzt mit einer Test-HTTP-Anfrage in der Konsole testen.

Testen der Funktion in der Konsole

1. Wählen Sie die Registerkarte Code für Ihre Funktion aus.
2. Wählen Sie im Abschnitt TESTEREIGNISSE die Option Neues Testereignis erstellen aus.
3. Geben Sie als Event name (Ereignisname) die Zeichenfolge **`myEvent`** ein.
4. Ersetzen Sie das vorhandene JSON, indem Sie Folgendes kopieren und in den Bereich Ereignis-JSON einfügen. Ersetzen Sie die Webhook-Signatur durch den im vorherigen Schritt berechneten Wert.

```
{
  "headers": {
    "Content-Type": "application/json",
    "x-webhook-signature":
 "2d672e7a0423fab740fbc040e801d1241f2df32d2ffd8989617a599486553e2a"
  },
  "body": "{\"type\": \"payment.success\", \"orderId\": \"1234\", \"amount\":
 \"99.99\"}"
}
```

5. Wählen Sie Speichern.
6. Wählen Sie aufrufen aus.

   Die Ausgabe sollte folgendermaßen oder ähnlich aussehen:

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
```

Testen der Funktion mit einer HTTP-Anfrage

Testen Sie Ihren Webhook-Endpunkt mit dem Befehlszeilentool curl.

Testen der Funktion mit HTTP-Anfragen

1. Führen Sie in einem Terminal- oder Shell-Programm den folgenden curl-Befehl aus. Ersetzen Sie die URL durch den Wert für Ihren eigenen Funktions-URL-Endpunkt und ersetzen Sie die Webhook-Signatur durch die Signatur, die Sie mit Ihrem eigenen geheimen Schlüssel berechnet haben.

   ```
   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"}'
   ```

   Die Ausgabe sollte folgendermaßen aussehen:

   ```
   {"received": true}
   ```

2. Prüfen Sie die CloudWatch Protokolle Ihrer Funktion, um sicherzustellen, dass sie die Nutzlast korrekt analysiert hat. Gehen Sie dazu wie folgt vor:

   a. Öffnen Sie die Gruppenseite Logs in der CloudWatch Amazon-Konsole.

   b. Wählen Sie die Protokollgruppe Ihrer Funktion (`/aws/lambda/myLambdaWebhook`).

   c. Wählen Sie den neuesten Protokoll-Stream.

      In den Protokollen Ihrer Funktion sollte eine Ausgabe ähnlich der folgenden zu sehen sein:

      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. Überprüfen Sie, ob Ihr Code eine ungültige Signatur erkennt, indem Sie den folgenden curl-Befehl ausführen. Ersetzen Sie die URL durch Ihren eigenen Funktions-URL-Endpunkt.

```
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"}'
```

   Die Ausgabe sollte folgendermaßen aussehen:

```
{"error": "Invalid signature"}
```

Bereinigen Ihrer Ressourcen

Sie können jetzt die Ressourcen, die Sie für dieses Tutorial erstellt haben, löschen, es sei denn, Sie möchten sie behalten. Durch das Löschen von AWS Ressourcen, die Sie nicht mehr verwenden, vermeiden Sie unnötige Kosten für Ihre AWS-Konto.

So löschen Sie die Lambda-Funktion:

1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.
2. Wählen Sie die Funktion aus, die Sie erstellt haben.
3. Wählen Sie Aktionen, Löschen aus.
4. Geben Sie **`confirm`** in das Texteingabefeld ein und wählen Sie Delete (Löschen) aus.

Als Sie die Lambda-Funktion in der Konsole erstellt haben, hat Lambda auch eine Ausführungsrolle für Ihre Funktion erstellt.

So löschen Sie die Ausführungsrolle

1. Öffnen Sie die Seite Roles (Rollen) in der IAM-Konsole.
2. Wählen Sie die von Lambda erstellte Ausführungsrolle aus. Die Rolle weist das Benennungsformat `myLambdaWebhook-role-<random string>` auf.
3. Wählen Sie Löschen aus.
4. Geben Sie den Namen der Rolle in das Texteingabefeld ein und wählen Sie Delete (Löschen) aus.

Verstehen der Skalierung von Lambda-Funktionen

Nebenläufigkeit ist die Anzahl der laufenden Anfragen, die Ihre AWS Lambda-Funktion gleichzeitig bearbeitet. Für jede gleichzeitige Anfrage stellt Lambda eine separate Instance Ihrer Ausführungsumgebung bereit. Wenn Ihre Funktionen mehr Anfragen erhalten, sorgt Lambda automatisch für die Skalierung der Anzahl der Ausführungsumgebungen, bis Sie das Gleichzeitigkeitslimit für Ihr Konto erreichen. Standardmäßig stellt Lambda Ihrem Konto eine Gesamtnebenläufigkeitsgrenze von 1 000 gleichzeitigen Ausführungen für alle Funktionen in einer AWS-Region bereit. Um Ihre speziellen Kontoanforderungen zu unterstützen, können Sie eine Kontingenterhöhung anfordern und Gleichzeitigkeitskontrollen auf Funktionsebene konfigurieren, damit Ihre kritischen Funktionen nicht gedrosselt werden.

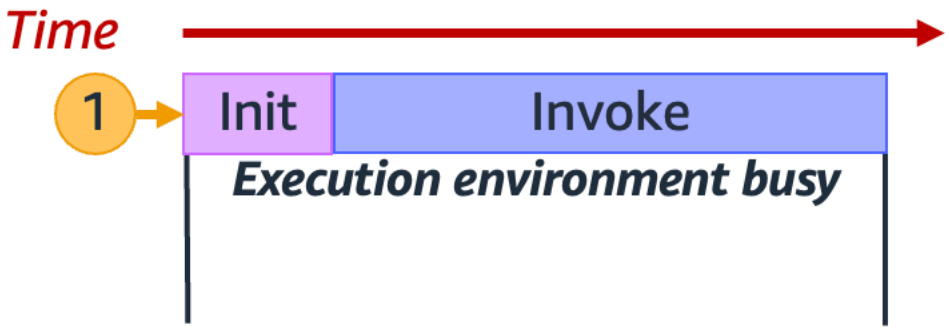
In diesem Thema werden Gleichzeitigkeitskonzepte und Funktionsskalierung in Lambda erläutert. Am Ende dieses Themas werden Sie in der Lage sein, die Gleichzeitigkeit zu berechnen, die beiden Hauptoptionen für die Gleichzeitigkeitskontrolle (reserviert und bereitgestellt) zu visualisieren, geeignete Einstellungen für die Gleichzeitigkeitskontrolle zu schätzen und Metriken zur weiteren Optimierung anzuzeigen.

Abschnitte

- Verstehen und Visualisieren der Gleichzeitigkeit
- Berechnen der Gleichzeitigkeit einer Funktion
- Verständnis der reservierten Gleichzeitigkeit und der bereitgestellten Gleichzeitigkeit
- Gleichzeitigkeit und Anforderungen pro Sekunde verstehen
- Gleichzeitigkeitskontingente
- Konfigurieren reservierter Gleichzeitigkeit für eine Funktion
- Konfigurieren von Provisioned Concurrency für eine Funktion
- Lambda-Skalierungsverhalten
- Überwachen der Gleichzeitigkeit

Verstehen und Visualisieren der Gleichzeitigkeit

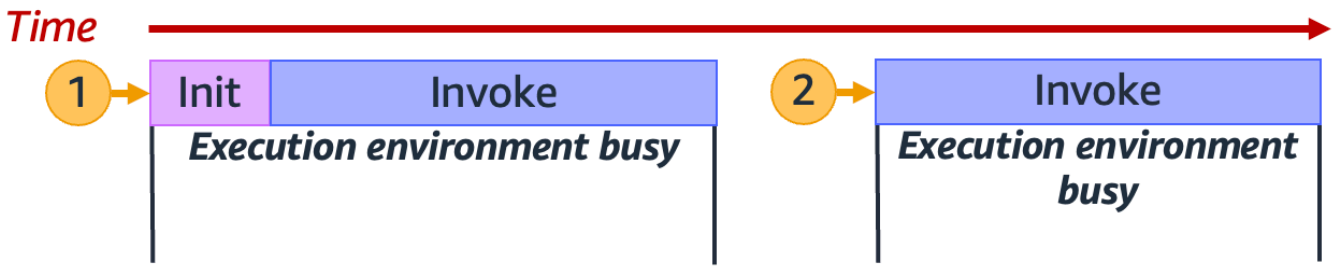
Lambda ruft Ihre Funktion in einer sicheren und isolierten Ausführungsumgebung auf. Um eine Anfrage zu bearbeiten, muss Lambda zuerst eine Ausführungsumgebung initialisieren (die Initialisierungsphase), bevor es zum Aufrufen Ihrer Funktion verwendet wird (die Aufrufphase):



> **Note**
>
> Die tatsächliche Initialisierungs- und Aufrufdauer kann von vielen Faktoren abhängen, z. B. von der von Ihnen ausgewählten Laufzeit und dem Lambda-Funktionscode. Das vorherige Diagramm soll nicht die genauen Proportionen der Initiierungs- und Aufrufphasendauer darstellen.

Im vorherigen Diagramm wird ein Rechteck verwendet, um eine einzelne Ausführungsumgebung darzustellen. Wenn Ihre Funktion ihre allererste Anfrage erhält (dargestellt durch den gelben Kreis mit der Bezeichnung 1), erstellt Lambda eine neue Ausführungsumgebung und führt den Code während der Initialisierungsphase außerhalb Ihres Haupthandlers aus. Anschließend führt Lambda den Haupthandlercode Ihrer Funktion während der Aufrufphase aus. Während des gesamten Prozesses ist diese Ausführungsumgebung ausgelastet und kann keine anderen Anfragen verarbeiten.

Wenn Lambda die Verarbeitung der ersten Anfrage abgeschlossen hat, kann diese Ausführungsumgebung weitere Anfragen für dieselbe Funktion verarbeiten. Für nachfolgende Anfragen muss Lambda die Umgebung nicht neu initialisieren.

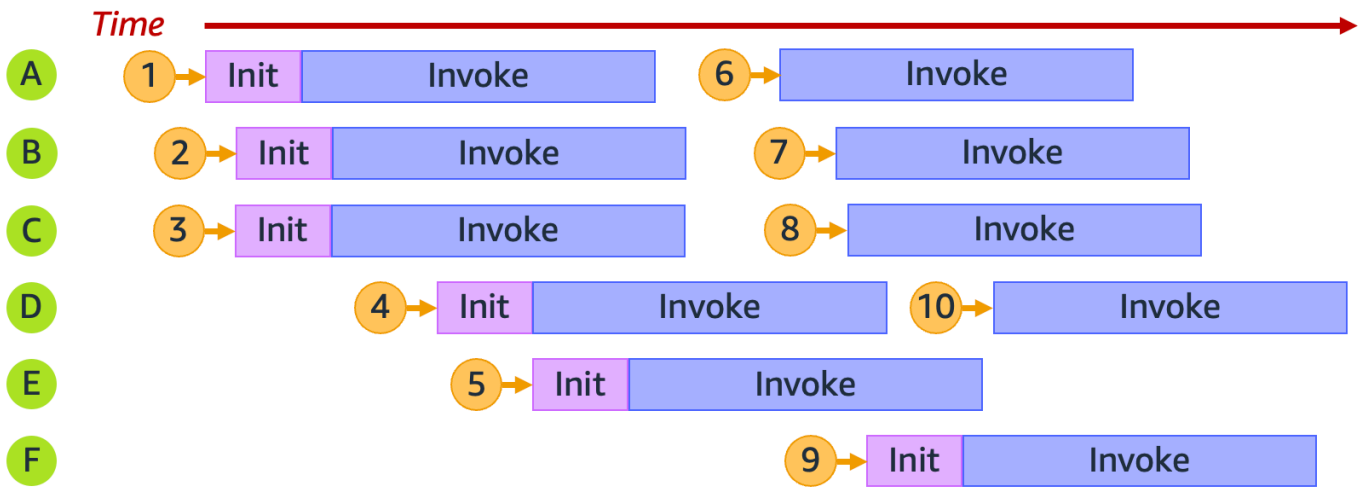


Im vorherigen Diagramm verwendet Lambda die Ausführungsumgebung erneut, um die zweite Anfrage zu verarbeiten (dargestellt durch den gelben Kreis mit der Bezeichnung 2).

Bisher haben wir uns nur auf eine einzige Instance Ihrer Ausführungsumgebung konzentriert (d. h. eine Gleichzeitigkeit von 1). In der Praxis muss Lambda möglicherweise mehrere Instances der Ausführungsumgebung parallel bereitstellen, um alle eingehenden Anfragen zu verarbeiten. Wenn Ihre Funktion eine neue Anfrage erhält, können zwei Dinge passieren:

- Wenn eine vorinitialisierte Instance der Ausführungsumgebung verfügbar ist, verwendet Lambda diese, um die Anfrage zu verarbeiten.
- Andernfalls erstellt Lambda eine neue Instance der Ausführungsumgebung, um die Anfrage zu verarbeiten.

Lassen Sie uns beispielsweise untersuchen, was passiert, wenn Ihre Funktion 10 Anfragen erhält:



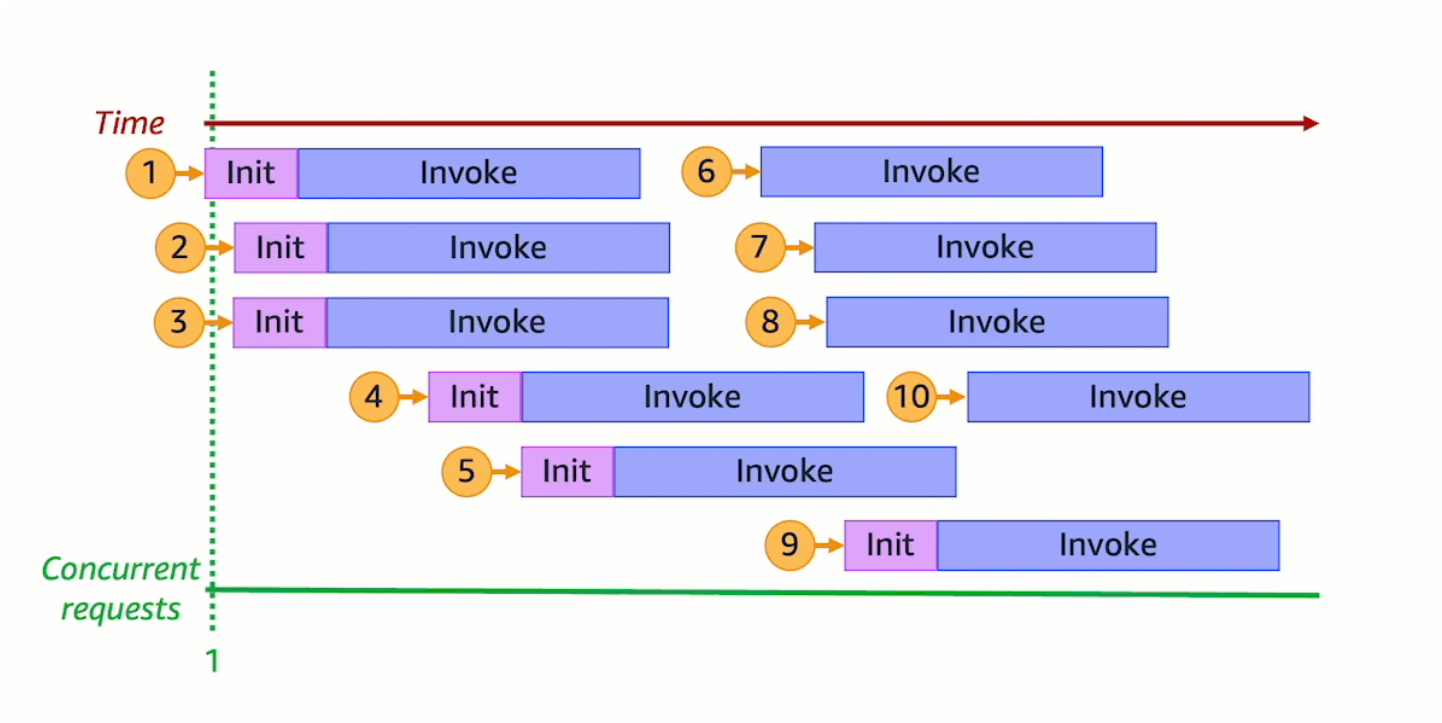
Im vorherigen Diagramm stellt jede horizontale Ebene eine einzelne Instance der Ausführungsumgebung dar (von A bis F). So behandelt Lambda jede Anfrage:

Anforderung	Lambda-Verhalten	Reasoning
1	Stellt neue Umgebung A bereit	Dies ist die erste Anfrage. Es sind keine Instances der Ausführungsumgebung verfügbar

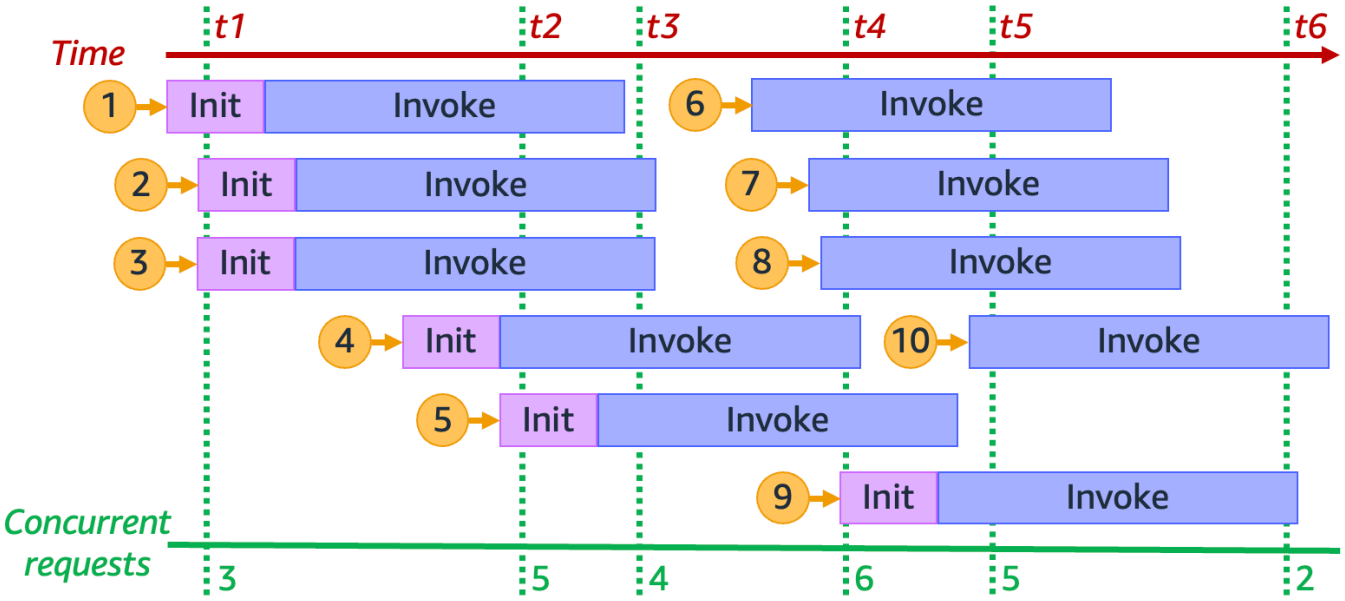
Anforderung	Lambda-Verhalten	Reasoning
2	Stellt neue Umgebung B bereit	Die vorhandene Instance der Ausführungsumgebung A ist ausgelastet
3	Stellt neue Umgebung C bereit	Die vorhandenen Instances der Ausführungsumgebung A und B sind beide ausgelastet
4	Stellt neue Umgebung D bereit	Die vorhandenen Instances der Ausführungsumgebung A, B und C sind alle ausgelastet
5	Stellt neue Umgebung E bereit	Die vorhandenen Instances der Ausführungsumgebung A, B, C und D sind alle ausgelast et
6	Wiederverwendung von Umgebung A	Die Instance der Ausführun gsumgebung A hat die Verarbeitung von Anfrage 1 abgeschlossen und ist jetzt verfügbar
7	Wiederverwendung von Umgebung B	Die Instance der Ausführun gsumgebung B hat die Verarbeitung von Anfrage 2 abgeschlossen und ist jetzt verfügbar
8	Wiederverwendung von Umgebung C	Die Instance der Ausführun gsumgebungB hat die Verarbeitung von Anfrage 3 abgeschlossen und ist jetzt verfügbar

Anforderung	Lambda-Verhalten	Reasoning
9	Stellt neue Umgebung F bereit	Die vorhandenen Instances der Ausführungsumgebung A, B, C, D und E sind alle ausgelastet
10	Wiederverwendung von Umgebung D	Die Instance der Ausführungsumgebung D hat die Verarbeitung von Anfrage 4 abgeschlossen und ist jetzt verfügbar

Wenn Ihre Funktion mehr gleichzeitige Anfragen erhält, erhöht Lambda als Reaktion darauf die Anzahl der Instances der Ausführungsumgebung. Die folgende Animation verfolgt die Anzahl der gleichzeitigen Anfragen im Laufe der Zeit:



Durch das Einfrieren der vorherigen Animation an sechs verschiedenen Zeitpunkten erhalten wir das folgende Diagramm:



Im vorherigen Diagramm können wir zu jedem Zeitpunkt eine vertikale Linie zeichnen und die Anzahl der Umgebungen zählen, die diese Linie schneiden. Dadurch erhalten wir die Anzahl der gleichzeitigen Anfragen zu diesem Zeitpunkt. Beispielsweise gibt es zum Zeitpunkt `t1` drei aktive Umgebungen, die drei gleichzeitige Anfragen bearbeiten. Die maximale Anzahl gleichzeitiger Anfragen in dieser Simulation wird zum Zeitpunkt `t4` erreicht, an dem sechs aktive Umgebungen sechs gleichzeitige Anfragen bearbeiten.

Zusammengefasst beschreibt die Gleichzeitigkeit Ihrer Funktion die Anzahl der Anfragen, die diese gleichzeitig verarbeitet. Als Reaktion auf eine Zunahme der Gleichzeitigkeit Ihrer Funktion stellt Lambda mehr Instances der Ausführungsumgebung bereit, um die Anfragenachfrage zu erfüllen.

Berechnen der Gleichzeitigkeit einer Funktion

Im Allgemeinen ist die Gleichzeitigkeit eines Systems die Fähigkeit, mehr als eine Aufgabe gleichzeitig zu bearbeiten. In Lambda beschreibt die Gleichzeitigkeit die Anzahl der Anfragen, die Ihre Funktion gleichzeitig bearbeitet. Eine schnelle und praktische Methode zur Messung der Gleichzeitigkeit einer Lambda-Funktion ist die Verwendung der folgenden Formel:

```
Concurrency = (average requests per second) * (average request duration in seconds)
```

Gleichzeitigkeit unterscheidet sich von Anfragen pro Sekunde. Nehmen wir beispielsweise an, Ihre Funktion erhält durchschnittlich 100 Anfragen pro Sekunde. Wenn die durchschnittliche Dauer einer Anfrage 1 Sekunde beträgt, beträgt die Nebenläufigkeit ebenfalls 100:

```
Concurrency = (100 requests/second) * (1 second/request) = 100
```

Wenn die durchschnittliche Anfragedauer jedoch 500 ms beträgt, liegt die Nebenläufigkeit bei 50:

```
Concurrency = (100 requests/second) * (0.5 second/request) = 50
```

Was bedeutet eine Gleichzeitigkeit von 50 in der Praxis? Wenn die durchschnittliche Anfragedauer 500 ms beträgt, können Sie davon ausgehen, dass eine Instance Ihrer Funktion zwei Anfragen pro Sekunde verarbeiten kann. Dementsprechend sind 50 Instances Ihrer Funktion erforderlich, um eine Last von 100 Anfragen pro Sekunde zu verarbeiten. Eine Gleichzeitigkeit von 50 bedeutet, dass Lambda 50 Instances der Ausführungsumgebung bereitstellen muss, um diesen Workload ohne Drosselung effizient zu bewältigen. So drückt man dies in Gleichungsform aus:

```
Concurrency = (100 requests/second) / (2 requests/second) = 50
```

Wenn Ihre Funktion die doppelte Anzahl von Anfragen empfängt (200 Anfragen pro Sekunde), aber nur die Hälfte der Zeit benötigt, um jede Anfrage zu verarbeiten (250 ms), beträgt die Nebenläufigkeit immer noch 50:

```
Concurrency = (200 requests/second) * (0.25 second/request) = 50
```

Testen Sie Ihr Verständnis von Gleichzeitigkeit

Angenommen, Sie haben eine Funktion, deren Ausführung im Durchschnitt 200 ms benötigt. Während der Spitzenlast beobachten Sie 5 000 Anfragen pro Sekunde. Wie hoch ist die Gleichzeitigkeit Ihrer Funktion während der Spitzenlast?

Antwort

Die durchschnittliche Funktionsdauer beträgt 200 ms oder 0,2 Sekunden. Mithilfe der Nebenläufigkeitsformel können Sie die Zahlen einfügen, um eine Nebenläufigkeit von 1 000 zu erhalten:

```
Concurrency = (5,000 requests/second) * (0.2 seconds/request) = 1,000
```

Alternativ bedeutet eine durchschnittliche Funktionsdauer von 200 ms, dass Ihre Funktion 5 Anfragen pro Sekunde verarbeiten kann. Um den Workload von 5 000 Anfragen pro Sekunde zu bewältigen, benötigen Sie 1 000 Instances der Ausführungsumgebung. Die Gleichzeitigkeit beträgt also 1 000:

```
Concurrency = (5,000 requests/second) / (5 requests/second) = 1,000
```

Verständnis der reservierten Gleichzeitigkeit und der bereitgestellten Gleichzeitigkeit

Standardmäßig verfügt Ihr Konto über eine Nebenläufigkeitsgrenze von 1 000 gleichzeitigen Ausführungen für alle Funktionen in einer Region. Ihre Funktionen teilen sich diesen Pool von 1 000 gleichzeitigen Anwendungen auf Bedarfsbasis. Ihre Funktionen werden gedrosselt (d. h. sie beginnen, Anfragen zu verwerfen), wenn Ihnen die verfügbare Nebenläufigkeit ausgeht.

Einige Ihrer Funktionen sind möglicherweise kritischer als andere. Daher sollten Sie die Nebenläufigkeitseinstellungen konfigurieren, um sicherzustellen, dass kritische Funktionen die benötigte Gleichzeitigkeit erhalten. Es gibt zwei Arten von Gleichzeitigkeitskontrollen: reservierte Gleichzeitigkeit und bereitgestellte Gleichzeitigkeit.

- Verwenden Sie die reservierte Parallelität, um die maximale und die minimale Anzahl gleichzeitiger Instances festzulegen, um einen Teil der Parallelität Ihres Kontos für eine Funktion zu reservieren. Dies ist nützlich, wenn Sie nicht möchten, dass andere Funktionen die gesamte verfügbare nicht reservierte Gleichzeitigkeit in Anspruch nehmen. Wenn für eine Funktion Gleichzeitigkeit reserviert ist, kann keine andere Funktion diese Gleichzeitigkeit nutzen.
- Verwenden Sie die bereitgestellte Gleichzeitigkeit, um eine Reihe von Umgebungs-Instances für eine Funktion vorab zu initialisieren. Dies ist nützlich, um Kaltstartlatenzen zu reduzieren.

Reservierte Gleichzeitigkeit

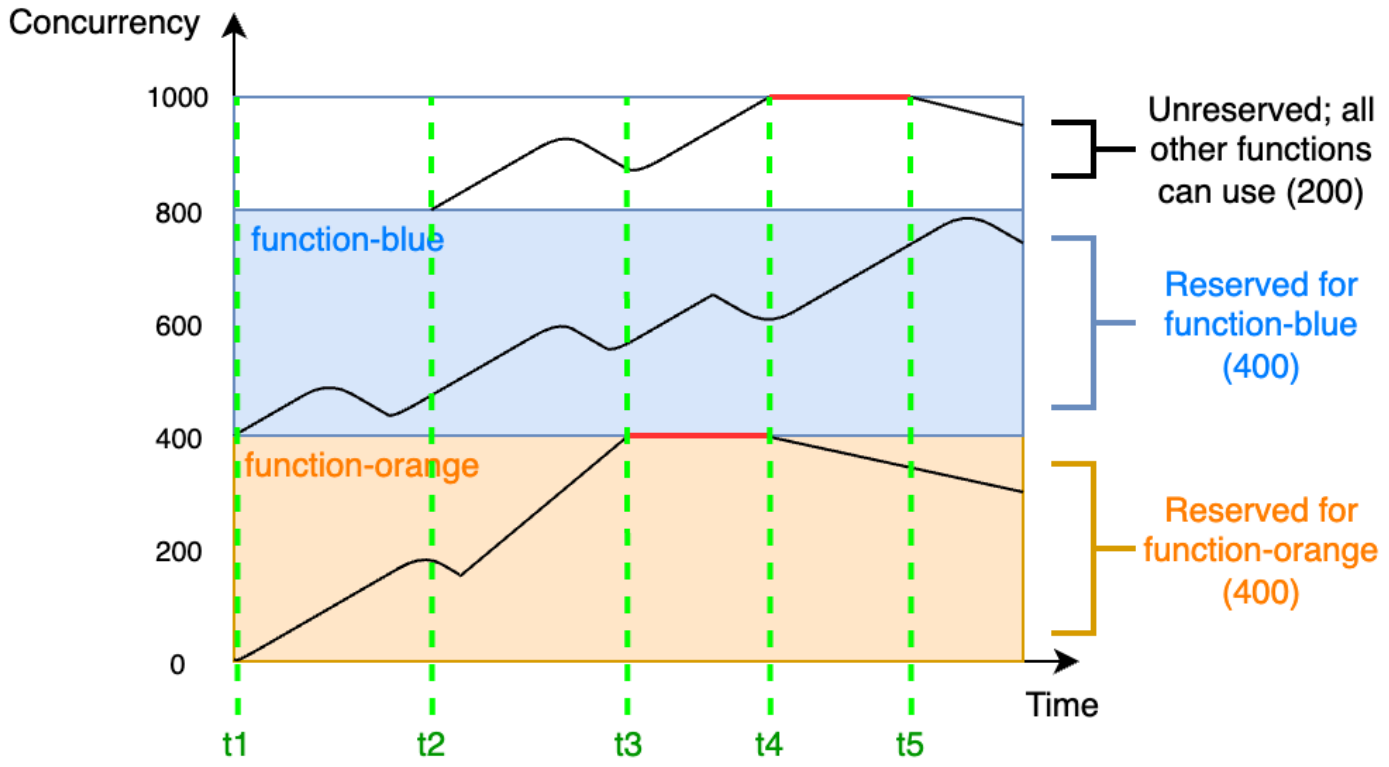
Wenn Sie sicherstellen möchten, dass für Ihre Funktion jederzeit ein gewisses Maß an Gleichzeitigkeit verfügbar ist, verwenden Sie reservierte Gleichzeitigkeit.

Reservierte Parallelität legt die maximale und minimale Anzahl der gleichzeitigen Instances fest, die Sie Ihrer Funktion zuweisen möchten. Wenn Sie einer Funktion reservierte Gleichzeitigkeit zuweisen, kann keine andere Funktion diese Gleichzeitigkeit nutzen. Mit anderen Worten, das Festlegen von reservierter Gleichzeitigkeit kann sich auf den Gleichzeitigkeits-Pool auswirken, der anderen

Funktionen zur Verfügung steht. Funktionen, die nicht über reservierte Gleichzeitigkeit verfügen, teilen sich den verbleibenden Pool an nicht reservierter Gleichzeitigkeit.

Das Konfigurieren der reservierten Gleichzeitigkeit wird auf das gesamte Gleichzeitigkeitslimit Ihres Kontos angerechnet. Für die Konfiguration reservierter Gleichzeitigkeit für eine Funktion wird keine Gebühr erhoben.

Betrachten Sie das folgende Diagramm, um die reservierte Gleichzeitigkeit besser zu verstehen:



In diesem Diagramm ist das Gleichzeitigkeitslimit Ihres Kontos für alle Funktionen in diesem Bereich auf das Standardlimit von 1 000 festgelegt. Angenommen, Sie haben zwei kritische Funktionen, `function-blue` und `function-orange`, die routinemäßig hohe Aufrufvolumina erwarten. Sie entscheiden sich, 400 Einheiten reservierter Gleichzeitigkeit für `function-blue` und 400 Einheiten reservierter Gleichzeitigkeit für `function-orange` einzusetzen. In diesem Beispiel müssen sich alle anderen Funktionen in Ihrem Konto die verbleibenden 200 Einheiten an nicht reservierter Gleichzeitigkeit teilen.

Das Diagramm enthält fünf wichtige Punkte:

- Bei `t1` beginnen sowohl `function-orange` als auch `function-blue` mit dem Empfang von Anfragen. Jede Funktion beginnt, ihren zugewiesenen Teil der reservierten Gleichzeitigkeitseinheiten zu verbrauchen.
- Bei `t2` erhalten `function-orange` und `function-blue` stetig mehr Anfragen. Gleichzeitig stellen Sie einige andere Lambda-Funktionen bereit, die mit dem Empfang von Anfragen beginnen. Sie weisen diesen anderen Funktionen keine reservierte Nebenläufigkeit zu. Diese beginnen mit der Verwendung der verbleibenden 200 Einheiten der nicht reservierten Gleichzeitigkeit.
- Bei `t3` erreicht `function-orange` die maximale Gleichzeitigkeit von 400. Obwohl es an anderer Stelle in Ihrem Konto ungenutzte Gleichzeitigkeit gibt, kann `function-orange` nicht darauf zugreifen. Die rote Linie zeigt an, dass `function-orange` gedrosselt wird und Lambda-Anfragen möglicherweise verwirft.
- Bei `t4` beginnt `function-orange` weniger Anfragen zu erhalten und wird nicht mehr gedrosselt. Ihre anderen Funktionen erfahren jedoch einen Anstieg des Datenverkehrs und werden gedrosselt. Obwohl es an anderer Stelle in Ihrem Konto ungenutzte Gleichzeitigkeit gibt, können diese anderen Funktionen nicht darauf zugreifen. Die rote Linie zeigt an, dass Ihre anderen Funktionen gedrosselt werden.
- Bei `t5` beginnen andere Funktionen weniger Anfragen zu erhalten und werden nicht mehr gedrosselt.

Anhand dieses Beispiels können Sie sehen, dass die Reservierung von Gleichzeitigkeit folgende Auswirkungen hat:

- Ihre Funktion kann unabhängig von anderen Funktionen in Ihrem Konto skaliert werden. Alle Funktionen Ihres Kontos in derselben Region, die über keine reservierte Gleichzeitigkeit verfügen, teilen sich den Pool an nicht reservierter Gleichzeitigkeit. Ohne reservierte Gleichzeitigkeit können andere Funktionen möglicherweise Ihre gesamte verfügbare Gleichzeitigkeit verbrauchen. Dadurch wird verhindert, dass kritische Funktion bei Bedarf hochskalieren.
- Ihre Funktion kann nicht unkontrolliert aufskaliert werden. Die reservierte Parallelität legt eine Obergrenze für die maximale und minimale Parallelität Ihrer Funktion fest. Dies bedeutet, dass Ihre Funktion keine für andere Funktionen reservierte Gleichzeitigkeit oder Gleichzeitigkeit aus dem nicht reservierten Pool verwenden kann. Darüber hinaus dient die reservierte Parallelität sowohl als Untergrenze als auch als Obergrenze. Sie reserviert die angegebene Kapazität ausschließlich für Ihre Funktion und verhindert gleichzeitig, dass sie über diese Grenze hinaus skaliert wird. Sie können Gleichzeitigkeit reservieren, um zu verhindern, dass Ihre Funktion die gesamte verfügbare Gleichzeitigkeit in Ihrem Konto verwendet oder nachgelagerte Ressourcen überlastet.

- Möglicherweise können Sie nicht die gesamte verfügbare Gleichzeitigkeit Ihres Kontos nutzen. Das Reservieren von Gleichzeitigkeit wird auf Ihr Kontolimit für Gleichzeitigkeit angerechnet, aber das bedeutet auch, dass andere Funktionen diesen Teil der reservierten Gleichzeitigkeit nicht verwenden können. Wenn Ihre Funktion nicht die gesamte Gleichzeitigkeit verbraucht, die Sie dafür reservieren, verschwenden Sie diese Gleichzeitigkeit effektiv. Das ist kein Problem, es sei denn, andere Funktionen in Ihrem Konto könnten von der ungenutzten Gleichzeitigkeit profitieren.

Informationen zur Verwaltung der reservierten Nebenläufigkeitseinstellungen für Ihre Funktionen finden Sie unter Konfigurieren reservierter Gleichzeitigkeit für eine Funktion.

Provisioned Concurrency

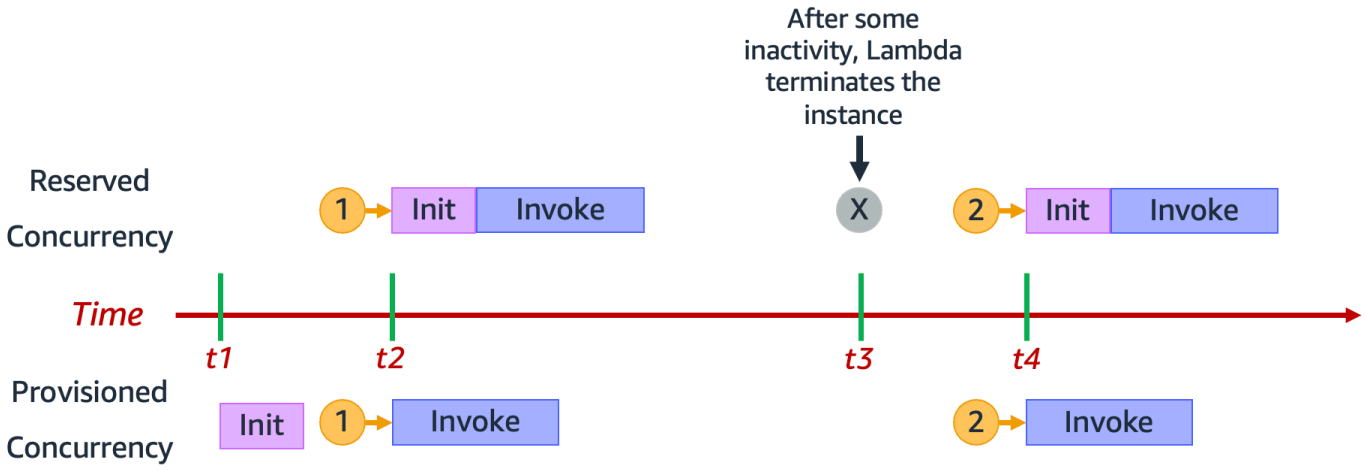
Sie verwenden die reservierte Gleichzeitigkeit, um die maximale Anzahl der für eine Lambda-Funktion reservierten Ausführungsumgebungen festzulegen. Keine dieser Umgebungen ist jedoch vorinitialisiert. Infolgedessen können Ihre Funktionsaufrufe länger dauern, da Lambda zuerst die neue Umgebung initialisieren muss, bevor diese zum Aufrufen Ihrer Funktion verwendet werden kann. Wenn Lambda eine neue Umgebung initialisieren muss, um einen Aufruf auszuführen, wird dies als Kaltstart bezeichnet. Um Kaltstarts zu vermeiden, können Sie die bereitgestellte Gleichzeitigkeit verwenden.

Die bereitgestellte Nebenläufigkeit beschreibt die Anzahl der vorinitialisierten Ausführungsumgebungen, die Sie Ihrer Funktion zuweisen möchten. Wenn Sie bereitgestellte Gleichzeitigkeit für eine Funktion festlegen, initialisiert Lambda diese Anzahl von Ausführungsumgebungen, damit sie bereit sind, sofort auf Funktionsanfragen zu reagieren.

> **Note**
>
> Die Verwendung von bereitgestellter Gleichzeitigkeit verursacht zusätzliche Gebühren für Ihr Konto. Wenn Sie mit den Java-11- oder Java-17-Laufzeiten arbeiten, können Sie auch Lambda SnapStart verwenden, um Kaltstartprobleme ohne zusätzliche Kosten zu beheben. SnapStart verwendet im Cache gespeicherte Snapshots Ihrer Ausführungsumgebung, um die Startleistung erheblich zu verbessern. Sie können SnapStart und bereitgestellte Gleichzeitigkeit nicht in derselben Funktionsversion verwenden. Weitere Informationen zu den Features, Einschränkungen und unterstützten Regionen von SnapStart finden Sie unter Verbesserung der Startleistung mit Lambda SnapStart.

Bei der Verwendung von bereitgestellter Gleichzeitigkeit erneuert Lambda weiterhin die Ausführungsumgebung im Hintergrund. Dies kann beispielsweise nach einem Aufruffehler der Fall sein. Lambda stellt jedoch zu jedem Zeitpunkt sicher, dass die Anzahl der vorinitialisierten Umgebungen dem Wert der von Ihrer Funktion bereitgestellten Gleichzeitigkeitseinstellung entspricht. Wichtig: Selbst wenn Sie die bereitgestellte Parallelität verwenden, kann es dennoch zu einer Kaltstartverzögerung kommen, wenn Lambda die Ausführungsumgebung zurücksetzen muss.

Im Gegensatz dazu kann Lambda bei Verwendung von reservierter Parallelität eine Umgebung nach einem Zeitraum der Inaktivität vollständig beenden. Das folgende Diagramm veranschaulicht dies, indem der Lebenszyklus einer einzelnen Ausführungsumgebung beim Konfigurieren Ihrer Funktion mit reservierter Gleichzeitigkeit mit bereitgestellter Gleichzeitigkeit verglichen wird.

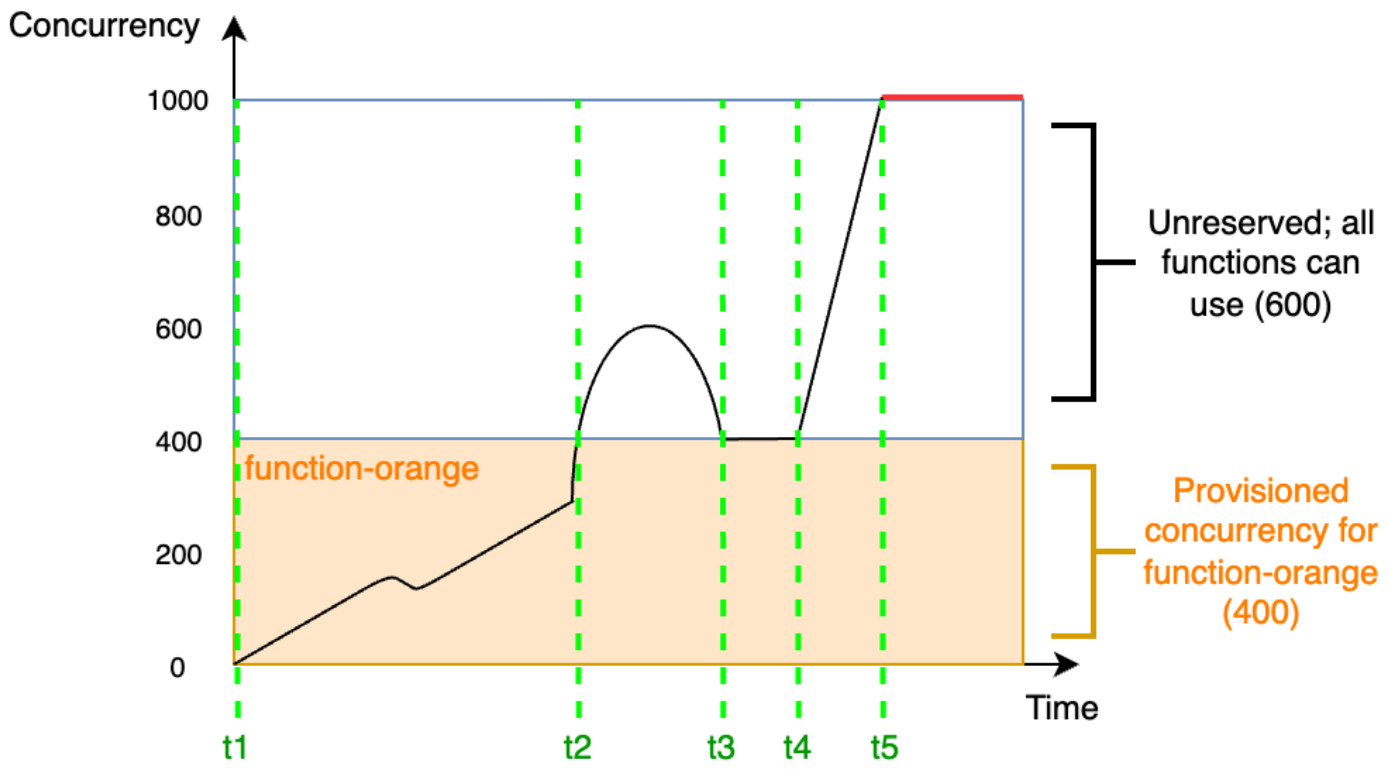


Das Diagramm enthält vier wichtige Punkte:

Zeit	Reservierte Gleichzeitigkeit	Provisioned Concurrency
t1	Es passiert nichts.	Lambda initialisiert eine Instance der Ausführun gsumgebung vorab.
t2	Anfrage 1 wird empfangen . Lambda muss eine neue Instance der Ausführun gsumgebung initialisieren.	Anfrage 1 wird empfangen . Lambda verwendet die vorinitialisierte Umgebungs-Instance.

Zeit	Reservierte Gleichzeitigkeit	Provisioned Concurrency
t3	Nach einer gewissen Inaktivität beendet Lambda die aktive Umgebungs-Instance.	Es passiert nichts.
t4	Anfrage 2 wird empfangen. Lambda muss eine neue Instance der Ausführungsumgebung initialisieren.	Anfrage 2 wird empfangen. Lambda verwendet die vorinitialisierte Umgebungs-Instance.

Betrachten Sie das folgende Diagramm, um die bereitgestellte Gleichzeitigkeit besser zu verstehen:

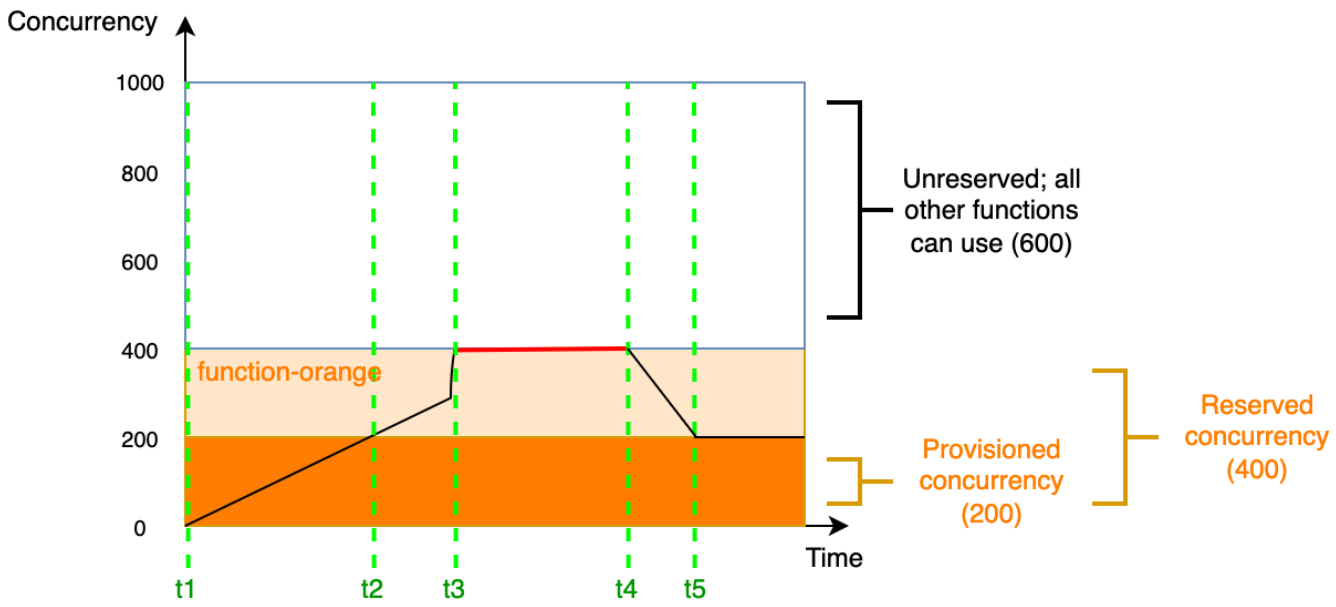


In diesem Diagramm haben Sie eine Kontogleichzeitigkeitslimit von 1 000. Sie entscheiden sich, 400 Einheiten der bereitgestellten Gleichzeitigkeit an `function-orange` zu vergeben. Alle Funktionen in Ihrem Konto, einschließlich `function-orange`, können die verbleibenden 600 Einheiten der nicht reservierten Gleichzeitigkeit nutzen.

Das Diagramm enthält fünf wichtige Punkte:

- Bei `t1` beginnt `function-orange` mit dem Empfang von Anfragen. Da Lambda 400 Instances der Ausführungsumgebung vorinitialisiert hat, ist `function-orange` für den sofortigen Aufruf bereit.
- Bei `t2` erreicht `function-orange` 400 gleichzeitige Anfragen. Infolgedessen geht `function-orange` die bereitgestellte Gleichzeitigkeit aus. Da jedoch immer noch keine nicht reservierte Gleichzeitigkeit verfügbar ist, kann Lambda dies verwenden, um zusätzliche Anfragen an `function-orange` zu verarbeiten (es gibt keine Drosselung). Lambda muss neue Instances erstellen, um diese Anfragen zu bearbeiten und bei Ihrer Funktion kann es zu Kaltstartlatenzen kommen.
- Bei `t3` kehrt `function-orange` nach einem kurzen Anstieg des Datenverkehrs zu 400 gleichzeitigen Anfragen zurück. Lambda ist wieder in der Lage, alle Anfragen ohne Kaltstartlatenzen zu bearbeiten.
- Bei `t4` kommt es bei Funktionen in Ihrem Konto zu einem starken Anstieg des Datenverkehrs. Dieser Anstieg kann von `function-orange` oder einer anderen Funktion in Ihrem Konto stammen. Lambda verwendet nicht reservierte Gleichzeitigkeit, um diese Anfragen zu bearbeiten.
- Bei `t5` erreichen die Funktionen in Ihrem Konto das maximale Gleichzeitigkeitslimit von 1 000 und es kommt zu Drosselungen.

Im vorherigen Beispiel wurde nur die bereitgestellte Nebenläufigkeit berücksichtigt. In der Praxis können Sie für eine Funktion sowohl die bereitgestellte als auch die reservierte Gleichzeitigkeit festlegen. Sie können dies beispielsweise tun, wenn Sie eine Funktion haben, die wochentags eine konstante Anzahl von Aufrufen verarbeitet, aber an den Wochenenden regelmäßig Datenverkehrsspitzen aufweist. In diesem Fall könnten Sie die bereitgestellte Gleichzeitigkeit verwenden, um eine Basismenge an Umgebungen festzulegen, die Anfragen an Wochentagen verarbeiten, und die reservierte Gleichzeitigkeit verwenden, um die Spitzenlasten am Wochenende zu verarbeiten. Betrachten Sie das folgende Diagramm:



Nehmen wir in diesem Diagramm an, dass Sie 200 Einheiten bereitgestellter Gleichzeitigkeit und 400 Einheiten reservierter Gleichzeitigkeit für `function-orange` konfigurieren. Da Sie die reservierte Gleichzeitigkeit konfiguriert haben, kann `function-orange` keine der 600 Einheiten der nicht reservierten Gleichzeitigkeit verwenden.

Dieses Diagramm enthält fünf wichtige Punkte:

- Bei `t1` beginnt `function-orange` mit dem Empfang von Anfragen. Da Lambda 200 Instances der Ausführungsumgebung vorinitialisiert hat, ist `function-orange` für den sofortigen Aufruf bereit.
- Bei `t2` verbraucht `function-orange` die gesamte bereitgestellte Gleichzeitigkeit. `function-orange` kann weiterhin Anfragen über reservierte Gleichzeitigkeit bearbeiten, allerdings kann es bei diesen Anfragen zu Kaltstartlatenzen kommen.
- Bei `t3` erreicht `function-orange` 400 gleichzeitige Anfragen. Dadurch verbraucht `function-orange` die gesamte reservierte Gleichzeitigkeit. Da `function-orange` keine nicht reservierte Gleichzeitigkeit nutzen kann, werden die Anfragen gedrosselt.
- Bei `t4` beginnt `function-orange` weniger Anfragen zu erhalten und wird nicht mehr gedrosselt.
- Bei `t5` sinkt `function-orange` auf 200 gleichzeitige Anfragen, sodass alle Anfragen wieder bereitgestellte Nebenläufigkeit verwenden können (d. h. keine Kaltstartlatenzen).

Sowohl die reservierte Gleichzeitigkeit als auch die bereitgestellte Gleichzeitigkeit werden auf das Gleichzeitigkeitslimit Ihres Kontos und die regionalen Kontingente angerechnet. Mit anderen Worten, die Zuweisung von reservierter und bereitgestellter Gleichzeitigkeit kann sich auf den Gleichzeitigkeit -Pool auswirken, der anderen Funktionen zur Verfügung steht. Die Konfiguration von bereitgestellter Gleichzeitigkeit verursacht Gebühren für Ihre AWS-Konto.

> **Note**
>
> Wenn die Menge der bereitgestellten Nebenläufigkeit für die Versionen und Aliase einer Funktion zur reservierten Nebenläufigkeit der Funktion hinzukommt, werden alle Aufrufe mit bereitgestellter Nebenläufigkeit ausgeführt. Diese Konfiguration hat auch zur Folge, dass die nicht veröffentlichte Version der Funktion (`$LATEST`) gedrosselt wird, was die Ausführung verhindert. Sie können nicht mehr bereitgestellte Gleichzeitigkeit als reservierte Gleichzeitigkeit für eine Funktion zuweisen.

Informationen zur Verwaltung der Einstellungen für reservierte Nebenläufigkeit für Ihre Funktionen finden Sie unter Konfigurieren von Provisioned Concurrency für eine Funktion. Informationen zum Automatisieren der bereitgestellten Gleichzeitigkeitsskalierung basierend auf einem Zeitplan oder der Anwendungsnutzung finden Sie unter Auto Scaling von Anwendungen zur Automatisierung der Verwaltung von Gleichzeitigkeit bei der Bereitstellung.

So weist Lambda bereitgestellte Gleichzeitigkeit zu

Bereitgestellte Gleichzeitigkeit wird nicht sofort nach der Konfiguration online geschaltet. Lambda beginnt die Zuweisung bereitgestellter Gleichzeitigkeit nach ein oder zwei Minuten Vorbereitung. Für jede Funktion kann Lambda bis zu 6 000 Ausführungsumgebungen pro Minute bereitstellen, unabhängig von AWS-Region. Dies entspricht exakt der Skalierungsrate für die Gleichzeitigkeit von Funktionen.

Wenn Sie eine Anforderung zur Zuweisung bereitgestellter Gleichzeitigkeit einreichen, können Sie erst wieder auf diese Umgebungen zugreifen, wenn Lambda die Zuweisung vollständig abgeschlossen hat. Wenn Sie beispielsweise 5 000 bereitgestellte Gleichzeitigkeit anfordern, kann keine Ihrer Anfragen bereitgestellte Gleichzeitigkeit verwenden, bis Lambda die Zuweisung der 5 000 Ausführungsumgebungen vollständig abgeschlossen hat.

Vergleich zwischen reservierter und bereitgestellter Gleichzeitigkeit

Die folgende Tabelle fasst reservierte und bereitgestellte Nebenläufigkeit zusammen und vergleicht sie.

Thema	Reservierte Gleichzeitigkeit	Provisioned Concurrency
Definition	Maximale Anzahl von Instances in der Ausführun gsumgebung für Ihre Funktion.	Festgelegte Anzahl von vorbereiteten Instances in der Ausführungsumgebung für Ihre Funktion.
Bereitstellungsverhalten	Lambda stellt neue Instances auf On-Demand-Basis bereit.	Lambda stellt Instances vorab bereit (d. h. bevor Ihre Funktion mit dem Empfang von Anfragen beginnt).
Kaltstartverhalten	Kaltstartlatenz möglich, da Lambda bei Bedarf neue Instances erstellen muss.	Eine Kaltstartlatenz ist nicht möglich, da Lambda keine Instances On-Demand erstellen muss.
Drosselungsverhalten	Die Funktion wurde gedrossel t, als das reservierte Gleichzei tigkeitslimit erreicht wurde.	Wenn die reservierte Gleichzei tigkeit nicht festgelegt ist: Die Funktion verwendet die nicht reservierte Gleichzei tigkeit, wenn das bereitges tellte Gleichzeitigkeitslimit erreicht ist. Wenn reservierte Gleichzei tigkeit festgelegt ist: Funktion wird gedrosselt, wenn das Limit für reservierte Gleichzei tigkeit erreicht wird.

Thema	Reservierte Gleichzeitigkeit	Provisioned Concurrency
Standardverhalten, falls nicht festgelegt	Die Funktion verwendet nicht reservierte Gleichzeitigkeit, die in Ihrem Konto verfügbar ist.	Lambda stellt keine Instances vorab bereit. Wenn die reservierte Gleichzeitigkeit nicht festgelegt ist, verwendet die Funktion stattdessen die nicht reservierte Gleichzei tigkeit, die in Ihrem Konto verfügbar ist. Wenn reservierte Gleichzei tigkeit festgelegt ist: die Funktion verwendet reservier te Gleichzeitigkeit.
Preisgestaltung	Keine zusätzlichen Gebühren.	Verursacht zusätzliche Gebühren.

Gleichzeitigkeit und Anforderungen pro Sekunde verstehen

Wie im vorherigen Abschnitt erwähnt, unterscheidet sich die Gleichzeitigkeit von den Anforderungen pro Sekunde. Diese Unterscheidung ist besonders wichtig, wenn Sie mit Funktionen arbeiten, die eine durchschnittliche Anforderungsdauer von weniger als 100 ms haben.

Für alle Funktionen in Ihrem Konto erzwingt Lambda ein Limit für Anfragen pro Sekunde, das dem 10-fachen der Gleichzeitigkeit Ihres Kontos entspricht. Da der Standardgrenzwert für die Gleichzeitigkeit von Konten beispielsweise 1 000 beträgt, können die Funktionen in Ihrem Konto maximal 10 000 Anfragen pro Sekunde verarbeiten.

Stellen Sie sich etwa eine Funktion mit einer durchschnittlichen Anforderungsdauer von 50 ms vor. Bei 20 000 Anforderungen pro Sekunde lautet die Gleichzeitigkeit dieser Funktion wie folgt:

```
Concurrency = (20,000 requests/second) * (0.05 second/request) = 1,000
```

Ausgehend von diesem Ergebnis könnte man annehmen, dass die Gleichzeitigkeitsgrenze von 1 000 Konten ausreicht, um diese Last zu bewältigen. Aufgrund der Begrenzung auf 10 000 Anfragen

pro Sekunde kann Ihre Funktion jedoch nur 10 000 der insgesamt 20 000 Anfragen pro Sekunde bearbeiten. Bei dieser Funktion kommt es zu einer Drosselung.

Fazit: Sie müssen bei der Konfiguration der Nebenläufigkeitseinstellungen für Ihre Funktionen sowohl Nebenläufigkeit als auch Anforderungen pro Sekunde berücksichtigen. In diesem Fall müssen Sie eine Erhöhung des Kontogleichzeitigkeitslimits auf 2 000 beantragen, da sich dadurch Ihr Gesamtlimit für Anfragen pro Sekunde auf 20 000 erhöhen würde.

> **Note**
>
> Basierend auf diesem Limit für Anfragen pro Sekunde ist es falsch zu sagen, dass jede Lambda-Ausführungsumgebung nur maximal zehn Anfragen pro Sekunde verarbeiten kann. Anstatt die Auslastung einer einzelnen Ausführungsumgebung zu beobachten, berücksichtigt Lambda bei der Berechnung der Quoten nur die Gesamtgleichzeitigkeit und die Gesamtanforderungen pro Sekunde.

Testen Sie Ihr Verständnis von Gleichzeitigkeit (Funktionen unter 100 ms)

Angenommen, Sie haben eine Funktion, deren Ausführung im Durchschnitt 20 ms benötigt. Während der Spitzenlast beobachten Sie 30 000 Anfragen pro Sekunde. Wie hoch ist die Gleichzeitigkeit Ihrer Funktion während der Spitzenlast?

Antwort

Die durchschnittliche Funktionsdauer beträgt 20 ms oder 0,02 Sekunden. Mithilfe der Parallelitätsformel können Sie die Zahlen einfügen, um eine Parallelität von 600 zu erhalten:

```
Concurrency = (30,000 requests/second) * (0.02 seconds/request) = 600
```

Standardmäßig scheint eine Kontogleichzeitigkeitslimit von 1 000 ausreichend zu sein, um diese Last zu bewältigen. Das Limit für Anfragen pro Sekunde von 10 000 reicht jedoch nicht aus, um die eingehenden 30 000 Anfragen pro Sekunde zu verarbeiten. Um den 30 000 Anfragen in vollem Umfang gerecht zu werden, müssen Sie eine Erhöhung des Kontogleichzeitigkeitslimits auf 3 000 oder mehr beantragen.

Das Limit für Anfragen pro Sekunde gilt für alle Kontingente in Lambda, die Gleichzeitigkeit beinhalten. Mit anderen Worten: Sie gilt für synchrone On-Demand-Funktionen, Funktionen, die bereitgestellte Gleichzeitigkeit verwenden, und Skalierungsverhalten bei Gleichzeitigkeit. Im

Folgenden finden Sie beispielsweise einige Szenarien, in denen Sie sowohl Ihre Grenzwerte für Parallelität als auch für Anfragen pro Sekunde sorgfältig abwägen müssen:

- Bei einer Funktion, die On-Demand-Parallelität verwendet, kann es zu einem Anstieg der Parallelität um 500 Mal alle 10 Sekunden oder um 5 000 Anfragen pro Sekunde alle 10 Sekunden kommen, je nachdem, was zuerst eintritt.
- Angenommen, Sie haben eine Funktion, die eine bereitgestellte Gleichzeitigkeitszuweisung von 10 hat. Diese Funktion geht nach 10 Gleichzeitigkeiten oder 100 Anfragen pro Sekunde, je nachdem, was zuerst eintritt, in die On-Demand-Gleichzeitigkeit über.

Gleichzeitigkeitskontingente

Lambda legt Kontingente für die Gesamtmenge der Nebenläufigkeit fest, die Sie für alle Funktionen in einer Region verwenden können. Diese Kontingente bestehen auf zwei Ebenen:

- Auf Kontoebene können Ihre Funktionen standardmäßig bis zu 1 000 Einheiten an Gleichzeitigkeit verwenden. Informationen zum Erhöhen dieses Limits finden Sie unter Anfordern einer Kontingenterhöhung im Benutzerhandbuch für Service Quotas.
- Auf Funktionsebene können Sie standardmäßig bis zu 900 Nebenläufigkeitseinheiten für alle Ihre Funktionen reservieren. Unabhängig von Ihrem Gesamtlimit für die Kontonebenläufigkeit reserviert Lambda immer 100 Nebenläufigkeitseinheiten für Ihre Funktionen, die Nebenläufigkeit nicht explizit reservieren. Wenn Sie beispielsweise Ihr Limit für die Nebenläufigkeit Ihres Kontos auf 2 000 erhöht haben, können Sie auf Funktionsebene bis zu 1 900 Einheiten an Nebenläufigkeit reservieren.
- Sowohl auf Konto- als auch auf Funktionsebene erzwingt Lambda außerdem ein Limit für Anfragen pro Sekunde, das dem Zehnfachen der entsprechenden Gleichzeitigkeitsquote entspricht. Dies gilt beispielsweise für die Gleichzeitigkeit auf Kontoebene, für Funktionen, die die Gleichzeitigkeit auf Abruf verwenden, für Funktionen, die die provisorische Gleichzeitigkeit verwenden und für das Skalierungsverhalten der Gleichzeitigkeit. Weitere Informationen finden Sie unter the section called “Gleichzeitigkeit und Anforderungen pro Sekunde verstehen”.

Um Ihr Kontingent für die Nebenläufigkeit auf Kontoebene zu überprüfen, verwenden Sie die AWS Command Line Interface (AWS CLI), um den folgenden Befehl auszuführen:

```
aws lambda get-account-settings
```

Die Ausgabe sollte ungefähr wie folgt aussehen:

```
{
    "AccountLimit": {
        "TotalCodeSize": 80530636800,
        "CodeSizeUnzipped": 262144000,
        "CodeSizeZipped": 52428800,
        "ConcurrentExecutions": 1000,
        "UnreservedConcurrentExecutions": 900
    },
    "AccountUsage": {
        "TotalCodeSize": 410759889,
        "FunctionCount": 8
    }
}
```

`ConcurrentExecutions` ist Ihr Gesamtkontingent für die Nebenläufigkeit auf Kontoebene. `UnreservedConcurrentExecutions` ist die Menge an verbleibender Nebenläufigkeit, die Sie Ihren Funktionen noch zuweisen können.

Erhält Ihre Funktion mehr Anfragen, sorgt Lambda automatisch für die Skalierung der Anzahl der Ausführungsumgebungen, bis Ihr Konto das Nebenläufigkeitskontingent erreicht. Zum Schutz vor einer Überskalierung als Reaktion auf plötzliche Datenverkehrsspitzen begrenzt Lambda jedoch, wie schnell Ihre Funktionen skaliert werden können. Diese Gleichzeitigkeits-Skalierungsrate ist die maximale Rate, mit der Funktionen in Ihrem Konto als Reaktion auf erhöhte Anforderungen skaliert werden können. (Das heißt, sie gibt an, wie schnell Lambda neue Ausführungsumgebungen erstellen kann.) Die Skalierungsrate für die Gleichzeitigkeit unterscheidet sich von der Gleichzeitigkeitsgrenze auf Kontoebene, die die Gesamtmenge der Gleichzeitigkeit darstellt, die Ihren Funktionen zur Verfügung steht.

In jedem AWS-Region und für jede Funktion beträgt die Skalierungsrate für die Gleichzeitigkeit 1 000 Instanzen der Ausführungsumgebung alle 10 Sekunden (oder 10 000 Anfragen pro Sekunde alle 10 Sekunden).. Mit anderen Worten: Lambda kann jeder Ihrer Funktionen alle 10 Sekunden höchstens 1 000 zusätzliche Instanzen der Ausführungsumgebung zuweisen oder 10 000 zusätzliche Anfragen pro Sekunde bearbeiten.

Normalerweise müssen Sie sich über diese Einschränkung keine Gedanken machen. Die Skalierungsrate von Lambda ist für die meisten Anwendungsfälle ausreichend.

Wichtig ist, dass die Skalierungsrate für die Gleichzeitigkeit eine Grenze auf Funktionsebene darstellt. Das bedeutet, dass jede Funktion in Ihrem Konto unabhängig von anderen Funktionen skaliert werden kann.

Weitere Informationen zum Skalierungsverhalten finden Sie unter Lambda-Skalierungsverhalten.

Konfigurieren reservierter Gleichzeitigkeit für eine Funktion

In Lambda beschreibt die Gleichzeitigkeit die Anzahl der In-Flight-Anfragen, die Ihre Funktion gleichzeitig bearbeitet. Es gibt zwei Arten von Gleichzeitigkeitskontrollen:

- Reservierte Gleichzeitigkeit – Hiermit wird sowohl die maximale als auch die minimale Anzahl der gleichzeitigen Instances festgelegt, die Ihrer Funktion zugewiesen sind. Wenn für eine Funktion Gleichzeitigkeit reserviert ist, kann keine andere Funktion diese Gleichzeitigkeit nutzen. Reservierte Gleichzeitigkeit ist nützlich, um sicherzustellen, dass Ihre kritischsten Funktionen immer genügend Gleichzeitigkeit haben, um eingehende Anfragen zu bearbeiten. Darüber hinaus kann reservierte Gleichzeitigkeit verwendet werden, um die Gleichzeitigkeit einzuschränken, um zu verhindern, dass nachgelagerte Ressourcen wie Datenbankverbindungen überlastet werden. Die reservierte Gleichzeitigkeit dient sowohl als Untergrenze als auch als Obergrenze. Sie reserviert die angegebene Kapazität ausschließlich für Ihre Funktion und verhindert gleichzeitig, dass sie über diese Grenze hinaus skaliert wird. Für die Konfiguration reservierter Gleichzeitigkeit für eine Funktion wird keine zusätzliche Gebühr erhoben.
- Die bereitgestellte Gleichzeitigkeit beschreibt die Anzahl der vorinitialisierten Ausführungsumgebungen, die Ihrer Funktion zugewiesen sind. Diese Ausführungsumgebungen sind bereit, sofort auf eingehende Funktionsanforderungen zu reagieren. Bereitgestellte Gleichzeitigkeit ist nützlich, um Kaltstartlatenzen für Funktionen zu reduzieren, und wurde entwickelt, um Funktionen mit Antwortzeiten im zweistelligen Millisekundenbereich verfügbar zu machen. Im Allgemeinen profitieren interaktive Workloads am meisten von diesem Feature. Dabei handelt es sich um Anwendungen, bei denen Benutzer Anfragen initiieren, wie z. B. Web- und Mobilanwendungen und sie reagieren am empfindlichsten auf Latenz. Asynchrone Workloads, wie Datenverarbeitungspipelines, reagieren häufig weniger empfindlich auf Latenz und benötigen daher in der Regel keine bereitgestellte Gleichzeitigkeit. Die Konfiguration von bereitgestellter Gleichzeitigkeit verursacht Gebühren für Ihr AWS-Konto.

In diesem Thema wird beschrieben, wie Sie reservierte Gleichzeitigkeit verwalten und konfigurieren. Eine konzeptionelle Übersicht über diese beiden Arten von Gleichzeitigkeitssteuerungen finden Sie unter Reservierte Gleichzeitigkeit und bereitgestellte Gleichzeitigkeit. Weitere Informationen zum Konfigurieren der bereitgestellten Gleichzeitigkeit finden Sie unter the section called “Konfigurieren von Provisioned Concurrency”.

> **Note**
>
> Lambda-Funktionen, die mit einer Amazon-MQ-Ereignisquellenzuordnung verknüpft sind, haben standardmäßig die maximale Gleichzeitigkeit. Für Apache Active MQ ist die maximale Anzahl der gleichzeitigen Instances 5. Für Rabbit MQ ist die maximale Anzahl der gleichzeitigen Instances 1. Wenn Sie für Ihre Funktion eine reservierte oder bereitgestellte Parallelität festlegen, werden diese Grenzwerte nicht geändert. Um eine Erhöhung der standardmäßigen maximalen Parallelität bei der Verwendung von Amazon MQ zu beantragen, wenden Sie sich an. Support

Abschnitte

- Konfigurieren reservierter Gleichzeitigkeit
- Genaue Schätzung reservierter Gleichzeitigkeit für eine Funktion

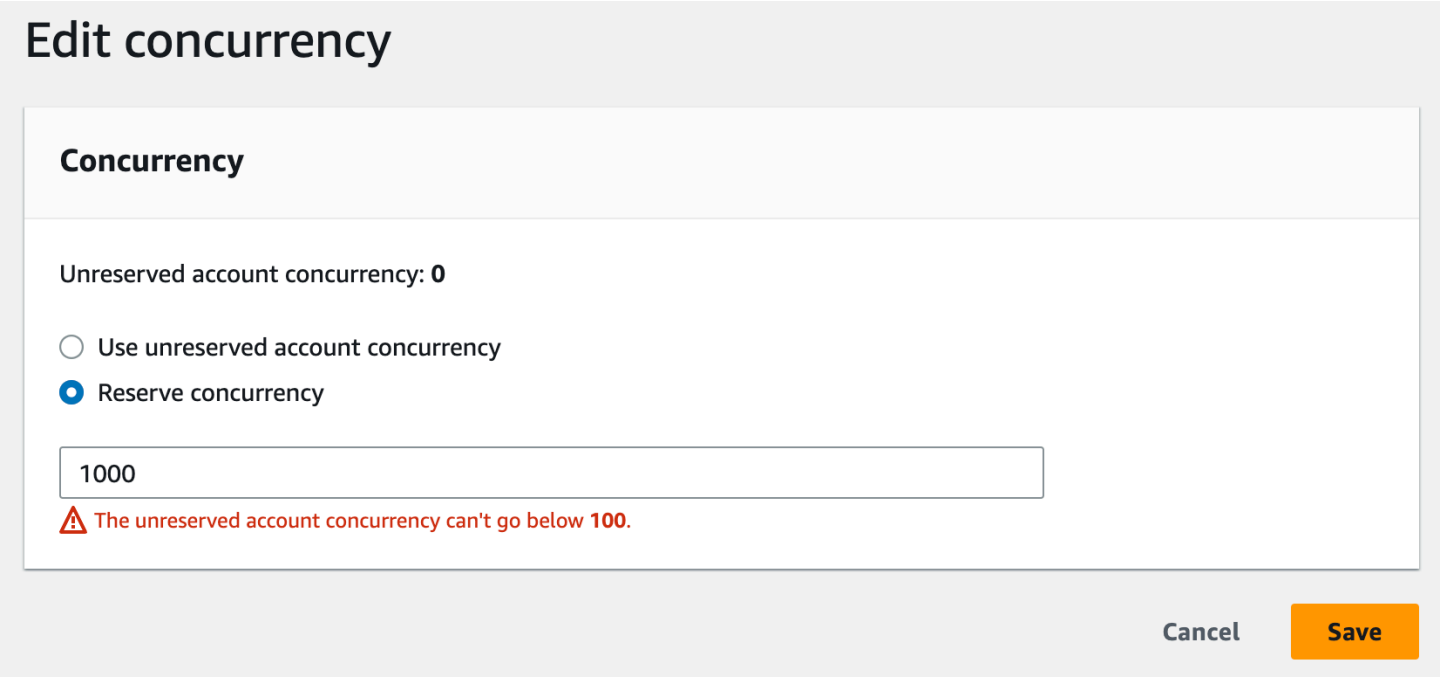
Konfigurieren reservierter Gleichzeitigkeit

Sie können die Einstellungen der reservierten Gleichzeitigkeit für eine Funktion in der Lambda-Konsole oder mit der Lambda-API konfigurieren.

Reservieren von Gleichzeitigkeit für eine Funktion (Konsole)

1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.
2. Wählen Sie die Funktion aus, für sie Sie Gleichzeitigkeit reservieren möchten.
3. Wählen Sie Konfiguration und anschließend Gleichzeitigkeit aus.
4. Wählen Sie unter Concurrency (Parallelität) die Option Edit (Bearbeiten).
5. Wählen Sie Reserve concurrency (Parallelität reservieren). Geben Sie die Menge an Gleichzeitigkeit an, die für die Funktion reserviert werden soll.
6. Wählen Sie Speichern.

Sie können bis zum Wert von Nicht reservierte Kontonebenläufigkeit minus 100 reservieren. Die verbleibenden 100 Gleichzeitigkeitseinheiten sind für Funktionen bestimmt, die keine reservierte Gleichzeitigkeit verwenden. Wenn für Ihr Konto beispielsweise eine Gleichzeitigkeitsbeschränkung von 1 000 gilt, können Sie nicht alle 1 000 Gleichzeitigkeitseinheiten für eine einzelne Funktion reservieren.



Das Reservieren von Gleichzeitigkeit für eine Funktion hat Auswirkungen auf den Pool, der für andere Funktionen verfügbar ist. Wenn Sie beispielsweise 100 Gleichzeitigkeitseinheiten für `function-a` reservieren, müssen sich andere Funktionen in Ihrem Konto die verbleibenden 900 Gleichzeitigkeitseinheiten teilen, auch wenn `function-a` nicht alle 100 reservierten Gleichzeitigkeitseinheiten verwendet.

Um eine Funktion absichtlich zu drosseln, setzen Sie die reservierte Gleichzeitigkeit dafür auf 0. Dadurch kann die Funktion keine Ereignisse mehr verarbeiten, bis Sie die Beschränkung aufheben.

Verwenden Sie für die Konfiguration von reservierter Gleichzeitigkeit mithilfe der Lambda-API die folgenden API-Vorgänge:

- PutFunctionConcurrency
- GetFunctionConcurrency
- DeleteFunctionConcurrency

Um beispielsweise reservierte Gleichzeitigkeit mit der AWS Command Line Interface (CLI) zu konfigurieren, verwenden Sie den `put-function-concurrency`-Befehl. Der folgende Befehl reserviert 100 Gleichzeitigkeitseinheiten für eine Funktion namens `my-function`:

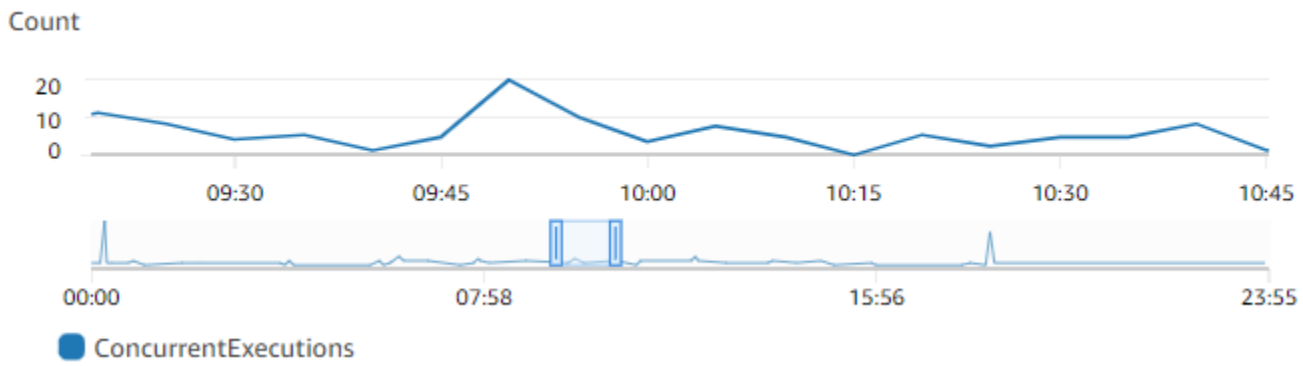
```
aws lambda put-function-concurrency --function-name my-function \
    --reserved-concurrent-executions 100
```

Die Ausgabe sollte ungefähr wie folgt aussehen:

```
{
    "ReservedConcurrentExecutions": 100
}
```

Genaue Schätzung reservierter Gleichzeitigkeit für eine Funktion

Wenn Ihre Funktion gerade Datenverkehr verarbeitet, können Sie ihre Gleichzeitigkeitsmetriken ganz einfach mit CloudWatch-Metriken anzeigen. Insbesondere zeigt Ihnen die `ConcurrentExecutions`-Metrik die Anzahl der gleichzeitigen Aufrufe für jede Funktion in Ihrem Konto.



Aus dem vorstehenden Diagramm geht hervor, dass diese Funktion jederzeit im Durchschnitt 5 bis 10 gleichzeitige Anforderungen verarbeitet und an einem typischen Tag Spitzenwerte von 20 Anforderungen erreicht. Angenommen, Ihr Konto enthält viele andere Funktionen. Wenn diese Funktion für Ihre Anwendung von entscheidender Bedeutung ist und Sie keine Anforderungen abbrechen möchten, verwenden Sie mindestens 20 als Einstellung für die reservierte Gleichzeitigkeit.

Denken Sie alternativ daran, dass Sie die Gleichzeitigkeit auch mit der folgenden Formel berechnen können:

```
Concurrency = (average requests per second) * (average request duration in seconds)
```

Wenn Sie die durchschnittlichen Anforderungen pro Sekunde mit der durchschnittlichen Anforderungsdauer in Sekunden multiplizieren, erhalten Sie eine grobe Schätzung, wie viel Gleichzeitigkeit Sie reservieren müssen. Sie können die durchschnittlichen Anfragen pro Sekunde anhand der `Invocation`-Metrik und die durchschnittliche Anfragedauer in Sekunden anhand der

`Duration`-Metrik schätzen. Weitere Details finden Sie unter Verwendung von CloudWatch-Metriken mit Lambda.

Sie sollten auch mit Ihren vor- und nachgelagerten Durchsatzbeschränkungen vertraut sein. Lambda-Funktionen lassen sich zwar nahtlos abhängig von der Last skalieren, aber vor- und nachgelagerte Abhängigkeiten verfügen möglicherweise nicht über die gleichen Durchsatzfähigkeiten. Wenn Sie die Skalierbarkeit Ihrer Funktion begrenzen müssen, konfigurieren Sie die reservierte Gleichzeitigkeit für Ihre Funktion.

Konfigurieren von Provisioned Concurrency für eine Funktion

In Lambda beschreibt die Gleichzeitigkeit die Anzahl der In-Flight-Anfragen, die Ihre Funktion gleichzeitig bearbeitet. Es gibt zwei Arten von Gleichzeitigkeitskontrollen:

- Reservierte Gleichzeitigkeit – Hiermit wird sowohl die maximale als auch die minimale Anzahl der gleichzeitigen Instances festgelegt, die Ihrer Funktion zugewiesen sind. Wenn für eine Funktion Gleichzeitigkeit reserviert ist, kann keine andere Funktion diese Gleichzeitigkeit nutzen. Reservierte Gleichzeitigkeit ist nützlich, um sicherzustellen, dass Ihre kritischsten Funktionen immer genügend Gleichzeitigkeit haben, um eingehende Anfragen zu bearbeiten. Darüber hinaus kann reservierte Gleichzeitigkeit verwendet werden, um die Gleichzeitigkeit einzuschränken, um zu verhindern, dass nachgelagerte Ressourcen wie Datenbankverbindungen überlastet werden. Die reservierte Gleichzeitigkeit dient sowohl als Untergrenze als auch als Obergrenze. Sie reserviert die angegebene Kapazität ausschließlich für Ihre Funktion und verhindert gleichzeitig, dass sie über diese Grenze hinaus skaliert wird. Für die Konfiguration reservierter Gleichzeitigkeit für eine Funktion wird keine zusätzliche Gebühr erhoben.
- Die bereitgestellte Gleichzeitigkeit beschreibt die Anzahl der vorinitialisierten Ausführungsumgebungen, die Ihrer Funktion zugewiesen sind. Diese Ausführungsumgebungen sind bereit, sofort auf eingehende Funktionsanforderungen zu reagieren. Bereitgestellte Gleichzeitigkeit ist nützlich, um Kaltstartlatenzen für Funktionen zu reduzieren, und wurde entwickelt, um Funktionen mit Antwortzeiten im zweistelligen Millisekundenbereich verfügbar zu machen. Im Allgemeinen profitieren interaktive Workloads am meisten von diesem Feature. Dabei handelt es sich um Anwendungen, bei denen Benutzer Anfragen initiieren, wie z. B. Web- und Mobilanwendungen und sie reagieren am empfindlichsten auf Latenz. Asynchrone Workloads, wie Datenverarbeitungspipelines, reagieren häufig weniger empfindlich auf Latenz und benötigen daher in der Regel keine bereitgestellte Gleichzeitigkeit. Die Konfiguration von bereitgestellter Gleichzeitigkeit verursacht Gebühren für Ihr AWS-Konto.

In diesem Thema wird beschrieben, wie Sie bereitgestellte Gleichzeitigkeit verwalten und konfigurieren. Eine konzeptionelle Übersicht über diese beiden Arten von Gleichzeitigkeitssteuerungen finden Sie unter Reservierte Gleichzeitigkeit und bereitgestellte Gleichzeitigkeit. Weitere Informationen zur Konfiguration der reservierten Gleichzeitigkeit finden Sie unter the section called “Konfigurieren reservierter Gleichzeitigkeit”.

> **Note**
>
> Lambda-Funktionen, die mit einer Amazon-MQ-Ereignisquellenzuordnung verknüpft sind, haben standardmäßig die maximale Gleichzeitigkeit. Für Apache Active MQ ist die maximale Anzahl der gleichzeitigen Instances 5. Für Rabbit MQ ist die maximale Anzahl der gleichzeitigen Instances 1. Wenn Sie für Ihre Funktion eine reservierte oder bereitgestellte Parallelität festlegen, werden diese Grenzwerte nicht geändert. Um eine Erhöhung der standardmäßigen maximalen Parallelität bei der Verwendung von Amazon MQ zu beantragen, wenden Sie sich an. Support

Abschnitte

- Konfigurieren von Provisioned Concurrency
- Genaue Schätzung der erforderlichen bereitgestellten Gleichzeitigkeit für eine Funktion
- Optimierung des Funktionscodes bei Verwendung der bereitgestellten Gleichzeitigkeit
- Verwendung von Umgebungsvariablen zur Anzeige und Steuerung des bereitgestellten Gleichzeitigkeitsverhaltens
- Verstehen des Protokollierungs- und Abrechnungsverhaltens bei bereitgestellter Gleichzeitigkeit
- Auto Scaling von Anwendungen zur Automatisierung der Verwaltung von Gleichzeitigkeit bei der Bereitstellung

Konfigurieren von Provisioned Concurrency

Sie können die Einstellungen der bereitgestellten Gleichzeitigkeit für eine Funktion mit der Lambda-Konsole oder mit der Lambda-API konfigurieren.

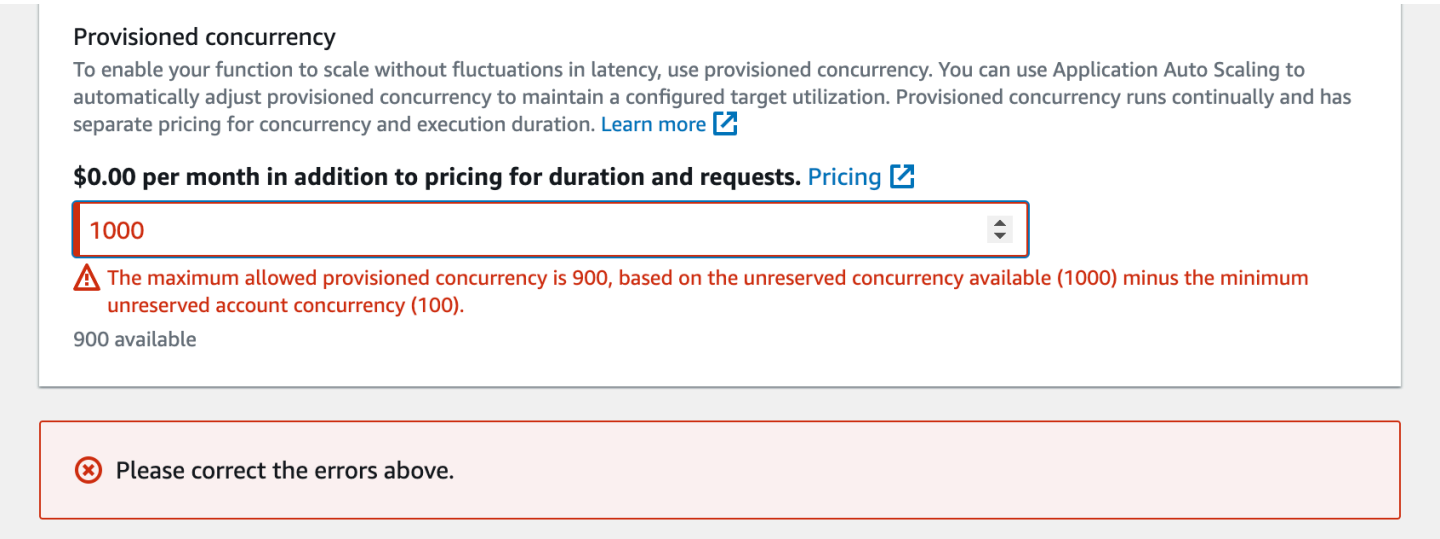
So weisen Sie einer Funktion bereitgestellte Gleichzeitigkeit zu (Konsole)

1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.
2. Wählen Sie die Funktion aus, der Sie bereitgestellte Gleichzeitigkeit zuweisen möchten.
3. Wählen Sie Konfiguration und anschließend Gleichzeitigkeit aus.
4. Wählen Sie unter Provisioned concurrency configurations (Bereitgestellte Gleichzeitigkeitskonfigurationen) die Option Add (Hinzufügen) aus.
5. Wählen Sie den Qualifizierungstyp und den Alias oder die Version aus.

> **Note**
>
> Sie können die bereitgestellte Gleichzeitigkeit nicht mit der $LATEST-Version einer Funktion verwenden.
> Wenn Ihre Funktion eine Ereignisquelle hat, müssen Sie sicherstellen, dass die Ereignisquelle auf den richtigen Funktionsalias bzw. die richtige Funktionsversion verweist. Anderenfalls verwendet Ihre Funktion keine bereitgestellten Gleichzeitigkeitsumgebungen.

6. Geben Sie unter Bereitgestellte Gleichzeitigkeit eine Zahl ein.
7. Wählen Sie Speichern.

Sie können in Ihrem Konto Gleichzeitigkeit bis zur nicht reservierten Konto-Gleichzeitigkeit minus 100 konfigurieren. Die verbleibenden 100 Gleichzeitigkeitseinheiten sind für Funktionen bestimmt, die keine reservierte Gleichzeitigkeit verwenden. Wenn Ihr Konto beispielsweise einen Gleichzeitigkeitsgrenzwert von 1 000 aufweist und Sie keiner anderen Funktion reservierte oder bereitgestellte Gleichzeitigkeit zugewiesen haben, können Sie maximal 900 bereitgestellte Gleichzeitigkeitseinheiten für eine einzelne Funktion konfigurieren.



Die Konfiguration der bereitgestellten Gleichzeitigkeit für eine Funktion wirkt sich auf den Pool für die Gleichzeitigkeit aus, der für andere Funktionen verfügbar ist. Wenn Sie beispielsweise 100 bereitgestellte Gleichzeitigkeitseinheiten für `function-a` konfigurieren, müssen sich andere Funktionen in Ihrem Konto die verbleibenden 900 Gleichzeitigkeitseinheiten teilen. Das gilt auch, wenn `function-a` nicht alle 100 Einheiten verwendet.

Sie können für eine Funktion sowohl reservierte als auch bereitgestellte Gleichzeitigkeit festlegen. In solchen Fällen darf die bereitgestellte Gleichzeitigkeit die reservierte Gleichzeitigkeit nicht überschreiten.

Dieser Grenzwert gilt auch für Funktionsversionen. Die maximale Menge an bereitgestellter Gleichzeitigkeit, die Sie einer bestimmten Funktionsversion zuweisen können, entspricht der reservierten Gleichzeitigkeit der Funktion abzüglich der in anderen Funktionsversionen bereitgestellten Gleichzeitigkeit.

Verwenden Sie für die Konfiguration der bereitgestellten Gleichzeitigkeit mithilfe der Lambda-API die folgenden API-Operationen:

- PutProvisionedConcurrencyConfig
- GetProvisionedConcurrencyConfig
- ListProvisionedConcurrencyConfigs
- DeleteProvisionedConcurrencyConfig

Um beispielsweise bereitgestellte Gleichzeitigkeit mit der AWS Command Line Interface (CLI) zu konfigurieren, verwenden Sie den `put-provisioned-concurrency-config`-Befehl. Der folgende Befehl weist 100 Einheiten bereitgestellter Gleichzeitigkeit für den Alias `BLUE` einer Funktion namens `my-function` zu:

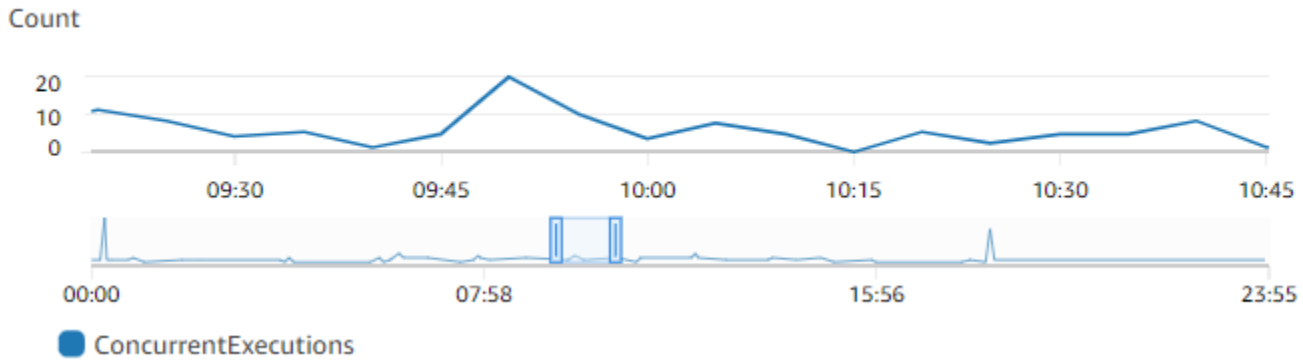
```
aws lambda put-provisioned-concurrency-config --function-name my-function \
  --qualifier BLUE \
  --provisioned-concurrent-executions 100
```

Die Ausgabe sollte ungefähr wie folgt aussehen:

```
{
  "Requested ProvisionedConcurrentExecutions": 100,
  "Allocated ProvisionedConcurrentExecutions": 0,
  "Status": "IN_PROGRESS",
  "LastModified": "2023-01-21T11:30:00+0000"
}
```

Genaue Schätzung der erforderlichen bereitgestellten Gleichzeitigkeit für eine Funktion

Sie können die Gleichzeitigkeitsmetriken jeder aktiven Funktion mithilfe von CloudWatch-Metriken anzeigen. Insbesondere zeigt Ihnen die `ConcurrentExecutions`-Metrik die Anzahl der gleichzeitigen Aufrufe für Funktionen in Ihrem Konto.



Aus dem vorstehenden Diagramm geht hervor, dass diese Funktion jederzeit im Durchschnitt 5 bis 10 gleichzeitige Anforderungen verarbeitet und Spitzenwerte von 20 Anforderungen erreicht. Angenommen, Ihr Konto enthält viele andere Funktionen. Wenn diese Funktion für Ihre Anwendung von entscheidender Bedeutung ist und Sie bei jedem Aufruf eine Reaktion mit geringer Latenz wünschen, legen Sie für die bereitgestellte Gleichzeitigkeit eine Zahl fest, die größer oder gleich 20 ist.

Denken Sie daran, dass Sie die Gleichzeitigkeit auch mit der folgenden Formel berechnen können:

```
Concurrency = (average requests per second) * (average request duration in seconds)
```

Um abzuschätzen, wie viel Gleichzeitigkeit Sie benötigen, multiplizieren Sie die durchschnittlichen Anfragen pro Sekunde mit der durchschnittlichen Anfragedauer in Sekunden. Sie können die durchschnittlichen Anfragen pro Sekunde anhand der `Invocation`-Metrik und die durchschnittliche Anfragedauer in Sekunden anhand der `Duration`-Metrik schätzen.

Beim Konfigurieren der bereitgestellten Gleichzeitigkeit schlägt Lambda vor, einen Puffer von 10 % zusätzlich zu der Menge an Gleichzeitigkeit einzuplanen, die Ihre Funktion normalerweise benötigt. Wenn die Funktion normalerweise bei 200 gleichzeitigen Anforderungen Spitzenwerte erreicht, sollten Sie die bereitgestellte Gleichzeitigkeit auf 220 festlegen (200 gleichzeitige Anforderungen + 10 % = 220 bereitgestellte Gleichzeitigkeit).

Optimierung des Funktionscodes bei Verwendung der bereitgestellten Gleichzeitigkeit

Wenn Sie bereitgestellte Gleichzeitigkeit verwenden, sollten Sie Ihren Funktionscode so umstrukturieren, dass er für niedrige Latenzzeiten optimiert ist. Bei Funktionen, die bereitgestellte Gleichzeitigkeit verwenden, führt Lambda jeglichen Initialisierungscode, wie das Laden von Bibliotheken und die Instanziierung von Clients, während der Zuweisungszeit aus. Daher bietet es sich an, möglichst viele Initialisierungen außerhalb des Hauptfunktionshandlers zu verschieben, um bei tatsächlichen Funktionsaufrufen Latenzprobleme zu vermeiden. Im Gegensatz dazu bedeutet die Initialisierung von Bibliotheken oder die Instanziierung von Clients innerhalb Ihres Haupthandler-Codes, dass Ihre Funktion dies bei jedem Aufruf ausführen muss (dies geschieht unabhängig davon, ob Sie die bereitgestellte Gleichzeitigkeit verwenden).

Bei On-Demand-Aufrufen muss Lambda den Initialisierungscode möglicherweise jedes Mal neu ausführen, wenn die Funktion einen Kaltstart durchführt. In diesem Fall können Sie die Initialisierung einzelner Fähigkeiten verschieben, bis Ihre Funktion sie benötigt. Sehen Sie sich zum Beispiel den folgenden Steuerungsablauf für einen Lambda-Handler an:

```
def handler(event, context):
    ...
    if ( some_condition ):
        // Initialize CLIENT_A to perform a task
    else:
        // Do nothing
```

Im vorherigen Beispiel entschied sich der Entwickler dagegen, `CLIENT_A` außerhalb des Haupthandlers zu initialisieren, und initialisierte ihn stattdessen innerhalb der `if`-Anweisung. Auf diese Weise führt Lambda diesen Code nur aus, wenn die Bedingung `some_condition` erfüllt wird. Wenn `CLIENT_A` außerhalb des Haupthandlers initialisiert wird, führt Lambda diesen Code bei jedem Kaltstart aus. Das kann die Gesamtlatenz erhöhen.

Sie können Kaltstarts messen, während Lambda aufsteigt, indem Sie Ihre Funktion um eine Röntgenüberwachung ergänzen. Eine Funktion, die bereitgestellte Gleichzeitigkeit verwendet, weist kein Kaltstartverhalten auf, da die Ausführungsumgebung vor dem Aufruf vorbereitet wird. Bereitgestellte Gleichzeitigkeit muss jedoch auf eine bestimmte Version oder einen Alias einer Funktion angewendet werden, nicht auf die $LATEST-Version. Falls weiterhin ein Kaltstartverhalten auftritt, stellen Sie sicher, dass Sie die Alias-Version aufrufen, für die bereitgestellte Gleichzeitigkeit konfiguriert ist.

Verwendung von Umgebungsvariablen zur Anzeige und Steuerung des bereitgestellten Gleichzeitigkeitsverhaltens

Es ist möglich, dass Ihre Funktion die gesamte bereitgestellte Gleichzeitigkeit verbraucht. Lambda verwendet On-Demand-Instances, um zusätzlichen Datenverkehr zu verarbeiten. Um festzustellen, welche Art von Initialisierung Lambda für eine bestimmte Umgebung verwendet hat, überprüfen Sie den Wert der Umgebungsvariable `AWS_LAMBDA_INITIALIZATION_TYPE`. Diese Variable hat zwei mögliche Werte: `provisioned-concurrency` oder `on-demand`. Der Wert von `AWS_LAMBDA_INITIALIZATION_TYPE` ist unveränderlich und bleibt während der gesamten Lebensdauer der Umgebung konstant. Um den Wert einer Umgebungsvariablen in Ihrem Funktionscode zu überprüfen, finden Sie unter Abrufen von Lambda-Umgebungsvariablen.

Bei Verwendung der .NET 8-Laufzeit können Sie die Umgebungsvariable `AWS_LAMBDA_DOTNET_PREJIT` konfigurieren, um die Latenz für Funktionen zu verbessern, auch wenn diese keine bereitgestellte Gleichzeitigkeit verwenden. Die .NET-Laufzeitumgebung führt für jede Bibliothek, die der Code zum ersten Mal aufruft, eine verzögerte Kompilierung und Initialisierung durch. Daher kann der erste Aufruf einer Lambda-Funktion länger dauern als nachfolgende Aufrufe. Um dies zu verhindern, können Sie einen von drei Werten für `AWS_LAMBDA_DOTNET_PREJIT` wählen:

- `ProvisionedConcurrency`: Lambda führt für alle Umgebungen, die bereitgestellte Gleichzeitigkeit verwenden, eine zeitliche JIT-Kompilierung durch. Dies ist der Standardwert.
- `Always`: Lambda führt für jede Umgebung eine zeitliche JIT-Kompilierung durch, auch wenn die Funktion keine bereitgestellte Gleichzeitigkeit verwendet.
- `Never`: Lambda deaktiviert die JIT-Kompilierung für alle Umgebungen vorab.

Verstehen des Protokollierungs- und Abrechnungsverhaltens bei bereitgestellter Gleichzeitigkeit

Für bereitgestellte Gleichzeitigkeitsumgebungen wird der Initialisierungscode Ihrer Funktion während der Zuordnung und alle paar Stunden ausgeführt, da Lambda aktive Instances Ihrer Umgebung recycelt. Lambda rechnet die Initialisierung auch dann ab, wenn die Umgebungs-Instance niemals eine Anforderung verarbeitet. Die bereitgestellte Gleichzeitigkeit wird laufend ausgeführt und getrennt von den Initialisierungs- und Aufrufkosten in Rechnung gestellt. Weitere Einzelheiten finden Sie unter AWS Lambda-Preise.

Wenn Sie eine Lambda-Funktion mit bereitgestellter Gleichzeitigkeit konfigurieren, initialisiert Lambda diese Ausführungsumgebung vor, sodass sie vor Aufrufanforderungen verfügbar ist. Lambda protokolliert bei jeder Initialisierung der Umgebung das Feld Initialisierungsdauer der Funktion in einem platform-initReport-Protokollereignis im JSON-Protokollierungsformat. Um dieses Protokollereignis zu sehen, konfigurieren Sie Ihre JSON-Protokollebene auf mindestens `INFO`. Sie können die Telemetrie-API auch verwenden, um Plattformereignisse zu verarbeiten, bei denen das Feld „Initialisierungsdauer“ gemeldet wird.

Auto Scaling von Anwendungen zur Automatisierung der Verwaltung von Gleichzeitigkeit bei der Bereitstellung

Application Auto Scaling ermöglicht es Ihnen, die bereitgestellte Gleichzeitigkeit nach einem Zeitplan oder basierend auf der Auslastung zu verwalten. Wenn Sie vorhersehbare Auslastungsmuster für Ihre Funktion beobachten, verwenden Sie die geplante Skalierung. Wenn Sie möchten, dass Ihre Funktion einen bestimmten Auslastungsprozentsatz beibehält, verwenden Sie eine Richtlinie für die Zielverfolgungsskalierung.

> **Note**
>
> Wenn Sie Auto Scaling von Anwendungen verwenden, um die bereitgestellte Gleichzeitigkeit Ihrer Funktion zu verwalten, stellen Sie sicher, dass Sie zuerst einen anfänglichen Wert für die bereitgestellte Gleichzeitigkeit konfigurieren. Wenn Ihre Funktion keinen anfänglichen Gleichzeitigkeitswert hat, kann die Application Auto Scaling die Funktionsskalierung möglicherweise nicht ordnungsgemäß durchführen.

Geplante Skalierung

Application Auto Scaling ermöglicht es Ihnen, Ihren eigenen Skalierungsplan entsprechend vorhersehbarer Laständerungen festzulegen. Weitere Informationen und Beispiele finden Sie unter Geplante Skalierung für Application Auto Scaling im Benutzerhandbuch für Application Auto Scaling und unter Planen von mit AWS Lambda bereitgestellter Gleichzeitigkeit für wiederkehrende Spitzenauslastungen auf dem AWS-Compute-Blog.

Zielverfolgung

Bei der Zielverfolgung erstellt und verwaltet Application Auto Scaling eine Reihe von CloudWatch-Alarmen basierend auf der von Ihnen definierten Skalierungsrichtlinie. Wenn diese Alarme aktiviert

werden, passt Application Auto Scaling mithilfe der bereitgestellten Gleichzeitigkeit automatisch die Anzahl der zugewiesenen Umgebungen an. Verwenden Sie die Zielverfolgung für Anwendungen, die keine vorhersehbaren Auslastungsmuster aufweisen.

Verwenden Sie die API-Operationen `RegisterScalableTarget` und `PutScalingPolicy` Application Auto Scaling, um die bereitgestellte Gleichzeitigkeit mithilfe der Zielverfolgung zu skalieren. Wenn Sie beispielsweise die AWS Command Line Interface (CLI) verwenden, gehen Sie wie folgt vor:

1. Registrieren Sie den Alias einer Funktion als Skalierungsziel. Im folgenden Beispiel wird der BLUE-Alias einer Funktion mit dem Namen registriert `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. Wenden Sie eine Skalierungsrichtlinie auf das Ziel an. Im folgenden Beispiel wird Application Auto Scaling so konfiguriert, dass die bereitgestellte Gleichzeitigkeitskonfiguration für einen Alias so angepasst wird, dass die Auslastung bei 70 % liegt. Sie können jedoch jeden beliebigen Wert zwischen 10 % und 90 % anwenden.

```
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" }}'
```

Die Ausgabe sollte in etwa wie folgt aussehen:

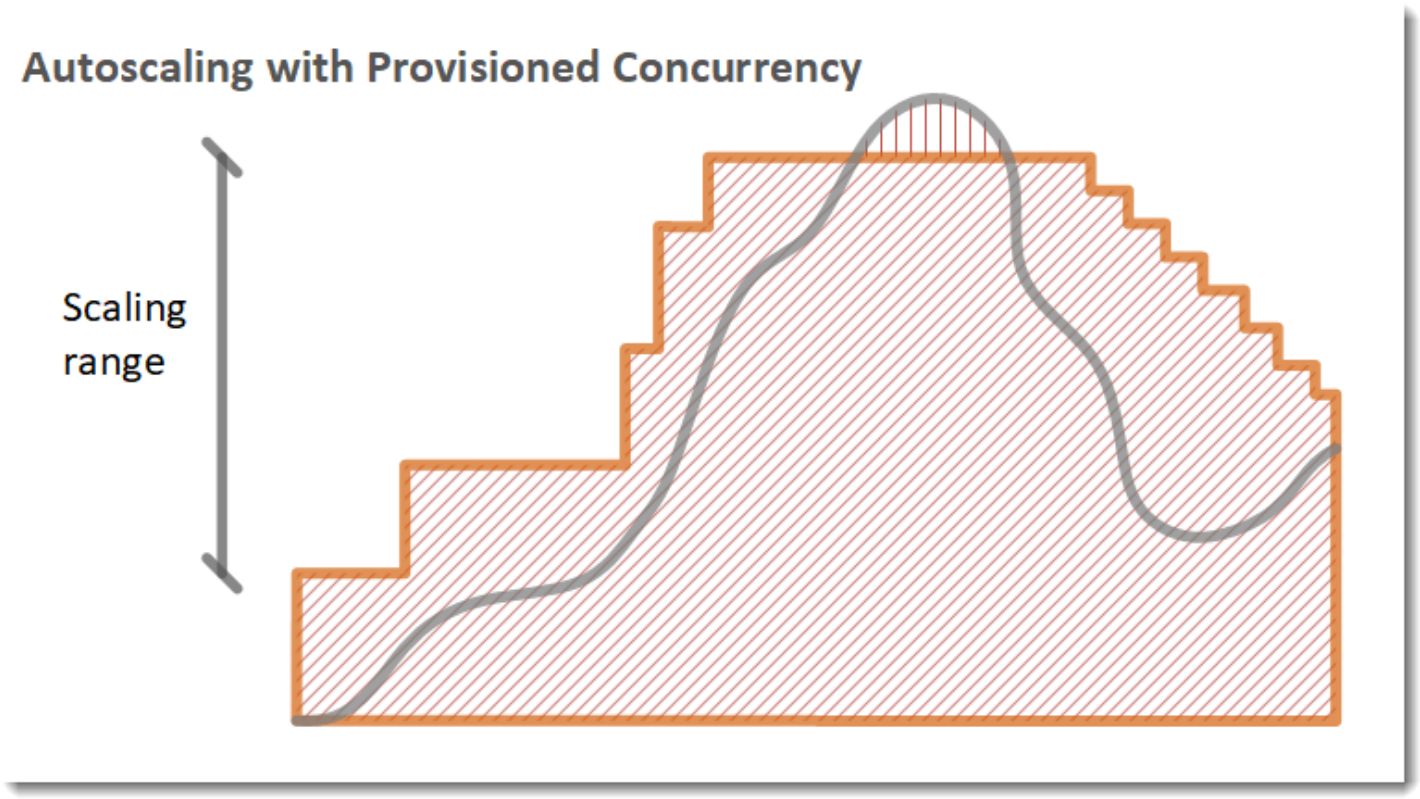
```
{
    "PolicyARN": "arn:aws:autoscaling:us-
east-2:123456789012:scalingPolicy:12266dbb-1524-xmpl-a64e-9a0a34b996fa:resource/lambda/
function:my-function:BLUE:policyName/my-policy",
    "Alarms": [
        {
```

```
            "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 erstellt zwei Alarme in CloudWatch. Der erste Alarm wird ausgelöst, wenn die Auslastung der bereitgestellten Gleichzeitigkeit konstant 70 % überschreitet. In diesem Fall weist Application Auto Scaling mehr bereitgestellte Gleichzeitigkeit zu, um die Auslastung zu reduzieren. Der zweite Alarm wird ausgelöst, wenn die Auslastung konstant unter 63 % (90 % des 70-%-Ziels) liegt. In diesem Fall reduziert Application Auto Scaling wird die bereitgestellte Gleichzeitigkeit des Alias.

> **Note**
>
> Lambda gibt die Metrik `ProvisionedConcurrencyUtilization` nur aus, wenn Ihre Funktion aktiv ist und Anforderungen empfängt. In Zeiten der Inaktivität werden keine Messwerte ausgegeben, und Ihre Auto-Scaling-Alarme wechseln in den Status `INSUFFICIENT_DATA`. Infolgedessen kann die Anwendung von Auto Scaling die von Ihrer Funktion bereitgestellte Gleichzeitigkeit nicht anpassen. Dies kann zu einer unerwarteten Abrechnung führen.

Im folgenden Beispiel skaliert eine Funktion basierend auf der Auslastung zwischen einem minimalen und einem maximalen Umfang bereitgestellter Gleichzeitigkeit.



Legende

- Funktions-Instances
- Offene Anforderungen
- Bereitgestellte Concurrency
- Standard-Gleichzeitigkeit

Wenn die Anzahl der offenen Anforderungen zunimmt, erhöht Application Auto Scaling die bereitgestellte Gleichzeitigkeit in großen Schritten, bis sie das konfigurierte Maximum erreicht. Danach kann die Funktion weiter die standardmäßige, nicht reservierte Gleichzeitigkeit skalieren, falls das Kontogleichzeitigkeitslimit nicht erreicht ist. Wenn die Auslastung sinkt und niedrig bleibt, verringert Application Auto Scaling die bereitgestellte Gleichzeitigkeit in kleineren periodischen Schritten.

Beide Alarme für Application Auto Scaling verwenden standardmäßig die Durchschnittsstatistik. Funktionen, die in kurzen Abständen auftretende Auslastungsmuster aufweisen, lösen diese Alarme möglicherweise nicht aus. Nehmen wir zum Beispiel an, Ihre Lambda-Funktion wird schnell ausgeführt (d. h. 20–100 ms) und der Datenverkehr kommt in kurzen Schüben (Bursts). In diesem Fall übersteigt die Anzahl der Anfragen die zugewiesene, bereitgestellte Gleichzeitigkeit während eines Bursts. Beim Application Auto Scaling muss die Burst-Last jedoch mindestens 3 Minuten lang aufrechterhalten werden, um zusätzliche Umgebungen bereitzustellen. Außerdem benötigen beide CloudWatch-Alarme 3 Datenpunkte, die den Zieldurchschnitt erreichen, damit die Auto-Scaling-Richtlinie aktiviert wird. Wenn bei Ihrer Funktion ein schneller Anstieg des Datenverkehrs auftritt, kann die Verwendung der Maximum-Statistik anstelle der Durchschnittsstatistik effektiver sein, um die bereitgestellte Gleichzeitigkeit zu skalieren und Kaltstarts zu minimieren.

Weitere Informationen zu Zielverfolgungs-Skalierungsrichtlinien finden Sie unter Zielverfolgungs-Skalierungsrichtlinien für das Application Auto Scaling.

Lambda-Skalierungsverhalten

Erhält Ihre Funktion mehr Anfragen, sorgt Lambda automatisch für die Skalierung der Anzahl der Ausführungsumgebungen, bis Ihr Konto das Nebenläufigkeitskontingent erreicht. Zum Schutz vor einer Überskalierung als Reaktion auf plötzliche Datenverkehrsspitzen begrenzt Lambda jedoch, wie schnell Ihre Funktionen skaliert werden können. Diese Gleichzeitigkeits-Skalierungsrate ist die maximale Rate, mit der Funktionen in Ihrem Konto als Reaktion auf erhöhte Anforderungen skaliert werden können. (Das heißt, sie gibt an, wie schnell Lambda neue Ausführungsumgebungen erstellen kann.) Die Skalierungsrate für die Gleichzeitigkeit unterscheidet sich von der Gleichzeitigkeitsgrenze auf Kontoebene, die die Gesamtmenge der Gleichzeitigkeit darstellt, die Ihren Funktionen zur Verfügung steht.

Nebenläufigkeitsskalierungsrate

In jedem AWS-Region und für jede Funktion beträgt die Skalierungsrate für die Gleichzeitigkeit 1 000 Instanzen der Ausführungsumgebung alle 10 Sekunden (oder 10 000 Anfragen pro Sekunde alle 10 Sekunden).. Mit anderen Worten: Lambda kann jeder Ihrer Funktionen alle 10 Sekunden höchstens 1 000 zusätzliche Instanzen der Ausführungsumgebung zuweisen oder 10 000 zusätzliche Anfragen pro Sekunde bearbeiten.

Normalerweise müssen Sie sich über diese Einschränkung keine Gedanken machen. Die Skalierungsrate von Lambda ist für die meisten Anwendungsfälle ausreichend.

Wichtig ist, dass die Skalierungsrate für die Gleichzeitigkeit eine Grenze auf Funktionsebene darstellt. Das bedeutet, dass jede Funktion in Ihrem Konto unabhängig von anderen Funktionen skaliert werden kann.

> **Note**
>
> In der Praxis versucht Lambda, Ihre Nebenläufigkeitsskalierungsrate kontinuierlich im Laufe der Zeit aufzufüllen, anstatt alle 10 Sekunden eine einzige Nachfüllung von 1 000 Einheiten durchzuführen.

Lambda sammelt keine ungenutzten Teile Ihrer Nebenläufigkeitsskalierungsrate an. Das bedeutet, dass Ihre Skalierungsrate zu jedem Zeitpunkt immer maximal 1 000 Nebenläufigkeitseinheiten beträgt. Wenn Sie beispielsweise in einem 10-Sekunden-Intervall keine der verfügbaren 1 000 Nebenläufigkeitseinheiten verwenden, werden Sie im nächsten 10-Sekunden-Intervall keine

1 000 zusätzlichen Einheiten ansammeln. Ihre Skalierungsrate für Nebenläufigkeit liegt im nächsten 10-Sekunden-Intervall immer noch bei 1 000.

Solange Ihre Funktion weiterhin eine steigende Anzahl von Anfragen erhält, skaliert Lambda mit der schnellsten Rate, die Ihnen zur Verfügung steht, bis zum Nebenläufigkeitslimit Ihres Kontos. Sie können das Volumen der Nebenläufigkeit einschränken, die einzelne Funktionen verwenden können, indem Sie die reservierte Nebenläufigkeit konfigurieren. Wenn Anforderungen schneller eingehen, als Ihre Funktion sie skalieren kann, oder wenn Ihre Funktion die maximale Nebenläufigkeit erreicht hat, schlagen weitere Anforderungen mit einem Drosselungsfehler (Statuscode 429) fehl.

Überwachen der Gleichzeitigkeit

Lambda gibt Amazon-CloudWatch-Metriken aus, damit Sie die Gleichzeitigkeit Ihrer Funktionen überwachen können. In diesem Thema werden diese Metriken und ihre Interpretation erläutert.

Abschnitte

- Allgemeine Metriken für Gleichzeitigkeit
- Metriken für Provisioned Concurrency
- Die Metrik ClaimedAccountConcurrency

Allgemeine Metriken für Gleichzeitigkeit

Verwenden Sie die folgenden Metriken, um die Gleichzeitigkeit Ihrer Lambda-Funktionen zu überwachen. Die Granularität für jede Metrik ist 1 Minute.

- `ConcurrentExecutions` – Die Anzahl der aktiven gleichzeitigen Aufrufe zu einem bestimmten Zeitpunkt. Lambda gibt diese Metrik für alle Funktionen, Versionen und Aliasse aus. Für jede Funktion in der Lambda-Konsole zeigt Lambda das Diagramm für `ConcurrentExecutions` nativ auf der Registerkarte Überwachung unter Metriken an. Sehen Sie sich diese Metrik mit MAX an.
- `UnreservedConcurrentExecutions` – Die Anzahl der aktiven gleichzeitigen Aufrufe, die nicht reservierte Gleichzeitigkeit verwenden. Lambda gibt diese Metriken for alle Funktionen in einer Region aus. Sehen Sie sich diese Metrik mit MAX an.
- `ClaimedAccountConcurrency` – Die Menge an Gleichzeitigkeit, die für On-Demand-Aufrufe nicht verfügbar ist. `ClaimedAccountConcurrency` entspricht `UnreservedConcurrentExecutions` plus dem Betrag der zugewiesenen Gleichzeitigkeit (d. h. der gesamten reservierten Gleichzeitigkeit plus der gesamten bereitgestellten Gleichzeitigkeit). Wenn das Kontogleichzeitigkeitslimit für `ClaimedAccountConcurrency` überschritten wird, können Sie ein höheres Kontogleichzeitigkeitslimit beantragen. Sehen Sie sich diese Metrik mit MAX an. Weitere Informationen finden Sie unter Die Metrik `ClaimedAccountConcurrency`.

Metriken für Provisioned Concurrency

Verwenden Sie die folgenden Metriken, um Lambda-Funktionen mit bereitgestellter Gleichzeitigkeit zu überwachen. Die Granularität für jede Metrik ist 1 Minute.

- `ProvisionedConcurrentExecutions` – Die Anzahl der Instances der Ausführungsumgebung, die einen Aufruf bei bereitgestellter Gleichzeitigkeit aktiv verarbeiten. Lambda gibt diese Metrik für jede Funktionsversion und jeden Alias aus, wobei die bereitgestellte Gleichzeitigkeit konfiguriert ist. Sehen Sie sich diese Metrik mit MAX an.

`ProvisionedConcurrentExecutions` entspricht nicht der Gesamtzahl der bereitgestellten Gleichzeitigkeit, die Sie zuweisen. Nehmen wir zum Beispiel an, Sie weisen einer Funktionsversion 100 Einheiten der bereitgestellten Gleichzeitigkeit zu. Wenn in einer bestimmten Minute höchstens 50 dieser 100 Ausführungsumgebungen gleichzeitig Aufrufe verarbeiten, dann ist der Wert von MAX(`ProvisionedConcurrentExecutions`) 50.

- `ProvisionedConcurrencyInvocations` – Die Häufigkeit, mit der Lambda Ihren Funktionscode mit bereitgestellter Gleichzeitigkeit aufruft. Lambda gibt diese Metrik für jede Funktionsversion und jeden Alias aus, wobei die bereitgestellte Gleichzeitigkeit konfiguriert ist. Sehen Sie sich diese Metrik mit SUM an.

`ProvisionedConcurrencyInvocations` unterscheidet sich von `ProvisionedConcurrentExecutions` dadurch, dass `ProvisionedConcurrencyInvocations` die Gesamtzahl der Aufrufe zählt, während `ProvisionedConcurrentExecutions` die Anzahl der aktiven Umgebungen zählt. Um diesen Unterschied zu verstehen, betrachten Sie das folgende Szenario:

In diesem Beispiel nehmen wir an, dass Sie 1 Aufruf pro Minute erhalten und jeder Aufruf 2 Minuten dauert. Jeder orangefarbene horizontale Balken steht für eine einzelne Anfrage. Nehmen wir an,

dass Sie dieser Funktion 10 Einheiten der bereitgestellten Gleichzeitigkeit zuweisen, sodass jede Anforderung mit bereitgestellter Gleichzeitigkeit ausgeführt wird.

Zwischen den Minuten 0 und 1 trifft `Request 1` ein. Bei Minute 1 ist der Wert für MAX(`ProvisionedConcurrentExecutions`) 1, da in der vergangenen Minute höchstens 1 Ausführungsumgebung aktiv war. Der Wert für SUM(`ProvisionedConcurrencyInvocations`) ist ebenfalls 1, da in der letzten Minute 1 neue Anforderung eingegangen ist.

Zwischen den Minuten 1 und 2 trifft `Request 2` ein, während `Request 1` weiterläuft. Bei Minute 2 ist der Wert für MAX(`ProvisionedConcurrentExecutions`) 2, da in der letzten Minute höchstens 2 Ausführungsumgebungen aktiv waren. Der Wert für SUM(`ProvisionedConcurrencyInvocations`) ist jedoch 1, da in der letzten Minute nur 1 neue Anforderung eingegangen ist. Dieses Verhalten der Metrik setzt sich bis zum Ende des Beispiels fort.

- `ProvisionedConcurrencySpilloverInvocations` – Die Häufigkeit, mit der Lambda Ihre Funktion bei standardmäßiger (reservierter oder nicht reservierter) Gleichzeitigkeit aufruft, wenn die gesamte bereitgestellte Gleichzeitigkeit verwendet wird. Lambda gibt diese Metrik für jede Funktionsversion und jeden Alias aus, wobei die bereitgestellte Gleichzeitigkeit konfiguriert ist. Sehen Sie sich diese Metrik mit SUM an. Der Wert von `ProvisionedConcurrencyInvocations` + `ProvisionedConcurrencySpilloverInvocations` sollte der Gesamtzahl der Funktionsaufrufe (d. h. der `Invocations`-Metrik) entsprechen.

  `ProvisionedConcurrencyUtilization` – Der prozentuale Anteil der bereitgestellten Gleichzeitigkeit, der genutzt wird (d. h. der Wert von `ProvisionedConcurrentExecutions` geteilt durch die Gesamtmenge der bereitgestellten Gleichzeitigkeit). Lambda gibt diese Metrik für jede Funktionsversion und jeden Alias aus, wobei die bereitgestellte Gleichzeitigkeit konfiguriert ist. Sehen Sie sich diese Metrik mit MAX an.

Nehmen wir zum Beispiel an, Sie teilen einer Funktionsversion 100 Einheiten der bereitgestellten Gleichzeitigkeit zu. Wenn in einer bestimmten Minute höchstens 60 dieser 100 Ausführungsumgebungen gleichzeitig Aufrufe verarbeiten, dann ist der Wert von MAX(`ProvisionedConcurrentExecutions`) 60 und der Wert von MAX(`ProvisionedConcurrencyUtilization`) ist 0,6.

Ein hoher Wert für `ProvisionedConcurrencySpilloverInvocations` kann darauf hinweisen, dass Sie zusätzliche bereitgestellte Gleichzeitigkeit für Ihre Funktion zuteilen müssen.

Alternativ können Sie Application Auto Scaling konfigurieren, um die automatische Skalierung der bereitgestellten Gleichzeitigkeit auf der Grundlage von vordefinierten Schwellenwerten zu steuern.

Umgekehrt können konstant niedrige Werte für `ProvisionedConcurrencyUtilization` darauf hindeuten, dass Sie Ihrer Funktion zu viel bereitgestellte Gleichzeitigkeit zugeteilt haben.

Die Metrik `ClaimedAccountConcurrency`

Lambda verwendet die Metrik `ClaimedAccountConcurrency`, um zu ermitteln, wie viel Gleichzeitigkeit in Ihrem Konto für On-Demand-Aufrufe verfügbar ist. Lambda berechnet `ClaimedAccountConcurrency` mit der folgenden Formel:

```
ClaimedAccountConcurrency = UnreservedConcurrentExecutions + (allocated concurrency)
```

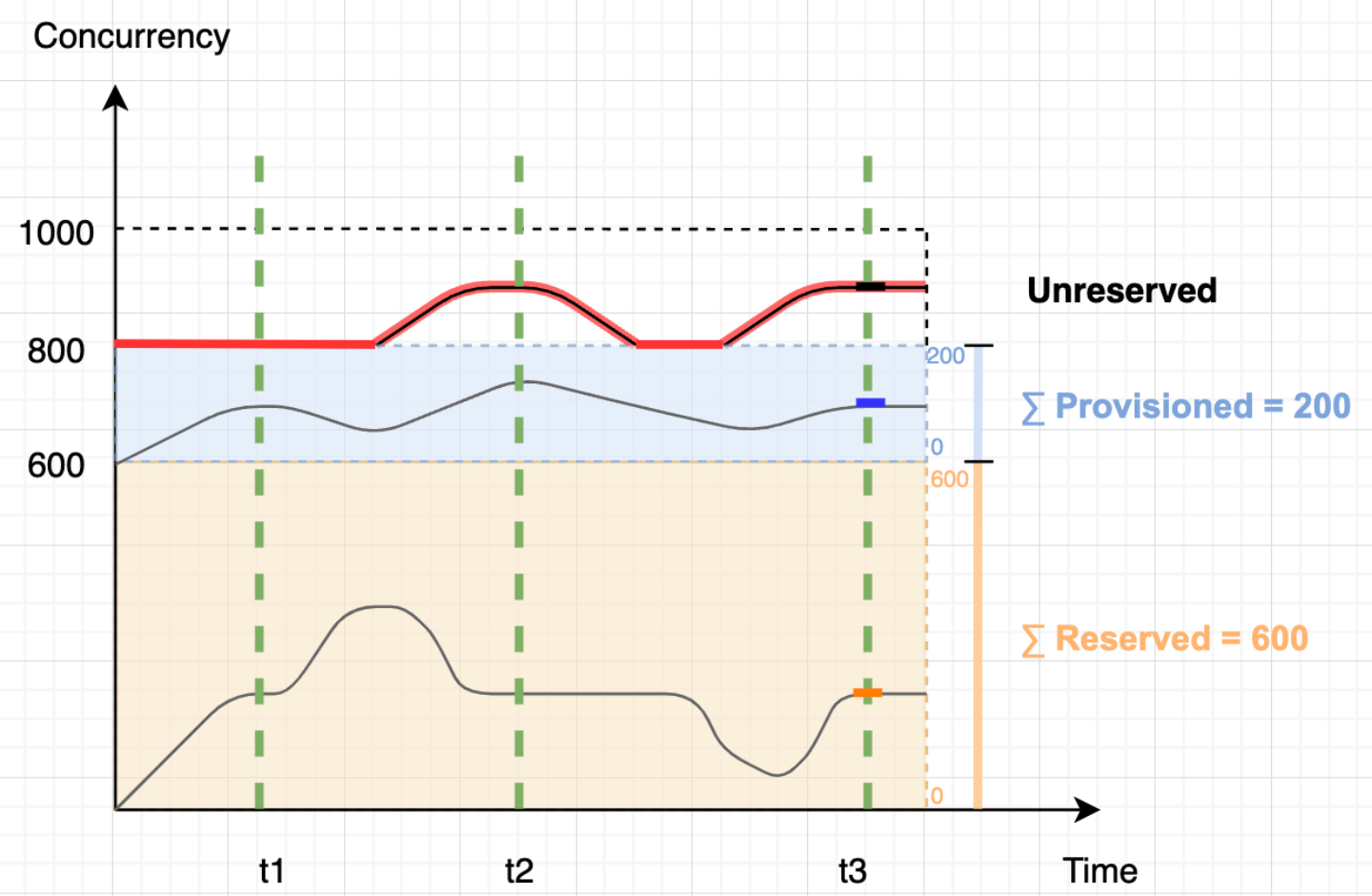
`UnreservedConcurrentExecutions` ist die Anzahl der aktiven gleichzeitigen Aufrufe, die nicht reservierte Gleichzeitigkeit verwenden. Die zugewiesene Gleichzeitigkeit ist die Summe der folgenden beiden Teile (wobei RC für „reservierte Gleichzeitigkeit“ und PC für „bereitgestellte Gleichzeitigkeit“ steht):

- Die gesamte RC aller Funktionen in einer Region.
- Die gesamte PC aller Funktionen in einer Region, die PC verwenden, ohne Funktionen, die RC verwenden.

> **Note**
>
> Sie können einer Funktion nicht mehr PC als RC zuweisen. Somit ist die RC einer Funktion immer größer oder gleich der PC. Um den Beitrag zur zugewiesenen Gleichzeitigkeit für solche Funktionen mit PC und RC zu berechnen, berücksichtigt Lambda nur RC, welches das Maximum der beiden ist.

Lambda verwendet die Metrik `ClaimedAccountConcurrency` anstelle von `ConcurrentExecutions`, um zu ermitteln, wie viel Gleichzeitigkeit für On-Demand-Aufrufe verfügbar ist. Die Metrik `ConcurrentExecutions` ist zwar nützlich, um die Anzahl der aktiven gleichzeitigen Aufrufe nachzuverfolgen, spiegelt jedoch nicht immer die tatsächliche Verfügbarkeit von Gleichzeitigkeit wider. Das liegt daran, dass Lambda bei der Bestimmung der Verfügbarkeit auch die reservierte und die bereitgestellte Gleichzeitigkeit berücksichtigt.

Stellen Sie sich zur Veranschaulichung von `ClaimedAccountConcurrency` ein Szenario vor, in dem Sie sehr viel reservierte und bereitgestellte Gleichzeitigkeit für Ihre Funktionen konfigurieren, die weitgehend ungenutzt bleiben. Gehen Sie im folgenden Beispiel nun davon aus, dass Ihr Kontogleichzeitigkeitslimit 1 000 beträgt und dass Ihr Konto über zwei Hauptfunktionen verfügt: `function-orange` und `function-blue`. Sie weisen `function-orange` 600 reservierte Gleichzeitigkeitseinheiten zu. Sie weisen `function-blue` 200 bereitgestellte Gleichzeitigkeitseinheiten zu. Nehmen wir an, dass Sie im Laufe der Zeit zusätzliche Funktionen bereitstellen und das folgende Datenverkehrsmuster beobachten:



Im vorherigen Diagramm stehen die schwarzen Linien für die tatsächliche Nutzung der Gleichzeitigkeit im Zeitverlauf, und die rote Linie für den Wert `ClaimedAccountConcurrency` im Zeitverlauf. In diesem Szenario beträgt der Wert von `ClaimedAccountConcurrency` mindestens 800, obwohl die tatsächliche Gleichzeitigkeitsauslastung aller Funktionen gering ist. Das liegt daran, dass Sie insgesamt 800 Gleichzeitigkeitseinheiten für `function-orange` und `function-blue` zugewiesen haben. Aus Sicht von Lambda haben Sie diese Gleichzeitigkeit zur Nutzung „beansprucht“, sodass Ihnen praktisch nur noch 200 Gleichzeitigkeitseinheiten für andere Funktionen zur Verfügung stehen.

Für dieses Szenario beträgt die zugewiesene Gleichzeitigkeit in der `ClaimedAccountConcurrency`-Formel 800. Daraus können wir den Wert von `ClaimedAccountConcurrency` an verschiedenen Stellen im Diagramm ableiten:

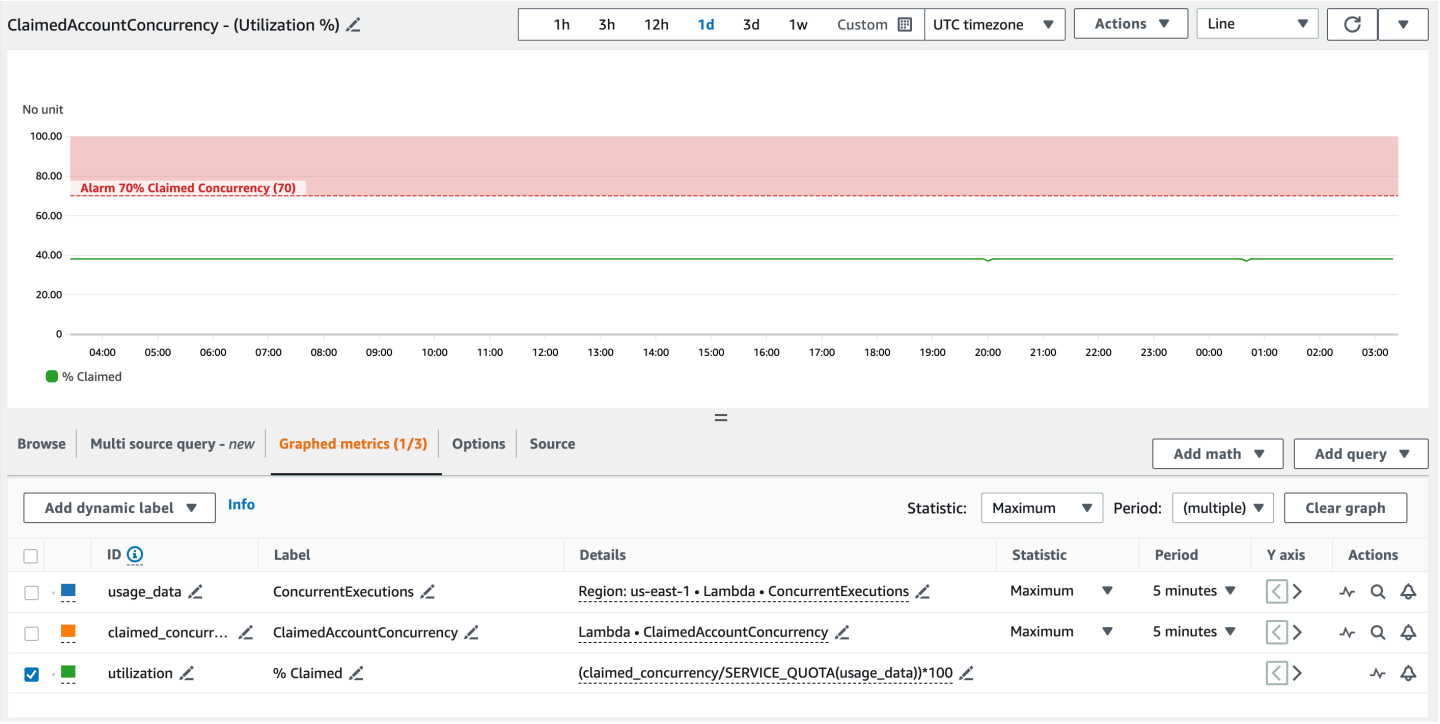
- Bei `t1`, `ClaimedAccountConcurrency` ist der Wert 800 (800 + 0 `UnreservedConcurrentExecutions`).
- Bei `t2`, `ClaimedAccountConcurrency` ist der Wert 900 (800 + 100 `UnreservedConcurrentExecutions`).
- Bei `t3`, `ClaimedAccountConcurrency` ist der Wert wieder 900 (800 + 100 `UnreservedConcurrentExecutions`).

Einrichtung der **`ClaimedAccountConcurrency`**-Metrik in CloudWatch

Lambda gibt die `ClaimedAccountConcurrency`-Metrik in CloudWatch aus. Verwenden Sie diese Metrik zusammen mit dem Wert von `SERVICE_QUOTA(ConcurrentExecutions)`, um die prozentuale Nutzung der Gleichzeitigkeit in Ihrem Konto zu ermitteln, wie in der folgenden Formel dargestellt:

```
Utilization = (ClaimedAccountConcurrency/SERVICE_QUOTA(ConcurrentExecutions)) * 100%
```

Die folgende Abbildung zeigt, wie Sie diese Formel in CloudWatch grafisch darstellen können. Die grüne `claim_utilization`-Linie steht für die Gleichzeitigkeitsauslastung in diesem Konto, die bei etwa 40 % liegt:



Die vorherige Abbildung enthält auch einen CloudWatch-Alarm, der in den Zustand `ALARM` wechselt, wenn die Gleichzeitigkeitsauslastung 70 % überschreitet. Sie können die `ClaimedAccountConcurrency`-Metrik zusammen mit ähnlichen Alarmen verwenden, um proaktiv zu bestimmen, wann Sie möglicherweise ein höheres Kontengleichzeitigkeitslimit beantragen müssen.

Erstellen von Lambda-Funktionen mit Node.js

Sie können JavaScript Code mit Node.js in AWS Lambda ausführen. Lambda bietet Laufzeiten für Node.js, die Ihren Code ausführen, um Ereignisse zu verarbeiten. Ihr Code wird in einer Umgebung ausgeführt, die die Rolle AWS SDK für JavaScript, with credentials from a AWS Identity and Access Management (IAM) enthält, die Sie verwalten. Weitere Informationen zu den SDK-Versionen, die in den Laufzeiten von Node.js enthalten sind, finden Sie unter the section called “SDK-Versionen, die zur Laufzeit enthalten sind”.

Lambda unterstützt die folgenden Node.js-Laufzeiten.

Name	ID	Betriebssystem	Datum der Veraltung	Blockfunktion erstellen	Blockfunktion aktualisieren
Node.js 24	`nodejs24.x`	Amazon Linux 2023	30. April 2028	1. Juni 2028	1. Juli 2028
Node.js 22	`nodejs22.x`	Amazon Linux 2023	30. April 2027	1. Juni 2027	1. Juli 2027
Node.js 20	`nodejs20.x`	Amazon Linux 2023	30. April 2026	31. August 2026	30. September 2026

Erstellen einer Funktion Node.js.

1. Öffnen Sie die Lambda-Konsole.
2. Wählen Sie Funktion erstellen.
3. Konfigurieren Sie die folgenden Einstellungen:
   - Funktionsname: Geben Sie einen Namen für die Funktion ein.
   - Runtime: Wählen Sie Node.js 24.x.
4. Wählen Sie Funktion erstellen.

Die Konsole erstellt eine Lambda-Funktion mit einer einzigen Quelldatei mit dem Namen `index.mjs`. Mit dem integrierten Code-Editor können Sie diese Datei bearbeiten und weitere

Dateien hinzufügen. Wählen Sie im Abschnitt BEREITSTELLEN die Option Bereitstellen aus, um den Code Ihrer Funktion zu aktualisieren. Um Ihren Code auszuführen, wählen Sie anschließend im Abschnitt TESTEREIGNISSE die Option Testereignis erstellen aus.

Die `index.mjs`-Datei exportiert eine Funktion mit dem Namen `handler`, die ein Ereignisobjekt und ein Kontext-Objekt übernimmt. Dies ist die Handler-Funktion, die bei einem Aufruf der Funktion von Lambda aufgerufen wird. Die Node.js-Funktionslaufzeit ruft Aufrufereignisse von Lambda ab und leitet sie an den Handler weiter. In der Konfiguration der Funktion lautet der Wert für den Handler `index.handler`.

Wenn Sie Ihren Funktionscode speichern, erstellt die Lambda-Konsole ein Bereitstellungspaket für das ZIP-Dateiarchiv. Wenn Sie Ihren Funktionscode außerhalb der Konsole (mit einer IDE) entwickeln, müssen Sie ein Bereitstellungspaket erstellen, um Ihren Code in die Lambda-Funktion hochzuladen.

Die Funktionslaufzeit übergibt neben dem Aufrufereignis ein Context-Objekt an den Handler. Das Context-Objekt enthält zusätzliche Informationen zum Aufruf, zur Funktion und zur Ausführungsumgebung. Weitere Informationen erhalten Sie über die Umgebungsvariablen.

Ihre Lambda-Funktion wird mit einer CloudWatch Logs-Protokollgruppe geliefert. Die Funktionslaufzeit sendet Details zu jedem Aufruf an CloudWatch Logs. Es leitet alle Protokolle weiter, die Ihre Funktion während des Aufrufs ausgibt. Wenn Ihre Funktion einen Fehler zurückgibt, formatiert Lambda den Fehler und gibt ihn an den Aufrufer zurück.

Themen

- SDK-Versionen, die zur Laufzeit enthalten sind
- Verwenden von Keepalive für TCP-Verbindungen
- CA-Zertifikat wird geladen
- Experimentelle Funktionen von Node.js
- Lambda-Funktionshandler in Node.js definieren
- Bereitstellen von Node.js Lambda-Funktionen mit ZIP-Dateiarchiven
- Bereitstellen von Node.js-Lambda-Funktionen mit Container-Images
- Arbeiten mit Ebenen für Node.js Lambda-Funktionen
- Verwenden des Lambda-Kontextobjekts zum Abrufen von Node.js-Funktionsinformationen
- Node.js-Lambda-Funktionen protokollieren und überwachen
- Instrumentierung von Node.js Code in AWS Lambda

SDK-Versionen, die zur Laufzeit enthalten sind

Alle unterstützten Lambda Node.js Runtimes enthalten eine bestimmte Nebenversion der Version AWS SDK für JavaScript 3, nicht die neueste Version. Welche Nebenversion in der Laufzeit enthalten ist, hängt von der Laufzeitversion und Ihrer AWS-Region ab. Um die spezifische Version des SDK zu finden, die in der von Ihnen verwendeten Laufzeit enthalten ist, erstellen Sie eine Lambda-Funktion mit dem folgenden Code.

Example index.mjs

```
import packageJson from '@aws-sdk/client-s3/package.json' with { type: 'json' };

export const handler = async () => ({ version: packageJson.version });
```

Dies gibt eine Antwort im folgenden Format zurück:

```
{
  "version": "3.632.0"
}
```

Weitere Informationen finden Sie unter Verwenden Sie das SDK für JavaScript v3 in Ihrem Handler.

Verwenden von Keepalive für TCP-Verbindungen

Der HTTP/HTTPS Standardagent Node.js erstellt für jede neue Anfrage eine neue TCP-Verbindung. Um die Kosten für den Aufbau neuer Verbindungen zu vermeiden, ist Keep-alive in allen unterstützten Node.js-Laufzeiten standardmäßig aktiviert. Keepalive kann die Anforderungszeiten für Lambda-Funktionen reduzieren, die mehrere API-Aufrufe unter Verwendung des SDK ausführen.

Informationen zum Deaktivieren von Keep-Alive finden Sie unter Wiederverwenden von Verbindungen mit Keep-Alive in Node.js im AWS SDK for 3.x Developer Guide. JavaScript Weitere Informationen zur Verwendung von Keep-Alive finden Sie im Developer Tools Blog unter HTTP keep-alive is default on in modular SDK. AWS JavaScript AWS

CA-Zertifikat wird geladen

Für Node.js-Laufzeitversionen bis Node.js 18 lädt Lambda automatisch Amazon-spezifische CA-Zertifikate (Certificate Authority, Zertifizierungsstelle), um Ihnen die Erstellung von Funktionen zu

erleichtern, die mit anderen AWS-Services interagieren. Lambda enthält beispielsweise die Amazon-RDS-Zertifikate, die für die Validierung des in Ihrer Amazon-RDS-Datenbank installierten Server-Identitätszertifikats erforderlich sind. Dieses Verhalten kann sich bei Kaltstarts auf die Leistung auswirken.

Ab Node.js 20 lädt Lambda standardmäßig keine zusätzlichen CA-Zertifikate mehr. Die Laufzeit Node.js 20 enthält eine Zertifikatsdatei mit allen Amazon-CA-Zertifikaten, die sich unter `/var/runtime/ca-cert.pem` befinden. Um dasselbe Verhalten aus Node.js 18 und früheren Laufzeiten wiederherzustellen, setzen Sie die Umgebungsvariable `NODE_EXTRA_CA_CERTS` auf `/var/runtime/ca-cert.pem`.

Für eine optimale Leistung empfehlen wir, nur die benötigten Zertifikate mit Ihrem Bereitstellungspaket zu bündeln und diese über die Umgebungsvariable `NODE_EXTRA_CA_CERTS` zu laden. Die Zertifikatsdatei sollte aus einem oder mehreren vertrauenswürdigen Stammzertifikaten oder Zertifikaten von Zwischenzertifizierungsstellen im PEM-Format bestehen. Fügen Sie für RDS die erforderlichen Zertifikate beispielsweise zusammen mit Ihrem Code als `certificates/rds.pem` hinzu. Laden Sie dann die Zertifikate, indem Sie `NODE_EXTRA_CA_CERTS` auf `/var/task/certificates/rds.pem` festlegen.

Experimentelle Funktionen von Node.js

Die Upstream-Sprachversionen von Node.js ermöglichen standardmäßig einige experimentelle Features. Lambda deaktiviert diese Features, um die Laufzeitstabilität und eine konsistente Leistung zu gewährleisten. In der folgenden Tabelle sind die experimentellen Features aufgeführt, die Lambda deaktiviert.

Experimentelle Features	Unterstützte Node.js-Versionen	Von Lambda angewendetes Node.js-Flag	Lambda-Flag zum erneuten Aktivieren
Support für den Import von Modulen mithilfe von „require“ in ES-Modulen	Node.js 20, Node.js 22	`--no-experimental-require-module`	`--experimental-require-module`
Support für die automatische Erkennung von ES-	Node.js 22	`--no-experimental-detect-module`	`--experimental-detect-module`

Experimentelle Features	Unterstützte Node.js-Versionen	Von Lambda angewendetes Node.js-Flag	Lambda-Flag zum erneuten Aktivieren
und CommonJS-Modulen			

Um ein deaktiviertes experimentelles Feature zu aktivieren, setzen Sie das Reaktivierungs-Flag in der Umgebungsvariablen „`NODE_OPTIONS`“. Beispiel: Um die Unterstützung für „require“ in ES-Modulen zu aktivieren, setzen Sie „`NODE_OPTIONS`“ auf „`--experimental-require-module`“. Lambda erkennt diese Überschreibung und entfernt das entsprechende Deaktivierungs-Flag.

> **Important**
>
> Die Verwendung experimenteller Features kann zu Instabilität und Leistungsproblemen führen. Diese Features könnten in zukünftigen Versionen von Node.js geändert oder entfernt werden. Funktionen, die experimentelle Features nutzen, sind nicht für das Lambda Service Level Agreement (SLA) oder AWS Support qualifiziert.

Lambda-Funktionshandler in Node.js definieren

Der Lambda-Funktionshandler ist die Methode in Ihrem Funktionscode, die Ereignisse verarbeitet. Wenn Ihre Funktion aufgerufen wird, führt Lambda die Handler-Methode aus. Ihre Funktion wird so lange ausgeführt, bis der Handler eine Antwort zurückgibt, beendet wird oder ein Timeout auftritt.

Auf dieser Seite wird beschrieben, wie Sie mit Lambda-Funktions-Handlern in Node.js arbeiten, einschließlich Optionen für die Projekteinrichtung, Benennungskonventionen und Best Practices. Diese Seite enthält auch ein Beispiel für eine Node.js-Lambda-Funktion, die Informationen über einen Auftrag aufnimmt, eine Textdatei als Quittung erstellt und diese Datei in einem Bucket von Amazon Simple Storage Service (Amazon S3) platziert. Informationen darüber, wie Sie Ihre Funktion nach dem Schreiben einsetzen können, finden Sie unter the section called “Bereitstellen von ZIP-Dateiarchiven” oder the section called “Bereitstellen von Container-Images”.

Themen

- Einrichten Ihres Node.js-Handler-Projekts
- Beispiel für Node.js-Lambda-Funktionscode
- CommonJS- und ES-Module
- Initialisierung von Node.js
- Namenskonventionen für Handler
- Definieren Sie das Eingabeereignisobjekt und greifen Sie darauf zu
- Gültige Handler-Muster für Node.js-Funktionen
- Verwenden Sie das SDK für JavaScript v3 in Ihrem Handler
- Zugriff auf Umgebungsvariablen
- Verwenden des globalen Zustands
- Bewährte Codemethoden für Node.js-Lambda-Funktionen

Einrichten Ihres Node.js-Handler-Projekts

Es gibt mehrere Möglichkeiten, ein Node.js-Lambda-Projekt zu initialisieren. Sie können beispielsweise ein Standard-Node.js-Projekt mit `npm` erstellen oder eine AWS SAM - bzw. AWS CDK -Anwendung erstellen.

So erstellen Sie das Projekt mit `npm`:

```
npm init
```

Dieser Befehl initialisiert Ihr Projekt und generiert eine `package.json`-Datei, die die Metadaten und Abhängigkeiten Ihres Projekts verwaltet.

Ihr Funktionscode befindet sich in einer `.js .mjs` JavaScript OR-Datei. Im folgenden Beispiel nennen wir diese Datei `index.mjs`, weil sie einen ES-Modul-Handler verwendet. Lambda unterstützt sowohl ES-Modul- als auch CommonJS-Handler. Weitere Informationen finden Sie unter CommonJS- und ES-Module.

Ein typisches Node.js-Lambda-Funktionsprojekt folgt dieser allgemeinen Struktur:

```
/project-root
  ### index.mjs – Contains main handler
  ### package.json – Project metadata and dependencies
  ### package-lock.json – Dependency lock file
  ### node_modules/ – Installed dependencies
```

Beispiel für Node.js-Lambda-Funktionscode

Das folgende Beispiel für einen Lambda-Funktionscode nimmt Informationen über eine Bestellung auf, erstellt eine Textdateiquittung und platziert diese Datei in einem Amazon-S3-Bucket.

Example Lambda-Funktion „index.mjs“

```
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}`);
    }
}
```

Diese `index.mjs`-Datei enthält die folgenden Abschnitte des Codes:

- `import`-Block: Verwenden Sie diesen Block, um Bibliotheken einzubinden, die Ihre Lambda-Funktion benötigt, z. B. AWS SDK-Clients.
- `const s3Client`-Deklaration: Hiermit wird ein Amazon-S3-Client außerhalb der Handler-Funktion initialisiert. Dies veranlasst Lambda, diesen Code während der Initialisierungsphase auszuführen und der Client wird für die Wiederverwendung über mehrere Aufrufe hinweg aufbewahrt.
- JSDoc Kommentarblock: Definieren Sie die Eingabe- und Ausgabetypen für Ihren Handler mithilfe von JSDoc Anmerkungen.
- `export const handler`: Dies ist die Haupt-Handler-Funktion, die Lambda aufruft. Geben Sie beim Bereitstellen Ihrer Funktion `index.handler` für die Handler-Eigenschaft an. Der Wert der Eigenschaft `Handler` besteht aus dem Dateinamen und dem Namen der exportierten Handler-Methode, getrennt durch einen Punkt.
- `uploadReceiptToS3`-Funktion: Dies ist eine Hilfsfunktion, auf die von der Haupt-Handler-Funktion verwiesen wird.

Damit diese Funktion ordnungsgemäß funktioniert, muss ihre Ausführungsrolle die `s3:PutObject`-Aktion zulassen. Stellen Sie außerdem sicher, dass Sie die `RECEIPT_BUCKET`-Umgebungsvariable definieren. Nach einem erfolgreichen Aufruf sollte der Amazon-S3-Bucket eine Empfangsdatei enthalten.

CommonJS- und ES-Module

Node.js unterstützt zwei Modulsysteme: CommonJS und ECMAScript Module (ES-Module). Lambda empfiehlt die Verwendung von ES-Modulen, da sie Await auf oberster Ebene unterstützen, wodurch asynchrone Aufgaben während der Initialisierung der Ausführungsumgebung abgeschlossen werden können.

Node.js behandelt Dateien mit einer `.cjs` Dateinamenerweiterung als CommonJS-Module, während eine `.mjs` Erweiterung ES-Module bezeichnet. Standardmäßig behandelt Node.js Dateien mit der `.js` Dateinamenerweiterung als CommonJS-Module. Sie können Node.js so konfigurieren, dass `.js` Dateien als ES-Module behandelt werden, indem Sie das `type` as `module` in der `package.json` Funktionsdatei angeben. Sie können Node.js in Lambda so konfigurieren, dass automatisch erkannt wird, ob eine `.js` Datei als CommonJS- oder als ES-Modul behandelt werden soll, indem Sie der `NODE_OPTIONS` Umgebungsvariablen das `–experimental-detect-module` Flag hinzufügen. Weitere Informationen finden Sie unter Experimentelle Funktionen von Node.js.

Die folgenden Beispiele zeigen Funktionshandler, die sowohl mit ES-Modulen als auch mit CommonJS-Modulen geschrieben wurden. Die übrigen Beispiele auf dieser Seite verwenden alle ES-Module.

ES module example

Example – ES-Modul-Handler

```
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 – CommonJS-Modul-Handler

```
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;
};
```

Initialisierung von Node.js

Node.js verwendet ein nicht blockierendes I/O Modell, das effiziente asynchrone Operationen mithilfe einer Ereignisschleife unterstützt. Wenn Node.js beispielsweise einen Netzwerkaufruf tätigt, verarbeitet die Funktion weiterhin andere Operationen, ohne eine Netzwerkantwort zu blockieren. Wenn die Netzwerkantwort empfangen wird, wird sie in die Rückrufwarteschlange gestellt. Aufgaben aus der Warteschlange werden verarbeitet, wenn die aktuelle Aufgabe abgeschlossen ist.

Lambda empfiehlt die Verwendung von Await auf oberster Ebene, damit asynchrone Aufgaben, die während der Initialisierung der Ausführungsumgebung initiiert wurden, während der Initialisierung abgeschlossen werden. Asynchrone Aufgaben, die während der Initialisierung nicht abgeschlossen werden, werden normalerweise beim ersten Funktionsaufruf ausgeführt. Dies kann zu unerwartetem Verhalten oder Fehlern führen. Beispielsweise kann Ihre Funktionsinitialisierung einen Netzwerkaufruf ausführen, um einen Parameter aus dem AWS Parameterspeicher abzurufen. Wenn diese Aufgabe während der Initialisierung nicht abgeschlossen wird, kann der Wert während eines Aufrufs Null sein. Es kann auch zu einer Verzögerung zwischen der Initialisierung und dem Aufruf kommen, was bei zeitkritischen Vorgängen zu Fehlern führen kann. Insbesondere AWS Serviceabrufe können sich auf zeitkritische Anforderungssignaturen stützen, was dazu führt, dass Serviceabrufe fehlschlagen, wenn der Anruf während der Initialisierungsphase nicht abgeschlossen wird. Das Abschließen von Aufgaben während der Initialisierung verbessert in der Regel die Kaltstartleistung und die Leistung beim ersten Aufrufen, wenn Provisioned Concurrency verwendet wird. Weitere Informationen finden Sie in unserem Blogbeitrag Using Node.js ES modules and top level await in. AWS Lambda

Namenskonventionen für Handler

Wenn Sie eine Funktion konfigurieren, besteht der Wert der Handler-Einstellung aus dem Dateinamen und dem Namen der exportierten Handler-Methode, getrennt durch einen Punkt. Der Standardwert für Funktionen, die in der Konsole erstellt werden und für die Beispiele in diesem Handbuch ist `index.handler`. Dies deutet auf die `handler`-Methode hin, die aus der `index.js`- oder `index.mjs`-Datei wurde.

Wenn Sie eine Funktion in der Konsole mit einem anderen Dateinamen oder Funktionshandlernamen erstellen, müssen Sie den Standardhandlernamen bearbeiten.

So ändern Sie den Funktionshandlernamen (Konsole)

1. Öffnen Sie die Seite Functions (Funktionen) der Lambda-Konsole und wählen Sie eine Funktion aus.
2. Wählen Sie die Registerkarte Code (Code).

3. Scrollen Sie nach unten zum Bereich Laufzeiteinstellungen und wählen Sie Bearbeiten.
4. Geben Sie unter Handler den neuen Namen für Ihren Funktionshandler ein.
5. Wählen Sie Speichern.

Definieren Sie das Eingabeereignisobjekt und greifen Sie darauf zu

JSON ist das gebräuchlichste und standardmäßigste Eingabeformat für Lambda-Funktionen. In diesem Beispiel erwartet die Funktion eine Eingabe ähnlich der folgenden:

```
{
    "order_id": "12345",
    "amount": 199.99,
    "item": "Wireless Headphones"
}
```

Wenn Sie mit Lambda-Funktionen in Node.js arbeiten, können Sie die erwartete Form des Eingabeereignisses mithilfe von JSDoc Anmerkungen definieren. In diesem Beispiel definieren wir die Eingabestruktur im Kommentar des Handlers JSDoc :

```
/**
 * 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
 */
```

Nachdem Sie diese Typen in Ihrem JSDoc Kommentar definiert haben, können Sie direkt in Ihrem Code auf die Felder des Event-Objekts zugreifen. Zum Beispiel holt `event.order_id` den Wert von `order_id` aus der ursprünglichen Eingabe.

Gültige Handler-Muster für Node.js-Funktionen

Wir empfehlen, dass Sie async/await verwenden, um den Funktionshandler zu deklarieren, anstatt Callbacks zu verwenden. Async/await ist eine präzise und lesbare Methode, asynchronen Code zu schreiben, ohne dass verschachtelte Callbacks oder verkettete Zusagen erforderlich sind. Mit Async/Await können Sie Code schreiben, der sich wie synchroner Code liest, aber dennoch asynchron und blockierungsfrei ist.

Handler für asynchrone Funktionen (empfohlen)

Das `async`-Schlüsselwort kennzeichnet eine Funktion als asynchron, und das `await`-Schlüsselwort unterbricht die Ausführung der Funktion, bis `Promise` aufgelöst ist. Der Handler akzeptiert die folgenden Argumente:

- `event`: Enthält die an Ihre Funktion übergebenen Eingabedaten.
- `context`: Enthält Informationen über den Aufruf, die Funktion und die Ausführungsumgebung. Weitere Informationen finden Sie unter Verwenden des Lambda-Kontextobjekts zum Abrufen von Node.js-Funktionsinformationen.

Hier sind die gültigen Signaturen für das async/await Muster:

```
export const handler = async (event) => { };
```

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

Handler für synchrone Funktionen

Wenn Ihre Funktion keine asynchronen Aufgaben ausführt, können Sie einen synchronen Funktionshandler verwenden, der eine der folgenden Funktionssignaturen verwendet:

```
export const handler = (event) => { };
```

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

Handler für Antwort-Streaming-Funktionen

Lambda unterstützt Response-Streaming mit Node.js. Handler für Antwort-Streaming-Funktionen verwenden den `awslambda.streamifyResponse()` Decorator und verwenden 3 Parameter:`event`, `responseStream` und. `context` Die Funktionssignatur lautet:

```
export const handler = awslambda.streamifyResponse(async (event, responseStream,
 context) => { });
```

Weitere Informationen finden Sie unter Antwortstreaming für Lambda-Funktionen.

Callback-basierte Funktionshandler

> **Note**
>
> Callback-basierte Funktionshandler werden nur bis zu Node.js 22 unterstützt. Ab Node.js 24 sollten asynchrone Aufgaben mithilfe von asynchronen Funktionshandlern implementiert werden.

Callback-basierte Funktionshandler müssen die Event-, Kontext- und Callback-Argumente verwenden. Beispiel:

```
export const handler = (event, context, callback) => { };
```

Die Callback-Funktion erwartet `Error` und eine Antwort, die JSON-serialisierbar sein muss. Die Funktion wird so lange ausgeführt, bis die Ereignisschleife leer ist oder ein Timeout auftritt. Die Antwort wird erst an den Aufrufer gesendet, wenn alle Ereignisschleifenaufgaben abgeschlossen sind. Wenn eine Zeitüberschreitung der Funktion auftritt, wird stattdessen ein Fehler zurückgegeben. Sie können die Laufzeit so konfigurieren, dass die Antwort sofort gesendet wird, indem Sie den Kontext festlegen. callbackWaitsForEmptyEventLoopauf „Falsch“ setzen.

Example – HTTP-Anforderung mit callback

Die folgende Beispielfunktion prüft eine URL und gibt den Statuscode an den Aufrufer zurück.

```
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));
  });
};
```

Verwenden Sie das SDK für JavaScript v3 in Ihrem Handler

Oft verwenden Sie Lambda-Funktionen, um mit anderen AWS Ressourcen zu interagieren oder diese zu aktualisieren. Die einfachste Art, eine Schnittstelle zu diesen Ressourcen herzustellen,

ist die Verwendung von AWS SDK für JavaScript. Alle unterstützten Lambda Node.js Runtimes enthalten das SDK für JavaScript Version 3. Wir empfehlen jedoch dringend, dass Sie die benötigten AWS SDK-Clients in Ihr Bereitstellungspaket aufnehmen. Dadurch wird die Abwärtskompatibilität bei zukünftigen Lambda-Laufzeit-Updates maximiert. Verlassen Sie sich nur dann auf das von der Laufzeit bereitgestellte SDK, wenn Sie keine zusätzlichen Pakete hinzufügen können (z. B. wenn Sie den Code-Editor der Lambda-Konsole oder Inline-Code in einer AWS CloudFormation Vorlage verwenden).

Um Ihrer Funktion SDK-Abhängigkeiten hinzuzufügen, verwenden Sie den `npm install`-Befehl für die spezifischen SDK-Clients, die Sie benötigen. Im Beispielcode haben wir den Amazon-S3-Client verwendet. Fügen Sie diese Abhängigkeit hinzu, indem Sie den folgenden Befehl in dem Verzeichnis ausführen, das Ihre `package.json`-Datei enthält:

```
npm install @aws-sdk/client-s3
```

Importieren Sie im Funktionscode den Client und die Befehle, die Sie benötigen, wie in der Beispielfunktion gezeigt:

```
import { S3Client, PutObjectCommand } from '@aws-sdk/client-s3';
```

Initialisieren Sie dann einen Amazon-S3-Client:

```
const s3Client = new S3Client();
```

In diesem Beispiel haben wir unseren Amazon-S3-Client außerhalb der Haupt-Handler-Funktion initialisiert, um zu vermeiden, dass wir ihn bei jedem Aufruf unserer Funktion initialisieren müssen. Nachdem Sie Ihren SDK-Client initialisiert haben, können Sie ihn verwenden, um API-Aufrufe für diesen Dienst zu tätigen. AWS Der Beispielcode ruft die Amazon S3 PutObjectS3-API-Aktion wie folgt auf:

```
const command = new PutObjectCommand({
    Bucket: bucketName,
    Key: key,
    Body: receiptContent
});
```

Zugriff auf Umgebungsvariablen

In Ihrem Handler-Code können Sie mithilfe von `process.env` auf beliebige Umgebungsvariablen verweisen. In diesem Beispiel verweisen wir mit den folgenden Codezeilen auf die definierte `RECEIPT_BUCKET`-Umgebungsvariable:

```
// Access environment variables
const bucketName = process.env.RECEIPT_BUCKET;
if (!bucketName) {
    throw new Error('RECEIPT_BUCKET environment variable is not set');
}
```

Verwenden des globalen Zustands

Lambda führt Ihren statischen Code während der Initialisierungsphase aus, bevor Ihre Funktion zum ersten Mal aufgerufen wird. Ressourcen, die während der Initialisierung erstellt werden, bleiben zwischen Aufrufen im Speicher, sodass Sie sie nicht bei jedem Aufruf Ihrer Funktion neu erstellen müssen.

Im Beispielcode befindet sich der Initialisierungscode des S3-Clients außerhalb des Handlers. Die Laufzeit initialisiert den Client, bevor die Funktion ihr erstes Ereignis ausführt, und der Client bleibt für die Wiederverwendung bei allen Aufrufen verfügbar.

Bewährte Codemethoden für Node.js-Lambda-Funktionen

Befolgen Sie diese Richtlinien, wenn Sie Lambda-Funktionen erstellen:

- Trennen Sie den Lambda-Handler von Ihrer Core-Logik. Auf diese Weise können Sie eine Funktion zur besseren Prüfbarkeit von Einheiten schaffen.
- Kontrollieren Sie die Abhängigkeiten im Bereitstellungspaket Ihrer Funktion. Die AWS Lambda Ausführungsumgebung enthält eine Reihe von Bibliotheken. Für die Laufzeiten Node.js und Python gehören dazu die AWS SDKs. Um die neuesten Funktionen und Sicherheitsupdates zu aktivieren, wird Lambda diese Bibliotheken regelmäßig aktualisieren. Diese Updates können das Verhalten Ihrer Lambda-Funktion geringfügig verändern. Um die Abhängigkeiten, die Ihre Funktion verwendet, vollständig zu kontrollieren, empfehlen wir, alle Abhängigkeiten mit Ihrem Bereitstellungspaket zu bündeln.
- Minimieren Sie die Komplexität Ihrer Abhängigkeiten. Ziehen Sie einfachere Frameworks vor, die sich schnell beim Start der Ausführungsumgebung laden lassen.

- Minimieren Sie die Größe Ihres Bereitstellungspakets auf die für die Laufzeit erforderliche Größe. Dadurch verkürzt sich die Zeit, die für das Herunterladen und Entpacken Ihres Bereitstellungspakets vor dem Aufruf benötigt wird.

Nutzen Sie die Wiederverwendung der Ausführungsumgebung zur Verbesserung Ihrer Funktion. Initialisieren Sie SDK-Clients und Datenbankverbindungen außerhalb des Funktions-Handlers und speichern Sie statische Komponenten lokal im `/tmp`-Verzeichnis. Nachfolgende Aufrufe, die von derselben Instance Ihrer Funktion verarbeitet werden, können diese Ressourcen wiederverwenden. Dies spart Kosten durch Reduzierung der Funktionslaufzeit.

Um potenzielle Datenlecks über Aufrufe hinweg zu vermeiden, verwenden Sie die Ausführungsumgebung nicht, um Benutzerdaten, Ereignisse oder andere Informationen mit Sicherheitsauswirkungen zu speichern. Wenn Ihre Funktion auf einem veränderbaren Zustand beruht, der nicht im Speicher innerhalb des Handlers gespeichert werden kann, sollten Sie für jeden Benutzer eine separate Funktion oder separate Versionen einer Funktion erstellen.

Verwenden Sie eine Keep-Alive-Direktive, um dauerhafte Verbindungen zu pflegen. Lambda bereinigt Leerlaufverbindungen im Laufe der Zeit. Der Versuch, eine Leerlaufverbindung beim Aufruf einer Funktion wiederzuverwenden, führt zu einem Verbindungsfehler. Um Ihre persistente Verbindung aufrechtzuerhalten, verwenden Sie die Keep-Alive-Direktive, die Ihrer Laufzeit zugeordnet ist. Ein Beispiel finden Sie unter Wiederverwenden von Verbindungen mit Keep-Alive in Node.js.

Verwenden Sie Umgebungsvariablen um Betriebsparameter an Ihre Funktion zu übergeben. Wenn Sie z. B. Daten in einen Amazon-S3-Bucket schreiben, anstatt den Bucket-Namen, in den Sie schreiben, hartzucodieren, konfigurieren Sie den Bucket-Namen als Umgebungsvariable.

Vermeiden Sie rekursive Aufrufe in Ihrer Lambda-Funktion, bei denen die Funktion sich selbst aufruft oder einen Prozess initiiert, der die Funktion erneut aufrufen kann. Dies kann zu unvorhergesehenen Mengen an Funktionsaufrufen führen und höhere Kosten zur Folge haben. Wenn Sie eine unbeabsichtigte Menge von Aufrufen feststellen, legen Sie die reservierte gleichzeitige Ausführung der Funktion auf `0` fest, um sofort alle Aufrufe der Funktion zu drosseln, während Sie den Code aktualisieren.

Verwenden Sie APIs in Ihrem Lambda-Funktionscode nicht undokumentiert, nicht öffentlich. Für AWS Lambda verwaltete Laufzeiten führt Lambda regelmäßig Sicherheits- und Funktionsupdates für interne Lambda-Laufzeiten durch. APIs Diese internen API-Updates können abwärtsinkompatibel sein, was zu unbeabsichtigten Folgen wie Aufruffehlern führen kann, wenn Ihre Funktion von diesen

nicht öffentlichen Daten abhängig ist. APIs Eine Liste der öffentlich verfügbaren Programme finden Sie in der API-Referenz. APIs

Schreiben Sie idempotenten Code. Das Schreiben idempotenter Code für Ihre Funktionen stellt sicher, dass doppelte Ereignisse auf die gleiche Weise behandelt werden. Ihr Code sollte Ereignisse ordnungsgemäß validieren und doppelte Ereignisse ordnungsgemäß behandeln. Weitere Informationen finden Sie unter Wie mache ich meine Lambda-Funktion idempotent?.

Bereitstellen von Node.js Lambda-Funktionen mit ZIP-Dateiarchiven

Der Code Ihrer AWS Lambda Funktion besteht aus einer .js- oder .mjs-Datei, die den Handlercode Ihrer Funktion enthält, zusammen mit allen zusätzlichen Paketen und Modulen, von denen Ihr Code abhängt. Sie verwenden ein Bereitstellungspaket, um Ihren Funktionscode in Lambda bereitzustellen. Dieses Paket kann entweder ein ZIP-Dateiarchiv oder ein Container-Image sein. Weitere Informationen zur Verwendung von Container-Images mit Node.js finden Sie unter Bereitstellen von Node.js-Lambda-Funktionen mit Container-Images.

Zum Erstellen des Bereitstellungspakets für ein ZIP-Dateiarchiv können Sie ein integriertes Dienstprogramm für ZIP-Dateien Ihres Befehlszeilen-Tools oder ein anderes Dienstprogramm für ZIP-Dateien verwenden, wie 7zip. In den Beispiele in den folgenden Abschnitten wird davon ausgegangen, dass Sie ein `zip`-Befehlszeilen-Tool in einer Linux- oder MacOS-Umgebung verwenden. Unter Windows können Sie das Windows-Subsystem für Linux installieren, um eine Windows-Version von Ubuntu und Bash zu erhalten und dieselben Befehle zu verwenden.

Da Lambda POSIX-Dateiberechtigungen verwendet, müssen Sie möglicherweise Berechtigungen für den Bereitstellungspaketordner festlegen, bevor Sie das ZIP-Dateiarchiv erstellen.

Themen

- Laufzeitabhängigkeiten in Node.js
- ZIP-Bereitstellungspakets ohne Abhängigkeiten erstellen
- ZIP-Bereitstellungspakets mit Abhängigkeiten erstellen
- Erstellen einer Node.js-Ebene für Ihre Abhängigkeiten
- Suchpfad für Abhängigkeiten und integrierte Laufzeit-Bibliotheken
- Erstellen und Aktualisieren von Node.js-Lambda-Funktionen mithilfe von ZIP-Dateien

Laufzeitabhängigkeiten in Node.js

Für Lambda-Funktionen, die die Node.js-Laufzeit verwenden, kann eine Abhängigkeit ein beliebiges Node.js-Modul sein. Die Laufzeitumgebung von Node.js enthält eine Reihe gängiger Bibliotheken sowie eine Version des AWS SDK für JavaScript. Alle unterstützten Node.js-Laufzeiten beinhalten Version 3 des SDK. Um Version 2 des SDK zu verwenden, müssen Sie das SDK zu Ihrem ZIP-Dateibereitstellungspaket hinzufügen. Die spezifische Version des SDK, die in der von Ihnen

verwendeten Laufzeit enthalten ist, finden Sie unter the section called “SDK-Versionen, die zur Laufzeit enthalten sind”.

Lambda aktualisiert die SDK-Bibliotheken in der Node.js-Laufzeit regelmäßig, um die aktuellen Features und Sicherheitsupgrades aufzunehmen. Lambda wendet auch Sicherheitspatches und Updates auf die anderen in der Laufzeit enthaltenen Bibliotheken an. Um die volle Kontrolle über die Abhängigkeiten in Ihrem Paket zu haben, können Sie Ihrem Bereitstellungspaket Ihre bevorzugte Version einer beliebigen in die Laufzeit eingeschlossenen Abhängigkeit hinzufügen. Wenn Sie beispielsweise eine bestimmte Version des SDK verwenden möchten JavaScript, können Sie sie als Abhängigkeit in Ihre ZIP-Datei aufnehmen. Weitere Informationen zum Hinzufügen von in die Laufzeit eingeschlossene Abhängigkeiten zu Ihrer ZIP-Datei finden Sie unter Suchpfad für Abhängigkeiten und integrierte Laufzeit-Bibliotheken.

Im Rahmen des AWS -Modells der geteilten Verantwortung sind Sie für die Verwaltung aller Abhängigkeiten in den Bereitstellungspaketen Ihrer Funktionen verantwortlich. Dies beinhaltet das Durchführen von Updates und Sicherheitspatches. Zum Aktualisieren von Abhängigkeiten im Bereitstellungspaket Ihrer Funktion erstellen Sie zunächst eine neue ZIP-Datei und laden Sie diese dann in Lambda hoch. Weitere Informationen finden Sie unter ZIP-Bereitstellungspakets mit Abhängigkeiten erstellen und Erstellen und Aktualisieren von Node.js-Lambda-Funktionen mithilfe von ZIP-Dateien.

ZIP-Bereitstellungspakets ohne Abhängigkeiten erstellen

Wenn Ihr Funktionscode neben den in der Lambda-Laufzeit eingeschlossenen Bibliotheken keine Abhängigkeiten hat, enthält Ihre ZIP-Datei nur die `index.js`- oder `index.mjs`-Datei mit dem Handler-Code Ihrer Funktion. Erstellen Sie mit Ihrem bevorzugten ZIP-Programm eine ZIP-Datei mit Ihrer `index.js`- oder `index.mjs`-Datei im Stammverzeichnis. Wenn sich die Datei mit Ihrem Handler-Code nicht im Stammverzeichnis Ihrer ZIP-Datei, kann Lambda Ihren Code nicht ausführen.

Informationen zum Bereitstellen Ihrer ZIP-Datei zum Erstellen einer neuen Lambda-Funktion oder Aktualisieren einer vorhandenen Funktion, finden Sie unter Erstellen und Aktualisieren von Node.js-Lambda-Funktionen mithilfe von ZIP-Dateien.

ZIP-Bereitstellungspakets mit Abhängigkeiten erstellen

Wenn Ihr Funktionscode von Paketen oder Modulen abhängt, die nicht in der Lambda-Node.js-Laufzeit enthalten sind, können Sie diese Abhängigkeiten entweder mit Ihrem Funktionscode zu Ihrer ZIP-Datei hinzufügen oder eine Lambda-Ebene verwenden. Die Anweisungen in diesem Abschnitt

zeigen Ihnen, wie Sie Ihre Abhängigkeiten in Ihr ZIP-Bereitstellungspaket aufnehmen. Anweisungen zum Einschließen Ihrer Abhängigkeiten in eine Ebene finden Sie unter the section called “Erstellen einer Node.js-Ebene für Ihre Abhängigkeiten”.

Die folgenden CLI-Befehle erstellen Sie eine ZIP-Datei mit dem Namen `my_deployment_package.zip`, welche die `index.js`- oder `index.mjs`-Datei mit dem Handler-Code Ihrer Funktion und die zugehörigen Abhängigkeiten enthält. Im Beispiel installieren Sie Abhängigkeiten mit dem npm-Paketmanager.

Erstellen des Bereitstellungspakets

1. Navigieren Sie zum Projektverzeichnis, das Ihre Quellcodedatei `index.js` oder `index.mjs` enthält. In diesem Beispiel trägt das Verzeichnis den Namen `my_function`.

   ```
   cd my_function
   ```

2. Installieren Sie die erforderlichen Bibliotheken Ihrer Funktion im Verzeichnis `node_modules` mit dem Befehl `npm install`. In diesem Beispiel installieren Sie das AWS X-Ray-SDK für Node.js.

   ```
   npm install aws-xray-sdk
   ```

   Dabei wird eine Ordnerstruktur erstellt, die etwa wie folgt aussieht:

   ```
   ~/my_function
   ### index.mjs
   ### node_modules
       ### async
       ### async-listener
       ### atomic-batcher
       ### aws-sdk
       ### aws-xray-sdk
       ### aws-xray-sdk-core
   ```

   Sie können Ihrem Bereitstellungspaket auch benutzerdefinierte Module hinzufügen, die Sie selbst erstellen. Erstellen Sie ein Verzeichnis unter `node_modules` mit dem Namen Ihres Moduls und speichern Sie Ihre benutzerdefinierten geschriebenen Pakete dort.

3. Erstellen Sie im Stammverzeichnis eine ZIP-Datei mit den Inhalten Ihres Projektordners. Benutzen Sie die `r` (rekursive) Option, um sicherzustellen, dass der zip-Befehl die Unterordner komprimiert.

```
zip -r my_deployment_package.zip .
```

Erstellen einer Node.js-Ebene für Ihre Abhängigkeiten

Die Anweisungen in diesem Abschnitt zeigen Ihnen, wie Sie Ihre Abhängigkeiten in eine Ebene einschließen. Anweisungen zum Einschließen Ihrer Abhängigkeiten in Ihr Bereitstellungspaket finden Sie unter the section called “ZIP-Bereitstellungspakets mit Abhängigkeiten erstellen”.

Wenn Sie einer Funktion eine Ebene hinzufügen, lädt Lambda den Ebeneninhalt in das Verzeichnis `/opt` der Ausführungsumgebung. Für jede Lambda-Laufzeit enthält die Variable `PATH` bereits spezifische Ordnerpfade innerhalb des Verzeichnisses `/opt`. Um sicherzustellen, dass Lambda den Inhalt Ihrer Ebene aufnimmt, müssen die Abhängigkeiten Ihrer ZIP-Datei in einem der folgenden Ordnerpfade enthalten sein:

- `nodejs/node_modules`
- `nodejs/node18/node_modules (NODE_PATH)`
- `nodejs/node20/node_modules (NODE_PATH)`
- `nodejs/node22/node_modules (NODE_PATH)`

Die Struktur Ihrer Ebene-ZIP-Datei könnte beispielsweise wie folgt aussehen:

```
xray-sdk.zip
# nodejs/node_modules/aws-xray-sdk
```

Darüber hinaus erkennt Lambda automatisch alle Bibliotheken im `/opt/lib`-Verzeichnis und alle Binärdateien im `/opt/bin`-Verzeichnis. Um sicherzustellen, dass Lambda Ihren Ebeneninhalt korrekt findet, können Sie auch eine Ebene mit der folgenden Struktur erstellen:

```
custom-layer.zip
# lib
    | lib_1
    | lib_2
# bin
    | bin_1
    | bin_2
```

Nachdem Sie Ihre Ebene gebündelt haben, sehen Sie sich the section called “Erstellen und Löschen von Ebenen” und the section called “Hinzufügen von Ebenen” an, um die Einrichtung Ihrer Ebene abzuschließen.

Suchpfad für Abhängigkeiten und integrierte Laufzeit-Bibliotheken

Die Laufzeitumgebung von Node.js enthält eine Reihe gängiger Bibliotheken sowie eine Version des AWS SDK für JavaScript. Wenn Sie eine andere Version einer in die Laufzeit eingeschlossenen Bibliothek verwenden möchten, können Sie sie mit Ihrer Funktion bündeln oder sie als Abhängigkeit zu Ihrem Bereitstellungspaket hinzufügen. Sie können beispielsweise eine andere Version des SDK verwenden, indem Sie es Ihrem ZIP-Bereitstellungspaket hinzufügen. Sie können sie auch in eine Lambda-Ebene für Ihre Funktion aufnehmen.

Wenn Sie in Ihrem Code eine `import`- oder `require`-Anweisung verwenden, durchsucht die Node.js-Laufzeit die Verzeichnisse im `NODE_PATH`-Pfad, bis das Modul gefunden wird. Standardmäßig ist der erste Speicherort, den die Laufzeit durchsucht, das Verzeichnis, in das Ihr ZIP-Bereitstellungspaket entpackt und bereitgestellt wird (`/var/task`). Nehmen Sie eine Version einer in der Laufzeit integrierten Bibliothek in Ihr Bereitstellungspaket auf, hat diese Version Vorrang vor der in der Laufzeit enthaltenen. Abhängigkeiten in Ihrem Bereitstellungspaket haben ebenfalls Vorrang vor Abhängigkeiten in Ebenen.

Wenn Sie einer Ebene eine Abhängigkeit hinzufügen, extrahiert Lambda diese in `/opt/nodejs/nodexx/node_modules`, wobei `nodexx` die von Ihnen verwendete Version der Laufzeitumgebung ist. Im Suchpfad hat dieses Verzeichnis Vorrang vor dem Verzeichnis mit den zur Laufzeit enthaltenen Bibliotheken (`/var/lang/lib/node_modules`). Bibliotheken in Funktionsebenen haben daher Vorrang vor Versionen, die in der Laufzeit enthalten sind.

Sie können den vollständigen Suchpfad für Ihre Lambda-Funktion sehen, indem Sie die folgende Codezeile hinzufügen:

```
console.log(process.env.NODE_PATH)
```

Fügen Sie Abhängigkeiten alternativ auch in einem separaten Ordner in Ihrem ZIP-Paket hinzu. Fügen Sie beispielsweise eine benutzerdefinierte Version zu einem Ordner in Ihrem ZIP-Paket mit dem Namen `common` hinzu. Wird Ihr ZIP-Paket entpackt und bereitgestellt, wird dieser Ordner im Verzeichnis „`/var/task`“ abgelegt. Um eine Abhängigkeit von einem Ordner in Ihrem ZIP-Bereitstellungspaket in Ihrem Code zu verwenden, verwenden Sie eine `import { } from`- oder `const { } = require()`-Anweisung, je nachdem, ob Sie die CJS- oder die ESM-Modulauflösung verwenden. Beispiel:

```
import { myModule } from './common'
```

Wenn Sie Ihren Code mit `esbuild`, `rollup` oder ähnlichem bündeln, werden die von Ihrer Funktion verwendeten Abhängigkeiten in einer oder mehreren Dateien gebündelt. Wir empfehlen, diese Methode zu verwenden, um Abhängigkeiten wann immer möglich zu vergeben. Im Vergleich zum Hinzufügen von Abhängigkeiten zu Ihrem Bereitstellungspaket führt das Bündeln Ihres Codes aufgrund der Reduzierung der Betriebsabläufe zu einer I/O Leistungssteigerung.

Erstellen und Aktualisieren von Node.js-Lambda-Funktionen mithilfe von ZIP-Dateien

Nach der Erstellung Ihres ZIP-Bereitstellungspakets können Sie es verwenden, um eine neue Lambda-Funktion zu erstellen oder eine vorhandene zu aktualisieren. Sie können Ihr .zip-Paket mithilfe der Lambda-Konsole, der und der AWS Command Line Interface Lambda-API bereitstellen. Sie können Lambda-Funktionen auch mit AWS Serverless Application Model (AWS SAM) und CloudFormation erstellen und aktualisieren.

Die maximale Größe eines ZIP-Bereitstellungspakets für Lambda beträgt 250 MB (entpackt). Beachten Sie, dass dieser Grenzwert für die kombinierte Größe aller hochgeladenen Dateien gilt, einschließlich aller Lambda-Ebenen.

Die Lambda-Laufzeit benötigt die Berechtigung zum Lesen der Dateien in Ihrem Bereitstellungspaket. In der oktalen Schreibweise von Linux-Berechtigungen benötigt Lambda 644 Berechtigungen für nicht ausführbare Dateien (rw-r--r--) und 755 Berechtigungen () für Verzeichnisse und ausführbare Dateien. rwxr-xr-x

Verwenden Sie unter Linux und MacOS den `chmod`-Befehl, um Dateiberechtigungen für Dateien und Verzeichnisse in Ihrem Bereitstellungspaket zu ändern. Um beispielsweise einer nicht ausführbaren Datei die richtigen Berechtigungen zu geben, führen Sie den folgenden Befehl aus.

```
chmod 644 <filepath>
```

Informationen zum Ändern von Dateiberechtigungen in Windows finden Sie unter Festlegen, Anzeigen, Ändern oder Entfernen von Berechtigungen für ein Objekt in der Microsoft-Windows-Dokumentation.

> **Note**
>
> Wenn Sie Lambda nicht die Berechtigungen gewähren, die es für den Zugriff auf Verzeichnisse in Ihrem Bereitstellungspaket benötigt, setzt Lambda die Berechtigungen für diese Verzeichnisse auf 755 (). rwxr-xr-x

Erstellen und Aktualisieren von Funktionen mit ZIP-Dateien unter Verwendung der Konsole

Eine neue Funktion müssen Sie zuerst in der Konsole erstellen und dann Ihr ZIP-Archiv hochladen. Zum Aktualisieren einer bestehenden Funktion öffnen Sie die Seite für Ihre Funktion und gehen dann genauso vor, um Ihre aktualisierte ZIP-Datei hinzuzufügen.

Bei einer ZIP-Datei mit unter 50 MB können Sie eine Funktion erstellen oder aktualisieren, indem Sie die Datei direkt von Ihrem lokalen Computer hochladen. Bei ZIP-Dateien mit einer Größe von mehr als 50 MB müssen Sie Ihr Paket zuerst in einen Amazon-S3-Bucket hochladen. Anweisungen zum Hochladen einer Datei in einen Amazon S3-Bucket mithilfe von finden Sie unter Erste Schritte mit Amazon S3. AWS-Managementkonsole Informationen zum Hochladen von Dateien mit dem AWS CLI finden Sie unter Objekte verschieben im AWS CLI Benutzerhandbuch.

> **Note**
>
> Sie können den Bereitstellungspakettyp (.zip oder Container-Image) für eine vorhandene Funktion nicht ändern. Sie können zum Beispiel eine Container-Image-Funktion nicht so umwandeln, dass sie ein .zip-Archiv verwendet. Sie müssen eine neue Funktion erstellen.

So erstellen Sie eine neue Funktion (Konsole)

1. Öffnen Sie die Funktionsseite der Lambda-Konsole und wählen Sie Funktion erstellen aus.
2. Wählen Sie Author from scratch aus.
3. Führen Sie unter Basic information (Grundlegende Informationen) die folgenden Schritte aus:
   a. Geben Sie als Funktionsname den Namen Ihrer Funktion ein.
   b. Wählen Sie für Laufzeit die Laufzeit aus, die Sie verwenden möchten.

   c. (Optional) Für Architektur wählen Sie die Befehlssatz-Architektur für Ihre Funktion aus. Die Standardarchitektur ist x86_64. Stellen Sie sicher, dass das ZIP-Bereitstellungspaket für Ihre Funktion mit der von Ihnen gewählten Befehlssatzarchitektur kompatibel ist.

4. (Optional) Erweitern Sie unter Berechtigungen die Option Standardausführungsrolle ändern. Sie können eine neue Ausführungsrolle erstellen oder eine vorhandene Rolle verwenden.
5. Wählen Sie Funktion erstellen. Lambda erstellt eine grundlegende „Hello World“-Funktion mit der von Ihnen gewählten Laufzeit.

So laden Sie ein ZIP-Archiv von Ihrem lokalen Computer hoch (Konsole)

1. Wählen Sie auf der Funktionsseite der Lambda-Konsole die Funktion aus, für die Sie die ZIP-Datei hochladen möchten.
2. Wählen Sie die Registerkarte Code aus.
3. Wählen Sie im Bereich Codequelle die Option Hochladen von aus.
4. Wählen Sie die ZIP-Datei aus.
5. Laden Sie die ZIP-Datei wie folgt hoch:

   a. Wählen Sie Hochladen und dann Ihre ZIP-Datei in der Dateiauswahl aus.

   b. Klicken Sie auf Open.

   c. Wählen Sie Speichern.

So laden Sie ein ZIP-Archiv aus einem Amazon-S3-Bucket hoch (Konsole)

1. Wählen Sie auf der Funktionsseite der Lambda-Konsole die Funktion aus, für die Sie eine neue ZIP-Datei hochladen möchten.
2. Wählen Sie die Registerkarte Code aus.
3. Wählen Sie im Bereich Codequelle die Option Hochladen von aus.
4. Wählen Sie den Amazon-S3-Speicherort aus.
5. Fügen Sie die Amazon-S3-Link-URL Ihrer ZIP-Datei ein und wählen Sie Speichern aus.

ZIP-Dateifunktionen mithilfe des Konsolencode-Editors aktualisieren

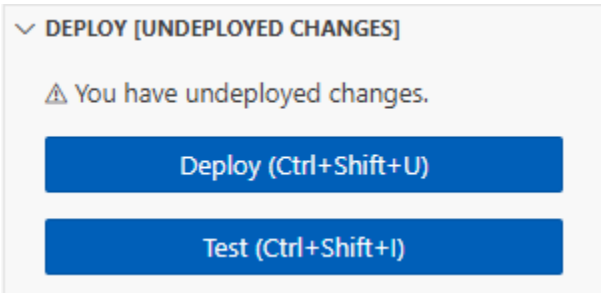
Für einige Funktionen mit ZIP-Bereitstellungspaketen können Sie Ihren Funktionscode direkt mit dem in der Lambda-Konsole integrierten Code-Editor aktualisieren. Zur Verwendung dieses Features muss Ihre Funktion folgende Kriterien erfüllen:

- Ihre Funktion muss eine der interpretierten Sprache der Laufzeit verwenden (Python, Node.js oder Ruby).
- Das Bereitstellungspaket Ihrer Funktion muss kleiner als 50 MB (entpackt) sein.

Funktionscode für Funktionen mit Container-Image-Bereitstellungspaketen kann nicht direkt in der Konsole bearbeitet werden.

So aktualisieren Sie Ihren Funktionscode mit dem Code-Editor

1. Öffnen Sie die Funktionsseite der Lambda-Konsole und wählen Sie Ihre Funktion aus.
2. Wählen Sie die Registerkarte Code aus.
3. Wählen Sie im Bereich Codequelle Ihre Quellcodedatei aus und bearbeiten Sie sie im integrierten Code-Editor.
4. Wählen Sie im Abschnitt BEREITSTELLEN die Option Bereitstellen aus, um den Code Ihrer Funktion zu aktualisieren:



Erstellen und Aktualisieren von Funktionen mit ZIP-Dateien mithilfe der AWS CLI

Sie können die AWS CLI verwenden, um eine neue Funktion zu erstellen oder eine vorhandene unter Verwendung einer ZIP-Datei zu aktualisieren. Verwenden Sie die Erstellungsfunktion und die update-function-codeBefehle, um Ihr .zip-Paket bereitzustellen. Wenn Ihre ZIP-Datei kleiner als 50 MB ist, können Sie das ZIP-Paket von einem Dateispeicherort auf Ihrem lokalen Build-Computer hochladen. Bei größeren Dateien müssen Sie Ihr ZIP-Paket aus einem Amazon-S3-Bucket

hochladen. Anweisungen zum Hochladen einer Datei in einen Amazon S3 S3-Bucket mithilfe von finden Sie unter Objekte verschieben im AWS CLI Benutzerhandbuch. AWS CLI

> **Note**
>
> Wenn Sie Ihre ZIP-Datei mithilfe von aus einem Amazon S3 S3-Bucket hochladen AWS CLI, muss sich der Bucket im selben Verzeichnis befinden AWS-Region wie Ihre Funktion.

Um eine neue Funktion mithilfe einer .zip-Datei mit dem zu erstellen AWS CLI, müssen Sie Folgendes angeben:

- Den Namen Ihrer Funktion (`--function-name`)
- Die Laufzeit Ihrer Funktion (`--runtime`)
- Den Amazon-Ressourcennamen (ARN) der Ausführungsrolle der Funktion (`--role`).
- Den Namen der Handler-Methode in Ihrem Funktionscode (`--handler`)

Sie müssen auch den Speicherort Ihrer ZIP-Datei angeben. Befindet sich Ihre ZIP-Datei in einem Ordner auf Ihrem lokalen Build-Computer, verwenden Sie die `--zip-file`-Option, um den Dateipfad anzugeben, wie im folgenden Beispielbefehl gezeigt.

```
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
```

Zur Angabe des Speicherorts der ZIP-Datei in einem Amazon-S3-Bucket verwenden Sie die im folgenden Beispielbefehl gezeigte `--code`-Option. Sie müssen den `S3ObjectVersion`-Parameter nur für versionierte Objekte verwenden.

```
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
```

Um eine vorhandene Funktion mit der CLI zu aktualisieren, geben Sie den Namen Ihrer Funktion unter Verwendung des `--function-name`-Parameters an. Sie müssen auch den Speicherort der

ZIP-Datei angeben, die Sie zum Aktualisieren Ihres Funktionscodes verwenden möchten. Befindet sich Ihre ZIP-Datei in einem Ordner auf Ihrem lokalen Build-Computer, verwenden Sie die `--zip-file`-Option, um den Dateipfad anzugeben, wie im folgenden Beispielbefehl gezeigt.

```
aws lambda update-function-code --function-name myFunction \
--zip-file fileb://myFunction.zip
```

Zur Angabe des Speicherorts der ZIP-Datei in einem Amazon-S3-Bucket verwenden Sie die im folgenden Beispielbefehl gezeigten `--s3-bucket`- und `--s3-key`-Optionen. Sie müssen den `--s3-object-version`-Parameter nur für versionierte Objekte verwenden.

```
aws lambda update-function-code --function-name myFunction \
--s3-bucket amzn-s3-demo-bucket --s3-key myFileName.zip --s3-object-version myObject
 Version
```

Erstellen und Aktualisieren von Funktionen mit ZIP-Dateien unter Verwendung der Lambda-API

Um Funktionen zu erstellen und zu konfigurieren, die ein ZIP-Dateiarchiv verwenden, verwenden Sie die folgenden API-Operationen:

- CreateFunction
- UpdateFunctionCode

Funktionen mit ZIP-Dateien erstellen und aktualisieren mit AWS SAM

Das AWS Serverless Application Model (AWS SAM) ist ein Toolkit, das dabei hilft, den Prozess der Erstellung und Ausführung serverloser Anwendungen zu optimieren. AWS Sie definieren die Ressourcen für Ihre Anwendung in einer YAML- oder JSON-Vorlage und verwenden die AWS SAM Befehlszeilenschnittstelle (AWS SAM CLI), um Ihre Anwendungen zu erstellen, zu verpacken und bereitzustellen. Wenn Sie eine Lambda-Funktion aus einer AWS SAM Vorlage erstellen, AWS SAM wird automatisch ein ZIP-Bereitstellungspaket oder ein Container-Image mit Ihrem Funktionscode und allen von Ihnen angegebenen Abhängigkeiten erstellt. Weitere Informationen zur Verwendung AWS SAM zum Erstellen und Bereitstellen von Lambda-Funktionen finden Sie unter Erste Schritte mit AWS SAM im AWS Serverless Application Model Entwicklerhandbuch.

Sie können es auch verwenden AWS SAM , um eine Lambda-Funktion mithilfe eines vorhandenen ZIP-Dateiarchivs zu erstellen. Um eine Lambda-Funktion zu erstellen AWS SAM, können Sie Ihre

ZIP-Datei in einem Amazon S3 S3-Bucket oder in einem lokalen Ordner auf Ihrem Build-Computer speichern. Anweisungen zum Hochladen einer Datei in einen Amazon S3 S3-Bucket mithilfe von finden Sie unter Objekte verschieben im AWS CLI Benutzerhandbuch. AWS CLI

In Ihrer AWS SAM Vorlage spezifiziert die `AWS::Serverless::Function` Ressource Ihre Lambda-Funktion. Legen Sie in dieser Ressource die folgenden Eigenschaften fest, um eine Funktion zu erstellen, die als ZIP-Datei-Archiv definiert ist:

- `PackageType` – festlegen auf `Zip`
- `CodeUri`- auf die Amazon S3 S3-URI, den Pfad zum lokalen Ordner oder FunctionCodeObjekt des Funktionscodes gesetzt
- `Runtime` – festlegen auf die gewünschte Laufzeit

Wenn Ihre ZIP-Datei größer als 50 MB ist, müssen Sie sie nicht zuerst in einen Amazon S3 S3-Bucket hochladen. AWS SAM AWS SAM kann .zip-Pakete bis zur maximal zulässigen Größe von 250 MB (entpackt) von einem Speicherort auf Ihrem lokalen Build-Computer hochladen.

Weitere Informationen zum Bereitstellen von Funktionen mithilfe der ZIP-Datei in finden Sie AWS::Serverless::Functionim AWS SAM Entwicklerhandbuch.AWS SAM

Erstellen und Aktualisieren von Funktionen mit ZIP-Dateien mithilfe von CloudFormation

Sie können verwenden CloudFormation , um eine Lambda-Funktion mithilfe eines ZIP-Dateiarchivs zu erstellen. Um eine Lambda-Funktion aus einer ZIP-Datei zu erstellen, müssen Sie Ihre Datei zunächst in einen Amazon-S3-Bucket hochladen. Anweisungen zum Hochladen einer Datei in einen Amazon S3 S3-Bucket mithilfe von finden Sie unter Objekte verschieben im AWS CLI Benutzerhandbuch. AWS CLI

In Ihrer CloudFormation Vorlage spezifiziert die `AWS::Lambda::Function` Ressource Ihre Lambda-Funktion. Legen Sie in dieser Ressource die folgenden Eigenschaften fest, um eine Funktion zu erstellen, die als ZIP-Datei-Archiv definiert ist:

- `PackageType` – festlegen auf `Zip`
- `Code` – Geben Sie den Namen des Amazon-S3-Buckets und den ZIP-Dateinamen in die Felder `S3Bucket` und `S3Key` ein
- `Runtime` – festlegen auf die gewünschte Laufzeit

Die CloudFormation generierte ZIP-Datei darf 4 MB nicht überschreiten. Weitere Informationen zum Bereitstellen von Funktionen mithilfe der ZIP-Datei finden Sie AWS::Lambda::Functionim CloudFormation Benutzerhandbuch. CloudFormation

Bereitstellen von Node.js-Lambda-Funktionen mit Container-Images

Es gibt drei Möglichkeiten, ein Container-Image für eine Node.js-Lambda-Funktion zu erstellen:

- Verwenden Sie ein Basis-Image für Node.js AWS

  Die AWS -Basis-Images sind mit einer Sprachlaufzeit, einem Laufzeitschnittstellen-Client zur Verwaltung der Interaktion zwischen Lambda und Ihrem Funktionscode und einem Laufzeitschnittstellen-Emulator für lokale Tests vorinstalliert.
- Es wird ein AWS reines Betriebssystem-Basis-Image verwendet

  AWS Basis-Images nur für Betriebssysteme enthalten eine Amazon Linux-Distribution und den Runtime-Interface-Emulator. Diese Images werden häufig verwendet, um Container-Images für kompilierte Sprachen wie Go und Rust sowie für eine Sprache oder Sprachversion zu erstellen, für die Lambda kein Basis-Image bereitstellt, wie Node.js 19. Sie können reine OS-Basis-Images auch verwenden, um eine benutzerdefinierte Laufzeit zu implementieren. Um das Image mit Lambda kompatibel zu machen, müssen Sie den Laufzeitschnittstellen-Client für Node.js in das Image aufnehmen.
- Verwenden Sie ein Nicht-Basis-Image AWS

  Sie können auch ein alternatives Basis-Image aus einer anderen Container-Registry verwenden. Sie können auch ein von Ihrer Organisation erstelltes benutzerdefiniertes Image verwenden. Um das Image mit Lambda kompatibel zu machen, müssen Sie den Laufzeitschnittstellen-Client für Node.js in das Image aufnehmen.

> **Tip**
>
> Um die Zeit zu reduzieren, die benötigt wird, bis Lambda-Container-Funktionen aktiv werden, siehe die Docker-Dokumentation unter Verwenden mehrstufiger Builds. Um effiziente Container-Images zu erstellen, folgen Sie den Bewährte Methoden für das Schreiben von Dockerfiles.

Auf dieser Seite wird erklärt, wie Sie Container-Images für Lambda erstellen, testen und bereitstellen.

Themen

- AWS Basis-Images für Node.js
- Verwenden Sie ein Basis-Image für Node.js AWS
- Verwenden eines alternativen Basis-Images mit dem Laufzeitschnittstellen-Client

AWS Basis-Images für Node.js

AWS stellt die folgenden Basis-Images für Node.js bereit:

Tags (Markier ngen)	Laufzeit	Betriebss ystem	Dockerfile	Ablehnung
24	Node.js 24	Amazon Linux 2023	Dockerfile für Node.js 24 auf GitHub	30. April 2028
22	Node.js 22	Amazon Linux 2023	Dockerfile für Node.js 22 auf GitHub	30. April 2027
20	Node.js 20	Amazon Linux 2023	Dockerfile für Node.js 2.0 auf GitHub	30. April 2026

Amazon ECR-Repository: gallery.ecr. aws/lambda/nodejs

Die Basis-Images von Node.js 20 und höher basieren auf dem minimalen Container-Image von Amazon Linux 2023. Frühere Basis-Images verwenden Amazon Linux 2. AL2023 bietet mehrere Vorteile gegenüber Amazon Linux 2, darunter einen geringeren Bereitstellungsaufwand und aktualisierte Versionen von Bibliotheken wie`glibc`.

AL2023-basierte Images verwenden `microdnf` (symbolisiert als`dnf`) als Paketmanager anstelle von`yum`, dem Standard-Paketmanager in Amazon Linux 2. `microdnf`ist eine eigenständige Implementierung von. `dnf` Eine Liste der Pakete, die in AL2 023-basierten Images enthalten sind, finden Sie in den Spalten Minimal Container unter Comparing packages installed on Amazon Linux 2023 Container Images. Weitere Informationen zu den Unterschieden zwischen AL2 023 und Amazon Linux 2 finden Sie unter Einführung in die Amazon Linux 2023 Runtime for AWS Lambda im AWS Compute-Blog.

> **Note**
>
> Um AL2 023-basierte Images lokal auszuführen, auch mit AWS Serverless Application Model (AWS SAM), müssen Sie Docker-Version 20.10.10 oder höher verwenden.

Verwenden Sie ein Basis-Image für Node.js AWS

Voraussetzungen

Zur Durchführung der Schritte in diesem Abschnitt benötigen Sie Folgendes:

- AWS CLI Version 2
- Docker (Mindestversion 25.0.0)
- Das Docker-buildx-Plugin.
- Node.js

Erstellen eines Images aus einem Base Image

Um ein Container-Image aus einem AWS Basis-Image für Node.js zu erstellen

1. Erstellen Sie ein Verzeichnis für das Projekt und wechseln Sie dann zu diesem Verzeichnis.

   ```
   mkdir example
   cd example
   ```

2. Erstellen Sie ein neues Node.js-Projekt mit `npm`. Um die im interaktiven Erlebnis bereitgestellten Standardoptionen zu akzeptieren, drücken Sie `Enter`.

   ```
   npm init
   ```

3. Erstellen Sie eine neue Datei mit dem Namen `index.js`. Sie können der Datei zum Testen den folgenden Beispielfunktionscode hinzufügen oder Ihren eigenen verwenden.

   Example CommonJS-Handler

   ```
   exports.handler = async (event) => {
       const response = {
           statusCode: 200,
   ```

```
        body: JSON.stringify('Hello from Lambda!'),
    };
    return response;
};
```

4. Wenn Ihre Funktion von anderen Bibliotheken als den abhängt AWS SDK für JavaScript, verwenden Sie npm, um sie Ihrem Paket hinzuzufügen.
5. Erstellen Sie eine neue Docker-Datei mit der folgenden Konfiguration:

   - Setzen Sie die `FROM`-Eigenschaft auf den URI des Basis-Images.
   - Verwenden Sie den Befehl COPY, um den Funktionscode und die Laufzeitabhängigkeiten in `{LAMBDA_TASK_ROOT}`, eine von Lambda definierte Umgebungsvariable zu kopieren.
   - Legen Sie das `CMD`-Argument auf den Lambda-Funktionshandler fest.

   Beachten Sie, dass das Dockerfile-Beispiel keine USER-Anweisung enthält. Wenn Sie ein Container-Image für Lambda bereitstellen, definiert Lambda automatisch einen Standard-Linux-Benutzer mit Berechtigungen mit geringsten Rechten. Dies unterscheidet sich vom Standardverhalten von Docker, bei dem standardmäßig der `root`-Benutzer verwendet wird, wenn keine `USER`-Anweisung bereitgestellt wird.

   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. Erstellen Sie Ihr Docker-Image mit dem docker build-Befehl. Das folgende Beispiel benennt das Bild in `docker-image` und gibt ihm den `test` Tag. Um Ihr Image mit Lambda kompatibel zu machen, müssen Sie die Option `--provenance=false` verwenden.

   ```
   docker buildx build --platform linux/amd64 --provenance=false -t docker-image:test
    .
   ```

> **Note**
>
> Der Befehl gibt die `--platform linux/amd64`-Option an, um sicherzustellen, dass Ihr Container mit der Lambda-Ausführungsumgebung kompatibel ist, unabhängig von der Architektur des Entwicklungsrechners. Wenn Sie beabsichtigen, eine Lambda-Funktion mithilfe der ARM64 Befehlssatzarchitektur zu erstellen, müssen Sie den Befehl so ändern, dass er stattdessen die `--platform linux/arm64` Option verwendet.

(Optional) Testen Sie das Image lokal

1. Starten Sie Ihr Docker-Image mit dem docker run-Befehl. In diesem Beispiel ist `docker-image` der Image-Name und `test` der Tag.

   ```
   docker run --platform linux/amd64 -p 9000:8080 docker-image:test
   ```

   Dieser Befehl führt das Image als Container aus und erstellt einen lokalen Endpunkt bei `localhost:9000/2015-03-31/functions/function/invocations`.

   > **Note**
   >
   > Wenn Sie das Docker-Image für die ARM64 Befehlssatzarchitektur erstellt haben, stellen Sie sicher, dass Sie stattdessen die `--platform linux/arm64` Option verwenden. `--platform linux/amd64`

2. Veröffentlichen Sie in einem neuen Terminalfenster ein Ereignis an den lokalen Endpunkt.

   Linux/macOS

   Führen Sie unter Linux oder macOS den folgenden `curl`-Befehl aus:

   ```
   curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{}'
   ```

   Dieser Befehl ruft die Funktion mit einem leeren Ereignis auf und gibt eine Antwort zurück. Wenn Sie Ihren eigenen Funktionscode anstelle des Beispielfunktionscodes verwenden, wird empfohlen, die Funktion mit einer JSON-Nutzlast aufzurufen. Beispiel:

```
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d
 '{"payload":"hello world!"}'
```

PowerShell

Führen Sie in PowerShell den folgenden `Invoke-WebRequest` Befehl aus:

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{}' -ContentType "application/json"
```

Dieser Befehl ruft die Funktion mit einem leeren Ereignis auf und gibt eine Antwort zurück. Wenn Sie Ihren eigenen Funktionscode anstelle des Beispielfunktionscodes verwenden, wird empfohlen, die Funktion mit einer JSON-Nutzlast aufzurufen. Beispiel:

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{"payload":"hello world!"}' -ContentType
 "application/json"
```

3. Die Container-ID erhalten.

```
docker ps
```

4. Verwenden Sie den Befehl docker kill, um den Container zu anzuhalten. Ersetzen Sie in diesem Befehl `3766c4ab331c` durch die Container-ID aus dem vorherigen Schritt.

```
docker kill 3766c4ab331c
```

Das Image bereitstellen

Um das Image in Amazon ECR hochzuladen und die Lambda-Funktion zu erstellen

1. Führen Sie den get-login-passwordBefehl aus, um die Docker-CLI bei Ihrer Amazon ECR-Registrierung zu authentifizieren.

   - Setzen Sie den `--region` Wert auf den AWS-Region Ort, an dem Sie das Amazon ECR-Repository erstellen möchten.
   - Ersetzen Sie es `111122223333` durch Ihre AWS-Konto ID.

```
aws ecr get-login-password --region us-east-1 | docker login --username AWS --
password-stdin 111122223333.dkr.ecr.us-east-1.amazonaws.com
```

2. Erstellen Sie ein Repository in Amazon ECR mithilfe des Befehls create-repository.

```
aws ecr create-repository --repository-name hello-world --region us-east-1 --image-
scanning-configuration scanOnPush=true --image-tag-mutability MUTABLE
```

> **Note**
>
> Das Amazon ECR-Repository muss sich im selben Format AWS-Region wie die Lambda-Funktion befinden.

Wenn erfolgreich, sehen Sie eine Antwort wie diese:

```
{
    "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. Kopieren Sie das `repositoryUri` aus der Ausgabe im vorherigen Schritt.
4. Führen Sie den Befehl docker tag aus, um Ihr lokales Image als neueste Version in Ihrem Amazon-ECR-Repository zu markieren. In diesem Befehl gilt Folgendes:

- `docker-image:test` ist der Name und das Tag Ihres Docker-Images. Dies sind der Imagename und das Tag, die Sie im `docker build`-Befehl angegeben haben.
- Ersetzen Sie <ECRrepositoryUri> durch den `repositoryUri`, den Sie kopiert haben. Stellen Sie sicher, dass Sie `:latest` am Ende der URI angeben.

```
docker tag docker-image:test <ECRrepositoryUri>:latest
```

Beispiel:

```
docker tag docker-image:test 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

5. Führen Sie den Befehl docker push aus, um Ihr lokales Image im Amazon-ECR-Repository bereitzustellen. Stellen Sie sicher, dass Sie `:latest` am Ende der Repository-URI angeben.

```
docker push 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

6. Erstellen Sie eine Ausführungsrolle für die Funktion, wenn Sie noch keine haben. Sie benötigen den Amazon-Ressourcennamen (ARN) der Rolle im nächsten Schritt.
7. So erstellen Sie die Lambda-Funktion: Geben Sie für `ImageUri` die Repository-URI von zuvor an. Stellen Sie sicher, dass Sie `:latest` am Ende der URI angeben.

```
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**
>
> Sie können eine Funktion mit einem Bild in einem anderen AWS Konto erstellen, sofern sich das Bild in derselben Region wie die Lambda-Funktion befindet. Weitere Informationen finden Sie unter Kontoübergreifende Berechtigungen von Amazon ECR.

8. Die Funktion aufrufen.

```
aws lambda invoke --function-name hello-world response.json
```

Das Ergebnis sollte ungefähr wie folgt aussehen:

```
{
  "ExecutedVersion": "$LATEST",
  "StatusCode": 200
}
```

9. Um die Ausgabe der Funktion zu sehen, überprüfen Sie die `response.json`-Datei.

Um den Funktionscode zu aktualisieren, müssen Sie das Image erneut erstellen, das neue Image in das Amazon ECR-Repository hochladen und dann den update-function-codeBefehl verwenden, um das Image für die Lambda-Funktion bereitzustellen.

Lambda löst das Image-Tag in einen bestimmten Image-Digest auf. Das heißt, wenn Sie das Image-Tag, das zur Bereitstellung der Funktion verwendet wurde, auf ein neues Image in Amazon ECR verweisen, aktualisiert Lambda die Funktion nicht automatisch, um das neue Image zu verwenden.

Um das neue Image für dieselbe Lambda-Funktion bereitzustellen, müssen Sie den update-function-codeBefehl verwenden, auch wenn das Image-Tag in Amazon ECR gleich bleibt. Im folgenden Beispiel erstellt die `--publish`-Option eine neue Version der Funktion unter Verwendung des aktualisierten Container-Images.

```
aws lambda update-function-code \
  --function-name hello-world \
  --image-uri 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --publish
```

Verwenden eines alternativen Basis-Images mit dem Laufzeitschnittstellen-Client

Wenn Sie ein OS-Basis-Image oder ein alternatives Basis-Image verwenden, müssen Sie den Laufzeitschnittstellen-Client in das Image einbinden. Der Laufzeitschnittstellen-Client erweitert die Laufzeit-API, die die Interaktion zwischen Lambda und Ihrem Funktionscode verwaltet.

Installieren Sie den Laufzeitschnittstellen-Client für Node.js mit dem npm-Paketmanager:

```
npm install aws-lambda-ric
```

Sie können den Runtime Interface-Client von Node.js auch herunterladen GitHub.

Das folgende Beispiel zeigt, wie ein Container-Image für Node.js mithilfe eines AWS Nicht-Basis-Images erstellt wird. Das Beispiel-Dockerfile verwendet ein `bookworm`-Basis-Image. Das Docker-File enthält den Laufzeitschnittstellen-Client.

Voraussetzungen

Zur Durchführung der Schritte in diesem Abschnitt benötigen Sie Folgendes:

- AWS CLI Version 2
- Docker (Mindestversion 25.0.0)
- Das Docker-buildx-Plugin.
- Node.js

Erstellen eines Images aus einem alternativen Basis-Image

Um ein Container-Image aus einem AWS Nicht-Basis-Image zu erstellen

1. Erstellen Sie ein Verzeichnis für das Projekt und wechseln Sie dann zu diesem Verzeichnis.

   ```
   mkdir example
   cd example
   ```

2. Erstellen Sie ein neues Node.js-Projekt mit `npm`. Um die im interaktiven Erlebnis bereitgestellten Standardoptionen zu akzeptieren, drücken Sie `Enter`.

   ```
   npm init
   ```

3. Erstellen Sie eine neue Datei mit dem Namen `index.js`. Sie können der Datei zum Testen den folgenden Beispielfunktionscode hinzufügen oder Ihren eigenen verwenden.

   Example CommonJS-Handler

   ```
   exports.handler = async (event) => {
       const response = {
           statusCode: 200,
   ```

```
        body: JSON.stringify('Hello from Lambda!'),
    };
    return response;
};
```

4. Erstellen Sie eine neue Docker-Datei. Das folgende Dockerfile verwendet ein `bookworm`-Basis-Image anstelle eines AWS -Basis-Images. Das Dockerfile enthält den Laufzeitschnittstellen-Client, der das Image mit Lambda kompatibel macht. Das folgende Dockerfile verwendet einen Build mit mehreren Phasen. In der ersten Phase wird ein Build-Image erstellt, bei dem es sich um eine Standardumgebung von Node.js handelt, in der die Abhängigkeiten der Funktion installiert werden. In der zweiten Phase wird ein schlankeres Image erstellt, das den Funktionscode und seine Abhängigkeiten enthält. Dadurch wird die endgültige Image-Größe reduziert.

   - Legen Sie `FROM`-Eigenschaft auf die Kennung des Basis-Images fest.
   - Verwenden Sie den Befehl `COPY`, um den Funktionscode und die Laufzeitabhängigkeiten zu kopieren.
   - Legen Sie `ENTRYPOINT` auf das Modul fest, das der Docker-Container beim Start ausführen soll. In diesem Fall ist das Modul der Laufzeitschnittstellen-Client.
   - Legen Sie das `CMD`-Argument auf den Lambda-Funktionshandler fest.

   Beachten Sie, dass das Dockerfile-Beispiel keine USER-Anweisung enthält. Wenn Sie ein Container-Image für Lambda bereitstellen, definiert Lambda automatisch einen Standard-Linux-Benutzer mit Berechtigungen mit geringsten Rechten. Dies unterscheidet sich vom Standardverhalten von Docker, bei dem standardmäßig der `root`-Benutzer verwendet wird, wenn keine `USER`-Anweisung bereitgestellt wird.

   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. Erstellen Sie Ihr Docker-Image mit dem docker build-Befehl. Das folgende Beispiel benennt das Bild in `docker-image` und gibt ihm den `test` Tag. Um Ihr Image mit Lambda kompatibel zu machen, müssen Sie die Option `--provenance=false` verwenden.

```
docker buildx build --platform linux/amd64 --provenance=false -t docker-image:test
 .
```

> **Note**
>
> Der Befehl gibt die `--platform linux/amd64`-Option an, um sicherzustellen, dass Ihr Container mit der Lambda-Ausführungsumgebung kompatibel ist, unabhängig von der Architektur des Entwicklungsrechners. Wenn Sie beabsichtigen, eine Lambda-Funktion mithilfe der ARM64 Befehlssatzarchitektur zu erstellen, müssen Sie den Befehl so ändern, dass er stattdessen die `--platform linux/arm64` Option verwendet.

(Optional) Testen Sie das Image lokal

Verwenden Sie den Laufzeit-Schnittstellen-Emulator, um das Image lokal zu testen. Sie können den Emulator in Ihr Image integrieren oder ihn mit dem folgenden Verfahren auf Ihrem lokalen Computer installieren.

Installieren des Laufzeitschnittstellen-Emulators auf Ihrem lokalen Computer

1. Führen Sie in Ihrem Projektverzeichnis den folgenden Befehl aus, um den Runtime-Interface-Emulator (x86-64-Architektur) herunterzuladen GitHub und auf Ihrem lokalen Computer zu installieren.

   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
   ```

   Um den arm64-Emulator zu installieren, ersetzen Sie die GitHub Repository-URL im vorherigen Befehl durch Folgendes:

   ```
   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
```

Um den arm64-Emulator zu installieren, ersetzen Sie das `$downloadLink` durch Folgendes:

```
https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/
download/aws-lambda-rie-arm64
```

2. Starten Sie Ihr Docker-Image mit dem docker run-Befehl. Beachten Sie Folgendes:

   - `docker-image` ist der Image-Name und `test` ist das Tag.
   - `/usr/local/bin/npx aws-lambda-ric index.handler` ist der `ENTRYPOINT` gefolgt von dem `CMD` aus Ihrem 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
   ```

Dieser Befehl führt das Image als Container aus und erstellt einen lokalen Endpunkt bei `localhost:9000/2015-03-31/functions/function/invocations`.

> **Note**
>
> Wenn Sie das Docker-Image für die ARM64 Befehlssatzarchitektur erstellt haben, stellen Sie sicher, dass Sie die `--platform linux/arm64` Option anstelle von verwenden. `--platform linux/amd64`

3. Veröffentlichen Sie ein Ereignis auf dem lokalen Endpunkt.

   Linux/macOS

   Führen Sie unter Linux oder macOS den folgenden `curl`-Befehl aus:

   ```
   curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{}'
   ```

   Dieser Befehl ruft die Funktion mit einem leeren Ereignis auf und gibt eine Antwort zurück. Wenn Sie Ihren eigenen Funktionscode anstelle des Beispielfunktionscodes verwenden, wird empfohlen, die Funktion mit einer JSON-Nutzlast aufzurufen. Beispiel:

   ```
   curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d
    '{"payload":"hello world!"}'
   ```

   PowerShell

   Führen Sie in PowerShell den folgenden `Invoke-WebRequest` Befehl aus:

   ```
   Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
   invocations" -Method Post -Body '{}' -ContentType "application/json"
   ```

   Dieser Befehl ruft die Funktion mit einem leeren Ereignis auf und gibt eine Antwort zurück. Wenn Sie Ihren eigenen Funktionscode anstelle des Beispielfunktionscodes verwenden, wird empfohlen, die Funktion mit einer JSON-Nutzlast aufzurufen. Beispiel:

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{"payload":"hello world!"}' -ContentType
 "application/json"
```

4. Die Container-ID erhalten.

```
docker ps
```

5. Verwenden Sie den Befehl docker kill, um den Container zu anzuhalten. Ersetzen Sie in diesem Befehl `3766c4ab331c` durch die Container-ID aus dem vorherigen Schritt.

```
docker kill 3766c4ab331c
```

Das Image bereitstellen

Um das Image in Amazon ECR hochzuladen und die Lambda-Funktion zu erstellen

1. Führen Sie den get-login-passwordBefehl aus, um die Docker-CLI bei Ihrer Amazon ECR-Registrierung zu authentifizieren.
   - Setzen Sie den `--region` Wert auf den AWS-Region Ort, an dem Sie das Amazon ECR-Repository erstellen möchten.
   - Ersetzen Sie es `111122223333` durch Ihre AWS-Konto ID.

```
aws ecr get-login-password --region us-east-1 | docker login --username AWS --
password-stdin 111122223333.dkr.ecr.us-east-1.amazonaws.com
```

2. Erstellen Sie ein Repository in Amazon ECR mithilfe des Befehls create-repository.

```
aws ecr create-repository --repository-name hello-world --region us-east-1 --image-
scanning-configuration scanOnPush=true --image-tag-mutability MUTABLE
```

> **Note**
>
> Das Amazon ECR-Repository muss sich im selben Format AWS-Region wie die Lambda-Funktion befinden.

Wenn erfolgreich, sehen Sie eine Antwort wie diese:

```
{
    "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. Kopieren Sie das `repositoryUri` aus der Ausgabe im vorherigen Schritt.
4. Führen Sie den Befehl docker tag aus, um Ihr lokales Image als neueste Version in Ihrem Amazon-ECR-Repository zu markieren. In diesem Befehl gilt Folgendes:

   - `docker-image:test` ist der Name und das Tag Ihres Docker-Images. Dies sind der Imagename und das Tag, die Sie im `docker build`-Befehl angegeben haben.
   - Ersetzen Sie <ECRrepositoryUri> durch den `repositoryUri`, den Sie kopiert haben. Stellen Sie sicher, dass Sie `:latest` am Ende der URI angeben.

```
docker tag docker-image:test <ECRrepositoryUri>:latest
```

Beispiel:

```
docker tag docker-image:test 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-
world:latest
```

5. Führen Sie den Befehl docker push aus, um Ihr lokales Image im Amazon-ECR-Repository bereitzustellen. Stellen Sie sicher, dass Sie `:latest` am Ende der Repository-URI angeben.

```
docker push 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

6. Erstellen Sie eine Ausführungsrolle für die Funktion, wenn Sie noch keine haben. Sie benötigen den Amazon-Ressourcennamen (ARN) der Rolle im nächsten Schritt.

7. So erstellen Sie die Lambda-Funktion: Geben Sie für `ImageUri` die Repository-URI von zuvor an. Stellen Sie sicher, dass Sie `:latest` am Ende der URI angeben.

```
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**
>
> Sie können eine Funktion mit einem Bild in einem anderen AWS Konto erstellen, sofern sich das Bild in derselben Region wie die Lambda-Funktion befindet. Weitere Informationen finden Sie unter Kontoübergreifende Berechtigungen von Amazon ECR.

8. Die Funktion aufrufen.

```
aws lambda invoke --function-name hello-world response.json
```

Das Ergebnis sollte ungefähr wie folgt aussehen:

```
{
  "ExecutedVersion": "$LATEST",
  "StatusCode": 200
}
```

9. Um die Ausgabe der Funktion zu sehen, überprüfen Sie die `response.json`-Datei.

Um den Funktionscode zu aktualisieren, müssen Sie das Image erneut erstellen, das neue Image in das Amazon ECR-Repository hochladen und dann den update-function-codeBefehl verwenden, um das Image für die Lambda-Funktion bereitzustellen.

Lambda löst das Image-Tag in einen bestimmten Image-Digest auf. Das heißt, wenn Sie das Image-Tag, das zur Bereitstellung der Funktion verwendet wurde, auf ein neues Image in Amazon ECR verweisen, aktualisiert Lambda die Funktion nicht automatisch, um das neue Image zu verwenden.

Um das neue Image für dieselbe Lambda-Funktion bereitzustellen, müssen Sie den update-function-codeBefehl verwenden, auch wenn das Image-Tag in Amazon ECR gleich bleibt. Im folgenden Beispiel erstellt die `--publish`-Option eine neue Version der Funktion unter Verwendung des aktualisierten Container-Images.

```
aws lambda update-function-code \
  --function-name hello-world \
  --image-uri 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --publish
```

Arbeiten mit Ebenen für Node.js Lambda-Funktionen

Verwenden Sie Lambda-Ebenen, um Code und Abhängigkeiten zu bündeln, die Sie für mehrere Funktionen wiederverwenden möchten. Ebenen enthalten üblicherweise Bibliotheksabhängigkeiten, eine benutzerdefinierte Laufzeit oder Konfigurationsdateien. Das Erstellen einer Ebene umfasst drei allgemeine Schritte:

1. Verpacken Ihres Ebeneninhalts. Dies bedeutet, dass Sie ein .zip-Dateiarchiv erstellen, das die Abhängigkeiten enthält, die Sie in Ihren Funktionen verwenden möchten.
2. Erstellen Sie die Ebene in Lambda.
3. Fügen Sie die Ebene zu Ihren Funktionen hinzu.

Themen

- Verpacken Ihres Ebeneninhalts
- Erstellen Sie die Ebene in Lambda
- Fügen Sie die Ebene zu Ihrer Funktion hinzu
- Beispiel-App

Verpacken Ihres Ebeneninhalts

Um eine Ebene zu erstellen, bündeln Sie Ihre Pakete in einem ZIP-Dateiarchiv, das die folgenden Anforderungen erfüllt:

- Erstellen Sie die Ebene mithilfe derselben Version von Node.js, die Sie für die Lambda-Funktion verwenden möchten. Wenn Sie Ihren Layer beispielsweise mit Node.js 24 erstellen, verwenden Sie die Laufzeit Node.js 24 für Ihre Funktion.
- Die ZIP-Datei Ihrer Ebene muss eine der folgenden Verzeichnisstrukturen verwenden:
  - `nodejs/node_modules`
  - `nodejs/node`*X*`/node_modules`(wo *X* ist zum Beispiel Ihre Node.js -Version`node22`)

  Weitere Informationen finden Sie unter Ebenenpfade für jede Lambda-Laufzeit.
- Die Pakete in Ihrer Ebene müssen mit Linux kompatibel sein. Lambda-Funktionen werden auf Amazon Linux ausgeführt.

Sie können Ebenen erstellen, die entweder Bibliotheken vom Typ Node.js von Drittanbietern enthalten, die mit installiert wurden `npm` (z. B. `axios` oder`lodash`), oder Ihre eigenen JavaScript Module.

Abhängigkeiten von Drittanbietern

So erstellen Sie eine Ebene mit npm-Paketen

1. Erstellen Sie die erforderliche Verzeichnisstruktur und installieren Sie Pakete direkt darin:

   ```
   mkdir -p nodejs
   npm install --prefix nodejs lodash axios
   ```

   Mit diesem Befehl werden die Pakete direkt in das Verzeichnis `nodejs/node_modules` installiert. Dies ist die Struktur, die Lambda benötigt.

   > **Note**
   >
   > Stellen Sie bei Paketen mit nativen Abhängigkeiten oder binären Komponenten (wie sharp oder bcrypt) sicher, dass sie mit der Lambda-Linux-Umgebung und der Architektur Ihrer Funktion kompatibel sind. Möglicherweise müssen Sie das Flag `--platform` verwenden:
   >
   > ```
   > npm install --prefix nodejs --platform=linux --arch=x64 sharp
   > ```
   >
   > Bei komplexeren nativen Abhängigkeiten ist es unter Umständen erforderlich, diese in einer Linux-Umgebung zu kompilieren, die der Lambda-Laufzeit entspricht. Zu diesem Zweck können Sie Docker verwenden.

2. Verpacken Sie den Ebeneninhalt:

   Linux/macOS

   ```
   zip -r layer.zip nodejs/
   ```

   PowerShell

   ```
   Compress-Archive -Path .\nodejs -DestinationPath .\layer.zip
   ```

Die Verzeichnisstruktur Ihrer ZIP-Datei sollte wie folgt aussehen:

```
nodejs/
### package.json
### package-lock.json
### node_modules/
    ### lodash/
    ### axios/
    ### (dependencies of the other packages)
```

> **Note**
>
> - Stellen Sie sicher, dass Ihre ZIP-Datei das `nodejs`-Verzeichnis auf der Stammebene enthält und sich `node_modules` darin befindet. Diese Struktur stellt sicher, dass Lambda Ihre Pakete finden und importieren kann.
> - Die Dateien `package.json` und `package-lock.json` im `nodejs/`-Verzeichnis werden von npm für die Abhängigkeitsverwaltung verwendet, sind aber für die Funktionalität der Ebenen von Lambda nicht erforderlich. Jedes installierte Paket enthält bereits eine eigene `package.json`-Datei, die definiert, wie Lambda das Paket importiert.

Benutzerdefinierte JavaScript Module

So erstellen Sie eine Ebene mithilfe Ihres eigenen Codes

1. Erstellen Sie die erforderliche Verzeichnisstruktur für Ihre Ebene:

   ```
   mkdir -p nodejs/node_modules/validator
   cd nodejs/node_modules/validator
   ```

2. Erstellen Sie eine `package.json`-Datei für Ihr benutzerdefiniertes Modul, um zu definieren, wie es importiert werden soll:

   Example nodejs/node_modules/validator/package.json

   ```
   {
   ```

```
  "name": "validator",
  "version": "1.0.0",
  "type": "module",
  "main": "index.mjs"
}
```

3. Erstellen Sie Ihre JavaScript Moduldatei:

   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. Verpacken Sie den Ebeneninhalt:

Linux/macOS

```
zip -r layer.zip nodejs/
```

PowerShell

```
Compress-Archive -Path .\nodejs -DestinationPath .\layer.zip
```

Die Verzeichnisstruktur Ihrer ZIP-Datei sollte wie folgt aussehen:

```
nodejs/
### node_modules/
    ### validator/
        ### package.json
        ### index.mjs
```

5. Importieren und verwenden Sie die Module in Ihrer Funktion. Beispiel:

```
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'
    });
  }
};
```

Sie können das folgende Testereignis verwenden, um die Funktion aufzurufen:

```
{
    "body": "{\"productId\": \"ABC123\", \"quantity\": 2, \"priority\": \"express
\"}"
}
```

Erwartete Antwort:

```
{
  "statusCode": 200,
  "body": "{\"message\":\"Order validated successfully\",\"order\":{\"productId\":
\"ABC123\",\"quantity\":2,\"shippingPriority\":\"express\"}}"
}
```

Erstellen Sie die Ebene in Lambda

Sie können Ihren Layer entweder mit der AWS CLI oder der Lambda-Konsole veröffentlichen.

AWS CLI

Führen Sie den publish-layer-version AWS CLI Befehl aus, um die Lambda-Schicht zu erstellen:

```
aws lambda publish-layer-version --layer-name my-layer --zip-file fileb://layer.zip
 --compatible-runtimes nodejs24.x
```

Der Parameter Kompatible Laufzeiten ist optional. Wenn angegeben, verwendet Lambda diesen Parameter zum Filtern von Ebenen in der Lambda-Konsole.

Console

So erstellen Sie eine Ebene (Konsole)

1. Öffnen Sie die Seite Ebenen der Lambda-Konsole.

2. Wählen Sie Create Layer (Ebene erstellen) aus.
3. Wählen Sie ZIP-Datei hochladen und laden Sie dann das zuvor erstellte ZIP-Archiv hoch.
4. (Optional) Wählen Sie für Kompatible Laufzeiten die Node.js-Laufzeit aus, die der Node.js-Version entspricht, mit der Sie die Ebene erstellt haben.
5. Wählen Sie Erstellen aus.

Fügen Sie die Ebene zu Ihrer Funktion hinzu

AWS CLI

Führen Sie den update-function-configuration AWS CLI Befehl aus, um die Ebene an Ihre Funktion anzuhängen. Verwenden Sie den ARN der Ebene für den `--layers`-Parameter. Der ARN muss die Version angeben (z. B. `arn:aws:lambda:us-east-1:123456789012:layer:my-layer:1`). Weitere Informationen finden Sie unter Ebenen und Ebenenversionen.

```
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"
```

Die cli-binary-format Option ist erforderlich, wenn Sie AWS CLI Version 2 verwenden. Um dies zur Standardeinstellung zu machen, führen Sie `aws configure set cli-binary-format raw-in-base64-out` aus. Weitere Informationen finden Sie unter Von AWS CLI unterstützte globale Befehlszeilenoptionen im AWS Command Line Interface -Benutzerhandbuch für Version 2.

Console

So fügen Sie eine Ebene zu einer Funktion hinzu

1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.
2. Wählen Sie die Funktion aus.
3. Blättern Sie nach unten bis zum Abschnitt Ebenen und wählen Sie dann Eine Ebene hinzufügen.
4. Wählen Sie unter Ebene auswählen die Option Benutzerdefinierte Ebenen und dann Ihre Ebene aus.

> **Note**
>
> Wenn Sie bei der Erstellung der Ebene keine kompatible Laufzeit hinzugefügt haben, wird die Ebene hier nicht aufgeführt. Sie können stattdessen den ARN der Ebene angeben.

5. Wählen Sie Hinzufügen aus.

Beispiel-App

Weitere Beispiele für die Verwendung von Lambda-Layern finden Sie in der Layer-Nodejs-Beispielanwendung im Developer Guide-Repository. AWS Lambda GitHub Diese Anwendung enthält eine Ebene, die die lodash-Bibliothek enthält. Nach dem Erstellen der Ebene können Sie die entsprechende Funktion bereitstellen und aufrufen, um zu bestätigen, dass die Ebene wie erwartet funktioniert.

Verwenden des Lambda-Kontextobjekts zum Abrufen von Node.js-Funktionsinformationen

Wenn Lambda Ihre Funktion ausführt, wird ein Context-Objekt an den Handler. übergeben. Dieses Objekt stellt Methoden und Eigenschaften mit Informationen zum Aufruf, zur Funktion und zur Ausführungsumgebung bereit.

Context-Methoden

- `getRemainingTimeInMillis()` – Gibt die Anzahl der verbleibenden Millisekunden zurück, bevor die Ausführung das Zeitlimit überschreitet.

Context-Eigenschaften

- `functionName` – Der Name der Lambda-Funktion.
- `functionVersion` – Die Version der Funktion.
- `invokedFunctionArn` – Der Amazon-Ressourcenname (ARN), der zum Aufrufen der Funktion verwendet wird. Gibt an, ob der Aufrufer eine Versionsnummer oder einen Alias angegeben hat.
- `memoryLimitInMB` – Die Menge an Arbeitsspeicher, die der Funktion zugewiesen ist.
- `awsRequestId` – Der Bezeichner der Aufrufanforderung.
- `logGroupName` – Protokollgruppe für die Funktion.
- `logStreamName` – Der Protokollstream für die Funktions-Instance.
- `identity` – Informationen zur Amazon-Cognito-Identität, die die Anforderung autorisiert hat.
  - `cognitoIdentityId`— Die authentifizierte Amazon-Cognito-Identität.
  - `cognitoIdentityPoolId` – Der Amazon-Cognito-Identitätspool, der den Aufruf autorisiert hat.
- `clientContext` – (mobile Apps) Clientkontext, der Lambda von der Clientanwendung bereitgestellt wird.
  - `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` – Benutzerdefinierte Werte, die durch die mobilen Anwendung festgelegt werden.
- `callbackWaitsForEmptyEventLoop`— Standardmäßig (`true`) wartet Lambda bei Verwendung eines Callback-basierten Funktionshandlers darauf, dass die Ereignisschleife leer ist, nachdem der Callback ausgeführt wurde, bevor der Funktionsaufruf beendet wird. Legt fest, dass `false` die Antwort gesendet und der Aufruf sofort beendet wird, nachdem der Callback ausgeführt wurde, anstatt darauf zu warten, dass die Event-Schleife leer ist. Ausstehende Ereignisse werden beim nächsten Aufruf weiterhin ausgeführt. Beachten Sie, dass Lambda Callback-basierte Funktionshandler nur für Node.js 22 und frühere Laufzeiten unterstützt.

Die folgende Beispielfunktion protokolliert Kontextinformationen und gibt den Speicherort der Protokolle zurück.

Example Datei index.js

```
exports.handler = async function(event, context) {
  console.log('Remaining time: ', context.getRemainingTimeInMillis())
  console.log('Function name: ', context.functionName)
  return context.logStreamName
}
```

Node.js-Lambda-Funktionen protokollieren und überwachen

AWS Lambda überwacht automatisch Lambda-Funktionen in Ihrem Namen und sendet Protokolle an Amazon CloudWatch. Ihre Lambda-Funktion enthält eine CloudWatch Logs-Log-Gruppe und einen Log-Stream für jede Instanz Ihrer Funktion. Die Lambda-Laufzeitumgebung sendet Details zu den einzelnen Aufrufen an den Protokollstream und leitet Protokolle und andere Ausgaben aus dem Code Ihrer Funktion weiter. Weitere Informationen finden Sie unter Senden von Lambda-Funktionsprotokollen an CloudWatch Logs.

Auf dieser Seite wird beschrieben, wie Sie aus dem Code Ihrer Lambda-Funktion eine Protokollausgabe erstellen und mit der AWS Command Line Interface Lambda-Konsole oder der CloudWatch Konsole auf Logs zugreifen.

Sections

- Erstellen einer Funktion, die Protokolle zurückgibt
- Verwenden von Lambda-Optionen für die erweiterte Protokollierung mit Node.js
- Anzeigen von Protokollen in der Lambda-Konsole
- Protokolle in der Konsole anzeigen CloudWatch
- Logs mit dem Befehl () anzeigen AWS Command Line InterfaceAWS CLI
- Löschen von Protokollen

Erstellen einer Funktion, die Protokolle zurückgibt

Um Protokolle aus dem Code Ihrer Funktion auszugeben, können Sie Methoden auf dem Konsolenobjekt oder eine Protokollierungsbibliothek verwenden, die zu `stdout` oder `stderr` schreibt. Das folgende Beispiel protokolliert die Werte der Umgebungsvariablen und des Ereignisobjekts.

> **Note**
>
> Wir empfehlen, bei der Protokollierung von Eingaben Techniken wie Eingabevalidierung und Ausgabekodierung zu verwenden. Wenn Sie Eingabedaten direkt protokollieren, kann ein Angreifer Ihren Code verwenden, um Manipulationen schwer zu erkennen, Protokolleinträge zu fälschen oder Protokollüberwachungen zu umgehen. Weitere Informationen finden Sie unter Unsachgemäße Neutralisierung der Ausgabe von Protokollen in der Common Weakness Enumeration.

Example Datei index.js – Protokollierung

```
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 Protokollformat

```
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
```

Die Node.js-Laufzeit protokolliert die Zeilen `START`, `END`, und `REPORT` für jeden Aufruf. Sie fügt jedem von der Funktion protokollierten Eintrag einen Zeitstempel, eine Anforderungs-ID und eine Protokollebene hinzu. Die Berichtszeile enthält die folgenden Details.

Datenfelder für REPORT-Zeilen

- RequestId— Die eindeutige Anforderungs-ID für den Aufruf.
- Dauer – Die Zeit, die die Handler-Methode Ihrer Funktion mit der Verarbeitung des Ereignisses verbracht hat.
- Fakturierte Dauer – Die für den Aufruf fakturierte Zeit.
- Speichergröße – Die der Funktion zugewiesene Speichermenge.
- Max. verwendeter Speicher – Die Speichermenge, die von der Funktion verwendet wird. Wenn Aufrufe eine Ausführungsumgebung gemeinsam nutzen, meldet Lambda den maximalen Speicherverbrauch für alle Aufrufe. Dieses Verhalten kann zu einem höheren als erwarteten gemeldeten Wert führen.
- Initialisierungsdauer – Für die erste Anfrage die Zeit, die zur Laufzeit zum Laden der Funktion und Ausführen von Code außerhalb der Handler-Methode benötigt wurde.
- XRAY TraceId — Für verfolgte Anfragen die AWS X-Ray Trace-ID.
- SegmentId— Für verfolgte Anfragen die X-Ray-Segment-ID.
- Stichprobe – Bei verfolgten Anforderungen das Stichprobenergebnis.

Sie können Logs in der Lambda-Konsole, in der CloudWatch Logs-Konsole oder über die Befehlszeile anzeigen.

Verwenden von Lambda-Optionen für die erweiterte Protokollierung mit Node.js

Um Ihnen mehr Kontrolle darüber zu geben, wie die Protokolle Ihrer Funktionen erfasst, verarbeitet und verwendet werden, können Sie die folgenden Protokollierungsoptionen für unterstützte Node.js-Laufzeiten konfigurieren:

- Protokollformat – Wählen Sie zwischen Klartext und einem strukturierten JSON-Format für die Protokolle Ihrer Funktion aus.
- Protokollebene — für Logs im JSON-Format wählen Sie die Detailebene der Logs, die Lambda an Amazon sendet CloudWatch, wie ERROR, DEBUG oder INFO
- Protokollgruppe — wählen Sie die CloudWatch Protokollgruppe aus, an die Ihre Funktion Protokolle sendet

Weitere Informationen zu diesen Protokollierungsoptionen und Anweisungen zur Konfiguration Ihrer Funktion für deren Verwendung finden Sie unter the section called “Konfigurieren erweiterter Protokollierungsoptionen für Lambda-Funktionen”.

Informationen zur Verwendung der Optionen für das Protokollformat und die Protokollebene mit Ihren Node.js-Lambda-Funktionen finden Sie in den folgenden Abschnitten.

Verwenden strukturierter JSON-Protokolle mit Node.js

Wenn Sie JSON für das Protokollformat Ihrer Funktion auswählen, sendet Lambda die Protokollausgabe mit den Konsolenmethoden`console.trace`,,`console.debug`, `console.log` `console.infoconsole.error`, und `console.warn` an CloudWatch als strukturiertes JSON. Jedes JSON-Protokollobjekt enthält mindestens vier Schlüssel-Wert-Paare mit den folgenden Schlüsseln:

- `"timestamp"` – die Uhrzeit, zu der die Protokollmeldung generiert wurde
- `"level"` – die der Meldung zugewiesene Protokollebene
- `"message"` – der Inhalt der Protokollmeldung
- `"requestId"` – die eindeutige Anforderungs-ID für den Funktionsaufruf

Abhängig von der Protokollierungsmethode, die Ihre Funktion verwendet, kann dieses JSON-Objekt auch zusätzliche Schlüsselpaare enthalten. Wenn Ihre Funktion beispielsweise `console`-Methoden verwendet, um Fehlerobjekte mit mehreren Argumenten zu protokollieren, enthält das JSON-Objekt zusätzliche Schlüssel-Wert-Paare mit den Schlüsseln `errorMessage`, `errorType` und `stackTrace`.

Wenn Ihr Code bereits eine andere Logging-Bibliothek wie Powertools for verwendet AWS Lambda, um strukturierte JSON-Logs zu erstellen, müssen Sie keine Änderungen vornehmen. Lambda codiert Protokolle, die bereits JSON-codiert sind, nicht doppelt, sodass die Anwendungsprotokolle Ihrer Funktion weiterhin wie zuvor erfasst werden.

Weitere Informationen zur Verwendung des Pakets Powertools for AWS Lambda Logging zur Erstellung strukturierter JSON-Logs in der Laufzeit von Node.js finden Sie unter. the section called “Protokollierung”

Beispiel für Protokollausgaben im JSON-Format

Die folgenden Beispiele zeigen, wie verschiedene Protokollausgaben, die mithilfe der `console` Methoden mit einzelnen und mehreren Argumenten generiert wurden, in CloudWatch Logs erfasst werden, wenn Sie das Protokollformat Ihrer Funktion auf JSON setzen.

Im ersten Beispiel wird die `console.error`-Methode verwendet, um eine einfache Zeichenfolge auszugeben.

Example Protokollcode für Node.js

```
export const handler = async (event) => {
  console.error("This is a warning message");
  ...
}
```

Example JSON-Protokolldatensatz

```
{
    "timestamp":"2025-11-01T00:21:51.358Z",
    "level":"ERROR",
    "message":"This is a warning message",
    "requestId":"93f25699-2cbf-4976-8f94-336a0aa98c6f"
}
```

Sie können mit den `console`-Methoden auch komplexere strukturierte Protokollnachrichten ausgeben, indem Sie entweder einzelne oder mehrere Argumente verwenden. Im nächsten Beispiel verwenden Sie `console.log`, um zwei Schlüssel-Wert-Paare mit einem einzigen Argument auszugeben. Beachten Sie, dass das `"message"` Feld im JSON-Objekt, das Lambda an CloudWatch Logs sendet, nicht stringifiziert ist.

Example Protokollcode für Node.js

```
export const handler = async (event) => {
  console.log({data: 12.3, flag: false});
  ...
}
```

Example JSON-Protokolldatensatz

```
{
```

```
    "timestamp": "2025-12-08T23:21:04.664Z",
    "level": "INFO",
    "requestId": "405a4537-9226-4216-ac59-64381ec8654a",
    "message": {
        "data": 12.3,
        "flag": false
    }
}
```

Im nächsten Beispiel verwenden Sie erneut die `console.log`-Methode, um eine Protokollausgabe zu erstellen. Diesmal verwendet die Methode zwei Argumente, eine Map mit zwei Schlüssel-Wert-Paaren und eine identifizierende Zeichenfolge. Beachten Sie, dass Lambda in diesem Fall das Feld `"message"` in eine Zeichenfolge umwandelt, da Sie zwei Argumente angegeben haben.

Example Protokollcode für Node.js

```
export const handler = async (event) => {
  console.log('Some object - ', {data: 12.3, flag: false});
  ...
}
```

Example JSON-Protokolldatensatz

```
{
    "timestamp": "2025-12-08T23:21:04.664Z",
    "level": "INFO",
    "requestId": "405a4537-9226-4216-ac59-64381ec8654a",
    "message": "Some object -  { data: 12.3, flag: false }"
}
```

Lambda weist Ausgaben, die mit `console.log` generiert wurden, der Protokollebene INFO zu.

Das letzte Beispiel zeigt, wie Fehlerobjekte mithilfe der Methoden in CloudWatch Logs ausgegeben werden können. `console` Beachten Sie, dass Lambda die Felder `errorMessage`, `errorType` und `stackTrace` zur Protokollausgabe hinzufügt, wenn Sie Fehlerobjekte mit mehreren Argumenten protokollieren.

Example Protokollcode für 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 JSON-Protokolldatensatz

```
{
    "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)"
    ]
}
```

Bei der Protokollierung mehrerer Fehlertypen werden die zusätzlichen Felder `errorMessage`, `errorType` und `stackTrace` aus dem ersten Fehlertyp extrahiert, der an die `console`-Methode übergeben wurde.

Verwenden von Clientbibliotheken im Embedded Metric Format (EMF) mit strukturierten JSON-Protokollen

AWS stellt Open-Source-Clientbibliotheken für Node.js bereit, mit denen Sie Logs im Embedded Metric Format (EMF) erstellen können. Wenn Sie bereits über Funktionen verfügen, die diese Bibliotheken verwenden, und Sie das Protokollformat Ihrer Funktion in JSON ändern, werden die von Ihrem Code ausgegebenen Metriken CloudWatch möglicherweise nicht mehr erkannt.

Wenn Ihr Code derzeit EMF-Protokolle direkt mit `console.log` oder mithilfe von Powertools for AWS Lambda (TypeScript) ausgibt, können CloudWatch Sie diese auch nicht analysieren, wenn Sie das Protokollformat Ihrer Funktion in JSON ändern.

> **Important**
>
> Um sicherzustellen, dass die EMF-Protokolle Ihrer Funktionen weiterhin ordnungsgemäß analysiert werden, aktualisieren Sie Ihre EMF- und Powertools for CloudWatch -Bibliotheken auf die neuesten Versionen. AWS Lambda Wenn Sie zum JSON-Protokollformat wechseln, empfehlen wir Ihnen zudem, Tests durchzuführen, um die Kompatibilität mit den eingebetteten Metriken Ihrer Funktion sicherzustellen. Wenn Ihr Code EMF-Protokolle direkt mit `console.log` ausgibt, ändern Sie Ihren Code so, dass diese Metriken direkt in `stdout` ausgegeben werden, wie im folgenden Codebeispiel gezeigt.

Example Code, der eingebettete Metriken in **`stdout`** ausgibt

```
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")
```

Verwenden der Filterung auf Protokollebene mit Node.js

AWS Lambda Um Ihre Anwendungsprotokolle nach ihrer Protokollebene zu filtern, muss Ihre Funktion Protokolle im JSON-Format verwenden. Sie können dies auf zwei Arten erreichen:

- Erstellen Sie Protokollausgaben mit den Standardmethoden der Konsole und konfigurieren Sie Ihre Funktion so, dass sie die JSON-Protokollformatierung verwendet. AWS Lambda filtert dann Ihre Protokollausgaben anhand des Schlüsselwertpaars „Level“ im JSON-Objekt, wie unter beschriebenthe section called “Verwenden strukturierter JSON-Protokolle mit Node.js”. Informationen zur Konfiguration des Protokollformats Ihrer Funktion finden Sie unter the section called “Konfigurieren erweiterter Protokollierungsoptionen für Lambda-Funktionen”.
- Verwenden Sie eine andere Protokollierungsbibliothek oder Methode, um strukturierte JSON-Protokolle in Ihrem Code zu erstellen, die ein „level“-Schlüssel-Wert-Paar enthalten, das die Ebene der Protokollausgabe definiert. Sie können Powertools zum Beispiel verwenden, AWS Lambda um strukturierte JSON-Protokollausgaben aus Ihrem Code zu generieren. Weitere Informationen zur Verwendung von Powertools mit der Laufzeit Node.js finden Sie unter the section called “Protokollierung”.

  Damit Lambda die Protokolle Ihrer Funktion filtern kann, müssen Sie auch ein `"timestamp"`-Schlüssel-Wert-Paar in Ihre JSON-Protokollausgabe aufnehmen. Die Uhrzeit muss im gültigen RFC 3339-Zeitstempelformat angegeben werden. Wenn Sie keinen gültigen Zeitstempel angeben, weist Lambda dem Protokoll die Stufe INFO zu und fügt einen Zeitstempel für Sie hinzu.

Wenn Sie Ihre Funktion für die Filterung auf Protokollebene konfigurieren, wählen Sie aus den folgenden Optionen die Ebene der CloudWatch Protokolle aus AWS Lambda , die Sie an Logs senden möchten:

Protokollebene	Standardnutzung
TRACE (am detailliertesten)	Die detailliertesten Informationen, die verwendet werden, um den Ausführungspfad Ihres Codes nachzuverfolgen
DEBUG	Detaillierte Informationen für das System-Debugging
INFO	Meldungen, die den normalen Betrieb Ihrer Funktion erfassen
WARN	Meldungen über mögliche Fehler, die zu unerwartetem Verhalten führen können, wenn sie nicht behoben werden
ERROR	Meldungen über Probleme, die verhindern, dass der Code wie erwartet funktioniert
FATAL (am wenigsten Details)	Meldungen über schwerwiegende Fehler, die dazu führen, dass die Anwendung nicht mehr funktioniert

Lambda sendet Protokolle der ausgewählten Stufe und niedriger bis CloudWatch. Wenn Sie beispielsweise die Protokollebene WARN konfigurieren, sendet Lambda Protokolle, die den Stufen WARN, ERROR und FATAL entsprechen.

Anzeigen von Protokollen in der Lambda-Konsole

Sie können die Lambda-Konsole verwenden, um die Protokollausgabe nach dem Aufrufen einer Lambda-Funktion anzuzeigen.

Wenn Ihr Code über den eingebetteten Code-Editor getestet werden kann, finden Sie Protokolle in den Ausführungsergebnissen. Wenn Sie das Feature Konsolentest verwenden, um eine Funktion aufzurufen, finden Sie die Protokollausgabe im Abschnitt Details.

Protokolle in der Konsole anzeigen CloudWatch

Sie können die CloudWatch Amazon-Konsole verwenden, um Protokolle für alle Lambda-Funktionsaufrufe anzuzeigen.

Um Protokolle auf der Konsole anzuzeigen CloudWatch

1. Öffnen Sie die Seite Protokollgruppen auf der CloudWatch Konsole.
2. Wählen Sie die Protokollgruppe für Ihre Funktion (***your-function-name***/aws/lambda/).
3. Wählen Sie eine Protokollstream aus.

Jeder Protokoll-Stream entspricht einer Instance Ihrer Funktion. Ein Protokollstream wird angezeigt, wenn Sie Ihre Lambda-Funktion aktualisieren und wenn zusätzliche Instanzen für gleichzeitige Aufrufe erstellt werden. Um Logs für einen bestimmten Aufruf zu finden, empfehlen wir, Ihre Funktion mit zu instrumentieren. AWS X-Ray X-Ray erfasst Details zu der Anforderung und dem Protokollstream in der Trace.

Logs mit dem Befehl () anzeigen AWS Command Line InterfaceAWS CLI

Das AWS CLI ist ein Open-Source-Tool, mit dem Sie mithilfe von Befehlen in Ihrer Befehlszeilen-Shell mit AWS Diensten interagieren können. Zur Durchführung der Schritte in diesem Abschnitt benötigen Sie AWS CLI Version 2.

Sie können die AWS CLI verwenden, um Protokolle für einen Aufruf mit der `--log-type`-Befehlsoption abzurufen. Die Antwort enthält das Feld `LogResult`, das bis zu 4 KB base64-verschlüsselte Protokolle aus dem Aufruf enthält.

Example eine Log-ID abrufen

Das folgende Beispiel zeigt, wie eine Protokoll-ID aus dem `LogResult`-Feld für eine Funktion namens `my-function` abgerufen wird.

```
aws lambda invoke --function-name my-function out --log-type Tail
```

Die Ausgabe sollte folgendermaßen aussehen:

```
{
    "StatusCode": 200,
```

```
    "LogResult":
 "U1RBUlQgUmVxdWVzdElkOiA4N2QwNDRiOC1mMTU0LTExZTgtOGNkYS0yOTc0YzVlNGZiMjEgVmVyc2lvb...",
    "ExecutedVersion": "$LATEST"
}
```

Example entschlüsseln der Protokolle

Verwenden Sie in derselben Eingabeaufforderung das `base64`-Dienstprogramm, um die Protokolle zu entschlüsseln. Das folgende Beispiel zeigt, wie Base64-codierte Logs für abgerufen werde `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
```

Die cli-binary-format Option ist erforderlich, wenn Sie AWS CLI Version 2 verwenden. Um dies zur Standardeinstellung zu machen, führen Sie `aws configure set cli-binary-format raw-in-base64-out` aus. Weitere Informationen finden Sie unter Von AWS CLI unterstützte globale Befehlszeilenoptionen im AWS Command Line Interface -Benutzerhandbuch für Version 2.

Die Ausgabe sollte folgendermaßen aussehen:

```
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
```

Das `base64`-Dienstprogramm ist unter Linux, macOS und Ubuntu auf Windowsverfügbar. macOS-Benutzer müssen möglicherweise `base64 -D` verwenden.

Example get-logs.sh-Skript

Verwenden Sie in derselben Eingabeaufforderung das folgende Skript, um die letzten fünf Protokollereignisse herunterzuladen. Das Skript verwendet `sed` zum Entfernen von Anführungszeichen aus der Ausgabedatei und wechselt 15 Sekunden lang in den Ruhezustand, um Zeit einzuräumen, damit Protokolle verfügbar werden können. Die Ausgabe enthält die Antwort von Lambda und die `get-log-events`Ausgabe des Befehls.

Kopieren Sie den Inhalt des folgenden Codebeispiels und speichern Sie es in Ihrem Lambda-Projektverzeichnis unter `get-logs.sh`.

Die cli-binary-format Option ist erforderlich, wenn Sie AWS CLI Version 2 verwenden. Um dies zur Standardeinstellung zu machen, führen Sie `aws configure set cli-binary-format raw-in-base64-out` aus. Weitere Informationen finden Sie unter Von AWS CLI unterstützte globale Befehlszeilenoptionen im AWS Command Line Interface -Benutzerhandbuch für 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 und Linux (nur diese Systeme)

In derselben Eingabeaufforderung müssen macOS- und Linux-Benutzer möglicherweise den folgenden Befehl ausführen, um sicherzustellen, dass das Skript ausführbar ist.

```
chmod -R 755 get-logs.sh
```

Example die letzten fünf Protokollereignisse abrufen

Führen Sie an derselben Eingabeaufforderung das folgende Skript aus, um die letzten fünf Protokollereignisse abzurufen.

```
./get-logs.sh
```

Die Ausgabe sollte folgendermaßen aussehen:

```
{
    "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"
}
```


Löschen von Protokollen

Wenn Sie eine Funktion löschen, werden Protokollgruppen nicht automatisch gelöscht. Um das unbegrenzte Speichern von Protokollen zu vermeiden, löschen Sie die Protokollgruppe oder konfigurieren Sie eine Aufbewahrungszeitraum nach dem Protokolle automatisch gelöscht werden.

Instrumentierung von Node.js Code in AWS Lambda

Lambda lässt sich integrieren AWS X-Ray , um Ihnen zu helfen, Lambda-Anwendungen zu verfolgen, zu debuggen und zu optimieren. Sie können mit X-Ray eine Anforderung verfolgen, während sie Ressourcen in Ihrer Anwendung durchläuft, die Lambda-Funktionen und andere AWS -Services enthalten können.

Um Protokollierungsdaten an X-Ray zu senden, können Sie eine von zwei SDK-Bibliotheken verwenden:

- AWS Distro for OpenTelemetry (ADOT) — Eine sichere, produktionsbereite und AWS unterstützte Distribution des () SDK. OpenTelemetry OTel
- AWS X-Ray-SDK für Node.js – Ein SDK zum Generieren und Senden von Nachverfolgungsdaten an X-Ray.

Jedes SDKs bietet Möglichkeiten, Ihre Telemetriedaten an den X-Ray-Dienst zu senden. Sie können dann mit X-Ray die Leistungsmetriken Ihrer Anwendung anzeigen, filtern und erhalten, um Probleme und Möglichkeiten zur Optimierung zu identifizieren.

> **Important**
>
> X-Ray und Powertools für AWS Lambda SDKs sind Teil einer eng integrierten Instrumentierungslösung, die von AWS angeboten wird. Die ADOT Lambda Layers sind Teil eines branchenweiten Standards für die Verfolgung von Instrumenten, die im Allgemeinen mehr Daten erfassen, aber möglicherweise nicht für alle Anwendungsfälle geeignet sind. Sie können die end-to-end Ablaufverfolgung in X-Ray mit beiden Lösungen implementieren. Weitere Informationen zur Auswahl zwischen beiden finden Sie unter Wählen zwischen der AWS Distribution für Open Telemetry und X-Ray. SDKs

Sections

- Verwenden von ADOT zur Instrumentierung Ihrer Node.js-Funktionen
- Verwenden des X-Ray-SDK zum Instrumentieren Ihrer Node.js-Funktionen
- Aktivieren der Nachverfolgung mit der Lambda-Konsole
- Aktivieren der Nachverfolgung mit der Lambda-API
- Die Ablaufverfolgung wird aktiviert mit CloudFormation

- Interpretieren einer X-Ray-Nachverfolgung
- Laufzeitabhängigkeiten in einer Ebene speichern (X-Ray-SDK)

Verwenden von ADOT zur Instrumentierung Ihrer Node.js-Funktionen

ADOT bietet vollständig verwaltete Lambda-Schichten, die alles zusammenfassen, was Sie für die Erfassung von Telemetriedaten mithilfe des SDK benötigen. OTel Indem Sie diese Ebene verwenden, können Sie Ihre Lambda-Funktionen instrumentieren, ohne einen Funktionscode ändern zu müssen. Sie können Ihren Layer auch so konfigurieren, dass er eine benutzerdefinierte Initialisierung von durchführt. OTel Weitere Informationen finden Sie unter Benutzerdefinierte Konfiguration für den ADOT Collector auf Lambda in der ADOT-Dokumentation.

Für Node.js-Laufzeiten können Sie den AWS -verwalteten Lambda-Ebene für ADOT Javascript hinzufügen, um Ihre Funktionen automatisch zu instrumentieren. Eine ausführliche Anleitung zum Hinzufügen dieser Ebene finden Sie unter AWS Distro for OpenTelemetry Lambda Support for JavaScript in der ADOT-Dokumentation.

Verwenden des X-Ray-SDK zum Instrumentieren Ihrer Node.js-Funktionen

Um Details zu Aufrufen aufzuzeichnen, die Ihre Lambda-Funktion an andere Ressourcen in Ihrer Anwendung vornimmt, können Sie auch AWS X-Ray-SDK für Node.js verwenden. Um das SDK zu erhalten, fügen Sie das `aws-xray-sdk-core`-Paket den Abhängigkeiten Ihrer Anwendung hinzu.

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"
  }
}
```

Um AWS SDK-Clients in Version 3 zu instrumentieren AWS SDK für JavaScript , schließen Sie die Client-Instanz mit der Methode ein. `captureAWSv3Client`

Example blank- nodejs/function/index .js — Einen AWS SDK-Client verfolgen

```
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 => {
  ...
```

Die Lambda-Laufzeit legt zur Konfiguration des X-Ray-SDK einige Umgebungsvariablen fest. Beispielsweise setzt Lambda `AWS_XRAY_CONTEXT_MISSING` auf `LOG_ERROR`, um Laufzeitfehler aus dem X-Ray-SDK zu vermeiden. Um eine benutzerdefinierte Strategie für fehlenden Kontext festzulegen, überschreiben Sie die Umgebungsvariable in der Funktionskonfiguration so, dass sie keinen Wert aufweist. Dann können Sie die Strategie für fehlenden Kontext programmgesteuert festlegen.

Example Beispiel-Initialisierungscode

```
const AWSXRay = require('aws-xray-sdk-core');

// Configure the context missing strategy to do nothing
AWSXRay.setContextMissingStrategy(() => {});
```

Weitere Informationen finden Sie unter the section called “Umgebungsvariablen”.

Aktivieren Sie nach Hinzufügen der richtigen Abhängigkeiten die Nachverfolgung in der Konfiguration Ihrer Funktion über die Lambda-Konsole oder die API.

Aktivieren der Nachverfolgung mit der Lambda-Konsole

Gehen Sie folgendermaßen vor, um die aktive Nachverfolgung Ihrer Lambda-Funktion mit der Konsole umzuschalten:

So aktivieren Sie die aktive Nachverfolgung

1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.
2. Wählen Sie eine Funktion aus.
3. Wählen Sie Configuration (Konfiguration) und dann Monitoring and operations tools (Überwachungs- und Produktionstools).
4. Wählen Sie unter Zusätzliche Überwachungstools die Option Bearbeiten aus.
5. Wählen Sie unter CloudWatch Anwendungssignale und AWS X-Ray die Option Enable for Lambda Service Traces aus.
6. Wählen Sie Speichern.

Aktivieren der Nachverfolgung mit der Lambda-API

Konfigurieren Sie die Ablaufverfolgung für Ihre Lambda-Funktion mit dem AWS CLI oder AWS SDK und verwenden Sie die folgenden API-Operationen:

- UpdateFunctionConfiguration
- GetFunctionConfiguration
- CreateFunction

Der folgende AWS CLI Beispielbefehl aktiviert die aktive Ablaufverfolgung für eine Funktion namens my-function.

```
aws lambda update-function-configuration --function-name my-function \
--tracing-config Mode=Active
```

Der Ablaufverfolgungsmodus ist Teil der versionsspezifischen Konfiguration, wenn Sie eine Version Ihrer Funktion veröffentlichen. Sie können den Ablaufverfolgungsmodus für eine veröffentlichte Version nicht ändern.

Die Ablaufverfolgung wird aktiviert mit CloudFormation

Um die Ablaufverfolgung für eine `AWS::Lambda::Function` Ressource in einer CloudFormation Vorlage zu aktivieren, verwenden Sie die `TracingConfig` Eigenschaft.

Example function-inline.yml – Ablaufverfolgungskonfiguration

```
Resources:
  function:
    Type: AWS::Lambda::Function
    Properties:
      TracingConfig:
        Mode: Active
      ...
```

Verwenden Sie für eine `AWS::Serverless::Function` Ressource AWS Serverless Application Model (AWS SAM) die `Tracing` Eigenschaft.

Example template.yml – Ablaufverfolgungskonfiguration

```
Resources:
  function:
    Type: AWS::Serverless::Function
    Properties:
      Tracing: Active
      ...
```

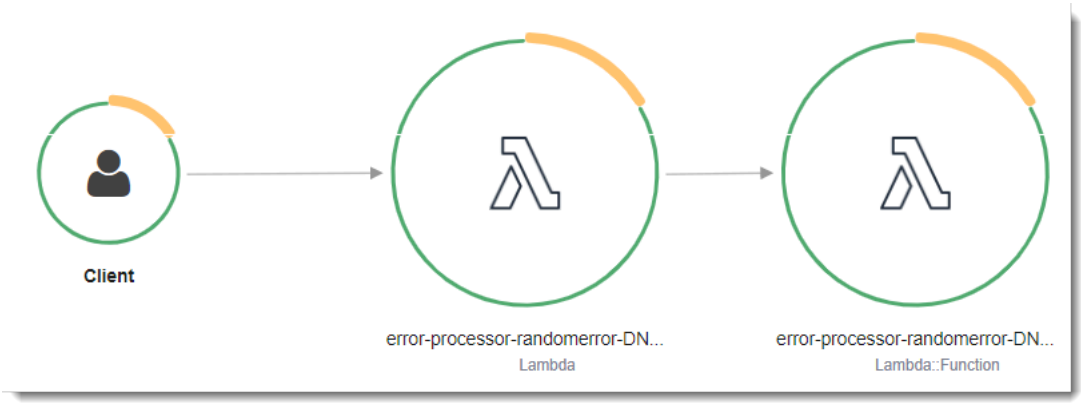
Interpretieren einer X-Ray-Nachverfolgung

Ihre Funktion benötigt die Berechtigung zum Hochladen von Trace-Daten zu X-Ray.
Wenn Sie die aktive Nachverfolgung in der Lambda-Konsole aktivieren, fügt Lambda der Ausführungsrolle Ihrer Funktion die erforderlichen Berechtigungen hinzu. Andernfalls fügen Sie die AWSXRayDaemonWriteAccessRichtlinie der Ausführungsrolle hinzu.

Nachdem Sie die aktive Nachverfolgung konfiguriert haben, können Sie bestimmte Anfragen über Ihre Anwendung beobachten. Das X-Ray-Service-Diagramm zeigt Informationen über Ihre Anwendung und alle ihre Komponenten an. Das folgende Beispiel zeigt eine Anwendung mit zwei Funktionen. Die primäre Funktion verarbeitet Ereignisse und gibt manchmal Fehler zurück. Die zweite Funktion an oberster Stelle verarbeitet Fehler, die in der Protokollgruppe der ersten auftreten, und verwendet das AWS SDK, um X-Ray, Amazon Simple Storage Service (Amazon S3) und Amazon CloudWatch Logs aufzurufen.

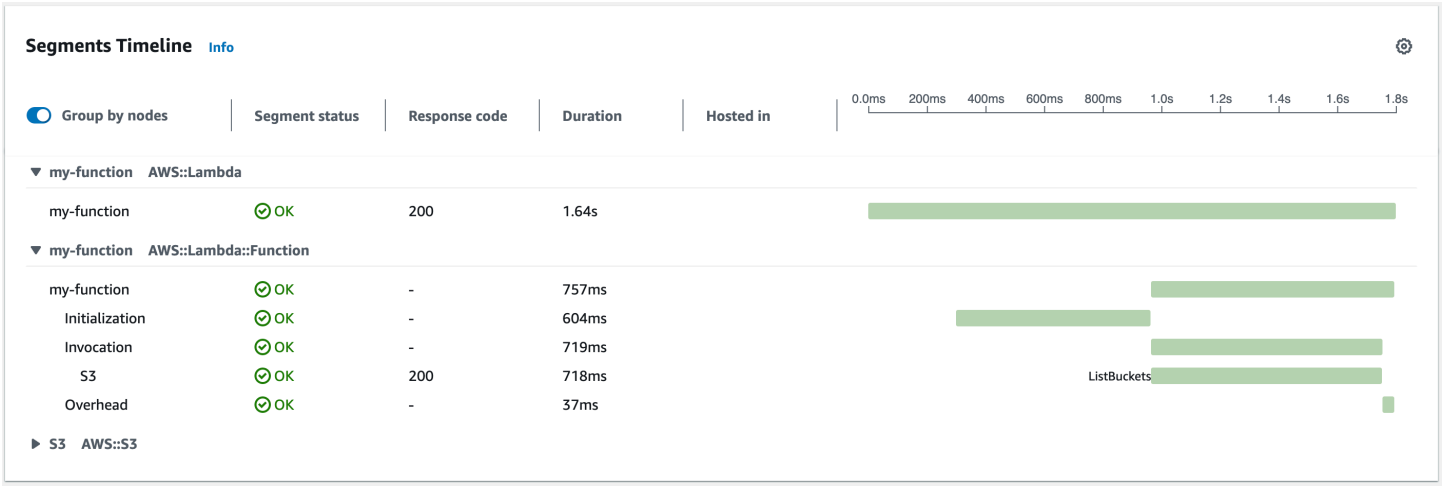
X-Ray verfolgt nicht alle Anfragen an Ihre Anwendung nach. X-Ray wendet einen Sampling-Algorithmus an, um sicherzustellen, dass die Nachverfolgung effizient ist, und stellt dennoch ein repräsentatives Beispiel aller Anfragen bereit. Die Samplingrate beträgt 1 Anforderung pro Sekunde und 5 Prozent aller weiteren Anforderungen. Sie können die X-Ray-Samplingrate nicht für Ihre Funktionen konfigurieren.

In X-Ray, zeichnet eine Ablaufverfolgung Informationen zu einer Anforderung auf, die von einem oder mehreren Services verarbeitet wird. Lambda zeichnet 2 Segmente pro Trace auf, wodurch zwei Knoten im Dienstgraphen entstehen. In der folgenden Abbildung werden diese beiden Knoten hervorgehoben:



Der erste Knoten auf der linken Seite stellt den Lambda-Service dar, der die Aufrufanforderung empfängt. Der zweite Knoten stellt Ihre spezifische Lambda-Funktion dar. Das folgende Beispiel zeigt eine Nachverfolgung mit diesen zwei Segmenten. Beide heißen my-function, aber einer hat einen Ursprung von `AWS::Lambda` und der andere hat einen Ursprung von `AWS::Lambda::Function`. Wenn das `AWS::Lambda`-Segment einen Fehler anzeigt, hatte der Lambda-Service ein Problem.

Wenn das `AWS::Lambda::Function`-Segment einen Fehler anzeigt, ist bei Ihrer Funktion ein Problem aufgetreten.



Dieses Beispiel erweitert das `AWS::Lambda::Function`-Segment, um seine drei Untersegmente anzuzeigen.

> **Note**
>
> AWS implementiert derzeit Änderungen am Lambda-Service. Aufgrund dieser Änderungen können geringfügige Unterschiede in Struktur und Inhalt der Systemprotokollmeldungen und Trace-Segmente auftreten, die von verschiedenen Lambda-Funktionen in Ihrem AWS-Konto. Der hier gezeigte Beispiel-Trace veranschaulicht das Funktionssegment im alten Stil. Die Unterschiede zwischen den Segmenten im alten und im neuen Stil werden in den folgenden Abschnitten beschrieben.
>
> Diese Änderungen werden in den kommenden Wochen umgesetzt, und alle Funktionen AWS-Regionen außer China und den GovCloud Regionen werden auf die Verwendung der Protokollnachrichten und Trace-Segmente im neuen Format umgestellt.

Das Funktionssegment im alten Stil enthält die folgenden Untersegmente:

- Initialisierung – Stellt die Zeit dar, die für das Laden Ihrer Funktion und das Ausführen des Initialisierungscodes aufgewendet wurde. Dieses Untersegment erscheint nur für das erste Ereignis, das jede Instance Ihrer Funktion verarbeitet.
- Invocation (Aufruf) – Stellt die Zeit dar, die beim Ausführen Ihres Handler-Codes vergeht.
- Overhead (Aufwand) – Stellt die Zeit dar, die von der Lambda-Laufzeitumgebung bei der Verarbeitung des nächsten Ereignisses verbraucht wird.

Das Funktionssegment im neuen Stil enthält kein `Invocation`-Untersegment. Stattdessen werden Kundenuntersegmente direkt an das Funktionssegment angehängt. Weitere Informationen über die Struktur der alten und neuen Funktionssegmente finden Sie unter the section called “Grundlegendes zu X-Ray-Ablaufverfolgungen”.

Sie können auch HTTP-Clients instrumentieren, SQL-Abfragen aufzeichnen und benutzerdefinierte Untersegmente mit Anmerkungen und Metadaten erstellen. Weitere Informationen finden Sie unter AWS X-Ray-SDK für Node.js im AWS X-Ray -Entwicklerhandbuch.

> **Preisgestaltung**
>
> Im Rahmen des kostenlosen Kontingents können Sie X-Ray Tracing jeden Monat bis zu einem bestimmten Limit AWS kostenlos nutzen. Über den Schwellenwert hinaus berechnet X-Ray Gebühren für die Speicherung und den Abruf der Nachverfolgung. Weitere Informationen finden Sie unter AWS X-Ray Preise.

Laufzeitabhängigkeiten in einer Ebene speichern (X-Ray-SDK)

Wenn Sie das X-Ray-SDK verwenden, um AWS SDK-Clients Ihren Funktionscode zu instrumentieren, kann Ihr Bereitstellungspaket ziemlich umfangreich werden. Um Laufzeitabhängigkeiten bei jeder Aktualisierung des Funktionscodes zu vermeiden, verpacken Sie das X-Ray-SDK in einer Lambda-Ebene.

Das folgende Beispiel zeigt eine `AWS::Serverless::LayerVersion`-Ressource, die das AWS X-Ray-SDK für Node.js speichert.

Example template.yml – Abhängigkeitenebene

```
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
```

Bei dieser Konfiguration aktualisieren Sie die Bibliotheksebene nur, wenn Sie Ihre Laufzeitabhängigkeiten ändern. Da das Funktionsbereitstellungspaket nur Ihren Code enthält, kann dies dazu beitragen, die Upload-Zeiten zu reduzieren.

Das Erstellen einer Ebene für Abhängigkeiten erfordert Build-Konfigurationsänderungen, um das Ebenen-Archiv vor der Bereitstellung zu generieren. Ein funktionierendes Beispiel finden Sie in der Beispielanwendung blank-nodejs .

Aufbau von Lambda-Funktionen mit TypeScript

Sie können die Laufzeit von Node.js verwenden, um TypeScript Code darin AWS Lambda auszuführen. Da Node.js TypeScript Code nicht nativ ausführt, müssen Sie Ihren TypeScript Code zuerst in das Format transpilieren. JavaScript Verwenden Sie dann die JavaScript Dateien, um Ihren Funktionscode für Lambda bereitzustellen. Ihr Code wird in einer Umgebung ausgeführt, die das AWS SDK für enthält JavaScript, mit Anmeldeinformationen aus einer AWS Identity and Access Management (IAM-) Rolle, die Sie verwalten. Weitere Informationen zu den SDK-Versionen, die in den Laufzeiten von Node.js enthalten sind, finden Sie unter the section called “SDK-Versionen, die zur Laufzeit enthalten sind”.

Lambda unterstützt die folgenden Node.js-Laufzeiten.

Name	ID	Betriebssystem	Datum der Veraltung	Blockfunktion erstellen	Blockfunktion aktualisieren
Node.js 24	`nodejs24.x`	Amazon Linux 2023	30. April 2028	1. Juni 2028	1. Juli 2028
Node.js 22	`nodejs22.x`	Amazon Linux 2023	30. April 2027	1. Juni 2027	1. Juli 2027
Node.js 20	`nodejs20.x`	Amazon Linux 2023	30. April 2026	31. August 2026	30. September 2026

Themen

- Einrichtung einer TypeScript Entwicklungsumgebung
- Typdefinitionen für Lambda
- Definieren Sie den Lambda-Funktionshandler in TypeScript
- Stellen Sie transpilierten TypeScript Code in Lambda mit ZIP-Dateiarchiven bereit
- Stellen Sie transpilierten TypeScript-Code in Lambda mit Container-Images bereit
- Verwenden des Lambda-Kontextobjekts zum Abrufen von TypeScript Funktionsinformationen
- TypeScript Lambda-Funktionen protokollieren und überwachen

- TypeScript Code nachverfolgen in AWS Lambda

Einrichtung einer TypeScript Entwicklungsumgebung

Verwenden Sie eine lokale integrierte Entwicklungsumgebung (IDE) oder einen Texteditor, um Ihren TypeScript Funktionscode zu schreiben. Sie können auf der Lambda-Konsole keinen TypeScript Code erstellen.

Sie können entweder esbuild oder den TypeScript Compiler (`tsc`) von Microsoft verwenden, um Ihren Code in zu transpilieren. TypeScript JavaScript AWS Serverless Application Model (AWS SAM) und AWS Cloud Development Kit (AWS CDK) verwenden beide „esbuild“.

Wenn Sie esbuild verwenden, beachten Sie Folgendes:

- Es gibt mehrere Vorbehalte. TypeScript
- Sie müssen Ihre TypeScript Transpilationseinstellungen so konfigurieren, dass sie der Laufzeit von Node.js entsprechen, die Sie verwenden möchten. Weitere Informationen finden Sie unter Ziel in der esbuild-Dokumentation. Ein Beispiel für eine tsconfig.json-Datei, die zeigt, wie Sie auf eine bestimmte, von Lambda unterstützte Version von Node.js abzielen, finden Sie im Repository. TypeScript GitHub
- esbuild führt keine Typ-Überprüfungen durch. Um Typen zu überprüfen, verwenden Sie den `tsc`-Compiler. Führen Sie `tsc -noEmit` aus oder fügen Sie einen `"noEmit"`-Parameter zu Ihrer tsconfig.json-Datei hinzu, wie im folgenden Beispiel gezeigt. Dies ist so konfiguriert`tsc`, dass keine Dateien ausgegeben werden. JavaScript Verwenden Sie nach der Überprüfung der Typen Esbuild, um die TypeScript Dateien in zu konvertieren. 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"]
}
```

Typdefinitionen für Lambda

Das Paket @types/aws-lambda enthält Typdefinitionen für Lambda-Funktionen. Installieren Sie dieses Paket, wenn Ihre Funktion eines der folgenden Elemente verwendet:

- Allgemeine AWS Ereignisquellen, wie zum Beispiel:
  - `APIGatewayProxyEvent`: Für Für Proxy-Integrationen in Amazon API Gateway
  - `SNSEvent`: Für Amazon-Simple-Notification-Service-Benachrichtigungen
  - `SQSEvent`: Für Amazon-Simple-Queue-Service-Nachrichten
  - `S3Event`: Für S3-Auslöserereignisse
  - `DynamoDBStreamEvent`: Für Amazon DynamoDB Streams
- Das Lambda-Kontext-Objekt
- Das Callback-Handler-Muster

Um die Lambda-Typdefinitionen zu Ihrer Funktion hinzuzufügen, installieren Sie `@types/aws-lambda` als eine Entwicklungsabhängigkeit:

```
npm install -D @types/aws-lambda
```

Importieren Sie dann die Typen aus `aws-lambda`:

```
import { Context, S3Event, APIGatewayProxyEvent } from 'aws-lambda';

export const handler = async (event: S3Event, context: Context) => {
    // Function code
};
```

Die `import ... from 'aws-lambda'`-Anweisung importiert die Typdefinitionen. Das `aws-lambda`-NPM-Paket wird nicht importiert, da es sich um ein unabhängiges Tool eines Drittanbieters handelt. Weitere Informationen finden Sie unter aws-lambda im Repository. DefinitelyTyped GitHub

> **Note**
>
> Sie benötigen @types/aws-lambda nicht, wenn Sie Ihre eigenen benutzerdefinierten Typdefinitionen verwenden. Eine Beispielfunktion, die ihren eigenen Typ für ein Ereignisobjekt definiert, finden Sie unter Beispiel für einen TypeScript Lambda-Funktionscode.

Definieren Sie den Lambda-Funktionshandler in TypeScript

Der Lambda-Funktionshandler ist die Methode in Ihrem Funktionscode, die Ereignisse verarbeitet. Wenn Ihre Funktion aufgerufen wird, führt Lambda die Handler-Methode aus. Ihre Funktion wird so lange ausgeführt, bis der Handler eine Antwort zurückgibt, beendet wird oder ein Timeout auftritt.

Auf dieser Seite wird beschrieben, wie Sie mit Lambda-Funktionshandlern arbeiten TypeScript, einschließlich Optionen für die Projekteinrichtung, Benennungskonventionen und Best Practices. Diese Seite enthält auch ein Beispiel für eine TypeScript Lambda-Funktion, die Informationen über eine Bestellung aufnimmt, eine Textdateiquittung erstellt und diese Datei in einem Amazon Simple Storage Service (Amazon S3) -Bucket ablegt. Informationen darüber, wie Sie Ihre Funktion nach dem Schreiben einsetzen können, finden Sie unter the section called “Bereitstellen von ZIP-Dateiarchiven” oder the section called “Bereitstellen von Container-Images”.

Themen

- Ihr TypeScript Projekt einrichten
- Beispiel für einen TypeScript Lambda-Funktionscode
- CommonJS- und ES-Module
- Initialisierung von Node.js
- Namenskonventionen für Handler
- Definieren Sie das Eingabeereignisobjekt und greifen Sie darauf zu
- Gültige Handlermuster für Funktionen TypeScript
- Verwenden Sie das SDK für JavaScript v3 in Ihrem Handler
- Zugriff auf Umgebungsvariablen
- Verwenden des globalen Zustands
- Bewährte Codepraktiken für TypeScript Lambda-Funktionen

Ihr TypeScript Projekt einrichten

Verwenden Sie eine lokale integrierte Entwicklungsumgebung (IDE) oder einen Texteditor, um Ihren TypeScript Funktionscode zu schreiben. Sie können auf der Lambda-Konsole keinen TypeScript Code erstellen.

Es gibt mehrere Möglichkeiten, ein TypeScript Lambda-Projekt zu initialisieren. Sie können beispielsweise ein Projekt mit `npm` erstellen oder eine AWS SAM - bzw. AWS CDK -Anwendung erstellen. So erstellen Sie das Projekt mit `npm`:

```
npm init
```

Ihr Funktionscode befindet sich in einer `.ts` Datei, die Sie bei der Erstellung in eine JavaScript Datei transpilieren. Sie können entweder esbuild oder den TypeScript Compiler (`tsc`) von Microsoft verwenden, um Ihren Code in diese Datei zu transpilieren. TypeScript JavaScript Um „esbuild“ zu verwenden, fügen Sie es als Entwicklungsabhängigkeit hinzu:

```
npm install -D esbuild
```

Ein typisches TypeScript Lambda-Funktionsprojekt folgt dieser allgemeinen Struktur:

```
/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
```

Beispiel für einen TypeScript Lambda-Funktionscode

Das folgende Beispiel für einen Lambda-Funktionscode nimmt Informationen über eine Bestellung auf, erstellt eine Textdateiquittung und platziert diese Datei in einem Amazon-S3-Bucket. Dieses Beispiel definiert einen benutzerdefinierten Ereignistyp (`OrderEvent`). Informationen zum Importieren von Typdefinitionen für AWS Ereignisquellen finden Sie unter Typdefinitionen für Lambda.

> **Note**
>
> In diesem Beispiel wird ein ES-Modul-Handler verwendet. Lambda unterstützt sowohl ES-Modul- als auch CommonJS-Handler. Weitere Informationen finden Sie unter CommonJS- und ES-Module.

Example Lambda-Funktion „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'}`);
    }
}
```

Diese `index.ts`-Datei enthält die folgenden Abschnitte des Codes:

- `import`-Block: Verwenden Sie diesen Block, um Bibliotheken einzubinden, die Ihre Lambda-Funktion benötigt, z. B. AWS SDK-Clients.
- `const s3Client`-Deklaration: Hiermit wird ein Amazon-S3-Client außerhalb der Handler-Funktion initialisiert. Dies veranlasst Lambda, diesen Code während der Initialisierungsphase auszuführen und der Client wird für die Wiederverwendung über mehrere Aufrufe hinweg aufbewahrt.
- `type OrderEvent`: Definiert die Struktur des erwarteten Eingabeereignisses.
- `export const handler`: Dies ist die Haupt-Handler-Funktion, die Lambda aufruft. Geben Sie beim Bereitstellen Ihrer Funktion `index.handler` für die Handler-Eigenschaft an. Der Wert der Eigenschaft `Handler` besteht aus dem Dateinamen und dem Namen der exportierten Handler-Methode, getrennt durch einen Punkt.
- `uploadReceiptToS3`-Funktion: Dies ist eine Hilfsfunktion, auf die von der Haupt-Handler-Funktion verwiesen wird.

Damit diese Funktion ordnungsgemäß funktioniert, muss ihre Ausführungsrolle die `s3:PutObject`-Aktion zulassen. Stellen Sie außerdem sicher, dass Sie die `RECEIPT_BUCKET`-Umgebungsvariable definieren. Nach einem erfolgreichen Aufruf sollte der Amazon-S3-Bucket eine Empfangsdatei enthalten.

CommonJS- und ES-Module

Node.js unterstützt zwei Modulsysteme: CommonJS und ECMAScript Module (ES-Module). Lambda empfiehlt die Verwendung von ES-Modulen, da sie Await auf oberster Ebene unterstützen, wodurch asynchrone Aufgaben während der Initialisierung der Ausführungsumgebung abgeschlossen werden können.

Node.js behandelt Dateien mit einer `.cjs` Dateinamenerweiterung als CommonJS-Module, während eine `.mjs` Erweiterung ES-Module bezeichnet. Standardmäßig behandelt Node.js Dateien mit der `.js` Dateinamenerweiterung als CommonJS-Module. Sie können Node.js so konfigurieren, dass `.js` Dateien als ES-Module behandelt werden, indem Sie das `type` as `module` in der `package.json` Funktionsdatei angeben. Sie können Node.js in Lambda so konfigurieren, dass automatisch erkannt wird, ob eine `.js` Datei als CommonJS- oder als ES-Modul behandelt werden soll, indem Sie der `NODE_OPTIONS` Umgebungsvariablen das `–experimental-detect-module` Flag hinzufügen. Weitere Informationen finden Sie unter Experimentelle Funktionen von Node.js.

Die folgenden Beispiele zeigen Funktionshandler, die sowohl mit ES-Modulen als auch mit CommonJS-Modulen geschrieben wurden. Die übrigen Beispiele auf dieser Seite verwenden alle ES-Module.

Initialisierung von Node.js

Node.js verwendet ein nicht blockierendes I/O Modell, das effiziente asynchrone Operationen mithilfe einer Ereignisschleife unterstützt. Wenn Node.js beispielsweise einen Netzwerkaufruf tätigt, verarbeitet die Funktion weiterhin andere Operationen, ohne eine Netzwerkantwort zu blockieren. Wenn die Netzwerkantwort empfangen wird, wird sie in die Rückrufwarteschlange gestellt. Aufgaben aus der Warteschlange werden verarbeitet, wenn die aktuelle Aufgabe abgeschlossen ist.

Lambda empfiehlt die Verwendung von Await auf oberster Ebene, damit asynchrone Aufgaben, die während der Initialisierung der Ausführungsumgebung initiiert wurden, während der Initialisierung abgeschlossen werden. Asynchrone Aufgaben, die während der Initialisierung nicht abgeschlossen werden, werden normalerweise beim ersten Funktionsaufruf ausgeführt. Dies kann zu unerwartetem Verhalten oder Fehlern führen. Beispielsweise kann Ihre Funktionsinitialisierung einen Netzwerkaufruf ausführen, um einen Parameter aus dem AWS Parameterspeicher abzurufen. Wenn diese Aufgabe während der Initialisierung nicht abgeschlossen wird, kann der Wert während eines Aufrufs Null sein. Es kann auch zu einer Verzögerung zwischen der Initialisierung und dem Aufruf kommen, was bei zeitkritischen Vorgängen zu Fehlern führen kann. Insbesondere AWS Serviceabrufe können sich auf zeitkritische Anforderungssignaturen stützen, was dazu führt, dass Serviceabrufe fehlschlagen, wenn der Anruf während der Initialisierungsphase nicht abgeschlossen wird. Das Abschließen von

Aufgaben während der Initialisierung verbessert in der Regel die Kaltstartleistung und die Leistung beim ersten Aufrufen, wenn Provisioned Concurrency verwendet wird. Weitere Informationen finden Sie in unserem Blogbeitrag Using Node.js ES modules and top level await in. AWS Lambda

Namenskonventionen für Handler

Wenn Sie eine Funktion konfigurieren, besteht der Wert der Handler-Einstellung aus dem Dateinamen und dem Namen der exportierten Handler-Methode, getrennt durch einen Punkt. Der Standardwert für Funktionen, die in der Konsole erstellt werden und für die Beispiele in diesem Handbuch ist `index.handler`. Dies deutet auf die `handler`-Methode hin, die aus der `index.js`- oder `index.mjs`-Datei wurde.

Wenn Sie eine Funktion in der Konsole mit einem anderen Dateinamen oder Funktionshandlernamen erstellen, müssen Sie den Standardhandlernamen bearbeiten.

So ändern Sie den Funktionshandlernamen (Konsole)

1. Öffnen Sie die Seite Functions (Funktionen) der Lambda-Konsole und wählen Sie eine Funktion aus.
2. Wählen Sie die Registerkarte Code (Code).
3. Scrollen Sie nach unten zum Bereich Laufzeiteinstellungen und wählen Sie Bearbeiten.
4. Geben Sie unter Handler den neuen Namen für Ihren Funktionshandler ein.
5. Wählen Sie Speichern.

Definieren Sie das Eingabeereignisobjekt und greifen Sie darauf zu

JSON ist das gebräuchlichste und standardmäßigste Eingabeformat für Lambda-Funktionen. In diesem Beispiel erwartet die Funktion eine Eingabe ähnlich der folgenden:

```
{
    "order_id": "12345",
    "amount": 199.99,
    "item": "Wireless Headphones"
}
```

Wenn Sie mit Lambda-Funktionen in arbeiten TypeScript, können Sie die Form des Eingabeereignisses mithilfe eines Typs oder einer Schnittstelle definieren. In diesem Beispiel definieren wir die Ereignisstruktur mithilfe eines Typs:

```
type OrderEvent = {
    order_id: string;
    amount: number;
    item: string;
}
```

Nachdem Sie den Typ oder die Schnittstelle definiert haben, verwenden Sie ihn bzw. sie in der Signatur Ihres Handlers, um die Typsicherheit zu gewährleisten:

```
export const handler = async (event: OrderEvent): Promise<string> => {
```

TypeScript Überprüft während der Kompilierung, ob das Ereignisobjekt die erforderlichen Felder mit den richtigen Typen enthält. Der TypeScript Compiler meldet beispielsweise einen Fehler, wenn Sie versuchen, eine Zahl oder `event.amount` eine Zeichenfolge zu verwenden`event.order_id`.

Gültige Handlermuster für Funktionen TypeScript

Wir empfehlen, dass Sie async/await verwenden, um den Funktionshandler zu deklarieren, anstatt Callbacks zu verwenden. Async/await ist eine präzise und lesbare Methode, asynchronen Code zu schreiben, ohne dass verschachtelte Callbacks oder verkettete Zusagen erforderlich sind. Mit Async/Await können Sie Code schreiben, der sich wie synchroner Code liest, aber dennoch asynchron und blockierungsfrei ist.

Für die Beispiele in diesem Abschnitt wird der Typ `S3Event` verwendet. Sie können jedoch alle anderen AWS Ereignistypen im Paket @types /aws-lambda verwenden oder Ihren eigenen Ereignistyp definieren. Verwenden von Typen aus @types/aws-lambda:

1. Fügen Sie das Paket @types/aws-lambda als Entwicklungsabhängigkeit hinzu:

   ```
   npm install -D @types/aws-lambda
   ```

2. Importieren Sie die Typen, die Sie benötigen, wie `Context`, `S3Event` oder `Callback`.

asynchrone Funktionshandler (empfohlen)

Das `async`-Schlüsselwort kennzeichnet eine Funktion als asynchron, und das `await`-Schlüsselwort unterbricht die Ausführung der Funktion, bis `Promise` aufgelöst ist. Der Handler akzeptiert die folgenden Argumente:

- `event`: Enthält die an Ihre Funktion übergebenen Eingabedaten.
- `context`: Enthält Informationen über den Aufruf, die Funktion und die Ausführungsumgebung. Weitere Informationen finden Sie unter Verwenden des Lambda-Kontextobjekts zum Abrufen von TypeScript Funktionsinformationen.

Hier sind die gültigen Signaturen für das async/await Muster:

```
export const handler = async (event: S3Event): Promise<void> => { };
```

```
export const handler = async (event: S3Event, context: Context): Promise<void> => { };
```

> **Note**
>
> Bei der asynchronen Verarbeitung von Arrays von Elementen sollten Sie „Await“ mit `Promise.all` verwenden, um sicherzustellen, dass alle Vorgänge abgeschlossen werden. Methoden wie `forEach` warten nicht darauf, dass asynchrone Callbacks abgeschlossen werden. Weitere Informationen finden Sie unter Array.prototype.forEach () in der Mozilla-Dokumentation.

Handler für synchrone Funktionen

Wenn Ihre Funktion keine asynchronen Aufgaben ausführt, können Sie einen synchronen Funktionshandler verwenden, der eine der folgenden Funktionssignaturen verwendet:

```
export const handler = (event: S3Event): void => { };
```

```
export const handler = (event: S3Event, context: Context): void => { };
```

Handler für Antwort-Streaming-Funktionen

Lambda unterstützt Response-Streaming mit Node.js. Handler für Antwort-Streaming-Funktionen verwenden den Dekorator AWSLambda.streamifyResponse () und verwenden drei Parameter: event, responseStream und context. Die Funktionssignatur lautet:

```
export const handler = awslambda.streamifyResponse(async (event: APIGatewayProxyEvent,
 responseStream: NodeJS.WritableStream, context: Context) => { });
```

Weitere Informationen finden Sie unter Antwortstreaming für Lambda-Funktionen.

Callback-basierte Funktionshandler

> **Note**
>
> Callback-basierte Funktionshandler werden nur bis zu Node.js 22 unterstützt. Ab Node.js 24 sollten asynchrone Aufgaben mithilfe von asynchronen Funktionshandlern implementiert werden.

Callback-basierte Funktionshandler können die Event-, Kontext- und Callback-Argumente verwenden. Das Callback-Argument erwartet `Error` und eine Antwort, die JSON-serialisierbar sein muss.

Hier ist die gültige Signatur für das Callback-Handler-Muster:

```
export const handler = (event: S3Event, context: Context, callback: Callback<void>):
 void => { };
```

Die Funktion wird so lange ausgeführt, bis die Ereignisschleife leer ist oder ein Timeout auftritt. Die Antwort wird erst an den Aufrufer gesendet, wenn alle Ereignisschleifenaufgaben abgeschlossen sind. Wenn eine Zeitüberschreitung der Funktion auftritt, wird stattdessen ein Fehler zurückgegeben. Sie können die Laufzeit so konfigurieren, dass die Antwort sofort gesendet wird, indem Sie den Kontext festlegen. callbackWaitsForEmptyEventLoopauf „Falsch“ setzen.

Example TypeScript Funktion mit Rückruf

Das folgende Beispiel verwendet `APIGatewayProxyCallback`, einen Callback-Typ, der speziell für API-Gateway-Integrationen gilt. Die meisten AWS Ereignisquellen verwenden den generischen `Callback` Typ, der in den obigen Signaturen gezeigt wird.

```
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',
        }),
    });
};
```

Verwenden Sie das SDK für JavaScript v3 in Ihrem Handler

Oft verwenden Sie Lambda-Funktionen, um mit anderen AWS Ressourcen zu interagieren oder diese zu aktualisieren. Die einfachste Art, eine Schnittstelle zu diesen Ressourcen herzustellen, ist die Verwendung von AWS SDK für JavaScript. Alle unterstützten Lambda Node.js Runtimes enthalten das SDK für JavaScript Version 3. Wir empfehlen jedoch dringend, dass Sie die benötigten AWS SDK-Clients in Ihr Bereitstellungspaket aufnehmen. Dadurch wird die Abwärtskompatibilität bei zukünftigen Lambda-Laufzeit-Updates maximiert.

Um Ihrer Funktion SDK-Abhängigkeiten hinzuzufügen, verwenden Sie den `npm install`-Befehl für die spezifischen SDK-Clients, die Sie benötigen. Im Beispielcode haben wir den Amazon-S3-Client verwendet. Fügen Sie diese Abhängigkeit hinzu, indem Sie den folgenden Befehl in dem Verzeichnis ausführen, das Ihre `package.json`-Datei enthält:

```
npm install @aws-sdk/client-s3
```

Importieren Sie im Funktionscode den Client und die Befehle, die Sie benötigen, wie in der Beispielfunktion gezeigt:

```
import { S3Client, PutObjectCommand } from '@aws-sdk/client-s3';
```

Initialisieren Sie dann einen Amazon-S3-Client:

```
const s3Client = new S3Client();
```

In diesem Beispiel haben wir unseren Amazon-S3-Client außerhalb der Haupt-Handler-Funktion initialisiert, um zu vermeiden, dass wir ihn bei jedem Aufruf unserer Funktion initialisieren müssen. Nachdem Sie Ihren SDK-Client initialisiert haben, können Sie ihn verwenden, um API-Aufrufe für diesen AWS Dienst zu tätigen. Der Beispielcode ruft die Amazon S3 PutObjectS3-API-Aktion wie folgt auf:

```
const command = new PutObjectCommand({
    Bucket: bucketName,
    Key: key,
```

```
    Body: receiptContent
});
```

Zugriff auf Umgebungsvariablen

In Ihrem Handler-Code können Sie mithilfe von `process.env` auf beliebige Umgebungsvariablen verweisen. In diesem Beispiel verweisen wir mit den folgenden Codezeilen auf die definierte `RECEIPT_BUCKET`-Umgebungsvariable:

```
// Access environment variables
const bucketName = process.env.RECEIPT_BUCKET;
if (!bucketName) {
    throw new Error('RECEIPT_BUCKET environment variable is not set');
}
```

Verwenden des globalen Zustands

Lambda führt Ihren statischen Code während der Initialisierungsphase aus, bevor Ihre Funktion zum ersten Mal aufgerufen wird. Ressourcen, die während der Initialisierung erstellt werden, bleiben zwischen Aufrufen im Speicher, sodass Sie sie nicht bei jedem Aufruf Ihrer Funktion neu erstellen müssen.

Im Beispielcode befindet sich der Initialisierungscode des S3-Clients außerhalb des Handlers. Die Laufzeit initialisiert den Client, bevor die Funktion ihr erstes Ereignis ausführt, und der Client bleibt für die Wiederverwendung bei allen Aufrufen verfügbar.

Bewährte Codepraktiken für TypeScript Lambda-Funktionen

Befolgen Sie diese Richtlinien, wenn Sie Lambda-Funktionen erstellen:

- Trennen Sie den Lambda-Handler von Ihrer Core-Logik. Auf diese Weise können Sie eine Funktion zur besseren Prüfbarkeit von Einheiten schaffen.
- Kontrollieren Sie die Abhängigkeiten im Bereitstellungspaket Ihrer Funktion. Die AWS Lambda Ausführungsumgebung enthält eine Reihe von Bibliotheken. Für die Laufzeiten Node.js und Python gehören dazu die AWS SDKs. Um die neuesten Funktionen und Sicherheitsupdates zu aktivieren, wird Lambda diese Bibliotheken regelmäßig aktualisieren. Diese Updates können das Verhalten Ihrer Lambda-Funktion geringfügig verändern. Um die Abhängigkeiten, die Ihre Funktion verwendet, vollständig zu kontrollieren, empfehlen wir, alle Abhängigkeiten mit Ihrem Bereitstellungspaket zu bündeln.

- Minimieren Sie die Komplexität Ihrer Abhängigkeiten. Ziehen Sie einfachere Frameworks vor, die sich schnell beim Start der Ausführungsumgebung laden lassen.
- Minimieren Sie die Größe Ihres Bereitstellungspakets auf die für die Laufzeit erforderliche Größe. Dadurch verkürzt sich die Zeit, die für das Herunterladen und Entpacken Ihres Bereitstellungspakets vor dem Aufruf benötigt wird.

Nutzen Sie die Wiederverwendung der Ausführungsumgebung zur Verbesserung Ihrer Funktion. Initialisieren Sie SDK-Clients und Datenbankverbindungen außerhalb des Funktions-Handlers und speichern Sie statische Komponenten lokal im `/tmp`-Verzeichnis. Nachfolgende Aufrufe, die von derselben Instance Ihrer Funktion verarbeitet werden, können diese Ressourcen wiederverwenden. Dies spart Kosten durch Reduzierung der Funktionslaufzeit.

Um potenzielle Datenlecks über Aufrufe hinweg zu vermeiden, verwenden Sie die Ausführungsumgebung nicht, um Benutzerdaten, Ereignisse oder andere Informationen mit Sicherheitsauswirkungen zu speichern. Wenn Ihre Funktion auf einem veränderbaren Zustand beruht, der nicht im Speicher innerhalb des Handlers gespeichert werden kann, sollten Sie für jeden Benutzer eine separate Funktion oder separate Versionen einer Funktion erstellen.

Verwenden Sie eine Keep-Alive-Direktive, um dauerhafte Verbindungen zu pflegen. Lambda bereinigt Leerlaufverbindungen im Laufe der Zeit. Der Versuch, eine Leerlaufverbindung beim Aufruf einer Funktion wiederzuverwenden, führt zu einem Verbindungsfehler. Um Ihre persistente Verbindung aufrechtzuerhalten, verwenden Sie die Keep-Alive-Direktive, die Ihrer Laufzeit zugeordnet ist. Ein Beispiel finden Sie unter Wiederverwenden von Verbindungen mit Keep-Alive in Node.js.

Verwenden Sie Umgebungsvariablen um Betriebsparameter an Ihre Funktion zu übergeben. Wenn Sie z. B. Daten in einen Amazon-S3-Bucket schreiben, anstatt den Bucket-Namen, in den Sie schreiben, hartzucodieren, konfigurieren Sie den Bucket-Namen als Umgebungsvariable.

Vermeiden Sie rekursive Aufrufe in Ihrer Lambda-Funktion, bei denen die Funktion sich selbst aufruft oder einen Prozess initiiert, der die Funktion erneut aufrufen kann. Dies kann zu unvorhergesehenen Mengen an Funktionsaufrufen führen und höhere Kosten zur Folge haben. Wenn Sie eine unbeabsichtigte Menge von Aufrufen feststellen, legen Sie die reservierte gleichzeitige Ausführung der Funktion auf `0` fest, um sofort alle Aufrufe der Funktion zu drosseln, während Sie den Code aktualisieren.

Verwenden Sie APIs in Ihrem Lambda-Funktionscode nicht undokumentiert, nicht öffentlich. Für AWS Lambda verwaltete Laufzeiten führt Lambda regelmäßig Sicherheits- und Funktionsupdates für interne Lambda-Laufzeiten durch. APIs Diese internen API-Updates können abwärtsinkompatibel

sein, was zu unbeabsichtigten Folgen wie Aufruffehlern führen kann, wenn Ihre Funktion von diesen nicht öffentlichen Daten abhängig ist. APIs Eine Liste der öffentlich verfügbaren Programme finden Sie in der API-Referenz. APIs

Schreiben Sie idempotenten Code. Das Schreiben idempotenter Code für Ihre Funktionen stellt sicher, dass doppelte Ereignisse auf die gleiche Weise behandelt werden. Ihr Code sollte Ereignisse ordnungsgemäß validieren und doppelte Ereignisse ordnungsgemäß behandeln. Weitere Informationen finden Sie unter Wie mache ich meine Lambda-Funktion idempotent?.

Stellen Sie transpilierten TypeScript Code in Lambda mit ZIP-Dateiarchiven bereit

Bevor Sie TypeScript Code bereitstellen können AWS Lambda, müssen Sie ihn transpilieren in. JavaScript Auf dieser Seite werden drei Möglichkeiten zum Erstellen und Bereitstellen von TypeScript Code für Lambda mit ZIP-Dateiarchiven beschrieben:

- Verwenden von AWS Serverless Application Model ()AWS SAM
- Mit dem AWS Cloud Development Kit (AWS CDK)
- Verwenden von AWS Command Line Interface (AWS CLI) und esbuild

AWS SAM und AWS CDK vereinfachen das Erstellen und Bereitstellen von TypeScript Funktionen. Die AWS SAM Vorlagenspezifikation bietet eine einfache und übersichtliche Syntax zur Beschreibung der Lambda-Funktionen APIs, Berechtigungen, Konfigurationen und Ereignisse, aus denen Ihre serverlose Anwendung besteht. Über das AWS CDK Sie können zuverlässige, skalierbare und kostengünstige Anwendungen in der Cloud mit der beträchtlichen Ausdruckskraft einer Programmiersprache erstellen. Das AWS CDK richtet sich an mäßig bis sehr erfahrene Benutzer. AWS AWS CDK Sowohl der als auch der AWS SAM verwenden Esbuild, um Code zu transpilieren TypeScript . JavaScript

Verwenden AWS SAM , um TypeScript Code für Lambda bereitzustellen

Gehen Sie wie folgt vor, um eine Hello TypeScript World-Beispielanwendung herunterzuladen, zu erstellen und bereitzustellen AWS SAM. Diese Anwendung implementiert ein grundlegendes API-Backend. Es besteht aus einem Amazon-API-Gateway-Endpunkt und einer Lambda-Funktion. Wenn Sie eine GET-Anforderung an den API-Gateway-Endpunkt senden, wird die Lambda-Funktion aufgerufen. Die Funktion gibt eine `hello world`-Nachricht zurück.

> **Note**
>
> AWS SAM verwendet esbuild, um Lambda-Funktionen von Node.js aus TypeScript Code zu erstellen. Die Esbuild-Unterstützung befindet sich derzeit in der öffentlichen Vorschauversion. Während der öffentlichen Vorschau kann der esbuild-Support Änderungen unterliegen, die nicht mehr rückwärtskompatibel sind.

Voraussetzungen

Zur Durchführung der Schritte in diesem Abschnitt benötigen Sie Folgendes:

- AWS CLI Version 2
- AWS SAM CLI Version 1.75 oder höher
- Node.js

Stellen Sie eine Beispielanwendung AWS SAM bereit

1. Initialisieren Sie die Anwendung mithilfe der Hello TypeScript World-Vorlage.

```
sam init --app-template hello-world-typescript --name sam-app --package-type Zip --runtime nodejs24.x
```

2. (Optional) Die Beispielanwendung enthält Konfigurationen für häufig verwendete Tools, z. B. ESLintfür Code-Linting und Jest für Komponententests. So führen Sie Lint- und Testbefehle aus:

```
cd sam-app/hello-world
npm install
npm run lint
npm run test
```

3. Entwickeln Sie die App.

```
cd sam-app
sam build
```

4. Stellen Sie die Anwendung bereit.

```
sam deploy --guided
```

5. Folgen Sie den Anweisungen auf dem Bildschirm. Um die im interaktiven Erlebnis bereitgestellten Standardoptionen zu akzeptieren, antworten Sie mit `Enter`.
6. Die Ausgabe zeigt den Endpunkt für die REST-API. Öffnen Sie den Endpunkt in einem Browser, um die Funktion zu testen. Folgende Antwort sollte angezeigt werden:

```
{"message":"hello world"}
```

7. Dies ist ein öffentlicher API-Endpunkt, der über das Internet zugänglich ist. Es wird empfohlen, dass Sie den Endpunkt nach dem Testen löschen.

```
sam delete
```

Verwenden von AWS CDK , um TypeScript Code für Lambda bereitzustellen

Gehen Sie wie folgt vor, um eine TypeScript Beispielanwendung mit dem zu erstellen und bereitzustellen AWS CDK. Diese Anwendung implementiert ein grundlegendes API-Backend. Es besteht aus einem API-Gateway-Endpunkt und einer Lambda-Funktion. Wenn Sie eine GET-Anforderung an den API-Gateway-Endpunkt senden, wird die Lambda-Funktion aufgerufen. Die Funktion gibt eine `hello world`-Nachricht zurück.

Voraussetzungen

Zur Durchführung der Schritte in diesem Abschnitt benötigen Sie Folgendes:

- AWS CLI Version 2
- AWS CDK Ausführung 2
- Node.js
- Entweder Docker oder esbuild

Stellen Sie eine AWS CDK Beispielanwendung bereit

1. Erstellen Sie ein Projektverzeichnis für Ihre neue Anwendung.

```
mkdir hello-world
cd hello-world
```

2. Initialisieren Sie die App.

```
cdk init app --language typescript
```

3. Fügen Sie das @types/aws-lambda-Paket als Entwicklungsabhängigkeit hinzu. Dieses Paket enthält die Typdefinitionen für Lambda.

```
npm install -D @types/aws-lambda
```

4. Öffnen Sie das lib-Verzeichnis. Sie sollten eine Datei mit dem Namen „hello-world-stack.ts“ sehen. Erstellen Sie neue zwei neue Dateien in diesem Verzeichnis: hello-world.function.ts und hello-world.ts.
5. Öffnen Sie hello-world.function.ts und fügen Sie den folgenden Code in die Datei ein. Dies ist der Code für die Lambda-Funktion.

> **Note**
>
> Die `import`-Anweisung importiert die Typdefinitionen aus @types/aws-lambda. Das `aws-lambda`-NPM-Paket wird nicht importiert, da es sich um ein unabhängiges Tool eines Drittanbieters handelt. Weitere Informationen finden Sie unter aws-lambda im Repository. 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. Öffnen Sie hello-world.ts und fügen Sie den folgenden Code in die Datei ein. Dies enthält das NodejsFunction Konstrukt, das die Lambda-Funktion erstellt, und das LambdaRestApi Konstrukt, das die REST-API erstellt.

```
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,
    });
  }
}
```

Das `NodejsFunction`-Konstrukt geht standardmäßig von folgendem aus:

- Ihr Funktions-Handler heißt `handler`.
- Die .ts-Datei, die den Funktionscode enthält (hello-world.function.ts), befindet sich im selben Verzeichnis wie die .ts-Datei, die das Konstrukt (hello-world.ts) enthält. Das Konstrukt verwendet die ID des Konstrukts („hello-world“) und den Namen der Lambda-Handler-Datei („function“), um den Funktionscode zu finden. Wenn sich Ihr Funktionscode beispielsweise in einer Datei namens hello-world.my-function.ts befindet, muss die Datei hello-world.ts wie folgt auf den Funktionscode verweisen:

  ```
  const helloFunction = new NodejsFunction(this, 'my-function');
  ```

Sie können dieses Verhalten ändern und andere esbuild-Parameter konfigurieren. Weitere Informationen finden Sie unter Konfiguration von esbuild in der AWS CDK API-Referenz.

7. Öffnen Sie hello-world-stack.ts. Dies ist der Code, der Ihren AWS CDK -Stack definiert. Ersetzen Sie den Code mit Folgendem:

   ```
   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. Stellen Sie Ihre Anwendung aus dem Verzeichnis `hello-world` bereit, das Ihre `cdk.json` Datei enthält.

```
cdk deploy
```

9. Der AWS CDK erstellt und verpackt die Lambda-Funktion mithilfe von esbuild und stellt die Funktion dann für die Lambda-Laufzeit bereit. Die Ausgabe zeigt den Endpunkt für die REST-API. Öffnen Sie den Endpunkt in einem Browser, um die Funktion zu testen. Folgende Antwort sollte angezeigt werden:

   ```
   {"message":"hello world"}
   ```

   Dies ist ein öffentlicher API-Endpunkt, der über das Internet zugänglich ist. Es wird empfohlen, dass Sie den Endpunkt nach dem Testen löschen.

Verwenden von AWS CLI und esbuild zum Bereitstellen von TypeScript Code für Lambda

Das folgende Beispiel zeigt, wie TypeScript Code mithilfe von esbuild transpiliert und für Lambda bereitgestellt wird AWS CLI. esbuild erzeugt eine JavaScript Datei mit allen Abhängigkeiten. Dies ist die einzige Datei, die Sie dem ZIP-Archiv hinzufügen müssen.

Voraussetzungen

Zur Durchführung der Schritte in diesem Abschnitt benötigen Sie Folgendes:

- AWS CLI Version 2
- Node.js
- Eine Ausführungsrolle für die Lambda-Funktion
- Für Windows-Benutzer ein ZIP-Datei-Hilfsprogramm wie 7zip.

Bereitstellen einer Beispielfunktion

1. Erstellen Sie auf Ihrem lokalen Computer ein Projektverzeichnis für Ihre neue Funktion.
2. Erstellen Sie ein neues Node.js Projekt mit npm oder einem Paketmanager Ihrer Wahl.

   ```
   npm init
   ```

3. Fügen Sie die Pakete @types /aws-lambda und esbuild als Entwicklungsabhängigkeiten hinzu. Das `@types/aws-lambda`-Paket enthält die Typdefinitionen für Lambda.

```
npm install -D @types/aws-lambda esbuild
```

4. Erstellen Sie eine neue Datei mit dem Namen index.ts. Fügen Sie den folgenden Code zur Datei hinzu. Dies ist der Code für die Lambda-Funktion. Die Funktion gibt eine `hello world`-Nachricht zurück. Die Funktion erstellt keine API-Gateway-Ressourcen.

> **Note**
>
> Die `import`-Anweisung importiert die Typdefinitionen aus @types/aws-lambda. Das `aws-lambda`-NPM-Paket wird nicht importiert, da es sich um ein unabhängiges Tool eines Drittanbieters handelt. Weitere Informationen finden Sie unter aws-lambda im Repository. 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. Hinzufügen eines Entwicklungs-Skripts zur package.json-Datei. Dies konfiguriert esbuild so, dass das ZIP-Bereitstellungspaket automatisch erstellt wird. Weitere Informationen finden Sie unter Entwicklungs-Skripte in der esbuild-Dokumentation.

   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

In diesem Beispiel verwendet der `"postbuild"` Befehl das 7zip-Hilfsprogramm, um Ihre ZIP-Datei zu erstellen. Verwenden Sie Ihr bevorzugtes Windows-ZIP-Hilfsprogramm und ändern Sie den Befehl nach Bedarf.

```
"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. Erstellen Sie das Paket.

```
npm run build
```

7. Erstellen Sie eine Lambda-Funktion mit dem ZIP-Bereitstellungspaket. Ersetzen Sie den markierten Text durch den Amazon-Ressourcennamen (ARN) Ihrer Ausführungsrolle.

```
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. Führen Sie ein Testereignis aus, um zu bestätigen, dass die Funktion die folgende Antwort zurückgibt. Wenn Sie diese Funktion mit API Gateway aufrufen möchten, Erstellen und Konfigurieren Sie eine REST-API.

```
{
  "statusCode": 200,
  "body": "{\"message\":\"hello world\"}"
}
```

Stellen Sie transpilierten TypeScript-Code in Lambda mit Container-Images bereit

Sie können Ihren TypeScript-Code als Node.js-Container-Image für eine AWS Lambda-Funktion bereitstellen. AWS stellt Basisimages für Node.js bereit, um Sie beim Erstellen des Container-Images zu unterstützen. Diese Basis-Images sind mit einer Sprachlaufzeit und anderen Komponenten vorgeladen, die zum Ausführen des Images auf Lambda erforderlich sind. AWS stellt eine Dockerdatei für jedes der Basis-Images bereit, um beim Erstellen Ihres Container-Images zu helfen.

Wenn Sie ein Community- oder Privatunternehmens-Basis-Image verwenden, müssen Sie den Node.js-Laufzeitschnittstellen-Client (RIC) zum Basis-Image hinzufügen, um es mit Lambda kompatibel zu machen.

Lambda bietet einen Emulator für die Laufzeitschnittstelle zum lokalen Testen. Die AWS-Basis-Images für Node.js enthalten den Emulator für die Laufzeitschnittstelle. Wenn Sie ein alternatives Basis-Image verwenden (beispielsweise ein Alpine Linux-Image oder ein Debian-Image), können Sie den Emulator in Ihr Image integrieren oder ihn auf Ihrem lokalen Computer installieren.

Erstellen und Paketieren von TypeScript-Funktionscode mithilfe eines Node.js-Basis-Image

Voraussetzungen

Zur Durchführung der Schritte in diesem Abschnitt benötigen Sie Folgendes:

- AWS CLI Version 2
- Docker (Mindestversion 25.0.0)
- Das Docker-buildx-Plugin.
- Node.js 22.x

Erstellen eines Images aus einem Base Image

Erstellen Sie ein Image aus einem AWS-Basis-Image für Lambda

1. Erstellen Sie auf Ihrem lokalen Computer ein Projektverzeichnis für Ihre neue Funktion.
2. Erstellen Sie ein neues Node.js-Projekt mit `npm` oder einem Paketmanager Ihrer Wahl.

```
npm init
```

3. Fügen Sie die Pakete @types/aws-lambda und esbuild als Entwicklungsabhängigkeiten hinzu. Das `@types/aws-lambda`-Paket enthält die Typdefinitionen für Lambda.

```
npm install -D @types/aws-lambda esbuild
```

4. Hinzufügen eines Entwicklungs-Skripts zur `package.json`-Datei.

```
  "scripts": {
  "build": "esbuild index.ts --bundle --minify --sourcemap --platform=node --
target=es2020 --outfile=dist/index.js"
}
```

5. Erstellen Sie eine neue Datei mit dem Namen `index.ts`. Fügen Sie den folgenden Beispiel-Code zur neuen Datei hinzu. Dies ist der Code für die Lambda-Funktion. Die Funktion gibt eine `hello world`-Nachricht zurück.

> **Note**
>
> Die `import`-Anweisung importiert die Typdefinitionen aus @types/aws-lambda. Das `aws-lambda`-NPM-Paket wird nicht importiert, da es sich um ein unabhängiges Tool eines Drittanbieters handelt. Weitere Informationen finden Sie unter aws-lambda im DefinitelyTyped-GitHub-Repository.

```
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. Erstellen Sie eine neue Docker-Datei mit der folgenden Konfiguration:

   - Setzen Sie die FROM-Eigenschaft auf den URI des Basis-Images.
   - Legen Sie das CMD-Argument zur Angabe des Lambda-Funktionshandlers fest.

   Die folgende Beispiel-Docker-Datei verwendet eine mehrstufige Entwicklung. Der erste Schritt überträgt den TypeScript-Code in JavaScript. Der zweite Schritt erzeugt ein Container-Image, das nur JavaScript-Dateien und Produktionsabhängigkeiten enthält.

   Beachten Sie, dass das Dockerfile-Beispiel keine USER-Anweisung enthält. Wenn Sie ein Container-Image für Lambda bereitstellen, definiert Lambda automatisch einen Standard-Linux-Benutzer mit Berechtigungen mit geringsten Rechten. Dies unterscheidet sich vom Standardverhalten von Docker, bei dem standardmäßig der `root`-Benutzer verwendet wird, wenn keine USER-Anweisung bereitgestellt wird.

   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. Erstellen Sie Ihr Docker-Image mit dem docker build-Befehl. Das folgende Beispiel benennt das Bild in `docker-image` und gibt ihm den `test` Tag. Um Ihr Image mit Lambda kompatibel zu machen, müssen Sie die Option `--provenance=false` verwenden.

   ```
   docker buildx build --platform linux/amd64 --provenance=false -t docker-image:test
    .
   ```

   > **Note**
   >
   > Der Befehl gibt die `--platform linux/amd64`-Option an, um sicherzustellen, dass Ihr Container mit der Lambda-Ausführungsumgebung kompatibel ist, unabhängig von

der Architektur des Entwicklungsrechners. Wenn Sie beabsichtigen, eine Lambda-Funktion mithilfe der ARM64-Befehlssatzarchitektur zu erstellen, müssen Sie den Befehl unbedingt so ändern, dass stattdessen die `--platform linux/arm64`-Option verwendet wird.

(Optional) Testen Sie das Image lokal

1. Starten Sie Ihr Docker-Image mit dem docker run-Befehl. In diesem Beispiel ist `docker-image` der Image-Name und `test` der Tag.

   ```
   docker run --platform linux/amd64 -p 9000:8080 docker-image:test
   ```

   Dieser Befehl führt das Image als Container aus und erstellt einen lokalen Endpunkt bei `localhost:9000/2015-03-31/functions/function/invocations`.

   > **Note**
   >
   > Wenn Sie das Docker-Image für die ARM64-Befehlssatz-Architektur erstellt haben, müssen Sie die Option `--platform linux/arm64` statt `--platform linux/amd64` verwenden.

2. Veröffentlichen Sie in einem neuen Terminalfenster ein Ereignis an den lokalen Endpunkt.

   Linux/macOS

   Führen Sie unter Linux oder macOS den folgenden `curl`-Befehl aus:

   ```
   curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{}'
   ```

   Dieser Befehl ruft die Funktion mit einem leeren Ereignis auf und gibt eine Antwort zurück. Wenn Sie Ihren eigenen Funktionscode anstelle des Beispielfunktionscodes verwenden, wird empfohlen, die Funktion mit einer JSON-Nutzlast aufzurufen. Beispiel:

   ```
   curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d
    '{"payload":"hello world!"}'
   ```

PowerShell

Führen Sie in PowerShell den folgenden `Invoke-WebRequest`-Befehl aus:

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{}' -ContentType "application/json"
```

Dieser Befehl ruft die Funktion mit einem leeren Ereignis auf und gibt eine Antwort zurück. Wenn Sie Ihren eigenen Funktionscode anstelle des Beispielfunktionscodes verwenden, wird empfohlen, die Funktion mit einer JSON-Nutzlast aufzurufen. Beispiel:

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{"payload":"hello world!"}' -ContentType
 "application/json"
```

3. Die Container-ID erhalten.

```
docker ps
```

4. Verwenden Sie den Befehl docker kill, um den Container zu anzuhalten. Ersetzen Sie in diesem Befehl `3766c4ab331c` durch die Container-ID aus dem vorherigen Schritt.

```
docker kill 3766c4ab331c
```

Das Image bereitstellen

Um das Image in Amazon ECR hochzuladen und die Lambda-Funktion zu erstellen

1. Führen Sie den Befehl get-login-password aus, um die Docker-CLI bei Ihrem Amazon-ECR-Registry zu authentifizieren.

   - Stellen Sie den `--region` Wert auf die AWS-Region ein, in der Sie das Amazon-ECR-Repository erstellen möchten.
   - Ersetzen Sie `111122223333` mit Ihrer AWS-Konto-ID.

```
aws ecr get-login-password --region us-east-1 | docker login --username AWS --
password-stdin 111122223333.dkr.ecr.us-east-1.amazonaws.com
```

2. Erstellen Sie ein Repository in Amazon ECR mithilfe des Befehls create-repository.

```
aws ecr create-repository --repository-name hello-world --region us-east-1 --image-scanning-configuration scanOnPush=true --image-tag-mutability MUTABLE
```

> **Note**
>
> Das Amazon ECR-Repository muss sich im selben AWS-Region wie die Lambda-Funktion befinden.

Wenn erfolgreich, sehen Sie eine Antwort wie diese:

```
{
    "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. Kopieren Sie das `repositoryUri` aus der Ausgabe im vorherigen Schritt.
4. Führen Sie den Befehl docker tag aus, um Ihr lokales Image als neueste Version in Ihrem Amazon-ECR-Repository zu markieren. In diesem Befehl gilt Folgendes:

   - `docker-image:test` ist der Name und das Tag Ihres Docker-Images. Dies sind der Imagename und das Tag, die Sie im `docker build`-Befehl angegeben haben.
   - Ersetzen Sie <ECRrepositoryUri> durch den `repositoryUri`, den Sie kopiert haben. Stellen Sie sicher, dass Sie `:latest` am Ende der URI angeben.

```
docker tag docker-image:test <ECRrepositoryUri>:latest
```

Beispiel:

```
docker tag docker-image:test 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

5. Führen Sie den Befehl docker push aus, um Ihr lokales Image im Amazon-ECR-Repository bereitzustellen. Stellen Sie sicher, dass Sie `:latest` am Ende der Repository-URI angeben.

```
docker push 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

6. Erstellen Sie eine Ausführungsrolle für die Funktion, wenn Sie noch keine haben. Sie benötigen den Amazon-Ressourcennamen (ARN) der Rolle im nächsten Schritt.
7. So erstellen Sie die Lambda-Funktion: Geben Sie für `ImageUri` die Repository-URI von zuvor an. Stellen Sie sicher, dass Sie `:latest` am Ende der URI angeben.

```
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**
>
> Sie können eine Funktion mit einem Bild in einem anderen AWS-Konto erstellen, sofern sich das Bild in derselben Region wie die Lambda-Funktion befindet. Weitere Informationen finden Sie unter Kontoübergreifende Berechtigungen von Amazon ECR.

8. Die Funktion aufrufen.

```
aws lambda invoke --function-name hello-world response.json
```

Das Ergebnis sollte ungefähr wie folgt aussehen:

```
{
  "ExecutedVersion": "$LATEST",
```

```
  "StatusCode": 200
}
```

9. Um die Ausgabe der Funktion zu sehen, überprüfen Sie die `response.json`-Datei.

Um den Funktionscode zu aktualisieren, müssen Sie das Image erneut erstellen, das neue Image in das Amazon-ECR-Repository hochladen und dann den Befehl update-function-code verwenden, um das Image für die Lambda-Funktion bereitzustellen.

Lambda löst das Image-Tag in einen bestimmten Image-Digest auf. Das heißt, wenn Sie das Image-Tag, das zur Bereitstellung der Funktion verwendet wurde, auf ein neues Image in Amazon ECR verweisen, aktualisiert Lambda die Funktion nicht automatisch, um das neue Image zu verwenden.

Um das neue Image für dieselbe Lambda-Funktion bereitzustellen, müssen Sie den Befehl update-function-code verwenden, auch wenn das Image-Tag in Amazon ECR gleich bleibt. Im folgenden Beispiel erstellt die `--publish`-Option eine neue Version der Funktion unter Verwendung des aktualisierten Container-Images.

```
aws lambda update-function-code \
  --function-name hello-world \
  --image-uri 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --publish
```

Verwenden des Lambda-Kontextobjekts zum Abrufen von TypeScript Funktionsinformationen

Wenn Lambda Ihre Funktion ausführt, wird ein Context-Objekt an den Handler. übergeben. Dieses Objekt stellt Methoden und Eigenschaften mit Informationen zum Aufruf, zur Funktion und zur Ausführungsumgebung bereit.

Um die Typüberprüfung für das Kontextobjekt zu aktivieren, müssen Sie das Paket @types/aws-lambda als Entwicklungsabhängigkeit hinzufügen und den Typ `Context` importieren. Weitere Informationen finden Sie unter Typdefinitionen für Lambda.

Context-Methoden

- `getRemainingTimeInMillis()` – Gibt die Anzahl der verbleibenden Millisekunden zurück, bevor die Ausführung das Zeitlimit überschreitet.

Context-Eigenschaften

- `functionName` – Der Name der Lambda-Funktion.
- `functionVersion` – Die Version der Funktion.
- `invokedFunctionArn` – Der Amazon-Ressourcenname (ARN), der zum Aufrufen der Funktion verwendet wird. Gibt an, ob der Aufrufer eine Versionsnummer oder einen Alias angegeben hat.
- `memoryLimitInMB` – Die Menge an Arbeitsspeicher, die der Funktion zugewiesen ist.
- `awsRequestId` – Der Bezeichner der Aufrufanforderung.
- `logGroupName` – Protokollgruppe für die Funktion.
- `logStreamName` – Der Protokollstream für die Funktions-Instance.
- `identity` – Informationen zur Amazon-Cognito-Identität, die die Anforderung autorisiert hat.
  - `cognitoIdentityId`— Die authentifizierte Amazon-Cognito-Identität.
  - `cognitoIdentityPoolId` – Der Amazon-Cognito-Identitätspool, der den Aufruf autorisiert hat.
- `clientContext` – (mobile Apps) Clientkontext, der Lambda von der Clientanwendung bereitgestellt wird.
  - `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` – Benutzerdefinierte Werte, die durch die mobilen Anwendung festgelegt werden.
- `callbackWaitsForEmptyEventLoop`— Standardmäßig (`true`) wartet Lambda bei Verwendung eines Callback-basierten Funktionshandlers darauf, dass die Ereignisschleife leer ist, nachdem der Callback ausgeführt wurde, bevor der Funktionsaufruf beendet wird. Legt fest, dass `false` die Antwort gesendet und der Aufruf sofort beendet wird, nachdem der Callback ausgeführt wurde, anstatt darauf zu warten, dass die Event-Schleife leer ist. Ausstehende Ereignisse werden beim nächsten Aufruf weiterhin ausgeführt. Beachten Sie, dass Lambda Callback-basierte Funktionshandler nur für Node.js 22 und frühere Laufzeiten unterstützt.

Example Datei index.js

Die folgende Beispielfunktion protokolliert Kontextinformationen und gibt den Speicherort der Protokolle zurück.

> **Note**
>
> Bevor Sie diesen Code in einer Lambda-Funktion verwenden, müssen Sie das Paket @types/aws-lambda als Entwicklungsabhängigkeit hinzufügen. Dieses Paket enthält die Typdefinitionen für Lambda. Weitere Informationen finden Sie unter Typdefinitionen für 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;
};
```

TypeScript Lambda-Funktionen protokollieren und überwachen

AWS Lambda überwacht automatisch Lambda-Funktionen und sendet Protokolleinträge an Amazon CloudWatch. Ihre Lambda-Funktion enthält eine CloudWatch Logs-Log-Gruppe und einen Log-Stream für jede Instanz Ihrer Funktion. Die Lambda-Laufzeitumgebung sendet Details zu den einzelnen Aufrufen an den Protokollstream und leitet Protokolle und andere Ausgaben aus dem Code Ihrer Funktion weiter. Weitere Informationen zu CloudWatch Logs finden Sie unterSenden von Lambda-Funktionsprotokollen an CloudWatch Logs.

Um Protokolle aus Ihrem Funktionscode auszugeben, können Sie Methoden auf dem Konsolenobjekt verwenden. Für eine detailliertere Protokollierung können Sie jede Protokollierungsbibliothek verwenden, die in `stdout` oder `stderr` schreibt.

Sections

- Tools und Bibliotheken für die Protokollierung verwenden
- Verwendung von Powertools für AWS Lambda (TypeScript) und für die strukturierte Protokollierung AWS SAM
- Verwenden Sie Powertools für AWS Lambda (TypeScript) und AWS CDK für die strukturierte Protokollierung
- Anzeigen von Protokollen in der Lambda-Konsole
- Logs in der CloudWatch Konsole anzeigen

Tools und Bibliotheken für die Protokollierung verwenden

Powertools for AWS Lambda (TypeScript) ist ein Entwickler-Toolkit zur Implementierung von Best Practices für Serverless und zur Steigerung der Entwicklergeschwindigkeit. Das Logger-Serviceprogramm bietet einen für Lambda optimierten Logger, der zusätzliche Informationen über den Funktionskontext all Ihrer Funktionen enthält, wobei die Ausgabe als JSON strukturiert ist. Mit diesem Serviceprogramm können Sie Folgendes tun:

- Erfassung von Schlüsselfeldern aus dem Lambda-Kontext, Kaltstart und Strukturen der Protokollierungsausgabe als JSON
- Protokollieren Sie Ereignisse von Lambda-Aufrufen, wenn Sie dazu aufgefordert werden (standardmäßig deaktiviert)
- Alle Protokolle nur für einen bestimmten Prozentsatz der Aufrufe über Protokollstichproben drucken (standardmäßig deaktiviert)

- Fügen Sie dem strukturierten Protokoll zu einem beliebigen Zeitpunkt zusätzliche Schlüssel hinzu
- Verwenden Sie einen benutzerdefinierten Protokollformatierer (Bring Your Own Formatter), um Protokolle in einer Struktur auszugeben, die mit dem Logging RFC Ihres Unternehmens kompatibel ist

Verwendung von Powertools für AWS Lambda (TypeScript) und für die strukturierte Protokollierung AWS SAM

Gehen Sie wie folgt vor, um mithilfe von eine Hello TypeScript World-Beispielanwendung mit integrierten Powertools for AWS Lambda (TypeScript) -Modulen herunterzuladen, zu erstellen und bereitzustellen. AWS SAM Diese Anwendung implementiert ein grundlegendes API-Backend und verwendet Powertools zum Ausgeben von Protokollen, Metriken und Traces. Es besteht aus einem Amazon-API-Gateway-Endpunkt und einer Lambda-Funktion. Wenn Sie eine GET-Anfrage an den API-Gateway-Endpunkt senden, ruft die Lambda-Funktion Logs und Metriken auf, sendet sie im Embedded Metric Format an CloudWatch und sendet Traces an. AWS X-Ray Die Funktion gibt eine `hello world`-Nachricht zurück.

Voraussetzungen

Zur Durchführung der Schritte in diesem Abschnitt benötigen Sie Folgendes:

- Node.js 20 oder höher
- AWS CLI Version 2
- AWS SAM CLI Version 1.75 oder höher. Wenn Sie eine ältere Version der AWS SAM CLI haben, finden Sie weitere Informationen unter Upgrade der AWS SAM CLI.

Stellen Sie eine AWS SAM Beispielanwendung bereit

1. Initialisieren Sie die Anwendung mithilfe der Hello TypeScript World-Vorlage.

```
sam init --app-template hello-world-powertools-typescript --name sam-app --package-type Zip --runtime nodejs24.x
```

2. Entwickeln Sie die App.

```
cd sam-app && sam build
```

3. Stellen Sie die Anwendung bereit.

```
sam deploy --guided
```

4. Folgen Sie den Anweisungen auf dem Bildschirm. Um die im interaktiven Erlebnis bereitgestellten Standardoptionen zu akzeptieren, drücken Sie `Enter`.

> **Note**
>
> Für ist HelloWorldFunction möglicherweise keine Autorisierung definiert. Ist das in Ordnung? , stellen Sie sicher, dass Sie eintreteny.

5. Rufen Sie die URL der bereitgestellten Anwendung ab:

```
aws cloudformation describe-stacks --stack-name sam-app --query
 'Stacks[0].Outputs[?OutputKey==`HelloWorldApi`].OutputValue' --output text
```

6. Rufen Sie den API-Endpunkt auf:

```
curl <URL_FROM_PREVIOUS_STEP>
```

Wenn der Link erfolgreich ausgeführt wurde, sehen Sie die folgende Antwort:

```
{"message":"hello world"}
```

7. Führen Sie sam logs aus, um die Protokolle für die Funktion abzurufen. Weitere Informationen finden Sie unter Arbeiten mit Protokollen im AWS Serverless Application Model - Entwicklerhandbuch.

```
sam logs --stack-name sam-app
```

Das Ergebnis sieht folgendermaßen aus:

```
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. Dies ist ein öffentlicher API-Endpunkt, der über das Internet zugänglich ist. Es wird empfohlen, dass Sie den Endpunkt nach dem Testen löschen.

```
sam delete
```

Verwalten der Protokollaufbewahrung

Wenn Sie eine Funktion löschen, werden Protokollgruppen nicht automatisch gelöscht. Um zu vermeiden, dass Protokolle auf unbestimmte Zeit gespeichert werden, löschen Sie die Protokollgruppe oder konfigurieren Sie einen Aufbewahrungszeitraum, nach dessen Ablauf die Protokolle CloudWatch automatisch gelöscht werden. Um die Aufbewahrung von Protokollen einzurichten, fügen Sie Ihrer AWS SAM Vorlage Folgendes hinzu:

```
Resources:
  HelloWorldFunction:
    Type: AWS::Serverless::Function
    Properties:
    # Omitting other properties

  LogGroup:
    Type: AWS::Logs::LogGroup
    Properties:
      LogGroupName: !Sub "/aws/lambda/${HelloWorldFunction}"
      RetentionInDays: 7
```

Verwenden Sie Powertools für AWS Lambda (TypeScript) und AWS CDK für die strukturierte Protokollierung

Gehen Sie wie folgt vor, um mithilfe von eine Hello TypeScript World-Beispielanwendung mit integrierten Powertools for AWS Lambda (TypeScript) -Modulen herunterzuladen, zu erstellen und bereitzustellen. AWS CDK Diese Anwendung implementiert ein grundlegendes API-Backend und verwendet Powertools zum Ausgeben von Protokollen, Metriken und Traces. Es besteht aus einem Amazon-API-Gateway-Endpunkt und einer Lambda-Funktion. Wenn Sie eine GET-Anfrage an den API-Gateway-Endpunkt senden, ruft die Lambda-Funktion Logs und Metriken auf, sendet sie im Embedded Metric Format an CloudWatch und sendet Traces an. AWS X-Ray Die Funktion gibt eine `hello world`-Nachricht zurück.

Voraussetzungen

Zur Durchführung der Schritte in diesem Abschnitt benötigen Sie Folgendes:

- Node.js 20 oder höher
- AWS CLI Version 2
- AWS CDK Ausführung 2
- AWS SAM CLI Version 1.75 oder höher. Wenn Sie eine ältere Version der AWS SAM CLI haben, finden Sie weitere Informationen unter Upgrade der AWS SAM CLI.

Stellen Sie eine AWS CDK Beispielanwendung bereit

1. Erstellen Sie ein Projektverzeichnis für Ihre neue Anwendung.

```
mkdir hello-world
cd hello-world
```

2. Initialisieren Sie die App.

```
cdk init app --language typescript
```

3. Fügen Sie das @types/aws-lambda-Paket als Entwicklungsabhängigkeit hinzu.

```
npm install -D @types/aws-lambda
```

4. Installieren Sie das Logger-Dienstprogramm von Powertools.

```
npm install @aws-lambda-powertools/logger
```

5. Öffnen Sie das lib-Verzeichnis. Sie sollten eine Datei mit dem Namen „hello-world-stack.ts“ sehen. Erstellen Sie neue zwei neue Dateien in diesem Verzeichnis: hello-world.function.ts und hello-world.ts.
6. Öffnen Sie hello-world.function.ts und fügen Sie den folgenden Code in die Datei ein. Dies ist der Code für die Lambda-Funktion.

```
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. Öffnen Sie hello-world.ts und fügen Sie den folgenden Code in die Datei ein. Dies enthält das NodejsFunction Konstrukt, das die Lambda-Funktion erstellt, Umgebungsvariablen für Powertools konfiguriert und die Protokollspeicherung auf eine Woche festlegt. Es beinhaltet auch das LambdaRestApi Konstrukt, das die REST-API erstellt.

```
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. Öffnen Sie hello-world-stack.ts. Dies ist der Code, der Ihren AWS CDK -Stack definiert. Ersetzen Sie den Code mit Folgendem:

```
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. Kehren Sie zum Projektverzeichnis zurück.

```
cd hello-world
```

10. Stellen Sie die Anwendung bereit.

```
cdk deploy
```

11. Rufen Sie die URL der bereitgestellten Anwendung ab:

```
aws cloudformation describe-stacks --stack-name HelloWorldStack --query
 'Stacks[0].Outputs[?ExportName==`apiUrl`].OutputValue' --output text
```

12. Rufen Sie den API-Endpunkt auf:

```
curl <URL_FROM_PREVIOUS_STEP>
```

Wenn der Link erfolgreich ausgeführt wurde, sehen Sie die folgende Antwort:

```
{"message":"hello world"}
```

13. Führen Sie sam logs aus, um die Protokolle für die Funktion abzurufen. Weitere Informationen finden Sie unter Arbeiten mit Protokollen im AWS Serverless Application Model - Entwicklerhandbuch.

```
sam logs --stack-name HelloWorldStack
```

Das Ergebnis sieht folgendermaßen aus:

```
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. Dies ist ein öffentlicher API-Endpunkt, der über das Internet zugänglich ist. Es wird empfohlen, dass Sie den Endpunkt nach dem Testen löschen.

```
cdk destroy
```

Anzeigen von Protokollen in der Lambda-Konsole

Sie können die Lambda-Konsole verwenden, um die Protokollausgabe nach dem Aufrufen einer Lambda-Funktion anzuzeigen.

Wenn Ihr Code über den eingebetteten Code-Editor getestet werden kann, finden Sie Protokolle in den Ausführungsergebnissen. Wenn Sie das Feature Konsolentest verwenden, um eine Funktion aufzurufen, finden Sie die Protokollausgabe im Abschnitt Details.

Logs in der CloudWatch Konsole anzeigen

Sie können die CloudWatch Amazon-Konsole verwenden, um Protokolle für alle Lambda-Funktionsaufrufe anzuzeigen.

Um Protokolle auf der Konsole anzuzeigen CloudWatch

1. Öffnen Sie die Seite Protokollgruppen auf der CloudWatch Konsole.
2. Wählen Sie die Protokollgruppe für Ihre Funktion (***your-function-name***/aws/lambda/).
3. Wählen Sie eine Protokollstream aus.

Jeder Protokoll-Stream entspricht einer Instance Ihrer Funktion. Ein Protokollstream wird angezeigt, wenn Sie Ihre Lambda-Funktion aktualisieren und wenn zusätzliche Instanzen für gleichzeitige Aufrufe erstellt werden. Um Protokolle für einen bestimmten Aufruf zu finden, empfehlen wir, Ihre Funktion mit zu instrumentieren. AWS X-Ray X-Ray erfasst Details zu der Anforderung und dem Protokollstream in der Trace.

TypeScript Code nachverfolgen in AWS Lambda

Lambda lässt sich integrieren AWS X-Ray , um Ihnen zu helfen, Lambda-Anwendungen zu verfolgen, zu debuggen und zu optimieren. Sie können mit X-Ray eine Anforderung verfolgen, während sie Ressourcen in Ihrer Anwendung durchläuft, die Lambda-Funktionen und andere AWS -Services enthalten können.

Um Protokollierungsdaten an X-Ray zu senden, können Sie eine von drei SDK-Bibliotheken verwenden:

- AWS Distro for OpenTelemetry (ADOT) — Eine sichere, produktionsbereite und AWS unterstützte Distribution des () SDK. OpenTelemetry OTel
- AWS X-Ray SDK für Node.js — Ein SDK zum Generieren und Senden von Trace-Daten an X-Ray.
- Powertools for AWS Lambda (TypeScript) — Ein Entwickler-Toolkit zur Implementierung von Best Practices für Serverless und zur Steigerung der Entwicklergeschwindigkeit.

Jedes SDKs bietet Möglichkeiten, Ihre Telemetriedaten an den X-Ray-Dienst zu senden. Sie können dann mit X-Ray die Leistungsmetriken Ihrer Anwendung anzeigen, filtern und erhalten, um Probleme und Möglichkeiten zur Optimierung zu identifizieren.

> **Important**
>
> X-Ray und Powertools für AWS Lambda SDKs sind Teil einer eng integrierten Instrumentierungslösung von AWS. Die ADOT Lambda Layers sind Teil eines branchenweiten Standards für die Verfolgung von Instrumenten, die im Allgemeinen mehr Daten erfassen, aber möglicherweise nicht für alle Anwendungsfälle geeignet sind. Sie können die end-to-end Ablaufverfolgung in X-Ray mit beiden Lösungen implementieren. Weitere Informationen zur Auswahl zwischen beiden finden Sie unter Wählen zwischen der AWS Distribution für Open Telemetry und X-Ray. SDKs

Sections

- Powertools für AWS Lambda (TypeScript) und für die Nachverfolgung verwenden AWS SAM
- Verwenden von Powertools for AWS Lambda (TypeScript) und AWS CDK for tracing
- Interpretieren einer X-Ray-Nachverfolgung

Powertools für AWS Lambda (TypeScript) und für die Nachverfolgung verwenden AWS SAM

Gehen Sie wie folgt vor, um mithilfe von eine Hello TypeScript World-Beispielanwendung mit integrierten Powertools for AWS Lambda (TypeScript) -Modulen herunterzuladen, zu erstellen und bereitzustellen. AWS SAM Diese Anwendung implementiert ein grundlegendes API-Backend und verwendet Powertools zum Ausgeben von Protokollen, Metriken und Traces. Es besteht aus einem Amazon-API-Gateway-Endpunkt und einer Lambda-Funktion. Wenn Sie eine GET-Anfrage an den API-Gateway-Endpunkt senden, ruft die Lambda-Funktion Logs und Metriken auf, sendet sie im Embedded Metric Format an CloudWatch und sendet Traces an. AWS X-Ray Die Funktion gibt eine `hello world`-Nachricht zurück.

Voraussetzungen

Zur Durchführung der Schritte in diesem Abschnitt benötigen Sie Folgendes:

- Node.js
- AWS CLI Version 2
- AWS SAM CLI Version 1.75 oder höher. Wenn Sie eine ältere Version der AWS SAM CLI haben, finden Sie weitere Informationen unter Upgrade der AWS SAM CLI.

Stellen Sie eine AWS SAM Beispielanwendung bereit

1. Initialisieren Sie die Anwendung mithilfe der Hello TypeScript World-Vorlage.

```
sam init --app-template hello-world-powertools-typescript --name sam-app --package-type Zip --runtime nodejs24.x --no-tracing
```

2. Entwickeln Sie die App.

```
cd sam-app && sam build
```

3. Stellen Sie die Anwendung bereit.

```
sam deploy --guided
```

4. Folgen Sie den Anweisungen auf dem Bildschirm. Um die im interaktiven Erlebnis bereitgestellten Standardoptionen zu akzeptieren, drücken Sie `Enter`.

> **Note**
>
> Für ist HelloWorldFunction möglicherweise keine Autorisierung definiert. Ist das in Ordnung? , stellen Sie sicher, dass Sie eintreteny.

5. Rufen Sie die URL der bereitgestellten Anwendung ab:

```
aws cloudformation describe-stacks --stack-name sam-app --query
 'Stacks[0].Outputs[?OutputKey==`HelloWorldApi`].OutputValue' --output text
```

6. Rufen Sie den API-Endpunkt auf:

```
curl <URL_FROM_PREVIOUS_STEP>
```

Wenn der Link erfolgreich ausgeführt wurde, sehen Sie die folgende Antwort:

```
{"message":"hello world"}
```

7. Führen Sie sam traces aus, um die Traces für die Funktion zu erhalten.

```
sam traces
```

Das Nachverfolgungsergebnis sieht folgendermaßen aus:

```
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. Dies ist ein öffentlicher API-Endpunkt, der über das Internet zugänglich ist. Es wird empfohlen, dass Sie den Endpunkt nach dem Testen löschen.

```
sam delete
```

X-Ray verfolgt nicht alle Anfragen an Ihre Anwendung nach. X-Ray wendet einen Sampling-Algorithmus an, um sicherzustellen, dass die Nachverfolgung effizient ist, und stellt dennoch ein repräsentatives Beispiel aller Anfragen bereit. Die Samplingrate beträgt 1 Anforderung pro Sekunde und 5 Prozent aller weiteren Anforderungen. Sie können die X-Ray-Samplingrate nicht für Ihre Funktionen konfigurieren.

Verwenden von Powertools for AWS Lambda (TypeScript) und AWS CDK for tracing

Gehen Sie wie folgt vor, um mithilfe von eine Hello TypeScript World-Beispielanwendung mit integrierten Powertools for AWS Lambda (TypeScript) -Modulen herunterzuladen, zu erstellen und bereitzustellen. AWS CDK Diese Anwendung implementiert ein grundlegendes API-Backend und verwendet Powertools zum Ausgeben von Protokollen, Metriken und Traces. Es besteht aus einem Amazon-API-Gateway-Endpunkt und einer Lambda-Funktion. Wenn Sie eine GET-Anfrage an den API-Gateway-Endpunkt senden, ruft die Lambda-Funktion Logs und Metriken auf, sendet sie im Embedded Metric Format an CloudWatch und sendet Traces an. AWS X-Ray Die Funktion gibt eine `hello world`-Nachricht zurück.

Voraussetzungen

Zur Durchführung der Schritte in diesem Abschnitt benötigen Sie Folgendes:

- Node.js
- AWS CLI Version 2
- AWS CDK Ausführung 2
- AWS SAM CLI Version 1.75 oder höher. Wenn Sie eine ältere Version der AWS SAM CLI haben, finden Sie weitere Informationen unter Upgrade der AWS SAM CLI.

Stellen Sie eine AWS Cloud Development Kit (AWS CDK) Beispielanwendung bereit

1. Erstellen Sie ein Projektverzeichnis für Ihre neue Anwendung.

```
mkdir hello-world
cd hello-world
```

2. Initialisieren Sie die App.

```
cdk init app --language typescript
```

3. Fügen Sie das @types/aws-lambda-Paket als Entwicklungsabhängigkeit hinzu.

```
npm install -D @types/aws-lambda
```

4. Installieren Sie das Tracer-Dienstprogramm von Powertools.

```
npm install @aws-lambda-powertools/tracer
```

5. Öffnen Sie das lib-Verzeichnis. Sie sollten eine Datei namens hello-world-stack.ts sehen. Erstellen Sie neue zwei neue Dateien in diesem Verzeichnis: hello-world.function.ts und hello-world.ts.
6. Öffnen Sie hello-world.function.ts und fügen Sie den folgenden Code in die Datei ein. Dies ist der Code für die Lambda-Funktion.

```
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. Öffnen Sie hello-world.ts und fügen Sie den folgenden Code in die Datei ein. Dies enthält das NodejsFunction Konstrukt, das die Lambda-Funktion erstellt, Umgebungsvariablen für Powertools konfiguriert und die Protokollspeicherung auf eine Woche festlegt. Es beinhaltet auch das LambdaRestApi Konstrukt, das die REST-API erstellt.

```
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. Öffnen Sie hello-world-stack.ts. Dies ist der Code, der Ihren AWS CDK -Stack definiert. Ersetzen Sie den Code mit Folgendem:

```
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. Stellen Sie die Anwendung bereit.

```
cd ..
cdk deploy
```

10. Rufen Sie die URL der bereitgestellten Anwendung ab:

```
aws cloudformation describe-stacks --stack-name HelloWorldStack --query
 'Stacks[0].Outputs[?ExportName==`apiUrl`].OutputValue' --output text
```

11. Rufen Sie den API-Endpunkt auf:

```
curl <URL_FROM_PREVIOUS_STEP>
```

Wenn der Link erfolgreich ausgeführt wurde, sehen Sie die folgende Antwort:

```
{"message":"hello world"}
```

12. Führen Sie sam traces aus, um die Traces für die Funktion zu erhalten.

```
sam traces
```

Das Nachverfolgungsergebnis sieht folgendermaßen aus:

```
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]
```

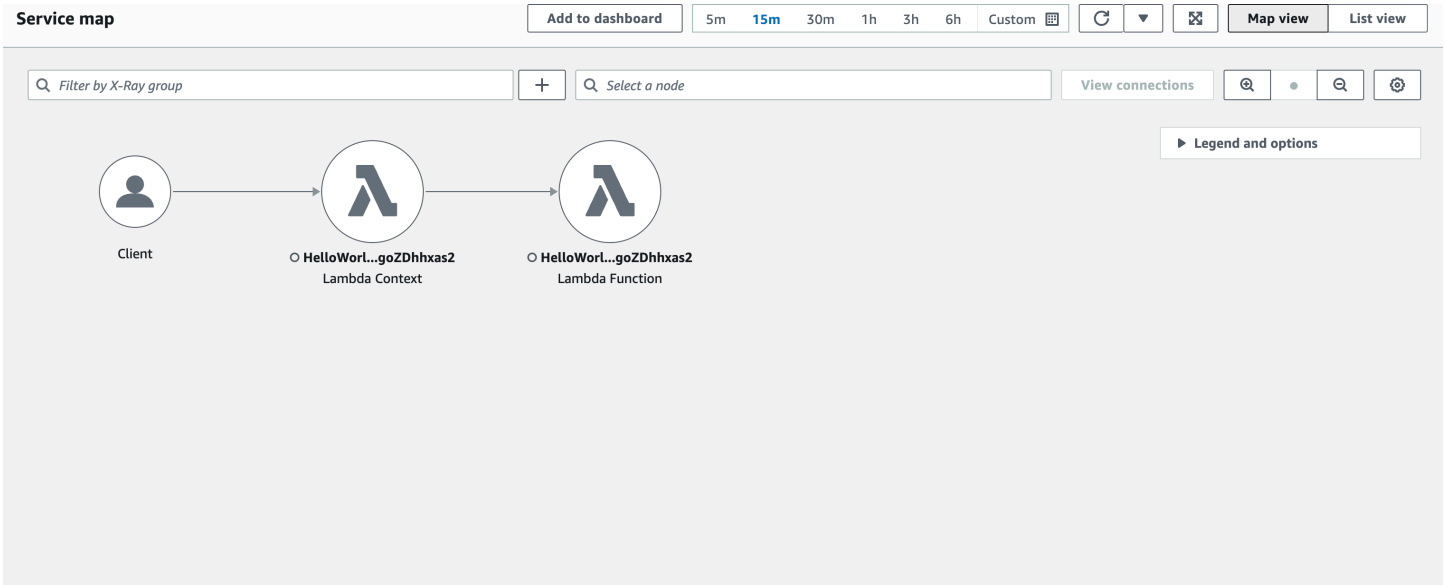
```
  - 0.157s - HelloWorldStack-helloworldfunction111A2BCD-Xyzv11a1bcde
    - 0.169s - Initialization
    - 0.058s - Invocation
      - 0.055s - ## index.handler
        - 0.000s - ### MySubSegment
    - 0.099s - Overhead
```

13. Dies ist ein öffentlicher API-Endpunkt, der über das Internet zugänglich ist. Es wird empfohlen, dass Sie den Endpunkt nach dem Testen löschen.

```
cdk destroy
```

Interpretieren einer X-Ray-Nachverfolgung

Nachdem Sie die aktive Nachverfolgung konfiguriert haben, können Sie bestimmte Anfragen über Ihre Anwendung beobachten. Die X-Ray-Trace-Map liefert Informationen über Ihre Anwendung und alle ihre Komponenten. Das folgende Beispiel zeigt eine Nachverfolgung aus der Beispielanwendung:



Erstellen von Lambda-Funktionen mit Python

Sie können Python-Code in AWS Lambda ausführen. Lambda bietet Laufzeiten für Python, die Ihren Code ausführen, um Ereignisse zu verarbeiten. Ihr Code wird in einer Umgebung ausgeführt, die das SDK for Python (Boto3) enthält, mit Anmeldeinformationen von einer AWS Identity and Access Management (IAM-) Rolle, die Sie verwalten. Weitere Informationen zu den SDK-Versionen, die in den Laufzeiten von Python enthalten sind, finden Sie unter the section called “SDK-Versionen, die zur Laufzeit enthalten sind”.

Lambda unterstützt die folgenden Python-Laufzeiten.

Name	ID	Betriebssystem	Datum der Veraltung	Blockfunktion erstellen	Blockfunktion aktualisieren
Python 3.14	`python3.14`	Amazon Linux 2023	30. Juni 2029	31. Juli 2029	31. August 2029
Python 3.13	`python3.13`	Amazon Linux 2023	30. Juni 2029	31. Juli 2029	31. August 2029
Python 3.12	`python3.12`	Amazon Linux 2023	31. Oktober 2028	30. November 2028	10. Januar 2029
Python 3.11	`python3.11`	Amazon Linux 2	30. Juni 2027	31. Juli 2027	31. August 2027
Python 3.10	`python3.10`	Amazon Linux 2	31. Oktober 2026	30. November 2026	15. Januar 2027

So erstellen Sie eine Python-Funktion

1. Öffnen Sie die Lambda-Konsole.
2. Wählen Sie Funktion erstellen.
3. Konfigurieren Sie die folgenden Einstellungen:

- Funktionsname: Geben Sie einen Namen für die Funktion ein.
- Runtime: Wähle Python 3.14.

4. Wählen Sie Funktion erstellen.

Die Konsole erstellt eine Lambda-Funktion mit einer einzigen Quelldatei mit dem Namen `lambda_function`. Mit dem integrierten Code-Editor können Sie diese Datei bearbeiten und weitere Dateien hinzufügen. Wählen Sie im Abschnitt BEREITSTELLEN die Option Bereitstellen aus, um den Code Ihrer Funktion zu aktualisieren. Um Ihren Code auszuführen, wählen Sie anschließend im Abschnitt TESTEREIGNISSE die Option Testereignis erstellen aus.

Ihre Lambda-Funktion wird mit einer CloudWatch Logs-Protokollgruppe geliefert. Die Funktionslaufzeit sendet Details zu jedem Aufruf an CloudWatch Logs. Es leitet alle Protokolle weiter, die Ihre Funktion während des Aufrufs ausgibt. Wenn Ihre Funktion einen Fehler zurückgibt, formatiert Lambda den Fehler und gibt ihn an den Aufrufer zurück.

Themen

- SDK-Versionen, die zur Laufzeit enthalten sind
- Deaktivierte Python-Funktionen
- Reaktionsformat
- Ordnungsgemäßes Herunterfahren von Erweiterungen
- Lambda-Funktionshandler in Python definieren
- Arbeiten mit ZIP-Dateiarchiven und Python-Lambda-Funktionen
- Bereitstellen von Python-Lambda-Funktionen mit Container-Images
- Arbeiten mit Ebenen für Python Lambda-Funktionen
- Verwenden des Lambda-Kontextobjekts zum Abrufen von Python-Funktionsinformationen
- Python-Lambda-Funktionen protokollieren und überwachen
- AWS Lambda-Funktionstests in Python
- Instrumentieren von Python-Code in AWS Lambda

SDK-Versionen, die zur Laufzeit enthalten sind

Die Version des AWS SDK, die in der Python-Laufzeit enthalten ist, hängt von der Laufzeitversion und Ihrer ab AWS-Region. Um die Version des SDK zu finden, die in der von Ihnen verwendeten Laufzeit enthalten ist, erstellen Sie eine Lambda-Funktion mit dem folgenden Code.

```
import boto3
import botocore

def lambda_handler(event, context):
   print(f'boto3 version: {boto3.__version__}')
   print(f'botocore version: {botocore.__version__}')
```

Deaktivierte Python-Funktionen

In der folgenden Tabelle sind Python-Funktionen aufgeführt, die in den von Lambda verwalteten Laufzeiten und Container-Basis-Images für Python deaktiviert sind. Diese Funktionen müssen aktiviert werden, wenn die ausführbare Python-Runtime-Datei kompiliert wird. Sie können nicht mithilfe eines Ausführungszeitflags aktiviert werden. Um diese Funktionen in Lambda zu verwenden, können Sie Ihren eigenen Python-Runtime-Build mit aktivierten Funktionen bereitstellen, indem Sie ein Container-Image oder eine benutzerdefinierte Laufzeit verwenden.

Python-Funktion	Betroffene Python-Versionen	Status
Just-in-Time (JIT) - Compiler	Python 3.13 und höher	Der JIT-Compiler ist experimentell und wird für Produktionsworkloads nicht empfohlen. Es ist daher in den Lambda-Python-Laufzeiten deaktiviert.
Kostenloses Threading	Python 3.13 und höher	Kostenloses Threading (Option zum Deaktivieren der globalen Interpretersperre) ist in Lambda-Python-Builds deaktiviert, da sich dies auf die Leistung von Single-Thread-Code auswirkt.

Reaktionsformat

In Python 3.12 und späteren Python-Laufzeiten geben Funktionen Unicode-Zeichen als Teil ihrer JSON-Antwort zurück. Frühere Python-Laufzeiten geben Escape-Sequenzen für Unicode-Zeichen in Antworten zurück. Wenn Sie beispielsweise in Python 3.11 eine Unicode-Zeichenfolge wie "こんにちは" zurückgeben, werden die Unicode-Zeichen mit Ecape-Zeichen versehen und es wird "\u3053\u3093\u306b\u3061\u306f" zurückgegeben. Die Python-3.12-Laufzeit gibt das Original "こんにちは" zurück.

Durch die Verwendung von Unicode-Antworten wird die Größe von Lambda-Antworten reduziert, sodass größere Antworten einfacher in die maximale Nutzlastgröße von 6 MB für synchrone Funktionen passen können. Im vorherigen Beispiel hat die Escaped-Version 32 Byte – die Unicode-Zeichenfolge hat 17 Byte.

Wenn Sie auf Python 3.12 oder spätere Python-Laufzeiten aktualisieren, müssen Sie möglicherweise Ihren Code anpassen, um das neue Antwortformat zu berücksichtigen. Wenn der Aufrufer Unicode in Escape-Zeichen erwartet, müssen Sie entweder der zurückgebenden Funktion Code hinzufügen, um den Unicode manuell mit Escape-Zeichen zu versehen, oder den Aufrufer so anpassen, dass er die Unicode-Rückgabe verarbeitet.

Ordnungsgemäßes Herunterfahren von Erweiterungen

Python 3.12 und neuere Python-Laufzeiten bieten verbesserte Funktionen zum ordnungsgemäßen Herunterfahren von Funktionen mit externen Erweiterungen. Wenn Lambda eine Ausführungsumgebung beendet, sendet es ein `SIGTERM`-Signal an die Laufzeitumgebung und dann ein `SHUTDOWN`-Ereignis an jede registrierte externe Erweiterung. Sie können das `SIGTERM`-Signal in Ihrer Lambda-Funktion abfangen und Ressourcen wie Datenbankverbindungen, die von der Funktion erstellt wurden, bereinigen.

Weitere Informationen zum Lebenszyklus der Ausführungsumgebung finden Sie unter Verständnis des Lebenszyklus der Lambda-Ausführungsumgebung. Beispiele für die Verwendung von Graceful Shutdown mit Erweiterungen finden Sie im Samples-Repository.AWS GitHub

Lambda-Funktionshandler in Python definieren

Der Lambda-Funktionshandler ist die Methode in Ihrem Funktionscode, die Ereignisse verarbeitet. Wenn Ihre Funktion aufgerufen wird, führt Lambda die Handler-Methode aus. Ihre Funktion wird so lange ausgeführt, bis der Handler eine Antwort zurückgibt, beendet wird oder ein Timeout auftritt.

Auf dieser Seite wird beschrieben, wie Sie mit Lambda-Funktionshandlern in Python arbeiten, einschließlich Namenskonventionen, gültigen Handler-Signaturen und Best Practices für Code. Diese Seite enthält auch ein Beispiel für eine Python-Lambda-Funktion, die Informationen über einen Auftrag aufnimmt, eine Textdatei als Quittung erstellt und diese Datei in einem Bucket von Amazon Simple Storage Service (Amazon S3) platziert.

Themen

- Beispiel für Python-Lambda-Funktionscode
- Namenskonventionen für Handler
- Verwenden des Lambda-Ereignisobjekts
- Zugreifen auf und Verwenden des Lambda-Kontextobjekts
- Gültige Handler-Signaturen für Python-Handler
- Rückgabe eines Wertes
- Verwenden Sie das AWS SDK für Python (Boto3) in Ihrem Handler
- Zugriff auf Umgebungsvariablen
- Bewährte Codemethoden für Python-Lambda-Funktionen

Beispiel für Python-Lambda-Funktionscode

Das folgende Beispiel für einen Python-Lambda-Funktionscode nimmt Informationen über eine Bestellung auf, erstellt eine Textdateiquittung und platziert diese Datei in einem Amazon-S3-Bucket:

Example Python-Lambda-Funktion

```
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
```

Diese -Datei enthält die folgenden Abschnitte des Codes:

- `import`-Block: Verwenden Sie diesen Block, um Bibliotheken einzubinden, die Ihre Lambda-Funktion benötigt.
- Globale Initialisierung von SDK-Client und Logger: Durch das Einfügen von Initialisierungscode außerhalb des Handlers wird die Wiederverwendung der Ausführungsumgebung genutzt, um die Leistung Ihrer Funktion zu verbessern. Weitere Informationen hierzu finden Sie unter the section called “Bewährte Codemethoden für Python-Lambda-Funktionen”.
- `def upload_receipt_to_s3(bucket_name, key, receipt_content):`: Dies ist eine Hilfsfunktion, die von der `lambda_handler`-Hauptfunktion aufgerufen wird.
- `def lambda_handler(event, context):`: Dies ist die Haupthandler-Funktion für Ihren Code, die Ihre Hauptanwendungslogik enthält. Wenn Lambda Ihren Funktionshandler aufruft, übergibt die Lambda-Laufzeit zwei Argumente an die Funktion: das Ereignisobjekt, das Daten für die Verarbeitung durch Ihre Funktion enthält, und das Kontextobjekt, das Informationen über den Funktionsaufruf enthält.

Namenskonventionen für Handler

Der zum Zeitpunkt der Erstellung einer Lambda-Funktion definierte Funktionshandlername wird abgeleitet von:

- dem Namen der Datei, in der sich die Lambda-Handler-Funktion befindet.

- dem Namen der Python-Handler-Funktion.

Wenn die Datei im obigen Beispiel `lambda_function.py` heißt, wird der Handler als `lambda_function.lambda_handler` angegeben. Dies ist der Standard-Handlername für Funktionen, die Sie mit der Lambda-Konsole erstellen.

Wenn Sie eine Funktion in der Konsole mit einem anderen Dateinamen oder Funktionshandlernamen erstellen, müssen Sie den Standardhandlernamen bearbeiten.

So ändern Sie den Funktionshandlernamen (Konsole)

1. Öffnen Sie die Seite Functions (Funktionen) der Lambda-Konsole und wählen Sie eine Funktion aus.
2. Wählen Sie die Registerkarte Code (Code).
3. Scrollen Sie nach unten zum Bereich Laufzeiteinstellungen und wählen Sie Bearbeiten.
4. Geben Sie unter Handler den neuen Namen für Ihren Funktionshandler ein.
5. Wählen Sie Speichern.

Verwenden des Lambda-Ereignisobjekts

Wenn Lambda Ihre Funktion aufruft, wird dem Funktionshandler ein Ereignisobjekt-Argument übergeben. JSON-Objekte sind das gängigste Ereignisformat für Lambda-Funktionen. Im Codebeispiel im vorherigen Abschnitt erwartet die Funktion eine Eingabe im folgenden Format:

```
{
    "Order_id": "12345",
    "Amount": 199.99,
    "Item": "Wireless Headphones"
}
```

Wenn Ihre Funktion von einem anderen AWS-Service aufgerufen wird, ist das Eingabeereignis ebenfalls ein JSON-Objekt. Das genaue Format des Ereignisobjekts hängt vom Service ab, der Ihre Funktion aufruft. Das Ereignisformat für einen bestimmten Service finden Sie auf der entsprechenden Seite im Kapitel „Integration anderer Services“.

Wenn das Eingabeereignis in Form eines JSON-Objekts vorliegt, konvertiert die Lambda-Laufzeit das Objekt in ein Python-Wörterbuch. Um Werte in der Eingabe-JSON Variablen in Ihrem Code

zuzuweisen, verwenden Sie die Standardmethoden für Python-Wörterbücher, wie im Beispielcode dargestellt.

Sie können Daten auch als JSON-Array oder als einen der anderen gültigen JSON-Datentypen an Ihre Funktion übergeben. In der folgenden Tabelle ist definiert, wie die Python-Laufzeit diese JSON-Typen konvertiert.

JSON-Datentyp	Python-Datentyp
object	Wörterbuch (`dict`)
Array	Liste (`list`)
Zahl	Ganzzahl (`int`) oder Fließkommazahl (`float`)
Zeichenfolge	Zeichenfolge (`str`)
Boolesch	Boolesch (`bool`)
Null	NoneType (`NoneType`)

Zugreifen auf und Verwenden des Lambda-Kontextobjekts

Das Lambda-Kontextobjekt enthält Informationen über den Funktionsaufruf und die Ausführungsumgebung. Lambda übergibt das Kontextobjekt automatisch an Ihre Funktion, wenn diese aufgerufen wird. Mit dem Kontextobjekt können Sie zu Überwachungszwecken Informationen über den Aufruf Ihrer Funktion ausgeben.

Das Kontextobjekt ist eine Python-Klasse, die im Schnittstellenclient der Lambda-Laufzeit definiert ist. Um den Wert einer der Eigenschaften des Kontextobjekts zurückzugeben, verwenden Sie die entsprechende Methode des Kontextobjekts. Im folgenden Codeausschnitt wird beispielsweise der Wert der `aws_request_id`-Eigenschaft (die Kennung für die Aufrufanforderung) einer Variablen namens `request` zugewiesen.

```
request = context.aws_request_id
```

Weitere Informationen zur Verwendung des Lambda-Kontextobjekts und eine vollständige Liste der verfügbaren Methoden und Eigenschaften finden Sie unter the section called “Kontext”.

Gültige Handler-Signaturen für Python-Handler

Wenn Sie Ihre Handler-Funktion in Python definieren, muss die Funktion zwei Argumente enthalten. Das erste dieser Argumente ist das Lambda-Ereignisobjekt und das zweite ist das Lambda-Kontextobjekt. Gemäß Konvention heißen diese Eingabeargumente in der Regel `event` und `context`, Sie können ihnen jedoch beliebige Namen geben. Wenn Sie Ihre Handler-Funktion mit einem einzigen Eingabeargument deklarieren, löst Lambda einen Fehler aus, wenn es versucht, Ihre Funktion auszuführen. Die gängigste Methode zum Deklarieren einer Handler-Funktion in Python ist wie folgt:

```
def lambda_handler(event, context):
```

Sie auch können Python-Typhinweise in Ihrer Funktionsdeklaration verwenden, wie im folgenden Beispiel gezeigt:

```
from typing import Dict, Any

def lambda_handler(event: Dict[str, Any], context: Any) -> Dict[str, Any]:
```

Um spezifische AWS-Typisierungen für Ereignisse zu verwenden, die von anderen AWS-Services generiert werden, sowie für das Kontextobjekt, fügen Sie das Paket `aws-lambda-typing` zum Bereitstellungspaket Ihrer Funktion hinzu. Sie können diese Bibliothek in Ihrer Entwicklungsumgebung installieren, indem Sie **`pip install aws-lambda-typing`** ausführen. Der folgende Codeausschnitt zeigt, wie Sie AWS-spezifische Typhinweise verwenden. In diesem Beispiel ist das erwartete Ereignis ein Amazon-S3-Ereignis.

```
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]:
```

Sie können den Python-Funktionstyp `async` nicht für Ihre Handler-Funktion verwenden.

Rückgabe eines Wertes

Optional kann ein Handler einen Wert zurückgeben, der JSON-serialisierbar sein muss. Zu den gängigen Rückgabetypen gehören `dict`, `list`, `str`, `int`, `float` und `bool`.

Was mit dem zurückgegebenen Wert passiert, hängt von der Aufrufart und dem Service ab, der die Funktion aufgerufen hat. Zum Beispiel:

- Wenn Sie den Aufruftyp `RequestResponse` verwenden, um eine Lambda-Funktion synchron aufzurufen, gibt Lambda das Ergebnis des Python-Funktionsaufrufs an den Client zurück, der die Lambda-Funktion aufruft (in der HTTP-Antwort auf die Aufrufanforderung, serialisiert in JSON). Wenn eine AWS Lambda-Konsole zum Beispiel den Aufruftyp `RequestResponse` verwendet, zeigt die Konsole den zurückgegebenen Wert an, nachdem die Funktion auf der Konsole abgerufen wurde.
- Wenn der Handler Objekte zurückgibt, die nicht von `json.dumps` serialisiert werden können, gibt die Laufzeit einen Fehler zurück.
- Wenn der Handler `None` zurückgibt, wie es Python-Funktionen ohne eine `return`-Anweisung implizit tun, gibt die Laufzeit `null` zurück.
- Wenn Sie den Aufruftyp `Event` (asynchroner Aufruf) verwenden, wird der Wert verworfen.

Im Beispielcode gibt der Handler das folgende Python-Wörterbuch zurück:

```
{
  "statusCode": 200,
  "message": "Receipt processed successfully"
}
```

Die Lambda-Laufzeit serialisiert dieses Wörterbuch und gibt es als JSON-Zeichenfolge an den Client zurück, der die Funktion aufgerufen hat.

> **Note**
>
> In Python 3.9 und späteren Versionen schließt Lambda die requestId des Aufrufs in die Fehlerantwort ein.

Verwenden Sie das AWS SDK für Python (Boto3) in Ihrem Handler

Oft verwenden Sie Lambda-Funktionen, um mit anderen AWS-Services und Ressourcen zu interagieren. Die einfachste Art, eine Schnittstelle zu diesen Ressourcen herzustellen, ist die Verwendung von AWS SDK für Python (Boto3). Alle unterstützten Lambda-Python-Laufzeiten enthalten eine Version des SDK für Python. Wir empfehlen jedoch dringend, das SDK in das

Bereitstellungspaket Ihrer Funktion aufzunehmen, wenn Ihr Code es verwenden muss. Durch das Einfügen des SDK in Ihr Bereitstellungspaket erhalten Sie die volle Kontrolle über Ihre Abhängigkeiten und verringern das Risiko von Versionskonflikten mit anderen Bibliotheken. Weitere Informationen dazu finden Sie unter the section called “Laufzeitabhängigkeiten in Python” und the section called “Abwärtskompatibilität”.

Um das SDK für Python in Ihrer Lambda-Funktion zu verwenden, fügen Sie die folgende Anweisung zum Importblock am Anfang Ihres Funktionscodes hinzu:

```
import boto3
```

Verwenden den `pip install`-Befehl, um Bibliothek `boto3` dem Bereitstellungspaket Ihrer Funktion hinzuzufügen. Ausführliche Anweisungen zum Hinzufügen von Abhängigkeiten zu einem ZIP-Bereitstellungspaket finden Sie unter the section called “ZIP-Bereitstellungspakets mit Abhängigkeiten erstellen”. Weitere Informationen zum Hinzufügen von Abhängigkeiten zu Lambda-Funktionen, die als Container-Images bereitgestellt werden, finden Sie unter the section called “Erstellen eines Images aus einem Base Image” oder the section called “Erstellen eines Images aus einem alternativen Basis-Image”.

Wenn Sie `boto3` in Ihrem Code verwenden, müssen Sie keine Anmeldeinformationen angeben, um einen Client zu initialisieren. Im Beispielcode verwenden wir beispielsweise die folgende Codezeile, um einen Amazon-S3-Client zu initialisieren:

```
# Initialize the S3 client outside of the handler
s3_client = boto3.client('s3')
```

Mit Python erstellt Lambda automatisch Umgebungsvariablen mit Anmeldeinformationen. Das `boto3` SDK überprüft die Umgebungsvariablen Ihrer Funktion bei der Initialisierung auf diese Anmeldeinformationen.

Zugriff auf Umgebungsvariablen

In Ihrem Handler-Code können Sie mithilfe der `os.environ.get`-Methode auf Umgebungsvariablen verweisen. Im Beispielcode verweisen wir mit der folgenden Codezeile auf die definierte `RECEIPT_BUCKET`-Umgebungsvariable:

```
# Access environment variables
bucket_name = os.environ.get('RECEIPT_BUCKET')
```

Vergessen Sie nicht, am Anfang Ihres Codes eine `import os`-Anweisung in den Importblock aufzunehmen.

Bewährte Codemethoden für Python-Lambda-Funktionen

Halten Sie sich an die Richtlinien in der folgenden Liste, um beim Erstellen Ihrer Lambda-Funktionen die besten Codierungspraktiken anzuwenden:

- Trennen Sie den Lambda-Handler von Ihrer Core-Logik. Auf diese Weise können Sie eine Funktion zur besseren Prüfbarkeit von Einheiten schaffen. In Python kann dies beispielsweise wie folgt aussehen:

```
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
```

- Kontrollieren Sie die Abhängigkeiten im Bereitstellungspaket Ihrer Funktion. Die AWS Lambda-Ausführungsumgebung enthält eine Reihe von Bibliotheken. Für die Laufzeiten Node.js und Python gehören dazu die AWS-SDKs. Um die neuesten Funktionen und Sicherheitsupdates zu aktivieren, wird Lambda diese Bibliotheken regelmäßig aktualisieren. Diese Updates können das Verhalten Ihrer Lambda-Funktion geringfügig verändern. Um die Abhängigkeiten, die Ihre Funktion verwendet, vollständig zu kontrollieren, empfehlen wir, alle Abhängigkeiten mit Ihrem Bereitstellungspaket zu bündeln.
- Minimieren Sie die Komplexität Ihrer Abhängigkeiten. Ziehen Sie einfachere Frameworks vor, die sich schnell beim Start der Ausführungsumgebung laden lassen.
- Minimieren Sie die Größe Ihres Bereitstellungspakets auf die für die Laufzeit erforderliche Größe. Dadurch verkürzt sich die Zeit, die für das Herunterladen und Entpacken Ihres Bereitstellungspakets vor dem Aufruf benötigt wird.

Nutzen Sie die Wiederverwendung der Ausführungsumgebung zur Verbesserung Ihrer Funktion. Initialisieren Sie SDK-Clients und Datenbankverbindungen außerhalb des Funktions-Handlers und speichern Sie statische Komponenten lokal im `/tmp`-Verzeichnis. Nachfolgende Aufrufe, die von derselben Instance Ihrer Funktion verarbeitet werden, können diese Ressourcen wiederverwenden. Dies spart Kosten durch Reduzierung der Funktionslaufzeit.

Um potenzielle Datenlecks über Aufrufe hinweg zu vermeiden, verwenden Sie die Ausführungsumgebung nicht, um Benutzerdaten, Ereignisse oder andere Informationen mit Sicherheitsauswirkungen zu speichern. Wenn Ihre Funktion auf einem veränderbaren Zustand beruht, der nicht im Speicher innerhalb des Handlers gespeichert werden kann, sollten Sie für jeden Benutzer eine separate Funktion oder separate Versionen einer Funktion erstellen.

Verwenden Sie eine Keep-Alive-Direktive, um dauerhafte Verbindungen zu pflegen. Lambda bereinigt Leerlaufverbindungen im Laufe der Zeit. Der Versuch, eine Leerlaufverbindung beim Aufruf einer Funktion wiederzuverwenden, führt zu einem Verbindungsfehler. Um Ihre persistente Verbindung aufrechtzuerhalten, verwenden Sie die Keep-Alive-Direktive, die Ihrer Laufzeit zugeordnet ist. Ein Beispiel finden Sie unter Wiederverwenden von Verbindungen mit Keep-Alive in Node.js.

Verwenden Sie Umgebungsvariablen um Betriebsparameter an Ihre Funktion zu übergeben. Wenn Sie z. B. Daten in einen Amazon-S3-Bucket schreiben, anstatt den Bucket-Namen, in den Sie schreiben, hartzucodieren, konfigurieren Sie den Bucket-Namen als Umgebungsvariable.

Vermeiden Sie rekursive Aufrufe in Ihrer Lambda-Funktion, bei denen die Funktion sich selbst aufruft oder einen Prozess initiiert, der die Funktion erneut aufrufen kann. Dies kann zu unvorhergesehenen Mengen an Funktionsaufrufen führen und höhere Kosten zur Folge haben. Wenn Sie eine unbeabsichtigte Menge von Aufrufen feststellen, legen Sie die reservierte gleichzeitige Ausführung der Funktion auf `0` fest, um sofort alle Aufrufe der Funktion zu drosseln, während Sie den Code aktualisieren.

Verwenden Sie keine nicht dokumentierten, nicht öffentlichen APIs in Ihrem Lambda-Funktionscode. Für AWS Lambda-verwaltete Laufzeiten wendet Lambda regelmäßig Sicherheits- und Funktionsupdates auf Lambdas interne APIs an. Diese internen API-Updates können abwärtskompatibel sein, was zu unbeabsichtigten Konsequenzen wie Aufruffehlern führt, wenn Ihre Funktion von diesen nicht öffentlichen APIs abhängig ist. Eine Liste öffentlich zugänglicher APIs finden Sie in der API-Referenz.

Schreiben Sie idempotenten Code. Das Schreiben idempotenter Code für Ihre Funktionen stellt sicher, dass doppelte Ereignisse auf die gleiche Weise behandelt werden. Ihr Code sollte Ereignisse ordnungsgemäß validieren und doppelte Ereignisse ordnungsgemäß behandeln. Weitere Informationen finden Sie unter Wie mache ich meine Lambda-Funktion idempotent?.

Arbeiten mit ZIP-Dateiarchiven und Python-Lambda-Funktionen

Der Code Ihrer AWS Lambda Funktion besteht aus einer .py-Datei, die den Handlercode Ihrer Funktion enthält, zusammen mit allen zusätzlichen Paketen und Modulen, von denen Ihr Code abhängt. Sie verwenden ein Bereitstellungspaket, um Ihren Funktionscode in Lambda bereitzustellen. Dieses Paket kann entweder ein ZIP-Dateiarchiv oder ein Container-Image sein. Weitere Informationen zur Verwendung von Container-Images mit Python finden Sie unter Bereitstellen von Python-Lambda-Funktionen mit Container-Images.

Zum Erstellen des Bereitstellungspakets für ein ZIP-Dateiarchiv können Sie ein integriertes Dienstprogramm für ZIP-Dateien Ihres Befehlzeilen-Tools oder ein anderes Dienstprogramm für ZIP-Dateien verwenden, wie 7zip. In den Beispiele in den folgenden Abschnitten wird davon ausgegangen, dass Sie ein `zip`-Befehlszeilen-Tool in einer Linux- oder MacOS-Umgebung verwenden. Unter Windows können Sie das Windows-Subsystem für Linux installieren, um eine Windows-Version von Ubuntu und Bash zu erhalten und dieselben Befehle zu verwenden.

Da Lambda POSIX-Dateiberechtigungen verwendet, müssen Sie möglicherweise Berechtigungen für den Bereitstellungspaketordner festlegen, bevor Sie das ZIP-Dateiarchiv erstellen.

Themen

- Laufzeitabhängigkeiten in Python
- ZIP-Bereitstellungspakets ohne Abhängigkeiten erstellen
- ZIP-Bereitstellungspakets mit Abhängigkeiten erstellen
- Suchpfad für Abhängigkeiten und integrierte Laufzeit-Bibliotheken
- __pycache__-Ordner verwenden
- ZIP-Bereitstellungspakete mit nativen Bibliotheken erstellen
- Python-Lambda-Funktionen mithilfe von ZIP-Dateien erstellen und aktualisieren

Laufzeitabhängigkeiten in Python

Für Lambda-Funktionen, die die Python-Laufzeit verwenden, kann eine Abhängigkeit ein beliebiges Python-Paket oder -Modul sein. Bei Bereitstellung Ihrer Funktion mithilfe eines ZIP-Archivs können Sie diese Abhängigkeiten entweder mit Ihrem Funktionscode zu Ihrer ZIP-Datei hinzufügen oder eine Lambda-Ebene verwenden. Eine Ebene ist ein separates ZIP-Dateiarchiv, das zusätzlichen Code und andere Daten enthalten kann. Weitere Informationen zur Verwendung von Lambda-Ebenen in Python finden Sie unter the section called “Layer”.

Die Lambda-Python-Laufzeiten beinhalten die AWS SDK für Python (Boto3) und ihre Abhängigkeiten. Lambda stellt das SDK in der Laufzeit für Bereitstellungsszenarien bereit, in denen Sie keine eigenen Abhängigkeiten hinzufügen können. Zu diesen Szenarien gehören die Erstellung von Funktionen in der Konsole mithilfe des integrierten Code-Editors oder die Verwendung von Inline-Funktionen in AWS Serverless Application Model (AWS SAM) oder CloudFormation Vorlagen.

Lambda aktualisiert die Bibliotheken in der Python-Laufzeit regelmäßig, um die neuesten Updates und Sicherheitspatches aufzunehmen. Wenn Ihre Funktion die in der Runtime enthaltene Version des Boto3-SDK verwendet, Ihr Bereitstellungspaket jedoch SDK-Abhängigkeiten enthält, kann dies zu Versionsproblemen führen. Ihr Bereitstellungspaket enthält beispielsweise die SDK-Abhängigkeit urllib3. Aktualisiert Lambda das SDK während der Laufzeit, können Kompatibilitätsprobleme zwischen der neuen Version der Laufzeit und der Version von urllib3 in Ihrem Bereitstellungspaket dazu führen, dass Ihre Funktion fehlschlägt.

> **Important**
>
> Um die volle Kontrolle über Ihre Abhängigkeiten zu behalten und mögliche Versionsprobleme zu vermeiden, empfehlen wir Ihnen, alle Abhängigkeiten Ihrer Funktion zu Ihrem Bereitstellungspaket hinzuzufügen, auch wenn Versionen davon in der Lambda-Laufzeit enthalten sind. Dazu gehört das Boto3-SDK.

Informationen darüber, welche Version des SDK für Python (Boto3) in der von Ihnen verwendeten Laufzeit enthalten ist, finden Sie unter the section called “SDK-Versionen, die zur Laufzeit enthalten sind”.

Im Rahmen des AWS -Modells der geteilten Verantwortung sind Sie für die Verwaltung aller Abhängigkeiten in den Bereitstellungspaketen Ihrer Funktionen verantwortlich. Dies beinhaltet das Durchführen von Updates und Sicherheitspatches. Zum Aktualisieren von Abhängigkeiten im Bereitstellungspaket Ihrer Funktion erstellen Sie zunächst eine neue ZIP-Datei und laden Sie diese dann in Lambda hoch. Weitere Informationen finden Sie unter ZIP-Bereitstellungspakets mit Abhängigkeiten erstellen und Python-Lambda-Funktionen mithilfe von ZIP-Dateien erstellen und aktualisieren.

ZIP-Bereitstellungspakets ohne Abhängigkeiten erstellen

Hat Ihr Funktionscode keine Abhängigkeiten, enthält Ihre ZIP-Datei nur die PY-Datei mit dem Handler-Code Ihrer Funktion. Erstellen Sie mit Ihrem bevorzugten ZIP-Programm eine ZIP-Datei mit

Ihrer PY-Datei im Stammverzeichnis. Befindet sich die PY-Datei nicht im Stammverzeichnis Ihrer ZIP-Datei, kann Lambda Ihren Code nicht ausführen.

Informationen zum Bereitstellen Ihrer ZIP-Datei zum Erstellen einer neuen Lambda-Funktion oder Aktualisieren einer vorhandenen Funktion, finden Sie unter Python-Lambda-Funktionen mithilfe von ZIP-Dateien erstellen und aktualisieren.

ZIP-Bereitstellungspakets mit Abhängigkeiten erstellen

Hängt Ihr Funktionscode von zusätzlichen Paketen und Modulen ab, können Sie diese Abhängigkeiten entweder mit Ihrem Funktionscode zu Ihrer ZIP-Datei hinzufügen oder eine Lambda-Ebene verwenden. Die Anweisungen in diesem Abschnitt zeigen Ihnen, wie Sie Ihre Abhängigkeiten in Ihr ZIP-Bereitstellungspaket aufnehmen. Damit Lambda Ihren Code ausführen kann, muss die PY-Datei, die Ihren Handler-Code und alle Abhängigkeiten Ihrer Funktion enthält, im Stammverzeichnis der ZIP-Datei installiert werden.

Angenommen, Ihr Funktionscode ist in einer Datei mit dem Namen „`lambda_function.py`“ gespeichert. Die folgenden CLI-Befehle erstellen Sie eine ZIP-Datei mit dem Namen „`my_deployment_package.zip`“, die Ihren Funktionscode und seine Abhängigkeiten enthält. Sie können Ihre Abhängigkeiten entweder direkt in einem Ordner in Ihrem Projektverzeichnis installieren oder eine virtuelle Python-Umgebung verwenden.

So erstellen Sie das Bereitstellungspaket (Projektverzeichnis)

1. Navigieren Sie zum Projektverzeichnis, das Ihre `lambda_function.py`-Quellcodedatei enthält. In diesem Beispiel trägt das Verzeichnis den Namen `my_function`.

   ```
   cd my_function
   ```

2. Erstellen Sie ein neues Verzeichnis mit dem Namen „package“, in das Sie Ihre Abhängigkeiten installieren.

   ```
   mkdir package
   ```

   Beachten Sie, dass Lambda bei einem ZIP-Bereitstellungspaket erwartet, dass sich Ihr Quellcode und seine Abhängigkeiten im Stammverzeichnis der ZIP-Datei befinden. Installieren Sie jedoch Abhängigkeiten direkt in Ihrem Projektverzeichnis, kann dies zu einer großen Anzahl neuer Dateien und Ordner führen und das Navigieren in Ihrer IDE erschweren. Sie erstellen hier ein separates `package`-Verzeichnis, um Ihre Abhängigkeiten von Ihrem Quellcode zu trennen.

3. Installieren Sie Ihre Abhängigkeiten im `package`-Verzeichnis. Im folgenden Beispiel wird das Boto3-SDK aus dem Python-Paketindex mithilfe von pip installiert. Verwendet Ihr Funktionscode selbst erstellte Python-Pakete, speichern Sie diese im `package`-Verzeichnis.

   ```
   pip install --target ./package boto3
   ```

4. Erstellen Sie eine ZIP-Datei mit den installierten Bibliotheken im Stammverzeichnis.

   ```
   cd package
   zip -r ../my_deployment_package.zip .
   ```

   Dadurch wird eine `my_deployment_package.zip`-Datei in Ihrem Projektverzeichnis generiert.

5. Fügen Sie die Datei „lambda_function.py“ dem Stammverzeichnis der ZIP-Datei hinzu.

   ```
   cd ..
   zip my_deployment_package.zip lambda_function.py
   ```

   Ihre ZIP-Datei sollte eine flache Verzeichnisstruktur haben, wobei der Handler-Code Ihrer Funktion und alle Ihre Abhängigkeitsordner wie folgt im Stammverzeichnis installiert sind.

   ```
   my_deployment_package.zip
   |- bin
   |  |-jp.py
   |- boto3
   |  |-compat.py
   |  |-data
   |  |-docs
   ...
   |- lambda_function.py
   ```

   Befindet sich die PY-Datei mit dem Handler-Code Ihrer Funktion nicht im Stammverzeichnis Ihrer ZIP-Datei, kann Lambda Ihren Code nicht ausführen.

So erstellen Sie das Bereitstellungspaket (virtuelle Umgebung)

1. Erstellen und aktivieren Sie eine virtuelle Umgebung in Ihrem Projektverzeichnis. In diesem Beispiel trägt das Projektverzeichnis den Namen „`my_function`“.

   ```
   ~$ cd my_function
   ```

```
~/my_function$ python3.14 -m venv my_virtual_env
~/my_function$ source ./my_virtual_env/bin/activate
```

2. Installieren Sie Ihre erforderlichen Bibliotheken mithilfe von pip. Im folgenden Beispiel wird das Boto3-SDK installiert.

```
(my_virtual_env) ~/my_function$ pip install boto3
```

3. Verwenden Sie `pip show`, um den Ort in Ihrer virtuellen Umgebung zu finden, an dem pip Ihre Abhängigkeiten installiert hat.

```
(my_virtual_env) ~/my_function$ pip show <package_name>
```

   Der Ordner, in dem pip Ihre Bibliotheken installiert, heißt entweder „`site-packages`“ oder „`dist-packages`“. Dieser Ordner befindet sich entweder im Verzeichnis „`lib/python3.x`“ oder „`lib64/python3.x`“ (wobei „python3.x“ die verwendete Python-Version ist).

4. Deaktivieren Sie die virtuelle Umgebung.

```
(my_virtual_env) ~/my_function$ deactivate
```

5. Navigieren Sie in das Verzeichnis mit den Abhängigkeiten, die Sie mit pip installiert haben, und erstellen Sie eine ZIP-Datei in Ihrem Projektverzeichnis mit den installierten Abhängigkeiten im Stammverzeichnis. In diesem Beispiel hat pip Ihre Abhängigkeiten im Verzeichnis „`my_virtual_env/lib/python3.14/site-packages`“ installiert.

```
~/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. Navigieren Sie zum Stammverzeichnis Ihres Projektverzeichnisses, in dem sich die PY-Datei mit Ihrem Handler-Code befindet, und fügen Sie diese Datei dem Stammverzeichnis Ihres ZIP-Pakets hinzu. In diesem Beispiel heißt Ihre Funktionscodedatei „`lambda_function.py`“.

```
~/my_function/my_virtual_env/lib/python3.14/site-packages$ cd ../../../../
~/my_function$ zip my_deployment_package.zip lambda_function.py
```

Suchpfad für Abhängigkeiten und integrierte Laufzeit-Bibliotheken

Wenn Sie in Ihrem Code eine `import`-Anweisung verwenden, durchsucht die Python-Laufzeit die Verzeichnisse in ihrem Suchpfad, bis sie das Modul oder Paket findet. Standardmäßig ist der erste Speicherort, den die Laufzeit durchsucht, das Verzeichnis, in das Ihr ZIP-Bereitstellungspaket entpackt und bereitgestellt wird (`/var/task`). Wenn Sie eine Version einer in der Laufzeit enthaltenen Bibliothek in Ihr Bereitstellungspaket einfügen, hat Ihre Version Vorrang vor der Version, die in der Laufzeit enthalten ist. Abhängigkeiten in Ihrem Bereitstellungspaket haben ebenfalls Vorrang vor Abhängigkeiten in Ebenen.

Wenn Sie einer Ebene eine Abhängigkeit hinzufügen, extrahiert Lambda diese in `/opt/python/lib/python3.x/site-packages` (wobei `python3.x` die von Ihnen verwendete Version der Laufzeit ist) oder in `/opt/python`. Im Suchpfad haben diese Verzeichnisse Vorrang vor den Verzeichnissen, die die in der Laufzeit enthaltenen Bibliotheken und die über pip installierten Bibliotheken enthalten (`/var/runtime` und `/var/lang/lib/python3.x/site-packages`). Bibliotheken in Funktionsebenen haben daher Vorrang vor Versionen, die in der Laufzeit enthalten sind.

> **Note**
>
> Im verwalteten Runtime- und Basis-Image von Python 3.11 sind das AWS SDK und seine Abhängigkeiten im `/var/lang/lib/python3.11/site-packages` Verzeichnis installiert.

Sie können den vollständigen Suchpfad für Ihre Lambda-Funktion sehen, indem Sie den folgenden Codeausschnitt hinzufügen.

```
import sys

search_path = sys.path
print(search_path)
```

> **Note**
>
> Da Abhängigkeiten in Ihrem Bereitstellungspaket oder Ihren Ebenen Vorrang vor Bibliotheken haben, die zur Laufzeit gehören, kann es zu Versionsproblemen kommen, wenn Sie eine SDK-Abhängigkeit wie urllib3 in Ihr Paket aufnehmen, ohne das SDK ebenfalls einzubetten. Stellen Sie Ihre eigene Version einer Boto3-Abhängigkeit bereit, müssen Sie Boto3 auch als

Abhängigkeit in Ihrem Bereitstellungspaket bereitstellen. Wir empfehlen, alle Abhängigkeiten Ihrer Funktion in ein Paket aufzunehmen, auch wenn Versionen davon in der Laufzeit enthalten sind.

Fügen Sie Abhängigkeiten alternativ auch in einem separaten Ordner in Ihrem ZIP-Paket hinzu. Fügen Sie beispielsweise eine Version des Boto3-SDK zu einem Ordner in Ihrem ZIP-Paket mit dem Namen „`common`“ hinzu. Wird Ihr ZIP-Paket entpackt und bereitgestellt, wird dieser Ordner im Verzeichnis „`/var/task`“ abgelegt. Mit einer `import from`-Anweisung verwenden Sie eine Abhängigkeit aus einem Ordner Ihres ZIP-Bereitstellungspakets in Ihrem Code. Verwenden Sie mithilfe folgender Anweisung beispielsweise eine Version von Boto3 aus einem Ordner, der in Ihrem ZIP-Paket „`common`“ heißt.

```
from common import boto3
```

__pycache__-Ordner verwenden

Es wird empfohlen, keine `__pycache__`-Ordner in das Bereitstellungspaket Ihrer Funktion aufzunehmen. Python-Bytecode, der auf einem Build-Computer mit einer anderen Architektur oder in einem anderen Betriebssystem kompiliert wurde, ist möglicherweise nicht mit der Lambda-Ausführungsumgebung kompatibel.

ZIP-Bereitstellungspakete mit nativen Bibliotheken erstellen

Verwendet Ihre Funktion nur reine Python-Pakete und -Module, können Sie mit dem `pip install`-Befehl Ihre Abhängigkeiten auf einem beliebigen lokalen Build-Computer installieren und Ihre ZIP-Datei erstellen. Viele beliebte Python-Bibliotheken, darunter NumPy Pandas, sind kein reines Python und enthalten in C oder C++ geschriebenen Code. Fügen Sie Ihrem Bereitstellungspaket Bibliotheken hinzu, die C-/C++-Code enthalten, müssen Sie Ihr Paket korrekt erstellen, damit es mit der Lambda-Ausführungsumgebung kompatibel ist.

Die meisten Pakete im Python-Paketindex (PyPI) sind als „Wheels“ (WHL-Dateien) verfügbar. Eine WHL-Datei ist eine Art von ZIP-Datei, die eine erstellte Distribution mit vorkompilierten Binärdateien für ein bestimmtes Betriebssystem und eine bestimmte Befehlssatzarchitektur enthält. Für die Kompatibilität Ihres Bereitstellungspakets mit Lambda installieren Sie das Wheel für Linux-Betriebssysteme und die Befehlssatzarchitektur Ihrer Funktion.

Einige Pakete sind möglicherweise nur als Quelldistributionen verfügbar. Für diese Pakete müssen Sie die C-/C++-Komponenten selbst kompilieren und erstellen.

Gehen Sie wie folgt vor, um anzuzeigen, welche Distributionen für Ihr erforderliches Paket verfügbar sind:

1. Suchen Sie auf der Hauptseite des Python-Paketindex nach dem Namen des Pakets.
2. Wählen Sie die Paketversion aus, die Sie verwenden möchten.
3. Wählen Sie Dateien zum Herunterladen aus.

Mit erstellten Distributionen (Wheels) arbeiten

Zum Herunterladen eines Wheels, das mit Lambda kompatibel ist, verwenden Sie die pip-Option „`--platform`“.

Verwendet Ihre Lambda-Funktion die x86_64-Befehlssatzarchitektur, führen Sie den folgenden `pip install`-Befehl aus, um ein kompatibles Wheel in Ihrem `package`-Verzeichnis zu installieren. Ersetzen Sie „`--python 3.x`“ durch die Version der verwendeten Python-Laufzeit.

```
pip install \
--platform manylinux2014_x86_64 \
--target=package \
--implementation cp \
--python-version 3.x \
--only-binary=:all: --upgrade \
<package_name>
```

Verwendet Ihre Funktion die arm64-Befehlssatzarchitektur, führen Sie den folgenden Befehl aus. Ersetzen Sie „`--python 3.x`“ durch die Version der verwendeten Python-Laufzeit.

```
pip install \
--platform manylinux2014_aarch64 \
--target=package \
--implementation cp \
--python-version 3.x \
--only-binary=:all: --upgrade \
<package_name>
```

Mit Quelldistributionen arbeiten

Ist Ihr Paket nur als Quelldistribution verfügbar, müssen Sie die C-/C++-Bibliotheken selbst erstellen. Für die Kompatibilität Ihres Pakets mit der Lambda-Ausführungsumgebung müssen Sie es in einer Umgebung erstellen, die dasselbe Amazon-Linux-Betriebssystem verwendet. Sie können dies tun, indem Sie Ihr Paket in einer Amazon Elastic Compute Cloud (Amazon EC2) Linux-Instance erstellen.

Informationen zum Starten und Herstellen einer Verbindung zu einer Amazon EC2 Linux-Instance finden Sie unter Erste Schritte mit Amazon EC2 im EC2 Amazon-Benutzerhandbuch.

Python-Lambda-Funktionen mithilfe von ZIP-Dateien erstellen und aktualisieren

Nach der Erstellung Ihres ZIP-Bereitstellungspakets können Sie es verwenden, um eine neue Lambda-Funktion zu erstellen oder eine vorhandene zu aktualisieren. Sie können Ihr .zip-Paket mithilfe der Lambda-Konsole, der und der AWS Command Line Interface Lambda-API bereitstellen. Sie können Lambda-Funktionen auch mit AWS Serverless Application Model (AWS SAM) und CloudFormation erstellen und aktualisieren.

Die maximale Größe eines ZIP-Bereitstellungspakets für Lambda beträgt 250 MB (entpackt). Beachten Sie, dass dieser Grenzwert für die kombinierte Größe aller hochgeladenen Dateien gilt, einschließlich aller Lambda-Ebenen.

Die Lambda-Laufzeit benötigt die Berechtigung zum Lesen der Dateien in Ihrem Bereitstellungspaket. In der oktalen Schreibweise von Linux-Berechtigungen benötigt Lambda 644 Berechtigungen für nicht ausführbare Dateien (rw-r--r--) und 755 Berechtigungen () für Verzeichnisse und ausführbare Dateien. rwxr-xr-x

Verwenden Sie unter Linux und MacOS den `chmod`-Befehl, um Dateiberechtigungen für Dateien und Verzeichnisse in Ihrem Bereitstellungspaket zu ändern. Um beispielsweise einer nicht ausführbaren Datei die richtigen Berechtigungen zu geben, führen Sie den folgenden Befehl aus.

```
chmod 644 <filepath>
```

Informationen zum Ändern von Dateiberechtigungen in Windows finden Sie unter Festlegen, Anzeigen, Ändern oder Entfernen von Berechtigungen für ein Objekt in der Microsoft-Windows-Dokumentation.

> **Note**
>
> Wenn Sie Lambda nicht die Berechtigungen gewähren, die es für den Zugriff auf Verzeichnisse in Ihrem Bereitstellungspaket benötigt, setzt Lambda die Berechtigungen für diese Verzeichnisse auf 755 (). rwxr-xr-x

Erstellen und Aktualisieren von Funktionen mit ZIP-Dateien unter Verwendung der Konsole

Eine neue Funktion müssen Sie zuerst in der Konsole erstellen und dann Ihr ZIP-Archiv hochladen. Zum Aktualisieren einer bestehenden Funktion öffnen Sie die Seite für Ihre Funktion und gehen dann genauso vor, um Ihre aktualisierte ZIP-Datei hinzuzufügen.

Bei einer ZIP-Datei mit unter 50 MB können Sie eine Funktion erstellen oder aktualisieren, indem Sie die Datei direkt von Ihrem lokalen Computer hochladen. Bei ZIP-Dateien mit einer Größe von mehr als 50 MB müssen Sie Ihr Paket zuerst in einen Amazon-S3-Bucket hochladen. Anweisungen zum Hochladen einer Datei in einen Amazon S3-Bucket mithilfe von finden Sie unter Erste Schritte mit Amazon S3. AWS-Managementkonsole Informationen zum Hochladen von Dateien mit dem AWS CLI finden Sie unter Objekte verschieben im AWS CLI Benutzerhandbuch.

> **Note**
>
> Sie können den Bereitstellungspakettyp (.zip oder Container-Image) für eine vorhandene Funktion nicht ändern. Sie können zum Beispiel eine Container-Image-Funktion nicht so umwandeln, dass sie ein .zip-Archiv verwendet. Sie müssen eine neue Funktion erstellen.

So erstellen Sie eine neue Funktion (Konsole)

1. Öffnen Sie die Funktionsseite der Lambda-Konsole und wählen Sie Funktion erstellen aus.
2. Wählen Sie Author from scratch aus.
3. Führen Sie unter Basic information (Grundlegende Informationen) die folgenden Schritte aus:
   a. Geben Sie als Funktionsname den Namen Ihrer Funktion ein.
   b. Wählen Sie für Laufzeit die Laufzeit aus, die Sie verwenden möchten.

c. (Optional) Für Architektur wählen Sie die Befehlssatz-Architektur für Ihre Funktion aus. Die Standardarchitektur ist x86_64. Stellen Sie sicher, dass das ZIP-Bereitstellungspaket für Ihre Funktion mit der von Ihnen gewählten Befehlssatzarchitektur kompatibel ist.

4. (Optional) Erweitern Sie unter Berechtigungen die Option Standardausführungsrolle ändern. Sie können eine neue Ausführungsrolle erstellen oder eine vorhandene Rolle verwenden.
5. Wählen Sie Funktion erstellen. Lambda erstellt eine grundlegende „Hello World“-Funktion mit der von Ihnen gewählten Laufzeit.

So laden Sie ein ZIP-Archiv von Ihrem lokalen Computer hoch (Konsole)

1. Wählen Sie auf der Funktionsseite der Lambda-Konsole die Funktion aus, für die Sie die ZIP-Datei hochladen möchten.
2. Wählen Sie die Registerkarte Code aus.
3. Wählen Sie im Bereich Codequelle die Option Hochladen von aus.
4. Wählen Sie die ZIP-Datei aus.
5. Laden Sie die ZIP-Datei wie folgt hoch:

   a. Wählen Sie Hochladen und dann Ihre ZIP-Datei in der Dateiauswahl aus.

   b. Klicken Sie auf Open.

   c. Wählen Sie Speichern.

So laden Sie ein ZIP-Archiv aus einem Amazon-S3-Bucket hoch (Konsole)

1. Wählen Sie auf der Funktionsseite der Lambda-Konsole die Funktion aus, für die Sie eine neue ZIP-Datei hochladen möchten.
2. Wählen Sie die Registerkarte Code aus.
3. Wählen Sie im Bereich Codequelle die Option Hochladen von aus.
4. Wählen Sie den Amazon-S3-Speicherort aus.
5. Fügen Sie die Amazon-S3-Link-URL Ihrer ZIP-Datei ein und wählen Sie Speichern aus.

ZIP-Dateifunktionen mithilfe des Konsolencode-Editors aktualisieren

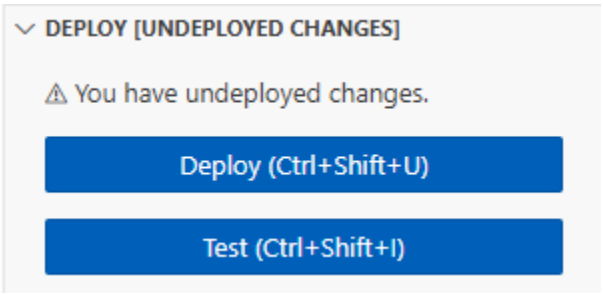
Für einige Funktionen mit ZIP-Bereitstellungspaketen können Sie Ihren Funktionscode direkt mit dem in der Lambda-Konsole integrierten Code-Editor aktualisieren. Zur Verwendung dieses Features muss Ihre Funktion folgende Kriterien erfüllen:

- Ihre Funktion muss eine der interpretierten Sprache der Laufzeit verwenden (Python, Node.js oder Ruby).
- Das Bereitstellungspaket Ihrer Funktion muss kleiner als 50 MB (entpackt) sein.

Funktionscode für Funktionen mit Container-Image-Bereitstellungspaketen kann nicht direkt in der Konsole bearbeitet werden.

So aktualisieren Sie Ihren Funktionscode mit dem Code-Editor

1. Öffnen Sie die Funktionsseite der Lambda-Konsole und wählen Sie Ihre Funktion aus.
2. Wählen Sie die Registerkarte Code aus.
3. Wählen Sie im Bereich Codequelle Ihre Quellcodedatei aus und bearbeiten Sie sie im integrierten Code-Editor.
4. Wählen Sie im Abschnitt BEREITSTELLEN die Option Bereitstellen aus, um den Code Ihrer Funktion zu aktualisieren:



Erstellen und Aktualisieren von Funktionen mit ZIP-Dateien mithilfe der AWS CLI

Sie können die AWS CLI verwenden, um eine neue Funktion zu erstellen oder eine vorhandene unter Verwendung einer ZIP-Datei zu aktualisieren. Verwenden Sie die Erstellungsfunktion und die update-function-codeBefehle, um Ihr .zip-Paket bereitzustellen. Wenn Ihre ZIP-Datei kleiner als 50 MB ist, können Sie das ZIP-Paket von einem Dateispeicherort auf Ihrem lokalen Build-Computer hochladen. Bei größeren Dateien müssen Sie Ihr ZIP-Paket aus einem Amazon-S3-Bucket

hochladen. Anweisungen zum Hochladen einer Datei in einen Amazon S3 S3-Bucket mithilfe von finden Sie unter Objekte verschieben im AWS CLI Benutzerhandbuch. AWS CLI

> **Note**
>
> Wenn Sie Ihre ZIP-Datei mithilfe von aus einem Amazon S3 S3-Bucket hochladen AWS CLI, muss sich der Bucket im selben Verzeichnis befinden AWS-Region wie Ihre Funktion.

Um eine neue Funktion mithilfe einer .zip-Datei mit dem zu erstellen AWS CLI, müssen Sie Folgendes angeben:

- Den Namen Ihrer Funktion (`--function-name`)
- Die Laufzeit Ihrer Funktion (`--runtime`)
- Den Amazon-Ressourcennamen (ARN) der Ausführungsrolle der Funktion (`--role`).
- Den Namen der Handler-Methode in Ihrem Funktionscode (`--handler`)

Sie müssen auch den Speicherort Ihrer ZIP-Datei angeben. Befindet sich Ihre ZIP-Datei in einem Ordner auf Ihrem lokalen Build-Computer, verwenden Sie die `--zip-file`-Option, um den Dateipfad anzugeben, wie im folgenden Beispielbefehl gezeigt.

```
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
```

Zur Angabe des Speicherorts der ZIP-Datei in einem Amazon-S3-Bucket verwenden Sie die im folgenden Beispielbefehl gezeigte `--code`-Option. Sie müssen den `S3ObjectVersion`-Parameter nur für versionierte Objekte verwenden.

```
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
```

Um eine vorhandene Funktion mit der CLI zu aktualisieren, geben Sie den Namen Ihrer Funktion unter Verwendung des `--function-name`-Parameters an. Sie müssen auch den Speicherort der ZIP-Datei angeben, die Sie zum Aktualisieren Ihres Funktionscodes verwenden möchten. Befindet

sich Ihre ZIP-Datei in einem Ordner auf Ihrem lokalen Build-Computer, verwenden Sie die `--zip-file`-Option, um den Dateipfad anzugeben, wie im folgenden Beispielbefehl gezeigt.

```
aws lambda update-function-code --function-name myFunction \
--zip-file fileb://myFunction.zip
```

Zur Angabe des Speicherorts der ZIP-Datei in einem Amazon-S3-Bucket verwenden Sie die im folgenden Beispielbefehl gezeigten `--s3-bucket`- und `--s3-key`-Optionen. Sie müssen den `--s3-object-version`-Parameter nur für versionierte Objekte verwenden.

```
aws lambda update-function-code --function-name myFunction \
--s3-bucket amzn-s3-demo-bucket --s3-key myFileName.zip --s3-object-version myObject
 Version
```

Erstellen und Aktualisieren von Funktionen mit ZIP-Dateien unter Verwendung der Lambda-API

Um Funktionen zu erstellen und zu konfigurieren, die ein ZIP-Dateiarchiv verwenden, verwenden Sie die folgenden API-Operationen:

- CreateFunction
- UpdateFunctionCode

Funktionen mit ZIP-Dateien erstellen und aktualisieren mit AWS SAM

Das AWS Serverless Application Model (AWS SAM) ist ein Toolkit, das dabei hilft, den Prozess der Erstellung und Ausführung serverloser Anwendungen zu optimieren. AWS Sie definieren die Ressourcen für Ihre Anwendung in einer YAML- oder JSON-Vorlage und verwenden die AWS SAM Befehlszeilenschnittstelle (AWS SAM CLI), um Ihre Anwendungen zu erstellen, zu verpacken und bereitzustellen. Wenn Sie eine Lambda-Funktion aus einer AWS SAM Vorlage erstellen, AWS SAM wird automatisch ein ZIP-Bereitstellungspaket oder ein Container-Image mit Ihrem Funktionscode und allen von Ihnen angegebenen Abhängigkeiten erstellt. Weitere Informationen zur Verwendung AWS SAM zum Erstellen und Bereitstellen von Lambda-Funktionen finden Sie unter Erste Schritte mit AWS SAM im AWS Serverless Application Model Entwicklerhandbuch.

Sie können es auch verwenden AWS SAM , um eine Lambda-Funktion mithilfe eines vorhandenen ZIP-Dateiarchivs zu erstellen. Um eine Lambda-Funktion zu erstellen AWS SAM, können Sie Ihre ZIP-Datei in einem Amazon S3 S3-Bucket oder in einem lokalen Ordner auf Ihrem Build-Computer

speichern. Anweisungen zum Hochladen einer Datei in einen Amazon S3 S3-Bucket mithilfe von finden Sie unter Objekte verschieben im AWS CLI Benutzerhandbuch. AWS CLI

In Ihrer AWS SAM Vorlage spezifiziert die `AWS::Serverless::Function` Ressource Ihre Lambda-Funktion. Legen Sie in dieser Ressource die folgenden Eigenschaften fest, um eine Funktion zu erstellen, die als ZIP-Datei-Archiv definiert ist:

- `PackageType` – festlegen auf `Zip`
- `CodeUri`- auf die Amazon S3 S3-URI, den Pfad zum lokalen Ordner oder FunctionCodeObjekt des Funktionscodes gesetzt
- `Runtime` – festlegen auf die gewünschte Laufzeit

Wenn Ihre ZIP-Datei größer als 50 MB ist, müssen Sie sie nicht zuerst in einen Amazon S3 S3-Bucket hochladen. AWS SAM AWS SAM kann .zip-Pakete bis zur maximal zulässigen Größe von 250 MB (entpackt) von einem Speicherort auf Ihrem lokalen Build-Computer hochladen.

Weitere Informationen zum Bereitstellen von Funktionen mithilfe der ZIP-Datei in finden Sie AWS::Serverless::Functionim AWS SAM Entwicklerhandbuch.AWS SAM

Erstellen und Aktualisieren von Funktionen mit ZIP-Dateien mithilfe von CloudFormation

Sie können verwenden CloudFormation , um eine Lambda-Funktion mithilfe eines ZIP-Dateiarchivs zu erstellen. Um eine Lambda-Funktion aus einer ZIP-Datei zu erstellen, müssen Sie Ihre Datei zunächst in einen Amazon-S3-Bucket hochladen. Anweisungen zum Hochladen einer Datei in einen Amazon S3 S3-Bucket mithilfe von finden Sie unter Objekte verschieben im AWS CLI Benutzerhandbuch. AWS CLI

Für Node.js- und Python-Laufzeiten können Sie auch Inline-Quellcode in Ihrer CloudFormation Vorlage bereitstellen. CloudFormation erstellt dann eine ZIP-Datei, die Ihren Code enthält, wenn Sie Ihre Funktion erstellen.

Eine vorhandene ZIP-Datei verwenden

In Ihrer CloudFormation Vorlage spezifiziert die `AWS::Lambda::Function` Ressource Ihre Lambda-Funktion. Legen Sie in dieser Ressource die folgenden Eigenschaften fest, um eine Funktion zu erstellen, die als ZIP-Datei-Archiv definiert ist:

- `PackageType` – festlegen auf `Zip`

- `Code` – Geben Sie den Namen des Amazon-S3-Buckets und den ZIP-Dateinamen in die Felder `S3Bucket` und `S3Key` ein
- `Runtime` – festlegen auf die gewünschte Laufzeit

Eine ZIP-Datei aus dem Inline-Code erstellen

Sie können einfache Funktionen, die in Python oder Node.js geschrieben wurden, inline in einer CloudFormation Vorlage deklarieren. Da der Code in YAML oder JSON eingebettet ist, können Sie Ihrem Bereitstellungspaket keine externen Abhängigkeiten hinzufügen. Das bedeutet, dass Ihre Funktion die Version des AWS SDK verwenden muss, die in der Laufzeit enthalten ist. Die Anforderungen der Vorlage, wie die Notwendigkeit von Escapezeichen für bestimmte Zeichen, erschweren ebenfalls die Verwendung von IDE-Features zur Syntaxprüfung und Codevervollständigung. Ihre Vorlage erfordert deshalb möglicherweise zusätzliche Tests. Aufgrund dieser Einschränkungen eignet sich die Inline-Deklaration von Funktionen am besten für sehr einfachen Code, der sich nicht häufig ändert.

Zum Erstellen einer ZIP-Datei aus Inline-Code für Node.js- und Python-Laufzeiten legen Sie die folgenden Eigenschaften in der Ressource Ihrer `AWS::Lambda::Function`-Vorlage fest:

- `PackageType` – festlegen auf `Zip`
- `Code` – Funktionscode in das `ZipFile`-Feld eingeben
- `Runtime` – festlegen auf die gewünschte Laufzeit

Die CloudFormation generierte ZIP-Datei darf 4 MB nicht überschreiten. Weitere Informationen zum Bereitstellen von Funktionen mithilfe der ZIP-Datei finden Sie AWS::Lambda::Functionim CloudFormation Benutzerhandbuch. CloudFormation

Bereitstellen von Python-Lambda-Funktionen mit Container-Images

Es gibt drei Möglichkeiten, ein Container-Image für eine Python-Lambda-Funktion zu erstellen:

- Verwenden eines AWS Basis-Images für Python

  Die AWS -Basis-Images sind mit einer Sprachlaufzeit, einem Laufzeitschnittstellen-Client zur Verwaltung der Interaktion zwischen Lambda und Ihrem Funktionscode und einem Laufzeitschnittstellen-Emulator für lokale Tests vorinstalliert.

- Es wird ein AWS reines Betriebssystem-Basis-Image verwendet

  AWS Basis-Images nur für Betriebssysteme enthalten eine Amazon Linux-Distribution und den Runtime-Interface-Emulator. Diese Images werden häufig verwendet, um Container-Images für kompilierte Sprachen wie Go und Rust sowie für eine Sprache oder Sprachversion zu erstellen, für die Lambda kein Basis-Image bereitstellt, wie Node.js 19. Sie können reine OS-Basis-Images auch verwenden, um eine benutzerdefinierte Laufzeit zu implementieren. Um das Image mit Lambda kompatibel zu machen, müssen Sie den Laufzeitschnittstellen-Client für Python in das Image aufnehmen.

- Verwenden Sie ein Image, das nicht zur Basisversion gehört AWS

  Sie können auch ein alternatives Basis-Image aus einer anderen Container-Registry verwenden. Sie können auch ein von Ihrer Organisation erstelltes benutzerdefiniertes Image verwenden. Um das Image mit Lambda kompatibel zu machen, müssen Sie den Laufzeitschnittstellen-Client für Python in das Image aufnehmen.

> **Tip**
>
> Um die Zeit zu reduzieren, die benötigt wird, bis Lambda-Container-Funktionen aktiv werden, siehe die Docker-Dokumentation unter Verwenden mehrstufiger Builds. Um effiziente Container-Images zu erstellen, folgen Sie den Bewährte Methoden für das Schreiben von Dockerfiles.

Auf dieser Seite wird erklärt, wie Sie Container-Images für Lambda erstellen, testen und bereitstellen.

Themen

- AWS Basisbilder für Python

- Verwenden eines AWS Basis-Images für Python
- Verwenden eines alternativen Basis-Images mit dem Laufzeitschnittstellen-Client

AWS Basisbilder für Python

AWS stellt die folgenden Basis-Images für Python bereit:

Tags (Markier ngen)	Laufzeit	Betriebss ystem	Dockerfile	Ablehnung
3.14	Python 3.14	Amazon Linux 2023	Dockerfile für Python 3.14 auf GitHub	30. Juni 2029
3.13	Python 3.13	Amazon Linux 2023	Dockerfile für Python 3.13 auf GitHub	30. Juni 2029
3,12	Python 3.12	Amazon Linux 2023	Dockerfile für Python 3.12 auf GitHub	31. Oktober 2028
3.11	Python 3.11	Amazon Linux 2	Dockerfile für Python 3.11 auf GitHub	30. Juni 2027
3,10	Python 3.10	Amazon Linux 2	Dockerfile für Python 3.10 auf GitHub	31. Oktober 2026

Amazon ECR-Repository: gallery.ecr. aws/lambda/python

Die Basis-Images für Python 3.12 und höher basieren auf dem Amazon Linux 2023 Minimal Container Image. Die Python 3.8-3.11-Basis-Images basieren auf dem Amazon Linux 2-Image. AL2023-basierte Images bieten mehrere Vorteile gegenüber Amazon Linux 2, darunter einen geringeren Bereitstellungsaufwand und aktualisierte Versionen von Bibliotheken wie. `glibc`

AL2023-basierte Images verwenden `microdnf` (symbolisiert als`dnf`) als Paketmanager anstelle von`yum`, dem Standard-Paketmanager in Amazon Linux 2. `microdnf`ist eine eigenständige Implementierung von. `dnf` Eine Liste der Pakete, die in AL2 023-basierten Images enthalten sind, finden Sie in den Spalten Minimal Container unter Comparing packages installed on Amazon Linux 2023 Container Images. Weitere Informationen zu den Unterschieden zwischen AL2 023 und

Amazon Linux 2 finden Sie unter Einführung in die Amazon Linux 2023 Runtime for AWS Lambda im AWS Compute-Blog.

> **Note**
>
> Um AL2 023-basierte Images lokal auszuführen, auch mit AWS Serverless Application Model (AWS SAM), müssen Sie Docker-Version 20.10.10 oder höher verwenden.

Suchpfad für Abhängigkeiten in den Basis-Images

Wenn Sie in Ihrem Code eine `import`-Anweisung verwenden, durchsucht die Python-Laufzeit die Verzeichnisse in ihrem Suchpfad, bis sie das Modul oder Paket findet. Standardmäßig durchsucht die Laufzeit zuerst das Verzeichnis `{LAMBDA_TASK_ROOT}`. Wenn Sie eine Version einer in der Laufzeit enthaltenen Bibliothek in Ihr Image einfügen, hat Ihre Version Vorrang vor der Version, die in der Laufzeit enthalten ist.

Andere Schritte im Suchpfad hängen davon ab, welche Version des Lambda-Basis-Images für Python Sie verwenden:

- Python 3.11 und neuer: In der Laufzeit enthaltene Bibliotheken und über pip installierte Bibliotheken werden im Verzeichnis `/var/lang/lib/python3.11/site-packages` installiert. Dieses Verzeichnis hat im Suchpfad Vorrang vor `/var/runtime`. Sie können das SDK außer Kraft setzen, indem Sie pip verwenden, um eine neuere Version zu installieren. Sie können mithilfe von pip überprüfen, ob das in der Laufzeit enthaltene SDK und seine Abhängigkeiten mit allen Paketen kompatibel sind, die Sie installieren.
- Python 3.8-3.10: In der Laufzeit enthaltene Bibliotheken werden im Verzeichnis `/var/runtime` installiert. Über pip installierte Bibliotheken werden im Verzeichnis `/var/lang/lib/python3.x/site-packages` installiert. Das Verzeichnis `/var/runtime` hat im Suchpfad Vorrang vor `/var/lang/lib/python3.x/site-packages`.

Sie können den vollständigen Suchpfad für Ihre Lambda-Funktion sehen, indem Sie den folgenden Codeausschnitt hinzufügen.

```
import sys

search_path = sys.path
print(search_path)
```

Verwenden eines AWS Basis-Images für Python

Voraussetzungen

Zur Durchführung der Schritte in diesem Abschnitt benötigen Sie Folgendes:

- AWS CLI Version 2
- Docker (Mindestversion 25.0.0)
- Das Docker-buildx-Plugin.
- Python

Erstellen eines Images aus einem Base Image

So erstellen Sie ein Container-Image aus einem AWS Basis-Image für Python

1. Erstellen Sie ein Verzeichnis für das Projekt und wechseln Sie dann zu diesem Verzeichnis.

   ```
   mkdir example
   cd example
   ```

2. Erstellen Sie eine neue Datei mit dem Namen `lambda_function.py`. Sie können der Datei zum Testen den folgenden Beispielfunktionscode hinzufügen oder Ihren eigenen verwenden.

   Example Python-Funktion

   ```
   import sys
   def handler(event, context):
       return 'Hello from AWS Lambda using Python' + sys.version + '!'
   ```

3. Erstellen Sie eine neue Datei mit dem Namen `requirements.txt`. Wenn Sie den Beispielfunktionscode aus dem vorherigen Schritt verwenden, können Sie die Datei leer lassen, da es keine Abhängigkeiten gibt. Andernfalls listen Sie jede benötigte Bibliothek auf. So sollte beispielsweise Ihre `requirements.txt` aussehen, wenn Ihre Funktion AWS SDK für Python (Boto3) verwendet:

   Example requirements.txt

   ```
   boto3
   ```

4. Erstellen Sie eine neue Docker-Datei mit der folgenden Konfiguration:

- Setzen Sie die FROM-Eigenschaft auf den URI des Basis-Images.
- Verwenden Sie den Befehl COPY, um den Funktionscode und die Laufzeitabhängigkeiten in {LAMBDA_TASK_ROOT}, eine von Lambda definierte Umgebungsvariable zu kopieren.
- Legen Sie das CMD-Argument auf den Lambda-Funktionshandler fest.

Beachten Sie, dass das Dockerfile-Beispiel keine USER-Anweisung enthält. Wenn Sie ein Container-Image für Lambda bereitstellen, definiert Lambda automatisch einen Standard-Linux-Benutzer mit Berechtigungen mit geringsten Rechten. Dies unterscheidet sich vom Standardverhalten von Docker, bei dem standardmäßig der `root`-Benutzer verwendet wird, wenn keine USER-Anweisung bereitgestellt wird.

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. Erstellen Sie Ihr Docker-Image mit dem docker build-Befehl. Das folgende Beispiel benennt das Bild in `docker-image` und gibt ihm den `test` Tag. Um Ihr Image mit Lambda kompatibel zu machen, müssen Sie die Option `--provenance=false` verwenden.

```
docker buildx build --platform linux/amd64 --provenance=false -t docker-image:test
 .
```

> **Note**
>
> Der Befehl gibt die `--platform linux/amd64`-Option an, um sicherzustellen, dass Ihr Container mit der Lambda-Ausführungsumgebung kompatibel ist, unabhängig von der Architektur des Entwicklungsrechners. Wenn Sie beabsichtigen, eine Lambda-Funktion mithilfe der ARM64 Befehlssatzarchitektur zu erstellen, müssen Sie den Befehl so ändern, dass er stattdessen die `--platform linux/arm64` Option verwendet.

(Optional) Testen Sie das Image lokal

1. Starten Sie Ihr Docker-Image mit dem docker run-Befehl. In diesem Beispiel ist `docker-image` der Image-Name und `test` der Tag.

   ```
   docker run --platform linux/amd64 -p 9000:8080 docker-image:test
   ```

   Dieser Befehl führt das Image als Container aus und erstellt einen lokalen Endpunkt bei `localhost:9000/2015-03-31/functions/function/invocations`.

   > **Note**
   >
   > Wenn Sie das Docker-Image für die ARM64 Befehlssatzarchitektur erstellt haben, stellen Sie sicher, dass Sie stattdessen die `--platform linux/arm64` Option verwenden. `--platform linux/amd64`

2. Veröffentlichen Sie in einem neuen Terminalfenster ein Ereignis an den lokalen Endpunkt.

   Linux/macOS

   Führen Sie unter Linux oder macOS den folgenden `curl`-Befehl aus:

   ```
   curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{}'
   ```

   Dieser Befehl ruft die Funktion mit einem leeren Ereignis auf und gibt eine Antwort zurück. Wenn Sie Ihren eigenen Funktionscode anstelle des Beispielfunktionscodes verwenden, wird empfohlen, die Funktion mit einer JSON-Nutzlast aufzurufen. Beispiel:

```
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d
 '{"payload":"hello world!"}'
```

PowerShell

Führen Sie in PowerShell den folgenden `Invoke-WebRequest` Befehl aus:

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{}' -ContentType "application/json"
```

Dieser Befehl ruft die Funktion mit einem leeren Ereignis auf und gibt eine Antwort zurück. Wenn Sie Ihren eigenen Funktionscode anstelle des Beispielfunktionscodes verwenden, wird empfohlen, die Funktion mit einer JSON-Nutzlast aufzurufen. Beispiel:

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{"payload":"hello world!"}' -ContentType
 "application/json"
```

3. Die Container-ID erhalten.

```
docker ps
```

4. Verwenden Sie den Befehl docker kill, um den Container zu anzuhalten. Ersetzen Sie in diesem Befehl `3766c4ab331c` durch die Container-ID aus dem vorherigen Schritt.

```
docker kill 3766c4ab331c
```

Das Image bereitstellen

Um das Image in Amazon ECR hochzuladen und die Lambda-Funktion zu erstellen

1. Führen Sie den get-login-passwordBefehl aus, um die Docker-CLI bei Ihrer Amazon ECR-Registrierung zu authentifizieren.

   - Setzen Sie den `--region` Wert auf den AWS-Region Ort, an dem Sie das Amazon ECR-Repository erstellen möchten.
   - Ersetzen Sie es `111122223333` durch Ihre AWS-Konto ID.

```
aws ecr get-login-password --region us-east-1 | docker login --username AWS --
password-stdin 111122223333.dkr.ecr.us-east-1.amazonaws.com
```

2. Erstellen Sie ein Repository in Amazon ECR mithilfe des Befehls create-repository.

```
aws ecr create-repository --repository-name hello-world --region us-east-1 --image-
scanning-configuration scanOnPush=true --image-tag-mutability MUTABLE
```

> **Note**
>
> Das Amazon ECR-Repository muss sich im selben Format AWS-Region wie die Lambda-Funktion befinden.

Wenn erfolgreich, sehen Sie eine Antwort wie diese:

```
{
    "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. Kopieren Sie das `repositoryUri` aus der Ausgabe im vorherigen Schritt.
4. Führen Sie den Befehl docker tag aus, um Ihr lokales Image als neueste Version in Ihrem Amazon-ECR-Repository zu markieren. In diesem Befehl gilt Folgendes:

- `docker-image:test` ist der Name und das Tag Ihres Docker-Images. Dies sind der Imagename und das Tag, die Sie im `docker build`-Befehl angegeben haben.
- Ersetzen Sie <ECRrepositoryUri> durch den `repositoryUri`, den Sie kopiert haben. Stellen Sie sicher, dass Sie `:latest` am Ende der URI angeben.

```
docker tag docker-image:test <ECRrepositoryUri>:latest
```

Beispiel:

```
docker tag docker-image:test 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

5. Führen Sie den Befehl docker push aus, um Ihr lokales Image im Amazon-ECR-Repository bereitzustellen. Stellen Sie sicher, dass Sie `:latest` am Ende der Repository-URI angeben.

```
docker push 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

6. Erstellen Sie eine Ausführungsrolle für die Funktion, wenn Sie noch keine haben. Sie benötigen den Amazon-Ressourcennamen (ARN) der Rolle im nächsten Schritt.
7. So erstellen Sie die Lambda-Funktion: Geben Sie für `ImageUri` die Repository-URI von zuvor an. Stellen Sie sicher, dass Sie `:latest` am Ende der URI angeben.

```
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**
>
> Sie können eine Funktion mit einem Bild in einem anderen AWS Konto erstellen, sofern sich das Bild in derselben Region wie die Lambda-Funktion befindet. Weitere Informationen finden Sie unter Kontoübergreifende Berechtigungen von Amazon ECR.

8. Die Funktion aufrufen.

```
aws lambda invoke --function-name hello-world response.json
```

Das Ergebnis sollte ungefähr wie folgt aussehen:

```
{
  "ExecutedVersion": "$LATEST",
  "StatusCode": 200
}
```

9. Um die Ausgabe der Funktion zu sehen, überprüfen Sie die `response.json`-Datei.

Um den Funktionscode zu aktualisieren, müssen Sie das Image erneut erstellen, das neue Image in das Amazon ECR-Repository hochladen und dann den update-function-codeBefehl verwenden, um das Image für die Lambda-Funktion bereitzustellen.

Lambda löst das Image-Tag in einen bestimmten Image-Digest auf. Das heißt, wenn Sie das Image-Tag, das zur Bereitstellung der Funktion verwendet wurde, auf ein neues Image in Amazon ECR verweisen, aktualisiert Lambda die Funktion nicht automatisch, um das neue Image zu verwenden.

Um das neue Image für dieselbe Lambda-Funktion bereitzustellen, müssen Sie den update-function-codeBefehl verwenden, auch wenn das Image-Tag in Amazon ECR gleich bleibt. Im folgenden Beispiel erstellt die `--publish`-Option eine neue Version der Funktion unter Verwendung des aktualisierten Container-Images.

```
aws lambda update-function-code \
  --function-name hello-world \
  --image-uri 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --publish
```

Verwenden eines alternativen Basis-Images mit dem Laufzeitschnittstellen-Client

Wenn Sie ein OS-Basis-Image oder ein alternatives Basis-Image verwenden, müssen Sie den Laufzeitschnittstellen-Client in das Image einbinden. Der Laufzeitschnittstellen-Client erweitert die Laufzeit-API, die die Interaktion zwischen Lambda und Ihrem Funktionscode verwaltet.

Installieren Sie den Laufzeitschnittstellen-Client für Python mit dem pip-Paketmanager:

```
pip install awslambdaric
```

Sie können den Python-Runtime-Interface-Client auch von herunterladen GitHub.

Das folgende Beispiel zeigt, wie ein Container-Image für Python mit einem AWS Nicht-Base-Image erstellt wird. Das Beispiel-Dockerfile verwendet ein offizielles Python-Basis-Image. Das Dockerfile enthält den Laufzeitschnittstellen-Client für Python.

Voraussetzungen

Zur Durchführung der Schritte in diesem Abschnitt benötigen Sie Folgendes:

- AWS CLI Version 2
- Docker (Mindestversion 25.0.0)
- Das Docker-buildx-Plugin.
- Python

Erstellen eines Images aus einem alternativen Basis-Image

Um ein Container-Image aus einem AWS Nicht-Basis-Image zu erstellen

1. Erstellen Sie ein Verzeichnis für das Projekt und wechseln Sie dann zu diesem Verzeichnis.

   ```
   mkdir example
   cd example
   ```

2. Erstellen Sie eine neue Datei mit dem Namen `lambda_function.py`. Sie können der Datei zum Testen den folgenden Beispielfunktionscode hinzufügen oder Ihren eigenen verwenden.

   Example Python-Funktion

   ```
   import sys
   def handler(event, context):
       return 'Hello from AWS Lambda using Python' + sys.version + '!'
   ```

3. Erstellen Sie eine neue Datei mit dem Namen `requirements.txt`. Wenn Sie den Beispielfunktionscode aus dem vorherigen Schritt verwenden, können Sie die Datei leer lassen, da es keine Abhängigkeiten gibt. Andernfalls listen Sie jede benötigte Bibliothek auf. So sollte

beispielsweise Ihre `requirements.txt` aussehen, wenn Ihre Funktion AWS SDK für Python (Boto3) verwendet:

Example requirements.txt

```
boto3
```

4. Erstellen Sie eine neue Docker-Datei. Das folgende Dockerfile verwendet ein offizielles Python-Basis-Image anstelle eines AWS -Basis-Images. Das Dockerfile enthält den Laufzeitschnittstellen-Client, der das Image mit Lambda kompatibel macht. Die folgende Beispiel-Docker-Datei verwendet eine mehrstufige Entwicklung.

   - Legen Sie die `FROM`-Eigenschaft auf das Basis-Image fest.
   - Legen Sie `ENTRYPOINT` auf das Modul fest, das der Docker-Container beim Start ausführen soll. In diesem Fall ist das Modul der Laufzeitschnittstellen-Client.
   - Legen Sie `CMD` auf den Lambda-Funktionshandler fest.

   Beachten Sie, dass das Dockerfile-Beispiel keine USER-Anweisung enthält. Wenn Sie ein Container-Image für Lambda bereitstellen, definiert Lambda automatisch einen Standard-Linux-Benutzer mit Berechtigungen mit geringsten Rechten. Dies unterscheidet sich vom Standardverhalten von Docker, bei dem standardmäßig der `root`-Benutzer verwendet wird, wenn keine `USER`-Anweisung bereitgestellt wird.

   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. Erstellen Sie Ihr Docker-Image mit dem docker build-Befehl. Das folgende Beispiel benennt das Bild in `docker-image` und gibt ihm den `test` Tag. Um Ihr Image mit Lambda kompatibel zu machen, müssen Sie die Option `--provenance=false` verwenden.

```
docker buildx build --platform linux/amd64 --provenance=false -t docker-image:test
 .
```

> **Note**
>
> Der Befehl gibt die `--platform linux/amd64`-Option an, um sicherzustellen, dass Ihr Container mit der Lambda-Ausführungsumgebung kompatibel ist, unabhängig von der Architektur des Entwicklungsrechners. Wenn Sie beabsichtigen, eine Lambda-Funktion mithilfe der ARM64 Befehlssatzarchitektur zu erstellen, müssen Sie den Befehl so ändern, dass er stattdessen die `--platform linux/arm64` Option verwendet.

(Optional) Testen Sie das Image lokal

Verwenden Sie den Laufzeit-Schnittstellen-Emulator, um das Image lokal zu testen. Sie können den Emulator in Ihr Image integrieren oder ihn mit dem folgenden Verfahren auf Ihrem lokalen Computer installieren.

Installieren des Laufzeitschnittstellen-Emulators auf Ihrem lokalen Computer

1. Führen Sie in Ihrem Projektverzeichnis den folgenden Befehl aus, um den Runtime-Interface-Emulator (x86-64-Architektur) herunterzuladen GitHub und auf Ihrem lokalen Computer zu installieren.

   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
   ```

   Um den arm64-Emulator zu installieren, ersetzen Sie die GitHub Repository-URL im vorherigen Befehl durch Folgendes:

   ```
   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
   ```

   Um den arm64-Emulator zu installieren, ersetzen Sie das `$downloadLink` durch Folgendes:

   ```
   https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/download/aws-lambda-rie-arm64
   ```

2. Starten Sie Ihr Docker-Image mit dem docker run-Befehl. Beachten Sie Folgendes:

   - `docker-image` ist der Image-Name und `test` ist das Tag.

- `/usr/local/bin/python -m awslambdaric lambda_function.handler` ist der `ENTRYPOINT` gefolgt von dem `CMD` aus Ihrem 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
```

Dieser Befehl führt das Image als Container aus und erstellt einen lokalen Endpunkt bei `localhost:9000/2015-03-31/functions/function/invocations`.

> **Note**
>
> Wenn Sie das Docker-Image für die ARM64 Befehlssatzarchitektur erstellt haben, stellen Sie sicher, dass Sie die `--platform linux/arm64` Option anstelle von verwenden. `--platform linux/amd64`

3. Veröffentlichen Sie ein Ereignis auf dem lokalen Endpunkt.

Linux/macOS

Führen Sie unter Linux oder macOS den folgenden `curl`-Befehl aus:

```
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{}'
```

Dieser Befehl ruft die Funktion mit einem leeren Ereignis auf und gibt eine Antwort zurück. Wenn Sie Ihren eigenen Funktionscode anstelle des Beispielfunktionscodes verwenden, wird empfohlen, die Funktion mit einer JSON-Nutzlast aufzurufen. Beispiel:

```
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d
 '{"payload":"hello world!"}'
```

PowerShell

Führen Sie in PowerShell den folgenden `Invoke-WebRequest` Befehl aus:

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{}' -ContentType "application/json"
```

Dieser Befehl ruft die Funktion mit einem leeren Ereignis auf und gibt eine Antwort zurück. Wenn Sie Ihren eigenen Funktionscode anstelle des Beispielfunktionscodes verwenden, wird empfohlen, die Funktion mit einer JSON-Nutzlast aufzurufen. Beispiel:

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{"payload":"hello world!"}' -ContentType
 "application/json"
```

4. Die Container-ID erhalten.

```
docker ps
```

5. Verwenden Sie den Befehl docker kill, um den Container zu anzuhalten. Ersetzen Sie in diesem Befehl `3766c4ab331c` durch die Container-ID aus dem vorherigen Schritt.

```
docker kill 3766c4ab331c
```

Das Image bereitstellen

Um das Image in Amazon ECR hochzuladen und die Lambda-Funktion zu erstellen

1. Führen Sie den get-login-passwordBefehl aus, um die Docker-CLI bei Ihrer Amazon ECR-Registrierung zu authentifizieren.

- Setzen Sie den `--region` Wert auf den AWS-Region Ort, an dem Sie das Amazon ECR-Repository erstellen möchten.
- Ersetzen Sie es `111122223333` durch Ihre AWS-Konto ID.

```
aws ecr get-login-password --region us-east-1 | docker login --username AWS --
password-stdin 111122223333.dkr.ecr.us-east-1.amazonaws.com
```

2. Erstellen Sie ein Repository in Amazon ECR mithilfe des Befehls create-repository.

```
aws ecr create-repository --repository-name hello-world --region us-east-1 --image-
scanning-configuration scanOnPush=true --image-tag-mutability MUTABLE
```

> **Note**
>
> Das Amazon ECR-Repository muss sich im selben Format AWS-Region wie die Lambda-Funktion befinden.

Wenn erfolgreich, sehen Sie eine Antwort wie diese:

```
{
    "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. Kopieren Sie das `repositoryUri` aus der Ausgabe im vorherigen Schritt.
4. Führen Sie den Befehl docker tag aus, um Ihr lokales Image als neueste Version in Ihrem Amazon-ECR-Repository zu markieren. In diesem Befehl gilt Folgendes:

   - `docker-image:test` ist der Name und das Tag Ihres Docker-Images. Dies sind der Imagename und das Tag, die Sie im `docker build`-Befehl angegeben haben.
   - Ersetzen Sie `<ECRrepositoryUri>` durch den `repositoryUri`, den Sie kopiert haben. Stellen Sie sicher, dass Sie `:latest` am Ende der URI angeben.

   ```
   docker tag docker-image:test <ECRrepositoryUri>:latest
   ```

   Beispiel:

   ```
   docker tag docker-image:test 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
   ```

5. Führen Sie den Befehl docker push aus, um Ihr lokales Image im Amazon-ECR-Repository bereitzustellen. Stellen Sie sicher, dass Sie `:latest` am Ende der Repository-URI angeben.

   ```
   docker push 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
   ```

6. Erstellen Sie eine Ausführungsrolle für die Funktion, wenn Sie noch keine haben. Sie benötigen den Amazon-Ressourcennamen (ARN) der Rolle im nächsten Schritt.
7. So erstellen Sie die Lambda-Funktion: Geben Sie für `ImageUri` die Repository-URI von zuvor an. Stellen Sie sicher, dass Sie `:latest` am Ende der URI angeben.

   ```
   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**
   >
   > Sie können eine Funktion mit einem Bild in einem anderen AWS Konto erstellen, sofern sich das Bild in derselben Region wie die Lambda-Funktion befindet. Weitere Informationen finden Sie unter Kontoübergreifende Berechtigungen von Amazon ECR.

8. Die Funktion aufrufen.

```
aws lambda invoke --function-name hello-world response.json
```

Das Ergebnis sollte ungefähr wie folgt aussehen:

```
{
  "ExecutedVersion": "$LATEST",
  "StatusCode": 200
}
```

9. Um die Ausgabe der Funktion zu sehen, überprüfen Sie die `response.json`-Datei.

Um den Funktionscode zu aktualisieren, müssen Sie das Image erneut erstellen, das neue Image in das Amazon ECR-Repository hochladen und dann den update-function-codeBefehl verwenden, um das Image für die Lambda-Funktion bereitzustellen.

Lambda löst das Image-Tag in einen bestimmten Image-Digest auf. Das heißt, wenn Sie das Image-Tag, das zur Bereitstellung der Funktion verwendet wurde, auf ein neues Image in Amazon ECR verweisen, aktualisiert Lambda die Funktion nicht automatisch, um das neue Image zu verwenden.

Um das neue Image für dieselbe Lambda-Funktion bereitzustellen, müssen Sie den update-function-codeBefehl verwenden, auch wenn das Image-Tag in Amazon ECR gleich bleibt. Im folgenden Beispiel erstellt die `--publish`-Option eine neue Version der Funktion unter Verwendung des aktualisierten Container-Images.

```
aws lambda update-function-code \
  --function-name hello-world \
  --image-uri 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --publish
```

Ein Beispiel für das Erstellen eines Python-Images aus einem Alpine-Basis-Image finden Sie unter Unterstützung für Container-Images für Lambda im AWS Blog.

Arbeiten mit Ebenen für Python Lambda-Funktionen

Verwenden Sie Lambda-Ebenen, um Code und Abhängigkeiten zu bündeln, die Sie für mehrere Funktionen wiederverwenden möchten. Ebenen enthalten üblicherweise Bibliotheksabhängigkeiten, eine benutzerdefinierte Laufzeit oder Konfigurationsdateien. Das Erstellen einer Ebene umfasst drei allgemeine Schritte:

1. Verpacken Ihres Ebeneninhalts. Dies bedeutet, dass Sie ein .zip-Dateiarchiv erstellen, das die Abhängigkeiten enthält, die Sie in Ihren Funktionen verwenden möchten.
2. Erstellen Sie die Ebene in Lambda.
3. Fügen Sie die Ebene zu Ihren Funktionen hinzu.

Themen

- Verpacken Ihres Ebeneninhalts
- Erstellen Sie die Ebene in Lambda
- Fügen Sie die Ebene zu Ihrer Funktion hinzu
- Beispiel-App

Verpacken Ihres Ebeneninhalts

Um eine Ebene zu erstellen, bündeln Sie Ihre Pakete in einem ZIP-Dateiarchiv, das die folgenden Anforderungen erfüllt:

- Erstellen Sie die Ebene mithilfe derselben Version von Python, die Sie für die Lambda-Funktion verwenden möchten. Wenn Sie Ihren Layer beispielsweise mit Python 3.14 erstellen, verwenden Sie die Python 3.14-Laufzeit für Ihre Funktion.
- Ihre ZIP-Datei muss ein `python`-Verzeichnis auf der Stammebene enthalten.
- Die Pakete in Ihrer Ebene müssen mit Linux kompatibel sein. Lambda-Funktionen werden auf Amazon Linux ausgeführt.

Sie können Ebenen erstellen, die entweder mit `pip` installierte Python-Bibliotheken von Drittanbietern (wie `requests` oder `pandas`) oder Ihre eigenen Python-Module und -Pakete enthalten.

Abhängigkeiten von Drittanbietern

So erstellen Sie eine Ebene mit pip-Paketen

1. Wählen Sie eine der folgenden Methoden, um `pip`-Pakete in das erforderliche Verzeichnis der obersten Ebene (`python/`) zu installieren:

   pip install

   Für reine Python-Pakete (wie Anfragen oder boto3):

   ```
   pip install requests -t python/
   ```

   Einige Python-Pakete, wie NumPy Pandas, enthalten kompilierte C-Komponenten. Wenn Sie eine Ebene mit diesen Paketen unter macOS oder Windows erstellen, müssen Sie möglicherweise diesen Befehl verwenden, um ein Linux-kompatibles Wheel zu installieren:

   ```
   pip install numpy --platform manylinux2014_x86_64 --only-binary=:all: -t python/
   ```

   Weitere Informationen zur Arbeit mit Python-Paketen, die kompilierte Komponenten enthalten, finden Sie unter ZIP-Bereitstellungspakete mit nativen Bibliotheken erstellen.

   requirements.txt

   Mithilfe einer `requirements.txt`-Datei können Sie Paketversionen verwalten und konsistente Installationen sicherstellen.

   Example requirements.txt

   ```
   requests==2.31.0
   boto3==1.37.34
   numpy==1.26.4
   ```

   Wenn Ihre `requirements.txt`-Datei nur reine Python-Pakete enthält (wie Anfragen oder boto3):

   ```
   pip install -r requirements.txt -t python/
   ```

Einige Python-Pakete, wie NumPy Pandas, enthalten kompilierte C-Komponenten. Wenn Sie eine Ebene mit diesen Paketen unter macOS oder Windows erstellen, müssen Sie möglicherweise diesen Befehl verwenden, um ein Linux-kompatibles Wheel zu installieren:

```
pip install -r requirements.txt --platform manylinux2014_x86_64 --only-
binary=:all: -t python/
```

Weitere Informationen zur Arbeit mit Python-Paketen, die kompilierte Komponenten enthalten, finden Sie unter ZIP-Bereitstellungspakete mit nativen Bibliotheken erstellen.

2. Zippen Sie den Inhalt des `python`-Verzeichnisses.

Linux/macOS

```
zip -r layer.zip python/
```

PowerShell

```
Compress-Archive -Path .\python -DestinationPath .\layer.zip
```

Die Verzeichnisstruktur Ihrer ZIP-Datei sollte wie folgt aussehen:

```
python/              # Required top-level directory
### requests/
### boto3/
### numpy/
### (dependencies of the other packages)
```

> **Note**
>
> Wenn Sie eine virtuelle Python-Umgebung (venv) zur Installation von Paketen verwenden, sieht Ihre Verzeichnisstruktur anders aus (zum Beispiel `python/lib/python3.x/site-packages`). Solange Ihre ZIP-Datei das `python`-Verzeichnis auf der Stammebene enthält, kann Lambda Ihre Pakete finden und importieren.

Benutzerdefinierte Python-Module

So erstellen Sie eine Ebene mithilfe Ihres eigenen Codes

1. Erstellen Sie das erforderliche Verzeichnis der obersten Ebene für Ihre Ebene:

```
mkdir python
```

2. Erstellen Sie Ihre Python-Module im `python`-Verzeichnis. Das folgende Beispielmodul validiert Bestellungen, indem es überprüft, ob sie die erforderlichen Informationen enthalten.

   Example Benutzerdefiniertes Modul: 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. Zippen Sie den Inhalt des `python`-Verzeichnisses.

   Linux/macOS

   ```
   zip -r layer.zip python/
   ```

   PowerShell

   ```
   Compress-Archive -Path .\python -DestinationPath .\layer.zip
   ```

   Die Verzeichnisstruktur Ihrer ZIP-Datei sollte wie folgt aussehen:

   ```
   python/     # Required top-level directory
   ### validator.py
   ```

4. Importieren und verwenden Sie die Module in Ihrer Funktion wie bei jedem anderen Python-Paket. Beispiel:

   ```
   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'
        })
```

Sie können das folgende Testereignis verwenden, um die Funktion aufzurufen:

```
{
    "body": "{\"product_id\": \"ABC123\", \"quantity\": 2, \"priority\": \"express
\"}"
}
```

Erwartete Antwort:

```
{
  "statusCode": 200,
  "body": "{\"message\": \"Order validated successfully\", \"order\": {\"product_id
\": \"ABC123\", \"quantity\": 2, \"shipping_priority\": \"express\"}}"
}
```

Erstellen Sie die Ebene in Lambda

Sie können Ihren Layer entweder mit der AWS CLI oder der Lambda-Konsole veröffentlichen.

AWS CLI

Führen Sie den publish-layer-version AWS CLI Befehl aus, um die Lambda-Schicht zu erstellen:

```
aws lambda publish-layer-version --layer-name my-layer --zip-file fileb://layer.zip
 --compatible-runtimes python3.14
```

Der Parameter Kompatible Laufzeiten ist optional. Wenn angegeben, verwendet Lambda diesen Parameter zum Filtern von Ebenen in der Lambda-Konsole.

Console

So erstellen Sie eine Ebene (Konsole)

1. Öffnen Sie die Seite Ebenen der Lambda-Konsole.
2. Wählen Sie Create Layer (Ebene erstellen) aus.
3. Wählen Sie ZIP-Datei hochladen und laden Sie dann das zuvor erstellte ZIP-Archiv hoch.

4. (Optional) Wählen Sie für Kompatible Laufzeiten die Python-Laufzeit aus, die der Python-Version entspricht, mit der Sie die Ebene erstellt haben.
5. Wählen Sie Erstellen aus.

Fügen Sie die Ebene zu Ihrer Funktion hinzu

AWS CLI

Führen Sie den update-function-configuration AWS CLI Befehl aus, um die Ebene an Ihre Funktion anzuhängen. Verwenden Sie den ARN der Ebene für den `--layers`-Parameter. Der ARN muss die Version angeben (z. B. `arn:aws:lambda:us-east-1:123456789012:layer:my-layer:1`). Weitere Informationen finden Sie unter Ebenen und Ebenenversionen.

```
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"
```

Die cli-binary-format Option ist erforderlich, wenn Sie AWS CLI Version 2 verwenden. Um dies zur Standardeinstellung zu machen, führen Sie `aws configure set cli-binary-format raw-in-base64-out` aus. Weitere Informationen finden Sie unter Von AWS CLI unterstützte globale Befehlszeilenoptionen im AWS Command Line Interface -Benutzerhandbuch für Version 2.

Console

So fügen Sie eine Ebene zu einer Funktion hinzu

1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.
2. Wählen Sie die Funktion aus.
3. Blättern Sie nach unten bis zum Abschnitt Ebenen und wählen Sie dann Eine Ebene hinzufügen.
4. Wählen Sie unter Ebene auswählen die Option Benutzerdefinierte Ebenen und dann Ihre Ebene aus.

> **Note**
>
> Wenn Sie bei der Erstellung der Ebene keine kompatible Laufzeit hinzugefügt haben, wird die Ebene hier nicht aufgeführt. Sie können stattdessen den ARN der Ebene angeben.

5. Wählen Sie Hinzufügen aus.

Beispiel-App

Weitere Beispiele für die Verwendung von Lambda-Layern finden Sie in der Layer-Python-Beispielanwendung im AWS Lambda Developer Guide-Repository. GitHub Diese Anwendung umfasst zwei Ebenen, die Python-Bibliotheken enthalten. Nach dem Erstellen der Ebenen können Sie die entsprechende Funktionen bereitstellen und aufrufen, um zu bestätigen, dass die Ebenen wie erwartet funktionieren.

Verwenden des Lambda-Kontextobjekts zum Abrufen von Python-Funktionsinformationen

Wenn Lambda Ihre Funktion ausführt, wird ein Context-Objekt an den Handler. übergeben. Dieses Objekt stellt Methoden und Eigenschaften mit Informationen zum Aufruf, zur Funktion und zur Ausführungsumgebung bereit. Weitere Informationen darüber, wie das Kontextobjekt an den Funktions-Handler übergeben wird, finden Sie unter Lambda-Funktionshandler in Python definieren.

Context-Methoden

- `get_remaining_time_in_millis` – Gibt die Anzahl der verbleibenden Millisekunden zurück, bevor die Ausführung das Zeitlimit überschreitet.

Context-Eigenschaften

- `function_name` – Der Name der Lambda-Funktion.
- `function_version` – Die Version der Funktion.
- `invoked_function_arn` – Der Amazon-Ressourcenname (ARN), der zum Aufrufen der Funktion verwendet wird. Gibt an, ob der Aufrufer eine Versionsnummer oder einen Alias angegeben hat.
- `memory_limit_in_mb` – Die Menge an Arbeitsspeicher, die der Funktion zugewiesen ist.
- `aws_request_id` – Der Bezeichner der Aufrufanforderung.
- `log_group_name` – Protokollgruppe für die Funktion.
- `log_stream_name` – Der Protokollstream für die Funktions-Instance.
- `identity` – Informationen zur Amazon-Cognito-Identität, die die Anforderung autorisiert hat.
  - `cognito_identity_id`— Die authentifizierte Amazon-Cognito-Identität.
  - `cognito_identity_pool_id` – Der Amazon-Cognito-Identitätspool, der den Aufruf autorisiert hat.
- `client_context` – (mobile Apps) Clientkontext, der Lambda von der Clientanwendung bereitgestellt wird.
  - `client.installation_id`
  - `client.app_title`
  - `client.app_version_name`
  - `client.app_version_code`
  - `client.app_package_name`

- `custom` – Ein `dict` mit benutzerdefinierten Werten, die von der mobilen Clientanwendung festgelegt wurden.
- `env` – Ein `dict` mit Umgebungsinformationen, die vom AWS SDK bereitgestellt wurden.

Powertools für Lambda (Python) bietet eine Schnittstellendefinition für das Lambda-Kontextobjekt. Sie können die Schnittstellendefinition für Typhinweise verwenden oder die Struktur des Lambda-Kontextobjekts genauer untersuchen. Die Schnittstellendefinition finden Sie unter lambda_context.py im powertools-lambda-python auf GitHub.

Das folgende Beispiel zeigt eine Handler-Funktion zur Protokollierung von Context-Informationen.

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())
```

Zusätzlich zu den oben aufgeführten Optionen können Sie das AWS X-Ray-SDK für Instrumentieren von Python-Code in AWS Lambda zudem verwenden, um kritische Code-Pfade zu identifizieren, deren Leistung nachzuverfolgen und die Daten für die Analyse zu erfassen.

Python-Lambda-Funktionen protokollieren und überwachen

AWS Lambda überwacht automatisch Lambda-Funktionen und sendet Protokolleinträge an Amazon CloudWatch. Ihre Lambda-Funktion enthält eine CloudWatch Logs-Log-Gruppe und einen Log-Stream für jede Instanz Ihrer Funktion. Die Lambda-Laufzeitumgebung sendet Details zu den einzelnen Aufrufen an den Protokollstream und leitet Protokolle und andere Ausgaben aus dem Code Ihrer Funktion weiter. Weitere Informationen zu CloudWatch Logs finden Sie unterSenden von Lambda-Funktionsprotokollen an CloudWatch Logs.

Zur Ausgabe von Protokollen aus Ihrem Funktionscode können Sie das integrierte `logging`-Modul verwenden. Für detailliertere Einträge können Sie jede Protokollierungsbibliothek verwenden, die in `stdout` oder `stderr` schreibt.

Ausdrucken in das Protokoll

Um eine grundlegende Ausgabe an die Protokolle zu senden, können Sie in Ihrer Funktion eine `print`-Methode verwenden. Im folgenden Beispiel werden die Werte der CloudWatch Loggruppe und des Streams Logs sowie das Event-Objekt protokolliert.

Beachten Sie, dass Lambda Protokollausgaben nur im Klartextformat an Logs senden kann, wenn Ihre Funktion CloudWatch Logs mithilfe von `print` Python-Anweisungen ausgibt. Um Protokolle in strukturiertem JSON zu erfassen, müssen Sie eine unterstützte Protokollierungsbibliothek verwenden. Weitere Informationen finden Sie unter the section called “Verwenden von Lambda-Optionen für die erweiterte Protokollierung mit 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 Protokollausgabe

```
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
```

Die Python-Laufzeit protokolliert die Zeilen `START`, `END` und `REPORT` für jeden Aufruf. Die `REPORT`-Zeile enthält die folgenden Daten:

Datenfelder für REPORT-Zeilen

- RequestId— Die eindeutige Anforderungs-ID für den Aufruf.
- Dauer – Die Zeit, die die Handler-Methode Ihrer Funktion mit der Verarbeitung des Ereignisses verbracht hat.
- Fakturierte Dauer – Die für den Aufruf fakturierte Zeit.
- Speichergröße – Die der Funktion zugewiesene Speichermenge.
- Max. verwendeter Speicher – Die Speichermenge, die von der Funktion verwendet wird. Wenn Aufrufe eine Ausführungsumgebung gemeinsam nutzen, meldet Lambda den maximalen Speicherverbrauch für alle Aufrufe. Dieses Verhalten kann zu einem höheren als erwarteten gemeldeten Wert führen.
- Initialisierungsdauer – Für die erste Anfrage die Zeit, die zur Laufzeit zum Laden der Funktion und Ausführen von Code außerhalb der Handler-Methode benötigt wurde.
- XRAY TraceId — Für verfolgte Anfragen die AWS X-Ray Trace-ID.
- SegmentId— Für verfolgte Anfragen die X-Ray-Segment-ID.
- Stichprobe – Bei verfolgten Anforderungen das Stichprobenergebnis.

Verwendung einer Protokollierungsbibliothek

Für detailliertere Protokolle verwenden Sie das Modul zur Protokollierung in der Standardbibliothek oder eine beliebige Protokollierungsbibliothek eines Drittanbieters, die in `stdout` oder `stderr` schreibt.

Für unterstützte Python-Laufzeiten können Sie wählen, ob mit dem `logging`-Standardmodul erstellte Protokolle im Klartext oder in JSON erfasst werden. Weitere Informationen hierzu finden Sie unter the section called “Verwenden von Lambda-Optionen für die erweiterte Protokollierung mit Python”.

Derzeit ist das Standard-Protokollformat für alle Python-Laufzeiten das Klartextformat. Das folgende Beispiel zeigt, wie Protokollausgaben, die mit dem `logging` Standardmodul erstellt wurden, im Klartext in CloudWatch Logs erfasst werden.

```
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)
```

Die Ausgabe aus `logger` umfasst die Protokollebene, den Zeitstempel und die Anforderungs-ID.

```
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**
>
> Wenn das Protokollformat Ihrer Funktion auf Klartext gesetzt ist, lautet die Standardeinstellung auf Protokollebene für Python-Laufzeiten WARN. Das bedeutet, dass Lambda nur Protokollausgaben der Stufe WARN und niedriger an CloudWatch Logs sendet. Um die Standardprotokollebene zu ändern, verwenden Sie die Python-Methode `logging setLevel()`, wie in diesem Beispielcode gezeigt. Wenn Sie das Protokollformat Ihrer Funktion auf JSON setzen, empfehlen wir, die Protokollebene der Funktion mithilfe der erweiterten Lambda Advanced Logging Controls zu konfigurieren und nicht die Protokollebene im Code festzulegen. Weitere Informationen hierzu finden Sie unter the section called “Verwenden der Filterung auf Protokollebene mit Python”.

Verwenden von Lambda-Optionen für die erweiterte Protokollierung mit Python

Um Ihnen mehr Kontrolle darüber zu geben, wie die Protokolle Ihrer Funktionen erfasst, verarbeitet und verwendet werden, können Sie die folgenden Protokollierungsoptionen für unterstützte Lambda-Python-Laufzeiten konfigurieren:

- Protokollformat – Wählen Sie zwischen Klartext und einem strukturierten JSON-Format für die Protokolle Ihrer Funktion aus.
- Protokollebene — für Logs im JSON-Format wählen Sie die Detailebene der Logs, die Lambda an Amazon sendet CloudWatch, wie ERROR, DEBUG oder INFO
- Protokollgruppe — wählen Sie die CloudWatch Protokollgruppe aus, an die Ihre Funktion Protokolle sendet

Weitere Informationen zu diesen Protokollierungsoptionen und Anweisungen zur Konfiguration Ihrer Funktion für deren Verwendung finden Sie unter the section called “Konfigurieren erweiterter Protokollierungsoptionen für Lambda-Funktionen”.

Weitere Informationen zur Verwendung der Optionen für das Protokollformat und die Protokollebene mit Ihren Python-Lambda-Funktionen finden Sie in den folgenden Abschnitten.

Verwenden strukturierter JSON-Protokolle mit Python

Wenn Sie JSON für das Protokollformat Ihrer Funktion auswählen, sendet Lambda die von der Python-Standard-Logging-Bibliothek ausgegebenen Logs CloudWatch als strukturiertes JSON an. Jedes JSON-Protokollobjekt enthält mindestens vier Schlüssel-Wert-Paare mit den folgenden Schlüsseln:

- `"timestamp"` – die Uhrzeit, zu der die Protokollmeldung generiert wurde
- `"level"` – die der Meldung zugewiesene Protokollebene
- `"message"` – der Inhalt der Protokollmeldung
- `"requestId"` – die eindeutige Anforderungs-ID für den Funktionsaufruf

Die `logging`-Bibliothek von Python kann auch zusätzliche Schlüssel-Wert-Paare wie `"logger"` zu diesem JSON-Objekt hinzufügen.

Die Beispiele in den folgenden Abschnitten zeigen, wie mit der `logging` Python-Bibliothek generierte Protokollausgaben in CloudWatch Logs erfasst werden, wenn Sie das Protokollformat Ihrer Funktion als JSON konfigurieren.

Beachten Sie Folgendes: Wenn Sie die `print`-Methode verwenden, um grundlegende Protokollausgaben zu erzeugen, wie unter the section called “Ausdrucken in das Protokoll” beschrieben, erfasst Lambda diese Ausgaben als Klartext, auch wenn Sie das Protokollierungsformat Ihrer Funktion als JSON konfigurieren.

Standard-JSON-Protokollausgaben mithilfe der Python-Protokollierungsbibliothek

Der folgende Beispielcodeausschnitt und die Protokollausgabe zeigen, wie mit der `logging` Python-Bibliothek generierte Standardprotokollausgaben in CloudWatch Logs erfasst werden, wenn das Protokollformat Ihrer Funktion auf JSON gesetzt ist.

Example Python-Protokollierungscode

```
import logging
logger = logging.getLogger()

def lambda_handler(event, context):
    logger.info("Inside the handler function")
```

Example JSON-Protokolldatensatz

```
{
    "timestamp":"2025-10-27T19:17:45.586Z",
    "level":"INFO",
    "message":"Inside the handler function",
    "logger": "root",
    "requestId":"79b4f56e-95b1-4643-9700-2807f4e68189"
}
```

Protokollieren zusätzlicher Parameter in JSON

Wenn das Protokollformat Ihrer Funktion auf JSON eingestellt ist, können Sie auch zusätzliche Parameter mit der standardmäßigen Python `logging`-Bibliothek protokollieren, indem Sie das Schlüsselwort `extra` verwenden, um ein Python-Wörterbuch an die Protokollausgabe zu übergeben.

Example Python-Protokollierungscode

```
import logging

def lambda_handler(event, context):
    logging.info(
        "extra parameters example",
        extra={"a":"b", "b": [3]},
    )
```

Example JSON-Protokolldatensatz

```
{
  "timestamp": "2025-11-02T15:26:28Z",
  "level": "INFO",
  "message": "extra parameters example",
  "logger": "root",
  "requestId": "3dbd5759-65f6-45f8-8d7d-5bdc79a3bd01",
  "a": "b",
  "b": [
    3
  ]
}
```

Protokollierungsausnahmen in JSON

Der folgende Codeausschnitt zeigt, wie Python-Ausnahmen in der Protokollausgabe Ihrer Funktion erfasst werden, wenn Sie das Protokollformat als JSON konfigurieren. Beachten Sie, dass den mit `logging.exception` generierten Protokollausgaben die Protokollebene ERROR zugewiesen wird.

Example Python-Protokollierungscode

```
import logging

def lambda_handler(event, context):
    try:
        raise Exception("exception")
    except:
        logging.exception("msg")
```

Example JSON-Protokolldatensatz

```
{
  "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"
}
```

Strukturierte JSON-Protokolle mit anderen Protokollierungstools

Wenn Ihr Code bereits eine andere Logging-Bibliothek wie Powertools for verwendet AWS Lambda, um strukturierte JSON-Logs zu erstellen, müssen Sie keine Änderungen vornehmen. AWS Lambda codiert keine Logs doppelt, die bereits JSON-kodiert sind. Selbst wenn Sie Ihre Funktion so konfigurieren, dass sie das JSON-Protokollformat verwendet, erscheinen Ihre Logging-Ausgaben CloudWatch in der von Ihnen definierten JSON-Struktur.

Das folgende Beispiel zeigt, wie Protokollausgaben, die mit dem AWS Lambda Paket Powertools for generiert wurden, in CloudWatch Logs erfasst werden. Das Format dieser Protokollausgabe ist dasselbe, unabhängig davon, ob die Protokollierungskonfiguration Ihrer Funktion auf JSON oder TEXT festgelegt ist. Weitere Informationen zur Verwendung von Powertools für finden Sie unter AWS Lambda und the section called “Verwendung von Powertools für AWS Lambda (Python) und AWS SAM für strukturiertes Logging” the section called “Verwendung von Powertools für AWS Lambda (Python) und AWS CDK für strukturiertes Logging”

Example Codeausschnitt für die Python-Protokollierung (mit Powertools für) AWS Lambda

```
from aws_lambda_powertools import Logger

logger = Logger()

def lambda_handler(event, context):
    logger.info("Inside the handler function")
```

Example JSON-Protokolldatensatz (unter Verwendung von Powertools für) 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"
}
```

Verwenden der Filterung auf Protokollebene mit Python

Durch die Konfiguration der Filterung auf Protokollebene können Sie festlegen, dass nur Protokolle mit einer bestimmten Protokollierungsebene oder niedriger an Logs gesendet werden. CloudWatch Informationen zur Konfiguration der Filterung auf Protokollebene für Ihre Funktion finden Sie unter the section called “Filterung auf Protokollebene”.

AWS Lambda Um Ihre Anwendungsprotokolle nach ihrer Protokollebene zu filtern, muss Ihre Funktion Protokolle im JSON-Format verwenden. Sie können dies auf zwei Arten erreichen:

- Erstellen Sie Protokollausgaben mit der `logging`-Python-Standardbibliothek und konfigurieren Sie Ihre Funktion so, dass sie die JSON-Protokollformatierung verwendet. AWS Lambda filtert dann Ihre Protokollausgaben mithilfe des Schlüssel-Wert-Paars „level“ im JSON-Objekt, wie

unter the section called “Verwenden strukturierter JSON-Protokolle mit Python” beschrieben. Informationen zur Konfiguration des Protokollformats Ihrer Funktion finden Sie unter the section called “Konfigurieren erweiterter Protokollierungsoptionen für Lambda-Funktionen”.

- Verwenden Sie eine andere Protokollierungsbibliothek oder Methode, um strukturierte JSON-Protokolle in Ihrem Code zu erstellen, die ein „level“-Schlüssel-Wert-Paar enthalten, das die Ebene der Protokollausgabe definiert. Sie können Powertools beispielsweise verwenden, AWS Lambda um strukturierte JSON-Protokollausgaben aus Ihrem Code zu generieren.

  Sie können auch eine „print“-Anweisung verwenden, um ein JSON-Objekt auszugeben, das eine Protokollebenenkennung enthält. Die folgende Print-Anweisung erzeugt eine Ausgabe im JSON-Format, bei der die Protokollebene auf INFO gesetzt ist. AWS Lambda sendet das JSON-Objekt an CloudWatch Logs, wenn die Protokollierungsebene Ihrer Funktion auf INFO, DEBUG oder TRACE gesetzt ist.

```
print('{"msg":"My log message", "level":"info"}')
```

Damit Lambda die Protokolle Ihrer Funktion filtern kann, müssen Sie auch ein `"timestamp"`-Schlüssel-Wert-Paar in Ihre JSON-Protokollausgabe aufnehmen. Die Uhrzeit muss im gültigen RFC 3339-Zeitstempelformat angegeben werden. Wenn Sie keinen gültigen Zeitstempel angeben, weist Lambda dem Protokoll die Stufe INFO zu und fügt einen Zeitstempel für Sie hinzu.

Anzeigen von Protokollen in der Lambda-Konsole

Sie können die Lambda-Konsole verwenden, um die Protokollausgabe nach dem Aufrufen einer Lambda-Funktion anzuzeigen.

Wenn Ihr Code über den eingebetteten Code-Editor getestet werden kann, finden Sie Protokolle in den Ausführungsergebnissen. Wenn Sie das Feature Konsolentest verwenden, um eine Funktion aufzurufen, finden Sie die Protokollausgabe im Abschnitt Details.

Logs in CloudWatch der Konsole anzeigen

Sie können die CloudWatch Amazon-Konsole verwenden, um Protokolle für alle Lambda-Funktionsaufrufe anzuzeigen.

Um Protokolle auf der Konsole anzuzeigen CloudWatch

1. Öffnen Sie die Seite Protokollgruppen auf der CloudWatch Konsole.

2. Wählen Sie die Protokollgruppe für Ihre Funktion (***your-function-name***/aws/lambda/).
3. Wählen Sie eine Protokollstream aus.

Jeder Protokoll-Stream entspricht einer Instance Ihrer Funktion. Ein Protokollstream wird angezeigt, wenn Sie Ihre Lambda-Funktion aktualisieren und wenn zusätzliche Instanzen für gleichzeitige Aufrufe erstellt werden. Um Protokolle für einen bestimmten Aufruf zu finden, empfehlen wir, Ihre Funktion mit zu instrumentieren. AWS X-Ray X-Ray erfasst Details zu der Anforderung und dem Protokollstream in der Trace.

Logs anzeigen mit AWS CLI

Das AWS CLI ist ein Open-Source-Tool, mit dem Sie mithilfe von Befehlen in Ihrer Befehlszeilen-Shell mit AWS Diensten interagieren können. Zur Durchführung der Schritte in diesem Abschnitt benötigen Sie AWS CLI Version 2.

Sie können die AWS CLI verwenden, um Protokolle für einen Aufruf mit der `--log-type`-Befehlsoption abzurufen. Die Antwort enthält das Feld `LogResult`, das bis zu 4 KB base64-verschlüsselte Protokolle aus dem Aufruf enthält.

Example eine Log-ID abrufen

Das folgende Beispiel zeigt, wie eine Protokoll-ID aus dem `LogResult`-Feld für eine Funktion namens `my-function` abgerufen wird.

```
aws lambda invoke --function-name my-function out --log-type Tail
```

Die Ausgabe sollte folgendermaßen aussehen:

```
{
    "StatusCode": 200,
    "LogResult":
 "U1RBUlQgUmVxdWVzdElkOiA4N2QwNDRiOC1mMTU0LTExZTgtOGNkYS0yOTc0YzVlNGZiMjEgVmVyc2lvb...",
    "ExecutedVersion": "$LATEST"
}
```

Example entschlüsseln der Protokolle

Verwenden Sie in derselben Eingabeaufforderung das `base64`-Dienstprogramm, um die Protokolle zu entschlüsseln. Das folgende Beispiel zeigt, wie Base64-codierte Logs für abgerufen werde `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
```

Die cli-binary-format Option ist erforderlich, wenn Sie AWS CLI Version 2 verwenden. Um dies zur Standardeinstellung zu machen, führen Sie `aws configure set cli-binary-format raw-in-base64-out` aus. Weitere Informationen finden Sie unter Von AWS CLI unterstützte globale Befehlszeilenoptionen im AWS Command Line Interface -Benutzerhandbuch für Version 2.

Die Ausgabe sollte folgendermaßen aussehen:

```
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
```

Das `base64`-Dienstprogramm ist unter Linux, macOS und Ubuntu auf Windowsverfügbar. macOS-Benutzer müssen möglicherweise `base64 -D` verwenden.

Example get-logs.sh-Skript

Verwenden Sie in derselben Eingabeaufforderung das folgende Skript, um die letzten fünf Protokollereignisse herunterzuladen. Das Skript verwendet `sed` zum Entfernen von Anführungszeichen aus der Ausgabedatei und wechselt 15 Sekunden lang in den Ruhezustand, um Zeit einzuräumen, damit Protokolle verfügbar werden können. Die Ausgabe enthält die Antwort von Lambda und die `get-log-events`Ausgabe des Befehls.

Kopieren Sie den Inhalt des folgenden Codebeispiels und speichern Sie es in Ihrem Lambda-Projektverzeichnis unter `get-logs.sh`.

Die cli-binary-format Option ist erforderlich, wenn Sie AWS CLI Version 2 verwenden. Um dies zur Standardeinstellung zu machen, führen Sie `aws configure set cli-binary-format raw-in-base64-out` aus. Weitere Informationen finden Sie unter Von AWS CLI unterstützte globale Befehlszeilenoptionen im AWS Command Line Interface -Benutzerhandbuch für 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 und Linux (nur diese Systeme)

In derselben Eingabeaufforderung müssen macOS- und Linux-Benutzer möglicherweise den folgenden Befehl ausführen, um sicherzustellen, dass das Skript ausführbar ist.

```
chmod -R 755 get-logs.sh
```

Example die letzten fünf Protokollereignisse abrufen

Führen Sie an derselben Eingabeaufforderung das folgende Skript aus, um die letzten fünf Protokollereignisse abzurufen.

```
./get-logs.sh
```

Die Ausgabe sollte folgendermaßen aussehen:

```
{
    "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"
}
```


Löschen von Protokollen

Wenn Sie eine Funktion löschen, werden Protokollgruppen nicht automatisch gelöscht. Um das unbegrenzte Speichern von Protokollen zu vermeiden, löschen Sie die Protokollgruppe oder konfigurieren Sie eine Aufbewahrungszeitraum nach dem Protokolle automatisch gelöscht werden.

Andere Tools und Bibliotheken für die Protokollierung verwenden

Powertools for AWS Lambda (Python) ist ein Entwickler-Toolkit zur Implementierung serverloser Best Practices und zur Steigerung der Entwicklergeschwindigkeit. Das Logger-Serviceprogramm bietet einen für Lambda optimierten Logger, der zusätzliche Informationen über den Funktionskontext all Ihrer Funktionen enthält, wobei die Ausgabe als JSON strukturiert ist. Mit diesem Serviceprogramm können Sie Folgendes tun:

- Erfassung von Schlüsselfeldern aus dem Lambda-Kontext, Kaltstart und Strukturen der Protokollierungsausgabe als JSON
- Protokollieren Sie Ereignisse von Lambda-Aufrufen, wenn Sie dazu aufgefordert werden (standardmäßig deaktiviert)

- Alle Protokolle nur für einen bestimmten Prozentsatz der Aufrufe über Protokollstichproben drucken (standardmäßig deaktiviert)
- Fügen Sie dem strukturierten Protokoll zu einem beliebigen Zeitpunkt zusätzliche Schlüssel hinzu
- Verwenden Sie einen benutzerdefinierten Protokollformatierer (Bring Your Own Formatter), um Protokolle in einer Struktur auszugeben, die mit dem Logging RFC Ihres Unternehmens kompatibel ist

Verwendung von Powertools für AWS Lambda (Python) und AWS SAM für strukturiertes Logging

Führen Sie die nachstehenden Schritte aus, um eine Hello-World-Python-Beispielanwendung mit integrierten -Powertools for Python-Modulen unter Verwendung von AWS SAM bereitzustellen. Diese Anwendung implementiert ein grundlegendes API-Backend und verwendet Powertools zum Ausgeben von Protokollen, Metriken und Traces. Es besteht aus einem Amazon-API-Gateway-Endpunkt und einer Lambda-Funktion. Wenn Sie eine GET-Anfrage an den API-Gateway-Endpunkt senden, ruft die Lambda-Funktion Logs und Metriken auf, sendet sie im Embedded Metric Format an CloudWatch und sendet Traces an. AWS X-Ray Die Funktion gibt eine `hello world`-Nachricht zurück.

Voraussetzungen

Zur Durchführung der Schritte in diesem Abschnitt benötigen Sie Folgendes:

- Python 3.9
- AWS CLI Version 2
- AWS SAM CLI Version 1.75 oder höher. Wenn Sie eine ältere Version der AWS SAM CLI haben, finden Sie weitere Informationen unter Upgrade der AWS SAM CLI.

Stellen Sie eine AWS SAM Beispielanwendung bereit

1. Initialisieren Sie die Anwendung mit der Hello World Python-Vorlage.

```
sam init --app-template hello-world-powertools-python --name sam-app --package-type
 Zip --runtime python3.9 --no-tracing
```

2. Entwickeln Sie die App.

```
cd sam-app && sam build
```

3. Stellen Sie die Anwendung bereit.

```
sam deploy --guided
```

4. Folgen Sie den Anweisungen auf dem Bildschirm. Um die im interaktiven Erlebnis bereitgestellten Standardoptionen zu akzeptieren, drücken Sie `Enter`.

> **Note**
>
> Für ist HelloWorldFunction möglicherweise keine Autorisierung definiert. Ist das in Ordnung? , stellen Sie sicher, dass Sie eintreteny.

5. Rufen Sie die URL der bereitgestellten Anwendung ab:

```
aws cloudformation describe-stacks --stack-name sam-app --query
 'Stacks[0].Outputs[?OutputKey==`HelloWorldApi`].OutputValue' --output text
```

6. Rufen Sie den API-Endpunkt auf:

```
curl GET <URL_FROM_PREVIOUS_STEP>
```

Wenn der Link erfolgreich ausgeführt wurde, sehen Sie die folgende Antwort:

```
{"message":"hello world"}
```

7. Führen Sie sam logs aus, um die Protokolle für die Funktion abzurufen. Weitere Informationen finden Sie unter Arbeiten mit Protokollen im AWS Serverless Application Model - Entwicklerhandbuch.

```
sam logs --stack-name sam-app
```

Das Ergebnis sieht folgendermaßen aus:

```
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. Dies ist ein öffentlicher API-Endpunkt, der über das Internet zugänglich ist. Es wird empfohlen, dass Sie den Endpunkt nach dem Testen löschen.

```
sam delete
```

Verwalten der Protokollaufbewahrung

Wenn Sie eine Funktion löschen, werden Protokollgruppen nicht automatisch gelöscht. Um zu vermeiden, dass Protokolle auf unbestimmte Zeit gespeichert werden, löschen Sie die Protokollgruppe oder konfigurieren Sie einen Aufbewahrungszeitraum, nach dessen Ablauf die Protokolle CloudWatch automatisch gelöscht werden. Um die Aufbewahrung von Protokollen einzurichten, fügen Sie Ihrer AWS SAM Vorlage Folgendes hinzu:

```
Resources:
  HelloWorldFunction:
    Type: AWS::Serverless::Function
    Properties:
      # Omitting other properties

  LogGroup:
    Type: AWS::Logs::LogGroup
    Properties:
      LogGroupName: !Sub "/aws/lambda/${HelloWorldFunction}"
      RetentionInDays: 7
```

Verwendung von Powertools für AWS Lambda (Python) und AWS CDK für strukturiertes Logging

Gehen Sie wie folgt vor, um eine Hello World Python-Beispielanwendung mit integrierten Powertools for AWS Lambda (Python) -Modulen herunterzuladen, zu erstellen und bereitzustellen. Verwenden Sie dazu den AWS CDK. Diese Anwendung implementiert ein grundlegendes API-Backend und verwendet Powertools zum Ausgeben von Protokollen, Metriken und Traces. Es besteht aus einem Amazon-API-Gateway-Endpunkt und einer Lambda-Funktion. Wenn Sie eine GET-Anfrage an den API-Gateway-Endpunkt senden, ruft die Lambda-Funktion Logs und Metriken auf, sendet sie im Embedded Metric Format an CloudWatch und sendet Traces an. AWS X-Ray Die Funktion gibt eine Hello-World-Nachricht zurück.

Voraussetzungen

Zur Durchführung der Schritte in diesem Abschnitt benötigen Sie Folgendes:

- Python 3.9
- AWS CLI Version 2
- AWS CDK Ausführung 2

- AWS SAM CLI Version 1.75 oder höher. Wenn Sie eine ältere Version der AWS SAM CLI haben, finden Sie weitere Informationen unter Upgrade der AWS SAM CLI.

Stellen Sie eine AWS CDK Beispielanwendung bereit

1. Erstellen Sie ein Projektverzeichnis für Ihre neue Anwendung.

```
mkdir hello-world
cd hello-world
```

2. Initialisieren Sie die App.

```
cdk init app --language python
```

3. Installieren Sie die Python-Abhängigkeiten.

```
pip install -r requirements.txt
```

4. Erstellen Sie ein Verzeichnis lambda_function unter dem Stammordner.

```
mkdir lambda_function
cd lambda_function
```

5. Erstellen Sie eine Datei namens app.py und fügen Sie den folgenden Code zur Datei hinzu. Dies ist der Code für die Lambda-Funktion.

```
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. Öffnen Sie das Verzeichnis hello_world. Sie sollten eine Datei mit dem Namen hello_world_stack.py sehen.

```
cd ..
cd hello_world
```

7. Öffnen Sie hello_world_stack.py und fügen Sie den folgenden Code in die Datei ein. Dies enthält den Lambda-Konstruktor, der die Lambda-Funktion erstellt, Umgebungsvariablen für Powertools konfiguriert und die Protokollspeicherung auf eine Woche festlegt, und den ApiGatewayv1-Konstruktor, der die REST-API erstellt.

```
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. Stellen Sie die Anwendung bereit.

```
cd ..
cdk deploy
```

9. Rufen Sie die URL der bereitgestellten Anwendung ab:

```
aws cloudformation describe-stacks --stack-name HelloWorldStack --query
 'Stacks[0].Outputs[?OutputKey==`apiUrl`].OutputValue' --output text
```

10. Rufen Sie den API-Endpunkt auf:

```
curl GET <URL_FROM_PREVIOUS_STEP>
```

Wenn der Link erfolgreich ausgeführt wurde, sehen Sie die folgende Antwort:

```
{"message":"hello world"}
```

11. Führen Sie sam logs aus, um die Protokolle für die Funktion abzurufen. Weitere Informationen finden Sie unter Arbeiten mit Protokollen im AWS Serverless Application Model - Entwicklerhandbuch.

```
sam logs --stack-name HelloWorldStack
```

Das Ergebnis sieht folgendermaßen aus:

```
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. Dies ist ein öffentlicher API-Endpunkt, der über das Internet zugänglich ist. Es wird empfohlen, dass Sie den Endpunkt nach dem Testen löschen.

```
cdk destroy
```

AWS Lambda-Funktionstests in Python

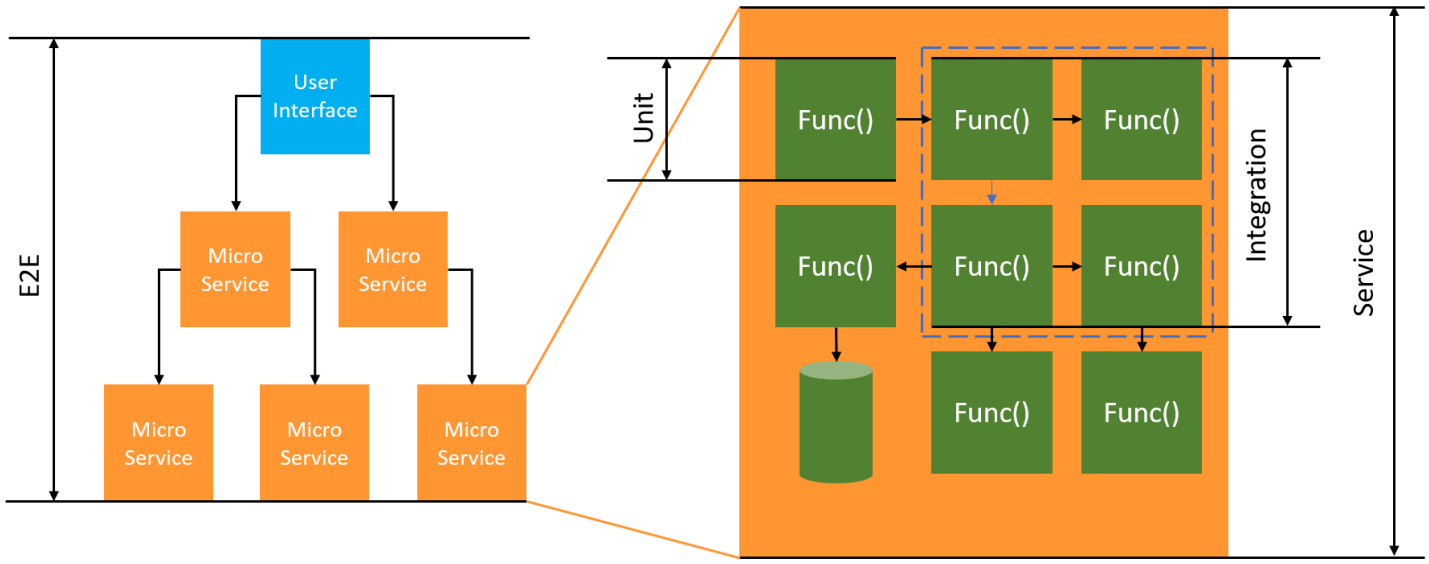
> **Note**
>
> Im Kapitel Testen von Funktionen finden Sie eine vollständige Einführung in Techniken und bewährte Methoden für das Testen von Serverless-Lösungen.

Beim Testen der Serverless-Funktionen werden herkömmliche Testtypen und -techniken verwendet. Erwägen Sie jedoch auch das Testen der Serverless-Anwendungen als Ganzes. Cloud-basierte Tests bieten das genaueste Maß für die Qualität sowohl Ihrer Funktionen als auch Ihrer Serverless-Anwendungen.

Eine Serverless-Anwendungsarchitektur umfasst verwaltete Services, die über API-Aufrufe wichtige Anwendungsfunktionen bereitstellen. Aus diesem Grund muss Ihr Entwicklungszyklus automatisierte Tests beinhalten, die bei der Interaktion Ihrer Funktionen und Services die Funktionalität überprüfen.

Wenn Sie keine cloud-basierten Tests erstellen, können aufgrund von Unterschieden zwischen Ihrer lokalen Umgebung und der bereitgestellten Umgebung Probleme auftreten. Ihr kontinuierlicher Integrationsprozess muss Tests anhand einer Reihe von Ressourcen durchführen, die in der Cloud bereitgestellt werden, bevor Ihr Code in die nächste Bereitstellungsumgebung wie QA, Staging oder Produktion übertragen wird.

Lesen Sie diesen kurzen Leitfaden weiter, um weitere Informationen zu Teststrategien für Serverless-Anwendungen zu erhalten, oder besuchen Sie das Serverless Test Samples Repository, um praktische Beispiele zu finden, die sich speziell auf die gewählte Sprache und Laufzeit beziehen.



Für Serverless-Tests schreiben Sie weiterhin Einheiten-, Integrations- und End-to-End-Tests.

- Einheitentests — Tests, die an einem isolierten Codeblock ausgeführt werden. Zum Beispiel die Überprüfung der Geschäftslogik zur Berechnung der Bereitstellungskosten für einen bestimmten Artikel und Bestimmungsort.
- Integrationstests — Tests, an denen zwei oder mehr Komponenten oder Dienste beteiligt sind, die interagieren, in der Regel in einer Cloud-Umgebung. Bei der Überprüfung einer Funktion werden beispielsweise Ereignisse aus einer Warteschlange verarbeitet.
- End-to-End-Tests — Tests, die das Verhalten einer gesamten Anwendung überprüfen. Stellen Sie beispielsweise sicher, dass die Infrastruktur korrekt eingerichtet ist und die Ereignisse zwischen den Services wie erwartet ablaufen, um die Bestellungen der Kunden aufzuzeichnen.

Testen Ihrer Serverless-Anwendungen

In der Regel verwenden Sie eine Mischung aus verschiedenen Ansätzen, um Ihren Serverless-Anwendungscode zu testen, einschließlich Tests in der Cloud, Tests mit Mock-Code und gelegentlich Tests mit Emulatoren.

Testen in der Cloud

Das Testen in der Cloud ist für alle Testphasen von Nutzen, einschließlich Einheitentests, Integrationstests und End-to-End-Tests. Sie führen Tests für Code durch, der in der Cloud bereitgestellt wird und mit cloud-basierten Services interagiert. Dieser Ansatz bietet das genaueste Maß für die Qualität Ihres Codes.

Eine bequeme Möglichkeit, Ihre Lambda-Funktion in der Cloud zu debuggen, ist die Verwendung der Konsole mit einem Testereignis. Ein Testereignis ist eine JSON-Eingabe für Ihre Funktion. Wenn Ihre Funktion keine Eingabe erfordert, kann das Ereignis ein leeres JSON-Dokument (`{}`) sein. Die Konsole bietet Beispielereignisse für eine Vielzahl von Service-Integrationen. Nachdem Sie ein Ereignis in der Konsole erstellt haben, können Sie es mit Ihrem Team teilen, um das Testen einfacher und einheitlicher zu gestalten.

> **Note**
>
> Das Testen einer Funktion in der Konsole ist ein schneller Einstieg, aber die Automatisierung Ihrer Testzyklen gewährleistet die Anwendungsqualität und die Entwicklungsgeschwindigkeit.

Test-Tools

Es gibt Tools und Techniken, um die Feedback-Schleifen bei der Entwicklung zu beschleunigen. AWSSAM Accelerate und AWSCDK Watch Mode reduzieren beispielsweise beide die Zeit, die für die Aktualisierung von Cloud-Umgebungen benötigt wird.

Moto ist eine Python-Bibliothek für das Mocking von AWS-Services und -Ressourcen, sodass Sie Ihre Funktionen mit wenigen oder gar keinen Änderungen testen können, indem Sie Dekoratoren zum Abfangen und Simulieren von Antworten verwenden.

Das Validierungsfeature von Powertools für AWS Lambda (Python) bietet Dekoratoren, mit denen Sie Eingabeereignisse und Ausgabeantworten Ihrer Python-Funktionen validieren können.

Weitere Informationen finden Sie im Blogbeitrag Unit Testing Lambda with Python and Mock AWS Services.

Informationen zur Verringerung der Latenzzeit bei Iterationen der Cloud-Bereitstellung finden Sie unter AWS Serverless Application Model (AWS SAM) Accelerate, AWS Cloud Development Kit (Watch-Modus AWS CDK). Diese Tools überwachen Ihre Infrastruktur und Ihren Code auf Änderungen. Sie reagieren auf diese Änderungen, indem sie automatisch inkrementelle Updates erstellen und in Ihrer Cloud-Umgebung bereitstellen.

Beispiele, in denen diese Tools verwendet werden, sind im Code-Repository mit Python-Testbeispielen verfügbar.

Instrumentieren von Python-Code in AWS Lambda

Lambda ist mit AWS X-Ray integriert, um Ihnen zu helfen, Lambda-Anwendungen zu verfolgen, zu debuggen und zu optimieren. Sie können mit X-Ray eine Anforderung verfolgen, während sie Ressourcen in Ihrer Anwendung durchläuft, die Lambda-Funktionen und andere AWS-Services enthalten können.

Um Protokollierungsdaten an X-Ray zu senden, können Sie eine von drei SDK-Bibliotheken verwenden:

- AWS Distro for OpenTelemetry (ADOT) – Eine sichere, produktionsbereite, AWS-unterstützte Verteilung des OpenElemetrie (OTel)-SDK.
- AWS X-Ray-SDK für Python – Ein SDK zum Generieren und Senden von Nachverfolgungsdaten an X-Ray.
- Powertools for AWS Lambda (Python) – Ein Entwickler-Toolkit zur Implementierung bewährter Methoden für Serverless und zur Steigerung der Entwicklergeschwindigkeit.

Jedes der SDKs bietet Möglichkeiten, Ihre Telemetriedaten an den X-Ray Service zu senden. Sie können dann mit X-Ray die Leistungsmetriken Ihrer Anwendung anzeigen, filtern und erhalten, um Probleme und Möglichkeiten zur Optimierung zu identifizieren.

> **Important**
>
> X-Ray und Powertools für AWS Lambda-SDKs sind Teil einer eng integrierten Instrumentierungslösung von AWS. Die ADOT Lambda Layers sind Teil eines branchenweiten Standards für die Verfolgung von Instrumenten, die im Allgemeinen mehr Daten erfassen, aber möglicherweise nicht für alle Anwendungsfälle geeignet sind. Sie können mit beiden Lösungen eine durchgehende Ablaufverfolgung in X-Ray implementieren. Weitere Informationen zur Auswahl zwischen ihnen finden Sie unter Auswählen zwischen der AWS-Distro für Open Telemetry und X-Ray-SDKs.

Abschnitte

- Verwenden von Powertools für AWS Lambda (Python) und AWS SAM für die Nachverfolgung
- Verwenden von Powertools für AWS Lambda (Python) und AWS CDK für die Nachverfolgung
- Verwenden von ADOT zum Instrumentieren Ihrer Python-Funktionen

- Verwenden des X-Ray-SDK zum Instrumentieren Ihrer Python-Funktionen
- Aktivieren der Nachverfolgung mit der Lambda-Konsole
- Aktivieren der Nachverfolgung mit der Lambda-API
- Aktivieren von Nachverfolgungen mit CloudFormation
- Interpretieren einer X-Ray-Nachverfolgung
- Laufzeitabhängigkeiten in einer Ebene speichern (X-Ray-SDK)

Verwenden von Powertools für AWS Lambda (Python) und AWS SAM für die Nachverfolgung

Führen Sie die nachstehenden Schritte aus, um eine Hello-World-Beispielanwendung für Python mit integrierten Modulen von Powertools for AWS Lambda (Python) unter Verwendung von AWS SAM bereitzustellen. Diese Anwendung implementiert ein grundlegendes API-Backend und verwendet Powertools zum Ausgeben von Protokollen, Metriken und Traces. Es besteht aus einem Amazon-API-Gateway-Endpunkt und einer Lambda-Funktion. Wenn Sie eine GET-Anforderung an den API-Gateway-Endpunkt senden, startet die Lambda-Funktion, sendet Protokolle und Metriken im Embedded Metric Format an CloudWatch und sendet Traces an AWS X-Ray. Die Funktion gibt eine Hello-World-Nachricht zurück.

Voraussetzungen

Zur Durchführung der Schritte in diesem Abschnitt benötigen Sie Folgendes:

- Python 3.11
- AWS CLI Version 2
- AWS SAM CLI-Version 1.75 oder höher. Wenn Sie eine ältere Version der AWS SAM-CLI haben, finden Sie weitere Informationen unter Aktualisieren der AWS SAM-CLI.

Bereitstellen einer AWS SAM-Beispielanwendung

1. Initialisieren Sie die Anwendung mit der Hello World Python-Vorlage.

```
sam init --app-template hello-world-powertools-python --name sam-app --package-type
 Zip --runtime python3.11 --no-tracing
```

2. Entwickeln Sie die App.

```
cd sam-app && sam build
```

3. Stellen Sie die Anwendung bereit.

```
sam deploy --guided
```

4. Folgen Sie den Anweisungen auf dem Bildschirm. Um die im interaktiven Erlebnis bereitgestellten Standardoptionen zu akzeptieren, drücken Sie `Enter`.

> **Note**
>
> Für HelloWorldFunction ist möglicherweise keine Autorisierung definiert. Ist das in Ordnung? , stellen Sie sicher, dass Sie y eingeben.

5. Rufen Sie die URL der bereitgestellten Anwendung ab:

```
aws cloudformation describe-stacks --stack-name sam-app --query
 'Stacks[0].Outputs[?OutputKey==`HelloWorldApi`].OutputValue' --output text
```

6. Rufen Sie den API-Endpunkt auf:

```
curl -X GET <URL_FROM_PREVIOUS_STEP>
```

Wenn der Link erfolgreich ausgeführt wurde, sehen Sie die folgende Antwort:

```
{"message":"hello world"}
```

7. Führen Sie sam traces aus, um die Traces für die Funktion zu erhalten.

```
sam traces
```

Das Nachverfolgungsergebnis sieht folgendermaßen aus:

```
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. Dies ist ein öffentlicher API-Endpunkt, der über das Internet zugänglich ist. Es wird empfohlen, dass Sie den Endpunkt nach dem Testen löschen.

```
sam delete
```

X-Ray verfolgt nicht alle Anfragen an Ihre Anwendung nach. X-Ray wendet einen Sampling-Algorithmus an, um sicherzustellen, dass die Nachverfolgung effizient ist, und stellt dennoch ein repräsentatives Beispiel aller Anfragen bereit. Die Samplingrate beträgt 1 Anforderung pro Sekunde und 5 Prozent aller weiteren Anforderungen. Sie können die X-Ray-Samplingrate nicht für Ihre Funktionen konfigurieren.

Verwenden von Powertools für AWS Lambda (Python) und AWS CDK für die Nachverfolgung

Führen Sie die nachstehenden Schritte aus, um eine Hello-World-Beispielanwendung für Python mit integrierten Modulen von Powertools for AWS Lambda (Python) unter Verwendung von AWS CDK bereitzustellen. Diese Anwendung implementiert ein grundlegendes API-Backend und verwendet Powertools zum Ausgeben von Protokollen, Metriken und Traces. Es besteht aus einem Amazon-API-Gateway-Endpunkt und einer Lambda-Funktion. Wenn Sie eine GET-Anforderung an den API-Gateway-Endpunkt senden, startet die Lambda-Funktion, sendet Protokolle und Metriken im Embedded Metric Format an CloudWatch und sendet Traces an AWS X-Ray. Die Funktion gibt eine Hello-World-Nachricht zurück.

Voraussetzungen

Zur Durchführung der Schritte in diesem Abschnitt benötigen Sie Folgendes:

- Python 3.11
- AWS CLI Version 2
- AWS CDK Version 2
- AWS SAM-CLI-Version 1.75 oder höher. Wenn Sie eine ältere Version der AWS SAM-CLI haben, finden Sie weitere Informationen unter Aktualisieren der AWS SAM-CLI.

Bereitstellen einer AWS CDK-Beispielanwendung

1. Erstellen Sie ein Projektverzeichnis für Ihre neue Anwendung.

```
mkdir hello-world
cd hello-world
```

2. Initialisieren Sie die App.

```
cdk init app --language python
```

3. Installieren Sie die Python-Abhängigkeiten.

```
pip install -r requirements.txt
```

4. Erstellen Sie ein Verzeichnis lambda_function unter dem Stammordner.

```
mkdir lambda_function
cd lambda_function
```

5. Erstellen Sie eine Datei namens app.py und fügen Sie den folgenden Code zur Datei hinzu. Dies ist der Code für die Lambda-Funktion.

```
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. Öffnen Sie das Verzeichnis hello_world. Sie sollten eine Datei mit dem Namen hello_world_stack.py sehen.

```
cd ..
cd hello_world
```

7. Öffnen Sie hello_world_stack.py und fügen Sie den folgenden Code in die Datei ein. Dieser enthält den Lambda Constructor, der die Lambda-Funktion erstellt, die Umgebungsvariablen für Powertools konfiguriert und die Protokollaufbewahrung auf eine Woche festlegt, sowie den ApiGatewayv1 Constructor, der die REST-API erstellt.

```
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. Stellen Sie die Anwendung bereit.

```
cd ..
cdk deploy
```

9. Rufen Sie die URL der bereitgestellten Anwendung ab:

```
aws cloudformation describe-stacks --stack-name HelloWorldStack --query
 'Stacks[0].Outputs[?OutputKey==`apiUrl`].OutputValue' --output text
```

10. Rufen Sie den API-Endpunkt auf:

```
curl -X GET <URL_FROM_PREVIOUS_STEP>
```

    Wenn der Link erfolgreich ausgeführt wurde, sehen Sie die folgende Antwort:

```
{"message":"hello world"}
```

11. Führen Sie sam traces aus, um die Traces für die Funktion zu erhalten.

```
sam traces
```

Das Nachverfolgungsergebnis sieht folgendermaßen aus:

```
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. Dies ist ein öffentlicher API-Endpunkt, der über das Internet zugänglich ist. Es wird empfohlen, dass Sie den Endpunkt nach dem Testen löschen.

```
cdk destroy
```

Verwenden von ADOT zum Instrumentieren Ihrer Python-Funktionen

ADOT bietet vollständig verwaltete Lambda-Ebenen, die alles packen, was Sie zum Sammeln von Telemetriedaten mit dem OTel-SDK benötigen. Indem Sie diese Ebene verwenden, können Sie Ihre Lambda-Funktionen instrumentieren, ohne einen Funktionscode ändern zu müssen. Sie können Ihren Ebenen auch für die benutzerdefinierte Initialisierung von OTel konfigurieren. Weitere Informationen finden Sie unter Benutzerdefinierte Konfiguration für den ADOT Collector auf Lambda in der ADOT-Dokumentation.

Für Python-Laufzeiten können Sie den AWS-verwaltete Lambda-Layer für ADOT Python hinzufügen, um Ihre Funktionen automatisch zu instrumentieren. Dieser Layer funktioniert sowohl für arm64- als auch für x86_64-Architekturen. Ausführliche Anweisungen dazu, wie Sie diesen Layer hinzufügen, finden Sie unter AWS Distro für OpenTelemetry-Lambda-Unterstützung für Python in der ADOT-Dokumentation.

Verwenden des X-Ray-SDK zum Instrumentieren Ihrer Python-Funktionen

Um Details zu Aufrufen aufzuzeichnen, die Ihre Lambda-Funktion an andere Ressourcen in Ihrer Anwendung vornimmt, können Sie auch AWS X-Ray-SDK für Python verwenden. Um das SDK zu erhalten, fügen Sie das `aws-xray-sdk`-Paket den Abhängigkeiten Ihrer Anwendung hinzu.

Example requirements.txt

```
jsonpickle==1.3
aws-xray-sdk==2.4.3
```

In Ihrem Funktionscode können Sie AWS-SDK-Clients durch Patchen der `boto3`-Bibliothek mit dem `aws_xray_sdk.core`-Modul instrumentieren.

Example Funktion – Nachverfolgen eines AWS-SDK-Clients

```
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)))
  ...
```

Aktivieren Sie nach Hinzufügen der richtigen Abhängigkeiten die Nachverfolgung in der Konfiguration Ihrer Funktion über die Lambda-Konsole oder die API.

Aktivieren der Nachverfolgung mit der Lambda-Konsole

Gehen Sie folgendermaßen vor, um die aktive Nachverfolgung Ihrer Lambda-Funktion mit der Konsole umzuschalten:

So aktivieren Sie die aktive Nachverfolgung

1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.
2. Wählen Sie eine Funktion aus.
3. Wählen Sie Configuration (Konfiguration) und dann Monitoring and operations tools (Überwachungs- und Produktionstools).
4. Wählen Sie unter Zusätzliche Überwachungstools die Option Bearbeiten aus.
5. Wählen Sie unter CloudWatch Application Signals und AWS X-Ray für Lambda-Service-Nachverfolgungen die Option Aktivieren aus.
6. Wählen Sie Speichern.

Aktivieren der Nachverfolgung mit der Lambda-API

Konfigurieren Sie die Nachverfolgung für Ihre Lambda-Funktion mit der AWS CLI oder dem AWS-SDK. Verwenden Sie die folgenden API-Operationen:

- UpdateFunctionConfiguration

- GetFunctionConfiguration
- CreateFunction

Der folgende AWS CLI-Beispielbefehl aktiviert die aktive Nachverfolgung für eine Funktion mit dem Namen my-function.

```
aws lambda update-function-configuration --function-name my-function \
--tracing-config Mode=Active
```

Der Ablaufverfolgungsmodus ist Teil der versionsspezifischen Konfiguration, wenn Sie eine Version Ihrer Funktion veröffentlichen. Sie können den Ablaufverfolgungsmodus für eine veröffentlichte Version nicht ändern.

Aktivieren von Nachverfolgungen mit CloudFormation

Verwenden Sie die `AWS::Lambda::Function`-Eigenschaft, um die Nachverfolgung für eine CloudFormation-Ressource in einer `TracingConfig`-Vorlage zu aktivieren.

Example function-inline.yml – Ablaufverfolgungskonfiguration

```
Resources:
  function:
    Type: AWS::Lambda::Function
    Properties:
      TracingConfig:
        Mode: Active
      ...
```

Verwenden Sie für eine AWS Serverless Application Model (AWS SAM) `AWS::Serverless::Function`-Ressource die Eigenschaft `Tracing`.

Example template.yml – Ablaufverfolgungskonfiguration

```
Resources:
  function:
    Type: AWS::Serverless::Function
    Properties:
      Tracing: Active
      ...
```

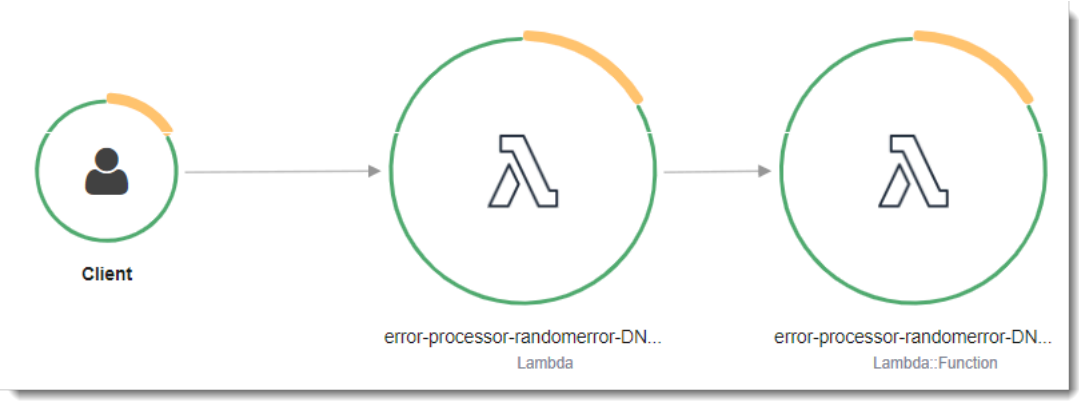
Interpretieren einer X-Ray-Nachverfolgung

Ihre Funktion benötigt die Berechtigung zum Hochladen von Trace-Daten zu X-Ray. Wenn Sie die aktive Nachverfolgung in der Lambda-Konsole aktivieren, fügt Lambda der Ausführungsrolle Ihrer Funktion die erforderlichen Berechtigungen hinzu. Andernfalls fügen Sie der Ausführungsrolle die AWSXRayDaemonWriteAccess-Richtlinie hinzu.

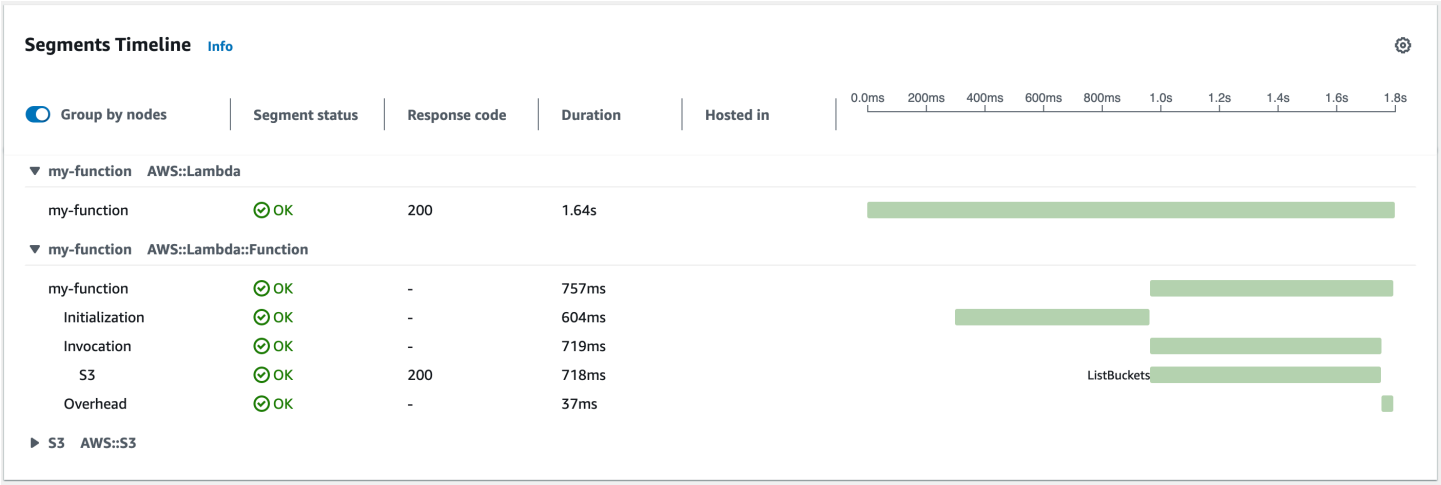
Nachdem Sie die aktive Nachverfolgung konfiguriert haben, können Sie bestimmte Anfragen über Ihre Anwendung beobachten. Das X-Ray-Service-Diagramm zeigt Informationen über Ihre Anwendung und alle ihre Komponenten an. Das folgende Beispiel zeigt eine Anwendung mit zwei Funktionen. Die primäre Funktion verarbeitet Ereignisse und gibt manchmal Fehler zurück. Die zweite Funktion ganz oben verarbeitet Fehler, die in der Protokollgruppe der ersten Protokollgruppe erscheinen, und verwendet das AWS-SDK, um X-Ray, Amazon Simple Storage Service (Amazon S3) und Amazon CloudWatch Logs aufzurufen.

X-Ray verfolgt nicht alle Anfragen an Ihre Anwendung nach. X-Ray wendet einen Sampling-Algorithmus an, um sicherzustellen, dass die Nachverfolgung effizient ist, und stellt dennoch ein repräsentatives Beispiel aller Anfragen bereit. Die Samplingrate beträgt 1 Anforderung pro Sekunde und 5 Prozent aller weiteren Anforderungen. Sie können die X-Ray-Samplingrate nicht für Ihre Funktionen konfigurieren.

In X-Ray, zeichnet eine Ablaufverfolgung Informationen zu einer Anforderung auf, die von einem oder mehreren Services verarbeitet wird. Lambda zeichnet 2 Segmente pro Trace auf, wodurch zwei Knoten im Dienstgraphen entstehen. In der folgenden Abbildung werden diese beiden Knoten hervorgehoben:



Der erste Knoten auf der linken Seite stellt den Lambda-Service dar, der die Aufrufanforderung empfängt. Der zweite Knoten stellt Ihre spezifische Lambda-Funktion dar. Das folgende Beispiel zeigt eine Nachverfolgung mit diesen zwei Segmenten. Beide heißen my-function, aber einer hat einen Ursprung von `AWS::Lambda` und der andere hat einen Ursprung von `AWS::Lambda::Function`. Wenn das `AWS::Lambda`-Segment einen Fehler anzeigt, hatte der Lambda-Service ein Problem. Wenn das `AWS::Lambda::Function`-Segment einen Fehler anzeigt, ist bei Ihrer Funktion ein Problem aufgetreten.



Dieses Beispiel erweitert das `AWS::Lambda::Function`-Segment, um seine drei Untersegmente anzuzeigen.

> **Note**
>
> AWS implementiert derzeit Änderungen am Lambda-Service. Aufgrund dieser Änderungen können geringfügige Unterschiede in Struktur und Inhalt der Systemprotokollmeldungen und Trace-Segmente auftreten, die von verschiedenen Lambda-Funktionen in Ihrem AWS-Konto.

Der hier gezeigte Beispiel-Trace veranschaulicht das Funktionssegment im alten Stil. Die Unterschiede zwischen den Segmenten im alten und im neuen Stil werden in den folgenden Abschnitten beschrieben.
Diese Änderungen werden in den kommenden Wochen implementiert, und alle Funktionen in allen AWS-Regionen-Regionen mit Ausnahme von China und GovCloud werden auf das neue Format der Protokollmeldungen und Trace-Segmente umgestellt.

Das Funktionssegment im alten Stil enthält die folgenden Untersegmente:

- Initialisierung – Stellt die Zeit dar, die für das Laden Ihrer Funktion und das Ausführen des Initialisierungscodes aufgewendet wurde. Dieses Untersegment erscheint nur für das erste Ereignis, das jede Instance Ihrer Funktion verarbeitet.
- Invocation (Aufruf) – Stellt die Zeit dar, die beim Ausführen Ihres Handler-Codes vergeht.
- Overhead (Aufwand) – Stellt die Zeit dar, die von der Lambda-Laufzeitumgebung bei der Verarbeitung des nächsten Ereignisses verbraucht wird.

Das Funktionssegment im neuen Stil enthält kein `Invocation`-Untersegment. Stattdessen werden Kundenuntersegmente direkt an das Funktionssegment angehängt. Weitere Informationen über die Struktur der alten und neuen Funktionssegmente finden Sie unter the section called “Grundlegendes zu X-Ray-Ablaufverfolgungen”.

Sie können auch HTTP-Clients instrumentieren, SQL-Abfragen aufzeichnen und benutzerdefinierte Untersegmente mit Anmerkungen und Metadaten erstellen. Weitere Informationen finden Sie unter AWS X-Ray-SDK für Python im AWS X-Ray-Entwicklerhandbuch.

Preisgestaltung

Sie können X-Ray-Nachverfolgung jeden Monat bis zu einem bestimmten Limit als Teil des AWS kostenlosen Kontingents kostenlos nutzen. Über den Schwellenwert hinaus berechnet X-Ray Gebühren für die Speicherung und den Abruf der Nachverfolgung. Weitere Informationen finden Sie unter AWS X-Ray Preise.

Laufzeitabhängigkeiten in einer Ebene speichern (X-Ray-SDK)

Wenn Sie das X-Ray-SDK verwenden, um mit AWS-SDK-Clients Ihren Funktionscode zu instrumentieren, kann Ihr Bereitstellungspaket ziemlich groß werden. Um Laufzeitabhängigkeiten bei jeder Aktualisierung des Funktionscodes zu vermeiden, verpacken Sie das X-Ray-SDK in einer Lambda-Ebene.

Das folgende Beispiel zeigt eine `AWS::Serverless::LayerVersion`-Ressource, die das AWS X-Ray-SDK für Python speichert.

Example template.yml – Abhängigkeitenebene

```
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
```

Bei dieser Konfiguration aktualisieren Sie die Bibliotheksebene nur, wenn Sie Ihre Laufzeitabhängigkeiten ändern. Da das Funktionsbereitstellungspaket nur Ihren Code enthält, kann dies dazu beitragen, die Upload-Zeiten zu reduzieren.

Das Erstellen einer Ebene für Abhängigkeiten erfordert Build-Konfigurationsänderungen, um das Ebenen-Archiv vor der Bereitstellung zu generieren. Ein funktionierendes Beispiel finden Sie in der Beispielanwendung blank-python .

Erstellen von Lambda-Funktionen mit Ruby

Sie können Ruby-Code in AWS Lambda ausführen. Lambda bietet Laufzeiten für Ruby, die Ihren Code ausführen, um Ereignisse zu verarbeiten. Ihr Code wird in einer Umgebung ausgeführt, die die Rolle AWS SDK für Ruby, with credentials from a AWS Identity and Access Management (IAM) enthält, die Sie verwalten. Weitere Informationen zu den SDK-Versionen, die in den Laufzeiten von Ruby enthalten sind, finden Sie unter the section called “SDK-Versionen, die zur Laufzeit enthalten sind”.

Lambda unterstützt die folgenden Ruby-Laufzeiten.

Name	ID	Betriebssystem	Datum der Veraltung	Blockfunktion erstellen	Blockfunktion aktualisieren
Ruby 3.4	`ruby3.4`	Amazon Linux 2023	31. März 2028	30. April 2028	31. Mai 2028
Ruby 3.3	`ruby3.3`	Amazon Linux 2023	31. März 2027	30. April 2027	31. Mai 2027
Ruby 3.2	`ruby3.2`	Amazon Linux 2	31. März 2026	31. August 2026	30. September 2026

So erstellen Sie eine Ruby-Funktion

1. Öffnen Sie die Lambda-Konsole.
2. Wählen Sie Funktion erstellen.
3. Konfigurieren Sie die folgenden Einstellungen:
   - Funktionsname: Geben Sie einen Namen für die Funktion ein.
   - Runtime: Wählen Sie Ruby 3.4.
4. Wählen Sie Funktion erstellen.

Die Konsole erstellt eine Lambda-Funktion mit einer einzigen Quelldatei mit dem Namen `lambda_function.rb`. Mit dem integrierten Code-Editor können Sie diese Datei bearbeiten und

weitere Dateien hinzufügen. Wählen Sie im Abschnitt BEREITSTELLEN die Option Bereitstellen aus, um den Code Ihrer Funktion zu aktualisieren. Um Ihren Code auszuführen, wählen Sie anschließend im Abschnitt TESTEREIGNISSE die Option Testereignis erstellen aus.

Die `lambda_function.rb`-Datei exportiert eine Funktion mit dem Namen `lambda_handler`, die ein Ereignisobjekt und ein Kontext-Objekt übernimmt. Dies ist die Handler-Funktion, die bei einem Aufruf der Funktion von Lambda aufgerufen wird. Die Ruby-Funktionslaufzeit ruft Aufrufereignisse von Lambda ab und leitet sie an den Handler weiter. In der Konfiguration der Funktion lautet der Wert für den Handler `lambda_function.lambda_handler`.

Wenn Sie Ihren Funktionscode speichern, erstellt die Lambda-Konsole ein Bereitstellungspaket für das ZIP-Dateiarchiv. Wenn Sie Ihren Funktionscode außerhalb der Konsole (mit einer IDE) entwickeln, müssen Sie ein Bereitstellungspaket erstellen, um Ihren Code in die Lambda-Funktion hochzuladen.

Die Funktionslaufzeit übergibt neben dem Aufrufereignis ein Context-Objekt an den Handler. Das Context-Objekt enthält zusätzliche Informationen zum Aufruf, zur Funktion und zur Ausführungsumgebung. Weitere Informationen erhalten Sie über die Umgebungsvariablen.

Ihre Lambda-Funktion wird mit einer CloudWatch Logs-Protokollgruppe geliefert. Die Funktionslaufzeit sendet Details zu jedem Aufruf an CloudWatch Logs. Es leitet alle Protokolle weiter, die Ihre Funktion während des Aufrufs ausgibt. Wenn Ihre Funktion einen Fehler zurückgibt, formatiert Lambda den Fehler und gibt ihn an den Aufrufer zurück.

Themen

- SDK-Versionen, die zur Laufzeit enthalten sind
- Aktivieren von Yet Another Ruby JIT (YJIT)
- Definieren des Lambda-Funktions-Handlers in Ruby
- Bereitstellen von Ruby-Lambda-Funktionen mit ZIP-Dateiarchiven
- Bereitstellen von Ruby-Lambda-Funktionen mit Container-Images
- Arbeiten mit Ebenen für Ruby Lambda-Funktionen
- Verwenden des Lambda-Kontextobjekts zum Abrufen von Ruby-Funktionsinformationen
- Ruby-Lambda-Funktionen protokollieren und überwachen
- Instrumentieren von Ruby-Code in AWS Lambda

SDK-Versionen, die zur Laufzeit enthalten sind

Die Version des AWS SDK, die in der Ruby-Runtime enthalten ist, hängt von der Laufzeitversion und Ihrer AWS-Region ab. Das AWS SDK for Ruby ist modular konzipiert und wird durch getrennt AWS-Service. Um die Versionsnummer eines bestimmten Dienst-Gems zu ermitteln, das in der von Ihnen verwendeten Laufzeitumgebung enthalten ist, erstellen Sie eine Lambda-Funktion mit Code in folgendem Format. Ersetzen Sie `aws-sdk-s3` und `Aws::S3` durch den Namen der Service-Gems, die Ihr Code verwendet.

```
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
```

Aktivieren von Yet Another Ruby JIT (YJIT)

Die Ruby-3.2-Laufzeit unterstützt YJIT, einen einfachen, minimalistischen Ruby-JIT-Compiler. YJIT bietet eine deutlich höhere Leistung, benötigt aber auch mehr Speicher als der Ruby-Interpreter. YJIT wird für Ruby-on-Rails-Workloads empfohlen.

YJIT ist standardmäßig nicht aktiviert. Um YJIT für eine Ruby-3.2-Funktion zu aktivieren, setzen Sie die Umgebungsvariable `RUBY_YJIT_ENABLE` auf 1. Um sicherzustellen, dass YJIT aktiviert ist, drucken Sie das Ergebnis der `RubyVM::YJIT.enabled?`-Methode aus.

Example– Sicherstellen, dass YJIT aktiviert ist

```
puts(RubyVM::YJIT.enabled?())
# => true
```

Definieren des Lambda-Funktions-Handlers in Ruby

Der Lambda-Funktionshandler ist die Methode in Ihrem Funktionscode, die Ereignisse verarbeitet. Wenn Ihre Funktion aufgerufen wird, führt Lambda die Handler-Methode aus. Ihre Funktion wird so lange ausgeführt, bis der Handler eine Antwort zurückgibt, beendet wird oder ein Timeout auftritt.

Themen

- Ruby-Handler-Grundlagen
- Bewährte Codemethoden für Ruby-Lambda-Funktionen

Ruby-Handler-Grundlagen

Im folgenden Beispiel definiert die Datei `function.rb` eine Handler-Methode mit dem Namen `handler`. Die Handler-Funktion nimmt zwei Objekten als Eingabe und gibt ein JSON-Dokument zurück.

Example function.rb

```
require 'json'

def handler(event:, context:)
    { event: JSON.generate(event), context: JSON.generate(context.inspect) }
end
```

In Ihrer Funktionskonfiguration gibt die Einstellung `handler` Lambda die Stelle an, an der sich der Handler befindet. Im vorherigen Beispiel ist der korrekte Wert für diese Einstellung **`function.handler`**. Er enthält zwei Namen, die durch einen Punkt getrennt werden: den Namen der Datei und den Namen der Handler-Methode.

Sie können Ihre Handler-Methode außerdem in einer Klasse definieren. Das folgende Beispiel definiert eine Handler-Methode mit dem Namen `process` in einer Klasse mit dem Namen `Handler` in einem Modul mit dem Namen `LambdaFunctions`.

Example source.rb

```
module LambdaFunctions
  class Handler
    def self.process(event:,context:)
```

```
      "Hello!"
    end
  end
end
```

In diesem Fall ist die Handler-Einstellung **`source.LambdaFunctions::Handler.process`**.

Die zwei Objekte, die der Handler akzeptiert, sind das Aufrufereignis und Context. Das Ereignis ist ein Ruby-Objekt, das die vom Aufrufer bereitgestellte Nutzlast enthält. Wenn es sich bei der Nutzlast um ein JSON-Dokument handelt, ist das Ereignisobjekt ein Ruby-Hash. Andernfalls ist es eine Zeichenfolge. Das Context-Objekt besitzt Methoden und Eigenschaften, die Informationen zum Aufruf, zur Funktion und zur Ausführungsumgebung bereitstellen.

Der Funktions-Handler wird jedes Mal ausgeführt, wenn Ihre Lambda-Funktion aufgerufen wird. Statischer Code außerhalb des Handler wird einmal pro Funktions-Instance ausgeführt. Wenn Ihr Handler Ressourcen wie SDK-Clients und Datenbankverbindungen verwendet, können Sie diese außerhalb der Handler-Methode erstellen, um sie für mehrere Aufrufe wiederzuverwenden.

Jede Instance Ihrer Funktion kann zwar mehrere Aufrufereignisse verarbeiten, verarbeitet jedoch jeweils nur ein Ereignis zur gleichen Zeit. Die Anzahl der Instances, die zu einem bestimmten Zeitpunkt ein Ereignis verarbeiten, ist die Nebenläufigkeit Ihrer Funktion. Weitere Informationen zur Lambda-Ausführungsumgebung finden Sie unter Verständnis des Lebenszyklus der Lambda-Ausführungsumgebung.

Bewährte Codemethoden für Ruby-Lambda-Funktionen

Halten Sie sich an die Richtlinien in der folgenden Liste, um beim Erstellen Ihrer Lambda-Funktionen die besten Codierungspraktiken anzuwenden:

- Trennen Sie den Lambda-Handler von Ihrer Core-Logik. Auf diese Weise können Sie eine Funktion zur besseren Prüfbarkeit von Einheiten schaffen. In Ruby kann dies beispielsweise wie folgt aussehen:

```
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
```

- Kontrollieren Sie die Abhängigkeiten im Bereitstellungspaket Ihrer Funktion. Die AWS Lambda-Ausführungsumgebung enthält eine Reihe von Bibliotheken. Für die Ruby-Laufzeit gehört dazu das AWS-SDK. Um die neuesten Funktionen und Sicherheitsupdates zu aktivieren, wird Lambda diese Bibliotheken regelmäßig aktualisieren. Diese Updates können das Verhalten Ihrer Lambda-Funktion geringfügig verändern. Um die Abhängigkeiten, die Ihre Funktion verwendet, vollständig zu kontrollieren, empfehlen wir, alle Abhängigkeiten mit Ihrem Bereitstellungspaket zu bündeln.
- Minimieren Sie die Komplexität Ihrer Abhängigkeiten. Ziehen Sie einfachere Frameworks vor, die sich schnell beim Start der Ausführungsumgebung laden lassen.
- Minimieren Sie die Größe Ihres Bereitstellungspakets auf die für die Laufzeit erforderliche Größe. Dadurch verkürzt sich die Zeit, die für das Herunterladen und Entpacken Ihres Bereitstellungspakets vor dem Aufruf benötigt wird. Für Funktionen, die in Ruby erstellt wurden, sollten Sie das Hochladen der gesamten AWS-SDK-Bibliothek als Teil Ihres Bereitstellungspakets vermeiden. Hängen Sie stattdessen selektiv von den Gems ab, die Komponenten des von Ihnen benötigten SDK aufnehmen (z. B. die DynamoDB- oder Amazon S3-SDK-Gems).

Nutzen Sie die Wiederverwendung der Ausführungsumgebung zur Verbesserung Ihrer Funktion. Initialisieren Sie SDK-Clients und Datenbankverbindungen außerhalb des Funktions-Handlers und speichern Sie statische Komponenten lokal im `/tmp`-Verzeichnis. Nachfolgende Aufrufe, die von derselben Instance Ihrer Funktion verarbeitet werden, können diese Ressourcen wiederverwenden. Dies spart Kosten durch Reduzierung der Funktionslaufzeit.

Um potenzielle Datenlecks über Aufrufe hinweg zu vermeiden, verwenden Sie die Ausführungsumgebung nicht, um Benutzerdaten, Ereignisse oder andere Informationen mit Sicherheitsauswirkungen zu speichern. Wenn Ihre Funktion auf einem veränderbaren Zustand beruht, der nicht im Speicher innerhalb des Handlers gespeichert werden kann, sollten Sie für jeden Benutzer eine separate Funktion oder separate Versionen einer Funktion erstellen.

Verwenden Sie eine Keep-Alive-Direktive, um dauerhafte Verbindungen zu pflegen. Lambda bereinigt Leerlaufverbindungen im Laufe der Zeit. Der Versuch, eine Leerlaufverbindung beim Aufruf einer Funktion wiederzuverwenden, führt zu einem Verbindungsfehler. Um Ihre persistente Verbindung aufrechtzuerhalten, verwenden Sie die Keep-Alive-Direktive, die Ihrer Laufzeit zugeordnet ist. Ein Beispiel finden Sie unter Wiederverwenden von Verbindungen mit Keep-Alive in Node.js.

Verwenden Sie Umgebungsvariablen um Betriebsparameter an Ihre Funktion zu übergeben. Wenn Sie z. B. Daten in einen Amazon-S3-Bucket schreiben, anstatt den Bucket-Namen, in den Sie schreiben, hartzucodieren, konfigurieren Sie den Bucket-Namen als Umgebungsvariable.

Vermeiden Sie rekursive Aufrufe in Ihrer Lambda-Funktion, bei denen die Funktion sich selbst aufruft oder einen Prozess initiiert, der die Funktion erneut aufrufen kann. Dies kann zu unvorhergesehenen Mengen an Funktionsaufrufen führen und höhere Kosten zur Folge haben. Wenn Sie eine unbeabsichtigte Menge von Aufrufen feststellen, legen Sie die reservierte gleichzeitige Ausführung der Funktion auf `0` fest, um sofort alle Aufrufe der Funktion zu drosseln, während Sie den Code aktualisieren.

Verwenden Sie keine nicht dokumentierten, nicht öffentlichen APIs in Ihrem Lambda-Funktionscode. Für AWS Lambda-verwaltete Laufzeiten wendet Lambda regelmäßig Sicherheits- und Funktionsupdates auf Lambdas interne APIs an. Diese internen API-Updates können abwärtskompatibel sein, was zu unbeabsichtigten Konsequenzen wie Aufruffehlern führt, wenn Ihre Funktion von diesen nicht öffentlichen APIs abhängig ist. Eine Liste öffentlich zugänglicher APIs finden Sie in der API-Referenz.

Schreiben Sie idempotenten Code. Das Schreiben idempotenter Code für Ihre Funktionen stellt sicher, dass doppelte Ereignisse auf die gleiche Weise behandelt werden. Ihr Code sollte Ereignisse ordnungsgemäß validieren und doppelte Ereignisse ordnungsgemäß behandeln. Weitere Informationen finden Sie unter Wie mache ich meine Lambda-Funktion idempotent?.

Bereitstellen von Ruby-Lambda-Funktionen mit ZIP-Dateiarchiven

Der Code Ihrer AWS Lambda-Funktion besteht aus einer RB-Datei, die den Handler-Code Ihrer Funktion sowie sämtliche zusätzliche Abhängigkeiten (Gems) enthält, von denen Ihr Code ggf. abhängt. Sie verwenden ein Bereitstellungspaket, um Ihren Funktionscode in Lambda bereitzustellen. Dieses Paket kann entweder ein ZIP-Dateiarchiv oder ein Container-Image sein. Weitere Informationen zur Verwendung von Container-Images mit Ruby finden Sie unter Bereitstellen von Ruby-Lambda-Funktionen mit Container-Images.

Zum Erstellen des Bereitstellungspakets für ein ZIP-Dateiarchiv können Sie ein integriertes Dienstprogramm für ZIP-Dateien Ihres Befehlszeilen-Tools oder ein anderes Dienstprogramm für ZIP-Dateien verwenden, wie 7zip. In den Beispiele in den folgenden Abschnitten wird davon ausgegangen, dass Sie ein `zip`-Befehlszeilen-Tool in einer Linux- oder MacOS-Umgebung verwenden. Unter Windows können Sie das Windows-Subsystem für Linux installieren, um eine Windows-Version von Ubuntu und Bash zu erhalten und dieselben Befehle zu verwenden.

Da Lambda POSIX-Dateiberechtigungen verwendet, müssen Sie möglicherweise Berechtigungen für den Bereitstellungspaketordner festlegen, bevor Sie das ZIP-Dateiarchiv erstellen.

In den Beispielbefehlen in den folgenden Abschnitten wird das Hilfsprogramm Bundler verwendet, um Ihrem Bereitstellungspaket Abhängigkeiten hinzuzufügen. Führen Sie zum Installieren von Bundler den folgenden Befehl aus:

```
gem install bundler
```

Abschnitte

- Abhängigkeiten in Ruby
- ZIP-Bereitstellungspakets ohne Abhängigkeiten erstellen
- ZIP-Bereitstellungspakets mit Abhängigkeiten erstellen
- Erstellen einer Ruby-Ebene für Ihre Abhängigkeiten
- Erstellen von ZIP-Bereitstellungspaketen mit nativen Bibliotheken
- Erstellen und Aktualisieren von Ruby-Lambda-Funktionen mithilfe von ZIP-Dateien

Abhängigkeiten in Ruby

Bei Lambda-Funktionen, die die Ruby-Laufzeit verwenden, kann eine Abhängigkeit ein beliebiges Ruby-Gem sein. Bei Bereitstellung Ihrer Funktion mithilfe eines ZIP-Archivs können Sie diese

Abhängigkeiten entweder mit Ihrem Funktionscode zu Ihrer ZIP-Datei hinzufügen oder eine Lambda-Ebene verwenden. Eine Ebene ist ein separates ZIP-Dateiarchiv, das zusätzlichen Code und andere Daten enthalten kann. Weitere Informationen zur Verwendung von Lambda-Ebenen finden Sie unter Lambda-Ebenen.

Die Ruby-Laufzeit beinhaltet die AWS SDK für Ruby. Wenn Ihre Funktion das SDK verwendet, müssen Sie es nicht mit Ihrem Code bündeln. Wenn Sie jedoch die volle Kontrolle über Ihre Abhängigkeiten behalten oder eine bestimmte Version des SDK verwenden möchten, können Sie es dem Bereitstellungspaket Ihrer Funktion hinzufügen. Sie können das SDK entweder in Ihre ZIP-Datei aufnehmen oder es mithilfe einer Lambda-Ebene hinzufügen. Abhängigkeiten in Ihrer ZIP-Datei oder auf Lambda-Ebenen haben Vorrang vor Versionen, die in der Laufzeit enthalten sind. Welche Version des SDK für Ruby in Ihrer Laufzeitversion enthalten ist, erfahren Sie unter the section called “SDK-Versionen, die zur Laufzeit enthalten sind”.

Im Rahmen des AWS-Modells der geteilten Verantwortung sind Sie für die Verwaltung aller Abhängigkeiten in den Bereitstellungspaketen Ihrer Funktionen verantwortlich. Dies beinhaltet das Durchführen von Updates und Sicherheitspatches. Zum Aktualisieren von Abhängigkeiten im Bereitstellungspaket Ihrer Funktion erstellen Sie zunächst eine neue ZIP-Datei und laden Sie diese dann in Lambda hoch. Weitere Informationen finden Sie unter ZIP-Bereitstellungspakets mit Abhängigkeiten erstellen und Erstellen und Aktualisieren von Ruby-Lambda-Funktionen mithilfe von ZIP-Dateien.

ZIP-Bereitstellungspakets ohne Abhängigkeiten erstellen

Hat Ihr Funktionscode keine Abhängigkeiten, enthält Ihre ZIP-Datei nur die RB-Datei mit dem Handler-Code Ihrer Funktion. Erstellen Sie mit Ihrem bevorzugten ZIP-Programm eine ZIP-Datei mit Ihrer RB-Datei im Stammverzeichnis. Befindet sich die RB-Datei nicht im Stammverzeichnis Ihrer ZIP-Datei, kann Lambda Ihren Code nicht ausführen.

Informationen zum Bereitstellen Ihrer ZIP-Datei zum Erstellen einer neuen Lambda-Funktion oder Aktualisieren einer vorhandenen Funktion, finden Sie unter Erstellen und Aktualisieren von Ruby-Lambda-Funktionen mithilfe von ZIP-Dateien.

ZIP-Bereitstellungspakets mit Abhängigkeiten erstellen

Hängt Ihr Funktionscode von zusätzlichen Ruby-Gems ab, können Sie diese Abhängigkeiten entweder mit Ihrem Funktionscode in Ihre ZIP-Datei aufnehmen oder eine Lambda-Ebene verwenden. In diesem Abschnitt erfahren Sie, wie Sie Ihre Abhängigkeiten in Ihr ZIP-

Bereitstellungspaket aufnehmen. Anweisungen zum Einschließen Ihrer Abhängigkeiten in eine Ebene finden Sie unter the section called “Erstellen einer Ruby-Ebene für Ihre Abhängigkeiten”.

Angenommen, Ihr Funktionscode ist in einer Datei mit dem Namen `lambda_function.rb` in Ihrem Projektverzeichnis gespeichert. Die folgenden CLI-Befehle erstellen Sie eine ZIP-Datei mit dem Namen „`my_deployment_package.zip`“, die Ihren Funktionscode und seine Abhängigkeiten enthält.

Erstellen des Bereitstellungspakets

1. Erstellen Sie in Ihrem Projektverzeichnis eine `Gemfile`-Datei für die Angabe Ihrer Abhängigkeiten.

   ```
   bundle init
   ```

2. Bearbeiten Sie die `Gemfile`-Datei mit Ihrem bevorzugten Texteditor, um die Abhängigkeiten Ihrer Funktion anzugeben. Wenn Sie also beispielsweise das TZInfo-Gem verwenden möchten, bearbeiten Sie Ihre `Gemfile`-Datei so, dass sie wie folgt aussieht:

   ```
   source "https://rubygems.org"
   gem "tzinfo"
   ```

3. Führen Sie den folgenden Befehl aus, um die in Ihrer `Gemfile`-Datei angegebenen Gems in Ihrem Projektverzeichnis zu installieren. Durch diesen Befehl wird `vendor/bundle` als Standardpfad für Gem-Installationen festgelegt:

   ```
   bundle config set --local path 'vendor/bundle' && bundle install
   ```

   Die Ausgabe sollte in etwa wie folgt aussehen:

   ```
   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**
>
> Wenn Sie Gems später wieder global installieren möchten, können Sie den folgenden Befehl ausführen:
>
> ```
> bundle config set --local system 'true'
> ```

4. Erstellen Sie ein ZIP-Dateiarchiv, das die `lambda_function.rb`-Datei mit dem Handler-Code Ihrer Funktion sowie mit den Abhängigkeiten enthält, die Sie im vorherigen Schritt installiert haben.

   ```
   zip -r my_deployment_package.zip lambda_function.rb vendor
   ```

   Die Ausgabe sollte in etwa wie folgt aussehen:

   ```
   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%)
   ...
   ```

Erstellen einer Ruby-Ebene für Ihre Abhängigkeiten

Informationen zum Verpacken Ihrer Ruby-Abhängigkeiten in eine Lambda-Ebene finden Sie unter the section called “Ebenen”.

Erstellen von ZIP-Bereitstellungspaketen mit nativen Bibliotheken

Viele gängige Ruby-Gems wie `nokogiri`, `nio4r` und `mysql` enthalten native, in C geschriebene Erweiterungen. Wenn Sie Ihrem Bereitstellungspaket Bibliotheken hinzufügen, die C-Code enthalten, müssen Sie Ihr Paket korrekt erstellen, damit es mit der Lambda-Ausführungsumgebung kompatibel ist.

Bei Produktionsanwendungen empfiehlt es sich, Ihren Code mit dem AWS Serverless Application Model (AWS SAM) zu erstellen und bereitzustellen. Verwenden Sie in AWS SAM die Option `sam build --use-container`, um Ihre Funktion in einem Lambda-ähnlichen Docker-Container zu erstellen. Weitere Informationen zur Bereitstellung Ihres Funktionscodes mithilfe von AWS SAM finden Sie im AWS SAM-Entwicklerhandbuch unter Building applications.

Wenn Sie ein ZIP-Bereitstellungspaket, das Gems mit nativen Erweiterungen enthält, ohne Verwendung von AWS SAM erstellen möchten, können Sie Ihre Abhängigkeiten alternativ mithilfe eines Containers in einer Umgebung bündeln, die mit der Lambda-Ruby-Laufzeitumgebung identisch ist. Für diese Schritte muss Docker auf Ihrem Build-Computer installiert sein. Weitere Informationen zur Installation von Docker finden Sie unter Install Docker Engine.

So erstellen Sie ein ZIP-Bereitstellungspaket in einem Docker-Container

1. Erstellen Sie auf Ihrem lokalen Build-Computer einen Ordner als Speicherort für Ihren Container. Erstellen Sie in diesem Ordner eine Datei mit dem Namen `dockerfile` und fügen Sie folgenden Code in die Datei ein:

   ```
   FROM public.ecr.aws/sam/build-ruby3.2:latest-x86_64
   RUN gem update bundler
   CMD "/bin/bash"
   ```

2. Führen Sie in dem Ordner, in dem Sie Ihre `dockerfile`-Datei erstellt haben, den folgenden Befehl aus, um den Docker-Container zu erstellen:

   ```
   docker build -t awsruby32 .
   ```

3. Navigieren Sie zu dem Projektverzeichnis, das die Datei vom Typ `.rb` mit dem Handler-Code Ihrer Funktion sowie die `Gemfile`-Datei mit den Abhängigkeiten Ihrer enthält. Führen Sie in diesem Verzeichnis den folgenden Befehl aus, um den Lambda-Ruby-Container zu starten:

   Linux/MacOS

   ```
   docker run --rm -it -v $PWD:/var/task -w /var/task awsruby32
   ```

> **Note**
>
> In macOS wird möglicherweise eine Warnung mit dem Hinweis angezeigt, dass die Plattform des angeforderten Images nicht der erkannten Hostplattform entspricht. Ignorieren Sie diese Warnung.

Windows PowerShell

```
docker run --rm -it -v ${pwd}:var/task -w /var/task awsruby32
```

Wenn Ihr Container startet, sollte eine Bash-Aufforderung angezeigt werden.

```
bash-4.2#
```

4. Konfigurieren Sie das Hilfsprogramm „bundle“, um die in Ihrer `Gemfile`-Datei angegebenen Gems in einem lokalen Verzeichnis namens `vendor/bundle` zu installieren und Ihre Abhängigkeiten zu installieren.

```
bash-4.2# bundle config set --local path 'vendor/bundle' && bundle install
```

5. Erstellen Sie das ZIP-Bereitstellungspaket mit Ihrem Funktionscode und den zugehörigen Abhängigkeiten. In diesem Beispiel hat die Datei, die den Handler-Code Ihrer Funktion enthält, den Namen `lambda_function.rb`.

```
bash-4.2# zip -r my_deployment_package.zip lambda_function.rb vendor
```

6. Verlassen Sie den Container und kehren Sie zu Ihrem lokalen Projektverzeichnis zurück.

```
bash-4.2# exit
```

Nun können Sie Ihre Lambda-Funktion mithilfe des ZIP-Bereitstellungspakets erstellen oder aktualisieren. Siehe Erstellen und Aktualisieren von Ruby-Lambda-Funktionen mithilfe von ZIP-Dateien

Erstellen und Aktualisieren von Ruby-Lambda-Funktionen mithilfe von ZIP-Dateien

Nach der Erstellung Ihres ZIP-Bereitstellungspakets können Sie es verwenden, um eine neue Lambda-Funktion zu erstellen oder eine vorhandene zu aktualisieren. Ihr ZIP-Bereitstellungspaket lässt sich über die Lambda-Konsole, das AWS Command Line Interface und die Lambda-API hochladen. Sie können Lambda-Funktionen auch mit AWS Serverless Application Model (AWS SAM) und CloudFormation erstellen und aktualisieren.

Die maximale Größe eines ZIP-Bereitstellungspakets für Lambda beträgt 250 MB (entpackt). Beachten Sie, dass dieser Grenzwert für die kombinierte Größe aller hochgeladenen Dateien gilt, einschließlich aller Lambda-Ebenen.

Die Lambda-Laufzeit benötigt die Berechtigung zum Lesen der Dateien in Ihrem Bereitstellungspaket. In der oktalen Linux-Notation der Berechtigungen benötigt Lambda 644 Berechtigungen für nicht ausführbare Dateien (rw-r--r--) und 755 Berechtigungen (rwxr-xr-x) für Verzeichnisse und ausführbare Dateien.

Verwenden Sie unter Linux und MacOS den `chmod`-Befehl, um Dateiberechtigungen für Dateien und Verzeichnisse in Ihrem Bereitstellungspaket zu ändern. Um beispielsweise einer nicht ausführbaren Datei die richtigen Berechtigungen zu geben, führen Sie den folgenden Befehl aus.

```
chmod 644 <filepath>
```

Informationen zum Ändern von Dateiberechtigungen in Windows finden Sie unter Festlegen, Anzeigen, Ändern oder Entfernen von Berechtigungen für ein Objekt in der Microsoft-Windows-Dokumentation.

> **Note**
>
> Wenn Sie Lambda nicht die Berechtigungen gewähren, die es für den Zugriff auf Verzeichnisse in Ihrem Bereitstellungspaket benötigt, setzt Lambda die Berechtigungen für diese Verzeichnisse auf 755 (rwxr-xr-x).

Erstellen und Aktualisieren von Funktionen mit ZIP-Dateien unter Verwendung der Konsole

Eine neue Funktion müssen Sie zuerst in der Konsole erstellen und dann Ihr ZIP-Archiv hochladen. Zum Aktualisieren einer bestehenden Funktion öffnen Sie die Seite für Ihre Funktion und gehen dann genauso vor, um Ihre aktualisierte ZIP-Datei hinzuzufügen.

Bei einer ZIP-Datei mit unter 50 MB können Sie eine Funktion erstellen oder aktualisieren, indem Sie die Datei direkt von Ihrem lokalen Computer hochladen. Bei ZIP-Dateien mit einer Größe von mehr als 50 MB müssen Sie Ihr Paket zuerst in einen Amazon-S3-Bucket hochladen. Anweisungen zum Hochladen einer Datei in einen Amazon-S3-Bucket unter Verwendung der AWS-Managementkonsole finden Sie unter Erste Schritte mit Amazon S3. Informationen zum Hochladen von Dateien mit der AWS CLI finden Sie im AWS CLI-Benutzerhandbuch unter Verschieben von Objekten.

> **Note**
>
> Sie können den Bereitstellungspakettyp (.zip oder Container-Image) für eine vorhandene Funktion nicht ändern. Sie können zum Beispiel eine Container-Image-Funktion nicht so umwandeln, dass sie ein .zip-Archiv verwendet. Sie müssen eine neue Funktion erstellen.

So erstellen Sie eine neue Funktion (Konsole)

1. Öffnen Sie die Funktionsseite der Lambda-Konsole und wählen Sie Funktion erstellen aus.
2. Wählen Sie Author from scratch aus.
3. Führen Sie unter Basic information (Grundlegende Informationen) die folgenden Schritte aus:
   a. Geben Sie als Funktionsname den Namen Ihrer Funktion ein.
   b. Wählen Sie für Laufzeit die Laufzeit aus, die Sie verwenden möchten.
   c. (Optional) Für Architektur wählen Sie die Befehlssatz-Architektur für Ihre Funktion aus. Die Standardarchitektur ist x86_64. Stellen Sie sicher, dass das ZIP-Bereitstellungspaket für Ihre Funktion mit der von Ihnen gewählten Befehlssatzarchitektur kompatibel ist.
4. (Optional) Erweitern Sie unter Berechtigungen die Option Standardausführungsrolle ändern. Sie können eine neue Ausführungsrolle erstellen oder eine vorhandene Rolle verwenden.
5. Wählen Sie Funktion erstellen. Lambda erstellt eine grundlegende „Hello World“-Funktion mit der von Ihnen gewählten Laufzeit.

So laden Sie ein ZIP-Archiv von Ihrem lokalen Computer hoch (Konsole)

1. Wählen Sie auf der Funktionsseite der Lambda-Konsole die Funktion aus, für die Sie die ZIP-Datei hochladen möchten.
2. Wählen Sie die Registerkarte Code aus.
3. Wählen Sie im Bereich Codequelle die Option Hochladen von aus.
4. Wählen Sie die ZIP-Datei aus.
5. Laden Sie die ZIP-Datei wie folgt hoch:
   a. Wählen Sie Hochladen und dann Ihre ZIP-Datei in der Dateiauswahl aus.
   b. Klicken Sie auf Open.
   c. Wählen Sie Speichern.

So laden Sie ein ZIP-Archiv aus einem Amazon-S3-Bucket hoch (Konsole)

1. Wählen Sie auf der Funktionsseite der Lambda-Konsole die Funktion aus, für die Sie eine neue ZIP-Datei hochladen möchten.
2. Wählen Sie die Registerkarte Code aus.
3. Wählen Sie im Bereich Codequelle die Option Hochladen von aus.
4. Wählen Sie den Amazon-S3-Speicherort aus.
5. Fügen Sie die Amazon-S3-Link-URL Ihrer ZIP-Datei ein und wählen Sie Speichern aus.

ZIP-Dateifunktionen mithilfe des Konsolencode-Editors aktualisieren

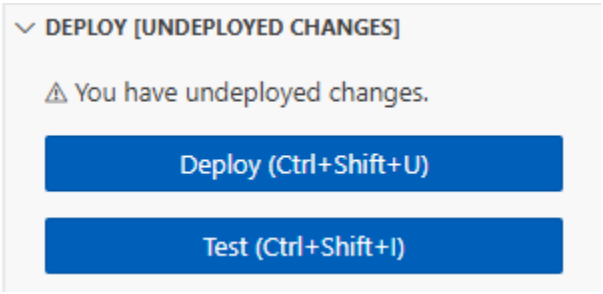
Für einige Funktionen mit ZIP-Bereitstellungspaketen können Sie Ihren Funktionscode direkt mit dem in der Lambda-Konsole integrierten Code-Editor aktualisieren. Zur Verwendung dieses Features muss Ihre Funktion folgende Kriterien erfüllen:

- Ihre Funktion muss eine der interpretierten Sprache der Laufzeit verwenden (Python, Node.js oder Ruby).
- Das Bereitstellungspaket Ihrer Funktion muss kleiner als 50 MB (entpackt) sein.

Funktionscode für Funktionen mit Container-Image-Bereitstellungspaketen kann nicht direkt in der Konsole bearbeitet werden.

So aktualisieren Sie Ihren Funktionscode mit dem Code-Editor

1. Öffnen Sie die Funktionsseite der Lambda-Konsole und wählen Sie Ihre Funktion aus.
2. Wählen Sie die Registerkarte Code aus.
3. Wählen Sie im Bereich Codequelle Ihre Quellcodedatei aus und bearbeiten Sie sie im integrierten Code-Editor.
4. Wählen Sie im Abschnitt BEREITSTELLEN die Option Bereitstellen aus, um den Code Ihrer Funktion zu aktualisieren:



Erstellen und Aktualisieren von Funktionen mit ZIP-Dateien unter Verwendung der AWS CLI

Sie können die AWS CLI verwenden, um eine neue Funktion zu erstellen oder eine vorhandene unter Verwendung einer ZIP-Datei zu aktualisieren. Verwenden Sie die Befehle Funktion erstellen und Funktionscode aktualisieren, um Ihr ZIP-Paket bereitzustellen. Wenn Ihre ZIP-Datei kleiner als 50 MB ist, können Sie das ZIP-Paket von einem Dateispeicherort auf Ihrem lokalen Build-Computer hochladen. Bei größeren Dateien müssen Sie Ihr ZIP-Paket aus einem Amazon-S3-Bucket hochladen. Anweisungen zum Hochladen einer Datei in einen Amazon-S3-Bucket unter Verwendung der AWS CLI finden Sie unter Verschieben von Objekten im AWS CLI-Benutzerhandbuch.

> **Note**
>
> Wenn Sie Ihre ZIP-Datei unter Verwendung der AWS CLI aus einem Amazon-S3-Bucket hochladen, muss sich der Bucket in der gleichen AWS-Region wie Ihre Funktion befinden.

Um eine neue Funktion unter Verwendung einer ZIP-Datei mit der AWS CLI zu erstellen, müssen Sie Folgendes angeben:

- Den Namen Ihrer Funktion (`--function-name`)

- Die Laufzeit Ihrer Funktion (`--runtime`)
- Den Amazon-Ressourcennamen (ARN) der Ausführungsrolle der Funktion (`--role`).
- Den Namen der Handler-Methode in Ihrem Funktionscode (`--handler`)

Sie müssen auch den Speicherort Ihrer ZIP-Datei angeben. Befindet sich Ihre ZIP-Datei in einem Ordner auf Ihrem lokalen Build-Computer, verwenden Sie die `--zip-file`-Option, um den Dateipfad anzugeben, wie im folgenden Beispielbefehl gezeigt.

```
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
```

Zur Angabe des Speicherorts der ZIP-Datei in einem Amazon-S3-Bucket verwenden Sie die im folgenden Beispielbefehl gezeigte `--code`-Option. Sie müssen den `S3ObjectVersion`-Parameter nur für versionierte Objekte verwenden.

```
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
```

Um eine vorhandene Funktion mit der CLI zu aktualisieren, geben Sie den Namen Ihrer Funktion unter Verwendung des `--function-name`-Parameters an. Sie müssen auch den Speicherort der ZIP-Datei angeben, die Sie zum Aktualisieren Ihres Funktionscodes verwenden möchten. Befindet sich Ihre ZIP-Datei in einem Ordner auf Ihrem lokalen Build-Computer, verwenden Sie die `--zip-file`-Option, um den Dateipfad anzugeben, wie im folgenden Beispielbefehl gezeigt.

```
aws lambda update-function-code --function-name myFunction \
--zip-file fileb://myFunction.zip
```

Zur Angabe des Speicherorts der ZIP-Datei in einem Amazon-S3-Bucket verwenden Sie die im folgenden Beispielbefehl gezeigten `--s3-bucket`- und `--s3-key`-Optionen. Sie müssen den `--s3-object-version`-Parameter nur für versionierte Objekte verwenden.

```
aws lambda update-function-code --function-name myFunction \
```

```
--s3-bucket amzn-s3-demo-bucket --s3-key myFileName.zip --s3-object-version myObject
 Version
```

Erstellen und Aktualisieren von Funktionen mit ZIP-Dateien unter Verwendung der Lambda-API

Um Funktionen zu erstellen und zu konfigurieren, die ein ZIP-Dateiarchiv verwenden, verwenden Sie die folgenden API-Operationen:

- CreateFunction
- UpdateFunctionCode

Erstellen und Aktualisieren von Funktionen mit ZIP-Dateien unter Verwendung von AWS SAM

Das AWS Serverless Application Model (AWS SAM) ist ein Toolkit, das den Prozess der Erstellung und Ausführung von Serverless-Anwendungen auf AWS optimiert. Sie definieren die Ressourcen für Ihre Anwendung in einer YAML- oder JSON-Vorlage und verwenden die AWS SAM-Befehlszeilenschnittstelle (AWS SAM CLI), um Ihre Anwendungen zu erstellen, zu verpacken und bereitzustellen. Wenn Sie eine Lambda-Funktion aus einer AWS SAM-Vorlage erstellen, erstellt AWS SAM automatisch ein ZIP-Bereitstellungspaket oder ein Container-Image mit Ihrem Funktionscode und allen von Ihnen angegebenen Abhängigkeiten. Weitere Informationen zur Verwendung von AWS SAM für die Erstellung und Bereitstellung von Lambda-Funktionen finden Sie unter Erste Schritte mit AWS SAM im AWS Serverless Application Model-Entwicklerhandbuch.

Sie können außerdem AWS SAM verwenden, um eine Lambda-Funktion unter Verwendung eines vorhandenen ZIP-Dateiarchivs zu erstellen. Um eine Lambda-Funktion unter Verwendung von AWS SAM zu erstellen, können Sie Ihre ZIP-Datei in einem Amazon-S3-Bucket oder in einem lokalen Ordner auf Ihrem Build-Computer speichern. Anweisungen zum Hochladen einer Datei in einen Amazon-S3-Bucket unter Verwendung der AWS CLI finden Sie unter Verschieben von Objekten im AWS CLI-Benutzerhandbuch.

In Ihrer AWS SAM-Vorlage gibt die `AWS::Serverless::Function`-Ressource Ihre Lambda-Funktion an. Legen Sie in dieser Ressource die folgenden Eigenschaften fest, um eine Funktion zu erstellen, die als ZIP-Datei-Archiv definiert ist:

- `PackageType` – festlegen auf `Zip`

- `CodeUri` – festlegen auf die Amazon-S3-URI, den Pfad zum lokalen Ordner oder das Funktionscode-Objekt
- `Runtime` – festlegen auf die gewünschte Laufzeit

Mit AWS SAM müssen Sie Ihre ZIP-Datei nicht zuerst in einen Amazon-S3-Bucket hochladen, auch wenn sie größer als 50 MB ist. AWS SAM kann ZIP-Pakete bis zur maximal zulässigen Dateigröße von 250 MB (entpackt) von einem Speicherort auf Ihrem lokalen Build-Computer hochladen.

Weitere Informationen zur Bereitstellung von Funktionen mit einer ZIP-Datei in AWS SAM finden Sie unter AWS::Serverless::Function im AWS SAM-Entwicklerhandbuch.

Erstellen und Aktualisieren von Funktionen mit ZIP-Dateien unter Verwendung von CloudFormation

Sie können CloudFormation verwenden, um eine Lambda-Funktion unter Verwendung eines ZIP-Dateiarchivs zu erstellen. Um eine Lambda-Funktion aus einer ZIP-Datei zu erstellen, müssen Sie Ihre Datei zunächst in einen Amazon-S3-Bucket hochladen. Anweisungen zum Hochladen einer Datei in einen Amazon-S3-Bucket unter Verwendung der AWS CLI finden Sie unter Verschieben von Objekten im AWS CLI-Benutzerhandbuch.

In Ihrer CloudFormation-Vorlage gibt die `AWS::Lambda::Function`-Ressource Ihre Lambda-Funktion an. Legen Sie in dieser Ressource die folgenden Eigenschaften fest, um eine Funktion zu erstellen, die als ZIP-Datei-Archiv definiert ist:

- `PackageType` – festlegen auf `Zip`
- `Code` – Geben Sie den Namen des Amazon-S3-Buckets und den ZIP-Dateinamen in die Felder `S3Bucket` und `S3Key` ein
- `Runtime` – festlegen auf die gewünschte Laufzeit

Die von CloudFormation generierte ZIP-Datei darf nicht größer als 4 MB sein. Weitere Informationen zur Bereitstellung von Funktionen mit einer ZIP-Datei in CloudFormation finden Sie unter AWS::Lambda::Function im CloudFormation-Entwicklerhandbuch.

Bereitstellen von Ruby-Lambda-Funktionen mit Container-Images

Es gibt drei Möglichkeiten, ein Container-Image für eine Ruby-Lambda-Funktion zu erstellen:

- Verwenden eines AWS Basis-Images für Ruby

  Die AWS -Basis-Images sind mit einer Sprachlaufzeit, einem Laufzeitschnittstellen-Client zur Verwaltung der Interaktion zwischen Lambda und Ihrem Funktionscode und einem Laufzeitschnittstellen-Emulator für lokale Tests vorinstalliert.

- Es wird ein AWS reines Betriebssystem-Basis-Image verwendet

  AWS Basis-Images nur für Betriebssysteme enthalten eine Amazon Linux-Distribution und den Runtime-Interface-Emulator. Diese Images werden häufig verwendet, um Container-Images für kompilierte Sprachen wie Go und Rust sowie für eine Sprache oder Sprachversion zu erstellen, für die Lambda kein Basis-Image bereitstellt, wie Node.js 19. Sie können reine OS-Basis-Images auch verwenden, um eine benutzerdefinierte Laufzeit zu implementieren. Um das Image mit Lambda kompatibel zu machen, müssen Sie den Laufzeitschnittstellen-Client für Ruby in das Image aufnehmen.

- Verwenden Sie ein Nicht-Basis-Image AWS

  Sie können auch ein alternatives Basis-Image aus einer anderen Container-Registry verwenden. Sie können auch ein von Ihrer Organisation erstelltes benutzerdefiniertes Image verwenden. Um das Image mit Lambda kompatibel zu machen, müssen Sie den Laufzeitschnittstellen-Client für Ruby in das Image aufnehmen.

> **Tip**
>
> Um die Zeit zu reduzieren, die benötigt wird, bis Lambda-Container-Funktionen aktiv werden, siehe die Docker-Dokumentation unter Verwenden mehrstufiger Builds. Um effiziente Container-Images zu erstellen, folgen Sie den Bewährte Methoden für das Schreiben von Dockerfiles.

Auf dieser Seite wird erklärt, wie Sie Container-Images für Lambda erstellen, testen und bereitstellen.

Themen

- AWS Basis-Images für Ruby

- Verwenden eines AWS Basis-Images für Ruby
- Verwenden eines alternativen Basis-Images mit dem Laufzeitschnittstellen-Client

AWS Basis-Images für Ruby

AWS stellt die folgenden Basis-Images für Ruby bereit:

Tags (Markier ngen)	Laufzeit	Betriebss ystem	Dockerfile	Ablehnung
3.4	Ruby 3.4	Amazon Linux 2023	Dockerfile für Ruby 3.4 auf GitHub	31. März 2028
3.3	Ruby 3.3	Amazon Linux 2023	Dockerfile für Ruby 3.3 auf GitHub	31. März 2027
3.2	Ruby 3.2	Amazon Linux 2	Dockerfile für Ruby 3.2 auf GitHub	31. März 2026

Amazon ECR-Repository: gallery.ecr. aws/lambda/ruby

Verwenden eines AWS Basis-Images für Ruby

Voraussetzungen

Zur Durchführung der Schritte in diesem Abschnitt benötigen Sie Folgendes:

- AWS CLI Version 2
- Docker (Mindestversion 25.0.0)
- Das Docker-buildx-Plugin.
- Ruby

Erstellen eines Images aus einem Base Image

Erstellen eines Container-Images für Ruby

1. Erstellen Sie ein Verzeichnis für das Projekt und wechseln Sie dann zu diesem Verzeichnis.

```
mkdir example
cd example
```

2. Erstellen Sie eine neue Datei mit dem Namen `Gemfile`. Hier listen Sie die benötigten RubyGems Pakete für Ihre Anwendung auf. Das AWS SDK für Ruby ist erhältlich bei RubyGems. Sie sollten bestimmte AWS Service Gems für die Installation auswählen. Um beispielsweise das Ruby-Gem für Lambda zu verwenden, sollte Ihr Gemfile so aussehen:

```
source 'https://rubygems.org'

gem 'aws-sdk-lambda'
```

   Alternativ enthält das aws-sdk-Gem jedes verfügbare AWS Service-Gem. Dieses Gem ist sehr groß. Wir empfehlen, es nur zu verwenden, wenn Sie auf viele AWS Dienste angewiesen sind.

3. Installieren Sie die im Gemfile angegebenen Abhängigkeiten mithilfe von bundle install.

```
bundle install
```

4. Erstellen Sie eine neue Datei mit dem Namen `lambda_function.rb`. Sie können der Datei zum Testen den folgenden Beispielfunktionscode hinzufügen oder Ihren eigenen verwenden.

   Example Ruby-Funktion

```
module LambdaFunction
  class Handler
    def self.process(event:,context:)
      "Hello from Lambda!"
    end
  end
end
```

5. Erstellen Sie eine neue Docker-Datei. Es folgt ein Beispiel für ein Dockerfile, das ein AWS - Basis-Image verwendet. Dieses Dockerfile verwendet die folgende Konfiguration:

   - Setzen Sie die `FROM`-Eigenschaft auf den URI des Basis-Images.

- Verwenden Sie den Befehl COPY, um den Funktionscode und die Laufzeitabhängigkeiten in `{LAMBDA_TASK_ROOT}`, eine von Lambda definierte Umgebungsvariable zu kopieren.
- Legen Sie das `CMD`-Argument auf den Lambda-Funktionshandler fest.

Beachten Sie, dass das Dockerfile-Beispiel keine USER-Anweisung enthält. Wenn Sie ein Container-Image für Lambda bereitstellen, definiert Lambda automatisch einen Standard-Linux-Benutzer mit Berechtigungen mit geringsten Rechten. Dies unterscheidet sich vom Standardverhalten von Docker, bei dem standardmäßig der `root`-Benutzer verwendet wird, wenn keine `USER`-Anweisung bereitgestellt wird.

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. Erstellen Sie Ihr Docker-Image mit dem docker build-Befehl. Das folgende Beispiel benennt das Bild in `docker-image` und gibt ihm den `test` Tag. Um Ihr Image mit Lambda kompatibel zu machen, müssen Sie die Option `--provenance=false` verwenden.

```
docker buildx build --platform linux/amd64 --provenance=false -t docker-image:test
 .
```

> **Note**
>
> Der Befehl gibt die `--platform linux/amd64`-Option an, um sicherzustellen, dass Ihr Container mit der Lambda-Ausführungsumgebung kompatibel ist, unabhängig von der Architektur des Entwicklungsrechners. Wenn Sie beabsichtigen, eine Lambda-Funktion mithilfe der ARM64 Befehlssatzarchitektur zu erstellen, müssen Sie den Befehl so ändern, dass er stattdessen die `--platform linux/arm64` Option verwendet.

(Optional) Testen Sie das Image lokal

1. Starten Sie Ihr Docker-Image mit dem docker run-Befehl. In diesem Beispiel ist `docker-image` der Image-Name und `test` der Tag.

   ```
   docker run --platform linux/amd64 -p 9000:8080 docker-image:test
   ```

   Dieser Befehl führt das Image als Container aus und erstellt einen lokalen Endpunkt bei `localhost:9000/2015-03-31/functions/function/invocations`.

   > **Note**
   >
   > Wenn Sie das Docker-Image für die ARM64 Befehlssatzarchitektur erstellt haben, stellen Sie sicher, dass Sie stattdessen die `--platform linux/arm64` Option verwenden. `--platform linux/amd64`

2. Veröffentlichen Sie in einem neuen Terminalfenster ein Ereignis an den lokalen Endpunkt.

   Linux/macOS

   Führen Sie unter Linux oder macOS den folgenden `curl`-Befehl aus:

   ```
   curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{}'
   ```

   Dieser Befehl ruft die Funktion mit einem leeren Ereignis auf und gibt eine Antwort zurück. Wenn Sie Ihren eigenen Funktionscode anstelle des Beispielfunktionscodes verwenden, wird empfohlen, die Funktion mit einer JSON-Nutzlast aufzurufen. Beispiel:

```
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d
 '{"payload":"hello world!"}'
```

PowerShell

Führen Sie in PowerShell den folgenden `Invoke-WebRequest` Befehl aus:

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{}' -ContentType "application/json"
```

Dieser Befehl ruft die Funktion mit einem leeren Ereignis auf und gibt eine Antwort zurück. Wenn Sie Ihren eigenen Funktionscode anstelle des Beispielfunktionscodes verwenden, wird empfohlen, die Funktion mit einer JSON-Nutzlast aufzurufen. Beispiel:

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{"payload":"hello world!"}' -ContentType
 "application/json"
```

3. Die Container-ID erhalten.

```
docker ps
```

4. Verwenden Sie den Befehl docker kill, um den Container zu anzuhalten. Ersetzen Sie in diesem Befehl `3766c4ab331c` durch die Container-ID aus dem vorherigen Schritt.

```
docker kill 3766c4ab331c
```

Das Image bereitstellen

Um das Image in Amazon ECR hochzuladen und die Lambda-Funktion zu erstellen

1. Führen Sie den get-login-passwordBefehl aus, um die Docker-CLI bei Ihrer Amazon ECR-Registrierung zu authentifizieren.

   - Setzen Sie den `--region` Wert auf den AWS-Region Ort, an dem Sie das Amazon ECR-Repository erstellen möchten.
   - Ersetzen Sie es `111122223333` durch Ihre AWS-Konto ID.

```
aws ecr get-login-password --region us-east-1 | docker login --username AWS --
password-stdin 111122223333.dkr.ecr.us-east-1.amazonaws.com
```

2. Erstellen Sie ein Repository in Amazon ECR mithilfe des Befehls create-repository.

```
aws ecr create-repository --repository-name hello-world --region us-east-1 --image-
scanning-configuration scanOnPush=true --image-tag-mutability MUTABLE
```

> **Note**
>
> Das Amazon ECR-Repository muss sich im selben Format AWS-Region wie die Lambda-Funktion befinden.

Wenn erfolgreich, sehen Sie eine Antwort wie diese:

```
{
    "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. Kopieren Sie das `repositoryUri` aus der Ausgabe im vorherigen Schritt.
4. Führen Sie den Befehl docker tag aus, um Ihr lokales Image als neueste Version in Ihrem Amazon-ECR-Repository zu markieren. In diesem Befehl gilt Folgendes:

- `docker-image:test` ist der Name und das Tag Ihres Docker-Images. Dies sind der Imagename und das Tag, die Sie im `docker build`-Befehl angegeben haben.
- Ersetzen Sie <ECRrepositoryUri> durch den `repositoryUri`, den Sie kopiert haben. Stellen Sie sicher, dass Sie `:latest` am Ende der URI angeben.

```
docker tag docker-image:test <ECRrepositoryUri>:latest
```

Beispiel:

```
docker tag docker-image:test 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

5. Führen Sie den Befehl docker push aus, um Ihr lokales Image im Amazon-ECR-Repository bereitzustellen. Stellen Sie sicher, dass Sie `:latest` am Ende der Repository-URI angeben.

```
docker push 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

6. Erstellen Sie eine Ausführungsrolle für die Funktion, wenn Sie noch keine haben. Sie benötigen den Amazon-Ressourcennamen (ARN) der Rolle im nächsten Schritt.
7. So erstellen Sie die Lambda-Funktion: Geben Sie für `ImageUri` die Repository-URI von zuvor an. Stellen Sie sicher, dass Sie `:latest` am Ende der URI angeben.

```
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**
>
> Sie können eine Funktion mit einem Bild in einem anderen AWS Konto erstellen, sofern sich das Bild in derselben Region wie die Lambda-Funktion befindet. Weitere Informationen finden Sie unter Kontoübergreifende Berechtigungen von Amazon ECR.

8. Die Funktion aufrufen.

```
aws lambda invoke --function-name hello-world response.json
```

Das Ergebnis sollte ungefähr wie folgt aussehen:

```
{
  "ExecutedVersion": "$LATEST",
  "StatusCode": 200
}
```

9. Um die Ausgabe der Funktion zu sehen, überprüfen Sie die `response.json`-Datei.

Um den Funktionscode zu aktualisieren, müssen Sie das Image erneut erstellen, das neue Image in das Amazon ECR-Repository hochladen und dann den update-function-codeBefehl verwenden, um das Image für die Lambda-Funktion bereitzustellen.

Lambda löst das Image-Tag in einen bestimmten Image-Digest auf. Das heißt, wenn Sie das Image-Tag, das zur Bereitstellung der Funktion verwendet wurde, auf ein neues Image in Amazon ECR verweisen, aktualisiert Lambda die Funktion nicht automatisch, um das neue Image zu verwenden.

Um das neue Image für dieselbe Lambda-Funktion bereitzustellen, müssen Sie den update-function-codeBefehl verwenden, auch wenn das Image-Tag in Amazon ECR gleich bleibt. Im folgenden Beispiel erstellt die `--publish`-Option eine neue Version der Funktion unter Verwendung des aktualisierten Container-Images.

```
aws lambda update-function-code \
  --function-name hello-world \
  --image-uri 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --publish
```

Verwenden eines alternativen Basis-Images mit dem Laufzeitschnittstellen-Client

Wenn Sie ein OS-Basis-Image oder ein alternatives Basis-Image verwenden, müssen Sie den Laufzeitschnittstellen-Client in das Image einbinden. Der Laufzeitschnittstellen-Client erweitert die Laufzeit-API, die die Interaktion zwischen Lambda und Ihrem Funktionscode verwaltet.

Installieren Sie den Lambda Runtime Interface Client für Ruby mit dem Paketmanager RubyGems .org:

```
gem install aws_lambda_ric
```

Sie können den Ruby-Runtime-Interface-Client auch von GitHub herunterladen.

Das folgende Beispiel zeigt, wie ein Container-Image für Ruby mithilfe eines AWS Nicht-Basis-Images erstellt wird. Das Beispiel-Dockerfile verwendet ein offizielles Ruby-Basis-Image. Das Docker-File enthält den Laufzeitschnittstellen-Client.

Voraussetzungen

Zur Durchführung der Schritte in diesem Abschnitt benötigen Sie Folgendes:

- AWS CLI Version 2
- Docker (Mindestversion 25.0.0)
- Das Docker-buildx-Plugin.
- Ruby

Erstellen eines Images aus einem alternativen Basis-Image

Erstellen eines Container-Images für Ruby mit einem alternativen Basis-Image

1. Erstellen Sie ein Verzeichnis für das Projekt und wechseln Sie dann zu diesem Verzeichnis.

   ```
   mkdir example
   cd example
   ```

2. Erstellen Sie eine neue Datei mit dem Namen `Gemfile`. Hier listen Sie die benötigten RubyGems Pakete für Ihre Anwendung auf. Das AWS SDK für Ruby ist erhältlich bei RubyGems. Sie sollten bestimmte AWS Service Gems für die Installation auswählen. Um beispielsweise das Ruby-Gem für Lambda zu verwenden, sollte Ihr Gemfile so aussehen:

   ```
   source 'https://rubygems.org'

   gem 'aws-sdk-lambda'
   ```

   Alternativ enthält das aws-sdk-Gem jedes verfügbare AWS Service-Gem. Dieses Gem ist sehr groß. Wir empfehlen, es nur zu verwenden, wenn Sie auf viele AWS Dienste angewiesen sind.
3. Installieren Sie die im Gemfile angegebenen Abhängigkeiten mithilfe von bundle install.

```
bundle install
```

4. Erstellen Sie eine neue Datei mit dem Namen `lambda_function.rb`. Sie können der Datei zum Testen den folgenden Beispielfunktionscode hinzufügen oder Ihren eigenen verwenden.

   Example Ruby-Funktion

   ```
   module LambdaFunction
     class Handler
       def self.process(event:,context:)
         "Hello from Lambda!"
       end
     end
   end
   ```

5. Erstellen Sie eine neue Docker-Datei. Das folgende Dockerfile verwendet ein Ruby-Basis-Image anstelle eines AWS -Basis-Images. Das Dockerfile enthält den Laufzeitschnittstellen-Client für Ruby, der das Image mit Lambda kompatibel macht. Alternativ können Sie den Laufzeitschnittstellen-Client zum Gemfile Ihrer Anwendung hinzufügen.

   - Legen Sie die `FROM`-Eigenschaft auf das Ruby-Basis-Image fest.
   - Erstellen Sie ein Verzeichnis für den Funktionscode und eine Umgebungsvariable, die auf dieses Verzeichnis verweist. In diesem Beispiel ist das Verzeichnis `/var/task`, das die Lambda-Ausführungsumgebung widerspiegelt. Sie können jedoch ein beliebiges Verzeichnis für den Funktionscode wählen, da das Dockerfile kein AWS Basis-Image verwendet.
   - Legen Sie `ENTRYPOINT` auf das Modul fest, das der Docker-Container beim Start ausführen soll. In diesem Fall ist das Modul der Laufzeitschnittstellen-Client.
   - Legen Sie das `CMD`-Argument auf den Lambda-Funktionshandler fest.

   Beachten Sie, dass das Dockerfile-Beispiel keine USER-Anweisung enthält. Wenn Sie ein Container-Image für Lambda bereitstellen, definiert Lambda automatisch einen Standard-Linux-Benutzer mit Berechtigungen mit geringsten Rechten. Dies unterscheidet sich vom Standardverhalten von Docker, bei dem standardmäßig der `root`-Benutzer verwendet wird, wenn keine `USER`-Anweisung bereitgestellt wird.

   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. Erstellen Sie Ihr Docker-Image mit dem docker build-Befehl. Das folgende Beispiel benennt das Bild in `docker-image` und gibt ihm den `test` Tag. Um Ihr Image mit Lambda kompatibel zu machen, müssen Sie die Option `--provenance=false` verwenden.

```
docker buildx build --platform linux/amd64 --provenance=false -t docker-image:test
 .
```

> **Note**
>
> Der Befehl gibt die `--platform linux/amd64`-Option an, um sicherzustellen, dass Ihr Container mit der Lambda-Ausführungsumgebung kompatibel ist, unabhängig von der Architektur des Entwicklungsrechners. Wenn Sie beabsichtigen, eine Lambda-

Funktion mithilfe der ARM64 Befehlssatzarchitektur zu erstellen, müssen Sie den Befehl so ändern, dass er stattdessen die `--platform linux/arm64` Option verwendet.

(Optional) Testen Sie das Image lokal

Verwenden Sie den Laufzeit-Schnittstellen-Emulator, um das Image lokal zu testen. Sie können den Emulator in Ihr Image integrieren oder ihn mit dem folgenden Verfahren auf Ihrem lokalen Computer installieren.

Installieren des Laufzeitschnittstellen-Emulators auf Ihrem lokalen Computer

1. Führen Sie in Ihrem Projektverzeichnis den folgenden Befehl aus, um den Runtime-Interface-Emulator (x86-64-Architektur) herunterzuladen GitHub und auf Ihrem lokalen Computer zu installieren.

   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
   ```

   Um den arm64-Emulator zu installieren, ersetzen Sie die GitHub Repository-URL im vorherigen Befehl durch Folgendes:

   ```
   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
   ```

Um den arm64-Emulator zu installieren, ersetzen Sie das `$downloadLink` durch Folgendes:

```
https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/
download/aws-lambda-rie-arm64
```

2. Starten Sie Ihr Docker-Image mit dem docker run-Befehl. Beachten Sie Folgendes:

   - `docker-image` ist der Image-Name und `test` ist das Tag.
   - `aws_lambda_ric lambda_function.LambdaFunction::Handler.process` ist der `ENTRYPOINT` gefolgt von dem `CMD` aus Ihrem 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
   ```

   Dieser Befehl führt das Image als Container aus und erstellt einen lokalen Endpunkt bei `localhost:9000/2015-03-31/functions/function/invocations`.

   > **Note**
   >
   > Wenn Sie das Docker-Image für die ARM64 Befehlssatzarchitektur erstellt haben, stellen Sie sicher, dass Sie die `--platform linux/arm64` Option anstelle von verwenden. `--platform linux/amd64`

3. Veröffentlichen Sie ein Ereignis auf dem lokalen Endpunkt.

Linux/macOS

Führen Sie unter Linux oder macOS den folgenden `curl`-Befehl aus:

```
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{}'
```

Dieser Befehl ruft die Funktion mit einem leeren Ereignis auf und gibt eine Antwort zurück. Wenn Sie Ihren eigenen Funktionscode anstelle des Beispielfunktionscodes verwenden, wird empfohlen, die Funktion mit einer JSON-Nutzlast aufzurufen. Beispiel:

```
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d
 '{"payload":"hello world!"}'
```

PowerShell

Führen Sie in PowerShell den folgenden `Invoke-WebRequest` Befehl aus:

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{}' -ContentType "application/json"
```

Dieser Befehl ruft die Funktion mit einem leeren Ereignis auf und gibt eine Antwort zurück. Wenn Sie Ihren eigenen Funktionscode anstelle des Beispielfunktionscodes verwenden, wird empfohlen, die Funktion mit einer JSON-Nutzlast aufzurufen. Beispiel:

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{"payload":"hello world!"}' -ContentType
 "application/json"
```

4. Die Container-ID erhalten.

```
docker ps
```

5. Verwenden Sie den Befehl docker kill, um den Container zu anzuhalten. Ersetzen Sie in diesem Befehl `3766c4ab331c` durch die Container-ID aus dem vorherigen Schritt.

```
docker kill 3766c4ab331c
```

Das Image bereitstellen

Um das Image in Amazon ECR hochzuladen und die Lambda-Funktion zu erstellen

1. Führen Sie den get-login-passwordBefehl aus, um die Docker-CLI bei Ihrer Amazon ECR-Registrierung zu authentifizieren.

   - Setzen Sie den `--region` Wert auf den AWS-Region Ort, an dem Sie das Amazon ECR-Repository erstellen möchten.
   - Ersetzen Sie es `111122223333` durch Ihre AWS-Konto ID.

   ```
   aws ecr get-login-password --region us-east-1 | docker login --username AWS --
   password-stdin 111122223333.dkr.ecr.us-east-1.amazonaws.com
   ```

2. Erstellen Sie ein Repository in Amazon ECR mithilfe des Befehls create-repository.

   ```
   aws ecr create-repository --repository-name hello-world --region us-east-1 --image-
   scanning-configuration scanOnPush=true --image-tag-mutability MUTABLE
   ```

   > **Note**
   >
   > Das Amazon ECR-Repository muss sich im selben Format AWS-Region wie die Lambda-Funktion befinden.

   Wenn erfolgreich, sehen Sie eine Antwort wie diese:

   ```
   {
       "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. Kopieren Sie das `repositoryUri` aus der Ausgabe im vorherigen Schritt.
4. Führen Sie den Befehl docker tag aus, um Ihr lokales Image als neueste Version in Ihrem Amazon-ECR-Repository zu markieren. In diesem Befehl gilt Folgendes:

   - `docker-image:test` ist der Name und das Tag Ihres Docker-Images. Dies sind der Imagename und das Tag, die Sie im `docker build`-Befehl angegeben haben.
   - Ersetzen Sie <ECRrepositoryUri> durch den `repositoryUri`, den Sie kopiert haben. Stellen Sie sicher, dass Sie `:latest` am Ende der URI angeben.

   ```
   docker tag docker-image:test <ECRrepositoryUri>:latest
   ```

   Beispiel:

   ```
   docker tag docker-image:test 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
   ```

5. Führen Sie den Befehl docker push aus, um Ihr lokales Image im Amazon-ECR-Repository bereitzustellen. Stellen Sie sicher, dass Sie `:latest` am Ende der Repository-URI angeben.

   ```
   docker push 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
   ```

6. Erstellen Sie eine Ausführungsrolle für die Funktion, wenn Sie noch keine haben. Sie benötigen den Amazon-Ressourcennamen (ARN) der Rolle im nächsten Schritt.
7. So erstellen Sie die Lambda-Funktion: Geben Sie für `ImageUri` die Repository-URI von zuvor an. Stellen Sie sicher, dass Sie `:latest` am Ende der URI angeben.

   ```
   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**
>
> Sie können eine Funktion mit einem Bild in einem anderen AWS Konto erstellen, sofern sich das Bild in derselben Region wie die Lambda-Funktion befindet. Weitere Informationen finden Sie unter Kontoübergreifende Berechtigungen von Amazon ECR.

8. Die Funktion aufrufen.

```
aws lambda invoke --function-name hello-world response.json
```

Das Ergebnis sollte ungefähr wie folgt aussehen:

```
{
  "ExecutedVersion": "$LATEST",
  "StatusCode": 200
}
```

9. Um die Ausgabe der Funktion zu sehen, überprüfen Sie die `response.json`-Datei.

Um den Funktionscode zu aktualisieren, müssen Sie das Image erneut erstellen, das neue Image in das Amazon ECR-Repository hochladen und dann den update-function-codeBefehl verwenden, um das Image für die Lambda-Funktion bereitzustellen.

Lambda löst das Image-Tag in einen bestimmten Image-Digest auf. Das heißt, wenn Sie das Image-Tag, das zur Bereitstellung der Funktion verwendet wurde, auf ein neues Image in Amazon ECR verweisen, aktualisiert Lambda die Funktion nicht automatisch, um das neue Image zu verwenden.

Um das neue Image für dieselbe Lambda-Funktion bereitzustellen, müssen Sie den update-function-codeBefehl verwenden, auch wenn das Image-Tag in Amazon ECR gleich bleibt. Im folgenden Beispiel erstellt die `--publish`-Option eine neue Version der Funktion unter Verwendung des aktualisierten Container-Images.

```
aws lambda update-function-code \
  --function-name hello-world \
  --image-uri 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --publish
```

Arbeiten mit Ebenen für Ruby Lambda-Funktionen

Verwenden Sie Lambda-Ebenen, um Code und Abhängigkeiten zu bündeln, die Sie für mehrere Funktionen wiederverwenden möchten. Ebenen enthalten üblicherweise Bibliotheksabhängigkeiten, eine benutzerdefinierte Laufzeit oder Konfigurationsdateien. Das Erstellen einer Ebene umfasst drei allgemeine Schritte:

1. Verpacken Ihres Ebeneninhalts. Dies bedeutet, dass Sie ein .zip-Dateiarchiv erstellen, das die Abhängigkeiten enthält, die Sie in Ihren Funktionen verwenden möchten.
2. Erstellen Sie die Ebene in Lambda.
3. Fügen Sie die Ebene zu Ihren Funktionen hinzu.

Themen

- Verpacken Ihres Ebeneninhalts
- Erstellen Sie die Ebene in Lambda
- Verwenden von Gems aus Ebenen in einer Funktion
- Fügen Sie die Ebene zu Ihrer Funktion hinzu
- Beispiel-App

Verpacken Ihres Ebeneninhalts

Um eine Ebene zu erstellen, bündeln Sie Ihre Pakete in einem ZIP-Dateiarchiv, das die folgenden Anforderungen erfüllt:

- Erstellen Sie die Ebene mithilfe derselben Version von Ruby, die Sie für die Lambda-Funktion verwenden möchten. Wenn Sie beispielsweise Ihre Ebene für Ruby 3.4 erstellen, verwenden Sie die Laufzeit Ruby 3.4 für Ihre Funktion.
- Die ZIP-Datei Ihrer Ebene muss eine der folgenden Verzeichnisstrukturen verwenden:
  - `ruby/gems/`*`x.x.x`* (wobei *`x.x.x`* Ihre Ruby-Version ist, z. B. `3.4.0`)
  - `ruby/lib`

  Weitere Informationen finden Sie unter Ebenenpfade für jede Lambda-Laufzeit.
- Die Pakete in Ihrer Ebene müssen mit Linux kompatibel sein. Lambda-Funktionen werden auf Amazon Linux ausgeführt.

Sie können Ebenen erstellen, die entweder Ruby-Gems von Drittanbietern oder Ihre eigenen Ruby-Module und -Klassen enthalten. Viele beliebte Ruby-Gems enthalten native Erweiterungen (C-Code), die für die Lambda-Linux-Umgebung kompiliert werden müssen.

Reine Ruby-Gems

Reine Ruby-Gems enthalten nur Ruby-Code und müssen nicht kompiliert werden. Diese Gems lassen sich einfacher verpacken und funktionieren plattformübergreifend.

So erstellen Sie eine Ebene mit reinen Ruby-Gems

1. Erstellen Sie eine `Gemfile`, um die reinen Ruby-Gems anzugeben, die Sie in Ihre Ebene aufnehmen möchten:

   Example Gemfile

   ```
   source 'https://rubygems.org'

   gem 'tzinfo'
   ```

2. Installieren Sie die Gems mit Bundler in einem `vendor/bundle`-Verzeichnis:

   ```
   bundle config set --local path vendor/bundle
   bundle install
   ```

3. Kopieren Sie die installierten Gems in die Verzeichnisstruktur, die Lambda benötigt (`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. Verpacken Sie den Ebeneninhalt:

   Linux/macOS

   ```
   zip -r layer.zip ruby/
   ```

   PowerShell

   ```
   Compress-Archive -Path .\ruby -DestinationPath .\layer.zip
   ```

Die Verzeichnisstruktur Ihrer ZIP-Datei sollte wie folgt aussehen:

```
ruby/
### gems/
    ### 3.4.0/
        ### gems/
        #   ### concurrent-ruby-1.3.5/
        #   ### tzinfo-2.0.6/
        ### specifications/
        ### cache/
        ### build_info/
        ### (other bundler directories)
```

> **Note**
>
> Sie müssen jede Gem einzeln in Ihrem Funktionscode als erforderlich festlegen. Sie können nicht `bundler/setup` oder `Bundler.require` verwenden. Weitere Informationen finden Sie unter Verwenden von Gems aus Ebenen in einer Funktion.

Gems mit nativen Erweiterungen

Viele beliebte Ruby-Gems enthalten native Erweiterungen (C-Code), die für die Zielplattform kompiliert werden müssen. Beliebte Gems mit nativen Erweiterungen sind unter anderem nokogiri, pg, mysql2, sqlite3 und ffi. Diese Gems müssen in einer Linux-Umgebung erstellt werden, die mit der Lambda-Laufzeit kompatibel ist.

So erstellen Sie eine Ebene mit Gems mit nativen Erweiterungen

1. Erstellen eines `Gemfile`.

   Example Gemfile

   ```
   source 'https://rubygems.org'

   gem 'nokogiri'
   gem 'httparty'
   ```

2. Verwenden Sie Docker, um die Gems in einer Linux-Umgebung zu erstellen, die mit Lambda kompatibel ist. Geben Sie ein AWS-Basisabbild in Ihrem Dockerfile an:

   Example Dockerfile für 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. Erstellen Sie das Bild und extrahieren Sie die Ebene:

```
docker build -t ruby-layer-builder .
docker run --rm -v $(pwd):/output --entrypoint cp ruby-layer-builder layer.zip /
output/
```

   Dadurch werden die Gems in der richtigen Linux-Umgebung erstellt und die `layer.zip`-Datei in Ihr lokales Verzeichnis kopiert. Die Verzeichnisstruktur Ihrer ZIP-Datei sollte wie folgt aussehen:

```
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**
>
> Sie müssen jede Gem einzeln in Ihrem Funktionscode als erforderlich festlegen. Sie können nicht `bundler/setup` oder `Bundler.require` verwenden. Weitere Informationen finden Sie unter Verwenden von Gems aus Ebenen in einer Funktion.

Benutzerdefinierte Ruby-Module

So erstellen Sie eine Ebene mithilfe Ihres eigenen Codes

1. Erstellen Sie die erforderliche Verzeichnisstruktur für Ihre Ebene:

   ```
   mkdir -p ruby/lib
   ```

2. Erstellen Sie Ihre Ruby-Module im `ruby/lib`-Verzeichnis. Das folgende Beispielmodul validiert Bestellungen, indem es überprüft, ob sie die erforderlichen Informationen enthalten.

   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. Verpacken Sie den Ebeneninhalt:

   Linux/macOS

   ```
   zip -r layer.zip ruby/
   ```

   PowerShell

   ```
   Compress-Archive -Path .\ruby -DestinationPath .\layer.zip
   ```

   Die Verzeichnisstruktur Ihrer ZIP-Datei sollte wie folgt aussehen:

   ```
   ruby/
   ### lib/
   ```

```
    ### order_validator.rb
```

4. Fordern Sie in Ihrer Funktion die Module an und verwenden Sie sie. Sie müssen jede Gem einzeln in Ihrem Funktionscode als erforderlich festlegen. Sie können nicht `bundler/setup` oder `Bundler.require` verwenden. Weitere Informationen finden Sie unter Verwenden von Gems aus Ebenen in einer Funktion. Beispiel:

```
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
```

Sie können das folgende Testereignis verwenden, um die Funktion aufzurufen:

```
{
    "body": "{\"product_id\": \"ABC123\", \"quantity\": 2, \"priority\": \"express
\"}"
}
```

Erwartete Antwort:

```
{
  "statusCode": 200,
  "body": "{\"message\":\"Order validated successfully\",\"order\":{\"product_id\":
\"ABC123\",\"quantity\":2,\"shipping_priority\":\"express\"}}"
}
```

Erstellen Sie die Ebene in Lambda

Sie können Ihre Ebene entweder über die AWS CLI oder die Lambda-Konsole veröffentlichen.

AWS CLI

Führen Sie den AWS CLI-Befehl publish-layer-version aus, um die Lambda-Ebene zu erstellen:

```
aws lambda publish-layer-version --layer-name my-layer --zip-file fileb://layer.zip
 --compatible-runtimes ruby3.4
```

Der Parameter Kompatible Laufzeiten ist optional. Wenn angegeben, verwendet Lambda diesen Parameter zum Filtern von Ebenen in der Lambda-Konsole.

Console

So erstellen Sie eine Ebene (Konsole)

1. Öffnen Sie die Seite Ebenen der Lambda-Konsole.
2. Wählen Sie Create Layer (Ebene erstellen) aus.
3. Wählen Sie ZIP-Datei hochladen und laden Sie dann das zuvor erstellte ZIP-Archiv hoch.
4. (Optional) Wählen Sie für Kompatible Laufzeiten die Ruby-Laufzeit aus, die der Ruby-Version entspricht, mit der Sie die Ebene erstellt haben.
5. Wählen Sie Erstellen aus.

Verwenden von Gems aus Ebenen in einer Funktion

In Ihrem Funktionscode müssen Sie jedes Gem, das Sie verwenden möchten, explizit als erforderlich festlegen. Bundler-Befehle wie `bundler/setup` und `Bundler.require` werden nicht unterstützt. So verwenden Sie Gems aus einer Ebene in einer Lambda-Funktion:

```
# 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
```

Fügen Sie die Ebene zu Ihrer Funktion hinzu

AWS CLI

Führen Sie den AWS CLI-Befehl update-function-configuration aus, um die Ebene Ihrer Funktion hinzuzufügen. Verwenden Sie den ARN der Ebene für den `--layers`-Parameter. Der ARN muss die Version angeben (z. B. `arn:aws:lambda:us-east-1:123456789012:layer:my-layer:1`). Weitere Informationen finden Sie unter Ebenen und Ebenenversionen.

```
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"
```

Die cli-binary-format-Option ist erforderlich, wenn Sie AWS CLI Version 2 verwenden. Um dies zur Standardeinstellung zu machen, führen Sie `aws configure set cli-binary-format raw-in-base64-out` aus. Weitere Informationen finden Sie unter Von AWS CLI unterstützte globale Befehlszeilenoptionen im AWS Command Line Interface-Benutzerhandbuch für Version 2.

Console

So fügen Sie eine Ebene zu einer Funktion hinzu

1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.
2. Wählen Sie die Funktion aus.
3. Blättern Sie nach unten bis zum Abschnitt Ebenen und wählen Sie dann Eine Ebene hinzufügen.

4. Wählen Sie unter Ebene auswählen die Option Benutzerdefinierte Ebenen und dann Ihre Ebene aus.

> **Note**
>
> Wenn Sie bei der Erstellung der Ebene keine kompatible Laufzeit hinzugefügt haben, wird die Ebene hier nicht aufgeführt. Sie können stattdessen den ARN der Ebene angeben.

5. Wählen Sie Hinzufügen aus.

Beispiel-App

Weitere Beispiele für die Verwendung von Lambda-Ebenen finden Sie in der Beispielanwendung layer-ruby im GitHub-Repository des AWS Lambda-Entwicklerhandbuchs. Diese Anwendung enthält eine Ebene, die die tzinfo-Bibliothek enthält. Nach dem Erstellen der Ebene können Sie die entsprechende Funktion bereitstellen und aufrufen, um zu bestätigen, dass die Ebene wie erwartet funktioniert.

Verwenden des Lambda-Kontextobjekts zum Abrufen von Ruby-Funktionsinformationen

Wenn Lambda Ihre Funktion ausführt, wird ein Context-Objekt an den Handler. übergeben. Dieses Objekt stellt Methoden und Eigenschaften mit Informationen zum Aufruf, zur Funktion und zur Ausführungsumgebung bereit.

Context-Methoden

- `get_remaining_time_in_millis` – Gibt die Anzahl der verbleibenden Millisekunden zurück, bevor die Ausführung das Zeitlimit überschreitet.

Context-Eigenschaften

- `function_name` – Der Name der Lambda-Funktion.
- `function_version` – Die Version der Funktion.
- `invoked_function_arn` – Der Amazon-Ressourcenname (ARN), der zum Aufrufen der Funktion verwendet wird. Gibt an, ob der Aufrufer eine Versionsnummer oder einen Alias angegeben hat.
- `memory_limit_in_mb` – Die Menge an Arbeitsspeicher, die der Funktion zugewiesen ist.
- `aws_request_id` – Der Bezeichner der Aufrufanforderung.
- `log_group_name` – Protokollgruppe für die Funktion.
- `log_stream_name` – Der Protokollstrom für die Funktionsinstance.
- `deadline_ms` – Das Datum, an dem eine Zeitüberschreitung für die Ausführung eintritt (in Unix-Millisekunden).
- `identity` – Informationen zur Amazon-Cognito-Identität, die die Anforderung autorisiert hat.
- `client_context` – (mobile Apps) Clientkontext, der Lambda von der Clientanwendung bereitgestellt wird.

Ruby-Lambda-Funktionen protokollieren und überwachen

AWS Lambda überwacht automatisch Lambda-Funktionen in Ihrem Namen und sendet Protokolle an Amazon CloudWatch. Ihre Lambda-Funktion enthält eine CloudWatch Logs-Log-Gruppe und einen Log-Stream für jede Instanz Ihrer Funktion. Die Lambda-Laufzeitumgebung sendet Details zu den einzelnen Aufrufen an den Protokollstream und leitet Protokolle und andere Ausgaben aus dem Code Ihrer Funktion weiter. Weitere Informationen finden Sie unter Senden von Lambda-Funktionsprotokollen an CloudWatch Logs.

Auf dieser Seite wird beschrieben, wie Sie eine Protokollausgabe aus dem Code Ihrer Lambda-Funktion erstellen und mithilfe der AWS Command Line Interface Lambda-Konsole oder der CloudWatch Konsole auf Logs zugreifen.

Sections

- Erstellen einer Funktion, die Protokolle zurückgibt
- Anzeigen von Protokollen in der Lambda-Konsole
- Protokolle in der Konsole anzeigen CloudWatch
- Logs mit dem Befehl () anzeigen AWS Command Line InterfaceAWS CLI
- Löschen von Protokollen
- Arbeiten mit der Ruby-Logger-Bibliothek

Erstellen einer Funktion, die Protokolle zurückgibt

Um Protokolle aus dem Code Ihrer Funktion auszugeben, können Sie `puts`-Anweisungen verwenden oder eine Protokollierungsbibliothek, die zu `stdout` oder `stderr` schreibt. Das folgende Beispiel protokolliert die Werte der Umgebungsvariablen und dem Ereignisobjekt.

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 Protokollformat

```
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
```

Die Ruby-Laufzeit protokolliert die Zeilen `START`, `END` und `REPORT` für jeden Aufruf. Die Berichtszeile enthält die folgenden Details.

Datenfelder für REPORT-Zeilen

- RequestId— Die eindeutige Anforderungs-ID für den Aufruf.
- Dauer – Die Zeit, die die Handler-Methode Ihrer Funktion mit der Verarbeitung des Ereignisses verbracht hat.
- Fakturierte Dauer – Die für den Aufruf fakturierte Zeit.
- Speichergröße – Die der Funktion zugewiesene Speichermenge.
- Max. verwendeter Speicher – Die Speichermenge, die von der Funktion verwendet wird. Wenn Aufrufe eine Ausführungsumgebung gemeinsam nutzen, meldet Lambda den maximalen Speicherverbrauch für alle Aufrufe. Dieses Verhalten kann zu einem höheren als erwarteten gemeldeten Wert führen.
- Initialisierungsdauer – Für die erste Anfrage die Zeit, die zur Laufzeit zum Laden der Funktion und Ausführen von Code außerhalb der Handler-Methode benötigt wurde.
- XRAY TraceId — Für verfolgte Anfragen die AWS X-Ray Trace-ID.
- SegmentId— Für verfolgte Anfragen die X-Ray-Segment-ID.
- Stichprobe – Bei verfolgten Anforderungen das Stichprobenergebnis.

Für detailliertere Protokolle verwenden Sie die the section called “Arbeiten mit der Ruby-Logger-Bibliothek”.

Anzeigen von Protokollen in der Lambda-Konsole

Sie können die Lambda-Konsole verwenden, um die Protokollausgabe nach dem Aufrufen einer Lambda-Funktion anzuzeigen.

Wenn Ihr Code über den eingebetteten Code-Editor getestet werden kann, finden Sie Protokolle in den Ausführungsergebnissen. Wenn Sie das Feature Konsolentest verwenden, um eine Funktion aufzurufen, finden Sie die Protokollausgabe im Abschnitt Details.

Protokolle in der Konsole anzeigen CloudWatch

Sie können die CloudWatch Amazon-Konsole verwenden, um Protokolle für alle Lambda-Funktionsaufrufe anzuzeigen.

Um Protokolle auf der Konsole anzuzeigen CloudWatch

1. Öffnen Sie die Seite Protokollgruppen auf der CloudWatch Konsole.
2. Wählen Sie die Protokollgruppe für Ihre Funktion (***your-function-name***/aws/lambda/).
3. Wählen Sie eine Protokollstream aus.

Jeder Protokoll-Stream entspricht einer Instance Ihrer Funktion. Ein Protokollstream wird angezeigt, wenn Sie Ihre Lambda-Funktion aktualisieren und wenn zusätzliche Instanzen für gleichzeitige Aufrufe erstellt werden. Um Logs für einen bestimmten Aufruf zu finden, empfehlen wir, Ihre Funktion mit zu instrumentieren. AWS X-Ray X-Ray erfasst Details zu der Anforderung und dem Protokollstream in der Trace.

Logs mit dem Befehl () anzeigen AWS Command Line InterfaceAWS CLI

Das AWS CLI ist ein Open-Source-Tool, mit dem Sie mithilfe von Befehlen in Ihrer Befehlszeilen-Shell mit AWS Diensten interagieren können. Zur Durchführung der Schritte in diesem Abschnitt benötigen Sie AWS CLI Version 2.

Sie können die AWS CLI verwenden, um Protokolle für einen Aufruf mit der `--log-type`-Befehlsoption abzurufen. Die Antwort enthält das Feld `LogResult`, das bis zu 4 KB base64-verschlüsselte Protokolle aus dem Aufruf enthält.

Example eine Log-ID abrufen

Das folgende Beispiel zeigt, wie eine Protokoll-ID aus dem `LogResult`-Feld für eine Funktion namens `my-function` abgerufen wird.

```
aws lambda invoke --function-name my-function out --log-type Tail
```

Die Ausgabe sollte folgendermaßen aussehen:

```
{
    "StatusCode": 200,
    "LogResult":
 "U1RBUlQgUmVxdWVzdElkOiA4N2QwNDRiOC1mMTU0LTExZTgtOGNkYS0yOTc0YzVlNGZiMjEgVmVyc2lvb...",
    "ExecutedVersion": "$LATEST"
}
```

Example entschlüsseln der Protokolle

Verwenden Sie in derselben Eingabeaufforderung das `base64`-Dienstprogramm, um die Protokolle zu entschlüsseln. Das folgende Beispiel zeigt, wie Base64-codierte Logs für abgerufen werde `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
```

Die cli-binary-format Option ist erforderlich, wenn Sie AWS CLI Version 2 verwenden. Um dies zur Standardeinstellung zu machen, führen Sie `aws configure set cli-binary-format raw-in-base64-out` aus. Weitere Informationen finden Sie unter Von AWS CLI unterstützte globale Befehlszeilenoptionen im AWS Command Line Interface -Benutzerhandbuch für Version 2.

Die Ausgabe sollte folgendermaßen aussehen:

```
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
```

Das `base64`-Dienstprogramm ist unter Linux, macOS und Ubuntu auf Windowsverfügbar. macOS-Benutzer müssen möglicherweise `base64 -D` verwenden.

Example get-logs.sh-Skript

Verwenden Sie in derselben Eingabeaufforderung das folgende Skript, um die letzten fünf Protokollereignisse herunterzuladen. Das Skript verwendet `sed` zum Entfernen von Anführungszeichen aus der Ausgabedatei und wechselt 15 Sekunden lang in den Ruhezustand, um Zeit einzuräumen, damit Protokolle verfügbar werden können. Die Ausgabe enthält die Antwort von Lambda und die `get-log-events`Ausgabe des Befehls.

Kopieren Sie den Inhalt des folgenden Codebeispiels und speichern Sie es in Ihrem Lambda-Projektverzeichnis unter `get-logs.sh`.

Die cli-binary-format Option ist erforderlich, wenn Sie AWS CLI Version 2 verwenden. Um dies zur Standardeinstellung zu machen, führen Sie `aws configure set cli-binary-format raw-in-base64-out` aus. Weitere Informationen finden Sie unter Von AWS CLI unterstützte globale Befehlszeilenoptionen im AWS Command Line Interface -Benutzerhandbuch für 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 und Linux (nur diese Systeme)

In derselben Eingabeaufforderung müssen macOS- und Linux-Benutzer möglicherweise den folgenden Befehl ausführen, um sicherzustellen, dass das Skript ausführbar ist.

```
chmod -R 755 get-logs.sh
```

Example die letzten fünf Protokollereignisse abrufen

Führen Sie an derselben Eingabeaufforderung das folgende Skript aus, um die letzten fünf Protokollereignisse abzurufen.

```
./get-logs.sh
```

Die Ausgabe sollte folgendermaßen aussehen:

```
{
```

```
    "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"
}
```

Löschen von Protokollen

Wenn Sie eine Funktion löschen, werden Protokollgruppen nicht automatisch gelöscht. Um das unbegrenzte Speichern von Protokollen zu vermeiden, löschen Sie die Protokollgruppe oder konfigurieren Sie eine Aufbewahrungszeitraum nach dem Protokolle automatisch gelöscht werden.

Arbeiten mit der Ruby-Logger-Bibliothek

Die Ruby-Logger-Bibliothek gibt optimierte Protokolle zurück, die leicht zu lesen sind. Verwenden Sie das Logger-Serviceprogramm, um detaillierte Informationen, Meldungen und Fehlercodes im Zusammenhang mit Ihrer Funktion auszugeben.

```
# 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
```

Die Ausgabe aus `logger` umfasst die Protokollebene, den Zeitstempel und die Anforderungs-ID.

```
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
```

Instrumentieren von Ruby-Code in AWS Lambda

Lambda ist mit AWS X-Ray integriert, um Ihnen zu ermöglichen, Lambda-Anwendungen zu verfolgen, zu debuggen und zu optimieren. Sie können mit X-Ray eine Anforderung verfolgen, während sie Ressourcen in Ihrer Anwendung durchläuft, von der Frontend-API bis hin zu Speicher und Datenbank im Backend. Indem Sie einfach die X-Ray-SDK-Bibliothek zu Ihrer Build-Konfiguration hinzufügen, können Sie Fehler und Latenz für jeden Aufruf aufzeichnen, den Ihre Funktion an einen AWS-Service durchführt.

Nachdem Sie die aktive Nachverfolgung konfiguriert haben, können Sie bestimmte Anfragen über Ihre Anwendung beobachten. Das X-Ray-Service-Diagramm zeigt Informationen über Ihre Anwendung und alle ihre Komponenten an. Das folgende Beispiel zeigt eine Anwendung mit zwei Funktionen. Die primäre Funktion verarbeitet Ereignisse und gibt manchmal Fehler zurück. Die zweite Funktion ganz oben verarbeitet Fehler, die in der Protokollgruppe der ersten Protokollgruppe erscheinen, und verwendet das AWS-SDK, um X-Ray, Amazon Simple Storage Service (Amazon S3) und Amazon CloudWatch Logs aufzurufen.

Gehen Sie folgendermaßen vor, um die aktive Nachverfolgung Ihrer Lambda-Funktion mit der Konsole umzuschalten:

So aktivieren Sie die aktive Nachverfolgung

1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.
2. Wählen Sie eine Funktion aus.

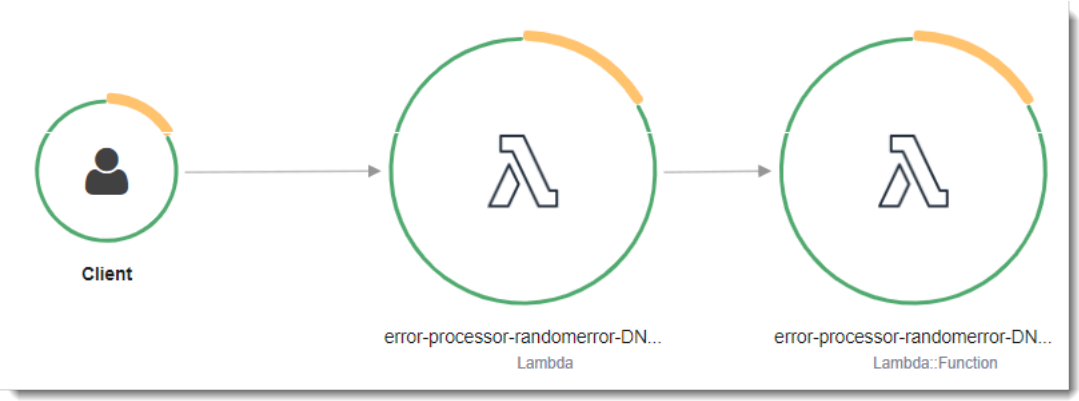
3. Wählen Sie Configuration (Konfiguration) und dann Monitoring and operations tools (Überwachungs- und Produktionstools).
4. Wählen Sie unter Zusätzliche Überwachungstools die Option Bearbeiten aus.
5. Wählen Sie unter CloudWatch Application Signals und AWS X-Ray für Lambda-Service-Nachverfolgungen die Option Aktivieren aus.
6. Wählen Sie Speichern.

> **Preisgestaltung**
>
> Sie können X-Ray-Nachverfolgung jeden Monat bis zu einem bestimmten Limit als Teil des AWS kostenlosen Kontingents kostenlos nutzen. Über den Schwellenwert hinaus berechnet X-Ray Gebühren für die Speicherung und den Abruf der Nachverfolgung. Weitere Informationen finden Sie unter AWS X-Ray Preise.

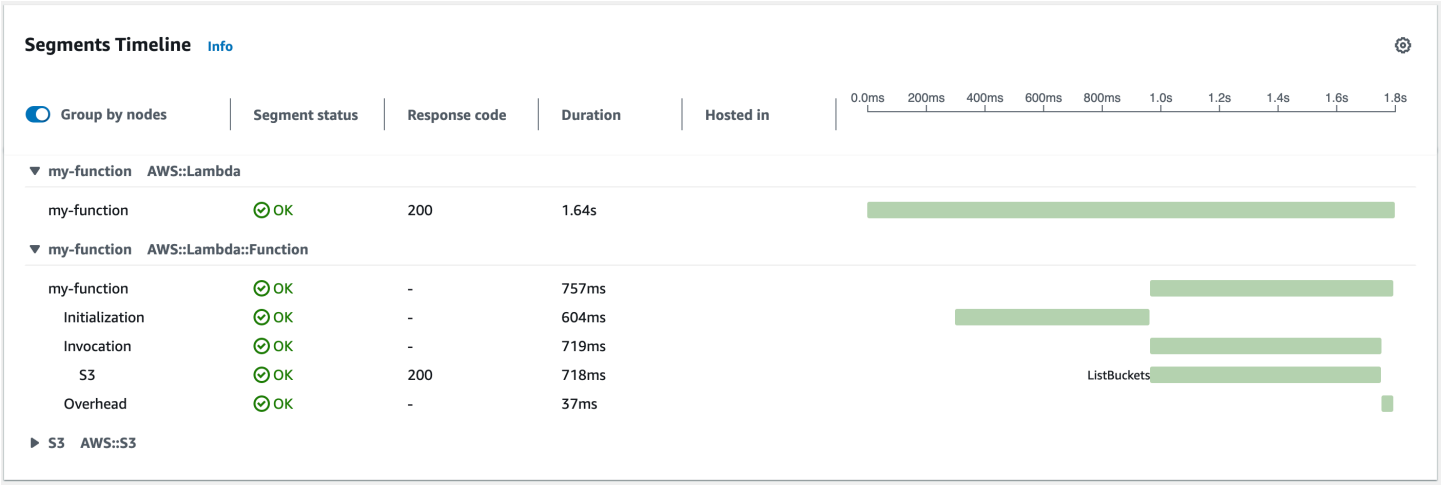
Ihre Funktion benötigt die Berechtigung zum Hochladen von Trace-Daten zu X-Ray. Wenn Sie die aktive Nachverfolgung in der Lambda-Konsole aktivieren, fügt Lambda der Ausführungsrolle Ihrer Funktion die erforderlichen Berechtigungen hinzu. Andernfalls fügen Sie der Ausführungsrolle die AWSXRayDaemonWriteAccess-Richtlinie hinzu.

X-Ray verfolgt nicht alle Anfragen an Ihre Anwendung nach. X-Ray wendet einen Sampling-Algorithmus an, um sicherzustellen, dass die Nachverfolgung effizient ist, und stellt dennoch ein repräsentatives Beispiel aller Anfragen bereit. Die Samplingrate beträgt 1 Anforderung pro Sekunde und 5 Prozent aller weiteren Anforderungen. Sie können die X-Ray-Samplingrate nicht für Ihre Funktionen konfigurieren.

In X-Ray, zeichnet eine Ablaufverfolgung Informationen zu einer Anforderung auf, die von einem oder mehreren Services verarbeitet wird. Lambda zeichnet 2 Segmente pro Trace auf, wodurch zwei Knoten im Dienstgraphen entstehen. In der folgenden Abbildung werden diese beiden Knoten hervorgehoben:



Der erste Knoten auf der linken Seite stellt den Lambda-Service dar, der die Aufrufanforderung empfängt. Der zweite Knoten stellt Ihre spezifische Lambda-Funktion dar. Das folgende Beispiel zeigt eine Nachverfolgung mit diesen zwei Segmenten. Beide heißen my-function, aber einer hat einen Ursprung von `AWS::Lambda` und der andere hat einen Ursprung von `AWS::Lambda::Function`. Wenn das `AWS::Lambda`-Segment einen Fehler anzeigt, hatte der Lambda-Service ein Problem. Wenn das `AWS::Lambda::Function`-Segment einen Fehler anzeigt, ist bei Ihrer Funktion ein Problem aufgetreten.



Dieses Beispiel erweitert das `AWS::Lambda::Function`-Segment, um seine drei Untersegmente anzuzeigen.

> **Note**
>
> AWS implementiert derzeit Änderungen am Lambda-Service. Aufgrund dieser Änderungen können geringfügige Unterschiede in Struktur und Inhalt der Systemprotokollmeldungen und Trace-Segmente auftreten, die von verschiedenen Lambda-Funktionen in Ihrem AWS-Konto.

Der hier gezeigte Beispiel-Trace veranschaulicht das Funktionssegment im alten Stil. Die Unterschiede zwischen den Segmenten im alten und im neuen Stil werden in den folgenden Abschnitten beschrieben.
Diese Änderungen werden in den kommenden Wochen implementiert, und alle Funktionen in allen AWS-Regionen-Regionen mit Ausnahme von China und GovCloud werden auf das neue Format der Protokollmeldungen und Trace-Segmente umgestellt.

Das Funktionssegment im alten Stil enthält die folgenden Untersegmente:

- Initialisierung – Stellt die Zeit dar, die für das Laden Ihrer Funktion und das Ausführen des Initialisierungscodes aufgewendet wurde. Dieses Untersegment erscheint nur für das erste Ereignis, das jede Instance Ihrer Funktion verarbeitet.
- Invocation (Aufruf) – Stellt die Zeit dar, die beim Ausführen Ihres Handler-Codes vergeht.
- Overhead (Aufwand) – Stellt die Zeit dar, die von der Lambda-Laufzeitumgebung bei der Verarbeitung des nächsten Ereignisses verbraucht wird.

Das Funktionssegment im neuen Stil enthält kein `Invocation`-Untersegment. Stattdessen werden Kundenuntersegmente direkt an das Funktionssegment angehängt. Weitere Informationen über die Struktur der alten und neuen Funktionssegmente finden Sie unter the section called “Grundlegendes zu X-Ray-Ablaufverfolgungen”.

Sie können Ihren Handler-Code instrumentieren, um Metadaten aufzuzeichnen und Downstream-Aufrufe nachzuverfolgen. Um Details zu Aufrufen aufzuzeichnen, die Ihr Handler an andere Ressourcen und Services macht, verwenden Sie das X-Ray-SDK für Ruby. Um das SDK zu erhalten, fügen Sie das `aws-xray-sdk`-Paket den Abhängigkeiten Ihrer Anwendung hinzu.

Example blank-ruby/function/Gemfile

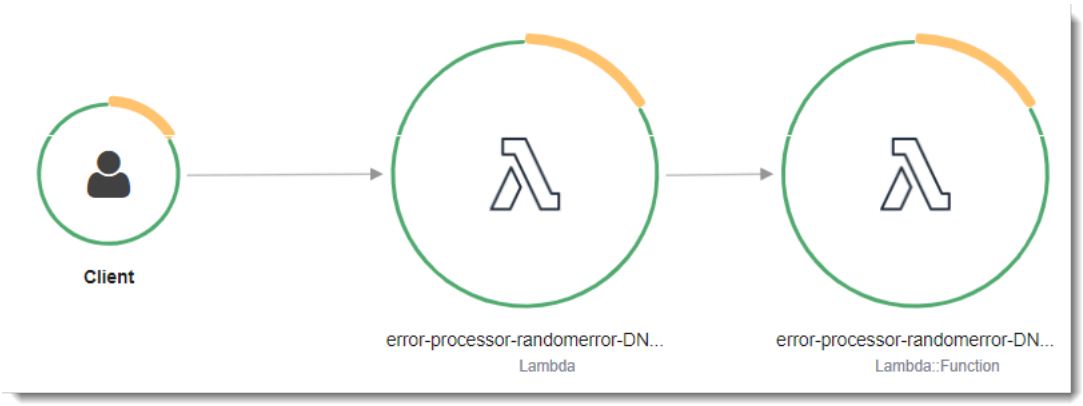
```
# Gemfile
source 'https://rubygems.org'

gem 'aws-xray-sdk', '0.11.4'
gem 'aws-sdk-lambda', '1.39.0'
gem 'test-unit', '3.3.5'
```

Um AWS SDK-Clients zu instrumentieren, benötigen Sie das `aws-xray-sdk/lambda`-Modul, nachdem Sie einen Client im Initialisierungscode erstellt haben.

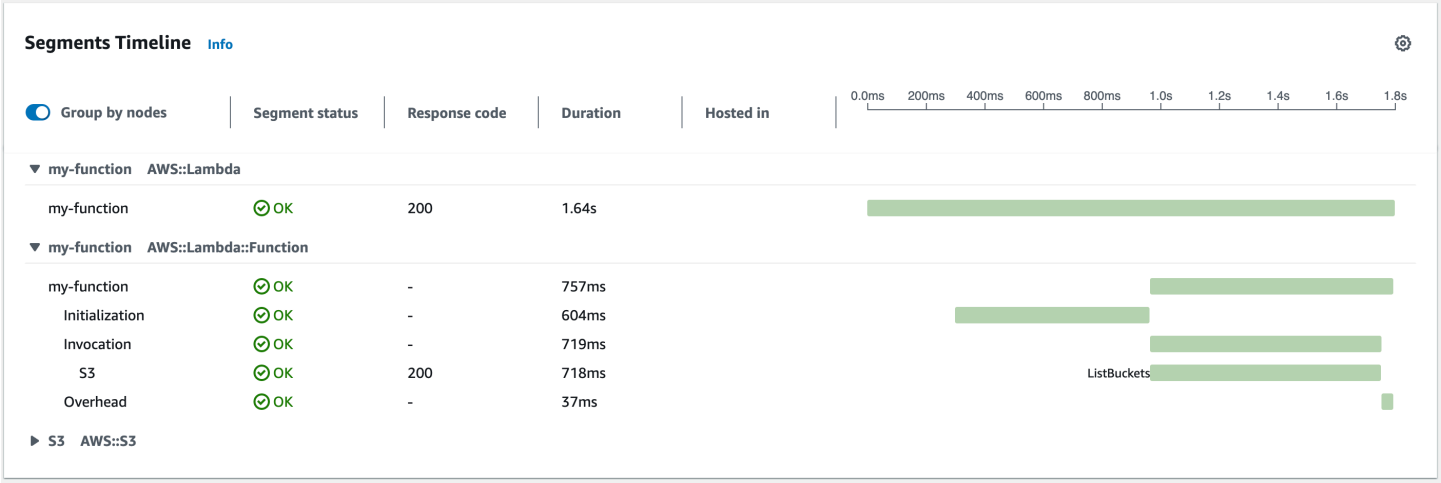
Example blank-ruby/function/lambda_function.rb – Nachverfolgung eines AWS SDK-Clients

```
# 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)
  ...
```

In X-Ray, zeichnet eine Ablaufverfolgung Informationen zu einer Anforderung auf, die von einem oder mehreren Services verarbeitet wird. Lambda zeichnet 2 Segmente pro Trace auf, wodurch zwei Knoten im Dienstgraphen entstehen. In der folgenden Abbildung werden diese beiden Knoten hervorgehoben:



Der erste Knoten auf der linken Seite stellt den Lambda-Service dar, der die Aufrufanforderung empfängt. Der zweite Knoten stellt Ihre spezifische Lambda-Funktion dar. Das folgende Beispiel zeigt eine Nachverfolgung mit diesen zwei Segmenten. Beide heißen my-function, aber einer hat einen Ursprung von `AWS::Lambda` und der andere hat einen Ursprung von `AWS::Lambda::Function`. Wenn das `AWS::Lambda`-Segment einen Fehler anzeigt, hatte der Lambda-Service ein Problem. Wenn das `AWS::Lambda::Function`-Segment einen Fehler anzeigt, ist bei Ihrer Funktion ein Problem aufgetreten.



Dieses Beispiel erweitert das `AWS::Lambda::Function`-Segment, um seine drei Untersegmente anzuzeigen.

> **Note**
>
> AWS implementiert derzeit Änderungen am Lambda-Service. Aufgrund dieser Änderungen können geringfügige Unterschiede in Struktur und Inhalt der Systemprotokollmeldungen und Trace-Segmente auftreten, die von verschiedenen Lambda-Funktionen in Ihrem AWS-Konto. Der hier gezeigte Beispiel-Trace veranschaulicht das Funktionssegment im alten Stil. Die Unterschiede zwischen den Segmenten im alten und im neuen Stil werden in den folgenden Abschnitten beschrieben.
>
> Diese Änderungen werden in den kommenden Wochen implementiert, und alle Funktionen in allen AWS-Regionen-Regionen mit Ausnahme von China und GovCloud werden auf das neue Format der Protokollmeldungen und Trace-Segmente umgestellt.

Das Funktionssegment im alten Stil enthält die folgenden Untersegmente:

- Initialisierung – Stellt die Zeit dar, die für das Laden Ihrer Funktion und das Ausführen des Initialisierungscodes aufgewendet wurde. Dieses Untersegment erscheint nur für das erste Ereignis, das jede Instance Ihrer Funktion verarbeitet.
- Invocation (Aufruf) – Stellt die Zeit dar, die beim Ausführen Ihres Handler-Codes vergeht.
- Overhead (Aufwand) – Stellt die Zeit dar, die von der Lambda-Laufzeitumgebung bei der Verarbeitung des nächsten Ereignisses verbraucht wird.

Das Funktionssegment im neuen Stil enthält kein `Invocation`-Untersegment. Stattdessen werden Kundenuntersegmente direkt an das Funktionssegment angehängt. Weitere Informationen über die Struktur der alten und neuen Funktionssegmente finden Sie unter the section called “Grundlegendes zu X-Ray-Ablaufverfolgungen”.

Sie können auch HTTP-Clients instrumentieren, SQL-Abfragen aufzeichnen und benutzerdefinierte Untersegmente mit Anmerkungen und Metadaten erstellen. Weitere Informationen finden Sie unter Das X-Ray-SDK für Ruby im AWS X-Ray-Entwicklerhandbuch.

Abschnitte

- Aktivieren der aktiven Ablaufverfolgung mit der Lambda-API
- Aktivieren von aktiver Ablaufverfolgung mit CloudFormation
- Speichern von Laufzeitabhängigkeiten in einer Ebene

Aktivieren der aktiven Ablaufverfolgung mit der Lambda-API

Verwenden Sie die folgenden API-Operationen, um die Protokollierungskonfiguration mit der AWS CLI oder dem AWS-SDK zu verwalten:

- UpdateFunctionConfiguration
- GetFunctionConfiguration
- CreateFunction

Der folgende AWS CLI-Beispielbefehl aktiviert die aktive Nachverfolgung für eine Funktion mit dem Namen my-function.

```
aws lambda update-function-configuration --function-name my-function \
--tracing-config Mode=Active
```

Der Ablaufverfolgungsmodus ist Teil der versionsspezifischen Konfiguration, wenn Sie eine Version Ihrer Funktion veröffentlichen. Sie können den Ablaufverfolgungsmodus für eine veröffentlichte Version nicht ändern.

Aktivieren von aktiver Ablaufverfolgung mit CloudFormation

Verwenden Sie die `AWS::Lambda::Function`-Eigenschaft, um die Nachverfolgung für eine CloudFormation-Ressource in einer `TracingConfig`-Vorlage zu aktivieren.

Example function-inline.yml – Ablaufverfolgungskonfiguration

```
Resources:
  function:
    Type: AWS::Lambda::Function
    Properties:
      TracingConfig:
        Mode: Active
      ...
```

Verwenden Sie für eine AWS Serverless Application Model (AWS SAM) `AWS::Serverless::Function`-Ressource die Eigenschaft `Tracing`.

Example template.yml – Ablaufverfolgungskonfiguration

```
Resources:
  function:
    Type: AWS::Serverless::Function
    Properties:
      Tracing: Active
      ...
```

Speichern von Laufzeitabhängigkeiten in einer Ebene

Wenn Sie das X-Ray-SDK verwenden, um mit AWS-SDK-Clients Ihren Funktionscode zu instrumentieren, kann Ihr Bereitstellungspaket ziemlich groß werden. Um Laufzeitabhängigkeiten bei jeder Aktualisierung des Funktionscodes zu vermeiden, verpacken Sie das X-Ray-SDK in einer Lambda-Ebene.

Das folgende Beispiel zeigt eine `AWS::Serverless::LayerVersion`-Ressource, die das X-Ray-SDK für Ruby speichert.

Example template.yml – Abhängigkeitenebene

```
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
```

Bei dieser Konfiguration aktualisieren Sie die Bibliotheksebene nur, wenn Sie Ihre Laufzeitabhängigkeiten ändern. Da das Funktionsbereitstellungspaket nur Ihren Code enthält, kann dies dazu beitragen, die Upload-Zeiten zu reduzieren.

Das Erstellen einer Ebene für Abhängigkeiten erfordert Build-Konfigurationsänderungen, um das Ebenen-Archiv vor der Bereitstellung zu generieren. Ein funktionierendes Beispiel finden Sie in der Beispielanwendung blank-ruby .

Erstellen von Lambda-Funktionen mit Java

Sie können Java-Code in AWS Lambda ausführen. Lambda bietet Laufzeiten für Java, die Ihren Code ausführen, um Ereignisse zu verarbeiten. Ihr Code wird in einer Amazon Linux-Umgebung ausgeführt, die AWS Anmeldeinformationen von einer AWS Identity and Access Management (IAM-) Rolle enthält, die Sie verwalten.

Lambda unterstützt die folgenden Java-Laufzeiten.

Name	ID	Betriebssystem	Datum der Veraltung	Blockfunktion erstellen	Blockfunktion aktualisieren
Java 2.5	`java25`	Amazon Linux 2023	30. Juni 2029	31. Juli 2029	31. August 2029
Java 21	`java21`	Amazon Linux 2023	30. Juni 2029	31. Juli 2029	31. August 2029
Java 17	`java17`	Amazon Linux 2	30. Juni 2027	31. Juli 2027	31. August 2027
Java 11	`java11`	Amazon Linux 2	30. Juni 2027	31. Juli 2027	31. August 2027
Java 8	`java8.al2`	Amazon Linux 2	30. Juni 2027	31. Juli 2027	31. August 2027

AWS stellt die folgenden Bibliotheken für Java-Funktionen bereit. Diese Bibliotheken sind über Maven Central Repositoryverfügbar.

- com.amazonaws: aws-lambda-java-core (erforderlich) — Definiert Handler-Methodenschnittstellen und das Kontextobjekt, das die Runtime an den Handler übergibt. Wenn Sie eigene Eingabetypen definieren, benötigen Sie nur diese eine Bibliothek.
- com.amazonaws: aws-lambda-java-events — Eingabetypen für Ereignisse von Diensten, die Lambda-Funktionen aufrufen.
- com.amazonaws: aws-lambda-java-log 4j2 — Eine Appender-Bibliothek für Apache Log4j 2, mit der Sie die Anforderungs-ID für den aktuellen Aufruf zu Ihren Funktionsprotokollen hinzufügen können.

- AWS SDK for Java 2.0 — Das offizielle AWS SDK für die Programmiersprache Java.

Fügen Sie die Bibliotheken wie folgt der Build-Definition hinzu.

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

Verwenden Sie keine privaten Komponenten der JDK API, wie private Felder, Methoden oder Klassen. Nicht öffentliche API-Komponenten können sich bei jedem Update ändern oder entfernt werden, was dazu führen kann, dass Ihre Anwendung nicht mehr funktioniert.

So erstellen Sie eine Java-Funktion

1. Öffnen Sie die Lambda-Konsole.
2. Wählen Sie Funktion erstellen.
3. Konfigurieren Sie die folgenden Einstellungen:
   - Funktionsname: Geben Sie einen Namen für die Funktion ein.
   - Runtime: Wählen Sie Java 25.
4. Wählen Sie Funktion erstellen.

Die Konsole erstellt eine Lambda-Funktion mit einer Handler-Klasse namens `Hello`. Da es sich bei Java um eine kompilierte Sprache handelt, können Sie den Quellcode in der Lambda-Konsole nicht anzeigen oder bearbeiten, aber Sie können seine Konfiguration ändern, ihn aufrufen und Auslöser konfigurieren.

> **Note**
>
> Um mit der Anwendungsentwicklung in Ihrer lokalen Umgebung zu beginnen, stellen Sie eine der Beispielanwendungen bereit, die im GitHub Repository dieses Handbuchs verfügbar sind.

Die `Hello`-Klasse hat eine Funktion mit dem Namen `handleRequest`, die ein Ereignisobjekt und ein Kontext-Objekt übernimmt. Dies ist die Handler-Funktion, die bei einem Aufruf der Funktion von Lambda aufgerufen wird. Die Java-Funktionslaufzeit ruft Aufrufereignisse von Lambda ab und leitet sie an den Handler weiter. In der Konfiguration der Funktion lautet der Wert für den Handler `example.Hello::handleRequest`.

Um den Funktionscode zu aktualisieren, erstellen Sie ein Bereitstellungspaket, d. h. ein ZIP-Datei-Archiv handelt, das Ihren Funktionscode enthält. Mit fortschreitender Funktionserstellung sollten Sie Ihren Funktionscode in der Versionskontrolle speichern, Bibliotheken hinzufügen und Bereitstellungen automatisieren. Beginnen Sie mit der Erstellung eines Bereitstellungspakets und der Aktualisierung Ihres Codes in der Befehlszeile.

Die Funktionslaufzeit übergibt neben dem Aufrufereignis ein Context-Objekt an den Handler. Das Context-Objekt enthält zusätzliche Informationen zum Aufruf, zur Funktion und zur Ausführungsumgebung. Weitere Informationen erhalten Sie über die Umgebungsvariablen.

Ihre Lambda-Funktion wird mit einer CloudWatch Logs-Protokollgruppe geliefert. Die Funktionslaufzeit sendet Details zu jedem Aufruf an CloudWatch Logs. Es leitet alle Protokolle weiter, die Ihre Funktion während des Aufrufs ausgibt. Wenn Ihre Funktion einen Fehler zurückgibt, formatiert Lambda den Fehler und gibt ihn an den Aufrufer zurück.

Themen

- Definieren des Lambda-Funktions-Handlers in Java
- Bereitstellen von Java-Lambda-Funktionen mit ZIP- oder JAR-Dateiarchiven
- Bereitstellen von Java-Lambda-Funktionen mit Container-Images
- Arbeiten mit Ebenen für Java Lambda-Funktionen
- Passen Sie die Serialisierung für Lambda-Java-Funktionen an
- Passen Sie das Startverhalten der Java-Laufzeit für Lambda-Funktionen an
- Verwenden des Lambda-Kontextobjekts zum Abrufen von Java-Funktionsinformationen
- Java-Lambda-Funktionen protokollieren und überwachen
- Instrumentierung von Java-Code in AWS Lambda
- Java-Beispielanwendungen für AWS Lambda

Definieren des Lambda-Funktions-Handlers in Java

Der Lambda-Funktionshandler ist die Methode in Ihrem Funktionscode, die Ereignisse verarbeitet. Wenn Ihre Funktion aufgerufen wird, führt Lambda die Handler-Methode aus. Ihre Funktion wird so lange ausgeführt, bis der Handler eine Antwort zurückgibt, beendet wird oder ein Timeout auftritt.

Auf dieser Seite wird beschrieben, wie Sie mit Lambda-Funktions-Handlern in Java arbeiten, einschließlich Optionen für die Projekteinrichtung, Benennungskonventionen und Best Practices. Diese Seite enthält auch ein Beispiel für eine Java-Lambda-Funktion, die Informationen über einen Auftrag aufnimmt, eine Textdatei als Quittung erstellt und diese Datei in einem Bucket von Amazon Simple Storage Service (Amazon S3) platziert. Informationen darüber, wie Sie Ihre Funktion nach dem Schreiben einsetzen können, finden Sie unter the section called “Bereitstellen von ZIP-Dateiarchiven” oder the section called “Bereitstellen von Container-Images”.

Abschnitte

- Einrichten Ihres Java-Handler-Projekts
- Beispiel für Java-Lambda-Funktionscode
- Gültige Klassendefinitionen für Java-Handler
- Namenskonventionen für Handler
- Definieren Sie das Eingabeereignisobjekt und greifen Sie darauf zu
- Zugreifen auf und Verwenden des Lambda-Kontextobjekts
- Verwenden des AWS SDK für Java v2 in Ihrem Handler
- Zugriff auf Umgebungsvariablen
- Verwenden des globalen Zustands
- Bewährte Codemethoden für Java-Lambda-Funktionen

Einrichten Ihres Java-Handler-Projekts

Wenn Sie mit Lambda-Funktionen in Java arbeiten, umfasst der Prozess das Schreiben Ihres Codes, das Kompilieren und das Bereitstellen der kompilierten Artefakte in Lambda. Sie können ein Java-Lambda-Projekt auf verschiedene Arten initialisieren. Sie können beispielsweise Tools wie Maven Archetype für Lambda-Funktionen, den AWS SAM-CLI-Befehl sam init oder sogar eine Standard-Java-Projekteinrichtung in Ihrer bevorzugten IDE wie IntelliJ IDEA oder Visual Studio Code verwenden. Alternativ können Sie die erforderliche Dateistruktur manuell erstellen.

Ein typisches Java-Lambda-Funktionsprojekt folgt dieser allgemeinen Struktur:

```
/project-root
    # src
        # main
            # java
                # example
                    # OrderHandler.java (contains main handler)
                    # <other_supporting_classes>
     # build.gradle OR pom.xml
```

Sie können entweder Maven oder Gradle verwenden, um Ihr Projekt zu erstellen und Abhängigkeiten zu verwalten.

Die Haupthandler-Logik für Ihre Funktion befindet sich in einer Java-Datei im Verzeichnis `src/main/java/example`. Im Beispiel auf dieser Seite nennen wir diese Datei `OrderHandler.java`. Neben dieser Datei können Sie bei Bedarf weitere Java-Klassen hinzufügen. Achten Sie beim Bereitstellen Ihrer Funktion in Lambda darauf, die Java-Klasse anzugeben, die die Haupthandler-Methode enthält, die Lambda bei einem Aufruf ausführen soll.

Beispiel für Java-Lambda-Funktionscode

Das folgende Beispiel für einen Java 21-Lambda-Funktionscode nimmt Informationen über eine Bestellung auf, erstellt eine Textdateiquittung und platziert diese Datei in einem Amazon-S3-Bucket.

Example **`OrderHandler.java`**-Lambda-Funktion

```
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);
        }
    }
}
```

Diese `OrderHandler.java`-Datei enthält die folgenden Abschnitte des Codes:

- `package example`: In Java kann dies beliebig sein, muss jedoch mit der Verzeichnisstruktur Ihres Projekts übereinstimmen. Hier verwenden wir `package example`, da die Verzeichnisstruktur `src/main/java/example` lautet.
- `import`-Anweisungen: Damit können Sie Java-Klassen importieren, die für Ihre Lambda-Funktion erforderlich sind.
- `public class OrderHandler ...`: Dies definiert Ihre Java-Klasse und muss eine gültige Klassendefinition sein.
- `private static final S3Client S3_CLIENT ...`: Dadurch wird ein S3-Client außerhalb einer der Methoden der Klasse initialisiert. Dies veranlasst Lambda, diesen Code während der Initialisierungsphase auszuführen.
- `public record Order ...`: Definieren Sie die Form des erwarteten Eingabeereignisses in diesem benutzerdefinierten Java-Datensatz.
- `public String handleRequest(Order event, Context context)`: Dies ist die Haupthandler-Methode, die Ihre Hauptanwendungslogik enthält.
- `private void uploadReceiptToS3(...) {}`: Dies ist eine Hilfsmethode, auf die von der `handleRequest`-Haupthandler-Methode verwiesen wird.

Beispieldateien „build.gradle“ und „pom.xml“

Die folgende `build.gradle`- oder `pom.xml`-Datei gehört zu dieser Funktion.

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>
```

Damit diese Funktion ordnungsgemäß funktioniert, muss ihre Ausführungsrolle die `s3:PutObject`-Aktion zulassen. Stellen Sie außerdem sicher, dass Sie die `RECEIPT_BUCKET`-Umgebungsvariable definieren. Nach einem erfolgreichen Aufruf sollte der Amazon-S3-Bucket eine Empfangsdatei enthalten.

> **Note**
>
> Diese Funktion erfordert möglicherweise zusätzliche Konfigurationseinstellungen, um ohne Zeitüberschreitung erfolgreich ausgeführt zu werden. Wir empfehlen, 256 MB Arbeitsspeicher und eine Zeitüberschreitung von 10 Sekunden zu konfigurieren. Der erste Aufruf kann aufgrund eines Kaltstarts zusätzliche Zeit in Anspruch nehmen. Nachfolgende Aufrufe sollten aufgrund der Wiederverwendung der Ausführungsumgebung wesentlich schneller ausgeführt werden.

Gültige Klassendefinitionen für Java-Handler

Zum Definieren Ihrer Klasse definiert die Bibliothek aws-lambda-java-core zwei Schnittstellen für Handler-Methoden. Verwenden Sie die bereitgestellten Schnittstellen, um die Handler-Konfiguration zu vereinfachen und die Methodensignatur zur Kompilierzeit zu validieren.

- com.amazonaws.services.lambda.runtime.RequestHandler
- com.amazonaws.services.lambda.runtime.RequestStreamHandler

Die `RequestHandler`-Schnittstelle ist ein generischer Typ, der zwei Parameter verwendet: den Eingabetyp und den Ausgabetyp. Beide Typen müssen Objekte sein. In diesem Beispiel implementiert unsere `OrderHandler`-Klasse `RequestHandler<OrderHandler.Order, String>`. Der Eingabetyp ist der `Order`-Datensatz, den wir innerhalb der Klasse definieren, und der Ausgabetyp ist `String`.

```
public class OrderHandler implements RequestHandler<OrderHandler.Order, String> {
    ...
}
```

Wenn Sie diese Schnittstelle verwenden, deserialisiert die Java-Laufzeit das Ereignis in das Objekt mit dem Eingabetyp und serialisiert die Ausgabe in Text. Verwenden Sie diese Schnittstelle, wenn die integrierte Serialisierung mit Ihren Ein- und Ausgabetypen funktioniert.

Um Ihre eigene Serialisierung zu verwenden, können Sie die `RequestStreamHandler`-Schnittstelle implementieren. Mit dieser Schnittstelle übergibt Lamda Ihrem Handler einen Eingabestream und einen Ausgabestream. Der Handler liest Bytes aus dem Eingabestream, schreibt in den Ausgabestream und gibt „void“ zurück. Ein Beispiel für die Verwendung der Java 21-Laufzeit finden Sie unter HandlerStream.java.

Wenn Sie in Ihrer Java-Funktion nur mit grundlegenden und generischen Typen (d. h. `String`, `Integer`, `List` oder `Map`) arbeiten, müssen Sie keine Schnittstelle implementieren. Wenn Ihre Funktion beispielsweise eine `Map<String, String>`-Eingabe akzeptiert und eine `String` zurückgibt, könnten Ihre Klassendefinition und Ihre Handler-Signatur wie folgt aussehen:

```
public class ExampleHandler {
    public String handleRequest(Map<String, String> input, Context context) {
        ...
    }
}
```

Wenn Sie keine Schnittstelle implementieren, ist das Kontextobjekt außerdem optional. Ihre Klassendefinition und Ihre Handler-Signatur könnten beispielsweise wie folgt aussehen:

```
public class NoContextHandler {
   public String handleRequest(Map<String, String> input) {
        ...
   }
}
```

Namenskonventionen für Handler

Wenn Sie für Lambda-Funktionen in Java entweder die `RequestHandler`- oder `RequestStreamHandler`-Schnittstelle implementieren, muss Ihre Haupthandler-Methode `handleRequest` heißen. Fügen Sie außerdem das Tag `@Override` über Ihrer `handleRequest`-Methode ein. Wenn Sie Ihre Funktion in Lambda bereitstellen, geben Sie den Haupthandler in der Konfiguration Ihrer Funktion im folgenden Format an:

- *<Paket>*.*<Klasse>* – Beispiel: `example.OrderHandler`.

Für Lambda-Funktionen in Java, die nicht die `RequestHandler`- oder `RequestStreamHandler`-Schnittstelle implementieren, können Sie einen beliebigen Namen für den Handler verwenden. Wenn Sie Ihre Funktion in Lambda bereitstellen, geben Sie den Haupthandler in der Konfiguration Ihrer Funktion im folgenden Format an:

- *<Paket>*.*<Klasse>*::*<Name_der_Handlermethode>* – Beispiel: `example.Handler::mainHandler`.

Definieren Sie das Eingabeereignisobjekt und greifen Sie darauf zu

JSON ist das gebräuchlichste und standardmäßigste Eingabeformat für Lambda-Funktionen. In diesem Beispiel erwartet die Funktion eine Eingabe ähnlich der folgenden:

```
{
    "orderId": "12345",
    "amount": 199.99,
    "item": "Wireless Headphones"
}
```

Wenn Sie mit Lambda-Funktionen in Java 17 oder neuer arbeiten, können Sie die Form des erwarteten Eingabeereignisses als Java-Datensatz definieren. In diesem Beispiel definieren wir einen Datensatz innerhalb der `OrderHandler`-Klasse, um ein `Order`-Objekt darzustellen:

```
public record Order(String orderId, double amount, String item) {}
```

Dieser Datensatz entspricht der erwarteten Eingabeform. Nachdem Sie Ihren Datensatz definiert haben, können Sie eine Handler-Signatur schreiben, die eine JSON-Eingabe entgegennimmt, die der Datensatzdefinition entspricht. Die Java-Laufzeit deserialisiert diese JSON-Eingabe automatisch

in ein Java-Objekt. Sie können dann auf die Felder des Objekts zugreifen. Zum Beispiel holt `event.orderId` den Wert von `orderId` aus der ursprünglichen Eingabe.

> **Note**
>
> Java-Datensätze sind nur in Java 17-Laufzeiten und neueren Versionen enthalten. In allen Java-Laufzeiten können Sie eine Klasse verwenden, um Ereignisdaten darzustellen. In solchen Fällen können Sie eine Bibliothek wie jackson verwenden, um JSON-Eingaben zu deserialisieren.

Andere Typen von Eingabeereignissen

Es gibt viele mögliche Eingabeereignisse für Lambda-Funktionen in Java:

- `Integer`, `Long`, `Double`, usw. – Das Ereignis ist eine Zahl ohne zusätzliche Formatierung, zum Beispiel: ., `3.5`. Die Java-Laufzeit wandelt den Wert in ein Objekt des angegebenen Typs um.
- `String` – Das Ereignis ist eine JSON-Zeichenfolge mit Anführungszeichen, z. B, `“My string”`. Die Laufzeit wandelt den Wert in ein `String`-Objekt ohne Anführungszeichen um.
- `List<Integer>`, `List<String>`, `List<Object>`, usw. – Das Ereignis ist ein JSON-Array. Die Laufzeitumgebung deserialisiert sie in ein Objekt des angegebenen Typs oder der angegebenen Schnittstelle.
- `InputStream` – Das Ereignis ist ein beliebiger JSON-Typ. Die Laufzeit übergibt einen Bytestream des Dokuments ohne Änderung an den Handler. Sie deserialisieren die Eingabe und schreiben Ausgabe in einen Ausgabestream.
- Bibliothekstyp – Bei Ereignissen, die von anderen AWS-Services gesendet werden, müssen Sie die Typen in der Bibliothek aws-lambda-java-events verwenden. Wenn Ihre Lambda-Funktion beispielsweise von Amazon Simple Queue Service (SQS) aufgerufen wird, verwenden Sie das `SQSEvent`-Objekt als Eingabe.

Zugreifen auf und Verwenden des Lambda-Kontextobjekts

Das Lambda-Kontextobjekt enthält Informationen über Aufruf, Funktion und Ausführungsumgebung. In diesem Beispiel ist das Kontextobjekt vom Typ `com.amazonaws.services.lambda.runtime.Context` und das zweite Argument der Haupthandlerfunktion.

```
public String handleRequest(Order event, Context context) {
    ...
}
```

Wenn Ihre Klasse entweder die RequestHandler oder die RequestStreamHandler-Schnittstelle implementiert, ist das Kontextobjekt ein erforderliches Argument. Andernfalls ist das Kontextobjekt optional. Weitere Informationen zu gültigen akzeptierten Handlersignaturen finden Sie unter the section called “Gültige Klassendefinitionen für Java-Handler”.

Wenn Sie mit dem AWS-SDK andere Dienste aufrufen, wird das Kontextobjekt in einigen wichtigen Bereichen benötigt. Um beispielsweise Funktionsprotokolle für Amazon CloudWatch zu erstellen, können Sie die `context.getLogger()`-Methode verwenden, um ein `LambdaLogger`-Objekt für die Protokollierung abzurufen. In diesem Beispiel können wir den Logger verwenden, um eine Fehlermeldung zu protokollieren, falls bei der Verarbeitung aus irgendeinem Grund ein Fehler auftritt:

```
context.getLogger().log("Failed to process order: " + e.getMessage());
```

Außerhalb der Protokollierung können Sie das Kontextobjekt auch für die Funktionsüberwachung verwenden. Weitere Informationen über das Kontextobjekt finden Sie unter the section called “Context”.

Verwenden des AWS SDK für Java v2 in Ihrem Handler

Oft verwenden Sie Lambda-Funktionen, um mit anderen AWS-Ressourcen zu interagieren oder diese zu aktualisieren. Die einfachste Art, eine Schnittstelle zu diesen Ressourcen herzustellen, ist die Verwendung des AWS SDK für Java v2.

> **Note**
>
> Das AWS SDK für Java (v1) befindet sich im Wartungsmodus und der Support wird am 31. Dezember 2025 eingestellt. Wir empfehlen, künftig nur noch das AWS SDK für Java v2 zu verwenden.

Um SDK-Abhängigkeiten zu Ihrer Funktion hinzuzufügen, fügen Sie diese in Ihrer `build.gradle`-Datei für Gradle oder der `pom.xml`-Datei für Maven hinzu. Wir empfehlen, nur die Bibliotheken hinzuzufügen, die Sie für Ihre Funktion benötigen. Im Beispielcode zuvor haben wir die Bibliothek `software.amazon.awssdk.services.s3` verwendet. In Gradle können Sie diese Abhängigkeit

hinzufügen, indem Sie die folgende Zeile in den Abhängigkeitenabschnitt von `build.gradle` einfügen:

```
implementation 'software.amazon.awssdk:s3:2.28.29'
```

Fügen Sie in Maven die folgenden Zeilen im `<dependencies>`-Abschnitt von `pom.xml` hinzu:

```
    <dependency>
        <groupId>software.amazon.awssdk</groupId>
        <artifactId>s3</artifactId>
        <version>2.28.29</version>
    </dependency>
```

> **Note**
>
> Dies ist möglicherweise nicht die neueste Version des SDK. Wählen Sie die geeignete SDK-Version für Ihre Anwendung.

Importieren Sie dann die Abhängigkeiten direkt in Ihree Java-Klasse:

```
import software.amazon.awssdk.services.s3.S3Client;
import software.amazon.awssdk.services.s3.model.PutObjectRequest;
import software.amazon.awssdk.services.s3.model.S3Exception;
```

Der Beispielcode initialisiert dann einen Amazon-S3-Client wie folgt:

```
private static final S3Client S3_CLIENT = S3Client.builder().build();
```

In diesem Beispiel haben wir unseren Amazon-S3-Client außerhalb der Haupt-Handler-Funktion initialisiert, um zu vermeiden, dass wir ihn bei jedem Aufruf unserer Funktion initialisieren müssen. Nachdem Sie Ihren SDK-Client initialisiert haben, können Sie ihn für die Interaktion mit anderen AWS-Services verwenden. Der Beispielcode ruft die Amazon S3 `PutObject`-API wie folgt auf:

```
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)));
```

Zugriff auf Umgebungsvariablen

In Ihrem Handler-Code können Sie mithilfe der `System.getenv()`-Methode auf beliebige Umgebungsvariablen verweisen. In diesem Beispiel verweisen wir mit der folgenden Codezeile auf die definierte `RECEIPT_BUCKET`-Umgebungsvariable:

```
String bucketName = System.getenv("RECEIPT_BUCKET");
if (bucketName == null || bucketName.isEmpty()) {
    throw new IllegalArgumentException("RECEIPT_BUCKET environment variable is not
 set");
}
```

Verwenden des globalen Zustands

Lambda führt Ihren statischen Code und den Klassenkonstruktor während der Initialisierungsphase aus, bevor Ihre Funktion zum ersten Mal aufgerufen wird. Ressourcen, die während der Initialisierung erstellt werden, bleiben zwischen Aufrufen im Speicher, sodass Sie sie nicht bei jedem Aufruf Ihrer Funktion neu erstellen müssen.

Im Beispielcode befindet sich der Initialisierungscode des S3-Clients außerhalb der Haupt-Handler-Methode. Die Laufzeit initialisiert den Client, bevor die Funktion ihr erstes Ereignis ausführt, und der Client bleibt für die Wiederverwendung bei allen Aufrufen verfügbar.

Bewährte Codemethoden für Java-Lambda-Funktionen

Halten Sie sich an die Richtlinien in der folgenden Liste, um beim Erstellen Ihrer Lambda-Funktionen die besten Codierungspraktiken anzuwenden:

- Trennen Sie den Lambda-Handler von Ihrer Core-Logik. Auf diese Weise können Sie eine Funktion zur besseren Prüfbarkeit von Einheiten schaffen.
- Kontrollieren Sie die Abhängigkeiten im Bereitstellungspaket Ihrer Funktion. Die AWS Lambda-Ausführungsumgebung enthält eine Reihe von Bibliotheken. Um die neuesten Funktionen und Sicherheitsupdates zu aktivieren, wird Lambda diese Bibliotheken regelmäßig aktualisieren. Diese Updates können das Verhalten Ihrer Lambda-Funktion geringfügig verändern. Um die Abhängigkeiten, die Ihre Funktion verwendet, vollständig zu kontrollieren, empfehlen wir, alle Abhängigkeiten mit Ihrem Bereitstellungspaket zu bündeln.

- Minimieren Sie die Komplexität Ihrer Abhängigkeiten. Ziehen Sie einfachere Frameworks vor, die sich schnell beim Start der Ausführungsumgebung laden lassen. Verwenden Sie beispielsweise einfachere Java Dependency Injection (IoC) Frameworks wie z. B. Dagger or Guice, als komplexere wie Spring Framework.
- Minimieren Sie die Größe Ihres Bereitstellungspakets auf die für die Laufzeit erforderliche Größe. Dadurch verkürzt sich die Zeit, die für das Herunterladen und Entpacken Ihres Bereitstellungspakets vor dem Aufruf benötigt wird. Für Funktionen, die in Java erstellt wurden, sollten Sie das Hochladen der gesamten AWS SDK-Bibliothek als Teil Ihres Bereitstellungspakets vermeiden. Stattdessen sollten Sie selektiv von den Modulen ausgehen, die Komponenten des SDK aufnehmen, die Sie benötigen (z. B. DynamoDB, Amazon-S3-SDK-Module und Lambda-Kernbibliotheken).

Nutzen Sie die Wiederverwendung der Ausführungsumgebung zur Verbesserung Ihrer Funktion. Initialisieren Sie SDK-Clients und Datenbankverbindungen außerhalb des Funktions-Handlers und speichern Sie statische Komponenten lokal im `/tmp`-Verzeichnis. Nachfolgende Aufrufe, die von derselben Instance Ihrer Funktion verarbeitet werden, können diese Ressourcen wiederverwenden. Dies spart Kosten durch Reduzierung der Funktionslaufzeit.

Um potenzielle Datenlecks über Aufrufe hinweg zu vermeiden, verwenden Sie die Ausführungsumgebung nicht, um Benutzerdaten, Ereignisse oder andere Informationen mit Sicherheitsauswirkungen zu speichern. Wenn Ihre Funktion auf einem veränderbaren Zustand beruht, der nicht im Speicher innerhalb des Handlers gespeichert werden kann, sollten Sie für jeden Benutzer eine separate Funktion oder separate Versionen einer Funktion erstellen.

Verwenden Sie eine Keep-Alive-Direktive, um dauerhafte Verbindungen zu pflegen. Lambda bereinigt Leerlaufverbindungen im Laufe der Zeit. Der Versuch, eine Leerlaufverbindung beim Aufruf einer Funktion wiederzuverwenden, führt zu einem Verbindungsfehler. Um Ihre persistente Verbindung aufrechtzuerhalten, verwenden Sie die Keep-Alive-Direktive, die Ihrer Laufzeit zugeordnet ist. Ein Beispiel finden Sie unter Wiederverwenden von Verbindungen mit Keep-Alive in Node.js.

Verwenden Sie Umgebungsvariablen um Betriebsparameter an Ihre Funktion zu übergeben. Wenn Sie z. B. Daten in einen Amazon-S3-Bucket schreiben, anstatt den Bucket-Namen, in den Sie schreiben, hartzucodieren, konfigurieren Sie den Bucket-Namen als Umgebungsvariable.

Vermeiden Sie rekursive Aufrufe in Ihrer Lambda-Funktion, bei denen die Funktion sich selbst aufruft oder einen Prozess initiiert, der die Funktion erneut aufrufen kann. Dies kann zu unvorhergesehenen Mengen an Funktionsaufrufen führen und höhere Kosten zur Folge haben. Wenn Sie eine unbeabsichtigte Menge von Aufrufen feststellen, legen Sie die reservierte gleichzeitige Ausführung

der Funktion auf `0` fest, um sofort alle Aufrufe der Funktion zu drosseln, während Sie den Code aktualisieren.

Verwenden Sie keine nicht dokumentierten, nicht öffentlichen APIs in Ihrem Lambda-Funktionscode. Für AWS Lambda-verwaltete Laufzeiten wendet Lambda regelmäßig Sicherheits- und Funktionsupdates auf Lambdas interne APIs an. Diese internen API-Updates können abwärtskompatibel sein, was zu unbeabsichtigten Konsequenzen wie Aufruffehlern führt, wenn Ihre Funktion von diesen nicht öffentlichen APIs abhängig ist. Eine Liste öffentlich zugänglicher APIs finden Sie in der API-Referenz.

Schreiben Sie idempotenten Code. Das Schreiben idempotenter Code für Ihre Funktionen stellt sicher, dass doppelte Ereignisse auf die gleiche Weise behandelt werden. Ihr Code sollte Ereignisse ordnungsgemäß validieren und doppelte Ereignisse ordnungsgemäß behandeln. Weitere Informationen finden Sie unter Wie mache ich meine Lambda-Funktion idempotent?.

- Vermeiden der Verwendung von Java-DNS-Cache. Lambda-Funktionen speichern DNS-Antworten bereits im Cache. Wenn Sie einen anderen DNS-Cache verwenden, kann es zu Verbindungs-Timeouts kommen.

  Die `java.util.logging.Logger`-Klasse kann indirekt den JVM-DNS-Cache aktivieren. Um die Standardeinstellungen zu überschreiben, setzen Sie networkaddress.cache.ttl auf 0, bevor Sie `logger` initialisieren. Beispiel:

  ```
  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);
  }
  ```

- Reduzieren Sie den Zeitaufwand, den Lambda für das Entpacken von Bereitstellungspaketen benötigt, die in Java erstellt wurden, indem Sie Ihre Abhängigkeitsdateien `.jar` in einem separaten /lib-Verzeichnis ablegen. Dies geht schneller, als den gesamten Code Ihrer Funktion in einem einzigen Jar mit einer großen Anzahl von `.class` Dateien zu speichern. Detaillierte Anweisungen finden Sie unter Bereitstellen von Java-Lambda-Funktionen mit ZIP- oder JAR-Dateiarchiven.

Bereitstellen von Java-Lambda-Funktionen mit ZIP- oder JAR-Dateiarchiven

Der Code Ihrer AWS Lambda Funktion besteht aus Skripten oder kompilierten Programmen und deren Abhängigkeiten. Sie verwenden ein Bereitstellungspaket, um Ihren Funktionscode in Lambda bereitzustellen. Lambda unterstützt zwei Arten von Bereitstellungspaketen: Container-Images und ZIP-Dateiarchiven.

Auf dieser Seite wird beschrieben, wie Sie Ihr Bereitstellungspaket als ZIP-Datei oder Jar-Datei erstellen und dann das Bereitstellungspaket verwenden, um Ihren Funktionscode AWS Lambda mithilfe von AWS Command Line Interface (AWS CLI) bereitzustellen.

> **Important**
>
> In Java 25 wurde die Unterstützung für Ahead-of-Time (AOT-) Caches eingeführt. Wir empfehlen dringend, keine AOT-Caches zu verwenden, wenn Sie Ihre Funktionen als.zip- oder JAR-Dateiarchive bereitstellen, da die Caches zu unerwartetem Verhalten führen können, wenn Lambda die verwaltete Laufzeit aktualisiert. Weitere Informationen finden Sie unter. Ahead-of-Time (AOT) und CDS-Caches

Sections

- Voraussetzungen
- Tools und Bibliotheken
- Erstellen eines Bereitstellungspakets mit Gradle
- Verwenden von Ebenen für Abhängigkeiten
- Erstellen eines Bereitstellungspakets mit Maven
- Hochladen eines Bereitstellungspakets mit der Lambda-Konsole
- Hochladen eines Bereitstellungspakets mit dem AWS CLI
- Hochladen eines Bereitstellungspakets mit AWS SAM

Voraussetzungen

Das AWS CLI ist ein Open-Source-Tool, mit dem Sie mithilfe von Befehlen in Ihrer Befehlszeilen-Shell mit AWS Diensten interagieren können. Zur Durchführung der Schritte in diesem Abschnitt benötigen Sie AWS CLI Version 2.

Tools und Bibliotheken

AWS stellt die folgenden Bibliotheken für Java-Funktionen bereit. Diese Bibliotheken sind über Maven Central Repositoryverfügbar.

- com.amazonaws: aws-lambda-java-core (erforderlich) — Definiert Handler-Methodenschnittstellen und das Kontextobjekt, das die Runtime an den Handler übergibt. Wenn Sie eigene Eingabetypen definieren, benötigen Sie nur diese eine Bibliothek.
- com.amazonaws: aws-lambda-java-events — Eingabetypen für Ereignisse von Diensten, die Lambda-Funktionen aufrufen.
- com.amazonaws: aws-lambda-java-log 4j2 — Eine Appender-Bibliothek für Apache Log4j 2, mit der Sie die Anforderungs-ID für den aktuellen Aufruf zu Ihren Funktionsprotokollen hinzufügen können.
- AWS SDK for Java 2.0 — Das offizielle AWS SDK für die Programmiersprache Java.

Fügen Sie die Bibliotheken wie folgt der Build-Definition hinzu.

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>
```

Um ein Bereitstellungspaket zu erstellen, kompilieren Sie den Funktionscode und die Abhängigkeiten in einer einzelnen ZIP- oder Java-Archivdatei (JAR). Verwenden Sie für Gradle den `Zip`-Build-Typ. Verwenden Sie für Apache Maven das Maven Shade Plugin. Verwenden Sie zum Hochladen Ihres Bereitstellungspakets die Lambda-Konsole, die Lambda-API oder AWS Serverless Application Model ()AWS SAM.

> **Note**
>
> Um den Umfang des Bereitstellungspakets gering zu halten, verpacken Sie die Abhängigkeiten Ihrer Funktion in Ebenen. Ebenen ermöglichen Ihnen, Ihre Abhängigkeiten unabhängig zu verwalten, können von mehreren Funktionen genutzt werden und können für andere Konten zur gemeinsamen Nutzung freigegeben werden. Weitere Informationen finden Sie unter Lambda-Ebenen.

Erstellen eines Bereitstellungspakets mit Gradle

Um in Gradle ein Bereitstellungspaket mit dem Code Ihrer Funktion und deren Abhängigkeiten zu erstellen, verwenden Sie den Build-Typ `Zip`. Hier ist ein Beispiel aus einer vollständigen build.gradle Datei:

Example build.gradle – Entwicklungs-Aufgabe

```
task buildZip(type: Zip) {
    into('lib') {
        from(jar)
        from(configurations.runtimeClasspath)
    }
```

```
}
```

Diese Build-Konfiguration erzeugt ein Bereitstellungspaket im `build/distributions`-Ordner. Innerhalb der `into('lib')`-Anweisung stellt die `jar`-Aufgabe ein JAR-Archiv mit Ihren Hauptklassen in einem Ordner mit dem Namen „`lib`“ zusammen. Die `configurations.runtimeClassPath`-Aufgabe kopiert außerdem Abhängigkeitsbibliotheken aus dem Klassenpfad des Builds in denselben Ordner mit dem Namen „`lib`“.

Example build.gradle – Abhängigkeiten

```
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 lädt JAR-Dateien in alphabetischer Reihenfolge (Unicode). Wenn mehrere JAR-Dateien im `lib`-Verzeichnis die gleiche Klasse enthalten, wird die erste verwendet. Sie können das folgende Shell-Skript zum Identifizieren von doppelten Klassen verwenden.

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
```

Verwenden von Ebenen für Abhängigkeiten

Sie können die Abhängigkeiten Ihrer Funktion in Ebenen packen, um Ihr Bereitstellungspaket klein zu halten und Abhängigkeiten unabhängig zu verwalten. Weitere Informationen finden Sie unter the section called “Layer”.

Erstellen eines Bereitstellungspakets mit Maven

Verwenden Sie das Maven Shade-Plugin, um ein Bereitstellungspaket mit Maven zu erstellen. Das Plugin erstellt eine JAR-Datei mit dem kompilierten Funktionscode und allen seinen Abhängigkeiten.

Example pom.xml – Plugin-Konfiguration

```
      <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>
```

Verwenden Sie den `mvn package`-Befehl, um das Bereitstellungspaket zu erstellen.

```
[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] ------------------------------------------------------------------------
```

Mit diesem Befehl wird eine JAR-Datei im `target`-Verzeichnis generiert.

> **Note**
>
> Wenn Sie mit einem Multi-Release-JAR (MRJAR) arbeiten, müssen Sie das MRJAR (d.h. das vom Maven Shade-Plugin erzeugte shaded JAR) in das `lib`-Verzeichnis aufnehmen und es zippen, bevor Sie Ihr Bereitstellungspaket zu Lambda hochladen. Andernfalls entpackt Lambda Ihre JAR-Datei möglicherweise nicht richtig, sodass Ihre `MANIFEST.MF`-Datei ignoriert wird.

Wenn Sie die Appender-Bibliothek (`aws-lambda-java-log4j2`) verwenden, müssen Sie auch einen Transformer für das Maven Shade-Plugin konfigurieren. Die Transformer-Bibliothek kombiniert Versionen einer Cache-Datei, die sowohl in der Appender-Bibliothek als auch in Log4j angezeigt wird.

Example pom.xml – Plugin-Konfiguration mit Log4j-2-Appender

```
      <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>
```

Hochladen eines Bereitstellungspakets mit der Lambda-Konsole

Eine neue Funktion müssen Sie zuerst in der Konsole erstellen und dann Ihre ZIP- oder JAR-Datei hochladen. Zum Aktualisieren einer bestehenden Funktion öffnen Sie die Seite für Ihre Funktion und gehen dann genauso vor, um Ihre aktualisierte ZIP- oder JAR-Datei hinzuzufügen.

Bei einer Bereitstellungspaketdatei unter 50 MB können Sie eine Funktion erstellen oder aktualisieren, indem Sie die Datei direkt von Ihrem lokalen Computer hochladen. Bei ZIP-oder JAR-Dateien mit einer Größe von mehr als 50 MB müssen Sie Ihr Paket zuerst in einen Amazon-S3-Bucket hochladen. Anweisungen zum Hochladen einer Datei in einen Amazon S3-Bucket mithilfe von finden Sie unter Erste Schritte mit Amazon S3. AWS-Managementkonsole Informationen zum Hochladen von Dateien mit dem AWS CLI finden Sie unter Objekte verschieben im AWS CLI Benutzerhandbuch.

> **Note**
>
> Sie können den Bereitstellungspakettyp (.zip oder Container-Image) für eine vorhandene Funktion nicht ändern. Sie können zum Beispiel eine Container-Image-Funktion nicht so umwandeln, dass sie ein .zip-Archiv verwendet. Sie müssen eine neue Funktion erstellen.

So erstellen Sie eine neue Funktion (Konsole)

1. Öffnen Sie die Funktionsseite der Lambda-Konsole und wählen Sie Funktion erstellen aus.
2. Wählen Sie Author from scratch aus.
3. Führen Sie unter Basic information (Grundlegende Informationen) die folgenden Schritte aus:

a. Geben Sie als Funktionsname den Namen Ihrer Funktion ein.

b. Wählen Sie für Laufzeit die Laufzeit aus, die Sie verwenden möchten.

c. (Optional) Für Architektur wählen Sie die Befehlssatz-Architektur für Ihre Funktion aus. Die Standardarchitektur ist x86_64. Stellen Sie sicher, dass das ZIP-Bereitstellungspaket für Ihre Funktion mit der von Ihnen gewählten Befehlssatzarchitektur kompatibel ist.

4. (Optional) Erweitern Sie unter Berechtigungen die Option Standardausführungsrolle ändern. Sie können eine neue Ausführungsrolle erstellen oder eine vorhandene Rolle verwenden.
5. Wählen Sie Funktion erstellen. Lambda erstellt eine grundlegende „Hello World“-Funktion mit der von Ihnen gewählten Laufzeit.

So laden Sie ein ZIP- oder JAR-Archiv von Ihrem lokalen Computer (Konsole) hoch

1. Wählen Sie auf der Funktionsseite der Lambda-Konsole die Funktion aus, für die Sie die ZIP- oder JAR-Datei hochladen möchten.
2. Wählen Sie die Registerkarte Code aus.
3. Wählen Sie im Bereich Codequelle die Option Hochladen von aus.
4. Wählen Sie ZIP- oder JAR-Datei aus.
5. Laden Sie die ZIP- oder JAR-Datei wie folgt hoch:

   a. Wählen Sie Hochladen und dann Ihre ZIP- oder JAR-Datei in der Dateiauswahl aus.

   b. Klicken Sie auf Open.

   c. Wählen Sie Speichern.

So laden Sie ein ZIP-Archiv aus einem Amazon-S3-Bucket hoch (Konsole)

1. Wählen Sie auf der Funktionsseite der Lambda-Konsole die Funktion aus, für die Sie die neue ZIP- oder JAR-Datei hochladen möchten.
2. Wählen Sie die Registerkarte Code aus.
3. Wählen Sie im Bereich Codequelle die Option Hochladen von aus.
4. Wählen Sie den Amazon-S3-Speicherort aus.
5. Fügen Sie die Amazon-S3-Link-URL Ihrer ZIP-Datei ein und wählen Sie Speichern aus.

Hochladen eines Bereitstellungspakets mit dem AWS CLI

Sie können die AWS CLI verwenden, um eine neue Funktion zu erstellen oder eine vorhandene unter Verwendung einer ZIP- oder JAR-Datei zu aktualisieren. Verwenden Sie die Create-Funktion und die update-function-codeBefehle, um Ihr .zip- oder JAR-Paket bereitzustellen. Wenn Ihre Datei kleiner als 50 MB ist, können Sie das Paket von einem Dateispeicherort auf Ihrem lokalen Build-Computer hochladen. Bei größeren Dateien müssen Sie Ihr ZIP-oder JAR-Paket aus einem Amazon-S3-Bucket hochladen. Anweisungen zum Hochladen einer Datei in einen Amazon S3 S3-Bucket mithilfe von finden Sie unter Objekte verschieben im AWS CLI Benutzerhandbuch. AWS CLI

> **Note**
>
> Wenn Sie Ihre .zip- oder JAR-Datei mithilfe von aus einem Amazon S3 S3-Bucket hochladen AWS CLI, muss sich der Bucket im selben Verzeichnis befinden AWS-Region wie Ihre Funktion.

Um eine neue Funktion mithilfe einer .zip- oder JAR-Datei mit der zu erstellen AWS CLI, müssen Sie Folgendes angeben:

- Den Namen Ihrer Funktion (`--function-name`)
- Die Laufzeit Ihrer Funktion (`--runtime`)
- Den Amazon-Ressourcennamen (ARN) der Ausführungsrolle der Funktion (`--role`).
- Den Namen der Handler-Methode in Ihrem Funktionscode (`--handler`)

Sie müssen auch den Speicherort Ihrer ZIP- oder JAR-Datei angeben. Befindet sich Ihre ZIP- oder JAR-Datei in einem Ordner auf Ihrem lokalen Build-Computer, verwenden Sie die `--zip-file`-Option, um den Dateipfad anzugeben, wie im folgenden Beispielbefehl gezeigt.

```
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
```

Zur Angabe des Speicherorts der ZIP-Datei in einem Amazon-S3-Bucket verwenden Sie die im folgenden Beispielbefehl gezeigte `--code`-Option. Sie müssen den `S3ObjectVersion`-Parameter nur für versionierte Objekte verwenden.

```
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
```

Um eine vorhandene Funktion mit der CLI zu aktualisieren, geben Sie den Namen Ihrer Funktion unter Verwendung des `--function-name`-Parameters an. Sie müssen auch den Speicherort der ZIP-Datei angeben, die Sie zum Aktualisieren Ihres Funktionscodes verwenden möchten. Befindet sich Ihre ZIP-Datei in einem Ordner auf Ihrem lokalen Build-Computer, verwenden Sie die `--zip-file`-Option, um den Dateipfad anzugeben, wie im folgenden Beispielbefehl gezeigt.

```
aws lambda update-function-code --function-name myFunction \
--zip-file fileb://myFunction.zip
```

Zur Angabe des Speicherorts der ZIP-Datei in einem Amazon-S3-Bucket verwenden Sie die im folgenden Beispielbefehl gezeigten `--s3-bucket`- und `--s3-key`-Optionen. Sie müssen den `--s3-object-version`-Parameter nur für versionierte Objekte verwenden.

```
aws lambda update-function-code --function-name myFunction \
--s3-bucket amzn-s3-demo-bucket --s3-key myFileName.zip --s3-object-version myObject
 Version
```

Hochladen eines Bereitstellungspakets mit AWS SAM

Sie können AWS SAM es verwenden, um die Bereitstellung Ihres Funktionscodes, Ihrer Konfiguration und Ihrer Abhängigkeiten zu automatisieren. AWS SAM ist eine Erweiterung von CloudFormation , die eine vereinfachte Syntax für die Definition serverloser Anwendungen bietet. Die folgende Beispielvorlage definiert eine Funktion mit einem Bereitstellungspaket in dem `build/distributions`-Verzeichnis, das Gradle verwendet.

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
```

Um die Funktion zu erstellen, verwenden Sie die Befehle „`package`“ und „`deploy`“. Bei diesen Befehlen handelt es sich um Anpassungen an der AWS CLI. Sie umschließen andere Befehle, um das Bereitstellungspaket zu Amazon S3 hochzuladen, die Vorlage mit dem Objekt-URI neu zu schreiben und den Code der Funktion zu aktualisieren.

Im folgenden Beispielskript wird ein Gradle-Build ausgeführt und das von ihm erstellte Bereitstellungspaket wird hochgeladen. Es erstellt einen CloudFormation Stack, wenn Sie es zum ersten Mal ausführen. Wenn der Stack bereits vorhanden ist, wird der vom Skript aktualisiert.

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
```

Ein vollständiges funktionierendes Beispiel finden Sie in den folgenden Beispielanwendungen.

Lambda-Beispielanwendungen in Java

- example-java – Eine Java-Funktion, die zeigt, wie Sie Lambda zur Bearbeitung von Aufträgen verwenden können. Diese Funktion veranschaulicht, wie Sie ein benutzerdefiniertes Eingabeereignisobjekt definieren und deserialisieren, das AWS SDK verwenden und wie Sie die Ausgabe protokollieren.

- Java-Basis – Eine Sammlung minimaler Java-Funktionen mit Einheitentests und variabler Protokollierungskonfiguration.
- Java-Ereignisse – Eine Sammlung von Java-Funktionen, die Grundcode für den Umgang mit Ereignissen aus verschiedenen Services wie Amazon API Gateway, Amazon SQS und Amazon Kinesis enthalten. Diese Funktionen verwenden die neueste Version der aws-lambda-java-eventsBibliothek (3.0.0 und neuer). Für diese Beispiele ist das AWS SDK nicht als Abhängigkeit erforderlich.
- s3-java – Eine Java-Funktion die Benachrichtigungsereignisse aus Amazon S3 verarbeitet und die Java Class Library (JCL) verwendet, um Miniaturansichten aus hochgeladenen Image-Dateien zu erstellen.
- layer-java – Eine Java-Funktion, die veranschaulicht, wie Sie mit einer Lambda-Ebene Abhängigkeiten getrennt von Ihrem Kernfunktionscode packen können.

Bereitstellen von Java-Lambda-Funktionen mit Container-Images

Es gibt drei Möglichkeiten, ein Container-Image für eine Java-Lambda-Funktion zu erstellen:

- Verwenden Sie ein Basis-Image für Java AWS

  Die AWS -Basis-Images sind mit einer Sprachlaufzeit, einem Laufzeitschnittstellen-Client zur Verwaltung der Interaktion zwischen Lambda und Ihrem Funktionscode und einem Laufzeitschnittstellen-Emulator für lokale Tests vorinstalliert.

- Es wird ein AWS reines Betriebssystem-Basis-Image verwendet

  AWS Basis-Images nur für Betriebssysteme enthalten eine Amazon Linux-Distribution und den Runtime-Interface-Emulator. Diese Images werden häufig verwendet, um Container-Images für kompilierte Sprachen wie Go und Rust sowie für eine Sprache oder Sprachversion zu erstellen, für die Lambda kein Basis-Image bereitstellt, wie Node.js 19. Sie können reine OS-Basis-Images auch verwenden, um eine benutzerdefinierte Laufzeit zu implementieren. Um das Image mit Lambda kompatibel zu machen, müssen Sie den Laufzeitschnittstellen-Client für Java in das Image aufnehmen.

- Verwenden Sie ein Nicht-Basis-Image AWS

  Sie können auch ein alternatives Basis-Image aus einer anderen Container-Registry verwenden. Sie können auch ein von Ihrer Organisation erstelltes benutzerdefiniertes Image verwenden. Um das Image mit Lambda kompatibel zu machen, müssen Sie den Laufzeitschnittstellen-Client für Java in das Image aufnehmen.

> **Tip**
>
> Um die Zeit zu reduzieren, die benötigt wird, bis Lambda-Container-Funktionen aktiv werden, siehe die Docker-Dokumentation unter Verwenden mehrstufiger Builds. Um effiziente Container-Images zu erstellen, folgen Sie den Bewährte Methoden für das Schreiben von Dockerfiles.

Auf dieser Seite wird erklärt, wie Sie Container-Images für Lambda erstellen, testen und bereitstellen.

Themen

- AWS Basis-Images für Java

- Verwenden Sie ein Basis-Image für Java AWS
- Verwenden eines alternativen Basis-Images mit dem Laufzeitschnittstellen-Client

AWS Basis-Images für Java

AWS stellt die folgenden Basis-Images für Java bereit:

Tags (Markier ngen)	Laufzeit	Betriebss ystem	Dockerfile	Ablehnung
25	Java 2.5	Amazon Linux 2023	Dockerfile für Java 25 auf GitHub	30. Juni 2029
21	Java 21	Amazon Linux 2023	Dockerfile für Java 21 auf GitHub	30. Juni 2029
17	Java 17	Amazon Linux 2	Dockerfile für Java 17 auf GitHub	30. Juni 2027
11	Java 11	Amazon Linux 2	Dockerfile für Java 11 auf GitHub	30. Juni 2027
8.al2	Java 8	Amazon Linux 2	Dockerfile für Java 8 auf GitHub	30. Juni 2027

Amazon ECR-Repository: gallery.ecr. aws/lambda/java

Die Basis-Images für Java 21 und höher basieren auf dem Amazon Linux 2023 Minimal Container Image. Frühere Basis-Images verwenden Amazon Linux 2. AL2023 bietet mehrere Vorteile gegenüber Amazon Linux 2, darunter einen geringeren Bereitstellungsaufwand und aktualisierte Versionen von Bibliotheken wie`glibc`.

AL2023-basierte Images verwenden `microdnf` (symbolisiert als`dnf`) als Paketmanager anstelle von`yum`, dem Standard-Paketmanager in Amazon Linux 2. `microdnf`ist eine eigenständige Implementierung von. `dnf` Eine Liste der Pakete, die in AL2 023-basierten Images enthalten sind, finden Sie in den Spalten Minimal Container unter Comparing packages installed on Amazon Linux 2023 Container Images. Weitere Informationen zu den Unterschieden zwischen AL2 023 und

Amazon Linux 2 finden Sie unter Einführung in die Amazon Linux 2023 Runtime for AWS Lambda im AWS Compute-Blog.

> **Note**
>
> Um AL2 023-basierte Images lokal auszuführen, auch mit AWS Serverless Application Model (AWS SAM), müssen Sie Docker-Version 20.10.10 oder höher verwenden.

Verwenden Sie ein Basis-Image für Java AWS

Voraussetzungen

Zur Durchführung der Schritte in diesem Abschnitt benötigen Sie Folgendes:

- Java (zum Beispiel Amazon Corretto)
- Apache Maven oder Gradle
- AWS CLI Version 2
- Docker (Mindestversion 25.0.0)
- Das Docker-buildx-Plugin.

Erstellen eines Images aus einem Base Image

Maven

1. Führen Sie den folgenden Befehl aus, um ein Maven-Projekt mit dem Archetyp für Lambda zu erstellen. Die folgenden Parameter sind erforderlich:

   - service — Der AWS-Service Client, der in der Lambda-Funktion verwendet werden soll. Eine Liste der verfügbaren Quellen finden Sie unter aws-sdk-java-v2/services on. GitHub
   - region — Der AWS-Region Ort, an dem Sie die Lambda-Funktion erstellen möchten.
   - groupId – Der vollständige Paket-Namespace Ihrer Anwendung.
   - artifactId – Ihr Projektname. Dies wird der Name des Verzeichnisses für Ihr Projekt.

   Führen Sie unter Linux und macOS folgenden Befehl aus:

   ```
   mvn -B archetype:generate \
   ```

```
    -DarchetypeGroupId=software.amazon.awssdk \
    -DarchetypeArtifactId=archetype-lambda -Dservice=s3 -Dregion=US_WEST_2 \
    -DgroupId=com.example.myapp \
    -DartifactId=myapp
```

Führen Sie in PowerShell diesen Befehl aus:

```
mvn -B archetype:generate `
    "-DarchetypeGroupId=software.amazon.awssdk" `
    "-DarchetypeArtifactId=archetype-lambda" "-Dservice=s3" "-Dregion=US_WEST_2"
 `
    "-DgroupId=com.example.myapp" `
    "-DartifactId=myapp"
```

Der Maven-Archetyp für Lambda ist für die Kompilierung mit Java SE 8 vorkonfiguriert und enthält eine Abhängigkeit von der AWS SDK für Java. Wenn Sie Ihr Projekt mit einem anderen Archetyp oder mit einer anderen Methode erstellen, müssen Sie den Java-Compiler für Maven konfigurieren und das SDK als Abhängigkeit deklarieren.

2. Öffnen Sie das *myapp*`/src/main/java/com/example/`*myapp*-Verzeichnis, und suchen Sie die `App.java`-Datei. Dies ist der Code für die Lambda-Funktion. Sie können den bereitgestellten Beispielcode zum Testen verwenden oder ihn durch Ihren eigenen ersetzen.
3. Navigieren Sie zurück zum Stammverzeichnis des Projekts und erstellen Sie ein neues Dockerfile mit der folgenden Konfiguration:

   - Setzen Sie die `FROM`-Eigenschaft auf den URI des Basis-Images.
   - Legen Sie das `CMD`-Argument auf den Lambda-Funktionshandler fest.

   Beachten Sie, dass das Dockerfile-Beispiel keine USER-Anweisung enthält. Wenn Sie ein Container-Image für Lambda bereitstellen, definiert Lambda automatisch einen Standard-Linux-Benutzer mit Berechtigungen mit geringsten Rechten. Dies unterscheidet sich vom Standardverhalten von Docker, bei dem standardmäßig der `root`-Benutzer verwendet wird, wenn keine `USER`-Anweisung bereitgestellt wird.

   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. Kompilieren Sie das Projekt und erfassen Sie die Laufzeitabhängigkeiten.

```
mvn compile dependency:copy-dependencies -DincludeScope=runtime
```

5. Erstellen Sie Ihr Docker-Image mit dem docker build-Befehl. Das folgende Beispiel benennt das Bild in `docker-image` und gibt ihm den `test` Tag. Um Ihr Image mit Lambda kompatibel zu machen, müssen Sie die Option `--provenance=false` verwenden.

```
docker buildx build --platform linux/amd64 --provenance=false -t docker-
image:test .
```

> **Note**
>
> Der Befehl gibt die `--platform linux/amd64`-Option an, um sicherzustellen, dass Ihr Container mit der Lambda-Ausführungsumgebung kompatibel ist, unabhängig von der Architektur des Entwicklungsrechners. Wenn Sie beabsichtigen, eine Lambda-Funktion mithilfe der ARM64 Befehlssatzarchitektur zu erstellen, müssen Sie den Befehl so ändern, dass er stattdessen die `--platform linux/arm64` Option verwendet.

Gradle

1. Erstellen Sie ein Verzeichnis für das Projekt und wechseln Sie dann zu diesem Verzeichnis.

```
mkdir example
cd example
```

2. Führen Sie den folgenden Befehl aus, um Gradle ein neues Java-Anwendungsprojekt im Verzeichnis `example` der Umgebung generieren zu lassen. Wählen Sie für Select Build Script DSL die Option 2: Groovy aus.

```
gradle init --type java-application
```

3. Öffnen Sie das */example*/app/src/main/java/*example*-Verzeichnis, und suchen Sie die App.java-Datei. Dies ist der Code für die Lambda-Funktion. Sie können den folgenden Beispielcode zum Testen verwenden oder ihn durch Ihren eigenen ersetzen.

   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. Öffnen Sie die build.gradle Datei. Wenn Sie den Beispiel-Funktionscode aus dem vorherigen Schritt verwenden, ersetzen Sie den Inhalt von build.gradle durch den folgenden. Wenn Sie Ihren eigenen Funktionscode verwenden, ändern Sie Ihre build.gradle-Datei nach Bedarf.

   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. Der `gradle init`-Befehl aus Schritt 2 generierte auch einen Dummy-Testfall im `app/test`-Verzeichnis. Überspringen Sie in diesem Tutorial das Ausführen von Tests, indem Sie das `/test`-Verzeichnis löschen.
6. Erstellen Sie das Projekt.

```
gradle build
```

7. Erstellen Sie im Stammverzeichnis (`/example`) des Projekts eine Docker-Datei mit der folgenden Konfiguration:

   - Setzen Sie die `FROM`-Eigenschaft auf den URI des Basis-Images.
   - Verwenden Sie den Befehl COPY, um den Funktionscode und die Laufzeitabhängigkeiten in `{LAMBDA_TASK_ROOT}`, eine von Lambda definierte Umgebungsvariable zu kopieren.
   - Legen Sie das `CMD`-Argument auf den Lambda-Funktionshandler fest.

   Beachten Sie, dass das Dockerfile-Beispiel keine USER-Anweisung enthält. Wenn Sie ein Container-Image für Lambda bereitstellen, definiert Lambda automatisch einen Standard-Linux-Benutzer mit Berechtigungen mit geringsten Rechten. Dies unterscheidet sich vom Standardverhalten von Docker, bei dem standardmäßig der `root`-Benutzer verwendet wird, wenn keine `USER`-Anweisung bereitgestellt wird.

   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. Erstellen Sie Ihr Docker-Image mit dem docker build-Befehl. Das folgende Beispiel benennt das Bild in `docker-image` und gibt ihm den `test` Tag. Um Ihr Image mit Lambda kompatibel zu machen, müssen Sie die Option `--provenance=false` verwenden.

```
docker buildx build --platform linux/amd64 --provenance=false -t docker-
image:test .
```

> **Note**
>
> Der Befehl gibt die `--platform linux/amd64`-Option an, um sicherzustellen, dass Ihr Container mit der Lambda-Ausführungsumgebung kompatibel ist, unabhängig von der Architektur des Entwicklungsrechners. Wenn Sie beabsichtigen, eine Lambda-Funktion mithilfe der ARM64 Befehlssatzarchitektur zu erstellen, müssen Sie den Befehl so ändern, dass er stattdessen die `--platform linux/arm64` Option verwendet.

(Optional) Testen Sie das Image lokal

1. Starten Sie Ihr Docker-Image mit dem docker run-Befehl. In diesem Beispiel ist `docker-image` der Image-Name und `test` der Tag.

   ```
   docker run --platform linux/amd64 -p 9000:8080 docker-image:test
   ```

   Dieser Befehl führt das Image als Container aus und erstellt einen lokalen Endpunkt bei `localhost:9000/2015-03-31/functions/function/invocations`.

   > **Note**
   >
   > Wenn Sie das Docker-Image für die ARM64 Befehlssatzarchitektur erstellt haben, stellen Sie sicher, dass Sie stattdessen die `--platform linux/arm64` Option verwenden. `--platform linux/amd64`

2. Veröffentlichen Sie in einem neuen Terminalfenster ein Ereignis an den lokalen Endpunkt.

   Linux/macOS

   Führen Sie unter Linux oder macOS den folgenden `curl`-Befehl aus:

   ```
   curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{}'
   ```

   Dieser Befehl ruft die Funktion mit einem leeren Ereignis auf und gibt eine Antwort zurück. Wenn Sie Ihren eigenen Funktionscode anstelle des Beispielfunktionscodes verwenden, wird empfohlen, die Funktion mit einer JSON-Nutzlast aufzurufen. Beispiel:

```
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d
 '{"payload":"hello world!"}'
```

PowerShell

Führen Sie in PowerShell den folgenden `Invoke-WebRequest` Befehl aus:

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{}' -ContentType "application/json"
```

Dieser Befehl ruft die Funktion mit einem leeren Ereignis auf und gibt eine Antwort zurück. Wenn Sie Ihren eigenen Funktionscode anstelle des Beispielfunktionscodes verwenden, wird empfohlen, die Funktion mit einer JSON-Nutzlast aufzurufen. Beispiel:

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{"payload":"hello world!"}' -ContentType
 "application/json"
```

3. Die Container-ID erhalten.

```
docker ps
```

4. Verwenden Sie den Befehl docker kill, um den Container zu anzuhalten. Ersetzen Sie in diesem Befehl `3766c4ab331c` durch die Container-ID aus dem vorherigen Schritt.

```
docker kill 3766c4ab331c
```

Das Image bereitstellen

Um das Image in Amazon ECR hochzuladen und die Lambda-Funktion zu erstellen

1. Führen Sie den get-login-passwordBefehl aus, um die Docker-CLI bei Ihrer Amazon ECR-Registrierung zu authentifizieren.

   - Setzen Sie den `--region` Wert auf den AWS-Region Ort, an dem Sie das Amazon ECR-Repository erstellen möchten.
   - Ersetzen Sie es `111122223333` durch Ihre AWS-Konto ID.

```
aws ecr get-login-password --region us-east-1 | docker login --username AWS --
password-stdin 111122223333.dkr.ecr.us-east-1.amazonaws.com
```

2. Erstellen Sie ein Repository in Amazon ECR mithilfe des Befehls create-repository.

```
aws ecr create-repository --repository-name hello-world --region us-east-1 --image-
scanning-configuration scanOnPush=true --image-tag-mutability MUTABLE
```

> **Note**
>
> Das Amazon ECR-Repository muss sich im selben Format AWS-Region wie die Lambda-Funktion befinden.

Wenn erfolgreich, sehen Sie eine Antwort wie diese:

```
{
    "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. Kopieren Sie das `repositoryUri` aus der Ausgabe im vorherigen Schritt.
4. Führen Sie den Befehl docker tag aus, um Ihr lokales Image als neueste Version in Ihrem Amazon-ECR-Repository zu markieren. In diesem Befehl gilt Folgendes:

- `docker-image:test` ist der Name und das Tag Ihres Docker-Images. Dies sind der Imagename und das Tag, die Sie im `docker build`-Befehl angegeben haben.
- Ersetzen Sie <ECRrepositoryUri> durch den `repositoryUri`, den Sie kopiert haben. Stellen Sie sicher, dass Sie `:latest` am Ende der URI angeben.

```
docker tag docker-image:test <ECRrepositoryUri>:latest
```

Beispiel:

```
docker tag docker-image:test 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

5. Führen Sie den Befehl docker push aus, um Ihr lokales Image im Amazon-ECR-Repository bereitzustellen. Stellen Sie sicher, dass Sie `:latest` am Ende der Repository-URI angeben.

```
docker push 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

6. Erstellen Sie eine Ausführungsrolle für die Funktion, wenn Sie noch keine haben. Sie benötigen den Amazon-Ressourcennamen (ARN) der Rolle im nächsten Schritt.
7. So erstellen Sie die Lambda-Funktion: Geben Sie für `ImageUri` die Repository-URI von zuvor an. Stellen Sie sicher, dass Sie `:latest` am Ende der URI angeben.

```
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**
>
> Sie können eine Funktion mit einem Bild in einem anderen AWS Konto erstellen, sofern sich das Bild in derselben Region wie die Lambda-Funktion befindet. Weitere Informationen finden Sie unter Kontoübergreifende Berechtigungen von Amazon ECR.

8. Die Funktion aufrufen.

```
aws lambda invoke --function-name hello-world response.json
```

Das Ergebnis sollte ungefähr wie folgt aussehen:

```
{
  "ExecutedVersion": "$LATEST",
  "StatusCode": 200
}
```

9. Um die Ausgabe der Funktion zu sehen, überprüfen Sie die `response.json`-Datei.

Um den Funktionscode zu aktualisieren, müssen Sie das Image erneut erstellen, das neue Image in das Amazon ECR-Repository hochladen und dann den update-function-codeBefehl verwenden, um das Image für die Lambda-Funktion bereitzustellen.

Lambda löst das Image-Tag in einen bestimmten Image-Digest auf. Das heißt, wenn Sie das Image-Tag, das zur Bereitstellung der Funktion verwendet wurde, auf ein neues Image in Amazon ECR verweisen, aktualisiert Lambda die Funktion nicht automatisch, um das neue Image zu verwenden.

Um das neue Image für dieselbe Lambda-Funktion bereitzustellen, müssen Sie den update-function-codeBefehl verwenden, auch wenn das Image-Tag in Amazon ECR gleich bleibt. Im folgenden Beispiel erstellt die `--publish`-Option eine neue Version der Funktion unter Verwendung des aktualisierten Container-Images.

```
aws lambda update-function-code \
  --function-name hello-world \
  --image-uri 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --publish
```

Verwenden eines alternativen Basis-Images mit dem Laufzeitschnittstellen-Client

Wenn Sie ein OS-Basis-Image oder ein alternatives Basis-Image verwenden, müssen Sie den Laufzeitschnittstellen-Client in das Image einbinden. Der Laufzeitschnittstellen-Client erweitert die Laufzeit-API, die die Interaktion zwischen Lambda und Ihrem Funktionscode verwaltet.

Installieren Sie den Laufzeitschnittstellen-Client für Java in Ihrem Dockerfile oder als Abhängigkeit in Ihrem Projekt. Um beispielsweise den Laufzeitschnittstellen-Client mit dem Maven-Paketmanager zu installieren, fügen Sie Ihrer `pom.xml`-Datei Folgendes hinzu:

```
<dependency>
    <groupId>com.amazonaws</groupId>
    <artifactId>aws-lambda-java-runtime-interface-client</artifactId>
    <version>2.3.2</version>
</dependency>
```

Einzelheiten zum Paket finden Sie unter AWS Lambda Java Runtime Interface Client im Maven-Central-Repository. Sie können den Quellcode des Runtime-Interface-Clients auch im GitHub Repository der AWS Lambda Java Support Libraries überprüfen.

Das folgende Beispiel zeigt, wie Sie mithilfe eines Amazon-Corretto-Images ein Container-Image für Java erstellen. Amazon Corretto ist eine kostenlose, plattformübergreifende und produktionsbereite Distribution des Open Java Development Kit (OpenJDK). Das Maven-Projekt beinhaltet den Laufzeitschnittstellen-Client als Abhängigkeit.

Voraussetzungen

Zur Durchführung der Schritte in diesem Abschnitt benötigen Sie Folgendes:

- Java (zum Beispiel Amazon Corretto)
- Apache Maven
- AWS CLI Version 2
- Docker (Mindestversion 25.0.0)
- Das Docker-buildx-Plugin.

Erstellen eines Images aus einem alternativen Basis-Image

1. Erstellen Sie ein Maven-Projekt. Die folgenden Parameter sind erforderlich:
    - groupId – Der vollständige Paket-Namespace Ihrer Anwendung.
    - artifactId – Ihr Projektname. Dies wird der Name des Verzeichnisses für Ihr Projekt.

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. Öffnen Sie das Projektverzeichnis.

```
cd myapp
```

3. Öffnen Sie die Datei `pom.xml` und ersetzen Sie den Inhalt durch Folgendes. Diese Datei enthält den aws-lambda-java-runtime-interface-client als Abhängigkeit. Alternativ können Sie den Laufzeitschnittstellen-Client im Dockerfile installieren. Der einfachste Ansatz besteht jedoch darin, die Bibliothek als Abhängigkeit einbinden.

```
<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. Öffnen Sie das *myapp*`/src/main/java/com/example/`*myapp*-Verzeichnis, und suchen Sie die `App.java`-Datei. Dies ist der Code für die Lambda-Funktion. Ersetzen Sie den Code durch Folgendes:

   Example Funktions-Handler

   ```
   package example;

   public class App {
       public static String sayHello() {
           return "Hello world!";
       }
   }
   ```

5. Der `mvn -B archetype:generate`-Befehl aus Schritt 1 generierte auch einen Dummy-Testfall im `src/test`-Verzeichnis. Überspringen Sie in diesem Tutorial das Ausführen von Tests, indem Sie das gesamte generierte Verzeichnis `/test` löschen.
6. Navigieren Sie zurück zum Stammverzeichnis des Projekts und erstellen Sie dann ein neues Dockerfile. Das folgende Beispiel-Dockerfile verwendet ein Amazon-Corretto-Image. Amazon Corretto ist eine kostenlose, plattformübergreifende und produktionsbereite Distribution des OpenJDK.

   - Setzen Sie die `FROM`-Eigenschaft auf den URI des Basis-Images.
   - Legen Sie `ENTRYPOINT` auf das Modul fest, das der Docker-Container beim Start ausführen soll. In diesem Fall ist das Modul der Laufzeitschnittstellen-Client.
   - Legen Sie das `CMD`-Argument auf den Lambda-Funktionshandler fest.

   Beachten Sie, dass das Dockerfile-Beispiel keine USER-Anweisung enthält. Wenn Sie ein Container-Image für Lambda bereitstellen, definiert Lambda automatisch einen Standard-Linux-Benutzer mit Berechtigungen mit geringsten Rechten. Dies unterscheidet sich vom Standardverhalten von Docker, bei dem standardmäßig der `root`-Benutzer verwendet wird, wenn keine `USER`-Anweisung bereitgestellt wird.

   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. Erstellen Sie Ihr Docker-Image mit dem docker build-Befehl. Das folgende Beispiel benennt das Bild in `docker-image` und gibt ihm den `test` Tag. Um Ihr Image mit Lambda kompatibel zu machen, müssen Sie die Option `--provenance=false` verwenden.

```
docker buildx build --platform linux/amd64 --provenance=false -t docker-image:test
 .
```

> **Note**
>
> Der Befehl gibt die `--platform linux/amd64`-Option an, um sicherzustellen, dass Ihr Container mit der Lambda-Ausführungsumgebung kompatibel ist, unabhängig von der Architektur des Entwicklungsrechners. Wenn Sie beabsichtigen, eine Lambda-Funktion mithilfe der ARM64 Befehlssatzarchitektur zu erstellen, müssen Sie den Befehl so ändern, dass er stattdessen die `--platform linux/arm64` Option verwendet.

(Optional) Testen Sie das Image lokal

Verwenden Sie den Laufzeit-Schnittstellen-Emulator, um das Image lokal zu testen. Sie können den Emulator in Ihr Image integrieren oder ihn mit dem folgenden Verfahren auf Ihrem lokalen Computer installieren.

Installieren des Laufzeitschnittstellen-Emulators auf Ihrem lokalen Computer

1. Führen Sie in Ihrem Projektverzeichnis den folgenden Befehl aus, um den Runtime-Interface-Emulator (x86-64-Architektur) herunterzuladen GitHub und auf Ihrem lokalen Computer zu installieren.

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
```

Um den arm64-Emulator zu installieren, ersetzen Sie die GitHub Repository-URL im vorherigen Befehl durch Folgendes:

```
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
```

Um den arm64-Emulator zu installieren, ersetzen Sie das `$downloadLink` durch Folgendes:

```
https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/download/aws-lambda-rie-arm64
```

2. Starten Sie Ihr Docker-Image mit dem docker run-Befehl. Beachten Sie Folgendes:

   - `docker-image` ist der Image-Name und `test` ist das Tag.
   - `/usr/bin/java -cp './*' com.amazonaws.services.lambda.runtime.api.client.AWSLambda example.App::sayHello` ist der `ENTRYPOINT` gefolgt von dem `CMD` aus Ihrem 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
```

Dieser Befehl führt das Image als Container aus und erstellt einen lokalen Endpunkt bei `localhost:9000/2015-03-31/functions/function/invocations`.

> **Note**
>
> Wenn Sie das Docker-Image für die ARM64 Befehlssatzarchitektur erstellt haben, stellen Sie sicher, dass Sie die `--platform linux/arm64` Option anstelle von verwenden. `--platform linux/amd64`

3. Veröffentlichen Sie ein Ereignis auf dem lokalen Endpunkt.

   Linux/macOS

   Führen Sie unter Linux oder macOS den folgenden `curl`-Befehl aus:

   ```
   curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{}'
   ```

Dieser Befehl ruft die Funktion mit einem leeren Ereignis auf und gibt eine Antwort zurück. Wenn Sie Ihren eigenen Funktionscode anstelle des Beispielfunktionscodes verwenden, wird empfohlen, die Funktion mit einer JSON-Nutzlast aufzurufen. Beispiel:

```
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d
 '{"payload":"hello world!"}'
```

PowerShell

Führen Sie in PowerShell den folgenden `Invoke-WebRequest` Befehl aus:

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{}' -ContentType "application/json"
```

Dieser Befehl ruft die Funktion mit einem leeren Ereignis auf und gibt eine Antwort zurück. Wenn Sie Ihren eigenen Funktionscode anstelle des Beispielfunktionscodes verwenden, wird empfohlen, die Funktion mit einer JSON-Nutzlast aufzurufen. Beispiel:

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{"payload":"hello world!"}' -ContentType
 "application/json"
```

4. Die Container-ID erhalten.

```
docker ps
```

5. Verwenden Sie den Befehl docker kill, um den Container zu anzuhalten. Ersetzen Sie in diesem Befehl `3766c4ab331c` durch die Container-ID aus dem vorherigen Schritt.

```
docker kill 3766c4ab331c
```

Das Image bereitstellen

Um das Image in Amazon ECR hochzuladen und die Lambda-Funktion zu erstellen

1. Führen Sie den get-login-passwordBefehl aus, um die Docker-CLI bei Ihrer Amazon ECR-Registrierung zu authentifizieren.

- Setzen Sie den `--region` Wert auf den AWS-Region Ort, an dem Sie das Amazon ECR-Repository erstellen möchten.
- Ersetzen Sie es `111122223333` durch Ihre AWS-Konto ID.

```
aws ecr get-login-password --region us-east-1 | docker login --username AWS --
password-stdin 111122223333.dkr.ecr.us-east-1.amazonaws.com
```

2. Erstellen Sie ein Repository in Amazon ECR mithilfe des Befehls create-repository.

```
aws ecr create-repository --repository-name hello-world --region us-east-1 --image-
scanning-configuration scanOnPush=true --image-tag-mutability MUTABLE
```

> **Note**
>
> Das Amazon ECR-Repository muss sich im selben Format AWS-Region wie die Lambda-Funktion befinden.

Wenn erfolgreich, sehen Sie eine Antwort wie diese:

```
{
    "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. Kopieren Sie das `repositoryUri` aus der Ausgabe im vorherigen Schritt.
4. Führen Sie den Befehl docker tag aus, um Ihr lokales Image als neueste Version in Ihrem Amazon-ECR-Repository zu markieren. In diesem Befehl gilt Folgendes:

   - `docker-image:test` ist der Name und das Tag Ihres Docker-Images. Dies sind der Imagename und das Tag, die Sie im `docker build`-Befehl angegeben haben.
   - Ersetzen Sie `<ECRrepositoryUri>` durch den `repositoryUri`, den Sie kopiert haben. Stellen Sie sicher, dass Sie `:latest` am Ende der URI angeben.

   ```
   docker tag docker-image:test <ECRrepositoryUri>:latest
   ```

   Beispiel:

   ```
   docker tag docker-image:test 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
   ```

5. Führen Sie den Befehl docker push aus, um Ihr lokales Image im Amazon-ECR-Repository bereitzustellen. Stellen Sie sicher, dass Sie `:latest` am Ende der Repository-URI angeben.

   ```
   docker push 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
   ```

6. Erstellen Sie eine Ausführungsrolle für die Funktion, wenn Sie noch keine haben. Sie benötigen den Amazon-Ressourcennamen (ARN) der Rolle im nächsten Schritt.
7. So erstellen Sie die Lambda-Funktion: Geben Sie für `ImageUri` die Repository-URI von zuvor an. Stellen Sie sicher, dass Sie `:latest` am Ende der URI angeben.

   ```
   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**
   >
   > Sie können eine Funktion mit einem Bild in einem anderen AWS Konto erstellen, sofern sich das Bild in derselben Region wie die Lambda-Funktion befindet. Weitere Informationen finden Sie unter Kontoübergreifende Berechtigungen von Amazon ECR.

8. Die Funktion aufrufen.

```
aws lambda invoke --function-name hello-world response.json
```

Das Ergebnis sollte ungefähr wie folgt aussehen:

```
{
  "ExecutedVersion": "$LATEST",
  "StatusCode": 200
}
```

9. Um die Ausgabe der Funktion zu sehen, überprüfen Sie die `response.json`-Datei.

Um den Funktionscode zu aktualisieren, müssen Sie das Image erneut erstellen, das neue Image in das Amazon ECR-Repository hochladen und dann den update-function-codeBefehl verwenden, um das Image für die Lambda-Funktion bereitzustellen.

Lambda löst das Image-Tag in einen bestimmten Image-Digest auf. Das heißt, wenn Sie das Image-Tag, das zur Bereitstellung der Funktion verwendet wurde, auf ein neues Image in Amazon ECR verweisen, aktualisiert Lambda die Funktion nicht automatisch, um das neue Image zu verwenden.

Um das neue Image für dieselbe Lambda-Funktion bereitzustellen, müssen Sie den update-function-codeBefehl verwenden, auch wenn das Image-Tag in Amazon ECR gleich bleibt. Im folgenden Beispiel erstellt die `--publish`-Option eine neue Version der Funktion unter Verwendung des aktualisierten Container-Images.

```
aws lambda update-function-code \
  --function-name hello-world \
  --image-uri 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --publish
```

Arbeiten mit Ebenen für Java Lambda-Funktionen

Verwenden Sie Lambda-Ebenen, um Code und Abhängigkeiten zu bündeln, die Sie für mehrere Funktionen wiederverwenden möchten. Ebenen enthalten üblicherweise Bibliotheksabhängigkeiten, eine benutzerdefinierte Laufzeit oder Konfigurationsdateien. Das Erstellen einer Ebene umfasst drei allgemeine Schritte:

1. Verpacken Ihres Ebeneninhalts. Dies bedeutet, dass Sie ein .zip-Dateiarchiv erstellen, das die Abhängigkeiten enthält, die Sie in Ihren Funktionen verwenden möchten.
2. Erstellen Sie die Ebene in Lambda.
3. Fügen Sie die Ebene zu Ihren Funktionen hinzu.

Themen

- Verpacken Ihres Ebeneninhalts
- Erstellen Sie die Ebene in Lambda
- Fügen Sie die Ebene zu Ihrer Funktion hinzu

Verpacken Ihres Ebeneninhalts

Um eine Ebene zu erstellen, bündeln Sie Ihre Pakete in einem ZIP-Dateiarchiv, das die folgenden Anforderungen erfüllt:

- Stellen Sie sicher, dass die Java-Version, auf die sich Maven oder Gradle bezieht, mit der Java-Version der Funktion übereinstimmt, die Sie bereitstellen möchten. Für eine Java 25-Funktion sollte der `mvn -v` Befehl beispielsweise Java 25 in der Ausgabe auflisten.
- Ihre Abhängigkeiten müssen im Verzeichnis `java/lib` im Stammverzeichnis der ZIP-Datei gespeichert werden. Weitere Informationen finden Sie unter Ebenenpfade für jede Lambda-Laufzeit.
- Die Pakete in Ihrer Ebene müssen mit Linux kompatibel sein. Lambda-Funktionen werden auf Amazon Linux ausgeführt.

Sie können Ebenen erstellen, die entweder Java-Bibliotheken von Drittanbietern oder Ihre eigenen Java-Module und -Pakete enthalten. Im Folgenden wird Maven verwendet. Sie können auch Gradle verwenden, um den Inhalt Ihrer Ebenen zu verpacken.

Erstellen einer Ebene mit Maven-Abhängigkeiten

1. Erstellen Sie ein Apache-Maven-Projekt mit einer `pom.xml`-Datei, die Ihre Abhängigkeiten definiert.

   Das folgende Beispiel enthält Jackson Databind für die JSON-Verarbeitung. `<build>`In diesem Abschnitt werden separate JAR-Dateien für jede Abhängigkeit erstellt, anstatt sie in einer einzigen Uber-Jar-Datei zu bündeln. maven-dependency-plugin Wenn Sie ein Uber-Jar erstellen möchten, verwenden Sie das. 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. Erstellen Sie das Projekt. Dieser Befehl erstellt alle JAR-Abhängigkeitsdateien im `target/lib/`-Verzeichnis.

```
mvn clean package
```

3. Erstellen Sie die erforderliche Verzeichnisstruktur für Ihre Ebene:

```
mkdir -p java/lib
```

4. Kopieren Sie die JAR-Abhängigkeitsdateien in das `java/lib`-Verzeichnis:

```
cp target/lib/*.jar java/lib/
```

5. Verpacken Sie den Ebeneninhalt:

   Linux/macOS

   ```
   zip -r layer.zip java/
   ```

   PowerShell

   ```
   Compress-Archive -Path .\java -DestinationPath .\layer.zip
   ```

   Die Verzeichnisstruktur Ihrer ZIP-Datei sollte wie folgt aussehen:

```
java/
### lib/
    ### jackson-databind-2.17.0.jar
    ### jackson-core-2.17.0.jar
```

```
### jackson-annotations-2.17.0.jar
```

> **Note**
>
> Stellen Sie sicher, dass Ihre ZIP-Datei das `java`-Verzeichnis auf der Stammebene enthält und sich `lib` darin befindet. Diese Struktur stellt sicher, dass Lambda Ihre Bibliotheken finden und importieren kann. Jede Abhängigkeit wird als separate JAR-Datei gespeichert und nicht in einem Uber-JAR gebündelt.

Erstellen Sie die Ebene in Lambda

Sie können Ihren Layer entweder mit der AWS CLI oder der Lambda-Konsole veröffentlichen.

AWS CLI

Führen Sie den publish-layer-version AWS CLI Befehl aus, um die Lambda-Schicht zu erstellen:

```
aws lambda publish-layer-version --layer-name my-layer --zip-file fileb://layer.zip
 --compatible-runtimes java25
```

Der Parameter Kompatible Laufzeiten ist optional. Wenn angegeben, verwendet Lambda diesen Parameter zum Filtern von Ebenen in der Lambda-Konsole.

Console

So erstellen Sie eine Ebene (Konsole)

1. Öffnen Sie die Seite Ebenen der Lambda-Konsole.
2. Wählen Sie Create Layer (Ebene erstellen) aus.
3. Wählen Sie ZIP-Datei hochladen und laden Sie dann das zuvor erstellte ZIP-Archiv hoch.
4. (Optional) Wählen Sie für Kompatible Laufzeiten die Java-Laufzeit aus, die der Java-Version entspricht, mit der Sie die Ebene erstellt haben.
5. Wählen Sie Erstellen aus.

Fügen Sie die Ebene zu Ihrer Funktion hinzu

AWS CLI

Führen Sie den update-function-configuration AWS CLI Befehl aus, um die Ebene an Ihre Funktion anzuhängen. Verwenden Sie den ARN der Ebene für den `--layers`-Parameter. Der ARN muss die Version angeben (z. B. `arn:aws:lambda:us-east-1:123456789012:layer:my-layer:1`). Weitere Informationen finden Sie unter Ebenen und Ebenenversionen.

```
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"
```

Die cli-binary-format Option ist erforderlich, wenn Sie AWS CLI Version 2 verwenden. Um dies zur Standardeinstellung zu machen, führen Sie `aws configure set cli-binary-format raw-in-base64-out` aus. Weitere Informationen finden Sie unter Von AWS CLI unterstützte globale Befehlszeilenoptionen im AWS Command Line Interface -Benutzerhandbuch für Version 2.

Console

So fügen Sie eine Ebene zu einer Funktion hinzu

1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.
2. Wählen Sie die Funktion aus.
3. Blättern Sie nach unten bis zum Abschnitt Ebenen und wählen Sie dann Eine Ebene hinzufügen.
4. Wählen Sie unter Ebene auswählen die Option Benutzerdefinierte Ebenen und dann Ihre Ebene aus.

   > **Note**
   >
   > Wenn Sie bei der Erstellung der Ebene keine kompatible Laufzeit hinzugefügt haben, wird die Ebene hier nicht aufgeführt. Sie können stattdessen den ARN der Ebene angeben.

5. Wählen Sie Hinzufügen aus.

Passen Sie die Serialisierung für Lambda-Java-Funktionen an

Die verwalteten Lambda-Java-Laufzeiten unterstützen die benutzerdefinierte Serialisierung für JSON-Ereignisse. Eine benutzerdefinierte Serialisierung kann Ihren Code vereinfachen und möglicherweise die Leistung verbessern.

Themen

- Wann sollte die benutzerdefinierte Serialisierung verwendet werden
- Benutzerdefinierte Serialisierung implementieren
- Testen der benutzerdefinierten Serialisierung

Wann sollte die benutzerdefinierte Serialisierung verwendet werden

Wenn Ihre Lambda-Funktion aufgerufen wird, müssen die Eingabeereignisdaten in ein Java-Objekt deserialisiert werden und die Ausgabe Ihrer Funktion muss wieder in ein Format serialisiert werden, das als Antwort der Funktion zurückgegeben werden kann. Die von Lambda Java verwalteten Laufzeiten bieten standardmäßige Serialisierungs- und Deserialisierungsfunktionen, die sich gut für die Verarbeitung von Ereignisnutzlasten verschiedener AWS-Dienste wie Amazon API Gateway und Amazon Simple Queue Service (Amazon SQS) eignen. Um mit diesen Serviceintegrationsereignissen in Ihrer Funktion zu arbeiten, fügen Sie Ihrem Projekt die Abhängigkeit aws-java-lambda-events hinzu. Diese AWS-Bibliothek enthält Java-Objekte, die diese Serviceintegrationsereignisse darstellen.

Sie können auch Ihre eigenen Objekte verwenden, um den Event-JSON darzustellen, den Sie an Ihre Lambda-Funktion übergeben. Die verwaltete Laufzeit versucht, den JSON mit seinem Standardverhalten in eine neue Instance Ihres Objekts zu serialisieren. Wenn der Standard-Serializer nicht das gewünschte Verhalten für Ihren Anwendungsfall aufweist, verwenden Sie die benutzerdefinierte Serialisierung.

Angenommen, eine `Vehicle`-Klasse mit dem Namen besitzt die folgende Verzeichnisstruktur:

```
public class Vehicle {
    private String vehicleType;
    private long vehicleId;
}
```

Die Nutzdaten des JSON-Ereignisses sieht jedoch wie folgt aus:

```
{
```

```
    "vehicle-type": "car",
    "vehicleID": 123
}
```

In diesem Szenario erwartet die Standard-Serialisierung in der verwalteten Laufzeitumgebung, dass die JSON-Eigenschaftsnamen mit den Eigenschaftsnamen der Java-Klassen (`vehicleType`, `vehicleId`) in Kamelform übereinstimmen. Da die Eigenschaftsnamen im JSON-Ereignis nicht in Großbuchstaben (`vehicle-type`, `vehicleID`) geschrieben sind, müssen Sie eine benutzerdefinierte Serialisierung verwenden.

Benutzerdefinierte Serialisierung implementieren

Verwenden Sie ein Service Provider-Interface, um einen Serializer Ihrer Wahl anstelle der Standard-Serialisierungslogik der verwalteten Laufzeit zu laden. Mithilfe der `RequestHandler`-Standardschnittstelle können Sie Ihre JSON-Event-Nutzdaten direkt in Java-Objekte serialisieren.

So verwenden Sie die benutzerdefinierte Serialisierung in Ihrer Lambda-Java-Funktion

1. Fügen Sie die aws-lambda-java-core-Bibliothek als Abhängigkeit hinzu. Diese Bibliothek enthält die CustomPojoSerializer-Schnittstelle sowie andere Schnittstellendefinitionen für die Arbeit mit Java in Lambda.
2. Erstellen Sie eine Datei namens `com.amazonaws.services.lambda.runtime.CustomPojoSerializer` im Verzeichnis `src/main/resources/META-INF/services/` Ihres Projekts.
3. Geben Sie in dieser Datei den vollqualifizierten Namen Ihrer benutzerdefinierten Serializer-Implementierung an, die die `CustomPojoSerializer`-Schnittstelle implementieren muss. Beispiel:

   ```
   com.mycompany.vehicles.CustomLambdaSerialzer
   ```
4. Implementieren Sie die `CustomPojoSerializer`-Schnittstelle, um Ihre benutzerdefinierte Serialisierungslogik bereitzustellen.
5. Verwenden Sie die `RequestHandler`-Standardschnittstelle in Ihrer Lambda-Funktion. Die verwaltete Laufzeit verwendet Ihren benutzerdefinierten Serializer.

Weitere Beispiele für die Implementierung einer benutzerdefinierten Serialisierung mit gängigen Bibliotheken wie fastJson, Gson, Moshi und jackson-jr finden Sie im Beispiel für custom-serialization im AWS-GitHub-Repository.

Testen der benutzerdefinierten Serialisierung

Testen Sie Ihre Funktion, um sicherzustellen, dass Ihre Serialisierungs- und Deserialisierungslogik wie erwartet funktioniert. Sie können die AWS Serverless Application Model-Befehlszeilenschnittstelle (AWS SAM CLI) verwenden, um den Aufruf Ihrer Lambda-Nutzdaten zu emulieren. Dies kann Ihnen helfen, Ihre Funktion schnell zu testen und zu iterieren, wenn Sie einen benutzerdefinierten Serializer einführen.

1. Erstellen Sie eine Datei mit der JSON-Ereignisnutzdaten, mit der Sie Ihre Funktion aufrufen möchten und rufen Sie dann die AWS SAM CLI auf.
2. Führen Sie denselben lokalen Invoke-Befehl aus, um Ihre Funktion lokal aufzurufen. Beispiel:

```
sam local invoke -e src/test/resources/event.json
```

Weitere Informationen erhalten Sie unter Lokaler Aufruf von Lambda-Funktionen mit AWS SAM.

Passen Sie das Startverhalten der Java-Laufzeit für Lambda-Funktionen an

Auf dieser Seite werden spezifische Einstellungen für Java-Funktionen in beschrieben AWS Lambda. Sie können diese Einstellungen verwenden, um das Startup-Verhalten der Java-Laufzeit anzupassen. Dies kann die allgemeine Funktionslatenz reduzieren und die Gesamtleistung der Funktionen verbessern, ohne dass Code geändert werden muss.

Sections

- Die JAVA_TOOL_OPTIONS-Umgebungsvariable verstehen
- Log4j-Patch für Log4Shell
- Ahead-of-Time (AOT) und CDS-Caches

Die **`JAVA_TOOL_OPTIONS`**-Umgebungsvariable verstehen

In Java unterstützt Lambda die Umgebungsvariable `JAVA_TOOL_OPTIONS`, um zusätzliche Befehlszeilenvariablen in Lambda festzulegen. Sie können diese Umgebungsvariable auf verschiedene Arten verwenden, z. B. um die Einstellungen für die gestufte Kompilierung anzupassen. Im nächsten Beispiel wird gezeigt, wie die Umgebungsvariable `JAVA_TOOL_OPTIONS` für diesen Anwendungsfall verwendet wird.

Beispiel: Einstellungen für die gestufte Kompilierung anpassen

Die gestufte Kompilierung ist ein Feature der virtuellen Java-Maschine (JVM). Sie können spezielle Einstellungen für die mehrstufige Kompilierung verwenden, um die Compiler der JVM just-in-time (JIT) optimal zu nutzen. In der Regel ist der C1-Compiler für eine schnelle Startzeit optimiert. Der C2-Compiler ist für die beste Gesamtleistung optimiert, benötigt aber auch mehr Speicher und es dauert länger, bis er erreicht ist. Die gestufte Kompilierung umfasst 5 verschiedene Stufen. Auf Stufe 0 interpretiert die JVM Java-Bytecode. Auf Stufe 4 verwendet die JVM den C2-Compiler, um die beim Startup der Anwendung gesammelten Profildaten zu analysieren. Im Laufe der Zeit überwacht sie die Code-Nutzung, um die besten Optimierungen zu identifizieren.

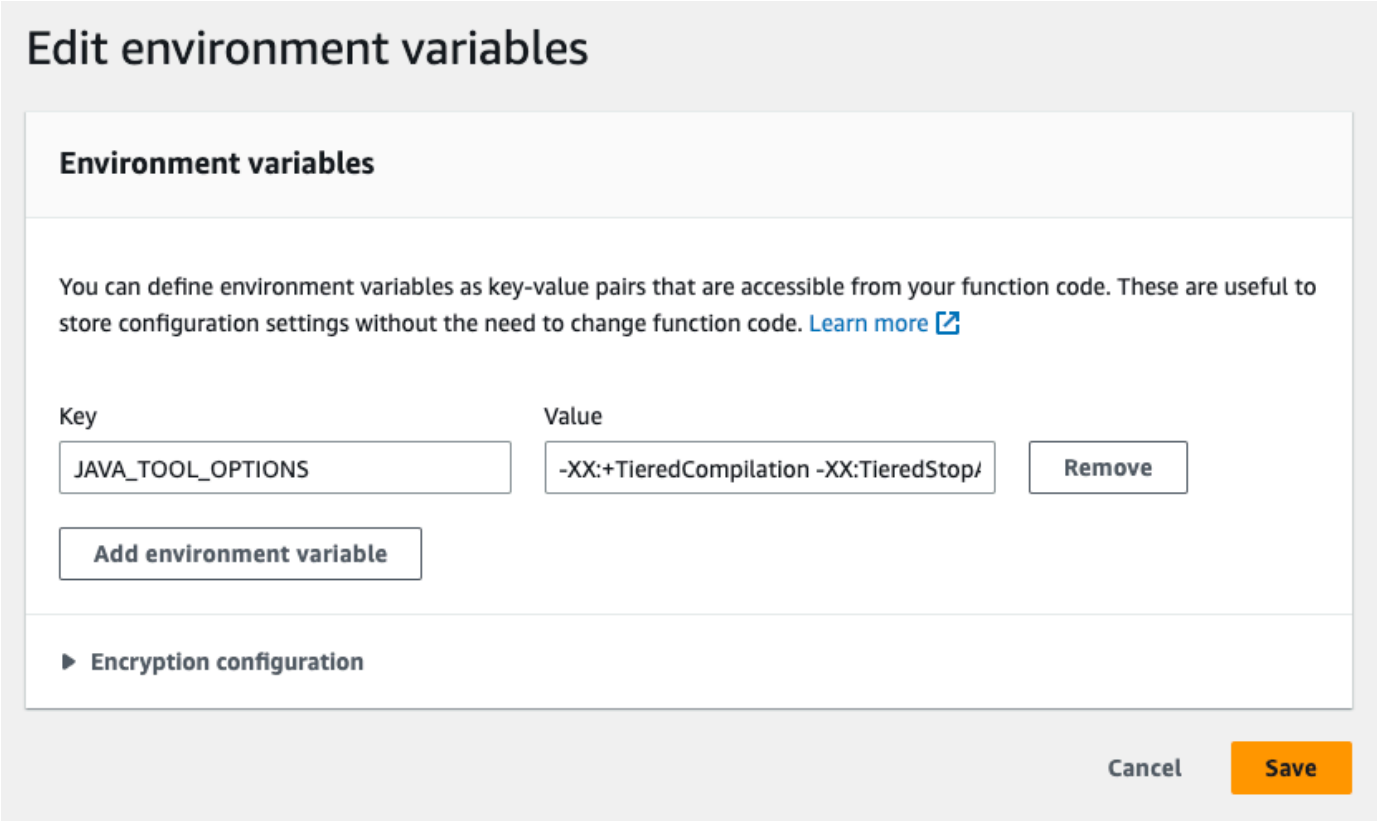
Das Anpassen der mehrstufigen Kompilierungsebene kann Ihnen helfen, die Leistung Ihrer Java-Funktionen zu optimieren. Bei kleinen Funktionen, die schnell ausgeführt werden, kann das Setzen der mehrstufigen Kompilierung auf Stufe 1 dazu beitragen, die Kaltstartleistung zu verbessern, indem die JVM den C1-Compiler verwendet. Diese Einstellung erzeugt schnell optimierten

systemeigenen Code, generiert jedoch keine Profilerstellungsdaten und verwendet niemals den C2-Compiler. Bei größeren, rechenintensiven Funktionen maximiert die Einstellung der mehrstufigen Kompilierung auf Stufe 4 die Gesamtleistung auf Kosten von zusätzlichem Speicherverbrauch und zusätzlicher Optimierungsarbeit bei den ersten Aufrufen nach der Bereitstellung jeder Lambda-Ausführungsumgebung.

Für Java 11-Laufzeiten und darunter verwendet Lambda die standardmäßigen JVM-Einstellungen für die mehrstufige Kompilierung. Für Java 17 und Java 21 konfiguriert Lambda die JVM so, dass die mehrstufige Kompilierung standardmäßig auf Stufe 1 beendet wird. Ab Java 25 stoppt Lambda die mehrstufige Kompilierung immer noch standardmäßig auf Stufe 1, außer wenn Parallelität SnapStart oder Provisioned verwendet wird. In diesem Fall werden die standardmäßigen JVM-Einstellungen verwendet. Dadurch wird die Leistung bei Parallelität SnapStart und Bereitstellung verbessert, ohne dass es zu einer Kaltstartstrafe kommt, da die gestaffelte Kompilierung in diesen Fällen außerhalb des Aufrufpfads erfolgt. Um diesen Vorteil zu maximieren, können Sie das Priming — das Ausführen von Codepfaden während der Funktionsinitialisierung — verwenden, um JIT auszulösen, bevor der SnapStart Snapshot erstellt wird, oder wenn die Ausführungsumgebungen von Provisioned Concurrency vorab bereitgestellt werden. Weitere Informationen finden Sie im Blogbeitrag Optimierung der Kaltstartleistung von AWS Lambda mithilfe fortschrittlicher Priming-Strategien mit. SnapStart

Anpassen der Einstellungen für die gestufte Kompilierung (Konsole)

1. Öffnen Sie die Seite Funktionen in der Lambda-Konsole.
2. Wählen Sie eine Java-Funktion, für die Sie die gestufte Kompilierung anpassen möchten.
3. Wählen Sie die Registerkarte Konfiguration und dann im linken Menü die Option Umgebungsvariablen aus.
4. Wählen Sie Bearbeiten aus.
5. Wählen Sie Umgebungsvariablen hinzufügen aus.
6. Geben Sie `JAVA_TOOL_OPTIONS` als Schlüssel ein. Geben Sie `-XX:+TieredCompilation -XX:TieredStopAtLevel=1` als Wert ein.



7. Wählen Sie Speichern.

> **Note**
>
> Sie können Lambda auch verwenden SnapStart , um Kaltstartprobleme zu beheben. SnapStartverwendet zwischengespeicherte Snapshots Ihrer Ausführungsumgebung, um die Startleistung erheblich zu verbessern. Weitere Informationen zu SnapStart Funktionen, Einschränkungen und unterstützten Regionen finden Sie unter. Verbesserung der Startleistung mit Lambda SnapStart

Beispiel: Anpassen des GC-Verhaltens mit JAVA_TOOL_OPTIONS

Java-11-Laufzeiten verwenden den Serial Garbage Collector (GC) für Garbage Collection. Standardmäßig verwenden Java-17-Laufzeiten auch den Serial GC. Mit Java 17 können Sie jedoch auch die Umgebungsvariable `JAVA_TOOL_OPTIONS` verwenden, um den Standard-GC zu ändern. Sie können zwischen dem Parallel GC und dem Shenandoah GC wählen.

Wenn Ihr Workload beispielsweise mehr und mehrere Speicher beansprucht CPUs, sollten Sie die Verwendung von Parallel GC in Betracht ziehen, um eine bessere Leistung zu erzielen. Hängen Sie dazu Folgendes an den Wert Ihrer Umgebungsvariablen `JAVA_TOOL_OPTIONS` an:

```
-XX:+UseParallelGC
```

Wenn Ihr Workload viele kurzlebige Objekte enthält, können Sie von einem geringeren Speicherverbrauch profitieren, indem Sie den in Java 25 eingeführten Generationsmodus des Shenandoah-Garbage-Collectors aktivieren. Hängen Sie dazu Folgendes an den Wert Ihrer Umgebungsvariablen an: `JAVA_TOOL_OPTIONS`

```
-XX:+UseShenandoahGC -XX:ShenandoahGCMode=generational
```

Log4j-Patch für Log4Shell

Die Lambda-Laufzeiten für Java 8, 11, 17 und 21 enthalten einen Patch zur Minderung der Log4Shell-Sicherheitslücke (CVE-2021-44228) in Log4j, einem beliebten Java-Logging-Framework. Dieser Patch verursacht einen Leistungsaufwand beim Kaltstart. Wenn Sie eine gepatchte Version von Log4j (Version 2.17.0 oder höher) verwenden, können Sie diesen Patch deaktivieren, um die Kaltstartleistung zu verbessern. Um den Patch zu deaktivieren, setzen Sie die Umgebungsvariable auf. `AWS_LAMBDA_DISABLE_CVE_2021_44228_PROTECTION` `true`

Ab Java 25 enthalten Lambda-Laufzeiten den Log4Shell-Patch nicht mehr. Sie müssen überprüfen, ob Sie Log4j Version 2.17.0 oder höher verwenden.

Ahead-of-Time (AOT) und CDS-Caches

Ab Java 25 umfasst die Lambda-Laufzeit einen Ahead-of-Time (AOT-) Cache für den Java Runtime Interface Client (RIC), eine Laufzeitkomponente, die aktiv nach Ereignissen von der Lambda Runtime API abfragt. Dies verbessert die Kaltstartleistung.

AOT-Caches sind spezifisch für einen JVM-Build. Wenn Lambda die verwaltete Laufzeit aktualisiert, aktualisiert es auch den AOT-Cache für das RIC. Wenn Sie jedoch Ihre eigenen AOT-Caches bereitstellen, können diese ungültig werden oder nach einem Runtime-Update zu unerwartetem Verhalten führen. Wir empfehlen daher dringend, keine AOT-Caches zu verwenden, wenn Sie verwaltete Laufzeiten verwenden. Um AOT-Caches zu verwenden, sollten Sie Ihre Funktionen mithilfe von Container-Images bereitstellen.

AOT-Caches können nicht mit Class Data Sharing (CDS) -Caches verwendet werden. Wenn Sie CDS-Caches in Ihrer Lambda-Funktion bereitstellen, deaktiviert Lambda den AOT-Cache.

Verwenden des Lambda-Kontextobjekts zum Abrufen von Java-Funktionsinformationen

Wenn Lambda Ihre Funktion ausführt, wird ein Context-Objekt an den Handler. übergeben. Dieses Objekt stellt Methoden und Eigenschaften mit Informationen zum Aufruf, zur Funktion und zur Ausführungsumgebung bereit.

Context-Methoden

- `getRemainingTimeInMillis()` – Gibt die Anzahl der verbleibenden Millisekunden zurück, bevor die Ausführung das Zeitlimit überschreitet.
- `getFunctionName()` – Gibt den Namen der Lambda-Funktion zurück.
- `getFunctionVersion()` – Gibt die Version der Funktion zurück.
- `getInvokedFunctionArn()` – Gibt den Amazon-Ressourcennamen (ARN) zurück, der zum Aufrufen der Funktion verwendet wird. Gibt an, ob der Aufrufer eine Versionsnummer oder einen Alias angegeben hat.
- `getMemoryLimitInMB()` – Gibt die Menge an Arbeitsspeicher zurück, die der Funktion zugewiesen ist.
- `getAwsRequestId()` – Gibt den Bezeichner der Aufrufanforderung zurück.
- `getLogGroupName()` – Gibt die Protokollgruppe für die Funktion zurück.
- `getLogStreamName()` – Gibt den Protokollstrom für die Funktionsinstance zurück.
- `getIdentity()` – Gibt Informationen zur Amazon-Cognito-Identität zurück, die die Anforderung autorisiert hat.
- `getClientContext()` – (mobile Apps) Gibt Clientkontext zurück, der Lambda von der Clientanwendung bereitgestellt wird.
- `getLogger()` – Gibt das Logger-Objekt für die Funktion zurück.

Das folgende Beispiel zeigt eine Funktion, die das Kontextobjekt verwendet, um auf den Lambda-Logger zuzugreifen.

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;
  }
}
```

Die Funktion protokolliert den Klassentyp des eingehenden Ereignisses, bevor sie `null` zurückgibt.

Example Protokollausgabe

```
EVENT TYPE: class java.util.LinkedHashMap
```

Die Schnittstelle für das Kontextobjekt ist in der aws-lambda-java-core-Bibliothek verfügbar. Sie können diese Schnittstelle implementieren, um eine Kontextklasse zum Testen zu erstellen. Das folgende Beispiel zeigt eine Kontextklasse, die Dummy-Werte für die meisten Eigenschaften und einen funktionierenden Testlogger zurückgibt.

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();
  }

}
```

Weitere Informationen zu Protokollierung finden Sie unter Java-Lambda-Funktionen protokollieren und überwachen.

Kontext in Beispielanwendungen

Das GitHub-Repository für dieses Handbuch enthält Beispielanwendungen, die die Verwendung des Kontextobjekts veranschaulichen. Jede Beispielanwendung enthält Skripts für die einfache Bereitstellung und Bereinigung, eine AWS Serverless Application Model (AWS SAM)-Vorlage und unterstützende Ressourcen.

Lambda-Beispielanwendungen in Java

- example-java – Eine Java-Funktion, die zeigt, wie Sie Lambda zur Bearbeitung von Aufträgen verwenden können. Diese Funktion veranschaulicht, wie Sie ein benutzerdefiniertes Eingabeereignisobjekt definieren und deserialisieren, das AWS SDK verwenden und Protokolle ausgeben.
- Java-Basis – Eine Sammlung minimaler Java-Funktionen mit Einheitentests und variabler Protokollierungskonfiguration.
- Java-Ereignisse – Eine Sammlung von Java-Funktionen, die Grundcode für den Umgang mit Ereignissen aus verschiedenen Services wie Amazon API Gateway, Amazon SQS und Amazon Kinesis enthalten. Diese Funktionen verwenden die neueste Version der aws-lambda-java-events-Bibliothek (3.0.0 und neuer). Für diese Beispiele ist das AWS SDK nicht als Abhängigkeit erforderlich.
- s3-java – Eine Java-Funktion die Benachrichtigungsereignisse aus Amazon S3 verarbeitet und die Java Class Library (JCL) verwendet, um Miniaturansichten aus hochgeladenen Image-Dateien zu erstellen.
- layer-java – Eine Java-Funktion, die veranschaulicht, wie Sie mit einer Lambda-Ebene Abhängigkeiten getrennt von Ihrem Kernfunktionscode packen können.

Java-Lambda-Funktionen protokollieren und überwachen

AWS Lambda überwacht automatisch Lambda-Funktionen und sendet Protokolleinträge an Amazon CloudWatch. Ihre Lambda-Funktion enthält eine CloudWatch Logs-Log-Gruppe und einen Log-Stream für jede Instanz Ihrer Funktion. Die Lambda-Laufzeitumgebung sendet Details zu den einzelnen Aufrufen an den Protokollstream und leitet Protokolle und andere Ausgaben aus dem Code Ihrer Funktion weiter. Weitere Informationen zu CloudWatch Logs finden Sie unterSenden von Lambda-Funktionsprotokollen an CloudWatch Logs.

Um Protokolle aus Ihrem Funktionscode auszugeben, können Sie Methoden von java.lang.System oder jedes Protokollierungsmodul verwenden, das nach stdout oder stderr schreibt.

Sections

- Erstellen einer Funktion, die Protokolle zurückgibt
- Verwenden von Lambda-Optionen für die erweiterte Protokollierung mit Java
- Implementierung der erweiterten Protokollierung mit Log4J2 und J SLF4
- Andere Tools und Bibliotheken für die Protokollierung verwenden
- Verwendung von Powertools für AWS Lambda (Java) und für strukturiertes Logging AWS SAM
- Anzeigen von Protokollen in der Lambda-Konsole
- Protokolle in der CloudWatch Konsole anzeigen
- Logs mit dem Befehl () anzeigen AWS Command Line InterfaceAWS CLI
- Löschen von Protokollen
- Beispielprotokolliercode

Erstellen einer Funktion, die Protokolle zurückgibt

Um Protokolle aus dem Code Ihrer Funktion auszugeben, können Sie Methoden für java.lang.System verwenden oder ein beliebiges Protokollierungsmodul, das in `stdout` oder `stderr` schreibt. Die aws-lambda-java-coreBibliothek stellt eine Logger-Klasse mit dem Namen bereit, auf `LambdaLogger` die Sie vom Kontextobjekt aus zugreifen können. Die Logger-Klasse unterstützt mehrzeilige Protokolle.

Im folgenden Beispiel wird der `LambdaLogger`-Logger verwendet, der vom Kontextobjekt bereitgestellt wird.

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 Protokollformat

```
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
```

Die Java-Laufzeit protokolliert die Zeilen `START`, `END` und `REPORT` für jeden Aufruf. Die Berichtszeile enthält die folgenden Details:

Datenfelder für REPORT-Zeilen

- RequestId— Die eindeutige Anforderungs-ID für den Aufruf.
- Dauer – Die Zeit, die die Handler-Methode Ihrer Funktion mit der Verarbeitung des Ereignisses verbracht hat.
- Fakturierte Dauer – Die für den Aufruf fakturierte Zeit.
- Speichergröße – Die der Funktion zugewiesene Speichermenge.
- Max. verwendeter Speicher – Die Speichermenge, die von der Funktion verwendet wird. Wenn Aufrufe eine Ausführungsumgebung gemeinsam nutzen, meldet Lambda den maximalen Speicherverbrauch für alle Aufrufe. Dieses Verhalten kann zu einem höheren als erwarteten gemeldeten Wert führen.
- Initialisierungsdauer – Für die erste Anfrage die Zeit, die zur Laufzeit zum Laden der Funktion und Ausführen von Code außerhalb der Handler-Methode benötigt wurde.
- XRAY TraceId — Für verfolgte Anfragen die AWS X-Ray Trace-ID.
- SegmentId— Für verfolgte Anfragen die X-Ray-Segment-ID.
- Stichprobe – Bei verfolgten Anforderungen das Stichprobenergebnis.

Verwenden von Lambda-Optionen für die erweiterte Protokollierung mit Java

Um Ihnen mehr Kontrolle darüber zu geben, wie die Protokolle Ihrer Funktionen erfasst, verarbeitet und verwendet werden, können Sie die folgenden Protokollierungsoptionen für unterstützte Java-Laufzeiten konfigurieren:

- Protokollformat – Wählen Sie zwischen Klartext und einem strukturierten JSON-Format für die Protokolle Ihrer Funktion aus.

- Protokollebene — für Logs im JSON-Format wählen Sie die Detailebene der Logs, an die Lambda sendet CloudWatch, wie ERROR, DEBUG oder INFO
- Protokollgruppe — wählen Sie die CloudWatch Protokollgruppe aus, an die Ihre Funktion Logs sendet

Weitere Informationen zu diesen Protokollierungsoptionen und Anweisungen zur Konfiguration Ihrer Funktion für deren Verwendung finden Sie unter the section called “Konfigurieren erweiterter Protokollierungsoptionen für Lambda-Funktionen”.

Informationen zur Verwendung der Optionen für das Protokollformat und die Protokollebene mit Ihren Java-Lambda-Funktionen finden Sie in den folgenden Abschnitten.

Verwenden des strukturierten JSON-Protokollformats mit Java

Wenn Sie JSON für das Protokollformat Ihrer Funktion auswählen, sendet Lambda die Protokollausgabe unter Verwendung der `LambdaLogger`-Klasse als strukturiertes JSON. Jedes JSON-Protokollobjekt enthält mindestens vier Schlüssel-Wert-Paare mit den folgenden Schlüsseln:

- `"timestamp"` – die Uhrzeit, zu der die Protokollmeldung generiert wurde
- `"level"` – die der Meldung zugewiesene Protokollebene
- `"message"` – der Inhalt der Protokollmeldung
- `"AWSrequestId"` – die eindeutige Anforderungs-ID für den Funktionsaufruf

Abhängig von der verwendeten Protokollierungsmethode können die im JSON-Format erfassten Protokollausgaben Ihrer Funktion auch zusätzliche Schlüssel-Wert-Paare enthalten.

Um Protokollen, die Sie mit dem `LambdaLogger`-Logger erstellen, eine Ebene zuzuweisen, müssen Sie in Ihrem Protokollierungsbefehl ein `LogLevel`-Argument angeben, wie im folgenden Beispiel gezeigt.

Example Protokollierungscode für Java

```
LambdaLogger logger = context.getLogger();
logger.log("This is a debug log", LogLevel.DEBUG);
```

Diese Protokollausgabe mit diesem Beispielcode würde wie folgt in CloudWatch Logs erfasst werden:

Example JSON-Protokolldatensatz

```
{
    "timestamp":"2023-11-01T00:21:51.358Z",
    "level":"DEBUG",
    "message":"This is a debug log",
    "AWSrequestId":"93f25699-2cbf-4976-8f94-336a0aa98c6f"
}
```

Wenn Sie Ihrer Protokollausgabe keine Ebene zuweisen, weist Lambda ihr automatisch die Ebene INFO zu.

Wenn Ihr Code bereits eine andere Protokollierungsbibliothek verwendet, um strukturierte JSON-Protokolle zu erstellen, müssen Sie keine Änderungen vornehmen. Lambda codiert Protokolle, die bereits JSON-codiert sind, nicht doppelt. Selbst wenn Sie Ihre Funktion so konfigurieren, dass sie das JSON-Protokollformat verwendet, erscheinen Ihre Logging-Ausgaben CloudWatch in der von Ihnen definierten JSON-Struktur.

Verwenden der Filterung auf Protokollebene mit Java

AWS Lambda Um Ihre Anwendungsprotokolle nach ihrer Protokollebene zu filtern, muss Ihre Funktion Protokolle im JSON-Format verwenden. Sie können dies auf zwei Arten erreichen:

- Erstellen Sie Protokollausgaben mithilfe der Standard-`LambdaLogger` und konfigurieren Sie Ihre Funktion so, dass sie die JSON-Protokollformatierung verwendet. Lambda filtert dann Ihre Protokollausgaben mithilfe des Schlüssel-Wert-Paars „level“ im JSON-Objekt, wie unter the section called “Verwenden des strukturierten JSON-Protokollformats mit Java” beschrieben. Informationen zur Konfiguration des Protokollformats Ihrer Funktion finden Sie unter the section called “Konfigurieren erweiterter Protokollierungsoptionen für Lambda-Funktionen”.
- Verwenden Sie eine andere Protokollierungsbibliothek oder Methode, um strukturierte JSON-Protokolle in Ihrem Code zu erstellen, die ein „level“-Schlüssel-Wert-Paar enthalten, das die Ebene der Protokollausgabe definiert. Sie können auch eine beliebige Protokollierungsbibliothek verwenden, die JSON-Protokolle in `stdout` oder `stderr` schreibt. Sie können beispielsweise Powertools for AWS Lambda oder das Paket Log4J2 verwenden, um strukturierte JSON-Protokollausgaben aus Ihrem Code zu generieren. Weitere Informationen dazu finden Sie unter the section called “Verwendung von Powertools für AWS Lambda (Java) und für strukturiertes Logging AWS SAM” und the section called “Implementierung der erweiterten Protokollierung mit Log4J2 und J SLF4”.

Wenn Sie Ihre Funktion so konfigurieren, dass sie Filterung auf Protokollebene verwendet, müssen Sie aus den folgenden Optionen für die Protokollebene auswählen, die Lambda an Logs senden soll: CloudWatch

Protokollebene	Standardnutzung
TRACE (am detailliertesten)	Die detailliertesten Informationen, die verwendet werden, um den Ausführungspfad Ihres Codes nachzuverfolgen
DEBUG	Detaillierte Informationen für das System-Debugging
INFO	Meldungen, die den normalen Betrieb Ihrer Funktion erfassen
WARN	Meldungen über mögliche Fehler, die zu unerwartetem Verhalten führen können, wenn sie nicht behoben werden
ERROR	Meldungen über Probleme, die verhindern, dass der Code wie erwartet funktioniert
FATAL (am wenigsten Details)	Meldungen über schwerwiegende Fehler, die dazu führen, dass die Anwendung nicht mehr funktioniert

Damit Lambda die Protokolle Ihrer Funktion filtern kann, müssen Sie auch ein `"timestamp"`-Schlüssel-Wert-Paar in Ihre JSON-Protokollausgabe aufnehmen. Die Uhrzeit muss im gültigen RFC 3339-Zeitstempelformat angegeben werden. Wenn Sie keinen gültigen Zeitstempel angeben, weist Lambda dem Protokoll die Stufe INFO zu und fügt einen Zeitstempel für Sie hinzu.

Lambda sendet Protokolle der ausgewählten Stufe und niedriger bis CloudWatch. Wenn Sie beispielsweise die Protokollebene WARN konfigurieren, sendet Lambda Protokolle, die den Stufen WARN, ERROR und FATAL entsprechen.

Implementierung der erweiterten Protokollierung mit Log4J2 und J SLF4

> **Note**
>
> AWS Lambda nimmt Log4j2 nicht in seine verwalteten Laufzeiten oder Basis-Container-Images auf. Diese sind daher von den in CVE-2021-44228, CVE-2021-45046 und CVE-2021-45105 beschriebenen Problemen nicht betroffen.
> Für Fälle, in denen eine Kundenfunktion eine betroffene Log4j2-Version enthält, haben wir eine Änderung an den verwalteten Lambda-Java-Laufzeiten und Basis-Container-Images vorgenommen, die dazu beiträgt, die Probleme in CVE-2021-44228, CVE-2021-45046 und CVE-2021-45105 zu entschärfen. Infolge dieser Änderung sehen Kunden, die Log4J2 verwenden, möglicherweise einen zusätzlichen Protokolleintrag ähnlich wie „`Transforming org/apache/logging/log4j/core/lookup/JndiLookup (java.net.URLClassLoader@...)`“. Alle Log-Strings, die in der Log4J2-Ausgabe auf den jndi-Mapper verweisen, werden durch „`Patched JndiLookup::lookup()`“ ersetzt.
> Unabhängig von dieser Änderung empfehlen wir allen Kunden, deren Funktionen Log4j2 enthalten, nachdrücklich, auf die neueste Version zu aktualisieren. Insbesondere Kunden, die die aws-lambda-java-log 4j2-Bibliothek in ihren Funktionen verwenden, sollten auf Version 1.5.0 (oder höher) aktualisieren und ihre Funktionen erneut bereitstellen. Diese Version aktualisiert die zugrunde liegenden Abhängigkeiten des Log4j2-Dienstprogramms auf Version 2.17.0 (oder höher). Die aktualisierte aws-lambda-java-log 4j2-Binärdatei ist im Maven-Repository verfügbar und der Quellcode ist auf Github verfügbar.
> Beachten Sie abschließend, dass Bibliotheken, die sich auf aws-lambda-java-log4j (v1.0.0 oder 1.0.1) beziehen, unter keinen Umständen verwendet werden sollten. Diese Bibliotheken beziehen sich auf Version 1.x von log4j, deren Nutzungsdauer 2015 abgelaufen ist. Die Bibliotheken werden nicht unterstützt, nicht gewartet, nicht gepatcht und haben bekannte Sicherheitslücken.

Um die Protokollausgabe anzupassen, die Protokollierung bei Komponententests zu unterstützen und AWS SDK-Aufrufe zu protokollieren, verwenden Sie Apache Log4j2 mit SLF4 J. Log4j ist eine Logging-Bibliothek für Java-Programme, mit der Sie Protokollebenen konfigurieren und Appender-Bibliotheken verwenden können. SLF4J ist eine Fassadenbibliothek, mit der Sie ändern können, welche Bibliothek Sie verwenden, ohne Ihren Funktionscode zu ändern.

Verwenden Sie den Appender in der aws-lambda-java-log4j2-Bibliothek, um die Anforderungs-ID zu den Protokollen Ihrer Funktion hinzuzufügen.

Example src/main/resources/log4j2.xml — Appender-Konfiguration

```
<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>
```

Sie können entscheiden, wie Ihre Log4j2-Protokolle entweder für Klartext- oder JSON-Ausgaben konfiguriert werden sollen, indem Sie unter den Tags `<LambdaTextFormat>` und `<LambdaJSONFormat>` ein Layout angeben.

In diesem Beispiel wird im Textmodus jeder Zeile das Datum, die Uhrzeit, die Anforderungs-ID, die Protokollstufe und der Klassenname vorangestellt. Im JSON-Modus wird das `<JsonTemplateLayout>` mit einer Konfiguration verwendet, die zusammen mit der `aws-lambda-java-log4j2`-Bibliothek geliefert wird.

SLF4J ist eine Fassadenbibliothek zum Einloggen in Java-Code. In Ihrem Funktionscode verwenden Sie die SLF4 J Logger Factory, um einen Logger mit Methoden für Log-Levels wie `info()` und abzurufen`warn()`. In Ihrer Build-Konfiguration nehmen Sie die Logging-Bibliothek und den SLF4 J-Adapter in den Klassenpfad auf. Indem Sie die Bibliotheken in der Build-Konfiguration ändern, können Sie den Logger-Typ ändern, ohne Ihren Funktionscode zu ändern. SLF4J ist erforderlich, um Protokolle aus dem SDK for Java zu erfassen.

Im folgenden Beispielcode verwendet die Handler-Klasse SLF4 J, um einen Logger abzurufen.

Example src/main/java/example/HandlerS3.java — Protokollierung mit 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";
    }
}
```

Es wird eine ähnliche Protokollausgabe wie die folgende erstellt:

Example Protokollformat

```
{
    "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/"
}
```

Die Build-Konfiguration verwendet Laufzeitabhängigkeiten vom Lambda-Appender und SLF4 J-Adapter sowie Implementierungsabhängigkeiten von Log4j2.

Example build.gradle – Protokollierungsabhängigkeiten

```
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,)',
    ...
}
```

Wenn Sie Ihren Code vor Ort für Tests ausführen, ist das Kontextobjekt mit dem Lambda-Logger nicht verfügbar, und es gibt keine Anforderungs-ID, die der Lambda-Appender verwenden kann. Beispiele für Testkonfigurationen finden Sie in den Beispielanwendungen im nächsten Abschnitt.

Andere Tools und Bibliotheken für die Protokollierung verwenden

Powertools for AWS Lambda (Java) ist ein Entwickler-Toolkit zur Implementierung von Best Practices für Serverless und zur Steigerung der Entwicklergeschwindigkeit. Das Logging-Serviceprogramm bietet einen für Lambda optimierten Logger, der zusätzliche Informationen zum Funktionskontext all Ihrer Funktionen enthält, wobei die Ausgabe als JSON strukturiert ist. Mit diesem Serviceprogramm können Sie Folgendes tun:

- Erfassung von Schlüsselfeldern aus dem Lambda-Kontext, Kaltstart und Strukturen der Protokollierungsausgabe als JSON
- Protokollieren Sie Ereignisse von Lambda-Aufrufen, wenn Sie dazu aufgefordert werden (standardmäßig deaktiviert)
- Alle Protokolle nur für einen bestimmten Prozentsatz der Aufrufe über Protokollstichproben drucken (standardmäßig deaktiviert)
- Fügen Sie dem strukturierten Protokoll zu einem beliebigen Zeitpunkt zusätzliche Schlüssel hinzu

- Verwenden Sie einen benutzerdefinierten Protokollformatierer (Bring Your Own Formatter), um Protokolle in einer Struktur auszugeben, die mit dem Logging RFC Ihres Unternehmens kompatibel ist

Verwendung von Powertools für AWS Lambda (Java) und für strukturiertes Logging AWS SAM

Gehen Sie wie folgt vor, um eine Hello World Java-Beispielanwendung mit integrierten Powertools für AWS Lambda (Java) -Modulen herunterzuladen, zu erstellen und bereitzustellen. Verwenden Sie dazu den. AWS SAM Diese Anwendung implementiert ein grundlegendes API-Backend und verwendet Powertools zum Ausgeben von Protokollen, Metriken und Traces. Es besteht aus einem Amazon-API-Gateway-Endpunkt und einer Lambda-Funktion. Wenn Sie eine GET-Anfrage an den API-Gateway-Endpunkt senden, ruft die Lambda-Funktion Logs und Metriken auf, sendet sie im Embedded Metric Format an CloudWatch und sendet Traces an. AWS X-Ray Die Funktion gibt eine `hello world`-Nachricht zurück.

Voraussetzungen

Zur Durchführung der Schritte in diesem Abschnitt benötigen Sie Folgendes:

- Java 11
- AWS CLI Version 2
- AWS SAM CLI Version 1.75 oder höher. Wenn Sie eine ältere Version der AWS SAM CLI haben, finden Sie weitere Informationen unter Upgrade der AWS SAM CLI.

Stellen Sie eine AWS SAM Beispielanwendung bereit

1. Initialisieren Sie die Anwendung mit der Hello World Java-Vorlage.

```
sam init --app-template hello-world-powertools-java --name sam-app --package-type
 Zip --runtime java11 --no-tracing
```

2. Entwickeln Sie die App.

```
cd sam-app && sam build
```

3. Stellen Sie die Anwendung bereit.

```
sam deploy --guided
```

4. Folgen Sie den Anweisungen auf dem Bildschirm. Um die im interaktiven Erlebnis bereitgestellten Standardoptionen zu akzeptieren, drücken Sie `Enter`.

> **Note**
>
> Für ist HelloWorldFunction möglicherweise keine Autorisierung definiert. Ist das in Ordnung? , stellen Sie sicher, dass Sie eintreteny.

5. Rufen Sie die URL der bereitgestellten Anwendung ab:

```
aws cloudformation describe-stacks --stack-name sam-app --query
 'Stacks[0].Outputs[?OutputKey==`HelloWorldApi`].OutputValue' --output text
```

6. Rufen Sie den API-Endpunkt auf:

```
curl -X GET <URL_FROM_PREVIOUS_STEP>
```

Wenn der Link erfolgreich ausgeführt wurde, sehen Sie die folgende Antwort:

```
{"message":"hello world"}
```

7. Führen Sie sam logs aus, um die Protokolle für die Funktion abzurufen. Weitere Informationen finden Sie unter Arbeiten mit Protokollen im AWS Serverless Application Model - Entwicklerhandbuch.

```
sam logs --stack-name sam-app
```

Das Ergebnis sieht folgendermaßen aus:

```
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. Dies ist ein öffentlicher API-Endpunkt, der über das Internet zugänglich ist. Es wird empfohlen, dass Sie den Endpunkt nach dem Testen löschen.

```
sam delete
```

Verwalten der Protokollaufbewahrung

Wenn Sie eine Funktion löschen, werden Protokollgruppen nicht automatisch gelöscht. Um zu vermeiden, dass Protokolle auf unbestimmte Zeit gespeichert werden, löschen Sie die Protokollgruppe oder konfigurieren Sie einen Aufbewahrungszeitraum, nach dessen Ablauf die Protokolle CloudWatch automatisch gelöscht werden. Um die Aufbewahrung von Protokollen einzurichten, fügen Sie Ihrer AWS SAM Vorlage Folgendes hinzu:

```
Resources:
  HelloWorldFunction:
    Type: AWS::Serverless::Function
    Properties:
      # Omitting other properties

  LogGroup:
    Type: AWS::Logs::LogGroup
    Properties:
      LogGroupName: !Sub "/aws/lambda/${HelloWorldFunction}"
      RetentionInDays: 7
```

Anzeigen von Protokollen in der Lambda-Konsole

Sie können die Lambda-Konsole verwenden, um die Protokollausgabe nach dem Aufrufen einer Lambda-Funktion anzuzeigen.

Wenn Ihr Code über den eingebetteten Code-Editor getestet werden kann, finden Sie Protokolle in den Ausführungsergebnissen. Wenn Sie das Feature Konsolentest verwenden, um eine Funktion aufzurufen, finden Sie die Protokollausgabe im Abschnitt Details.

Protokolle in der CloudWatch Konsole anzeigen

Sie können die CloudWatch Amazon-Konsole verwenden, um Protokolle für alle Lambda-Funktionsaufrufe anzuzeigen.

Um Protokolle auf der Konsole anzuzeigen CloudWatch

1. Öffnen Sie die Seite Protokollgruppen auf der CloudWatch Konsole.
2. Wählen Sie die Protokollgruppe für Ihre Funktion (***your-function-name***/aws/lambda/).
3. Wählen Sie eine Protokollstream aus.

Jeder Protokoll-Stream entspricht einer Instance Ihrer Funktion. Ein Protokollstream wird angezeigt, wenn Sie Ihre Lambda-Funktion aktualisieren und wenn zusätzliche Instanzen für gleichzeitige Aufrufe erstellt werden. Um Protokolle für einen bestimmten Aufruf zu finden, empfehlen wir, Ihre Funktion mit zu instrumentieren. AWS X-Ray X-Ray erfasst Details zu der Anforderung und dem Protokollstream in der Trace.

Logs mit dem Befehl () anzeigen AWS Command Line InterfaceAWS CLI

Das AWS CLI ist ein Open-Source-Tool, mit dem Sie mithilfe von Befehlen in Ihrer Befehlszeilen-Shell mit AWS Diensten interagieren können. Zur Durchführung der Schritte in diesem Abschnitt benötigen Sie AWS CLI Version 2.

Sie können die AWS CLI verwenden, um Protokolle für einen Aufruf mit der `--log-type`-Befehlsoption abzurufen. Die Antwort enthält das Feld `LogResult`, das bis zu 4 KB base64-verschlüsselte Protokolle aus dem Aufruf enthält.

Example eine Log-ID abrufen

Das folgende Beispiel zeigt, wie eine Protokoll-ID aus dem `LogResult`-Feld für eine Funktion namens `my-function` abgerufen wird.

```
aws lambda invoke --function-name my-function out --log-type Tail
```

Die Ausgabe sollte folgendermaßen aussehen:

```
{
    "StatusCode": 200,
    "LogResult":
 "U1RBUlQgUmVxdWVzdElkOiA4N2QwNDRiOC1mMTU0LTExZTgtOGNkYS0yOTc0YzVlNGZiMjEgVmVyc2lvb...",
```

```
    "ExecutedVersion": "$LATEST"
}
```

Example entschlüsseln der Protokolle

Verwenden Sie in derselben Eingabeaufforderung das `base64`-Dienstprogramm, um die Protokolle zu entschlüsseln. Das folgende Beispiel zeigt, wie Base64-codierte Logs für abgerufen werde `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
```

Die cli-binary-format Option ist erforderlich, wenn Sie AWS CLI Version 2 verwenden. Um dies zur Standardeinstellung zu machen, führen Sie `aws configure set cli-binary-format raw-in-base64-out` aus. Weitere Informationen finden Sie unter Von AWS CLI unterstützte globale Befehlszeilenoptionen im AWS Command Line Interface -Benutzerhandbuch für Version 2.

Die Ausgabe sollte folgendermaßen aussehen:

```
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
```

Das `base64`-Dienstprogramm ist unter Linux, macOS und Ubuntu auf Windowsverfügbar. macOS-Benutzer müssen möglicherweise `base64 -D` verwenden.

Example get-logs.sh-Skript

Verwenden Sie in derselben Eingabeaufforderung das folgende Skript, um die letzten fünf Protokollereignisse herunterzuladen. Das Skript verwendet `sed` zum Entfernen von Anführungszeichen aus der Ausgabedatei und wechselt 15 Sekunden lang in den Ruhezustand, um Zeit einzuräumen, damit Protokolle verfügbar werden können. Die Ausgabe enthält die Antwort von Lambda und die `get-log-events`Ausgabe des Befehls.

Kopieren Sie den Inhalt des folgenden Codebeispiels und speichern Sie es in Ihrem Lambda-Projektverzeichnis unter `get-logs.sh`.

Die cli-binary-format Option ist erforderlich, wenn Sie AWS CLI Version 2 verwenden. Um dies zur Standardeinstellung zu machen, führen Sie `aws configure set cli-binary-format raw-in-base64-out` aus. Weitere Informationen finden Sie unter Von AWS CLI unterstützte globale Befehlszeilenoptionen im AWS Command Line Interface -Benutzerhandbuch für 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 und Linux (nur diese Systeme)

In derselben Eingabeaufforderung müssen macOS- und Linux-Benutzer möglicherweise den folgenden Befehl ausführen, um sicherzustellen, dass das Skript ausführbar ist.

```
chmod -R 755 get-logs.sh
```

Example die letzten fünf Protokollereignisse abrufen

Führen Sie an derselben Eingabeaufforderung das folgende Skript aus, um die letzten fünf Protokollereignisse abzurufen.

```
./get-logs.sh
```

Die Ausgabe sollte folgendermaßen aussehen:

```
{
    "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"
}
```


Löschen von Protokollen

Wenn Sie eine Funktion löschen, werden Protokollgruppen nicht automatisch gelöscht. Um das unbegrenzte Speichern von Protokollen zu vermeiden, löschen Sie die Protokollgruppe oder konfigurieren Sie eine Aufbewahrungszeitraum nach dem Protokolle automatisch gelöscht werden.

Beispielprotokolliercode

Das GitHub Repository für dieses Handbuch enthält Beispielanwendungen, die die Verwendung verschiedener Protokollierungskonfigurationen demonstrieren. Jede Beispielanwendung enthält Skripts für die einfache Bereitstellung und Bereinigung, eine AWS SAM Vorlage und unterstützende Ressourcen.

Lambda-Beispielanwendungen in Java

- example-java – Eine Java-Funktion, die zeigt, wie Sie Lambda zur Bearbeitung von Aufträgen verwenden können. Diese Funktion veranschaulicht, wie Sie ein benutzerdefiniertes Eingabeereignisobjekt definieren und deserialisieren, das AWS SDK verwenden und die Ausgabeprotokollierung durchführen.
- Java-Basis – Eine Sammlung minimaler Java-Funktionen mit Einheitentests und variabler Protokollierungskonfiguration.
- Java-Ereignisse – Eine Sammlung von Java-Funktionen, die Grundcode für den Umgang mit Ereignissen aus verschiedenen Services wie Amazon API Gateway, Amazon SQS und Amazon Kinesis enthalten. Diese Funktionen verwenden die neueste Version der aws-lambda-java-eventsBibliothek (3.0.0 und neuer). Für diese Beispiele ist das AWS SDK nicht als Abhängigkeit erforderlich.
- s3-java – Eine Java-Funktion die Benachrichtigungsereignisse aus Amazon S3 verarbeitet und die Java Class Library (JCL) verwendet, um Miniaturansichten aus hochgeladenen Image-Dateien zu erstellen.
- layer-java – Eine Java-Funktion, die veranschaulicht, wie Sie mit einer Lambda-Ebene Abhängigkeiten getrennt von Ihrem Kernfunktionscode packen können.

Die `java-basic`-Beispielanwendung zeigt eine minimale Protokollierungskonfiguration, die Protokollierungstests unterstützt. Der Handler-Code verwendet den `LambdaLogger`-Logger, der vom Kontextobjekt bereitgestellt wird. Für Tests verwendet die Anwendung eine benutzerdefinierte `TestLogger`-Klasse, die die `LambdaLogger`-Schnittstelle mit einem Log4j2-Logger implementiert. Aus Gründen der Kompatibilität mit dem AWS SDK wird SLF4 J als Fassade verwendet. Protokollbibliotheken werden von der Build-Ausgabe ausgeschlossen, um das Bereitstellungspaket klein zu halten.

Instrumentierung von Java-Code in AWS Lambda

Lambda lässt sich integrieren AWS X-Ray , um Ihnen zu helfen, Lambda-Anwendungen zu verfolgen, zu debuggen und zu optimieren. Sie können mit X-Ray eine Anforderung verfolgen, während sie Ressourcen in Ihrer Anwendung durchläuft, die Lambda-Funktionen und andere AWS -Services enthalten können.

Um Protokollierungsdaten an X-Ray zu senden, können Sie eine von zwei SDK-Bibliotheken verwenden:

- AWS Distro for OpenTelemetry (ADOT) — Eine sichere, produktionsbereite und AWS unterstützte Distribution des () SDK. OpenTelemetry OTel
- AWS X-Ray SDK für Java – Ein SDK zum Generieren und Senden von Nachverfolgungsdaten an X-Ray.
- Powertools for AWS Lambda (Java) — Ein Entwickler-Toolkit zur Implementierung von Best Practices für Serverless und zur Steigerung der Entwicklergeschwindigkeit.

Jedes SDKs bietet Möglichkeiten, Ihre Telemetriedaten an den X-Ray-Dienst zu senden. Sie können dann mit X-Ray die Leistungsmetriken Ihrer Anwendung anzeigen, filtern und erhalten, um Probleme und Möglichkeiten zur Optimierung zu identifizieren.

> **Important**
>
> X-Ray und Powertools für AWS Lambda SDKs sind Teil einer eng integrierten Instrumentierungslösung, die von AWS angeboten wird. Die ADOT Lambda Layers sind Teil eines branchenweiten Standards für die Verfolgung von Instrumenten, die im Allgemeinen mehr Daten erfassen, aber möglicherweise nicht für alle Anwendungsfälle geeignet sind. Sie können die end-to-end Ablaufverfolgung in X-Ray mit beiden Lösungen implementieren. Weitere Informationen zur Auswahl zwischen beiden finden Sie unter Wählen zwischen der AWS Distribution für Open Telemetry und X-Ray. SDKs

Sections

- Verwendung von Powertools für AWS Lambda (Java) und für die Nachverfolgung AWS SAM
- Verwendung von Powertools für AWS Lambda (Java) und AWS CDK für die Ablaufverfolgung
- Verwenden von ADOT zur Instrumentierung Ihrer Java-Funktionen

- Instrumentieren Sie mit dem X-Ray-SDK Ihre Java-Funktionen
- Aktivieren der Nachverfolgung mit der Lambda-Konsole
- Aktivieren der Nachverfolgung mit der Lambda-API
- Die Ablaufverfolgung wird aktiviert mit CloudFormation
- Interpretieren einer X-Ray-Nachverfolgung
- Laufzeitabhängigkeiten in einer Ebene speichern (X-Ray-SDK)
- X-Ray-Nachverfolgung in Beispielanwendungen (X-Ray-SDK)

Verwendung von Powertools für AWS Lambda (Java) und für die Nachverfolgung AWS SAM

Gehen Sie wie folgt vor, um eine Hello World Java-Beispielanwendung mit integrierten Powertools für AWS Lambda (Java) -Modulen herunterzuladen, zu erstellen und bereitzustellen. Verwenden Sie dazu den. AWS SAM Diese Anwendung implementiert ein grundlegendes API-Backend und verwendet Powertools zum Ausgeben von Protokollen, Metriken und Traces. Es besteht aus einem Amazon-API-Gateway-Endpunkt und einer Lambda-Funktion. Wenn Sie eine GET-Anfrage an den API-Gateway-Endpunkt senden, ruft die Lambda-Funktion Logs und Metriken auf, sendet sie im Embedded Metric Format an CloudWatch und sendet Traces an. AWS X-Ray Die Funktion gibt eine `hello world`-Nachricht zurück.

Voraussetzungen

Zur Durchführung der Schritte in diesem Abschnitt benötigen Sie Folgendes:

- Java 11 oder höher
- AWS CLI Version 2
- AWS SAM CLI Version 1.75 oder höher. Wenn Sie eine ältere Version der AWS SAM CLI haben, finden Sie weitere Informationen unter Upgrade der AWS SAM CLI.

Stellen Sie eine AWS SAM Beispielanwendung bereit

1. Initialisieren Sie die Anwendung mit der Hello World Java-Vorlage.

```
sam init --app-template hello-world-powertools-java --name sam-app --package-type
 Zip --runtime java11 --no-tracing
```

2. Entwickeln Sie die App.

```
cd sam-app && sam build
```

3. Stellen Sie die Anwendung bereit.

```
sam deploy --guided
```

4. Folgen Sie den Anweisungen auf dem Bildschirm. Um die im interaktiven Erlebnis bereitgestellten Standardoptionen zu akzeptieren, drücken Sie `Enter`.

> **Note**
>
> Für ist HelloWorldFunction möglicherweise keine Autorisierung definiert. Ist das in Ordnung? , stellen Sie sicher, dass Sie eintreteny.

5. Rufen Sie die URL der bereitgestellten Anwendung ab:

```
aws cloudformation describe-stacks --stack-name sam-app --query
 'Stacks[0].Outputs[?OutputKey==`HelloWorldApi`].OutputValue' --output text
```

6. Rufen Sie den API-Endpunkt auf:

```
curl -X GET <URL_FROM_PREVIOUS_STEP>
```

Wenn der Link erfolgreich ausgeführt wurde, sehen Sie die folgende Antwort:

```
{"message":"hello world"}
```

7. Führen Sie sam traces aus, um die Traces für die Funktion zu erhalten.

```
sam traces
```

Das Nachverfolgungsergebnis sieht folgendermaßen aus:

```
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. Dies ist ein öffentlicher API-Endpunkt, der über das Internet zugänglich ist. Es wird empfohlen, dass Sie den Endpunkt nach dem Testen löschen.

```
sam delete
```

Verwendung von Powertools für AWS Lambda (Java) und AWS CDK für die Ablaufverfolgung

Gehen Sie wie folgt vor, um eine Hello World Java-Beispielanwendung mit integrierten Powertools for AWS Lambda (Java) -Modulen herunterzuladen, zu erstellen und bereitzustellen. Verwenden Sie dazu den. AWS CDK Diese Anwendung implementiert ein grundlegendes API-Backend und verwendet Powertools zum Ausgeben von Protokollen, Metriken und Traces. Es besteht aus einem Amazon-API-Gateway-Endpunkt und einer Lambda-Funktion. Wenn Sie eine GET-Anfrage an den API-Gateway-Endpunkt senden, ruft die Lambda-Funktion Logs und Metriken auf, sendet sie im Embedded Metric Format an CloudWatch und sendet Traces an. AWS X-Ray Die Funktion gibt eine Hello-World-Nachricht zurück.

Voraussetzungen

Zur Durchführung der Schritte in diesem Abschnitt benötigen Sie Folgendes:

- Java 11 oder höher
- AWS CLI Version 2
- AWS CDK Ausführung 2
- AWS SAM CLI Version 1.75 oder höher. Wenn Sie eine ältere Version der AWS SAM CLI haben, finden Sie weitere Informationen unter Upgrade der AWS SAM CLI.

Stellen Sie eine AWS CDK Beispielanwendung bereit

1. Erstellen Sie ein Projektverzeichnis für Ihre neue Anwendung.

```
mkdir hello-world
cd hello-world
```

2. Initialisieren Sie die App.

```
cdk init app --language java
```

3. Erstellen Sie ein Maven-Projekt mit dem folgenden Befehl:

```
mkdir app
cd app
mvn archetype:generate -DgroupId=helloworld -DartifactId=Function -
DarchetypeArtifactId=maven-archetype-quickstart -DinteractiveMode=false
```

4. Öffnen Sie `pom.xml` im `hello-world\app\Function`-Verzeichnis und ersetzen Sie den vorhandenen Code durch den folgenden Code, der Abhängigkeiten und Maven-Plugins für Powertools enthält.

```
<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. Erstellen Sie das `hello-world\app\src\main\resource`-Verzeichnis und erstellen Sie `log4j.xml` für die Protokollkonfiguration.

```
mkdir -p src/main/resource
cd src/main/resource
touch log4j.xml
```

6. Öffnen Sie `log4j.xml` und fügen Sie den folgenden Code hinzu.

```
<?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. Öffnen Sie `App.java` aus dem `hello-world\app\Function\src\main\java\helloworld`-Verzeichnis und ersetzen Sie den vorhandenen Code durch den folgenden Code. Dies ist der Code für die Lambda-Funktion.

```
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. Öffnen Sie `HelloWorldStack.java` aus dem `hello-world\src\main\java\com\myorg`-Verzeichnis und ersetzen Sie den vorhandenen Code durch den folgenden Code. Dieser Code verwendet Lambda Constructor und den ApiGatewayv2 Constructor, um eine REST-API und eine Lambda-Funktion zu erstellen.

```
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. Öffnen Sie `pom.xml` aus dem `hello-world`-Verzeichnis und ersetzen Sie den vorhandenen Code durch den folgenden Code.

```
<?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. Stellen Sie sicher, dass Sie sich im `hello-world`-Verzeichnis befinden, und stellen Sie Ihre Anwendung bereit.

```
cdk deploy
```

11. Rufen Sie die URL der bereitgestellten Anwendung ab:

```
aws cloudformation describe-stacks --stack-name HelloWorldStack --query
 'Stacks[0].Outputs[?OutputKey==`HttpApi`].OutputValue' --output text
```

12. Rufen Sie den API-Endpunkt auf:

```
curl -X GET <URL_FROM_PREVIOUS_STEP>
```

Wenn der Link erfolgreich ausgeführt wurde, sehen Sie die folgende Antwort:

```
{"message":"hello world"}
```

13. Führen Sie sam traces aus, um die Traces für die Funktion zu erhalten.

```
sam traces
```

Das Nachverfolgungsergebnis sieht folgendermaßen aus:

```
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. Dies ist ein öffentlicher API-Endpunkt, der über das Internet zugänglich ist. Es wird empfohlen, dass Sie den Endpunkt nach dem Testen löschen.

```
cdk destroy
```

Verwenden von ADOT zur Instrumentierung Ihrer Java-Funktionen

ADOT bietet vollständig verwaltete Lambda-Schichten, die alles zusammenfassen, was Sie für die Erfassung von Telemetriedaten mithilfe des SDK benötigen. OTel Indem Sie diese Ebene verwenden, können Sie Ihre Lambda-Funktionen instrumentieren, ohne einen Funktionscode ändern zu müssen. Sie können Ihren Layer auch so konfigurieren, dass er eine benutzerdefinierte Initialisierung von durchführt. OTel Weitere Informationen finden Sie unter Benutzerdefinierte Konfiguration für den ADOT Collector auf Lambda in der ADOT-Dokumentation.

Für Java-Laufzeiten können Sie zwischen zwei Ebenen wählen, die verbraucht werden sollen:

- AWS verwaltete Lambda-Schicht für ADOT Java (Auto-Instrumentation Agent) — Diese Schicht transformiert Ihren Funktionscode beim Start automatisch, um Tracing-Daten zu sammeln. Detaillierte Anweisungen, wie Sie diese Ebene zusammen mit dem ADOT-Java-Agenten verwenden können, finden Sie in der ADOT-Dokumentation unter AWS Distro for OpenTelemetry Lambda Support for Java (Auto-Instrumentation Agent).
- AWS verwaltete Lambda-Schicht für ADOT Java — Diese Schicht bietet auch eine integrierte Instrumentierung für Lambda-Funktionen, erfordert jedoch einige manuelle Codeänderungen, um das SDK zu initialisieren. OTel Eine ausführliche Anleitung zur Nutzung dieser Ebene finden Sie unter AWS Distro for OpenTelemetry Lambda Support for Java in der ADOT-Dokumentation.

Instrumentieren Sie mit dem X-Ray-SDK Ihre Java-Funktionen

Um Daten über Aufrufe Ihrer Funktion an andere Ressourcen und Services in Ihrer Anwendung aufzuzeichnen, können Sie Ihrer Build-Konfiguration das X-Ray-SDK für Java hinzufügen. Das folgende Beispiel zeigt eine Gradle-Build-Konfiguration, die die Bibliotheken enthält, die die automatische Instrumentierung von Clients aktivieren. AWS SDK for Java 2.x

Example build.gradle – Ablaufverfolgung von Abhängigkeiten

```
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'
    ...
}
```

Aktivieren Sie nach Hinzufügen der richtigen Abhängigkeiten die Nachverfolgung in der Konfiguration Ihrer Funktion über die Lambda-Konsole oder die API.

Aktivieren der Nachverfolgung mit der Lambda-Konsole

Gehen Sie folgendermaßen vor, um die aktive Nachverfolgung Ihrer Lambda-Funktion mit der Konsole umzuschalten:

So aktivieren Sie die aktive Nachverfolgung

1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.
2. Wählen Sie eine Funktion aus.
3. Wählen Sie Configuration (Konfiguration) und dann Monitoring and operations tools (Überwachungs- und Produktionstools).
4. Wählen Sie unter Zusätzliche Überwachungstools die Option Bearbeiten aus.
5. Wählen Sie unter CloudWatch Anwendungssignale und AWS X-Ray die Option Enable for Lambda Service Traces aus.
6. Wählen Sie Speichern.

Aktivieren der Nachverfolgung mit der Lambda-API

Konfigurieren Sie die Ablaufverfolgung für Ihre Lambda-Funktion mit dem AWS CLI oder AWS SDK und verwenden Sie die folgenden API-Operationen:

- UpdateFunctionConfiguration
- GetFunctionConfiguration
- CreateFunction

Der folgende AWS CLI Beispielbefehl aktiviert die aktive Ablaufverfolgung für eine Funktion namens my-function.

```
aws lambda update-function-configuration --function-name my-function \
--tracing-config Mode=Active
```

Der Ablaufverfolgungsmodus ist Teil der versionsspezifischen Konfiguration, wenn Sie eine Version Ihrer Funktion veröffentlichen. Sie können den Ablaufverfolgungsmodus für eine veröffentlichte Version nicht ändern.

Die Ablaufverfolgung wird aktiviert mit CloudFormation

Um die Ablaufverfolgung für eine `AWS::Lambda::Function` Ressource in einer CloudFormation Vorlage zu aktivieren, verwenden Sie die `TracingConfig` Eigenschaft.

Example function-inline.yml – Ablaufverfolgungskonfiguration

```
Resources:
  function:
    Type: AWS::Lambda::Function
    Properties:
      TracingConfig:
        Mode: Active
      ...
```

Verwenden Sie für eine `AWS::Serverless::Function` Ressource AWS Serverless Application Model (AWS SAM) die `Tracing` Eigenschaft.

Example template.yml – Ablaufverfolgungskonfiguration

```
Resources:
```

```
  function:
    Type: AWS::Serverless::Function
    Properties:
      Tracing: Active
      ...
```

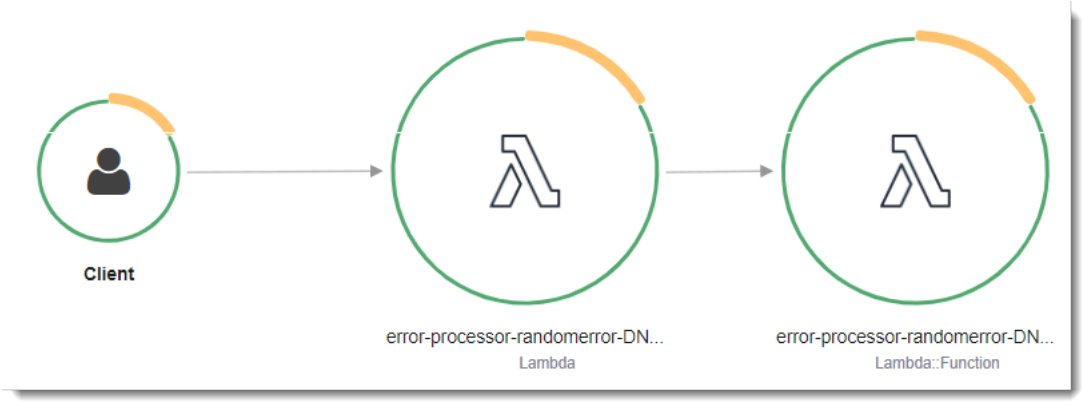
Interpretieren einer X-Ray-Nachverfolgung

Ihre Funktion benötigt die Berechtigung zum Hochladen von Trace-Daten zu X-Ray. Wenn Sie die aktive Nachverfolgung in der Lambda-Konsole aktivieren, fügt Lambda der Ausführungsrolle Ihrer Funktion die erforderlichen Berechtigungen hinzu. Andernfalls fügen Sie die AWSXRayDaemonWriteAccessRichtlinie der Ausführungsrolle hinzu.

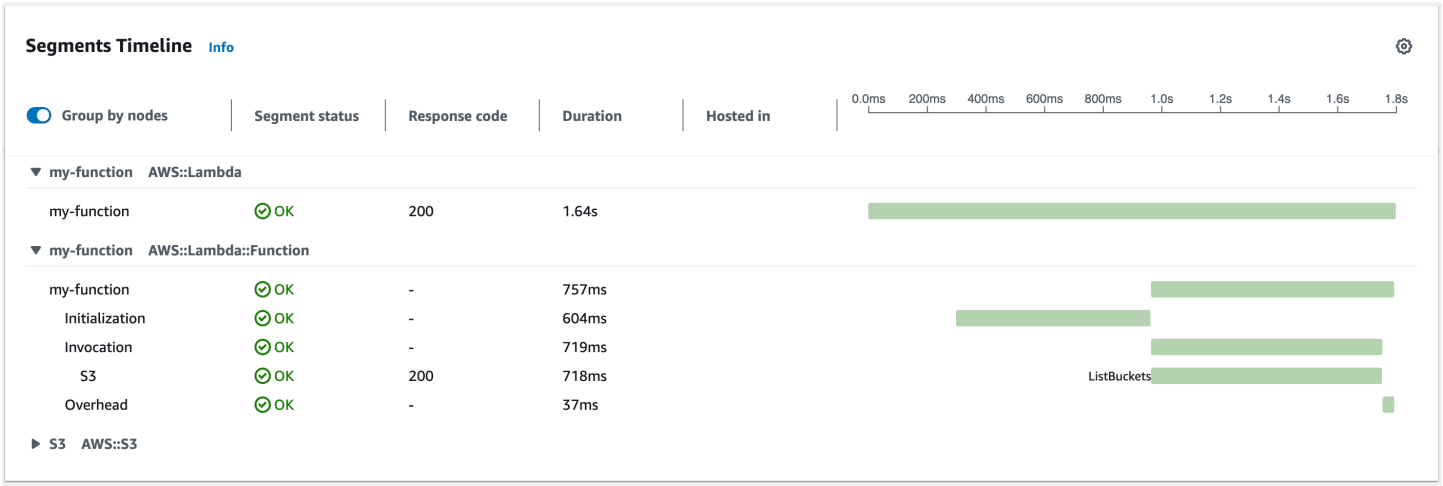
Nachdem Sie die aktive Nachverfolgung konfiguriert haben, können Sie bestimmte Anfragen über Ihre Anwendung beobachten. Das X-Ray-Service-Diagramm zeigt Informationen über Ihre Anwendung und alle ihre Komponenten an. Das folgende Beispiel zeigt eine Anwendung mit zwei Funktionen. Die primäre Funktion verarbeitet Ereignisse und gibt manchmal Fehler zurück. Die zweite Funktion an oberster Stelle verarbeitet Fehler, die in der Protokollgruppe der ersten auftreten, und verwendet das AWS SDK, um X-Ray, Amazon Simple Storage Service (Amazon S3) und Amazon CloudWatch Logs aufzurufen.

X-Ray verfolgt nicht alle Anfragen an Ihre Anwendung nach. X-Ray wendet einen Sampling-Algorithmus an, um sicherzustellen, dass die Nachverfolgung effizient ist, und stellt dennoch ein repräsentatives Beispiel aller Anfragen bereit. Die Samplingrate beträgt 1 Anforderung pro Sekunde und 5 Prozent aller weiteren Anforderungen. Sie können die X-Ray-Samplingrate nicht für Ihre Funktionen konfigurieren.

In X-Ray, zeichnet eine Ablaufverfolgung Informationen zu einer Anforderung auf, die von einem oder mehreren Services verarbeitet wird. Lambda zeichnet 2 Segmente pro Trace auf, wodurch zwei Knoten im Dienstgraphen entstehen. In der folgenden Abbildung werden diese beiden Knoten hervorgehoben:



Der erste Knoten auf der linken Seite stellt den Lambda-Service dar, der die Aufrufanforderung empfängt. Der zweite Knoten stellt Ihre spezifische Lambda-Funktion dar. Das folgende Beispiel zeigt eine Nachverfolgung mit diesen zwei Segmenten. Beide heißen my-function, aber einer hat einen Ursprung von `AWS::Lambda` und der andere hat einen Ursprung von `AWS::Lambda::Function`. Wenn das `AWS::Lambda`-Segment einen Fehler anzeigt, hatte der Lambda-Service ein Problem. Wenn das `AWS::Lambda::Function`-Segment einen Fehler anzeigt, ist bei Ihrer Funktion ein Problem aufgetreten.



Dieses Beispiel erweitert das `AWS::Lambda::Function`-Segment, um seine drei Untersegmente anzuzeigen.

> **Note**
>
> AWS implementiert derzeit Änderungen am Lambda-Service. Aufgrund dieser Änderungen können geringfügige Unterschiede in Struktur und Inhalt der Systemprotokollmeldungen und Trace-Segmente auftreten, die von verschiedenen Lambda-Funktionen in Ihrem AWS-Konto. Der hier gezeigte Beispiel-Trace veranschaulicht das Funktionssegment im alten Stil. Die Unterschiede zwischen den Segmenten im alten und im neuen Stil werden in den folgenden Abschnitten beschrieben.
> Diese Änderungen werden in den kommenden Wochen umgesetzt, und alle Funktionen AWS-Regionen außer China und den GovCloud Regionen werden auf die Verwendung der Protokollnachrichten und Trace-Segmente im neuen Format umgestellt.

Das Funktionssegment im alten Stil enthält die folgenden Untersegmente:

- Initialisierung – Stellt die Zeit dar, die für das Laden Ihrer Funktion und das Ausführen des Initialisierungscodes aufgewendet wurde. Dieses Untersegment erscheint nur für das erste Ereignis, das jede Instance Ihrer Funktion verarbeitet.
- Invocation (Aufruf) – Stellt die Zeit dar, die beim Ausführen Ihres Handler-Codes vergeht.
- Overhead (Aufwand) – Stellt die Zeit dar, die von der Lambda-Laufzeitumgebung bei der Verarbeitung des nächsten Ereignisses verbraucht wird.

Das Funktionssegment im neuen Stil enthält kein `Invocation`-Untersegment. Stattdessen werden Kundenuntersegmente direkt an das Funktionssegment angehängt. Weitere Informationen über die Struktur der alten und neuen Funktionssegmente finden Sie unter the section called “Grundlegendes zu X-Ray-Ablaufverfolgungen”.

> **Note**
>
> Zu den Lambda SnapStart-Funktionen gehört auch ein `Restore`-Teilsegment. Das Teilsegment `Restore` zeigt die Zeit an, die Lambda benötigt, um einen Snapshot wiederherzustellen, die Laufzeit zu laden und vorhandene Laufzeit-Hooks nach der Wiederherstellung auszuführen. Der Prozess der Wiederherstellung von Snapshots kann Zeit beinhalten, die für Aktivitäten außerhalb der MicroVM aufgewendet wird. Diese Zeit wird im `Restore`-Untersegment erfasst. Die Zeit, die Sie außerhalb der microVM für die Wiederherstellung eines Snapshots aufwenden, wird Ihnen nicht in Rechnung gestellt.

Sie können auch HTTP-Clients instrumentieren, SQL-Abfragen aufzeichnen und benutzerdefinierte Untersegmente mit Anmerkungen und Metadaten erstellen. Weitere Informationen finden Sie unter AWS X-Ray SDK für Java im AWS X-Ray -Entwicklerhandbuch.

> **Preisgestaltung**
>
> Im Rahmen des kostenlosen Kontingents können Sie X-Ray Tracing jeden Monat bis zu einem bestimmten Limit AWS kostenlos nutzen. Über den Schwellenwert hinaus berechnet X-Ray Gebühren für die Speicherung und den Abruf der Nachverfolgung. Weitere Informationen finden Sie unter AWS X-Ray Preise.

Laufzeitabhängigkeiten in einer Ebene speichern (X-Ray-SDK)

Wenn Sie das X-Ray-SDK verwenden, um AWS SDK-Clients Ihren Funktionscode zu instrumentieren, kann Ihr Bereitstellungspaket ziemlich umfangreich werden. Um Laufzeitabhängigkeiten bei jeder Aktualisierung des Funktionscodes zu vermeiden, verpacken Sie das X-Ray-SDK in einer Lambda-Ebene.

Das folgende Beispiel zeigt eine `AWS::Serverless::LayerVersion`-Ressource, die AWS SDK für Java und das X-Ray-SDK für Java speichert.

Example template.yml – Abhängigkeitenebene

```
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
```

Bei dieser Konfiguration aktualisieren Sie die Bibliotheksebene nur, wenn Sie Ihre Laufzeitabhängigkeiten ändern. Da das Funktionsbereitstellungspaket nur Ihren Code enthält, kann dies dazu beitragen, die Upload-Zeiten zu reduzieren.

Das Erstellen einer Ebene für Abhängigkeiten erfordert Build-Konfigurationsänderungen, um das Layer-Archiv vor der Bereitstellung zu generieren. Ein funktionierendes Beispiel finden Sie in der Java-Basic-Beispielanwendung unter. GitHub

X-Ray-Nachverfolgung in Beispielanwendungen (X-Ray-SDK)

Das GitHub Repository für dieses Handbuch enthält Beispielanwendungen, die die Verwendung von X-Ray Tracing demonstrieren. Jede Beispielanwendung enthält Skripts für die einfache Bereitstellung und Bereinigung, eine AWS SAM Vorlage und unterstützende Ressourcen.

Lambda-Beispielanwendungen in Java

- example-java – Eine Java-Funktion, die zeigt, wie Sie Lambda zur Bearbeitung von Aufträgen verwenden können. Diese Funktion veranschaulicht, wie Sie ein benutzerdefiniertes Eingabeereignisobjekt definieren und deserialisieren, das AWS SDK verwenden und die Ausgabeprotokollierung durchführen.
- Java-Basis – Eine Sammlung minimaler Java-Funktionen mit Einheitentests und variabler Protokollierungskonfiguration.
- Java-Ereignisse – Eine Sammlung von Java-Funktionen, die Grundcode für den Umgang mit Ereignissen aus verschiedenen Services wie Amazon API Gateway, Amazon SQS und Amazon Kinesis enthalten. Diese Funktionen verwenden die neueste Version der aws-lambda-java-eventsBibliothek (3.0.0 und neuer). Für diese Beispiele ist das AWS SDK nicht als Abhängigkeit erforderlich.
- s3-java – Eine Java-Funktion die Benachrichtigungsereignisse aus Amazon S3 verarbeitet und die Java Class Library (JCL) verwendet, um Miniaturansichten aus hochgeladenen Image-Dateien zu erstellen.
- layer-java – Eine Java-Funktion, die veranschaulicht, wie Sie mit einer Lambda-Ebene Abhängigkeiten getrennt von Ihrem Kernfunktionscode packen können.

Für alle Beispielanwendungen ist die aktive Ablaufverfolgung für Lambda-Funktionen aktiviert. Die `s3-java` Anwendung zeigt beispielsweise die automatische Instrumentierung von AWS SDK

for Java 2.x Clients, Segmentmanagement für Tests, benutzerdefinierte Untersegmente und die Verwendung von Lambda-Schichten zum Speichern von Laufzeitabhängigkeiten.

Java-Beispielanwendungen für AWS Lambda

Das GitHub Repository für dieses Handbuch enthält Beispielanwendungen, die die Verwendung von Java in demonstrieren AWS Lambda. Jede Beispielanwendung enthält Skripts für die einfache Bereitstellung und Bereinigung, eine CloudFormation Vorlage und unterstützende Ressourcen.

Lambda-Beispielanwendungen in Java

- example-java – Eine Java-Funktion, die zeigt, wie Sie Lambda zur Bearbeitung von Aufträgen verwenden können. Diese Funktion veranschaulicht, wie Sie ein benutzerdefiniertes Eingabeereignisobjekt definieren und deserialisieren, das AWS SDK verwenden und die Ausgabeprotokollierung durchführen.
- Java-Basis – Eine Sammlung minimaler Java-Funktionen mit Einheitentests und variabler Protokollierungskonfiguration.
- Java-Ereignisse – Eine Sammlung von Java-Funktionen, die Grundcode für den Umgang mit Ereignissen aus verschiedenen Services wie Amazon API Gateway, Amazon SQS und Amazon Kinesis enthalten. Diese Funktionen verwenden die neueste Version der aws-lambda-java-eventsBibliothek (3.0.0 und neuer). Für diese Beispiele ist das AWS SDK nicht als Abhängigkeit erforderlich.
- s3-java – Eine Java-Funktion die Benachrichtigungsereignisse aus Amazon S3 verarbeitet und die Java Class Library (JCL) verwendet, um Miniaturansichten aus hochgeladenen Image-Dateien zu erstellen.
- layer-java – Eine Java-Funktion, die veranschaulicht, wie Sie mit einer Lambda-Ebene Abhängigkeiten getrennt von Ihrem Kernfunktionscode packen können.

Ausführen beliebter Java-Frameworks auf Lambda

- spring-cloud-function-samples— Ein Beispiel aus Spring, das zeigt, wie das Spring Cloud Function Framework verwendet wird, um AWS Lambda-Funktionen zu erstellen.
- Serverlose Spring Boot-Anwendungsdemo — Ein Beispiel, das zeigt, wie eine typische Spring Boot-Anwendung in einer verwalteten Java-Laufzeit mit und ohne SnapStart oder als natives GraalVM-Image mit einer benutzerdefinierten Laufzeit eingerichtet wird.
- Serverlose Micronaut-Anwendungsdemo — Ein Beispiel, das zeigt, wie Micronaut in einer verwalteten Java-Laufzeit mit und ohne oder als natives SnapStart GraalVM-Image mit einer benutzerdefinierten Laufzeit verwendet wird. Erfahren Sie mehr in den Micronaut/Lambda-Leitfäden.

- Serverlose Quarkus-Anwendungsdemo — Ein Beispiel, das zeigt, wie Quarkus in einer verwalteten Java-Laufzeit mit und ohne oder als natives GraalVM-Image mit einer SnapStart benutzerdefinierten Laufzeit verwendet werden kann. Weitere Informationen finden Sie im Quarkus/Lambda-Leitfaden und im Quarkus/-Leitfaden. SnapStart

Wenn Lambda-Funktionen in Java für Sie neu sind, beginnen mit den `java-basic`-Beispielen. Für erste Schritte mit Lambda-Ereignisquellen schauen Sie sich die `java-events`-Beispiele an. Beide Beispielsätze zeigen die Verwendung der Java-Bibliotheken, Umgebungsvariablen, des SDK und des SDK von Lambda. AWS AWS X-Ray Diese Beispiele erfordern nur eine minimale Einrichtung. Sie können sie in weniger als einer Minute von der Befehlszeile aus bereitstellen.

Erstellen von Lambda-Funktionen mit Go

Go wird anders implementiert als andere verwaltete Laufzeiten. Da Go nativ zu einer ausführbaren Binärdatei kompiliert wird, ist keine spezielle Sprachlaufzeit erforderlich. Verwenden Sie eine reine Betriebssystemlaufzeit (die `provided`-Laufzeit-Familie), um Go-Funktionen für Lambda bereitzustellen.

Themen

- Unterstützte Go-Laufzeiten
- Tools und Bibliotheken
- Definieren des Lambda-Funktions-Handlers in Go
- Verwenden des Lambda-Kontextobjekts zum Abrufen von Go-Funktionsinformationen
- Bereitstellen von Lambda-Go-Funktionen mit ZIP-Dateiarchiven
- Bereitstellen von Go-Lambda-Funktionen mit Container-Images
- Arbeiten mit Ebenen für Go Lambda-Funktionen
- Go-Lambda-Funktionen protokollieren und überwachen
- Instrumentieren von Go-Code in AWS Lambda

Unterstützte Go-Laufzeiten

Die verwaltete Laufzeit Go 1.x für Lambda ist veraltet. Wenn Sie Funktionen haben, die die Go 1.x-Laufzeit verwenden, müssen Sie Ihre Funktionen nach `provided.al2023` oder `provided.al2` migrieren. Die `provided.al2` Laufzeiten `provided.al2023` und bieten mehrere Vorteile gegenüber`go1.x`, darunter Unterstützung für die Arm64-Architektur (AWS Graviton2-Prozessoren), kleinere Binärdateien und etwas schnellere Aufrufzeiten.

Für diese Migration sind keine Codeänderungen erforderlich. Die einzigen erforderlichen Änderungen betreffen die Erstellung Ihres Bereitstellungspakets sowie die Laufzeit, die Sie zur Erstellung Ihrer Funktion verwenden. Weitere Informationen finden Sie im Compute-Blog unter AWS Lambda Funktionen von der GO1.x-Laufzeit zur benutzerdefinierten Laufzeit auf Amazon Linux 2 migrieren.AWS

Name	ID	Betriebssystem	Datum der Veraltung	Blockfunktion erstellen	Blockfunktion aktualisieren
Reine OS-Laufzeit	`provided.al2023`	Amazon Linux 2023	30. Juni 2029	31. Juli 2029	31. August 2029
Reine OS-Laufzeit	`provided.al2`	Amazon Linux 2	31. Juli 2026	31. August 2026	30. September 2026

Tools und Bibliotheken

Lambda stellt die folgenden Tools und Bibliotheken für die Go-Laufzeit bereit:

- AWS SDK für Go v2: Das offizielle AWS SDK für die Programmiersprache Go.
- Github. com/aws/aws-lambda-go/lambda: Die Implementierung des Lambda-Programmiermodells für Go. Dieses Paket wird von verwendet AWS Lambda , um Ihren Handler aufzurufen.
- Github. com/aws/aws-lambda-go/lambdacontext: Helfer für den Zugriff auf Kontextinformationen aus dem Kontextobjekt.
- Github. com/aws/aws-lambda-go/events: Diese Bibliothek bietet Typdefinitionen für gängige Integrationen mit Ereignisquellen.
- Github. com/aws/aws-lambda-go/cmd/build-lambda-zip: Dieses Tool kann verwendet werden, um ein ZIP-Dateiarchiv unter Windows zu erstellen.

Weitere Informationen finden Sie unter. aws-lambda-go GitHub

Lambda stellt die folgenden Beispielanwendungen für die Go-Laufzeit bereit:

Lambda-Beispielanwendungen in Go

- go-al2: Eine Hello World-Funktion, die die öffentliche IP-Adresse zurückgibt. Diese App verwendet die benutzerdefinierte Laufzeit `provided.al2`.
- blank-go — Eine Go-Funktion, die die Verwendung der Go-Bibliotheken, der Protokollierung, der Umgebungsvariablen und des SDK von Lambda zeigt. AWS Diese App verwendet die Laufzeit `go1.x`.

Definieren des Lambda-Funktions-Handlers in Go

Der Lambda-Funktionshandler ist die Methode in Ihrem Funktionscode, die Ereignisse verarbeitet. Wenn Ihre Funktion aufgerufen wird, führt Lambda die Handler-Methode aus. Ihre Funktion wird so lange ausgeführt, bis der Handler eine Antwort zurückgibt, beendet wird oder ein Timeout auftritt.

Auf dieser Seite wird beschrieben, wie Sie mit Lambda-Funktionshandlern in Go arbeiten, einschließlich der Projekteinrichtung, Benennungskonventionen und Best Practices. Diese Seite enthält auch ein Beispiel für eine Go-Lambda-Funktion, die Informationen über einen Auftrag aufnimmt, eine Textdatei als Quittung erstellt und diese Datei in einem Bucket von Amazon Simple Storage Service (Amazon S3) platziert. Informationen darüber, wie Sie Ihre Funktion nach dem Schreiben einsetzen können, finden Sie unter the section called “Bereitstellen von ZIP-Dateiarchiven” oder the section called “Bereitstellen von Container-Images”.

Themen

- Einrichten Ihres Go-Handler-Projekts
- Beispiel für Go Lambda-Funktionscode
- Namenskonventionen für Handler
- Definieren Sie das Eingabeereignisobjekt und greifen Sie darauf zu
- Zugreifen auf und Verwenden des Lambda-Kontextobjekts
- Gültige Handler-Signaturen für Go-Handler
- Verwenden Sie die AWS SDK für Go v2 in Ihrem Handler
- Zugriff auf Umgebungsvariablen
- Verwenden des globalen Zustands
- Bewährte Codemethoden für Go-Lambda-Funktionen

Einrichten Ihres Go-Handler-Projekts

Eine Lambda-Funktion in Go wird als ausführbare Go-Datei erstellt. Sie können ein Go-Lambda-Funktionsprojekt auf die gleiche Weise initialisieren wie jedes andere Go-Projekt, indem Sie den folgenden `go mod init`-Befehl verwenden:

```
go mod init example-go
```

Hier ist `example-go` der Name des Moduls. Sie können dies durch alles ersetzen. Dieser Befehl initialisiert Ihr Projekt und erzeugt die `go.mod`-Datei, in der die Abhängigkeiten Ihres Projekts aufgelistet sind.

Verwenden Sie den Befehl `go get`, um externe Abhängigkeiten zu Ihrem Projekt hinzuzufügen. Für alle Lambda-Funktionen in Go müssen Sie zum Beispiel das Paket github.com/aws/aws-lambda-go/lambda einbinden, das das Lambda-Programmiermodell für Go implementiert. Binden Sie dieses Paket mit dem folgenden `go get`-Befehl ein:

```
go get github.com/aws/aws-lambda-go
```

Ihr Funktionscode sollte sich in einer Go-Datei befinden. Im folgenden Beispiel benennen wir diese Datei `main.go`. In dieser Datei implementieren Sie Ihre Kernfunktionslogik in einer Handler-Methode sowie in einer `main()`-Funktion, die diesen Handler aufruft.

Beispiel für Go Lambda-Funktionscode

Das folgende Beispiel für einen Go-Lambda-Funktionscode nimmt Informationen über eine Bestellung auf, erstellt eine Textdateiquittung und platziert diese Datei in einem Amazon-S3-Bucket.

Example **`main.go`**-Lambda-Funktion

```
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)
}
```

Diese `main.go`-Datei enthält die folgenden Abschnitte des Codes:

- `package main`: In Go muss das Paket, das Ihre `func main()`-Funktion enthält, immer `main` heißen.
- `import`-Block: Verwenden Sie diesen Block, um Bibliotheken einzubinden, die Ihre Lambda-Funktion benötigt.
- `type Order struct {}`-Block: Definieren Sie die Form des erwarteten Eingabeereignisses in dieser Go-Struktur.
- `var ()`-Block: Verwenden Sie diesen Block, um alle globalen Variablen zu definieren, die Sie in Ihrer Lambda-Funktion verwenden werden.
- `func init() {}`: Fügen Sie in diese `init()`-Methode beliebigen Code ein, den Lambda während der Initialisierungsphase ausführen soll.
- `func uploadReceiptToS3(...) {}`: Dies ist eine Hilfsmethode, auf die von der `handleRequest`-Haupthandler-Methode verwiesen wird.
- `func handleRequest(ctx context.Context, event json.RawMessage) error {}`: Dies ist die Haupthandler-Methode, die Ihre Hauptanwendungslogik enthält.
- `func main() {}`: Dies ist ein erforderlicher Einstiegspunkt für Ihren Lambda-Handler. Das Argument für die `lambda.Start()`-Methode ist Ihre Haupthandler-Methode.

Damit diese Funktion ordnungsgemäß funktioniert, muss ihre Ausführungsrolle die `s3:PutObject`-Aktion zulassen. Stellen Sie außerdem sicher, dass Sie die `RECEIPT_BUCKET`-Umgebungsvariable definieren. Nach einem erfolgreichen Aufruf sollte der Amazon-S3-Bucket eine Empfangsdatei enthalten.

Namenskonventionen für Handler

Für Lambda-Funktionen in Go können Sie einen beliebigen Namen für den Handler verwenden. In diesem Beispiel lautet der Name der Handler-Methode `handleRequest`. Sie können die Umgebungsvariable `_HANDLER` verwenden, um in Ihrem Code auf den Handler-Wert zu verweisen.

Bei Go-Funktionen, die mit einem .zip-Bereitstellungspaket bereitgestellt werden, muss die ausführbare Datei, die Ihren Funktionscode enthält, den Namen `bootstrap` tragen. Darüber hinaus muss sich die `bootstrap`-Datei im Stammverzeichnis der .zip-Datei befinden. Für Go-Funktionen, die über ein Container-Image bereitgestellt werden, können Sie einen beliebigen Namen für die ausführbare Datei verwenden.

Definieren Sie das Eingabeereignisobjekt und greifen Sie darauf zu

JSON ist das gebräuchlichste und standardmäßigste Eingabeformat für Lambda-Funktionen. In diesem Beispiel erwartet die Funktion eine Eingabe ähnlich der folgenden:

```
{
    "order_id": "12345",
    "amount": 199.99,
    "item": "Wireless Headphones"
}
```

Wenn Sie mit Lambda-Funktionen in Go arbeiten, können Sie die Form des erwarteten Eingabeereignisses als Go-Struktur definieren. In diesem Beispiel definieren wir eine Struktur, die einen `Order` darstellt:

```
type Order struct {
    OrderID string  `json:"order_id"`
    Amount  float64 `json:"amount"`
    Item    string  `json:"item"`
}
```

Diese Struktur entspricht der erwarteten Eingabeform. Nachdem Sie Ihre Struktur definiert haben, können Sie eine Handler-Signatur schreiben, die einen generischen JSON-Typ verwendet, der

mit der encoding/json-Standardbibliothek kompatibel ist. Sie können es dann mit der Funktion func Unmarshal in Ihre Struktur deserialisieren. Dies wird in den ersten Zeilen des Handlers veranschaulicht:

```
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
    ...
}
```

Nach dieser Deserialisierung können Sie auf die Felder der `order`-Variablen zugreifen. Zum Beispiel holt `order.OrderID` den Wert von `"order_id"` aus der ursprünglichen Eingabe.

> **Note**
>
> Das `encoding/json`-Paket kann nur auf exportierte Felder zugreifen. Um exportiert werden, müssen Feldnamen in der Ereignisstruktur großgeschrieben werden.

Zugreifen auf und Verwenden des Lambda-Kontextobjekts

Das Lambda-Kontextobjekt enthält Informationen über Aufruf, Funktion und Ausführungsumgebung. In diesem Beispiel haben wir diese Variable wie `ctx` in der Handler-Signatur deklariert:

```
func handleRequest(ctx context.Context, event json.RawMessage) error {
    ...
}
```

Die `ctx context.Context`-Eingabe ist ein optionales Argument in Ihrem Funktionshandler. Weitere Informationen zu akzeptierten Handler-Signaturen finden Sie unter the section called “Gültige Handler-Signaturen für Go-Handler”.

Wenn Sie mit dem AWS-SDK andere Dienste aufrufen, wird das Kontextobjekt in einigen wichtigen Bereichen benötigt. Um beispielsweise Ihre SDK-Clients ordnungsgemäß zu initialisieren, können Sie die richtige AWS-SDK-Konfiguration mithilfe des Kontextobjekts wie folgt laden:

```
// Load AWS SDK configuration using the default credential provider chain
```

```
    cfg, err := config.LoadDefaultConfig(ctx)
```

SDK-Aufrufe selbst benötigen möglicherweise das Kontextobjekt als Eingabe. Der Aufruf von `s3Client.PutObject` akzeptiert beispielsweise das Kontextobjekt als erstes Argument:

```
// Upload the receipt to S3
    _, err = s3Client.PutObject(ctx, &s3.PutObjectInput{
        ...
    })
```

Außerhalb von AWS-SDK-Anfragen können Sie das Kontextobjekt auch für die Funktionsüberwachung verwenden. Weitere Informationen über das Kontextobjekt finden Sie unter the section called “Kontext”.

Gültige Handler-Signaturen für Go-Handler

Sie haben bei der Erstellung eines Lambda-Funktions-Handlers in Go mehrere Möglichkeiten. Allerdings müssen die folgenden Regeln beachten:

- Der Handler muss eine Funktion sein.
- Der Handler kann zwischen 0 und 2 Argumente aufnehmen. Bei zwei Argumenten muss das erste Argument implementiere `context.Context`.
- Der Handler kann zwischen 0 und 2 Argumente zurückgeben. Bei einem einzigen Rückgabewert muss er implementiere `error`. Bei zwei Rückgabewerten muss der zweite Wert implementiere `error`.

Im Folgenden werden gültige Handler-Signaturen aufgeführt. `TIn` und `TOut` stellen Typen dar, die mit der Standardbibliothek encoding/json kompatibel sind. Weitere Informationen zur Deserialisierung dieser Typen finden Sie unter func Unmarshal.

- `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)`

Verwenden Sie die AWS SDK für Go v2 in Ihrem Handler

Oft verwenden Sie Lambda-Funktionen, um mit anderen AWS-Ressourcen zu interagieren oder diese zu aktualisieren. Die einfachste Art, eine Schnittstelle zu diesen Ressourcen herzustellen, ist die Verwendung von AWS SDK für Go v2.

> **Note**
>
> AWS SDK für Go (v1) befindet sich im Wartungsmodus und der Support wird am 31. Juli 2025 eingestellt. Wir empfehlen, dass Sie in Zukunft nur noch AWS SDK für Go v2 verwenden.

Um Ihrer Funktion SDK-Abhängigkeiten hinzuzufügen, verwenden Sie den `go get`-Befehl für die spezifischen SDK-Clients, die Sie benötigen. Im Beispielcode zuvor haben wir die `config`-Bibliothek und die `s3`-Bibliothek verwendet. Fügen Sie diese Abhängigkeiten hinzu, indem Sie die folgenden Befehle in dem Verzeichnis ausführen, das Ihre `go.mod`- und `main.go` -Dateien enthält:

```
go get github.com/aws/aws-sdk-go-v2/config
go get github.com/aws/aws-sdk-go-v2/service/s3
```

Importieren Sie dann die Abhängigkeiten entsprechend in den Importblock Ihrer Funktion:

```
import (
    ...
    "github.com/aws/aws-sdk-go-v2/config"
    "github.com/aws/aws-sdk-go-v2/service/s3"
```

```
)
```

Wenn Sie das SDK in Ihrem Handler verwenden, konfigurieren Sie Ihre Clients mit den richtigen Einstellungen. Der einfachste Weg, dies zu tun, besteht darin, die standardmäßige Anbieterkette für Anmeldeinformationen zu verwenden. Dieses Beispiel zeigt eine Möglichkeit, diese Konfiguration zu laden:

```
// 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
    }
```

Nachdem Sie diese Konfiguration in die `cfg`-Variable geladen haben, können Sie diese Variable an Client-Instanziierungen übergeben. Der Beispielcode instanziiert einen Amazon-S3-Client wie folgt:

```
// Create an S3 client
    s3Client := s3.NewFromConfig(cfg)
```

In diesem Beispiel haben wir unseren Amazon-S3-Client in der `init()`-Funktion initialisiert, um zu vermeiden, dass wir ihn bei jedem Aufruf unserer Funktion initialisieren müssen. Das Problem ist, dass Lambda in der `init()`-Funktion keinen Zugriff auf das Kontextobjekt hat. Als Abhilfe können Sie in der Initialisierungsphase einen Platzhalter wie `context.TODO()` übergeben. Wenn Sie später über den Client einen Anruf tätigen, übergeben Sie das vollständige Kontextobjekt. Diese Abhilfemaßnahme wird auch in the section called “Verwendung des Kontexts bei Initialisierungen und Aufrufen von AWS-SDK-Clients” beschrieben.

Nachdem Sie Ihren SDK-Client konfiguriert und initialisiert haben, können Sie ihn für die Interaktion mit anderen AWS-Diensten verwenden. Der Beispielcode ruft die Amazon S3 `PutObject`-API wie folgt auf:

```
_, err = s3Client.PutObject(ctx, &s3.PutObjectInput{
    Bucket: &bucketName,
    Key:    &key,
    Body:   strings.NewReader(receiptContent),
})
```

Zugriff auf Umgebungsvariablen

In Ihrem Handler-Code können Sie mithilfe der `os.Getenv()`-Methode auf beliebige Umgebungsvariablen verweisen. In diesem Beispiel verweisen wir mit der folgenden Codezeile auf die definierte `RECEIPT_BUCKET`-Umgebungsvariable:

```
// 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")
    }
```

Verwenden des globalen Zustands

Um zu vermeiden, dass bei jedem Funktionsaufruf neue Ressourcen erstellt werden, können Sie globale Variablen außerhalb des Handler-Codes Ihrer Lambda-Funktion deklarieren und ändern. Sie definieren diese globalen Variablen in einem `var`-Block oder einer Anweisung. Darüber hinaus deklariert Ihr Handler möglicherweise eine `init()`-Funktion, die während der Initialisierungsphase ausgeführt wird. Die `init`-Methode verhält sich in AWS Lambda genauso wie in normalen Go-Programmen.

Bewährte Codemethoden für Go-Lambda-Funktionen

Halten Sie sich an die Richtlinien in der folgenden Liste, um beim Erstellen Ihrer Lambda-Funktionen die besten Codierungspraktiken anzuwenden:

- Trennen Sie den Lambda-Handler von Ihrer Core-Logik. Auf diese Weise können Sie eine Funktion zur besseren Prüfbarkeit von Einheiten schaffen.
- Minimieren Sie die Komplexität Ihrer Abhängigkeiten. Ziehen Sie einfachere Frameworks vor, die sich schnell beim Start der Ausführungsumgebung laden lassen.
- Minimieren Sie die Größe Ihres Bereitstellungspakets auf die für die Laufzeit erforderliche Größe. Dadurch verkürzt sich die Zeit, die für das Herunterladen und Entpacken Ihres Bereitstellungspakets vor dem Aufruf benötigt wird.

Nutzen Sie die Wiederverwendung der Ausführungsumgebung zur Verbesserung Ihrer Funktion. Initialisieren Sie SDK-Clients und Datenbankverbindungen außerhalb des Funktions-Handlers und speichern Sie statische Komponenten lokal im `/tmp`-Verzeichnis. Nachfolgende Aufrufe, die von

derselben Instance Ihrer Funktion verarbeitet werden, können diese Ressourcen wiederverwenden. Dies spart Kosten durch Reduzierung der Funktionslaufzeit.

Um potenzielle Datenlecks über Aufrufe hinweg zu vermeiden, verwenden Sie die Ausführungsumgebung nicht, um Benutzerdaten, Ereignisse oder andere Informationen mit Sicherheitsauswirkungen zu speichern. Wenn Ihre Funktion auf einem veränderbaren Zustand beruht, der nicht im Speicher innerhalb des Handlers gespeichert werden kann, sollten Sie für jeden Benutzer eine separate Funktion oder separate Versionen einer Funktion erstellen.

Verwenden Sie eine Keep-Alive-Direktive, um dauerhafte Verbindungen zu pflegen. Lambda bereinigt Leerlaufverbindungen im Laufe der Zeit. Der Versuch, eine Leerlaufverbindung beim Aufruf einer Funktion wiederzuverwenden, führt zu einem Verbindungsfehler. Um Ihre persistente Verbindung aufrechtzuerhalten, verwenden Sie die Keep-Alive-Direktive, die Ihrer Laufzeit zugeordnet ist. Ein Beispiel finden Sie unter Wiederverwenden von Verbindungen mit Keep-Alive in Node.js.

Verwenden Sie Umgebungsvariablen um Betriebsparameter an Ihre Funktion zu übergeben. Wenn Sie z. B. Daten in einen Amazon-S3-Bucket schreiben, anstatt den Bucket-Namen, in den Sie schreiben, hartzucodieren, konfigurieren Sie den Bucket-Namen als Umgebungsvariable.

Vermeiden Sie rekursive Aufrufe in Ihrer Lambda-Funktion, bei denen die Funktion sich selbst aufruft oder einen Prozess initiiert, der die Funktion erneut aufrufen kann. Dies kann zu unvorhergesehenen Mengen an Funktionsaufrufen führen und höhere Kosten zur Folge haben. Wenn Sie eine unbeabsichtigte Menge von Aufrufen feststellen, legen Sie die reservierte gleichzeitige Ausführung der Funktion auf `0` fest, um sofort alle Aufrufe der Funktion zu drosseln, während Sie den Code aktualisieren.

Verwenden Sie keine nicht dokumentierten, nicht öffentlichen APIs in Ihrem Lambda-Funktionscode. Für AWS Lambda-verwaltete Laufzeiten wendet Lambda regelmäßig Sicherheits- und Funktionsupdates auf Lambdas interne APIs an. Diese internen API-Updates können abwärtskompatibel sein, was zu unbeabsichtigten Konsequenzen wie Aufruffehlern führt, wenn Ihre Funktion von diesen nicht öffentlichen APIs abhängig ist. Eine Liste öffentlich zugänglicher APIs finden Sie in der API-Referenz.

Schreiben Sie idempotenten Code. Das Schreiben idempotenter Code für Ihre Funktionen stellt sicher, dass doppelte Ereignisse auf die gleiche Weise behandelt werden. Ihr Code sollte Ereignisse ordnungsgemäß validieren und doppelte Ereignisse ordnungsgemäß behandeln. Weitere Informationen finden Sie unter Wie mache ich meine Lambda-Funktion idempotent?.

Verwenden des Lambda-Kontextobjekts zum Abrufen von Go-Funktionsinformationen

In Lambda bietet das Kontextobjekt Methoden und Eigenschaften mit Informationen über den Aufruf, die Funktion und die Ausführungsumgebung. Wenn Lambda Ihre Funktion ausführt, wird ein Context-Objekt an den Handler übergeben. Um das Kontextobjekt in Ihrem Handler zu verwenden, können Sie es optional als Eingabeparameter für Ihren Handler deklarieren. Das Kontextmenüobjekt ist erforderlich, wenn Sie in Ihrem Handler Folgendes tun möchten:

- Sie benötigen Zugriff auf eine der globalen Variablen, Methoden oder Eigenschaften, die vom Kontextobjekt angeboten werden. Diese Methoden und Eigenschaften sind nützlich für Aufgaben wie die Bestimmung der Entität, die Ihre Funktion aufgerufen hat, oder die Messung der Aufrufzeit Ihrer Funktion, wie in the section called “Zugreifen auf Aufrufkontextinformationen” dargestellt.
- Sie müssen das AWS SDK für Go verwenden, um Anrufe an andere Dienste zu tätigen. Das Kontextobjekt ist ein wichtiger Eingabeparameter für die meisten dieser Aufrufe. Weitere Informationen finden Sie unter the section called “Verwendung des Kontexts bei Initialisierungen und Aufrufen von AWS-SDK-Clients”.

Themen

- Unterstützte Variablen, Methoden und Eigenschaften im Kontextmenüobjekt
- Zugreifen auf Aufrufkontextinformationen
- Verwendung des Kontexts bei Initialisierungen und Aufrufen von AWS-SDK-Clients

Unterstützte Variablen, Methoden und Eigenschaften im Kontextmenüobjekt

Die Lambda-Kontextbibliothek bietet die folgenden globalen Variablen, Methoden und Eigenschaften.

Globale Variablen

- `FunctionName` – Der Name der Lambda-Funktion.
- `FunctionVersion` – Die Version der Funktion.
- `MemoryLimitInMB` – Die Menge an Arbeitsspeicher, die der Funktion zugewiesen ist.
- `LogGroupName` – Protokollgruppe für die Funktion.
- `LogStreamName` – Der Protokollstrom für die Funktionsinstance.

Context-Methoden

- `Deadline` – Gibt das Datum zurück, an dem eine Zeitüberschreitung für die Ausführung eintritt (in Unix-Millisekunden).

Context-Eigenschaften

- `InvokedFunctionArn` – Der Amazon-Ressourcenname (ARN), der zum Aufrufen der Funktion verwendet wird. Gibt an, ob der Aufrufer eine Versionsnummer oder einen Alias angegeben hat.
- `AwsRequestID` – Der Bezeichner der Aufrufanforderung.
- `Identity` – Informationen zur Amazon-Cognito-Identität, die die Anforderung autorisiert hat.
- `ClientContext` – (mobile Apps) Clientkontext, der Lambda von der Clientanwendung bereitgestellt wird.

Zugreifen auf Aufrufkontextinformationen

Lambda-Funktionen haben Zugriff auf Metadaten über ihre Umgebung und die Aufrufanforderung. Darauf kann unter Paketkontext zugegriffen werden. Falls Ihr Handler `context.Context` als Parameter umfasst, fügt Lambda Informationen über Ihre Funktion in der `Value`-Eigenschaft des Kontexts ein. Beachten Sie, dass Sie die `lambdacontext`-Bibliothek importieren müssen, um auf die Inhalte des `context.Context`-Objekts zuzugreifen.

```
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)
```

```
}
```

Im Beispiel oben ist `lc` die Variable, mit der die vom Kontextobjekt erfassten Informationen verarbeitet werden, und `log.Print(lc.Identity.CognitoIdentityPoolID)` druckt diese Informationen. In diesem Fall ist dies CognitoIdentityPoolID.

Das folgende Beispiel bietet eine Einführung in die Verwendung der Kontextobjekte zur Überwachung der Abschlussdauer Ihrer Lambda-Funktion. Damit können Sie die Leistungserwartungen analysieren und Ihren Funktionscode bei Bedarf entsprechend anpassen.

```
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)
}
```

Verwendung des Kontexts bei Initialisierungen und Aufrufen von AWS-SDK-Clients

Wenn Ihr Handler das AWS SDK für Go verwenden muss, um andere Dienste aufzurufen, schließen Sie das Kontextobjekt als Eingabe in Ihren Handler ein. In AWS ist es eine bewährte Praxis, das Kontextobjekt in den meisten AWS-SDK-Aufrufen zu übergeben. Der Aufruf von Amazon S3 `PutObject` akzeptiert beispielsweise das Kontextobjekt (`ctx`) als erstes Argument:

```
// Upload an object to S3
    _, err = s3Client.PutObject(ctx, &s3.PutObjectInput{
        ...
    })
```

Um Ihre SDK-Clients richtig zu initialisieren, können Sie auch das Kontext-Objekt verwenden, um die richtige Konfiguration zu laden, bevor Sie dieses Konfigurationsobjekt an den Client übergeben:

```
// Load AWS SDK configuration using the default credential provider chain
    cfg, err := config.LoadDefaultConfig(ctx)
    ...
    s3Client = s3.NewFromConfig(cfg)
```

Wenn Sie Ihre SDK-Clients außerhalb Ihres Haupthandlers initialisieren möchten (d. h. während der Initialisierungsphase), können Sie ein Platzhalter-Kontextobjekt übergeben:

```
func init() {
 // Initialize the S3 client outside of the handler, during the init phase
 cfg, err := config.LoadDefaultConfig(context.TODO())
 ...
 s3Client = s3.NewFromConfig(cfg)
}
```

Wenn Sie Ihre Clients auf diese Weise initialisieren, stellen Sie sicher, dass Sie bei SDK-Aufrufen von Ihrem Haupthandler das richtige Kontextobjekt übergeben.

Bereitstellen von Lambda-Go-Funktionen mit ZIP-Dateiarchiven

Der Code Ihrer AWS Lambda-Funktion besteht aus Skripts oder kompilierten Programmen und deren Abhängigkeiten. Sie verwenden ein Bereitstellungspaket, um Ihren Funktionscode in Lambda bereitzustellen. Lambda unterstützt zwei Arten von Bereitstellungspaketen: Container-Images und ZIP-Dateiarchiven.

Auf dieser Seite wird beschrieben, wie Sie eine ZIP-Datei als Bereitstellungspaket für die Go-Laufzeitumgebung erstellen und die ZIP-Datei dann verwenden, um Ihren Funktionscode auf AWS Lambda bereitzustellen, indem Sie AWS-Managementkonsole, AWS Command Line Interface (AWS CLI) und AWS Serverless Application Model (AWS SAM) verwenden.

Da Lambda POSIX-Dateiberechtigungen verwendet, müssen Sie möglicherweise Berechtigungen für den Bereitstellungspaketordner festlegen, bevor Sie das ZIP-Dateiarchiv erstellen.

Abschnitte

- Erstellen einer ZIP-Datei unter macOS und Linux
- Erstellen einer ZIP-Datei unter Windows
- Go Lambda-Funktionen mithilfe von ZIP-Dateien erstellen und aktualisieren

Erstellen einer ZIP-Datei unter macOS und Linux

Im Folgenden wird beschrieben, wie Sie Ihre ausführbare Datei mithilfe des Befehls `go build` kompilieren und ein Bereitstellungspaket in Form einer ZIP-Datei für Lambda erstellen. Vergewissern Sie sich vor dem Kompilieren des Codes, dass Sie das Paket lambda von GitHub installiert haben. Dieses Modul stellt eine Implementierung der Laufzeitschnittstelle bereit, die die Interaktion zwischen Lambda und Ihrem Funktionscode verwaltet. Führen Sie den folgenden Befehl aus, um diese Bibliothek herunterzuladen.

```
go get github.com/aws/aws-lambda-go/lambda
```

Wenn Ihre Funktion das AWS SDK für Go verwendet, laden Sie den Standardsatz von SDK-Modulen zusammen mit allen für Ihre Anwendung erforderlichen AWS-Service-API-Clients herunter. Informationen zur Installation des SDK für Go finden Sie unter Erste Schritte mit AWS SDK für Go V2.

Verwenden der bereitgestellten Laufzeitfamilie

Go wird anders implementiert als andere verwaltete Laufzeiten. Da Go nativ zu einer ausführbaren Binärdatei kompiliert wird, ist keine spezielle Sprachlaufzeit erforderlich. Verwenden Sie eine reine Betriebssystemlaufzeit (die `provided`-Laufzeit-Familie), um Go-Funktionen für Lambda bereitzustellen.

So erstellen Sie ein ZIP-Bereitstellungspaket (macOS/Linux)

1. Kompilieren Sie Ihre ausführbare Datei in dem Projektverzeichnis, das die Datei `main.go` Ihrer Anwendung enthält. Beachten Sie Folgendes:

   - Die ausführbare Datei muss `bootstrap` heißen. Weitere Informationen finden Sie unter Namenskonventionen für Handler.
   - Legen Sie die Befehlssatzarchitektur des Ziels fest. Reine Betriebssystem-Laufzeiten unterstützen sowohl arm64 als auch x86_64.
   - Sie können das optionale Tag `lambda.norpc` verwenden, um die RPC-Komponente (Remote Procedure Call, Remoteprozeduraufruf) der Bibliothek lambda auszuschließen. Die RPC-Komponente ist nur erforderlich, wenn Sie die veraltete Go 1.x-Laufzeit verwenden. Durch Ausschließen der RPC-Komponente verringert sich die Größe des Bereitstellungspakets.

   arm64-Architektur:

   ```
   GOOS=linux GOARCH=arm64 go build -tags lambda.norpc -o bootstrap main.go
   ```

   x86_64-Architektur:

   ```
   GOOS=linux GOARCH=amd64 go build -tags lambda.norpc -o bootstrap main.go
   ```

2. (Optional) Möglicherweise müssen Sie Pakete mit Einstellung von `CGO_ENABLED=0` in Linux kompilieren:

   ```
   GOOS=linux GOARCH=arm64 CGO_ENABLED=0 go build -o bootstrap -tags lambda.norpc
    main.go
   ```

   Dieser Befehl erstellt ein stabiles Binärpaket für Standard-C-Bibliotheksversionen (`libc`), was auf Lambda und anderen Geräten unterschiedlich sein kann.
3. Erstellen Sie ein Bereitstellungspaket, indem Sie die ausführbare Datei in eine ZIP-Datei packen.

```
zip myFunction.zip bootstrap
```

> **Note**
>
> Fügen Sie die `bootstrap`-Datei dem Stamm der ZIP-Datei hinzu.

4. Erstellen der -Funktion Beachten Sie Folgendes:

   - Die Binärdatei muss `bootstrap` benannt werden, aber der Handlername kann beliebig sein. Weitere Informationen finden Sie unter Namenskonventionen für Handler.
   - Die `--architectures`-Option ist nur bei Verwendung von „arm64“ erforderlich. Der Standardwert ist „x86_64“.
   - Geben Sie für `--role` den Amazon-Ressourcennamen (ARN) der Ausführungsrolle an.

```
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
```

Erstellen einer ZIP-Datei unter Windows

Im Folgenden wird gezeigt, wie Sie das Tool build-lambda-zip für Windows von GitHub herunterladen, Ihre ausführbare Datei kompilieren und ein ZIP-Bereitstellungspaket erstellen.

> **Note**
>
> Falls noch nicht geschehen, müssen Sie git installieren und die ausführbare `git`-Datei Ihrer Windows- %PATH%-Umgebungsvariable hinzufügen.

Stellen Sie vor dem Kompilieren des Codes sicher, dass Sie die Lambda-Bibliothek von GitHub installiert haben. Führen Sie den folgenden Befehl aus, um diese Bibliothek herunterzuladen.

```
go get github.com/aws/aws-lambda-go/lambda
```

Wenn Ihre Funktion das AWS SDK für Go verwendet, laden Sie den Standardsatz von SDK-Modulen zusammen mit allen für Ihre Anwendung erforderlichen AWS-Service-API-Clients herunter. Informationen zur Installation des SDK für Go finden Sie unter Erste Schritte mit AWS SDK für Go V2.

Verwenden der bereitgestellten Laufzeitfamilie

Go wird anders implementiert als andere verwaltete Laufzeiten. Da Go nativ zu einer ausführbaren Binärdatei kompiliert wird, ist keine spezielle Sprachlaufzeit erforderlich. Verwenden Sie eine reine Betriebssystemlaufzeit (die `provided`-Laufzeit-Familie), um Go-Funktionen für Lambda bereitzustellen.

So erstellen Sie ein ZIP-Bereitstellungspaket (Windows)

1. Laden Sie das build-lambda-zip-Tool von GitHub herunter.

   ```
   go install github.com/aws/aws-lambda-go/cmd/build-lambda-zip@latest
   ```

2. Verwenden Sie das Tool aus Ihrer `GOPATH`, um eine ZIP-Datei zu erstellen. Bei einer Go-Standardinstallation befindet sich das Tool in der Regel unter `%USERPROFILE%\Go\bin`. Navigieren Sie andernfalls an den Installationsort der Go-Laufzeit und führen Sie einen der folgenden Schritte aus:

   cmd.exe

   Verwenden Sie in „cmd.exe“ je nach Befehlssatzarchitektur des Ziels eine der folgenden Optionen. Reine Betriebssystem-Laufzeiten unterstützen sowohl arm64 als auch x86_64.

   Sie können das optionale Tag `lambda.norpc` verwenden, um die RPC-Komponente (Remote Procedure Call, Remoteprozeduraufruf) der Bibliothek lambda auszuschließen. Die RPC-Komponente ist nur erforderlich, wenn Sie die veraltete Go 1.x-Laufzeit verwenden. Durch Ausschließen der RPC-Komponente verringert sich die Größe des Bereitstellungspakets.

   Example für die x86_64-Architektur

   ```
   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 für die arm64-Architektur

```
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

Verwenden Sie in PowerShell je nach Befehlssatzarchitektur des Ziels eine der folgenden Optionen. Reine Betriebssystem-Laufzeiten unterstützen sowohl arm64 als auch x86_64.

Sie können das optionale Tag `lambda.norpc` verwenden, um die RPC-Komponente (Remote Procedure Call, Remoteprozeduraufruf) der Bibliothek lambda auszuschließen. Die RPC-Komponente ist nur erforderlich, wenn Sie die veraltete Go 1.x-Laufzeit verwenden. Durch Ausschließen der RPC-Komponente verringert sich die Größe des Bereitstellungspakets.

x86_64-Architektur:

```
$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
```

arm64-Architektur:

```
$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. Erstellen der -Funktion Beachten Sie Folgendes:

- Die Binärdatei muss `bootstrap` benannt werden, aber der Handlername kann beliebig sein. Weitere Informationen finden Sie unter Namenskonventionen für Handler.

- Die `--architectures`-Option ist nur bei Verwendung von „arm64“ erforderlich. Der Standardwert ist „x86_64“.
- Geben Sie für `--role` den Amazon-Ressourcennamen (ARN) der Ausführungsrolle an.

```
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
```

Go Lambda-Funktionen mithilfe von ZIP-Dateien erstellen und aktualisieren

Nach der Erstellung Ihres ZIP-Bereitstellungspakets können Sie es verwenden, um eine neue Lambda-Funktion zu erstellen oder eine vorhandene zu aktualisieren. Ihr ZIP-Bereitstellungspaket lässt sich über die Lambda-Konsole, das AWS Command Line Interface und die Lambda-API hochladen. Sie können Lambda-Funktionen auch mit AWS Serverless Application Model (AWS SAM) und CloudFormation erstellen und aktualisieren.

Die maximale Größe eines ZIP-Bereitstellungspakets für Lambda beträgt 250 MB (entpackt). Beachten Sie, dass dieser Grenzwert für die kombinierte Größe aller hochgeladenen Dateien gilt, einschließlich aller Lambda-Ebenen.

Die Lambda-Laufzeit benötigt die Berechtigung zum Lesen der Dateien in Ihrem Bereitstellungspaket. In der oktalen Linux-Notation der Berechtigungen benötigt Lambda 644 Berechtigungen für nicht ausführbare Dateien (rw-r--r--) und 755 Berechtigungen (rwxr-xr-x) für Verzeichnisse und ausführbare Dateien.

Verwenden Sie unter Linux und MacOS den `chmod`-Befehl, um Dateiberechtigungen für Dateien und Verzeichnisse in Ihrem Bereitstellungspaket zu ändern. Um beispielsweise einer nicht ausführbaren Datei die richtigen Berechtigungen zu geben, führen Sie den folgenden Befehl aus.

```
chmod 644 <filepath>
```

Informationen zum Ändern von Dateiberechtigungen in Windows finden Sie unter Festlegen, Anzeigen, Ändern oder Entfernen von Berechtigungen für ein Objekt in der Microsoft-Windows-Dokumentation.

> **Note**
>
> Wenn Sie Lambda nicht die Berechtigungen gewähren, die es für den Zugriff auf Verzeichnisse in Ihrem Bereitstellungspaket benötigt, setzt Lambda die Berechtigungen für diese Verzeichnisse auf 755 (rwxr-xr-x).

Erstellen und Aktualisieren von Funktionen mit ZIP-Dateien unter Verwendung der Konsole

Eine neue Funktion müssen Sie zuerst in der Konsole erstellen und dann Ihr ZIP-Archiv hochladen. Zum Aktualisieren einer bestehenden Funktion öffnen Sie die Seite für Ihre Funktion und gehen dann genauso vor, um Ihre aktualisierte ZIP-Datei hinzuzufügen.

Bei einer ZIP-Datei mit unter 50 MB können Sie eine Funktion erstellen oder aktualisieren, indem Sie die Datei direkt von Ihrem lokalen Computer hochladen. Bei ZIP-Dateien mit einer Größe von mehr als 50 MB müssen Sie Ihr Paket zuerst in einen Amazon-S3-Bucket hochladen. Anweisungen zum Hochladen einer Datei in einen Amazon-S3-Bucket unter Verwendung der AWS-Managementkonsole finden Sie unter Erste Schritte mit Amazon S3. Informationen zum Hochladen von Dateien mit der AWS CLI finden Sie im AWS CLI-Benutzerhandbuch unter Verschieben von Objekten.

> **Note**
>
> Sie können eine vorhandene Container-Image-Funktion nicht konvertieren, um ein ZIP-Archiv zu verwenden. Sie müssen eine neue Funktion erstellen.

So erstellen Sie eine neue Funktion (Konsole)

1. Öffnen Sie die Funktionsseite der Lambda-Konsole und wählen Sie Funktion erstellen aus.
2. Wählen Sie Author from scratch aus.
3. Führen Sie unter Basic information (Grundlegende Informationen) die folgenden Schritte aus:
   a. Geben Sie als Funktionsname den Namen Ihrer Funktion ein.
   b. Wählen Sie unter Runtime (Laufzeit) `provided.al2023` aus.
4. (Optional) Erweitern Sie unter Berechtigungen die Option Standardausführungsrolle ändern. Sie können eine neue Ausführungsrolle erstellen oder eine vorhandene Rolle verwenden.

5. Wählen Sie Funktion erstellen. Lambda erstellt eine grundlegende „Hello World“-Funktion mit der von Ihnen gewählten Laufzeit.

So laden Sie ein ZIP-Archiv von Ihrem lokalen Computer hoch (Konsole)

1. Wählen Sie auf der Funktionsseite der Lambda-Konsole die Funktion aus, für die Sie die ZIP-Datei hochladen möchten.
2. Wählen Sie die Registerkarte Code aus.
3. Wählen Sie im Bereich Codequelle die Option Hochladen von aus.
4. Wählen Sie die ZIP-Datei aus.
5. Laden Sie die ZIP-Datei wie folgt hoch:
   a. Wählen Sie Hochladen und dann Ihre ZIP-Datei in der Dateiauswahl aus.
   b. Klicken Sie auf Open.
   c. Wählen Sie Speichern.

So laden Sie ein ZIP-Archiv aus einem Amazon-S3-Bucket hoch (Konsole)

1. Wählen Sie auf der Funktionsseite der Lambda-Konsole die Funktion aus, für die Sie eine neue ZIP-Datei hochladen möchten.
2. Wählen Sie die Registerkarte Code aus.
3. Wählen Sie im Bereich Codequelle die Option Hochladen von aus.
4. Wählen Sie den Amazon-S3-Speicherort aus.
5. Fügen Sie die Amazon-S3-Link-URL Ihrer ZIP-Datei ein und wählen Sie Speichern aus.

Erstellen und Aktualisieren von Funktionen mit ZIP-Dateien unter Verwendung der AWS CLI

Sie können die AWS CLI verwenden, um eine neue Funktion zu erstellen oder eine vorhandene unter Verwendung einer ZIP-Datei zu aktualisieren. Verwenden Sie die Befehle Funktion erstellen und Funktionscode aktualisieren, um Ihr ZIP-Paket bereitzustellen. Wenn Ihre ZIP-Datei kleiner als 50 MB ist, können Sie das ZIP-Paket von einem Dateispeicherort auf Ihrem lokalen Build-Computer hochladen. Bei größeren Dateien müssen Sie Ihr ZIP-Paket aus einem Amazon-S3-Bucket hochladen. Anweisungen zum Hochladen einer Datei in einen Amazon-S3-Bucket unter Verwendung der AWS CLI finden Sie unter Verschieben von Objekten im AWS CLI-Benutzerhandbuch.

> **Note**
>
> Wenn Sie Ihre ZIP-Datei unter Verwendung der AWS CLI aus einem Amazon-S3-Bucket hochladen, muss sich der Bucket in der gleichen AWS-Region wie Ihre Funktion befinden.

Um eine neue Funktion unter Verwendung einer ZIP-Datei mit der AWS CLI zu erstellen, müssen Sie Folgendes angeben:

- Den Namen Ihrer Funktion (`--function-name`)
- Die Laufzeit Ihrer Funktion (`--runtime`)
- Den Amazon-Ressourcennamen (ARN) der Ausführungsrolle der Funktion (`--role`).
- Den Namen der Handler-Methode in Ihrem Funktionscode (`--handler`)

Sie müssen auch den Speicherort Ihrer ZIP-Datei angeben. Befindet sich Ihre ZIP-Datei in einem Ordner auf Ihrem lokalen Build-Computer, verwenden Sie die `--zip-file`-Option, um den Dateipfad anzugeben, wie im folgenden Beispielbefehl gezeigt.

```
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
```

Zur Angabe des Speicherorts der ZIP-Datei in einem Amazon-S3-Bucket verwenden Sie die im folgenden Beispielbefehl gezeigte `--code`-Option. Sie müssen den `S3ObjectVersion`-Parameter nur für versionierte Objekte verwenden.

```
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
```

Um eine vorhandene Funktion mit der CLI zu aktualisieren, geben Sie den Namen Ihrer Funktion unter Verwendung des `--function-name`-Parameters an. Sie müssen auch den Speicherort der ZIP-Datei angeben, die Sie zum Aktualisieren Ihres Funktionscodes verwenden möchten. Befindet sich Ihre ZIP-Datei in einem Ordner auf Ihrem lokalen Build-Computer, verwenden Sie die `--zip-file`-Option, um den Dateipfad anzugeben, wie im folgenden Beispielbefehl gezeigt.

```
aws lambda update-function-code --function-name myFunction \
--zip-file fileb://myFunction.zip
```

Zur Angabe des Speicherorts der ZIP-Datei in einem Amazon-S3-Bucket verwenden Sie die im folgenden Beispielbefehl gezeigten `--s3-bucket`- und `--s3-key`-Optionen. Sie müssen den `--s3-object-version`-Parameter nur für versionierte Objekte verwenden.

```
aws lambda update-function-code --function-name myFunction \
--s3-bucket amzn-s3-demo-bucket --s3-key myFileName.zip --s3-object-version myObject
 Version
```

Erstellen und Aktualisieren von Funktionen mit ZIP-Dateien unter Verwendung der Lambda-API

Um Funktionen zu erstellen und zu konfigurieren, die ein ZIP-Dateiarchiv verwenden, verwenden Sie die folgenden API-Operationen:

- CreateFunction
- UpdateFunctionCode

Erstellen und Aktualisieren von Funktionen mit ZIP-Dateien unter Verwendung von AWS SAM

Das AWS Serverless Application Model (AWS SAM) ist ein Toolkit, das den Prozess der Erstellung und Ausführung von Serverless-Anwendungen auf AWS optimiert. Sie definieren die Ressourcen für Ihre Anwendung in einer YAML- oder JSON-Vorlage und verwenden die AWS SAM-Befehlszeilenschnittstelle (AWS SAM CLI), um Ihre Anwendungen zu erstellen, zu verpacken und bereitzustellen. Wenn Sie eine Lambda-Funktion aus einer AWS SAM-Vorlage erstellen, erstellt AWS SAM automatisch ein ZIP-Bereitstellungspaket oder ein Container-Image mit Ihrem Funktionscode und allen von Ihnen angegebenen Abhängigkeiten. Weitere Informationen zur Verwendung von AWS SAM für die Erstellung und Bereitstellung von Lambda-Funktionen finden Sie unter Erste Schritte mit AWS SAM im AWS Serverless Application Model-Entwicklerhandbuch.

Sie können außerdem AWS SAM verwenden, um eine Lambda-Funktion unter Verwendung eines vorhandenen ZIP-Dateiarchivs zu erstellen. Um eine Lambda-Funktion unter Verwendung von AWS SAM zu erstellen, können Sie Ihre ZIP-Datei in einem Amazon-S3-Bucket oder in einem lokalen Ordner auf Ihrem Build-Computer speichern. Anweisungen zum Hochladen einer Datei in einen

Amazon-S3-Bucket unter Verwendung der AWS CLI finden Sie unter Verschieben von Objekten im AWS CLI-Benutzerhandbuch.

In Ihrer AWS SAM-Vorlage gibt die `AWS::Serverless::Function`-Ressource Ihre Lambda-Funktion an. Legen Sie in dieser Ressource die folgenden Eigenschaften fest, um eine Funktion zu erstellen, die als ZIP-Datei-Archiv definiert ist:

- `PackageType` – festlegen auf `Zip`
- `CodeUri` – festlegen auf die Amazon-S3-URI, den Pfad zum lokalen Ordner oder das Funktionscode-Objekt
- `Runtime` – festlegen auf die gewünschte Laufzeit

Mit AWS SAM müssen Sie Ihre ZIP-Datei nicht zuerst in einen Amazon-S3-Bucket hochladen, auch wenn sie größer als 50 MB ist. AWS SAM kann ZIP-Pakete bis zur maximal zulässigen Dateigröße von 250 MB (entpackt) von einem Speicherort auf Ihrem lokalen Build-Computer hochladen.

Weitere Informationen zur Bereitstellung von Funktionen mit einer ZIP-Datei in AWS SAM finden Sie unter AWS::Serverless::Function im AWS SAM-Entwicklerhandbuch.

Beispiel: Verwenden von AWS SAM zum Erstellen einer Go-Funktion mit „provided.al2023“

1. Erstellen Sie eine AWS SAM-Vorlage mit folgenden Eigenschaften:

   - BuildMethod: Gibt den Compiler für Ihre Anwendung an. Verwenden Sie `go1.x`.
   - Runtime: Verwenden Sie `provided.al2023`.
   - CodeUri: Geben Sie den Pfad zu Ihrem Code ein.
   - Architectures: Verwenden Sie `[arm64]` für die arm64-Architektur. Verwenden Sie für die x86_64-Befehlssatzarchitektur die Option `[amd64]` oder entfernen Sie die Eigenschaft `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. Verwenden Sie den Befehl sam build, um die ausführbare Datei zu kompilieren.

```
sam build
```

3. Verwenden Sie den Befehl sam deploy, um die Funktion in Lambda bereitzustellen.

```
sam deploy --guided
```

Erstellen und Aktualisieren von Funktionen mit ZIP-Dateien unter Verwendung von CloudFormation

Sie können CloudFormation verwenden, um eine Lambda-Funktion unter Verwendung eines ZIP-Dateiarchivs zu erstellen. Um eine Lambda-Funktion aus einer ZIP-Datei zu erstellen, müssen Sie Ihre Datei zunächst in einen Amazon-S3-Bucket hochladen. Anweisungen zum Hochladen einer Datei in einen Amazon-S3-Bucket unter Verwendung der AWS CLI finden Sie unter Verschieben von Objekten im AWS CLI-Benutzerhandbuch.

In Ihrer CloudFormation-Vorlage gibt die `AWS::Lambda::Function`-Ressource Ihre Lambda-Funktion an. Legen Sie in dieser Ressource die folgenden Eigenschaften fest, um eine Funktion zu erstellen, die als ZIP-Datei-Archiv definiert ist:

- `PackageType` – festlegen auf `Zip`
- `Code` – Geben Sie den Namen des Amazon-S3-Buckets und den ZIP-Dateinamen in die Felder `S3Bucket` und `S3Key` ein
- `Runtime` – festlegen auf die gewünschte Laufzeit

Die von CloudFormation generierte ZIP-Datei darf nicht größer als 4 MB sein. Weitere Informationen zur Bereitstellung von Funktionen mit einer ZIP-Datei in CloudFormation finden Sie unter AWS::Lambda::Function im CloudFormation-Entwicklerhandbuch.

Bereitstellen von Go-Lambda-Funktionen mit Container-Images

Es gibt zwei Möglichkeiten, ein Container-Image für eine Go-Lambda-Funktion zu erstellen:

- Verwenden eines AWS reinen Betriebssystem-Basisimages

  Go wird anders implementiert als andere verwaltete Laufzeiten. Da Go nativ zu einer ausführbaren Binärdatei kompiliert wird, ist keine spezielle Sprachlaufzeit erforderlich. Verwenden Sie ein Basis-Image nur für das Betriebssystem, um Go-Images für Lambda zu erstellen. Um das Image mit Lambda kompatibel zu machen, müssen Sie das `aws-lambda-go/lambda`-Paket in das Image aufnehmen.
- Verwenden Sie ein Nicht-Basis-Image AWS

  Sie können auch ein alternatives Basis-Image aus einer anderen Container-Registry verwenden. Sie können auch ein von Ihrer Organisation erstelltes benutzerdefiniertes Image verwenden. Um das Image mit Lambda kompatibel zu machen, müssen Sie das `aws-lambda-go/lambda`-Paket in das Image aufnehmen.

> **Tip**
>
> Um die Zeit zu reduzieren, die benötigt wird, bis Lambda-Container-Funktionen aktiv werden, siehe die Docker-Dokumentation unter Verwenden mehrstufiger Builds. Um effiziente Container-Images zu erstellen, folgen Sie den Bewährte Methoden für das Schreiben von Dockerfiles.

Auf dieser Seite wird erklärt, wie Sie Container-Images für Lambda erstellen, testen und bereitstellen.

AWS Basis-Images für die Bereitstellung von Go-Funktionen

Go wird anders implementiert als andere verwaltete Laufzeiten. Da Go nativ zu einer ausführbaren Binärdatei kompiliert wird, ist keine spezielle Sprachlaufzeit erforderlich. Verwenden Sie ein reines Betriebssystem-Basisimage, um Go-Funktionen für Lambda bereitzustellen.

Name	ID	Betriebssystem	Datum der Veraltung	Blockfunktion erstellen	Blockfunktion aktualisieren
Reine OS-Laufzeit	`provided.al2023`	Amazon Linux 2023	30. Juni 2029	31. Juli 2029	31. August 2029
Reine OS-Laufzeit	`provided.al2`	Amazon Linux 2	31. Juli 2026	31. August 2026	30. September 2026

Öffentliche Galerie von Amazon Elastic Container Registry: gallery.ecr. aws/lambda/provided

Laufzeitschnittstellen-Clients von Go

Das `aws-lambda-go/lambda`-Paket enthält eine Implementierung der Laufzeitschnittstelle. Beispiele für die Verwendung von `aws-lambda-go/lambda` in Ihrem Image finden Sie unter Verwenden eines AWS reinen Betriebssystem-Basisimages oder Verwenden Sie ein Nicht-Basis-Image AWS.

Verwenden eines AWS reinen Betriebssystem-Basisimages

Go wird anders implementiert als andere verwaltete Laufzeiten. Da Go nativ zu einer ausführbaren Binärdatei kompiliert wird, ist keine spezielle Sprachlaufzeit erforderlich. Verwenden Sie ein reines Betriebssystem-Basis-Image, um Container-Images für Go-Funktionen zu erstellen.

Tags (Markierungen)	Laufzeit	Betriebssystem	Dockerfile	Ablehnung
al2023	Reine OS-Laufzeit	Amazon Linux 2023	Dockerfile für reine Betriebssystem-Runtime aktiviert GitHub	30. Juni 2029
al2	Reine OS-Laufzeit	Amazon Linux 2	Dockerfile nur für Betriebssystem, Runtime aktiviert GitHub	31. Juli 2026

Weitere Informationen zu diesen Basis-Images finden Sie in der öffentlichen Amazon-ECR-Galerie unter provided.

Sie müssen das aws-lambda-go/lambda-Paket in Ihren Go-Handler aufnehmen. Dieses Paket implementiert das Programmiermodell für Go, einschließlich der Laufzeitschnittstelle.

Voraussetzungen

Zur Durchführung der Schritte in diesem Abschnitt benötigen Sie Folgendes:

- AWS CLI Version 2
- Docker (Mindestversion 25.0.0)
- Das Docker-buildx-Plugin.
- Go

Erstellen eines Images aus dem Basis-Image „provided.al2023“

So erstellen und implementieren Sie eine Go-Funktion mit dem **`provided.al2023`**-Basis-Image.

1. Erstellen Sie ein Verzeichnis für das Projekt und wechseln Sie dann zu diesem Verzeichnis.

```
mkdir hello
cd hello
```

2. Initialisieren Sie ein neues Go-Modul.

```
go mod init example.com/hello-world
```

3. Fügen Sie die Lambda-Bibliothek als Abhängigkeit Ihres neuen Moduls hinzu.

```
go get github.com/aws/aws-lambda-go/lambda
```

4. Erstellen Sie eine Datei namens `main.go` und öffnen Sie diese dann in einem Text-Editor. Dies ist der Code für die Lambda-Funktion. Sie können den folgenden Beispielcode zum Testen verwenden oder ihn durch Ihren eigenen ersetzen.

```
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. Verwenden Sie einen Texteditor, um ein Dockerfile in Ihrem Projektverzeichnis zu erstellen.

   - Die folgende Beispiel-Docker-Datei verwendet eine mehrstufige Entwicklung. Dadurch können Sie in jedem Schritt ein anderes Basis-Image verwenden. Sie können ein Image verwenden, z. B. ein Go-Basis-Image, um Ihren Code zu kompilieren und die ausführbare Binärdatei zu erstellen. Sie können dann ein anderes Image verwenden, z. B. `provided.al2023` in der `FROM`-Abschlusserklärung, um das Image zu definieren, das Sie für Lambda bereitstellen. Der Build-Prozess ist vom endgültigen Bereitstellungs-Image getrennt, so dass das endgültige Image nur die Dateien enthält, die zum Ausführen der Anwendung benötigt werden.
   - Sie können das optionale Tag `lambda.norpc` verwenden, um die RPC-Komponente (Remote Procedure Call, Remoteprozeduraufruf) der Bibliothek lambda auszuschließen. Die RPC-Komponente ist nur erforderlich, wenn Sie die veraltete Go 1.x-Laufzeit verwenden. Durch Ausschließen der RPC-Komponente verringert sich die Größe des Bereitstellungspakets.
   - Beachten Sie, dass das Dockerfile-Beispiel keine USER-Anweisung enthält. Wenn Sie ein Container-Image für Lambda bereitstellen, definiert Lambda automatisch einen Standard-Linux-Benutzer mit Berechtigungen mit geringsten Rechten. Dies unterscheidet sich vom Standardverhalten von Docker, bei dem standardmäßig der `root`-Benutzer verwendet wird, wenn keine `USER`-Anweisung bereitgestellt wird.

Example– Mehrstufige Docker-Datei

> **Note**
>
> Stellen Sie sicher, dass die Version von Go, die Sie in der Docker-Datei angeben (z. B. `golang:1.20`), dieselbe Version von Go ist, mit der Sie Ihre Anwendung erstellt haben.

```
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. Erstellen Sie Ihr Docker-Image mit dem docker build-Befehl. Das folgende Beispiel benennt das Bild in `docker-image` und gibt ihm den `test` Tag. Um Ihr Image mit Lambda kompatibel zu machen, müssen Sie die Option `--provenance=false` verwenden.

```
docker buildx build --platform linux/amd64 --provenance=false -t docker-image:test
 .
```

> **Note**
>
> Der Befehl gibt die `--platform linux/amd64`-Option an, um sicherzustellen, dass Ihr Container mit der Lambda-Ausführungsumgebung kompatibel ist, unabhängig von der Architektur des Entwicklungsrechners. Wenn Sie beabsichtigen, eine Lambda-Funktion mithilfe der ARM64 Befehlssatzarchitektur zu erstellen, müssen Sie den Befehl so ändern, dass er stattdessen die `--platform linux/arm64` Option verwendet.

(Optional) Testen Sie das Image lokal

Verwenden Sie den Laufzeit-Schnittstellen-Emulator, um Ihr Image lokal zu testen. Der Laufzeit-Schnittstellen-Emulator ist im `provided.al2023`-Basis-Image enthalten.

So führen Sie den Laufzeit-Schnittstellen-Emulator auf Ihrem lokalen Computer aus

1. Starten Sie Ihr Docker-Image mit dem docker run-Befehl. Beachten Sie Folgendes:

   - `docker-image` ist der Image-Name und `test` ist das Tag.
   - `./main` ist `ENTRYPOINT` aus Ihrem Dockerfile.

   ```
   docker run -d -p 9000:8080 \
   --entrypoint /usr/local/bin/aws-lambda-rie \
   docker-image:test ./main
   ```

   Dieser Befehl führt das Image als Container aus und erstellt einen lokalen Endpunkt bei `localhost:9000/2015-03-31/functions/function/invocations`.

2. Veröffentlichen Sie in einem neuen Terminalfenster ein Ereignis mit einem curl-Befehl an den folgenden Endpunkt:

   ```
   curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{}'
   ```

   Dieser Befehl ruft die Funktion mit einem leeren Ereignis auf und gibt eine Antwort zurück. Für einige Funktionen ist möglicherweise eine JSON-Nutzlast erforderlich. Beispiel:

   ```
   curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d
    '{"payload":"hello world!"}'
   ```

3. Die Container-ID erhalten.

   ```
   docker ps
   ```

4. Verwenden Sie den Befehl docker kill, um den Container zu anzuhalten. Ersetzen Sie in diesem Befehl `3766c4ab331c` durch die Container-ID aus dem vorherigen Schritt.

   ```
   docker kill 3766c4ab331c
   ```

Das Image bereitstellen

Um das Image in Amazon ECR hochzuladen und die Lambda-Funktion zu erstellen

1. Führen Sie den get-login-passwordBefehl aus, um die Docker-CLI bei Ihrer Amazon ECR-Registrierung zu authentifizieren.

   - Setzen Sie den `--region` Wert auf den AWS-Region Ort, an dem Sie das Amazon ECR-Repository erstellen möchten.
   - Ersetzen Sie es `111122223333` durch Ihre AWS-Konto ID.

   ```
   aws ecr get-login-password --region us-east-1 | docker login --username AWS --
   password-stdin 111122223333.dkr.ecr.us-east-1.amazonaws.com
   ```

2. Erstellen Sie ein Repository in Amazon ECR mithilfe des Befehls create-repository.

   ```
   aws ecr create-repository --repository-name hello-world --region us-east-1 --image-
   scanning-configuration scanOnPush=true --image-tag-mutability MUTABLE
   ```

   > **Note**
   >
   > Das Amazon ECR-Repository muss sich im selben Format AWS-Region wie die Lambda-Funktion befinden.

   Wenn erfolgreich, sehen Sie eine Antwort wie diese:

   ```
   {
       "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. Kopieren Sie das `repositoryUri` aus der Ausgabe im vorherigen Schritt.
4. Führen Sie den Befehl docker tag aus, um Ihr lokales Image als neueste Version in Ihrem Amazon-ECR-Repository zu markieren. In diesem Befehl gilt Folgendes:

   - `docker-image:test` ist der Name und das Tag Ihres Docker-Images. Dies sind der Imagename und das Tag, die Sie im `docker build`-Befehl angegeben haben.
   - Ersetzen Sie `<ECRrepositoryUri>` durch den `repositoryUri`, den Sie kopiert haben. Stellen Sie sicher, dass Sie `:latest` am Ende der URI angeben.

   ```
   docker tag docker-image:test <ECRrepositoryUri>:latest
   ```

   Beispiel:

   ```
   docker tag docker-image:test 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
   ```

5. Führen Sie den Befehl docker push aus, um Ihr lokales Image im Amazon-ECR-Repository bereitzustellen. Stellen Sie sicher, dass Sie `:latest` am Ende der Repository-URI angeben.

   ```
   docker push 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
   ```

6. Erstellen Sie eine Ausführungsrolle für die Funktion, wenn Sie noch keine haben. Sie benötigen den Amazon-Ressourcennamen (ARN) der Rolle im nächsten Schritt.
7. So erstellen Sie die Lambda-Funktion: Geben Sie für `ImageUri` die Repository-URI von zuvor an. Stellen Sie sicher, dass Sie `:latest` am Ende der URI angeben.

   ```
   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**
>
> Sie können eine Funktion mit einem Bild in einem anderen AWS Konto erstellen, sofern sich das Bild in derselben Region wie die Lambda-Funktion befindet. Weitere Informationen finden Sie unter Kontoübergreifende Berechtigungen von Amazon ECR.

8. Die Funktion aufrufen.

```
aws lambda invoke --function-name hello-world response.json
```

Das Ergebnis sollte ungefähr wie folgt aussehen:

```
{
  "ExecutedVersion": "$LATEST",
  "StatusCode": 200
}
```

9. Um die Ausgabe der Funktion zu sehen, überprüfen Sie die `response.json`-Datei.

Um den Funktionscode zu aktualisieren, müssen Sie das Image erneut erstellen, das neue Image in das Amazon ECR-Repository hochladen und dann den update-function-codeBefehl verwenden, um das Image für die Lambda-Funktion bereitzustellen.

Lambda löst das Image-Tag in einen bestimmten Image-Digest auf. Das heißt, wenn Sie das Image-Tag, das zur Bereitstellung der Funktion verwendet wurde, auf ein neues Image in Amazon ECR verweisen, aktualisiert Lambda die Funktion nicht automatisch, um das neue Image zu verwenden.

Um das neue Image für dieselbe Lambda-Funktion bereitzustellen, müssen Sie den update-function-codeBefehl verwenden, auch wenn das Image-Tag in Amazon ECR gleich bleibt. Im folgenden Beispiel erstellt die `--publish`-Option eine neue Version der Funktion unter Verwendung des aktualisierten Container-Images.

```
aws lambda update-function-code \
  --function-name hello-world \
  --image-uri 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --publish
```

Verwenden Sie ein Nicht-Basis-Image AWS

Sie können ein Container-Image für Go aus einem AWS Nicht-Base-Image erstellen. Das Beispiel-Dockerfile in den folgenden Schritten verwendet ein Alpine-Basis-Image.

Sie müssen das aws-lambda-go/lambda-Paket in Ihren Go-Handler aufnehmen. Dieses Paket implementiert das Programmiermodell für Go, einschließlich der Laufzeitschnittstelle.

Voraussetzungen

Zur Durchführung der Schritte in diesem Abschnitt benötigen Sie Folgendes:

- AWS CLI Version 2
- Docker (Mindestversion 25.0.0)
- Das Docker-buildx-Plugin.
- Go

Erstellen eines Images aus einem alternativen Basis-Image

Erstellen und Bereitstellen einer Go-Funktion mit einem Alpine-Basis-Image

1. Erstellen Sie ein Verzeichnis für das Projekt und wechseln Sie dann zu diesem Verzeichnis.

```
mkdir hello
cd hello
```

2. Initialisieren Sie ein neues Go-Modul.

```
go mod init example.com/hello-world
```

3. Fügen Sie die Lambda-Bibliothek als Abhängigkeit Ihres neuen Moduls hinzu.

```
go get github.com/aws/aws-lambda-go/lambda
```

4. Erstellen Sie eine Datei namens `main.go` und öffnen Sie diese dann in einem Text-Editor. Dies ist der Code für die Lambda-Funktion. Sie können den folgenden Beispielcode zum Testen verwenden oder ihn durch Ihren eigenen ersetzen.

```
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. Verwenden Sie einen Texteditor, um ein Dockerfile in Ihrem Projektverzeichnis zu erstellen. Das folgende Beispiel-Dockerfile verwendet ein Alpine-Basis-Image. Beachten Sie, dass das Dockerfile-Beispiel keine USER-Anweisung enthält. Wenn Sie ein Container-Image für Lambda bereitstellen, definiert Lambda automatisch einen Standard-Linux-Benutzer mit Berechtigungen mit geringsten Rechten. Dies unterscheidet sich vom Standardverhalten von Docker, bei dem standardmäßig der `root`-Benutzer verwendet wird, wenn keine USER-Anweisung bereitgestellt wird.

   Example Dockerfile

   > **Note**
   >
   > Stellen Sie sicher, dass die Version von Go, die Sie in der Docker-Datei angeben (z. B. `golang:1.20`), dieselbe Version von Go ist, mit der Sie Ihre Anwendung erstellt haben.

```
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. Erstellen Sie Ihr Docker-Image mit dem docker build-Befehl. Das folgende Beispiel benennt das Bild in `docker-image` und gibt ihm den `test` Tag. Um Ihr Image mit Lambda kompatibel zu machen, müssen Sie die Option `--provenance=false` verwenden.

```
docker buildx build --platform linux/amd64 --provenance=false -t docker-image:test
 .
```

> **Note**
>
> Der Befehl gibt die `--platform linux/amd64`-Option an, um sicherzustellen, dass Ihr Container mit der Lambda-Ausführungsumgebung kompatibel ist, unabhängig von der Architektur des Entwicklungsrechners. Wenn Sie beabsichtigen, eine Lambda-Funktion mithilfe der ARM64 Befehlssatzarchitektur zu erstellen, müssen Sie den Befehl so ändern, dass er stattdessen die `--platform linux/arm64` Option verwendet.

(Optional) Testen Sie das Image lokal

Verwenden Sie den Laufzeit-Schnittstellen-Emulator, um das Image lokal zu testen. Sie können den Emulator in Ihr Image integrieren oder ihn mit dem folgenden Verfahren auf Ihrem lokalen Computer installieren.

Installieren des Laufzeitschnittstellen-Emulators auf Ihrem lokalen Computer

1. Führen Sie in Ihrem Projektverzeichnis den folgenden Befehl aus, um den Runtime-Interface-Emulator (x86-64-Architektur) herunterzuladen GitHub und auf Ihrem lokalen Computer zu installieren.

   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
```

Um den arm64-Emulator zu installieren, ersetzen Sie die GitHub Repository-URL im vorherigen Befehl durch Folgendes:

```
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
```

Um den arm64-Emulator zu installieren, ersetzen Sie das `$downloadLink` durch Folgendes:

```
https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/
download/aws-lambda-rie-arm64
```

2. Starten Sie Ihr Docker-Image mit dem docker run-Befehl. Beachten Sie Folgendes:

   - `docker-image` ist der Image-Name und `test` ist das Tag.
   - `/main` ist `ENTRYPOINT` aus Ihrem 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
```

Dieser Befehl führt das Image als Container aus und erstellt einen lokalen Endpunkt bei `localhost:9000/2015-03-31/functions/function/invocations`.

> **Note**
>
> Wenn Sie das Docker-Image für die ARM64 Befehlssatzarchitektur erstellt haben, stellen Sie sicher, dass Sie die `--platform linux/arm64` Option anstelle von verwenden. `--platform linux/amd64`

3. Veröffentlichen Sie ein Ereignis auf dem lokalen Endpunkt.

Linux/macOS

Führen Sie unter Linux oder macOS den folgenden `curl`-Befehl aus:

```
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{}'
```

Dieser Befehl ruft die Funktion mit einem leeren Ereignis auf und gibt eine Antwort zurück. Wenn Sie Ihren eigenen Funktionscode anstelle des Beispielfunktionscodes verwenden, wird empfohlen, die Funktion mit einer JSON-Nutzlast aufzurufen. Beispiel:

```
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d
 '{"payload":"hello world!"}'
```

PowerShell

Führen Sie in PowerShell den folgenden `Invoke-WebRequest` Befehl aus:

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{}' -ContentType "application/json"
```

Dieser Befehl ruft die Funktion mit einem leeren Ereignis auf und gibt eine Antwort zurück. Wenn Sie Ihren eigenen Funktionscode anstelle des Beispielfunktionscodes verwenden, wird empfohlen, die Funktion mit einer JSON-Nutzlast aufzurufen. Beispiel:

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{"payload":"hello world!"}' -ContentType
 "application/json"
```

4. Die Container-ID erhalten.

```
docker ps
```

5. Verwenden Sie den Befehl docker kill, um den Container zu anzuhalten. Ersetzen Sie in diesem Befehl `3766c4ab331c` durch die Container-ID aus dem vorherigen Schritt.

```
docker kill 3766c4ab331c
```

Das Image bereitstellen

Um das Image in Amazon ECR hochzuladen und die Lambda-Funktion zu erstellen

1. Führen Sie den get-login-passwordBefehl aus, um die Docker-CLI bei Ihrer Amazon ECR-Registrierung zu authentifizieren.

   - Setzen Sie den `--region` Wert auf den AWS-Region Ort, an dem Sie das Amazon ECR-Repository erstellen möchten.
   - Ersetzen Sie es `111122223333` durch Ihre AWS-Konto ID.

```
aws ecr get-login-password --region us-east-1 | docker login --username AWS --
password-stdin 111122223333.dkr.ecr.us-east-1.amazonaws.com
```

2. Erstellen Sie ein Repository in Amazon ECR mithilfe des Befehls create-repository.

```
aws ecr create-repository --repository-name hello-world --region us-east-1 --image-
scanning-configuration scanOnPush=true --image-tag-mutability MUTABLE
```

> **Note**
>
> Das Amazon ECR-Repository muss sich im selben Format AWS-Region wie die Lambda-Funktion befinden.

Wenn erfolgreich, sehen Sie eine Antwort wie diese:

```
{
    "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. Kopieren Sie das `repositoryUri` aus der Ausgabe im vorherigen Schritt.
4. Führen Sie den Befehl docker tag aus, um Ihr lokales Image als neueste Version in Ihrem Amazon-ECR-Repository zu markieren. In diesem Befehl gilt Folgendes:

   - `docker-image:test` ist der Name und das Tag Ihres Docker-Images. Dies sind der Imagename und das Tag, die Sie im `docker build`-Befehl angegeben haben.
   - Ersetzen Sie <ECRrepositoryUri> durch den `repositoryUri`, den Sie kopiert haben. Stellen Sie sicher, dass Sie `:latest` am Ende der URI angeben.

```
docker tag docker-image:test <ECRrepositoryUri>:latest
```

Beispiel:

```
docker tag docker-image:test 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

5. Führen Sie den Befehl docker push aus, um Ihr lokales Image im Amazon-ECR-Repository bereitzustellen. Stellen Sie sicher, dass Sie `:latest` am Ende der Repository-URI angeben.

```
docker push 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

6. Erstellen Sie eine Ausführungsrolle für die Funktion, wenn Sie noch keine haben. Sie benötigen den Amazon-Ressourcennamen (ARN) der Rolle im nächsten Schritt.
7. So erstellen Sie die Lambda-Funktion: Geben Sie für `ImageUri` die Repository-URI von zuvor an. Stellen Sie sicher, dass Sie `:latest` am Ende der URI angeben.

```
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**
>
> Sie können eine Funktion mit einem Bild in einem anderen AWS Konto erstellen, sofern sich das Bild in derselben Region wie die Lambda-Funktion befindet. Weitere Informationen finden Sie unter Kontoübergreifende Berechtigungen von Amazon ECR.

8. Die Funktion aufrufen.

```
aws lambda invoke --function-name hello-world response.json
```

Das Ergebnis sollte ungefähr wie folgt aussehen:

```
{
  "ExecutedVersion": "$LATEST",
  "StatusCode": 200
}
```

9. Um die Ausgabe der Funktion zu sehen, überprüfen Sie die `response.json`-Datei.

Um den Funktionscode zu aktualisieren, müssen Sie das Image erneut erstellen, das neue Image in das Amazon ECR-Repository hochladen und dann den update-function-codeBefehl verwenden, um das Image für die Lambda-Funktion bereitzustellen.

Lambda löst das Image-Tag in einen bestimmten Image-Digest auf. Das heißt, wenn Sie das Image-Tag, das zur Bereitstellung der Funktion verwendet wurde, auf ein neues Image in Amazon ECR verweisen, aktualisiert Lambda die Funktion nicht automatisch, um das neue Image zu verwenden.

Um das neue Image für dieselbe Lambda-Funktion bereitzustellen, müssen Sie den update-function-codeBefehl verwenden, auch wenn das Image-Tag in Amazon ECR gleich bleibt. Im folgenden Beispiel erstellt die `--publish`-Option eine neue Version der Funktion unter Verwendung des aktualisierten Container-Images.

```
aws lambda update-function-code \
  --function-name hello-world \
  --image-uri 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --publish
```

Arbeiten mit Ebenen für Go Lambda-Funktionen

Wir empfehlen nicht, Ebenen zu verwenden, um Abhängigkeiten für in Go geschriebene Lambda-Funktionen zu verwalten. Dies liegt daran, dass Lambda-Funktionen in Go zu einer einzigen ausführbaren Datei kompiliert werden, die Sie Lambda bei der Bereitstellung Ihrer Funktion zur Verfügung stellen. Diese ausführbare Datei enthält Ihren kompilierten Funktionscode zusammen mit all seinen Abhängigkeiten. Die Verwendung von Ebenen verkompliziert nicht nur diesen Vorgang, sondern führt auch zu längeren Kaltstartzeiten, da Ihre Funktionen während der Init-Phase zusätzliche Assemblys manuell in den Speicher laden müssen.

Um externe Abhängigkeiten mit Ihren Go-Handlern zu verwenden, nehmen Sie sie direkt in Ihr Bereitstellungspaket auf. Auf diese Weise vereinfachen Sie den Bereitstellungsprozess und nutzen auch die Vorteile der integrierten Go-Compiler-Optimierungen. Ein Beispiel dafür, wie Sie eine Abhängigkeit wie das AWS SDK for Go in Ihrer Funktion importieren und verwenden, finden Sie unter the section called “Handler”.

Go-Lambda-Funktionen protokollieren und überwachen

AWS Lambda überwacht automatisch Lambda-Funktionen in Ihrem Namen und sendet Protokolle an Amazon CloudWatch. Ihre Lambda-Funktion enthält eine CloudWatch Logs-Log-Gruppe und einen Log-Stream für jede Instanz Ihrer Funktion. Die Lambda-Laufzeitumgebung sendet Details zu den einzelnen Aufrufen an den Protokollstream und leitet Protokolle und andere Ausgaben aus dem Code Ihrer Funktion weiter. Weitere Informationen finden Sie unter Senden von Lambda-Funktionsprotokollen an CloudWatch Logs.

Auf dieser Seite wird beschrieben, wie Sie eine Protokollausgabe aus dem Code Ihrer Lambda-Funktion erstellen und mithilfe der AWS Command Line Interface Lambda-Konsole oder der CloudWatch Konsole auf Logs zugreifen.

Sections

- Erstellen einer Funktion, die Protokolle zurückgibt
- Anzeigen von Protokollen in der Lambda-Konsole
- Protokolle in der Konsole anzeigen CloudWatch
- Logs mit dem Befehl () anzeigen AWS Command Line InterfaceAWS CLI
- Löschen von Protokollen

Erstellen einer Funktion, die Protokolle zurückgibt

Um Protokolle aus dem Code Ihrer Funktion auszugeben, können Sie Methoden für das Fmt-Paket verwenden oder eine Protokollierungsbibliothek, die zu `stdout` oder `stderr` schreibt. Im folgenden Beispiel wird das Protokollpaket verwendet.

Example main.go – Protokollierung

```
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 Protokollformat

```
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
```

Die Go-Laufzeit protokolliert die Zeilen `START`, `END` und `REPORT` für jeden Aufruf. Die Berichtszeile enthält die folgenden Details.

Datenfelder für REPORT-Zeilen

- RequestId— Die eindeutige Anforderungs-ID für den Aufruf.
- Dauer – Die Zeit, die die Handler-Methode Ihrer Funktion mit der Verarbeitung des Ereignisses verbracht hat.
- Fakturierte Dauer – Die für den Aufruf fakturierte Zeit.

- Speichergröße – Die der Funktion zugewiesene Speichermenge.
- Max. verwendeter Speicher – Die Speichermenge, die von der Funktion verwendet wird. Wenn Aufrufe eine Ausführungsumgebung gemeinsam nutzen, meldet Lambda den maximalen Speicherverbrauch für alle Aufrufe. Dieses Verhalten kann zu einem höheren als erwarteten gemeldeten Wert führen.
- Initialisierungsdauer – Für die erste Anfrage die Zeit, die zur Laufzeit zum Laden der Funktion und Ausführen von Code außerhalb der Handler-Methode benötigt wurde.
- XRAY TraceId — Für verfolgte Anfragen die AWS X-Ray Trace-ID.
- SegmentId— Für verfolgte Anfragen die X-Ray-Segment-ID.
- Stichprobe – Bei verfolgten Anforderungen das Stichprobenergebnis.

Anzeigen von Protokollen in der Lambda-Konsole

Sie können die Lambda-Konsole verwenden, um die Protokollausgabe nach dem Aufrufen einer Lambda-Funktion anzuzeigen.

Wenn Ihr Code über den eingebetteten Code-Editor getestet werden kann, finden Sie Protokolle in den Ausführungsergebnissen. Wenn Sie das Feature Konsolentest verwenden, um eine Funktion aufzurufen, finden Sie die Protokollausgabe im Abschnitt Details.

Protokolle in der Konsole anzeigen CloudWatch

Sie können die CloudWatch Amazon-Konsole verwenden, um Protokolle für alle Lambda-Funktionsaufrufe anzuzeigen.

Um Protokolle auf der Konsole anzuzeigen CloudWatch

1. Öffnen Sie die Seite Protokollgruppen auf der CloudWatch Konsole.
2. Wählen Sie die Protokollgruppe für Ihre Funktion (***your-function-name***/aws/lambda/).
3. Wählen Sie eine Protokollstream aus.

Jeder Protokoll-Stream entspricht einer Instance Ihrer Funktion. Ein Protokollstream wird angezeigt, wenn Sie Ihre Lambda-Funktion aktualisieren und wenn zusätzliche Instanzen für gleichzeitige Aufrufe erstellt werden. Um Protokolle für einen bestimmten Aufruf zu finden, empfehlen wir, Ihre Funktion mit zu instrumentieren. AWS X-Ray X-Ray erfasst Details zu der Anforderung und dem Protokollstream in der Trace.

Logs mit dem Befehl () anzeigen AWS Command Line InterfaceAWS CLI

Das AWS CLI ist ein Open-Source-Tool, mit dem Sie mithilfe von Befehlen in Ihrer Befehlszeilen-Shell mit AWS Diensten interagieren können. Zur Durchführung der Schritte in diesem Abschnitt benötigen Sie AWS CLI Version 2.

Sie können die AWS CLI verwenden, um Protokolle für einen Aufruf mit der `--log-type`-Befehlsoption abzurufen. Die Antwort enthält das Feld `LogResult`, das bis zu 4 KB base64-verschlüsselte Protokolle aus dem Aufruf enthält.

Example eine Log-ID abrufen

Das folgende Beispiel zeigt, wie eine Protokoll-ID aus dem `LogResult`-Feld für eine Funktion namens `my-function` abgerufen wird.

```
aws lambda invoke --function-name my-function out --log-type Tail
```

Die Ausgabe sollte folgendermaßen aussehen:

```
{
    "StatusCode": 200,
    "LogResult":
 "U1RBUlQgUmVxdWVzdElkOiA4N2QwNDRiOC1mMTU0LTExZTgtOGNkYS0yOTc0YzVlNGZiMjEgVmVyc2lvb...",
    "ExecutedVersion": "$LATEST"
}
```

Example entschlüsseln der Protokolle

Verwenden Sie in derselben Eingabeaufforderung das `base64`-Dienstprogramm, um die Protokolle zu entschlüsseln. Das folgende Beispiel zeigt, wie Base64-codierte Logs für abgerufen werde `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
```

Die cli-binary-format Option ist erforderlich, wenn Sie AWS CLI Version 2 verwenden. Um dies zur Standardeinstellung zu machen, führen Sie `aws configure set cli-binary-format raw-in-base64-out` aus. Weitere Informationen finden Sie unter Von AWS CLI unterstützte globale Befehlszeilenoptionen im AWS Command Line Interface -Benutzerhandbuch für Version 2.

Die Ausgabe sollte folgendermaßen aussehen:

```
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
```

Das `base64`-Dienstprogramm ist unter Linux, macOS und Ubuntu auf Windowsverfügbar. macOS-Benutzer müssen möglicherweise `base64 -D` verwenden.

Example get-logs.sh-Skript

Verwenden Sie in derselben Eingabeaufforderung das folgende Skript, um die letzten fünf Protokollereignisse herunterzuladen. Das Skript verwendet `sed` zum Entfernen von Anführungszeichen aus der Ausgabedatei und wechselt 15 Sekunden lang in den Ruhezustand, um Zeit einzuräumen, damit Protokolle verfügbar werden können. Die Ausgabe enthält die Antwort von Lambda und die `get-log-events`Ausgabe des Befehls.

Kopieren Sie den Inhalt des folgenden Codebeispiels und speichern Sie es in Ihrem Lambda-Projektverzeichnis unter `get-logs.sh`.

Die cli-binary-format Option ist erforderlich, wenn Sie AWS CLI Version 2 verwenden. Um dies zur Standardeinstellung zu machen, führen Sie `aws configure set cli-binary-format raw-in-base64-out` aus. Weitere Informationen finden Sie unter Von AWS CLI unterstützte globale Befehlszeilenoptionen im AWS Command Line Interface -Benutzerhandbuch für 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 und Linux (nur diese Systeme)

In derselben Eingabeaufforderung müssen macOS- und Linux-Benutzer möglicherweise den folgenden Befehl ausführen, um sicherzustellen, dass das Skript ausführbar ist.

```
chmod -R 755 get-logs.sh
```

Example die letzten fünf Protokollereignisse abrufen

Führen Sie an derselben Eingabeaufforderung das folgende Skript aus, um die letzten fünf Protokollereignisse abzurufen.

```
./get-logs.sh
```

Die Ausgabe sollte folgendermaßen aussehen:

```
{
    "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"
}
```


Löschen von Protokollen

Wenn Sie eine Funktion löschen, werden Protokollgruppen nicht automatisch gelöscht. Um das unbegrenzte Speichern von Protokollen zu vermeiden, löschen Sie die Protokollgruppe oder konfigurieren Sie eine Aufbewahrungszeitraum nach dem Protokolle automatisch gelöscht werden.

Instrumentieren von Go-Code in AWS Lambda

Lambda ist mit AWS X-Ray integriert, um Ihnen zu helfen, Lambda-Anwendungen zu verfolgen, zu debuggen und zu optimieren. Sie können mit X-Ray eine Anforderung verfolgen, während sie Ressourcen in Ihrer Anwendung durchläuft, die Lambda-Funktionen und andere AWS-Services enthalten können.

Um Protokollierungsdaten an X-Ray zu senden, können Sie eine von zwei SDK-Bibliotheken verwenden:

- AWS Distro for OpenTelemetry (ADOT) – Eine sichere, produktionsbereite, AWS-unterstützte Verteilung des OpenElemetrie (OTel)-SDK.
- AWS X-Ray SDK für Go – Ein SDK zum Generieren und Senden von Nachverfolgungsdaten an X-Ray.

Jedes der SDKs bietet Möglichkeiten, Ihre Telemetriedaten an den X-Ray Service zu senden. Sie können dann mit X-Ray die Leistungsmetriken Ihrer Anwendung anzeigen, filtern und erhalten, um Probleme und Möglichkeiten zur Optimierung zu identifizieren.

> **Important**
>
> X-Ray und Powertools für AWS Lambda-SDKs sind Teil einer eng integrierten Instrumentierungslösung von AWS. Die ADOT Lambda Layers sind Teil eines branchenweiten Standards für die Verfolgung von Instrumenten, die im Allgemeinen mehr Daten erfassen, aber möglicherweise nicht für alle Anwendungsfälle geeignet sind. Sie können mit beiden Lösungen eine durchgehende Ablaufverfolgung in X-Ray implementieren. Weitere Informationen zur Auswahl zwischen ihnen finden Sie unter Auswählen zwischen der AWS-Distro für Open Telemetry und X-Ray-SDKs.

Abschnitte

- Verwenden von ADOT zur Instrumentierung Ihrer Go-Funktionen
- Instrumentierung Ihrer Go-Funktionen mithilfe von X-Ray-SDK
- Aktivieren der Nachverfolgung mit der Lambda-Konsole
- Aktivieren der Nachverfolgung mit der Lambda-API
- Aktivieren von Nachverfolgungen mit CloudFormation

- Interpretieren einer X-Ray-Nachverfolgung

Verwenden von ADOT zur Instrumentierung Ihrer Go-Funktionen

ADOT bietet vollständig verwaltete Lambda-Ebenen, die alles packen, was Sie zum Sammeln von Telemetriedaten mit dem OTel-SDK benötigen. Indem Sie diese Ebene verwenden, können Sie Ihre Lambda-Funktionen instrumentieren, ohne einen Funktionscode ändern zu müssen. Sie können Ihren Ebenen auch für die benutzerdefinierte Initialisierung von OTel konfigurieren. Weitere Informationen finden Sie unter Benutzerdefinierte Konfiguration für den ADOT Collector auf Lambda in der ADOT-Dokumentation.

Für Go-Laufzeiten können Sie den AWS-verwaltete Lambda-Ebene für ADOT Go hinzufügen, um Ihre Funktionen automatisch zu instrumentieren. Ausführliche Anweisungen zum Hinzufügen dieser Ebene finden Sie unter AWS-Distro für OpenTelemetry-Lambda-Unterstützung für Go in der ADOT-Dokumentation.

Instrumentierung Ihrer Go-Funktionen mithilfe von X-Ray-SDK

Um Details zu Aufrufen aufzuzeichnen, die Ihre Lambda-Funktion an andere Ressourcen in Ihrer Anwendung vornimmt, können Sie auch AWS X-Ray-SDK verwenden. Um das SDK zu erhalten, laden Sie das SDK aus seinem GitHub-Repository mit `go get` herunter:

```
go get github.com/aws/aws-xray-sdk-go
```

Um AWS SDK-Clients zu instrumentieren, müssen Sie den Client an die `xray.AWS()`-Methode übergeben. Anschließend können Sie Aufrufe verfolgen, indem Sie die `WithContext`-Version der Methode verwenden.

```
svc := s3.New(session.New())
xray.AWS(svc.Client)
...
svc.ListBucketsWithContext(ctx aws.Context, input *ListBucketsInput)
```

Aktivieren Sie nach Hinzufügen der richtigen Abhängigkeiten die Nachverfolgung in der Konfiguration Ihrer Funktion über die Lambda-Konsole oder die API.

Aktivieren der Nachverfolgung mit der Lambda-Konsole

Gehen Sie folgendermaßen vor, um die aktive Nachverfolgung Ihrer Lambda-Funktion mit der Konsole umzuschalten:

So aktivieren Sie die aktive Nachverfolgung

1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.
2. Wählen Sie eine Funktion aus.
3. Wählen Sie Configuration (Konfiguration) und dann Monitoring and operations tools (Überwachungs- und Produktionstools).
4. Wählen Sie unter Zusätzliche Überwachungstools die Option Bearbeiten aus.
5. Wählen Sie unter CloudWatch Application Signals und AWS X-Ray für Lambda-Service-Nachverfolgungen die Option Aktivieren aus.
6. Wählen Sie Speichern.

Aktivieren der Nachverfolgung mit der Lambda-API

Konfigurieren Sie die Nachverfolgung für Ihre Lambda-Funktion mit der AWS CLI oder dem AWS-SDK. Verwenden Sie die folgenden API-Operationen:

- UpdateFunctionConfiguration
- GetFunctionConfiguration
- CreateFunction

Der folgende AWS CLI-Beispielbefehl aktiviert die aktive Nachverfolgung für eine Funktion mit dem Namen my-function.

```
aws lambda update-function-configuration --function-name my-function \
--tracing-config Mode=Active
```

Der Ablaufverfolgungsmodus ist Teil der versionsspezifischen Konfiguration, wenn Sie eine Version Ihrer Funktion veröffentlichen. Sie können den Ablaufverfolgungsmodus für eine veröffentlichte Version nicht ändern.

Aktivieren von Nachverfolgungen mit CloudFormation

Verwenden Sie die `AWS::Lambda::Function`-Eigenschaft, um die Nachverfolgung für eine CloudFormation-Ressource in einer `TracingConfig`-Vorlage zu aktivieren.

Example function-inline.yml – Ablaufverfolgungskonfiguration

```
Resources:
  function:
    Type: AWS::Lambda::Function
    Properties:
      TracingConfig:
        Mode: Active
      ...
```

Verwenden Sie für eine AWS Serverless Application Model (AWS SAM) `AWS::Serverless::Function`-Ressource die Eigenschaft `Tracing`.

Example template.yml – Ablaufverfolgungskonfiguration

```
Resources:
  function:
    Type: AWS::Serverless::Function
    Properties:
      Tracing: Active
      ...
```

Interpretieren einer X-Ray-Nachverfolgung

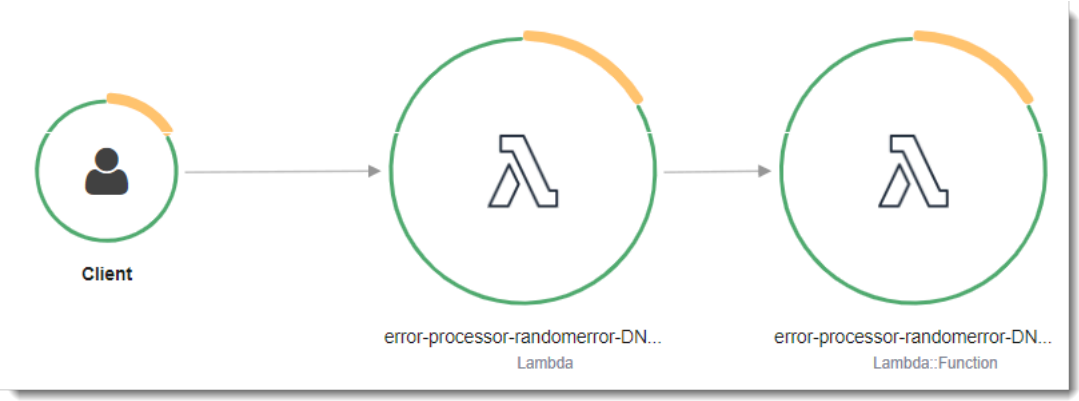
Ihre Funktion benötigt die Berechtigung zum Hochladen von Trace-Daten zu X-Ray. Wenn Sie die aktive Nachverfolgung in der Lambda-Konsole aktivieren, fügt Lambda der Ausführungsrolle Ihrer Funktion die erforderlichen Berechtigungen hinzu. Andernfalls fügen Sie der Ausführungsrolle die AWSXRayDaemonWriteAccess-Richtlinie hinzu.

Nachdem Sie die aktive Nachverfolgung konfiguriert haben, können Sie bestimmte Anfragen über Ihre Anwendung beobachten. Das X-Ray-Service-Diagramm zeigt Informationen über Ihre Anwendung und alle ihre Komponenten an. Das folgende Beispiel zeigt eine Anwendung mit zwei Funktionen. Die primäre Funktion verarbeitet Ereignisse und gibt manchmal Fehler zurück. Die zweite Funktion ganz oben verarbeitet Fehler, die in der Protokollgruppe der ersten Protokollgruppe

erscheinen, und verwendet das AWS-SDK, um X-Ray, Amazon Simple Storage Service (Amazon S3) und Amazon CloudWatch Logs aufzurufen.

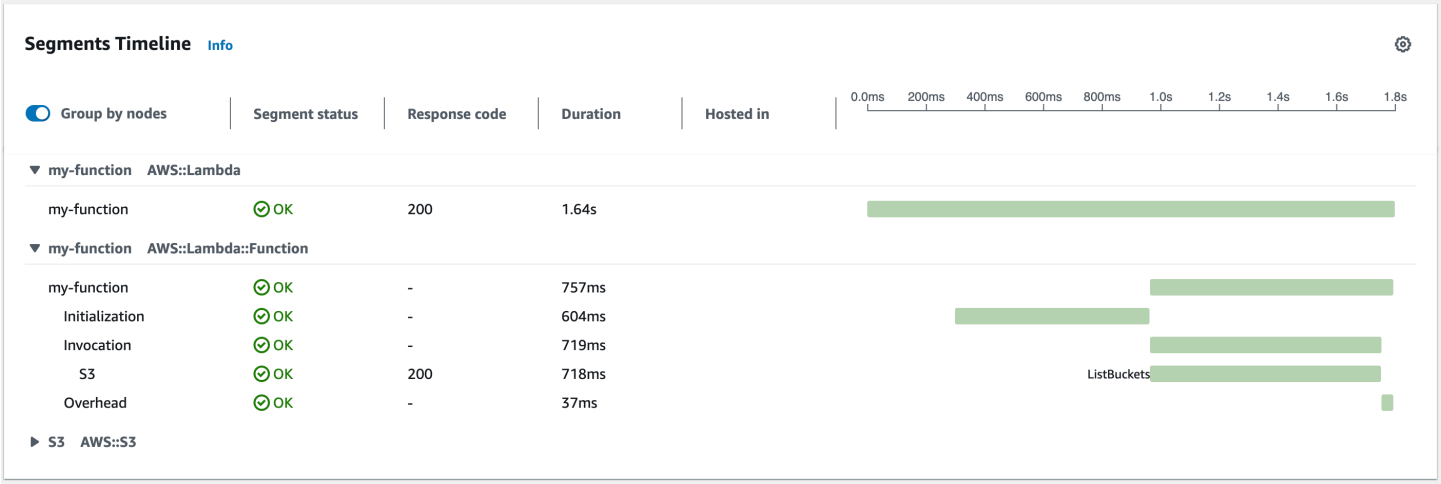
X-Ray verfolgt nicht alle Anfragen an Ihre Anwendung nach. X-Ray wendet einen Sampling-Algorithmus an, um sicherzustellen, dass die Nachverfolgung effizient ist, und stellt dennoch ein repräsentatives Beispiel aller Anfragen bereit. Die Samplingrate beträgt 1 Anforderung pro Sekunde und 5 Prozent aller weiteren Anforderungen. Sie können die X-Ray-Samplingrate nicht für Ihre Funktionen konfigurieren.

In X-Ray, zeichnet eine Ablaufverfolgung Informationen zu einer Anforderung auf, die von einem oder mehreren Services verarbeitet wird. Lambda zeichnet 2 Segmente pro Trace auf, wodurch zwei Knoten im Dienstgraphen entstehen. In der folgenden Abbildung werden diese beiden Knoten hervorgehoben:



Der erste Knoten auf der linken Seite stellt den Lambda-Service dar, der die Aufrufanforderung empfängt. Der zweite Knoten stellt Ihre spezifische Lambda-Funktion dar. Das folgende Beispiel zeigt eine Nachverfolgung mit diesen zwei Segmenten. Beide heißen my-function, aber einer hat einen

Ursprung von `AWS::Lambda` und der andere hat einen Ursprung von `AWS::Lambda::Function`. Wenn das `AWS::Lambda`-Segment einen Fehler anzeigt, hatte der Lambda-Service ein Problem. Wenn das `AWS::Lambda::Function`-Segment einen Fehler anzeigt, ist bei Ihrer Funktion ein Problem aufgetreten.



Dieses Beispiel erweitert das `AWS::Lambda::Function`-Segment, um seine drei Untersegmente anzuzeigen.

> **Note**
>
> AWS implementiert derzeit Änderungen am Lambda-Service. Aufgrund dieser Änderungen können geringfügige Unterschiede in Struktur und Inhalt der Systemprotokollmeldungen und Trace-Segmente auftreten, die von verschiedenen Lambda-Funktionen in Ihrem AWS-Konto. Der hier gezeigte Beispiel-Trace veranschaulicht das Funktionssegment im alten Stil. Die Unterschiede zwischen den Segmenten im alten und im neuen Stil werden in den folgenden Abschnitten beschrieben.
>
> Diese Änderungen werden in den kommenden Wochen implementiert, und alle Funktionen in allen AWS-Regionen-Regionen mit Ausnahme von China und GovCloud werden auf das neue Format der Protokollmeldungen und Trace-Segmente umgestellt.

Das Funktionssegment im alten Stil enthält die folgenden Untersegmente:

- Initialisierung – Stellt die Zeit dar, die für das Laden Ihrer Funktion und das Ausführen des Initialisierungscodes aufgewendet wurde. Dieses Untersegment erscheint nur für das erste Ereignis, das jede Instance Ihrer Funktion verarbeitet.
- Invocation (Aufruf) – Stellt die Zeit dar, die beim Ausführen Ihres Handler-Codes vergeht.

- Overhead (Aufwand) – Stellt die Zeit dar, die von der Lambda-Laufzeitumgebung bei der Verarbeitung des nächsten Ereignisses verbraucht wird.

Das Funktionssegment im neuen Stil enthält kein `Invocation`-Untersegment. Stattdessen werden Kundenuntersegmente direkt an das Funktionssegment angehängt. Weitere Informationen über die Struktur der alten und neuen Funktionssegmente finden Sie unter the section called “Grundlegendes zu X-Ray-Ablaufverfolgungen”.

Sie können auch HTTP-Clients instrumentieren, SQL-Abfragen aufzeichnen und benutzerdefinierte Untersegmente mit Anmerkungen und Metadaten erstellen. Weitere Informationen finden Sie unter AWS X-Ray-SDK für Go im AWS X-Ray-Entwicklerhandbuch.

> **Preisgestaltung**
>
> Sie können X-Ray-Nachverfolgung jeden Monat bis zu einem bestimmten Limit als Teil des AWS kostenlosen Kontingents kostenlos nutzen. Über den Schwellenwert hinaus berechnet X-Ray Gebühren für die Speicherung und den Abruf der Nachverfolgung. Weitere Informationen finden Sie unter AWS X-Ray – Preise.

Erstellen von Lambda-Funktionen mit C#

Sie können Ihre .NET-Anwendung in Lambda mit der verwalteten .NET 8-Laufzeit, einer benutzerdefinierten Laufzeit oder einem Container-Image ausführen. Nachdem Ihr Anwendungscode kompiliert wurde, können Sie ihn entweder als ZIP-Datei oder als Container-Image für Lambda bereitstellen. Lambda stellt die folgenden Laufzeiten für .NET-Sprachen bereit:

Name	ID	Betriebssystem	Datum der Veraltung	Blockfunktion erstellen	Blockfunktion aktualisieren
.NET 10	`dotnet10`	Amazon Linux 2023	14. November 2028	14. Dezember 2028	15. Januar 2029
.NET 9 (nur Container)	`dotnet9`	Amazon Linux 2023	10. November 2026	Nicht geplant	Nicht geplant
.NET 8	`dotnet8`	Amazon Linux 2023	10. November 2026	10. Dezember 2026	11. Januar 2027

Einrichten der .NET-Entwicklungsumgebung

Um Ihre Lambda-Funktionen zu entwickeln und zu erstellen, können Sie jede der allgemein verfügbaren.NET-integrierten Entwicklungsumgebungen (IDEs) verwenden, einschließlich Microsoft Visual Studio, Visual Studio Code und JetBrains Rider. Zur Vereinfachung Ihrer Entwicklungserfahrung AWS bietet es eine Reihe von .NET-Projektvorlagen sowie die `Amazon.Lambda.Tools` Befehlszeilenschnittstelle (CLI).

Führen Sie die folgenden .NET-CLI-Befehle aus, um diese Projektvorlagen und Befehlszeilentools zu installieren.

Installieren der .NET-Projektvorlagen

Zur Installation der Projektvorlagen führen Sie den folgenden Befehl aus:

```
dotnet new install Amazon.Lambda.Templates
```

Installation und Aktualisierung der CLI-Tools

Führen Sie die folgenden Befehle aus, um die `Amazon.Lambda.Tools`-CLI zu installieren, zu aktualisieren und zu deinstallieren.

Installieren der Befehlszeilen-Tools:

```
dotnet tool install -g Amazon.Lambda.Tools
```

Aktualisieren der Befehlszeilen-Tools:

```
dotnet tool update -g Amazon.Lambda.Tools
```

Deinstallieren der Befehlszeilen-Tools:

```
dotnet tool uninstall -g Amazon.Lambda.Tools
```

Definieren des Lambda-Funktions-Handlers in C#

Der Lambda-Funktionshandler ist die Methode in Ihrem Funktionscode, die Ereignisse verarbeitet. Wenn Ihre Funktion aufgerufen wird, führt Lambda die Handler-Methode aus. Ihre Funktion wird so lange ausgeführt, bis der Handler eine Antwort zurückgibt, beendet wird oder ein Timeout auftritt.

Auf dieser Seite wird beschrieben, wie Sie Lambda-Funktionshandler in C# verwenden, um mit der verwalteten .NET-Laufzeit zu arbeiten, einschließlich der Optionen für Projekteinrichtung, Benennungskonventionen und Best Practices. Diese Seite enthält auch ein Beispiel für eine C#-Lambda-Funktion, die Informationen über eine Bestellung aufnimmt, eine Textdatei als Beleg erstellt und diese Datei in einen Bucket in Amazon Simple Storage Service (S3) stellt. Informationen darüber, wie Sie Ihre Funktion nach dem Schreiben einsetzen können, finden Sie unter the section called “Bereitstellungspaket” oder the section called “Bereitstellen von Container-Images”.

Themen

- Einrichten Ihres C#-Handler-Projekts
- Beispiel für den C#-Lambda-Funktionscode
- Handler für Klassenbibliotheken
- Ausführbare Assembly-Handler
- Gültige Handler-Signaturen für C#-Funktionen
- Namenskonventionen für Handler
- Serialisierung in C#-Lambda-Funktionen
- Dateibasierte Funktionen
- Zugreifen auf und Verwenden des Lambda-Kontextobjekts
- Verwenden Sie die SDK für .NET Version 3 in Ihrem Handler
- Zugriff auf Umgebungsvariablen
- Verwenden des globalen Zustands
- Vereinfachen Sie den Funktionscode mit dem Lambda Annotations Framework
- Bewährte Codemethoden für C#-Lambda-Funktionen

Einrichten Ihres C#-Handler-Projekts

Wenn Sie mit Lambda-Funktionen in C# arbeiten, umfasst der Prozess das Schreiben Ihres Codes und das anschließende Bereitstellen Ihres Codes in Lambda. Es gibt zwei

verschiedene Ausführungsmodelle für die Bereitstellung von Lambda-Funktionen in .NET: den Klassenbibliotheksansatz und den Ansatz für ausführbare Assemblys.

Beim Klassenbibliotheksansatz packen Sie Ihren Funktionscode als .NET-Assembly (`.dll`) und stellen ihn mit der verwalteten .NET-Laufzeit (`dotnet8`) in Lambda bereit. Für den Namen des Handlers erwartet Lambda eine Zeichenfolge im Format `AssemblyName::Namespace.Classname::Methodname`. Während der Initialisierungsphase der Funktion wird die Klasse Ihrer Funktion initialisiert und der gesamte Code im Konstruktor wird ausgeführt.

Beim Ansatz für ausführbare Assemblys verwenden Sie das Feature für Anweisungen der obersten Ebene, die erstmals in C# 9 eingeführt wurde. Dieser Ansatz generiert eine ausführbare Assembly, die Lambda immer dann ausführt, wenn sie einen Aufrufbefehl für Ihre Funktion empfängt. Bei diesem Ansatz verwenden Sie außerdem die verwaltete .NET-Laufzeit (`dotnet8`). Für den Namen des Handlers geben Sie in Lambda den Namen der ausführbaren Assembly an, die ausgeführt werden soll.

Das Hauptbeispiel auf dieser Seite veranschaulicht den Klassenbibliotheksansatz. Sie können Ihr C#-Lambda-Projekt auf verschiedene Arten initialisieren. Am einfachsten ist es jedoch, die .NET-CLI mit der `Amazon.Lambda.Tools`-CLI zu verwenden. Richten Sie die `Amazon.Lambda.Tools`-CLI ein, indem Sie die Schritte unter the section called “Entwicklungsumgebung” befolgen. Initialisieren Sie das Projekt dann mit dem folgenden Befehl:

```
dotnet new lambda.EmptyFunction --name ExampleCS
```

Dieser Befehl generiert die folgende Dateistruktur:

```
/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
```

In dieser Dateistruktur befindet sich die Hauptlogik des Handlers für Ihre Funktion in der Datei `Function.cs`.

Beispiel für den C#-Lambda-Funktionscode

Das folgende Beispiel für einen C#-Lambda-Funktionscode nimmt Informationen über eine Bestellung auf, erstellt eine Textdatei als Beleg und platziert diese Datei in einem Amazon-S3-Bucket.

Example **`Function.cs`**-Lambda-Funktion

```
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);
        }
    }
}
```

Diese `Function.cs`-Datei enthält die folgenden Abschnitte des Codes:

- `using`-Anweisungen: Damit können Sie C#-Klassen importieren, die für Ihre Lambda-Funktion erforderlich sind.
- `[assembly: LambdaSerializer(...)]`: `LambdaSerializer` ist ein Assembly-Attribut, das Lambda anweist, JSON-Ereignisnutzdaten automatisch in C#-Objekte zu konvertieren, bevor sie an Ihre Funktion übergeben werden.
- `namespace ExampleLambda`: Dies definiert den Namespace. In C# muss der Namespace-Name nicht mit dem Dateinamen übereinstimmen.
- `public class Order {...}`: Hiermit wird die Form des erwarteten Eingabeereignisses definiert.
- `public class OrderHandler {...}`: Hiermit wird die C#-Klasse definiert. Darin definieren Sie die Haupt-Handler-Methode und alle anderen Hilfsmethoden.
- `private static readonly AmazonS3Client s3Client = new();`: Dadurch wird ein Amazon-S3-Client mit der standardmäßigen Anbieterkette von Anmeldeinformationen außerhalb der Haupt-Handler-Methode initialisiert. Dies veranlasst Lambda, diesen Code während der Initialisierungsphase auszuführen.
- `public async ... HandleRequest (Order order, ILambdaContext context)`: Dies ist die Haupthandler-Methode, die Ihre Hauptanwendungslogik enthält.
- `private async Task UploadReceiptToS3(...) {}`: Dies ist eine Hilfsmethode, auf die von der `handleRequest`-Haupthandler-Methode verwiesen wird.

Da für diese Funktion ein SDK-Client für Amazon S3 erforderlich ist, müssen Sie sie zu den Abhängigkeiten Ihres Projekts hinzufügen. Dazu können Sie zu `src/ExampleCS` navigieren und den folgenden Befehl ausführen:

```
dotnet add package AWSSDK.S3
```

Fügen Sie Metadateninformationen zu aws-lambda-tools-defaults .json hinzu

Standardmäßig enthält die generierte Datei `aws-lambda-tools-defaults.json` keine `profile`- oder `region`-Informationen für Ihre Funktion. Aktualisieren Sie außerdem die `function-handler`-Zeichenfolge auf den richtigen Wert (`ExampleCS::ExampleLambda.OrderHandler::HandleRequest`). Sie können diese Aktualisierung manuell vornehmen und die erforderlichen Metadaten hinzufügen, um ein bestimmtes Anmeldeinformationsprofil und eine bestimmte Region für Ihre Funktion zu verwenden. Die `aws-lambda-tools-defaults.json`-Datei sollte etwa wie folgt aussehen:

```
{
  "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"
}
```

Damit diese Funktion ordnungsgemäß funktioniert, muss ihre Ausführungsrolle die `s3:PutObject`-Aktion zulassen. Stellen Sie außerdem sicher, dass Sie die `RECEIPT_BUCKET`-Umgebungsvariable definieren. Nach einem erfolgreichen Aufruf sollte der Amazon-S3-Bucket eine Empfangsdatei enthalten.

Handler für Klassenbibliotheken

Der Hauptbeispielcode auf dieser Seite veranschaulicht den Klassenbibliotheks-Handler. Klassenbibliotheks-Handler haben die folgende Struktur:

```
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSerializer

namespace NAMESPACE;

...

public class CLASSNAME {
    public async Task<string> METHODNAME (...) {
    ...
    }
```

```
}
```

Wenn Sie eine Lambda-Funktion erstellen, müssen Sie Lambda Informationen über den Handler Ihrer Funktion in Form einer Zeichenfolge im Feld Handler zur Verfügung stellen. Dadurch wird Lambda mitgeteilt, welche Methode in Ihrem Code ausgeführt werden soll, wenn Ihre Funktion aufgerufen wird. In C# lautet das Format der Handler-Zeichenfolge für Klassenbibliotheks-Handler `ASSEMBLY::TYPE::METHOD`. Dabei gilt Folgendes:

- `ASSEMBLY` ist der Name der .NET-Assembly-Datei für Ihre Anwendung. Wenn Sie die `Amazon.Lambda.Tools`-CLI zur Erstellung Ihrer Anwendung verwenden und den Assembly-Namen nicht über die Eigenschaft `AssemblyName` in der `.csproj`-Datei festlegen, ist `ASSEMBLY` einfach der Name Ihrer `.csproj`-Datei.
- `TYPE` ist der vollständige Name des Handler-Typs, der `NAMESPACE.CLASSNAME` lautet.
- `METHOD` ist der Name der Haupt-Handler-Methode in Ihrem Code. Er lautet `METHODNAME`.

Wenn die Assembly im Hauptbeispielcode auf dieser Seite den Namen `ExampleCS` trägt, lautet die vollständige Zeichenfolge für den Handler `ExampleCS::ExampleLambda.OrderHandler::HandleRequest`.

Ausführbare Assembly-Handler

Sie können Lambda-Funktionen in C# auch als ausführbare Assembly definieren. Handler für ausführbare Assemblys nutzen das C#-Feature für Anweisungen der obersten Ebene, bei der der Compiler die `Main()`-Methode generiert und Ihren Funktionscode darin platziert. Bei der Verwendung ausführbarer Assemblys muss die Lambda-Laufzeit gebootet werden. Verwenden Sie dazu die `LambdaBootstrapBuilder.Create`-Methode in Ihrem Code. Die Eingaben für diese Methode stellen die Haupt-Handler-Funktion sowie den Lambda-Serialisierer dar, die verwendet werden sollen. Im Anschluss sehen Sie ein Beispiel für einen Handler für ausführbare Assemblys in 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)
    };
};
```

Im Feld Handler für Handler für ausführbare Assemblys ist die Handler-Zeichenfolge, die Lambda mitteilt, wie Ihr Code ausgeführt werden soll, der Name der Assembly. In diesem Beispiel ist dies `GetProductHandler`.

Gültige Handler-Signaturen für C#-Funktionen

In C# können gültige Lambda-Handler-Signaturen zwischen 0 und 2 Argumente enthalten. Normalerweise hat Ihre Handler-Signatur zwei Argumente, wie im Hauptbeispiel gezeigt:

```
public async Task<string> HandleRequest(Order order, ILambdaContext context)
```

Wenn Sie zwei Argumente angeben, muss das erste Argument die Ereigniseingabe und das zweite Argument das Lambda-Kontextobjekt sein. Beide Argumente sind optional. Die folgenden sind beispielsweise ebenfalls gültige Lambda-Handler-Signaturen in C#:

- `public async Task<string> HandleRequest()`
- `public async Task<string> HandleRequest(Order order)`
- `public async Task<string> HandleRequest(ILambdaContext context)`

Neben der Basissyntax der Handler-Signatur gibt es einige zusätzliche Einschränkungen:

- Sie können das Schlüsselwort `unsafe` nicht in der Handler-Signatur verwenden. Sie können jedoch den `unsafe`-Kontext innerhalb der Handler-Methode und ihrer Abhängigkeiten verwenden. Weitere Informationen finden Sie unter unsafe (C#-Referenz) auf der Microsoft-Dokumentationswebsite.

- Der Handler darf das Schlüsselwort `params` nicht verwenden oder `ArgIterator` als Eingabe- oder Rückgabeparameter verwenden. Diese Schlüsselwörter unterstützen eine variable Anzahl von Parametern. Ihr Handler kann maximal zwei Argumente akzeptieren.
- Der Handler ist möglicherweise keine generische Methode. Mit anderen Worten: Er kann keine generischen Typparameter wie <T> verwenden.
- Lambda unterstützt keine asynchronen Handler mit `async void` in der Signatur.

Namenskonventionen für Handler

Lambda-Handler in C# haben keine strengen Benennungsbeschränkungen. Sie müssen jedoch sicherstellen, dass Sie Lambda bei der Bereitstellung Ihrer Funktion die richtige Handler-Zeichenfolge zur Verfügung stellen. Die richtige Handler-Zeichenfolge hängt davon ab, ob Sie einen Klassenbibliotheks-Handler oder einen Handler für ausführbare Assemblys bereitstellen.

Sie können zwar einen beliebigen Namen für Ihren Handler verwenden, Funktionsnamen in C# sind jedoch generell in. PascalCase Auch wenn der Dateiname nicht mit dem Klassennamen oder dem Handler-Namen übereinstimmen muss, ist es im Allgemeinen eine bewährte Methode, einen Dateinamen wie `OrderHandler.cs` zu verwenden, wenn der Klassenname `OrderHandler` lautet. Sie können den Dateinamen in diesem Beispiel beispielsweise von `Function.cs` in `OrderHandler.cs` ändern.

Serialisierung in C#-Lambda-Funktionen

JSON ist das gebräuchlichste und standardmäßigste Eingabeformat für Lambda-Funktionen. In diesem Beispiel erwartet die Funktion eine Eingabe ähnlich der folgenden:

```
{
    "orderId": "12345",
    "amount": 199.99,
    "item": "Wireless Headphones"
}
```

In C# können Sie die Form des erwarteten Eingabeereignisses in einer Klasse definieren. In diesem Beispiel definieren wir die `Order`-Klasse, um diese Eingabe zu modellieren:

```
public class Order
{
```

```
    public string OrderId { get; set; } = string.Empty;
    public double Amount { get; set; }
    public string Item { get; set; } = string.Empty;
}
```

Wenn Ihre Lambda-Funktion andere Eingabe- oder Ausgabetypen als ein `Stream`-Objekt verwendet, müssen Sie eine Serialisierungsbibliothek zu Ihrer Anwendung hinzufügen. Auf diese Weise können Sie die JSON-Eingabe in eine Instanz der von Ihnen definierten Klasse konvertieren. Es gibt zwei Methoden der Serialisierung für C#-Funktionen in Lambda: reflexionsbasierte Serialisierung und quellgenerierte Serialisierung.

Reflexionsbasierte Serialisierung

AWS stellt vorgefertigte Bibliotheken bereit, die Sie schnell zu Ihrer Anwendung hinzufügen können. Diese Bibliotheken implementieren Serialisierung mithilfe von Reflexion. Verwenden Sie eins der folgenden Pakete, um die reflexionsbasierte Serialisierung zu implementieren:

- `Amazon.Lambda.Serialization.SystemTextJson`: Im Backend nutzt dieses Paket `System.Text.Json` zur Ausführung von Serialisierungsaufgaben.
- `Amazon.Lambda.Serialization.Json`: Im Backend nutzt dieses Paket `Newtonsoft.Json` zur Ausführung von Serialisierungsaufgaben.

Sie können auch Ihre eigene Serialisierungsbibliothek erstellen, indem Sie die `ILambdaSerializer`-Schnittstelle implementieren; diese ist als Teil der `Amazon.Lambda.Core`-Bibliothek verfügbar. Diese Schnittstelle definiert zwei Methoden:

- `T Deserialize<T>(Stream requestStream);`

  Sie implementieren diese Methode für die Deserialisierung der Anfragenutzlast von der `Invoke`-API in das Objekt, das Ihrem Lambda-Funktions-Handler übergeben wird.
- `T Serialize<T>(T response, Stream responseStream);`

  Sie implementieren diese Methode, um das von Ihrem Lambda-Funktions-Handler zurückgegebene Ergebnis in die Antwort-Nutzlast zu serialisieren, die der `Invoke`-API-Vorgang zurückgibt.

Das Hauptbeispiel auf dieser Seite verwendet reflexionsbasierte Serialisierung. Die auf Reflection basierende Serialisierung ist sofort einsatzbereit AWS Lambda und erfordert keine zusätzliche

Einrichtung, weshalb sie aus Gründen der Einfachheit eine gute Wahl ist. Es ist jedoch mehr Funktionsspeicher erforderlich. Aufgrund von Laufzeitreflexionen kann es auch zu höheren Funktionslatenzen kommen.

Quellgenerierte Serialisierung

Bei der quellgenerierten Serialisierung wird der Serialisierungscode zur Kompilierzeit generiert. Dies macht Reflexion überflüssig und kann die Leistung Ihrer Funktion verbessern. Gehen Sie wie folgt vor, um die quellgenerierte Serialisierung in Ihrer Funktion zu verwenden:

- Erstellen Sie eine neue Teilklasse, die von `JsonSerializerContext` erbt, und fügen Sie `JsonSerializable`-Attribute für alle Typen hinzu, die serialisiert oder deserialisiert werden müssen.
- Konfigurieren Sie das `LambdaSerializer` so, dass es ein `SourceGeneratorLambdaJsonSerializer<T>` verwendet.
- Aktualisieren Sie alle manuellen Serialisierungen und Deserialisierungen in Ihrem Anwendungscode, um die neu erstellte Klasse zu verwenden.

Das folgende Beispiel zeigt, wie Sie das Hauptbeispiel auf dieser Seite, das reflexionsbasierte Serialisierung verwendet, so ändern können, dass stattdessen die quellgenerierte Serialisierung verwendet wird.

```
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);

    ...

    }

}
```

Für die quellgenerierte Serialisierung sind mehr Einrichtungsschritte erforderlich als für die reflexionsbasierte Serialisierung. Funktionen, die quellgenerierte Serialisierung verwenden, verbrauchen jedoch tendenziell weniger Speicher und weisen aufgrund der Codegenerierung während der Kompilierung eine bessere Leistung auf. Um Kaltstarts von Funktionen zu vermeiden, sollten Sie in Erwägung ziehen, zur quellgenerierten Serialisierung zu wechseln.

> **Note**
>
> Wenn Sie die native ahead-of-time Kompilierung (AOT) mit Lambda verwenden möchten, müssen Sie die quellengenerierte Serialisierung verwenden.

Dateibasierte Funktionen

Mit dateibasierten Apps, die in .NET 10 eingeführt wurden, können Sie.NET-Anwendungen aus einer einzigen `.cs` Datei ohne Datei- oder Verzeichnisstruktur `.csproj` erstellen. Lambda unterstützt dateibasierte Funktionen, beginnend mit.NET 10. Sie bieten eine optimierte, einfache Möglichkeit, Lambda-Funktionen in C# zu erstellen.

Der schnellste Weg, um mit der Erstellung einer auf C#-Dateien basierenden Lambda-Funktion zu beginnen, ist die Verwendung des Pakets. `Amazon.Lambda.Templates` Zur Installation dieses Pakets führen Sie den folgenden Befehl aus:

```
dotnet new install Amazon.Lambda.Templates
```

Erstellen Sie als Nächstes eine auf einer C#-Datei basierende Lambda-Beispielfunktion:

```
dotnet new lambda.FileBased -n MyLambdaFunction
```

Dateibasierte Funktionen verwenden ausführbare Assembly-Handler. Sie müssen daher das `Amazon.Lambda.RuntimeSupport` NuGet Paket einschließen und die `LambdaBootstrapBuilder.Create` Methode verwenden, um die .NET-Handlerfunktion für den Ereignistyp zu registrieren und den.NET Lambda-Runtime-Client zu starten.

Dateibasierte Funktionen verwenden standardmäßig .NET Native AOT, was eine quellengenerierte Serialisierung erfordert. Sie können Native AOT deaktivieren, indem Sie es in Ihrer Quelldatei angeben. `#:property PublishAot=false` Weitere Informationen zur Verwendung von Native AOT in Lambda finden Sie unter. the section called “Native AOT-Kompilierung”

Zugreifen auf und Verwenden des Lambda-Kontextobjekts

Das Lambda-Kontextobjekt enthält Informationen über Aufruf, Funktion und Ausführungsumgebung. In diesem Beispiel ist das Kontextobjekt vom Typ `Amazon.Lambda.Core.ILambdaContext` und das zweite Argument der Haupthandlerfunktion.

```
public async Task<string> HandleRequest(Order order, ILambdaContext context) {
    ...
}
```

Das Kontextobjekt ist eine optionale Eingabe. Weitere Informationen zu gültigen akzeptierten Handlersignaturen finden Sie unter the section called “Gültige Handler-Signaturen für C#-Funktionen”.

Das Kontextobjekt ist nützlich, um Funktionsprotokolle für Amazon zu erstellen CloudWatch. Sie können die `context.getLogger()`-Methode verwenden, um ein `LambdaLogger`-Objekt für die Protokollierung abzurufen. In diesem Beispiel können wir den Logger verwenden, um eine Fehlermeldung zu protokollieren, falls bei der Verarbeitung aus irgendeinem Grund ein Fehler auftritt:

```
context.Logger.LogError($"Failed to process order: {ex.Message}");
```

Außerhalb der Protokollierung können Sie das Kontextobjekt auch für die Funktionsüberwachung verwenden. Weitere Informationen über das Kontextobjekt finden Sie unter the section called “Context”.

Verwenden Sie die SDK für .NET Version 3 in Ihrem Handler

Oft verwenden Sie Lambda-Funktionen, um mit anderen AWS Ressourcen zu interagieren oder diese zu aktualisieren. Die einfachste Art, eine Schnittstelle zu diesen Ressourcen herzustellen, ist die Verwendung von SDK für .NET v3.

> **Note**
>
> Die SDK für .NET (v2) ist veraltet. Wir empfehlen, nur Version 3 zu verwenden. SDK für .NET

Mit dem folgenden `Amazon.Lambda.Tools`-Befehl können Sie Ihrem Projekt SDK-Abhängigkeiten hinzufügen:

```
dotnet add package <package_name>
```

Im Hauptbeispiel auf dieser Seite müssen wir beispielsweise die Amazon-S3-API verwenden, um einen Beleg in S3 hochzuladen. Wir können den SDK-Client für Amazon S3 mit dem folgenden Befehl importieren:

```
dotnet add package AWSSDK.S3
```

Dieser Befehl fügt die Abhängigkeit zu Ihrem Projekt hinzu. In der `.csproj`-Datei Ihres Projekts sollte außerdem eine Zeile ähnlich der folgenden angezeigt werden:

```
<PackageReference Include="AWSSDK.S3" Version="3.7.2.18" />
```

Importieren Sie dann die Abhängigkeiten direkt in Ihren C#-Code:

```
using Amazon.S3;
using Amazon.S3.Model;
```

Der Beispielcode initialisiert dann wie folgt einen Amazon-S3-Client (unter Verwendung der standardmäßigen Anbieterkette für Anmeldeinformationen):

```
private static readonly AmazonS3Client s3Client = new();
```

In diesem Beispiel haben wir unseren Amazon-S3-Client außerhalb der Haupt-Handler-Funktion initialisiert, um zu vermeiden, dass wir ihn bei jedem Aufruf unserer Funktion initialisieren müssen. Nachdem Sie Ihren SDK-Client initialisiert haben, können Sie ihn für die Interaktion mit anderen AWS Diensten verwenden. Der Beispielcode ruft die Amazon S3 `PutObject`-API wie folgt auf:

```
var putRequest = new PutObjectRequest
{
    BucketName = bucketName,
    Key = key,
    ContentBody = receiptContent,
    ContentType = "text/plain"
};

await s3Client.PutObjectAsync(putRequest);
```

Zugriff auf Umgebungsvariablen

In Ihrem Handler-Code können Sie mithilfe der `System.Environment.GetEnvironmentVariable`-Methode auf beliebige Umgebungsvariablen verweisen. In diesem Beispiel verweisen wir mit den folgenden Codezeilen auf die definierte `RECEIPT_BUCKET`-Umgebungsvariable:

```
string? bucketName = Environment.GetEnvironmentVariable("RECEIPT_BUCKET");
if (string.IsNullOrWhiteSpace(bucketName))
{
    throw new ArgumentException("RECEIPT_BUCKET environment variable is not set");
}
```

Verwenden des globalen Zustands

Lambda führt Ihren statischen Code und den Klassenkonstruktor während der Initialisierungsphase aus, bevor Ihre Funktion zum ersten Mal aufgerufen wird. Ressourcen, die während der Initialisierung erstellt werden, bleiben zwischen Aufrufen im Speicher, sodass Sie sie nicht bei jedem Aufruf Ihrer Funktion neu erstellen müssen.

Im Beispielcode befindet sich der Initialisierungscode des S3-Clients außerhalb der Haupt-Handler-Methode. Die Laufzeit initialisiert den Client, bevor die Funktion ihr erstes Ereignis verarbeitet, was zu längeren Verarbeitungszeiten führen kann. Nachfolgende Ereignisse sind viel schneller, da Lambda den Client nicht erneut initialisieren muss.

Vereinfachen Sie den Funktionscode mit dem Lambda Annotations Framework

Lambda Annotations ist ein Framework für .NET 8, das das Schreiben von Lambda-Funktionen mit C# vereinfacht. Das Annotations-Framework verwendet Quellgeneratoren, um Code zu generieren, der vom Lambda-Programmiermodell in den vereinfachten Code übersetzt wird. Mit dem Annotations-Framework können Sie einen Großteil des Codes in einer Lambda-Funktion ersetzen, die mit dem regulären Programmiermodell geschrieben wurde. Code, der mit dem Framework geschrieben wurde, verwendet einfachere Ausdrücke, sodass Sie sich auf Ihre Geschäftslogik konzentrieren können. Beispiele finden Sie unter Amazon.Lambda.Annotations in der NuGet-Dokumentation.

Ein Beispiel für eine vollständige Anwendung, die Lambda-Annotationen verwendet, finden Sie im PhotoAssetManagerBeispiel im `awsdocs/aws-doc-sdk-examples` GitHub Repository. Die Hauptdatei `Function.cs` im Verzeichnis `PamApiAnnotations` verwendet Lambda Annotations. Zum Vergleich: Das Verzeichnis `PamApi` enthält äquivalente Dateien, die mit dem regulären Lambda-Programmiermodell geschrieben wurden.

Abhängigkeitsinjektion mit dem Lambda Annotations-Framework

Sie können auch das Lambda Annotations-Framework verwenden, um Ihren Lambda-Funktionen eine Dependency Injection hinzuzufügen, indem Sie die Syntax verwenden, mit der Sie vertraut sind. Wenn Sie einer `[LambdaStartup]`-Datei ein `Startup.cs`-Attribut hinzufügen, generiert das Lambda Annotations-Framework den erforderlichen Code zur Kompilierzeit.

```
[LambdaStartup]
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddSingleton<IDatabaseRepository, DatabaseRepository>();
    }
}
```

Ihre Lambda-Funktion kann Dienste entweder durch Konstruktorinjektion oder durch Injektion in einzelne Methoden unter Verwendung des `[FromServices]`-Attributs injizieren.

```
[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);
    }
}
```

Bewährte Codemethoden für C#-Lambda-Funktionen

Halten Sie sich an die Richtlinien in der folgenden Liste, um beim Erstellen Ihrer Lambda-Funktionen die besten Codierungspraktiken anzuwenden:

- Trennen Sie den Lambda-Handler von Ihrer Core-Logik. Auf diese Weise können Sie eine Funktion zur besseren Prüfbarkeit von Einheiten schaffen.
- Kontrollieren Sie die Abhängigkeiten im Bereitstellungspaket Ihrer Funktion. Die AWS Lambda -Ausführungsumgebung enthält eine Reihe von Bibliotheken. Um die neuesten Funktionen und Sicherheitsupdates zu aktivieren, wird Lambda diese Bibliotheken regelmäßig aktualisieren. Diese Updates können das Verhalten Ihrer Lambda-Funktion geringfügig verändern. Um die Abhängigkeiten, die Ihre Funktion verwendet, vollständig zu kontrollieren, empfehlen wir, alle Abhängigkeiten mit Ihrem Bereitstellungspaket zu bündeln.
- Minimieren Sie die Komplexität Ihrer Abhängigkeiten. Ziehen Sie einfachere Frameworks vor, die sich schnell beim Start der Ausführungsumgebung laden lassen.
- Minimieren Sie die Größe Ihres Bereitstellungspakets auf die für die Laufzeit erforderliche Größe. Dadurch verkürzt sich die Zeit, die für das Herunterladen und Entpacken Ihres Bereitstellungspakets vor dem Aufruf benötigt wird. Vermeiden Sie es bei in .NET verfassten Funktionen, die gesamte AWS SDK-Bibliothek als Teil Ihres Bereitstellungspakets hochzuladen. Stattdessen sollten Sie selektiv von den Modulen ausgehen, die Komponenten des SDK

aufnehmen, die Sie benötigen (z. B. DynamoDB, Amazon-S3-SDK-Module und Lambda-Kernbibliotheken).

Nutzen Sie die Wiederverwendung der Ausführungsumgebung zur Verbesserung Ihrer Funktion. Initialisieren Sie SDK-Clients und Datenbankverbindungen außerhalb des Funktions-Handlers und speichern Sie statische Komponenten lokal im `/tmp`-Verzeichnis. Nachfolgende Aufrufe, die von derselben Instance Ihrer Funktion verarbeitet werden, können diese Ressourcen wiederverwenden. Dies spart Kosten durch Reduzierung der Funktionslaufzeit.

Um potenzielle Datenlecks über Aufrufe hinweg zu vermeiden, verwenden Sie die Ausführungsumgebung nicht, um Benutzerdaten, Ereignisse oder andere Informationen mit Sicherheitsauswirkungen zu speichern. Wenn Ihre Funktion auf einem veränderbaren Zustand beruht, der nicht im Speicher innerhalb des Handlers gespeichert werden kann, sollten Sie für jeden Benutzer eine separate Funktion oder separate Versionen einer Funktion erstellen.

Verwenden Sie eine Keep-Alive-Direktive, um dauerhafte Verbindungen zu pflegen. Lambda bereinigt Leerlaufverbindungen im Laufe der Zeit. Der Versuch, eine Leerlaufverbindung beim Aufruf einer Funktion wiederzuverwenden, führt zu einem Verbindungsfehler. Um Ihre persistente Verbindung aufrechtzuerhalten, verwenden Sie die Keep-Alive-Direktive, die Ihrer Laufzeit zugeordnet ist. Ein Beispiel finden Sie unter Wiederverwenden von Verbindungen mit Keep-Alive in Node.js.

Verwenden Sie Umgebungsvariablen um Betriebsparameter an Ihre Funktion zu übergeben. Wenn Sie z. B. Daten in einen Amazon-S3-Bucket schreiben, anstatt den Bucket-Namen, in den Sie schreiben, hartzucodieren, konfigurieren Sie den Bucket-Namen als Umgebungsvariable.

Vermeiden Sie rekursive Aufrufe in Ihrer Lambda-Funktion, bei denen die Funktion sich selbst aufruft oder einen Prozess initiiert, der die Funktion erneut aufrufen kann. Dies kann zu unvorhergesehenen Mengen an Funktionsaufrufen führen und höhere Kosten zur Folge haben. Wenn Sie eine unbeabsichtigte Menge von Aufrufen feststellen, legen Sie die reservierte gleichzeitige Ausführung der Funktion auf `0` fest, um sofort alle Aufrufe der Funktion zu drosseln, während Sie den Code aktualisieren.

Verwenden Sie APIs in Ihrem Lambda-Funktionscode nicht undokumentiert, nicht öffentlich. Für AWS Lambda verwaltete Laufzeiten führt Lambda regelmäßig Sicherheits- und Funktionsupdates für interne Lambda-Laufzeiten durch. APIs Diese internen API-Updates können abwärtsinkompatibel sein, was zu unbeabsichtigten Folgen wie Aufruffehlern führen kann, wenn Ihre Funktion von diesen nicht öffentlichen Daten abhängig ist. APIs Eine Liste der öffentlich verfügbaren Programme finden Sie in der API-Referenz. APIs

Schreiben Sie idempotenten Code. Das Schreiben idempotenter Code für Ihre Funktionen stellt sicher, dass doppelte Ereignisse auf die gleiche Weise behandelt werden. Ihr Code sollte Ereignisse ordnungsgemäß validieren und doppelte Ereignisse ordnungsgemäß behandeln. Weitere Informationen finden Sie unter Wie mache ich meine Lambda-Funktion idempotent?.

Erstellen und Bereitstellen von C#-Lambda-Funktionen mit ZIP-Dateiarchiven

Ein .NET-Bereitstellungspaket (ZIP-Dateiarchiv) enthält die kompilierten Assembly Ihrer Funktion sowie aller ihrer Assembly-Abhängigkeiten. Das Paket enthält auch eine *proj*`.deps.json`-Datei. Dies signalisiert der .NET-Laufzeit alle Abhängigkeiten Ihrer Funktion sowie eine *proj*`.runtimeconfig.json`-Datei, die für die Konfiguration der Laufzeit verwendet wird.

Um einzelne Lambda-Funktionen bereitzustellen, können Sie die `Amazon.Lambda.Tools` .NET Lambda Global CLI verwenden. Mit dem `dotnet lambda deploy-function` Befehl wird automatisch ein ZIP-Bereitstellungspaket erstellt und auf Lambda bereitgestellt. Wir empfehlen jedoch, Frameworks wie AWS Serverless Application Model (AWS SAM) oder AWS Cloud Development Kit (AWS CDK) zu verwenden, um Ihre .NET-Anwendungen auf AWS bereitzustellen.

Serverless-Anwendungen bestehen in der Regel aus einer Kombination von Lambda-Funktionen und anderen verwalteten AWS-Services, die zusammenarbeiten, um eine bestimmte Geschäftsaufgabe zu erfüllen. AWS SAM und AWS CDK vereinfachen die Erstellung und Bereitstellung von Lambda-Funktionen mit anderen AWS-Services in großem Umfang. Die AWS SAM-Vorlagenspezifikation bietet eine einfache und saubere Syntax zur Beschreibung der Lambda-Funktionen, APIs, Berechtigungen, Konfigurationen und andere AWS-Ressourcen, aus denen Ihre Serverless-Anwendung besteht. Über das AWS CDK definieren Sie die Cloud-Infrastruktur als Code, mit dem Sie zuverlässige, skalierbare und kostengünstige Anwendungen in der Cloud mit modernen Programmiersprachen und Frameworks wie .NET erstellen können. Sowohl die AWS CDK als auch die AWS SAM verwenden das.NET Lambda Global CLI, um Ihre Funktionen zu verpacken.

Obwohl es möglich ist, Lambda-Ebenen mit Funktionen in C# mithilfe der .NET-Core-CLI zu verwenden, raten wir davon ab. Funktionen in C#, die Ebenen verwenden, laden die freigegebenen Baugruppen während des Init-Phase manuell in den Speicher, was die Kaltstartzeiten verlängern kann. Fügen Sie stattdessen den gesamten gemeinsam genutzten Code zur Kompilierungszeit ein, um die Auswirkungen auf die Leistung durch das Laden von Assemblys zur Laufzeit zu vermeiden.

In den folgenden Abschnitten finden Sie Anweisungen zum Erstellen und Bereitstellen von .NET-Lambda-Funktionen mit der AWS SAM, der AWS CDK und der .NET Lambda Global CLI.

Themen

- Verwenden der.NET Lambda Global CLI
- Bereitstellen von C#-Lambda-Funktionen mit AWS SAM

- Bereitstellen von C#-Lambda-Funktionen mit AWS CDK
- ASP.NET-Anwendungen bereitstellen

Verwenden der.NET Lambda Global CLI

Die .NET-CLI und die Erweiterung .NET Lambda Global Tools (`Amazon.Lambda.Tools`) bieten eine plattformübergreifende Möglichkeit, .NET-basierte Lambda-Anwendungen zu erstellen, zu verpacken und für Lambda bereitzustellen. In diesem Abschnitt erfahren Sie, wie Sie neue Lambda .NET-Projekte mit den .NET CLI- und Amazon Lambda-Vorlagen erstellen und diese mithilfe von `Amazon.Lambda.Tools` verpacken und bereitstellen

Themen

- Voraussetzungen
- .NET-Projekte mit der .NET-CLI erstellen
- .NET-Projekte über die .NET-CLI bereitstellen
- Verwendung von Lambda-Ebenen mit der.NET-CLI

Voraussetzungen

.NET 8-SDK

Falls Sie das noch nicht getan haben, installieren Sie das .NET 8-SDK und Laufzeit.

AWS Amazon.Lambda.Templates .NET-Projektvorlagen

Verwenden Sie das Amazon.Lambda.Templates-NuGet-Paket, um Ihren Lambda-Funktionscode zu generieren. Zur Installation dieses Vorlagenpakets führen Sie den folgenden Befehl aus:

```
dotnet new install Amazon.Lambda.Templates
```

AWS Amazon.Lambda.Tools .NET Global CLI-Tools

Verwenden Sie zum Erstellen Ihrer Lambda-Funktionen die Amazon.Lambda.Tools-.NET Global Tools-Erweiterung. Um Amazon.Lambda.Tools zu installieren, führen Sie den folgenden Befehl aus:

```
dotnet tool install -g Amazon.Lambda.Tools
```

Weitere Informationen über die Amazon.Lambda.Tools .NET CLI-Erweiterung finden Sie im AWS-Erweiterungen für .NET-CLI-Repository auf GitHub.

.NET-Projekte mit der .NET-CLI erstellen

Verwenden Sie in der .NET CLI den `dotnet new`-Befehl zum Erstellen von .NET-Projekten in der Befehlszeile. Lambda bietet mithilfe des `Amazon.Lambda.Templates`-NuGet-Pakets zusätzliche Vorlagen an.

Nach der Installation des Pakets führen Sie den folgenden Befehl aus, um eine Liste des verfügbaren Vorlagen zu sehen.

```
dotnet new list
```

Wenn Sie Details zu einer Vorlage einsehen möchten, verwenden Sie die `help`-Option. Um beispielsweise Details über die `lambda.EmptyFunction`-Vorlage zu sehen, führen Sie den folgenden Befehl aus.

```
dotnet new lambda.EmptyFunction --help
```

Verwenden Sie die `lambda.EmptyFunction`-Vorlage, um eine Basisvorlage für eine .NET-Lambda-Funktion zu erstellen. Dadurch wird eine einfache Funktion erstellt, die eine Zeichenfolge als Eingabe verwendet und diese mithilfe der `ToUpper`-Methode in Großbuchstaben umwandelt. Diese Vorlage unterstützt die folgenden Optionen.

- `--name` – Der Name der Funktion.
- `--region` – Die AWS-Region, in der die Funktion erstellt werden soll.
- `--profile` – Der Name eines Profils in Ihrer AWS SDK für .NET-Anmeldeinformationsdatei. Weitere Informationen zu Anmeldeinformationsprofilen in.NET findenSie unter AWS-Anmeldeinformation konfigurieren im AWS-SDK für .NET Entwicklerhandbuch.

In diesem Beispiel erstellen wir eine neue leere Funktion mit dem Namen `myDotnetFunction` unter Verwendung des Standardprofils und der AWS-Region-Einstellungen:

```
dotnet new lambda.EmptyFunction --name myDotnetFunction
```

Mit diesem Befehl werden die folgenden Dateien und Verzeichnisse in Ihrem Projektverzeichnis erstellt.

```
### myDotnetFunction
    ### src
    #   ### myDotnetFunction
    #       ### Function.cs
    #       ### Readme.md
    #       ### aws-lambda-tools-defaults.json
    #       ### myDotnetFunction.csproj
    ### test
        ### myDotnetFunction.Tests
            ### FunctionTest.cs
            ### myDotnetFunction.Tests.csproj
```

Zeigen Sie im Verzeichnis `src/myDotnetFunction` die folgenden Dateien an:

- aws-lambda-tools-defaults.json: Hier geben Sie die Befehlszeilenoptionen an, wenn Sie Ihre Lambda-Funktion bereitstellen. Zum Beispiel:

  ```
    "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: Der Funktionscode Ihres Lambda-Handlers. Dies ist eine C #-Vorlage mit der `Amazon.Lambda.Core`-Standardbibliothek und einem `LambdaSerializer`-Standardattribut. Weitere Informationen zu den Serialisierungsanforderungen und -optionen finden Sie unter Serialisierung in C#-Lambda-Funktionen. Es ist auch eine Beispielfunktion enthalten, die Sie bearbeiten können, um Lambda-Funktionscode anzuwenden.

  ```
  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: Eine MSBuild-Datei, in der die Dateien und Assemblys aufgeführt werden, aus denen Ihre Anwendung besteht.

```
<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: Verwenden Sie diese Datei, um Ihre Lambda-Funktion zu dokumentieren.

Zeigen Sie im Verzeichnis `myfunction/test` die folgenden Dateien an:

- myDotnetFunction.Tests.csproj: Wie bereits erwähnt, handelt es sich hierbei um eine MSBuild-Datei, in der die Dateien und Assembys aufgeführt werden, aus denen Ihr Testprojekt besteht. Beachten Sie auch, dass die `Amazon.Lambda.Core`-Bibliothek enthalten ist. Damit können Sie nahtlos beliebige Lambda-Vorlagen integrieren, die zum Testen Ihrer Funktion erforderlich sind.

```
<Project Sdk="Microsoft.NET.Sdk">
   ...

    <PackageReference Include="Amazon.Lambda.Core" Version="2.2.0 " />
   ...
```

- FunctionTest.cs: Die gleiche C # -Vorlagendatei, die sich im `src`-Verzeichnis befindet. Bearbeiten Sie diese Datei, damit sie dem Produktionscode Ihrer Funktion gleicht, und testen Sie sie vor dem Hochladen Ihrer Lambda-Funktion in eine Produktionsumgebung.

```
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);
        }
    }
}
```

.NET-Projekte über die .NET-CLI bereitstellen

Um Ihr Bereitstellungspaket zu erstellen und es auf Lambda bereitzustellen, verwenden Sie die `Amazon.Lambda.Tools`-CLI-Tools. Um Ihre Funktion anhand der Dateien bereitzustellen, die Sie in den vorherigen Schritten erstellt haben, navigieren Sie zunächst zu dem Ordner, der die `.csproj`-Datei Ihrer Funktion enthält.

```
cd myDotnetFunction/src/myDotnetFunction
```

Führen Sie den folgenden Befehl aus, um Ihren Code als ZIP-Bereitstellungspaket für Lambda bereitzustellen. Wählen Sie Ihren eigenen Funktionsnamen.

```
dotnet lambda deploy-function myDotnetFunction
```

Während der Bereitstellung fordert Sie der Assistent auf, einen the section called “Ausführungsrolle (Berechtigungen für Funktionen zum Zugriff auf andere Ressourcen)” auszuwählen. Wählen Sie für dieses Beispiel die `lambda_basic_role`.

Nachdem Sie Ihre Funktion bereitgestellt haben, können Sie sie mit dem `dotnet lambda invoke-function`-Befehl in der Cloud testen. Für den Beispielcode in der `lambda.EmptyFunction`-Vorlage können Sie Ihre Funktion testen, indem Sie mit der `--payload`-Option eine Zeichenfolge übergeben.

```
dotnet lambda invoke-function myDotnetFunction --payload "Just checking if everything
 is OK"
```

Wenn Ihre Funktion erfolgreich bereitgestellt wurde, sollten Sie eine Ausgabe ähnlich der folgenden sehen.

```
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
```

Verwendung von Lambda-Ebenen mit der.NET-CLI

> **Note**
>
> Es ist zwar möglich, Ebenen mit Funktionen in .NET zu verwenden, wir raten jedoch davon ab. Funktionen in .NET, die Ebenen verwenden, laden die gemeinsam genutzten Assemblys während der `Init`-Phase manuell in den Speicher, was die Kaltstartzeiten verlängern kann. Schließen Sie stattdessen den gesamten freigegebenen Code zur Kompilierzeit ein, um die integrierten Optimierungen des .NET-Compilers zu nutzen.

Die .NET-CLI unterstützt Befehle, mit denen Sie Ebenen veröffentlichen und C#-Funktionen bereitstellen können, die Ebenen nutzen. Um eine Ebene in einem angegebenen Amazon-S3-Bucket zu veröffentlichen, führen Sie den folgenden Befehl im selben Verzeichnis wie Ihre `.csproj`-Datei aus:

```
dotnet lambda publish-layer <layer_name> --layer-type runtime-package-store --s3-
bucket <s3_bucket_name>
```

Wenn Sie dann Ihre Funktion mithilfe der .NET-CLI bereitstellen, geben Sie im folgenden Befehl den Ebenen-ARN an, den Sie verwenden wollen:

```
dotnet lambda deploy-function <function_name> --function-layers arn:aws:lambda:us-
east-1:123456789012:layer:layer-name:1
```

Ein vollständiges Beispiel einer Hello-World-Funktion finden Sie im Beispiel blank-csharp-with-layer.

Bereitstellen von C#-Lambda-Funktionen mit AWS SAM

Das AWS Serverless Application Model (AWS SAM) ist ein Toolkit, das den Prozess der Erstellung und Ausführung von Serverless-Anwendungen auf AWS optimiert. Sie definieren die Ressourcen für Ihre Anwendung in einer YAML- oder JSON-Vorlage und verwenden die AWS SAM-Befehlszeilenschnittstelle (AWS SAM CLI), um Ihre Anwendungen zu erstellen, zu verpacken und bereitzustellen. Wenn Sie eine Lambda-Funktion aus einer AWS SAM-Vorlage erstellen, erstellt AWS SAM automatisch ein ZIP-Bereitstellungspaket oder ein Container-Image mit Ihrem Funktionscode

und allen von Ihnen angegebenen Abhängigkeiten. AWS SAM setzt dann Ihre Funktion mit Hilfe eines CloudFormation-Stacks ein. Weitere Informationen zur Verwendung von AWS SAM für die Erstellung und Bereitstellung von Lambda-Funktionen finden Sie unter Erste Schritte mit AWS SAM im AWS Serverless Application Model-Entwicklerhandbuch.

Die folgenden Schritte zeigen Ihnen, wie Sie eine .NET Hello World-Beispielanwendung mit AWS SAM herunterladen, erstellen und bereitstellen können. Diese Beispielanwendung verwendet eine Lambda-Funktion und einen Amazon API Gateway-Endpunkt, um ein grundlegendes API-Backend zu implementieren. Wenn Sie eine HTTP GET-Anforderung an Ihren API Gateway-Endpunkt senden, ruft API Gateway Ihre Lambda-Funktion auf. Die Funktion gibt eine „Hello World“ -Meldung zusammen mit der IP-Adresse der Lambda-Funktionsinstance zurück, die Ihre Anfrage verarbeitet.

Wenn Sie Ihre Anwendung mithilfe von AWS SAM erstellen und bereitstellen, verwendet die AWS SAM-CLI hinter den Kulissen den Befehl `dotnet lambda package`, um die einzelnen Lambda-Funktions-Codebündel zu verpacken.

Voraussetzungen

.NET 8-SDK

Installieren Sie das .NET 8-SDK und Runtime.

AWS SAM-CLI-Version 1.39 oder höher

Wie Sie die neueste Version der AWS SAM-CLI installieren können, erfahren Sie unter Installation der AWS SAM-CLI.

Bereitstellen einer AWS SAM-Beispielanwendung

1. Initialisieren Sie die Anwendung mit der Hello World-.NET-Vorlage mit dem folgenden Befehl.

```
sam init --app-template hello-world --name sam-app \
--package-type Zip --runtime dotnet8
```

Mit diesem Befehl werden die folgenden Dateien und Verzeichnisse in Ihrem Projektverzeichnis erstellt.

```
### 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. Navigieren Sie in das Verzeichnis, das die `template.yaml file` enthält. Diese Datei ist eine Vorlage, die die AWS-Ressourcen für Ihre Anwendung definiert, einschließlich Ihrer Lambda-Funktion und einer API-Gateway-API.

```
cd sam-app
```

3. Um den Quellcode Ihrer Anwendung zu erstellen, führen Sie folgenden Befehl aus.

```
sam build
```

4. Führen Sie den folgenden Befehl aus, um Ihre Anwendung auf AWS bereitzustellen.

```
sam deploy --guided
```

Dieser Befehl packt Ihre Anwendung und stellt sie mit der folgenden Reihe von Eingabeaufforderungen bereit. Um die Standardoptionen zu übernehmen, drücken Sie die Eingabetaste.

> **Note**
>
> Für HelloWorldFunction ist möglicherweise keine Autorisierung definiert. Ist das in Ordnung?, stellen Sie sicher, dass Sie y eingeben.

- Stack-Name: Der Name des Stacks, der in CloudFormation bereitgestellt werden soll. Dieser Name muss für Ihr AWS-Konto und AWS-Region eindeutig sein.

- AWS-Region: Die AWS-Region, in der Sie Ihre Anwendung bereitstellen möchten.
- Änderungen vor der Bereitstellung bestätigen: Wählen Sie Ja, um alle Änderungssätze manuell zu überprüfen, bevor AWS SAM die Anwendungsänderungen bereitstellt. Wenn Sie nein wählen, werden Anwendungsänderungen automatisch von der AWS SAM CLI bereitgestellt.
- Erlauben Sie die Erstellung von SAM-CLI-IAM-Rollen: Viele AWS SAM-Vorlagen, einschließlich der Hello world-Vorlage in diesem Beispiel, erstellen AWS Identity and Access Management (IAM)-Rollen, um Ihren Lambda-Funktionen die Berechtigung zum Zugriff auf andere AWS-Services zu geben. Wählen Sie Ja, um die Berechtigung zur Bereitstellung eines CloudFormation-Stacks zu erteilen, der IAM-Rollen erstellt oder ändert.
- Rollback deaktivieren: Wenn AWS SAM bei der Erstellung oder Bereitstellung Ihres Stacks auf einen Fehler stößt, wird der Stack standardmäßig auf die vorherige Version zurückgesetzt. Wählen Sie Nein, um diese Standardeinstellung zu akzeptieren.
- HelloWorldFunction hat möglicherweise keine Berechtigung definiert, ist dies in Ordnung: Geben Sie y ein.
- Argumente in samconfig.toml speichern: Wählen Sie Ja, um Ihre Konfigurationsauswahl zu speichern. In Zukunft können Sie `sam deploy` ohne Parameter erneut ausführen, um Änderungen an Ihrer Anwendung vorzunehmen.

5. Wenn die Bereitstellung Ihrer Anwendung abgeschlossen ist, gibt die CLI den Amazon Resource Name (ARN) der Hello World Lambda-Funktion und die für sie erstellte IAM-Rolle zurück. Sie zeigt auch den Endpunkt Ihrer API-Gateway-API an. Um Ihre Anwendung zu testen, öffnen Sie den Endpunkt in einem Browser. Es wird eine Antwort ähnlich der folgenden angezeigt.

```
{"message":"hello world","location":"34.244.135.203"}
```

6. Um Ihre Ressourcen zu löschen, führen Sie den folgenden Befehl aus. Beachten Sie, dass der von Ihnen erstellte API-Endpunkt ein öffentlicher Endpunkt ist, auf den über das Internet zugegriffen werden kann. Es wird empfohlen, dass Sie diesen Endpunkt nach dem Testen löschen.

```
sam delete
```

Nächste Schritte

Weitere Informationen über die Verwendung von AWS SAM zur Erstellung und Bereitstellung von Lambda-Funktionen mit .NET finden Sie in den folgenden Ressourcen:

- Das AWS Serverless Application Model (AWS SAM) Entwicklerhandbuch
- Serverless .NET-Anwendungen mit AWS Lambda und der SAM-CLI

Bereitstellen von C#-Lambda-Funktionen mit AWS CDK

Das AWS Cloud Development Kit (AWS CDK) ist ein Open-Source-Framework für die Softwareentwicklung zur Definition von Cloud-Infrastrukturen als Code mit modernen Programmiersprachen und Frameworks wie .NET. AWS CDK-Projekte werden ausgeführt, um CloudFormation-Vorlagen zu generieren, die dann für die Bereitstellung Ihres Codes verwendet werden.

Befolgen Sie die Anweisungen in den folgenden Abschnitten, um eine Hello world .NET-Beispielanwendung mit dem AWS CDK zu erstellen und bereitzustellen. Die Beispielanwendung implementiert ein grundlegendes API-Backend, das aus einem API-Gateway-Endpunkt und einer Lambda-Funktion besteht. API Gateway ruft die Lambda-Funktion auf, wenn Sie eine HTTP-GET-Anforderung an den Endpunkt senden. Die Funktion gibt eine Hello-World-Nachricht zusammen mit der IP-Adresse der Lambda-Instance zurück, die Ihre Anfrage verarbeitet.

Voraussetzungen

.NET 8-SDK

Installieren Sie das .NET 8-SDK und Runtime.

AWS CDK Version 2

Informationen zur Installation der neuesten Version von AWS CDK finden Sie unter Erste Schritte mit AWS CDK im AWS Cloud Development Kit (AWS CDK)-v2-Entwicklerhandbuch.

Bereitstellen einer AWS CDK-Beispielanwendung

1. Erstellen Sie ein Projektverzeichnis für die Beispielanwendung und navigieren Sie dorthin.

```
mkdir hello-world
```

```
cd hello-world
```

2. Initialisieren Sie eine neue AWS CDK-Anwendung, indem Sie den folgenden Befehl ausführen.

```
cdk init app --language csharp
```

Der Befehl erstellt die folgenden Dateien und Verzeichnisse in Ihrem Projektverzeichnis

```
### README.md
### cdk.json
### src
    ### HelloWorld
    #   ### GlobalSuppressions.cs
    #   ### HelloWorld.csproj
    #   ### HelloWorldStack.cs
    #   ### Program.cs
    ### HelloWorld.sln
```

3. Öffnen Sie das `src`-Verzeichnis und erstellen Sie eine neue Lambda-Funktion mit der.NET-CLI. Dies ist die Funktion, die Sie mit dem AWS CDK einsetzen werden. In diesem Beispiel erstellen Sie eine Hello world-Funktion mit dem Namen `HelloWorldLambda` unter Verwendung der `lambda.EmptyFunction`-Vorlage.

```
cd src
dotnet new lambda.EmptyFunction -n HelloWorldLambda
```

Nach diesem Schritt sollte Ihre Verzeichnisstruktur innerhalb Ihres Projektverzeichnisses wie folgt aussehen.

```
### 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. Öffnen Sie die `HelloWorldStack.cs`-Datei aus dem Verzeichnis `src/HelloWorld`. Ersetzen Sie den Inhalt der Datei durch den folgenden Code.

```
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
                }),
            });
        }
    }
}
```

Dies ist der Code zum Kompilieren und Bündeln des Anwendungscodes sowie die Definition der Lambda-Funktion selbst. Mit dem `BundlingOptions`-Objekt kann eine Zip-Datei erstellt werden, zusammen mit einer Reihe von Befehlen, die zur Erzeugung des Inhalts der Zip-Datei verwendet werden. In diesem Fall wird der Befehl `dotnet lambda package` zum Kompilieren und Erzeugen der Zip-Datei verwendet.

5. Um Ihre Anwendung bereitzustellen, führen Sie den folgenden Befehl aus.

```
cdk deploy
```

6. Rufen Sie Ihre bereitgestellte Lambda-Funktion mit der .NET Lambda CLI auf.

```
dotnet lambda invoke-function HelloWorldFunction -p "hello world"
```

7. Nach Abschluss der Tests können Sie die erstellten Ressourcen löschen, es sei denn, Sie möchten sie beibehalten. Führen Sie den folgenden Befehl aus, um Ihre Ressourcen zu löschen.

```
cdk destroy
```

Nächste Schritte

Weitere Informationen über die Verwendung von AWS CDK zur Erstellung und Bereitstellung von Lambda-Funktionen mit .NET finden Sie in den folgenden Ressourcen:

- Arbeiten mit AWS-CDK in C#
- Erstellen, Verpacken und Veröffentlichen von .NET-C#-Lambda-Funktionen mit dem CDK AWS

ASP.NET-Anwendungen bereitstellen

Sie können nicht nur ereignisgesteuerte Funktionen hosten, sondern auch .NET mit Lambda verwenden, um leichtgewichtige ASP.NET-Anwendungen zu hosten. Sie können ASP.NET-Anwendungen mithilfe des `Amazon.Lambda.AspNetCoreServer`-NuGet-Pakets erstellen und bereitstellen. In diesem Abschnitt erfahren Sie, wie Sie eine ASP.NET-Web-API mit dem .NET Lambda CLI-Tooling in Lambda bereitstellen.

Themen

- Voraussetzungen
- Eine ASP.NET-Web-API für Lambda bereitstellen
- Bereitstellung minimaler ASP.NET-APIs für Lambda

Voraussetzungen

.NET 8-SDK

Installieren Sie das .NET 8-SDK und ASP.NET Core Runtime.

Amazon.Lambda.Tools

Verwenden Sie zum Erstellen Ihrer Lambda-Funktionen die Amazon.Lambda.Tools-.NET Global Tools-Erweiterung. Um Amazon.Lambda.Tools zu installieren, führen Sie den folgenden Befehl aus:

```
dotnet tool install -g Amazon.Lambda.Tools
```

Weitere Informationen über die Amazon.Lambda.Tools .NET CLI-Erweiterung finden Sie im AWS-Erweiterungen für .NET-CLI-Repository auf GitHub.

Amazon.Lambda.Templates

Verwenden Sie das Amazon.Lambda.Templates-NuGet-Paket, um Ihren Lambda-Funktionscode zu generieren. Zur Installation dieses Vorlagenpakets führen Sie den folgenden Befehl aus:

```
dotnet new --install Amazon.Lambda.Templates
```

Eine ASP.NET-Web-API für Lambda bereitstellen

Um eine Web-API mit ASP.NET bereitzustellen, können Sie die .NET-Lambda-Vorlagen verwenden, um ein neues Web-API-Projekt zu erstellen. Verwenden Sie den folgenden Befehl, um ein neues ASP.NET-Web-API-Projekt zu initialisieren. Im Beispielbefehl nennen wir das Projekt `AspNetOnLambda`.

```
dotnet new serverless.AspNetCoreWebAPI -n AspNetOnLambda
```

Mit diesem Befehl werden die folgenden Dateien und Verzeichnisse in Ihrem Projektverzeichnis erstellt.

```
.
### 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
```

Wenn Lambda Ihre Funktion aufruft, wird als Einstiegspunkt die Datei `LambdaEntryPoint.cs` verwendet. Die von der .NET Lambda-Vorlage erstellte Datei enthält den folgenden Code.

```
namespace AspNetOnLambda;

public class LambdaEntryPoint : Amazon.Lambda.AspNetCoreServer.APIGatewayProxyFunction
```

```
{
    protected override void Init(IWebHostBuilder builder)
    {
        builder
            .UseStartup#Startup#();
    }

    protected override void Init(IHostBuilder builder)
    {
    }
}
```

Der von Lambda verwendete Einstiegspunkt muss von einer der drei Basisklassen im Paket `Amazon.Lambda.AspNetCoreServer` erben. Diese drei Basisklassen sind:

- `APIGatewayProxyFunction`
- `APIGatewayHttpApiV2ProxyFunction`
- `ApplicationLoadBalancerFunction`

Die Standardklasse, die verwendet wird, wenn Sie Ihre `LambdaEntryPoint.cs`-Datei mit der bereitgestellten .NET Lambda-Vorlage erstellen, ist `APIGatewayProxyFunction`. Die Basisklasse, die Sie in Ihrer Funktion verwenden, hängt davon ab, welche API-Schicht vor Ihrer Lambda-Funktion steht.

Jede der drei Basisklassen enthält eine öffentliche Methode namens `FunctionHandlerAsync`. Der Name dieser Methode wird Teil der Handler-String sein, die Lambda verwendet, um Ihre Funktion aufzurufen. Die `FunctionHandlerAsync`-Methode wandelt den eingehenden Ereignis-Payload in das korrekte ASP.NET-Format und die ASP.NET-Antwort zurück in einen Lambda-Antwort-Payload um. Für das gezeigte Beispielprojekt `AspNetOnLambda` würde der Handler-String wie folgt lauten.

```
AspNetOnLambda::AspNetOnLambda.LambdaEntryPoint::FunctionHandlerAsync
```

Um die API in Lambda bereitzustellen, führen Sie die folgenden Befehle aus, um in das Verzeichnis mit Ihrer Quellcodedatei zu navigieren und Ihre Funktion mit CloudFormation bereitzustellen.

```
cd AspNetOnLambda/src/AspNetOnLambda
dotnet lambda deploy-serverless
```

> **Tip**
>
> Wenn Sie eine API mit dem Befehl **`dotnet lambda deploy-serverless`** bereitstellen, gibt CloudFormation Ihrer Lambda-Funktion einen Namen, der auf dem Stapelnamen basiert, den Sie während der Bereitstellung angeben. Um Ihrer Lambda-Funktion einen benutzerdefinierten Namen zu geben, bearbeiten Sie die `serverless.template`-Datei, um der `AWS::Serverless::Function`-Ressource eine `FunctionName`-Eigenschaft hinzuzufügen. Weitere Informationen finden Sie unter Namenstyp im CloudFormation-Benutzerhandbuch.

Bereitstellung minimaler ASP.NET-APIs für Lambda

Um eine ASP.NET-Minimal-API für Lambda bereitzustellen, können Sie die .NET-Lambda-Vorlagen verwenden, um ein neues Minimal-API-Projekt zu erstellen. Verwenden Sie den folgenden Befehl, um ein neues minimales API-Projekt zu initialisieren. In diesem Beispiel nennen wir das Projekt `MinimalApiOnLambda`.

```
dotnet new serverless.AspNetCoreMinimalAPI -n MinimalApiOnLambda
```

Der Befehl erstellt die folgenden Dateien und Verzeichnisse in Ihrem Projektverzeichnis.

```
### MinimalApiOnLambda
    ### src
        ### MinimalApiOnLambda
            ### Controllers
            #   ### CalculatorController.cs
            ### MinimalApiOnLambda.csproj
            ### Program.cs
            ### Readme.md
            ### appsettings.Development.json
            ### appsettings.json
            ### aws-lambda-tools-defaults.json
            ### serverless.template
```

Die Datei `Program.cs` enthält den folgenden Code.

```
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();
```

Um Ihre Minimal-API für die Ausführung auf Lambda zu konfigurieren, müssen Sie diesen Code möglicherweise bearbeiten, damit Anfragen und Antworten zwischen Lambda und ASP.NET Core ordnungsgemäß übersetzt werden. Standardmäßig ist die Funktion für eine REST-API-Ereignisquelle konfiguriert. Für eine HTTP-API oder einen Application Load Balancer ersetzen Sie `(LambdaEventSource.RestApi)` durch eine der folgenden Optionen:

- `(LambdaEventSource.HttpAPi)`
- `(LambdaEventSource.ApplicationLoadBalancer)`

Um Ihre Minimal-API in Lambda bereitzustellen, führen Sie die folgenden Befehle aus, um in das Verzeichnis mit Ihrer Quellcodedatei zu navigieren und Ihre Funktion mit CloudFormation bereitzustellen.

```
cd MinimalApiOnLambda/src/MinimalApiOnLambda
dotnet lambda deploy-serverless
```

Arbeiten mit Ebenen für .NET-Lambda-Funktionen

Wir empfehlen nicht, Ebenen zu verwenden, um Abhängigkeiten für in .NET geschriebene Lambda-Funktionen zu verwalten. Der Grund dafür ist, dass .NET eine kompilierte Sprache ist und Ihre Funktionen während der Init-Phase trotzdem alle gemeinsam genutzten Assemblys manuell in den Speicher laden müssen, was die Kaltstartzeiten verlängern kann. Die Verwendung von Ebenen verkompliziert nicht nur den Bereitstellungsprozess, sondern verhindert auch, dass Sie die Vorteile der integrierten Compiler-Optimierungen nutzen können.

Um externe Abhängigkeiten mit Ihren .NET-Handlern zu verwenden, fügen Sie diese bei der Kompilierung direkt in Ihr Bereitstellungspaket ein. So vereinfachen Sie den Bereitstellungsprozess und können außerdem die integrierten .NET-Compiler-Optimierungen nutzen. Ein Beispiel dafür, wie Sie Abhängigkeiten wie NuGet-Pakete in Ihre Funktion importieren und verwenden können, finden Sie unter the section called “Handler”.

Bereitstellen von .NET-Lambda-Funktionen mit Container-Images

Es gibt drei Möglichkeiten, ein Container-Image für eine .NET-Lambda-Funktion zu erstellen:

- Verwenden eines AWS Basis-Images für.NET

  Die AWS -Basis-Images sind mit einer Sprachlaufzeit, einem Laufzeitschnittstellen-Client zur Verwaltung der Interaktion zwischen Lambda und Ihrem Funktionscode und einem Laufzeitschnittstellen-Emulator für lokale Tests vorinstalliert.

- Es wird ein AWS reines Betriebssystem-Basis-Image verwendet

  AWS Basis-Images nur für Betriebssysteme enthalten eine Amazon Linux-Distribution und den Runtime-Interface-Emulator. Diese Images werden häufig verwendet, um Container-Images für kompilierte Sprachen wie Go und Rust sowie für eine Sprache oder Sprachversion zu erstellen, für die Lambda kein Basis-Image bereitstellt, wie Node.js 19. Sie können reine OS-Basis-Images auch verwenden, um eine benutzerdefinierte Laufzeit zu implementieren. Um das Image mit Lambda kompatibel zu machen, müssen Sie den Laufzeitschnittstellen-Client für .NET in das Image aufnehmen.

- Verwenden eines Nicht-Basis-Images AWS

  Sie können auch ein alternatives Basis-Image aus einer anderen Container-Registry verwenden. Sie können auch ein von Ihrer Organisation erstelltes benutzerdefiniertes Image verwenden. Um das Image mit Lambda kompatibel zu machen, müssen Sie den Laufzeitschnittstellen-Client für .NET in das Image aufnehmen.

> **Tip**
>
> Um die Zeit zu reduzieren, die benötigt wird, bis Lambda-Container-Funktionen aktiv werden, siehe die Docker-Dokumentation unter Verwenden mehrstufiger Builds. Um effiziente Container-Images zu erstellen, folgen Sie den Bewährte Methoden für das Schreiben von Dockerfiles.

Auf dieser Seite wird erklärt, wie Sie Container-Images für Lambda erstellen, testen und bereitstellen.

Themen

- AWS Basis-Images für.NET

- Verwenden eines AWS Basis-Images für.NET
- Verwenden eines alternativen Basis-Images mit dem Laufzeitschnittstellen-Client

AWS Basis-Images für.NET

AWS stellt die folgenden Basis-Images für.NET bereit:

Tags (Markier ngen)	Laufzeit	Betriebss ystem	Dockerfile	Ablehnung
10	.NET 10	Amazon Linux 2023	Dockerfile für .NET 10 auf GitHub	14. November 2028
9	.NET 9	Amazon Linux 2023	Dockerfile für .NET 9 auf GitHub	10. November 2026
8	.NET 8	Amazon Linux 2023	Dockerfile für .NET 8 auf GitHub	10. November 2026

Amazon ECR-Repository: gallery.ecr. aws/lambda/dotnet

Verwenden eines AWS Basis-Images für.NET

Voraussetzungen

Zur Durchführung der Schritte in diesem Abschnitt benötigen Sie Folgendes:

- .NET SDK – Die folgenden Schritte verwenden das .NET 8-Basis-Image. Stellen Sie sicher, dass Ihre .NET-Version mit der Version des Basis-Images übereinstimmt, die Sie in Ihrer Docker-Datei angeben.
- Docker (Mindestversion 25.0.0)
- Das Docker-buildx-Plugin.

Erstellen und Bereitstellen eines Images mithilfe eines Basis-Images

In den folgenden Schritten verwenden Sie Amazon.Lambda.Templates und Amazon.Lambda.Tools, um ein.NET-Projekt zu erstellen. Anschließend erstellen Sie ein Docker-Image, laden das Image auf Amazon ECR hoch und stellen es auf einer Lambda-Funktion bereit.

1. Installieren Sie das Amazon.Lambda.Templates-Paket. NuGet

   ```
   dotnet new install Amazon.Lambda.Templates
   ```

2. Erstellen Sie ein .NET-Projekt mithilfe der `lambda.image.EmptyFunction`-Vorlage.

   ```
   dotnet new lambda.image.EmptyFunction --name MyFunction --region us-east-1
   ```

   Die Projektdateien werden im *`MyFunction`*`/src/`*`MyFunction`*-Verzeichnis gespeichert:

   - aws-lambda-tools-defaults.json: Gibt die Befehlszeilenoptionen für die Bereitstellung Ihrer Lambda-Funktion an.
   - Function.cs: Der Funktionscode Ihres Lambda-Handlers. Dies ist eine C#-Vorlage mit der `Amazon.Lambda.Core`-Standardbibliothek und einem `LambdaSerializer`-Standardattribut. Weitere Informationen über Serialisierungsanforderungen und -optionen finden Sie unter Serialisierung in C#-Lambda-Funktionen. Sie können den bereitgestellten Code zum Testen verwenden oder ihn durch Ihren eigenen ersetzen.
   - MyFunction.csproj: Eine .NET-Projektdatei, die die Dateien und Assemblys auflistet, aus denen Ihre Anwendung besteht.
   - Dockerfile: Sie können die bereitgestellte Docker-Datei zum Testen verwenden oder sie durch Ihre eigene ersetzen. Wenn Sie Ihre eigenen verwenden, stellen Sie sicher, dass Sie:
     - Setzen Sie die `FROM`-Eigenschaft auf den URI des Basis-Images. Das Basis-Image und das `TargetFramework` in der Datei `MyFunction.csproj` müssen beide dieselbe .NET-Version verwenden. Um beispielsweise .NET 9 zu verwenden:
       - Docker-Datei: `FROM` *`public.ecr.aws/lambda/dotnet:9`*
       - MyFunction.csproj: `<TargetFramework>`*`net9.0`*`</TargetFramework>`
     - Legen Sie das `CMD`-Argument auf den Lambda-Funktionshandler fest. Das sollte mit dem `image-command` in `aws-lambda-tools-defaults.json` übereinstimmen.

3. Installieren Sie Amazon.Lambda.Tools .NET Global Tool.

```
dotnet tool install -g Amazon.Lambda.Tools
```

Wenn Amazon.Lambda.Tools bereits installiert ist, vergewissern Sie sich, dass Sie über die neueste Version verfügen.

```
dotnet tool update -g Amazon.Lambda.Tools
```

4. Ändern Sie das Verzeichnis zu *MyFunction*/src/*MyFunction*, sofern Sie dies noch nicht getan haben.

```
cd src/MyFunction
```

5. Verwenden Sie Amazon.Lambda.Tools, um das Docker-Image zu erstellen, es in ein neues Amazon-ECR-Repository zu verschieben und die Lambda-Funktion bereitzustellen.

   Für `--function-role` geben Sie den Rollennamen – nicht den Amazon-Ressourcenname (ARN) – der Ausführungsrolle für die Funktion an. Beispiel, `lambda-role`.

```
dotnet lambda deploy-function MyFunction --function-role lambda-role
```

   Weitere Informationen zum Amazon.Lambda.Tools .NET Global Tool finden Sie im AWS Extensions for .NET CLI Repository unter. GitHub

6. Die Funktion aufrufen.

```
dotnet lambda invoke-function MyFunction --payload "Testing the function"
```

   Wenn alles erfolgreich ist, sehen Sie eine Antwort ähnlich der folgenden:

```
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. Löschen Sie die Lambda-Funktion.

```
dotnet lambda delete-function MyFunction
```

Verwenden eines alternativen Basis-Images mit dem Laufzeitschnittstellen-Client

Wenn Sie ein OS-Basis-Image oder ein alternatives Basis-Image verwenden, müssen Sie den Laufzeitschnittstellen-Client in das Image einbinden. Der Laufzeitschnittstellen-Client erweitert die Laufzeit-API, die die Interaktion zwischen Lambda und Ihrem Funktionscode verwaltet.

Das folgende Beispiel zeigt, wie Sie mithilfe eines AWS Nicht-Base-Images ein Container-Image für .NET erstellen und wie Sie das Amazon.Lambda hinzufügen. RuntimeSupport Paket, das der Lambda-Runtime-Interface-Client für .NET ist. Das Beispiel-Dockerfile verwendet das Basis-Image „Microsoft .NET 8“.

Voraussetzungen

Zur Durchführung der Schritte in diesem Abschnitt benötigen Sie Folgendes:

- .NET SDK — Für die folgenden Schritte wird ein .NET 9-Basisimage verwendet. Stellen Sie sicher, dass Ihre .NET-Version mit der Version des Basis-Images übereinstimmt, die Sie in Ihrer Docker-Datei angeben.
- Docker (Mindestversion 25.0.0)
- Das Docker-buildx-Plugin.

Erstellen und Bereitstellen eines Images mithilfe eines alternativen Basis-Images

1. Installieren Sie das Amazon.Lambda.Templates-Paket. NuGet

```
dotnet new install Amazon.Lambda.Templates
```

2. Erstellen Sie ein .NET-Projekt mithilfe der `lambda.CustomRuntimeFunction`-Vorlage. Diese Vorlage enthält das Amazon.Lambda. RuntimeSupportPaket.

```
dotnet new lambda.CustomRuntimeFunction --name MyFunction --region us-east-1
```

3. Navigieren Sie zum *MyFunction*/src/*MyFunction* Verzeichnis . Hier werden die Projektdateien gespeichert. Prüfen Sie die folgenden Dateien:

   - aws-lambda-tools-defaults.json — In dieser Datei geben Sie die Befehlszeilenoptionen an, wenn Sie Ihre Lambda-Funktion bereitstellen.
   - Function.cs – Der Code enthält eine Klasse mit einer `Main`-Methode, die die Bibliothek `Amazon.Lambda.RuntimeSupport` als Bootstrap initialisiert. Die `Main`-Methode ist der Einstiegspunkt für den Prozess der Funktion. Die `Main`-Methode verpackt den Funktionshandler in einen Wrapper, mit dem der Bootstrap arbeiten kann. Weitere Informationen finden Sie unter Amazon.Lambda verwenden. RuntimeSupport als Klassenbibliothek im Repository. GitHub
   - MyFunction.csproj — Eine.NET-Projektdatei, die die Dateien und Assemblys auflistet, aus denen Ihre Anwendung besteht.
   - Readme.md – Diese Datei enthält weitere Informationen zur Lambda-Beispielfunktion.

4. Öffnen Sie die Datei `aws-lambda-tools-defaults.json` und fügen Sie die folgenden Zeilen hinzu:

```
  "package-type": "image",
  "docker-host-build-output-dir": "./bin/Release/lambda-publish"
```

   - package-type: Definiert das Bereitstellungspaket als Container-Image.
   - docker-host-build-output-dir: Legt das Ausgabeverzeichnis für den Build-Prozess fest.

   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. Erstellen Sie ein Dockerfile in Verzeichnis *MyFunction*/src/*MyFunction*. Das folgende Beispiel-Dockerfile verwendet ein Microsoft .NET-Basisimage anstelle eines AWS -Basis-Images.

   - Legen Sie FROM-Eigenschaft auf die Kennung des Basis-Images fest. Das Basis-Image und das `TargetFramework` in der Datei `MyFunction.csproj` müssen beide dieselbe .NET-Version verwenden.
   - Verwenden Sie den COPY-Befehl, um die Funktion in das Verzeichnis `/var/task` zu kopieren.
   - Legen Sie ENTRYPOINT auf das Modul fest, das der Docker-Container beim Start ausführen soll. In diesem Fall ist das Modul der Bootstrap, der die Bibliothek `Amazon.Lambda.RuntimeSupport` initialisiert.

   Beachten Sie, dass das Dockerfile-Beispiel keine USER-Anweisung enthält. Wenn Sie ein Container-Image für Lambda bereitstellen, definiert Lambda automatisch einen Standard-Linux-Benutzer mit Berechtigungen mit geringsten Rechten. Dies unterscheidet sich vom Standardverhalten von Docker, bei dem standardmäßig der `root`-Benutzer verwendet wird, wenn keine USER-Anweisung bereitgestellt wird.

   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. Installieren Sie die Amazon.Lambda.Tools .NET Global Tools-Erweiterung.

```
dotnet tool install -g Amazon.Lambda.Tools
```

Wenn Amazon.Lambda.Tools bereits installiert ist, vergewissern Sie sich, dass Sie über die neueste Version verfügen.

```
dotnet tool update -g Amazon.Lambda.Tools
```

7. Verwenden Sie Amazon.Lambda.Tools, um das Docker-Image zu erstellen, es in ein neues Amazon-ECR-Repository zu verschieben und die Lambda-Funktion bereitzustellen.

Für `--function-role` geben Sie den Rollennamen – nicht den Amazon-Ressourcenname (ARN) – der Ausführungsrolle für die Funktion an. Beispiel, `lambda-role`.

```
dotnet lambda deploy-function MyFunction --function-role lambda-role
```

Weitere Informationen zur .NET-CLI-Erweiterung Amazon.Lambda.Tools finden Sie im AWS Extensions for .NET CLI-Repository unter. GitHub

8. Die Funktion aufrufen.

```
dotnet lambda invoke-function MyFunction --payload "Testing the function"
```

Wenn alles erfolgreich ist, sehen Sie Folgendes:

```
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. Löschen Sie die Lambda-Funktion.

```
dotnet lambda delete-function MyFunction
```

Kompilieren Sie.NET-Lambda-Funktionscode in ein natives Laufzeitformat

.NET 8 unterstützt die native Ahead-of-Time-Kompilierung (AOT). Mit nativem AOT können Sie Ihren Lambda-Funktionscode in ein natives Laufzeitformat kompilieren, wodurch die Notwendigkeit entfällt, .NET-Code zur Laufzeit zu kompilieren. Die native AOT-Kompilierung kann die Kaltstartzeit für Lambda-Funktionen reduzieren, die Sie in .NET schreiben. Weitere Informationen finden Sie unter Introducing the .NET 8 runtime for AWS Lambda im AWS-Compute-Blog.

Abschnitte

- Lambda-Laufzeit
- Voraussetzungen
- Erste Schritte
- Serialisierung
- Trimmen
- Fehlerbehebung

Lambda-Laufzeit

Um eine Lambda-Funktion mit nativer AOT-Kompilierung zu erstellen, verwenden Sie die verwaltete .NET 8 Lambda-Laufzeit. Diese Laufzeit unterstützt die Verwendung sowohl der x86_64- als auch arm64-Architektur.

Wenn Sie eine .NET Lambda-Funktion ohne AOT bereitstellen, wird Ihre Anwendung zunächst in IL-Code (Intermediate Language) kompiliert. Zur Laufzeit nimmt der Just-in-Time-Compiler (JIT) in der Lambda-Laufzeitumgebung den IL-Code und kompiliert ihn nach Bedarf in Maschinencode. Mit einer Lambda-Funktion, die im Voraus mit nativem AOT kompiliert wird, kompilieren Sie Ihren Code bei der Bereitstellung Ihrer Funktion in Maschinencode, sodass Sie nicht auf die .NET-Laufzeit oder das SDK in der Lambda-Laufzeit angewiesen sind, um Ihren Code vor der Ausführung zu kompilieren.

Eine Einschränkung von AOT ist, dass Ihr Anwendungscode in einer Umgebung mit demselben Amazon Linux 2023 (AL2023) Betriebssystem kompiliert werden muss, das auch die .NET 8-Laufzeitumgebung verwendet. Die .NET Lambda-CLI bietet Funktionen zum Kompilieren Ihrer Anwendung in einem Docker-Container mithilfe eines AL2023-Images.

Um mögliche Probleme mit der architekturübergreifenden Kompatibilität zu vermeiden, empfehlen wir dringend, dass Sie Ihren Code in einer Umgebung mit derselben Prozessorarchitektur kompilieren, die Sie für Ihre Funktion konfigurieren. Weitere Informationen zu den Einschränkungen der architekturübergreifenden Kompilierung finden Sie unter Cross-Compilierung in der Microsoft-.NET-Dokumentation.

Voraussetzungen

Docker

Um natives AOT zu verwenden, muss Ihr Funktionscode in einer Umgebung mit demselben AL2023-Betriebssystem wie die .NET 8-Laufzeit kompiliert werden. Die .NET-CLI-Befehle in den folgenden Abschnitten verwenden Docker, um Lambda-Funktionen in einer AL2023-Umgebung zu entwickeln und zu erstellen.

.NET 8-SDK

Die native AOT-Kompilierung ist ein Feature von .NET 8. Sie müssen das .NET 8-SDK auf Ihrer Build-Machine installieren, nicht nur die Laufzeit.

Amazon.Lambda.Tools

Verwenden Sie zum Erstellen Ihrer Lambda-Funktionen die Amazon.Lambda.Tools-.NET Global Tools-Erweiterung. Um Amazon.Lambda.Tools zu installieren, führen Sie den folgenden Befehl aus:

```
dotnet tool install -g Amazon.Lambda.Tools
```

Weitere Informationen über die Amazon.Lambda.Tools .NET CLI-Erweiterung finden Sie im AWS-Erweiterungen für .NET-CLI-Repository auf GitHub.

Amazon.Lambda.Templates

Verwenden Sie das Amazon.Lambda.Templates-NuGet-Paket, um Ihren Lambda-Funktionscode zu generieren. Zur Installation dieses Vorlagenpakets führen Sie den folgenden Befehl aus:

```
dotnet new install Amazon.Lambda.Templates
```

Erste Schritte

Sowohl das.NET Global CLI als auch AWS Serverless Application Model (AWS SAM) bieten Vorlagen für erste Schritte zum Erstellen von Anwendungen mit nativem AOT. Um Ihre erste native AOT-Lambda-Funktion zu erstellen, führen Sie die Schritte in den folgenden Anweisungen aus.

Um eine native AOT-kompilierte Lambda-Funktion zu initialisieren und bereitzustellen

1. Initialisieren Sie ein neues Projekt unter Verwendung der nativen AOT-Vorlage und navigieren Sie dann in das Verzeichnis, das die erstellten `.cs`- und `.csproj`-Dateien enthält. In diesem Beispiel nennen wir unsere Funktion `NativeAotSample`.

   ```
   dotnet new lambda.NativeAOT -n NativeAotSample
   cd ./NativeAotSample/src/NativeAotSample
   ```

   Die von der nativen AOT-Vorlage erstellte `Function.cs`-Datei enthält den folgenden Funktionscode.

   ```
   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
```

Native AOT kompiliert Ihre Anwendung in eine einzige native Binärdatei. Der Einstiegspunkt dieser Binärdatei ist die `static Main`-Methode. Innerhalb von `static Main` wird die Lambda-Laufzeit gebootstrapped und die `FunctionHandler`-Methode eingerichtet. Als Teil des Runtime-Bootstrap wird ein quellgenerierter Serializer mit `new SourceGeneratorLambdaJsonSerializer<LambdaFunctionJsonSerializerContext>()` konfiguriert

2. Um Ihre Anwendung auf Lambda bereitzustellen, stellen Sie sicher, dass Docker in Ihrer lokalen Umgebung ausgeführt wird, und führen Sie den folgenden Befehl aus.

```
dotnet lambda deploy-function
```

Hinter den Kulissen lädt das .NET Global CLI ein AL2023-Docker-Image herunter und kompiliert Ihren Anwendungscode in einem laufenden Container. Die kompilierte Binärdatei wird zurück in Ihr lokales Dateisystem ausgegeben, bevor sie auf Lambda bereitgestellt wird.

3. Testen Sie Ihre Funktion, indem Sie den folgenden Befehl ausführen. Ersetzen Sie `<FUNCTION_NAME>` durch den Namen, den Sie für Ihre Funktion im Bereitstellungsassistenten gewählt haben.

```
dotnet lambda invoke-function <FUNCTION_NAME> --payload "hello world"
```

Die Antwort der CLI enthält Leistungsdetails für den Kaltstart (Initialisierungsdauer) und die Gesamtlaufzeit für Ihren Funktionsaufruf.

4. Führen Sie den folgenden Befehl aus, um die AWS-Ressourcen zu löschen, die Sie mit den vorherigen Schritten erstellt haben. Ersetzen Sie `<FUNCTION_NAME>` durch den Namen, den Sie für Ihre Funktion im Bereitstellungsassistenten gewählt haben. Durch das Löschen von AWS-

Ressourcen, die Sie nicht mehr verwenden, können Sie verhindern, dass unnötige Gebühren Ihrem AWS-Konto-Konto in Rechnung gestellt werden.

```
dotnet lambda delete-function <FUNCTION_NAME>
```

Serialisierung

Um Funktionen mithilfe von nativem AOT für Lambda bereitzustellen, muss Ihr Funktionscode die quellgenerierte Serialisierung verwenden. Anstatt mithilfe der Laufzeitreflexion die Metadaten zu sammeln, die für den Zugriff auf Objekteigenschaften für die Serialisierung erforderlich sind, generieren Quellgeneratoren C#-Quelldateien, die beim Erstellen Ihrer Anwendung kompiliert werden. Um Ihren quellgenerierten Serializer korrekt zu konfigurieren, stellen Sie sicher, dass Sie alle Eingabe- und Ausgabeobjekte, die Ihre Funktion verwendet, sowie alle benutzerdefinierten Typen einbeziehen. Beispielsweise würde eine Lambda-Funktion, die Ereignisse von einer API-Gateway-REST-API empfängt und einen benutzerdefinierten `Product`-Typ zurückgibt, einen Serializer enthalten, der wie folgt definiert ist.

```
[JsonSerializable(typeof(APIGatewayProxyRequest))]
[JsonSerializable(typeof(APIGatewayProxyResponse))]
[JsonSerializable(typeof(Product))]
public partial class CustomSerializer : JsonSerializerContext
{
}
```

Trimmen

Natives AOT kürzt Ihren Anwendungscode als Teil der Kompilierung, um sicherzustellen, dass die Binärdatei so klein wie möglich ist. .NET 8 for Lambda bietet im Vergleich zu früheren Versionen von .NET eine verbesserte Trimmunterstützung. Support wurde zu den Lambda-Laufzeitbibliotheken, AWS .NET SDK, .NET Lambda Annotations und .NET 8 selbst hinzugefügt.

Diese Verbesserungen bieten das Potenzial, Warnungen beim Trimmen während der Erstellung zu eliminieren, aber .NET wird niemals vollständig trimmsicher sein. Das bedeutet, dass Teile von Bibliotheken, auf die Ihre Funktion angewiesen ist, im Rahmen des Kompilierungsschritts entfernt werden können. Sie können dies verwalten, indem Sie `TrimmerRootAssemblies` als Teil Ihrer `.csproj`-Datei definieren, wie im folgenden Beispiel gezeigt.

```
<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>
```

Beachten Sie, dass bei einer Trim-Warnung das Hinzufügen der Klasse, die die Warnung erzeugt, zu `TrimmerRootAssembly` das Problem möglicherweise nicht behebt. Eine Trim-Warnung weist darauf hin, dass die Klasse versucht, auf eine andere Klasse zuzugreifen, die erst zur Laufzeit ermittelt werden kann. Um Laufzeitfehler zu vermeiden, fügen Sie diese zweite Klasse zu `TrimmerRootAssembly` hinzu.

Weitere Informationen zur Verwaltung von Trimmwarnungen finden Sie in der Microsoft.NET-Dokumentation unter Einführung in Trimmwarnungen.

Fehlerbehebung

Fehler: Betriebssystemübergreifende native Kompilierung wird nicht unterstützt.

> Ihre Version des globalen Amazon.Lambda.Tools-.NET-Core-Tools ist veraltet. Aktualisieren Sie auf die neueste Version und versuchen Sie es erneut.

Docker: das Image-Betriebssystem „Linux“ kann auf dieser Plattform nicht verwendet werden.

> Docker ist auf Ihrem System für die Verwendung von Windows-Containern konfiguriert. Wechseln Sie zu Linux-Containern, um die native AOT-Entwicklungsumgebung auszuführen.

Weitere Informationen zu häufigen Fehlern finden Sie im AWS NativeAOT für .NET-Repository auf GitHub.

Verwenden des Lambda-Kontextobjekts zum Abrufen von C#-Funktionsinformationen

Wenn Lambda Ihre Funktion ausführt, wird ein Context-Objekt an den Handler übergeben. Dieses Objekt stellt Eigenschaften mit Informationen zum Aufruf, zur Funktion und zur Ausführungsumgebung bereit.

Context-Eigenschaften

- `FunctionName` – Der Name der Lambda-Funktion.
- `FunctionVersion` – Die Version der Funktion.
- `InvokedFunctionArn` – Der Amazon-Ressourcenname (ARN), der zum Aufrufen der Funktion verwendet wird. Gibt an, ob der Aufrufer eine Versionsnummer oder einen Alias angegeben hat.
- `MemoryLimitInMB` – Die Menge an Arbeitsspeicher, die der Funktion zugewiesen ist.
- `AwsRequestId` – Der Bezeichner der Aufrufanforderung.
- `LogGroupName` – Protokollgruppe für die Funktion.
- `LogStreamName` – Der Protokollstrom für die Funktionsinstance.
- `RemainingTime` (`TimeSpan`) – Die Anzahl der Millisekunden, die vor der Zeitüberschreitung der Ausführung verbleiben.
- `Identity` – Informationen zur Amazon-Cognito-Identität, die die Anforderung autorisiert hat.
- `ClientContext` – (mobile Apps) Clientkontext, der Lambda von der Clientanwendung bereitgestellt wird.
- `Logger` Das Logger-Objekt für die Funktion.

Sie können die Informationen im `ILambdaContext`-Objekt verwenden, um zu Überwachungszwecken Informationen über den Aufruf Ihrer Funktion auszugeben. Der folgende Code enthält ein Beispiel dafür, wie Sie einem strukturierten Logging-Framework Kontextinformationen hinzufügen können. In diesem Beispiel fügt die Funktion `AwsRequestId` zu den Protokollausgaben hinzu. Die Funktion verwendet auch die `RemainingTime`-Eigenschaft, um eine Aufgabe während des Fluges abzubrechen, wenn die Zeitüberschreitung der Lambda-Funktion bald erreicht ist.

```
[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();
        }
```

```
    }
}
```

C#-Lambda-Funktionen protokollieren und überwachen

AWS Lambda überwacht automatisch Lambda-Funktionen und sendet Protokolleinträge an Amazon CloudWatch. Ihre Lambda-Funktion enthält eine CloudWatch Logs-Log-Gruppe und einen Log-Stream für jede Instanz Ihrer Funktion. Die Lambda-Laufzeitumgebung sendet Details zu den einzelnen Aufrufen an den Protokollstream und leitet Protokolle und andere Ausgaben aus dem Code Ihrer Funktion weiter. Weitere Informationen zu CloudWatch Logs finden Sie unterSenden von Lambda-Funktionsprotokollen an CloudWatch Logs.

Sections

- Erstellen einer Funktion, die Protokolle zurückgibt
- Verwenden von Lambda-Optionen für die erweiterte Protokollierung mit .NET
- Zusätzliche Protokollierungswerkzeuge und Bibliotheken
- Verwendung von Powertools für AWS Lambda (.NET) und für strukturiertes Logging AWS SAM
- Anzeigen von Protokollen in der Lambda-Konsole
- Protokolle in der CloudWatch Konsole anzeigen
- Logs mit dem Befehl () anzeigen AWS Command Line InterfaceAWS CLI
- Löschen von Protokollen

Erstellen einer Funktion, die Protokolle zurückgibt

Um Logs aus Ihrem Funktionscode auszugeben, können Sie den ILambdaLogger für das Kontextobjekt, die Methoden der Console-Klasse oder eine beliebige Logging-Bibliothek verwenden, die in `stdout` oder schreibt`stderr`.

Die .NET-Laufzeit protokolliert die Zeilen `START`, `END` und `REPORT` für jeden Aufruf. Die Berichtszeile enthält die folgenden Details.

Datenfelder für REPORT-Zeilen

- RequestId— Die eindeutige Anforderungs-ID für den Aufruf.
- Dauer – Die Zeit, die die Handler-Methode Ihrer Funktion mit der Verarbeitung des Ereignisses verbracht hat.
- Fakturierte Dauer – Die für den Aufruf fakturierte Zeit.
- Speichergröße – Die der Funktion zugewiesene Speichermenge.

- Max. verwendeter Speicher – Die Speichermenge, die von der Funktion verwendet wird. Wenn Aufrufe eine Ausführungsumgebung gemeinsam nutzen, meldet Lambda den maximalen Speicherverbrauch für alle Aufrufe. Dieses Verhalten kann zu einem höheren als erwarteten gemeldeten Wert führen.
- Initialisierungsdauer – Für die erste Anfrage die Zeit, die zur Laufzeit zum Laden der Funktion und Ausführen von Code außerhalb der Handler-Methode benötigt wurde.
- XRAY TraceId — Für verfolgte Anfragen die AWS X-Ray Trace-ID.
- SegmentId— Für verfolgte Anfragen die X-Ray-Segment-ID.
- Stichprobe – Bei verfolgten Anforderungen das Stichprobenergebnis.

Verwenden von Lambda-Optionen für die erweiterte Protokollierung mit .NET

Um Ihnen mehr Kontrolle darüber zu geben, wie die Protokolle Ihrer Funktionen erfasst, verarbeitet und verwendet werden, können Sie die folgenden Protokollierungsoptionen für unterstützte .NET-Laufzeiten konfigurieren:

- Protokollformat – Wählen Sie zwischen Klartext und einem strukturierten JSON-Format für die Protokolle Ihrer Funktion aus.
- Protokollebene — für Logs im JSON-Format wählen Sie die Detailebene der Logs, an die Lambda sendet CloudWatch, wie ERROR, DEBUG oder INFO
- Protokollgruppe — wählen Sie die CloudWatch Protokollgruppe aus, an die Ihre Funktion Logs sendet

Weitere Informationen zu diesen Protokollierungsoptionen und Anweisungen zur Konfiguration Ihrer Funktion für deren Verwendung finden Sie unter the section called “Konfigurieren erweiterter Protokollierungsoptionen für Lambda-Funktionen”.

Informationen zur Verwendung der Optionen für das Protokollformat und die Protokollebene mit Ihren .NET-Lambda-Funktionen finden Sie in den folgenden Abschnitten.

Verwenden des strukturierten JSON-Protokollformats mit .NET

Wenn Sie JSON für das Protokollformat Ihrer Funktion auswählen, sendet Lambda die Protokollausgabe mithilfe von ILambdaLogger als strukturiertes JSON. Jedes JSON-Protokollobjekt enthält mindestens fünf Schlüssel-Wert-Paare mit den folgenden Schlüsseln:

- `"timestamp"` – die Uhrzeit, zu der die Protokollmeldung generiert wurde
- `"level"` – die der Meldung zugewiesene Protokollebene
- `"requestId"` – die eindeutige Anforderungs-ID für den Funktionsaufruf
- `"traceId"` – Die Umgebungsvariable `_X_AMZN_TRACE_ID`
- `"message"` – der Inhalt der Protokollmeldung

Die `ILambdaLogger`-Instance kann zusätzliche Schlüssel-Wert-Paare hinzufügen, beispielsweise beim Protokollieren von Ausnahmen. Sie können auch Ihre eigenen zusätzlichen Parameter angeben, wie im Abschnitt the section called “Vom Kunden bereitgestellte Protokollparameter” beschrieben.

> **Note**
>
> Wenn Ihr Code bereits eine andere Protokollierungsbibliothek verwendet, um JSON-formatierte Protokolle zu erstellen, stellen Sie sicher, dass das Protokollformat Ihrer Funktion auf Klartext gesetzt ist. Wenn Sie das Protokollformat auf JSON setzen, werden Ihre Protokollausgaben doppelt codiert.

Der folgende Beispiel-Logging-Befehl zeigt, wie Sie eine Protokollnachricht mit dem Level `INFO` schreiben.

Example.NET-Protokollierungscode

```
context.Logger.LogInformation("Fetching cart from database");
```

Sie können auch eine generische Protokollebene verwenden, die die Protokollebene als Argument verwendet, wie im folgenden Beispiel gezeigt.

```
context.Logger.Log(LogLevel.Information, "Fetching cart from database");
```

Die Protokollausgabe dieser Beispielcodefragmente würde wie folgt in CloudWatch Logs erfasst:

Example JSON-Protokolldatensatz

```
{
    "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**
>
> Wenn Sie das Protokollformat Ihrer Funktion so konfigurieren, dass statt JSON reiner Text verwendet wird, entspricht die in der Meldung erfasste Protokollstufe der Microsoft-Konvention, die eine vierstellige Bezeichnung vorsieht. Die Protokollebene von `Debug` wird in der Nachricht beispielsweise als `dbug` dargestellt.
> Wenn Sie Ihre Funktion für die Verwendung von Protokollen im JSON-Format konfigurieren, verwendet die im Protokoll erfasste Protokollebene die vollständige Bezeichnung, wie im Beispiel für den JSON-Protokolldatensatz gezeigt.

Wenn Sie Ihrer Protokollausgabe keine Ebene zuweisen, weist Lambda ihr automatisch die Ebene INFO zu.

Protokollierungsausnahmen in JSON

Wenn Sie die strukturierte JSON-Protokollierung mit `ILambdaLogger` verwenden, können Sie Ausnahmen in Ihrem Code protokollieren, wie im folgenden Beispiel gezeigt.

Example Verwendung der Ausnahmeprotokollierung

```
try
{
    connection.ExecuteQuery(query);
}
catch(Exception e)
{
    context.Logger.LogWarning(e, "Error executing query");
}
```

Das von diesem Code ausgegebene Protokollformat wird im folgenden JSON-Beispiel gezeigt. Beachten Sie, dass die `message`-Eigenschaft im JSON mithilfe des im `LogWarning`-Aufruf angegebenen Nachrichtenarguments aufgefüllt wird, während die `errorMessage`-Eigenschaft aus der `Message`-Eigenschaft der Ausnahme selbst stammt.

Example JSON-Protokolldatensatz

```
{
    "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>"]
}
```

Wenn das Protokollierungsformat Ihrer Funktion auf JSON gesetzt ist, gibt Lambda auch Protokollnachrichten im JSON-Format aus, wenn Ihr Code eine nicht abgefangene Ausnahme auslöst. Der folgende Beispielcodeausschnitt und die Protokollnachricht zeigen, wie nicht abgefangene Ausnahmen protokolliert werden.

Example Ausnahmecode

```
throw new ApplicationException("Invalid data");
```

Example JSON-Protokolldatensatz

```
{
    "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>"]
}
```

Vom Kunden bereitgestellte Protokollparameter

Bei JSON-formatierten Protokollmeldungen können Sie zusätzliche Protokollparameter angeben und diese in das Protokoll `message` aufnehmen. Der folgende Codeausschnitt zeigt einen Befehl zum Hinzufügen von zwei vom Benutzer angegebenen Parametern mit der Bezeichnung `retryAttempt`

und `uri`. Im Beispiel stammt der Wert dieser Parameter aus den Argumenten `retryAttempt` und `uriDestination`, die an den Protokollierungsbefehl übergeben wurden.

Example JSON-Protokollierungsbefehl mit zusätzlichen Parametern

```
context.Logger.LogInformation("Starting retry {retryAttempt} to make GET request to
 {uri}", retryAttempt, uriDestination);
```

Die von diesem Befehl ausgegebene Protokollmeldung ist im folgenden JSON-Beispiel dargestellt.

Example JSON-Protokolldatensatz

```
{
    "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**
>
> Sie können bei der Angabe zusätzlicher Parameter auch Positionseigenschaften anstelle von Namen verwenden. Beispielsweise könnte der Protokollierungsbefehl im vorherigen Beispiel auch wie folgt geschrieben werden:
>
> ```
> context.Logger.LogInformation("Starting retry {0} to make GET request to {1}",
>  retryAttempt, uriDestination);
> ```

Beachten Sie, dass Lambda bei der Angabe zusätzlicher Protokollierungsparameter diese als Top-Level-Eigenschaften im JSON-Protokollsatz erfasst. Dieser Ansatz unterscheidet sich von einigen gängigen .NET-Protokollierungsbibliotheken, wie z. B. `Serilog`, die zusätzliche Parameter in einem separaten untergeordneten Objekt erfassen.

Wenn das Argument, das Sie für einen zusätzlichen Parameter angeben, ein komplexes Objekt ist, verwendet Lambda standardmäßig die `ToString()`-Methode, um den Wert bereitzustellen. Um anzugeben, dass ein Argument JSON-serialisiert werden soll, verwenden Sie das Präfix `@`, wie im

folgenden Codeschnipsel gezeigt. In diesem Beispiel ist `User` ein Objekt mit den Eigenschaften `FirstName` und `LastName`.

Example JSON-Protokollierungsbefehl mit serialisiertem JSON-Objekt

```
context.Logger.LogInformation("User {@user} logged in", User);
```

Die von diesem Befehl ausgegebene Protokollmeldung ist im folgenden JSON-Beispiel dargestellt.

Example JSON-Protokolldatensatz

```
{
    "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"
    }
}
```

Wenn das Argument für einen zusätzlichen Parameter ein Array ist oder `IList` oder `IDictionary` implementiert, fügt Lambda das Argument der JSON-Protokollnachricht als Array hinzu, wie im folgenden Beispiel-JSON-Protokollsatz gezeigt. In diesem Beispiel nimmt `{users}` ein `IList`-Argument entgegen, das Instanzen der `User`-Eigenschaft mit demselben Format wie im vorherigen Beispiel enthält. Lambda wandelt dieses `IList` in ein Array um, wobei jeder Wert mit der `ToString`-Methode erstellt wird.

Example JSON-Protokolldatensatz mit einem **`IList`**-Argument

```
{
    "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"
    ]
}
```

Sie können die Liste auch mit JSON serialisieren, indem Sie das @-Präfix in Ihrem Protokollierungsbefehl verwenden. Im folgenden JSON-Protokolldatensatz ist die Eigenschaft `users` in JSON serialisiert.

Example JSON-Protokolldatensatz mit einem serialisierten JSON-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":
    [
        {
            "FirstName": "Alejandro",
            "LastName": "Rosalez"
        },
        {
            "FirstName": "John",
            "LastName": "Stiles"
        }
    ]
}
```

Verwenden der Filterung auf Protokollebene mit .NET

Durch die Konfiguration der Filterung auf Protokollebene können Sie festlegen, dass nur Protokolle mit einer bestimmten Detailebene oder niedriger an Logs gesendet werden. CloudWatch Informationen zur Konfiguration der Filterung auf Protokollebene für Ihre Funktion finden Sie unter the section called “Filterung auf Protokollebene”.

AWS Lambda Um Ihre Protokollnachrichten nach Protokollebene zu filtern, können Sie entweder Protokolle im JSON-Format oder die `Console` .NET-Methoden zur Ausgabe von Protokollnachrichten verwenden. Um Protokolle im JSON-Format zu erstellen, konfigurieren Sie den Protokolltyp Ihrer Funktion auf JSON und verwenden Sie die `ILambdaLogger`-Instance.

Bei JSON-formatierten Protokollen filtert Lambda Ihre Protokollausgaben anhand des Schlüssel-Wert-Paares „level“ im JSON-Objekt, das in the section called “Verwenden des strukturierten JSON-Protokollformats mit .NET” beschrieben wird.

Wenn Sie die `Console` .NET-Methoden verwenden, um Nachrichten in CloudWatch Logs zu schreiben, wendet Lambda Log-Levels wie folgt auf Ihre Nachrichten an:

- Konsole. WriteLine method - Lambda wendet ein Log-Level von `INFO`
- Console.Error-Methode – Lambda wendet eine Protokollebene von `ERROR` an

Wenn Sie Ihre Funktion so konfigurieren, dass sie Filterung auf Protokollebene verwendet, müssen Sie aus den folgenden Optionen für die Protokollebene auswählen, die Lambda an Logs senden soll. CloudWatch Beachten Sie die Zuordnung der von Lambda verwendeten Protokollebenen zu den Microsoft-Standardstufen, die von .NET `ILambdaLogger` verwendet werden.

Lambda-Protokollebene	Äquivalente Microsoft-Ebene	Standardnutzung
TRACE (am detailliertesten)	Trace	Die detailliertesten Informationen, die verwendet werden, um den Ausführungspfad Ihres Codes nachzuverfolgen
DEBUG	Debuggen	Detaillierte Informationen für das System-Debugging
INFO	Informationen	Meldungen, die den normalen Betrieb Ihrer Funktion erfassen
WARN	Warnung	Meldungen über mögliche Fehler, die zu unerwartetem Verhalten führen können, wenn sie nicht behoben werden
ERROR	Fehler	Meldungen über Probleme, die verhindern, dass der Code wie erwartet funktioniert

Lambda-Protokollebene	Äquivalente Microsoft-Ebene	Standardnutzung
FATAL (am wenigsten Details)	Kritisch	Meldungen über schwerwiegende Fehler, die dazu führen, dass die Anwendung nicht mehr funktioniert

Lambda sendet Protokolle mit der ausgewählten Detailebene und niedriger bis CloudWatch. Wenn Sie beispielsweise die Protokollebene WARN konfigurieren, sendet Lambda Protokolle, die den Stufen WARN, ERROR und FATAL entsprechen.

Zusätzliche Protokollierungswerkzeuge und Bibliotheken

Powertools for AWS Lambda (.NET) ist ein Entwickler-Toolkit zur Implementierung von Best Practices für Serverless und zur Steigerung der Entwicklergeschwindigkeit. Das Logging-Serviceprogramm bietet einen für Lambda optimierten Logger, der zusätzliche Informationen zum Funktionskontext all Ihrer Funktionen enthält, wobei die Ausgabe als JSON strukturiert ist. Mit diesem Serviceprogramm können Sie Folgendes tun:

- Erfassung von Schlüsselfeldern aus dem Lambda-Kontext, Kaltstart und Strukturen der Protokollierungsausgabe als JSON
- Protokollieren Sie Ereignisse von Lambda-Aufrufen, wenn Sie dazu aufgefordert werden (standardmäßig deaktiviert)
- Alle Protokolle nur für einen bestimmten Prozentsatz der Aufrufe über Protokollstichproben drucken (standardmäßig deaktiviert)
- Fügen Sie dem strukturierten Protokoll zu einem beliebigen Zeitpunkt zusätzliche Schlüssel hinzu
- Verwenden Sie einen benutzerdefinierten Protokollformatierer (Bring Your Own Formatter), um Protokolle in einer Struktur auszugeben, die mit dem Logging RFC Ihres Unternehmens kompatibel ist

Verwendung von Powertools für AWS Lambda (.NET) und für strukturiertes Logging AWS SAM

Gehen Sie wie folgt vor, um eine Hello World C#-Beispielanwendung mit integrierten Powertools for AWS Lambda (.NET) -Modulen herunterzuladen, zu erstellen und bereitzustellen. Verwenden

Sie dazu die. AWS SAM Diese Anwendung implementiert ein grundlegendes API-Backend und verwendet Powertools zum Ausgeben von Protokollen, Metriken und Traces. Es besteht aus einem Amazon-API-Gateway-Endpunkt und einer Lambda-Funktion. Wenn Sie eine GET-Anfrage an den API-Gateway-Endpunkt senden, ruft die Lambda-Funktion Logs und Metriken auf, sendet sie im Embedded Metric Format an CloudWatch und sendet Traces an. AWS X-Ray Die Funktion gibt eine `hello world`-Nachricht zurück.

Voraussetzungen

Zur Durchführung der Schritte in diesem Abschnitt benötigen Sie Folgendes:

- .NET 8
- AWS CLI Version 2
- AWS SAM CLI Version 1.75 oder höher. Wenn Sie eine ältere Version der AWS SAM CLI haben, finden Sie weitere Informationen unter Upgrade der AWS SAM CLI.

Stellen Sie eine AWS SAM Beispielanwendung bereit

1. Initialisieren Sie die Anwendung mithilfe der Hello TypeScript World-Vorlage.

```
sam init --app-template hello-world-powertools-dotnet --name sam-app --package-type
 Zip --runtime dotnet6 --no-tracing
```

2. Entwickeln Sie die App.

```
cd sam-app && sam build
```

3. Stellen Sie die Anwendung bereit.

```
sam deploy --guided
```

4. Folgen Sie den Anweisungen auf dem Bildschirm. Um die im interaktiven Erlebnis bereitgestellten Standardoptionen zu akzeptieren, drücken Sie `Enter`.

> **Note**
>
> Für ist HelloWorldFunction möglicherweise keine Autorisierung definiert. Ist das in Ordnung? , stellen Sie sicher, dass Sie eintreteny.

5. Rufen Sie die URL der bereitgestellten Anwendung ab:

```
aws cloudformation describe-stacks --stack-name sam-app --query
 'Stacks[0].Outputs[?OutputKey==`HelloWorldApi`].OutputValue' --output text
```

6. Rufen Sie den API-Endpunkt auf:

```
curl -X GET <URL_FROM_PREVIOUS_STEP>
```

   Wenn der Link erfolgreich ausgeführt wurde, sehen Sie die folgende Antwort:

```
{"message":"hello world"}
```

7. Führen Sie sam logs aus, um die Protokolle für die Funktion abzurufen. Weitere Informationen finden Sie unter Arbeiten mit Protokollen im AWS Serverless Application Model - Entwicklerhandbuch.

```
sam logs --stack-name sam-app
```

   Das Ergebnis sieht folgendermaßen aus:

```
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. Dies ist ein öffentlicher API-Endpunkt, der über das Internet zugänglich ist. Es wird empfohlen, dass Sie den Endpunkt nach dem Testen löschen.

```
sam delete
```

Verwalten der Protokollaufbewahrung

Wenn Sie eine Funktion löschen, werden Protokollgruppen nicht automatisch gelöscht. Um zu vermeiden, dass Protokolle auf unbestimmte Zeit gespeichert werden, löschen Sie die Protokollgruppe oder konfigurieren Sie einen Aufbewahrungszeitraum, nach dessen Ablauf die Protokolle CloudWatch automatisch gelöscht werden. Um die Aufbewahrung von Protokollen einzurichten, fügen Sie Ihrer AWS SAM Vorlage Folgendes hinzu:

```
Resources:
  HelloWorldFunction:
    Type: AWS::Serverless::Function
    Properties:
      # Omitting other properties

  LogGroup:
    Type: AWS::Logs::LogGroup
    Properties:
      LogGroupName: !Sub "/aws/lambda/${HelloWorldFunction}"
      RetentionInDays: 7
```

Anzeigen von Protokollen in der Lambda-Konsole

Sie können die Lambda-Konsole verwenden, um die Protokollausgabe nach dem Aufrufen einer Lambda-Funktion anzuzeigen.

Wenn Ihr Code über den eingebetteten Code-Editor getestet werden kann, finden Sie Protokolle in den Ausführungsergebnissen. Wenn Sie das Feature Konsolentest verwenden, um eine Funktion aufzurufen, finden Sie die Protokollausgabe im Abschnitt Details.

Protokolle in der CloudWatch Konsole anzeigen

Sie können die CloudWatch Amazon-Konsole verwenden, um Protokolle für alle Lambda-Funktionsaufrufe anzuzeigen.

Um Protokolle auf der Konsole anzuzeigen CloudWatch

1. Öffnen Sie die Seite Protokollgruppen auf der CloudWatch Konsole.
2. Wählen Sie die Protokollgruppe für Ihre Funktion (***your-function-name***/aws/lambda/).
3. Wählen Sie eine Protokollstream aus.

Jeder Protokoll-Stream entspricht einer Instance Ihrer Funktion. Ein Protokollstream wird angezeigt, wenn Sie Ihre Lambda-Funktion aktualisieren und wenn zusätzliche Instanzen für gleichzeitige Aufrufe erstellt werden. Um Logs für einen bestimmten Aufruf zu finden, empfehlen wir, Ihre Funktion mit zu instrumentieren. AWS X-Ray X-Ray erfasst Details zu der Anforderung und dem Protokollstream in der Trace.

Logs mit dem Befehl () anzeigen AWS Command Line InterfaceAWS CLI

Das AWS CLI ist ein Open-Source-Tool, mit dem Sie mithilfe von Befehlen in Ihrer Befehlszeilen-Shell mit AWS Diensten interagieren können. Zur Durchführung der Schritte in diesem Abschnitt benötigen Sie AWS CLI Version 2.

Sie können die AWS CLI verwenden, um Protokolle für einen Aufruf mit der `--log-type`-Befehlsoption abzurufen. Die Antwort enthält das Feld `LogResult`, das bis zu 4 KB base64-verschlüsselte Protokolle aus dem Aufruf enthält.

Example eine Log-ID abrufen

Das folgende Beispiel zeigt, wie eine Protokoll-ID aus dem `LogResult`-Feld für eine Funktion namens `my-function` abgerufen wird.

```
aws lambda invoke --function-name my-function out --log-type Tail
```

Die Ausgabe sollte folgendermaßen aussehen:

```
{
    "StatusCode": 200,
    "LogResult":
 "U1RBUlQgUmVxdWVzdElkOiA4N2QwNDRiOC1mMTU0LTExZTgtOGNkYS0yOTc0YzVlNGZiMjEgVmVyc2lvb...",
    "ExecutedVersion": "$LATEST"
}
```

Example entschlüsseln der Protokolle

Verwenden Sie in derselben Eingabeaufforderung das `base64`-Dienstprogramm, um die Protokolle zu entschlüsseln. Das folgende Beispiel zeigt, wie Base64-codierte Logs für abgerufen werde `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
```

Die cli-binary-format Option ist erforderlich, wenn Sie AWS CLI Version 2 verwenden. Um dies zur Standardeinstellung zu machen, führen Sie `aws configure set cli-binary-format raw-in-base64-out` aus. Weitere Informationen finden Sie unter Von AWS CLI unterstützte globale Befehlszeilenoptionen im AWS Command Line Interface -Benutzerhandbuch für Version 2.

Die Ausgabe sollte folgendermaßen aussehen:

```
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
```

Das `base64`-Dienstprogramm ist unter Linux, macOS und Ubuntu auf Windowsverfügbar. macOS-Benutzer müssen möglicherweise `base64 -D` verwenden.

Example get-logs.sh-Skript

Verwenden Sie in derselben Eingabeaufforderung das folgende Skript, um die letzten fünf Protokollereignisse herunterzuladen. Das Skript verwendet `sed` zum Entfernen von Anführungszeichen aus der Ausgabedatei und wechselt 15 Sekunden lang in den Ruhezustand, um Zeit einzuräumen, damit Protokolle verfügbar werden können. Die Ausgabe enthält die Antwort von Lambda und die `get-log-events`Ausgabe des Befehls.

Kopieren Sie den Inhalt des folgenden Codebeispiels und speichern Sie es in Ihrem Lambda-Projektverzeichnis unter `get-logs.sh`.

Die cli-binary-format Option ist erforderlich, wenn Sie AWS CLI Version 2 verwenden. Um dies zur Standardeinstellung zu machen, führen Sie `aws configure set cli-binary-format raw-in-base64-out` aus. Weitere Informationen finden Sie unter Von AWS CLI unterstützte globale Befehlszeilenoptionen im AWS Command Line Interface -Benutzerhandbuch für 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 und Linux (nur diese Systeme)

In derselben Eingabeaufforderung müssen macOS- und Linux-Benutzer möglicherweise den folgenden Befehl ausführen, um sicherzustellen, dass das Skript ausführbar ist.

```
chmod -R 755 get-logs.sh
```

Example die letzten fünf Protokollereignisse abrufen

Führen Sie an derselben Eingabeaufforderung das folgende Skript aus, um die letzten fünf Protokollereignisse abzurufen.

```
./get-logs.sh
```

Die Ausgabe sollte folgendermaßen aussehen:

```
{
```

```
    "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"
}
```

Löschen von Protokollen

Wenn Sie eine Funktion löschen, werden Protokollgruppen nicht automatisch gelöscht. Um das unbegrenzte Speichern von Protokollen zu vermeiden, löschen Sie die Protokollgruppe oder konfigurieren Sie eine Aufbewahrungszeitraum nach dem Protokolle automatisch gelöscht werden.

Instrumentieren von C# -Code in AWS Lambda

Lambda ist mit AWS X-Ray integriert, um Ihnen zu helfen, Lambda-Anwendungen zu verfolgen, zu debuggen und zu optimieren. Sie können mit X-Ray eine Anforderung verfolgen, während sie Ressourcen in Ihrer Anwendung durchläuft, die Lambda-Funktionen und andere AWS-Services enthalten können.

Um Protokollierungsdaten an X-Ray zu senden, können Sie eine von drei SDK-Bibliotheken verwenden:

- AWS Distro for OpenTelemetry (ADOT) – Eine sichere, produktionsbereite, AWS-unterstützte Verteilung des OpenElemetrie (OTel)-SDK.
- AWS X-Ray SDK für .NET – Ein SDK zum Generieren und Senden von Nachverfolgungsdaten an X-Ray.
- Powertools für AWS Lambda (.NET) – Ein Entwickler-Toolkit zur Implementierung bewährter Methoden für Serverless und zur Steigerung der Entwicklergeschwindigkeit.

Jedes der SDKs bietet Möglichkeiten, Ihre Telemetriedaten an den X-Ray Service zu senden. Sie können dann mit X-Ray die Leistungsmetriken Ihrer Anwendung anzeigen, filtern und erhalten, um Probleme und Möglichkeiten zur Optimierung zu identifizieren.

> **Important**
>
> X-Ray und Powertools für AWS Lambda-SDKs sind Teil einer eng integrierten Instrumentierungslösung von AWS. Die ADOT Lambda Layers sind Teil eines branchenweiten Standards für die Verfolgung von Instrumenten, die im Allgemeinen mehr Daten erfassen, aber möglicherweise nicht für alle Anwendungsfälle geeignet sind. Sie können mit beiden Lösungen eine durchgehende Ablaufverfolgung in X-Ray implementieren. Weitere Informationen zur Auswahl zwischen ihnen finden Sie unter Auswählen zwischen der AWS-Distro für Open Telemetry und X-Ray-SDKs.

Abschnitte

- Verwenden von Powertools für AWS Lambda (.NET) und AWS SAM für die Nachverfolgung
- Instrumentierung Ihrer .NET-Funktionen mithilfe von X-Ray-SDK
- Aktivieren der Nachverfolgung mit der Lambda-Konsole

- Aktivieren der Nachverfolgung mit der Lambda-API
- Aktivieren von Nachverfolgungen mit CloudFormation
- Interpretieren einer X-Ray-Nachverfolgung

Verwenden von Powertools für AWS Lambda (.NET) und AWS SAM für die Nachverfolgung

Führen Sie die nachstehenden Schritte aus, um eine Hello-World-Beispielanwendung für C# mit integrierten Modulen von Powertools für AWS Lambda (.NET) unter Verwendung von AWS SAM bereitzustellen. Diese Anwendung implementiert ein grundlegendes API-Backend und verwendet Powertools zum Ausgeben von Protokollen, Metriken und Traces. Es besteht aus einem Amazon-API-Gateway-Endpunkt und einer Lambda-Funktion. Wenn Sie eine GET-Anforderung an den API-Gateway-Endpunkt senden, startet die Lambda-Funktion, sendet Protokolle und Metriken im Embedded Metric Format an CloudWatch und sendet Traces an AWS X-Ray. Die Funktion gibt eine Hello-World-Nachricht zurück.

Voraussetzungen

Zur Durchführung der Schritte in diesem Abschnitt benötigen Sie Folgendes:

- .NET 8
- AWS CLI Version 2
- AWS SAM CLI-Version 1.75 oder höher. Wenn Sie eine ältere Version der AWS SAM-CLI haben, finden Sie weitere Informationen unter Aktualisieren der AWS SAM-CLI.

Bereitstellen einer AWS SAM-Beispielanwendung

1. Initialisieren Sie die Anwendung mit der Hello World TypeScript-Vorlage.

```
sam init --app-template hello-world-powertools-dotnet --name sam-app --package-type
 Zip --runtime dotnet6 --no-tracing
```

2. Entwickeln Sie die App.

```
cd sam-app && sam build
```

3. Stellen Sie die Anwendung bereit.

```
sam deploy --guided
```

4. Folgen Sie den Anweisungen auf dem Bildschirm. Um die im interaktiven Erlebnis bereitgestellten Standardoptionen zu akzeptieren, drücken Sie `Enter`.

> **Note**
>
> Für HelloWorldFunction ist möglicherweise keine Autorisierung definiert. Ist das in Ordnung? , stellen Sie sicher, dass Sie y eingeben.

5. Rufen Sie die URL der bereitgestellten Anwendung ab:

```
aws cloudformation describe-stacks --stack-name sam-app --query
 'Stacks[0].Outputs[?OutputKey==`HelloWorldApi`].OutputValue' --output text
```

6. Rufen Sie den API-Endpunkt auf:

```
curl <URL_FROM_PREVIOUS_STEP>
```

Wenn der Link erfolgreich ausgeführt wurde, sehen Sie die folgende Antwort:

```
{"message":"hello world"}
```

7. Führen Sie sam traces aus, um die Traces für die Funktion zu erhalten.

```
sam traces
```

Das Nachverfolgungsergebnis sieht folgendermaßen aus:

```
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. Dies ist ein öffentlicher API-Endpunkt, der über das Internet zugänglich ist. Es wird empfohlen, dass Sie den Endpunkt nach dem Testen löschen.

```
sam delete
```

X-Ray verfolgt nicht alle Anfragen an Ihre Anwendung nach. X-Ray wendet einen Sampling-Algorithmus an, um sicherzustellen, dass die Nachverfolgung effizient ist, und stellt dennoch ein repräsentatives Beispiel aller Anfragen bereit. Die Samplingrate beträgt 1 Anforderung pro Sekunde und 5 Prozent aller weiteren Anforderungen. Sie können die X-Ray-Samplingrate nicht für Ihre Funktionen konfigurieren.

Instrumentierung Ihrer .NET-Funktionen mithilfe von X-Ray-SDK

Sie können Ihren Funktionscode instrumentieren, um Metadaten aufzuzeichnen und Downstream-Aufrufe zu verfolgen. Um Details zu Aufrufen aufzuzeichnen, die Ihre Funktion an andere Ressourcen und Dienste vornimmt, verwenden Sie das AWS X-Ray SDK für .NET. Um das SDK zu erhalten, fügen Sie die `AWSXRayRecorder`-Pakete Ihrer Projektdatei hinzu.

```
<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>
```

Es gibt eine Reihe von Nuget-Paketen, die Auto-Instrumentierung für AWS SDKs, Entity Framework und HTTP-Anfragen bieten. Den vollständigen Satz der Konfigurationsoptionen finden Sie unter AWS X-Ray SDK for .NET im AWS X-Ray-Entwicklerhandbuch.

Nachdem Sie die gewünschten Nuget-Pakete hinzugefügt haben, konfigurieren Sie die automatische Instrumentierung. Es empfiehlt sich, diese Konfiguration außerhalb der Handler-Funktion Ihrer Funktion durchzuführen. So können Sie die Wiederverwendung der Ausführungsumgebung nutzen, um die Leistung Ihrer Funktion zu verbessern. Im folgenden Codebeispiel wird die `RegisterXRayForAllServices`-Methode im Funktionskonstruktor aufgerufen, um die Instrumentierung für alle AWS-SDK-Aufrufe hinzuzufügen.

```
[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)
        };
    }
}
```

Aktivieren der Nachverfolgung mit der Lambda-Konsole

Gehen Sie folgendermaßen vor, um die aktive Nachverfolgung Ihrer Lambda-Funktion mit der Konsole umzuschalten:

So aktivieren Sie die aktive Nachverfolgung

1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.
2. Wählen Sie eine Funktion aus.

3. Wählen Sie Configuration (Konfiguration) und dann Monitoring and operations tools (Überwachungs- und Produktionstools).
4. Wählen Sie unter Zusätzliche Überwachungstools die Option Bearbeiten aus.
5. Wählen Sie unter CloudWatch Application Signals und AWS X-Ray für Lambda-Service-Nachverfolgungen die Option Aktivieren aus.
6. Wählen Sie Speichern.

Aktivieren der Nachverfolgung mit der Lambda-API

Konfigurieren Sie die Nachverfolgung für Ihre Lambda-Funktion mit der AWS CLI oder dem AWS-SDK. Verwenden Sie die folgenden API-Operationen:

- UpdateFunctionConfiguration
- GetFunctionConfiguration
- CreateFunction

Der folgende AWS CLI-Beispielbefehl aktiviert die aktive Nachverfolgung für eine Funktion mit dem Namen my-function.

```
aws lambda update-function-configuration --function-name my-function \
--tracing-config Mode=Active
```

Der Ablaufverfolgungsmodus ist Teil der versionsspezifischen Konfiguration, wenn Sie eine Version Ihrer Funktion veröffentlichen. Sie können den Ablaufverfolgungsmodus für eine veröffentlichte Version nicht ändern.

Aktivieren von Nachverfolgungen mit CloudFormation

Verwenden Sie die `AWS::Lambda::Function`-Eigenschaft, um die Nachverfolgung für eine CloudFormation-Ressource in einer `TracingConfig`-Vorlage zu aktivieren.

Example function-inline.yml – Ablaufverfolgungskonfiguration

```
Resources:
  function:
    Type: AWS::Lambda::Function
```

```
    Properties:
      TracingConfig:
        Mode: Active
      ...
```

Verwenden Sie für eine AWS Serverless Application Model (AWS SAM) `AWS::Serverless::Function`-Ressource die Eigenschaft `Tracing`.

Example template.yml – Ablaufverfolgungskonfiguration

```
Resources:
  function:
    Type: AWS::Serverless::Function
    Properties:
      Tracing: Active
      ...
```

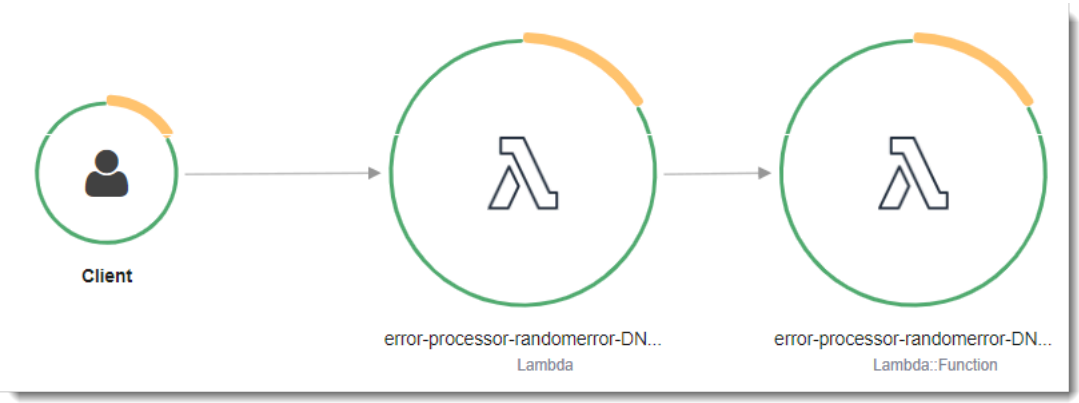
Interpretieren einer X-Ray-Nachverfolgung

Ihre Funktion benötigt die Berechtigung zum Hochladen von Trace-Daten zu X-Ray. Wenn Sie die aktive Nachverfolgung in der Lambda-Konsole aktivieren, fügt Lambda der Ausführungsrolle Ihrer Funktion die erforderlichen Berechtigungen hinzu. Andernfalls fügen Sie der Ausführungsrolle die AWSXRayDaemonWriteAccess-Richtlinie hinzu.

Nachdem Sie die aktive Nachverfolgung konfiguriert haben, können Sie bestimmte Anfragen über Ihre Anwendung beobachten. Das X-Ray-Service-Diagramm zeigt Informationen über Ihre Anwendung und alle ihre Komponenten an. Das folgende Beispiel zeigt eine Anwendung mit zwei Funktionen. Die primäre Funktion verarbeitet Ereignisse und gibt manchmal Fehler zurück. Die zweite Funktion ganz oben verarbeitet Fehler, die in der Protokollgruppe der ersten Protokollgruppe erscheinen, und verwendet das AWS-SDK, um X-Ray, Amazon Simple Storage Service (Amazon S3) und Amazon CloudWatch Logs aufzurufen.

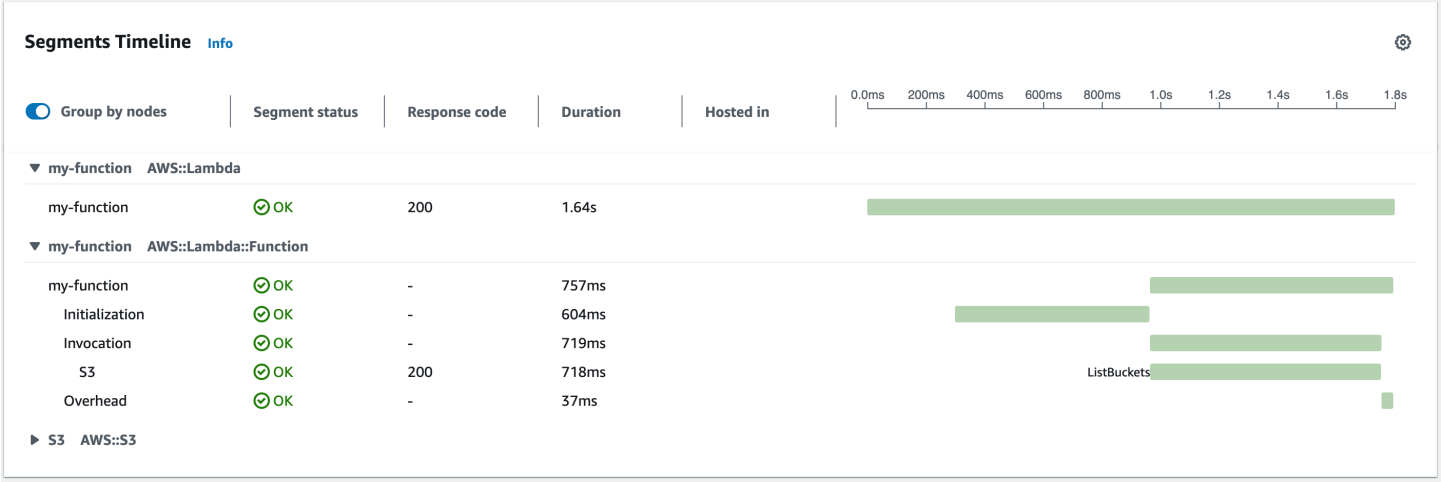
X-Ray verfolgt nicht alle Anfragen an Ihre Anwendung nach. X-Ray wendet einen Sampling-Algorithmus an, um sicherzustellen, dass die Nachverfolgung effizient ist, und stellt dennoch ein repräsentatives Beispiel aller Anfragen bereit. Die Samplingrate beträgt 1 Anforderung pro Sekunde und 5 Prozent aller weiteren Anforderungen. Sie können die X-Ray-Samplingrate nicht für Ihre Funktionen konfigurieren.

In X-Ray, zeichnet eine Ablaufverfolgung Informationen zu einer Anforderung auf, die von einem oder mehreren Services verarbeitet wird. Lambda zeichnet 2 Segmente pro Trace auf, wodurch zwei Knoten im Dienstgraphen entstehen. In der folgenden Abbildung werden diese beiden Knoten hervorgehoben:



Der erste Knoten auf der linken Seite stellt den Lambda-Service dar, der die Aufrufanforderung empfängt. Der zweite Knoten stellt Ihre spezifische Lambda-Funktion dar. Das folgende Beispiel zeigt eine Nachverfolgung mit diesen zwei Segmenten. Beide heißen my-function, aber einer hat einen Ursprung von `AWS::Lambda` und der andere hat einen Ursprung von `AWS::Lambda::Function`. Wenn das `AWS::Lambda`-Segment einen Fehler anzeigt, hatte der Lambda-Service ein Problem.

Wenn das `AWS::Lambda::Function`-Segment einen Fehler anzeigt, ist bei Ihrer Funktion ein Problem aufgetreten.



Dieses Beispiel erweitert das `AWS::Lambda::Function`-Segment, um seine drei Untersegmente anzuzeigen.

> **Note**
>
> AWS implementiert derzeit Änderungen am Lambda-Service. Aufgrund dieser Änderungen können geringfügige Unterschiede in Struktur und Inhalt der Systemprotokollmeldungen und Trace-Segmente auftreten, die von verschiedenen Lambda-Funktionen in Ihrem AWS-Konto. Der hier gezeigte Beispiel-Trace veranschaulicht das Funktionssegment im alten Stil. Die Unterschiede zwischen den Segmenten im alten und im neuen Stil werden in den folgenden Abschnitten beschrieben.
>
> Diese Änderungen werden in den kommenden Wochen implementiert, und alle Funktionen in allen AWS-Regionen-Regionen mit Ausnahme von China und GovCloud werden auf das neue Format der Protokollmeldungen und Trace-Segmente umgestellt.

Das Funktionssegment im alten Stil enthält die folgenden Untersegmente:

- Initialisierung – Stellt die Zeit dar, die für das Laden Ihrer Funktion und das Ausführen des Initialisierungscodes aufgewendet wurde. Dieses Untersegment erscheint nur für das erste Ereignis, das jede Instance Ihrer Funktion verarbeitet.
- Invocation (Aufruf) – Stellt die Zeit dar, die beim Ausführen Ihres Handler-Codes vergeht.
- Overhead (Aufwand) – Stellt die Zeit dar, die von der Lambda-Laufzeitumgebung bei der Verarbeitung des nächsten Ereignisses verbraucht wird.

Das Funktionssegment im neuen Stil enthält kein `Invocation`-Untersegment. Stattdessen werden Kundenuntersegmente direkt an das Funktionssegment angehängt. Weitere Informationen über die Struktur der alten und neuen Funktionssegmente finden Sie unter the section called “Grundlegendes zu X-Ray-Ablaufverfolgungen”.

Sie können auch HTTP-Clients instrumentieren, SQL-Abfragen aufzeichnen und benutzerdefinierte Untersegmente mit Anmerkungen und Metadaten erstellen. Weitere Informationen finden Sie unter AWS X-Ray SDK für .NET im AWS X-Ray-Entwicklerhandbuch.

> **Preisgestaltung**
>
> Sie können X-Ray-Nachverfolgung jeden Monat bis zu einem bestimmten Limit als Teil des AWS kostenlosen Kontingents kostenlos nutzen. Über den Schwellenwert hinaus berechnet X-Ray Gebühren für die Speicherung und den Abruf der Nachverfolgung. Weitere Informationen finden Sie unter AWS X-Ray – Preise.

AWS Lambda-Funktions-Testing in C#

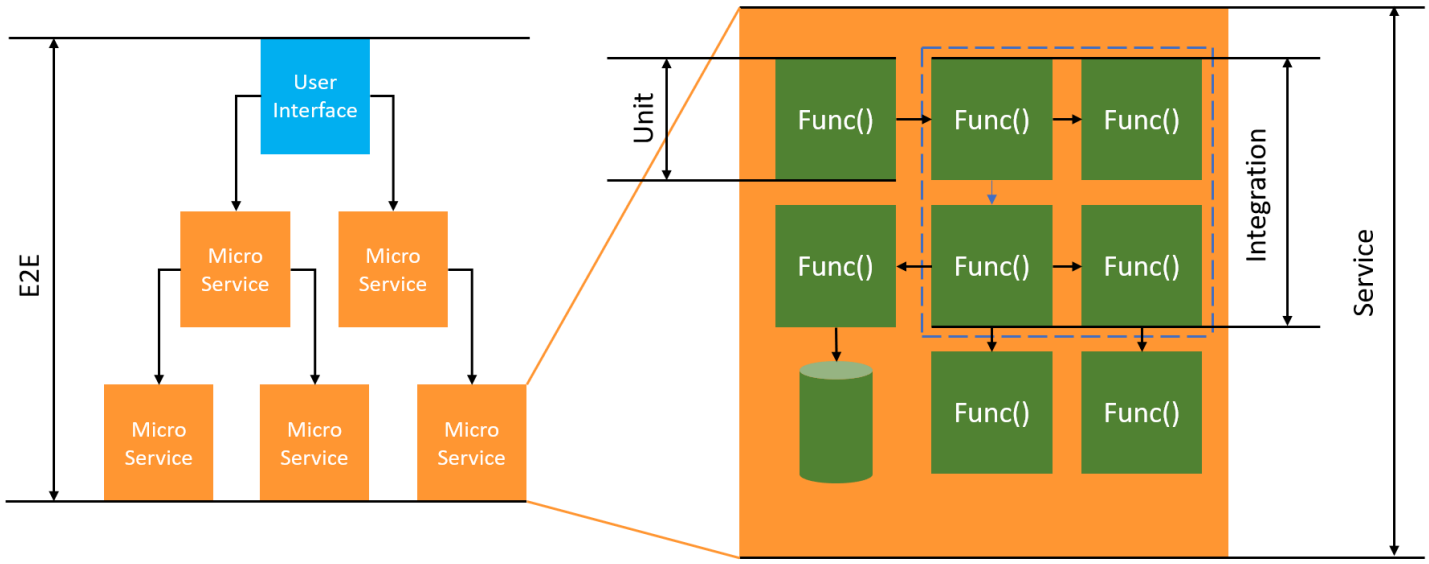
> **Note**
>
> Im Kapitel Testen von Funktionen finden Sie eine vollständige Einführung in Techniken und bewährte Methoden für das Testen von Serverless-Lösungen.

Beim Testen der Serverless-Funktionen werden herkömmliche Testtypen und -techniken verwendet. Erwägen Sie jedoch auch das Testen der Serverless-Anwendungen als Ganzes. Cloud-basierte Tests bieten das genaueste Maß für die Qualität sowohl Ihrer Funktionen als auch Ihrer Serverless-Anwendungen.

Eine Serverless-Anwendungsarchitektur umfasst verwaltete Services, die über API-Aufrufe wichtige Anwendungsfunktionen bereitstellen. Aus diesem Grund muss Ihr Entwicklungszyklus automatisierte Tests beinhalten, die bei der Interaktion Ihrer Funktionen und Services die Funktionalität überprüfen.

Wenn Sie keine cloud-basierten Tests erstellen, können aufgrund von Unterschieden zwischen Ihrer lokalen Umgebung und der bereitgestellten Umgebung Probleme auftreten. Ihr kontinuierlicher Integrationsprozess muss Tests anhand einer Reihe von Ressourcen durchführen, die in der Cloud bereitgestellt werden, bevor Ihr Code in die nächste Bereitstellungsumgebung wie QA, Staging oder Produktion übertragen wird.

Lesen Sie diesen kurzen Leitfaden weiter, um weitere Informationen zu Teststrategien für Serverless-Anwendungen zu erhalten, oder besuchen Sie das Serverless Test Samples Repository, um praktische Beispiele zu finden, die sich speziell auf die gewählte Sprache und Laufzeit beziehen.



Für Serverless-Tests schreiben Sie weiterhin Einheiten-, Integrations- und End-to-End-Tests.

- Einheitentests — Tests, die an einem isolierten Codeblock ausgeführt werden. Zum Beispiel die Überprüfung der Geschäftslogik zur Berechnung der Bereitstellungskosten für einen bestimmten Artikel und Bestimmungsort.
- Integrationstests — Tests, an denen zwei oder mehr Komponenten oder Dienste beteiligt sind, die interagieren, in der Regel in einer Cloud-Umgebung. Bei der Überprüfung einer Funktion werden beispielsweise Ereignisse aus einer Warteschlange verarbeitet.
- End-to-End-Tests — Tests, die das Verhalten einer gesamten Anwendung überprüfen. Stellen Sie beispielsweise sicher, dass die Infrastruktur korrekt eingerichtet ist und die Ereignisse zwischen den Services wie erwartet ablaufen, um die Bestellungen der Kunden aufzuzeichnen.

Testen Ihrer Serverless-Anwendungen

In der Regel verwenden Sie eine Mischung aus verschiedenen Ansätzen, um Ihren Serverless-Anwendungscode zu testen, einschließlich Tests in der Cloud, Tests mit Mock-Code und gelegentlich Tests mit Emulatoren.

Testen in der Cloud

Das Testen in der Cloud ist für alle Testphasen von Nutzen, einschließlich Einheitentests, Integrationstests und End-to-End-Tests. Sie führen Tests für Code durch, der in der Cloud bereitgestellt wird und mit cloud-basierten Services interagiert. Dieser Ansatz bietet das genaueste Maß für die Qualität Ihres Codes.

Eine bequeme Möglichkeit, Ihre Lambda-Funktion in der Cloud zu debuggen, ist die Verwendung der Konsole mit einem Testereignis. Ein Testereignis ist eine JSON-Eingabe für Ihre Funktion. Wenn Ihre Funktion keine Eingabe erfordert, kann das Ereignis ein leeres JSON-Dokument (`{}`) sein. Die Konsole bietet Beispielereignisse für eine Vielzahl von Service-Integrationen. Nachdem Sie ein Ereignis in der Konsole erstellt haben, können Sie es mit Ihrem Team teilen, um das Testen einfacher und einheitlicher zu gestalten.

> **Note**
>
> Das Testen einer Funktion in der Konsole ist ein schneller Einstieg, aber die Automatisierung Ihrer Testzyklen gewährleistet die Anwendungsqualität und die Entwicklungsgeschwindigkeit.

Test-Tools

Um Ihren Entwicklungszyklus zu beschleunigen, gibt es eine Reihe von Tools und Techniken, die Sie beim Testen Ihrer Funktionen verwenden können. AWS SAMAccelerate und AWS CDKWatch Mode reduzieren beispielsweise beide die Zeit, die für die Aktualisierung von Cloud-Umgebungen benötigt wird.

Die Art und Weise, wie Sie Ihren Lambda-Funktionscode definieren, macht es einfach, Komponententests hinzuzufügen. Lambda benötigt einen öffentlichen, parameterlosen Konstruktor, um Ihre Klasse zu initialisieren. Durch die Einführung eines zweiten, internen Konstruktors haben Sie die Kontrolle über die Abhängigkeiten, die Ihre Anwendung verwendet.

```
[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)
        };
    }
}
```

Um einen Test für diese Funktion zu schreiben, können Sie eine neue Instance Ihrer `Function`-Klasse initialisieren und eine simulierte Implementierung von `IDatabaseRepository` übergeben. Die folgenden Beispiele verwenden `XUnit`, `Moq` und `FluentAssertions`, um einen einfachen Test zu schreiben, der sicherstellt, dass `FunctionHandler` einen 200-Statuscode zurückgibt.

```
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);
```

```
    }
}
```

Ausführlichere Beispiele, einschließlich Beispiele für asynchrone Tests, finden Sie im Repository für .NET-Testbeispiele auf GitHub.

Aufbau von Lambda-Funktionen mit PowerShell

In den folgenden Abschnitten wird erklärt, wie gängige Programmiermuster und Kernkonzepte beim Verfassen von Lambda-Funktionscode in PowerShell angewendet werden.

Lambda bietet die folgenden Beispielanwendungen für PowerShell:

- blank-powershell — Eine PowerShell Funktion, die die Verwendung von Logging, Umgebungsvariablen und dem SDK veranschaulicht. AWS

Bevor Sie beginnen, müssen Sie zunächst eine PowerShell Entwicklungsumgebung einrichten. Anweisungen dazu finden Sie unter Einrichten einer PowerShell-Entwicklungsumgebung.

Weitere Informationen zur Verwendung des AWSLambda PSCore Moduls zum Herunterladen von PowerShell Beispielprojekten aus Vorlagen, zum Erstellen von PowerShell Bereitstellungspaketen und zum Bereitstellen von PowerShell Funktionen in der AWS Cloud finden Sie unterPowerShell-Lambda-Funktionen mit .zip-Dateiarchiven bereitstellen.

Lambda stellt die folgenden Laufzeiten für .NET-Sprachen bereit:

Name	ID	Betriebssystem	Datum der Veraltung	Blockfunktion erstellen	Blockfunktion aktualisieren
.NET 10	`dotnet10`	Amazon Linux 2023	14. November 2028	14. Dezember 2028	15. Januar 2029
.NET 9 (nur Container)	`dotnet9`	Amazon Linux 2023	10. November 2026	Nicht geplant	Nicht geplant
.NET 8	`dotnet8`	Amazon Linux 2023	10. November 2026	10. Dezember 2026	11. Januar 2027

Themen

- Einrichten einer PowerShell-Entwicklungsumgebung

- PowerShell-Lambda-Funktionen mit .zip-Dateiarchiven bereitstellen
- Definieren von Lambda-Funktions-Handlern in PowerShell
- Verwenden des Lambda-Kontextobjekts zum Abrufen von PowerShell-Funktionsinformationen
- Powershell-Lambda-Funktionen protokollieren und überwachen

Einrichten einer PowerShell-Entwicklungsumgebung

Lambda bietet eine Reihe von Tools und Bibliotheken für die PowerShell-Laufzeit. Eine Installationsanleitung finden Sie unter Lambda Tools for PowerShell auf GitHub.

Das AWSLambdaPSCore-Modul enthält die folgenden Cmdlets, um das Erstellen und Veröffentlichen von PowerShell-Lambda-Funktionen zu unterstützen:

- Get-AWSPowerShellLambdaTemplate – Gibt eine Liste der Einstiegs-Vorlagen zurück.
- New-AWSPowerShellLambda – Erstellt eine anfängliche PowerShell-Skript basierend auf einer Vorlage.
- Publish-AWSPowerShellLambda – Veröffentlicht ein bestimmtes PowerShell-Skript in Lambda.
- New-AWSPowerShellLambdaPackage – Erstellt ein Lambda-Bereitstellungspaket, das in einem CI/CD-System für die Bereitstellung verwendet werden kann.

PowerShell-Lambda-Funktionen mit .zip-Dateiarchiven bereitstellen

Ein Bereitstellungspaket für die PowerShell-Laufzeit enthält Ihr PowerShell-Skript, PowerShell-Module, die für Ihr PowerShell-Skript erforderlich sind, und die Komponenten, die zum Hosten von PowerShell Core erforderlich sind.

Erstellen der Lambda-Funktion

Um die ersten Schritte zum Schreiben und Aufrufen eines PowerShell-Skripts mit Lambda auszuführen, können Sie das `New-AWSPowerShellLambda`-Cmdlet verwenden, um ein Starter-Skript basierend auf einer Vorlage zu erstellen. Sie können das `Publish-AWSPowerShellLambda`-Cmdlet verwenden, um Ihr Skript in Lambda bereitzustellen. Anschließend können Sie Ihr Skript entweder über die Befehlszeile oder die Lambda-Konsole testen.

Zum Erstellen, Hochladen und Testen eines neuen PowerShell-Skripts gehen Sie wie folgt vor:

1. Führen Sie den folgenden Befehl aus, um die Liste der verfügbaren Vorlagen anzuzeigen:

   ```
   PS C:\> Get-AWSPowerShellLambdaTemplate

   Template               Description
   --------               -----------
   Basic                  Bare bones script
   CodeCommitTrigger      Script to process AWS CodeCommit Triggers
   ...
   ```

2. Führen Sie den folgenden Befehl aus, um ein Beispielskript anhand der `Basic`-Vorlage zu erstellen:

   ```
   New-AWSPowerShellLambda -ScriptName MyFirstPSScript -Template Basic
   ```

   Eine neue Datei mit dem Namen `MyFirstPSScript.ps1` wird in einem neuen Unterverzeichnis des aktuellen Verzeichnisses erstellt. Der Name des Verzeichnisses basiert auf dem `-ScriptName`-Parameter. Sie können den `-Directory`-Parameter verwenden, um ein alternatives Verzeichnis auswählen.

   Sie können sehen, dass die neue Datei folgenden Inhalt hat:

   ```
   # 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. Wenn Sie wissen möchten, wie Log-Nachrichten von Ihrem PowerShell-Skript an Amazon CloudWatch Logs gesendet werden, entfernen Sie die Kommentarzeichen in der `Write-Host`-Zeile des Beispielskripts.

   Um zu veranschaulichen, wie Sie Daten von Ihren Lambda-Funktionen zurückgeben können, fügen Sie mit eine neue Zeile am Ende des Skripts hinzu `$PSVersionTable`. Damit wird `$PSVersionTable` der PowerShell-Pipeline hinzugefügt. Nachdem das PowerShell-Skript abgeschlossen ist, ist das letzte Objekt der PowerShell-Pipeline der Rückgabewert für die Lambda-Funktion. Bei `$PSVersionTable` handelt es sich um eine globale PowerShell-Variable, die außerdem Informationen über die laufende Umgebung bietet.

   Nachdem Sie alle Änderungen vorgenommen haben, sehen die beiden letzten Zeilen des Beispielskripts wie folgt aus:

   ```
   Write-Host (ConvertTo-Json -InputObject $LambdaInput -Compress -Depth 5)
   $PSVersionTable
   ```

4. Nachdem Sie die `MyFirstPSScript.ps1`-Datei bearbeitet haben, ändern Sie das Verzeichnis in den Speicherort des Skripts. Führen Sie anschließend führen Sie den folgenden Befehl aus, um das Skript in Lambda zu veröffentlichen:

   ```
   Publish-AWSPowerShellLambda -ScriptPath .\MyFirstPSScript.ps1 -Name
    MyFirstPSScript -Region us-east-2
   ```

Beachten Sie, dass der `-Name`-Parameter den Lambda-Funktionsnamen angibt, der in der Lambda-Konsole angezeigt wird. Sie können diese Funktion auch verwenden, um Ihr Skript manuell aufzurufen.

5. Rufen Sie Ihre Funktion mit dem AWS Command Line Interface (AWS CLI) `invoke`-Befehl auf.

```
> aws lambda invoke --function-name MyFirstPSScript out
```

Definieren von Lambda-Funktions-Handlern in PowerShell

Beim Aufrufen einer Lambda-Funktion ruft der Lambda-Handler das PowerShell-Skript auf.

Wenn das PowerShell-Skript aufgerufen wird, werden die folgenden Variablen vordefiniert:

- *$LambdaInput* – Ein PSObject mit den Eingabedaten für den Handler. Diese Eingabedaten können (von einer Ereignisquelle veröffentlichte) Ereignisdaten oder beliebige andere Eingabedaten sein, die Sie in Form einer Zeichenfolge oder eines benutzerdefinierten Datenobjekts übergeben.
- *$LambdaContext* – Ein Amazon.Lambda.Core.ILambdaContext-Objekt, das Sie für den Zugriff auf Informationen über den aktuellen Aufrufverwenden können, wie etwa den Namen der aktuellen Funktion, das Speicherlimit, die verbleibende Ausführungszeit und die Protokollierung.

Betrachten Sie beispielsweise das folgende PowerShell-Codebeispiel.

```
#Requires -Modules @{ModuleName='AWSPowerShell.NetCore';ModuleVersion='3.3.618.0'}
Write-Host 'Function Name:' $LambdaContext.FunctionName
```

Dieses Skript gibt die FunctionName-Eigenschaft zurück, das es von der $LambdaContext-Variable USD erhält.

> **Note**
>
> Sie müssen die `#Requires`-Anweisung innerhalb Ihres PowerShell-Skripts verwenden, um die Module anzugeben, von denen Ihre Skripts abhängen. Diese Anweisung erfüllt zwei wichtige Aufgaben. 1) Sie teilt anderen Entwicklern mit, welche Module das Skript verwendet und 2) identifiziert die abhängigen Module, die AWS PowerShell-Tools mit dem Skript im Rahmen der Bereitstellung verpacken müssen. Weitere Informationen über die `#Requires`-Anweisung in PowerShell finden Sie unter About Requires. Weitere Informationen zu PowerShell Bereitstellungspaketen finden Sie unter PowerShell-Lambda-Funktionen mit .zip-Dateiarchiven bereitstellen.
>
> Wenn Ihre PowerShell Lambda-Funktion AWS PowerShell-Cmdlets verwendet, dann stellen Sie sicher, dass Sie eine `AWSPowerShell.NetCore`-Anweisung festlegen, die auf das `#Requires`-Modul verweist. Dieses Modul unterstützt PowerShell Core und nicht das `AWSPowerShell`-Modul, das nur Windows PowerShell unterstützt. Stellen Sie zudem sicher, dass Sie `AWSPowerShell.NetCore` Version 3.3.270.0 oder höher verwenden, da diese

den Cmdlet-Importvorgang optimiert. Mit älteren Versionen dauern Kaltstarts länger. Weitere Informationen finden Sie unter AWS-Tools for PowerShell.

Zurückgeben von Daten

Einige Lambda-Aufrufe sollen Daten zurück an ihren Aufrufer zurückgeben. Beispiel: Wenn ein Aufruf als Reaktion auf eine Webanforderung aus API Gateway erfolgt ist, muss unsere Lambda-Funktion die Antwort zurückgeben. Für PowerShell Lambda ist das letzte Objekt, das der PowerShell-Pipeline hinzugefügt wird, die Rückgabedaten aus dem Lambda-Aufruf. Wenn es sich bei dem Objekt um eine Zeichenfolge handelt, werden die Daten im Ist-Zustand zurückgegeben. Andernfalls wird das Objekt mithilfe des `ConvertTo-Json`-Cmdlets in JSON konvertiert.

Betrachten Sie dazu die folgende Beispiel PowerShell-Anweisung, mit der `$PSVersionTable` der PowerShell-Pipeline hinzugefügt wird:

```
$PSVersionTable
```

Nachdem das PowerShell-Skript beendet ist, ist das letzte Objekt der PowerShell-Pipeline der Rückgabewert für die Lambda-Funktion. Bei `$PSVersionTable` handelt es sich um eine globale PowerShell-Variable, die außerdem Informationen über die laufende Umgebung bietet.

Verwenden des Lambda-Kontextobjekts zum Abrufen von PowerShell-Funktionsinformationen

Wenn Lambda Ihre Funktion ausführt, werden Context-Informationen übergeben, indem eine `$LambdaContext`-Variable für den Handler verfügbar gemacht wird. Diese Variable stellt Methoden und Eigenschaften mit Informationen zum Aufruf, zur Funktion und zur Ausführungsumgebung bereit.

Context-Eigenschaften

- `FunctionName` – Der Name der Lambda-Funktion.
- `FunctionVersion` – Die Version der Funktion.
- `InvokedFunctionArn` – Der Amazon-Ressourcenname (ARN), der zum Aufrufen der Funktion verwendet wird. Gibt an, ob der Aufrufer eine Versionsnummer oder einen Alias angegeben hat.
- `MemoryLimitInMB` – Die Menge an Arbeitsspeicher, die der Funktion zugewiesen ist.
- `AwsRequestId` – Der Bezeichner der Aufrufanforderung.
- `LogGroupName` – Protokollgruppe für die Funktion.
- `LogStreamName` – Der Protokollstrom für die Funktionsinstance.
- `RemainingTime` – Die Anzahl der Millisekunden, die vor der Zeitüberschreitung der Ausführung verbleiben.
- `Identity` – Informationen zur Amazon-Cognito-Identität, die die Anforderung autorisiert hat.
- `ClientContext` – (mobile Apps) Clientkontext, der Lambda von der Clientanwendung bereitgestellt wird.
- `Logger` – Das Logger-Objekt für die Funktion.

Der folgende PowerShell-Codeausschnitt zeigt eine einfache Handler-Funktion zum Drucken einiger der Kontextinformationen.

```
#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
```

Powershell-Lambda-Funktionen protokollieren und überwachen

AWS Lambda überwacht automatisch Lambda-Funktionen in Ihrem Namen und sendet Protokolle an Amazon CloudWatch. Ihre Lambda-Funktion enthält eine CloudWatch Logs-Log-Gruppe und einen Log-Stream für jede Instanz Ihrer Funktion. Die Lambda-Laufzeitumgebung sendet Details zu den einzelnen Aufrufen an den Protokollstream und leitet Protokolle und andere Ausgaben aus dem Code Ihrer Funktion weiter. Weitere Informationen finden Sie unter Senden von Lambda-Funktionsprotokollen an CloudWatch Logs.

Auf dieser Seite wird beschrieben, wie Sie eine Protokollausgabe aus dem Code Ihrer Lambda-Funktion erstellen und mithilfe der AWS Command Line Interface Lambda-Konsole oder der CloudWatch Konsole auf Logs zugreifen.

Sections

- Erstellen einer Funktion, die Protokolle zurückgibt
- Anzeigen von Protokollen in der Lambda-Konsole
- Protokolle in der Konsole anzeigen CloudWatch
- Logs mit dem Befehl () anzeigen AWS Command Line InterfaceAWS CLI
- Löschen von Protokollen

Erstellen einer Funktion, die Protokolle zurückgibt

Um Protokolle aus Ihrem Funktionscode auszugeben, können Sie Cmdlets auf Microsoft verwenden. PowerShell.Utility oder ein beliebiges Protokollierungsmodul, das in oder schreibt. `stdout` `stderr` Im folgenden Beispiel wird verwende `Write-Host`.

Example function/Handler.ps1 – Protokollierung

```
#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 Protokollformat

```
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
```

Die .NET-Laufzeit protokolliert die Zeilen START, END und REPORT für jeden Aufruf. Die Berichtszeile enthält die folgenden Details.

Datenfelder für REPORT-Zeilen

- RequestId— Die eindeutige Anforderungs-ID für den Aufruf.

- Dauer – Die Zeit, die die Handler-Methode Ihrer Funktion mit der Verarbeitung des Ereignisses verbracht hat.
- Fakturierte Dauer – Die für den Aufruf fakturierte Zeit.
- Speichergröße – Die der Funktion zugewiesene Speichermenge.
- Max. verwendeter Speicher – Die Speichermenge, die von der Funktion verwendet wird. Wenn Aufrufe eine Ausführungsumgebung gemeinsam nutzen, meldet Lambda den maximalen Speicherverbrauch für alle Aufrufe. Dieses Verhalten kann zu einem höheren als erwarteten gemeldeten Wert führen.
- Initialisierungsdauer – Für die erste Anfrage die Zeit, die zur Laufzeit zum Laden der Funktion und Ausführen von Code außerhalb der Handler-Methode benötigt wurde.
- XRAY TraceId — Für verfolgte Anfragen die AWS X-Ray Trace-ID.
- SegmentId— Für verfolgte Anfragen die X-Ray-Segment-ID.
- Stichprobe – Bei verfolgten Anforderungen das Stichprobenergebnis.

Anzeigen von Protokollen in der Lambda-Konsole

Sie können die Lambda-Konsole verwenden, um die Protokollausgabe nach dem Aufrufen einer Lambda-Funktion anzuzeigen.

Wenn Ihr Code über den eingebetteten Code-Editor getestet werden kann, finden Sie Protokolle in den Ausführungsergebnissen. Wenn Sie das Feature Konsolentest verwenden, um eine Funktion aufzurufen, finden Sie die Protokollausgabe im Abschnitt Details.

Protokolle in der Konsole anzeigen CloudWatch

Sie können die CloudWatch Amazon-Konsole verwenden, um Protokolle für alle Lambda-Funktionsaufrufe anzuzeigen.

Um Protokolle auf der Konsole anzuzeigen CloudWatch

1. Öffnen Sie die Seite Protokollgruppen auf der CloudWatch Konsole.
2. Wählen Sie die Protokollgruppe für Ihre Funktion (***your-function-name***/aws/lambda/).
3. Wählen Sie eine Protokollstream aus.

Jeder Protokoll-Stream entspricht einer Instance Ihrer Funktion. Ein Protokollstream wird angezeigt, wenn Sie Ihre Lambda-Funktion aktualisieren und wenn zusätzliche Instanzen für gleichzeitige

Aufrufe erstellt werden. Um Protokolle für einen bestimmten Aufruf zu finden, empfehlen wir, Ihre Funktion mit zu instrumentieren. AWS X-Ray X-Ray erfasst Details zu der Anforderung und dem Protokollstream in der Trace.

Logs mit dem Befehl () anzeigen AWS Command Line InterfaceAWS CLI

Das AWS CLI ist ein Open-Source-Tool, mit dem Sie mithilfe von Befehlen in Ihrer Befehlszeilen-Shell mit AWS Diensten interagieren können. Zur Durchführung der Schritte in diesem Abschnitt benötigen Sie AWS CLI Version 2.

Sie können die AWS CLI verwenden, um Protokolle für einen Aufruf mit der `--log-type`-Befehlsoption abzurufen. Die Antwort enthält das Feld `LogResult`, das bis zu 4 KB base64-verschlüsselte Protokolle aus dem Aufruf enthält.

Example eine Log-ID abrufen

Das folgende Beispiel zeigt, wie eine Protokoll-ID aus dem `LogResult`-Feld für eine Funktion namens `my-function` abgerufen wird.

```
aws lambda invoke --function-name my-function out --log-type Tail
```

Die Ausgabe sollte folgendermaßen aussehen:

```
{
    "StatusCode": 200,
    "LogResult":
 "U1RBUlQgUmVxdWVzdElkOiA4N2QwNDRiOC1mMTU0LTExZTgtOGNkYS0yOTc0YzVlNGZiMjEgVmVyc2lvb...",
    "ExecutedVersion": "$LATEST"
}
```

Example entschlüsseln der Protokolle

Verwenden Sie in derselben Eingabeaufforderung das `base64`-Dienstprogramm, um die Protokolle zu entschlüsseln. Das folgende Beispiel zeigt, wie Base64-codierte Logs für abgerufen werde `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
```

Die cli-binary-format Option ist erforderlich, wenn Sie AWS CLI Version 2 verwenden. Um dies zur Standardeinstellung zu machen, führen Sie `aws configure set cli-binary-format raw-in-base64-out` aus. Weitere Informationen finden Sie unter Von AWS CLI unterstützte globale Befehlszeilenoptionen im AWS Command Line Interface -Benutzerhandbuch für Version 2.

Die Ausgabe sollte folgendermaßen aussehen:

```
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
```

Das `base64`-Dienstprogramm ist unter Linux, macOS und Ubuntu auf Windowsverfügbar. macOS-Benutzer müssen möglicherweise `base64 -D` verwenden.

Example get-logs.sh-Skript

Verwenden Sie in derselben Eingabeaufforderung das folgende Skript, um die letzten fünf Protokollereignisse herunterzuladen. Das Skript verwendet `sed` zum Entfernen von Anführungszeichen aus der Ausgabedatei und wechselt 15 Sekunden lang in den Ruhezustand, um Zeit einzuräumen, damit Protokolle verfügbar werden können. Die Ausgabe enthält die Antwort von Lambda und die `get-log-events`Ausgabe des Befehls.

Kopieren Sie den Inhalt des folgenden Codebeispiels und speichern Sie es in Ihrem Lambda-Projektverzeichnis unter `get-logs.sh`.

Die cli-binary-format Option ist erforderlich, wenn Sie AWS CLI Version 2 verwenden. Um dies zur Standardeinstellung zu machen, führen Sie `aws configure set cli-binary-format raw-in-base64-out` aus. Weitere Informationen finden Sie unter Von AWS CLI unterstützte globale Befehlszeilenoptionen im AWS Command Line Interface -Benutzerhandbuch für 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 und Linux (nur diese Systeme)

In derselben Eingabeaufforderung müssen macOS- und Linux-Benutzer möglicherweise den folgenden Befehl ausführen, um sicherzustellen, dass das Skript ausführbar ist.

```
chmod -R 755 get-logs.sh
```

Example die letzten fünf Protokollereignisse abrufen

Führen Sie an derselben Eingabeaufforderung das folgende Skript aus, um die letzten fünf Protokollereignisse abzurufen.

```
./get-logs.sh
```

Die Ausgabe sollte folgendermaßen aussehen:

```
{
    "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"
}
```


Löschen von Protokollen

Wenn Sie eine Funktion löschen, werden Protokollgruppen nicht automatisch gelöscht. Um das unbegrenzte Speichern von Protokollen zu vermeiden, löschen Sie die Protokollgruppe oder konfigurieren Sie eine Aufbewahrungszeitraum nach dem Protokolle automatisch gelöscht werden.

Erstellen von Lambda-Funktionen mit Rust

Da Rust zu nativem Code kompiliert wird, benötigen Sie keine spezielle Laufzeit, um Rust-Code auf Lambda auszuführen. Verwenden Sie stattdessen den Rust-Runtime-Client, um Ihr Projekt lokal zu erstellen, und stellen Sie es dann mithilfe einer reinen Betriebssystemlaufzeit auf Lambda bereit. Wenn Sie eine reine Betriebssystem-Runtime verwenden, hält Lambda das Betriebssystem automatisch mit den neuesten Patches auf dem neuesten Stand.

Tools und Bibliotheken für Rust

- AWS SDK für Rust: Das AWS SDK für Rust bietet Rust APIs die Möglichkeit, mit den Infrastrukturdiensten von Amazon Web Services zu interagieren.
- Rust-Laufzeitclient für Lambda: Der Rust-Runtime-Client macht es einfach, in Rust geschriebene Lambda-Funktionen auszuführen.
- Cargo Lambda: Dies ist eine Open-Source-Erweiterung eines Drittanbieters für das Cargo-Befehlszeilentool, die das Erstellen und Bereitstellen von Rust Lambda-Funktionen vereinfacht.
- Lambda HTTP: Diese Bibliothek bietet einen Wrapper für die Arbeit mit HTTP-Ereignissen.
- Lambda-Erweiterung: Diese Bibliothek bietet Unterstützung für das Schreiben von Lambda-Erweiterungen mit Rust.
- AWS Lambda Ereignisse: Diese Bibliothek bietet Typdefinitionen für gängige Integrationen mit Ereignisquellen.

Beispiele für Lambda-Anwendungen für Rust

- Grundlegende Lambda-Funktion: Eine Rust-Funktion, die zeigt, wie grundlegende Ereignisse verarbeitet werden.
- Lambda-Funktion mit Fehlerbehandlung: Eine Rust-Funktion, die zeigt, wie benutzerdefinierte Rust-Fehler in Lambda behandelt werden.
- Lambda-Funktion mit gemeinsam genutzten Ressourcen: Ein Rust-Projekt, das gemeinsam genutzte Ressourcen initialisiert, bevor die Lambda-Funktion erstellt wird.
- Lambda-HTTP-Ereignisse: Eine Rust-Funktion, die HTTP-Ereignisse verarbeitet.
- Lambda-HTTP-Ereignisse mit CORS-Headern: Eine Rust-Funktion, die Tower verwendet, um CORS-Header zu injizieren.
- Lambda-REST-API: Eine REST-API, die Axum und Diesel verwendet, um eine Verbindung zu einer PostgreSQL-Datenbank herzustellen.

- Serverlose Rust-Demo: Ein Rust-Projekt, das die Verwendung der Rust-Bibliotheken, der Protokollierung, der Umgebungsvariablen und des SDK von Lambda zeigt. AWS
- Basic Lambda-Erweiterung: Eine Rust-Erweiterung, die zeigt, wie grundlegende Erweiterungsereignisse verarbeitet werden.
- Lambda-Protokolle und Erweiterung für Amazon Data Firehose: Eine Rust-Erweiterung, die das Senden von Lambda-Protokollierung an Firehose aufzeigt.

Themen

- Definieren von Lambda-Funktionshandlern in Rust
- Verwenden des Lambda-Kontextobjekts zum Abrufen von Rust-Funktionsinformationen
- Verarbeitung von HTTP-Ereignissen mit Rust
- Bereitstellen von Lambda-Rust-Funktionen mit ZIP-Dateiarchiven
- Arbeiten mit Ebenen für Rust-Lambda-Funktionen
- Rust-Lambda-Funktionen protokollieren und überwachen

Definieren von Lambda-Funktionshandlern in Rust

Der Lambda-Funktionshandler ist die Methode in Ihrem Funktionscode, die Ereignisse verarbeitet. Wenn Ihre Funktion aufgerufen wird, führt Lambda die Handler-Methode aus. Ihre Funktion wird so lange ausgeführt, bis der Handler eine Antwort zurückgibt, beendet wird oder ein Timeout auftritt.

Auf dieser Seite wird beschrieben, wie Sie mit Lambda-Funktionshandlern in Rust arbeiten, einschließlich der Projektinitialisierung, Benennungskonventionen und Best Practices. Diese Seite enthält auch ein Beispiel für eine Rust-Lambda-Funktion, die Informationen über eine Bestellung aufnimmt, eine Textdatei als Quittung erstellt und diese Datei in einen Amazon Simple Storage Service (S3)-Bucket stellt. Weitere Informationen darüber, wie Sie Ihre Funktion nach dem Schreiben einsetzen können, finden Sie unter the section called “Bereitstellen von ZIP-Dateiarchiven”.

Themen

- Einrichten Ihres Rust-Handler-Projekts
- Beispiel für Rust-Lambda-Funktionscode
- Gültige Klassendefinitionen für Rust-Handler
- Namenskonventionen für Handler
- Definieren Sie das Eingabeereignisobjekt und greifen Sie darauf zu
- Zugreifen auf und Verwenden des Lambda-Kontextobjekts
- Verwenden Sie den AWS SDK für Rust in Ihrem Handler
- Zugriff auf Umgebungsvariablen
- Geteilten Zustand verwenden
- Bewährte Codemethoden für Rust-Lambda-Funktionen

Einrichten Ihres Rust-Handler-Projekts

Wenn Sie mit Lambda-Funktionen in Rust arbeiten, umfasst der Prozess das Schreiben Ihres Codes, das Kompilieren und das Bereitstellen der kompilierten Artefakte in Lambda. Der einfachste Weg, ein Lambda-Handler-Projekt in Rust einzurichten, ist die Verwendung von AWS Lambda Runtime for Rust. Trotz ihres Namens ist AWS Lambda Runtime for Rust keine verwaltete Laufzeit im gleichen Sinne wie Lambda für Python, Java oder Node.js. Stattdessen ist die AWS Lambda Runtime for Rust eine crate (`lambda_runtime`), die das Schreiben von Lambda-Funktionen in Rust und die Anbindung an die Ausführungsumgebung AWS Lambda unterstützt.

Verwenden Sie den folgenden Befehl, um Cargo Lambda zu installieren, eine Open-Source-Erweiterung eines Drittanbieters für das Cargo-Befehlszeilentool, die das Erstellen und Bereitstellen von Rust Lambda-Funktionen vereinfacht:

```
cargo install cargo-lambda
```

Verwenden Sie nach der erfolgreichen Installation von `cargo-lambda` den folgenden Befehl, um ein neues Rust-Lambda-Funktions-Handler-Projekt zu initialisieren:

```
cargo lambda new example-rust
```

Wenn Sie diesen Befehl ausführen, stellt Ihnen die Befehlszeilenschnittstelle (CLI) einige Fragen zu Ihrer Lambda-Funktion:

- HTTP-Funktion: Wenn Sie beabsichtigen, Ihre Funktion über API Gateway oder eine Funktions-URL aufzurufen, antworten Sie mit Ja. Andernfalls antworten Sie mit Nein. Im Beispielcode auf dieser Seite rufen wir unsere Funktion mit einem benutzerdefinierten JSON-Ereignis auf und wählen daher Nein.
- Ereignistyp: Wenn Sie eine vordefinierte Ereignisform verwenden möchten, um Ihre Funktion aufzurufen, wählen Sie den richtigen erwarteten Ereignistyp aus. Andernfalls lassen Sie diese Option leer. Im Beispielcode auf dieser Seite rufen wir unsere Funktion mit einem benutzerdefinierten JSON-Ereignis auf, daher lassen wir diese Option leer.

Nachdem der Befehl erfolgreich ausgeführt wurde, geben Sie das Hauptverzeichnis Ihres Projekts ein:

```
cd example-rust
```

Mit diesem Befehl werden eine `generic_handler.rs`- und eine `main.rs`-Datei im `src`-Verzeichnis generiert. `generic_handler.rs` kann verwendet werden, um einen generischen Ereignis-Handler anzupassen. Die `main.rs`-Datei enthält Ihre Hauptanwendungslogik. Die `Cargo.toml`-Datei enthält Metadaten über Ihr Paket und listet dessen externe Abhängigkeiten auf.

Beispiel für Rust-Lambda-Funktionscode

Das folgende Beispiel für einen Rust-Lambda-Funktionscode nimmt Informationen über eine Bestellung auf, erstellt eine Textdateiquittung und platziert diese Datei in einem Amazon-S3-Bucket.

Example **`main.rs`**-Lambda-Funktion

```
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
}
```

Diese `main.rs`-Datei enthält die folgenden Abschnitte des Codes:

- `use`-Anweisungen: Damit können Sie Rust-Crates und -Methoden importieren, die für Ihre Lambda-Funktion erforderlich sind.
- `#[derive(Deserialize, Serialize)]`: Definieren Sie die Form des erwarteten Eingabeereignisses in dieser Rust-Struktur.
- `async fn function_handler(event: LambdaEvent<Value>) -> Result<String, Error>`: Dies ist die Haupthandler-Methode, die Ihre Hauptanwendungslogik enthält.
- `async fn upload_receipt_to_s3 (...)`: Dies ist eine Hilfsmethode, auf die von der Haupt-`function_handler`-Methode verwiesen wird.
- `#[tokio::main]`: Dies ist ein Makro, das den Einstiegspunkt eines Rust-Programms markiert. Außerdem wird eine Tokio-Laufzeit eingerichtet, die es Ihrer `main()`-Methode ermöglicht, `async`/`await` asynchron zu verwenden und auszuführen.
- `async fn main() -> Result<(), Error>`: Die `main()`-Funktion ist der Einstiegspunkt Ihres Codes. Darin spezifizieren wir `function_handler` als Haupt-Handler-Methode.

Beispiel für eine Cargo.toml-Datei

Die folgende `Cargo.toml`-Datei gehört zu dieser Funktion.

```
[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"] }
```

Damit diese Funktion ordnungsgemäß funktioniert, muss ihre Ausführungsrolle die `s3:PutObject`-Aktion zulassen. Stellen Sie außerdem sicher, dass Sie die `RECEIPT_BUCKET`-Umgebungsvariable definieren. Nach einem erfolgreichen Aufruf sollte der Amazon-S3-Bucket eine Empfangsdatei enthalten.

Gültige Klassendefinitionen für Rust-Handler

In den meisten Fällen haben Lambda-Handler-Signaturen, die Sie in Rust definieren, das folgende Format:

```
async fn function_handler(event: LambdaEvent<T>) -> Result<U, Error>
```

Für diesen Handler:

- Der Name dieses Handlers ist `function_handler`.
- Der einzige Eingabewert für den Handler ist „event“ und hat den Typ `LambdaEvent<T>`.
  - `LambdaEvent` ist ein Wrapper, die aus der `lambda_runtime`-Crate stammt. Durch die Verwendung dieses Wrappers erhalten Sie Zugriff auf das Kontextobjekt, das Lambda-spezifische Metadaten wie die Anforderungs-ID des Aufrufs enthält.
  - T ist der deserialisierte Ereignistyp. Dies kann beispielsweise `serde_json::Value` sein, wodurch der Handler jede generische JSON-Eingabe entgegennehmen kann. Alternativ kann dies ein Typ wie `ApiGatewayProxyRequest` sein, wenn Ihre Funktion einen bestimmten, vordefinierten Eingabetyp erwartet.
- Der Rückgabetyp des Handlers ist `Result<U, Error>`.
  - U ist der deserialisierte Ausgabetyp. U muss das `serde::Serialize`-Merkmal implementieren, damit Lambda den Rückgabewert in JSON konvertieren kann. Beispielsweise kann U ein

einfacher Typ wie `String`, `serde_json::Value` oder eine benutzerdefinierte Struktur sein, solange es `Serialize` implementiert. Wenn Ihr Code eine Ok(U)-Anweisung erreicht, bedeutet dies, dass die Ausführung erfolgreich war, und Ihre Funktion gibt einen Wert vom Typ `U` zurück.

- Wenn Ihr Code auf einen Fehler stößt (d. h.`Err(Error)`), protokolliert Ihre Funktion den Fehler in Amazon CloudWatch und gibt eine Fehlerantwort vom Typ zurück`Error`.

In unserem Beispiel sieht die Handler-Signatur wie folgt aus:

```
async fn function_handler(event: LambdaEvent<Value>) -> Result<String, Error>
```

Andere gültige Handler-Signaturen können Folgendes enthalten:

- Weglassen des `LambdaEvent`-Wrappers: Wenn Sie `LambdaEvent` weglassen, verlieren Sie den Zugriff auf das Lambda-Kontextobjekt innerhalb Ihrer Funktion. Es folgt ein Beispiel für diese Art von Signatur:

  ```
  async fn handler(event: serde_json::Value) -> Result<String, Error>
  ```

- Verwendung des Einheitentyps als Eingabe: Bei Rust können Sie den Einheitentyp verwenden, um eine leere Eingabe darzustellen. Dies wird häufig für Funktionen mit periodischen, geplanten Aufrufen verwendet. Es folgt ein Beispiel für diese Art von Signatur:

  ```
  async fn handler(_: ()) -> Result<Value, Error>
  ```

Namenskonventionen für Handler

Lambda-Handler in Rust haben keine strengen Benennungsbeschränkungen. Sie können zwar einen beliebigen Namen für Ihren Handler verwenden, Funktionsnamen in Rust werden aber im Allgemeinen in `snake_case` angegeben.

Für kleinere Anwendungen wie in diesem Beispiel können Sie eine einzige `main.rs`-Datei verwenden, die Ihren gesamten Code enthält. Bei größeren Projekten sollte `main.rs` den Einstiegspunkt zu Ihrer Funktion enthalten, aber Sie können zusätzliche Dateien dafür haben, die Ihren Code in logische Module unterteilen. Beispielsweise könnte Ihre Dateistruktur wie folgt aussehen:

```
/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
```

Definieren Sie das Eingabeereignisobjekt und greifen Sie darauf zu

JSON ist das gebräuchlichste und standardmäßigste Eingabeformat für Lambda-Funktionen. In diesem Beispiel erwartet die Funktion eine Eingabe ähnlich der folgenden:

```
{
    "order_id": "12345",
    "amount": 199.99,
    "item": "Wireless Headphones"
}
```

In Rust können Sie die Form des erwarteten Eingabeereignisses in einer Struktur definieren. In diesem Beispiel definieren wir die folgende Struktur, die einen `Order` darstellt:

```
#[derive(Deserialize, Serialize)]
struct Order {
    order_id: String,
    amount: f64,
    item: String,
}
```

Diese Struktur entspricht der erwarteten Eingabeform. In diesem Beispiel generiert das `#[derive(Deserialize, Serialize)]`-Makro automatisch Code für die Serialisierung und Deserialisierung. Das bedeutet, dass wir den generischen Eingabe-JSON-Typ mithilfe der `serde_json::from_value()`-Methode in unsere Struktur deserialisieren können. Dies wird in den ersten Zeilen des Handlers veranschaulicht:

```
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)?;
    ...
```

```
}
```

Sie können dann auf die Felder des Objekts zugreifen. Zum Beispiel holt `order.order_id` den Wert von `order_id` aus der ursprünglichen Eingabe.

Vordefinierte Eingabe-Ereignistypen

In der `aws_lambda_events`-Crate sind viele vordefinierte Eingabe-Ereignistypen verfügbar. Wenn Sie beispielsweise beabsichtigen, Ihre Funktion mit API Gateway aufzurufen, einschließlich des folgenden Imports:

```
use aws_lambda_events::event::apigw::ApiGatewayProxyRequest;
```

Stellen Sie dann sicher, dass Ihr Haupt-Handler die folgende Signatur verwendet:

```
async fn handler(event: LambdaEvent<ApiGatewayProxyRequest>) -> Result<String, Error> {
    let body = event.payload.body.unwrap_or_default();
    ...
}
```

Weitere Informationen zu anderen vordefinierten Eingabe-Ereignistypen finden Sie in der Crate aws_lambda_events.

Zugreifen auf und Verwenden des Lambda-Kontextobjekts

Das Lambda-Kontextobjekt enthält Informationen über Aufruf, Funktion und Ausführungsumgebung. In Rust enthält der `LambdaEvent`-Wrapper das Kontextobjekt. Beispielsweise können Sie das Kontextobjekt verwenden, um die Anforderungs-ID des aktuellen Aufrufs mit dem folgenden Code abzurufen:

```
async fn function_handler(event: LambdaEvent<Value>) -> Result<String, Error> {
    let request_id = event.context.request_id;
    ...
}
```

Weitere Informationen über das Kontextobjekt finden Sie unter the section called “Kontext”.

Verwenden Sie den AWS SDK für Rust in Ihrem Handler

Oft verwenden Sie Lambda-Funktionen, um mit anderen AWS Ressourcen zu interagieren oder diese zu aktualisieren. Die einfachste Art, eine Schnittstelle zu diesen Ressourcen herzustellen, ist die Verwendung von AWS SDK für Rust.

Um SDK-Abhängigkeiten zu Ihrer Funktion hinzuzufügen, fügen Sie diese in Ihrer `Cargo.toml`-Datei hinzu. Wir empfehlen, nur die Bibliotheken hinzuzufügen, die Sie für Ihre Funktion benötigen. Im Beispielcode zuvor haben wir `aws_sdk_s3::Client` verwendet. In der `Cargo.toml`-Datei können Sie diese Abhängigkeit hinzufügen, indem Sie die folgende Zeile unter dem `[dependencies]`-Abschnitt hinzufügen:

```
aws-sdk-s3 = "1.78.0"
```

> **Note**
>
> Dies ist möglicherweise nicht die neueste Version. Wählen Sie die passende Version für Ihre Anwendung.

Importieren Sie dann die Abhängigkeiten direkt in Ihren Code:

```
use aws_sdk_s3::{Client, primitives::ByteStream};
```

Der Beispielcode initialisiert dann einen Amazon-S3-Client wie folgt:

```
let config = aws_config::load_defaults(aws_config::BehaviorVersion::latest()).await;
let s3_client = Client::new(&config);
```

Nachdem Sie Ihren SDK-Client initialisiert haben, können Sie ihn für die Interaktion mit anderen AWS Diensten verwenden. Der Beispielcode ruft die Amazon-S3-`PutObject`-API in der Hilfsfunktion `upload_receipt_to_s3` auf.

Zugriff auf Umgebungsvariablen

In Ihrem Handler-Code können Sie mithilfe der `env::var`-Methode auf beliebige Umgebungsvariablen verweisen. In diesem Beispiel verweisen wir mit der folgenden Codezeile auf die definierte `RECEIPT_BUCKET`-Umgebungsvariable:

```
let bucket_name = env::var("RECEIPT_BUCKET")
    .map_err(|_| "RECEIPT_BUCKET environment variable is not set")?;
```

Geteilten Zustand verwenden

Sie können geteilte Variablen deklarieren, die unabhängig vom Handler-Code Ihrer Lambda-Funktion sind. Diese Variablen können Ihnen helfen, Zustandsinformationen während des Init-Phase zu laden, bevor Ihre Funktion Ereignisse empfängt. Beispielsweise können Sie den Code auf dieser Seite so ändern, dass bei der Initialisierung des Amazon-S3-Clients ein gemeinsamer Status verwendet wird, indem Sie die `main`-Funktion und die Handler-Signatur aktualisieren:

```
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
```

Bewährte Codemethoden für Rust-Lambda-Funktionen

Halten Sie sich an die Richtlinien in der folgenden Liste, um beim Erstellen Ihrer Lambda-Funktionen die besten Codierungspraktiken anzuwenden:

- Trennen Sie den Lambda-Handler von Ihrer Core-Logik. Auf diese Weise können Sie eine Funktion zur besseren Prüfbarkeit von Einheiten schaffen.
- Minimieren Sie die Komplexität Ihrer Abhängigkeiten. Ziehen Sie einfachere Frameworks vor, die sich schnell beim Start der Ausführungsumgebung laden lassen.

- Minimieren Sie die Größe Ihres Bereitstellungspakets auf die für die Laufzeit erforderliche Größe. Dadurch verkürzt sich die Zeit, die für das Herunterladen und Entpacken Ihres Bereitstellungspakets vor dem Aufruf benötigt wird.

Nutzen Sie die Wiederverwendung der Ausführungsumgebung zur Verbesserung Ihrer Funktion. Initialisieren Sie SDK-Clients und Datenbankverbindungen außerhalb des Funktions-Handlers und speichern Sie statische Komponenten lokal im `/tmp`-Verzeichnis. Nachfolgende Aufrufe, die von derselben Instance Ihrer Funktion verarbeitet werden, können diese Ressourcen wiederverwenden. Dies spart Kosten durch Reduzierung der Funktionslaufzeit.

Um potenzielle Datenlecks über Aufrufe hinweg zu vermeiden, verwenden Sie die Ausführungsumgebung nicht, um Benutzerdaten, Ereignisse oder andere Informationen mit Sicherheitsauswirkungen zu speichern. Wenn Ihre Funktion auf einem veränderbaren Zustand beruht, der nicht im Speicher innerhalb des Handlers gespeichert werden kann, sollten Sie für jeden Benutzer eine separate Funktion oder separate Versionen einer Funktion erstellen.

Verwenden Sie eine Keep-Alive-Direktive, um dauerhafte Verbindungen zu pflegen. Lambda bereinigt Leerlaufverbindungen im Laufe der Zeit. Der Versuch, eine Leerlaufverbindung beim Aufruf einer Funktion wiederzuverwenden, führt zu einem Verbindungsfehler. Um Ihre persistente Verbindung aufrechtzuerhalten, verwenden Sie die Keep-Alive-Direktive, die Ihrer Laufzeit zugeordnet ist. Ein Beispiel finden Sie unter Wiederverwenden von Verbindungen mit Keep-Alive in Node.js.

Verwenden Sie Umgebungsvariablen um Betriebsparameter an Ihre Funktion zu übergeben. Wenn Sie z. B. Daten in einen Amazon-S3-Bucket schreiben, anstatt den Bucket-Namen, in den Sie schreiben, hartzucodieren, konfigurieren Sie den Bucket-Namen als Umgebungsvariable.

Vermeiden Sie rekursive Aufrufe in Ihrer Lambda-Funktion, bei denen die Funktion sich selbst aufruft oder einen Prozess initiiert, der die Funktion erneut aufrufen kann. Dies kann zu unvorhergesehenen Mengen an Funktionsaufrufen führen und höhere Kosten zur Folge haben. Wenn Sie eine unbeabsichtigte Menge von Aufrufen feststellen, legen Sie die reservierte gleichzeitige Ausführung der Funktion auf `0` fest, um sofort alle Aufrufe der Funktion zu drosseln, während Sie den Code aktualisieren.

Verwenden Sie APIs in Ihrem Lambda-Funktionscode nicht undokumentiert, nicht öffentlich. Für AWS Lambda verwaltete Laufzeiten führt Lambda regelmäßig Sicherheits- und Funktionsupdates für interne Lambda-Laufzeiten durch. APIs Diese internen API-Updates können abwärtsinkompatibel sein, was zu unbeabsichtigten Folgen wie Aufruffehlern führen kann, wenn Ihre Funktion von diesen

nicht öffentlichen Daten abhängig ist. APIs Eine Liste der öffentlich verfügbaren Programme finden Sie in der API-Referenz. APIs

Schreiben Sie idempotenten Code. Das Schreiben idempotenter Code für Ihre Funktionen stellt sicher, dass doppelte Ereignisse auf die gleiche Weise behandelt werden. Ihr Code sollte Ereignisse ordnungsgemäß validieren und doppelte Ereignisse ordnungsgemäß behandeln. Weitere Informationen finden Sie unter Wie mache ich meine Lambda-Funktion idempotent?.

Verwenden des Lambda-Kontextobjekts zum Abrufen von Rust-Funktionsinformationen

Wenn Lambda Ihre Funktion ausführt, fügt es dem, das der Handler empfängt LambdaEvent , ein Kontextobjekt hinzu. Dieses Objekt stellt Eigenschaften mit Informationen zum Aufruf, zur Funktion und zur Ausführungsumgebung bereit.

Context-Eigenschaften

- `request_id`: Die vom Lambda-Service generierte AWS Anforderungs-ID.
- `deadline`: Die Ausführungsfrist für den aktuellen Aufruf in Millisekunden.
- `invoked_function_arn`: Der Amazon-Ressourcenname (ARN) der aufgerufenen Lambda-Funktion.
- `xray_trace_id`: Die AWS X-Ray Trace-ID für den aktuellen Aufruf.
- `client_content`: Das vom SDK für AWS Mobilgeräte gesendete Client-Kontextobjekt. Dieses Feld ist leer, es sei denn, die Funktion wird mit einem AWS mobilen SDK aufgerufen.
- `identity`: Die Amazon-Cognito-Identität, die die Funktion aufgerufen hat. Dieses Feld ist leer, es sei denn, die Aufrufanforderung an das Lambda APIs wurde mit AWS Anmeldeinformationen gestellt, die von Amazon Cognito Cognito-Identitätspools ausgestellt wurden.
- `env_config`: Die Lambda-Funktionskonfiguration aus den lokalen Umgebungsvariablen. Diese Eigenschaft umfasst Informationen wie den Funktionsnamen, die Speicherzuweisung, die Version und die Protokollstreams.

Zugreifen auf Aufrufkontextinformationen

Lambda-Funktionen haben Zugriff auf Metadaten über ihre Umgebung und die Aufrufanforderung. Das `LambaEvent` Objekt, das Ihr Funktionshandler empfängt, enthält die `context`-Metadaten:

```
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
}
```

Verarbeitung von HTTP-Ereignissen mit Rust

Amazon API Gateway APIs, Application Load Balancers und Lambda-Funktionen URLs können HTTP-Ereignisse an Lambda senden. Sie können den aws_lambda_events Crate von crates.io verwenden, um Ereignisse aus diesen Quellen zu verarbeiten.

Example– API Gateway Proxy-Anfrage bearbeiten

Beachten Sie Folgendes:

- `use aws_lambda_events::apigw::{ApiGatewayProxyRequest, ApiGatewayProxyResponse}`: Der aws_lambda_events Crate enthält viele Lambda-Ereignisse. Um die Kompilierungszeit zu verkürzen, verwenden Sie Feature-Flags, um die benötigten Ereignisse zu aktivieren. Beispiel: `aws_lambda_events = { version = "0.8.3", default-features = false, features = ["apigw"] }`.
- `use http::HeaderMap`: Für diesen Import müssen Sie den HTTP Crate zu Ihren Abhängigkeiten hinzufügen.

```
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
```

```
}
```

Der Rust-Laufzeit-Client für Lambda bietet auch eine Abstraktion dieser Ereignistypen, die es Ihnen ermöglicht, mit nativen HTTP-Typen zu arbeiten, unabhängig davon, welcher Service die Ereignisse sendet. Der folgende Code entspricht dem vorherigen Beispiel und funktioniert sofort mit Lambda-Funktion URLs, Application Load Balancers und API Gateway.

> **Note**
>
> Der lambda_http Crate verwendet den darunter liegenden lambda_runtime Crate. Sie müssen `lambda_runtime` nicht separat importieren.

Example– Bearbeitung von HTTP-Anforderungen

```
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
}
```

Ein weiteres Beispiel für die Verwendung `lambda_http` finden Sie im Codebeispiel http-axum im Labs-Repository. AWS GitHub

Beispiel für HTTP-Lambda-Ereignisse für Rust

- Lambda-HTTP-Ereignisse: Eine Rust-Funktion, die HTTP-Ereignisse verarbeitet.
- Lambda-HTTP-Ereignisse mit CORS-Headern: Eine Rust-Funktion, die Tower verwendet, um CORS-Header zu injizieren.

- Lambda-HTTP-Ereignisse mit gemeinsam genutzten Ressourcen: Eine Rust-Funktion, die gemeinsam genutzte Ressourcen verwendet, die initialisiert werden, bevor der Funktionshandler erstellt wird.

Bereitstellen von Lambda-Rust-Funktionen mit ZIP-Dateiarchiven

Auf dieser Seite wird beschrieben, wie Sie Ihre Rust-Funktion kompilieren und dann die kompilierte Binärdatei AWS Lambda mithilfe von Cargo Lambda bereitstellen. Es zeigt auch, wie die kompilierte Binärdatei mit der AWS Command Line Interface und der AWS Serverless Application Model CLI bereitgestellt wird.

Sections

- Voraussetzungen
- Aufbau von Rust-Funktionen auf macOS, Windows oder Linux
- Bereitstellung der Binärdatei der Rust-Funktion mit Cargo Lambda
- Aufrufen Ihrer Rust-Funktion mit Cargo Lambda

Voraussetzungen

- Rust
- AWS CLI Version 2

Aufbau von Rust-Funktionen auf macOS, Windows oder Linux

Die folgenden Schritte zeigen, wie Sie das Projekt für Ihre erste Lambda-Funktion mit Rust erstellen und es mit Cargo Lambda kompilieren, einer Open-Source-Erweiterung eines Drittanbieters für das Cargo-Befehlszeilentool, die das Erstellen und Bereitstellen von Rust-Lambda-Funktionen vereinfacht.

1. Installieren Sie Cargo Lambda, eine Open-Source-Erweiterung eines Drittanbieters für das Cargo-Befehlszeilentool, die das Erstellen und Bereitstellen von Rust Lambda-Funktionen vereinfacht:

   ```
   cargo install cargo-lambda
   ```

   Weitere Installationsoptionen finden Sie unter Installation in der Cargo Lambda-Dokumentation.
2. Erstellen Sie die Paketstruktur. Dieser Befehl erstellt einen grundlegenden Funktionscode in `src/main.rs`. Sie können diesen Code zum Testen verwenden oder ihn durch Ihren eigenen ersetzen.

```
cargo lambda new my-function
```

3. Führen Sie im Stammverzeichnis des Pakets den Unterbefehl build aus, um den Code in Ihrer Funktion zu kompilieren.

   ```
   cargo lambda build --release
   ```

   (Optional) Wenn Sie AWS Graviton2 auf Lambda verwenden möchten, fügen Sie das `--arm64` Flag hinzu, um Ihren Code für ARM zu kompilieren. CPUs

   ```
   cargo lambda build --release --arm64
   ```

4. Bevor Sie Ihre Rust-Funktion bereitstellen, konfigurieren Sie die AWS Anmeldeinformationen auf Ihrem Computer.

   ```
   aws configure
   ```

Bereitstellung der Binärdatei der Rust-Funktion mit Cargo Lambda

Verwenden Sie den Unterbefehl bereitstellen, um die kompilierte Binärdatei für Lambda bereitzustellen. Dieser Befehl erstellt eine Ausführungsrolle und erstellt dann die Lambda-Funktion. Um eine bestehende Ausführungsrolle anzugeben, verwenden Sie die --iam-role Flag.

```
cargo lambda deploy my-function
```

Bereitstellen Ihrer Rust-Funktionsbinärdatei mit dem AWS CLI

Sie können Ihre Binärdatei auch mit dem bereitstellen AWS CLI.

1. Verwenden Sie den Unterbefehl build, um das .zip-Bereitstellungspaket zu erstellen.

   ```
   cargo lambda build --release --output-format zip
   ```

2. Um das .zip-Paket in Lambda bereitzustellen, führen Sie den Befehl create-function aus.

   - Legen Sie für `--runtime` die Option `provided.al2023` fest. Dies ist eine reine Betriebssystem-Laufzeit. Reine Betriebssystemlaufzeiten werden verwendet, um kompilierte Binärdateien und benutzerdefinierte Laufzeiten für Lambda bereitzustellen.

- Für `--role` geben Sie den ARN der Ausführungsrolle an.

```
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
```

Bereitstellen Ihrer Rust-Funktionsbinärdatei mit der AWS SAM CLI

Sie können Ihre Binärdatei auch mit der AWS SAM CLI bereitstellen.

1. Erstellen Sie eine AWS SAM Vorlage mit der Ressourcen- und Eigenschaftsdefinition. Legen Sie für `Runtime` die Option `provided.al2023` fest. Dies ist eine reine Betriebssystem-Laufzeit. Reine Betriebssystemlaufzeiten werden verwendet, um kompilierte Binärdateien und benutzerdefinierte Laufzeiten für Lambda bereitzustellen.

   Weitere Informationen zur Bereitstellung von Lambda-Funktionen mit AWS SAM finden Sie AWS::Serverless::Functionim AWS Serverless Application Model Entwicklerhandbuch.

   Example SAM-Ressourcen- und Eigenschaftsdefinition für eine Rust-Binärdatei

   ```
   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. Verwenden Sie den Unterbefehl build, um die Funktion zu kompilieren.

```
cargo lambda build --release
```

3. Verwenden Sie den Befehl sam deploy, um die Funktion in Lambda bereitzustellen.

```
sam deploy --guided
```

Weitere Informationen zum Erstellen von Rust-Funktionen mit der AWS SAM CLI finden Sie unter Build Rust Lambda functions with Cargo Lambda im AWS Serverless Application Model Developer Guide.

Aufrufen Ihrer Rust-Funktion mit Cargo Lambda

Verwenden Sie den Unterbefehl invoke, um Ihre Funktion mit einer Nutzlast zu testen.

```
cargo lambda invoke --remote --data-ascii '{"command": "Hello world"}' my-function
```

Aufrufen Ihrer Rust-Funktion mit dem AWS CLI

Sie können die Funktion auch verwenden AWS CLI , um die Funktion aufzurufen.

```
aws lambda invoke --function-name my-function --cli-binary-format raw-in-base64-out --
payload '{"command": "Hello world"}' /tmp/out.txt
```

Die cli-binary-format Option ist erforderlich, wenn Sie AWS CLI Version 2 verwenden. Um dies zur Standardeinstellung zu machen, führen Sie `aws configure set cli-binary-format raw-in-base64-out` aus. Weitere Informationen finden Sie unter Von AWS CLI unterstützte globale Befehlszeilenoptionen im AWS Command Line Interface -Benutzerhandbuch für Version 2.

Arbeiten mit Ebenen für Rust-Lambda-Funktionen

Wir empfehlen, keine Ebenen zur Verwaltung von Abhängigkeiten für in Rust geschriebene Lambda-Funktionen zu verwenden. Der Grund dafür ist, dass Lambda-Funktionen in Rust zu einer einzigen ausführbaren Datei kompiliert werden, die Sie Lambda bei der Bereitstellung Ihrer Funktion zur Verfügung stellen. Diese ausführbare Datei enthält Ihren kompilierten Funktionscode zusammen mit all seinen Abhängigkeiten. Die Verwendung von Ebenen verkompliziert nicht nur diesen Vorgang, sondern führt auch zu längeren Kaltstartzeiten, da Ihre Funktionen während der Init-Phase zusätzliche Assemblys manuell in den Speicher laden müssen.

Um externe Abhängigkeiten mit Ihren Rust-Handlern zu verwenden, fügen Sie diese direkt in Ihr Bereitstellungspaket ein. So vereinfachen Sie den Bereitstellungsprozess und profitieren außerdem von den integrierten Optimierungen des Rust-Compilers. Ein Beispiel dafür, wie Sie eine Abhängigkeit wie das AWS SDK für Rust in Ihre Funktion importieren und verwenden, finden Sie unter the section called “Handler”.

Rust-Lambda-Funktionen protokollieren und überwachen

AWS Lambda überwacht automatisch Lambda-Funktionen in Ihrem Namen und sendet Protokolle an Amazon CloudWatch. Ihre Lambda-Funktion enthält eine CloudWatch Logs-Log-Gruppe und einen Log-Stream für jede Instanz Ihrer Funktion. Die Lambda-Laufzeitumgebung sendet Details zu den einzelnen Aufrufen an den Protokollstream und leitet Protokolle und andere Ausgaben aus dem Code Ihrer Funktion weiter. Weitere Informationen finden Sie unter Senden von Lambda-Funktionsprotokollen an CloudWatch Logs. Hinweise zur Konfiguration von Protokollformaten finden Sie unterKonfiguration der JSON- und Klartext-Protokollformate. Auf dieser Seite wird beschrieben, wie Sie Protokollausgaben aus dem Code Ihrer Lambda-Funktion erstellen.

Erstellen einer Funktion, die Protokolle schreibt

Um Protokolle aus Ihrem Funktionscode auszugeben, können Sie jede Protokollierungsfunktion verwenden, die in `stdout` oder `stderr` schreibt, wie z. B. das `println!`-Makro. Das folgende Beispiel verwendet `println!`, um eine Nachricht zu drucken, wenn der Funktionshandler gestartet wird und bevor er beendet wird.

```
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
}
```

Implementierung der erweiterten Protokollierung mit der Tracing Crate

Tracing ist ein Framework zur Instrumentierung von Rust-Programmen zur Erfassung strukturierter, ereignisbasierter Diagnoseinformationen. Dieses Framework bietet Hilfsprogramme zum Anpassen der Logging-Ausgabestufen und -formate, wie zum Beispiel das Erstellen strukturierter JSON-Protokollnachrichten. Um dieses Framework verwenden zu können, müssen Sie einen `subscriber`

initialisieren, bevor Sie den Funktionshandler implementieren. Anschließend können Sie Tracing-Makros wie `debug`, `info` und `error` verwenden, um die gewünschte Protokollierungsebene für jedes Szenario anzugeben.

Example– Verwenden von Tracing Crate

Beachten Sie Folgendes:

- `tracing_subscriber::fmt().json()`: Wenn diese Option enthalten ist, werden die Protokolle in JSON formatiert. Um diese Option verwenden zu können, müssen Sie das `json`-Feature in die `tracing-subscriber`-Abhängigkeit aufnehmen (z. B. `tracing-subscriber = { version = "0.3.11", features = ["json"] }`).
- `#[tracing::instrument(skip(event), fields(req_id = %event.context.request_id))]`: Diese Annotation generiert bei jedem Aufruf des Handlers eine Spanne. Diese Spanne fügt jeder Protokollzeile die Anforderungs-ID hinzu.
- `{ %first_name }`: Dieses Konstrukt fügt das `first_name`-Feld der Protokollzeile hinzu, in der es verwendet wird. Der Wert für dieses Feld entspricht der Variablen mit dem gleichen Namen.

```
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
}
```

Wenn diese Rust-Funktion aufgerufen wird, druckt sie zwei Protokollzeilen, die den folgenden ähneln:

```
{"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"}]}
```

Bewährte Methoden für die Arbeit mit AWS Lambda Funktionen

Im Folgenden finden Sie einige bewährte Methoden für die Verwendung vo AWS Lambda:

Themen

- Funktionscode
- Konfiguration der Funktionen
- Skalierbarkeit von Funktionen
- Metriken und Alarme
- Arbeiten mit Streams
- Bewährte Methoden für die Gewährleistung der Sicherheit

Funktionscode

Nutzen Sie die Wiederverwendung der Ausführungsumgebung zur Verbesserung Ihrer Funktion. Initialisieren Sie SDK-Clients und Datenbankverbindungen außerhalb des Funktions-Handlers und speichern Sie statische Komponenten lokal im `/tmp`-Verzeichnis. Nachfolgende Aufrufe, die von derselben Instance Ihrer Funktion verarbeitet werden, können diese Ressourcen wiederverwenden. Dies spart Kosten durch Reduzierung der Funktionslaufzeit.

Um potenzielle Datenlecks über Aufrufe hinweg zu vermeiden, verwenden Sie die Ausführungsumgebung nicht, um Benutzerdaten, Ereignisse oder andere Informationen mit Sicherheitsauswirkungen zu speichern. Wenn Ihre Funktion auf einem veränderbaren Zustand beruht, der nicht im Speicher innerhalb des Handlers gespeichert werden kann, sollten Sie für jeden Benutzer eine separate Funktion oder separate Versionen einer Funktion erstellen.

Verwenden Sie eine Keep-Alive-Direktive, um dauerhafte Verbindungen zu pflegen. Lambda bereinigt Leerlaufverbindungen im Laufe der Zeit. Der Versuch, eine Leerlaufverbindung beim Aufruf einer Funktion wiederzuverwenden, führt zu einem Verbindungsfehler. Um Ihre persistente Verbindung aufrechtzuerhalten, verwenden Sie die Keep-Alive-Direktive, die Ihrer Laufzeit zugeordnet ist. Ein Beispiel finden Sie unter Wiederverwenden von Verbindungen mit Keep-Alive in Node.js.

Verwenden Sie Umgebungsvariablen um Betriebsparameter an Ihre Funktion zu übergeben. Wenn Sie z. B. Daten in einen Amazon-S3-Bucket schreiben, anstatt den Bucket-Namen, in den Sie schreiben, hartzucodieren, konfigurieren Sie den Bucket-Namen als Umgebungsvariable.

Vermeiden Sie rekursive Aufrufe in Ihrer Lambda-Funktion, bei denen die Funktion sich selbst aufruft oder einen Prozess initiiert, der die Funktion erneut aufrufen kann. Dies kann zu unvorhergesehenen Mengen an Funktionsaufrufen führen und höhere Kosten zur Folge haben. Wenn Sie eine unbeabsichtigte Menge von Aufrufen feststellen, legen Sie die reservierte gleichzeitige Ausführung der Funktion auf `0` fest, um sofort alle Aufrufe der Funktion zu drosseln, während Sie den Code aktualisieren.

Verwenden Sie APIs in Ihrem Lambda-Funktionscode nicht undokumentiert, nicht öffentlich. Für AWS Lambda verwaltete Laufzeiten führt Lambda regelmäßig Sicherheits- und Funktionsupdates für interne Lambda-Laufzeiten durch. APIs Diese internen API-Updates können abwärtsinkompatibel sein, was zu unbeabsichtigten Folgen wie Aufruffehlern führen kann, wenn Ihre Funktion von diesen nicht öffentlichen Daten abhängig ist. APIs Eine Liste der öffentlich verfügbaren Programme finden Sie in der API-Referenz. APIs

Schreiben Sie idempotenten Code. Das Schreiben idempotenter Code für Ihre Funktionen stellt sicher, dass doppelte Ereignisse auf die gleiche Weise behandelt werden. Ihr Code sollte Ereignisse ordnungsgemäß validieren und doppelte Ereignisse ordnungsgemäß behandeln. Weitere Informationen finden Sie unter Wie mache ich meine Lambda-Funktion idempotent?.

> **Note**
>
> Sie können Powertools verwenden, um Funktionen AWS Lambda idempotent zu machen. Weitere Informationen finden Sie unter:
>
> - Python – Dienstprogramm für Idempotenz
> - TypeScript - Hilfsprogramm für Idempotenz
> - Java – Dienstprogramm für Idempotenz
> - .NET – Dienstprogramm für Idempotenz

Bewährte Methoden für sprachspezifischen Code finden Sie in den folgenden Abschnitten:

- Bewährte Codemethoden für Node.js-Lambda-Funktionen
- Bewährte Codepraktiken für TypeScript Lambda-Funktionen

- Bewährte Codemethoden für Python-Lambda-Funktionen
- Bewährte Codemethoden für Ruby-Lambda-Funktionen
- Bewährte Codemethoden für Java-Lambda-Funktionen
- Bewährte Codemethoden für Go-Lambda-Funktionen
- Bewährte Codemethoden für C#-Lambda-Funktionen
- Bewährte Codemethoden für Rust-Lambda-Funktionen

Konfiguration der Funktionen

Die Leistungstests Ihrer Lambda-Funktion sind ein entscheidender Faktor, um sicherzustellen, dass Sie die optimale Speichergrößenkonfiguration auswählen. Jede Erhöhung des Arbeitsspeichers führt zu einer gleichwertigen Erhöhung der CPU-Verfügbarkeit Ihrer Funktion. Der Speicherverbrauch für Ihre Funktion wird pro Aufruf bestimmt und kann in Amazon eingesehen werden. CloudWatch Bei jedem Aufruf erfolgt ein `REPORT:` Eintrag, wie nachfolgend dargestellt:

```
REPORT RequestId: 3604209a-e9a3-11e6-939a-754dd98c7be3 Duration: 12.34 ms Billed
 Duration: 100 ms Memory Size: 128 MB Max Memory Used: 18 MB
```

Durch die Analyse `Max Memory Used:` des Feldes können Sie feststellen, ob Ihre Funktion mehr Speicher benötigt oder ob die Speichergröße Ihrer Funktion überdimensioniert ist.

Um die richtige Speicherkonfiguration für Ihre Funktionen zu finden, empfehlen wir die Verwendung des Open-Source-Projekts AWS Lambda Power Tuning. Weitere Informationen finden Sie unter AWS Lambda Power Tuning on GitHub.

Um die Funktionsleistung zu optimieren, empfehlen wir außerdem den Einsatz von Bibliotheken, die Advanced Vector Extensions 2 (AVX2) nutzen können. Auf diese Weise können Sie anspruchsvolle Workloads verarbeiten, einschließlich Inferencing für Machine Learning, Medienverarbeitung, High Performance Computing (HPC), wissenschaftliche Simulationen und Finanzmodellierung. Weitere Informationen finden Sie unter Schnellere AWS Lambda Funktionen erstellen mit AVX2.

Unterziehen Sie Ihre Lambda-Funktion zur Ermittlung der optimalen Zeitbeschränkung einem Belastungstest. Es sollte analysiert werden, wie lange Ihre Funktion ausgeführt wird, damit Sie Probleme mit einem Dependency-Service besser erkennen können, welche möglicherweise die Gleichzeitigkeit der Funktion über das hinaus erhöhen, was Sie erwarten. Dies ist besonders wichtig, wenn Ihre Lambda-Funktion Netzwerkaufrufe an Ressourcen durchführt, die möglicherweise die

Skalierung von Lambda nicht beherrschen. Weitere Informationen über Lasttests Ihrer Anwendung finden Sie unter Verteilte Lasttests auf AWS.

Verwenden Sie die strengsten Berechtigungen bei der Einrichtung von IAM-Richtlinien. Informieren Sie sich darüber, welche Ressourcen und Vorgänge Ihre Lambda-Funktion benötigt und beschränken Sie die Ausführungsrolle auf diese Berechtigungen. Weitere Informationen finden Sie unter Verwaltung von Berechtigungen in AWS Lambda.

Machen Sie sich mit Lambda-Kontingente vertraut. Nutzlastgröße, Dateideskriptoren und /tmp space werden bei der Ermittlung von Laufzeit-Ressourcen-Limits oft übersehen.

Löschen Sie nicht mehr benötigte Lambda-Funktionen. Auf diese Weise werden die nicht ungenutzten Funktionen nicht unnötigerweise auf die Größenbeschränkung Ihres Bereitstellungspakets angerechnet.

Wenn Sie Amazon Simple Queue Service als Ereignisquelle verwenden, stellen Sie sicher, dass der Wert der erwarteten Aufrufzeit der Funktion nicht größer ist als der Wert der Zeitbeschränkung für die Sichtbarkeit in der Warteschlange. Dies gilt für CreateFunction und für UpdateFunctionConfiguration.

- Im Fall von schlägt CreateFunction AWS Lambda die Funktionserstellung fehl.
- Im Fall von UpdateFunctionConfigurationkönnte dies zu doppelten Aufrufen der Funktion führen.

Skalierbarkeit von Funktionen

Machen Sie sich mit Ihren vor- und nachgelagerten Durchsatzbeschränkungen vertraut. Lambda-Funktionen lassen sich zwar nahtlos abhängig von der Last skalieren, aber vor- und nachgelagerte Abhängigkeiten verfügen möglicherweise nicht über die gleichen Durchsatzfähigkeiten. Wenn Sie die Skalierbarkeit Ihrer Funktion begrenzen müssen, können Sie die reservierte Gleichzeitigkeit für Ihre Funktion konfigurieren.

Integrierte Drosseltoleranz. Wenn bei Ihrer synchronen Funktion eine Drosselung auftritt, weil der Datenverkehr die Skalierungsrate von Lambda überschreitet, können Sie die folgenden Strategien anwenden, um die Drosselungstoleranz zu verbessern:

- Timeouts, Wiederholungen und Backoff mit Jitter verwenden. Durch die Implementierung dieser Strategien werden wiederholte Aufrufe vereinfacht und es wird sichergestellt, dass Lambda innerhalb von Sekunden skaliert werden kann, um die Drosselung durch Endbenutzer zu minimieren.

- Bereitgestellte Parallelität verwenden. Die bereitgestellte Nebenläufigkeit beschreibt die Anzahl der vorinitialisierten Ausführungsumgebungen, die Lambda Ihrer Funktion zuweist. Lambda verarbeitet eingehende Anforderungen mit bereitgestellter Gleichzeitigkeit, sofern verfügbar. Lambda kann Ihre Funktion bei Bedarf auch über die vorgesehene Gleichzeitigkeitseinstellung hinaus skalieren. Durch die Konfiguration der bereitgestellten Parallelität fallen zusätzliche Gebühren für Ihr Konto an. AWS

Metriken und Alarme

Verwenden Sie Verwendung von CloudWatch-Metriken mit Lambda und CloudWatch Alarme, anstatt eine Metrik in Ihrem Lambda-Funktionscode zu erstellen oder zu aktualisieren. Es ist eine viel effizientere Art und Weise, die Zustandsprüfung Ihrer Lambda-Funktionen nachzuverfolgen, so dass Sie Probleme frühzeitig im Entwicklungsprozess erkennen können. Beispielsweise können Sie einen Alarm auf Basis der voraussichtlichen Dauer der Aufurfzeit Ihrer Lambda-Funktion konfigurieren, um Engpässe oder Latenzen zu beheben, die auf Ihren Funktionscode zurückzuführen sind.

Geben Sie benutzerdefinierte Metriken asynchron mithilfe des Embedded Metric Format (EMF) aus. Anstatt synchrone API-Aufrufe zu tätigen CloudWatch, verwenden Sie EMF, um Metriken über die Protokolle Ihrer Funktion auszugeben. Dieser Ansatz reduziert die Latenz und verbessert die Leistung. Das Metrics-Dienstprogramm in Powertools for AWS Lambda verarbeitet die EMF-Formatierung automatisch. Weitere Informationen finden Sie in der AWS Lambda Dokumentation unter Python - TypeScript, Java - oder .NET Metrics-Dienstprogramme in den Powertools. Informationen zur Verwendung von EMF zur Generierung von Protokollen im Metrikformat finden Sie unter Veröffentlichen von Protokollen mit dem eingebetteten metrischen Format im CloudWatch Amazon-Benutzerhandbuch.

Verwenden Sie die strukturierte JSON-Protokollierung für eine bessere Beobachtbarkeit. Die strukturierte Protokollierung erleichtert das Suchen, Filtern und Analysieren der Protokolle Ihrer Funktion. Erwägen Sie die Verwendung des Logger-Dienstprogramms von Powertools für die AWS Lambda automatische Formatierung von Protokollen in JSON. Weitere Informationen finden Sie in der AWS Lambda Dokumentation unter Python - TypeScript, Java - oder .NET Logger-Dienstprogramme in den Powertools.

Nutzen Sie Ihre Protokollierungsbibliothek und AWS Lambda -Metriken und Dimensionen, um Anwendungsfehler (z. B. ERR, ERROR, WARNING usw.) zu erkennen.

Verwenden Sie die AWS -Kostenanomalie-Erkennung, um ungewöhnliche Aktivitäten in Ihrem Konto zu erkennen. Die Kostenanomalie-Erkennung verwendet Machine Learning, um Ihre Kosten

und Ihre Nutzung kontinuierlich zu überwachen und gleichzeitig falsch positive Warnungen zu minimieren. Die Erkennung von Kostenanomalien verwendet Daten von AWS Cost Explorer, was zu einer Verzögerung von bis zu 24 Stunden führt. Daher kann es nach der Nutzung bis zu 24 Stunden dauern, bis eine Anomalie erkannt wird. Um die Kostenanomalie-Erkennung verwenden zu können, müssen Sie sich zunächst bei Cost Explorer anmelden. Anschließend können Sie auf die Kostenanomalie-Erkennung zugreifen.

Arbeiten mit Streams

Testen Sie mit unterschiedlichen Stapel- und Datensatzgrößen, so dass die Abfragefrequenz jeder Ereignisquelle darauf abgestimmt ist, wie schnell Ihre Funktion ihre Aufgabe erledigen kann. Der CreateEventSourceMapping BatchSize Parameter steuert die maximale Anzahl von Datensätzen, die bei jedem Aufruf an Ihre Funktion gesendet werden können. Eine höhere Stapelgröße kann den mit dem Aufruf-Overhead über eine größere Datensatzgruppe hinweg oft effizienter verarbeiten und Ihren Durchsatz erhöhen.

Standardmäßig ruft Lambda Ihre Funktion auf, sobald Datensätze verfügbar sind. Wenn der Batch, den Lambda aus der Ereignisquelle liest, nur einen Datensatz enthält, sendet Lambda nur einen Datensatz an die Funktion. Damit die Funktion nicht mit einer kleinen Anzahl von Datensätzen aufgerufen wird, können Sie die Ereignisquelle anweisen, Datensätze bis zu 5 Minuten lang zu puffern, indem Sie ein Batch-Fenster konfigurieren. Bevor die Funktion aufgerufen wird, liest Lambda so lange Datensätze aus der Ereignisquelle, bis es einen vollständigen Batch erfasst hat, das Batch-Verarbeitungsfenster abläuft oder der Batch die Nutzlastgrenze von 6 MB erreicht. Weitere Informationen finden Sie unter Batching-Verhalten.

> **Warning**
>
> Zuordnung von Lambda-Ereignisquellen verarbeiten jedes Ereignis mindestens einmal und es kann zu einer doppelten Verarbeitung von Datensätzen kommen. Um mögliche Probleme im Zusammenhang mit doppelten Ereignissen zu vermeiden, empfehlen wir Ihnen dringend, Ihren Funktionscode idempotent zu machen. Weitere Informationen finden Sie im Knowledge Center unter Wie mache ich meine Lambda-Funktion idempotent?. AWS

Aktivieren Sie die teilweise Batch-Antwort für die Stream-Verarbeitung. Wenn Sie Batches von Datensätzen aus Streams wie Kinesis oder DynamoDB Streams verarbeiten, aktivieren Sie die teilweise Batch-Antwort, damit Lambda nur die fehlgeschlagenen Datensätze und nicht den gesamten Batch erneut versuchen kann. Dies verbessert die Verarbeitungseffizienz und reduziert unnötige

Wiederholungsvorgänge. Sie können optional das Batch-Hilfsprogramm von Powertools verwenden AWS Lambda , um die Stapelverarbeitungsmuster zu vereinfachen.

> **Note**
>
> Sie können Powertools AWS Lambda für die Stapelverarbeitung verwenden. Weitere Informationen finden Sie unter:
>
> - Python – Batchverarbeitung
> - TypeScript - Stapelverarbeitung
> - Java – Batchverarbeitung
> - .NET – Batchverarbeitung

Erhöhen Sie den Durchsatz bei der Verarbeitung des Kinesis-Streams durch Hinzufügen von Shards. Ein Kinesis Stream besteht aus einem oder mehreren Shards. Die Geschwindigkeit, mit der Lambda Daten von Kinesis lesen kann, skaliert linear mit der Anzahl der Shards. Eine Erhöhung der Anzahl der Shards erhöht direkt die Anzahl der maximalen gleichzeitigen Lambda-Funktionsaufrufe und kann den Durchsatz Ihrer Kinesis-Streamverarbeitung steigern. Weitere Informationen über die Beziehung zwischen Shards und Funktionsaufrufen finden Sie unter the section called “Abfragen und Stapeln von Streams”. Wenn Sie die Anzahl der Shards in einem Kineses-Stream erhöhen, stellen Sie sicher, dass Sie für Ihre Daten einen guten Partitionsschlüssel ausgewählt haben, (siehe Partitionsschlüssel), so dass Bezugsdatensätze auf denselben Shards gespeichert werden und Ihre Daten gut verteilt sind.

Verwenden Sie Amazon CloudWatch on IteratorAge , um festzustellen, ob Ihr Kinesis-Stream verarbeitet wird. Konfigurieren Sie beispielsweise einen CloudWatch Alarm mit einer Maximaleinstellung von 30000 (30 Sekunden).

Bewährte Methoden für die Gewährleistung der Sicherheit

Überwachen Sie Ihre Nutzung von AWS Lambda in Bezug auf bewährte Sicherheitsmethoden mithilfe AWS Security Hub CSPM von. Security Hub CSPM verwendet Sicherheitskontrollen, um Ressourcenkonfigurationen und Sicherheitsstandards zu bewerten und Sie bei der Einhaltung verschiedener Compliance-Frameworks zu unterstützen. Weitere Informationen zur Verwendung von Security Hub CSPM zur Evaluierung von Lambda-Ressourcen finden Sie unter AWS Lambda Kontrollen im AWS Security Hub CSPM Benutzerhandbuch.

Überwachen Sie Lambda-Netzwerkaktivitätsprotokolle mit Amazon GuardDuty Lambda Protection. GuardDuty Lambda-Schutz hilft Ihnen dabei, potenzielle Sicherheitsbedrohungen zu identifizieren, wenn Lambda-Funktionen in Ihrem aufgerufen werden. AWS-Konto Zum Beispiel, wenn eine Ihrer Funktionen eine IP-Adresse abfragt, die mit Aktivitäten im Zusammenhang mit Kryptowährungen verknüpft ist. GuardDuty überwacht die Netzwerkaktivitätsprotokolle, die generiert werden, wenn eine Lambda-Funktion aufgerufen wird. Weitere Informationen finden Sie unter Lambda-Schutz im GuardDuty Amazon-Benutzerhandbuch.

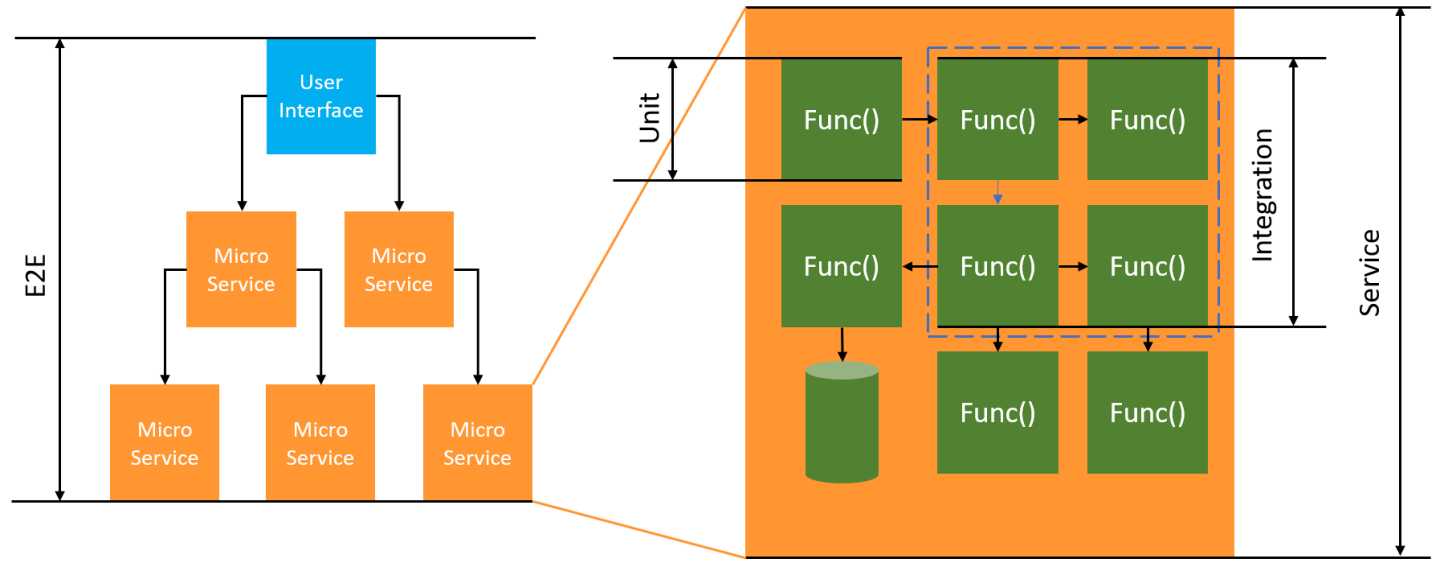
Wie man Serverless-Funktionen und Anwendungen testet

Beim Testen der Serverless-Funktionen werden herkömmliche Testtypen und -techniken verwendet. Erwägen Sie jedoch auch das Testen der Serverless-Anwendungen als Ganzes. Cloud-basierte Tests bieten das genaueste Maß für die Qualität sowohl Ihrer Funktionen als auch Ihrer Serverless-Anwendungen.

Eine Serverless-Anwendungsarchitektur umfasst verwaltete Services, die über API-Aufrufe wichtige Anwendungsfunktionen bereitstellen. Aus diesem Grund muss Ihr Entwicklungszyklus automatisierte Tests beinhalten, die bei der Interaktion Ihrer Funktionen und Services die Funktionalität überprüfen.

Wenn Sie keine cloud-basierten Tests erstellen, können aufgrund von Unterschieden zwischen Ihrer lokalen Umgebung und der bereitgestellten Umgebung Probleme auftreten. Ihr kontinuierlicher Integrationsprozess muss Tests anhand einer Reihe von Ressourcen durchführen, die in der Cloud bereitgestellt werden, bevor Ihr Code in die nächste Bereitstellungsumgebung wie QA, Staging oder Produktion übertragen wird.

Lesen Sie diesen kurzen Leitfaden weiter, um weitere Informationen zu Teststrategien für Serverless-Anwendungen zu erhalten, oder besuchen Sie das Serverless Test Samples Repository, um praktische Beispiele zu finden, die sich speziell auf die gewählte Sprache und Laufzeit beziehen.



Für Serverless-Tests schreiben Sie weiterhin Einheiten-, Integrations- und End-to-End-Tests.

- Einheitentests — Tests, die an einem isolierten Codeblock ausgeführt werden. Zum Beispiel die Überprüfung der Geschäftslogik zur Berechnung der Bereitstellungskosten für einen bestimmten Artikel und Bestimmungsort.
- Integrationstests — Tests, an denen zwei oder mehr Komponenten oder Dienste beteiligt sind, die interagieren, in der Regel in einer Cloud-Umgebung. Bei der Überprüfung einer Funktion werden beispielsweise Ereignisse aus einer Warteschlange verarbeitet.
- End-to-End-Tests — Tests, die das Verhalten einer gesamten Anwendung überprüfen. Stellen Sie beispielsweise sicher, dass die Infrastruktur korrekt eingerichtet ist und die Ereignisse zwischen den Services wie erwartet ablaufen, um die Bestellungen der Kunden aufzuzeichnen.

Gezielte Geschäftsergebnisse

Beim Testen von Serverless-Lösungen kann das Einrichten von Tests, die ereignisgesteuerte Interaktionen zwischen Services überprüfen, etwas mehr Zeit in Anspruch nehmen. Denken Sie beim Lesen dieses Leitfadens an die folgenden praktischen Geschäftsgründe:

- Erhöhen Sie die Qualität Ihrer Anwendung
- Verkürzen Sie die Zeit für das Entwickeln von Features und das Beheben von Fehlern

Die Qualität einer Anwendung hängt davon ab, ob zur Überprüfung der Funktionalität verschiedene Szenarien getestet werden. Durch sorgfältiges Abwägen der Geschäftsszenarien und die Automatisierung dieser Tests für die Ausführung mit Cloud-Services lässt sich die Qualität Ihrer Anwendung erhöhen.

Softwarefehler und Konfigurationsprobleme haben die geringsten Auswirkungen auf Kosten und Zeitplan, wenn sie während eines iterativen Entwicklungszyklus entdeckt werden. Wenn Probleme während der Entwicklung unentdeckt bleiben, erfordert das Erkennen und Beheben in der Produktion mehr Aufwand durch mehr Mitarbeiter.

Eine gut geplante Serverless-Teststrategie erhöht die Softwarequalität und verbessert die Iterationszeit, indem überprüft wird, dass Ihre Lambda-Funktionen und -Anwendungen in einer Cloud-Umgebung wie erwartet funktionieren.

Was muss getestet werden?

Wir empfehlen, eine Teststrategie anzuwenden, die das Verhalten der verwalteten Services, die Cloud-Konfiguration, Sicherheitsrichtlinien und die Integration in Ihren Code testet, um die

Softwarequalität zu verbessern. Verhaltenstests, auch Blackbox-Tests genannt, verifizieren, dass ein System wie erwartet funktioniert, ohne dass alle internen Daten bekannt sind.

- Führen Sie Einheitentests durch, um die Geschäftslogik innerhalb der Lambda-Funktionen zu überprüfen.
- Stellen Sie sicher, dass die integrierten Services tatsächlich aufgerufen werden und die Eingabeparameter korrekt sind.
- Stellen Sie sicher, dass ein Ereignis alle erwarteten Dienste in einem Workflow durchläuft.

In der traditionellen serverbasierten Architektur definieren Teams häufig einen Testumfang, der nur den Code umfasst, der auf dem Anwendungsserver ausgeführt wird. Andere Komponenten, Services oder Abhängigkeiten werden oft als extern betrachtet und liegen außerhalb des Testbereichs.

Serverless-Anwendungen bestehen oft aus kleinen Arbeitseinheiten, wie z. B. Lambda-Funktionen, die Produkte aus einer Datenbank abrufen oder Elemente aus einer Warteschlange verarbeiten oder die Größe eines Image im Speicher ändern. Jede Komponente läuft in ihrer eigenen Umgebung. Teams werden wahrscheinlich für viele dieser kleinen Einheiten innerhalb einer einzigen Anwendung zuständig sein.

Einige Anwendungsfunktionen können vollständig an verwaltete Services wie Amazon S3 delegiert oder ohne Verwendung von intern entwickeltem Code erstellt werden. Das Testen dieser verwalteten Services ist nicht erforderlich. Jedoch müssen Sie die Integration in diese Services testen.

So wird Serverless getestet

Wahrscheinlich sind Sie mit dem Testen lokal bereitgestellter Anwendungen vertraut: Sie schreiben Tests, die gegen Code laufen, der ausschließlich auf Ihrem Desktop-Betriebssystem oder in Containern ausgeführt wird. Beispielsweise können Sie eine lokale Web-Service-Komponente mit einer Anfrage aufrufen und dann Assertionen über die Antwort erstellen.

Serverless-Lösungen werden aus Ihrem Funktionscode und cloudbasierten verwalteten Services wie Warteschlangen, Datenbanken, Ereignisbussen und Messaging-Systemen erstellt. Diese Komponenten sind alle durch eine ereignisgesteuerte Architektur miteinander verbunden, in der Nachrichten, sogenannte Ereignisse, von einer Ressource zur anderen fließen. Diese Interaktionen können synchron sein, z. B. wenn ein Web-Service sofort Ergebnisse zurückgibt, oder es kann sich um eine asynchrone Aktion handeln, die zu einem späteren Zeitpunkt abgeschlossen wird, z. B. das Platzieren von Elementen in eine Warteschlange oder das Starten eines Workflow-Schritts. Ihre Teststrategie muss beide Szenarien beinhalten und die Interaktionen zwischen den Services

testen. Bei asynchronen Interaktionen müssen Sie möglicherweise Nebenwirkungen in Downstream-Komponenten erkennen, die eventuell nicht sofort erkennbar sind.

Die Replikation einer gesamten Cloud-Umgebung, einschließlich Warteschlangen, Datenbanktabellen, Ereignisbussen, Sicherheitsrichtlinien und mehr, ist nicht praktikabel. Aufgrund der Unterschiede zwischen Ihrer lokalen Umgebung und Ihren bereitgestellten Umgebungen in der Cloud werden Sie unweigerlich auf Probleme stoßen. Die Unterschiede zwischen Ihren Umgebungen verlängern die Zeit für die Reproduktion und das Beheben von Fehlern.

Bei Serverless-Anwendungen befinden sich die Architekturkomponenten in der Regel vollständig in der Cloud, sodass Tests mit Code und Services in der Cloud notwendig sind, um Features zu entwickeln und Fehler zu beheben.

Testmethoden

In der Realität wird Ihre Teststrategie wahrscheinlich eine Mischung von Techniken beinhalten, um die Qualität Ihrer Lösungen zu erhöhen. Sie werden schnelle interaktive Tests verwenden, um Funktionen in der Konsole zu debuggen, automatisierte Einheitentests, um isolierte Geschäftslogik zu überprüfen, Verifizierung von Aufrufen externer Services mit Mocks und gelegentliche Tests gegen Emulatoren, die einen Service imitieren.

- the section called “Testen in der Cloud”: Sie stellen Infrastruktur und Code bereit, um mit tatsächlichen Services, Sicherheitsrichtlinien, Konfigurationen und infrastrukturspezifischen Parametern zu testen. Cloud-basierte Tests bieten das genaueste Maß für die Qualität Ihres Codes.

  Das Debuggen einer Funktion in der Konsole ist eine schnelle Möglichkeit, in der Cloud zu testen. Sie können aus einer Bibliothek mit Beispieltestereignissen wählen oder ein benutzerdefiniertes Ereignis erstellen, um eine Funktion isoliert zu testen. Sie können die Testereignisse auch über die Konsole mit Ihrem Team teilen.

  Um das Testen während des Entwicklungs- und Build-Lebenszyklus zu automatisieren, müssen Sie außerhalb der Konsole testen. In den Abschnitten zum sprachspezifischen Testen in diesem Handbuch finden Sie Automatisierungsstrategien und Ressourcen.
- the section called “Testen mit Mocks”: Mocks sind Objekte in Ihrem Code, die einen externen Service simulieren und diesen repräsentieren. Mocks bieten vordefiniertes Verhalten zur Überprüfung von Service-Aufrufen und Parametern. Ein Fake ist eine Scheinimplementierung, die Abkürzungen verwendet, um die Leistung zu vereinfachen oder zu verbessern. Beispielsweise könnte ein gefälschtes Datenzugriffsobjekt Daten aus einem In-Memory-Datenspeicher

zurückgeben. Mocks können komplexe Abhängigkeiten nachahmen und vereinfachen, können aber auch zu weiteren Mocks führen, um verschachtelte Abhängigkeiten zu ersetzen.

- the section called “Lokales Testen mit AWS SAM CLI”: Verwenden Sie die AWS SAM CLI, um Lambda-Funktionen lokal in Docker-Containern aufzurufen, die dieselbe Laufzeitumgebung wie AWS Lambda verwenden. Sie können die Funktionslogik und die Ereignisverarbeitung testen, ohne sie in der Cloud bereitstellen zu müssen.
- the section called “Testen mit Emulation”: Verwenden Sie die LocalStack-Integration in VS Code, um mehrere AWS-Services lokal zu emulieren und so die Serviceintegrationen zu testen.

Testen in der Cloud

Das Testen in der Cloud ist für alle Testphasen von Nutzen, einschließlich Einheitentests, Integrationstests und End-to-End-Tests. Wenn Sie Tests mit cloud-basiertem Code durchführen, der auch mit cloud-basierten Diensten interagiert, erhalten Sie das genaueste Maß für die Qualität Ihres Codes.

Eine bequeme Möglichkeit, eine Lambda-Funktion in der Cloud auszuführen, ist ein Testereignis in der AWS-Managementkonsole. Ein Testereignis ist eine JSON-Eingabe für Ihre Funktion. Wenn Ihre Funktion keine Eingabe erfordert, kann das Ereignis ein leeres JSON-Dokument sein (`{}`). Die Konsole bietet Beispielereignisse für eine Vielzahl von Service-Integrationen. Nachdem Sie ein Ereignis in der Konsole erstellt haben, können Sie es auch mit Ihrem Team teilen, um das Testen einfacher und einheitlicher zu gestalten.

Erfahren Sie, wie Sie eine Beispielfunktion in der Konsole debuggen.

> **Note**
>
> Obwohl das Ausführen von Funktionen in der Konsole eine schnelle Möglichkeit zum Debuggen ist, ist die Automatisierung Ihrer Testzyklen unerlässlich, um die Anwendungsqualität und die Entwicklungsgeschwindigkeit zu erhöhen.

Beispiele für die Testautomatisierung sind im Serverless Test Samples Repository verfügbar. Mit der folgenden Befehlszeile wird ein Beispiel für einen Python-Integrationstest automatisch ausgeführt:

```
python -m pytest -s tests/integration -v
```

Obwohl der Test lokal ausgeführt wird, interagiert er mit cloud-basierten Ressourcen. Diese Ressourcen wurden mithilfe des Befehlszeilentools AWS Serverless Application Model und AWS

SAM bereitgestellt. Der Testcode ruft zunächst die bereitgestellten Stack-Ausgaben ab, zu denen der API-Endpunkt, die Funktions-ARN und die Sicherheitsrolle gehören. Als Nächstes sendet der Test eine Anfrage an den API-Endpunkt, der mit einer Liste von Amazon-S3-Buckets antwortet. Dieser Test wird ausschließlich mit cloud-basierten Ressourcen ausgeführt, um sicherzustellen, dass diese Ressourcen bereitgestellt und gesichert sind und wie erwartet funktionieren.

```
========================= 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 ==========================
```

Für die Entwicklung cloudnativer Anwendungen bietet das Testen in der Cloud die folgenden Vorteile:

- Sie können jeden verfügbaren Service testen.
- Sie verwenden immer die neuesten Service-APIs und Rückgabewerte.
- Eine Cloud-Testumgebung ähnelt stark Ihrer Produktionsumgebung.
- Tests können Sicherheitsrichtlinien, Service Quotas, Konfigurationen und infrastrukturspezifische Parameter umfassen.
- Jeder Entwickler kann schnell eine oder mehrere Testumgebungen in der Cloud erstellen.
- Cloud-Tests erhöhen die Sicherheit, dass Ihr Code in der Produktion korrekt ausgeführt wird.

Das Testen in der Cloud hat einige Nachteile. Der offensichtlichste Nachteil von Tests in der Cloud ist, dass Bereitstellungen in Cloud-Umgebungen in der Regel länger dauern als Bereitstellungen in lokalen Desktop-Umgebungen.

Tools wie AWS Serverless Application Model (AWS SAM) Accelerate, AWS Cloud Development Kit (AWS CDK) Watch Mode und SST (Drittanbieter) reduzieren die mit Iterationen der Cloud-Bereitstellung verbundene Latenz. Diese Tools können Ihre Infrastruktur und Ihren Code überwachen und inkrementelle Updates in Ihrer Cloud-Umgebung automatisch bereitstellen.

> **Note**
>
> Im Serverless Developer Guide erfahren Sie, wie Sie Infrastruktur als Code erstellen, und erhalten weitere Informationen zu AWS Serverless Application Model, CloudFormation und AWS Cloud Development Kit (AWS CDK).

Im Gegensatz zu lokalen Tests sind beim Testen in der Cloud weitere Ressourcen erforderlich, wodurch Servicekosten entstehen können. Das Schaffen isolierter Testumgebungen kann die Belastung Ihrer DevOps-Teams erhöhen, insbesondere in Organisationen mit strengen Kontrollen in Bezug auf Konten und Infrastruktur. Bei der Arbeit mit komplexen Infrastrukturszenarien können die Kosten für die Einrichtung und Pflege einer komplizierten lokalen Umgebung jedoch ähnlich hoch sein wie bei der Verwendung von Einweg-Testumgebungen, die mit Automatisierungstools für Infrastruktur in Form von Code erstellt werden (oder sogar höher).

Tests in der Cloud sind trotz dieser Überlegungen immer noch der beste Weg, um die Qualität Ihrer Serverless-Lösungen zu gewährleisten.

Testen mit Mocks

Das Testen mit Mocks ist eine Technik, bei der Sie Ersatzobjekte in Ihrem Code erstellen, um das Verhalten eines Cloud-Services zu simulieren.

Beispielsweise könnten Sie einen Test schreiben, der einen Mock des Amazon-S3-Services verwendet, der bei jedem Aufruf der CreateObject-Methode eine bestimmte Antwort zurückgibt. Wenn ein Test ausgeführt wird, gibt der Mock die programmierte Antwort zurück, ohne Amazon S3 oder andere Service-Endpunkte aufzurufen.

Mock-Objekte werden oft von einem Mock-Framework generiert, um den Entwicklungsaufwand zu reduzieren. Einige Mock-Frameworks sind generisch und andere wurden speziell für AWS SDKs entwickelt, wie Moto, eine Python-Bibliothek zum Mocking von AWS-Services und Ressourcen.

Mock-Objekte unterscheiden sich von Emulatoren dadurch, dass Mocks in der Regel von einem Entwickler als Teil des Testcodes erstellt oder konfiguriert werden, wohingegen Emulatoren eigenständige Anwendungen sind, die die Funktionalität auf dieselbe Weise wie die Systeme, die sie emulieren, offenlegen.

Die Verwendung von Mocks bietet folgende Vorteile:

- Mocks können Dienste von Drittanbietern simulieren, die sich der Kontrolle Ihrer Anwendung entziehen, wie APIs und Software-as-a-Service (SaaS)-Anbieter, ohne dass ein direkter Zugriff auf diese Dienste erforderlich ist.
- Mocks sind nützlich für das Testen von Fehlerbedingungen, insbesondere wenn solche Bedingungen schwer zu simulieren sind, wie z. B. ein Service-Ausfall.
- Mocks können nach der Konfiguration schnelle lokale Tests ermöglichen.
- Mocks können Ersatzverhalten für praktisch jede Art von Objekt bereitstellen, so dass Mocking-Strategien eine breitere Palette von Diensten abdecken können als Emulatoren.
- Wenn neue Features oder Verhaltensweisen verfügbar werden, können Mock-Tests schneller reagieren. Durch die Verwendung eines generischen Mock-Frameworks können Sie neue Features simulieren, sobald das aktualisierte AWS-SDK verfügbar ist.

Mock-Tests haben die folgenden Nachteile:

- Mocks erfordern in der Regel einen nicht unerheblichen Einrichtungs- und Konfigurationsaufwand, insbesondere dann, wenn versucht wird, Rückgabewerte von verschiedenen Services zu ermitteln, um die Antworten korrekt nachzuahmen.

- Mocks werden von Entwicklern geschrieben, konfiguriert und müssen von ihnen gewartet werden, was ihre Verantwortung erhöht.
- Möglicherweise benötigen Sie Zugriff auf die Cloud, um die APIs und Rückgabewerte von Diensten zu verstehen.
- Mocks können schwierig zu warten sein. Wenn sich die Signaturen der nachgebildeten Cloud-APIs ändern oder die Rückgabewertschemata weiterentwickelt werden, müssen Sie Ihre Mocks aktualisieren. Mocks erfordern auch Updates, wenn Sie Ihre Anwendungslogik erweitern, um Aufrufe an neue APIs zu tätigen.
- Tests, die Mocks verwenden, können in Desktop-Umgebungen erfolgreich sein, in der Cloud jedoch fehlschlagen. Die Ergebnisse stimmen möglicherweise nicht mit der aktuellen API überein. Die Servicekonfiguration und die Kontingente können nicht getestet werden.
- Mock-Frameworks sind beim Testen oder Erkennen von AWS Identitäts- und Zugriffsmanagement (IAM)-Richtlinien oder Kontingentbeschränkungen eingeschränkt. Mocks können zwar besser simulieren, wenn die Autorisierung fehlschlägt oder ein Kontingent überschritten wird, aber Tests können nicht bestimmen, welches Ergebnis in einer Produktionsumgebung tatsächlich eintreten wird.

Lokales Testen mit AWS SAM CLI

Verwenden Sie AWS SAM CLI für das Testen Ihrer Funktionen in Docker-Containern mit derselben Laufzeitumgebung wie AWS Lambda. Sie können die Funktionslogik und die Ereignisverarbeitung lokal testen, ohne sie in der Cloud bereitstellen zu müssen. Wenn Ihre Funktion API-Aufrufe an andere AWS-Services sendet, erreichen diese Aufrufe echte AWS-Ressourcen.

Tests mit lokalen Containern bieten unter anderem folgende Vorteile:

- Verwendet Laufzeitumgebungen von AWS Lambda für genaue Tests.
- Ermöglicht schnelle lokale Entwicklungsiterationen ohne Cloud-Bereitstellung.
- Unterstützt das Debuggen mit vertrauten lokalen Entwicklungstools.

Das Testen mit lokalen Containern hat folgende Einschränkungen:

- AWS-Service-Aufrufe Ihrer Funktion interagieren mit realen AWS-Ressourcen, was Kosten verursachen und sich auf die Produktionsdaten auswirken kann.
- Erfordert, dass Docker installiert ist und lokal ausgeführt wird.

Testen mit Emulation

Emulatoren sind lokal ausgeführte Anwendungen, die AWS-Services nachahmen, indem sie ähnliche APIs und Rückgabewerte bereitstellen. LocalStack ist ein beliebtes Emulationstool, das eine vollständige lokale Entwicklungsumgebung zum Testen von Service-Integrationen bereitstellt.

LocalStack ist ein AWS-Cloud-Emulator, mit dem Sie Serverless-Anwendungen lokal testen können. Sie können Lambda-Funktionen testen, die in Services wie DynamoDB, Amazon S3 und Amazon SQS integriert sind, ohne eine Verbindung zu tatsächlichen AWS-Services herzustellen. Sie können LocalStack im AWS-Toolkit für VS Code verwenden.

Tests mit Emulatoren bieten unter anderem folgende Vorteile:

- Emulatoren können schnelle lokale Entwicklungsiterationen und Tests ermöglichen.
- Emulatoren bieten Entwicklern, die es gewohnt sind, Code in einer lokalen Umgebung zu entwickeln, eine vertraute Umgebung. Wenn Sie beispielsweise mit der Entwicklung einer n-Tier-Anwendung sind, verfügen Sie möglicherweise über einen Datenbank-Engine und einen Webserver, ähnlich denen, die in der Produktion laufen, auf Ihrem lokalen Rechner, um schnelle, lokale, isolierte Testfunktionen bereitzustellen.
- Emulatoren erfordern keine Änderungen an der Cloud-Infrastruktur (z. B. Cloud-Konten für Entwickler), sodass sie einfach mit vorhandenen Testmustern implementiert werden können.
- Da Emulatoren keine tatsächlichen AWS-Ressourcen verwenden, fallen keine unerwarteten Gebühren an, wenn Sie mehrere Services starten oder wenn Sie einige Ressourcen über einen längeren Zeitraum ausführen.

Das Testen mit Emulatoren hat folgende Nachteile:

- Emulatoren können schwierig einzurichten und zu replizieren sein, insbesondere wenn sie in CI/CD-Pipelines verwendet werden. Dies kann den Workload von IT-Mitarbeitern oder Entwicklern erhöhen, die ihre eigene Software verwalten.
- Emulierte Features und APIs hinken in der Regel den Service-Updates hinterher. Dies kann zu Fehlern führen, da der getestete Code nicht mit der tatsächlichen API übereinstimmt, und die Einführung neuer Features behindern.
- Emulatoren benötigen Unterstützung, Updates, Bugfixes und Verbesserungen der Featureparität. Diese liegen in der Verantwortung des Emulatorautors, bei dem es sich um ein Drittunternehmen handeln kann.

- Tests, die auf Emulatoren basieren, können lokal zu erfolgreichen Ergebnissen führen, schlagen in der Cloud jedoch aufgrund von Produktionssicherheitsrichtlinien, dienstübergreifenden Konfigurationen oder der Überschreitung von Lambda-Kontingenten fehl.

Bewährte Methoden

Die folgenden Abschnitte enthalten Empfehlungen für erfolgreiche Tests für Serverless-Anwendungen.

Praktische Beispiele für Tests und Testautomatisierung finden Sie im Serverless Test Samples Repository.

Priorisieren Sie Tests in der Cloud

Tests in der Cloud bieten die zuverlässigste, genaueste und vollständigste Testabdeckung. Durch die Durchführung von Tests im Kontext der Cloud werden nicht nur die Geschäftslogik, sondern auch Sicherheitsrichtlinien, Service-Konfigurationen, Kontingente und die aktuellsten API-Signaturen und Rückgabewerte umfassend getestet.

Strukturieren Sie Ihren Code zum Testen

Vereinfachen Sie Ihre Tests und Lambda-Funktionen, indem Sie Lambda-spezifischen Code von Ihrer Kerngeschäftslogik trennen.

Ihr Lambda-Funktions-Handler sollte ein schlanker Adapter sein, der Ereignisdaten aufnimmt und nur die Details an Ihre Geschäftslogikmethode(n) weiterleitet. Mit dieser Strategie können Sie umfassende Tests rund um Ihre Geschäftslogik durchführen, ohne sich über Lambda-spezifische Details Gedanken machen zu müssen. Ihre AWS-Lambda-Funktionen sollten nicht die Einrichtung einer komplexen Umgebung oder einer großen Anzahl von Abhängigkeiten erfordern, um die zu testende Komponente zu erstellen und zu initialisieren.

Im Allgemeinen sollten Sie einen Handler schreiben, der Daten aus den eingehenden Ereignis- und Kontext-Objekten extrahiert und validiert und diese Eingabe dann an Methoden sendet, die Ihre Geschäftslogik ausführen.

Entwicklungs-Feedback-Schleifen beschleunigen

Es gibt Tools und Techniken, um die Entwicklungs-Feedback-Schleifen zu beschleunigen. AWSSAM Accelerate und AWSCDK Watch Mode reduzieren beispielsweise beide die Zeit, die für die Aktualisierung von Cloud-Umgebungen benötigt wird.

In den Beispielen im GitHub Serverless Test Samples Repository werden einige dieser Techniken untersucht.

Wir empfehlen außerdem, dass Sie Cloud-Ressourcen so früh wie möglich während der Entwicklung erstellen und testen — nicht erst, nachdem Sie sich bei der Quellverwaltung angemeldet haben. Diese Praxis ermöglicht ein schnelleres Erkunden und Experimentieren bei der Entwicklung von Lösungen. Darüber hinaus hilft Ihnen die Automatisierung der Bereitstellung von einem Entwicklungscomputer aus dabei, Probleme mit der Cloud-Konfiguration schneller zu erkennen und unnötigen Aufwand für Updates und Code-Review-Prozesse zu reduzieren.

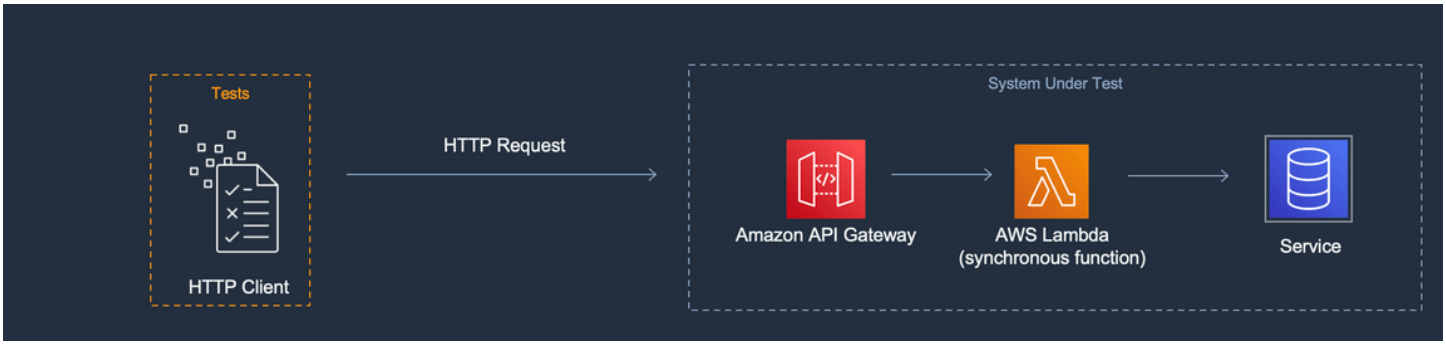
Fokus auf Integrationstests

Beim Erstellen von Anwendungen mit Lambda hat es sich bewährt, Komponenten gemeinsam zu testen.

Tests, die mit zwei oder mehr Architekturkomponenten ausgeführt werden, werden als Integrationstests bezeichnet. Das Ziel der Integrationstests besteht darin, nicht nur zu verstehen, wie Ihr Code komponentenübergreifend ausgeführt wird, sondern auch, wie sich die Umgebung, in der Ihr Code gehostet wird, verhält. End-to-End-Tests sind spezielle Arten von Integrationstests, mit denen das Verhalten einer gesamten Anwendung überprüft wird.

Um Integrationstests zu erstellen, stellen Sie Ihre Anwendung in einer Cloud-Umgebung bereit. Dies kann von einer lokalen Umgebung aus oder über eine CI/CD-Pipeline erfolgen. Schreiben Sie dann Tests, um das zu testende System (SUT) zu testen und das erwartete Verhalten zu validieren.

Das zu testende System könnte beispielsweise eine Anwendung sein, die API Gateway, Lambda und DynamoDB verwendet. Ein Test könnte einen synthetischen HTTP-Aufruf an einen API-Gateway-Endpunkt ausführen und überprüfen, ob die Antwort die erwartete Nutzlast enthielt. Dieser Test bestätigt, dass der AWS Lambda-Code korrekt ist und dass jeder Service korrekt konfiguriert ist, um die Anfrage zu verarbeiten, einschließlich der IAM-Berechtigungen zwischen ihnen. Darüber hinaus könnten Sie den Test so gestalten, dass Datensätze unterschiedlicher Größe geschrieben werden, um zu überprüfen, ob Ihre Service-Kontingente, wie z. B. die maximale Datensatzgröße in DynamoDB, korrekt eingerichtet sind.



Erstellen Sie isolierte Testumgebungen

Tests in der Cloud erfordern in der Regel isolierte Entwicklerumgebungen, sodass sich Tests, Daten und Ereignisse nicht überschneiden.

Ein Ansatz besteht darin, jedem Entwickler ein eigenes AWS-Konto zur Verfügung zu stellen. Dadurch werden Konflikte mit der Ressourcenbenennung vermieden, die auftreten können, wenn mehrere Entwickler in einer gemeinsamen Codebasis arbeiten, versuchen, Ressourcen bereitzustellen oder eine API aufzurufen.

Automatisierte Testprozesse sollten für jeden Stapel eindeutig benannte Ressourcen erstellen. Sie können beispielsweise Skripte oder TOML-Konfigurationsdateien so einrichten, dass die AWS SAMCLI-Befehle sam deploy oder sam sync automatisch einen Stack mit einem eindeutigen Präfix angeben.

In einigen Fällen teilen sich Entwickler ein AWS-Konto. Dies kann daran liegen, dass in Ihrem Stack Ressourcen vorhanden sind, deren Betrieb oder Bereitstellung und Konfiguration teuer sind. Beispielsweise kann eine Datenbank gemeinsam genutzt werden, um die korrekte Einrichtung und Bereitstellung der Daten zu vereinfachen.

Wenn Entwickler ein Konto gemeinsam nutzen, müssen Sie Grenzen festlegen, um die Inhaberschaft zu ermitteln und Überschneidungen zu vermeiden. Eine Möglichkeit, dies zu tun, besteht darin, den Stack-Namen die Benutzer-IDs der Entwickler voranzustellen. Ein weiterer beliebter Ansatz ist das Einrichten von Stacks, die auf Code-Branches basieren. Aufgrund der Filialgrenzen sind die Umgebungen isoliert, aber Entwickler können dennoch Ressourcen gemeinsam nutzen, z. B. eine relationale Datenbank. Dieser Ansatz ist eine bewährte Methode, wenn Entwickler an mehr als einer Verzweigung gleichzeitig arbeiten.

Das Testen in der Cloud ist für alle Testphasen von Nutzen, einschließlich Einheitentests, Integrationstests und End-to-End-Tests. Das Aufrechterhalten einer ordnungsgemäßen Isolierung ist von entscheidender Bedeutung. Dennoch sollten Sie darauf achten, dass Ihre QA-Umgebung

der Produktionsumgebung so nahe wie möglich kommt. Aus diesem Grund fügen Teams den QA-Umgebungen Änderungssteuerungsvorgänge hinzu.

In Vorproduktions- und Produktionsumgebungen werden die Grenzen in der Regel auf Kontoebene gezogen, um Arbeitslasten von „Noisy-Neighbor“-Problemen zu isolieren und Sicherheitskontrollen mit geringsten Berechtigungen zum Schutz sensibler Daten zu implementieren. Für Workloads gibt es Kontingente. Ihre Tests dürfen weder die für die Produktion zugewiesenen Kontingente verbrauchen („Noisy Neighbor“) noch Zugriff auf Kundendaten haben. Lasttests sind eine weitere Aktivität, die Sie von Ihrem Produktions-Stack isolieren sollten.

In jedem Fall müssen die Umgebungen mit Warnungen und Kontrollen konfiguriert werden, um unnötige Ausgaben zu vermeiden. Sie können beispielsweise die Art, Stufe oder Größe der Ressourcen einschränken, die erstellt werden können, und E-Mail-Benachrichtigungen einrichten, wenn die geschätzten Kosten einen bestimmten Schwellenwert überschreiten.

Mocks für isolierte Geschäftslogik verwenden

Mock-Frameworks sind ein wertvolles Tool zum Schreiben schneller Einheitentests. Sie sind besonders nützlich, wenn Tests komplexe interne Geschäftslogiken wie mathematische oder finanzielle Berechnungen oder Simulationen abdecken. Suchen Sie nach Einheitentests, die eine große Anzahl von Testfällen oder Eingabevariationen enthalten, bei denen diese Eingaben das Muster oder den Inhalt von Aufrufen an andere Cloud-Services nicht ändern.

Code, der durch Komponententests mit Mocks abgedeckt wird, muss auch durch Tests in der Cloud abgedeckt werden. Dies wird empfohlen, da ein Entwickler-Laptop oder eine Build-Machine-Umgebung anders konfiguriert werden kann als eine Produktionsumgebung in der Cloud. Beispielsweise beanspruchen Ihre Lambda-Funktionen möglicherweise mehr Speicher oder Zeit als zugewiesen, wenn sie mit bestimmten Eingabeparametern ausgeführt werden. Möglicherweise enthält Ihr Code auch Umgebungsvariablen, die nicht auf die gleiche Weise (oder überhaupt nicht) konfiguriert sind, und die Unterschiede könnten dazu führen, dass sich der Code anders verhält oder fehlschlägt.

Der Nutzen von Mocks ist bei Integrationstests geringer, da der Aufwand für die Implementierung der notwendigen Mocks mit der Anzahl der Verbindungspunkte steigt. Bei End-to-End-Tests dürfen keine Mocks verwendet werden, da sich diese Tests im Allgemeinen mit Zuständen und komplexer Logik befassen, die mit Mock-Frameworks nicht einfach simuliert werden können.

Vermeiden Sie schließlich die Verwendung von nachgebildeten Cloud-Services, um die ordnungsgemäße Implementierung von Service-Aufrufen zu überprüfen. Führen Sie stattdessen

Cloud-Service-Aufrufe in der Cloud durch, um Verhalten, Konfiguration und funktionale Implementierung zu überprüfen.

Emulatoren sparsam verwenden

Emulatoren können für einige Anwendungsfälle praktisch sein, z. B. für ein Entwicklungsteam mit begrenztem, unzuverlässigem oder langsamem Internetzugang. In den meisten Fällen sollten Sie Emulatoren jedoch sparsam verwenden.

Indem Sie Emulatoren vermeiden, können Sie die neuesten Servicefeatures und aktuellen APIs entwickeln und Innovationen mit ihnen umsetzen. Sie werden nicht auf die Veröffentlichung von Anbietern warten müssen, um die Featureparität zu erreichen. Sie reduzieren Ihre Vorabkosten und laufenden Kosten für den Kauf und die Konfiguration mehrerer Entwicklungssysteme und bauen Maschinen. Darüber hinaus vermeiden Sie das Problem, dass bei vielen Cloud-Services einfach keine Emulatoren verfügbar sind. Eine Teststrategie, die auf Emulation basiert, macht es unmöglich, diese Services zu nutzen (was zu potenziell teureren Problemumgehungen führt) oder Code und Konfigurationen zu erstellen, die nicht gut getestet wurden.

Wenn Sie die Emulation zum Testen verwenden, müssen Sie dennoch in der Cloud testen, um die Konfiguration zu überprüfen und Interaktionen mit Cloud-Diensten zu testen, die in einer emulierten Umgebung nur simuliert oder nachgeahmt werden können.

Herausforderungen beim Testen vor Ort

Wenn Sie Emulatoren und simulierte Aufrufe verwenden, um auf Ihrem lokalen Desktop zu testen, können Testinkonsistenzen auftreten, wenn Ihr Code in Ihrer CI/CD-Pipeline von Umgebung zu Umgebung fortschreitet. Einheitentests zur Validierung der Geschäftslogik Ihrer Anwendung auf Ihrem Desktop testen möglicherweise wichtige Aspekte der Cloud-Services nicht genau.

Die folgenden Beispiele zeigen, worauf beim lokalen Testen mit Mocks und Emulatoren zu achten ist:

Beispiel: Die Lambda-Funktion erstellt einen S3-Bucket

Wenn die Logik einer Lambda-Funktion von der Erstellung eines S3-Buckets abhängt, sollte ein vollständiger Test bestätigen, dass Amazon S3 aufgerufen und der Bucket erfolgreich erstellt wurde.

- In einem Test-Setup können Sie eine Erfolgsreaktion simulieren und möglicherweise einen Testfall hinzufügen, um eine Fehlerreaktion zu behandeln.
- In einem Emulationstestszenario kann die CreateBucket-API aufgerufen werden. Sie müssen sich jedoch bewusst sein, dass die Identität, die den lokalen Aufruf durchführt, nicht vom Lambda-

Service stammt. Die Anrufindentität übernimmt keine Sicherheitsrolle wie in der Cloud. Daher wird stattdessen eine Platzhalterauthentifizierung verwendet, möglicherweise mit einer permissiveren Rolle oder Benutzeridentität, die bei Ausführung in der Cloud anders sein wird.

Die Mock- und Emulations-Setups testen, was die Lambda-Funktion tut, wenn sie Amazon S3 aufruft. Diese Tests überprüfen jedoch nicht, ob die Lambda-Funktion, wie konfiguriert, in der Lage ist, den Amazon-S3-Bucket erfolgreich zu erstellen. Sie müssen sicherstellen, dass der Rolle, die der Funktion zugewiesen ist, eine Sicherheitsrichtlinie angehängt ist, die es der Funktion ermöglicht, die `s3:CreateBucket`-Aktion auszuführen. Andernfalls schlägt die Funktion wahrscheinlich fehl, wenn sie in einer Cloud-Umgebung bereitgestellt wird.

Beispiel: Verwenden Sie eine Lambda-Funktion, um Nachrichten aus einer Amazon-SQS-Warteschlange zu verarbeiten.

Wenn eine Amazon-SQS-Warteschlange die Quelle einer Lambda-Funktion ist, muss durch einen vollständigen Test überprüft werden, ob die Lambda-Funktion erfolgreich aufgerufen wird, wenn eine Nachricht in eine Warteschlange gestellt wird.

Emulationstests und Mock-Tests werden in der Regel so eingerichtet, dass der Lambda-Funktionscode direkt ausgeführt wird und die Amazon-SQS-Integration simuliert wird, indem eine JSON-Event-Payload (oder ein deserialisiertes Objekt) als Eingabe des Funktions-Handlers übergeben wird.

Bei lokalen Tests, bei denen die Amazon-SQS-Integration simuliert wird, wird getestet, was die Lambda-Funktion tut, wenn sie von Amazon SQS mit einer bestimmten Nutzlast aufgerufen wird. Der Test bestätigt jedoch nicht, dass Amazon SQS die Lambda-Funktion erfolgreich aufruft, wenn sie in einer Cloud-Umgebung bereitgestellt wird.

Einige Beispiele für Konfigurationsprobleme, die bei Amazon SQS und Lambda auftreten können, sind die folgenden:

- Das Amazon-SQS-Sichtbarkeits-Timeout ist zu niedrig, was zu mehreren Aufrufen führt, obwohl nur einer vorgesehen war.
- Die Ausführungsrolle der Lambda-Funktion erlaubt es nicht, Nachrichten aus der Warteschlange zu lesen (durch `sqs:ReceiveMessage`,`sqs:DeleteMessage`, oder `sqs:GetQueueAttributes`).
- Das Beispielereignis, das an die Lambda-Funktion übergeben wird, überschreitet das Amazon-SQS-Nachrichtengrößenkontingent. Daher ist der Test ungültig, da Amazon SQS niemals in der Lage wäre, eine Nachricht dieser Größe zu senden.

Wie diese Beispiele zeigen, liefern Tests, die sich mit der Geschäftslogik, aber nicht mit den Konfigurationen zwischen Cloud-Services befassen, wahrscheinlich zu unzuverlässigen Ergebnissen.

Häufig gestellte Fragen

Ich habe eine Lambda-Funktion, die Berechnungen durchführt und ein Ergebnis zurückgibt, ohne andere Dienste aufzurufen. Muss ich es wirklich in der Cloud testen?

Ja. Lambda-Funktionen haben Konfigurationsparameter, die das Testergebnis verändern könnten. Der gesamte Lambda-Funktionscode ist von Timeout- und Speichereinstellungen abhängig, was dazu führen kann, dass die Funktion fehlschlägt, wenn diese Einstellungen nicht richtig gesetzt sind. Lambda-Richtlinien ermöglichen auch die standardmäßige Output-Protokollierung in Amazon CloudWatch. Auch wenn Ihr Code CloudWatch nicht direkt aufruft, ist eine Genehmigung erforderlich, um das Protokoll zu aktivieren. Diese erforderliche Erlaubnis kann nicht korrekt verspottet oder nachgeahmt werden.

Wie können Tests in der Cloud beim Einheitentest helfen? Wenn es sich in der Cloud befindet und eine Verbindung zu anderen Ressourcen herstellt, ist das nicht ein Integrationstest?

Wir definieren Komponententests als Tests, die isoliert auf Architekturkomponenten ausgeführt werden. Dies hindert Tests jedoch nicht daran, Komponenten einzubeziehen, die möglicherweise andere Services aufrufen oder Netzwerkkommunikation nutzen.

Viele Serverless-Anwendungen verfügen über Architekturkomponenten, die isoliert getestet werden können, sogar in der Cloud. Ein Beispiel ist eine Lambda-Funktion, um Eingaben entgegenzunehmen, die Daten zu verarbeiten und eine Nachricht an eine Amazon-SQS-Warteschlange zu senden. Ein Komponententest dieser Funktion würde wahrscheinlich testen, ob Eingabewerte dazu führen, dass bestimmte Werte in der Nachricht in der Warteschlange vorhanden sind.

Stellen Sie sich einen Test vor, der nach dem Muster „Arrange, Act, Assert“ geschrieben wurde:

- Arrange: Zuweisung von Ressourcen (eine Warteschlange zum Empfang von Nachrichten und die zu testende Funktion).
- Act: Aufrufen der zu testenden Funktion.
- Assert: Abrufen der von der Funktion gesendeten Nachricht und Validieren der Ausgabe.

Ein Mock-Testansatz würde das Mocking der Warteschlange mit einem prozessinternen Mock-Objekt und das Erstellen einer prozessinternen Instance der Klasse oder des Moduls, das den

Lambda-Funktionscode enthält, beinhalten. Während der Assert-Phase würde die Nachricht in der Warteschlange aus dem simulierten Objekt abgerufen.

Bei einem cloud-basierten Ansatz würde der Test eine Amazon-SQS-Warteschlange für die Testzwecke erstellen und die Lambda-Funktion mit Umgebungsvariablen bereitstellen, die so konfiguriert sind, dass sie die isolierte Amazon-SQS-Warteschlange als Ausgabeziel verwenden. Nach dem Ausführen der Lambda-Funktion würde der Test die Nachricht aus der Amazon-SQS-Warteschlange abrufen.

Der cloud-basierte Test würde denselben Code ausführen, dasselbe Verhalten bestätigen und die funktionale Korrektheit der Anwendung überprüfen. Er hätte jedoch den zusätzlichen Vorteil, dass die Einstellungen der Lambda-Funktion validiert werden könnten: die IAM-Rolle, die IAM-Richtlinien sowie die Timeout- und Speichereinstellungen der Funktion.

Nächste Schritte und Ressourcen

Verwenden Sie die folgenden Ressourcen, um weitere Informationen und praktische Testbeispiele zu erhalten.

Beispielimplementierungen

Das Serverless Test Samples Repository auf GitHub enthält konkrete Beispiele für Tests, die den in diesem Handbuch beschriebenen Mustern und Best Practices folgen. Das Repository enthält Beispielcode und Anleitungen zu den Mock-, Emulations- und Cloud-Testprozessen, die in den vorherigen Abschnitten beschrieben wurden. Verwenden Sie dieses Repository, um sich über die neuesten Anleitungen für Serverless-Tests von AWS auf dem Laufenden zu halten.

Weitere Informationen

Besuchen Sie Serverless Land, um auf die neuesten Blogs, Videos und Schulungen für AWS Serverless-Technologien zuzugreifen.

Die folgenden AWS Blogbeiträge werden ebenfalls zur Lektüre empfohlen:

- Beschleunigung der Serverless-Entwicklung mit AWS SAM Accelerate (AWSBlogbeitrag)
- Erhöhung der Entwicklungsgeschwindigkeit mit CDK Watch (AWSBlogbeitrag)
- Mocking-Service-Integrationen mit AWS Step Functions Lokal (AWSBlogbeitrag)
- Erste Schritte beim Testen von Serverless-Anwendungen (AWSBlogbeitrag)

Tools

- AWS SAM – Testen und Debuggen von Serverless-Anwendungen
- AWS SAM – Integration in automatisierte Tests
- Lambda — Testen von Lambda-Funktionen mit Hilfe der Lambda-Konsole

Verbesserung der Startleistung mit Lambda SnapStart

Lambda SnapStart kann eine Startleistung von nur unter einer Sekunde bieten, normalerweise ohne Änderungen an Ihrem Funktionscode. SnapStart macht es einfacher, reaktionsschnelle und skalierbare Anwendungen zu erstellen, ohne Ressourcen bereitzustellen oder komplexe Leistungsoptimierungen zu implementieren.

Der größte Beitrag zur Startup-Latenz (oft als Kaltstartzeit bezeichnet) ist die Zeit, die Lambda für die Initialisierung der Funktion aufwendet. Dazu gehören das Laden des Funktionscodes, der Start der Laufzeit und die Initialisierung des Funktionscodes. Mit initialisiert Lambda Ihre Funktion SnapStart, wenn Sie eine Funktionsversion veröffentlichen. Lambda erstellt einen Firecracker-microVM-Snapshot des Arbeitsspeichers und des Festplattenstatus der initialisierten Ausführungsumgebung, verschlüsselt den Snapshot und legt ihn in einem intelligenten Cache ab, um die Abruflatenz zu optimieren.

Um die Stabilität zu gewährleisten, verwaltet Lambda mehrere Kopien jedes Snapshots. Lambda patcht Snapshots und ihre Kopien automatisch mit den neuesten Laufzeit- und Sicherheitsupdates. Wenn Sie die Funktionsversion zum ersten Mal aufrufen und wenn die Aufrufe hochskalieren, nimmt Lambda neue Ausführungsumgebungen aus dem im Cache gespeicherten Snapshot wieder auf, anstatt sie von Grund auf neu zu initialisieren, wodurch die Startup-Latenz verbessert wird.

> **Important**
>
> Wenn Ihre Anwendungen von der Eindeutigkeit des Zustands abhängen, müssen Sie Ihren Funktionscode auswerten und sicherstellen, dass er gegenüber Snapshot-Vorgängen widerstandsfähig ist. Weitere Informationen finden Sie unter Handhabung der Eindeutigkeit mit Lambda SnapStart.

Themen

- Wann sollte es verwendet werden SnapStart
- Unterstützte Funktionen und Einschränkungen
- Unterstützte Regionen
- Erwägungen zur Kompatibilität
- SnapStart Preisgestaltung

- Lambda aktivieren und verwalten SnapStart
- Handhabung der Eindeutigkeit mit Lambda SnapStart
- Implementieren Sie Code vor oder nach Lambda-Funktions-Snapshots
- Überwachung für Lambda SnapStart
- Sicherheitsmodell für Lambda SnapStart
- Maximieren Sie die Leistung von Lambda SnapStart
- Fehlerbehebung bei SnapStart-Fehlern für Lambda-Funktionen

Wann sollte es verwendet werden SnapStart

Lambda SnapStart wurde entwickelt, um die Latenzvariabilität zu beheben, die durch einmaligen Initialisierungscode verursacht wird, z. B. beim Laden von Modulabhängigkeiten oder Frameworks. Beim ersten Aufruf kann es manchmal mehrere Sekunden dauern, bis diese Vorgänge abgeschlossen sind. Wird verwendet SnapStart , um diese Latenz in optimalen Szenarien von mehreren Sekunden auf weniger als eine Sekunde zu reduzieren. SnapStart funktioniert am besten, wenn es mit Funktionsaufrufen in großem Maßstab verwendet wird. Funktionen, die selten aufgerufen werden, weisen möglicherweise nicht dieselben Leistungsverbesserungen auf.

SnapStart ist besonders vorteilhaft für zwei Haupttypen von Anwendungen:

- Latenzabhängige Abläufe APIs und Benutzerabläufe: Funktionen, die Teil kritischer API-Endpunkte oder benutzerorientierter Abläufe sind, können von der geringeren Latenz und den verbesserten SnapStart Reaktionszeiten profitieren.
- Workflows zur latenzsensitiven Datenverarbeitung: Zeitgebundene Datenverarbeitungsworkflows, die Lambda-Funktionen verwenden, können einen besseren Durchsatz erzielen, indem sie die Latenz bei der Initialisierung von Ausreißerfunktionen reduzieren.

Die bereitgestellte Gleichzeitigkeit sorgt dafür, dass die Funktionen im zweistelligen Millisekundenbereich initialisiert und reaktionsbereit sind. Verwenden Sie bereitgestellte Parallelität, wenn Ihre Anwendung strenge Kaltstart-Latenzanforderungen hat, die nicht angemessen erfüllt werden können. SnapStart

Unterstützte Funktionen und Einschränkungen

SnapStart ist für die folgenden von Lambda verwalteten Laufzeiten verfügbar:

- Java 11 und höher
- Python 3.12 oder höher
- .NET 8 und höher. Wenn Sie das Lambda Annotations Framework für.NET verwenden, führen Sie ein Upgrade auf Amazon.Lambda.Annotations Version 1.6.0 oder höher durch, um die Kompatibilität mit sicherzustellen. SnapStart

Andere verwaltete Laufzeiten (wie `nodejs24.x` und `ruby3.4`), Reine OS-Laufzeiten und Container-Images werden nicht unterstützt.

SnapStart unterstützt keine bereitgestellte Parallelität, Amazon Elastic File System (Amazon EFS) oder kurzlebigen Speicher mit mehr als 512 MB.

> **Note**
>
> Sie können es SnapStart nur für veröffentlichte Funktionsversionen und Aliase verwenden, die auf Versionen verweisen. Sie können es nicht für die unveröffentlichte Version einer Funktion ($LATEST) verwenden SnapStart .

Unterstützte Regionen

Lambda SnapStart ist in allen Handelsregionen außer Asien-Pazifik (Neuseeland) und Asien-Pazifik (Taipeh) erhältlich.

Erwägungen zur Kompatibilität

Mit SnapStart verwendet Lambda einen einzelnen Snapshot als Ausgangszustand für mehrere Ausführungsumgebungen. Wenn Ihre Funktion während der Initialisierungsphase eines der folgenden Elemente verwendet, müssen Sie vor der Verwendung möglicherweise einige Änderungen vornehmen: SnapStart

Eindeutigkeit

Wenn Ihr Initialisierungscode eindeutigen Inhalt generiert, der im Snapshot enthalten ist, ist der Inhalt möglicherweise nicht eindeutig, wenn er in Ausführungsumgebungen wiederverwendet wird. Um die Einzigartigkeit bei der Verwendung zu wahren SnapStart, müssen Sie nach

der Initialisierung eindeutige Inhalte generieren. Dazu gehören einzigartige IDs, einzigartige Geheimnisse und Entropie, die verwendet wird, um Pseudozufälligkeit zu erzeugen. Informationen zum Wiederherstellen von Eindeutigkeit finden Sie unter Handhabung der Eindeutigkeit mit Lambda SnapStart.

Netzwerkverbindungen

Der Zustand der Verbindungen, die Ihre Funktion während der Initialisierungsphase herstellt, ist nicht garantiert, wenn Lambda Ihre Funktion von einem Snapshot fortsetzt. Überprüfen Sie den Status Ihrer Netzwerkverbindungen und stellen Sie diese bei Bedarf wieder her. In den meisten Fällen werden Netzwerkverbindungen, die ein SDK herstellt, automatisch wieder aufgenommen. AWS Für andere Verbindungen lesen Sie die bewährten Methoden.

Temporäre Daten

Einige Funktionen laden während der Initialisierungsphase kurzlebige Daten herunter oder initialisieren diese, wie etwa temporäre Anmeldeinformationen oder im Cache gespeicherte Zeitstempel. Aktualisieren Sie kurzlebige Daten im Funktionshandler, bevor Sie sie verwenden, auch wenn Sie sie nicht verwenden. SnapStart

SnapStart Preisgestaltung

> **Note**
>
> Für Java-verwaltete Laufzeiten fallen keine zusätzlichen Kosten für SnapStart an. Die Abrechnung erfolgt basierend auf der Anzahl der Anfragen für Ihre Funktionen, der Zeit, die Ihr Code zum Ausführen benötigt, und dem für Ihre Funktion konfigurierten Arbeitsspeicher.

Die Kosten für die Nutzung SnapStart beinhalten Folgendes:

- Caching: Für jede Funktionsversion, die Sie mit SnapStart aktivierter Option veröffentlichen, zahlen Sie für die Zwischenspeicherung und Wartung des Snapshots. Der Preis hängt von der Speichermenge ab, die Sie Ihrer Funktion zuweisen. Ihnen werden mindestens 3 Stunden in Rechnung gestellt. Ihnen werden Gebühren berechnet, solange Ihre Funktion aktiv ist. Verwenden Sie die ListVersionsByFunctionAPI-Aktion, um Funktionsversionen zu identifizieren, und verwenden Sie sie dann, um ungenutzte Versionen DeleteFunctionzu löschen. Informationen zum automatischen Löschen ungenutzter Funktionsversionen finden Sie im Lambda Version Cleanup-Pattern auf Serverless Land.

- Wiederherstellung: Jedes Mal, wenn eine Funktions-Instance aus einem Snapshot wiederhergestellt wird, zahlen Sie eine Wiederherstellungsgebühr. Der Preis hängt von der Speichermenge ab, die Sie Ihrer Funktion zuweisen.

Wie bei allen Lambda-Funktionen fallen Dauergebühren für Code an, der im Funktionshandler ausgeführt wird. Bei SnapStart Funktionen fallen Gebühren für die Dauer auch für Initialisierungscode an, der außerhalb des Handlers deklariert wurde, für die Zeit, die zum Laden der Laufzeit benötigt wird, und für jeden Code, der in einem Runtime-Hook ausgeführt wird. Die Dauer wird vom Beginn der Ausführung Ihres Codes bis zur Rückgabe oder einem anderen Ende berechnet, aufgerundet auf die nächste 1 ms. Lambda verwaltet aus Gründen der Ausfallsicherheit zwischengespeicherte Kopien Ihres Snapshots und wendet automatisch Softwareupdates wie Laufzeit-Upgrades und Sicherheitspatches auf diese an. Jedes Mal, wenn Lambda Ihren Initialisierungscode erneut ausführt, um Softwareupdates zu installieren, fallen Gebühren an.

Weitere Informationen zu den Nutzungskosten finden Sie unter SnapStart AWS Lambda Preisgestaltung.

Lambda aktivieren und verwalten SnapStart

Um es zu verwenden SnapStart, aktivieren Sie es SnapStart auf einer neuen oder vorhandenen Lambda-Funktion. Veröffentlichen und rufen Sie dann eine Funktionsversion auf.

Themen

- Aktivierung SnapStart (Konsole)
- SnapStart Aktiviert (AWS CLI)
- Aktivierung SnapStart (API)
- Lambda SnapStart und Funktionszustände
- Aktualisieren eines Snapshots
- SnapStart Verwenden mit AWS SDKs
- Verwenden Sie SnapStart mit CloudFormationAWS SAM, und AWS CDK
- Löschen von Snapshots

Aktivierung SnapStart (Konsole)

Zur Aktivierung SnapStart für eine Funktion

1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.
2. Wählen Sie den Namen einer Funktion aus.
3. Wählen Sie Configuration (Konfiguration) und dann General configuration (Allgemeine Konfiguration).
4. Wählen Sie im Bereich General configuration (Allgemeine Konfiguration) die Option Edit (Bearbeiten) aus.
5. Wählen Sie auf der Seite Grundeinstellungen bearbeiten für die SnapStartOption Veröffentlichte Versionen aus.
6. Wählen Sie Speichern.
7. Veröffentlichen einer Funktionsversion. Lambda initialisiert Ihren Code, erstellt einen Snapshot der initialisierten Ausführungsumgebung und speichert den Snapshot im Cache für einen Zugriff mit geringer Latenz.
8. Rufen Sie die Funktionsversion auf.

SnapStart Aktiviert (AWS CLI)

Zur Aktivierung SnapStart für eine bestehende Funktion

1. Aktualisieren Sie die Funktionskonfiguration, indem update-function-configurationSie den Befehl mit der --snap-start Option ausführen.

```
aws lambda update-function-configuration \
  --function-name my-function \
  --snap-start ApplyOn=PublishedVersions
```

2. Veröffentlichen Sie eine Funktionsversion mit dem publish-version-Befehl.

```
aws lambda publish-version \
  --function-name my-function
```

3. Bestätigen Sie, dass sie für die Funktionsversion aktiviert SnapStart ist, indem Sie den get-function-configurationBefehl ausführen und die Versionsnummer angeben. Das folgende Beispiel legt Version 1 fest.

```
aws lambda get-function-configuration \
  --function-name my-function:1
```

   Wenn in der Antwort angezeigt OptimizationStatuswird, dass dies der Fall ist On und der Status SnapStart lautet`Active`, dann ist aktiviert und ein Snapshot ist für die angegebene Funktionsversion verfügbar.

```
"SnapStart": {
    "ApplyOn": "PublishedVersions",
    "OptimizationStatus": "On"
 },
 "State": "Active",
```

4. Rufen Sie die Funktionsversion auf, indem Sie den Aufruf-Befehl ausführen und die Version angeben. Das folgende Beispiel ruft Version 1 auf.

```
aws lambda invoke \
  --cli-binary-format raw-in-base64-out \
  --function-name my-function:1 \
  --payload '{ "name": "Bob" }' \
```

```
  response.json
```

Die cli-binary-format Option ist erforderlich, wenn Sie AWS CLI Version 2 verwenden. Um dies zur Standardeinstellung zu machen, führen Sie `aws configure set cli-binary-format raw-in-base64-out` aus. Weitere Informationen finden Sie unter Von AWS CLI unterstützte globale Befehlszeilenoptionen im AWS Command Line Interface -Benutzerhandbuch für Version 2.

Zur Aktivierung SnapStart , wenn Sie eine neue Funktion erstellen

1. Erstellen Sie eine Funktion, indem Sie den create-function-Befehl mit der --snap-start-Option ausführen. Geben Sie für --role den Amazon-Ressourcennamen (ARN) Ihrer Ausführungsrolle an.

```
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. Erstellen Sie eine Version mit dem publish-version-Befehl.

```
aws lambda publish-version \
  --function-name my-function
```

3. Bestätigen Sie, dass sie für die Funktionsversion aktiviert SnapStart ist, indem Sie den get-function-configurationBefehl ausführen und die Versionsnummer angeben. Das folgende Beispiel legt Version 1 fest.

```
aws lambda get-function-configuration \
  --function-name my-function:1
```

Wenn in der Antwort angezeigt OptimizationStatuswird, dass dies der Fall ist On und der Status SnapStart lautet`Active`, dann ist aktiviert und ein Snapshot ist für die angegebene Funktionsversion verfügbar.

```
"SnapStart": {
```

```
    "ApplyOn": "PublishedVersions",
    "OptimizationStatus": "On"
  },
  "State": "Active",
```

4. Rufen Sie die Funktionsversion auf, indem Sie den Aufruf-Befehl ausführen und die Version angeben. Das folgende Beispiel ruft Version 1 auf.

```
aws lambda invoke \
  --cli-binary-format raw-in-base64-out \
  --function-name my-function:1 \
  --payload '{ "name": "Bob" }' \
  response.json
```

Die cli-binary-format Option ist erforderlich, wenn Sie AWS CLI Version 2 verwenden. Um dies zur Standardeinstellung zu machen, führen Sie `aws configure set cli-binary-format raw-in-base64-out` aus. Weitere Informationen finden Sie unter Von AWS CLI unterstützte globale Befehlszeilenoptionen im AWS Command Line Interface -Benutzerhandbuch für Version 2.

Aktivierung SnapStart (API)

Um zu aktivieren SnapStart

1. Führen Sie eine der folgenden Aktionen aus:
   - Erstellen Sie eine neue Funktion mit SnapStart aktiviertem Wert, indem Sie die CreateFunctionAPI-Aktion mit dem SnapStartParameter verwenden.
   - Aktivieren Sie sie SnapStart für eine vorhandene Funktion, indem Sie die UpdateFunctionConfigurationAktion mit dem SnapStartParameter verwenden.
2. Veröffentlichen Sie eine Funktionsversion mit der PublishVersionAktion. Lambda initialisiert Ihren Code, erstellt einen Snapshot der initialisierten Ausführungsumgebung und speichert den Snapshot im Cache für einen Zugriff mit geringer Latenz.
3. Bestätigen Sie mithilfe der GetFunctionConfigurationAktion, dass sie für die Funktionsversion aktiviert SnapStart ist. Geben Sie eine Versionsnummer an, um zu bestätigen, dass die Version für diese Version aktiviert SnapStart ist. Wenn in der Antwort angezeigt OptimizationStatuswird, dass dies der Fall ist On und der Status SnapStart lautet`Active`, dann ist aktiviert und ein Snapshot ist für die angegebene Funktionsversion verfügbar.

```
"SnapStart": {
        "ApplyOn": "PublishedVersions",
        "OptimizationStatus": "On"
    },
    "State": "Active",
```

4. Rufen Sie die Funktionsversion mit der Aufruf-Aktion auf.

Lambda SnapStart und Funktionszustände

Die folgenden Funktionszustände können auftreten, wenn Sie verwenden SnapStart.

Ausstehend

Lambda initialisiert Ihren Code und erstellt einen Snapshot der initialisierten Ausführungsumgebung. Alle Aufrufe oder andere API-Aktionen, die auf der Funktionsversion ausgeführt werden, schlagen fehl.

Aktiv

Die Snapshot-Erstellung ist abgeschlossen und Sie können die Funktion aufrufen. Zur Verwendung SnapStart müssen Sie die veröffentlichte Funktionsversion aufrufen, nicht die unveröffentlichte Version ($LATEST).

Inaktiv

Der Status `Inactive` kann auftreten, wenn Lambda regelmäßig Funktions-Snapshots neu generiert, um Software-Updates anzuwenden. In diesem Fall kann Ihre Funktion in einen `Inactive`-Zustand übergehen, wenn sie nicht initialisiert werden kann.

Bei Funktionen, die eine Java-Laufzeit verwenden, löscht Lambda Snapshots nach 14 Tagen ohne Aufruf. Wenn Sie die Funktionsversion nach 14 Tagen aufrufen, gibt Lambda eine `SnapStartNotReadyException`-Antwort zurück und beginnt mit der Initialisierung eines neuen Snapshots. Warten Sie, bis die Funktionsversion den `Active`-Status erreicht hat, und rufen Sie es dann erneut auf.

Fehlgeschlagen

Lambda hat beim Ausführen des Initialisierungscodes oder beim Erstellen des Snapshots einen Fehler festgestellt.

Aktualisieren eines Snapshots

Lambda erstellt für jede veröffentlichte Funktionsversion einen Snapshot. Um einen Snapshot zu aktualisieren, veröffentlichen Sie eine neue Funktionsversion.

SnapStart Verwenden mit AWS SDKs

Um AWS SDK-Aufrufe von Ihrer Funktion aus durchzuführen, generiert Lambda einen kurzlebigen Satz von Anmeldeinformationen, indem es die Ausführungsrolle Ihrer Funktion übernimmt. Diese Anmeldeinformationen sind während des Aufrufs Ihrer Funktion als Umgebungsvariablen verfügbar. Sie müssen keine Anmeldeinformationen für das SDK direkt im Code bereitstellen. Standardmäßig überprüft die Kette der Anbieter von Anmeldeinformationen nacheinander jeden Ort, an dem Sie Anmeldeinformationen festlegen können, und wählt die erste verfügbare aus – in der Regel die Umgebungsvariablen (`AWS_ACCESS_KEY_ID`, `AWS_SECRET_ACCESS_KEY` und `AWS_SESSION_TOKEN`).

> **Note**
>
> Wenn SnapStart aktiviert, verwendet die Lambda-Laufzeit automatisch die Container-Anmeldeinformationen (`AWS_CONTAINER_CREDENTIALS_FULL_URI`und`AWS_CONTAINER_AUTHORIZATION_TOKEN`) anstelle der Umgebungsvariablen für die Zugriffsschlüssel. Dadurch wird verhindert, dass Anmeldeinformationen ablaufen, bevor die Funktion wiederhergestellt wird.

Verwenden Sie SnapStart mit CloudFormationAWS SAM, und AWS CDK

- AWS CloudFormation: Deklarieren Sie die SnapStartEntität in Ihrer Vorlage.
- AWS Serverless Application Model (AWS SAM): Deklarieren Sie die SnapStartEigenschaft in Ihrer Vorlage.
- AWS Cloud Development Kit (AWS CDK): Verwenden Sie den SnapStartPropertyTyp.

Löschen von Snapshots

Lambda löscht Snapshots, wenn:

- Sie die Funktion oder Funktionsversion löschen.

- Nur Java-Laufzeiten – Sie die Funktionsversion 14 Tage lang nicht aufrufen. Wenn nach 14 Tagen ohne Aufruf die Funktionsversion in den Status Inaktiv übergeht. Wenn Sie die Funktionsversion nach 14 Tagen aufrufen, gibt Lambda eine `SnapStartNotReadyException`-Antwort zurück und beginnt mit der Initialisierung eines neuen Snapshots. Warten Sie, bis die Funktionsversion den Status Aktiv erreicht hat, und rufen Sie es dann erneut auf.

Lambda entfernt alle Ressourcen, die mit gelöschten Snapshots verknüpft sind, gemäß der Allgemeinen Datenschutzverordnung (DSGVO).

Handhabung der Eindeutigkeit mit Lambda SnapStart

Wenn Aufrufe für eine SnapStart-Funktion hochskaliert werden, verwendet Lambda einen einzelnen initialisierten Snapshot, um mehrere Ausführungsumgebungen fortzusetzen. Wenn Ihr Initialisierungscode eindeutigen Inhalt generiert, der im Snapshot enthalten ist, ist der Inhalt möglicherweise nicht eindeutig, wenn er in Ausführungsumgebungen wiederverwendet wird. Um die Eindeutigkeit bei der Verwendung von SnapStart zu gewährleisten, müssen Sie nach der Initialisierung eindeutige Inhalte erstellen. Dazu gehören eindeutige IDs, eindeutige Geheimnisse und Entropie, die zum Generieren von Pseudozufälligkeiten verwendet wird.

Im Folgenden finden Sie einige bewährte Methoden für die Aufrechterhaltung der Eindeutigkeit Ihres Codes. Für Java-Funktionen bietet Lambda auch ein Open-Source-Scan-Tool SnapStart, mit dem sich überprüfen lässt, ob der Code Eindeutigkeit voraussetzt. Wenn Sie während der Initialisierungsphase eindeutige Daten generieren, können Sie eine Laufzeit-Hook verwenden, um die Eindeutigkeit wiederherzustellen. Mit Laufzeit-Hooks können Sie bestimmten Code ausführen, unmittelbar bevor Lambda einen Snapshot aufnimmt oder unmittelbar nachdem Lambda eine Funktion aus einem Snapshot fortsetzt.

Vermeiden des Speicherns eines Zustands, der von der Eindeutigkeit während der Initialisierung abhängt

Vermeiden Sie während der Initialisierungsphase Ihrer Funktion das Zwischenspeichern von Daten, die eindeutig sein sollen, z. B. das Generieren einer eindeutigen ID für die Protokollierung oder das Setzen von Seeds für Zufallsfunktionen. Stattdessen empfehlen wir Ihnen, eindeutige Daten zu generieren oder Seeds für Zufallsfunktionen in Ihrem Funktionshandler festzulegen – oder einen Laufzeit-Hook zu verwenden.

Die folgenden Beispiele zeigen, wie eine UUID im Funktionshandler erzeugt werden kann.

Java

Example – Generierung einer eindeutigen ID im Funktionshandler

```
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 – Generierung einer eindeutigen ID im Funktionshandler

```
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 – Generierung einer eindeutigen ID im Funktionshandler

```
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}";
    }
}
```

Kryptografisch sichere Pseudozufallszahlengeneratoren (CSPRNGs) verwenden

Wenn Ihre Anwendung von Zufälligkeit abhängt, empfehlen wir Ihnen, kryptographisch sichere Zufallszahlengeneratoren (CSPRNGs) zu verwenden. Zusätzlich zu OpenSSL 1.0.2 enthalten die verwalteten Lambda-Laufzeiten auch die folgenden integrierten CSPRNGs:

- Java: `java.security.SecureRandom`
- Python: `random.SystemRandom`
- .NET: `System.Security.Cryptography.RandomNumberGenerator`

Software, die immer Zufallszahlen von `/dev/random` oder `/dev/urandom` erhält, behält auch die Zufälligkeit in SnapStart bei.

AWS-Kryptographie-Bibliotheken erhalten automatisch die Zufälligkeit mit SnapStart ab den in der folgenden Tabelle angegebenen Mindestversionen. Wenn Sie diese Bibliotheken mit Ihren Lambda-Funktionen verwenden, stellen Sie sicher, dass Sie die folgenden Mindestversionen oder neuere Versionen verwenden:

Bibliothek	Unterstützte Mindestversion (x86)	Unterstützte Mindestversion (ARM)
AWS libcrypto (AWS-LC)	1.16.0	1.30.0

Bibliothek	Unterstützte Mindestversion (x86)	Unterstützte Mindestversion (ARM)
AWS libcrypto FIPS	2.0.13	2.0.13

Wenn Sie die vorangehenden kryptografischen Bibliotheken mit Ihren Lambda-Funktionen als transitive Abhängigkeiten über die folgenden Bibliotheken paketieren, stellen Sie sicher, dass Sie die folgenden Mindestversionen oder neuere Versionen verwenden:

Bibliothek	Unterstützte Mindestversion (x86)	Unterstützte Mindestversion (ARM)
AWS SDK for Java 2.x	2.23.20	2.26.12
AWS Allgemeine Laufzeit für Java	0.29.8	0.29.25
Kryptoanbieter Amazon Corretto	2.4.1	2.4.1
Amazon Corretto Kryptoanbieter FIPS	2.4.1	2.4.1

Die folgenden Beispiele zeigen, wie CSPRNGs verwendet werden können, um eindeutige Zahlenfolgen zu garantieren, selbst wenn die Funktion aus einem Snapshot wiederhergestellt wird.

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!";
    }
}
```

SnapStart-Scan-Tool (nur Java)

Lambda bietet ein Scan-Tool für Java an, um nach Code zu suchen, der Eindeutigkeit voraussetzt. Das SnapStart-Scan-Tool ist ein Open-Source-SpotBugs-Plugin, das eine statische Analyse anhand einer Reihe von Regeln durchführt. Das Scan-Tool hilft dabei, potenzielle Codeimplementierungen zu identifizieren, die Annahmen bezüglich der Eindeutigkeit widerlegen könnten. Installationsanweisungen und eine Liste der vom Scan-Tool durchgeführten Prüfungen finden Sie im aws-lambda-snapstart-java-rules-Repository auf GitHub.

Weitere Informationen zum Umgang mit Einzigartigkeit mit SnapStart finden Sie im AWS-Compute-Blog unter Schnelleres Starten mit AWS Lambda SnapStart.

Implementieren Sie Code vor oder nach Lambda-Funktions-Snapshots

Sie können Laufzeit-Hooks verwenden, um Code zu implementieren, bevor Lambda einen Snapshot erstellt oder nachdem Lambda eine Funktion aus einem Snapshot fortsetzt. Laufzeit-Hooks sind für eine Vielzahl von Zwecken nützlich, z. B.:

- Bereinigung und Initialisierung: Bevor ein Snapshot erstellt wird, können Sie einen Laufzeit-Hook verwenden, um Bereinigungs- oder Ressourcenfreigabeoperationen durchzuführen. Nachdem ein Snapshot wiederhergestellt wurde, können Sie einen Laufzeit-Hook verwenden, um alle Ressourcen oder Zustände, die nicht im Snapshot erfasst wurden, neu zu initialisieren.
- Dynamische Konfiguration: Sie können Laufzeit-Hooks verwenden, um die Konfiguration oder andere Metadaten dynamisch zu aktualisieren, bevor ein Snapshot erstellt wird oder nachdem er wiederhergestellt wurde. Dies kann nützlich sein, wenn sich Ihre Funktion an Änderungen in der Laufzeitumgebung anpassen muss.
- Externe Integrationen: Sie können Laufzeit-Hooks zur Integration mit externen Diensten oder Systemen verwenden, z. B. zum Senden von Benachrichtigungen oder zum Aktualisieren des externen Status als Teil des Checkpointing- und Wiederherstellungsprozesses.
- Leistungsoptimierung: Sie können Laufzeit-Hooks verwenden, um die Startsequenz Ihrer Funktion fein abzustimmen, z. B. durch Vorladen von Abhängigkeiten. Weitere Informationen finden Sie unter Leistungsoptimierung.

Auf den folgenden Seiten wird erläutert, wie Sie Laufzeit-Hooks für Ihre bevorzugte Laufzeitumgebung implementieren.

Themen

- Lambda SnapStart-Laufzeit-Hooks für Java
- Lambda SnapStart-Laufzeit-Hooks für Python
- Lambda SnapStart-Laufzeit-Hooks für .NET

Lambda SnapStart-Laufzeit-Hooks für Java

Sie können Laufzeit-Hooks verwenden, um Code zu implementieren, bevor Lambda einen Snapshot erstellt oder nachdem Lambda eine Funktion aus einem Snapshot fortsetzt. aufzeit-Hooks sind als Teil des Open-Source-Projekts Coordinated Restore at Checkpoint (CRaC) verfügbar. CRaC wird für

das Open-Java-Entwicklungs-Kit (OpenJDK) entwickelt. Ein Beispiel für die Verwendung von CRaC mit einer Referenzanwendung finden Sie im CRaC-Verzeichnis auf GitHub. CRaC verwendet drei Hauptelemente:

- `Resource` – Eine Schnittstelle mit zwei Methoden, `beforeCheckpoint()` und `afterRestore()`. Verwenden Sie diese Methoden, um den Code zu implementieren, den Sie vor einem Snapshot und nach einer Wiederherstellung ausführen möchten.
- `Context <R extends Resource>` – Um Benachrichtigungen für Prüfpunkte und Wiederherstellungen zu erhalten, muss eine `Resource` mit einem `Context` registriert sein.
- `Core` – Der Koordinationsservice, der das standardmäßige globale `Context` über die statische Methode `Core.getGlobalContext()` bereitstellt.

Weitere Informationen zum `Context` und `Resource` finden Sie unter Paket org.crac in der CRaC-Dokumentation.

Gehen Sie wie folgt vor, um Laufzeit-Hooks mit dem Paket org.crac zu implementieren. Die Lambda-Laufzeit enthält eine angepasste CRaC-Kontextimplementierung, die Ihre Laufzeit-Hooks vor dem Checkpointing und nach der Wiederherstellung aufruft.

Registrierung und Ausführung von Laufzeit-Hooks

Die Reihenfolge, in der Lambda Ihre Laufzeit-Hooks ausführt, wird durch die Reihenfolge der Registrierung bestimmt. Die Reihenfolge der Registrierung entspricht der Reihenfolge des Imports, der Definition oder der Ausführung in Ihrem Code.

- `beforeCheckpoint()`: Wird in umgekehrter Reihenfolge der Registrierung ausgeführt
- `afterRestore()`: Wird in Reihenfolge der Registrierung ausgeführt

Vergewissern Sie sich, dass alle registrierten Hooks ordnungsgemäß importiert und in den Code Ihrer Funktion aufgenommen wurden. Wenn Sie Laufzeit-Hooks in einer separaten Datei oder einem Modul registrieren, müssen Sie sicherstellen, dass das Modul entweder direkt oder als Teil eines größeren Pakets in der Handler-Datei Ihrer Funktion importiert wird. Wenn die Datei oder das Modul nicht in den Funktionshandler importiert wird, ignoriert Lambda die Laufzeithaken.

> **Note**
>
> Wenn Lambda einen Snapshot erstellt, kann Ihr Initialisierungscode bis zu 15 Minuten lang ausgeführt werden. Das Zeitlimit beträgt 130 Sekunden oder das konfigurierte

Funktions-Timeout (maximal 900 Sekunden), je nachdem, welcher Wert höher ist. Ihre `beforeCheckpoint()`-Laufzeit-Hooks werden auf das Zeitlimit des Initialisierungscodes angerechnet. Wenn Lambda einen Snapshot wiederherstellt, muss die Laufzeit geladen werden und `afterRestore()`-Laufzeit-Hooks müssen innerhalb des Timeout-Limits (10 Sekunden) abgeschlossen werden. Andernfalls erhalten Sie eine SnapStartTimeoutException.

Schritt 1: Aktualisieren der Entwicklungskonfiguration

Fügen Sie die `org.crac`-Abhängigkeit zur Entwicklungskonfiguration hinzu. Im folgenden Beispiel wird Gradle verwendet. Beispiele für andere Entwicklungssysteme finden Sie in der Apache-Maven-Dokumentation.

```
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'
}
```

Schritt 2: Aktualisieren des Lambda-Handlers

Der Lambda-Funktionshandler ist die Methode in Ihrem Funktionscode, die Ereignisse verarbeitet. Wenn Ihre Funktion aufgerufen wird, führt Lambda die Handler-Methode aus. Ihre Funktion wird so lange ausgeführt, bis der Handler eine Antwort zurückgibt, beendet wird oder ein Timeout auftritt.

Weitere Informationen finden Sie unter Definieren des Lambda-Funktions-Handlers in Java.

Der folgende Beispielhandler zeigt, wie Code vor Checkpointing (`beforeCheckpoint()`) und nach der Wiederherstellung (`afterRestore()`) ausgeführt wird. Dieser Handler registriert auch die `Resource` im Laufzeit-verwalteten globalen `Context`.

> **Note**
>
> Wenn Lambda einen Snapshot erstellt, kann Ihr Initialisierungscode bis zu 15 Minuten lang ausgeführt werden. Das Zeitlimit beträgt 130 Sekunden oder das konfigurierte Funktions-Timeout (maximal 900 Sekunden), je nachdem, welcher Wert höher ist. Ihre

`beforeCheckpoint()`-Laufzeit-Hooks werden auf das Zeitlimit des Initialisierungscodes angerechnet. Wenn Lambda einen Snapshot wiederherstellt, muss die Laufzeit (JVM) geladen werden und `afterRestore()`-Laufzeit-Hooks müssen innerhalb des Timeout-Limits (10 Sekunden) abgeschlossen werden. Andernfalls erhalten Sie eine 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");
```

Der `Context` verwaltet nur ein `WeakReference` auf dem registrierten Objekt. Wenn für eine `Resource` eine Garbage Collection durchgeführt wird, werden Laufzeit-Hooks nicht ausgeführt. Ihr Code muss einen starken Verweis auf die `Resource` beibehalten, um sicherzustellen, dass der Laufzeit-Hook ausgeführt wird.

Hier sind zwei Beispiele für zu vermeidende Muster:

Example – Objekt ohne starke Referenz

```
Core.getGlobalContext().register( new MyResource() );
```

Example – Objekte anonymer Klassen

```
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 {
    // ...
   }

} );
```

Behalten Sie stattdessen eine starke Referenz bei. Im folgenden Beispiel wird die registrierte Ressource nicht von der Garbage Collection erfasst und Laufzeit-Hooks werden konsistent ausgeführt.

Example – Objekt mit starker Referenz

```
Resource myResource = new MyResource(); // This reference must be maintained to prevent
 the registered resource from being garbage collected
Core.getGlobalContext().register( myResource );
```

Lambda SnapStart-Laufzeit-Hooks für Python

Sie können Laufzeit-Hooks verwenden, um Code zu implementieren, bevor Lambda einen Snapshot erstellt oder nachdem Lambda eine Funktion aus einem Snapshot fortsetzt. Python-Laufzeit-Hooks sind als Teil der Open-Source-Bibliothek Snapshot Restore für Python verfügbar, die in den von Python verwalteten Laufzeiten enthalten ist. Diese Bibliothek bietet zwei Dekoratoren, mit denen Sie Ihre Laufzeit-Hooks definieren können:

- `@register_before_snapshot`: Für Funktionen, die Sie ausführen möchten, bevor Lambda einen Snapshot erstellt.
- `@register_after_restore`: Für Funktionen, die Sie ausführen möchten, wenn Lambda eine Funktion aus einem Snapshot wieder aufnimmt.

Alternativ können Sie auch die folgenden Methoden verwenden, um Callables für Laufzeit-Hooks zu registrieren:

- `register_before_snapshot(func, *args, **kwargs)`
- `register_after_restore(func, *args, **kwargs)`

Registrierung und Ausführung von Laufzeit-Hooks

Die Reihenfolge, in der Lambda Ihre Laufzeit-Hooks ausführt, wird durch die Reihenfolge der Registrierung bestimmt:

- Vor dem Snapshot: Wird in umgekehrter Reihenfolge der Registrierung ausgeführt
- Nach dem Snapshot: Wird in Reihenfolge der Registrierung ausgeführt

Die Reihenfolge der Registrierung von Hooks zur Laufzeit hängt davon ab, wie Sie die Hooks definieren. Wenn Sie Dekoratoren (`@register_before_snapshot` und `@register_after_restore`) verwenden, folgt die Reihenfolge der Registrierung der Reihenfolge des Imports, der Definition oder der Ausführung in Ihrem Code. Wenn Sie mehr Kontrolle über die Registrierungsreihenfolge benötigen, verwenden Sie die `register_before_snapshot()`- und `register_after_restore()`-Methoden anstelle von Dekoratoren.

Vergewissern Sie sich, dass alle registrierten Hooks ordnungsgemäß importiert und in den Code Ihrer Funktion aufgenommen wurden. Wenn Sie Laufzeit-Hooks in einer separaten Datei oder einem Modul registrieren, müssen Sie sicherstellen, dass das Modul entweder direkt oder als Teil eines größeren Pakets in der Handler-Datei Ihrer Funktion importiert wird. Wenn die Datei oder das Modul nicht in den Funktionshandler importiert wird, ignoriert Lambda die Laufzeithaken.

> **Note**
>
> Wenn Lambda einen Snapshot erstellt, kann Ihr Initialisierungscode bis zu 15 Minuten lang ausgeführt werden. Das Zeitlimit beträgt 130 Sekunden oder das konfigurierte Funktions-Timeout (maximal 900 Sekunden), je nachdem, welcher Wert höher ist. Ihre `@register_before_snapshot`-Laufzeit-Hooks werden auf das Zeitlimit des Initialisierungscodes angerechnet. Wenn Lambda einen Snapshot wiederherstellt, muss die Laufzeit geladen werden und `@register_after_restore`-Laufzeit-Hooks müssen innerhalb des Timeout-Limits (10 Sekunden) abgeschlossen werden. Andernfalls erhalten Sie eine SnapStartTimeoutException.

Beispiel

Der folgende Beispielhandler zeigt, wie Code vor Checkpointing (`@register_before_snapshot`) und nach der Wiederherstellung (`@register_after_restore`) ausgeführt wird.

```
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
```

Weitere Beispiele finden Sie unter Snapshot Restore für Python im AWS GitHub-Repository.

Lambda SnapStart-Laufzeit-Hooks für .NET

Sie können Laufzeit-Hooks verwenden, um Code zu implementieren, bevor Lambda einen Snapshot erstellt oder nachdem Lambda eine Funktion aus einem Snapshot fortsetzt. .NET-Laufzeit-Hooks sind als Teil des Amazon.Lambda.Core-Pakets (Version 2.5.0 oder höher) verfügbar. Diese Bibliothek bietet zwei Methoden, mit denen Sie Ihre Laufzeit-Hooks definieren können:

- `RegisterBeforeSnapshot()`: Code, der vor der Snapshot-Erstellung ausgeführt werden soll
- `RegisterAfterSnapshot()`: Code, der nach der Wiederaufnahme einer Funktion aus einem Snapshot ausgeführt wird

> **Note**
>
> Wenn Sie das Lambda Annotations Framework für .NET verwenden, führen Sie ein Upgrade auf Amazon.Lambda.Annotations Version 1.6.0 oder höher durch, um die Kompatibilität mit SnapStart sicherzustellen.

Registrierung und Ausführung von Laufzeit-Hooks

Registrieren Sie Ihre Hooks in Ihrem Initialisierungscode. Beachten Sie die folgenden Richtlinien, die auf dem Ausführungsmodell Ihrer Lambda-Funktion basieren:

- Für den Ansatz der ausführbaren Assemblierung müssen Sie Ihre Hooks registrieren, bevor Sie den Lambda-Bootstrap mit `RunAsync` starten.
- Für den Ansatz der Klassenbibliothek registrieren Sie Ihre Hooks im Konstruktor der Handler-Klasse.
- Für ASP.NET Core-Anwendungen registrieren Sie Ihre Hooks, bevor Sie die `WebApplications.Run`-Methode aufrufen.

Um Laufzeit-Hooks für SnapStart in .NET zu registrieren, verwenden Sie die folgenden Methoden:

```
Amazon.Lambda.Core.SnapshotRestore.RegisterBeforeSnapshot(BeforeCheckpoint);
Amazon.Lambda.Core.SnapshotRestore.RegisterAfterRestore(AfterCheckpoint);
```

Wenn mehrere Hook-Typen registriert sind, wird die Reihenfolge, in der Lambda Ihre Laufzeit-Hooks ausführt, durch die Reihenfolge der Registrierung bestimmt:

- `RegisterBeforeSnapshot()`: Wird in umgekehrter Reihenfolge der Registrierung ausgeführt
- `RegisterAfterSnapshot()`: Wird in Reihenfolge der Registrierung ausgeführt

> **Note**
>
> Wenn Lambda einen Snapshot erstellt, kann Ihr Initialisierungscode bis zu 15 Minuten lang ausgeführt werden. Das Zeitlimit beträgt 130 Sekunden oder das konfigurierte Funktions-Timeout (maximal 900 Sekunden), je nachdem, welcher Wert höher ist. Ihre `RegisterBeforeSnapshot()`-Laufzeit-Hooks werden auf das Zeitlimit des Initialisierungscodes angerechnet. Wenn Lambda einen Snapshot wiederherstellt, muss die Laufzeit geladen werden und `RegisterAfterSnapshot()`-Laufzeit-Hooks müssen innerhalb des Timeout-Limits (10 Sekunden) abgeschlossen werden. Andernfalls erhalten Sie eine SnapStartTimeoutException.

Beispiel

Die folgende Beispielfunktion zeigt, wie Code vor Checkpointing (`RegisterBeforeSnapshot`) und nach der Wiederherstellung (`RegisterAfterRestore`) ausgeführt wird.

```
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
        };
    }
}
```

Überwachung für Lambda SnapStart

Sie können Ihre Lambda-SnapStart-Funktionen mit Amazon CloudWatch, AWS X-Ray und der Zugriff auf Echtzeit-Telemetriedaten für Erweiterungen über die Telemetrie-API überwachen.

> **Note**
>
> Die Umgebungsvariablen `AWS_LAMBDA_LOG_GROUP_NAME` und `AWS_LAMBDA_LOG_STREAM_NAME` sind in den Lambda-SnapStart-Funktionen nicht verfügbar.

Verstehen des Protokollierungs- und Abrechnungsverhaltens mit SnapStart

Es gibt einige Unterschiede zum CloudWatch-Protokollstream-Format für SnapStart-Funktionen:

- Initialisierungsprotokolle – Wenn eine neue Ausführungsumgebung erstellt wird, enthält `REPORT` das Feld `Init Duration` nicht. Das liegt daran, dass Lambda SnapStart-Funktionen initialisiert, wenn Sie eine Version erstellen und nicht während des Funktionsaufrufs. Für SnapStart-Funktionen befindet sich das `Init Duration`-Feld im `INIT_REPORT`-Datensatz. Dieser Datensatz zeigt die Dauer von Init-Phase, einschließlich der Dauer aller `beforeCheckpoint`-Laufzeit-Hooks an.
- Aufrufprotokolle – Wenn eine neue Ausführungsumgebung erstellt wird, enthält `REPORT` die Felder `Restore Duration` und `Billed Restore Duration`:
  - `Restore Duration`: Die Zeit, die Lambda benötigt, um einen Snapshot wiederherzustellen, die Laufzeit zu laden und ggf. Laufzeit-Hooks nach der Wiederherstellung auszuführen. Der Prozess der Wiederherstellung von Snapshots kann Zeit beinhalten, die für Aktivitäten außerhalb der MicroVM aufgewendet wird. Diese Zeit wird in erfasst `Restore Duration`.
  - `Billed Restore Duration`: Die Zeit, die Lambda benötigt, um die Laufzeit zu laden und ggf. Laufzeit-Hooks nach der Wiederherstellung auszuführen.

> **Note**
>
> Wie bei allen Lambda-Funktionen fallen Dauergebühren für Code an, der im Funktionshandler ausgeführt wird. Bei SnapStart-Funktionen gelten die Gebühren für die Dauer auch für Initialisierungscode, der außerhalb des Handlers deklariert wird, für die

Zeit, die die Laufzeit zum Laden benötigt, und für jeden Code, der in einem Laufzeit-Hook ausgeführt wird.

Die Kaltstartdauer ist die Summe von `Restore Duration` + `Duration`.

Das folgende Beispiel ist eine Lambda-Insights-Abfrage, die die Latenzperzentile für SnapStart-Funktionen zurückgibt. Weitere Informationen zu Lambda-Insights-Abfragen finden Sie unter Beispiel-Workflow mit Abfragen zur Fehlerbehebung einer Funktion.

```
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
```

Aktive X-Ray-Ablaufverfolgung für SnapStart

Sie können X-Ray verwenden, um Anforderungen an Lambda-SnapStart-Funktionen nachzuverfolgen. Bei den X-Ray-Teilsegmenten für SnapStart-Funktionen gibt es einige Unterschiede:

- Es gibt kein `Initialization`-Teilsegment für SnapStart-Funktionen.
- Das Teilsegment `Restore` zeigt die Zeit an, die Lambda benötigt, um einen Snapshot wiederherzustellen, die Laufzeit zu laden und vorhandene Laufzeit-Hooks nach der Wiederherstellung auszuführen. Der Prozess der Wiederherstellung von Snapshots kann Zeit beinhalten, die für Aktivitäten außerhalb der MicroVM aufgewendet wird. Diese Zeit wird im `Restore`-Untersegment erfasst. Die Zeit, die Sie außerhalb der microVM für die Wiederherstellung eines Snapshots aufwenden, wird Ihnen nicht in Rechnung gestellt.

Telemetrie-API-Ereignisse für SnapStart

Lambda sendet die folgenden SnapStart-Ereignisse an Telemetrie-API:

- `platform.restoreStart` – Zeigt die Zeit an, zu der die `Restore`-Phase gestartet wurde.
- `platform.restoreRuntimeDone` – Zeigt an, ob die `Restore`-Phase erfolgreich war. Lambda sendet diese Nachricht, wenn die Laufzeit eine `restore/next`-Laufzeit-API-Anfrage sendet. Es gibt drei mögliche Status: erfolgreich, fehlgeschlagen und Timeout.
- `platform.restoreReport` – Zeigt an, wie lange die `Restore`-Phase gedauert hat und wie viele Millisekunden Ihnen während dieser Phase in Rechnung gestellt wurden.

Amazon-API-Gateway- und Funktions-URL-Metriken

Wenn Sie eine Web-API mit API-Gateway erstellen, können Sie die IntegrationLatency-Metrik verwenden, um die durchgehende Latenz zu messen (die Zeit zwischen der Weiterleitung einer Anfrage an das Backend durch API-Gateway und dem Empfang einer Antwort vom Backend.

Wenn Sie eine Lambda-Funktions-URL verwenden, können Sie die UrlRequestLatency-Metrik verwenden, um die durchgehende Latenz zu messen (die Zeit zwischen dem Empfang einer Anfrage durch die Funktions-URL und der Rückgabe einer Antwort durch die Funktions-URL).

Sicherheitsmodell für Lambda SnapStart

Lambda SnapStart unterstützt die Verschlüsselung im Ruhezustand. Lambda verschlüsselt Snapshots mit einem AWS KMS key. Standardmäßig verwendet Lambda einen Von AWS verwalteter Schlüssel. Wenn dieses Standardverhalten für Ihren Workflow geeignet ist, müssen Sie nichts weiter einrichten. Andernfalls können Sie die `--kms-key-arn`-Option im create-function- oder update-function-configuration-Befehl verwenden, um einen vom AWS KMS-Kunden verwalteten Schlüssel bereitzustellen. Sie können dies tun, um die Rotation des KMS-Schlüssels zu steuern oder um die Anforderungen Ihrer Organisation für die Verwaltung von KMS-Schlüsseln zu erfüllen. Für vom Kunden verwaltete Schlüssel fallen Standard-AWS KMS-Gebühren an. Weitere Informationen finden Sie unter AWS Key Management Service Preise.

Wenn Sie eine SnapStart-Funktion oder Funktionsversion löschen, schlagen alle `Invoke`-Anfragen an diese Funktion oder Funktionsversion fehl. Lambda entfernt alle Ressourcen, die mit gelöschten Snapshots verknüpft sind, gemäß der Allgemeinen Datenschutzverordnung (DSGVO).

Maximieren Sie die Leistung von Lambda SnapStart

Themen

- Leistungsoptimierung
- Bewährte Praktiken für das Netzwerk

Leistungsoptimierung

Um die Vorteile von SnapStart zu maximieren, sollten Sie die folgenden Empfehlungen zur Codeoptimierung für Ihre Laufzeit berücksichtigen.

> **Note**
>
> SnapStart funktioniert am besten, wenn es mit Funktionsaufrufen im großen Maßstab verwendet wird. Funktionen, die selten aufgerufen werden, weisen möglicherweise nicht dieselben Leistungsverbesserungen auf.

Java

Um die Vorteile von SnapStart zu maximieren, empfehlen wir, Abhängigkeiten vorzuladen und Ressourcen, die zur Startlatenz beitragen, in Ihrem Initialisierungscode und nicht im Funktionshandler zu initialisieren. Dadurch wird die Latenz, die mit dem Laden umfangreicher Klassen verbunden ist, aus dem Aufrufpfad entfernt und die Startleistung mit SnapStart optimiert.

Wenn Sie Abhängigkeiten oder Ressourcen nicht während der Initialisierung vorladen können, empfehlen wir Ihnen, sie mit Dummy-Aufrufen vorzuladen. Aktualisieren Sie dazu den Funktions-Handler-Code, wie im folgenden Beispiel aus der Pet-Store-Funktion im AWS-Labs von GitHub-Repository gezeigt.

```
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

Um die Vorteile von SnapStart zu maximieren, sollten Sie sich auf eine effiziente Codeorganisation und Ressourcenverwaltung innerhalb Ihrer Python-Funktionen konzentrieren. Als allgemeine Richtlinie gilt, dass Sie während der Initialisierungsphase umfangreiche Rechenaufgaben ausführen sollten. Mit diesem Ansatz werden zeitaufwändige Operationen aus dem Aufrufpfad

herausgenommen, was die Gesamtleistung der Funktion verbessert. Um diese Strategie wirksam umzusetzen, empfehlen wir die folgenden bewährten Verfahren:

- Import von Abhängigkeiten außerhalb des Funktionshandlers.
- Erstellen Sie `boto3`-Instances außerhalb des Handlers.
- Initialisieren Sie statische Ressourcen oder Konfigurationen, bevor der Handler aufgerufen wird.
- Erwägen Sie die Verwendung eines Laufzeit-Hooks vor dem Snapshot für ressourcenintensive Aufgaben wie das Herunterladen externer Dateien, das Vorladen von Frameworks wie Django oder das Laden von Machine-Learning-Modellen.

Example – Optimieren Sie die Python-Funktion für 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

Um die Just-in-Time-Kompilierung (JIT) und das Laden von Assemblys zu reduzieren, sollten Sie erwägen, Ihren Funktionshandler von einem `RegisterBeforeCheckpoint` Laufzeit-Hook aus aufzurufen. Aufgrund der Funktionsweise von .NET Tiered Compilation erhalten Sie optimale Ergebnisse, wenn Sie den Handler mehrfach aufrufen, wie im folgenden Beispiel gezeigt.

> **Important**
>
> Vergewissern Sie sich, dass Ihr Dummy-Funktionsaufruf keine unbeabsichtigten Nebeneffekte hervorruft, wie z. B. das Auslösen von Geschäftstransaktionen.

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);
    }
}
```

Bewährte Praktiken für das Netzwerk

Der Zustand der Verbindungen, die Ihre Funktion während der Initialisierungsphase erstellt, wird nicht garantiert, wenn Lambda Ihre Funktion aus einem Snapshot wieder aufnimmt. In den meisten Fällen werden Netzwerkverbindungen, die ein AWS-SDK herstellt, automatisch fortgesetzt. Für andere Verbindungen empfehlen wir die folgenden bewährten Methoden.

Wiederherstellen von Netzwerkverbindungen

Stellen Sie Ihre Netzwerkverbindungen immer wieder her, wenn Ihre Funktion von einem Snapshot fortgesetzt wird. Wir empfehlen, dass Sie die Netzwerkverbindungen im Funktionshandler wiederherstellen. Alternativ können Sie ein After-Restore-Laufzeit-Hook verwenden.

Verwenden Sie den Hostnamen nicht als eindeutige ID der Ausführungsumgebung

Wir raten davon ab, `hostname` zu verwenden, um Ihre Ausführungsumgebung als eindeutigen Knoten oder Container in Ihren Anwendungen zu identifizieren. Mit SnapStart wird ein einziger Snapshot als Ausgangszustand für mehrere Ausführungsumgebungen verwendet. Alle Ausführungsumgebungen geben denselben `hostname`-Wert für `InetAddress.getLocalHost()` (Java), `socket.gethostname()` (Python) und `Dns.GetHostName()` (.NET) zurück. Für Anwendungen, die eine eindeutige Identität oder einen eindeutigen `hostname`-Wert der Ausführungsumgebung erfordern, empfehlen wir, im Funktionshandler eine eindeutige ID zu generieren. Oder verwenden Sie einen After-Restore-Laufzeit-Hook, um eine eindeutige ID zu generieren und verwenden Sie dann die eindeutige ID für die Ausführungsumgebung.

Vermeiden der Bindung von Verbindungen an feste Quellports

Wir empfehlen, Netzwerkverbindungen nicht an feste Quellports zu binden. Wenn eine Funktion nach einem Snapshot wieder aufgenommen wird, werden die Verbindungen neu aufgebaut, und Netzwerkverbindungen, die an einen festen Quellport gebunden sind, können fehlschlagen.

Vermeiden der Verwendung von Java-DNS-Cache

Lambda-Funktionen speichern DNS-Antworten bereits im Cache. Wenn Sie einen anderen DNS-Cache mit SnapStart verwenden, kann es zu Verbindungs-Timeouts kommen, wenn die Funktion von einem Snapshot aus fortgesetzt wird.

Die `java.util.logging.Logger`-Klasse kann indirekt den JVM-DNS-Cache aktivieren. Um die Standardeinstellungen zu überschreiben, setzen Sie networkaddress.cache.ttl auf 0, bevor Sie `logger` initialisieren. Beispiel:

```
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);
}
```

Um `UnknownHostException`-Ausfälle in der Java 11-Laufzeit zu vermeiden, empfehlen wir, `networkaddress.cache.negative.ttl` auf 0 zu setzen. In Java 17 und späteren Laufzeiten ist dieser Schritt nicht erforderlich. Sie können diese Eigenschaft für eine Lambda-Funktion mit der Umgebungsvariablen `AWS_LAMBDA_JAVA_NETWORKADDRESS_CACHE_NEGATIVE_TTL=0` festlegen.

Das Deaktivieren des JVM-DNS-Cache deaktiviert nicht das verwaltete DNS-Caching von Lambda.

Fehlerbehebung bei SnapStart-Fehlern für Lambda-Funktionen

Auf dieser Seite werden häufige Probleme behandelt, die bei der Verwendung von Lambda SnapStart auftreten, darunter Fehler bei der Snapshot-Erstellung, Zeitüberschreitungsfehler und interne Servicefehler.

SnapStartNotReadyException

Fehler: Beim Aufruf des Vorgangs Invoke20150331 ist ein Fehler aufgetreten (SnapStartNotReadyException): Lambda initialisiert gerade Ihre Funktion. Sie ist aufrufbereit, sobald der Status Ihrer Funktion auf AKTIV gesetzt wird.

Häufige Ursachen

Dieser Fehler tritt auf, wenn Sie versuchen, eine Funktionsversion aufzurufen, die sich im `Inactive` Status befindet. Die Version Ihrer Funktion wird zu `Inactive`, wenn sie 14 Tage lang nicht mehr aufgerufen wurde oder wenn Lambda die Ausführungsumgebung periodisch recycelt

Auflösung

Warten Sie, bis die Funktionsversion den `Active`-Status erreicht hat, und rufen Sie es dann erneut auf.

SnapStartTimeoutException

Problem: Sie erhalten ein `SnapStartTimeoutException`, wenn Sie versuchen, eine SnapStart-Funktionsversion aufzurufen.

Häufige Ursache

In der Wiederherstellungsphase stellt Lambda die Java-Laufzeit wieder her und führt alle Laufzeit-Hooks nach der Wiederherstellung aus. Wenn ein Laufzeit-Hook nach der Wiederherstellung länger als 10 Sekunden läuft, wird die `Restore`-Phase unterbrochen und Sie erhalten einen Fehler, wenn Sie versuchen, die Funktion aufzurufen. Probleme mit der Netzwerkverbindung und den Anmeldeinformationen können auch zu `Restore`-Phase-Timeouts führen.

Auflösung

Überprüfen Sie die CloudWatch-Protokolle der Funktion auf Timeoutfehler, die während der Wiederherstellungsphase aufgetreten sind. Stellen Sie sicher, dass alle Hooks nach der Wiederherstellung in weniger als 10 Sekunden abgeschlossen sind.

Example CloudWatch-Protokoll

```
{ "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"}
```

500 Interner Dienstfehler

Fehler: Lambda konnte keinen neuen Snapshot erstellen, weil Sie das Limit für die gleichzeitige Erstellung von Snapshots erreicht haben.

Häufige Ursache

Ein 500-Fehler ist ein interner Fehler innerhalb des Lambda-Service selbst und kein Problem mit Ihrer Funktion oder Ihrem Code. Diese Fehler treten häufig sporadisch auf.

Auflösung

Versuchen Sie erneut, die Funktionsversion zu veröffentlichen.

401 Nicht autorisiert

Fehler: Falsches Sitzungstoken oder Header-Schlüssel

Häufige Ursache

Dieser Fehler tritt auf, wenn der AWS Systems Manager-Parameter Store und die AWS Secrets Manager-Erweiterung mit Lambda SnapStart verwendet werden.

Auflösung

Der AWS Systems Manager-Parameter Store und die AWS Secrets Manager-Erweiterung sind nicht mit SnapStart kompatibel. Die Erweiterung generiert Anmeldeinformationen für die Kommunikation mit AWS Secrets Manager während der Funktionsinitialisierung, was bei der Verwendung mit SnapStart zu Fehlern wegen abgelaufener Anmeldeinformationen führt.

UnknownHostException (Java)

Fehler: HTTP-Anfrage kann nicht ausgeführt werden: Das Zertifikat für `abc.us-east-1.amazonaws.com` stimmt mit keinem der alternativen Namen des Betreffs überein.

Häufige Ursache

Lambda-Funktionen speichern DNS-Antworten bereits im Cache. Wenn Sie einen anderen DNS-Cache mit SnapStart verwenden, kann es zu Verbindungs-Timeouts kommen, wenn die Funktion von einem Snapshot aus fortgesetzt wird.

Auflösung

Um `UnknownHostException`-Ausfälle in der Java 11-Laufzeit zu vermeiden, empfehlen wir, `networkaddress.cache.negative.ttl` auf 0 zu setzen. In Java 17 und späteren Laufzeiten ist dieser Schritt nicht erforderlich. Sie können diese Eigenschaft für eine Lambda-Funktion mit der Umgebungsvariablen `AWS_LAMBDA_JAVA_NETWORKADDRESS_CACHE_NEGATIVE_TTL=0` festlegen.

Fehler bei der Erstellung von Snapshots

Fehler: AWS Lambda konnte Ihre SnapStart-Funktion nicht aufrufen. Wenn dieser Fehler weiterhin besteht, überprüfen Sie die CloudWatch-Protokolle Ihrer Funktion auf Initialisierungsfehler.

Auflösung

Überprüfen Sie die Amazon CloudWatch Logs Ihrer Funktion auf Zeitüberschreitungen beim Laufzeit-Hook vor dem Prüfpunkt. Sie können auch versuchen, eine neue Funktionsversion zu veröffentlichen, wodurch das Problem manchmal behoben werden kann.

Latenzzeit bei der Snapshot-Erstellung

Problem: Wenn Sie eine neue Funktionsversion veröffentlichen, bleibt die Funktion lange im `Pending` Status.

Häufige Ursache

Wenn Lambda einen Snapshot erstellt, kann Ihr Initialisierungscode bis zu 15 Minuten lang ausgeführt werden. Das Zeitlimit beträgt 130 Sekunden oder das konfigurierte Funktions-Timeout (maximal 900 Sekunden), je nachdem, welcher Wert höher ist.

Wenn Ihre Funktion mit einer VPC verbunden ist, muss Lambda möglicherweise auch Netzwerkschnittstellen erstellen, bevor die Funktion `Active` wird. Wenn Sie versuchen, die Funktionsversion aufzurufen, während die Funktion `Pending` ist, erhalten Sie möglicherweise eine 409 `ResourceConflictException`. Wenn die Funktion über einen Amazon-API-Gateway-Endpunkt aufgerufen wird, erhalten Sie möglicherweise einen 500-Fehler im API Gateway.

Auflösung

Warten Sie mindestens 15 Minuten, bis die Funktionsversion initialisiert ist, bevor Sie sie aufrufen.

Isolierung von Mietern

Verwenden Sie den Mandantenisolationsmodus, wenn Sie eine isolierte Anforderungsverarbeitung für einzelne Endbenutzer oder Mandanten benötigen, die eine Lambda-Funktion aufrufen. Diese Funktion vereinfacht die Erstellung von Mehrmandantenanwendungen, die mandantenspezifischen Code oder Daten verarbeiten, wie z. B. SaaS-Plattformen für die Workflow-Automatisierung oder Codeausführung, da die Verwaltung mandantenspezifischer Funktionsressourcen und die Anforderungsrouting-Logik entfällt.

Für Anwendungen mit mehreren Mandanten gelten strenge Isolationsanforderungen, wenn sie Code ausführen oder Daten für einzelne Mandanten oder Endbenutzer verarbeiten. Im Tenant-Isolationsmodus verwendet Lambda eine vom Kunden angegebene Mandanten-ID, um Anfragen an die zugrunde liegenden Funktionsausführungsumgebungen weiterzuleiten. Dadurch wird sichergestellt, dass die Ausführungsumgebungen einer Funktion nur für Aufrufe vom angegebenen Endbenutzer oder Mandanten verwendet werden. Die Funktionsausführungsumgebungen von Lambda nutzen die Firecracker-Virtualisierung, um Workloads zu isolieren.

Wenn eine Funktion, die den Tenant-Isolationsmodus verwendet, einen Aufruf mit einer Mandanten-ID empfängt, versucht Lambda zunächst, eine verfügbare Ausführungsumgebung zu finden, die dieser Mandanten-ID zugeordnet ist. Wenn keine Ausführungsumgebungen vorhanden sind, erstellt Lambda eine neue Ausführungsumgebung und weist diesem Mandanten eine neue Ausführungsumgebung zu. Wenn Funktionsaufrufe mit der angegebenen Mandanten-ID immer größer werden, sucht Lambda nach Bedarf nach neuen Ausführungsumgebungen oder erstellt sie.

Themen

- Wann sollte der Mandantenisolationsmodus verwendet werden
- Unterstützte Funktionen und Einschränkungen
- Unterstützt AWS-Regionen
- Überlegungen
- Preisgestaltung
- Isolationsmodus
- Aktivierung der Mandantenisolierung für Lambda-Funktionen
- Aufrufen von Lambda-Funktionen mit Mandantenisolierung
- Zugreifen auf die Mandanten-ID im Lambda-Funktionscode
- Überwachung von Lambda-Funktionen mit Mandantenisolierung

- Problembehandlung bei der Mandantenisolierung für Lambda-Funktionen

Wann sollte der Mandantenisolationsmodus verwendet werden

Verwenden Sie den Mandantenisolationsmodus, wenn Sie mehrere Endbenutzer oder Mandanten mit einer einzigen Lambda-Funktion bedienen müssen, und stellen Sie gleichzeitig sicher, dass die Ausführungsumgebungen, die zur Verarbeitung von Aufrufen für einzelne Mandanten verwendet werden, voneinander isoliert bleiben. Diese strikte Isolierung der Ausführungsumgebungen ist für Mehrmandantenanwendungen erforderlich, die:

- Vom Endbenutzer bereitgestellten Code ausführen: Durch die Beibehaltung separater Ausführungsumgebungen für einzelne Mandanten können die Auswirkungen der Ausführung von vom Benutzer bereitgestelltem Code, der falsch oder bösartig sein kann, begrenzt werden.
- Verarbeitung mandantenspezifischer Daten: Durch die Beibehaltung separater Ausführungsumgebungen für einzelne Mandanten kann verhindert werden, dass vertrauliche mandantenspezifische Daten an andere Mandanten weitergegeben werden.

Durch mehrere Aufrufanfragen desselben Mandanten kann dieselbe Funktionsausführungsumgebung wiederverwendet werden, wodurch Kaltstarts reduziert und die Reaktionszeiten für latenzempfindliche Anwendungen verbessert werden.

Unterstützte Funktionen und Einschränkungen

Der Isolationsmodus für Mandanten wird bei Funktionen, die Function, Provisioned Concurrency oder verwenden URLs, nicht unterstützt. SnapStart Sie können Anfragen an eine vom Mandanten isolierte Funktion senden, indem Sie synchrone Aufrufe, asynchrone Aufrufeoder Amazon API Gateway als Event-Trigger verwenden.

Unterstützt AWS-Regionen

Der Mandantenisolationsmodus wird in allen kommerziellen Regionen außer Asien-Pazifik (Neuseeland) unterstützt.

Überlegungen

Beachten Sie Folgendes, wenn Sie Tenant Isolation mit Ihren Lambda-Funktionen verwenden:

- Unveränderliche Konfiguration: Die Mandantenisolierung ist eine unveränderliche Funktionseigenschaft. Sie kann nur aktiviert werden, wenn eine Funktion erstellt wird.
- Erforderlicher Tenant-ID-Parameter: Funktionen, die den Tenant-Isolationsmodus verwenden, müssen mit einem Parameter aufgerufen werden. `tenant-id` Wenn Sie diesen Parameter weglassen, schlagen Funktionsaufrufe fehl.
- Die Ausführungsrolle gilt für alle Mandanten: Aufrufe von allen Mandanten verwenden die in der Ausführungsrolle Ihrer Lambda-Funktion definierten Berechtigungen.
- Parallelität: Es gibt keine Änderungen an der Parallelität oder dem Skalierungsverhalten Ihrer Funktion, wenn Sie die Mandantenisolierung verwenden. Lambda legt eine Obergrenze von 2.500 mandantenisolierten Ausführungsumgebungen (aktiv oder inaktiv) für jeweils 1.000 gleichzeitige Ausführungen fest, die für Ihre Lambda-Funktion konfiguriert sind.

Preisgestaltung

Ihnen wird eine Gebühr berechnet, wenn Lambda eine neue, vom Mandanten isolierte Ausführungsumgebung erstellt. Der Preis hängt von der Speichermenge ab, die Sie Ihrer Funktion zuweisen, und von der CPU-Architektur, die Sie verwenden. Weitere Informationen finden Sie unter AWS Lambda Preise.

Isolationsmodus

In der folgenden Tabelle werden die Unterschiede zwischen Lambda-Funktionen mit und ohne Mandantenisolierung beschrieben.

Feature	Mit Mandantenisolierung	Ohne Isolierung der Mieter
Art der Isolierung	Isolierung auf Mandantenebene	Isolierung auf Funktionsebene
Wiederverwendung der Umgebung	Ausführungsumgebungen werden niemals mandantenübergreifend wiederverwendet	Ausführungsumgebungen können bei Aufrufen derselben Funktion wiederverwendet werden
Datenisolierung	Auf Daten von anderen Mandanten kann nicht zugegriffen werden	Auf Daten aus früheren Aufrufen derselben Funktionsversion kann möglicherweise zugegriffen werden

Feature	Mit Mandantenisolierung	Ohne Isolierung der Mieter
Kalte Starts	Mehr Kaltstarts aufgrund von mieterspezifischen Umgebungen	Weniger Kaltstarts aufgrund der Wiederverwendung der Umgebung
Preisgestaltung	Zusätzliche Gebühr neben den Standard-Lambda-Preisen	Lambda-Standardpreise

Aktivierung der Mandantenisolierung für Lambda-Funktionen

Um den Tenant-Isolationsmodus zu aktivieren, erstellen Sie eine neue Lambda-Funktion. Sie können die Mandantenisolierung für bestehende Funktionen nicht aktivieren.

Themen

- Aktivierung der Mandantenisolierung (Konsole)
- Aktivierung der Mandantenisolierung (AWS CLI)
- Aktivierung der Mandantenisolierung (API)
- Aktivierung der Mandantenisolierung (CloudFormation)

Aktivierung der Mandantenisolierung (Konsole)

Um eine Lambda-Funktion mit der Konsole zu erstellen

1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.
2. Wählen Sie Funktion erstellen.
3. Wählen Sie Verfassen von Grund auf aus.
4. Geben Sie im Bereich Grundlegende Informationen als Funktionsname **`image-analysis`** ein.
5. Wählen Sie für Runtime eine der unterstützten Lambda-Laufzeiten aus.
6. Wählen Sie unter Zusätzliche Konfigurationen, Tenant-Isolationsmodus, die Option Aktivieren aus.
7. Überprüfen Sie Ihre Einstellungen und wählen Sie Funktion erstellen.

Aktivierung der Mandantenisolierung (AWS CLI)

Funktion mit Mandantenisolierung erstellen

Wenn Sie eine neue Funktion mit der CLI erstellen, fügen Sie die `--tenancy-config '{"TenantIsolationMode": "PER_TENANT"}'` Option zu Ihrer Create-Function-Anfrage hinzu. Beispiel:

```
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"}'
```

Aktivierung der Mandantenisolierung (API)

So aktivieren Sie die Mandantenisolierung mithilfe der Lambda-API

1. Erstellen Sie eine neue Funktion mit aktivierter Mandantenisolierung, indem Sie die CreateFunctionAPI-Aktion mit dem `TenancyConfig` Parameter verwenden.
2. Vergewissern Sie sich mithilfe der GetFunctionConfigurationAktion, dass die Mandantenisolierung für die Funktion aktiviert ist. Wenn die Antwort anzeigt, dass dies der `PER_TENANT` Fall `TenantIsolationMode` ist, ist die Mandantenisolierung für die Funktion aktiviert:

   ```
   "TenancyConfig": {
           "TenantIsolationMode": "PER_TENANT"
       }
   ```

Rufen Sie die Funktionsversion mit der Aufruf-Aktion auf. Weitere Informationen finden Sie unter Aufrufen von Lambda-Funktionen mit Mandantenisolierung.

Aktivierung der Mandantenisolierung (CloudFormation)

Die folgende CloudFormation Vorlage erstellt eine neue Lambda-Funktion mit aktivierter Mandantenisolierung:

```
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
```

Aufrufen von Lambda-Funktionen mit Mandantenisolierung

Wenn Sie eine Funktion aufrufen, für die die Mandantenisolierung aktiviert ist, müssen Sie einen Parameter angeben. `tenant-id` Dieser Parameter stellt sicher, dass Ihr Funktionsaufruf in einer Ausführungsumgebung verarbeitet wird, die diesem bestimmten Mandanten zugewiesen ist.

Aufrufen von Funktionen mit Tenant Isolation ()AWS CLI

Synchroner Aufruf

Fügen Sie für synchrone Aufrufe den `--tenant-id` Parameter zu Ihrem Invoke-Befehl hinzu:

```
aws lambda invoke \
    --function-name image-analysis \
    --tenant-id blue \
    response.json
```

Asynchroner Aufruf

Geben Sie bei asynchronen Aufrufen sowohl die als auch die Parameter an: `--tenant-id --invocation-type Event`

```
aws lambda invoke \
    --function-name image-analysis \
    --tenant-id blue \
    --invocation-type Event \
    response.json
```

Funktionen mit Mandantenisolierung (API) aufrufen

Wenn Sie die Aktion „API aufrufen“ direkt verwenden, geben Sie die Mandanten-ID mithilfe des `X-Amzn-Tenant-Id` Parameters in Ihre Anfrage ein.

Beispiel für eine API-Anfrage

```
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"
}
```

Aufrufen von Funktionen mit Mandantenisolierung (API Gateway)

Wenn Sie API Gateway REST verwenden APIs , um mandantenisolierte Lambda-Funktionen auszulösen, müssen Sie API Gateway so konfigurieren, dass die Eigenschaften der Client-Anfrage dem von Lambda erwarteten `X-Amz-Tenant-Id` Header zugeordnet werden. API Gateway verwendet die Aktion „API aufrufen“ von Lambda, bei der die Mandanten-ID über den `X-Amz-Tenant-Id` HTTP-Header übergeben werden muss. Sie können API Gateway so konfigurieren, dass dieser HTTP-Header mit einem Wert, der aus Eigenschaften der Client-Anfrage wie HTTP-Headern, Abfrageparametern oder Pfadparametern abgerufen wird, in die Lambda-Aufrufanforderung eingefügt wird. Sie müssen zuerst die Eigenschaft der Client-Anfrage zuordnen, bevor Sie den Header überschreiben können. `X-Amz-Tenant-Id`

> **Note**
>
> Sie können HTTP nicht verwenden APIs , um mandantenisolierte Lambda-Funktionen aufzurufen, da es nicht möglich ist, den Header zu überschreiben. `X-Amz-Tenant-Id`

Verwenden von Anforderungsheadern

Konfigurieren Sie Ihre API-Gateway-Integration so, dass ein benutzerdefinierter Header von der Client-Anfrage dem `X-Amz-Tenant-Id` Header zugeordnet wird. Das folgende Beispiel zeigt eine Client-Anfrage mit einem `x-tenant-id` Header:

```
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"
}
```

In Ihrer API Gateway Gateway-Methodenkonfiguration müssen Sie:

1. Aktivieren Sie den Header-Parameter für die Client-Anfrage (zum Beispiel`method.request.header.x-tenant-id`)
2. Ordnen Sie den Client-Header dem Lambda-Integrationsheader zu, indem Sie `integration.request.header.X-Amz-Tenant-Id`

Verwenden von Abfrageparametern

Auf ähnliche Weise können Sie dem `X-Amz-Tenant-Id` Header Abfrageparameter zuordnen:

```
GET /api/process?tenant-id=blue&data=sample HTTP/1.1
Host: your-api-id.execute-api.us-east-1.amazonaws.com
```

Konfigurieren Sie die Methode so, dass der Abfrageparameter aktiviert wird, und ordnen Sie ihn dem Integrationsheader zu.

Funktionen mit Tenant Isolation (SDK) aufrufen

Wenn Sie Funktionen verwenden AWS SDKs , die vom Mandanten isoliert sind, geben Sie die Mandanten-ID in Ihre Aufrufanforderung ein.

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);
    }
}
```

Zugreifen auf die Mandanten-ID im Lambda-Funktionscode

Wenn für Ihre Lambda-Funktion die Mandantenisolierung aktiviert ist, wird die Mandanten-ID, die zum Aufrufen Ihrer Funktion verwendet wurde, in dem Kontextobjekt verfügbar gemacht, das an Ihren Funktionshandler übergeben wird. Sie können diesen Bezeichner verwenden, um mandantenspezifische Logik-, Überwachungs- und Debugging-Funktionen zu implementieren.

Themen

- Zugreifen auf die Mandanten-ID
- Allgemeine Nutzungsmuster
- Überwachung und Debugging

Zugreifen auf die Mandanten-ID

Die Mandanten-ID ist über die `tenantId` Eigenschaft des Kontextobjekts verfügbar. Beachten Sie, dass diese Eigenschaft während der Aufrufphase und nicht während der Initialisierungsphase verfügbar ist.

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);
        }
    }
}
```

Allgemeine Nutzungsmuster

Im Folgenden finden Sie gängige Methoden zur Verwendung der Mandanten-ID in Ihrem Funktionscode:

Mandantenspezifische Konfiguration

Verwenden Sie die Mandanten-ID, um die mandantenspezifische Konfiguration oder Einstellungen zu laden:

```
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)
```

Mandantenspezifischer Datenzugriff

Verwenden Sie die Mandanten-ID, um die Datenisolierung und Zugriffskontrolle sicherzustellen:

```
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)
```

Überwachung und Debugging

Die Mandanten-ID wird automatisch in Lambda-Logs aufgenommen, wenn Sie die JSON-Protokollierung aktiviert haben, sodass mandantenspezifische Probleme leichter überwacht und debuggt werden können. Sie können die Mandanten-ID auch für benutzerdefinierte Metriken und die Ablaufverfolgung verwenden.

Example Benutzerdefinierte Metriken mit Mandanten-ID

Das folgende Beispiel zeigt, wie Sie die Mandanten-ID verwenden, um mandantenspezifische CloudWatch Metriken zur Überwachung von Nutzungsmustern und Leistung pro Mandant zu erstellen:

```
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)
```

Überwachung von Lambda-Funktionen mit Mandantenisolierung

Sie können Ihre mandantenisolierten Lambda-Funktionen mithilfe von Amazon CloudWatch überwachen und mithilfe der Telemetrie-API auf Echtzeit-Telemetriedaten für Erweiterungen zugreifen. AWS X-Ray

Grundlegendes zur Protokollierung für den isolierten Mandantenmodus

Bei Funktionen, die Mandantenisolierung verwenden, nimmt Lambda die Mandanten-ID automatisch in Funktionsprotokolle auf, wenn Sie die JSON-Protokollierung aktiviert haben, was die Überwachung und das Debuggen mandantenspezifischer Probleme erleichtert. Lambda erstellt für jede Ausführungsumgebung einen separaten CloudWatch Protokollstream. Sie können CloudWatch Logs Insights verwenden, um Log-Streams zu finden, die zu einem bestimmten Mandanten gehören, indem Sie nach der Mandanten-ID filtern:

```
fields @logStream, @message
| filter tenantId=='BlueTenant' or record.tenantId=='BlueTenant'
| stats count() as logCount by @logStream
| sort @timestamp desc
```

Sie können diesen Parameter auch verwenden, um mandantenspezifische Protokolle für alle Log-Streams abzurufen:

```
fields @message
| filter tenantId=='BlueTenant' or record.tenantId=='BlueTenant'
| limit 1000
```

Die `tenantId` Eigenschaft ist für Plattformereignisse (wie `platform.start` und`platform.report`) und benutzerdefinierte Protokolle, die Sie in Ihrem Funktionscode drucken, enthalten, wie unten dargestellt:

```
{
    "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"
}
```

Problembehandlung bei der Mandantenisolierung für Lambda-Funktionen

Auf dieser Seite werden häufig auftretende Probleme behandelt, die bei der Verwendung von Tenant Isolation für auftreten AWS Lambda.

InvalidParameterValueException

Fehler: Die Mandanten-ID-Konfiguration wurde nicht angegeben oder an eine Funktion übergeben, wenn die Mandantenisolierung nicht aktiviert ist.

Häufige Ursachen

Dieser Fehler tritt auf, wenn eine vom Mandanten isolierte Funktion ohne Mandanten-ID oder eine non-tenant-isolated Funktion mit einer Mandanten-ID aufgerufen wird.

Auflösung

Fügen Sie eine Mandanten-ID hinzu, wenn für die Funktion die Mandantenisolierung aktiviert ist, oder entfernen Sie die Mandanten-ID, wenn für die Funktion die Mandantenisolierung nicht aktiviert ist.

TooManyRequestsException

Fehler: Rate überschritten

Häufige Ursachen

Zusätzlich zur Ratenbegrenzung, die auf der maximalen Anzahl gleichzeitiger Ausführungen und der Funktionsskalierungsrate basiert, begrenzt Lambda die maximale Anzahl von mandantenfähigen Ausführungsumgebungen (aktiv oder inaktiv), die gleichzeitig existieren können, auf 2.500 pro 1.000 gleichzeitige Ausführungen Ihrer Funktion.

Auflösung

Um dieses Problem zu beheben, können Sie entweder die Rate verringern, mit der Aufrufanforderungen mit eindeutigen Mandantenkennungen gestellt werden, Wiederholungsversuche mit Backoff und Jitter implementieren oder eine Erhöhung des Parallelitätslimits für Funktionen beantragen.

Lambda mit Ereignissen aus anderen Diensten aufrufen AWS

Einige AWS-Services können Lambda-Funktionen mithilfe von Triggern direkt aufrufen. Diese Dienste leiten Ereignisse an Lambda weiter und die Funktion wird sofort aufgerufen, wenn das angegebene Ereignis eintritt. Trigger eignen sich für diskrete Ereignisse und die Verarbeitung in Echtzeit. Wenn Sie mit der Lambda-Konsole einen Trigger erstellen, interagiert die Konsole mit dem entsprechenden AWS Dienst, um die Ereignisbenachrichtigung für diesen Dienst zu konfigurieren. Der Auslöser wird tatsächlich von dem Dienst gespeichert und verwaltet, der die Ereignisse generiert, nicht von Lambda.

Die Ereignisse sind im JSON-Format gegliedert. Die JSON-Struktur variiert je nach Service, der sie generiert und dem Ereignistyp, aber sie alle enthalten die Daten, die die Funktion benötigt, um das Ereignis zu verarbeiten.

Eine Funktion kann mehrere Auslöser haben. Jeder Auslöser fungiert als Client, der Ihre Funktion unabhängig aufruft, und jedes Ereignis, das Lambda an Ihre Funktion weitergibt, enthält Daten von nur einem Auslöser. Lambda wandelt das Ereignisdokument in ein Objekt um und leitet es an Ihren Funktions-Handler weiter.

Je nach Service kann der ereignisgesteuerte Aufruf synchron oder asynchron erfolgen.

- Beim synchronen Aufruf wartet der Service, der das Ereignis generiert, auf die Antwort Ihrer Funktion. Dieser Service definiert die Daten, die die Funktion in der Antwort zurückgeben muss. Der Service steuert die Fehlerstrategie, z. B. ob bei Fehlern ein erneuter Versuch unternommen werden soll.
- Bei asynchronen Aufrufen verschiebt Lambda das Ereignis in die Warteschlange, bevor sie es an Ihre Funktion übergibt. Wenn Lambda das Ereignis in die Warteschlange stellt, sendet es sofort eine Erfolgsantwort an den Service der das Ereignis generiert hat. Nachdem die Funktion das Ereignis verarbeitet hat, gibt Lambda keine Antwort auf den ereignisgenerierenden Service zurück.

Erstellen eines Auslösers

Der einfachste Weg, einen Auslöser zu erstellen, ist die Verwendung der Lambda-Konsole. Wenn Sie einen Auslöser über die Konsole erstellen, fügt Lambda automatisch die erforderlichen Berechtigungen zur ressourcenbasierten Richtlinie der Funktion hinzu.

So erstellen Sie einen Auslöser mit der Lambda-Konsole

1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.
2. Wählen Sie die Funktion aus, für die Sie einen Auslöser erstellen möchten.
3. Wählen Sie im Bereich Function overview (Funktionsübersicht) die Option Add trigger (Auslöser hinzufügen).
4. Wählen Sie den AWS Dienst aus, für den Sie Ihre Funktion aufrufen möchten.
5. Füllen Sie die Optionen im Bereich Trigger-Konfiguration aus und wählen Sie Hinzufügen. Je nachdem, wie AWS-Service Sie Ihre Funktion aufrufen möchten, unterscheiden sich die Konfigurationsoptionen für den Trigger.

Services, die Lambda-Funktionen aufrufen können

In der folgenden Tabelle sind Dienste aufgeführt, die Lambda-Funktionen aufrufen können.

Service	Methode des Aufrufs
Amazon Managed Streaming für Apache Kafka	Zuordnung von Ereignisquellen
Selbstverwaltetes Apache Kafka	Zuordnung von Ereignisquellen
Amazon API Gateway	Ereignisgesteuert; synchroner Aufruf
AWS CloudFormation	Ereignisgesteuert; asynchroner Aufruf
CloudWatch Amazon-Protokolle	Ereignisgesteuert; asynchroner Aufruf
AWS CodeCommit	Ereignisgesteuert; asynchroner Aufruf
AWS CodePipeline	Ereignisgesteuert; asynchroner Aufruf
Amazon Cognito	Ereignisgesteuert; synchroner Aufruf
AWS Config	Ereignisgesteuert; asynchroner Aufruf

Service	Methode des Aufrufs
Amazon Connect	Ereignisgesteuert; synchroner Aufruf
Amazon DocumentDB	Zuordnung von Ereignisquellen
Amazon-DynamoDB	Zuordnung von Ereignisquellen
Elastic Load Balancing (Application Load Balancer)	Ereignisgesteuert; synchroner Aufruf
Amazon EventBridge (CloudWatch Veranstaltungen)	Ereignisgesteuert; asynchroner Aufruf (Ereignisbusse), synchroner oder asynchroner Aufruf (Pipes und Zeitpläne)
AWS IoT	Ereignisgesteuert; asynchroner Aufruf
Amazon Kinesis	Zuordnung von Ereignisquellen
Amazon Data Firehose	Ereignisgesteuert; synchroner Aufruf
Amazon Lex	Ereignisgesteuert; synchroner Aufruf
Amazon MQ	Zuordnung von Ereignisquellen
Amazon Simple Email Service	Ereignisgesteuert; asynchroner Aufruf
Amazon Simple Notification Service	Ereignisgesteuert; asynchroner Aufruf
Amazon Simple Queue Service	Zuordnung von Ereignisquellen
Amazon Simple Storage Service (Amazon-S3)	Ereignisgesteuert; asynchroner Aufruf
Amazon Simple Storage Service Batch	Ereignisgesteuert; synchroner Aufruf
Secrets Manager	Secret-Rotation
AWS Step Functions	Ereignisgesteuert; synchroner oder asynchroner Aufruf

Service	Methode des Aufrufs
Amazon VPC Lattice	Ereignisgesteuert; synchroner Aufruf

Verwendung von Lambda mit Apache Kafka

Lambda unterstützt Apache Kafka als Ereignisquelle. Apache Kafka ist eine Open-Source-Plattform für Ereignisstreaming, die für die Verarbeitung von Datenpipelines und Streaminganwendungen mit hohem Durchsatz in Echtzeit entwickelt wurde. Es gibt zwei Hauptmethoden für die Verwendung von Lambda mit Apache Kafka:

- the section called “MSK”— Amazon Managed Streaming for Apache Kafka (Amazon MSK) ist ein vollständig verwalteter Service von. AWS Amazon MSK unterstützt die Automatisierung der Verwaltung Ihrer Kafka-Infrastruktur, einschließlich Bereitstellung, Patching und Skalierung.
- the section called “Selbstverwaltetes Apache Kafka”— In der AWS Terminologie umfasst ein selbstverwalteter Cluster auch nicht gehostete Kafka-Cluster.AWS Sie können Lambda beispielsweise weiterhin mit einem Kafka-Cluster verwenden, der bei einem AWS Nicht-Cloud-Anbieter wie Confluent Cloud oder Redpanda gehostet wird.

Bei der Entscheidung zwischen Amazon MSK und selbstverwaltetem Apache Kafka sollten Sie Ihre betrieblichen Anforderungen und Kontrollanforderungen berücksichtigen. Amazon MSK ist die bessere Wahl, wenn Sie Sie schnell AWS bei der Verwaltung eines skalierbaren, produktionsbereiten Kafka-Setups mit minimalem Betriebsaufwand unterstützen möchten. Es vereinfacht die Sicherheit, Überwachung und Hochverfügbarkeit und hilft Ihnen, sich auf die Anwendungsentwicklung statt auf die Infrastrukturverwaltung zu konzentrieren. Auf der anderen Seite eignet sich der selbstverwaltete Apache Kafka besser für Anwendungsfälle, die in nicht AWS gehosteten Umgebungen ausgeführt werden, einschließlich lokaler Cluster.

Themen

- Verwenden von Lambda mit Amazon MSK
- Verwendung von Lambda mit selbstverwaltetem Apache Kafka
- Skalierungsmodi des Ereignis-Pollers von Apache Kafka in Lambda
- Apache-Kafka-Abfrage und Stream-Startpositionen in Lambda
- Anpassbare Verbrauchergruppen-ID in Lambda
- Filtern von Ereignissen aus Amazon MSK und selbstverwalteten Apache-Kafka-Ereignisquellen
- Verwenden von Schemaregistern mit Kafka-Ereignisquellen in Lambda
- Verarbeitung mit niedriger Latenz für Kafka-Ereignisquellen
- Konfiguration der Fehlerbehandlungssteuerungen für Kafka-Ereignisquellen

- Erfassen verworfener Batches für Amazon MSK und selbstverwaltete Apache-Kafka-Ereignisquellen
- Ein Kafka-Thema als Ziel für den Fall eines Fehlers verwenden
- Fehlerbehebung bei der Kafka-Zuordnung von Ereignisquellen

Verwenden von Lambda mit Amazon MSK

Amazon Managed Streaming for Apache Kafka (Amazon MSK) ist ein vollständig verwalteter Service, mit dem Sie Anwendungen erstellen und ausführen können, die Apache Kafka zum Verarbeiten von Streamingdaten verwenden. Amazon MSK vereinfacht die Einrichtung, Skalierung und Verwaltung von Kafka-Clustern. Amazon MSK erleichtert auch die Konfiguration Ihrer Anwendung für mehrere Availability Zones und aus Sicherheitsgründen mit AWS Identity and Access Management (IAM).

In diesem Kapitel wird erklärt, wie Sie einen Amazon-MSK-Cluster als Ereignisquelle für Ihre Lambda-Funktion verwenden. Der allgemeine Prozess zur Integration von Amazon MSK mit Lambda umfasst die folgenden Schritte:

1. Cluster- und Netzwerk-Setup: Richten Sie zunächst Ihren Amazon-MSK-Cluster ein. Dazu gehört die richtige Netzwerkkonfiguration, damit Lambda auf Ihren Cluster zugreifen kann.
2. Einrichtung der Zuordnung von Ereignisquellen: Erstellen Sie anschließend die Ressource für die Zuordnung von Ereignisquellen, die Lambda benötigt, um Ihren Amazon-MSK-Cluster sicher mit Ihrer Funktion zu verbinden.
3. Einrichtung von Funktionen und Berechtigungen: Stellen Sie abschließend sicher, dass Ihre Funktion korrekt eingerichtet ist und über die erforderlichen Berechtigungen in ihrer Ausführungsrolle verfügt.

> **Note**
>
> Sie können jetzt Ihre Amazon MSK-Ereignisquellenzuordnungen direkt von der Lambda- oder der Amazon MSK-Konsole aus erstellen und verwalten. Beide Konsolen bieten die Option, die Einrichtung der erforderlichen Berechtigungen für die Lambda-Ausführungsrolle automatisch vorzunehmen, um den Konfigurationsprozess zu optimieren.

Beispiele für die Einrichtung einer Lambda-Integration mit einem Amazon MSK-Cluster finden Sie unter the section called “Tutorial” Verwenden von Amazon MSK als Ereignisquelle für AWS Lambda im AWS Compute-Blog und Amazon MSK Lambda-Integration in den Amazon MSK Labs.

Themen

- Beispielereignis
- Konfiguration Ihres Amazon-MSK-Clusters und Ihres Amazon-VPC-Netzwerks für Lambda
- Konfiguration von Lambda-Berechtigungen für Amazon MSK-Ereignisquellenzuordnungen

- Konfigurieren von Amazon MSK-Ereignisquellen für Lambda
- Tutorial: Verwenden einer Amazon-MSK-Zuordnung von Ereignisquellen zum Aufrufen einer Lambda-Funktion

Beispielereignis

Lambda sendet den Batch von Nachrichten im Ereignisparameter, wenn es Ihre Funktion aufruft. Die Ereignisnutzlast enthält ein Array von Meldungen. Jedes Array-Element enthält Details zum Amazon-MSK-Thema und zur Partitions-ID sowie einen Zeitstempel und eine base64-codierte Nachricht.


```
{
   "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
                  ]
```

```
                }
            ]
        }
    ]
  }
}
```

Konfiguration Ihres Amazon-MSK-Clusters und Ihres Amazon-VPC-Netzwerks für Lambda

Um Ihre AWS Lambda-Funktion mit Ihrem Amazon-MSK-Cluster zu verbinden, müssen Sie Ihren Cluster und die Amazon Virtual Private Cloud (VPC), in der er sich befindet, korrekt konfigurieren. Auf dieser Seite wird beschrieben, wie Sie Ihren Cluster und Ihre VPC konfigurieren. Wenn Ihr Cluster und Ihre VPC bereits ordnungsgemäß konfiguriert sind, lesen Sie unter the section called "Konfigurieren der Ereignisquelle" nach, wie Sie die Zuordnung von Ereignisquellen konfigurieren.

Themen

- Überblick über die Anforderungen an die Netzwerkkonfiguration für Lambda- und MSK-Integrationen
- Konfiguration eines NAT-Gateways für eine MSK-Ereignisquelle
- Konfiguration von AWS PrivateLink-Endpunkten für eine MSK-Ereignisquelle

Überblick über die Anforderungen an die Netzwerkkonfiguration für Lambda- und MSK-Integrationen

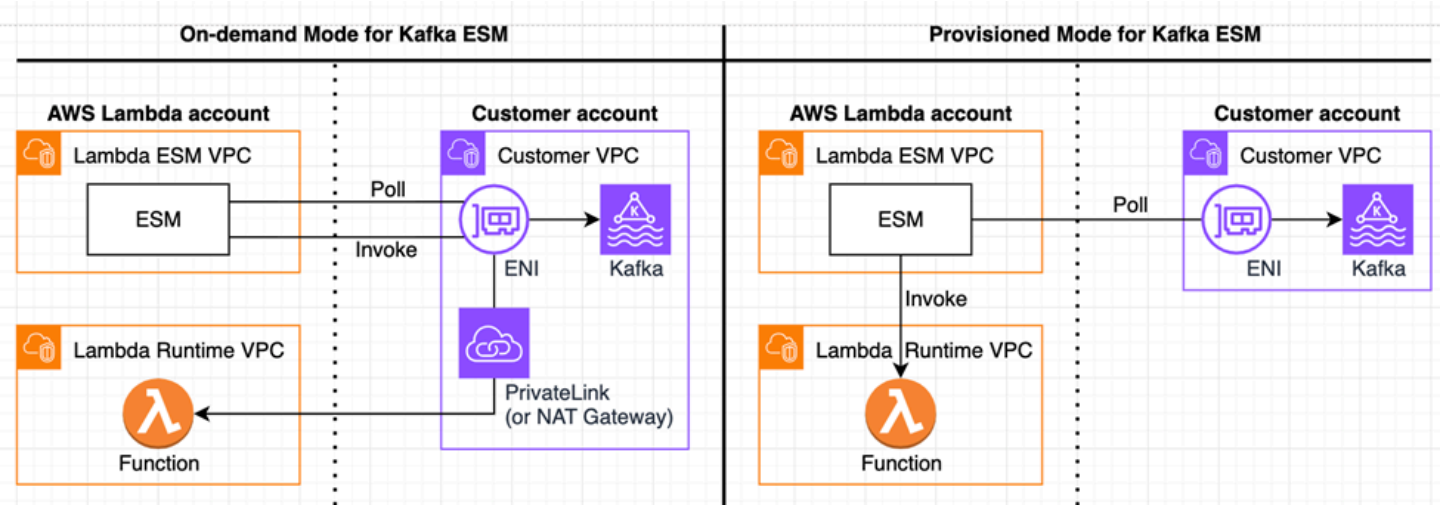
Die für eine Lambda- und MSK-Integration erforderliche Netzwerkkonfiguration hängt von der Netzwerkarchitektur Ihrer Anwendung ab. An dieser Integration sind drei Hauptressourcen beteiligt: der Amazon-MSK-Cluster, die Lambda-Funktion und die Lambda-Zuordnung von Ereignisquellen. Jede dieser Ressourcen befindet sich in einer anderen VPC:

- Ihr Amazon-MSK-Cluster befindet sich normalerweise in einem privaten Subnetz einer VPC, die Sie verwalten.
- Ihre Lambda-Funktion befindet sich in einer von AWS verwalteten VPC, die Lambda gehört.
- Ihre Lambda-Zuordnung von Ereignisquellen befindet sich in einer anderen von AWS verwalteten VPC, die Lambda gehört, getrennt von der VPC, die Ihre Funktion enthält.

Die Zuordnung von Ereignisquellen ist die Zwischenressource zwischen dem MSK-Cluster und der Lambda-Funktion. Die Zuordnung von Ereignisquellen hat zwei Hauptaufgaben. Zunächst fragt

sie Ihren MSK-Cluster nach neuen Nachrichten ab. Anschließend wird Ihre Lambda-Funktion mit diesen Nachrichten aufgerufen. Da sich diese drei Ressourcen in unterschiedlichen VPCs befinden, erfordern sowohl der Abruf- als auch der Aufrufvorgang VPC-übergreifende Netzwerkaufrufe.

Die Anforderungen an die Netzwerkkonfiguration für Ihre Zuordnung von Ereignisquellen hängen davon ab, ob der Bereitstellungsmodus oder der On-Demand-Modus verwendet wird, wie im folgenden Diagramm dargestellt:



Die Art und Weise, wie die Lambda-Zuordnung von Ereignisquellen Ihren MSK-Cluster nach neuen Nachrichten abfragt, ist in beiden Modi dieselbe. Um eine Verbindung zwischen Ihrer Zuordnung von Ereignisquellen und Ihrem MSK-Cluster herzustellen, erstellt Lambda in Ihrem privaten Subnetz eine Hyperplane-ENI (oder verwendet eine vorhandene, falls verfügbar), um eine sichere Verbindung herzustellen. Wie im Diagramm dargestellt, verwendet diese Hyperplane-ENI die Subnetz- und Sicherheitsgruppenkonfiguration Ihres MSK-Clusters, nicht Ihre Lambda-Funktion.

Nachdem Sie die Nachricht vom Cluster abgerufen haben, ist die Art und Weise, wie Lambda Ihre Funktion aufruft, in jedem Modus unterschiedlich:

- Im Bereitstellungsmodus verarbeitet Lambda automatisch die Verbindung zwischen der VPC für die Zuordnung von Ereignisquellen und der Funktions-VPC. Sie benötigen also keine zusätzlichen Netzwerkkomponenten, um Ihre Funktion erfolgreich aufzurufen.
- Im On-Demand-Modus ruft Ihre Lambda-Zuordnung von Ereignisquellen Ihre Funktion über einen Pfad durch Ihre vom Kunden verwaltete VPC auf. Aus diesem Grund müssen Sie entweder ein NAT-Gateway im öffentlichen Subnetz Ihrer VPC oder AWS PrivateLink-Endpunkte im privaten Subnetz der VPC konfigurieren, die Zugriff auf Lambda, AWS -Security-Token-Service (STS) und optional AWS Secrets Manager bieten. Die korrekte Konfiguration einer dieser Optionen ermöglicht

eine Verbindung zwischen Ihrer VPC und der von Lambda verwalteten Laufzeit-VPC, die zum Aufrufen Ihrer Funktion erforderlich ist.

Ein NAT-Gateway ermöglicht Ressourcen in Ihrem privaten Subnetz den Zugriff auf das öffentliche Internet. Bei dieser Konfiguration wird Ihr Datenverkehr über das Internet geleitet, bevor die Lambda-Funktion aufgerufen wird. AWS PrivateLink-Endpunkte ermöglichen es privaten Subnetzen, sich sicher mit AWS-Services oder anderen privaten VPC-Ressourcen zu verbinden, ohne das öffentliche Internet zu nutzen. Weitere Informationen zur Konfiguration dieser Ressourcen finden Sie unter the section called “Konfiguration eines NAT-Gateways für eine MSK-Ereignisquelle” oder the section called “Konfiguration von AWS PrivateLink-Endpunkten für eine MSK-Ereignisquelle”.

Bisher gingen wir davon aus, dass sich Ihr MSK-Cluster in einem privaten Subnetz innerhalb Ihrer VPC befindet, was der häufigste Fall ist. Selbst wenn sich Ihr MSK-Cluster in einem öffentlichen Subnetz innerhalb Ihrer VPC befindet, müssen Sie AWS PrivateLink-Endpunkte konfigurieren, um eine sichere Verbindung zu gewährleisten. In der folgenden Tabelle sind die Anforderungen an die Netzwerkkonfiguration zusammengefasst, je nachdem, wie Sie Ihren MSK-Cluster und die Lambda-Zuordnung von Ereignisquellen konfigurieren:

Standort des MSK-Clusters (in der vom Kunden verwalteten VPC)	Skalierungsmodus der Lambda-Zuordnung von Ereignisquellen	Erforderliche Netzwerkkonfiguration
Privates Subnetz	On-Demand-Modus	NAT-Gateway (im öffentlichen Subnetz Ihrer VPC) oder AWS PrivateLink-Endpunkte (im privaten Subnetz Ihrer VPC), um Zugriff auf Lambda, AWS STS und optional Secrets Manager zu ermöglichen.
Öffentliches Subnetz	On-Demand-Modus	AWS PrivateLink-Endpunkte (im öffentlichen Subnetz Ihrer VPC), um Zugriff auf Lambda, AWS STS und optional Secrets Manager zu ermöglichen.

Standort des MSK-Clusters (in der vom Kunden verwalteten VPC)	Skalierungsmodus der Lambda-Zuordnung von Ereignisquellen	Erforderliche Netzwerkkonfiguration
Privates Subnetz	Modus bereitgestellter Kapazität	Keine
Öffentliches Subnetz	Modus bereitgestellter Kapazität	Keine

Darüber hinaus müssen die mit Ihrem MSK-Cluster verknüpften Sicherheitsgruppen den Datenverkehr über die richtigen Ports zulassen. Stellen Sie sicher, dass Sie die folgenden Regeln für Sicherheitsgruppen konfiguriert haben:

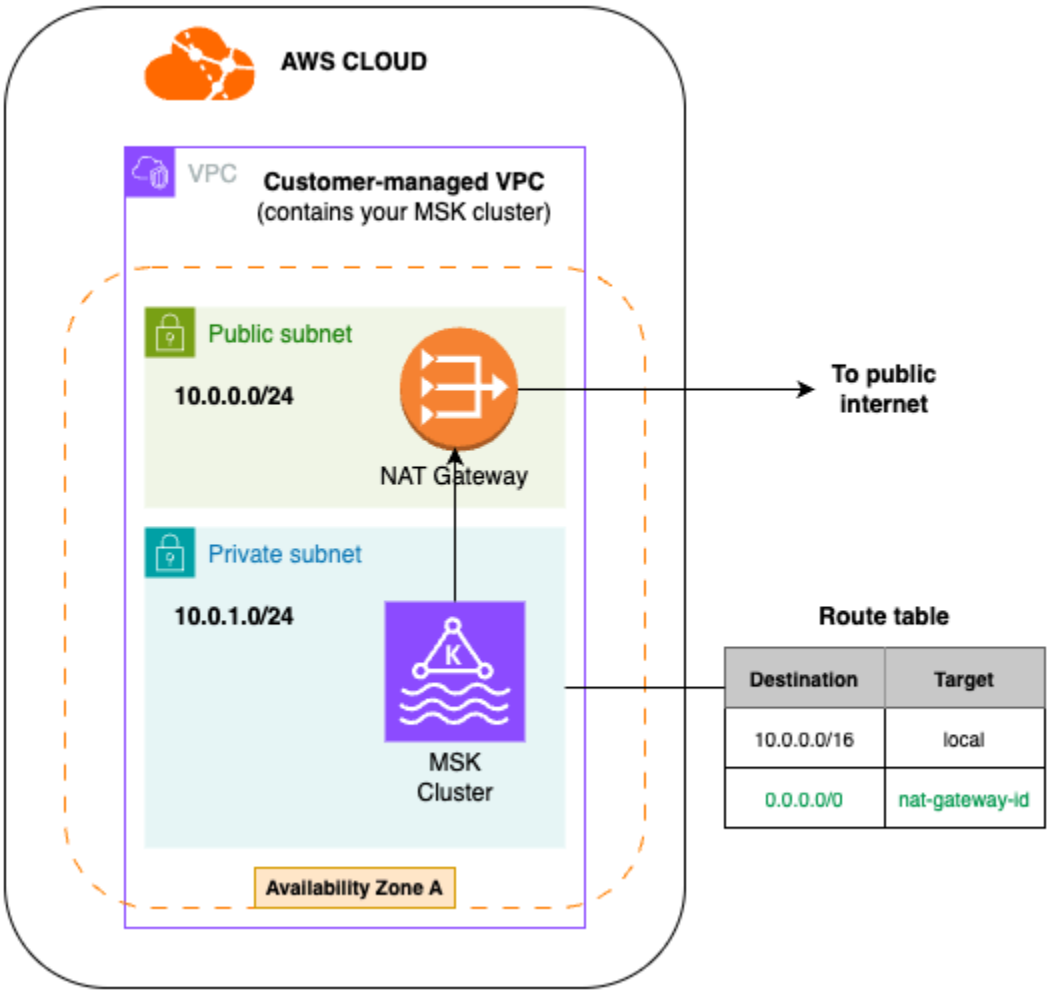
- Regeln für eingehenden Datenverkehr – Erlauben Sie den gesamten Datenverkehr auf dem standardmäßigen Broker-Port. Der von MSK verwendete Port hängt vom Authentifizierungstyp des Clusters ab: `9098` für IAM-Authentifizierung, `9096` für SASL/SCRAM und `9094` für TLS. Alternativ können Sie eine selbstreferenzierende Regel für Sicherheitsgruppen verwenden, um den Zugriff von Instances innerhalb derselben Sicherheitsgruppe zuzulassen.
- Regeln für ausgehenden Datenverkehr – Erlauben Sie den gesamten Datenverkehr an Port `443` für externe Ziele, wenn Ihre Funktion mit anderen AWS-Services kommunizieren muss. Alternativ können Sie eine selbstreferenzierende Regel für Sicherheitsgruppen verwenden, um den Zugriff auf den Broker einzuschränken, wenn Sie nicht mit anderen AWS-Services kommunizieren müssen.
- Regeln für eingehenden Datenverkehr auf Amazon-VPC-Endpunkten – Wenn Sie einen Amazon-VPC-Endpunkt verwenden, muss die mit dem Endpunkt verbundene Sicherheitsgruppe eingehenden Datenverkehr auf Port `443` von der Sicherheitsgruppe des Clusters zulassen.

Konfiguration eines NAT-Gateways für eine MSK-Ereignisquelle

Sie können ein NAT-Gateway so konfigurieren, dass Ihre Zuordnung von Ereignisquellen Nachrichten von Ihrem Cluster abfragen und die Funktion über einen Pfad durch Ihre VPC aufrufen kann. Dies ist nur erforderlich, wenn Ihre Zuordnung von Ereignisquellen den On-Demand-Modus verwendet und sich Ihr Cluster in einem privaten Subnetz Ihrer VPC befindet. Wenn sich Ihr Cluster in einem öffentlichen Subnetz Ihrer VPC befindet oder Ihre Zuordnung von Ereignisquellen den Bereitstellungsmodus verwendet, müssen Sie kein NAT-Gateway konfigurieren.

Ein NAT-Gateway ermöglicht Ressourcen in einem privaten Subnetz den Zugriff auf das öffentliche Internet. Wenn Sie eine private Verbindung zu Lambda benötigen, finden Sie stattdessen weitere Informationen unter the section called “Konfiguration von AWS PrivateLink-Endpunkten für eine MSK-Ereignisquelle”.

Nachdem Sie Ihr NAT-Gateway konfiguriert haben, müssen Sie die entsprechenden Routing-Tabellen konfigurieren. Dadurch kann der Datenverkehr aus Ihrem privaten Subnetz über das NAT-Gateway in das öffentliche Internet geleitet werden.



Die folgenden Schritte führen Sie durch die Konfiguration eines NAT-Gateways über die Konsole. Wiederholen Sie diese Schritte bei Bedarf für jede Availability Zone (AZ).

So konfigurieren Sie ein NAT-Gateway und das richtige Routing (Konsole)

1. Folgen Sie den Schritten unter Erstellen eines NAT-Gateways und beachten Sie dabei Folgendes:

- NAT-Gateways sollten sich immer in einem öffentlichen Subnetz befinden. Erstellen Sie NAT-Gateways mit öffentlicher Konnektivität.
- Wenn Ihr MSK-Cluster über mehrere AZs repliziert wird, erstellen Sie ein NAT-Gateway pro AZ. In jeder AZ sollte Ihre VPC beispielsweise ein privates Subnetz mit Ihrem Cluster und ein öffentliches Subnetz mit Ihrem NAT-Gateway haben. Bei einem Setup mit drei AZs verfügen Sie über drei private Subnetze, drei öffentliche Subnetze und drei NAT-Gateways.

2. Nachdem Sie Ihr NAT-Gateway erstellt haben, öffnen Sie die Amazon VPC-Konsole und wählen Sie im linken Menü Routing-Tabellen aus.
3. Klicken Sie auf Create Route Table (Routing-Tabelle erstellen).
4. Ordnen Sie diese Routing-Tabelle der VPC zu, die Ihren MSK-Cluster enthält. Geben Sie optional einen Namen für Ihre Routing-Tabelle ein.
5. Klicken Sie auf Create Route Table (Routing-Tabelle erstellen).
6. Wählen Sie die Routing-Tabelle aus, die Sie gerade erstellt haben.
7. Wählen Sie auf der Registerkarte Subnetzzuordnungen die Option Subnetzzuordnungen bearbeiten aus.
   - Ordnen Sie diese Routing-Tabelle dem privaten Subnetz zu, das Ihren MSK-Cluster enthält.
8. Wählen Sie Routen bearbeiten aus.
9. Wählen Sie Route hinzufügen aus:
   1. Geben Sie für Ziel die Zeichenfolge `0.0.0.0/0` an.
   2. Wählen Sie für Ziel die Option NAT-Gateway aus.
   3. Wählen Sie im Suchfeld das in Schritt 1 erstellte NAT-Gateway aus. Dies sollte das NAT-Gateway in derselben AZ sein wie das private Subnetz, das Ihren MSK-Cluster enthält (das private Subnetz, das Sie dieser Routing-Tabelle in Schritt 6 zugeordnet haben).
10. Wählen Sie Änderungen speichern aus.

Konfiguration von AWS PrivateLink-Endpunkten für eine MSK-Ereignisquelle

Sie können AWS PrivateLink-Endpunkte so konfigurieren, dass sie Nachrichten von Ihrem Cluster abfragen, und die Funktion über einen Pfad durch Ihre VPC aufrufen. Diese Endpunkte sollten Ihrem MSK-Cluster den Zugriff auf Folgendes ermöglichen:

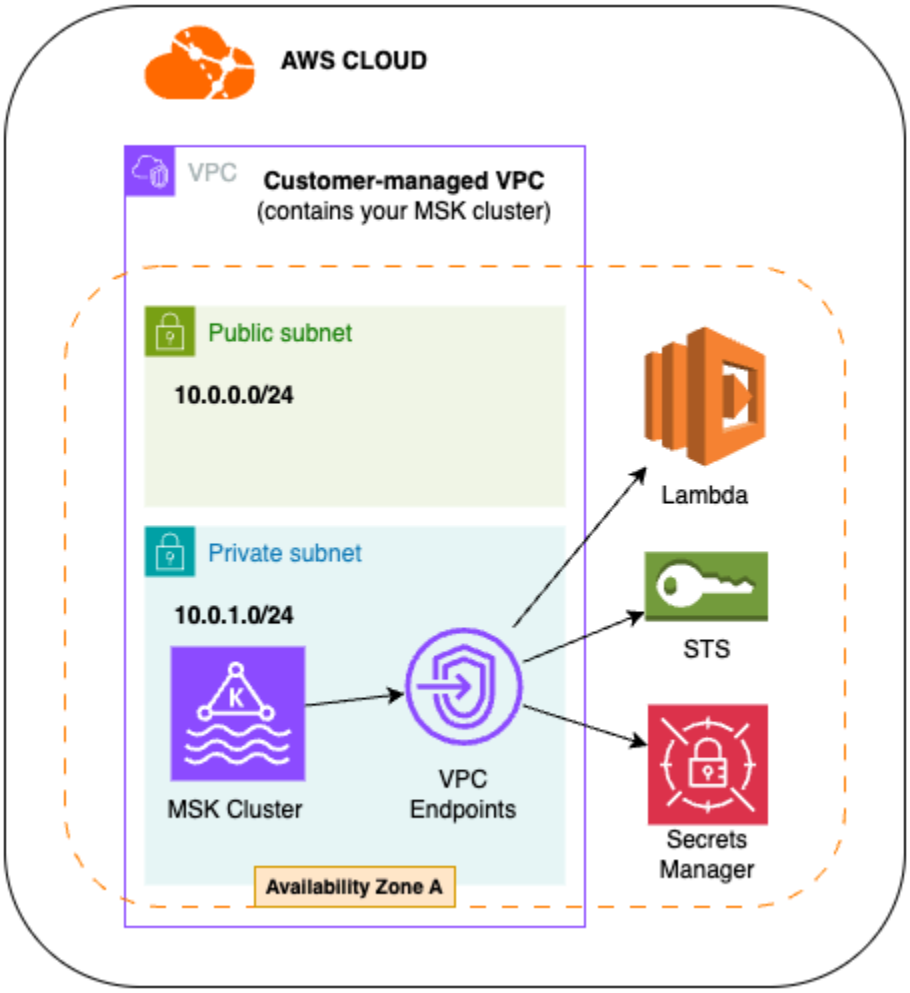
- Den Lambda-Service

- Den AWS -Security-Token-Service (STS)
- Optional den AWS Secrets Manager-Service. Dies ist erforderlich, wenn das für die Cluster-Authentifizierung erforderliche Secret im Secrets Manager gespeichert ist.

Die Konfiguration von PrivateLink-Endpunkten ist nur erforderlich, wenn Ihre Zuordnung von Ereignisquellen den On-Demand-Modus verwendet. Wenn Ihre Zuordnung von Ereignisquellen den Bereitstellungsmodus verwendet, richtet Lambda die erforderlichen Verbindungen für Sie ein.

PrivateLink-Endpunkte ermöglichen einen sicheren, privaten Zugriff auf AWS-Services über AWS PrivateLink. Alternativ können Sie ein NAT-Gateway konfigurieren, um Ihrem MSK-Cluster Zugriff auf das öffentliche Internet zu ermöglichen (siehe the section called “Konfiguration eines NAT-Gateways für eine MSK-Ereignisquelle”).

Nachdem Sie Ihre VPC-Endpunkte konfiguriert haben, sollte Ihr MSK-Cluster direkten und privaten Zugriff auf Lambda, STS und optional Secrets Manager haben.



Die folgenden Schritte führen Sie durch die Konfiguration eines PrivateLink-Endpunkts über die Konsole. Wiederholen Sie diese Schritte nach Bedarf für jeden Endpunkt (Lambda, STS, Secrets Manager).

So konfigurieren Sie VPC-PrivateLink-Endpunkte (Konsole)

1. Öffnen Sie die Amazon VPC-Konsole und wählen Sie im linken Menü Endpunkte aus.
2. Wählen Sie Endpunkt erstellen aus.
3. Geben Sie optional einen Namen für Ihren Endpunkt ein.
4. Wählen Sie für Typ die Option AWS-Services aus.
5. Beginnen Sie unter Services mit der Eingabe des Namens des Service. Um beispielsweise einen Endpunkt für die Verbindung mit Lambda zu erstellen, geben Sie `lambda` in das Suchfeld ein.
6. In den Ergebnissen sollte der Service-Endpunkt in der aktuellen Region angezeigt werden. In der Region USA Ost (Nord-Virginia) sollte beispielsweise `com.amazonaws.us-east-2.lambda` angezeigt werden. Wählen Sie diesen Service aus.
7. Wählen Sie unter Netzwerkeinstellungen die VPC aus, die Ihren MSK-Cluster enthält.
8. Wählen Sie unter Subnetze die AZs aus, in denen sich Ihr MSK-Cluster befindet.
   - Wählen Sie für jede AZ unter Subnetz-ID das private Subnetz aus, das Ihren MSK-Cluster enthält.
9. Wählen Sie unter Sicherheitsgruppen die Ihrem MSK-Cluster zugeordneten Sicherheitsgruppen aus.
10. Wählen Sie Endpunkt erstellen aus.

Standardmäßig verfügen Amazon VPC-Endpunkte über offene IAM-Richtlinien, die einen umfassenden Zugriff auf Ressourcen ermöglichen. Es empfiehlt sich, diese Richtlinien auf die Durchführung der erforderlichen Aktionen über diesen Endpunkt zu beschränken. Für Ihren Secrets-Manager-Endpunkt können Sie beispielsweise dessen Richtlinie so ändern, dass nur die Ausführungsrolle Ihrer Funktion auf das Secret zugreifen kann.

Example VPC-Endpunktrichtlinie – Secrets-Manager-Endpunkt

```
{
    "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"
        }
    ]
}
```

Für die AWS STS- und Lambda-Endpunkte können Sie den aufrufenden Prinzipal auf den Lambda-Service-Prinzipal beschränken. Stellen Sie jedoch sicher, dass Sie in diesen Richtlinien `"Resource": "*"` verwenden.

Example VPC-Endpunktrichtlinie – AWS STS-Endpunkt

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

Example VPC-Endpunktrichtlinie – Lambda-Endpunkt

```
{
    "Statement": [
        {
            "Action": "lambda:InvokeFunction",
            "Effect": "Allow",
            "Principal": {
                "Service": [
```

```
                    "lambda.amazonaws.com"
                ]
            },
            "Resource": "*"
        }
    ]
}
```

Konfiguration von Lambda-Berechtigungen für Amazon MSK-Ereignisquellenzuordnungen

Um auf den Amazon-MSK-Cluster zugreifen zu können, benötigen Ihre Funktion und Zuordnung von Ereignisquellen Berechtigungen zum Ausführen verschiedener Amazon-MSK-API-Aktionen. Fügen Sie diese Berechtigungen zur Ausführungsrolle der Funktion hinzu. Wenn Ihre Benutzer Zugriff benötigen, fügen Sie die erforderlichen Berechtigungen zur Identitätsrichtlinie für Benutzer oder Rolle hinzu.

Die Richtlinie „AWSLambdaMSKExecutionRollenverwaltung“ enthält die erforderlichen Mindestberechtigungen für Amazon MSK Lambda-Ereignisquellenzuordnungen. Um den Genehmigungsprozess zu vereinfachen, können Sie:

- Fügen Sie Ihrer AWSLambdaMSKExecutionAusführungsrolle die Richtlinie „Rollenverwaltete Verwaltung“ hinzu.
- Lassen Sie die Lambda-Konsole die Berechtigungen für Sie generieren. Wenn Sie eine Amazon MSK-Ereignisquellenzuordnung in der Konsole erstellen, bewertet Lambda Ihre Ausführungsrolle und warnt Sie, wenn Berechtigungen fehlen. Wählen Sie Berechtigungen generieren, um Ihre Ausführungsrolle automatisch zu aktualisieren. Dies funktioniert nicht, wenn Sie Ihre Richtlinien für Ausführungsrollen manuell erstellt oder geändert haben oder wenn die Richtlinien mehreren Rollen zugeordnet sind. Beachten Sie, dass für Ihre Ausführungsrolle möglicherweise weiterhin zusätzliche Berechtigungen erforderlich sind, wenn Sie erweiterte Funktionen wie On-Failure Destination oder AWS Glue Schema Registry verwenden.

Themen

- Erforderliche Berechtigungen
- Optionale Berechtigungen

Erforderliche Berechtigungen

Ihre Lambda-Funktionsausführungsrolle muss über die folgenden erforderlichen Berechtigungen für Amazon MSK-Ereignisquellenzuordnungen verfügen. Diese Berechtigungen sind in der Richtlinie „Rollenverwaltung“ enthalten. AWSLambda MSKExecution

CloudWatch Protokolliert Berechtigungen

Die folgenden Berechtigungen ermöglichen es Lambda, Protokolle in Amazon CloudWatch Logs zu erstellen und zu speichern.

- Protokolle: CreateLogGroup
- Protokolle: CreateLogStream
- Protokolle: PutLogEvents

MSK-Cluster-Berechtigungen

Die folgenden Berechtigungen ermöglichen Lambda, in Ihrem Namen auf Ihren Amazon MSK-Cluster zuzugreifen:

- Kafka: DescribeCluster
- Kafka: V2 DescribeCluster
- Kafka: GetBootstrapBrokers

Wir empfehlen die Verwendung von kafka: DescribeCluster V2 anstelle von kafka:. DescribeCluster Die v2-Berechtigung funktioniert sowohl mit bereitgestellten als auch mit serverlosen Amazon MSK-Clustern. In Ihrer Richtlinie benötigen Sie nur eine dieser Berechtigungen.

VPC-Berechtigungen

Die folgenden Berechtigungen ermöglichen es Lambda, Netzwerkschnittstellen zu erstellen und zu verwalten, wenn eine Verbindung zu Ihrem Amazon MSK-Cluster hergestellt wird:

- ec2: CreateNetworkInterface
- ec2: DescribeNetworkInterfaces
- ec2: DescribeVpcs
- ec2: DeleteNetworkInterface
- ec2: DescribeSubnets

- ec2: DescribeSecurityGroups

Optionale Berechtigungen

Ihre Lambda-Funktion benötigt möglicherweise auch Berechtigungen für Folgendes:

- Greifen Sie auf kontenübergreifende Amazon MSK-Cluster zu. Für kontenübergreifende Zuordnungen von Ereignisquellen benötigen Sie kafka: in der Ausführungsrolle. DescribeVpcConnection Ein IAM-Prinzipal, der eine kontenübergreifende Zuordnung von Ereignisquellen erstellt, benötigt kafka:. ListVpcConnections
- Greifen Sie auf Ihr SCRAM-Secret zu, wenn Sie die SASL/SCRAM-Authentifizierung verwenden. Dadurch kann Ihre Funktion einen Benutzernamen und ein Passwort verwenden, um eine Verbindung zu Kafka herzustellen.
- Beschreiben Sie Ihr Secrets Manager-Geheimnis, falls Sie die SASL/SCRAM mTLS-Authentifizierung verwenden. Dadurch kann Ihre Funktion die für sichere Verbindungen erforderlichen Anmeldeinformationen oder Zertifikate abrufen.
- Greifen Sie auf Ihren vom AWS KMS Kunden verwalteten Schlüssel zu, wenn Ihr AWS Secrets Manager Geheimnis mit einem vom AWS KMS Kunden verwalteten Schlüssel verschlüsselt ist.
- Greifen Sie auf Ihre Schemaregistrierungs-Secrets zu, wenn Sie eine Schemaregistrierung mit Authentifizierung verwenden:
  - Für AWS Glue Schema Registry: Ihre Funktionsanforderungen `glue:GetRegistry` und `glue:GetSchemaVersion` Berechtigungen. Dadurch kann Ihre Funktion die in AWS Glue gespeicherten Nachrichtenformatregeln nachschlagen und verwenden.
  - Für Confluent Schema Registry mit `BASIC_AUTH` oder`CLIENT_CERTIFICATE_TLS_AUTH`: Ihre Funktion benötigt die Berechtigung `secretsmanager:GetSecretValue` für das Secret, das die Authentifizierungsdaten enthält. Auf diese Weise kann Ihre Funktion die Oder-Zertifikate abrufen, die username/password für den Zugriff auf die Confluent Schema Registry erforderlich sind.
  - Für private CA-Zertifikate: Ihre Funktion benötigt die GetSecretValue Berechtigung secretsmanager: für das Secret, das das Zertifikat enthält. Dadurch kann Ihre Funktion die Identität von Schemaregistrierungen überprüfen, die benutzerdefinierte Zertifikate verwenden.
- Greifen Sie auf Kafka-Cluster-Nutzergruppen und Abfragemeldungen aus dem Thema zu, wenn Sie die IAM-Authentifizierung für die Zuordnung der Ereignisquellen verwenden.

Diese entsprechen den folgenden erforderlichen Berechtigungen:

- kafka: ListScramSecrets — Ermöglicht die Auflistung von SCRAM-Geheimnissen für die Kafka-Authentifizierung
- secretsmanager: GetSecretValue - Ermöglicht das Abrufen von Geheimnissen aus Secrets Manager
- kms:Decrypt — Ermöglicht die Entschlüsselung verschlüsselter Daten mit AWS KMS
- glue: GetRegistry - Ermöglicht den Zugriff auf die Schemaregistrierung AWS Glue
- glue: GetSchemaVersion - Ermöglicht das Abrufen bestimmter Schemaversionen aus der AWS Glue Schemaregistry
- kafka-cluster:Connect — Erteilt die Erlaubnis, eine Verbindung zum Cluster herzustellen und sich dort zu authentifizieren
- kafka-cluster: AlterGroup - Erteilt die Erlaubnis, Gruppen in einem Cluster beizutreten, was der READ GROUP-ACL von Apache Kafka entspricht
- kafka-cluster: DescribeGroup - Erteilt die Erlaubnis, Gruppen in einem Cluster zu beschreiben, was der DESCRIBE GROUP-ACL von Apache Kafka entspricht
- kafka-cluster: DescribeTopic - Erteilt die Erlaubnis, Themen in einem Cluster zu beschreiben, was der DESCRIBE TOPIC-ACL von Apache Kafka entspricht
- kafka-cluster: ReadData - Erteilt die Berechtigung, Daten aus Themen in einem Cluster zu lesen, was der READ TOPIC-ACL von Apache Kafka entspricht

Wenn Sie außerdem Aufzeichnungen fehlgeschlagener Aufrufe an ein Ziel für Fehlerfälle senden möchten, benötigen Sie je nach Zieltyp die folgenden Berechtigungen:

- Für Amazon SQS SQS-Ziele: sqs: SendMessage — Ermöglicht das Senden von Nachrichten an eine Amazon SQS SQS-Warteschlange
- Für Amazon-SNS-Ziele: sns:Publish – erlaubt die Veröffentlichung von Nachrichten zu einem Amazon-SNS-Thema
- Für Amazon S3 S3-Bucket-Ziele: s3: PutObject und s3: ListBucket — Ermöglicht das Schreiben und Auflisten von Objekten in einem Amazon S3 S3-Bucket

Informationen zur Behebung von Authentifizierungs- und Autorisierungsfehlern finden Sie unter the section called “Fehlerbehebung”.

Konfigurieren von Amazon MSK-Ereignisquellen für Lambda

Um einen selbstverwalteten Amazon-MSK-Cluster als Ereignisquelle für Ihre Lambda-Funktion zu verwenden, erstellen Sie eine Zuordnung von Ereignisquellen, die die beiden Ressourcen miteinander verbindet. Auf dieser Seite wird beschrieben, wie Sie eine Zuordnung von Ereignisquellen für Amazon MSK erstellen.

Auf dieser Seite wird davon ausgegangen, dass Sie Ihren MSK-Cluster und die Amazon Virtual Private Cloud (VPC), in der er sich befindet, bereits ordnungsgemäß konfiguriert haben. Wenn Sie Ihren Cluster oder Ihre VPC einrichten müssen, finden Sie entsprechende Informationen unter the section called “Einrichtung von Cluster und Netzwerk”. Informationen zum Konfigurieren des Wiederholungsverhaltens für die Fehlerbehandlung finden Sie unterthe section called “Versuchen Sie es erneut mit den Konfigurationen”.

Themen

- Verwenden eines Amazon MSK-Clusters als Ereignisquelle
- Konfiguration von Amazon MSK-Cluster-Authentifizierungsmethoden in Lambda
- Erstellen einer Lambda-Zuordnung von Ereignisquellen für eine Amazon-MSK-Ereignisquelle
- Erstellen von kontenübergreifenden Zuordnungen von Ereignisquellen in Lambda
- Alle Amazon-MSK-Konfigurationsparameter von Ereignisquellen in Lambda

Verwenden eines Amazon MSK-Clusters als Ereignisquelle

Wenn Sie Ihren Apache Kafka-Cluster oder Amazon MSK als Auslöser für Ihre Lambda-Funktion hinzufügen, wird der Cluster als Ereignisquelle verwendet.

Lambda liest Ereignisdaten aus den Kafka-Themen, die Sie wie `Topics` in einer CreateEventSourceMappingAnfrage angeben, basierend auf der von Ihnen angegebenen Startposition. Nach erfolgreicher Verarbeitung wird Ihr Kafka-Thema Ihrem Kafka-Cluster zugeordnet.

Lambda liest Nachrichten sequentiell für jede Kafka-Themenpartition. Eine einzelne Lambda-Nutzlast kann Nachrichten von mehreren Partitionen enthalten. Wenn mehr Datensätze verfügbar sind, setzt Lambda die Verarbeitung von Datensätzen stapelweise fort, basierend auf dem BatchSize Wert, den Sie in einer CreateEventSourceMappingAnfrage angeben, bis Ihre Funktion das Thema eingeholt hat.

Nachdem Lambda jeden Batch verarbeitet hat, werden die Offsets der Nachrichten in diesem Batch festgeschrieben. Wenn Ihre Funktion einen Fehler für eine der Nachrichten in einem Batch zurückgibt, wiederholt Lambda den gesamten Nachrichtenbatch, bis die Verarbeitung erfolgreich

ist oder die Nachrichten ablaufen. Sie können Datensätze, bei denen alle Wiederholungsversuche fehlschlagen, zur späteren Verarbeitung an ein Ausfallziel senden.

> **Note**
>
> Während Lambda-Funktionen in der Regel ein maximales Timeout-Limit von 15 Minuten haben, unterstützen Ereignisquellenzuordnungen für Amazon MSK, selbstverwaltetes Apache Kafka, Amazon DocumentDB, Amazon MQ für ActiveMQ und RabbitMQ nur Funktionen mit einem maximalen Timeout-Limit von 14 Minuten.

Konfiguration von Amazon MSK-Cluster-Authentifizierungsmethoden in Lambda

Lambda benötigt eine Berechtigung, um auf Ihren Amazon-MSK-Cluster zuzugreifen, Datensätze abzurufen und andere Aufgaben auszuführen. Amazon MSK unterstützt mehrere Methoden zur Authentifizierung bei Ihrem MSK-Cluster.

Cluster-Authentifizierungsmethoden

- Nicht authentifizierter Zugriff
- SASL/SCRAM-Authentifizierung
- Gegenseitige TLS-Authentifizierung
- IAM-Authentifizierung
- So wählt Lambda einen Bootstrap-Broker

Nicht authentifizierter Zugriff

Wenn keine Clients über das Internet auf den Cluster zugreifen, können Sie einen nicht authentifizierten Zugriff verwenden.

SASL/SCRAM-Authentifizierung

Lambda unterstützt die Authentifizierung Simple Authentication und Security Layer/Salted Challenge Response Authentication Mechanism (SASL/SCRAM) mit der SHA-512-Hash-Funktion und der Transport Layer Security (TLS) -Verschlüsselung. Damit Lambda eine Verbindung zum Cluster herstellen kann, speichern Sie die Authentifizierungsanmeldeinformationen (Benutzername und Passwort) in einem Secrets-Manager-Geheimnis und verweisen Sie bei der Konfiguration Ihrer Zuordnung von Ereignisquellen auf dieses Geheimnis.

Weitere Informationen zur Verwendung von Secrets Manager finden Sie unter Authentifizierung der Anmeldeinformationen mit Secrets Manager im Entwicklerhandbuch für Amazon Managed Streaming for Kafka.

> **Note**
>
> Amazon MSK unterstützt keine SASL/PLAIN Authentifizierung.

Gegenseitige TLS-Authentifizierung

Gegenseitige TLS (mTLS) bietet eine bidirektionale Authentifizierung zwischen Client und Server. Der Client sendet ein Zertifikat an den Server, damit dieser den Client überprüfen kann. Der Server sendet ebenfalls ein Zertifikat an den Client, damit dieser den Server überprüfen kann.

Bei Amazon-MSK-Integrationen mit Lambda fungiert Ihr MSK-Cluster als Server und Lambda als Client.

- Damit Lambda Ihren MSK-Cluster überprüfen kann, konfigurieren Sie ein Client-Zertifikat als Geheimnis in Secrets Manager und verweisen Sie in Ihrer Konfiguration der Zuordnung von Ereignisquellen auf dieses Zertifikat. Das Clientzertifikat muss von einer Zertifizierungsstelle (CA) im Trust Store des Servers signiert sein.
- Der MSK-Cluster sendet auch ein Serverzertifikat an Lambda. Das Serverzertifikat muss von einer Zertifizierungsstelle (CA) im AWS Trust Store signiert werden.

Amazon MSK unterstützt keine selbstsignierten Serverzertifikate. Alle Broker in Amazon MSK verwenden öffentliche Zertifikate, die von Amazon Trust Services signiert wurden CAs, denen Lambda standardmäßig vertraut.

Konfigurieren des mTLS-Secrets

Das Secret CLIENT_CERTIFICATE_TLS_AUTH erfordert ein Zertifikatfeld und ein Feld für einen privaten Schlüssel. Für einen verschlüsselten privaten Schlüssel erfordert das Secret ein Passwort für den privaten Schlüssel. Sowohl das Zertifikat als auch der private Schlüssel müssen im PEM-Format vorliegen.

> **Note**
>
> Lambda unterstützt die Verschlüsselungsalgorithmen mit privaten Schlüsseln PBES1(aber nicht PBES2).

Das Zertifikatfeld muss eine Liste von Zertifikaten enthalten, beginnend mit dem Client-Zertifikat, gefolgt von etwaigen Zwischenzertifikaten und endend mit dem Root-Zertifikat. Jedes Zertifikat muss in einer neuen Zeile mit der folgenden Struktur beginnen:

```
-----BEGIN CERTIFICATE-----
        <certificate contents>
-----END CERTIFICATE-----
```

Secrets Manager unterstützt Secrets von bis zu 65 536 Bytes, was genügend Platz für lange Zertifikatsketten bietet.

Der private Schlüssel muss im Format PKCS #8 mit folgender Struktur vorliegen:

```
-----BEGIN PRIVATE KEY-----
         <private key contents>
-----END PRIVATE KEY-----
```

Verwenden Sie für einen verschlüsselten privaten Schlüssel die folgende Struktur:

```
-----BEGIN ENCRYPTED PRIVATE KEY-----
          <private key contents>
-----END ENCRYPTED PRIVATE KEY-----
```

Im folgenden Beispiel sehen Sie den Inhalt eines Secrets für mTLS-Authentifizierung mit einem verschlüsselten privaten Schlüssel. Fügen Sie für einen verschlüsselten privaten Schlüssel das Passwort für den privaten Schlüssel in das Secret ein.

```
{
 "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-----"
}
```

Weitere Informationen zu mTLS für Amazon MSK und Anweisungen zum Generieren eines Clientzertifikats finden Sie unter Gegenseitige TLS-Authentifizierung für Amazon MSK im Entwicklerhandbuch für Amazon Managed Streaming for Apache Kafka.

IAM-Authentifizierung

Sie können AWS Identity and Access Management (IAM) verwenden, um die Identität von Clients zu authentifizieren, die eine Verbindung zum MSK-Cluster herstellen. Bei der IAM-Authentifizierung nutzt Lambda die Berechtigungen in der Ausführungsrolle Ihrer Funktion, um eine Verbindung zum Cluster herzustellen, Datensätze abzurufen und andere erforderliche Aktionen auszuführen. Ein Beispiel für eine Richtlinie, welche die erforderlichen Berechtigungen enthält, finden Sie unter Erstellen von Autorisierungsrichtlinien für die IAM-Rolle im Entwicklerhandbuch für Amazon Managed Streaming for Apache Kafka.

Wenn die IAM-Authentifizierung für Ihren MSK-Cluster aktiv ist und Sie kein Geheimnis angeben, verwendet Lambda automatisch standardmäßig die IAM-Authentifizierung.

Weitere Informationen zur IAM-Authentifizierung in Amazon MSK finden Sie unter IAM-Zugriffskontrolle.

So wählt Lambda einen Bootstrap-Broker

Lambda wählt einen Bootstrap-Broker basierend auf den auf Ihrem Cluster verfügbaren Authentifizierungsmethoden aus und ob Sie ein Geheimnis für die Authentifizierung angeben.. Wenn Sie ein Geheimnis für MTLs oder SASL/SCRAM angeben, wählt Lambda automatisch

diese Authentifizierungsmethode. Wenn Sie kein Geheimnis angeben, wählt Lambda die stärkste Authentifizierungsmethode aus, die in Ihrem Cluster aktiv ist. Im Folgenden ist die Reihenfolge der Priorität aufgeführt, in der Lambda einen Broker auswählt, von der stärksten zur schwächsten Authentifizierung:

- mTLS (Geheimnis für mTLS bereitgestellt)
- SASL/SCRAM (secret provided for SASL/SCRAM)
- SASL IAM (kein Geheimnis angegeben und IAM-Authentifizierung aktiv)
- Nicht authentifiziertes TLS (kein Geheimnis bereitgestellt und IAM-Authentifizierung nicht aktiv)
- Klartext (kein Geheimnis angegeben und sowohl IAM-Authentifizierung als auch nicht authentifiziertes TLS sind nicht aktiv)

> **Note**
>
> Wenn Lambda keine Verbindung zum sichersten Brokertyp herstellen kann, versucht Lambda nicht, eine Verbindung zu einem anderen (schwächeren) Brokertyp herzustellen. Wenn Sie möchten, dass Lambda einen schwächeren Brokertyp wählt, deaktivieren Sie alle stärkeren Authentifizierungsmethoden in Ihrem Cluster.

Erstellen einer Lambda-Zuordnung von Ereignisquellen für eine Amazon-MSK-Ereignisquelle

Zum Erstellen einer Zuordnung von Ereignisquellen können Sie die Lambda-Konsole, die AWS Command Line Interface (CLI) oder ein AWS SDK verwenden.

> **Note**
>
> Wenn Sie die Zuordnung von Ereignisquellen erstellen, erstellt Lambda eine Hyperplane-ENI im privaten Subnetz, das Ihren MSK-Cluster enthält, sodass Lambda eine sichere Verbindung herstellen kann. Diese Hyperplane-ENI ermöglicht die Verwendung der Subnetz- und Sicherheitsgruppenkonfiguration Ihres MSK-Clusters, nicht Ihrer Lambda-Funktion.

Mit den folgenden Schritten in der Konsole fügen Sie einen Amazon-MSK-Cluster als Auslöser für Ihre Lambda-Funktion hinzu. Im Hintergrund wird dadurch eine Zuordnung von Ereignisquellen erstellt.

So fügen Sie Ihrer Lambda-Funktion (Konsole) einen Amazon MSK-Trigger hinzu

1. Öffnen Sie die Seite Funktion der Lambda-Konsole.
2. Wählen Sie den Namen der Lambda-Funktion aus, der Sie einen Amazon-MSK-Auslöser hinzufügen möchten.
3. Wählen Sie unter Function overview (Funktionsübersicht) die Option Add trigger (Trigger hinzufügen).
4. Wählen Sie unter Auslöser-Konfiguration die Option MSK aus.
5. Führen Sie die folgenden Schritte aus, um Ihre Kafka-Cluster-Details anzugeben:
    1. Wählen Sie für MSK-Cluster Ihren Cluster aus.
    2. Geben Sie für Themenname den Namen des Kafka-Themas ein, von dem Nachrichten verbraucht werden sollen.
    3. Geben Sie für Verbrauchergruppen-ID ggf. die ID einer Kafka-Verbrauchergruppe ein, der Sie beitreten möchten. Weitere Informationen finden Sie unter the section called “Verbrauchergruppen-ID”.
6. Nehmen Sie für die Cluster-Authentifizierung die erforderlichen Konfigurationen vor. Weitere Informationen über Cluster-Authentifizierung finden Sie unter the section called “Cluster-Authentifizierung”.
    - Aktivieren Sie Authentifizierung verwenden, wenn Lambda beim Verbindungsaufbau die Authentifizierung bei Ihrem MSK-Cluster durchführen soll. Die Authentifizierung wird empfohlen.
    - Wenn Sie Authentifizierung verwenden, wählen Sie für Authentifizierungsmethode die zu verwendende Authentifizierungsmethode aus.
    - Wenn Sie Authentifizierung verwenden, wählen Sie für Secrets-Manager-Schlüssel den Secrets-Manager-Schlüssel aus, der die für den Zugriff auf Ihren Cluster erforderlichen Authentifizierungsanmeldeinformationen enthält.
7. Nehmen Sie unter Konfiguration des Ereignis-Pollers die erforderlichen Konfigurationen vor.
    - Wählen Sie Auslöser aktivieren, um den Auslöser unmittelbar nach der Erstellung zu aktivieren.
    - Wählen Sie aus, ob Sie den Bereitstellungsmodus für Ihre Zuordnung von Ereignisquellen konfigurieren möchten. Weitere Informationen finden Sie unter the section called “Skalierung des Ereignis-Pollers”.

- Wenn Sie den Bereitstellungsmodus konfigurieren, geben Sie einen Wert für Minimale Anzahl von Ereignisabfragen, einen Wert für Maximale Anzahl von Ereignisabfragen und einen optionalen Wert ein, PollerGroupName um die Gruppierung mehrerer ESMs innerhalb derselben Ereignisquellen-VPC anzugeben.

- Wählen Sie unter Startposition aus, wie Lambda mit dem Lesen aus Ihrem Stream beginnen soll. Weitere Informationen finden Sie unter the section called “Abfrage und Stream-Positionen”.

8. Nehmen Sie unter Batching die erforderlichen Konfigurationen vor. Weitere Informationen zu Batching finden Sie unter the section called “Batching-Verhalten”.

   1. Geben Sie für Batchgröße die maximale Anzahl von Nachrichten ein, die in einem einzelnen Batch empfangen werden sollen.
   2. Geben Sie für Batch-Fenster die maximale Zeit in Sekunden ein, die Lambda mit dem Sammeln von Datensätzen verbringt, bevor die Funktion aufgerufen wird.

9. Nehmen Sie unter Filterung die erforderlichen Konfigurationen vor. Weitere Informationen zur Filterung erhalten Sie unter the section called “Ereignisfilterung”.

   - Fügen Sie unter Filterkriterien Definitionen für Filterkriterien hinzu, um zu bestimmen, ob ein Ereignis verarbeitet werden soll oder nicht.

10. Nehmen Sie unter Fehlerbehandlung die erforderlichen Konfigurationen vor. Weitere Informationen zur Fehlerbehandlung finden Sie unter the section called “Fehlgeschlagene Aufrufe beibehalten”.

    - Geben Sie für Fehlerziel den ARN Ihres Fehlerziels an.

11. Geben Sie unter Tags die Tags ein, die dieser Zuordnung von Ereignisquellen zugeordnet werden sollen.
12. Wählen Sie hinzufügen aus, um den Auslöser zu erstellen.

Sie können die Ereignisquellenzuordnung auch mithilfe der AWS CLI mit dem create-event-source-mappingBefehl erstellen. Im folgenden Beispiel wird eine Zuordnung von Ereignisquellen erstellt, um die Lambda-Funktion `my-msk-function` ab der `LATEST`-Nachricht dem `AWSKafkaTopic`-Thema zuzuordnen. Dieser Befehl verwendet das SourceAccessConfigurationObjekt auch, um Lambda anzuweisen, bei der Verbindung mit dem Cluster die SASL/SCRAM-Authentifizierung zu verwenden.

```
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"}]'
```

Wenn der Cluster mTLS-Authentifizierung verwendet, fügen Sie ein SourceAccessConfigurationObjekt hinzu, das einen Secrets Manager Manager-Schlüssel-ARN spezifiziert`CLIENT_CERTIFICATE_TLS_AUTH`. Das wird im folgenden Befehl veranschaulicht:

```
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"}]'
```

Wenn der Cluster die IAM-Authentifizierung verwendet, benötigen Sie kein SourceAccessConfigurationObjekt. Das wird im folgenden Befehl veranschaulicht:

```
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
```

Erstellen von kontenübergreifenden Zuordnungen von Ereignisquellen in Lambda

Sie können private Multi-VPC-Konnektivität verwenden, um eine Lambda-Funktion mit einem bereitgestellten MSK-Cluster in einem anderen AWS-Konto zu verbinden. Multi-VPC-Konnektivität verwendet AWS PrivateLink, wodurch der gesamte Datenverkehr im AWS Netzwerk bleibt.

> **Note**
>
> Sie können keine kontenübergreifenden Zuordnungen von Ereignisquellen für Serverless-MSK-Cluster erstellen.

Um eine kontenübergreifende Zuordnung von Ereignisquellen zu erstellen, müssen Sie zunächst die Multi-VPC-Konnektivität für den MSK-Cluster konfigurieren. Verwenden Sie beim Erstellen der Zuordnung von Ereignisquellen den ARN der verwalteten VPC-Verbindung anstelle des Cluster-ARNs, wie in den folgenden Beispielen gezeigt. Der CreateEventSourceMappingVorgang unterscheidet sich auch je nachdem, welchen Authentifizierungstyp der MSK-Cluster verwendet.

Example– Erstellen Sie eine kontoübergreifende Zuordnung von Ereignisquellen für einen Cluster, der die IAM-Authentifizierung verwendet

Wenn der Cluster die rollenbasierte IAM-Authentifizierung verwendet, benötigen Sie kein Objekt. SourceAccessConfiguration Beispiel:

```
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— Erstellen Sie eine kontenübergreifende Zuordnung von Ereignisquellen für einen Cluster, der Authentifizierung verwendet SASL/SCRAM

Wenn der Cluster die SASL/SCRAM-Authentifizierung verwendet, müssen Sie ein SourceAccessConfigurationObjekt angeben, das einen geheimen Secrets Manager Manager-ARN spezifiziert, `SASL_SCRAM_512_AUTH` und einen geheimen ARN hinzufügen.

Es gibt zwei Möglichkeiten, Geheimnisse für kontoübergreifende Amazon MSK-Ereignisquellenzuordnungen mit Authentifizierung zu verwenden: SASL/SCRAM

- Erstellen Sie ein Secret im Lambda-Funktionskonto und synchronisieren Sie es mit dem Cluster-Secret. Erstellen Sie eine Rotation, um die beiden Secrets synchron zu halten. Mit dieser Option können Sie das Secret vom Funktionskonto aus kontrollieren.
- Verwenden Sie das Secret, das dem MSK-Cluster zugeordnet ist. Dieses Secret muss kontoübergreifenden Zugriff auf das Lambda-Funktionskonto ermöglichen. Weitere Informationen

finden Sie unter Berechtigungen für AWS Secrets Manager -Secrets für Benutzer in einem anderen Konto.

```
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– Erstellen Sie eine kontoübergreifende Zuordnung von Ereignisquellen für einen Cluster, der die mTLS-Authentifizierung verwendet

Wenn der Cluster mTLS-Authentifizierung verwendet, müssen Sie ein SourceAccessConfigurationObjekt angeben, das einen geheimen ARN von Secrets Manager spezifiziert`CLIENT_CERTIFICATE_TLS_AUTH`. Das Secret kann im Clusterkonto oder im Lambda-Funktionskonto gespeichert werden.

```
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"}]'
```

Alle Amazon-MSK-Konfigurationsparameter von Ereignisquellen in Lambda

Alle Lambda-Ereignisquelltypen verwenden dieselben CreateEventSourceMappingUpdateEventSourceMappingAPI-Operationen. Allerdings gelten nur einige der Parameter für selbstverwaltetes Amazon MSK, wie in der folgenden Tabelle dargestellt.

Parameter	Erforderlich	Standard	Hinweise
AmazonMan agedKafkaEventSour ceConfig	N	Enthält das ConsumerGroupId Feld, das standardm	Kann nur auf „Erstelle n“ festgelegt werden

Parameter	Erforderlich	Standard	Hinweise
		äßig einen eindeutig en Wert hat.	
BatchSize	N	100	Höchstwert: 10 000.
DestinationConfig	N	–	the section called “Fehlgeschlagene Aufrufe beibehalten”
Aktiviert	N	Wahr	
BisectBatchOnFunctionError	N	Falsch	the section called “Versuchen Sie es erneut mit den Konfigurationen”
FunctionResponseTypes	N	–	the section called “Versuchen Sie es erneut mit den Konfigurationen”
MaximumRecordAgeInSeconds	N	-1 (unendlich)	the section called “Versuchen Sie es erneut mit den Konfigurationen”
MaximumRetryAttempts	N	-1 (unendlich)	the section called “Versuchen Sie es erneut mit den Konfigurationen”
EventSourceArn	Y	–	Kann nur auf „Erstelle n“ festgelegt werden
FilterCriteria	N	–	Steuern Sie, welche Ereignisse Lambda an Ihre Funktion sendet

Parameter	Erforderlich	Standard	Hinweise
FunctionName	Y	–	
KMSKeyArn	N	–	the section called “Verschlüsselung der Filterkriterien”
MaximumBatchingWindowInSeconds	N	500 ms	Batching-Verhalten
ProvisionedPollersConfig	N	`MinimumPollers` : Standardwert von 1, wenn nicht angegeben `MaximumPollers` : Standardwert von 200, wenn nicht angegeben `PollerGroupName` : N/A	the section called “Modus bereitgestellter Kapazität”
SourceAccessConfigurations	N	Keine Anmeldedaten	Anmeldeinformationen zur SASL/SCRAM- oder CLIENT_CERTIFICATE_TLS_AUTH (MutualTLS)-Authentifizierung für Ihre Ereignisquelle
StartingPosition	Y	–	AT_TIMESTAMP, TRIM_HORIZON, oder LATEST Kann nur auf „Erstellen“ festgelegt werden

Parameter	Erforderlich	Standard	Hinweise
StartingPositionTimestamp	N	–	Erforderlich, wenn es auf StartingPosition AT_TIMESTAMP gesetzt ist
Tags (Markierungen)	N	–	the section called “Tags für die Zuordnung von Ereignisquellen”
Topics	Y	–	Kafka-Thema-Name Kann nur auf „Erstelle n“ festgelegt werden

Note

Wenn Sie a angeben`PollerGroupName`, können sich mehrere ESMs innerhalb derselben Amazon VPC die Kapazität der Event Poller Unit (EPU) teilen. Sie können diese Option verwenden, um die Kosten für den Bereitstellungsmodus für Sie zu optimieren. ESMs Anforderungen für die ESM-Gruppierung:

- ESMs muss sich in derselben Amazon-VPC befinden
- Maximal 100 ESMs pro Befragungsgruppe
- Die Gesamtzahl der Befragten ESMs in einer Gruppe darf 2000 nicht überschreiten

Sie können den aktualisieren`PollerGroupName`, um einen ESM in eine andere Gruppe zu verschieben, oder einen ESM aus einer Gruppe entfernen, indem Sie ihn `PollerGroupName` auf eine leere Zeichenfolge („“) setzen.

Tutorial: Verwenden einer Amazon-MSK-Zuordnung von Ereignisquellen zum Aufrufen einer Lambda-Funktion

In diesem Tutorial führen Sie Folgendes durch:

- Erstellen Sie eine Lambda-Funktion in demselben AWS Konto wie ein vorhandener Amazon MSK-Cluster.
- Konfigurieren Sie Netzwerk und Authentifizierung für Lambda für die Kommunikation mit Amazon MSK.
- Richten Sie eine Lambda Amazon MSK-Zuordnung von Ereignisquellen ein, die Ihre Lambda-Funktion ausführt, wenn Ereignisse im Thema auftauchen.

Nachdem Sie diese Schritte durchgeführt haben, können Sie, wenn Ereignisse an Amazon MSK gesendet werden, eine Lambda-Funktion einrichten, um diese Ereignisse automatisch mit Ihrem eigenen benutzerdefinierten Lambda-Code zu verarbeiten.

Was können Sie mit diesem Feature machen?

Beispiellösung: Verwenden Sie eine MSK-Zuordnung von Ereignisquellen, um Ihren Kunden Live-Ergebnisse zu liefern.

Stellen Sie sich das folgende Szenario vor: Ihr Unternehmen hostet eine Webanwendung, mit der Ihre Kunden Informationen über Live-Ereignisse, z. B. Sportspiele, abrufen können. Aktuelle Informationen aus dem Spiel werden Ihrem Team über ein Kafka-Thema auf Amazon MSK zur Verfügung gestellt. Sie möchten eine Lösung entwerfen, die Aktualisierungen aus dem MSK-Thema abruft, um den Kunden in einer von Ihnen entwickelten Anwendung eine aktualisierte Ansicht des Live-Ereignisses zu bieten. Sie haben sich für den folgenden Designansatz entschieden: Ihre Client-Anwendungen werden mit einem Serverless-Backend kommunizieren, das in AWS gehostet wird. Clients stellen mithilfe der Amazon WebSocket API Gateway eine Verbindung über Websocket-Sitzungen her.

Bei dieser Lösung benötigen Sie eine Komponente, die MSK-Ereignisse liest, eine benutzerdefinierte Logik ausführt, um diese Ereignisse für die Anwendungsschicht vorzubereiten und diese Informationen dann an die API-Gateway-API weiterleitet. Sie können diese Komponente implementieren AWS Lambda, indem Sie Ihre benutzerdefinierte Logik in einer Lambda-Funktion bereitstellen und sie dann mit einer AWS Lambda Amazon MSK-Ereignisquellenzuordnung aufrufen.

Weitere Informationen zur Implementierung von Lösungen mit der Amazon API Gateway WebSocket API finden Sie WebSocket in den API-Tutorials in der API Gateway-Dokumentation.

Voraussetzungen

Ein AWS Konto mit den folgenden vorkonfigurierten Ressourcen:

Um diese Voraussetzungen zu erfüllen, empfehlen wir den Abschnitt Erste Schritte mit Amazon MSK in der Amazon MSK-Dokumentation.

- Ein Amazon-MSK-Cluster. Siehe Erstellen eines Amazon MSK-Clusters in Erste Schritte mit Amazon MSK.
- Die folgende Konfiguration:
  - Vergewissern Sie sich, dass die rollenbasierte IAM-Authentifizierung in den Sicherheitseinstellungen Ihres Clusters aktiviert ist. Dies verbessert Ihre Sicherheit, da Ihre Lambda-Funktion nur auf die benötigten Amazon MSK-Ressourcen zugreifen kann. Dies ist bei neuen Amazon MSK-Clustern standardmäßig aktiviert.
  - Vergewissern Sie sich, dass der öffentliche Zugang in den Netzwerkeinstellungen Ihres Clusters deaktiviert ist. Wenn Sie den Zugang Ihres Amazon MSK Clusters zum Internet einschränken, erhöht sich Ihre Sicherheit, da die Anzahl der Vermittler, die Ihre Daten verarbeiten, begrenzt wird. Dies ist bei neuen Amazon MSK-Clustern standardmäßig aktiviert.
- Ein Kafka-Thema in Ihrem Amazon MSK-Cluster, das Sie für diese Lösung verwenden können. Weitere Informationen finden Sie unter Erstellen eines Themas unter Erste Schritte mit Amazon MSK.
- Ein Kafka-Admin-Host, der so eingerichtet ist, dass er Informationen aus Ihrem Kafka-Cluster abruft und Kafka-Ereignisse zum Testen an Ihr Thema sendet, z. B. eine EC2 Amazon-Instance, auf der die Kafka-Admin-CLI und die Amazon MSK IAM-Bibliothek installiert sind. Siehe Erstellen eines Client-Rechners in Erste Schritte mit Amazon MSK.

Sobald Sie diese Ressourcen eingerichtet haben, sammeln Sie die folgenden Informationen aus Ihrem AWS Konto, um zu bestätigen, dass Sie bereit sind, fortzufahren.

- Der Name Ihres Amazon MSK-Clusters. Sie finden diese Informationen in der Amazon-MSK-Konsole.
- Die Cluster-UUID, Teil des ARN für Ihren Amazon MSK-Cluster, den Sie in der Amazon MSK-Konsole finden können. Folgen Sie den Verfahren unter Cluster auflisten in der Amazon MSK-Dokumentation, um diese Informationen zu finden.
- Die mit Ihrem Amazon MSK-Cluster verbundenen Sicherheitsgruppen. Sie finden diese Informationen in der Amazon-MSK-Konsole. Bezeichnen Sie diese in den folgenden Schritten als Ihre*clusterSecurityGroups*.
- Die ID der Amazon VPC, die Ihren Amazon MSK-Cluster enthält. Sie können diese Informationen finden, indem Sie in der Amazon MSK-Konsole die mit Ihrem Amazon MSK-Cluster verbundenen

Subnetze identifizieren und dann in der Amazon VPC-Konsole die mit dem Subnetz verbundene Amazon VPC identifizieren.

- Der Name des in Ihrer Lösung verwendeten Kafka-Themas. Sie können diese Informationen finden, indem Sie Ihren Amazon MSK-Cluster mit der Kafka `topics` CLI von Ihrem Kafka-Administrationshost aus aufrufen. Weitere Informationen zur Themen-CLI finden Sie in der Kafka-Dokumentation unter Themen hinzufügen und entfernen.
- Der Name einer Verbrauchergruppe für Ihr Kafka-Thema, geeignet für die Verwendung durch Ihre Lambda-Funktion. Diese Gruppe kann automatisch von Lambda erstellt werden, Sie müssen sie also nicht mit der Kafka-CLI erstellen. Wenn Sie Ihre Verbrauchergruppen verwalten müssen, finden Sie weitere Informationen über die Verbrauchergruppen-CLI unter Verwalten von Verbrauchergruppen in der Kafka-Dokumentation.

Die folgenden Berechtigungen in Ihrem AWS Konto:

- Berechtigung zur Erstellung und Verwaltung einer Lambda-Funktion.
- Erlaubnis, IAM-Richtlinien zu erstellen und sie mit Ihrer Lambda-Funktion zu verknüpfen.
- Berechtigung zum Erstellen von Amazon VPC-Endpunkten und zum Ändern der Netzwerkkonfiguration in der Amazon VPC, die Ihren Amazon MSK-Cluster hostet.

Installiere das AWS Command Line Interface

Wenn Sie das noch nicht installiert haben AWS Command Line Interface, folgen Sie den Schritten unter Installieren oder Aktualisieren der neuesten Version von AWS CLI, um es zu installieren.

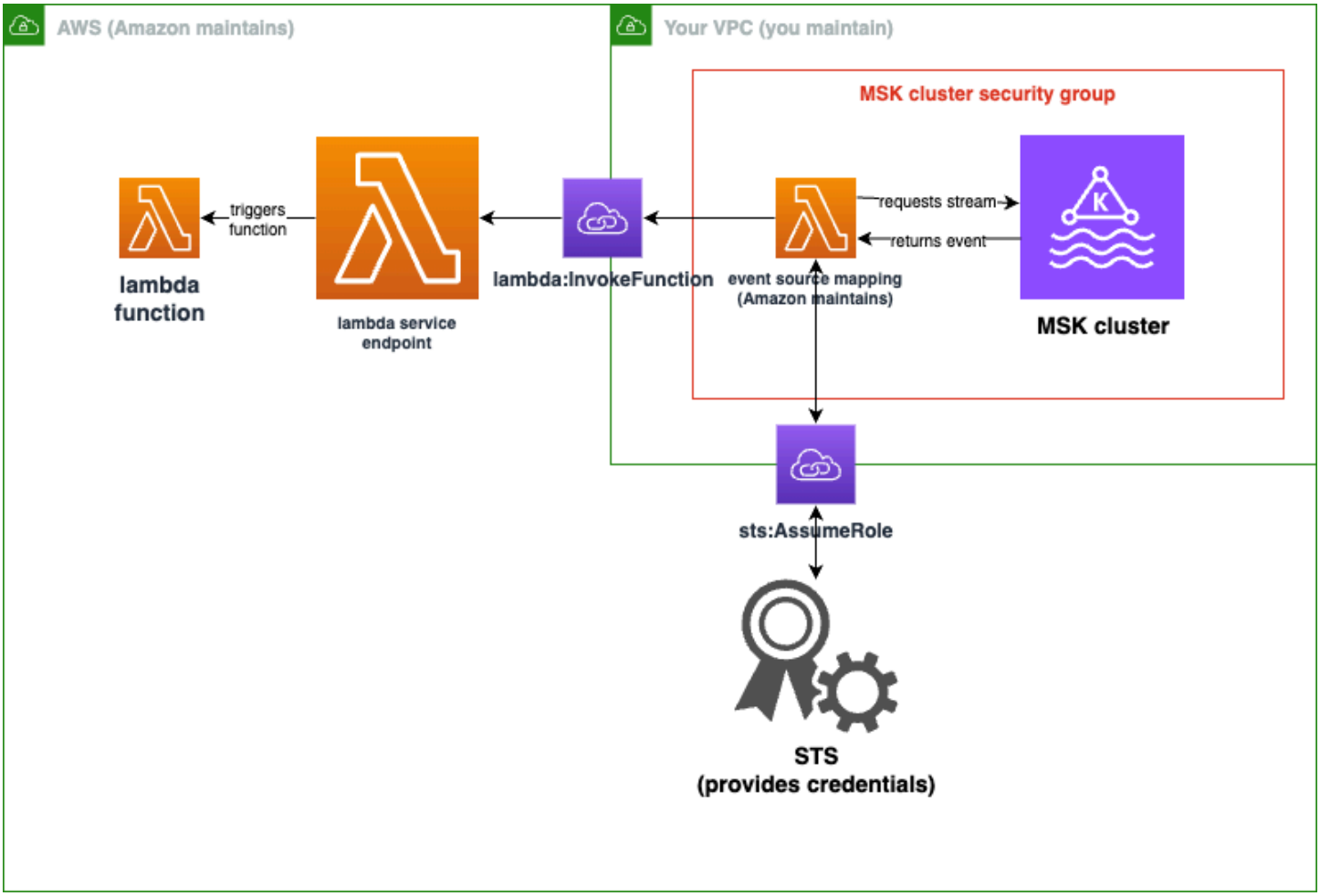
Das Tutorial erfordert zum Ausführen von Befehlen ein Befehlszeilenterminal oder eine Shell. Verwenden Sie unter Linux und macOS Ihre bevorzugte Shell und Ihren bevorzugten Paketmanager.

> **Note**
>
> In Windows werden einige Bash-CLI-Befehle, die Sie häufig mit Lambda verwenden (z. B. `zip`), von den integrierten Terminals des Betriebssystems nicht unterstützt. Um eine in Windows integrierte Version von Ubuntu und Bash zu erhalten, installieren Sie das Windows-Subsystem für Linux.

Konfigurieren Sie die Netzwerkkonnektivität für Lambda zur Kommunikation mit Amazon MSK

Wird verwendet AWS PrivateLink , um Lambda und Amazon MSK zu verbinden. Sie können dies tun, indem Sie Amazon VPC-Endpunkte in der Amazon VPC-Konsole erstellen. Weitere Informationen zur Netzwerkkonfiguration finden Sie unter the section called “Einrichtung von Cluster und Netzwerk”.

Wenn eine Amazon MSK-Zuordnung von Ereignisquellen im Namen einer Lambda-Funktion ausgeführt wird, übernimmt es die Ausführungsrolle der Lambda-Funktion. Diese IAM-Rolle autorisiert die Zuordnung für den Zugriff auf durch IAM gesicherte Ressourcen, wie z. B. Ihren Amazon MSK-Cluster. Obwohl die Komponenten eine gemeinsame Ausführungsrolle haben, haben das Amazon MSK-Mapping und Ihre Lambda-Funktion separate Konnektivitätsanforderungen für ihre jeweiligen Aufgaben, wie im folgenden Diagramm dargestellt.



Ihre Zuordnung von Ereignisquellen gehört zu Ihrer Amazon MSK-Cluster-Sicherheitsgruppe. In diesem Vernetzungsschritt erstellen Sie Amazon VPC-Endpunkte von Ihrer Amazon MSK-Cluster-VPC, um die Zuordnung von Ereignisquellen mit den Lambda- und STS-Services zu verbinden. Sichern Sie diese Endpunkte, damit sie Datenverkehr von Ihrer Amazon MSK-Cluster-

Sicherheitsgruppe akzeptieren. Passen Sie dann die Sicherheitsgruppen des Amazon MSK-Clusters an, damit die Zuordnung von Ereignisquellen mit dem Amazon MSK-Cluster kommunizieren kann.

Sie können die folgenden Schritte mit dem AWS-Managementkonsole.

So konfigurieren Sie Schnittstellen-Amazon-VPC-Endpunkte, um Lambda und Amazon MSK zu verbinden

1. Erstellen Sie eine Sicherheitsgruppe für Ihre Schnittstelle Amazon VPC-Endpunkte,*endpointSecurityGroup*, die eingehenden TCP-Verkehr auf 443 von ermöglicht. *clusterSecurityGroups* Folgen Sie den Anweisungen unter Sicherheitsgruppe erstellen in der EC2 Amazon-Dokumentation, um eine Sicherheitsgruppe zu erstellen. Folgen Sie dann dem Verfahren unter Regeln zu einer Sicherheitsgruppe hinzufügen in der EC2 Amazon-Dokumentation, um die entsprechenden Regeln hinzuzufügen.

   Erstellen Sie eine Sicherheitsgruppe mit den folgenden Informationen:

   Wenn Sie Ihre Regeln für eingehenden Datenverkehr hinzufügen, erstellen Sie eine Regel für jede Sicherheitsgruppe in*clusterSecurityGroups*. Für jede Regel:

   - Wählen Sie für Type (Typ) HTTPS aus.
   - Wählen Sie für Quelle eine von. *clusterSecurityGroups*
2. Erstellen Sie einen Endpunkt, der den Lambda-Service mit der Amazon VPC verbindet, die Ihren Amazon MSK-Cluster enthält. Folgen Sie dem Verfahren unter Schnittstellenendpunkt erstellen.

   Erstellen Sie einen Schnittstellenendpunkt mit den folgenden Informationen:

   - Wählen Sie als Dienstname aus`com.amazonaws.`*regionName*`.lambda`, wo Ihre Lambda-Funktion *regionName* gehostet wird.
   - Wählen Sie für VPC die Amazon VPC aus, die Ihren Amazon MSK-Cluster enthält.
   - Wählen Sie für Sicherheitsgruppen die Gruppen aus*endpointSecurityGroup*, die Sie zuvor erstellt haben.
   - Wählen Sie für Subnetze die Subnetze aus, die Ihren Amazon MSK-Cluster hosten.
   - Geben Sie für die Richtlinie das folgende Richtliniendokument an, das den Endpunkt für die Verwendung durch den Lambda-Serviceprinzipal für die `lambda:InvokeFunction`-Aktion sichert.

     ```
     {
     ```

```
    "Statement": [
        {
            "Action": "lambda:InvokeFunction",
            "Effect": "Allow",
            "Principal": {
                "Service": [
                    "lambda.amazonaws.com"
                ]
            },
            "Resource": "*"
        }
    ]
}
```

- Stellen Sie sicher, dass DNS-Name aktivieren weiterhin aktiviert ist.

3. Erstellen Sie einen Endpunkt, der den AWS STS Service mit der Amazon VPC verbindet, die Ihren Amazon MSK-Cluster enthält. Folgen Sie dem Verfahren unter Schnittstellenendpunkt erstellen.

   Erstellen Sie einen Schnittstellenendpunkt mit den folgenden Informationen:

   - Wählen Sie als Dienstname die Option aus. AWS STS
   - Wählen Sie für VPC die Amazon VPC aus, die Ihren Amazon MSK-Cluster enthält.
   - Wählen Sie für Sicherheitsgruppen die Option aus*endpointSecurityGroup*.
   - Wählen Sie für Subnetze die Subnetze aus, die Ihren Amazon MSK-Cluster hosten.
   - Geben Sie für die Richtlinie das folgende Richtliniendokument an, das den Endpunkt für die Verwendung durch den Lambda-Serviceprinzipal für die `sts:AssumeRole`-Aktion sichert.

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

```
}
```

- Stellen Sie sicher, dass DNS-Name aktivieren weiterhin aktiviert ist.

4. Lassen Sie für jede Sicherheitsgruppe, die Ihrem Amazon MSK-Cluster zugeordnet ist, d. h. in*clusterSecurityGroups*, Folgendes zu:

   - Erlauben Sie allen eingehenden und ausgehenden TCP-Verkehr auf 9098 für alle*clusterSecurityGroups*, auch für sich selbst.
   - Den gesamten ausgehenden TCP-Verkehr auf 443 zulassen.

   Ein Teil dieses Datenverkehrs wird durch die Standardregeln der Sicherheitsgruppe zugelassen. Wenn Ihr Cluster also nur einer einzigen Sicherheitsgruppe angehört und diese Gruppe über Standardregeln verfügt, sind zusätzliche Regeln nicht erforderlich. Um die Regeln für Sicherheitsgruppen anzupassen, folgen Sie den Anweisungen unter Regeln zu einer Sicherheitsgruppe hinzufügen in der EC2 Amazon-Dokumentation.

   Fügen Sie Ihren Sicherheitsgruppen Regeln mit den folgenden Informationen hinzu:

   - Geben Sie für jede eingehende oder ausgehende Regel für Port 9098 Folgendes an
     - Wählen Sie für Type (Typ) Custom TCP (Benutzerdefiniertes TCP).
     - Geben Sie als Portbereich 9098 an.
     - Geben Sie als Quelle eine von an*clusterSecurityGroups*.
   - Wählen Sie für jede eingehende Regel für Port 443 als Typ die Option HTTPS aus.

Erstellen Sie eine IAM-Rolle, die Lambda aus Ihrem Amazon-MSK-Thema lesen kann

Identifizieren Sie die Authentifizierungsanforderungen für Lambda, um aus Ihrem Amazon MSK-Thema zu lesen und definieren Sie sie dann in einer Richtlinie. Erstellen Sie eine Rolle*lambdaAuthRole*, die Lambda autorisiert, diese Berechtigungen zu verwenden. Autorisieren Sie Aktionen in Ihrem Amazon MSK-Cluster mithilfe von `kafka-cluster` IAM-Aktionen. Autorisieren Sie Lambda anschließend, Amazon MSK `kafka` - und EC2 Amazon-Aktionen durchzuführen, die zur Erkennung Ihres Amazon MSK-Clusters und zur Verbindung mit ihm erforderlich sind, sowie CloudWatch Aktionen, damit Lambda protokollieren kann, was es getan hat.

So beschreiben Sie die Authentifizierungsanforderungen für Lambda, um von Amazon MSK zu lesen

1. Schreiben Sie ein IAM-Richtliniendokument (ein JSON-Dokument), das es Lambda ermöglicht*clusterAuthPolicy*, mithilfe Ihrer Kafka-Verbrauchergruppe aus Ihrem Kafka-Thema in Ihrem Amazon MSK-Cluster zu lesen. Für Lambda muss beim Lesen eine Kafka-Verbrauchergruppe festgelegt werden.

   Ändern Sie die folgende Vorlage, um sie an Ihre Voraussetzungen anzupassen:

   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"
            ]
        }
    ]
}
```

   Weitere Informationen erhalten Sie von the section called “Konfigurieren von -Berechtigungen”. Wenn Sie Ihre Richtlinie schreiben:

- Ersetzen Sie *us-east-1* und *111122223333* durch das AWS-Region und AWS-Konto Ihres Amazon MSK-Clusters.
- Geben Sie für *mskClusterName* den Namen Ihres Amazon MSK-Clusters an.
- Geben Sie für *cluster-uuid* die UUID im ARN für Ihren Amazon MSK-Cluster an.
- Geben Sie für *mskTopicName* den Namen Ihres Kafka-Themas an.
- Geben Sie für *mskGroupName* den Namen Ihrer Kafka-Verbrauchergruppe an.

2. Identifizieren Sie Amazon MSK, Amazon EC2 und die CloudWatch Berechtigungen, die Lambda benötigt, um Ihren Amazon MSK-Cluster zu erkennen und zu verbinden, und protokollieren Sie diese Ereignisse.

   Die `AWSLambdaMSKExecutionRole`-verwaltete Richtlinie definiert die erforderlichen Berechtigungen auf unzulässige Weise. Verwenden Sie es in den folgenden Schritten.

   Beurteilen Sie in einer Produktionsumgebung `AWSLambdaMSKExecutionRole`, um Ihre Ausführungsrollenrichtlinie auf der Grundlage des Prinzips der geringsten Berechtigung einzuschränken und schreiben Sie dann eine Richtlinie für Ihre Rolle, die diese verwaltete Richtlinie ersetzt.

Weitere Informationen zur IAM-Richtliniensprache finden Sie unter IAM-Dokumentation.

Nachdem Sie nun Ihr Richtliniendokument verfasst haben, erstellen Sie eine IAM-Richtlinie, die Sie mit Ihrer Rolle verknüpfen können. Sie können dies über die Konsole mit dem folgenden Verfahren tun.

So erstellen Sie eine IAM-Richtlinie aus Ihrem Richtliniendokument

1. Melden Sie sich bei der an AWS-Managementkonsole und öffnen Sie die IAM-Konsole unter. https://console.aws.amazon.com/iam/
2. Wählen Sie im Navigationsbereich auf der linken Seite Policies (Richtlinien).
3. Wählen Sie Richtlinie erstellen aus.
4. Wählen Sie im Bereich Policy editor (Richtlinien-Editor) die Option JSON aus.
5. Einfügen*clusterAuthPolicy*.
6. Wenn Sie mit dem Hinzufügen von Berechtigungen zur Richtlinie fertig sind, wählen Sie Next (Weiter) aus.

7. Geben Sie auf der Seite Review and create (Überprüfen und erstellen) unter Name einen Namen und unter Description (Beschreibung) (optional) eine Beschreibung für die Richtlinie ein, die Sie erstellen. Überprüfen Sie Permissions defined in this policy (In dieser Richtlinie definierte Berechtigungen), um die Berechtigungen einzusehen, die von Ihrer Richtlinie gewährt werden.
8. Wählen Sie Create policy (Richtlinie erstellen) aus, um Ihre neue Richtlinie zu speichern.

Weitere Informationen finden Sie unter Erstellen von IAM-Richtlinien in der IAM-Dokumentation.

Nachdem Sie nun über die entsprechenden IAM-Richtlinien verfügen, erstellen Sie eine Rolle und ordnen Sie sie dieser zu. Sie können dies über die Konsole mit dem folgenden Verfahren tun.

So erstellen Sie eine Ausführungsrolle in der IAM-Konsole

1. Öffnen Sie die Seite Roles (Rollen) in der IAM-Konsole.
2. Wählen Sie Rolle erstellen aus.
3. Wählen Sie unter Typ der vertrauenswürdigen Entität die Option AWS -Service aus.
4. Wählen Sie unter Anwendungsfall Lambda aus.
5. Wählen Sie Weiter aus.
6. Wählen Sie die folgenden Richtlinien:
   - *clusterAuthPolicy*
   - AWSLambdaMSKExecutionRole
7. Wählen Sie Weiter aus.
8. Geben Sie als Rollenname den Text Rolle erstellen ein ***lambdaAuthRole*** und wählen Sie dann Create role aus.

Weitere Informationen finden Sie unter the section called “Ausführungsrolle (Berechtigungen für Funktionen zum Zugriff auf andere Ressourcen)”.

Erstellen Sie eine Lambda-Funktion zum Lesen aus Ihrem Amazon MSK-Thema

Erstellen Sie eine Lambda-Funktion, die für die Verwendung Ihrer IAM-Rolle konfiguriert ist. Sie können Ihre Lambda-Funktion über die Konsole erstellen.

So erstellen Sie eine Lambda-Funktion mit Ihrer Authentifizierungskonfiguration

1. Öffnen Sie die Lambda-Konsole und wählen Sie im Header die Option Funktion erstellen aus.

2. Wählen Sie Verfassen von Grund auf aus.
3. Geben Sie als Funktionsnamen einen geeigneten Namen Ihrer Wahl an.
4. Wählen Sie für Laufzeit die neueste unterstützte Version von `Node.js`, um den in diesem Tutorial bereitgestellten Code zu verwenden.
5. Wählen Sie Standardeausführungsrolle ändern aus.
6. Wählen Sie Eine vorhandene Rolle verwenden aus.
7. Wählen Sie für Existierende Rolle die Option aus *`lambdaAuthRole`*.

In einer Produktionsumgebung müssen Sie der Ausführungsrolle für Ihre Lambda-Funktion normalerweise weitere Richtlinien hinzufügen, um Ihre Amazon MSK-Ereignisse sinnvoll verarbeiten zu können. Weitere Informationen zum Hinzufügen von Richtlinien zu Ihrer Rolle finden Sie unter Hinzufügen oder Entfernen von Identitätsberechtigungen in der IAM-Dokumentation.

Erstellen einer Zuordnung der Ereignisquelle zu Ihrer Lambda-Funktion

Ihre Amazon MSK-Zuordnung von Ereignisquellen liefert dem Lambda-Service die notwendigen Informationen, um Ihr Lambda aufzurufen, wenn entsprechende Amazon MSK-Ereignisse auftreten. Sie können über die Konsole eine Amazon MSK-Zuordnung erstellen. Erstellen Sie einen Lambda-Trigger, dann wird die Zuordnung von Ereignisquellen automatisch eingerichtet.

So erstellen Sie einen Lambda-Trigger (und eine Zuordnung von Ereignisquellen)

1. Navigieren Sie zur Übersichtsseite Ihrer Lambda-Funktion.
2. Wählen Sie im Abschnitt Funktionsübersicht unten links Auslöser hinzufügen.
3. Wählen Sie in der Dropdownliste Quelle auswählen die Option Amazon MSK aus.
4. Legen Sie keine Authentifizierung fest.
5. Wählen Sie für MSK-Cluster den Namen Ihres Clusters aus.
6. Geben Sie für Batch size (Stapelgröße) 1 ein. Dieser Schritt erleichtert das Testen dieses Features, ist aber für die Produktion nicht ideal.
7. Geben Sie für Themenname den Namen Ihres Kafka-Themas ein.
8. Geben Sie unter Verbrauchergruppen-ID die ID Ihrer Kafka-Verbrauchergruppe an.

Aktualisieren Sie Ihre Lambda-Funktion, um Ihre Streaming-Daten zu lesen

Lambda liefert Informationen über Kafka-Ereignisse über den Parameter der Ereignismethode. Ein Beispiel für die Struktur eines Amazon MSK-Ereignisses finden Sie unter the section called “ Beispielereignis”. Nachdem Sie verstanden haben, wie die von Lambda weitergeleiteten Amazon MSK-Ereignisse zu interpretieren sind, können Sie Ihren Lambda-Funktionscode so ändern, dass er die von ihnen bereitgestellten Informationen nutzt.

Fügen Sie den folgenden Code in Ihre Lambda-Funktion ein, um den Inhalt eines Lambda Amazon MSK-Ereignisses zu Testzwecken zu protokollieren:

.NET

SDK für .NET

> **Note**
>
> Es gibt noch mehr dazu GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines Amazon MSK-Ereignisses mit Lambda unter Verwendung von .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

SDK für Go V2

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines Amazon MSK-Ereignisses mit Lambda unter Verwendung von 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 für Java 2.x

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines Amazon MSK-Ereignisses mit Lambda unter Verwendung von 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 für JavaScript (v3)

> **Note**
>
> Es gibt noch mehr dazu GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Ein Amazon MSK-Ereignis mit Lambda verwenden. JavaScript

```
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)
      })
    }
}
```

Ein Amazon MSK-Ereignis mit Lambda verwenden. TypeScript

```
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

SDK für PHP

> **Note**
>
> Es gibt noch mehr dazu. GitHub Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines Amazon MSK-Ereignisses mit Lambda unter Verwendung von 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

SDK für Python (Boto3)

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines Amazon MSK-Ereignisses mit Lambda unter Verwendung von 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

SDK für Ruby

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines Amazon MSK-Ereignisses mit Lambda unter Verwendung von 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 für Rust

> **Note**
>
> Es gibt noch mehr dazu GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines Amazon-MSK-Ereignisses mit Lambda unter Verwendung von Rust.

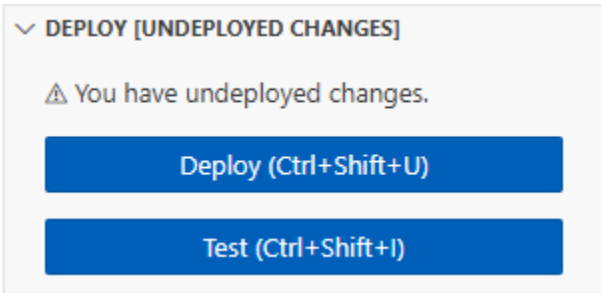
```
use aws_lambda_events::event::kafka::KafkaEvent;
use lambda_runtime::{run, service_fn, tracing, Error, LambdaEvent};
use base64::prelude::*;
use serde_json::{Value};
use tracing::{info};

/// 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
}
```

Sie können Ihrem Lambda Funktionscode über die Konsole zur Verfügung stellen.

So aktualiseren Sie Ihren Funktionscode mit dem Code-Editor

1. Öffnen Sie die Funktionsseite der Lambda-Konsole und wählen Sie Ihre Funktion aus.
2. Wählen Sie die Registerkarte Code aus.
3. Wählen Sie im Bereich Codequelle Ihre Quellcodedatei aus und bearbeiten Sie sie im integrierten Code-Editor.
4. Wählen Sie im Abschnitt BEREITSTELLEN die Option Bereitstellen aus, um den Code Ihrer Funktion zu aktualisieren:



Testen Sie Ihre Lambda-Funktion, um zu überprüfen, ob sie mit Ihrem Amazon MSK-Thema verbunden ist

Sie können jetzt überprüfen, ob Ihr Lambda von der Ereignisquelle aufgerufen wird, indem Sie die Ereignisprotokolle überprüfen CloudWatch .

So überprüfen Sie, ob Ihre Lambda-Funktion aufgerufen wird

1. Verwenden Sie Ihren Kafka-Admin-Host, um Kafka-Ereignisse mithilfe der `kafka-console-producer`-CLI zu generieren. Weitere Informationen finden Sie in der Kafka-Dokumentation unter Einige Ereignisse in das Thema schreiben. Senden Sie so viele Ereignisse, dass der durch die Stapelgröße definierte Stapel für die im vorherigen Schritt definierte Zuordnung von Ereignisquellen gefüllt wird, oder Lambda wartet auf weitere Informationen, um den Aufruf zu starten.
2. Wenn Ihre Funktion ausgeführt wird, schreibt Lambda, was passiert ist. CloudWatch Navigieren Sie in der Konsole zur Detailseite Ihrer Lambda-Funktion.
3. Wählen Sie die Registerkarte Configuration aus.
4. Wählen Sie in der Seitenleiste die Option Überwachungs- und Betriebstools.
5. Identifizieren Sie die CloudWatch Protokollgruppe unter Logging-Konfiguration. Die Protokollgruppe sollte mit `/aws/lambda` beginnen. Wählen Sie den Link zur Protokollgruppe.
6. Suchen Sie in der CloudWatch Konsole in den Protokollereignissen nach den Protokollereignissen, die Lambda an den Protokollstream gesendet hat. Stellen Sie fest, ob es Protokollereignisse gibt, die die Nachricht aus Ihrem Kafka-Ereignis enthalten, wie in der folgenden Abbildung dargestellt. Falls ja, haben Sie erfolgreich eine Lambda-Funktion mit einer Lambda-Zuordnung von Ereignisquellen mit Amazon MSK verbunden.

Verwendung von Lambda mit selbstverwaltetem Apache Kafka

In diesem Thema wird die Verwendung von Lambda mit einem selbstverwalteten Kafka-Cluster beschrieben. In der AWS Terminologie umfasst ein selbstverwalteter Cluster auch nicht AWS gehostete Kafka-Cluster. Sie können Ihren Kafka-Cluster beispielsweise bei einem Cloud-Anbieter wie Confluent Cloud oder Redpanda hosten.

In diesem Kapitel wird erklärt, wie Sie einen selbstverwalteten Apache-Kafka-Cluster als Ereignisquelle für Ihre Lambda-Funktion verwenden. Der allgemeine Prozess zur Integration von selbstverwaltetem Apache Kafka mit Lambda umfasst die folgenden Schritte:

1. Cluster- und Netzwerkeinrichtung – Richten Sie zunächst Ihren selbstverwalteten Apache-Kafka-Cluster mit der richtigen Netzwerkkonfiguration ein, damit Lambda auf Ihren Cluster zugreifen kann.
2. Einrichtung der Zuordnung von Ereignisquellen – Erstellen Sie anschließend die Ressource der Zuordnung von Ereignisquellen, die Lambda benötigt, um Ihren Apache-Kafka-Cluster sicher mit Ihrer Funktion zu verbinden.
3. Einrichtung von Funktionen und Berechtigungen – Stellen Sie abschließend sicher, dass Ihre Funktion korrekt eingerichtet ist und über die erforderlichen Berechtigungen in ihrer Ausführungsrolle verfügt.

Apache Kafka als Ereignisquelle funktioniert ähnlich wie die Verwendung von Amazon Simple Queue Service (Amazon SQS) oder Amazon Kinesis. Lambda fragt intern neue Nachrichten von der Ereignisquelle ab und ruft dann synchron die Ziel-Lambda-Funktion auf. Lambda liest die Nachrichten in Batches und stellt diese Ihrer Funktion als Ereignisnutzlast zur Verfügung. Die maximale Batchgröße ist konfigurierbar (Standardeinstellung: 100 Nachrichten). Weitere Informationen finden Sie unter the section called “Batching-Verhalten”.

Um den Durchsatz Ihrer selbstverwalteten Apache Kafka-Zuordnungen von Ereignisquellen zu optimieren, konfigurieren Sie den Bereitstellungsmodus. Im Bereitstellungsmodus können Sie die minimale und maximale Anzahl von Event-Pollern definieren, die Zuordnung von Ereignisquellen zugewiesen werden. Dies kann die Fähigkeit Ihrer Zuordnung von Ereignisquellen verbessern, unerwartete Nachrichtenspitzen zu bewältigen. Weitere Informationen finden Sie unter Bereitstellungsmodus.

> **Warning**
>
> Zuordnung von Lambda-Ereignisquellen verarbeiten jedes Ereignis mindestens einmal und es kann zu einer doppelten Verarbeitung von Datensätzen kommen. Um mögliche Probleme im Zusammenhang mit doppelten Ereignissen zu vermeiden, empfehlen wir Ihnen dringend, Ihren Funktionscode idempotent zu machen. Weitere Informationen finden Sie im Knowledge Center unter Wie mache ich meine Lambda-Funktion idempotent?. AWS

Für Kafka-basierte Ereignisquellen unterstützt Lambda die Verarbeitung von Steuerungsparametern wie Batch-Fenster und -Größe. Weitere Informationen finden Sie unter Batching-Verhalten.

Ein Beispiel für die Verwendung von selbstverwaltetem Kafka als Ereignisquelle finden Sie im Compute-Blog unter Verwenden von selbst gehostetem Apache Kafka als Ereignisquelle für. AWS Lambda AWS

Themen

- Beispielereignis
- Konfiguration Ihres selbstverwalteten Apache-Kafka-Clusters und -Netzwerks für Lambda
- Konfigurieren von Rollenberechtigungen für Lambda-Ausführungen
- Konfigurieren von selbstverwalteten Apache Kafka-Ereignisquellen für Lambda

Beispielereignis

Lambda sendet den Batch von Nachrichten im Ereignisparameter, wenn es Ihre Lambda-Funktion aufruft. Die Ereignisquelle enthält ein Array von Nachrichten. Jedes Array-Element enthält Details zum Kafka-Thema und Kafka-Partitions-ID, zusammen mit einem Zeitstempel und einer base64-codierten Nachricht.

```
{
   "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
                  ]
               }
            ]
         }
      ]
   }
}
```

Konfiguration Ihres selbstverwalteten Apache-Kafka-Clusters und -Netzwerks für Lambda

Um Ihre Lambda-Funktion mit Ihrem selbstverwalteten Apache-Kafka-Cluster zu verbinden, müssen Sie Ihren Cluster und das Netzwerk, in dem er sich befindet, korrekt konfigurieren. Auf dieser Seite wird beschrieben, wie Sie Ihren Cluster und Ihr Netzwerk konfigurieren. Wenn Ihr Cluster und Ihr Netzwerk bereits ordnungsgemäß konfiguriert sind, lesen Sie unter the section called “Konfigurieren der Ereignisquelle” nach, wie Sie die Zuordnung von Ereignisquellen konfigurieren.

Themen

- Einrichten eines selbstverwalteten Apache Kafka-Clusters
- Konfigurieren der Netzwerksicherheit

Einrichten eines selbstverwalteten Apache Kafka-Clusters

Sie können Ihren selbstverwalteten Apache-Kafka-Cluster bei Cloud-Anbietern wie Confluent Cloud oder Redpanda hosten oder ihn in Ihrer eigenen Infrastruktur ausführen. Stellen Sie sicher, dass Ihr Cluster ordnungsgemäß konfiguriert ist und über das Netzwerk zugänglich ist, mit dem Ihre Lambda-Zuordnung von Ereignisquellen verbunden wird.

Konfigurieren der Netzwerksicherheit

Um Lambda vollen Zugriff auf selbstverwaltetes Apache Kafka über Ihre Zuordnung von Ereignisquellen zu ermöglichen, muss Ihr Cluster entweder einen öffentlichen Endpunkt (öffentliche IP-Adresse) verwenden oder Sie müssen Zugriff auf die Amazon VPC gewähren, in der Sie den Cluster erstellt haben.

Wenn Sie selbstverwaltetes Apache Kafka mit Lambda verwenden, erstellen Sie AWS PrivateLink VPC-Endpunkte, die Ihrer Funktion Zugriff auf die Ressourcen in Ihrer Amazon VPC bieten.

> **Note**
>
> AWS PrivateLink VPC-Endpunkte sind für Funktionen mit Zuordnungen von Ereignisquellen erforderlich, die den Standardmodus (auf Abruf) für Ereignisabfragen verwenden. Wenn Ihre Zuordnung von Ereignisquellen den bereitgestellten Modus verwendet, müssen Sie keine AWS PrivateLink-VPC-Endpunkte konfigurieren.

Erstellen Sie einen Endpunkt, der den Zugriff auf die folgenden Ressourcen ermöglicht:

- Lambda – Erstellen Sie einen Endpunkt für den Lambda-Serviceprinzipal.
- AWS STS – Erstellen Sie einen Endpunkt für den AWS STS, damit ein Serviceprinzipal eine Rolle in Ihrem Namen übernehmen kann.
- Secrets Manager – Wenn Ihr Cluster Secrets Manager zum Speichern von Anmeldeinformationen verwendet, erstellen Sie einen Endpunkt für Secrets Manager.

Alternativ konfigurieren Sie ein NAT-Gateway auf jedem öffentlichen Subnetz in der Amazon-VPC. Weitere Informationen finden Sie unter the section called “Internetzugriff für VPC-Funktionen”.

Wenn Sie eine Zuordnung von Ereignisquellen für selbstverwaltetes Apache Kafka erstellen, prüft Lambda, ob Elastic-Network-Schnittstelle (ENI) für die für Ihre Amazon VPC konfigurierten Subnetze und Sicherheitsgruppen bereits vorhanden sind. Findet Lambda vorhandene ENIs, versucht es,

diese wiederzuverwenden. Andernfalls erstellt Lambda neue ENIs, um eine Verbindung mit der Ereignisquelle herzustellen und Ihre Funktion aufzurufen.

> **Note**
>
> Lambda-Funktionen werden immer in einer VPCs ausgeführt, die dem Lambda-Service gehört. Die VPC-Konfiguration Ihrer Funktion hat keinen Einfluss auf die Zuordnung von Ereignisquellen. Nur die Netzwerkkonfiguration der Ereignisquelle bestimmt, wie Lambda sich mit Ihrer Ereignisquelle verbindet.

Konfigurieren Sie die Sicherheitsgruppen für die Amazon VPC, die Ihren Cluster enthält. Standardmäßig verwendet der selbstverwaltete Apache Kafka die folgenden Ports: `9092`.

- Eingehende Regeln – Erlauben Sie den gesamten Datenverkehr auf dem Standard-Broker-Port für die Sicherheitsgruppe, die mit Ihrer Ereignisquelle verbunden ist. Alternativ können Sie eine selbstreferenzierende Regel für Sicherheitsgruppen verwenden, um den Zugriff von Instances innerhalb derselben Sicherheitsgruppe zuzulassen.
- Regeln für ausgehenden Datenverkehr – Erlauben Sie den gesamten Datenverkehr an Port `443` für externe Ziele, wenn Ihre Funktion mit AWS-Services kommunizieren muss. Alternativ können Sie auch eine selbstreferenzierende Regel für Sicherheitsgruppen verwenden, um den Zugriff auf den Broker einzuschränken, wenn Sie nicht mit anderen AWS-Services kommunizieren müssen.
- Regeln für eingehenden Datenverkehr auf Amazon-VPC-Endpunkten – Wenn Sie einen Amazon-VPC-Endpunkt verwenden, muss die Sicherheitsgruppe, die mit Ihrem Amazon-VPC-Endpunkt verbunden ist, eingehenden Verkehr auf Port `443` von der Cluster-Sicherheitsgruppe zulassen.

Wenn Ihr Cluster eine Authentifizierung verwendet, können Sie auch die Endpunktrichtlinie für den Secrets-Manager-Endpunkt einschränken. Für den Aufruf der Secrets-Manager-API verwendet Lambda Ihre Funktionsrolle und nicht den Lambda-Serviceprinzipal.

Example VPC-Endpunktrichtlinie – Secrets Manager-Endpunkt

```
{
      "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"
        }
    ]
}
```

Wenn Sie Amazon VPC-Endpunkte verwenden, leitet AWS Ihre API-Aufrufe weiter, um Ihre Funktion über die Elastic-Network-Schnittstelle (ENI) des Endpunkts aufzurufen. Der Prinzipal des Lambda-Dienstes muss `lambda:InvokeFunction` für alle Rollen und Funktionen aufrufen, die diese ENI verwenden.

Standardmäßig verfügen Amazon VPC-Endpunkte über offene IAM-Richtlinien, die einen umfassenden Zugriff auf Ressourcen ermöglichen. Es empfiehlt sich, diese Richtlinien auf die Durchführung der erforderlichen Aktionen über diesen Endpunkt zu beschränken. Um sicherzustellen, dass Ihre Zuordnung von Ereignisquellen Ihre Lambda-Funktion aufrufen kann, muss die VPC-Endpunktrichtlinie dem Lambda-Serviceprinzipal erlauben, `sts:AssumeRole` und `lambda:InvokeFunction` aufzurufen. Wenn Sie Ihre VPC-Endpunktrichtlinien so einschränken, dass nur API-Aufrufe zugelassen werden, die aus Ihrem Unternehmen stammen, kann die Zuordnung von Ereignisquellen nicht richtig funktionieren, weshalb in diesen Richtlinien `"Resource": "*"` erforderlich ist.

Die folgenden Beispiel-VPC-Endpunktrichtlinien zeigen, wie der erforderliche Zugriff auf den Lambda-Serviceprinzipal für die AWS STS- und Lambda-Endpunkte gewährt wird.

Example VPC-Endpunktrichtlinie – AWS STS-Endpunkt

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

```
            }
        ]
    }
```

Example VPC-Endpunktrichtlinie – Lambda-Endpunkt

```
{
      "Statement": [
          {
              "Action": "lambda:InvokeFunction",
              "Effect": "Allow",
              "Principal": {
                  "Service": [
                      "lambda.amazonaws.com"
                  ]
              },
              "Resource": "*"
          }
      ]
  }
```

Konfigurieren von Rollenberechtigungen für Lambda-Ausführungen

Neben dem Zugriff auf Ihren selbstverwalteten Kafka-Cluster benötigt Ihre Lambda-Funktion auch Berechtigungen, um verschiedene API-Aktionen auszuführen. Sie fügen diese Berechtigungen zur Ausführungsrolle der Funktion hinzu. Wenn Ihre Benutzer Zugriff auf API-Aktionen benötigen, fügen Sie der Identitätsrichtlinie für den AWS Identity and Access Management (IAM) -Benutzer oder die Rolle die erforderlichen Berechtigungen hinzu.

Themen

- Benötigte Lambda-Funktionsberechtigungen
- Optionale Lambda-Funktionsberechtigungen
- Hinzufügen von Berechtigungen zu Ihrer Ausführungsrolle
- Gewähren von Benutzerzugriff mit einer IAM-Richtlinie

Benötigte Lambda-Funktionsberechtigungen

Um Protokolle in einer Protokollgruppe in Amazon CloudWatch Logs zu erstellen und zu speichern, muss Ihre Lambda-Funktion in ihrer Ausführungsrolle über die folgenden Berechtigungen verfügen:

- Protokolle: CreateLogGroup
- Protokolle: CreateLogStream
- Protokolle: PutLogEvents

Optionale Lambda-Funktionsberechtigungen

Ihre Lambda-Funktion benötigt möglicherweise auch Berechtigungen für Folgendes:

- Beschreiben Ihres Secrets-Manager-Secrets
- Greifen Sie auf Ihren AWS Key Management Service (AWS KMS) vom Kunden verwalteten Schlüssel zu.
- Zugriff auf Ihre Amazon VPC
- Senden von Datensätzen zu fehlgeschlagenen Aufrufen an ein Ziel

Secrets Manager und AWS KMS Berechtigungen

Abhängig von der Art der Zugriffskontrolle, die Sie für Ihre Kafka-Broker konfigurieren, benötigt Ihre Lambda-Funktion möglicherweise die Erlaubnis, auf Ihr Secrets Manager-Geheimnis zuzugreifen oder Ihren vom AWS KMS Kunden verwalteten Schlüssel zu entschlüsseln. Um auf diese Ressourcen zuzugreifen, muss die Ausführungsrolle Ihrer Funktion die folgenden Berechtigungen besitzen:

- Secretsmanager: GetSecretValue
- kms:Decrypt

VPC-Berechtigungen

Wenn nur Benutzer innerhalb einer VPC auf Ihren selbstverwalteten Apache-Kafka-Cluster zugreifen können, muss Ihre Lambda-Funktion die Berechtigung zum Zugriff auf Ihre Amazon-VPC-Ressourcen haben. Zu diesen Ressourcen gehören Ihre VPC, Subnetze, Sicherheitsgruppen und Netzwerkschnittstellen. Um auf diese Ressourcen zuzugreifen, muss die Ausführungsrolle Ihrer Funktion die folgenden Berechtigungen besitzen:

- ec2: CreateNetworkInterface
- ec2: DescribeNetworkInterfaces
- ec2: DescribeVpcs

- ec2: DeleteNetworkInterface
- ec2: DescribeSubnets
- ec2: DescribeSecurityGroups

Hinzufügen von Berechtigungen zu Ihrer Ausführungsrolle

Um auf andere AWS Dienste zuzugreifen, die Ihr selbstverwalteter Apache Kafka-Cluster verwendet, verwendet Lambda die Berechtigungsrichtlinien, die Sie in der Ausführungsrolle Ihrer Lambda-Funktion definieren.

Standardmäßig ist es Lambda nicht gestattet, die erforderlichen oder optionalen Aktionen für einen selbstverwalteten Apache-Kafka-Cluster durchzuführen. Sie müssen diese Aktionen in einer IAM-Vertrauensrichtlinie für Ihre Ausführungsrolle erstellen und definieren. In diesem Beispiel wird gezeigt, wie Sie eine Richtlinie erstellen können, die Lambda den Zugriff auf Ihre Amazon-VPC-Ressourcen erlaubt.

JSON

```
{
        "Version":"2012-10-17",
        "Statement":[
           {
              "Effect":"Allow",
              "Action":[
                 "ec2:CreateNetworkInterface",
                 "ec2:DescribeNetworkInterfaces",
                 "ec2:DescribeVpcs",
                 "ec2:DeleteNetworkInterface",
                 "ec2:DescribeSubnets",
                 "ec2:DescribeSecurityGroups"
              ],
              "Resource":"*"
           }
        ]
    }
```

Gewähren von Benutzerzugriff mit einer IAM-Richtlinie

Standardmäßig haben Benutzer und Rollen keine Berechtigung zum Ausführen von Ereignisquellen-API-Vorgängen. Um Benutzern in Ihrem Unternehmen oder Ihrem Konto Zugriff zu gewähren, erstellen oder aktualisieren Sie eine identitätsbasierte Richtlinie. Weitere Informationen finden Sie unter Steuern des Zugriffs auf AWS Ressourcen mithilfe von Richtlinien im IAM-Benutzerhandbuch.

Informationen zur Behebung von Authentifizierungs- und Autorisierungsfehlern finden Sie unter the section called “Fehlerbehebung”.

Konfigurieren von selbstverwalteten Apache Kafka-Ereignisquellen für Lambda

Um einen selbstverwalteten Apache-Kafka-Cluster als Ereignisquelle für Ihre Lambda-Funktion zu verwenden, erstellen Sie eine Zuordnung von Ereignisquellen, die die beiden Ressourcen miteinander verbindet. Auf dieser Seite wird beschrieben, wie Sie eine Zuordnung von Ereignisquellen für selbstverwaltetes Apache Kafka erstellen.

Auf dieser Seite wird davon ausgegangen, dass Sie Ihren Kafka-Cluster und das Netzwerk, in dem er sich befindet, bereits ordnungsgemäß konfiguriert haben. Wenn Sie Ihren Cluster oder Ihr Netzwerk einrichten müssen, finden Sie entsprechende Informationen unter the section called “Einrichtung von Cluster und Netzwerk”.

Themen

- Verwenden eines selbstverwalteten Apache-Kafka-Clusters als Ereignisquelle
- Konfiguration der Cluster-Authentifizierungsmethoden in Lambda
- Erstellen einer Lambda-Zuordnung von Ereignisquellen für eine selbstverwaltete Apache-Kafka-Ereignisquelle
- Alle Konfigurationsparameter für selbstverwaltete Apache-Kafka-Ereignisquellen in Lambda

Verwenden eines selbstverwalteten Apache-Kafka-Clusters als Ereignisquelle

Wenn Sie Ihren Apache Kafka-Cluster oder Amazon MSK als Auslöser für Ihre Lambda-Funktion hinzufügen, wird der Cluster als Ereignisquelle verwendet.

Lambda liest Ereignisdaten aus den Kafka-Themen, die Sie wie `Topics` in einer CreateEventSourceMappingAnfrage angeben, basierend auf der von Ihnen angegebenen Startposition. Nach erfolgreicher Verarbeitung wird Ihr Kafka-Thema Ihrem Kafka-Cluster zugeordnet.

Lambda liest Nachrichten sequentiell für jede Kafka-Themenpartition. Eine einzelne Lambda-Nutzlast kann Nachrichten von mehreren Partitionen enthalten. Wenn mehr Datensätze verfügbar sind, setzt Lambda die Verarbeitung von Datensätzen stapelweise fort, basierend auf dem BatchSize Wert, den Sie in einer CreateEventSourceMappingAnfrage angeben, bis Ihre Funktion das Thema eingeholt hat.

Nachdem Lambda jeden Batch verarbeitet hat, werden die Offsets der Nachrichten in diesem Batch festgeschrieben. Wenn Ihre Funktion einen Fehler für eine der Nachrichten in einem Batch zurückgibt, wiederholt Lambda den gesamten Nachrichtenbatch, bis die Verarbeitung erfolgreich ist oder die Nachrichten ablaufen. Sie können Datensätze, bei denen alle Wiederholungsversuche fehlschlagen, zur späteren Verarbeitung an ein Ausfallziel senden.

> **Note**
>
> Während Lambda-Funktionen in der Regel ein maximales Timeout-Limit von 15 Minuten haben, unterstützen Ereignisquellenzuordnungen für Amazon MSK, selbstverwaltetes Apache Kafka, Amazon DocumentDB, Amazon MQ für ActiveMQ und RabbitMQ nur Funktionen mit einem maximalen Timeout-Limit von 14 Minuten.

Konfiguration der Cluster-Authentifizierungsmethoden in Lambda

Lambda unterstützt mehrere Methoden zur Authentifizierung mit Ihrem selbstverwalteten Apache-Kafka-Cluster. Stellen Sie sicher, dass Sie den Kafka-Cluster für die Verwendung einer der folgenden unterstützten Authentifizierungsmethoden konfigurieren. Weitere Informationen zur Sicherheit von Kafka finden Sie unter Sicherheit in der Kafka-Dokumentation.

SASL/SCRAM-Authentifizierung

Lambda unterstützt die Authentifizierung Simple Authentication und Security Layer/Salted Challenge Response Authentication Mechanism (SASL/SCRAM) mit Transport Layer Security (TLS) -Verschlüsselung (). `SASL_SSL` Lambda sendet die verschlüsselten Anmeldeinformationen zur Authentifizierung mit dem Cluster. Lambda unterstützt SASL/SCRAM mit Klartext () `SASL_PLAINTEXT` nicht. Weitere Informationen zur SASL/SCRAM Authentifizierung finden Sie unter RFC 5802.

Lambda unterstützt auch die SASL/PLAIN Authentifizierung. Da dieser Mechanismus Anmeldeinformationen im Klartext verwendet, muss die Verbindung zum Server TLS-Verschlüsselung verwenden, um sicherzustellen, dass die Anmeldeinformationen geschützt sind.

Für die SASL-Authentifizierung speichern Sie die Anmeldeinformationen als Secret in AWS Secrets Manager. Weitere Informationen zur Verwendung von Secrets Manager finden Sie unter Create an AWS Secrets Manager Secret im AWS Secrets Manager Benutzerhandbuch.

> **Important**
>
> Um Secrets Manager für die Authentifizierung zu verwenden, müssen Secrets in derselben AWS Region wie Ihre Lambda-Funktion gespeichert werden.

Gegenseitige TLS-Authentifizierung

Gegenseitige TLS (mTLS) bietet eine bidirektionale Authentifizierung zwischen Client und Server. Der Client sendet ein Zertifikat an den Server, damit der Server den Client überprüfen kann, und der Server sendet ein Zertifikat an den Client, damit der Client den Server überprüfen kann.

Im selbstverwalteten Apache Kafka fungiert Lambda als Client. Sie konfigurieren ein Client-Zertifikat (als Secret in Secrets Manager), um Lambda für Ihre Kafka-Broker zu authentifizieren. Das Client-Zertifikat muss von einer Zertifizierungsstelle im Trust Store des Servers signiert sein.

Der Kafka-Cluster sendet ein Serverzertifikat an Lambda, um die Kafka-Broker für Lambda zu authentifizieren. Das Serverzertifikat kann ein öffentliches CA-Zertifikat oder ein privat CA/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 signiertes Zertifikat sein. Sie konfigurieren das Root-CA-Zertifikat des Servers (als geheim in Secrets Manager). Lambda verwendet das Root-Zertifikat, um die Kafka-Broker zu verifizieren.

Weitere Informationen über mTLS finden Sie unter Vorstellung von gegenseitiger TLS-Authentifizierung für Amazon MSK als Ereignisquelle.

Konfigurieren des Client-Zertifikat-Secrets

Das Secret CLIENT_CERTIFICATE_TLS_AUTH erfordert ein Zertifikatfeld und ein Feld für einen privaten Schlüssel. Für einen verschlüsselten privaten Schlüssel erfordert das Secret ein Passwort für den privaten Schlüssel. Sowohl das Zertifikat als auch der private Schlüssel müssen im PEM-Format vorliegen.

> **Note**
>
> Lambda unterstützt die Verschlüsselungsalgorithmen mit privaten Schlüsseln PBES1(aber nicht PBES2).

Das Zertifikatfeld muss eine Liste von Zertifikaten enthalten, beginnend mit dem Client-Zertifikat, gefolgt von etwaigen Zwischenzertifikaten und endend mit dem Root-Zertifikat. Jedes Zertifikat muss in einer neuen Zeile mit der folgenden Struktur beginnen:

```
-----BEGIN CERTIFICATE-----
            <certificate contents>
-----END CERTIFICATE-----
```

Secrets Manager unterstützt Secrets von bis zu 65 536 Bytes, was genügend Platz für lange Zertifikatsketten bietet.

Der private Schlüssel muss im Format PKCS #8 mit folgender Struktur vorliegen:

```
-----BEGIN PRIVATE KEY-----
             <private key contents>
-----END PRIVATE KEY-----
```

Verwenden Sie für einen verschlüsselten privaten Schlüssel die folgende Struktur:

```
-----BEGIN ENCRYPTED PRIVATE KEY-----
              <private key contents>
-----END ENCRYPTED PRIVATE KEY-----
```

Im folgenden Beispiel sehen Sie den Inhalt eines Secrets für mTLS-Authentifizierung mit einem verschlüsselten privaten Schlüssel. Fügen Sie für einen verschlüsselten privaten Schlüssel das Passwort für den privaten Schlüssel in das Secret ein.

```
{"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-----"
}
```

Konfigurieren des Secrets des Server-Root-CA-Zertifikats

Sie erstellen dieses Secret, wenn Ihre Kafka-Broker TLS-Verschlüsselung mit Zertifikaten verwenden, die von einer privaten CA signiert wurden. Sie können die TLS-Verschlüsselung für die VPC- SASL/SCRAM, SASL/PLAIN oder mTLS-Authentifizierung verwenden.

Das Secret des Server-Root-CA-Zertifikats erfordert ein Feld, das das Root-CA-Zertifikat des Kafka-Brokers im PEM-Format enthält. Das folgende Beispiel zeigt die Struktur des Secrets.

```
{"certificate":"-----BEGIN CERTIFICATE-----
MIID7zCCAtegAwIBAgIBADANBgkqhkiG9w0BAQsFADCBmDELMAkGA1UEBhMCVVMx
EDAOBgNVBAgTB0FyaXpvbmExEzARBgNVBAcTClNjb3R0c2RhbGUxJTAjBgNVBAoT
HFN0YXJmaWVsZCBUZWNobm9sb2dpZXMsIEluYy4xOzA5BgNVBAMTMlN0YXJmaWVs
ZCBTZXJ2aWNlcyBSb290IENlcnRpZmljYXRlIEF1dG...
-----END CERTIFICATE-----"
}
```

Erstellen einer Lambda-Zuordnung von Ereignisquellen für eine selbstverwaltete Apache-Kafka-Ereignisquelle

Zum Erstellen einer Zuordnung von Ereignisquellen können Sie die Lambda-Konsole, die AWS Command Line Interface (CLI) oder ein AWS SDK verwenden.

Mit den folgenden Schritten in der Konsole fügen Sie einen selbstverwalteten Apache-Kafka-Cluster als Auslöser für Ihre Lambda-Funktion hinzu. Im Hintergrund wird dadurch eine Zuordnung von Ereignisquellen erstellt.

Voraussetzungen

- Selbstverwaltetes Apache-Kafka-Cluster. Lambda unterstützt Apache Kafka Version 0.10.1.0 und höher.
- Eine Ausführungsrolle mit der Berechtigung, auf die AWS Ressourcen zuzugreifen, die Ihr selbstverwalteter Kafka-Cluster verwendet.

Hinzufügen eines selbstverwalteten Kafka-Clusters (Konsole)

Befolgen Sie diese Schritte, um Ihren selbstverwalteten Apache-Kafka-Cluster und ein Kafka-Thema als Auslöser für Ihre Lambda-Funktion hinzuzufügen.

So fügen Sie Ihrer Lambda-Funktion (Konsole) einen Apache-Kafka-Auslöser hinzu

1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.
2. Wählen Sie den Namen Ihrer Lambda-Funktion aus.
3. Wählen Sie unter Function overview (Funktionsübersicht) die Option Add trigger (Trigger hinzufügen).
4. Führen Sie unter Auslöser-Konfiguration die folgenden Schritte aus:

    a. Wählen Sie den Apache-Kafka-Auslösertyp.

    b. Geben Sie für Bootstrap-Server die Host- und Portpaaradresse eines Kafka-Brokers in Ihrem Cluster ein und wählen Sie dann Hinzufügen. Wiederholen Sie den Vorgang für jeden Kafka-Broker im Cluster.

    c. Geben Sie für Themenname den Namen des Kafka-Themas ein, das zum Speichern von Datensätzen im Cluster verwendet wird.

    d. Wenn Sie den Bereitstellungsmodus konfigurieren, geben Sie einen Wert für Minimale Anzahl von Ereignisabfragen, einen Wert für Maximale Anzahl an Ereignisabfragen und einen optionalen Wert ein, PollerGroupName um die Gruppierung mehrerer ESMs innerhalb derselben Ereignisquellen-VPC anzugeben.

    e. (Optional) Geben Sie für Batchgröße die maximale Anzahl von Datensätzen ein, die in einem einzelnen Batch empfangen werden sollen.

    f. Geben Sie für Batch window (Batch-Fenster) die maximale Zeit in Sekunden ein, die Lambda mit dem Sammeln von Datensätzen verbringt, bevor die Funktion aufgerufen wird.

    g. (Optional) Geben Sie für Konsumentengruppen-ID die ID einer Kafka-Konsumentengruppe ein, der Sie beitreten möchten.

h. (Optional) Wählen Sie für Startposition die Option Neueste, um mit dem Lesen des Streams aus dem letzten Datensatz zu beginnen, Horizont trimmen, um mit dem frühesten verfügbaren Datensatz zu beginnen, oder Am Zeitstempel, um einen Zeitstempel anzugeben, ab dem mit dem Lesen begonnen werden soll.

i. (Optional) Wählen Sie bei VPC die Amazon VPC für Ihren Kafka-Cluster aus. Wählen Sie dann VPC subnets (VPC-Subnetze) und VPC security groups (VPC-Sicherheitsgruppen) aus.

   Diese Einstellung ist erforderlich, wenn nur Benutzer innerhalb Ihrer VPC auf Ihre Broker zugreifen.

j. (Optional) Wählen Sie bei Authentication (Authentifizierung) die Option Add (Hinzufügen) aus und gehen Sie dann folgendermaßen vor:

   i. Wählen Sie das Zugriffs- oder Authentifizierungsprotokoll der Kafka-Broker in Ihrem Cluster aus.

      - Wenn Ihr Kafka-Broker Authentifizierung verwendet, wählen Sie BASIC_AUTH. SASL/PLAIN
      - Wenn Ihr Broker SASL/SCRAM Authentifizierung verwendet, wählen Sie eines der Protokolle aus. SASL_SCRAM
      - Falls Sie die mTLS-Authentifizierung konfigurieren, wählen Sie das Protokoll CLIENT_CERTIFICATE_TLS_AUTH aus.

   ii. Wählen Sie für SASL/SCRAM die mTLS-Authentifizierung den geheimen Schlüssel von Secrets Manager, der die Anmeldeinformationen für Ihren Kafka-Cluster enthält.

k. (Optional) Wählen Sie bei Encryption (Verschlüsselung) das Secrets-Manager-Secret aus, das das Root-CA-Zertifikat enthält, das Ihre Kafka-Broker zur TLS-Verschlüsselung verwenden, falls Ihre Kafka-Broker von einer privaten Zertifizierungsstelle signierte Zertifikate verwenden.

   Diese Einstellung gilt für die TLS-Verschlüsselung für SASL/SCRAM oder SASL/PLAIN und für die mTLS-Authentifizierung.

l. Um den Auslöser zu Testzwecken in einem deaktivierten Zustand zu erstellen (empfohlen), deaktivieren Sie Auslöser aktivieren. Um den Auslöser sofort zu aktivieren, wählen Sie Auslöser aktivieren.

5. Wählen Sie hinzufügen aus, um den Auslöser zu erstellen.

Selbstverwaltetes Apache-Kafka-Cluster hinzufügen (AWS CLI)

Verwenden Sie die folgenden AWS CLI Beispielbefehle, um einen selbstverwalteten Apache Kafka-Trigger für Ihre Lambda-Funktion zu erstellen und anzuzeigen.

Verwenden von SASL/SCRAM

Wenn Kafka-Benutzer über das Internet auf Ihre Kafka-Broker zugreifen, geben Sie das Secrets Manager Manager-Geheimnis an, das Sie für SASL/SCRAM die Authentifizierung erstellt haben. Im folgenden Beispiel wird der create-event-source-mapping AWS CLI Befehl verwendet, um eine Lambda-Funktion mit dem Namen einem Kafka-Thema namens `my-kafka-function` zuzuordnen. `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"]}}'
```

Verwenden einer VPC

Wenn nur Kafka-Benutzer in Ihrer VPC auf Ihre Kafka-Broker zugreifen, müssen Sie Ihre VPC, Subnetze und VPC-Sicherheitsgruppe angeben. Im folgenden Beispiel wird der create-event-source-mapping AWS CLI Befehl verwendet, um eine Lambda-Funktion mit dem Namen einem Kafka-Thema namens `my-kafka-function` zuzuordnen. `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"]}}'
```

Den Status mit dem anzeigen AWS CLI

Im folgenden Beispiel wird der get-event-source-mapping AWS CLI Befehl verwendet, um den Status der von Ihnen erstellten Ereignisquellenzuordnung zu beschreiben.

```
aws lambda get-event-source-mapping
              --uuid dh38738e-992b-343a-1077-3478934hjkfd7
```

Alle Konfigurationsparameter für selbstverwaltete Apache-Kafka-Ereignisquellen in Lambda

Alle Lambda-Ereignisquelltypen verwenden dieselben CreateEventSourceMappingUpdateEventSourceMappingAPI-Operationen. Allerdings gelten nur einige der Parameter für selbstverwaltetes Apache Kafka, wie in der folgenden Tabelle dargestellt.

Parameter	Erforderlich	Standard	Hinweise
BatchSize	N	100	Höchstwert: 10 000.
DestinationConfig	N	–	the section called “Fehlgeschlagene Aufrufe beibehalten”
Aktiviert	N	Wahr	
FilterCriteria	N	–	Steuern Sie, welche Ereignisse Lambda an Ihre Funktion sendet
FunctionName	Y	–	
KMSKeyArn	N	–	the section called “Verschlüsselung der Filterkriterien”
MaximumBatchingWindowInSeconds	N	500 ms	Batching-Verhalten
ProvisionedPollersConfig	N	`MinimumPollers` : Standardwert von 1, wenn nicht angegeben `MaximumPollers` : Standardwert von	the section called “Modus bereitgestellter Kapazität”

Parameter	Erforderlich	Standard	Hinweise
		200, wenn nicht angegeben `PollerGroupName` : N/A	
SelfManagedEventSource	Y	–	Liste der Kafka Broker. Kann nur auf „Erstellen“ festgelegt werden
SelfManagedKafkaEventSourceConfig	N	Enthält das ConsumerGroupId Feld, das standardmäßig einen eindeutigen Wert hat.	Kann nur auf „Erstellen“ festgelegt werden
SourceAccessConfigurations	N	Keine Anmeldeinformationen	VPC-Informationen oder Authentifizierungsanmeldeinformationen für den Cluster Für SASL_PLAIN, auf BASIC_AUTH gesetzt
StartingPosition	Y	–	AT_TIMESTAMP, TRIM_HORIZON, oder LATEST Kann nur auf „Erstellen“ festgelegt werden
StartingPositionTimestamp	N	–	Erforderlich, wenn der Wert auf StartingPosition AT_TIMESTAMP gesetzt ist

Parameter	Erforderlich	Standard	Hinweise
Tags (Markierungen)	N	–	the section called “Tags für die Zuordnung von Ereignisquellen”
Topics	Y	–	Themenname Kann nur auf „Erstelle n“ festgelegt werden

> **Note**
>
> Wenn Sie a angeben`PollerGroupName`, können sich mehrere ESMs innerhalb derselben Amazon VPC die Kapazität der Event Poller Unit (EPU) teilen. Sie können diese Option verwenden, um die Kosten für den Bereitstellungsmodus für Sie zu optimieren. ESMs Anforderungen für die ESM-Gruppierung:
>
> - ESMs muss sich in derselben Amazon-VPC befinden
> - Maximal 100 ESMs pro Befragungsgruppe
> - Die Gesamtzahl der Befragten ESMs in einer Gruppe darf 2000 nicht überschreiten
>
> Sie können den aktualisieren`PollerGroupName`, um einen ESM in eine andere Gruppe zu verschieben, oder einen ESM aus einer Gruppe entfernen, indem Sie ihn `PollerGroupName` auf eine leere Zeichenfolge („“) setzen.

Skalierungsmodi des Ereignis-Pollers von Apache Kafka in Lambda

Sie können zwischen zwei Skalierungsmodi von Ereignis-Pollern für Zuordnungen von Ereignisquellen von Amazon MSK und selbstverwaltetem Apache Kafka wählen.

- On-Demand-Modus (Standard)
- Modus bereitgestellter Kapazität

On-Demand-Modus (Standard)

Wenn Sie zuerst eine Kafka-Ereignisquelle erstellen, weist Lambda eine Standardanzahl von Ereignis-Pollern zu, um alle Partitionen im Kafka-Thema zu verarbeiten. Lambda erhöht oder verringert automatisch die Anzahl der Ereignis-Poller je nach Nachrichtenlast.

In Intervallen von einer Minute wertet Lambda den Offset-Lag aller Partitionen des Themas aus. Wenn die Offset-Verzögerung zu hoch ist, empfängt die Partition Nachrichten schneller als Lambda sie verarbeiten kann. Falls erforderlich, fügt Lambda dem Thema Ereignis-Poller hinzu oder entfernt sie. Dieser Prozess der automatischen Skalierung durch Hinzufügen oder Entfernen von Ereignis-Pollern erfolgt innerhalb von drei Minuten nach der Auswertung.

Wenn Ihre Ziel-Lambda-Funktion gedrosselt ist, reduziert Lambda die Anzahl der Ereignis-Poller. Diese Aktion verringert den Workload der Funktion, indem sie die Anzahl der Nachrichten reduziert, die Ereignis-Poller abrufen und an die Funktion senden können.

Modus bereitgestellter Kapazität

Für Workloads, bei denen Sie den Durchsatz Ihrer Zuordnung von Ereignisquellen optimieren müssen, können Sie den Bereitstellungsmodus verwenden. Im Bereitstellungsmodus definieren Sie Mindest- und Höchstgrenzen für die Anzahl der bereitgestellten Ereignis-Poller. Diese bereitgestellten Event-Poller sind auf Ihre Zuordnung von Ereignisquellen ausgerichtet und können unerwartete Nachrichtenspitzen durch reaktionsschnelle Autoskalierung bewältigen. Wir empfehlen die Verwendung des Bereitstellungsmodus für Kafka-Workloads, die strenge Leistungsanforderungen haben.

In Lambda ist ein Event Poller eine Recheneinheit mit Durchsatzmöglichkeiten, die je nach Art der Ereignisquelle variieren. Bei Amazon MSK und selbstverwaltetem Apache Kafka kann jeder Event Poller bis zu 5% MB/sec des Durchsatzes oder bis zu 5 gleichzeitige Aufrufe verarbeiten. Wenn Ihre Ereignisquelle beispielsweise eine durchschnittliche Nutzlast von 1 MB erzeugt und die

durchschnittliche Dauer Ihrer Funktion 1 Sekunde beträgt, kann ein einziger Kafka-Event-Poller 5 MB/sec Durchsatz- und 5 gleichzeitige Lambda-Aufrufe unterstützen (vorausgesetzt, es gibt keine Payload-Transformation). Für Amazon SQS kann jeder Event Poller bis zu 1 MB/sec Durchsatz oder bis zu 10 gleichzeitige Aufrufe verarbeiten. Bei der Verwendung des Bereitstellungsmodus fallen zusätzliche Kosten an, die von Ihrer Nutzung des Event-Pollers abhängen. Details zu den Preisen finden Sie unter AWS Lambda -Preise.

> **Note**
>
> Wenn Sie den Bereitstellungsmodus verwenden, müssen Sie im Rahmen Ihrer Netzwerkkonfiguration keine AWS PrivateLink VPC-Endpoints erstellen oder die zugehörigen Berechtigungen gewähren.

Im Bereitstellungsmodus liegt der Bereich der akzeptierten Werte für die Mindestanzahl der Ereignis-Poller (`MinimumPollers`) zwischen 1 und 200 (einschließlich). Der Bereich der akzeptierten Werte für die maximale Anzahl von Event-Pollern (`MaximumPollers`) liegt zwischen 1 und einschließlich 2.000. `MaximumPollers` muss größer als oder gleich `MinimumPollers` sein. Um eine geordnete Verarbeitung innerhalb der Partitionen zu gewährleisten, begrenzt Lambda das `MaximumPollers` auf die Anzahl der Partitionen im Thema.

Weitere Einzelheiten zur Auswahl geeigneter Werte für minimale und maximale Ereignis-Poller finden Sie unter Best Practices.

Sie können den Bereitstellungsmodus für Ihre Kafka-Zuordnung von Ereignisquellen über die Konsole oder die Lambda-API konfigurieren.

Konfigurieren des Bereitstellungsmodus für eine vorhandene Zuordnung von Ereignisquellen (Konsole)

1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.
2. Wählen Sie die Funktion mit der Zuordnung von Ereignisquellen aus, für die Sie den Bereitstellungsmodus konfigurieren möchten.
3. Wählen Sie Konfiguration und anschließend Auslöser aus.
4. Wählen Sie die Zuordnung von Ereignisquellen, für die Sie den Bereitstellungsmodus konfigurieren möchten, und wählen Sie dann Bearbeiten.
5. Wählen Sie unter Bereitstellungsmodus die Option Konfigurieren aus.

- Geben Sie für Minimum Event Pollers einen Wert zwischen 1 und 200 ein. Wenn Sie keinen Wert angeben, wählt Lambda einen Standardwert von 1.
- Geben Sie für Maximum Event Pollers einen Wert zwischen 1 und 2000 ein. Dieser Wert muss größer oder gleich Ihrem Wert für Minimum Event Pollers sein. Wenn Sie keinen Wert angeben, wählt Lambda einen Standardwert von 200.

6. Wählen Sie Speichern.

Sie können den Bereitstellungsmodus programmgesteuert mithilfe des Objekts in Ihrem konfigurieren. ProvisionedPollerConfig EventSourceMappingConfiguration Der folgende UpdateEventSourceMappingCLI-Befehl konfiguriert beispielsweise einen `MinimumPollers` Wert von 5 und einen `MaximumPollers` Wert von 100.

```
aws lambda update-event-source-mapping \
    --uuid a1b2c3d4-5678-90ab-cdef-EXAMPLE11111 \
    --provisioned-poller-config '{"MinimumPollers": 5, "MaximumPollers": 100}'
```

Nachdem Sie den Bereitstellungsmodus konfiguriert haben, können Sie die Verwendung von Event-Pollern für Ihren Workload beobachten, indem Sie die `ProvisionedPollers`-Metrik überwachen. Weitere Informationen finden Sie unter the section called “Metriken zur Zuordnung von Ereignisquellen”.

Um den Bereitstellungsmodus zu deaktivieren und zum Standardmodus (auf Abruf) zurückzukehren, können Sie den folgenden UpdateEventSourceMappingCLI-Befehl verwenden:

```
aws lambda update-event-source-mapping \
    --uuid a1b2c3d4-5678-90ab-cdef-EXAMPLE11111 \
    --provisioned-poller-config '{}'
```

Erweiterte Funktionen zur Fehlerbehandlung und Leistung

Für Kafka-Ereignisquellenzuordnungen mit aktiviertem Bereitstellungsmodus können Sie zusätzliche Funktionen konfigurieren, um die Fehlerbehandlung und Leistung zu verbessern:

- Wiederholungskonfigurationen — Steuern Sie, wie Lambda mit fehlgeschlagenen Datensätzen umgeht, indem Sie maximale Wiederholungsversuche, Altersbeschränkungen für Datensätze, Batch-Splitting und partielle Batch-Antworten festlegen.

- Kafka-Ziele bei einem Ausfall — Senden Sie fehlgeschlagene Datensätze zur späteren Verarbeitung oder Analyse an ein Kafka-Thema.

Bewährte Verfahren und Überlegungen bei der Verwendung des Bereitstellungsmodus

Die optimale Konfiguration der minimalen und maximalen Ereignispoller für Ihre Zuordnung von Ereignisquellen hängt von den Leistungsanforderungen Ihrer Anwendung ab. Wir empfehlen Ihnen, mit den standardmäßigen Minimum Event Pollers zu beginnen, um das Leistungsprofil zu erstellen. Passen Sie Ihre Konfiguration auf der Grundlage der beobachteten Nachrichtenverarbeitungsmuster und Ihres gewünschten Leistungsprofils an.

Erhöhen Sie bei Workloads mit starkem Datenverkehr und strengen Leistungsanforderungen die Mindestanzahl der Event-Poller, um plötzliche Nachrichtenfluten zu bewältigen. Um zu ermitteln, wie viele Event-Poller mindestens erforderlich sind, sollten Sie die Nachrichten pro Sekunde Ihres Workloads und die durchschnittliche Payload-Größe berücksichtigen und die Durchsatzkapazität eines einzelnen Event-Pollers (bis zu 5) als Referenz verwenden. MBps

Um eine geordnete Verarbeitung innerhalb einer Partition aufrechtzuerhalten, begrenzt Lambda die maximale Anzahl der Event-Poller auf die Anzahl der Partitionen im Thema. Darüber hinaus hängt die maximale Anzahl von Event-Pollern, auf die Ihr Zuordnung von Ereignisquellen skaliert werden kann, von den Gleichzeitigkeitseinstellungen der Funktion ab.

Wenn Sie den Bereitstellungsmodus aktivieren, aktualisieren Sie Ihre Netzwerkeinstellungen, um AWS PrivateLink VPC-Endpunkte und zugehörige Berechtigungen zu entfernen.

Kostenoptimierung für den Bereitstellungsmodus

Preisgestaltung im Bereitstellungsmodus

Der Bereitstellungsmodus wird auf der Grundlage der bereitgestellten Mindestanzahl an Event-Pollern und der Anzahl der während der automatischen Skalierung verbrauchten Event-Poller berechnet. Die Gebühren werden anhand einer Abrechnungseinheit berechnet, die als Event Poller Unit (EPU) bezeichnet wird. Sie zahlen für die Anzahl und Dauer der EPUs Nutzung, gemessen in. Event-Poller-Unit-hours Sie können den Bereitstellungsmodus entweder mit einem einzigen ESM für leistungssensitive Anwendungen verwenden oder Sie können mehrere ESMs innerhalb derselben VPC gruppieren, um EPU-Kapazität und Kosten gemeinsam zu nutzen. Wir werden uns eingehend mit zwei Funktionen befassen, mit denen Sie Ihre Kosten für den Bereitstellungsmodus optimieren können. Preisdetails finden Sie unter AWS Lambda-Preise.

Verbesserte EPU-Nutzung

Jede EPU unterstützt bis zu 20 MB/s Durchsatzkapazitäten für Ereignisabfragen und unterstützt standardmäßig 10 Ereignisabfragen. Wenn Sie einen Bereitstellungsmodus für Kafka ESM erstellen, indem Sie Mindest- und Höchstwerte für Poller festlegen, wird für die Bereitstellung eine Mindestanzahl von Pollern verwendet, EPUs basierend auf der Standardeinstellung von 10 Event-Pollern pro EPU. Jeder Event-Poller kann jedoch unabhängig skaliert werden, um bis zu 5% MB/s der Durchsatzkapazität zu unterstützen, was möglicherweise eine geringere Dichte von Event-Pollern auf einer bestimmten EPU erfordert und eine Skalierung von auslösen kann. EPUs Die Anzahl der Event Poller, die einer EPU zugewiesen sind, hängt von der Rechenkapazität ab, die von jedem Event Poller verbraucht wird. Dieser Ansatz der verbesserten EPU-Nutzung ermöglicht es Event-Pollern mit unterschiedlichen Durchsatzanforderungen, die EPU-Kapazität effektiv zu nutzen und so die Kosten für alle zu senken. ESMs

ESM-Gruppierung

Um Ihre Kosten für den Bereitstellungsmodus weiter zu optimieren, können Sie mehrere Kafka gruppieren, um die EPU-Kapazität gemeinsam ESMs zu nutzen. Mit ESM-Gruppierung und verbesserter EPU-Nutzung können Sie die Kosten für Workloads mit geringem Durchsatz im Bereitstellungsmodus um bis zu 90% senken, verglichen mit der Ausführung im ESM-Modus. Alle ESMs , die weniger als 1 EPU-Kapazität benötigen, profitieren von der ESM-Gruppierung. Für solche Anwendungen sind ESMs in der Regel nur wenige Mindestanzahl an Event-Pollern erforderlich, um ihren Durchsatzbedarf zu decken. Mit dieser Funktion können Sie den Bereitstellungsmodus für all Ihre Kafka-Workloads verwenden und von Funktionen wie Schemavalidierung, Filterung von Avro/Protobuf Ereignissen, Aufrufen mit geringer Latenz und verbesserter Fehlerbehandlung profitieren, die nur im Bereitstellungsmodus verfügbar sind.

Wenn Sie den `PollerGroupName` Parameter mit demselben Wert für mehrere ESMs innerhalb derselben Amazon VPC konfigurieren, ESMs teilen sich diese EPU-Ressourcen, anstatt jeweils eine eigene EPU-Kapazität zu benötigen. Sie können bis zu 100 ESMs pro Abfragergruppe gruppieren, wobei die maximale Gesamtzahl der Abfragenden ESMs in einer Gruppe 2000 nicht überschreiten darf.

So konfigurieren Sie die ESM-Gruppierung (Konsole)

1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.
2. Wählen Sie Ihre Funktion.
3. Wählen Sie Konfiguration und anschließend Auslöser aus.

4. Wenn Sie eine neue Kafka-Ereignisquellenzuordnung erstellen oder eine bestehende bearbeiten, wählen Sie im Bereitstellungsmodus die Option Konfigurieren aus.
5. Geben Sie für Minimum Event Pollers einen Wert zwischen 1 und 200 ein.
6. Geben Sie für Maximum Event Pollers einen Wert zwischen 1 und 2000 ein.
7. Geben Sie unter Poller-Gruppenname einen Bezeichner für die Gruppe ein. Verwenden Sie denselben Namen für andere Personen, die ESMs Sie gruppieren möchten.
8. Wählen Sie Speichern.

So konfigurieren Sie die ESM-Gruppierung (CLI)AWS

Im folgenden Beispiel wird ein ESM mit einer Poller-Gruppe namens erstellt: `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"
  }'
```

Um derselben Gruppe einen weiteren ESM hinzuzufügen (mit gemeinsamer EPU-Kapazität), verwenden Sie dasselbe: 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**
>
> Sie können den aktualisierenPollerGroupName, um einen ESM in eine andere Gruppe zu verschieben, oder einen ESM aus einer Gruppe entfernen, indem Sie eine leere Zeichenfolge („“) übergeben für: 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": ""
  }'
```

Überlegungen zur Gruppierungsstrategie

- Anwendungsgrenze — Gruppen ESMs , die zu denselben Anwendungen oder Diensten gehören, um die Kosten besser verteilen und verwalten zu können. Erwägen Sie die Verwendung von Namenskonventionen wie app-name-environment (z. B.order-processor-prod).
- Verkehrsmuster — Vermeiden Sie Gruppierungen ESMs mit hohem Durchsatz und starken Datenverkehrsmustern, da dies zu Ressourcenkonflikten führen kann.
- Explosionsradius — Berücksichtigen Sie die Auswirkungen, wenn Probleme mit der gemeinsam genutzten Infrastruktur auftreten. Alle ESMs Mitglieder derselben Gruppe sind von Beschränkungen gemeinsam genutzter Ressourcen betroffen. Für geschäftskritische Workloads möchten Sie möglicherweise separate oder dedizierte Gruppen verwenden. ESMs

Beispiel für Kostenoptimierung

Stellen Sie sich ein Szenario vor, in dem Sie 10 haben ESMs, die jeweils mit einem Event Poller und einem Durchsatz von unter 2 MB/s konfiguriert sind:

Ohne Gruppierung:

- Jeder ESM benötigt eine eigene EPU
- Insgesamt EPUs benötigt: 10
- Kosten pro EPU: 0,185 USD/Stunde in USA Ost (Nord-Virginia)
- Monatliche EPU-Kosten (720 Stunden): 10 × 720 × 0,185$ = 1.332$

Mit Gruppierung:

- Alle 10 ESMs teilen sich die EPU-Kapazität
- 10 Event-Poller passen in 1 EPU (mit neuer Unterstützung für 10 Poller pro EPU)
- Insgesamt benötigt: 1 EPUs
- Monatliche EPU-Kosten (720 Stunden): 1 × 720 × 0,185$ = 133,20$
- Kosteneinsparungen: 90% (Einsparungen von 1.198,80 USD pro Monat)

Apache-Kafka-Abfrage und Stream-Startpositionen in Lambda

Der StartingPosition-Parameter teilt Lambda mit, wann mit dem Lesen von Nachrichten aus Ihrem Stream von Amazon MSK oder selbstverwaltetem Apache Kafka begonnen werden soll. Es stehen drei Optionen zur Auswahl:

- Neueste – Lambda beginnt unmittelbar nach dem neuesten Datensatz im Kafka-Thema mit dem Lesen.
- Horizont trimmen –Lambda beginnt ab dem letzten ungetrimmten Datensatz im Kafka-Thema mit dem Lesen. Dies ist auch der älteste Datensatz im Thema.
- Am Zeitstempel – Lambda beginnt an einer durch einen Zeitstempel definierten Position in Unix-Zeitsekunden mit dem Lesen. Verwenden Sie den Parameter StartingPositionTimeStamp, um den Zeitstempel anzugeben.

Die Stream-Abfrage während der Erstellung oder Aktualisierung einer Zuordnung von Ereignisquellen ist letztendlich konsistent:

- Bei der Erstellung der Zuordnung von Ereignisquellen kann es mehrere Minuten dauern, bis mit der Abfrage von Ereignissen aus dem Stream begonnen wird.
- Während Aktualisierungen der Zuordnung von Ereignisquellen kann es bis zu 90 Sekunden dauern, bis die Abfrage von Ereignissen aus dem Stream gestoppt und neu gestartet wird.

Dieses Verhalten bedeutet, dass die Zuordnung von Ereignisquellen Ereignisse während der Erstellung oder Aktualisierung verpassen könnte, wenn Sie `LATEST` als Startposition für den Stream angeben. Um sicherzustellen, dass keine Ereignisse verpasst werden, geben Sie entweder `TRIM_HORIZON` oder `AT_TIMESTAMP` an.

Anpassbare Verbrauchergruppen-ID in Lambda

Wenn Sie Amazon MSK oder selbstverwaltetes Apache Kafka als Quelle einrichten, können Sie eine Verbrauchergruppen-ID angeben. Diese Verbrauchergruppen-ID ist eine vorhandene Kennung für die Kafka-Verbrauchergruppe, der Ihre Lambda-Funktion beitreten soll. Mit diesem Feature können Sie alle laufenden Kafka-Datensatzverarbeitungs-Setups nahtlos von anderen Verbrauchern auf Lambda migrieren.

Kafka verteilt Nachrichten an alle Verbraucher in einer Verbrauchergruppe. Wenn Sie eine Verbrauchergruppen-ID angeben, die andere aktive Verbraucher hat, empfängt Lambda nur einen Teil der Nachrichten aus dem Kafka-Thema. Wenn Sie möchten, dass Lambda alle Nachrichten im Thema verarbeitet, deaktivieren Sie alle anderen Verbraucher in dieser Verbrauchergruppe.

Wenn Sie außerdem eine Verbrauchergruppen-ID angeben und Kafka eine gültige vorhandene Verbrauchergruppe mit derselben ID findet, ignoriert Lambda die StartingPosition für Ihre Zuordnung von Ereignisquellen. Stattdessen beginnt Lambda mit der Verarbeitung von Datensätzen gemäß dem zugesagten Versatz der Konsumentengruppe. Wenn Sie eine Konsumentengruppen-ID angeben und Kafka keine vorhandene Konsumentengruppe finden kann, konfiguriert Lambda Ihre Ereignisquelle mit der angegebenen `StartingPosition`.

Die Konsumentengruppen-ID, die Sie angeben, muss unter all Ihren Kafka-Ereignisquellen eindeutig sein. Nachdem Sie eine Kafka-Ereignisquellenzuordnung mit der angegebenen Konsumentengruppen-ID erstellt haben, können Sie diesen Wert nicht aktualisieren.

Filtern von Ereignissen aus Amazon MSK und selbstverwalteten Apache-Kafka-Ereignisquellen

Sie können die Ereignisfilterung verwenden, um zu steuern, welche Datensätze aus einem Stream oder einer Warteschlange Lambda an Ihre Funktion sendet. Allgemeine Informationen über die Funktionsweise der Ereignisfilterung finden Sie unter the section called “Ereignisfilterung”.

> **Note**
>
> Zuordnungen von Ereignisquellen von Amazon MSK und selbstverwaltetem Apache Kafka unterstützen nur die Filterung nach dem Schlüssel `value`.

Themen

- Grundlagen der Kafka-Ereignisfilterung

Grundlagen der Kafka-Ereignisfilterung

Angenommen, ein Producer schreibt Nachrichten zu einem Thema in Ihrem Kafka-Cluster, entweder im gültigen JSON-Format oder als einfache Zeichenfolgen. Ein Beispieldatensatz würde wie folgt aussehen, wobei die Nachricht im `value`-Feld in eine Base64-kodierte Zeichenfolge umgewandelt wurde.

```
{
    "mytopic-0":[
        {
            "topic":"mytopic",
            "partition":0,
            "offset":15,
            "timestamp":1545084650987,
            "timestampType":"CREATE_TIME",
            "value":"SGVsbG8sIHRoaXMgaXMgYSB0ZXN0Lg==",
            "headers":[]
        }
    ]
}
```

Angenommen, Ihr Apache-Kafka-Producer schreibt Nachrichten zu Ihrem Thema im folgenden JSON-Format.

```
{
    "device_ID": "AB1234",
    "session":{
        "start_time": "yyyy-mm-ddThh:mm:ss",
        "duration": 162
    }
}
```

Sie können den `value`-Schlüssel verwenden, um Datensätze zu filtern. Angenommen, Sie möchten nur die Datensätze filtern, bei denen `device_ID` mit den Buchstaben AB beginnen. Das `FilterCriteria`-Objekt würde wie folgt aussehen.

```
{
    "Filters": [
        {
            "Pattern": "{ \"value\" : { \"device_ID\" : [ { \"prefix\": \"AB\" } ] } }"
        }
    ]
}
```

Zur Verdeutlichung sehen Sie hier den Wert des Filter-`Pattern` in reinem JSON.

```
{
    "value": {
        "device_ID": [ { "prefix": "AB" } ]
      }
}
```

Sie können Ihren Filter mithilfe der Konsole, AWS CLI oder einer AWS SAM-Vorlage hinzufügen.

Console

Um diesen Filter mithilfe der Konsole hinzuzufügen, folgen Sie den Anweisungen unter Anhängen von Filterkriterien an eine Ereignisquellenzuordnung (Konsole) und geben Sie die folgende Zeichenfolge für die Filterkriterien ein.

```
{ "value" : { "device_ID" : [ { "prefix":  "AB" } ] } }
```

AWS CLI

Um eine neue Zuordnung von Ereignisquellen mit diesen Filterkriterien über die AWS Command Line Interface (AWS CLI) zu erstellen, führen Sie den folgenden Befehl aus.

```
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\" } ] } }"}]}'
```

Führen Sie den folgenden Befehl aus, um diese Filterkriterien zu einer vorhandenen Zuordnung von Ereignisquellen hinzuzufügen.

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"value\" : { \"device_ID\" :
 [ { \"prefix\":  \"AB\" } ] } }"}]}'
```

AWS SAM

Um diesen Filter mit der AWS SAM hinzuzufügen, fügen Sie das folgende Snippet zur YAML-Vorlage für Ihre Ereignisquelle hinzu.

```
FilterCriteria:
  Filters:
    - Pattern: '{ "value" : { "device_ID" : [ { "prefix":  "AB" } ] } }'
```

Mit Kafka können Sie auch Datensätze filtern, bei denen die Nachricht eine einfache Zeichenfolge ist. Angenommen, Sie möchten die Meldungen ignorieren, in denen die Zeichenfolge „error“ ist. Das `FilterCriteria`-Objekt würde wie folgt aussehen.

```
{
    "Filters": [
        {
            "Pattern": "{ \"value\" : [ { \"anything-but\": [ \"error\" ] } ] }"
        }
    ]
```

```
}
```

Zur Verdeutlichung sehen Sie hier den Wert des Filter-`Pattern` in reinem JSON.

```
{
    "value": [
        {
        "anything-but": [ "error" ]
        }
    ]
}
```

Sie können Ihren Filter mithilfe der Konsole, AWS CLI oder einer AWS SAM-Vorlage hinzufügen.

Console

Um diesen Filter mithilfe der Konsole hinzuzufügen, folgen Sie den Anweisungen unter Anhängen von Filterkriterien an eine Ereignisquellenzuordnung (Konsole) und geben Sie die folgende Zeichenfolge für die Filterkriterien ein.

```
{ "value" : [ { "anything-but": [ "error" ] } ] }
```

AWS CLI

Um eine neue Zuordnung von Ereignisquellen mit diesen Filterkriterien über die AWS Command Line Interface (AWS CLI) zu erstellen, führen Sie den folgenden Befehl aus.

```
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\" ] } ] }"}]}'
```

Führen Sie den folgenden Befehl aus, um diese Filterkriterien zu einer vorhandenen Zuordnung von Ereignisquellen hinzuzufügen.

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"value\" : [ { \"anything-but\":
 [ \"error\" ] } ] }"}]}'
```

AWS SAM

Um diesen Filter mit der AWS SAM hinzuzufügen, fügen Sie das folgende Snippet zur YAML-Vorlage für Ihre Ereignisquelle hinzu.

```
FilterCriteria:
  Filters:
    - Pattern: '{ "value" : [ { "anything-but": [ "error" ] } ] }'
```

Kafka-Nachrichten müssen UTF-8-codierte Zeichenfolgen sein (entweder einfache Zeichenfolgen oder im JSON-Format). Das liegt daran, dass Lambda Byte-Arrays von Kafka vor Anwendung der Filterkriterien in UTF-8 dekodiert. Wenn Ihre Nachrichten eine andere Kodierung nutzen, z. B. UTF-16 oder ASCII, oder das Nachrichtenformat nicht dem `FilterCriteria`-Format entspricht, verarbeitet Lambda nur Metadatenfilter. In der folgenden Tabelle ist das Verhalten zusammengefasst:

-Format der eingehenden Nachricht	Filtermusterformat für Nachrichteneigenschaften	Resultierende Aktion
Einfache Zeichenfolge	Einfache Zeichenfolge	Lambda filtert basierend auf Ihren Filterkriterien.
Einfache Zeichenfolge	Kein Filtermuster für Dateneigenschaften	Lambda filtert (nur für die anderen Metadateneigenschaften) basierend auf Ihren Filterkriterien.
Einfache Zeichenfolge	Gültiges JSON	Lambda filtert (nur für die anderen Metadateneigenschaften) basierend auf Ihren Filterkriterien.
Gültiges JSON	Einfache Zeichenfolge	Lambda filtert (nur für die anderen Metadateneigenschaften) basierend auf Ihren Filterkriterien.
Gültiges JSON	Kein Filtermuster für Dateneigenschaften	Lambda filtert (nur für die anderen Metadateneigenscha

-Format der eingehenden Nachricht	Filtermusterformat für Nachrichteneigenschaften	Resultierende Aktion
		ften) basierend auf Ihren Filterkriterien.
Gültiges JSON	Gültiges JSON	Lambda filtert basierend auf Ihren Filterkriterien.
Nicht UTF-8-kodierte Zeichenfolge	JSON, einfache Zeichenfolge oder kein Muster	Lambda filtert (nur für die anderen Metadateneigenscha ften) basierend auf Ihren Filterkriterien.

Verwenden von Schemaregistern mit Kafka-Ereignisquellen in Lambda

Schemaregister unterstützen Sie bei der Definition und Verwaltung von Datenstromschemas. Ein Schema definiert die Struktur und das Format eines Datensatzes. Im Zusammenhang mit Zuordnungen von Kafka-Ereignisquellen können Sie eine Schemaregistrierung konfigurieren, um die Struktur und das Format von Kafka-Nachrichten anhand vordefinierter Schemas zu validieren, bevor sie Ihre Lambda-Funktion erreichen. Dies erweitert Ihre Anwendung um eine Ebene der Daten-Governance und ermöglicht Ihnen die effiziente Verwaltung von Datenformaten, die Sicherstellung der Schema-Compliance und die Kostenoptimierung durch Ereignisfilterung.

Dieses Feature ist mit allen Programmiersprachen kompatibel, jedoch sollten Sie folgende wichtige Punkte beachten:

- Powertools for Lambda bietet spezifische Unterstützung für Java, Python und TypeScript gewährleistet die Konsistenz mit bestehenden Kafka-Entwicklungsmustern und ermöglicht den direkten Zugriff auf Geschäftsobjekte ohne benutzerdefinierten Deserialisierungscode
- Dieses Feature ist nur für Zuordnungen von Ereignisquellen im Bereitstellungsmodus verfügbar. Die Schemaregistrierung unterstützt keine Zuordnungen von Ereignisquellen im On-Demand-Modus. Wenn Sie den Bereitstellungsmodus verwenden und eine Schemaregistrierung konfiguriert haben, können Sie nicht in den On-Demand-Modus wechseln, es sei denn, Sie entfernen zuvor Ihre Schemaregistrierungskonfiguration. Weitere Informationen finden Sie unter the section called “Modus bereitgestellter Kapazität”.
- Sie können nur eine Schemaregistrierung pro Zuordnung von Ereignisquellen (Event Source Mapping, (ESM) konfigurieren. Die Verwendung einer Schemaregistrierung mit Ihrer Kafka-Ereignisquelle kann zu einer erhöhten Nutzung Ihrer Lambda Event Poller Unit (EPU) führen, die eine Preisdimension für den Bereitstellungsmodus darstellt.

Topics

- Optionen für die Schemaregistrierung
- So führt Lambda die Schemavalidierung für Kafka-Nachrichten durch
- Konfiguration einer Kafka-Schemaregistrierung
- Filterung nach Avro und Protobuf
- Nutzdatenformate und Deserialisierungsverhalten
- Arbeiten mit deserialisierten Daten in Lambda-Funktionen
- Authentifizierungsmethoden für Ihre Schemaregistrierung

- Fehlerbehandlung und -behebung bei Problemen mit der Schemaregistrierung

Optionen für die Schemaregistrierung

Lambda unterstützt die folgenden Optionen für die Schemaregistrierung:

- AWS Glue Schemaregistrierung
- Confluent-Cloud-Schemaregistrierung
- Selbstverwaltete Confluent-Schemaregistrierung

Ihre Schemaregistrierung unterstützt die Validierung von Nachrichten in den folgenden Datenformaten:

- Apache Avro
- Protokollpuffer (Protobuf)
- JSON-Schema (JSON-SE)

Um eine Schemaregistrierung zu verwenden, stellen Sie zunächst sicher, dass sich Ihre Zuordnung von Ereignisquellen im Bereitstellungsmodus befindet. Wenn Sie eine Schemaregistrierung verwenden, fügt Lambda Metadaten zum Schema zu den Nutzdaten hinzu. Weitere Informationen finden Sie unter Nutzdatenformate und Deserialisierungsverhalten.

So führt Lambda die Schemavalidierung für Kafka-Nachrichten durch

Wenn Sie eine Schemaregistrierung konfigurieren, führt Lambda für jede Kafka-Nachricht die folgenden Schritte aus:

1. Lambda fragt den Kafka-Datensatz aus Ihrem Cluster ab.
2. Lambda validiert ausgewählte Nachrichtenattribute im Datensatz anhand eines bestimmten Schemas in Ihrer Schemaregistrierung.
   - Wenn das der Nachricht zugeordnete Schema nicht in der Registrierung gefunden wird, sendet Lambda die Nachricht mit dem Ursachencode `SCHEMA_NOT_FOUND` an eine DLQ.
3. Lambda deserialisiert die Nachricht gemäß der Schemaregistrierungskonfiguration, um die Nachricht zu validieren. Wenn eine Ereignisfilterung konfiguriert ist, führt Lambda anschließend eine Filterung basierend auf den konfigurierten Filterkriterien durch.

- Wenn die Deserialisierung fehlschlägt, sendet Lambda die Nachricht mit dem Ursachencode `DESERIALIZATION_ERROR` an eine DLQ. Wenn keine DLQ konfiguriert ist, verwirft Lambda die Nachricht.

4. Wenn die Nachricht von der Schemaregistrierung validiert und nicht durch Ihre Filterkriterien herausgefiltert wird, ruft Lambda Ihre Funktion mit der Nachricht auf.

Dieses Feature dient zur Validierung von Nachrichten, die bereits mit Kafka-Clients erstellt wurden, die in eine Schemaregistrierung integriert sind. Wir empfehlen, Ihre Kafka-Producer so zu konfigurieren, dass sie mit Ihrer Schemaregistrierung zusammenarbeiten, um korrekt formatierte Nachrichten zu erstellen.

Konfiguration einer Kafka-Schemaregistrierung

Die folgenden Konsolenschritte fügen Ihrer Zuordnung von Ereignisquellen eine Kafka-Schemaregistrierungskonfiguration hinzu.

Hinzufügen einer Kafka-Schemaregistrierungskonfiguration zu Ihrer Zuordnung von Ereignisquellen (Konsole)

1. Öffnen Sie die Seite Funktion der Lambda-Konsole.
2. Wählen Sie Konfiguration.
3. Wählen Sie Auslöser.
4. Wählen Sie die Kafka-Zuordnung von Ereignisquellen, für die Sie eine Schemaregistrierung konfigurieren möchten und wählen Sie Bearbeiten.
5. Wählen Sie unter Konfiguration des Event-Pollers die Option Schemaregistrierung konfigurieren aus. Ihre Zuordnung von Ereignisquellen muss sich im Bereitstellungsmodus befinden, damit diese Option angezeigt wird.
6. Geben Sie für Schema Registry URI den ARN Ihrer AWS Glue Schemaregistry oder die HTTPS-URL Ihrer Confluent Cloud-Schemaregistry oder Self-Managed Confluent Schema Registry ein.
7. Die folgenden Konfigurationsschritte teilen Lambda mit, wie auf Ihre Schemaregistrierung zugegriffen werden kann. Weitere Informationen finden Sie unter ???.

    - Wählen Sie für Zugriffskonfigurationstyp den Authentifizierungstyp aus, den Lambda für den Zugriff auf Ihre Schemaregistrierung verwendet.

- Geben Sie für URI der Zugriffskonfiguration die ARN des Secrets-Manager-Geheimnisses ein, um sich bei Ihrer Schemaregistrierung zu authentifizieren, falls zutreffend. Stellen Sie sicher, dass die Ausführungsrolle Ihrer Funktion die richtigen Berechtigungen enthält.

8. Das Feld Verschlüsselung gilt nur, wenn Ihre Schemaregistrierung von einer privaten Zertifizierungsstelle (CA) oder einer Zertifizierungsstelle (CA) signiert ist, die sich nicht im Lambda-Vertrauensspeicher befindet. Geben Sie gegebenenfalls den geheimen Schlüssel mit dem privaten CA-Zertifikat an, das von Ihrer Schemaregistrierung für die TLS-Verschlüsselung verwendet wird.
9. Wählen Sie unter Ereignisdatensatzformat aus, wie Lambda die Datensätze nach der Schemavalidierung an Ihre Funktion übermitteln soll. Weitere Informationen finden Sie unter Beispiele für das Nutzdatenformat.

    - Wenn Sie JSON wählen, liefert Lambda die Attribute, die Sie unten unter „Schemavalidierungsattribut“ auswählen, im Standard-JSON-Format. Die Attribute, die Sie nicht auswählen, liefert Lambda unverändert.
    - Wenn Sie SOURCE wählen, liefert Lambda die Attribute, die Sie unten unter „Schemavalidierungsattribut“ auswählen, im ursprünglichen Quellformat.

10. Wählen Sie für Schemavalidierungsattribut die Nachrichtenattribute aus, die Lambda mithilfe Ihrer Schemaregistrierung validieren und deserialisieren soll. Sie müssen mindestens entweder KEY oder VALUE auswählen. Wenn Sie JSON als Ereignisdatensatzformat ausgewählt haben, deserialisiert Lambda auch die ausgewählten Nachrichtenattribute, bevor sie an Ihre Funktion gesendet werden. Weitere Informationen finden Sie unter Nutzdatenformate und Deserialisierungsverhalten.
11. Wählen Sie Speichern.

Sie können auch die Lambda-API verwenden, um Ihre Zuordnung von Ereignisquellen mit einer Schemaregistrierungskonfiguration zu erstellen oder zu aktualisieren. Die folgenden Beispiele zeigen, wie Sie eine AWS Glue oder Confluent-Schemaregistry mithilfe von konfigurieren AWS CLI, was den API-Operationen UpdateEventSourceMappingund CreateEventSourceMappingin der API-Referenz entspricht:AWS Lambda

> **Important**
>
> Wenn Sie ein Konfigurationsfeld für die Schemaregistrierung mithilfe der AWS CLI oder der `update-event-source-mapping` API aktualisieren, müssen Sie alle Felder der Schemaregistrierungskonfiguration aktualisieren.

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

Um eine Schemaregistrierungskonfiguration aus Ihrer Ereignisquellenzuordnung zu entfernen, können Sie den CLI-Befehl UpdateEventSourceMappingin der AWS Lambda API-Referenz verwenden.

```
aws lambda update-event-source-mapping \
    --uuid a1b2c3d4-5678-90ab-cdef-EXAMPLE11111 \
    --amazon-managed-kafka-event-source-mapping '{
        "SchemaRegistryConfig" : {}
    }'
```

Filterung nach Avro und Protobuf

Wenn Sie die Formate Avro oder Protobuf mit einer Schemaregistrierung verwenden, können Sie die Ereignisfilterung auf Ihre Lambda-Funktion anwenden. Die Filtermuster werden nach der Schemavalidierung auf die deserialisierte klassische JSON-Darstellung Ihrer Daten angewendet. Mit einem Avro-Schema, das Produktdetails einschließlich des Preises definiert, können Sie beispielsweise Nachrichten basierend auf dem Preiswert filtern:

> **Note**
>
> Bei der Deserialisierung wird Avro in Standard-JSON konvertiert, was bedeutet, dass es nicht direkt wieder in ein Avro-Objekt konvertiert werden kann. Wenn Sie eine Konvertierung in ein Avro-Objekt benötigen, verwenden Sie stattdessen das SOURCE-Format.
> Bei der Protobuf-Deserialisierung stimmen die Feldnamen im resultierenden JSON mit denen überein, die in Ihrem Schema definiert sind, anstatt wie bei Protobuf üblich in Camelcase konvertiert zu werden. Beachten Sie dies bei der Erstellung von Filtermustern.

```
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\"] } }"
            }
        ]
    }'
```

In diesem Beispiel analysiert das Filtermuster das `value`-Objekt und gleicht Nachrichten in `field_1` mit `"value1"` und `field_2` mit `"value2"` ab. Die Filterkriterien werden anhand der deserialisierten Daten ausgewertet, nachdem Lambda die Nachricht vom Avro-Format in JSON konvertiert hat.

Ausführlichere Informationen zur Ereignisfilterung finden Sie unter Lambda-Ereignisfilterung.

Nutzdatenformate und Deserialisierungsverhalten

Bei Verwendung einer Schemaregistrierung liefert Lambda die endgültigen Nutzdaten in einem Format, das den regulären Ereignisnutzdaten ähnelt, jedoch einige zusätzliche Felder enthält. Die zusätzlichen Felder hängen vom Parameter `SchemaValidationConfigs` ab. Für jedes Attribut, das Sie für die Validierung auswählen (Schlüssel oder Wert), fügt Lambda den Nutzdaten entsprechende Schemametadaten hinzu.

> **Note**
>
> Sie müssen Ihr System aws-lambda-java-eventsauf Version 3.16.0 oder höher aktualisieren, um Schema-Metadatenfelder verwenden zu können.

Wenn Sie beispielsweise das Feld `value` validieren, fügt Lambda Ihren Nutzdaten ein Feld namens `valueSchemaMetadata` hinzu. In ähnlicher Weise fügt Lambda für das Feld `key` ein Feld namens `keySchemaMetadata` hinzu. Diese Metadaten enthalten Informationen über das Datenformat und die Schema-ID, die für die Validierung verwendet werden:

```
"valueSchemaMetadata": {
    "dataFormat": "AVRO",
    "schemaId": "a1b2c3d4-5678-90ab-cdef-EXAMPLE11111"
}
```

Der Parameter `EventRecordFormat` kann entweder auf `JSON` oder `SOURCE` gesetzt werden. Damit wird festgelegt, wie Lambda schemavalidierte Daten verarbeitet, bevor sie an Ihre Funktion übermittelt werden. Jede Option bietet unterschiedliche Verarbeitungsmöglichkeiten:

- `JSON` – Lambda deserialisiert die validierten Attribute in das Standard-JSON-Format, sodass die Daten direkt in Sprachen mit nativer JSON-Unterstützung verwendet werden können. Dieses Format ist ideal, wenn Sie das ursprüngliche Binärformat nicht beibehalten oder mit generierten Klassen arbeiten müssen.
- `SOURCE` – Lambda behält das ursprüngliche Binärformat der Daten als Base64-codierte Zeichenfolge bei, sodass eine direkte Konvertierung in Avro- oder Protobuf-Objekte möglich ist. Dieses Format ist unerlässlich, wenn Sie mit stark typisierten Sprachen arbeiten oder die volle Leistungsfähigkeit von Avro- oder Protobuf-Schemas beibehalten müssen.

Basierend auf diesen Formatmerkmalen und sprachspezifischen Überlegungen empfehlen wir die folgenden Formate:

Empfohlene Formate basierend auf der Programmiersprache

Sprache	Avro	Protobuf	JSON
Java	SOURCE	SOURCE	SOURCE

Sprache	Avro	Protobuf	JSON
Python	JSON	JSON	JSON
NodeJS	JSON	JSON	JSON
.NET	SOURCE	SOURCE	SOURCE
Weitere	JSON	JSON	JSON

In den folgenden Abschnitten werden diese Formate ausführlich beschrieben und Beispielnutzdaten für jedes Format bereitgestellt.

JSON-Format

Wenn Sie dies wählen `JSONEventRecordFormat`, validiert und deserialisiert Lambda die Nachrichtenattribute, die Sie in dem `SchemaValidationConfigs` Feld ausgewählt haben (die Attribute). `key` and/or `value` Lambda liefert diese ausgewählten Attribute als Base64-codierte Zeichenfolgen ihrer Standard-JSON-Darstellung in Ihrer Funktion.

> **Note**
>
> Bei der Deserialisierung wird Avro in Standard-JSON konvertiert, was bedeutet, dass es nicht direkt wieder in ein Avro-Objekt konvertiert werden kann. Wenn Sie eine Konvertierung in ein Avro-Objekt benötigen, verwenden Sie stattdessen das SOURCE-Format.
> Bei der Protobuf-Deserialisierung stimmen die Feldnamen im resultierenden JSON mit denen überein, die in Ihrem Schema definiert sind, anstatt wie bei Protobuf üblich in Camelcase konvertiert zu werden. Beachten Sie dies bei der Erstellung von Filtermustern.

Im Folgenden finden Sie ein Beispiel für Nutzdaten, wobei davon ausgegangen wird, dass Sie `JSON` als `EventRecordFormat` und sowohl die `key`- als auch die `value`-Attribute als `SchemaValidationConfigs` wählen.

```
{
   "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
                  ]
               }
            ]
         }
      ]
   }
}
```

In diesem Beispiel:

- Sowohl `key` als auch `value` sind Base64-codierte Zeichenfolgen ihrer JSON-Darstellung nach der Deserialisierung.
- Lambda enthält Schemametadaten für beide Attribute in `keySchemaMetadata` und`valueSchemaMetadata`.
- Ihre Funktion kann die `key`- und `value`-Zeichenfolgen decodieren, um auf die deserialisierten JSON-Daten zuzugreifen.

Das JSON-Format wird für Sprachen empfohlen, die nicht stark typisiert sind, wie Python oder Node.js. Diese Sprachen bieten native Unterstützung für die Konvertierung von JSON in Objekte.

Quellformat

Wenn Sie `SOURCE` als `EventRecordFormat` auswählen, validiert Lambda den Datensatz trotzdem anhand der Schemaregistrierung, liefert jedoch die ursprünglichen Binärdaten ohne Deserialisierung an Ihre Funktion. Diese Binärdaten werden als Base64-codierte Zeichenfolge der ursprünglichen Byte-Daten geliefert, wobei die vom Producer angehängten Metadaten entfernt werden. Dadurch können Sie die binären Rohdaten direkt in Avro- und Protobuf-Objekte innerhalb Ihres Funktionscodes konvertieren. Wir empfehlen die Verwendung von Powertools for AWS Lambda, das die rohen Binärdaten deserialisiert und Ihnen Avro- und Protobuf-Objekte direkt zur Verfügung stellt.

Wenn Sie beispielsweise Lambda so konfigurieren, dass sowohl die `key`- als auch die `value`-Attribute validiert werden, aber das `SOURCE`-Format verwenden, erhält Ihre Funktion Nutzdaten wie diese:

```
{
    "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
                        ]
                    }
                ]
            }
        ]
    }
}
```

In diesem Beispiel:

- Sowohl `key` als auch `value` enthalten die ursprünglichen Binärdaten als Base64-codierte Zeichenfolgen.
- Ihre Funktion muss die Deserialisierung mit den entsprechenden Bibliotheken verarbeiten.

Die Auswahl von `SOURCE` für `EventRecordFormat` wird empfohlen, wenn Sie mit Avro oder Protobuf generierte Objekte verwenden, insbesondere mit Java-Funktionen. Dies liegt daran, dass Java stark typisiert ist und spezifische Deserialisierer für die Formate Avro und Protobuf erfordert. In Ihrem Funktionscode können Sie Ihre bevorzugte Avro- oder Protobuf-Bibliothek verwenden, um die Daten zu deserialisieren.

Arbeiten mit deserialisierten Daten in Lambda-Funktionen

Powertools for AWS Lambda hilft Ihnen dabei, Kafka-Datensätze in Ihrem Funktionscode auf der Grundlage des von Ihnen verwendeten Formats zu deserialisieren. Dieses Tool vereinfacht die Arbeit mit Kafka-Datensätzen, indem es die Datenkonvertierung übernimmt und Objekte bereitstellt. ready-to-use

Um Powertools for AWS Lambda in Ihrer Funktion zu verwenden, müssen Sie Powertools for AWS Lambda entweder als Ebene oder als Abhängigkeit hinzufügen, wenn Sie Ihre Lambda-Funktion erstellen. Anweisungen zur Einrichtung und weitere Informationen finden Sie in der AWS Lambda Powertools-Dokumentation für Ihre bevorzugte Sprache:

- Powertools für AWS Lambda (Java)
- Powertools für AWS Lambda (Python)
- Elektrowerkzeuge für AWS Lambda () TypeScript
- Elektrowerkzeuge für AWS Lambda (.NET)

> **Note**
>
> Bei der Arbeit mit der Schemaregistrierungsintegration können Sie zwischen dem `SOURCE`- und dem `JSON`-Format wählen. Jede Option unterstützt verschiedene Serialisierungsformate, wie unten gezeigt:

Format	Unterstützt
SOURCE	Avro und Protobuf (unter Verwendung der Lambda-Schemaregistrierungsintegration)
JSON	JSON-Daten

Wenn Sie das JSON Format SOURCE oder verwenden, können Sie Powertools for verwenden, AWS um die Daten in Ihrem Funktionscode zu deserialisieren. Hier sind Beispiele für den Umgang mit verschiedenen Datenformaten:

AVRO

Java-Beispiel:

```
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";
    }

}
```

Python-Beispiel:

```
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 Beispiel:

```
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,
        },
    }
);
```

.NET-Beispiel:

```
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

Java-Beispiel:

```
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";
    }

}
```

Python-Beispiel:

```
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 Beispiel:

```
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,
```

```
        },
    }
);
```

.NET-Beispiel:

```
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

Java-Beispiel:

```
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";
    }
}
```

Python-Beispiel:

```
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 Beispiel:

```
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',
        },
    }
);
```

.NET-Beispiel:

```
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";
    }
}
```

Authentifizierungsmethoden für Ihre Schemaregistrierung

Um eine Schemaregistrierung verwenden zu können, muss Lambda in der Lage sein, sicher darauf zuzugreifen. Wenn Sie mit einer AWS Glue Schemaregistrierung arbeiten, stützt sich Lambda auf die IAM-Authentifizierung. Das bedeutet, dass die Ausführungsrolle Ihrer Funktion über die folgenden Berechtigungen für den Zugriff auf die AWS Glue Registrierung verfügen muss:

- GetRegistryin der AWS Glue Web-API-Referenz
- GetSchemaVersionin der AWS Glue Web-API-Referenz

Beispiel für die erforderliche IAM-Richtlinie:

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "glue:GetRegistry",
                "glue:GetSchemaVersion"
            ],
            "Resource": [
                "*"
            ]
        }
    ]
}
```

> **Note**
>
> Wenn Sie bei AWS Glue Schemaregistern eine AWS Glue Registrierung angeben`AccessConfigs`, gibt Lambda eine Validierungsausnahme zurück.

Wenn Sie mit einer Confluent-Schemaregistry arbeiten, können Sie eine von drei unterstützten Authentifizierungsmethoden für den `Type` Parameter Ihres Objekts wählen: KafkaSchemaRegistryAccessConfig

- BASIC_AUTH – Lambda verwendet die Authentifizierung mit Benutzername und Passwort oder API-Schlüssel und API-Geheimnis, um auf Ihre Registrierung zuzugreifen. Wenn Sie diese Option auswählen, geben Sie die Secrets-Manager-ARN mit Ihren Anmeldeinformationen im Feld „URI“ an.
- CLIENT_CERTIFICATE_TLS_AUTH – Lambda verwendet die gegenseitige TLS-Authentifizierung mit Client-Zertifikaten. Um diese Option zu verwenden, benötigt Lambda Zugriff sowohl auf das Zertifikat als auch auf den privaten Schlüssel. Geben Sie den Secrets-Manager-ARN mit diesen Anmeldeinformationen im URI-Feld an.
- NO_AUTH – Das öffentliche CA-Zertifikat muss von einer Zertifizierungsstelle (CA) signiert sein, die sich im Lambda-Vertrauensspeicher befindet. Für ein privates CA-/selbstsigniertes Zertifikat konfigurieren Sie das Root-CA-Zertifikat des Servers. Um diese Option zu verwenden, lassen Sie den Parameter `AccessConfigs` weg.

Wenn Lambda Zugriff auf ein privates CA-Zertifikat benötigt, um das TLS-Zertifikat Ihrer Schemaregistrierung zu überprüfen, wählen Sie zusätzlich `SERVER_ROOT_CA_CERT` als `Type` und geben Sie den Secrets-Manager-ARN für das Zertifikat in das URI-Feld ein.

> **Note**
>
> Um die Option `SERVER_ROOT_CA_CERT` in der Konsole zu konfigurieren, geben Sie den Geheimnis-ARN, der das Zertifikat enthält, in das Feld Verschlüsselung ein.

Die Authentifizierungskonfiguration für Ihre Schemaregistrierung ist unabhängig von der Authentifizierung, die Sie für Ihren Kafka-Cluster konfiguriert haben. Sie müssen beide separat konfigurieren, auch wenn sie ähnliche Authentifizierungsmethoden verwenden.

Fehlerbehandlung und -behebung bei Problemen mit der Schemaregistrierung

Bei der Verwendung einer Schemaregistrierung mit Ihrer Amazon-MSK-Ereignisquelle können verschiedene Fehler auftreten. Dieser Abschnitt enthält Anleitungen zu häufigen Problemen und deren Behebung.

Konfigurationsfehler

Diese Fehler treten beim Einrichten Ihrer Schemaregistrierungskonfiguration auf.

Bereitstellungsmodus erforderlich

- Fehlermeldung: `SchemaRegistryConfig is only available for Provisioned Mode. To configure Schema Registry, please enable Provisioned Mode by specifying MinimumPollers in ProvisionedPollerConfig.`
- Lösung: Aktivieren Sie den Bereitstellungsmodus für Ihre Zuordnung von Ereignisquellen, indem Sie den Parameter `MinimumPollers` in `ProvisionedPollerConfig` konfigurieren.

Ungültige Schemaregistrierungs-URL

- Fehlermeldung: `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.`
- Lösung: Geben Sie eine gültige HTTPS-URL für Confluent Schema Registry oder einen gültigen ARN für AWS Glue Schema Registry an.

Ungültiges oder fehlendes Ereignisdatensatzformat

- Fehlermeldung: `EventRecordFormat is a required field for SchemaRegistryConfig. Please provide one of supported format types: SOURCE, JSON.`
- Lösung: Geben Sie EventRecordFormat in Ihrer Schemaregistrierungskonfiguration entweder SOURCE oder JSON an.

Doppelte Validierungsattribute

- Fehlermeldung: `Duplicate KEY/VALUE Attribute in SchemaValidationConfigs. SchemaValidationConfigs must contain at most one KEY/VALUE Attribute.`
- Lösung: Entfernen Sie doppelte KEY- oder VALUE-Attribute aus Ihrem SchemaValidationConfigs. Jeder Attributtyp darf nur einmal vorkommen.

Fehlende Validierungskonfiguration

Fehlermeldung: `SchemaValidationConfigs is a required field for SchemaRegistryConfig.`

Lösung: Fügen Sie SchemaValidationConfigs es zu Ihrer Konfiguration hinzu und geben Sie mindestens ein Validierungsattribut (KEY oder VALUE) an.

Zugriffs- und Berechtigungsfehler

Diese Fehler treten auf, wenn Lambda aufgrund von Berechtigungs- oder Authentifizierungsproblemen nicht auf die Schemaregistrierung zugreifen kann.

AWS Glue Zugriff auf die Schemaregistrierung verweigert

Fehlermeldung: `Cannot access Glue Schema with provided role. Please ensure the provided role can perform the GetRegistry and GetSchemaVersion Actions on your schema.`

Lösung: Fügen Sie die erforderlichen Berechtigungen (`glue:GetRegistry` und `glue:GetSchemaVersion`) zur Ausführungsrolle Ihrer Funktion hinzu.

Zugriff auf Confluent-Schemaregistrierung verweigert

Fehlermeldung: `Cannot access Confluent Schema with the provided access configuration.`

Lösung: Stellen Sie sicher, dass Ihre Authentifizierungsanmeldeinformationen (gespeichert in Secrets Manager) korrekt sind und über die erforderlichen Berechtigungen für den Zugriff auf die Schemaregistrierung verfügen.

Kontoübergreifende AWS Glue Schemaregistrierung

Fehlermeldung: `Cross-account Glue Schema Registry ARN not supported.`

Lösung: Verwenden Sie eine AWS Glue Schemaregistry, die sich im selben AWS Konto wie Ihre Lambda-Funktion befindet.

Regionsübergreifende Schemaregistrierung AWS Glue

Fehlermeldung: `Cross-region Glue Schema Registry ARN not supported.`

Lösung: Verwenden Sie eine AWS Glue Schemaregistry, die sich in derselben Region wie Ihre Lambda-Funktion befindet.

Probleme beim Zugriff auf Geheimnisse

Fehlermeldung: `Lambda received InvalidRequestException from Secrets Manager.`

Lösung: Stellen Sie sicher, dass die Ausführungsrolle Ihrer Funktion berechtigt ist, auf das Geheimnis zuzugreifen, und dass das Geheimnis nicht mit einem AWS KMS Standardschlüssel verschlüsselt ist, wenn Sie von einem anderen Konto aus zugreifen.

Verbindungsfehler

Diese Fehler treten auf, wenn Lambda keine Verbindung mit der Schemaregistrierung herstellen kann.

Probleme mit der VPC-Konnektivität

Fehlermeldung: `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.`

Lösung: Konfigurieren Sie Ihr VPC-Netzwerk so, dass Verbindungen zur Schemaregistry über AWS PrivateLink ein NAT-Gateway oder VPC-Peering möglich sind.

TLS-Handshake-Fehler

Fehlermeldung: `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.`

Lösung: Überprüfen Sie, ob Ihre CA-Zertifikate und Client-Zertifikate (für mTLS) korrekt sind und ordnungsgemäß in Secrets Manager konfiguriert wurden.

Drosselung

Fehlermeldung: `Receiving throttling errors when accessing the schema registry. Please increase API TPS limits for your schema registry.`

Lösung: Erhöhen Sie die API-Ratenlimits für Ihre Schemaregistrierung oder reduzieren Sie die Anzahl der Anforderungen von Ihrer Anwendung.

Fehler bei selbstverwalteten Schemaregistrierungen

Fehlermeldung: `Lambda received an internal server an unexpected error from the provided self-managed schema registry.`

Lösung: Überprüfen Sie den Zustand und die Konfiguration des Servers Ihrer selbstverwalteten Schemaregistrierung.

Verarbeitung mit niedriger Latenz für Kafka-Ereignisquellen

AWS Lambda unterstützt nativ die Verarbeitung von Ereignissen mit geringer Latenz für Anwendungen, die konsistente End-to-End-Latenzen von weniger als 100 Millisekunden erfordern. Auf dieser Seite finden Sie Konfigurationsdetails und Empfehlungen zur Aktivierung von Workflows mit geringer Latenz.

Verarbeitung mit geringer Latenz aktivieren

Um die Verarbeitung mit geringer Latenz für eine Kafka-Zuordnung von Ereignisquellen zu aktivieren, ist die folgende Grundkonfiguration erforderlich:

- Aktivieren Sie den Bereitstellungsmodus. Weitere Informationen finden Sie unter Modus bereitgestellter Kapazität.
- Setzen Sie den Parameter `MaximumBatchingWindowInSeconds` der Zuordnung von Ereignisquellen auf 0. Weitere Informationen finden Sie unter Batching-Verhalten.

Feinabstimmung Ihres Kafka ESM mit niedriger Latenz

Beachten Sie die folgenden Empfehlungen, um Ihre Kafka-Zuordnung von Ereignisquellen für geringe Latenz zu optimieren:

Konfiguration des Bereitstellungsmodus

Im Bereitstellungsmodus für die Kafka-Zuordnung von Ereignisquellen können Sie mit Lambda den Durchsatz Ihrer Zuordnung von Ereignisquellen optimieren, indem Sie eine minimale und maximale Anzahl von Ressourcen konfigurieren, die als Ereignis-Poller bezeichnet werden. Ein Ereignis-Poller (oder ein Poller) stellt eine Rechenressource dar, die eine Zuordnung von Ereignisquellen im Bereitstellungsmodus unterstützt und einen Durchsatz von bis zu 5 MB/s zuweist. Jeder Ereignis-Poller unterstützt bis zu 5 gleichzeitige Lambda-Aufrufe.

Um die optimale Poller-Konfiguration für Ihre Anwendung zu ermitteln, berücksichtigen Sie Ihre maximale Erfassungsrate und Ihre Verarbeitungsanforderungen. Sehen wir uns ein vereinfachtes Beispiel an:

Bei einer Batchgröße von 20 Datensätzen und einer durchschnittlichen Zielfunktionsdauer von 50 ms kann jeder Poller 2 000 Datensätze pro Sekunde verarbeiten, vorbehaltlich einer Begrenzung von 5 MB/s. Dies wird wie folgt berechnet: (20 Datensätze × 1 000 ms/50 ms) × 5 gleichzeitige Lambda-

Aufrufe. Wenn Ihre gewünschte Spitzenerfassungsrate also 20 000 Datensätze pro Sekunde beträgt, benötigen Sie mindestens 10 Ereignis-Poller.

> **Note**
>
> Wir empfehlen, zusätzliche Ereignis-Pollers als Puffer bereitzustellen, um einen konstanten Betrieb bei maximaler Kapazität zu vermeiden.

Der Bereitstellungsmodus skaliert Ihre Ereignis-Poller automatisch basierend auf den Datenverkehrsmustern innerhalb der konfigurierten minimalen und maximalen Ereignis-Poller, was eine Neugewichtung auslösen und somit zusätzliche Latenz verursachen kann. Sie können Auto Scaling deaktivieren, indem Sie denselben Wert für den minimalen und maximalen Ereignis-Poller konfigurieren.

Weitere Überlegungen

Zu den weiteren Überlegungen gehören:

- Kaltstarts durch den Aufruf Ihrer Lambda-Zielfunktion können möglicherweise die End-to-End-Latenz erhöhen. Um dieses Risiko zu verringern, sollten Sie die bereitgestellte Gleichzeitigkeit oder SnapStart für die Zielfunktion Ihrer Zuordnung von Ereignisquellen aktivieren. Optimieren Sie außerdem die Speicherzuweisung Ihrer Funktion, um eine konsistente und optimale Ausführung sicherzustellen.
- Wenn `MaximumBatchingWindowInSeconds` auf 0 gesetzt ist, verarbeitet Lambda alle verfügbaren Datensätze sofort, ohne auf das Erreichen der vollständigen Batchgröße zu warten. Wenn Ihre Batchgröße beispielsweise auf 1 000 Datensätze festgelegt ist, aber nur 100 Datensätze verfügbar sind, verarbeitet Lambda diese 100 Datensätze sofort, anstatt zu warten, bis die vollständigen 1 000 Datensätze gesammelt sind.

> **Important**
>
> Die optimale Konfiguration für eine Verarbeitung mit geringer Latenz hängt stark von Ihrer spezifischen Workload ab. Wir empfehlen dringend, verschiedene Konfigurationen mit Ihrer tatsächlichen Workload zu testen, um die besten Einstellungen für Ihren Anwendungsfall zu ermitteln.

Konfiguration der Fehlerbehandlungssteuerungen für Kafka-Ereignisquellen

Sie können konfigurieren, wie Lambda Fehler und Wiederholungen für Ihre Kafka-Ereignisquellenzuordnungen behandelt. Mit diesen Konfigurationen können Sie steuern, wie Lambda fehlgeschlagene Datensätze verarbeitet und das Wiederholungsverhalten verwaltet.

Verfügbare Wiederholungskonfigurationen

Die folgenden Wiederholungskonfigurationen sind sowohl für Amazon MSK als auch für selbstverwaltete Kafka-Ereignisquellen verfügbar:

- Maximale Wiederholungsversuche — Die maximale Anzahl von Wiederholungsversuchen von Lambda, wenn Ihre Funktion einen Fehler zurückgibt. Der erste Aufrufversuch wird dabei nicht mitgezählt. Die Standardeinstellung ist -1 (unendlich).
- Maximales Datensatzalter — Das maximale Alter eines Datensatzes, den Lambda an Ihre Funktion sendet. Die Standardeinstellung ist -1 (unendlich).
- Batch bei Fehler aufteilen — Wenn Ihre Funktion einen Fehler zurückgibt, teilen Sie den Batch in zwei kleinere Batches auf und versuchen Sie es jeweils einzeln erneut. Dies hilft, problematische Datensätze zu isolieren.
- Partielle Batch-Antwort — Erlauben Sie Ihrer Funktion, Informationen darüber zurückzugeben, bei welchen Datensätzen in einem Batch die Verarbeitung fehlgeschlagen ist, sodass Lambda nur die fehlgeschlagenen Datensätze erneut versuchen kann.

Konfiguration von Steuerelementen zur Fehlerbehandlung (Konsole)

Sie können das Wiederholungsverhalten konfigurieren, wenn Sie eine Kafka-Ereignisquellenzuordnung in der Lambda-Konsole erstellen oder aktualisieren.

So konfigurieren Sie das Wiederholungsverhalten für eine Kafka-Ereignisquelle (Konsole)

1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.
2. Wählen Sie den Namen Ihrer Funktion.
3. Führen Sie eine der folgenden Aktionen aus:
    - Um einen neuen Kafka-Trigger hinzuzufügen, wählen Sie unter Funktionsübersicht die Option Trigger hinzufügen aus.

- Um einen vorhandenen Kafka-Trigger zu ändern, wählen Sie den Auslöser aus und klicken Sie dann auf Bearbeiten.

4. Wählen Sie unter Konfiguration des Event-Pollers den Bereitstellungsmodus aus, um die Kontrollen zur Fehlerbehandlung zu konfigurieren:

   a. Geben Sie für Wiederholungsversuche die maximale Anzahl von Wiederholungsversuchen ein (0-10000 oder -1 für unendlich).

   b. Geben Sie für Maximales Datensatzalter das Höchstalter in Sekunden ein (60-604800 oder -1 für unendlich).

   c. Um die Stapelaufteilung bei Fehlern zu aktivieren, wählen Sie Batch bei Fehler aufteilen aus.

   d. Um die teilweise Batch-Antwort zu aktivieren, wählen Sie ReportBatchItemFailures.

5. Wählen Sie Hinzufügen oder Speichern.

Konfiguration des Wiederholungsverhaltens ()AWS CLI

Verwenden Sie die folgenden AWS CLI Befehle, um das Wiederholungsverhalten für Ihre Kafka-Ereignisquellenzuordnungen zu konfigurieren.

Erstellen einer Ereignisquellenzuordnung mit Wiederholungskonfigurationen

Im folgenden Beispiel wird eine selbstverwaltete Kafka-Ereignisquellenzuordnung mit Steuerelementen zur Fehlerbehandlung erstellt:

```
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"
```

Für Amazon MSK-Ereignisquellen:

```
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"
```

Wiederholungskonfigurationen werden aktualisiert

Verwenden Sie den `update-event-source-mapping` Befehl, um Wiederholungskonfigurationen für eine bestehende Ereignisquellenzuordnung zu ändern:

```
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

Die partielle Batch-Antwort, auch bekannt als ReportBatchItemFailures, ist eine wichtige Funktion für die Fehlerbehandlung bei der Integration von Lambda mit Kafka-Quellen. Ohne diese Funktion führt ein Fehler in einem der Elemente in einem Batch dazu, dass alle Nachrichten in diesem Batch erneut verarbeitet werden. Wenn die partielle Batch-Antwort aktiviert und implementiert ist, gibt der Handler nur Identifikatoren für die fehlgeschlagenen Nachrichten zurück, sodass Lambda nur diese spezifischen Elemente erneut versuchen kann. Dies ermöglicht eine bessere Kontrolle darüber, wie Batches mit fehlgeschlagenen Nachrichten verarbeitet werden.

Um Batchfehler zu melden, verwenden Sie dieses JSON-Schema:

```
{
  "batchItemFailures": [
    {
```

```
      "itemIdentifier": {
        "topic-partition": "topic-partition_number",
        "offset": 100
      }
    },
    ...
  ]
}
```

> **Important**
>
> Wenn Sie einen leeren, gültigen JSON-Code oder einen Nullwert zurückgeben, betrachtet die Zuordnung der Ereignisquelle einen Batch als erfolgreich verarbeitet. Jede zurückgegebene ungültige Topic-Partition_Number oder Offset, die im aufgerufenen Ereignis nicht enthalten waren, wird als Fehler behandelt, und der gesamte Batch wird erneut versucht.

Die folgenden Codebeispiele zeigen, wie eine partielle Batch-Antwort für Lambda-Funktionen implementiert wird, die Ereignisse aus Kafka-Quellen empfangen. Die Funktion meldet die Batch-Elementfehler in der Antwort und signalisiert Lambda, diese Nachrichten später erneut zu versuchen.

Hier ist eine Python-Lambda-Handler-Implementierung, die diesen Ansatz zeigt:

```
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
```

Hier ist eine Version von 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 };
```

Erfassen verworfener Batches für Amazon MSK und selbstverwaltete Apache-Kafka-Ereignisquellen

Um Datensätze zu fehlgeschlagenen Aufrufen zur Zuordnung von Ereignisquellen beizubehalten, fügen Sie der Zuordnung von Ereignisquellen Ihrer Funktion ein Ziel hinzu. Jeder an das Ziel gesendete Datensatz ist ein JSON-Dokument mit Metadaten über den fehlgeschlagenen Aufruf. Bei Amazon S3-Zielen sendet Lambda auch den gesamten Aufrufdatensatz zusammen mit den Metadaten. Sie können jedes Amazon SNS SNS-Thema, jede Amazon SQS SQS-Warteschlange, jeden Amazon S3 S3-Bucket oder Kafka als Ziel konfigurieren.

Bei Amazon-S3-Zielen können Sie das Feature Amazon S3 Event Notifications verwenden, um Benachrichtigungen zu erhalten, wenn Objekte in Ihren S3-Ziel-Bucket hochgeladen werden. Sie können S3-Ereignisbenachrichtigungen auch so konfigurieren, dass sie eine andere Lambda-Funktion aufrufen, um eine automatische Verarbeitung für fehlgeschlagene Stapel durchzuführen.

Ihre Ausführungsrolle muss über Berechtigungen für das Ziel verfügen:

- Für ein SQS-Ziel: sqs: SendMessage
- Für ein SNS-Ziel: sns:Publish
- Für ein S3-Ziel: s3: und s3: PutObject ListBucket
- Für ein Kafka-Ziel: kafka-cluster: WriteData

Sie können ein Kafka-Thema als Ziel für den Fall eines Fehlers für Ihre Quellenzuordnungen für Kafka-Ereignisse konfigurieren. Wenn Lambda Datensätze nach anstrengenden Wiederholungsversuchen nicht verarbeiten kann oder wenn Datensätze das Höchstalter überschreiten, sendet Lambda die fehlgeschlagenen Datensätze zur späteren Verarbeitung an das angegebene Kafka-Thema. Weitere Informationen finden Sie unter the section called “Kafka-Ziel bei einem Ausfall”.

Sie müssen einen VPC-Endpunkt für Ihren Fehlerziel-Service innerhalb Ihrer Kafka-Cluster-VPC bereitstellen.

Wenn Sie einen KMS-Schlüssel für Ihr Ziel konfiguriert haben, benötigt Lambda außerdem je nach Zieltyp folgende Berechtigungen:

- Wenn Sie die Verschlüsselung mit Ihrem eigenen KMS-Schlüssel für ein S3-Ziel aktiviert haben, ist kms: erforderlich. GenerateDataKey Wenn sich der KMS-Schlüssel und das S3-Bucket-Ziel in einem anderen Konto als Ihre Lambda-Funktion und Ausführungsrolle befinden,

konfigurieren Sie den KMS-Schlüssel so, dass er der Ausführungsrolle vertraut, die zugelassen kms: GenerateDataKey wird.

- Wenn Sie die Verschlüsselung mit Ihrem eigenen KMS-Schlüssel für das SQS-Ziel aktiviert haben, sind kms:Decrypt und kms: erforderlich. GenerateDataKey Wenn sich der KMS-Schlüssel und das SQS-Warteschlangenziel in einem anderen Konto als Ihre Lambda-Funktion und Ausführungsrolle befinden, konfigurieren Sie den KMS-Schlüssel so, dass er der Ausführungsrolle vertraut, um kms: Decrypt, kms:GenerateDataKey, kms: und kms: DescribeKey zuzulassen. ReEncrypt
- Wenn Sie die Verschlüsselung mit Ihrem eigenen KMS-Schlüssel für das SNS-Ziel aktiviert haben, sind kms:Decrypt und kms: erforderlich. GenerateDataKey Wenn sich der KMS-Schlüssel und das SNS-Themenziel in einem anderen Konto als Ihre Lambda-Funktion und Ausführungsrolle befinden, konfigurieren Sie den KMS-Schlüssel so, dass er der Ausführungsrolle vertraut, um kms: Decrypt, kms:GenerateDataKey, kms: und kms: DescribeKey zuzulassen. ReEncrypt

Konfigurieren von Fehlerzielen für eine Kafka-Zuordnung von Ereignisquellen

Gehen Sie folgendermaßen vor, um ein Ausfallziel mit der Konsole zu konfigurieren:

1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.
2. Wählen Sie eine Funktion aus.
3. Wählen Sie unter Function overview (Funktionsübersicht) die Option Add destination (Ziel hinzufügen).
4. Wählen Sie als Quelle die Option Aufruf der Zuordnung von Ereignisquellen aus.
5. Wählen Sie für die Zuordnung von Ereignisquellen eine Ereignisquelle aus, die für diese Funktion konfiguriert ist.
6. Wählen Sie für Bedingung die Option Bei Ausfall aus. Für Aufrufe zur Zuordnung von Ereignisquellen ist dies die einzig akzeptierte Bedingung.
7. Wählen Sie unter Zieltyp den Zieltyp aus, an den Lambda Aufrufdatensätze sendet.
8. Wählen Sie unter Destination (Ziel) eine Ressource aus.
9. Wählen Sie Speichern.

Sie können mit der AWS CLI auch ein Ausfallziel konfigurieren. Mit dem folgenden create-event-source-mappingBefehl wird beispielsweise eine Ereignisquellenzuordnung mit einem SQS-Ziel für den Fall eines Fehlers hinzugefügt: `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"}}'
```

Mit dem folgenden update-event-source-mappingBefehl wird der mit der Eingabe verknüpften Ereignisquelle ein S3-Ziel für den Fall eines Fehlers hinzugefügt: `uuid`

```
aws lambda update-event-source-mapping \
--uuid f89f8514-cdd9-4602-9e1f-01a5b77d449b \
--destination-config '{"OnFailure": {"Destination": "arn:aws:s3:::dest-bucket"}}'
```

Um ein Ziel zu entfernen, geben Sie eine leere Zeichenfolge als Argument für den `destination-config`-Parameter an:

```
aws lambda update-event-source-mapping \
--uuid f89f8514-cdd9-4602-9e1f-01a5b77d449b \
--destination-config '{"OnFailure": {"Destination": ""}}'
```

Bewährte Methoden für die Sicherheit in Amazon S3-Zielen

Das Löschen eines S3-Buckets, der als Ziel konfiguriert ist, ohne das Ziel aus der Konfiguration Ihrer Funktion zu entfernen, kann ein Sicherheitsrisiko darstellen. Wenn ein anderer Benutzer den Namen Ihres Ziel-Buckets kennt, kann er den Bucket in seinem AWS-Konto neu erstellen. Aufzeichnungen über fehlgeschlagene Aufrufe werden an den entsprechenden Bucket gesendet, wodurch möglicherweise Daten aus Ihrer Funktion verfügbar gemacht werden.

> **Warning**
>
> Um sicherzustellen, dass Aufrufdatensätze Ihrer Funktion nicht an einen S3-Bucket in einem anderen gesendet werden können AWS-Konto, fügen Sie der Ausführungsrolle Ihrer Funktion eine Bedingung hinzu, die die `s3:PutObject` Berechtigungen auf Buckets in Ihrem Konto beschränkt.

-Das folgende Beispiel zeigt eine IAM-Richtlinie, die die `s3:PutObject`-Berechtigungen Ihrer Funktion auf Buckets in Ihrem Konto beschränkt. Diese Richtlinie gibt Lambda auch die `s3:ListBucket`-Erlaubnis, einen S3-Bucket als Ziel zu verwenden.

```
{
    "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"
                }
            }
        }
    ]
}
```

Um der Ausführungsrolle Ihrer Funktion mithilfe von AWS-Managementkonsole oder eine Berechtigungsrichtlinie hinzuzufügen AWS CLI, lesen Sie die Anweisungen in den folgenden Verfahren:

Console

So fügen Sie der Ausführungsrolle einer Funktion (Konsole) eine Berechtigungsrichtlinie hinzu

1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.
2. Wählen Sie die Lambda-Funktion aus, deren Ausführungsrolle Sie ändern möchten.
3. Klicken Sie in der Registerkarte Konfiguration auf die Option Berechtigungen.
4. Wählen Sie auf der Registerkarte Ausführungsrolle den Rollennamen Ihrer Funktion aus, um die IAM-Konsolenseite der Rolle zu öffnen.
5. Fügen Sie der Rolle wie folgt eine Richtlinie mit Berechtigungen hinzu:

    a. Wählen Sie im Bereich Berechtigungsrichtlinien die Optionen Berechtigungen hinzufügen und dann Inline-Richtlinie erstellen aus.

b. Wählen Sie im Richtlinien-Editor JSON aus.

c. Fügen Sie die Richtlinie, die Sie hinzufügen möchten, in den Editor ein (indem Sie die vorhandene JSON-Datei ersetzt) und wählen Sie dann Weiter aus.

d. Geben Sie unter Richtliniendetails für den Richtliniennamen ein.

e. Wählen Sie Richtlinie erstellen aus.

AWS CLI

So fügen Sie der Ausführungsrolle einer Funktion (CLI) eine Berechtigungsrichtlinie hinzu

1. Erstellen Sie ein JSON-Richtliniendokument mit den erforderlichen Berechtigungen und speichern Sie es in einem lokalen Verzeichnis.
2. Verwenden Sie den `put-role-policy` IAM-CLI-Befehl, um die Berechtigungen zur Ausführungsrolle Ihrer Funktion hinzuzufügen. Führen Sie den folgenden Befehl in dem Verzeichnis aus, in dem Sie Ihr JSON-Richtliniendokument gespeichert haben und ersetzen Sie den Rollennamen, den Richtliniennamen und das Richtliniendokument durch Ihre eigenen Werte.

```
aws iam put-role-policy \
--role-name my_lambda_role \
--policy-name LambdaS3DestinationPolicy \
--policy-document file://my_policy.json
```

SNS- und SQS-Beispiel-Aufrufsatz

Das folgende Beispiel zeigt, was Lambda bei einem fehlgeschlagenen Aufruf der Kafka-Ereignisquelle an ein SNS-Thema oder eine SQS-Warteschlange sendet. Jeder der Schlüssel unter `recordsInfo` enthält sowohl das Kafka-Thema als auch die Kafka-Partition, getrennt durch einen Bindestrich. Bei dem Schlüssel `"Topic-0"` handelt es sich beispielsweise bei `Topic` um das Kafka-Thema und bei `0` um die Partition. Für jedes Thema und jede Partition können Sie die Offsets und Zeitstempeldaten verwenden, um die ursprünglichen Aufrufdatensätze zu finden.

```
{
    "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",
            }
        }
    }
}
```

S3-Ziel Beispiel-Aufrufsatz

Für S3-Ziele sendet Lambda den gesamten Aufrufdatensatz zusammen mit den Metadaten an das Ziel. Das folgende Beispiel zeigt, was Lambda bei einem fehlgeschlagenen Aufruf der Kafka-Ereignisquelle an ein S3-Bucket-Ziel sendet. Zusätzlich zu allen Feldern aus dem vorherigen Beispiel für SQS- und SNS-Ziele enthält das Feld `payload` den ursprünglichen Aufrufdatensatz als maskierte JSON-Zeichenfolge.

```
{
    "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**
>
> Wir empfehlen außerdem, die S3-Versionsverwaltung in Ihrem Ziel-Bucket zu aktivieren.

Ein Kafka-Thema als Ziel für den Fall eines Fehlers verwenden

Sie können ein Kafka-Thema als Ziel für den Fall eines Fehlers für Ihre Quellenzuordnungen für Kafka-Ereignisse konfigurieren. Wenn Lambda Datensätze nach anstrengenden Wiederholungsversuchen nicht verarbeiten kann oder wenn Datensätze das Höchstalter überschreiten, sendet Lambda die fehlgeschlagenen Datensätze zur späteren Verarbeitung an das angegebene Kafka-Thema.

So funktioniert ein Kafka-Ziel für den Fall eines Fehlers

Wenn Sie ein Kafka-Thema als Ziel für den Fall eines Fehlers konfigurieren, fungiert Lambda als Kafka-Producer und schreibt fehlgeschlagene Datensätze in das Zielthema. Auf diese Weise entsteht in Ihrer Kafka-Infrastruktur ein Muster mit veralteten Themen (DLT).

- Gleiche Clusteranforderung — Das Zielthema muss im selben Kafka-Cluster wie Ihre Quellthemen vorhanden sein.
- Tatsächlicher Datensatzinhalt — Kafka-Ziele erhalten die tatsächlich fehlgeschlagenen Datensätze zusammen mit den Fehlermetadaten.
- Rekursionsprävention — Lambda verhindert Endlosschleifen, indem es Konfigurationen blockiert, bei denen Quell- und Zielthemen identisch sind.

Konfiguration eines Kafka-Ziels für den Fall eines Fehlers

Sie können ein Kafka-Thema als Ziel für den Fall eines Fehlers konfigurieren, wenn Sie eine Kafka-Ereignisquellenzuordnung erstellen oder aktualisieren.

Konfiguration eines Kafka-Ziels (Konsole)

Um ein Kafka-Thema als Ziel für den Fall eines Fehlers zu konfigurieren (Konsole)

1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.
2. Wählen Sie den Namen Ihrer Funktion.
3. Führen Sie eine der folgenden Aktionen aus:

    - Um einen neuen Kafka-Trigger hinzuzufügen, wählen Sie unter Funktionsübersicht die Option Trigger hinzufügen aus.
    - Um einen vorhandenen Kafka-Trigger zu ändern, wählen Sie den Auslöser aus und klicken Sie dann auf Bearbeiten.

4. Wählen Sie unter Zusätzliche Einstellungen für Ziel bei einem Fehler die Option Kafka-Thema aus.
5. Geben Sie als Themenname den Namen des Kafka-Themas ein, an das Sie fehlerhafte Datensätze senden möchten.
6. Wählen Sie Hinzufügen oder Speichern.

Konfiguration eines Kafka-Ziels ()AWS CLI

Verwenden Sie das `kafka://` Präfix, um ein Kafka-Thema als Ziel für den Fall eines Fehlers anzugeben.

Erstellen einer Ereignisquellenzuordnung mit einem Kafka-Ziel

Im folgenden Beispiel wird eine Amazon MSK-Ereignisquellenzuordnung mit einem Kafka-Thema als Ziel für den Fall eines Fehlers erstellt:

```
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"}}'
```

Verwenden Sie für selbstverwaltetes Kafka dieselbe Syntax:

```
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"}}'
```

Aktualisierung einer Kafka-Destination

Verwenden Sie den `update-event-source-mapping` Befehl, um ein Kafka-Ziel hinzuzufügen oder zu ändern:

```
aws lambda update-event-source-mapping \
```

```
  --uuid 12345678-1234-1234-1234-123456789012 \
  --destination-config '{"OnFailure":{"Destination":"kafka://failed-records-topic"}}'
```

Datensatzformat für ein Kafka-Ziel

Wenn Lambda fehlgeschlagene Datensätze an ein Kafka-Thema sendet, enthält jede Nachricht sowohl Metadaten über den Fehler als auch den eigentlichen Datensatzinhalt.

Metadaten für den Fehler

Die Metadaten enthalten Informationen darüber, warum der Datensatz fehlgeschlagen ist, und Details zum ursprünglichen Batch:

```
{
  "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=="
        }
      ]
    }
  }
}
```

Verhalten der Partitionsschlüssel

Lambda verwendet beim Produzieren zum Zielthema denselben Partitionsschlüssel wie im Originaldatensatz. Wenn der ursprüngliche Datensatz keinen Schlüssel hatte, verwendet Lambda die standardmäßige Round-Robin-Partitionierung von Kafka für alle verfügbaren Partitionen im Zielthema.

Anforderungen und Einschränkungen

- Bereitgestellter Modus erforderlich — Ein Kafka-Ziel für den Fall eines Fehlers ist nur für Zuordnungen von Ereignisquellen verfügbar, bei denen der Bereitstellungsmodus aktiviert ist.
- Nur derselbe Cluster — Das Zielthema muss im selben Kafka-Cluster wie Ihre Quellthemen vorhanden sein.
- Themenberechtigungen — Ihre Zuordnung zur Ereignisquelle muss über Schreibberechtigungen für das Zielthema verfügen. Beispiel:

```
{
    "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": "*"
        }
    ]
}
```

- Keine Rekursion — Der Name des Zielthemas darf nicht mit einem Ihrer Quellthemennamen identisch sein.

Fehlerbehebung bei der Kafka-Zuordnung von Ereignisquellen

Die folgenden Themen enthalten Hinweise zur Fehlerbehebung für Fehler und Probleme, die bei der Verwendung von Amazon MSK oder selbstverwaltetem Apache Kafka mit Lambda auftreten können.

Weitere Hilfe zur Fehlerbehebung finden Sie im AWS-Wissenszentrum.

Authentifizierungs- und Autorisierungsfehler

Wenn eine der Berechtigungen, die zur Nutzung von Daten aus dem Kafka-Cluster erforderlich sind, fehlt, zeigt Lambda eine der folgenden Fehlermeldungen in der Ereignisquellenzuordnung unter LastProcessingResult an.

Fehlermeldungen

- Cluster konnte Lambda nicht autorisieren
- SASL-Authentifizierung fehlgeschlagen
- Server konnte Lambda nicht authentifizieren
- Lambda konnte Server nicht authentifizieren
- Bereitgestelltes Zertifikat oder bereitgestellter privater Schlüssel ist ungültig

Cluster konnte Lambda nicht autorisieren

Bei SASL/SCRAM oder mTLS weist dieser Fehler darauf hin, dass der angegebene Benutzer nicht über alle nachfolgenden erforderlichen Berechtigungen für die Kafka-Zugriffskontrollliste (ACL) verfügt:

- DescribeConfigs-Cluster
- Beschreiben von Gruppe
- Gruppe lesen
- Thema beschreiben
- Thema lesen

Wenn Sie Kafka-ACLs mit den erforderlichen `kafka-cluster`-Berechtigungen erstellen, geben Sie das Thema und die Gruppe als Ressourcen an. Der Themenname muss mit dem Thema in der Ereignisquellenzuordnung übereinstimmen. Der Gruppenname muss mit der UUID der Ereignisquellenzuordnung übereinstimmen.

Nachdem Sie der Ausführungsrolle die erforderlichen Berechtigungen hinzugefügt haben, kann es einige Minuten dauern, bis die Änderungen wirksam werden.

SASL-Authentifizierung fehlgeschlagen

Bei SASL/SCRAM oder SASL/PLAIN weist dieser Fehler darauf hin, dass die angegebenen Anmeldeinformationen ungültig sind.

Zur IAM-Zugriffskontrolle fehlt der Ausführungsrolle die Berechtigung `kafka-cluster:Connect` für den Cluster. Fügen Sie der Rolle diese Berechtigung hinzu und geben Sie den Amazon-Ressourcenname (ARN) des Clusters als Ressource an.

Dieser Fehler wird Ihnen möglicherweise in zeitlichen Abständen angezeigt. Der Cluster lehnt Verbindungen ab, wenn die Anzahl der TCP-Verbindungen das Servicekontingent überschreitet. Lambda zieht sich zurück und versucht es erneut, bis eine Verbindung erfolgreich ist. Nachdem Lambda eine Verbindung zum Cluster hergestellt hat und nach Datensätzen abfragt, ändert sich das letzte Verarbeitungsergebnis zu `OK`.

Server konnte Lambda nicht authentifizieren

Dieser Fehler weist darauf hin, dass der Kafka-Broker Lambda nicht authentifizieren konnte. Dieser Fehler kann aus folgenden Gründen auftreten:

- Sie haben kein Client-Zertifikat für die mTLS-Authentifizierung bereitgestellt.
- Sie haben ein Client-Zertifikat bereitgestellt, aber die Kafka-Broker sind nicht für die Verwendung der mTLS-Authentifizierung konfiguriert.
- Die Kafka-Broker vertrauen einem Client-Zertifikat nicht.

Lambda konnte Server nicht authentifizieren

Dieser Fehler weist darauf hin, dass Lambda den Kafka-Broker nicht authentifizieren konnte. Dieser Fehler kann aus folgenden Gründen auftreten:

- Für selbstverwaltetes Apache Kafka: Die Kafka-Broker verwenden selbstsignierte Zertifikate oder eine private Zertifizierungsstelle (CA), haben jedoch das Root-CA-Zertifikat des Servers nicht bereitgestellt.
- Für selbstverwaltetes Apache Kafka: Das Root-CA-Zertifikat des Servers stimmt nicht mit der Root-CA überein, die das Zertifikat des Brokers signiert hat.
- Die Überprüfung des Hostnamens ist fehlgeschlagen, weil das Zertifikat des Brokers nicht den DNS-Namen oder die IP-Adresse des Brokers als alternativen Betreffnamen enthält.

Bereitgestelltes Zertifikat oder bereitgestellter privater Schlüssel ist ungültig

Dieser Fehler weist darauf hin, dass der Kafka-Konsument das bereitgestellte Zertifikat oder den bereitgestellten privaten Schlüssel nicht verwenden konnte. Stellen Sie sicher, dass das Zertifikat und der Schlüssel das PEM-Format haben und die Verschlüsselung des privaten Schlüssels einen PBES1-Algorithmus nutzt.

Fehler bei der Ereignisquellen-Zuweisung

Wenn Sie Ihren Apache-Kafka-Cluster als Ereignisquelle für Ihre Lambda-Funktion hinzufügen und Ihre Funktion auf einen Fehler stößt, beendet Ihr Kafka-Verbraucher die Verarbeitung von Datensätzen. Verbraucher einer Themenpartition sind diejenigen, die Ihre Datensätze abonnieren, lesen und verarbeiten. Ihre anderen Kafka-Verbraucher können weiterhin Datensätze verarbeiten, sofern sie nicht auf denselben Fehler stoßen.

Um die Ursache eines gestoppten Verbrauchers zu ermitteln, überprüfen Sie das `StateTransitionReason`-Feld in der Antwort von `EventSourceMapping`. In der folgenden Liste werden die Ereignisquellfehler beschrieben, die Sie erhalten können:

`ESM_CONFIG_NOT_VALID`

Die Konfiguration der Ereignisquellenzuordnung ist ungültig.

`EVENT_SOURCE_AUTHN_ERROR`

Lambda konnte die Ereignisquelle nicht authentifizieren.

`EVENT_SOURCE_AUTHZ_ERROR`

Lambda verfügt nicht über die erforderlichen Berechtigungen für den Zugriff auf die Ereignisquelle.

`FUNCTION_CONFIG_NOT_VALID`

Die Konfiguration der Funktion ist ungültig.

> **Note**
>
> Wenn Ihre Lambda-Ereignisdatensätze die zulässige Größenbeschränkung von 6 MB überschreiten, können sie unbearbeitet bleiben.

Aufrufen einer Lambda-Funktion über einen Amazon API Gateway-Endpunkt

Sie können eine Web-API mit einem HTTP-Endpunkt für Ihre Lambda Funktion erstellen, indem Sie Amazon API Gateway verwenden. API Gateway bietet Tools zum Erstellen und Dokumentieren von Websites APIs , die HTTP-Anfragen an Lambda-Funktionen weiterleiten. Sie können den Zugriff auf Ihre API mit Authentifizierungs- und Autorisierungskontrollen sichern. Sie APIs können Traffic über das Internet bereitstellen oder nur innerhalb Ihrer VPC darauf zugreifen.

> **Tip**
>
> Lambda bietet zwei Möglichkeiten, Ihre Funktion über einen HTTP-Endpunkt aufzurufen: API Gateway und Lambda-Funktion. URLs Wenn Sie sich nicht sicher sind, welche Methode für Ihren Anwendungsfall am besten geeignet ist, siehe the section called “API Gateway im Vergleich zur Funktion URLs”.

Ressourcen in Ihrer API definieren mindestens eine Methode, z. B. GET oder POST. Methoden haben eine Integration, die Anfragen an eine Lambda-Funktion oder einen anderen Integrationstyp weiterleitet. Sie können jede Ressource und jede Methode einzeln definieren oder spezielle Ressourcen- und Methodenarten verwenden, um alle Anforderungen abzugleichen, die einem Muster entsprechen. Eine Proxy-Ressource fängt alle Pfade unter einer Ressource ab. Die `ANY`-Methode fängt alle HTTP-Methoden ab.

Sections

- Auswählen eines API-Typs
- Hinzufügen eines Endpunkts zur Lambda-Funktion
- Proxy-Integration
- Ereignisformat
- Reaktionsformat
- Berechtigungen
- Beispielanwendung
- Der Event-Handler von Powertools für Lambda AWS
- Tutorial: Verwenden von Lambda mit API Gateway

- Behandlung von Lambda-Fehlern mit einer API-Gateway-API
- Wählen Sie eine Methode, um Ihre Lambda-Funktion über eine HTTP-Anfrage aufzurufen

Auswählen eines API-Typs

API Gateway unterstützt drei Typen von APIs Lambda-Funktionen:

- HTTP-API: Eine schlanke API mit niedriger RESTful Latenz.
- REST-API: Eine anpassbare, funktionsreiche RESTful API.
- WebSocket API: Eine Web-API, die persistente Verbindungen zu Clients für die Vollduplex-Kommunikation unterhält.

APIs Sowohl HTTP als APIs RESTful APIs auch REST verarbeiten HTTP-Anfragen und geben Antworten zurück. HTTP APIs sind neuer und wurden mit der API Gateway Version 2 API erstellt. Die folgenden Funktionen sind neu für HTTP APIs:

HTTP-API-Features

- Automatische Bereitstellungen – wenn Sie Routen oder Integrationen ändern, werden Änderungen automatisch in Phasen bereitgestellt, bei denen die automatische Bereitstellung aktiviert ist.
- Standardphase – Sie können eine Standardphase (`$default`) erstellen, um Anforderungen am Stammpfad der URL Ihrer API zu bedienen. Bei benannten Phasen müssen Sie den Schrittnamen am Anfang des Pfades angeben.
- CORS-Konfiguration – Sie können Ihre API so konfigurieren, dass ausgehenden Antworten CORS-Header hinzugefügt werden, anstatt sie manuell in Ihrem Funktionscode hinzufügen zu müssen.

REST APIs sind der Klassiker RESTful APIs , den API Gateway seit dem Start unterstützt. REST bietet APIs derzeit mehr Anpassungs-, Integrations- und Verwaltungsfunktionen.

REST-API-Funktionen

- Integrationstypen — REST APIs unterstützt benutzerdefinierte Lambda-Integrationen. Mit einer benutzerdefinierten Integration können Sie nur den Text der Anforderung an die Funktion senden oder eine Transformationsvorlage auf den Anforderungstext anwenden, bevor Sie sie an die Funktion senden.
- Zugriffskontrolle — REST APIs unterstützt mehr Optionen für Authentifizierung und Autorisierung.

- Überwachung und Ablaufverfolgung — REST APIs unterstützt AWS X-Ray Tracing und zusätzliche Protokollierungsoptionen.

Einen ausführlichen Vergleich finden Sie unter Wählen Sie zwischen HTTP APIs und REST APIs im API Gateway Developer Guide.

WebSocket APIs Verwenden Sie auch die API Gateway Version 2 API und unterstützen Sie einen ähnlichen Funktionsumfang. Verwenden Sie eine WebSocket API für Anwendungen, die von einer dauerhaften Verbindung zwischen dem Client und der API profitieren. WebSocket APIs bieten Vollduplex-Kommunikation, was bedeutet, dass sowohl der Client als auch die API kontinuierlich Nachrichten senden können, ohne auf eine Antwort warten zu müssen.

HTTP APIs unterstützt ein vereinfachtes Ereignisformat (Version 2.0). Ein Beispiel für ein Ereignis aus einer HTTP-API finden Sie unter Erstellen von AWS Lambda Proxyintegrationen für HTTP APIs in API Gateway.

Weitere Informationen finden Sie unter AWS Lambda Proxyintegrationen für HTTP APIs in API Gateway erstellen.

Hinzufügen eines Endpunkts zur Lambda-Funktion

So fügen Sie Ihrer Lambda-Funktion einen öffentlichen Endpunkt hinzu

1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.
2. Wählen Sie eine Funktion aus.
3. Wählen Sie unter Function overview (Funktionsübersicht) die Option Add trigger (Trigger hinzufügen).
4. Wählen Sie API Gateway aus.
5. Wählen Sie Create an API (API erstellen) oder Use an existing API (Vorhandene API verwenden).

    a. Neue API: Wählen Sie als API type (API-Typ) HTTP API aus. Weitere Informationen finden Sie unter Auswählen eines API-Typs.

    b. Vorhandene API: Wählen Sie die API aus der Dropdown-Liste aus oder geben Sie die API-ID ein (z. B. r3pmxmplak).
6. Wählen Sie unter Security (Sicherheit) die Option Open (Öffnen) aus.
7. Wählen Sie Add aus.

Proxy-Integration

API Gateway APIs besteht aus Stufen, Ressourcen, Methoden und Integrationen. Die Phase und die Ressource bestimmen den Pfad des Endpunkts:

API-Pfadformat

- `/prod/` – Die `prod`-Phase und die Root-Ressource.
- `/prod/user` – Die `prod`-Phase und die `user`-Ressource.
- `/dev/{proxy+}` – Jede Route in der `dev`-Phase.
- `/` — (HTTP APIs) Die Standard-Stage- und Root-Ressource.

Eine Lambda-Integration ordnet einer Lambda-Funktion eine Pfad- und HTTP-Methodenkombination zu. Sie können API Gateway so konfigurieren, dass der Hauptteil der HTTP-Anforderung unverändert (benutzerdefinierte Integration) übergeben oder der Anforderungstext in einem Dokument mit allen Anforderungsinformationen, einschließlich Header, Ressource, Pfad und Methode, gekapselt wird.

Weitere Informationen finden Sie unter Lambda-Proxy-Integrationen in API Gateway.

Ereignisformat

Amazon API Gateway ruft Ihre Funktion synchron mit einem Ereignis auf, das eine JSON-Darstellung der HTTP-Anforderung enthält. Bei einer benutzerdefinierten Integration ist das Ereignis der Text der Anforderung. Bei einer Proxy-Integration hat das Ereignis eine definierte Struktur. Ein Beispiel für ein Proxy-Ereignis von einer API Gateway REST API finden Sie unter Eingabeformat einer Lambda-Funktion für die Proxy-Integration im API Gateway Developer Guide.

Reaktionsformat

API Gateway wartet auf eine Antwort von Ihrer Funktion und leitet das Ergebnis an den Aufrufer weiter. Für eine benutzerdefinierte Integration definieren Sie eine Integrationsantwort und eine Methodenantwort, um die Ausgabe von der Funktion in eine HTTP-Antwort zu konvertieren. Für eine Proxy-Integration muss die Funktion mit einer Darstellung der Antwort in einem bestimmten Format antworten.

Das folgende Beispiel zeigt ein Antwortobjekt aus einer Node.js-Funktion. Das Antwortobjekt stellt eine erfolgreiche HTTP-Antwort dar, die ein JSON-Dokument enthält.

Example index.mjs – Antwortobjekt der Proxy-Integration (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}"
    }
```

Die Lambda-Laufzeit serialisiert das Antwortobjekt in JSON und sendet es an die API. Die API analysiert die Antwort und verwendet sie zur Erstellung einer HTTP-Antwort verwendet, die sie dann an den Client sendet, der die ursprüngliche Anforderung gestellt hat.

Example HTTP-Antwort

```
< 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
  }
```

Berechtigungen

Amazon API Gateway erhält die Berechtigung zum Aufrufen Ihrer Funktion über die ressourcenbasierte Richtlinie der Funktion. Sie können eine Aufrufberechtigung für eine gesamte API erteilen oder einen eingeschränkten Zugriff auf eine Phase, eine Ressource oder eine Methode gewähren.

Wenn Sie Ihrer Funktion mithilfe der Lambda-Konsole, mithilfe der API-Gateway-Konsole oder in einer AWS SAM -Vorlage eine API hinzufügen, wird die ressourcenbasierte Richtlinie der Funktion automatisch aktualisiert. Es folgt eine Beispiel-Funktionsrichtlinie.

Example Funktionsrichtlinie

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/"
        }
      }
    }
  ]
}
```

Sie können Funktionsrichtlinienberechtigungen manuell mit den folgenden API-Operationen verwalten:

- AddPermission
- RemovePermission
- GetPolicy

Mit dem `add-permission`-Befehl können Sie einer vorhandenen API Aufrufberechtigung erteilen. Beispiel:

```
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/"
```

Die Ausgabe sollte folgendermaßen aussehen:

```
{
    "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**
>
> Wenn sich Ihre Funktion und API unterscheiden AWS-Regionen, muss die Regionskennung im Quell-ARN mit der Region der Funktion übereinstimmen, nicht mit der Region der API. Bei Aufruf einer API-Gateway-Funktion verwendet diese einen Ressourcen-ARN, der auf dem ARN der API basiert, aber so geändert wird, dass er dem Bereich der Funktion entspricht.

Der Quell-ARN in diesem Beispiel erteilt eine Berechtigung für eine Integration in die GET-Methode der Root-Ressource in der Standardphase einer API mit ID `mnh1xmpli7`. Sie können ein Sternchen im Quell-ARN verwenden, um Berechtigungen für mehrere Phasen, Methoden oder Ressourcen zu erteilen.

Ressourcenmuster

- `mnh1xmpli7/*/GET/*` – GET-Methode bei allen Ressourcen in allen Phasen.
- `mnh1xmpli7/prod/ANY/user` – JEDE Methode bei der `user`-Ressource in der `prod`-Phase.
- `mnh1xmpli7/*/*/*` – Jede Methode bei allen Ressourcen in allen Phasen.

Weitere Informationen zum Anzeigen der Richtlinie und zum Entfernen von Anweisungen finden Sie unter Anzeigen von ressourcenbasierten IAM-Richtlinien in Lambda.

Beispielanwendung

Die Beispiel-App API Gateway with Node.js enthält eine Funktion mit einer AWS SAM Vorlage, die eine REST-API erstellt, für die die AWS X-Ray Ablaufverfolgung aktiviert ist. Es enthält auch Skripte zum Bereitstellen und Aufrufen der Funktion sowie zum Testen der API und zum Bereinigen.

Der Event-Handler von Powertools für Lambda AWS

Der Event-Handler aus dem Powertools for AWS Lambda-Toolkit bietet Routing, Middleware, CORS-Konfiguration, Generierung von OpenAPI-Spezifikationen, Anforderungsvalidierung, Fehlerbehandlung und andere nützliche Funktionen beim Schreiben von Lambda-Funktionen, die von einem API-Gateway-Endpunkt (HTTP oder REST) aufgerufen werden. Das Event-Handler-Hilfsprogramm ist für Python TypeScript und/ verfügbarJavaScript. Weitere Informationen finden Sie unter Event Handler REST API in der Powertools for AWS Lambda (Python) -Dokumentation und Event Handler REST API in der Powertools for AWS Lambda () -Dokumentation. 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)
```

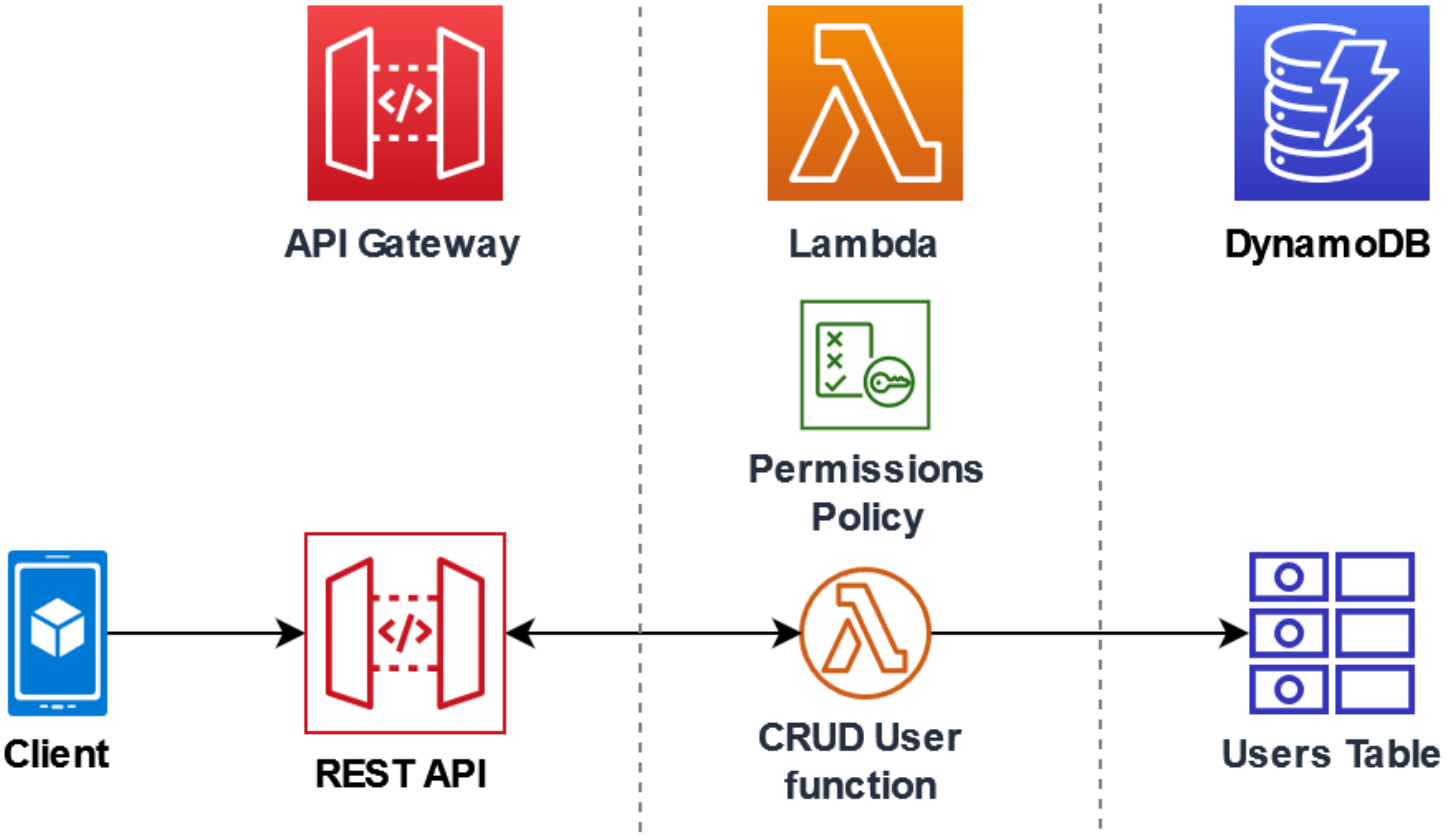
TypeScript

```
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);
};
```

Tutorial: Verwenden von Lambda mit API Gateway

In diesem Tutorial erstellen Sie eine REST-API, über die Sie eine Lambda-Funktion mithilfe einer HTTP-Anfrage aufrufen. Ihre Lambda-Funktion führt CRUD-Operationen (Erstellen, Lesen, Aktualisieren und Löschen) für eine DynamoDB-Tabelle durch. Diese Funktion wird hier zu Demonstrationszwecken bereitgestellt, aber Sie werden lernen, eine API-Gateway-REST-API zu konfigurieren, die jede Lambda-Funktion aufrufen kann.



Die Verwendung von API-Gateway bietet Benutzern einen sicheren HTTP-Endpunkt zum Aufrufen Ihrer Lambda-Funktion und kann dabei helfen, große Mengen an Aufrufen an Ihre Funktion zu verwalten, indem der Datenverkehr gedrosselt und API-Aufrufe automatisch validiert und autorisiert werden. API Gateway bietet auch flexible Sicherheitskontrollen mithilfe von AWS Identity and Access Management (IAM) und Amazon Cognito. Dies ist nützlich für Anwendungsfälle, in denen eine vorherige Autorisierung für Aufrufe zu Ihrer Anwendung erforderlich ist.

> **Tip**
>
> Lambda bietet zwei Möglichkeiten, Ihre Funktion über einen HTTP-Endpunkt aufzurufen: API Gateway und Lambda-Funktion. URLs Wenn Sie sich nicht sicher sind, welche Methode für Ihren Anwendungsfall am besten geeignet ist, siehe the section called “API Gateway im Vergleich zur Funktion URLs”.

Um dieses Tutorial abzuschließen, werden Sie die folgenden Phasen durchlaufen:

1. Erstellen und Konfigurieren einer Lambda-Funktion in Python oder Node.js, um Operationen in einer DynamoDB-Tabelle auszuführen.

2. Erstellen einer REST-API in API Gateway, um eine Verbindung zu Ihrer Lambda-Funktion herzustellen.
3. Erstellen einer DynamoDB-Tabelle und Testen dieser Tabelle mit Ihrer Lambda-Funktion in der Konsole.
4. Bereitstellen Ihrer API und Testen der vollständigen Einrichtung mithilfe von curl in einem Terminal.

Durch Abschluss dieser Phasen lernen Sie, wie Sie mit API Gateway einen HTTP-Endpunkt erstellen, der eine Lambda-Funktion in jeder Größenordnung sicher aufrufen kann. Sie erfahren auch, wie Sie Ihre API bereitstellen und wie Sie sie in der Konsole und durch Senden einer HTTP-Anfrage über ein Terminal testen.

Erstellen einer Berechtigungsrichtlinie

Bevor Sie eine Ausführungsrolle für Ihre Lambda-Funktion erstellen können, müssen Sie zunächst eine Berechtigungsrichtlinie erstellen, um Ihrer Funktion die Erlaubnis zu erteilen, auf die erforderlichen AWS Ressourcen zuzugreifen. In diesem Tutorial ermöglicht die Richtlinie Lambda, CRUD-Operationen in einer DynamoDB-Tabelle durchzuführen und in Amazon Logs zu schreiben. CloudWatch

So erstellen Sie die Richtlinie

1. Öffnen Sie die Seite Richtlinien in der IAM-Konsole.
2. Wählen Sie Richtlinie erstellen aus.
3. Wählen Sie die Registerkarte JSON aus und kopieren Sie dann die folgende benutzerdefinierte JSON-Richtlinie in den JSON-Editor.

   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. Wählen Sie Next: Tags (Weiter: Tags) aus.
5. Klicken Sie auf Weiter: Prüfen.
6. Geben Sie unter Review policy (Richtlinie prüfen) für den Richtlinien-Namen **`lambda-apigateway-policy`** ein.
7. Wählen Sie Richtlinie erstellen aus.

Erstellen einer Ausführungsrolle

Eine Ausführungsrolle ist eine AWS Identity and Access Management (IAM-) Rolle, die einer Lambda-Funktion Zugriff AWS-Services und Ressourcen gewährt. Damit Ihre Funktion Vorgänge an einer DynamoDB-Tabelle ausführen kann, fügen Sie die Berechtigungsrichtlinie an, die Sie im vorherigen Schritt erstellt haben.

So erstellen Sie eine Ausführungsrolle und fügen Ihre benutzerdefinierte Berechtigungsrichtlinie hinzu

1. Öffnen Sie die Seite Roles (Rollen) in der IAM-Konsole.
2. Wählen Sie Rolle erstellen aus.
3. Wählen Sie als Typ der vertrauenswürdigen Entität AWS -Service und dann als Anwendungsfall Lambda aus.

4. Wählen Sie Weiter aus.
5. Geben Sie im Feld für die Richtliniensuche **`lambda-apigateway-policy`** ein.
6. Wählen Sie in den Suchergebnissen die von Ihnen erstellte Richtlinie (`lambda-apigateway-policy`) und dann die Option Next (Weiter) aus.
7. Geben Sie unter Role details (Rollendetails) für den Role name (Rollennamen) **`lambda-apigateway-role`** ein und wählen Sie dann Create role (Rolle erstellen) aus.

So erstellen Sie die Lambda-Funktion:

1. Öffnen Sie die Funktionsseite der Lambda-Konsole und wählen Sie Funktion erstellen aus.
2. Wählen Sie Von Grund auf neu schreiben aus.
3. Geben Sie für Function name (Funktionsname) `LambdaFunctionOverHttps` ein.
4. Wählen Sie für Laufzeit die neueste Node.js- oder Python-Laufzeit aus.
5. Erweitern Sie unter Berechtigungen die Option Standardausführungsrolle ändern.
6. Wählen Sie Vorhandene Rolle verwenden und anschließend die zuvor erstellte Rolle „**`lambda-apigateway-role`**“ aus.
7. Wählen Sie Funktion erstellen.
8. Ersetzen Sie im Bereich Codequelle den Standardcode durch den folgenden Node.js- oder Python-Code.

   Node.js

   Die `region` Einstellung muss mit dem AWS-Region Ort übereinstimmen, an dem Sie die Funktion bereitstellen und die DynamoDB-Tabelle erstellen.

   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

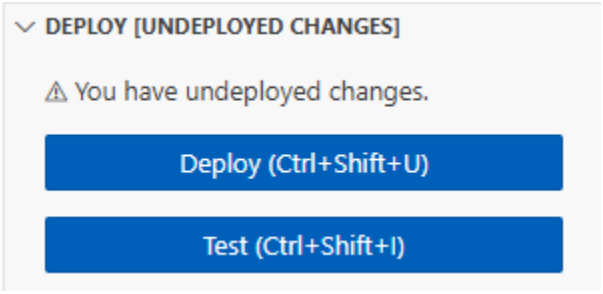
Example lambda_function.py

```
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**
>
> In diesem Beispiel ist der Name der DynamoDB-Tabelle als Variable in Ihrem Funktionscode definiert. In einer realen Anwendung besteht die bewährte Methode darin, diesen Parameter als Umgebungsvariable zu übergeben und den Tabellennamen nicht fest zu codieren. Weitere Informationen finden Sie unter AWS Lambda Umgebungsvariablen verwenden.

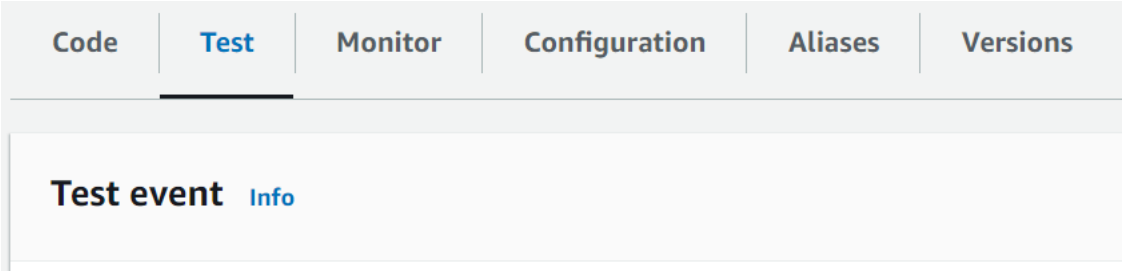
9. Wählen Sie im Abschnitt BEREITSTELLEN die Option Bereitstellen aus, um den Code Ihrer Funktion zu aktualisieren:



Testen der Funktion

Bestätigen Sie vor der Integration Ihrer Funktion mit API-Gateway, dass Sie die Funktion erfolgreich bereitgestellt haben. Senden Sie mithilfe der Lambda-Konsole ein Testereignis an Ihre Funktion.

1. Wählen Sie auf der Lambda-Konsolenseite für Ihre Funktion die Registerkarte Test aus.



2. Scrollen Sie nach unten zum Abschnitt Ereignis-JSON und ersetzen Sie das Standardereignis durch Folgendes. Dieses Ereignis entspricht der von der Lambda-Funktion erwarteten Struktur.

```
{
    "operation": "echo",
    "payload": {
        "somekey1": "somevalue1",
        "somekey2": "somevalue2"
    }
}
```

3. Wählen Sie Test aus.
4. Erweitern Sie unter Funktionsausführung: erfolgreich die Option Details. Sie sollten die folgende Antwort sehen:

```
{
  "somekey1": "somevalue1",
  "somekey2": "somevalue2"
}
```

Erstellen Sie eine REST-API mit API Gateway

In diesem Schritt erstellen Sie die API-Gateway-REST-API, die Sie zum Aufrufen Ihrer Lambda-Funktion verwenden.

So erstellen Sie die API

1. Öffnen Sie die API Gateway-Konsole.
2. Wählen Sie Create API (API erstellen) aus.
3. Wählen Sie im Feld REST-API die Option Entwickeln aus.
4. Lassen Sie unter API details (API-Details) die Option New API (Neue API) ausgewählt und geben Sie für API Name (API-Name) **`DynamoDBOperations`** ein.

5. Wählen Sie Create API (API erstellen) aus.

Erstellen einer Ressource für Ihre REST-API

Um Ihrer API eine HTTP-Methode hinzuzufügen, müssen Sie zunächst eine Ressource erstellen, mit der diese Methode ausgeführt werden kann. Hier erstellen Sie die Ressource zur Verwaltung Ihrer DynamoDB-Tabelle.

So erstellen Sie die Ressource

1. Wählen Sie in der API-Gateway-Konsole auf der Seite Ressourcen für Ihre API die Option Ressource erstellen aus.
2. Geben Sie im Feld Resource details (Ressourcendetails) für Resource name (Ressourcenname) **`DynamoDBManager`** ein.
3. Wählen Sie Create Resource (Ressource erstellen) aus.

Erstellen einer HTTP-POST-Methode

In diesem Schritt erstellen Sie eine Methode (`POST`) für Ihre `DynamoDBManager`-Ressource. Sie verknüpfen diese `POST`-Methode mit Ihrer Lambda-Funktion, sodass API-Gateway Ihre Lambda-Funktion aufruft, wenn die Methode eine HTTP-Anfrage empfängt.

> **Note**
>
> Für die Zwecke dieses Tutorials wird eine HTTP-Methode (`POST`) verwendet, um eine einzelne Lambda-Funktion aufzurufen, die alle Operationen in Ihrer DynamoDB-Tabelle ausführt. In einer realen Anwendung besteht die bewährte Methode darin, für jede Operation eine andere Lambda-Funktion und HTTP-Methode zu verwenden. Weitere Informationen finden Sie bei Serverless Land unter The Lambda monolith.

So erstellen Sie die POST-Methode

1. Vergewissern Sie sich, dass die `/DynamoDBManager`-Ressource auf der Seite Resources (Ressourcen) für Ihre API markiert ist. Wählen Sie dann im Bereich Methods (Methoden) die Option Create Method (Methode erstellen) aus.
2. Wählen Sie in Method type (Methodentyp) POST.

3. Lassen Sie für den Integration type (Integrationstyp) die Option Lambda function (Lambda-Funktion) ausgewählt.
4. Wählen Sie für die Lambda function (Lambda-Funktion) den Amazon Ressourcennamen (ARN) für Ihre Funktion (`LambdaFunctionOverHttps`).
5. Wählen Sie Methode erstellen aus.

Erstellen einer DynamoDB-Tabelle

Erstellen Sie eine leere DynamoDB-Tabelle, an der Ihre Lambda-Funktion CRUD-Operationen ausführt.

Erstellen einer DynamoDB-Tabelle

1. Öffnen Sie die Seite Tables (Tabellen) in der DynamoDB-Konsole.
2. Wählen Sie Create table (Tabelle erstellen) aus.
3. Führen Sie unter Tabellendetails die folgenden Schritte aus:
    1. Geben Sie unter Tabellenname **`lambda-apigateway`** ein.
    2. Geben Sie für Partitionsschlüssel **`id`** ein und behalten Sie den Datentyp als Zeichenfolge bei.
4. Behalten Sie unter Table settings (Tabelleneinstellungen) die Default settings (Standardeinstellungen) bei.
5. Wählen Sie Create table (Tabelle erstellen) aus.

Testen der Integration von API-Gateway, Lambda und DynamoDB

Sie können nun die Integration Ihrer APIä-Gateway-API-Methode mit Ihrer Lambda-Funktion und Ihrer DynamoDB-Tabelle testen. Mit der API Gateway-Konsole senden Sie Anfragen mithilfe der Testfunktion der Konsole direkt an Ihre `POST`-Methode. In diesem Schritt verwenden Sie zuerst eine `create`-Operation, um Ihrer DynamoDB-Tabelle ein neues Element hinzuzufügen, und verwenden dann eine `update`-Operation, um das Element zu ändern.

Test 1: So erstellen Sie ein neues Element in Ihrer DynamoDB-Tabelle

1. Wählen Sie in der API-Gateway-Konsole Ihre API aus (`DynamoDBOperations`).
2. Wählen Sie unter der Ressource `DynamoDBManager` die Methode POST.

3. Wählen Sie die Registerkarte Test. Möglicherweise müssen Sie die rechte Pfeiltaste wählen, um die Registerkarte anzuzeigen.
4. Lassen Sie unter Test method (Testmethode) die Felder Query strings (Query-Strings) und Headers (Header) leer. Fügen Sie für Request body (Anforderungstext) den folgenden JSON-Code ein:

```
{
  "operation": "create",
  "payload": {
    "Item": {
      "id": "1234ABCD",
      "number": 5
    }
  }
}
```

5. Wählen Sie Test aus.

   Die Ergebnisse, die nach Abschluss des Tests angezeigt werden, sollten den Status `200` anzeigen. Dieser Statuscode zeigt an, dass die `create`-Operation erfolgreich war.

   Überprüfen Sie zur Bestätigung, ob Ihre DynamoDB-Tabelle jetzt das neue Element enthält.
6. Öffnen Sie die Seite Tabellen der DynamoDB-Konsole und wählen Sie die `lambda-apigateway`-Tabelle aus.
7. Wählen Sie Explore table items (Tabellenelemente erkunden) aus. Im Bereich Items returned (Zurückgegebene Elemente) sollten ein Element mit der id (ID) `1234ABCD` und der number (Nummer) `5` angezeigt werden. Beispiel:

Test 2: So aktualisieren Sie das Element in Ihrer DynamoDB-Tabelle

1. Kehren Sie in der API-Gateway-Konsole zum Bereich Test Ihrer POST-Methode zurück.

2. Lassen Sie unter Test method (Testmethode) die Felder Query strings (Query-Strings) und Headers (Header) leer. Fügen Sie für Request body (Anforderungstext) den folgenden JSON-Code ein:

```
{
    "operation": "update",
    "payload": {
        "Key": {
            "id": "1234ABCD"
        },
        "UpdateExpression": "SET #num = :newNum",
        "ExpressionAttributeNames": {
            "#num": "number"
        },
        "ExpressionAttributeValues": {
            ":newNum": 10
        }
    }
}
```

3. Wählen Sie Test aus.

   Die Ergebnisse, die nach Abschluss des Tests angezeigt werden, sollten den Status `200` anzeigen. Dieser Statuscode zeigt an, dass die `update`-Operation erfolgreich war.

   Überprüfen Sie zur Bestätigung, ob das Element in Ihrer DynamoDB-Tabelle geändert wurde.

4. Öffnen Sie die Seite Tabellen der DynamoDB-Konsole und wählen Sie die `lambda-apigateway`-Tabelle aus.

5. Wählen Sie Explore table items (Tabellenelemente erkunden) aus. Im Bereich Items returned (Zurückgegebene Elemente) sollten ein Element mit der id (ID) `1234ABCD` und der number (Nummer) `10` angezeigt werden.

Bereitstellen der API

Damit ein Client die API aufrufen kann, müssen Sie eine Bereitstellung und eine zugehörige Stufe erstellen. Eine Phase stellt eine Momentaufnahme Ihrer API dar, einschließlich ihrer Methoden und Integrationen.

So stellen Sie die API bereit

1. Öffnen Sie die APIsSeite der API Gateway Gateway-Konsole und wählen Sie die `DynamoDBOperations` API aus.
2. Wählen Sie auf der Seite Resources (Ressourcen) für Ihre API die Option Deploy API (API bereitstellen) aus.
3. Wählen Sie für Stage (Stufe) *New stage* (*Neue Stufe*) und geben Sie dann als Stage name (Phasenname) **`test`** ein.
4. Wählen Sie Bereitstellen.
5. Kopieren Sie im Bereich Stage details (Stufendetails) die Invoke URL (Aufruf-URL). Sie werden diese im nächsten Schritt verwenden, um Ihre Funktion mithilfe einer HTTP-Anfrage aufzurufen.

Verwenden von curl zum Aufrufen Ihrer Funktion mithilfe von HTTP-Anfragen

Sie können Ihre Lambda-Funktion jetzt aufrufen, indem Sie eine HTTP-Anfrage an Ihre API senden. In diesem Schritt erstellen Sie ein neues Element in Ihrer DynamoDB-Tabelle und führen dann Lese-, Aktualisierungs- und Löschvorgänge für dieses Element durch.

So erstellen Sie mit curl ein Element in Ihrer DynamoDB-Tabelle

1. Öffnen Sie ein Terminal oder eine Eingabeaufforderung auf Ihrem lokalen Computer und führen Sie den folgenden `curl`-Befehl mit der im vorherigen Schritt kopierten Aufruf-URL aus. Dieser Befehl verwendet die folgenden Optionen:

   - `-H`: Fügt der Anforderung einen benutzerdefinierten Header hinzu. Hier wird der Inhaltstyp als JSON angegeben.
   - `-d`: Sendet Daten im Anforderungstext. Diese Option verwendet standardmäßig eine HTTP-POST-Methode.

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}}}'
```

Wenn der Vorgang erfolgreich war, sollten Sie eine Antwort mit einem HTTP-Statuscode von 200 erhalten.

2. Sie können auch die DynamoDB-Konsole verwenden, um zu überprüfen, ob sich das neue Element in Ihrer Tabelle befindet, indem Sie Folgendes tun:

   1. Öffnen Sie die Seite Tabellen in der DynamoDB-Konsole und wählen Sie die `lambda-apigateway`-Tabelle aus.
   2. Wählen Sie Explore Table Items (Tabellenelemente erkudnen) aus. Im Bereich Items returned (Zurückgegebene Elemente) sollte ein Element mit der id (ID) `5678EFGH` und der number (Nummer) 15 angezeigt werden.

So lesen Sie das Element in Ihrer DynamoDB-Tabelle mit curl

- Führen Sie in Ihrem Terminal oder Ihrer Eingabeaufforderung den folgenden `curl`-Befehl aus, um den Wert des soeben erstellten Elements zu lesen. Verwenden Sie Ihre eigene Aufruf-URL.

  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\"}}}'
```

Je nachdem, ob Sie den Node.js-Funktionscode oder den Python-Funktionscode gewählt haben, sollten Sie eine der folgenden Ausgaben sehen:

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}}
```

So aktualisieren Sie das Element in Ihrer DynamoDB-Tabelle mit curl

1. Führen Sie in Ihrem Terminal oder Ihrer Eingabeaufforderung den folgenden `curl`-Befehl aus, um das soeben erstellte Element durch Ändern des `number`-Werts zu aktualisieren. Verwenden Sie Ihre eigene Aufruf-URL.

   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. Um zu bestätigen, dass der Wert von `number` für das Element aktualisiert wurde, führen Sie einen weiteren Lesebefehl aus:

   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\"}}}'
```

So löschen Sie das Element in Ihrer DynamoDB-Tabelle mit curl

1. Führen Sie in Ihrem Terminal oder Ihrer Eingabeaufforderung den folgenden `curl`-Befehl aus, um das soeben erstellte Element zu löschen. Verwenden Sie Ihre eigene Aufruf-URL.

   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. Bestätigen Sie, dass der Löschvorgang erfolgreich war. Stellen Sie im Bereich Zurückgegebene Elemente der Seite Elemente entdecken der DynamoDB-Konsole sicher, dass das Element mit der ID `5678EFGH` nicht mehr in der Tabelle enthalten ist.

Bereinigen Ihrer Ressourcen (optional)

Sie können jetzt die Ressourcen, die Sie für dieses Tutorial erstellt haben, löschen, es sei denn, Sie möchten sie behalten. Durch das Löschen von AWS Ressourcen, die Sie nicht mehr verwenden, vermeiden Sie unnötige Kosten für Ihre AWS-Konto.

So löschen Sie die Lambda-Funktion:

1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.
2. Wählen Sie die Funktion aus, die Sie erstellt haben.
3. Wählen Sie Aktionen, Löschen aus.
4. Geben Sie **`confirm`** in das Texteingabefeld ein und wählen Sie Delete (Löschen) aus.

So löschen Sie die Ausführungsrolle

1. Öffnen Sie die Seite Roles in der IAM-Konsole.
2. Wählen Sie die von Ihnen erstellte Ausführungsrolle aus.
3. Wählen Sie Löschen aus.
4. Geben Sie den Namen der Rolle in das Texteingabefeld ein und wählen Sie Delete (Löschen) aus.

So löschen Sie die API

1. Öffnen Sie die APIs Seite der API Gateway Gateway-Konsole.
2. Wählen Sie die von Ihnen erstellte API aus.
3. Wählen Sie Aktionen, Löschen aus.

4. Wählen Sie Löschen aus.

So löschen Sie die DynamoDB-Tabelle

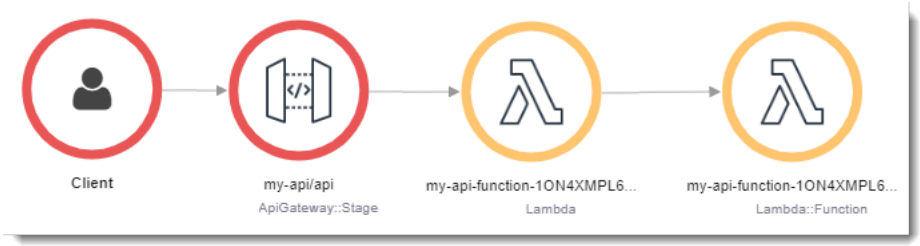
1. Öffnen Sie die Seite Tables (Tabellen) in der DynamoDB-Konsole.
2. Wählen Sie die von Ihnen erstellte Tabelle aus.
3. Wählen Sie Löschen aus.
4. Geben Sie **`delete`** in das Textfeld ein.
5. Wählen Sie Delete Table (Tabelle löschen).

Behandlung von Lambda-Fehlern mit einer API-Gateway-API

API Gateway behandelt alle Aufruf- und Funktionsfehler als interne Fehler. Wenn die Lambda-API die Aufruf-Anforderung ablehnt, gibt API Gateway einen 500-Fehlercode zurück. Wenn die Funktion ausgeführt wird, aber einen Fehler oder eine Antwort im falschen Format zurückgibt, gibt API Gateway den Fehlercode 502 zurück. In beiden Fällen lautet der Text der Antwort von API Gateway „`{"message": "Internal server error"}`“.

> **Note**
>
> API Gateway wiederholt keine Lambda-Aufrufe. Wenn Lambda einen Fehler zurückgibt, gibt API Gateway eine Fehlerantwort an den Client zurück.

Das folgende Beispiel zeigt eine X-Ray-Ablaufverfolgungszuordnung für eine Anforderung, die zu einem Funktionsfehler und einen 502-Fehler von API Gateway führte. Der Client erhält die generische Fehlermeldung.

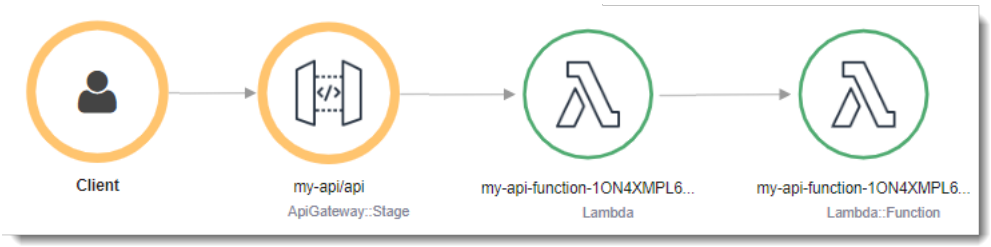


Um die Fehlerantwort anzupassen, müssen Sie Fehler im Code abfangen und eine Antwort im erforderlichen Format formatieren.

Example index.mjs – Formatierfehler

```
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 konvertiert diese Antwort in einen HTTP-Fehler mit einem benutzerdefinierten Statuscode und Text. In der Ablaufverfolgungszuweisung ist der Funktionsknoten grün, da er den Fehler behandelt hat.



Wählen Sie eine Methode, um Ihre Lambda-Funktion über eine HTTP-Anfrage aufzurufen

Viele häufige Anwendungsfälle für Lambda beinhalten den Aufruf Ihrer Funktion über eine HTTP-Anfrage. Beispielsweise möchten Sie möglicherweise, dass eine Webanwendung Ihre Funktion über eine Browseranforderung aufruft. Lambda-Funktionen können auch verwendet werden, um vollständige REST zu erstellen APIs, Benutzerinteraktionen von mobilen Apps aus abzuwickeln, Daten von externen Diensten über HTTP-Aufrufe zu verarbeiten oder benutzerdefinierte Webhooks zu erstellen.

In den folgenden Abschnitten wird erläutert, welche Möglichkeiten Sie haben, Lambda über HTTP aufzurufen und es werden Informationen bereitgestellt, die Ihnen helfen, die richtige Entscheidung für Ihren speziellen Anwendungsfall zu treffen.

Welche Optionen stehen Ihnen bei der Auswahl einer HTTP-Aufrufmethode zur Verfügung?

Lambda bietet zwei Hauptmethoden zum Aufrufen einer Funktion mithilfe einer HTTP-Anfrage: Funktion URLs und API Gateway. Die wichtigsten Unterschiede zwischen diesen beiden Optionen sind die folgenden:

- Die Lambda-Funktion URLs bietet einen einfachen, direkten HTTP-Endpunkt für eine Lambda-Funktion. Sie sind auf Einfachheit und Kosteneffizienz optimiert und bieten den schnellsten Weg, um eine Lambda-Funktion über HTTP bereitzustellen.
- API Gateway ist ein fortschrittlicherer Dienst zum Erstellen von Funktionen mit vollem Funktionsumfang APIs. API Gateway ist für den Aufbau und die Verwaltung von Produktionen APIs in großem Maßstab optimiert und bietet umfassende Tools für Sicherheit, Überwachung und Verkehrsmanagement.

Empfehlungen, falls Sie Ihre Anforderungen bereits kennen

Wenn Sie sich über Ihre Anforderungen bereits im Klaren sind, finden Sie hier unsere grundlegenden Empfehlungen:

Wir empfehlen Funktionen URLs für einfache Anwendungen oder für die Prototypenentwicklung, bei denen Sie nur grundlegende Authentifizierungsmethoden und request/response deren Handhabung benötigen und bei denen Sie Kosten und Komplexität auf ein Minimum reduzieren möchten.

API Gateway ist eine bessere Wahl für Produktionsanwendungen in großem Maßstab oder für Fälle, in denen Sie erweiterte Funktionen wie OpenAPI-Description, eine Auswahl an Authentifizierungsoptionen, benutzerdefinierte Domainnamen oder umfangreiche request/response Funktionen wie Drosselung, Caching und Transformation benötigen. request/response

Was ist bei der Auswahl einer Methode zum Aufrufen Ihrer Lambda-Funktion zu beachten?

Bei der Auswahl zwischen Funktion URLs und API Gateway müssen Sie die folgenden Faktoren berücksichtigen:

- Ihre Authentifizierungsanforderungen, z. B. ob Sie Amazon Cognito für die Benutzerauthentifizierung benötigen OAuth
- Ihre Skalierungsanforderungen und die Komplexität der API, die Sie implementieren möchten

- Ob Sie erweiterte Funktionen wie die Validierung und Formatierung von Anfragen benötigen request/response
- Ihre Überwachungsanforderungen
- Was sind Ihre Ziele?

Wenn Sie diese Faktoren verstehen, können Sie die Option auswählen, die Ihren Anforderungen an Sicherheit, Komplexität und Kosten am besten gerecht wird.

Die folgenden Informationen fassen die wichtigsten Unterschiede zwischen den beiden Optionen zusammen.

Authentifizierung

- Die Funktion URLs bietet grundlegende Authentifizierungsoptionen über AWS Identity and Access Management (IAM). Sie können Ihre Endpunkte so konfigurieren, dass sie entweder öffentlich sind (keine Authentifizierung) oder dass sie eine IAM-Authentifizierung erfordern. Mit der IAM-Authentifizierung können Sie AWS Standardanmeldedaten oder IAM-Rollen verwenden, um den Zugriff zu steuern. Dieser Ansatz ist zwar einfach einzurichten, bietet aber im Vergleich zu anderen Authentifizierungsmethoden nur begrenzte Optionen.
- API Gateway bietet Zugriff auf eine umfassendere Palette von Authentifizierungsoptionen. Neben der IAM-Authentifizierung können Sie Lambda-Autorisierer (benutzerdefinierte Authentifizierungslogik), Amazon Cognito Cognito-Benutzerpools und .0-Flows verwenden. OAuth2 Diese Flexibilität ermöglicht es Ihnen, komplexe Authentifizierungsschemata zu implementieren, darunter Authentifizierungsanbieter von Drittanbietern, tokenbasierte Authentifizierung und Multi-Faktor-Authentifizierung.

Anfrage-/Antwort-Anforderungen

- Die Funktion URLs bietet eine grundlegende Behandlung von HTTP-Anfragen und -Antworten. Sie unterstützen Standard-HTTP-Methoden und bieten integrierte CORS-Unterstützung (Cross-Origin Resource Sharing). Sie können zwar JSON-Nutzdaten und Abfrageparameter auf natürliche Weise verarbeiten, bieten jedoch keine Funktionen zur Transformation oder Validierung von Anfragen. Die Bearbeitung von Antworten ist ähnlich einfach – der Client erhält die Antwort von Ihrer Lambda-Funktion genau so, wie Lambda sie zurückgibt.
- API Gateway bietet ausgefeilte Funktionen zur Bearbeitung von Anfragen und Antworten. Sie können Anforderungsvalidatoren definieren, Anfragen und Antworten mithilfe von Zuordnungsvorlagen transformieren, request/response Header einrichten und das

Zwischenspeichern von Antworten implementieren. API Gateway unterstützt auch binäre Nutzdaten und benutzerdefinierte Domainnamen und kann Antworten ändern, bevor sie den Client erreichen. Sie können Modelle für die request/response Validierung und Transformation mithilfe des JSON-Schemas einrichten.

Skalierung

- Die Funktion URLs skaliert direkt mit den Parallelitätsgrenzen Ihrer Lambda-Funktion und bewältigt Verkehrsspitzen, indem Sie Ihre Funktion bis zum konfigurierten Maximallimit skalieren. Sobald dieses Limit erreicht ist, reagiert Lambda auf weitere Anfragen mit HTTP 429-Antworten. Es gibt keinen integrierten Warteschlangenmechanismus, sodass der Umgang mit der Skalierung vollständig von der Konfiguration Ihrer Lambda-Funktion abhängt. Standardmäßig haben Lambda-Funktionen ein Limit von 1.000 gleichzeitigen Ausführungen pro. AWS-Region
- API Gateway bietet zusätzlich zur Lambda-eigenen Skalierung zusätzliche Skalierungsfunktionen. Es umfasst integrierte Funktionen zur Warteschleife und Drosselung von Anfragen, sodass Sie Datenverkehrsspitzen besser bewältigen können. API Gateway kann standardmäßig bis zu 10.000 Anfragen pro Sekunde pro Region verarbeiten, mit einer Burst-Kapazität von 5.000 Anfragen pro Sekunde. Es bietet auch Tools zum Drosseln von Anfragen auf verschiedenen Ebenen (API, Phase oder Methode), um Ihr Backend zu schützen.

Überwachen

- Function URLs bietet eine grundlegende Überwachung anhand von CloudWatch Amazon-Metriken, einschließlich Anzahl der Anfragen, Latenz und Fehlerraten. Sie erhalten Zugriff auf standardmäßige Lambda-Metriken und -Protokolle, die die Rohanfragen anzeigen, die in Ihre Funktion eingehen. Dies bietet zwar wichtige betriebliche Transparenz, die Metriken konzentrieren sich jedoch hauptsächlich auf die Funktionsausführung.
- API Gateway bietet umfassende Überwachungsfunktionen, einschließlich detaillierter Metriken, Protokollierungs- und Verfolgungsoptionen. Sie können API-Aufrufe, Latenz, Fehlerraten und hit/miss Cache-Raten über überwachen CloudWatch. API Gateway lässt sich auch AWS X-Ray für verteiltes Tracing integrieren und bietet anpassbare Protokollierungsformate.

Cost (Kosten)

- Die Funktionen URLs folgen dem Standard-Lambda-Preismodell — Sie zahlen nur für Funktionsaufrufe und Rechenzeit. Für den URL-Endpunkt selbst fallen keine zusätzlichen

Gebühren an. Dies macht es zu einer kostengünstigen Wahl für einfache Anwendungen APIs oder Anwendungen mit geringem Datenverkehr, wenn Sie die zusätzlichen Funktionen von API Gateway nicht benötigen.

- API Gateway bietet ein kostenloses Kontingent, das eine Million API-Aufrufe für REST APIs und eine Million API-Aufrufe für HTTP umfasst APIs. Danach berechnet API Gateway Gebühren für API-Aufrufe, Datenübertragung und Caching (falls aktiviert). Weitere Informationen zu den Kosten für Ihren eigenen Anwendungsfall finden Sie auf der Seite mit den Preisen für API Gateway.

Weitere Funktionen

- URLsDie Funktionen sind auf Einfachheit und direkte Lambda-Integration ausgelegt. Sie unterstützen sowohl HTTP- als auch HTTPS-Endpunkte, bieten integrierte CORS-Unterstützung und bieten Dual-Stack- (IPv4 und) Endpunkte. IPv6 Ihnen fehlen zwar erweiterte Features, sie zeichnen sich jedoch in Szenarien aus, in denen Sie Lambda-Funktionen schnell und unkompliziert über HTTP verfügbar machen müssen.
- API Gateway umfasst zahlreiche zusätzliche Funktionen wie API-Versionierung, Stagemanagement, API-Schlüssel für Nutzungspläne, API-Dokumentation über Swagger/ OpenAPI WebSocket APIs, private Funktionen APIs innerhalb einer VPC und WAF-Integration für zusätzliche Sicherheit. Es unterstützt auch kanarische Implementierungen, Scheinintegrationen zum Testen und die Integration mit anderen Systemen AWS-Services als Lambda.

Wählen Sie eine Methode, um Ihre Lambda-Funktion aufzurufen

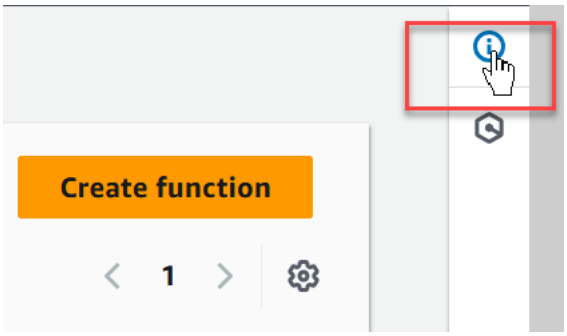
Nachdem Sie sich mit den Kriterien für die Auswahl zwischen Lambda-Funktion URLs und API Gateway und den wichtigsten Unterschieden zwischen ihnen vertraut gemacht haben, können Sie die Option auswählen, die Ihren Anforderungen am besten entspricht, und die folgenden Ressourcen verwenden, um Ihnen den Einstieg in die Verwendung zu erleichtern.

Function URLs

Nutzen Sie die folgenden Ressourcen, um URLs mit Function zu beginnen

- Folgen Sie dem Tutorial Erstellen einer Lambda-Funktion mit einer Funktions-URL
- Weitere Informationen zur Funktion finden Sie URLs im the section called “Funktion URLs” Kapitel dieses Handbuchs
- Probieren Sie das in der Konsole enthaltene Tutorial Erstellen Sie eine einfache Web-App aus, indem Sie wie folgt vorgehen:

1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.
2. Öffnen Sie das Hilfe-Panel, indem Sie das Symbol in der oberen rechten Ecke des Bildschirms wählen.



3. Wählen Sie Tutorials aus.
4. Wählen Sie unter Einfache Web-App erstellen die Option Tutorial starten aus.

API Gateway

Beginnen Sie mit Lambda und API Gateway mit den folgenden Ressourcen

- Folgen Sie dem Tutorial Verwenden von Lambda mit API Gateway, um eine REST-API zu erstellen, die in eine Backend-Lambda-Funktion integriert ist.
- In den folgenden Abschnitten des Amazon API Gateway Developer Guide erfahren Sie mehr über die verschiedenen Arten von APIs, die von API Gateway angeboten werden:
  - API Gateway REST APIs
  - API Gateway HTTP APIs
  - API Gateway WebSocket APIs
- Probieren Sie eines oder mehrere der Beispiele im Abschnitt Tutorials und Workshops im Amazon API Gateway Developer Guide aus.

Verwenden AWS Lambda mit AWS Infrastructure Composer

AWS Infrastructure Composer ist ein Visual Builder zum Entwerfen moderner Anwendungen auf AWS. Sie entwerfen Ihre Anwendungsarchitektur, indem Sie sie auf einer visuellen Leinwand ziehen, gruppieren und verbinden AWS-Services . Infrastructure Composer erstellt anhand Ihres Entwurfs IaC-Vorlagen (Infrastructure as Code), die Sie mithilfe von AWS SAM oder CloudFormation bereitstellen können.

Exportieren einer Lambda-Funktion in Infrastructure Composer

Sie können mit der Verwendung von Infrastructure Composer beginnen, indem Sie mithilfe der Lambda-Konsole ein neues Projekt erstellen, das auf der Konfiguration einer vorhandenen Lambda-Funktion basiert. Gehen Sie wie folgt vor, um die Konfiguration und den Code Ihrer Funktion nach Infrastructure Composer zu exportieren und ein neues Projekt zu erstellen:

1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.
2. Wählen Sie die Funktion aus, die Sie als Grundlage für Ihr Infrastructure-Composer-Projekt verwenden möchten.
3. Wählen Sie im Bereich Funktionsübersicht die Option Nach Infrastructure Composer exportieren aus.

   Um die Konfiguration und den Code Ihrer Funktion nach Infrastructure Composer zu exportieren, erstellt Lambda einen Amazon-S3-Bucket in Ihrem Konto, um diese Daten vorübergehend zu speichern.
4. Wählen Sie im Dialogfeld Projekt bestätigen und erstellen aus, um den Standardnamen für diesen Bucket zu akzeptieren und die Konfiguration und den Code Ihrer Funktion nach Infrastructure Composer zu exportieren.
5. (Optional) Um einen anderen Namen für den von Lambda erstellten Amazon-S3-Bucket auszuwählen, geben Sie einen neuen Namen ein und wählen Sie Projekt bestätigen und erstellen aus. Die Amazon-S3-Bucket-Namen müssen global eindeutig sein und den Regeln für die Benennung von Buckets entsprechen.
6. Um Ihre Projekt- und Funktionsdateien in Infrastructure Composer zu speichern, aktivieren Sie den lokalen Synchronisierungsmodus.

> **Note**
>
> Wenn Sie das Feature Nach Application Composer exportieren bereits verwendet und einen Amazon-S3-Bucket mit dem Standardnamen erstellt haben, kann Lambda diesen Bucket wiederverwenden, falls er noch existiert. Akzeptieren Sie den Standard-Bucket-Namen im Dialogfeld, um den vorhandenen Bucket wiederzuverwenden.

Konfiguration des Amazon-S3-Transfer-Buckets

Der Amazon-S3-Bucket, den Lambda für die Übertragung der Konfiguration Ihrer Funktion erstellt, verschlüsselt Objekte automatisch mit dem Verschlüsselungsstandard AES 256. Lambda konfiguriert den Bucket auch so, dass er die Bucket-Besitzer-Bedingung verwendet, um sicherzustellen, dass nur Sie AWS-Konto Objekte zum Bucket hinzufügen können.

Lambda konfiguriert den Bucket so, dass Objekte 10 Tage nach dem Hochladen automatisch gelöscht werden. Lambda löscht den Bucket selbst jedoch nicht automatisch. Um den Bucket aus Ihrem zu löschen AWS-Konto, folgen Sie den Anweisungen unter Bucket löschen. Der Standard-Bucket-Name verwendet das Präfix`lambdasam`, eine 10-stellige alphanumerische Zeichenfolge und das, in dem AWS-Region Sie Ihre Funktion erstellt haben:

```
lambdasam-06f22da95b-us-east-1
```

Um zusätzliche Gebühren zu vermeiden, empfehlen wir Ihnen AWS-Konto, den Amazon S3 S3-Bucket zu löschen, sobald Sie den Export Ihrer Funktion in Infrastructure Composer abgeschlossen haben.

Es gelten die Standardpreise von Amazon S3.

Erforderliche Berechtigungen

Um die Funktion Lambda-Integration mit Infrastructure Composer verwenden zu können, benötigen Sie bestimmte Berechtigungen, um eine AWS SAM Vorlage herunterzuladen und die Konfiguration Ihrer Funktion in Amazon S3 zu schreiben.

Um eine AWS SAM Vorlage herunterzuladen, benötigen Sie die Erlaubnis, die folgenden API-Aktionen zu verwenden:

- GetPolicy
- Ich bin: GetPolicyVersion
- ich bin: GetRole
- ich bin: GetRolePolicy
- ich bin: ListAttachedRolePolicies
- ich bin: ListRolePolicies
- ich bin: ListRoles

Sie können die Erlaubnis zur Verwendung all dieser Aktionen erteilen, indem Sie die `AWSLambda_ReadOnlyAccess` AWS verwaltete Richtlinie zu Ihrer IAM-Benutzerrolle hinzufügen.

Damit Lambda die Konfiguration Ihrer Funktion in Amazon S3 schreiben kann, benötigen Sie die Erlaubnis, die folgenden API-Aktionen zu verwenden:

- S3: PutObject
- S 3: CreateBucket
- S 3: PutBucketEncryption
- S 3: PutBucketLifecycleConfiguration

Wenn Sie die Konfiguration Ihrer Funktion nicht nach Infrastructure Composer exportieren können, überprüfen Sie, ob Ihr Konto über die erforderlichen Berechtigungen für diese Operationen verfügt. Wenn Sie über die erforderlichen Berechtigungen verfügen, die Konfiguration Ihrer Funktion aber immer noch nicht exportieren können, suchen Sie nach ressourcenbasierten Richtlinien, die möglicherweise den Zugriff auf Amazon S3 einschränken.

Sonstige Ressourcen

Ein ausführlicheres Tutorial zum Entwerfen einer Serverless-Anwendung in Infrastructure Composer auf der Grundlage einer vorhandenen Lambda-Funktion finden Sie unter the section called “Infrastructure as Code (IaC)”.

Um Infrastructure Composer AWS SAM zu verwenden und eine vollständige serverlose Anwendung mit Lambda zu entwerfen und bereitzustellen, können Sie auch dem AWS Infrastructure Composer Tutorial im AWS Serverless Patterns Workshop folgen.

Verwenden von AWS Lambda mit CloudFormation

In einer AWS CloudFormation-Vorlage können Sie eine Lambda-Funktion als Ziel einer benutzerdefinierten Ressource angeben. Mittels benutzerdefinierter Ressourcen können Sie Parameter verarbeiten, Konfigurationswerte abrufen oder während Stack-Lebenszyklusereignissen andere AWS-Services aufrufen.

Das folgende Beispiel ruft eine Funktion auf, die an anderer Stelle in der Vorlage definiert ist.

Example – benutzerdefinierte Ressourcendefinition

```
Resources:
  primerinvoke:
    Type: AWS::CloudFormation::CustomResource
    Version: "1.0"
    Properties:
      ServiceToken: !GetAtt primer.Arn
      FunctionName: !Ref randomerror
```

Das Service-Token ist der Amazon-Ressourcenname (ARN) der Funktion, die von CloudFormation aufgerufen wird, wenn Sie den Stack erstellen, aktualisieren oder löschen. Sie können auch zusätzliche Eigenschaften wie `FunctionName` einschließen, die von CloudFormation unverändert an Ihre Funktion übergeben werden.

CloudFormation ruft Ihre Lambda-Funktion asynchron mit einem Ereignis mit einer Callback-URL auf.

Example – CloudFormation-Nachrichtenereignis

```
{
    "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"
    }
}
```

Die Funktion ist für die Rückgabe einer Antwort an die Rückruf-URL zuständig, die Erfolg oder Misserfolg bedeutet. Die vollständige Antwortsyntax finden Sie unter Benutzerdefinierte Ressourcenantwortobjekte.

Example – CloudFormation benutzerdefinierte -Ressourcenantwort

```
{
    "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 bietet eine Bibliothek mit dem Namen `cfn-response`, die das Senden der Antwort bearbeitet. Wenn Sie Ihre Funktion in einer Vorlage definieren, können Sie die Bibliothek nach Namen anfordern. CloudFormation fügt die Bibliothek dann dem Bereitstellungspaket hinzu, das es für die Funktion erstellt.

Wenn Ihre Funktion, die eine benutzerdefinierte Ressource verwendet, mit einer Elastic Network Interface verbunden ist, fügen Sie die folgenden Ressourcen zur VPC-Richtlinie hinzu, wobei **`region`** die Region ist, in der sich die Funktion befindet, ohne Bindestriche angegeben ist. Zum Beispie, `us-east-1` ist `useast1`. Dadurch kann die benutzerdefinierte Ressource auf die Callback-URL reagieren, die ein Signal zurück an den CloudFormation-Stack sendet.

```
arn:aws:s3:::cloudformation-custom-resource-response-region",
"arn:aws:s3:::cloudformation-custom-resource-response-region/*",
```

Die folgende Beispielfunktion ruft eine zweite Funktion auf. Wenn der Aufruf erfolgreich ist, sendet die Funktion eine entsprechende Antwort an CloudFormation, und die Stack-Aktualisierung wird

fortgesetzt. Die Vorlage verwendet den Ressourcentyp AWS::Serverless::Function, der von AWS Serverless Application Model bereitgestellt wird.

Example – Benutzerdefinierte Ressourcenfunktion

```
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
```

Wenn die von der benutzerdefinierten Ressource aufgerufene Funktion in keiner Vorlage definiert ist, können Sie den Quellcode für `cfn-response` dem cfn-response-Modul im AWS CloudFormation-Benutzerhandbuch entnehmen.

Weitere Informationen zu benutzerdefinierten Ressourcen finden Sie unter Benutzerdefinierte Ressourcen im AWS CloudFormation-Benutzerhandbuch.

Amazon DocumentDB-Ereignisse mit Lambda verarbeiten

Sie können eine Lambda-Funktion verwenden, um Ereignisse in einem Änderungsstream von Amazon DocumentDB (mit MongoDB-Kompatibilität) zu verarbeiten, indem Sie einen Amazon-DocumentDB-Cluster als Ereignisquelle konfigurieren. Anschließend können Sie ereignisgesteuerte Workloads automatisieren, indem Sie Ihre Lambda-Funktion jedes Mal aufrufen, wenn sich Daten in Ihrem Amazon-DocumentDB-Cluster ändern.

> **Note**
>
> Lambda unterstützt nur die Versionen 4.0 und 5.0 von Amazon DocumentDB. Lambda unterstützt Version 3.6 nicht.
> Für die Zuordnung von Ereignisquellen unterstützt Lambda außerdem nur Instance-basierte Cluster und regionale Cluster. Lambda unterstützt keine elastischen Cluster oder globalen Cluster. Diese Einschränkung gilt nicht, wenn Lambda als Client verwendet wird, um eine Verbindung mit Amazon DocumentDB herzustellen. Lambda kann eine Verbindung mit allen Clustertypen herstellen, um CRUD-Vorgänge auszuführen.

Lambda verarbeitet Ereignisse aus Amazon-DocumentDB-Änderungsstreams sequentiell in der Reihenfolge, in der sie ankommen. Aus diesem Grund kann Ihre Funktion jeweils nur einen Aufruf von Amazon DocumentDB gleichzeitig verarbeiten. Sie können Ihre Funktion anhand ihrer Gleichzeitigkeitsmetriken überwachen.

> **Warning**
>
> Zuordnung von Lambda-Ereignisquellen verarbeiten jedes Ereignis mindestens einmal und es kann zu einer doppelten Verarbeitung von Datensätzen kommen. Um mögliche Probleme im Zusammenhang mit doppelten Ereignissen zu vermeiden, empfehlen wir Ihnen dringend, Ihren Funktionscode idempotent zu machen. Weitere Informationen finden Sie im Knowledge Center unter Wie mache ich meine Lambda-Funktion idempotent?. AWS

Topics

- Beispiel für Amazon-DocumentDB-Ereignis
- Voraussetzungen und Berechtigungen
- Konfigurieren Sie die Netzwerksicherheit

- Erstellen einer Zuordnung von Ereignisquellen von Amazon DocumentDB (Konsole)
- Erstellen einer Zuordnung von Ereignisquellen von Amazon DocumentDB (SDK oder CLI)
- Startpositionen für Abfragen und Streams
- Überwachen Ihrer Amazon-DocumentDB-Ereignisquelle
- Tutorial: Verwendung AWS Lambda mit Amazon DocumentDB DocumentDB-Streams

Beispiel für Amazon-DocumentDB-Ereignis


```
{
    "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"
}
```

Weitere Informationen zu den Ereignissen in diesem Beispiel und den zugehörigen Formen finden Sie unter Änderungsereignisse auf der MongoDB-Dokumentationswebsite.

Voraussetzungen und Berechtigungen

Bevor Sie Amazon DocumentDB als Ereignisquelle für Ihre Lambda-Funktion verwenden können, beachten Sie die folgenden Voraussetzungen. Folgendes muss zutreffen:

- Haben Sie einen vorhandenen Amazon DocumentDB-Cluster in derselben AWS-Konto und AWS-Region wie Ihre Funktion. Wenn Sie über keinen bestehenden Cluster verfügen, können Sie anhand der Anleitung in Erste Schritte mit Amazon DocumentDB im Amazon-DocumentDB-Entwicklerhandbuch einen erstellen. Alternativ dazu führen Sie die ersten Schritte in Tutorial: Verwendung AWS Lambda mit Amazon DocumentDB DocumentDB-Streams durch die Erstellung eines Amazon DocumentDB-Clusters mit allen erforderlichen Voraussetzungen.
- Sie müssen Lambda den Zugriff auf die Ressourcen von Amazon Virtual Private Cloud (Amazon VPC) erlauben, die Ihrem Amazon-DocumentDB-Cluster zugeordnet sind. Weitere Informationen finden Sie unter Konfigurieren Sie die Netzwerksicherheit.
- Aktivieren Sie TLS in Ihrem Amazon-DocumentDB-Cluster. Dies ist die Standardeinstellung. Wenn Sie TLS deaktivieren, kann Lambda nicht mit Ihrem Cluster kommunizieren.
- Aktivieren Sie Änderungsstreams in Ihrem Amazon-DocumentDB-Cluster. Weitere Informationen finden Sie unter Verwenden von Änderungsstreams mit Amazon DocumentDB im Amazon-DocumentDB-Entwicklerhandbuch.
- Stellen Sie Lambda Anmeldeinformationen für den Zugriff auf Ihren Amazon-DocumentDB-Cluster zur Verfügung. Geben Sie bei der Einrichtung der Ereignisquelle den AWS Secrets Manager-Schlüssel mit den Authentifizierungsdetails (Benutzername und Passwort) an, die für den Zugriff auf Ihren Cluster erforderlich sind. Um diesen Schlüssel während der Einrichtung bereitzustellen, haben Sie die folgenden Möglichkeiten:
  - Wenn Sie die Lambda-Konsole für die Einrichtung verwenden, geben Sie diesen Schlüssel in das Feld Secrets-Manager-Schlüssel ein.
  - Wenn Sie das AWS Command Line Interface (AWS CLI) für die Einrichtung verwenden, geben Sie diesen Schlüssel in der `source-access-configurations` Option an. Sie können diese Option entweder in den `create-event-source-mapping`-Befehl oder in den `update-event-source-mapping`-Befehl einschließen. Beispiel:

```
aws lambda create-event-source-mapping \
    ...
    --source-access-configurations
 '[{"Type":"BASIC_AUTH","URI":"arn:aws:secretsmanager:us-
west-2:123456789012:secret:DocDBSecret-AbC4E6"}]' \
    ...
```

- Sie müssen Lambda Berechtigungen zur Verwaltung von Ressourcen gewähren, die zu Ihrem Amazon-DocumentDB-Stream gehören. Fügen Sie der zu der Ausführungsrolle Ihrer Funktion manuell die folgenden Berechtigungen hinzu:
  - RDS: Beschreiben DBClusters
  - RDS:Parameter beschreiben DBCluster
  - RDS:Gruppen DBSubnet beschreiben
  - ec2: CreateNetworkInterface
  - ec2: DescribeNetworkInterfaces
  - ec2: DescribeVpcs
  - ec2: DeleteNetworkInterface
  - ec2: DescribeSubnets
  - ec2: DescribeSecurityGroups
  - kms:Decrypt
  - Verwalter von Geheimnissen: GetSecretValue
- Halten Sie die Größe der Amazon-DocumentDB-Änderungsstream-Ereignisse, die Sie an Lambda senden, bei unter 6 MB. Lambda unterstützt Nutzlasten mit einer Größe von bis zu 6 MB. Wenn Ihr Änderungsstream versucht, Lambda ein Ereignis zu senden, das größer als 6 MB ist, löscht Lambda die Nachricht und gibt die `OversizedRecordCount`-Metrik aus. Lambda gibt alle Metriken auf die bestmögliche Weise aus.

> **Note**
>
> Während Lambda-Funktionen in der Regel ein maximales Timeout-Limit von 15 Minuten haben, unterstützen Ereignisquellenzuordnungen für Amazon MSK, selbstverwaltetes Apache Kafka, Amazon DocumentDB, Amazon MQ für ActiveMQ und RabbitMQ nur Funktionen mit einem maximalen Timeout-Limit von 14 Minuten. Diese Einschränkung stellt sicher, dass

die Ereignisquellenzuordnung Funktionsfehler und Wiederholungsversuche ordnungsgemäß verarbeiten kann.

Konfigurieren Sie die Netzwerksicherheit

Um Lambda vollen Zugriff auf Amazon DocumentDB über Ihre Zuordnung von Ereignisquellen zu ermöglichen, muss Ihr Cluster entweder einen öffentlichen Endpunkt (öffentliche IP-Adresse) verwenden oder Sie müssen Zugriff auf die Amazon VPC gewähren, in der Sie den Cluster erstellt haben.

Wenn Sie Amazon DocumentDB mit Lambda verwenden, erstellen Sie AWS PrivateLink VPC-Endpunkte, die Ihrer Funktion Zugriff auf die Ressourcen in Ihrer Amazon VPC bieten.

> **Note**
>
> AWS PrivateLink VPC-Endpunkte sind für Funktionen mit Ereignisquellenzuordnungen erforderlich, die den Standardmodus (auf Abruf) für Ereignisabfragen verwenden. Wenn Ihre Ereignisquellenzuordnung den Bereitstellungsmodus verwendet, müssen Sie keine AWS PrivateLink VPC-Endpunkte konfigurieren.

Erstellen Sie einen Endpunkt, der den Zugriff auf die folgenden Ressourcen ermöglicht:

- Lambda – Erstellen Sie einen Endpunkt für den Lambda-Serviceprinzipal.
- AWS STS — Erstellen Sie einen Endpunkt für den, damit ein Service Principal eine Rolle AWS STS in Ihrem Namen übernehmen kann.
- Secrets Manager – Wenn Ihr Cluster Secrets Manager zum Speichern von Anmeldeinformationen verwendet, erstellen Sie einen Endpunkt für Secrets Manager.

Alternativ konfigurieren Sie ein NAT-Gateway auf jedem öffentlichen Subnetz in der Amazon-VPC. Weitere Informationen finden Sie unter the section called “Internetzugriff für VPC-Funktionen”.

Wenn Sie eine Ereignisquellenzuordnung für Amazon DocumentDB erstellen, prüft Lambda, ob Elastic Network Interfaces (ENIs) bereits für die für Ihre Amazon VPC konfigurierten Subnetze und Sicherheitsgruppen vorhanden sind. Wenn Lambda feststellt, dass sie vorhanden ENIs sind, versucht es, sie wiederzuverwenden. Andernfalls erstellt Lambda neue, ENIs um eine Verbindung zur Ereignisquelle herzustellen und Ihre Funktion aufzurufen.

> **Note**
>
> Lambda-Funktionen werden immer intern ausgeführt, die dem Lambda-Dienst VPCs gehören. Die VPC-Konfiguration Ihrer Funktion hat keinen Einfluss auf die Zuordnung von Ereignisquellen. Nur die Netzwerkkonfiguration der Ereignisquelle bestimmt, wie Lambda sich mit Ihrer Ereignisquelle verbindet.

Konfigurieren Sie die Sicherheitsgruppen für die Amazon VPC, die Ihren Cluster enthält. Standardmäßig verwendet Amazon DocumentDB die folgenden Ports: `27017`.

- Eingehende Regeln – Erlauben Sie den gesamten Datenverkehr auf dem Standard-Broker-Port für die Sicherheitsgruppe, die mit Ihrer Ereignisquelle verbunden ist. Alternativ können Sie eine selbstreferenzierende Regel für Sicherheitsgruppen verwenden, um den Zugriff von Instances innerhalb derselben Sicherheitsgruppe zuzulassen.
- Regeln für ausgehenden Datenverkehr — Lassen Sie den gesamten Datenverkehr über den Port `443` für externe Ziele zu, wenn Ihre Funktion mit Diensten kommunizieren muss. AWS Alternativ können Sie auch eine selbstreferenzierende Sicherheitsgruppenregel verwenden, um den Zugriff auf den Broker einzuschränken, wenn Sie nicht mit anderen Diensten kommunizieren müssen. AWS
- Regeln für eingehenden Datenverkehr auf Amazon-VPC-Endpunkten – Wenn Sie einen Amazon-VPC-Endpunkt verwenden, muss die Sicherheitsgruppe, die mit Ihrem Amazon-VPC-Endpunkt verbunden ist, eingehenden Verkehr auf Port `443` von der Cluster-Sicherheitsgruppe zulassen.

Wenn Ihr Cluster eine Authentifizierung verwendet, können Sie auch die Endpunktrichtlinie für den Secrets-Manager-Endpunkt einschränken. Für den Aufruf der Secrets-Manager-API verwendet Lambda Ihre Funktionsrolle und nicht den Lambda-Serviceprinzipal.

Example VPC-Endpunktrichtlinie – Secrets Manager-Endpunkt

```
{
      "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"
        }
    ]
}
```

Wenn Sie Amazon VPC-Endpunkte verwenden, AWS leitet Ihre API-Aufrufe zum Aufrufen Ihrer Funktion über das Elastic Network Interface (ENI) des Endpunkts weiter. Der Lambda-Serviceprinzipal muss alle Rollen und Funktionen aufrufen`lambda:InvokeFunction`, die diese ENIs verwenden.

Standardmäßig verfügen Amazon VPC-Endpunkte über offene IAM-Richtlinien, die einen umfassenden Zugriff auf Ressourcen ermöglichen. Es empfiehlt sich, diese Richtlinien auf die Durchführung der erforderlichen Aktionen über diesen Endpunkt zu beschränken. Um sicherzustellen, dass Ihre Zuordnung von Ereignisquellen Ihre Lambda-Funktion aufrufen kann, muss die VPC-Endpunktrichtlinie dem Lambda-Serviceprinzipal erlauben, `sts:AssumeRole` und `lambda:InvokeFunction` aufzurufen. Wenn Sie Ihre VPC-Endpunktrichtlinien so einschränken, dass nur API-Aufrufe zugelassen werden, die aus Ihrem Unternehmen stammen, kann die Zuordnung von Ereignisquellen nicht richtig funktionieren, weshalb in diesen Richtlinien `"Resource": "*"` erforderlich ist.

Die folgenden Beispiel-VPC-Endpunktrichtlinien zeigen, wie der erforderliche Zugriff auf den Lambda-Serviceprinzipal für die AWS STS - und Lambda-Endpunkte gewährt wird.

Example VPC-Endpunktrichtlinie — AWS STS Endpunkt

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

```
    }
```

Example VPC-Endpunktrichtlinie – Lambda-Endpunkt

```
{
      "Statement": [
          {
              "Action": "lambda:InvokeFunction",
              "Effect": "Allow",
              "Principal": {
                  "Service": [
                      "lambda.amazonaws.com"
                  ]
              },
              "Resource": "*"
          }
      ]
  }
```

Erstellen einer Zuordnung von Ereignisquellen von Amazon DocumentDB (Konsole)

Erstellen Sie für eine Funktion zum Lesen aus dem Änderungsstream eines Amazon-DocumentDB-Clusters eine Zuordnung von Ereignisquellen. In diesem Abschnitt wird die entsprechende Vorgehensweise über die Lambda-Konsole beschrieben. AWS SDK und AWS CLI Anweisungen finden Sie unterthe section called “Erstellen einer Zuordnung von Ereignisquellen von Amazon DocumentDB (SDK oder CLI)”.

So erstellen Sie eine Zuordnung von Ereignisquellen von Amazon DocumentDB (Konsole)

1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.
2. Wählen Sie den Namen einer Funktion aus.
3. Wählen Sie unter Function overview (Funktionsübersicht) die Option Add trigger (Trigger hinzufügen).
4. Wählen Sie unter Auslöserkonfiguration in der Dropdown-Liste die Option DocumentDB aus.
5. Konfigurieren Sie die erforderlichen Optionen und wählen Sie dann Add (Hinzufügen) aus.

Lambda unterstützt die folgenden Optionen für Amazon-DocumentDB-Ereignisquellen:

- DocumentDB-Cluster – Wählen Sie einen Amazon-DocumentDB-Cluster aus.
- Auslöser aktivieren – Wählen Sie aus, ob Sie den Auslöser sofort aktivieren möchten. Wenn Sie dieses Kontrollkästchen aktivieren, beginnt Ihre Funktion sofort, Datenverkehr aus dem angegebenen Amazon-DocumentDB-Änderungsstream zu empfangen, sobald die Zuordnung von Ereignisquellen erstellt wurde. Wir empfehlen, das Kontrollkästchen zu deaktivieren, um die Zuordnung von Ereignisquellen zu Testzwecken in einem deaktivierten Zustand zu erstellen. Nach der Erstellung können Sie die Zuordnung von Ereignisquellen jederzeit aktivieren.
- Datenbankname – Geben Sie den Namen einer Datenbank innerhalb des Clusters ein, die verwendet werden soll.
- (Optional) Sammlungsname – Geben Sie den Namen einer Sammlung in der Datenbank ein, die verwendet werden soll. Wenn Sie keine Sammlung angeben, hört Lambda auf alle Ereignisse aus jeder Sammlung in der Datenbank.
- Batchgröße – Legen Sie die maximale Anzahl von Nachrichten fest, die in einem einzelnen Batch abgerufen werden sollen bis zu 10 000. Die Standardgröße beträgt 100.
- Startposition – Wählen Sie die Position im Stream aus, von der aus das Lesen von Datensätzen beginnen.
  - Neueste – Verarbeiten Sie Datensätze, die neu zum Stream hinzugefügt wurden. Ihre Funktion beginnt erst mit der Verarbeitung von Datensätzen, nachdem Lambda die Erstellung Ihrer Ereignisquelle abgeschlossen hat. Das bedeutet, dass einige Datensätze gelöscht werden können, bis Ihre Ereignisquelle erfolgreich erstellt wurde.
  - Horizont trimmen – Verarbeiten Sie alle Datensätze im Stream. Lambda verwendet die Protokollspeicherdauer Ihres Clusters, um zu bestimmen, ab wo mit dem Lesen von Ereignissen begonnen werden soll. Insbesondere beginnt Lambda mit dem Lesen von `current_time - log_retention_duration`. Ihr Änderungsstream muss bereits vor diesem Zeitstempel aktiv sein, damit Lambda alle Ereignisse korrekt lesen kann.
  - Am Zeitstempel – Verarbeiten Sie Datensätze ab einem bestimmten Zeitpunkt. Ihr Änderungsstream muss bereits vor dem angegebenen Zeitstempel aktiv sein, damit Lambda alle Ereignisse korrekt lesen kann.
- Authentifizierung – Wählen Sie unter Authentifizierungsmethode den Zugriff auf den Broker in Ihrem Cluster aus.
  - BASIC_AUTH – Bei der Standardauthentifizierung müssen Sie den Secrets-Manager-Schlüssel angeben, der die Anmeldeinformationen für den Zugriff auf Ihren Cluster enthält.

- Secrets-Manager-Schlüssel – Wählen Sie den Secrets-Manager-Schlüssel aus, der die Authentifizierungsdetails (Benutzername und Passwort) enthält, die für den Zugriff auf Ihren Amazon-DocumentDB-Cluster erforderlich sind.
- (Optional) Batch-Fenster – Legen Sie die maximale Zeitspanne zur Erfassung von Datensätzen vor dem Aufruf der Funktion in Sekunden fest (bis zu 300).
- (Optional) Vollständige Dokumentkonfiguration – Wählen Sie für Dokumentaktualisierungsvorgänge aus, was Sie an den Stream senden möchten. Der Standardwert ist `Default`, was bedeutet, dass Amazon DocumentDB für jedes Änderungsstream-Ereignis nur ein Delta sendet, das die vorgenommenen Änderungen beschreibt. Weitere Informationen zu diesem Feld finden Sie FullDocumentin der MongoDB Javadoc-API-Dokumentation.
  - Standard – Lambda sendet nur ein Teildokument, das die vorgenommenen Änderungen beschreibt.
  - UpdateLookup— Lambda sendet ein Delta, das die Änderungen beschreibt, zusammen mit einer Kopie des gesamten Dokuments.

Erstellen einer Zuordnung von Ereignisquellen von Amazon DocumentDB (SDK oder CLI)

Um eine Zuordnung von Ereignisquellen von Amazon DocumentDB mit einem AWS SDK zu verwalten, können Sie die folgenden API-Operationen verwenden:

- CreateEventSourceMapping
- ListEventSourceMappings
- GetEventSourceMapping
- UpdateEventSourceMapping
- DeleteEventSourceMapping

Verwenden Sie den `create-event-source-mapping`Befehl, um die Zuordnung der AWS CLI Ereignisquelle mit dem zu erstellen. Im folgenden Beispiel wird dieser Befehl verwendet, um eine Funktion namens `my-function` einem Amazon-DocumentDB-Änderungsstream zuzuordnen. Der Ereignisquelle wird mit dem Amazon-Ressourcennamen (ARN) angegeben, mit einer Batchgröße von 500, beginnend ab dem Zeitstempel in Unix-Zeit. Der Befehl gibt auch den Secrets-Manager-Schlüssel an, den Lambda verwendet, um eine Verbindung zu Amazon DocumentDB herzustellen.

Darüber hinaus enthält er `document-db-event-source-config`-Parameter, die die Datenbank und die Sammlung angeben, aus der gelesen werden soll.

```
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"}' \
```

Die Ausgabe sollte in etwa wie folgt aussehen:

```
{
    "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"
}
```

Nach der Erstellung können Sie den `update-event-source-mapping`-Befehl verwenden, um die Einstellungen für Ihre Amazon-DocumentDB-Ereignisquelle zu aktualisieren. Im folgenden Beispiel werden die Batchgröße auf 1 000 und das Batchfenster auf 10 Sekunden aktualisiert. Für diesen Befehl benötigen Sie die UUID Ihrer Zuordnung von Ereignisquellen, die Sie über den `list-event-source-mapping`-Befehl oder die Lambda-Konsole abrufen können.

```
aws lambda update-event-source-mapping --function-name my-function \
```

```
    --uuid f89f8514-cdd9-4602-9e1f-01a5b77d449b \
    --batch-size 1000 \
    --batch-window 10
```

Die Ausgabe sollte in etwa wie folgt aussehen:

```
{
    "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"
}
```

Da Lambda die Einstellungen asynchron aktualisiert, werden Sie diese Änderungen möglicherweise erst nach Abschluss des Vorgangs in der Ausgabe sehen. Verwenden Sie den `get-event-source-mapping`-Befehl, um die aktuellen Einstellungen Ihrer Zuordnung von Ereignisquellen anzuzeigen.

```
aws lambda get-event-source-mapping --uuid f89f8514-cdd9-4602-9e1f-01a5b77d449b
```

Die Ausgabe sollte in etwa wie folgt aussehen:

```
{
    "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"
}
```

Um Ihre Zuordnung von Ereignisquellen von Amazon DocumentDB zu löschen, verwenden Sie den `delete-event-source-mapping`-Befehl.

```
aws lambda delete-event-source-mapping \
    --uuid 2b733gdc-8ac3-cdf5-af3a-1827b3b11284
```

Startpositionen für Abfragen und Streams

Beachten Sie, dass die Stream-Abfrage bei der Erstellung und Aktualisierung der Zuordnung von Ereignisquellen letztendlich konsistent ist.

- Bei der Erstellung der Zuordnung von Ereignisquellen kann es mehrere Minuten dauern, bis mit der Abfrage von Ereignissen aus dem Stream begonnen wird.
- Bei Aktualisierungen der Zuordnung von Ereignisquellen kann es mehrere Minuten dauern, bis die Abfrage von Ereignissen aus dem Stream gestoppt und neu gestartet wird.

Dieses Verhalten bedeutet, dass, wenn Sie `LATEST` als Startposition für den Stream angeben, die Zuordnung von Ereignisquellen bei der Erstellung oder Aktualisierung möglicherweise Ereignisse übersieht. Um sicherzustellen, dass keine Ereignisse übersehen werden, geben Sie die Startposition des Streams als `TRIM_HORIZON` oder `AT_TIMESTAMP` an.

Überwachen Ihrer Amazon-DocumentDB-Ereignisquelle

Um Sie bei der Überwachung Ihrer Amazon-DocumentDB-Ereignisquelle zu unterstützen, gibt Lambda die Metrik `IteratorAge` aus, wenn die Funktion das Verarbeiten von Datensätzen abgeschlossen hat. Das Iterator-Alter ist der Unterschied zwischen dem Zeitstempel des letzten Ereignisses und dem aktuellen Zeitstempel. Im Wesentlichen gibt die Metrik `IteratorAge` an, wie alt der letzte Datensatz im Batch bei Fertigstellung der Verarbeitung war. Wenn Ihre Funktion derzeit neue Ereignisse verarbeitet, können Sie mit dem Iterator-Alter die Latenz zwischen dem Zeitpunkt,

zu dem ein Datensatz hinzugefügt wird, und dem Zeitpunkt, zu dem die Funktion verarbeitet wird, schätzen. Eine ansteigende Tendenz bei `IteratorAge` kann auf Probleme mit Ihrer Funktion hindeuten. Weitere Informationen finden Sie unter Verwendung von CloudWatch-Metriken mit Lambda.

Amazon DocumentDB-Change-Streams sind nicht für die Bewältigung großer Zeitlücken zwischen Ereignissen optimiert. Wenn Ihre Amazon DocumentDB-Ereignisquelle über einen längeren Zeitraum keine Ereignisse empfängt, kann Lambda die Zuordnung von Ereignisquellen deaktivieren. Die Dauer dieses Zeitraums kann je nach Größe des Clusters und anderen Workloads zwischen einigen Wochen und einigen Monaten variieren.

Lambda unterstützt Nutzlasten von bis zu 6 MB. Änderungsstream-Ereignisse von Amazon DocumentDB können jedoch bis zu 16 MB groß sein. Wenn Ihr Änderungsstream versucht, Lambda ein Änderungsstream-Ereignis zu senden, das größer als 6 MB ist, löscht Lambda die Nachricht und gibt die Metrik `OversizedRecordCount` aus. Lambda gibt alle Metriken auf die bestmögliche Weise aus.

Tutorial: Verwendung AWS Lambda mit Amazon DocumentDB DocumentDB-Streams

In diesem Tutorial erstellen Sie eine einfache Lambda-Funktion, die Ereignisse aus einem Amazon DocumentDB (mit MongoDB-Kompatibilität) Change-Stream konsumiert. Um dieses Tutorial abzuschließen, werden Sie die folgenden Phasen durchlaufen:

- Richten Sie Ihren Amazon-DocumentDB-Cluster ein, stellen Sie eine Verbindung zu ihm her und aktivieren Sie Change-Streams darauf.
- Erstellen Sie Ihre Lambda-Funktion und konfigurieren Sie Ihren Amazon-DocumentDB-Cluster als Ereignisquelle für Ihre Funktion.
- Testen Sie die Einrichtung, indem Sie Elemente in Ihre Amazon-DocumentDB-Datenbank einfügen.

Erstellen Sie den Amazon DocumentDB-Cluster

1. Öffnen Sie die Amazon DocumentDB-Konsole. Wählen Sie unter Cluster die Option Erstellen aus.
2. Erstellen Sie einen Cluster mit der folgenden Konfiguration:
   - Wählen Sie als Cluster-Typ die Option Instance-basierter Cluster aus. Dies ist die Standardoption.

- Stellen Sie sicher, dass unter Clusterkonfiguration die Engine-Version 5.0.0 ausgewählt ist. Dies ist die Standardoption.
- Unter Instance-Konfiguration:
  - Wählen Sie für DB-Instance-Klasse die Option Speicheroptimierte Klassen aus. Dies ist die Standardoption.
  - Wählen Sie für Anzahl der regulären Replikat-Instances den Wert 1 aus.
  - Verwenden Sie für Instance-Klasse die Standardauswahl.
- Geben Sie unter Authentifizierung einen Benutzernamen für den Hauptbenutzer ein und wählen Sie dann Selbstverwaltet aus. Geben Sie ein Passwort ein und bestätigen Sie es.
- Behalten Sie die alle Standardeinstellung bei.

3. Wählen Sie Cluster erstellen.

Erstellen des Secrets in Secrets Manager

Während Amazon DocumentDB Ihren Cluster erstellt, erstellen Sie ein AWS Secrets Manager Geheimnis, um Ihre Datenbankanmeldedaten zu speichern. Sie geben diesen geheimen Schlüssel an, wenn Sie in einem späteren Schritt die Zuordnung von Ereignisquellen in Lambda erstellen.

Um das Secret in Secrets Manager zu erstellen

1. Öffnen Sie die Secrets-Manager-Konsole und wählen Sie Neues Secret speichern.
2. Wählen Sie für Secret-Typ auswählen eine der folgenden Optionen aus:
   - Unter Grundlegende Details:
     - Secret-Typ: Anmeldeinformationen für die Amazon-DocumentDB-Datenbank
     - Geben Sie unter Anmeldeinformationen den Benutzernamen und das Passwort ein, den bzw. das Sie zum Erstellen Ihres Amazon-DocumentDB-Clusters verwendet haben.
     - Datenbank: Wählen Sie Ihren Amazon DocumentDB-Cluster aus.
     - Wählen Sie Weiter aus.
3. Wählen Sie unter Secret konfigurieen eine der folgenden Optionen aus:
   - Geheimer Name: `DocumentDBSecret`
   - Wählen Sie Weiter aus.
4. Wählen Sie Weiter aus.

5. Wählen Sie Store (Speichern) aus.
6. Aktualisieren Sie die Konsole, um zu überprüfen, ob Sie das `DocumentDBSecret`-Secret erfolgreich gespeichert haben.

Notieren Sie sich den geheimen ARN. Sie werden es in einem späteren Schritt noch benötigen.

Verbinden mit dem Cluster

Connect zu Ihrem Amazon DocumentDB-Cluster her mit AWS CloudShell

1. Suchen Sie in der Amazon-DocumentDB-Verwaltungskonsole unter Cluster den von Ihnen erstellten Cluster. Wählen Sie Ihren Cluster aus, indem Sie das Kontrollkästchen daneben aktivieren.
2. Wählen Sie Mit Cluster verbinden aus. Der Befehlsbildschirm CloudShell Ausführen wird angezeigt.
3. Geben Sie im Feld Neuer Umgebungsname einen eindeutigen Namen ein, z. B. „Test“, und wählen Sie Erstellen und ausführen aus.
4. Geben Sie bei der Aufforderung Ihr Passwort ein. Wenn der Prompt zu `rs0 [direct: primary] <env-name>>` wechselt, sind Sie erfolgreich mit Ihrem Amazon-DocumentDB-Cluster verbunden.

Change-Streams aktivieren

In diesem Tutorial verfolgen Sie Änderungen an der `products`-Sammlung der `docdbdemo`-Datenbank in Ihrem Amazon DocumentDB-Cluster. Sie tun dies, indem Sie Change-Streams aktivieren.

Um eine neue Datenbank in Ihrem Cluster zu erstellen

1. Führen Sie den folgenden Befehl aus, um eine neue Datenbank mit dem Namen `docdbdemo` zu erstellen:

```
use docdbdemo
```

2. Geben Sie im Terminalfenster den folgenden Befehl ein, um einen Datensatz in `docdbdemo` einzufügen:

```
db.products.insertOne({"hello":"world"})
```

Sie sollten eine Ausgabe wie die Folgende sehen:

```
{
  acknowledged: true,
  insertedId: ObjectId('67f85066ca526410fd531d59')
}
```

3. Dann aktivieren Sie mit dem folgenden Befehl Change-Streams für die `products`-Sammlung der `docdbdemo`-Datenbank:

```
db.adminCommand({modifyChangeStreams: 1,
    database: "docdbdemo",
    collection: "products",
    enable: true});
```

Die Ausgabe sollte in etwa wie folgt aussehen:

```
{ "ok" : 1, "operationTime" : Timestamp(1680126165, 1) }
```

Schnittstellen-VPC-Endpunkte erstellen

Erstellen Sie als Nächstes Schnittstellen-VPC-Endpunkte, um sicherzustellen, dass Lambda und Secrets Manager (der später zum Speichern unserer Cluster-Anmeldeinformationen verwendet wird) eine Verbindung zu Ihrer Standard-VPC herstellen können.

Um Schnittstellen-VPC-Endpunkte zu erstellen

1. Öffnen Sie die VPC;-Konsole. Wählen Sie im linken Menü unter Virtual Private Cloud die Option Endpunkte aus.
2. Wählen Sie Endpunkt erstellen aus. Erstellen Sie einen Endpunkt mit der folgenden Konfiguration:
   - Geben Sie für Name Tag `lambda-default-vpc` ein.
   - Wählen Sie als Dienstkategorie die Option AWS Dienste aus.

- Geben Sie unter Services `lambda` in das Suchfeld ein. Wählen Sie den Service im Format `com.amazonaws.<region>.lambda`.
- Wählen Sie für VPC die VPC aus, in der sich Ihr Amazon-DocumentDB-Cluster befindet. Dies ist in der Regel die Standard-VPC.
- Aktivieren Sie für Subnetze die Kästchen neben jeder Availability Zone. Wählen Sie die richtige Subnetz-ID für jede Availability Zone.
- Wählen Sie als IP-Adresstyp die Option aus IPv4.
- Wählen Sie für Sicherheitsgruppen die Sicherheitsgruppe aus, die Ihr Amazon-DocumentDB-Cluster verwendet. Dies ist in der Regel die `default`-Sicherheitsgruppe.
- Behalten Sie die alle Standardeinstellung bei.
- Wählen Sie Endpunkt erstellen aus.

3. Wählen Sie erneut Endpunkt erstellen. Erstellen Sie einen Endpunkt mit der folgenden Konfiguration:

   - Geben Sie für Name Tag `secretsmanager-default-vpc` ein.
   - Wählen Sie als Dienstkategorie die Option AWS Dienste aus.
   - Geben Sie unter Services `secretsmanager` in das Suchfeld ein. Wählen Sie den Service im Format `com.amazonaws.<region>.secretsmanager`.
   - Wählen Sie für VPC die VPC aus, in der sich Ihr Amazon-DocumentDB-Cluster befindet. Dies ist in der Regel die Standard-VPC.
   - Aktivieren Sie für Subnetze die Kästchen neben jeder Availability Zone. Wählen Sie die richtige Subnetz-ID für jede Availability Zone.
   - Wählen Sie als IP-Adresstyp die Option aus IPv4.
   - Wählen Sie für Sicherheitsgruppen die Sicherheitsgruppe aus, die Ihr Amazon-DocumentDB-Cluster verwendet. Dies ist in der Regel die `default`-Sicherheitsgruppe.
   - Behalten Sie die alle Standardeinstellung bei.
   - Wählen Sie Endpunkt erstellen aus.

Damit ist der Abschnitt zur Einrichtung eines Clusters in diesem Tutorial abgeschlossen.

Erstellen der Ausführungsrolle

In den nächsten Schritten erstellen Sie Ihre Lambda-Funktion. Zunächst müssen Sie die Ausführungsrolle erstellen, die Ihrer Funktion die Berechtigung zum Zugriff auf Ihren Cluster gibt. Dazu erstellen Sie zunächst eine IAM-Richtlinie und verknüpfen diese dann mit einer IAM-Rolle.

So erstellen Sie eine IAM-Richtlinie

1. Öffnen Sie in der IAM-Konsole die Seite Richtlinien und wählen Sie dann Richtlinie erstellen aus.
2. Wählen Sie den Tab JSON. Ersetzen Sie in der folgenden Richtlinie den ARN der Secrets-Manager-Ressource in der letzten Zeile der Anweisung durch Ihren geheimen ARN von zuvor, und kopieren Sie die Richtlinie in den Editor.

   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. Wählen Sie Weiter: Tags und dann Weiter: Prüfen aus.
4. Geben Sie unter Name `AWSDocumentDBLambdaPolicy` ein.
5. Wählen Sie Richtlinie erstellen aus.

So erstellen Sie die IAM-Rolle

1. Öffnen Sie die Seite Rollen in der IAM-Konsole und wählen Sie Rolle erstellen.
2. Wählen Sie für Vertrauenswürdige Entität auswählen die folgenden Optionen:
   - Typ der vertrauenswürdigen Entität: AWS Dienst
   - Service oder Anwendungsfall: Lambda
   - Wählen Sie Weiter aus.
3. Wählen Sie für Berechtigungen hinzufügen die `AWSDocumentDBLambdaPolicy` Richtlinie aus, die Sie gerade erstellt haben, sowie die, `AWSLambdaBasicExecutionRole` um Ihrer Funktion Berechtigungen zum Schreiben in Amazon CloudWatch Logs zu erteilen.
4. Wählen Sie Weiter aus.
5. Geben Sie für Role name (Rollenname) den Namen `AWSDocumentDBLambdaExecutionRole` ein.
6. Wählen Sie Rolle erstellen aus.

So erstellen Sie die Lambda-Funktion:

Dieses Tutorial verwendet die Python 3.14-Laufzeit, aber wir haben auch Beispielcodedateien für andere Laufzeiten bereitgestellt. Sie können die Registerkarte im folgenden Feld auswählen, um den Code für die gewünschte Laufzeit anzusehen.

Der Code empfängt eine Amazon-DocumentDB-Ereigniseingabe und verarbeitet die darin enthaltene Nachricht.

So erstellen Sie die Lambda-Funktion:

1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.
2. Wählen Sie Funktion erstellen.
3. Wählen Sie Ohne Vorgabe erstellen aus.
4. Führen Sie unter Basic information (Grundlegende Informationen) die folgenden Schritte aus:
    a. Geben Sie unter Funktionsname `ProcessDocumentDBRecords` ein.
    b. Wählen Sie für Runtime Python 3.14.
    c. Wählen Sie für Architektur x86_64 aus.
5. Gehen Sie auf der Registerkarte Standard-Ausführungsrolle ändern wie folgt vor:
    a. Erweitern Sie die Registerkarte und wählen Sie dann Verwenden einer vorhandenen Rolle aus.
    b. Wählen Sie die zuvor erstellte `AWSDocumentDBLambdaExecutionRole` aus.
6. Wählen Sie Funktion erstellen.

So stellen Sie den Funktionscode bereit

1. Wählen Sie im folgenden Feld die Registerkarte Python und kopieren Sie den Code.

.NET

SDK für .NET

> **Note**
>
> Es gibt noch mehr dazu. GitHub Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines Amazon DocumentDB-Ereignisses mit Lambda unter Verwendung von .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

SDK für Go V2

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines Amazon DocumentDB-Ereignisses mit Lambda unter Verwendung von 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 für Java 2.x

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines Amazon DocumentDB-Ereignisses mit Lambda unter Verwendung von 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 für JavaScript (v3)

> **Note**
>
> Es gibt noch mehr dazu GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Verwenden eines Amazon DocumentDB DocumentDB-Ereignisses mit Lambda. 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));
};
```

Ein Amazon DocumentDB DocumentDB-Ereignis mit Lambda verwenden 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

SDK für PHP

> **Note**
>
> Es gibt noch mehr dazu. GitHub Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines Amazon DocumentDB-Ereignisses mit Lambda unter Verwendung von 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

SDK für Python (Boto3)

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines Amazon DocumentDB-Ereignisses mit Lambda unter Verwendung von 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

SDK für Ruby

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines Amazon DocumentDB-Ereignisses mit Lambda unter Verwendung von 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 für Rust

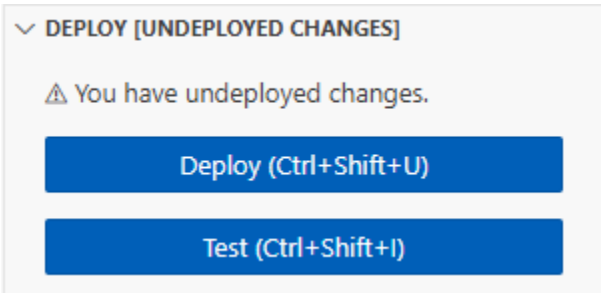
> **Note**
>
> Es gibt noch mehr dazu GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines Amazon DocumentDB-Ereignisses mit Lambda unter Verwendung von 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. Fügen Sie den Code im Code-Quellbereich der Lambda-Konsole in den Code-Editor ein und ersetzen Sie dabei den von Lambda erstellten Code.
3. Wählen Sie im Abschnitt DEPLOY die Option Deploy aus, um den Code Ihrer Funktion zu aktualisieren:



Erstellen Sie die Zuordnung von Ereignisquellen in Lambda

Erstellen Sie die Zuordnung von Ereignisquellen, die Ihren Amazon DocumentDB-Change-Stream mit Ihrer Lambda-Funktion verknüpft. Nachdem Sie diese Ereignisquellenzuordnung erstellt haben, wird AWS Lambda sofort mit der Abfrage des Streams begonnen.

Die Zuordnung von Ereignisquellen erstellen

1. Öffnen Sie die Seite Funktionen in der Lambda-Konsole.
2. Wählen Sie die `ProcessDocumentDBRecords`-Funktion aus, die Sie zuvor erstellt haben.
3. Wählen Sie die Registerkarte Konfiguration und dann im linken Menü die Option Auslöser aus.
4. Wählen Sie Add trigger.
5. Wählen Sie unter Trigger-Konfiguration als Quelle die Option Amazon DocumentDB aus.
6. Erstellen Sie die Zuordnung von Ereignisquellen mit der folgenden Konfiguration:
   - Amazon-DocumentDB-Cluster: Wählen Sie den Cluster aus, den Sie zuvor erstellt haben.
   - Datenbankname: docdbdemo
   - Name der Kollektion:Produkte
   - Chargengröße: 1
   - Startposition: Neueste
   - Authentifizierung: BASIC_AUTH
   - Secrets-Manager-Schlüssel: Wählen Sie das Secret für Ihren Amazon-DocumentDB-Cluster aus. Der Name ist ähnlich wie `rds!cluster-12345678-a6f0-52c0-b290-db4aga89274f`.
   - Batchfenster: 1
   - Vollständige Konfiguration des Dokuments: UpdateLookup

7. Wählen Sie Hinzufügen aus. Die Erstellung Ihrer Zuordnung von Ereignisquellen kann einige Minuten dauern.

Testen der -Funktion

Warten Sie, bis die Zuordnung von Ereignisquellen den Status Aktiviert erreicht hat. Dies kann mehrere Minuten dauern. Testen Sie dann das end-to-end Setup, indem Sie Datenbankeinträge einfügen, aktualisieren und löschen. Bevor Sie beginnen:

1. Stellen Sie erneut eine Verbindung zu Ihrem Amazon DocumentDB-Cluster in Ihrer CloudShell Umgebung her.
2. Führen Sie den folgenden Befehl aus, um sicherzustellen, dass Sie die `docdbdemo`-Datenbank verwenden:

```
use docdbdemo
```

Einfügen eines Datensatzes

Fügen Sie einen Datensatz in die `products`-Sammlung der `docdbdemo`-Datenbank ein:

```
db.products.insertOne({"name":"Pencil", "price": 1.00})
```

Überprüfen Sie anhand von Logs, ob Ihre Funktion dieses Ereignis erfolgreich verarbeitet hat. CloudWatch Sie sollten einen Protokolleintrag wie diesen sehen:

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	}

Aktualisieren von Datensätzen

Aktualisieren Sie den gerade eingefügten Datensatz mit dem folgenden Befehl:

```
db.products.updateOne(
    { "name": "Pencil" },
    { $set: { "price": 0.50 }}
)
```

Überprüfen Sie, ob Ihre Funktion dieses Ereignis erfolgreich verarbeitet hat, indem Sie die Option CloudWatch Logs überprüfen. Sie sollten einen Protokolleintrag wie diesen sehen:

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	}

Einen Datensatz löschen

Löschen Sie den Datensatz, den Sie gerade aktualisiert haben, mit dem folgenden Befehl:

```
db.products.deleteOne( { "name": "Pencil" } )
```

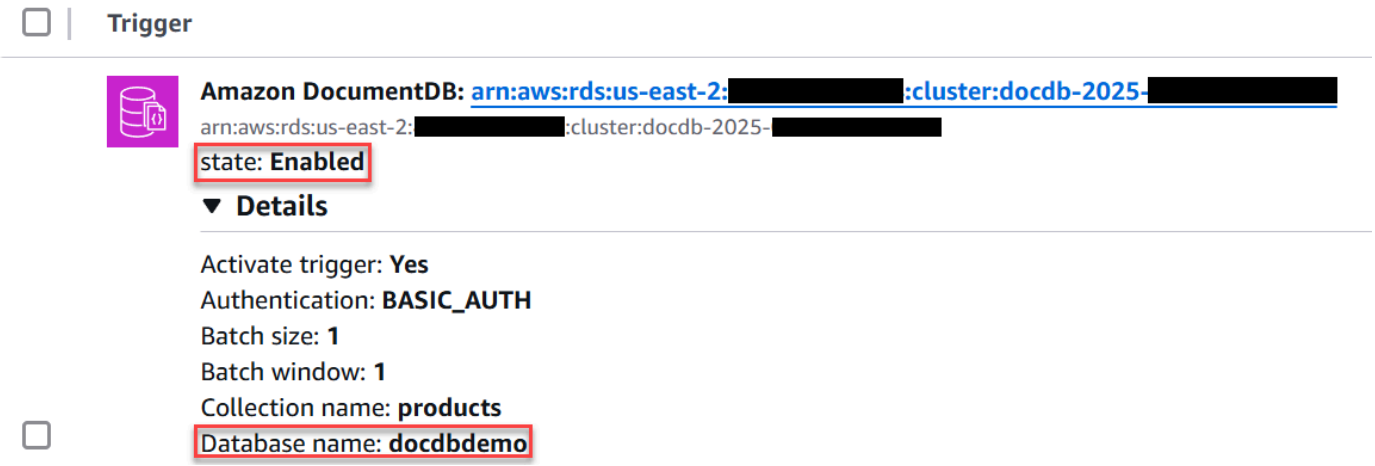
Überprüfen Sie anhand von CloudWatch Logs, ob Ihre Funktion dieses Ereignis erfolgreich verarbeitet hat. Sie sollten einen Protokolleintrag wie diesen sehen:

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: {}

Fehlerbehebung

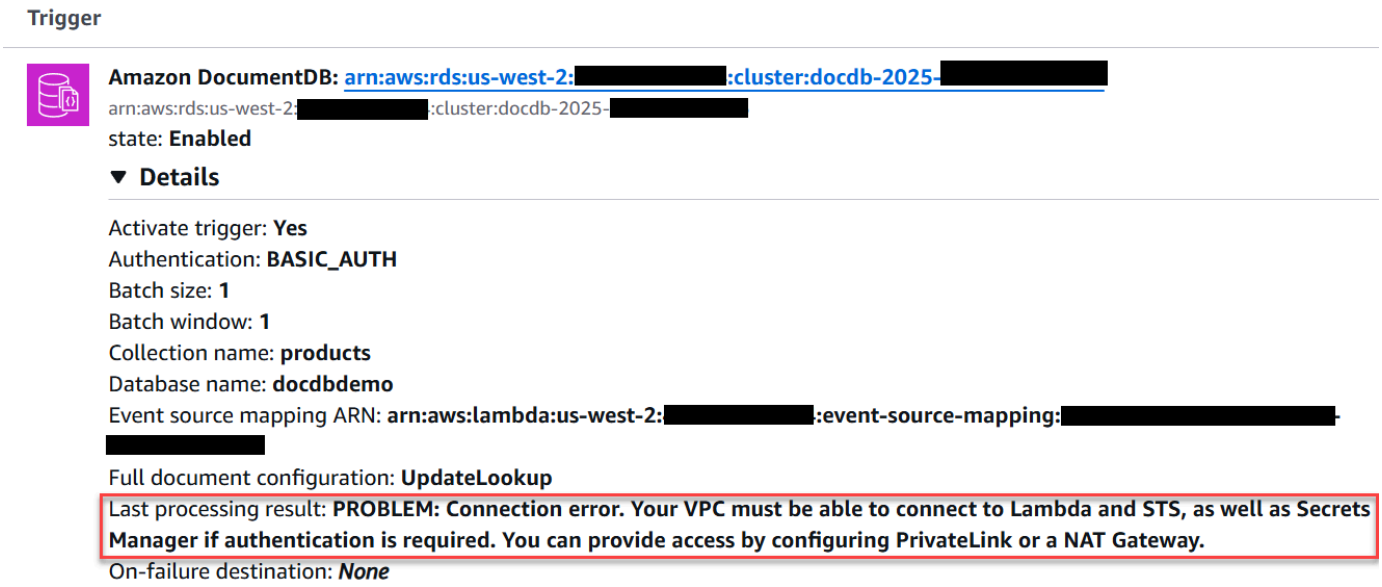
Wenn Sie in den CloudWatch Protokollen Ihrer Funktion keine Datenbankereignisse sehen, überprüfen Sie Folgendes:

- Stellen Sie sicher, dass sich die Zuordnung von Ereignisquellen in Lambda (auch als Auslöser bezeichnet) im Status Aktiviert befindet. Die Zuordnung von Ereignisquellen kann mehrere Minuten in Anspruch nehmen.
- Wenn die Zuordnung der Ereignisquellen aktiviert ist, Sie aber immer noch keine Datenbankereignisse sehen in CloudWatch:
  - Stellen Sie sicher, dass der Datenbankname in der Zuordnung von Ereignisquellen auf `docdbdemo` eingestellt ist.



  - Überprüfen Sie das Feld Letztes Verarbeitungsergebnis der Zuordnung von Ereignisquellen auf die folgende Meldung: „PROBLEM: Verbindungsfehler. Ihre VPC muss eine Verbindung zu Lambda und STS sowie zu Secrets Manager herstellen können, wenn eine Authentifizierung erforderlich ist.“ Wenn diese Fehlermeldung angezeigt wird, stellen Sie sicher, dass Sie die

Lambda- und Secrets-Manager-VPC-Schnittstellenendpunkte erstellt haben und dass die Endpunkte dieselbe VPC und dieselben Subnetze verwenden wie Ihr Amazon-DocumentDB-Cluster.



Bereinigen Ihrer Ressourcen

Sie können jetzt die Ressourcen, die Sie für dieses Tutorial erstellt haben, löschen, es sei denn, Sie möchten sie behalten. Durch das Löschen von AWS -Ressourcen, die Sie nicht mehr verwenden, können Sie verhindern, dass unnötige Gebühren in Ihrem AWS-Konto-Konto anfallen.

So löschen Sie die Lambda-Funktion:

1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.
2. Wählen Sie die Funktion aus, die Sie erstellt haben.
3. Wählen Sie Aktionen, Löschen aus.
4. Geben Sie **`confirm`** in das Texteingabefeld ein und wählen Sie Delete (Löschen) aus.

So löschen Sie die Ausführungsrolle

1. Öffnen Sie die Seite Roles in der IAM-Konsole.
2. Wählen Sie die von Ihnen erstellte Ausführungsrolle aus.
3. Wählen Sie Löschen aus.

4. Geben Sie den Namen der Rolle in das Texteingabefeld ein und wählen Sie Delete (Löschen) aus.

Löschen von VPC-Endpunkten

1. Öffnen Sie die VPC;-Konsole. Wählen Sie im linken Menü unter Virtual Private Cloud die Option Endpunkte aus.
2. Wählen Sie die Endpunkte aus, die Sie erstellt haben.
3. Wählen Sie Actions (Aktionen), Delete VPC Endpoint (VPC-Endpunkte löschen).
4. Geben Sie **`delete`** in das Texteingabefeld ein.
5. Wählen Sie Löschen aus.

So löschen Sie den Amazon DocumentDB-Cluster

1. Öffnen Sie die Amazon DocumentDB-Konsole.
2. Wählen Sie den Amazon DocumentDB-Cluster aus, den Sie für dieses Tutorial erstellt haben, und deaktivieren Sie den Löschschutz.
3. Wählen Sie auf der Cluster-Hauptseite erneut Ihren Amazon DocumentDB-Cluster aus.
4. Wählen Sie Aktionen, Löschen aus.
5. Wählen Sie für Finalen Cluster-Snapshot erstellen die Option Nein aus.
6. Geben Sie **`delete`** in das Texteingabefeld ein.
7. Wählen Sie Löschen aus.

So löschen Sie das Secret im Secrets Manager

1. Öffnen Sie die Secrets Manager-Konsole.
2. Wählen Sie das Secret aus, das Sie für dieses Tutorial erstellt haben.
3. Wählen Sie Aktionen, Secret löschen.
4. Wählen Sie Schedule deletion.

Verwenden von AWS Lambda mit Amazon DynamoDB

> **Note**
>
> Wenn Sie Daten an ein anderes Ziel als eine Lambda-Funktion senden oder die Daten vor dem Senden anreichern möchten, finden Sie weitere Informationen unter Amazon-EventBridge-Pipes.

Sie können eine AWS Lambda-Funktion verwenden, um Datensätze in einem Amazon-DynamoDB-Stream zu verarbeiten. Mit DynamoDB Streams können Sie eine Lambda-Funktion auslösen, damit jedes Mal, wenn eine DynamoDB-Tabelle aktualisiert wird, weitere Aufgaben ausgeführt werden.

Bei der Verarbeitung von DynamoDB-Streams müssen Sie eine Logik für teilweise Batch-Antworten implementieren, um zu verhindern, dass versucht wird, erfolgreich verarbeitete Datensätze erneut zu verarbeiten, wenn einige Datensätze in einem Batch fehlschlagen. Das Batch-Prozessor-Dienstprogramm von Powertools for AWS Lambda ist in Python, TypeScript, .NET und Java verfügbar und vereinfacht diese Implementierung, indem es automatisch die Logik für teilweise Batch-Antworten verarbeitet, wodurch die Entwicklungszeit reduziert und die Zuverlässigkeit verbessert wird.

Themen

- Abrufen und Stapeln von Streams
- Startpositionen für Abfragen und Streams
- Gleichzeitige Leser eines Shard in DynamoDB Streams
- Beispielereignis
- DynamoDB-Datensätze mit Lambda verarbeiten
- Konfigurieren einer teilweisen Batch-Antwort mit DynamoDB und Lambda
- Verworfene Datensätze für eine DynamoDB-Ereignisquelle in Lambda aufbewahren
- Implementierung der statusbehafteten DynamoDB-Stream-Verarbeitung in Lambda
- Lambdaparameter für Amazon DynamoDB-SQS-Zuordnungen von Ereignisquellen
- Verwendung der Ereignisfilterung mit einer DynamoDB-Ereignisquelle
- Tutorial: Verwendung AWS Lambda mit Amazon DynamoDB DynamoDB-Streams

Abrufen und Stapeln von Streams

Lambda fragt Shards in Ihrem DynamoDB-Stream nach Datensätzen bei einer Basisrate von viermal pro Sekunde ab. Sind Datensätze verfügbar, ruft Lambda Ihre Funktion auf und wartet auf das Ergebnis. Ist die Verarbeitung erfolgreich, setzt Lambda die Abrufe fort, bis es weitere Datensätze erhält.

Standardmäßig ruft Lambda Ihre Funktion auf, sobald Datensätze verfügbar sind. Wenn der Batch, den Lambda aus der Ereignisquelle liest, nur einen Datensatz enthält, sendet Lambda nur einen Datensatz an die Funktion. Damit die Funktion nicht mit einer kleinen Anzahl von Datensätzen aufgerufen wird, können Sie die Ereignisquelle anweisen, Datensätze bis zu 5 Minuten lang zu puffern, indem Sie ein Batch-Fenster konfigurieren. Bevor die Funktion aufgerufen wird, liest Lambda so lange Datensätze aus der Ereignisquelle, bis es einen vollständigen Batch erfasst hat, das Batch-Verarbeitungsfenster abläuft oder der Batch die Nutzlastgrenze von 6 MB erreicht. Weitere Informationen finden Sie unter Batching-Verhalten.

> **Warning**
>
> Zuordnung von Lambda-Ereignisquellen verarbeiten jedes Ereignis mindestens einmal und es kann zu einer doppelten Verarbeitung von Datensätzen kommen. Um mögliche Probleme im Zusammenhang mit doppelten Ereignissen zu vermeiden, empfehlen wir Ihnen dringend, Ihren Funktionscode idempotent zu machen. Weitere Informationen finden Sie unter Wie mache ich meine Lambda-Funktion idempotent im AWS-Wissenszentrum.

Lambda wartet mit dem Senden des nächsten zu verarbeitenden Stapels nicht, bis ggf. konfigurierte Erweiterungen abgeschlossen sind. Anders ausgedrückt: Ihre Erweiterungen werden möglicherweise weiter ausgeführt, während Lambda den nächsten Stapel von Datensätzen verarbeitet. Dies kann zu Drosselungsproblemen führen, wenn Sie gegen eine Einstellung oder gegen einen Grenzwert im Zusammenhang mit der Parallelität Ihres Kontos verstoßen. Um zu erkennen, ob möglicherweise ein Problem vorliegt, müssen Sie Ihre Funktionen überwachen sowie überprüfen, ob für Ihre Zuordnung von Ereignisquellen unerwartet hohe Parallelitätsmetriken vorliegen. Aufgrund der kurzen Zeit zwischen den Aufrufen kann Lambda kurzzeitig eine höhere Gleichzeitigkeitsnutzung als die Anzahl der Shards melden. Dies kann sogar für Lambda-Funktionen ohne Erweiterungen gelten.

Konfigurieren Sie die ParallelizationFactor-Einstellung, um einen Shard eines DynamoDB-Stroms mit mehr als einem Lambda-Aufruf gleichzeitig zu verarbeiten. Sie können die Anzahl der gleichzeitigen Batches angeben, die Lambda von einem Shard über einen Parallelisierungsfaktor von 1 (Standard)

bis 10 abfragt. Wenn `ParallelizationFactor` beispielsweise auf 2 gesetzt ist, können Sie maximal 200 gleichzeitige Lambda-Aufrufe haben, um 100 DynamoDB-Stream-Shards zu verarbeiten (in der Praxis werden womöglich andere Werte für die Metrik `ConcurrentExecutions` angezeigt). Dies hilft, den Verarbeitungsdurchsatz hochzuskalieren, wenn das Datenvolumen flüchtig ist und IteratorAge hoch ist. Wenn Sie die Anzahl der gleichzeitigen Stapel pro Shard erhöhen, sorgt Lambda weiterhin für eine ordnungsgemäße Verarbeitung auf der Ebene der Elemente (Partition und Sortierschlüssel).

Startpositionen für Abfragen und Streams

Beachten Sie, dass die Stream-Abfrage bei der Erstellung und Aktualisierung der Zuordnung von Ereignisquellen letztendlich konsistent ist.

- Bei der Erstellung der Zuordnung von Ereignisquellen kann es mehrere Minuten dauern, bis mit der Abfrage von Ereignissen aus dem Stream begonnen wird.
- Bei Aktualisierungen der Zuordnung von Ereignisquellen kann es mehrere Minuten dauern, bis die Abfrage von Ereignissen aus dem Stream gestoppt und neu gestartet wird.

Dieses Verhalten bedeutet, dass, wenn Sie `LATEST` als Startposition für den Stream angeben, die Zuordnung von Ereignisquellen bei der Erstellung oder Aktualisierung möglicherweise Ereignisse übersieht. Um sicherzustellen, dass keine Ereignisse übersehen werden, geben Sie die Stream-Startposition als `TRIM_HORIZON` an.

Gleichzeitige Leser eines Shard in DynamoDB Streams

Für Einzelregionstabellen, bei denen es sich nicht um globale Tabellen handelt, können Sie bis zu zwei Lambda-Funktionen entwerfen, die gleichzeitig aus demselben DynamoDB-Streams-Shard lesen. Eine Überschreitung dieses Grenzwerts kann zu einer Anforderungsdrosselung führen. Für globale Tabellen empfehlen wir, die Anzahl der gleichzeitigen Funktionen auf eine zu beschränken, um eine Anforderungsdrosselung zu vermeiden.

Beispielereignis

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"
    }
  ]}
```

DynamoDB-Datensätze mit Lambda verarbeiten

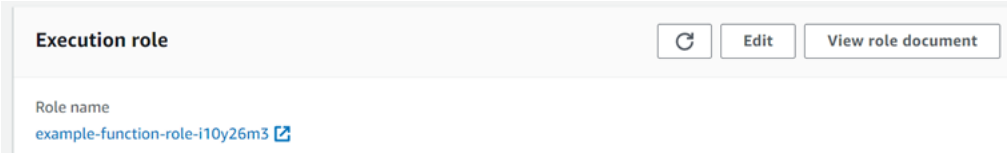
Erstellen Sie eine Ereignisquellenzuordnung, um Lambda anzuweisen, Datensätze aus Ihrem Stream an eine Lambda-Funktion zu senden. Sie können mehrere Ereignisquellen-Zuweisung erstellen, um gleiche Daten mit mehreren Lambda-Funktionen oder Elemente aus mehreren Streams mit nur einer Funktion zu verarbeiten.

Sie können Zuordnungen von Ereignisquellen konfigurieren, um Datensätze aus einem Stream in einem anderen AWS-Konto zu verarbeiten. Weitere Informationen hierzu finden Sie unter the section called “Kontenübergreifende Zuordnungen”.

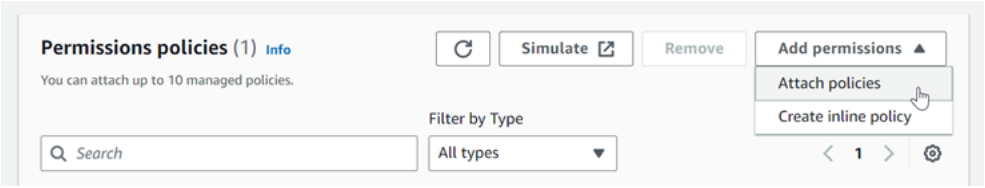
Um Ihre Funktion so zu konfigurieren, dass sie aus DynamoDB Streams liest, fügen Sie die AWS verwaltete AWSLambdaDBExecutionDynamo-Rollenrichtlinie an Ihre Ausführungsrolle an und erstellen Sie dann einen DynamoDB-Trigger.

So fügen Sie Berechtigungen hinzu und erstellen einen Auslöser

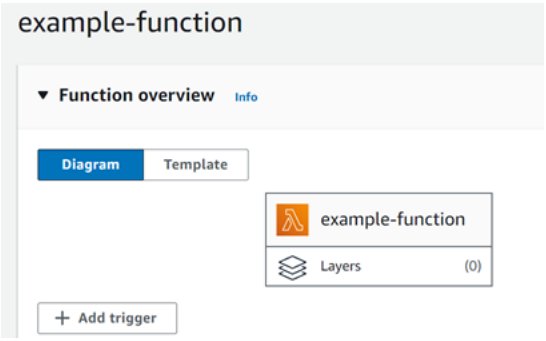
1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.
2. Wählen Sie den Namen einer Funktion aus.
3. Wählen Sie die Registerkarte Konfiguration und dann Berechtigungen aus.
4. Wählen Sie unter Rollenname den Link zu Ihrer Ausführungsrolle. Dieser Link öffnet die Rolle in der IAM-Konsole.



5. Wählen Sie Berechtigungen hinzufügen aus und wählen Sie dann Richtlinien direkt anhängen aus.



6. Geben Sie im Suchfeld `AWSLambdaDynamoDBExecutionRole` ein. Fügen Sie diese Richtlinie zu Ihrer Ausführungsrolle hinzu. Dies ist eine AWS verwaltete Richtlinie, die die Berechtigungen enthält, die Ihre Funktion zum Lesen aus dem DynamoDB-Stream benötigt. Weitere Informationen zu dieser Richtlinie finden Sie unter AWSLambdaDBExecutionDynamo-Rolle in der Referenz für AWS verwaltete Richtlinien.
7. Gehen Sie zurück zu Ihrer Funktion in der Lambda-Konsole. Wählen Sie unter Function overview (Funktionsübersicht) die Option Add trigger (Trigger hinzufügen).



8. Wählen Sie einen Auslösertyp aus.
9. Konfigurieren Sie die erforderlichen Optionen und wählen Sie dann Add (Hinzufügen) aus.

Lambda unterstützt die folgenden Optionen für DynamoDB-Ereignisquellen:

Optionen für die Ereignisquelle

- DynamoDB-Tabelle – Die DynamoDB-Tabelle, aus der Datensätze gelesen werden sollen.
- Batchgröße – Die Anzahl der Datensätze, die in jedem Batch bis zu 10.000 an die Funktion gesendet werden sollen. Lambda übergibt alle Datensätze im Batch in einem einzigen Aufruf an die

Funktion, solange die Gesamtgröße der Ereignisse nicht das Nutzlast-Limit für synchrone Aufrufe überschreitet (6 MB).

- Batchfenster – Geben Sie die maximale Zeitspanne zur Erfassung von Datensätzen vor dem Aufruf der Funktion in Sekunden an.
- Startposition – Verarbeiten Sie nur neue Datensätze oder alle vorhandenen Datensätze.
  - Neueste – Verarbeiten Sie Datensätze, die neu zum Stream hinzugefügt wurden.
  - Horizont trimmen – Verarbeiten Sie alle Datensätze im Stream.

  Nach der Verarbeitung aller vorhandenen Datensätze hat die Funktion aufgeholt und setzt die Verarbeitung neuer Datensätze fort.
- On-failure destination (Ziel bei Ausfall) – Eine SQS-Warteschlange oder ein SNS-Thema für Datensätze, die nicht verarbeitet werden können. Wenn Lambda einen Datensatz-Batch verwirft, weil er zu alt ist oder alle Wiederholungen erschöpft hat, sendet es Details zum Batch an die Warteschlange oder das Thema.
- Wiederholungsversuche – Die maximale Anzahl von Wiederholungen von Lambda, wenn die Funktion einen Fehler zurückgibt. Dies gilt nicht für Servicefehler oder Drosselungen, bei denen der Batch die Funktion nicht erreicht hat.
- Höchstalter des Datensatzes – Das maximale Alter eines Datensatzes, den Lambda an Ihre Funktion sendet.
- Batch bei Fehler aufteilen – Wenn die Funktion einen Fehler zurückgibt, teilen Sie den Batch vor dem erneuten Versuch in zwei Teile. Ihre ursprüngliche Einstellung für die Batch-Größe bleibt unverändert.
- Gleichzeitige Batches pro Shard – Verarbeitet gleichzeitig mehrere Batches aus demselben Shard.
- Aktiviert – Auf „true“ festlegen, um die Ereignisquellenzuordnung zu aktivieren. Auf "false" festlegen, um die Verarbeitung von Datensätzen zu beenden. Lambda merkt sich den zuletzt verarbeiteten Datensatz und setzt die Verarbeitung nach erneuter Aktivierung der Zuordnung an dieser Stelle fort.

> **Note**
>
> Für GetRecords API-Aufrufe, die von Lambda als Teil von DynamoDB-Triggern aufgerufen werden, fallen keine Gebühren an.

Um die Konfiguration der Ereignisquelle zu einem späteren Zeitpunkt zu verwalten, wählen Sie den Auslöser im Designer aus.

Erstellen einer kontoübergreifenden Zuordnung von Ereignisquellen

Amazon DynamoDB unterstützt jetzt ressourcenbasierte Richtlinien. Mit dieser Funktion können Sie Daten aus einem DynamoDB-Stream in einem AWS-Konto mit einer Lambda-Funktion in einem anderen Konto verarbeiten.

Um eine Ereignisquellenzuordnung für Ihre Lambda-Funktion mithilfe eines DynamoDB-Streams in einem anderen zu erstellen AWS-Konto, müssen Sie den Stream mithilfe einer ressourcenbasierten Richtlinie konfigurieren, um Ihrer Lambda-Funktion die Berechtigung zum Lesen von Datensätzen zu erteilen. Informationen dazu, wie Sie Ihren Stream für den kontoübergreifenden Zugriff konfigurieren, finden Sie unter Zugriff mit kontoübergreifenden Lambda-Funktionen teilen im Amazon DynamoDB DynamoDB-Entwicklerhandbuch.

Nachdem Sie Ihren Stream mit einer ressourcenbasierten Richtlinie konfiguriert haben, die Ihrer Lambda-Funktion die erforderlichen Berechtigungen erteilt, erstellen Sie die Ereignisquellenzuordnung mit Ihrem kontoübergreifenden Stream-ARN. Sie finden den Stream-ARN auf der Registerkarte Exporte und Streams der Tabelle in der kontoübergreifenden DynamoDB-Konsole.

Wenn Sie die Lambda-Konsole verwenden, fügen Sie den Stream-ARN direkt in das Eingabefeld der DynamoDB-Tabelle auf der Seite zur Erstellung der Ereignisquellenzuordnung ein.

Hinweis: Regionsübergreifende Trigger werden nicht unterstützt.

Konfigurieren einer teilweisen Batch-Antwort mit DynamoDB und Lambda

Beim Konsumieren und Verarbeiten von Streaming-Daten aus einer Ereignisquelle werden standardmäßig Lambda-Checkpoints auf die höchste Sequenznummer eines Batches nur dann überprüft, wenn der Batch ein voller Erfolg ist. Lambda behandelt alle anderen Ergebnisse als einen vollständigen Fehler und versucht, den Batch bis zum Wiederholungslimit zu verarbeiten. Um beim Verarbeiten von Stapeln aus einem Stream Teilerfolge zu ermöglichen, aktivieren Sie `ReportBatchItemFailures`. Das Zulassen von Teilerfolgen kann dazu beitragen, die Anzahl der Wiederholungen in einer Aufzeichnung zu reduzieren, obwohl die Möglichkeit von Wiederholungen in einer erfolgreichen Aufzeichnung nicht vollständig verhindert wird.

Zum Aktivieren von `ReportBatchItemFailures` fügen Sie den Enum-Wert **`ReportBatchItemFailures`** der FunctionResponseTypes-Liste hinzu. Diese Liste zeigt an, welche

Antworttypen für Ihre Funktion aktiviert sind. Sie können diese Liste konfigurieren, wenn Sie eine Zuordnung von Ereignisquellen erstellen oder aktualisieren.

> **Note**
>
> Selbst wenn Ihr Funktionscode teilweise Antworten bei Batch-Fehlern zurückgibt, werden diese Antworten nicht von Lambda verarbeitet, es sei denn, das `ReportBatchItemFailures`-Feature ist explizit für Ihre Zuordnung von Ereignisquellen aktiviert.

Berichtsyntax

Beim Konfigurieren von Berichten zu Batch-Elementfehlern wird die `StreamsEventResponse`-Klasse mit einer Liste von Batch-Elementfehlern zurückgegeben. Sie können ein `StreamsEventResponse`-Objekt verwenden, um die Sequenznummer des ersten fehlgeschlagenen Datensatzes im Batch zurückzugeben. Sie können auch Ihre eigene benutzerdefinierte Klasse mit der richtigen Antwortsyntax erstellen. Die folgende JSON-Struktur zeigt die erforderliche Antwortsyntax:

```
{
  "batchItemFailures": [
        {
            "itemIdentifier": "<SequenceNumber>"
        }
    ]
}
```

> **Note**
>
> Wenn das `batchItemFailures`-Array mehrere Elemente enthält, verwendet Lambda den Datensatz mit der niedrigsten Sequenznummer als Kontrollpunkt. Lambda wiederholt dann alle Datensätze ab diesem Kontrollpunkt.

Erfolgs- und Misserfolgsbedingungen

Lambda behandelt einen Batch als vollständigen Erfolg, wenn Sie eines der folgenden Elemente zurückgeben:

- Eine leere `batchItemFailure`-Liste
- Eine ungültige `batchItemFailure`-Liste
- Ein leeres `EventResponse`
- Ein ungültiges `EventResponse`

Lambda behandelt einen Batch als vollständigen Misserfolg, wenn Sie eines der folgenden Elemente zurückgeben:

- Eine leere Zeichenfolge `itemIdentifier`
- Ein ungültiges `itemIdentifier`
- Ein `itemIdentifier` mit einem falschen Schlüsselnamen

Lambda wiederholt Fehler basierend auf Ihrer Wiederholungsstrategie.

Einen Batch halbieren

Wenn Ihr Aufruf fehlschlägt und `BisectBatchOnFunctionError` eingeschaltet ist, wird der Stapel unabhängig von Ihrer `ReportBatchItemFailures`-Einstellung halbiert.

Wenn eine partielle Batch-Erfolgsantwort empfangen wird und sowohl `BisectBatchOnFunctionError` als auch `ReportBatchItemFailures` aktiviert sind, wird der Batch mit der zurückgegebenen Sequenznummer halbiert und Lambda versucht nur die verbleibenden Datensätze erneut.

Um die Implementierung der partiellen Batch-Antwortlogik zu vereinfachen, sollten Sie in Erwägung ziehen, das Batch Processor Utility von Powertools for zu verwenden AWS Lambda, das diese Komplexität automatisch für Sie erledigt.

Hier sind einige Beispiele für Funktionscode, der die Liste der fehlgeschlagenen Nachrichten IDs im Batch zurückgibt:

.NET

SDK für .NET

> **Note**
>
> Es gibt noch mehr dazu GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Melden von DynamoDB-Batchelementfehlern mit Lambda unter Verwendung von .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

SDK für Go V2

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Melden von DynamoDB-Batchelementfehlern mit Lambda unter Verwendung von 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 für Java 2.x

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Melden von DynamoDB-Batchelementfehlern mit Lambda unter Verwendung von 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 für JavaScript (v3)

> **Note**
>
> Es gibt noch mehr dazu GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Melden von DynamoDB-Batchelementfehlern mit Lambda unter Verwendung von. 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: [] };
};
```

Melden von DynamoDB-Batchelementfehlern mit Lambda unter Verwendung von. 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

SDK für PHP

> **Note**
>
> Es gibt noch mehr dazu. GitHub Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Melden von DynamoDB-Batchelementfehlern mit Lambda unter Verwendung von 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

SDK für Python (Boto3)

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Melden von DynamoDB-Batchelementfehlern mit Lambda unter Verwendung von 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

SDK für Ruby

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Melden von DynamoDB-Batchelementfehlern mit Lambda unter Verwendung von 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 für Rust

> **Note**
>
> Es gibt noch mehr dazu GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Melden von DynamoDB-Batchelementfehlern mit Lambda unter Verwendung von 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
}
```

Powertools für die AWS Lambda Batch-Verarbeitung verwenden

Das Batchverarbeitungsprogramm von Powertools for verarbeitet AWS Lambda automatisch die Logik für partielle Batch-Antworten und reduziert so die Komplexität der Implementierung von Batch-Fehlerberichten. Hier sind Beispiele für die Verwendung des Batch-Prozessors:

Python

> **Note**
>
> Umfassende Beispiele und Anweisungen zur Einrichtung finden Sie in der Dokumentation zum Batch-Prozessor.

Verarbeitung von DynamoDB-Stream-Datensätzen mit AWS Lambda Batch-Prozessor.

```
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**
>
> Umfassende Beispiele und Anweisungen zur Einrichtung finden Sie in der Dokumentation zum Batch-Prozessor.

Verarbeitung von DynamoDB-Stream-Datensätzen mit AWS Lambda Batch-Prozessor.

```
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**
>
> Umfassende Beispiele und Anweisungen zur Einrichtung finden Sie in der Dokumentation zum Batch-Prozessor.

Verarbeitung von DynamoDB-Stream-Datensätzen mit AWS Lambda Batch-Prozessor.

```
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**
>
> Umfassende Beispiele und Anweisungen zur Einrichtung finden Sie in der Dokumentation zum Batch-Prozessor.

Verarbeitung von DynamoDB-Stream-Datensätzen mit AWS Lambda Batch-Prozessor.

```
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;
    }
}
```

Verworfene Datensätze für eine DynamoDB-Ereignisquelle in Lambda aufbewahren

Die Fehlerbehandlung für DynamoDB-Zuordnungen von Ereignisquellen hängt davon ab, ob der Fehler vor dem Aufruf der Funktion oder während des Funktionsaufrufs auftritt:

- Vor dem Aufruf: Wenn eine Lambda-Ereignisquellenzuordnung die Funktion aufgrund von Drosselung oder anderen Problemen nicht aufrufen kann, versucht sie es erneut, bis die Datensätze ablaufen oder das in der Ereignisquellenzuordnung () konfigurierte Höchstalter überschreiten. MaximumRecordAgeInSeconds
- Während des Aufrufs: Wenn die Funktion aufgerufen wird, aber einen Fehler zurückgibt, versucht Lambda es erneut, bis die Datensätze ablaufen, das Höchstalter (MaximumRecordAgeInSeconds) überschreiten oder die konfigurierte Wiederholungsquote () erreicht haben. MaximumRetryAttempts Bei Funktionsfehlern können Sie auch konfigurieren BisectBatchOnFunctionError, dass ein fehlgeschlagener Batch in zwei kleinere Batches aufgeteilt wird, wodurch fehlerhafte Datensätze isoliert und Timeouts vermieden werden. Durch das Aufteilen von Batches wird die Quote für Wiederholungen nicht verbraucht.

Wenn die Fehlerbehandlungsmaßnahmen fehlschlagen, verwirft Lambda die Datensätze und setzt die Verarbeitung von Batches aus dem Stream fort. Bei den Standardeinstellungen bedeutet dies, dass ein fehlerhafter Datensatz die Verarbeitung auf dem betroffenen Shard für bis zu einen Tag blockieren kann. Um dies zu vermeiden, konfigurieren Sie die Ereignisquellenzuordnung Ihrer Funktion mit einer angemessenen Anzahl von Wiederholungen und einem maximalen Datensatzalter, das zu Ihrem Anwendungsfall passt.

Konfigurieren von Zielen für fehlgeschlagene Aufrufe

Um Datensätze zu fehlgeschlagenen Aufrufen zur Zuordnung von Ereignisquellen beizubehalten, fügen Sie der Zuordnung von Ereignisquellen Ihrer Funktion ein Ziel hinzu. Jeder an das Ziel gesendete Datensatz ist ein JSON-Dokument mit Metadaten über den fehlgeschlagenen Aufruf. Bei Amazon S3-Zielen sendet Lambda auch den gesamten Aufrufdatensatz zusammen mit den Metadaten. Sie können jedes Amazon SNS SNS-Thema, jede Amazon SQS SQS-Warteschlange, jeden Amazon S3 S3-Bucket oder Kafka als Ziel konfigurieren.

Bei Amazon-S3-Zielen können Sie das Feature Amazon S3 Event Notifications verwenden, um Benachrichtigungen zu erhalten, wenn Objekte in Ihren S3-Ziel-Bucket hochgeladen werden. Sie

können S3-Ereignisbenachrichtigungen auch so konfigurieren, dass sie eine andere Lambda-Funktion aufrufen, um eine automatische Verarbeitung für fehlgeschlagene Stapel durchzuführen.

Ihre Ausführungsrolle muss über Berechtigungen für das Ziel verfügen:

- Für ein SQS-Ziel: sqs: SendMessage
- Für ein SNS-Ziel: sns:Publish
- Für ein S3-Ziel: s3: und s3: PutObject ListBucket
- Für ein Kafka-Ziel: kafka-cluster: WriteData

Sie können ein Kafka-Thema als Ziel für den Fall eines Fehlers für Ihre Quellenzuordnungen für Kafka-Ereignisse konfigurieren. Wenn Lambda Datensätze nach anstrengenden Wiederholungsversuchen nicht verarbeiten kann oder wenn Datensätze das Höchstalter überschreiten, sendet Lambda die fehlgeschlagenen Datensätze zur späteren Verarbeitung an das angegebene Kafka-Thema. Weitere Informationen finden Sie unter the section called “Kafka-Ziel bei einem Ausfall”.

Wenn Sie die Verschlüsselung mit Ihrem eigenen KMS-Schlüssel für ein S3-Ziel aktiviert haben, muss die Ausführungsrolle Ihrer Funktion auch die Berechtigung haben, kms: aufzurufen. GenerateDataKey Wenn sich der KMS-Schlüssel und das S3-Bucket-Ziel in einem anderen Konto als Ihre Lambda-Funktion und Ausführungsrolle befinden, konfigurieren Sie den KMS-Schlüssel so, dass er der Ausführungsrolle vertraut, die zugelassen kms: GenerateDataKey wird.

Gehen Sie folgendermaßen vor, um ein Ausfallziel mit der Konsole zu konfigurieren:

1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.
2. Wählen Sie eine Funktion aus.
3. Wählen Sie unter Function overview (Funktionsübersicht) die Option Add destination (Ziel hinzufügen).
4. Wählen Sie als Quelle die Option Aufruf der Zuordnung von Ereignisquellen aus.
5. Wählen Sie für die Zuordnung von Ereignisquellen eine Ereignisquelle aus, die für diese Funktion konfiguriert ist.
6. Wählen Sie für Bedingung die Option Bei Ausfall aus. Für Aufrufe zur Zuordnung von Ereignisquellen ist dies die einzig akzeptierte Bedingung.
7. Wählen Sie unter Zieltyp den Zieltyp aus, an den Lambda Aufrufdatensätze sendet.
8. Wählen Sie unter Destination (Ziel) eine Ressource aus.

9. Wählen Sie Speichern.

Sie können mit AWS Command Line Interface (AWS CLI) auch ein Ziel für den Fall eines Fehlers konfigurieren. Mit dem folgenden create-event-source-mappingBefehl wird beispielsweise eine Zuordnung der Ereignisquelle mit einem SQS-Ziel für den Fall eines Fehlers hinzugefügt: `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"}}'
```

Der folgende update-event-source-mappingBefehl aktualisiert eine Ereignisquellenzuordnung, sodass fehlgeschlagene Aufrufdatensätze nach zwei Wiederholungsversuchen oder wenn die Datensätze älter als eine Stunde sind, an ein SNS-Ziel gesendet werden.

```
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"}}'
```

Aktualisierte Einstellungen werden asynchron angewendet und werden erst nach Abschluss des Vorgangs in der Ausgabe berücksichtigt. Verwenden Sie den get-event-source-mapping-Befehl, um den aktuellen Status anzuzeigen.

Um ein Ziel zu entfernen, geben Sie eine leere Zeichenfolge als Argument für den `destination-config`-Parameter an:

```
aws lambda update-event-source-mapping \
--uuid f89f8514-cdd9-4602-9e1f-01a5b77d449b \
--destination-config '{"OnFailure": {"Destination": ""}}'
```

Bewährte Methoden für die Sicherheit in Amazon S3-Zielen

Das Löschen eines S3-Buckets, der als Ziel konfiguriert ist, ohne das Ziel aus der Konfiguration Ihrer Funktion zu entfernen, kann ein Sicherheitsrisiko darstellen. Wenn ein anderer Benutzer

den Namen Ihres Ziel-Buckets kennt, kann er den Bucket in seinem AWS-Konto neu erstellen. Aufzeichnungen über fehlgeschlagene Aufrufe werden an den entsprechenden Bucket gesendet, wodurch möglicherweise Daten aus Ihrer Funktion verfügbar gemacht werden.

> **Warning**
>
> Um sicherzustellen, dass Aufrufdatensätze Ihrer Funktion nicht an einen S3-Bucket in einem anderen gesendet werden können AWS-Konto, fügen Sie der Ausführungsrolle Ihrer Funktion eine Bedingung hinzu, die die `s3:PutObject` Berechtigungen auf Buckets in Ihrem Konto beschränkt.

-Das folgende Beispiel zeigt eine IAM-Richtlinie, die die `s3:PutObject`-Berechtigungen Ihrer Funktion auf Buckets in Ihrem Konto beschränkt. Diese Richtlinie gibt Lambda auch die `s3:ListBucket`-Erlaubnis, einen S3-Bucket als Ziel zu verwenden.

```
{
    "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"
                }
            }
        }
    ]
}
```

Um der Ausführungsrolle Ihrer Funktion mithilfe von AWS-Managementkonsole oder eine Berechtigungsrichtlinie hinzuzufügen AWS CLI, lesen Sie die Anweisungen in den folgenden Verfahren:

Console

So fügen Sie der Ausführungsrolle einer Funktion (Konsole) eine Berechtigungsrichtlinie hinzu

1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.
2. Wählen Sie die Lambda-Funktion aus, deren Ausführungsrolle Sie ändern möchten.
3. Klicken Sie in der Registerkarte Konfiguration auf die Option Berechtigungen.
4. Wählen Sie auf der Registerkarte Ausführungsrolle den Rollennamen Ihrer Funktion aus, um die IAM-Konsolenseite der Rolle zu öffnen.
5. Fügen Sie der Rolle wie folgt eine Richtlinie mit Berechtigungen hinzu:

   a. Wählen Sie im Bereich Berechtigungsrichtlinien die Optionen Berechtigungen hinzufügen und dann Inline-Richtlinie erstellen aus.
   b. Wählen Sie im Richtlinien-Editor JSON aus.
   c. Fügen Sie die Richtlinie, die Sie hinzufügen möchten, in den Editor ein (indem Sie die vorhandene JSON-Datei ersetzt) und wählen Sie dann Weiter aus.
   d. Geben Sie unter Richtliniendetails für den Richtliniennamen ein.
   e. Wählen Sie Richtlinie erstellen aus.

AWS CLI

So fügen Sie der Ausführungsrolle einer Funktion (CLI) eine Berechtigungsrichtlinie hinzu

1. Erstellen Sie ein JSON-Richtliniendokument mit den erforderlichen Berechtigungen und speichern Sie es in einem lokalen Verzeichnis.
2. Verwenden Sie den `put-role-policy` IAM-CLI-Befehl, um die Berechtigungen zur Ausführungsrolle Ihrer Funktion hinzuzufügen. Führen Sie den folgenden Befehl in dem Verzeichnis aus, in dem Sie Ihr JSON-Richtliniendokument gespeichert haben und ersetzen Sie den Rollennamen, den Richtliniennamen und das Richtliniendokument durch Ihre eigenen Werte.

```
aws iam put-role-policy \
--role-name my_lambda_role \
```

```
--policy-name LambdaS3DestinationPolicy \
--policy-document file://my_policy.json
```

Beispiel für einen Amazon SNS- und Amazon SQS-Aufrufsatz

Das folgende Beispiel zeigt einen Aufrufsatz, den Lambda an ein SQS- oder SNS-Ziel für einen DynamoDB-Stream sendet.

```
{
    "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"
    }
}
```

Sie können diese Informationen verwenden, um die betroffenen Datensätze aus dem Stream für die Fehlersuche abzurufen. Die tatsächlichen Datensätze sind nicht enthalten, daher müssen Sie diesen Datensatz verarbeiten und aus dem Stream abrufen, bevor sie ablaufen und verloren gehen.

Beispiel eines Amazon S3-Aufrufdatensatzes

Das folgende Beispiel zeigt einen Aufrufsatz, den Lambda für einen DynamoDB-Stream an einen S3-Bucket sendet. Zusätzlich zu allen Feldern aus dem vorherigen Beispiel für SQS- und SNS-Ziele enthält das Feld `payload` den ursprünglichen Aufrufdatensatz als maskierte JSON-Zeichenfolge.

```
{
    "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-000000015736898471 84-864758bb",
        "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"
    },
    "payload": "<Whole Event>" // Only available in S3
}
```

Das S3-Objekt, das den Aufrufdatensatz enthält, verwendet die folgende Namenskonvention:

```
aws/lambda/<ESM-UUID>/<shardID>/YYYY/MM/DD/YYYY-MM-DDTHH.MM.SS-<Random UUID>
```

Implementierung der statusbehafteten DynamoDB-Stream-Verarbeitung in Lambda

Lambda-Funktionen können kontinuierliche Stream-Verarbeitungsanwendungen ausführen. Ein Stream entspricht einer unbegrenzten Menge von Daten, die kontinuierlich durch Ihre Anwendung fließen. Um Informationen aus dieser sich ständig aktualisierenden Eingabe zu analysieren, können Sie die enthaltenen Datensätze mithilfe eines zeitlich definierten Fensters binden.

Rollierende Fenster sind unterschiedliche Zeitfenster, die sich in regelmäßigen Abständen öffnen und schließen. Standardmäßig sind Lambda-Aufrufe zustandslos – Sie können sie nicht für die Verarbeitung von Daten über mehrere kontinuierliche Aufrufe hinweg ohne eine externe Datenbank verwenden. Mit rollierenden Fenstern können Sie jedoch Ihren Status über Aufrufe hinweg beibehalten. Dieser Zustand enthält das Gesamtergebnis der Nachrichten, die zuvor für das aktuelle Fenster verarbeitet wurden. Ihr Zustand kann maximal 1 MB pro Shard betragen. Wenn er diese Größe überschreitet, wird Lambda das Fenster vorzeitig beenden.

Jeder Datensatz in einem Stream gehört zu einem bestimmten Fenster. Lambda verarbeitet jeden Datensatz mindestens einmal, garantiert jedoch nicht, dass jeder Datensatz nur einmal verarbeitet wird. In seltenen Fällen, etwa bei der Fehlerbehandlung, werden einige Datensätze möglicherweise mehrmals verarbeitet. Datensätze werden beim ersten Mal immer in der richtigen Reihenfolge verarbeitet. Wenn Datensätze mehr als einmal verarbeitet werden, werden sie nicht in der richtigen Reihenfolge verarbeitet.

Aggregation und Verarbeitung

Ihre benutzerverwaltete Funktion wird sowohl zur Aggregation als auch zur Verarbeitung der Endergebnisse dieser Aggregation aufgerufen. Lambda aggregiert alle im Fenster empfangenen Datensätze. Sie können diese Datensätze in mehreren Stapeln erhalten, jeweils als ein separater Aufruf. Jeder Aufruf erhält einen Zustand. Wenn Sie also rollierende Fenster verwenden, muss Ihre Lambda-Funktionsantwort eine `state`-Eigenschaft enthalten. Wenn die Antwort keine `state`-Eigenschaft enthält, betrachtet Lambda dies als fehlgeschlagenen Aufruf. Um diese Bedingung zu erfüllen, kann Ihre Funktion ein `TimeWindowEventResponse`-Objekt zurückgeben, das die folgende JSON-Form aufweist:

Example **`TimeWindowEventResponse`**-Werte

```
{
    "state": {
        "1": 282,
```

```
        "2": 715
    },
    "batchItemFailures": []
}
```

> **Note**
>
> Für Java-Funktionen empfehlen wir, eine `Map<String, String>` zu verwenden, um den Status darzustellen.

Am Ende des Fensters wird das Flag `isFinalInvokeForWindow` auf `true` gesetzt, um anzugeben, dass es sich um den Endzustand handelt und dass es für die Verarbeitung bereit ist. Nach der Verarbeitung werden das Fenster und Ihr endgültiger Aufruf wird abgeschlossen, und dann wird der Zustand gelöscht.

Am Ende Ihres Fensters verwendet Lambda die endgültige Verarbeitung für Aktionen an den Aggregationsergebnissen. Ihre endgültige Verarbeitung wird synchron aufgerufen. Nach erfolgreichem Aufruf zeigt Ihre Funktion auf die Sequenznummer und die Stream-Verarbeitung wird fortgesetzt. Wenn der Aufruf nicht erfolgreich ist, unterbricht Ihre Lambda-Funktion die weitere Verarbeitung bis zu einem erfolgreichen Aufruf.

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
}
```

Konfiguration

Sie können rollierende Fenster konfigurieren, wenn Sie eine Ereignisquellenzuordnung erstellen oder aktualisieren. Um ein rollierendes Fenster zu konfigurieren, geben Sie das Fenster in Sekunden (TumblingWindowInSeconds) an. Der folgende AWS Command Line Interface (AWS CLI) - Beispielbefehl erstellt eine Streaming-Ereignisquellenzuordnung, die ein rollierendes Fenster von 120 Sekunden hat. Die für Aggregation und Verarbeitung definierte Lambda-Funktion wird genannt `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 bestimmt die rollierenden Fenstergrenzen basierend auf dem Zeitpunkt, zu dem Datensätze in den Stream eingefügt wurden. Für alle Datensätze steht ein ungefährer Zeitstempel zur Verfügung, den Lambda in Grenzbestimmungen verwendet.

Rollierende Fensteraggregationen unterstützen kein Resharding. Wenn der Shard endet, betrachtet Lambda das Fenster als geschlossen und die untergeordneten Shards beginnen ihr eigenes Fenster in einem neuen Zustand.

Rollierende Fenster unterstützen vollständig die bestehenden Wiederholungsrichtlinien `maxRetryAttempts` und `maxRecordAge`.

Example Handler.py – Aggregation und Verarbeitung

Die folgende Python-Funktion veranschaulicht, wie Sie Ihren Endzustand aggregieren und dann verarbeiten:

```
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}
```

Lambdaparameter für Amazon DynamoDB-SQS-Zuordnungen von Ereignisquellen

Alle Lambda-Ereignisquellentypen verwenden dieselben API-Operationen CreateEventSourceMapping und UpdateEventSourceMapping. Allerdings gelten nur einige der Parameter für DynamoDB Streams.

Parameter	Erforderlich	Standard	Hinweise
BatchSize	N	100	Höchstwert: 10 000.
BisectBatchOnFunctionError	N	false	Keine
DestinationConfig	N	N/A	Ein Ziel der Amazon-SQS-Standardwarteschlange oder des Amazon-SNS-Standardthemas für verworfene Datensätze
Aktiviert	N	true	Keine

Parameter	Erforderlich	Standard	Hinweise
EventSourceArn	Y	N/A	Der ARN des Datenstroms oder eines Stream-Konsumenten
FilterCriteria	N	N/A	Steuern Sie, welche Ereignisse Lambda an Ihre Funktion sendet
FunctionName	Y	N/A	Keine
FunctionResponseTypes	N	N/A	Damit Ihre Funktion bestimmte Fehler in einem Batch meldet, beziehen Sie den Wert `ReportBatchItemFailures` in `FunctionResponseTypes` ein. Weitere Informationen finden Sie unter Konfigurieren einer teilweisen Batch-Antwort mit DynamoDB und Lambda.
MaximumBatchingWindowInSeconds	N	0	Keine

Parameter	Erforderlich	Standard	Hinweise
MaximumRecordAgeInSeconds	N	-1	-1 bedeutet unendlich: Fehlgeschlagene Datensätze werden wiederholt, bis der Datensatz abläuft. Das Datenaufbewahrungslimit für DynamoDB Streams beträgt 24 Stunden. Minimum: -1 Höchstwert: 604 800
MaximumRetryAttempts	N	-1	-1 bedeutet unendlich: Fehlgeschlagene Datensätze werden wiederholt, bis der Datensatz abläuft Minimum: 0 Höchstwert: 10 000.
ParallelizationFactor	N	1	Maximum: 10
StartingPosition	Y	N/A	TRIM_HORIZON oder LATEST
TumblingWindowInSeconds	N	N/A	Minimum: 0 Maximum: 900

Verwendung der Ereignisfilterung mit einer DynamoDB-Ereignisquelle

Sie können die Ereignisfilterung verwenden, um zu steuern, welche Datensätze aus einem Stream oder einer Warteschlange Lambda an Ihre Funktion sendet. Allgemeine Informationen über die Funktionsweise der Ereignisfilterung finden Sie unter the section called “Ereignisfilterung”.

Dieser Abschnitt konzentriert sich auf die Ereignisfilterung für DynamoDB-Ereignisquellen.

> **Note**
>
> DynamoDB-Zuordnungen von Ereignisquellen unterstützen nur das Filtern nach dem `dynamodb`-Schlüssel.

Themen

- DynamoDB-Ereignis
- Filterung mit Tabellenattributen
- Filterung mit booleschen Ausdrücken
- Verwenden des Exists-Operators
- JSON-Format für DynamoDB-Filterung

DynamoDB-Ereignis

Angenommen, Sie haben eine DynamoDB-Tabelle mit dem Primärschlüssel `CustomerName` und den Attributen `AccountManager` und `PaymentTerms`. Im Folgenden sehen Sie einen Beispieldatensatz aus dem Stream Ihrer DynamoDB-Tabelle.

```
{
      "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"
        }
    }
```

Um anhand der Schlüssel- und Attributwerte in Ihrer DynamoDB-Tabelle zu filtern, verwenden Sie den dynamodb-Schlüssel im Datensatz. Die folgenden Abschnitte enthalten Beispiele für verschiedene Filtertypen.

Filtern mit Tabellenschlüsseln

Angenommen, Sie möchten, dass Ihre Funktion nur die Datensätze verarbeitet, deren Primärschlüssel `CustomerName` "AnyCompany Industries" lautet. Das `FilterCriteria`-Objekt würde wie folgt aussehen.

```
{
    "Filters": [
        {
            "Pattern": "{ \"dynamodb\" : { \"Keys\" : { \"CustomerName\" : { \"S\" :
 [ \"AnyCompany Industries\" ] } } } }"
        }
    ]
}
```

Zur Verdeutlichung sehen Sie hier den Wert des Filter-`Pattern` in reinem JSON.

```
{
    "dynamodb": {
        "Keys": {
            "CustomerName": {
```

```
                "S": [ "AnyCompany Industries" ]
                }
            }
        }
}
```

Sie können Ihren Filter mithilfe der Konsole, AWS CLI oder einer AWS SAM-Vorlage hinzufügen.

Console

Um diesen Filter mithilfe der Konsole hinzuzufügen, folgen Sie den Anweisungen unter Anhängen von Filterkriterien an eine Ereignisquellenzuordnung (Konsole) und geben Sie die folgende Zeichenfolge für die Filterkriterien ein.

```
{ "dynamodb" : { "Keys" : { "CustomerName" : { "S" : [ "AnyCompany
 Industries" ] } } } }
```

AWS CLI

Um eine neue Zuordnung von Ereignisquellen mit diesen Filterkriterien über die AWS Command Line Interface (AWS CLI) zu erstellen, führen Sie den folgenden Befehl aus.

```
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\" ] } } } }"}]}'
```

Führen Sie den folgenden Befehl aus, um diese Filterkriterien zu einer vorhandenen Zuordnung von Ereignisquellen hinzuzufügen.

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"dynamodb\" : { \"Keys\" :
 { \"CustomerName\" : { \"S\" : [ \"AnyCompany Industries\" ] } } } }"}]}'
```

AWS SAM

Um diesen Filter mit der AWS SAM hinzuzufügen, fügen Sie das folgende Snippet zur YAML-Vorlage für Ihre Ereignisquelle hinzu.

```
FilterCriteria:
   Filters:
     - Pattern: '{ "dynamodb" : { "Keys" : { "CustomerName" : { "S" : [ "AnyCompany
 Industries" ] } } } }'
```

Filterung mit Tabellenattributen

Mit DynamoDB können Sie auch die Schlüssel `NewImage` und `OldImage` verwenden, um nach Attributwerten zu filtern. Angenommen, Sie möchten Datensätze filtern, bei denen das `AccountManager`-Attribut im letzten Tabellenbild "Pat Candella" oder "Shirley Rodriguez" lautet. Das `FilterCriteria`-Objekt würde wie folgt aussehen.

```
{
    "Filters": [
        {
            "Pattern": "{ \"dynamodb\" : { \"NewImage\" : { \"AccountManager\" : { \"S
\" : [ \"Pat Candella\", \"Shirley Rodriguez\" ] } } } }"
        }
    ]
}
```

Zur Verdeutlichung sehen Sie hier den Wert des Filter-`Pattern` in reinem JSON.

```
{
    "dynamodb": {
        "NewImage": {
            "AccountManager": {
                "S": [ "Pat Candella", "Shirley Rodriguez" ]
            }
        }
    }
}
```

Sie können Ihren Filter mithilfe der Konsole, AWS CLI oder einer AWS SAM-Vorlage hinzufügen.

Console

Um diesen Filter mithilfe der Konsole hinzuzufügen, folgen Sie den Anweisungen unter Anhängen von Filterkriterien an eine Ereignisquellenzuordnung (Konsole) und geben Sie die folgende Zeichenfolge für die Filterkriterien ein.

```
{ "dynamodb" : { "NewImage" : { "AccountManager" : { "S" : [ "Pat Candella",
 "Shirley Rodriguez" ] } } } }
```

AWS CLI

Um eine neue Zuordnung von Ereignisquellen mit diesen Filterkriterien über die AWS Command Line Interface (AWS CLI) zu erstellen, führen Sie den folgenden Befehl aus.

```
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
\" ] } } } }"}]}'
```

Führen Sie den folgenden Befehl aus, um diese Filterkriterien zu einer vorhandenen Zuordnung von Ereignisquellen hinzuzufügen.

```
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

Um diesen Filter mit der AWS SAM hinzuzufügen, fügen Sie das folgende Snippet zur YAML-Vorlage für Ihre Ereignisquelle hinzu.

```
FilterCriteria:
  Filters:
    - Pattern: '{ "dynamodb" : { "NewImage" : { "AccountManager" : { "S" : [ "Pat
 Candella", "Shirley Rodriguez" ] } } } }'
```

Filterung mit booleschen Ausdrücken

Sie können Filter auch mithilfe von booleschen UND-Ausdrücken erstellen. Diese Ausdrücke können sowohl die Schlüssel- als auch die Attributparameter Ihrer Tabelle enthalten. Angenommen, Sie möchten Datensätze filtern, bei denen der `NewImage`-Wert von `AccountManager` „Pat Candella“

und der `OldImage`-Wert „Terry Whitlock“ ist. Das `FilterCriteria`-Objekt würde wie folgt aussehen.

```
{
    "Filters": [
        {
            "Pattern": "{ \"dynamodb\" : { \"NewImage\" : { \"AccountManager\" : { \"S
\" : [ \"Pat Candella\" ] } } } , \"dynamodb\" : { \"OldImage\" : { \"AccountManager
\" : { \"S\" : [ \"Terry Whitlock\" ] } } } }"
        }
    ]
}
```

Zur Verdeutlichung sehen Sie hier den Wert des Filter-`Pattern` in reinem JSON.

```
{
    "dynamodb" : {
        "NewImage" : {
            "AccountManager" : {
                "S" : [
                    "Pat Candella"
                ]
            }
        }
    },
    "dynamodb": {
        "OldImage": {
            "AccountManager": {
                "S": [
                    "Terry Whitlock"
                ]
            }
        }
    }
}
```

Sie können Ihren Filter mithilfe der Konsole, AWS CLI oder einer AWS SAM-Vorlage hinzufügen.

Console

Um diesen Filter mithilfe der Konsole hinzuzufügen, folgen Sie den Anweisungen unter Anhängen von Filterkriterien an eine Ereignisquellenzuordnung (Konsole) und geben Sie die folgende Zeichenfolge für die Filterkriterien ein.

```
{ "dynamodb" : { "NewImage" : { "AccountManager" : { "S" : [ "Pat
 Candella" ] } } } , "dynamodb" : { "OldImage" : { "AccountManager" : { "S" :
 [ "Terry Whitlock" ] } } } }
```

AWS CLI

Um eine neue Zuordnung von Ereignisquellen mit diesen Filterkriterien über die AWS Command Line Interface (AWS CLI) zu erstellen, führen Sie den folgenden Befehl aus.

```
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\" ] } } } }
 "}]}'
```

Führen Sie den folgenden Befehl aus, um diese Filterkriterien zu einer vorhandenen Zuordnung von Ereignisquellen hinzuzufügen.

```
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

Um diesen Filter mit der AWS SAM hinzuzufügen, fügen Sie das folgende Snippet zur YAML-Vorlage für Ihre Ereignisquelle hinzu.

```
FilterCriteria:
  Filters:
```

```
    - Pattern: '{ "dynamodb" : { "NewImage" : { "AccountManager" : { "S" : [ "Pat
Candella" ] } } } , "dynamodb" : { "OldImage" : { "AccountManager" : { "S" :
[ "Terry Whitlock" ] } } } }'
```

> **Note**
>
> Die DynamoDB-Ereignisfilterung unterstützt nicht die Verwendung von numerischen Operatoren (numerisch gleich und numerisch Bereich). Auch wenn Elemente in Ihrer Tabelle als Zahlen gespeichert sind, werden diese Parameter im JSON-Datensatzobjekt in Strings umgewandelt.

Verwenden des Exists-Operators

Aufgrund der Struktur von JSON-Ereignisobjekten aus DynamoDB ist bei der Verwendung des Operators „Exists“ besondere Vorsicht geboten. Der Exists-Operator funktioniert nur bei Blattknoten im Ereignis-JSON. Wenn Ihr Filtermuster also Exists verwendet, um nach einem Zwischenknoten zu suchen, wird es nicht funktionieren. Betrachten Sie das folgende DynamoDB-Tabellenelement:

```
{
  "UserID": {"S": "12345"},
  "Name": {"S": "John Doe"},
  "Organizations": {"L": [
      {"S":"Sales"},
      {"S":"Marketing"},
      {"S":"Support"}
    ]
  }
}
```

Möglicherweise möchten Sie ein Filtermuster wie das folgende erstellen, das nach Ereignissen sucht, die `"Organizations"` enthalten:

```
{ "dynamodb" : { "NewImage" : { "Organizations" : [ { "exists": true } ] } } }
```

Dieses Filtermuster würde jedoch nie eine Übereinstimmung ergeben, da `"Organizations"` kein Blattknoten ist. Das folgende Beispiel zeigt, wie der Operator „Exists“ richtig verwendet wird, um das gewünschte Filtermuster zu erstellen:

```
{ "dynamodb" : { "NewImage" : {"Organizations": {"L": {"S": [ {"exists":
 true } ] } } } } }
```

JSON-Format für DynamoDB-Filterung

Um Ereignisse aus DynamoDB-Quellen richtig zu filtern, müssen sowohl das Datenfeld als auch Ihre Filterkriterien für das Datenfeld (`dynamodb`) im gültigen JSON-Format vorliegen. Wenn eines der Felder kein gültiges JSON-Format hat, verwirft Lambda die Nachricht oder gibt eine Ausnahme aus. In der folgenden Tabelle ist das Verhalten zusammengefasst:

Format der eingehenden Daten	Filtermusterformat für Dateneigenschaften	Resultierende Aktion
Gültiges JSON	Gültiges JSON	Lambda filtert basierend auf Ihren Filterkriterien.
Gültiges JSON	Kein Filtermuster für Dateneigenschaften	Lambda filtert (nur für die anderen Metadateneigenschaften) basierend auf Ihren Filterkriterien.
Gültiges JSON	Kein JSON	Lambda gibt zum Zeitpunkt der Erstellung oder Aktualisierung der Ereignisquellenzuordnung eine Ausnahme aus. Das Filtermuster für Dateneigenschaften muss ein gültiges JSON-Format haben.
Kein JSON	Gültiges JSON	Lambda verwirft den Datensatz.
Kein JSON	Kein Filtermuster für Dateneigenschaften	Lambda filtert (nur für die anderen Metadateneigenschaften) basierend auf Ihren Filterkriterien.

Format der eingehenden Daten	Filtermusterformat für Dateneigenschaften	Resultierende Aktion
Kein JSON	Kein JSON	Lambda gibt zum Zeitpunkt der Erstellung oder Aktualisierung der Ereignisquellenzuordnung eine Ausnahme aus. Das Filtermuster für Dateneigenschaften muss ein gültiges JSON-Format haben.

Tutorial: Verwendung AWS Lambda mit Amazon DynamoDB DynamoDB-Streams

In diesem Tutorial erstellen Sie eine Lambda-Funktion zum Verarbeiten von Ereignissen aus einem Amazon-DynamoDB-Stream.

Voraussetzungen

Installieren Sie das AWS Command Line Interface

Wenn Sie das noch nicht installiert haben AWS Command Line Interface, folgen Sie den Schritten unter Installieren oder Aktualisieren der neuesten Version von AWS CLI, um es zu installieren.

Das Tutorial erfordert zum Ausführen von Befehlen ein Befehlszeilenterminal oder eine Shell. Verwenden Sie unter Linux und macOS Ihre bevorzugte Shell und Ihren bevorzugten Paketmanager.

> **Note**
>
> In Windows werden einige Bash-CLI-Befehle, die Sie häufig mit Lambda verwenden (z. B. `zip`), von den integrierten Terminals des Betriebssystems nicht unterstützt. Um eine in Windows integrierte Version von Ubuntu und Bash zu erhalten, installieren Sie das Windows-Subsystem für Linux.

Erstellen der Ausführungsrolle

Erstellen Sie die Ausführungsrolle, die Ihrer Funktion die Berechtigung zum Zugriff auf AWS Ressourcen erteilt.

So erstellen Sie eine Ausführungsrolle

1. Öffnen Sie die Seite Roles (Rollen) in der IAM-Konsole.
2. Wählen Sie Rolle erstellen aus.
3. Erstellen Sie eine Rolle mit den folgenden Eigenschaften.
   - Vertrauenswürdige Entität – Lambda.
   - Berechtigungen — AWSLambdaDBExecutionDynamo-Rolle.
   - Role name (Name der Rolle – **`lambda-dynamodb-role`**.

Die AWSLambdaDBExecutionDynamo-Rolle verfügt über die Berechtigungen, die die Funktion benötigt, um Elemente aus DynamoDB zu lesen und Protokolle in Logs zu schreiben. CloudWatch

Erstellen der Funktion

Erstellen Sie eine Lambda-Funktion, die Ihre DynamoDB-Ereignisse verarbeitet. Der Funktionscode schreibt einige der eingehenden Ereignisdaten in Logs. CloudWatch

.NET

SDK für .NET

> **Note**
>
> Es gibt noch mehr dazu GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines DynamoDB-Ereignisses mit Lambda unter Verwendung von .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

SDK für Go V2

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines DynamoDB-Ereignisses mit Lambda unter Verwendung von 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 für Java 2.x

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines DynamoDB-Ereignisses mit Lambda unter Verwendung von 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 für JavaScript (v3)

> **Note**
>
> Es gibt noch mehr dazu GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Konsumieren eines DynamoDB-Ereignisses mit Lambda unter Verwendung. 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)}`);
};
```

Konsumieren eines DynamoDB-Ereignisses mit Lambda unter Verwendung. 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

SDK für PHP

> **Note**
>
> Es gibt noch mehr dazu. GitHub Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines DynamoDB-Ereignisses mit Lambda unter Verwendung von 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

SDK für Python (Boto3)

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines DynamoDB-Ereignisses mit Lambda unter Verwendung von 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

SDK für Ruby

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines DynamoDB-Ereignisses mit Lambda unter Verwendung von 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 für Rust

> **Note**
>
> Es gibt noch mehr dazu GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines DynamoDB-Ereignisses mit Lambda unter Verwendung von 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(())

}
```

So erstellen Sie die Funktion

1. Kopieren Sie den Beispiel-Code in eine Datei mit dem Namen `example.js`.
2. Erstellen Sie ein Bereitstellungspaket.

```
zip function.zip example.js
```

3. Erstellen Sie eine Lambda-Funktion mit dem Befehl `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
```

Lambda-Funktion testen

In diesem Schritt rufen Sie Ihre Lambda-Funktion manuell mit dem `invoke` AWS Lambda CLI-Befehl und dem folgenden DynamoDB-Beispielereignis auf. Kopieren Sie den folgenden Code in eine Datei mit dem Namen `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"
        }
    ]
}
```

Führen Sie den Befehl `invoke` aus.

```
aws lambda invoke --function-name ProcessDynamoDBRecords \
    --cli-binary-format raw-in-base64-out \
    --payload file://input.txt outputfile.txt
```

Die cli-binary-format Option ist erforderlich, wenn Sie Version 2 verwenden. AWS CLI Um dies zur Standardeinstellung zu machen, führen Sie `aws configure set cli-binary-format raw-in-base64-out` aus. Weitere Informationen finden Sie unter Von AWS CLI unterstützte globale Befehlszeilenoptionen im AWS Command Line Interface -Benutzerhandbuch für Version 2.

Die Funktion gibt die Zeichenfolge `message` im Antworttext zurück.

Überprüfen Sie die Ausgabe in der Datei `outputfile.txt`.

Erstellen einer DynamoDB-Tabelle mit einem aktivierten Stream

Erstellen einer Amazon-DynamoDB-Tabelle mit einem aktivierten Stream.

So erstellen Sie eine DynamoDB-Tabelle

1. Öffnen Sie die DynamoDB-Konsole.
2. Wählen Sie Create table aus.
3. Erstellen Sie eine Tabelle mit den folgenden Einstellungen:
   - Tabellenname – **`lambda-dynamodb-stream`**
   - Primärschlüssel – **`id`** (Zeichenfolge)
4. Wählen Sie Erstellen aus.

So aktivieren Sie Streams

1. Öffnen Sie die DynamoDB-Konsole.
2. Wählen Sie Tables (Tabellen) aus.
3. Wählen Sie die lambda-dynamodb-stream Tabelle aus.
4. Wählen Sie unter Exports and streams (Exporte und Streams) die Option DynamoDB stream details (Details zu DynamoDB-Streams) aus.
5. Wählen Sie Turn on (Einschalten) aus.
6. Wählen Sie als Ansichtstyp die Option Nur Schlüsselattribute aus.
7. Wählen Sie Stream einschalten.

Notieren Sie sich den Stream-ARN. Sie benötigen diesen im nächsten Schritt, um die Lambda-Funktion dem Stream zuzuordnen. Weitere Informationen zur Aktivierung von Streams finden Sie unter Erfassen von Tabellenaktivitäten mit DynamoDB Streams.

Fügen Sie eine Ereignisquelle hinzu in AWS Lambda

Erstellen Sie eine Ereignisquellenzuordnung in AWS Lambda. Diese Ereignisquellenzuordnung ordnet den DynamoDB-Stream Ihrer Lambda-Funktion zu. Nachdem Sie diese Ereignisquellenzuordnung erstellt haben, AWS Lambda beginnt die Abfrage des Streams.

Führen Sie den Befehl AWS CLI `create-event-source-mapping` aus. Notieren Sie sich die UUID, nachdem Sie den Befehl ausgeführt haben. Sie benötigen diese UUID, um in Befehlen auf die Ereignisquellenzuordnung zu verweisen (beispielsweise beim Löschen der Ereignisquellen-Zuweisung).

```
aws lambda create-event-source-mapping --function-name ProcessDynamoDBRecords \
    --batch-size 100 --starting-position LATEST --event-source DynamoDB-stream-arn
```

Dadurch wird eine Zuweisung zwischen dem angegebenen DynamoDB-Stream und der Lambda-Funktion erstellt. Sie können einen DynamoDB-Stream mehreren Lambda-Funktionen und dieselbe Lambda-Funktion mehreren Streams zuordnen. Die Lambda-Funktionen teilen jedoch den Lesedurchsatz für den geteilten Stream.

Sie erhalten die Liste der Ereignisquellen-Zuweisungen, indem Sie folgenden Befehl ausführen.

```
aws lambda list-event-source-mappings
```

Die Liste gibt alle von Ihnen erstellten Ereignisquellenzuordnungen zurück und für jede Zuordnung zeigt sie u. a. den `LastProcessingResult`. Dieses Feld wird verwendet, um eine informative Meldung bereitzustellen, wenn Probleme auftreten. Werte wie `No records processed` (gibt an, dass die Abfrage nicht gestartet AWS Lambda wurde oder dass der Stream keine Datensätze enthält) und `OK` (gibt an, dass Datensätze AWS Lambda erfolgreich aus dem Stream gelesen und Ihre Lambda-Funktion aufgerufen wurden) zeigen an, dass keine Probleme vorliegen. Bei Problemen erhalten Sie eine Fehlermeldung.

Bei einer großen Anzahl von Ereignisquellen-Zuweisungen verwenden Sie den Funktionsnamen-Parameter, um die Ergebnisse einzugrenzen.

```
aws lambda list-event-source-mappings --function-name ProcessDynamoDBRecords
```

Testen der Einrichtung

Testen Sie das Erlebnis. end-to-end Beim Aktualisieren der Tabelle schreibt DynamoDB Ereignisdatensätze in den Stream. Da AWS Lambda den Stream abfragt, erkennt es neue Datensätze im Stream und ruft Ihre Lambda-Funktion für Sie auf, indem Ereignisse an die Funktion übergeben werden.

1. Fügen Sie in der DynamoDB-Konsole Elemente hinzu, aktualisieren und löschen Sie diese und fügen Sie diese hinzu. DynamoDB schreibt Datensätze dieser Aktionen in den Stream.
2. AWS Lambda fragt den Stream ab und wenn es Aktualisierungen am Stream erkennt, ruft es Ihre Lambda-Funktion auf, indem es die im Stream gefundenen Ereignisdaten weitergibt.
3. Ihre Funktion wird in Amazon ausgeführt und erstellt Protokolle CloudWatch. Sie können die in der CloudWatch Amazon-Konsole gemeldeten Protokolle überprüfen.

Nächste Schritte

In diesem Tutorial haben Sie die Grundlagen der Verarbeitung von DynamoDB-Stream-Ereignissen mit Lambda kennengelernt. Bei Produktions-Workloads sollten Sie erwägen, eine Logik für teilweise Batch-Antworten zu implementieren, um einzelne Datensatzfehler effizienter zu handhaben. Das Batchverarbeitungsprogramm von Powertools for AWS Lambda ist in Python TypeScript, .NET und Java verfügbar und bietet hierfür eine robuste Lösung, die die Komplexität von partiellen Batch-Antworten automatisch handhabt und die Anzahl der Wiederholungen für erfolgreich verarbeitete Datensätze reduziert.

Bereinigen Ihrer Ressourcen

Sie können jetzt die Ressourcen, die Sie für dieses Tutorial erstellt haben, löschen, es sei denn, Sie möchten sie behalten. Durch das Löschen von AWS Ressourcen, die Sie nicht mehr verwenden, vermeiden Sie unnötige Kosten für Ihre. AWS-Konto

So löschen Sie die Lambda-Funktion:

1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.
2. Wählen Sie die Funktion aus, die Sie erstellt haben.
3. Wählen Sie Aktionen, Löschen aus.
4. Geben Sie **`confirm`** in das Texteingabefeld ein und wählen Sie Delete (Löschen) aus.

So löschen Sie die Ausführungsrolle

1. Öffnen Sie die Seite Roles in der IAM-Konsole.
2. Wählen Sie die von Ihnen erstellte Ausführungsrolle aus.
3. Wählen Sie Löschen aus.
4. Geben Sie den Namen der Rolle in das Texteingabefeld ein und wählen Sie Delete (Löschen) aus.

So löschen Sie die DynamoDB-Tabelle

1. Öffnen Sie die Seite Tables (Tabellen) in der DynamoDB-Konsole.
2. Wählen Sie die von Ihnen erstellte Tabelle aus.
3. Wählen Sie Löschen aus.
4. Geben Sie **`delete`** in das Textfeld ein.
5. Wählen Sie Delete Table (Tabelle löschen).

Verarbeiten Sie Amazon EC2-Lebenszyklusereignisse mit einer Lambda-Funktion

Sie können AWS Lambda verwenden, um Lebenszyklusereignisse aus Amazon Elastic Compute Cloud zu verarbeiten und Amazon-EC2-Ressourcen zu verwalten. Amazon EC2 sendet Ereignisse an Amazon EventBridge (CloudWatch Events) für Lebenszyklus-Ereignisse, z. B. wenn eine Instance ihren Status ändert, wenn ein Volume-Snapshot von Amazon Elastic Block Store abgeschlossen wird oder wenn eine Spot Instance beendet werden soll. Sie konfigurieren EventBridge (CloudWatch Events), um diese Ereignisse zur Verarbeitung an eine Lambda-Funktion weiterzuleiten.

EventBridge (CloudWatch Events) ruft Ihre Lambda-Funktion asynchron mit dem Ereignisdokument von Amazon EC2 auf.

Example Instance-Lebenszyklusereignis

```
{
    "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"
    }
}
```

Einzelheiten zur Konfiguration von Ereignissen finden Sie unter Aufrufen einer Lambda-Funktion nach einem Zeitplan. Eine Beispielfunktion, die Amazon-EBS-Snapshot-Benachrichtigungen verarbeitet, finden Sie unter EventBridge Scheduler für Amazon EBS.

Sie können auch das AWS-SDK verwenden, um Instances und andere Ressourcen mit der Amazon-EC2-API zu verwalten.

Erteilen von Berechtigungen für EventBridge (CloudWatch Events)

Um Lebenszyklusereignisse von Amazon EC2 zu verarbeiten, benötigt EventBridge (CloudWatch Events) die Erlaubnis, Ihre Funktion aufzurufen. Diese Berechtigung stammt aus der ressourcenbasierten Richtlinie der Funktion. Wenn Sie die EventBridge (CloudWatch Events) Konsole verwenden, um einen Ereignisauslöser zu konfigurieren, aktualisiert die Konsole die ressourcenbasierte Richtlinie in Ihrem Namen. Andernfalls fügen Sie eine Anweisung wie die folgende hinzu:

Example Ressourcenbasierte Richtlinienanweisung für Amazon-EC2-Lebenszyklusbenachrichtigungen

```
{
  "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/*"
    }
  }
}
```

Verwenden Sie den `add-permission` AWS CLI-Befehl, um eine Anweisung hinzuzufügen.

```
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/*'
```

Wenn Ihre Funktion das AWS-SDK zum Verwalten von Amazon-EC2-Ressourcen verwendet, fügen Sie der Ausführungsrolle der Funktion Amazon-EC2-Berechtigungen hinzu.

Verarbeiten von Application Load Balancer-Anfragen mit Lambda

Sie können eine Lambda-Funktion verwenden, um Anforderungen aus einem Application Load Balancer zu verarbeiten. Elastic Load Balancing unterstützt Lambda-Funktionen als Ziel für Application Load Balancer. Sie können Load Balancer-Regeln zum Weiterleiten von HTTP-Anforderungen an eine Funktion verwenden, basierend auf Pfad- oder Header-Werten. Die Anforderung wird verarbeitet und es wird eine HTTP-Antwort aus Ihrer Lambda-Funktion zurückgegeben.

Elastic Load Balancing ruft Ihre Lambda-Funktion synchron mit einem Ereignis auf, das den Text und die Metadaten der Anforderung enthält.

Example Anforderungsereignis von einem 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
}
```

Ihre Funktion verarbeitet das Ereignis und gibt ein Antwortdokument in JSON an den Load Balancer zurück. Elastic Load Balancing konvertiert das Dokument in eine HTTP-Erfolgs- oder Fehlerantwort und gibt es an den Benutzer zurück.

Example Format des Antwortdokuments

```
{
    "statusCode": 200,
    "statusDescription": "200 OK",
    "isBase64Encoded": False,
    "headers": {
        "Content-Type": "text/html"
    },
    "body": "<h1>Hello from Lambda!</h1>"
}
```

Um eine Application Load Balancer als Funktionsauslöser zu konfigurieren, gewähren Sie Elastic Load Balancing die Berechtigung zur Ausführung der Funktion, erstellen eine Zielgruppe, die Anforderungen an die Funktion weiterleitet und fügen dem Load Balancer eine Regel hinzu, die Anforderungen an die Zielgruppe sendet.

Sie können den Befehl `add-permission` verwenden, um der ressourcenbasierten Richtlinie Ihrer Funktion eine Berechtigungsanweisung hinzuzufügen.

```
aws lambda add-permission --function-name alb-function \
--statement-id load-balancer --action "lambda:InvokeFunction" \
--principal elasticloadbalancing.amazonaws.com
```

Die Ausgabe sollte folgendermaßen aussehen:

```
{
    "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\"}"
}
```

Anweisungen zum Konfigurieren des Application Load Balancer-Listeners und der Zielgruppe finden Sie unter Lambda-Funktionen als Ziel im Benutzerhandbuch für Application Load Balancer.

Event-Handler von Powertools für Lambda AWS

Der Event-Handler aus dem Powertools for AWS Lambda-Toolkit bietet Routing, Middleware, CORS-Konfiguration, Generierung von OpenAPI-Spezifikationen, Anforderungsvalidierung, Fehlerbehandlung und andere nützliche Funktionen beim Schreiben von Lambda-Funktionen, die von einem Application Load Balancer aufgerufen werden. Das Dienstprogramm für den Ereignishandler ist für Python verfügbar. Weitere Informationen finden Sie unter Event Handler REST API in der Dokumentation zu 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)
```

Aufrufen einer Lambda-Funktion nach einem Zeitplan

Amazon EventBridge Scheduler ist ein Serverless-Scheduler, mit dem Sie Aufgaben von einem zentralen, verwalteten Service aus erstellen, ausführen und verwalten können. Mit EventBridge Scheduler können Sie mithilfe von Cron- und Rate-Ausdrücken Zeitpläne für wiederkehrende Muster erstellen oder einmalige Aufrufe konfigurieren. Sie können flexible Zeitfenster für die Zustellung einrichten, Wiederholungslimits definieren und die maximale Aufbewahrungszeit für unverarbeitete Ereignisse festlegen.

Wenn Sie EventBridge Scheduler mit Lambda einrichten, ruft EventBridge Scheduler Ihre Lambda-Funktion asynchron auf. Auf dieser Seite wird erläutert, wie Sie mit EventBridge Scheduler eine Lambda-Funktion nach einem Zeitplan aufrufen.

Einrichten der Ausführungsrolle

Wenn Sie einen neuen Zeitplan erstellen, muss EventBridge Scheduler über die Berechtigung verfügen, seinen Ziel-API-Vorgang in Ihrem Namen aufzurufen. Sie gewähren EventBridge Scheduler diese Berechtigungen mithilfe einer Ausführungsrolle. Die Berechtigungsrichtlinie, die Sie der Ausführungsrolle Ihres Zeitplans hinzufügen, definiert die erforderlichen Berechtigungen. Diese Berechtigungen hängen von der Ziel-API ab, die EventBridge Scheduler aufrufen soll.

Wenn Sie die EventBridge-Scheduler-Konsole zum Erstellen eines Zeitplans verwenden, wie im folgenden Verfahren, richtet EventBridge Scheduler automatisch eine Ausführungsrolle basierend auf Ihrem ausgewählten Ziel ein. Wenn Sie einen Zeitplan mit einem der EventBridge-Scheduler-SDKs (AWS CLI oder CloudFormation) erstellen möchten, müssen Sie über eine vorhandene Ausführungsrolle verfügen, die die Berechtigungen gewährt, die EventBridge Scheduler zum Aufrufen eines Ziels benötigt. Weitere Informationen zum manuellen Einrichten einer Ausführungsrolle für Ihren Zeitplan finden Sie unter Einrichten einer Ausführungsrolle im Benutzerhandbuch für EventBridge Scheduler.

Erstellen eines Zeitplans

So erstellen Sie einen Zeitplan mithilfe der Konsole

1. Öffnen Sie die Amazon-EventBridge-Scheduler-Konsole unter https://console.aws.amazon.com/scheduler/home.
2. Wählen Sie auf der Seite Zeitpläne die Option Zeitplan erstellen aus.

3. Gehen Sie auf der Seite Zeitplandetails angeben im Abschnitt Zeitplanname und -beschreibung wie folgt vor:

   a. Geben Sie unter Zeitplanname einen Namen für Ihren Zeitplan ein. Beispiel, **`MyTestSchedule`**.

   b. (Optional) Geben Sie unter Beschreibung eine Beschreibung für Ihren Zeitplan ein. Beispiel, **`My first schedule`**.

   c. Wählen Sie für Zeitplangruppe eine Zeitplangruppe aus der Dropdown-Liste aus. Wenn Sie noch keine Gruppe haben, wählen Sie Standard. Um eine Zeitplangruppe zu erstellen, wählen Sie Eigenen Zeitplan erstellen.

   Sie verwenden Zeitplangruppen, um Tags zu Zeitplangruppen hinzuzufügen.

4. • Wählen Sie Ihre Zeitplanoptionen.

Vorkommen	Vorgehensweise	
Einmaliger Zeitplan Ein einmaliger Zeitplan ruft ein Ziel nur einmal zu dem von Ihnen angegebenen Datum und der angegeben en Uhrzeit auf.	Gehen Sie für Datum und Uhrzeit wie folgt vor: • Geben Sie ein gültiges Datum im `YYYY/MM/DD` -Format ein. • Geben Sie einen Zeitstempel im 24-Stunden-Format (`hh:mm`) ein. • Wählen Sie unter Zeitzone die Zeitzone aus.	
Wiederkehrender Zeitplan Ein wiederkehrender Zeitplan ruft ein Ziel mit einer Rate auf, die Sie mit einem cron-Ausdruck	a. Gehen Sie bei Zeitplant yp wie folgt vor: • Um den Zeitplan mithilfe eines Cron-Ausdrucks zu definiere n, wählen Sie Cron-	

Vorkommen	Vorgehensweise	
oder einem Rate-Ausdruck angeben.	basierter Zeitplan und geben Sie den Cron-Ausdruck ein. • Um den Zeitplan mithilfe eines Rate-Ausdrucks zu definieren, wählen Sie Rate-Ausdruck ein. Weitere Informationen zu Cron- und Rate-Ausdrücken finden Sie unter Zeitplantypen im EventBridge Scheduler im Benutzerhandbuch zu Amazon EventBridge Scheduler. b. Wählen Sie für Flexibles Zeitfenster die Option Aus, um die Option zu deaktivieren, oder wählen Sie eines der vordefinierten Zeitfenster aus. Wenn Sie beispielsweise 15 Minuten auswählen und einen wiederkehrenden Zeitplan festlegen, der sein Ziel einmal pro Stunde aufruft, wird der Zeitplan innerhalb von 15 Minuten nach Beginn jeder Stunde ausgeführt.	

5. (Optional) Wenn Sie im vorherigen Schritt Wiederkehrender Zeitplan ausgewählt haben, gehen Sie im Abschnitt Zeitrahmen wie folgt vor:

   a. Wählen Sie unter Zeitzone eine Zeitzone aus.

   b. Geben Sie für Startdatum und -uhrzeit ein gültiges Datum im `YYYY/MM/DD`-Format ein und geben Sie dann einen Zeitstempel im 24-Stunden-Format (`hh:mm`) an.

   c. Geben Sie für Enddatum und -uhrzeit ein gültiges Datum im `YYYY/MM/DD`-Format ein und geben Sie dann einen Zeitstempel im 24-Stunden-Format (`hh:mm`) an.

6. Wählen Sie Weiter aus.

7. Wählen Sie auf der Seite Ziel auswählen den AWS-API-Vorgang aus, den EventBridge Scheduler aufruft:

   a. Wählen Sie AWS Lambda aufrufen aus.

   b. Wählen Sie im Abschnitt Aufrufen eine Funktion aus oder wählen Sie Neue Lambda-Funktion erstellen.

   c. (Optional) Geben Sie eine JSON-Nutzlast ein. Wenn Sie keine Nutzlast eingeben, verwendet EventBridge Scheduler ein leeres Ereignis, um die Funktion aufzurufen.

8. Wählen Sie Weiter aus.

9. Führen Sie auf der Seite Settings (Einstellungen) die folgenden Schritte aus:

   a. Um den Zeitplan zu aktivieren, schalten Sie unter Zeitplanstatus die Option Zeitplan aktivieren ein.

   b. Um eine Wiederholungsrichtlinie für Ihren Zeitplan zu konfigurieren, gehen Sie unter Wiederholungsrichtlinie und Warteschlange für unzustellbare Nachrichten (DLQ) wie folgt vor:

      - Aktivieren Sie die Option Wiederholen.
      - Geben Sie unter Maximales Alter des Ereignisses die maximale(n) Stunde(n) und Minute(n) ein, die EventBridge Scheduler ein unverarbeitetes Ereignis aufbewahren muss.
      - Die Höchstdauer beträgt 24 Stunden.
      - Geben Sie unter Maximale Wiederholungsversuche an, wie oft EventBridge Scheduler den Zeitplan maximal wiederholen soll, wenn das Ziel einen Fehler zurückgibt.

        Der Maximalwert beträgt 185 Wiederholungen.

Bei Wiederholungsrichtlinien führt EventBridge Scheduler den Zeitplan erneut aus, wenn ein Zeitplan sein Ziel nicht aufrufen kann. Falls konfiguriert, müssen Sie die maximale Aufbewahrungszeit und Wiederholungsversuche für den Zeitplan festlegen.

c. Wählen Sie aus, wo EventBridge Scheduler nicht zugestellte Ereignisse speichert.

Option für Warteschlange für unzustellbare Nachrichten (DLQ)	Vorgehensweise
Nicht speichern	Wählen Sie None.
Speichert das Ereignis im selben AWS-Konto, in dem Sie den Zeitplan erstellen	a. Wählen Sie Eine Amazon-SQS-Warteschlange in meinem AWS-Konto als DLQ auswählen. b. Wählen Sie den Amazon-Ressourcennamen (ARN) der Amazon SQS-Warteschlange.
Speichert das Ereignis in einem anderen AWS-Konto als dem, in dem Sie den Zeitplan erstellen	a. Wählen Sie Angeben einer Amazon-SQS-Warteschlange in anderen AWS-Konten als DLQ. b. Geben Sie den Amazon-Ressourcennamen (ARN) der Amazon-SQS-Warteschlange ein.

d. Um einen kundenverwalteten Schlüssel zur Verschlüsselung Ihrer Zieleingabe zu verwenden, wählen Sie unter Verschlüsselung die Option Verschlüsselungseinstellungen anpassen (erweitert).

Wenn Sie diese Option wählen, geben Sie einen vorhandenen CMK-ARN ein oder wählen Sie Erstellen eines AWS KMS key, um zur AWS KMS-Konsole zu navigieren. Weitere Informationen darüber, wie EventBridge Scheduler Ihre Daten im Ruhezustand verschlüsselt, finden Sie unter Verschlüsselung im Ruhezustand im Benutzerhandbuch zu Amazon EventBridge Scheduler.

e. Damit EventBridge Scheduler eine neue Ausführungsrolle für Sie erstellt, wählen Sie Neue Rolle für diesen Zeitplan erstellen. Geben Sie dann einen Namen für Rollenname ein. Wenn Sie diese Option wählen, fügt EventBridge Scheduler der Rolle die erforderlichen Berechtigungen für Ihr Vorlagenziel hinzu.

10. Wählen Sie Weiter aus.
11. Überprüfen Sie auf der Seite Zeitplan überprüfen und erstellen die Details Ihres Zeitplans. Wählen Sie in jedem Abschnitt Bearbeiten aus, um zu diesem Schritt zurückzukehren und seine Details zu bearbeiten.
12. Wählen Sie Zeitplan erstellen.

    Auf der Seite Zeitpläne können Sie eine Liste Ihrer neuen und vorhandenen Zeitpläne anzeigen. Überprüfen Sie in der Spalte Status, ob Ihr neuer Zeitplan aktiviert ist.

Um zu bestätigen, dass EventBridge Scheduler die Funktion aufgerufen hat, überprüfen Sie die Amazon CloudWatch Logs der Funktion.

Zugehörige Ressourcen

Weitere Informationen zum EventBridge Scheduler finden Sie hier:

- EventBridge-Scheduler-Benutzerhandbuch
- EventBridge-Scheduler-API-Referenz
- EventBridge Scheduler – Preisgestaltung

Verwenden von AWS Lambda mit AWS IoT

AWS IoT bietet eine sichere Kommunikation zwischen über das Internet verbundenen Geräten (z. B. Sensoren) und der AWS Cloud. Auf diese Weise können Sie die Telemetriedaten von mehreren Geräten erfassen, speichern und analysieren.

Sie können AWS IoT-Regeln für Ihre Geräte erstellen, um mit AWS-Services zu interagieren. Die AWS IoT-Regel-Engine bietet eine SQL-basierte Sprache zur Auswahl von Daten aus Nachrichten-Nutzlasten und zum Senden von Daten zu anderen Services, z. B. Amazon S3, Amazon DynamoDB und AWS Lambda. Sie definieren eine Regel zum Aufrufen einer Lambda-Funktion, wenn Sie einen anderen AWS-Service oder den Service eines Drittanbieters aufrufen möchten.

Wenn eine eingehende IoT-Nachricht die Regel auslöst, ruft AWS IoT Ihre Lambda-Funktion asynchron auf und übergibt Daten aus der IoT-Nachricht an die Funktion.

Das folgende Beispiel zeigt einen Feuchtigkeitsablesung von einem Gewächshaussensor. Die Werte row und pos bestimmen die Position des Sensors. Dieses Beispielereignis basiert auf dem Gewächshaustyp in den AWS IoT-Regel-Tutorials.

Example AWS IoT-Nachrichtenereignis

```
{
    "row" : "10",
    "pos" : "23",
    "moisture" : "75"
}
```

Bei asynchronem Aufruf stellt Lambda die Nachricht in die Warteschlange und wiederholt den Vorgang, wenn Ihre Funktion einen Fehler zurückgibt. Konfigurieren Sie Ihre Funktion mit einem Ziel, um Ereignisse beizubehalten, die Ihre Funktion nicht verarbeiten konnte.

Sie müssen die Berechtigung für den AWS IoT-Service erteilen, um Ihre Lambda-Funktion aufzurufen. Sie können den Befehl `add-permission` verwenden, um der ressourcenbasierten Richtlinie Ihrer Funktion eine Berechtigungsanweisung hinzuzufügen.

```
aws lambda add-permission --function-name my-function \
--statement-id iot-events --action "lambda:InvokeFunction" --principal
 iot.amazonaws.com
```

Die Ausgabe sollte folgendermaßen aussehen:

```
{
    "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\"}"
}
```

Weitere Informationen zur Verwendung von Lambda mit AWS IoT finden Sie unter Erstellen einer AWS Lambda-Regel.

Verarbeiten von Datensätzen aus Amazon Kinesis Data Streams mit Lambda

Sie können eine Lambda-Funktion verwenden, um Datensätze in einem Amazon Kinesis-Datenstrom zu verarbeiten. Sie können eine Lambda-Funktion zu einem Kinesis Data Streams Konsumenten mit gemeinsam genutztem Durchsatz (Standard-Iterator) oder zu einem Konsumenten mit dediziertem Durchsatz mit erweitertem Rundsenden zuweisen. Bei Standard-Iteratoren fragt Lambda jeden Shard in Ihrem Kinesis-Stream nach Datensätzen ab, die das HTTP-Protokoll verwenden. Die Ereignisquellenzuordnung teilt den Lesedurchsatz mit anderen Konsumenten des Shards zusammen.

Weitere Informationen zu Kinesis-Datenströmen finden Sie unter Daten aus Amazon Kinesis Data Streams.

> **Note**
>
> Kinesis berechnet Gebühren für jeden Shard, sowie bei verbessertem Rundsenden für Daten, die aus dem Stream gelesen werden. Details zu den Preisen finden Sie unter Amazon-Kinesis- Preise.

Abfragen und Stapeln von Streams

Lambda liest Datensätze aus dem Datenstrom und ruft Ihre Funktion synchron mit einem Ereignis auf, das Stream-Datensätze enthält. Lambda liest Datensätze in Batches und ruft Ihre Funktion auf, um Datensätze aus dem Batch zu verarbeiten. Jeder Batch enthält Datensätze aus einem einzelnen Shard/Datenstrom.

Ihre Lambda-Funktion ist eine Konsumentenanwendung für Ihren Daten-Stream. Sie verarbeitet jeweils einen Batch Datensätzen aus jedem Shard. Sie können eine Lambda-Funktion zu einem Konsumenten mit gemeinsam genutztem Durchsatz (Standard-Iterator) oder zu einem Konsumenten mit dediziertem Durchsatz mit erweitertem Rundsenden zuweisen.

- Standard-Iterator: Lambda fragt jeden Shard in Ihrem Kinesis-Stream mit einer Basisrate von einmal pro Sekunde nach Datensätzen ab. Wenn mehr Datensätze verfügbar sind, verarbeitet Lambda Batches, bis die Funktion mit dem Stream gleichzieht. Die Ereignisquellenzuordnung teilt den Lesedurchsatz mit anderen Konsumenten des Shards zusammen.
- Erweitertes Rundsenden: Um die Latenz zu minimieren und den Lesedurchsatz zu maximieren, erstellen Sie einen Daten-Stream-Konsumenten mit erweitertem Rundsenden. Stream-

Konsumenten mit erweitertem Rundsenden erhalten eine dedizierte Verbindung für jeden Shard, der keine Auswirkungen auf andere Anwendungen hat, die aus dem Stream lesen. Stream-Konsumenten verwenden HTTP/2, um die Latenz zu reduzieren, indem Datensätze über eine langlebige Verbindung an Lambda übertragen und Anforderungs-Header komprimiert werden. Sie können einen Stream-Konsumenten mit der Kinesis-RegisterStreamConsumer-API erstellen.

```
aws kinesis register-stream-consumer \
--consumer-name con1 \
--stream-arn arn:aws:kinesis:us-east-2:123456789012:stream/lambda-stream
```

Die Ausgabe sollte folgendermaßen aussehen:

```
{
    "Consumer": {
        "ConsumerName": "con1",
        "ConsumerARN": "arn:aws:kinesis:us-east-2:123456789012:stream/lambda-stream/
consumer/con1:1540591608",
        "ConsumerStatus": "CREATING",
        "ConsumerCreationTimestamp": 1540591608.0
    }
}
```

Um die Geschwindigkeit zu erhöhen, mit der Ihre Funktion Datensätze verarbeitet, fügen Sie Ihrem Datenstrom Shards hinzu. Lambda verarbeitet Datensätze in jedem Shard in der Reihenfolge. Es beendet die Verarbeitung zusätzlicher Datensätze in einem Shard, wenn Ihre Funktion einen Fehler zurückgibt. Mehr Shards bedeutet, dass mehr Stapel verarbeitet und gleichzeitig die Auswirkungen von Fehlern auf die Nebenläufigkeit verringert werden.

Wenn Ihre Funktion nicht hochskalieren kann, um alle gleichzeitigen Stapel zu verarbeiten, fordern Sie eine Kontingenterhöhung an oder reservieren Sie Gleichzeitigkeit für Ihre Funktion.

Standardmäßig ruft Lambda Ihre Funktion auf, sobald Datensätze verfügbar sind. Wenn der Batch, den Lambda aus der Ereignisquelle liest, nur einen Datensatz enthält, sendet Lambda nur einen Datensatz an die Funktion. Damit die Funktion nicht mit einer kleinen Anzahl von Datensätzen aufgerufen wird, können Sie die Ereignisquelle anweisen, Datensätze bis zu 5 Minuten lang zu puffern, indem Sie ein Batch-Fenster konfigurieren. Bevor die Funktion aufgerufen wird, liest Lambda so lange Datensätze aus der Ereignisquelle, bis es einen vollständigen Batch erfasst hat, das Batch-Verarbeitungsfenster abläuft oder der Batch die Nutzlastgrenze von 6 MB erreicht. Weitere Informationen finden Sie unter Batching-Verhalten.

> **Warning**
>
> Zuordnung von Lambda-Ereignisquellen verarbeiten jedes Ereignis mindestens einmal und es kann zu einer doppelten Verarbeitung von Datensätzen kommen. Um mögliche Probleme im Zusammenhang mit doppelten Ereignissen zu vermeiden, empfehlen wir Ihnen dringend, Ihren Funktionscode idempotent zu machen. Weitere Informationen finden Sie unter Wie mache ich meine Lambda-Funktion idempotent im AWS-Wissenszentrum.

Lambda wartet mit dem Senden des nächsten zu verarbeitenden Stapels nicht, bis ggf. konfigurierte Erweiterungen abgeschlossen sind. Anders ausgedrückt: Ihre Erweiterungen werden möglicherweise weiter ausgeführt, während Lambda den nächsten Stapel von Datensätzen verarbeitet. Dies kann zu Drosselungsproblemen führen, wenn Sie gegen eine Einstellung oder gegen einen Grenzwert im Zusammenhang mit der Parallelität Ihres Kontos verstoßen. Um zu erkennen, ob möglicherweise ein Problem vorliegt, müssen Sie Ihre Funktionen überwachen sowie überprüfen, ob für Ihre Zuordnung von Ereignisquellen unerwartet hohe Parallelitätsmetriken vorliegen. Aufgrund der kurzen Zeit zwischen den Aufrufen kann Lambda kurzzeitig eine höhere Gleichzeitigkeitsnutzung als die Anzahl der Shards melden. Dies kann sogar für Lambda-Funktionen ohne Erweiterungen gelten.

Konfigurieren Sie die ParallelizationFactor-Einstellung, um einen Shard eines Kinesis-Datenstroms mit mehr als einem Lambda-Aufruf gleichzeitig zu verarbeiten. Sie können die Anzahl der gleichzeitigen Batches angeben, die Lambda von einem Shard über einen Parallelisierungsfaktor von 1 (Standard) bis 10 abfragt. Wenn `ParallelizationFactor` beispielsweise auf 2 gesetzt ist, können Sie maximal 200 gleichzeitige Lambda-Aufrufe haben, um 100 Kinesis-Daten-Shards zu verarbeiten (in der Praxis werden womöglich andere Werte für die Metrik `ConcurrentExecutions` angezeigt). Dies hilft, den Verarbeitungsdurchsatz hochzuskalieren, wenn das Datenvolumen flüchtig ist und `IteratorAge` hoch ist. Wenn Sie die Anzahl gleichzeitiger Batches pro Shard erhöhen, stellt Lambda weiterhin die Auftragsverarbeitung auf Partitionsschlüsselebene sicher.

Sie können `ParallelizationFactor` auch mit der Kinesis-Aggregation verwenden. Das Verhalten der Zuordnung von Ereignisquellen hängt davon ab, ob Sie das erweiterte Rundsenden verwenden:

- Ohne erweitertes Rundsenden: Alle Ereignisse innerhalb eines aggregierten Ereignisses müssen denselben Partitionsschlüssel haben. Der Partitionsschlüssel muss außerdem mit dem des aggregierten Ereignisses übereinstimmen. Wenn die Ereignisse innerhalb des aggregierten Ereignisses unterschiedliche Partitionsschlüssel haben, kann Lambda nicht garantieren, dass die Ereignisse in der richtigen Reihenfolge nach Partitionsschlüssel verarbeitet werden.

- Mit verbessertem Rundsenden: Zunächst dekodiert Lambda das aggregierte Ereignis in seine einzelnen Ereignisse. Das aggregierte Ereignis kann einen anderen Partitionsschlüssel haben als die darin enthaltenen Ereignisse. Ereignisse, die nicht dem Partitionsschlüssel entsprechen, werden jedoch gelöscht und gehen verloren. Lambda verarbeitet diese Ereignisse nicht und sendet sie nicht an ein konfiguriertes Fehlerziel.

Beispielereignis

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"
        }
    ]
}
```

Verarbeiten von Amazon Kinesis Data Streams-Datensätzen mit Lambda

Um Amazon-Kinesis-Data-Streams-Datensätze mit Lambda zu verarbeiten, erstellen Sie eine Lambda-Zuordnung von Ereignisquellen. Sie können eine Lambda-Funktion zu einem Standard-Iterator oder zu einem Konsumenten für das erweiterte Rundsenden zuweisen. Weitere Informationen finden Sie unter Abfragen und Stapeln von Streams.

Erstellen einer Zuordnung von Ereignisquellen für Kinesis

Um Ihre Lambda-Funktion mit Datensätzen aus Ihrem Datenstrom aufzurufen, erstellen Sie eine Zuordnung von Ereignisquellen. Sie können mehrere Ereignisquellenzuordnungen erstellen, um gleiche Daten mit mehreren Lambda-Funktionen oder Elemente aus mehreren Daten-Streams mit nur einer Funktion zu verarbeiten. Bei der Verarbeitung von Elementen aus mehreren Datenströmen enthält jeder Batch Datensätze aus nur einem einzigen Shard oder Stream.

Sie können Zuordnungen von Ereignisquellen konfigurieren, um Datensätze aus einem Stream in einem anderen AWS-Konto zu verarbeiten. Weitere Informationen hierzu finden Sie unter the section called “Kontenübergreifende Zuordnungen”.

Bevor Sie ein Zuordnung von Ereignisquellen erstellen, müssen Sie Ihrer Lambda-Funktion die Berechtigung zum Lesen aus einem Kinesis-Datenstrom erteilen. Lambda benötigt die folgenden Berechtigungen zum Verwalten von Ressourcen, die zu Ihrem Kinesis-Datenstrom gehören:

- Kinese: DescribeStream
- Kinese: DescribeStreamSummary
- Kinese: GetRecords
- Kinese: GetShardIterator
- Kinese: ListShards

- Kinese: SubscribeToShard

Die AWS verwaltete Richtlinie AWSLambdaKinesisExecutionRoleumfasst diese Berechtigungen. Fügen Sie diese verwaltete Richtlinie zu Ihrer Funktion hinzu, wie im folgenden Verfahren beschrieben.

> **Note**
>
> - Sie benötigen keine `kinesis:ListStreams`-Berechtigung, um Zuordnungen von Ereignisquellen für Kinesis zu erstellen und zu verwalten. Wenn Sie jedoch eine Zuordnung von Ereignisquellen in der Konsole erstellen und nicht über diese Berechtigung verfügen, können Sie keinen Kinesis-Stream aus einer Dropdown-Liste auswählen und die Konsole zeigt einen Fehler an. Um die Zuordnung von Ereignisquellen zu erstellen, müssen Sie den Amazon-Ressourcennamen (ARN) Ihres Streams manuell eingeben.
> - Lambda ruft die APIs `kinesis:GetRecords` und `kinesis:GetShardIterator` auf, wenn fehlgeschlagene Aufrufe erneut versucht werden.

AWS-Managementkonsole

So fügen Sie Kinesis-Berechtigungen zu Ihrer Funktion hinzu

1. Öffnen Sie die Funktionsseite der Lambda-Konsole und wählen Sie Ihre Funktion aus.
2. Klicken Sie in der Registerkarte Konfiguration auf die Option Berechtigungen.
3. Wählen Sie im Bereich Ausführungsrolle unter Rollenname den Link zur Ausführungsrolle Ihrer Funktion aus. Dieser Link öffnet die Seite für diese Rolle in der IAM-Konsole.
4. Unter Berechtigungsrichtlinien im Abschnitt Berechtigungen hinzufügen wählen Sie dann Richtlinien anfügen aus.
5. Geben Sie im Suchfeld **`AWSLambdaKinesisExecutionRole`** ein.
6. Aktivieren Sie das Kontrollkästchen neben der Richtlinie und wählen Sie Berechtigung hinzufügen.

AWS CLI

So fügen Sie Kinesis-Berechtigungen zu Ihrer Funktion hinzu

- Führen Sie den folgenden CLI-Befehl aus, um die `AWSLambdaKinesisExecutionRole`-Richtlinie zur Ausführungsrolle Ihrer Funktion hinzuzufügen:

```
aws iam attach-role-policy \
--role-name MyFunctionRole \
--policy-arn arn:aws:iam::aws:policy/service-role/AWSLambdaKinesisExecutionRole
```

AWS SAM

So fügen Sie Kinesis-Berechtigungen zu Ihrer Funktion hinzu

- Fügen Sie die `Policies`-Eigenschaft wie im folgenden Beispiel gezeigt in der Definition Ihrer Funktion hinzu:

```
Resources:
  MyFunction:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: ./my-function/
      Handler: index.handler
      Runtime: nodejs24.x
      Policies:
        - AWSLambdaKinesisExecutionRole
```

Nachdem Sie die erforderlichen Berechtigungen konfiguriert haben, erstellen Sie die Zuordnung von Ereignisquellen.

AWS-Managementkonsole

So erstellen Sie die Kinesis-Zuordnung von Ereignisquellen

1. Öffnen Sie die Funktionsseite der Lambda-Konsole und wählen Sie Ihre Funktion aus.
2. Wählen Sie im Bereich Function overview (Funktionsübersicht) die Option Add trigger (Auslöser hinzufügen).
3. Wählen Sie unter Trigger-Konfiguration als Quelle die Option Kinesis aus.

4. Wählen Sie den Kinesis-Stream, für den Sie die Zuordnung von Ereignisquellen erstellen möchten und optional einen Konsumenten Ihres Streams.
5. (Optional) Bearbeiten Sie die Batch-Größe, die Startposition und das Batch-Fenster für Ihre Zuordnung von Ereignisquellen.
6. Wählen Sie Hinzufügen aus.

Wenn Sie Ihre Ereignisquellenzuordnung von der Konsole aus erstellen, muss Ihre IAM-Rolle über die Berechtigungen kinesis: ListStreams und kinesis: verfügen. ListStreamConsumers

AWS CLI

So erstellen Sie die Kinesis-Zuordnung von Ereignisquellen

- Führen Sie den folgenden CLI-Befehl aus, um eine Kinesis-Zuordnung von Ereignisquellen zu erstellen. Wählen Sie Ihre eigene Batch-Größe und Startposition entsprechend Ihrem Anwendungsfall.

```
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
```

Um ein Batching-Fenster anzugeben, fügen Sie die `--maximum-batching-window-in-seconds`-Option hinzu. Weitere Informationen zur Verwendung dieses und anderer Parameter finden Sie create-event-source-mappingin der AWS CLI Befehlsreferenz.

AWS SAM

So erstellen Sie die Kinesis-Zuordnung von Ereignisquellen

- Fügen Sie die `KinesisEvent`-Eigenschaft wie im folgenden Beispiel gezeigt in der Definition Ihrer Funktion hinzu:

```
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
```

Weitere Informationen zum Erstellen einer Ereignisquellenzuordnung für Kinesis Data Streams finden Sie unter Kinesis im AWS Serverless Application Model Entwicklerhandbuch. AWS SAM

Abfrage und Startposition des Streams

Beachten Sie, dass die Stream-Abfrage bei der Erstellung und Aktualisierung der Zuordnung von Ereignisquellen letztendlich konsistent ist.

- Bei der Erstellung der Zuordnung von Ereignisquellen kann es mehrere Minuten dauern, bis mit der Abfrage von Ereignissen aus dem Stream begonnen wird.
- Bei Aktualisierungen der Zuordnung von Ereignisquellen kann es mehrere Minuten dauern, bis die Abfrage von Ereignissen aus dem Stream gestoppt und neu gestartet wird.

Dieses Verhalten bedeutet, dass, wenn Sie `LATEST` als Startposition für den Stream angeben, die Zuordnung von Ereignisquellen bei der Erstellung oder Aktualisierung möglicherweise Ereignisse übersieht. Um sicherzustellen, dass keine Ereignisse übersehen werden, geben Sie die Startposition des Streams als `TRIM_HORIZON` oder `AT_TIMESTAMP` an.

Erstellen einer kontoübergreifenden Zuordnung von Ereignisquellen

Amazon Kinesis Data Streams unterstützt ressourcenbasierte Richtlinien. Aus diesem Grund können Sie in einen Stream aufgenommene Daten in einem Konto AWS-Konto mit einer Lambda-Funktion in einem anderen Konto verarbeiten.

Um eine Ereignisquellenzuordnung für Ihre Lambda-Funktion mithilfe eines Kinesis-Streams in einem anderen zu erstellen AWS-Konto, müssen Sie den Stream mithilfe einer ressourcenbasierten Richtlinie konfigurieren, um Ihrer Lambda-Funktion die Berechtigung zum Lesen von Elementen zu erteilen. Informationen dazu, wie Sie Ihren Stream so konfigurieren, dass er kontoübergreifenden Zugriff ermöglicht, finden Sie unter Zugriff mit kontoübergreifenden AWS Lambda Funktionen teilen im Amazon Kinesis Streams Streams-Entwicklerhandbuch.

Sobald Sie Ihren Stream mit einer ressourcenbasierten Richtlinie konfiguriert haben, die Ihrer Lambda-Funktion die erforderlichen Berechtigungen erteilt, erstellen Sie die Zuordnung von Ereignisquellen mit einer der im vorherigen Abschnitt beschriebenen Methoden.

Wenn Sie Ihre Zuordnung von Ereignisquellen über die Lambda-Konsole erstellen möchten, fügen Sie den ARN Ihres Streams direkt in das Eingabefeld ein. Wenn Sie einen Verbraucher für Ihren Stream angeben möchten, wird durch Einfügen der ARN des Verbrauchers automatisch das Stream-Feld ausgefüllt.

Konfigurieren einer teilweisen Batch-Antwort mit Kinesis Data Streams und Lambda

Beim Konsumieren und Verarbeiten von Streaming-Daten aus einer Ereignisquelle werden standardmäßig Lambda-Checkpoints auf die höchste Sequenznummer eines Batches nur dann überprüft, wenn der Batch ein voller Erfolg ist. Lambda behandelt alle anderen Ergebnisse als einen vollständigen Fehler und versucht, den Batch bis zum Wiederholungslimit zu verarbeiten. Um beim Verarbeiten von Stapeln aus einem Stream Teilerfolge zu ermöglichen, aktivieren Sie `ReportBatchItemFailures`. Das Zulassen von Teilerfolgen kann dazu beitragen, die Anzahl der Wiederholungen in einer Aufzeichnung zu reduzieren, obwohl die Möglichkeit von Wiederholungen in einer erfolgreichen Aufzeichnung nicht vollständig verhindert wird.

Zum Aktivieren von `ReportBatchItemFailures` fügen Sie den Enum-Wert **`ReportBatchItemFailures`** der FunctionResponseTypes-Liste hinzu. Diese Liste zeigt an, welche Antworttypen für Ihre Funktion aktiviert sind. Sie können diese Liste konfigurieren, wenn Sie eine Zuordnung von Ereignisquellen erstellen oder aktualisieren.

> **Note**
>
> Selbst wenn Ihr Funktionscode teilweise Antworten bei Batch-Fehlern zurückgibt, werden diese Antworten nicht von Lambda verarbeitet, es sei denn, das

`ReportBatchItemFailures`-Feature ist explizit für Ihre Zuordnung von Ereignisquellen aktiviert.

Berichtsyntax

Beim Konfigurieren von Berichten zu Batch-Elementfehlern wird die `StreamsEventResponse`-Klasse mit einer Liste von Batch-Elementfehlern zurückgegeben. Sie können ein `StreamsEventResponse`-Objekt verwenden, um die Sequenznummer des ersten fehlgeschlagenen Datensatzes im Batch zurückzugeben. Sie können auch Ihre eigene benutzerdefinierte Klasse mit der richtigen Antwortsyntax erstellen. Die folgende JSON-Struktur zeigt die erforderliche Antwortsyntax:

```
{
  "batchItemFailures": [
        {
            "itemIdentifier": "<SequenceNumber>"
        }
    ]
}
```

Note

Wenn das `batchItemFailures`-Array mehrere Elemente enthält, verwendet Lambda den Datensatz mit der niedrigsten Sequenznummer als Kontrollpunkt. Lambda wiederholt dann alle Datensätze ab diesem Kontrollpunkt.

Erfolgs- und Misserfolgsbedingungen

Lambda behandelt einen Batch als vollständigen Erfolg, wenn Sie eines der folgenden Elemente zurückgeben:

- Eine leere `batchItemFailure`-Liste
- Eine ungültige `batchItemFailure`-Liste
- Ein leeres `EventResponse`
- Ein ungültiges `EventResponse`

Lambda behandelt einen Batch als vollständigen Misserfolg, wenn Sie eines der folgenden Elemente zurückgeben:

- Eine leere Zeichenfolge `itemIdentifier`
- Ein ungültiges `itemIdentifier`
- Ein `itemIdentifier` mit einem falschen Schlüsselnamen

Lambda wiederholt Fehler basierend auf Ihrer Wiederholungsstrategie.

Einen Batch halbieren

Wenn Ihr Aufruf fehlschlägt und `BisectBatchOnFunctionError` eingeschaltet ist, wird der Stapel unabhängig von Ihrer `ReportBatchItemFailures`-Einstellung halbiert.

Wenn eine partielle Batch-Erfolgsantwort empfangen wird und sowohl `BisectBatchOnFunctionError` als auch `ReportBatchItemFailures` aktiviert sind, wird der Batch mit der zurückgegebenen Sequenznummer halbiert und Lambda versucht nur die verbleibenden Datensätze erneut.

Um die Implementierung der partiellen Batch-Antwortlogik zu vereinfachen, sollten Sie in Erwägung ziehen, das Batch Processor Utility von Powertools for zu verwenden AWS Lambda, das diese Komplexität automatisch für Sie erledigt.

Hier sind einige Beispiele für Funktionscode, der die Liste der fehlgeschlagenen Nachrichten IDs im Batch zurückgibt:

.NET

SDK für .NET

> **Note**
>
> Es gibt noch mehr dazu GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Melden von Fehlern bei Kinesis-Batchelementen mit Lambda unter Verwendung von .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

SDK für Go V2

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Melden von Fehlern bei Kinesis-Batchelementen mit Lambda unter Verwendung von 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 für Java 2.x

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Melden von Fehlern bei Kinesis-Batchelementen mit Lambda unter Verwendung von 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 für JavaScript (v3)

> **Note**
>
> Es gibt noch mehr dazu GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Melden von Fehlern bei Kinesis-Batchelementen mit Lambda unter Verwendung von 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;
}
```

Melden von Fehlern Kinesis Kinesis-Batch-Elementen mit Lambda unter Verwendung von. 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

SDK für PHP

> **Note**
>
> Es gibt noch mehr dazu. GitHub Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Melden von Fehlern bei Kinesis-Batchelementen mit Lambda unter Verwendung von 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

SDK für Python (Boto3)

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Melden von Fehlern bei Kinesis-Batchelementen mit Lambda unter Verwendung von 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

SDK für Ruby

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Melden von Fehlern bei Kinesis-Batchelementen mit Lambda unter Verwendung von 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 für Rust

> **Note**
>
> Es gibt noch mehr dazu GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Melden von Fehlern bei Kinesis-Batchelementen mit Lambda unter Verwendung von 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
}
```

Powertools für die AWS Lambda Batch-Verarbeitung verwenden

Das Batchverarbeitungsprogramm von Powertools for verarbeitet AWS Lambda automatisch die Logik für partielle Batch-Antworten und reduziert so die Komplexität der Implementierung von Batch-Fehlerberichten. Hier sind Beispiele für die Verwendung des Batch-Prozessors:

Python

> **Note**
>
> Umfassende Beispiele und Anweisungen zur Einrichtung finden Sie in der Dokumentation zum Batch-Prozessor.

Verarbeitung von Kinesis Data Streams Streams-Stream-Datensätzen mit einem AWS Lambda Batch-Prozessor.

```
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**
>
> Umfassende Beispiele und Anweisungen zur Einrichtung finden Sie in der Dokumentation zum Batch-Prozessor.

Verarbeitung von Kinesis Data Streams Streams-Stream-Datensätzen mit einem AWS Lambda Batch-Prozessor.

```
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**
>
> Umfassende Beispiele und Anweisungen zur Einrichtung finden Sie in der Dokumentation zum Batch-Prozessor.

Verarbeitung von Kinesis Data Streams Streams-Stream-Datensätzen mit einem AWS Lambda Batch-Prozessor.

```
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**
>
> Umfassende Beispiele und Anweisungen zur Einrichtung finden Sie in der Dokumentation zum Batch-Prozessor.

Verarbeitung von Kinesis Data Streams Streams-Stream-Datensätzen mit einem AWS Lambda Batch-Prozessor.

```
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;
    }
}
```

Aufbewahrung verworfener Batch-Datensätze für eine Kinesis Data Streams-Ereignisquelle in Lambda

Die Fehlerbehandlung für Kinesis-Zuordnungen von Ereignisquellen hängt davon ab, ob der Fehler vor dem Aufruf der Funktion oder während des Funktionsaufrufs auftritt:

- Vor dem Aufruf: Wenn eine Lambda-Ereignisquellenzuordnung die Funktion aufgrund von Drosselung oder anderen Problemen nicht aufrufen kann, versucht sie es erneut, bis die Datensätze ablaufen oder das in der Ereignisquellenzuordnung () konfigurierte Höchstalter überschreiten. MaximumRecordAgeInSeconds
- Während des Aufrufs: Wenn die Funktion aufgerufen wird, aber einen Fehler zurückgibt, versucht Lambda es erneut, bis die Datensätze ablaufen, das Höchstalter (MaximumRecordAgeInSeconds) überschreiten oder die konfigurierte Wiederholungsquote () erreicht haben. MaximumRetryAttempts Bei Funktionsfehlern können Sie auch konfigurieren BisectBatchOnFunctionError, dass ein

fehlgeschlagener Batch in zwei kleinere Batches aufgeteilt wird, wodurch fehlerhafte Datensätze isoliert und Timeouts vermieden werden. Durch das Aufteilen von Batches wird die Quote für Wiederholungen nicht verbraucht.

Wenn die Fehlerbehandlungsmaßnahmen fehlschlagen, verwirft Lambda die Datensätze und setzt die Verarbeitung von Batches aus dem Stream fort. Bei den Standardeinstellungen bedeutet dies, dass ein fehlerhafter Datensatz die Verarbeitung auf dem betroffenen Shard für bis zu eine Woche blockieren kann. Um dies zu vermeiden, konfigurieren Sie die Ereignisquellenzuordnung Ihrer Funktion mit einer angemessenen Anzahl von Wiederholungen und einem maximalen Datensatzalter, das zu Ihrem Anwendungsfall passt.

Konfigurieren von Zielen für fehlgeschlagene Aufrufe

Um Datensätze zu fehlgeschlagenen Aufrufen zur Zuordnung von Ereignisquellen beizubehalten, fügen Sie der Zuordnung von Ereignisquellen Ihrer Funktion ein Ziel hinzu. Jeder an das Ziel gesendete Datensatz ist ein JSON-Dokument mit Metadaten über den fehlgeschlagenen Aufruf. Bei Amazon S3-Zielen sendet Lambda auch den gesamten Aufrufdatensatz zusammen mit den Metadaten. Sie können jedes Amazon SNS SNS-Thema, jede Amazon SQS SQS-Warteschlange, jeden Amazon S3 S3-Bucket oder Kafka als Ziel konfigurieren.

Bei Amazon-S3-Zielen können Sie das Feature Amazon S3 Event Notifications verwenden, um Benachrichtigungen zu erhalten, wenn Objekte in Ihren S3-Ziel-Bucket hochgeladen werden. Sie können S3-Ereignisbenachrichtigungen auch so konfigurieren, dass sie eine andere Lambda-Funktion aufrufen, um eine automatische Verarbeitung für fehlgeschlagene Stapel durchzuführen.

Ihre Ausführungsrolle muss über Berechtigungen für das Ziel verfügen:

- Für ein SQS-Ziel: sqs: SendMessage
- Für ein SNS-Ziel: sns:Publish
- Für ein S3-Ziel: s3: und s3: PutObject ListBucket
- Für ein Kafka-Ziel: kafka-cluster: WriteData

Sie können ein Kafka-Thema als Ziel für den Fall eines Fehlers für Ihre Quellenzuordnungen für Kafka-Ereignisse konfigurieren. Wenn Lambda Datensätze nach anstrengenden Wiederholungsversuchen nicht verarbeiten kann oder wenn Datensätze das Höchstalter überschreiten, sendet Lambda die fehlgeschlagenen Datensätze zur späteren Verarbeitung an das

angegebene Kafka-Thema. Weitere Informationen finden Sie unter the section called “Kafka-Ziel bei einem Ausfall”.

Wenn Sie die Verschlüsselung mit Ihrem eigenen KMS-Schlüssel für ein S3-Ziel aktiviert haben, muss die Ausführungsrolle Ihrer Funktion auch die Berechtigung haben, kms: aufzurufen. GenerateDataKey Wenn sich der KMS-Schlüssel und das S3-Bucket-Ziel in einem anderen Konto als Ihre Lambda-Funktion und Ausführungsrolle befinden, konfigurieren Sie den KMS-Schlüssel so, dass er der Ausführungsrolle vertraut, die zugelassen kms: GenerateDataKey wird.

Gehen Sie folgendermaßen vor, um ein Ausfallziel mit der Konsole zu konfigurieren:

1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.
2. Wählen Sie eine Funktion aus.
3. Wählen Sie unter Function overview (Funktionsübersicht) die Option Add destination (Ziel hinzufügen).
4. Wählen Sie als Quelle die Option Aufruf der Zuordnung von Ereignisquellen aus.
5. Wählen Sie für die Zuordnung von Ereignisquellen eine Ereignisquelle aus, die für diese Funktion konfiguriert ist.
6. Wählen Sie für Bedingung die Option Bei Ausfall aus. Für Aufrufe zur Zuordnung von Ereignisquellen ist dies die einzig akzeptierte Bedingung.
7. Wählen Sie unter Zieltyp den Zieltyp aus, an den Lambda Aufrufdatensätze sendet.
8. Wählen Sie unter Destination (Ziel) eine Ressource aus.
9. Wählen Sie Speichern.

Sie können mit AWS Command Line Interface (AWS CLI) auch ein Ziel für den Fall eines Fehlers konfigurieren. Mit dem folgenden create-event-source-mappingBefehl wird beispielsweise eine Zuordnung der Ereignisquelle mit einem SQS-Ziel für den Fall eines Fehlers hinzugefügt: `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"}}'
```

Der folgende update-event-source-mappingBefehl aktualisiert eine Ereignisquellenzuordnung, sodass fehlgeschlagene Aufrufdatensätze nach zwei Wiederholungsversuchen oder wenn die Datensätze älter als eine Stunde sind, an ein SNS-Ziel gesendet werden.

```
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"}}'
```

Aktualisierte Einstellungen werden asynchron angewendet und werden erst nach Abschluss des Vorgangs in der Ausgabe berücksichtigt. Verwenden Sie den get-event-source-mapping-Befehl, um den aktuellen Status anzuzeigen.

Um ein Ziel zu entfernen, geben Sie eine leere Zeichenfolge als Argument für den `destination-config`-Parameter an:

```
aws lambda update-event-source-mapping \
--uuid f89f8514-cdd9-4602-9e1f-01a5b77d449b \
--destination-config '{"OnFailure": {"Destination": ""}}'
```

Bewährte Methoden für die Sicherheit in Amazon S3-Zielen

Das Löschen eines S3-Buckets, der als Ziel konfiguriert ist, ohne das Ziel aus der Konfiguration Ihrer Funktion zu entfernen, kann ein Sicherheitsrisiko darstellen. Wenn ein anderer Benutzer den Namen Ihres Ziel-Buckets kennt, kann er den Bucket in seinem AWS-Konto neu erstellen. Aufzeichnungen über fehlgeschlagene Aufrufe werden an den entsprechenden Bucket gesendet, wodurch möglicherweise Daten aus Ihrer Funktion verfügbar gemacht werden.

> **Warning**
>
> Um sicherzustellen, dass Aufrufdatensätze Ihrer Funktion nicht an einen S3-Bucket in einem anderen gesendet werden können AWS-Konto, fügen Sie der Ausführungsrolle Ihrer Funktion eine Bedingung hinzu, die die `s3:PutObject` Berechtigungen auf Buckets in Ihrem Konto beschränkt.

-Das folgende Beispiel zeigt eine IAM-Richtlinie, die die `s3:PutObject`-Berechtigungen Ihrer Funktion auf Buckets in Ihrem Konto beschränkt. Diese Richtlinie gibt Lambda auch die `s3:ListBucket`-Erlaubnis, einen S3-Bucket als Ziel zu verwenden.

```
{
    "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"
                }
            }
        }
    ]
}
```

Um der Ausführungsrolle Ihrer Funktion mithilfe von AWS-Managementkonsole oder eine Berechtigungsrichtlinie hinzuzufügen AWS CLI, lesen Sie die Anweisungen in den folgenden Verfahren:

Console

So fügen Sie der Ausführungsrolle einer Funktion (Konsole) eine Berechtigungsrichtlinie hinzu

1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.
2. Wählen Sie die Lambda-Funktion aus, deren Ausführungsrolle Sie ändern möchten.
3. Klicken Sie in der Registerkarte Konfiguration auf die Option Berechtigungen.
4. Wählen Sie auf der Registerkarte Ausführungsrolle den Rollennamen Ihrer Funktion aus, um die IAM-Konsolenseite der Rolle zu öffnen.

5. Fügen Sie der Rolle wie folgt eine Richtlinie mit Berechtigungen hinzu:

   a. Wählen Sie im Bereich Berechtigungsrichtlinien die Optionen Berechtigungen hinzufügen und dann Inline-Richtlinie erstellen aus.

   b. Wählen Sie im Richtlinien-Editor JSON aus.

   c. Fügen Sie die Richtlinie, die Sie hinzufügen möchten, in den Editor ein (indem Sie die vorhandene JSON-Datei ersetzt) und wählen Sie dann Weiter aus.

   d. Geben Sie unter Richtliniendetails für den Richtliniennamen ein.

   e. Wählen Sie Richtlinie erstellen aus.

AWS CLI

So fügen Sie der Ausführungsrolle einer Funktion (CLI) eine Berechtigungsrichtlinie hinzu

1. Erstellen Sie ein JSON-Richtliniendokument mit den erforderlichen Berechtigungen und speichern Sie es in einem lokalen Verzeichnis.
2. Verwenden Sie den `put-role-policy` IAM-CLI-Befehl, um die Berechtigungen zur Ausführungsrolle Ihrer Funktion hinzuzufügen. Führen Sie den folgenden Befehl in dem Verzeichnis aus, in dem Sie Ihr JSON-Richtliniendokument gespeichert haben und ersetzen Sie den Rollennamen, den Richtliniennamen und das Richtliniendokument durch Ihre eigenen Werte.

   ```
   aws iam put-role-policy \
   --role-name my_lambda_role \
   --policy-name LambdaS3DestinationPolicy \
   --policy-document file://my_policy.json
   ```

Beispiel für einen Amazon SNS- und Amazon SQS-Aufrufsatz

Das folgende Beispiel zeigt, was Lambda bei einem fehlgeschlagenen Aufruf der Kinesis-Ereignisquelle an eine SQS-Warteschlange oder ein SNS-Thema sendet. Da Lambda nur die Metadaten für diese Zieltypen sendet, verwenden Sie die Felder `streamArn`, `shardId`, `startSequenceNumber` und `endSequenceNumber`, um den vollständigen Originaldatensatz abzurufen. Alle in der `KinesisBatchInfo`-Eigenschaft angezeigten Felder sind immer vorhanden.

```
{
    "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"
    }
}
```

Sie können diese Informationen verwenden, um die betroffenen Datensätze aus dem Stream für die Fehlersuche abzurufen. Die tatsächlichen Datensätze sind nicht enthalten, daher müssen Sie diesen Datensatz verarbeiten und aus dem Stream abrufen, bevor sie ablaufen und verloren gehen.

Beispiel eines Amazon S3-Aufrufdatensatzes

Das folgende Beispiel zeigt, was Lambda bei einem fehlgeschlagenen Aufruf der Kinesis-Ereignisquelle an einen Amazon S3-Bucket sendet. Zusätzlich zu allen Feldern aus dem vorherigen Beispiel für SQS- und SNS-Ziele enthält das Feld `payload` den ursprünglichen Aufrufdatensatz als maskierte JSON-Zeichenfolge.

```
{
    "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
}
```

Das S3-Objekt, das den Aufrufdatensatz enthält, verwendet die folgende Namenskonvention:

```
aws/lambda/<ESM-UUID>/<shardID>/YYYY/MM/DD/YYYY-MM-DDTHH.MM.SS-<Random UUID>
```

Implementierung der zustandsabhängigen Verarbeitung von Kinesis-Datenströmen in Lambda

Lambda-Funktionen können kontinuierliche Stream-Verarbeitungsanwendungen ausführen. Ein Stream entspricht einer unbegrenzten Menge von Daten, die kontinuierlich durch Ihre Anwendung fließen. Um Informationen aus dieser sich ständig aktualisierenden Eingabe zu analysieren, können Sie die enthaltenen Datensätze mithilfe eines zeitlich definierten Fensters binden.

Rollierende Fenster sind unterschiedliche Zeitfenster, die sich in regelmäßigen Abständen öffnen und schließen. Standardmäßig sind Lambda-Aufrufe zustandslos – Sie können sie nicht für die Verarbeitung von Daten über mehrere kontinuierliche Aufrufe hinweg ohne eine externe Datenbank verwenden. Mit rollierenden Fenstern können Sie jedoch Ihren Status über Aufrufe hinweg beibehalten. Dieser Zustand enthält das Gesamtergebnis der Nachrichten, die zuvor für das aktuelle Fenster verarbeitet wurden. Ihr Zustand kann maximal 1 MB pro Shard betragen. Wenn er diese Größe überschreitet, wird Lambda das Fenster vorzeitig beenden.

Jeder Datensatz in einem Stream gehört zu einem bestimmten Fenster. Lambda verarbeitet jeden Datensatz mindestens einmal, garantiert jedoch nicht, dass jeder Datensatz nur einmal verarbeitet wird. In seltenen Fällen, etwa bei der Fehlerbehandlung, werden einige Datensätze möglicherweise mehrmals verarbeitet. Datensätze werden beim ersten Mal immer in der richtigen Reihenfolge verarbeitet. Wenn Datensätze mehr als einmal verarbeitet werden, werden sie nicht in der richtigen Reihenfolge verarbeitet.

Aggregation und Verarbeitung

Ihre benutzerverwaltete Funktion wird sowohl zur Aggregation als auch zur Verarbeitung der Endergebnisse dieser Aggregation aufgerufen. Lambda aggregiert alle im Fenster empfangenen Datensätze. Sie können diese Datensätze in mehreren Stapeln erhalten, jeweils als ein separater Aufruf. Jeder Aufruf erhält einen Zustand. Wenn Sie also rollierende Fenster verwenden, muss Ihre Lambda-Funktionsantwort eine `state`-Eigenschaft enthalten. Wenn die Antwort keine `state`-Eigenschaft enthält, betrachtet Lambda dies als fehlgeschlagenen Aufruf. Um diese Bedingung zu erfüllen, kann Ihre Funktion ein `TimeWindowEventResponse`-Objekt zurückgeben, das die folgende JSON-Form aufweist:

Example **`TimeWindowEventResponse`**-Werte

```
{
    "state": {
        "1": 282,
        "2": 715
    },
    "batchItemFailures": []
}
```

> **Note**
>
> Für Java-Funktionen empfehlen wir, eine `Map<String, String>` zu verwenden, um den Status darzustellen.

Am Ende des Fensters wird das Flag `isFinalInvokeForWindow` auf `true` gesetzt, um anzugeben, dass es sich um den Endzustand handelt und dass es für die Verarbeitung bereit ist. Nach der Verarbeitung werden das Fenster und Ihr endgültiger Aufruf wird abgeschlossen, und dann wird der Zustand gelöscht.

Am Ende Ihres Fensters verwendet Lambda die endgültige Verarbeitung für Aktionen an den Aggregationsergebnissen. Ihre endgültige Verarbeitung wird synchron aufgerufen. Nach erfolgreichem Aufruf zeigt Ihre Funktion auf die Sequenznummer und die Stream-Verarbeitung wird fortgesetzt. Wenn der Aufruf nicht erfolgreich ist, unterbricht Ihre Lambda-Funktion die weitere Verarbeitung bis zu einem erfolgreichen Aufruf.

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
```

```
}
```

Konfiguration

Sie können rollierende Fenster konfigurieren, wenn Sie eine Ereignisquellenzuordnung erstellen oder aktualisieren. Um ein rollierendes Fenster zu konfigurieren, geben Sie das Fenster in Sekunden (TumblingWindowInSeconds) an. Der folgende AWS Command Line Interface (AWS CLI) - Beispielbefehl erstellt eine Streaming-Ereignisquellenzuordnung, die ein rollierendes Fenster von 120 Sekunden hat. Die für Aggregation und Verarbeitung definierte Lambda-Funktion wird genannt `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 bestimmt die rollierenden Fenstergrenzen basierend auf dem Zeitpunkt, zu dem Datensätze in den Stream eingefügt wurden. Für alle Datensätze steht ein ungefährer Zeitstempel zur Verfügung, den Lambda in Grenzbestimmungen verwendet.

Rollierende Fensteraggregationen unterstützen kein Resharding. Wenn ein Shard endet, betrachtet Lambda das aktuelle Fenster als geschlossen, und alle untergeordneten Shards beginnen ihr eigenes Fenster in einem neuen Zustand. Wenn dem aktuellen Fenster keine neuen Datensätze hinzugefügt werden, wartet Lambda bis zu 2 Minuten, bevor es annimmt, dass das Fenster zu Ende ist. Dadurch wird sichergestellt, dass die Funktion alle Datensätze im aktuellen Fenster liest, auch wenn die Datensätze in Abständen hinzugefügt werden.

Rollierende Fenster unterstützen vollständig die bestehenden Wiederholungsrichtlinien `maxRetryAttempts` und `maxRecordAge`.

Example Handler.py – Aggregation und Verarbeitung

Die folgende Python-Funktion veranschaulicht, wie Sie Ihren Endzustand aggregieren und dann verarbeiten:

```
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}
```

Lambda-Parameter für Amazon Kinesis Data Streams Zuordnungen von Ereignisquellen

Alle Lambda-Zuordnungen von Ereignisquellen verwenden dieselben API-Operationen CreateEventSourceMapping und UpdateEventSourceMapping. Allerdings gelten nur einige der Parameter für Kinesis.

Parameter	Erforderlich	Standard	Hinweise
BatchSize	N	100	Höchstwert: 10 000.
BisectBatchOnFunctionError	N	false	Keine
DestinationConfig	N	N/A	Ein Ziel der Amazon-SQS-Warteschlange oder des Amazon-SNS-Themas für verworfene Datensätze. Weitere Informationen finden Sie unter

Parameter	Erforderlich	Standard	Hinweise
			Konfigurieren von Zielen für fehlgeschlagene Aufrufe.
Aktiviert	N	true	Keine
EventSourceArn	Y	N/A	Der ARN des Datenstroms oder eines Stream-Konsumenten
FunctionName	Y	N/A	Keine
FunctionResponseTypes	N	N/A	Damit Ihre Funktion bestimmte Fehler in einem Batch meldet, beziehen Sie den Wert `ReportBatchItemFailures` in `FunctionResponseTypes` ein. Weitere Informationen finden Sie unter Konfigurieren einer teilweisen Batch-Antwort mit Kinesis Data Streams und Lambda.
MaximumBatchingWindowInSeconds	N	0	Keine

Parameter	Erforderlich	Standard	Hinweise
MaximumRecordAgeInSeconds	N	-1	-1 bedeutet unendlich: Lambda verwirft keine Datensätze (Kinesis Data Streams-Datenaufbewahrungseinstellungen gelten weiterhin) Minimum: -1 Höchstwert: 604 800
MaximumRetryAttempts	N	-1	-1 bedeutet unendlich: Fehlgeschlagene Datensätze werden wiederholt, bis der Datensatz abläuft Minimum: -1 Höchstwert: 10 000.
ParallelizationFactor	N	1	Maximum: 10
StartingPosition	Y	N/A	AT_TIMESTAMP, TRIM_HORIZON, oder LATEST
StartingPositionTimestamp	N	N/A	Nur gültig, wenn StartingPosition auf AT_TIMESTAMP gesetzt ist. Die Zeit, ab der mit dem Lesen begonnen werden soll, in Unix-Zeit sekunden

Parameter	Erforderlich	Standard	Hinweise
TumblingWindowInSeconds	N	N/A	Minimum: 0 Maximum: 900

Verwendung der Ereignisfilterung mit einer Kinesis-Ereignisquelle

Sie können die Ereignisfilterung verwenden, um zu steuern, welche Datensätze aus einem Stream oder einer Warteschlange Lambda an Ihre Funktion sendet. Allgemeine Informationen über die Funktionsweise der Ereignisfilterung finden Sie unter the section called “Ereignisfilterung”.

Dieser Abschnitt konzentriert sich auf die Ereignisfilterung für Kinesis-Ereignisquellen.

> **Note**
>
> Kinesis-Zuordnungen von Ereignisquellen unterstützen nur das Filtern nach dem `data`-Schlüssel.

Themen

- Grundlagen der Kinesis-Ereignisfilterung
- Filtern von aggregierten Kinesis-Datensätzen

Grundlagen der Kinesis-Ereignisfilterung

Angenommen, ein Producer gibt JSON-formatierte Daten in Ihren Kinesis-Datenstrom ein. Ein Beispieldatensatz würde wie folgt aussehen, wobei die JSON-Daten im `data`-Feld in eine Base64-kodierte Zeichenfolge umgewandelt wurden.

```
{
    "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"
}
```

Solange die Daten, die der Producer in den Stream einspeist, als JSON gültig sind, können Sie die Ereignisfilterung verwenden, um Datensätze anhand des `data`-Schlüssels zu filtern. Nehmen wir an, ein Produzent gibt Datensätze im folgenden JSON-Format in Ihren Kinesis-Stream ein.

```
{
    "record": 12345,
    "order": {
        "type": "buy",
        "stock": "ANYCO",
        "quantity": 1000
        }
}
```

Um nur die Datensätze zu filtern, bei denen der Auftragstyp "Kaufen" ist, würde das `FilterCriteria`-Objekt wie folgt aussehen.

```
{
    "Filters": [
        {
            "Pattern": "{ \"data\" : { \"order\" : { \"type\" : [ \"buy\" ] } } }"
        }
    ]
}
```

Zur Verdeutlichung sehen Sie hier den Wert des Filter-`Pattern` in reinem JSON.

```
{
    "data": {
        "order": {
```

```
            "type": [ "buy" ]
            }
      }
}
```

Sie können Ihren Filter mithilfe der Konsole, AWS CLI oder einer AWS SAM-Vorlage hinzufügen.

Console

Um diesen Filter mithilfe der Konsole hinzuzufügen, folgen Sie den Anweisungen unter Anhängen von Filterkriterien an eine Ereignisquellenzuordnung (Konsole) und geben Sie die folgende Zeichenfolge für die Filterkriterien ein.

```
{ "data" : { "order" : { "type" : [ "buy" ] } } }
```

AWS CLI

Um eine neue Zuordnung von Ereignisquellen mit diesen Filterkriterien über die AWS Command Line Interface (AWS CLI) zu erstellen, führen Sie den folgenden Befehl aus.

```
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\" ] } } }"}]}'
```

Führen Sie den folgenden Befehl aus, um diese Filterkriterien zu einer vorhandenen Zuordnung von Ereignisquellen hinzuzufügen.

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"data\" : { \"order\" : { \"type
\" : [ \"buy\" ] } } }"}]}'
```

AWS SAM

Um diesen Filter mit der AWS SAM hinzuzufügen, fügen Sie das folgende Snippet zur YAML-Vorlage für Ihre Ereignisquelle hinzu.

```
FilterCriteria:
```

```
Filters:
  - Pattern: '{ "data" : { "order" : { "type" : [ "buy" ] } } }'
```

Um Ereignisse aus Kinesis-Quellen ordnungsgemäß zu filtern, müssen sowohl das Datenfeld als auch die Filterkriterien für das Datenfeld ein gültiges JSON-Format aufweisen. Wenn eines der Felder kein gültiges JSON-Format hat, verwirft Lambda die Nachricht oder gibt eine Ausnahme aus. In der folgenden Tabelle ist das Verhalten zusammengefasst:

Format der eingehenden Daten	Filtermusterformat für Dateneigenschaften	Resultierende Aktion
Gültiges JSON	Gültiges JSON	Lambda filtert basierend auf Ihren Filterkriterien.
Gültiges JSON	Kein Filtermuster für Dateneigenschaften	Lambda filtert (nur für die anderen Metadateneigenschaften) basierend auf Ihren Filterkriterien.
Gültiges JSON	Kein JSON	Lambda gibt zum Zeitpunkt der Erstellung oder Aktualisierung der Ereignisquellenzuordnung eine Ausnahme aus. Das Filtermuster für Dateneigenschaften muss ein gültiges JSON-Format haben.
Kein JSON	Gültiges JSON	Lambda verwirft den Datensatz.
Kein JSON	Kein Filtermuster für Dateneigenschaften	Lambda filtert (nur für die anderen Metadateneigenschaften) basierend auf Ihren Filterkriterien.
Kein JSON	Kein JSON	Lambda gibt zum Zeitpunkt der Erstellung oder Aktualisierung der Ereignisquellenzuo

Format der eingehenden Daten	Filtermusterformat für Dateneigenschaften	Resultierende Aktion
		rdnung eine Ausnahme aus. Das Filtermuster für Dateneige nschaften muss ein gültiges JSON-Format haben.

Filtern von aggregierten Kinesis-Datensätzen

Mit Kinesis können Sie mehrere Datensätze in einem einzigen Kinesis-Datenstrom-Datensatz zusammenfassen, um Ihren Datendurchsatz zu erhöhen. Lambda kann Filterkriterien nur auf aggregierte Datensätze anwenden, wenn Sie Kinesis Enhanced Fanout verwenden. Das Filtern aggregierter Datensätze mit Standard-Kinesis wird nicht unterstützt. Bei der Verwendung von Enhanced Fan-Out konfigurieren Sie einen dedizierten Kinesis-Durchsatzverbraucher, der als Auslöser für Ihre Lambda-Funktion dient. Lambda filtert dann die aggregierten Datensätze und übergibt nur die Datensätze, die Ihren Filterkriterien entsprechen.

Weitere Informationen zur Aggregation von Kinesis-Datensätzen finden Sie im Abschnitt Aggregation auf der Seite mit den wichtigsten Konzepten der Kinesis Producer Library (KPL). Weitere Informationen zur Verwendung von Lambda mit Kinesis Enhanced Fanout finden Sie im AWS-Compute-Blog unter Erhöhung der Streamverarbeitungsleistung in Echtzeit mit Amazon Kinesis Data Streams Enhanced Fanout und AWS Lambda.

Anleitung: Verwenden von Lambda mit Kinesis Data Streams

In dieser Anleitung erstellen Sie eine Lambda-Funktion, um Ereignisse aus einem Amazon Kinesis Data Streams zu nutzen.

1. Die benutzerdefinierte Anwendung schreibt die Datensätze zum Stream.
2. AWS Lambda fragt den Stream ab und ruft, wenn es neue Datensätze im Stream erkennt, Ihre Lambda-Funktion auf.
3. AWS Lambda führt die Lambda-Funktion aus, indem es die Ausführungsrolle annimmt, die Sie bei der Erstellung der Lambda-Funktion angegeben haben.

Voraussetzungen

Installieren Sie die AWS Command Line Interface

Wenn Sie das noch nicht installiert haben AWS Command Line Interface, folgen Sie den Schritten unter Installieren oder Aktualisieren der neuesten Version von AWS CLI, um es zu installieren.

Das Tutorial erfordert zum Ausführen von Befehlen ein Befehlszeilenterminal oder eine Shell. Verwenden Sie unter Linux und macOS Ihre bevorzugte Shell und Ihren bevorzugten Paketmanager.

> **Note**
>
> In Windows werden einige Bash-CLI-Befehle, die Sie häufig mit Lambda verwenden (z. B. `zip`), von den integrierten Terminals des Betriebssystems nicht unterstützt. Um eine in Windows integrierte Version von Ubuntu und Bash zu erhalten, installieren Sie das Windows-Subsystem für Linux.

Erstellen der Ausführungsrolle

Erstellen Sie die Ausführungsrolle, die Ihrer Funktion die Berechtigung zum Zugriff auf AWS Ressourcen erteilt.

So erstellen Sie eine Ausführungsrolle

1. Öffnen Sie die Seite Roles (Rollen) in der IAM-Konsole.
2. Wählen Sie Rolle erstellen aus.
3. Erstellen Sie eine Rolle mit den folgenden Eigenschaften.
   - Trusted entity (Vertrauenswürdige Entität – AWS Lambda.
   - Berechtigungen — AWSLambdaKinesisExecutionRole.
   - Role name (Name der Rolle – **`lambda-kinesis-role`**.

Die AWSLambdaKinesisExecutionRoleRichtlinie verfügt über die Berechtigungen, die die Funktion zum Lesen von Elementen aus Kinesis und zum CloudWatch Schreiben von Protokollen in Logs benötigt.

Erstellen der Funktion

Erstellen Sie eine Lambda-Funktion, die Ihre Kinesis-Nachrichten verarbeitet. Der Funktionscode protokolliert die Ereignis-ID und die Ereignisdaten des Kinesis-Datensatzes in CloudWatch Logs.

In diesem Tutorial wird die Runtime Node.js 24 verwendet, aber wir haben auch Beispielcode in anderen Laufzeitsprachen bereitgestellt. Sie können die Registerkarte im folgenden Feld auswählen, um Code für die gewünschte Laufzeit anzusehen. Der JavaScript Code, den Sie in diesem Schritt verwenden, befindet sich im ersten Beispiel, das auf der JavaScriptRegisterkarte angezeigt wird.

.NET

SDK für .NET

> **Note**
>
> Es gibt noch mehr dazu GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines Kinesis-Ereignisses mit Lambda unter Verwendung von .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

SDK für Go V2

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines Kinesis-Ereignisses mit Lambda unter Verwendung von 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 für Java 2.x

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines Kinesis-Ereignisses mit Lambda unter Verwendung von 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 für JavaScript (v3)

> **Note**
>
> Es gibt noch mehr dazu GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Ein Kinesis-Ereignis mit Lambda unter Verwendung von. 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;
}
```

Ein Kinesis-Ereignis mit Lambda unter Verwendung von. 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

SDK für PHP

> **Note**
>
> Es gibt noch mehr dazu. GitHub Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines Kinesis-Ereignisses mit Lambda unter Verwendung von 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

SDK für Python (Boto3)

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines Kinesis-Ereignisses mit Lambda unter Verwendung von 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

SDK für Ruby

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines Kinesis-Ereignisses mit Lambda unter Verwendung von 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 für Rust

> **Note**
>
> Es gibt noch mehr dazu GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines Kinesis-Ereignisses mit Lambda unter Verwendung von 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
}
```

So erstellen Sie die Funktion

1. Erstellen Sie ein Verzeichnis für das Projekt und wechseln Sie dann zu diesem Verzeichnis.

```
mkdir kinesis-tutorial
cd kinesis-tutorial
```

2. Kopieren Sie den JavaScript Beispielcode in eine neue Datei mit dem Namen`index.js`.
3. Erstellen Sie ein Bereitstellungspaket.

```
zip function.zip index.js
```

4. Erstellen Sie eine Lambda-Funktion mit dem Befehl `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
```

Lambda-Funktion testen

Rufen Sie Ihre Lambda-Funktion manuell mit dem `invoke` AWS Lambda CLI-Befehl und einem Kinesis-Beispielereignis auf.

Lambda-Funktion testen

1. Kopieren Sie das folgende JSON in eine Datei und speichern Sie sie unter `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. Verwenden Sie den `invoke`-Befehl, um das Ereignis an die Funktion zu senden.

```
aws lambda invoke --function-name ProcessKinesisRecords \
```

```
--cli-binary-format raw-in-base64-out \
--payload file://input.txt outputfile.txt
```

Die cli-binary-format Option ist erforderlich, wenn Sie Version 2 verwenden AWS CLI . Um dies zur Standardeinstellung zu machen, führen Sie `aws configure set cli-binary-format raw-in-base64-out` aus. Weitere Informationen finden Sie unter Von AWS CLI unterstützte globale Befehlszeilenoptionen im AWS Command Line Interface -Benutzerhandbuch für Version 2.

Die Antwort wird in gespeicher `out.txt`.

Kinesis-Stream erstellen

Verwenden Sie den Befehl `create-stream` , um einen Stream zu erstellen.

```
aws kinesis create-stream --stream-name lambda-stream --shard-count 1
```

Führen Sie den folgenden `describe-stream`-Befehl aus, um den Stream-ARN zu erhalten.

```
aws kinesis describe-stream --stream-name lambda-stream
```

Die Ausgabe sollte folgendermaßen aussehen:

```
{
    "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
    }
}
```

Sie benötigen im nächsten Schritt den Stream-ARN, um die Lambda-Funktion dem Stream zuzuordnen.

Hinzufügen einer Ereignisquelle in AWS Lambda

Führen Sie den Befehl AWS CLI `add-event-source` aus.

```
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
```

Notieren Sie die Zuweisungs-ID zur späteren Verwendung. Sie erhalten eine Liste der Zuweisungen von Ereignisquellen, indem Sie den `list-event-source-mappings`-Befehl ausführen.

```
aws lambda list-event-source-mappings --function-name ProcessKinesisRecords \
--event-source arn:aws:kinesis:us-east-1:111122223333:stream/lambda-stream
```

In der Antwort können Sie überprüfen, ob der Statuswert is `enabled`. Ereignisquellen-Zuweisungen können deaktiviert werden, um Abfragen vorübergehend zu pausieren, ohne dass Datensätze verloren gehen.

Testen der Einrichtung

Zum Testen der Ereignisquellen-Zuweisung fügen Sie Ihrem Kinesis-Stream Ereignisdatensätze hinzu. Der Wert `--data` ist eine Zeichenfolge, die die CLI vor dem Senden an Kinesis mit base64 verschlüsselt. Der gleiche Befehl kann mehrmals ausgeführt werden, um dem Stream mehrere Datensätze hinzuzufügen.

```
aws kinesis put-record --stream-name lambda-stream --partition-key 1 \
--data "Hello, this is a test."
```

Lambda verwendet die Ausführungsrolle, um Datensätze aus dem Stream zu lesen. Anschließend wird Ihre Lambda-Funktion aufgerufen und Datensatzbatches werden übergeben. Die Funktion dekodiert Daten aus jedem Datensatz, protokolliert sie und sendet die Ausgabe an CloudWatch Logs. Zeigen Sie die Protokolle in der CloudWatch -Konsole an.

Bereinigen Ihrer Ressourcen

Sie können jetzt die Ressourcen, die Sie für dieses Tutorial erstellt haben, löschen, es sei denn, Sie möchten sie behalten. Durch das Löschen von AWS Ressourcen, die Sie nicht mehr verwenden, vermeiden Sie unnötige Kosten für Ihre AWS-Konto.

So löschen Sie die Ausführungsrolle

1. Öffnen Sie die Seite Roles in der IAM-Konsole.
2. Wählen Sie die von Ihnen erstellte Ausführungsrolle aus.
3. Wählen Sie Löschen aus.
4. Geben Sie den Namen der Rolle in das Texteingabefeld ein und wählen Sie Delete (Löschen) aus.

So löschen Sie die Lambda-Funktion:

1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.
2. Wählen Sie die Funktion aus, die Sie erstellt haben.
3. Wählen Sie Aktionen, Löschen aus.
4. Geben Sie **`confirm`** in das Texteingabefeld ein und wählen Sie Delete (Löschen) aus.

So löschen Sie den Kinesis-Stream

1. Melden Sie sich bei der an AWS-Managementkonsole und öffnen Sie die Kinesis-Konsole unter https://console.aws.amazon.com/kinesis.
2. Wählen Sie den von Ihnen erstellten Stream aus
3. Wählen Sie Aktionen, Löschen aus.
4. Geben Sie **`delete`** in das Texteingabefeld ein.

5. Wählen Sie Löschen aus.

Verwendung von Lambda mit Kubernetes

Sie können Lambda-Funktionen mit der Kubernetes-API mithilfe von AWS-Controllern für Kubernetes (ACK) oder Crossplane bereitstellen und verwalten.

AWS-Controller für Kubernetes (ACK)

Sie können ACK zum Bereitstellen und Verwalten von AWS-Ressourcen über die Kubernetes-API verwenden. Über ACK stellt AWS benutzerdefinierte Open-Source-Controller für AWS-Services wie Lambda, Amazon Elastic Container Registry (Amazon ECR), Amazon Simple Storage Service (Amazon S3) und Amazon SageMaker AI bereit. Jeder unterstützte AWS-Service verfügt über einen eigenen benutzerdefinierten Controller. Installieren Sie in Ihrem Kubernetes-Cluster einen Controller für jeden AWS-Service, den Sie verwenden möchten. Erstellen Sie anschließend eine benutzerdefinierte Ressourcendefinition (Custom Resource Definition, CRD), um die AWS-Ressourcen zu definieren.

Wir empfehlen die Verwendung von Helm 3.8 oder höher, um ACK-Controller zu installieren. Jeder ACK-Controller verfügt über ein eigenes Helm-Chart, das den Controller, CRDs und Kubernetes RBAC-Regeln installiert. Weitere Informationen finden Sie unter Installieren eines ACK-Controllers in der ACK-Dokumentation.

Nachdem Sie die benutzerdefinierte ACK-Ressource erstellt haben, können Sie sie wie jedes andere integrierte Kubernetes-Objekt verwenden. Beispielsweise können Sie Lambda-Funktionen mit Ihren bevorzugten Kubernetes-Toolchains, einschließlich kubectl, bereitstellen und verwalten.

Hier sind einige Beispielanwendungsfälle für die Bereitstellung von Lambda-Funktionen über ACK:

- Ihr Unternehmen verwendet rollenbasierte Zugriffssteuerung (RBAC) und IAM-Rollen für Service-Konten, um Berechtigungsgrenzen zu erstellen. Mit ACK können Sie dieses Sicherheitsmodell für Lambda wiederverwenden, ohne neue Benutzer und Richtlinien erstellen zu müssen.
- Ihr Unternehmen verfügt über einen DevOps-Prozess, um Ressourcen mithilfe von Kubernetes-Manifesten in einem Amazon Elastic Kubernetes Service (Amazon EKS)-Cluster bereitzustellen. Mit ACK können Sie ein Manifest verwenden, um Lambda-Funktionen bereitzustellen, ohne eine separate Infrastruktur in Form von Code-Vorlagen zu erstellen.

Weitere Informationen zur Verwendung von ACK finden Sie im Lambda-Tutorial in der ACK-Dokumentation.

Crossplane

Crossplane ist ein Open-Source-Projekt der Cloud Native Computing Foundation (CNCF), das Kubernetes zur Verwaltung von Cloud-Infrastrukturressourcen verwendet. Mit Crossplane können Entwickler Infrastruktur anfordern, ohne deren Komplexität verstehen zu müssen. Plattformteams behalten die Kontrolle über die Bereitstellung und Verwaltung der Infrastruktur.

Mit Crossplane können Sie Lambda-Funktionen mit Ihren bevorzugten Kubernetes-Toolchains wie kubectl und jeder CI/CD-Pipeline bereitstellen und verwalten, die Manifeste für Kubernetes bereitstellen kann. Hier sind einige Beispielanwendungsfälle für die Bereitstellung von Lambda-Funktionen über Crossplane:

- Ihr Unternehmen möchte Compliance durchsetzen, indem es sicherstellt, dass Lambda-Funktionen über die richtigen Tags verfügen. Plattformteams können Crossplane Compositions verwenden, um diese Richtlinie durch API-Abstraktionen zu definieren. Entwickler können diese Abstraktionen dann verwenden, um Lambda-Funktionen mit Tags bereitzustellen.
- Ihr Projekt verwendet GitOps mit Kubernetes. In diesem Modell gleicht Kubernetes kontinuierlich das Git-Repository (gewünschter Status) mit den im Cluster ausgeführten Ressourcen (aktueller Status) ab. Bei Unterschieden nimmt der GitOps-Prozess automatisch Änderungen am Cluster vor. Sie können GitOps mit Kubernetes zum Bereitstellen und Verwalten von Lambda-Funktionen über Crossplane verwenden und dabei bekannte Kubernetes-Tools und -Konzepte wie CRDs und Controller verwenden.

Weitere Informationen zur Verwendung von Crossplane mit Lambda finden Sie hier:

- AWS-Blueprints für Crossplane: Dieses Repository enthält Beispiele für die Verwendung von Crossplane zum Bereitstellen von AWS-Ressourcen, einschließlich Lambda-Funktionen.

  > **Note**
  >
  > AWS-Blueprints für Crossplane befinden sich in der aktiven Entwicklung und sollten nicht in der Produktion verwendet werden.

- Bereitstellung von Lambda mit Amazon EKS und Crossplane: Dieses Video demonstriert ein fortgeschrittenes Beispiel für das Bereitstellen einer Serverless AWS-Architektur mit Crossplane, wobei das Design sowohl aus der Entwickler- als auch aus der Plattformperspektive untersucht wird.

Verwenden von Lambda mit Amazon MQ

> **Note**
>
> Wenn Sie Daten an ein anderes Ziel als eine Lambda-Funktion senden oder die Daten vor dem Senden anreichern möchten, finden Sie weitere Informationen unter Amazon EventBridge Pipes.

Amazon MQ ist ein verwalteter Message-Broker-Service für Apache ActiveMQ und RabbitMQ. Mit einem Message Broker können Software-Anwendungen und -Komponenten mithilfe verschiedener Programmiersprachen, Betriebssysteme und formeller Messaging-Protokolle entweder über Themen- oder Queue-Ereigniszielen miteinander kommunizieren.

Amazon MQ kann auch Amazon Elastic Compute Cloud (Amazon EC2) -Instances in Ihrem Namen verwalten, indem es ActiveMQ- oder RabbitMQ-Broker installiert und verschiedene Netzwerktopologien und andere Infrastrukturanforderungen bereitstellt.

Verwenden Sie eine Lambda-Funktion, um Datensätze von Ihrem Amazon-MQ-Message-Broker zu verarbeiten. Lambda ruft Ihre Funktion über ein Ereignisquellen-Zuweisung auf, eine Lambda-Ressource, die Nachrichten von Ihrem Broker liest und die Funktion synchron aufruft.

> **Warning**
>
> Zuordnungen von Lambda-Ereignisquellen verarbeiten jedes Ereignis mindestens einmal und es kann zu einer doppelten Verarbeitung von Datensätzen kommen. Um mögliche Probleme im Zusammenhang mit doppelten Ereignissen zu vermeiden, empfehlen wir Ihnen dringend, Ihren Funktionscode idempotent zu machen. Weitere Informationen finden Sie im Knowledge Center unter Wie mache ich meine Lambda-Funktion idempotent?. AWS

Das Amazon-MQ-Ereignisquellen-Zuweisung hat die folgenden Konfigurationseinschränkungen:

- Parallelität – Lambda-Funktionen, die eine Amazon MQ-Zuordnung von Ereignisquellen verwenden, haben eine Standardeinstellung für maximale Parallelität. Für ActiveMQ begrenzt der Lambda-Service die Anzahl der gleichzeitigen Ausführungsumgebungen auf fünf pro Amazon MQ-Zuordnung von Ereignisquellen. Für RabbitMQ ist die Anzahl der gleichzeitigen Ausführungsumgebungen auf 1 pro Amazon MQ-Zuordnung von Ereignisquellen begrenzt.

Selbst wenn Sie die reservierten oder bereitgestellten Parallelitätseinstellungen Ihrer Funktion ändern, stellt der Lambda-Service keine weiteren Ausführungsumgebungen zur Verfügung. Um eine Erhöhung der standardmäßigen maximalen Parallelität für eine einzelne Amazon MQ MQ-Ereignisquellenzuordnung zu beantragen, wenden Sie sich an Support die UUID der Ereignisquellenzuweisung sowie an die Region. Da Erhöhungen auf der Ebene der spezifischen Zuordnung von Ereignisquellen und nicht auf der Ebene des Kontos oder der Region vorgenommen werden, müssen Sie für jede Zuordnung von Ereignisquellen manuell eine Skalierungserhöhung anfordern.

- Kontoübergreifend – Lambda unterstützt keine kontoübergreifende Verarbeitung. Sie können Lambda nicht zur Verarbeitung von Datensätzen von einem Amazon-MQ-Message Broker verwenden, der sich in einem anderen AWS-Konto befindet.
- Authentifizierung — Für ActiveMQ wird nur SimpleAuthenticationPluginActiveMQ unterstützt. Für RabbitMQ wird nur der PLAIN-Authentifizierungsmechanismus unterstützt. Benutzer müssen es verwenden, AWS Secrets Manager um ihre Anmeldeinformationen zu verwalten. Weitere Informationen zur ActiveMQ-Authentifizierung finden Sie unter Integrieren von ActiveMQ-Brokern mit LDAP im Amazon-MQ-Entwicklerhandbuch.
- Verbindungskontingent – Broker haben eine maximale Anzahl zulässiger Verbindungen pro Wire-Level-Protokoll. Dieses Kontingent basiert auf dem Instance-Typ des Brokers. Weitere Informationen finden Sie im Abschnitt Broker in Kontingente in Amazon MQ im Amazon-MQ-Entwicklerhandbuch.
- Konnektivität – Sie können Broker in einer öffentlichen oder privaten Virtual Private Cloud (VPC) erstellen. Für private Zwecke VPCs benötigt Ihre Lambda-Funktion Zugriff auf die VPC, um Nachrichten zu empfangen. Weitere Informationen finden Sie unter the section called “Konfigurieren Sie die Netzwerksicherheit”, weiter unten in diesem Abschnitt.
- Ereignisziele – Es werden nur Warteschlangenziele unterstützt. Sie können jedoch ein virtuelles Thema verwenden, das sich intern wie ein Thema verhält, während es mit Lambda als eine Warteschlange interagiert. Weitere Informationen finden Sie unter virtuelle Ziele auf der Apache-ActiveMQ-Website und virtuelle Hosts auf der RabbitMQ-Website.
- Netzwerktopologie – Für ActiveMQ wird nur ein Einzelinstance- oder Standby-Broker pro Ereignisquellen-Zuweisung unterstützt. Für RabbitMQ wird nur eine Einzelinstance-Broker- oder Cluster-Bereitstellung pro Ereignisquellen-Zuweisung unterstützt. Single-Instance-Broker benötigen einen Failover-Endpunkt. Weitere Informationen zu diesen Broker-Bereitstellungsmodi finden Sie unter Aktive MQ-Broker-Architektur und Broker-Architektur von Rabbit MQ im Entwicklerhandbuch für Amazon MQ.
- Protokolle – Die unterstützten Protokolle hängen vom Typ der Amazon-MQ-Integration ab.

- Für ActiveMQ-Integrationen verwendet Lambda Nachrichten mithilfe des OpenWire/Java Message Service (JMS) -Protokolls. Es werden keine anderen Protokolle unterstützt, um Nachrichten zu verbrauchen. Innerhalb des JMS-Protokolls werden nur `TextMessage` und `BytesMessage` unterstützt. Lambda unterstützt auch benutzerdefinierte JMS-Eigenschaften. Weitere Informationen zum OpenWire Protokoll finden Sie OpenWireauf der Apache ActiveMQ-Website.
- Bei RabbitMQ-Integrationen verbraucht Lambda Nachrichten mit dem AMQP-0-9-1-Protokoll. Es werden keine anderen Protokolle unterstützt, um Nachrichten zu verbrauchen. Weitere Informationen zur Implementierung des AMQP 0-9-1-Protokolls durch RabbitMQ finden Sie im Kompletten AMQ-0-9-1-Referenzhandbuch auf der RabbitMQ-Website.

Lambda unterstützt automatisch die neuesten Versionen von ActiveMQ und RabbitMQ, die Amazon MQ unterstützt. Die neuesten unterstützten Versionen finden Sie in den Amazon-MQ-Versionshinweisen im Amazon-MQ-Entwicklerhandbuch.

> **Note**
>
> Amazon MQ hat standardmäßig ein wöchentliches Wartungsfenster für Broker. Während dieses Zeitfensters sind Broker nicht verfügbar. Für Broker ohne Standby kann Lambda während dieses Fensters keine Nachrichten verarbeiten.

Themen

- Verstehen der Lambda-Verbrauchergruppe für Amazon MQ
- Konfiguration der Amazon MQ-Ereignisquelle für Lambda
- Parameter für die Zuordnung von Ereignisquellen
- Ereignisse aus einer Amazon MQ-Ereignisquelle filtern
- Beheben Sie Fehler Amazon MQ-Zuordnung von Ereignisquellen

Verstehen der Lambda-Verbrauchergruppe für Amazon MQ

Um mit Amazon MQ zu interagieren, erstellt Lambda eine Verbrauchergruppe, die aus Ihren Amazon-MQ-Brokern lesen kann. Die Verbrauchergruppe wird mit derselben ID wie die Ereignisquellen-Zuweisung-UUID erstellt.

Für Amazon-MQ-Ereignisquellen batcht Lambda Datensätze und sendet sie in einer einzigen Nutzlast an Ihre Funktion. Um das Verhalten zu steuern, können Sie das Batch-Fenster und die Batch-Größe konfigurieren. Lambda pullt Nachrichten, bis es die Nutzlastgröße von maximal 6 MB verarbeitet, das Batch-Fenster abläuft oder die Anzahl der Datensätze die volle Batch-Größe erreicht. Weitere Informationen finden Sie unter Batching-Verhalten.

Die Konsumentengruppe ruft die Nachrichten als ein BLOB von Bytes ab, base64-codiert sie dann in eine einzelne JSON-Nutzlast und ruft dann Ihre Funktion auf. Wenn Ihre Funktion einen Fehler für eine der Nachrichten in einem Batch zurückgibt, wiederholt Lambda den gesamten Nachrichtenbatch, bis die Verarbeitung erfolgreich ist oder die Nachrichten ablaufen.

> **Note**
>
> Während Lambda-Funktionen in der Regel ein maximales Timeout-Limit von 15 Minuten haben, unterstützen Ereignisquellenzuordnungen für Amazon MSK, selbstverwaltetes Apache Kafka, Amazon DocumentDB, Amazon MQ für ActiveMQ und RabbitMQ nur Funktionen mit einem maximalen Timeout-Limit von 14 Minuten. Diese Einschränkung stellt sicher, dass die Ereignisquellenzuordnung Funktionsfehler und Wiederholungsversuche ordnungsgemäß verarbeiten kann.

Sie können die Parallelitätsnutzung einer bestimmten Funktion mithilfe der `ConcurrentExecutions` Metrik in Amazon CloudWatch überwachen. Weitere Hinweise zur Gleichzeitigkeitsskalierung finden Sie unter the section called “Konfigurieren reservierter Gleichzeitigkeit”.

Example Amazon-MQ-Datensatzereignisse

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**
>
> Im RabbitMQ-Beispiel ist `pizzaQueue` der Name der RabbitMQ-Warteschlange und `/` der Name des virtuellen Hosts. Beim Empfang von Nachrichten listet die Ereignisquelle Nachrichten unter `pizzaQueue::/` auf.

Konfiguration der Amazon MQ-Ereignisquelle für Lambda

Themen

- Konfigurieren Sie die Netzwerksicherheit
- Die Zuordnung von Ereignisquellen erstellen

Konfigurieren Sie die Netzwerksicherheit

Um Lambda über die Zuordnung von Ereignisquellen Vollzugriff auf Amazon MQ zu gewähren, muss Ihr Broker entweder einen öffentlichen Endpunkt (öffentliche IP-Adresse) verwenden oder Sie müssen Zugriff auf die Amazon-VPC gewähren, in der Sie den Broker erstellt haben.

Wenn Sie Amazon MQ mit Lambda verwenden, erstellen Sie AWS PrivateLink VPC-Endpunkte, die Ihrer Funktion Zugriff auf die Ressourcen in Ihrer Amazon VPC bieten.

> **Note**
>
> AWS PrivateLink VPC-Endpunkte sind für Funktionen mit Ereignisquellenzuordnungen erforderlich, die den Standardmodus (auf Abruf) für Ereignisabfragen verwenden. Wenn Ihre Ereignisquellenzuordnung den Bereitstellungsmodus verwendet, müssen Sie keine AWS PrivateLink VPC-Endpunkte konfigurieren.

Erstellen Sie einen Endpunkt, der den Zugriff auf die folgenden Ressourcen ermöglicht:

- Lambda – Erstellen Sie einen Endpunkt für den Lambda-Serviceprinzipal.
- AWS STS — Erstellen Sie einen Endpunkt für den, damit ein Service Principal eine Rolle AWS STS in Ihrem Namen übernehmen kann.
- Secrets Manager – Wenn Ihr Broker Secrets Manager zum Speichern von Anmeldeinformationen verwendet, erstellen Sie einen Endpunkt für Secrets Manager.

Alternativ konfigurieren Sie ein NAT-Gateway auf jedem öffentlichen Subnetz in der Amazon VPC. Weitere Informationen finden Sie unter the section called “Internetzugriff für VPC-Funktionen”.

Wenn Sie eine Ereignisquellenzuordnung für Amazon MQ erstellen, prüft Lambda, ob Elastic Network Interfaces (ENIs) bereits für die für Ihre Amazon VPC konfigurierten Subnetze und Sicherheitsgruppen vorhanden sind. Wenn Lambda feststellt, dass sie vorhanden ENIs sind, versucht es, sie wiederzuverwenden. Andernfalls erstellt Lambda neue, ENIs um eine Verbindung zur Ereignisquelle herzustellen und Ihre Funktion aufzurufen.

> **Note**
>
> Lambda-Funktionen werden immer intern ausgeführt, die dem Lambda-Dienst VPCs gehören. Die VPC-Konfiguration Ihrer Funktion hat keinen Einfluss auf die Zuordnung von Ereignisquellen. Nur die Netzwerkkonfiguration der Ereignisquelle bestimmt, wie Lambda sich mit Ihrer Ereignisquelle verbindet.

Konfigurieren Sie die Sicherheitsgruppen für die Amazon VPC, die Ihren Broker enthält. Standardmäßig verwendet Amazon MQ die folgenden Ports: `61617` (Amazon MQ für ActiveMQ) und `5671` (Amazon MQ für RabbitMQ).

- Eingehende Regeln – Erlauben Sie den gesamten Datenverkehr auf dem Standard-Broker-Port für die Sicherheitsgruppe, die mit Ihrer Ereignisquelle verbunden ist. Alternativ können Sie eine selbstreferenzierende Regel für Sicherheitsgruppen verwenden, um den Zugriff von Instances innerhalb derselben Sicherheitsgruppe zuzulassen.
- Regeln für ausgehenden Datenverkehr — Erlauben Sie den gesamten Datenverkehr auf dem Port 443 für externe Ziele, wenn Ihre Funktion mit Diensten kommunizieren muss. AWS Alternativ können Sie auch eine selbstreferenzierende Sicherheitsgruppenregel verwenden, um den Zugriff auf den Broker einzuschränken, wenn Sie nicht mit anderen Diensten kommunizieren müssen. AWS
- Amazon VPC-Endpunkt-Eingangsregeln – Wenn Sie einen Amazon VPC-Endpunkt verwenden, muss die Sicherheitsgruppe, die mit Ihrem Amazon VPC-Endpunkt verbunden ist, eingehenden Verkehr auf Port 443 von der Broker-Sicherheitsgruppe zulassen.

Wenn Ihr Broker Authentifizierung verwendet, können Sie auch die Endpunktrichtlinie für den Secrets Manager-Endpunkt einschränken. Für den Aufruf der Secrets Manager-API verwendet Lambda Ihre Funktionsrolle und nicht den Lambda-Serviceprinzipal.

Example VPC-Endpunktrichtlinie – Secrets Manager-Endpunkt

```
{
      "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"
          }
      ]
  }
```

Wenn Sie Amazon VPC-Endpunkte verwenden, AWS leitet Ihre API-Aufrufe zum Aufrufen Ihrer Funktion über das Elastic Network Interface (ENI) des Endpunkts weiter. Der Lambda-Serviceprinzipal muss alle Rollen und Funktionen aufrufen`lambda:InvokeFunction`, die diese ENIs verwenden.

Standardmäßig verfügen Amazon VPC-Endpunkte über offene IAM-Richtlinien, die einen umfassenden Zugriff auf Ressourcen ermöglichen. Es empfiehlt sich, diese Richtlinien auf die Durchführung der erforderlichen Aktionen über diesen Endpunkt zu beschränken. Um sicherzustellen, dass Ihre Zuordnung von Ereignisquellen Ihre Lambda-Funktion aufrufen kann, muss die VPC-Endpunktrichtlinie dem Lambda-Serviceprinzipal erlauben, `sts:AssumeRole` und `lambda:InvokeFunction` aufzurufen. Wenn Sie Ihre VPC-Endpunktrichtlinien so einschränken, dass nur API-Aufrufe zugelassen werden, die aus Ihrem Unternehmen stammen, kann die Zuordnung von Ereignisquellen nicht richtig funktionieren, weshalb in diesen Richtlinien `"Resource": "*"` erforderlich ist.

Die folgenden Beispiel-VPC-Endpunktrichtlinien zeigen, wie der erforderliche Zugriff auf den Lambda-Serviceprinzipal für die AWS STS - und Lambda-Endpunkte gewährt wird.

Example VPC-Endpunktrichtlinie — AWS STS Endpunkt

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

Example VPC-Endpunktrichtlinie – Lambda-Endpunkt

```
{
      "Statement": [
          {
              "Action": "lambda:InvokeFunction",
              "Effect": "Allow",
              "Principal": {
                  "Service": [
                      "lambda.amazonaws.com"
                  ]
              },
```

```
            "Resource": "*"
        }
    ]
}
```

Die Zuordnung von Ereignisquellen erstellen

Erstellen Sie ein Ereignisquellen-Zuweisung, um Lambda anzuweisen, Datensätze aus einem Amazon-MQ-Broker an eine Lambda-Funktion zu senden. Sie können mehrere Ereignisquellen-Zuweisungen erstellen, um gleiche Daten mit mehreren Funktionen oder Elemente aus mehreren Streams mit nur einer Funktion zu verarbeiten.

Um Ihre Funktion für das Lesen von Amazon MQ zu konfigurieren, fügen Sie die erforderlichen Berechtigungen hinzu und erstellen einen MQ-Auslöser in der Lambda-Konsole.

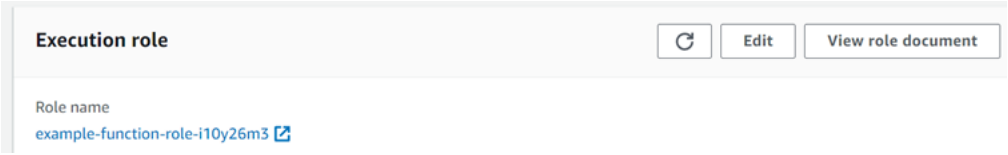
Um Datensätze von einem Amazon-MQ-Broker zu lesen, benötigt Ihre Lambda-Funktion die folgenden Berechtigungen, die ihrer Ausführungsrolle hinzugefügt werden: Sie gewähren Lambda die Erlaubnis, mit Ihrem Amazon MQ-Broker und den zugrunde liegenden Ressourcen zu interagieren, indem Sie Ihrer Funktionsausführungsrolle Berechtigungsanweisungen hinzufügen:

- mq: DescribeBroker
- Verwalter von Geheimnissen: GetSecretValue
- ec2: CreateNetworkInterface
- ec2: DeleteNetworkInterface
- ec2: DescribeNetworkInterfaces
- ec2: DescribeSecurityGroups
- ec2: DescribeSubnets
- ec2: DescribeVpcs
- Protokolle: CreateLogGroup
- Protokolle: CreateLogStream
- Protokolle: PutLogEvents

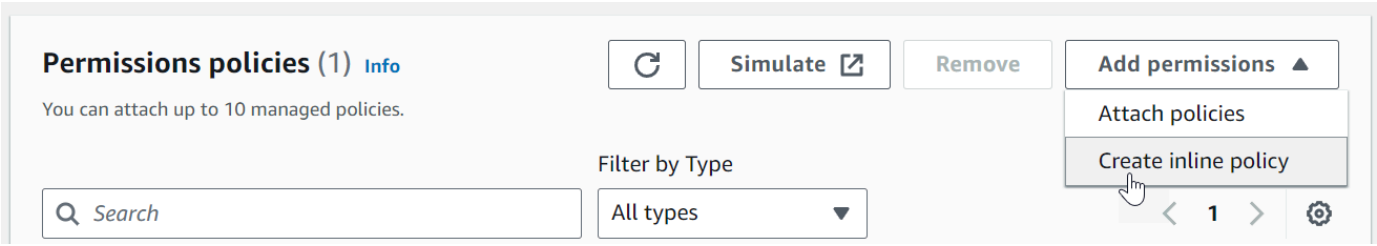
> **Note**
>
> Wenn Sie einen verschlüsselten, von Kunden verwalteten Schlüssel verwenden, fügen Sie auch die `kms:Decrypt`-Berechtigung hinzu.

So fügen Sie Berechtigungen hinzu und erstellen einen Auslöser

1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.
2. Wählen Sie den Namen einer Funktion aus.
3. Wählen Sie die Registerkarte Konfiguration und dann Berechtigungen aus.
4. Wählen Sie unter Rollenname den Link zu Ihrer Ausführungsrolle. Dieser Link öffnet die Rolle in der IAM-Konsole.



5. Wählen Sie Add permissions (Berechtigungen hinzufügen) und dann Create inline policy (Inline-Richtlinie erstellen) aus.



6. Wählen Sie im Abschnitt Richtlinien-Editor JSON aus. Geben Sie die folgende Richtlinie ein: Ihre Funktion benötigt diese Berechtigungen, um von einem Amazon MQ-Broker zu lesen.

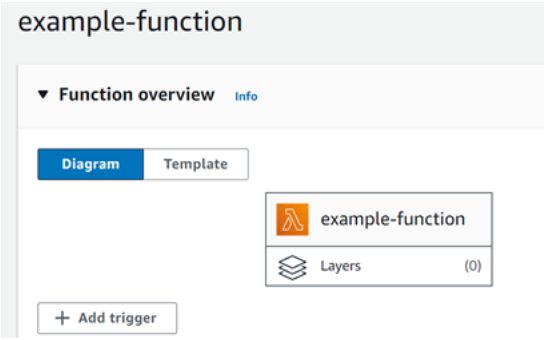
   JSON

```
{
    "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**
>
> Wenn Sie einen verschlüsselten, vom Kunden verwalteten Schlüssel verwenden, müssen Sie auch die `kms:Decrypt`-Berechtigung hinzufügen.

7. Wählen Sie Weiter aus. Geben Sie einen Richtliniennamen ein und wählen Sie anschließend Create policy (Richtlinie erstellen) aus.
8. Gehen Sie zurück zu Ihrer Funktion in der Lambda-Konsole. Wählen Sie unter Function overview (Funktionsübersicht) die Option Add trigger (Trigger hinzufügen).



9. Wählen Sie den MQ-Auslösertyp.
10. Konfigurieren Sie die erforderlichen Optionen und wählen Sie dann Add (Hinzufügen) aus.

Lambda unterstützt die folgenden Optionen für Amazon-MQ-Ereignisquellen.

- MQ-Broker – Wählen Sie einen Amazon-MQ-Broker aus.
- Batchgröße – Legen Sie die maximale Anzahl von Nachrichten fest, die in einem einzelnen Batch abgerufen werden sollen.
- Name der Warteschlange – Geben Sie die zu konsumierende Amazon-MQ-Warteschlange ein.
- Konfiguration des Zugriffs – Geben Sie die Informationen zum virtuellen Host und das Secrets Manager-Geheimnis ein, in dem Ihre Broker-Anmeldeinformationen gespeichert sind.

- Auslöser aktivieren – Deaktivieren Sie den Auslöser, um die Verarbeitung von Datensätzen anzuhalten.

Um den Auslöser zu aktivieren oder zu deaktivieren (oder zu löschen), wählen Sie den MQ-Auslöser im Designer aus. Verwenden Sie zum Neukonfigurieren des Auslösers die API-Vorgänge für die Ereignisquellen-Zuweisung.

Parameter für die Zuordnung von Ereignisquellen

Alle Lambda-Ereignisquellentypen verwenden dieselben API-Operationen CreateEventSourceMapping und UpdateEventSourceMapping. Allerdings gelten nur einige der Parameter für Amazon MQ und RabbitMQ.

Parameter	Erforderlich	Standard	Hinweise
BatchSize	N	100	Höchstwert: 10 000.
Enabled	N	true	Keine
FunctionName	Y	N/A	Keine
FilterCriteria	N	N/A	Steuern Sie, welche Ereignisse Lambda an Ihre Funktion sendet
MaximumBatchingWindowInSeconds	N	500 ms	Batching-Verhalten
Warteschlangen	N	N/A	Der Name der zu verwendenden Zielwarteschlange des Amazon-MQ-Brokers.
SourceAccessConfigurations	N	N/A	Für ActiveMQ BASIC_AUTH-Anmeldeinformationen. Für RabbitMQ können sowohl BASIC_AUT

Parameter	Erforderlich	Standard	Hinweise
			H Anmeldein formationen als auch VIRTUAL_H OST Informationen enthalten sein.

Ereignisse aus einer Amazon MQ-Ereignisquelle filtern

Sie können die Ereignisfilterung verwenden, um zu steuern, welche Datensätze aus einem Stream oder einer Warteschlange Lambda an Ihre Funktion sendet. Allgemeine Informationen über die Funktionsweise der Ereignisfilterung finden Sie unter the section called “Ereignisfilterung”.

Dieser Abschnitt konzentriert sich auf die Ereignisfilterung für Amazon MQ-Ereignisquellen.

> **Note**
>
> Amazon-MQ-Zuordnungen von Ereignisquellen unterstützen nur das Filtern nach dem `data`-Schlüssel.

Themen

- Grundlagen der Amazon MQ-Ereignisfilterung

Grundlagen der Amazon MQ-Ereignisfilterung

Angenommen, Ihre Amazon-MQ-Nachrichtenwarteschlange enthält Nachrichten entweder im gültigen JSON-Format oder als einfache Zeichenfolgen. Ein Beispieldatensatz würde wie folgt aussehen, wobei die Daten im `data`-Feld in eine Base64-kodierte Zeichenfolge umgewandelt wurden.

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=="
}
```

Sowohl für Active MQ- als auch für Rabbit MQ-Broker können Sie die Ereignisfilterung verwenden, um Datensätze anhand des `data`-Schlüssels zu filtern. Angenommen, Ihre Amazon-MQ-Warteschlange enthält Nachrichten im folgenden JSON-Format.

```
{
    "timeout": 0,
    "IPAddress": "203.0.113.254"
}
```

Um nur die Datensätze zu filtern, bei denen das `timeout`-Feld größer als 0 ist, würde das `FilterCriteria`-Objekt wie folgt aussehen.

```
{
    "Filters": [
        {
            "Pattern": "{ \"data\" : { \"timeout\" : [ { \"numeric\": [ \">\",
 0] } } ] } }"
        }
```

```
    ]
}
```

Zur Verdeutlichung sehen Sie hier den Wert des Filter-`Pattern` in reinem JSON.

```
{
    "data": {
        "timeout": [ { "numeric": [ ">", 0 ] } ]
        }
}
```

Sie können Ihren Filter mithilfe der Konsole, AWS CLI oder einer AWS SAM-Vorlage hinzufügen.

Console

Um diesen Filter mithilfe der Konsole hinzuzufügen, folgen Sie den Anweisungen unter Anhängen von Filterkriterien an eine Ereignisquellenzuordnung (Konsole) und geben Sie die folgende Zeichenfolge für die Filterkriterien ein.

```
{ "data" : { "timeout" : [ { "numeric": [ ">", 0 ] } ] } }
```

AWS CLI

Um eine neue Zuordnung von Ereignisquellen mit diesen Filterkriterien über die AWS Command Line Interface (AWS CLI) zu erstellen, führen Sie den folgenden Befehl aus.

```
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 ] } ] } }"}]}'
```

Führen Sie den folgenden Befehl aus, um diese Filterkriterien zu einer vorhandenen Zuordnung von Ereignisquellen hinzuzufügen.

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"data\" : { \"timeout\" :
 [ { \"numeric\": [ \">\", 0 ] } ] } }"}]}'
```

Führen Sie den folgenden Befehl aus, um diese Filterkriterien zu einer vorhandenen Zuordnung von Ereignisquellen hinzuzufügen.

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"data\" : { \"timeout\" :
 [ { \"numeric\": [ \">\", 0 ] } ] } }"}]}'
```

AWS SAM

Um diesen Filter mit der AWS SAM hinzuzufügen, fügen Sie das folgende Snippet zur YAML-Vorlage für Ihre Ereignisquelle hinzu.

```
FilterCriteria:
  Filters:
    - Pattern: '{ "data" : { "timeout" : [ { "numeric": [ ">", 0 ] } ] } }'
```

Mit Amazon MQ können Sie auch Datensätze filtern, bei denen die Nachricht eine einfache Zeichenfolge ist. Angenommen, Sie möchten nur Datensätze verarbeiten, bei denen die Meldung mit „Ergebnis:“ beginnt. Das `FilterCriteria`-Objekt würde wie folgt aussehen.

```
{
    "Filters": [
        {
            "Pattern": "{ \"data\" : [ { \"prefix\": \"Result: \" } ] }"
        }
    ]
}
```

Zur Verdeutlichung sehen Sie hier den Wert des Filter-`Pattern` in reinem JSON.

```
{
    "data": [
        {
        "prefix": "Result: "
        }
    ]
}
```

Sie können Ihren Filter mithilfe der Konsole, AWS CLI oder einer AWS SAM-Vorlage hinzufügen.

Console

Um diesen Filter mithilfe der Konsole hinzuzufügen, folgen Sie den Anweisungen unter Anhängen von Filterkriterien an eine Ereignisquellenzuordnung (Konsole) und geben Sie die folgende Zeichenfolge für die Filterkriterien ein.

```
{ "data" : [ { "prefix": "Result: " } ] }
```

AWS CLI

Um eine neue Zuordnung von Ereignisquellen mit diesen Filterkriterien über die AWS Command Line Interface (AWS CLI) zu erstellen, führen Sie den folgenden Befehl aus.

```
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: \" } ] }"}]}'
```

Führen Sie den folgenden Befehl aus, um diese Filterkriterien zu einer vorhandenen Zuordnung von Ereignisquellen hinzuzufügen.

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"data\" : [ { \"prefix\":
 \"Result: \" } ] }"}]}'
```

AWS SAM

Um diesen Filter mit der AWS SAM hinzuzufügen, fügen Sie das folgende Snippet zur YAML-Vorlage für Ihre Ereignisquelle hinzu.

```
FilterCriteria:
  Filters:
    - Pattern: '{ "data" : [ { "prefix": "Result " } ] }'
```

Amazon-MQ-Nachrichten müssen UTF-8-kodierte Zeichenfolgen sein (entweder einfache Zeichenfolgen oder im JSON-Format). Das liegt daran, dass Lambda Byte-Arrays von Amazon MQ

vor Anwendung der Filterkriterien in UTF-8 dekodiert. Wenn Ihre Nachrichten eine andere Kodierung nutzen, z. B. UTF-16 oder ASCII, oder das Nachrichtenformat nicht dem `FilterCriteria`-Format entspricht, verarbeitet Lambda nur Metadatenfilter. In der folgenden Tabelle ist das Verhalten zusammengefasst:

-Format der eingehenden Nachricht	Filtermusterformat für Nachrichteneigenschaften	Resultierende Aktion
Einfache Zeichenfolge	Einfache Zeichenfolge	Lambda filtert basierend auf Ihren Filterkriterien.
Einfache Zeichenfolge	Kein Filtermuster für Dateneigenschaften	Lambda filtert (nur für die anderen Metadateneigenschaften) basierend auf Ihren Filterkriterien.
Einfache Zeichenfolge	Gültiges JSON	Lambda filtert (nur für die anderen Metadateneigenschaften) basierend auf Ihren Filterkriterien.
Gültiges JSON	Einfache Zeichenfolge	Lambda filtert (nur für die anderen Metadateneigenschaften) basierend auf Ihren Filterkriterien.
Gültiges JSON	Kein Filtermuster für Dateneigenschaften	Lambda filtert (nur für die anderen Metadateneigenschaften) basierend auf Ihren Filterkriterien.
Gültiges JSON	Gültiges JSON	Lambda filtert basierend auf Ihren Filterkriterien.
Nicht UTF-8-kodierte Zeichenfolge	JSON, einfache Zeichenfolge oder kein Muster	Lambda filtert (nur für die anderen Metadateneigenschaften) basierend auf Ihren Filterkriterien.

Beheben Sie Fehler Amazon MQ-Zuordnung von Ereignisquellen

Wenn eine Lambda-Funktion auf einen nicht behebbaren Fehler stößt, stoppt Ihr Amazon-MQ-Konsument die Verarbeitung von Datensätzen. Alle anderen Konsumenten können die Verarbeitung fortsetzen, sofern sie nicht auf denselben Fehler stoßen. Um die potenzielle Ursache für einen gestoppten Konsumenten zu ermitteln, überprüfen Sie das `StateTransitionReason`-Feld in den Rücksendedetails Ihres `EventSourceMapping` auf einen der folgenden Codes:

`ESM_CONFIG_NOT_VALID`

Die Konfiguration der Ereignisquellen-Zuweisung ist ungültig.

`EVENT_SOURCE_AUTHN_ERROR`

Lambda konnte die Ereignisquelle nicht authentifizieren.

`EVENT_SOURCE_AUTHZ_ERROR`

Lambda verfügt nicht über die erforderlichen Berechtigungen für den Zugriff auf die Ereignisquelle.

`FUNCTION_CONFIG_NOT_VALID`

Die Konfiguration der Funktion ist ungültig.

Datensätze bleiben auch unbearbeitet, wenn Lambda sie aufgrund ihrer Größe fallen lässt. Die Größenbeschränkung für Lambda-Datensätze beträgt 6 MB. Um Nachrichten bei Funktionsfehlern erneut zuzustellen, können Sie eine Warteschlange für unzustellbare Nachrichten (DLQ) verwenden. Weitere Informationen finden Sie unter Erneute Zustellung von Nachrichten und DLQ-Handhabung auf der Apache-ActiveMQ-Website und Zuverlässigkeits-Leitfaden auf der RabbitMQ-Website.

> **Note**
>
> Lambda unterstützt keine benutzerdefinierten Richtlinien für die erneute Bereitstellung. Stattdessen verwendet Lambda eine Richtlinie mit den Standardwerten von der Seite Richtlinie für erneute Zustellung auf der Apache-ActiveMQ-Website, wobei `maximumRedeliveries` auf 6 gesetzt ist.

Verwendung AWS Lambda mit Amazon RDS

Sie können eine Lambda-Funktion direkt und über einen Amazon-RDS-Proxy mit einer Amazon Relational Database Service (Amazon RDS)-Datenbank verbinden. Direkte Verbindungen sind in einfachen Szenarien nützlich, und Proxys werden für die Produktion empfohlen. Ein Datenbank-Proxy verwaltet einen Pool gemeinsam genutzter Datenbankverbindungen, sodass Ihre Funktion eine hohe Gleichkeitigkeitsstufe erreichen kann, ohne dass die Datenbankverbindungen erschöpft werden.

Wir empfehlen die Verwendung von Amazon-RDS-Proxy für Lambda-Funktionen, die häufig kurze Datenbankverbindungen herstellen oder eine große Anzahl von Datenbankverbindungen öffnen und schließen. Weitere Informationen finden Sie unter Automatisches Verbinden einer Lambda-Funktion und einer DB-Instance im Amazon Relational Database Service Developer Guide.

> **Tip**
>
> Um eine Lambda-Funktion schnell mit einer Amazon-RDS-Datenbank zu verbinden, können Sie den Assistenten verwenden, der in die Konsole integriert ist. Gehen Sie wie folgt vor, um den Assistenten zu öffnen:
>
> 1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.
> 2. Wählen Sie die Funktion aus, mit der Sie eine Datenbank verbinden möchten.
> 3. Wählen Sie auf der Registerkarte Konfiguration die Option RDS-Datenbanken aus.
> 4. Wählen Sie Verbindung zur RDS-Datenbank herstellen aus.
>
> Nachdem Sie Ihre Funktion mit einer Datenbank verbunden haben, können Sie über die Option Proxy hinzufügen einen Proxy erstellen.

Konfiguration Ihrer Funktion für die Verwendung mit RDS-Ressourcen

In der Lambda-Konsole können Sie Amazon-RDS-Datenbankinstanzen und Proxy-Ressourcen bereitstellen und konfigurieren. Sie können dies tun, indem Sie auf der Registerkarte Konfiguration zu den RDS-Datenbanken navigieren. Alternativ können Sie auch Verbindungen zu Lambda-Funktionen in der Amazon-RDS-Konsole erstellen und konfigurieren. Wenn Sie eine RDS-Datenbank-Instance für die Verwendung mit Lambda konfigurieren, beachten Sie die folgenden Kriterien:

- Um eine Verbindung zu einer Datenbank herzustellen, muss sich die Funktion in derselben Amazon VPC befinden, in der Ihre Datenbank ausgeführt wird.
- Sie können Amazon-RDS-Datenbanken mit MySQL-, MariaDB-, PostgreSQL- oder Microsoft-SQL-Server-Engines verwenden.
- Sie können Aurora-DB-Cluster auch mit MySQL- oder PostgreSQL-Engines verwenden.
- Sie müssen ein Secrets-Manager-Geheimnis für die Authentifizierung der Datenbank angeben.
- Eine IAM-Rolle muss die Berechtigung zur Verwendung des Geheimnisses sowie eine Vertrauensrichtlinie bereitstellen, mit der Amazon RDS die Rolle übernehmen kann.
- Der IAM-Prinzipal, der die Konsole verwendet, um die Amazon-RDS-Ressource zu konfigurieren und sie mit Ihrer Funktion zu verbinden, muss über die folgenden Berechtigungen verfügen:

Beispiel für eine Berechtigungsrichtlinie

> **Note**
>
> Sie benötigen die Amazon-RDS-Proxy Berechtigungen nur, wenn Sie einen Amazon-RDS-Proxy konfigurieren, um einen Pool Ihrer Datenbankverbindungen zu verwalten.

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 berechnet einen Stundensatz für Proxys, der auf der Größe der Datenbank-Instance basiert. Weitere Informationen finden Sie unter RDS-Proxy-Preise. Weitere Informationen zu Proxy-Verbindungen finden Sie unter Verwendung von Amazon-RDS-Proxy im Amazon-RDS-Benutzerhandbuch.

SSL/TLS-Anforderungen für Amazon-RDS-Verbindungen

Um sichere SSL/TLS Verbindungen zu einer Amazon RDS-Datenbank-Instance herzustellen, muss Ihre Lambda-Funktion die Identität des Datenbankservers mithilfe eines vertrauenswürdigen Zertifikats verifizieren. Lambda verarbeitet diese Zertifikate je nach Art Ihres Bereitstellungspakets unterschiedlich:

- ZIP-Dateiarchive: Die Zertifikatsverarbeitung variiert je nach Laufzeit:
  - Node.js 18 und früher: Lambda schließt automatisch CA-Zertifikate und RDS-Zertifikate ein.
  - Node.js 20 und höher: Lambda lädt standardmäßig keine zusätzlichen CA-Zertifikate mehr. Legen Sie die Umgebungsvariable `NODE_EXTRA_CA_CERTS` auf `/var/runtime/ca-cert.pem` fest.

  Es kann bis zu 4 Wochen dauern, bis Amazon RDS-Zertifikate AWS-Regionen zu den verwalteten Lambda-Laufzeiten hinzugefügt werden.
- Container-Images: AWS Basis-Images enthalten nur CA-Zertifikate. Wenn Ihre Funktion eine Verbindung zu einer Amazon-RDS-Datenbank-Instance herstellt, müssen Sie die entsprechenden Zertifikate in Ihr Container-Image einbinden. Laden Sie in Ihrem Dockerfile das Zertifikatspaket herunter, das dem entspricht, AWS-Region wo Sie Ihre Datenbank hosten. Beispiel:

```
RUN curl https://truststore.pki.rds.amazonaws.com/us-east-1/us-east-1-bundle.pem -o /us-east-1-bundle.pem
```

Mit diesem Befehl wird das Amazon-RDS-Zertifikatspaket heruntergeladen und im absoluten Pfad `/us-east-1-bundle.pem` im Stammverzeichnis Ihres Containers gespeichert. Beim Konfigurieren der Datenbankverbindung in Ihrem Funktionscode müssen Sie genau diesen Pfad angeben. Beispiel:

Node.js

Die Funktion „`readFileSync`“ ist erforderlich, da Node.js-Datenbankclients den eigentlichen Zertifikatsinhalt im Speicher benötigen, nicht nur den Pfad zur Zertifikatsdatei. Ohne „`readFileSync`“ interpretiert der Client die Pfadzeichenfolge als Zertifikatsinhalt, was zu dem Fehler „Selbstsigniertes Zertifikat in der Zertifikatskette“ führt.

Example Node.js-Verbindungskonfiguration für die OCI-Funktion

```
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 Python-Verbindungskonfiguration für die OCI-Funktion

```
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

Für Java-Funktionen mit JDBC-Verbindungen muss die Verbindungszeichenfolge Folgendes enthalten:

- `useSSL=true`
- `requireSSL=true`
- Einen `sslCA`-Parameter, der auf den Speicherort des Amazon-RDS-Zertifikats im Container-Image verweist

Example Java-Verbindungszeichenfolge für die OCI-Funktion

```
// 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.NET-Verbindungszeichenfolge für MySQL-Verbindung in der OCI-Funktion

```
/// 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

Für Go-Funktionen mit MySQL-Verbindungen laden Sie das Amazon-RDS-Zertifikat in einen Zertifikatspool und registrieren es beim MySQL-Treiber. Die Verbindungszeichenfolge muss dann mithilfe des `tls`-Parameters auf diese Konfiguration verweisen.

Example Go-Code für MySQL-Verbindung in der OCI-Funktion

```
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 Ruby-Verbindungskonfiguration für die OCI-Funktion

```
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
)
```

Herstellen einer Verbindung mit einer Amazon-RDS-Datenbank in einer Lambda-Funktion

Die folgenden Codebeispiele veranschaulichen, wie eine Lambda-Funktion implementiert wird, die eine Verbindung zu einer Amazon-RDS-Datenbank herstellt. Die Funktion stellt eine einfache Datenbankanfrage und gibt das Ergebnis zurück.

> **Note**
>
> Diese Codebeispiele gelten nur für ZIP-Bereitstellungspakete. Wenn Sie Ihre Funktion mithilfe eines Container-Images bereitstellen, müssen Sie die Amazon-RDS-Zertifikatsdatei in Ihrem Funktionscode angeben, wie im vorherigen Abschnitt beschrieben.

.NET

SDK für .NET

> **Note**
>
> Es gibt noch mehr dazu. GitHub Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Herstellen einer Verbindung mit einer Amazon-RDS-Datenbank in einer Lambda-Funktion mithilfe von .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

SDK für Go V2

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Herstellen einer Verbindung zu einer Amazon-RDS-Datenbank in einer Lambda-Funktion mit 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 für Java 2.x

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Herstellen einer Verbindung zu einer Amazon-RDS-Datenbank in einer Lambda-Funktion mit 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 für JavaScript (v3)

> **Note**
>
> Es gibt noch mehr dazu GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Herstellen einer Verbindung zu einer Amazon RDS-Datenbank in einer Lambda-Funktion mithilfe von 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)
  }
};
```

Herstellen einer Verbindung zu einer Amazon RDS-Datenbank in einer Lambda-Funktion mithilfe von 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

SDK für PHP

> **Note**
>
> Es gibt noch mehr dazu. GitHub Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Herstellen einer Verbindung zu einer Amazon-RDS-Datenbank in einer Lambda-Funktion mit 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

SDK für Python (Boto3)

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Herstellen einer Verbindung zu einer Amazon-RDS-Datenbank in einer Lambda-Funktion mit 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

SDK für Ruby

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Herstellen einer Verbindung zu einer Amazon-RDS-Datenbank in einer Lambda-Funktion mit 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 für Rust

> **Note**
>
> Es gibt noch mehr dazu GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Herstellen einer Verbindung zu einer Amazon-RDS-Datenbank in einer Lambda-Funktion mit 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}")
    }))
}
```

Verarbeitung von Ereignisbenachrichtigungen von Amazon RDS

Sie können Lambda verwenden, um Ereignisbenachrichtigungen aus einer Amazon-RDS-Datenbank zu verarbeiten. Amazon RDS sendet Benachrichtigungen an ein Amazon-Simple-Notification-Service-(Amazon-SNS)-Thema, das Sie so konfigurieren können, dass eine Lambda-Funktion aufgerufen wird. Amazon SNS wickelt die Nachricht von Amazon RDS in ein eigenes Ereignisdokument und sendet sie an Ihre Funktion.

Weitere Informationen zum Konfigurieren einer Amazon-RDS-Datenbank zum Senden von Benachrichtigungen finden Sie unter Verwendung von Amazon-RDS-Ereignisbenachrichtigungen.

Example Amazon-RDS-Nachricht in einem Amazon-SNS-Ereignis

```
{
        "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"
            }
          }
        ]
      }
```

Vollständiges Lambda- und Amazon-RDS-Tutorial

- Verwendung einer Lambda-Funktion für den Zugriff auf eine Amazon-RDS-Datenbank – Im Benutzerhandbuch zu Amazon RDS wird beschrieben, wie Sie eine Lambda-Funktion verwenden, um Daten über einen Amazon-RDS-Proxy in eine Amazon-RDS-Datenbank zu schreiben. Ihre Lambda-Funktion liest Datensätze aus einer Amazon-SQS-Warteschlange und schreibt jedes Mal, wenn eine Nachricht hinzugefügt wird, neue Elemente in eine Tabelle in Ihrer Datenbank.

Wählen Sie einen Datenbankservice für Ihre Lambda-basierten Anwendungen.

Viele serverlose Anwendungen müssen Daten speichern und abrufen. AWS bietet mehrere Datenbankoptionen, die mit Lambda-Funktionen funktionieren. Zwei der beliebtesten Optionen sind Amazon DynamoDB, ein NoSQL-Datenbankservice, und Amazon RDS, eine traditionelle relationale Datenbanklösung. In den folgenden Abschnitten werden die wichtigsten Unterschiede zwischen

diesen Services bei der Verwendung mit Lambda erläutert und Sie bei der Auswahl des passenden Datenbankservices für Ihre Serverless-Anwendung unterstützt.

Weitere Informationen über die anderen von AWS angebotenen Datenbankdienste und ein allgemeines Verständnis ihrer Anwendungsfälle und Kompromisse finden Sie unter Auswahl eines AWS Datenbankdienstes. Alle AWS -Datenbankservices sind mit Lambda kompatibel, aber nicht alle eignen sich für Ihren speziellen Anwendungsfall.

Welche Möglichkeiten haben Sie bei der Auswahl eines Datenbankservices für Lambda?

AWS bietet mehrere Datenbankdienste an. Für Serverless-Anwendungen sind DynamoDB und Amazon RDS zwei der beliebtesten Optionen.

- DynamoDB ist ein vollständig verwalteter NoSQL-Datenbankservice, der für Serverless-Anwendungen optimiert ist. Er bietet nahtlose Skalierung und eine konsistente Leistung im einstelligen Millisekundenbereich in jeder Größenordnung.
- Amazon RDS ist ein verwalteter relationaler Datenbankservice, der mehrere Datenbank-Engines unterstützt, darunter MySQL und PostgreSQL. Er bietet vertraute SQL-Funktionen mit verwalteter Infrastruktur.

Empfehlungen, falls Sie Ihre Anforderungen bereits kennen

Wenn Sie sich über Ihre Anforderungen bereits im Klaren sind, finden Sie hier unsere grundlegenden Empfehlungen:

Wir empfehlen DynamoDB für Serverless-Anwendungen, die eine konsistente Leistung mit niedriger Latenz und automatische Skalierung benötigen und keine komplexen Verknüpfungen oder Transaktionen erfordern. Aufgrund seiner Serverless-Architektur eignet er sich besonders gut für Lambda-basierte Anwendungen.

Amazon RDS ist die bessere Wahl, wenn Sie komplexe SQL-Abfragen oder Verknüpfungen benötigen oder bereits Anwendungen mit relationalen Datenbanken verwenden. Beachten Sie jedoch, dass die Verbindung von Lambda-Funktionen mit Amazon RDS zusätzliche Konfigurationen erfordert und die Kaltstartzeiten beeinflussen kann.

Was Sie bei der Auswahl eines Datenbankservices beachten sollten

Bei der Auswahl zwischen DynamoDB und Amazon RDS für Ihre Lambda-Anwendungen sollten Sie die folgenden Faktoren berücksichtigen:

- Verbindungsmanagement und Kaltstarts
- Datenzugriffsmuster
- Abfragekomplexität
- Anforderungen für Datenkonsistenz.
- Skalierungsmerkmale
- Kostenmodell

Wenn Sie diese Faktoren verstehen, können Sie die Option auswählen, die Ihren spezifischen Anwendungsfällen am besten entspricht.

Verbindungsmanagement und Kaltstarts

- DynamoDB verwendet eine HTTP-API für alle Operationen. Lambda-Funktionen können Anfragen sofort stellen, ohne Verbindungen aufrechtzuerhalten, was zu einer besseren Kaltstartleistung führt. Jede Anfrage wird mithilfe von AWS Anmeldeinformationen ohne Verbindungsaufwand authentifiziert.
- Amazon RDS erfordert die Verwaltung von Verbindungspools, da es herkömmliche Datenbankverbindungen verwendet. Dies kann sich auf Kaltstarts auswirken, da neue Lambda-Instances Verbindungen herstellen müssen. Sie müssen Strategien für das Verbindungspooling implementieren und möglicherweise den Amazon-RDS-Proxy verwenden, um Verbindungen effektiv zu verwalten. Beachten Sie, dass für die Verwendung des Amazon-RDS-Proxy zusätzliche Kosten anfallen.

Datenzugriffsmuster

- DynamoDB eignet sich am besten für bekannte Zugriffsmuster und Einzeltabellendesigns. Es ist ideal für Lambda-Anwendungen, die einen konsistenten Zugriff mit geringer Latenz auf Daten basierend auf Primärschlüsseln oder Sekundärindizes benötigen.
- Amazon RDS bietet Flexibilität für komplexe Abfragen und sich ändernde Zugriffsmuster. Es ist besser geeignet, wenn Ihre Lambda-Funktionen einzigartige, maßgeschneiderte Abfragen oder komplexe Verknüpfungen über mehrere Tabellen hinweg durchführen müssen.

Abfragekomplexität

- DynamoDB zeichnet sich durch einfache, schlüsselbasierte Operationen und vordefinierte Zugriffsmuster aus. Komplexe Abfragen müssen um Indexstrukturen herum entworfen werden, und Verknüpfungen müssen im Anwendungscode behandelt werden.
- Amazon RDS unterstützt komplexe SQL-Abfragen mit Verknüpfungen, Unterabfragen und Aggregationen. Dies kann Ihren Lambda-Funktionscode vereinfachen, wenn komplexe Datenoperationen erforderlich sind.

Anforderungen für Datenkonsistenz.

- DynamoDB bietet sowohl Optionen für letztendliche als auch für starke Konsistenz, wobei starke Konsistenz für Lesevorgänge einzelner Elemente verfügbar ist. Transaktionen werden unterstützt, jedoch mit einigen Einschränkungen.
- Amazon RDS bietet vollständige ACID-Konformität (Atomizität, Konsistenz, Isolierung und Zuverlässigkeit) sowie Unterstützung für komplexe Transaktionen. Wenn Ihre Lambda-Funktionen komplexe Transaktionen oder starke Konsistenz über mehrere Datensätze hinweg benötigen, ist Amazon RDS möglicherweise besser geeignet.

Skalierungsmerkmale

- DynamoDB skaliert automatisch mit Ihrer Workload. Es kann plötzliche Spitzen im Datenverkehr durch Lambda-Funktionen ohne vorherige Bereitstellung bewältigen. Sie können den On-Demand-Kapazitätsmodus verwenden, um nur für das zu bezahlen, was Sie tatsächlich nutzen. Dies entspricht perfekt dem Skalierungsmodell von Lambda.
- Amazon RDS verfügt über eine feste Kapazität, die auf der von Ihnen ausgewählten Instance-Größe basiert. Wenn mehrere Lambda-Funktionen versuchen, gleichzeitig eine Verbindung herzustellen, überschreiten Sie möglicherweise Ihr Verbindungskontingent. Sie müssen die Verbindungspools sorgfältig verwalten und möglicherweise eine Wiederholungslogik implementieren.

Kostenmodell

- Die Preisgestaltung von DynamoDB ist gut auf Serverless-Anwendungen abgestimmt. Mit On-Demand-Kapazität zahlen Sie nur für die tatsächlichen Lese- und Schreibvorgänge Ihrer Lambda-Funktionen. Für Leerlaufzeiten fallen keine Gebühren an.

- Amazon RDS berechnet Gebühren für die laufende Instance unabhängig von der Nutzung. Dies kann bei sporadischen Workloads, wie sie in Serverless-Anwendungen typisch sind, weniger kosteneffizient sein. Für Workloads mit hohem Durchsatz und konstanter Nutzung kann es jedoch wirtschaftlicher sein.

Erste Schritte mit dem von Ihnen ausgewählten Datenbankservice

Nachdem Sie die Auswahlkriterien für DynamoDB und Amazon RDS sowie die wichtigsten Unterschiede kennengelernt haben, können Sie die Option wählen, die Ihren Bedürfnissen am besten entspricht, und mithilfe der folgenden Ressourcen loslegen.

DynamoDB

Nutzen Sie die folgenden Ressourcen, um mit DynamoDB zu beginnen

- Eine Einführung in den DynamoDB-Service finden Sie im Entwicklerhandbuch für Amazon DynamoDB unter Was ist DynamoDB?.
- Im Tutorial Verwenden von Lambda mit API Gateway sehen Sie ein Beispiel für die Verwendung einer Lambda-Funktion zum Ausführen von CRUD-Vorgängen in einer DynamoDB-Tabelle als Antwort auf eine API-Anfrage.
- Lesen Sie Programming with DynamoDB und die AWS SDKs im Amazon DynamoDB Developer Guide, um mehr darüber zu erfahren, wie Sie von Ihrer Lambda-Funktion aus auf DynamoDB zugreifen können, indem Sie eine der AWS SDKs

Amazon RDS

Nutzen Sie die folgenden Ressourcen, um mit Amazon RDS zu beginnen

- Eine Einführung in den Amazon-RDS-Service finden Sie im Abschnitt Was ist Amazon Relational Database Service (Amazon RDS)? im Benutzerhandbuch zu Amazon Relational Database Service.
- Folgen Sie dem Tutorial Verwenden einer Lambda-Funktion für den Zugriff auf eine Amazon-RDS-Datenbank im Benutzerhandbuch zu Amazon Relational Database Service.
- Weitere Informationen zur Verwendung von Lambda mit Amazon RDS finden Sie unter the section called “RDS”.

Verarbeitung von Amazon-S3-Ereignisbenachrichtigungen mit Lambda

Sie können Lambda verwenden, um Ereignisbenachrichtigungen von Amazon Simple Storage Service zu verarbeiten. Amazon S3 kann ein Ereignis an eine Lambda-Funktion senden, wenn ein Objekt erstellt oder gelöscht wird. Sie konfigurieren Benachrichtigungseinstellungen für einen Bucket und erteilen die Amazon-S3-Berechtigung zum Aufrufen einer Funktion in der ressourcenbasierten Berechtigungsrichtlinie der Funktion.

> **Warning**
>
> Wenn Ihre Lambda-Funktion den gleichen Bucket verwendet, der sie auslöst, könnte dies dazu führen, dass die Funktion in einer Schleife ausgeführt wird. Wenn der Bucket z. B. eine Funktion immer dann auslöst, wenn ein Objekt hochgeladen wird, und die Funktion ein Objekt in den Bucket hochlädt, löst die Funktion sich indirekt selbst aus. Um dies zu vermeiden, verwenden Sie zwei Buckets oder konfigurieren Sie den Auslöser so, dass er nur für einen Präfix gilt, der für eingehende Objekte verwendet wird.

Amazon S3 ruft Ihre Funktion asynchron mit einem Ereignis auf, das Details über das Objekt enthält. Das folgende Beispiel zeigt ein Ereignis, das Amazon S3 gesendet hat, als ein Bereitstellungspaket in Amazon S3 hochgeladen wurde.

Example Amazon-S3-Benachrichtigungsereignis

```
{
  "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"
        }
      }
    }
  ]
}
```

Um Ihre Funktion aufzurufen, benötigt Amazon S3 die Berechtigung von der ressourcenbasierten Richtlinie der Funktion. Wenn Sie einen Amazon-S3-Auslöser in der Lambda-Konsole konfigurieren, ändert die Konsole die ressourcenbasierte Richtlinie so, dass Amazon S3 die Funktion aufrufen kann, wenn der Bucket-Name und die Konto-ID übereinstimmen. Wenn Sie die Benachrichtigung in Amazon S3 konfigurieren, verwenden Sie die Lambda-API, um die Richtlinie zu aktualisieren. Sie können auch die Lambda-API verwenden, um einem anderen Konto Berechtigungen zu erteilen oder die Berechtigung auf einen bestimmten Alias zu beschränken.

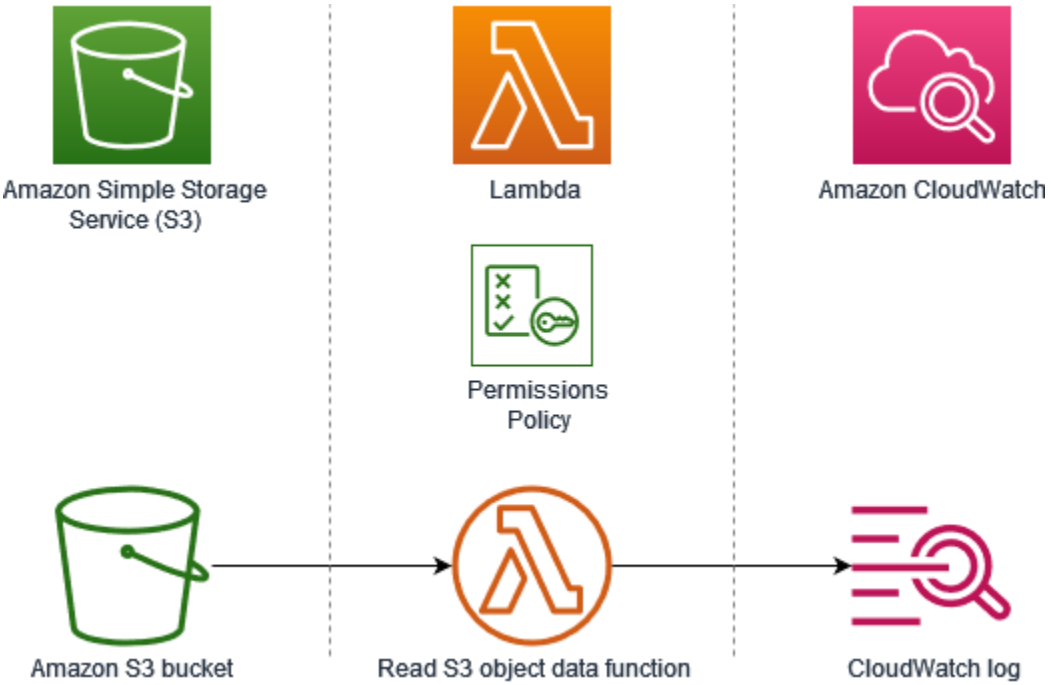
Wenn Ihre Funktion das AWS SDK zur Verwaltung von Amazon S3 S3-Ressourcen verwendet, benötigt sie in ihrer Ausführungsrolle auch Amazon S3 S3-Berechtigungen.

Themen

- Tutorial: Verwenden eines Amazon-S3-Auslösers zum Aufrufen einer Lambda-Funktion
- Tutorial: Verwenden eines Amazon-S3-Auslösers zum Erstellen von Miniaturbildern

Tutorial: Verwenden eines Amazon-S3-Auslösers zum Aufrufen einer Lambda-Funktion

In diesem Tutorial verwenden Sie die Konsole, um eine Lambda-Funktion zu erstellen und einen Auslöser für einen Amazon-Simple-Storage-Service-Bucket (Amazon-S3-Bucket) zu konfigurieren. Jedes Mal, wenn Sie Ihrem Amazon S3 S3-Bucket ein Objekt hinzufügen, wird Ihre Funktion ausgeführt und der Objekttyp wird an Amazon CloudWatch Logs ausgegeben.

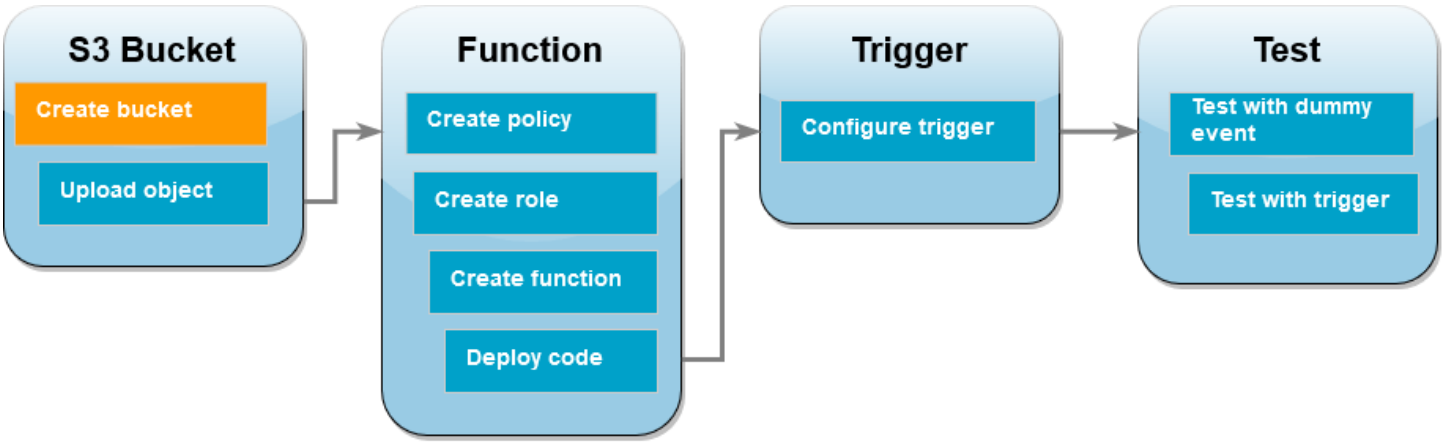


Dieses Tutorial zeigt, wie man:

1. Erstellen Sie einen Amazon-S3-Bucket.
2. Erstellen Sie eine Lambda-Funktion, die den Objekttyp von Objekten in einem Amazon-S3-Bucket ausgibt.
3. Konfigurieren Sie einen Lambda-Auslöser, der Ihre Funktion aufruft, wenn Objekte in Ihren Bucket hochgeladen werden.
4. Testen Sie Ihre Funktion, zuerst mit einem Dummy-Ereignis und dann mit dem Auslöser.

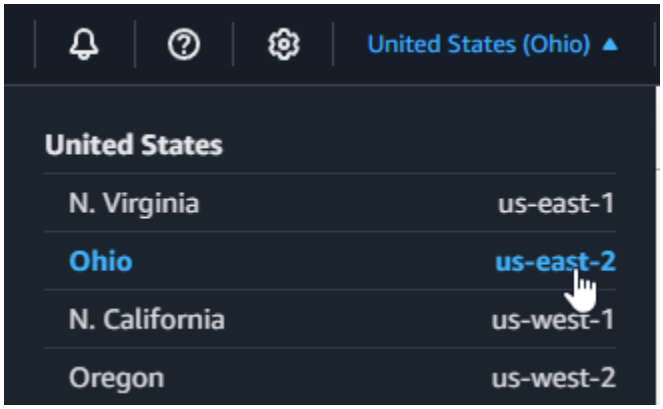
Durch das Ausführen dieser Schritte erfahren Sie, wie Sie eine Lambda-Funktion so konfigurieren, dass sie ausgeführt wird, wenn Objekte einem Amazon-S3-Bucket hinzugefügt oder daraus gelöscht werden. Sie können dieses Tutorial abschließen, indem Sie nur die AWS-Managementkonsole verwenden.

Erstellen Sie einen Amazon S3 S3-Bucket



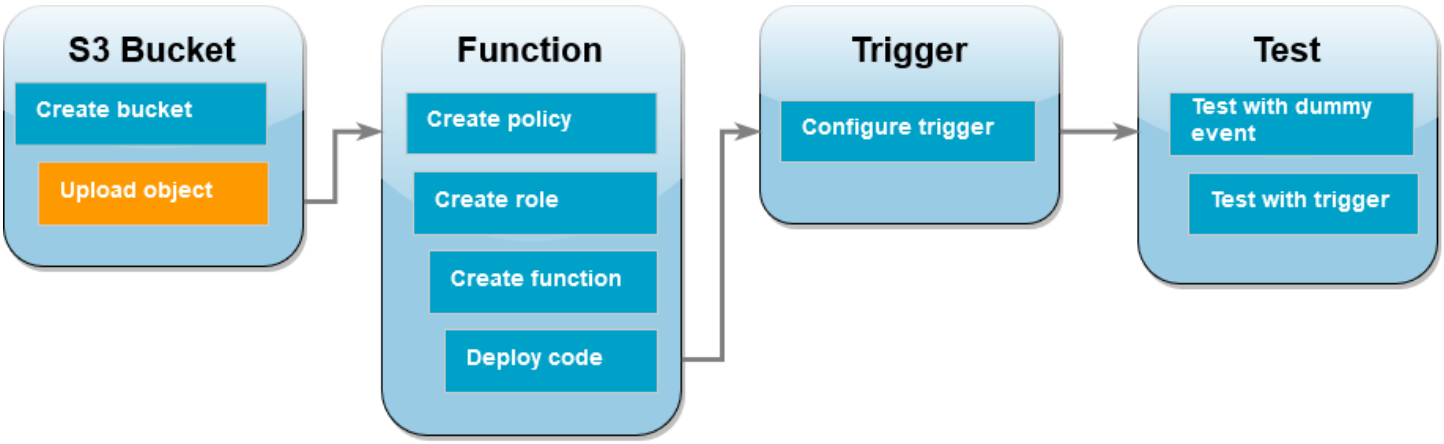
So erstellen Sie einen Amazon-S3-Bucket

1. Öffnen Sie die Amazon-S3-Konsole und wählen Sie die Seite Allzweck-Buckets aus.
2. Wählen Sie den, der Ihrem geografischen Standort AWS-Region am nächsten liegt. Sie können Ihre Region mithilfe der Dropdown-Liste am oberen Bildschirmrand ändern. Später im Tutorial müssen Sie eine Lambda-Funktion in derselben Region erstellen.



3. Wählen Sie Create Bucket (Bucket erstellen) aus.
4. Führen Sie unter Allgemeine Konfiguration die folgenden Schritte aus:
   a. Stellen Sie sicher, dass für Bucket-Typ die Option Allzweck ausgewählt ist.
   b. Geben Sie für den Bucket-Namen einen global eindeutigen Namen ein, der den Regeln für die Bucket-Benennung von Amazon S3 entspricht. Bucket-Namen dürfen nur aus Kleinbuchstaben, Zahlen, Punkten (.) und Bindestrichen (-) bestehen.
5. Belassen Sie alle anderen Optionen auf ihren Standardwerten und wählen Sie Bucket erstellen aus.

Hochladen eines Testobjekts in Ihren Bucket

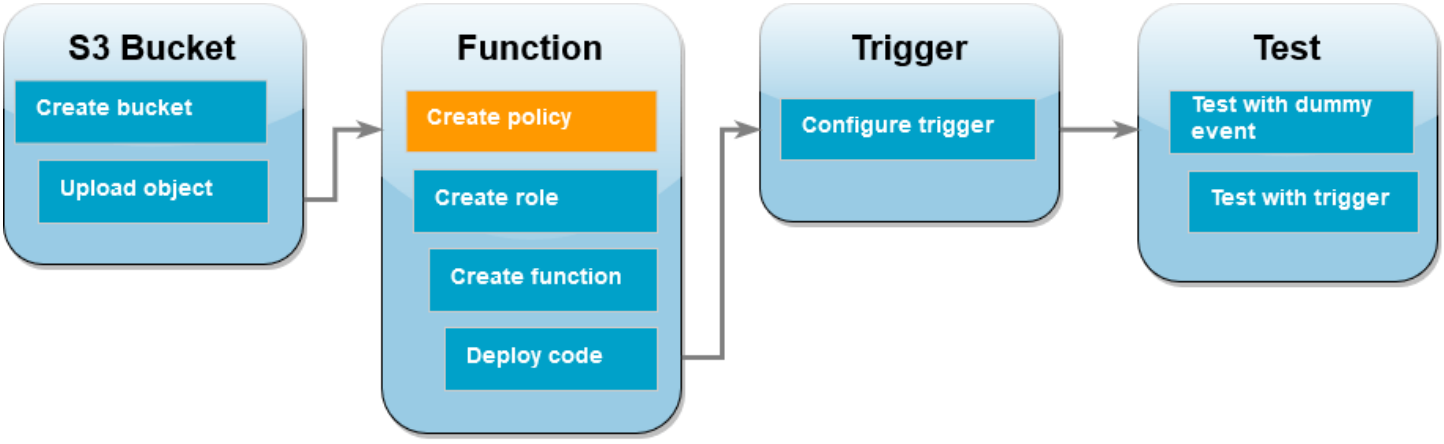


Hochladen eines Testobjekts

1. Öffnen Sie die Buckets-Seite der Amazon-S3-Konsole und wählen Sie den Bucket aus, den Sie im vorherigen Schritt erstellt haben.
2. Klicken Sie auf Hochladen.
3. Wählen Sie Dateien hinzufügen aus und wählen Sie das Objekt, das Sie hochladen möchten. Sie können eine beliebige Datei auswählen (z. B. `HappyFace.jpg`).
4. Wählen Sie Öffnen und anschließend Hochladen aus.

Im weiteren Verlauf des Tutorials werden Sie Ihre Lambda-Funktion mit diesem Objekt testen.

Erstellen einer Berechtigungsrichtlinie



Erstellen Sie eine Berechtigungsrichtlinie, die es Lambda ermöglicht, Objekte aus einem Amazon S3 S3-Bucket abzurufen und in Amazon CloudWatch Logs zu schreiben.

So erstellen Sie die Richtlinie

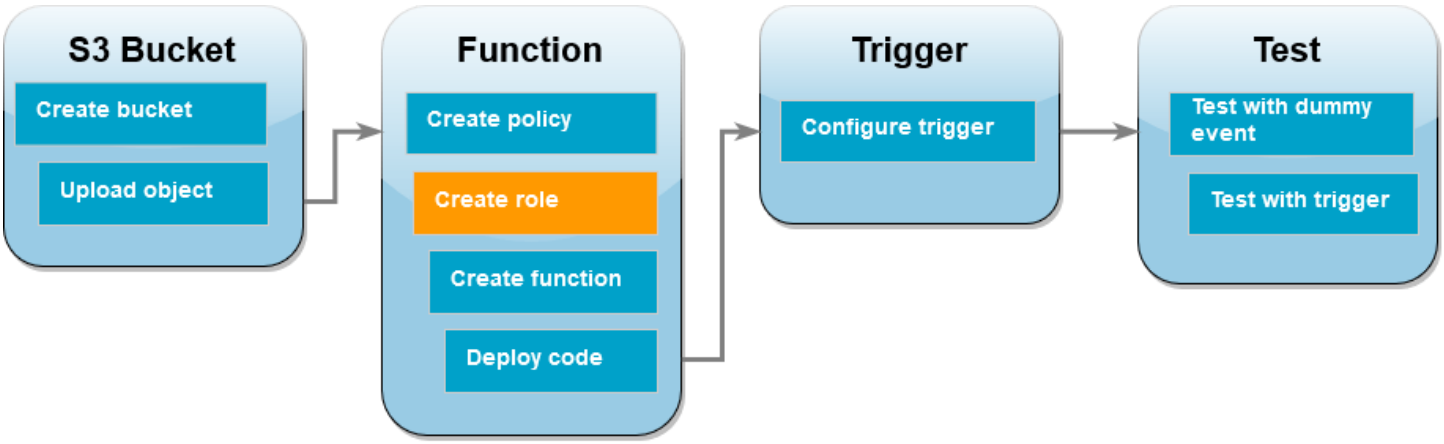
1. Öffnen Sie die Seite Richtlinien in der IAM-Konsole.
2. Wählen Sie Richtlinie erstellen aus.
3. Wählen Sie die Registerkarte JSON aus und kopieren Sie dann die folgende benutzerdefinierte JSON-Richtlinie in den JSON-Editor.

   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. Wählen Sie Next: Tags (Weiter: Tags) aus.
5. Klicken Sie auf Weiter: Prüfen.
6. Geben Sie unter Review policy (Richtlinie prüfen) für den Richtlinien-Namen **s3-trigger-tutorial** ein.
7. Wählen Sie Richtlinie erstellen aus.

Erstellen einer Ausführungsrolle

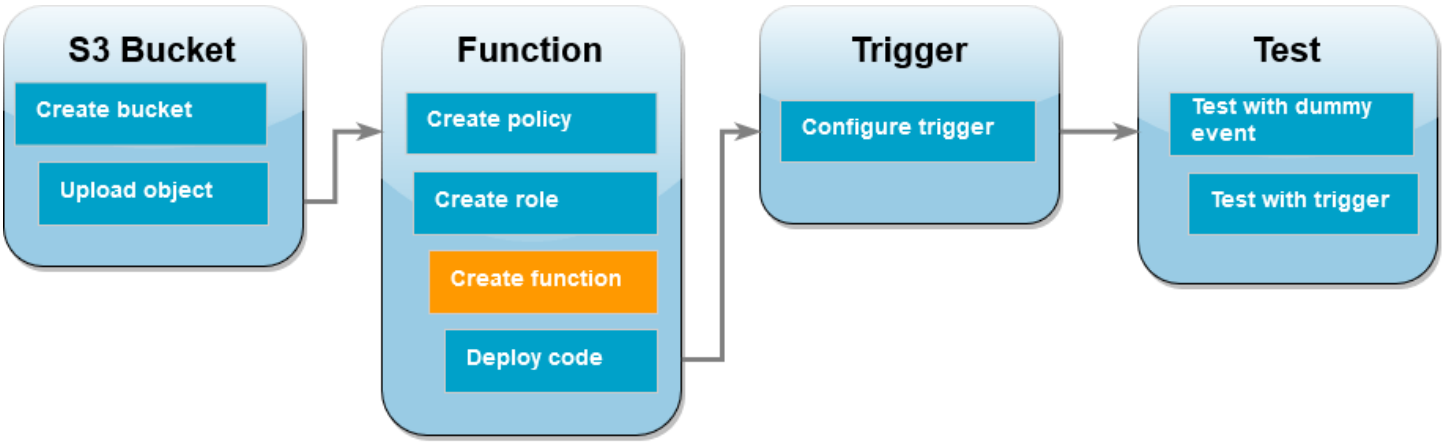


Eine Ausführungsrolle ist eine AWS Identity and Access Management (IAM-) Rolle, die einer Lambda-Funktion Zugriff AWS-Services und Ressourcen gewährt. In diesem Schritt erstellen Sie eine Ausführungsrolle unter Verwendung der Berechtigungsrichtlinie, die Sie im vorherigen Schritt erstellt haben.

So erstellen Sie eine Ausführungsrolle und fügen Ihre benutzerdefinierte Berechtigungsrichtlinie hinzu

1. Öffnen Sie die Seite Roles (Rollen) in der IAM-Konsole.
2. Wählen Sie Rolle erstellen aus.
3. Wählen Sie als Typ der vertrauenswürdigen Entität AWS -Service und dann als Anwendungsfall Lambda aus.
4. Wählen Sie Weiter aus.
5. Geben Sie im Feld für die Richtliniensuche **`s3-trigger-tutorial`** ein.
6. Wählen Sie in den Suchergebnissen die von Ihnen erstellte Richtlinie (`s3-trigger-tutorial`) und dann die Option Next (Weiter) aus.
7. Geben Sie unter Role details (Rollendetails) für den Role name (Rollennamen) **`lambda-s3-trigger-role`** ein und wählen Sie dann Create role (Rolle erstellen) aus.

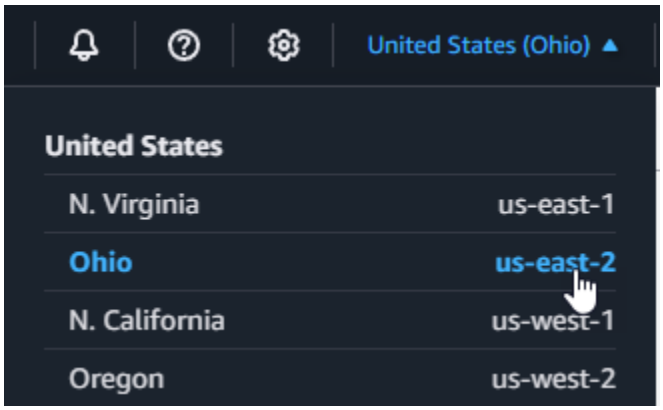
So erstellen Sie die Lambda-Funktion:



Erstellen Sie mit der Python 3.14-Laufzeit eine Lambda-Funktion in der Konsole.

So erstellen Sie die Lambda-Funktion:

1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.
2. Stellen Sie sicher, dass Sie in demselben Modus arbeiten, in dem AWS-Region Sie Ihren Amazon S3 S3-Bucket erstellt haben. Sie können Ihre Region mithilfe der Dropdown-Liste oben auf dem Bildschirm ändern.



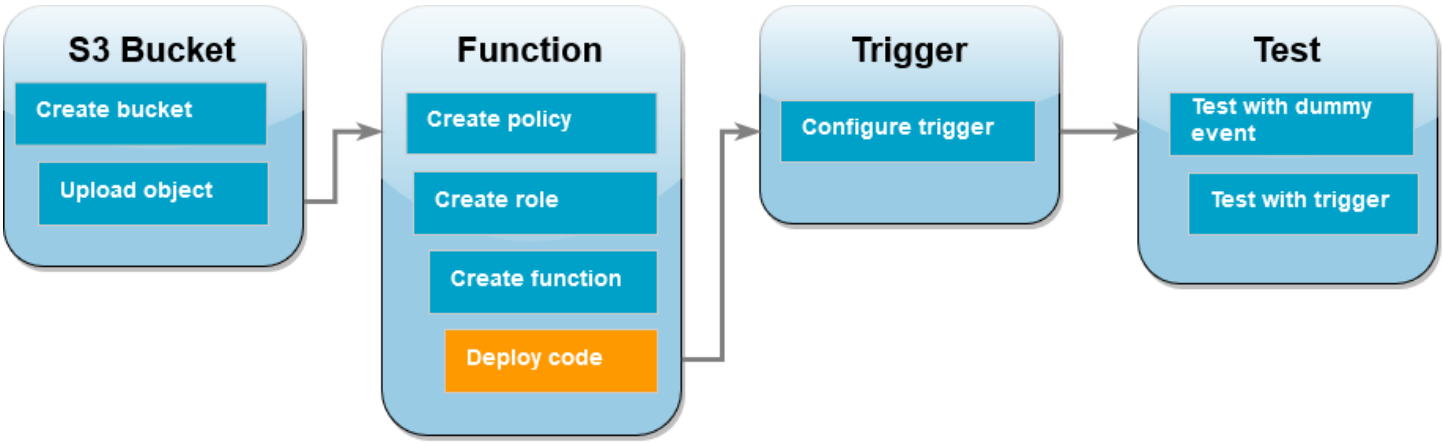
3. Wählen Sie Funktion erstellen.
4. Wählen Sie Ohne Vorgabe erstellen aus.
5. Führen Sie unter Basic information (Grundlegende Informationen) die folgenden Schritte aus:
   a. Geben Sie unter Funktionsname `s3-trigger-tutorial` ein.
   b. Wählen Sie für Runtime Python 3.14.
   c. Wählen Sie für Architektur x86_64 aus.
6. Gehen Sie auf der Registerkarte Standard-Ausführungsrolle ändern wie folgt vor:

a. Erweitern Sie die Registerkarte und wählen Sie dann Verwenden einer vorhandenen Rolle aus.

b. Wählen Sie die zuvor erstellte `lambda-s3-trigger-role` aus.

7. Wählen Sie Funktion erstellen.

Bereitstellen des Funktionscodes



Dieses Tutorial verwendet die Python 3.14-Laufzeit, aber wir haben auch Beispielcodedateien für andere Laufzeiten bereitgestellt. Sie können die Registerkarte im folgenden Feld auswählen, um den Code für die gewünschte Laufzeit anzusehen.

Die Lambda-Funktion ruft den Schlüsselnamen des hochgeladenen Objekts und den Namen des Buckets aus dem `event`-Parameter ab, den sie von Amazon S3 erhält. Die Funktion verwendet dann die Methode get_object von, AWS SDK für Python (Boto3) um die Metadaten des Objekts abzurufen, einschließlich des Inhaltstyps (MIME-Typ) des hochgeladenen Objekts.

So stellen Sie den Funktionscode bereit

1. Wählen Sie im folgenden Feld die Registerkarte Python und kopieren Sie den Code.

.NET

SDK für .NET

> **Note**
>
> Es gibt noch mehr dazu. GitHub Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines S3-Ereignisses mit Lambda unter Verwendung von .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

SDK für Go V2

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines S3-Ereignisses mit Lambda unter Verwendung von 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 für Java 2.x

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines S3-Ereignisses mit Lambda unter Verwendung von Java

```
// 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 für JavaScript (v3)

> **Note**
>
> Es gibt noch mehr dazu GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Konsumieren eines S3-Ereignisses mit Lambda unter Verwendung 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);
    }
};
```

Konsumieren eines S3-Ereignisses mit Lambda unter Verwendung 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

SDK für PHP

> **Note**
>
> Es gibt noch mehr dazu. GitHub Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines S3-Ereignisses mit Lambda unter Verwendung von 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

SDK für Python (Boto3)

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines S3-Ereignisses mit Lambda unter Verwendung von 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

SDK für Ruby

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines S3-Ereignisses mit Lambda unter Verwendung von 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 für Rust

> **Note**
>
> Es gibt noch mehr dazu GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

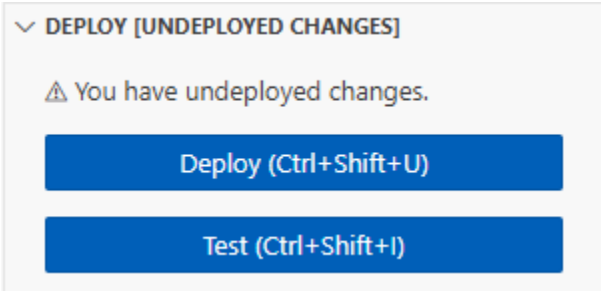
Nutzen eines S3-Ereignisses mit Lambda unter Verwendung von Rust

```
// 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> {
```

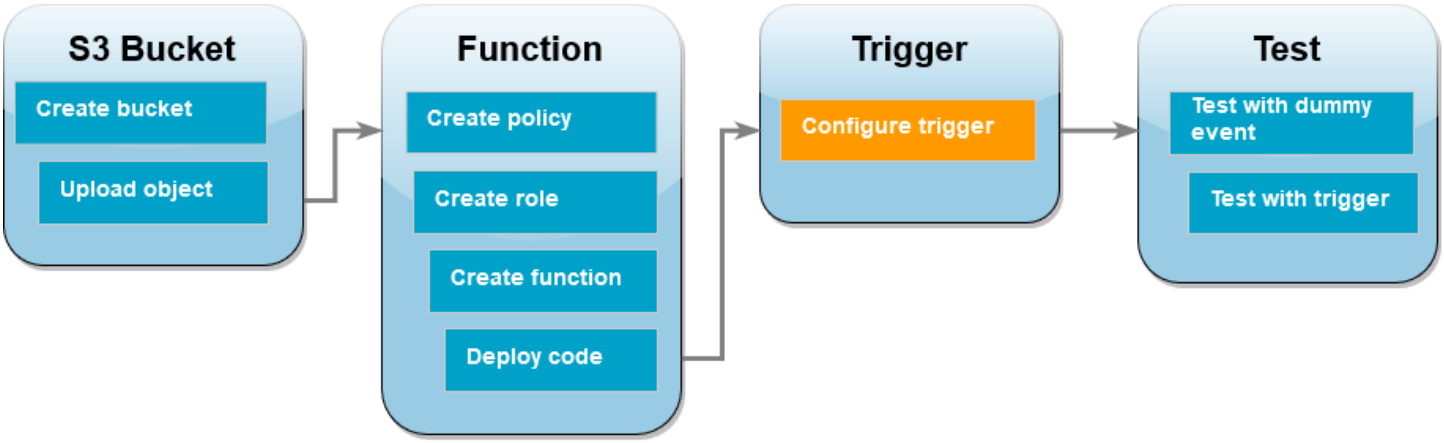
```
    tracing_subscriber::fmt()
        .with_max_level(tracing::Level::INFO)
        .with_target(false)
        .without_time()
        .init();

    // 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. Fügen Sie den Code im Code-Quellbereich der Lambda-Konsole in den Code-Editor ein und ersetzen Sie dabei den von Lambda erstellten Code.
3. Wählen Sie im Abschnitt DEPLOY die Option Deploy aus, um den Code Ihrer Funktion zu aktualisieren:

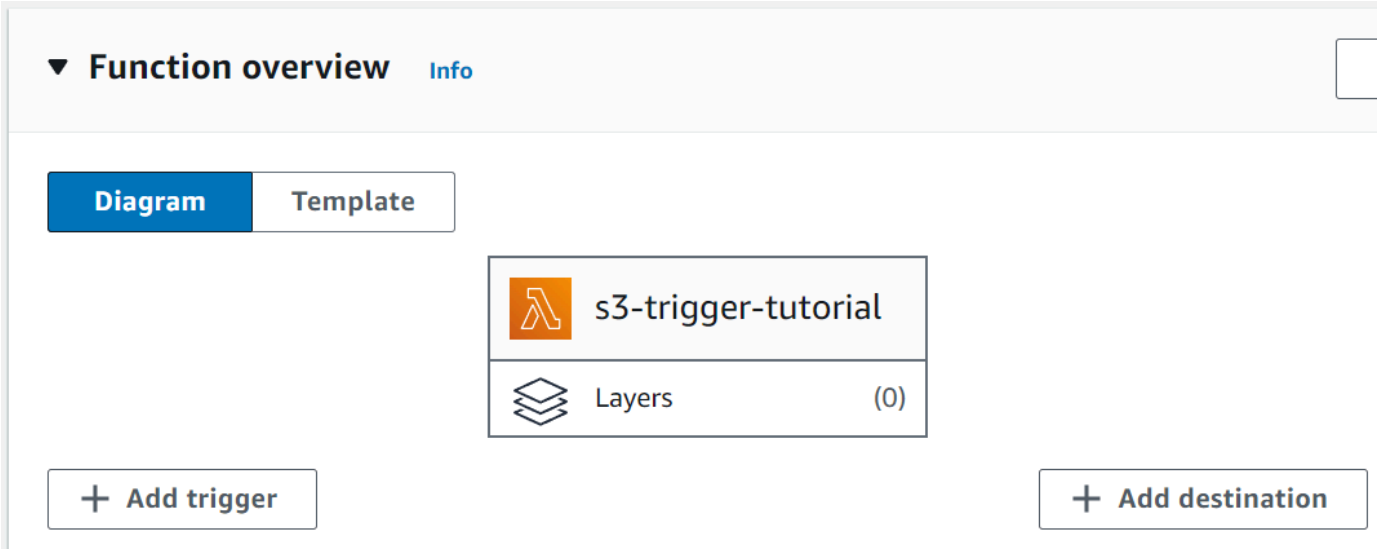


Erstellen des Amazon-S3-Auslösers



Erstellen des Amazon-S3-Auslösers

1. Wählen Sie im Bereich Function overview (Funktionsübersicht) die Option Add trigger (Auslöser hinzufügen).



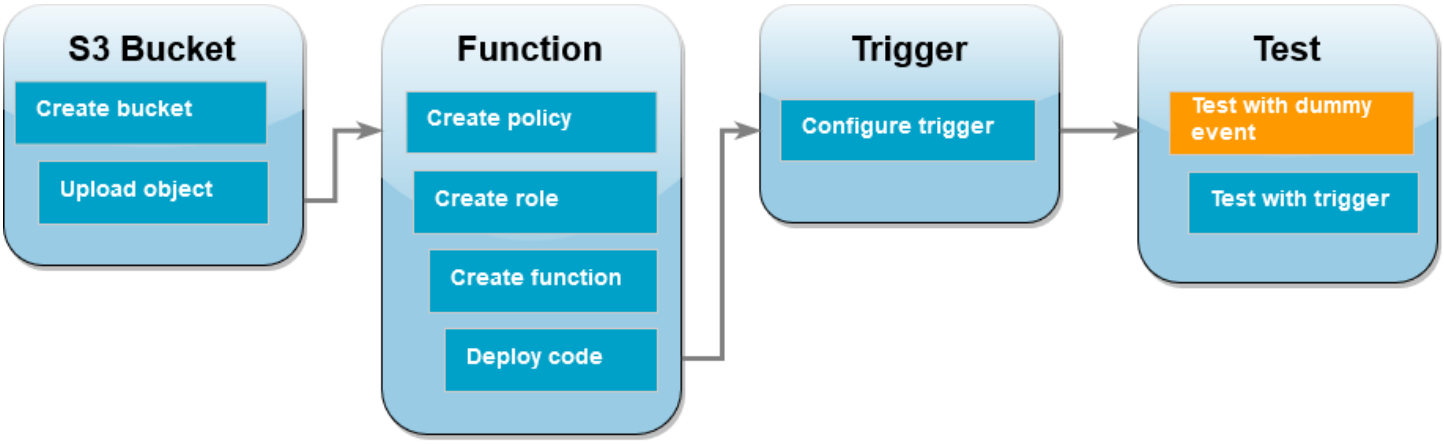
2. Wählen Sie S3 aus.
3. Wählen Sie unter Bucket den Bucket aus, den Sie zuvor im Tutorial erstellt haben.
4. Vergewissern Sie sich, dass unter Ereignistypen die Option Alle Ereignisse zur Objekterstellung ausgewählt ist.
5. Aktivieren Sie unter Rekursiver Aufruf das Kontrollkästchen, um zu bestätigen, dass die Verwendung desselben Amazon-S3-Buckets für die Ein- und Ausgabe nicht empfohlen wird.
6. Wählen Sie Hinzufügen aus.

> **Note**
>
> Wenn Sie einen Amazon-S3-Trigger für eine Lambda-Funktion mithilfe der Lambda-Konsole erstellen, konfiguriert Amazon S3 eine Ereignisbenachrichtigung für den von Ihnen angegebenen Bucket. Bevor diese Ereignisbenachrichtigung konfiguriert wird, führt Amazon S3 eine Reihe von Prüfungen durch, um zu bestätigen, dass das Ereignisziel existiert und die erforderlichen IAM-Richtlinien hat. Amazon S3 führt diese Tests auch bei allen anderen für diesen Bucket konfigurierten Ereignisbenachrichtigungen durch.
>
> Wenn der Bucket zuvor Ereignisziele für Ressourcen konfiguriert hat, die nicht mehr existieren oder für Ressourcen, die nicht über die erforderlichen Berechtigungsrichtlinien verfügen, kann Amazon S3 aufgrund dieser Prüfung keine neue Ereignisbenachrichtigung erstellen. Es wird die folgende Fehlermeldung angezeigt, die besagt, dass Ihr Auslöser nicht erstellt werden konnte:

```
An error occurred when creating the trigger: Unable to validate the following
 destination configurations.
```

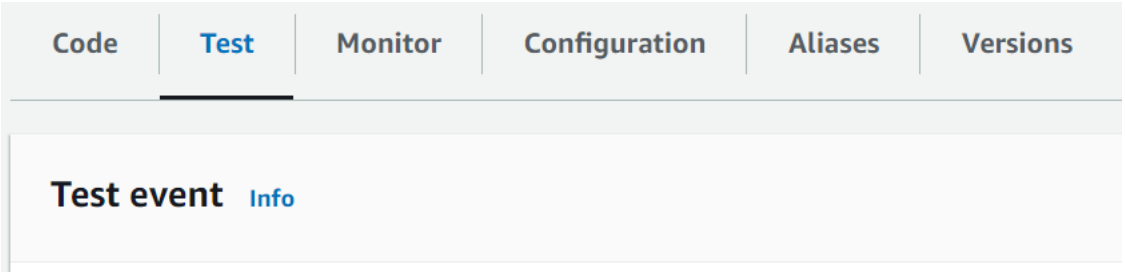
Dieser Fehler kann auftreten, wenn Sie zuvor einen Auslöser für eine andere Lambda-Funktion konfiguriert haben, die denselben Bucket verwendet und Sie die Funktion inzwischen gelöscht oder ihre Berechtigungsrichtlinien geändert haben.

Testen Ihrer Lambda-Funktion mit einem Dummy-Ereignis



So testen Sie die Lambda-Funktion mit einem Dummy-Ereignis

1. Wählen Sie auf der Konsolenseite für Ihre Funktion die Registerkarte Test aus.



2. Geben Sie für Event name (Ereignisname) `MyTestEvent` ein.
3. Fügen Sie im Event-JSON das folgende Testereignis ein. Achten Sie darauf, diese Werte zu ersetzen:

   - Ersetzen Sie `us-east-1` durch die Region, in der Sie den Amazon-S3-Bucket erstellt haben.
   - Ersetzen Sie beide Vorkommen von `amzn-s3-demo-bucket` durch den Namen Ihres eigenen Amazon-S3-Buckets.

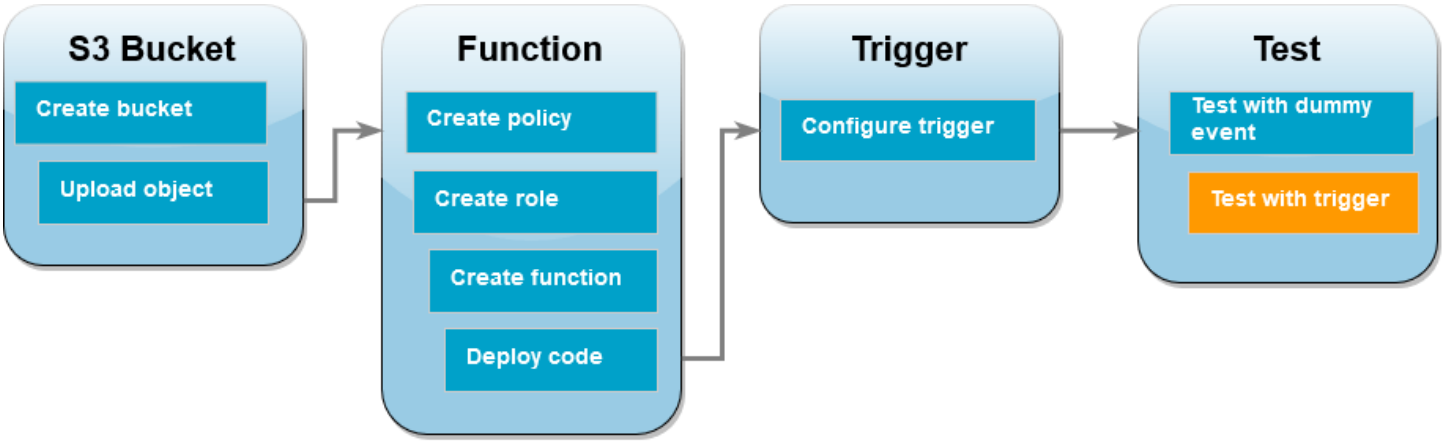
- Ersetzen Sie `test%2FKey` durch den Namen des Testobjekts, das Sie zuvor in Ihren Bucket hochgeladen haben (z. B.`HappyFace.jpg`).

```
{
  "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. Wählen Sie Speichern.
5. Wählen Sie Test aus.
6. Wenn Ihre Funktion erfolgreich ausgeführt wird, wird auf der Registerkarte Ausführungsergebnisse eine Ausgabe angezeigt, die der folgenden ähnelt.

```
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
```

Testen der Lambda-Funktion mit dem Amazon-S3-Auslöser



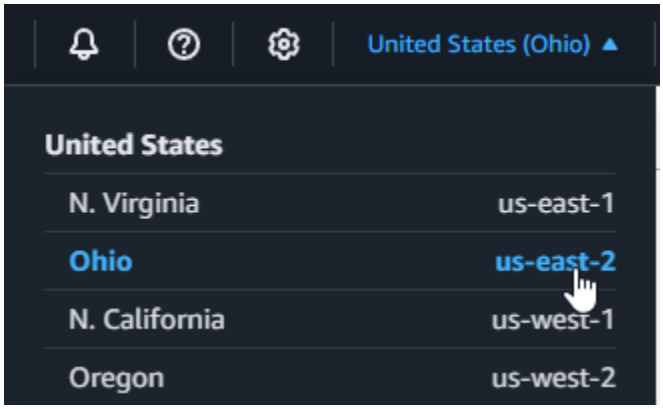
Um Ihre Funktion mit dem konfigurierten Auslöser zu testen, können Sie unter Verwendung der Konsole ein Objekt in Ihren Amazon-S3-Bucket hochladen. Um zu überprüfen, ob Ihre Lambda-Funktion wie erwartet ausgeführt wurde, verwenden Sie CloudWatch Logs, um die Ausgabe Ihrer Funktion einzusehen.

So laden Sie Objekte in Ihren Amazon-S3-Bucket hoch

1. Öffnen Sie die Buckets-Seite der Amazon-S3-Konsole und wählen Sie den Bucket aus, den Sie zuvor erstellt haben.
2. Klicken Sie auf Upload.
3. Wählen Sie Dateien hinzufügen aus und wählen Sie mit der Dateiauswahl ein Objekt aus, das Sie hochladen möchten. Bei diesem Objekt kann es sich um eine beliebige von Ihnen ausgewählte Datei handeln.
4. Wählen Sie Öffnen und anschließend Hochladen aus.

Um den Funktionsaufruf mithilfe von Logs zu überprüfen CloudWatch

1. Öffnen Sie die CloudWatch-Konsole.
2. Stellen Sie sicher, dass Sie in derselben Umgebung arbeiten, in der AWS-Region Sie Ihre Lambda-Funktion erstellt haben. Sie können Ihre Region mithilfe der Dropdown-Liste oben auf dem Bildschirm ändern.



3. Wählen Sie Protokolle und anschließend Protokollgruppen aus.
4. Wählen Sie die Protokollgruppe für Ihre Funktion (`/aws/lambda/s3-trigger-tutorial`) aus.
5. Wählen Sie im Bereich Protokollstreams den neuesten Protokollstream aus.
6. Wenn Ihre Funktion als Reaktion auf Ihren Amazon-S3-Auslöser korrekt aufgerufen wurde, erhalten Sie eine Ausgabe, die der folgenden ähnelt. Welchen `CONTENT TYPE` Sie sehen, hängt von der Art der Datei ab, die Sie in Ihren Bucket hochgeladen haben.

```
2022-05-09T23:17:28.702Z 0cae7f5a-b0af-4c73-8563-a3430333cc10 INFO CONTENT
 TYPE: image/jpeg
```

Bereinigen Ihrer Ressourcen

Sie können jetzt die Ressourcen, die Sie für dieses Tutorial erstellt haben, löschen, es sei denn, Sie möchten sie behalten. Indem Sie AWS Ressourcen löschen, die Sie nicht mehr verwenden, vermeiden Sie unnötige Kosten für Ihre AWS-Konto.

So löschen Sie die Lambda-Funktion:

1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.
2. Wählen Sie die Funktion aus, die Sie erstellt haben.
3. Wählen Sie Aktionen, Löschen aus.
4. Geben Sie **`confirm`** in das Texteingabefeld ein und wählen Sie Delete (Löschen) aus.

So löschen Sie die Ausführungsrolle

1. Öffnen Sie die Seite Roles in der IAM-Konsole.
2. Wählen Sie die von Ihnen erstellte Ausführungsrolle aus.
3. Wählen Sie Löschen aus.
4. Geben Sie den Namen der Rolle in das Texteingabefeld ein und wählen Sie Delete (Löschen) aus.

So löschen Sie den S3-Bucket:

1. Öffnen Sie die Amazon S3-Konsole.
2. Wählen Sie den Bucket aus, den Sie erstellt haben.
3. Wählen Sie Löschen aus.
4. Geben Sie den Namen des Buckets in das Texteingabefeld ein.
5. Wählen Sie Bucket löschen aus.

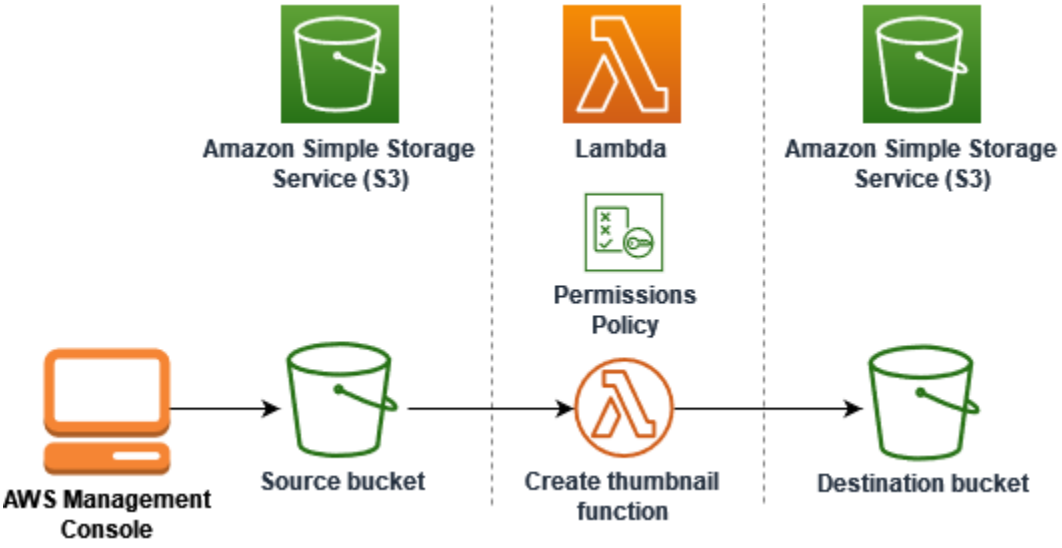
Nächste Schritte

In Tutorial: Verwenden eines Amazon-S3-Auslösers zum Erstellen von Miniaturbildern ruft der Amazon-S3-Auslöser eine Funktion auf, die für jede Bilddatei, die in einen Bucket hochgeladen wird, ein Miniaturbild erstellt. Dieses Tutorial erfordert ein gewisses Maß AWS an Wissen über Lambda-Domänen. Es zeigt, wie Ressourcen mithilfe von AWS Command Line Interface (AWS CLI)

erstellt werden und wie ein Bereitstellungspaket für das ZIP-Dateiarchiv für die Funktion und ihre Abhängigkeiten erstellt wird.

Tutorial: Verwenden eines Amazon-S3-Auslösers zum Erstellen von Miniaturbildern

In diesem Tutorial wird eine Lambda-Funktion erstellt und konfiguriert, die die Größe von Bildern anpasst, die einem Amazon Simple Storage Service (Amazon S3)-Bucket hinzugefügt werden. Wenn Sie Ihrem Bucket eine Bilddatei hinzufügen, ruft Amazon S3 Ihre Lambda-Funktion auf. Die Funktion erstellt daraufhin eine Miniaturversion des Bilds und gibt sie an einen anderen Amazon-S3-Bucket aus.



Führen Sie für dieses Tutorial die folgenden Schritte aus:

1. Erstellen Sie Quell- und Ziel-Buckets für Amazon S3 und laden Sie ein Beispielbild hoch.
2. Erstellen Sie eine Lambda-Funktion, die die Größe eines Bildes anpasst und eine Miniaturansicht an einen Amazon-S3-Bucket ausgibt.
3. Konfigurieren Sie einen Lambda-Auslöser, der Ihre Funktion aufruft, wenn Objekte in Ihren Quell-Bucket hochgeladen werden.
4. Testen Sie Ihre Funktion zunächst mit einem Dummy-Ereignis und anschließend durch Hochladen eines Bilds in Ihren Quell-Bucket.

Diese Schritte zeigen, wie Sie Lambda verwenden, um eine Dateiverarbeitungsaufgabe für Objekte auszuführen, die einem Amazon-S3-Bucket hinzugefügt werden. Sie können dieses Tutorial mit AWS Command Line Interface (AWS CLI) oder dem AWS-Managementkonsole abschließen.

Ein einfacheres Beispiel, das zeigt, wie Sie einen Amazon-S3-Auslöser für Lambda konfigurieren, finden Sie unter Tutorial: Verwenden eines Amazon-S3-Auslösers zum Aufrufen einer Lambda-Funktion.

Themen

- Voraussetzungen
- Erstellen von zwei Amazon-S3-Buckets
- Hochladen eines Testbilds in Ihren Quell-Bucket
- Erstellen einer Berechtigungsrichtlinie
- Erstellen einer Ausführungsrolle
- Erstellen des Bereitstellungspakets für die Funktion
- So erstellen Sie die Lambda-Funktion:
- Konfigurieren von Amazon S3 zum Aufrufen der Funktion
- Testen Ihrer Lambda-Funktion mit einem Dummy-Ereignis
- Testen Ihrer Funktion mit dem Amazon-S3-Auslöser
- Bereinigen Ihrer Ressourcen

Voraussetzungen

Wenn Sie den verwenden möchten, AWS CLI um das Tutorial abzuschließen, installieren Sie die neueste Version von AWS Command Line Interface.

Für Ihren Lambda-Funktionscode können Sie Python, Node.js oder Java verwenden. Installieren Sie die Tools zur Sprachunterstützung und einen Paketmanager für die gewünschte Sprache.

Installieren Sie das AWS Command Line Interface

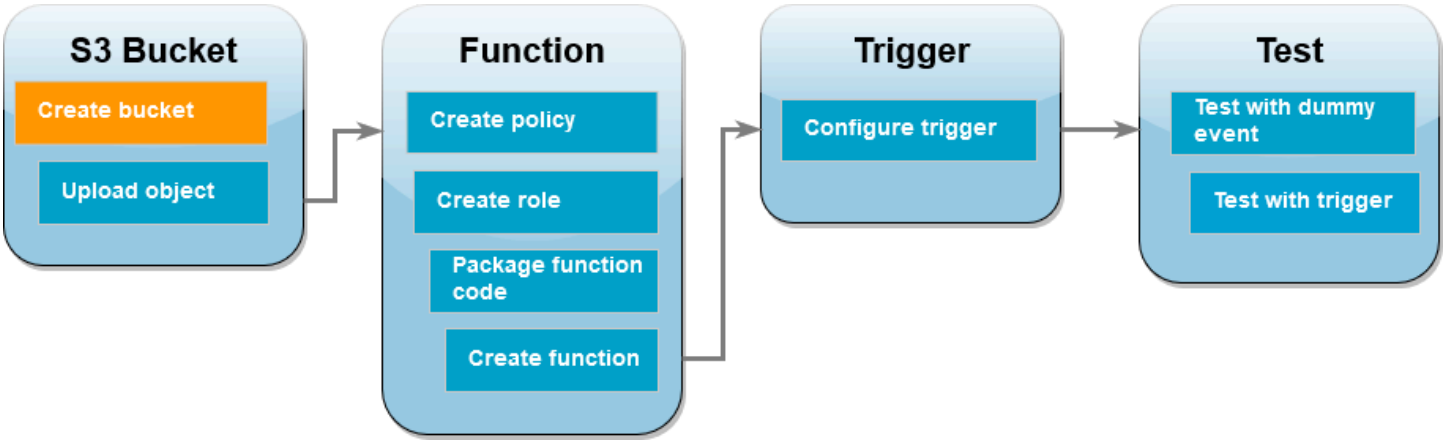
Wenn Sie das noch nicht installiert haben AWS Command Line Interface, folgen Sie den Schritten unter Installieren oder Aktualisieren der neuesten Version von AWS CLI, um es zu installieren.

Das Tutorial erfordert zum Ausführen von Befehlen ein Befehlszeilenterminal oder eine Shell. Verwenden Sie unter Linux und macOS Ihre bevorzugte Shell und Ihren bevorzugten Paketmanager.

> **Note**
>
> In Windows werden einige Bash-CLI-Befehle, die Sie häufig mit Lambda verwenden (z. B. `zip`), von den integrierten Terminals des Betriebssystems nicht unterstützt. Um eine in

Windows integrierte Version von Ubuntu und Bash zu erhalten, installieren Sie das Windows-Subsystem für Linux.

Erstellen von zwei Amazon-S3-Buckets

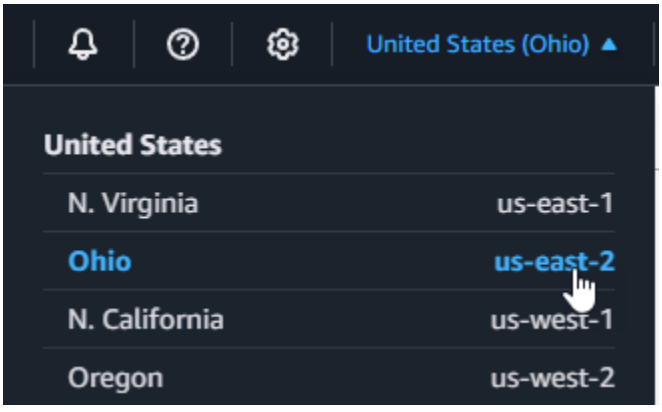


Erstellen Sie zunächst zwei Amazon-S3-Buckets. Der erste Bucket ist der Quell-Bucket, in den Sie Ihre Bilder hochladen. Der zweite Bucket wird von Lambda verwendet, um das verkleinerte Miniaturbild zu speichern, wenn Sie Ihre Funktion aufrufen.

AWS-Managementkonsole

So erstellen Sie die Amazon-S3-Buckets (Konsole)

1. Öffnen Sie die Amazon-S3-Konsole und wählen Sie die Seite Allzweck-Buckets aus.
2. Wählen Sie den Ort aus, der Ihrem geografischen Standort AWS-Region am nächsten liegt. Sie können Ihre Region mithilfe der Dropdown-Liste am oberen Bildschirmrand ändern. Später im Tutorial müssen Sie eine Lambda-Funktion in derselben Region erstellen.



3. Wählen Sie Create Bucket (Bucket erstellen) aus.

4. Führen Sie unter Allgemeine Konfiguration die folgenden Schritte aus:

   a. Stellen Sie sicher, dass für Bucket-Typ die Option Allzweck ausgewählt ist.

   b. Geben Sie für den Bucket-Namen einen global eindeutigen Namen ein, der den Regeln für die Bucket-Benennung von Amazon S3 entspricht. Bucket-Namen dürfen nur aus Kleinbuchstaben, Zahlen, Punkten (.) und Bindestrichen (-) bestehen.

5. Belassen Sie alle anderen Optionen auf ihren Standardwerten und wählen Sie Bucket erstellen aus.

6. Wiederholen Sie die Schritte 1 bis 5, um Ihren Ziel-Bucket zu erstellen. Geben Sie unter Bucket-Name den Namen **`amzn-s3-demo-source-bucket-resized`** ein. **`amzn-s3-demo-source-bucket`** ist hierbei der Name des Quell-Buckets, den Sie soeben erstellt haben.

AWS CLI

So erstellen Sie die Amazon-S3-Buckets (AWS CLI)

1. Führen Sie den folgenden CLI-Befehl aus, um Ihren Quell-Bucket zu erstellen. Der gewählte Name für Ihren Bucket muss global eindeutig sein und den Regeln für die Benennung von Buckets für Amazon S3 entsprechen. Namen dürfen nur Kleinbuchstaben, Zahlen, Punkte (.) und Bindestriche (-) enthalten. Wählen Sie für `region` und `LocationConstraint` die AWS-Region aus, die Ihrem geografischen Standort am nächsten ist.

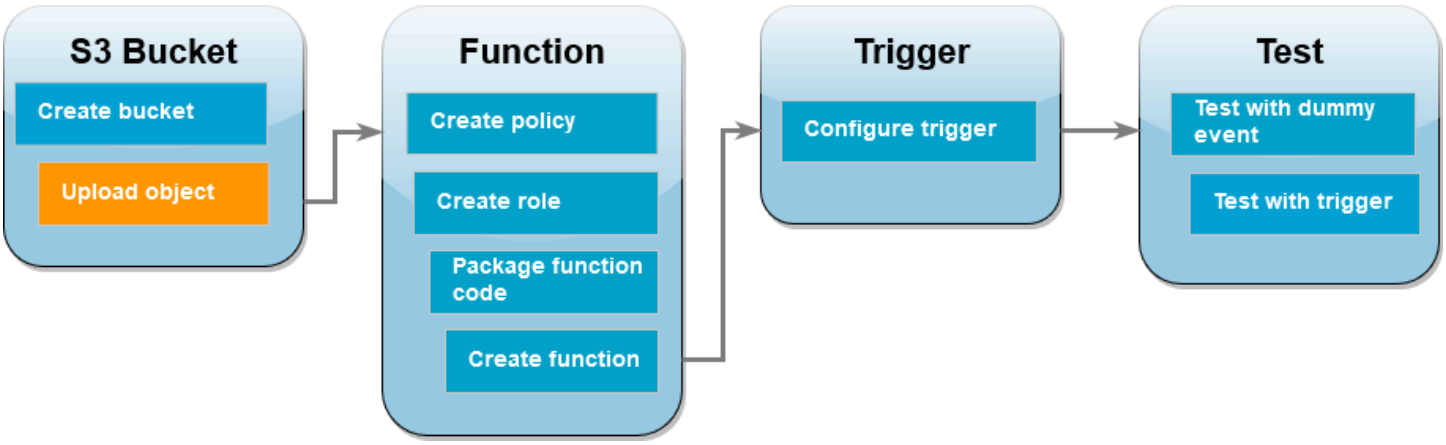
   ```
   aws s3api create-bucket --bucket amzn-s3-demo-source-bucket --region us-east-1 \
   --create-bucket-configuration LocationConstraint=us-east-1
   ```

   Später im Tutorial müssen Sie Ihre Lambda-Funktion im selben Verzeichnis AWS-Region wie in Ihrem Quell-Bucket erstellen. Notieren Sie sich also die Region, die Sie ausgewählt haben.

2. Führen Sie den folgenden Befehl aus, um Ihren Ziel-Bucket zu erstellen. Verwenden Sie für den Bucket-Namen **`amzn-s3-demo-source-bucket-resized`**. **`amzn-s3-demo-source-bucket`** ist hierbei der Name des Quell-Buckets, den Sie in Schritt 1 erstellt haben. Wählen Sie für `region` und dasselbe aus`LocationConstraint`, das AWS-Region Sie zur Erstellung Ihres Quell-Buckets verwendet haben.

   ```
   aws s3api create-bucket --bucket amzn-s3-demo-source-bucket-resized --region us-
   east-1 \
   --create-bucket-configuration LocationConstraint=us-east-1
   ```

Hochladen eines Testbilds in Ihren Quell-Bucket



Später im Tutorial testen Sie Ihre Lambda-Funktion, indem Sie sie mit der AWS CLI oder der Lambda-Konsole aufrufen. Damit Sie sich vergewissern können, dass Ihre Funktion ordnungsgemäß funktioniert, muss Ihr Quell-Bucket ein Testbild enthalten. Bei diesem Bild kann es sich um eine beliebige JPG- oder PNG-Datei handeln.

AWS-Managementkonsole

So laden Sie ein Testbild in Ihren Quell-Bucket hoch (Konsole)

1. Öffnen Sie die Seite Buckets der Amazon-S3-Konsole.
2. Wählen Sie den Quell-Bucket aus, den Sie im vorherigen Schritt erstellt haben.
3. Klicken Sie auf Hochladen.
4. Wählen Sie Dateien hinzufügen aus und wählen Sie mit der Dateiauswahl das Objekt aus, das Sie hochladen möchten.
5. Wählen Sie Öffnen und anschließend Hochladen aus.

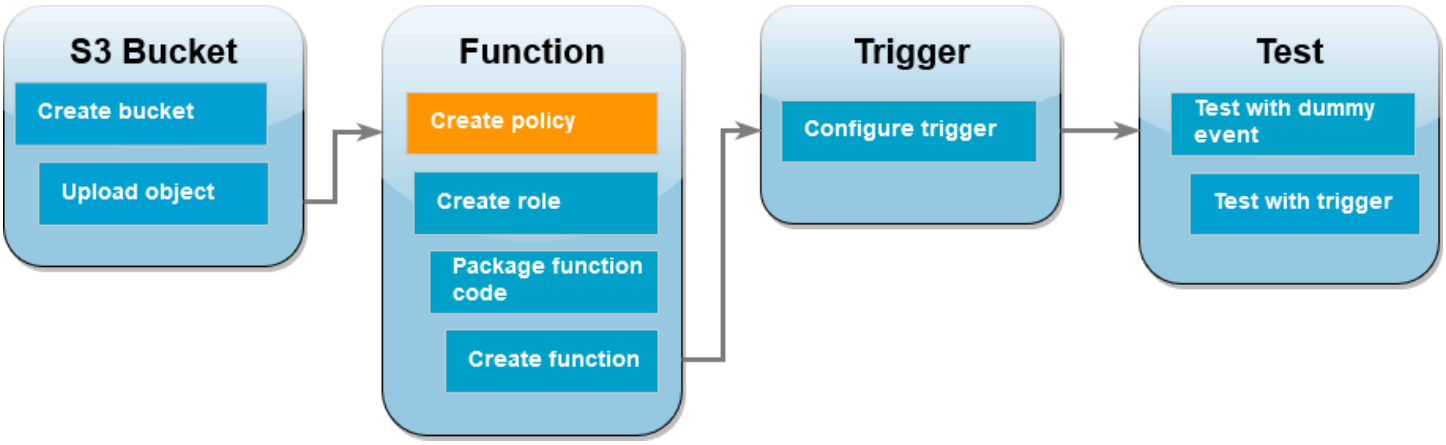
AWS CLI

So laden Sie ein Testbild in Ihren Quell-Bucket hoch (AWS CLI)

- Führen Sie in dem Verzeichnis, das das hochzuladende Bild enthält, den folgenden CLI-Befehl aus. Ersetzen Sie dabei den Parameter `--bucket` durch den Namen Ihres Quell-Buckets. Verwenden Sie für die Parameter `--key` und `--body` den Dateinamen Ihres Testbilds.

```
aws s3api put-object --bucket amzn-s3-demo-source-bucket --key HappyFace.jpg --
body ./HappyFace.jpg
```

Erstellen einer Berechtigungsrichtlinie



Im ersten Schritt der Erstellung Ihrer Lambda-Funktion wird eine Berechtigungsrichtlinie erstellt. Diese Richtlinie gibt Ihrer Funktion die Berechtigungen, die sie für den Zugriff auf andere Ressourcen benötigt. AWS In diesem Tutorial gewährt die Richtlinie Lambda Lese- und Schreibberechtigungen für Amazon S3 S3-Buckets und ermöglicht es ihm, in Amazon CloudWatch Logs zu schreiben.

AWS-Managementkonsole

So erstellen Sie die Richtlinie (Konsole)

1. Öffnen Sie die Seite Richtlinien der AWS Identity and Access Management (IAM-) Konsole.
2. Wählen Sie Richtlinie erstellen aus.
3. Wählen Sie die Registerkarte JSON aus und kopieren Sie dann die folgende benutzerdefinierte JSON-Richtlinie in den JSON-Editor.

```
{
    "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. Wählen Sie Weiter aus.
5. Geben Sie unter Richtliniendetails für den Richtliniennamen **`LambdaS3Policy`** ein.
6. Wählen Sie Richtlinie erstellen aus.

AWS CLI

So erstellen Sie die Richtlinie (AWS CLI)

1. Speichern Sie den folgenden JSON-Code in einer Datei mit dem Namen `policy.json`:

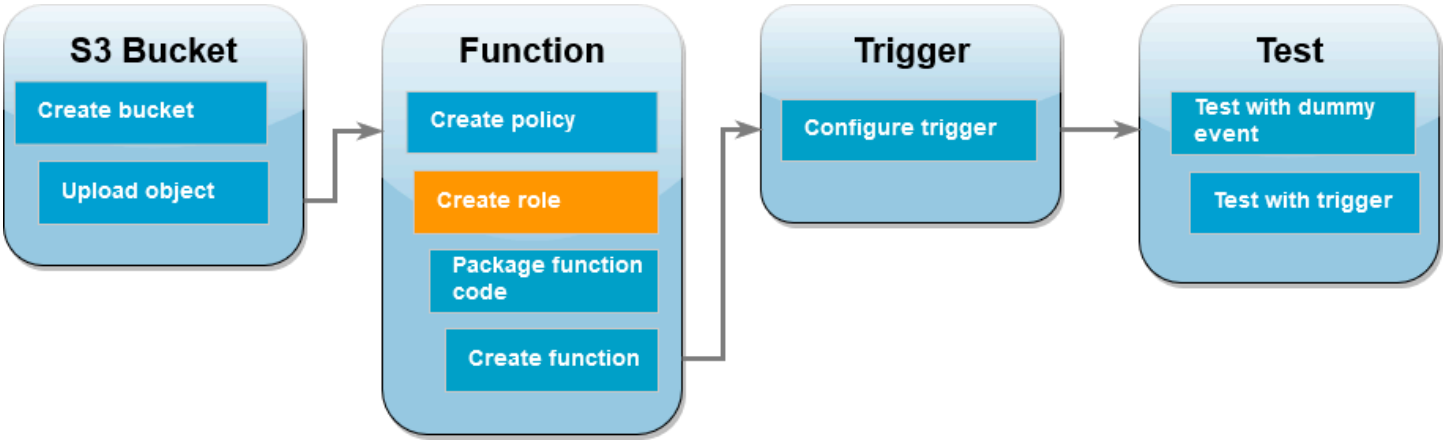
```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "logs:PutLogEvents",
                "logs:CreateLogGroup",
                "logs:CreateLogStream"
            ],
```

```
            "Resource": "arn:aws:logs:*:*:*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "s3:GetObject"
            ],
            "Resource": "arn:aws:s3:::*/*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "s3:PutObject"
            ],
            "Resource": "arn:aws:s3:::*/*"
        }
    ]
}
```

2. Führen Sie in dem Verzeichnis, in dem Sie das JSON-Richtliniendokument gespeichert haben, den folgenden CLI-Befehl aus:

```
aws iam create-policy --policy-name LambdaS3Policy --policy-document file://
policy.json
```

Erstellen einer Ausführungsrolle



Eine Ausführungsrolle ist eine IAM-Rolle, die einer Lambda-Funktion Zugriff AWS-Services und Ressourcen gewährt. Um Ihrer Funktion Lese- und Schreibzugriff auf einen Amazon-S3-Bucket zu erteilen, fügen Sie die Berechtigungsrichtlinie an, die Sie im vorherigen Schritt erstellt haben.

AWS-Managementkonsole

So erstellen Sie eine Ausführungsrolle und fügen Ihre Berechtigungsrichtlinie hinzu (Konsole)

1. Öffnen Sie die Seite Rollen der (IAM-)Konsole.
2. Wählen Sie Rolle erstellen aus.
3. Wählen Sie unter Vertrauenswürdiger Entitätstyp die Option AWS-Service und unter Anwendungsfall die Option Lambda aus.
4. Wählen Sie Weiter aus.
5. Fügen Sie die Berechtigungsrichtlinie hinzu, die Sie im vorherigen Schritt erstellt haben:
   a. Geben Sie im Feld für die Richtliniensuche **`LambdaS3Policy`** ein.
   b. Aktivieren Sie in den Suchergebnissen das Kontrollkästchen für `LambdaS3Policy`.
   c. Wählen Sie Weiter aus.
6. Geben Sie unter Rollendetails im Feld Rollenname den Namen **`LambdaS3Role`** ein.
7. Wählen Sie Rolle erstellen aus.

AWS CLI

So erstellen Sie eine Ausführungsrolle und fügen Ihre Berechtigungsrichtlinie an (AWS CLI)

1. Speichern Sie den folgenden JSON-Code in einer Datei mit dem Namen `trust-policy.json`: Diese Vertrauensrichtlinie ermöglicht es Lambda, die Berechtigungen der Rolle zu verwenden, indem dem Dienstprinzipal die `lambda.amazonaws.com` Erlaubnis erteilt wird, die `AssumeRole` Aktion AWS -Security-Token-Service (AWS STS) aufzurufen.

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "lambda.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
```

```
}
```

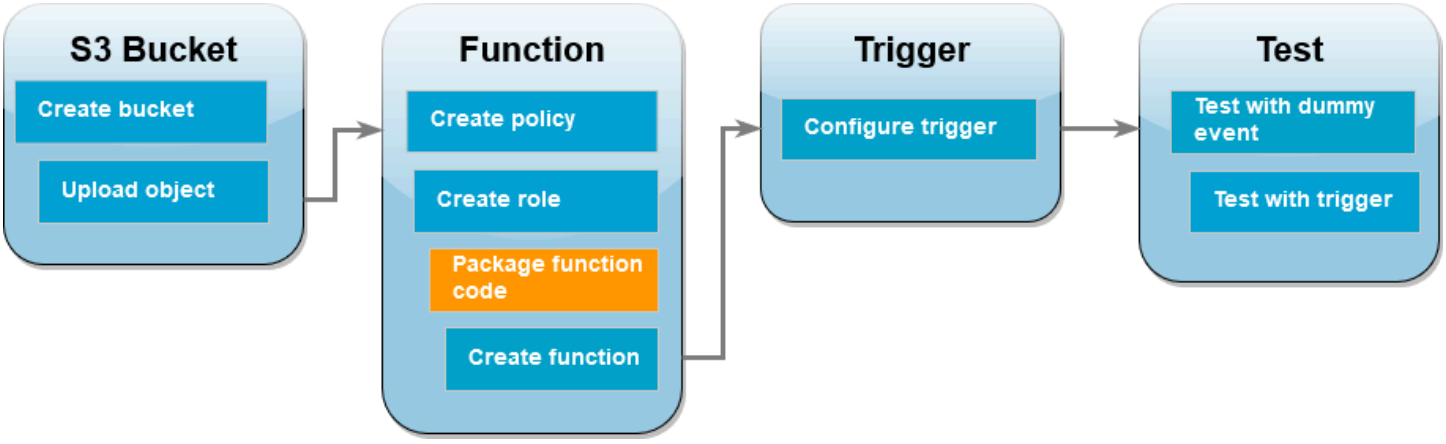
2. Führen Sie in dem Verzeichnis, in dem Sie das JSON-Vertrauensrichtliniendokument gespeichert haben, den folgenden CLI-Befehl aus, um die Ausführungsrolle zu erstellen:

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

3. Führen Sie den folgenden CLI-Befehl aus, um die im vorherigen Schritt erstellte Berechtigungsrolle anzufügen. Ersetzen Sie die AWS-Konto Nummer im ARN der Police durch Ihre eigene Kontonummer.

```
aws iam attach-role-policy --role-name LambdaS3Role --policy-arn
 arn:aws:iam::123456789012:policy/LambdaS3Policy
```

Erstellen des Bereitstellungspakets für die Funktion



Um Ihre Funktion zu erstellen, erstellen Sie ein Bereitstellungspaket, das Ihren Funktionscode und die zugehörigen Abhängigkeiten enthält. Bei der hier verwendeten Funktion `CreateThumbnail` verwendet Ihr Funktionscode eine separate Bibliothek für die Anpassung der Bildgröße. Folgen Sie den Anweisungen für die von Ihnen gewählte Sprache, um ein Bereitstellungspaket zu erstellen, das die erforderliche Bibliothek enthält.

Node.js

So erstellen Sie das Bereitstellungspaket (Node.js)

1. Erstellen Sie ein Verzeichnis namens `lambda-s3` für Ihren Funktionscode und die Abhängigkeiten und navigieren Sie dorthin.

```
mkdir lambda-s3
cd lambda-s3
```

2. Erstellen Sie ein neues Node.js-Projekt mit npm. Um die im interaktiven Erlebnis bereitgestellten Standardoptionen zu akzeptieren, drücken Sie `Enter`.

```
npm init
```

3. Speichern Sie den folgenden Funktionscode in einer lokalen Datei namens `index.mjs`: Stellen Sie sicher, dass Sie `us-east-1` durch das AWS-Region ersetzen, in dem Sie Ihre eigenen Quell- und Ziel-Buckets erstellt haben.

```
// 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. Installieren Sie die Sharp-Bibliothek mithilfe von npm im Verzeichnis `lambda-s3`. Beachten Sie, dass die neueste Version von Sharp (0.33) nicht mit Lambda kompatibel ist. Installieren Sie für dieses Tutorial daher Version 0.32.6.

```
npm install sharp@0.32.6
```

Der npm-Befehl `install` erstellt ein Verzeichnis namens `node_modules` für Ihre Module. Nach diesem Schritt sollte Ihre Verzeichnisstruktur wie folgt aussehen:

```
lambda-s3
|- index.mjs
|- node_modules
|  |- base64js
|  |- bl
|  |- buffer
...
|- package-lock.json
|- package.json
```

5. Erstellen Sie ein ZIP-Bereitstellungspaket, das Ihren Funktionscode und die zugehörigen Abhängigkeiten enthält. Führen Sie unter macOS oder Linux den folgenden Befehl aus:

```
zip -r function.zip .
```

Verwenden Sie unter Windows Ihr bevorzugtes ZIP-Programm, um eine ZIP-Datei zu erstellen. Achten Sie darauf, dass sich die Dateien `index.mjs`, `package.json` und `package-lock.json` und das Verzeichnis `node_modules` im Stammverzeichnis der ZIP-Datei befinden.

Python

So erstellen Sie das Bereitstellungspaket (Python)

1. Speichern Sie den Beispielcode als Datei mit dem Namen `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. Erstellen Sie in dem Verzeichnis, in dem Sie auch Ihre `lambda_function.py`-Datei erstellt haben, ein neues Verzeichnis mit dem Namen `package` und installieren Sie die Bibliothek Pillow (PIL) sowie das AWS SDK für Python (Boto3). Die Lambda-Python-Laufzeit enthält zwar eine Version des Boto3 SDK, es empfiehlt sich jedoch, dem Bereitstellungspaket alle Abhängigkeiten Ihrer Funktion hinzuzufügen, auch wenn sie in der Laufzeit enthalten sind. Weitere Informationen finden Sie in der Laufzeitabhängigkeiten in Python.

```
mkdir package
pip install \
--platform manylinux2014_x86_64 \
--target=package \
--implementation cp \
--python-version 3.12 \
--only-binary=:all: --upgrade \
pillow boto3
```

   Die Pillow-Bibliothek enthält Code in C/C++. Mithilfe der Optionen `--platform manylinux_2014_x86_64` und `--only-binary=:all:` lädt pip eine Version von Pillow herunter und installiert sie, die vorkompilierte Binärdateien enthält, die mit dem Betriebssystem Amazon Linux 2 kompatibel sind. Dadurch wird sichergestellt, dass Ihr Bereitstellungspaket in der Lambda-Ausführungsumgebung funktioniert, unabhängig vom Betriebssystem und der Architektur Ihrer lokalen Build-Maschine.

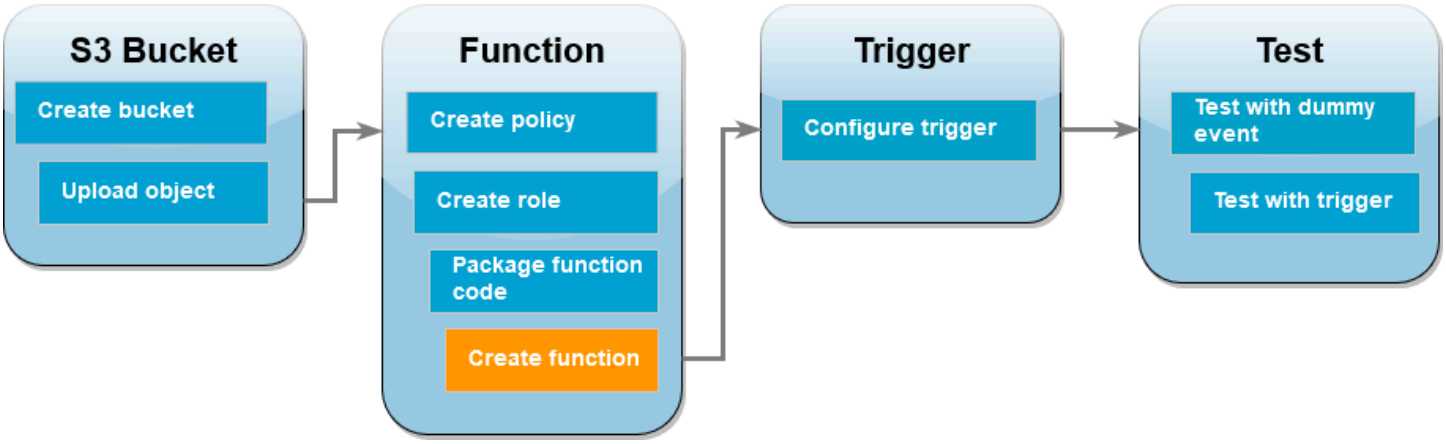
3. Erstellen Sie eine ZIP-Datei, die Ihren Anwendungscode sowie die Pillow- und die Boto3-Bibliothek enthält. Führen Sie unter Linux oder MacOS die folgenden Befehle über Ihre Befehlszeilenschnittstelle aus:

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

   Verwenden Sie unter Windows Ihr bevorzugtes ZIP-Tool, um die `lambda_function.zip`-Datei zu erstellen. Achten Sie darauf, dass sich die Datei `lambda_function.py` und die Ordner, die Ihre Abhängigkeiten enthalten, im Stammverzeichnis der ZIP-Datei befinden.

Sie können Ihr Bereitstellungspaket auch in einer virtuellen Python-Umgebung erstellen. Siehe Arbeiten mit ZIP-Dateiarchiven und Python-Lambda-Funktionen

So erstellen Sie die Lambda-Funktion:



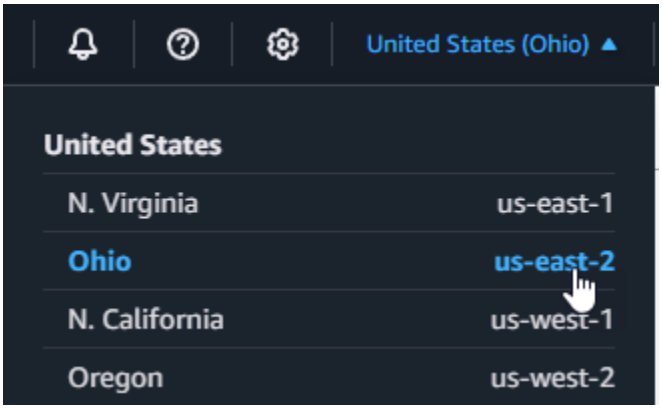
Sie können Ihre Lambda-Funktion entweder mit der AWS CLI oder der Lambda-Konsole erstellen. Folgen Sie den Anweisungen für die von Ihnen gewählte Sprache, um die Funktion zu erstellen.

AWS-Managementkonsole

So erstellen Sie die Funktion (Konsole)

Wenn Sie Ihre Lambda-Funktion über die Konsole erstellen möchten, erstellen Sie zunächst eine Basisfunktion, die etwas Hello-World-Code enthält. Anschließend ersetzen Sie diesen Code durch Ihren eigenen Funktionscode, indem Sie die im vorherigen Schritt erstellte ZIP- oder JAR-Datei hochladen.

1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.
2. Stellen Sie sicher, dass Sie in demselben Modus arbeiten, in dem AWS-Region Sie Ihren Amazon S3 S3-Bucket erstellt haben. Sie können Ihre Region mithilfe der Dropdown-Liste am oberen Bildschirmrand ändern.



3. Wählen Sie Funktion erstellen.
4. Wählen Sie Author from scratch aus.
5. Führen Sie unter Basic information (Grundlegende Informationen) die folgenden Schritte aus:
   a. Geben Sie für Function name (Funktionsname) **`CreateThumbnail`** ein.
   b. Wählen Sie für Laufzeit entweder Node.js 22.x oder Python 3.12, je nachdem, welche Sprache Sie für Ihre Funktion gewählt haben.
   c. Wählen Sie für Architektur x86_64 aus.
6. Gehen Sie auf der Registerkarte Standard-Ausführungsrolle ändern wie folgt vor:
   a. Erweitern Sie die Registerkarte und wählen Sie dann Verwenden einer vorhandenen Rolle aus.
   b. Wählen Sie die zuvor erstellte `LambdaS3Role` aus.
7. Wählen Sie Funktion erstellen.

So laden Sie den Funktionscode hoch (Konsole)

1. Wählen Sie im Bereich Codequelle die Option Hochladen von aus.
2. Wählen Sie die ZIP-Datei aus.
3. Klicken Sie auf Upload.
4. Wählen Sie in der Dateiauswahl Ihre ZIP-Datei und anschließend Öffnen aus.
5. Wählen Sie Speichern.

AWS CLI

So erstellen Sie die Funktion (AWS CLI)

- Führen Sie den CLI-Befehl für die von Ihnen gewählte Sprache aus. Stellen Sie sicher, dass Sie den `role` Parameter durch Ihre eigene AWS-Konto ID `123456789012` ersetzen. Ersetzen Sie für den Parameter `region` den Wert `us-east-1` durch die Region, in der Sie Ihre Amazon-S3-Buckets erstellt haben.

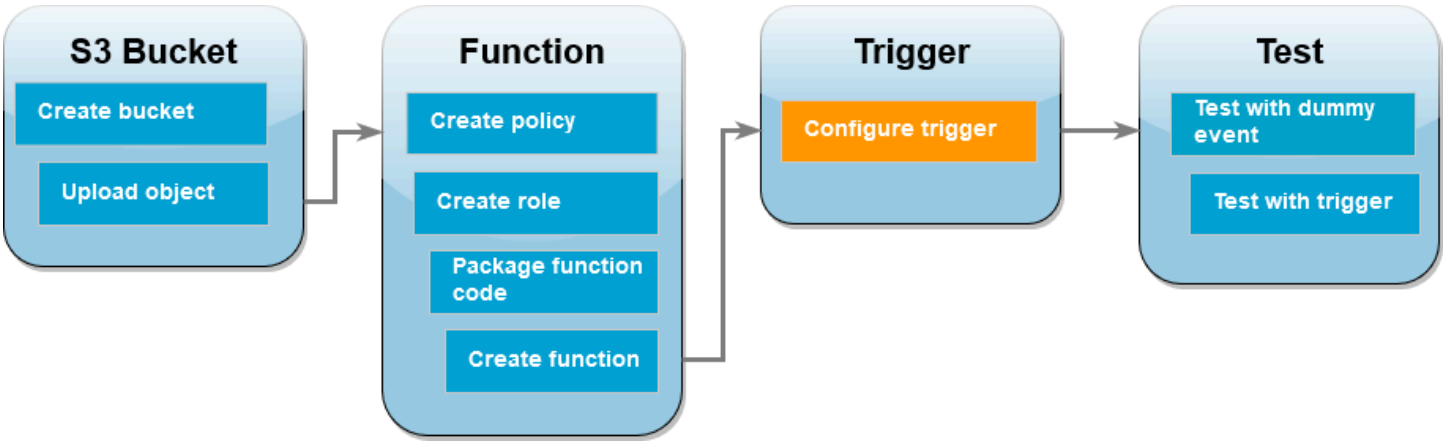
  - Führen Sie für Node.js den folgenden Befehl in dem Verzeichnis aus, das die Datei `function.zip` enthält:

```
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
```

  - Führen Sie für Python den folgenden Befehl in dem Verzeichnis aus, das die Datei `lambda_function.zip` enthält:

```
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
```

Konfigurieren von Amazon S3 zum Aufrufen der Funktion



Damit Ihre Lambda-Funktion ausgeführt wird, wenn Sie ein Bild in Ihren Quell-Bucket hochladen, müssen Sie einen Auslöser für Ihre Funktion konfigurieren. Sie können den Amazon-S3-Auslöser entweder über die Konsole oder mithilfe der AWS CLI konfigurieren.

> **Important**
>
> Mit diesem Verfahren wird der Amazon-S3-Bucket so konfiguriert, dass die Funktion jedes Mal aufgerufen wird, wenn ein Objekt im Bucket erstellt wird. Konfigurieren Sie dieses Verhalten nur für den Quell-Bucket. Wenn Ihre Lambda-Funktion Objekte in dem Bucket erstellt, der sie aufruft, kann es passieren, dass Ihre Funktion kontinuierlich in einer Schleife aufgerufen wird. Dies kann dazu führen, dass Ihnen AWS-Konto unerwartete Gebühren in Rechnung gestellt werden.

AWS-Managementkonsole

So konfigurieren Sie den Amazon-S3-Auslöser (Konsole)

1. Öffnen Sie die Seite Funktionen der Lambda-Konsole und wählen Sie Ihre Funktion (`CreateThumbnail`) aus.
2. Wählen Sie Add trigger.
3. Wählen Sie S3 aus.
4. Wählen Sie unter Bucket Ihren Quell-Bucket aus.
5. Wählen Sie unter Ereignistypen die Option Alle Objekterstellungsereignisse aus.
6. Aktivieren Sie unter Rekursiver Aufruf das Kontrollkästchen, um zu bestätigen, dass die Verwendung desselben Amazon-S3-Buckets für die Ein- und Ausgabe nicht empfohlen wird. Weitere Informationen zu rekursiven Aufrufmustern in Lambda finden Sie bei Serverless Land unter Recursive patterns that cause run-away Lambda functions.
7. Wählen Sie Hinzufügen aus.

   Wenn Sie einen Auslöser über die Lambda-Konsole erstellen, erstellt Lambda automatisch eine ressourcenbasierte Richtlinie, um dem von Ihnen ausgewählten Dienst die Berechtigung zum Aufrufen Ihrer Funktion zu erteilen.

AWS CLI

So konfigurieren Sie den Amazon-S3-Auslöser (AWS CLI)

1. Damit Ihr Amazon-S3-Quell-Bucket Ihre Funktion aufruft, wenn Sie eine Bilddatei hinzufügen, müssen Sie zunächst mithilfe einer ressourcenbasierten Richtlinie Berechtigungen für Ihre Funktion konfigurieren. Eine ressourcenbasierte Grundsatzerklärung erteilt anderen Personen die AWS-Services Erlaubnis, Ihre Funktion aufzurufen. Führen Sie den folgenden CLI-Befehl aus, um Amazon S3 die Berechtigung zum Aufrufen Ihrer Funktion zu erteilen. Achten Sie darauf, den `source-account` Parameter durch Ihre eigene AWS-Konto ID zu ersetzen und Ihren eigenen Quell-Bucket-Namen zu verwenden.

```
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
```

   Die mit diesem Befehl definierte Richtlinie ermöglicht es Amazon S3 nur, Ihre Funktion aufzurufen, wenn eine Aktion in Ihrem Quell-Bucket ausgeführt wird.

   > **Note**
   >
   > Namen von Amazon-S3-Buckets sind zwar global eindeutig, trotzdem empfiehlt es sich, bei der Verwendung ressourcenbasierter Richtlinien anzugeben, dass der Bucket zu Ihrem Konto gehören muss. Denn wenn Sie einen Bucket löschen, kann ein anderer AWS-Konto Bucket einen Bucket mit demselben Amazon-Ressourcennamen (ARN) erstellen.

2. Speichern Sie den folgenden JSON-Code in einer Datei mit dem Namen `notification.json`: Wenn dieser JSON-Code auf Ihren Quell-Bucket angewendet wird, wird der Bucket so konfiguriert, dass er jedes Mal, wenn ein neues Objekt hinzugefügt wird, eine Benachrichtigung an Ihre Lambda-Funktion sendet. Ersetzen Sie die AWS-Konto Nummer und AWS-Region in der Lambda-Funktion ARN durch Ihre eigene Kontonummer und Region.

```
{
"LambdaFunctionConfigurations": [
    {
```

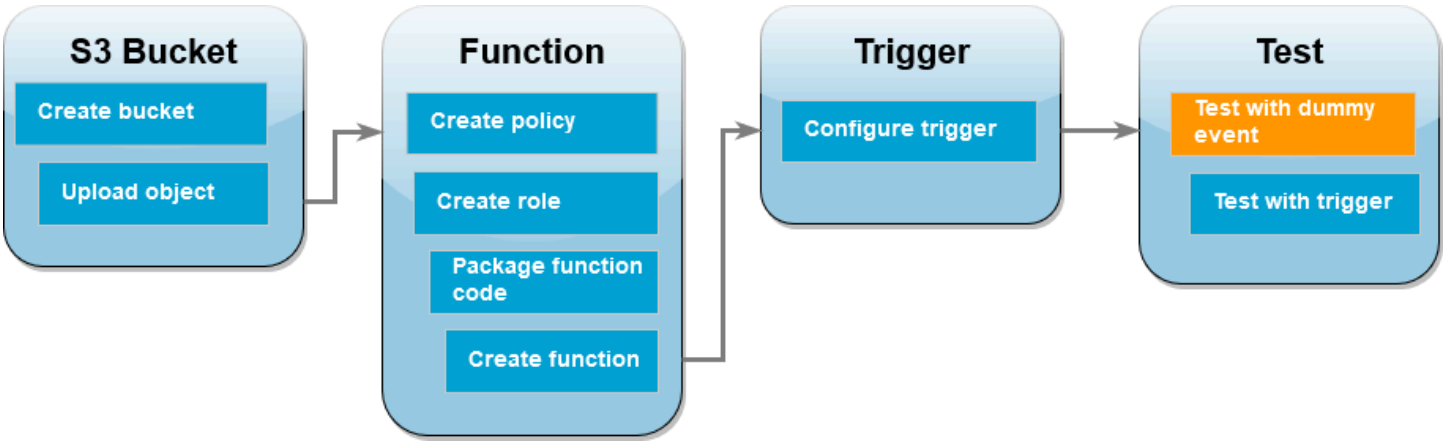
```
      "Id": "CreateThumbnailEventConfiguration",
      "LambdaFunctionArn": "arn:aws:lambda:us-
east-1:123456789012:function:CreateThumbnail",
      "Events": [ "s3:ObjectCreated:Put" ]
    }
  ]
}
```

3. Führen Sie den folgenden CLI-Befehl aus, um die Benachrichtigungseinstellungen in der von Ihnen erstellten JSON-Datei auf Ihren Quell-Bucket anzuwenden. Ersetzen Sie dabei `amzn-s3-demo-source-bucket` durch den Namen Ihres Quell-Buckets.

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

   Weitere Informationen zu dem `put-bucket-notification-configuration` Befehl und der `notification-configuration` Option finden Sie put-bucket-notification-configurationin der AWS CLI-Befehlsreferenz.

Testen Ihrer Lambda-Funktion mit einem Dummy-Ereignis



Bevor Sie Ihr gesamtes Setup testen, indem Sie Ihrem Amazon-S3-Quell-Bucket eine Bilddatei hinzufügen, testen Sie, ob Ihre Lambda-Funktion ordnungsgemäß funktioniert, indem Sie sie mit einem Dummy-Ereignis aufrufen. In Lambda ist ein Ereignis ein Dokument im JSON-Format, das Daten enthält, die von Ihrer Lambda-Funktion verarbeitet werden sollen. Wenn Ihre Funktion von Amazon S3 aufgerufen wird, enthält das an Ihre Funktion gesendete Ereignis Informationen wie den Bucket-Namen, den Bucket-ARN und den Objektschlüssel.

AWS-Managementkonsole

So testen Sie Ihre Lambda-Funktion mit einem Dummy-Ereignis (Konsole)

1. Öffnen Sie die Seite Funktionen der Lambda-Konsole und wählen Sie Ihre Funktion (`CreateThumbnail`) aus.
2. Wählen Sie die Registerkarte Test.
3. Gehen Sie im Bereich Testereignis wie folgt vor, um Ihr Testereignis zu erstellen:
   a. Wählen Sie unter Testereignisaktion die Option Neues Ereignis erstellen aus.
   b. Geben Sie für Event name (Ereignisname) **`myTestEvent`** ein.
   c. Wählen Sie unter Vorlage die Option S3 Put aus.
   d. Ersetzen Sie die Werte für die folgenden Parameter durch Ihre eigenen Werte:
      - Ersetzen Sie für durch den`awsRegion`, in `us-east-1` dem AWS-Region Sie Ihre Amazon S3 S3-Buckets erstellt haben.
      - Ersetzen Sie für `name` den Wert `amzn-s3-demo-bucket` durch den Namen Ihres eigenen Amazon-S3-Quell-Buckets.
      - Ersetzen Sie für `key` den Wert `test%2Fkey` durch den Dateinamen des Testobjekts, das Sie im Schritt Hochladen eines Testbilds in Ihren Quell-Bucket in Ihren Quell-Bucket hochgeladen haben.

```
{
  "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. Wählen Sie Speichern.

4. Wählen Sie im Bereich Testereignis die Option Testen aus.
5. Gehen Sie wie folgt vor, um zu überprüfen, ob Ihre Funktion eine verkleinerte Version Ihres Bilds erstellt und in Ihrem Amazon-S3-Ziel-Bucket gespeichert hat:

   a. Öffnen Sie die Seite Buckets der Amazon-S3-Konsole.

   b. Wählen Sie Ihren Ziel-Bucket aus und vergewissern Sie sich, dass die Datei, deren Größe angepasst wurde, im Bereich Objekte vorhanden ist.

AWS CLI

So testen Sie Ihre Lambda-Funktion mit einem Dummy-Ereignis (AWS CLI)

1. Speichern Sie den folgenden JSON-Code in einer Datei mit dem Namen `dummyS3Event.json`: Ersetzen Sie die Werte für die folgenden Parameter durch Ihre eigenen Werte:

- Ersetzen Sie für durch den`awsRegion`, in `us-east-1` dem AWS-Region Sie Ihre Amazon S3 S3-Buckets erstellt haben.
- Ersetzen Sie für `name` den Wert `amzn-s3-demo-bucket` durch den Namen Ihres eigenen Amazon-S3-Quell-Buckets.
- Ersetzen Sie für `key` den Wert `test%2Fkey` durch den Dateinamen des Testobjekts, das Sie im Schritt Hochladen eines Testbilds in Ihren Quell-Bucket in Ihren Quell-Bucket hochgeladen haben.

```
{
  "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. Rufen Sie die Funktion in dem Verzeichnis auf, in dem Sie die Datei `dummyS3Event.json` gespeichert haben, indem Sie den folgenden CLI-Befehl ausführen. Dieser Befehl ruft Ihre Lambda-Funktion synchron auf, indem `RequestResponse` als Wert des Parameters „invocation-type“ angegeben wird. Weitere Informationen zu synchronen und asynchronen Aufrufen finden Sie unter Aufrufen von Lambda-Funktionen.

```
aws lambda invoke --function-name CreateThumbnail \
--invocation-type RequestResponse --cli-binary-format raw-in-base64-out \
--payload file://dummyS3Event.json outputfile.txt
```

Die cli-binary-format Option ist erforderlich, wenn Sie Version 2 von verwenden. AWS CLI Um dies zur Standardeinstellung zu machen, führen Sie `aws configure set cli-binary-format raw-in-base64-out` aus. Weitere Informationen finden Sie unter AWS CLI - unterstützte globale Befehlszeilenoptionen.

3. Vergewissern Sie sich, dass Ihre Funktion eine Miniaturversion Ihres Bilds erstellt und in Ihrem Amazon-S3-Ziel-Bucket gespeichert hat. Führen Sie den folgenden CLI-Befehl aus und ersetzen Sie dabei `amzn-s3-demo-source-bucket-resized` durch den Namen Ihres eigenen Ziel-Buckets:

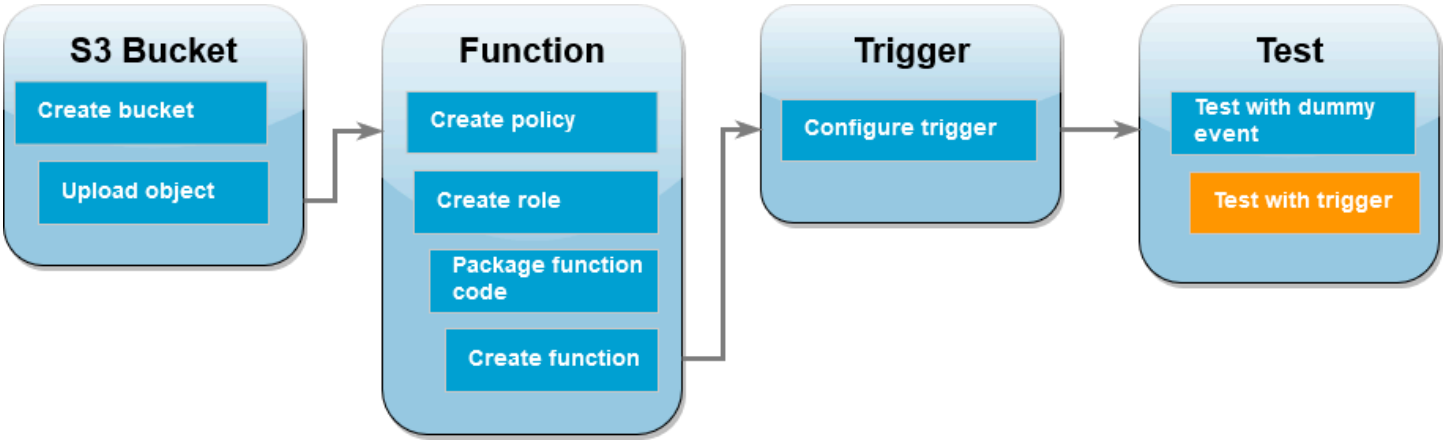
```
aws s3api list-objects-v2 --bucket amzn-s3-demo-source-bucket-resized
```

Die Ausgabe sollte in etwa wie folgt aussehen: Der Parameter `Key` zeigt den Dateinamen Ihrer verkleinerten Bilddatei an.

```
{
    "Contents": [
        {
            "Key": "resized-HappyFace.jpg",
            "LastModified": "2023-06-06T21:40:07+00:00",
            "ETag": "\"d8ca652ffe83ba6b721ffc20d9d7174a\"",
            "Size": 2633,
```

```
            "StorageClass": "STANDARD"
        }
    ]
}
```

Testen Ihrer Funktion mit dem Amazon-S3-Auslöser



Nachdem Sie sich vergewissert haben, dass Ihre Lambda-Funktion ordnungsgemäß funktioniert, können Sie Ihr gesamtes Setup testen, indem Sie Ihrem Amazon-S3-Quell-Bucket eine Bilddatei hinzufügen. Wenn Sie Ihr Bild dem Quell-Bucket hinzufügen, sollte Ihre Lambda-Funktion automatisch aufgerufen werden. Ihre Funktion erstellt eine verkleinerte Version der Datei und speichert sie in Ihrem Ziel-Bucket.

AWS-Managementkonsole

So testen Sie Ihre Lambda-Funktion unter Verwendung des Amazon-S3-Auslösers (Konsole)

1. Gehen Sie wie folgt vor, um ein Bild in Ihren Amazon-S3-Bucket hochzuladen:
   a. Öffnen Sie die Seite Buckets der Amazon-S3-Konsole und wählen Sie Ihren Quell-Bucket aus.
   b. Klicken Sie auf Upload.
   c. Wählen Sie Dateien hinzufügen und anschließend über die Dateiauswahl die Bilddatei aus, die Sie hochladen möchten. Ihr Bildobjekt kann eine beliebige JPG- oder PNG-Datei sein.
   d. Wählen Sie Öffnen und anschließend Hochladen aus.
2. Vergewissern Sie sich, dass Lambda eine verkleinerte Version Ihrer Bilddatei in Ihrem Ziel-Bucket gespeichert hat:

a. Navigieren Sie wieder zur Seite Buckets der Amazon-S3-Konsole und wählen Sie Ihren Ziel-Bucket aus.

b. Im Bereich Objekte sollten jetzt zwei verkleinerte Bilddateien angezeigt werden (jeweils eine aus den beiden Tests Ihrer Lambda-Funktion). Wählen Sie zum Herunterladen Ihres verkleinerten Bilds die Datei und anschließend Herunterladen aus.

AWS CLI

So testen Sie Ihre Lambda-Funktion unter Verwendung des Amazon-S3-Auslösers (AWS CLI)

1. Führen Sie in dem Verzeichnis, das das hochzuladende Bild enthält, den folgenden CLI-Befehl aus. Ersetzen Sie dabei den Parameter `--bucket` durch den Namen Ihres Quell-Buckets. Verwenden Sie für die Parameter `--key` und `--body` den Dateinamen Ihres Testbilds. Ihr Testbild kann eine beliebige JPG- oder PNG-Datei sein.

```
aws s3api put-object --bucket amzn-s3-demo-source-bucket --key SmileyFace.jpg --
body ./SmileyFace.jpg
```

2. Vergewissern Sie sich, dass Ihre Funktion eine Miniaturversion Ihres Bilds erstellt und in Ihrem Amazon-S3-Ziel-Bucket gespeichert hat. Führen Sie den folgenden CLI-Befehl aus und ersetzen Sie dabei `amzn-s3-demo-source-bucket-resized` durch den Namen Ihres eigenen Ziel-Buckets:

```
aws s3api list-objects-v2 --bucket amzn-s3-demo-source-bucket-resized
```

Wenn Ihre Funktion erfolgreich ausgeführt wird, erhalten Sie eine Ausgabe wie die folgende. Ihr Ziel-Bucket sollte jetzt zwei verkleinerte Dateien enthalten.

```
{
    "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"
        }
    ]
}
```

Bereinigen Ihrer Ressourcen

Sie können jetzt die Ressourcen, die Sie für dieses Tutorial erstellt haben, löschen, es sei denn, Sie möchten sie behalten. Durch das Löschen von AWS Ressourcen, die Sie nicht mehr verwenden, vermeiden Sie unnötige Kosten für Ihre AWS-Konto.

So löschen Sie die Lambda-Funktion:

1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.
2. Wählen Sie die Funktion aus, die Sie erstellt haben.
3. Wählen Sie Aktionen, Löschen aus.
4. Geben Sie **`confirm`** in das Texteingabefeld ein und wählen Sie Delete (Löschen) aus.

So löschen Sie die von Ihnen erstellte Richtlinie

1. Öffnen Sie die Seite Richtlinien in der IAM-Konsole.
2. Wählen Sie die Richtlinie aus, die Sie erstellt haben (AWSLambdaS3Policy).
3. Wählen Sie Policy actions (Richtlinienaktionen) und anschließend Delete (Löschen) aus.
4. Wählen Sie Löschen aus.

So löschen Sie die Ausführungsrolle

1. Öffnen Sie die Seite Roles in der IAM-Konsole.
2. Wählen Sie die von Ihnen erstellte Ausführungsrolle aus.
3. Wählen Sie Löschen aus.
4. Geben Sie den Namen der Rolle in das Texteingabefeld ein und wählen Sie Delete (Löschen) aus.

So löschen Sie den S3-Bucket:

1. Öffnen Sie die Amazon S3-Konsole.
2. Wählen Sie den Bucket aus, den Sie erstellt haben.
3. Wählen Sie Löschen aus.
4. Geben Sie den Namen des Buckets in das Texteingabefeld ein.
5. Wählen Sie Bucket löschen aus.

Verwenden von Secrets-Manager-Secrets in Lambda-Funktionen

AWS Secrets Manager hilft Ihnen bei der Verwaltung von Anmeldeinformationen, API-Schlüsseln und anderen Geheimnissen, die Ihre Lambda-Funktionen benötigen. Sie haben zwei Hauptansätze für das Abrufen von Geheimnissen in Ihren Lambda-Funktionen, die beide eine bessere Leistung und geringere Kosten bieten als das direkte Abrufen von Geheimnissen mithilfe des SDK: AWS

- AWS Lambda-Erweiterung für Parameter und Geheimnisse — Eine laufzeitunabhängige Lösung, die eine einfache HTTP-Schnittstelle zum Abrufen von Geheimnissen bietet
- Powertools for AWS Lambda Parameters Utility — Eine codeintegrierte Lösung, die mehrere Anbieter (Secrets Manager, Parameter Store AppConfig) mit integrierten Transformationen unterstützt

Beide Ansätze verwenden lokale Caches für Secrets, sodass Ihre Funktion nicht bei jedem Aufruf den Secrets Manager aufrufen muss. Wenn Ihre Funktion ein Secret anfordert, wird zuerst der Cache überprüft. Ist das Secret verfügbar und nicht abgelaufen, wird es sofort zurückgegeben. Andernfalls wird es aus dem Secrets Manager abgerufen, zwischengespeichert und zurückgegeben. Dieser Caching-Mechanismus führt zu schnelleren Antwortzeiten und geringeren Kosten durch die Minimierung von API-Aufrufen.

Auswahl des Ansatzes

Berücksichtigen Sie diese Faktoren, wenn Sie zwischen der Erweiterung und folgenden Optionen wählen: PowerTools

Verwenden Sie die Lambda-Erweiterung AWS parameters and secrets, wenn:

- Sie eine laufzeitunabhängige Lösung benötigen, die mit jeder Lambda-Laufzeit funktioniert.
- Sie keine Codeabhängigkeiten zu Ihrer Funktion hinzufügen möchten.
- Sie nur Secrets aus Secrets Manager oder Parameter Store abrufen müssen.

Verwenden Sie Powertools for AWS Lambda Parameters Utility, wenn:

- Sie eine integrierte Entwicklungserfahrung mit Ihrem Anwendungscode wünschen.
- Sie benötigen Unterstützung für mehrere Anbieter (Secrets Manager, Parameter Store, AppConfig)
- Sie integrierte Datentransformationen (JSON-Parsing, Base64-Decodierung) benötigen.
- Sie verwenden Python- TypeScript, Java- oder .NET-Laufzeiten

Wann Sie den Secrets Manager mit Lambda verwenden sollten

Zu den häufigsten Anwendungsfällen für den Secrets Manager mit Lambda gehören:

- Speichern von Datenbank-Anmeldeinformationen, die Ihre Funktion für die Verbindung zu Amazon RDS oder anderen Datenbanken verwendet.
- Verwalten von API-Schlüsseln für externe Dienste, die Ihre Funktion aufruft.
- Speichern von Verschlüsselungsschlüsseln oder anderen sensiblen Konfigurationsdaten.
- Automatische Rotation der Anmeldeinformationen, ohne dass Sie Ihren Funktionscode aktualisieren müssen.

Verwenden der Lambda-Erweiterung AWS Parameter und Geheimnisse

Die Lambda-Erweiterung AWS Parameters and Secrets verwendet eine einfache HTTP-Schnittstelle, die mit jeder Lambda-Laufzeit kompatibel ist. Standardmäßig werden Secrets 300 Sekunden (5 Minuten) lang zwischengespeichert und es können bis zu 1 000 Secrets gespeichert werden. Sie können diese Einstellungen mit Umgebungsvariablen anpassen.

Verwendung des Secrets Manager in einer Lambda-Funktion

In diesem Abschnitt wird davon ausgegangen, dass Sie bereits ein Secret im Secrets Manager besitzen. Informationen zum Erstellen eines Secrets finden Sie unter Create an AWS Secrets Manager Secret.

Erstellen des Bereitstellungspakets

Wählen Sie Ihre bevorzugte Laufzeit und folgen Sie den Schritten, um eine Funktion zu erstellen, die Secrets aus dem Secrets Manager abruft. Die Beispielfunktion ruft ein Secrets aus dem Secrets Manager ab und kann verwendet werden, um auf Datenbank-Anmeldeinformationen, API-Schlüssel oder andere sensible Konfigurationsdaten in Ihren Anwendungen zuzugreifen.

Python

So erstellen Sie eine Python-Funktion

1. Erstellen Sie ein neues Projektverzeichnis und navigieren Sie dorthin. Beispiel:

```
mkdir my_function
```

```
cd my_function
```

2. Erstellen Sie eine Datei mit dem Namen `lambda_function.py` und dem folgenden Code. Verwenden Sie für `secret_name` den Namen oder den Amazon-Ressourcennamen (ARN) Ihres Secrets.

```
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. Erstellen Sie eine Datei namens „`requirements.txt`“ mit folgendem Inhalt:

```
requests
```

4. Installieren Sie die Abhängigkeiten:

```
pip install -r requirements.txt -t .
```

5. Erstellen Sie eine ZIP-Datei, die alle Dateien enthält:

```
zip -r function.zip .
```

Node.js

Erstellen einer Funktion Node.js.

1. Erstellen Sie ein neues Projektverzeichnis und navigieren Sie dorthin. Beispiel:

```
mkdir my_function
cd my_function
```

2. Erstellen Sie eine Datei mit dem Namen `index.mjs` und dem folgenden Code. Verwenden Sie für `secret_name` den Namen oder den Amazon-Ressourcennamen (ARN) Ihres Secrets.

```
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. Erstellen Sie eine ZIP-Datei, die die Datei „`index.mjs`“ enthält:

```
zip -r function.zip index.mjs
```

Java

So erstellen Sie eine Java-Funktion

1. Erstellen Sie ein Maven-Projekt:

```
mvn archetype:generate \
    -DgroupId=example \
    -DartifactId=lambda-secrets-demo \
    -DarchetypeArtifactId=maven-archetype-quickstart \
    -DarchetypeVersion=1.4 \
    -DinteractiveMode=false
```

2. Navigieren Sie zum Projektverzeichnis:

```
cd lambda-secrets-demo
```

3. Öffnen Sie die Datei „`pom.xml`“ und ersetzen Sie deren Inhalt durch Folgendes:

```
<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. Benennen Sie die Datei „`/lambda-secrets-demo/src/main/java/example/App.java`“ in „`Hello.java`“ um, um sie an den Standardnamen des Java-Handlers von Lambda (`example.Hello::handleRequest`) anzupassen:

```
mv src/main/java/example/App.java src/main/java/example/Hello.java
```

5. Öffnen Sie die Datei `Hello.java` und ersetzen Sie den Inhalt durch Folgendes. Verwenden Sie für `secretName` den Namen oder den Amazon-Ressourcennamen (ARN) Ihres Secrets.

```
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. Entfernen Sie das Testverzeichnis. Maven erstellt dies standardmäßig, aber wir benötigen es für dieses Beispiel nicht.

```
rm -rf src/test
```

7. Erstellen Sie das Projekt:

```
mvn package
```

8. Laden Sie die JAR-Datei (`target/function.jar`) zur späteren Verwendung herunter.

Erstellen der Funktion

1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.
2. Wählen Sie Funktion erstellen.
3. Wählen Sie Verfassen von Grund auf aus.
4. Geben Sie für Function name (Funktionsname) **`secret-retrieval-demo`** ein.
5. Wählen Sie Ihre bevorzugte Laufzeit.
6. Wählen Sie Funktion erstellen.

So laden Sie das Bereitstellungspaket hoch

1. Wählen Sie auf der Registerkarte Code der Funktion die Option Hochladen von und dann ZIP-Datei (für Python und Node.js) oder JAR-Datei (für Java) aus.
2. Laden Sie das Bereitstellungspaket hoch, das Sie zuvor erstellt haben.
3. Wählen Sie Speichern.

Hinzufügen der Erweiterung

Um die Lambda-Erweiterung AWS Parameters and Secrets als Ebene hinzuzufügen

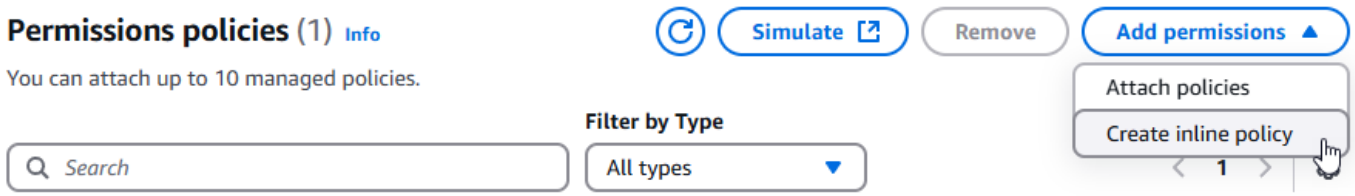
1. Scrollen Sie auf der Registerkarte Code der Funktion nach unten zu Ebenen.
2. Wählen Sie Add a layer (Layer hinzufügen) aus.
3. Wählen Sie AWS -Ebenen aus.
4. Wählen Sie AWS-Parameters-and-Secrets-Lambda-Extension aus.
5. Wählen Sie die neueste Version aus.
6. Wählen Sie Hinzufügen aus.

Berechtigungen hinzufügen

So fügen Sie Ihrer Ausführungsrolle Berechtigungen für den Secrets Manager hinzu

1. Wählen Sie die Registerkarte Konfiguration und dann Berechtigungen aus.
2. Wählen Sie unter Rollenname den Link zu Ihrer Ausführungsrolle. Dieser Link öffnet die Rolle in der IAM-Konsole.

3. Wählen Sie Add permissions (Berechtigungen hinzufügen) und dann Create inline policy (Inline-Richtlinie erstellen) aus.



4. Wählen Sie die Registerkarte JSON aus und fügen Sie die folgende Richtlinie hinzu. Geben Sie für `Resource` den ARN Ihres Secrets ein.

   JSON

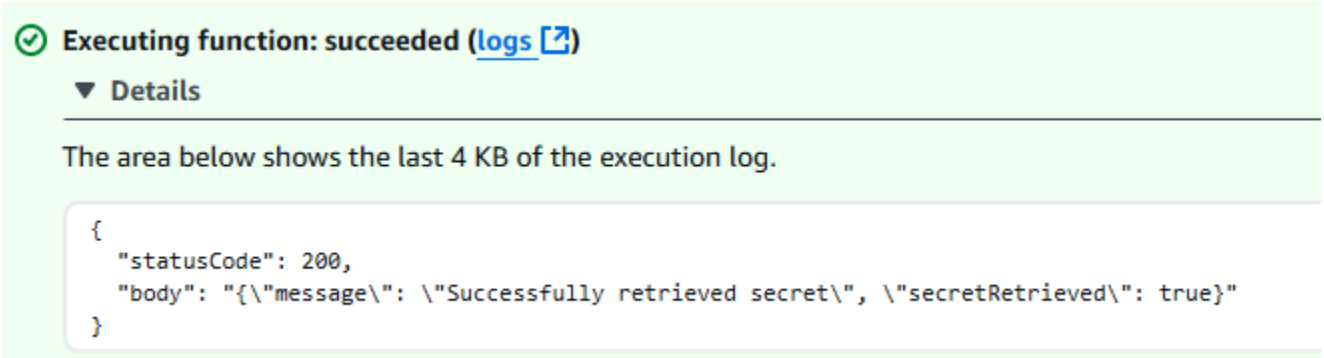
```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "secretsmanager:GetSecretValue",
            "Resource": "arn:aws:secretsmanager:us-
east-1:111122223333:secret:SECRET_NAME"
        }
    ]
}
```

5. Wählen Sie Weiter aus.
6. Geben Sie einen Namen für die Richtlinie ein.
7. Wählen Sie Richtlinie erstellen aus.

Testen der Funktion

So testen Sie die -Funktion

1. Kehren Sie zur Lambda-Konsole zurück.
2. Wählen Sie die Registerkarte Test.
3. Wählen Sie Test aus. Sie sollten die folgende Antwort sehen:



Umgebungsvariablen

Die Lambda-Erweiterung AWS Parameters and Secrets verwendet die folgenden Standardeinstellungen. Sie können diese Einstellungen überschreiben, indem Sie die entsprechenden Umgebungsvariablen erstellen. Setzen Sie `PARAMETERS_SECRETS_EXTENSION_LOG_LEVEL` auf `DEBUG`, um die aktuellen Einstellungen für eine Funktion anzuzeigen. Die Erweiterung CloudWatch protokolliert ihre Konfigurationsinformationen zu Beginn jedes Funktionsaufrufs in Logs.

Einstellung	Standardwert	Zulässige Werte	Umgebungsvariable	Details
HTTP-Port	2773	1–65 535	PARAMETERS_SECRETS_EXTENSION_HTTP_PORT	Port für den lokalen HTTP-Server
Cache aktiviert	TRUE	TRUE | FALSE	PARAMETERS_SECRETS_EXTENSION_CACHE_ENABLED	Aktivieren oder Deaktivieren des Cache

Einstellung	Standardwert	Zulässige Werte	Umgebungsvariable	Details
Cache-Größe	1000	0–1 000	PARAMETERS_SECRETS_EXTENSION_CACHE_SIZE	Auf 0 setzen, um das Caching zu deaktivieren
Secrets Manager TTL	300 Sekunden	0–300 Sekunde	SECRETS_MANAGER_TTL	Time-to-live für zwischengespeicherte Geheimnisse. Auf 0 setzen, um das Caching zu deaktivieren. Diese Variable wird ignoriert, wenn der Wert für PARAMETERS_SECRETS_EXTENSION_CACHE_SIZE 0 ist.
Parameter Store TTL	300 Sekunden	0–300 Sekunde	SSM_PARAMETER_STORE_TTL	Time-to-live für zwischengespeicherte Parameter. Auf 0 setzen, um das Caching zu deaktivieren. Diese Variable wird ignoriert, wenn der Wert für PARAMETERS_SECRETS_EXTENSION_CACHE_SIZE 0 ist.
Protokollebene	INFO	DEBUG | INFO | WARN | ERROR | NONE	PARAMETERS_SECRETS_EXTENSION_LOG_LEVEL	Der Detaillierungsgrad, der in Protokollen für die Erweiterung gemeldet wird

Einstellung	Standardwert	Zulässige Werte	Umgebungsvariable	Details
Maximale Verbindungen	3	1 oder höher	PARAMETERS_SECRETS_EXTENSION_MAX_CONNECTIONS	Maximale Anzahl von HTTP-Verbindungen für Anforderungen an den Parameter Store oder Secrets Manager
Timeout für Secrets Manager	0 (kein Timeout)	Alle ganzen Zahlen	SECRETS_MANAGER_TIMEOUT_MILLIS	Timeout für Anfragen an Secrets Manager (in Millisekunden)
Timeout für Parameter Store	0 (kein Timeout)	Alle ganzen Zahlen	SSM_PARAMETER_STORE_TIMEOUT_MILLIS	Timeout für Anfragen an den Parameter Store (in Millisekunden)

Arbeiten mit Secret-Rotation

Wenn Sie Secrets häufig rotieren, kann die standardmäßige Cachedauer von 300 Sekunden dazu führen, dass Ihre Funktion veraltete Secrets verwendet. Sie haben zwei Möglichkeiten, um sicherzustellen, dass Ihre Funktion den neuesten Secret-Wert verwendet:

- Reduzieren Sie die Cache-TTL, indem Sie die `SECRETS_MANAGER_TTL`-Umgebungsvariable auf einen niedrigeren Wert (in Sekunden) setzen. Wenn Sie sie beispielsweise auf `60` einstellen, wird sichergestellt, dass Ihre Funktion niemals ein Secret verwendet, das älter als eine Minute ist.
- Verwenden Sie die Staging-Bezeichnungen `AWSCURRENT` oder `AWSPREVIOUS` in Ihrer Secret-Anfrage, um sicherzustellen, dass Sie die gewünschte Version erhalten:

```
secretsmanager/get?secretId=YOUR_SECRET_NAME&versionStage=AWSCURRENT
```

Wählen Sie den Ansatz, der Ihren Anforderungen an Leistung und Aktualität am besten gerecht wird. Eine niedrigere TTL bedeutet häufigere Aufrufe von Secrets Manager, stellt jedoch sicher, dass Sie mit den aktuellsten Secret-Werten arbeiten.

Verwenden Sie das Parameter-Hilfsprogramm von Powertools für AWS Lambda

Das Parameter-Hilfsprogramm von Powertools for AWS Lambda bietet eine einheitliche Oberfläche zum Abrufen von Geheimnissen von mehreren Anbietern, darunter Secrets Manager, Parameterspeicher und. AppConfig Es übernimmt Caching und Transformationen und bietet im Vergleich zum Erweiterungsansatz eine integriertere Entwicklungserfahrung.

Vorteile des Parameter-Dienstprogramms

- Mehrere Anbieter — Rufen Sie Parameter aus Secrets Manager und Parameter Store ab und AppConfig verwenden Sie dieselbe Schnittstelle
- Integrierte Transformationen – Automatisches JSON-Parsing, Base64-Decodierung und weitere Datentransformationen
- Integriertes Caching – Konfigurierbares Caching mit TTL-Unterstützung zur Reduzierung von API-Aufrufen
- Typsicherheit — Starke Tippunterstützung in TypeScript und anderen unterstützten Laufzeiten
- Fehlerbehandlung – Integrierte Wiederholungslogik und Fehlerbehandlung

Codebeispiele

Die folgenden Beispiele zeigen, wie Secrets mit dem Parameter-Dienstprogramm in verschiedenen Laufzeiten abgerufen werden:

Python

> **Note**
>
> Vollständige Beispiele und Anweisungen zur Einrichtung finden Sie in der Dokumentation zum Parameter-Dienstprogramm.

Abrufen von Geheimnissen aus Secrets Manager mit dem Hilfsprogramm 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**
>
> Vollständige Beispiele und Anweisungen zur Einrichtung finden Sie in der Dokumentation zum Parameter-Dienstprogramm.

Abrufen von Geheimnissen aus Secrets Manager mit dem Hilfsprogramm 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**
>
> Vollständige Beispiele und Anweisungen zur Einrichtung finden Sie in der Dokumentation zum Parameter-Dienstprogramm.

Abrufen von Geheimnissen aus Secrets Manager mit dem Hilfsprogramm 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**
>
> Vollständige Beispiele und Anweisungen zur Einrichtung finden Sie in der Dokumentation zum Parameter-Dienstprogramm.

Abrufen von Geheimnissen aus Secrets Manager mit dem Hilfsprogramm 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
    }
}
```

Einrichtung und Berechtigungen

Um das Parameter-Dienstprogramm zu verwenden, müssen Sie Folgendes tun:

1. Installieren Sie Powertools für AWS Lambda für Ihre Runtime. Details hierzu finden Sie unter Powertools for AWS Lambda.
2. Fügen Sie der Ausführungsrolle Ihrer Funktion die erforderlichen IAM-Berechtigungen hinzu. Weitere Einzelheiten finden Sie unter Verwaltung von Berechtigungen in AWS Lambda.
3. Konfigurieren Sie optionale Einstellungen über Umgebungsvariablen.

Die erforderlichen IAM-Berechtigungen sind dieselben wie beim Erweiterungsansatz. Das Dienstprogramm übernimmt automatisch das Caching und die API-Aufrufe an den Secrets Manager basierend auf Ihrer Konfiguration.

Verwenden von Lambda mit Amazon SQS

> **Note**
>
> Wenn Sie Daten an ein anderes Ziel als eine Lambda-Funktion senden oder die Daten vor dem Senden anreichern möchten, finden Sie weitere Informationen unter Amazon EventBridge Pipes.

Mit einer Lambda-Funktion können Sie Nachrichten in einer Amazon-SQS-Warteschlange (Amazon Simple Queue Service) verarbeiten. Lambda unterstützt sowohl Standard-Warteschlangen als auch First-In-First-Out-Warteschlangen (FIFO) für Zuordnungen von Ereignisquellen. Sie können den Bereitstellungsmodus auch verwenden, um spezielle Abfrageressourcen für Ihre Amazon SQS SQS-Ereignisquellenzuordnungen zuzuweisen. Die Lambda-Funktion und die Amazon SQS SQS-Warteschlange müssen sich in derselben befinden AWS-Region, obwohl sie sich unterscheiden können. AWS-Konten

Bei der Verarbeitung von Amazon-SQS-Nachrichten müssen Sie eine Logik für teilweise Batch-Antworten implementieren, um zu verhindern, dass versucht wird, erfolgreich verarbeitete Nachrichten erneut zu verarbeiten, wenn einige Nachrichten in einem Batch fehlschlagen. Das Batch Processor Utility von Powertools for AWS Lambda vereinfacht diese Implementierung, indem es automatisch die Logik der partiellen Batch-Antwort verarbeitet, wodurch die Entwicklungszeit reduziert und die Zuverlässigkeit verbessert wird.

Themen

- Verständnis des Polling- und Batching-Verhaltens für Amazon-SQS-Zuordnungen von Ereignisquellen
- Verwenden des Bereitstellungsmodus mit Amazon SQS SQS-Ereignisquellenzuordnungen
- Konfiguration des Bereitstellungsmodus für die Zuordnung von Amazon SQS SQS-Ereignisquellen
- Beispiel für ein Standard-Warteschlangen-Nachrichtenereignis
- Beispiel für ein FIFO-Warteschlangen-Nachrichtenereignis
- Erstellen und Konfigurieren einer Amazon SQS-Zuordnung von Ereignisquellen
- Konfiguration des Skalierungsverhaltens für SQS-Zuordnungen von Ereignisquellen
- Behandlung von Fehlern für eine SQS-Ereignisquelle in Lambda
- Lambdaparameter für Amazon-SQS-Zuordnungen von Ereignisquellen

- Verwendung der Ereignisfilterung mit einer Amazon-SQS-Ereignisquelle
- Tutorial: Verwenden von Lambda mit Amazon SQS
- Tutorial: Verwenden einer kontoübergreifenden Amazon SQS SQS-Warteschlange als Ereignisquelle

Verständnis des Polling- und Batching-Verhaltens für Amazon-SQS-Zuordnungen von Ereignisquellen

Mit Amazon-SQS-Zuordnungen von Ereignisquellen fragt Lambda die Warteschlange ab und ruft Ihre Funktion synchron mit einem Ereignis auf. Jedes Ereignis kann einen Batch von mehreren Nachrichten aus der Warteschlange enthalten. Lambda empfängt diese Ereignisse jeweils stapelweise und ruft Ihre Funktion für jeden Stapel einmal auf. Wenn Ihre Funktion einen Batch erfolgreich verarbeitet, löscht Lambda deren Nachrichten aus der Warteschlange.

Wenn Lambda einen Batch empfängt, bleiben die Nachrichten in der Warteschlange, werden aber für die Dauer der Zeitbeschränkung für die Sichtbarkeit der Warteschlange ausgeblendet. Wenn Ihre Funktion alle Nachrichten im Batch erfolgreich verarbeitet hat, löscht Lambda die Nachrichten aus der Warteschlange. Wenn Ihre Funktion bei der Verarbeitung eines Batches auf einen Fehler stößt, werden standardmäßig alle Nachrichten in diesem Batch nach Ablauf des Zeitlimits für die Sichtbarkeit wieder in der Warteschlange sichtbar. Deshalb muss der Funktionscode in der Lage sein, dieselbe Nachricht mehrmals ohne unbeabsichtigte Begleiterscheinungen zu verarbeiten.

> **Warning**
>
> Zuordnung von Lambda-Ereignisquellen verarbeiten jedes Ereignis mindestens einmal und es kann zu einer doppelten Verarbeitung von Datensätzen kommen. Um mögliche Probleme im Zusammenhang mit doppelten Ereignissen zu vermeiden, empfehlen wir Ihnen dringend, Ihren Funktionscode idempotent zu machen. Weitere Informationen finden Sie im Knowledge Center unter Wie mache ich meine Lambda-Funktion idempotent? AWS

Um zu verhindern, dass Lambda eine Nachricht mehrfach verarbeitet, können Sie entweder Ihre Ereignisquellenzuordnung so konfigurieren, dass Batch-Elementfehler in Ihre Funktionsantwort aufgenommen werden, oder Sie können die DeleteMessageAPI verwenden, um Nachrichten aus der Warteschlange zu entfernen, wenn Ihre Lambda-Funktion sie erfolgreich verarbeitet.

Weitere Informationen zu Konfigurationsparametern, die Lambda für SQS-Zuordnungen von Ereignisquellen unterstützt, finden Sie unter the section called “Erstellen einer SQS-Zuordnung von Ereignisquellen”.

Verwenden des Bereitstellungsmodus mit Amazon SQS SQS-Ereignisquellenzuordnungen

Für Workloads, bei denen Sie den Durchsatz Ihrer Zuordnung von Ereignisquellen optimieren müssen, können Sie den Bereitstellungsmodus verwenden. Im Bereitstellungsmodus definieren Sie Mindest- und Höchstgrenzen für die Anzahl der bereitgestellten Ereignis-Poller. Diese bereitgestellten Event-Poller sind auf Ihre Zuordnung von Ereignisquellen ausgerichtet und können unerwartete Nachrichtenspitzen durch reaktionsschnelle Autoskalierung bewältigen. Die mit dem Bereitstellungsmodus konfigurierte Amazon SQS SQS-Ereignisquellenzuweisung skaliert dreimal schneller (bis zu 1.000 gleichzeitige Aufrufe pro Minute) und unterstützt eine 16-mal höhere Parallelität (bis zu 20.000 gleichzeitige Aufrufe) als die standardmäßige Amazon SQS SQS-Funktion zur Zuordnung von Ereignisquellen. Wir empfehlen, den Bereitstellungsmodus für ereignisgesteuerte Amazon SQS SQS-Workloads zu verwenden, für die strenge Leistungsanforderungen gelten, z. B. Finanzdienstleister, die Marktdatenfeeds verarbeiten, E-Commerce-Plattformen, die personalisierte Empfehlungen in Echtzeit bereitstellen, und Spieleunternehmen, die Live-Spielerinteraktionen verwalten. Die Verwendung des Bereitstellungsmodus verursacht zusätzliche Kosten. Eine detaillierte Preisgestaltung finden Sie unter Preise.AWS Lambda

Jeder Event Poller im Bereitstellungsmodus kann bis zu 1 MB/s Durchsatz, bis zu 10 gleichzeitige Aufrufe oder bis zu 10 Amazon SQS-Polling-API-Aufrufe pro Sekunde verarbeiten. Der Bereich der akzeptierten Werte für die Mindestanzahl von Event-Pollern (MinimumPollers) liegt zwischen 2 und 200, der Standardwert ist 2. Der Bereich der akzeptierten Werte für die maximale Anzahl von Event-Pollern (MaximumPollers) liegt zwischen 2 und 2.000, der Standardwert ist 200. MaximumPollers muss größer oder gleich sein MinimumPollers.

Ermittlung der erforderlichen Event-Poller

Um die Anzahl der Ereignisabfragen zu schätzen, die erforderlich sind, um eine optimale Nachrichtenverarbeitungsleistung sicherzustellen, wenn Sie den Bereitstellungsmodus für SQS ESM verwenden, erfassen Sie die folgenden Metriken für Ihre Anwendung: maximale SQS-Ereignisse pro Sekunde, die eine Verarbeitung mit niedriger Latenz erfordern, durchschnittliche Größe der SQS-Ereignisnutzlast, durchschnittliche Lambda-Funktionsdauer und konfigurierte Batchgröße.

Zunächst können Sie anhand der folgenden Formel die Anzahl der SQS-Ereignisse pro Sekunde (EPS) schätzen, die von einem Event Poller für Ihren Workload unterstützt werden:

```
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)
                )
```

Anschließend können Sie anhand der folgenden Formel die Anzahl der mindestens erforderlichen Poller berechnen. Diese Berechnung stellt sicher, dass Sie ausreichend Kapazität bereitstellen, um Ihren Spitzenverkehrsanforderungen gerecht zu werden.

```
Required event pollers = (Peak number of events per second in Queue) / EPS per event
 poller
```

Stellen Sie sich einen Workload mit einer Standard-Batchgröße von 10, einer durchschnittlichen Ereignisgröße von 3 KB, einer durchschnittlichen Funktionsdauer von 100 ms und der Anforderung, 1.000 Ereignisse pro Sekunde zu verarbeiten, vor. In diesem Szenario unterstützt jeder Event Poller ungefähr 100 Ereignisse pro Sekunde (EPS). Daher sollten Sie die Mindestanzahl an Pollern auf 10 festlegen, um Ihren Anforderungen an den Spitzenverkehr angemessen gerecht zu werden. Wenn Ihr Workload dieselben Eigenschaften hat, aber eine durchschnittliche Funktionsdauer von 1 Sekunde hat, unterstützt jeder Poller nur 10 EPS. Sie müssen also mindestens 100 Poller konfigurieren, um 1.000 Ereignisse pro Sekunde bei niedriger Latenz zu unterstützen.

Wir empfehlen die Verwendung einer Standard-Batchgröße von 10 oder höher, um die Effizienz von Event-Pollern im Bereitstellungsmodus zu maximieren. Höhere Batchgrößen ermöglichen es jedem Poller, mehr Ereignisse pro Aufruf zu verarbeiten, was den Durchsatz und die Kosteneffizienz verbessert. Berücksichtigen Sie bei der Planung Ihrer Event-Poller-Kapazität potenzielle Verkehrsspitzen und ziehen Sie in Betracht, den MinimumPollers-Wert etwas höher als das berechnete Minimum festzulegen, um einen Puffer bereitzustellen. Überwachen Sie außerdem Ihre Workload-Merkmale im Zeitverlauf, da Änderungen der Nachrichtengröße, der Funktionsdauer oder der Verkehrsmuster möglicherweise Anpassungen an Ihrer Event-Poller-Konfiguration erforderlich machen, um eine optimale Leistung und Kosteneffizienz zu gewährleisten. Für eine genaue Kapazitätsplanung empfehlen wir, Ihre spezifische Arbeitslast zu testen, um den tatsächlichen Gewinn pro Leistung zu ermitteln, den jeder Event Poller erzielen kann.

Konfiguration des Bereitstellungsmodus für die Zuordnung von Amazon SQS SQS-Ereignisquellen

Sie können den Bereitstellungsmodus für Ihre Amazon SQS SQS-Ereignisquellenzuordnung mithilfe der Konsole oder der Lambda-API konfigurieren.

So konfigurieren Sie den Bereitstellungsmodus für eine bestehende Amazon SQS SQS-Ereignisquellenzuordnung (Konsole)

1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.
2. Wählen Sie die Funktion mit der Amazon SQS SQS-Ereignisquellenzuordnung aus, für die Sie den Bereitstellungsmodus konfigurieren möchten.
3. Wählen Sie Konfiguration und anschließend Auslöser aus.
4. Wählen Sie die Amazon SQS SQS-Ereignisquellenzuordnung aus, für die Sie den Bereitstellungsmodus konfigurieren möchten, und klicken Sie dann auf Bearbeiten.
5. Wählen Sie unter Konfiguration der Zuordnung von Ereignisquellen die Option Bereitstellungsmodus konfigurieren aus.
   - Geben Sie für Minimum Event Pollers einen Wert zwischen 2 und 200 ein. Wenn Sie keinen Wert angeben, wählt Lambda den Standardwert 2.
   - Geben Sie für Maximum Event Pollers einen Wert zwischen 2 und 2.000 ein. Dieser Wert muss größer oder gleich Ihrem Wert für Minimum Event Pollers sein. Wenn Sie keinen Wert angeben, wählt Lambda einen Standardwert von 200.
6. Wählen Sie Speichern.

Sie können den Bereitstellungsmodus programmgesteuert mithilfe des `ProvisionedPollerConfig` Objekts in Ihrem konfigurieren. `EventSourceMappingConfiguration` Der folgende `UpdateEventSourceMapping` CLI-Befehl konfiguriert beispielsweise einen `MinimumPollers` Wert von 5 und einen `MaximumPollers` Wert von 100.

```
aws lambda update-event-source-mapping \
    --uuid a1b2c3d4-5678-90ab-cdef-EXAMPLE11111 \
    --provisioned-poller-config '{"MinimumPollers": 5, "MaximumPollers": 100}'
```

Nachdem Sie den Bereitstellungsmodus konfiguriert haben, können Sie die Verwendung von Event-Pollern für Ihren Workload beobachten, indem Sie die `ProvisionedPollers`-Metrik überwachen. Weitere Informationen finden Sie unter Metriken zur Zuordnung von Ereignisquellen.

Um den Bereitstellungsmodus zu deaktivieren und zum Standardmodus (auf Abruf) zurückzukehren, können Sie den folgenden `UpdateEventSourceMapping` CLI-Befehl verwenden:

```
aws lambda update-event-source-mapping \
    --uuid a1b2c3d4-5678-90ab-cdef-EXAMPLE11111 \
    --provisioned-poller-config '{}'
```

> **Note**
>
> Der Bereitstellungsmodus kann nicht in Verbindung mit der Einstellung für maximale Parallelität verwendet werden. Wenn Sie den Bereitstellungsmodus verwenden, steuern Sie die maximale Parallelität durch die maximale Anzahl von Event-Pollern.

Weitere Informationen zur Konfiguration des Bereitstellungsmodus finden Sie unter. the section called “Zuordnung erstellen”

Beispiel für ein Standard-Warteschlangen-Nachrichtenereignis

Example Amazon-SQS-Nachrichtenereignis (Standardwarteschlange)

```
{
    "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"
        }
    ]
}
```

Lambda fragt standardmäßig bis zu 10 Nachrichten in Ihrer Warteschlange sofort ab und sendet diesen Batch an die Funktion. Damit die Funktion nicht mit einer kleinen Anzahl von Datensätzen aufgerufen wird, können Sie die Ereignisquelle so konfigurieren, dass die Datensätze bis zu 5 Minuten lang gepuffert werden, indem Sie ein Batch-Fenster konfigurieren. Vor dem Aufrufen der Funktion fragt Lambda weiter Nachrichten aus der Standardwarteschlange ab, bis das Batch-Fenster abläuft, das Nutzlastkontingent pro Aufruf erreicht ist oder die konfigurierte maximale Batch-Größe erreicht ist.

Wenn Sie ein Batch-Fenster verwenden und Ihre SQS-Warteschlange sehr wenig Datenverkehr enthält, wartet Lambda möglicherweise bis zu 20 Sekunden, bevor Sie Ihre Funktion aufruft. Dies gilt auch, wenn Sie ein Batch-Fenster unter 20 Sekunden festlegen.

> **Note**
>
> In Java können beim Deserialisieren von JSON Nullzeigerfehler auftreten. Dies könnte daran liegen, dass die Groß- und Kleinschreibung von „Records“ und „eventSourceARN“ vom JSON-Objektmapper konvertiert wird.

Beispiel für ein FIFO-Warteschlangen-Nachrichtenereignis

Bei FIFO-Warteschlangen enthalten Datensätze zusätzliche Attribute, die mit der Deduplizierung und Sequenzierung zusammenhängen.

Example Amazon-SQS-Nachrichtenereignis (FIFO-Warteschlange)

```
{
    "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"
        }
    ]
}
```

Erstellen und Konfigurieren einer Amazon SQS-Zuordnung von Ereignisquellen

Um Amazon-SQS-Nachrichten mit Lambda zu verarbeiten, konfigurieren Sie Ihre Warteschlange mit den entsprechenden Einstellungen und erstellen Sie dann eine Lambda-Zuordnung von Ereignisquellen.

Themen

- Konfigurieren einer Warteschlange zur Verwendung mit Lambda
- Einrichten von Lambda-Ausführungsrollenberechtigungen
- Erstellen einer SQS-Zuordnung von Ereignisquellen

Konfigurieren einer Warteschlange zur Verwendung mit Lambda

Wenn Sie noch nicht über eine Amazon-SQS-Warteschlange verfügen, erstellen Sie eine als Ereignisquelle für Ihre Lambda-Funktion. Die Lambda-Funktion und die Amazon SQS SQS-Warteschlange müssen sich in derselben befinden AWS-Region, obwohl sie sich unterscheiden können. AWS-Konten

Damit Ihre Funktion genügend Zeit hat, um jeden Stapel von Datensätzen zu verarbeiten, setzen Sie die Zeitüberschreitung für die Sichtbarkeit der Ausgangswarteschlange auf mindestens das Sechsfache der Konfigurationszeitüberschreitung Ihrer Funktion. Diese zusätzliche Zeit ermöglicht es für Lambda, einen erneuten Versuch zu machen, wenn die Funktion gedrosselt wird, während ein früherer Batch verarbeitet wird.

> **Note**
>
> Das Timeout Ihrer Funktion muss kleiner oder gleich dem Sichtbarkeits-Timeout der Warteschlange sein. Lambda validiert diese Anforderung, wenn Sie eine Zuordnung von Ereignisquellen erstellen oder aktualisieren, und gibt einen Fehler zurück, wenn das Funktions-Timeout das Sichtbarkeits-Timeout der Warteschlange überschreitet.

Wenn Lambda bei der Verarbeitung eines Stapels auf einen Fehler stößt, kehren standardmäßig alle Nachrichten in diesem Stapel in die Warteschlange zurück. Nach dem Sichtbarkeits-Timeout werden die Nachrichten wieder für Lambda sichtbar. Sie können Ihre Zuordnung von Ereignisquellen so konfigurieren, dass Sie partielle Batch-Antworten verwenden, um nur die fehlgeschlagenen

Nachrichten an die Warteschlange zurückzugeben. Wenn Ihre Funktion eine Nachricht mehrmals nicht verarbeiten kann, kann Amazon SQS sie außerdem an eine Warteschlange für unzustellbare Nachrichten senden. Wir empfehlen, das `maxReceiveCount` in der Redrive-Richtlinie Ihrer Quellwarteschlange auf mindestens 5 zu setzen. Dadurch erhält Lambda einige Chancen, es erneut zu versuchen, bevor fehlgeschlagene Nachrichten direkt an die Warteschlange für unzustellbare Nachrichten gesendet werden.

Einrichten von Lambda-Ausführungsrollenberechtigungen

Die AWSLambdaSQSQueueExecutionRole AWS verwaltete Richtlinie umfasst die Berechtigungen, die Lambda benötigt, um aus Ihrer Amazon SQS SQS-Warteschlange zu lesen. Sie können diese verwaltete Richtlinie der Ausführungsrolle Ihrer Funktion hinzufügen.

Wenn Sie optional eine verschlüsselte Warteschlange verwenden, müssen Sie Ihrer Ausführungsrolle auch die folgende Berechtigung hinzufügen:

- kms:Decrypt

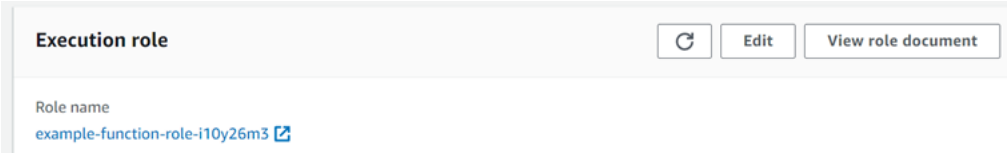
Erstellen einer SQS-Zuordnung von Ereignisquellen

Erstellen Sie ein Ereignisquellen-Mapping, um Lambda anzuweisen, Elemente aus Ihrer Warteschlange an eine Lambda-Funktion zu senden. Sie können mehrere Ereignisquellen-Zuweisungen zum Verarbeiten von Elementen aus mehreren Warteschlangen mit nur einer Funktion erstellen. Wenn Lambda die Zielfunktion aufruft, kann das Ereignis mehrere Elemente bis zu einer konfigurierbaren maximalen Batch-Größe enthalten.

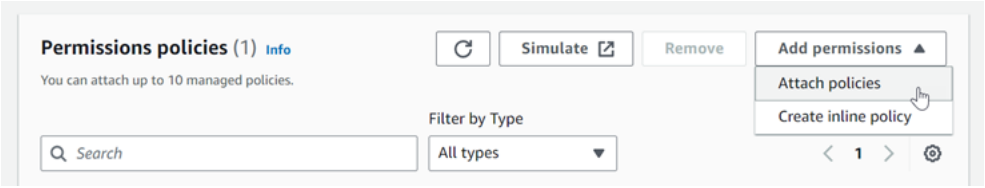
Um Ihre Funktion so zu konfigurieren, dass sie aus Amazon SQS liest, fügen Sie die AWSLambdaSQSQueueExecutionRole AWS verwaltete Richtlinie Ihrer Ausführungsrolle hinzu. Erstellen Sie anschließend mithilfe der folgenden Schritte eine Zuordnung der SQS-Ereignisquellen von der Konsole aus.

So fügen Sie Berechtigungen hinzu und erstellen einen Auslöser

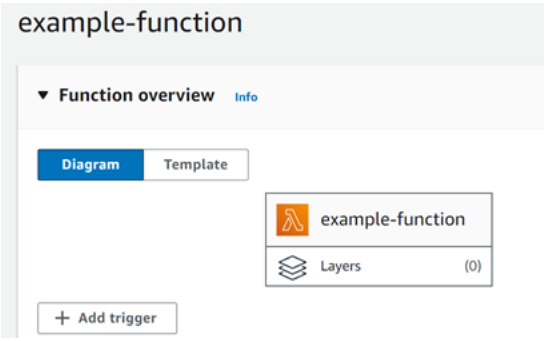
1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.
2. Wählen Sie den Namen einer Funktion aus.
3. Wählen Sie die Registerkarte Konfiguration und dann Berechtigungen aus.
4. Wählen Sie unter Rollenname den Link zu Ihrer Ausführungsrolle. Dieser Link öffnet die Rolle in der IAM-Konsole.



5. Wählen Sie Berechtigungen hinzufügen aus und wählen Sie dann Richtlinien direkt anhängen aus.



6. Geben Sie im Suchfeld `AWSLambdaSQSQueueExecutionRole` ein. Fügen Sie diese Richtlinie zu Ihrer Ausführungsrolle hinzu. Dies ist eine AWS verwaltete Richtlinie, die die Berechtigungen enthält, die Ihre Funktion zum Lesen aus einer Amazon SQS SQS-Warteschlange benötigt. Weitere Informationen zu dieser Richtlinie finden Sie AWSLambdaSQSQueueExecutionRolein der Referenz zu AWS verwalteten Richtlinien.
7. Gehen Sie zurück zu Ihrer Funktion in der Lambda-Konsole. Wählen Sie unter Function overview (Funktionsübersicht) die Option Add trigger (Trigger hinzufügen).



8. Wählen Sie einen Auslösertyp aus.
9. Konfigurieren Sie die erforderlichen Optionen und wählen Sie dann Add (Hinzufügen) aus.

Lambda unterstützt die folgenden Konfigurationsoptionen für Amazon-SQS-Ereignisquellen:

SQS Queue

Die Amazon-SQS-Warteschlange, aus der Datensätze gelesen werden sollen. Die Lambda-Funktion und die Amazon SQS SQS-Warteschlange müssen sich in derselben befinden AWS-Region, obwohl sie sich unterscheiden können. AWS-Konten

Aktivieren von Auslösern

Der Status der Zuordnung von Ereignisquellen. Enable trigger (Auslöser aktivieren) ist standardmäßig ausgewählt.

Batch-Größe

Die maximale Anzahl der Datensätze, die in jedem Stapel an die Funktion gesendet werden sollen. Bei einer Standardwarteschlange können dies bis zu 10.000 Datensätze sein. Bei einer FIFO-Warteschlange liegt der Höchstwert bei 10. Bei einer Stapelgröße über 10 müssen Sie das Stapelfenster (`MaximumBatchingWindowInSeconds`) zusätzlich auf mindestens 1 Sekunde festlegen.

Konfigurieren Sie Ihr Funktions-Timeout so, dass genügend Zeit bleibt, um einen ganzen Stapel von Elementen zu verarbeiten. Wenn die Verarbeitung bestimmter Elemente lange Zeit in Anspruch nimmt, wählen Sie eine kleinere Batch-Größe. Eine große Batchgröße kann die Effizienz für Workloads erhöhen, die sehr schnell sind oder viel Overhead aufweisen. Wenn Sie reservierte Nebenläufigkeit für Ihre Funktion konfigurieren, legen Sie mindestens fünf gleichzeitige Ausführungen fest, um das Risiko von Drosselungsfehlern zu senken, wenn Lambda die Funktion aufruft.

Lambda übergibt alle Datensätze im Batch in einem einzigen Aufruf an die Funktion, solange die Gesamtgröße der Ereignisse nicht das Nutzlastkontingent pro Aufruf für synchrone Aufrufe überschreitet (6 MB). Sowohl Lambda als auch Amazon SQS generieren Metadaten für jeden Datensatz. Diese zusätzlichen Metadaten werden auf die Gesamtnutzlastgröße angerechnet und können dazu führen, dass die Gesamtzahl der in einem Batch gesendeten Datensätze niedriger ist als die konfigurierte Batch-Größe. Die Metadatenfelder, die Amazon SQS sendet, können in der Länge variabel sein. Weitere Informationen zu den Amazon SQS-Metadatenfeldern finden Sie in der Dokumentation zum ReceiveMessageAPI-Betrieb in der Amazon Simple Queue Service API-Referenz.

Batchfenster

Die maximale Zeitspanne zur Erfassung von Datensätzen vor dem Aufruf der Funktion in Sekunden. Dies gilt nur für Standardwarteschlangen.

Wenn Sie ein Stapelfenster von mehr als 0 Sekunden verwenden, müssen Sie die erhöhte Verarbeitungszeit im Sichtbarkeits-Timeout Ihrer Warteschlange berücksichtigen. Wir empfehlen, das Sichtbarkeits-Timeout Ihrer Warteschlange auf das Sechsfache Ihres Funktions-Timeouts plus den Wert von `MaximumBatchingWindowInSeconds` festzulegen. Dies gibt Ihrer Lambda-

Funktion Zeit, jeden Ereignis-Batch zu verarbeiten und es im Falle eines Drosselungsfehlers erneut zu versuchen.

Wenn Nachrichten verfügbar werden, beginnt Lambda, Nachrichten stapelweise zu verarbeiten. Lambda beginnt mit der gleichzeitigen Verarbeitung von fünf Stapeln mit fünf gleichzeitigen Aufrufen Ihrer Funktion. Wenn Nachrichten noch verfügbar sind, addiert Lambda bis zu 300 gleichzeitige Aufrufe Ihrer Funktion pro Minute, bis zu einem Maximum von 1.250 gleichzeitigen Aufrufen. Im Bereitstellungsmodus kann jeder Event Poller bis zu 1 MB/s Durchsatz, bis zu 10 gleichzeitige Aufrufe oder bis zu 10 Amazon SQS-Polling-API-Aufrufe pro Sekunde verarbeiten. Lambda skaliert die Anzahl der Event-Poller zwischen Ihrem konfigurierten Minimum und Maximum und fügt schnell bis zu 1.000 gleichzeitige Aufrufe pro Minute hinzu, um eine Verarbeitung Ihrer Amazon SQS SQS-Ereignisse mit geringer Latenz zu gewährleisten. Sie steuern Skalierung und Parallelität über diese minimalen und maximalen Event-Poller-Einstellungen. Weitere Informationen über Funktionsskalierung und Parallelität finden Sie unter. Verstehen der Skalierung von Lambda-Funktionen

Um mehr Nachrichten zu verarbeiten, können Sie Ihre Lambda-Funktion für einen höheren Durchsatz optimieren. Weitere Informationen finden Sie unter Grundlegendes zur AWS Lambda Skalierung mit Amazon SQS SQS-Standardwarteschlangen.

Filterkriterien

Verwenden Sie Filterkriterien, um zu steuern, welche Ereignisse Lambda zur Verarbeitung an Ihre Funktion sendet. Weitere Informationen finden Sie unter Steuern Sie, welche Ereignisse Lambda an Ihre Funktion sendet.

Maximale Parallelität

Die maximale Anzahl der gleichzeitigen Funktionen, die die Ereignisquelle aufrufen kann. Kann nicht verwendet werden, wenn der Bereitstellungsmodus aktiviert ist. Weitere Informationen finden Sie unter Konfigurieren der maximalen Gleichzeitigkeit für Amazon-SQS-Ereignisquellen.

Modus bereitgestellter Kapazität

Wenn diese Option aktiviert ist, werden spezielle Abfrageressourcen für Ihre Ereignisquellenzuordnung zugewiesen. Sie können die minimale (2-200) und die maximale (2-2000) Anzahl von Event-Pollern konfigurieren. Jeder Event Poller kann bis zu 1 MB/sec Durchsatz, bis zu 10 gleichzeitige Aufrufe oder bis zu 10 Amazon SQS-Polling-API-Aufrufe pro Sekunde verarbeiten.

> **Note**
>
> Hinweis: Sie können den Bereitstellungsmodus und die Option Maximale Parallelität nicht zusammen verwenden. Wenn der Bereitstellungsmodus aktiviert ist, verwenden Sie die Einstellung für maximale Polleranzahl, um die Parallelität zu steuern.

Konfiguration des Skalierungsverhaltens für SQS-Zuordnungen von Ereignisquellen

Sie können das Skalierungsverhalten Ihrer Amazon SQS SQS-Ereignisquellenzuordnungen entweder über die Einstellungen für maximale Parallelität oder durch die Aktivierung des Bereitstellungsmodus steuern. Diese Optionen schließen sich gegenseitig aus.

Standardmäßig skaliert Lambda Event-Poller automatisch auf der Grundlage des Nachrichtenvolumens. Wenn Sie den Bereitstellungsmodus aktivieren, weisen Sie eine Mindest- und Höchstzahl an dedizierten Abfrageressourcen zu, die bereit sind, die erwarteten Datenverkehrsmuster zu verarbeiten. Auf diese Weise können Sie die Leistung Ihrer Ereignisquellenzuordnung auf zwei Arten optimieren:

- Standardmodus (Standard): Lambda verwaltet die Skalierung automatisch, wobei mit einer kleinen Anzahl von Pollern begonnen und je nach Arbeitslast nach oben oder unten skaliert wird.
- Bereitgestellter Modus: Sie konfigurieren dedizierte Abfrageressourcen mit Mindest- und Höchstgrenzen, was eine dreimal schnellere Skalierung und eine bis zu 16-mal höhere Verarbeitungskapazität ermöglicht.

Bei Standardwarteschlangen verwendet Lambda Langabfrage, um eine Warteschlange so lange abzufragen, bis sie aktiv wird. Wenn Nachrichten verfügbar sind, beginnt Lambda mit der gleichzeitigen Verarbeitung von fünf Stapeln mit fünf gleichzeitigen Aufrufen Ihrer Funktion. Wenn Nachrichten weiterhin verfügbar sind, erhöht Lambda die Anzahl der Prozesse, die Batches lesen, um bis zu 300 gleichzeitige Aufrufe pro Minute. Die maximale Anzahl von Aufrufen, die eine Ereignisquellenzuordnung gleichzeitig verarbeiten kann, beträgt 1.250. Wenn der Verkehr gering ist, reduziert Lambda die Verarbeitung auf fünf gleichzeitige Aufrufe und kann auf bis zu zwei gleichzeitige Aufrufe optimiert werden, um die Amazon SQS SQS-Aufrufe und die entsprechenden Kosten zu reduzieren. Diese Optimierung ist jedoch nicht verfügbar, wenn Sie die Einstellung für maximale Gleichzeitigkeit aktivieren.

Bei FIFO-Warteschlangen sendet Lambda Nachrichten in der Reihenfolge, in der sie empfangen wurden, an Ihre Funktion. Wenn Sie eine Nachricht an eine FIFO-Warteschlange senden, geben Sie eine Nachrichtengruppen-ID an. Amazon SQS stellt sicher, dass Nachrichten in derselben Gruppe in der Reihenfolge an Lambda übermittelt werden. Wenn Lambda Ihre Nachrichten in Batches liest, kann jeder Batch Nachrichten aus mehr als einer Nachrichtengruppe enthalten, aber die Reihenfolge der Nachrichten bleibt erhalten. Wenn die Funktion einen Fehler zurückgibt, führt sie alle Wiederholungsversuche für die betroffenen Nachrichten aus, bevor Lambda weitere Nachrichten von derselben Gruppe erhält.

Im Bereitstellungsmodus kann jeder Event Poller bis zu 1 MB/sec Durchsatz, bis zu 10 gleichzeitige Aufrufe oder bis zu 10 Amazon SQS-Polling-API-Aufrufe pro Sekunde verarbeiten. Lambda skaliert die Anzahl der Event-Poller zwischen Ihrem konfigurierten Minimum und Maximum und fügt schnell bis zu 1.000 Parallelität pro Minute hinzu, um eine konsistente Verarbeitung Ihrer Amazon SQS SQS-Ereignisse mit niedriger Latenz zu gewährleisten. Die Verwendung des Bereitstellungsmodus verursacht zusätzliche Kosten. Eine detaillierte Preisgestaltung finden Sie unter Preise.AWS Lambda Jeder Event Poller verwendet lange Abfragen in Ihrer SQS-Warteschlange mit bis zu 10 Abfragen pro Sekunde, wodurch Kosten für SQS-API-Anfragen anfallen. Weitere Informationen finden Sie unter Amazon SQS SQS-Preise. Sie steuern Skalierung und Parallelität über diese minimalen und maximalen Event-Poller-Einstellungen, anstatt die Einstellung für maximale Parallelität zu verwenden, da diese Optionen nicht zusammen verwendet werden können.

> **Note**
>
> Sie können die Einstellung für maximale Parallelität und den Bereitstellungsmodus nicht gleichzeitig verwenden. Wenn der Bereitstellungsmodus aktiviert ist, steuern Sie die Skalierung und Parallelität Ihrer Amazon SQS SQS-Ereignisquellenzuordnung durch die minimale und maximale Anzahl von Event-Pollern.

Konfigurieren der maximalen Gleichzeitigkeit für Amazon-SQS-Ereignisquellen

Sie können die Einstellung für die maximale Gleichzeitigkeit verwenden, um das Skalierungsverhalten für Ihre SQS-Ereignisquellen zu steuern. Beachten Sie, dass die maximale Parallelität nicht verwendet werden kann, wenn der Bereitstellungsmodus aktiviert ist. Die Einstellung für die maximale Gleichzeitigkeit begrenzt die Anzahl der gleichzeitigen Instances der Funktion, die eine Amazon-SQS-Ereignisquelle aufrufen kann. Die maximale Gleichzeitigkeit ist eine Einstellung auf der Ebene der Ereignisquelle. Wenn Sie einer Funktion mehrere Amazon-SQS-Ereignisquellen zugeordnet haben, kann jede Ereignisquelle eine separate Einstellung für die maximale Gleichzeitigkeit haben.

Sie können die maximale Parallelität verwenden, um zu verhindern, dass eine Warteschlange die gesamte reservierte Gleichzeitigkeit der Funktion oder den Rest des Gleichzeitigkeitskontingents des Kontos verwendet. Für die Konfiguration der maximalen Gleichzeitigkeit auf einer Amazon-SQS-Ereignisquelle fallen keine Gebühren an.

Wichtig ist, dass maximale und reservierte Parallelität zwei unabhängige Einstellungen sind. Setzen Sie die maximale Parallelität nicht höher als die reservierte Parallelität der Funktion. Wenn Sie maximale Parallelität konfigurieren, stellen Sie sicher, dass die reservierte Parallelität Ihrer Funktion größer oder gleich der gesamten maximalen Parallelität für alle Amazon SQS-Ereignisquellen auf der Funktion ist. Andernfalls könnte Lambda Ihre Nachrichten drosseln.

Wenn die Gleichzeitigkeitsquote Ihres Kontos auf den Standardwert von 1 000 eingestellt ist, kann eine Amazon-SQS-Zuordnung von Ereignisquellen skaliert werden, um Funktionsinstanzen bis zu diesem Wert aufzurufen, es sei denn, Sie geben eine maximale Gleichzeitigkeit an.

Wenn Sie eine Erhöhung der standardmäßigen Gleichzeitigkeitsquote Ihres Kontos erhalten, kann Lambda möglicherweise keine gleichzeitigen Funktionsinstanzen bis zur neuen Kontingente aufrufen. Standardmäßig kann Lambda so skaliert werden, dass bis zu 1.250 gleichzeitige Funktionsinstanzen für eine Amazon-SQS-Zuordnung von Ereignisquellen aufgerufen werden. Wenn dies für Ihren Anwendungsfall nicht ausreicht, wenden Sie sich an den AWS Support, um eine Erhöhung der Parallelität der Amazon SQS SQS-Ereignisquellenzuordnungen für Ihr Konto zu besprechen.

> **Note**
>
> Bei FIFO-Warteschlangen werden gleichzeitige Aufrufe entweder durch die Anzahl der Nachrichtengruppen IDs (`messageGroupId`) oder durch die Einstellung für maximale Parallelität begrenzt — je nachdem, welcher Wert niedriger ist. Wenn Sie beispielsweise über sechs Nachrichtengruppen verfügen IDs und die maximale Parallelität auf 10 festgelegt ist, kann Ihre Funktion maximal sechs gleichzeitige Aufrufe haben.

Sie können die maximale Gleichzeitigkeit für neue und vorhandene Zuordnung von Ereignisquellen in Amazon SQS konfigurieren.

Konfigurieren der maximalen Gleichzeitigkeit mithilfe der Lambda-Konsole

1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.
2. Wählen Sie den Namen einer Funktion aus.

3. Wählen Sie unter Function overview (Funktionsübersicht) die Option SQS aus. Dadurch wird die Registerkarte Configuration (Konfiguration) geöffnet.
4. Wählen Sie den Amazon-SQS-Auslöser aus und klicken Sie auf Edit (Bearbeiten).
5. Geben Sie für Maximum concurrency (Maximale Gleichzeitigkeit) eine Zahl zwischen 2 und 1 000 ein. Um die maximale Gleichzeitigkeit zu deaktivieren, lassen Sie das Feld leer.
6. Wählen Sie Speichern.

Konfigurieren Sie die maximale Parallelität mit der () AWS Command Line InterfaceAWS CLI

Verwenden Sie den Befehl update-event-source-mapping mit der Option `--scaling-config` . Beispiel:

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
    --scaling-config '{"MaximumConcurrency":5}'
```

Um die maximale Gleichzeitigkeit zu deaktivieren, geben Sie einen leeren Wert für `--scaling-config` ein:

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
    --scaling-config "{}"
```

Konfigurieren der maximalen Gleichzeitigkeit mithilfe der Lambda-API

Verwenden Sie die UpdateEventSourceMappingAktion CreateEventSourceMappingoder mit einem ScalingConfigObjekt.

Behandlung von Fehlern für eine SQS-Ereignisquelle in Lambda

Um Fehler im Zusammenhang mit einer SQS-Ereignisquelle zu behandeln, verwendet Lambda automatisch eine Wiederholungsstrategie mit einer Backoff-Strategie. Sie können das Verhalten bei der Fehlerbehandlung auch anpassen, indem Sie Ihre SQS-Zuordnung von Ereignisquellen so konfigurieren, dass sie partielle Batch-Antworten zurückgibt.

Backoff-Strategie für fehlgeschlagene Aufrufe

Wenn ein Aufruf fehlschlägt, versucht Lambda, den Aufruf zu wiederholen, während eine Backoff-Strategie implementiert wird. Die Backoff-Strategie unterscheidet sich geringfügig, je nachdem, ob

Lambda den Fehler aufgrund eines Fehlers in Ihrem Funktionscode oder aufgrund einer Drosselung festgestellt hat.

- Wenn Ihr Funktionscode den Fehler verursacht hat, stoppt Lambda die Verarbeitung und versucht erneut, den Aufruf zu starten. In der Zwischenzeit zieht sich Lambda allmählich zurück und reduziert die Menge der Gleichzeitigkeit, die Ihrer Amazon SQS-Zuordnung von Ereignisquellen zugewiesen wird. Nachdem die Zeitspanne für die Sichtbarkeit Ihrer Warteschlange abgelaufen ist, wird die Nachricht wieder in der Warteschlange erscheinen.
- Wenn der Aufruf aufgrund von Drosselung fehlschlägt, setzt Lambda schrittweise Wiederholungsversuche zurück, indem es die Menge der Gleichzeitigkeit reduziert, die Ihrer Zuordnung von Ereignisquellen in Amazon SQS zugewiesen ist. Lambda fährt fort, die Nachricht erneut zu versuchen, bis der Zeitstempel der Nachricht das Sichtbarkeits-Timeout Ihrer Warteschlange überschreitet, an welchem Punkt Lambda die Nachricht verwirft.

Implementierung von partiellen Batch-Antworten

Wenn die Lamda-Funktion während der Verarbeitung eines Batches auf einen Fehler stößt, werden standardmäßig alle Nachrichten in diesem Batch wieder in der Warteschlange sichtbar, einschließlich Nachrichten, die Lambda erfolgreich verarbeitet hat. Infolgedessen kann es passieren, dass die Funktion dieselbe Nachricht mehrmals verarbeitet.

Damit nicht erfolgreich bearbeiten Nachrichten in einem fehlgeschlagenen Batch erneut verarbeitet werden, können Sie die Zuordnung von Ereignisquellen so konfigurieren, dass nur die fehlgeschlagenen Nachrichten wieder sichtbar werden. Dies wird als partielle Batch-Antwort bezeichnet. Um partielle Batch-Antworten zu aktivieren, geben Sie `ReportBatchItemFailures` dies bei der Konfiguration Ihrer Ereignisquellenzuordnung für die FunctionResponseTypesAktion an. Dadurch kann die Funktion einen Teilerfolg zurückgeben, was die Anzahl unnötiger Wiederholungsversuche für Datensätze reduzieren kann.

> **Note**
>
> Das Batch Processor Utility von Powertools for AWS Lambda verarbeitet die gesamte partielle Batch-Antwortlogik automatisch. Dieses Tool vereinfacht die Implementierung von Batch-Verarbeitungsmustern und reduziert den Bedarf an benutzerdefiniertem Code, der für die korrekte Behandlung von Fehlern bei Batch-Elementen erforderlich ist. Es ist für Python, Java, Typescript und .NET verfügbar.

Wenn `ReportBatchItemFailures` aktiviert ist, skaliert Lambda die Nachrichtenabfrage nicht herunter, wenn Funktionsaufrufe fehlschlagen. Wenn Sie erwarten, dass einige Nachrichten ausfallen – und Sie nicht möchten, dass sich diese Fehler auf die Nachrichtenverarbeitungsrate auswirken – verwenden Sie `ReportBatchItemFailures`.

> **Note**
>
> Berücksichtigen Sie bei Verwendung von partiellen Batch-Antworten Folgendes:
>
> - Wenn die Funktion eine Ausnahme ausgibt, gilt der gesamte Batch als fehlgeschlagen.
> - Wenn Sie dieses Feature mit einer FIFO-Warteschlange verwenden, sollte Ihre Funktion die Verarbeitung von Nachrichten nach dem ersten Fehler beenden und alle fehlgeschlagenen und nicht verarbeiteten Nachrichten in `batchItemFailures` zurückgeben. Das hilft, die Reihenfolge der Nachrichten in der Warteschlange beizubehalten.

So aktivieren Sie partielle Batchberichterstattung

1. Sehen Sie sich die bewährten Methoden für die Implementierung von partiellen Batch-Antworten an.
2. Führen Sie den folgenden Befehl aus, um `ReportBatchItemFailures` für Ihre Funktion zu aktivieren. Führen Sie den Befehl aus, um die UUID Ihrer Ereignisquellenzuordnung abzurufen. list-event-source-mappings AWS CLI

   ```
   aws lambda update-event-source-mapping \
   --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
   --function-response-types "ReportBatchItemFailures"
   ```

3. Aktualisieren Sie Ihren Funktionscode, um alle Ausnahmen abzufangen und fehlgeschlagene Nachrichten in einer `batchItemFailures`-JSON-Antwort zurückzugeben. Die `batchItemFailures` Antwort muss eine Liste von Nachrichten IDs in Form von `itemIdentifier` JSON-Werten enthalten.

   Nehmen wir beispielsweise an, Sie haben einen Stapel von fünf Nachrichten mit Nachricht IDs `id1id2`,`id3`,`id4`, und`id5`. Die Funktion verarbeitet erfolgreich `id1`, `id3` und `id5`. Damit die Nachrichten `id2` und `id4` in der Warteschlange wieder sichtbar werden, sollte Ihre Funktion folgende Antwort zurückgeben:

```
{
  "batchItemFailures": [
        {
            "itemIdentifier": "id2"
        },
        {
            "itemIdentifier": "id4"
        }
    ]
}
```

Hier sind einige Beispiele für Funktionscode, der die Liste der fehlgeschlagenen Nachrichten IDs im Batch zurückgibt:

.NET

SDK für .NET

> **Note**
>
> Es gibt noch mehr dazu GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Melden von SQS-Batchelementfehlern mit Lambda unter Verwendung von .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

SDK für Go V2

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Melden von SQS-Batchelementfehlern mit Lambda unter Verwendung von 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 für Java 2.x

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Melden von Fehlern bei SQS-Batchelementen mit Lambda unter Verwendung von 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 für JavaScript (v3)

> **Note**
>
> Es gibt noch mehr dazu GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Melden von SQS-Batch-Elementfehlern mit Lambda unter Verwendung von. JavaScript

```
// 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}`);
}
```

Melden von SQS-Batch-Elementfehlern mit Lambda unter Verwendung von. TypeScript

```
// 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

SDK für PHP

> **Note**
>
> Es gibt noch mehr dazu. GitHub Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Melden von SQS-Batchelementfehlern mit Lambda unter Verwendung von 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

SDK für Python (Boto3)

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Melden von Fehlern bei SQS-Batchelementen mit Lambda unter Verwendung von 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

SDK für Ruby

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Melden von Fehlern bei SQS-Batchelementen mit Lambda unter Verwendung von 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 für Rust

> **Note**
>
> Es gibt noch mehr dazu GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Melden von Fehlern bei SQS-Batchelementen mit Lambda unter Verwendung von 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
}
```

Wenn die fehlgeschlagenen Ereignisse nicht in die Warteschlange zurückkehren, finden Sie weitere Informationen unter Wie behebe ich Probleme mit der Lambda-Funktion ReportBatchItemFailures SQS? im AWS Knowledge Center.

Erfolgs- und Misserfolgsbedingungen

Lambda behandelt einen Batch als komplett erfolgreich, wenn die Funktion eines der folgenden Elemente zurückgibt:

- Eine leere `batchItemFailures`-Liste
- Eine ungültige `batchItemFailures`-Liste
- Ein leeres `EventResponse`
- Ein ungültiges `EventResponse`

Lambda behandelt einen Batch als komplett fehlgeschlagen, wenn die Funktion eines der folgenden Elemente zurückgibt:

- Eine ungültige JSON-Antwort
- Eine leere Zeichenfolge `itemIdentifier`
- Ein ungültiges `itemIdentifier`
- Ein `itemIdentifier` mit einem falschen Schlüsselnamen
- Einen `itemIdentifier`-Wert mit einer Nachrichten-ID, die nicht existiert

CloudWatch Metriken

Um festzustellen, ob Ihre Funktion Fehler bei Chargenartikeln korrekt meldet, können Sie die Amazon SQS-Metriken `NumberOfMessagesDeleted` und die `ApproximateAgeOfOldestMessage` Amazon SQS-Metriken in Amazon CloudWatch überwachen.

- `NumberOfMessagesDeleted` verfolgt die Anzahl der Nachrichten, die aus der Warteschlange entfernt wurden. Wenn der Wert auf 0 sinkt, ist dies ein Zeichen dafür, dass die Funktionsantwort fehlgeschlagene Nachrichten nicht korrekt zurückgibt.
- `ApproximateAgeOfOldestMessage` verfolgt, wie lange die älteste Nachricht in der Warteschlange geblieben ist. Ein starker Anstieg dieser Metrik kann darauf hinweisen, dass die Funktion fehlgeschlagene Nachrichten nicht korrekt zurückgibt.

Verwendung von Powertools für AWS Lambda die Batch-Verarbeitung

Das Batchverarbeitungsprogramm von Powertools for verarbeitet AWS Lambda automatisch die Logik für partielle Batch-Antworten und reduziert so die Komplexität der Implementierung von Batch-Fehlerberichten. Hier sind Beispiele für die Verwendung des Batch-Prozessors:

Python

> **Note**
>
> Umfassende Beispiele und Anweisungen zur Einrichtung finden Sie in der Dokumentation zum Batch-Prozessor.

Verarbeitung von Amazon SQS SQS-Nachrichten mit AWS Lambda Batch-Prozessor.

```
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**
>
> Umfassende Beispiele und Anweisungen zur Einrichtung finden Sie in der Dokumentation zum Batch-Prozessor.

Verarbeitung von Amazon SQS SQS-Nachrichten mit AWS Lambda Batch-Prozessor.

```
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,
    });
};
```

Lambdaparameter für Amazon-SQS-Zuordnungen von Ereignisquellen

Alle Lambda-Ereignisquelltypen verwenden dieselben CreateEventSourceMappingUpdateEventSourceMappingAPI-Operationen. Allerdings gelten nur einige der Parameter für Amazon SQS.

Parameter	Erforderlich	Standard	Hinweise
BatchSize	N	10	Bei Standardw arteschlangen beträgt der Maximalwert 10 000. Bei FIFO-Warteschlange n beträgt der Maximalwert 10.
Aktiviert	N	true	Keine
EventSourceArn	Y	–	Der ARN des Datenstroms oder eines Stream-Ko nsumenten
FunctionName	Y	–	Keine
FilterCriteria	N	–	Steuern Sie, welche Ereignisse Lambda an Ihre Funktion sendet
FunctionResponseTy pes	N	–	Damit Ihre Funktion bestimmte Fehler in einem Batch meldet, beziehen Sie den Wert `ReportBat chItemFailures` in `FunctionR esponseTypes` ein. Weitere Informati

Parameter	Erforderlich	Standard	Hinweise
			onen finden Sie unter Implementierung von partiellen Batch-Ant worten.
MaximumBatchingWindowInSeconds	N	0	Das Batching-Fenster wird für FIFO-Wart eschlangen nicht unterstützt
ProvisionedPollerConfig	N	–	Konfiguriert die minimale (2-200) und maximale (2-2000) Anzahl von dediziert en Event-Pollern für die SQS-Ereig nisquellenzuordnun g. Jeder Poller kann bis zu 1 MB/sec Durchsatz und 10 gleichzeitige Aufrufe verarbeiten.
ScalingConfig	N	–	Konfigurieren der maximalen Gleichzei tigkeit für Amazon-SQ S-Ereignisquellen

Verwendung der Ereignisfilterung mit einer Amazon-SQS-Ereignisquelle

Sie können die Ereignisfilterung verwenden, um zu steuern, welche Datensätze aus einem Stream oder einer Warteschlange Lambda an Ihre Funktion sendet. Allgemeine Informationen über die Funktionsweise der Ereignisfilterung finden Sie unter the section called “Ereignisfilterung”.

Dieser Abschnitt konzentriert sich auf die Ereignisfilterung für Amazon-SQS-Ereignisquellen.

> **Note**
>
> Amazon-SQS-Zuordnungen von Ereignisquellen unterstützen nur das Filtern nach dem body-Schlüssel.

Themen

- Grundlagen der Amazon-SQS-Ereignisfilterung

Grundlagen der Amazon-SQS-Ereignisfilterung

Angenommen, Ihre Amazon-SQS-Warteschlange enthält Nachrichten im folgenden JSON-Format.

```
{
    "RecordNumber": 1234,
    "TimeStamp": "yyyy-mm-ddThh:mm:ss",
    "RequestCode": "AAAA"
}
```

Ein Beispieldatensatz für diese Warteschlange würde wie folgt aussehen.

```
{
    "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"
}
```

Um anhand des Inhalts Ihrer Amazon-SQS-Nachrichten zu filtern, verwenden Sie den `body`-Schlüssel im Amazon-SQS-Nachrichtendatensatz. Angenommen, Sie möchten nur die Datensätze verarbeiten, bei denen `RequestCode` in Ihrer Amazon-SQS-Nachricht „BBBB“ ist. Das `FilterCriteria`-Objekt würde wie folgt aussehen.

```
{
    "Filters": [
        {
            "Pattern": "{ \"body\" : { \"RequestCode\" : [ \"BBBB\" ] } }"
        }
    ]
}
```

Zur Verdeutlichung sehen Sie hier den Wert des Filter-`Pattern` in reinem JSON.

```
{
    "body": {
        "RequestCode": [ "BBBB" ]
        }
}
```

Sie können Ihren Filter mithilfe der Konsole, AWS CLI oder einer AWS SAM-Vorlage hinzufügen.

Console

Um diesen Filter mithilfe der Konsole hinzuzufügen, folgen Sie den Anweisungen unter Anhängen von Filterkriterien an eine Ereignisquellenzuordnung (Konsole) und geben Sie die folgende Zeichenfolge für die Filterkriterien ein.

```
{ "body" : { "RequestCode" : [ "BBBB" ] } }
```

AWS CLI

Um eine neue Zuordnung von Ereignisquellen mit diesen Filterkriterien über die AWS Command Line Interface (AWS CLI) zu erstellen, führen Sie den folgenden Befehl aus.

```
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\" ] } }"}]}'
```

Führen Sie den folgenden Befehl aus, um diese Filterkriterien zu einer vorhandenen Zuordnung von Ereignisquellen hinzuzufügen.

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"body\" : { \"RequestCode\" :
 [ \"BBBB\" ] } }"}]}'
```

AWS SAM

Um diesen Filter mit der AWS SAM hinzuzufügen, fügen Sie das folgende Snippet zur YAML-Vorlage für Ihre Ereignisquelle hinzu.

```
FilterCriteria:
  Filters:
    - Pattern: '{ "body" : { "RequestCode" : [ "BBBB" ] } }'
```

Angenommen, Sie möchten, dass Ihre Funktion nur die Datensätze verarbeitet, bei denen `RecordNumber` größer als 9999 ist. Das `FilterCriteria`-Objekt würde wie folgt aussehen.

```
{
    "Filters": [
        {
            "Pattern": "{ \"body\" : { \"RecordNumber\" : [ { \"numeric\": [ \">\",
 9999 ] } ] } }"
        }
    ]
}
```

Zur Verdeutlichung sehen Sie hier den Wert des Filter-`Pattern` in reinem JSON.

```
{
    "body": {
        "RecordNumber": [
            {
                "numeric": [ ">", 9999 ]
            }
        ]
    }
}
```

Sie können Ihren Filter mithilfe der Konsole, AWS CLI oder einer AWS SAM-Vorlage hinzufügen.

Console

Um diesen Filter mithilfe der Konsole hinzuzufügen, folgen Sie den Anweisungen unter Anhängen von Filterkriterien an eine Ereignisquellenzuordnung (Konsole) und geben Sie die folgende Zeichenfolge für die Filterkriterien ein.

```
{ "body" : { "RecordNumber" : [ { "numeric": [ ">", 9999 ] } ] } }
```

AWS CLI

Um eine neue Zuordnung von Ereignisquellen mit diesen Filterkriterien über die AWS Command Line Interface (AWS CLI) zu erstellen, führen Sie den folgenden Befehl aus.

```
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 ] } ] } }"}]}'
```

Führen Sie den folgenden Befehl aus, um diese Filterkriterien zu einer vorhandenen Zuordnung von Ereignisquellen hinzuzufügen.

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"body\" : { \"RecordNumber\" :
 [ { \"numeric\": [ \">\", 9999 ] } ] } }"}]}'
```

AWS SAM

Um diesen Filter mit der AWS SAM hinzuzufügen, fügen Sie das folgende Snippet zur YAML-Vorlage für Ihre Ereignisquelle hinzu.

```
FilterCriteria:
  Filters:
    - Pattern: '{ "body" : { "RecordNumber" : [ { "numeric": [ ">", 9999 ] } ] } }'
```

Für Amazon SQS kann der Nachrichtenkörper eine beliebige Zeichenfolge sein. Dies kann jedoch problematisch sein, wenn die `FilterCriteria` erwarten, dass `body` ein gültiges JSON-Format

hat. Umgekehrt gilt dasselbe – wenn der Textkörper der eingehenden Nachricht ein JSON-Format aufweist, die Filterkriterien jedoch erwarten, dass `body` eine einfache Zeichenfolge ist, kann dies zu unbeabsichtigtem Verhalten führen.

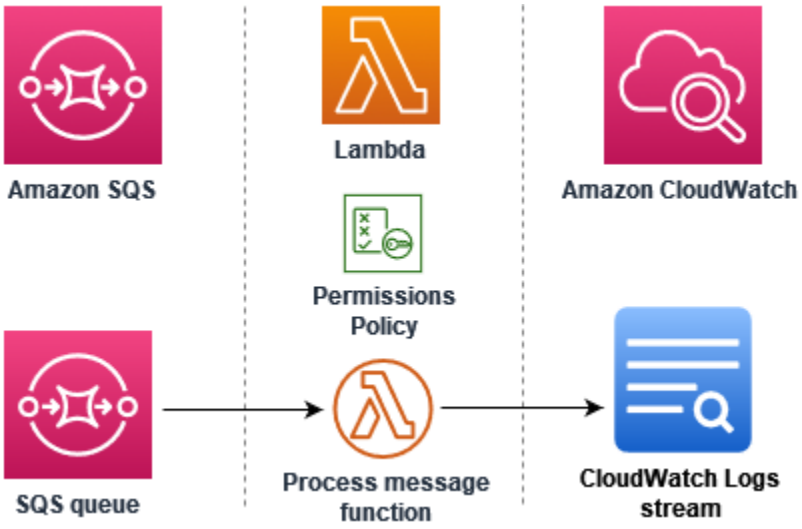
Um dieses Problem zu vermeiden, stellen Sie sicher, dass das Körperformat in Ihren `FilterCriteria` mit dem erwarteten Format von `body` in den Nachrichten übereinstimmt, die Sie von Ihrer Warteschlange erhalten. Bevor Sie Ihre Nachrichten filtern, wertet Lambda automatisch das Format des Körpers der eingehenden Nachricht und Ihres Filtermusters für `body` aus. Wenn es keine Übereinstimmung gibt, verwirft Lambda die Nachricht. In der folgenden Tabelle ist diese Auswertung zusammengefasst:

body-Format der eingehenden Nachricht	**body**-Format des Filtermusters	Resultierende Aktion
Einfache Zeichenfolge	Einfache Zeichenfolge	Lambda filtert basierend auf Ihren Filterkriterien.
Einfache Zeichenfolge	Kein Filtermuster für Dateneigenschaften	Lambda filtert (nur für die anderen Metadateneigenschaften) basierend auf Ihren Filterkriterien.
Einfache Zeichenfolge	Gültiges JSON	Lambda verwirft die Nachricht.
Gültiges JSON	Einfache Zeichenfolge	Lambda verwirft die Nachricht.
Gültiges JSON	Kein Filtermuster für Dateneigenschaften	Lambda filtert (nur für die anderen Metadateneigenschaften) basierend auf Ihren Filterkriterien.
Gültiges JSON	Gültiges JSON	Lambda filtert basierend auf Ihren Filterkriterien.

Tutorial: Verwenden von Lambda mit Amazon SQS

In diesem Tutorial erstellen Sie eine Lambda-Funktion, die Nachrichten aus einer Warteschlange von Amazon Simple Queue Service (Amazon SQS) verarbeitet. Die Lambda-Funktion wird ausgeführt,

wenn der Warteschlange eine neue Nachricht hinzugefügt wird. Die Funktion schreibt die Nachrichten in einen Amazon CloudWatch Logs-Stream. Das folgende Diagramm zeigt die AWS -Ressourcen, die Sie zur Durchführung des Tutorials verwenden.



Führen Sie für dieses Tutorial die folgenden Schritte aus:

1. Erstellen Sie eine Lambda-Funktion, die Nachrichten in CloudWatch Logs schreibt.
2. Erstellen einer Amazon SQS-Warteschlange
3. Erstellen Sie eine Zuordnung von Ereignisquellen in Lambda. Die Zuordnung von Ereignisquellen liest die Amazon-SQS-Warteschlange und ruft Ihre Lambda-Funktion auf, wenn eine neue Nachricht hinzugefügt wird.
4. Testen Sie das Setup, indem Sie Nachrichten zu Ihrer Warteschlange hinzufügen und die Ergebnisse in CloudWatch Logs überwachen.

Voraussetzungen

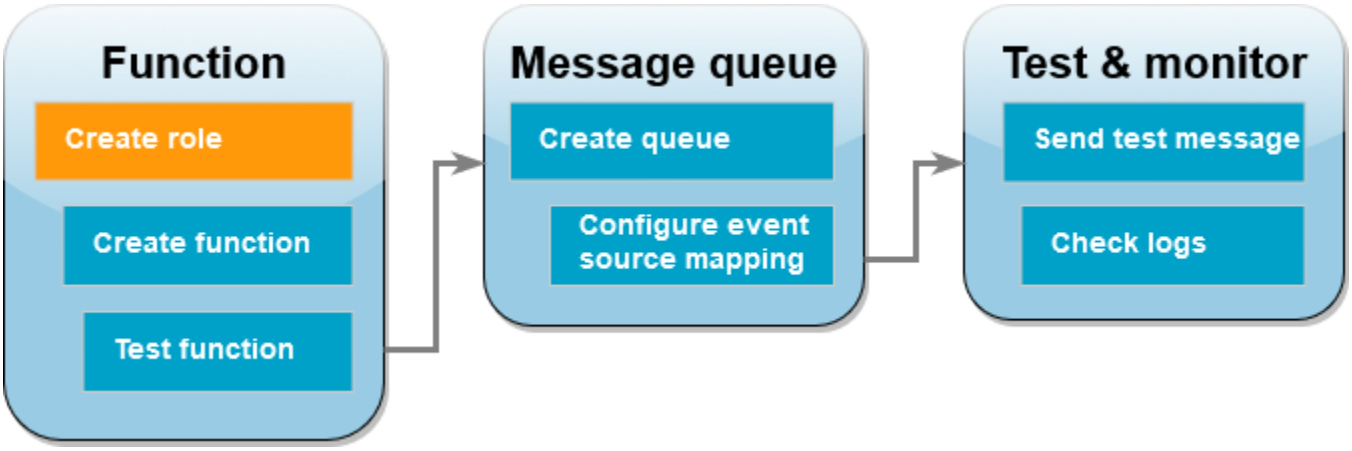
Installieren Sie das AWS Command Line Interface

Wenn Sie das noch nicht installiert haben AWS Command Line Interface, folgen Sie den Schritten unter Installieren oder Aktualisieren der neuesten Version von AWS CLI, um es zu installieren.

Das Tutorial erfordert zum Ausführen von Befehlen ein Befehlszeilenterminal oder eine Shell. Verwenden Sie unter Linux und macOS Ihre bevorzugte Shell und Ihren bevorzugten Paketmanager.

> **Note**
>
> In Windows werden einige Bash-CLI-Befehle, die Sie häufig mit Lambda verwenden (z. B. `zip`), von den integrierten Terminals des Betriebssystems nicht unterstützt. Um eine in Windows integrierte Version von Ubuntu und Bash zu erhalten, installieren Sie das Windows-Subsystem für Linux.

Erstellen der Ausführungsrolle



Eine Ausführungsrolle ist eine AWS Identity and Access Management (IAM-) Rolle, die einer Lambda-Funktion Zugriff AWS-Services und Ressourcen gewährt. Damit Ihre Funktion Elemente aus Amazon SQS lesen kann, fügen Sie die AWSLambdaSQSQueueExecutionRoleBerechtigungsrichtlinie bei.

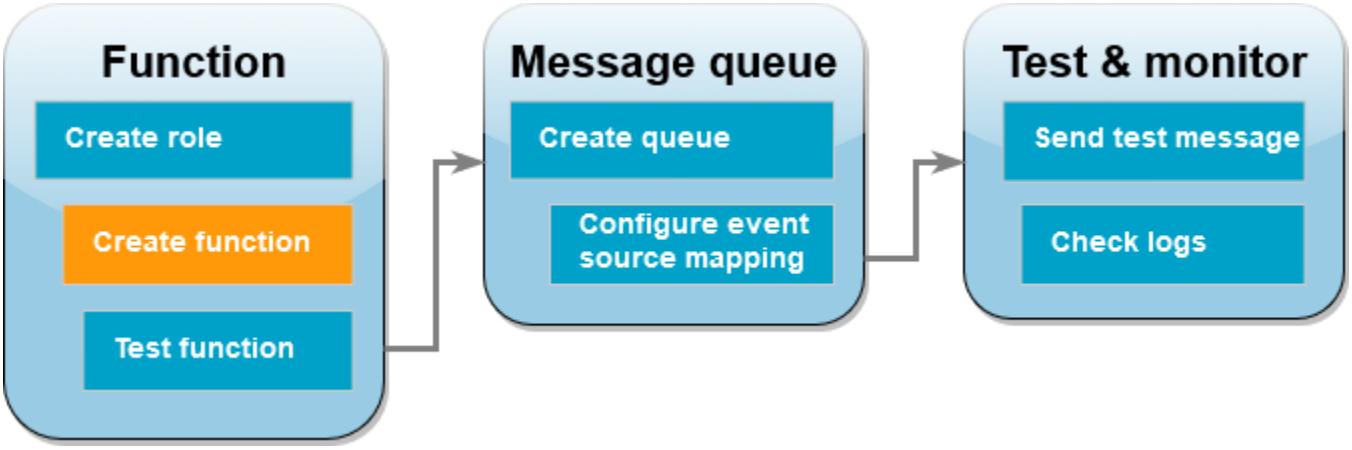
So erstellen Sie eine Ausführungsrolle und fügen eine Amazon-SQS-Berechtigungsrichtlinie hinzu

1. Öffnen Sie die Seite Roles (Rollen) in der IAM-Konsole.
2. Wählen Sie Rolle erstellen aus.
3. Wählen Sie unter Vertrauenswürdiger Entitätstyp die Option AWS -Service aus.
4. Wählen Sie unter Anwendungsfall die Option Lambda aus.
5. Wählen Sie Weiter aus.
6. Geben Sie im Suchfeld Berechtigungsrichtlinien die Zeichenfolge **`AWSLambdaSQSQueueExecutionRole`** ein.
7. Wählen Sie die AWSLambdaSQSQueueExecutionRoleRichtlinie aus und klicken Sie dann auf Weiter.

8. Geben Sie unter Rollendetails für Rollenname den Namen **`lambda-sqs-role`** ein und wählen Sie anschließend Rolle erstellen aus.

Schreiben Sie sich nach der Erstellung der Rolle den Amazon-Ressourcennamen (ARN) Ihrer Ausführungsrolle auf. Sie werden ihn in späteren Schritten noch benötigen.

Erstellen der Funktion



Erstellen Sie eine Lambda-Funktion, die Ihre Amazon-SQS-Nachrichten verarbeitet. Der Funktionscode protokolliert den Hauptteil der Amazon SQS-Nachricht in CloudWatch Logs.

Dieses Tutorial verwendet die Runtime Node.js 24, aber wir haben auch Beispielcode in anderen Runtime-Sprachen bereitgestellt. Sie können die Registerkarte im folgenden Feld auswählen, um Code für die gewünschte Laufzeit anzusehen. Der JavaScript Code, den Sie in diesem Schritt verwenden, befindet sich im ersten Beispiel, das auf der JavaScriptRegisterkarte angezeigt wird.

.NET

SDK für .NET

> **Note**
>
> Es gibt noch mehr dazu GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines SQS-Ereignisses mit Lambda unter Verwendung von .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

SDK für Go V2

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines SQS-Ereignisses mit Lambda unter Verwendung von 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 für Java 2.x

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines SQS-Ereignisses mit Lambda unter Verwendung von 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 für JavaScript (v3)

> **Note**
>
> Es gibt noch mehr dazu GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Konsumieren eines SQS-Ereignisses mit Lambda unter Verwendung. JavaScript

```
// 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;
  }
}
```

Konsumieren eines SQS-Ereignisses mit Lambda unter Verwendung. TypeScript

```
// 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

SDK für PHP

> **Note**
>
> Es gibt noch mehr dazu. GitHub Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines SQS-Ereignisses mit Lambda unter Verwendung von 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

SDK für Python (Boto3)

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines SQS-Ereignisses mit Lambda unter Verwendung von 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

SDK für Ruby

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines SQS-Ereignisses mit Lambda unter Verwendung von 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 für Rust

> **Note**
>
> Es gibt noch mehr dazu GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines SQS-Ereignisses mit Lambda unter Verwendung von 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
}
```

So erstellen Sie eine Node.js-Lambda-Funktion

1. Erstellen Sie ein Verzeichnis für das Projekt und wechseln Sie dann zu diesem Verzeichnis.

```
mkdir sqs-tutorial
cd sqs-tutorial
```

2. Kopieren Sie den JavaScript Beispielcode in eine neue Datei mit dem Namen`index.js`.
3. Erstellen Sie ein Bereitstellungspaket mit dem folgenden `zip`-Befehl.

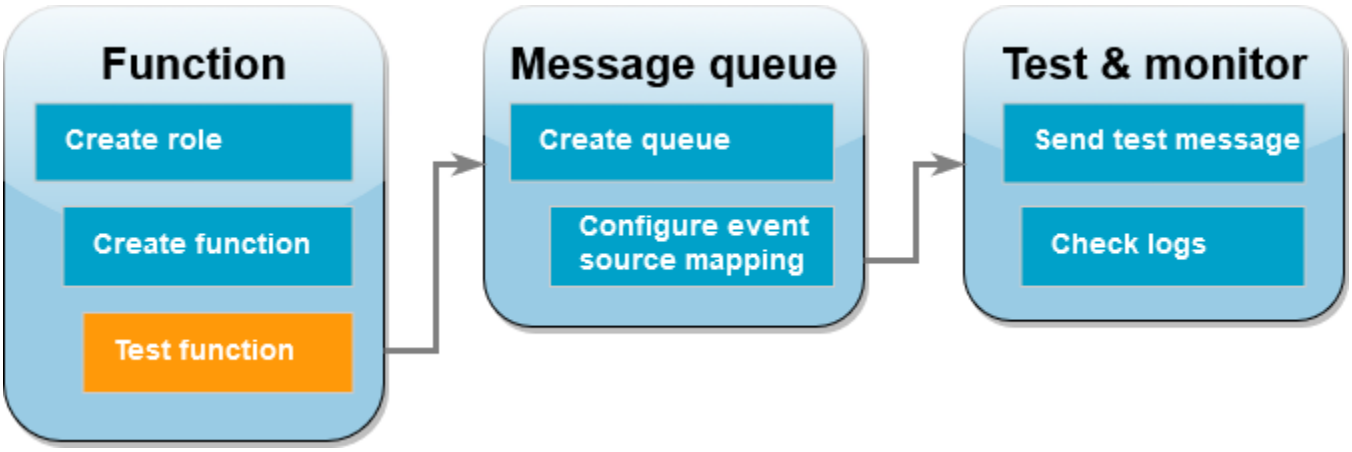
```
zip function.zip index.js
```

4. Erstellen Sie eine Lambda-Funktion mithilfe des AWS CLI -Befehls create-function. Geben Sie für den `role`-Parameter den ARN der Ausführungsrolle ein, die Sie zuvor erstellt haben.

> **Note**
>
> Die Lambda-Funktion und die Amazon-SQS-Warteschlange müssen sich in demselben AWS-Region befinden.

```
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
```

Testen der Funktion



Rufen Sie Ihre Lambda-Funktion manuell mit dem `invoke` AWS CLI Befehl und einem Amazon SQS SQS-Beispielereignis auf.

So rufen Sie die Lambda-Funktion mit einem Beispielereignis auf

1. Speichern Sie die folgende JSON als Datei mit dem Namen `input.json`. Dieser JSON-Code simuliert ein Ereignis, das so ggf. von Amazon SQS an Ihre Lambda-Funktion gesendet wird, wobei `"body"` die tatsächliche Nachricht aus der Warteschlange enthält. In diesem Beispiel lautet die Nachricht `"test"`.

   Example Amazon-SQS-Ereignis

   Dies ist ein Testereignis. Nachricht und Kontonummer müssen nicht geändert werden.

   ```
   {
       "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. Führen Sie den folgenden Invoke-Befehl aus. AWS CLI Dieser Befehl gibt CloudWatch Logs in der Antwort zurück. Weitere Informationen über das Abrufen von Protokollen finden Sie unter Zugriffs-Logs mit dem AWS CLI.

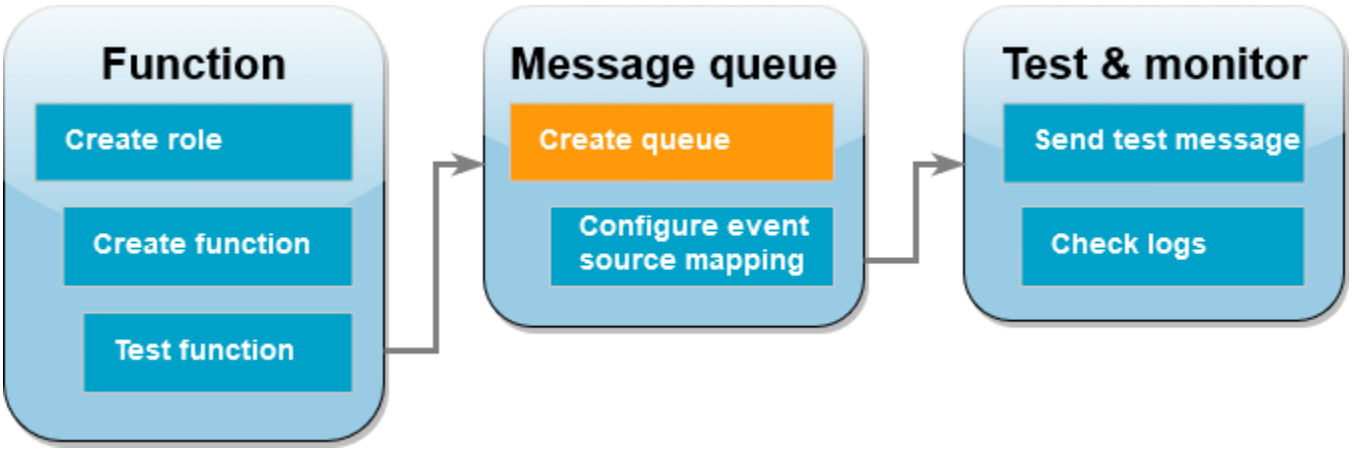
```
aws lambda invoke --function-name ProcessSQSRecord --payload file://input.json out
 --log-type Tail \
--query 'LogResult' --output text --cli-binary-format raw-in-base64-out | base64 --
decode
```

   Die cli-binary-format Option ist erforderlich, wenn Sie AWS CLI Version 2 verwenden. Um dies zur Standardeinstellung zu machen, führen Sie `aws configure set cli-binary-format raw-in-base64-out` aus. Weitere Informationen finden Sie unter Von AWS CLI unterstützte globale Befehlszeilenoptionen im AWS Command Line Interface -Benutzerhandbuch für Version 2.

3. Suchen Sie in der Antwort nach dem Protokoll `INFO`. Dort protokolliert die Lambda-Funktion den Nachrichtentext. Sie sollten Protokolle sehen, die wie folgt aussehen:

```
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
```

Erstellen einer Amazon-SQS-Warteschlange



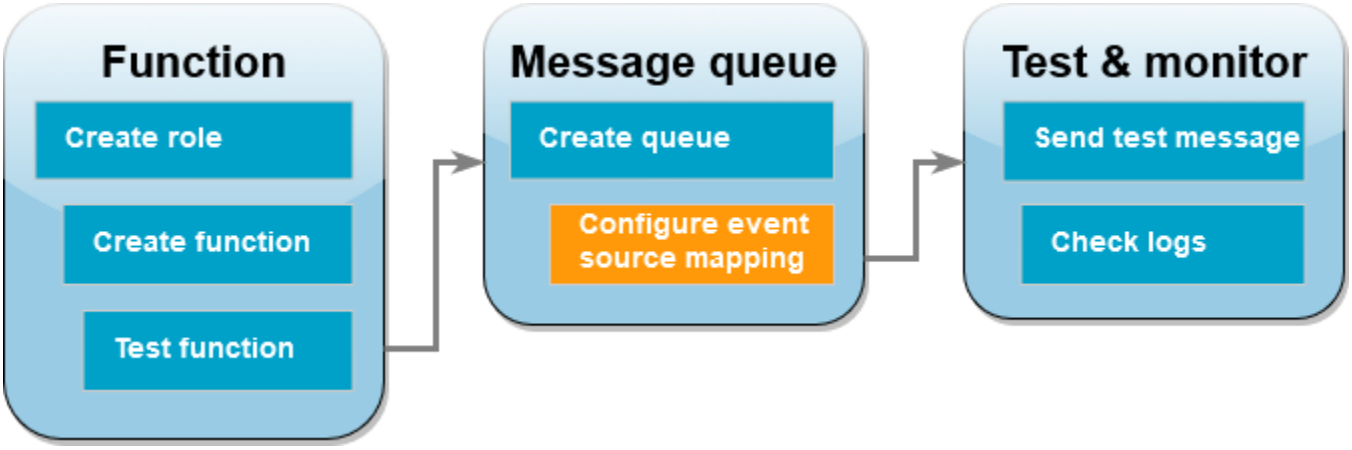
Erstellen Sie eine Amazon-SQS-Warteschlange, die die Lambda-Funktion als Ereignisquelle verwenden kann. Die Lambda-Funktion und die Amazon-SQS-Warteschlange müssen sich in demselben AWS-Region befinden.

So erstellen Sie eine Warteschlange

1. Öffnen Sie die Amazon-SQS-Konsole.
2. Wählen Sie Create queue (Warteschlange erstellen) aus.
3. Geben Sie einen Namen für die Warteschlange ein. Übernehmen Sie bei allen anderen Optionen die Standardeinstellungen.
4. Wählen Sie Create queue (Warteschlange erstellen) aus.

Notieren Sie sich nach dem Erstellen der Warteschlange ihren ARN. Sie benötigen ihn im nächsten Schritt, um die Warteschlange Ihrer Lambda-Funktion zuzuordnen.

Konfigurieren der Ereignisquelle



Verbinden Sie die Amazon-SQS-Warteschlange mit Ihrer Lambda-Funktion, indem Sie eine Zuordnung von Ereignisquellen erstellen. Die Zuordnung von Ereignisquellen liest die Amazon-SQS-Warteschlange und ruft Ihre Lambda-Funktion auf, wenn eine neue Nachricht hinzugefügt wird.

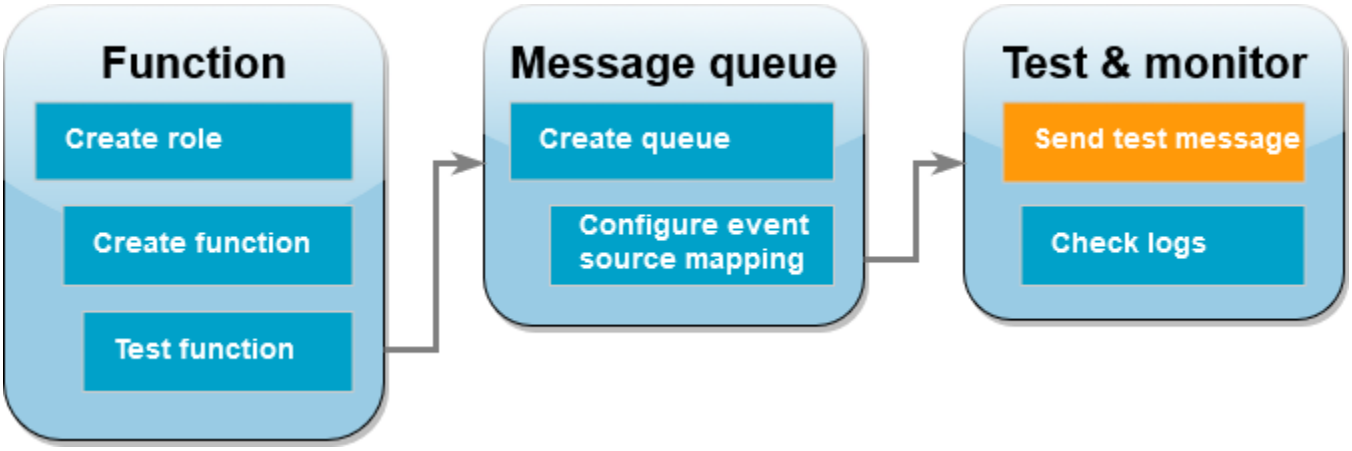
Verwenden Sie den Befehl, um eine Zuordnung zwischen Ihrer Amazon SQS SQS-Warteschlange und Ihrer Lambda-Funktion zu erstellen. create-event-source-mapping AWS CLI Beispiel:

```
aws lambda create-event-source-mapping --function-name ProcessSQSRecord  --batch-size
 10 \
--event-source-arn arn:aws:sqs:us-east-1:111122223333:my-queue
```

Verwenden Sie den Befehl, um eine Liste Ihrer Ereignisquellenzuordnungen abzurufen. list-event-source-mappings Beispiel:

```
aws lambda list-event-source-mappings --function-name ProcessSQSRecord
```

Senden einer Testnachricht

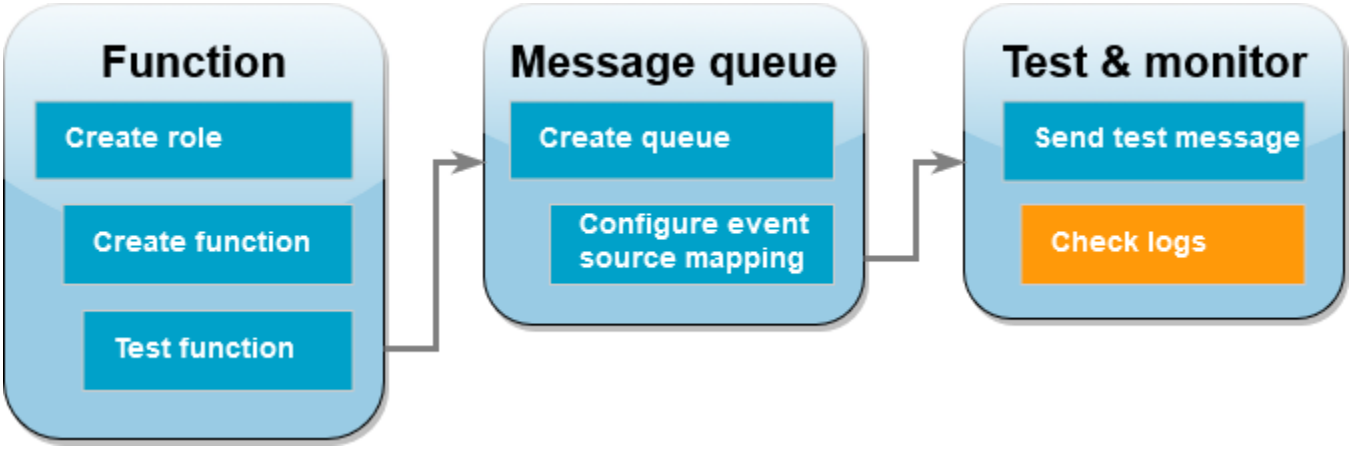


So senden Sie eine Amazon-SQS-Nachricht an die Lambda-Funktion

1. Öffnen Sie die Amazon-SQS-Konsole.
2. Wählen Sie die Warteschlange aus, die Sie zuvor erstellt haben.
3. Wählen Sie Nachrichten senden und empfangen.
4. Geben Sie unter Nachrichtentext eine Testnachricht ein (z. B. „Dies ist eine Testnachricht.“).
5. Klicken Sie auf Send Message (Nachricht senden).

Lambda fragt die Warteschlange nach Aktualisierungen ab. Wenn eine neue Nachricht vorliegt, ruft Lambda Ihre Funktion mit diesen neuen Ereignisdaten aus der Warteschlange auf. Wenn der Funktions-Handler ohne Ausnahmen zurückgegeben wird, betrachtet Lambda die Nachricht als erfolgreich verarbeitet und beginnt mit dem Lesen neuer Nachrichten in der Warteschlange. Nach erfolgreicher Verarbeitung einer Nachricht löscht Lambda diese automatisch aus der Warteschlange. Wenn der Handler eine Ausnahme auslöst, betrachtet Lambda den Nachrichten-Batch als nicht erfolgreich verarbeitet und Lambda ruft die Funktion mit demselben Nachrichten-Batch auf.

Überprüfen Sie die Protokolle CloudWatch



So vergewissern Sie sich, dass die Funktion die Nachricht verarbeitet hat

1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.
2. Wählen Sie die SQSRecord Funktion „Verarbeiten“.
3. Wählen Sie Überwachen aus.
4. Wählen Sie CloudWatch Protokolle anzeigen.
5. Wählen Sie in der CloudWatch Konsole den Protokollstream für die Funktion aus.
6. Suchen Sie das Protokoll `INFO`. Dort protokolliert die Lambda-Funktion den Nachrichtentext. Sie sollten die Nachricht sehen, die Sie über die Amazon-SQS-Warteschlange gesendet haben. Beispiel:

```
2023-09-11T22:49:12.730Z b0c41e9c-0556-5a8b-af83-43e59efeec71 INFO Processed
 message this is a test message.
```

Bereinigen Ihrer Ressourcen

Sie können jetzt die Ressourcen, die Sie für dieses Tutorial erstellt haben, löschen, es sei denn, Sie möchten sie behalten. Durch das Löschen von AWS Ressourcen, die Sie nicht mehr verwenden, vermeiden Sie unnötige Kosten für Ihre AWS-Konto.

So löschen Sie die Ausführungsrolle

1. Öffnen Sie die Seite Roles in der IAM-Konsole.
2. Wählen Sie die von Ihnen erstellte Ausführungsrolle aus.

3. Wählen Sie Löschen aus.
4. Geben Sie den Namen der Rolle in das Texteingabefeld ein und wählen Sie Delete (Löschen) aus.

So löschen Sie die Lambda-Funktion:

1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.
2. Wählen Sie die Funktion aus, die Sie erstellt haben.
3. Wählen Sie Aktionen, Löschen aus.
4. Geben Sie **`confirm`** in das Texteingabefeld ein und wählen Sie Delete (Löschen) aus.

So löschen Sie die Amazon-SQS-Warteschlange

1. Melden Sie sich bei der an AWS-Managementkonsole und öffnen Sie die Amazon SQS SQS-Konsole unter https://console.aws.amazon.com/sqs/.
2. Wählen Sie die Warteschlange aus, die Sie erstellt haben.
3. Wählen Sie Löschen aus.
4. Geben Sie **`confirm`** in das Texteingabefeld ein.
5. Wählen Sie Löschen aus.

Tutorial: Verwenden einer kontoübergreifenden Amazon SQS SQS-Warteschlange als Ereignisquelle

In diesem Tutorial erstellen Sie eine Lambda-Funktion, die Nachrichten aus einer Amazon Simple Queue Service (Amazon SQS) -Warteschlange in einem anderen Konto verarbeitet. AWS Dieses Tutorial umfasst zwei AWS Konten: Konto A bezieht sich auf das Konto, das Ihre Lambda-Funktion enthält, und Konto B bezieht sich auf das Konto, das die Amazon SQS SQS-Warteschlange enthält.

Voraussetzungen

Installieren Sie AWS Command Line Interface

Wenn Sie das noch nicht installiert haben AWS Command Line Interface, folgen Sie den Schritten unter Installieren oder Aktualisieren der neuesten Version von AWS CLI, um es zu installieren.

Das Tutorial erfordert zum Ausführen von Befehlen ein Befehlszeilenterminal oder eine Shell. Verwenden Sie unter Linux und macOS Ihre bevorzugte Shell und Ihren bevorzugten Paketmanager.

> **Note**
>
> In Windows werden einige Bash-CLI-Befehle, die Sie häufig mit Lambda verwenden (z. B. `zip`), von den integrierten Terminals des Betriebssystems nicht unterstützt. Um eine in Windows integrierte Version von Ubuntu und Bash zu erhalten, installieren Sie das Windows-Subsystem für Linux.

Erstellen der Ausführungsrolle (Konto A)

Erstellen Sie in Konto A eine Ausführungsrolle, die Ihrer Funktion den Zugriff auf die erforderlichen AWS Ressourcen gewährt.

So erstellen Sie eine Ausführungsrolle

1. Öffnen Sie die Seite Rollen in der AWS Identity and Access Management (IAM-) Konsole.
2. Wählen Sie Rolle erstellen aus.
3. Erstellen Sie eine Rolle mit den folgenden Eigenschaften.
   - Vertrauenswürdige Entität – AWS Lambda.
   - Berechtigungen – AWSLambdaSQSQueueExecutionRole
   - Role name (Name der Rolle – **`cross-account-lambda-sqs-role`**

Die AWSLambdaSQSQueueExecutionRoleRichtlinie verfügt über die Berechtigungen, die die Funktion zum Lesen von Elementen aus Amazon SQS und zum Schreiben von Protokollen in Amazon CloudWatch Logs benötigt.

Erstellen Sie die Funktion (Account A)

Erstellen Sie in Konto A eine Lambda-Funktion, die Ihre Amazon SQS-Nachrichten verarbeitet. Die Lambda-Funktion und die Amazon-SQS-Warteschlange müssen sich in demselben AWS-Region befinden.

Das folgende Codebeispiel für Node.js schreibt jede Nachricht in ein Protokoll in CloudWatch Logs.

Example index.mjs

```
export const handler = async function(event, context) {
  event.Records.forEach(record => {
    const { body } = record;
    console.log(body);
  });
  return {};
}
```

So erstellen Sie die Funktion

> **Note**
>
> Mit diesen Schritten wird eine Node.js-Funktion erstellt. Für andere Sprachen sind die Schritte ähnlich, aber einige Details unterscheiden sich.

1. Speichern Sie das Codebeispiel als Datei mit dem Namen `index.mjs`.
2. Erstellen Sie ein Bereitstellungspaket.

   ```
   zip function.zip index.mjs
   ```

3. Erstellen Sie die Funktion mit dem Befehl `create-function` AWS Command Line Interface (AWS CLI). Ersetzen Sie `arn:aws:iam::111122223333:role/cross-account-lambda-sqs-role` durch den ARN der Ausführungsrolle, die Sie zuvor erstellt haben.

   ```
   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
   ```

Testen Sie die Funktion (Konto A)

Testen Sie in Konto A Ihre Lambda-Funktion manuell mithilfe des `invoke` AWS CLI Befehls und eines Amazon SQS SQS-Beispielereignisses.

Wenn der Handler normal und ohne Ausnahmen zurückkehrt, betrachtet Lambda die Nachricht als erfolgreich verarbeitet und beginnt mit dem Lesen neuer Nachrichten in der Warteschlange. Nach erfolgreicher Verarbeitung einer Nachricht löscht Lambda diese automatisch aus der Warteschlange.

Wenn der Handler eine Ausnahme auslöst, betrachtet Lambda den Nachrichten-Batch als nicht erfolgreich verarbeitet und Lambda ruft die Funktion mit demselben Nachrichten-Batch auf.

1. Speichern Sie die folgende JSON als Datei mit dem Namen `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"
        }
    ]
}
```

   Das vorangehende JSON simuliert ein Ereignis, das Amazon SQS an Ihre Lambda-Funktion senden könnte, wobei `"body"` die tatsächliche Nachricht aus der Warteschlange enthält.

2. Führen Sie den Befehl `invoke` AWS CLI aus.

```
aws lambda invoke --function-name CrossAccountSQSExample \
--cli-binary-format raw-in-base64-out \
--payload file://input.txt outputfile.txt
```

   Die cli-binary-format Option ist erforderlich, wenn Sie AWS CLI Version 2 verwenden. Um dies zur Standardeinstellung zu machen, führen Sie `aws configure set cli-binary-format raw-in-base64-out` aus. Weitere Informationen finden Sie unter Von AWS CLI unterstützte globale Befehlszeilenoptionen im AWS Command Line Interface -Benutzerhandbuch für Version 2.

3. Überprüfen Sie die Ausgabe in der Datei `outputfile.txt`.

Erstellen einer Amazon SQS-Warteschlange (Konto B)

Erstellen Sie in Konto B eine Amazon SQS-Warteschlange, die die Lambda-Funktion in Konto A als Ereignisquelle verwenden kann. Die Lambda-Funktion und die Amazon-SQS-Warteschlange müssen sich in demselben AWS-Region befinden.

So erstellen Sie eine Warteschlange

1. Öffnen Sie die Amazon-SQS-Konsole.
2. Wählen Sie Create queue (Warteschlange erstellen) aus.
3. Erstellen Sie eine Warteschlange mit den folgenden Eigenschaften.
   - Typ – Standard
   - Name (Name – LambdaCrossAccountQueue
   - Konfiguration – Behalten Sie die Standardeinstellungen bei.
   - Zugriffsrichtlinie – Wählen Sie Advanced (Erweitert). Fügen Sie die folgende JSON-Richtlinie ein. Ersetzen Sie die folgenden Werte:
     - 111122223333: AWS-Konto ID für Konto A
     - 444455556666: AWS-Konto ID für Konto 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"
        }
    ]
```

```
}
```

Diese Richtlinie gewährt der Lambda-Ausführungsrolle in Konto A Berechtigungen zur Nutzung von Nachrichten aus dieser Amazon-SQS-Warteschlange.

4. Zeichnen Sie nach dem Erstellen der Warteschlange ihren Amazon-Ressourcennamen (ARN) auf. Sie benötigen ihn im nächsten Schritt, um die Warteschlange Ihrer Lambda-Funktion zuzuordnen.

Konfigurieren Sie die Ereignisquelle (Konto A)

Erstellen Sie in Konto A eine Ereignisquellenzuordnung zwischen der Amazon SQS SQS-Warteschlange in Konto B und Ihrer Lambda-Funktion, indem Sie den folgenden `create-event-source-mapping` AWS CLI Befehl ausführen. Ersetzen Sie `arn:aws:sqs:us-east-1:444455556666:LambdaCrossAccountQueue` durch den ARN der Amazon-SQS-Warteschlange, die Sie im vorherigen Schritt erstellt haben.

```
aws lambda create-event-source-mapping --function-name CrossAccountSQSExample --batch-size 10 \
--event-source-arn arn:aws:sqs:us-east-1:444455556666:LambdaCrossAccountQueue
```

Führen Sie den folgenden Befehl aus, um eine Liste Ihrer Ereignisquellen-Zuweisung abzurufen.

```
aws lambda list-event-source-mappings --function-name CrossAccountSQSExample \
--event-source-arn arn:aws:sqs:us-east-1:444455556666:LambdaCrossAccountQueue
```

Testen der Einrichtung

Sie können die Einrichtung nun wie folgt testen:

1. In Konto B öffnen Sie die Amazon-SQS-Konsole.
2. Wählen Sie LambdaCrossAccountQueue, was Sie zuvor erstellt haben.
3. Wählen Sie Nachrichten senden und empfangen.
4. Geben Sie unter Nachrichtentext eine Testnachricht ein.
5. Klicken Sie auf Send Message (Nachricht senden).

Ihre Lambda-FunktionKonto Asollte die Nachricht erhalten. Lambda wird die Warteschlange weiterhin nach Updates abfragen. Wenn eine neue Nachricht vorliegt, ruft Lambda Ihre Funktion mit diesen

neuen Ereignisdaten aus der Warteschlange auf. Ihre Funktion wird in Amazon ausgeführt und erstellt Protokolle CloudWatch. Sie können die Protokolle in der CloudWatch -Konsole ansehen.

Bereinigen Ihrer Ressourcen

Sie können jetzt die Ressourcen, die Sie für dieses Tutorial erstellt haben, löschen, es sei denn, Sie möchten sie behalten. Durch das Löschen von AWS Ressourcen, die Sie nicht mehr verwenden, vermeiden Sie unnötige Kosten für Ihre AWS-Konto.

In :Konto ABereinigen Sie Ihre Ausführungsrolle und Lambda-Funktion.

So löschen Sie die Ausführungsrolle

1. Öffnen Sie die Seite Roles in der IAM-Konsole.
2. Wählen Sie die von Ihnen erstellte Ausführungsrolle aus.
3. Wählen Sie Löschen aus.
4. Geben Sie den Namen der Rolle in das Texteingabefeld ein und wählen Sie Delete (Löschen) aus.

So löschen Sie die Lambda-Funktion:

1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.
2. Wählen Sie die Funktion aus, die Sie erstellt haben.
3. Wählen Sie Aktionen, Löschen aus.
4. Geben Sie **`confirm`** in das Texteingabefeld ein und wählen Sie Delete (Löschen) aus.

In :Konto BBereinigen Sie die Amazon SQS SQS-Warteschlange.

So löschen Sie die Amazon-SQS-Warteschlange

1. Melden Sie sich bei der an AWS-Managementkonsole und öffnen Sie die Amazon SQS SQS-Konsole unter https://console.aws.amazon.com/sqs/.
2. Wählen Sie die Warteschlange aus, die Sie erstellt haben.
3. Wählen Sie Löschen aus.
4. Geben Sie **`confirm`** in das Texteingabefeld ein.
5. Wählen Sie Löschen aus.

Orchestrierung von Lambda-Funktionen mit Step Functions

Lambda-Funktionen, die mehrere Aufgaben verwalten, Wiederholungslogik implementieren oder Verzweigungslogik enthalten, sind Anti-Muster. Stattdessen empfehlen wir, Lambda-Funktionen zu schreiben, die einzelne Aufgaben ausführen, und AWS Step Functions zur Orchestrierung Ihrer Anwendungsworkflows zu verwenden.

Beispielsweise kann die Bearbeitung eines Auftrags die Bestätigung der Auftragsdetails, die Überprüfung der Lagerbestände, die Bearbeitung der Zahlung und die Erstellung einer Rechnung erfordern. Schreiben Sie separate Lambda-Funktionen für jede Aufgabe und verwenden Sie Step Functions, um den Workflow zu verwalten. Step Functions koordiniert den Datenfluss zwischen Ihren Funktionen und behandelt Fehler bei jedem Schritt. Diese Trennung macht es einfacher, Ihre Workflows zu visualisieren, zu ändern und zu verwalten, wenn sie immer komplexer werden.

Wann sollte Step Functions mit Lambda verwendet werden?

Die folgenden Szenarien sind gute Beispiele dafür, wann Step Functions zur Orchestrierung von Lambda-basierten Anwendungen verwendet werden sollte.

- Sequenzielle Verarbeitung
- Komplexe Fehlerbehandlung
- Bedingte Workflows und Genehmigungen durch Menschen
- Parallele Verarbeitung

Sequenzielle Verarbeitung

Bei sequenzieller Verarbeitung muss eine Aufgabe abgeschlossen sein, bevor die nächste Aufgabe beginnen kann. In einem Auftragsverarbeitungssystem kann die Zahlungsabwicklung beispielsweise erst beginnen, wenn die Auftragsvalidierung abgeschlossen ist, und die Rechnungsstellung muss auf die Zahlungsbestätigung warten. Schreiben Sie für jede Aufgabe separate Lambda-Funktionen und verwenden Sie Step Functions, um die Reihenfolge zu verwalten und den Datenfluss zwischen den Funktionen zu steuern.

Beispiel für Anti-Muster

Eine einzige Lambda-Funktion verwaltet den gesamten Auftragsabwicklungsablauf durch Folgendes:

- Sequenzielles Aufrufen anderer Lambda-Funktionen

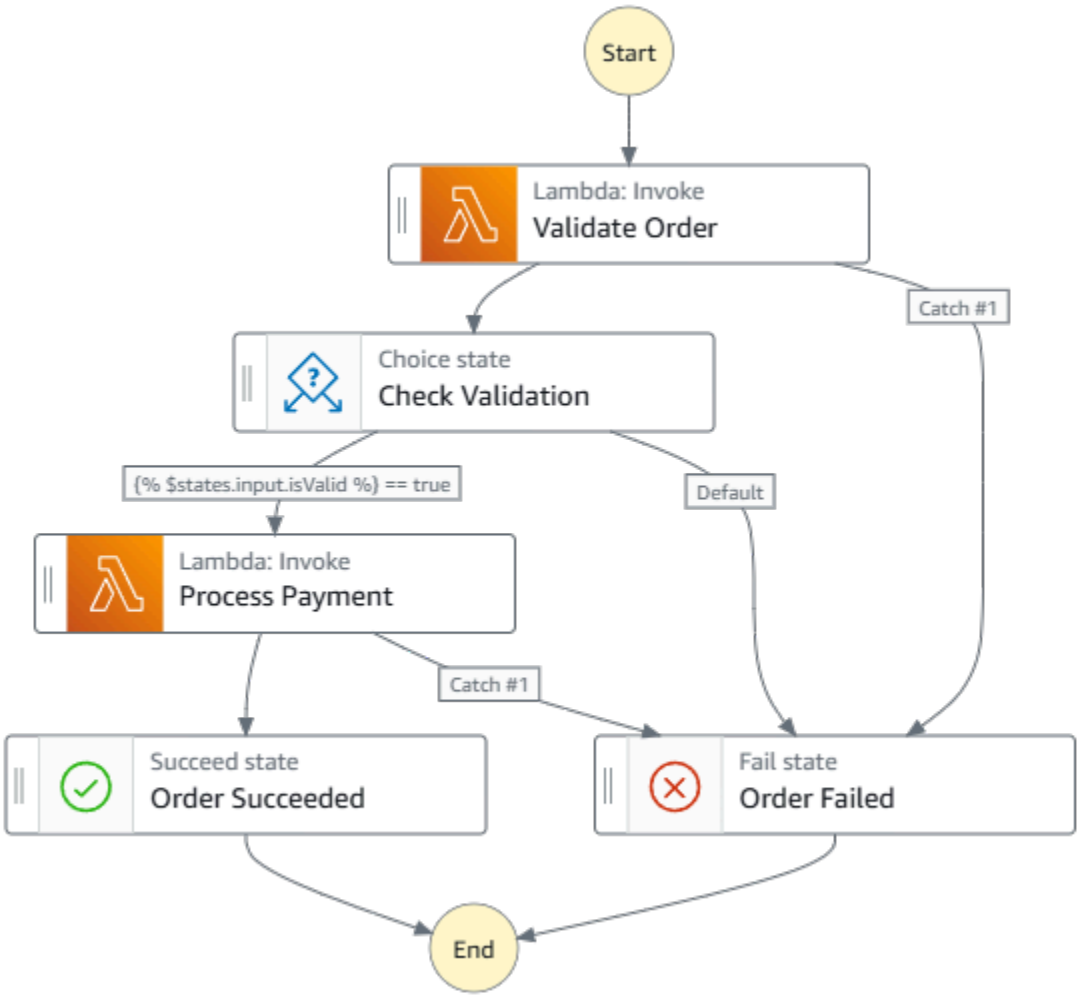
- Analysieren und Validieren der Antworten jeder Funktion
- Implementierung von Fehlerbehandlungs- und Wiederherstellungslogik
- Verwaltung des Datenflusses zwischen den Funktionen

Empfohlener Ansatz

Verwenden Sie zwei Lambda-Funktionen: eine zur Bestätigung des Auftrags und eine zur Zahlungsabwicklung. Step Functions koordiniert diese Funktionen durch Folgendes:

- Ausführung von Aufgaben in der richtigen Reihenfolge
- Weitergabe von Daten zwischen Funktionen
- Implementierung der Fehlerbehandlung bei jedem Schritt
- Verwendung von Choice-Zuständen, um sicherzustellen, dass nur gültige Aufträge zur Zahlung weitergeleitet werden

Example Workflow-Diagramm



Komplexe Fehlerbehandlung

Während Lambda Wiederholungsfunktionen für asynchrone Aufrufe und Zuordnungen von Ereignisquellen bietet, bietet Step Functions eine ausgefeiltere Fehlerbehandlung für komplexe Workflows. Sie können automatische Wiederholungsversuche mit exponentiellem Backoff konfigurieren und unterschiedliche Wiederholungsrichtlinien für verschiedene Fehlertypen festlegen. Wenn alle Wiederholungsversuche ausgeschöpft sind, verwenden Sie `Catch`, um Fehler in einen Fallback-Zustand umzuleiten. Dies ist besonders nützlich, wenn Sie eine Fehlerbehandlung auf Workflow-Ebene benötigen, die mehrere Funktionen und Services koordiniert.

Weitere Informationen zur Fehlerbehandlung von Lambda-Funktionen in einem Zustandsautomaten finden Sie im AWS Step Functions-Workshop unter Fehlerbehandlung.

Beispiel für Anti-Muster

Eine einzige Lambda-Funktion behandelt alle folgenden Aufgaben:

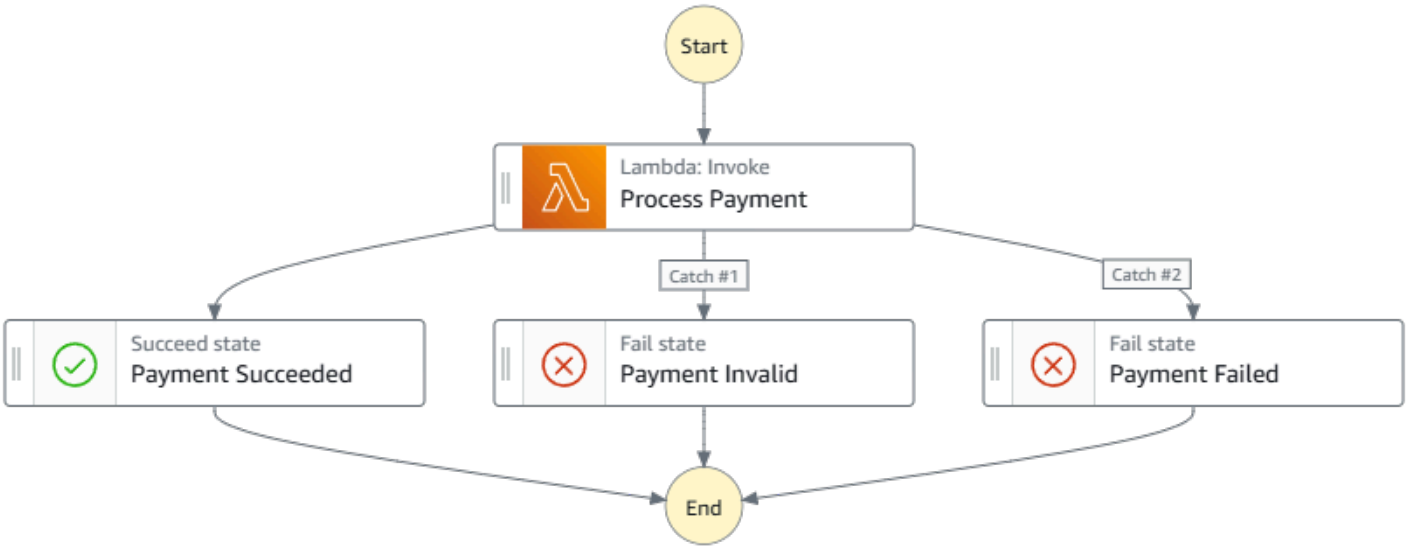
- Aufrufversuche eines Zahlungsabwicklungsdienstes
- Wenn der Zahlungsdienst nicht verfügbar ist, wartet die Funktion und versucht es später erneut.
- Implementierung eines benutzerdefinierten exponentiellen Backoffs für die Wartezeit
- Nachdem alle Versuche fehlgeschlagen sind, wird der Fehler abgefangen und ein anderer Ablauf gewählt.

Empfohlener Ansatz

Verwenden Sie eine einzige Lambda-Funktion, die sich ausschließlich auf die Zahlungsabwicklung konzentriert. Step Functions verwaltet die Fehlerbehandlung durch Folgendes:

- Automatische Wiederholung fehlgeschlagener Aufgaben mit konfigurierbaren Backoff-Zeiten
- Anwendung unterschiedlicher Wiederholungsrichtlinien je nach Fehlertyp
- Weiterleitung verschiedener Fehlertypen an geeignete Fallback-Zustände
- Beibehaltung des Status und des Verlaufs der Fehlerbehandlung

Example Workflow-Diagramm



Bedingte Workflows und Genehmigungen durch Menschen

Verwenden Sie den Choice-Zustand von Step Functions, um Workflows basierend auf der Funktionsausgabe weiterzuleiten, und das Suffix „waitForTaskToken“, um Workflows für menschliche Entscheidungen anzuhalten. Um beispielsweise eine Anfrage zur Erhöhung des Kreditlimits zu bearbeiten, verwenden Sie eine Lambda-Funktion, um Risikofaktoren zu bewerten. Verwenden Sie dann Step Functions, um Anfragen mit hohem Risiko an die manuelle Genehmigung und Anfragen mit geringem Risiko an die automatische Genehmigung weiterzuleiten.

Einen Beispiel-Workflow mit Integrationsmuster für einen Callback-Task-Token finden Sie im AWS Step Functions-Workshop unter Callback with Task Token.

Beispiel für Anti-Muster

Eine einzige Lambda-Funktion verwaltet einen komplexen Genehmigungsworkflow durch Folgendes:

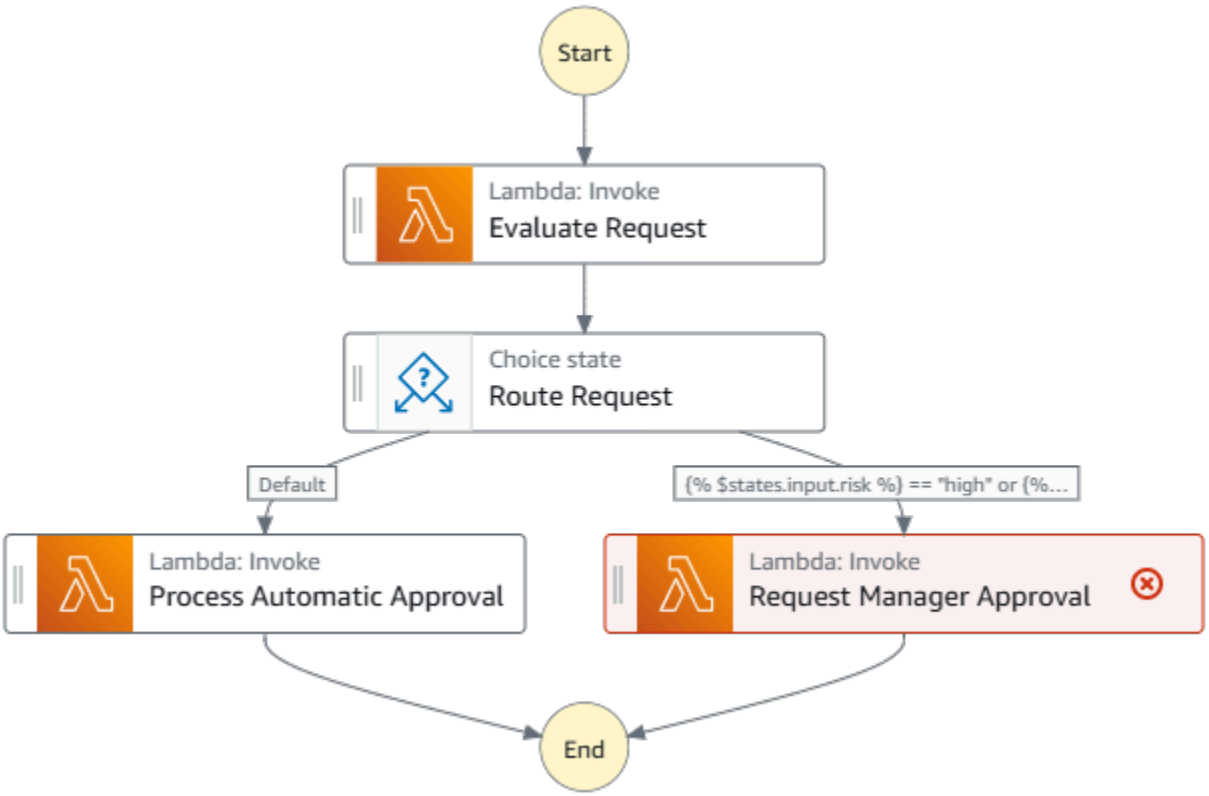
- Implementierung einer verschachtelten bedingten Logik zur Bewertung von Kreditanfragen
- Aufrufen verschiedener Genehmigungsfunktionen auf der Grundlage der Anzahl der Anfragen
- Verwaltung mehrerer Genehmigungspfade und Entscheidungspunkte
- Nachverfolgung des Status ausstehender Genehmigungen
- Implementierung einer Timeout- und Benachrichtigungslogik für Genehmigungen

Empfohlener Ansatz

Verwenden Sie drei Lambda-Funktionen: eine, um das Risiko jeder Anfrage zu bewerten, eine, um Anfragen mit geringem Risiko zu genehmigen, und eine, um Anfragen mit hohem Risiko zur Überprüfung an einen Manager weiterzuleiten. Step Functions steuert den Workflow durch:

- Verwendung von Choice-Zuständen zur Weiterleitung von Anfragen basierend auf Betrag und Risikostufe
- Pausieren der Ausführung bis zur Genehmigung durch einen Menschen
- Verwaltung von Timeouts für ausstehende Genehmigungen
- Bereitstellung von Einblicken in den aktuellen Status jeder Anfrage

Example Workflow-Diagramm



Parallele Verarbeitung

Step Functions bietet drei Möglichkeiten, die Parallelverarbeitung zu handhaben:

- Im parallelen Status werden mehrere Zweige Ihres Workflows gleichzeitig ausgeführt. Verwenden Sie diese Option, wenn Sie verschiedene Funktionen parallel ausführen müssen, z. B. das Generieren von Miniaturansichten beim Extrahieren von Bildmetadaten.
- Der Inline-Zuordnungszustand verarbeitet Datenarrays mit bis zu 40 gleichzeitigen Iterationen. Verwenden Sie dies für kleine bis mittlere Datensätze, bei denen Sie für jedes Element dieselbe Operation ausführen müssen.
- Der Zustand „Verteilte Karte“ verarbeitet umfangreiche Parallelverarbeitung mit bis zu 10 000 gleichzeitigen Ausführungen und unterstützt sowohl JSON-Arrays als auch Datenquellen von Amazon Simple Storage Service (Amazon S3). Verwenden Sie diese Option, wenn Sie große Datensätze verarbeiten oder wenn Sie eine höhere Gleichzeitigkeit benötigen.

Beispiel für Anti-Muster

Eine einzelne Lambda-Funktion versucht, die Parallelverarbeitung durch Folgendes zu verwalten:

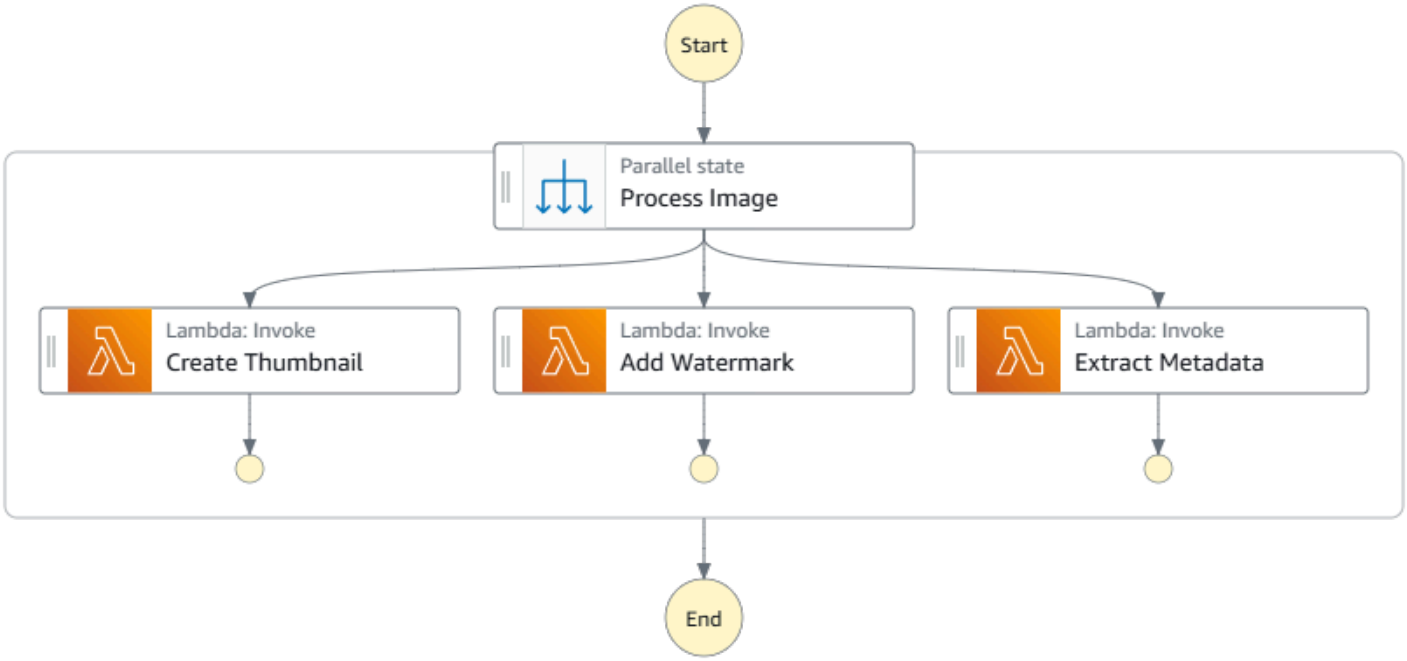
- Gleichzeitiges Aufrufen mehrerer Bildverarbeitungsfunktionen
- Implementierung einer benutzerdefinierten Logik für die parallel Ausführung
- Verwaltung von Timeouts und Fehlerbehandlung für jede parallele Aufgabe
- Sammeln und Aggregieren von Ergebnissen aus allen Funktionen

Empfohlener Ansatz

Verwenden Sie drei Lambda-Funktionen: eine zum Erstellen eines Miniaturbilds, eine zum Hinzufügen eines Wasserzeichens und eine zum Extrahieren der Metadaten. Step Functions verwaltet diese Funktionen wie folgt:

- Gleichzeitiges Ausführen aller Funktionen im parallelen Status
- Sammeln der Ergebnisse jeder Funktion in einem geordneten Array
- Verwaltung von Timeouts und Fehlerbehandlung bei allen parallelen Ausführungen
- Fortsetzung erst nach Abschluss aller parallelen Zweige

Example Workflow-Diagramm



Wann sollte Step Functions mit Lambda nicht verwendet werden?

Nicht alle Lambda-basierten Anwendungen profitieren von der Verwendung von Step Functions. Berücksichtigen Sie diese Szenarien bei der Auswahl Ihrer Anwendungsarchitektur.

- Einfache Anwendungen
- Komplexe Datenverarbeitung
- CPU-intensive Workloads

Einfache Anwendungen

Bei Anwendungen, die keine komplexe Orchestrierung erfordern, kann die Verwendung von Step Functions zu unnötiger Komplexität führen. Wenn Sie beispielsweise einfach Nachrichten aus einer Amazon-SQS-Warteschlange verarbeiten oder auf Amazon-EventBridge-Ereignisse reagieren, können Sie diese Services so konfigurieren, dass sie Ihre Lambda-Funktionen direkt aufrufen. Besteht Ihre Anwendung hingegen nur aus ein oder zwei Lambda-Funktionen mit unkomplizierter Fehlerbehandlung, so sind direkte Lambda-Aufrufe oder ereignisgesteuerte Architekturen möglicherweise einfacher bereitzustellen und zu verwalten.

Komplexe Datenverarbeitung

Mit dem Step-Functions-Zustand Verteilte Karte können Sie große Amazon-S3-Datensätze parallel mit Lambda-Funktionen verarbeiten. Dies ist für viele umfangreiche parallele Workloads effektiv, einschließlich der Verarbeitung halbstrukturierter Daten wie JSON- oder CSV-Dateien. Für komplexere Datentransformationen oder erweiterte Analytik sollten Sie jedoch die folgenden Alternativen in Betracht ziehen:

- Datentransformationspipelines: Verwenden Sie AWS Glue für ETL-Aufträge, die strukturierte oder halbstrukturierte Daten aus verschiedenen Quellen verarbeiten. AWS Glue ist besonders nützlich, wenn Sie integrierte Funktionen für Datenkatalog und Schemaverwaltung benötigen.
- Datenanalytik: Verwenden Sie Amazon EMR für Datenanalytik im Petabyte-Bereich, insbesondere wenn Sie Tools des Apache Hadoop-Ökosystems benötigen oder für Machine-Learning-Workloads, die die Speichergrenzen von Lambda überschreiten.

CPU-intensive Workloads

Step Functions kann zwar CPU-intensive Aufgaben orchestrieren, Lambda-Funktionen sind jedoch aufgrund ihrer begrenzten CPU-Ressourcen möglicherweise nicht für diese Workloads geeignet. Für rechenintensive Operationen innerhalb Ihrer Workflows sollten Sie die folgenden Alternativen in Betracht ziehen:

- Container-Orchestrierung: Verwenden Sie Step Functions zur Verwaltung von Aufgaben von Amazon Elastic Container Service (Amazon ECS) für konsistentere und skalierbarere Rechenressourcen.
- Batchverarbeitung: Integrieren Sie AWS Batch mit Step Functions zur Verwaltung rechenintensiver Batch-Aufträge, die eine dauerhafte CPU-Auslastung erfordern.

Aufrufen einer Lambda-Funktion mit Amazon-S3-Batchereignissen

Sie können Amazon-S3-Batchvorgänge verwenden, um eine Lambda-Funktion für einen großen Satz von Amazon-S3-Objekten aufzurufen. Amazon S3 verfolgt den Fortschritt von Batchvorgängen, sendet Benachrichtigungen und speichert einen Abschlussbericht, der den Status jeder Aktion anzeigt.

Zum Ausführen eines Batchvorgangs erstellen Sie einen Amazon-S3-Batchvorgangsauftrag. Wenn Sie den Auftrag erstellen, stellen Sie ein Manifest (die Liste der Objekte) bereit und konfigurieren die Aktion für diese Objekte.

Wenn der Batchauftrag gestartet wird, ruft Amazon S3 die Lambda-Funktion synchron für jedes Objekt im Manifest auf. Der Ereignis-Parameter enthält die Namen des Buckets und des Objekts.

Das folgende Beispiel zeigt das Ereignis, das Amazon S3 an die Lambda-Funktion für ein Objekt mit dem Namen customerImage1.jpg im Bucket amzn-s3-demo-bucket sendet.

Example Batch-Anforderungsereignis für 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"
        }
    ]
}
```

Ihre Lambda-Funktion muss ein JSON-Objekt mit den Feldern zurückgeben, wie im folgenden Beispiel gezeigt. Sie können den Parameter `invocationId` und `taskId` aus dem Ereignisparameter kopieren. Sie können eine Zeichenfolge in zurückgeben `resultString`. Amazon S3 speichert die `resultString`-Werte im Abschlussbericht.

Example Batch-Anforderungantwort für Amazon S3

```
{
  "invocationSchemaVersion": "1.0",
  "treatMissingKeysAs" : "PermanentFailure",
  "invocationId" : "YXNkbGZqYWRmaiBhc2RmdW9hZHNmZGpmaGFzbGtkaGZza2RmaAo",
  "results": [
    {
      "taskId": "dGFza2lkZ29lc2hlcmUK",
      "resultCode": "Succeeded",
      "resultString": "[\"Alice\", \"Bob\"]"
    }
  ]
}
```

Aufrufen von Lambda-Funktionen aus Amazon-S3-Batchvorägngen

Sie können die Lambda-Funktion mit einem nicht qualifizierten oder einem qualifizierten Funktions-ARN aufrufen. Wenn Sie dieselbe Funktionsversion für den gesamten Stapelauftrag verwenden möchten, konfigurieren Sie beim Erstellen des Auftrags eine bestimmte Funktionsversion im Parameter `FunctionARN`. Wenn Sie einen Alias oder den Qualifizierer $LATEST konfigurieren, beginnt der Stapelauftrag sofort, die neue Version der Funktion aufzurufen, wenn der Alias oder $LATEST während der Auftragsausführung aktualisiert wird.

Beachten Sie, dass Sie eine vorhandene ereignisbasierte Amazon-S3-Funktion nicht für Batchvorgänge verwenden können. Dies liegt daran, dass der Amazon-S3-Batchvorgang einen anderen Ereignisparameter an die Lambda-Funktion übergibt und eine Rückgabemeldung mit einer bestimmten JSON-Struktur erwartet.

Stellen Sie in der ressourcenbasierten Richtlinie, die Sie für den Amazon-S3-Batchauftrag erstellen, sicher, dass Sie die Berechtigung für den Auftrag zum Aufrufen der Lambda-Funktion festlegen.

Legen Sie in der Ausführungsrolle für die Funktion eine Vertrauensrichtlinie fest, damit Amazon S3 die Rolle übernimmt, wenn die Funktion ausgeführt wird.

Wenn Ihre Funktion das AWS SDK zum Verwalten von Amazon-S3-Ressourcen verwendet, müssen Sie der Ausführungsrolle Amazon-S3-Berechtigungen hinzufügen.

Wenn der Auftrag ausgeführt wird, startet Amazon S3 mehrere Funktionsinstances, um die Amazon-S3-Objekte bis zur Parallelitätsgrenze der Funktion parallel zu verarbeiten. Amazon S3 begrenzt den anfänglichen Hochlauf von Instances, um Überkosten für kleinere Aufträge zu vermeiden.

Wenn die Lambda-Funktion den Antwortcode `TemporaryFailure` zurückgibt, wiederholt Amazon S3 den Vorgang.

Weitere Informationen zu Amazon-S3-Stapelvorgängen finden Sie unter Durchführen von Stapelvorgängen im Amazon-S3-Entwicklerhandbuch.

Ein Beispiel für die Verwendung einer Lambda-Funktion in Amazon-S3-Stapelvorgängen finden Sie unter Aufrufen einer Lambda-Funktion von Amazon-S3-Stapelvorgängen im Amazon-S3-Entwicklerhandbuch.

Aufrufen von Lambda-Funktionen mit Amazon-SNS-Benachrichtigungen

Verwenden Sie eine Lambda-Funktion, um Amazon-Simple-Notification-Service-(Amazon-SNS)-Benachrichtigungen zu verarbeiten. Amazon SNS unterstützt Lambda-Funktionen als Ziel für Nachrichten, die an ein Thema gesendet werden. Sie können Ihre Funktion für Themen in demselben Konto oder in anderen AWS-Konten abonnieren. Eine ausführliche exemplarische Vorgehensweise finden Sie unter the section called “Tutorial”.

Lambda unterstützt SNS-Auslöser nur für standardmäßige SNS-Themen. FIFO-Themen werden nicht unterstützt.

Lambda verarbeitet SNS-Nachrichten asynchron, indem die Nachrichten in eine Warteschlange gestellt und Wiederholungsversuche verarbeitet werden. Wenn Amazon SNS Lambda nicht erreichen kann, oder die Nachricht abgelehnt wird, wiederholt Amazon SNS den Vorgang in zunehmenden Intervallen über mehrere Stunden. Weitere Details finden Sie unter Zuverlässigkeit in den häufig gestellten Fragen zu Amazon SNS.

> **Warning**
>
> Asynchrone Lambda-Aufrufe verarbeiten jedes Ereignis mindestens einmal und es kann zu einer doppelten Verarbeitung von Datensätzen kommen. Um mögliche Probleme im Zusammenhang mit doppelten Ereignissen zu vermeiden, empfehlen wir Ihnen dringend, Ihren Funktionscode idempotent zu machen. Weitere Informationen finden Sie unter Wie mache ich meine Lambda-Funktion idempotent im AWS-Wissenszentrum.

Idempotenz-Dienstprogramm von Powertools for AWS Lambda

Das Idempotenz-Dienstprogramm von Powertools for AWS Lambda macht Ihre Lambda-Funktionen idempotent. Es ist für Python, TypeScript, Java und .NET verfügbar. Weitere Informationen finden Sie unter Idempotency Utility in der Dokumentation zu Powertools for AWS Lambda (Python), Idempotency Utility in der Dokumentation zu Powertools for AWS Lambda (TypeScript), Idempotency Utility in der Dokumentation zu Powertools for AWS Lambda (Java) und Idempotency Utility in der Dokumentation zu Powertools for AWS Lambda (.NET).

Themen

- Hinzufügen eines Amazon-SNS-Themenauslösers für eine Lambda-Funktion mithilfe der Konsole
- Manuelles Hinzufügen eines Amazon-SNS-Themen-Auslösers für eine Lambda-Funktion
- Beispiel für eine SNS-Eventform
- Tutorial: Verwendung AWS Lambda mit Amazon Simple Notification Service

Hinzufügen eines Amazon-SNS-Themenauslösers für eine Lambda-Funktion mithilfe der Konsole

Um ein SNS-Thema als Auslöser für eine Lambda-Funktion hinzuzufügen, verwenden Sie am einfachsten die Lambda-Konsole. Wenn Sie den Auslöser über die Konsole hinzufügen, richtet Lambda automatisch die erforderlichen Berechtigungen und Abonnements ein, um mit dem Empfang von Ereignissen aus dem SNS-Thema zu beginnen.

So fügen Sie ein SNS-Thema als Auslöser für eine Lambda-Funktion hinzu (Konsole)

1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.
2. Wählen Sie den Namen einer Funktion, für die Sie den Auslöser hinzufügen möchten.
3. Wählen Sie Konfiguration und anschließend Auslöser aus.
4. Wählen Sie Add trigger.
5. Wählen Sie unter Auslöserkonfiguration im Dropdown-Menü die Option SNS aus.
6. Wählen Sie für das SNS-Thema das SNS-Thema, das Sie abonnieren möchten.

Manuelles Hinzufügen eines Amazon-SNS-Themen-Auslösers für eine Lambda-Funktion

Um einen SNS-Auslöser für eine Lambda-Funktion manuell einzurichten, müssen Sie die folgenden Schritte ausführen:

- Definieren Sie eine ressourcenbasierte Richtlinie für Ihre Funktion, damit SNS sie aufrufen kann.
- Abonnieren Sie Ihre Lambda-Funktion für das Amazon-SNS-Thema.

> **Note**
>
> Wenn sich Ihr SNS-Thema und Ihre Lambda-Funktion in verschiedenen AWS-Konten befinden, müssen Sie außerdem zusätzliche Berechtigungen erteilen, um

kontenübergreifende Abonnements für das SNS-Thema zu ermöglichen. Weitere Informationen finden Sie unter Kontoübergreifende Berechtigung für Amazon-SNS-Abonnement erteilen.

Sie können das AWS Command Line Interface (AWS CLI) verwenden, um diese beiden Schritte abzuschließen. Verwenden Sie zunächst den folgenden AWS CLI-Befehl, um eine ressourcenbasierte Richtlinie für eine Lambda-Funktion zu definieren, die SNS-Aufrufe zulässt. Ersetzen Sie den Wert von `--function-name` durch den Namen Ihrer Lambda-Funktion und den Wert von `--source-arn` durch den ARN Ihres SNS-Themas.

```
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
```

Verwenden Sie den folgenden AWS CLI-Befehl, um Ihre Funktion für das SNS-Thema zu abonnieren. Ersetzen Sie den Wert von `--topic-arn` durch Ihren SNS-Themen-ARN und den Wert von `--notification-endpoint` durch Ihren Lambda-Funktions-ARN.

```
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
```

Beispiel für eine SNS-Eventform

Amazon SNS ruft Ihre Funktion asynchron mit einem Ereignis auf, das eine Nachricht und Metadaten enthält.

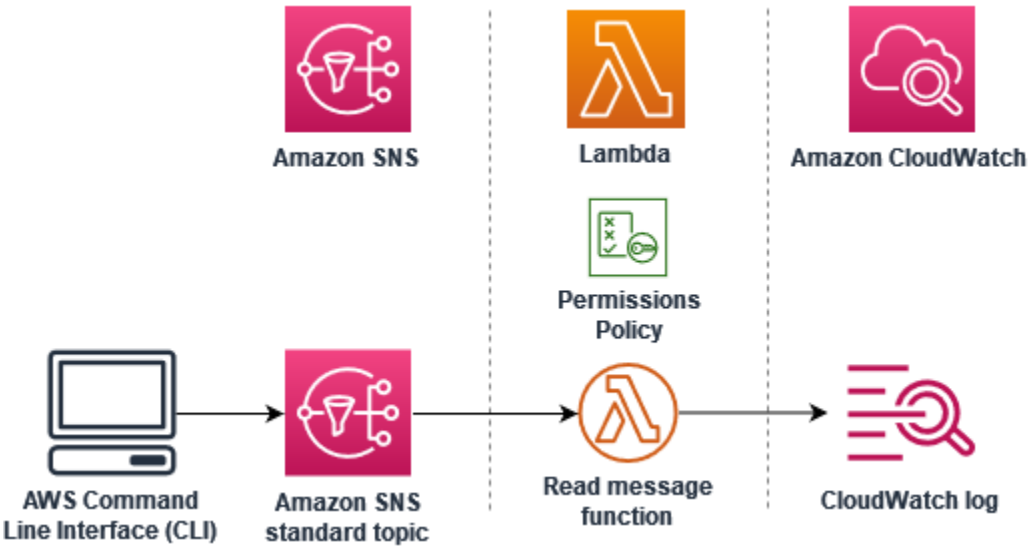
Example Amazon-SNS-Nachrichtenereignis

```
{
  "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"
      }
    }
  ]
}
```

Tutorial: Verwendung AWS Lambda mit Amazon Simple Notification Service

In diesem Tutorial verwenden Sie eine Lambda-Funktion in einem, AWS-Konto um ein Amazon Simple Notification Service (Amazon SNS) -Thema in einem separaten zu abonnieren. AWS-Konto Wenn Sie Nachrichten zu Ihrem Amazon SNS SNS-Thema veröffentlichen, liest Ihre Lambda-Funktion den Inhalt der Nachricht und gibt ihn in Amazon CloudWatch Logs aus. Um dieses Tutorial abzuschließen, verwenden Sie die AWS Command Line Interface ()AWS CLI.



Gehen Sie für dieses Tutorial wie folgt vor:

- Erstellen Sie in Konto A ein Amazon-SNS-Thema.
- Erstellen Sie in Konto B eine Lambda-Funktion, die Nachrichten aus dem Thema liest.
- Erstellen Sie in Konto B ein Abonnement für das Thema.
- Veröffentlichen Sie Nachrichten zum Amazon SNS SNS-Thema in Konto A und stellen Sie sicher, dass die Lambda-Funktion in Konto B sie in Logs ausgibt. CloudWatch

Anhand dieser Schritte lernen Sie, wie Sie ein Amazon-SNS-Thema konfigurieren, um eine Lambda-Funktion aufzurufen. Sie erfahren auch, wie Sie eine AWS Identity and Access Management (IAM-) Richtlinie erstellen, die einer Ressource in einer anderen die Erlaubnis erteilt AWS-Konto , Lambda aufzurufen.

In dem Tutorial werden zwei separate AWS-Konten verwendet. Die AWS CLI Befehle veranschaulichen dies anhand von zwei benannten Profilen`accountB`, die aufgerufen `accountA` und jeweils für die Verwendung mit einem anderen konfiguriert sind. AWS-Konto Informationen zur Konfiguration der AWS CLI Verwendung verschiedener Profile finden Sie unter Einstellungen für Konfiguration und Anmeldeinformationsdatei im AWS Command Line Interface Benutzerhandbuch für Version 2. Stellen Sie sicher, dass Sie AWS-Region für beide Profile dieselbe Standardeinstellung konfigurieren.

Wenn die AWS CLI Profile, die Sie für die beiden erstellen, unterschiedliche Namen AWS-Konten verwenden, oder wenn Sie das Standardprofil und ein benanntes Profil verwenden, ändern Sie die AWS CLI Befehle in den folgenden Schritten nach Bedarf.

Voraussetzungen

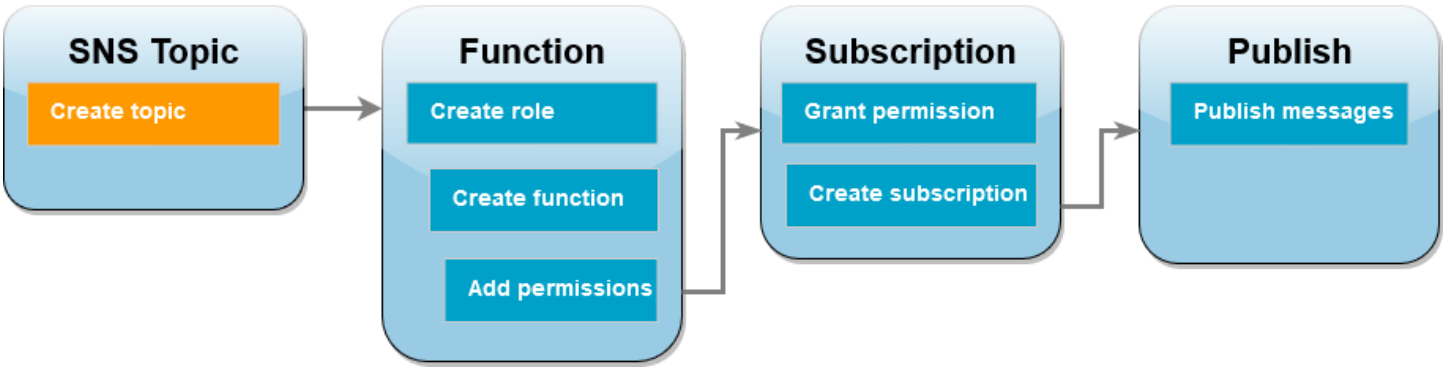
Installieren Sie das AWS Command Line Interface

Wenn Sie das noch nicht installiert haben AWS Command Line Interface, folgen Sie den Schritten unter Installieren oder Aktualisieren der neuesten Version von AWS CLI, um es zu installieren.

Das Tutorial erfordert zum Ausführen von Befehlen ein Befehlszeilenterminal oder eine Shell. Verwenden Sie unter Linux und macOS Ihre bevorzugte Shell und Ihren bevorzugten Paketmanager.

> **Note**
>
> In Windows werden einige Bash-CLI-Befehle, die Sie häufig mit Lambda verwenden (z. B. `zip`), von den integrierten Terminals des Betriebssystems nicht unterstützt. Um eine in Windows integrierte Version von Ubuntu und Bash zu erhalten, installieren Sie das Windows-Subsystem für Linux.

Erstellen eines Amazon-SNS-Themas (Konto A)



So erstellen Sie das -Thema

- Erstellen Sie in Konto A mit dem folgenden AWS CLI Befehl ein Amazon SNS SNS-Standardthema.

  ```
  aws sns create-topic --name sns-topic-for-lambda --profile accountA
  ```

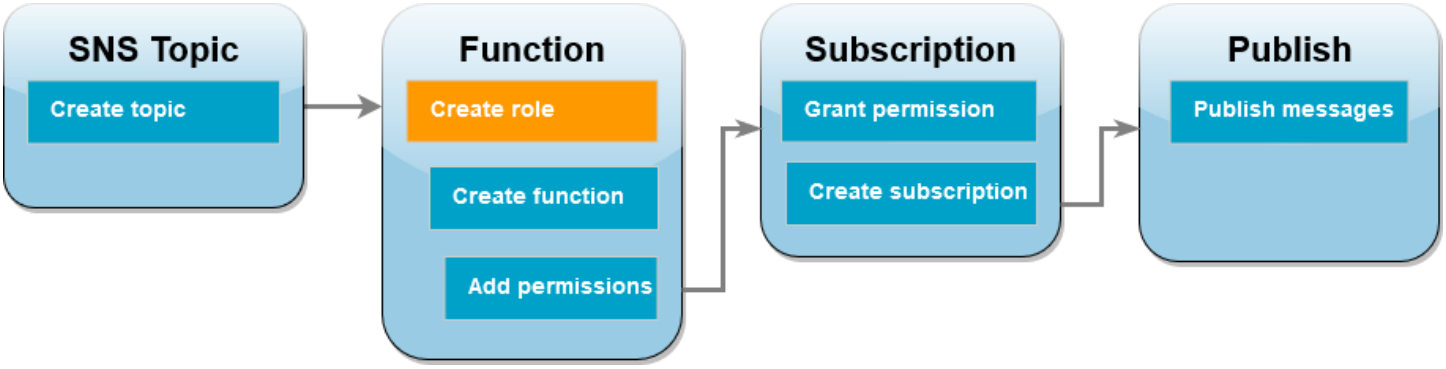
  Die Ausgabe sollte in etwa wie folgt aussehen:

  ```
  {
      "TopicArn": "arn:aws:sns:us-west-2:123456789012:sns-topic-for-lambda"
  ```

```
}
```

Notieren Sie sich den Amazon-Ressourcennamen (ARN) Ihres Themas. Sie benötigen ihn im weiteren Verlauf des Tutorials, wenn Sie Ihrer Lambda-Funktion Berechtigungen zum Abonnieren des Themas hinzufügen.

Erstellen einer Funktionsausführungsrolle (Konto B)



Eine Ausführungsrolle ist eine IAM-Rolle, die einer Lambda-Funktion Zugriff AWS-Services und Ressourcen gewährt. Bevor Sie Ihre Funktion in Konto B erstellen, erstellen Sie eine Rolle, die der Funktion grundlegende Berechtigungen zum Schreiben von Protokollen in Logs erteilt. CloudWatch Die Berechtigungen zum Lesen aus Ihrem Amazon-SNS-Thema werden in einem späteren Schritt hinzugefügt.

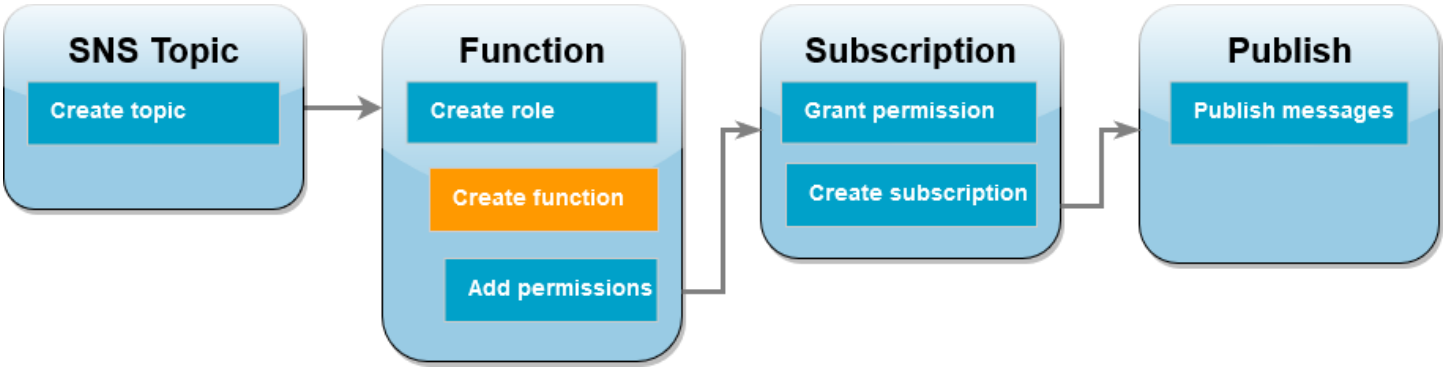
So erstellen Sie eine Ausführungsrolle

1. Öffnen Sie in Konto B die Seite Rollen in der IAM-Konsole.
2. Wählen Sie Rolle erstellen aus.
3. Wählen Sie unter Vertrauenswürdiger Entitätstyp die Option AWS -Service aus.
4. Wählen Sie unter Anwendungsfall die Option Lambda aus.
5. Wählen Sie Weiter aus.
6. Fügen Sie der Rolle wie folgt eine Richtlinie mit grundlegenden Berechtigungen hinzu:
   a. Geben Sie im Suchfeld Berechtigungsrichtlinien die Zeichenfolge **`AWSLambdaBasicExecutionRole`** ein.
   b. Wählen Sie Weiter aus.
7. Schließen Sie die Rollenerstellung ab:

a. Geben Sie unter Rollendetails im Feld Rollenname den Namen **`lambda-sns-role`** ein.

b. Wählen Sie Rolle erstellen aus.

Erstellen einer Lambda-Funktion (Konto B)



Erstellen Sie eine Lambda-Funktion, die Ihre Amazon-SNS-Nachrichten verarbeitet. Der Funktionscode protokolliert den Nachrichteninhalt jedes Datensatzes in Amazon CloudWatch Logs.

Dieses Tutorial verwendet die Runtime Node.js 24, aber wir haben auch Beispielcode in anderen Runtime-Sprachen bereitgestellt. Sie können die Registerkarte im folgenden Feld auswählen, um Code für die gewünschte Laufzeit anzusehen. Der JavaScript Code, den Sie in diesem Schritt verwenden, befindet sich im ersten Beispiel, das auf der JavaScriptRegisterkarte angezeigt wird.

.NET

SDK für .NET

> **Note**
>
> Es gibt noch mehr dazu GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines SNS-Ereignisses mit Lambda unter Verwendung von .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

SDK für Go V2

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines SNS-Ereignisses mit Lambda unter Verwendung von 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 für Java 2.x

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines SNS-Ereignisses mit Lambda unter Verwendung von 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 für JavaScript (v3)

> **Note**
>
> Es gibt noch mehr dazu GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Konsumieren eines SNS-Ereignisses mit Lambda unter Verwendung. 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) {
    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;
  }
}
```

Konsumieren eines SNS-Ereignisses mit Lambda unter Verwendung. TypeScript

```
// 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

SDK für PHP

> **Note**
>
> Es gibt noch mehr dazu. GitHub Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines SNS-Ereignisses mit Lambda unter Verwendung von 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

SDK für Python (Boto3)

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines SNS-Ereignisses mit Lambda unter Verwendung von 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

SDK für Ruby

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines SNS-Ereignisses mit Lambda unter Verwendung von 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 für Rust

> **Note**
>
> Es gibt noch mehr dazu GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines SNS-Ereignisses mit Lambda unter Verwendung von 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
}
```

So erstellen Sie die Funktion

1. Erstellen Sie ein Verzeichnis für das Projekt und wechseln Sie dann zu diesem Verzeichnis.

   ```
   mkdir sns-tutorial
   cd sns-tutorial
   ```

2. Kopieren Sie den JavaScript Beispielcode in eine neue Datei mit dem Namen`index.js`.
3. Erstellen Sie ein Bereitstellungspaket mit dem folgenden `zip`-Befehl.

```
zip function.zip index.js
```

4. Führen Sie den folgenden AWS CLI Befehl aus, um Ihre Lambda-Funktion in Konto B zu erstellen.

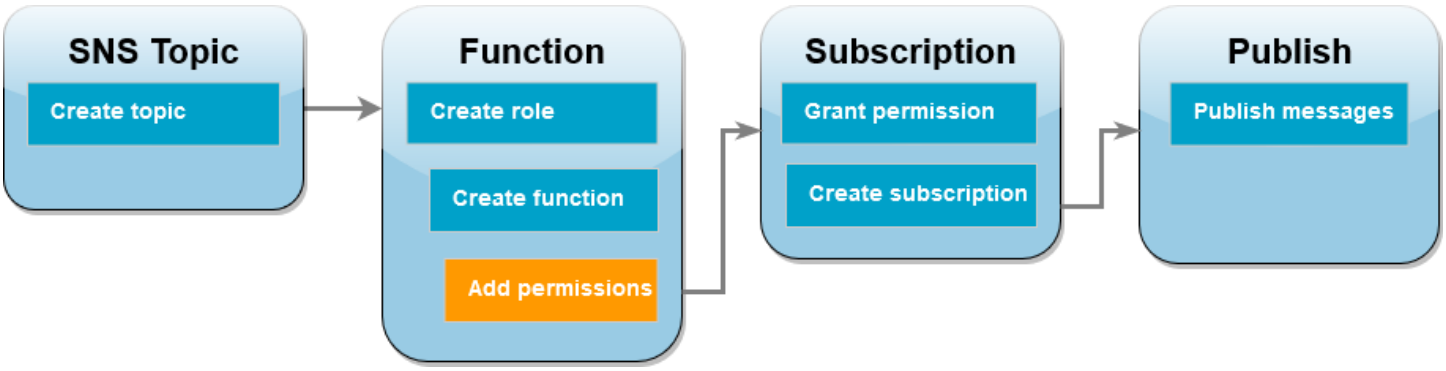
```
aws lambda create-function --function-name Function-With-SNS \
    --zip-file fileb://function.zip --handler index.handler --runtime nodejs24.x \
    --role arn:aws:iam::<AccountB_ID>:role/lambda-sns-role  \
    --timeout 60 --profile accountB
```

Die Ausgabe sollte in etwa wie folgt aussehen:

```
{
    "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. Notieren Sie sich den Amazon-Ressourcennamen (ARN) Ihrer Funktion. Sie benötigen ihn im weiteren Verlauf des Tutorials, wenn Sie Berechtigungen hinzufügen, um Amazon SNS das Aufrufen Ihrer Funktion zu ermöglichen.

Hinzufügen von Berechtigungen zur Funktion (Konto B)



Damit Amazon SNS Ihre Funktion aufrufen kann, muss die dafür notwendige Berechtigung in einer Anweisung einer ressourcenbasierten Richtlinie erteilt werden. Sie fügen diese Anweisung mit dem AWS CLI `add-permission` Befehl hinzu.

Erteilen der Berechtigung zum Aufrufen Ihrer Funktion durch Amazon SNS

- Führen Sie in Konto B den folgenden AWS CLI Befehl mit dem ARN für Ihr Amazon SNS SNS-Thema aus, das Sie zuvor aufgezeichnet haben.

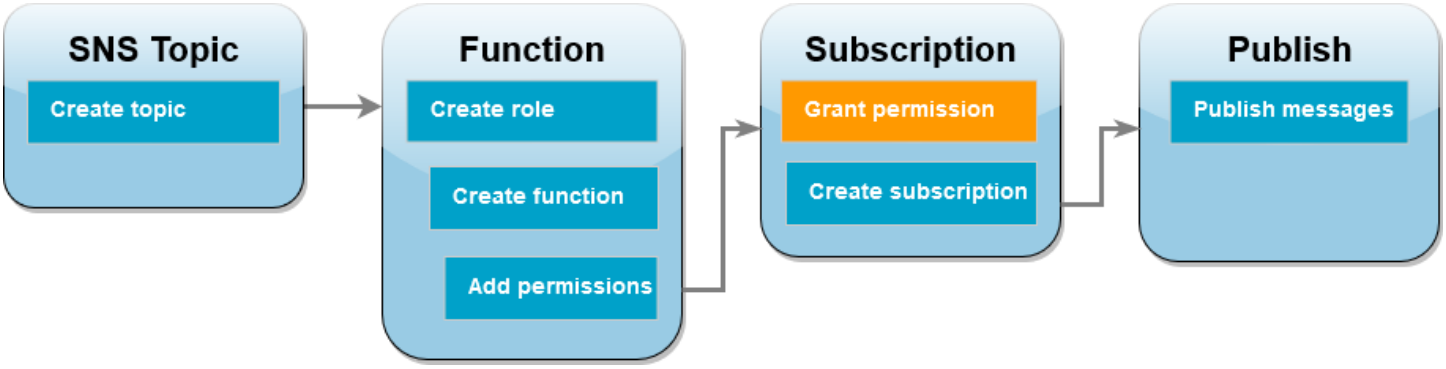
```
aws lambda add-permission --function-name Function-With-SNS \
    --source-arn arn:aws:sns:us-east-1:<AccountA_ID>:sns-topic-for-lambda \
    --statement-id function-with-sns --action "lambda:InvokeFunction" \
    --principal sns.amazonaws.com --profile accountB
```

Die Ausgabe sollte in etwa wie folgt aussehen:

```
{
    "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**
>
> Wenn das Konto mit dem Amazon SNS SNS-Thema in einem Opt-In gehostet wird AWS-Region, müssen Sie die Region im Prinzipal angeben. Wenn Sie beispielsweise mit einem Amazon-SNS-Thema in der Region Asien-Pazifik (Hongkong) arbeiten, muss für den Prinzipal `sns.ap-east-1.amazonaws.com` anstelle von `sns.amazonaws.com` angegeben werden.

Erteilen einer kontoübergreifenden Berechtigung für das Amazon-SNS-Abonnement (Konto A)



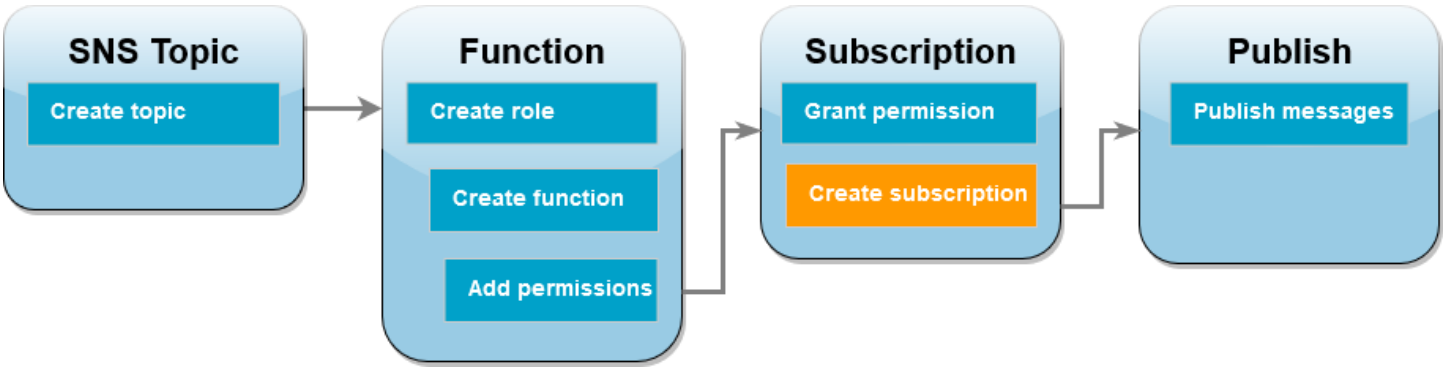
Damit Ihre Lambda-Funktion in Konto B das Amazon-SNS-Thema abonnieren kann, das Sie in Konto A erstellt haben, müssen Sie Konto B die Berechtigung zum Abonnieren Ihres Themas erteilen. Sie erteilen diese Berechtigung mit dem AWS CLI `add-permission` Befehl.

Erteilen der Berechtigung zum Abonnieren des Themas durch Konto B

- Führen Sie in Konto A den folgenden AWS CLI Befehl aus. Verwenden Sie den ARN für das Amazon-SNS-Thema, den Sie sich zuvor notiert haben.

```
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
```

Erstellen eines Abonnements (Konto B)



In Konto B abonnieren Sie nun für Ihre Lambda-Funktion das Amazon-SNS-Thema, das Sie zu Beginn des Tutorials in Konto A erstellt haben. Wenn eine Nachricht an dieses Thema (`sns-topic-for-lambda`) gesendet wird, ruft Amazon SNS Ihre Lambda-Funktion `Function-With-SNS` in Konto B auf.

Erstellen eines Abonnements

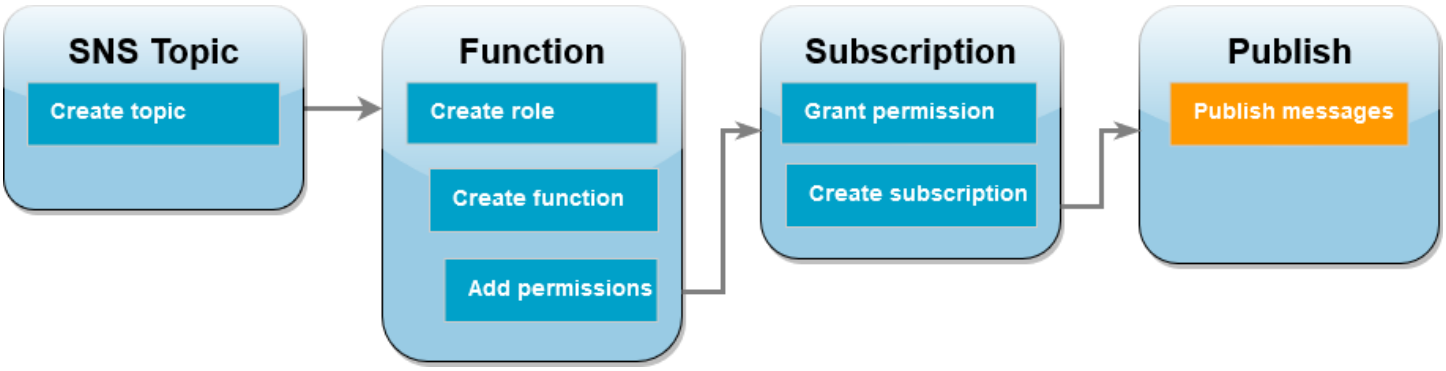
- Führen Sie in Konto B den folgenden AWS CLI Befehl aus. Verwenden Sie Ihre Standardregion, in der Sie Ihr Thema erstellt haben, und die ARNs für Ihr Thema und die Lambda-Funktion.

```
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
```

  Die Ausgabe sollte in etwa wie folgt aussehen:

```
{
    "SubscriptionArn": "arn:aws:sns:us-east-1:<AccountA_ID>:sns-topic-for-
lambda:5d906xxxx-7c8x-45dx-a9dx-0484e31c98xx"
}
```

Veröffentlichen von Nachrichten für das Thema (Konto A und Konto B)



Ihre Lambda-Funktion in Konto B hat nun Ihr Amazon-SNS-Thema in Konto A abonniert und Sie können Ihr Setup testen, indem Sie Nachrichten für das Thema veröffentlichen. Um zu bestätigen, dass Amazon SNS Ihre Lambda-Funktion aufgerufen hat, verwenden Sie CloudWatch Logs, um die Ausgabe Ihrer Funktion anzusehen.

Veröffentlichen einer Nachricht für Ihr Thema und Anzeigen der Ausgabe Ihrer Funktion

1. Geben Sie `Hello World` in eine Textdatei ein und speichern Sie sie als `message.txt`.
2. Führen Sie aus demselben Verzeichnis, in dem Sie Ihre Textdatei gespeichert haben, den folgenden AWS CLI Befehl in Konto A aus. Verwenden Sie den ARN für Ihr eigenes Thema.

   ```
   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
   ```

   Dadurch wird eine eindeutige Nachrichten-ID zurückgegeben, die angibt, dass Amazon SNS die Nachricht akzeptiert hat. Anschließend versucht Amazon SNS, die Nachricht den Abonnenten des Themas zuzustellen. Um zu bestätigen, dass Amazon SNS Ihre Lambda-Funktion aufgerufen hat, verwenden Sie CloudWatch Logs, um die Ausgabe Ihrer Funktion einzusehen:
3. Öffnen Sie in Konto B die Seite Protokollgruppen der CloudWatch Amazon-Konsole.
4. Wählen Sie die Protokollgruppe für Ihre Funktion (`/aws/lambda/Function-With-SNS`) aus.
5. Wählen Sie den neuesten Protokollstreams aus.
6. Wenn Ihre Funktion korrekt aufgerufen wurde, sieht die Ausgabe in etwa wie folgt aus und enthält die Inhalte der Nachricht, die Sie für Ihr Thema veröffentlicht haben.

   ```
   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
```

Bereinigen Ihrer Ressourcen

Sie können jetzt die Ressourcen, die Sie für dieses Tutorial erstellt haben, löschen, es sei denn, Sie möchten sie behalten. Durch das Löschen von AWS Ressourcen, die Sie nicht mehr verwenden, vermeiden Sie unnötige Kosten für Ihre AWS-Konto.

Bereinigen Sie über Konto A Ihr Amazon-SNS-Thema.

So löschen Sie das Amazon-SNS-Thema

1. Öffnen Sie die Seite Themen der Amazon-SNS-Konsole.
2. Wählen Sie das Thema aus, das Sie gerade erstellt haben.
3. Wählen Sie Löschen aus.
4. Geben Sie **`delete me`** in das Texteingabefeld ein.
5. Wählen Sie Löschen aus.

Bereinigen Sie über Konto B Ihre Ausführungsrolle, die Lambda-Funktion und das Amazon-SNS-Abonnement.

So löschen Sie die Ausführungsrolle

1. Öffnen Sie die Seite Roles in der IAM-Konsole.
2. Wählen Sie die von Ihnen erstellte Ausführungsrolle aus.
3. Wählen Sie Löschen aus.
4. Geben Sie den Namen der Rolle in das Texteingabefeld ein und wählen Sie Delete (Löschen) aus.

So löschen Sie die Lambda-Funktion:

1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.
2. Wählen Sie die Funktion aus, die Sie erstellt haben.
3. Wählen Sie Aktionen, Löschen aus.
4. Geben Sie **`confirm`** in das Texteingabefeld ein und wählen Sie Delete (Löschen) aus.

So löschen Sie das Amazon-SNS-Abonnement

1. Öffnen Sie die Seite Abonnements der Amazon-SNS-Konsole.
2. Wählen Sie das von Ihnen erstellte Abonnement aus.
3. Wählen Sie Löschen, Löschen aus.

Verwaltung von Berechtigungen in AWS Lambda

Sie können AWS Identity and Access Management (IAM) verwenden, um Berechtigungen in AWS Lambda zu verwalten. Es gibt zwei Hauptkategorien von Berechtigungen, die Sie bei der Arbeit mit Lambda-Funktionen berücksichtigen müssen:

- Berechtigungen, die Ihre Lambda-Funktionen benötigen, um API-Aktionen auszuführen und auf andere AWS Ressourcen zuzugreifen
- Berechtigungen, die andere AWS Benutzer und Entitäten für den Zugriff auf Ihre Lambda-Funktionen benötigen

Lambda-Funktionen müssen häufig auf andere AWS Ressourcen zugreifen und verschiedene API-Operationen mit diesen Ressourcen ausführen. Sie verfügen beispielsweise möglicherweise über eine Lambda-Funktion, die auf ein Ereignis reagiert, indem sie Einträge in einer Amazon-DynamoDB-Datenbank aktualisiert. In diesem Fall benötigt Ihre Funktion Berechtigungen für den Zugriff auf die Datenbank sowie Berechtigungen zum Ablegen oder Aktualisieren von Elementen in dieser Datenbank.

Sie definieren die Berechtigungen, die Ihre Lambda-Funktion benötigt, in einer speziellen IAM-Rolle, der sogenannten Ausführungsrolle. In dieser Rolle können Sie eine Richtlinie anhängen, die alle Berechtigungen definiert, die Ihre Funktion für den Zugriff auf andere AWS Ressourcen und das Lesen aus Ereignisquellen benötigt. Jede Lambda-Funktion muss eine Ausführungsrolle haben. Ihre Ausführungsrolle muss mindestens Zugriff auf Amazon haben, CloudWatch da Lambda-Funktionen standardmäßig in CloudWatch Logs protokolliert werden. Sie können die `AWSLambdaBasicExecutionRole-verwaltete Richtlinie` an Ihre Ausführungsrolle anhängen, um diese Anforderung zu erfüllen.

Um anderen AWS-Konten Organisationen und Diensten Zugriff auf Ihre Lambda-Ressourcen zu gewähren, haben Sie mehrere Möglichkeiten:

- Sie können identitätsbasierte Richtlinien verwenden, um anderen Benutzern Zugriff auf Ihre Lambda-Ressourcen zu gewähren. Identitätsbasierte Richtlinien können direkt auf Benutzer oder auf Gruppen und Rollen angewendet werden, die einem Benutzer zugeordnet sind.
- Sie können ressourcenbasierte Richtlinien verwenden, um anderen Konten und AWS-Services Berechtigungen für den Zugriff auf Ihre Lambda-Ressourcen zu gewähren. Wenn ein Benutzer versucht, auf eine Lambda-Ressource zuzugreifen, berücksichtigt Lambda sowohl die identitätsbasierten Richtlinien des Benutzers als auch die ressourcenbasierte Richtlinie der

Ressource. Wenn ein AWS Service wie Amazon Simple Storage Service (Amazon S3) Ihre Lambda-Funktion aufruft, berücksichtigt Lambda nur die ressourcenbasierte Richtlinie.

- Sie können ein attributbasiertes Zugriffskontrollmodell (ABAC) verwenden, um den Zugriff auf Ihre Lambda-Funktionen zu steuern. Mit ABAC können Sie Tags an eine Lambda-Funktion anhängen, sie in bestimmten API-Anforderungen übergeben oder sie an den IAM-Prinzipal anfügen, der die Anforderung stellt. Geben Sie dieselben Tags im Bedingungselement einer IAM-Richtlinie an, um den Funktionszugriff zu steuern.

In ist es eine bewährte Methode AWS, nur die Berechtigungen zu gewähren, die für die Ausführung einer Aufgabe erforderlich sind (Berechtigungen mit den geringsten Rechten). Um dies in Lambda zu implementieren, empfehlen wir, mit einer AWS -verwalteten Richtlinie zu beginnen. Sie können diese verwalteten Richtlinien unverändert oder als Ausgangspunkt zum Schreiben Ihrer eigenen restriktiveren Richtlinien verwenden.

Um Ihnen bei der Optimierung Ihrer Berechtigungen für den Zugriff mit den geringsten Rechten zu helfen, bietet Lambda einige zusätzliche Bedingungen, die Sie in Ihre Richtlinien aufnehmen können. Weitere Informationen finden Sie unter the section called “Ressourcen und Bedingungen”.

Weitere Informationen zu IAM finden Sie im IAM-Benutzerhandbuch.

Definieren von Lambda-Funktionsberechtigungen mit einer Ausführungsrolle

Die Ausführungsrolle einer Lambda-Funktion ist eine AWS Identity and Access Management (IAM-) Rolle, die der Funktion Zugriff AWS-Services und Ressourcen gewährt. Sie könnten beispielsweise eine Ausführungsrolle erstellen, die berechtigt ist, Protokolle an Amazon zu senden CloudWatch und Trace-Daten hochzuladen AWS X-Ray. Diese Seite enthält Informationen zum Erstellen, Anzeigen und Verwalten der Ausführungsrolle einer Lambda-Funktion.

Lambda übernimmt automatisch Ihre Ausführungsrolle an, wenn Sie Ihre Funktion aufrufen. Sie sollten vermeiden, `sts:AssumeRole` manuell aufzurufen, um die Ausführungsrolle in Ihrem Funktionscode zu übernehmen. Wenn Ihr Anwendungsfall erfordert, dass die Rolle sich selbst annimmt, müssen Sie die Rolle selbst als vertrauenswürdigen Prinzipal in die Vertrauensrichtlinie Ihrer Rolle aufnehmen. Weitere Informationen zum Ändern einer Rollenvertrauensrichtlinie finden Sie unter Ändern einer Rollenvertrauensrichtlinie (Konsole) im IAM-Benutzerhandbuch.

Damit Lambda Ihre Ausführungsrolle ordnungsgemäß übernehmen kann, muss in der Vertrauensrichtlinie der Rolle der Lambda-Serviceprinzipal (`lambda.amazonaws.com`) als vertrauenswürdiger Service angegeben sein.

Themen

- Erstellen einer Ausführungsrolle in der IAM-Konsole
- Rollen erstellen und verwalten mit dem AWS CLI
- Gewähren Sie den Zugriff auf Ihre Lambda-Ausführungsrolle mit den geringsten Berechtigungen
- Berechtigungen in der Ausführungsrolle anzeigen und aktualisieren
- In der Ausführungsrolle mit AWS verwalteten Richtlinien arbeiten
- Verwendung der Quellfunktion ARN zur Steuerung des Funktionszugriffsverhaltens

Erstellen einer Ausführungsrolle in der IAM-Konsole

Standardmäßig erstellt Lambda beim Erstellen einer Funktion in der Lambda-Konsole eine Ausführungsrolle mit minimalen Berechtigungen. Insbesondere umfasst diese Ausführungsrolle die `AWSLambdaBasicExecutionRole`verwaltete Richtlinie, die Ihrer Funktion grundlegende Berechtigungen zum Protokollieren von Ereignissen in Amazon CloudWatch Logs erteilt.

Ihre Funktionen benötigen in der Regel zusätzliche Berechtigungen, um aussagekräftigere Aufgaben ausführen zu können. Sie verfügen beispielsweise möglicherweise über eine Lambda-Funktion, die auf ein Ereignis reagiert, indem sie Einträge in einer Amazon-DynamoDB-Datenbank aktualisiert. Sie können über die IAM-Konsole eine Ausführungsrolle mit den erforderlichen Berechtigungen erstellen.

So erstellen Sie eine Ausführungsrolle in der IAM-Konsole

1. Öffnen Sie die Seite Roles (Rollen) in der IAM-Konsole.
2. Wählen Sie Rolle erstellen aus.
3. Wählen Sie unter Typ der vertrauenswürdigen Entität die Option AWS -Service aus.
4. Wählen Sie unter Anwendungsfall Lambda aus.
5. Wählen Sie Weiter aus.
6. Wählen Sie die AWS verwalteten Richtlinien aus, die Sie Ihrer Rolle zuordnen möchten. Wenn Ihre Funktion beispielsweise auf DynamoDB zugreifen muss, wählen Sie die verwaltete Richtlinie AWSLambdaDynamo DBExecution Role aus.
7. Wählen Sie Weiter aus.
8. Geben Sie einen Role name (Rollennamen) ein und klicken Sie auf Create Role (Rolle erstellen).

Ausführliche Anweisungen finden Sie unter Erstellen einer Rolle für einen AWS Dienst (Konsole) im IAM-Benutzerhandbuch.

Nachdem Sie Ihre Ausführungsrolle erstellt haben, fügen Sie sie an Ihre Funktion an. Wenn Sie eine Funktion in der Lambda-Konsole erstellen, können Sie eine beliebige Ausführungsrolle, die Sie zuvor erstellt haben, an die Funktion anhängen. Wenn Sie einer vorhandenen Funktion eine neue Ausführungsrolle zuordnen möchten, folgen Sie den Schritten unter the section called “Aktualisierung der Ausführungsrolle einer Funktion”.

Rollen erstellen und verwalten mit dem AWS CLI

Um eine Ausführungsrolle mit dem AWS Command Line Interface (AWS CLI) zu erstellen, verwenden Sie den create-role Befehl. Wenn Sie diesen Befehl verwenden, können Sie die Vertrauensrichtlinie angeben. Über die Vertrauensrichtlinie einer Rolle wird den angegebenen Prinzipalen die Berechtigung gegeben, die Rolle zu übernehmen. Im folgenden Beispiel erteilen Sie dem Lambda-Serviceprinzipal die Berechtigung, Ihre Rolle zu übernehmen. Die Anforderungen für Escape-Anführungszeichen in der JSON-Zeichenfolge können je nach Shell variieren.

```
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"}]}'
```

Sie können die Vertrauensrichtlinie für die Rolle auch mithilfe einer separaten JSON-Datei definieren. Im folgenden Beispiel ist `trust-policy.json` eine Datei im aktuellen Verzeichnis.

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
```

Um der Rolle Berechtigungen hinzuzufügen, verwenden Sie den attach-policy-to-role-Befehl. Mit dem folgenden Befehl wird die verwaltete Richtlinie `AWSLambdaBasicExecutionRole` der Ausführungsrolle `lambda-ex` hinzugefügt.

```
aws iam attach-role-policy --role-name lambda-ex --policy-arn arn:aws:iam::aws:policy/
service-role/AWSLambdaBasicExecutionRole
```

Nachdem Sie Ihre Ausführungsrolle erstellt haben, fügen Sie sie an Ihre Funktion an. Wenn Sie eine Funktion in der Lambda-Konsole erstellen, können Sie eine beliebige Ausführungsrolle, die Sie zuvor erstellt haben, an die Funktion anhängen. Wenn Sie einer vorhandenen Funktion eine neue

Ausführungsrolle zuordnen möchten, folgen Sie den Schritten unter the section called “Aktualisierung der Ausführungsrolle einer Funktion”.

Gewähren Sie den Zugriff auf Ihre Lambda-Ausführungsrolle mit den geringsten Berechtigungen

Wenn Sie während der Entwicklungsphase zum ersten Mal eine IAM-Rolle für Ihre Lambda-Funktion erstellen, können Sie manchmal Berechtigungen erteilen, die über das erforderliche Maß hinausgehen. Bevor Sie Ihre Funktion in der Produktionsumgebung veröffentlichen, sollten Sie als bewährte Methode die Richtlinie so anpassen, dass sie nur die erforderlichen Berechtigungen enthält. Weitere Informationen finden Sie unter Anwenden von Berechtigungen mit geringsten Berechtigungen im IAM-Benutzerhandbuch.

Verwenden Sie IAM Access Analyzer, um die erforderlichen Berechtigungen für die IAM-Ausführungsrollenrichtlinie zu identifizieren. IAM Access Analyzer überprüft Ihre AWS CloudTrail Protokolle über den von Ihnen angegebenen Zeitraum und generiert eine Richtlinienvorlage mit nur den Berechtigungen, die die Funktion in diesem Zeitraum verwendet hat. Sie können die Vorlage verwenden, um eine verwaltete Richtlinie mit definierten Berechtigungen zu erstellen und sie dann an die IAM-Rolle anzuhängen. Auf diese Weise gewähren Sie nur die Berechtigungen, die die Rolle für die Interaktion mit AWS Ressourcen für Ihren speziellen Anwendungsfall benötigt.

Weitere Informationen finden Sie unter Generieren von Richtlinien basierend auf Zugriffsaktivitäten im IAM-Benutzerhandbuch.

Berechtigungen in der Ausführungsrolle anzeigen und aktualisieren

In diesem Thema wird beschrieben, wie Sie die Ausführungsrolle Ihrer Funktion anzeigen und aktualisieren können.

Themen

- Anzeigen der Ausführungsrolle einer Funktion
- Aktualisierung der Ausführungsrolle einer Funktion

Anzeigen der Ausführungsrolle einer Funktion

Um die Ausführungsrolle einer Funktion anzuzeigen, verwenden Sie die Lambda-Konsole.

So zeigen Sie die Ausführungsrolle einer Funktion an (Konsole)

1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.
2. Wählen Sie den Namen einer Funktion aus.
3. Wählen Sie Configuration (Konfiguration) und anschließend Permissions (Berechtigungen) aus.
4. Unter Ausführungsrolle können Sie die Rolle anzeigen, die derzeit als Ausführungsrolle der Funktion verwendet wird. Der Einfachheit halber können Sie im Abschnitt Ressourcenübersicht alle Ressourcen und Aktionen anzeigen, auf die die Funktion zugreifen kann. Sie können auch einen Dienst aus der Dropdown-Liste auswählen, um alle mit diesem Dienst verbundenen Berechtigungen anzuzeigen.

Aktualisierung der Ausführungsrolle einer Funktion

Sie können Berechtigungen jederzeit zur Ausführungsrolle einer Funktion hinzufügen bzw. daraus entfernen oder Ihre Funktion so konfigurieren, dass sie eine andere Rolle verwendet. Wenn Ihre Funktion Zugriff auf andere Dienste oder Ressourcen benötigt, müssen Sie der Ausführungsrolle die erforderlichen Berechtigungen hinzufügen.

Wenn Sie Ihrer Funktion Berechtigungen hinzufügen, führen Sie auch eine triviale Aktualisierung ihres Codes oder ihrer Konfiguration durch. Dies zwingt ausgeführte Instances Ihrer Funktion, die veraltete Anmeldeinformationen haben, anzuhalten und ersetzt zu werden.

Um die Ausführungsrolle einer Funktion zu aktualisieren, können Sie die Lambda-Konsole verwenden.

So aktualisieren Sie die Ausführungsrolle einer Funktion (Konsole)

1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.
2. Wählen Sie den Namen einer Funktion aus.
3. Wählen Sie Configuration (Konfiguration) und anschließend Permissions (Berechtigungen) aus.
4. Wählen Sie unter Ausführungsrolle die Option Bearbeiten aus.
5. Wenn Sie Ihre Funktion so aktualisieren möchten, dass sie eine andere Rolle als Ausführungsrolle verwendet, wählen Sie die neue Rolle im Dropdownmenü unter Bestehende Rolle aus.

> **Note**
>
> Wenn Sie die Berechtigungen innerhalb einer vorhandenen Ausführungsrolle aktualisieren möchten, können Sie dies nur in der AWS Identity and Access Management (IAM-) Konsole tun.

Wenn Sie eine neue Rolle als Ausführungsrolle erstellen möchten, wählen Sie unter Ausführungsrolle die Option Neue Rolle aus AWS Richtlinienvorlagen erstellen aus. Geben Sie dann unter Rollenname einen Namen für Ihre neue Rolle ein und geben Sie unter Richtlinienvorlagen alle Richtlinien an, die Sie der neuen Rolle zuordnen möchten.

6. Wählen Sie Speichern.

In der Ausführungsrolle mit AWS verwalteten Richtlinien arbeiten

Die folgenden AWS verwalteten Richtlinien bieten Berechtigungen, die für die Verwendung von Lambda-Funktionen erforderlich sind.

Änderungen	Beschreibung	Date
AWSLambdaMSKExecutionRolle — Lambda hat dieser Richtlinie die Berechtigung kafka: DescribeCluster V2 hinzugefügt.	`AWSLambdaMSKExecutionRole` gewährt Berechtigungen zum Lesen und Zugreifen auf Datensätze aus einem Amazon Managed Streaming for Apache Kafka (Amazon MSK) -Cluster, zum Verwalten elastischer Netzwerkschnittstellen (ENIs) und zum CloudWatch Schreiben in Protokolle.	17. Juni 2022
AWSLambdaBasicExecutionRole— Lambda hat damit begonnen, Änderungen an	`AWSLambdaBasicExecutionRole` erteilt Berechtigungen, um Protokolle auf CloudWatch hochzuladen.	14. Februar 2022

Änderungen	Beschreibung	Date
dieser Richtlinie nachzuver folgen.		
AWSLambdaDBExecuti onDynamo-Rolle — Lambda hat damit begonnen, Änderungen an dieser Richtlini e nachzuverfolgen.	`AWSLambdaDynamoDBE xecutionRole` gewährt Berechtigungen zum Lesen von Datensätzen aus einem Amazon DynamoDB DynamoDB-Stream und zum CloudWatch Schreiben in Logs.	14. Februar 2022
AWSLambdaKinesisEx ecutionRole— Lambda hat damit begonnen, Änderungen an dieser Richtlinie nachzuver folgen.	`AWSLambdaKinesisEx ecutionRole` gewährt Berechtigungen zum Lesen von Ereignissen aus einem Amazon Kinesis Kinesis-Datenstream und zum Schreiben in CloudWatch Protokolle.	14. Februar 2022
AWSLambdaMSKExecut ionRolle — Lambda hat damit begonnen, Änderungen an dieser Richtlinie nachzuver folgen.	`AWSLambdaMSKExecut ionRole` gewährt Berechtig ungen zum Lesen und Zugreifen auf Datensätze aus einem Amazon Managed Streaming for Apache Kafka (Amazon MSK) -Cluster, zum Verwalten elastisch er Netzwerkschnittstellen (ENIs) und zum CloudWatch Schreiben in Protokolle.	14. Februar 2022

Änderungen	Beschreibung	Date
AWSLambdaSQSQueueExecutionRole— Lambda hat damit begonnen, Änderungen an dieser Richtlinie nachzuverfolgen.	`AWSLambdaSQSQueueExecutionRole` gewährt Berechtigungen zum Lesen einer Nachricht aus einer Amazon Simple Queue Service (Amazon SQS) - Warteschlange und zum Schreiben in CloudWatch Protokolle.	14. Februar 2022
AWSLambdaVPCAccessExecutionRole— Lambda hat damit begonnen, Änderungen an dieser Richtlinie nachzuverfolgen.	`AWSLambdaVPCAccessExecutionRole` gewährt Berechtigungen zum Verwalten ENIs innerhalb einer Amazon VPC und zum Schreiben in CloudWatch Logs.	14. Februar 2022
AWSXRayDaemonWriteAccess— Lambda hat damit begonnen, Änderungen an dieser Richtlinie nachzuverfolgen.	`AWSXRayDaemonWriteAccess` erteilt Berechtigungen zum Hochladen von Nachverfolgungsdaten auf X-Ray.	14. Februar 2022
CloudWatchLambdaInsightsExecutionRolePolicy— Lambda hat damit begonnen, Änderungen an dieser Richtlinie nachzuverfolgen.	`CloudWatchLambdaInsightsExecutionRolePolicy` gewährt Berechtigungen zum Schreiben von Laufzeitmetriken in CloudWatch Lambda Insights.	14. Februar 2022

Änderungen	Beschreibung	Date
AmazonS3 ObjectLambdaExecutionRolePolicy — Lambda hat begonnen, Änderungen an dieser Richtlinie nachzuverfolgen.	`AmazonS3ObjectLambdaExecutionRolePolicy` gewährt Berechtigungen zur Interaktion mit dem Amazon Simple Storage Service (Amazon S3) -Objekt Lambda und zum Schreiben in CloudWatch Logs.	14. Februar 2022

Bei einigen Funktionen versucht die Lambda-Konsole, Ihrer Ausführungsrolle in einer vom Kunden verwalteten Richtlinie fehlende Berechtigungen hinzuzufügen. Diese Politiken können zahlreich werden. Fügen Sie der Ausführungsrolle die relevanten AWS -verwalteten Richtlinien hinzu, bevor Sie Funktionen aktivieren, um das Erstellen zusätzlicher Richtlinien zu vermeiden.

Wenn Sie eine Ereignisquellen-Zuweisung zum Aufrufen Ihrer Funktion verwenden, verwendet Lambda die Ausführungsrolle zum Lesen von Ereignisdaten. Zum Beispiel liest eine Ereignisquellen-Zuweisung für Kinesis-Ereignisse aus einem Datenstrom und sendet diese in Batches an Ihre Funktion.

Wenn ein Dienst eine Rolle in Ihrem Konto übernimmt, können Sie die globalen Bedingungskontextschlüssel `aws:SourceAccount` und `aws:SourceArn` in Ihrer Rollenvertrauensrichtlinie übernehmen, um den Zugriff auf die Rolle nur auf Anforderungen zu beschränken, die von erwarteten Ressourcen generiert werden. Weitere Informationen finden Sie unter Vermeidung des Problems des verwirrten Stellvertreters im dienstübergreifenden Szenario für AWS -Security-Token-Service.

Zusätzlich zu den AWS verwalteten Richtlinien bietet die Lambda-Konsole Vorlagen für die Erstellung einer benutzerdefinierten Richtlinie mit Berechtigungen für zusätzliche Anwendungsfälle. Wenn Sie eine Funktion in der Lambda-Konsole erstellen, haben Sie die Wahl, eine neue Ausführungsrolle mit Berechtigungen aus einer oder mehreren Vorlagen zu erstellen. Diese Vorlagen werden auch automatisch angewendet, wenn Sie eine Funktion von einer Vorlage erstellen oder wenn Sie Optionen konfigurieren, die Zugriff auf andere Services erfordern. Beispielvorlagen sind im GitHubRepository dieses Handbuchs verfügbar.

Verwendung der Quellfunktion ARN zur Steuerung des Funktionszugriffsverhaltens

Es ist üblich, dass Ihr Lambda-Funktionscode API-Anorderungen an andere AWS-Services stellt. Um diese Anforderungen zu stellen, generiert Lambda einen kurzlebigen Satz von Anmeldeinformationen, indem es die Ausführungsrolle Ihrer Funktion übernimmt. Diese Anmeldeinformationen sind während des Aufrufs Ihrer Funktion als Umgebungsvariablen verfügbar. Wenn Sie mit arbeiten AWS SDKs, müssen Sie die Anmeldeinformationen für das SDK nicht direkt im Code angeben. Standardmäßig überprüft die Kette der Anbieter von Anmeldeinformationen nacheinander jeden Ort, an dem Sie Anmeldeinformationen festlegen können, und wählt die erste verfügbare aus – in der Regel die Umgebungsvariablen (`AWS_ACCESS_KEY_ID`, `AWS_SECRET_ACCESS_KEY` und `AWS_SESSION_TOKEN`).

Lambda fügt die Quellfunktion ARN in den Anmeldeinformationskontext ein, wenn es sich bei der Anfrage um eine AWS API-Anforderung handelt, die aus Ihrer Ausführungsumgebung stammt. Lambda fügt auch den Quellfunktions-ARN für die folgenden AWS -API-Anfragen ein, die Lambda in Ihrem Namen außerhalb Ihrer Ausführungsumgebung durchführt:

Service	Action	Grund
CloudWatch Logs	`CreateLogGroup` , `CreateLogStream` , `PutLogEvents`	Um Logs in einer CloudWatch Logs-Protokollgruppe zu speichern
X-Ray	`PutTraceSegments`	So senden Sie Verfolgungsdaten an X-Ray
Amazon EFS	`ClientMount`	So stellen Sie eine Verbindung zwischen Ihrer Funktion und einem Amazon Elastic File System (Amazon EFS) Dateisystem her

Andere AWS API-Aufrufe, die Lambda außerhalb Ihrer Ausführungsumgebung in Ihrem Namen mit derselben Ausführungsrolle durchführt, enthalten nicht die Quellfunktion ARN. Beispiele für solche API-Aufrufe außerhalb der Ausführungsumgebung sind:

- Ruft AWS Key Management Service (AWS KMS) auf, um Ihre Umgebungsvariablen automatisch zu verschlüsseln und zu entschlüsseln.
- Ruft Amazon Elastic Compute Cloud (Amazon EC2) auf, um elastische Netzwerkschnittstellen (ENIs) für eine VPC-fähige Funktion zu erstellen.
- Aufrufe AWS-Services, z. B. Amazon Simple Queue Service (Amazon SQS), um aus einer Ereignisquelle zu lesen, die als Ereignisquellen-Mapping eingerichtet ist.

Mit dem ARN der Quellfunktion im Kontext der Anmeldeinformationen können Sie überprüfen, ob ein Aufruf Ihrer Ressource aus dem Code einer bestimmten Lambda-Funktion stammt. Um dies zu überprüfen, verwenden Sie die `lambda:SourceFunctionArn`-Zustandsschlüssel in einer identitätsbasierten IAM-Richtlinie oder einer Service-Kontrollrichtlinie (SCP).

> **Note**
>
> Sie können den `lambda:SourceFunctionArn`-Bedingungsschlüssel nicht in ressourcenbasierten Richtlinien verwenden.

Mit diesem Bedingungsschlüssel in Ihren identitätsbasierten Richtlinien oder SCPs können Sie Sicherheitskontrollen für die API-Aktionen implementieren, die Ihr Funktionscode für andere vornimmt. AWS-Services Dies hat einige wichtige Sicherheitsanwendungen, z. B. um Ihnen zu helfen, die Quelle eines Anmeldeinformationslecks zu identifizieren.

> **Note**
>
> Der `lambda:SourceFunctionArn`-Bedingungsschlüssel unterscheidet sich von den `lambda:FunctionArn` und `aws:SourceArn`-Bedingungsschlüsseln. Der `lambda:FunctionArn`-Bedingungsschlüssel gilt nur für Ereignisquellenzuordnungen und hilft bei der Definition der Funktionen, die Ihre Ereignisquelle aufrufen kann. Der `aws:SourceArn` Bedingungsschlüssel gilt nur für Richtlinien, bei denen Ihre Lambda-Funktion die Zielressource ist, und hilft zu definieren, welche anderen AWS-Services AMD-Ressourcen diese Funktion aufrufen können. Der `lambda:SourceFunctionArn` Bedingungsschlüssel kann für jede identitätsbasierte Richtlinie oder jeden SCP gelten, um die spezifischen Lambda-Funktionen zu definieren, die berechtigt sind, bestimmte AWS API-Aufrufe an andere Ressourcen zu tätigen.

Um `lambda:SourceFunctionArn` in Ihrer Richtlinie zu verwenden, fügen Sie es als Bedingung mit einem der ARN-Bedingungsoperatoren ein. Der Wert des Schlüssels muss ein gültiger ARN sein.

Angenommen, Ihr Lambda-Funktionscode macht einen `s3:PutObject`-Aufruf, der auf einen bestimmten Amazon-S3-Bucket abzielt. Möglicherweise möchten Sie nur einer bestimmten Lambda-Funktion erlauben, dass `s3:PutObject` auf diesen Bucket zugreift. In diesem Fall sollte der Ausführungsrolle Ihrer Funktion eine Richtlinie angefügt sein, die wie folgt aussieht:

Example Richtlinie, die einer bestimmten Lambda-Funktion Zugriff auf eine Amazon-S3-Ressource gewährt

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"
                }
            }
        }
    ]
}
```

Diese Richtlinie erlaubt nur `s3:PutObject` Zugriff wenn die Quelle die Lambda-Funktion mit ARN `arn:aws:lambda:us-east-1:123456789012:function:source_lambda` ist. Diese Richtlinie erlaubt `s3:PutObject` keinen Zugriff auf jede andere aufrufende Identität. Dies gilt auch dann, wenn eine andere Funktion oder Entität einen `s3:PutObject`-Aufruf mit der gleichen Ausführungsrolle tätigt.

> **Note**
>
> Der Bedingungsschlüssel `lambda:SourceFunctionARN` unterstützt keine Lambda-Funktionsversionen oder -Funktionsaliase. Wenn Sie den ARN für eine bestimmte Funktionsversion oder einen bestimmten Alias verwenden, ist Ihre Funktion nicht berechtigt, die von Ihnen angegebene Aktion auszuführen. Achten Sie darauf, den unqualifizierten ARN für Ihre Funktion ohne Versions- oder Alias-Suffix zu verwenden.

Sie können auch in verwenden. `lambda:SourceFunctionArn` SCPs Angenommen, Sie möchten den Zugriff auf Ihren Bucket entweder auf den Code einer einzelnen Lambda-Funktion oder auf Aufrufe aus einer bestimmten Amazon Virtual Private Cloud (VPC) beschränken. Das folgende SCP illustriert dies.

Example Richtlinie, die den Zugriff auf Amazon S3 unter bestimmten Bedingungen verweigert

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"
                }
            }
        }
    ]
}
```

Diese Richtlinie verweigert alle S3-Aktionen, sofern sie nicht von einer bestimmten Lambda-Funktion mit ARN `arn:aws:lambda:*:123456789012:function:source_lambda` stammen, oder sofern sie nicht aus dem angegebenen VPC stammen. Der `StringNotEqualsIfExists`-Operator weist IAM an, diese Bedingung nur zu verarbeiten, wenn der `aws:SourceVpc`-Schlüssel in der Anfrage vorhanden ist. In ähnlicher Weise berücksichtigt IAM den `ArnNotEqualsIfExists`-Operator nur, wenn `lambda:SourceFunctionArn` vorhanden ist.

Anderen AWS Entitäten Zugriff auf Ihre Lambda-Funktionen gewähren

Um anderen AWS-Konten Organisationen und Diensten Berechtigungen für den Zugriff auf Ihre Lambda-Ressourcen zu erteilen, haben Sie mehrere Möglichkeiten:

- Sie können identitätsbasierte Richtlinien verwenden, um anderen Benutzern Zugriff auf Ihre Lambda-Ressourcen zu gewähren. Identitätsbasierte Richtlinien können direkt auf Benutzer oder auf Gruppen und Rollen angewendet werden, die einem Benutzer zugeordnet sind.
- Sie können ressourcenbasierte Richtlinien verwenden, um anderen Konten und AWS-Services Berechtigungen für den Zugriff auf Ihre Lambda-Ressourcen zu gewähren. Wenn ein Benutzer versucht, auf eine Lambda-Ressource zuzugreifen, berücksichtigt Lambda sowohl die identitätsbasierten Richtlinien des Benutzers als auch die ressourcenbasierte Richtlinie der Ressource. Wenn ein AWS Service wie Amazon Simple Storage Service (Amazon S3) Ihre Lambda-Funktion aufruft, berücksichtigt Lambda nur die ressourcenbasierte Richtlinie.
- Sie können ein attributbasiertes Zugriffskontrollmodell (ABAC) verwenden, um den Zugriff auf Ihre Lambda-Funktionen zu steuern. Mit ABAC können Sie Tags an eine Lambda-Funktion anhängen, sie in bestimmten API-Anforderungen übergeben oder sie an den IAM-Prinzipal anfügen, der die Anforderung stellt. Geben Sie dieselben Tags im Bedingungselement einer IAM-Richtlinie an, um den Funktionszugriff zu steuern.

Um Ihnen bei der Optimierung Ihrer Berechtigungen für den Zugriff mit den geringsten Rechten zu helfen, bietet Lambda einige zusätzliche Bedingungen, die Sie in Ihre Richtlinien aufnehmen können. Weitere Informationen finden Sie unter the section called “Ressourcen und Bedingungen”.

Identitätsbasierte IAM-Richtlinien für Lambda

Sie können identitätsbasierte Richtlinien in AWS Identity and Access Management (IAM) verwenden, um Benutzern in Ihrem Konto Zugriff auf Lambda zu gewähren. Identitätsbasierte Richtlinien können sich auf Benutzer, Benutzergruppen oder Rollen beziehen. Sie können auch Benutzern in einem anderen Konto die Berechtigung erteilen, eine Rolle in Ihrem Konto zu übernehmen und auf Ihre Lambda-Ressourcen zuzugreifen.

Lambda bietet AWS verwaltete Richtlinien, die Zugriff auf Lambda-API-Aktionen und in einigen Fällen Zugriff auf andere AWS Dienste gewähren, die zur Entwicklung und Verwaltung von Lambda-Ressourcen verwendet werden. Lambda aktualisiert je nach Bedarf die verwalteten Richtlinien, um

sicherzustellen, dass Ihre Benutzer Zugriff auf neue Funktionen haben, sobald diese veröffentlicht werden.

- AWSLambda_FullAccess— Gewährt vollen Zugriff auf Lambda-Aktionen und andere AWS Dienste, die zur Entwicklung und Wartung von Lambda-Ressourcen verwendet werden.
- AWSLambda_ReadOnlyAccess— Gewährt schreibgeschützten Zugriff auf Lambda-Ressourcen.
- AWSLambdaRolle — Erteilt Berechtigungen zum Aufrufen von Lambda-Funktionen.

AWS verwaltete Richtlinien gewähren Berechtigungen für API-Aktionen, ohne die Lambda-Funktionen oder -Layer einzuschränken, die ein Benutzer ändern kann. Für eine feinere Steuerung können Sie eigene Richtlinien erstellen, die den Umfang der Berechtigungen eines Benutzers einschränken.

Themen

- Benutzern den Zugriff auf eine Lambda-Funktion gewähren
- Benutzern den Zugriff auf eine Lambda-Ebene gewähren

Benutzern den Zugriff auf eine Lambda-Funktion gewähren

Verwenden Sie identitätsbasierte Richtlinien, um Benutzern, Benutzergruppen oder Rollen die Durchführung von Vorgängen mit Lambda-Funktionen zu ermöglichen.

> **Note**
>
> Für eine als Container-Image definierte Funktion muss die Benutzerberechtigung für den Zugriff auf das Image in Amazon Elastic Container Registry (Amazon ECR) konfiguriert werden. Ein Beispiel finden Sie unter Amazon ECR-Repository-Richtlinien.

Im Folgenden finden Sie ein Beispiel für eine Berechtigungsrichtlinie mit eingeschränktem Umfang. Sie ermöglicht Benutzern das Erstellen und Verwalten von Lambda-Funktionen mit einem bestimmten Präfix (`intern-`), die mit einer bestimmten Ausführungsrolle konfiguriert sind.

Example Richtlinie für die Funktionsentwicklung

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-*"
        }
    ]
}
```

Die Berechtigungen in der Richtlinie sind in Anweisungen basierend auf den Ressourcen und Bedingungen organisiert, die sie unterstützen.

- `ReadOnlyPermissions` – Die Lambda-Konsole verwendet diese Berechtigungen, wenn Sie Funktionen durchsuchen und anzeigen. Sie unterstützen keine Ressourcenmuster oder -bedingungen.

```
        "Action": [
            "lambda:GetAccountSettings",
            "lambda:GetEventSourceMapping",
            "lambda:GetFunction",
            "lambda:GetFunctionConfiguration",
```

```
            "lambda:GetFunctionCodeSigningConfig",
            "lambda:GetFunctionConcurrency",
            "lambda:ListEventSourceMappings",
            "lambda:ListFunctions",
            "lambda:ListTags",
            "iam:ListRoles"
        ],
        "Resource": "*"
```

- `DevelopFunctions` – Verwendet eine beliebige Lambda-Aktion, die auf Funktionen mit dem Präfix `intern-` ausgeführt wird, außer `AddPermission` und `PutFunctionConcurrency`. `AddPermission` ändert die ressourcenbasierte Richtlinie der Funktion und kann mögliche Sicherheitsprobleme darstellen. `PutFunctionConcurrency` reserviert Skalierungskapazitäten für eine Funktion und kann anderen Funktionen Kapazitäten wegnehmen.

```
        "NotAction": [
            "lambda:AddPermission",
            "lambda:PutFunctionConcurrency"
        ],
        "Resource": "arn:aws:lambda:*:*:function:intern-*"
```

- `DevelopEventSourceMappings` – Verwaltet Ereignis-Quellzuweisungen für Funktionen mit dem Präfix `intern-`. Diese Aktionen werden für Ereignis-Quellzuweisungen angewendet. Sie können sie jedoch mit einer Bedingung nach Funktion einschränken.

```
        "Action": [
            "lambda:DeleteEventSourceMapping",
            "lambda:UpdateEventSourceMapping",
            "lambda:CreateEventSourceMapping"
        ],
        "Resource": "*",
        "Condition": {
            "StringLike": {
                "lambda:FunctionArn": "arn:aws:lambda:*:*:function:intern-*"
            }
        }
```

- `PassExecutionRole` – Zeigt und übergibt nur eine Rolle namens `intern-lambda-execution-role`, die von einem Benutzer mit IAM-Berechtigungen erstellt und verwaltet werden muss. `PassRole` wird verwendet, wenn Sie eine Ausführungsrolle einer Funktion zuweisen.

```
            "Action": [
                "iam:ListRolePolicies",
                "iam:ListAttachedRolePolicies",
                "iam:GetRole",
                "iam:GetRolePolicy",
                "iam:PassRole",
                "iam:SimulatePrincipalPolicy"
            ],
            "Resource": "arn:aws:iam::*:role/intern-lambda-execution-role"
```

- `ViewLogs`— Verwenden Sie CloudWatch Logs, um Logs für Funktionen anzuzeigen, denen das Präfix vorangestellt ist. `intern-`

```
            "Action": [
                "logs:*"
            ],
            "Resource": "arn:aws:logs:*:*:log-group:/aws/lambda/intern-*"
```

Diese Richtlinie erlaubt es einem Benutzer, Lambda zu verwenden, ohne die Ressourcen anderer Benutzer zu gefährden. Es erlaubt einem Benutzer nicht, eine Funktion so zu konfigurieren, dass sie von anderen AWS Diensten ausgelöst wird oder diese aufruft, wofür umfassendere IAM-Berechtigungen erforderlich sind. Es beinhaltet auch keine Genehmigung für Dienste, die keine Richtlinien mit begrenztem Geltungsbereich unterstützen, wie X-Ray CloudWatch . Verwenden Sie die schreibgeschützten Richtlinien für diese Services, um dem Benutzer Zugriff auf Metriken und Ablaufverfolgungsdaten zu gewähren.

Wenn Sie Auslöser für Ihre Funktion konfigurieren, benötigen Sie Zugriff, um den AWS Dienst verwenden zu können, der Ihre Funktion aufruft. Wenn Sie z. B. einen Amazon-S3-Auslöser konfigurieren, müssen Sie zur Verwendung der Amazon-S3-Aktionen berechtigt sein, mit denen Bucket-Benachrichtigungen verwaltet werden. Viele dieser Berechtigungen sind in der AWSLambda_FullAccessverwalteten Richtlinie enthalten.

Benutzern den Zugriff auf eine Lambda-Ebene gewähren

Verwenden Sie identitätsbasierte Richtlinien, um Benutzern, Benutzergruppen oder Rollen die Durchführung von Vorgängen mit Lambda-Ebenen zu ermöglichen. Die folgende Richtlinie erteilt einem Benutzer die Berechtigung, Ebenen zu erstellen und sie mit Funktionen zu verwenden. Die

Ressourcenmuster ermöglichen es dem Benutzer, in jeder AWS-Region beliebigen Layer-Version zu arbeiten, sofern der Name der Ebene mit beginnt`test-`.

Example Richtlinie für die Ebenenentwicklung

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-*:*"
        }
    ]
}
```

Sie können die Ebenenverwendung auch bei der Erstellung und Konfiguration von Funktionen mit der `lambda:Layer`-Bedingung erzwingen. Sie können beispielsweise Benutzer daran hindern, Ebenen zu verwenden, die von anderen Konten veröffentlicht wurden. Die folgende Richtlinie fügt eine Bedingung zu den Aktionen `CreateFunction` und `UpdateFunctionConfiguration` hinzu, um zu verlangen, dass alle angegebenen Ebenen vom Konto `123456789012` stammen.

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:*:*"
                    ]
                }
            }
        }
    ]
}
```

Um sicherzustellen, dass die Bedingung angewendet wird, stellen Sie sicher, dass keine anderen Anweisungen dem Benutzer die Berechtigung für diese Aktionen erteilen.

Anzeigen von ressourcenbasierten IAM-Richtlinien in Lambda

Lambda unterstützt ressourcenbasierte Berechtigungsrichtlinien für Lambda-Funktionen und -Ebenen. Sie können ressourcenbasierte Richtlinien verwenden, um anderen AWS Konten, Organisationen oder Diensten Zugriff zu gewähren. Ressourcenbasierte Richtlinien gelten für einzelne Funktionen, Versionen, Aliasnamen oder Ebenenversionen.

Console

So zeigen Sie die ressourcenbasierte Richtlinie einer Funktion an

1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.
2. Wählen Sie eine Funktion aus.
3. Wählen Sie Konfiguration und anschließend Berechtigungen aus.
4. Scrollen Sie nach unten zu Ressourcenbasierte Richtlinie und wählen Sie dann Richtliniendokument anzeigen aus. Die ressourcenbasierte Richtlinie zeigt die Berechtigungen, die angewendet werden, wenn ein anderes Konto oder ein anderer AWS

Dienst versucht, auf die Funktion zuzugreifen. Das folgende Beispiel zeigt eine Anweisung, mit der Amazon S3 eine Funktion mit dem Namen `my-function` für einen Bucket mit dem Namen `amzn-s3-demo-bucket` im Konto `123456789012` aufrufen kann.

Example Ressourcenbasierte Richtlinie

```
{
    "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

Um die ressourcenbasierte Richtlinie einer Funktion anzuzeigen, verwenden Sie den Befehl `get-policy`.

```
aws lambda get-policy \
  --function-name my-function \
  --output text
```

Die Ausgabe sollte folgendermaßen aussehen:

```
{"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*"}}}]}
```

Fügen Sie bei Versionen und Aliassen die Versionsnummer oder den Alias an den Funktionsnamen an.

```
aws lambda get-policy --function-name my-function:PROD
```

Um Berechtigungen von Ihrer Funktion zu entfernen, verwenden Sie `remove-permission`.

```
aws lambda remove-permission \
  --function-name example \
  --statement-id sns
```

Verwenden Sie den Befehl `get-layer-version-policy`, um die Berechtigungen für einen Layer anzuzeigen.

```
aws lambda get-layer-version-policy \
  --layer-name my-layer \
  --version-number 3 \
  --output text
```

Die Ausgabe sollte folgendermaßen aussehen:

```
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"}}}"
```

Verwenden Sie `remove-layer-version-permission`, um Anweisungen aus der Richtlinie zu entfernen.

```
aws lambda remove-layer-version-permission --layer-name my-layer --version-number 3
 --statement-id engineering-org
```

Unterstützte API-Aktionen

Die folgenden Lambda-API-Aktionen unterstützen ressourcenbasierte Richtlinien:

- CreateAlias
- DeleteAlias
- DeleteFunction
- DeleteFunctionConcurrency
- DeleteFunctionEventInvokeConfig
- DeleteProvisionedConcurrencyConfig
- GetAlias
- GetFunction
- GetFunctionConcurrency
- GetFunctionConfiguration
- GetFunctionEventInvokeConfig
- GetPolicy
- GetProvisionedConcurrencyConfig
- Invoke
- InvokeFunctionUrl(nur mit Genehmigung)
- ListAliases
- ListFunctionEventInvokeConfigs
- ListProvisionedConcurrencyConfigs
- ListTags
- ListVersionsByFunction
- PublishVersion
- PutFunctionConcurrency
- PutFunctionEventInvokeConfig
- PutProvisionedConcurrencyConfig

- TagResource
- UntagResource
- UpdateAlias
- UpdateFunctionCode
- UpdateFunctionEventInvokeConfig

Lambda-Funktionszugriff gewähren für AWS-Services

Wenn Sie einen AWS Dienst verwenden, um Ihre Funktion aufzurufen, erteilen Sie die Erlaubnis in einer Erklärung zu einer ressourcenbasierten Richtlinie. Sie können die Anweisung auf die gesamte Funktion anwenden oder die Anweisung auf eine einzelne Version oder einen einzelnen Alias beschränken.

> **Note**
>
> Wenn Sie Ihrer Funktion mit der Lambda-Konsole einen Auslöser hinzufügen, aktualisiert die Konsole die ressourcenbasierte Richtlinie der Funktion, damit der Service sie aufrufen kann. Um Berechtigungen für andere Konten oder Services zu erteilen, die in der Lambda-Konsole nicht verfügbar sind, verwenden Sie die AWS CLI-CLI.

Fügen Sie eine Anweisung mit dem Befehl add-permission hinzu. Die einfachste ressourcenbasierte Richtlinienanweisung ermöglicht es einem Service, eine Funktion aufzurufen. Der folgende Befehl gewährt die Amazon-Simple Notification Service-Berechtigung zum Aufrufen einer Funktion namens `my-function`.

```
aws lambda add-permission \
  --function-name my-function \
  --action lambda:InvokeFunction \
  --statement-id sns \
  --principal sns.amazonaws.com \
  --output text
```

Die Ausgabe sollte folgendermaßen aussehen:

```
{"Sid":"sns","Effect":"Allow","Principal":
{"Service":"sns.amazonaws.com"},"Action":"lambda:InvokeFunction","Resource":"arn:aws:lambda:us-
east-2:123456789012:function:my-function"}
```

Auf diese Weise kann Amazon SNS die API-Aktion Aufruf für die Funktion aufrufen, ohne das Amazon-SNS-Thema einzuschränken, durch das der Aufruf ausgelöst wird. Um sicherzustellen, dass Ihre Funktion nur von einer bestimmten Ressource aufgerufen wird, geben Sie den Amazon-Ressourcennamen (ARN) der Ressource mit der Option `source-arn` an. Der folgende Befehl erlaubt Amazon SNS nur das Aufrufen der Funktion für Abonnements für ein Thema namens `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
```

Einige Services können Funktionen in anderen Konten aufrufen. Wenn Sie einen Quell-ARN angeben, der Ihre Konto-ID enthält, ist das kein Problem. Für Amazon S3 ist die Quelle jedoch ein Bucket, dessen ARN nicht über eine Konto-ID verfügt. Es ist möglich, dass Sie den Bucket löschen und ein anderes Konto einen Bucket mit demselben Namen erstellt. Verwenden Sie die Option `source-account` mit Ihrer Konto-ID, um sicherzustellen, dass nur Ressourcen in Ihrem Konto die Funktion aufrufen können.

```
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
```

Gewähren des Funktionszugriffs auf eine Organisation

Um einer Organisation Berechtigungen in AWS Organizations zu gewähren, geben Sie die Organisations-ID als `principal-org-id` an. Der folgende add-permission-Befehl gewährt allen Benutzern in der Organisation `o-a1b2c3d4e5f` Aufrufzugriff.

```
aws lambda add-permission \
  --function-name example \
  --statement-id PrincipalOrgIDExample \
  --action lambda:InvokeFunction \
  --principal * \
  --principal-org-id o-a1b2c3d4e5f
```

> **Note**
>
> In diesem Befehl ist `Principal *`. Dies bedeutet, dass alle Benutzer in der Organisation `o-a1b2c3d4e5f` Berechtigungen für den Funktionsaufruf erhalten. Wenn Sie eine AWS-Konto Oder-Rolle als angeben`Principal`, erhält nur dieser Prinzipal die Berechtigungen für den Funktionsaufruf, aber nur, wenn er auch Teil der Organisation ist. `o-a1b2c3d4e5f`

Dieser Befehl erstellt eine ressourcenbasierte Richtlinie, die wie folgt aussieht:

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"
                }
            }
        }
    ]
}
```

Weitere Informationen finden Sie unter aws: PrincipalOrg ID im IAM-Benutzerhandbuch.

Gewähren des Lambda-Funktionszugriffs für andere Konten

Um eine Funktion mit einer anderen gemeinsam zu nutzen AWS-Konto, fügen Sie der ressourcenbasierten Richtlinie der Funktion eine kontoübergreifende Berechtigungsanweisung hinzu. Führen Sie den Befehl add-permission aus und geben Sie die Konto-ID als `principal` an. Im folgenden Beispiel wird dem Konto `111122223333` die Berechtigung zum Aufrufen von `my-function` mit dem Alias `prod` erteilt.

```
aws lambda add-permission \
  --function-name my-function:prod \
  --statement-id xaccount \
  --action lambda:InvokeFunction \
  --principal 111122223333 \
  --output text
```

Die Ausgabe sollte folgendermaßen aussehen:

```
{"Sid":"xaccount","Effect":"Allow","Principal":
{"AWS":"arn:aws:iam::111122223333:root"},"Action":"lambda:InvokeFunction","Resource":"arn:aws:l
east-1:123456789012:function:my-function"}
```

Die ressourcenbasierte Richtlinie erteilt dem anderen Konto die Berechtigung für den Zugriff auf die Funktion, erlaubt den Benutzern in diesem Konto jedoch nicht, ihre Berechtigungen zu überschreiten. Benutzer in dem anderen Konto müssen über die entsprechenden Benutzerberechtigungen verfügen, um die Lambda-API verwenden zu können.

Um den Zugriff auf einen Benutzer oder eine Rolle in einem anderen Konto einzuschränken, geben Sie den vollständigen ARN der Identität als Prinzipal an. Beispiel, `arn:aws:iam::123456789012:user/developer`.

Der Alias schränkt ein, welche Version das andere Konto aufrufen kann. Dies erfordert, dass das andere Konto den Alias in den ARN der Funktion einschließt.

```
aws lambda invoke \
  --function-name arn:aws:lambda:us-east-2:123456789012:function:my-function:prod out
```

Die Ausgabe sollte folgendermaßen aussehen:

```
{
```

```
    "StatusCode": 200,
    "ExecutedVersion": "1"
}
```

Der Funktionsbesitzer kann dann den Alias so aktualisieren, dass er auf eine neue Version verweist, ohne dass der Aufrufer die Methode ändern muss, mit der er Ihre Funktion aufruft. Dadurch wird sichergestellt, dass das andere Konto seinen Code nicht ändern muss, um die neue Version zu verwenden, und nur zum Aufruf der Version der Funktion berechtigt ist, die dem Alias zugeordnet ist.

Sie können kontoübergreifenden Zugriff für die meisten API-Aktionen erteilen, die für eine vorhandene Funktion ausgeführt werden. Beispielsweise könnten Sie Zugriff auf `lambda:ListAliases` gewähren, damit ein Konto eine Liste der Aliasse abrufen kann, oder `lambda:GetFunction`, damit es Ihren Funktionscode herunterladen kann. Fügen Sie jede Berechtigung separat hinzu oder verwenden Sie `lambda:*`, um Zugriff auf alle Aktionen der angegebenen Funktion zu gewähren.

Um anderen Konten Berechtigungen für mehrere Funktionen oder für Aktionen zu erteilen, die nicht für eine Funktion ausgeführt werden, empfehlen wir die Verwendung von IAM-Rollen.

Gewähren des Lambda-Ebenenzugriffs für andere Konten

Um eine Ebene mit einer anderen gemeinsam zu nutzen AWS-Konto, fügen Sie der ressourcenbasierten Richtlinie der Ebene eine kontoübergreifende Berechtigungserklärung hinzu. Führen Sie den add-layer-version-permissionBefehl aus und geben Sie die Konto-ID als. `principal` In jeder Anweisung können Sie einem einzelnen Konto, allen Konten oder einer Organisation in AWS Organizations eine Berechtigung erteilen.

Im folgenden Beispiel wird dem Konto 111122223333 Zugriff auf Version 2 der `bash-runtime`-Ebene gewährt.

```
aws lambda add-layer-version-permission \
  --layer-name bash-runtime \
  --version-number 2 \
  --statement-id xaccount \
  --action lambda:GetLayerVersion \
  --principal 111122223333 \
  --output text
```

Die Ausgabe sollte folgendermaßen oder ähnlich aussehen:

```
{"Sid":"xaccount","Effect":"Allow","Principal":
{"AWS":"arn:aws:iam::111122223333:root"},"Action":"lambda:GetLayerVersion","Resource":"arn:aws:
east-1:123456789012:layer:bash-runtime:2"}
```

Berechtigungen gelten nur für eine Version mit einer Ebene. Wiederholen Sie den Vorgang bei jeder Erstellung einer neuen Ebenenversion.

Um Berechtigungen für alle Konten in einer AWS Organizations Organisation zu erteilen, verwenden Sie die Option `organization-id`. Im folgenden Beispiel wird allen Konten in Organisation `o-t194hfs8cz` die Berechtigung für die Verwendung von Version 3 einer `my-layer` erteilt.

```
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
```

Die Ausgabe sollte folgendermaßen aussehen:

```
{"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"}}}"
```

Um die Berechtigung für mehrere Konten oder Organisationen zu erteilen, müssen Sie mehrere Bescheinigungen hinzufügen.

Verwendung der attributbasierten Zugriffskontrolle in Lambda

Mit attributbasierter Zugriffskontrolle (ABAC) können Sie Tags verwenden, um den Zugriff auf Ihre Lambda-Ressourcen zu steuern. Sie können Tags an bestimmte Lambda-Ressourcen anhängen, sie an bestimmte API-Anfragen anhängen oder sie an den AWS Identity and Access Management (IAM-) Principal anhängen, der die Anfrage stellt. Weitere Informationen darüber, wie attributebasierten Zugriff AWS gewährt wird, finden Sie im IAM-Benutzerhandbuch unter Steuern des Zugriffs auf AWS Ressourcen mithilfe von Tags.

Sie können ABAC verwenden, um die geringsten Berechtigungen zu gewähren, ohne einen Amazon-Ressourcennamen (ARN) oder ein ARN-Muster in der IAM-Richtlinie anzugeben. Stattdessen können Sie ein Tag im Bedingungselement einer IAM-Richtlinie angeben, um den Zugriff zu steuern. Die Skalierung ist mit ABAC einfacher, da Sie Ihre IAM-Richtlinien nicht aktualisieren müssen, wenn Sie neue Ressourcen erstellen. Fügen Sie stattdessen Tags zu den neuen Ressourcen hinzu, um den Zugriff zu steuern.

In Lambda funktionieren Tags für die folgenden Ressourcen:

- Funktionen – Weitere Informationen zu Tagging-Funktionen finden Sie unter the section called “Tags (Markierungen)”.
- Codesignaturkonfigurationen – Weitere Informationen zum Markieren von Codesignaturkonfigurationen finden Sie unter the section called “Code-Signaturkonfigurations-Tags”.
- Zuordnungen von Ereignisquellen – Weitere Informationen zum Markieren von Zuordnungen von Ereignisquellen finden Sie unter the section called “Tags für die Zuordnung von Ereignisquellen”.

Tags werden für Ebenen nicht unterstützt.

Sie können die folgenden Bedingungsschlüssel verwenden, um auf Tag basierende IAM-Richtlinienregeln zu schreiben:

- aws: ResourceTag /tag-key: Steuert den Zugriff anhand der Tags, die an eine Lambda-Ressource angehängt sind.
- aws: RequestTag /tag-key: Erfordert, dass Tags in einer Anfrage vorhanden sind, z. B. beim Erstellen einer neuen Funktion.
- aws: PrincipalTag /tag-key : Steuert anhand der Tags, die ihrem IAM-Benutzer oder ihrer IAM-Rolle zugewiesen sind, was der IAM-Prinzipal (die Person, die die Anfrage stellt) tun darf.
- aws:TagKeys: Steuert, ob bestimmte Tag-Schlüssel in einer Anfrage verwendet werden können.

Sie können nur Bedingungen für Aktionen angeben, die diese Bedingungen unterstützen. Eine Liste der Bedingungen, die von jeder Lambda-Aktion unterstützt werden, finden Sie unter Aktionen, Ressourcen und Bedingungsschlüssel für AWS Lambda in der Referenz zur Dienstautorisierung. Informationen zur Unterstützung von aws: ResourceTag /tag-key finden Sie unter „Ressourcentypen definiert von“. AWS Lambda Informationen zu aws: RequestTag /tag-key und aws: TagKeys support finden Sie unter „Aktionen definiert von“. AWS Lambda

Themen

- Sichern Sie Ihre Funktionen per Tag

Sichern Sie Ihre Funktionen per Tag

Die folgenden Schritte zeigen eine Möglichkeit, wie Sie mit ABAC Berechtigungen für Funktionen einrichten können. In diesem Beispielszenario erstellen Sie vier IAM-Berechtigungsrichtlinien. Anschließend fügen Sie diese Richtlinien einer neuen IAM-Rolle hinzu. Schließlich erstellen Sie einen IAM-Benutzer und erteilen diesem Benutzer die Berechtigung, die neue Rolle zu übernehmen.

Themen

- Voraussetzungen
- Schritt 1: Tags für neue Funktionen anfordern
- Schritt 2: Aktionen basierend auf Tags zulassen, die einer Lambda-Funktion und einem IAM-Prinzipal zugeordnet sind
- Schritt 3: Erteilen von Listenberechtigungen
- Schritt 4: Erteilen von IAM-Berechtigungen
- Schritt 5: Erstellen der IAM-Rolle
- Schritt 6: Erstellen eines IAM-Benutzers
- Schritt 7: Testen der Berechtigungen
- Schritt 8: Bereinigen Ihrer Ressourcen

Voraussetzungen

Stellen Sie sicher, dass Sie über eine Lambda-Ausführungsrolle verfügen. Sie verwenden diese Rolle, wenn Sie IAM-Berechtigungen erteilen und eine Lambda-Funktion erstellen.

Schritt 1: Tags für neue Funktionen anfordern

Wenn Sie ABAC mit Lambda verwenden, empfiehlt es sich, dass alle Funktionen über Tags verfügen. Auf diese Weise können Sie sicherstellen, dass Ihre ABAC-Berechtigungsrichtlinien wie erwartet funktionieren.

Erstellen Sie eine IAM-Richtlinie ähnlich dem folgenden Beispiel. Diese Richtlinie verwendet die TagKeys Bedingungsschlüssel aws: RequestTag /tag-key, aws: ResourceTag /tag-key und aws:, um zu verlangen, dass sowohl neue Funktionen als auch der IAM-Prinzipal, der die Funktionen erstellt,

das Tag haben. `project` Der `ForAllValues`-Modifikator stellt sicher, dass `project` das einzige zulässige Tag ist. Wenn Sie den `ForAllValues`-Modifikator nicht angeben, können die Benutzer der Funktion andere Tags hinzufügen, solange diese auch `project` übergeben.

Example– Tags für neue Funktionen anfordern

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"
        }
      }
    }
  }
```

Schritt 2: Aktionen basierend auf Tags zulassen, die einer Lambda-Funktion und einem IAM-Prinzipal zugeordnet sind

Erstellen Sie eine zweite IAM-Richtlinie mithilfe des ResourceTagBedingungsschlüssels aws: / tag-key, um zu verlangen, dass das Tag des Prinzipals mit dem Tag übereinstimmt, das an die Funktion angehängt ist. Die folgende Beispielrichtlinie erlaubt es Prinzipalen mit dem `project`-Tag, Funktionen mit dem `project`-Tag aufzurufen. Wenn eine Funktion andere Tags hat, wird die Aktion verweigert.

Example– Erfordert übereinstimmende Tags für Funktion und IAM-Prinzipal

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}"
          }
        }
      }
    ]
  }
```

Schritt 3: Erteilen von Listenberechtigungen

Erstellen Sie eine Richtlinie, die es dem Prinzipal ermöglicht, Lambda-Funktionen und IAM-Rollen aufzulisten. Dadurch kann der Prinzipal alle Lambda-Funktionen und IAM-Rollen auf der Konsole und beim Aufrufen der API-Aktionen sehen.

Example– Erteilen von Lambda- und IAM-Listenberechtigungen

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
      {
        "Sid": "AllResourcesLambdaNoTags",
        "Effect": "Allow",
        "Action": [
```

```
            "lambda:GetAccountSettings",
            "lambda:ListFunctions",
            "iam:ListRoles"
        ],
        "Resource": "*"
      }
    ]
  }
```

Schritt 4: Erteilen von IAM-Berechtigungen

Erstellen Sie eine Richtlinie, die iam: zulässt. PassRole Diese Berechtigung ist erforderlich, wenn Sie einer Funktion eine Ausführungsrolle zuweisen. Ersetzen Sie in der folgenden Beispielrichtlinie den Beispiel-ARN durch den ARN Ihrer Lambda-Ausführungsrolle.

> **Note**
>
> Verwenden Sie den `ResourceTag`-Bedingungsschlüssel nicht in einer Richtlinie mit der `iam:PassRole`-Aktion. Sie können das Tag nicht für eine IAM-Rolle verwenden, um zu steuern, wer Zugriff zur Weiterleitung diese Rolle haben soll. Weitere Informationen zu den Berechtigungen, die für die Übergabe einer Rolle an einen Dienst erforderlich sind, finden Sie unter Einem Benutzer Berechtigungen zur Übergabe einer Rolle an einen AWS Dienst gewähren.

Example– Erteilt die Berechtigung zum Übergeben der Ausführungsrolle

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
      {
        "Sid": "VisualEditor0",
        "Effect": "Allow",
        "Action": [
          "iam:PassRole"
        ],
        "Resource": "arn:aws:iam::111122223333:role/lambda-ex"
```

```
      }
    ]
  }
```

Schritt 5: Erstellen der IAM-Rolle

Es hat sich bewährt, Rollen zum Delegieren von Berechtigungen zu verwenden. Erstellen Sie eine IAM-Rolle mit dem Namen `abac-project-role`:

- In Schritt 1: Vertrauenswürdige Entität auswählen: Wählen Sie wählen AWS -Konto und wählen Sie dann Dieses Konto.
- In Schritt 2: Hinzufügen von Berechtigungen: Fügen Sie die vier IAM-Richtlinien hinzu, die Sie in den vorherigen Schritten erstellt haben.
- In Schritt 3: Benennen, Überprüfen und Erstellen: Wählen Sie Tag hinzufügen. Geben Sie für Key (Schlüssel) `project` ein. Geben Sie keinen Value (Wert) ein.

Schritt 6: Erstellen eines IAM-Benutzers

Erstellen Sie einen IAM-Benutzer mit dem Namen `abac-test-user`. Wählen Sie im Abschnitt Set permissions (Berechtigungen festlegen) die Option Attach existing policies directly (Vorhandene Richtlinien direkt anhängen) und dann Create policy (Richtlinie erstellen) aus. Geben Sie die folgende Richtliniendefinition ein. Ersetzen Sie *111122223333* durch Ihre AWS -Konto-ID. Diese Richtlinie erlaubt es `abac-test-user`, `abac-project-role` anzunehmen.

Example– IAM-Benutzer die Übernahme der ABAC-Rolle erlauben

JSON

```
{
    "Version":"2012-10-17",
    "Statement": {
      "Effect": "Allow",
      "Action": "sts:AssumeRole",
      "Resource": "arn:aws:iam::111122223333:role/abac-project-role"
    }
  }
```

Schritt 7: Testen der Berechtigungen

1. Melden Sie sich an der AWS Konsole an als `abac-test-user`. Weitere Informationen finden Sie unter Anmeldung als IAM-Benutzer.
2. Wechseln Sie zur `abac-project-role`-Rolle. Weitere Informationen finden Sie unter Wechseln zu einer Rolle (Konsole).
3. Erstellen einer Lambda-Funktion:
   - Wählen Sie unter Permissions (Berechtigungen) die Option Change default execution role (Standardausführungsrolle ändern) und dann unter Execution role (Ausführungsrolle) die Option Use an existing role (Vorhandene Rolle verwenden). Wählen Sie dieselbe Ausführungsrolle aus, die Sie in Schritt 4: Erteilen von IAM-Berechtigungen verwendet haben.
   - Wählen Sie unter Advanced settings (Erweiterte Einstellungen) die Option Enable tags (Tags ermöglichen) und wählen Sie dann Add new tag (Neues Tag hinzufügen). Geben Sie für Key (Schlüssel) `project` ein. Geben Sie keinen Value (Wert) ein.
4. Testen der Funktion.
5. Erstellen Sie eine zweite Lambda-Funktion und fügen Sie ein anderes Tag hinzu, z. B. `environment`. Dieser Vorgang sollte fehlschlagen, da die ABAC-Richtlinie, die Sie in Schritt 1: Tags für neue Funktionen anfordern erstellt haben, dem Prinzipal nur erlaubt, Funktionen mit dem `project`-Tag zu erstellen.
6. Erstellen Sie eine dritte Funktion ohne Tags. Dieser Vorgang sollte fehlschlagen, weil die ABAC-Richtlinie, die Sie in Schritt 1: Tags für neue Funktionen anfordern erstellt haben, dem Prinzipal nicht erlaubt, Funktionen ohne Tags zu erstellen.

Mit dieser Autorisierungsstrategie können Sie den Zugriff steuern, ohne für jeden neuen Benutzer neue Richtlinien erstellen zu müssen. Um neuen Benutzern Zugriff zu gewähren, erteilen Sie ihnen einfach die Berechtigung, die Rolle zu übernehmen, die ihrem zugewiesenen Projekt entspricht.

Schritt 8: Bereinigen Ihrer Ressourcen

Löschen Sie die IAM-Rolle wie folgt:

1. Öffnen Sie die Seite Roles (Rollen) in der IAM-Konsole.
2. Wählen Sie die Rolle aus, die Sie im Schritt 5 erstellt haben.
3. Wählen Sie Löschen aus.

4. Geben Sie den Namen der Rolle in das Texteingabefeld ein, um den Löschvorgang zu bestätigen.
5. Wählen Sie Löschen aus.

So löschen Sie den IAM-Benutzer

1. Öffnen Sie die Seite Users in der IAM-Konsole.
2. Wählen Sie den IAM-Benutzer aus, den Sie in Schritt 6 erstellt haben.
3. Wählen Sie Löschen aus.
4. Geben Sie den Namen des Benutzers in das Texteingabefeld ein, um den Löschvorgang zu bestätigen.
5. Wählen Sie Benutzer löschen.

So löschen Sie die Lambda-Funktion:

1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.
2. Wählen Sie die Funktion aus, die Sie erstellt haben.
3. Wählen Sie Aktionen, Löschen aus.
4. Geben Sie **`confirm`** in das Texteingabefeld ein und wählen Sie Delete (Löschen) aus.

Feinabstimmung der Abschnitte „Ressourcen“ und „Bedingungen“ der Richtlinien

Sie können den Umfang der Berechtigungen eines Benutzers einschränken, indem Sie Ressourcen und Bedingungen in einer AWS Identity and Access Management (IAM)-Richtlinie angeben. Jede Aktion in einer Richtlinie unterstützt eine Kombination aus Ressourcen- und Bedingungstypen, die je nach Verhalten der Aktion variiert.

Jede IAM-Richtlinienanweisung erteilt die Berechtigung für eine Aktion, die auf eine Ressource ausgeführt wird. Wenn die Aktion nicht auf eine benannte Ressource reagiert oder Sie die Berechtigung zum Ausführen der Aktion für alle Ressourcen erteilen, wird der Wert der Ressource in der Richtlinie als Platzhalter () dargeste (*). Für viele Aktionen können Sie die Ressourcen einschränken, die ein Benutzer ändern kann, indem Sie den Amazon-Ressourcennamen (ARN) einer Ressource oder ein ARN-Muster angeben, das mehreren Ressourcen entspricht.

Je nach Ressourcentyp lässt sich der Umfang einer Maßnahme generell wie folgt einschränken:

- Funktionen–Aktionen, die auf eine Funktion wirken, können auf eine bestimmte Funktion nach Funktion, Version oder Alias-ARN beschränkt werden.
- Zuordnungen von Ereignisquellen – Aktionen können per ARN auf bestimmte Ressourcen zur Zuordnung von Ereignisquellen beschränkt werden. Zuordnungen von Ereignisquellen sind immer mit einer Funktion verknüpft. Sie können auch die `lambda:FunctionArn`-Bedingung verwenden, um Aktionen durch die zugehörige Funktion einzuschränken.
- Ebenen – Aktionen, die sich auf die Verwendung von Ebenen und Berechtigungen beziehen, wirken sich auf eine Version einer Ebene aus.
- Codesignaturkonfiguration – Aktionen können per ARN auf bestimmte Codesignatur-Konfigurationsressourcen beschränkt werden.
- Tags – Verwenden Sie standardmäßige Tag-Bedingungen. Weitere Informationen finden Sie unter the section called “Attributbasierte Zugriffskontrolle”.

Zum Einschränken von Berechtigungen nach Ressource bestimmen Sie die Ressource unter Angabe des ARN.

Format des Lambda-Ressourcen-ARN

- Funktion – `arn:aws:lambda:`*`us-west-2`*`:`*`123456789012`*`:function:`*`my-function`*
- Funktionsversion – `arn:aws:lambda:`*`us-west-2`*`:`*`123456789012`*`:function:`*`my-function`*`:`*`1`*
- Funktionsalias – `arn:aws:lambda:`*`us-west-2`*`:`*`123456789012`*`:function:`*`my-function`*`:`*`TEST`*
- Mapping von Ereignisquellen – `arn:aws:lambda:`*`us-west-2`*`:`*`123456789012`*`:event-source-mapping:`*`fa123456-14a1-4fd2-9fec-83de64ad683de6d47`*
- Ebene – `arn:aws:lambda:`*`us-west-2`*`:`*`123456789012`*`:layer:`*`my-layer`*
- Ebenenversion – `arn:aws:lambda:`*`us-west-2`*`:`*`123456789012`*`:layer:`*`my-layer`*`:`*`1`*
- Codesignaturkonfiguration – `arn:aws:lambda:`*`us-west-2`*`:`*`123456789012`*`:code-signing-config:`*`my-csc`*

Die folgende Richtlinie ermöglicht es einem Benutzer beispielsweise, eine Funktion AWS-Konto `123456789012` aufzurufen, die `my-function` in der AWS Region USA West (Oregon) benannt ist.

Example Aufrufen der Funktionsrichtlinie

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "Invoke",
            "Effect": "Allow",
            "Action": [
                "lambda:InvokeFunction"
            ],
            "Resource": "arn:aws:lambda:us-west-2:123456789012:function:my-
function"
        }
    ]
}
```

Dies ist ein Sonderfall, da sich die Aktionskennung (`lambda:InvokeFunction`) von der API-Operation (Invoke) unterscheidet. Bei anderen Aktionen ist die Aktionskennung der Operationsname mit dem Präfix `lambda:`.

Sections

- Den Abschnitt „Bedingungen“ in Richtlinien verstehen
- Referenzierung von Funktionen im Abschnitt „Ressourcen“ von Richtlinien
- Unterstützte IAM-Aktionen und Funktionsverhalten

Den Abschnitt „Bedingungen“ in Richtlinien verstehen

Bedingungen sind ein optionales Richtlinienelement, das zusätzliche Logik anwendet, um zu bestimmen, ob eine Aktion zulässig ist. Zusätzlich zu gemeinsamen Bedingungen, die alle Aktionen unterstützen, definiert Lambda Bedingungstypen, die Sie verwenden können, um die Werte zusätzlicher Parameter für einige Aktionen einzuschränken.

Mit der Bedingung `lambda:Principal` können Sie beispielsweise den Service oder das Konto beschränken, für das ein Benutzer in der ressourcenbasierten Richtlinie einer Funktion Aufrufzugriff

gewähren kann. Mit der folgenden Richtlinie kann ein Benutzer Amazon Simple Notification Service (Amazon SNS)-Themen die Berechtigung erteilen, eine Funktion mit dem Namen `test` aufzurufen.

Example Verwalten von Berechtigungen für Funktionsrichtlinien

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"
                }
            }
        }
    ]
}
```

Die Bedingung erfordert, dass der Prinzipal Amazon SNS und kein anderer Service bzw. kein anderes Konto ist. Das Ressourcenmuster erfordert, dass der Funktionsname `test` lautet und eine Versionsnummer oder einen Alias enthält. Beispiel, `test:v1`.

Weitere Informationen zu Ressourcen und Bedingungen für Lambda und andere AWS Dienste finden Sie unter Aktionen, Ressourcen und Bedingungsschlüssel für AWS Dienste in der Service Authorization Reference.

Referenzierung von Funktionen im Abschnitt „Ressourcen“ von Richtlinien

Sie referenzieren eine Lambda-Funktion in einer Richtlinienanweisung mithilfe eines Amazon-Ressourcennamens (ARN). Das Format des ARN für eine Funktion hängt davon ab, ob Sie die

gesamte Funktion, eine Funktionsversion (unqualifiziert), eine Funktionsversion oder einen Alias referenzieren.

Bei Lambda-API-Aufrufen können Benutzer eine Version oder einen Alias angeben, indem sie einen Versions-ARN oder Alias-ARN im GetFunction`FunctionName`Parameter übergeben oder einen Wert im GetFunction`Qualifier`Parameter festlegen. Lambda trifft Autorisierungsentscheidungen, indem es das Ressourcenelement in der IAM-Richtlinie sowohl mit dem in API-Aufrufen übergebenen `FunctionNamen` als auch dem `Qualifier` vergleicht. Wenn es keine Übereinstimmung gibt, lehnt Lambda die Anforderung ab.

Unabhängig davon, ob Sie eine Aktion für Ihre Funktion zulassen oder verweigern, müssen Sie die richtigen ARN-Typen für Funktionen in Ihrer Richtlinienanweisung verwenden, um die erwarteten Ergebnisse zu erzielen. Wenn Ihre Richtlinie beispielsweise auf den unqualifizierten ARN verweist, akzeptiert Lambda Anfragen, die auf den unqualifizierten ARN verweisen, lehnt jedoch Anfragen ab, die auf einen qualifizierten ARN verweisen.

> **Note**
>
> Sie können kein Platzhalterzeichen (*) verwenden, um die Konto-ID abzugleichen. Weitere Informationen zur zulässigen Syntax finden Sie unter IAM JSON-Richtlinienreferenz im IAM-Benutzerhandbuch.

Example Zulassen des Aufrufs eines nicht qualifizierten ARN

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "lambda:InvokeFunction",
            "Resource": "arn:aws:lambda:us-
west-2:123456789012:function:myFunction"
        }
    ]
}
```

Wenn Ihre Richtlinie auf einen bestimmten qualifizierten ARN verweist, akzeptiert Lambda Anfragen, die auf diesen ARN verweisen, lehnt aber Anfragen ab, die auf den nicht qualifizierten ARN oder einen anderen qualifizierten ARN verweisen, z. B. `myFunction:2`.

Example Zulassen des Aufrufs eines bestimmten qualifizierten ARN

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "lambda:InvokeFunction",
            "Resource": "arn:aws:lambda:us-
west-2:123456789012:function:myFunction:1"
        }
    ]
}
```

Wenn Ihre Richtlinie mit `:*` einen qualifizierten ARN referenziert, akzeptiert Lambda alle qualifizierten ARN, lehnt jedoch Anfragen ab, die den unqualifizierten ARN referenzieren.

Example Zulassen des Aufrufs eines beliebigen qualifizierten ARN

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "lambda:InvokeFunction",
            "Resource": "arn:aws:lambda:us-
west-2:123456789012:function:myFunction:*"
        }
    ]
}
```

Wenn Ihre Richtlinie mit * einen ARN referenziert, akzeptiert Lambda alle qualifizierten oder unqualifizierten ARNs.

Example Zulassen des Aufrufs eines beliebigen qualifizierten oder nicht qualifizierten ARN

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "lambda:InvokeFunction",
            "Resource": "arn:aws:lambda:us-
west-2:123456789012:function:myFunction*"
        }
    ]
}
```

Unterstützte IAM-Aktionen und Funktionsverhalten

Aktionen definieren, was durch IAM-Richtlinien zugelassen werden kann. Eine Liste der in Lambda unterstützten Aktionen finden Sie unter Aktionen, Ressourcen und Bedingungsschlüssel für AWS Lambda in der Referenz zur Dienstautorisierung. Wenn eine IAM-Aktion eine Lambda-API-Aktion zulässt, ist der Name der IAM-Aktion in den meisten Fällen derselbe wie der Name der Lambda-API-Aktion, mit den folgenden Ausnahmen:

API-Aktion	IAM-Aktion
Invoke	`lambda:InvokeFunction`
GetLayerVersion GetLayerVersionByArn	`lambda:GetLayerVersion`

Zusätzlich zu den Ressourcen und Bedingungen, die in der Referenz für die Service-Autorisierung definiert sind, unterstützt Lambda die folgenden Ressourcen und Bedingungen für bestimmte Aktionen. Viele davon beziehen sich auf Referenzfunktionen im Ressourcenbereich der Richtlinien.

Aktionen, die für eine Funktion verwender werden, können – wie in der folgenden Tabelle beschrieben – nach Funktion, Version oder Alias-ARN auf eine spezifische Funktion eingeschränkt werden.

Action	Ressource	Bedingung
AddPermission RemovePermission Aufrufen (Berechtigung: `lambda:InvokeFunction`)	Funktionsversion Funktionsalias	–
UpdateFunctionConfiguration	–	`lambda:CodeSigningConfigArn`
CreateFunctionUrlConfig DeleteFunctionUrlConfig GetFunctionUrlConfig UpdateFunctionUrlConfig	Funktionsalias	–

Sicherheit in AWS Lambda

Cloud-Sicherheit AWS hat höchste Priorität. Als AWS Kunde profitieren Sie von einer Rechenzentrums- und Netzwerkarchitektur, die darauf ausgelegt sind, die Anforderungen der sicherheitssensibelsten Unternehmen zu erfüllen.

Sicherheit ist eine gemeinsame Verantwortung von Ihnen AWS und Ihnen. Das Modell der geteilten Verantwortung beschreibt dies als Sicherheit der Cloud und Sicherheit in der Cloud:

- Sicherheit der Cloud — AWS ist verantwortlich für den Schutz der Infrastruktur, die AWS-Services in der AWS Cloud läuft. AWS bietet Ihnen auch Dienste, die Sie sicher nutzen können. Auditoren von Drittanbietern testen und überprüfen die Effektivität unserer Sicherheitsmaßnahmen im Rahmen der AWS -Compliance-Programme regelmäßig. Weitere Informationen zu den Compliance-Programmen, die für gelten AWS Lambda, finden Sie AWS-Services unter Umfang nach Compliance-Programmen.
- Sicherheit in der Cloud – Ihr Verantwortungsumfang wird durch den AWS -Dienst bestimmt, den Sie verwenden. Sie sind auch für andere Faktoren verantwortlich, etwa für die Vertraulichkeit Ihrer Daten, für die Anforderungen Ihres Unternehmens und für die geltenden Gesetze und Vorschriften.

Diese Dokumentation erläutert, wie das Modell der geteilten Verantwortung bei der Verwendung von Lambda zum Tragen kommt. Die folgenden Themen veranschaulichen, wie Sie Lambda zur Erfüllung Ihrer Sicherheits- und Compliance-Ziele konfigurieren können. Sie lernen auch, wie Sie andere verwenden können AWS-Services , die Ihnen helfen, Ihre Lambda-Ressourcen zu überwachen und zu sichern.

Weitere Informationen zur Anwendung von Sicherheitsprinzipien auf Lambda-Anwendungen finden Sie bei Serverless Land unter Sicherheit.

Topics

- Datenschutz in AWS Lambda
- Verwenden von serviceverknüpften Rollen für Lambda
- Identitäts- und Zugriffsverwaltung für AWS Lambda
- Erstellen Sie eine Governance-Strategie für Lambda-Funktionen und -Layer
- Compliance-Validierung für AWS Lambda
- Ausfallsicherheit in AWS Lambda

- Sicherheit der Infrastruktur in AWS Lambda
- Sicherung von Workloads mit öffentlichen Endpunkten
- Verwendung von Codesignatur zur Überprüfung der Codeintegrität mit Lambda

Datenschutz in AWS Lambda

Das Modell der geteilten Verantwortung von AWS gilt für den Datenschutz in AWS Lambda. Wie in diesem Modell beschrieben, ist AWS verantwortlich für den Schutz der globalen Infrastruktur, in der die gesamte AWS Cloud ausgeführt wird. Sie sind dafür verantwortlich, die Kontrolle über Ihre in dieser Infrastruktur gehosteten Inhalte zu behalten. Sie sind auch für die Sicherheitskonfiguration und die Verwaltungsaufgaben für die von Ihnen verwendeten AWS-Services verantwortlich. Weitere Informationen zum Datenschutz finden Sie unter Häufig gestellte Fragen zum Datenschutz. Informationen zum Datenschutz in Europa finden Sie im Blog-Beitrag AWS-Modell der geteilten Verantwortung und in der DSGVO im AWS-Sicherheitsblog.

Aus Datenschutzgründen empfehlen wir, AWS-Konto-Anmeldeinformationen zu schützen und einzelne Benutzer mit AWS IAM Identity Center oder AWS Identity and Access Management (IAM) einzurichten. So erhält jeder Benutzer nur die Berechtigungen, die zum Durchführen seiner Aufgaben erforderlich sind. Außerdem empfehlen wir, die Daten mit folgenden Methoden schützen:

- Verwenden Sie für jedes Konto die Multi-Faktor-Authentifizierung (MFA).
- Verwenden Sie SSL/TLS für die Kommunikation mit AWS-Ressourcen. Wir benötigen TLS 1.2 und empfehlen TLS 1.3.
- Richten Sie die API und die Protokollierung von Benutzeraktivitäten mit AWS CloudTrail ein. Informationen zur Verwendung von CloudTrail-Trails zur Erfassung von AWS-Aktivitäten finden Sie unter Arbeiten mit CloudTrail-Trails im AWS CloudTrail-Benutzerhandbuch.
- Verwenden Sie AWS-Verschlüsselungslösungen zusammen mit allen Standardsicherheitskontrollen in AWS-Services.
- Verwenden Sie erweiterte verwaltete Sicherheitsservices wie Amazon Macie, die dabei helfen, in Amazon S3 gespeicherte persönliche Daten zu erkennen und zu schützen.
- Wenn Sie für den Zugriff auf AWS über eine Befehlszeilenschnittstelle oder über eine API FIPS 140-3-validierte kryptografische Module benötigen, verwenden Sie einen FIPS-Endpunkt. Weitere Informationen über verfügbare FIPS-Endpunkte finden Sie unter Federal Information Processing Standard (FIPS) 140-3.

Wir empfehlen dringend, in Freitextfeldern, z. B. im Feld Name, keine vertraulichen oder sensiblen Informationen wie die E-Mail-Adressen Ihrer Kunden einzugeben. Dies gilt auch, wenn Sie mit Lambda oder anderen AWS-Services über die Konsole, API, AWS CLI oder AWS-SDKs arbeiten. Alle Daten, die Sie in Tags oder Freitextfelder eingeben, die für Namen verwendet werden, können für Abrechnungs- oder Diagnoseprotokolle verwendet werden. Wenn Sie eine URL für einen externen Server bereitstellen, empfehlen wir dringend, keine Anmeldeinformationen zur Validierung Ihrer Anforderung an den betreffenden Server in die URL einzuschließen.

Abschnitte

- Verschlüsselung während der Übertragung
- Verschlüsselung der Daten im Ruhezustand für AWS Lambda

Verschlüsselung während der Übertragung

Lambda-API-Endpunkte unterstützen ausschließlich sichere Verbindungen über HTTPS. Wenn Sie Lambda-Ressourcen mit der AWS-Managementkonsole, dem AWS SDK oder der Lambda-API verwalten, wird die gesamte Kommunikation mit Transport Layer Security (TLS) verschlüsselt. Eine vollständige Liste der API-Endpunkte finden Sie unter AWSRegionen und Endpunkte in Allgemeine AWS-Referenz.

Wenn Sie Ihre Funktion mit einem Dateisystem verbinden, verwendet Lambda Verschlüsselung während der Übertragung für alle Verbindungen. Weitere Informationen finden Sie unter Datenverschlüsselung in Amazon EFS im Benutzerhandbuch zu Amazon Elastic File System.

Wenn Sie Umgebungsvariablen verwenden, können Sie Konsolenverschlüsselungshelfer aktivieren, um die clientseitige Verschlüsselung zu verwenden, um die Umgebungsvariablen während der Übertragung zu schützen. Weitere Informationen finden Sie unter Sicherung von Lambda-Umgebungsvariablen.

Verschlüsselung der Daten im Ruhezustand für AWS Lambda

Lambda bietet immer Verschlüsselung im Ruhezustand für die folgenden Ressourcen mit einem AWS-eigener Schlüssel oder einem Von AWS verwalteter Schlüssel:

- Umgebungsvariablen
- Dateien, die Sie zu Lambda hochladen, einschließlich Bereitstellungspakete und Ebenenarchive
- Objekte mit Filterkriterien für die Zuordnung von Ereignisquellen

Sie können Lambda optional so konfigurieren, dass es einen vom Kunden verwalteten Schlüssel verwendet, um Ihre Umgebungsvariablen, .zip-Bereitstellungspakete und Filterkriterienobjekte zu verschlüsseln.

Amazon CloudWatch Logs und AWS X-Ray verschlüsseln Daten darüber hinaus standardmäßig und können für die Verwendung kundenverwalteter Schlüssel konfiguriert werden. Details hierzu finden Sie unter Verschlüsseln von Protokolldaten in CloudWatch Logs und Datenschutz in AWS X-Ray.

Überwachen Ihrer Verschlüsselungsschlüssel für Lambda

Wenn Sie einen von AWS KMS-Kunden verwalteten Schlüssel mit Lambda verwenden, können Sie AWS CloudTrail verwenden. Die folgenden Beispiele sind CloudTrail-Ereignisse für `Decrypt`-, `DescribeKey`- und `GenerateDataKey`-Aufrufe von Lambda, um auf Daten zuzugreifen, die mit Ihrem vom Kunden verwalteten Schlüssel verschlüsselt sind.

Decrypt

Wenn Sie einen vom Kunden verwalteten AWS KMS-Schlüssel verwendet haben, um Ihr Filterkriterienobjekt zu verschlüsseln, sendet Lambda in Ihrem Namen eine `Decrypt`-Anforderung, wenn Sie versuchen, im Klartext darauf zuzugreifen (z. B. über einen `ListEventSourceMappings`-Aufruf). Das folgende Beispielereignis zeichnet den Vorgang `Decrypt` auf:

```
{
    "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

Wenn Sie einen vom Kunden verwalteten AWS KMS-Schlüssel verwendet haben, um Ihr Filterkriterienobjekt zu verschlüsseln, sendet Lambda in Ihrem Namen eine `DescribeKey`-Anforderung, wenn Sie versuchen, darauf zuzugreifen (z. B. über einen `GetEventSourceMapping`-Aufruf). Das folgende Beispielereignis zeichnet den Vorgang `DescribeKey` auf:

```
{
    "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

Wenn Sie einen vom Kunden verwalteten AWS KMS-Schlüssel verwenden, um Ihr Filterkriterienobjekt in einem `CreateEventSourceMapping`- oder `UpdateEventSourceMapping`-Aufruf zu verschlüsseln, sendet Lambda in Ihrem Namen eine `GenerateDataKey`-Anfrage, um einen Datenschlüssel zur Verschlüsselung der Filterkriterien zu generieren (Umschlagverschlüsselung). Das folgende Beispielereignis zeichnet den Vorgang `GenerateDataKey` auf:

```
{
    "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"
}
```

Verwenden von serviceverknüpften Rollen für Lambda

Lambda verwendet AWS Identity and Access Management (IAM) serviceverknüpfte Rollen. Eine serviceverknüpfte Rolle ist eine einzigartige Art von IAM-Rolle, die direkt mit Lambda verknüpft ist. Dienstbezogene Rollen sind von Lambda vordefiniert und enthalten Berechtigungen, die der Dienst benötigt, um andere AWS Dienste in Ihrem Namen aufzurufen.

Lambda definiert die Berechtigungen seiner dienstbezogenen Rollen, und nur Lambda kann seine Rollen übernehmen. Die definierten Berechtigungen umfassen die Vertrauens- und Berechtigungsrichtlinie. Diese Berechtigungsrichtlinie kann keinen anderen IAM-Entitäten zugewiesen werden.

Sie können eine serviceverknüpfte Rolle erst löschen, nachdem ihre verwandten Ressourcen gelöscht wurden. Dies schützt Ihre Lambda-Ressourcen, da Sie die Zugriffsberechtigung für die Ressourcen nicht versehentlich entfernen können.

Informationen zu anderen Diensten, die dienstverknüpfte Rollen unterstützen, finden Sie unter AWS Dienste, die mit IAM funktionieren. Suchen Sie in der Spalte Dienstverknüpfte Rollen nach den Diensten, für die Ja steht. Wählen Sie über einen Link Ja aus, um die Dokumentation zu einer serviceverknüpften Rolle für diesen Service anzuzeigen.

Dienstbezogene Rollenberechtigungen für Lambda

Lambda verwendet die mit dem Dienst verknüpfte Rolle namens. AWSServiceRoleForLambda Die serviceverknüpfte Rolle vertraut darauf, dass die folgenden Services die Rolle annehmen:

- `lambda.amazonaws.com`

Die genannte Rollenberechtigungsrichtlinie AWSLambda ServiceRolePolicy ermöglicht es Lambda, die folgenden Aktionen für die angegebenen Ressourcen durchzuführen:

- Aktion: `ec2:TerminateInstances` aktiviert `arn:aws:ec2:*:*:instance/*` mit der Bedingung, die entspricht `ec2:ManagedResourceOperator` `scaler.lambda.amazonaws.com`
- Aktion: `ec2:DescribeInstanceStatus` und `ec2:DescribeInstances` auf `*`

Sie müssen Berechtigungen konfigurieren, damit eine Benutzer, Gruppen oder Rollen eine serviceverknüpfte Rolle erstellen, bearbeiten oder löschen können. Weitere Informationen finden Sie unter serviceverknüpfte Rollenberechtigung im IAM-Benutzerhandbuch.

Informationen zu verwalteten Richtlinienaktualisierungen finden Sie unter Verwaltete Lambda-Richtlinien.

Eine serviceverknüpfte Rolle für Lambda erstellen

Sie müssen eine serviceverknüpfte Rolle nicht manuell erstellen. Wenn Sie einen Lambda-Kapazitätsanbieter in der AWS-Managementkonsole, der oder der AWS API erstellen AWS CLI, erstellt Lambda die serviceverknüpfte Rolle für Sie.

Wenn Sie diese serviceverknüpfte Rolle löschen und sie dann erneut erstellen müssen, können Sie dasselbe Verfahren anwenden, um die Rolle in Ihrem Konto neu anzulegen. Wenn Sie einen Lambda-Kapazitätsanbieter erstellen, erstellt Lambda die serviceverknüpfte Rolle erneut für Sie.

Sie können auch die IAM-Konsole verwenden, um eine serviceverknüpfte Rolle mit dem Anwendungsfall zu erstellen. AWSServiceRoleForLambda Erstellen Sie in der AWS CLI oder der AWS API eine dienstverknüpfte Rolle mit dem `lambda.amazonaws.com` Dienstnamen. Weitere Informationen finden Sie unter Erstellen einer serviceverknüpften Rolle im IAM-Benutzerhandbuch. Wenn Sie diese serviceverknüpfte Rolle löschen, können Sie mit demselben Verfahren die Rolle erneut erstellen.

Bearbeiten einer serviceverknüpften Rolle für Lambda

Lambda erlaubt Ihnen nicht, die AWSService RoleForLambda serviceverknüpfte Rolle zu bearbeiten. Da möglicherweise verschiedene Entitäten auf die Rolle verweisen, kann der Rollenname nach dem Erstellen einer serviceverknüpften Rolle nicht mehr geändert werden. Sie können jedoch die Beschreibung der Rolle mit IAM bearbeiten. Weitere Informationen finden Sie unter Bearbeiten einer serviceverknüpften Rolle im IAM-Benutzerhandbuch.

Löschen einer serviceverknüpften Rolle für Lambda

Wenn Sie ein Feature oder einen Dienst, die bzw. der eine serviceverknüpften Rolle erfordert, nicht mehr benötigen, sollten Sie diese Rolle löschen. Auf diese Weise haben Sie keine ungenutzte juristische Stelle, die nicht aktiv überwacht oder verwaltet wird. Sie müssen jedoch die Ressourcen für Ihre serviceverknüpften Rolle zunächst bereinigen, bevor Sie sie manuell löschen können.

> **Note**
>
> Wenn der Lambda-Dienst die Rolle verwendet, wenn Sie versuchen, die Ressourcen zu löschen, schlägt das Löschen möglicherweise fehl. Wenn dies passiert, warten Sie einige Minuten und versuchen Sie es erneut.

Um Lambda-Ressourcen zu löschen, die von der AWSService RoleForLambda

1. Entfernen Sie alle Lambda-Kapazitätsanbieter aus Ihrem Konto. Sie können dies mit der Lambda-Konsole, CLI oder API tun.
2. Stellen Sie sicher, dass in Ihrem Konto keine Lambda-Kapazitätsanbieter mehr vorhanden sind, bevor Sie versuchen, die serviceverknüpfte Rolle zu löschen.

So löschen Sie die serviceverknüpfte Rolle mit IAM

Verwenden Sie die IAM-Konsole, die oder die AWS API AWS CLI, um die serviceverknüpfte Rolle zu löschen. AWSService RoleForLambda Weitere Informationen finden Sie unter Löschen einer serviceverknüpften Rolle im IAM-Benutzerhandbuch.

Unterstützte Regionen für dienstverknüpfte Lambda-Rollen

Lambda unterstützt nicht die Verwendung von dienstbezogenen Rollen in jeder Region, in der der Dienst verfügbar ist. AWSServiceRoleForLambda wird in den folgenden Regionen unterstützt.

Name der Region	Regions-ID	Support in Lambda
USA Ost (Nord-Virginia)	us-east-1	Ja
USA Ost (Ohio)	us-east-2	Ja
USA West (Nordkalifornien)	us-west-1	Ja
USA West (Oregon)	us-west-2	Ja
Afrika (Kapstadt)	af-south-1	Nein
Asien-Pazifik (Hongkong)	ap-east-1	Ja

Name der Region	Regions-ID	Support in Lambda
Asien-Pazifik (Jakarta)	ap-southeast-3	Ja
Asien-Pazifik (Bangkok)	ap-southeast-7	Ja
Asien-Pazifik (Mumbai)	ap-south-1	Ja
Asien-Pazifik (Osaka)	ap-northeast-3	Nein
Asien-Pazifik (Seoul)	ap-northeast-2	Nein
Asien-Pazifik (Singapur)	ap-southeast-1	Ja
Asien-Pazifik (Sydney)	ap-southeast-2	Ja
Asien-Pazifik (Tokyo)	ap-northeast-1	Ja
Kanada (Zentral)	ca-central-1	Nein
Europa (Frankfurt)	eu-central-1	Ja
Europa (Ireland)	eu-west-1	Ja
Europa (London)	eu-west-2	Ja
Europa (Milan)	eu-south-1	Nein
Europa (Paris)	eu-west-3	Nein
Europa (Stockholm)	eu-north-1	Nein
Naher Osten (Bahrain)	me-south-1	Nein
Naher Osten (VAE)	me-central-1	Nein
Südamerika (São Paulo)	sa-east-1	Nein
AWS GovCloud (US-Ost)	us-gov-east-1	Nein
AWS GovCloud (US-West)	us-gov-west-1	Nein

Identitäts- und Zugriffsverwaltung für AWS Lambda

AWS Identity and Access Management (IAM) ist ein AWS-Service, mit dem Administratoren den Zugriff auf AWS-Ressourcen sicher steuern können. IAM-Administratoren steuern, wer authentifiziert (angemeldet) und autorisiert (Berechtigungen besitzt) ist, um Lambda-Ressourcen zu nutzen. IAM ist ein AWS-Service, den Sie ohne zusätzliche Kosten verwenden können.

Themen

- Zielgruppe
- Authentifizierung mit Identitäten
- Verwalten des Zugriffs mit Richtlinien
- Funktionsweise von AWS Lambda mit IAM
- Beispiele für identitätsbasierte Richtlinien für AWS Lambda
- AWS verwaltete Richtlinien für AWS Lambda
- Fehlerbehebung für AWS Lambda-Identität und -Zugriff

Zielgruppe

Wie Sie AWS Identity and Access Management (IAM) verwenden, hängt von Ihrer Rolle ab:

- Servicebenutzer – Fordern Sie von Ihrem Administrator Berechtigungen an, wenn Sie nicht auf Funktionen zugreifen können (siehe Fehlerbehebung für AWS Lambda-Identität und -Zugriff).
- Serviceadministrator – Bestimmen Sie den Benutzerzugriff und stellen Sie Berechtigungsanfragen (siehe Funktionsweise von AWS Lambda mit IAM).
- IAM-Administrator – Schreiben Sie Richtlinien zur Zugriffsverwaltung (siehe Beispiele für identitätsbasierte Richtlinien für AWS Lambda).

Authentifizierung mit Identitäten

Sie melden sich über eine Authentifizierung mit Ihren Anmeldeinformationen bei AWS an. Die Authentifizierung muss als Root-Benutzer des AWS-Kontos, als IAM-Benutzer oder durch Übernahme einer IAM-Rolle erfolgen.

Sie können sich als Verbundidentität anmelden, indem Sie Anmeldeinformationen aus einer Identitätsquelle wie AWS IAM Identity Center (IAM Identity Center), Single-Sign-On-Authentifizierung oder Google-/Facebook-Anmeldeinformationen verwenden. Weitere Informationen zum Anmelden finden Sie unter So melden Sie sich bei Ihrem AWS-Konto an im Benutzerhandbuch für AWS-Anmeldung.

AWS bietet für den programmatischen Zugriff ein SDK und eine CLI zum kryptografischen Signieren von Anfragen. Weitere Informationen finden Sie unter AWS Signature Version 4 for API requests im IAM-Benutzerhandbuch.

AWS-Konto-Root-Benutzer

Wenn Sie ein AWS-Konto neu erstellen, enthält es zunächst nur eine einzelne Anmeldeidentität, die als Root-Benutzer für das AWS-Konto bezeichnet wird und über kompletten Zugriff auf sämtliche AWS-Services und Ressourcen im Konto verfügt. Wir raten ausdrücklich davon ab, den Root-Benutzer für Alltagsaufgaben zu verwenden. Eine Liste der Aufgaben, für die Sie sich als Root-Benutzer anmelden müssen, finden Sie unter Tasks that require root user credentials im IAM-Benutzerhandbuch.

Verbundidentität

Fordern Sie als bewährte Methode menschliche Benutzer auf, den Verbund mit einem Identitätsanbieter zu verwenden, um mit temporären Anmeldeinformationen auf AWS-Services zuzugreifen.

Eine Verbundidentität ist ein Benutzer aus Ihrem Unternehmensverzeichnis, Ihrem Web-Identitätsanbieter oder Directory Service, der mithilfe von Anmeldeinformationen aus einer Identitätsquelle auf AWS-Services zugreift. Verbundidentitäten übernehmen Rollen, die temporäre Anmeldeinformationen bereitstellen.

Für die zentrale Zugriffsverwaltung empfehlen wir AWS IAM Identity Center. Weitere Informationen finden Sie unter Was ist IAM Identity Center? im AWS IAM Identity Center-Benutzerhandbuch.

IAM-Benutzer und -Gruppen

Ein IAM-Benutzer ist eine Identität mit bestimmten Berechtigungen für eine einzelne Person oder Anwendung. Verwenden Sie möglichst temporäre Anmeldeinformationen statt IAM-Benutzer mit langfristigen Anmeldeinformationen. Weitere Informationen finden Sie unter Menschliche Benutzer auffordern, den Verbund mit einem Identitätsanbieter zu verwenden, um mit temporären Anmeldeinformationen auf AWS zuzugreifen im IAM-Benutzerhandbuch.

Eine IAM-Gruppe gibt eine Sammlung von IAM-Benutzern an und vereinfacht die Verwaltung von Berechtigungen bei großer Benutzerzahl. Weitere Informationen finden Sie unter Use cases for IAM users im IAM-Benutzerhandbuch.

IAM-Rollen

Eine IAM-Rolle ist eine Identität mit bestimmten Berechtigungen, die temporäre Anmeldeinformationen bereitstellt. Sie können eine IAM-Rolle übernehmen, indem Sie von einem Benutzer zu einer IAM-Rolle (Konsole) wechseln oder eine AWS CLI- bzw. AWS-API-Operation aufrufen. Weitere Informationen finden Sie unter Methods to assume a role im IAM-Benutzerhandbuch.

IAM-Rollen sind nützlich für Verbundbenutzerzugriff, temporäre IAM-Benutzerberechtigungen, kontoübergreifenden Zugriff, serviceübergreifenden Zugriff und Anwendungen, die in Amazon EC2 ausgeführt werden. Weitere Informationen finden Sie unter Kontoübergreifender Ressourcenzugriff in IAM im IAM-Benutzerhandbuch.

Verwalten des Zugriffs mit Richtlinien

Für die Zugriffssteuerung in AWS erstellen Sie Richtlinien und weisen diese den AWS-Identitäten oder -Ressourcen zu. Eine Richtlinie definiert Berechtigungen, wenn sie einer Identität oder Ressource zugeordnet wird. AWS wertet diese Richtlinien aus, sobald ein Prinzipal eine Anfrage stellt. Die meisten Richtlinien werden in AWS als JSON-Dokumente gespeichert. Weitere Informationen zu JSON-Richtliniendokumenten finden Sie unter Overview of JSON policies im IAM-Benutzerhandbuch.

Mithilfe von Richtlinien legen Administratoren fest, wer auf was Zugriff hat, indem sie definieren, welcher Prinzipal Aktionen mit welchen Ressourcen und unter welchen Bedingungen ausführen kann.

Standardmäßig haben Benutzer, Gruppen und Rollen keine Berechtigungen. Ein IAM-Administrator erstellt IAM-Richtlinien und fügt sie Rollen hinzu, die Benutzer dann übernehmen können. IAM-Richtlinien definieren Berechtigungen unabhängig von der Methode, die zur Ausführung der Operation verwendet wird.

Identitätsbasierte Richtlinien

Identitätsbasierte Richtlinien sind JSON-Berechtigungsrichtliniendokumente, die Sie einer Identität (Benutzer, Gruppe oder Rolle) anfügen können. Diese Richtlinien steuern, welche Aktionen Identitäten für welche Ressourcen und unter welchen Bedingungen ausführen können. Informationen

zum Erstellen identitätsbasierter Richtlinien finden Sie unter Definieren benutzerdefinierter IAM-Berechtigungen mit vom Kunden verwalteten Richtlinien im IAM-Benutzerhandbuch.

Identitätsbasierte Richtlinien können Inline-Richtlinien (Richtlinien, die direkt in eine einzelne Identität eingebettet sind) oder verwaltete Richtlinien (eigenständige Richtlinien, die mehreren Identitäten zugeordnet sind) sein. Informationen dazu, wie Sie zwischen verwalteten und Inline-Richtlinien wählen, finden Sie unter Choose between managed policies and inline policies im IAM-Benutzerhandbuch.

Ressourcenbasierte Richtlinien

Ressourcenbasierte Richtlinien sind JSON-Richtliniendokumente, die Sie an eine Ressource anfügen. Beispiele hierfür sind Vertrauensrichtlinien für IAM-Rollen und Bucket-Richtlinien von Amazon S3. In Services, die ressourcenbasierte Richtlinien unterstützen, können Service-Administratoren sie verwenden, um den Zugriff auf eine bestimmte Ressource zu steuern. Sie müssen in einer ressourcenbasierten Richtlinie einen Prinzipal angeben.

Ressourcenbasierte Richtlinien sind Richtlinien innerhalb dieses Diensts. Sie können verwaltete AWS-Richtlinien von IAM nicht in einer ressourcenbasierten Richtlinie verwenden.

Weitere Richtlinientypen

AWS unterstützt zusätzliche Richtlinientypen, die die maximalen Berechtigungen festlegen, welche von den häufiger verwendeten Richtlinientypen erteilt werden können:

- Berechtigungsgrenzen – Eine Berechtigungsgrenze legt die maximalen Berechtigungen fest, die eine identitätsbasierte Richtlinie einer IAM-Entität erteilen kann. Weitere Informationen finden Sie unter Berechtigungsgrenzen für IAM-Entitäten im IAM-Benutzerhandbuch.
- Service-Kontrollrichtlinien (SCPs) – SCPs legen die maximalen Berechtigungen für eine Organisation oder Organisationseinheit in AWS Organizations fest. Weitere Informationen finden Sie unter Service-Kontrollrichtlinien im AWS Organizations-Benutzerhandbuch.
- Ressourcen-Kontrollrichtlinien (RCPs) – RCPs definieren die maximale Anzahl an Berechtigungen, die Ressourcen in Ihren Konten zur Verfügung stehen. Weitere Informationen finden Sie unter Ressourcen-Kontrollrichtlinien im AWS Organizations-Benutzerhandbuch.
- Sitzungsrichtlinien – Sitzungsrichtlinien sind erweiterte Richtlinien, die als Parameter übergeben werden, wenn Sie eine temporäre Sitzung für eine Rolle oder einen Verbundbenutzer erstellen. Weitere Informationen finden Sie unter Sitzungsrichtlinien im IAM-Benutzerhandbuch.

Mehrere Richtlinientypen

Wenn mehrere auf eine Anforderung mehrere Richtlinientypen angewendet werden können, sind die entsprechenden Berechtigungen komplizierter. Informationen dazu, wie AWS die Zulässigkeit einer Anforderung ermittelt, wenn mehrere Richtlinientypen beteiligt sind, finden Sie unter Logik für die Richtlinienauswertung im IAM-Benutzerhandbuch.

Funktionsweise von AWS Lambda mit IAM

Bevor Sie IAM zum Verwalten des Zugriffs auf Lambda verwenden, erfahren Sie, welche IAM-Features Sie mit Lambda verwenden können.

IAM-Feature	Lambda-Unterstützung
Identitätsbasierte Richtlinien	Ja
Ressourcenbasierte Richtlinien	Ja
Richtlinienaktionen	Ja
Richtlinienressourcen	Ja
Richtlinienbedingungsschlüssel (services pezifisch)	Ja
ACLs	Nein
ABAC (Tags in Richtlinien)	Teilweise
Temporäre Anmeldeinformationen	Ja
Forward Access Sessions (FAS)	Nein
Servicerollen	Ja
Service-verknüpfte Rollen	Teilweise

Einen Überblick über das Zusammenwirken von Lambda und anderen AWS-Services mit den meisten IAM-Features finden Sie unter AWS-Services, die mit IAM funktionieren im IAM-Benutzerhandbuch.

Identitätsbasierte Richtlinien für Lambda

Unterstützt Richtlinien auf Identitätsbasis: Ja

Identitätsbasierte Richtlinien sind JSON-Berechtigungsrichtliniendokumente, die Sie einer Identität anfügen können, wie z. B. IAM-Benutzern, -Benutzergruppen oder -Rollen. Diese Richtlinien steuern, welche Aktionen die Benutzer und Rollen für welche Ressourcen und unter welchen Bedingungen ausführen können. Informationen zum Erstellen identitätsbasierter Richtlinien finden Sie unter Definieren benutzerdefinierter IAM-Berechtigungen mit vom Kunden verwalteten Richtlinien im IAM-Benutzerhandbuch.

Mit identitätsbasierten IAM-Richtlinien können Sie angeben, welche Aktionen und Ressourcen zugelassen oder abgelehnt werden. Darüber hinaus können Sie die Bedingungen festlegen, unter denen Aktionen zugelassen oder abgelehnt werden. Informationen zu sämtlichen Elementen, die Sie in einer JSON-Richtlinie verwenden, finden Sie in der IAM-Referenz für JSON-Richtlinienelemente im IAM-Benutzerhandbuch.

Beispiele für identitätsbasierte Richtlinien für Lambda

Beispiele für identitätsbasierte Lambda-Richtlinien finden Sie unter Beispiele für identitätsbasierte Richtlinien für AWS Lambda.

Ressourcenbasierte Richtlinien in Lambda

Unterstützt ressourcenbasierte Richtlinien: Ja

Ressourcenbasierte Richtlinien sind JSON-Richtliniendokumente, die Sie an eine Ressource anfügen. Beispiele für ressourcenbasierte Richtlinien sind IAM-Rollen-Vertrauensrichtlinien und Amazon-S3-Bucket-Richtlinien. In Services, die ressourcenbasierte Richtlinien unterstützen, können Service-Administratoren sie verwenden, um den Zugriff auf eine bestimmte Ressource zu steuern. Für die Ressource, an welche die Richtlinie angehängt ist, legt die Richtlinie fest, welche Aktionen ein bestimmter Prinzipal unter welchen Bedingungen für diese Ressource ausführen kann. Sie müssen in einer ressourcenbasierten Richtlinie einen Prinzipal angeben. Prinzipale können Konten, Benutzer, Rollen, Verbundbenutzer oder AWS-Services umfassen.

Um kontoübergreifenden Zugriff zu ermöglichen, können Sie ein gesamtes Konto oder IAM-Entitäten in einem anderen Konto als Prinzipal in einer ressourcenbasierten Richtlinie angeben. Weitere Informationen finden Sie unter Kontoübergreifender Ressourcenzugriff in IAM im IAM-Benutzerhandbuch.

Sie können eine ressourcenbasierte Richtlinie an eine Lambda-Funktion oder -Ebene anhängen. Diese Richtlinie legt fest, welche Prinzipale Aktionen auf der Funktion oder Ebene durchführen können.

Weitere Informationen zum Anfügen einer ressourcenbasierten Richtlinie an eine Funktion oder eine Ebene finden Sie unter Anzeigen von ressourcenbasierten IAM-Richtlinien in Lambda.

Richtlinienaktionen für Lambda

Unterstützt Richtlinienaktionen: Ja

Administratoren können mit AWS-JSON-Richtlinien festlegen, welche Personen zum Zugriff auf welche Ressourcen berechtigt sind. Das heißt, welcher Prinzipal Aktionen für welche Ressourcen und unter welchen Bedingungen ausführen kann.

Das Element `Action` einer JSON-Richtlinie beschreibt die Aktionen, mit denen Sie den Zugriff in einer Richtlinie zulassen oder verweigern können. Schließen Sie Aktionen in eine Richtlinie ein, um Berechtigungen zur Durchführung der zugeordneten Operation zu erteilen.

Eine Liste der Lambda-Aktionen finden Sie unter Von AWS Lambda definierte Aktionen in der Service-Autorisierungs-Referenz.

Richtlinienaktionen in Lambda verwenden das folgende Präfix vor der Aktion:

```
lambda
```

Um mehrere Aktionen in einer einzigen Anweisung anzugeben, trennen Sie sie mit Kommata:

```
"Action": [
      "lambda:action1",
      "lambda:action2"
         ]
```

Beispiele für identitätsbasierte Lambda-Richtlinien finden Sie unter Beispiele für identitätsbasierte Richtlinien für AWS Lambda.

Richtlinienressourcen für Lambda

Unterstützt Richtlinienressourcen: Ja

Administratoren können mit AWS-JSON-Richtlinien festlegen, welche Personen zum Zugriff auf welche Ressourcen berechtigt sind. Das heißt, welcher Prinzipal Aktionen für welche Ressourcen und unter welchen Bedingungen ausführen kann.

Das JSON-Richtlinienelement `Resource` gibt die Objekte an, auf welche die Aktion angewendet wird. Als bewährte Methode geben Sie eine Ressource mit dem zugehörigen Amazon-Ressourcennamen (ARN) an. Verwenden Sie für Aktionen, die keine Berechtigungen auf Ressourcenebene unterstützen, einen Platzhalter (*), um anzugeben, dass die Anweisung für alle Ressourcen gilt.

```
"Resource": "*"
```

Eine Liste der Lambda-Ressourcentypen und ihrer ARNs finden Sie unter Ressourcentypen definiert durch AWS Lambda in der Service Authorization Reference. Informationen zu den Aktionen, mit denen Sie den ARN einzelner Ressourcen angeben können, finden Sie unter Von AWS Lambda definierte Aktionen.

Beispiele für identitätsbasierte Lambda-Richtlinien finden Sie unter Beispiele für identitätsbasierte Richtlinien für AWS Lambda.

Richtlinien-Bedingungsschlüssel für Lambda

Unterstützt servicespezifische Richtlinienbedingungsschlüssel: Ja

Administratoren können mit AWS-JSON-Richtlinien festlegen, welche Personen zum Zugriff auf welche Ressourcen berechtigt sind. Das heißt, welcher Prinzipal Aktionen für welche Ressourcen und unter welchen Bedingungen ausführen kann.

Das `Condition`-Element legt fest, ob Anweisungen auf der Grundlage definierter Kriterien ausgeführt werden. Sie können bedingte Ausdrücke erstellen, die Bedingungsoperatoren verwenden, z. B. ist gleich oder kleiner als, damit die Bedingung in der Richtlinie mit Werten in der Anforderung

übereinstimmt. Eine Liste aller globalen AWS-Bedingungsschlüssel finden Sie unter Globale AWS-Bedingungskontextschlüssel im IAM-Benutzerhandbuch.

Eine Liste von Lambda-Bedingungsschlüsseln finden Sie unter Bedingungsschlüssel für AWS Lambda in der Service-Autorisierungs-Referenz. Informationen dazu, mit welchen Aktionen und Ressourcen Sie einen Bedingungsschlüssel verwenden können, finden Sie unter Von AWS Lambda definierte Aktionen.

Beispiele für identitätsbasierte Lambda-Richtlinien finden Sie unter Beispiele für identitätsbasierte Richtlinien für AWS Lambda.

ACLs in Lambda

Unterstützt ACLs: Nein

Zugriffssteuerungslisten (ACLs) steuern, welche Prinzipale (Kontomitglieder, Benutzer oder Rollen) auf eine Ressource zugreifen können. ACLs sind ähnlich wie ressourcenbasierte Richtlinien, verwenden jedoch nicht das JSON-Richtliniendokumentformat.

ABAC mit Lambda

Unterstützt ABAC (Tags in Richtlinien): Teilweise

Die attributbasierte Zugriffskontrolle (ABAC) ist eine Autorisierungsstrategie, bei der Berechtigungen basierend auf Attributen, auch als Tags bezeichnet, definiert werden. Sie können Tags IAM-Entitäten und AWS-Ressourcen anfügen. Anschließend entwerfen Sie ABAC-Richtlinien, um Operationen zuzulassen, sofern das Tag des Prinzipals mit dem Tag der Ressource übereinstimmt.

Um den Zugriff auf der Grundlage von Tags zu steuern, geben Sie im Bedingungselement einer Richtlinie Tag-Informationen an, indem Sie die Schlüssel `aws:ResourceTag/`*`key-name`*, `aws:RequestTag/`*`key-name`*, oder Bedingung `aws:TagKeys` verwenden.

Wenn ein Service alle drei Bedingungsschlüssel für jeden Ressourcentyp unterstützt, lautet der Wert für den Service Ja. Wenn ein Service alle drei Bedingungsschlüssel für nur einige Ressourcentypen unterstützt, lautet der Wert Teilweise.

Weitere Informationen zu ABAC finden Sie unter Definieren von Berechtigungen mit ABAC-Autorisierung im IAM-Benutzerhandbuch. Um ein Tutorial mit Schritten zur Einstellung von ABAC anzuzeigen, siehe Attributbasierte Zugriffskontrolle (ABAC) verwenden im IAM-Benutzerhandbuch.

Weitere Informationen über das Markieren von Lambda-Ressourcen mit Tags finden Sie unter Verwendung der attributbasierten Zugriffskontrolle in Lambda.

Verwenden temporärer Anmeldeinformationen mit Lambda

Unterstützt temporäre Anmeldeinformationen: Ja

Temporäre Anmeldeinformationen ermöglichen den kurzfristigen Zugriff auf AWS-Ressourcen und werden automatisch erstellt, wenn Sie einen Verbund verwenden oder die Rollen wechseln. AWS empfiehlt, temporäre Anmeldeinformationen anstelle langfristiger Zugriffsschlüssel zu verwenden. Weitere Informationen finden Sie unter Temporary security credentials in IAM und AWS-Services that work with IAM im IAM-Benutzerhandbuch.

Forward Access Sessions für Lambda

Unterstützt Forward Access Sessions (FAS): Nein

Forward access sessions (FAS) verwendet die Berechtigungen des Prinzipals, der einen AWS-Service aufruft, in Kombination mit der Anforderung an den AWS-Service, Anforderungen an nachgelagerte Services zu stellen. Einzelheiten zu den Richtlinien für FAS-Anfragen finden Sie unter Zugriffssitzungen weiterleiten.

Servicerollen für Lambda

Unterstützt Servicerollen: Ja

Eine Servicerolle ist eine IAM-Rolle, die ein Service annimmt, um Aktionen in Ihrem Namen auszuführen. Ein IAM-Administrator kann eine Servicerolle innerhalb von IAM erstellen, ändern und löschen. Weitere Informationen finden Sie unter Erstellen einer Rolle zum Delegieren von Berechtigungen an einen AWS-Service im IAM-Benutzerhandbuch.

In Lambda wird eine Servicerolle als Ausführungsrolle bezeichnet.

> **Warning**
>
> Das Ändern der Berechtigungen für eine Ausführungsrolle kann die Lambda-Funktionalität beeinträchtigen.

Serviceverknüpfte Rollen für Lambda

Unterstützt serviceverknüpfte Rollen: Ja

Eine serviceverknüpfte Rolle ist eine Art von Servicerolle, die mit einem verknüpft ist AWS-Service. Der Service kann die Rolle übernehmen, um eine Aktion in Ihrem Namen auszuführen. Serviceverknüpfte Rollen werden in Ihrem AWS-Konto angezeigt und gehören zum Service. Ein IAM-Administrator kann die Berechtigungen für Service-verknüpfte Rollen anzeigen, aber nicht bearbeiten.

Lambda verfügt im Gegensatz zu Lambda@Edge nicht über serviceverknüpfte Rollen. Weitere Informationen finden Sie unter Serviceverknüpfte Rollen für Lambda@Edge im Amazon-CloudFront-Entwicklerhandbuch.

Details zum Erstellen oder Verwalten von serviceverknüpften Rollen finden Sie unter AWS-Services, die mit IAM funktionieren. Suchen Sie in der Tabelle nach einem Service mit einem `Yes` in der Spalte Service-linked role (Serviceverknüpfte Rolle). Wählen Sie den Link Yes (Ja) aus, um die Dokumentation für die serviceverknüpfte Rolle für diesen Service anzuzeigen.

Beispiele für identitätsbasierte Richtlinien für AWS Lambda

Standardmäßig verfügen Benutzer und Rollen nicht über die Berechtigung, Lambda-Ressourcen zu erstellen oder zu ändern. Ein IAM-Administrator muss IAM-Richtlinien erstellen, die Benutzern die Berechtigung erteilen, Aktionen für die Ressourcen auszuführen, die sie benötigen.

Informationen dazu, wie Sie unter Verwendung dieser beispielhaften JSON-Richtliniendokumente eine identitätsbasierte IAM-Richtlinie erstellen, finden Sie unter Erstellen von IAM-Richtlinien (Konsole) im IAM-Benutzerhandbuch.

Einzelheiten zu den von Lambda definierten Aktionen und Ressourcentypen, einschließlich des Formats der ARNs für die einzelnen Ressourcentypen, finden Sie unter Aktionen, Ressourcen und Bedingungsschlüssel für AWS Lambda in der Service-Autorisierungs-Referenz.

Themen

- Bewährte Methoden für Richtlinien
- Verwenden von Lambda-Konsole
- Gewähren der Berechtigung zur Anzeige der eigenen Berechtigungen für Benutzer

Bewährte Methoden für Richtlinien

Identitätsbasierte Richtlinien legen fest, ob jemand Lambda-Ressourcen in Ihrem Konto erstellen, aufrufen oder löschen kann. Dies kann zusätzliche Kosten für Ihr verursachen AWS-Konto. Befolgen

Sie beim Erstellen oder Bearbeiten identitätsbasierter Richtlinien die folgenden Anleitungen und Empfehlungen:

- Erste Schritte mit AWS-verwaltete Richtlinien und Umstellung auf Berechtigungen mit den geringsten Berechtigungen – Um Ihren Benutzern und Workloads Berechtigungen zu gewähren, verwenden Sie die AWS-verwaltete Richtlinien die Berechtigungen für viele allgemeine Anwendungsfälle gewähren. Sie sind in Ihrem verfügbar AWS-Konto. Wir empfehlen Ihnen, die Berechtigungen weiter zu reduzieren, indem Sie vom Kunden verwaltete AWS-Richtlinien definieren, die speziell auf Ihre Anwendungsfälle zugeschnitten sind. Weitere Informationen finden Sie unter AWS-verwaltete Richtlinien oder AWS-verwaltete Richtlinien für Auftrags-Funktionen im IAM-Benutzerhandbuch.
- Anwendung von Berechtigungen mit den geringsten Rechten – Wenn Sie mit IAM-Richtlinien Berechtigungen festlegen, gewähren Sie nur die Berechtigungen, die für die Durchführung einer Aufgabe erforderlich sind. Sie tun dies, indem Sie die Aktionen definieren, die für bestimmte Ressourcen unter bestimmten Bedingungen durchgeführt werden können, auch bekannt als die geringsten Berechtigungen. Weitere Informationen zur Verwendung von IAM zum Anwenden von Berechtigungen finden Sie unter Richtlinien und Berechtigungen in IAM im IAM-Benutzerhandbuch.
- Verwenden von Bedingungen in IAM-Richtlinien zur weiteren Einschränkung des Zugriffs – Sie können Ihren Richtlinien eine Bedingung hinzufügen, um den Zugriff auf Aktionen und Ressourcen zu beschränken. Sie können beispielsweise eine Richtlinienbedingung schreiben, um festzulegen, dass alle Anforderungen mithilfe von SSL gesendet werden müssen. Sie können auch Bedingungen verwenden, um Zugriff auf Service-Aktionen zu gewähren, wenn diese durch ein bestimmtes AWS-Service, wie beispielsweise CloudFormation, verwendet werden. Weitere Informationen finden Sie unter IAM-JSON-Richtlinienelemente: Bedingung im IAM-Benutzerhandbuch.
- Verwenden von IAM Access Analyzer zur Validierung Ihrer IAM-Richtlinien, um sichere und funktionale Berechtigungen zu gewährleisten – IAM Access Analyzer validiert neue und vorhandene Richtlinien, damit die Richtlinien der IAM-Richtliniensprache (JSON) und den bewährten IAM-Methoden entsprechen. IAM Access Analyzer stellt mehr als 100 Richtlinienprüfungen und umsetzbare Empfehlungen zur Verfügung, damit Sie sichere und funktionale Richtlinien erstellen können. Weitere Informationen finden Sie unter Richtlinienvalidierung mit IAM Access Analyzer im IAM-Benutzerhandbuch.
- Bedarf einer Multi-Faktor-Authentifizierung (MFA) – Wenn Sie ein Szenario haben, das IAM-Benutzer oder Root-Benutzer in Ihrem AWS-Konto erfordert, aktivieren Sie MFA für zusätzliche Sicherheit. Um MFA beim Aufrufen von API-Vorgängen anzufordern, fügen Sie Ihren Richtlinien

MFA-Bedingungen hinzu. Weitere Informationen finden Sie unter Sicherer API-Zugriff mit MFA im IAM-Benutzerhandbuch.

Weitere Informationen zu bewährten Methoden in IAM finden Sie unter Bewährte Methoden für die Sicherheit in IAM im IAM-Benutzerhandbuch.

Verwenden von Lambda-Konsole

Um auf die AWS Lambda-Konsole zuzugreifen, müssen Sie über einen Mindestsatz von Berechtigungen verfügen. Diese Berechtigungen müssen Ihnen das Auflisten und Anzeigen von Details zu den Lambda-Ressourcen in Ihrem AWS-Konto gestatten. Wenn Sie eine identitätsbasierte Richtlinie erstellen, die strenger ist als die mindestens erforderlichen Berechtigungen, funktioniert die Konsole nicht wie vorgesehen für Entitäten (Benutzer oder Rollen) mit dieser Richtlinie.

Für Benutzer, die nur Aufrufe an die AWS CLI oder AWS-API durchführen, müssen Sie keine Mindestberechtigungen in der Konsole erteilen. Stattdessen sollten Sie nur Zugriff auf die Aktionen zulassen, die der API-Operation entsprechen, die die Benutzer ausführen möchten.

Eine Beispielrichtlinie, die minimalen Zugriff für die Funktionsentwicklung gewährt, finden Sie unter Benutzern den Zugriff auf eine Lambda-Funktion gewähren. Zusätzlich zu den Lambda-APIs verwendet die Lambda-Konsole weitere Services, um Auslöserkonfigurationen anzuzeigen und neue Auslöser hinzuzufügen. Wenn Ihre Benutzer Lambda mit anderen Services nutzen, benötigen sie auch Zugriff auf diese Services. Details zum Konfigurieren weiterer Services mit Lambda finden Sie unter Lambda mit Ereignissen aus anderen Diensten aufrufen AWS.

Gewähren der Berechtigung zur Anzeige der eigenen Berechtigungen für Benutzer

In diesem Beispiel wird gezeigt, wie Sie eine Richtlinie erstellen, die IAM-Benutzern die Berechtigung zum Anzeigen der eingebundenen Richtlinien und verwalteten Richtlinien gewährt, die ihrer Benutzeridentität angefügt sind. Diese Richtlinie enthält Berechtigungen für die Ausführung dieser Aktion auf der Konsole oder für die programmgesteuerte Ausführung über die AWS CLI oder die AWS-API.

```
{
    "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 verwaltete Richtlinien für AWS Lambda

Eine AWS verwaltete Richtlinie ist eine eigenständige Richtlinie, die von erstellt und verwaltet AWS wird. AWS Verwaltete Richtlinien sind so konzipiert, dass sie Berechtigungen für viele gängige Anwendungsfälle bereitstellen, sodass Sie damit beginnen können, Benutzern, Gruppen und Rollen Berechtigungen zuzuweisen.

Beachten Sie, dass AWS verwaltete Richtlinien für Ihre speziellen Anwendungsfälle möglicherweise keine Berechtigungen mit den geringsten Rechten gewähren, da sie allen AWS Kunden zur Verfügung stehen. Wir empfehlen Ihnen, die Berechtigungen weiter zu reduzieren, indem Sie vom Kunden verwaltete Richtlinien definieren, die speziell auf Ihre Anwendungsfälle zugeschnitten sind.

Sie können die in AWS verwalteten Richtlinien definierten Berechtigungen nicht ändern. Wenn die in einer AWS verwalteten Richtlinie definierten Berechtigungen AWS aktualisiert werden, wirkt sich das

Update auf alle Prinzidentitäten (Benutzer, Gruppen und Rollen) aus, denen die Richtlinie zugeordnet ist. AWS aktualisiert eine AWS verwaltete Richtlinie höchstwahrscheinlich, wenn eine neue Richtlinie eingeführt AWS-Service wird oder neue API-Operationen für bestehende Dienste verfügbar werden.

Weitere Informationen finden Sie unter Von AWS verwaltete Richtlinien im IAM-Benutzerhandbuch.

Themen

- AWS verwaltete Richtlinie: AWSLambda_FullAccess
- AWS verwaltete Richtlinie: AWSLambda_ReadOnlyAccess
- AWS verwaltete Richtlinie: AWSLambda BasicExecutionRole
- AWS verwaltete Richtlinie: AWSLambda BasicDurableExecutionRolePolicy
- AWS verwaltete Richtlinie: Dynamo-Rolle AWSLambda DBExecution
- AWS verwaltete Richtlinie: Zugriff AWSLambda ENIManagement
- AWS verwaltete Richtlinie: AWSLambdaInvocation-DynamoDB
- AWS verwaltete Richtlinie: AWSLambda KinesisExecutionRole
- AWS verwaltete Richtlinie: AWSLambda MSKExecution Rolle
- AWS verwaltete Richtlinie: AWSLambda Rolle
- AWS verwaltete Richtlinie: AWSLambda SQSQueue ExecutionRole
- AWS verwaltete Richtlinie: AWSLambda VPCAccess ExecutionRole
- AWS verwaltete Richtlinie: AWSLambda Verwaltet EC2 ResourceOperator
- AWS verwaltete Richtlinie: AWSLambda ServiceRolePolicy
- Lambda-Updates für AWS verwaltete Richtlinien

AWS verwaltete Richtlinie: AWSLambda_FullAccess

Diese Richtlinie ermöglicht Vollzugriff auf alle Lambda-Aktionen. Es gewährt auch Berechtigungen für andere AWS Dienste, die zur Entwicklung und Wartung von Lambda-Ressourcen verwendet werden.

Die Richtlinie `AWSLambda_FullAccess` kann an Benutzer, Gruppen und Rollen angefügt werden.

Details zu Berechtigungen

Diese Richtlinie umfasst die folgenden Berechtigungen:

- `lambda`: Ermöglicht Prinzipalen Vollzugriff auf Lambda.

- `cloudformation`— Ermöglicht Prinzipalen, AWS CloudFormation Stacks zu beschreiben und die Ressourcen in diesen Stacks aufzulisten.
- `cloudwatch`— Ermöglicht Prinzipalen, CloudWatch Amazon-Metriken aufzulisten und Metrikdaten abzurufen.
- `ec2`— Ermöglicht Prinzipalen die Beschreibung von Sicherheitsgruppen, Subnetzen und. VPCs
- `iam`: Ermöglicht es Prinzipalen, Richtlinien, Richtlinienversionen, Rollen, Rollenrichtlinien, angefügte Rollenrichtlinien und die Liste der Rollen abzurufen. Diese Richtlinie ermöglicht es Prinzipalen außerdem, Rollen an Lambda zu übergeben. Die Berechtigung `PassRole` wird verwendet, wenn Sie einer Funktion eine Ausführungsrolle zuweisen. Die `CreateServiceLinkedRole` Berechtigung wird verwendet, wenn eine dienstbezogene Rolle erstellt wird.
- `kms`— Ermöglicht Prinzipalen, Aliase aufzulisten und den Schlüssel für die Volumenverschlüsselung zu beschreiben.
- `logs`: Ermöglicht Prinzipalen das Beschreiben von Protokollstreams, das Abrufen und Filtern von Protokollereignissen sowie das Starten und Beenden von Live-Tail-Sitzungen.
- `states`— Ermöglicht es Prinzipalen, Zustandsmaschinen zu beschreiben und aufzulisten AWS Step Functions .
- `tag`: Ermöglicht es Prinzipalen, Ressourcen auf der Grundlage ihrer Tags abzurufen.
- `xray`— Ermöglicht Prinzipalen das Abrufen von AWS X-Ray Trace-Zusammenfassungen und das Abrufen einer nach ID spezifizierten Liste von Traces.

Weitere Informationen zu dieser Richtlinie, einschließlich des JSON-Richtliniendokuments und der Richtlinienversionen, finden Sie AWSLambda_FullAccessim Referenzhandbuch für AWS verwaltete Richtlinien.

AWS verwaltete Richtlinie: AWSLambda_ReadOnlyAccess

Diese Richtlinie gewährt schreibgeschützten Zugriff auf Lambda-Ressourcen und andere AWS Dienste, die zur Entwicklung und Wartung von Lambda-Ressourcen verwendet werden.

Die Richtlinie `AWSLambda_ReadOnlyAccess` kann an Benutzer, Gruppen und Rollen angefügt werden.

Details zu Berechtigungen

Diese Richtlinie umfasst die folgenden Berechtigungen:

- `lambda`: Ermöglicht es Prinzipalen, alle Ressourcen abzurufen und aufzulisten.
- `cloudformation`— Ermöglicht Prinzipalen, Stacks zu beschreiben und aufzulisten und die Ressourcen in AWS CloudFormation diesen Stacks aufzulisten.
- `cloudwatch`— Ermöglicht Prinzipalen, CloudWatch Amazon-Metriken aufzulisten und Metrikdaten abzurufen.
- `ec2`— Ermöglicht Prinzipalen die Beschreibung von Sicherheitsgruppen, Subnetzen und. VPCs
- `iam`: Ermöglicht es Prinzipalen, Richtlinien, Richtlinienversionen, Rollen, Rollenrichtlinien, angefügte Rollenrichtlinien und die Liste der Rollen abzurufen.
- `kms`: Ermöglicht Prinzipalen das Auflisten von Aliasen.
- `logs`: Ermöglicht Prinzipalen das Beschreiben von Protokollstreams, das Abrufen und Filtern von Protokollereignissen sowie das Starten und Beenden von Live-Tail-Sitzungen.
- `states`— Ermöglicht es Prinzipalen, Zustandsmaschinen zu beschreiben und aufzulisten AWS Step Functions .
- `tag`: Ermöglicht es Prinzipalen, Ressourcen auf der Grundlage ihrer Tags abzurufen.
- `xray`— Ermöglicht Prinzipalen das Abrufen von AWS X-Ray Trace-Zusammenfassungen und das Abrufen einer nach ID spezifizierten Liste von Traces.

Weitere Informationen zu dieser Richtlinie, einschließlich des JSON-Richtliniendokuments und der Richtlinienversionen, finden Sie AWSLambda_ReadOnlyAccessim Referenzhandbuch für AWS verwaltete Richtlinien.

AWS verwaltete Richtlinie: AWSLambda BasicExecutionRole

Diese Richtlinie gewährt Berechtigungen zum Hochladen von Protokollen in CloudWatch Logs.

Die Richtlinie `AWSLambdaBasicExecutionRole` kann an Benutzer, Gruppen und Rollen angefügt werden.

Weitere Informationen zu dieser Richtlinie, einschließlich des JSON-Richtliniendokuments und der Richtlinienversionen, finden Sie AWSLambdaBasicExecutionRoleim Referenzhandbuch für AWS verwaltete Richtlinien.

AWS verwaltete Richtlinie: AWSLambda BasicDurableExecutionRolePolicy

Diese Richtlinie gewährt Schreibberechtigungen für CloudWatch Protokolle und read/write Berechtigungen für die dauerhafte Ausführung, die von dauerhaften Lambda-Funktionen APIs

verwendet werden. Diese Richtlinie stellt die grundlegenden Berechtigungen bereit, die für dauerhafte Lambda-Funktionen erforderlich sind, die eine dauerhafte Ausführung verwenden, APIs um den Fortschritt aufrechtzuerhalten und den Status bei Funktionsaufrufen aufrechtzuerhalten.

Die Richtlinie `AWSLambdaBasicDurableExecutionRolePolicy` kann an Benutzer, Gruppen und Rollen angefügt werden.

Details zu Berechtigungen

Diese Richtlinie umfasst die folgenden Berechtigungen:

- `logs`— Ermöglicht es Prinzipalen, Protokollgruppen und Protokollstreams zu erstellen und Protokollereignisse in Logs zu schreiben. CloudWatch
- `lambda`— Ermöglicht es Prinzipalen, den dauerhaften Ausführungsstatus zu überprüfen und den dauerhaften Ausführungsstatus für langlebige Lambda-Funktionen abzurufen.

Weitere Einzelheiten zu dieser Richtlinie, einschließlich der neuesten Version des JSON-Richtliniendokuments, finden Sie unter AWSLambdaBasicDurableExecutionRolePolicy im AWS Referenzhandbuch für verwaltete Richtlinien.

AWS verwaltete Richtlinie: Dynamo-Rolle AWSLambda DBExecution

Diese Richtlinie gewährt Berechtigungen zum Lesen von Datensätzen aus einem Amazon DynamoDB DynamoDB-Stream und zum Schreiben in CloudWatch Protokolle.

Die Richtlinie `AWSLambdaDynamoDBExecutionRole` kann an Benutzer, Gruppen und Rollen angefügt werden.

Weitere Informationen zu dieser Richtlinie, einschließlich des JSON-Richtliniendokuments und der Richtlinienversionen, finden Sie unter AWSLambdaDynamo DBExecution Role im Referenzhandbuch für AWS verwaltete Richtlinien.

AWS verwaltete Richtlinie: Zugriff AWSLambda ENIManagement

Diese Richtlinie erteilt Berechtigungen zum Erstellen, Beschreiben und Löschen von Elastic Network-Schnittstellen, die von einer VPC-fähigen Lambda-Funktion verwendet werden.

Die Richtlinie `AWSLambdaENIManagementAccess` kann an Benutzer, Gruppen und Rollen angefügt werden.

Weitere Informationen zu dieser Richtlinie, einschließlich des JSON-Richtliniendokuments und der Richtlinienversionen, finden Sie unter AWSLambdaENIManagementAccess im Referenzhandbuch für AWS verwaltete Richtlinien.

AWS verwaltete Richtlinie: AWSLambdaInvocation-DynamoDB

Diese Richtlinie gewährt Lesezugriff auf Amazon DynamoDB Streams.

Die Richtlinie `AWSLambdaInvocation-DynamoDB` kann an Benutzer, Gruppen und Rollen angefügt werden.

Weitere Informationen zu dieser Richtlinie, einschließlich des JSON-Richtliniendokuments und der Richtlinienversionen, finden Sie AWSLambdaInvocation-DynamoDBim Referenzhandbuch für AWS verwaltete Richtlinien.

AWS verwaltete Richtlinie: AWSLambda KinesisExecutionRole

Diese Richtlinie gewährt Berechtigungen zum Lesen von Ereignissen aus einem Amazon Kinesis Kinesis-Datenstream und zum Schreiben in CloudWatch Protokolle.

Die Richtlinie `AWSLambdaKinesisExecutionRole` kann an Benutzer, Gruppen und Rollen angefügt werden.

Weitere Informationen zu dieser Richtlinie, einschließlich des JSON-Richtliniendokuments und der Richtlinienversionen, finden Sie AWSLambdaKinesisExecutionRoleim Referenzhandbuch für AWS verwaltete Richtlinien.

AWS verwaltete Richtlinie: AWSLambda MSKExecution Rolle

Diese Richtlinie gewährt Berechtigungen zum Lesen und Zugreifen auf Datensätze aus einem Amazon Managed Streaming for Apache Kafka Kafka-Cluster, zur Verwaltung elastischer Netzwerkschnittstellen und zum Schreiben in CloudWatch Protokolle.

Die Richtlinie `AWSLambdaMSKExecutionRole` kann an Benutzer, Gruppen und Rollen angefügt werden.

Weitere Informationen zu dieser Richtlinie, einschließlich des JSON-Richtliniendokuments und der Richtlinienversionen, finden Sie unter AWSLambdaMSKExecutionRolle im Referenzhandbuch für AWS verwaltete Richtlinien.

AWS verwaltete Richtlinie: AWSLambda Rolle

Diese Richtlinie erteilt Berechtigungen zum Aufrufen von Lambda-Funktionen.

Die Richtlinie `AWSLambdaRole` kann an Benutzer, Gruppen und Rollen angefügt werden.

Weitere Informationen zu dieser Richtlinie, einschließlich des JSON-Richtliniendokuments und der Richtlinienversionen, finden Sie unter AWSLambdaRolle im Referenzhandbuch für AWS verwaltete Richtlinien.

AWS verwaltete Richtlinie: AWSLambda SQSQueue ExecutionRole

Diese Richtlinie gewährt Berechtigungen zum Lesen und Löschen von Nachrichten aus einer Amazon Simple Queue Service-Warteschlange sowie Schreibberechtigungen für CloudWatch Logs.

Die Richtlinie `AWSLambdaSQSQueueExecutionRole` kann an Benutzer, Gruppen und Rollen angefügt werden.

Weitere Informationen zu dieser Richtlinie, einschließlich des JSON-Richtliniendokuments und der Richtlinienversionen, finden Sie AWSLambdaSQSQueueExecutionRoleim Referenzhandbuch für AWS verwaltete Richtlinien.

AWS verwaltete Richtlinie: AWSLambda VPCAccess ExecutionRole

Diese Richtlinie gewährt Berechtigungen zur Verwaltung elastischer Netzwerkschnittstellen innerhalb einer Amazon Virtual Private Cloud und zum Schreiben in CloudWatch Logs.

Die Richtlinie `AWSLambdaVPCAccessExecutionRole` kann an Benutzer, Gruppen und Rollen angefügt werden.

Weitere Informationen zu dieser Richtlinie, einschließlich des JSON-Richtliniendokuments und der Richtlinienversionen, finden Sie AWSLambdaVPCAccessExecutionRoleim Referenzhandbuch für AWS verwaltete Richtlinien.

AWS verwaltete Richtlinie: AWSLambda Verwaltet EC2 ResourceOperator

Diese Richtlinie ermöglicht automatisiertes Amazon Elastic Compute Cloud-Instanzmanagement für Lambda-Kapazitätsanbieter. Es gewährt dem Lambda Scaler Service Berechtigungen, um Instanzlebenszyklusoperationen in Ihrem Namen durchzuführen.

Die Richtlinie `AWSLambdaManagedEC2ResourceOperator` kann an Benutzer, Gruppen und Rollen angefügt werden.

Details zu Berechtigungen

Diese Richtlinie umfasst die folgenden Berechtigungen:

- `ec2:RunInstances`— Ermöglicht Lambda den Start neuer EC2 Amazon-Instances unter der Bedingung, dass ec2: ManagedResourceOperator scaler.lambda.amazonaws.com entspricht und beschränkt die AMI-Nutzung nur auf Amazon-eigene Images.
- `ec2:DescribeInstances`und `ec2:DescribeInstanceStatus` — Ermöglicht Lambda, den Instanzstatus zu überwachen und Instanzinformationen abzurufen.
- `ec2:CreateTags`— Ermöglicht Lambda, EC2 Amazon-Ressourcen zu Verwaltungs- und Identifikationszwecken zu kennzeichnen.
- `ec2:DescribeAvailabilityZones`— Ermöglicht Lambda, verfügbare Zonen einzusehen, um Entscheidungen zur Instanzplatzierung zu treffen.
- `ec2:DescribeCapacityReservations`— Ermöglicht Lambda, Kapazitätsreservierungen für eine optimale Instance-Platzierung zu überprüfen.
- `ec2:DescribeInstanceTypes`und `ec2:DescribeInstanceTypeOfferings` — Ermöglicht Lambda, verfügbare Instance-Typen und deren Angebote zu überprüfen.
- `ec2:DescribeSubnets`— Ermöglicht Lambda, Subnetzkonfigurationen für die Netzwerkplanung zu untersuchen.
- `ec2:DescribeSecurityGroups`— Ermöglicht Lambda das Abrufen von Sicherheitsgruppeninformationen für die Konfiguration der Netzwerkschnittstelle.
- `ec2:CreateNetworkInterface`— Ermöglicht Lambda die Erstellung von Netzwerkschnittstellen und die Verwaltung von Subnetz- und Sicherheitsgruppenzuordnungen.
- `ec2:AttachNetworkInterface`— Ermöglicht Lambda das Anhängen von Netzwerkschnittstellen an EC2 Amazon-Instances mit der Bedingung `ec2:ManagedResourceOperator` scaler.lambda.amazonaws.com.

Weitere Informationen zu dieser Richtlinie, einschließlich des JSON-Richtliniendokuments und der Richtlinienversionen, finden Sie unter Verwaltet im Referenzhandbuch für verwaltete Richtlinien. AWSLambda EC2 ResourceOperator AWS

AWS verwaltete Richtlinie: AWSLambda ServiceRolePolicy

Diese Richtlinie ist der serviceverknüpften Rolle zugeordnet, die so benannt ist AWSServiceRoleForLambda , dass Lambda Instances beenden kann, die als Teil von Lambda-Kapazitätsanbietern verwaltet werden.

Details zu Berechtigungen

Diese Richtlinie umfasst die folgenden Berechtigungen:

- `ec2:TerminateInstances`— Ermöglicht Lambda, EC2 Instances unter der Bedingung zu beenden, dass ec2: ManagedResourceOperator gleich scaler.lambda.amazonaws.com ist.
- `ec2:DescribeInstanceStatus`und `ec2:DescribeInstances` — Ermöglicht Lambda, EC2 Instanzen zu beschreiben.

Weitere Informationen zu dieser Richtlinie finden Sie unter Verwenden von dienstverknüpften Rollen für Lambda.

Lambda-Updates für AWS verwaltete Richtlinien

Änderungen	Beschreibung	Date
AWSLambdaVerwaltet EC2 ResourceOperator — Neue Richtlinie	Lambda hat eine neue verwaltete Richtlinie hinzugefügt, um ein automatisiertes EC2 Amazon-Instance-Management für Lambda-Kapazitätsanbieter zu ermöglichen, sodass der Scaler-Service Instance-Lebenszyklusoperationen durchführen kann.	30. November 2025
AWSLambdaServiceRolePolicy – Neue Richtlinie	Lambda hat eine neue verwaltete Richtlinie für die serviceverknüpfte Rolle hinzugefügt, damit Lambda Instanzen beenden kann, die als Teil von Lambda-Kapazitätsanbietern verwaltet werden.	30. November 2025
AWSLambda_FullAccess— Veränderung	Lambda hat die `AWSLambda_FullAccess` Richtlinie aktualisiert, um die	30. November 2025

Änderungen	Beschreibung	Date
	`iam:CreateServiceLinkedRole` Aktionen `kms:DescribeKey` und zuzulassen.	
AWSLambdaBasicDurableExecutionRolePolicy— Neue verwaltete Richtlinie	Lambda hat eine neue verwaltete Richtlinie veröffentlicht`AWSLambdaBasicDurableExecutionRolePolicy` , die Schreibberechtigungen für CloudWatch Protokolle und read/write Berechtigungen für die dauerhafte Ausführung bereitstellt, die von dauerhaften Lambda-Funktionen APIs verwendet werden.	1. Dezember 2025
AWSLambda_ReadOnlyAccessund AWSLambda_FullAccess— Veränderung	Lambda hat die Richtlinien `AWSLambda_ReadOnlyAccess` und `AWSLambda_FullAccess` aktualisiert, um die Aktionen `logs:StartLiveTail` und `logs:StopLiveTail` zuzulassen.	17. März 2025
AWSLambdaVPCAccessExecutionRole— Veränderung	Lambda hat die `AWSLambdaVPCAccessExecutionRole` -Richtlinie aktualisiert, um die Aktion `ec2:DescribeSubnets` zuzulassen.	5. Januar 2024

Änderungen	Beschreibung	Date
AWSLambda_ReadOnly Access— Veränderung	Lambda hat die `AWSLambda_ReadOnlyAccess` Richtlinie aktualisiert, sodass Prinzipale Stacks auflisten CloudFormation können.	27. Juli 2023
AWS Lambda hat begonnen, Änderungen zu verfolgen	AWS Lambda hat begonnen, Änderungen für die AWS verwalteten Richtlinien zu verfolgen.	27. Juli 2023

Fehlerbehebung für AWS Lambda-Identität und -Zugriff

Diagnostizieren und beheben Sie mithilfe der folgenden Informationen gängige Probleme, die bei der Verwendung von Lambda und IAM auftreten können.

Themen

- Ich bin nicht autorisiert, eine Aktion in Lambda auszuführen.
- Ich bin nicht zur Ausführung von iam:PassRole autorisiert
- Ich möchte Personen außerhalb meines AWS-Konto Zugriff auf meine Lambda-Ressourcen gewähren

Ich bin nicht autorisiert, eine Aktion in Lambda auszuführen.

Wenn Sie eine Fehlermeldung erhalten, dass Sie nicht zur Durchführung einer Aktion berechtigt sind, müssen Ihre Richtlinien aktualisiert werden, damit Sie die Aktion durchführen können.

Der folgende Beispielfehler tritt auf, wenn der IAM-Benutzer `mateojackson` versucht, über die Konsole Details zu einer fiktiven *`my-example-widget`*-Ressource anzuzeigen, jedoch nicht über `lambda:`*`GetWidget`*-Berechtigungen verfügt.

```
User: arn:aws:iam::123456789012:user/mateojackson is not authorized to perform:
 lambda:GetWidget on resource: my-example-widget
```

In diesem Fall muss die Richtlinie für den Benutzer `mateojackson` aktualisiert werden, damit er mit der `lambda:GetWidget`-Aktion auf die `my-example-widget`-Ressource zugreifen kann.

Wenden Sie sich an Ihren AWS-Administrator, falls Sie weitere Unterstützung benötigen. Ihr Administrator hat Ihnen Ihre Anmeldeinformationen zur Verfügung gestellt.

Ich bin nicht zur Ausführung von iam:PassRole autorisiert

Wenn Sie die Fehlermeldung erhalten, dass Sie nicht zur Ausführung der Aktion „`iam:PassRole`“ autorisiert sind, müssen Ihre Richtlinien aktualisiert werden, um eine Rolle an Lambda übergeben zu können.

Einige AWS-Services erlauben die Übergabe einer vorhandenen Rolle an diesen Dienst, sodass keine neue Servicerolle oder serviceverknüpfte Rolle erstellt werden muss. Hierzu benötigen Sie Berechtigungen für die Übergabe der Rolle an den Dienst.

Der folgende Beispielfehler tritt auf, wenn ein IAM-Benutzer mit dem Namen `marymajor` versucht, die Konsole zu verwenden, um eine Aktion in Lambda auszuführen. Die Aktion erfordert jedoch, dass der Service über Berechtigungen verfügt, die durch eine Servicerolle gewährt werden. Mary besitzt keine Berechtigungen für die Übergabe der Rolle an den Dienst.

```
User: arn:aws:iam::123456789012:user/marymajor is not authorized to perform:
 iam:PassRole
```

In diesem Fall müssen die Richtlinien von Mary aktualisiert werden, um die Aktion `iam:PassRole` ausführen zu können.

Wenden Sie sich an Ihren AWS-Administrator, falls Sie weitere Unterstützung benötigen. Ihr Administrator hat Ihnen Ihre Anmeldeinformationen zur Verfügung gestellt.

Ich möchte Personen außerhalb meines AWS-Konto Zugriff auf meine Lambda-Ressourcen gewähren

Sie können eine Rolle erstellen, die Benutzer in anderen Konten oder Personen außerhalb Ihrer Organisation für den Zugriff auf Ihre Ressourcen verwenden können. Sie können festlegen, wem die Übernahme der Rolle anvertraut wird. Im Fall von Diensten, die ressourcenbasierte Richtlinien oder Zugriffskontrolllisten (Access Control Lists, ACLs) verwenden, können Sie diese Richtlinien verwenden, um Personen Zugriff auf Ihre Ressourcen zu gewähren.

Weitere Informationen finden Sie hier:

- Informationen dazu, ob Lambda diese Funktionen unterstützt, finden Sie unter Funktionsweise von AWS Lambda mit IAM.
- Informationen zum Gewähren des Zugriffs auf Ihre Ressourcen für alle Ihre AWS-Konten finden Sie unter Gewähren des Zugriffs für einen IAM-Benutzer in einem anderen Ihrer AWS-Konto im IAM-Benutzerhandbuch.
- Informationen dazu, wie Sie AWS-Konten-Drittanbieter Zugriff auf Ihre Ressourcen bereitstellen, finden Sie unter Gewähren des Zugriffs auf AWS-Konten von externen Benutzern im IAM-Benutzerhandbuch.
- Informationen dazu, wie Sie über einen Identitätsverbund Zugriff gewähren, finden Sie unter Gewähren von Zugriff für extern authentifizierte Benutzer (Identitätsverbund) im IAM-Benutzerhandbuch.
- Informationen zum Unterschied zwischen der Verwendung von Rollen und ressourcenbasierten Richtlinien für den kontoübergreifenden Zugriff finden Sie unter Kontoübergreifender Ressourcenzugriff in IAM im IAM-Benutzerhandbuch.

Erstellen Sie eine Governance-Strategie für Lambda-Funktionen und -Layer

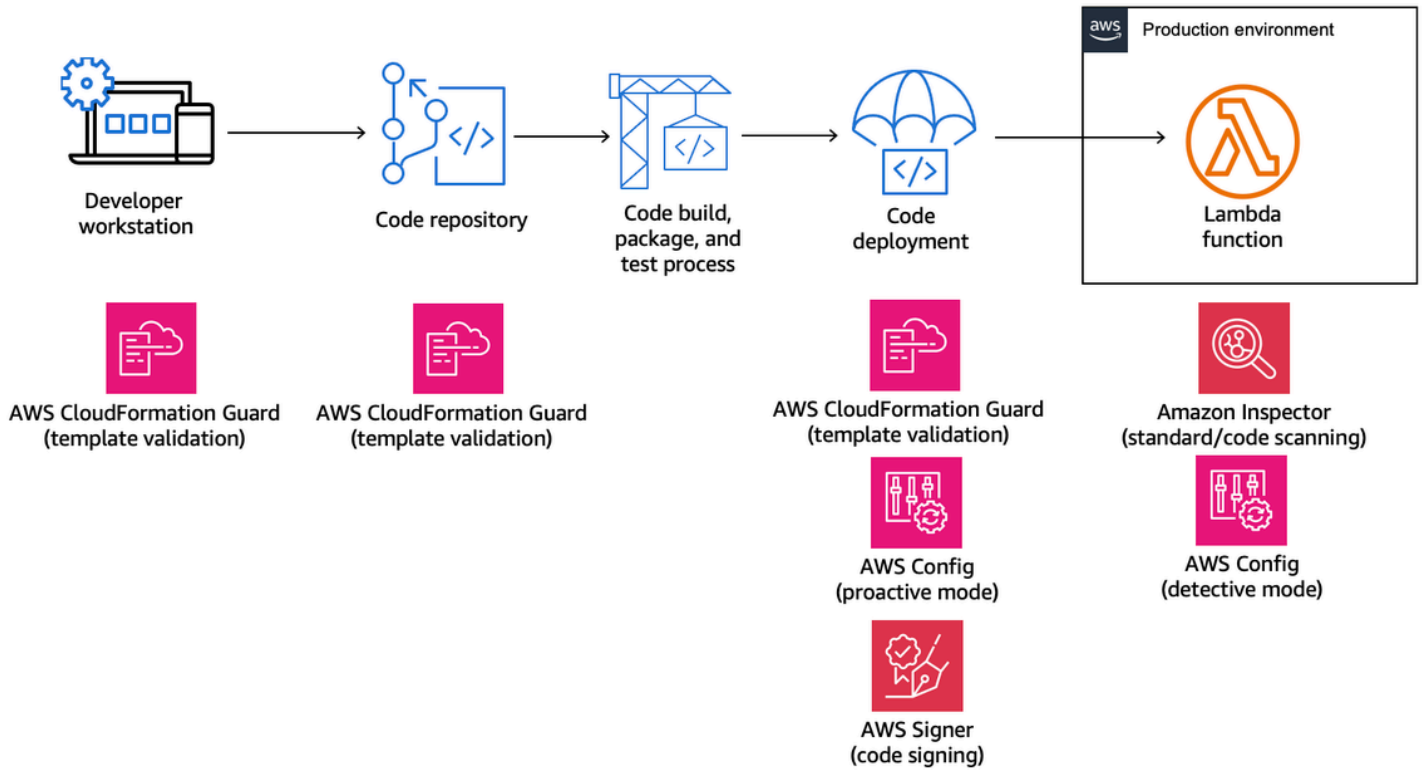
Bei der Entwicklung und Bereitstellung cloudnativer Serverless-Anwendungen müssen Sie durch angemessene Governance- und Schutzmaßnahmen für Agilität und eine schnelle Markteinführung sorgen. Dazu legen Sie auf Unternehmensebene Prioritäten fest, wobei Sie entweder die Agilität oder durch geeignete Governance, Schutzmaßnahmen und Kontrollen die Risikovermeidung an erste Stelle setzen. Normalerweise werden Sie keine so klare Grenze ziehen, sondern beide Aspekte – Agilität und Sicherheit – in Ihrem Softwareentwicklungszyklus abbilden möchten. Ganz gleich, wo im Lebenszyklus Ihres Unternehmens diese Anforderungen stehen, Sie werden mit hoher Wahrscheinlichkeit Governance-Funktionen in Ihre Prozesse und Toolchains implementieren müssen.

Hier sind einige Beispiele für Governance-Kontrollen, die ein Unternehmen für Lambda implementieren könnte:

- Lambda-Funktionen dürfen nicht öffentlich zugänglich sein.
- Lambda-Funktionen müssen mit einer VPC verbunden sein.
- Lambda-Funktionen sollten keine veralteten Laufzeiten verwenden.
- Lambda-Funktionen müssen mit einer Reihe erforderlicher Tags markiert werden.

- Lambda-Layer dürfen außerhalb der Organisation nicht zugänglich sein.
- Wenn Lambda-Funktionen mit einer Sicherheitsgruppe verbunden sind, müssen beide übereinstimmende Tags haben.
- Lambda-Funktionen mit einem verbundenen Layer müssen eine genehmigte Version verwenden
- Lambda-Umgebungsvariablen müssen im Ruhezustand mit einem vom Kunden verwalteten Schlüssel verschlüsselt sein.

Das folgende Diagramm ist ein Beispiel für eine umfassende Governance-Strategie, bei der Kontrollen und Richtlinien während des gesamten Softwareentwicklungs- und Bereitstellungsprozesses implementiert werden:



In den folgenden Themen wird erklärt, wie Sie Kontrollen für die Entwicklung und Bereitstellung von Lambda-Funktionen in Ihrem Unternehmen implementieren. Die Ausführungen gelten für Startups und große Unternehmen gleichermaßen. Möglicherweise verfügt Ihr Unternehmen bereits über geeignete Tools. In den folgenden Themen wird ein modularer Ansatz für diese Kontrollen verfolgt, damit Sie die Komponenten auswählen können, die Sie tatsächlich benötigen.

Themen

- Proaktive Kontrollen für Lambda mit AWS CloudFormation Guard

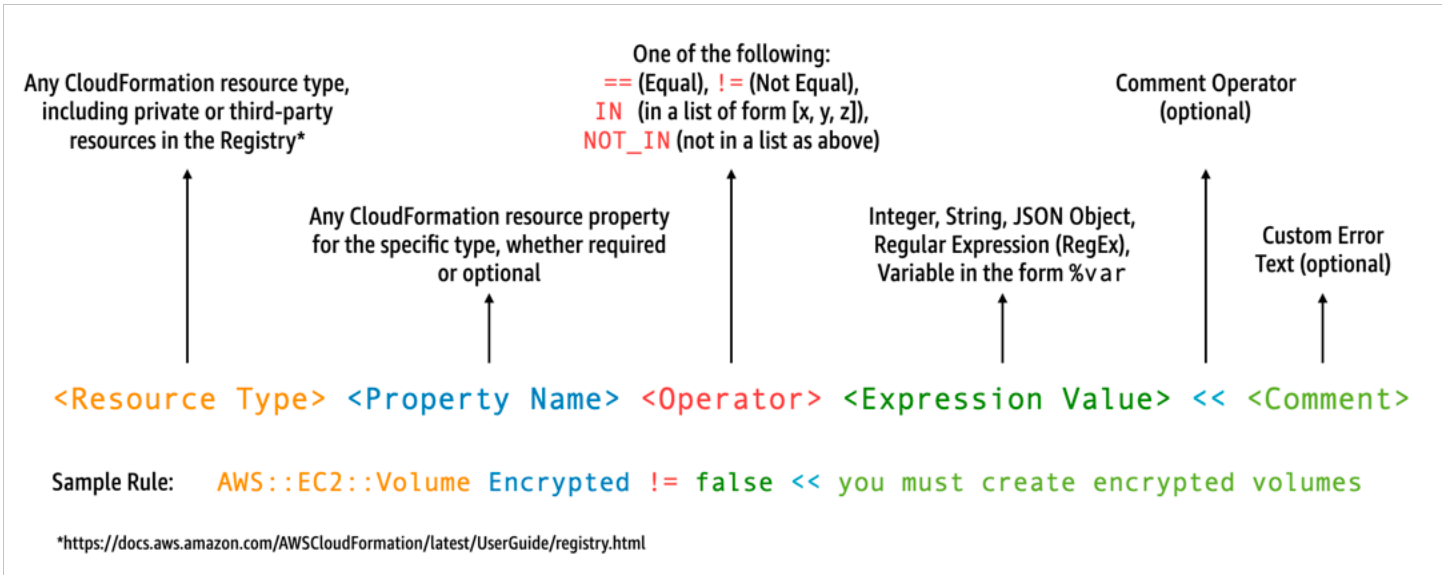
- Implementieren Sie präventive Kontrollen für Lambda mit AWS Config
- Erkennen Sie nicht konforme Lambda-Bereitstellungen und -Konfigurationen mit AWS Config
- Lambda-Codesignatur mit AWS Signer
- Automatisieren Sie Sicherheitsbewertungen für Lambda mit Amazon Inspector
- Implementieren von Beobachtbarkeit für Lambda-Sicherheit und -Compliance

Proaktive Kontrollen für Lambda mit AWS CloudFormation Guard

AWS CloudFormation Guard ist ein allgemein verwendbares Open-Source-Tool zur Policy-as-Code-Evaluierung. Es kann für präventive Governance und Compliance verwendet werden, indem Infrastructure as Code (IaC)-Vorlagen und Servicezusammenstellungen anhand von Richtlinienregeln validiert werden. Diese Regeln können an die Anforderungen Ihres Teams oder Ihrer Organisation angepasst werden. Für Lambda-Funktionen können die Guard-Regeln verwendet werden, um die Ressourcenerstellung und Konfigurationsupdates zu steuern. Dazu werden die erforderlichen Eigenschaftseinstellungen definiert, die beim Erstellen oder Aktualisieren einer Lambda-Funktion erforderlich sind.

Compliance-Administratoren definieren die Liste der Kontrollen und Governance-Richtlinien, die für die Bereitstellung und Aktualisierung von Lambda-Funktionen erforderlich sind. Plattformadministratoren implementieren die Kontrollen in CI/CD-Pipelines als Pre-Commit-Validierungs-Webhooks mit Code-Repositorys und stellen Entwicklern Befehlszeilentools zur Validierung von Vorlagen und Code auf lokalen Workstations zur Verfügung. Entwickler schreiben Code, validieren Vorlagen mit Befehlszeilentools und übertragen dann den Code in Repositorys, die vor der Bereitstellung in einer AWS-Umgebung automatisch über die CI/CD-Pipelines validiert werden.

Guard ermöglicht es Ihnen wie folgt, Regeln zu schreiben und Kontrollen in einer domainspezifischen Sprache zu implementieren.



Angenommen, Sie möchten, dass Entwickler nur die neuesten Laufzeiten verwenden. Sie könnten zwei verschiedene Richtlinien erstellen. Eine, um Laufzeiten zu bestimmen, die bereits veraltet sind,

und eine andere, um Laufzeiten zu bestimmen, die demnächst veraltet sein werden. Dazu könnten Sie die folgende `etc/rules.guard`-Datei schreiben:

```
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.>>
            }
        }
    }
}
```

Nehmen wir nun an, Sie schreiben die folgende `iac/lambda.yaml`-CloudFormation-Vorlage, die eine Lambda-Funktion definiert:

```
  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
```

Nach der Installation des Guard-Dienstprogramms überprüfen Sie die Vorlage:

```
cfn-guard validate --rules etc/rules.guard --data iac/lambda.yaml
```

Das Ergebnis sieht folgendermaßen aus:

```
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 zeigt Entwicklern auf ihren lokalen Workstations, dass sie die Vorlage aktualisieren müssen, um eine Laufzeit zu verwenden, die von der Organisation zugelassen ist. Dies geschieht, bevor ein Commit in ein Code-Repository übernommen wird und anschließend Prüfungen innerhalb einer CI/CD-Pipeline fehlschlagen. Ihre Entwickler erhalten direktes Feedback, wie sie konforme Vorlagen entwickeln und können ihre Zeit darauf verwenden, Code zu schreiben, der geschäftlichen Nutzen bietet. Diese Kontrolle kann vor der Bereitstellung auf der lokalen Entwickler-Workstation, in einem Pre-Commit-Validierungs-Webhook und/oder in der CI/CD-Pipeline angewendet werden.

Einschränkungen

Wenn Sie AWS Serverless Application Model-(AWS SAM)-Vorlagen verwenden, um Lambda-Funktionen zu definieren, beachten Sie, dass Sie die Guard-Regel aktualisieren müssen, um nach dem Ressourcentyp `AWS::Serverless::Function` wie folgt zu suchen.

```
let lambda_functions = Resources.*[
    Type == "AWS::Serverless::Function"
]
```

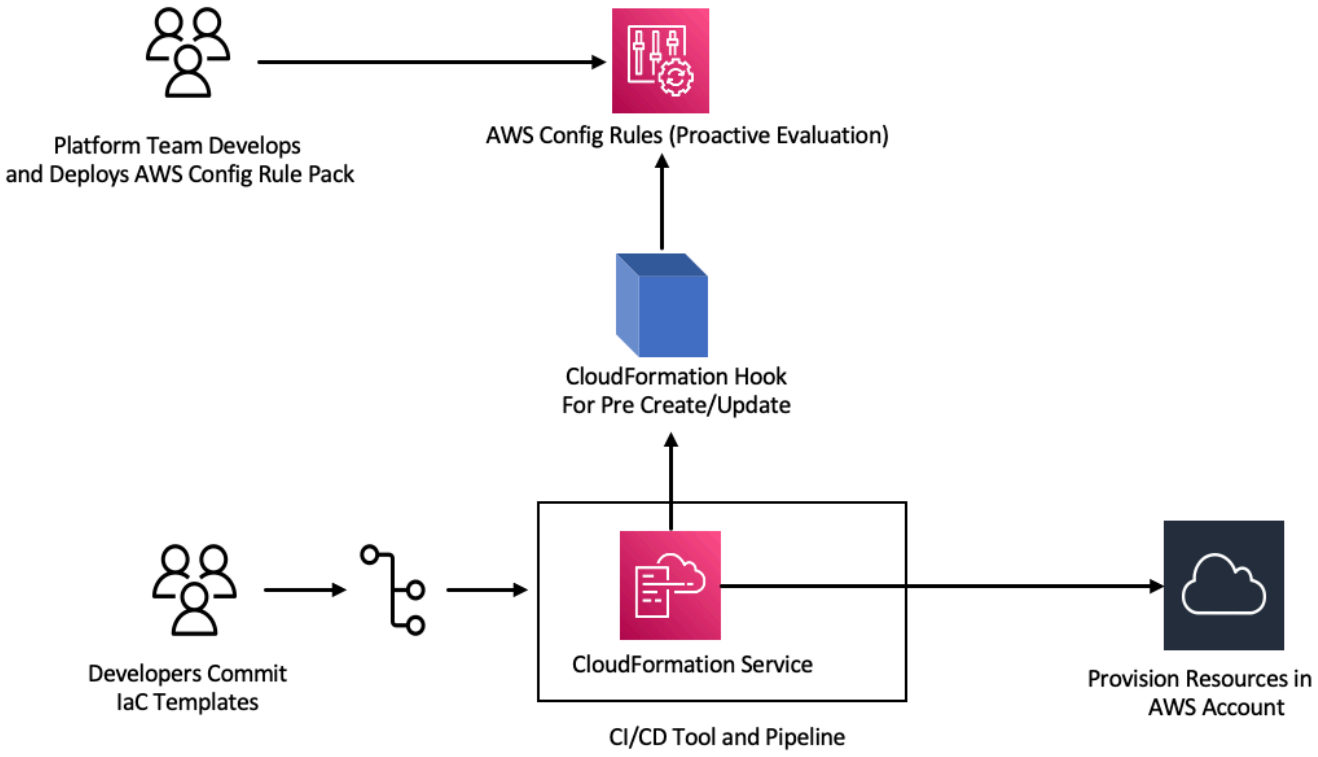
Guard erwartet außerdem, dass die Eigenschaften in der Ressourcendefinition enthalten sind. AWS SAM-Vorlagen ermöglichen die Angabe von Eigenschaften in einem separaten Globals-Abschnitt. Eigenschaften, die im Globals-Abschnitt definiert sind, werden nicht mit Ihren Guard-Regeln validiert.

Beachten Sie, dass Guard, wie in der Dokumentation zur Fehlerbehebung bei Guard beschrieben, keine intrinsischen Kurzformen wie `!GetAtt` oder `!Sub` unterstützt, sondern stattdessen die Verwendung der erweiterten Formen erfordert: `Fn::GetAtt` und `Fn::Sub`. (Im vorherigen Beispiel wird die Role-Eigenschaft nicht ausgewertet, daher wurde der Einfachheit halber die intrinsische Kurzform verwendet.)

Implementieren Sie präventive Kontrollen für Lambda mit AWS Config

Es ist wichtig, so früh wie möglich im Entwicklungsprozess die Konformität Ihrer Serverless-Anwendungen sicherzustellen. In diesem Thema behandeln wir die Implementierung präventiver Kontrollen mithilfe von AWS Config. Auf diese Weise können Sie Konformitätsprüfungen zu einem früheren Zeitpunkt im Entwicklungsprozess implementieren und dieselben Kontrollen in Ihren Pipelines implementieren. CI/CD Dadurch werden auch Ihre Kontrollen in einem zentral verwalteten Regelspeicher standardisiert, sodass Sie Ihre Kontrollen konsistent auf alle Konten anwenden können. AWS

Nehmen wir beispielsweise an, Ihre Compliance-Administratoren haben eine Anforderung definiert, um sicherzustellen, dass alle Lambda-Funktionen die AWS X-Ray Ablaufverfolgung beinhalten. Mit dem AWS Config proaktiven Modus können Sie vor der Bereitstellung Konformitätsprüfungen Ihrer Lambda-Funktionsressourcen durchführen, wodurch das Risiko der Bereitstellung falsch konfigurierter Lambda-Funktionen verringert und Entwicklern Zeit gespart wird, indem sie ihnen schnelleres Feedback zur Infrastruktur als Codevorlagen geben. Im Folgenden wird der Ablauf präventiver Kontrollen visualisiert mit: AWS Config



Nehmen wir an, es wird festgelegt, dass für alle Lambda-Funktionen die Ablaufverfolgung aktiviert sein muss. Als Reaktion darauf stellt das Plattformteam fest, dass eine bestimmte AWS Config Regel proaktiv für alle Konten ausgeführt werden muss. Diese Regel kennzeichnet jede Lambda-Funktion, für die keine X-Ray-Ablaufverfolgung konfiguriert ist, als nicht konforme Ressource. Das Team entwickelt eine Regel, verpackt sie in ein Konformitätspaket und stellt das Konformitätspaket für alle Konten bereit, um sicherzustellen, dass alle AWS Konten in der Organisation diese Kontrollen einheitlich anwenden. Sie können die Regel in der AWS CloudFormation Guard 2.x.x-Syntax schreiben, die folgende Form hat:

```
rule name when condition { assertion }
```

Im Folgenden finden Sie ein Beispiel für eine Guard-Regel, die überprüft, ob für Lambda-Funktionen die Ablaufverfolgung aktiviert ist:

```
rule lambda_tracing_check {
  when configuration.tracingConfig exists {
      configuration.tracingConfig.mode == "Active"
  }
}
```

Das Plattformteam ergreift weitere Maßnahmen und schreibt vor, dass bei jeder AWS CloudFormation Bereitstellung ein Pre-Create/Update-Hook aufgerufen wird. Das Team übernimmt die volle Verantwortung für die Entwicklung dieses Hooks und die Konfiguration der Pipeline, mit dem Ziel, die zentrale Kontrolle der Compliance-Regeln zu stärken und ihre konsistente Anwendung in allen Implementierungen sicherzustellen. Informationen zur Entwicklung, Paketierung und Registrierung eines AWS CloudFormation Hooks finden Sie in der Dokumentation zur CloudFormation Befehlszeilenschnittstelle (CFN-CLI) unter Developing Hooks. Sie können die CloudFormation CLI verwenden, um das Hook-Projekt zu erstellen:

```
cfn init
```

Der Befehl fragt Sie nach einigen grundlegenden Informationen zu Ihrem Hook-Projekt und erstellt ein Projekt mit den folgenden Dateien:

```
README.md
<hook-name>.json
rpdk.log
src/handler.py
template.yml
```

```
hook-role.yaml
```

Der Hook-Entwickler fügt der Konfigurationsdatei `<hook-name>.json` den gewünschten Zielressourcentyp hinzu. In der folgenden Konfiguration ist ein Hook so konfiguriert, dass er ausgeführt wird, bevor eine Lambda-Funktion mit CloudFormation erstellt wird. Sie können ähnliche Handler auch für die Aktionen `preUpdate` und `preDelete` hinzufügen.

```
    "handlers": {
        "preCreate": {
            "targetNames": [
                "AWS::Lambda::Function"
            ],
            "permissions": []
        }
    }
```

Sie müssen auch sicherstellen, dass der CloudFormation Hook über die entsprechenden Berechtigungen zum Aufrufen von verfügt. AWS Config APIs Aktualisieren Sie dazu die Rollendefinitionsdatei `hook-role.yaml`. Die Rollendefinitionsdatei hat standardmäßig die folgende Vertrauensrichtlinie, die es ermöglicht, die Rolle CloudFormation zu übernehmen.

```
      AssumeRolePolicyDocument:
        Version: '2012-10-17'
        Statement:
          - Effect: Allow
            Principal:
              Service:
                - hooks.cloudformation.amazonaws.com
                - resources.cloudformation.amazonaws.com
```

Damit dieser Hook die Konfiguration aufrufen kann APIs, müssen Sie der Policy-Anweisung die folgenden Berechtigungen hinzufügen. Anschließend reichen Sie das Hook-Projekt mit dem `cfn submit` Befehl ein, der eine Rolle mit den erforderlichen Berechtigungen für Sie CloudFormation erstellt.

```
      Policies:
        - PolicyName: HookTypePolicy
          PolicyDocument:
            Version: '2012-10-17'
            Statement:
              - Effect: Allow
```

```
                Action:
                  - "config:Describe*"
                  - "config:Get*"
                  - "config:List*"
                  - "config:SelectResourceConfig"
                Resource: "*
```

Als Nächstes müssen Sie eine Lambda-Funktion in eine `src/handler.py`-Datei schreiben. In dieser Datei finden Sie Methoden mit dem Namen und `preCreate`, `preUpdate` und `preDelete`, die bereits bei der Initiierung des Projekts erstellt wurden. Sie möchten eine allgemeine, wiederverwendbare Funktion schreiben, die die AWS Config `StartResourceEvaluation` API im proaktiven Modus aufruft, indem Sie den verwenden AWS SDK für Python (Boto3). Dieser API-Aufruf verwendet Ressourceneigenschaften als Eingabe und vergleicht die Ressource mit der Regeldefinition.

```
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.")
```

Jetzt können Sie die allgemeine Funktion vom Handler für den Pre-Create-Hook aus aufrufen. Ein Beispiel für den Handler:

```
@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}")
```

Nach diesem Schritt können Sie den Hook registrieren und ihn so konfigurieren, dass er alle Ereignisse bei der AWS Lambda Funktionserstellung abhört.

Ein Entwickler bereitet die IaC-Vorlage (Infrastructure as Code) für einen Serverless-Microservice mithilfe von Lambda vor. Diese Vorbereitung umfasst die Einhaltung interner Standards, gefolgt von lokalen Tests und dem Commit der Vorlage in das Repository. Hier ist ein Beispiel für eine IaC-Vorlage:

```
  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
```

Im Rahmen des CI/CD Prozesses ruft der CloudFormation Dienst bei der Bereitstellung der CloudFormation Vorlage den Hook vor der Erstellung und Aktualisierung unmittelbar vor der Bereitstellung des Ressourcentyps auf. `AWS::Lambda::Function` Der Hook verwendet AWS Config Regeln, die im proaktiven Modus ausgeführt werden, um zu überprüfen, ob die Lambda-Funktionskonfiguration die vorgeschriebene Ablaufverfolgungskonfiguration enthält. Die Antwort des Hooks bestimmt den nächsten Schritt. Bei Einhaltung der Vorschriften signalisiert der Hook den Erfolg und CloudFormation fährt mit der Bereitstellung der Ressourcen fort. Wenn nicht, schlägt die CloudFormation Stack-Bereitstellung fehl, die Pipeline wird sofort gestoppt und das System zeichnet die Details zur späteren Überprüfung auf. An die relevanten Stakeholder werden Compliance-Benachrichtigungen gesendet.

Sie finden die success/fail Hook-Informationen in der CloudFormation Konsole:

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	-

Wenn Sie Logs für Ihren CloudFormation Hook aktiviert haben, können Sie das Ergebnis der Hook-Auswertung erfassen. Hier ist ein Beispielprotokoll für einen Hook mit dem Status „Fehlgeschlagen“, was darauf hinweist, dass für die Lambda-Funktion X-Ray nicht aktiviert ist:

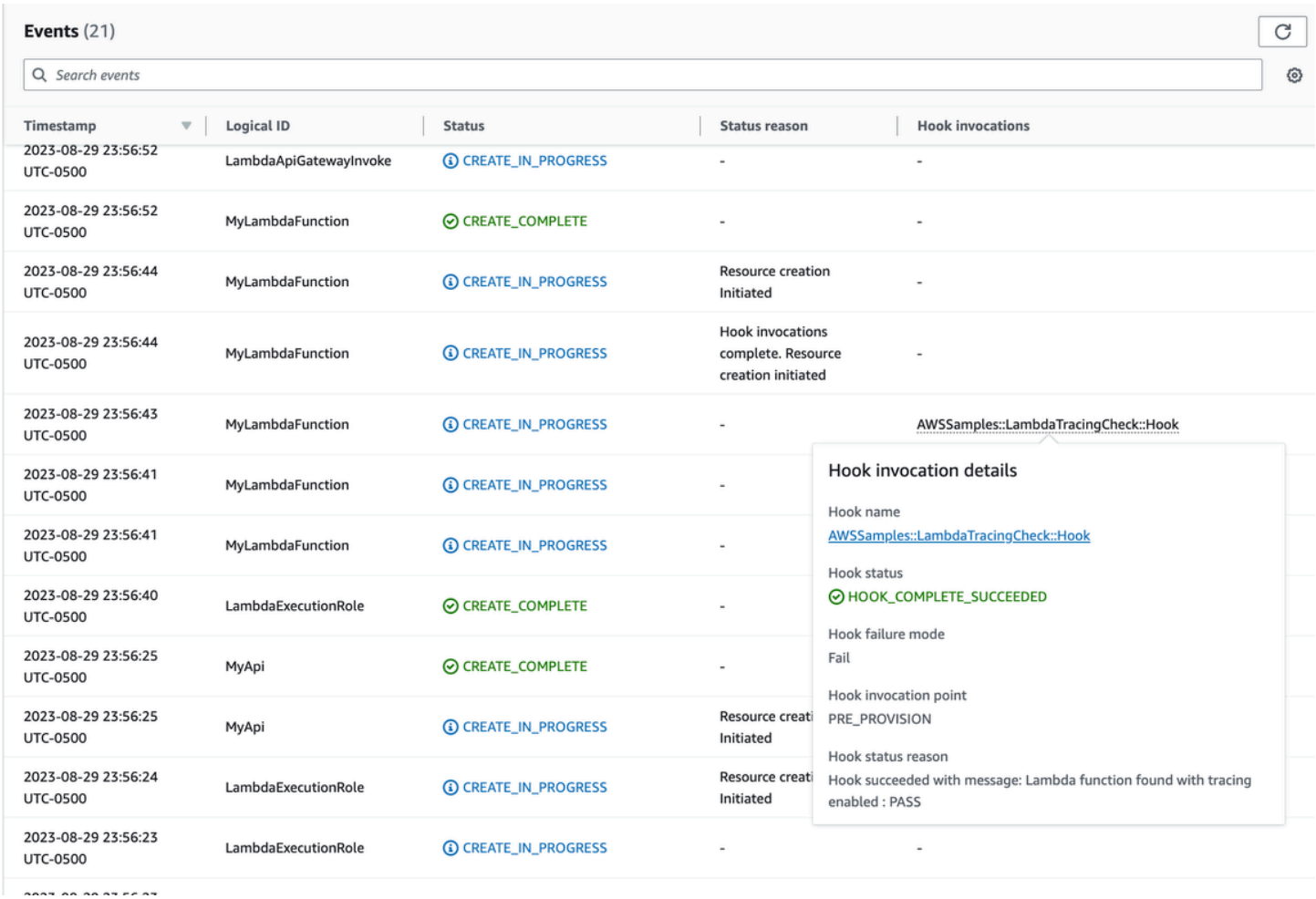
2023-08-29T23:50:17.574-05:00 ProgressEvent(status=<OperationStatus.FAILED: 'FAILED'>, errorCode=<HandlerErrorCode.NonCompliant: 'NonCompliant'…

```
ProgressEvent(status=<OperationStatus.FAILED: 'FAILED'>, errorCode=<HandlerErrorCode.NonCompliant: 'NonCompliant'>, message='Lambda
function found with no tracing enabled : FAILED', result=None, callbackContext=None, callbackDelaySeconds=0, resourceModel=None,
resourceModels=None, nextToken=None)
```

Copy

No newer events at this moment. *Auto retry paused.* Resume

Hat der Entwickler entschieden, die IaC so zu ändern, dass der Wert `TracingConfig Mode` in `Active` aktualisiert wird und die Bereitstellung erneut erfolgt, wird der Hook erfolgreich ausgeführt und der Stack fährt mit der Erstellung der Lambda-Ressource fort.



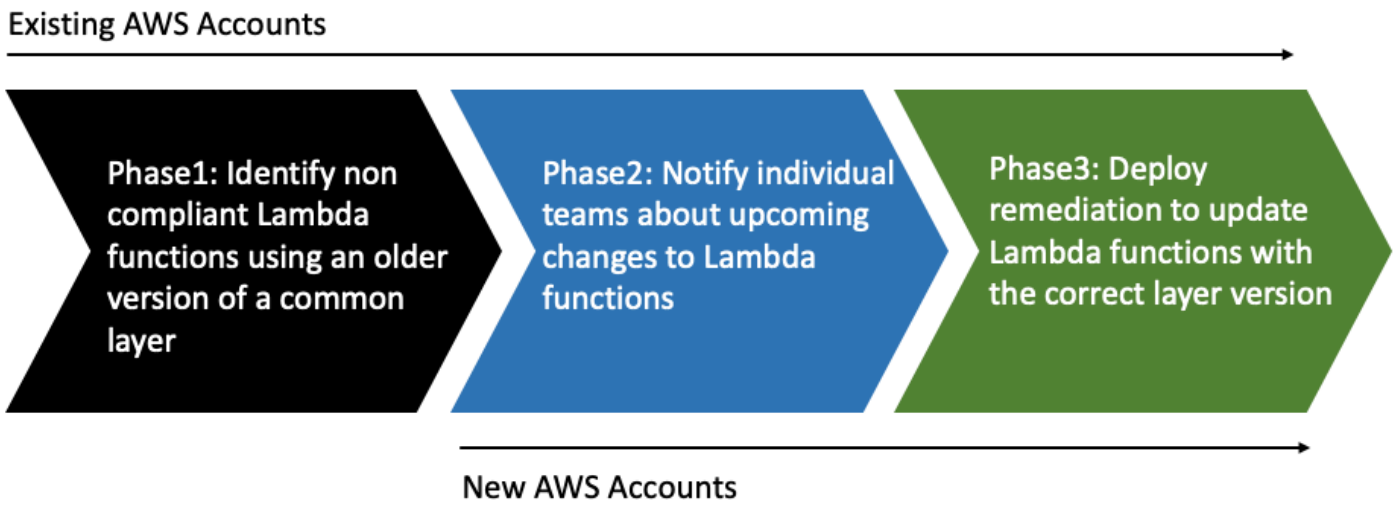
Auf diese Weise können Sie präventive Kontrollen AWS Config im proaktiven Modus implementieren, wenn Sie serverlose Ressourcen in Ihren AWS Konten entwickeln und bereitstellen. Durch die Integration von AWS Config Regeln in die CI/CD Pipeline können Sie nicht konforme Ressourcenbereitstellungen identifizieren und optional blockieren, z. B. Lambda-Funktionen, denen eine aktive Ablaufverfolgungskonfiguration fehlt. Dadurch wird sichergestellt, dass nur Ressourcen in Ihren Umgebungen eingesetzt werden, die den neuesten Governance-Richtlinien entsprechen. AWS

Erkennen Sie nicht konforme Lambda-Bereitstellungen und -Konfigurationen mit AWS Config

Neben der proaktiven Evaluierung kann AWS Config auch im Nachhinein Ressourcenbereitstellungen und -konfigurationen erkennen, die nicht Ihren Governance-Richtlinien entsprechen. Das ist wichtig, weil sich Governance-Richtlinien ändern können, wenn Ihr Unternehmen sich weiterentwickelt und neue Best-Practices eingeführt werden.

Angenommen, Sie legen bei der Bereitstellung oder Aktualisierung von Lambda-Funktionen eine brandneue Richtlinie fest: Alle Lambda-Funktionen müssen immer eine bestimmte, genehmigte Lambda-Layer-Version verwenden. Sie können AWS Config konfigurieren, um neue oder aktualisierte Funktionen für Layer-Konfigurationen zu überwachen. Wenn AWS Config auf eine Funktion stößt, die keine genehmigte Layer-Version verwendet, wird die Funktion als nicht konforme Ressource gekennzeichnet. Sie können AWS Config optional so konfigurieren, dass die Ressource automatisch repariert wird, indem Sie mithilfe eines AWS Systems Manager-Automatisierungsdokuments eine Behebungsaktion angeben. Sie könnten ein Automatisierungsdokument in Python mit AWS SDK für Python (Boto3) schreiben, das die nicht konforme Funktion aktualisiert, sodass sie auf die genehmigte Layer-Version verweist. AWS Config dient somit sowohl als detektive als auch als korrektive Kontrolle und automatisiert das Compliance-Management.

Wir brechen diesen Prozess in drei zentrale Implementierungsphasen auf:



Phase 1: Identifizieren der Zugriffsressourcen

Aktivieren Sie zunächst AWS Config für alle Ihre Konten und konfigurieren Sie es so, dass AWS-Lambda-Funktionen aufgezeichnet werden. AWS Config kann so erkennen, wenn Lambda-

Funktionen erstellt oder aktualisiert werden. Anschließend können Sie benutzerdefinierte Richtlinienregeln konfigurieren, um nach bestimmten Richtlinienverstößen zu suchen, die AWS CloudFormation Guard-Syntax verwenden. Guard-Regeln haben die folgende allgemeine Form:

```
rule name when condition { assertion }
```

Im Folgenden finden Sie eine Beispielregel, mit der überprüft wird, ob ein Layer auf eine alte Layer-Version eingestellt ist:

```
rule desiredlayer when configuration.layers !empty {
    some configuration.layers[*].arn != CONFIG_RULE_PARAMETERS.OldLayerArn
}
```

Sehen wir uns die Syntax und den Regelaufbau etwas genauer an:

- Regelname: Der Name der Regel im Beispiel lautet `desiredlayer`.
- Bedingung: Diese Klausel gibt die Bedingung an, unter der die Regel überprüft werden soll. Im angegebenen Beispiel lautet die Bedingung `configuration.layers !empty`. Das bedeutet, dass die Ressource nur ausgewertet werden sollte, wenn die `layers`-Eigenschaft in der Konfiguration nicht leer ist.
- Assertion: Nach der `when`-Klausel bestimmt die Assertion, was die Regel überprüft. Die Assertion (Behauptung) `some configuration.layers[*].arn != CONFIG_RULE_PARAMETERS.OldLayerArn` prüft, ob einer der ARNs des Lambda-Layers nicht mit dem Wert `OldLayerArn` übereinstimmt. Wenn es keine Übereinstimmung gibt, ist die Assertion wahr und die Regel gilt als bestanden. Andernfalls schlägt sie fehl.

`CONFIG_RULE_PARAMETERS` ist ein spezieller Parametersatz, mit dem die AWS Config-Regel konfiguriert wird. In diesem Fall ist `OldLayerArn` ein Parameter in `CONFIG_RULE_PARAMETERS`. Benutzer können so einen bestimmten ARN-Wert als veraltet oder abgelaufen festlegen. Anschließend überprüft die Regel, ob Lambda-Funktionen diesen alten ARN verwenden.

Phase 2: Visualisierung und Entwicklung

AWS Config erfasst Konfigurationsdaten und speichert sie in Amazon Simple Storage Service (Amazon S3)-Buckets. Sie können Amazon Athena verwenden, um diese Daten direkt aus Ihren S3-Buckets abzufragen. Mit Athena können Sie diese Daten auf Organisationsebene aggregieren und sich einen ganzheitlichen Überblick über die Ressourcenkonfigurationen in all Ihren Konten

verschaffen. Informationen zum Einrichten der Aggregation von Ressourcenkonfigurationsdaten finden Sie unter Visualizing AWS Config data using Athena and Amazon Quick Suite im Blog „ AWS Cloud Operations & Management“.

Im Folgenden finden Sie ein Beispiel für eine Athena-Abfrage zur Identifizierung aller Lambda-Funktionen, die einen bestimmten Layer-ARN verwenden:

```
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%'
```

Hier sind die Ergebnisse der Abfrage:

Query results | Query stats

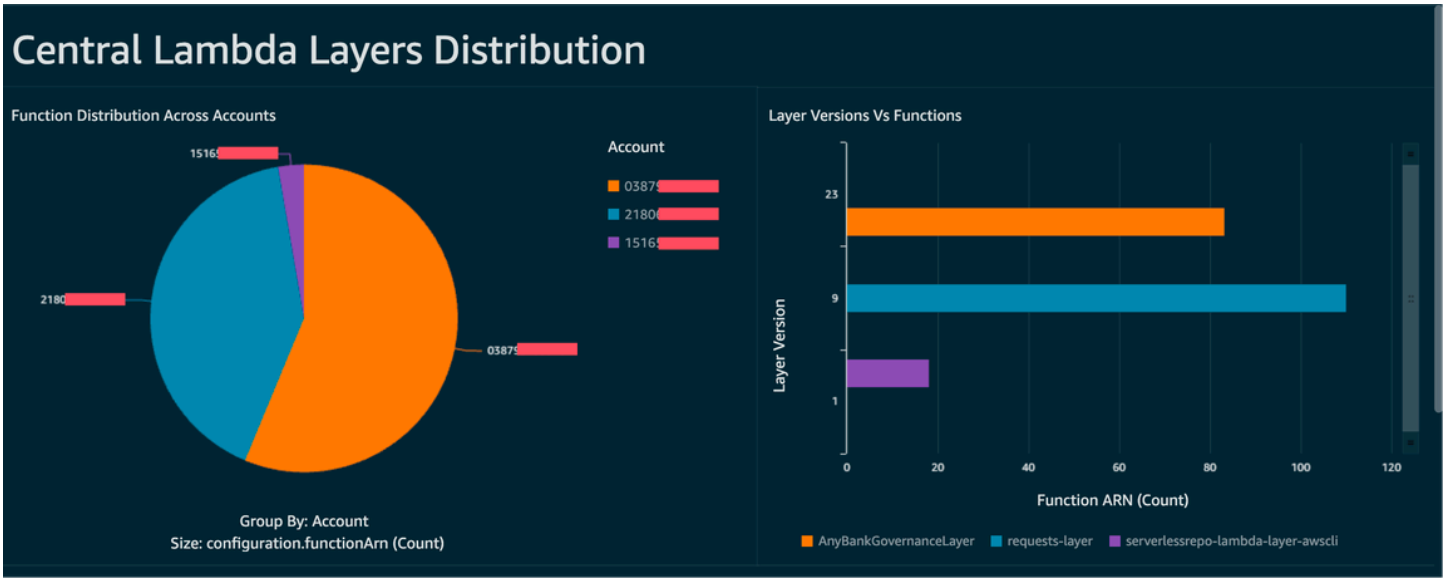
Completed — Time in queue: 127 ms — Run time: 1.803 sec — Data scanned: 239.40 KB

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

Nachdem die AWS Config-Daten unternehmensweit aggregiert wurden, können Sie mit Amazon Quick Suite ein Dashboard erstellen. Sie importieren die Ergebnisse aus Athena in Quick Suite und können dann visualisieren, wie gut Ihre Lambda-Funktionen der Ebenenversionsregel entsprechen. Im Dashboard können Sie konforme und nicht konforme Ressourcen hervorheben und dementsprechend Durchsetzungsrichtlinien festlegen, wie im nächsten Abschnitt beschrieben. Die folgende Abbildung zeigt ein Beispiel-Dashboard, in dem die Verteilung der Layer-Versionen auf Funktionen innerhalb der Organisation aufgeschlüsselt ist.



Phase 3: Implementierung und Durchsetzung

Sie können Ihre Layer-Versionsregel, die Sie in Phase 1 erstellt haben, jetzt optional über ein Systems-Manager-Automatisierungsdokument, das Sie als Python-Skript mit AWS SDK für Python (Boto3) erstellt haben, mit einer Behebungsaktion verknüpfen. Das Skript ruft für jede Lambda-

Funktion die API-Aktion UpdateFunctionConfiguration auf und aktualisiert die Funktionskonfiguration mit dem neuen Layer-ARN. Alternativ können Sie das Skript so schreiben, dass eine Pull-Anfrage an das Code-Repository gesendet wird, um den Layer-ARN zu aktualisieren. Auf diese Weise werden künftige Codebereitstellungen ebenfalls mit dem richtigen Layer-ARN aktualisiert.

Lambda-Codesignatur mit AWS Signer

AWS Signer ist ein vollständig verwalteter Codesignaturdienst, mit dem Sie Ihren Code anhand einer digitalen Signatur validieren können, um sicherzustellen, dass er unverändert ist und von einem vertrauenswürdigen Publisher stammt. AWS Signer kann in Verbindung mit AWS Lambda verwendet werden, um vor der Implementierung in Ihren AWS-Umgebungen zu prüfen, ob Funktionen und Layer unverändert sind. Ihr Unternehmen bleibt so vor böswilligen Akteuren geschützt, die sich möglicherweise Zugangsdaten verschafft haben, um neue Funktionen zu entwickeln oder bestehende Funktionen zu verändern.

Um die Codesignatur für Ihre Lambda-Funktionen einzurichten, erstellen Sie zunächst einen S3-Bucket mit aktivierter Versionierung. Erstellen Sie anschließend ein Signaturprofil mit AWS Signer, geben Sie Lambda als Plattform und die Gültigkeitsdauer des Signaturprofils in Tagen an. Beispiel:

```
Signer:
  Type: AWS::Signer::SigningProfile
  Properties:
    PlatformId: AWSLambda-SHA384-ECDSA
    SignatureValidityPeriod:
      Type: DAYS
      Value: !Ref pValidDays
```

Verwenden Sie dann das Signaturprofil und erstellen Sie eine Signaturkonfiguration mit Lambda. Legen Sie fest, was geschehen soll, wenn in der Signaturkonfiguration ein Artefakt erkannt wird, das nicht mit der erwarteten digitalen Signatur übereinstimmt: warnen (aber die Bereitstellung zulassen) oder erzwingen (und die Bereitstellung blockieren). Das folgende Beispiel ist für Erzwingen und Bereitstellungen blockieren konfiguriert.

```
SigningConfig:
  Type: AWS::Lambda::CodeSigningConfig
  Properties:
    AllowedPublishers:
      SigningProfileVersionArns:
        - !GetAtt Signer.ProfileVersionArn
    CodeSigningPolicies:
      UntrustedArtifactOnDeployment: Enforce
```

Sie haben AWS Signer mit Lambda jetzt so konfiguriert, dass nicht vertrauenswürdige Bereitstellungen blockiert werden. Nehmen wir an, Sie haben eine Feature-Anfrage fertig programmiert und sind nun bereit, die Funktion bereitzustellen. Der erste Schritt besteht darin, den

Code mit den entsprechenden Abhängigkeiten zu komprimieren und dann das Artefakt mit dem von Ihnen erstellten Signaturprofil zu signieren. Sie können dies tun, indem Sie das ZIP-Artefakt in den S3-Bucket hochladen und dann einen Signaturauftrag starten.

```
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
```

Das Ergebnis ist eine Ausgabe wie die folgende, wobei `jobId` das Objekt ist, das im Ziel-Bucket und Präfix erstellt wurde, und `jobOwner` die 12-stellige AWS-Konto-ID, unter der der Auftrag ausgeführt wurde.

```
{
    "jobId": "87a3522b-5c0b-4d7d-b4e0-4255a8e05388",
    "jobOwner": "111122223333"
  }
```

Jetzt können Sie Ihre Funktion mithilfe des signierten S3-Objekts und der von Ihnen erstellten Codesignatur-Konfiguration bereitstellen.

```
  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
```

Sie können eine Funktionsbereitstellung alternativ mit dem ursprünglichen unsignierten Quell-ZIP-Artefakt testen. Die Bereitstellung sollte mit der folgenden Fehlermeldung fehlschlagen:

```
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.
```

Wenn Sie Ihre Funktionen mithilfe von AWS Serverless Application Model (AWS SAM) erstellen und bereitstellen, übernimmt der Paketbefehl das Hochladen des ZIP-Artefakts auf S3 und startet auch

den Signaturauftrag und ruft das signierte Artefakt ab. Verwenden Sie dazu den folgenden Befehl mit diesen Parametern:

```
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
```

Mit AWS Signer können Sie überprüfen, ob ZIP-Artefakte, die in Ihren Konten bereitgestellt werden, vertrauenswürdig sind. Sie können den oben genannten Prozess in Ihre CI/CD-Pipelines aufnehmen und verlangen, dass allen Funktionen eine Codesignatur-Konfiguration zugewiesen ist. Verwenden Sie dabei die in den vorherigen Themen beschriebenen Techniken. Durch die Verwendung einer Codesignatur bei der Bereitstellung von Lambda-Funktionen verhindern Sie, dass böswillige Akteure, die möglicherweise Anmeldeinformationen für die Erstellung oder Änderung von Funktionen erhalten haben, bösartigen Code in Ihre Funktionen einschleusen.

Automatisieren Sie Sicherheitsbewertungen für Lambda mit Amazon Inspector

Amazon Inspector ist ein automatisierter Schwachstellen-Management-Service, der Workloads kontinuierlich auf bekannte Softwareschwachstellen und unbeabsichtigte Netzwerkfreigabe durchsucht. Amazon Inspector erstellt einen Bericht, der die Schwachstelle beschreibt, die betroffene Ressource identifiziert, den Schweregrad der Schwachstelle bewertet und Hilfestellung zur Behebung gibt.

Amazon Inspector bietet eine kontinuierliche, automatisierte Schwachstellenanalyse für Lambda-Funktionen und -Layer. Amazon Inspector stellt zwei Scantypen für Lambda bereit:

- Lambda-Standardscan (Standard): Scannt Anwendungsabhängigkeiten innerhalb einer Lambda-Funktion und ihrer Layer auf Paketschwachstellen.
- Lambda-Codescan: Scannt den benutzerdefinierten Anwendungscode in Funktionen und Layern auf Code-Schwachstellen. Sie können entweder den Lambda-Standardscan einzeln oder zusammen mit dem Lambda-Codescan aktivieren.

Um Amazon Inspector zu aktivieren, navigieren Sie zur Amazon-Inspector-Konsole, erweitern Sie den Bereich Einstellungen und wählen Sie Kontoverwaltung. Wählen Sie auf der Registerkarte Konten die Option Aktivieren und wählen Sie dann eine der Scanoptionen aus.

Sie können Amazon Inspector für mehrere Konten aktivieren und bei der Einrichtung von Amazon Inspector die Berechtigungen zur Verwaltung von Amazon Inspector für die Organisation an bestimmte Konten delegieren. Während der Aktivierung müssen Sie Amazon Inspector Berechtigungen erteilen, indem Sie die folgende Rolle erstellen: `AWSServiceRoleForAmazonInspector2`. In der Amazon-Inspector-Konsole können Sie diese Rolle mit einem Klick erstellen.

Beim Lambda-Standardscan initiiert Amazon Inspector Schwachstellenscans von Lambda-Funktionen in den folgenden Situationen:

- Sobald Amazon Inspector eine bestehende Lambda-Funktion entdeckt.
- Wenn Sie eine neue Lambda-Funktion bereitstellen.
- Wenn Sie ein Update für den Anwendungscode oder die Abhängigkeiten einer vorhandenen Lambda-Funktion oder ihrer Layer bereitstellen.

- Immer wenn Amazon Inspector seiner Datenbank ein neues CVE-Element (Common Vulnerabilities and Exposures) hinzufügt und dieses CVE für Ihre Funktion relevant ist.

Beim Lambda-Codescan wertet Amazon Inspector Ihren Lambda-Funktionscode mithilfe von Automated Reasoning und Machine Learning aus. Dabei wird der Anwendungscode auf die allgemeine Einhaltung der Sicherheitsregeln hin analysiert. Wenn Amazon Inspector eine Sicherheitslücke in Ihrem Anwendungscode für Lambda-Funktionen entdeckt, erstellt Amazon Inspector einen detaillierten Schwachstellenbericht für den Code. Eine Liste möglicher Schwachstellen finden Sie in der Amazon CodeGuru Detector Library.

Rufen Sie die Amazon-Inspector-Konsole auf, um die Berichte einzusehen. Wählen Sie im Menü Funde die Option Nach Lambda-Funktion, um die Ergebnisse der Sicherheitsscans anzuzeigen, die für Lambda-Funktionen durchgeführt wurden.

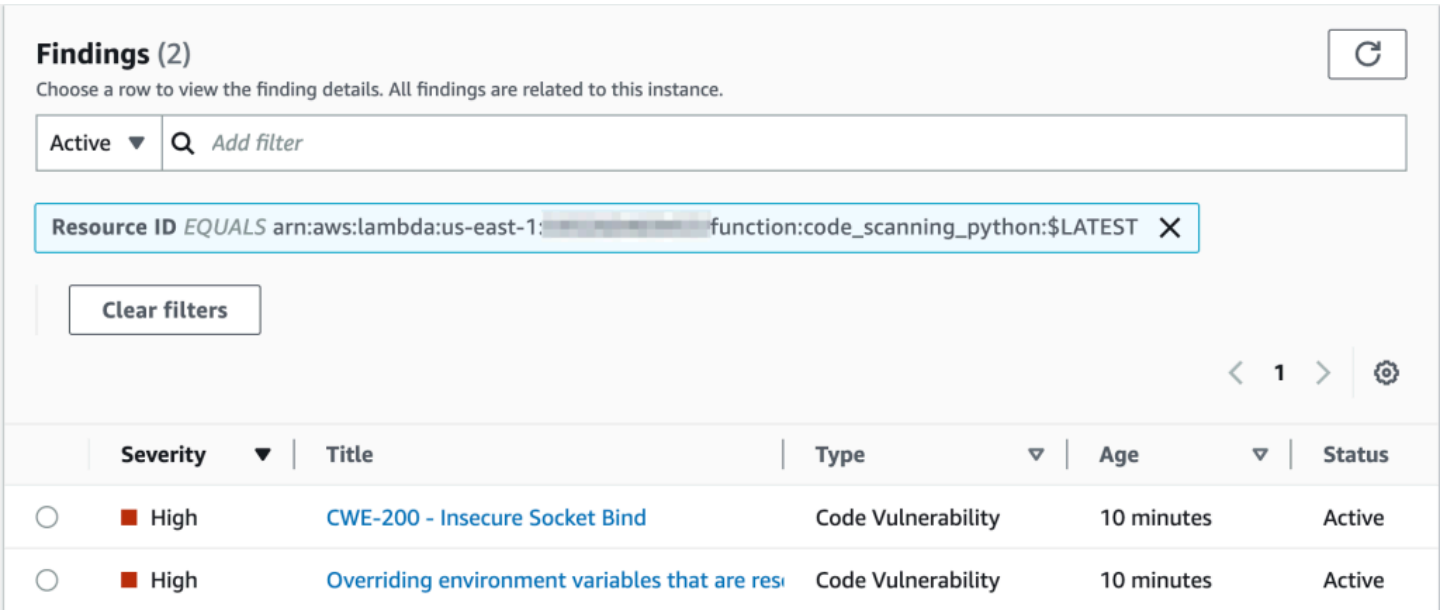
Um eine Lambda-Funktion vom Standardscan auszuschließen, kennzeichnen Sie die Funktion mit dem folgenden Schlüssel-Wert-Paar:

- `Key:InspectorExclusion`
- `Value:LambdaStandardScanning`

Um eine Lambda-Funktion von Codescans auszuschließen, kennzeichnen Sie die Funktion mit dem folgenden Schlüssel-Wert-Paar:

- `Key:InspectorCodeExclusion`
- `Value:LambdaCodeScanning`

Wie in der folgenden Abbildung dargestellt, erkennt Amazon Inspector Schwachstellen automatisch und kategorisiert die Funde beispielsweise als Codeschwachstelle. Das deutet darauf hin, dass sich die Schwachstelle im Funktionscode befindet und nicht in einer der codeabhängigen Bibliotheken. Sie können diese Details für eine bestimmte Funktion oder mehrere Funktionen gleichzeitig einsehen.



Sie können jeden Befund eingehend untersuchen und erfahren, wie Sie das Problem beheben können.

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.

Achten Sie bei der Verwendung von Lambda-Funktionen darauf, dass Sie die Namenskonventionen einhalten. Weitere Informationen finden Sie unter Arbeiten mit Lambda-Umgebungsvariablen.

Sie sind für die Lösungsvorschläge verantwortlich, die Sie akzeptieren. Lesen Sie sich die Lösungsvorschläge immer durch, bevor Sie sie annehmen. Möglicherweise müssen Sie Anpassungen an den Lösungsvorschlägen vornehmen, damit Ihr Code am Ende auch das tut, was Sie beabsichtigt haben.

Implementieren von Beobachtbarkeit für Lambda-Sicherheit und -Compliance

AWS Config ist ein nützliches Tool, um nicht konforme serverlose AWS Ressourcen zu finden und zu reparieren. Jede Änderung, die Sie an Ihren serverlosen Ressourcen vornehmen, wird in aufgezeichnet. AWS Config AWS Config Ermöglicht es Ihnen außerdem, Konfigurations-Snapshot-Daten auf S3 zu speichern. Sie können Amazon Athena und Amazon Quick Suite verwenden, um Dashboards zu erstellen und Daten anzuzeigen AWS Config . In Erkennen Sie nicht konforme Lambda-Bereitstellungen und -Konfigurationen mit AWS Config haben wir besprochen, wie sich bestimmte Konfigurationen wie Lambda-Layer visualisieren lassen. Dieses Thema erweitert diese Konzepte.

Abrufen von Informationen zu Lambda-Konfigurationen

Sie können Abfragen verwenden, um wichtige Konfigurationen wie AWS-Konto ID, Region, AWS X-Ray Ablaufverfolgungskonfiguration, VPC-Konfiguration, Speichergröße, Laufzeit und Tags abzurufen. Hier ist eine Beispielabfrage, mit der Sie diese Informationen von Athena abrufen können:

```
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
```

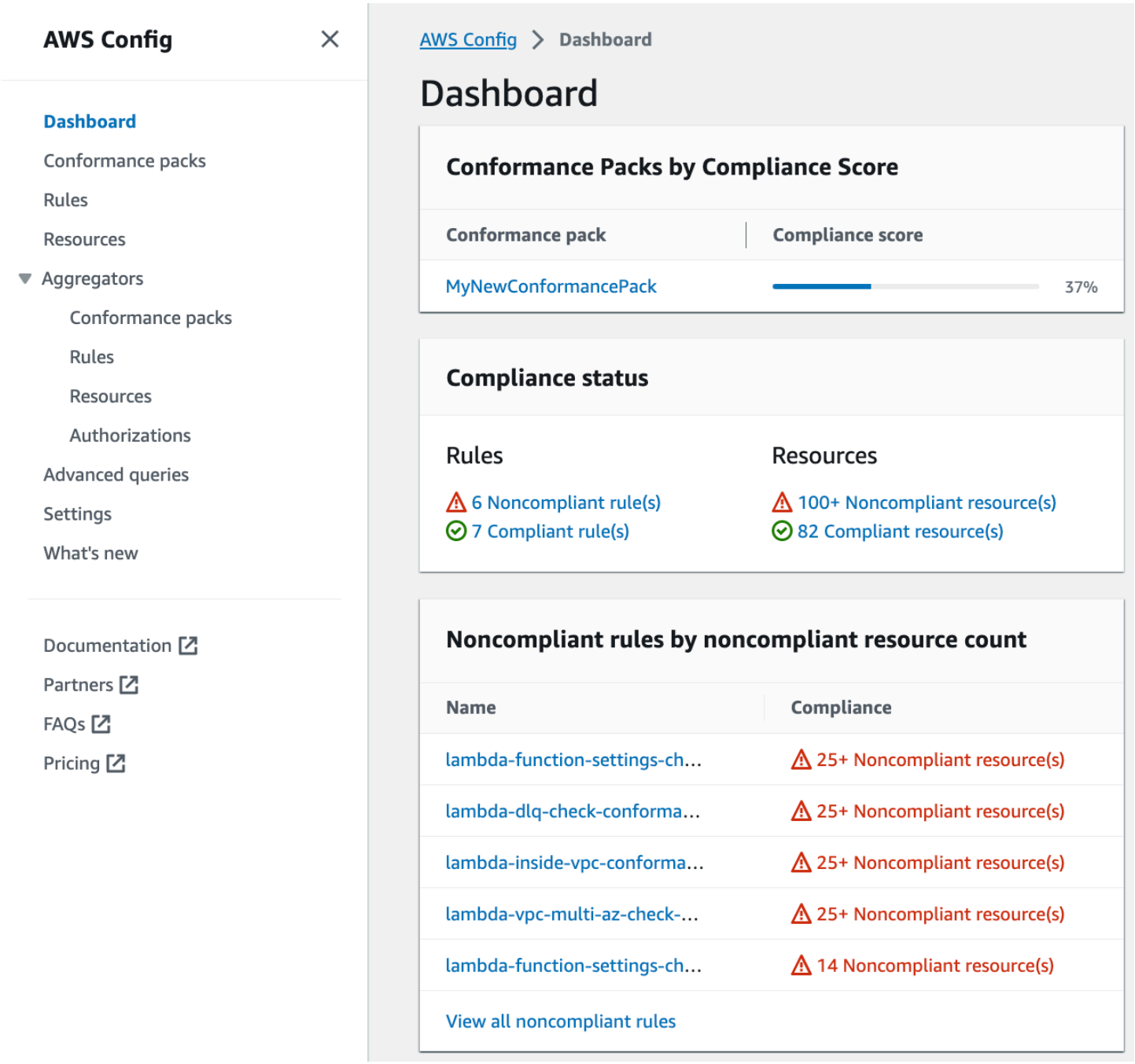
Sie können die Abfrage verwenden, um ein Quick-Suite-Dashboard zu erstellen und die Daten zu visualisieren. Informationen zum Aggregieren von AWS Ressourcenkonfigurationsdaten, zum Erstellen von Tabellen in Athena und zum Erstellen von Quick Suite-Dashboards mit den Daten von Athena finden Sie unter Visualisieren von AWS Config Daten mit Athena und Amazon Quick Suite im AWS Cloud Operations and Management-Blog. Diese Abfrage ruft auch Tag-Informationen für die Funktionen ab. Dies ermöglicht eine genauere Untersuchung Ihrer Workloads und Umgebungen, insbesondere wenn Sie benutzerdefinierte Tags verwenden.

Weitere Informationen über die Aktionen, die Sie ergreifen können, finden Sie im Abschnitt Behandlung von Beobachtbarkeitsbefunden weiter unten in diesem Thema.

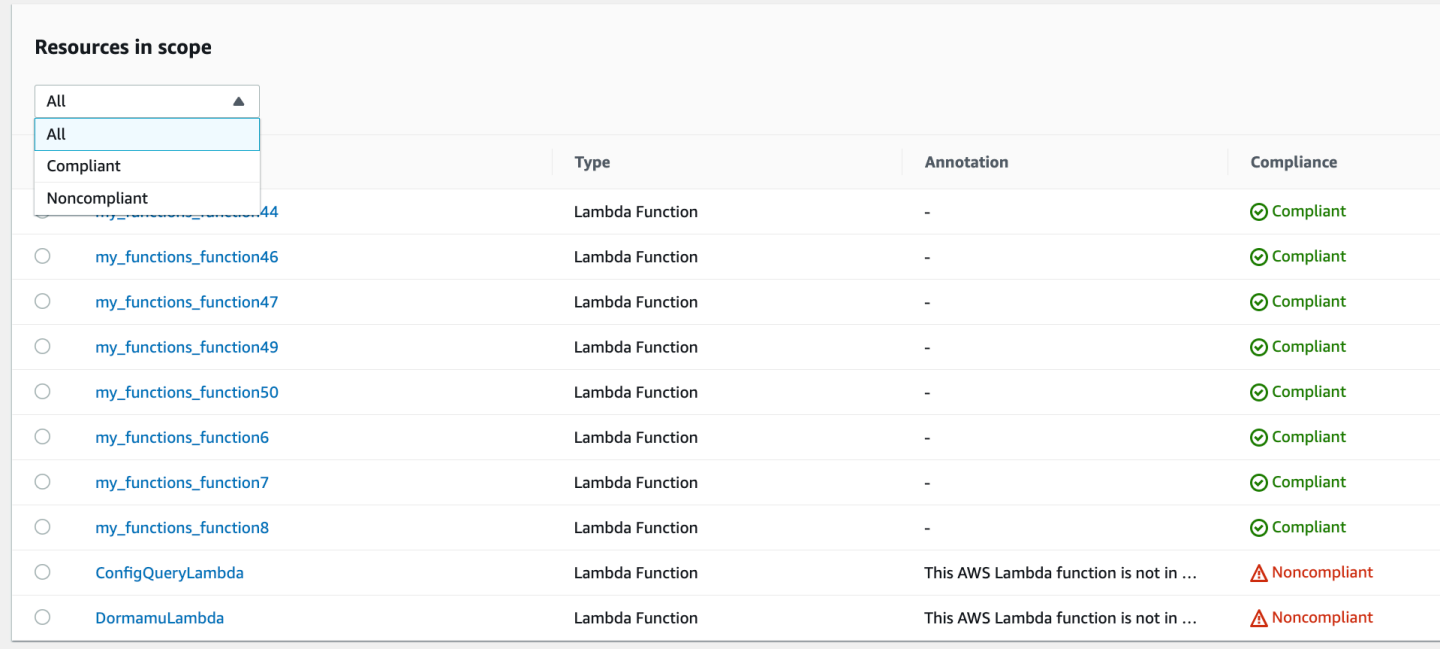
Abrufen von Informationen zur Lambda-Konformität

Mit den von generierten Daten können Sie Dashboards auf Organisationsebene erstellen AWS Config, um die Einhaltung der Vorschriften zu überwachen. Dies ermöglicht eine konsistente Nachverfolgung und Überwachung von:

- Compliance-Paketen nach Konformitätsbewertung
- Regeln von nicht konformen Ressourcen
- Compliance status (Compliance-Status)

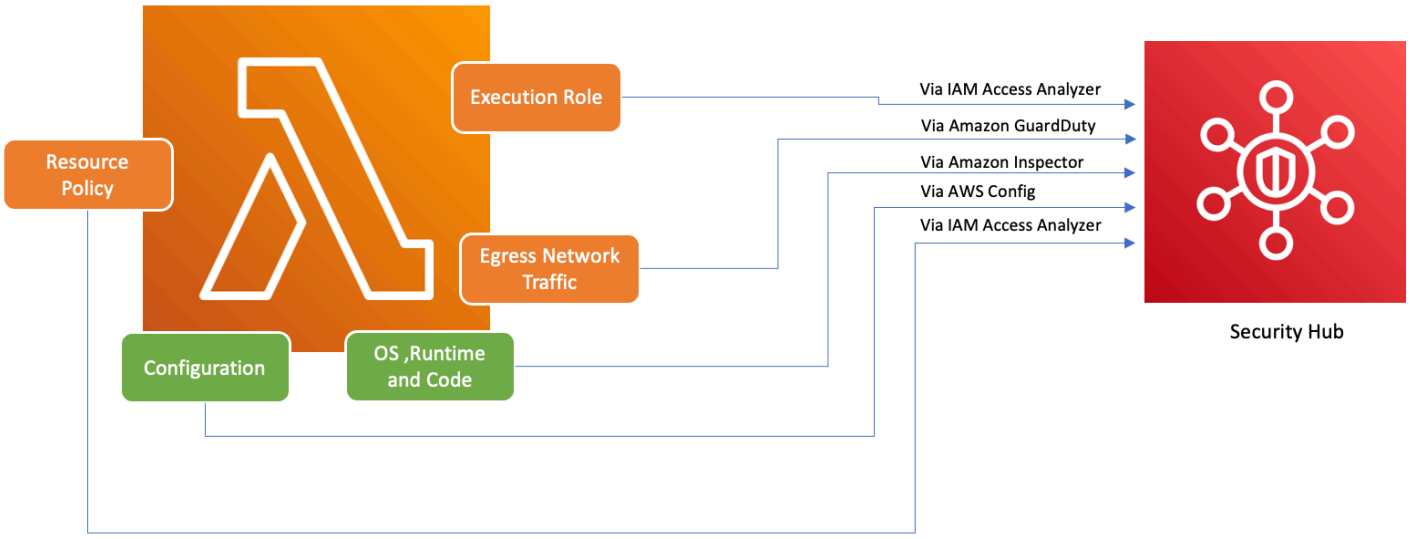


Überprüfen Sie jede Regel, um Ressourcen zu identifizieren, die für diese Regel nicht konform sind. Wenn Ihre Organisation beispielsweise vorschreibt, dass alle Lambda-Funktionen einer VPC zugeordnet werden müssen, und Sie eine AWS Config Regel zur Konformitätserkennung bereitgestellt haben, können Sie die `lambda-inside-vpc` Regel in der obigen Liste auswählen.



Weitere Informationen über die Aktionen, die Sie ergreifen können, finden Sie im Abschnitt [Behandlung von Beobachtbarkeitsbefunden](#) weiter unten.

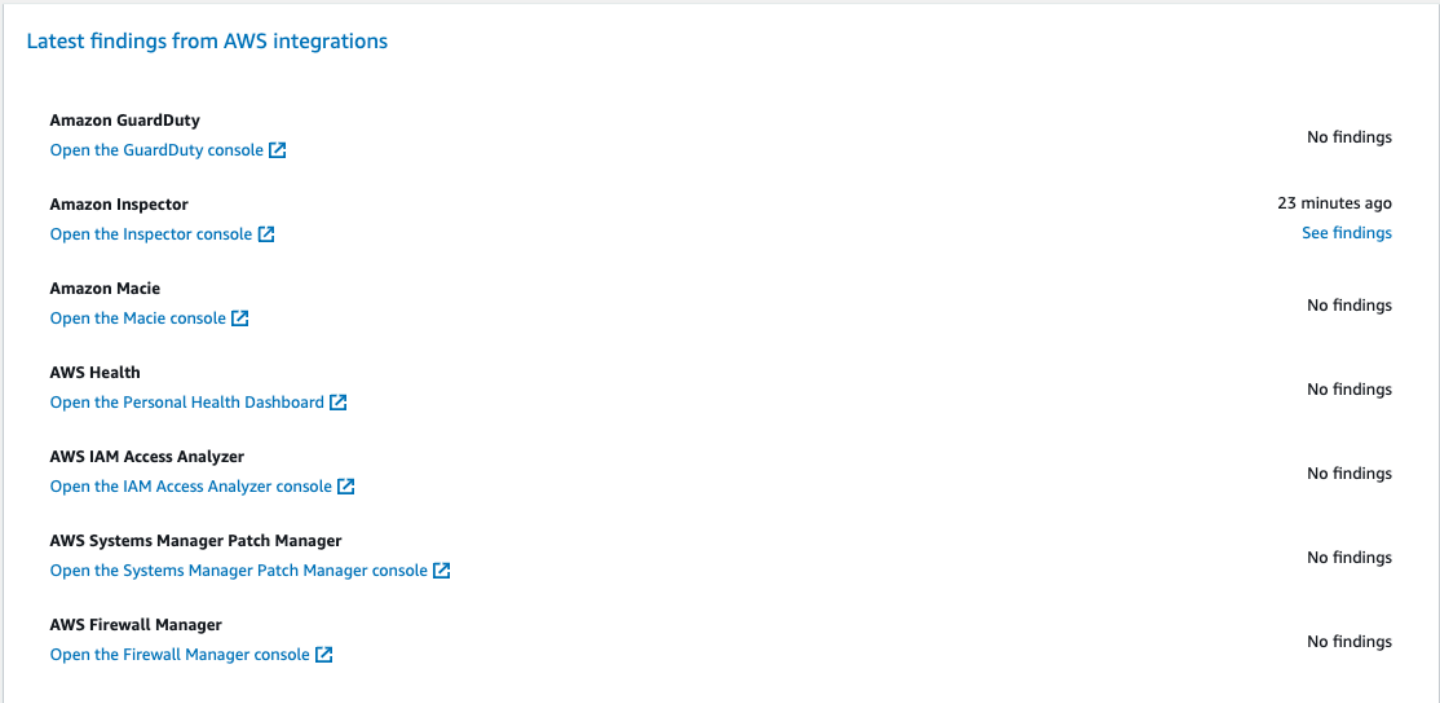
Transparenz der Lambda-Funktionsgrenzen mithilfe von Security Hub CSPM



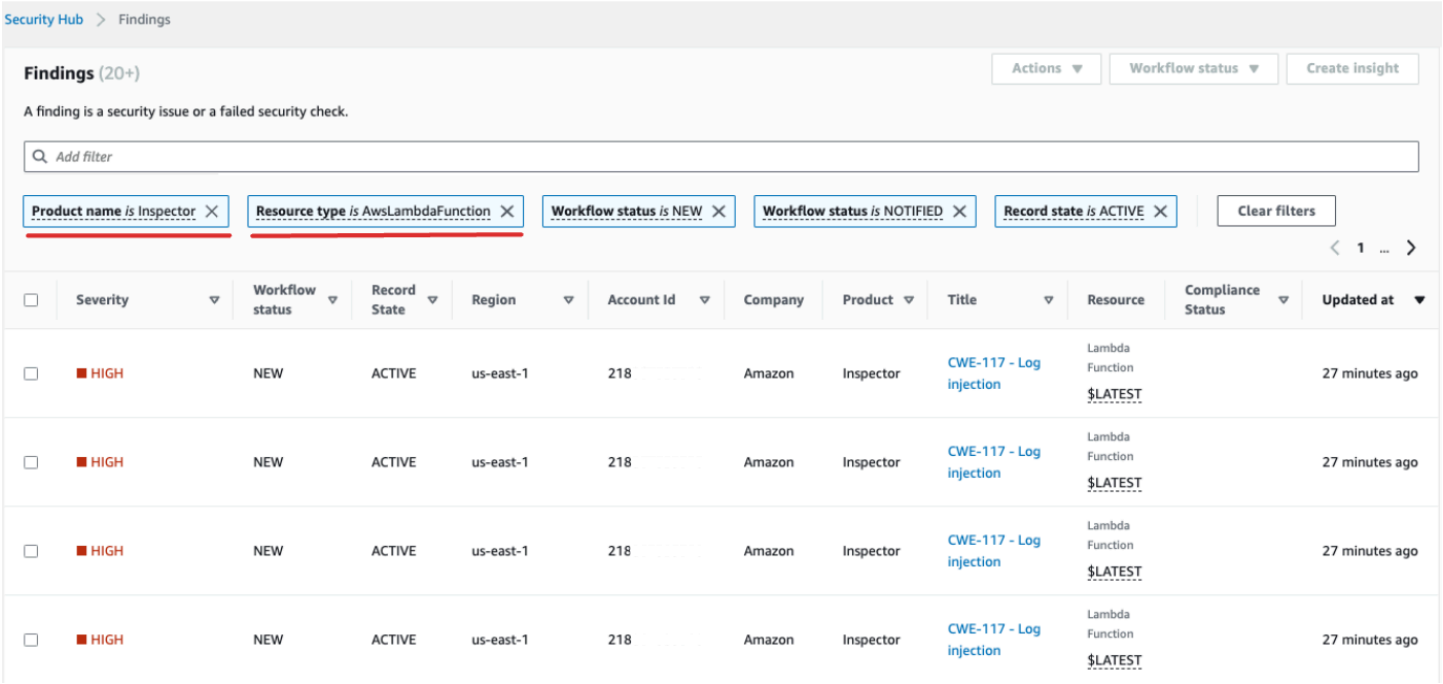
Um sicherzustellen, dass AWS Dienste wie Lambda sicher verwendet werden, AWS wurden die Foundational Security Best Practices v1.0.0 eingeführt. Diese Reihe von bewährten Methoden enthält klare Richtlinien für die Sicherung von Ressourcen und Daten in der AWS Umgebung und unterstreicht, wie wichtig es ist, ein solides Sicherheitsniveau aufrechtzuerhalten. Dies wird durch ein einheitliches Sicherheits- und Compliance-Center AWS Security Hub CSPM ergänzt. Es aggregiert,

organisiert und priorisiert Sicherheitserkenntnisse aus verschiedenen AWS Diensten wie Amazon Inspector und Amazon. AWS Identity and Access Management Access Analyzer GuardDuty

Wenn Sie Security Hub CSPM, Amazon Inspector und IAM Access Analyzer in Ihrem AWS Unternehmen GuardDuty aktiviert haben, aggregiert Security Hub CSPM automatisch die Ergebnisse dieser Services. Betrachten wir zum Beispiel Amazon Inspector. Mit Security Hub CSPM können Sie Code- und Paketschwachstellen in Lambda-Funktionen effizient identifizieren. Navigieren Sie in der Security Hub CSPM-Konsole zum unteren Abschnitt mit der Bezeichnung Aktuelle Erkenntnisse aus AWS Integrationen. Hier können Sie Ergebnisse aus verschiedenen integrierten Diensten einsehen und analysieren. AWS



Für Details klicken Sie in der zweiten Spalte auf den Link Befunde anzeigen. Daraufhin wird eine nach Produkten gefilterte Liste mit Ergebnissen angezeigt, z. B. Amazon Inspector. Um Ihre Suche auf Lambda-Funktionen zu beschränken, setzen Sie `ResourceType` auf `AwsLambdaFunction`. Nun werden Ergebnisse von Amazon Inspector zu Lambda-Funktionen angezeigt.



Denn GuardDuty Sie können verdächtige Muster im Netzwerkverkehr identifizieren. Solche Anomalien könnten auf die Existenz von potenziell bösartigem Code in Ihrer Lambda-Funktion hindeuten.

Mit IAM Access Analyzer können Sie Richtlinien überprüfen, insbesondere solche mit Bedingungsanweisungen, die externen Entitäten Funktionszugriff gewähren. Darüber hinaus bewertet IAM Access Analyzer die Berechtigungen, die bei der Verwendung des AddPermissionVorgangs in der Lambda-API zusammen mit einem festgelegt wurden. `EventSourceToken`

Behandlung von Beobachtbarkeitsbefunden

Angesichts der vielfältigen Konfigurationen, die für Lambda-Funktionen möglich sind, und ihrer unterschiedlichen Anforderungen ist eine standardisierte Automatisierungslösung für die Problembehebung möglicherweise nicht für jede Situation geeignet. Darüber hinaus werden Änderungen in den verschiedenen Umgebungen unterschiedlich implementiert. Wenn Sie auf eine Konfiguration stoßen, die nicht konform zu sein scheint, beachten Sie die folgenden Richtlinien:

1. Strategie zur Kennzeichnung

   Wir empfehlen die Implementierung einer umfassenden Kennzeichnungsstrategie. Jede Lambda-Funktion sollte mit folgenden Schlüsselinformationen gekennzeichnet sein:

   - Eigentümer: Die Person oder das Team, die für die Funktion verantwortlich ist.

- Umgebung: Produktion, Staging, Entwicklung oder Sandbox.
- Anwendung: Der breitere Kontext, zu dem diese Funktion gehört, falls zutreffend.

2. Kontakt zum Eigentümer

   Anstatt notwendige Änderungen zu automatisieren (wie die Anpassung der VPC-Konfiguration), sollten Sie sich an die Eigentümer nicht konformer Funktionen (siehe Eigentümer-Tag) wenden und ihnen genügend Zeit geben, um entweder:

   - nicht konforme Konfigurationen für Lambda-Funktionen anzupassen
   - eine Erklärung bereitzustellen und eine Ausnahme anzufordern oder die Compliance-Standards anzupassen

3. Fühen Sie eine Configuration Management Database (CMDB)

   Tags können zwar unmittelbar Kontext liefern, eine zentralisierte CMDB kann jedoch genauere Informationen liefern. Die Datenbank kann detaillierte Informationen über jede Lambda-Funktion, ihre Abhängigkeiten und andere wichtige Metadaten enthalten. Eine CMDB ist eine unschätzbare Ressource für Audits, Konformitätsprüfungen und die Identifizierung der Eigentümer einer Funktion.

Da sich Serverless-Infrastrukturen kontinuierlich weiterentwickeln, ist es wichtig, Überwachungsmaßnahmen proaktiv zu organisieren. Mit Tools wie AWS Config Security Hub CSPM und Amazon Inspector können potenzielle Anomalien oder nicht konforme Konfigurationen schnell identifiziert werden. Tools allein können jedoch keine vollständige Konformität oder optimale Konfigurationen gewährleisten. Es ist wichtig, diese Tools mit gut dokumentierten Prozessen und bewährten Verfahren zu kombinieren.

- Feedback-Schleife: Sobald Korrekturmaßnahmen ergriffen werden, sollten Sie sicherstellen, dass es eine Feedback-Schleife gibt. Dabei werden nicht konforme Ressourcen regelmäßig überprüft, um zu ermitteln, ob sie aktualisiert wurden oder immer noch dieselben Probleme haben.
- Dokumentation: Beobachtungen, Maßnahmen und alle gewährten Ausnahmen sollten immer dokumentiert werden. Eine ordnungsgemäße Dokumentation hilft nicht nur bei Audits, sondern trägt auch dazu bei, den Prozess zu verbessern, um die Compliance und Sicherheit dauerhaft zu stärken.
- Schulung und Sensibilisierung: Alle Beteiligten, insbesondere die Eigentümer der Lambda-Funktionen, sollten regelmäßig geschult und über bewährte Verfahren, Unternehmensrichtlinien und Compliance-Vorschriften informiert werden. Regelmäßige Workshops, Webinare oder

Schulungen können einen großen Beitrag dazu leisten, dass alle Beteiligten bezüglich Sicherheit und Compliance auf dem gleichen Stand sind.

Zusammenfassend lässt sich sagen, dass Tools und Technologien zwar die Erkennung und Meldung von potenziellen Problemen unterstützen, der Faktor Mensch –Verständnis, Kommunikation, Schulung und Dokumentation – aber weiterhin von zentraler Bedeutung ist. Beides zusammen ergibt eine leistungsstarke Kombination, um sicherzustellen, dass Ihre Lambda-Funktionen und die größere Infrastruktur konform, sicher und für Ihre Geschäftsanforderungen optimiert sind.

Compliance-Validierung für AWS Lambda

Die Auditoren Dritter bewerten die Sicherheit und die Compliance von AWS Lambda im Rahmen mehrerer AWS-Compliance-Programme. Hierzu zählen unter anderem SOC, PCI, FedRAMP und HIPAA.

Eine Liste der AWS-Services, die in den Geltungsbereich bestimmter Compliance-Programme fallen, finden Sie auf der Seite AWS-Services in Scope nach Compliance-Programm. Allgemeine Informationen finden Sie unter AWS-Compliance-Programme.

Die Auditberichte von Drittanbietern lassen sich mit herunterladen AWS Artifact. Weitere Informationen finden Sie unter Herunterladen von Berichten in AWS-Artifact.

Ihre Compliance-Verantwortung bei Verwendung von Lambda hängt von der Vertraulichkeit der Daten, den Compliance-Zielen des Unternehmens und den geltenden Gesetzen und Vorschriften ab. Sie können Governance-Kontrollen implementieren, um sicherzustellen, dass die Lambda-Funktionen Ihres Unternehmens Ihren Compliance-Anforderungen entsprechen. Weitere Informationen finden Sie unter Erstellen Sie eine Governance-Strategie für Lambda-Funktionen und -Layer.

Ausfallsicherheit in AWS Lambda

Im Zentrum der globalen AWS-Infrastruktur stehen die AWS-Regionen und Availability Zones. AWS Regionen stellen mehrere physisch getrennte und isolierte Availability Zones bereit, die mit Netzwerken mit geringer Latenz, hohem Durchsatz und hochredundanten Vernetzungen verbunden sind. Mithilfe von Availability Zones können Sie Anwendungen und Datenbanken erstellen und ausführen, die automatisch Failover zwischen Availability Zones ausführen, ohne dass es zu Unterbrechungen kommt. Availability Zones sind besser hoch verfügbar, fehlertoleranter und skalierbarer als herkömmliche Infrastrukturen mit einem oder mehreren Rechenzentren.

Weitere Informationen über AWS-Regionen und -Availability Zones finden Sie unter Globale AWS-Infrastruktur.

Zusätzlich zur globalen AWS-Infrastruktur stellt Lambda verschiedene Funktionen bereit, um Ihren Anforderungen in Bezug auf Ausfallsicherheit und Datensicherung zu erfüllen.

- Versioning – Sie können in Lambda Versioning verwenden, um Code und Konfiguration Ihrer Funktion während ihrer Entwicklung zu speichern. In Verbindung mit Aliasnamen können Sie Versioning verwenden, um blaue/grüne und fortlaufende Bereitstellungen ausfzuführen. Details hierzu finden Sie unter Versionen der Lambda-Funktion verwalten.

- Skalierung – Wenn Ihre Funktion während der Verarbeitung einer früheren Anforderung eine weitere Anforderung empfängt, startet Lambda eine weitere Instance Ihrer Funktion, um die höhere Last zu verarbeiten. Lambda wird automatisch skaliert, um 1.000 gleichzeitige Ausführungen pro Region zu verarbeiten, ein Kontingent, das bei Bedarf erhöht werden kann. Details hierzu finden Sie unter Verstehen der Skalierung von Lambda-Funktionen.
- Hohe Verfügbarkeit – Lambda führt Ihre Funktion in mehreren Availability Zones aus, um sicherzustellen, dass sie für die Verarbeitung von Ereignissen verfügbar ist, wenn der Service in einer Zone unterbrochen wird. Wenn Sie Ihre Funktion für die Herstellung von Verbindungen mit einer Virtual Private Cloud (VPC) in Ihrem Konto konfigurieren, geben Sie Subnetze in mehreren Availability Zones an, um eine hohe Verfügbarkeit sicherzustellen. Details hierzu finden Sie unter Lambda-Funktionen Zugriff auf Ressourcen in einer Amazon VPC gewähren.
- Reservierte Gleichzeitigkeit – Um sicherzustellen, dass Ihre Funktion stets skaliert werden kann, um zusätzliche Anforderungen zu verarbeiten, können Sie für sie Gleichzeitigkeit reservieren. Die Reservierung von Gleichzeitigkeit für eine Funktion stellt sicher, dass sie bis zu einer angegebenen Anzahl gleichzeitiger Aufrufe skaliert werden kann, diese Anzahl jedoch nicht überschreitet. Auf diese Weise wird sichergestellt, dass keine Anforderungen verloren gehen, da andere Funktionen der gesamte verfügbare Gleichzeitigkeit verbrauchen. Details hierzu finden Sie unter Konfigurieren reservierter Gleichzeitigkeit für eine Funktion.
- Wiederholversuche – Im Fall asynchroner Aufrufe und bestimmter, von anderen Diensten ausgelöster Aufrufe führt Lambda bei Fehlern automatisch Wiederholversuche mit Verzögerungen zwischen den einzelnen Wiederholversuchen aus. Für die Ausführung der Wiederholversuche sind andere Clients und AWS-Services, die Funktionen synchron aufrufen, verantwortlich. Details hierzu finden Sie unter Grundlegendes zum Wiederholungsverhalten in Lambda.
- Warteschlange für unzustellbare Nachrichten – Im Fall asynchroner Aufrufe können Sie Lambda für das Senden von Anforderungen an eine Warteschlange für unzustellbare Nachrichten konfigurieren, wenn Wiederholversuche fehlschlagen. Eine Warteschlange für unzustellbare Nachrichten ist ein Amazon-SNS-Thema oder eine Amazon-SQS-Warteschlange, die Ereignisse zur Fehlerbehebung oder erneuten Verarbeitung empfängt. Details hierzu finden Sie unter Hinzufügen einer Warteschlange für unzustellbare Nachrichten.

Sicherheit der Infrastruktur in AWS Lambda

Als verwalteter Service ist AWS Lambda durch die globalen Verfahren zur Gewährleistung der Netzwerksicherheit von AWS geschützt. Informationen zu AWS-Sicherheitsdiensten und wie AWS die Infrastruktur schützt, finden Sie unter AWSCloud-Sicherheit. Informationen zum Entwerfen Ihrer

AWS-Umgebung anhand der bewährten Methoden für die Infrastruktursicherheit finden Sie unter Infrastrukturschutz im Security Pillar AWS Well-Architected Framework.

Sie verwenden durch AWS veröffentlichte API-Aufrufe, um über das Netzwerk auf Lambda zuzugreifen. Kunden müssen Folgendes unterstützen:

- Transport Layer Security (TLS). Wir benötigen TLS 1.2 und empfehlen TLS 1.3.
- Verschlüsselungs-Suiten mit Perfect Forward Secrecy (PFS) wie DHE (Ephemeral Diffie-Hellman) oder ECDHE (Elliptic Curve Ephemeral Diffie-Hellman). Die meisten modernen Systeme wie Java 7 und höher unterstützen diese Modi.

Sicherung von Workloads mit öffentlichen Endpunkten

Für Workloads, die öffentlich zugänglich sind, bietet AWS eine Reihe von Features und Diensten, mit denen bestimmte Risiken gemindert werden können. Dieser Abschnitt behandelt die Authentifizierung und Autorisierung von Anwendungsbenutzern und den Schutz von API-Endpunkten.

Authentifizierung und Autorisierung

Authentifizierung bezieht sich auf Identität und Autorisierung bezieht sich auf Aktionen. Verwenden Sie die Authentifizierung, um zu kontrollieren, wer eine Lambda-Funktion aufrufen kann und verwenden Sie dann die Autorisierung, um zu kontrollieren, was sie tun können. Für viele Anwendungen ist IAM ausreichend, um beide Kontrollmechanismen zu verwalten.

Für Anwendungen mit externen Benutzern, wie z. B. Web- oder Mobilanwendungen, ist es üblich, JSON-Web-Tokens (JWTs) zur Verwaltung der Authentifizierung und Autorisierung zu verwenden. Im Gegensatz zur herkömmlichen, serverbasierten Passwortverwaltung werden JWTs bei jeder Anfrage vom Client übergeben. Sie sind eine kryptografisch sichere Methode, um Identität und Ansprüche anhand der vom Client übermittelten Daten zu überprüfen. Bei Lambda-basierten Anwendungen können Sie so jeden Aufruf an jeden API-Endpunkt sichern, ohne sich bei der Authentifizierung auf einen zentralen Server verlassen zu müssen.

Sie können JWTs mit Amazon Cognito implementieren, einem Benutzerverzeichnisdienst, der Registrierung, Authentifizierung, Kontowiederherstellung und andere gängige Kontoverwaltungsvorgänge abwickeln kann. Amplify Framework bietet Bibliotheken, die die Integration dieses Dienstes in Ihre Frontend-Anwendung vereinfachen. Sie können auch Partnerdienste von Drittanbietern wie Auth0 erwägen.

Angesichts der wichtigen Sicherheitsrolle eines Identitätsanbieterdienstes ist es wichtig, professionelle Tools zum Schutz Ihrer Anwendung zu verwenden. Es wird nicht empfohlen, eigene Dienste für die Authentifizierung oder Autorisierung zu schreiben. Schwachstellen in benutzerdefinierten Bibliotheken können erhebliche Auswirkungen auf die Sicherheit Ihres Workloads und seiner Daten haben.

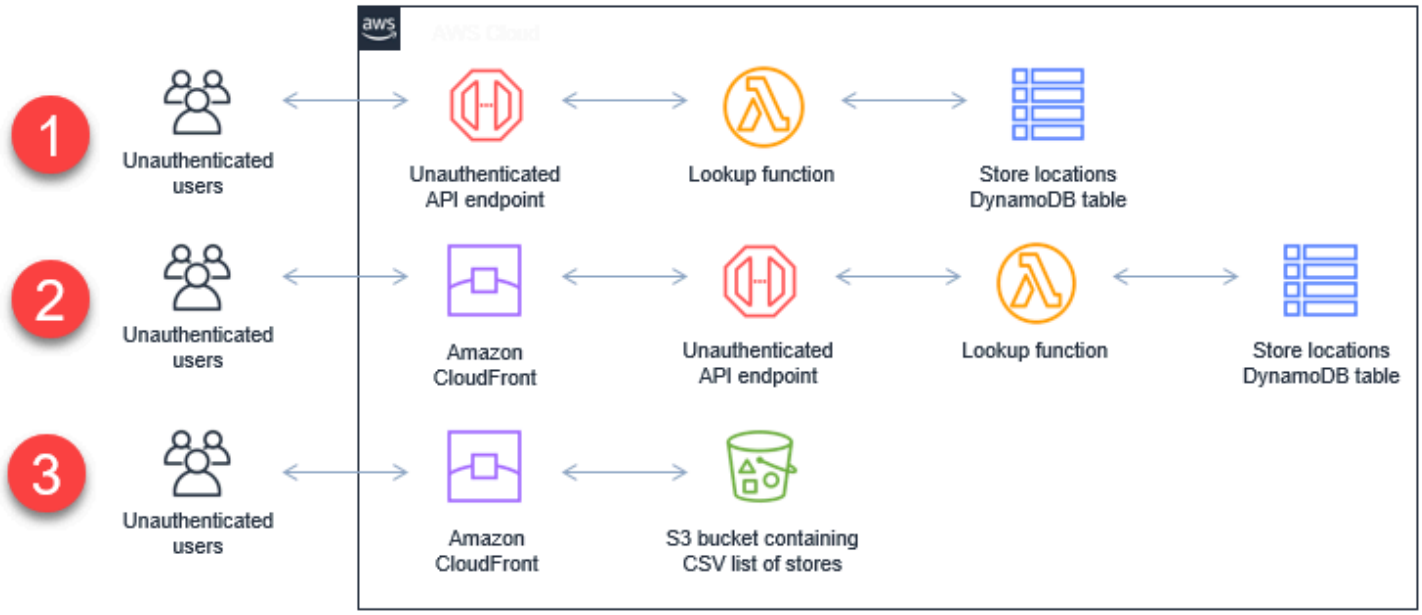
API-Endpunkte schützen

Für Serverless-Anwendungen ist die Verwendung von Amazon API Gateway die bevorzugte Methode, eine Backend-Anwendung öffentlich bereitzustellen. Dies kann Ihnen helfen, eine API vor böswilligen Benutzern oder Datenverkehrsspitzen zu schützen.

API Gateway bietet zwei Endpunkttypen für Serverless-Entwickler: REST-APIs und HTTP-APIs. Beide unterstützen die Autorisierung mit AWS Lambda, IAM oder Amazon Cognito. Bei der Verwendung von IAM oder Amazon Cognito werden eingehende Anfragen ausgewertet. Fehlt ein erforderliches Token oder enthält es eine ungültige Authentifizierung, wird die Anfrage abgelehnt. Diese Anfragen werden Ihnen nicht in Rechnung gestellt und sie werden auch nicht auf die Drosselungskontingente angerechnet.

Auf nicht authentifizierte API-Routen kann jeder im öffentlichen Internet zugreifen. Es wird daher empfohlen, die Verwendung nicht authentifizierter APIs einzuschränken. Wenn Sie nicht authentifizierte APIs verwenden müssen, ist es wichtig, diese vor häufigen Risiken wie Denial-of-Service-Angriffen (DoS) zu schützen. Die Anwendung von AWS WAF auf diese APIs kann dazu beitragen, Ihre Anwendung vor SQL-Injektion- und Cross-Site-Scripting (XSS)-Angriffen zu schützen. API Gateway implementiert auch Drosselung auf AWS-Kontoebene und auf Client-Ebene, wenn API-Schlüssel verwendet werden.

In vielen Fällen kann die Funktionalität einer nicht authentifizierten API mit einem alternativen Ansatz erreicht werden. Beispielsweise kann eine Webanwendung Benutzern, die nicht angemeldet sind, eine Liste von Einzelhandelsgeschäften von Kunden aus einer DynamoDB-Tabelle bereitstellen. Diese Anfrage kann von einer Frontend-Webanwendung oder von einer anderen Quelle stammen, die den URL-Endpunkt aufruft. In diesem Diagramm werden drei Lösungen verglichen:



1. Diese nicht authentifizierte API kann von jedem Benutzer im Internet aufgerufen werden. Bei einem Denial-of-Service-Angriff ist es möglich, die API-Drosselungsgrenzen, die Lambda-Gleichzeitigkeit oder die von DynamoDB bereitgestellte Lesekapazität für eine zugrunde liegende Tabelle auszuschöpfen.
2. Eine CloudFront-Verteilung vor dem API-Endpunkt mit einer geeigneten Time-to-Live (TTL)-Konfiguration würde den größten Teil des Datenverkehrs bei einem DoS-Angriff absorbieren, ohne die zugrunde liegende Lösung für das Abrufen der Daten zu ändern.
3. Alternativ könnte die CloudFront-Distribution für statische Daten, die sich selten ändern, die Daten aus einem Amazon-S3-Bucket bereitstellen.

Verwendung von Codesignatur zur Überprüfung der Codeintegrität mit Lambda

Die Codesignatur hilft sicherzustellen, dass nur vertrauenswürdiger Code in Ihren Lambda-Funktionen bereitgestellt wird. Mithilfe AWS Signer können Sie digital signierte Codepakete für Ihre Funktionen erstellen. Wenn Sie einer Funktion eine Code-Signaturkonfiguration hinzufügen, überprüft Lambda, ob alle neuen Codebereitstellungen von einer vertrauenswürdigen Quelle signiert sind. Da die Validierungsprüfungen von Codesignatur zum Zeitpunkt der Bereitstellung ausgeführt werden, gibt es keine Auswirkungen auf die Ausführung der Funktion.

> **Important**
>
> Codesignatur-Konfigurationen verhindern nur neue Bereitstellungen von unsigniertem Code. Wenn Sie einer vorhandenen Funktion mit unsigniertem Code eine Codesignatur-Konfiguration hinzufügen, wird dieser Code so lange ausgeführt, bis Sie ein neues Codepaket bereitstellen.

Wenn Sie die Codesignatur für eine Funktion aktivieren, müssen alle Ebenen, die Sie der Funktion hinzufügen, auch von einem zulässigen Signaturprofil signiert werden.

Es fallen keine zusätzlichen Gebühren für die Verwendung AWS Signer oder Codesignatur für an AWS Lambda.

Signaturvalidierung

Lambda führt die folgenden Validierungsprüfungen durch, wenn Sie ein signiertes Codepaket für Ihre Funktion bereitstellen:

1. Integrität: Überprüft, dass das Codepaket seit seiner Unterzeichnung nicht geändert wurde. Lambda vergleicht den Hash des Pakets mit dem Hash aus der Signatur.
2. Ablauf: Bestätigt, dass die Signatur des Codepakets nicht abgelaufen ist.
3. Diskrepanz: Bestätigt, dass das Codepaket mit einem der zulässigen Signaturprofil signiert ist.
4. Widerruf: Bestätigt, dass die Signatur des Codepakets nicht widerrufen wurde.

Wenn Sie eine Codesignaturkonfiguration erstellen, können Sie den UntrustedArtifactOnDeploymentParameter verwenden, um anzugeben, wie Lambda reagieren soll, wenn die Ablauf-, Nichtübereinstimmungs- oder Sperrprüfungen fehlschlagen. Sie können eine der folgenden Aktionen auswählen:

- `Warn`: Dies ist die Standardeinstellung. Lambda ermöglicht die Bereitstellung des Codepakets, gibt jedoch eine Warnung aus. Lambda gibt eine neue CloudWatch Amazon-Metrik (`SignatureValidationErrors`) aus und speichert die Warnung auch im CloudTrail Protokoll.
- `Enforce` – Lambda gibt eine Warnung aus (wie bei der `Warn`-Aktion) und blockiert die Bereitstellung des Codepakets.

Themen

- Erstellen von Codesignatur-Konfigurationen für Lambda
- Konfigurieren von IAM-Richtlinien für Lambda-Codesignaturkonfigurationen
- Verwendung von Tags in Codesignatur-Konfigurationen

Erstellen von Codesignatur-Konfigurationen für Lambda

Um die Codesignatur für eine Funktion zu aktivieren, erstellen Sie eine Codesignatur-Konfiguration und hängen sie an die Funktion an. Eine Codesignatur-Konfiguration definiert eine Liste der zulässigen Signaturprofile und die Richtlinienaktion, die durchgeführt werden soll, wenn eine der Validierungsprüfungen fehlschlägt.

> **Note**
>
> Funktionen, die als Container-Images definiert sind, unterstützen keine Codesignatur.

Abschnitte

- Voraussetzungen für die Konfiguration
- Erstellen von Codesignatur-Konfigurationen
- Aktivieren der Codesignatur für eine Funktion

Voraussetzungen für die Konfiguration

Bevor Sie die Codesignatur für eine Lambda-Funktion konfigurieren können, verwenden Sie AWS Signer, um Folgendes zu tun:

- Erstellen Sie mindestens ein Signaturprofil.
- Verwenden Sie ein Signaturprofil, um ein signiertes Codepaket für Ihre Funktion zu erstellen.

Erstellen von Codesignatur-Konfigurationen

Mit einer Codesignatur-Konfiguration wird eine Liste der zulässigen Signaturprofile und die Richtlinie für Signaturvalidierung definiert.

So erstellen Sie eine Codesignatur-Konfiguration (Konsole)

1. Öffnen Sie die Seite Codesignatur-Konfigurationen der Lambda-Konsole.

2. Wählen Sie Konfiguration erstellen.
3. Geben Sie unter Description (Beschreibung) einen aussagekräftigen Namen für die Konfiguration ein.
4. Fügen Sie der Konfiguration unter Signaturprofile bis zu 20 Signaturprofile hinzu.
   a. Wählen Sie für ARN der Signaturprofilversion den Amazon-Ressourcennamen (ARN) einer Profilversion aus oder geben Sie den ARN ein.
   b. Um ein zusätzliches Signaturprofil hinzuzufügen, wählen Sie Signaturprofil hinzufügen aus.
5. Wählen Sie unter Richtlinie zur Signaturvalidierung die Option Warnen oder Durchsetzen aus.
6. Wählen Sie Konfiguration erstellen.

Aktivieren der Codesignatur für eine Funktion

Um die Codesignatur für eine Funktion zu aktivieren, fügen Sie der Funktion eine Codesignatur-Konfiguration hinzu.

> **Important**
>
> Codesignatur-Konfigurationen verhindern nur neue Bereitstellungen von unsigniertem Code. Wenn Sie einer vorhandenen Funktion mit unsigniertem Code eine Codesignatur-Konfiguration hinzufügen, wird dieser Code so lange ausgeführt, bis Sie ein neues Codepaket bereitstellen.

So verknüpfen Sie eine Codesignatur-Konfiguration mit einer Funktion (Konsole)

1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.
2. Wählen Sie die Funktion aus, für die Sie die Codesignatur aktivieren möchten.
3. Öffnen Sie die Registerkarte Configuration (Konfiguration).
4. Scrollen Sie nach unten und wählen Sie Codesignatur aus.
5. Wählen Sie Bearbeiten aus.
6. Wählen Sie i n Codesignatur bearbeiten eine Codesignatur-Konfiguration für diese Funktion aus.
7. Wählen Sie Speichern.

Konfigurieren von IAM-Richtlinien für Lambda-Codesignaturkonfigurationen

Um einem Benutzer die Berechtigung zum Zugriff auf die Lambda-Codesignatur-API-Vorgänge zu erteilen, fügen Sie der Benutzerrichtlinie eine oder mehrere Richtlinienanweisungen bei. Weitere Informationen zu Benutzerrichtlinien finden Sie unter Identitätsbasierte IAM-Richtlinien für Lambda.

Die folgende Beispiel-Richtlinienanweisung gewährt die Berechtigung zum Erstellen, Aktualisieren und Abrufen von Codesignatur-Konfigurationen.

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
          "lambda:CreateCodeSigningConfig",
          "lambda:UpdateCodeSigningConfig",
          "lambda:GetCodeSigningConfig"
        ],
      "Resource": "*"
    }
  ]
}
```

Administratoren können den `CodeSigningConfigArn`-Bedingungsschlüssel verwenden, um die Codesignatur-Konfigurationen anzugeben, mit denen Entwickler Ihre Funktionen erstellen oder aktualisieren müssen.

Die folgende Beispielrichtlinienanweisung gewährt die Berechtigung zum Erstellen einer Funktion. Die Richtlinienanweisung enthält eine `lambda:CodeSigningConfigArn`-Bedingung, um die zulässige Konfiguration für die Codesignierung anzugeben. Lambda blockiert `CreateFunction` API-Anfragen, wenn der CodeSigningConfigArnParameter fehlt oder nicht mit dem Wert in der Bedingung übereinstimmt.

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"
        }
      }
    }
  ]
}
```

Verwendung von Tags in Codesignatur-Konfigurationen

Sie können Codesignaturkonfigurationen taggen, um Ihre Ressourcen zu organisieren und zu verwalten. Tags sind frei formulierte Schlüssel-Wert-Paare, die mit Ihren Ressourcen verknüpft sind und von AWS-Services unterstützt werden. Weitere Informationen zu Anwendungsfällen für Tags finden Sie unter Allgemeine Tagging-Strategien im Tagging AWS Resources and Tag Editor Guide.

Sie können die AWS Lambda API verwenden, um Tags anzuzeigen und zu aktualisieren. Sie können auch Tags anzeigen und aktualisieren, während Sie eine bestimmte Codesignatur-Konfiguration in der Lambda-Konsole verwalten.

Sections

- Erforderliche Berechtigungen zum Arbeiten mit Tags
- Verwendung von Tags mit der Lambda-Konsole
- Verwenden von Tags mit dem AWS CLI

Erforderliche Berechtigungen zum Arbeiten mit Tags

Um einer AWS Identity and Access Management -(IAM)-Identität (Benutzer, Gruppe oder Rolle) das Lesen oder Setzen von Tags auf einer Ressource zu ermöglichen, gewähren Sie ihr die entsprechenden Berechtigungen:

- lambda: ListTags —Wenn eine Ressource Tags enthält, gewähren Sie diese Berechtigung jedem, der sie aufrufen `ListTags` muss. Für Funktionen mit Tags ist diese Berechtigung auch für `GetFunction` erforderlich.
- lambda: TagResource —Erteilen Sie diese Berechtigung jedem, der bei create ein Tag aufrufen `TagResource` oder ausführen muss.

Optional können Sie erwägen, auch die lambda: UntagResource -Berechtigung zu erteilen, um `UntagResource` Aufrufe der Ressource zuzulassen.

Weitere Informationen finden Sie unter Identitätsbasierte IAM-Richtlinien für Lambda.

Verwendung von Tags mit der Lambda-Konsole

Sie können die Lambda-Konsole verwenden, um Codesignatur-Konfigurationen mit Tags zu erstellen, Tags zu vorhandenen Codesignatur-Konfigurationen hinzuzufügen und Codesignatur-Konfigurationen nach Tags zu filtern.

So fügen Sie ein Tag hinzu, wenn Sie eine Codesignatur-Konfiguration erstellen

1. Öffnen Sie Codesignatur-Konfigurationen der Lambda-Konsole.
2. Wählen Sie in der Kopfzeile des Inhaltsfensters die Option Konfiguration erstellen aus.
3. Richten Sie im Abschnitt oben im Inhaltsbereich Ihre Codesignatur-Konfiguration ein. Weitere Informationen zum Konfigurieren von Codesignatur-Konfigurationen finden Sie unter the section called “Codesignatur”.
4. Klicken Sie im Abschnitt Tags auf Neuen Tag hinzufügen.
5. Geben Sie im Feld Schlüssel Ihren Tag-Schlüssel ein. Informationen zu Tagging-Einschränkungen finden Sie unter Beschränkungen und Anforderungen für die Benennung von Tags im Tagging AWS Resources and Tag Editor Guide.
6. Wählen Sie Create configuration (Konfiguration erstellen).

So fügen Sie einer vorhandenen Codesignaturkonfiguration ein Tag hinzu

1. Öffnen Sie Codesignatur-Konfigurationen der Lambda-Konsole.
2. Wählen Sie den Namen der Codesignatur-Konfiguration aus.
3. Wählen Sie auf den Registerkarten unter dem Detailbereich die Option Tags aus.
4. Wählen Sie Tags verwalten aus.
5. Wählen Sie Neues Tag hinzufügen aus.
6. Geben Sie im Feld Schlüssel Ihren Tag-Schlüssel ein. Informationen zu Einschränkungen beim Taggen finden Sie unter Beschränkungen und Anforderungen für die Benennung von Tags im Leitfaden für AWS Tag-Ressourcen und Tag-Editor.
7. Wählen Sie Speichern.

So filtern Sie Codesignatur-Konfigurationen nach Tags

1. Öffnen Sie Codesignatur-Konfigurationen der Lambda-Konsole.
2. Wählen Sie das Suchfeld.
3. Wählen Sie in der Dropdown-Liste Ihr Tag unter der Tag-Unterüberschrift aus.
4. Wählen Sie Verwenden: „Tag-Name“, um alle Codesignaturkonfigurationen zu sehen, die mit dieser Taste gekennzeichnet sind oder wählen Sie einen Operator, um weiter nach Werten zu filtern.
5. Wählen Sie Ihren Tag-Wert aus, um nach einer Kombination aus Tag-Schlüssel und -Wert zu filtern.

Das Suchfeld unterstützt auch die Suche nach Tag-Schlüsseln. Geben Sie den Namen eines Schlüssels ein, um ihn in der Liste zu finden.

Verwenden von Tags mit dem AWS CLI

Mit der Lambda-API können Sie Tags zu vorhandenen Lambda-Ressourcen hinzufügen und entfernen, einschließlich Codesignaturkonfigurationen. Sie können auch Tags hinzufügen, wenn Sie eine Codesignaturkonfiguration erstellen, die es Ihnen ermöglicht, eine Ressource während ihres gesamten Lebenszyklus zu kennzeichnen.

Aktualisieren von Tags mit dem Lambda-Tag APIs

Sie können Tags für unterstützte Lambda-Ressourcen über die TagResourceund UntagResourceAPI-Operationen hinzufügen und entfernen.

Sie können diese Operationen mit der Taste AWS CLI aufrufen. Um Tags einer vorhandenen Ressource hinzuzufügen, verwenden Sie den folgenden `tag-resource`-Befehl. In diesem Beispiel werden zwei Tags hinzugefügt, eines mit dem Schlüssel *`Department`* und eines mit dem Schlüssel*`CostCenter`*.

```
aws lambda tag-resource \
--resource arn:aws:lambda:us-east-2:123456789012:resource-type:my-resource \
--tags Department=Marketing,CostCenter=1234ABCD
```

Mit dem Befehl `untag-resource` können Sie Tags entfernen. In diesem Beispiel wird das Tag mit dem Schlüssel entfernt*`Department`*.

```
aws lambda untag-resource --resource arn:aws:lambda:us-east-1:123456789012:resource-type:resource-identifier \
--tag-keys Department
```

Hinzufügen von Tags beim Erstellen einer Codesignatur-Konfiguration

Verwenden Sie den CreateCodeSigningConfigAPI-Vorgang, um eine neue Lambda-Codesignaturkonfiguration mit Tags zu erstellen. Geben Sie den Parameter `Tags` an. Sie können diesen Vorgang mit dem `create-code-signing-config` AWS CLI Befehl und der `--tags` Option aufrufen. Weitere Informationen zum CLI-Befehl finden Sie create-code-signing-configin der AWS CLI Befehlsreferenz.

Vergewissern Sie sich vor der Verwendung des `Tags`-Parameters mit `CreateCodeSigningConfig`, dass Ihre Rolle neben den üblichen Berechtigungen, die für diesen Vorgang erforderlich sind, auch die Berechtigung hat, Ressourcen zu taggen. Weitere Informationen zu den Berechtigungen für das Tagging finden Sie unter the section called “Erforderliche Berechtigungen zum Arbeiten mit Tags”.

Tags mit dem Lambda-Tag anzeigen APIs

Um die Tags anzuzeigen, die auf eine bestimmte Lambda-Ressource angewendet werden, verwenden Sie die `ListTags`-API-Operation. Weitere Informationen finden Sie unter ListTags.

Sie können diesen Vorgang mit dem `list-tags` AWS CLI Befehl aufrufen, indem Sie einen ARN (Amazon Resource Name) angeben.

```
aws lambda list-tags --resource arn:aws:lambda:us-east-1:123456789012:resource-type:resource-identifier
```

Filtern von Ressourcen nach Tag

Sie können den AWS Resource Groups Tagging API GetResourcesAPI-Vorgang verwenden, um Ihre Ressourcen nach Tags zu filtern. Die `GetResources`-Operation empfängt bis zu 10 Filter, wobei jeder Filter einen Tag-Schlüssel und bis zu 10 Tag-Werte enthält. Sie stellen `GetResources` mit einem `ResourceType` zur Verfügung, um nach bestimmten Ressourcentypen zu filtern.

Sie können diesen Vorgang mit dem `get-resources` AWS CLI Befehl aufrufen. Beispiele für die Verwendung von `get-resources` finden Sie unter get-resources in der AWS -CLI-Befehlsreferenz.

Überwachung, Debbugging und Fehlerbehebung bei Lambda-Funktionen

AWS Lambda lässt sich in andere AWS-Services integrieren, die Ihnen helfen, Ihre Lambda-Funktionen zu überwachen und Fehler an ihnen zu beheben. Lambda überwacht automatisch Lambda-Funktionen in Ihrem Namen und meldet Metriken über Amazon CloudWatch. Damit Sie Ihren Code während der Ausführung überwachen können, verfolgt Lambda automatisch die Anzahl der Anfragen, die Dauer des Aufrufs pro Anfrage und die Anzahl der Anfragen, die zu einem Fehler führen.

Sie können andere AWS-Services verwenden, um Probleme mit Ihren Lambda-Funktionen zu beheben. In diesem Abschnitt wird beschrieben, wie Sie diese AWS-Services verwenden, um Ihre Lambda-Funktionen und -Anwendungen zu überwachen, zu verfolgen, zu debuggen und zu beheben. Einzelheiten zur Funktionsprotokollierung und zu Fehlern in jeder Laufzeit finden Sie in den einzelnen Laufzeitabschnitten.

Abschnitte

- Preisgestaltung
- Verwendung von CloudWatch-Metriken mit Lambda
- Arbeiten mit Lambda-Funktionsprotokollen
- Protokollieren von AWS Lambda-API-Aufrufen mit AWS CloudTrail
- Visualisieren Sie Lambda-Funktionsaufrufe mit AWS X-Ray
- Überwachen Sie die Funktionsleistung mit Amazon CloudWatch Lambda Insights
- Überwachen von Lambda-Anwendungen
- Überwachen der Anwendungsleistung mit Amazon CloudWatch Application Signals
- Remote-Debugging von Lambda-Funktionen mit Visual Studio Code

Preisgestaltung

CloudWatch verfügt über unbeschränktes kostenloses Kontingent. Über den Schwellenwert für das kostenlose Kontingent hinaus berechnet CloudWatch Gebühren für Metriken, Dashboards, Alarme, Protokolle und Erkenntnisse. Weitere Informationen hierzu finden Sie unter Amazon CloudWatch – Preise.

Verwendung von CloudWatch-Metriken mit Lambda

Wenn Ihre AWS Lambda-Funktion die Verarbeitung eines Ereignisses beendet hat, sendet Lambda automatisch Metriken über den Aufruf an Amazon CloudWatch. Sie müssen Ihrer Ausführungsrolle keine zusätzlichen Berechtigungen erteilen, um Funktionsmetriken zu erhalten, und für diese Metriken fallen keine zusätzlichen Gebühren an.

Es gibt viele Arten von Metriken, die mit Lambda-Funktionen verknüpft sind. Dazu gehören Aufrufmetriken, Leistungsmetriken, Gleichzeitigkeitsmetriken, Metriken für asynchrone Aufrufe und Metriken für die Zuordnung von Ereignisquellen. Weitere Informationen finden Sie unter the section called “Metriktypen”.

In der CloudWatch-Konsole können Sie diese Metriken anzeigen und daraus Diagramme und Dashboards erstellen. Sie können auch Alarme so einstellen, dass sie auf Änderungen bei Auslastung, Leistung oder Fehlerraten reagieren. Lambda sendet Metrikdaten an CloudWatch in Intervallen von 1 Minute. Für einen direkteren Einblick in Ihre Lambda-Funktion können Sie hochauflösende benutzerdefinierte Metriken erstellen. Für benutzerdefinierte Metriken und CloudWatch-Alarme fallen Gebühren an. Weitere Informationen hierzu finden Sie unter Amazon CloudWatch – Preise.

Anzeigen von Metriken für Lambda-Funktionen

Verwenden Sie die CloudWatch-Konsole, um Metriken für Ihre Lambda-Funktionen anzuzeigen. In der Konsole können Sie Funktionsmetriken nach Funktionsname, Alias, Version oder UUID der Zuordnung von Ereignisquellen filtern und sortieren.

So zeigen Sie Metriken in der CloudWatch-Konsole an

1. Öffnen Sie die Seite Metrics (Metriken) (`AWS/Lambda`-Namespace) in der CloudWatch-Konsole.
2. Wählen Sie auf der Registerkarte Durchsuchen unter Metriken eine der folgenden Dimensionen aus:

   - Nach Funktionsname (`FunctionName`) – Zeigen Sie Aggregatmetriken für alle Versionen und Aliase einer Funktion anzeigen.
   - Nach Ressource (`Resource`) – Zeigen Sie Metriken für eine Version oder den Alias einer Funktion an.

- Nach ausgeführter Version (`ExecutedVersion`) – Zeigen Sie Metriken für eine Kombination aus Alias und Version an. Verwenden Sie die Dimension `ExecutedVersion`, um Fehlerraten für zwei Versionen einer Funktion zu vergleichen, die beide Ziele eines gewichteten Alias sind.
- Nach Zuordnung von Ereignisquellen UUID (`EventSourceMappingUUID`) — Metriken für eine Zuordnung von Ereignisquellen anzeigen.
- Über alle Funktionen (keine) – Zeigen Sie aggregierte Metriken für alle Funktionen in der aktuellen AWS-Region an.

3. Wählen Sie eine Metrik. Die Metrik sollte automatisch im visuellen Diagramm sowie auf der Registerkarte Graphische Metriken angezeigt werden.

Standardmäßig verwenden Diagramme die `Sum`-Statistik für alle Metriken. Um eine andere Statistik auszuwählen und das Diagramm anzupassen, verwenden Sie die Optionen auf der Registerkarte Graphed metrics (Graphierte Metriken).

> **Note**
>
> Der Zeitstempel einer Metrik gibt an, wann die Funktion aufgerufen wurde. Abhängig von der Dauer des Aufrufs kann es einige Minuten dauern, bis die Metrik ausgegeben wird. Wenn Ihre Funktion beispielsweise ein Timeout von 10 Minuten aufweist, suchen Sie dann über 10 Minuten in der Vergangenheit hinaus nach genauen Metriken.

Weitere Informationen über CloudWatch finden Sie im Amazon-CloudWatch-Benutzerhandbuch.

Arten von Metriken für Lambda-Funktionen

In diesem Abschnitt werden die Arten von Lambda-Metriken beschrieben, die in der CloudWatch-Konsole verfügbar sind.

Themen

- Aufrufmetriken
- Bereitstellungsmetriken
- Leistungsmetriken
- Gleichzeitigkeitsmetriken
- Metriken asynchroner Aufrufe
- Metriken zur Zuordnung von Ereignisquellen

Aufrufmetriken

Aufrufmetriken sind binäre Indikatoren für das Ergebnis eines Lambda-Funktionsaufrufs. Betrachten Sie diese Metriken mit der Statistik `Sum`. Wenn die Funktion beispielsweise einen Fehler zurückgibt, sendet Lambda die `Errors`-Metrik mit dem Wert 1. Um die Anzahl der Funktionsfehler zu erhalten, die jede Minute aufgetreten sind, lassen Sie sich die `Sum` der `Errors`-Metrik mit einem Zeitraum von 1 Minute anzeigen.

- `Invocations` – Die Häufigkeit, mit der Ihr Funktionscode aufgerufen wird, einschließlich erfolgreicher Aufrufe und Aufrufe, die zu einem Funktionsfehler führen. Aufrufe werden nicht aufgezeichnet, wenn die Aufrufanforderung gedrosselt wird oder anderweitig zu einem Aufruffehler führt. Der Wert von `Invocations` entspricht der Anzahl der fakturierten Anforderungen.
- `Errors` – Die Anzahl der Aufrufe, die zu einem Funktionsfehler führen. Funktionsfehler schließen Ausnahmen ein, die Ihr Code ausgibt, und Ausnahmen, die die Lambda-Laufzeit ausgibt. Die Laufzeit gibt Fehler für Probleme wie Timeouts und Konfigurationsfehler zurück. Um die Fehlerquote zu berechnen, teilen Sie den Wert von `Errors` durch den Wert von `Invocations`. Beachten Sie, dass der Zeitstempel für eine Fehlermetrik widerspiegelt, wann die Funktion aufgerufen wurde, und nicht, wann der Fehler aufgetreten ist.
- `DeadLetterErrors` – Bei asynchronen Aufrufen die Häufigkeit, mit der Lambda versucht, ein Ereignis an eine Warteschlange für unzustellbare Nachrichten zu senden, was jedoch fehlschlägt. Zustellungsfehler können aufgrund von falsch eingestellten Ressourcen oder Größenbeschränkungen auftreten.
- `DestinationDeliveryFailures` – Bei asynchronen Aufrufen und unterstützten Zuordnung von Ereignisquellen die Häufigkeit, mit der Lambda versucht, ein Ereignis an ein Ziel zu senden, dies jedoch fehlschlägt. Für die Zuordnung von Ereignisquellen unterstützt Lambda Ziele für Streamquellen (DynamoDB und Kinesis). Zustellungsfehler können aufgrund von Berechtigungsfehlern, falsch konfigurierten Ressourcen oder Größenbeschränkungen auftreten. Fehler können auch dann passieren, wenn Ihre Konfiguration einen nicht unterstützten Zieltyp enthält, z. B. eine Amazon-SQS-FIFO-Warteschlange oder ein Amazon-SNS-FIFO-Thema.
- `Throttles` – Die Anzahl der Aufruf-Anforderungen, die gedrosselt werden. Wenn alle Funktions-Instances Anforderungen verarbeiten und keine Nebenläufigkeit zum Hochskalieren verfügbar ist, lehnt Lambda zusätzliche Anforderungen mit einem `TooManyRequestsException`-Fehler ab. Gedrosselte Anforderungen und andere Aufruffehler zählen nicht als `Invocations` oder `Errors`.
- `OversizedRecordCount`: Bei Amazon-DocumentDB-Ereignisquellen die Anzahl von Ereignissen mit einer Größe von mehr als 6 MB, die Ihre Funktion aus Ihrem Change-Stream empfängt. Lambda verwirft die Nachricht und gibt diese Metrik aus.

- `ProvisionedConcurrencyInvocations` – Gibt an, wie oft Ihr Funktionscode für bereitgestellte Gleichzeitigkeit aufgerufen wird.
- `ProvisionedConcurrencySpilloverInvocations` – Gibt an, wie oft Ihr Funktionscode für Standard-Gleichzeitigkeit aufgerufen wird, wenn die gesamte bereitgestellte Gleichzeitigkeit verwendet wird.
- `RecursiveInvocationsDropped` – Gibt an, wie oft Lambda den Aufruf Ihrer Funktion beendet hat, weil festgestellt wurde, dass Ihre Funktion Teil einer unendlichen rekursiven Schleife ist. Die Erkennung rekursiver Schleifen überwacht, wie oft eine Funktion als Teil einer Kette von Anforderungen aufgerufen wird, indem Metadaten verfolgt werden, die von unterstützten AWS-SDKs hinzugefügt wurden. Standardmäßig lässt Lambda den nächsten Aufruf aus, wenn Ihre Funktion als Teil einer Kette von Anforderungen etwa 16 Mal aufgerufen wird. Wenn Sie die rekursive Schleifenerkennung deaktivieren, wird diese Metrik nicht ausgegeben. Weitere Informationen über dieses Feature finden Sie unter Verwenden Sie die rekursive Lambda-Schleifenerkennung, um Endlosschleifen zu verhindern.

Bereitstellungsmetriken

Bereitstellungsmetriken stellen Informationen über Ereignisse bei der Bereitstellung von Lambda-Funktionen und zugehörige Validierungsprozesse bereit.

- `SignatureValidationErrors` – Gibt an, wie oft bei der Bereitstellung eines Codepakets Fehler bei der Signaturvalidierung aufgetreten sind, obwohl die Codesignatur-Konfigurationsrichtlinie auf `Warn` festgelegt ist. Diese Metrik wird ausgegeben, wenn die Prüfung auf Ablauf, Diskrepanz oder Widerruf scheitert, die Bereitstellung jedoch aufgrund der `Warn`-Richtlinieneinstellung weiterhin zulässig ist. Weitere Informationen zur Codesignatur finden Sie unter Verwendung von Codesignatur zur Überprüfung der Codeintegrität mit Lambda.

Leistungsmetriken

Performance-Metriken stellen Performance-Details zu einem einzelnen Funktionsaufruf bereit. Die Metrik `Duration` gibt beispielsweise die Zeit in Millisekunden an, die Ihre Funktion auf die Verarbeitung eines Ereignisses aufwendet. Um einen Eindruck davon zu erhalten, wie schnell Ihre Funktion Ereignisse verarbeitet, können Sie sich diese Metriken mit der Statistik `Average` oder `Max` anzeigen lassen.

- `Duration` – Die Zeit, die Ihr Funktionscode auf die Verarbeitung eines Ereignisses aufwendet. Die fakturierte Dauer für einen Aufruf ist der Wert von `Duration`, aufgerundet auf die nächste Millisekunde. `Duration` beinhaltet nicht die Kaltstartzeit.
- `PostRuntimeExtensionsDuration` – Die kumulative Zeit, die die Laufzeit nach Abschluss des Funktionscodes mit Erweiterungen verbringt.
- `IteratorAge` – Bei DynamoDB-, Kinesis- und Amazon-DocumentDB-Ereignisquellen das Alter des letzten Datensatzes im Ereignis in Millisekunden. Diese Matrik misst die Zeit zwischen dem Zeitpunkt, zu dem ein Stream den Datensatz empfängt, und dem Zeitpunkt, zu dem die Zuordnung von Ereignisquellen das Ereignis an die Funktion sendet.
- `OffsetLag` – Bei selbstverwalteten Apache-Kafka- und Amazon Managed Streaming für Apache Kafka (Amazon MSK)-Ereignisquellen, die Versatzdifferenz zwischen dem letzten Datensatz, der in ein Thema geschrieben wurde, und dem letzten Datensatz, den die Konsumentengruppe Ihrer Funktion verarbeitet hat. Ein Kafka-Thema kann zwar mehrere Partitionen haben, diese Metrik misst die Offset-Verzögerung jedoch auf Themenebene.

`Duration` unterstützt auch Perzentilstatistiken (p). Verwenden Sie Perzentile, um Ausreißerwerte auszuschließen, die `Averages`- und `Maximum`-Statistiken verzerren. Die p95-Statistik zeigt beispielsweise die maximale Dauer von 95 % der Aufrufe, ohne die langsamsten 5 %. Weitere Informationen finden Sie unter Perzentile im Benutzerhandbuch zu Amazon CloudWatch.

Gleichzeitigkeitsmetriken

Lambda meldet Gleichzeitigkeitsmetriken als aggregierte Anzahl der Instances, die Ereignisse in einer Funktion, Version, Alias oder AWS-Region verarbeiten. Zeigen Sie diese Metriken mit der Statistik `Max` an, um zu sehen, wie nah Sie an den Grenzwerten für Gleichzeitigkeit sind.

- `ConcurrentExecutions` – Die Anzahl der Funktionsinstances, die Ereignisse verarbeiten. Wenn diese Zahl das Kontingent für gleichzeitige Ausführungen für die Region oder das Limit reservierter Gleichzeitigkeit für die Funktion erreicht, drosselt Lambda weitere Aufrufanforderungen.
- `ProvisionedConcurrentExecutions` – Die Anzahl der Funktions-Instances, die Ereignisse für bereitgestellte Gleichzeitigkeit verarbeiten. Für jeden Aufruf eines Alias oder einer Version mit Provisioned Concurrency gibt Lambda die aktuelle Anzahl aus. Wenn Ihre Funktion inaktiv ist oder keine Anforderungen empfängt, gibt Lambda diese Metrik nicht aus.
- `ProvisionedConcurrencyUtilization` – Für eine Version oder einen Alias der Wert von `ProvisionedConcurrentExecutions` geteilt durch die Gesamtmenge der konfigurierten Gleichzeitigkeit. Wenn Sie beispielsweise für Ihre Funktion eine bereitgestellte Gleichzeitigkeit

von 10 konfigurieren und Ihr `ProvisionedConcurrentExecutions` 7 ist, dann ist Ihr `ProvisionedConcurrencyUtilization` 0,7.

Wenn Ihre Funktion inaktiv ist oder keine Anforderungen empfängt, gibt Lambda diese Metrik nicht aus, da sie auf `ProvisionedConcurrentExecutions` basiert. Beachten Sie dies, wenn Sie `ProvisionedConcurrencyUtilization` als Grundlage für CloudWatch-Alarme verwenden.

- `UnreservedConcurrentExecutions` – Für eine Region die Anzahl der Ereignisse, die Funktionen ohne reservierte Nebenläufigkeit verarbeiten.
- `ClaimedAccountConcurrency` – Mit Bezug auf eine Region die Menge an Gleichzeitigkeit, die für On-Demand-Aufrufe nicht verfügbar ist. `ClaimedAccountConcurrency` entspricht `UnreservedConcurrentExecutions` plus dem Betrag der zugewiesenen Gleichzeitigkeit (d. h. der gesamten reservierten Gleichzeitigkeit plus der gesamten bereitgestellten Gleichzeitigkeit). Weitere Informationen finden Sie unter Die Metrik `ClaimedAccountConcurrency`.

Metriken asynchroner Aufrufe

Metriken asynchroner Aufrufe liefern Details zu asynchronen Aufrufen aus Ereignisquellen und direkten Aufrufen. Sie können Schwellenwerte und Alarme festlegen, um Sie über bestimmte Änderungen zu informieren. Zum Beispiel, wenn die Anzahl der Ereignisse, die für die Verarbeitung in die Warteschlange gestellt werden, ungewollt ansteigt (`AsyncEventsReceived`). Oder wenn ein Ereignis lange auf seine Verarbeitung gewartet hat (`AsyncEventAge`).

- `AsyncEventsReceived` – Die Anzahl der Ereignisse, die Lambda erfolgreich zur Verarbeitung in die Warteschlange stellt. Diese Metrik bietet einen Einblick in die Anzahl der Ereignisse, die eine Lambda-Funktion empfängt. Überwachen Sie diese Metrik und stellen Sie Alarme für Schwellenwerte ein, um nach Problemen zu suchen. Zum Beispiel, um eine unerwünschte Anzahl von Ereignissen zu erkennen, die an Lambda gesendet werden, und um schnell Probleme zu diagnostizieren, die auf falsche Auslöser- oder Funktionskonfigurationen zurückzuführen sind. Diskrepanzen zwischen `AsyncEventsReceived` und `Invocations` können auf eine unterschiedliche Verarbeitung, verworfene Ereignisse oder einen möglichen Rückstand in der Warteschlange hinweisen.
- `AsyncEventAge` – Die Zeit zwischen dem erfolgreichen Einstellen des Ereignisses in die Warteschlange durch Lambda und dem Aufruf der Funktion. Der Wert dieser Metrik steigt, wenn Ereignisse aufgrund von Aufruffehlern oder Drosselung erneut versucht werden. Überwachen Sie diese Metrik und legen Sie Alarme für Schwellenwerte für verschiedene Statistiken fest, wenn sich eine Warteschlange immer länger wird. Um einen Anstieg dieser Metrik zu beheben, sehen Sie

sich die `Errors`-Metrik an, um Funktionsfehler zu identifizieren, und die `Throttles`-Metrik, um Gleichzeitigkeitsprobleme zu identifizieren.

- `AsyncEventsDropped` – Die Anzahl der Ereignisse, die verworfen werden, ohne die Funktion erfolgreich auszuführen. Wenn Sie eine Warteschlange für unzustellbare Nachrichten oder ein `OnFailure`-Ziel konfigurieren, werden Ereignisse dorthin gesendet, bevor sie gelöscht werden. Ereignisse werden aus verschiedenen Gründen verworfen. Beispielsweise können Ereignisse das maximale Ereignisalter überschreiten oder die maximale Anzahl von Wiederholungsversuchen ausschöpfen, oder die reservierte Gleichzeitigkeit ist möglicherweise auf 0 gesetzt. Um herauszufinden, warum Ereignisse verworfen werden, sehen Sie sich die `Errors`-Metrik an, um Funktionsfehler zu identifizieren, und die `Throttles`-Metrik, um Gleichzeitigkeitsprobleme zu identifizieren.

Metriken zur Zuordnung von Ereignisquellen

Metriken zur Zuordnung von Ereignisquellen bieten Einblicke in das Verarbeitungsverhalten Ihrer Zuordnung von Ereignisquellen. Mithilfe dieser Metriken können Sie den Fluss und den Status von Ereignissen überwachen, einschließlich der Ereignisse, die von Ihrer Zuordnung von Ereignisquellen erfolgreich verarbeitet, gefiltert oder gelöscht wurden.

Sie müssen sich anmelden, um Metriken zu erhalten, die sich auf Zählungen beziehen (`PolledEventCount`, `FilteredOutEventCount`, `InvokedEventCount`, `FailedInvokeEventCount`, `DroppedEventCount`, `OnFailureDestinationDeliveredEventCount`, und `DeletedEventCount`). Um sich anzumelden, können Sie die Konsole oder die Lambda-API verwenden.

So aktivieren Sie Metriken oder eine Zuordnung von Ereignisquellen (Konsole)

1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.
2. Wählen Sie die Funktion, für die Sie Metriken aktivieren möchten.
3. Wählen Sie Konfiguration und anschließend Auslöser aus.
4. Wählen Sie die Zuordnung von Ereignisquellen, für die Sie Metriken aktivieren möchten und wählen Sie dann Bearbeiten.
5. Wählen Sie unter Konfiguration der Zuordnung von Ereignisquellen die Option Metriken aktivieren aus.
6. Wählen Sie Speichern.

Alternativ können Sie Metriken für Ihre Zuordnung von Ereignisquellen programmgesteuert aktivieren, indem Sie das EventSourceMappingMetricsConfig-Objekt in Ihrer EventSourceMappingConfiguration verwenden. Der folgende CLI-Befehl UpdateEventSourceMapping aktiviert beispielsweise Metriken für eine Zuordnung von Ereignisquellen:

```
aws lambda update-event-source-mapping \
    --uuid a1b2c3d4-5678-90ab-cdef-EXAMPLE11111 \
    --metrics-config Metrics=EventCount
```

Mit der Sum-Statistik können Sie Metriken im Zusammenhang mit der Anzahl der Ereignisse anzeigen.

> **Warning**
>
> Zuordnung von Lambda-Ereignisquellen verarbeiten jedes Ereignis mindestens einmal und es kann zu einer doppelten Verarbeitung von Datensätzen kommen. Aus diesem Grund können Ereignisse in Metriken, die Ereigniszählungen beinhalten, mehrfach gezählt werden.

- `PolledEventCount` – Die Anzahl der Ereignisse, die Lambda erfolgreich aus der Ereignisquelle liest. Wenn Lambda nach Ereignissen abfragt, aber eine leere Umfrage erhält (keine neuen Datensätze), gibt Lambda einen Wert von 0 für diese Metrik aus. Verwenden Sie diese Metrik, um zu ermitteln, ob Ihre Zuordnung von Ereignisquellen korrekt nach neuen Ereignissen fragt.
- `FilteredOutEventCount` – Bei der Zuordnung von Ereignisquellen mit einem Filterkriterium die Anzahl der Ereignisse, die nach diesen Filterkriterien herausgefiltert wurden. Verwenden Sie diese Metrik, um festzustellen, ob Ihre Zuordnung von Ereignisquellen Ereignisse ordnungsgemäß herausfiltert. Für Ereignisse, die den Filterkriterien entsprechen, gibt Lambda eine 0-Metrik aus.
- `InvokedEventCount` – Die Anzahl der Ereignisse, die Ihre Lambda-Funktion aufgerufen haben. Verwenden Sie diese Metrik, um zu überprüfen, ob Ereignisse Ihre Funktion ordnungsgemäß aufrufen. Wenn ein Ereignis zu einem Funktionsfehler oder einer Drosselung führt, kann `InvokedEventCount` aufgrund automatischer Wiederholungsversuche mehrfach für dasselbe abgefragte Ereignis zählen.
- `FailedInvokeEventCount` – Die Anzahl der Ereignisse, mit denen Lambda versucht hat, Ihre Funktion aufzurufen, was jedoch fehlgeschlagen ist. Aufrufe können aus Gründen wie Netzwerkkonfigurationsproblemen, falschen Berechtigungen oder einer gelöschten Lambda-Funktion, Version oder einem Alias fehlschlagen. Wenn für Ihre Zuordnung von Ereignisquellen

partielle Batch-Antworten aktiviert sind, enthält `FailedInvokeEventCount` jedes Ereignis mit einem nicht leeren `BatchItemFailures` in der Antwort.

> **Note**
>
> Der Zeitstempel für die `FailedInvokeEventCount`-Metrik steht für das Ende des Funktionsaufrufs. Dieses Verhalten unterscheidet sich von anderen Lambda-Aufruffehlermetriken, die zu Beginn des Funktionsaufrufs mit einem Zeitstempel versehen werden.

- `DroppedEventCount` – Die Anzahl der Ereignisse, die Lambda aufgrund von Ablauf oder wiederholter Erschöpfung ausgab. Genauer gesagt ist dies die Anzahl der Datensätze, die die von Ihnen konfigurierten Werte für `MaximumRecordAgeInSeconds` oder `MaximumRetryAttempts` überschreiten. Wichtig ist, dass dies nicht die Anzahl der Datensätze beinhaltet, die aufgrund einer Überschreitung der Aufbewahrungseinstellungen Ihrer Ereignisquelle ablaufen. Verworfene Ereignisse schließen auch Ereignisse aus, die Sie bei einem Ausfall an ein Ziel senden. Verwenden Sie diese Metrik, um einen zunehmenden Rückstand an Ereignissen zu erkennen.
- `OnFailureDestinationDeliveredEventCount` – Bei Zuordnungen von Ereignisquellen mit konfiguriertem Ausfallziel, die Anzahl der Ereignisse, die an dieses Ziel gesendet wurden. Verwenden Sie diese Metrik, um Funktionsfehler im Zusammenhang mit Aufrufen von dieser Ereignisquelle zu überwachen. Wenn die Lieferung an das Ziel fehlschlägt, verarbeitet Lambda die Metriken wie folgt:
  - Lambda gibt die `OnFailureDestinationDeliveredEventCount`-Metrik nicht aus.
  - Für die `DestinationDeliveryFailures`-Metrik gibt Lambda eine 1 aus.
  - Für die `DroppedEventCount`-Metrik gibt Lambda eine Zahl aus, die der Anzahl der Ereignisse entspricht, bei denen die Übermittlung fehlgeschlagen ist.
- `DeletedEventCount` – Die Anzahl der Ereignisse, die Lambda nach der Verarbeitung erfolgreich löscht. Wenn Lambda versucht, ein Ereignis zu löschen, dies jedoch fehlschlägt, gibt Lambda eine 0-Metrik aus. Verwenden Sie diese Metrik, um sicherzustellen, dass erfolgreich verarbeitete Ereignisse aus Ihrer Ereignisquelle gelöscht werden.

Wenn Ihre Zuordnung von Ereignisquellen deaktiviert ist, erhalten Sie keine Metriken zur Zuordnung von Ereignisquellen. Möglicherweise werden Ihnen auch fehlende Metriken angezeigt, wenn die Verfügbarkeit von CloudWatch oder Lambda beeinträchtigt ist.

Nicht jede Metrik zur Zuordnung von Ereignisquellen ist für jede Ereignisquelle verfügbar. Derzeit sind Metriken zur Zuordnung von Ereignisquellen für Amazon SQS-, Kinesis- und DynamoDB-Streams-Ereignisquellen verfügbar. Die folgende Verfügbarkeitsmatrix fasst die unterstützten Metriken für jede Art von Ereignisquelle zusammen.

Metrik zur Zuordnung von Ereignisquellen	Support für Amazon SQS	Unterstützung für Kinesis- und DynamoDB-Streams
`PolledEventCount`	Ja	Ja
`FilteredOutEventCount`	Ja	Ja
`InvokedEventCount`	Ja	Ja
`FailedInvokeEventCount`	Ja	Ja
`DroppedEventCount`	Nein	Ja
`OnFailureDestinationDeliveredEventCount`	Nein	Ja
`DeletedEventCount`	Ja	Nein

Wenn sich Ihre Zuordnung von Ereignisquellen im Bereitstellungsmodus befindet, stellt Lambda außerdem die folgende Metrik bereit:

- `ProvisionedPollers` – Bei Zuordnungen von Ereignisquellen im Bereitstellungsmodus die Anzahl der aktiv laufenden Ereignis-Poller. Zeigen Sie diese Metrik mithilfe der `MAX`-Metrik an.

Arbeiten mit Lambda-Funktionsprotokollen

Überwacht Lambda-Funktionen AWS Lambda automatisch in Ihrem Namen, um Ihnen bei der Behebung von Fehlern zu helfen. Sie können Protokolle für Lambda-Funktionen mithilfe der Lambda-Konsole, der CloudWatch Konsole, der AWS Command Line Interface (AWS CLI) und der API anzeigen. CloudWatch Sie können Lambda auch so konfigurieren, dass Protokolle an Amazon S3 und Firehose gesendet werden.

Solange die Ausführungsrolle Ihrer Funktion über die erforderlichen Berechtigungen verfügt, erfasst Lambda Protokolle für alle Anfragen, die von Ihrer Funktion bearbeitet werden, und sendet sie an Amazon CloudWatch Logs, das Standardziel. Sie können die Lambda-Konsole auch verwenden, um Amazon S3 oder Firehose als Protokollierungsziele zu konfigurieren.

- CloudWatch Logs ist das Standard-Logging-Ziel für Lambda-Funktionen. CloudWatch Logs bietet Funktionen zur Anzeige und Analyse von Protokollen in Echtzeit und unterstützt die Erstellung von Metriken und Alarmen auf der Grundlage Ihrer Protokolldaten.
- Amazon S3 ist wirtschaftlich für die langfristige Speicherung, und Services wie Athena können zur Analyse von Protokollen verwendet werden. Die Latenz ist in der Regel höher.
- Firehose bietet verwaltetes Streaming von Protokollen an verschiedene Ziele. Wenn Sie Protokolle an andere AWS Dienste (z. B. OpenSearch Service oder Redshift Data API) oder Plattformen von Drittanbietern (wie Datadog, New Relic oder Splunk) senden müssen, vereinfacht Firehose diesen Prozess durch die Bereitstellung vorgefertigter Integrationen. Sie können auch an benutzerdefinierte HTTP-Endpunkte streamen, ohne zusätzliche Infrastruktur einrichten zu müssen.

Auswahl eines Serviceziels für das Senden von Protokollen

Berücksichtigen Sie bei der Auswahl eines Serviceziels für Funktionsprotokolle die folgenden wichtigen Faktoren:

- Das Kostenmanagement ist je nach Service unterschiedlich. Amazon S3 bietet in der Regel die wirtschaftlichste Option für die Langzeitspeicherung, während CloudWatch Logs es Ihnen ermöglicht, Protokolle einzusehen, zu verarbeiten und Benachrichtigungen in Echtzeit einzurichten. Die Kosten für Firehose umfassen sowohl den Streamingservice als auch die Kosten für die von Ihnen konfigurierten Streamingziele.
- Die Analysefunktionen unterscheiden sich je nach Service. CloudWatch Logs zeichnet sich durch eine hervorragende Echtzeitüberwachung aus und lässt sich nativ in andere CloudWatch

Funktionen wie Logs Insights und Live Tail integrieren. Amazon S3 funktioniert gut mit Analysetools wie Athena und kann in verschiedene Services integriert werden, erfordert jedoch möglicherweise zusätzliche Einstellungen. Firehose vereinfacht das direkte Streaming zu bestimmten AWS Diensten (wie OpenSearch Service und Redshift Data API) und unterstützten Plattformen von Drittanbietern (wie Datadog und Splunk), indem vorgefertigte Integrationen bereitgestellt werden, wodurch der Konfigurationsaufwand potenziell reduziert wird.

- Einrichtung und Benutzerfreundlichkeit variieren je nach Service. CloudWatch Logs ist das standardmäßige Protokollziel. Es funktioniert sofort ohne zusätzliche Konfiguration und ermöglicht die einfache Anzeige und Analyse von Protokollen über die CloudWatch Konsole. Wenn Sie Protokolle an Amazon S3 senden möchten, müssen Sie zunächst einige Einstellungen in der Lambda-Konsole vornehmen und Bucket-Berechtigungen konfigurieren. Wenn Sie Protokolle benötigen, die direkt an Dienste wie OpenSearch Service oder Analyseplattformen von Drittanbietern gesendet werden, kann Firehose diesen Prozess vereinfachen.

Konfigurieren von Protokollzielen

AWS Lambda unterstützt mehrere Ziele für Ihre Funktionsprotokolle. In diesem Leitfaden werden die verfügbaren Protokollziele erläutert und Ihnen bei der Auswahl der für Ihre Anforderungen geeigneten Option geholfen. Unabhängig vom gewählten Ziel bietet Lambda Optionen zur Steuerung des Protokollformats, der Filterung und der Zustellung.

Lambda unterstützt sowohl JSON- als auch Klartextformate für die Protokolle Ihrer Funktion. Strukturierte JSON-Protokolle bieten eine verbesserte Suchbarkeit und ermöglichen eine automatisierte Analyse, während Klartextprotokolle einfach zu handhaben sind und potenziell geringere Speicherkosten verursachen. Sie können steuern, welche Protokolle Lambda an das von Ihnen gewählte Ziel sendet, indem Sie Protokollebenen für System- und Anwendungsprotokolle konfigurieren. Die Filterung hilft Ihnen, die Speicherkosten zu verwalten, und erleichtert das Auffinden relevanter Protokolleinträge beim Debuggen.

Ausführliche Anweisungen zur Einrichtung für jedes Ziel finden Sie in den folgenden Abschnitten:

- Senden von Lambda-Funktionsprotokollen an CloudWatch Logs
- Senden von Lambda-Funktionsprotokollen an Firehose
- Senden von Lambda-Funktionsprotokollen an Amazon S3

Konfigurieren erweiterter Protokollierungsoptionen für Lambda-Funktionen

Um Ihnen mehr Kontrolle über die Erfassung, Verarbeitung und Nutzung Ihrer Funktionsprotokolle zu geben, bietet Lambda die folgenden Optionen zur Protokollierungskonfiguration:

- Protokollformat – Wählen Sie zwischen Klartext und einem strukturierten JSON-Format für die Protokolle Ihrer Funktion aus.
- Protokollebene — Wählen Sie für strukturierte JSON-Logs die Detailebene der Logs aus, an die Lambda sendet CloudWatch`FATAL`, z. B.`ERROR`,`WARN`,`INFO`,`DEBUG`, und`TRACE`.
- Protokollgruppe — wählen Sie die CloudWatch Protokollgruppe aus, an die Ihre Funktion Logs sendet.

Weitere Informationen zur Konfiguration erweiterter Protokollierungssteuerungen finden Sie in den folgenden Abschnitten:

- Konfiguration der JSON- und Klartext-Protokollformate
- Filterung auf Protokollebene
- Konfiguration von CloudWatch Protokollgruppen

Konfiguration der JSON- und Klartext-Protokollformate

Das Erfassen Ihrer Protokollausgaben als JSON-Schlüssel-Wert-Paare erleichtert das Suchen und Filtern beim Debuggen Ihrer Funktionen. Mit Protokollen im JSON-Format können Sie Ihren Protokollen auch Tags und Kontextinformationen hinzufügen. Das kann Ihnen bei der automatisierten Analyse großer Mengen von Protokolldaten helfen. Sofern Ihr Entwicklungsworkflow nicht auf vorhandenen Tools basiert, die Lambda-Protokolle im Klartext verarbeiten, empfehlen wir Ihnen, JSON als Protokollformat auszuwählen.

> **Von Lambda verwaltete Instanzen**
>
> Lambda Managed Instances unterstützen nur das JSON-Protokollformat. Wenn Sie eine Funktion für verwaltete Instanzen erstellen, konfiguriert Lambda das Protokollformat automatisch auf JSON, und Sie können es nicht in Klartext ändern. Weitere Informationen zu Managed Instances finden Sie unter. Von Lambda verwaltete Instanzen

Für alle von Lambda verwalteten Laufzeiten können Sie wählen, ob die Systemprotokolle Ihrer Funktion im unstrukturierten Klartext- oder CloudWatch JSON-Format an Logs gesendet werden. Systemprotokolle sind die von Lambda generierten Protokolle und werden manchmal auch als Plattformereignisprotokolle bezeichnet.

Wenn Sie für unterstützte Laufzeiten eine der unterstützten integrierten Protokollierungsmethoden verwenden, kann Lambda auch die Anwendungsprotokolle Ihrer Funktion (die Protokolle, die Ihr Funktionscode generiert) im strukturierten JSON-Format ausgeben. Wenn Sie das Protokollformat Ihrer Funktion für diese Laufzeiten konfigurieren, gilt die von Ihnen ausgewählte Konfiguration sowohl für System- als auch für Anwendungsprotokolle.

Wenn Ihre Funktion für unterstützte Laufzeiten eine unterstützte Protokollierungsbibliothek oder -methode verwendet, müssen Sie keine Änderungen an Ihrem vorhandenen Code vornehmen, damit Lambda Protokolle in strukturiertem JSON erfasst.

> **Note**
>
> Durch die Verwendung der JSON-Protokollformatierung werden zusätzliche Metadaten hinzugefügt und Protokollmeldungen als JSON-Objekte kodiert, die eine Reihe von Schlüssel-Wert-Paaren enthalten. Aus diesem Grund kann die Größe der Protokollmeldungen Ihrer Funktion zunehmen.

Unterstützte Laufzeiten und Protokollierungsmethoden

Lambda unterstützt derzeit die Option, strukturierte JSON-Anwendungsprotokolle für folgende Laufzeiten auszugeben.

Sprache	Unterstützte Versionen
Java	Alle Java-Laufzeiten außer Java 8 auf Amazon Linux 1
.NET	.NET 8
Node.js	Node.js 16 und höher
Python	Python 3.8 oder höher

Sprache	Unterstützte Versionen
Rust	–

Damit Lambda die Anwendungsprotokolle Ihrer Funktion CloudWatch im strukturierten JSON-Format senden kann, muss Ihre Funktion die folgenden integrierten Protokollierungstools zur Ausgabe von Protokollen verwenden:

- Java: Der `LambdaLogger` Logger oder Log4j2. Weitere Informationen finden Sie unter the section called “Protokollierung”.
- .NET: Die `ILambdaLogger` Instanz auf dem Kontextobjekt. Weitere Informationen finden Sie unter C#-Lambda-Funktionen protokollieren und überwachen.
- Node.js — Die Konsolenmethoden `console.trace`console.debug,`console.log`,`console.info`,`console.error`, und`console.warn`. Weitere Informationen finden Sie unter the section called “Protokollierung”.
- Python: Die `logging` Standard-Python-Bibliothek. Weitere Informationen finden Sie unter the section called “Protokollierung”.
- Rust: Die `tracing` Kiste. Weitere Informationen finden Sie unter the section called “Protokollierung”.

Für andere verwaltete Lambda-Laufzeiten unterstützt Lambda derzeit nur nativ die Erfassung von Systemprotokollen im strukturierten JSON-Format. Sie können jedoch weiterhin Anwendungsprotokolle zu jeder Laufzeit im strukturierten JSON-Format erfassen, indem Sie Protokollierungstools wie Powertools für AWS Lambda die Ausgabe von Protokollausgaben im JSON-Format verwenden.

Standardprotokollformate

Derzeit ist das Standard-Protokollformat für alle Lambda-Laufzeiten das Klartextformat. Für Lambda Managed Instances ist das Protokollformat immer JSON und kann nicht geändert werden.

Wenn Sie bereits Logging-Bibliotheken wie Powertools für AWS Lambda die Generierung Ihrer Funktionsprotokolle im strukturierten JSON-Format verwenden, müssen Sie Ihren Code nicht ändern, wenn Sie die JSON-Protokollformatierung auswählen. Lambda codiert Protokolle, die bereits JSON-codiert sind, nicht doppelt, sodass die Anwendungsprotokolle Ihrer Funktion weiterhin wie zuvor erfasst werden.

JSON-Format für Systemprotokolle

Wenn Sie das Protokollformat Ihrer Funktion als JSON konfigurieren, wird jedes Systemprotokollelement (Plattformereignis) als JSON-Objekt erfasst, das Schlüssel-Wert-Paare mit den folgenden Schlüsseln enthält:

- `"time"` – die Uhrzeit, zu der die Protokollmeldung generiert wurde
- `"type"` – die Art des Ereignisses, das protokolliert wird
- `"record"` – der Inhalt der Protokollausgabe

Das Format des `"record"`-Werts hängt von der Art des protokollierten Ereignisses ab. Weitere Informationen finden Sie unter the section called “Telemetrie-API `Event`-Objekttypen”. Weitere Hinweise zu den Protokollebenen, die Systemprotokollereignissen zugewiesen sind, finden Sie unter the section called “Zuordnung von Ereignissen auf Systemprotokollebene”.

Zum Vergleich zeigen die folgenden beiden Beispiele dieselbe Protokollausgabe sowohl im Klartext- als auch im strukturierten JSON-Format. Beachten Sie, dass Systemprotokollereignisse in den meisten Fällen mehr Informationen enthalten, wenn sie im JSON-Format ausgegeben werden, als wenn sie im Klartext ausgegeben werden.

Example Klartext:

```
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 strukturiertes JSON:

```
{
  "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**
>
> Die the section called “Telemetrie-API” gibt immer Plattformereignisse wie `START` und `REPORT` im JSON-Format aus. Die Konfiguration des Formats der Systemprotokolle, an die Lambda sendet, CloudWatch hat keinen Einfluss auf das Verhalten der Lambda-Telemetrie-API.

JSON-Format für Anwendungsprotokolle

Wenn Sie das Protokollformat Ihrer Funktion als JSON konfigurieren, werden Anwendungsprotokollausgaben, die mit unterstützten Protokollierungsbibliotheken und -methoden geschrieben wurden, als JSON-Objekt erfasst, das Schlüssel-Wert-Paare mit folgenden Schlüsseln enthält.

- `"timestamp"` – die Uhrzeit, zu der die Protokollmeldung generiert wurde
- `"level"` – die der Meldung zugewiesene Protokollebene
- `"message"` – der Inhalt der Protokollmeldung
- `"requestId"` (Python, .NET und Node.js) oder `"AWSrequestId"` (Java) – die eindeutige Anforderungs-ID für den Funktionsaufruf

Abhängig von der Laufzeit und der Protokollierungsmethode, die Ihre Funktion verwendet, kann dieses JSON-Objekt auch zusätzliche Schlüsselpaare enthalten. Wenn Sie Node.js verwenden und Ihre Funktion beispielsweise `console`-Methoden nutzt, um Fehlerobjekte mit mehreren Argumenten zu protokollieren, enthält das JSON-Objekt zusätzliche Schlüssel-Wert-Paare mit den Schlüsseln `errorMessage`, `errorType` und `stackTrace`. Weitere Informationen zu JSON-formatierten Protokollen in verschiedenen Lambda-Laufzeiten finden Sie unter the section called “Protokollierung”, the section called “Protokollierung” und the section called “Protokollierung”.

> **Note**
>
> Der Schlüssel, den Lambda für den Zeitstempelwert verwendet, unterscheidet sich für Systemprotokolle und für Anwendungsprotokolle. Bei Systemprotokollen verwendet Lambda den Schlüssel `"time"`, um die Konsistenz mit der Telemetrie-API aufrechtzuerhalten. Bei Anwendungsprotokollen folgt Lambda den Konventionen der unterstützten Laufzeiten und verwendet `"timestamp"`.

Zum Vergleich zeigen die folgenden beiden Beispiele dieselbe Protokollausgabe sowohl im Klartext- als auch im strukturierten JSON-Format.

Example Klartext:

```
2024-10-27T19:17:45.586Z 79b4f56e-95b1-4643-9700-2807f4e68189 INFO some log message
```

Example strukturiertes JSON:

```
{
    "timestamp":"2024-10-27T19:17:45.586Z",
    "level":"INFO",
    "message":"some log message",
    "requestId":"79b4f56e-95b1-4643-9700-2807f4e68189"
}
```

Festlegen des Protokollformats Ihrer Funktion

Um das Protokollformat für Ihre Funktion zu konfigurieren, können Sie die Lambda-Konsole oder die AWS Command Line Interface (AWS CLI) verwenden. Sie können das Protokollformat einer Funktion auch mithilfe der API-Befehle CreateFunctionund UpdateFunctionConfigurationLambda, der AWS::Serverless::FunctionRessource AWS Serverless Application Model (AWS SAM) und der CloudFormation AWS::Lambda::FunctionRessource konfigurieren.

Das Ändern des Protokollformats Ihrer Funktion hat keine Auswirkungen auf bestehende Protokolle, die in CloudWatch Logs gespeichert sind. Nur neue Protokolle verwenden das aktualisierte Format.

Wenn Sie das Protokollformat Ihrer Funktion in JSON ändern und keine Protokollebene festlegen, setzt Lambda die Anwendungsprotokollebene und die Systemprotokollebene Ihrer Funktion automatisch auf INFO. Das bedeutet, dass Lambda nur Protokollausgaben der Stufe INFO und niedriger an CloudWatch Logs sendet. Weitere Informationen zur Filterung auf Anwendungs- und Systemprotokollebene finden Sie unter the section called “Filterung auf Protokollebene”

> **Note**
>
> Wenn das Protokollformat Ihrer Funktion auf Klartext gesetzt ist, ist die Standardeinstellung auf Protokollebene für Python-Laufzeiten WARN. Das bedeutet, dass Lambda nur Protokollausgaben der Stufe WARN und niedriger an CloudWatch Logs sendet. Wenn Sie das Protokollformat Ihrer Funktion in JSON ändern, ändert sich dieses Standardverhalten.

Weitere Informationen zur Protokollierung in Python finden Sie unter the section called "Protokollierung".

Bei Funktionen von Node.js, die EMF-Logs (Embedded Metric Format) ausgeben, kann die Änderung des Protokollformats Ihrer Funktion in JSON dazu führen, dass Ihre Metriken nicht erkannt CloudWatch werden können.

Important

Wenn Ihre Funktion Powertools for AWS Lambda (TypeScript) oder die Open-Source-EMF-Clientbibliotheken zur Ausgabe von EMF-Protokollen verwendet, aktualisieren Sie Ihre Powertools - und EMF-Bibliotheken auf die neuesten Versionen, um sicherzustellen, dass Ihre Protokolle weiterhin korrekt analysiert CloudWatch werden können. Wenn Sie zum JSON-Protokollformat wechseln, empfehlen wir Ihnen zudem, Tests durchzuführen, um die Kompatibilität mit den eingebetteten Metriken Ihrer Funktion sicherzustellen. Weitere Hinweise zu den Funktionen von node.js, die EMF-Protokolle ausgeben, finden Sie unter the section called "Verwenden von Clientbibliotheken im Embedded Metric Format (EMF) mit strukturierten JSON-Protokollen".

So konfigurieren Sie das Protokollformat einer Funktion (Konsole)

1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.
2. Auswählen einer Funktion
3. Wählen Sie auf der Konfigurationsseite der Funktion die Option Überwachungs- und Betriebstools aus.
4. Wählen Sie im Bereich Protokollierungskonfiguration die Option Bearbeiten aus.
5. Wählen Sie unter Protokollinhalt für Protokollformat entweder Text oder JSON aus.
6. Wählen Sie Speichern.

So ändern Sie das Protokollformat einer vorhandenen Funktion (AWS CLI)

- Verwenden Sie den Befehl update-function-configuration, um das Protokollformat einer vorhandenen Funktion zu ändern. Legen Sie die `LogFormat`-Option in `LoggingConfig` entweder auf `JSON` oder `Text` fest.

```
aws lambda update-function-configuration \
  --function-name myFunction \
  --logging-config LogFormat=JSON
```

So legen Sie das Protokollformat fest, wenn Sie eine Funktion erstellen (AWS CLI)

- Verwenden Sie die Option `--logging-config` im Befehl create-function, um das Protokollformat zu konfigurieren, wenn Sie eine neue Funktion erstellen. Legen Sie `LogFormat` auf `JSON` oder `Text` fest. Der folgende Beispielbefehl erstellt eine Node.js-Funktion, die Protokolle in strukturiertem JSON ausgibt.

  Wenn Sie beim Erstellen einer Funktion kein Protokollformat angeben, verwendet Lambda das Standardprotokollformat für die von Ihnen ausgewählte Laufzeitversion. Informationen zu den Standard-Protokollierungsformaten finden Sie unter the section called “Standardprotokollformate”.

  ```
  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
  ```

Filterung auf Protokollebene

Lambda kann die Protokolle Ihrer Funktion filtern, sodass nur Protokolle mit einer bestimmten Detailebene oder niedriger an CloudWatch Logs gesendet werden. Sie können die Filterung auf Protokollebene separat für die Systemprotokolle Ihrer Funktion (die von Lambda generierten Protokolle) und Anwendungsprotokolle (die von Ihrem Funktionscode generierten Protokolle) konfigurieren.

Für the section called “Unterstützte Laufzeiten und Protokollierungsmethoden” müssen Sie keine Änderungen an Ihrem Funktionscode vornehmen, damit Lambda die Anwendungsprotokolle Ihrer Funktion filtert.

Für alle anderen Laufzeiten und Protokollierungsmethoden muss Ihr Funktionscode Protokollereignisse in `stdout` oder `stderr` als JSON-formatierte Objekte ausgeben, die ein Schlüssel-Wert-Paar mit dem Schlüssel `"level"` enthalten. Lambda interpretiert beispielsweise die folgende Ausgabe an `stdout` als ein Protokoll auf Ebene DEBUG.

```
print('{"level": "debug", "msg": "my debug log", "timestamp":
 "2024-11-02T16:51:31.587199Z"}')
```

Wenn das Feld für den `"level"`-Wert ungültig ist oder fehlt, weist Lambda der Protokollausgabe die Ebene INFO zu. Damit Lambda das Zeitstempelfeld verwenden kann, müssen Sie die Zeit im gültigen RFC 3339-Zeitstempelformat angeben. Wenn Sie keinen gültigen Zeitstempel angeben, weist Lambda dem Protokoll die Ebene INFO zu und fügt einen Zeitstempel für Sie hinzu.

Halten Sie sich bei der Benennung des Zeitstempelschlüssels an die Benennungskonventionen der Laufzeit, die Sie verwenden. Lambda unterstützt die meisten gängigen Namenskonventionen, die von den verwalteten Laufzeiten verwendet werden.

> **Note**
>
> Um die Filterung auf Protokollebene verwenden zu können, muss Ihre Funktion für die Verwendung des JSON-Protokollformats konfiguriert sein. Derzeit ist das Standard-Protokollformat für alle verwalteten Lambda-Laufzeiten das Klartextformat. Informationen zur Konfiguration des JASON-Protokollformats Ihrer Funktion finden Sie unter the section called “Festlegen des Protokollformats Ihrer Funktion”.

Bei Anwendungsprotokollen (den durch Ihren Funktionscode generierten Protokollen) können Sie zwischen den folgenden Protokollebenen wählen.

Protokollebene	Standardnutzung
TRACE (am detailliertesten)	Die detailliertesten Informationen, die verwendet werden, um den Ausführungspfad Ihres Codes nachzuverfolgen
DEBUG	Detaillierte Informationen für das System-De bugging

Protokollebene	Standardnutzung
INFO	Meldungen, die den normalen Betrieb Ihrer Funktion erfassen
WARN	Meldungen über mögliche Fehler, die zu unerwartetem Verhalten führen können, wenn sie nicht behoben werden
ERROR	Meldungen über Probleme, die verhindern, dass der Code wie erwartet funktioniert
FATAL (am wenigsten Details)	Meldungen über schwerwiegende Fehler, die dazu führen, dass die Anwendung nicht mehr funktioniert

Wenn Sie eine Protokollebene auswählen, sendet Lambda Protokolle auf dieser Ebene und niedriger an CloudWatch Logs. Wenn Sie beispielsweise die Anwendungsprotokollebene einer Funktion auf WARN setzen, sendet Lambda keine Protokollausgaben auf den Protokollebenen INFO und DEBUG. Die Standardebene des Anwendungsprotokolls für die Protokollfilterung ist INFO.

Wenn Lambda die Anwendungsprotokolle Ihrer Funktion filtert, wird Protokollmeldungen ohne Ebene die Protokollebene INFO zugewiesen.

Für Systemprotokolle (die vom Lambda-Service generierten Protokolle) können Sie zwischen den folgenden Protokollebenen wählen.

Protokollebene	Usage
DEBUG (am detailliertesten)	Detaillierte Informationen für das System-De bugging
INFO	Meldungen, die den normalen Betrieb Ihrer Funktion erfassen
WARN (am wenigsten Details)	Meldungen über mögliche Fehler, die zu unerwartetem Verhalten führen können, wenn sie nicht behoben werden

Wenn Sie eine Protokollebene auswählen, sendet Lambda Protokolle auf dieser Ebene und niedriger. Wenn Sie beispielsweise die Protokollebene einer Funktion auf INFO setzen, sendet Lambda keine Protokollausgaben auf der Protokollebene DEBUG.

Standardmäßig setzt Lambda die Protokollebene des Systems auf INFO. Mit dieser Einstellung sendet `"start"` und `"report"` protokolliert Lambda automatisch Nachrichten an CloudWatch. Um mehr oder weniger detaillierte Systemprotokolle zu erhalten, ändern Sie die Protokollebene in DEBUG oder WARN. Eine Liste der Protokollebenen, denen Lambda verschiedene Systemprotokollereignisse zuordnet, finden Sie unter the section called “Zuordnung von Ereignissen auf Systemprotokollebene”.

Konfigurieren der Filterung auf Protokollebene

Um die Filterung auf Anwendungs- und Systemprotokollebene für Ihre Funktion zu konfigurieren, können Sie die Lambda-Konsole oder die AWS Command Line Interface () verwenden.AWS CLI Sie können die Protokollebene einer Funktion auch mithilfe der API-Befehle CreateFunctionund UpdateFunctionConfigurationLambda, der AWS::Serverless::FunctionRessource AWS Serverless Application Model (AWS SAM) und der CloudFormation AWS::Lambda::FunctionRessource konfigurieren.

Beachten Sie, dass, wenn Sie die Protokollebene Ihrer Funktion in Ihrem Code festlegen, diese Einstellung Vorrang vor allen anderen von Ihnen konfigurierten Einstellungen auf Protokollebene hat. Wenn Sie beispielsweise die Python-Methode `logging setLevel()` verwenden, um die Protokollierungsebene Ihrer Funktion auf INFO zu setzen, hat diese Einstellung Vorrang vor der Einstellung WARN, die Sie mit der Lambda-Konsole konfigurieren.

So konfigurieren Sie die Anwendungs- oder Systemprotokollebene (Konsole) einer vorhandenen Funktion

1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.
2. Wählen Sie eine Funktion aus.
3. Wählen Sie auf der Konfigurationsseite der Funktion die Option Überwachungs- und Betriebstools aus.
4. Wählen Sie im Bereich Protokollierungskonfiguration die Option Bearbeiten aus.
5. Stellen Sie sicher, dass unter Protokollinhalt für Protokollformat JSON ausgewählt ist.
6. Wählen Sie mit den Optionsfeldern die gewünschte Anwendungsprotokollebene und die gewünschte Systemprotokollebene für Ihre Funktion aus.
7. Wählen Sie Speichern.

So konfigurieren Sie die Anwendungs- oder Systemprotokollebene einer vorhandenen Funktion (AWS CLI)

- Verwenden Sie den Befehl update-function-configuration, um die Anwendungs- oder Systemprotokollebene einer vorhandenen Funktion zu ändern. Verwenden Sie `--logging-config`, um `SystemLogLevel` auf einen der Werte `DEBUG`, `INFO` oder `WARN` zu setzen. Legen Sie `ApplicationLogLevel` auf `DEBUG`, `INFO`, `WARN`, `ERROR` oder `FATAL` fest.

```
aws lambda update-function-configuration \
  --function-name myFunction \
  --logging-config LogFormat=JSON,ApplicationLogLevel=ERROR,SystemLogLevel=WARN
```

So konfigurieren Sie die Filterung auf Protokollebene beim Erstellen einer Funktion

- Um die Filterung der Protokollebene zu konfigurieren, wenn Sie eine neue Funktion erstellen, verwenden Sie `--logging-config`, um die Tasten `SystemLogLevel` und `ApplicationLogLevel` im Befehl create-function zu setzen. Legen Sie `SystemLogLevel` auf `DEBUG`, `INFO` oder `WARN` fest. Legen Sie `ApplicationLogLevel` auf `DEBUG`, `INFO`, `WARN`, `ERROR` oder `FATAL` fest.

```
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
```

Zuordnung von Ereignissen auf Systemprotokollebene

Für von Lambda generierte Protokollereignisse auf Systemebene ist in der folgenden Tabelle die Protokollebene definiert, die jedem Ereignis zugewiesen ist. Weitere Informationen zu den in der Tabelle aufgeführten Ereignissen finden Sie unter the section called “Referenz zum `Event`-Schema”

Ereignisname	Bedingung	Zugewiesene Protokollebene
initStart	„runtimeVersion“ ist festgelegt	INFO

Ereignisname	Bedingung	Zugewiesene Protokollebene
initStart	„runtimeVersion“ ist nicht festgelegt	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“ ist festgelegt	INFO
restoreStart	„runtimeVersion“ ist nicht festgelegt	DEBUG
restoreRuntimeDone	status=success	DEBUG
restoreRuntimeDone	status!=success	WARN
restoreReport	status=success	INFO
restoreReport	status!=success	WARN
start	-	INFO
runtimeDone	status=success	DEBUG
runtimeDone	status!=success	WARN
report	status=success	INFO
report	status!=success	WARN
extension	state=success	INFO
extension	state!=success	WARN

Ereignisname	Bedingung	Zugewiesene Protokollebene
logSubscription	-	INFO
telemetrySubscription	-	INFO
logsDropped	-	WARN

> **Note**
>
> Die the section called “Telemetrie-API” gibt immer den vollständigen Satz von Plattformereignissen aus. Die Konfiguration der Ebene der Systemprotokolle, an die Lambda sendet, CloudWatch hat keinen Einfluss auf das Verhalten der Lambda-Telemetrie-API.

Filterung auf Anwendungsprotokollebene mit benutzerdefinierten Laufzeiten

Wenn Sie die Filterung auf Anwendungsprotokollebene für Ihre Funktion konfigurieren, legt Lambda hinter den Kulissen die Anwendungsprotokollebene in der Laufzeit mithilfe der Umgebungsvariable `AWS_LAMBDA_LOG_LEVEL` fest. Lambda legt auch das Protokollformat Ihrer Funktion mithilfe der Umgebungsvariable `AWS_LAMBDA_LOG_FORMAT` fest. Sie können diese Variablen verwenden, um erweiterte Lambda-Protokollierungsoptionen in eine benutzerdefinierte Laufzeit zu integrieren.

Um Protokollierungseinstellungen für eine Funktion konfigurieren zu können, die eine benutzerdefinierte Laufzeit mit der Lambda-Konsole und Lambda verwendet AWS CLI APIs, konfigurieren Sie Ihre benutzerdefinierte Laufzeit so, dass sie den Wert dieser Umgebungsvariablen überprüft. Anschließend können Sie die Logger Ihrer Laufzeit gemäß dem von Ihnen ausgewählten Protokollformat und den von Ihnen ausgewählten Protokollebenen konfigurieren.

Senden von Lambda-Funktionsprotokollen an CloudWatch Logs

Standardmäßig erfasst Lambda automatisch Protokolle für alle Funktionsaufrufe und sendet sie an CloudWatch Logs, sofern die Ausführungsrolle Ihrer Funktion über die notwendigen Berechtigungen verfügt. Diese Protokolle werden standardmäßig in einer Protokollgruppe mit dem Namen „/aws/lambda/*<Funktionsname>*“ gespeichert. Um das Debugging zu verbessern, können Sie benutzerdefinierte Protokollierungsanweisungen in Ihren Code einfügen, die Lambda nahtlos in CloudWatch Logs integriert. Falls erforderlich, können Sie Ihre Funktion so konfigurieren, dass Protokolle an eine andere Gruppe gesendet werden. Sie können dazu die Lambda-Konsole, die

AWS CLI oder die Lambda-API verwenden. Weitere Informationen hierzu finden Sie unter the section called “ CloudWatch Funktionsprotokolle konfigurieren”.

Sie können Protokolle für Lambda-Funktionen mithilfe der Lambda-Konsole, der CloudWatch-Konsole, der AWS Command Line Interface (AWS CLI) oder der CloudWatch-API anzeigen. Weitere Informationen finden Sie unter CloudWatch Logs für Lambda-Funktionen anzeigen.

> **Note**
>
> Es kann 5 bis 10 Minuten dauern, bis Protokolle nach einem Funktionsaufruf angezeigt werden.

Erforderliche IAM-Berechtigungen

Ihre Ausführungsrolle benötigt die folgenden Berechtigungen, um Protokolle zu CloudWatch Logs hochzuladen:

- `logs:CreateLogGroup`
- `logs:CreateLogStream`
- `logs:PutLogEvents`

Weitere Informationen finden Sie unter Verwendung von identitätsbasierten Richtlinien (IAM-Richtlinien) für CloudWatch Logs im Amazon CloudWatch User Guide.

Sie können diese CloudWatch-Logs-Berechtigungen mithilfe einer von Lambda bereitgestellten von `AWSLambdaBasicExecutionRole`AWS verwalteten Richtlinie hinzufügen. Führen Sie den folgenden Befehl aus, um diese Richtlinie zu Ihrer Rolle hinzuzufügen:

```
aws iam attach-role-policy --role-name your-role --policy-arn arn:aws:iam::aws:policy/
service-role/AWSLambdaBasicExecutionRole
```

Weitere Informationen finden Sie unter the section called “AWS verwaltete Richtlinien”.

Preisgestaltung

Für die Verwendung von Lambda-Protokollen fallen keine zusätzlichen Gebühren an; Es fallen jedoch die Standardgebühren für CloudWatch Logs an. Weitere Informationen hierzu finden Sie unter Amazon CloudWatch – Preise.

Konfiguration von CloudWatch Protokollgruppen

Standardmäßig wird CloudWatch automatisch eine Protokollgruppe erstellt, die nach Ihrer Funktion benannt ist, wenn sie `/aws/lambda/<function name>` zum ersten Mal aufgerufen wird. Um Ihre Funktion so zu konfigurieren, dass Protokolle an eine bestehende Protokollgruppe gesendet werden, oder um eine neue Protokollgruppe für Ihre Funktion zu erstellen, können Sie die Lambda-Konsole oder die AWS CLI verwenden. Sie können benutzerdefinierte Protokollgruppen auch mithilfe der Befehle CreateFunctionund UpdateFunctionConfigurationLambda API und der Ressource AWS Serverless Application Model (AWS SAM) AWS: :Serverless: :Function konfigurieren.

Sie können mehrere Lambda-Funktionen so konfigurieren, dass sie Protokolle an dieselbe CloudWatch Protokollgruppe senden. Beispielsweise könnten Sie eine einzelne Protokollgruppe verwenden, um Protokolle für alle Lambda-Funktionen zu speichern, aus denen eine bestimmte Anwendung besteht. Wenn Sie eine benutzerdefinierte Protokollgruppe für eine Lambda-Funktion verwenden, enthalten die von Lambda erstellten Protokollstreams den Funktionsnamen und die Funktionsversion. Dadurch wird sichergestellt, dass die Zuordnung zwischen Protokollmeldungen und Funktionen erhalten bleibt, auch wenn Sie dieselbe Protokollgruppe für mehrere Funktionen verwenden.

Das Protokollstream-Benennungsformat für benutzerdefinierte Protokollgruppen folgt dieser Benennungskonvention:

```
YYYY/MM/DD/<function_name>[<function_version>][<execution_environment_GUID>]
```

Beachten Sie, dass bei der Konfiguration einer benutzerdefinierten Protokollgruppe der Name, den Sie für Ihre Protokollgruppe auswählen, den Benennungsregeln für CloudWatch Protokolle entsprechen muss. Darüber hinaus dürfen benutzerdefinierte Protokollgruppennamen nicht mit der Zeichenfolge `aws/` beginnen. Wenn Sie eine benutzerdefinierte Protokollgruppe erstellen, die mit `aws/` beginnt, kann Lambda die Protokollgruppe nicht erstellen. Aus diesem Grund werden die Protokolle Ihrer Funktion nicht an gesendet CloudWatch.

So ändern Sie die Protokollgruppe einer Funktion (Konsole)

1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.
2. Wählen Sie eine Funktion aus.
3. Wählen Sie auf der Konfigurationsseite der Funktion die Option Überwachungs- und Betriebstools aus.
4. Wählen Sie im Bereich Protokollierungskonfiguration die Option Bearbeiten aus.

5. Wählen Sie im Bereich Protokollgruppe für CloudWatch Protokollgruppe die Option Benutzerdefiniert aus.
6. Geben Sie unter Benutzerdefinierte Protokollgruppe den Namen der CloudWatch Protokollgruppe ein, an die Ihre Funktion Protokolle senden soll. Wenn Sie den Namen einer vorhandenen Protokollgruppe eingeben, verwendet Ihre Funktion diese Gruppe. Wenn keine Protokollgruppe mit dem von Ihnen eingegebenen Namen existiert, erstellt Lambda eine neue Protokollgruppe für Ihre Funktion mit diesem Namen.

So ändern Sie die Protokollgruppe einer Funktion (AWS CLI)

- Verwenden Sie den Befehl update-function-configuration, um die Protokollgruppe einer vorhandenen Funktion zu ändern.

```
aws lambda update-function-configuration \
  --function-name myFunction \
  --logging-config LogGroup=myLogGroup
```

So geben Sie eine benutzerdefinierte Protokollgruppe an, wenn Sie eine Funktion erstellen (AWS CLI)

- Um eine benutzerdefinierte Protokollgruppe anzugeben, wenn Sie eine neue Lambda-Funktion mit dem erstellen AWS CLI, verwenden Sie die `--logging-config` Option. Mit dem folgenden Beispielbefehl wird eine Node.js-Lambda-Funktion erstellt, die Protokolle an eine Protokollgruppe mit dem Namen `myLogGroup` sendet.

```
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
```

Berechtigungen für die Ausführungsrolle

Damit Ihre Funktion Logs an CloudWatch Logs senden kann, muss sie über die Logs: PutLogEvents -Berechtigung verfügen. Wenn Sie die Protokollgruppe Ihrer Funktion mithilfe der Lambda-Konsole konfigurieren, fügt Lambda diese Berechtigung der Rolle unter folgenden Bedingungen hinzu:

- Das Serviceziel ist auf CloudWatch Logs gesetzt
- Die Ausführungsrolle Ihrer Funktion ist nicht berechtigt, Logs in Logs (das Standardziel) hochzuladen CloudWatch

> **Note**
>
> Lambda fügt keine Put-Berechtigungen für Amazon S3- oder Firehose-Protokollziele hinzu.

Wenn Lambda diese Berechtigung hinzufügt, erteilt es der Funktion die Erlaubnis, Protokolle an jede CloudWatch Logs-Protokollgruppe zu senden.

Um zu verhindern, dass Lambda die Ausführungsrolle der Funktion automatisch aktualisiert, damit Sie diese stattdessen manuell bearbeiten, erweitern Sie Berechtigungen und deaktivieren Sie Erforderliche Berechtigungen hinzufügen.

Wenn Sie die Protokollgruppe Ihrer Funktion mithilfe von konfigurieren AWS CLI, fügt Lambda die `logs:PutLogEvents` Berechtigung nicht automatisch hinzu. Fügen Sie die Berechtigung zur Ausführungsrolle Ihrer Funktion hinzu, falls noch nicht geschehen. Diese Berechtigung ist in der AWSLambdaBasicExecutionRoleverwalteten Richtlinie enthalten.

CloudWatch Protokollierung für Lambda Managed Instances

Bei der Verwendung von Lambda Managed Instances gibt es zusätzliche Überlegungen zum Senden von Protokollen an CloudWatch Logs:

VPC-Netzwerkanforderungen

Lambda Managed Instances werden auf kundeneigenen EC2 Instances innerhalb Ihrer VPC ausgeführt. Um CloudWatch Logs an Logs und Traces an X-Ray zu senden, müssen Sie sicherstellen, dass diese von Ihrer VPC aus routingfähig AWS APIs sind. Sie haben mehrere Möglichkeiten:

- AWS PrivateLink (empfohlen): Wird verwendet AWS PrivateLink, um VPC-Endpunkte für CloudWatch Logs und X-Ray-Dienste zu erstellen. Dadurch können Ihre Instances privat auf diese Dienste zugreifen, ohne dass ein Internet-Gateway oder NAT-Gateway erforderlich ist. Weitere Informationen finden Sie unter Verwenden von CloudWatch Protokollen mit VPC-Endpunkten der Schnittstelle.
- NAT-Gateway: Konfigurieren Sie ein NAT-Gateway, um ausgehenden Internetzugang von Ihren privaten Subnetzen aus zu ermöglichen.
- Internet Gateway: Stellen Sie für öffentliche Subnetze sicher, dass für Ihre VPC ein Internet-Gateway konfiguriert ist.

Wenn CloudWatch Logs oder X-Ray nicht von Ihrer VPC aus routingfähig APIs sind, werden Ihre Funktionsprotokolle und Traces nicht zugestellt.

Gleichzeitige Aufrufe und Protokollzuweisung

Ausführungsumgebungen für Lambda Managed Instances können mehrere Aufrufe gleichzeitig verarbeiten. Wenn mehrere Aufrufe gleichzeitig ausgeführt werden, werden ihre Protokolleinträge im selben Protokollstream verschachtelt. Um Protokolle von gleichzeitigen Aufrufen effektiv zu filtern und zu analysieren, sollten Sie sicherstellen, dass jeder Protokolleintrag die Anforderungs-ID enthält. AWS

Wir empfehlen einen der folgenden Ansätze:

- Verwenden Sie standardmäßige Lambda-Laufzeitlogger (empfohlen): Die von Lambda Managed Runtimes bereitgestellten Standard-Logging-Bibliotheken enthalten automatisch die Anforderungs-ID in jedem Protokolleintrag.
- Implementieren Sie strukturierte JSON-Protokollierung: Wenn Sie eine benutzerdefinierte Laufzeit erstellen oder eine benutzerdefinierte Protokollierung benötigen, implementieren Sie Protokolle im JSON-Format, die die Anforderungs-ID in jedem Eintrag enthalten. Lambda Managed Instances unterstützen nur das JSON-Protokollformat. Nehmen Sie das `requestId` Feld in Ihre JSON-Logs auf, um die Filterung nach Aufrufen zu ermöglichen:

```
{
  "timestamp": "2025-01-15T10:30:00.000Z",
  "level": "INFO",
  "requestId": "a1b2c3d4-e5f6-7890-abcd-ef1234567890",
  "message": "Processing request"
}
```

Mit der Zuordnung der Anfrage-ID können Sie CloudWatch Log-Log-Log-Einträge für einen bestimmten Aufruf mithilfe von CloudWatch Logs Insights-Abfragen filtern. Beispiel:

```
fields @timestamp, @message
| filter requestId = "a1b2c3d4-e5f6-7890-abcd-ef1234567890"
| sort @timestamp asc
```

Weitere Informationen zu den Protokollierungsanforderungen für Lambda Managed Instances finden Sie unterGrundlegendes zur Ausführungsumgebung von Lambda Managed Instances.

CloudWatch Logs für Lambda-Funktionen anzeigen

Sie können CloudWatch Amazon-Protokolle für Ihre Lambda-Funktion mit der Lambda-Konsole, der CloudWatch Konsole oder der AWS Command Line Interface ()AWS CLI anzeigen. Folgen Sie den Anweisungen in den folgenden Abschnitten, um auf die Protokolle Ihrer Funktion zuzugreifen.

Streamen Sie CloudWatch Funktionsprotokolle mit Logs Live Tail

Amazon CloudWatch Logs Live Tail hilft Ihnen bei der schnellen Fehlerbehebung bei Ihren Funktionen, indem es eine Streaming-Liste mit neuen Protokollereignissen direkt in der Lambda-Konsole anzeigt. Sie können die von Ihren Lambda-Funktionen aufgenommenen Protokolle in Echtzeit anzeigen und filtern, sodass Sie Probleme schnell erkennen und beheben können.

> **Note**
>
> Bei Live Tail-Sitzungen fallen Kosten pro Minute an. Weitere Informationen zur Preisgestaltung finden Sie unter CloudWatch Amazon-Preise.

Vergleich von Live Tail und --log-type Tail

Es gibt mehrere Unterschiede zwischen CloudWatch Logs Live Tail und der Option LogType: Tail in der Lambda-API (`--log-type Tail`in der AWS CLI):

- `--log-type Tail` gibt nur die ersten 4 KB der Aufruf-Logs zurück. Live Tail hat dieses Limit nicht gemeinsam und kann bis zu 500 Protokollereignisse pro Sekunde empfangen.
- `--log-type Tail` erfasst und sendet die Protokolle zusammen mit der Antwort, was sich auf die Antwortlatenz der Funktion auswirken kann. Live Tail hat keinen Einfluss auf die Reaktionslatenz der Funktionen.

- `--log-type Tail` unterstützt nur synchrone Aufrufe. Live Tail funktioniert sowohl für synchrone als auch für asynchrone Aufrufe.

> **Note**
>
> Lambda Managed Instances unterstützt `--log-type Tail` diese Option nicht. Verwenden Sie CloudWatch Logs Live Tail oder fragen Sie CloudWatch Logs direkt ab, um Logs für Funktionen von Managed Instances einzusehen.

Berechtigungen

Die folgenden Berechtigungen sind erforderlich, um CloudWatch Logs Live Tail-Sitzungen zu starten und zu beenden:

- `logs:DescribeLogGroups`
- `logs:StartLiveTail`
- `logs:StopLiveTail`

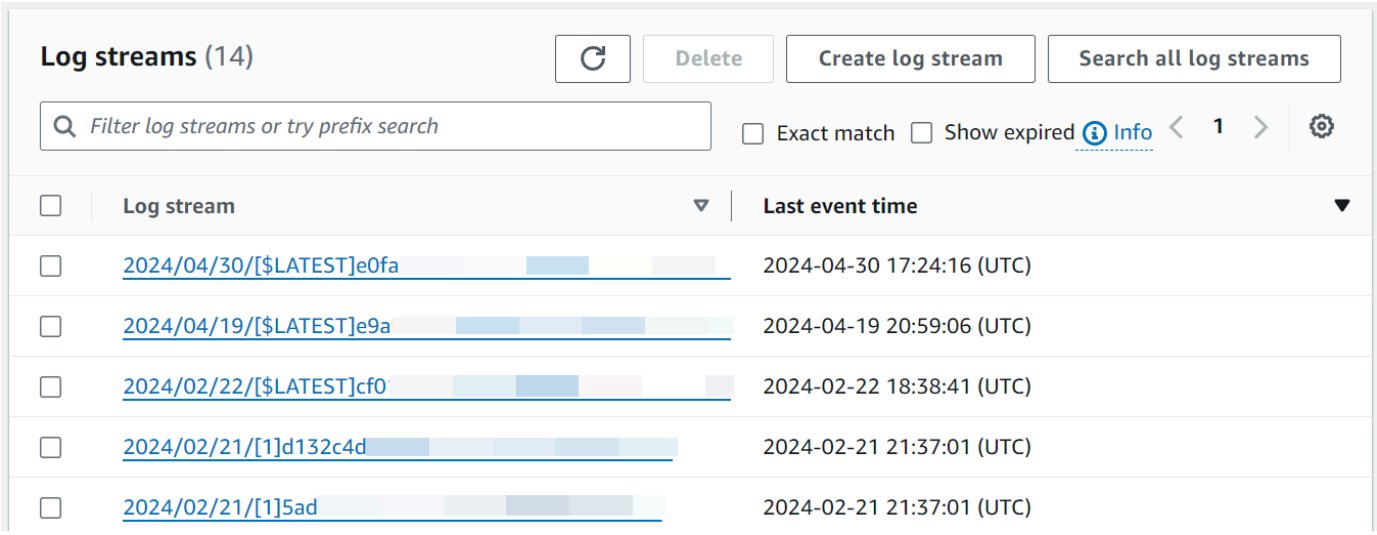
Starten einer Live-Tail-Sitzung in der Lambda-Konsole

1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.
2. Wählen Sie den Namen einer Funktion aus.
3. Wählen Sie die Registerkarte Test.
4. Wählen Sie im Bereich Testereignis die Option CloudWatch Logs Live Tail aus.
5. Für Protokollgruppen auswählen ist standardmäßig die Protokollgruppe der Funktion ausgewählt. Sie können bis zu fünf Protokollgruppen gleichzeitig auswählen.
6. (Optional) Um nur Protokollereignisse anzuzeigen, die bestimmte Wörter oder andere Zeichenfolgen enthalten, geben Sie das Wort oder die Zeichenfolge in das Feld Filtermuster hinzufügen ein. Im Feld „Filter“ wird zwischen Groß- und Kleinschreibung unterschieden. Sie können mehrere Begriffe und Musteroperatoren in dieses Feld eingeben, einschließlich regulärer Ausdrücke (regex). Weitere Informationen zur Mustersyntax finden Sie unter Filtermustersyntax. im Amazon CloudWatch Logs-Benutzerhandbuch.
7. Wählen Sie Starten. Im Fenster werden nun übereinstimmende Protokollereignisse angezeigt.
8. Zum Beenden der Live-Tail-Sitzung wählen Sie Stoppen.

> **Note**
>
> Die Live-Tail-Sitzung wird automatisch nach 15 Minuten Inaktivität oder nach Ablauf der Zeit für die Lambda-Konsolensitzung beendet.

Zugriff auf Funktionsprotokolle über die Konsole

1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.
2. Wählen Sie eine Funktion aus.
3. Wählen Sie den Tab Überwachung.
4. Wählen Sie CloudWatch Protokolle anzeigen, um die CloudWatch Konsole zu öffnen.
5. Scrollen Sie nach unten und wählen Sie den Protokollstream für die Funktionsaufrufen aus, die Sie sich ansehen möchten.

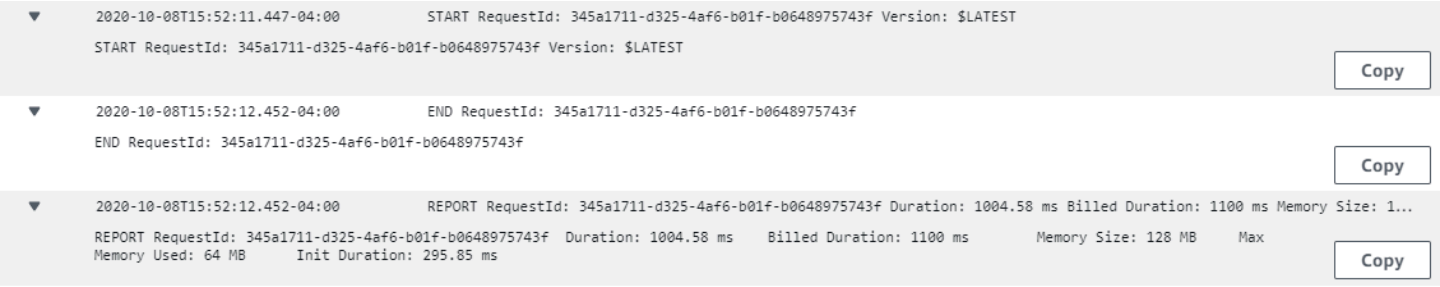


Jede Instance einer Lambda-Funktion hat einen eigenen Protokollstream. Wenn eine Funktion skaliert wird, hat jede gleichzeitige Instance ihren eigenen Protokollstream. Jedes Mal, wenn als Reaktion auf einen Aufruf eine neue Ausführungsumgebung erstellt wird, wird ein neuer Protokollstream generiert. Die Benennungskonvention für Protokollstreams lautet:

```
YYYY/MM/DD[Function version][Execution environment GUID]
```

Eine einzelne Ausführungsumgebung schreibt während ihrer Lebensdauer in denselben Protokollstream. Der Protokollstream enthält Nachrichten aus dieser Ausführungsumgebung sowie

alle Ausgaben aus dem Code Ihrer Lambda-Funktion. Jede Nachricht wird mit einem Zeitstempel versehen, auch Ihre benutzerdefinierten Protokolle. Selbst wenn Ihre Funktion keine Ausgabe Ihres Codes protokolliert, werden pro Aufruf mindestens drei Protokollanweisungen generiert (START, END und REPORT):



Aus diesen Protokollen geht Folgendes hervor:

- RequestId— Dies ist eine eindeutige ID, die pro Anfrage generiert wird. Wenn die Lambda-Funktion eine Anfrage erneut versucht, ändert sich diese ID nicht und erscheint in den Protokollen für jeden weiteren Versuch.
- Start/Ende: Diese setzen ein Lesezeichen für einen einzelnen Aufruf, sodass jede Protokollzeile zwischen diesen Aufrufen zu demselben Aufruf gehört.
- Dauer: Die gesamte Aufrufzeit für die Handler-Funktion, ohne `INIT`-Code.
- Dauer der Abrechnung: Wendet die Rundungslogik für Abrechnungszwecke an.
- Speichergröße: Die der Funktion zugewiesene Speichermenge.
- Maximal genutzter Speicher: Der maximale während des Aufrufs genutzte Speicher.
- Initialisierungsdauer: Die Zeit, die benötigt wurde, um den `INIT`-Codeabschnitt außerhalb des Haupthandlers auszuführen.

Zugriffs-Logs mit dem AWS CLI

Das AWS CLI ist ein Open-Source-Tool, mit dem Sie mithilfe von Befehlen in Ihrer Befehlszeilen-Shell mit AWS Diensten interagieren können. Zur Durchführung der Schritte in diesem Abschnitt benötigen Sie AWS CLI Version 2.

Sie können die AWS CLI verwenden, um Protokolle für einen Aufruf mit der `--log-type`-Befehlsoption abzurufen. Die Antwort enthält das Feld `LogResult`, das bis zu 4 KB base64-verschlüsselte Protokolle aus dem Aufruf enthält.

Example eine Log-ID abrufen

Das folgende Beispiel zeigt, wie eine Protokoll-ID aus dem `LogResult`-Feld für eine Funktion namens `my-function` abgerufen wird.

```
aws lambda invoke --function-name my-function out --log-type Tail
```

Die Ausgabe sollte folgendermaßen aussehen:

```
{
    "StatusCode": 200,
    "LogResult":
 "U1RBUlQgUmVxdWVzdElkOiA4N2QwNDRiOC1mMTU0LTExZTgtOGNkYS0yOTc0YzVlNGZiMjEgVmVyc2lvb...",
    "ExecutedVersion": "$LATEST"
}
```

Example entschlüsseln der Protokolle

Verwenden Sie in derselben Eingabeaufforderung das `base64`-Dienstprogramm, um die Protokolle zu entschlüsseln. Das folgende Beispiel zeigt, wie Base64-codierte Logs für abgerufen werde `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
```

Die cli-binary-format Option ist erforderlich, wenn Sie AWS CLI Version 2 verwenden. Um dies zur Standardeinstellung zu machen, führen Sie `aws configure set cli-binary-format raw-in-base64-out` aus. Weitere Informationen finden Sie unter Von AWS CLI unterstützte globale Befehlszeilenoptionen im AWS Command Line Interface -Benutzerhandbuch für Version 2.

Die Ausgabe sollte folgendermaßen aussehen:

```
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
```

Das `base64`-Dienstprogramm ist unter Linux, macOS und Ubuntu auf Windowsverfügbar. macOS-Benutzer müssen möglicherweise `base64 -D` verwenden.

Example get-logs.sh-Skript

Verwenden Sie in derselben Eingabeaufforderung das folgende Skript, um die letzten fünf Protokollereignisse herunterzuladen. Das Skript verwendet `sed` zum Entfernen von Anführungszeichen aus der Ausgabedatei und wechselt 15 Sekunden lang in den Ruhezustand, um Zeit einzuräumen, damit Protokolle verfügbar werden können. Die Ausgabe enthält die Antwort von Lambda und die `get-log-events`Ausgabe des Befehls.

Kopieren Sie den Inhalt des folgenden Codebeispiels und speichern Sie es in Ihrem Lambda-Projektverzeichnis unter `get-logs.sh`.

Die cli-binary-format Option ist erforderlich, wenn Sie AWS CLI Version 2 verwenden. Um dies zur Standardeinstellung zu machen, führen Sie `aws configure set cli-binary-format raw-in-base64-out` aus. Weitere Informationen finden Sie unter Von AWS CLI unterstützte globale Befehlszeilenoptionen im AWS Command Line Interface -Benutzerhandbuch für 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 und Linux (nur diese Systeme)

In derselben Eingabeaufforderung müssen macOS- und Linux-Benutzer möglicherweise den folgenden Befehl ausführen, um sicherzustellen, dass das Skript ausführbar ist.

```
chmod -R 755 get-logs.sh
```

Example die letzten fünf Protokollereignisse abrufen

Führen Sie an derselben Eingabeaufforderung das folgende Skript aus, um die letzten fünf Protokollereignisse abzurufen.

```
./get-logs.sh
```

Die Ausgabe sollte folgendermaßen aussehen:


```
{
    "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"
}
```

Parsing von Protokollen und strukturierte Protokollierung

Mit CloudWatch Logs Insights können Sie Protokolldaten mithilfe einer speziellen Abfragesyntax suchen und analysieren. Es führt Abfragen über mehrere Protokollgruppen durch und bietet eine leistungsstarke Filterung mithilfe von Mustervergleichen mit Glob- und regulären Ausdrücken.

Sie können diese Funktionen nutzen, indem Sie strukturierte Protokollierung in Ihren Lambda-Funktionen implementieren. Strukturierte Protokollierung organisiert Ihre Protokolle in einem vordefinierten Format, wodurch die Abfrage vereinfacht wird. Die Verwendung von Protokollebenen ist ein wichtiger erster Schritt zur Erstellung filterfreundlicher Protokolle, die Informationsmeldungen von Warnungen oder Fehlern trennen. Betrachten Sie beispielsweise den folgenden Node.js-Code:

```
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")
}
```

Die resultierende CloudWatch Protokolldatei enthält ein separates Feld, das die Protokollebene angibt:

```
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
```

Eine CloudWatch Logs Insights-Abfrage kann dann auf Protokollebene filtern. Um beispielsweise nur nach Fehlern zu suchen, können Sie die folgende Abfrage verwenden:

```
fields @timestamp, @message
| filter @message like /ERROR/
| sort @timestamp desc
```

JSON-strukturierte Protokollierung

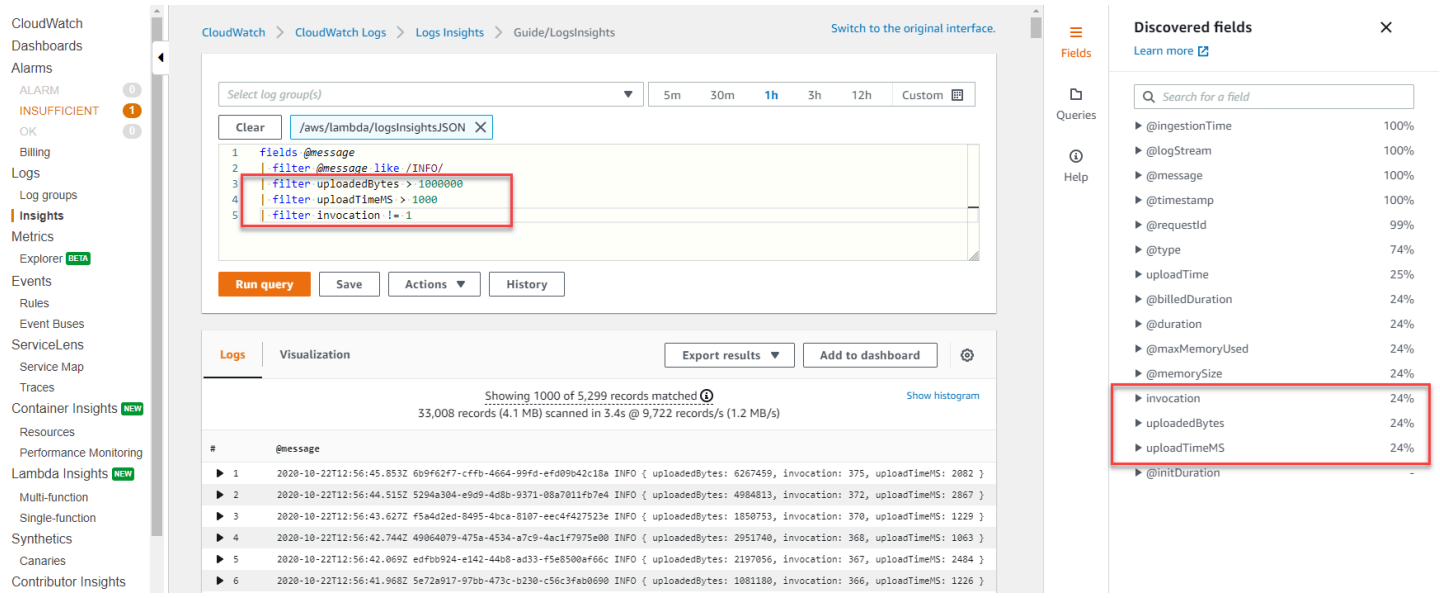
JSON wird häufig verwendet, um Anwendungsprotokolle zu strukturieren. Im folgenden Beispiel wurden die Protokolle in JSON konvertiert, um drei unterschiedliche Werte auszugeben:

```
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
```

Die CloudWatch Logs Insights-Funktion erkennt automatisch Werte in der JSON-Ausgabe und analysiert die Nachrichten als Felder, ohne dass ein benutzerdefinierter Glob oder ein regulärer Ausdruck erforderlich ist. Unter Verwendung der JSON-strukturierten Protokolle findet die folgende Abfrage Aufrufe, bei denen die hochgeladene Datei größer als 1 MB war, die Upload-Zeit mehr als 1 Sekunde betrug und der Aufruf kein Kaltstart war:

```
fields @message
| filter @message like /INFO/
| filter uploadedBytes > 1000000
| filter uploadTimeMS > 1000
| filter invocation != 1
```

Diese Abfrage könnte folgendes Ergebnis liefern:



Die in JSON erkannten Felder werden automatisch im Menü Erkannte Felder auf der rechten Seite ausgefüllt. Den vom Lambda-Dienst ausgegebenen Standardfeldern ist ein „@“ vorangestellt

und Sie können diese Felder auf die gleiche Weise abfragen. Lambda-Logs enthalten immer die Felder @timestamp, @logStream, @message, @requestId, @duration, @billedDuration, @type, @maxMemoryUsed, @memorySize. Wenn X-Ray für eine Funktion aktiviert ist, enthalten die Protokolle auch @ xrayTraceId und @xraySegmentId.

Wenn eine AWS Ereignisquelle wie Amazon S3, Amazon SQS oder Amazon Ihre Funktion EventBridge aufruft, wird der Funktion das gesamte Ereignis als JSON-Objekteingabe zur Verfügung gestellt. Indem Sie dieses Ereignis in der ersten Zeile der Funktion protokollieren, können Sie anschließend mithilfe CloudWatch von Logs Insights jedes der verschachtelten Felder abfragen.

Nützliche Insights-Abfragen

Die folgende Tabelle zeigt Beispiele für Insights-Abfragen, die für die Überwachung von Lambda-Funktionen nützlich sein können.

Description	Beispiel für Abfragesyntax

Die letzten 100 Fehler	`fields Timestamp, LogLevel, Message`
	`| filter LogLevel == "ERR"`
	`| sort @timestamp desc`
	`| limit 100`

Die 100 Aufrufe mit den höchsten Abrechnungen	`filter @type = "REPORT"`
	`| fields @requestId, @billedDuration`
	`| sort by @billedDuration desc`
	`| limit 100`

Prozentualer Anteil der Kaltstarts an der Gesamtzahl der Aufrufe	`filter @type = "REPORT"`
	`| stats sum(strcontains(@message,`
	`"Init Duration"))/count(*) * 100 as`
	`coldStartPct, avg(@duration)`
	`by bin(5m)`

Perzentilbericht zur Lambda-Dauer	`filter @type = "REPORT"`
	`| stats`
	`avg(@billedDuration) as Average,`
	`percentile(@billedDuration, 99)`
	`as NinetyNinth,`

Description	Beispiel für Abfragesyntax
	`percentile(@billedDuration, 95) as NinetyFifth, percentile(@billedDuration, 90) as Ninetieth by bin(30m)`
Perzentilbericht zur Lambda-Speichernutzung	`filter @type="REPORT" | stats avg(@maxMemoryUsed/1024/1024) as mean_MemoryUsed, min(@maxMemoryUsed/1024/1024) as min_MemoryUsed, max(@maxMemoryUsed/1024/1024) as max_MemoryUsed, percentile(@maxMemoryUsed/1024/1024, 95) as Percentile95`
Aufrufe, die 100 % des zugewiesenen Speichers beanspruchen	`filter @type = "REPORT" and @maxMemoryUsed=@memorySize | stats count_distinct(@requestId) by bin(30m)`
Durchschnittlicher Speicherverbrauch bei Aufrufen	`avgMemoryUsedPERC, avg(@billedDuration) as avgDurationMS by bin(5m)`

Description	Beispiel für Abfragesyntax

Visualisierung von Speicherstatistiken	`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)`

Aufrufe, bei denen Lambda beendet wurde	`filter @message like /Process exited/` `| stats count() by bin(30m)`
Aufrufe mit Zeitüberschreitung	`filter @message like /Task timed out/` `| stats count() by bin(30m)`

Latenzbericht	`filter @type = "REPORT"`
	`| stats avg(@duration), max(@dura`
	`tion), min(@duration)`
	`by bin(5m)`

Description	Beispiel für Abfragesyntax
Übermäßig bereitgestellter Arbeitsspeicher	`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`

Protokollvisualisierung und Dashboards

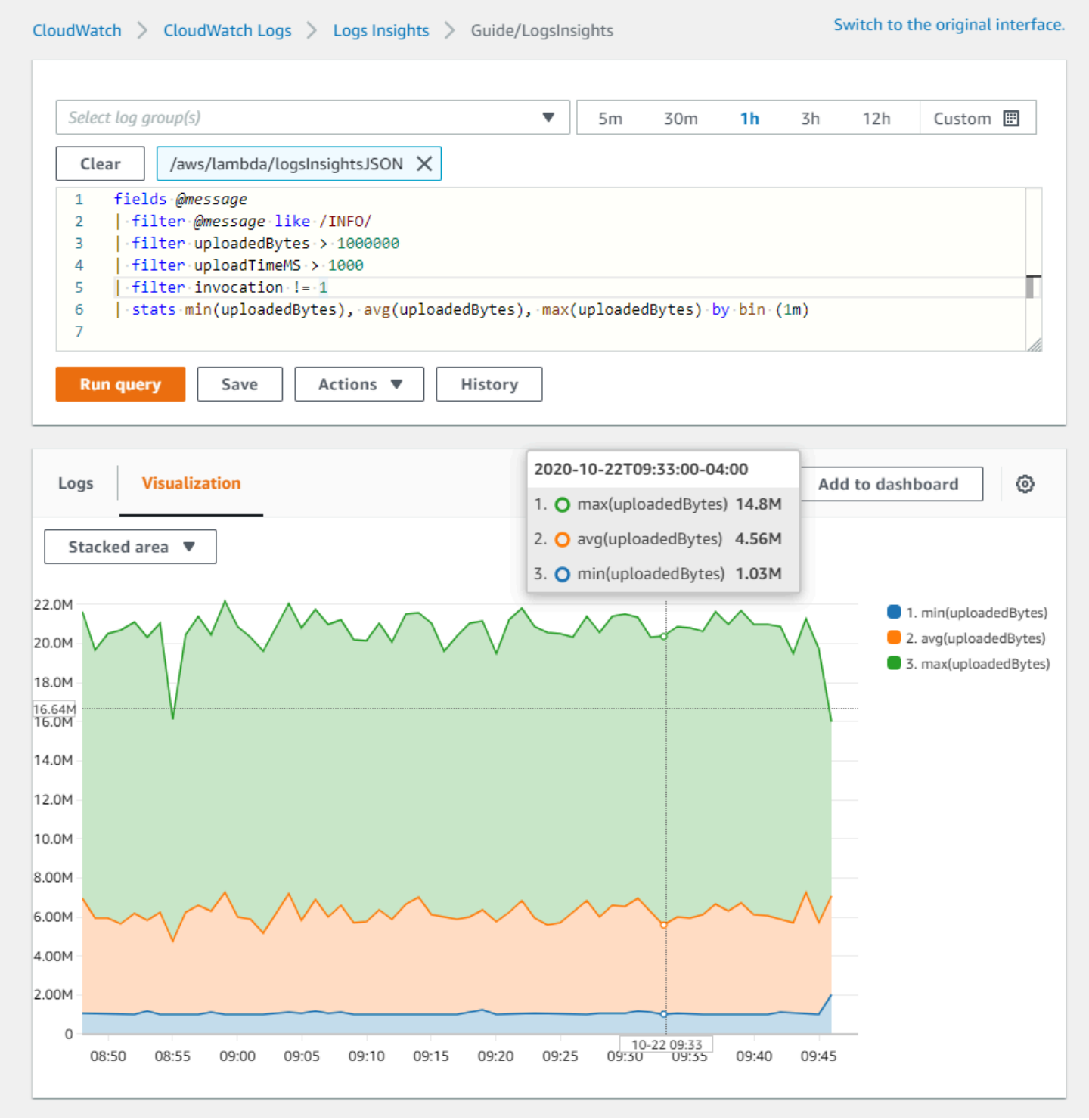
Für jede CloudWatch Logs Insights-Abfrage können Sie die Ergebnisse im Markdown- oder CSV-Format exportieren. In einigen Fällen kann es sinnvoller sein, Visualisierungen aus Abfragen zu erstellen, sofern mindestens eine Aggregationsfunktion vorhanden ist. Mit der `stats`-Funktion können Sie Aggregationen und Gruppierungen definieren.

Das vorherige logInsightsJSON-Beispiel hat nach Upload-Größe und Upload-Zeit gefiltert und erste Aufrufe ausgeschlossen. Dies führte zu einer Datentabelle. Für die Überwachung eines Produktionssystems kann es sinnvoller sein, die minimalen, maximalen und durchschnittlichen Dateigrößen zu visualisieren, um Ausreißer zu erkennen. Wenden Sie dazu die Statistikfunktion mit den erforderlichen Aggregaten an und gruppieren Sie nach einem Zeitwert, z. B. jede Minute:

Nehmen Sie beispielsweise die folgende Abfrage. Dies ist dieselbe Beispielabfrage wie im Abschnitt the section called “JSON-strukturierte Protokollierung”, jedoch mit zusätzlichen Aggregationsfunktionen:

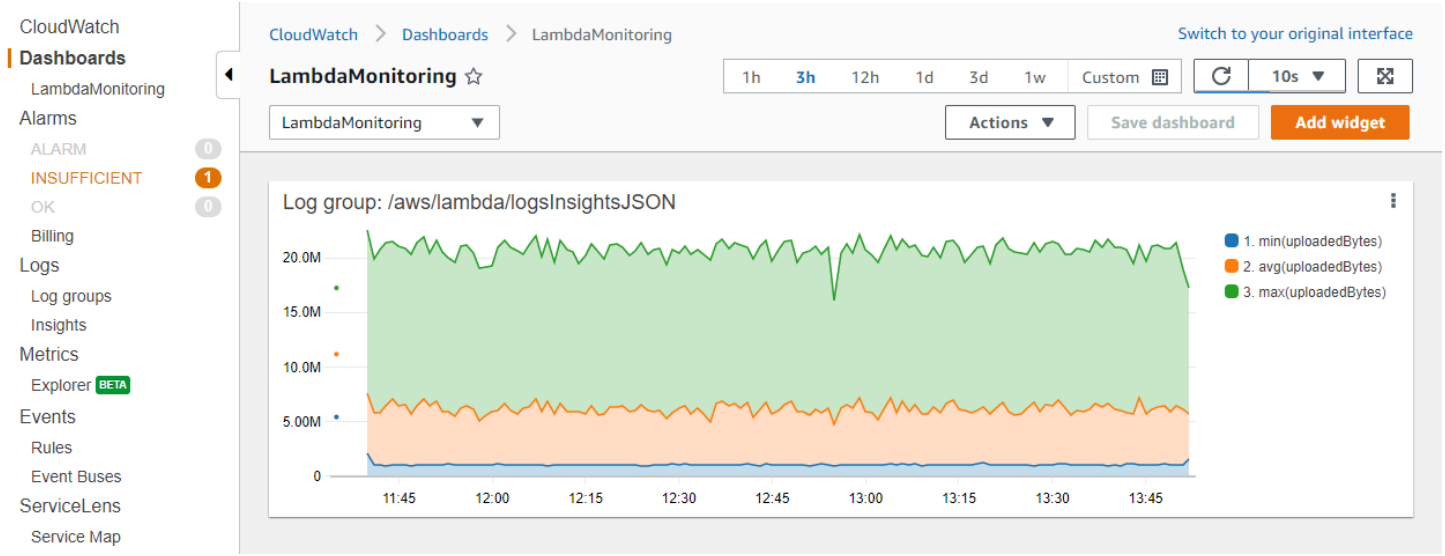
```
fields @message
| filter @message like /INFO/
| filter uploadedBytes > 1000000
| filter uploadTimeMS > 1000
| filter invocation != 1
| stats min(uploadedBytes), avg(uploadedBytes), max(uploadedBytes) by bin (1m)
```

Wir haben diese Aggregate aufgenommen, da es möglicherweise hilfreicher ist, die minimalen, maximalen und durchschnittlichen Dateigrößen zu visualisieren, um Ausreißer zu finden. Sie können die Ergebnisse auf der Registerkarte Visualisierung einsehen:



Nachdem Sie die Erstellung der Visualisierung abgeschlossen haben, können Sie das Diagramm optional zu einem CloudWatch Dashboard hinzufügen. Wählen Sie dazu oberhalb der Visualisierung

die Option Zum Dashboard hinzufügen aus. Dadurch wird die Abfrage als Widget hinzugefügt und Sie können automatische Aktualisierungsintervalle auswählen, was die kontinuierliche Überwachung der Ergebnisse erleichtert:



Senden von Lambda-Funktionsprotokollen an Firehose

Die Lambda-Konsole bietet jetzt die Möglichkeit, Funktionsprotokolle an Firehose zu senden. Dies ermöglicht das Echtzeit-Streaming Ihrer Protokolle an verschiedene von Firehose unterstützte Ziele, einschließlich Analysetools von Drittanbietern und benutzerdefinierten Endpunkten.

> **Note**
>
> Sie können Lambda-Funktionsprotokolle so konfigurieren, dass sie über die Lambda-Konsole, AWS CLI, AWS CloudFormation und alle AWS-SDKs an Firehose gesendet werden.

Preisgestaltung

Informationen zu Preisen finden Sie unter Amazon CloudWatch – Preise.

Erforderliche Berechtigungen für das Firehose-Protokollziel

Wenn Sie die Lambda-Konsole verwenden, um Firehose als Protokollziel Ihrer Funktion zu konfigurieren, benötigen Sie Folgendes:

1. Die erforderlichen IAM-Berechtigungen für die Verwendung von CloudWatch Logs mit Lambda.

2. Sie müssen Abonnementfilter mit Firehose einrichten. Dieser Filter definiert, welche Protokollereignisse für Ihren Firehose-Stream bereitgestellt werden.

Senden von Lambda-Funktionsprotokollen an Firehose

In der Lambda-Konsole können Sie Funktionsprotokolle direkt an Firehose senden, nachdem Sie eine neue Funktion erstellt haben. Führen Sie dazu diese Schritte aus:

1. Melden Sie sich bei der AWS-Managementkonsole an, und öffnen Sie die Lambda-Konsole.
2. Wählen Sie den Namen Ihrer Funktion.
3. Wählen Sie die Registerkarte Konfiguration aus.
4. Wählen Sie die Registerkarte Überwachungs- und Betriebs-Tools aus.
5. Wählen Sie im Abschnitt „Protokollierungskonfiguration“ die Option Bearbeiten aus.
6. Wählen Sie im Abschnitt „Protokollinhalt“ ein Protokollformat aus.
7. Führen Sie im Abschnitt „Protokollziel“ die folgenden Schritte aus:
   a. Wählen Sie einen Zielservice aus.
   b. Wählen Sie Neue Protokollgruppe erstellen oder Vorhandene Protokollgruppe aus.

      > **Note**
      >
      > Wenn Sie eine vorhandene Protokollgruppe für ein Firehose-Ziel auswählen, stellen Sie sicher, dass es sich bei der ausgewählten Protokollgruppe um einen `Delivery`-Protokollgruppentyp handelt.

   c. Wählen Sie einen Firehose-Stream aus.
   d. Die Protokollgruppe für CloudWatch `Delivery` wird angezeigt.
8. Wählen Sie Speichern.

> **Note**
>
> Wenn die in der Konsole angegebene IAM-Rolle nicht über die erforderlichen Berechtigungen verfügt, schlägt die Einrichtung des Ziels fehl. Um dieses Problem zu beheben, finden Sie unter „Erforderliche Berechtigungen für das Firehose-Protokollziel“ die erforderlichen Berechtigungen.

Kontenübergreifende Protokollierung

Sie können Lambda so konfigurieren, dass Protokolle in einem anderen AWS-Konto an den Firehose-Bereitstellungsstream gesendet werden. Dies erfordert die Einrichtung eines Ziels und die Konfiguration der entsprechenden Berechtigungen in beiden Konten.

Detaillierte Anweisungen zur Einrichtung der kontoübergreifenden Protokollierung, einschließlich der erforderlichen IAM-Rollen und -Richtlinien, finden Sie unter Einrichten eines neuen kontoübergreifenden Abonnements in der CloudWatch-Logs-Dokumentation.

Senden von Lambda-Funktionsprotokollen an Amazon S3

Sie können Ihre Lambda-Funktion so konfigurieren, dass Protokolle mithilfe der Lambda-Konsole direkt an Amazon S3 gesendet werden. Dieses Feature bietet eine kostengünstige Lösung für die langfristige Speicherung von Protokollen und ermöglicht leistungsstarke Analyseoptionen mithilfe von Services wie Athena.

> **Note**
>
> Sie können Lambda-Funktionsprotokolle so konfigurieren, dass sie über die Lambda-Konsole, AWS CLI, AWS CloudFormation und alle AWS-SDKs an Amazon S3 gesendet werden.

Preisgestaltung

Informationen zu Preisen finden Sie unter Amazon CloudWatch – Preise.

Erforderliche Berechtigungen für das Amazon-S3-Protokollziel

Wenn Sie die Lambda-Konsole verwenden, um Amazon S3 als Protokollziel Ihrer Funktion zu konfigurieren, benötigen Sie Folgendes:

1. Die erforderlichen IAM-Berechtigungen für die Verwendung von CloudWatch Logs mit Lambda.
2. Zu … Einrichten eines CloudWatch-Logs-Abonnementfilters, um Lambda-Funktionsprotokolle an Amazon S3 zu senden. Dieser Filter definiert, welche Protokollereignisse für Ihren Amazon-S3-Bucket bereitgestellt werden.

Einrichten eines CloudWatch-Logs-Abonnementfilters, um Lambda-Funktionsprotokolle an Amazon S3 zu senden

Um Protokolle von CloudWatch Logs an Amazon S3 zu senden, müssen Sie einen Abonnementfilter erstellen. Dieser Filter definiert, welche Protokollereignisse für Ihren Amazon-S3-Bucket bereitgestellt werden. Ihr Amazon-S3-Bucket muss sich in derselben Region befinden wie Ihre Protokollgruppe.

So erstellen Sie einen Abonnementfilter für Amazon S3

1. Erstellen Sie einen Amazon-Simple-Storage-Service-(Amazon S3)-Bucket. Wir empfehlen, dass Sie einen Bucket verwenden, der speziell für CloudWatch Logs erstellt wurde. Wenn Sie jedoch einen vorhandenen Bucket verwenden möchten, gehen Sie direkt zu Schritt 2.

   Führen Sie den folgenden Befehl aus, und ersetzen Sie die Platzhalter-Region mit der Region, die Sie verwenden möchten:

   ```
   aws s3api create-bucket --bucket amzn-s3-demo-bucket2 --create-bucket-configuration
    LocationConstraint=region
   ```

   > **Note**
   >
   > `amzn-s3-demo-bucket2` ist ein Beispiel für den Namen eines Amazon-S3-Buckets. Er ist reserviert. Damit dieses Verfahren funktioniert, müssen Sie ihn durch den eindeutigen Namen Ihres Amazon-S3-Buckets ersetzen.

   Das Folgende ist Ausgabebeispiel:

   ```
   {
       "Location": "/amzn-s3-demo-bucket2"
   }
   ```

2. Erstellen Sie die IAM-Rolle, die CloudWatch Logs die Berechtigung erteilt, Daten an Ihren Amazon-S3-Bucket zu senden. Diese Richtlinie enthält einen globalen Bedingungskontextschlüssel vom Typ „aws:SourceArn“, um das Confused-Deputy-Problem zu vermeiden. Weitere Informationen finden Sie unter Confused-Deputy-Prävention.

   a. Verwenden Sie einen Text-Editor zum Erstellen einer Vertrauensrichtlinie in einer Datei `~/TrustPolicyForCWL.json` wie folgt:

```
{
    "Statement": {
        "Effect": "Allow",
        "Principal": { "Service": "logs.amazonaws.com" },
        "Condition": {
            "StringLike": {
                "aws:SourceArn": "arn:aws:logs:region:123456789012:*"
            }
         },
        "Action": "sts:AssumeRole"
    }
}
```

b. Verwenden Sie den Befehl create-role, um die IAM-Rolle zu erstellen und die Datei mit der Vertrauensrichtlinie anzugeben. Notieren Sie den zurückgegebenen Wert Role.Arn, da Sie ihn in einem späteren Schritt benötigen:

```
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. Erstellen Sie eine Berechtigungsrichtlinie, um zu definieren, welche Aktionen CloudWatch Logs an Ihrem Konto durchführen darf. Verwenden Sie zunächst einen Text-Editor zum Erstellen einer Berechtigungsrichtlinie in einer Datei `~/PermissionsForCWL.json`:

```
{
  "Statement": [
    {
      "Effect": "Allow",
      "Action": ["s3:PutObject"],
      "Resource": ["arn:aws:s3:::amzn-s3-demo-bucket2/*"]
    }
  ]
}
```

   Verknüpfen Sie die Berechtigungsrichtlinie mit dem folgenden Befehl `put-role-policy` mit der Rolle:

```
aws iam put-role-policy --role-name CWLtoS3Role --policy-name Permissions-Policy-For-S3 --policy-document file://~/PermissionsForCWL.json
```

4. Erstellen Sie eine `Delivery`-Protokollgruppe oder verwenden Sie eine vorhandene `Delivery`-Protokollgruppe.

```
aws logs create-log-group --log-group-name my-logs --log-group-class DELIVERY --region REGION_NAME
```

5. `PutSubscriptionFilter` zum Einrichten des Ziels

```
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
```

Senden von Lambda-Funktionsprotokollen an Amazon S3

In der Lambda-Konsole können Sie Funktionsprotokolle direkt an Amazon S3 senden, nachdem Sie eine neue Funktion erstellt haben. Führen Sie dazu diese Schritte aus:

1. Melden Sie sich bei der AWS-Managementkonsole an, und öffnen Sie die Lambda-Konsole.
2. Wählen Sie den Namen Ihrer Funktion.
3. Wählen Sie die Registerkarte Konfiguration aus.
4. Wählen Sie die Registerkarte Überwachungs- und Betriebstools aus.
5. Wählen Sie im Abschnitt „Protokollierungskonfiguration“ die Option Bearbeiten aus.
6. Wählen Sie im Abschnitt „Protokollinhalt“ ein Protokollformat aus.
7. Führen Sie im Abschnitt „Protokollziel“ die folgenden Schritte aus:
   a. Wählen Sie einen Zielservice aus.
   b. Wählen Sie Neue Protokollgruppe erstellen oder Vorhandene Protokollgruppe aus.

      > **Note**
      >
      > Wenn Sie eine vorhandene Protokollgruppe für ein Amazon-S3-Ziel auswählen, stellen Sie sicher, dass es sich bei der ausgewählten Protokollgruppe um einen `Delivery`-Protokollgruppentyp handelt.

   c. Wählen Sie einen Amazon-S3-Bucket als Zielort für Ihre Funktionsprotokolle.
   d. Die Protokollgruppe für CloudWatch `Delivery` wird angezeigt.
8. Wählen Sie Speichern.

> **Note**
>
> Wenn die in der Konsole angegebene IAM-Rolle nicht über die erforderlichen Berechtigungen verfügt, schlägt die Einrichtung des Ziels fehl. Informationen zur Behebung dieses Problems finden Sie unter Erforderliche Berechtigungen für das Amazon-S3-Protokollziel.

Kontoübergreifende Protokollierung

Sie können Lambda so konfigurieren, dass Protokolle in einem anderen AWS-Konto an ein Amazon-S3-Bucket gesendet werden. Dies erfordert die Einrichtung eines Ziels und die Konfiguration der entsprechenden Berechtigungen in beiden Konten.

Detaillierte Anweisungen zur Einrichtung der kontoübergreifenden Protokollierung, einschließlich der erforderlichen IAM-Rollen und -Richtlinien, finden Sie unter Einrichten eines neuen kontoübergreifenden Abonnements in der CloudWatch-Logs-Dokumentation.

Protokollieren von AWS Lambda-API-Aufrufen mit AWS CloudTrail

AWS Lambda ist in AWS CloudTrail integriert, einem Service, der eine Aufzeichnung der Aktionen bereitstellt, die von einem Benutzer, einer Rolle oder einem AWS-Service ausgeführt werden. CloudTrail erfasst API-Aufrufe für Lambda als Ereignisse. Zu den erfassten Aufrufen gehören Aufrufe über die Lambda-Konsole und Codeaufrufe der Lambda-API-Operationen. Mit den von CloudTrail erfassten Informationen können Sie die an Lambda gestellte Anfrage, die IP-Adresse, von der die Anfrage gestellt wurde, den Initiator der Anfrage, den Zeitpunkt der Anfrage und zusätzliche Details bestimmen.

Jeder Ereignis- oder Protokolleintrag enthält Informationen zu dem Benutzer, der die Anforderung generiert hat. Die Identitätsinformationen unterstützen Sie bei der Ermittlung der folgenden Punkte:

- Ob die Anfrage mit Anmeldeinformationen des Root-Benutzers oder des Benutzers gestellt wurde.
- Die Anforderung wurde im Namen eines IAM-Identity-Center-Benutzers erstellt.
- Gibt an, ob die Anforderung mit temporären Sicherheitsanmeldeinformationen für eine Rolle oder einen Verbundbenutzer gesendet wurde.
- Ob die Anforderung aus einem anderen gesendet wurde AWS-Service.

Bei Kontoerstellung ist CloudTrail standardmäßig in AWS-Konto aktiviert und Sie haben automatisch Zugriff auf den CloudTrail-Ereignisverlauf. Der CloudTrail-Ereignisverlauf stellt eine anzeigbare, durchsuchbare, herunterladbare und unveränderliche Aufzeichnung der Verwaltungsereignisse der letzten 90 Tage in einer AWS-Region bereit. Weitere Informationen finden Sie unter Arbeiten mit dem CloudTrail-Ereignisverlauf im AWS CloudTrail-Benutzerhandbuch. Für die Anzeige des Eventverlaufs fallen keine CloudTrail-Gebühren an.

Erstellen Sie einen Trail oder einen Ereignisdatenspeicher in CloudTrail Lake, um Aktivitäten und Ereignisse in Ihrer AWS-Konto über 90 Tage hinaus fortlaufend aufzuzeichnen.

CloudTrail-Trails

> Ein Trail ermöglicht es CloudTrail, Protokolldateien in einem Amazon-S3-Bucket bereitzustellen. Trails, die mit der AWS-Managementkonsole erstellt wurden, sind multiregional. Sie können mithilfe von AWS CLI einen Einzel-Region- oder einen Multi-Region-Trail erstellen. Das Erstellen eines Multi-Region-Trails wird empfohlen, da Sie so die Aktivitäten in allen AWS-Regionen in Ihrem Konto erfassen. Wenn Sie einen Einzel-Region-Trail erstellen, können Sie nur die Ereignisse anzeigen, die im AWS-Region des Trails protokolliert wurden. Weitere Informationen

zu Trails finden Sie unter Erstellen eines Trails für Ihr AWS-Konto und Erstellen eines Trails für eine Organisation im AWS CloudTrail-Benutzerhandbuch.

Sie können eine Kopie Ihrer laufenden Verwaltungsereignisse kostenlos von CloudTrail an Ihren Amazon-S3-Bucket senden, indem Sie einen Trail erstellen. Es fallen jedoch Speichergebühren für Amazon S3 an. Weitere Informationen zu CloudTrail-Preisen finden Sie unter AWS CloudTrail – Preise. Informationen zu Amazon-S3-Preisen finden Sie unter Amazon S3 – Preise.

CloudTrail-Lake-Ereignisdatenspeicher

Mit CloudTrail Lake können Sie SQL-basierte Abfragen zu Ihren Ereignissen ausführen. CloudTrail Lake konvertiert vorhandene Ereignisse im zeilenbasierten JSON-Format in das Apache ORC-Format. ORC ist ein spaltenförmiges Speicherformat, das für den schnellen Abruf von Daten optimiert ist. Die Ereignisse werden in Ereignisdatenspeichern zusammengefasst, bei denen es sich um unveränderliche Sammlungen von Ereignissen handelt, die auf Kriterien basieren, die Sie mit Hilfe von erweiterten Ereignisselektoren auswählen. Die Selektoren, die Sie auf einen Ereignisdatenspeicher anwenden, steuern, welche Ereignisse bestehen bleiben und für Sie zur Abfrage verfügbar sind. Weitere Informationen zu CloudTrail Lake finden Sie unter Arbeiten mit AWS CloudTrail-Lake im AWS CloudTrail-Benutzerhandbuch.

Für CloudTrail-Lake-Ereignisdatenspeicher und -abfragen fallen Gebühren an. Beim Erstellen eines Ereignisdatenspeichers wählen Sie die Preisoption aus, die für den Ereignisdatenspeicher genutzt werden soll. Die Preisoption bestimmt die Kosten für die Erfassung und Speicherung von Ereignissen sowie die standardmäßige und maximale Aufbewahrungsdauer für den Ereignisdatenspeicher. Weitere Informationen zu CloudTrail-Preisen finden Sie unter AWS CloudTrail – Preise.

Lambda-Datenereignisse in CloudTrail

Datenereignisse liefern Informationen über die Ressourcenoperationen, die auf oder in einer Ressource ausgeführt werden (z. B. Lesen oder Schreiben in ein Amazon-S3-Objekt). Sie werden auch als Vorgänge auf der Datenebene bezeichnet. Datenereignisse sind oft Aktivitäten mit hohem Volume. Standardmäßig protokolliert CloudTrail die meisten Datenereignisse nicht und der CloudTrail-Ereignisverlauf zeichnet sie nicht auf.

Ein CloudTrail-Datenereignis, das standardmäßig für unterstützte Dienste protokolliert wird, ist `LambdaESMDisabled`. Weitere Informationen zur Verwendung dieses Ereignisses zur Behebung von Problemen mit Lambda-Zuordnungen von Ereignisquellen finden Sie unter the section called “Verwenden von CloudTrail zur Problembehandlung deaktivierter Lambda-Ereignisquellen”.

Für Datenereignisse werden zusätzliche Gebühren fällig. Weitere Informationen zu CloudTrail-Preisen finden Sie unter AWS CloudTrail – Preise.

Sie können Datenereignisse für den Ressourcentyp `AWS::Lambda::Function` über die CloudTrail-Konsole, AWS CLI oder CloudTrail-API-Vorgänge protokollieren. Weitere Informationen zum Protokollieren von Datenereignissen finden Sie unter Protokollieren von Datenereignissen mit dem AWS-Managementkonsole und Protokollieren von Datenereignissen mit dem AWS Command Line Interface im AWS CloudTrail-Benutzerhandbuch.

In der folgenden Tabelle ist der Lambda-Ressourcentyp aufgeführt, für die Sie Datenereignisse protokollieren können. In der Spalte Datenereignistyp (Konsole) wird der Wert angezeigt, der in der CloudTrail-Konsole aus der Liste Datenereignistyp ausgewählt werden kann. In der Wertspalte resources.type wird der `resources.type` Wert angezeigt, den Sie angeben würden, wenn Sie erweiterte Event-Selektoren mithilfe der APIs AWS CLI oder CloudTrail konfigurieren würden. In der Spalte Daten-APIs, die in CloudTrail protokolliert wurden, werden die API-Aufrufe angezeigt, die für den Ressourcentyp in CloudTrail protokolliert wurden.

Typ des Datenereignisses (Konsole)	resources.type-Wert	Bei CloudTrail protokollierte Daten-APIs
Lambda	`AWS::Lambda::Function`	Aufrufen

Sie können erweiterte Event-Selektoren so konfigurieren, dass sie nach den Feldern `eventName`, `readOnly` und `resources.ARN` filtern, sodass nur die Ereignisse protokolliert werden, die für Sie wichtig sind. Das folgende Beispiel ist die JSON-Ansicht einer Datenereigniskonfiguration, die nur Ereignisse für eine bestimmte Funktion protokolliert. Weitere Informationen zu diesen Kontingenten finden Sie unter AdvancedFieldSelector in derAWS CloudTrail-API-Referenz.

```
[
  {
    "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"
        ]
      }
    ]
  }
]
```

Lambda-Verwaltungsereignisse in CloudTrail

Verwaltungsereignisse liefern Informationen zu Verwaltungsvorgängen, die für Ressourcen in Ihrem AWS-Konto ausgeführt werden. Sie werden auch als Vorgänge auf Steuerebene bezeichnet. CloudTrail protokolliert standardmäßig Verwaltungsereignisse.

Lambda unterstützt die Protokollierung der folgenden Aktionen als Verwaltungsereignisse in CloudTrail-Protokolldateien.

> **Note**
>
> In der CloudTrail-Protokolldatei kann das `eventName` zwar Datums- und Versionsinformationen enthalten, aber es bezieht sich immer noch auf dieselbe öffentliche API-Aktion. Zum Beispiel erscheint die Aktion `GetFunction` als `GetFunction20150331v2`. Der folgende Liste gibt an, wann sich der Ereignisname vom API-Aktionsnamen unterscheidet.

- AddLayerVersionPermission
- AddPermission (Name des Ereignisses: `AddPermission20150331v2`)
- CreateAlias (Ereignisname: `CreateAlias20150331`)
- CreateEventSourceMapping (Ereignisname: `CreateEventSourceMapping20150331`)
- CreateFunction (Ereignisname:`CreateFunction20150331`)

(Die `Environment`- und `ZipFile`-Parameter fließen nicht in die CloudTrail-Protokolle für `CreateFunction` mit ein.)

- CreateFunctionUrlConfig
- DeleteAlias (Ereignisname: `DeleteAlias20150331`)
- DeleteCodeSigningConfig
- DeleteEventSourceMapping (Ereignisname: `DeleteEventSourceMapping20150331`)
- DeleteFunction (Ereignisname: `DeleteFunction20150331`)
- DeleteFunctionConcurrency (Ereignisname: `DeleteFunctionConcurrency20171031`)
- DeleteFunctionUrlConfig
- DeleteProvisionedConcurrencyConfig
- GetAlias (Name des Ereignisses:`GetAlias20150331`)
- GetEventSourceMapping
- GetFunction
- GetFunctionUrlConfig
- GetFunctionConfiguration
- GetLayerVersionPolicy
- GetPolicy
- ListEventSourceMappings
- ListFunctions
- ListFunctionUrlConfigs
- PublishLayerVersion (Ereignisname: `PublishLayerVersion20181031`)

  (Der `ZipFile`-Parameter fließt nicht in die CloudTrail-Protokolle für `PublishLayerVersion` mit ein.)
- PublishVersion (Name des Ereignisses: `PublishVersion20150331`)
- PutFunctionConcurrency (Ereignisname: `PutFunctionConcurrency20171031`)
- PutFunctionCodeSigningConfig
- PutFunctionEventInvokeConfig
- PutProvisionedConcurrencyConfig
- PutRuntimeManagementConfig
- RemovePermission (Name des Ereignisses: `RemovePermission20150331v2`)

- TagResource (Name des Ereignisses: `TagResource20170331v2`)
- UntagResource (Name des Ereignisses: `UntagResource20170331v2`)
- UpdateAlias (Ereignisname: `UpdateAlias20150331`)
- UpdateCodeSigningConfig
- UpdateEventSourceMapping (Ereignisname: `UpdateEventSourceMapping20150331`)
- UpdateFunctionCode (Ereignisname: `UpdateFunctionCode20150331v2`)

  (Der `ZipFile`-Parameter fließt nicht in die CloudTrail-Protokolle für `UpdateFunctionCode` mit ein.)
- UpdateFunctionConfiguration (Ereignisname: `UpdateFunctionConfiguration20150331v2`)

  (Der `Environment`-Parameter fließt nicht in die CloudTrail-Protokolle für `UpdateFunctionConfiguration` mit ein.)
- UpdateFunctionEventInvokeConfig
- UpdateFunctionUrlConfig

Verwenden von CloudTrail zur Problembehandlung deaktivierter Lambda-Ereignisquellen

Wenn Sie den Status einer Zuordnung von Ereignisquellen mithilfe der API-Aktion UpdateEventSourceMapping ändern, wird der API-Aufruf als Verwaltungsereignis in CloudTrail protokolliert. Zuordnungen von Ereignisquellen können aufgrund von Fehlern auch direkt in den `Disabled`-Status übergehen.

Für die folgenden Dienste veröffentlicht Lambda das `LambdaESMDisabled`-Datenereignis in CloudTrail, wenn Ihre Ereignisquelle in den Status Deaktiviert übergeht:

- Amazon Simple Queue Service (Amazon SQS)
- Amazon DynamoDB
- Amazon Kinesis

Lambda unterstützt dieses Ereignis nicht für andere Ereignisquellen-Zuordnungstypen.

Um Warnungen zu erhalten, wenn Zuordnungen von Ereignisquellen für unterstützte Services in den `Disabled`-Status übergehen, richten Sie einen Alarm in Amazon CloudWatch ein, indem Sie

das `LambdaESMDisabled`-CloudTrail-Ereignis verwenden. Um mehr über das Einrichten eines CloudWatch-Alarms zu erfahren, siehe Erstellen von CloudWatch-Alarmen für CloudTrail-Ereignisse: Beispiele.

Die `serviceEventDetails`-Entität in der `LambdaESMDisabled`-Ereignismeldung enthält einen der folgenden Fehlercodes.

`RESOURCE_NOT_FOUND`

Die in der Anforderung angegebene Ressource ist nicht vorhanden.

`FUNCTION_NOT_FOUND`

Die an die Ereignisquelle angefügte Funktion ist nicht vorhanden.

`REGION_NAME_NOT_VALID`

Ein Regionsname, der der Ereignisquelle oder -funktion zur Verfügung gestellt wird, ist ungültig.

`AUTHORIZATION_ERROR`

Es wurden keine Berechtigungen festgelegt oder sie wurden falsch konfiguriert.

`FUNCTION_IN_FAILED_STATE`

Der Funktionscode wird nicht kompiliert, hat eine nicht wiederherstellbare Ausnahme festgestellt oder eine fehlerhafte Bereitstellung ist aufgetreten.

Beispiele für Lambda-Ereignisse

Ein Ereignis stellt eine einzelne Anforderung aus einer beliebigen Quelle dar und enthält Informationen über die angeforderte API-Aktion, das Datum und die Uhrzeit der Operation, Anforderungsparameter usw. CloudTrail-Protokolldateien sind kein geordnetes Stack-Trace der öffentlichen API-Aufrufe und Ereignisse und erscheinen daher in keiner bestimmten Reihenfolge.

Das folgende Beispiel zeigt den CloudTrail-Protokolleintrag für die Aktionen , `GetFunction` und `DeleteFunction`.

> **Note**
>
> Die `eventName` kann Datums- und Versionsinformationen enthalten, wie z. B. die `"GetFunction20150331"`, aber sie bezieht sich immer noch auf dieselbe öffentliche API.

```
{
  "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"
    }
  ]
}
```

Informationen zu den Inhalten von CloudTrail-Datensätzen finden Sie unter CloudTrail-Datensatzinhalte im AWS CloudTrail-Benutzerhandbuch.

Visualisieren Sie Lambda-Funktionsaufrufe mit AWS X-Ray

Sie können AWS X-Ray es verwenden, um die Komponenten Ihrer Anwendung zu visualisieren, Leistungsengpässe zu identifizieren und Anfragen zu beheben, die zu einem Fehler geführt haben. Ihre Lambda-Funktionen senden Ablaufverfolgungs-Daten an X-Ray und X-Ray verarbeitet die Daten, um eine Service-Map und durchsuchbare Ablaufverfolgungs-Zusammenfassungen zu generieren.

Lambda unterstützt zwei Nachverfolgungs-Modi für X-Ray: `Active` und `PassThrough`. Mit der `Active`-Nachverfolgung erstellt Lambda automatisch Nachverfolgungssegmente für Funktionsaufrufe und sendet diese an X-Ray. Der `PassThrough`-Modus hingegen leitet den Nachverfolgungskontext einfach an Downstream-Services weiter. Wenn Sie die `Active`-Nachverfolgung für Ihre Funktion aktiviert haben, sendet Lambda automatisch Nachverfolgungen für Stichprobenanfragen an X-Ray. In der Regel entscheidet ein Upstream-Service, wie Amazon API Gateway oder eine auf Amazon gehostete Anwendung EC2 , die mit dem X-Ray SDK ausgestattet ist, ob eingehende Anfragen verfolgt werden sollen, und fügt dann diese Stichprobenentscheidung als Tracing-Header hinzu. Lambda verwendet diesen Header, um zu entscheiden, ob Nachverfolgungen gesendet werden sollen oder nicht. Traces von Upstream-Nachrichtenproduzenten wie Amazon SQS werden automatisch mit Traces von nachgeschalteten Lambda-Funktionen verknüpft, sodass eine end-to-end Ansicht der gesamten Anwendung entsteht. Weitere Informationen finden Sie unter Ablaufverfolgung ereignisgesteuerter Anwendungen im AWS X-Ray -Entwicklerhandbuch.

> **Note**
>
> X-Ray Tracing wird derzeit nicht für Lambda-Funktionen mit Amazon Managed Streaming für Apache Kafka (Amazon MSK), selbstverwaltetem Apache Kafka, Amazon MQ mit ActiveMQ und RabbitMQ oder Zuordnungen von Amazon-DocumentDB-Ereignisquellen unterstützt.

Gehen Sie folgendermaßen vor, um die aktive Nachverfolgung Ihrer Lambda-Funktion mit der Konsole umzuschalten:

So aktivieren Sie die aktive Nachverfolgung

1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.
2. Wählen Sie eine Funktion aus.
3. Wählen Sie Configuration (Konfiguration) und dann Monitoring and operations tools (Überwachungs- und Produktionstools).
4. Wählen Sie unter Zusätzliche Überwachungstools die Option Bearbeiten aus.

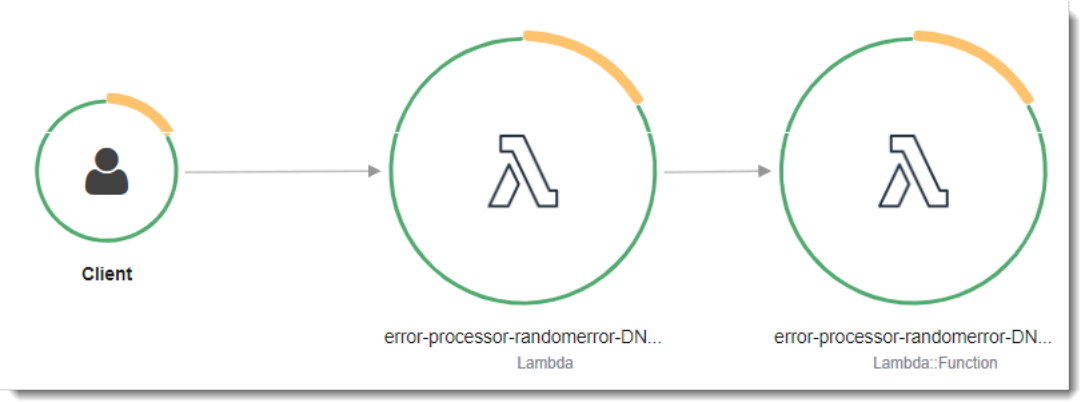
5. Wählen Sie unter CloudWatch Anwendungssignale und AWS X-Ray die Option Enable for Lambda Service Traces aus.
6. Wählen Sie Speichern.

Ihre Funktion benötigt die Berechtigung zum Hochladen von Trace-Daten zu X-Ray. Wenn Sie die aktive Nachverfolgung in der Lambda-Konsole aktivieren, fügt Lambda der Ausführungsrolle Ihrer Funktion die erforderlichen Berechtigungen hinzu. Andernfalls fügen Sie die AWSXRayDaemonWriteAccessRichtlinie der Ausführungsrolle hinzu.

X-Ray verfolgt nicht alle Anfragen an Ihre Anwendung nach. X-Ray wendet einen Sampling-Algorithmus an, um sicherzustellen, dass die Nachverfolgung effizient ist, und stellt dennoch ein repräsentatives Beispiel aller Anfragen bereit. Die Samplingrate beträgt 1 Anforderung pro Sekunde und 5 Prozent aller weiteren Anforderungen. Sie können die X-Ray-Samplingrate nicht für Ihre Funktionen konfigurieren.

Grundlegendes zu X-Ray-Ablaufverfolgungen

In X-Ray, zeichnet eine Ablaufverfolgung Informationen zu einer Anforderung auf, die von einem oder mehreren Services verarbeitet wird. Lambda zeichnet 2 Segmente pro Trace auf, wodurch zwei Knoten im Dienstgraphen entstehen. In der folgenden Abbildung werden diese beiden Knoten hervorgehoben:



Der erste Knoten auf der linken Seite stellt den Lambda-Service dar, der die Aufrufanforderung empfängt. Der zweite Knoten stellt Ihre spezifische Lambda-Funktion dar.

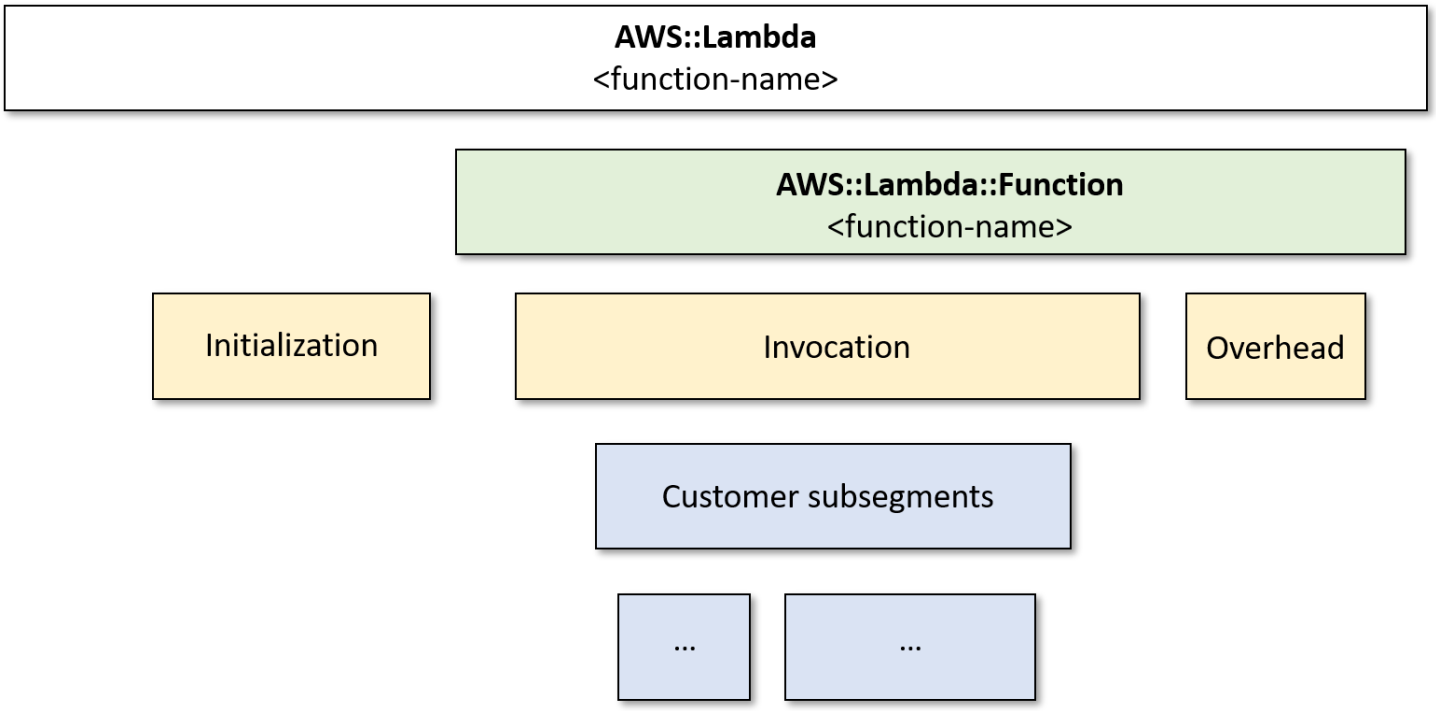
Das für den Lambda-Dienst `AWS::Lambda` aufgezeichnete Segment umfasst alle Schritte, die zur Vorbereitung der Lambda-Ausführungsumgebung erforderlich sind. Dazu gehören das Scheduling der MicroVM, das Erstellen oder Aufheben des Einfrierens einer Ausführungsumgebung mit den von Ihnen konfigurierten Ressourcen sowie das Herunterladen Ihres Funktionscodes und aller Schichten.

Das `AWS::Lambda::Function`-Segment bezieht sich auf die Arbeit, die von der Funktion geleistet wird.

> **Note**
>
> AWS implementiert derzeit Änderungen am Lambda-Service. Aufgrund dieser Änderungen können geringfügige Unterschiede in Struktur und Inhalt der Systemprotokollmeldungen und Trace-Segmente auftreten, die von verschiedenen Lambda-Funktionen in Ihrem AWS-Konto. Diese Änderung betrifft die Teilsegmente des Funktionssegments. In den folgenden Abschnitten werden sowohl das alte als auch das neue Format für diese Teilsegmente beschrieben.
> Diese Änderungen werden in den kommenden Wochen umgesetzt, und alle Funktionen AWS-Regionen außer China und den GovCloud Regionen werden auf die Verwendung der Protokollnachrichten und Trace-Segmente im neuen Format umgestellt.

AWS X-Ray Lambda-Segmentstruktur im alten Stil

Die herkömmliche X-Ray-Struktur für das `AWS::Lambda`-Segment sieht wie folgt aus:



In diesem Format hat das Funktionssegment Untersegmente für `Initialization`, `Invocation` und `Overhead`. Nur für Lambda SnapStart gibt es auch ein Teilsegment `Restore` (in dieser Abbildung nicht dargestellt).

Das `Initialization`-Teilsegment stellt die Init-Phase des Lebenszyklus der Lambda-Ausführungsumgebung dar. In dieser Phase initialisiert Lambda Erweiterungen, initialisiert die Laufzeit und führt den Initialisierungscode der Funktion aus.

Das `Invocation`-Teilsegment stellt die Aufrufphase dar, in welcher der Lambda-Funktionshandler aufgerufen wird. Dies beginnt mit der Laufzeit- und Erweiterungsregistrierung und endet, wenn die Laufzeit bereit ist, die Antwort zu senden.

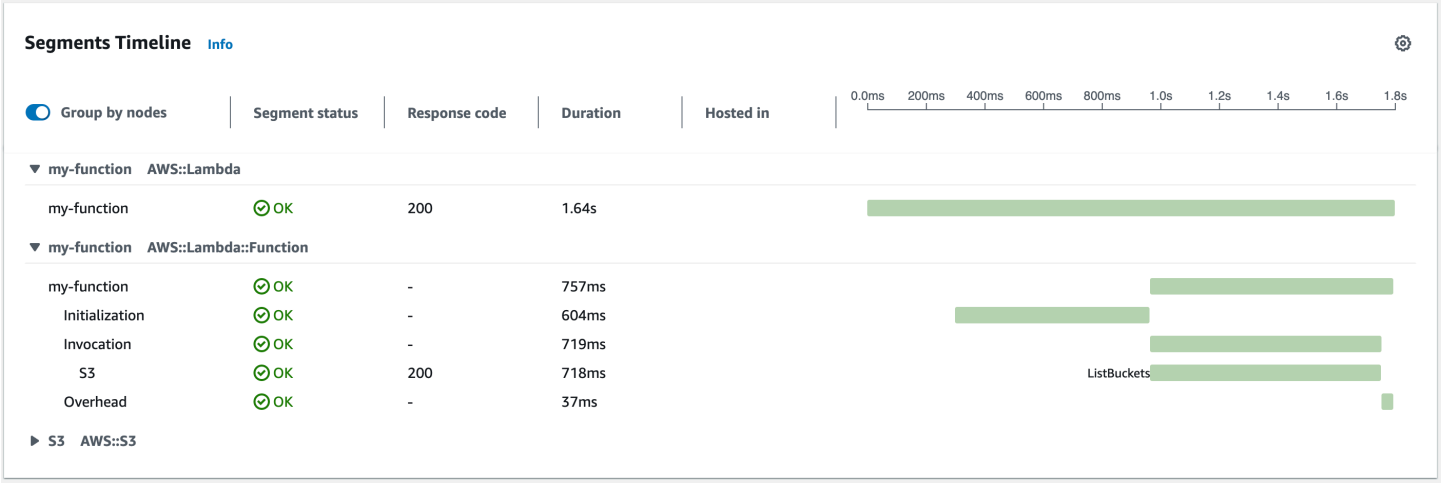
(SnapStart Nur Lambda) Das `Restore` Untersegment zeigt die Zeit, die Lambda benötigt, um einen Snapshot wiederherzustellen, die Laufzeit zu laden und alle Runtime-Hooks nach der Wiederherstellung auszuführen. Der Prozess der Wiederherstellung von Snapshots kann Zeit beinhalten, die für Aktivitäten außerhalb der MicroVM aufgewendet wird. Diese Zeit wird im `Restore`-Untersegment erfasst. Die Zeit, die Sie außerhalb der microVM für die Wiederherstellung eines Snapshots aufwenden, wird Ihnen nicht in Rechnung gestellt.

Das `Overhead`-Teilsegment stellt die Phase dar, die zwischen dem Zeitpunkt, zu dem die Laufzeit die Antwort sendet, und dem Signal für den nächsten Aufruf auftritt. Während dieser Zeit beendet die Laufzeit alle Aufgaben im Zusammenhang mit einem Aufruf und bereitet sich auf das Einfrieren der Sandbox vor.

> **Important**
>
> Sie können das X-Ray-SDK verwenden, um das `Invocation`-Teilsegment mit zusätzlichen Teilsegmenten für nachgeschaltete Aufrufe, Anmerkungen und Metadaten zu erweitern. Sie können nicht direkt auf das Funktionssegment zugreifen oder Arbeiten außerhalb des Handler-Aufruffunktionsbereichs aufzeichnen.

Weitere Informationen zu den Phasen der Lambda-Ausführungsumgebung finden Sie unter the section called “Ausführungsumgebung”.

Das folgende Diagramm zeigt ein Beispiel für einen Trace mit der alten X-Ray-Struktur.

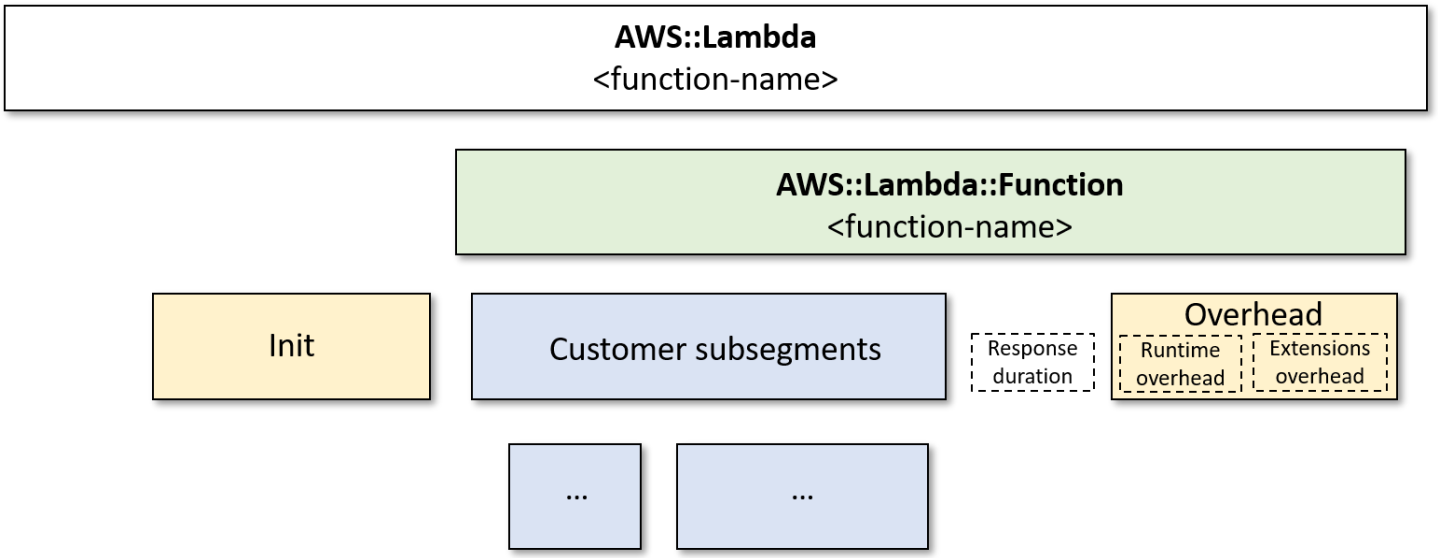


Beachten Sie die beiden Segmente im Beispiel. Beide heißen my-function, aber einer hat einen Ursprung von `AWS::Lambda` und der andere hat einen Ursprung von `AWS::Lambda::Function`. Wenn das `AWS::Lambda`-Segment einen Fehler anzeigt, hatte der Lambda-Service ein Problem. Wenn das `AWS::Lambda::Function`-Segment einen Fehler anzeigt, ist bei Ihrer Funktion ein Problem aufgetreten.

> **Note**
>
> Gelegentlich bemerken Sie in Ihren X-Ray-Ablaufverfolgungen eine große Lücke zwischen den Phasen der Funktionsinitialisierung und des Funktionsaufrufs. Bei Funktionen, die bereitgestellte Gleichzeitigkeit verwenden, liegt dass daran, dass Lambda Ihre Funktions-Instances lange vor dem Aufruf initialisiert. Für Funktionen, die unreservierte (On-demand) Gleichzeitigkeit verwenden, kann Lambda eine Funktions-Instance proaktiv initialisieren, auch wenn kein Aufruf erfolgt. Visuell werden diese beiden Fälle als Zeitlücke zwischen der Initialisierungs- und der Aufrufphase angezeigt.

AWS X-Ray Lambda-Segmentstruktur im neuen Stil

Die neue X-Ray-Struktur für das `AWS::Lambda`-Segment sieht wie folgt aus:

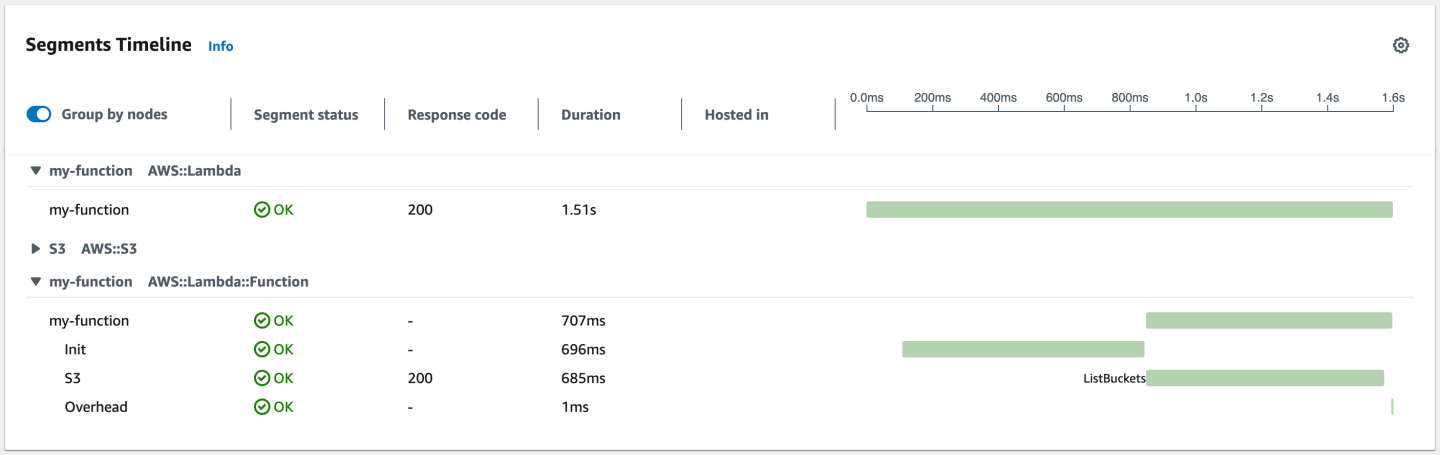


In diesem neuen Format stellt das Teilsegment `Init` wie bisher die Init-Phase des Lebenszyklus der Lambda-Ausführungsumgebung dar.

Im neuen Format gibt es kein Aufrufsegment. Stattdessen werden Kundenuntersegmente direkt an das `AWS::Lambda::Function`-Segment angehängt. Dieses Segment enthält die folgenden Metriken als Anmerkungen:

- `aws.responseLatency` – die Zeit, die benötigt wurde, bis die Funktion ausgeführt wurde
- `aws.responseDuration` – die Zeit, die benötigt wurde, um die Antwort an den Kunden zu übermitteln
- `aws.runtimeOverhead` – die zusätzliche Zeit, die die Laufzeit bis zum Abschluss benötigt hat
- `aws.extensionOverhead` – die zusätzliche Zeit, die die Erweiterungen zum Abschluss benötigt hat

Das folgende Diagramm zeigt ein Beispiel für einen Trace mit der neuen X-Ray-Struktur.



Beachten Sie die beiden Segmente im Beispiel. Beide heißen my-function, aber einer hat einen Ursprung von `AWS::Lambda` und der andere hat einen Ursprung von `AWS::Lambda::Function`. Wenn das `AWS::Lambda`-Segment einen Fehler anzeigt, hatte der Lambda-Service ein Problem. Wenn das `AWS::Lambda::Function`-Segment einen Fehler anzeigt, ist bei Ihrer Funktion ein Problem aufgetreten.

In den folgenden Themen finden Sie eine sprachspezifische Einführung in die Ablaufverfolgung in Lambda:

- Instrumentierung von Node.js Code in AWS Lambda
- Instrumentieren von Python-Code in AWS Lambda
- Instrumentieren von Ruby-Code in AWS Lambda
- Instrumentierung von Java-Code in AWS Lambda
- Instrumentieren von Go-Code in AWS Lambda
- Instrumentieren von C# -Code in AWS Lambda

Eine vollständige Liste der Services, welche die aktive Instrumentierung unterstützen, finden Sie unter Unterstützte AWS-Services im AWS X-Ray -Entwicklerhandbuch.

Standardmäßiges Nachverfolgungsverhalten in Lambda

Wenn die `Active`-Nachverfolgung nicht aktiviert ist, verwendet Lambda standardmäßig den `PassThrough`-Nachverfolgungsmodus.

Im `PassThrough`-Modus leitet Lambda den X-Ray-Nachverfolgungs-Header an Downstream-Services weiter, sendet aber nicht automatisch Nachverfolgungen. Dies gilt auch dann, wenn der

Nachverfolgungs-Header eine Entscheidung zum Sampling der Anfrage enthält. Wenn der Upstream-Service keinen X-Ray-Nachverfolgungs-Header bereitstellt, generiert Lambda einen Header und trifft die Entscheidung, kein Sampling durchzuführen. Sie können jedoch Ihre eigenen Nachverfolgungen senden, indem Sie Nachverfolgungsbibliotheken aus Ihrem Funktionscode aufrufen.

> **Note**
>
> Bisher sendete Lambda automatisch Nachverfolgungen, wenn Upstream-Services wie Amazon API Gateway einen Nachverfolgungs-Header hinzufügten. Da Lambda Nachverfolgungen nicht automatisch sendet, gibt Ihnen Lambda die Kontrolle, die für Sie wichtigen Funktionen nachzuverfolgen. Wenn Ihre Lösung von diesem passiven Nachverfolgungsverhalten abhängt, wechseln Sie zur `Active`-Nachverfolgung.

Berechtigungen für die Ausführungsrolle

Lambda benötigt die folgenden Berechtigungen, um Ablaufverfolgungsdaten an X-Ray zu senden. Fügen Sie sie der Ausführungsrolle Ihrer Funktion hinzu.

- röntgen: PutTraceSegments
- röntgen: PutTelemetryRecords

Diese Berechtigungen sind in der AWSXRayDaemonWriteAccessverwalteten Richtlinie enthalten.

Aktivieren der **`Active`**-Nachverfolgung mit der Lambda-API

Verwenden Sie die folgenden API-Operationen, um die Ablaufverfolgungskonfiguration mit dem AWS SDK AWS CLI oder zu verwalten:

- UpdateFunctionConfiguration
- GetFunctionConfiguration
- CreateFunction

Der folgende AWS CLI Beispielbefehl aktiviert die aktive Ablaufverfolgung für eine Funktion namens my-function.

```
aws lambda update-function-configuration --function-name my-function \
```

```
--tracing-config Mode=Active
```

Der Ablaufverfolgungsmodus ist Teil der versionsspezifischen Konfiguration, wenn Sie eine Version Ihrer Funktion veröffentlichen. Sie können den Ablaufverfolgungsmodus für eine veröffentlichte Version nicht ändern.

Aktivieren der Ablaufverfolgung mit **`Active`**CloudFormation

Verwenden Sie die Eigenschaft, um die Ablaufverfolgung `AWS::Lambda::Function` für eine Ressource in einer CloudFormation Vorlage zu aktivieren. `TracingConfig`

Example function-inline.yml – Ablaufverfolgungskonfiguration

```
Resources:
  function:
    Type: AWS::Lambda::Function
    Properties:
      TracingConfig:
        Mode: Active
      ...
```

Verwenden Sie für eine `AWS::Serverless::Function` Ressource AWS Serverless Application Model (AWS SAM) die `Tracing` Eigenschaft.

Example template.yml – Ablaufverfolgungskonfiguration

```
Resources:
  function:
    Type: AWS::Serverless::Function
    Properties:
      Tracing: Active
      ...
```

Überwachen Sie die Funktionsleistung mit Amazon CloudWatch Lambda Insights

Amazon CloudWatch Lambda Insights erfasst und aggregiert Lambda-Funktionslaufzeit-Leistungs-Metriken und Protokolle für Ihre Serverless-Anwendungen. Auf dieser Seite wird beschrieben, wie Sie Lambda Insights aktivieren und verwenden, um Probleme mit Ihren Lambda-Funktionen zu diagnostizieren.

Abschnitte

- Wie Lambda Insights Serverless-Anwendungen überwacht
- Preisgestaltung
- Unterstützte Laufzeiten
- Lambda Insights in der Lambda-Konsole aktivieren
- Programmgesteuertes Aktivieren von Lambda Insights
- Verwenden des Lambda-Insights-Dashboards
- Beispiel-Workflow zum Erkennen von Funktionsanomalien
- Beispiel-Workflow mit Abfragen zur Fehlerbehebung einer Funktion
- Als nächstes

Wie Lambda Insights Serverless-Anwendungen überwacht

CloudWatch Lambda Insights ist eine Überwachungs- und Fehlerbehebungslösung für Serverless-Anwendungen, die auf AWS Lambda ausgeführt werden. Die Lösung erfasst, aggregiert und fasst Metriken auf Systemebene zusammen, einschließlich CPU-Zeit, Arbeitsspeicher, Datenträger- und Netzwerknutzung. Sie erfasst, aggregiert und fasst Diagnoseinformationen wie Kaltstart und Lambda-Worker-Abschaltungen zusammen, um Probleme mit Ihren Lambda-Funktionen zu isolieren und schnell zu beheben.

Lambda Insights verwendet eine neue CloudWatch-Lambda-Insights-Erweiterung, die als Lambda-Ebene angeboten wurde. Wenn Sie diese Erweiterung für eine Lambda-Funktion für eine unterstützte Laufzeit aktivieren, werden Metriken auf Systemebene erfasst und ein einzelnes Performance-Protokollereignis für jeden Aufruf dieser Lambda-Funktion ausgegeben. CloudWatch verwendet eingebettete Metrikformatierung, um Metriken aus den Protokollereignissen zu extrahieren. Weitere Informationen finden Sie unter Verwendung von AWS Lambda-Erweiterungen.

Die Lambda-Insights-Ebene erweitert die `CreateLogStream` und `PutLogEvents` für die `/aws/lambda-insights/`-Protokollgruppe.

Preisgestaltung

Wenn Sie Lambda Insights für Ihre Lambda-Funktion aktivieren, meldet Lambda Insights 8 Metriken pro Funktion und jeder Funktionsaufruf sendet etwa 1 KB Protokolldaten an CloudWatch. Sie zahlen nur für die Metriken und Protokolle, die Lambda Insights für Ihre Funktion gemeldet haben. Es fallen keine Mindestgebühren oder Mindestnutzungsanforderungen an. Sie zahlen nicht für Lambda Insights, wenn die Funktion nicht aufgerufen wird. Ein Preisbeispiel finden Sie unter Amazon-CloudWatch-Preise.

Unterstützte Laufzeiten

Sie können Lambda Insights mit jeder Laufzeitumgebung verwenden, die Lambda-Erweiterungen unterstützen.

Lambda Insights in der Lambda-Konsole aktivieren

Sie können die erweiterte Lambda-Insights-Überwachung neuer und vorhandener Lambda-Funktionen aktivieren. Wenn Sie Lambda Insights auf einer Funktion in der Lambda-Konsole für eine unterstützte Laufzeit aktivieren, fügt Lambda die Lambda-Insights-Erweiterung als Ebene Ihrer Funktion hinzu und überprüft die `CloudWatchLambdaInsightsExecutionRolePolicy`-Richtlinie oder versucht, diese der Ausführungsrolle Ihrer Funktion zuzuweisen.

Lambda Insights in der Lambda-Konsole aktivieren

1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.
2. Wählen Sie Ihre Funktion.
3. Wählen Sie die Registerkarte Konfiguration aus.
4. Wählen Sie im linken Menü die Option Überwachungs- und Betriebstools.
5. Wählen Sie im Bereich Zusätzliche Überwachungstools die Option Edit (Bearbeiten).
6. Aktivieren Sie unter CloudWatch Lambda Insights die erweiterte Überwachung.
7. Wählen Sie Speichern.

Programmgesteuertes Aktivieren von Lambda Insights

Sie können Lambda Insights auch über die AWS Command Line Interface, (AWS CLI), AWS Serverless Application Model, (SAM)-CLI, das CloudFormation oder das AWS Cloud Development Kit (AWS CDK) aktivieren. Wenn Sie Lambda Insights programmgesteuert auf einer Funktion für eine unterstützte Laufzeit aktivieren, hängt CloudWatch die `CloudWatchLambdaInsightsExecutionRolePolicy`-Richtlinie an die Ausführungsrolle Ihrer Funktion an.

Weitere Informationen finden Sie unter Erste Schritte mit Lambda Insights im Amazon-CloudWatch-Benutzerhandbuch.

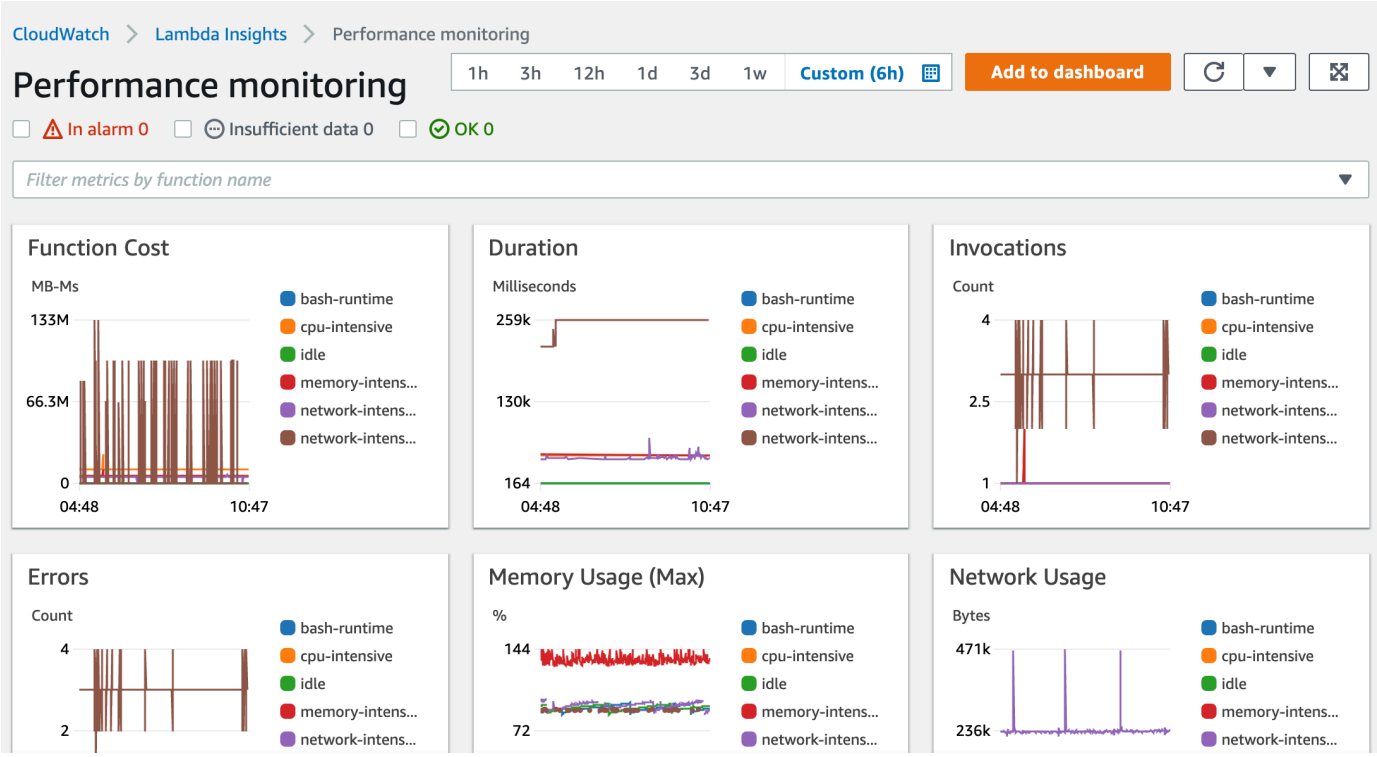
Verwenden des Lambda-Insights-Dashboards

Das Lambda-Insights-Dashboard hat zwei Ansichten in der CloudWatch-Konsole: die Multifunktionsübersicht und die Einzelfunktionsansicht. Die Multifunktionsübersicht aggregiert die Laufzeitmetriken für die Lambda-Funktionen im aktuellen AWS-Konto und in der Region. Die Einzelfunktionsansicht zeigt die verfügbaren Laufzeit-Metriken für eine einzelne Lambda-Funktion an.

Sie können die Multifunktionsübersicht des Lambda-Insights-Dashboards in der CloudWatch-Konsole verwenden, um übermäßig und nicht ausgelastete Lambda-Funktionen zu identifizieren. Sie können die Einzelfunktionsansicht des Lambda-Insights-Dashboards in der CloudWatch-Konsole verwenden, um Probleme bei einzelnen Anforderungen zu beheben.

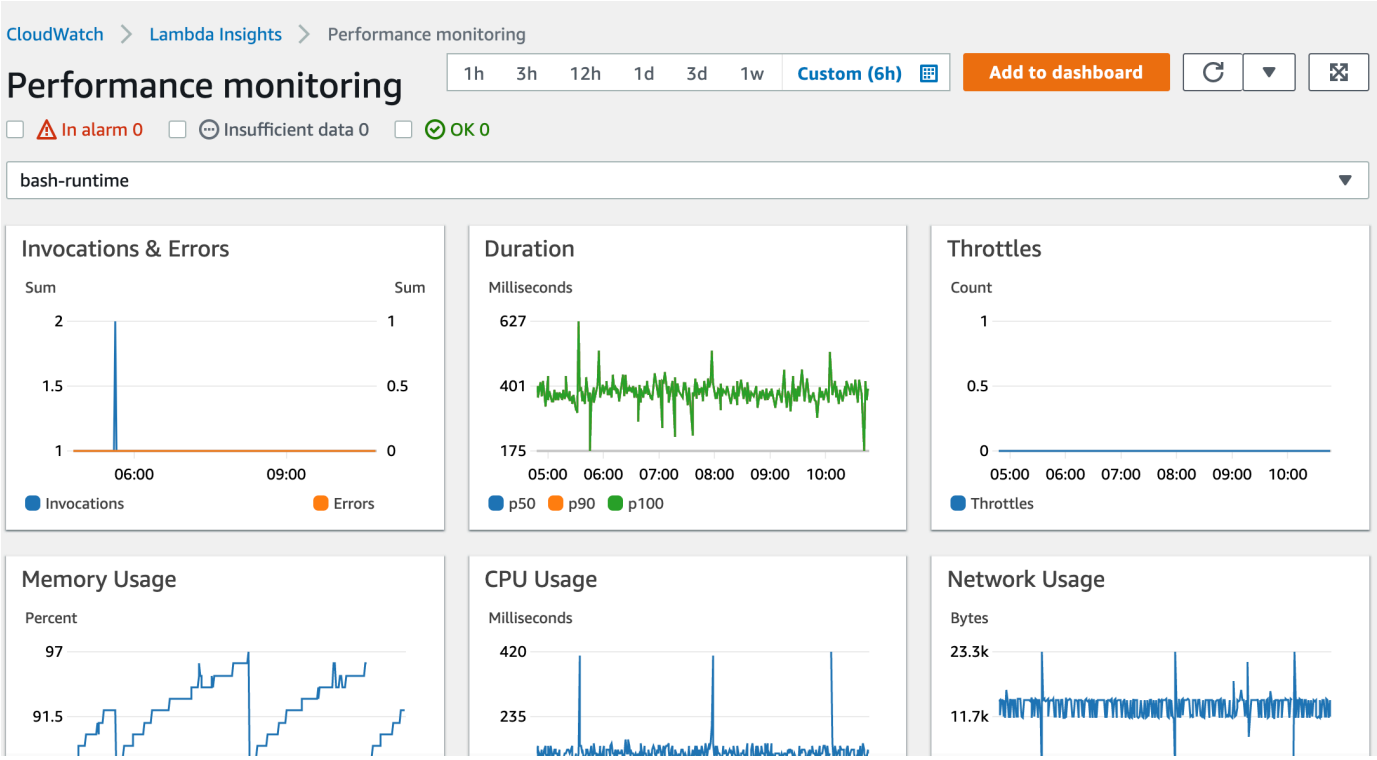
So zeigen Sie die Laufzeit-Metriken für alle Funktionen an:

1. Öffnen Sie die Seite Multifunktion in der CloudWatch-Konsole.
2. Wählen Sie aus den vordefinierten Zeitbereichen oder wählen Sie einen benutzerdefinierten Zeitbereich aus.
3. (Optional) Wählen Sie Zu Dashboard hinzufügen, um die Widgets zu Ihrem CloudWatch-Dashboard hinzuzufügen.



So zeigen Sie die Laufzeit-Metriken einer einzelnen Funktion an:

1. Öffnen Sie die Einzelfunktion in der CloudWatch-Konsole.
2. Wählen Sie aus den vordefinierten Zeitbereichen oder wählen Sie einen benutzerdefinierten Zeitbereich aus.
3. (Optional) Wählen Sie Zu Dashboard hinzufügen, um die Widgets zu Ihrem CloudWatch-Dashboard hinzuzufügen.



Weitere Informationen finden Sie unter Erstellen und Arbeiten mit Widgets in CloudWatch-Dashboards.

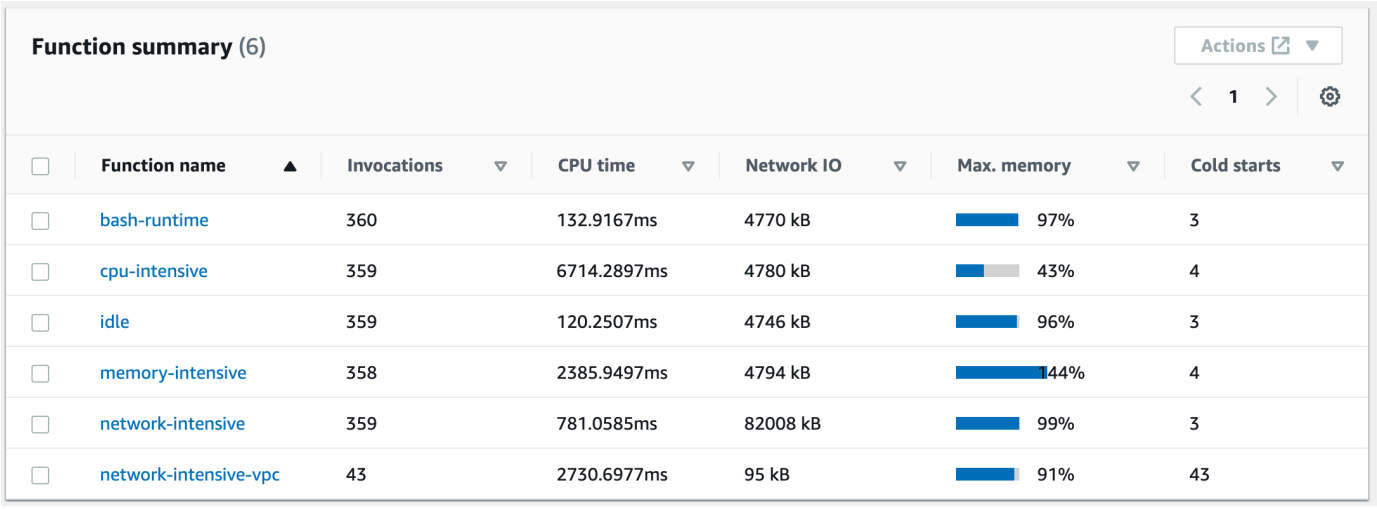
Beispiel-Workflow zum Erkennen von Funktionsanomalien

Sie können die Multifunktionsübersicht auf dem Lambda-Insights-Dashboard verwenden, um Anomalien des Rechenspeichers mit Ihrer Funktion zu identifizieren und zu erkennen. Wenn beispielsweise die Multifunktionsübersicht anzeigt, dass eine Funktion eine große Menge Speicher verwendet, können Sie detaillierte Metriken zur Speicherauslastung im Bereich Memory Usage (Speicherverwendung) anzeigen. Sie können dann zum Metrik-Dashboard wechseln, um die Anomalieerkennung zu aktivieren oder einen Alarm zu erstellen.

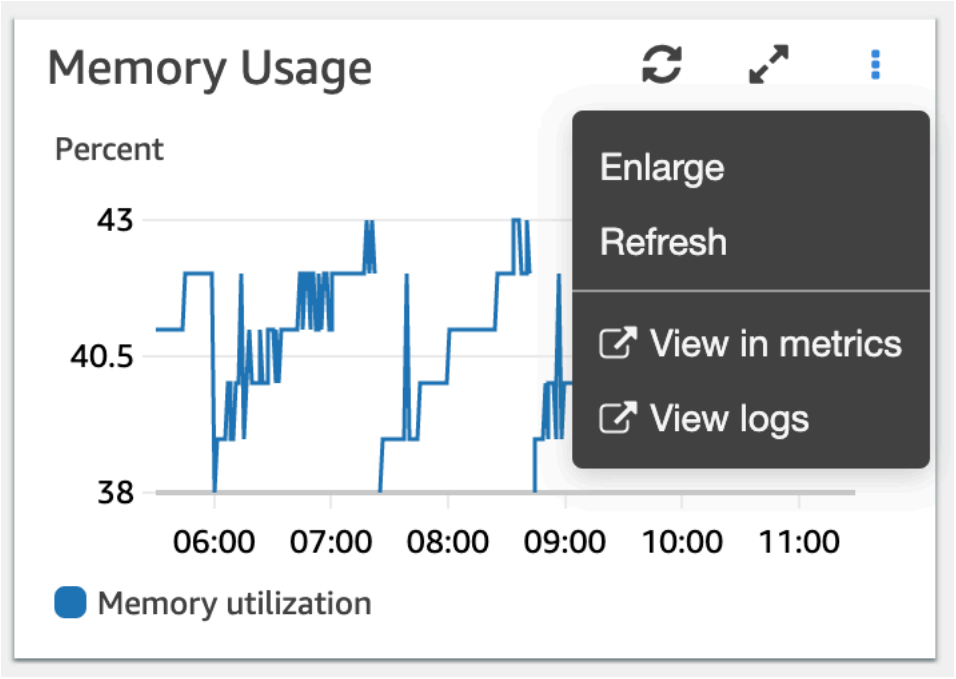
So aktivieren Sie die Anomalieerkennung für eine Funktion:

1. Öffnen Sie die Seite Multifunktion in der CloudWatch-Konsole.
2. Wählen Sie unter Function summary (Funktionsübersicht) den Namen Ihrer Funktion aus.

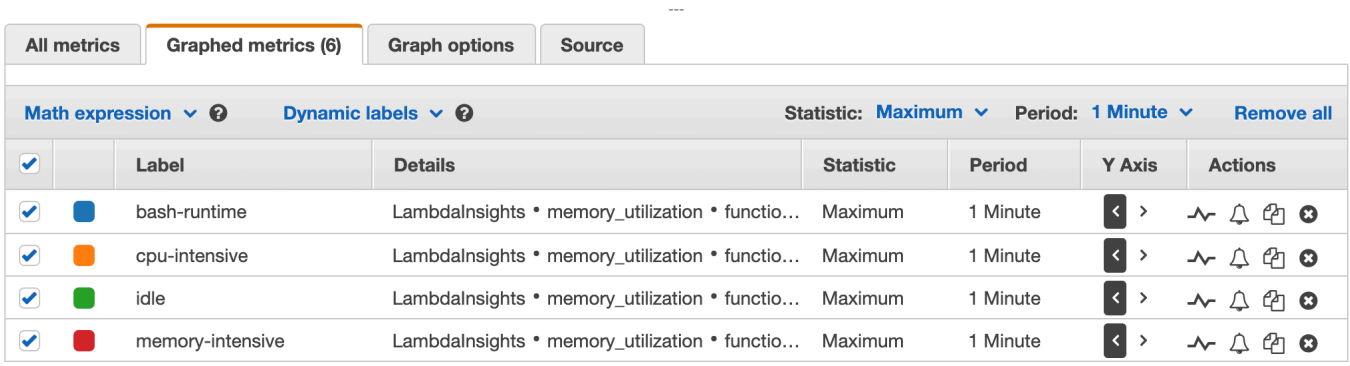
   Die Einzelfunktionsansicht wird mit den Funktionslaufzeitmetriken geöffnet.



3. Wählen Sie im Bereich Memory Usage (Speicherverwendung) die drei vertikalen Punkte aus, und wählen Sie dann View in metrics (In Metriken anzeigen), um das Metrics (Metriken)-Dashboard zu öffnen.



4. Wählen Sie auf der Registerkarte Graphed metrics (Metriken mit Diagrammen) in der Spalte Actions (Aktionen) das erste Symbol aus, um die Anomalieerkennung für die Funktion zu aktivieren.



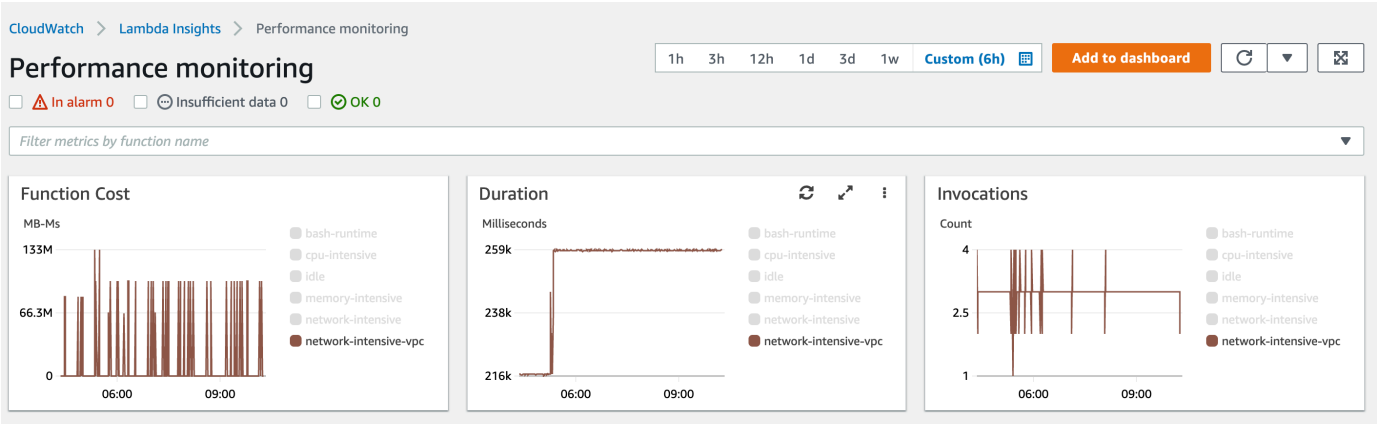
Weitere Informationen finden Sie unter Verwenden von CloudWatch-Anomalieerkennung.

Beispiel-Workflow mit Abfragen zur Fehlerbehebung einer Funktion

Sie können die Einzelfunktionsansicht im Lambda-Insights-Dashboard verwenden, um die Ursache eines Spitzenwerts der Funktionsdauer zu ermitteln. Wenn beispielsweise die Multifunktionsübersicht eine starke Erhöhung der Funktionsdauer anzeigt, können Sie jede Funktion im Bereich Duration (Dauer) anhalten oder auswählen, um festzustellen, welche Funktion die Erhöhung verursacht. Sie können dann zur Einzelfunktionsansicht wechseln und die Anwendungsprotokolle überprüfen, um die Ursache zu ermitteln.

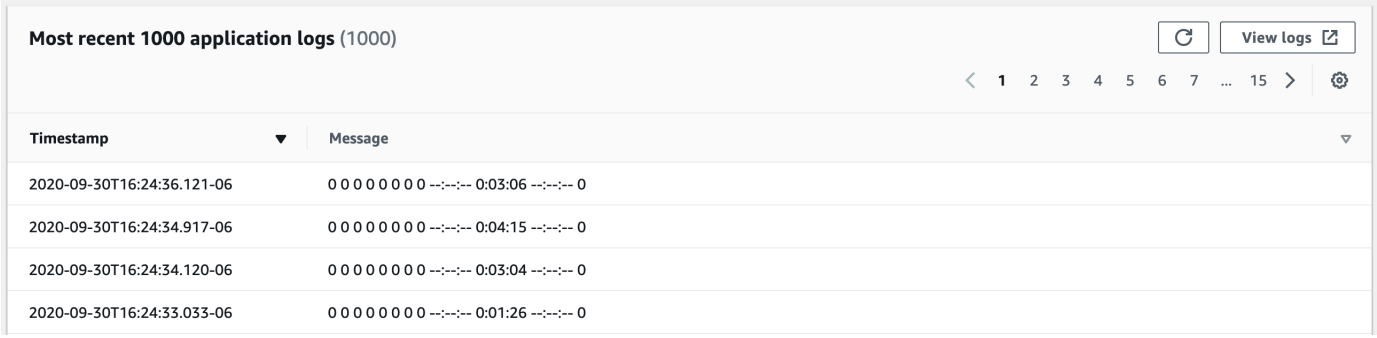
So führen Sie Abfragen für eine Funktion aus:

1. Öffnen Sie die Seite Multifunktion in der CloudWatch-Konsole.
2. Wählen Sie im Bereich Duration (Dauer) Ihre Funktion aus, um die Dauer-Metriken zu filtern.

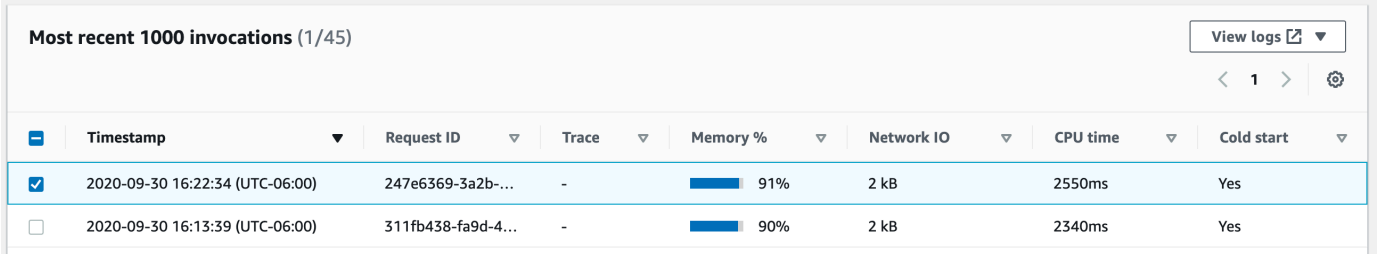


3. Öffnen Sie die Seite Einzelfunktionsansicht.
4. Wählen Sie die Dropdown-Liste Filter metrics by function name (Metriken nach Funktionsnamen filtern) und wählen Sie dann Ihre Funktion aus.

5. Um die neuesten 1000 Anwendungsprotokolle anzuzeigen, wählen Sie die Registerkarte Application logs (Anwendungsprotokolle).
6. Überprüfen Sie den Zeitstempel und die Meldung um die Aufrufanforderung zu identifizieren, für die Sie Fehler beheben möchten.

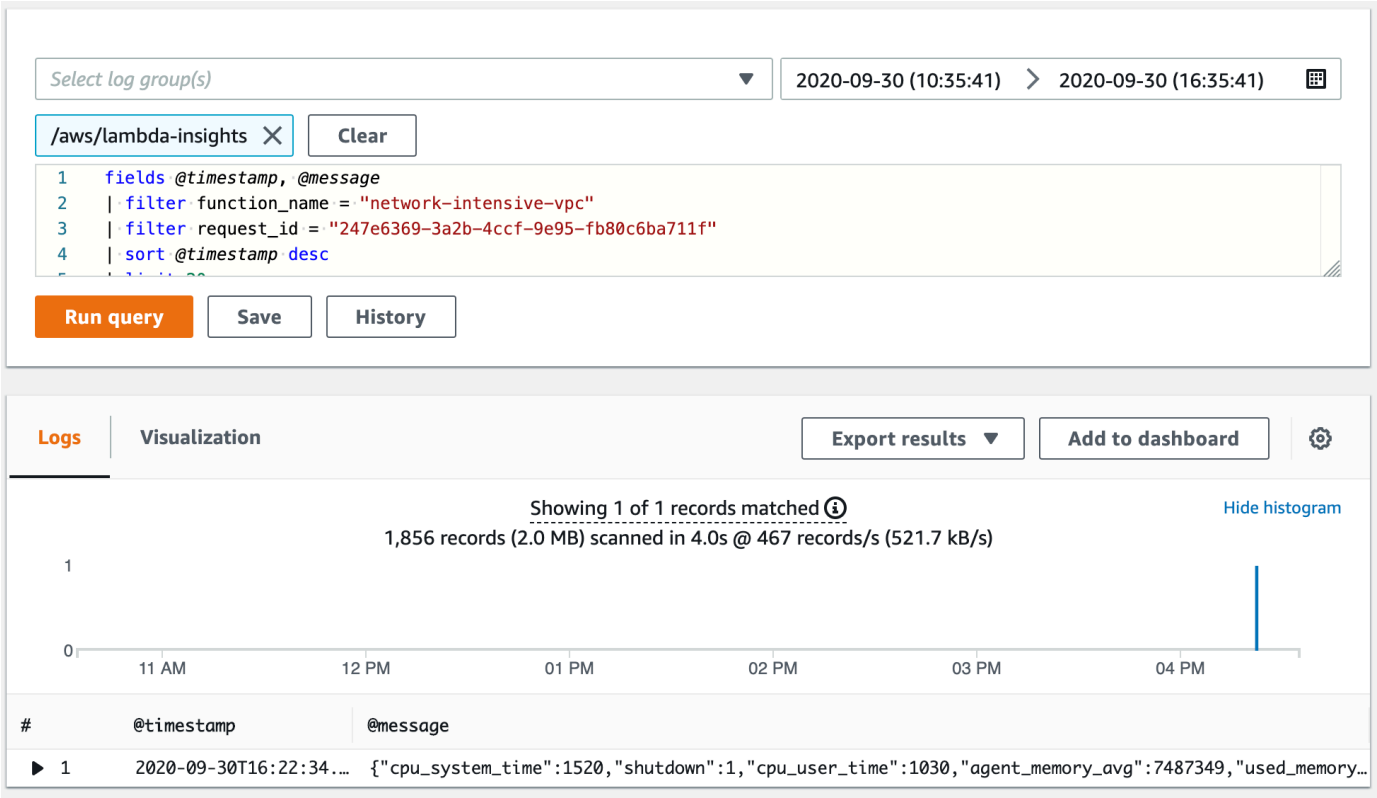


7. Um die letzten 1000 Aufrufe anzuzeigen, wählen Sie die Registerkarte Invocations (Aufrufe).
8. Wählen Sie den Zeitstempel oder die Meldung für die Aufrufanforderung aus, für die Sie Fehler beheben möchten.



9. Wählen Sie die Dropdown-Liste View logs (Protokolle anzeigen) und wählen Sie dann View performance logs (Leistungsprotokolle anzeigen) aus.

   Im Dashboard Logs Insights (Protokolleinblicke) wird eine automatisch generierte Abfrage für Ihre Funktion geöffnet.
10. Wählen Sie Run query (Abfrage ausführen), um eine Logs (Protokolle)-Meldung für die Aufrufanforderung zu generieren.



Als nächstes

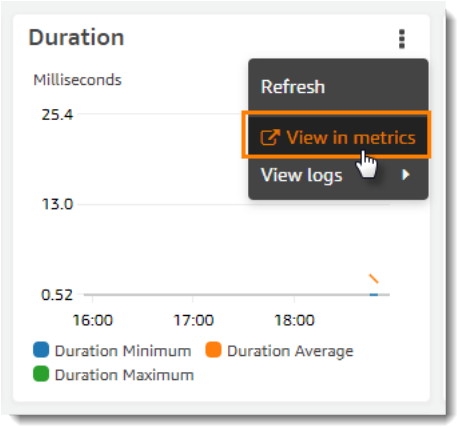
- Informationen zum Erstellen eines CloudWatch Logs-Dashboards finden Sie unter Erstellen eines Dashboards im Amazon-CloudWatch-Benutzerhandbuch.
- Informationen zum Hinzufügen von Abfragen zu einem CloudWatch-Logs-Dashboard finden Sie unter Abfrage zum Dashboard hinzufügen oder Abfrageergebnisse exportieren im Amazon-CloudWatch-Benutzerhandbuch.

Überwachen von Lambda-Anwendungen

Der Abschnitt Anwendungen der Lambda-Konsole enthält eine Registerkarte Überwachung, auf der Sie ein Amazon CloudWatch-Dashboard mit aggregierten Metriken für die Ressourcen in Ihrer Anwendung anzeigen können.

So überwachen Sie eine Lambda-Anwendung

1. Öffnen Sie die Seite Anwendungen der Lambda-Konsole.
2. Wählen Sie Monitoring.
3. Um weitere Details zu den Metriken in einem beliebigen Diagramm anzuzeigen, wählen Sie im Dropdown-Menü Ansicht in Metriken.



   Das Diagramm erscheint in einer neuen Registerkarte, wobei die relevanten Metriken unterhalb des Diagramms aufgelistet sind. Sie können Ihre Ansicht dieses Diagramms anpassen, indem Sie die angezeigten Metriken und Ressourcen, die Statistik, den Zeitraum und andere Faktoren ändern, um ein besseres Verständnis der aktuellen Situation zu erhalten.

Standardmäßig zeigt die Lambda-Konsole ein grundlegendes Dashboard an. Sie können diese Seite anpassen, indem Sie Ihrer Anwendungsvorlage ein oder mehrere Amazon-CloudWatch-Dashboards mit dem AWS::CloudWatch::Dashboard-Ressourcentyp hinzufügen. Wenn Ihre Vorlage ein oder mehrere Dashboards enthält, zeigt die Seite Ihre Dashboards anstelle des Standard-Dashboards an. Sie mit dem Dropdown-Menü oben rechts auf der Seite zwischen Dashboards wechseln. Das folgende Beispiel erstellt ein Dashboard mit einem einzigen Widget, das die Anzahl der Aufrufe einer Funktion mit dem Namen grafisch darstell `my-function`.

Example Dashboard-Funktionsvorlage

```
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
                    }
                }
            ]
        }
```

Weitere Informationen zum Erstellen von CloudWatch-Dashboards und -Widgets finden Sie im Artikel zur Dashboard-Textstruktur und -Syntax in der Amazon-CloudWatch-API-Referenz.

Überwachen der Anwendungsleistung mit Amazon CloudWatch Application Signals

Amazon CloudWatch Application Signals ist eine Lösung zur Überwachung der Anwendungsleistung, mit der Entwickler und Betreiber den Zustand und die Leistung ihrer mit Lambda erstellten Serverless-Anwendungen überwachen können. Sie können Application Signals mit einem Klick von der Lambda-Konsole aus aktivieren und Sie müssen Ihrer Lambda-Funktion keinen Instrumentierungscode oder externe Abhängigkeiten hinzufügen. Nachdem Sie Application Signals aktiviert haben, können Sie alle gesammelten Metriken und Traces in der CloudWatch-Konsole anzeigen. Auf dieser Seite wird beschrieben, wie Sie Application-Signals-Telemetriedaten für Ihre Anwendungen aktivieren und anzeigen.

Themen

- So lässt sich Application Signals in Lambda integrieren
- Preisgestaltung
- Unterstützte Laufzeiten
- Aktivieren von Application Signals in der Lambda-Konsole
- Verwenden des Application Signals-Dashboards

So lässt sich Application Signals in Lambda integrieren

Application Signals instrumentiert Ihre Lambda-Funktionen automatisch mithilfe erweiterter AWS Distro for OpenTelemetry (ADOT)-Bibliotheken, die über eine Lambda-Ebene bereitgestellt werden. Application Signals liest die von der Ebene gesammelten Daten und generiert Dashboards mit wichtigen Leistungsmetriken für Ihre Anwendungen.

Sie können diese Ebene mit einem Klick anhängen, indem Sie Application Signals in der Lambda-Konsole aktivieren. Wenn Sie Application Signals über die Konsole aktivieren, führt Lambda Folgendes in Ihrem Namen aus:

- Aktualisiert die Ausführungsrolle Ihrer Funktion mit der `CloudWatchLambdaApplicationSignalsExecutionRolePolicy`. Diese Richtlinie bietet Schreibzugriff auf AWS X-Ray- und CloudWatch-Protokollgruppen, die für Application Signals verwendet werden.
- Fügt Ihrer Funktion eine Ebene hinzu, die die Funktion automatisch so instrumentiert, dass sie Telemetriedaten wie Anfragen, Verfügbarkeit, Latenz, Fehler und Störungen erfasst. Um

sicherzustellen, dass Application Signals ordnungsgemäß funktioniert, entfernen Sie jeglichen vorhandenen X-Ray-SDK-Instrumentierungscode aus Ihrer Funktion. Benutzerdefinierter X-Ray SDK-Instrumentierungscode kann die von der Ebene bereitgestellte Instrumentierung beeinträchtigen.

- Fügt Ihrer Funktion die `AWS_LAMBDA_EXEC_WRAPPER`-Umgebungsvariable hinzu und setzt ihren Wert auf `/opt/otel-instrument`. Diese Umgebungsvariable ändert das Startverhalten Ihrer Funktion, um die Application Signals-Ebene zu nutzen. Sie ist für die richtige Instrumentierung erforderlich. Wenn diese Umgebungsvariable bereits existiert, stellen Sie sicher, dass sie auf den erforderlichen Wert gesetzt ist.

Preisgestaltung

Die Verwendung von Application Signals für Ihre Lambda-Funktionen verursacht Kosten. Preisinformationen finden Sie unter Amazon-CloudWatch-Preisinformationen.

Unterstützte Laufzeiten

Die Application Signal-Integration mit Lambda funktioniert mit den folgenden Laufzeiten:

- .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

Aktivieren von Application Signals in der Lambda-Konsole

Sie können Application Signals für jede vorhandene Lambda-Funktion mit einer unterstützten Laufzeit aktivieren. In den folgenden Schritten wird beschrieben, wie Sie Application Signals mit einem Klick in der Lambda-Konsole aktivieren.

So aktivieren Sie Application Signals in der Lambda-Konsole

1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.
2. Wählen Sie Ihre Funktion.
3. Wählen Sie die Registerkarte Konfiguration aus.
4. Wählen Sie im linken Menü die Option Überwachungs- und Betriebstools.
5. Wählen Sie im Bereich Zusätzliche Überwachungstools die Option Edit (Bearbeiten).
6. Wählen Sie unter CloudWatch Application Signals und AWS X-Ray unter Application Signals die Option Aktivieren aus.
7. Wählen Sie Speichern.

Wenn Sie Application Signals zum ersten Mal für Ihre Funktion aktivieren, müssen Sie auch eine einmalige Serviceerkennung für Application Signals in der CloudWatch-Konsole einrichten. Nachdem Sie dieses einmalige Serviceerkennungs-Setup abgeschlossen haben, erkennt Application Signals automatisch alle zusätzlichen Lambda-Funktionen, für die Sie Application Signals aktivieren, in allen Regionen.

> **Note**
>
> Nachdem Sie Ihre aktualisierte Funktion aufgerufen haben, kann es bis zu 10 Minuten dauern, bis Servicedaten im Application-Signals-Dashboard in der CloudWatch-Konsole angezeigt werden.

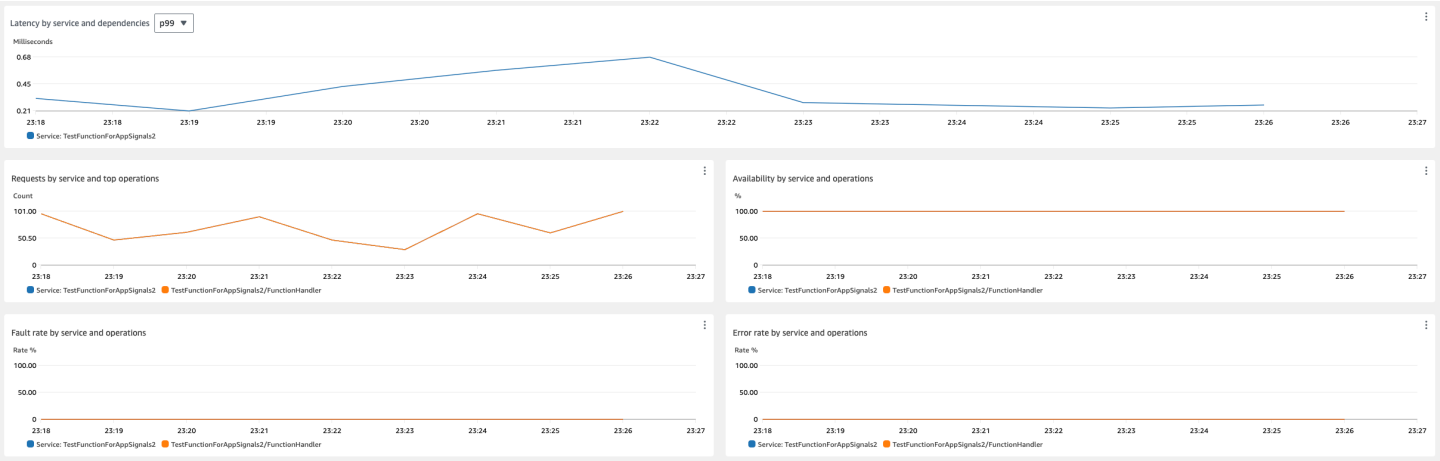
Verwenden des Application Signals-Dashboards

Nachdem Sie Application Signals für Ihre Funktion aktiviert haben, können Sie Ihre Anwendungsmetriken in der CloudWatch-Konsole visualisieren. Mit den folgenden Schritten können Sie das zugehörige Application-Signals-Dashboard schnell von der Lambda-Konsole aus aufrufen:

So rufen Sie das Application Signals-Dashboard für Ihre Funktion auf

1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.
2. Wählen Sie Ihre Funktion.
3. Wählen Sie den Tab Überwachung.
4. Wählen Sie die Schaltfläche Anwendungssignale anzeigen. Dies führt Sie direkt zur Anwendungssignalen-Übersicht für Ihren Dienst in der CloudWatch-Konsole.

Der folgende Screenshot zeigt beispielsweise Metriken für die Latenzzeit, die Anzahl der Anfragen, die Verfügbarkeit, die Fehlerrate und die Fehlerquote für eine Funktion in einem Zeitfenster von 10 Minuten.



Um das Beste aus Ihrer Integration mit Application Signals herauszuholen, können Sie Service Level Objectives (SLOs) für Ihre Anwendung erstellen. Sie können beispielsweise Latenz-SLOs erstellen, um sicherzustellen, dass Ihre Anwendung schnell auf Benutzeranfragen reagiert, und Verfügbarkeits-SLOs, um die Verfügbarkeit zu verfolgen. SLOs können Ihnen helfen, Leistungseinbußen oder Ausfälle zu erkennen, bevor sie sich auf Ihre Benutzer auswirken. Weitere Informationen finden Sie unter Service Level Objectives (SLOs) im Amazon-CloudWatch-Benutzerhandbuch.

Remote-Debugging von Lambda-Funktionen mit Visual Studio Code

Mit dem Remote-Debugging-Feature im AWS Toolkit for Visual Studio Code können Sie Ihre Lambda-Funktionen debuggen, die direkt in der AWS-Cloud ausgeführt werden. Dies ist nützlich, wenn Sie Probleme untersuchen, die lokal schwer zu reproduzieren sind oder nur anhand von Protokollen diagnostiziert werden können.

Mit Remote-Debugging können Sie:

- Haltepunkte in Ihrem Lambda-Funktionscode setzen.
- Die Codeausführung in Echtzeit durchlaufen.
- Variablen und Status während der Laufzeit überprüfen.
- Lambda-Funktionen debuggen, die in AWS bereitgestellt wurden, einschließlich solcher in VPCs oder mit bestimmten IAM-Berechtigungen.

Unterstützte Laufzeiten

Remote-Debugging wird für die folgenden Laufzeiten unterstützt:

- Python (AL2023)
- Java
- JavaScript/Node.js (AL2023)

> **Note**
>
> Remote-Debugging wird sowohl für x86_64- als auch für arm64-Architekturen unterstützt.

Sicherheit und Remote-Debugging

Das Remote-Debugging erfolgt innerhalb der bestehenden Lambda-Sicherheitsgrenzen. Benutzer können mithilfe der Berechtigung `UpdateFunctionConfiguration`, die bereits Zugriff auf Funktionsumgebungsvariablen und -konfigurationen ermöglicht, Ebenen an eine Funktion anhängen. Das Remote-Debugging geht nicht über diese bestehenden Berechtigungen hinaus.

Stattdessen werden zusätzliche Sicherheitskontrollen durch sicheres Tunneling und automatische Sitzungsverwaltung hinzugefügt. Darüber hinaus ist das Remote-Debugging ein vollständig vom Kunden kontrolliertes Feature, das explizite Berechtigungen und Aktionen erfordert:

- Erstellung eines sicheren IoT-Tunnels: Das AWS-Toolkit muss einen sicheren IoT-Tunnel erstellen, was nur mit der ausdrücklichen Berechtigung des Benutzers über `iot:OpenTunnel` möglich ist.
- Anhängen der Debugging-Ebene und Tokenverwaltung: Der Debugging-Prozess gewährleistet die Sicherheit durch folgende Kontrollen:
  - Die Debugging-Ebene muss an die Lambda-Funktion angehängt werden, wofür die folgenden Berechtigungen erforderlich sind: `lambda:UpdateFunctionConfiguration` und `lambda:GetLayerVersion`.
  - Ein Sicherheitstoken (generiert über `iot:OpenTunnel`) muss vor jeder Debugging-Sitzung in der Funktionsumgebungsvariablen aktualisiert werden, was ebenfalls `lambda:UpdateFunctionConfiguration` erfordert.
  - Aus Sicherheitsgründen wird dieses Token automatisch rotiert und die Debug-Ebene wird am Ende jeder Debugging-Sitzung automatisch entfernt und kann nicht wiederverwendet werden.

> **Note**
>
> Remote-Debugging wird sowohl für x86_64- als auch für arm64-Architekturen unterstützt.

Voraussetzungen

Bevor Sie mit dem Remote-Debugging beginnen, stellen Sie sicher, dass Sie über Folgendes verfügen:

1. Eine Lambda-Funktion, die für Ihr AWS-Konto bereitgestellt wurde.
2. AWS Toolkit for Visual Studio Code. Installationsanweisungen finden Sie unter Einrichten des AWS Toolkit for Visual Studio Code.
3. Die Version des AWS-Toolkits, die Sie installiert haben, ist mindestens 3.69.0.
4. AWS-Anmeldeinformationen, die in AWS Toolkit for Visual Studio Code konfiguriert sind. Weitere Informationen finden Sie unter Authentifizierung und Zugriffskontrolle.

Remote-Debugging von Lambda-Funktionen

Führen Sie die folgenden Schritte aus, um eine Remote-Debugging-Sitzung zu starten:

1. Öffnen Sie den AWS Explorer in VS Code, indem Sie das AWS-Symbol in der linken Seitenleiste auswählen.
2. Erweitern Sie den Abschnitt „Lambda“, um Ihre Funktionen anzuzeigen.
3. Klicken Sie mit der rechten Maustaste auf die Funktion, die Sie debuggen möchten.
4. Wählen Sie im Kontextmenü die Option Remote aufrufen aus.
5. Aktivieren Sie im sich öffnenden Aufruffenster das Kontrollkästchen Debugging aktivieren.
6. Klicken Sie auf Aufrufen, um die Remote-Debugging-Sitzung zu starten.

> **Note**
>
> Lambda-Funktionen haben ein kombiniertes Limit von 250 MB für Funktionscode und alle angehängten Ebenen. Die Remote-Debugging-Ebene erhöht die Größe Ihrer Funktion um ca. 40 MB.

Eine Remote-Debugging-Sitzung endet, wenn Sie:

- auf dem Bildschirm mit der Remoteaufruf-Konfiguration“ die Option Debug-Setup entfernen auswählen
- in den Debugging-Steuerelementen von VS Code das Symbol „Trennen“ auswählen
- die Handler-Datei im VS-Code-Editor auswählen.

> **Note**
>
> Die Debugging-Ebene wird automatisch entfernt, wenn nach Ihrem letzten Aufruf 60 Sekunden lang keine Aktivität stattfindet.

Deaktivieren von Remote-Debugging

Es gibt drei Möglichkeiten, dieses Feature zu deaktivieren:

- Funktionsupdates verweigern: `lambda:UpdateFunctionConfiguration` auf `deny` setzen
- IoT-Berechtigungen einschränken: IoT-bezogene Berechtigungen verweigern
- Debug-Ebenen blockieren: `lambda:GetLayerVersion` für die folgenden ARNs verweigern:
  - `arn:aws:lambda:*:*:layer:LDKLayerX86:*`
  - `arn:aws:lambda:*:*:layer:LDKLayerArm64:*`

> **Note**
>
> Durch Deaktivieren dieses Features wird verhindert, dass die Debugging-Ebene während der Aktualisierung der Funktionskonfiguration hinzugefügt wird.

Zusätzliche Informationen

Weitere Informationen zur Verwendung von Lambda in VS Code finden Sie unter Lokale Entwicklung von Lambda-Funktionen mit VS Code.

Ausführliche Anweisungen zur Fehlerbehebung, zu erweiterten Anwendungsfällen und zur regionalen Verfügbarkeit finden Sie unter Remote-Debuggen von Lambda-Funktionen im AWS Toolkit for Visual Studio Code-Benutzerhandbuch.

Verwaltung von Lambda-Abhängigkeiten mit Ebenen

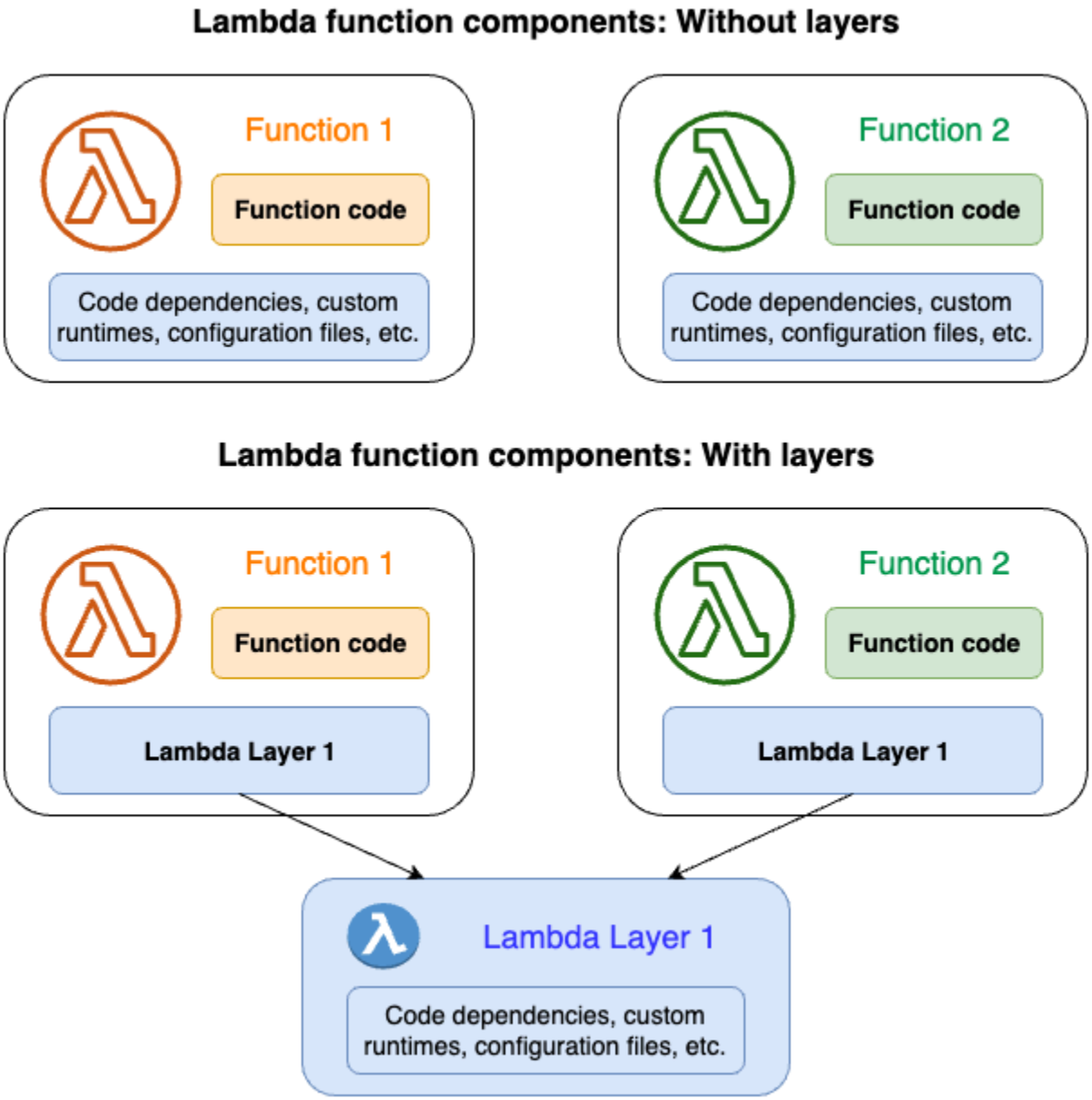
Eine Lambda-Ebene ist ein ZIP-Dateiarchiv mit ergänzendem Code oder ergänzenden Daten. Ebenen enthalten üblicherweise Bibliotheksabhängigkeiten, eine benutzerdefinierte Laufzeit oder Konfigurationsdateien.

Die Verwendung von Ebenen ermöglicht Folgendes:

- Verringern der Größe Ihrer Bereitstellungspakete: Platzieren Sie Ihre Funktionsabhängigkeiten in einer Ebene, anstatt sie zusammen mit Ihrem Funktionscode in Ihr Bereitstellungspaket einzuschließen. So bleiben Bereitstellungspakete klein und übersichtlich.
- Trennen von grundlegender Funktionslogik und Abhängigkeiten: Mit Ebenen können Sie Ihre Funktionsabhängigkeiten unabhängig von Ihrem Funktionscode aktualisieren (und umgekehrt). Dies trägt zur Trennung von Belangen bei und hilft Ihnen dabei, sich auf Ihre Funktionslogik zu konzentrieren.
- Verwenden von Abhängigkeiten für mehrere Funktionen: Erstellte Ebenen können auf eine beliebige Anzahl von Funktionen in Ihrem Konto angewendet werden. Ohne Ebenen müssen die gleichen Abhängigkeiten in jedes einzelne Bereitstellungspaket eingefügt werden.
- Verwenden des Code-Editors der Lambda-Konsole: Der Code-Editor ist ein praktisches Tool zum schnellen Testen kleinerer Funktionscode-Aktualisierungen. Sie können den Editor jedoch nicht verwenden, wenn Ihr Bereitstellungspaket zu groß ist. Die Verwendung von Ebenen verringert die Paketgröße und kann die Nutzung des Code-Editors ermöglichen.
- Zum Sperren einer eingebetteten SDK-Version.Die eingebetteten SDKs können sich ohne vorherige Ankündigung ändern, wenn AWS neue Services und Features veröffentlicht. Sie können eine Version des SDK sperren, indem Sie eine Lambda-Ebene mit der jeweils benötigten Version erstellen. Die Funktion verwendet dann immer die Version in der Ebene, auch wenn sich die im Service eingebettete Version ändert.

Wenn Sie mit Lambda-Funktionen in Go oder Rust arbeiten, empfehlen wir, keine Ebenen zu verwenden. Bei Go- und Rust-Funktionen stellen Sie Ihren Funktionscode als ausführbare Datei bereit, die Ihren kompilierten Funktionscode zusammen mit allen Abhängigkeiten enthält. Wenn Sie Ihre Abhängigkeiten in eine Ebene einfügen, muss Ihre Funktion während der Initialisierungsphase manuell weitere Baugruppen laden, was die Kaltstartzeiten erhöhen kann. Für eine optimale Leistung der Go- und Rust-Funktionen sollten Sie Ihre Abhängigkeiten zusammen mit Ihrem Bereitstellungspaket angeben.

Das folgende Diagramm veranschaulicht die allgemeinen architektonischen Unterschiede zwischen zwei Funktionen mit gemeinsamen Abhängigkeiten. Bei einer Variante werden Lambda-Ebenen verwendet, bei der anderen nicht.



Wenn Sie einer Funktion eine Ebene hinzufügen, extrahiert Lambda die Inhalte der Ebene in das Verzeichnis `/opt` in der Ausführungsumgebung Ihrer Funktion. Alle nativ unterstützten Lambda-Laufzeiten enthalten Pfade zu spezifischen Verzeichnissen innerhalb des Verzeichnisses `/opt`. Dadurch erhält die Funktion Zugriff auf Ihre Ebeneninhalte. Weitere Informationen zu diesen

spezifischen Pfaden und zum richtigen Verpacken Ihrer Ebenen finden Sie unter the section called “Verpacken von Ebenen”.

Sie können pro Funktion bis zu fünf Ebenen einschließen. Ebenen können zudem nur mit Lambda-Funktionen verwendet werden, die als ZIP-Dateiarchiv bereitgestellt werden. Bei als Container-Image definierten Funktionen werden Ihre bevorzugte Laufzeit und alle Codeabhängigkeiten beim Erstellen des Container-Images verpackt. Weitere Informationen finden Sie unter Arbeiten mit Lambda-Ebenen und Erweiterungen in Container-Images im AWS-Computing-Blog.

Themen

- Verwenden von Ebenen
- Ebenen und Ebenenversionen
- Verpacken Ihres Ebeneninhalts
- Erstellen und Löschen von Ebenen in Lambda
- Hinzufügen von Ebenen zu Funktionen
- Verwendung von AWS CloudFormation mit Ebenen
- Verwendung von AWS SAM mit Ebenen

Verwenden von Ebenen

Zum Erstellen einer Ebene müssen Sie Ihre Abhängigkeiten in einer ZIP-Datei verpacken – ähnlich wie beim Erstellen eines normalen Bereitstellungspakets. Genauer gesagt umfasst der allgemeine Prozess der Erstellung und Verwendung von Ebenen die drei folgenden Schritte:

- Verpacken Sie zuerst Ihren Ebeneninhalt. Erstellen Sie also ein ZIP-Dateiarchiv. Weitere Informationen finden Sie unter the section called “Verpacken von Ebenen”.
- Erstellen Sie als Nächstes die Ebene in Lambda. Weitere Informationen finden Sie unter the section called “Erstellen und Löschen von Ebenen”.
- Fügen Sie die Ebene Ihren Funktionen hinzu). Weitere Informationen finden Sie unter the section called “Hinzufügen von Ebenen”.

Ebenen und Ebenenversionen

Eine Ebenenversion ist eine unveränderliche Momentaufnahme einer spezifischen Version einer Ebene. Wenn Sie eine neue Ebene erstellen, erstellt Lambda eine neue Ebenenversion mit der

Versionsnummer 1. Bei jeder Veröffentlichung einer Aktualisierung für die Ebene erhöht Lambda die Versionsnummer und erstellt eine neue Ebenenversion.

Jede Ebenenversion wird durch einen eindeutigen Amazon-Ressourcennamen (ARN) identifiziert. Beim Hinzufügen einer Ebene zur Funktion muss die exakte Ebenenversion angegeben werden, die Sie verwenden möchten (z. B. `arn:aws:lambda:us-east-1:123456789012:layer:my-layer:1`).

Verpacken Ihres Ebeneninhalts

Eine Lambda-Ebene ist ein ZIP-Dateiarchiv mit ergänzendem Code oder ergänzenden Daten. Ebenen enthalten üblicherweise Bibliotheksabhängigkeiten, eine benutzerdefinierte Laufzeit oder Konfigurationsdateien.

In diesem Abschnitt erfahren Sie, wie Sie Ihren Ebeneninhalt ordnungsgemäß verpacken. Weitere konzeptionelle Informationen zu Ebenen sowie dazu, warum Sie ggf. welche verwenden sollten, finden Sie unter Lambda-Ebenen.

Der erste Schritt beim Erstellen einer Ebene besteht darin, den gesamten Ebeneninhalt in einem ZIP-Dateiarchiv zu bündeln. Da Lambda-Funktionen unter Amazon Linux ausgeführt werden, muss Ihr Ebeneninhalt in einer Linux-Umgebung kompiliert und erstellt werden können.

Um sicherzustellen, dass Ihr Ebeneninhalt ordnungsgemäß in einer Linux-Umgebung funktioniert, empfiehlt es sich, den Ebeneninhalt mit einem Tool wie Docker zu erstellen.

Themen

- Ebenenpfade für jede Lambda-Laufzeit

Ebenenpfade für jede Lambda-Laufzeit

Wenn Sie einer Funktion eine Ebene hinzufügen, lädt Lambda den Ebeneninhalt in das Verzeichnis `/opt` der Ausführungsumgebung. Für jede Lambda-Laufzeit enthält die Variable `PATH` bereits spezifische Ordnerpfade innerhalb des Verzeichnisses `/opt`. Um sicherzustellen, dass Lambda den Inhalt Ihrer Ebene aufnimmt, müssen die Abhängigkeiten Ihrer ZIP-Datei in einem der folgenden Ordnerpfade enthalten sein:

Laufzeit	Pfad
Node.js	`nodejs/node_modules`
	`nodejs/node18/node_modules` (`NODE_PATH`)
	`nodejs/node20/node_modules` (`NODE_PATH`)
	`nodejs/node22/node_modules` (`NODE_PATH`)
Python	`python`

Laufzeit	Pfad
	`python/lib/ python3.x /site-packages` (Site-Verzeichnisse)
Java	`java/lib` (`CLASSPATH`)
Ruby	`ruby/gems/3.4.0` (`GEM_PATH`)
	`ruby/lib` (`RUBYLIB`)
Alle Laufzeiten	`bin` (`PATH`)
	`lib` (`LD_LIBRARY_PATH`)

Die folgenden Beispiele zeigen, wie Sie die Ordner im ZIP-Archiv Ihrer Ebene strukturieren können.

Node.js

Example Dateistruktur für das AWS X-Ray-SDK für 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 Dateistruktur für das JSON-Gem

```
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 Dateistruktur für Jackson-JAR-Datei

```
layer_content.zip
# java
    # lib
        # jackson-core-2.17.0.jar
        # <other potential dependencies>
        # ...
```

All

Example Dateistruktur für jq-Bibliothek

```
jq.zip
# bin/jq
```

Eine sprachspezifische Anleitung zum Verpacken, Erstellen und Hinzufügen einer Ebene finden Sie auf den folgenden Seiten:

- Node.js – the section called “Layer”
- Python – the section called “Layer”
- Ruby – the section called “Ebenen”
- Java – the section called “Layer”

Wir empfehlen, keine Ebenen zur Verwaltung von Abhängigkeiten für in Go und Rust geschriebene Lambda-Funktionen zu verwenden. Der Grund dafür ist, dass in diesen Sprachen geschriebene Lambda-Funktionen zu einer einzigen ausführbaren Datei kompiliert werden, die Sie Lambda

bei der Bereitstellung Ihrer Funktion zur Verfügung stellen. Diese ausführbare Datei enthält Ihren kompilierten Funktionscode zusammen mit all seinen Abhängigkeiten. Die Verwendung von Ebenen verkompliziert nicht nur diesen Vorgang, sondern führt auch zu längeren Kaltstartzeiten, da Ihre Funktionen während der Init-Phase zusätzliche Assemblys manuell in den Speicher laden müssen.

Um externe Abhängigkeiten mit Lambda-Funktionen in Go und Rust zu verwenden, nehmen Sie sie direkt in Ihr Bereitstellungspaket auf.

Erstellen und Löschen von Ebenen in Lambda

Eine Lambda-Ebene ist ein ZIP-Dateiarchiv mit ergänzendem Code oder ergänzenden Daten. Ebenen enthalten üblicherweise Bibliotheksabhängigkeiten, eine benutzerdefinierte Laufzeit oder Konfigurationsdateien.

In diesem Abschnitt erfahren Sie, wie Sie Ebenen in Lambda erstellen und löschen. Weitere konzeptionelle Informationen zu Ebenen sowie dazu, warum Sie ggf. welche verwenden sollten, finden Sie unter Lambda-Ebenen.

Nach dem Verpacken Ihres Ebeneninhalts muss die Ebene in Lambda erstellt werden. In diesem Abschnitt wird nur gezeigt, wie Ebenen mithilfe der Lambda-Konsole oder mithilfe der Lambda-API erstellt und gelöscht werden. Eine Anleitung zum Erstellen einer Ebene mithilfe von AWS CloudFormation finden Sie unter the section called “Ebenen mit CloudFormation”. Eine Anleitung zum Erstellen einer Ebene mithilfe von AWS Serverless Application Model (AWS SAM) finden Sie unter the section called “Ebenen mit AWS SAM”.

Themen

- Erstellen einer Ebene
- Löschen einer Ebenen-Version

Erstellen einer Ebene

Zum Erstellen einer Ebene können Sie das ZIP-Archiv entweder über Ihren lokalen Computer oder über Amazon Simple Storage Service (Amazon S3) hochladen. Lambda extrahiert den Ebenen-Inhalt in das `/opt`-Verzeichnis, wenn die Ausführungsumgebung für die Funktion eingerichtet wird.

Ebenen können eine oder mehrere Ebenenversionen haben. Wenn Sie eine Ebene erstellen, legt Lambda die Ebenen-Version auf Version 1 fest. Die Berechtigungen für eine bereits vorhandene Ebenenversion können jederzeit geändert werden. Um den Code zu aktualisieren oder andere Konfigurationsänderungen vorzunehmen, müssen Sie allerdings eine neue Version der Ebene erstellen.

So erstellen Sie eine Ebene (Konsole)

1. Öffnen Sie die Seite Ebenen der Lambda-Konsole.
2. Wählen Sie Create Layer (Ebene erstellen) aus.
3. Geben Sie unter Ebenen-Konfiguration unter Name einen Namen für Ihre Ebene ein.

4. (Optional) Geben Sie im Feld Description (Beschreibung) eine Beschreibung für Ihre Ebene ein.
5. Um Ihren Ebenen-Code upzuloaden, führen Sie einen der folgenden Schritte aus:
   - Um eine ZIP-Datei von Ihrem Computer upzuloaden, wählen Sie Eine .zip-Datei hochladen. Wählen Sie Upload, um Ihre lokale ZIP-Datei auszuwählen.
   - Um eine Datei aus Amazon S3 upzuloaden, wählen Sie Hochladen einer Datei aus Amazon S3 aus. Geben Sie dann für Amazon-S3-Link-URL einen Link zu der Datei ein.
6. (Optional) Für Kompatible Architekturen, wählen Sie einen Wert oder beide Werte aus. Weitere Informationen finden Sie unter the section called “Befehlssätze (ARM/x86)”.
7. (Optional) Wählen Sie unter Kompatible Laufzeiten die Laufzeiten aus, mit denen Ihre Ebene kompatibel ist.
8. (Optional) Für License (Lizenz) geben Sie alle erforderlichen Lizenzinformationen ein.
9. Wählen Sie Erstellen aus.

Alternativ können Sie den Befehl publish-layer-version AWS Command Line Interface (CLI) ausführen. Beispiel:

```
aws lambda publish-layer-version --layer-name my-layer --zip-file fileb://layer.zip --
compatible-runtimes python3.14
```

Bei jeder Ausführung von `publish-layer-version` erstellt Lambda eine neue Version der Ebene.

Löschen einer Ebenen-Version

Verwenden Sie den DeleteLayerVersionAPI-Vorgang, um eine Layer-Version zu löschen. Führen Sie den delete-layer-version AWS CLI Befehl beispielsweise mit dem angegebenen Layer-Namen und der angegebenen Layer-Version aus.

```
aws lambda delete-layer-version --layer-name my-layer --version-number 1
```

Wenn Sie eine Ebenenversion löschen, können Sie keine Lambda-Funktionen mehr für deren Verwendung konfigurieren. Funktionen, die diese Version bereits vorhanden, können jedoch weiterhin darauf zugreifen. Außerdem werden Versionsnummern für einen Ebenennamen nie von Lambda wiederverwendet.

Bei der Berechnung von Kontingenten bedeutet das Löschen einer Layer-Version, dass sie nicht mehr als Teil des standardmäßigen 75-GB-Kontingents für die Speicherung von Funktionen

und Layern gezählt wird. Bei Funktionen, die eine gelöschte Layer-Version verwenden, wird der Layer-Inhalt jedoch weiterhin auf das Größenkontingent für das Bereitstellungspaket der Funktion angerechnet (d. h. 250 MB für .zip-Dateiarchive).

Hinzufügen von Ebenen zu Funktionen

Eine Lambda-Ebene ist ein ZIP-Dateiarchiv mit ergänzendem Code oder ergänzenden Daten. Ebenen enthalten üblicherweise Bibliotheksabhängigkeiten, eine benutzerdefinierte Laufzeit oder Konfigurationsdateien.

In diesem Abschnitt erfahren Sie, wie Sie einer Lambda-Funktion eine Ebene hinzufügen. Weitere konzeptionelle Informationen zu Ebenen sowie dazu, warum Sie ggf. welche verwenden sollten, finden Sie unter Lambda-Ebenen.

Bevor Sie eine Lambda-Funktion für die Verwendung einer Ebene konfigurieren können, ist Folgendes erforderlich:

- Verpacken Ihres Ebeneninhalts
- Erstellen einer Ebene in Lambda
- Stellen Sie sicher, dass Sie die Berechtigung haben, die GetLayerVersionAPI in der Layer-Version aufzurufen. Für Funktionen in Ihrem AWS-Konto müssen Sie diese Berechtigung in Ihrer Benutzerrichtlinie haben. Um eine Ebene in einem anderen Konto zu verwenden, muss der Eigentümer des anderen Kontos Ihrem Konto die Berechtigung in einer ressourcenbasierten Richtlinie erteilen. Beispiele finden Sie unter the section called “Ebenenzugriff für andere Konten”.

Sie können bis zu fünf Ebenen zu einer Lambda-Funktion hinzufügen. Die entpackte Gesamtgröße der Funktion und aller Ebenen darf das Größenlimit des entpackten Bereitstellungspakets von 250 MB nicht überschreiten. Weitere Informationen finden Sie unter Lambda-Kontingente.

Ihre Funktionen können eine beliebige Ebenenversion, die Sie bereits hinzugefügt haben, weiterhin nutzen, auch wenn diese Ebenenversion gelöscht wurde oder Ihnen die Zugriffsberechtigung für die Ebene entzogen wurde. Sie können jedoch keine neue Funktion erstellen, die eine Version einer gelöschten Ebene verwendet.

So fügen Sie eine Ebene zu einer Funktion hinzu

1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.
2. Wählen Sie die Funktion aus.
3. Blättern Sie nach unten bis zum Abschnitt Ebenen und wählen Sie dann Eine Ebene hinzufügen.
4. Wählen Sie unter Ebene auswählen eine Ebenenquelle aus:

    a. AWS Ebenen: Wählen Sie aus der Liste der AWS-verwalteten Erweiterungen aus.

   b. Benutzerdefinierte Ebenen: Wählen Sie eine Ebene aus, die in Ihrem AWS-Konto erstellt wurde.
   c. Geben Sie einen ARN an: Um einen Layer aus einem anderen AWS-Konto, z. B. einer Drittanbieter-Erweiterung, zu verwenden, geben Sie den Amazon-Ressourcennamen (ARN) ein.
5. Wählen Sie Hinzufügen aus.

Die Reihenfolge, in der Sie die Ebenen hinzufügen, ist die Reihenfolge, in der Lambda den Ebeneninhalt in der Ausführungsumgebung zusammenführt. Sie können die Reihenfolge für Zusammenführungen der Layers mit der -Konsole ändern.

So aktualisieren Sie die Zusammenführungsreihenfolge von Ebenen für Ihre Funktion (Konsole)

1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.
2. Wählen Sie die zu konfigurierende Funktion aus.
3. Wählen Sie unter Ebenen die Option Bearbeiten
4. Wählen Sie eine der Ebenen aus.
5. Klicken Sie aufFrüher MergeoderSpäter Zusammenführungenum die Reihenfolge der Ebenen anzupassen.
6. Wählen Sie Speichern.

Ebenen werden versioniert. Der Inhalt jeder Ebenenversion ist unveränderlich. Der Besitzer einer Ebene kann neue Ebenenversionen veröffentlichen, um aktualisierte Inhalte bereitzustellen.
Sie können die Konsole verwenden, um die mit Ihren Funktionen verknüpfte Ebenenversion zu aktualisieren.

So aktualisieren Sie Ebenenversionen für Ihre Funktion (Konsole)

1. Öffnen Sie die Seite Ebenen der Lambda-Konsole.
2. Wählen Sie die Ebene aus, für die Sie die Version aktualisieren möchten.
3. Wählen Sie die Registerkarte Funktionen mit dieser Version aus.
4. Wählen Sie die Funktionen aus, die Sie ändern möchten, und wählen Sie anschließend Bearbeiten aus.
5. Wählen Sie unter Ebenenversion die Ebenenversion aus, zu der Sie wechseln möchten.
6. Klicken Sie auf Funktionen aktualisieren.

Sie können die Versionen der Funktionsebene nicht AWS kontenübergreifend aktualisieren.

Suche nach Ebeneninformationen

Verwenden Sie die ListLayersAPI, um Ebenen in Ihrem Konto zu finden, die mit der Laufzeit Ihrer Funktion kompatibel sind. Sie können beispielsweise den folgenden Befehl list-layers AWS Command Line Interface (CLI) verwenden:

```
aws lambda list-layers --compatible-runtime python3.14
```

Die Ausgabe sollte folgendermaßen oder ähnlich aussehen:

```
{
    "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"
                ]
            }
        }
    ]
}
```

Wenn Sie alle Ebenen in Ihrem Konto auflisten möchten, lassen Sie die Option `--compatible-runtime` weg. Die Antwortdetails enthalten jeweils die neueste Version der einzelnen Ebenen.

Sie können die neueste Version eines Layers auch mithilfe der ListLayerVersionsAPI abrufen. Sie können z. B. den folgenden CLI-Befehl vom Typ `list-layer-versions` verwenden:

```
aws lambda list-layer-versions --layer-name my-layer
```

Die Ausgabe sollte folgendermaßen oder ähnlich aussehen:

```
{
    "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"
            ]
        }
    ]
}
```

Verwendung von AWS CloudFormation mit Ebenen

Sie können CloudFormation verwenden, um eine Ebene zu erstellen und die Ebene mit Ihrer Lambda-Funktion verknüpfen. Die folgende Beispielvorlage erstellt eine Ebene mit dem Namen `my-lambda-layer` und fügt sie mithilfe der Eigenschaft Layers an die Lambda-Funktion an.

In diesem Beispiel gibt die Vorlage den Amazon-Ressourcennamen (ARN) einer bestehenden IAM-Ausführungsrolle an. Sie können mit der CloudFormation-Ressource AWS::IAM::Role auch eine neue Ausführungsrolle in der Vorlage erstellen.

Ihre Funktion benötigt keine speziellen Berechtigungen, um Ebenen zu verwenden.

```
---
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
```

Verwendung von AWS SAM mit Ebenen

Sie können das AWS Serverless Application Model (AWS SAM) verwenden, um die Erstellung von Ebenen in Ihrer Anwendung zu automatisieren. Der Ressourcentyp `AWS::Serverless::LayerVersion` erstellt eine Ebenenversion, auf die Sie in Ihrer Lambda-Funktionskonfiguration verweisen können.

```
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
```

Lambda-Funktionen mit Lambda-Erweiterungen erweitern

Sie können Lambda-Erweiterungen verwenden, um Ihre Lambda-Funktionen zu erweitern. Verwenden Sie beispielsweise Lambda-Erweiterungen, um Funktionen in Ihre bevorzugten Überwachungs-, Beobachtbarkeits-, Sicherheits- und Governance-Tools zu integrieren. Sie können aus einer Vielzahl von Tools wählen, die von AWS Lambda-Partnern bereitgestellt werden, oder Sie können Ihre eigenen Lambda-Erweiterungen erstellen.

Lambda unterstützt externe und interne Erweiterungen. Eine externe Erweiterung wird als unabhängiger Prozess in der Ausführungsumgebung ausgeführt und wird weiterhin ausgeführt, nachdem der Funktionsaufruf vollständig verarbeitet wurde. Da Erweiterungen als separate Prozesse ausgeführt werden, können Sie sie in einer anderen Sprache als die Funktion schreiben. Alle Lambda-Laufzeiten unterstützen Erweiterungen.

Eine interne Erweiterung wird als Teil des Laufzeitprozesses ausgeführt. Ihre Funktion greift auf interne Erweiterungen zu, indem Sie Wrapper-Skripte oder In-Process-Mechanismen wie verwende `JAVA_TOOL_OPTIONS`. Weitere Informationen finden Sie unter Ändern der Laufzeitumgebung.

Sie können Erweiterungen zu einer Funktion hinzufügen, indem Sie die Lambda-Konsole, die AWS Command Line Interface (AWS CLI) oder die Infrastructure-as-Code-Services (IaC) oder Tools wie CloudFormation, AWS Serverless Application Model (AWS SAM) und Terraform verwenden.

Ihnen wird die Ausführungszeit berechnet, die von der Erweiterung verbraucht wird (in Schritten von 1 ms). Es fallen keine Kosten für die Installation eigener Erweiterungen an. Weitere Informationen zur Preisgestaltung für Erweiterungen finden Sie unter AWS Lambda-Preise. Preisinformationen zu Partnererweiterungen finden Sie auf den Websites dieser Partner. Eine Liste der offiziellen Partnererweiterungen finden Sie unter the section called “Partner für Erweiterungen”.

Ein Tutorial zu Erweiterungen und deren Verwendung mit Ihren Lambda-Funktionen finden Sie im AWS Lambda Extensions Workshop.

Themen

- Ausführungsumgebung
- Auswirkungen auf Leistung und Ressourcen
- Berechtigungen
- Konfigurieren von Lambda-Erweiterungen

- AWS Lambda-Partner für Erweiterungen
- Verwendung der Lambda-Erweiterungs-API zur Erstellung von Erweiterungen
- Zugriff auf Echtzeit-Telemetriedaten für Erweiterungen über die Telemetrie-API

Ausführungsumgebung

Lambda ruft Ihre Funktion in einer Ausführungsumgebung auf, die eine sichere und isolierte Laufzeitumgebung bereitstellt. Die Ausführungsumgebung verwaltet die Ressourcen, die zum Ausführen Ihrer Funktion erforderlich sind, und bietet Lebenszyklusunterstützung für die Laufzeit und die Erweiterungen der Funktion.

Der Lebenszyklus der Ausführungsumgebung umfasst die folgenden Phasen:

- **`Init`**: In dieser Phase erstellt oder hebt Lambda eine Ausführungsumgebung mit den konfigurierten Ressourcen auf, lädt den Code für die Funktion und alle Ebenen herunter, initialisiert alle Erweiterungen, initialisiert die Laufzeit und führt dann den Initialisierungscode der Funktion (der Code außerhalb des Haupthandlers) aus. Die Phase `Init` erfolgt entweder während des ersten Aufrufs oder vor Funktionsaufrufen, wenn Sie die bereitgestellte Parallelität aktiviert haben.

  Die `Init`-Phase ist in drei Unterphasen unterteilt: `Extension init`, `Runtime init` und `Function init`. Diese Unterphasen stellen sicher, dass alle Erweiterungen und die Laufzeit ihre Einrichtungs-Aufgaben abschließen, bevor der Funktionscode ausgeführt wird.

  Wenn Lambda SnapStart aktiviert ist, findet die `Init`-Phase statt, wenn Sie eine Funktionsversion veröffentlichen. Lambda speichert einen Snapshot des Arbeitsspeichers und des Festplattenzustands der initialisierten Ausführungsumgebung, speichert den verschlüsselten Snapshot und speichert ihn im Cache für den Zugriff mit geringer Latenz. Wenn Sie einen Laufzeit-Hook vor dem Prüfpunkt haben, wird der Code am Ende der `Init`-Phase ausgeführt.
- **`Restore`** (Nur SnapStart): Wenn Sie zum ersten Mal eine SnapStart-Funktion aufrufen und wenn die Funktion hochskaliert wird, nimmt Lambda neue Ausführungsumgebungen aus dem persistenten Snapshot wieder auf, anstatt die Funktion von Grund auf neu zu initialisieren. Wenn Sie über einen Laufzeit-Hook nach der Wiederherstellung verfügen, wird der Code am Ende der `Restore`-Phase ausgeführt. Die Dauer von Laufzeit-Hooks nach der Wiederherstellung wird Ihnen in Rechnung gestellt. Die Laufzeit muss geladen werden und Laufzeit-Hooks nach der Wiederherstellung müssen innerhalb des Timeout-Limits(10 Sekunden) abgeschlossen werden. Andernfalls erhalten Sie eine SnapStartTimeoutException. Wenn die `Restore`-Phase abgeschlossen ist, ruft Lambda den Funktionshandler (Invoke-Phase) auf.

- **`Invoke`**: In dieser Phase ruft Lambda den Funktionshandler auf. Nachdem die Funktion vollständig ausgeführt wurde, bereitet sich Lambda auf die Verarbeitung eines weiteren Funktionsaufrufs vor.
- **`Shutdown`**: Diese Phase wird ausgelöst, wenn die Lambda-Funktion für einen bestimmten Zeitraum keine Aufrufe empfängt. In dieser `Shutdown`-Phase fährt Lambda die Laufzeit herunter, warnt die Erweiterungen, damit sie sauber beendet werden können und entfernt dann die Umgebung. Lambda sendet ein `Shutdown`-Ereignis an jede Erweiterung, das der Erweiterung mitteilt, dass die Umgebung beendet wird.

Während der `Init`-Phase extrahiert Lambda Ebenen, die Erweiterungen enthalten, in das `/opt`-Verzeichnis in der Ausführungsumgebung. Lambda sucht im `/opt/extensions/`-Verzeichnis nach Erweiterungen, interpretiert jede Datei als ausführbaren Bootstrap zum Starten der Erweiterung und startet alle Erweiterungen parallel.

Auswirkungen auf Leistung und Ressourcen

Die Größe der Erweiterungen Ihrer Funktion zählt für die Größenbeschränkung des Bereitstellungspakets. Bei einem ZIP-Dateiarchiv darf die entpackte Gesamtgröße der Funktion und sämtlicher Erweiterungen das Limit des entpackten Bereitstellungspakets von 250 MB nicht überschreiten.

Erweiterungen können sich auf die Leistung Ihrer Funktion auswirken, da sie Funktionsressourcen wie CPU, Arbeitsspeicher und Speicher gemeinsam nutzen. Wenn eine Erweiterung beispielsweise rechenintensive Operationen ausführt, kann die Ausführungsdauer Ihrer Funktion erhöht werden.

Jede Erweiterung muss ihre Initialisierung abschließen, bevor Lambda die Funktion aufruft. Daher kann eine Erweiterung, die erhebliche Initialisierungszeit verbraucht, die Latenz des Funktionsaufrufs erhöhen.

Um die zusätzliche Zeit zu messen, die die Erweiterung nach der Funktionsausführung benötigt, können Sie die Funktionsmetrik `PostRuntimeExtensionsDuration` verwenden. Um die Zunahme des verwendeten Speichers zu messen, können Sie die `MaxMemoryUsed`-Metrik verwenden. Um die Auswirkungen einer bestimmten Erweiterung zu verstehen, können Sie verschiedene Versionen Ihrer Funktionen nebeneinander ausführen.

> **Note**
>
> `MaxMemoryUsed` ist eine der Metriken, die von Lambda Insights erfasst werden, und keine native Lambda-Metrik.

Berechtigungen

Erweiterungen haben Zugriff auf die gleichen Ressourcen wie Funktionen. Da Erweiterungen in derselben Umgebung wie die Funktion ausgeführt werden, werden Berechtigungen von der Funktion und der Erweiterung gemeinsam genutzt.

Für ein ZIP-Dateiarchiv können Sie eine CloudFormation-Vorlage erstellen, um das Anhängen derselben Erweiterungskonfiguration – einschließlich AWS Identity and Access Management-(IAM)-Berechtigungen – an mehrere Funktionen zu vereinfachen.

Konfigurieren von Lambda-Erweiterungen

Konfigurieren von Erweiterungen (ZIP-Dateiarchiv)

Sie können Ihrer Funktion eine Erweiterung als Lambda-Ebene hinzufügen. Mithilfe von Ebenen können Sie Erweiterungen in Ihrer Organisation oder in der gesamten Lambda-Entwicklercommunity freigeben. Sie können einer Ebene eine oder mehrere Erweiterungen hinzufügen. Sie können bis zu 10 Erweiterungen für eine Funktion registrieren.

Sie fügen die Erweiterung zu Ihrer Funktion hinzu, indem Sie dieselbe Methode verwenden wie für jede Ebene. Weitere Informationen finden Sie unter Lambda-Ebenen.

Hinzufügen einer Erweiterung zu Ihrer Funktion (Konsole)

1. Öffnen Sie die Seite Funktionen der Lambda-Konsole.
2. Wählen Sie eine Funktion aus.
3. Wählen Sie die Registerkarte Code aus, falls sie noch nicht ausgewählt ist.
4. Wählen Sie unter Ebenen die Option Bearbeiten aus.
5. Geben Sie unter Choose a layer (Ebene auswählen) Specify an ARN (Einen ARN angeben).
6. Geben Sie unter Specify an ARN (Einen ARN angeben), den Amazon-Ressourcennamen (ARN) einer Erweiterungsebene ein.
7. Wählen Sie Add aus.

Verwenden von Erweiterungen in Container-Images

Sie können Ihrem Container-Image Erweiterungen hinzufügen. Die Container-Image-Einstellung ENTRYPOINT gibt den Hauptprozess für die Funktion an. Konfigurieren Sie die Einstellung ENTRYPOINT im Dockerfile oder als Überschreibung in der Funktionskonfiguration.

Sie können mehrere Prozesse in einem Container ausführen. Lambda verwaltet den Lebenszyklus des Hauptprozesses und aller zusätzlichen Prozesse. Lambda verwendet die Erweiterungs-API, um den Erweiterungslebenszyklus zu verwalten.

Beispiel: Hinzufügen einer externen Erweiterung

Eine externe Erweiterung wird in einem separaten Prozess von der Lambda-Funktion ausgeführt. Lambda startet einen Prozess für jede Erweiterung im `/opt/extensions/`-Verzeichnis. Lambda

verwendet die Erweiterungs-API, um den Erweiterungslebenszyklus zu verwalten. Nachdem die Funktion bis zum Abschluss ausgeführt wurde, sendet Lambda ein `Shutdown`-Ereignis an jede externe Erweiterung.

Example Hinzufügen einer externen Erweiterung zu einem Python-Basis-Image

```
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
```

Nächste Schritte

Um mehr über Erweiterungen zu erfahren, empfehlen wir die folgenden Ressourcen:

- Ein grundlegendes Arbeitsbeispiel finden Sie unter Erweiterungen erstellen für AWS Lambda im AWS-Compute-Blog.
- Informationen zu Erweiterungen, die AWS Lambda-Partner bereitstellen, finden Sie unter Einführung von AWS Lambda-Erweiterungen im AWS Compute Blog.
- Verfügbare Beispiel-Erweiterungen und Wrapper-Skripte finden Sie unter AWS Lambda Erweiterungen im AWS Samples GitHub Repository.

AWS Lambda-Partner für Erweiterungen

AWS Lambda hat sich mit mehreren Drittunternehmen zusammengetan, um Erweiterungen zur Integration in Ihre Lambda-Funktionen bereitzustellen. In der folgenden Liste werden Erweiterungen von Drittanbietern aufgeführt, die Sie jederzeit verwenden können.

- AppDynamics – Bietet die automatische Instrumentierung von Node.js- oder Python-Lambda-Funktionen und bietet Sichtbarkeit und Warnungen zur Funktionsleistung.
- Axiom – Bietet Dashboards zur Überwachung der Leistung von Lambda-Funktionen und aggregierte Metriken auf Systemebene.
- Datadog – Bietet umfassende Echtzeit-Transparenz für Ihre Serverless-Anwendungen durch die Verwendung von Metriken, Nachverfolgungen und Protokollen.
- Dynatrace – Bietet Einblick in Nachverfolgungen und Metriken und nutzt KI für die automatisierte Fehlererkennung und Ursachenanalyse über den gesamten Anwendungs-Stack.
- Elastic – Bietet Application Performance Monitoring (APM) zur Identifizierung und Behebung von Ursachenproblemen mithilfe korrelierter Traces, Metriken und Logs.
- Epsagon – Hört sich Aufrufereignisse an, speichert Nachverfolgungen und sendet sie parallel an Lambda-Funktionsausführungen.
- Fastly– Schützt Ihre Lambda-Funktionen vor verdächtigen Aktivitäten wie Angriffen im Injektionsstil, Kontoübernahme durch Ausfüllen von Anmeldeinformationen, böswilligen Bots und API-Missbrauch.
- HashiCorp Vault – Verwaltet Geheimnisse und stellt sie Entwicklern zur Verwendung im Funktionscode zur Verfügung, ohne Funktionen vaultbewusst zu machen.
- Honeycomb – Beobachtbarkeits-Tool zum Debuggen Ihres App-Stacks.
- Lumigo – Profiliert Lambda-Funktionsaufrufe und sammelt Metriken zur Behebung von Problemen in Serverless- und Microservice-Umgebungen.
- New Relic – Läuft zusammen mit Lambda-Funktionen und sammelt, verbessert und transportiert Telemetriedaten automatisch zur einheitlichen Beobachtungsplattform von New Relic.
- Sedai – Eine autonome, auf KI/ML basierende Cloud-Management-Plattform, die eine kontinuierliche Optimierung für Cloud-Betriebsteams ermöglicht, um Cloud-Kosteneinsparungen, Leistung und Verfügbarkeit im großen Maßstab zu maximieren.
- Sentry – Diagnostizieren, beheben und optimieren Sie die Leistung von Lambda-Funktionen.
- Site24x7 – Erreichen Sie Echtzeit-Beobachtbarkeit in Ihren Lambda-Umgebungen

- Splunk – Sammelt hochauflösende Metriken mit niedriger Latenz für eine effiziente und effektive Überwachung von Lambda-Funktionen.
- Sumo Logic – Bietet Einblick in den Zustand und die Leistung von Serverless-Anwendungen.
- Salt Security – Vereinfacht die API-Status-Governance und die API-Sicherheit für Lambda-Funktionen durch automatisiertes Setup und Support für verschiedene Laufzeiten.

AWSVon verwaltete Erweiterungen

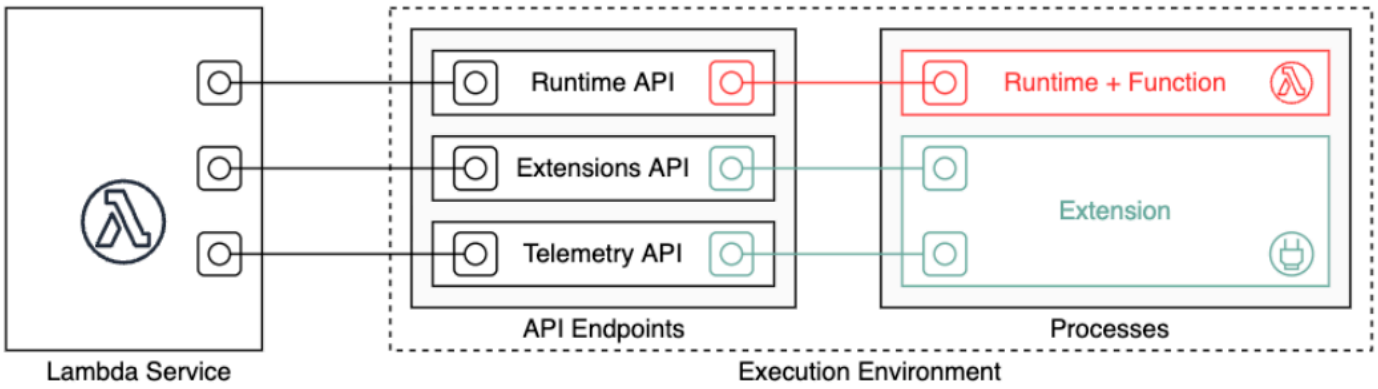
AWS stellt eigene verwaltete Erweiterungen bereit, einschließlich:

- AWS AppConfig – Verwenden Sie Feature-Flags und dynamische Daten, um Ihre Lambda-Funktionen zu aktualisieren. Sie können diese Erweiterung auch verwenden, um andere dynamische Konfigurationen wie Ops-Drosselung und Optimierung zu aktualisieren.
- Amazon CodeGuru Profiler – Verbessert die Anwendungsleistung und senkt den Preis, indem die teuerste Codezeile einer Anwendung lokalisiert wird und Empfehlungen zur Codeverbesserung erstellt werden.
- CloudWatch-Lambda-Insights – Überwachen, beheben und optimieren Sie die Leistung Ihrer Lambda-Funktionen durch automatisierte Dashboards.
- AWS Distro für OpenTelemetry (ADOT) – Aktiviert Funktionen zum Senden von Ablaufverfolgungsdaten an AWS-Überwachungsdienste wie AWS X-Ray und an Ziele, die OpenTelemetry unterstützen, wie Honeycomb und Lightstep.
- AWS-Parameter und -Geheimnisse – Sicheres Abrufen von Parametern aus AWS Systems Manager Parameter Store und Geheimnissen aus AWS Secrets Manager in Lambda-Funktionen.

Weitere Erweiterungs-Beispiele und Demo-Projekte finden Sie unter Erweiterungen für AWS Lambda.

Verwendung der Lambda-Erweiterungs-API zur Erstellung von Erweiterungen

Lambda-Funktionsautoren verwenden Erweiterungen, um Lambda in ihre bevorzugten Tools zur Überwachung, Beobachtbarkeit, Sicherheit und Governance zu integrieren. Funktionsautoren können Erweiterungen von AWS, AWS Partnern und Open-Source-Projekten verwenden. Weitere Informationen zur Verwendung von Erweiterungen finden Sie unter Einführung in AWS Lambda Erweiterungen im AWS Compute-Blog. In diesem Abschnitt wird beschrieben, wie Sie die Lambda-Erweiterungs-API, den Lebenszyklus der Lambda-Ausführungsumgebung und die Lambda-Erweiterungs-API-Referenz verwenden.



Als Erweiterungsautor können Sie die Lambda-Erweiterungs-API verwenden, um sich tief in die Lambda-Ausführungsumgebung zu integrieren. Ihre Erweiterung kann sich für Lebenszyklusereignisse für Funktions- und Ausführungsumgebung registrieren. Als Reaktion auf diese Ereignisse können Sie neue Prozesse starten, Logik ausführen und alle Phasen des Lambda-Lebenszyklus steuern und an ihnen teilnehmen: Initialisierung, Aufruf und Herunterfahren. Darüber hinaus können Sie die Laufzeitprotokoll-API verwenden, um einen Stream von Protokollen zu erhalten.

Eine Erweiterung wird als unabhängiger Prozess in der Ausführungsumgebung ausgeführt und kann weiterhin ausgeführt werden, nachdem der Funktionsaufruf vollständig verarbeitet wurde. Da Erweiterungen als Prozesse ausgeführt werden, können Sie diese in einer anderen Sprache als die Funktion schreiben. Es wird empfohlen, Erweiterungen mit einer kompilierten Sprache zu implementieren. In diesem Fall handelt es sich bei der Erweiterung um eine eigenständige Binärdatei, die mit unterstützten Laufzeiten kompatibel ist. Alle Lambda-Laufzeiten unterstützen Erweiterungen. Wenn Sie eine nicht kompilierte Sprache verwenden, stellen Sie sicher, dass Sie eine kompatible Laufzeit in die Erweiterung aufnehmen.

Lambda unterstützt auch interne Erweiterungen. Eine interne Erweiterung wird als separater Thread im Laufzeitprozess ausgeführt. Die Laufzeit startet und stoppt die interne Erweiterung. Eine alternative Möglichkeit zur Integration in die Lambda-Umgebung ist die Verwendung sprachspezifischer Umgebungsvariablen und Wrapper-Skripte. Sie können diese nutzen, um die Laufzeitumgebung zu konfigurieren und das Startup-Verhalten des Laufzeitprozesses zu verändern.

Sie können einer Funktion auf zwei Arten Erweiterungen hinzufügen. Bei einer Funktion, die als .zip-Dateiarchivbereitgestellt wird, stellen Sie Ihre Erweiterung als Ebenebereit. Bei einer Funktion, die als Container-Image definiert ist, fügen Sie die Erweiterungen Ihrem Container-Image hinzu.

> **Note**
>
> Beispiele für Erweiterungen und Wrapper-Skripte finden Sie unter AWS Lambda Erweiterungen im AWS GitHub Samples-Repository.

Themen

- Lebenszyklus der Lambda-Ausführungsumgebung
- Erweiterungs-API-Referenz

Lebenszyklus der Lambda-Ausführungsumgebung

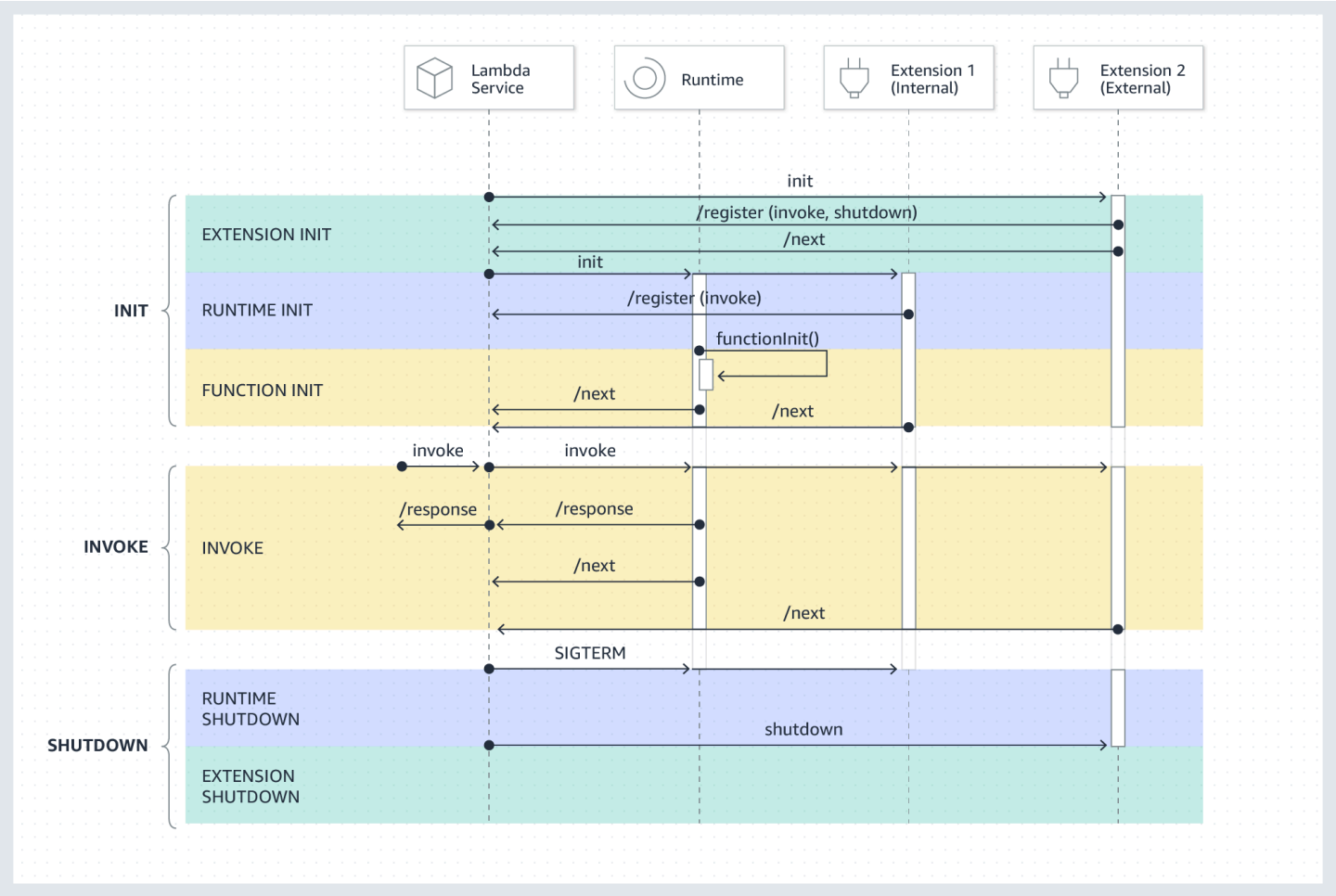
Der Lebenszyklus der Ausführungsumgebung umfasst die folgenden Phasen:

- **`Init`**: In dieser Phase erstellt oder hebt Lambda eine Ausführungsumgebung mit den konfigurierten Ressourcen auf, lädt den Code für die Funktion und alle Ebenen herunter, initialisiert alle Erweiterungen, initialisiert die Laufzeit und führt dann den Initialisierungscode der Funktion (der Code außerhalb des Haupthandlers) aus. Die Phase `Init` erfolgt entweder während des ersten Aufrufs oder vor Funktionsaufrufen, wenn Sie die bereitgestellte Parallelität aktiviert haben.

  Die `Init`-Phase ist in drei Unterphasen unterteilt: `Extension init`, `Runtime init` und `Function init`. Diese Unterphasen stellen sicher, dass alle Erweiterungen und die Laufzeit ihre Einrichtungs-Aufgaben abschließen, bevor der Funktionscode ausgeführt wird.
- **`Invoke`**: In dieser Phase ruft Lambda den Funktionshandler auf. Nachdem die Funktion vollständig ausgeführt wurde, bereitet sich Lambda auf die Verarbeitung eines weiteren Funktionsaufrufs vor.
- **`Shutdown`**: Diese Phase wird ausgelöst, wenn die Lambda-Funktion für einen bestimmten Zeitraum keine Aufrufe empfängt. In dieser `Shutdown`-Phase fährt Lambda die Laufzeit herunter,

warnt die Erweiterungen, damit sie sauber beendet werden können und entfernt dann die Umgebung. Lambda sendet ein `Shutdown`-Ereignis an jede Erweiterung, das der Erweiterung mitteilt, dass die Umgebung beendet wird.

Jede Phase beginnt mit einem Ereignis von Lambda zur Laufzeit und zu allen registrierten Erweiterungen. Die Laufzeit und jede Erweiterung zeigen den Abschluss durch Senden einer `Next`-API-Anfrage an. Lambda friert die Ausführungsumgebung ein, wenn jeder Prozess abgeschlossen ist und keine ausstehenden Ereignisse vorhanden sind.



Themen

- Init-Phase
- Invoke-Phase
- Shutdown-Phase
- Berechtigungen und Konfiguration

- Fehlerbehandlung
- Fehlerbehebung bei Erweiterungen

Init-Phase

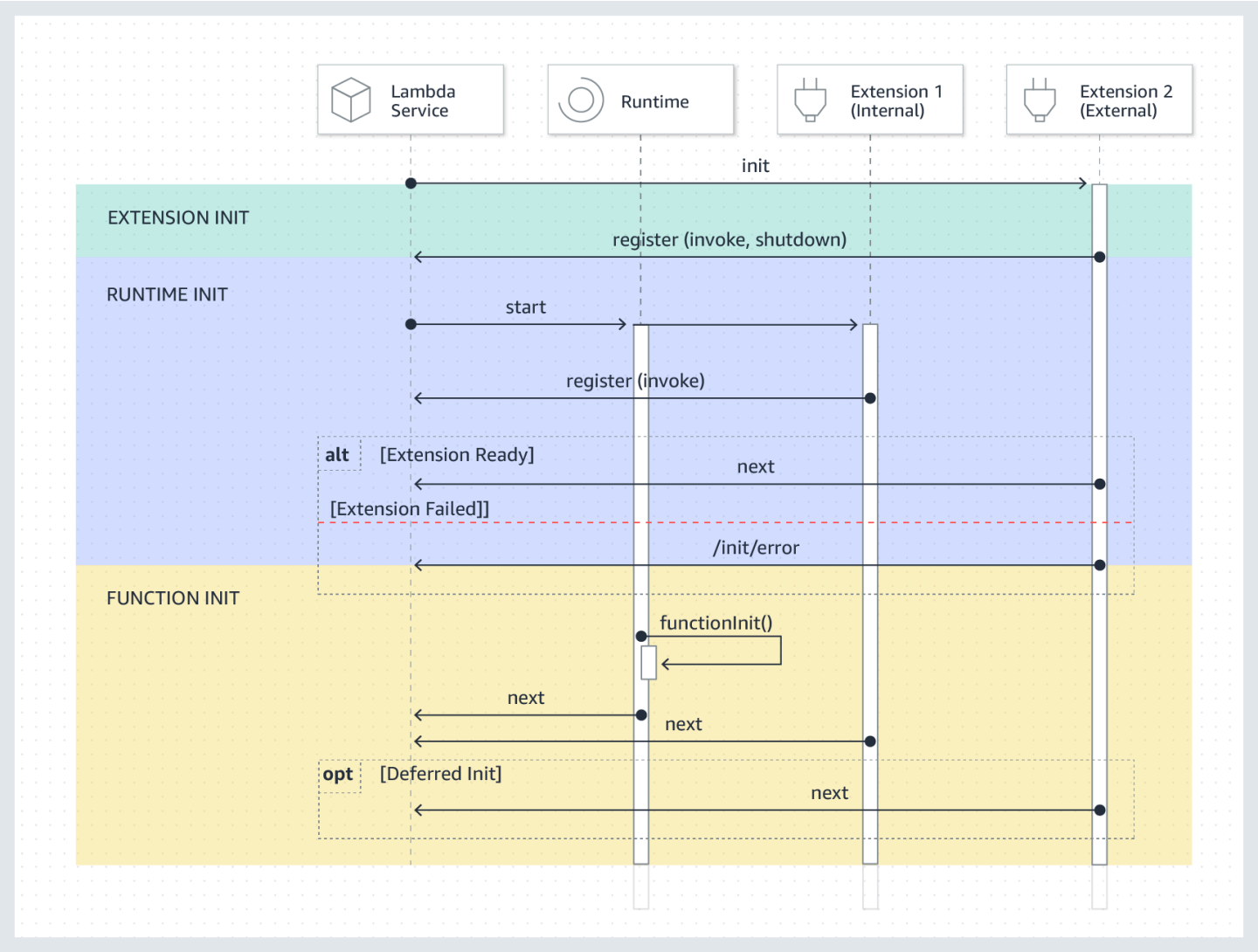
Während dieser `Extension init`-Phase muss sich jede Erweiterung bei Lambda registrieren, um Ereignisse zu empfangen. Lambda verwendet den vollständigen Dateinamen der Erweiterung, um zu überprüfen, ob die Erweiterung die Bootstrap-Sequenz abgeschlossen hat. Daher muss jeder `Register`-API-Aufruf den `Lambda-Extension-Name`-Header mit dem vollständigen Dateinamen der Erweiterung enthalten.

Sie können bis zu 10 Erweiterungen für eine Funktion registrieren. Dieses Limit wird durch den `Register`-API-Aufruf erzwungen.

Nachdem jede Erweiterung registriert ist, startet Lambda die `Runtime init`-Phase. Der Laufzeitprozess ruft `functionInit` auf, um die `Function init`-Phase zu starten.

Die `Init`-Phase wird nach der Laufzeit abgeschlossen und jede registrierte Erweiterung zeigt den Abschluss durch Senden einer `Next`-API-Anforderung an.

> **Note**
>
> Erweiterungen können ihre Initialisierung an jedem beliebigen Punkt in der `Init`-Phase abschließen.



Invoke-Phase

Wenn eine Lambda-Funktion als Antwort auf eine `Next`-API-Anforderung aufgerufen wird, sendet Lambda ein `Invoke`-Ereignis an die Laufzeitumgebung und an jede Erweiterung, die für das `Invoke`-Ereignis registriert ist.

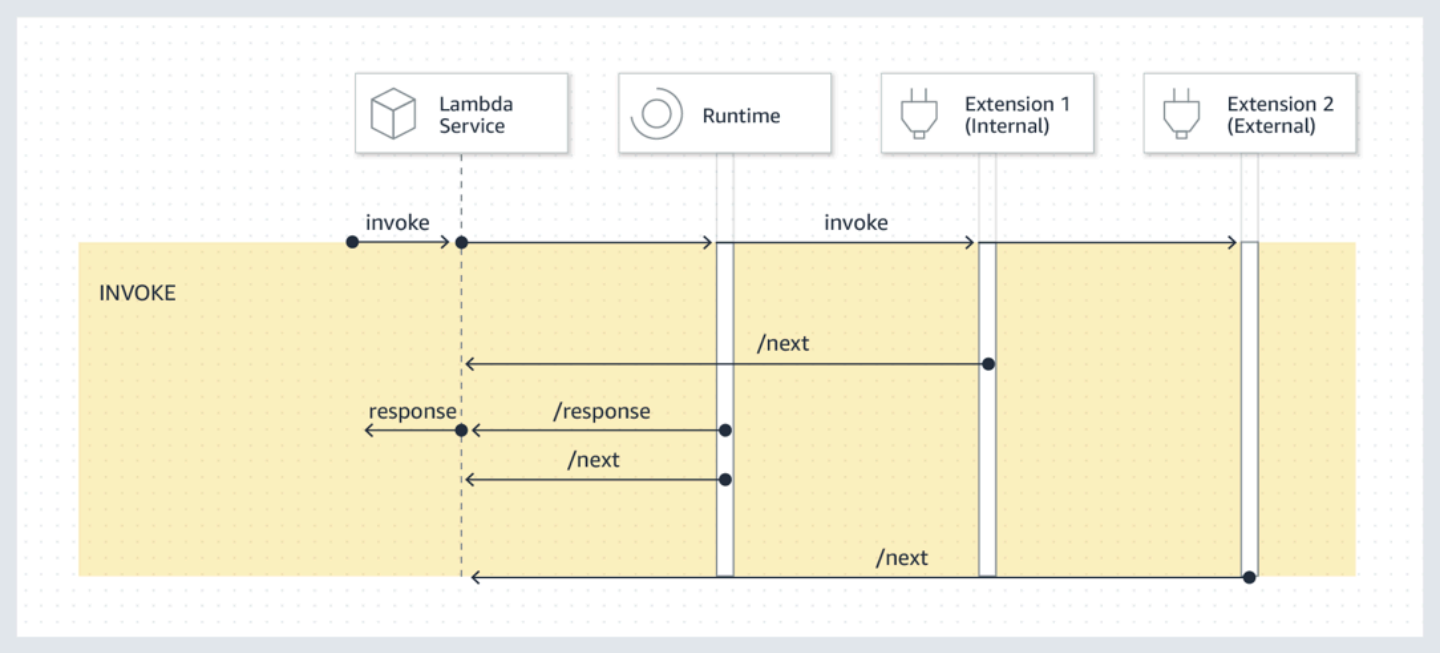
> **Note**
>
> Lambda Managed Instances: Erweiterungen für Funktionen von Lambda Managed Instances können sich nicht für das `Invoke` Ereignis registrieren. Da Lambda Managed Instances gleichzeitige Aufrufe innerhalb einer einzigen Ausführungsumgebung unterstützt, wird das `Invoke` Ereignis nicht unterstützt. Erweiterungen können sich nur für das Ereignis

registrieren. `Shutdown` Wenn Sie nachverfolgen möchten, wann Aufrufe beginnen und enden, verwenden Sie das `platform.report` Plattformereignis über die Telemetrie-API.

Während des Aufrufs werden externe Erweiterungen parallel zur Funktion ausgeführt. Sie werden auch weiter ausgeführt, nachdem die Funktion abgeschlossen ist. Auf diese Weise können Sie Diagnoseinformationen erfassen oder Protokolle, Metriken und Traces an einen Ort Ihrer Wahl senden.

Nachdem Erhalt der Funktionsantwort von der Laufzeitumgebung wird die Antwort von Lambda an den Client zurückgegeben, selbst wenn noch Erweiterungen ausgeführt werden.

Die `Invoke`-Phase endet nach der Laufzeit und alle Erweiterungen signalisieren durch Senden einer `Next`-API-Anforderung, dass sie ausgeführt wurden.



Ereignisnutzlast: Das Ereignis, das an die Laufzeit (und die Lambda-Funktion) gesendet wird, trägt die gesamte Anforderung, Header (z. B. `RequestId`) und Nutzlast. Das an jede Erweiterung gesendete Ereignis enthält Metadaten, die den Ereignisinhalt beschreiben. Dieses Lebenszyklusereignis umfasst den Typ des Ereignisses, die Zeit, zu der das Timeout der Funktion (`deadlineMs`), das `requestId`, die aufgerufene Funktion, den Amazon Resource Name (ARN) der aufgerufenen Funktion und die Tracing-Header.

Erweiterungen, die auf den Funktionsereigniskörper zugreifen möchten, können ein In-Laufzeit-SDK verwenden, das mit der Erweiterung kommuniziert. Funktionsentwickler verwenden das In-Laufzeit-SDK, um die Nutzlast an die Erweiterung zu senden, wenn die Funktion aufgerufen wird.

Hier ist ein Beispiel für eine Nutzlast:

```
{
    "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"
    }
 }
```

Begrenzung der Dauer: Die Timeout-Einstellung der Funktion begrenzt die Dauer der gesamten `Invoke`-Phase. Wenn Sie beispielsweise die Zeitüberschreitung für die Funktion auf 360 Sekunden festlegen, müssen die Funktion und alle Erweiterungen innerhalb von 360 Sekunden abgeschlossen werden. Beachten Sie, dass es keine unabhängige Post-Invoke-Phase gibt. Die Dauer ist die Gesamtzeit, die Ihre Laufzeit und alle Aufrufe Ihrer Erweiterungen benötigen und wird erst berechnet, wenn die Funktion und alle Erweiterungen abgeschlossen sind.

Performance impact and extension overhead (Leistungsauswirkungen und Erweiterungsoverhead): Erweiterungen können sich auf die Funktionsleistung auswirken. Als Autor einer Erweiterung haben Sie die Kontrolle über die Auswirkungen Ihrer Erweiterung auf die Leistung. Ein Beispiel: Eine Erweiterung führt rechenintensive Operationen aus. In diesem Fall könnten Sie feststellen, dass die Dauer des Aufrufs Ihrer Funktion zunimmt, da Erweiterung und Funktion sich CPU-Ressourcen teilen. Wenn Ihre Erweiterung umfangreiche Operationen ausführt, nachdem der Funktionsaufruf abgeschlossen ist, verlängert sich die Dauer des Funktionsaufrufs, da die `Invoke`-Phase fortgesetzt wird, bis alle Erweiterungen signalisieren, dass sie abgeschlossen sind.

> **Note**
>
> Lambda weist CPU-Leistung im Verhältnis zur Speichereinstellung der Funktion zu. Bei niedrigeren Speichereinstellungen wird möglicherweise eine erhöhte Ausführungs- und Initialisierungsdauer angezeigt, da die Funktions- und Erweiterungsprozesse um die gleichen

CPU-Ressourcen konkurrieren. Erhöhen Sie die Speichereinstellung, um die Ausführungs- und Initialisierungsdauer zu reduzieren.

Um die Leistungsbeeinträchtigung zu identifizieren, die durch Erweiterungen in der `Invoke`-Phase verursacht wird, gibt Lambda die `PostRuntimeExtensionsDuration`-Metrik aus. Diese Metrik misst die kumulative Zeit, die zwischen der `Next`-Laufzeit-API-Anforderung und der letzten `Next`-Erweiterungs-API-Anforderung vergeht. Verwenden Sie die `MaxMemoryUsed`-Metrik, um die Zunahme des verwendeten Speichers zu messen. Weitere Informationen zu Funktionsmetriken finden Sie unter Verwendung von CloudWatch-Metriken mit Lambda.

Funktionsentwickler können verschiedene Versionen ihrer Funktionen nebeneinander ausführen, um die Auswirkungen einer bestimmten Erweiterung zu verstehen. Wir empfehlen, dass Erweiterungsautoren den erwarteten Ressourcenverbrauch veröffentlichen, um Funktionsentwicklern die Auswahl einer geeigneten Erweiterung zu erleichtern.

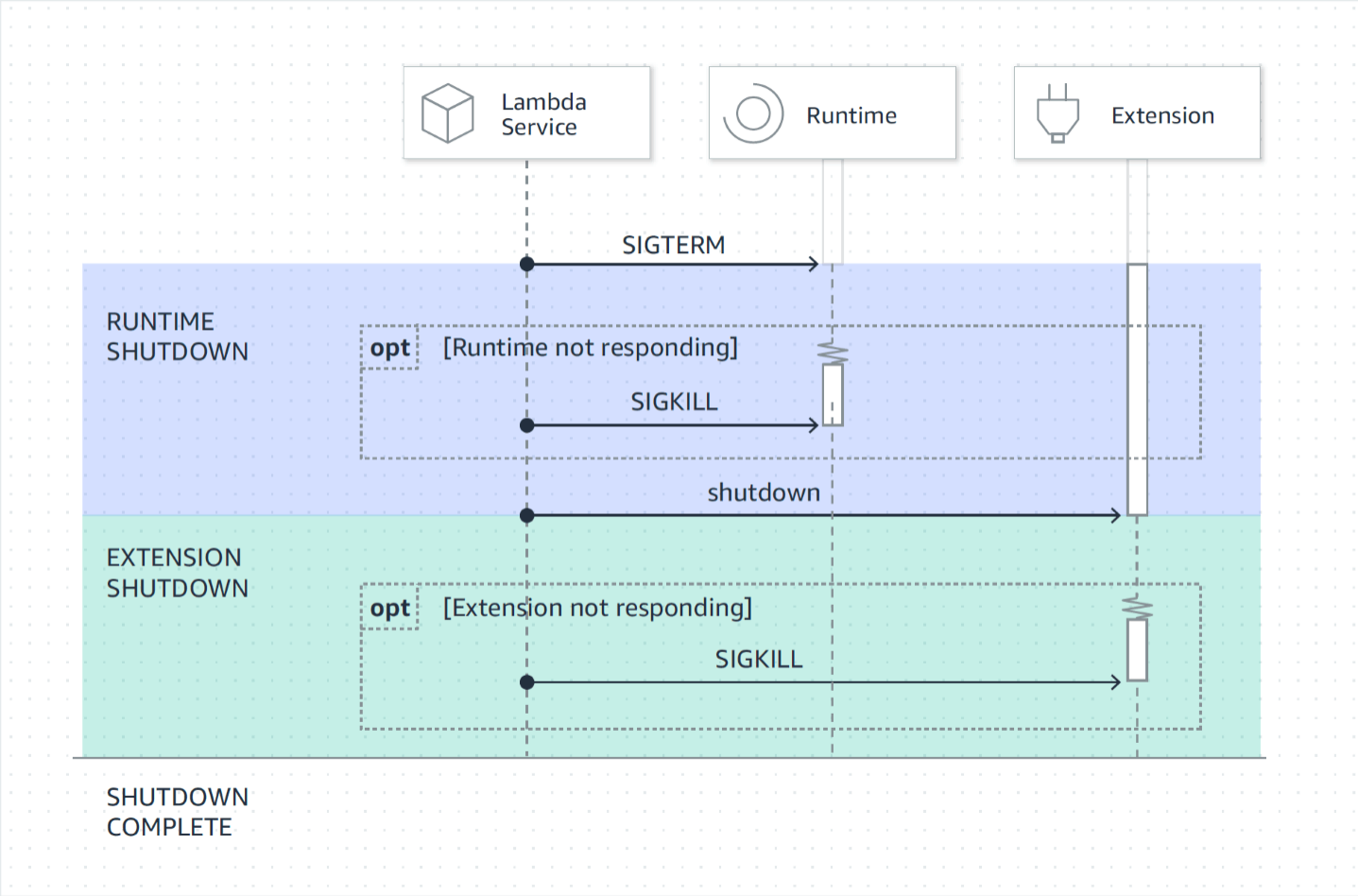
Shutdown-Phase

Wenn Lambda dabei ist, die Laufzeit zu beenden, sendet es ein `Shutdown` an die Laufzeitumgebung und dann an jede registrierte externe Erweiterung. Erweiterungen können diese Zeit für abschließende Bereinigungsaufgaben verwenden. Das `Shutdown`-Ereignis wird als Antwort auf eine `Next`-API-Anforderung gesendet.

Duration Limit (Begrenzung der Dauer): Die maximale Dauer der `Shutdown`-Phase hängt von der Konfiguration der registrierten Erweiterungen ab:

- 0 ms – Eine Funktion ohne registrierte Erweiterungen
- 500 ms – Eine Funktion mit einer registrierten internen Erweiterung
- 2.000 ms – Eine Funktion mit einer oder mehreren registrierten externen Erweiterungen

Wenn die Laufzeit oder eine Erweiterung nicht innerhalb des Limits auf das `Shutdown`-Ereignis reagiert, beendet Lambda den Prozess mit einem `SIGKILL`-Signal.



Event Payload (Ereignis-Nutzlast): Das `Shutdown`-Ereignis enthält den Grund für das Abschalten und die verbleibende Zeit in Millisekunden.

Das `shutdownReason` beinhaltet die folgenden Werte:

- SPINDOWN – Normale Abschaltung
- TIMEOUT – Zeitlimit abgelaufen
- FAILURE – Fehlerzustand, etwa ein `out-of-memory`-Ereignis

```
{
  "eventType": "SHUTDOWN",
  "shutdownReason": "reason for shutdown",
  "deadlineMs": "the time and date that the function times out in Unix time
 milliseconds"
}
```

Berechtigungen und Konfiguration

Erweiterungen werden in derselben Ausführungsumgebung wie die Lambda-Funktion ausgeführt. Erweiterungen teilen auch Ressourcen mit der Funktion, wie CPU, Arbeitsspeicher und `/tmp`-Datenträgerspeicher. Darüber hinaus verwenden Erweiterungen dieselbe AWS Identity and Access Management (IAM-) Rolle und denselben Sicherheitskontext wie die Funktion.

File system and network access permissions (Dateisystem- und Netzwerkzugriffsberechtigungen): Erweiterungen werden in demselben Dateisystem- und Netzwerk-Namespace wie die Funktionslaufzeit ausgeführt. Dies bedeutet, dass Erweiterungen mit dem zugehörigen Betriebssystem kompatibel sein müssen. Wenn für eine Erweiterung zusätzliche ausgehende Netzwerkverkehrsregeln erforderlich sind, müssen Sie diese Regeln auf die Funktionskonfiguration anwenden.

> **Note**
>
> Da das Funktionscode-Verzeichnis schreibgeschützt ist, können Erweiterungen den Funktionscode nicht ändern.

Environment variables (Umgebungsvariablen): Erweiterungen können auf die Umgebungsvariablen der Funktion zugreifen, mit Ausnahme der folgenden Variablen, die für den Laufzeitprozess spezifisch sind:

- `AWS_EXECUTION_ENV`
- `AWS_LAMBDA_LOG_GROUP_NAME`
- `AWS_LAMBDA_LOG_STREAM_NAME`
- `AWS_LAMBDA_MAX_CONCURRENCY`(Nur verwaltete Lambda-Instanzen)
- `AWS_XRAY_CONTEXT_MISSING`
- `AWS_XRAY_DAEMON_ADDRESS`
- `LAMBDA_RUNTIME_DIR`
- `LAMBDA_TASK_ROOT`
- `_AWS_XRAY_DAEMON_ADDRESS`
- `_AWS_XRAY_DAEMON_PORT`

- `_HANDLER`

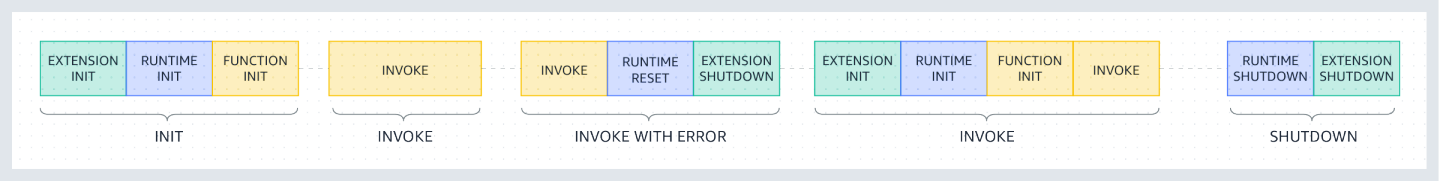
> **Note**
>
> Erkennung von Lambda Managed Instances: Erweiterungen können anhand der `AWS_LAMBDA_INITIALIZATION_TYPE` Umgebungsvariablen überprüfen, ob sie auf Lambda Managed Instances oder auf Lambda-Funktionen (Standard) ausgeführt werden. Dies ist die empfohlene Methode für Erweiterungen, um ihr Verhalten an den Typ der Ausführungsumgebung anzupassen.

Fehlerbehandlung

Initialization failures (Initialisierungsfehler): Wenn eine Erweiterung fehlschlägt, startet Lambda die Ausführungsumgebung neu, um konsistentes Verhalten zu erzwingen und Fail Fast für Erweiterungen zu unterstützen. Für einige Kunden müssen die Erweiterungen auch geschäftskritische Anforderungen wie Protokollierung, Sicherheit, Governance und Telemetrieerfassung erfüllen.

Invoke failures (Aufruffehler) (z. B. kein Speicher mehr, Funktions-Timeout): Da Erweiterungen Ressourcen mit der Laufzeit gemeinsam nutzen, sind sie eventuell von Speichermangel betroffen. Wenn die Laufzeit ausfällt, nehmen alle Erweiterungen und die Laufzeit selbst an der `Shutdown`-Phase teil. Darüber hinaus wird die Laufzeitumgebung entweder automatisch als Teil des aktuellen Aufrufs oder über einen verzögerten Neuinitialisierungsmechanismus neu gestartet.

Wenn bei `Invoke` ein Fehler (z. B. eine Funktions-Zeitüberschreitung oder Laufzeitfehler) auftritt, führt der Lambda-Service einen Neustart durch. Der Reset verhält sich wie ein `Shutdown`-Ereignis. Zuerst beendet Lambda die Laufzeit und sendet dann ein `Shutdown`-Ereignis an jede registrierte externe Erweiterung. Das Ereignis enthält den Grund für das Abschalten. Wenn diese Umgebung für einen neuen Aufruf verwendet wird, werden die Erweiterung und die Laufzeit als Teil des nächsten Aufrufers neu initialisiert.



Eine ausführlichere Erläuterung des vorherigen Diagramms finden Sie unter Fehler während der Aufrufphase.

Erweiterungsprotokolle: Lambda sendet die Protokollausgabe von Erweiterungen an CloudWatch Logs. Lambda generiert auch ein zusätzliches Protokollereignis für jede Erweiterung während `Init`. Das Protokollereignis zeichnet den Namen und die Registrierungseinstellung (Ereignis, Konfiguration) bei Erfolg oder die Fehlerursache bei einem Fehler auf.

Fehlerbehebung bei Erweiterungen

- Wenn eine `Register`-Anforderung fehlschlägt, stellen Sie sicher, dass der `Lambda-Extension-Name`-Header im `Register`-API-Aufruf den vollständigen Dateinamen der Erweiterung enthält.
- Wenn die `Register`-Anforderung für eine interne Erweiterung fehlschlägt, stellen Sie sicher, dass sich die Anforderung nicht für das `Shutdown`-Ereignis registriert.

Erweiterungs-API-Referenz

Die OpenAPI-Spezifikation für die Erweiterungs-API Version 2020-01-01 finden Sie hier: extensions-api.zip

Sie können den Wert des API-Endpunkts aus der `AWS_LAMBDA_RUNTIME_API`-Umgebungsvariablen abrufen. Um eine `Register`-Anforderung zu senden, verwenden Sie das Präfix `2020-01-01/` vor jedem API-Pfad. Zum Beispiel:

```
http://${AWS_LAMBDA_RUNTIME_API}/2020-01-01/extension/register
```

API-Methoden

- Registrieren
- Next
- Init-Fehler
- Exit-Fehler

Registrieren

Während `Extension init` müssen sich alle Erweiterungen bei Lambda registrieren, um Ereignisse zu empfangen. Lambda verwendet den vollständigen Dateinamen der Erweiterung, um zu überprüfen, ob die Erweiterung die Bootstrap-Sequenz abgeschlossen hat. Daher muss jeder `Register`-API-Aufruf den `Lambda-Extension-Name`-Header mit dem vollständigen Dateinamen der Erweiterung enthalten.

Interne Erweiterungen werden vom Laufzeitprozess gestartet und gestoppt, sodass sie sich nicht für das Shutdown-Ereignis registrieren können.

Pfad – `/extension/register`

Methode – POST

Anfordern von Headern

- `Lambda-Extension-Name` – Der vollständige Dateiname der Erweiterung. Erforderlich: ja Typ: Zeichenkette
- `Lambda-Extension-Accept-Feature` – Verwenden Sie dies, um optionale Erweiterungsfunktionen bei der Registrierung anzugeben. Erforderlich: Nein. Typ: durch Kommas getrennte Zeichenfolge. Funktionen, die mit dieser Einstellung angegeben werden können:
  - `accountId` – Falls angegeben, enthält die Registrierungsantwort der Erweiterung die Konto-ID, die der Lambda-Funktion zugeordnet ist, für die Sie die Erweiterung registrieren.

Anfrage von Textparametern

- `events` – Array der Ereignisse, für die die Registrierung durchgeführt werden soll. Erforderlich: Nein Typ: Zeichenfolge-Array Gültige Zeichenfolgen: `INVOKE`, `SHUTDOWN`.

> **Note**
>
> Lambda Managed Instances: Erweiterungen für Funktionen von Lambda Managed Instances können sich nur für das `SHUTDOWN` Ereignis registrieren. Der Versuch, sich für das `INVOKE` Ereignis zu registrieren, führt zu einem Fehler. Dies liegt daran, dass Lambda Managed Instances gleichzeitige Aufrufe innerhalb einer einzigen Ausführungsumgebung unterstützt.

Antwort-Header

- `Lambda-Extension-Identifier` – Generierte eindeutige Agent-ID (UUID-Zeichenfolge), die für alle nachfolgenden Anforderungen erforderlich ist.

Antwortcodes

- 200 – Antworttext enthält den Funktionsnamen, die Funktionsversion und den Namen des Handlers.
- 400 – Ungültige Anfrage
- 403 – Verboten
- 500 – Container-Fehler. Nicht wiederherstellbarer Zustand. Die Erweiterung sollte umgehend beendet werden.

Example Beispielanfragetext

```
{
    'events': [ 'INVOKE', 'SHUTDOWN']
}
```

Example Beispielantworttext

```
{
    "functionName": "helloWorld",
    "functionVersion": "$LATEST",
    "handler": "lambda_function.lambda_handler"
}
```

Example Beispiel-Antworttext mit optionalem AccountID-Feature

```
{
    "functionName": "helloWorld",
    "functionVersion": "$LATEST",
    "handler": "lambda_function.lambda_handler",
    "accountId": "123456789012"
}
```

Next

Erweiterungen senden eine `Next`-API-Anforderung für den Empfang des nächsten Ereignisses, bei dem es sich um ein `Invoke`-Ereignis oder ein `Shutdown`-Ereignis handeln kann. Der Antworttext enthält die Nutzlast, bei der es sich um ein JSON-Dokument handelt, das Ereignisdaten enthält.

Die Erweiterung sendet eine `Next`-API-Anforderung, um zu signalisieren, dass sie bereit ist, neue Ereignisse zu empfangen. Das ist ein blockierender Aufruf.

Legen Sie für den GET-Aufruf kein Timeout fest, da die Erweiterung für einen bestimmten Zeitraum angehalten werden kann, bis ein Ereignis zurückgegeben werden soll.

Pfad – `/extension/event/next`

Methode – GET

Anfordern von Headern

- `Lambda-Extension-Identifier` – Eindeutiger Bezeichner für die Erweiterung (UUID-Zeichenfolge). Erforderlich: ja Typ: UUID-Zeichenfolge

Antwort-Header

- `Lambda-Extension-Event-Identifier` – Eindeutiger Bezeichner für das Ereignis (UUID-Zeichenfolge).

Antwortcodes

- 200 – Die Antwort enthält Informationen über das nächste Ereignis (`EventInvoke` oder `EventShutdown`).
- 403 – Verboten
- 500 – Container-Fehler. Nicht wiederherstellbarer Zustand. Die Erweiterung sollte umgehend beendet werden.

Init-Fehler

Die Erweiterung verwendet diese Methode, um einen Initialisierungsfehler an Lambda zu melden. Rufen Sie sie auf, wenn die Erweiterung nach der Registrierung nicht initialisiert werden kann. Nachdem der Fehler von Lambda empfangen wurde, sind keine weiteren API-Aufrufe erfolgreich. Die Erweiterung sollte beendet werden, nachdem sie die Antwort von Lambda erhalten hat.

Pfad – `/extension/init/error`

Methode – POST

Anfordern von Headern

- `Lambda-Extension-Identifier` – Eindeutiger Bezeichner für die Erweiterung. Erforderlich: ja Typ: UUID-Zeichenfolge
- `Lambda-Extension-Function-Error-Type` – Der Fehlertyp, auf den die Erweiterung gestoßen ist. Erforderlich: ja Dieser Header besteht aus einem Zeichenfolgen-Wert. Lambda akzeptiert jede Zeichenfolge, aber wir empfehlen ein Format von <category.reason>. Beispiel:
  - Erweiterung. NoSuchHandler
  - Erweiterung. APIKeyNotFound
  - Erweiterung. ConfigInvalid
  - Erweiterung. UnknownReason

Anfrage von Textparametern

- `ErrorRequest` – Informationen über den Fehler. Erforderlich: Nein

Dieses Feld ist ein JSON-Objekt mit der folgenden Struktur:

```
{
      errorMessage: string (text description of the error),
      errorType: string,
      stackTrace: array of strings
}
```

Beachten Sie, dass Lambda jeden Wert für `errorType` akzeptiert.

Das folgende Beispiel zeigt eine Lambda-Funktionsfehlermeldung, in der die Funktion die im Aufruf bereitgestellten Ereignisdaten nicht analysieren konnte.

Example Funktionsfehler

```
{
      "errorMessage" : "Error parsing event data.",
      "errorType" : "InvalidEventDataException",
      "stackTrace": [ ]
}
```

Antwortcodes

- 202 – Akzeptiert
- 400 – Ungültige Anfrage
- 403 – Verboten
- 500 – Container-Fehler. Nicht wiederherstellbarer Zustand. Die Erweiterung sollte umgehend beendet werden.

Exit-Fehler

Die Erweiterung verwendet diese Methode, um Lambda vor dem Beenden einen Fehler zu melden. Rufen Sie sie auf, wenn ein unerwarteter Fehler auftritt. Nachdem der Fehler von Lambda empfangen wurde, sind keine weiteren API-Aufrufe erfolgreich. Die Erweiterung sollte beendet werden, nachdem sie die Antwort von Lambda erhalten hat.

Pfad – `/extension/exit/error`

Methode – POST

Anfordern von Headern

- `Lambda-Extension-Identifier` – Eindeutiger Bezeichner für die Erweiterung. Erforderlich: ja Typ: UUID-Zeichenfolge
- `Lambda-Extension-Function-Error-Type` – Der Fehlertyp, auf den die Erweiterung gestoßen ist. Erforderlich: ja Dieser Header besteht aus einem Zeichenfolgen-Wert. Lambda akzeptiert jede Zeichenfolge, aber wir empfehlen ein Format von <category.reason>. Beispiel:
  - Erweiterung. NoSuchHandler
  - Erweiterung. APIKeyNotFound
  - Erweiterung. ConfigInvalid
  - Erweiterung. UnknownReason

Anfrage von Textparametern

- `ErrorRequest` – Informationen über den Fehler. Erforderlich: Nein

Dieses Feld ist ein JSON-Objekt mit der folgenden Struktur:

```
{
      errorMessage: string (text description of the error),
      errorType: string,
      stackTrace: array of strings
}
```

Beachten Sie, dass Lambda jeden Wert für `errorType` akzeptiert.

Das folgende Beispiel zeigt eine Lambda-Funktionsfehlermeldung, in der die Funktion die im Aufruf bereitgestellten Ereignisdaten nicht analysieren konnte.

Example Funktionsfehler

```
{
      "errorMessage" : "Error parsing event data.",
      "errorType" : "InvalidEventDataException",
      "stackTrace": [ ]
}
```

Antwortcodes

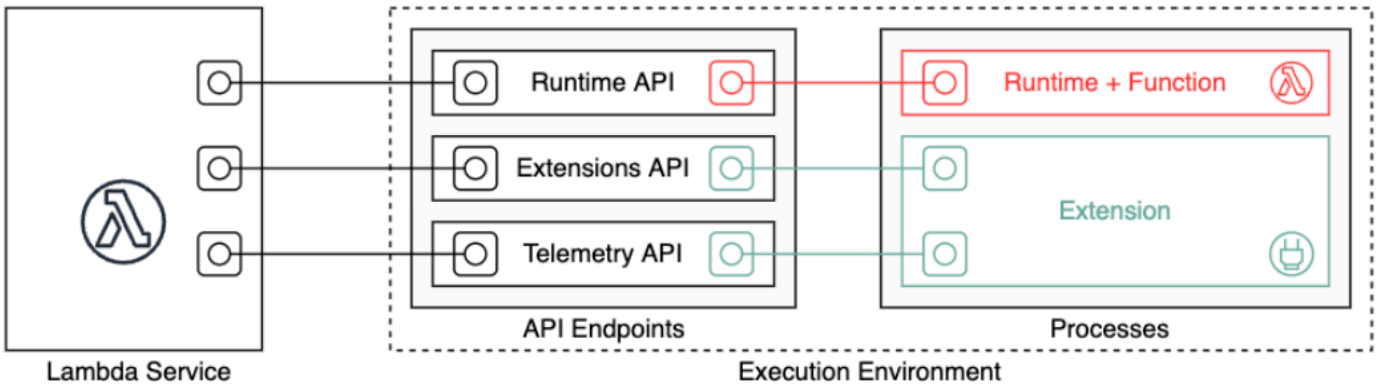
- 202 – Akzeptiert
- 400 – Ungültige Anfrage
- 403 – Verboten
- 500 – Container-Fehler. Nicht wiederherstellbarer Zustand. Die Erweiterung sollte umgehend beendet werden.

Zugriff auf Echtzeit-Telemetriedaten für Erweiterungen über die Telemetrie-API

Über die Telemetry API können Ihre Erweiterungen Telemetriedaten direkt von Lambda empfangen. Während der Initialisierung und des Aufrufs von Funktionen erfasst Lambda automatisch Telemetriedaten wie Protokolle, Plattform-Metriken und Plattform-Ablaufverfolgungen. Über die Telemetry API können Erweiterungen diese Telemetriedaten direkt von Lambda in nahezu Echtzeit abrufen.

Sie können die Telemetrie-Streams für Ihre Lambda-Erweiterungen direkt in der Lambda-Ausführungsumgebung abonnieren. Nach dem Abonnement sendet Lambda automatisch alle Telemetriedaten an Ihre Erweiterungen. Sie können diese Daten dann verarbeiten, filtern und an Ihr bevorzugtes Ziel senden, z. B. einen Amazon Simple Storage Service (Amazon S3)-Bucket oder einen Drittanbieter von Beobachtbarkeits-Tools.

Das folgende Diagramm zeigt, wie die Extension API und die Telemetry API Erweiterungen von der Ausführungsumgebung aus mit Lambda verbinden. Die Runtime API verbindet zudem die Laufzeit und Funktion mit Lambda.



> **Important**
>
> Die Lambda-Telemetrie-API ersetzt die Lambda-Protokoll-API. Die Protokoll-API bleibt zwar voll funktionsfähig, wir empfehlen jedoch, in Zukunft nur die Telemetrie-API zu verwenden. Sie können Ihre Erweiterung für einen Telemetrie-Stream entweder über die Telemetrie-API oder die Protokoll-API abonnieren. Nach dem Abonnieren mit einer dieser APIs wird bei jedem Versuch APIs, ein Abonnement über die andere API abzuschließen, ein Fehler zurückgegeben.

Anforderung an die Schemaversion von Lambda Managed Instances

Lambda Managed Instances unterstützen nur die `2025-01-29` Schemaversion der Telemetrie-API. Wenn Sie Telemetriestreams für Funktionen der verwalteten Instanz abonnieren, müssen Sie dies `"schemaVersion": "2025-01-29"` in Ihrer Abonnementanfrage verwenden. Die Verwendung früherer Schemaversionen führt dazu, dass Ereignisse von Lambda zurückgewiesen werden.
Die `2025-01-29` Schemaversion ist abwärtskompatibel und kann sowohl mit Lambda Managed Instances als auch mit Lambda-Funktionen (Standard) verwendet werden. Wir empfehlen, diese Version für alle neuen Erweiterungen zu verwenden, um die Kompatibilität zwischen beiden Bereitstellungsmodellen sicherzustellen.

Erweiterungen können die Telemetrie-API verwenden, um drei verschiedene Telemetrie-Streams zu abonnieren:

- Plattformtelemetrie – Protokolle, Metriken und Ablaufverfolgungen, die Ereignisse und Fehler im Zusammenhang mit dem Laufzeitlebenszyklus der Ausführungsumgebung, dem Erweiterungslebenszyklus und Funktionsaufrufen beschreiben.
- Funktionsprotokolle – Benutzerdefinierte Protokolle, die der Lambda-Funktionscode generiert.
- Erweiterungsprotokolle – Benutzerdefinierte Protokolle, die der Lambda-Erweiterungscode generiert.

Note

Lambda sendet Logs und Metriken an CloudWatch X-Ray und Traces an X-Ray (sofern Sie Tracing aktiviert haben), auch wenn eine Erweiterung Telemetriestreams abonniert.

Sections

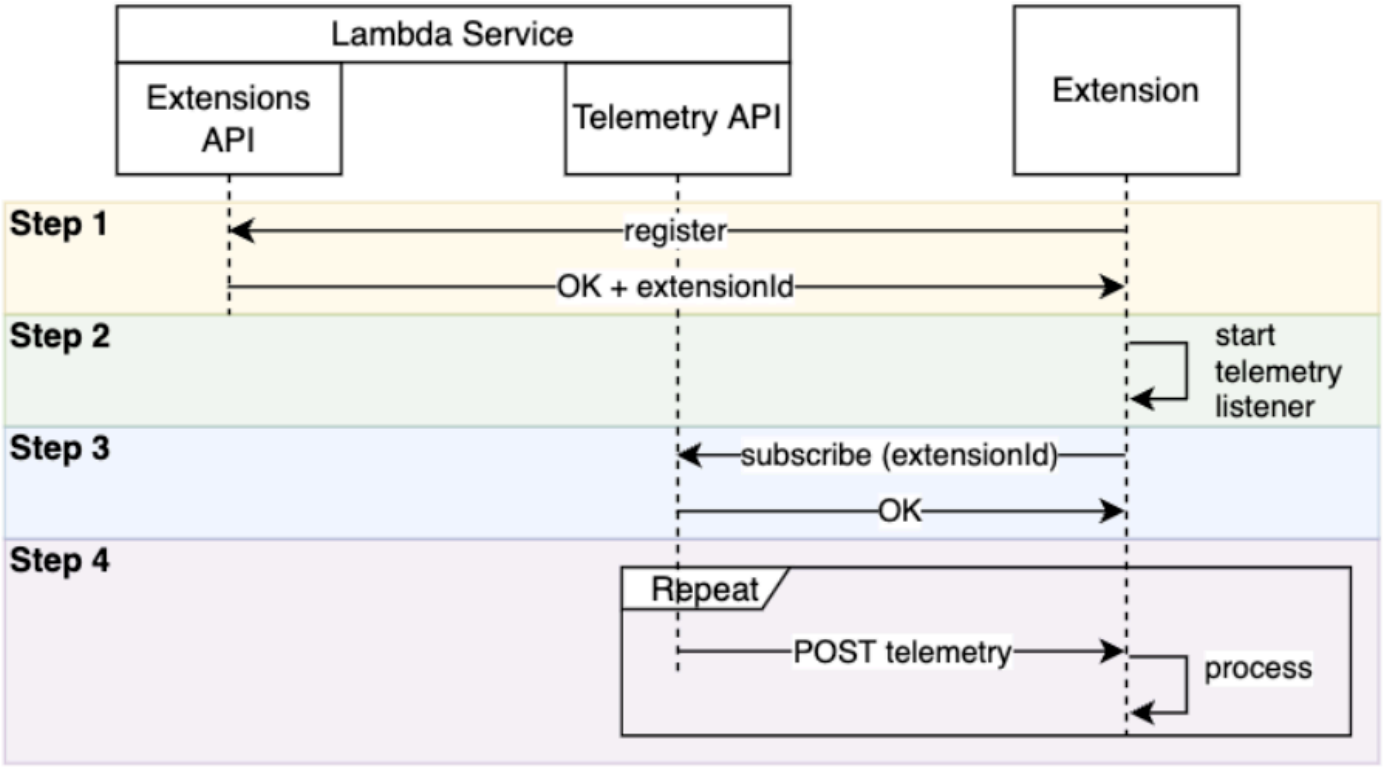
- Erstellen von Erweiterungen mithilfe der Telemetrie-API
- Registrieren Ihrer Erweiterung
- Erstellen eines Telemetrie-Listeners
- Festlegen eines Zielprotokolls
- Konfiguration der Speichernutzung und Pufferung

- Senden einer Abonnementanfrage an die Telemetrie-API
- Eingehende Telemetrie-API-Nachrichten
- Referenz zur Lambda-Telemetrie-API
- Referenz zum Event-Schema der Lambda-Telemetrie-API
- Konvertieren von Event-Objekte der Lambda-Telemetrie-API in OpenTelemetry-Spans
- Verwendung der Lambda-Protokoll-API

Erstellen von Erweiterungen mithilfe der Telemetrie-API

Lambda-Erweiterungen werden als unabhängige Prozesse in der Ausführungsumgebung ausgeführt. Erweiterungen können nach Abschluss des Funktionsaufrufs weiter ausgeführt werden. Da Erweiterungen separate Prozesse sind, können Sie diese in einer anderen Sprache als dem Funktionscode schreiben. Es wird empfohlen, Erweiterungen mit einer kompilierten Sprache wie Golang oder Rust zu schreiben. Auf diese Weise ist die Erweiterung eine in sich geschlossene Binärdatei, die mit jeder unterstützten Laufzeitumgebung kompatibel sein kann.

Das folgende Diagramm zeigt einen vierstufigen Prozess zum Erstellen einer Erweiterung, die Telemetriedaten mithilfe der Telemetrie-API empfängt und verarbeitet.



Hier sind die einzelnen Schritte im Detail:

1. Registrieren Sie Ihre Erweiterung mithilfe von the section called “Erweiterungs-API”. Dadurch erhalten Sie einen `Lambda-Extension-Identifier`, den Sie in den folgenden Schritten benötigen. Weitere Informationen zum Registrieren einer Erweiterung finden Sie unter the section called “Registrieren Ihrer Erweiterung”.
2. Erstellen eines Telemetrie-Listeners. Dies kann ein einfacher HTTP- oder TCP-Server sein. Lambda verwendet den URI des Telemetrie-Listeners, um Telemetriedaten an Ihre Erweiterung zu senden. Weitere Informationen finden Sie unter the section called “Erstellen eines Telemetrie-Listeners”.
3. Verwenden Sie die Subscribe API in der Telemetry API, um die gewünschten Telemetrie-Streams für Ihre Erweiterung zu abonnieren. Für diesen Schritt benötigen Sie den URI Ihres Telemetrie-Listeners. Weitere Informationen finden Sie unter the section called “Senden einer Abonnementanfrage an die Telemetrie-API”.
4. Rufen Sie Telemetriedaten von Lambda über den Telemetrie-Listener ab. Sie können diese Daten beliebig weiterverarbeiten, z. B. die Daten an Amazon S3 oder an einen externen Beobachtbarkeits-Service weiterleiten.

> **Note**
>
> Die Ausführungsumgebung einer Lambda-Funktion kann im Rahmen ihres Lebenszyklus mehrmals gestartet und angehalten werden. Im Allgemeinen wird Ihr Erweiterungscode während Funktionsaufrufen und auch bis zu 2 Sekunden während der Herunterfahrphase ausgeführt. Wir empfehlen, Telemetriedaten zu Batches zusammenzufassen, sobald sie bei Ihrem Listener eingehen. Verwenden Sie dann die Lebenszyklusereignisse `Invoke` und `Shutdown`, um jeden Batch an die gewünschten Ziele zu senden.

Registrieren Ihrer Erweiterung

Bevor Sie Telemetriedaten abonnieren können, müssen Sie Ihre Lambda-Erweiterung registrieren. Die Registrierung erfolgt während der Initialisierungsphase der Erweiterung. Das folgende Beispiel zeigt eine HTTP-Anfrage zur Registrierung einer Erweiterung.

```
POST http://${AWS_LAMBDA_RUNTIME_API}/2020-01-01/extension/register
 Lambda-Extension-Name: lambda_extension_name
{
```

```
    'events': [ 'INVOKE', 'SHUTDOWN']
}
```

Wenn die Anfrage erfolgreich ist, erhält der Subscriber eine Erfolgsantwort von HTTP 200. Der Antwort-Header enthält den `Lambda-Extension-Identifier`. Der Antworttext enthält weitere Eigenschaften der Funktion.

```
HTTP/1.1 200 OK
Lambda-Extension-Identifier: a1b2c3d4-5678-90ab-cdef-EXAMPLE11111
{
    "functionName": "lambda_function",
    "functionVersion": "$LATEST",
    "handler": "lambda_handler",
    "accountId": "123456789012"
}
```

Weitere Informationen hierzu finden Sie unter the section called “Erweiterungs-API-Referenz”.

Erstellen eines Telemetrie-Listeners

Ihre Lambda-Erweiterung muss über einen Listener verfügen, der eingehende Anfragen der Telemetrie-API verarbeitet. Der folgende Code zeigt eine Beispielimplementierung eines Telemetrie-Listeners in 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
}
```

Festlegen eines Zielprotokolls

Wenn Sie den Empfang von Telemetrie mithilfe der Telemetrie-API abonnieren, können Sie zusätzlich zur Ziel-URI ein Zielprotokoll angeben:

```
{
    "destination": {
        "protocol": "HTTP",
        "URI": "http://sandbox.localdomain:8080"
    }
}
```

Lambda akzeptiert zwei Protokolle für den Empfang von Telemetrie:

- HTTP (empfohlen) – Lambda übermittelt Telemetrie an einen lokalen HTTP-Endpunkt (`http://sandbox.localdomain:${PORT}/${PATH}`) als ein Array von Datensätzen im JSON-Format. Der Parameter `$PATH` ist optional. Lambda-unterstützt nur HTTP, nicht HTTPS. Lambda übermittelt Telemetrie über POST-Anfragen.
- TCP – Lambda übermittelt Telemtrie an einen TCP-Anschluss im Newline-delimited-JSON-(NDJSON)-Format.

> **Note**
>
> Wir empfehlen dringend, HTTP und nicht TCP zu verwenden. Mit TCP kann die Lambda-Plattform nicht bestätigen, dass es Telemetrie an die Anwendungsebene übermittelt. Wenn Ihre Erweiterung abstürzt, verlieren Sie möglicherweise Telemetriedaten. Bei HTTP gibt es diese Einschränkung nicht.

Bevor Sie sich für den Empfang von Telemetriedaten registrieren, richten Sie den lokalen HTTP-Listener oder TCP-Port ein. Beachten Sie bei der Einrichtung Folgendes:

- Lambda sendet Telemetriedaten nur an Ziele, die sich innerhalb der Ausführungsumgebung befinden.
- Lambda wiederholt den Versuch, die Telemetriedaten (mit Backoff) zu senden, wenn es keinen Listener gibt oder wenn die POST-Anfrage zu einem Fehler führt. Wenn der Telemetrie-Listener abstürzt, erhält er nach dem Neustart der Ausführungsumgebung durch Lambda wieder Telemetriedaten.
- Lambda reserviert Port 9001. Es gibt keine weiteren Einschränkungen oder Empfehlungen für Portnummern.

Konfiguration der Speichernutzung und Pufferung

Der Speicherverbrauch in einer Ausführungsumgebung wächst linear mit der Anzahl der Abonnenten. Abonnements verbrauchen Speicherressourcen, da jedes einen neuen Speicherpuffer zum Speichern von Telemetriedaten öffnet. Die Pufferspeichernutzung wird zum Gesamtspeicherverbrauch in der Ausführungsumgebung gezählt.

Wenn Sie den Empfang von Telemetriedaten mithilfe der Telemetry API abonnieren, können Sie Telemetriedaten puffern und sie in Batches an Abonnenten übermitteln. Um die Speichernutzung zu optimieren, können Sie eine Pufferungskonfiguration angeben:

```
{
    "buffering": {
        "maxBytes": 256*1024,
        "maxItems": 1000,
        "timeoutMs": 100
    }
}
```

Parameter	Description	Standardwerte und Limits
`maxBytes`	Das maximale Telemetri evolumen (in Bytes), das im Speicher gepuffert werden soll.	Standard: 262 144 Mindestwert: 262 144 Höchstwert: 1 048 576
`maxItems`	Die maximale Anzahl von Ereignissen, die im Speicher gepuffert werden sollen.	Standard: 10 000 Mindestwert 1 000 Höchstwert: 10 000.
`timeoutMs`	Die maximale Zeit (in Milliseku nden) zum Puffern eines Batches.	Standard: 1 000 Minimum: 25 Höchstwert: 30 000

Beachten Sie beim Einrichten der Pufferung die folgenden Punkte:

- Wenn einer der Eingabestreams geschlossen wird, leert Lambda die Protokolle. Dies kann beispielsweise passieren, wenn die Laufzeit abstürzt.
- Jeder Subscriber kann in seiner Abonnementanfrage eine andere Pufferungskonfiguration angeben.

- Rechnen Sie beim Bestimmen der Puffergröße für das Lesen der Daten mit eingehenden Nutzlasten in der Größenordnung von `2 * maxBytes + metadataBytes`, wobei `maxBytes` ein Bestandteil Ihrer Pufferungskonfiguration ist. Anhand der folgenden Metadaten können Sie `metadataBytes` abschätzen. Lambda hängt jedem Datensatz ähnliche Metadaten an:

```
{
   "time": "2022-08-20T12:31:32.123Z",
   "type": "function",
   "record": "Hello World"
}
```

- Wenn der Subscriber eingehende Telemetriedaten nicht schnell genug verarbeiten kann oder wenn Ihr Funktionscode ein sehr hohes Protokollvolumen erzeugt, löscht Lambda möglicherweise Datensätze, um die Speichernutzung zu begrenzen. In diesem Fall sendet Lambda ein `platform.logsDropped`-Ereignis.

Senden einer Abonnementanfrage an die Telemetrie-API

Lambda-Erweiterungen können den Empfang von Telemetriedaten abonnieren, indem sie eine Abonnementanfrage an die Telemetrie-API senden. Die Abonnementanfrage sollte Informationen über die Arten von Ereignissen enthalten, die die Erweiterung abonnieren soll. Darüber hinaus kann die Anfrage Informationen zum Lieferziel und eine Pufferkonfiguration enthalten.

Bevor Sie eine Abonnementanfrage senden, benötigen Sie eine Erweiterungs-ID (`Lambda-Extension-Identifier`). Wenn Sie Ihre Erweiterung in der Erweiterungs-API registrieren, erhalten Sie eine Erweiterungs-ID aus der API-Antwort.

Das Abonnement erfolgt während der Initialisierungsphase der Erweiterung. Das folgende Beispiel zeigt eine HTTP-Anfrage zum Abonnieren aller drei Telemetriestreams: Plattformtelemetrie, Funktionsprotokolle und Erweiterungsprotokolle.

```
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"
    }
}
```

Ist die Anfrage erfolgreich, erhält der Subscriber eine HTTP 200-Erfolgsantwort.

```
HTTP/1.1 200 OK
"OK"
```

Eingehende Telemetrie-API-Nachrichten

Nach dem Abonnieren mit der Telemetry API beginnt eine Erweiterung automatisch, Telemetriedaten von Lambda über POST-Anfragen zu empfangen. Jeder POST-Anforderungstext enthält ein Array von `Event` Objekten. Jedes `Event` hat das folgende Schema:

```
{
   time: String,
   type: String,
   record: Object
}
```

- Die `time`-Eigenschaft definiert, wann die Lambda-Plattform das Ereignis generiert hat. Das ist nicht identisch mit dem Zeitpunkt, zu dem das Ereignis tatsächlich stattgefunden hat. Der Zeichenfolgenwert von `time` ist ein Zeitstempel im ISO 8601-Format.
- Die `type`-Eigenschaft definiert den Ereignistyp. Die folgende Tabelle beschreibt alle möglichen Werte.
- Die `record`-Eigenschaft definiert ein JSON-Objekt, das die Telemetriedaten enthält. Das Schema dieses JSON-Objekts hängt von dem `type` ab.

> **Bestellung von Ereignissen mit gleichzeitigen Aufrufen**
>
> Bei Lambda Managed Instances können mehrere Funktionsaufrufen gleichzeitig in derselben Ausführungsumgebung ausgeführt werden. In diesem Fall ist die Reihenfolge

der `platform.report` Ereignisse `platform.start` und zwischen verschiedenen gleichzeitigen Aufrufen nicht garantiert. Erweiterungen müssen Ereignisse aus mehreren Aufrufen verarbeiten, die parallel ausgeführt werden, und sollten keine sequentielle Reihenfolge annehmen.
Um Ereignisse bestimmten Aufrufen korrekt zuzuordnen, sollten Erweiterungen das `requestId` Feld verwenden, das in diesen Plattformereignissen vorhanden ist. Jeder Aufruf hat eine eindeutige Anforderungs-ID, die für alle Ereignisse dieses Aufrufs einheitlich bleibt, sodass Erweiterungen Ereignisse auch dann korrekt korrelieren können, wenn sie nicht in der richtigen Reihenfolge ankommen.

Die folgende Tabelle fasst alle `Event`-Objekttypen zusammen und enthält Links zur `Event`-Telemetrie-API-Schemareferenz für jeden Ereignistyp.

Kategorie	Ereignistyp	Description	Schema der Ereignisaufzeichnung
Plattform-Ereignis	`platform.initStart`	Die Funktionsinitialisierung wurde gestartet.	the section called “`platform.initStart`”-Schema
Plattform-Ereignis	`platform.initRuntimeDone`	Die Funktionsinitialisierung ist abgeschlossen.	the section called “`platform.initRuntimeDone`”-Schema
Plattform-Ereignis	`platform.initReport`	Ein Bericht über die Funktionsinitialisierung.	the section called “`platform.initReport`”-Schema
Plattform-Ereignis	`platform.start`	Der Funktionsaufruf wurde gestartet.	the section called “`platform.start`”-Schema
Plattform-Ereignis	`platform.runtimeDone`	Die Laufzeit hat die Verarbeitung eines	the section called “`platform.`

Kategorie	Ereignistyp	Description	Schema der Ereignisaufzeichnung
		Ereignisses erfolgreich oder mit einem Fehler abgeschlossen.	`runtimeDone` -Schema
Plattform-Ereignis	`platform.report`	Ein Bericht über den Funktionsaufruf.	the section called “`platform.report`”-Schema
Plattform-Ereignis	`platform.restoreStart`	Die Laufzeitwiederherstellung wurde gestartet.	the section called “`platform.restoreStart`”-Schema
Plattform-Ereignis	`platform.restoreRuntimeDone`	Die Laufzeitwiederherstellung wurde abgeschlossen.	the section called “`platform.restoreRuntimeDone`”-Schema
Plattform-Ereignis	`platform.restoreReport`	Bericht über die Laufzeitwiederherstellung.	the section called “`platform.restoreReport`”-Schema
Plattform-Ereignis	`platform.telemetrySubscription`	Die Erweiterung hat die Telemetrie-API abonniert.	the section called “`platform.telemetrySubscription`”-Schema
Plattform-Ereignis	`platform.logsDropped`	Lambda hat die Protokolleinträge gelöscht.	the section called “`platform.logsDropped`”-Schema

Kategorie	Ereignistyp	Description	Schema der Ereignisaufzeichnung
Funktionsprotokolle	`function`	Eine Protokollzeile aus dem Funktionscode.	the section called “`function`”-Schema
Erweiterungsprotokolle	`extension`	Eine Protokollzeile aus dem Erweiterungscode.	the section called “`extension `”-Schema

Referenz zur Lambda-Telemetrie-API

Verwenden Sie den Lambda-Telemetrie-API-Endpunkt, um Erweiterungen für Telemetrie-Streams zu abonnieren. Sie können den Telemtrie-API-Endpunkts aus der `AWS_LAMBDA_RUNTIME_API`-Umgebungsvariable abrufen. Um eine API-Anfrage zu senden, fügen Sie die API-Version (`2022-07-01/`) und `telemetry/` an. Beispiel:

```
http://${AWS_LAMBDA_RUNTIME_API}/2022-07-01/telemetry/
```

Die Definition der OpenAPI-Spezifikation (OAS) der Abonnementantwortversion `2025-01-29` finden Sie unter:

- HTTP — telemetry-api-http-schema.zip
- TCP — .zip telemetry-api-tcp-schema

API-Operationen

- Abonnieren

Abonnieren

Um einen Telemetrie-Stream zu abonnieren, kann eine Lambda-Erweiterung eine Abonnement-API-Anfrage senden.

- Pfad – `/telemetry`
- Methode – PUT
- Header
  - `Content-Type`: `application/json`
- Anfrage von Textparametern
  - schemaVersion
    - Erforderlich: Ja
    - Typ: Zeichenfolge
    - Gültige Werte: `"2025-01-29"`, `"2022-12-13"` oder `"2022-07-01"`
    - Hinweis: Für Lambda Managed Instances ist Folgendes erforderlich`"2025-01-29"`. Diese Version ist abwärtskompatibel mit Lambda-Funktionen (Standard).

- Ziel – Die Konfigurationseinstellungen, die das Ziel des Telemetrieereignisses und das Protokoll für die Ereignisübermittlung definieren.
  - Erforderlich: Ja
  - Typ: Objekt

```
{
    "protocol": "HTTP",
    "URI": "http://sandbox.localdomain:8080"
}
```

  - Protokoll – Das Protokoll, das Lambda zum Senden von Telemetriedaten verwendet.
    - Erforderlich: Ja
    - Typ: Zeichenfolge
    - Gültige Werte: `"HTTP"|"TCP"`
  - URI – Der URI, an den Telemetriedaten gesendet werden sollen.
    - Erforderlich: Ja
    - Typ: Zeichenfolge
  - Weitere Informationen finden Sie unter the section called “Festlegen eines Zielprotokolls”.
- Typen – Die Telemetrietypen, die die Erweiterung abonnieren soll.
  - Erforderlich: Ja
  - Typ: Zeichenfolgen-Array
  - Zulässige Werte: `"platform"|"function"|"extension"`
- Pufferung – Die Konfigurationseinstellungen für die Ereignispufferung.
  - Erforderlich: Nein
  - Typ: Objekt

```
{
   "buffering": {
        "maxItems": 1000,
        "maxBytes": 256*1024,
        "timeoutMs": 100
   }
}
```

  - maxItems – Die maximale Anzahl der Ereignisse im Speicher, die gepuffert werden sollen.

    - Erforderlich: Nein
    - Typ: Ganzzahl
    - Standard: 1 000
    - Mindestwert 1 000
    - Höchstwert: 10 000.
- maxBytes – Das maximale Telemetrievolumen (in Bytes), das im Speicher gepuffert werden soll.
    - Erforderlich: Nein
    - Typ: Ganzzahl
    - Standard: 262 144
    - Mindestwert: 262 144
    - Höchstwert: 1 048 576
- timeoutMs – Die maximale Zeit (in Millisekunden) zum Puffern eines Batches.
    - Erforderlich: Nein
    - Typ: Ganzzahl
    - Standard: 1 000
    - Minimum: 25
    - Höchstwert: 30 000
- Weitere Informationen finden Sie unter the section called “Konfiguration der Speichernutzung und Pufferung”.

Beispiel für eine Abonnement-API-Anfrage

```
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"
    }
}
```

Wenn die Abonnement-Anfrage erfolgreich war, erhält die Erweiterung eine Erfolgsbestätigung (HTTP 200):

```
HTTP/1.1 200 OK
"OK"
```

Wenn die Abonnement-Anfrage fehlschlägt, erhält die Erweiterung eine Fehlerantwort. Beispiel:

```
HTTP/1.1 400 OK
{
    "errorType": "ValidationError",
    "errorMessage": "URI port is not provided; types should not be empty"
}
```

Hier sind einige zusätzliche Antwortcodes, die die Erweiterung empfangen kann:

- 200 – Anfrage erfolgreich abgeschlossen
- 202 – Anfrage wurde akzeptiert. Antwort auf die Abonnement-Anfrage in einer lokalen Testumgebung
- 400 – Ungültige Anfrage
- 500 – Servicefehler

Referenz zum **`Event`**-Schema der Lambda-Telemetrie-API

Verwenden Sie den Lambda-Telemetrie-API-Endpunkt, um Erweiterungen für Telemetrie-Streams zu abonnieren. Sie können den Telemtrie-API-Endpunkts aus der `AWS_LAMBDA_RUNTIME_API`-Umgebungsvariable abrufen. Um eine API-Anfrage zu senden, fügen Sie die API-Version (`2022-07-01/`) und `telemetry/` an. Beispiel:

```
http://${AWS_LAMBDA_RUNTIME_API}/2022-07-01/telemetry/
```

Die Definition der OpenAPI-Spezifikation (OAS) der Abonnementantwortversion `2025-01-29` finden Sie unter:

- HTTP — telemetry-api-http-schema.zip
- TCP — .zip telemetry-api-tcp-schema

Die folgende Tabelle enthält eine Zusammenfassung aller `Event`-Objekttypen, die die Telemetrie-API unterstützt.

Kategorie	Ereignistyp	Description	Schema der Ereignisaufzeichnung
Plattform-Ereignis	`platform.initStart`	Die Funktionsinitialisierung wurde gestartet.	the section called “`platform.initStart`”-Schema
Plattform-Ereignis	`platform.initRuntimeDone`	Die Funktionsinitialisierung ist abgeschlossen.	the section called “`platform.initRuntimeDone`”-Schema
Plattform-Ereignis	`platform.initReport`	Ein Bericht über die Funktionsinitialisierung.	the section called “`platform.initReport`”-Schema

Kategorie	Ereignistyp	Description	Schema der Ereignisaufzeichnung
Plattform-Ereignis	`platform.start`	Der Funktionsaufruf wurde gestartet.	the section called “`platform.start`”-Schema
Plattform-Ereignis	`platform.runtimeDone`	Die Laufzeit hat die Verarbeitung eines Ereignisses erfolgreich oder mit einem Fehler abgeschlossen.	the section called “`platform.runtimeDone`”-Schema
Plattform-Ereignis	`platform.report`	Ein Bericht über den Funktionsaufruf.	the section called “`platform.report`”-Schema
Plattform-Ereignis	`platform.restoreStart`	Die Laufzeitwiederherstellung wurde gestartet.	the section called “`platform.restoreStart`”-Schema
Plattform-Ereignis	`platform.restoreRuntimeDone`	Die Laufzeitwiederherstellung wurde abgeschlossen.	the section called “`platform.restoreRuntimeDone`”-Schema
Plattform-Ereignis	`platform.restoreReport`	Bericht über die Laufzeitwiederherstellung.	the section called “`platform.restoreReport`”-Schema

Kategorie	Ereignistyp	Description	Schema der Ereignisaufzeichnung
Plattform-Ereignis	`platform.telemetrySubscription`	Die Erweiterung hat die Telemetrie-API abonniert.	the section called “`platform.telemetrySubscription`”-Schema
Plattform-Ereignis	`platform.logsDropped`	Lambda hat die Protokolleinträge gelöscht.	the section called “`platform.logsDropped`”-Schema
Funktionsprotokolle	`function`	Eine Protokollzeile aus dem Funktionscode.	the section called “`function`”-Schema
Erweiterungsprotokolle	`extension`	Eine Protokollzeile aus dem Erweiterungscode.	the section called “`extension`”-Schema

Inhalt

- Telemetrie-API Event-Objekttypen
  - 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
- Freigegebene Objekttypen
  - InitPhase
  - InitReportMetrics
  - InitType
  - ReportMetrics
  - RestoreReportMetrics
  - RuntimeDoneMetrics
  - Span
  - Status
  - TraceContext
  - TracingType

Telemetrie-API **`Event`**-Objekttypen

Dieser Abschnitt beschreibt die Arten der `Event`-Objekte, die die Lambda-Telemetrie-API unterstützt. In den Ereignisbeschreibungen weist ein Fragezeichen (?) darauf hin, dass das Attribut möglicherweise nicht in diesem Objekt vorhanden ist.

`platform.initStart`

Ein `platform.initStart`-Ereignis zeigt an, dass die Initialisierungsphase der Funktion begonnen hat. Ein `platform.initStart` `Event`-Objekt hat die folgende Form:

```
Event: Object
- time: String
- type: String = platform.initStart
- record: PlatformInitStart
```

Dieses `PlatformInitStart`-Objekt hat die folgenden Attribute:

- functionName – `String`

- functionVersion – `String`
- initializationType – the section called “InitType”-Objekt
- instanceId? – `String`
- instanceMaxMemory? – `Integer`
- phase – the section called “InitPhase”-Objekt
- runtimeVersion? – `String`
- runtimeVersionArn? – `String`

Es folgt ein Beispiel für `Event` des Typs `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`

Ein `platform.initRuntimeDone`-Ereignis zeigt an, dass die Initialisierungsphase der Funktion abgeschlossen wurde. Ein `platform.initRuntimeDone` `Event`-Objekt hat die folgende Form:

```
Event: Object
- time: String
- type: String = platform.initRuntimeDone
- record: PlatformInitRuntimeDone
```

Dieses `PlatformInitRuntimeDone`-Objekt hat die folgenden Attribute:

- initializationType – the section called “InitType”-Objekt
- phase – the section called “InitPhase”-Objekt

- status – the section called “Status”-Objekt
- spans? – Liste von the section called “Span”-Objekten

Es folgt ein Beispiel für `Event` des Typs `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`

Ein `platform.initReport`-Ereignis enthält einen Gesamtbericht über die Initialisierungsphase der Funktion. Ein `platform.initReport` `Event`-Objekt hat die folgende Form:

```
Event: Object
- time: String
- type: String = platform.initReport
- record: PlatformInitReport
```

Dieses `PlatformInitReport`-Objekt hat die folgenden Attribute:

- errorType? – Zeichenfolge
- initializationType – the section called “InitType”-Objekt
- phase – the section called “InitPhase”-Objekt
- metrics – the section called “InitReportMetrics”-Objekt
- spans? – Liste von the section called “Span”-Objekten
- status – the section called “Status”-Objekt

Es folgt ein Beispiel für `Event` des Typs `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`

Ein `platform.start`-Ereignis zeigt an, dass die Funktionsaufrufphase begonnen hat. Ein `platform.start` `Event` Objekt hat die folgende Form:

```
Event: Object
- time: String
- type: String = platform.start
- record: PlatformStart
```

Dieses `PlatformStart`-Objekt hat die folgenden Attribute:

- requestId – `String`
- version? – `String`
- tracing? – the section called “TraceContext”

Es folgt ein Beispiel für `Event` des Typs `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`

Ein `platform.runtimeDone`-Ereignis zeigt an, dass die Funktionsaufrufphase abgeschlossen ist. Ein `platform.runtimeDone` `Event`-Objekt hat die folgende Form:

> **Von Lambda verwaltete Instanzen**
>
> Das `platform.runtimeDone` Ereignis wird für Lambda Managed Instances nicht unterstützt. Erweiterungen, die auf verwalteten Instanzen ausgeführt werden, erhalten dieses Ereignis nicht, da Erweiterungen das `INVOKE` Ereignis auf verwalteten Instanzen nicht abonnieren können. Aufgrund des Modells der gleichzeitigen Ausführung, bei dem mehrere Aufrufe gleichzeitig verarbeitet werden können, können Erweiterungen keine Verarbeitung nach dem Aufrufen für einzelne Aufrufe durchführen, wie dies üblicherweise bei Lambda-Funktionen (Standard) der Fall ist.
> Bei verwalteten Instanzen sind die Bereiche `responseLatency` und, die normalerweise `responseDuration` enthalten sind, stattdessen im Ereignis verfügbar. `platform.runtimeDone` `platform.report` Details dazu finden Sie unter the section called “`platform.report`”.

```
Event: Object
- time: String
- type: String = platform.runtimeDone
- record: PlatformRuntimeDone
```

Dieses `PlatformRuntimeDone`-Objekt hat die folgenden Attribute:

- errorType? – `String`
- metrics? – the section called “RuntimeDoneMetrics” Objekt
- requestId – `String`
- status – the section called “Status”-Objekt
- spans? – Liste von the section called “Span”-Objekten
- tracing? – the section called “TraceContext”-Objekt

Es folgt ein Beispiel für `Event` des Typs `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`

Ein `platform.report`-Ereignis enthält einen Gesamtbericht über die Phase des Funktionsaufrufs. Ein `platform.report` `Event`-Objekt hat die folgende Form:

Von Lambda verwaltete Instanzen

Das `platform.report` Ereignis für Lambda Managed Instances hat im Vergleich zu Lambda-Funktionen (Standard) andere Metriken und Spannweiten. Für verwaltete Instanzen:

- Spans: Enthält `responseLatency` und `responseDuration` statt. `extensionOverhead` Die `extensionOverhead` Spanne ist nicht verfügbar, da Erweiterungen das `INVOKE` Ereignis auf verwalteten Instanzen aufgrund des parallelen Ausführungsmodells nicht abonnieren können.
- Metriken: Beinhaltet `durationMs` nur. Die folgenden Metriken sind nicht enthalten: `billedDurationMs`,`initDurationMs`,`maxMemoryUsedMB`, und`memorySizeMB`. Diese Metriken pro Aufruf sind in der Umgebung für die gleichzeitige Ausführung nicht anwendbar. Verwenden Überwachung von Lambda-verwalteten Instanzen Sie Lambda Insights für Kennzahlen zur Ressourcennutzung.

```
Event: Object
- time: String
- type: String = platform.report
- record: PlatformReport
```

Dieses `PlatformReport`-Objekt hat die folgenden Attribute:

- metrics – the section called “ReportMetrics”-Objekt
- requestId – `String`
- spans? – Liste von the section called “Span”-Objekten
- Status – the section called “Status”-Objekt
- Nachverfolgung? – the section called “TraceContext”-Objekt

Es folgt ein Beispiel für `Event` des Typs `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`

Ein `platform.restoreStart`-Ereignis zeigt an, dass ein Ereignis zur Wiederherstellung der Funktionsumgebung gestartet wurde. Bei einem Wiederherstellungsereignis einer Umgebung erstellt Lambda die Umgebung aus einem im Cache gespeicherten Snapshot, anstatt diese von Grund auf neu zu initialisieren. Weitere Informationen finden Sie unter Lambda SnapStart. Ein `platform.restoreStart` `Event` Objekt hat die folgende Form:

```
Event: Object
- time: String
- type: String = platform.restoreStart
- record: PlatformRestoreStart
```

Dieses `PlatformRestoreStart`-Objekt hat die folgenden Attribute:

- functionName – `String`
- functionVersion – `String`
- instanceId? – `String`
- instanceMaxMemory? – `String`
- runtimeVersion? – `String`
- runtimeVersionArn? – `String`

Es folgt ein Beispiel für `Event` des Typs `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`

Ein `platform.restoreRuntimeDone`-Ereignis weist darauf hin, dass ein Wiederherstellungsereignis der Funktionsumgebung abgeschlossen wurde. Bei einem Wiederherstellungsereignis einer Umgebung erstellt Lambda die Umgebung aus einem im Cache gespeicherten Snapshot, anstatt diese von Grund auf neu zu initialisieren. Weitere Informationen finden Sie unter Lambda SnapStart. Ein `platform.restoreRuntimeDone` `Event` Objekt hat die folgende Form:

```
Event: Object
- time: String
- type: String = platform.restoreRuntimeDone
- record: PlatformRestoreRuntimeDone
```

Dieses `PlatformRestoreRuntimeDone`-Objekt hat die folgenden Attribute:

- errorType? – `String`
- spans? – Liste von `the section called “Span”`-Objekten
- status – `the section called “Status”`-Objekt

Es folgt ein Beispiel für `Event` des Typs `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`

Ein `platform.restoreReport`-Ereignis enthält einen Gesamtbericht über ein Ereignis zur Funktionswiederherstellung. Ein `platform.restoreReport` `Event`-Objekt hat die folgende Form:

```
Event: Object
- time: String
- type: String = platform.restoreReport
- record: PlatformRestoreReport
```

Dieses `PlatformRestoreReport`-Objekt hat die folgenden Attribute:

- errorType? – Zeichenfolge
- metrics? – the section called “RestoreReportMetrics” Objekt
- spans? – Liste von the section called “Span”-Objekten
- status – the section called “Status”-Objekt

Es folgt ein Beispiel für `Event` des Typs `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

Ein extension-Ereignis enthält Protokolle aus dem Erweiterungscode. Ein extension Event-Objekt hat die folgende Form:

```
Event: Object
- time: String
- type: String = extension
- record: {}
```

Dieses PlatformExtension-Objekt hat die folgenden Attribute:

- events – Liste von String
- name – String
- status – String

Es folgt ein Beispiel für Event des Typs platform.extension:

```
{
    "time": "2022-10-12T00:02:15.000Z",
    "type": "platform.extension",
    "record": {
        "events": [ "INVOKE", "SHUTDOWN" ],
        "name": "my-telemetry-extension",
        "state": "Ready"
    }
}
```

platform.telemetrySubscription

Ein platform.telemetrySubscription-Ereignis enthält Informationen zu einem Erweiterungsabonnement. Ein platform.telemetrySubscription Event-Objekt hat die folgende Form:

```
Event: Object
- time: String
- type: String = platform.telemetrySubscription
- record: PlatformTelemetrySubscription
```

Dieses PlatformTelemetrySubscription-Objekt hat die folgenden Attribute:

- name – `String`
- status – `String`
- types – Liste von `String`

Es folgt ein Beispiel für `Event` des Typs `platform.telemetrySubscription`:

```
{
    "time": "2022-10-12T00:02:35.000Z",
    "type": "platform.telemetrySubscription",
    "record": {
        "name": "my-telemetry-extension",
        "state": "Subscribed",
        "types": [ "platform", "function" ]
    }
}
```

`platform.logsDropped`

Ein `platform.logsDropped`-Ereignis enthält Informationen über gelöschte Ereignisse. Lambda gibt das `platform.logsDropped`-Ereignis aus, wenn eine Funktion Protokolle schneller ausgibt, als Lambda sie verarbeiten kann. Wenn Lambda Protokolle nicht mit der Geschwindigkeit an CloudWatch oder an die Erweiterung senden kann, die die Telemetrie-API abonniert hat, werden Protokolle gelöscht, um zu verhindern, dass die Ausführung der Funktion verlangsamt wird. Ein `platform.logsDropped` `Event`-Objekt hat die folgende Form:

```
Event: Object
- time: String
- type: String = platform.logsDropped
- record: PlatformLogsDropped
```

Dieses `PlatformLogsDropped`-Objekt hat die folgenden Attribute:

- droppedBytes – `Integer`
- droppedRecords – `Integer`
- reason – `String`

Es folgt ein Beispiel für `Event` des Typs `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`

Ein `function`-Ereignis enthält Protokolle aus dem Funktionscode. Ein `function` `Event`-Objekt hat die folgende Form:

```
Event: Object
- time: String
- type: String = function
- record: {}
```

Das Format des Felds `record` hängt davon ab, ob die Protokolle Ihrer Funktion im Klartext- oder JSON-Format formatiert sind. Weitere Informationen zu den Konfigurationsoptionen für das Protokollformat finden Sie unter the section called “Protokollformate”.

Im Folgenden finden Sie ein Beispiel für `Event` vom Typ `function`, bei dem das Protokollformat Klartext ist:

```
{
    "time": "2022-10-12T00:03:50.000Z",
    "type": "function",
    "record": "[INFO] Hello world, I am a function!"
}
```

Im Folgenden finden Sie ein Beispiel für `Event` vom Typ `function`, bei dem das Protokollformat JSON ist:

```
{
    "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**
>
> Wenn die von Ihnen verwendete Schemaversion älter als die `2022-12-13`-Version ist, wird `"record"` immer als Zeichenfolge gerendert, auch wenn das Protokollierungsformat Ihrer Funktion als JSON konfiguriert ist. Für Lambda Managed Instances müssen Sie die Schemaversion `2025-01-29` verwenden.

`extension`

Ein `extension`-Ereignis enthält Protokolle aus dem Erweiterungscode. Ein `extension` `Event`-Objekt hat die folgende Form:

```
Event: Object
- time: String
- type: String = extension
- record: {}
```

Das Format des Felds `record` hängt davon ab, ob die Protokolle Ihrer Funktion im Klartext- oder JSON-Format formatiert sind. Weitere Informationen zu den Konfigurationsoptionen für das Protokollformat finden Sie unter the section called “Protokollformate”.

Im Folgenden finden Sie ein Beispiel für `Event` vom Typ `extension`, bei dem das Protokollformat Klartext ist:

```
{
    "time": "2022-10-12T00:03:50.000Z",
    "type": "extension",
    "record": "[INFO] Hello world, I am an extension!"
}
```

Im Folgenden finden Sie ein Beispiel für `Event` vom Typ `extension`, bei dem das Protokollformat JSON ist:

```
{
    "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**
>
> Wenn die von Ihnen verwendete Schemaversion älter als die `2022-12-13`-Version ist, wird `"record"` immer als Zeichenfolge gerendert, auch wenn das Protokollierungsformat Ihrer Funktion als JSON konfiguriert ist. Für Lambda Managed Instances müssen Sie die Schemaversion `2025-01-29` verwenden.

Freigegebene Objekttypen

Dieser Abschnitt beschreibt die Arten von freigegebenen Objekten, die die Lambda-Telemetrie-API unterstützt.

`InitPhase`

Eine Zeichenfolgenaufzählung, die die Phase beschreibt, in der der Initialisierungsschritt auftritt. In den meisten Fällen führt Lambda den Funktionsinitialisierungscode während der `init`-Phase aus. In einigen Fehlerfällen kann Lambda jedoch den Funktionsinitialisierungscode während der `invoke`-Phase erneut ausführen. (Dies wird als suppressed init bezeichnet.)

- Typ: – `String`
- Gültige Werte – `init|invoke|snap-start`

`InitReportMetrics`

Ein Objekt, das Metriken zu einer Initialisierungsphase enthält.

- Typ: – `Object`

Ein `InitReportMetrics`-Objekt hat die folgende Form:

```
InitReportMetrics: Object
- durationMs: Double
```

Es folgt ein Beispiel für ein `InitReportMetrics`-Objekt:

```
{
    "durationMs": 247.88
}
```

`InitType`

Eine Zeichenfolgenaufzählung, die beschreibt, wie Lambda die Umgebung initialisiert hat.

- Typ: – `String`
- Gültige Werte – `on-demand|provisioned-concurrency`

`ReportMetrics`

Ein Objekt, das Details zu einer abgeschlossenen Phase enthält.

- Typ: – `Object`

Ein `ReportMetrics`-Objekt hat die folgende Form:

```
ReportMetrics: Object
- billedDurationMs: Integer
- durationMs: Double
- initDurationMs?: Double
- maxMemoryUsedMB: Integer
- memorySizeMB: Integer
- restoreDurationMs?: Double
```

Es folgt ein Beispiel für ein `ReportMetrics`-Objekt:

```
{
    "billedDurationMs": 694,
    "durationMs": 693.92,
    "initDurationMs": 397.68,
```

```
    "maxMemoryUsedMB": 84,
    "memorySizeMB": 128
}
```

`RestoreReportMetrics`

Ein Objekt, das Metriken über eine abgeschlossene Wiederherstellungsphase enthält.

- Typ: – `Object`

Ein `RestoreReportMetrics`-Objekt hat die folgende Form:

```
RestoreReportMetrics: Object
- durationMs: Double
```

Es folgt ein Beispiel für ein `RestoreReportMetrics`-Objekt:

```
{
    "durationMs": 15.19
}
```

`RuntimeDoneMetrics`

Ein Objekt, das Metriken über eine Aufrufphase enthält.

- Typ: – `Object`

Ein `RuntimeDoneMetrics`-Objekt hat die folgende Form:

```
RuntimeDoneMetrics: Object
- durationMs: Double
- producedBytes?: Integer
```

Es folgt ein Beispiel für ein `RuntimeDoneMetrics`-Objekt:

```
{
    "durationMs": 200.0,
    "producedBytes": 15
}
```

`Span`

Ein Objekt, das Details einem Span enthält. Ein Span stellt eine Arbeitseinheit oder einen Vorgang in einer Ablaufverfolgung dar. Weitere Informationen zu Spans finden Sie unter Span auf der Tracing-API-Seite der OpenTelemetry Docs-Website.

Lambda unterstützt die folgenden zwei Bereiche für das `platform.RuntimeDone`-Ereignis:

- Der `responseLatency`-Span beschreibt, wie lange es gedauert hat, bis Ihre Lambda-Funktion mit dem Senden der Antwort begonnen hat.
- Der `responseDuration`-Spanbeschreibt, wie lange es gedauert hat, bis Ihre Lambda-Funktion das Senden der Antwort abgeschlossen hat.
- Die `runtimeOverhead`-Spanne beschreibt, wie lange es gedauert hat, bis die Lambda-Laufzeit signalisiert hat, dass sie für die Verarbeitung des nächsten Funktionsaufrufs bereit ist. Dies ist die Zeit, die die Laufzeit benötigt hat, um die nächste Aufruf-API aufzurufen, um das nächste Ereignis nach der Rückgabe Ihrer Funktionsantwort zu erhalten.

Es folgt ein Beispiel für ein Objekt des `responseLatency`-Spans:

```
{
        "name": "responseLatency",
        "start": "2022-08-02T12:01:23.521Z",
        "durationMs": 23.02
      }
```

`Status`

Ein Objekt, das den Status einer Initialisierungs- oder Aufrufphase beschreibt. Wenn der Status entweder `failure` oder `error` ist, dann enthält das `Status`-Objekt auch ein `errorType`-Feld, das den Fehler beschreibt.

- Typ: – `Object`
- Gültige Statuswerte – `success`|`failure`|`error`|`timeout`

`TraceContext`

Ein Objekt, das die Eigenschaften einer Ablaufverfolgung beschreibt.

- Typ: – `Object`

Ein `TraceContext`-Objekt hat die folgende Form:

```
TraceContext: Object
- spanId?: String
- type: TracingType enum
- value: String
```

Es folgt ein Beispiel für ein `TraceContext`-Objekt:

```
{
    "spanId": "073a49012f3c312e",
    "type": "X-Amzn-Trace-Id",
    "value":
 "Root=1-62e900b2-710d76f009d6e7785905449a;Parent=0efbd19962d95b05;Sampled=1"
}
```

`TracingType`

Eine Zeichenfolgenaufzählung, die den Ablaufverfolgungstyp in einem the section called “`TraceContext`”-Objekt beschreibt.

- Typ: – `String`
- Gültige Werte – `X-Amzn-Trace-Id`

Konvertieren von `Event`-Objekte der Lambda-Telemetrie-API in OpenTelemetry-Spans

Das AWS Lambda-Telemetrie-API-Schema ist semantisch kompatibel mit OpenTelemetry (OTel). Das bedeutet, dass Sie Ihre AWS Lambda-Telemetrie-API `Event`-Objekte zu OpenTelemetry (OTel)-Spans konvertieren können. Beim Konvertieren sollten Sie kein einzelnes `Event`-Objekt einem einzelnen OTel Span zuordnen. Stattdessen sollten Sie alle drei Ereignisse, die sich auf eine Lebenszyklusphase beziehen, in einem einzigen OTel-Span darstellen. Die `start`, `runtimeDone`, und `runtimeReport`-Ereignisse repräsentieren beispielsweise einen einzelnen Funktionsaufruf. Stellen Sie alle drei Ereignisse als einen einzigen OTel-Span dar.

Sie können Ihre Ereignisse mithilfe von Span-Ereignissen oder untergeordneten (verschachtelten) Spans konvertieren. Die Tabellen auf dieser Seite beschreiben die Zuordnungen zwischen Telemetrie-API-Schemaeigenschaften und OTel-Span-Eigenschaften für beide Ansätze. Weitere Informationen zu OTel Spans finden Sie unter Span auf der Seite Ablaufverfolgungs-API auf der OpenTelemetry-Docs-Website.

Abschnitte

- Zuordnung zu OTel Spans mit Span-Ereignissen
- Zuordnung zu OTel-Spans mit untergeordneten Spans

Zuordnung zu OTel Spans mit Span-Ereignissen

In den folgenden Tabellen stellt e das Ereignis dar, das von der Telemetrie-Quelle stammt.

Zuordnung der *Start-Ereignisse

OpenTelemetry	Schema der Lambda-Telemetrie-API
`Span.Name`	Ihre Erweiterung generiert diesen Wert basierend auf dem `type`-Feld.
`Span.StartTime`	Verwenden Sie `e.time`.
`Span.EndTime`	Nicht zutreffend, da das Ereignis noch nicht abgeschlossen ist.
`Span.Kind`	Setzen Sie auf `Server`.

OpenTelemetry	Schema der Lambda-Telemetrie-API
`Span.Status`	Setzen Sie auf `Unset`.
`Span.TraceId`	Analysieren Sie den AWS X-Ray-Header, der in `e.tracing.value` gefunden wurde, und verwenden Sie dann den `TraceId`-Wert.
`Span.ParentId`	Analysieren Sie den X-Ray-Header, der in `e.tracing.value` gefunden wurde, und verwenden Sie dann den `Parent`-Wert.
`Span.SpanId`	Verwenden Sie `e.tracing.spanId` , falls verfügbar. Generieren Sie andernfalls eine neue `SpanId`.
`Span.SpanContext.TraceState`	Nicht zutreffend für einen X-Ray-Ablaufverfol gungskontext.
`Span.SpanContext.TraceFlags`	Analysieren Sie den X-Ray-Header, der in `e.tracing.value` gefunden wurde, und verwenden Sie dann den `Sampled`-Wert.
`Span.Attributes`	Ihre Erweiterung kann hier beliebige benutzerd efinierte Werte hinzufügen.

Zuordnung der *RuntimeDone-Ereignisse

OpenTelemetry	Schema der Lambda-Telemetrie-API
`Span.Name`	Ihre Erweiterung generiert den Wert basierend auf dem `type`-Feld.
`Span.StartTime`	Verwenden Sie `e.time` aus dem übereinst immenden `*Start`-Ereignis. Als alternative Vorgehensweise verwenden Sie `e.time - e.metrics.durationMs` .

OpenTelemetry	Schema der Lambda-Telemetrie-API
`Span.EndTime`	Nicht zutreffend, da das Ereignis noch nicht abgeschlossen ist.
`Span.Kind`	Setzen Sie auf `Server`.
`Span.Status`	Wenn `e.status` nicht gleich `success` ist, dann Wert auf `Error` festlegen. Ansonsten auf festlegen `0k`.
`Span.Events[]`	Verwenden Sie `e.spans[]` .
`Span.Events[i].Name`	Verwenden Sie `e.spans[i].name` .
`Span.Events[i].Time`	Verwenden Sie `e.spans[i].start` .
`Span.TraceId`	Analysieren Sie den AWS X-Ray-Header, der in `e.tracing.value` gefunden wurde, und verwenden Sie dann den `TraceId`-Wert.
`Span.ParentId`	Analysieren Sie den X-Ray-Header, der in `e.tracing.value` gefunden wurde, und verwenden Sie dann den `Parent`-Wert.
`Span.SpanId`	Verwenden Sie dieselbe `SpanId` aus dem `*Start`-Ereignis. Falls nicht verfügbar, verwenden Sie `e.tracing.spanId` oder generieren Sie eine neue `SpanId`.
`Span.SpanContext.TraceState`	Nicht zutreffend für einen X-Ray-Ablaufverfol gungskontext.
`Span.SpanContext.TraceFlags`	Analysieren Sie den X-Ray-Header, der in `e.tracing.value` gefunden wurde, und verwenden Sie dann den `Sampled`-Wert.
`Span.Attributes`	Ihre Erweiterung kann hier beliebige benutzerd efinierte Werte hinzufügen.

Zuordnung der *Report-Ereignisse

OpenTelemetry	Schema der Lambda-Telemetrie-API
`Span.Name`	Ihre Erweiterung generiert den Wert basierend auf dem `type`-Feld.
`Span.StartTime`	Verwenden Sie `e.time` aus dem übereinstimmenden `*Start`-Ereignis. Als alternative Vorgehensweise verwenden Sie `e.time - e.metrics.durationMs` .
`Span.EndTime`	Verwenden Sie `e.time`.
`Span.Kind`	Setzen Sie auf `Server`.
`Span.Status`	Verwenden Sie den gleichen Wert wie das `*RuntimeDone` -Ereignis.
`Span.TraceId`	Analysieren Sie den AWS X-Ray-Header, der in `e.tracing.value` gefunden wurde, und verwenden Sie dann den `TraceId`-Wert.
`Span.ParentId`	Analysieren Sie den X-Ray-Header, der in `e.tracing.value` gefunden wurde, und verwenden Sie dann den `Parent`-Wert.
`Span.SpanId`	Verwenden Sie dieselbe `SpanId` aus dem `*Start`-Ereignis. Falls nicht verfügbar, verwenden Sie `e.tracing.spanId` oder generieren Sie eine neue `SpanId`.
`Span.SpanContext.TraceState`	Nicht zutreffend für einen X-Ray-Ablaufverfolgungskontext.
`Span.SpanContext.TraceFlags`	Analysieren Sie den X-Ray-Header, der in `e.tracing.value` gefunden wurde, und verwenden Sie dann den `Sampled`-Wert.

OpenTelemetry	Schema der Lambda-Telemetrie-API
`Span.Attributes`	Ihre Erweiterung kann hier beliebige benutzerdefinierte Werte hinzufügen.

Zuordnung zu OTel-Spans mit untergeordneten Spans

Die folgende Tabelle beschreibt, wie Lambda-Telemetrie-API-Ereignisse in OTel-Spans mit untergeordneten (verschachtelten) Spans für `*RuntimeDone`-Spans konvertiert werden. Informationen zu `*Start`- und `*Report`-Zuordnungen finden Sie in den Tabellen in the section called “Zuordnung zu OTel Spans mit Span-Ereignissen”, da diese für untergeordnete Spans identisch sind. In dieser Tabelle stellt `e` das Ereignis dar, das von der Telemetrie-Quelle stammt.

Zuordnung der *RuntimeDone-Ereignisse

OpenTelemetry	Schema der Lambda-Telemetrie-API
`Span.Name`	Ihre Erweiterung generiert den Wert basierend auf dem `type`-Feld.
`Span.StartTime`	Verwenden Sie `e.time` aus dem übereinstimmenden `*Start`-Ereignis. Als alternative Vorgehensweise verwenden Sie `e.time - e.metrics.durationMs` .
`Span.EndTime`	Nicht zutreffend, da das Ereignis noch nicht abgeschlossen ist.
`Span.Kind`	Setzen Sie auf `Server`.
`Span.Status`	Wenn `e.status` nicht gleich `success` ist, dann Wert auf `Error` festlegen. Ansonsten auf festlegen `0k`.

OpenTelemetry	Schema der Lambda-Telemetrie-API
`Span.TraceId`	Analysieren Sie den AWS X-Ray-Header, der in `e.tracing.value` gefunden wurde, und verwenden Sie dann den `TraceId`-Wert.
`Span.ParentId`	Analysieren Sie den X-Ray-Header, der in `e.tracing.value` gefunden wurde, und verwenden Sie dann den `Parent`-Wert.
`Span.SpanId`	Verwenden Sie dieselbe `SpanId` aus dem `*Start`-Ereignis. Falls nicht verfügbar, verwenden Sie `e.tracing.spanId` oder generieren Sie eine neue `SpanId`.
`Span.SpanContext.TraceState`	Nicht zutreffend für einen X-Ray-Ablaufverfol gungskontext.
`Span.SpanContext.TraceFlags`	Analysieren Sie den X-Ray-Header, der in `e.tracing.value` gefunden wurde, und verwenden Sie dann den `Sampled`-Wert.
`Span.Attributes`	Ihre Erweiterung kann hier beliebige benutzerd efinierte Werte hinzufügen.
`ChildSpan[i].Name`	Verwenden Sie `e.spans[i].name` .
`ChildSpan[i].StartTime`	Verwenden Sie `e.spans[i].start` .
`ChildSpan[i].EndTime`	Verwenden Sie `e.spans[i].start + e.spans[i].durations` .
`ChildSpan[i].Kind`	Wie bei übergeordnetem `Span.Kind` .
`ChildSpan[i].Status`	Wie bei übergeordnetem `Span.Status` .
`ChildSpan[i].TraceId`	Wie bei übergeordnetem `Span.TraceId` .
`ChildSpan[i].ParentId`	Verwenden Sie die übergeordnete `Span.Span Id` .

OpenTelemetry	Schema der Lambda-Telemetrie-API
`ChildSpan[i].SpanId`	Generieren Sie eine neue `SpanId`.
`ChildSpan[i].SpanContext.TraceState`	Nicht zutreffend für einen X-Ray-Ablaufverfolgungskontext.
`ChildSpan[i].SpanContext.TraceFlags`	Wie bei übergeordnetem `Span.SpanContext.TraceFlags` .

Verwendung der Lambda-Protokoll-API

Important

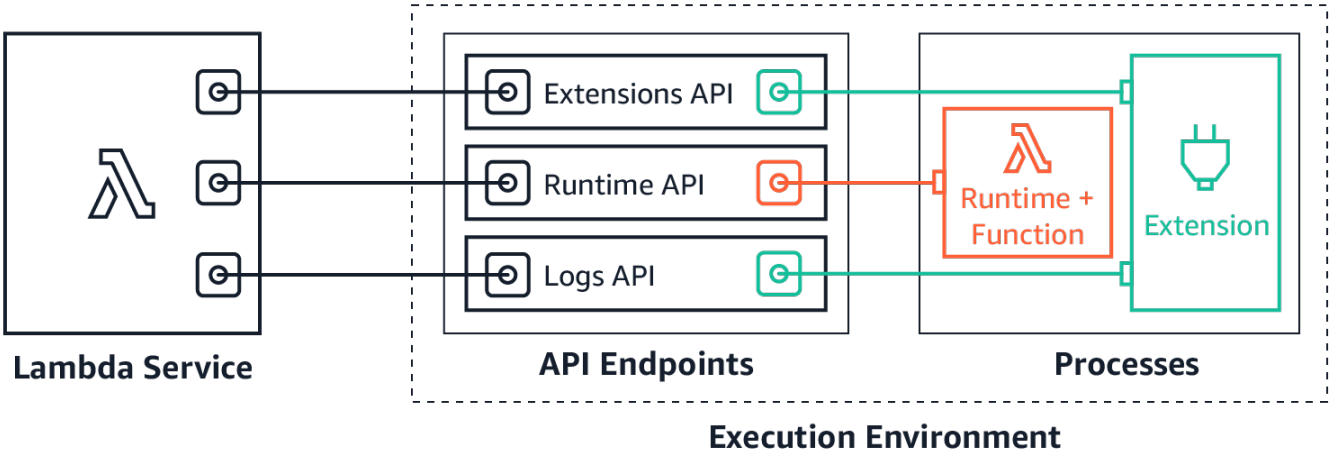
Die Lambda-Telemetrie-API ersetzt die Lambda-Protokoll-API. Die Protokoll-API bleibt zwar voll funktionsfähig, wir empfehlen jedoch, in Zukunft nur die Telemetrie-API zu verwenden. Sie können Ihre Erweiterung für einen Telemetrie-Stream entweder über die Telemetrie-API oder die Protokoll-API abonnieren. Nach dem Abonnieren mit einer dieser APIs wird bei jedem Versuch APIs, ein Abonnement über die andere API abzuschließen, ein Fehler zurückgegeben.

Lambda Managed Instances unterstützen die Logs-API nicht

Lambda Managed Instances unterstützen die Logs-API nicht. Wenn Sie Funktionen für verwaltete Instanzen verwenden, verwenden Sie stattdessen die Telemetrie-API. Die Telemetrie-API bietet erweiterte Funktionen für die Erfassung und Verarbeitung von Telemetriedaten aus Ihren Lambda-Funktionen.

Lambda erfasst automatisch Laufzeitprotokolle und streamt sie an Amazon CloudWatch. Dieser Protokolldatenstream enthält die Protokolle, die Ihr Funktionscode und Ihre Erweiterungen generieren, sowie die Protokolle, die Lambda im Rahmen des Funktionsaufrufens generiert.

Lambda-Erweiterungen können die Lambda-Laufzeitprotokoll-API verwenden, um Protokollstreams direkt aus der Lambda-Ausführungsumgebung zu abonnieren. Lambda streamt die Protokolle zur Erweiterung und die Erweiterung kann die Protokolle dann an ein beliebiges bevorzugtes Ziel verarbeiten, filtern und senden.



Die Logs API ermöglicht es Erweiterungen, drei verschiedene Protokollstreams zu abonnieren:

- Funktionsprotokolle, die die Lambda-Funktion generiert und in `stdout` oder `stderr` schreibt.
- Erweiterungsprotokolle, die der Erweiterungscode generiert.
- Lambda-Plattformprotokolle, die Ereignisse und Fehler im Zusammenhang mit Aufrufen und Erweiterungen aufzeichnen.

> **Note**
>
> Lambda sendet alle Protokolle an CloudWatch, auch wenn eine Erweiterung einen oder mehrere der Protokollstreams abonniert.

Themen

- Abonnieren des Empfangs von Protokollen
- Speicherauslastung
- Ziel-Protokolle
- Pufferung der Konfiguration
- Beispielabonnement
- Beispielcode für Logs API
- Logs API-Referenz
- Protokollmeldungen

Abonnieren des Empfangs von Protokollen

Eine Lambda-Erweiterung kann den Empfang von Protokollen abonnieren, indem sie eine Abonnementanfrage an die Protokoll-API sendet.

Um den Empfang von Protokollen zu abonnieren, benötigen Sie die Erweiterungskennung (`Lambda-Extension-Identifier`). Registrieren Sie zunächst die Erweiterung , um die Erweiterungskennung zu erhalten. Abonnieren Sie dann während der Initialisierung die Logs API. Lambda verarbeitet nach Abschluss der Initialisierungsphase keine Abonnementanfragen.

> **Note**
>
> Das Logs API-Abonnement ist idempotent. Doppelte Abonnementanfragen führen nicht zu doppelten Abonnements.

Speicherauslastung

Die Speichernutzung steigt linear, wenn die Anzahl der Abonnenten zunimmt. Abonnements verbrauchen Speicherressourcen, da jedes Abonnement einen neuen Speicherpuffer zum Speichern der Protokolle öffnet. Um die Speicherauslastung zu optimieren, können Sie die Pufferungskonfigurationanpassen. Die Pufferspeichernutzung wird zum Gesamtspeicherverbrauch in der Ausführungsumgebung gezählt.

Ziel-Protokolle

Sie können eines der folgenden Protokolle auswählen, um die Logs zu erhalten:

1. HTTP (empfohlen) – Lambda stellt Protokolle als Array von Datensätzen im JSON-Format an einen lokalen HTTP-Endpunkt (`http://sandbox.localdomain:${PORT}/${PATH}`) bereit. Der Parameter `$PATH` ist optional. Beachten Sie, dass nur HTTP unterstützt wird, nicht HTTPS. Sie können wählen, ob Sie Logs über PUT oder POST erhalten möchten.
2. TCP – Lambda stellt Protokolle an einen TCP-Port im Newline-delimited-JSON-(NDJSON)-Format bereit.

Wir empfehlen, HTTP statt TCP zu verwenden. Mit TCP kann die Lambda-Plattform nicht bestätigen, wenn es Protokolle an die Anwendungsebene übermittelt. Daher können Sie Protokolle verlieren, wenn Ihre Erweiterung abstürzt. Bei HTTP gilt diese Einschränkung nicht.

Wir empfehlen außerdem, den lokalen HTTP-Listener oder den TCP-Port einzurichten, bevor Sie sich für den Empfang von Protokollen anmelden. Beachten Sie bei der Einrichtung Folgendes:

- Lambda sendet Protokolle nur an Ziele, die sich innerhalb der Ausführungsumgebung befinden.
- Lambda wiederholt den Versuch, die Protokolle (mit Backoff) zu senden, wenn es keinen Listener gibt oder wenn die POST- oder PUT-Anfrage zu einem Fehler führt. Wenn der Protokoll-Subscriber abstürzt, erhält er nach dem Neustart der Ausführungsumgebung durch Lambda weiterhin Protokolle.
- Lambda reserviert Port 9001. Es gibt keine weiteren Einschränkungen oder Empfehlungen für Portnummern.

Pufferung der Konfiguration

Lambda kann Protokolle puffern und an den Subscriber liefern. Sie können dieses Verhalten in der Abonnementanfrage konfigurieren, indem Sie die folgenden optionalen Felder angeben. Beachten Sie, dass Lambda den Standardwert für jedes Feld verwenden wird, das Sie nicht angeben.

- timeoutMs – Die maximale Zeit (in Millisekunden) zum Puffern eines Batches. Standard: 1 000. Minimum: 25. Höchstwert: 30 000.
- maxBytes – Die maximale Größe (in Byte) der Protokolle, die im Speicher gepuffert werden sollen. Standard: 262 144. Mindestwert: 262 144. Höchstwert: 1 048 576.
- maxItems – Die maximale Anzahl der Ereignisse im Speicher, die gepuffert werden sollen. Standard: 10 000. Mindestwert 1 000. Höchstwert: 10 000.

Beachten Sie bei der Pufferung die folgenden Punkte:

- Lambda bereinigt die Protokolle, wenn einer der Eingabestreams geschlossen wird, z. B. wenn die Laufzeit abstürzt.
- Jeder Subscriber kann in seiner Abonnementanfrage eine andere Pufferungskonfiguration angeben.
- Berücksichtigen Sie die Puffergröße, die Sie zum Lesen der Daten benötigen. Gehen Sie davon aus, dass die Nutzlasten so groß wie `2*maxBytes+metadata` sein werden, wo `maxBytes` in der Abonnementanfrage konfiguriert ist. Lambda fügt beispielsweise jedem Datensatz die folgenden Metadaten-Bytes hinzu:

```
{
```

```
"time": "2020-08-20T12:31:32.123Z",
"type": "function",
"record": "Hello World"
}
```

- Wenn der Subscriber eingehende Protokolle nicht schnell genug verarbeiten kann, wird Lambda Protokolle möglicherweise löschen, um die Speicherauslastung begrenzt zu halten. Um die Anzahl der gelöschten Datensätze anzugeben, schickt Lambda ein `platform.logsDropped`-Protokoll. Weitere Informationen finden Sie unter the section called “Lambda: Nicht alle Protokolle meiner Funktion werden angezeigt”.

Beispielabonnement

Das folgende Beispiel zeigt eine Anfrage zum Abonnieren der Plattform- und Funktionsprotokolle.

```
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"
  }
}
```

Wenn die Anfrage erfolgreich ist, erhält der Subscriber eine Erfolgsantwort von HTTP 200.

```
HTTP/1.1 200 OK
"OK"
```

Beispielcode für Logs API

Einen Beispielcode, der zeigt, wie Logs an ein benutzerdefiniertes Ziel gesendet werden, finden Sie im AWS LambdaAWS Compute-Blog unter Verwenden von Erweiterungen zum Senden von Logs an benutzerdefinierte Ziele.

Codebeispiele für Python und Go, die zeigen, wie Sie eine grundlegende Lambda-Erweiterung entwickeln und die Logs-API abonnieren, finden Sie unter AWS Lambda Erweiterungen im AWS GitHub Samples-Repository. Für weitere Informationen zum Erstellen einer Lambda-Erweiterung siehe the section called “Erweiterungs-API”.

Logs API-Referenz

Sie können den Wert des API-Endpunkts aus der `AWS_LAMBDA_RUNTIME_API`-Umgebungsvariablen abrufen. Um eine API-Anfrage zu senden, verwenden Sie das Präfix `2020-08-15/` vor dem API-Pfad. Beispiel:

```
http://${AWS_LAMBDA_RUNTIME_API}/2020-08-15/logs
```

Die OpenAPI-Spezifikation für die Logs-API-Version 2020-08-15 ist hier verfügbar: .zip logs-api-request

Abonnieren

Um einen oder mehrere der in der Lambda-Ausführungsumgebung verfügbaren Protokollstreams zu abonnieren, senden Erweiterungen eine Abonnement-API-Anforderung.

Pfad – `/logs`

Methode – PUT

Body-Parameter

`destination` – Siehe the section called “Ziel-Protokolle”. Erforderlich: ja Typ: Strings.

`buffering` – Siehe the section called “Pufferung der Konfiguration”. Erforderlich: Nein Typ: Strings.

`types` – Ein Array der zu empfangenden Protokollentypen. Erforderlich: ja Typ: Zeichenfolge-Array Gültige Werte: „platform“, „function“, „extension“.

`schemaVersion` – Erforderlich: Nein. Standardwert: „2020-08-15“. Stellen Sie „2021-03-18“ ein, damit die Erweiterung `platform.runtimeDone`-Meldungen empfangen kann.

Antwortparameter

Die OpenAPI-Spezifikationen für die Abonnementantworten, Version 2020-08-15, stehen für HTTP- und TCP-Protokolle zur Verfügung:

- logs-api-http-responseHTTP: .zip
- TCP: .zip logs-api-tcp-response

Antwortcodes

- 200 – Anfrage erfolgreich abgeschlossen
- 202 – Anfrage wurde akzeptiert. Antwort auf eine Abonnementanfrage während lokaler Tests.
- 4XX – Ungültige Anfrage
- 500 – Servicefehler

Wenn die Anfrage erfolgreich ist, erhält der Subscriber eine Erfolgsantwort von HTTP 200.

```
HTTP/1.1 200 OK
"OK"
```

Schlägt die Anfrage fehl, erhält der Subscriber eine Fehlerantwort. Zum Beispiel:

```
HTTP/1.1 400 OK
{
    "errorType": "Logs.ValidationError",
    "errorMessage": URI port is not provided; types should not be empty"
}
```

Protokollmeldungen

Die Logs API ermöglicht es Erweiterungen, drei verschiedene Protokollstreams zu abonnieren:

- Funktion – Protokolle, die die Lambda-Funktion generiert und in `stdout` oder `stderr` schreibt.
- Erweiterung – Protokolle, die der Erweiterungscode generiert.

- Plattform – Protokolle, die von der Laufzeitplattform generiert werden und die Ereignisse und Fehler im Zusammenhang mit Aufrufen und Erweiterungen aufzeichnen.

Themen

- Funktionsprotokolle
- Erweiterungsprotokolle
- Plattformprotokolle

Funktionsprotokolle

Die Lambda-Funktion und interne Erweiterungen generieren Funktionsprotokolle und schreiben sie in `stdout` oder `stderr`.

Das folgende Beispiel zeigt das Format einer Funktionsprotokollmeldung. {„time“: „2020-08-20T12:31:32.123Z“, „type“: „function“, „record“: „ERROR encountered. Stack trace:\n\my-function (line 10)\n" }

Erweiterungsprotokolle

Erweiterungen können Erweiterungsprotokolle generieren. Das Protokollformat ist das gleiche wie bei einem Funktionsprotokoll.

Plattformprotokolle

Lambda generiert Protokollmeldungen für Plattformereignisse wie `platform.start`, `platform.end` und `platform.fault`.

Optional können Sie die Version 2021-03-18 des Logs API-Schemas abonnieren, welche die `platform.runtimeDone`-Protokollmeldungen enthält.

Beispiel für Plattformprotokollmeldungen

Das folgende Beispiel zeigt die Plattformstart- und Plattformendprotokolle. Diese Protokolle zeigen die Startzeit des Aufrufs und die Endzeit des Aufrufs für den Aufruf an, den die RequestID angibt.

```
{
    "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"}
}
```

Die Plattform. initRuntimeDoneDie Protokollnachricht zeigt den Status der `Runtime init` Unterphase an, die Teil der Init-Lebenszyklusphase ist. Wenn `Runtime init` erfolgreich ist, sendet die Laufzeit eine `/next`-Laufzeit-API-Anfrage (für die `on-demand`- und `provisioned-concurrency`-Initialisierungstypen) oder `restore/next` (für den `snap-start`-Initialisierungstyp). Das folgende Beispiel zeigt eine erfolgreiche Plattform. initRuntimeDoneProtokollnachricht für den `snap-start` Initialisierungstyp.

```
{
  "time":"2022-07-17T18:41:57.083Z",
  "type":"platform.initRuntimeDone",
  "record":{
      "initializationType":"snap-start",
      "status":"success"
  }
}
```

Die platform.initReport-Protokollnachricht zeigt, wie lange die `Init`-Phase gedauert hat und wie viele Millisekunden Ihnen während dieser Phase in Rechnung gestellt wurden. Wenn der Initialisierungstyp `provisioned-concurrency` lautet, sendet Lambda diese Nachricht während des Aufrufs. Wenn der Initialisierungstyp `snap-start` lautet, sendet Lambda diese Nachricht nach der Wiederherstellung des Snapshots. Das folgende Beispiel zeigt eine platform.initReport-Protokollnachricht für den `snap-start`-Initialisierungstyp.

```
{
  "time":"2022-07-17T18:41:57.083Z",
  "type":"platform.initReport",
  "record":{
      "initializationType":"snap-start",
      "metrics":{
          "durationMs":731.79,
          "billedDurationMs":732
          }
  }
}
```

Das Berichtsprotokoll der Plattform enthält Metriken über den Aufruf, den die RequestID angibt. Das `initDurationMs`-Feld ist nur dann im Protokoll enthalten, wenn der Aufruf einen Kaltstart enthielt. Wenn die AWS X-Ray -Ablaufverfolgung aktiv ist, enthält das Protokoll X-Ray-Metadaten. Das folgende Beispiel zeigt ein Plattformberichtsprotokoll für einen Aufruf, der einen Kaltstart enthielt.

```
{
    "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
        }
    }
}
```

Das Plattformfehlerprotokoll erfasst Fehler bei der Laufzeit oder der Ausführungsumgebung. Das folgende Beispiel zeigt eine Fehlerprotokollmeldung der Plattform.

```
{
    "time": "2020-08-20T12:31:32.123Z",
    "type": "platform.fault",
    "record": "RequestId: d783b35e-a91d-4251-af17-035953428a2c Process exited before
 completing request"
}
```

> **Note**
>
> AWS implementiert derzeit Änderungen am Lambda-Service. Aufgrund dieser Änderungen können geringfügige Unterschiede in Struktur und Inhalt der Systemprotokollmeldungen und Trace-Segmente auftreten, die von verschiedenen Lambda-Funktionen in Ihrem AWS-Konto. Eine der von dieser Änderung betroffenen Protokollausgaben ist das Feld `"record"` des Plattformfehlerprotokolls. Die folgenden Beispiele zeigen illustrative `"record"`-Felder im alten und neuen Format. Der neue Stil des Fehlerprotokolls enthält eine prägnantere Meldung Diese Änderungen werden in den kommenden Wochen umgesetzt, und alle Funktionen AWS-Regionen außer China und den GovCloud Regionen werden auf die Verwendung der Protokollnachrichten und Trace-Segmente im neuen Format umgestellt.

Example Aufzeichnung des Plattformfehlerprotokolls (alter Stil)

```
"record":"RequestId: ...\tError: Runtime exited with error: exit status
 255\nRuntime.ExitError"
```

Example Aufzeichnung des Plattformfehlerprotokolls (neuer Stil)

```
"record":"RequestId: ... Status: error\tErrorType: Runtime.ExitError"
```

Lambda generiert ein Plattformerweiterungsprotokoll, wenn sich eine Erweiterung bei der Erweiterungs-API registriert. Das folgende Beispiel zeigt eine Plattformerweiterungsmeldung.

```
{
    "time": "2020-08-20T12:31:32.123Z",
    "type": "platform.extension",
    "record": {"name": "Foo.bar",
        "state": "Ready",
        "events": ["INVOKE", "SHUTDOWN"]
     }
}
```

Lambda generiert ein Plattformprotokoll-Abonnementprotokoll, wenn eine Erweiterung die Protokoll-API abonniert. Das folgende Beispiel zeigt eine Protokoll-Abonnementmeldung.

```
{
    "time": "2020-08-20T12:31:32.123Z",
    "type": "platform.logsSubscription",
    "record": {"name": "Foo.bar",
        "state": "Subscribed",
        "types": ["function", "platform"],
    }
}
```

Lambda generiert ein vom Plattformprotokoll abgelegtes Protokoll, wenn eine Erweiterung nicht in der Lage ist, die Anzahl der Protokolle zu verarbeiten, die sie empfängt. Das folgende Beispiel zeigt eine `platform.logsDropped`-Protokollmeldung.

```
{
    "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
    }
}
```

Die platform.restoreStart-Protokollnachricht zeigt den Zeitpunkt an, zu dem die `Restore`-Phase begonnen hat (nur `snap-start`-Initialisierungstyp). Beispiel:

```
{
  "time":"2022-07-17T18:43:44.782Z",
  "type":"platform.restoreStart",
  "record":{}
}
```

Die platform.restoreReport-Protokollnachricht zeigt, wie lange die `Restore`-Phase gedauert hat und wie viele Millisekunden Ihnen während dieser Phase in Rechnung gestellt wurden (nur `snap-start`-Initialisierungstyp). Beispiel:

```
{
  "time":"2022-07-17T18:43:45.936Z",
  "type":"platform.restoreReport",
  "record":{
      "metrics":{
          "durationMs":70.87,
          "billedDurationMs":13
      }
  }
}
```

Plattform **`runtimeDone`**-Meldungen

Wenn Sie die Schemaversion in der Abonnementanforderung auf „2021-03-18“ setzen, schickt Lambda eine `platform.runtimeDone`-Meldung, nachdem der Funktionsaufruf entweder erfolgreich oder mit einem Fehler abgeschlossen wurde. Die Erweiterung kann diese Meldung verwenden, um die gesamte Telemetrie-Sammlung für diesen Funktionsaufruf anzuhalten.

Die OpenAPI-Spezifikation für den Protokollereignistyp in der Schemaversion 2021-03-18 ist hier verfügbar: schema-2021-03-18.zip

Lambda generiert die `platform.runtimeDone`-Protokollmeldung, wenn die Laufzeitumgebung eine `Next`- oder `Error`-Laufzeit-API-Anforderung sendet. Das `platform.runtimeDone`-Protokoll informiert die Verbraucher über die Protokoll-API, dass der Funktionsaufruf abgeschlossen ist. Erweiterungen können diese Informationen verwenden, um zu entscheiden, wann die gesamte während dieses Aufrufs gesammelte Telemetrie gesendet werden soll.

Beispiele

Lambda sendet die `platform.runtimeDone`-Meldung, nachdem die Laufzeit die NEXT-Anfrage sendet, wenn der Funktionsaufruf abgeschlossen wird. Die folgenden Beispiele zeigen Meldungen für jeden der Statuswerte: Erfolg, Misserfolg und Timeout.

Example Beispiel Erfolgsmeldung

```
{
    "time": "2021-02-04T20:00:05.123Z",
    "type": "platform.runtimeDone",
    "record": {
       "requestId":"6f7f0961f83442118a7af6fe80b88",
       "status": "success"
    }
}
```

Example Beispiel Misserfolgsmeldung

```
{
   "time": "2021-02-04T20:00:05.123Z",
   "type": "platform.runtimeDone",
   "record": {
      "requestId":"6f7f0961f83442118a7af6fe80b88",
      "status": "failure"
   }
}
```

Example Beispiel Timeout-Meldung

```
{
   "time": "2021-02-04T20:00:05.123Z",
   "type": "platform.runtimeDone",
   "record": {
      "requestId":"6f7f0961f83442118a7af6fe80b88",
```

```
      "status": "timeout"
  }
}
```

Example Beispiel für eine Plattform. restoreRuntimeDone Nachricht (nur **`snap-start`** Initialisierungstyp)

Die Plattform. restoreRuntimeDoneDie Protokollnachricht zeigt, ob die `Restore` Phase erfolgreich war oder nicht. Lambda sendet diese Nachricht, wenn die Laufzeit eine `restore/next`-Laufzeit-API-Anfrage sendet. Es gibt drei mögliche Status: erfolgreich, fehlgeschlagen und Timeout. Das folgende Beispiel zeigt eine erfolgreiche Plattform. restoreRuntimeDoneNachricht protokollieren.

```
{
  "time":"2022-07-17T18:43:45.936Z",
  "type":"platform.restoreRuntimeDone",
  "record":{
      "status":"success"
  }
}
```

Beheben von Problemen in Lambda

Die folgenden Themen enthalten Ratschläge zur Fehlerbehebung bei Fehlern und Problemen, die bei der Verwendung der Lambda-API, -Konsole oder -Tools auftreten können. Wenn Sie auf ein Problem stoßen, das hier nicht aufgeführt ist, können Sie die Schaltfläche Feedback auf dieser Seite verwenden, um es zu melden.

Weitere Tipps zur Fehlerbehebung und Antworten auf häufig gestellte Supportfragen finden Sie im AWS- Wissenscenter.

Weitere Informationen zum Debuggen und zur Problembehandlung für Lambda-Anwendungen finden Sie bei Serverless Land unter Debugging.

Themen

- Beheben von Konfigurationsproblemen in Lambda
- Beheben von Bereitstellungsproblemen in Lambda
- Beheben von Aufruf-Problemen in Lambda
- Beheben von Ausführungsproblemen in Lambda
- Behebung von Problemen mit der Zuordnung von Ereignisquellen in Lambda
- Beheben von Netzwerkproblemen in Lambda

Beheben von Konfigurationsproblemen in Lambda

Ihre Funktionskonfigurationseinstellungen können sich auf die Gesamtleistung und das Verhalten Ihrer Lambda-Funktion auswirken. Diese verursachen möglicherweise keine tatsächlichen Funktionsfehler, können jedoch zu unerwarteten Timeouts und Ergebnissen führen.

Die folgenden Themen enthalten Tipps zur Fehlerbehebung für häufig auftretende Probleme im Zusammenhang mit den Konfigurationseinstellungen von Lambda-Funktionen.

Themen

- Arbeitsspeicherkonfigurationen
- CPU-gebundene Konfigurationen
- Zeitüberschreitungen
- Speicherlecks zwischen Aufrufen

- Asynchrone Ergebnisse wurden bei einem späteren Aufruf zurückgegeben

Arbeitsspeicherkonfigurationen

Sie können eine Lambda-Funktion so konfigurieren, dass sie zwischen 128 MB und 10 240 MB Speicher verwendet. Standardmäßig wird jeder in der Konsole erstellten Funktion die kleinste Speichermenge zugewiesen. Viele Lambda-Funktionen sind bei dieser niedrigsten Einstellung leistungsfähig. Wenn Sie jedoch große Codebibliotheken importieren oder speicherintensive Aufgaben ausführen, reichen 128 MB nicht aus.

Wenn Ihre Funktionen viel langsamer als erwartet ausgeführt werden, besteht der erste Schritt darin, die Speichereinstellung zu erhöhen. So können Sie die Funktion Ihres Handlers nutzen, um die Leistung Ihrer Funktion zu verbessern.

CPU-gebundene Konfigurationen

Bei rechenintensiven Vorgängen kann es vorkommen, dass Ihre Funktion eine geringere Leistung als erwartet aufweist. Dies kann daran liegen, dass Ihre Funktion CPU-gebunden ist. In diesem Fall kann die Berechnungskapazität der Funktion nicht mit der Arbeit Schritt halten.

Mit Lambda können Sie die CPU-Konfiguration zwar nicht direkt ändern, die CPU wird jedoch indirekt über die Speichereinstellungen gesteuert. Der Lambda-Dienst weist proportional mehr virtuelle CPU zu, wenn Sie mehr Arbeitsspeicher zuweisen. Bei 1,8 GB Arbeitsspeicher wird einer Lambda-Funktion eine ganze vCPU zugewiesen und oberhalb dieser Grenze hat sie Zugriff auf mehr als einen vCPU-Kern. Mit 10 240 MB hat sie 6 vCPUs zur Verfügung. Mit anderen Worten: Sie können die Leistung verbessern, indem Sie die Speicherzuweisung erhöhen, auch wenn die Funktion nicht den gesamten Speicher nutzt.

Zeitüberschreitungen

Timeouts für Lambda-Funktionen können zwischen 1 und 900 Sekunden (15 Minuten) eingestellt werden. Standardmäßig ist dieser Wert in der Lambda-Konsole auf 3 Sekunden eingestellt. Der Timeout-Wert ist eine Sicherheitsvorrichtung, die sicherstellt, dass Funktionen nicht unbegrenzt ausgeführt werden. Nachdem der Timeout-Wert erreicht ist, stoppt Lambda den Funktionsaufruf.

Wenn ein Timeout-Wert nahe der durchschnittlichen Dauer einer Funktion festgelegt wird, erhöht sich das Risiko, dass die Funktion unerwartet abbricht. Die Dauer einer Funktion kann je nach Umfang der Datenübertragung und -verarbeitung sowie der Latenz aller Dienste, mit denen die Funktion interagiert, variieren. Häufige Ursachen für Timeouts:

- Beim Herunterladen von Daten aus S3-Buckets oder anderen Datenspeichern ist der Download größer oder dauert länger als im Durchschnitt.
- Eine Funktion sendet eine Anfrage an einen anderen Service, dessen Beantwortung länger dauert.
- Die einer Funktion zur Verfügung gestellten Parameter erfordern eine höhere Rechenkomplexität in der Funktion, wodurch der Aufruf länger dauert.

Stellen Sie beim Testen Ihrer Anwendung sicher, dass Ihre Tests die Größe und Menge der Daten sowie realistische Parameterwerte genau wiedergeben. Wichtig ist, dass Sie Datensätze verwenden, die an der Obergrenze dessen liegen, was für Ihren Workload vernünftigerweise zu erwarten ist.

Implementieren Sie zusätzlich Obergrenzen für Ihren Workload, wo immer dies möglich ist. In diesem Beispiel könnte die Anwendung für jeden Dateityp eine maximale Größenbeschränkung verwenden. Anschließend können Sie die Leistung Ihrer Anwendung für eine Reihe erwarteter Dateigrößen bis einschließlich der Höchstgrenzen testen.

Speicherlecks zwischen Aufrufen

Globale Variablen und Objekte, die in der INIT-Phase eines Lambda-Aufrufs gespeichert sind, behalten ihren Status zwischen warmen Aufrufen bei. Sie werden erst vollständig zurückgesetzt, wenn die Ausführungsumgebung zum ersten Mal ausgeführt wird (auch als „Kaltstart“ bezeichnet). Alle im Handler gespeicherten Variablen werden zerstört, wenn der Handler beendet wird. Die INIT-Phase wird am besten zum Einrichten von Datenbankverbindungen, Laden von Bibliotheken, Erstellen von Caches und Laden unveränderlicher Assets verwendet.

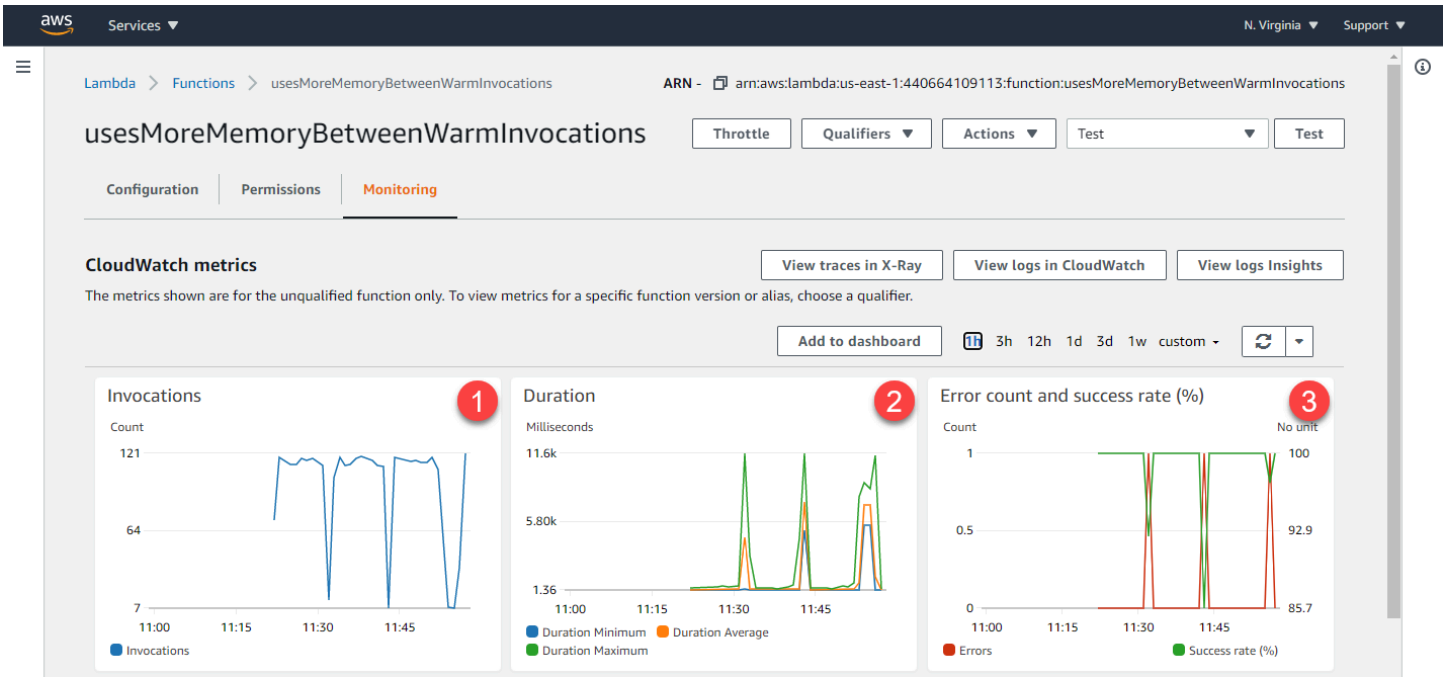
Wenn Sie Bibliotheken von Drittanbietern für mehrere Aufrufe in derselben Ausführungsumgebung verwenden, sollten Sie deren Dokumentation zur Verwendung in einer Serverless-Computerumgebung überprüfen. Einige Bibliotheken für Datenbankverbindungen und Protokollierung speichern möglicherweise Zwischenaufrufergebnisse und andere Daten. Dies führt dazu, dass der Speicherverbrauch dieser Bibliotheken mit nachfolgenden Warmaufrufen zunimmt. In diesem Fall kann es vorkommen, dass die Lambda-Funktion nicht über genügend Speicher verfügt, selbst wenn Ihr benutzerdefinierter Code Variablen korrekt entsorgt.

Dieses Problem betrifft Aufrufe, die in warmen Ausführungsumgebungen auftreten. Der folgende Code erzeugt beispielsweise ein Speicherleck zwischen Aufrufen. Die Lambda-Funktion verbraucht bei jedem Aufruf zusätzlichen Speicher, indem sie die Größe eines globalen Arrays erhöht:

```
let a = []
```

```
exports.handler = async (event) => {
    a.push(Array(100000).fill(1))
}
```

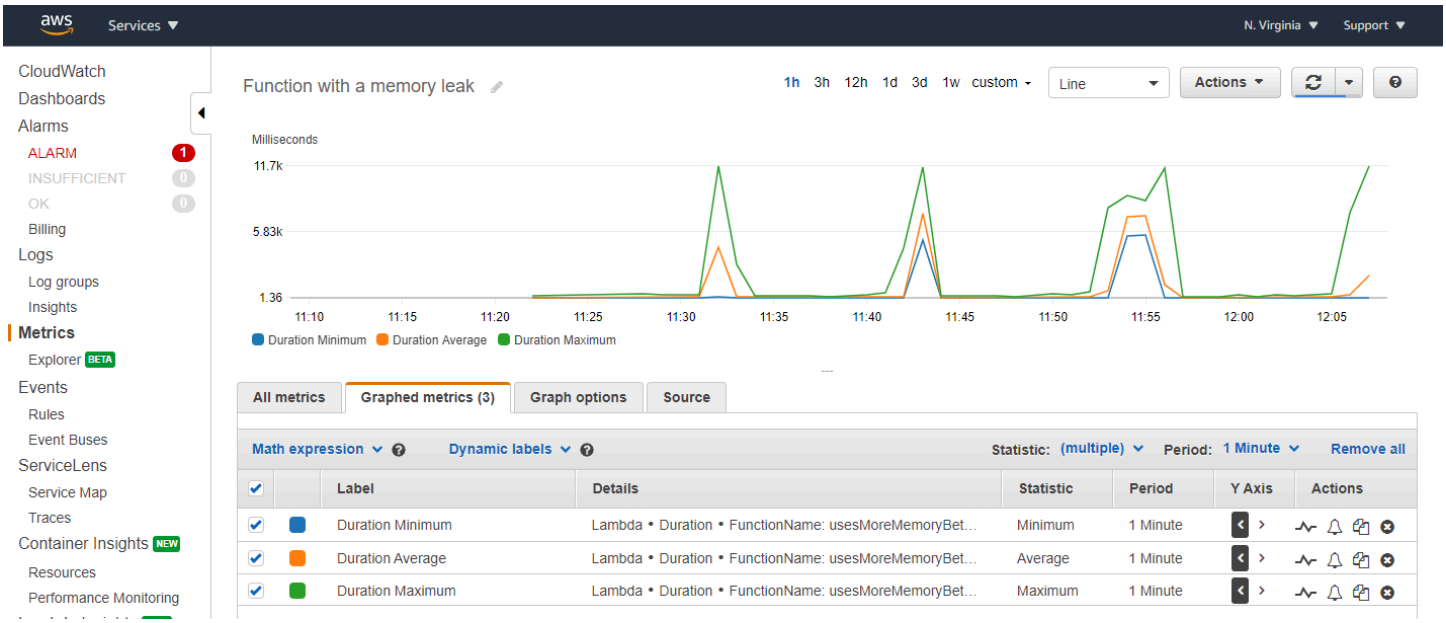
Bei einer Konfiguration mit 128 MB Speicher zeigt die Registerkarte Überwachung der Lambda-Funktion nach 1 000 Aufrufen dieser Funktion die typischen Änderungen bei Aufrufen, Dauer und Fehleranzahl, wenn ein Speicherleck auftritt:



1. Aufrufe: Eine gleichmäßige Transaktionsrate wird periodisch unterbrochen, da die Aufrufe länger dauern, bis sie abgeschlossen sind. Im eingeschwungenen Zustand verbraucht das Speicherleck nicht den gesamten der Funktion zugewiesenen Speicher. Wenn die Leistung abnimmt, paginiert das Betriebssystem den lokalen Speicher, um den wachsenden Speicherbedarf der Funktion zu decken, was dazu führt, dass weniger Transaktionen abgeschlossen werden.
2. Dauer: Bevor der Funktion der Speicherplatz ausgeht, beendet sie Aufrufe mit einer konstanten zweistelligen Millisekundenrate. Wenn ein Paging stattfindet, verlängert sich die Dauer um eine Größenordnung.
3. Anzahl der Fehler: Wenn der Speicherverlust den zugewiesenen Speicher übersteigt, kommt es schließlich zu einem Funktionsfehler, weil die Berechnung den Timeout überschreitet, oder die Ausführungsumgebung stoppt die Funktion.

Nach dem Fehler startet Lambda die Ausführungsumgebung neu, was erklärt, warum alle drei Diagramme eine Rückkehr zum ursprünglichen Zustand zeigen. Wenn Sie die CloudWatch-Metriken

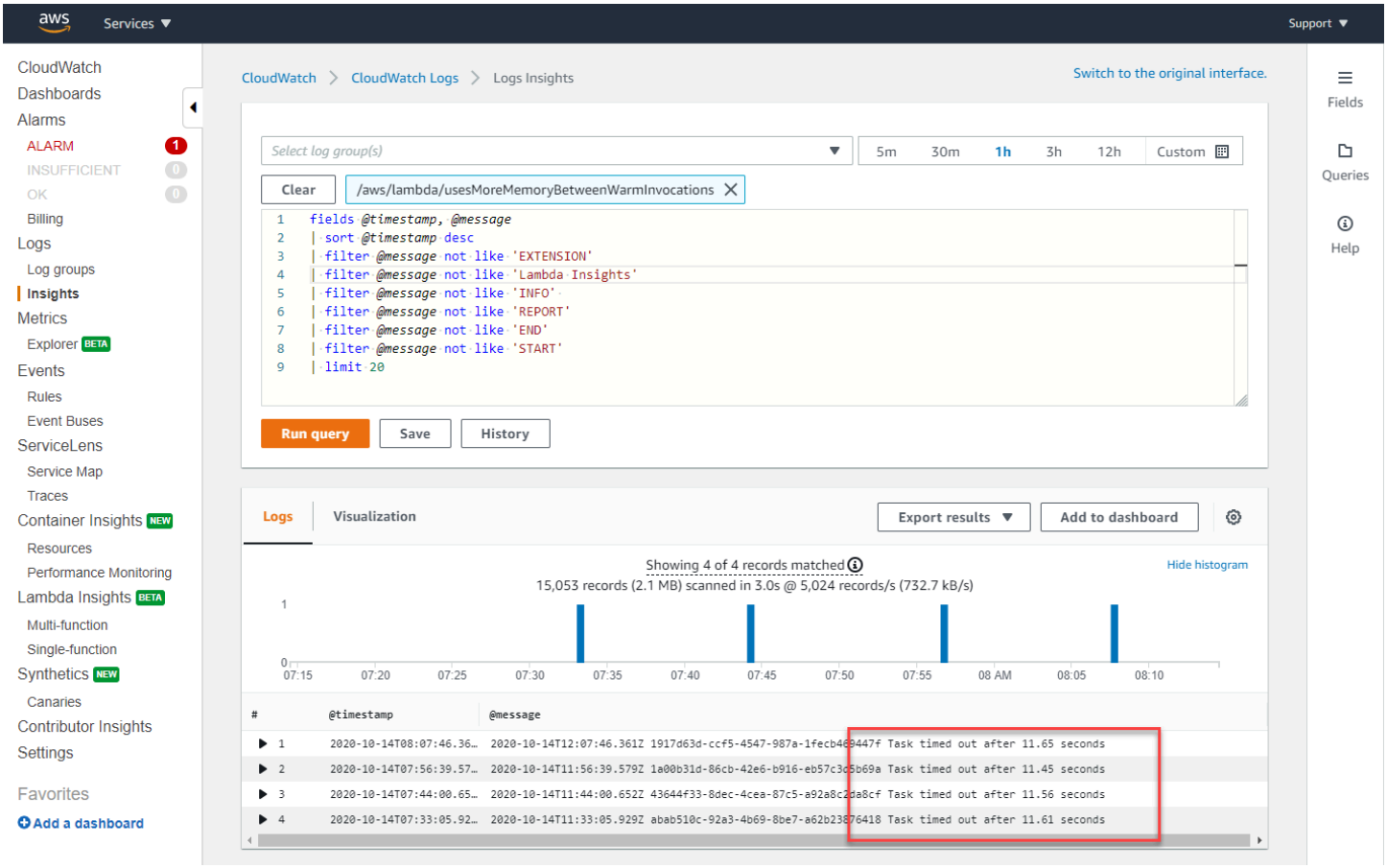
für die Dauer erweitern, erhalten Sie mehr Details zu den Statistiken für die minimale, maximale und durchschnittliche Dauer:



Um die Fehler zu finden, die bei den 1 000 Aufrufen erzeugt wurden, können Sie die Abfragesprache CloudWatch Insights verwenden. Die folgende Abfrage schließt Informationsprotokolle aus, um nur die Fehler zu melden:

```
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'
```

Ein Abgleich mit der Protokollgruppe für diese Funktion zeigt, dass Zeitüberschreitungen für die periodischen Fehler verantwortlich waren:



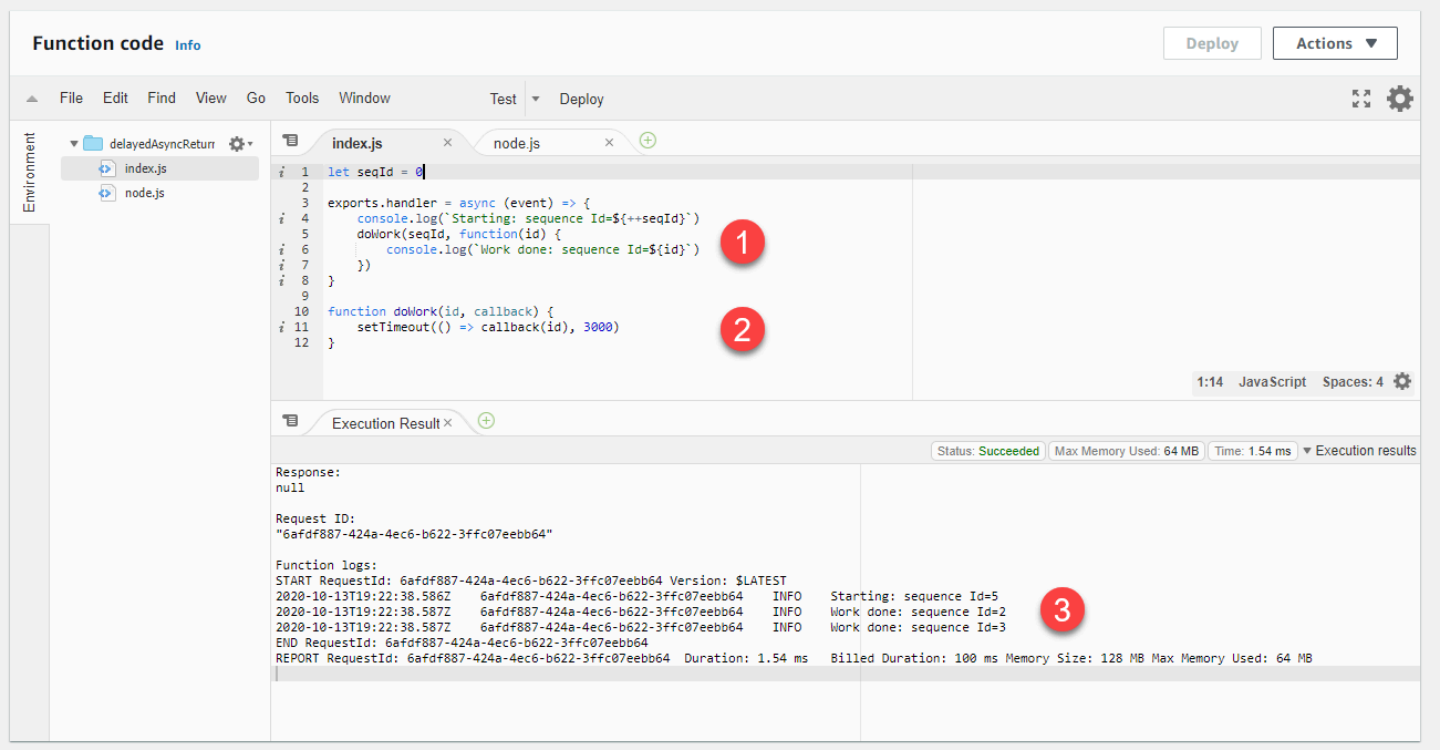
Asynchrone Ergebnisse wurden bei einem späteren Aufruf zurückgegeben

Bei Funktionscode, der asynchrone Muster verwendet, ist es möglich, dass die Callback-Ergebnisse eines Aufrufs in einem zukünftigen Aufruf zurückgegeben werden. In diesem Beispiel wird Node.js verwendet, aber dieselbe Logik kann auch auf andere Laufzeiten angewendet werden, die asynchrone Muster verwenden. Die Funktion verwendet die traditionelle Callback-Syntax in JavaScript. Sie ruft eine asynchrone Funktion mit einem inkrementellen Zähler auf, der die Anzahl der Aufrufe verfolgt:

```
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)
}
```

Wenn sie mehrmals hintereinander aufgerufen werden, treten die Ergebnisse der Callbacks bei nachfolgenden Aufrufen auf:



1. Der Code ruft die `doWork`-Funktion auf und stellt als letzten Parameter eine Callback-Funktion bereit.
2. Es dauert einige Zeit, bis die `doWork`-Funktion abgeschlossen ist, bevor der Callback aufgerufen wird.
3. Die Protokollierung der Funktion zeigt an, dass der Aufruf beendet wird, bevor die Ausführung der `doWork`-Funktion abgeschlossen ist. Darüber hinaus werden nach dem Start einer Iteration Callbacks aus früheren Iterationen verarbeitet, wie in den Protokollen gezeigt.

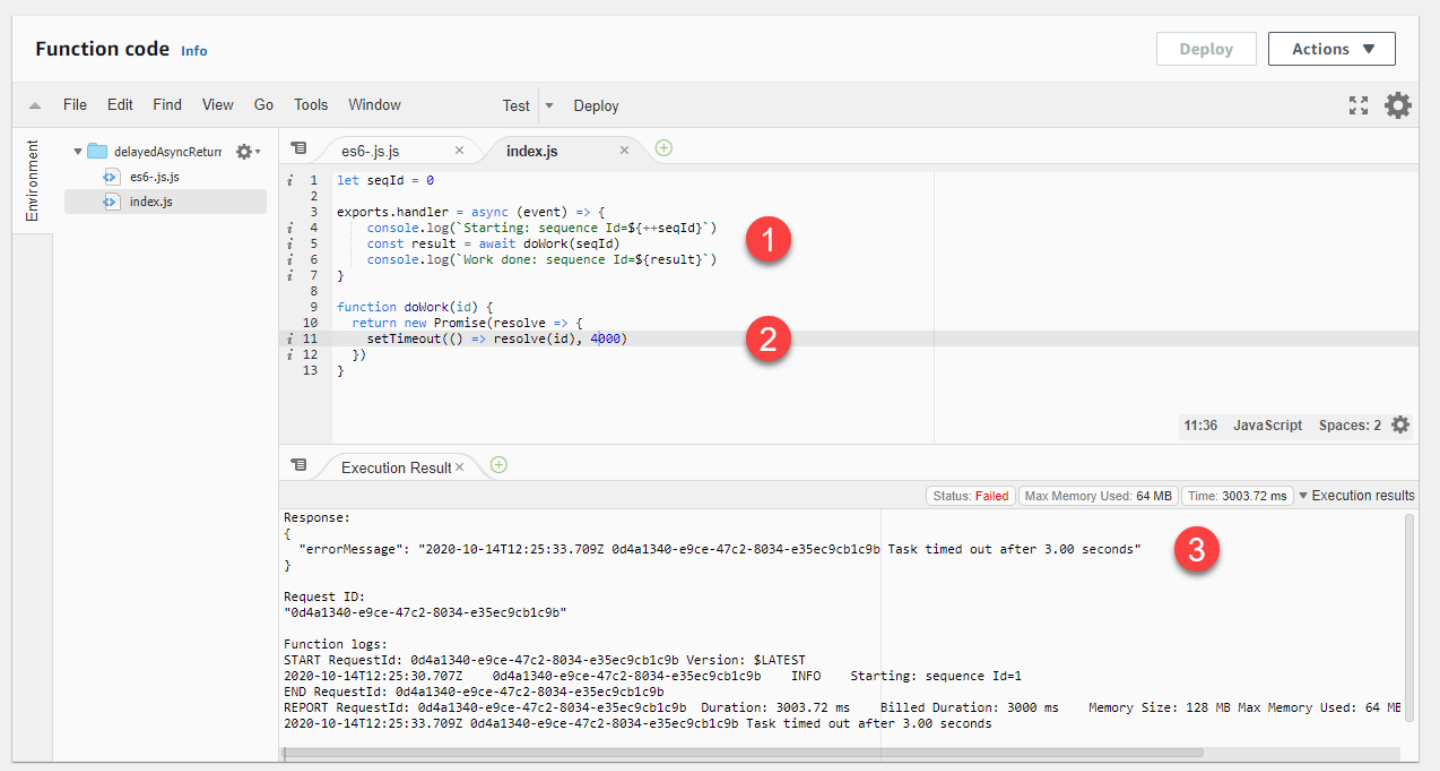
In JavaScript werden asynchrone Callbacks mit einer Event-Schleife behandelt. Andere Laufzeiten verwenden andere Mechanismen zur Behandlung der Gleichzeitigkeit. Wenn die Ausführungsumgebung der Funktion endet, friert Lambda die Umgebung bis zum nächsten Aufruf ein. Nach der Wiederaufnahme setzt JavaScript die Verarbeitung der Ereignisschleife fort, die in diesem Fall einen asynchronen Rückruf von einem früheren Aufruf enthält. Ohne diesen Kontext kann es den Anschein haben, dass die Funktion ohne Grund Code ausführt und beliebige Daten

zurückgibt. Sie ist vielmehr ein Artefakt der Interaktion zwischen der Gleichzeitigkeit zur Laufzeit und den Ausführungsumgebungen.

Dadurch besteht die Möglichkeit, dass private Daten aus einem früheren Aufruf in einem späteren Aufruf erscheinen. Es gibt zwei Möglichkeiten, die Funktion Ihres Handlers aufzuweisen. Erstens bietet JavaScript die Schlüsselwörter async und await, um die asynchrone Entwicklung zu vereinfachen und die Codeausführung zu erzwingen, bis ein asynchroner Aufruf abgeschlossen ist. Die obige Funktion kann mit diesem Ansatz wie folgt umgeschrieben werden:

```
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)
  })
}
```

Die Verwendung dieser Syntax verhindert, dass der Handler beendet wird, bevor die asynchrone Funktion beendet ist. Wenn in diesem Fall der Callback länger dauert als das Timeout der Lambda-Funktion, gibt die Funktion einen Fehler aus, anstatt das Callback-Ergebnis bei einem späteren Aufruf zurückzugeben:

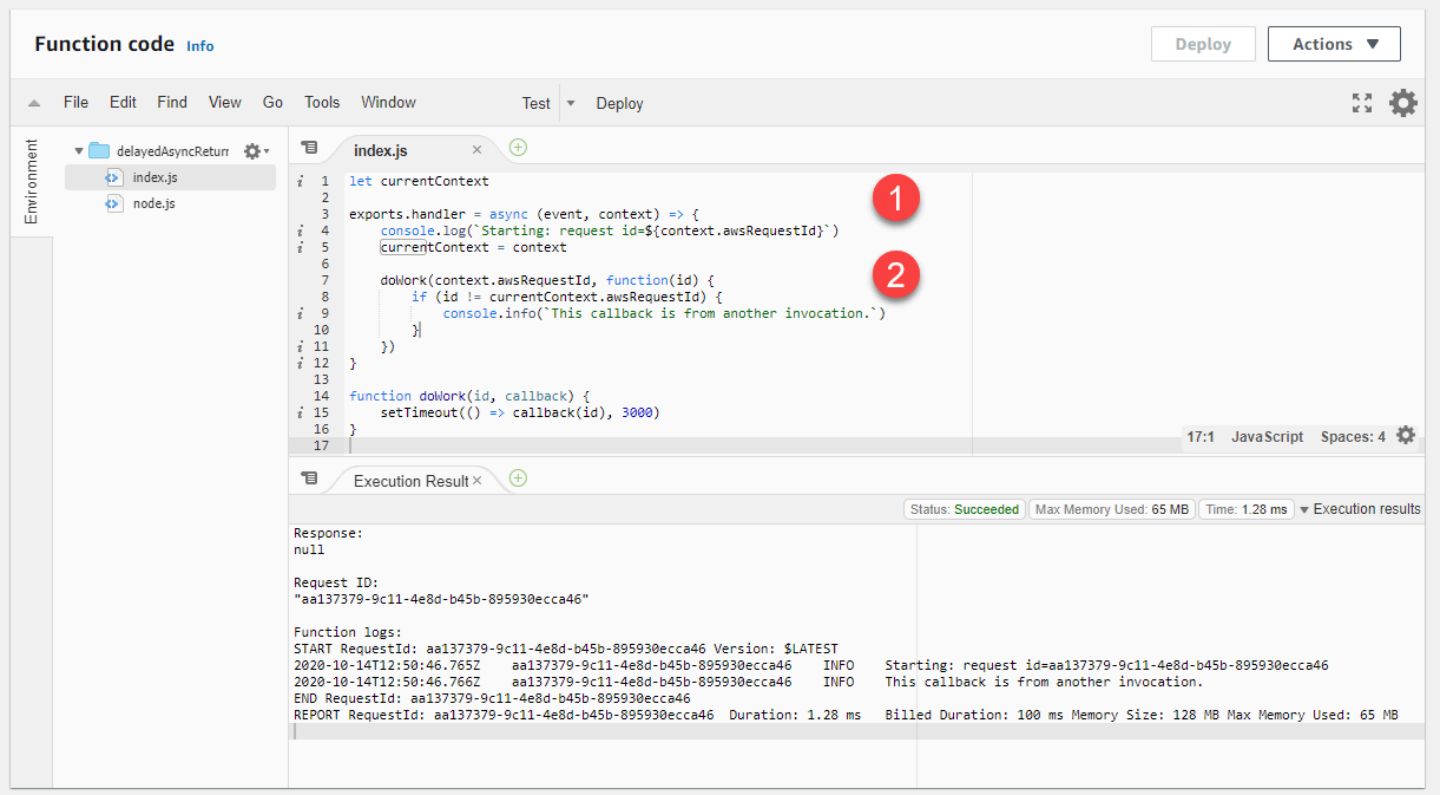


1. Der Code ruft die asynchrone `doWork`-Funktion unter Verwendung des Schlüsselworts „await“ im Handler auf.
2. Die Funktion `doWork` benötigt eine gewisse Zeit, bevor sie das Versprechen auflöst.
3. Bei der Funktion kommt es zu einem Timeout, da `doWork` länger dauert, als der Timeout erlaubt, und das Ergebnis des Rückrufs bei einem späteren Aufruf nicht zurückgegeben wird.

Im Allgemeinen sollten Sie sicherstellen, dass alle Hintergrundprozesse oder Rückrufe im Code abgeschlossen sind, bevor der Code beendet wird. Wenn dies in Ihrem Anwendungsfall nicht möglich ist, können Sie einen Bezeichner verwenden, um sicherzustellen, dass der Rückruf zum aktuellen Aufruf gehört. Dazu können Sie die vom Kontextobjekt bereitgestellte awsRequestId verwenden. Indem Sie diesen Wert an den asynchronen Callback übergeben, können Sie den übergebenen Wert mit dem aktuellen Wert vergleichen, um festzustellen, ob der Callback von einem anderen Aufruf stammt:

```
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. Der Lambda-Funktionshandler verwendet den Kontextparameter, der Zugriff auf eine eindeutige Aufrufanforderungs-ID ermöglicht.
2. Die `awsRequestId` wird an die doWork-Funktion übergeben. Im Callback wird die ID mit der `awsRequestId` des aktuellen Aufrufs verglichen. Wenn diese Werte unterschiedlich sind, kann der Code entsprechende Maßnahmen ergreifen.

Beheben von Bereitstellungsproblemen in Lambda

Wenn Sie Ihre Funktion aktualisieren, stellt Lambda die Änderung durch Starten der neuen Instances der Funktion mit aktualisiertem Code oder aktualisierten Einstellungen bereit. Bereitstellungsfehler verhindern die Verwendung der neuen Version und können auf Probleme mit Bereitstellungspaket, Code, Berechtigungen oder Tools zurückzuführen sein.

Wenn Sie Updates für Ihre Funktion direkt mit der Lambda-API oder mit einem Client wie dem bereitstellen AWS CLI, können Sie Fehler von Lambda direkt in der Ausgabe sehen. Wenn Sie Dienste wie AWS CloudFormation, oder verwenden AWS CodeDeploy AWS CodePipeline, suchen Sie in den Protokollen oder im Eventstream für diesen Dienst nach der Antwort von Lambda.

Die folgenden Themen enthalten Ratschläge zur Fehlerbehebung bei Fehlern und Problemen, die bei der Verwendung der Lambda-API, -Konsole oder -Tools auftreten können. Wenn Sie auf ein Problem stoßen, das hier nicht aufgeführt ist, können Sie die Schaltfläche Feedback auf dieser Seite verwenden, um es zu melden.

Weitere Tipps zur Fehlerbehebung und Antworten auf häufig gestellte Supportfragen finden Sie im AWS - Wissenscenter.

Weitere Informationen zum Debuggen und zur Problembehandlung für Lambda-Anwendungen finden Sie bei Serverless Land unter Debugging.

Themen

- Allgemein: Berechtigung wird verweigert/Kann solche Datei nicht laden
- Allgemein: Beim Aufrufen von UpdateFunctionCode
- Amazon S3: Fehlercode PermanentRedirect.
- Allgemein: Kann nicht gefunden werden, kann nicht geladen werden, Klasse nicht gefunden, keine solche Datei oder kein Verzeichnis
- Allgemein: Undefinierter Methodenhandler
- Allgemein: Speicherlimit für Lambda-Code überschritten
- Lambda: Ebenenkonvertierung ist fehlgeschlagen.
- Lambda: oder InvalidParameterValueException RequestEntityTooLargeException
- Lambda: InvalidParameterValueException
- Lambda: Kontingente für Gleichzeitigkeit und Speicher
- Lambda: Ungültige Aliaskonfiguration für bereitgestellte Gleichzeitigkeit

Allgemein: Berechtigung wird verweigert/Kann solche Datei nicht laden

Fehler: EACCES: Zugriff verweigert, öffnen Sie '/ .js' var/task/index

Fehler: kann eine derartige Datei nicht laden – Funktion

Fehler: [Errno 13] Zugriff verweigert: '/ .py' var/task/function

Die Lambda-Laufzeit benötigt die Berechtigung zum Lesen der Dateien in Ihrem Bereitstellungspaket. In der oktalen Schreibweise von Linux-Berechtigungen benötigt Lambda 644 Berechtigungen für nicht ausführbare Dateien (rw-r--r--) und 755 Berechtigungen () für Verzeichnisse und ausführbare Dateien. rwxr-xr-x

Verwenden Sie unter Linux und MacOS den `chmod`-Befehl, um Dateiberechtigungen für Dateien und Verzeichnisse in Ihrem Bereitstellungspaket zu ändern. Um beispielsweise einer nicht ausführbaren Datei die richtigen Berechtigungen zu geben, führen Sie den folgenden Befehl aus.

```
chmod 644 <filepath>
```

Informationen zum Ändern von Dateiberechtigungen in Windows finden Sie unter Festlegen, Anzeigen, Ändern oder Entfernen von Berechtigungen für ein Objekt in der Microsoft-Windows-Dokumentation.

> **Note**
>
> Wenn Sie Lambda nicht die Berechtigungen gewähren, die es für den Zugriff auf Verzeichnisse in Ihrem Bereitstellungspaket benötigt, setzt Lambda die Berechtigungen für diese Verzeichnisse auf 755 (). rwxr-xr-x

Allgemein: Beim Aufrufen von UpdateFunctionCode

Fehler: Beim Aufrufen der UpdateFunctionCode Operation ist ein Fehler aufgetreten (RequestEntityTooLargeException)

Wenn Sie ein Bereitstellungspaket oder ein Ebenenarchiv direkt in Lambda hochladen, ist die Größe der ZIP-Datei auf 50 MB begrenzt. Um eine größere Datei hochzuladen, speichern Sie sie in Amazon S3 und verwenden Sie die S3Bucket- und S3Key--Parameter.

> **Note**
>
> Wenn Sie eine Datei direkt mit dem AWS CLI AWS SDK oder auf andere Weise hochladen, wird die binäre ZIP-Datei in ein Base64-Format konvertiert, wodurch sich ihre Größe um etwa 30% erhöht. Um dies und die Größe anderer Parameter in der Anforderung zu ermöglichen, ist die tatsächliche von Lambda angewendete Größenbeschränkung der Anforderung größer. Aus diesem Grund ist der Grenzwert von 50 MB ein ungefährer Wert.

Amazon S3: Fehlercode PermanentRedirect.

Fehler: Während ist ein Fehler aufgetreten GetObject. S3-Fehlercode: PermanentRedirect. S3-Fehlermeldung: Der Bucket befindet sich in dieser Region: us-east-2. Verwenden Sie diese Region, um die Anfrage zu wiederholen

Wenn Sie das Bereitstellungspaket einer Funktion aus einem Amazon-S3-Bucket hochladen, muss sich der Bucket in derselben Region wie die Funktion befinden. Dieses Problem kann auftreten, wenn Sie ein Amazon S3 S3-Objekt in einem Aufruf angeben oder die Befehle package und deploy in der AWS CLI oder AWS SAM CLI verwenden. UpdateFunctionCode Erstellen Sie einen Bucket mit Bereitstellungsartefakt für jede Region, in der Sie Anwendungen entwickeln.

Allgemein: Kann nicht gefunden werden, kann nicht geladen werden, Klasse nicht gefunden, keine solche Datei oder kein Verzeichnis

Fehler: Modul 'function' kann nicht gefunden werden

Fehler: kann eine derartige Datei nicht laden – Funktion

Fehler: Modul 'function' kann nicht importiert werden

Fehler: Klasse nicht gefunden: function.Handler

Fehlerfork/exec /var/task/function: Keine solche Datei oder kein solches Verzeichnis

Fehler: Der Typ 'Function.Handler' kann nicht aus der Assembly 'Function' geladen werden.'

Der Name der Datei oder Klasse in der Handler-Konfiguration Ihrer Funktion stimmt nicht mit Ihrem Code überein. Weitere Informationen finden Sie im folgenden Abschnitt .

Allgemein: Undefinierter Methodenhandler

Fehler: index.handler ist nicht definiert oder wurde nicht exportiert

Fehler: Handler 'handler' fehlt für Modul 'function'

Fehler: undefinierte Methode `Handler' für #<:0x000055b76ccebf98> LambdaHandler

Fehler: Keine öffentliche Methode namens handleRequest mit der entsprechenden Methodensignatur für Klasse function.Handler gefunden

Fehler: Die Methode 'handleRequest' konnte nicht im Typ 'Function.Handler' der Assembly 'Function' gefunden werden

Der Name der Handler-Methode in der Handler-Konfiguration Ihrer Funktion stimmt nicht mit Ihrem Code überein. Jede Laufzeit definiert eine Namenskonvention für Handler, wie z. *`filename methodname`*. Der Handler ist die Methode im Code Ihrer Funktion, die von der Laufzeit ausführt wird, wenn Ihre Funktion aufgerufen wird.

Lambda stellt für einige Sprachen eine Bibliothek mit einer Schnittstelle bereit, die erwartet, dass eine Handler-Methode einen bestimmten Namen hat. Weitere Informationen zur Handlerbenennung für jede Sprache finden Sie in den folgenden Themen.

- Erstellen von Lambda-Funktionen mit Node.js
- Erstellen von Lambda-Funktionen mit Python
- Erstellen von Lambda-Funktionen mit Ruby
- Erstellen von Lambda-Funktionen mit Java
- Erstellen von Lambda-Funktionen mit Go
- Erstellen von Lambda-Funktionen mit C#
- Aufbau von Lambda-Funktionen mit PowerShell

Allgemein: Speicherlimit für Lambda-Code überschritten

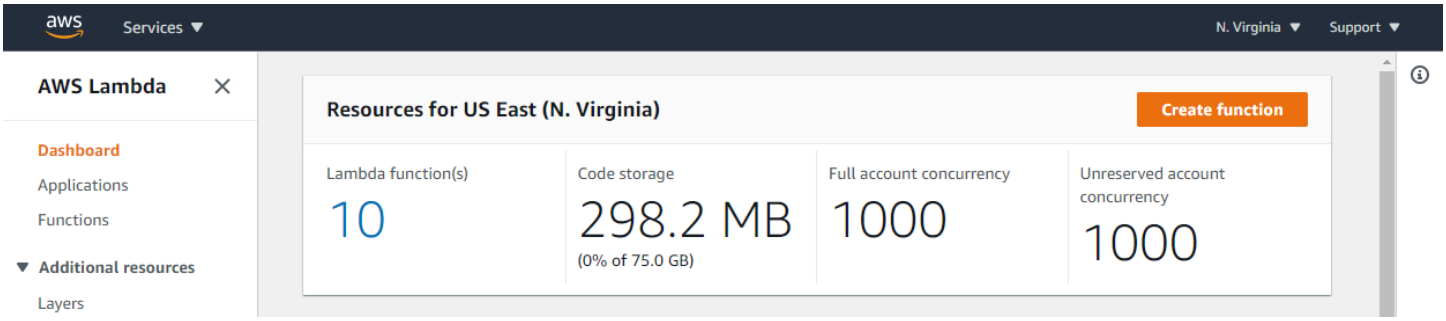
Fehler: Das Code-Speicherlimit wurde überschritten.

Lambda speichert Ihren Funktionscode in einem internen S3-Bucket, der nur für Ihr Konto bestimmt ist. Jedem AWS Konto werden 75 GB Speicherplatz in jeder Region zugewiesen. Der Codespeicher umfasst den gesamten Speicherplatz, der sowohl von Lambda-Funktionen als

auch von Lambda-Ebenen verwendet wird. Wenn Sie das Kontingent erreichen, erhalten Sie eine, CodeStorageExceededExceptionwenn Sie versuchen, neue Funktionen bereitzustellen.

Verwalten Sie den verfügbaren Speicherplatz, indem Sie alte Versionen von Funktionen bereinigen, ungenutzten Code entfernen oder Lambda-Ebenen verwenden. Darüber hinaus empfiehlt es sich, separate AWS Konten für separate Workloads zu verwenden, um die Verwaltung der Speicherkontingente zu erleichtern.

Sie können Ihre gesamte Speichernutzung in der Lambda-Konsole im Untermenü Dashboard einsehen:



Lambda: Ebenenkonvertierung ist fehlgeschlagen.

Fehler: Lambda-Ebenenkonvertierung ist fehlgeschlagen. Hinweise zur Lösung dieses Problems finden Sie auf der Seite „Beheben von Bereitstellungsproblemen in Lambda“ im Lambda-Benutzerhandbuch.

Wenn Sie eine Lambda-Funktion mit einer Ebene konfigurieren, führt Lambda die Ebene mit Ihrem Funktionscode zusammen. Wenn dieser Prozess nicht abgeschlossen wird, gibt Lambda diesen Fehler zurück. Führen Sie in diesem Fall die folgenden Schritte aus:

- Löschen Sie alle ungenutzten Dateien von Ihrer Ebene.
- Löschen Sie alle symbolischen Links in Ihrer Ebene.
- Benennen Sie alle Dateien um, die denselben Namen wie ein Verzeichnis in einer der Ebenen Ihrer Funktion haben.

Lambda: oder InvalidParameterValueException RequestEntityTooLargeException

FehlerInvalidParameterValueException: Lambda konnte Ihre Umgebungsvariablen nicht konfigurieren, da die von Ihnen angegebenen Umgebungsvariablen das Limit von 4 KB überschritten haben. Gemessene Zeichenfolge: {"A1":" u SFe Y5 7ATNx5BSM... cyPiPn

Fehler:RequestEntityTooLargeException: Die Anfrage muss für den Vorgang kleiner als 5120 Byte sein UpdateFunctionConfiguration

Die maximale Größe des Variablenobjekts, das in der Konfiguration der Funktion gespeichert ist, darf 4096 Bytes nicht überschreiten. Dazu gehören Schlüsselnamen, Werte, Anführungszeichen, Kommas und Klammern. Die Gesamtgröße des HTTP-Anforderungstexts ist ebenfalls begrenzt.

```
{
    "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"
        }
    },
    ...
}
```

In diesem Beispiel umfasst das Objekt 39 Zeichen und benötigt 39 Bytes, wenn es als Zeichenfolge gespeichert wird (ohne Leerzeichen `{"BUCKET":"amzn-s3-demo-bucket","KEY":"file.txt"}`. Standard-ASCII-Zeichen in Umgebungsvariablenwerten verwenden jeweils ein Byte. Erweiterte ASCII- und Unicode-Zeichen können zwischen 2 und 4 Bytes pro Zeichen verwenden.

Lambda: InvalidParameterValueException

FehlerInvalidParameterValueException: Lambda konnte Ihre Umgebungsvariablen nicht konfigurieren, da die von Ihnen angegebenen Umgebungsvariablen reservierte Schlüssel enthalten, deren Änderung derzeit nicht unterstützt wird.

Lambda reserviert einige Umgebungsvariablenschlüssel zur internen Verwendung. Beispielsweise wird `AWS_REGION` von der Laufzeit verwendet, um die aktuelle Region zu bestimmen, und kann nicht außer Kraft gesetzt werden. Andere Variablen, wie z. B. `PATH`, werden von der Laufzeit verwendet, können aber in Ihrer Funktionskonfiguration erweitert werden. Eine vollständige Liste finden Sie unter Definierte Laufzeitumgebungsvariablen.

Lambda: Kontingente für Gleichzeitigkeit und Speicher

Fehler: Die ConcurrentExecutions für die Funktion angegebene Funktion senkt den Wert des Kontos UnreservedConcurrentExecution unter den Mindestwert

Fehler: Der Wert MemorySize '' hat die Einschränkung nicht erfüllt: Das Element muss einen Wert haben, der kleiner oder gleich 3008 ist

Diese Fehler treten auf, wenn Sie für Ihr Konto die Kontingente für Gleichzeitigkeit oder Speicher überschreiten. Bei neuen AWS Konten wurden Parallelität und Speicherkontingente reduziert. Um Fehler im Zusammenhang mit der Gleichzeitigkeit zu beheben, können Sie eine Kontingenterhöhung beantragen. Sie können keine Erhöhung des Speicherkontingents beantragen.

- Parallelität: Ein Fehler kann auftreten, wenn Sie wenn Sie versuchen, eine Funktion mit reservierter oder bereitgestellter Parallelität zu erstellen, oder wenn Ihre funktionsspezifische Parallelitätsanforderung (PutFunctionConcurrency) das Parallelitätskontingent Ihres Kontos übersteigt.
- Speicher: Fehler treten auf, wenn die der Funktion zugewiesene Speichermenge das Speicherkontingent Ihres Kontos übersteigt.

Lambda: Ungültige Aliaskonfiguration für bereitgestellte Gleichzeitigkeit

Fehler: Ungültige Aliaskonfiguration für bereitgestellte Gleichzeitigkeit

Dieser Fehler tritt auf, wenn Sie versuchen, den Code oder die Konfiguration einer Lambda-Funktion zu aktualisieren, während ein Alias mit bereitgestellter Gleichzeitigkeit auf eine Version verweist, bei der Probleme auftreten. Lambda initialisiert Ausführungsumgebungen für die bereitgestellte Gleichzeitigkeit vorab. Wenn diese Umgebungen aufgrund von Codefehlern, Ressourcenbeschränkungen oder einem betroffenen Stack und Alias nicht ordnungsgemäß initialisiert werden können, schlägt die Bereitstellung fehl. Führen Sie in diesem Fall die folgenden Schritte aus:

1. Den Alias rückgängig machen: Aktualisieren Sie den Alias vorübergehend, sodass er auf die zuvor funktionierende Version verweist.

```
aws lambda update-alias \
 --function-name <function-name> \
 --name <alias-name> \
 --function-version <known-good-version>
```

2. Lambda-Initialisierungscode korrigieren: Stellen Sie sicher, dass der Initialisierungscode, der außerhalb des Handlers ausgeführt wird, keine nicht abgefangenen Ausnahmen enthält, und initialisieren Sie die Clients und Verbindungen.
3. Sicherheit erneut bereitstellen: Stellen Sie korrigierten Code bereit und veröffentlichen Sie eine neue Version. Aktualisieren Sie dann den Alias so, dass er auf die korrigierte Version verweist. Aktivieren Sie optional die bereitgestellte Gleichzeitigkeit erneut, falls erforderlich.

Falls Sie dies verwenden AWS CloudFormation, aktualisieren Sie die Stack-Definition, `FunctionVersion:!GetAtt version.Version` sodass der Alias auf die funktionierende Version verweist:

```
alias:
 Type: AWS::Lambda::Alias
 Properties:
 FunctionName: !Ref function
FunctionVersion: !GetAtt version.Version
 Name: BLUE
 ProvisionedConcurrencyConfig:
 ProvisionedConcurrentExecutions: 1
```

Beheben von Aufruf-Problemen in Lambda

Wenn Sie eine Lambda-Funktion aufrufen, validiert Lambda die Anforderung und überprüft die Skalierungskapazität, bevor es das Ereignis an Ihre Funktion oder bei asynchronem Aufruf an die Ereigniswarteschlange sendet. Aufruffehler können durch Probleme mit Anforderungsparametern, Ereignisstruktur, Funktionseinstellungen, Benutzerberechtigungen, Ressourcenberechtigungen oder Grenzwerten verursacht werden.

Wenn Sie Ihre Funktion direkt aufrufen, sehen Sie Aufruffehler in der Antwort von Lambda. Wenn Sie Ihre Funktion asynchron mit einer Ereignisquellen-Zuweisung oder über einen anderen Service aufrufen, finden Sie möglicherweise Fehler in Protokollen, eine Warteschlange für unzustellbare

Nachrichten oder ein Ziel für fehlerhafte Ereignisse. Die Optionen für die Fehlerbehandlung und das Wiederholungsverhalten variieren je nachdem, wie Sie Ihre Funktion aufrufen, und nach der Art des Fehlers.

Eine Liste der Fehlertypen, die der `Invoke`-Vorgang zurückgeben kann, finden Sie unter Aufrufen.

Themen

- Lambda: Funktion bricht während der Init-Phase ab (Sandbox.Timedout)
- IAM: Lambda: InvokeFunction nicht autorisiert
- Lambda: Es konnte kein gültiger Bootstrap gefunden werden (Runtime). InvalidEntrypoint)
- Lambda: Vorgang kann nicht ausgeführt werden ResourceConflictException
- Lambda: Funktion bleibt im Status Pending
- Lambda: Eine Funktion verwendet alle Parallelität
- Allgemein: Funktion kann nicht mit anderen Konten oder Diensten aufgerufen werden
- Allgemein: Funktionsaufruf bleibt im Looping
- Lambda: Alias-Routing mit bereitgestellter Parallelität
- Lambda: Kaltstarts mit bereitgestellter Parallelität
- Lambda: Kaltstarts mit neuen Versionen
- Lambda: Unerwartetes Beenden von Node.js während der Laufzeit (Runtime). NodejsExit)
- EFS: Die Funktion konnte das EFS-Dateisystem nicht mounten
- EFS: Die Funktion konnte keine Verbindung zum EFS-Dateisystem herstellen
- EFS: Die Funktion konnte das EFS-Dateisystem aufgrund eines Timeouts nicht mounten
- Lambda: Lambda hat einen IO-Prozess erkannt, der zu lange dauerte
- Container: Fehler CodeArtifactUserException
- Container: InvalidEntrypoint Fehler

Lambda: Funktion bricht während der Init-Phase ab (Sandbox.Timedout)

Fehler: Zeitüberschreitung der Aufgabe nach 3,00 Sekunden

Wenn die Init-Phase abläuft, initialisiert Lambda die Ausführungsumgebung erneut, indem es die `Init`-Phase erneut durchläuft, wenn die nächste Aufrufanforderung eintrifft. Dies wird als unterdrückte Initialisierung bezeichnet. Wenn Ihre Funktion jedoch mit einer kurzen Timeout-Dauer (in der Regel etwa 3 Sekunden) konfiguriert ist, kann es sein, dass die unterdrückte Initialisierung

während der zugewiesenen Timeout-Dauer nicht abgeschlossen wird, sodass für die `Init`-Phase erneut ein Timeout auftritt. Alternativ wird die unterdrückte Initialisierung abgeschlossen, es bleibt aber nicht genügend Zeit für den Abschluss der Invoke-Phase übrig, was zu einem Timeout für die `Invoke`-Phase führt.

Um Timeout-Fehler zu reduzieren, verwenden Sie eine oder mehrere der folgenden Strategien:

- Erhöhen Sie die Dauer des Funktions-Timeouts: Verlängern Sie das Timeout, um den `Init`- und `Invoke`-Phasen Zeit zu geben, erfolgreich abgeschlossen zu werden.
- Erhöhen Sie die Zuweisung des Funktionsspeichers: Mehr Arbeitsspeicher bedeutet auch eine proportionalere CPU-Zuweisung, wodurch sowohl die `Init`-Phase als auch die `Invoke`-Phasen beschleunigt werden können.
- Optimieren Sie den Code für die Funktionsinitialisierung: Reduzieren Sie die für die Initialisierung benötigte Zeit, um sicherzustellen, dass die `Init`- und `Invoke`-Phase innerhalb der konfigurierten Zeitspanne abgeschlossen werden kann.

IAM: Lambda: InvokeFunction nicht autorisiert

Fehler: Benutzer: arn:aws:iam: :123456789012:user/developer ist nicht berechtigt, Folgendes auszuführen: lambda: on resource: my-function InvokeFunction

Ihr Benutzer oder die Rolle, die Sie annehmen, muss über die Berechtigung zum Aufrufen einer Funktion verfügen. Diese Anforderung gilt auch für Lambda-Funktionen und andere Datenverarbeitungsressourcen, die Funktionen aufrufen. Fügen Sie Ihrem Benutzer die AWSLambdaRolle der AWS verwalteten Richtlinie hinzu, oder fügen Sie eine benutzerdefinierte Richtlinie hinzu, die die Aktion für die Zielfunktion ermöglicht. `lambda:InvokeFunction`

> **Note**
>
> Der Name der IAM-Aktion (`lambda:InvokeFunction`) bezieht sich auf den `Invoke`-Lambda-API-Vorgang.

Weitere Informationen finden Sie unter Verwaltung von Berechtigungen in AWS Lambda.

Lambda: Es konnte kein gültiger Bootstrap gefunden werden (Runtime). InvalidEntrypoint)

Fehler: Es konnten keine gültigen Bootstrap (s) gefunden werden: [/var/task/bootstrap /opt/bootstrap]

Dieser Fehler tritt normalerweise auf, wenn das Stammverzeichnis Ihres Bereitstellungspakets keine ausführbare Datei mit dem Namen `bootstrap` enthält. Wenn Sie beispielsweise eine `provided.al2023`-Funktion über eine ZIP-Datei bereitstellen, muss sich die `bootstrap`-Datei im Stammverzeichnis der ZIP-Datei befinden, nicht in einem Verzeichnis.

Lambda: Vorgang kann nicht ausgeführt werden ResourceConflictException

FehlerResourceConflictException: Der Vorgang kann derzeit nicht ausgeführt werden. Die Funktion befindet sich derzeit in folgendem Zustand: Ausstehend

Wenn Sie zum Zeitpunkt der Erstellung eine Funktion mit einer Virtual Private Cloud (VPC) verbinden, wechselt die Funktion in einen `Pending`-Zustand, während Lambda Elastic-Network-Schnittstellen erstellt. Während dieser Zeit können Sie Ihre Funktion nicht aufrufen oder ändern. Wenn Sie Ihre Funktion nach der Erstellung mit einer VPC verbinden, können Sie sie aufrufen, während das Update aussteht, aber Sie können den Code oder die Konfiguration nicht ändern.

Weitere Informationen finden Sie unter Lambda-Funktionszustände.

Lambda: Funktion bleibt im Status Pending

Fehler: Eine Funktion bleibt mehrere Minuten im *`Pending`*-Zustand.

Wenn eine Funktion länger als sechs Minuten im `Pending`-Zustand ist, rufen Sie eine der folgenden API-Operationen auf, um die Blockierung aufzuheben.

- UpdateFunctionCode
- UpdateFunctionConfiguration
- PublishVersion

Lambda bricht den ausstehenden Vorgang ab und versetzt die Funktion in den `Failed`-Zustand. Sie können dann versuchen, eine weitere Aktualisierung durchzuführen.

Lambda: Eine Funktion verwendet alle Parallelität

Problem: Eine Funktion verwendet die gesamte verfügbare Gleichzeitigkeit, wodurch andere Funktionen gedrosselt werden.

Um die verfügbare Parallelität Ihres AWS Kontos in einer AWS Region in Pools aufzuteilen, verwenden Sie reservierte Parallelität. Durch reservierte Gleichzeitigkeit wird sichergestellt, dass eine Funktion immer auf ihre zugewiesene Gleichzeitigkeit skalieren kann und nicht über die zugewiesene Gleichzeitigkeit hinaus skaliert wird.

Allgemein: Funktion kann nicht mit anderen Konten oder Diensten aufgerufen werden

Problem: Sie können Ihre Funktion direkt aufrufen. Sie wird jedoch nicht ausgeführt, wenn sie von einem anderen Service oder Konto aufgerufen wird.

Sie erteilen anderen Services und Konten die Berechtigung zum Aufrufen einer Funktion in der ressourcenbasierten Richtlinieder Funktion. Wenn sich der Aufrufer in einem anderen Konto befindet, benötigt dieser Benutzer auch die Berechtigung zum Aufrufen von Funktionen.

Allgemein: Funktionsaufruf bleibt im Looping

Problem: Die Funktion wird kontinuierlich in einer Schleife aufgerufen.

Dies ist in der Regel der Fall, wenn Ihre Funktion Ressourcen in demselben AWS Dienst verwaltet, der sie auslöst. Beispielsweise ist es möglich, eine Funktion zu erstellen, die ein Objekt in einem Bucket von Amazon Simple Storage Service (Amazon S3) speichert, der mit einer Benachrichtigung konfiguriert ist, die die Funktion erneut aufruft. Um die Ausführung der Funktion zu stoppen, reduzieren Sie die verfügbare Gleichzeitigkeit auf Null, wodurch alle zukünftigen Aufrufe gedrosselt werden. Identifizieren Sie dann den Codepfad oder Konfigurationsfehler, der den rekursiven Aufruf verursacht hat. Lambda erkennt und stoppt automatisch rekursive Schleifen für einige AWS Dienste und. SDKs Weitere Informationen finden Sie unter the section called “Erkennung rekursiver Schleifen”.

Lambda: Alias-Routing mit bereitgestellter Parallelität

Problem: Bereitgestellte Parallelitäts-Überlauf-Aufrufe während des Alias-Routings.

Lambda verwendet ein einfaches probabilistisches Modell, um den Datenverkehr zwischen den beiden Funktionsversionen zu verteilen. Bei niedrigem Datenverkehr sehen Sie möglicherweise

eine hohe Abweichung zwischen dem konfigurierten und dem tatsächlichen Prozentsatz des Datenverkehrs für jede Version. Wenn Ihre Funktion bereitgestellte Parallelität verwendet, können Sie Überlaufaufrufe durch Konfigurieren einer höheren Anzahl von bereitgestellten Parallelitätsinstances während des aktiven Alias-Routings vermeiden.

Lambda: Kaltstarts mit bereitgestellter Parallelität

Problem: Sie erleben Kaltstarts nach dem Aktivieren der bereitgestellten Parallelität.

Wenn die Anzahl der gleichzeitigen Ausführungen einer Funktion kleiner oder gleich der konfigurierten Ebene der bereitgestellten Parallelität ist, sollte es keine Kaltstarts geben. Um Ihnen zu helfen zu bestätigen, ob die bereitgestellte Parallelität normal funktioniert, gehen Sie wie folgt vor:

- Überprüfen Sie, ob die bereitgestellte Parallelität auf der Funktionsversion oder dem Alias aktiviert ist.

  > **Note**
  >
  > Die bereitgestellte Gleichzeitigkeit kann in der unveröffentlichten Version der Funktion nicht konfiguriert werden.

- Stellen Sie sicher, dass Ihre Auslöser die richtige Funktionsversion oder den richtigen Alias aufrufen. Wenn Sie zum Beispiel Amazon API Gateway verwenden, überprüfen Sie, dass Amazon API die Funktionsversion oder den Alias mit bereitgestellter Parallelität aufruft, nicht $LATEST. Um zu überprüfen, ob die bereitgestellte Parallelität verwendet wird, können Sie die ProvisionedConcurrencyInvocations CloudWatch Amazon-Metrik überprüfen. Ein Wert ungleich Null gibt an, dass die Funktion Aufrufe in initialisierten Ausführungsumgebungen verarbeitet.
- Stellen Sie anhand der Metrik fest, ob Ihre Funktionsparallelität die konfigurierte Stufe der bereitgestellten Parallelität überschreitet. ProvisionedConcurrencySpilloverInvocations CloudWatch Ein Wert ungleich Null gibt an, dass die gesamte bereitgestellte Parallelität verwendet wird und einige Aufrufe mit einem Kaltstart stattgefunden haben.
- Überprüfen Sie Ihre Aufrufhäufigkeit (Anfragen pro Sekunde) Funktionen mit bereitgestellter Parallelität haben eine maximale Rate von 10 Anfragen pro Sekunde pro bereitgestellte Parallelität. Beispielsweise kann eine Funktion, die mit 100 bereitgestellter Parallelität konfiguriert ist, 1.000 Anfragen pro Sekunde bearbeiten. Wenn die Aufrufrate 1.000 Anfragen pro Sekunde übersteigt, kann es zu Kaltstarts kommen.

Lambda: Kaltstarts mit neuen Versionen

Problem: Sie erleben Kaltstarts, wenn Sie neue Versionen Ihrer Funktion bereitstellen.

Wenn Sie einen Funktionsalias aktualisieren, verschiebt Lambda die bereitgestellte Parallelität basierend auf den für den Alias konfigurierten Gewichtungen automatisch auf die neue Version.

Fehler: KMSDisabled Ausnahme: Lambda konnte die Umgebungsvariablen nicht entschlüsseln, da der verwendete KMS-Schlüssel deaktiviert ist. Bitte überprüfen Sie die KMS-Schlüsseleinstellungen der Funktion.

Dieser Fehler kann auftreten, wenn Ihr AWS Key Management Service (AWS KMS) -Schlüssel deaktiviert ist oder wenn die Genehmigung, die Lambda die Verwendung des Schlüssels ermöglicht, widerrufen wird. Wenn die Gewährung fehlt, konfigurieren Sie die Funktion so, dass sie einen anderen Schlüssel verwendet. Weisen Sie dann den benutzerdefinierten Schlüssel neu zu, um die Erteilung neu zu erstellen.

Lambda: Unerwartetes Beenden von Node.js während der Laufzeit (Runtime). NodejsExit)

Problem: Der Lambda-Runtime-Client hat einen unerwarteten Exit-Code für Node.js erkannt.

Dieser Fehler tritt auf, wenn Ihre Funktion beendet wird, bevor alle Versprechen erfüllt sind, z. B. aufgrund eines Codefehlers. Er kann auch auftreten, wenn Node.js einen Deadlock erkennt, der verhindert, dass Promises abgewickelt werden. Dieser Fehler betrifft nur Handler im asynchronen Stil, nicht Handler im Callback-Stil.

Betroffene Laufzeiten: Node.js 18 und höher.

Um dieses Problem zu lösen:

1. Überprüfen Sie Ihren Funktionscode auf ungeklärte Versprechen in asynchronen Handlern.
2. Stellen Sie sicher, dass alle Versprechen ordnungsgemäß erfüllt (gelöst oder abgelehnt) wurden, bevor die Funktion abgeschlossen ist.
3. Überprüfen Sie Ihren Code auf mögliche Rennbedingungen bei asynchronen Vorgängen.

Weitere Informationen zu den Exit-Codes von Node.js und zur Beendigung von Prozessen finden Sie in der Dokumentation zu Node.js.

EFS: Die Funktion konnte das EFS-Dateisystem nicht mounten

Fehler EFSMountFailureException: Die Funktion konnte das EFS-Dateisystem mit dem Access Point arn:aws:elasticfilesystem:us-east- 2:123456789012:access-point/fsap-015cxmplb72b405fd nicht mounten.

Die Mounting-Anforderung der Funktion an das Dateisystem wurde abgelehnt. Überprüfen Sie die Berechtigungen der Funktion und bestätigen Sie, dass das Dateisystem und der Zugriffspunkt vorhanden und einsatzbereit sind.

EFS: Die Funktion konnte keine Verbindung zum EFS-Dateisystem herstellen

Fehler EFSMountConnectivityException: Die Funktion konnte keine Verbindung zum Amazon EFS-Dateisystem mit dem Access Point arn:aws:elasticfilesystem:us-east - 2:123456789012:access-point/fsap-015cxmplb72b405fd herstellen. Überprüfen Sie die Netzwerkkonfiguration und versuchen Sie es erneut.

Die Funktion konnte keine Verbindung zum Dateisystem der Funktion mit dem NFS-Protokoll (TCP-Port 2049) herstellen. Überprüfen Sie die Sicherheitsgruppe und die Routingkonfiguration für die Subnetze der VPC.

Wenn Sie diese Fehler nach der Aktualisierung der VPC-Konfigurationseinstellungen Ihrer Funktion erhalten, versuchen Sie, das Dateisystem auszuhängen und erneut auszuhängen.

EFS: Die Funktion konnte das EFS-Dateisystem aufgrund eines Timeouts nicht mounten

Fehler EFSMountTimeoutException: Die Funktion konnte das EFS-Dateisystem mit dem Access Point {arn:aws:elasticfilesystem:us-east- 2:123456789012:access-point/fsap-015cxmplb72b405fd} aufgrund eines Mount-Timeouts nicht mounten.

Die Funktion konnte eine Verbindung mit dem Dateisystem der Funktion herstellen, aber bei der Mounting-Operation trat eine Zeitüberschreitung auf. Versuchen Sie es nach kurzer Zeit erneut und erwägen Sie, die Nebenläufigkeit der Funktion zu begrenzen, um die Belastung des Dateisystems zu reduzieren.

Lambda: Lambda hat einen IO-Prozess erkannt, der zu lange dauerte

EFSIOException: Diese Funktionsinstanz wurde gestoppt, weil Lambda einen IO-Prozess erkannt hat, der zu lange dauerte.

Bei einem vorherigen Aufruf trat eine Zeitüberschreitung auf und Lambda konnte den Funktions-Handler nicht beenden. Dieses Problem kann auftreten, wenn ein angehängtes Dateisystem keine Burst-Credits hat und der Basisdurchsatz nicht ausreicht. Um den Durchsatz zu erhöhen, können Sie die Größe des Dateisystems erhöhen oder den bereitgestellten Durchsatz verwenden.

Container: Fehler CodeArtifactUserException

Fehler: CodeArtifactUserPendingException Fehlermeldung

Die Optimierung CodeArtifact steht noch aus. Die Funktion wechselt in den aktiven Zustand, wenn Lambda die Optimierung abgeschlossen hat. HTTP-Antwortcode 409.

Fehler: CodeArtifactUserDeletedException Fehlermeldung

Das CodeArtifact soll voraussichtlich gelöscht werden. HTTP-Antwortcode 409.

Fehler: CodeArtifactUserFailedException Fehlermeldung

Lambda konnte den Code nicht optimieren. Sie müssen den Code korrigieren und erneut hochladen. HTTP-Antwortcode 409.

Container: InvalidEntrypoint Fehler

Fehler: Laufzeit. ExitError oder „errorType“: „Runtime. InvalidEntrypoint„

Stellen Sie sicher, dass der ENTRYPOINT zu Ihrem Container-Image den absoluten Pfad als Standort enthält. Stellen Sie außerdem sicher, dass das Image keinen Symlink als ENTRYPOINT enthält.

Fehler: Sie verwenden eine CloudFormation Vorlage und Ihr Container-ENTRYPOINT wird mit einem Nullwert oder einem leeren Wert überschrieben.

Überprüfen Sie die ImageConfigRessource in der Vorlage. CloudFormation Wenn Sie eine `ImageConfig`-Ressource in Ihrer Vorlage deklarieren, müssen Sie nichtleere Werte für alle drei Eigenschaften angeben.

Beheben von Ausführungsproblemen in Lambda

Wenn die Lambda-Laufzeitumgebung Ihren Funktionscode ausführt, wird das Ereignis möglicherweise auf einer Instance der Funktion verarbeitet, auf der schon länger Ereignisse verarbeitet werden, oder muss möglicherweise eine neue Instance initialisiert werden. Fehler können während der Funktionsinitialisierung, wenn Ihr Handler-Code das Ereignis verarbeitet oder wenn Ihre Funktion eine Antwort zurückgibt (oder nicht zurückgibt), auftreten.

Funktionsausführungsfehler können durch Probleme mit Code, Funktionskonfiguration, nachgeschalteten Ressourcen oder Berechtigungen verursacht werden. Wenn Sie Ihre Funktion direkt aufrufen, sehen Sie Funktionsfehler in der Antwort von Lambda. Wenn Sie Ihre Funktion asynchron mit einer Ereignisquellen-Zuweisung oder über einen anderen Service aufrufen, finden Sie möglicherweise Fehler in Protokollen, eine Warteschlange für unzustellbare Nachrichten oder ein Ziel bei Ausfall. Die Optionen für die Fehlerbehandlung und das Wiederholungsverhalten variieren je nachdem, wie Sie Ihre Funktion aufrufen, und nach der Art des Fehlers.

Wenn Ihr Funktionscode oder die Lambda-Laufzeit einen Fehler zurückgibt, ist der Statuscode in der Antwort von Lambda „200 OK“. Das Vorhandensein eines Fehlers in der Antwort wird durch einen Header namens `X-Amz-Function-Error` angezeigt. Statuscodes der Serien 400 und 500 sind für Aufruffehler reserviert.

Themen

- Lambda: Remote-Debugging mit Visual Studio Code
- Lambda: Die Ausführung dauert zu lange
- Lamda: Unerwartete Ereignis-Nutzdaten
- Lambda: Unerwartet große Nutzdaten
- Lambda: Fehler beim Kodieren und Dekodieren von JSON
- Lambda: Protokolle oder Ablaufverfolgungen erscheinen nicht
- Lambda: Nicht alle Protokolle meiner Funktion werden angezeigt
- Lambda: Die Funktion kehrt zurück, bevor die Ausführung beendet ist
- Lamda: Ausführen einer unbeabsichtigten Funktionsversion oder eines Alias
- Lambda: Erkennen von Endlosschleifen
- Allgemein: Nichtverfügbarkeit nachgelagerter Dienste
- AWS SDK: Versionen und Updates

- Python: Bibliotheken werden falsch geladen
- Java: Ihre Funktion braucht länger, um Ereignisse zu verarbeiten, nachdem sie von Java 11 auf Java 17 aktualisiert wurde
- Kafka: Probleme bei der Fehlerbehandlung und beim erneuten Versuch der Konfiguration

Lambda: Remote-Debugging mit Visual Studio Code

Problem: Schwierigkeiten bei der Behebung des komplexen Verhaltens von Lambda-Funktionen in der tatsächlichen Umgebung AWS

Lambda bietet ein Remote-Debugging-Feature über das AWS Toolkit for Visual Studio Code. Anweisungen zur Einrichtung und allgemeinen Nutzung finden Sie unter Remote-Debugging von Lambda-Funktionen mit Visual Studio Code.

Ausführliche Anweisungen zur Fehlerbehebung, zu erweiterten Anwendungsfällen und zur regionalen Verfügbarkeit finden Sie unter Lambda-Funktionen per Fernzugriff im AWS Toolkit for Visual Studio Code Benutzerhandbuch.

Lambda: Die Ausführung dauert zu lange

Problem: Die Ausführung der Funktion dauert zu lange.

Wenn die Ausführung Ihres Codes in Lambda viel länger dauert als auf Ihrem lokalen Computer, kann er durch den der Funktion zur Verfügung stehenden Speicher oder die Rechenleistung eingeschränkt sein. Konfigurieren Sie die Funktion mit zusätzlichem Arbeitsspeicher, um sowohl Arbeitsspeicher als auch CPU zu erhöhen.

Lamda: Unerwartete Ereignis-Nutzdaten

Problem: Funktionsfehler im Zusammenhang mit fehlerhaftem JSON oder unzureichender Datenvalidierung

Alle Lambda-Funktionen erhalten eine Ereignis-Nutzdaten im ersten Parameter des Handlers. Die Ereignis- Nutzdaten sind eine JSON-Struktur, die Arrays und verschachtelte Elemente enthalten kann.

Missgebildetes JSON kann auftreten, wenn es von vorgelagerten Diensten bereitgestellt wird, die kein robustes Verfahren zur Überprüfung von JSON-Strukturen verwenden. Dies tritt auf,

wenn Dienste Textzeichenfolgen miteinander verknüpfen oder Benutzereingaben einbetten, die nicht bereinigt wurden. JSON wird auch häufig für die Weitergabe zwischen Diensten serialisiert. Analysieren Sie JSON-Strukturen immer sowohl als Produzent als auch als Nutzer von JSON, um sicherzustellen, dass die Struktur gültig ist.

Ebenso kann es zu Fehlern kommen, wenn nicht auf Wertebereiche in den Ereignis-Nutzdaten geprüft wird. Eine Funktion, die an den Wert des Handlers übergeben wird:

```
exports.handler = async (event) => {
    let pct = event.taxPct
    let salary = event.salary

    // Calculate % of paycheck for taxes
    return (salary * pct)
}
```

Diese Funktion verwendet ein Gehalt und einen Steuersatz aus den Ereignis-Nutzdaten, um die Berechnung durchzuführen. Der Code überprüft jedoch nicht, ob die Attribute vorhanden sind. Außerdem werden Datentypen nicht überprüft oder Grenzen nicht gewährleistet, z. B. die Sicherstellung, dass der Steuersatz zwischen 0 und 1 liegt. Daher führen Werte außerhalb dieser Grenzen zu unsinnigen Ergebnissen. Ein falscher Typ oder ein fehlendes Attribut führt zu einem Laufzeitfehler.

Erstellen Sie Tests, um sicherzustellen, dass Ihre Funktion größere Nutzdaten verarbeitet. Die maximale Größe für eine Lambda-Ereignis-Nutzlast beträgt 1 MB. Je nach Inhalt können größere Nutzdaten bedeuten, dass mehr Elemente an die Funktion übergeben werden oder dass mehr Binärdaten in ein JSON-Attribut eingebettet sind. In beiden Fällen kann dies zu mehr Verarbeitung für eine Lambda-Funktion führen.

Größere Nutzdaten können auch zu Timeouts führen. Eine Lambda-Funktion verarbeitet beispielsweise einen Datensatz pro 100 ms und hat ein Timeout von 3 Sekunden. Die Verarbeitung ist für 0-29 Elemente in den Nutzdaten erfolgreich. Sobald die Nutzdaten jedoch mehr als 30 Elemente enthalten, bricht die Funktion ab und gibt einen Fehler aus. Um dies zu vermeiden, stellen Sie sicher, dass die Timeouts so eingestellt sind, dass sie die zusätzliche Verarbeitungszeit für die maximale Anzahl erwarteter Elemente berücksichtigen.

Lambda: Unerwartet große Nutzdaten

Problem: Funktionen verursachen Timeouts und Fehler aufgrund großer Nutzdaten

Größere Nutzdaten können zu Timeouts und Fehlern führen. Wir empfehlen, Tests zu erstellen, um sicherzustellen, dass Ihre Funktion die größten zu erwartenden Nutzdaten verarbeiten kann, und zu prüfen, ob das Zeitlimit für die Funktion richtig eingestellt ist.

Darüber hinaus können bestimmte Ereignis-Nutzdaten Verweise auf andere Ressourcen enthalten. Beispielsweise kann eine Lambda-Funktion mit 128 MB Speicher eine Bildverarbeitung für eine JPG-Datei durchführen, die als Objekt in S3 gespeichert ist. Die Funktion funktioniert erwartungsgemäß mit kleineren Bilddateien. Wenn jedoch eine größere JPG-Datei als Eingabe bereitgestellt wird, gibt die Lambda-Funktion einen Fehler aus, da nicht genügend Speicherplatz zur Verfügung steht. Um dies zu vermeiden, sollten die Testfälle Beispiele aus dem oberen Bereich der erwarteten Datengrößen enthalten. Der Code sollte auch die Nutzdatengröße validieren.

Lambda: Fehler beim Kodieren und Dekodieren von JSON

Problem: NoSuchKey Ausnahme beim Parsen von JSON-Eingaben.

Stellen Sie sicher, dass Sie JSON-Attribute korrekt verarbeiten. Für Ereignisse, die von S3 generiert wurden, enthält das Attribut `s3.object.key` beispielsweise einen URL-codierten Objektschlüsselnamen. Viele Funktionen verarbeiten dieses Attribut als Text, um das referenzierte S3-Objekt zu laden:

Example

```
const originalText = await s3.getObject({
  Bucket: event.Records[0].s3.bucket.name,
  Key: event.Records[0].s3.object.key
}).promise()
```

Dieser Code funktioniert mit dem Schlüsselnamen `james.jpg`, löst aber einen `NoSuchKey`-Fehler aus, wenn der Name `james beswick.jpg` lautet. Da die URL-Kodierung Leerzeichen und andere Zeichen in einem Schlüsselnamen konvertiert, müssen Sie sicherstellen, dass Funktionen Schlüssel dekodieren, bevor Sie diese Daten verwenden:

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: Protokolle oder Ablaufverfolgungen erscheinen nicht

Problem: Protokolle werden nicht in CloudWatch Logs angezeigt.

Problem: Spuren werden nicht in angezeigt AWS X-Ray.

Ihre Funktion benötigt die Erlaubnis, CloudWatch Logs und X-Ray aufzurufen. Aktualisieren Sie die Ausführungsrolle, um ihr die Berechtigung zu erteilen. Fügen Sie die folgenden verwalteten Richtlinien hinzu, um Protokolle und Ablaufverfolgung zu aktivieren.

- AWSLambdaBasicExecutionRole
- AWSXRayDaemonWriteAccess

Wenn Sie Ihrer Funktion Berechtigungen hinzufügen, führen Sie auch eine triviale Aktualisierung ihres Codes oder ihrer Konfiguration durch. Dies zwingt ausgeführte Instances Ihrer Funktion, die veraltete Anmeldeinformationen haben, anzuhalten und ersetzt zu werden.

> **Note**
>
> Es kann 5 bis 10 Minuten dauern, bis Protokolle nach einem Funktionsaufruf angezeigt werden.

Lambda: Nicht alle Protokolle meiner Funktion werden angezeigt

Problem: Funktionsprotokolle fehlen in CloudWatch Logs, obwohl meine Berechtigungen korrekt sind

Wenn Ihr System die Kontingentgrenzen für CloudWatch Logs AWS-Konto erreicht, wird die Funktionsprotokollierung CloudWatch gedrosselt. In diesem Fall werden einige der von Ihren Funktionen ausgegebenen Protokolle möglicherweise nicht in CloudWatch den Protokollen angezeigt.

Wenn Ihre Funktion Logs mit einer zu hohen Rate ausgibt, als dass Lambda sie verarbeiten könnte, kann dies auch dazu führen, dass Protokollausgaben nicht in CloudWatch Logs erscheinen. Wenn Lambda Protokolle nicht mit der Geschwindigkeit senden kann, CloudWatch an die Ihre Funktion sie generiert, werden Protokolle gelöscht, um zu verhindern, dass die Ausführung Ihrer Funktion verlangsamt wird. Gehen Sie davon aus, dass gelöschte Protokolle konsistent beobachtet werden, wenn Ihr Protokolldurchsatz MB/s für einen einzelnen Protokollstream 2 überschreitet.

Wenn Ihre Funktion für die Verwendung von Protokollen im JSON-Format konfiguriert ist, versucht Lambda, beim Löschen von Protokollen ein `logsDropped`Ereignis an CloudWatch Logs zu senden. Wenn jedoch die Protokollierung Ihrer Funktion CloudWatch gedrosselt wird, erreicht dieses Ereignis möglicherweise nicht CloudWatch Logs, sodass Sie nicht immer einen Datensatz sehen, wenn Lambda Logs löscht.

Gehen Sie wie folgt vor, um zu überprüfen, ob Ihr System die Kontingentgrenzen für CloudWatch Logs erreicht AWS-Konto hat:

1. Öffnen Sie die Service Quotas-Konsole.
2. Wählen Sie im Navigationsbereich AWS -Services.
3. Suchen Sie in der AWS Serviceliste nach Amazon CloudWatch Logs.
4. Wählen Sie in der Liste mit den Service Quotas die Kontingente `CreateLogGroup throttle limit in transactions per second`, `CreateLogStream throttle limit in transactions per second` und `PutLogEvents throttle limit in transactions per second` aus, um die Auslastung anzuzeigen.

Sie können auch CloudWatch Alarme einrichten, die Sie benachrichtigen, wenn Ihre Kontoauslastung ein von Ihnen für diese Kontingente festgelegtes Limit überschreitet. Weitere Informationen finden Sie unter Einen CloudWatch Alarm auf der Grundlage eines statischen Schwellenwerts erstellen.

Wenn die standardmäßigen Kontingentgrenzen für CloudWatch Logs für Ihren Anwendungsfall nicht ausreichen, können Sie eine Erhöhung des Kontingents beantragen.

Lambda: Die Funktion kehrt zurück, bevor die Ausführung beendet ist

Problem: (Node.js) Rückgabe der Funktion erfolgt, bevor Code ausgeführt wird

Viele Bibliotheken, einschließlich des AWS SDK, arbeiten asynchron. Wenn Sie einen Netzwerkaufruf tätigen oder einen anderen Vorgang ausführen, für den auf eine Antwort gewartet werden muss, geben Bibliotheken ein Objekt zurück, das als Zusage bezeichnet wird und mit dem der Status der Operation im Hintergrund nachverfolgt wird.

Um zu warten, bis die Zusage in eine Antwort aufgelöst wird, verwenden Sie das Schlüsselwort `await`. Dadurch wird verhindert, dass der Handler-Code ausgeführt wird, bis die Zusage in ein Objekt aufgelöst wird, das die Antwort enthält. Wenn Sie die Daten aus der Antwort in Ihrem Code nicht verwenden müssen, können Sie die Zusage direkt an die Laufzeit zurückgeben.

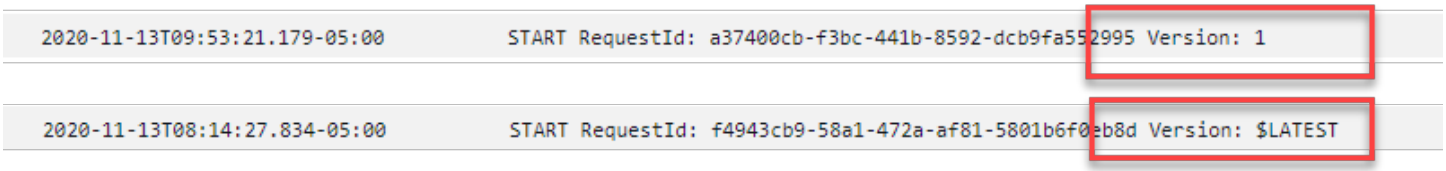
Einige Bibliotheken geben keine Zusagen zurück, können aber in Code verpackt werden, der dies tut. Weitere Informationen finden Sie unter Lambda-Funktionshandler in Node.js definieren.

Lamda: Ausführen einer unbeabsichtigten Funktionsversion oder eines Alias

Problem: Funktionsversion oder Alias wurde nicht aufgerufen

Wenn Sie neue Lambda-Funktionen in der Konsole veröffentlichen oder verwenden AWS SAM, wird die neueste Codeversion durch `$LATEST` dargestellt. Standardmäßig zielen Aufrufe, die keine Version oder keinen Alias angeben, automatisch auf die `$LATEST`-Version Ihres Funktionscodes ab.

Wenn Sie bestimmte Funktionsversionen oder Aliase verwenden, handelt es sich dabei um unveränderliche veröffentlichte Versionen einer Funktion zusätzlich zu `$LATEST`. Stellen Sie bei der Problembehebung dieser Funktionen zunächst fest, ob der Aufrufer die beabsichtigte Version oder den Alias aufgerufen hat. Überprüfen Sie dazu Ihre Funktionsprotokolle. Die Version der aufgerufenen Funktion wird immer in der START-Protokollzeile angezeigt:

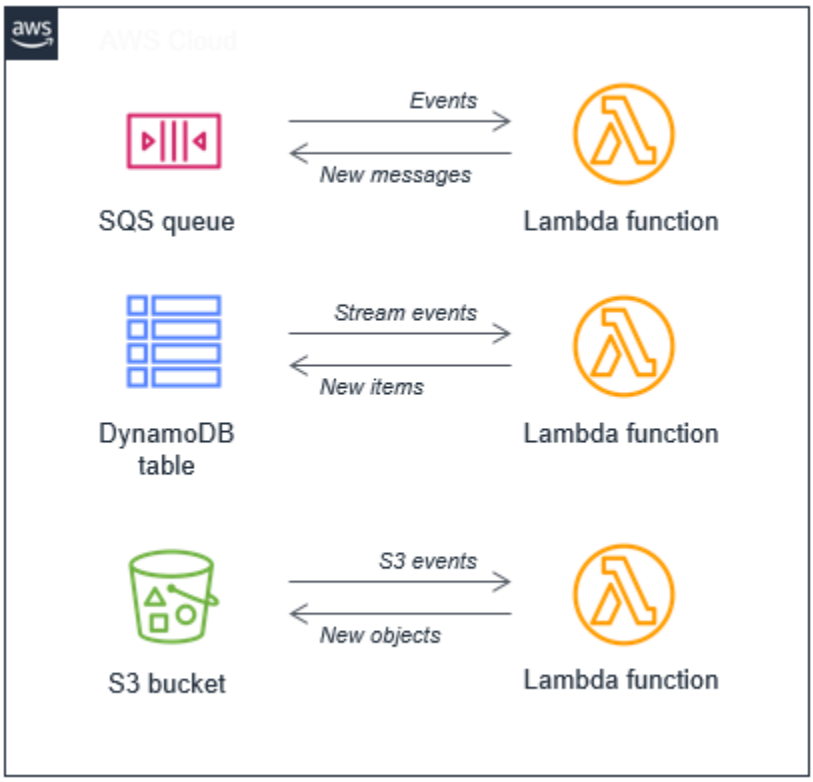


Lambda: Erkennen von Endlosschleifen

Problem: Endlosschleifenmuster im Zusammenhang mit Lambda-Funktionen

Es gibt zwei Arten von Endlosschleifen in Lambda-Funktionen. Die erste befindet sich innerhalb der Funktion selbst und wird durch eine Schleife verursacht, die niemals beendet wird. Der Aufruf endet erst, wenn die Funktion ein Timeout verursacht. Sie können diese identifizieren, indem Sie Timeouts überwachen und dann das Schleifenverhalten beheben.

Die zweite Art von Schleife besteht zwischen Lambda-Funktionen und anderen AWS Ressourcen. Diese treten auf, wenn ein Ereignis aus einer Ressource wie einem S3-Bucket eine Lambda-Funktion aufruft, die dann mit derselben Quellressource interagiert, um ein weiteres Ereignis auszulösen. Dadurch wird die Funktion erneut aufgerufen, wodurch eine weitere Interaktion mit demselben S3-Bucket erstellt wird, und so weiter. Diese Arten von Schleifen können durch eine Reihe verschiedener AWS Ereignisquellen verursacht werden, darunter Amazon SQS SQS-Warteschlangen und DynamoDB-Tabellen. Sie können diese Muster mithilfe der rekursiven Schleifenerkennung identifizieren.



Sie können diese Schleifen vermeiden, indem Sie sicherstellen, dass Lambda-Funktionen in Ressourcen schreiben, die nicht mit der verbrauchenden Ressource identisch sind. Wenn Sie Daten wieder auf der verbrauchenden Ressource veröffentlichen müssen, stellen Sie sicher, dass die neuen Daten nicht dasselbe Ereignis auslösen. Verwenden Sie alternativ die Ereignisfilterung. Hier sind beispielsweise zwei Lösungsvorschläge für Endlosschleifen mit S3- und DynamoDB-Ressourcen:

- Wenn Sie in denselben S3-Bucket zurückschreiben, verwenden Sie ein anderes Präfix oder Suffix als beim Ereignisauslöser.
- Wenn Sie Elemente in dieselbe DynamoDB-Tabelle schreiben, fügen Sie ein Attribut hinzu, nach dem eine verbrauchende Lambda-Funktion filtern kann. Wenn Lambda das Attribut findet, führt dies nicht zu einem weiteren Aufruf.

Allgemein: Nichtverfügbarkeit nachgelagerter Dienste

Problem: Die nachgelagerten Dienste, auf die Ihre Lambda-Funktion angewiesen ist, sind nicht verfügbar

Stellen Sie bei Lambda-Funktionen, die Drittanbieter-Endpunkte oder andere nachgelagerte Ressourcen aufrufen, sicher, dass sie Servicefehler und Timeouts behandeln können. Diese nachgelagerten Ressourcen können unterschiedliche Reaktionszeiten haben oder aufgrund von Serviceunterbrechungen nicht verfügbar sein. Je nach Implementierung können diese nachgelagerten Fehler als Lambda-Timeouts oder Ausnahmen erscheinen, wenn die Fehlerantwort des Dienstes nicht im Code der Funktion behandelt wird.

Implementieren Sie eine geeignete Fehlerbehandlung und Wiederholungslogik, wenn eine Funktion von einem nachgelagerten Dienst abhängt, z. B. von einem API-Aufruf. Für kritische Dienste sollte die Lambda-Funktion Metriken oder Protokolle veröffentlichen. CloudWatch Wenn beispielsweise eine Zahlungs-API eines Drittanbieters nicht mehr verfügbar ist, kann die Lambda-Funktion diese Informationen protokollieren. Sie können dann CloudWatch Alarme einrichten, um Benachrichtigungen zu diesen Fehlern zu senden.

Da Lambda schnell skalieren kann, können nachgelagerte Dienste, die nicht Serverless sind, Schwierigkeiten haben, Datenverkehrsspitzen zu bewältigen. Es gibt drei gängige Ansätze, um damit umzugehen:

- Caching: Erwägen Sie, die Ergebnisse von Werten, die von Drittanbieterdiensten zurückgegeben werden, zwischenzuspeichern, wenn diese sich nicht häufig ändern. Sie können diese Werte in einer globalen Variablen in Ihrer Funktion oder einem anderen Dienst speichern. Beispielsweise könnten die Ergebnisse einer Produktlistenabfrage von einer Amazon RDS-Instance innerhalb der Funktion für einen bestimmten Zeitraum gespeichert werden, um redundante Abfragen zu vermeiden.
- Warteschlangen: Fügen Sie beim Speichern oder Aktualisieren von Daten eine Amazon-SQS-Warteschlange zwischen der Lambda-Funktion und der Ressource hinzu. In der Warteschlange werden Daten dauerhaft gespeichert, während der nachgelagerte Dienst Nachrichten verarbeitet.
- Proxys: Wenn in der Regel langlebige Verbindungen verwendet werden, z. B. für Amazon-RDS-Instances, verwenden Sie eine Proxyebene, um diese Verbindungen zu bündeln und wiederzuverwenden. Für relationale Datenbanken ist Amazon-RDS-Proxy ein Service, der dazu beitragen soll, die Skalierbarkeit und Stabilität von Lambda–basierten Anwendungen zu verbessern.

AWS SDK: Versionen und Updates

Problem: Das in der Runtime enthaltene AWS SDK ist nicht die neueste Version

Problem: Das in der Runtime enthaltene AWS SDK wird automatisch aktualisiert

Zu den Laufzeiten für interpretierte Sprachen gehört eine Version des AWS SDK. Lambda aktualisiert diese Laufzeiten regelmäßig, um die neueste SDK-Version zu verwenden. Die in Ihrer Laufzeit enthaltene SDK-Version finden Sie in den folgenden Abschnitten:

- In der Laufzeit enthaltene SDK-Versionen (Node.js)
- In der Laufzeit enthaltene SDK-Versionen (Python)
- In der Laufzeit enthaltene SDK-Versionen (Ruby)

Um eine neuere Version des AWS SDK zu verwenden oder Ihre Funktionen an eine bestimmte Version zu binden, können Sie die Bibliothek mit Ihrem Funktionscode bündeln oder eine Lambda-Schicht erstellen. Weitere Informationen zum Erstellen eines Bereitstellungspakets mit Abhängigkeiten finden Sie in den folgenden Themen:

Node.js

Bereitstellen von Node.js Lambda-Funktionen mit ZIP-Dateiarchiven

Python

Arbeiten mit ZIP-Dateiarchiven und Python-Lambda-Funktionen

Ruby

Bereitstellen von Ruby-Lambda-Funktionen mit ZIP-Dateiarchiven

Java

Bereitstellen von Java-Lambda-Funktionen mit ZIP- oder JAR-Dateiarchiven

Go

Bereitstellen von Lambda-Go-Funktionen mit ZIP-Dateiarchiven

C#

Erstellen und Bereitstellen von C#-Lambda-Funktionen mit ZIP-Dateiarchiven

PowerShell

PowerShell-Lambda-Funktionen mit .zip-Dateiarchiven bereitstellen

Python: Bibliotheken werden falsch geladen

Problem: (Python) Einige Bibliotheken werden nicht korrekt aus dem Bereitstellungspaket geladen

Bibliotheken mit Erweiterungsmodulen, die in C oder C ++ geschrieben sind, müssen in einer Umgebung mit derselben Prozessorarchitektur wie Lambda (Amazon Linux) kompiliert werden. Weitere Informationen finden Sie unter Arbeiten mit ZIP-Dateiarchiven und Python-Lambda-Funktionen.

Java: Ihre Funktion braucht länger, um Ereignisse zu verarbeiten, nachdem sie von Java 11 auf Java 17 aktualisiert wurde

Problem: (Java) Ihre Funktion braucht länger, um Ereignisse zu verarbeiten, nachdem sie von Java 11 auf Java 17 aktualisiert wurde

Optimieren Sie Ihren Compiler mithilfe des `JAVA_TOOL_OPTIONS`-Parameters. Lambda-Laufzeiten für Java 17 und spätere Java-Versionen ändern die Standard-Compileroptionen. Die Änderung verbessert die Kaltstartzeiten für kurzlebige Funktionen, aber das bisherige Verhalten ist besser für rechenintensive, länger laufende Funktionen geeignet. Setzen Sie `JAVA_TOOL_OPTIONS` auf `-XX:-TieredCompilation`, um zum Verhalten von Java 11 zurückzukehren. Weitere Informationen zum Parameter `JAVA_TOOL_OPTIONS` erhalten Sie unter the section called “Die `JAVA_TOOL_OPTIONS`-Umgebungsvariable verstehen”.

Kafka: Probleme bei der Fehlerbehandlung und beim erneuten Versuch der Konfiguration

Problem: Bei der Zuordnung von Kafka-Ereignisquellen können Einstellungen für Wiederholungsversuche oder Ziele für den Fall eines Fehlers nicht konfiguriert werden

Kafka-Wiederholungskonfigurationen und Ziele für den Fall eines Fehlers sind nur für Zuordnungen von Ereignisquellen verfügbar, bei denen der Bereitstellungsmodus aktiviert ist. Stellen Sie sicher, dass Sie Ihre Konfiguration vorgenommen haben, `ProvisionedPollerConfig` bevor Sie versuchen, `MinimumPollers` Wiederholungskonfigurationen festzulegen.

Häufige Konfigurationsfehler:

- Unendliche Wiederholungen mit Bisect Batch — Sie können die Option nicht aktivieren`BisectBatchOnFunctionError`, wenn die Option auf -1 (unendlich) gesetzt `MaximumRetryAttempts` ist. Legen Sie ein begrenztes Limit für Wiederholungsversuche fest oder deaktivieren Sie Bisect Batch.
- Rekursion zum selben Thema — Das Kafka-Zielthema bei einem Fehler darf nicht mit einem Ihrer Quellthemen identisch sein. Wählen Sie einen anderen Themennamen für Ihr aktuelles Thema.

- Ungültiges Kafka-Zielformat — Verwenden Sie das `kafka://<topic-name>` Format, wenn Sie ein Kafka-Thema als Ziel für den Fall eines Fehlers angeben.
- kafka: WriteData Berechtigungsprobleme — Stellen Sie sicher, dass Ihre Ausführungsrolle über `kafka-cluster:WriteData` Berechtigungen für das Zielthema verfügt. Thema existiert nicht, Timeout-Ausnahmen oder Probleme mit der Schreib-API-Drosselung erfordern möglicherweise eine Erhöhung der Kontolimits.

Behebung von Problemen mit der Zuordnung von Ereignisquellen in Lambda

Probleme in Lambda, die sich auf die Zuordnung von Ereignisquellen beziehen, können komplexer sein, da sie das Debuggen mehrerer Services beinhalten. Darüber hinaus kann das Verhalten der Ereignisquelle je nach der verwendeten Ereignisquelle unterschiedlich sein. In diesem Abschnitt werden häufig auftretende Probleme im Zusammenhang mit der Zuordnung von Ereignisquellen aufgeführt und Hinweise zu deren Identifizierung und Behebung gegeben.

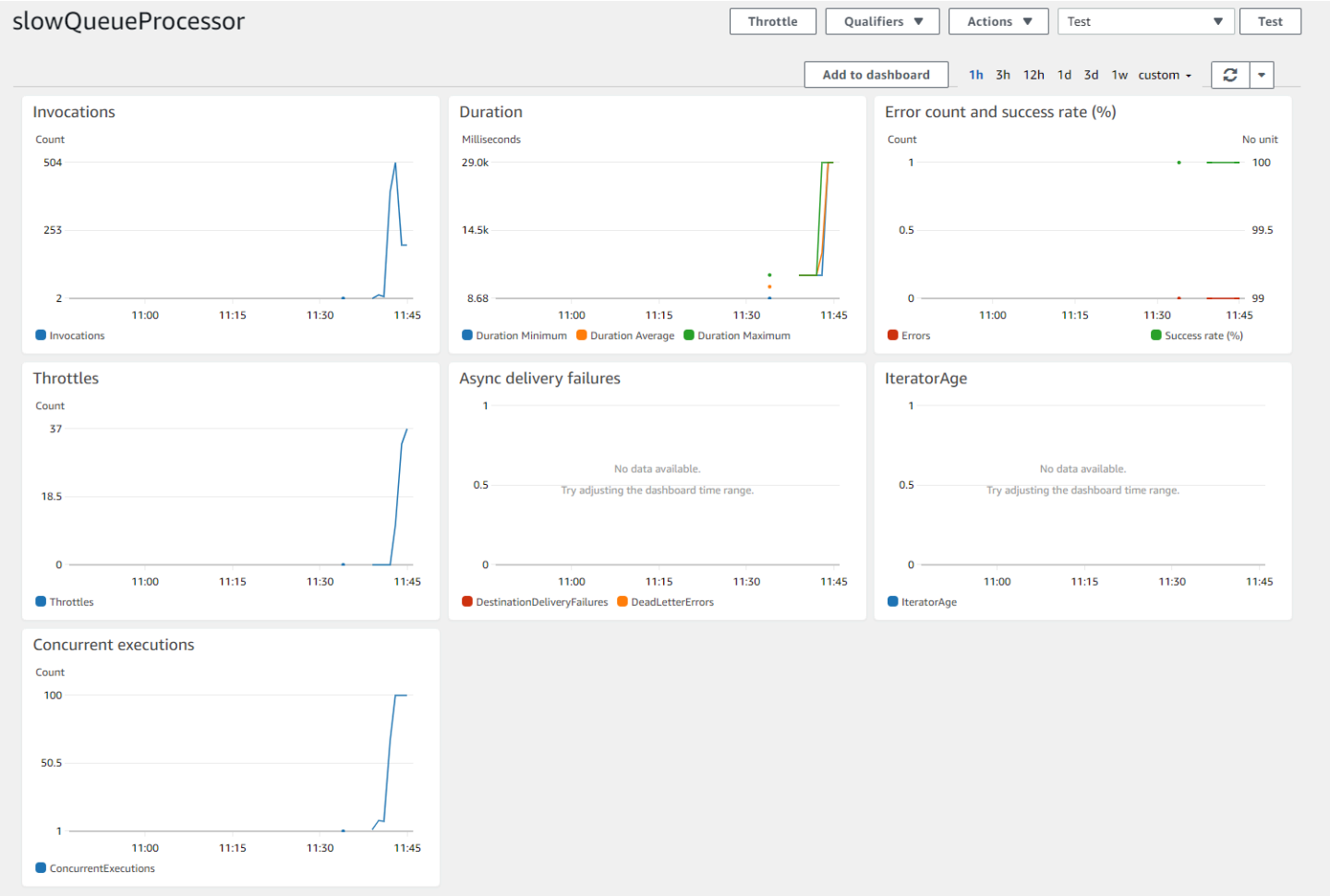
> **Note**
>
> In diesem Abschnitt wird zur Veranschaulichung eine Amazon-SQS-Ereignisquelle verwendet, aber die Prinzipien gelten auch für andere Zuordnungen von Ereignisquellen, die Nachrichten für Lambda-Funktionen in die Warteschlange stellen.

Drosselung erkennen und verwalten

In Lambda erfolgt die Drosselung, wenn das Gleichzeitigkeitslimit Ihrer Funktion oder Ihres Kontos erreicht ist. Betrachten Sie das folgende Beispiel, in dem es eine Lambda-Funktion gibt, um Nachrichten aus einer Amazon-SQS-Warteschlange zu lesen. Diese Lambda-Funktion simuliert 30-Sekunden-Aufrufe und hat eine Batchgröße von 1. Das bedeutet, dass die Funktion alle 30 Sekunden nur eine Nachricht verarbeitet:

```
const doWork = (ms) => new Promise(resolve => setTimeout(resolve, ms))

exports.handler = async (event) => {
    await doWork(30000)

}
```

Bei einer so langen Aufrufzeit kommen Nachrichten schneller in der Warteschlange an als sie verarbeitet werden. Wenn die nicht reservierte Gleichzeitigkeit Ihres Kontos 100 beträgt, skaliert Lambda auf 100 gleichzeitige Ausführungen hoch, und dann erfolgt eine Drosselung. Sie können dieses Muster in den CloudWatch-Metriken für die Funktion sehen:



Die CloudWatch-Metriken für die Funktion zeigen keine Fehler, aber das Diagramm Gleichzeitige Ausführungen zeigt, dass die maximale Gleichzeitigkeit von 100 erreicht ist. Aus diesem Grund zeigt das Diagramm Drosselungen, dass die Drosselung bereits besteht.

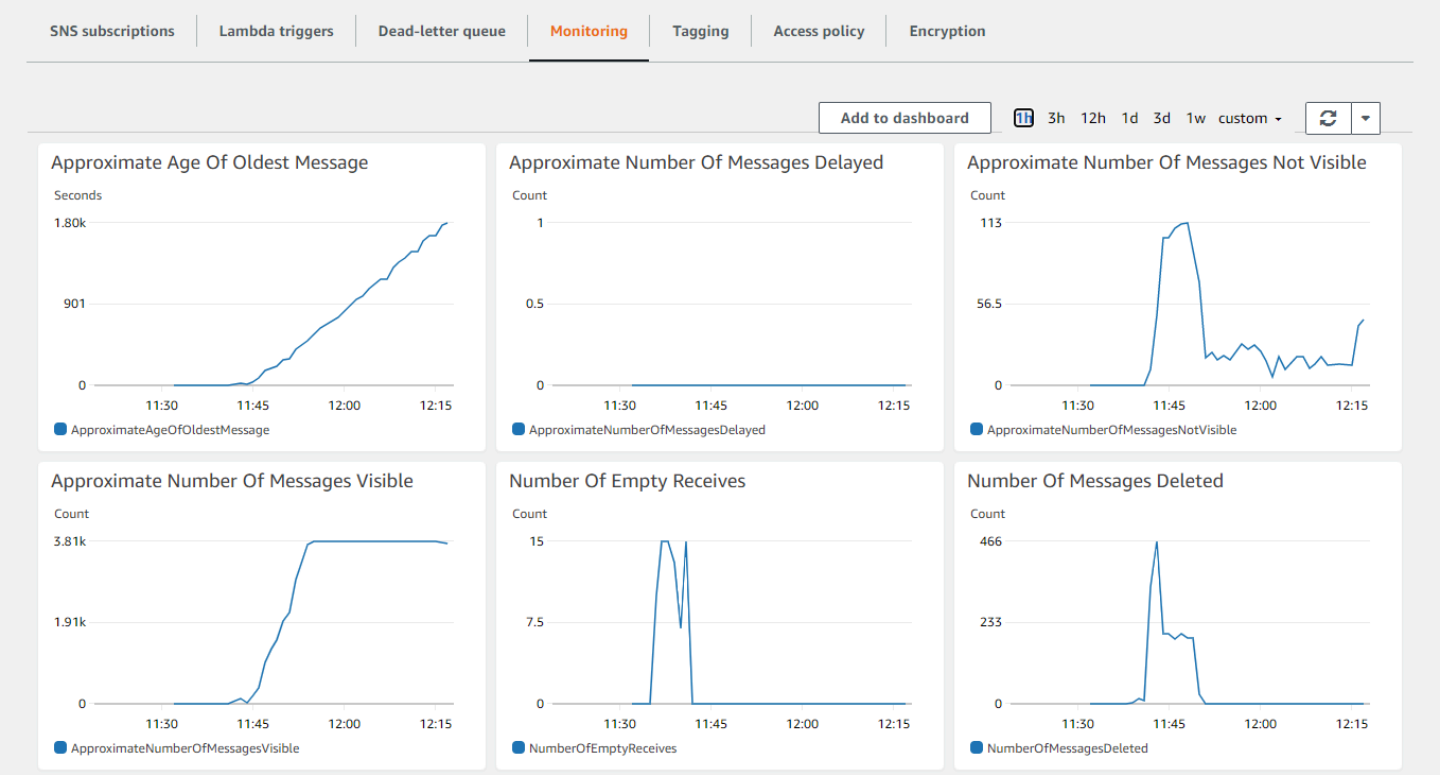
Sie können Drosselungen mit CloudWatch-Alarmen erkennen und einen Alarm einstellen, sobald die Drosselungsmetrik für eine Funktion größer als 0 ist. Nachdem Sie das Drosselungsproblem identifiziert haben, stehen Ihnen einige Lösungsmöglichkeiten zur Verfügung:

- Beantragen Sie eine Erhöhung der Gleichzeitigkeit von AWS-Support in dieser Region.
- Identifizieren Sie Leistungsprobleme in der Funktion, um die Verarbeitungsgeschwindigkeit und damit den Durchsatz zu verbessern.

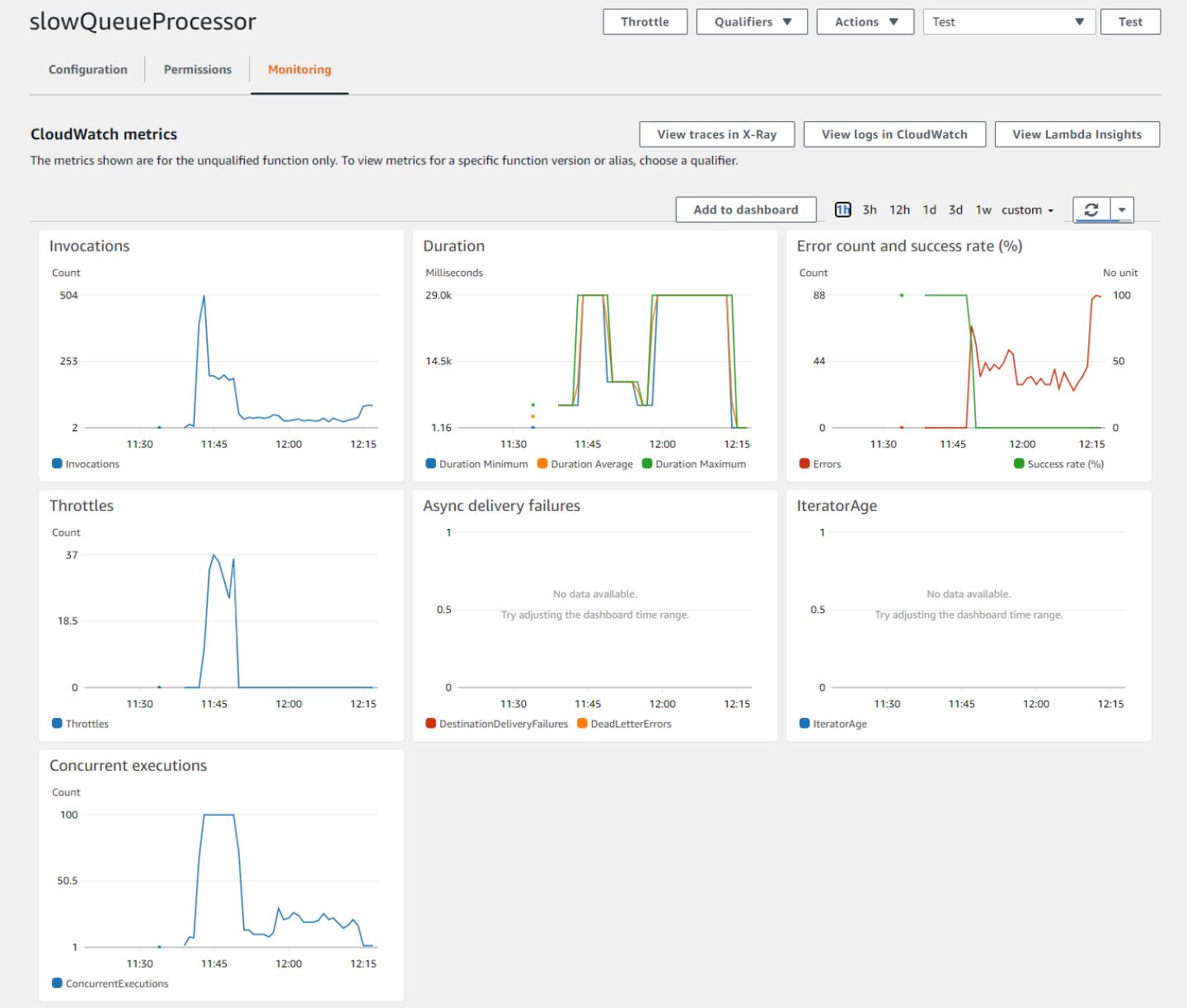
- Erhöhen Sie die Batchgröße der Funktion, sodass bei jedem Aufruf mehr Nachrichten verarbeitet werden.

Fehler in der Verarbeitungsfunktion

Wenn die Verarbeitungsfunktion Fehler ausgibt, gibt Lambda die Nachrichten an die SQS-Warteschlange zurück. Lambda verhindert, dass Ihre Funktion skaliert wird, um Skalierungsfehler zu vermeiden. Die folgenden SQS-Metriken in CloudWatch weisen auf ein Problem mit der Warteschlangenverarbeitung hin:

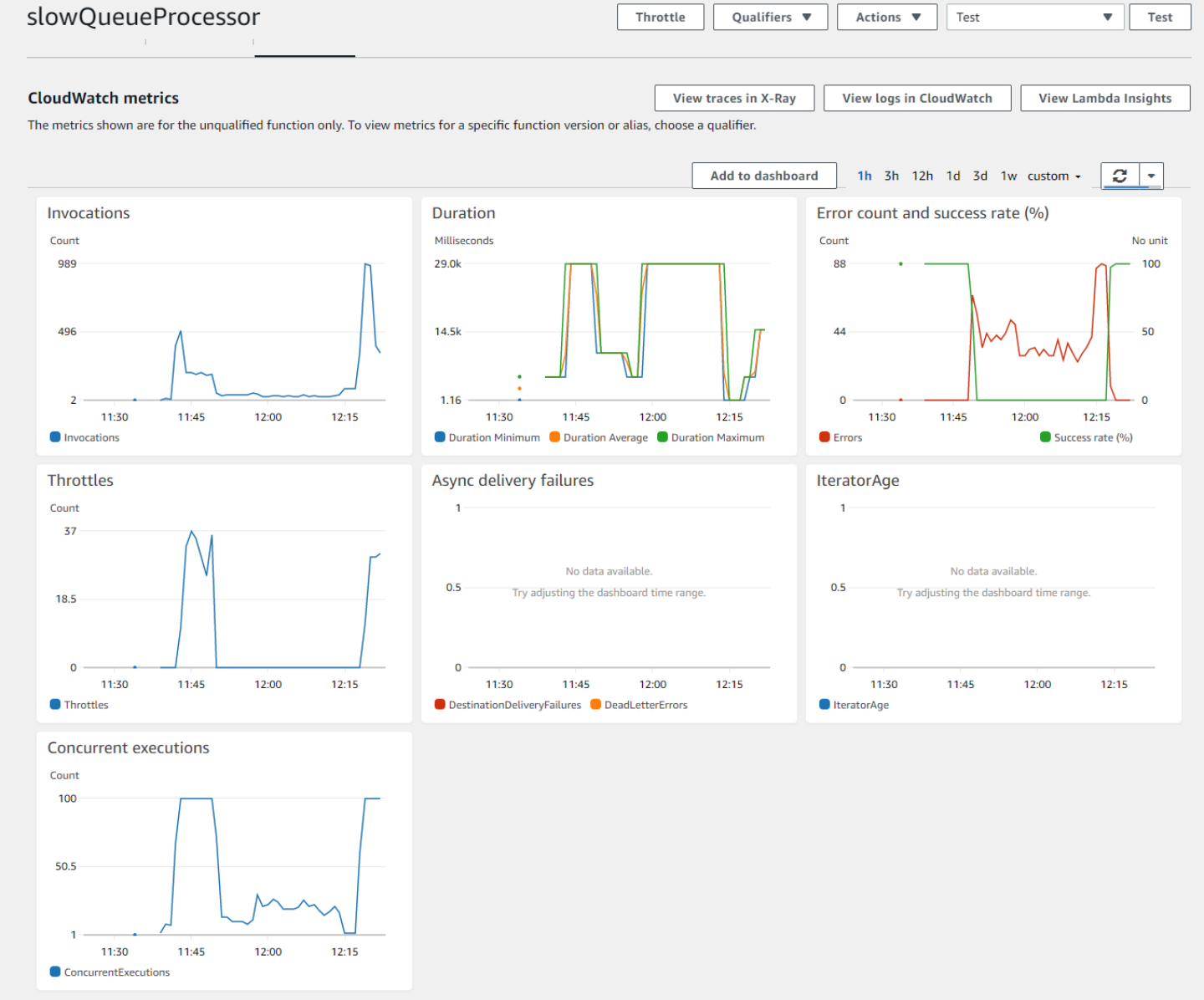


Insbesondere nehmen sowohl das Alter der ältesten Nachricht als auch die Anzahl der sichtbaren Nachrichten zu, während keine Nachrichten gelöscht werden. Die Warteschlange wächst weiter, aber Nachrichten werden nicht verarbeitet. Die CloudWatch-Metriken für die verarbeitende Lambda-Funktion zeigen ebenfalls, dass ein Problem vorliegt:



Die Metrik für die Anzahl der Fehler ist ungleich Null und nimmt weiter zu, während die Anzahl der gleichzeitigen Ausführungen zurückgegangen ist und die Drosselung eingestellt wurde. Dies deutet darauf hin, dass Lambda die Hochskalierung Ihrer Funktion aufgrund von Fehlern eingestellt hat. Die CloudWatch Logs für die Funktion enthalten Details zur Art des Fehlers.

Sie können dieses Problem lösen, indem Sie die Funktion identifizieren, die den Fehler verursacht hat und dann den Fehler finden und beheben. Nachdem Sie den Fehler behoben und den neuen Funktionscode bereitgestellt haben, sollten die CloudWatch-Metriken die Wiederherstellung der Verarbeitung zeigen:



Hier sinkt die Metrik Anzahl der Fehler auf Null und die Metrik Erfolgsquote kehrt auf 100 % zurück. Lambda beginnt erneut mit der Hochskalierung der Funktion, wie im Diagramm Gleichzeitige Ausführungen dargestellt.

Identifizierung und Behandlung von Gegendruck

Wenn ein Ereignisproduzent konsistent Nachrichten für eine SQS-Warteschlange schneller generiert, als eine Lambda-Funktion diese verarbeiten kann, entsteht Gegendruck. In diesem Fall sollte die SQS-Überwachung das Alter der ältesten Nachricht linear ansteigen lassen, zusammen mit der ungefähren Anzahl der sichtbaren Nachrichten. Mithilfe von CloudWatch-Alarmen können Sie Gegendruck in Warteschlangen erkennen.

Die Schritte zur Behebung von Gegendruck hängen von Ihrem Workload ab. Wenn das Hauptziel darin besteht, die Verarbeitungskapazität und den Durchsatz durch die Lambda-Funktion zu erhöhen, haben Sie einige Möglichkeiten:

- Beantragen Sie bei AWS-Support eine Erhöhung der Gleichzeitigkeit in der betreffenden Region.
- Erhöhen Sie die Batchgröße der Funktion, sodass bei jedem Aufruf mehr Nachrichten verarbeitet werden.

Beheben von Netzwerkproblemen in Lambda

Standardmäßig führt Lambda Ihre Funktionen in einer internen Virtual Private Cloud (VPC) mit Konnektivität zu AWS -Services und dem Internet aus. Um auf lokale Netzwerkressourcen zuzugreifen, können Sie Ihre Funktion so konfigurieren, dass sie sich mit einer VPC in Ihrem Konto verbindet. Wenn Sie dieses Feature verwenden, verwalten Sie den Internetzugriff und die Netzwerkkonnektivität der Funktion mit Ressourcen von Amazon Virtual Private Cloud (Amazon VPC).

Netzwerkverbindungsfehler können auf Probleme mit der Routing-Konfiguration Ihrer VPC, den Sicherheitsgruppenregeln, AWS Identity and Access Management (IAM) -Rollenberechtigungen oder der Network Address Translation (NAT) oder auf die Verfügbarkeit von Ressourcen wie IP-Adressen oder Netzwerkschnittstellen zurückzuführen sein. Je nach Problem wird möglicherweise ein bestimmter Fehler oder Timeout angezeigt, wenn eine Anfrage ihr Ziel nicht erreichen kann.

Themen

- VPC: Funktion verliert Internetzugriff oder läuft ab
- VPC: TCP- oder UDP-Verbindung bricht zeitweise ab
- VPC: Die Funktion benötigt Zugriff, AWS-Services ohne das Internet zu nutzen
- VPC: Grenzwert für Elastic-Network-Schnittstellen erreicht
- EC2: Elastische Netzwerkschnittstelle mit dem Typ „Lambda“
- DNS: Verbindung zu Hosts mit UNKNOWNHOSTEXCEPTION schlägt fehl

VPC: Funktion verliert Internetzugriff oder läuft ab

Problem: Ihre Lambda-Funktion verliert Internetzugriff nach der Verbindung mit einer VPC.

Fehler: Fehler: Verbindung ETIMEDOUT 176.32.98.189:443

Fehler: Fehler: Zeitüberschreitung der Aufgabe nach 10,00 Sekunden

FehlerReadTimeoutError: Timeout beim Lesen. (Timeout beim Lesen = 15)

Wenn Sie eine Funktion mit einer VPC verbinden, werden alle ausgehenden Anforderungen über die VPC geleitet. Um eine Verbindung zum Internet herzustellen, konfigurieren Sie Ihre VPC so, dass ausgehender Datenverkehr aus dem Subnetz der Funktion an ein NAT-Gateway in einem öffentlichen Subnetz gesendet wird. Weitere Informationen und VPC-Beispielkonfigurationen finden Sie unter the section called “Internetzugriff für VPC-Funktionen”.

Wenn einige Ihrer TCP-Verbindungen ein Timeout aufweisen, überprüfen Sie the section called “VPC: TCP- oder UDP-Verbindung bricht zeitweise ab”, ob Ihr Subnetz eine Netzwerkzugriffskontrollliste (NACL) verwendet. Andernfalls liegt dies wahrscheinlich an einer Paketfragmentierung. Lambda-Funktionen können eingehende fragmentierte TCP-Anfragen nicht verarbeiten, da Lambda keine IP-Fragmentierung für TCP oder ICMP unterstützt.

VPC: TCP- oder UDP-Verbindung bricht zeitweise ab

> **Note**
>
> Dieses Problem tritt nur auf, wenn Ihr Subnetz eine Zugriffssteuerungsliste (ACL) für Netzwerke verwendet. Für Lambda ist kein Netzwerk ACLs erforderlich, um eine Verbindung zu Ihren Subnetzen herzustellen.

Problem: Lambda verliert zeitweise die Verbindung zu Ihren VPC-Subnetzen, für die Sie eine Netzwerk-Zugriffssteuerungsliste (ACL) konfiguriert haben.

Für VPC-fähige Lambda-Funktionen AWS erstellt es eine Hyperplane ENIs im Kundenkonto und verwendet kurzlebige Ports, um Lambda mit der VPC des Kunden `1024 65535` zu verbinden. Wenn Sie das Netzwerk ACLs im Zielsubnetz verwenden, müssen Sie den Portbereich sowohl für TCP als auch für UDP zulassen. `1024 65535` Wenn dieser vollständige Portbereich nicht zugelassen wird, kann es zu zeitweiligen Verbindungsausfällen kommen.

VPC: Die Funktion benötigt Zugriff, AWS-Services ohne das Internet zu nutzen

Problem: Ihre Lambda-Funktion benötigt Zugriff auf, AWS-Services ohne das Internet zu verwenden.

Verwenden Sie VPC-Endpunkte, um eine Funktion AWS-Services von einem privaten Subnetz ohne Internetzugang aus zu verbinden.

VPC: Grenzwert für Elastic-Network-Schnittstellen erreicht

Fehler ENILimitReachedException: Das elastic network interface Network-Schnittstellenlimit wurde für die VPC der Funktion erreicht.

Wenn Sie eine Funktion mit einer Lambda-VPC verbinden, erstellt Lambda für jede Kombination aus Subnetz und Sicherheitsgruppe, die der Funktion zugeordnet ist, eine Elastic-Network-Schnittstelle. Das Standard-Servicekontingent beträgt 250 Netzwerkschnittstellen pro VPC. Zum Anfordern einer Erhöhung für ein Kontingent können Sie die Service-Quotas-Konsole verwenden.

EC2: Elastische Netzwerkschnittstelle mit dem Typ „Lambda“

Fehlercode: Client. OperationNotPermitted

Fehlermeldung: Die Sicherheitsgruppe kann für diesen Schnittstellentyp nicht geändert werden

Sie erhalten diesen Fehler, wenn Sie versuchen, eine Elastic-Network-Schnittstelle (ENI) zu ändern, das von Lambda verwaltet wird. Das `ModifyNetworkInterfaceAttribute` ist nicht in der Lambda-API für Aktualisierungsvorgänge auf elastischen Netzwerkschnittstellen enthalten, die von Lambda erstellt wurden.

DNS: Verbindung zu Hosts mit UNKNOWNHOSTEXCEPTION schlägt fehl

Fehlermeldung: UNKNOWNHOSTEXCEPTION

Lambda-Funktionen unterstützen maximal 20 gleichzeitige TCP-Verbindungen für die DNS-Auflösung. Ihre Funktion hat möglicherweise dieses Limit ausgeschöpft. Die meisten DNS-Anfragen werden über UDP gestellt. Wenn Ihre Funktion nur UDP-DNS-Verbindungen herstellt, ist es unwahrscheinlich, dass dies Ihr Problem ist. Bevor Sie also Ihren DNS-Verkehr eingehend untersuchen, vergewissern Sie sich, dass Ihre DNS-Infrastruktur ordnungsgemäß konfiguriert und in Ordnung ist und dass sich Ihre Lambda-Funktion auf einen in DNS angegebenen Host bezieht.

Wenn Sie Ihr Problem im Zusammenhang mit der maximalen TCP-Verbindung diagnostizieren, beachten Sie, dass Sie keine Erhöhung dieses Limits beantragen können. Wenn Ihre Lambda-Funktion aufgrund großer DNS-Payloads auf TCP DNS zurückgreift, überprüfen Sie, ob Ihre Lösung Bibliotheken verwendet, die EDNS unterstützen. Weitere Informationen über EDNS finden Sie in der

Norm RFC 6891. Wenn Ihre DNS-Nutzdaten die EDNS-Maximalgröße ständig überschreiten, kann Ihre Lösung das TCP DNS-Limit dennoch ausschöpfen.

Lambda-Beispielanwendungen

Das GitHub-Repository für dieses Handbuch enthält Beispielanwendungen, welche die Verwendung verschiedener Sprachen und AWS-Services demonstrieren. Jede Beispielanwendung enthält Skripts für die einfache Bereitstellung und Bereinigung und unterstützende Ressourcen.

Node.js

Lambda-Beispielanwendungen in Node.js

- blank-nodejs – Eine Node.js-Funktion, welche die Verwendung von Protokollierung, Umgebungsvariablen, AWS X-Ray-Ablaufverfolgung, Ebenen, Einheitentests und des AWS SDK aufzeigt.
- nodejs-apig – Eine Funktion mit einem öffentlichen API-Endpunkt, die ein Ereignis aus API Gateway verarbeitet und eine HTTP-Antwort zurückgibt.

Python

Lambda-Beispielanwendungen in Python

- blank-python AWS X-Ray Eine Python-Funktion, die die Verwendung von Protokollierung, Umgebungsvariablen, AWS-Ablaufverfolgung, Ebenen, Einheitentests und des SDK aufzeigt.

Ruby

Lambda-Beispielanwendungen in Ruby

- blank-ruby – Eine Ruby-Funktion, die die Verwendung von Protokollierung, Umgebungsvariablen, AWS X-Ray-Ablaufverfolgung, Ebenen, Einheitentests und des AWS SDK aufzeigt.
- Ruby-Codebeispiele für AWS Lambda – In Ruby geschriebene Codebeispiele, die zeigen, wie mit AWS Lambda interagiert wird.

Java

Lambda-Beispielanwendungen in Java

- example-java – Eine Java-Funktion, die zeigt, wie Sie Lambda zur Bearbeitung von Aufträgen verwenden können. Diese Funktion veranschaulicht, wie Sie ein benutzerdefiniertes Eingabeereignisobjekt definieren und deserialisieren, das AWS SDK verwenden und Protokolle ausgeben.
- Java-Basis – Eine Sammlung minimaler Java-Funktionen mit Einheitentests und variabler Protokollierungskonfiguration.
- Java-Ereignisse – Eine Sammlung von Java-Funktionen, die Grundcode für den Umgang mit Ereignissen aus verschiedenen Services wie Amazon API Gateway, Amazon SQS und Amazon Kinesis enthalten. Diese Funktionen verwenden die neueste Version der aws-lambda-java-events-Bibliothek (3.0.0 und neuer). Für diese Beispiele ist das AWS SDK nicht als Abhängigkeit erforderlich.
- s3-java – Eine Java-Funktion die Benachrichtigungsereignisse aus Amazon S3 verarbeitet und die Java Class Library (JCL) verwendet, um Miniaturansichten aus hochgeladenen Image-Dateien zu erstellen.
- layer-java – Eine Java-Funktion, die veranschaulicht, wie Sie mit einer Lambda-Ebene Abhängigkeiten getrennt von Ihrem Kernfunktionscode packen können.

Ausführen beliebter Java-Frameworks auf Lambda

- spring-cloud-function-samples – Ein Beispiel, das zeigt, wie das Spring Cloud Function-Framework verwendet wird, um AWS-Lambda-Funktionen zu erstellen.
- Serverless Spring Boot-Anwendungsdemo – Ein Beispiel, das zeigt, wie eine typische Spring Boot-Anwendung in einer verwalteten Java-Laufzeit mit und ohne SnapStart oder als natives GraalVM-Image mit einer benutzerdefinierten Laufzeit eingerichtet wird.
- Serverless Micronaut-Anwendungsdemo – Ein Beispiel, das zeigt, wie Micronaut in einer verwalteten Java-Laufzeit mit und ohne SnapStart oder als natives GraalVM-Image mit einer benutzerdefinierten Laufzeit verwendet wird. Erfahren Sie mehr in den Micronaut/Lambda-Leitfäden.
- Serverless Quarkus-Anwendungsdemo – Ein Beispiel, das zeigt, wie Quarkus in einer verwalteten Java-Laufzeit mit und ohne SnapStart oder als natives GraalVM-Image mit einer benutzerdefinierten Laufzeit verwendet werden kann. Weitere Informationen finden Sie im Quarkus/Lambda-Handbuch und im Quarkus/SnapStart-Handbuch.

Go

Lambda stellt die folgenden Beispielanwendungen für die Go-Laufzeit bereit:

Lambda-Beispielanwendungen in Go

- go-al2: Eine Hello World-Funktion, die die öffentliche IP-Adresse zurückgibt. Diese App verwendet die benutzerdefinierte Laufzeit `provided.al2`.
- blank-go Eine Go-Funktion, welche die Verwendung der Lambda-Go-Bibliotheken, Protokollierung, Umgebungsvariablen und des AWS SDK aufzeigt. Diese App verwendet die Laufzeit `go1.x`.

C#

Lambda-Beispielanwendungen in C#

- blank-csharp AWS X-Ray Eine C #-Funktion, welche die Verwendung der Lambda-NET-Bibliotheken, Protokollierung, Umgebungsvariablen, -Nachverfolgung, Einheitentests und des AWS SDK aufzeigt.
- blank-csharp-with-layer – Eine C#-Funktion, die die .NET-CLI verwendet, um eine Ebene zu erstellen, die die Abhängigkeiten der Funktion bündelt.
- ec2-spot – Eine Funktion, die Spot-Instance-Anforderungen in Amazon EC2 verwaltet.

PowerShell

Lambda stellt die folgenden Beispielanwendungen für PowerShell bereit:

- blank-powershell – AWS Eine PowerShell-Funktion, die die Verwendung von Protokollierung, Umgebungsvariablen und -SDK aufzeigt.

Um eine Beispielanwendung bereitzustellen, befolgen Sie die Anweisungen in der README-Datei.

Lambda mit einem AWS SDK verwenden

AWS Software Development Kits (SDKs) sind für viele beliebte Programmiersprachen verfügbar. Jedes SDK bietet eine API, Codebeispiele und Dokumentation, die es Entwicklern erleichtern, Anwendungen in ihrer bevorzugten Sprache zu erstellen.

SDK-Dokumentation	Codebeispiele
AWS SDK für C++	AWS SDK für C++ Codebeispiele
AWS CLI	AWS CLI Code-Beispiele
AWS SDK für Go	AWS SDK für Go Code-Beispiele
AWS SDK für Java	AWS SDK für Java Code-Beispiele
AWS SDK für JavaScript	AWS SDK für JavaScript Code-Beispiele
AWS SDK für Kotlin	AWS SDK für Kotlin Code-Beispiele
AWS SDK für .NET	AWS SDK für .NET Code-Beispiele
AWS SDK für PHP	AWS SDK für PHP Code-Beispiele
AWS -Tools für PowerShell	AWS -Tools für PowerShell Code-Beispiele
AWS SDK für Python (Boto3)	AWS SDK für Python (Boto3) Code-Beispiele
AWS SDK für Ruby	AWS SDK für Ruby Code-Beispiele
AWS SDK für Rust	AWS SDK für Rust Code-Beispiele
AWS SDK für SAP ABAP	AWS SDK für SAP ABAP Code-Beispiele
AWS SDK für Swift	AWS SDK für Swift Code-Beispiele

Für Beispiele, die sich speziell auf Lambda beziehen, siehe Codebeispiele für Lambda mit AWS SDKs.

Beispiel für die Verfügbarkeit

Sie können nicht finden, was Sie brauchen? Fordern Sie ein Codebeispiel an, indem Sie unten den Link Provide feedback (Feedback geben) auswählen.

Codebeispiele für Lambda mit AWS SDKs

Die folgenden Codebeispiele zeigen, wie Lambda mit einem AWS Software Development Kit (SDK) verwendet wird.

Bei Grundlagen handelt es sich um Codebeispiele, die Ihnen zeigen, wie Sie die wesentlichen Vorgänge innerhalb eines Services ausführen.

Aktionen sind Codeauszüge aus größeren Programmen und müssen im Kontext ausgeführt werden. Während Aktionen Ihnen zeigen, wie Sie einzelne Service-Funktionen aufrufen, können Sie Aktionen im Kontext der zugehörigen Szenarien anzeigen.

Szenarien sind Codebeispiele, die Ihnen zeigen, wie Sie bestimmte Aufgaben ausführen, indem Sie mehrere Funktionen innerhalb eines Service aufrufen oder mit anderen AWS-Services kombinieren.

AWS Community-Beiträge sind Beispiele, die von mehreren AWS Teams erstellt wurden und verwaltet werden. Um Feedback zu geben, verwenden Sie den Mechanismus, der in den verlinkten Repositorys bereitgestellt wird.

Eine vollständige Liste der AWS SDK-Entwicklerhandbücher und Codebeispiele finden Sie unterLambda mit einem AWS SDK verwenden. Dieses Thema enthält auch Informationen zu den ersten Schritten und Details zu früheren SDK-Versionen.

Codebeispiele

- Grundlegende Beispiele für die Verwendung von Lambda AWS SDKs
  - Hallo Lambda
  - Lernen Sie die Grundlagen von Lambda mit einem SDK AWS
  - Aktionen für Lambda unter Verwendung AWS SDKs
    - Verwendung von CreateAlias mit einer CLI
    - Verwendung CreateFunction mit einem AWS SDK oder CLI
    - Verwendung von DeleteAlias mit einer CLI
    - Verwendung DeleteFunction mit einem AWS SDK oder CLI
    - Verwendung von DeleteFunctionConcurrency mit einer CLI
    - Verwendung von DeleteProvisionedConcurrencyConfig mit einer CLI
    - Verwendung von GetAccountSettings mit einer CLI
    - Verwendung von GetAlias mit einer CLI

- Verwendung GetFunction mit einem AWS SDK oder CLI
- Verwendung von GetFunctionConcurrency mit einer CLI
- Verwendung von GetFunctionConfiguration mit einer CLI
- Verwendung von GetPolicy mit einer CLI
- Verwendung von GetProvisionedConcurrencyConfig mit einer CLI
- Verwendung Invoke mit einem AWS SDK oder CLI
- Verwendung ListFunctions mit einem AWS SDK oder CLI
- Verwendung von ListProvisionedConcurrencyConfigs mit einer CLI
- Verwendung von ListTags mit einer CLI
- Verwendung von ListVersionsByFunction mit einer CLI
- Verwendung von PublishVersion mit einer CLI
- Verwendung von PutFunctionConcurrency mit einer CLI
- Verwendung von PutProvisionedConcurrencyConfig mit einer CLI
- Verwendung von RemovePermission mit einer CLI
- Verwendung von TagResource mit einer CLI
- Verwendung von UntagResource mit einer CLI
- Verwendung von UpdateAlias mit einer CLI
- Verwendung UpdateFunctionCode mit einem AWS SDK oder CLI
- Verwendung UpdateFunctionConfiguration mit einem AWS SDK oder CLI

- Szenarien für die Verwendung von Lambda AWS SDKs
  - Bestätigen Sie bekannte Amazon Cognito Cognito-Benutzer automatisch mit einer Lambda-Funktion mithilfe eines SDK AWS
  - Automatisches Migrieren bekannter Amazon Cognito Cognito-Benutzer mit einer Lambda-Funktion mithilfe eines SDK AWS
  - Erstellen einer API-Gateway-REST-API zur Verfolgung von COVID-19-Daten
  - Leihbibliothek-REST-API erstellen
  - Erstellen einer Messenger-Anwendung mit Step Functions
  - Eine Anwendung für Foto-Asset-Management erstellen, mit der Benutzer Fotos mithilfe von Labels verwalten können
  - Erstellen einer Websocket-Chat-Anwendung mit API Gateway
  - Erstellen einer Anwendung, die Kundenfeedback analysiert und Audio generiert

- Aufrufen einer Lambda-Funktion von einem Browser aus
- Transformieren von Daten für Ihre Anwendung mit S3 Object Lambda
- Verwenden von API Gateway zum Aufrufen einer Lambda-Funktion
- Verwenden von Step Functions, um Lambda-Funktionen aufzurufen
- Verwendung geplanter Ereignisse zum Aufrufen einer Lambda-Funktion
- Verwenden der Amazon Neptune-API zur Entwicklung einer Lambda-Funktion, die Graphdaten abfragt
- Schreiben Sie benutzerdefinierte Aktivitätsdaten mit einer Lambda-Funktion nach der Amazon Cognito Cognito-Benutzerauthentifizierung mithilfe eines SDK AWS

- Serverless-Beispiele für Lambda
  - Herstellen einer Verbindung mit einer Amazon-RDS-Datenbank in einer Lambda-Funktion
  - Aufrufen einer Lambda-Funktion über einen Kinesis-Auslöser
  - Aufrufen einer Lambda-Funktion über einen DynamoDB-Auslöser
  - Aufrufen einer Lambda-Funktion über einen Amazon DocumentDB-Auslöser
  - Aufrufen einer Lambda-Funktion über einen Amazon-MSK-Auslöser
  - Aufrufen einer Lambda-Funktion über einen Amazon-S3-Auslöser
  - Eine Lambda-Funktion über einen Amazon-SNS-Trigger aufrufen
  - Aufrufen einer Lambda-Funktion über einen Amazon-SQS-Auslöser
  - Melden von Batch-Elementfehlern für Lambda-Funktionen mit einem Kinesis-Auslöser
  - Melden von Batch-Elementfehlern für Lambda-Funktionen mit einem DynamoDB-Auslöser
  - Melden von Batch-Elementfehlern für Lambda-Funktionen mit einem Amazon-SQS-Auslöser
- AWS Community-Beiträge für Lambda
  - Erstellen und Testen einer Serverless-Anwendung

Grundlegende Beispiele für die Verwendung von Lambda AWS SDKs

Die folgenden Codebeispiele zeigen, wie die Grundlagen von AWS Lambda with AWS SDKs verwendet werden.

Beispiele

- Hallo Lambda

- Lernen Sie die Grundlagen von Lambda mit einem SDK AWS
- Aktionen für Lambda unter Verwendung AWS SDKs
  - Verwendung von CreateAlias mit einer CLI
  - Verwendung CreateFunction mit einem AWS SDK oder CLI
  - Verwendung von DeleteAlias mit einer CLI
  - Verwendung DeleteFunction mit einem AWS SDK oder CLI
  - Verwendung von DeleteFunctionConcurrency mit einer CLI
  - Verwendung von DeleteProvisionedConcurrencyConfig mit einer CLI
  - Verwendung von GetAccountSettings mit einer CLI
  - Verwendung von GetAlias mit einer CLI
  - Verwendung GetFunction mit einem AWS SDK oder CLI
  - Verwendung von GetFunctionConcurrency mit einer CLI
  - Verwendung von GetFunctionConfiguration mit einer CLI
  - Verwendung von GetPolicy mit einer CLI
  - Verwendung von GetProvisionedConcurrencyConfig mit einer CLI
  - Verwendung Invoke mit einem AWS SDK oder CLI
  - Verwendung ListFunctions mit einem AWS SDK oder CLI
  - Verwendung von ListProvisionedConcurrencyConfigs mit einer CLI
  - Verwendung von ListTags mit einer CLI
  - Verwendung von ListVersionsByFunction mit einer CLI
  - Verwendung von PublishVersion mit einer CLI
  - Verwendung von PutFunctionConcurrency mit einer CLI
  - Verwendung von PutProvisionedConcurrencyConfig mit einer CLI
  - Verwendung von RemovePermission mit einer CLI
  - Verwendung von TagResource mit einer CLI
  - Verwendung von UntagResource mit einer CLI
  - Verwendung von UpdateAlias mit einer CLI
  - Verwendung UpdateFunctionCode mit einem AWS SDK oder CLI
  - Verwendung UpdateFunctionConfiguration mit einem AWS SDK oder CLI

Hallo Lambda

Die folgenden Codebeispiele zeigen, wie Sie mit der Verwendung von Lambda beginnen.

.NET

SDK für .NET

> **Note**
>
> Es gibt noch mehr dazu GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
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}");
        });
    }
}
```

- Einzelheiten zur API finden Sie ListFunctionsin der AWS SDK für .NET API-Referenz.

C++

SDK für C++

> **Note**
>
> Es gibt noch mehr dazu GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

Code für die CMake Datei 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 für die Quelldatei „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;
}
```

- Einzelheiten zur API finden Sie ListFunctionsunter AWS SDK für C++ API-Referenz.

Go

SDK für Go V2

> **Note**
>
> Es gibt noch mehr dazu GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
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)
  }
 }
}
```

- Einzelheiten zur API finden Sie ListFunctionsin der AWS SDK für Go API-Referenz.

Java

SDK für Java 2.x

> **Note**
>
> Es gibt noch mehr dazu GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
    /**
     * 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);
        }
    }
```

- Einzelheiten zur API finden Sie ListFunctionsin der AWS SDK for Java 2.x API-Referenz.

JavaScript

SDK für JavaScript (v3)

> **Note**
>
> Es gibt noch mehr dazu GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
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;
};
```

- Einzelheiten zur API finden Sie ListFunctionsin der AWS SDK für JavaScript API-Referenz.

Python

SDK für Python (Boto3)

> **Note**
>
> Es gibt noch mehr dazu GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
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()
```

- Einzelheiten zur API finden Sie ListFunctionsin AWS SDK for Python (Boto3) API Reference.

Ruby

SDK für Ruby

> **Note**
>
> Es gibt noch mehr dazu. GitHub Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
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
```

- Einzelheiten zur API finden Sie ListFunctionsin der AWS SDK für Ruby API-Referenz.

Eine vollständige Liste der AWS SDK-Entwicklerhandbücher und Codebeispiele finden Sie unterLambda mit einem AWS SDK verwenden. Dieses Thema enthält auch Informationen zu den ersten Schritten und Details zu früheren SDK-Versionen.

Lernen Sie die Grundlagen von Lambda mit einem SDK AWS

Die folgenden Code-Beispiele veranschaulichen Folgendes:

- Erstellen Sie eine IAM-Rolle und eine Lambda-Funktion und laden Sie den Handlercode hoch.
- Rufen Sie die Funktion mit einem einzigen Parameter auf und erhalten Sie Ergebnisse.
- Aktualisieren Sie den Funktionscode und konfigurieren Sie mit einer Umgebungsvariablen.
- Rufen Sie die Funktion mit neuen Parametern auf und erhalten Sie Ergebnisse. Zeigt das zurückgegebene Ausführungsprotokoll an.
- Listen Sie die Funktionen für Ihr Konto auf und bereinigen Sie dann die Ressourcen.

Weitere Informationen zur Verwendung von Lambda finden Sie unter Erstellen einer Lambda-Funktion mit der Konsole.

.NET

SDK für .NET

> **Note**
>
> Es gibt noch mehr dazu GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

Erstellen Sie Methoden, die Lambda-Aktionen ausführen.

```
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;
    }


}
```

Erstellen Sie eine Funktion, die das Szenario ausführt.

```
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();
    }
}
```

Definieren Sie einen Lambda-Handler, der eine Zahl inkrementiert.

```
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;
        }
    }
}
```

Definieren Sie einen zweiten Lambda-Handler, der arithmetische Operationen ausführt.

```
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;
    }
}
```

- Weitere API-Informationen finden Sie in den folgenden Themen der AWS SDK für .NET - API-Referenz.
  - CreateFunction
  - DeleteFunction
  - GetFunction
  - Invoke
  - ListFunctions
  - UpdateFunctionCode
  - UpdateFunctionConfiguration

C++

SDK für C++

> **Note**
>
> Es gibt noch mehr GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
//! 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;
}
```

- Weitere API-Informationen finden Sie in den folgenden Themen der AWS SDK für C++ -API-Referenz.
  - CreateFunction
  - DeleteFunction
  - GetFunction
  - Invoke
  - ListFunctions
  - UpdateFunctionCode
  - UpdateFunctionConfiguration

Go

SDK für Go V2

> **Note**
>
> Es gibt noch mehr GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

Erstellen Sie ein interaktives Szenario, das Ihnen zeigt, wie Sie mit Lambda-Funktionen loslegen können.

```
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
}
```

Erstellen einer Struktur, die die einzelnen Lambda-Aktionen umschließt.

```
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"`
}
```

Definieren Sie einen Lambda-Handler, der eine Zahl inkrementiert.

```
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
```

Definieren Sie einen zweiten Lambda-Handler, der arithmetische Operationen ausführt.

```
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
```

- Weitere API-Informationen finden Sie in den folgenden Themen der AWS SDK für Go -API-Referenz.
  - CreateFunction
  - DeleteFunction
  - GetFunction
  - Invoke
  - ListFunctions
  - UpdateFunctionCode
  - UpdateFunctionConfiguration

Java

SDK für Java 2.x

> **Note**
>
> Es gibt noch mehr GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
/*
 *  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);
        }
    }
}
```

- Weitere API-Informationen finden Sie in den folgenden Themen der AWS SDK for Java 2.x - API-Referenz.
  - CreateFunction
  - DeleteFunction
  - GetFunction
  - Invoke
  - ListFunctions
  - UpdateFunctionCode
  - UpdateFunctionConfiguration

JavaScript

SDK für JavaScript (v3)

> **Note**
>
> Es gibt noch mehr dazu GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

Erstellen Sie eine AWS Identity and Access Management (IAM-) Rolle, die Lambda die Berechtigung erteilt, in Protokolle zu schreiben.

```
    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);
};
```

Erstellen Sie eine Lambda-Funktion und laden Sie Handlercode hoch.

```
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);
};
```

Rufen Sie die Funktion mit einem einzigen Parameter auf und erhalten Sie Ergebnisse.

```
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 };
};
```

Aktualisieren Sie den Funktionscode und konfigurieren Sie seine Lambda-Umgebung mit einer Umgebungsvariablen.

```
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;
};
```

Listen Sie die Funktionen für Ihr Konto auf.

```
const listFunctions = () => {
  const client = new LambdaClient({});
  const command = new ListFunctionsCommand({});

  return client.send(command);
};
```

Löschen Sie die IAM-Rolle für Ihre Lambda-Funktion.

```
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);
};
```

- Weitere API-Informationen finden Sie in den folgenden Themen der AWS SDK für JavaScript -API-Referenz.
  - CreateFunction
  - DeleteFunction
  - GetFunction
  - Invoke
  - ListFunctions

- UpdateFunctionCode
- UpdateFunctionConfiguration

Kotlin

SDK für Kotlin

> **Note**
>
> Es gibt noch mehr dazu. GitHub Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
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")
    }
}
```

- Weitere API-Informationen finden Sie in den folgenden Themen der API-Referenz zum AWS -SDK für Kotlin.
  - CreateFunction
  - DeleteFunction
  - GetFunction
  - Invoke
  - ListFunctions
  - UpdateFunctionCode

- UpdateFunctionConfiguration

PHP

SDK für PHP

> **Note**
>
> Es gibt noch mehr GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
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";
    }
}
```

- Weitere API-Informationen finden Sie in den folgenden Themen der AWS SDK für PHP - API-Referenz.
  - CreateFunction
  - DeleteFunction
  - GetFunction
  - Invoke
  - ListFunctions
  - UpdateFunctionCode
  - UpdateFunctionConfiguration

Python

SDK für Python (Boto3)

> **Note**
>
> Es gibt noch mehr GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

Definieren Sie einen Lambda-Handler, der eine Zahl inkrementiert.

```
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
```

Definieren Sie einen zweiten Lambda-Handler, der arithmetische Operationen ausführt.

```
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
```

Erstellen Sie Funktionen, die Lambda-Aktionen umschließen.

```
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
```

Erstellen Sie eine Funktion, die das Szenario ausführt.

```
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!")
```

- Weitere API-Informationen finden Sie in den folgenden Themen der API-Referenz zum AWS SDK für Python (Boto3).
  - CreateFunction
  - DeleteFunction
  - GetFunction
  - Invoke
  - ListFunctions
  - UpdateFunctionCode
  - UpdateFunctionConfiguration

Ruby

SDK für Ruby

> **Note**
>
> Es gibt noch mehr GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

Richten Sie die erforderlichen IAM-Berechtigungen für eine Lambda-Funktion ein, die Protokolle schreiben kann.

```
  # 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
```

Definieren Sie einen Lambda-Handler, der eine als Aufrufparameter bereitgestellte Zahl inkrementiert.

```
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
```

Komprimieren (ZIP) Sie Ihre Lambda-Funktion in ein Bereitstellungspaket.

```
  # 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
```

Erstellen Sie eine neue Lambda-Funktion.

```
  # 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
```

Rufen Sie Ihre Lambda-Funktion mit optionalen Laufzeitparametern auf.

```
 # 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
```

Aktualisieren Sie die Konfiguration Ihrer Lambda-Funktion, um eine neue Umgebungsvariable einzufügen.

```
 # 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
```

Aktualisieren Sie den Code Ihrer Lambda-Funktion mit einem anderen Bereitstellungspaket, das anderen Code enthält.

```
 # 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
```

Listen Sie alle vorhandenen Lambda-Funktionen mithilfe des eingebauten Paginators auf.

```
 # 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
```

Löschen Sie eine bestimmte Lambda-Funktion.

```
 # 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
```

- Weitere API-Informationen finden Sie in den folgenden Themen der AWS SDK für Ruby - API-Referenz.
  - CreateFunction
  - DeleteFunction
  - GetFunction
  - Invoke
  - ListFunctions
  - UpdateFunctionCode
  - UpdateFunctionConfiguration

Rust

SDK für Rust

> **Note**
>
> Es gibt noch mehr dazu GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

Die Datei „Cargo.toml“ mit in diesem Szenario verwendeten Abhängigkeiten.

```
[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"
```

Eine Sammlung von Hilfsprogrammen, die das Aufrufen von Lambda für dieses Szenario optimieren. Diese Datei ist „src/ations.rs“ in der Kiste.

```
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(),
        );
    }
```

```
}
```

Eine Binärdatei, um das Szenario vom Anfang bis Ende auszuführen, wobei Befehlszeilen-Flags verwendet werden, um einige Verhaltensweisen zu steuern. Diese Datei ist src/bin/scenario .rs in der Kiste.

```
/*
## 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");
    }
}
```

- Weitere API-Informationen finden Sie in den folgenden Themen der API-Referenz zum AWS -SDK für Rust.
  - CreateFunction
  - DeleteFunction
  - GetFunction
  - Invoke
  - ListFunctions
  - UpdateFunctionCode
  - UpdateFunctionConfiguration

SAP ABAP

SDK für SAP ABAP

> **Note**
>
> Es gibt noch mehr dazu. GitHub Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
   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.
```

- Weitere API-Informationen finden Sie in den folgenden Themen der API-Referenz zum AWS -SDK für SAP ABAP.
  - CreateFunction
  - DeleteFunction
  - GetFunction
  - Invoke

- ListFunctions
- UpdateFunctionCode
- UpdateFunctionConfiguration

Swift

SDK für Swift

> **Note**
>
> Es gibt noch mehr dazu GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

Definieren Sie die erste Lambda-Funktion, die einfach den angegebenen Wert erhöht.

```
// 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()
```

Definieren Sie die zweite Lambda-Funktion, die eine arithmetische Operation mit zwei Zahlen ausführt.

```
// 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()
```

Definieren Sie das Hauptprogramm, das die beiden Lambda-Funktionen aufruft.

```
// 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."
        }
    }
}
```

- Weitere API-Informationen finden Sie in den folgenden Themen der API-Referenz zum AWS SDK für Swift.
  - CreateFunction
  - DeleteFunction
  - GetFunction
  - Invoke
  - ListFunctions
  - UpdateFunctionCode
  - UpdateFunctionConfiguration

Eine vollständige Liste der AWS SDK-Entwicklerhandbücher und Codebeispiele finden Sie unterLambda mit einem AWS SDK verwenden. Dieses Thema enthält auch Informationen zu den ersten Schritten und Details zu früheren SDK-Versionen.

Aktionen für Lambda unter Verwendung AWS SDKs

Die folgenden Codebeispiele zeigen, wie einzelne Lambda-Aktionen mit AWS SDKs ausgeführt werden. Jedes Beispiel enthält einen Link zu GitHub, wo Sie Anweisungen zum Einrichten und Ausführen des Codes finden.

Diese Auszüge rufen die Lambda-API auf und sind Codeauszüge aus größeren Programmen, die im Kontext ausgeführt werden müssen. Sie können Aktionen im Kontext unter Szenarien für die Verwendung von Lambda AWS SDKs anzeigen.

Die folgenden Beispiele enthalten nur die am häufigsten verwendeten Aktionen. Eine vollständige Liste finden Sie in der AWS Lambda -API-Referenz.

Beispiele

- Verwendung von CreateAlias mit einer CLI
- Verwendung CreateFunction mit einem AWS SDK oder CLI
- Verwendung von DeleteAlias mit einer CLI
- Verwendung DeleteFunction mit einem AWS SDK oder CLI
- Verwendung von DeleteFunctionConcurrency mit einer CLI
- Verwendung von DeleteProvisionedConcurrencyConfig mit einer CLI
- Verwendung von GetAccountSettings mit einer CLI
- Verwendung von GetAlias mit einer CLI
- Verwendung GetFunction mit einem AWS SDK oder CLI
- Verwendung von GetFunctionConcurrency mit einer CLI
- Verwendung von GetFunctionConfiguration mit einer CLI
- Verwendung von GetPolicy mit einer CLI
- Verwendung von GetProvisionedConcurrencyConfig mit einer CLI
- Verwendung Invoke mit einem AWS SDK oder CLI
- Verwendung ListFunctions mit einem AWS SDK oder CLI
- Verwendung von ListProvisionedConcurrencyConfigs mit einer CLI
- Verwendung von ListTags mit einer CLI
- Verwendung von ListVersionsByFunction mit einer CLI
- Verwendung von PublishVersion mit einer CLI

- Verwendung von PutFunctionConcurrency mit einer CLI
- Verwendung von PutProvisionedConcurrencyConfig mit einer CLI
- Verwendung von RemovePermission mit einer CLI
- Verwendung von TagResource mit einer CLI
- Verwendung von UntagResource mit einer CLI
- Verwendung von UpdateAlias mit einer CLI
- Verwendung UpdateFunctionCode mit einem AWS SDK oder CLI
- Verwendung UpdateFunctionConfiguration mit einem AWS SDK oder CLI

Verwendung von **`CreateAlias`** mit einer CLI

Die folgenden Code-Beispiele zeigen, wie `CreateAlias` verwendet wird.

CLI

AWS CLI

So erstellen Sie einen Alias für eine Lambda-Funktion

Im folgenden Beispiel für `create-alias` wird ein Alias namens `LIVE` erstellt, der auf Version 1 der `my-function`-Lambda-Funktion verweist.

```
aws lambda create-alias \
    --function-name my-function \
    --description "alias for live version of function" \
    --function-version 1 \
    --name LIVE
```

Ausgabe:

```
{
    "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"
}
```

Weitere Informationen finden Sie unter Konfiguration von AWS Lambda-Funktionsaliasen im AWS Lambda Developer Guide.

- Einzelheiten zur API finden Sie unter Befehlsreferenz CreateAlias.AWS CLI

PowerShell

Tools für PowerShell V4

Beispiel 1: In diesem Beispiel wird ein neuer Lambda-Alias für die angegebene Version und Routing-Konfiguration erstellt, um den Prozentsatz der Aufrufe für dieses System anzugeben.

```
New-LMAlias -FunctionName "MylambdaFunction123" -
RoutingConfig_AdditionalVersionWeight @{Name="1";Value="0.6} -Description "Alias
 for version 4" -FunctionVersion 4 -Name "PowershellAlias"
```

- Einzelheiten zur API finden Sie unter CreateAlias AWS -Tools für PowerShellCmdlet-Referenz (V4).

Tools für V5 PowerShell

Beispiel 1: In diesem Beispiel wird ein neuer Lambda-Alias für die angegebene Version und Routing-Konfiguration erstellt, um den Prozentsatz der Aufrufe für dieses System anzugeben.

```
New-LMAlias -FunctionName "MylambdaFunction123" -
RoutingConfig_AdditionalVersionWeight @{Name="1";Value="0.6} -Description "Alias
 for version 4" -FunctionVersion 4 -Name "PowershellAlias"
```

- Einzelheiten zur API finden Sie unter CreateAlias AWS -Tools für PowerShellCmdlet-Referenz (V5).

Eine vollständige Liste der AWS SDK-Entwicklerhandbücher und Codebeispiele finden Sie unter. Lambda mit einem AWS SDK verwenden Dieses Thema enthält auch Informationen zu den ersten Schritten und Details zu früheren SDK-Versionen.

Verwendung **`CreateFunction`** mit einem AWS SDK oder CLI

Die folgenden Code-Beispiele zeigen, wie `CreateFunction` verwendet wird.

Beispiele für Aktionen sind Codeauszüge aus größeren Programmen und müssen im Kontext ausgeführt werden. Im folgenden Codebeispiel können Sie diese Aktion im Kontext sehen:

- Kennenlernen der Grundlagen

.NET

SDK für .NET

> **Note**
>
> Es gibt noch mehr dazu GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
/// <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;
    }
```

- Einzelheiten zur API finden Sie CreateFunctionin der AWS SDK für .NET API-Referenz.

C++

SDK für C++

> **Note**
>
> Es gibt noch mehr dazu GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
        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;
        }
```

- Einzelheiten zur API finden Sie CreateFunctionin der AWS SDK für C++ API-Referenz.

CLI

AWS CLI

So erstellen Sie eine Lambda-Funktion

Im folgenden Beispiel für `create-function` wird eine Lambda-Funktion mit dem Namen `my-function` erstellt.

```
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
```

Inhalt von `my-function.zip`:

```
This file is a deployment package that contains your function code and any
 dependencies.
```

Ausgabe:

```
{
    "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": ""
}
```

Weitere Informationen finden Sie unter Konfigurieren des Speichers der Lambda-Funktion im AWS -Lambda-Entwicklerhandbuch.

- Einzelheiten zur API finden Sie CreateFunctionin der AWS CLI Befehlsreferenz.

Go

SDK für Go V2

> **Note**
>
> Es gibt noch mehr dazu GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
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
}
```

- Einzelheiten zur API finden Sie CreateFunctionin der AWS SDK für Go API-Referenz.

Java

SDK für Java 2.x

> **Note**
>
> Es gibt noch mehr dazu GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
    /**
     * 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 "";
    }
```

- Einzelheiten zur API finden Sie CreateFunctionin der AWS SDK for Java 2.x API-Referenz.

JavaScript

SDK für JavaScript (v3)

> **Note**
>
> Es gibt noch mehr dazu GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
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);
};
```

- Einzelheiten zur API finden Sie CreateFunctionin der AWS SDK für JavaScript API-Referenz.

Kotlin

SDK für Kotlin

> **Note**
>
> Es gibt noch mehr dazu GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
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
    }
}
```

- API-Details finden Sie CreateFunctionin der API-Referenz zum AWS SDK für Kotlin.

PHP

SDK für PHP

> **Note**
>
> Es gibt noch mehr dazu. GitHub Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
    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",
            ]);
        });
    }
```

- Einzelheiten zur API finden Sie CreateFunctionin der AWS SDK für PHP API-Referenz.

PowerShell

Tools für PowerShell V4

Beispiel 1: In diesem Beispiel wird eine neue C#-Funktion (dotnetcore1.0 Runtime) mit dem Namen AWS Lambda erstellt, die die kompilierten Binärdateien für die Funktion aus einer ZIP-Datei MyFunction im lokalen Dateisystem bereitstellt (relative oder absolute Pfade können verwendet werden). C#-Lambda-Funktionen spezifizieren den Handler für die Funktion mit der Bezeichnung AssemblyName: :Namespace. ClassName::. MethodName Sie sollten die Teile Assembly-Name (ohne .dll-Suffix), Namespace, Klassenname und

Methodenname der Handler-Spezifikation entsprechend ersetzen. Für die neue Funktion werden die Umgebungsvariablen 'envvar1' und 'envvar2' aus den bereitgestellten Werten eingerichtet.

```
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" }
```

Ausgabe:

```
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        :
```

Beispiel 2: Dieses Beispiel ähnelt dem vorherigen, mit dem Unterschied, dass die Funktions-Binärdateien zunächst in einen Amazon S3-Bucket hochgeladen werden (der sich in derselben Region befinden muss wie die vorgesehene Lambda-Funktion) und das resultierende S3-Objekt dann bei der Erstellung der Funktion referenziert wird.

```
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" }
```

- Einzelheiten zur API finden Sie unter CreateFunction AWS -Tools für PowerShellCmdlet-Referenz (V4).

Tools für V5 PowerShell

Beispiel 1: In diesem Beispiel wird eine neue C#-Funktion (dotnetcore1.0 Runtime) mit dem Namen AWS Lambda erstellt, die die kompilierten Binärdateien für die Funktion aus einer ZIP-Datei MyFunction im lokalen Dateisystem bereitstellt (relative oder absolute Pfade können verwendet werden). C#-Lambda-Funktionen spezifizieren den Handler für die Funktion mit der Bezeichnung AssemblyName: :Namespace. ClassName::. MethodName Sie sollten die Teile Assembly-Name (ohne .dll-Suffix), Namespace, Klassenname und Methodenname der Handler-Spezifikation entsprechend ersetzen. Für die neue Funktion werden die Umgebungsvariablen 'envvar1' und 'envvar2' aus den bereitgestellten Werten eingerichtet.

```
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" }
```

Ausgabe:

```
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        :
```

Beispiel 2: Dieses Beispiel ähnelt dem vorherigen, mit dem Unterschied, dass die Funktions-Binärdateien zunächst in einen Amazon S3-Bucket hochgeladen werden (der sich in derselben Region befinden muss wie die vorgesehene Lambda-Funktion) und das resultierende S3-Objekt dann bei der Erstellung der Funktion referenziert wird.

```
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" }
```

- Einzelheiten zur API finden Sie unter CreateFunction AWS -Tools für PowerShellCmdlet-Referenz (V5).

Python

SDK für Python (Boto3)

> **Note**
>
> Es gibt noch mehr dazu. GitHub Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
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
```

- Einzelheiten zur API finden Sie CreateFunctionin AWS SDK for Python (Boto3) API Reference.

Ruby

SDK für Ruby

> **Note**
>
> Es gibt noch mehr dazu. GitHub Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
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
```

- Einzelheiten zur API finden Sie CreateFunctionin der AWS SDK für Ruby API-Referenz.

Rust

SDK für Rust

> **Note**
>
> Es gibt noch mehr dazu GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
   /**
    * 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())
    }
```

- Einzelheiten zur API finden Sie CreateFunctionin der API-Referenz zum AWS SDK für Rust.

SAP ABAP

SDK für SAP ABAP

> **Note**
>
> Es gibt noch mehr dazu. GitHub Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
    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.
```

- Einzelheiten zur API finden Sie CreateFunctionin der API-Referenz zum AWS SDK für SAP ABAP.

Swift

SDK für Swift

> **Note**
>
> Es gibt noch mehr dazu GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
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
        }
```

- Einzelheiten zur API finden Sie CreateFunctionin der API-Referenz zum AWS SDK für Swift.

Eine vollständige Liste der AWS SDK-Entwicklerhandbücher und Codebeispiele finden Sie unterLambda mit einem AWS SDK verwenden. Dieses Thema enthält auch Informationen zu den ersten Schritten und Details zu früheren SDK-Versionen.

Verwendung von **`DeleteAlias`** mit einer CLI

Die folgenden Code-Beispiele zeigen, wie `DeleteAlias` verwendet wird.

CLI

AWS CLI

So löschen Sie einen Alias einer Lambda-Funktion

Im folgenden Beispiel für `delete-alias` wird der Alias mit dem Namen `LIVE` aus der `my-function`-Lambda-Funktion gelöscht.

```
aws lambda delete-alias \
    --function-name my-function \
```

```
    --name LIVE
```

Mit diesem Befehl wird keine Ausgabe zurückgegeben.

Weitere Informationen finden Sie unter Konfiguration von AWS Lambda-Funktionsaliasen im AWS Lambda Developer Guide.

- Einzelheiten zur API finden Sie unter Befehlsreferenz DeleteAlias.AWS CLI

PowerShell

Tools für PowerShell V4

Beispiel 1: In diesem Beispiel wird die im Befehl erwähnte Lambda-Funktion Alias gelöscht.

```
Remove-LMAlias -FunctionName "MylambdaFunction123" -Name "NewAlias"
```

- Einzelheiten zur API finden Sie unter DeleteAlias AWS -Tools für PowerShellCmdlet-Referenz (V4).

Tools für V5 PowerShell

Beispiel 1: In diesem Beispiel wird die im Befehl erwähnte Lambda-Funktion Alias gelöscht.

```
Remove-LMAlias -FunctionName "MylambdaFunction123" -Name "NewAlias"
```

- Einzelheiten zur API finden Sie unter DeleteAlias AWS -Tools für PowerShellCmdlet-Referenz (V5).

Eine vollständige Liste der AWS SDK-Entwicklerhandbücher und Codebeispiele finden Sie unter. Lambda mit einem AWS SDK verwenden Dieses Thema enthält auch Informationen zu den ersten Schritten und Details zu früheren SDK-Versionen.

Verwendung **`DeleteFunction`** mit einem AWS SDK oder CLI

Die folgenden Code-Beispiele zeigen, wie `DeleteFunction` verwendet wird.

Beispiele für Aktionen sind Codeauszüge aus größeren Programmen und müssen im Kontext ausgeführt werden. Im folgenden Codebeispiel können Sie diese Aktion im Kontext sehen:

- Kennenlernen der Grundlagen

.NET

SDK für .NET

> **Note**
>
> Es gibt noch mehr dazu GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
    /// <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;
    }
```

- Einzelheiten zur API finden Sie DeleteFunctionin der AWS SDK für .NET API-Referenz.

C++

SDK für C++

> **Note**
>
> Es gibt noch mehr dazu GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
        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;
    }
```

- Einzelheiten zur API finden Sie DeleteFunctionin der AWS SDK für C++ API-Referenz.

CLI

AWS CLI

Beispiel 1: So löschen Sie eine Lambda-Funktion anhand des Funktionsnamens

Im folgenden Beispiel für `delete-function` wird die Lambda-Funktion `my-function` durch Angabe des Funktionsnamens gelöscht.

```
aws lambda delete-function \
    --function-name my-function
```

Mit diesem Befehl wird keine Ausgabe zurückgegeben.

Beispiel 2: Eine Lambda-Funktion anhand des Funktions-ARN löschen

Im folgenden Beispiel für `delete-function` wird die Lambda-Funktion `my-function` durch Angabe des ARN der Funktion gelöscht.

```
aws lambda delete-function \
    --function-name arn:aws:lambda:us-west-2:123456789012:function:my-function
```

Mit diesem Befehl wird keine Ausgabe zurückgegeben.

Beispiel 3: Eine Lambda-Funktion anhand eines teilweisen Funktions-ARN löschen

Im folgenden Beispiel für `delete-function` wird die Lambda-Funktion `my-function` durch Angabe des teilweisen ARN der Funktion gelöscht.

```
aws lambda delete-function \
    --function-name 123456789012:function:my-function
```

Mit diesem Befehl wird keine Ausgabe zurückgegeben.

Weitere Informationen finden Sie unter Konfigurieren von AWS -Lambda-Funktionen im AWS -Lambda-Entwicklerhandbuch.

- Einzelheiten zur API finden Sie DeleteFunctionin der AWS CLI Befehlsreferenz.

Go

SDK für Go V2

> **Note**
>
> Es gibt noch mehr dazu GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
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)
 }
}
```

- Einzelheiten zur API finden Sie DeleteFunctionin der AWS SDK für Go API-Referenz.

Java

SDK für Java 2.x

> **Note**
>
> Es gibt noch mehr dazu GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
   /**
    * 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);
       }
   }
```

- Einzelheiten zur API finden Sie DeleteFunctionin der AWS SDK for Java 2.x API-Referenz.

JavaScript

SDK für JavaScript (v3)

> **Note**
>
> Es gibt noch mehr dazu GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
/**
 * @param {string} funcName
 */
const deleteFunction = (funcName) => {
  const client = new LambdaClient({});
  const command = new DeleteFunctionCommand({ FunctionName: funcName });
  return client.send(command);
};
```

- Einzelheiten zur API finden Sie DeleteFunctionin der AWS SDK für JavaScript API-Referenz.

Kotlin

SDK für Kotlin

> **Note**
>
> Es gibt noch mehr dazu GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
suspend fun delLambdaFunction(myFunctionName: String) {
    val request =
        DeleteFunctionRequest {
            functionName = myFunctionName
        }
```

```
    LambdaClient { region = "us-east-1" }.use { awsLambda ->
        awsLambda.deleteFunction(request)
        println("$myFunctionName was deleted")
    }
}
```

- API-Details finden Sie DeleteFunctionin der API-Referenz zum AWS SDK für Kotlin.

PHP

SDK für PHP

> **Note**
>
> Es gibt noch mehr dazu. GitHub Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
    public function deleteFunction($functionName)
    {
        return $this->lambdaClient->deleteFunction([
            'FunctionName' => $functionName,
        ]);
    }
```

- Einzelheiten zur API finden Sie DeleteFunctionin der AWS SDK für PHP API-Referenz.

PowerShell

Tools für PowerShell V4

Beispiel 1: In diesem Beispiel wird eine bestimmte Version einer Lambda-Funktion gelöscht

```
Remove-LMFunction -FunctionName "MylambdaFunction123" -Qualifier '3'
```

- Einzelheiten zur API finden Sie unter DeleteFunction AWS -Tools für PowerShellCmdlet-Referenz (V4).

Tools für V5 PowerShell

Beispiel 1: In diesem Beispiel wird eine bestimmte Version einer Lambda-Funktion gelöscht

```
Remove-LMFunction -FunctionName "MylambdaFunction123" -Qualifier '3'
```

- Einzelheiten zur API finden Sie unter DeleteFunction AWS -Tools für PowerShellCmdlet-Referenz (V5).

Python

SDK für Python (Boto3)

> **Note**
>
> Es gibt noch mehr dazu. GitHub Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
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
```

- Einzelheiten zur API finden Sie DeleteFunctionin AWS SDK for Python (Boto3) API Reference.

Ruby

SDK für Ruby

> **Note**
>
> Es gibt noch mehr dazu. GitHub Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
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
```

- Einzelheiten zur API finden Sie DeleteFunctionin der AWS SDK für Ruby API-Referenz.

Rust

SDK für Rust

> **Note**
>
> Es gibt noch mehr dazu GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
   /** 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)
    }
```

- Einzelheiten zur API finden Sie DeleteFunctionin der API-Referenz zum AWS SDK für Rust.

SAP ABAP

SDK für SAP ABAP

> **Note**
>
> Es gibt noch mehr dazu. GitHub Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
    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.
```

- Einzelheiten zur API finden Sie DeleteFunctionin der API-Referenz zum AWS SDK für SAP ABAP.

Swift

SDK für Swift

> **Note**
>
> Es gibt noch mehr dazu GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
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.")
        }
```

- Einzelheiten zur API finden Sie DeleteFunctionin der API-Referenz zum AWS SDK für Swift.

Eine vollständige Liste der AWS SDK-Entwicklerhandbücher und Codebeispiele finden Sie unterLambda mit einem AWS SDK verwenden. Dieses Thema enthält auch Informationen zu den ersten Schritten und Details zu früheren SDK-Versionen.

Verwendung von **`DeleteFunctionConcurrency`** mit einer CLI

Die folgenden Code-Beispiele zeigen, wie `DeleteFunctionConcurrency` verwendet wird.

CLI

AWS CLI

So entfernen Sie das Limit für die reservierte gleichzeitige Ausführung einer Funktion

Im folgenden `delete-function-concurrency`-Beispiel wird das reservierte Limit für gleichzeitige Ausführung aus der `my-function`-Funktion gelöscht.

```
aws lambda delete-function-concurrency \
    --function-name  my-function
```

Mit diesem Befehl wird keine Ausgabe zurückgegeben.

Weitere Informationen finden Sie unter Gleichzeitigkeit für eine Lambda-Funktion reservieren im AWS -Lambda-Entwicklerhandbuch.

- Einzelheiten zur API finden Sie DeleteFunctionConcurrencyin der AWS CLI Befehlsreferenz.

PowerShell

Tools für PowerShell V4

Beispiel 1: In diesem Beispiel wird die Funktionsgleichzeitigkeit der Lambda-Funktion aufgehoben.

```
Remove-LMFunctionConcurrency -FunctionName "MylambdaFunction123"
```

- Einzelheiten zur API finden Sie unter DeleteFunctionConcurrency AWS -Tools für PowerShellCmdlet-Referenz (V4).

Tools für V5 PowerShell

Beispiel 1: In diesem Beispiel wird die Funktionsgleichzeitigkeit der Lambda-Funktion aufgehoben.

```
Remove-LMFunctionConcurrency -FunctionName "MylambdaFunction123"
```

- Einzelheiten zur API finden Sie unter DeleteFunctionConcurrency AWS -Tools für PowerShellCmdlet-Referenz (V5).

Eine vollständige Liste der AWS SDK-Entwicklerhandbücher und Codebeispiele finden Sie unter. Lambda mit einem AWS SDK verwenden Dieses Thema enthält auch Informationen zu den ersten Schritten und Details zu früheren SDK-Versionen.

Verwendung von **`DeleteProvisionedConcurrencyConfig`** mit einer CLI

Die folgenden Code-Beispiele zeigen, wie `DeleteProvisionedConcurrencyConfig` verwendet wird.

CLI

AWS CLI

So löschen Sie eine bereitgestellte Gleichzeitigkeitskonfiguration

Das folgende `delete-provisioned-concurrency-config`-Beispiel löscht die bereitgestellte Gleichzeitigkeitskonfiguration für den `GREEN`-Alias der angegebenen Funktion.

```
aws lambda delete-provisioned-concurrency-config \
    --function-name my-function \
    --qualifier GREEN
```

- Einzelheiten zur API finden Sie DeleteProvisionedConcurrencyConfigin der AWS CLI Befehlsreferenz.

PowerShell

Tools für PowerShell V4

Beispiel 1: In diesem Beispiel wird die bereitgestellte Gleichzeitigkeitskonfiguration für einen bestimmten Alias entfernt.

```
Remove-LMProvisionedConcurrencyConfig -FunctionName "MylambdaFunction123" -
Qualifier "NewAlias1"
```

- Einzelheiten zur API finden Sie unter DeleteProvisionedConcurrencyConfig AWS -Tools für PowerShellCmdlet-Referenz (V4).

Tools für V5 PowerShell

Beispiel 1: In diesem Beispiel wird die bereitgestellte Gleichzeitigkeitskonfiguration für einen bestimmten Alias entfernt.

```
Remove-LMProvisionedConcurrencyConfig -FunctionName "MylambdaFunction123" -
Qualifier "NewAlias1"
```

- Einzelheiten zur API finden Sie unter DeleteProvisionedConcurrencyConfig AWS -Tools für PowerShellCmdlet-Referenz (V5).

Eine vollständige Liste der AWS SDK-Entwicklerhandbücher und Codebeispiele finden Sie unter. Lambda mit einem AWS SDK verwenden Dieses Thema enthält auch Informationen zu den ersten Schritten und Details zu früheren SDK-Versionen.

Verwendung von **`GetAccountSettings`** mit einer CLI

Die folgenden Code-Beispiele zeigen, wie `GetAccountSettings` verwendet wird.

CLI

AWS CLI

Um Details zu Ihrem Konto in einer AWS Region abzurufen

Im folgenden `get-account-settings`-Beispiel werden die Lambda-Grenzwerte und Nutzungsinformationen für Ihr Konto angezeigt.

```
aws lambda get-account-settings
```

Ausgabe:

```
{
    "AccountLimit": {
       "CodeSizeUnzipped": 262144000,
       "UnreservedConcurrentExecutions": 1000,
       "ConcurrentExecutions": 1000,
       "CodeSizeZipped": 52428800,
       "TotalCodeSize": 80530636800
    },
    "AccountUsage": {
```

```
        "FunctionCount": 4,
        "TotalCodeSize": 9426
    }
}
```

Weitere Informationen finden Sie unter AWS Lambda Limits im AWS -Lambda-Entwicklerhandbuch.

- Einzelheiten zur API finden Sie GetAccountSettingsin der AWS CLI Befehlsreferenz.

PowerShell

Tools für PowerShell V4

Beispiel 1: Dieses Beispiel wird angezeigt, um das Kontolimit und die Kontonutzung zu vergleichen

```
Get-LMAccountSetting | Select-Object
 @{Name="TotalCodeSizeLimit";Expression={$_.AccountLimit.TotalCodeSize}},
 @{Name="TotalCodeSizeUsed";Expression={$_.AccountUsage.TotalCodeSize}}
```

Ausgabe:

```
TotalCodeSizeLimit TotalCodeSizeUsed
------------------ -----------------
       80530636800          15078795
```

- Einzelheiten zur API finden Sie unter GetAccountSettings AWS -Tools für PowerShellCmdlet-Referenz (V4).

Tools für V5 PowerShell

Beispiel 1: Dieses Beispiel wird angezeigt, um das Kontolimit und die Kontonutzung zu vergleichen

```
Get-LMAccountSetting | Select-Object
 @{Name="TotalCodeSizeLimit";Expression={$_.AccountLimit.TotalCodeSize}},
 @{Name="TotalCodeSizeUsed";Expression={$_.AccountUsage.TotalCodeSize}}
```

Ausgabe:

```
TotalCodeSizeLimit TotalCodeSizeUsed
```

```
------------------ -----------------
       80530636800          15078795
```

- Einzelheiten zur API finden Sie unter GetAccountSettings AWS -Tools für PowerShellCmdlet-Referenz (V5).

Eine vollständige Liste der AWS SDK-Entwicklerhandbücher und Codebeispiele finden Sie unter. Lambda mit einem AWS SDK verwenden Dieses Thema enthält auch Informationen zu den ersten Schritten und Details zu früheren SDK-Versionen.

Verwendung von **`GetAlias`** mit einer CLI

Die folgenden Code-Beispiele zeigen, wie `GetAlias` verwendet wird.

CLI

AWS CLI

So rufen Sie Details über einen Funktionsalias ab

Das folgende `get-alias`-Beispiel zeigt Details für den Alias namens `LIVE` für die `my-function`-Lambda-Funktion an.

```
aws lambda get-alias \
    --function-name my-function \
    --name LIVE
```

Ausgabe:

```
{
    "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"
}
```

Weitere Informationen finden Sie unter Konfiguration von AWS Lambda-Funktionsaliasen im AWS Lambda Developer Guide.

- Einzelheiten zur API finden Sie unter Befehlsreferenz GetAlias.AWS CLI

PowerShell

Tools für PowerShell V4

Beispiel 1: In diesem Beispiel werden die Gewichtungen der Routing-Konfiguration für einen bestimmten Lambda-Funktionsalias abgerufen.

```
Get-LMAlias -FunctionName "MylambdaFunction123" -Name "newlabel1" -Select
 RoutingConfig
```

Ausgabe:

```
AdditionalVersionWeights
------------------------
{[1, 0.6]}
```

- Einzelheiten zur API finden Sie unter GetAlias AWS -Tools für PowerShellCmdlet-Referenz (V4).

Tools für V5 PowerShell

Beispiel 1: In diesem Beispiel werden die Gewichtungen der Routing-Konfiguration für einen bestimmten Lambda-Funktionsalias abgerufen.

```
Get-LMAlias -FunctionName "MylambdaFunction123" -Name "newlabel1" -Select
 RoutingConfig
```

Ausgabe:

```
AdditionalVersionWeights
------------------------
{[1, 0.6]}
```

- Einzelheiten zur API finden Sie unter GetAlias AWS -Tools für PowerShellCmdlet-Referenz (V5).

Eine vollständige Liste der AWS SDK-Entwicklerhandbücher und Codebeispiele finden Sie unter. Lambda mit einem AWS SDK verwenden Dieses Thema enthält auch Informationen zu den ersten Schritten und Details zu früheren SDK-Versionen.

Verwendung **`GetFunction`** mit einem AWS SDK oder CLI

Die folgenden Code-Beispiele zeigen, wie `GetFunction` verwendet wird.

Beispiele für Aktionen sind Codeauszüge aus größeren Programmen und müssen im Kontext ausgeführt werden. Im folgenden Codebeispiel können Sie diese Aktion im Kontext sehen:

- Kennenlernen der Grundlagen

.NET

SDK für .NET

> **Note**
>
> Es gibt noch mehr dazu GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
   /// <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;
   }
```

- Einzelheiten zur API finden Sie GetFunctionin der AWS SDK für .NET API-Referenz.

C++

SDK für C++

> **Note**
>
> Es gibt noch mehr dazu GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
        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;
        }
```

- Einzelheiten zur API finden Sie GetFunctionin der AWS SDK für C++ API-Referenz.

CLI

AWS CLI

So rufen Sie Informationen über eine Funktion ab

Das folgende Beispiel für `get-function` zeigt Informationen zur Funktion `my-function` an.

```
aws lambda get-function \
    --function-name  my-function
```

Ausgabe:

```
{
    "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": ""
    }
}
```

Weitere Informationen finden Sie unter Konfigurieren des Speichers der Lambda-Funktion im AWS -Lambda-Entwicklerhandbuch.

- Einzelheiten zur API finden Sie GetFunctionin der AWS CLI Befehlsreferenz.

Go

SDK für Go V2

> **Note**
>
> Es gibt noch mehr dazu GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
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
}
```

- Einzelheiten zur API finden Sie GetFunctionin der AWS SDK für Go API-Referenz.

Java

SDK für Java 2.x

> **Note**
>
> Es gibt noch mehr dazu GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
    /**
     * 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);
        }
    }
```

- Einzelheiten zur API finden Sie GetFunctionin der AWS SDK for Java 2.x API-Referenz.

JavaScript

SDK für JavaScript (v3)

> **Note**
>
> Es gibt noch mehr dazu GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
const getFunction = (funcName) => {
  const client = new LambdaClient({});
  const command = new GetFunctionCommand({ FunctionName: funcName });
  return client.send(command);
};
```

- Einzelheiten zur API finden Sie GetFunctionin der AWS SDK für JavaScript API-Referenz.

PHP

SDK für PHP

> **Note**
>
> Es gibt noch mehr dazu GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
    public function getFunction($functionName)
    {
        return $this->lambdaClient->getFunction([
            'FunctionName' => $functionName,
        ]);
    }
```

- Einzelheiten zur API finden Sie GetFunctionin der AWS SDK für PHP API-Referenz.

Python

SDK für Python (Boto3)

> **Note**
>
> Es gibt noch mehr dazu GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
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
```

- Einzelheiten zur API finden Sie GetFunctionin AWS SDK for Python (Boto3) API Reference.

Ruby

SDK für Ruby

> **Note**
>
> Es gibt noch mehr dazu. GitHub Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
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
```

- Einzelheiten zur API finden Sie GetFunctionin der AWS SDK für Ruby API-Referenz.

Rust

SDK für Rust

> **Note**
>
> Es gibt noch mehr dazu GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
    /** 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)
    }
```

- Einzelheiten zur API finden Sie GetFunctionin der API-Referenz zum AWS SDK für Rust.

SAP ABAP

SDK für SAP ABAP

> **Note**
>
> Es gibt noch mehr dazu. GitHub Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
    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.
```

- Einzelheiten zur API finden Sie GetFunctionin der API-Referenz zum AWS SDK für SAP ABAP.

Swift

SDK für Swift

> **Note**
>
> Es gibt noch mehr dazu GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
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
    }
```

- Einzelheiten zur API finden Sie GetFunctionin der API-Referenz zum AWS SDK für Swift.

Eine vollständige Liste der AWS SDK-Entwicklerhandbücher und Codebeispiele finden Sie unterLambda mit einem AWS SDK verwenden. Dieses Thema enthält auch Informationen zu den ersten Schritten und Details zu früheren SDK-Versionen.

Verwendung von **`GetFunctionConcurrency`** mit einer CLI

Die folgenden Code-Beispiele zeigen, wie `GetFunctionConcurrency` verwendet wird.

CLI

AWS CLI

So zeigen Sie die reservierte Gleichzeitigkeitseinstellung für eine Funktion an

Das folgende `get-function-concurrency`-Beispiel ruft die reservierte Gleichzeitigkeitseinstellung für die angegebene Funktion ab.

```
aws lambda get-function-concurrency \
    --function-name my-function
```

Ausgabe:

```
{
    "ReservedConcurrentExecutions": 250
}
```

- Einzelheiten zur API finden Sie GetFunctionConcurrencyin der AWS CLI Befehlsreferenz.

PowerShell

Tools für PowerShell V4

Beispiel 1: Dieses Beispiel ruft die reservierte Gleichzeitigkeit für die Lambda-Funktion ab

```
Get-LMFunctionConcurrency -FunctionName "MylambdaFunction123" -Select *
```

Ausgabe:

```
ReservedConcurrentExecutions
----------------------------
100
```

- Einzelheiten zur API finden Sie unter GetFunctionConcurrency AWS -Tools für PowerShellCmdlet-Referenz (V4).

Tools für V5 PowerShell

Beispiel 1: Dieses Beispiel ruft die reservierte Gleichzeitigkeit für die Lambda-Funktion ab

```
Get-LMFunctionConcurrency -FunctionName "MylambdaFunction123" -Select *
```

Ausgabe:

```
ReservedConcurrentExecutions
----------------------------
```

```
100
```

- Einzelheiten zur API finden Sie unter GetFunctionConcurrency AWS -Tools für PowerShellCmdlet-Referenz (V5).

Eine vollständige Liste der AWS SDK-Entwicklerhandbücher und Codebeispiele finden Sie unter. Lambda mit einem AWS SDK verwenden Dieses Thema enthält auch Informationen zu den ersten Schritten und Details zu früheren SDK-Versionen.

Verwendung von **`GetFunctionConfiguration`** mit einer CLI

Die folgenden Code-Beispiele zeigen, wie `GetFunctionConfiguration` verwendet wird.

CLI

AWS CLI

So rufen Sie die versionsspezifischen Einstellungen einer Lambda-Funktion ab

Das folgende `get-function-configuration`-Beispiel zeigt die Einstellungen für Version 2 der `my-function`-Funktion.

```
aws lambda get-function-configuration \
    --function-name  my-function:2
```

Ausgabe:

```
{
    "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"
}
```

Weitere Informationen finden Sie unter Konfigurieren von AWS -Lambda-Funktionen im AWS -Lambda-Entwicklerhandbuch.

- Einzelheiten zur API finden Sie GetFunctionConfigurationin der AWS CLI Befehlsreferenz.

PowerShell

Tools für PowerShell V4

Beispiel 1: Dieses Beispiel gibt die versionsspezifische Konfiguration einer Lambda-Funktion zurück.

```
Get-LMFunctionConfiguration -FunctionName "MylambdaFunction123" -Qualifier
 "PowershellAlias"
```

Ausgabe:

```
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
```

- Einzelheiten zur API finden Sie unter GetFunctionConfiguration AWS -Tools für PowerShellCmdlet-Referenz (V4).

Tools für V5 PowerShell

Beispiel 1: Dieses Beispiel gibt die versionsspezifische Konfiguration einer Lambda-Funktion zurück.

```
Get-LMFunctionConfiguration -FunctionName "MylambdaFunction123" -Qualifier
 "PowershellAlias"
```

Ausgabe:

```
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
```

- Einzelheiten zur API finden Sie unter GetFunctionConfiguration AWS -Tools für PowerShellCmdlet-Referenz (V5).

Eine vollständige Liste der AWS SDK-Entwicklerhandbücher und Codebeispiele finden Sie unter. Lambda mit einem AWS SDK verwenden Dieses Thema enthält auch Informationen zu den ersten Schritten und Details zu früheren SDK-Versionen.

Verwendung von **`GetPolicy`** mit einer CLI

Die folgenden Code-Beispiele zeigen, wie `GetPolicy` verwendet wird.

CLI

AWS CLI

So rufen Sie die ressourcenbasierte IAM-Richtlinie für eine Funktion, eine Version oder einen Alias ab

Das folgende `get-policy`-Beispiel zeigt Richtlinieninformationen über die `my-function`-Lambda-Funktion an.

```
aws lambda get-policy \
    --function-name my-function
```

Ausgabe:

```
{
    "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"
}
```

Weitere Informationen finden Sie unter Using Resource-based Policies for AWS Lambda im Lambda Developer Guide AWS .

- Einzelheiten zur API finden Sie unter GetPolicyBefehlsreferenz.AWS CLI

PowerShell

Tools für PowerShell V4

Beispiel 1: In diesem Beispiel wird die Funktionsrichtlinie der Lambda-Funktion angezeigt

```
Get-LMPolicy -FunctionName test -Select Policy
```

Ausgabe:

```
{"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"}]}
```

- Einzelheiten zur API finden Sie unter GetPolicy AWS -Tools für PowerShellCmdlet-Referenz (V4).

Tools für V5 PowerShell

Beispiel 1: In diesem Beispiel wird die Funktionsrichtlinie der Lambda-Funktion angezeigt

```
Get-LMPolicy -FunctionName test -Select Policy
```

Ausgabe:

```
{"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"}]}
```

- Einzelheiten zur API finden Sie unter GetPolicy AWS -Tools für PowerShellCmdlet-Referenz (V5).

Eine vollständige Liste der AWS SDK-Entwicklerhandbücher und Codebeispiele finden Sie unter. Lambda mit einem AWS SDK verwenden Dieses Thema enthält auch Informationen zu den ersten Schritten und Details zu früheren SDK-Versionen.

Verwendung von **`GetProvisionedConcurrencyConfig`** mit einer CLI

Die folgenden Code-Beispiele zeigen, wie `GetProvisionedConcurrencyConfig` verwendet wird.

CLI

AWS CLI

So zeigen Sie eine bereitgestellte Gleichzeitigkeitskonfiguration an

Im folgenden `get-provisioned-concurrency-config`-Beispiel werden Details für die bereitgestellte Gleichzeitigkeitskonfiguration für den BLUE-Alias der angegebenen Funktion angezeigt.

```
aws lambda get-provisioned-concurrency-config \
    --function-name my-function \
    --qualifier BLUE
```

Ausgabe:

```
{
    "RequestedProvisionedConcurrentExecutions": 100,
    "AvailableProvisionedConcurrentExecutions": 100,
    "AllocatedProvisionedConcurrentExecutions": 100,
    "Status": "READY",
```

```
    "LastModified": "2019-12-31T20:28:49+0000"
}
```

- Einzelheiten zur API finden Sie GetProvisionedConcurrencyConfigin der AWS CLI Befehlsreferenz.

PowerShell

Tools für PowerShell V4

Beispiel 1: In diesem Beispiel wird die bereitgestellte Gleichzeitigkeitskonfiguration für den angegebenen Alias der Lambda-Funktion abgerufen.

```
C:\>Get-LMProvisionedConcurrencyConfig -FunctionName "MylambdaFunction123" -
Qualifier "NewAlias1"
```

Ausgabe:

```
AllocatedProvisionedConcurrentExecutions : 0
AvailableProvisionedConcurrentExecutions : 0
LastModified                             : 2020-01-15T03:21:26+0000
RequestedProvisionedConcurrentExecutions : 70
Status                                   : IN_PROGRESS
StatusReason                             :
```

- Einzelheiten zur API finden Sie unter GetProvisionedConcurrencyConfig AWS -Tools für PowerShellCmdlet-Referenz (V4).

Tools für V5 PowerShell

Beispiel 1: In diesem Beispiel wird die bereitgestellte Gleichzeitigkeitskonfiguration für den angegebenen Alias der Lambda-Funktion abgerufen.

```
C:\>Get-LMProvisionedConcurrencyConfig -FunctionName "MylambdaFunction123" -
Qualifier "NewAlias1"
```

Ausgabe:

```
AllocatedProvisionedConcurrentExecutions : 0
AvailableProvisionedConcurrentExecutions : 0
LastModified                             : 2020-01-15T03:21:26+0000
```

```
RequestedProvisionedConcurrentExecutions : 70
Status                                   : IN_PROGRESS
StatusReason                             :
```

- Einzelheiten zur API finden Sie unter GetProvisionedConcurrencyConfig AWS -Tools für PowerShellCmdlet-Referenz (V5).

Eine vollständige Liste der AWS SDK-Entwicklerhandbücher und Codebeispiele finden Sie unter. Lambda mit einem AWS SDK verwenden Dieses Thema enthält auch Informationen zu den ersten Schritten und Details zu früheren SDK-Versionen.

Verwendung **`Invoke`** mit einem AWS SDK oder CLI

Die folgenden Code-Beispiele zeigen, wie `Invoke` verwendet wird.

Beispiele für Aktionen sind Codeauszüge aus größeren Programmen und müssen im Kontext ausgeführt werden. Im folgenden Codebeispiel können Sie diese Aktion im Kontext sehen:

- Kennenlernen der Grundlagen

.NET

SDK für .NET

> **Note**
>
> Es gibt noch mehr dazu GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
    /// <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;
    }
```

- Weitere API-Informationen finden Sie unter Invoke in der AWS SDK für .NET -API-Referenz.

C++

SDK für C++

> **Note**
>
> Es gibt noch mehr GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
        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;
        }
```

- Weitere API-Informationen finden Sie unter Invoke in der AWS SDK für C++ -API-Referenz.

CLI

AWS CLI

Beispiel 1: Eine Lambda-Funktion synchron aufrufen

Im folgenden Beispiel für `invoke` wird die Funktion `my-function` synchron aufgerufen. Die `cli-binary-format` Option ist erforderlich, wenn Sie AWS CLI Version 2 verwenden. Weitere Informationen finden Sie unter Von der AWS CLI unterstützte globale Befehlszeilenoptionen im AWS -CLI-Benutzerhandbuch.

```
aws lambda invoke \
    --function-name my-function \
    --cli-binary-format raw-in-base64-out \
    --payload '{ "name": "Bob" }' \
    response.json
```

Ausgabe:

```
{
```

```
    "ExecutedVersion": "$LATEST",
    "StatusCode": 200
}
```

Weitere Informationen finden Sie unter Synchrones Aufrufen einer Lambda-Funktion im Entwicklerhandbuch für AWS Lambda.

Beispiel 2: Eine Lambda-Funktion asynchron aufrufen

Im folgenden Beispiel für `invoke` wird die Funktion `my-function` asynchron aufgerufen. Die `cli-binary-format` Option ist erforderlich, wenn Sie AWS CLI Version 2 verwenden. Weitere Informationen finden Sie unter Von der AWS CLI unterstützte globale Befehlszeilenoptionen im AWS -CLI-Benutzerhandbuch.

```
aws lambda invoke \
    --function-name my-function \
    --invocation-type Event \
    --cli-binary-format raw-in-base64-out \
    --payload '{ "name": "Bob" }' \
    response.json
```

Ausgabe:

```
{
    "StatusCode": 202
}
```

Weitere Informationen finden Sie unter Asynchrones Aufrufen einer Lambda-Funktion im Entwicklerhandbuch für AWS Lambda.

- Weitere API-Informationen finden Sie unter Invoke in der AWS CLI -Befehlsreferenz.

Go

SDK für Go V2

> **Note**
>
> Es gibt noch mehr dazu GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
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
}
```

- Weitere API-Informationen finden Sie unter Invoke in der AWS SDK für Go -API-Referenz.

Java

SDK für Java 2.x

> **Note**
>
> Es gibt noch mehr GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
   /**
    * 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);
        }
    }
```

- Weitere API-Informationen finden Sie unter Invoke in der AWS SDK for Java 2.x -API-Referenz.

JavaScript

SDK für JavaScript (v3)

> **Note**
>
> Es gibt noch mehr dazu GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
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 };
};
```

- Weitere API-Informationen finden Sie unter Invoke in der AWS SDK für JavaScript -API-Referenz.

Kotlin

SDK für Kotlin

> **Note**
>
> Es gibt noch mehr GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
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}")
    }
}
```

- Weitere API-Informationen finden Sie unter Invoke in der API-Referenz zum AWS -SDK für Kotlin.

PHP

SDK für PHP

> **Note**
>
> Es gibt noch mehr GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
    public function invoke($functionName, $params, $logType = 'None')
    {
        return $this->lambdaClient->invoke([
            'FunctionName' => $functionName,
            'Payload' => json_encode($params),
            'LogType' => $logType,
        ]);
    }
```

- Weitere API-Informationen finden Sie unter Invoke in der AWS SDK für PHP -API-Referenz.

Python

SDK für Python (Boto3)

> **Note**
>
> Es gibt noch mehr GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
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
```

- Weitere API-Informationen finden Sie unter Invoke in der API-Referenz zum AWS -SDK für Python (Boto3).

Ruby

SDK für Ruby

> **Note**
>
> Es gibt noch mehr GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
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
```

- Weitere API-Informationen finden Sie unter Invoke in der AWS SDK für Ruby -API-Referenz.

Rust

SDK für Rust

> **Note**
>
> Es gibt noch mehr dazu GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
    /** 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")
    }
}
```

- Weitere API-Informationen finden Sie unter Aufrufen in der API-Referenz zum AWS -SDK für Rust.

SAP ABAP

SDK für SAP ABAP

> **Note**
>
> Es gibt noch mehr dazu. GitHub Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
    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.
```

- Weitere API-Informationen finden Sie unter Invoke (Aufrufen) in der API-Referenz für das AWS -SDK für SAP ABAP.

Swift

SDK für Swift

> **Note**
>
> Es gibt noch mehr dazu GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
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
        }
    }
```

- Detaillierte API-Informationen finden Sie unter Aufrufen in der API-Referenz zum AWS SDK für Swift.

Eine vollständige Liste der AWS SDK-Entwicklerhandbücher und Codebeispiele finden Sie unterLambda mit einem AWS SDK verwenden. Dieses Thema enthält auch Informationen zu den ersten Schritten und Details zu früheren SDK-Versionen.

Verwendung **`ListFunctions`** mit einem AWS SDK oder CLI

Die folgenden Code-Beispiele zeigen, wie `ListFunctions` verwendet wird.

Beispiele für Aktionen sind Codeauszüge aus größeren Programmen und müssen im Kontext ausgeführt werden. Im folgenden Codebeispiel können Sie diese Aktion im Kontext sehen:

- Kennenlernen der Grundlagen

.NET

SDK für .NET

> **Note**
>
> Es gibt noch mehr dazu GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
/// <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;
}
```

- Einzelheiten zur API finden Sie ListFunctionsin der AWS SDK für .NET API-Referenz.

C++

SDK für C++

> **Note**
>
> Es gibt noch mehr dazu GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
        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());
```

- Einzelheiten zur API finden Sie ListFunctionsin der AWS SDK für C++ API-Referenz.

CLI

AWS CLI

Eine Liste der Lambda-Funktionen abrufen

Im folgenden Beispiel für `list-functions` wird eine Liste aller Funktionen für den aktuellen Benutzer angezeigt.

```
aws lambda list-functions
```

Ausgabe:

```
{
    "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": ""
        }
    ]
}
```

Weitere Informationen finden Sie unter Konfigurieren des Speichers der Lambda-Funktion im AWS -Lambda-Entwicklerhandbuch.

- Einzelheiten zur API finden Sie ListFunctionsin der AWS CLI Befehlsreferenz.

Go

SDK für Go V2

> **Note**
>
> Es gibt noch mehr dazu GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
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
}
```

- Einzelheiten zur API finden Sie ListFunctionsin der AWS SDK für Go API-Referenz.

JavaScript

SDK für JavaScript (v3)

> **Note**
>
> Es gibt noch mehr dazu GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
const listFunctions = () => {
  const client = new LambdaClient({});
  const command = new ListFunctionsCommand({});

  return client.send(command);
};
```

- Einzelheiten zur API finden Sie ListFunctionsin der AWS SDK für JavaScript API-Referenz.

PHP

SDK für PHP

> **Note**
>
> Es gibt noch mehr dazu GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
    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,
        ]);
    }
```

- Einzelheiten zur API finden Sie ListFunctionsin der AWS SDK für PHP API-Referenz.

PowerShell

Tools für PowerShell V4

Beispiel 1: In diesem Beispiel werden alle Lambda-Funktionen mit sortierter Codegröße angezeigt

```
Get-LMFunctionList | Sort-Object -Property CodeSize | Select-Object FunctionName,
 RunTime, Timeout, CodeSize
```

Ausgabe:

```
FunctionName                                                  Runtime   Timeout
 CodeSize
```

```
------------                                                    -------   -------
 --------
test                                                            python2.7       3
   243
MylambdaFunction123                                             python3.8     600
   659
myfuncpython1                                                   python3.8     303
   675
```

- Einzelheiten zur API finden Sie unter ListFunctions AWS -Tools für PowerShellCmdlet-Referenz (V4).

Tools für V5 PowerShell

Beispiel 1: In diesem Beispiel werden alle Lambda-Funktionen mit sortierter Codegröße angezeigt

```
Get-LMFunctionList | Sort-Object -Property CodeSize | Select-Object FunctionName,
 RunTime, Timeout, CodeSize
```

Ausgabe:

```
FunctionName                                                    Runtime   Timeout
 CodeSize
------------                                                    -------   -------
 --------
test                                                            python2.7       3
   243
MylambdaFunction123                                             python3.8     600
   659
myfuncpython1                                                   python3.8     303
   675
```

- Einzelheiten zur API finden Sie unter ListFunctions AWS -Tools für PowerShellCmdlet-Referenz (V5).

Python

SDK für Python (Boto3)

> **Note**
>
> Es gibt noch mehr dazu. GitHub Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
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
```

- Einzelheiten zur API finden Sie ListFunctionsin AWS SDK for Python (Boto3) API Reference.

Ruby

SDK für Ruby

> **Note**
>
> Es gibt noch mehr dazu. GitHub Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
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
```

- Einzelheiten zur API finden Sie ListFunctionsin der AWS SDK für Ruby API-Referenz.

Rust

SDK für Rust

> **Note**
>
> Es gibt noch mehr dazu GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
   /** 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)
   }
```

- Einzelheiten zur API finden Sie ListFunctionsin der API-Referenz zum AWS SDK für Rust.

SAP ABAP

SDK für SAP ABAP

> **Note**
>
> Es gibt noch mehr dazu. GitHub Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
   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.
```

- Einzelheiten zur API finden Sie ListFunctionsin der API-Referenz zum AWS SDK für SAP ABAP.

Swift

SDK für Swift

> **Note**
>
> Es gibt noch mehr dazu GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
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
    }
```

- Einzelheiten zur API finden Sie ListFunctionsin der API-Referenz zum AWS SDK für Swift.

Eine vollständige Liste der AWS SDK-Entwicklerhandbücher und Codebeispiele finden Sie unterLambda mit einem AWS SDK verwenden. Dieses Thema enthält auch Informationen zu den ersten Schritten und Details zu früheren SDK-Versionen.

Verwendung von **`ListProvisionedConcurrencyConfigs`** mit einer CLI

Die folgenden Code-Beispiele zeigen, wie `ListProvisionedConcurrencyConfigs` verwendet wird.

CLI

AWS CLI

So erhalten Sie eine Liste der bereitgestellten Gleichzeitigkeitskonfigurationen

Das folgende `list-provisioned-concurrency-configs`-Beispiel listet die bereitgestellten Gleichzeitigkeitskonfigurationen für die angegebene Funktion auf.

```
aws lambda list-provisioned-concurrency-configs \
    --function-name my-function
```

Ausgabe:

```
{
    "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"
        }
    ]
}
```

- Einzelheiten zur API finden Sie ListProvisionedConcurrencyConfigsin der AWS CLI Befehlsreferenz.

PowerShell

Tools für PowerShell V4

Beispiel 1: In diesem Beispiel wird die Liste der bereitgestellten Gleichzeitigkeitskonfigurationen für eine Lambda-Funktion abgerufen.

```
Get-LMProvisionedConcurrencyConfigList -FunctionName "MylambdaFunction123"
```

- Einzelheiten zur API finden Sie unter ListProvisionedConcurrencyConfigs AWS -Tools für PowerShellCmdlet-Referenz (V4).

Tools für V5 PowerShell

Beispiel 1: In diesem Beispiel wird die Liste der bereitgestellten Gleichzeitigkeitskonfigurationen für eine Lambda-Funktion abgerufen.

```
Get-LMProvisionedConcurrencyConfigList -FunctionName "MylambdaFunction123"
```

- Einzelheiten zur API finden Sie unter ListProvisionedConcurrencyConfigs AWS -Tools für PowerShellCmdlet-Referenz (V5).

Eine vollständige Liste der AWS SDK-Entwicklerhandbücher und Codebeispiele finden Sie unter. Lambda mit einem AWS SDK verwenden Dieses Thema enthält auch Informationen zu den ersten Schritten und Details zu früheren SDK-Versionen.

Verwendung von **`ListTags`** mit einer CLI

Die folgenden Code-Beispiele zeigen, wie `ListTags` verwendet wird.

CLI

AWS CLI

So rufen Sie die Liste der Tags für eine Lambda-Funktion ab

Das folgende `list-tags`-Beispiel zeigt die an die `my-function`-Lambda-Funktion angehängten Tags an.

```
aws lambda list-tags \
    --resource arn:aws:lambda:us-west-2:123456789012:function:my-function
```

Ausgabe:

```
{
    "Tags": {
        "Category": "Web Tools",
        "Department": "Sales"
    }
}
```

Weitere Informationen finden Sie unter Tagging von Lambda-Funktionen im AWS -Lambda-Entwicklerhandbuch.

- Einzelheiten zur API finden Sie ListTagsin der AWS CLI Befehlsreferenz.

PowerShell

Tools für PowerShell V4

Beispiel 1: Ruft die Tags und ihre Werte ab, die derzeit für die angegebene Funktion festgelegt sind.

```
Get-LMResourceTag -Resource "arn:aws:lambda:us-
west-2:123456789012:function:MyFunction"
```

Ausgabe:

```
Key        Value
---        -----
California Sacramento
Oregon     Salem
Washington Olympia
```

- Einzelheiten zur API finden Sie unter ListTags AWS -Tools für PowerShellCmdlet-Referenz (V4).

Tools für V5 PowerShell

Beispiel 1: Ruft die Tags und ihre Werte ab, die derzeit für die angegebene Funktion festgelegt sind.

```
Get-LMResourceTag -Resource "arn:aws:lambda:us-
west-2:123456789012:function:MyFunction"
```

Ausgabe:

```
Key        Value
---        -----
California Sacramento
Oregon     Salem
Washington Olympia
```

- Einzelheiten zur API finden Sie unter ListTags AWS -Tools für PowerShellCmdlet-Referenz (V5).

Eine vollständige Liste der AWS SDK-Entwicklerhandbücher und Codebeispiele finden Sie unter. Lambda mit einem AWS SDK verwenden Dieses Thema enthält auch Informationen zu den ersten Schritten und Details zu früheren SDK-Versionen.

Verwendung von **`ListVersionsByFunction`** mit einer CLI

Die folgenden Code-Beispiele zeigen, wie `ListVersionsByFunction` verwendet wird.

CLI

AWS CLI

So rufen Sie eine Liste der Versionen einer Funktion ab

Im folgenden `list-versions-by-function`-Beispiel wird die Liste der Versionen für die `my-function`-Lambda-Funktion angezeigt.

```
aws lambda list-versions-by-function \
    --function-name my-function
```

Ausgabe:

```
{
    "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"
        }
    ]
}
```

Weitere Informationen finden Sie unter Konfiguration von AWS Lambda-Funktionsaliasen im AWS Lambda Developer Guide.

- Einzelheiten zur API finden Sie unter Befehlsreferenz ListVersionsByFunction.AWS CLI

PowerShell

Tools für PowerShell V4

Beispiel 1: Dieses Beispiel gibt die Liste der versionsspezifischen Konfigurationen für jede Version der Lambda-Funktion zurück.

```
Get-LMVersionsByFunction -FunctionName "MylambdaFunction123"
```

Ausgabe:

```
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
```

- Einzelheiten zur API finden Sie unter ListVersionsByFunction AWS -Tools für PowerShellCmdlet-Referenz (V4).

Tools für V5 PowerShell

Beispiel 1: Dieses Beispiel gibt die Liste der versionsspezifischen Konfigurationen für jede Version der Lambda-Funktion zurück.

```
Get-LMVersionsByFunction -FunctionName "MylambdaFunction123"
```

Ausgabe:

```
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
```

- Einzelheiten zur API finden Sie unter ListVersionsByFunction AWS -Tools für PowerShellCmdlet-Referenz (V5).

Eine vollständige Liste der AWS SDK-Entwicklerhandbücher und Codebeispiele finden Sie unter. Lambda mit einem AWS SDK verwenden Dieses Thema enthält auch Informationen zu den ersten Schritten und Details zu früheren SDK-Versionen.

Verwendung von **`PublishVersion`** mit einer CLI

Die folgenden Code-Beispiele zeigen, wie `PublishVersion` verwendet wird.

CLI

AWS CLI

So veröffentlichen Sie eine neue Version einer Funktion

Im folgenden Beispiel für `publish-version` wird eine neue Version der `my-function`-Lambda-Funktion veröffentlicht.

```
aws lambda publish-version \
    --function-name my-function
```

Ausgabe:

```
{
    "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": ""
}
```

Weitere Informationen finden Sie unter Konfiguration von AWS Lambda-Funktionsaliasen im AWS Lambda Developer Guide.

- Einzelheiten zur API finden Sie unter Befehlsreferenz PublishVersion.AWS CLI

PowerShell

Tools für PowerShell V4

Beispiel 1: In diesem Beispiel wird eine Version für den vorhandenen Snapshot von Lambda-Funktionscode erstellt

```
Publish-LMVersion -FunctionName "MylambdaFunction123" -Description "Publishing
 Existing Snapshot of function code as a  new version through Powershell"
```

- Einzelheiten zur API finden Sie unter PublishVersion AWS -Tools für PowerShellCmdlet-Referenz (V4).

Tools für V5 PowerShell

Beispiel 1: In diesem Beispiel wird eine Version für den vorhandenen Snapshot von Lambda-Funktionscode erstellt

```
Publish-LMVersion -FunctionName "MylambdaFunction123" -Description "Publishing
 Existing Snapshot of function code as a  new version through Powershell"
```

- Einzelheiten zur API finden Sie unter PublishVersion AWS -Tools für PowerShellCmdlet-Referenz (V5).

Eine vollständige Liste der AWS SDK-Entwicklerhandbücher und Codebeispiele finden Sie unter. Lambda mit einem AWS SDK verwenden Dieses Thema enthält auch Informationen zu den ersten Schritten und Details zu früheren SDK-Versionen.

Verwendung von **`PutFunctionConcurrency`** mit einer CLI

Die folgenden Code-Beispiele zeigen, wie `PutFunctionConcurrency` verwendet wird.

CLI

AWS CLI

So konfigurieren Sie eine reservierte Gleichzeitigkeitsgrenze für eine Funktion

Im folgenden `put-function-concurrency`-Beispiel werden 100 reservierte gleichzeitige Ausführungen für die `my-function`-Funktion konfiguriert.

```
aws lambda put-function-concurrency \
```

```
    --function-name  my-function  \
    --reserved-concurrent-executions 100
```

Ausgabe:

```
{
    "ReservedConcurrentExecutions": 100
}
```

Weitere Informationen finden Sie unter Gleichzeitigkeit für eine Lambda-Funktion reservieren im AWS -Lambda-Entwicklerhandbuch.

- Einzelheiten zur API finden Sie PutFunctionConcurrencyin der AWS CLI Befehlsreferenz.

PowerShell

Tools für PowerShell V4

Beispiel 1: In diesem Beispiel werden die Gleichzeitigkeitseinstellungen für die Funktion als Ganzes angewendet.

```
Write-LMFunctionConcurrency -FunctionName "MylambdaFunction123" -
ReservedConcurrentExecution 100
```

- Einzelheiten zur API finden Sie unter PutFunctionConcurrency AWS -Tools für PowerShellCmdlet-Referenz (V4).

Tools für V5 PowerShell

Beispiel 1: In diesem Beispiel werden die Gleichzeitigkeitseinstellungen für die Funktion als Ganzes angewendet.

```
Write-LMFunctionConcurrency -FunctionName "MylambdaFunction123" -
ReservedConcurrentExecution 100
```

- Einzelheiten zur API finden Sie unter PutFunctionConcurrency AWS -Tools für PowerShellCmdlet-Referenz (V5).

Eine vollständige Liste der AWS SDK-Entwicklerhandbücher und Codebeispiele finden Sie unter. Lambda mit einem AWS SDK verwenden Dieses Thema enthält auch Informationen zu den ersten Schritten und Details zu früheren SDK-Versionen.

Verwendung von **`PutProvisionedConcurrencyConfig`** mit einer CLI

Die folgenden Code-Beispiele zeigen, wie `PutProvisionedConcurrencyConfig` verwendet wird.

CLI

AWS CLI

So weisen Sie bereitgestellte Gleichzeitigkeit zu

Das folgende `put-provisioned-concurrency-config`-Beispiel weist dem BLUE-Alias der angegebenen Funktion 100 bereitgestellte Gleichzeitigkeit zu.

```
aws lambda put-provisioned-concurrency-config \
    --function-name my-function \
    --qualifier BLUE \
    --provisioned-concurrent-executions 100
```

Ausgabe:

```
{
    "Requested ProvisionedConcurrentExecutions": 100,
    "Allocated ProvisionedConcurrentExecutions": 0,
    "Status": "IN_PROGRESS",
    "LastModified": "2019-11-21T19:32:12+0000"
}
```

- Einzelheiten zur API finden Sie PutProvisionedConcurrencyConfigin der AWS CLI Befehlsreferenz.

PowerShell

Tools für PowerShell V4

Beispiel 1: In diesem Beispiel wird dem Alias einer Funktion eine bereitgestellte Gleichzeitigkeitskonfiguration hinzugefügt

```
Write-LMProvisionedConcurrencyConfig -FunctionName "MylambdaFunction123" -
ProvisionedConcurrentExecution 20 -Qualifier "NewAlias1"
```

- Einzelheiten zur API finden Sie unter PutProvisionedConcurrencyConfig AWS -Tools für PowerShellCmdlet-Referenz (V4).

Tools für V5 PowerShell

Beispiel 1: In diesem Beispiel wird dem Alias einer Funktion eine bereitgestellte Gleichzeitigkeitskonfiguration hinzugefügt

```
Write-LMProvisionedConcurrencyConfig -FunctionName "MylambdaFunction123" -
ProvisionedConcurrentExecution 20 -Qualifier "NewAlias1"
```

- Einzelheiten zur API finden Sie unter PutProvisionedConcurrencyConfig AWS -Tools für PowerShellCmdlet-Referenz (V5).

Eine vollständige Liste der AWS SDK-Entwicklerhandbücher und Codebeispiele finden Sie unter. Lambda mit einem AWS SDK verwenden Dieses Thema enthält auch Informationen zu den ersten Schritten und Details zu früheren SDK-Versionen.

Verwendung von **`RemovePermission`** mit einer CLI

Die folgenden Code-Beispiele zeigen, wie `RemovePermission` verwendet wird.

CLI

AWS CLI

So entfernen Sie Berechtigungen aus einer vorhandenen Lambda-Funktion

Das folgende `remove-permission`-Beispiel entfernt die Berechtigung zum Aufrufen einer Funktion namens `my-function`.

```
aws lambda remove-permission \
    --function-name my-function \
    --statement-id sns
```

Mit diesem Befehl wird keine Ausgabe zurückgegeben.

Weitere Informationen finden Sie unter Using Resource-based Policies for AWS Lambda im Lambda Developer Guide AWS .

- Einzelheiten zur API finden Sie unter RemovePermissionBefehlsreferenz.AWS CLI

PowerShell

Tools für PowerShell V4

Beispiel 1: In diesem Beispiel wird die Funktionsrichtlinie für die angegebene StatementId Lambda-Funktion entfernt.

```
$policy =  Get-LMPolicy -FunctionName "MylambdaFunction123" -Select Policy |
 ConvertFrom-Json| Select-Object -ExpandProperty Statement
Remove-LMPermission -FunctionName "MylambdaFunction123" -StatementId
 $policy[0].Sid
```

- Einzelheiten zur API finden Sie unter RemovePermission AWS -Tools für PowerShellCmdlet-Referenz (V4).

Tools für V5 PowerShell

Beispiel 1: In diesem Beispiel wird die Funktionsrichtlinie für die angegebene StatementId Lambda-Funktion entfernt.

```
$policy =  Get-LMPolicy -FunctionName "MylambdaFunction123" -Select Policy |
 ConvertFrom-Json| Select-Object -ExpandProperty Statement
Remove-LMPermission -FunctionName "MylambdaFunction123" -StatementId
 $policy[0].Sid
```

- Einzelheiten zur API finden Sie unter RemovePermission AWS -Tools für PowerShellCmdlet-Referenz (V5).

Eine vollständige Liste der AWS SDK-Entwicklerhandbücher und Codebeispiele finden Sie unter. Lambda mit einem AWS SDK verwenden Dieses Thema enthält auch Informationen zu den ersten Schritten und Details zu früheren SDK-Versionen.

Verwendung von **`TagResource`** mit einer CLI

Die folgenden Code-Beispiele zeigen, wie `TagResource` verwendet wird.

CLI

AWS CLI

So fügen Sie einer vorhandenen Lambda-Funktion Tags hinzu

Das folgende `tag-resource`-Beispiel fügt der angegebenen Lambda-Funktion ein Tag mit dem Schlüsselnamen `DEPARTMENT` und einem Wert von `Department A` hinzu.

```
aws lambda tag-resource \
    --resource arn:aws:lambda:us-west-2:123456789012:function:my-function \
    --tags "DEPARTMENT=Department A"
```

Mit diesem Befehl wird keine Ausgabe zurückgegeben.

Weitere Informationen finden Sie unter Tagging von Lambda-Funktionen im AWS -Lambda-Entwicklerhandbuch.

- Einzelheiten zur API finden Sie TagResourcein der AWS CLI Befehlsreferenz.

PowerShell

Tools für PowerShell V4

Beispiel 1: Fügt die drei Tags (Washington, Oregon und Kalifornien) und ihre zugehörigen Werte zu der angegebenen Funktion hinzu, die durch ihre ARN identifiziert wird.

```
Add-LMResourceTag -Resource "arn:aws:lambda:us-
west-2:123456789012:function:MyFunction" -Tag @{ "Washington" = "Olympia";
 "Oregon" = "Salem"; "California" = "Sacramento" }
```

- Einzelheiten zur API finden Sie unter TagResource AWS -Tools für PowerShellCmdlet-Referenz (V4).

Tools für V5 PowerShell

Beispiel 1: Fügt die drei Tags (Washington, Oregon und Kalifornien) und ihre zugehörigen Werte zu der angegebenen Funktion hinzu, die durch ihre ARN identifiziert wird.

```
Add-LMResourceTag -Resource "arn:aws:lambda:us-
west-2:123456789012:function:MyFunction" -Tag @{ "Washington" = "Olympia";
 "Oregon" = "Salem"; "California" = "Sacramento" }
```

- Einzelheiten zur API finden Sie unter TagResource AWS -Tools für PowerShellCmdlet-Referenz (V5).

Eine vollständige Liste der AWS SDK-Entwicklerhandbücher und Codebeispiele finden Sie unter. Lambda mit einem AWS SDK verwenden Dieses Thema enthält auch Informationen zu den ersten Schritten und Details zu früheren SDK-Versionen.

Verwendung von **`UntagResource`** mit einer CLI

Die folgenden Code-Beispiele zeigen, wie `UntagResource` verwendet wird.

CLI

AWS CLI

So entfernen Sie Tags aus einer vorhandenen Lambda-Funktion

Das folgende `untag-resource`-Beispiel entfernt das Tag mit dem Schlüsselnamen `DEPARTMENT`-Tag aus der Lambda-Funktion `my-function`.

```
aws lambda untag-resource \
    --resource arn:aws:lambda:us-west-2:123456789012:function:my-function \
    --tag-keys DEPARTMENT
```

Mit diesem Befehl wird keine Ausgabe zurückgegeben.

Weitere Informationen finden Sie unter Tagging von Lambda-Funktionen im AWS -Lambda-Entwicklerhandbuch.

- Einzelheiten zur API finden Sie UntagResourcein der AWS CLI Befehlsreferenz.

PowerShell

Tools für PowerShell V4

Beispiel 1: Entfernt die angegebenen Tags aus einer Funktion. Das Cmdlet fordert Sie zur Bestätigung auf, bevor der Vorgang fortgesetzt wird, sofern der Schalter -Force nicht angegeben ist. Es wird ein einziger Aufruf an den Dienst gesendet, um die Tags zu entfernen.

```
Remove-LMResourceTag -Resource "arn:aws:lambda:us-
west-2:123456789012:function:MyFunction" -TagKey
 "Washington","Oregon","California"
```

Beispiel 2: Entfernt die angegebenen Tags aus einer Funktion. Das Cmdlet fordert Sie zur Bestätigung auf, bevor der Vorgang fortgesetzt wird, sofern der Schalter -Force nicht angegeben ist. Sobald der Dienst pro bereitgestelltem Tag aufgerufen wurde.

```
"Washington","Oregon","California" | Remove-LMResourceTag -Resource
 "arn:aws:lambda:us-west-2:123456789012:function:MyFunction"
```

- Einzelheiten zur API finden Sie unter UntagResource AWS -Tools für PowerShellCmdlet-Referenz (V4).

Tools für V5 PowerShell

Beispiel 1: Entfernt die angegebenen Tags aus einer Funktion. Das Cmdlet fordert Sie zur Bestätigung auf, bevor der Vorgang fortgesetzt wird, sofern der Schalter -Force nicht angegeben ist. Es wird ein einziger Aufruf an den Dienst gesendet, um die Tags zu entfernen.

```
Remove-LMResourceTag -Resource "arn:aws:lambda:us-
west-2:123456789012:function:MyFunction" -TagKey
 "Washington","Oregon","California"
```

Beispiel 2: Entfernt die angegebenen Tags aus einer Funktion. Das Cmdlet fordert Sie zur Bestätigung auf, bevor der Vorgang fortgesetzt wird, sofern der Schalter -Force nicht angegeben ist. Sobald der Dienst pro bereitgestelltem Tag aufgerufen wurde.

```
"Washington","Oregon","California" | Remove-LMResourceTag -Resource
 "arn:aws:lambda:us-west-2:123456789012:function:MyFunction"
```

- Einzelheiten zur API finden Sie unter UntagResource AWS -Tools für PowerShellCmdlet-Referenz (V5).

Eine vollständige Liste der AWS SDK-Entwicklerhandbücher und Codebeispiele finden Sie unter. Lambda mit einem AWS SDK verwenden Dieses Thema enthält auch Informationen zu den ersten Schritten und Details zu früheren SDK-Versionen.

Verwendung von **`UpdateAlias`** mit einer CLI

Die folgenden Code-Beispiele zeigen, wie `UpdateAlias` verwendet wird.

CLI

AWS CLI

So aktualisieren Sie einen Funktionsalias

Das folgende `update-alias`-Beispiel aktualisiert den Alias mit dem Namen `LIVE` so, dass er auf Version 3 der `my-function`-Lambda-Funktion verweist.

```
aws lambda update-alias \
    --function-name my-function \
    --function-version 3 \
    --name LIVE
```

Ausgabe:

```
{
    "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"
}
```

Weitere Informationen finden Sie unter Konfiguration von AWS Lambda-Funktionsaliasen im AWS Lambda Developer Guide.

- Einzelheiten zur API finden Sie unter Befehlsreferenz UpdateAlias.AWS CLI

PowerShell

Tools für PowerShell V4

Beispiel 1: Dieses Beispiel aktualisiert die Konfiguration einer vorhandenen Lambda-Funktion-Alias. Der RoutingConfiguration Wert wird aktualisiert, sodass 60% (0,6) des Datenverkehrs auf Version 1 umgestellt werden

```
Update-LMAlias -FunctionName "MylambdaFunction123" -Description
 " Alias for version 2" -FunctionVersion 2 -Name "newlabel1" -
RoutingConfig_AdditionalVersionWeight @{Name="1";Value="0.6}
```

- Einzelheiten zur API finden Sie unter UpdateAlias AWS -Tools für PowerShellCmdlet-Referenz (V4).

Tools für V5 PowerShell

Beispiel 1: Dieses Beispiel aktualisiert die Konfiguration einer vorhandenen Lambda-Funktion-Alias. Der RoutingConfiguration Wert wird aktualisiert, sodass 60% (0,6) des Datenverkehrs auf Version 1 umgestellt werden

```
Update-LMAlias -FunctionName "MylambdaFunction123" -Description
 " Alias for version 2" -FunctionVersion 2 -Name "newlabel1" -
RoutingConfig_AdditionalVersionWeight @{Name="1";Value="0.6}
```

- Einzelheiten zur API finden Sie unter UpdateAlias AWS -Tools für PowerShellCmdlet-Referenz (V5).

Eine vollständige Liste der AWS SDK-Entwicklerhandbücher und Codebeispiele finden Sie unter. Lambda mit einem AWS SDK verwenden Dieses Thema enthält auch Informationen zu den ersten Schritten und Details zu früheren SDK-Versionen.

Verwendung **`UpdateFunctionCode`** mit einem AWS SDK oder CLI

Die folgenden Code-Beispiele zeigen, wie `UpdateFunctionCode` verwendet wird.

Beispiele für Aktionen sind Codeauszüge aus größeren Programmen und müssen im Kontext ausgeführt werden. Im folgenden Codebeispiel können Sie diese Aktion im Kontext sehen:

- Kennenlernen der Grundlagen

.NET

SDK für .NET

> **Note**
>
> Es gibt noch mehr dazu GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
    /// <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}.");
    }
```

- Einzelheiten zur API finden Sie UpdateFunctionCodein der AWS SDK für .NET API-Referenz.

C++

SDK für C++

> **Note**
>
> Es gibt noch mehr dazu GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
        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;
        }
```

- Einzelheiten zur API finden Sie UpdateFunctionCodein der AWS SDK für C++ API-Referenz.

CLI

AWS CLI

Den Code einer Lambda-Funktion aktualisieren

Im folgenden Beispiel für `update-function-code` wird der Code der unveröffentlichten Version ($LATEST) der Funktion `my-function` durch den Inhalt der angegebenen ZIP-Datei ersetzt.

```
aws lambda update-function-code \
    --function-name  my-function \
    --zip-file fileb://my-function.zip
```

Ausgabe:

```
{
    "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"
}
```

Weitere Informationen finden Sie unter Konfigurieren von AWS -Lambda-Funktionen im AWS -Lambda-Entwicklerhandbuch.

- Einzelheiten zur API finden Sie UpdateFunctionCodein der AWS CLI Befehlsreferenz.

Go

SDK für Go V2

> **Note**
>
> Es gibt noch mehr dazu GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
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
}
```

- Einzelheiten zur API finden Sie UpdateFunctionCodein der AWS SDK für Go API-Referenz.

Java

SDK für Java 2.x

> **Note**
>
> Es gibt noch mehr dazu GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
   /**
    * 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);
        }
    }
```

- Einzelheiten zur API finden Sie UpdateFunctionCodein der AWS SDK for Java 2.x API-Referenz.

JavaScript

SDK für JavaScript (v3)

> **Note**
>
> Es gibt noch mehr dazu GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
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);
};
```

- Einzelheiten zur API finden Sie UpdateFunctionCodein der AWS SDK für JavaScript API-Referenz.

PHP

SDK für PHP

> **Note**
>
> Es gibt noch mehr dazu GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
public function updateFunctionCode($functionName, $s3Bucket, $s3Key)
{
    return $this->lambdaClient->updateFunctionCode([
        'FunctionName' => $functionName,
        'S3Bucket' => $s3Bucket,
        'S3Key' => $s3Key,
    ]);
}
```

- Einzelheiten zur API finden Sie UpdateFunctionCodein der AWS SDK für PHP API-Referenz.

PowerShell

Tools für PowerShell V4

Beispiel 1: Aktualisiert die Funktion mit dem Namen 'MyFunction' mit neuem Inhalt, der in der angegebenen ZIP-Datei enthalten ist. Für eine C#.NET-Core-Lambda-Funktion sollte die ZIP-Datei die kompilierte Assembly enthalten.

```
Update-LMFunctionCode -FunctionName MyFunction -ZipFilename .\UpdatedCode.zip
```

Beispiel 2: Dieses Beispiel ähnelt dem vorherigen, verwendet aber ein Amazon S3-Objekt, das den aktualisierten Code enthält, um die Funktion zu aktualisieren.

```
Update-LMFunctionCode -FunctionName MyFunction -BucketName amzn-s3-demo-bucket -
Key UpdatedCode.zip
```

- Einzelheiten zur API finden Sie unter UpdateFunctionCode AWS -Tools für PowerShellCmdlet-Referenz (V4).

Tools für V5 PowerShell

Beispiel 1: Aktualisiert die Funktion mit dem Namen 'MyFunction' mit neuem Inhalt, der in der angegebenen ZIP-Datei enthalten ist. Für eine C#.NET-Core-Lambda-Funktion sollte die ZIP-Datei die kompilierte Assembly enthalten.

```
Update-LMFunctionCode -FunctionName MyFunction -ZipFilename .\UpdatedCode.zip
```

Beispiel 2: Dieses Beispiel ähnelt dem vorherigen, verwendet aber ein Amazon S3-Objekt, das den aktualisierten Code enthält, um die Funktion zu aktualisieren.

```
Update-LMFunctionCode -FunctionName MyFunction -BucketName amzn-s3-demo-bucket -
Key UpdatedCode.zip
```

- Einzelheiten zur API finden Sie unter UpdateFunctionCode AWS -Tools für PowerShellCmdlet-Referenz (V5).

Python

SDK für Python (Boto3)

> **Note**
>
> Es gibt noch mehr dazu. GitHub Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
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
```

- Einzelheiten zur API finden Sie UpdateFunctionCodein AWS SDK for Python (Boto3) API Reference.

Ruby

SDK für Ruby

> **Note**
>
> Es gibt noch mehr dazu. GitHub Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
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
```

- Einzelheiten zur API finden Sie UpdateFunctionCodein der AWS SDK für Ruby API-Referenz.

Rust

SDK für Rust

> **Note**
>
> Es gibt noch mehr dazu GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
    /** 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())
    }
```

- Einzelheiten zur API finden Sie UpdateFunctionCodein der API-Referenz zum AWS SDK für Rust.

SAP ABAP

SDK für SAP ABAP

> **Note**
>
> Es gibt noch mehr dazu. GitHub Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
    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.
```

- Einzelheiten zur API finden Sie UpdateFunctionCodein der API-Referenz zum AWS SDK für SAP ABAP.

Swift

SDK für Swift

> **Note**
>
> Es gibt noch mehr dazu GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
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
        }
```

- Einzelheiten zur API finden Sie UpdateFunctionCodein der API-Referenz zum AWS SDK für Swift.

Eine vollständige Liste der AWS SDK-Entwicklerhandbücher und Codebeispiele finden Sie unterLambda mit einem AWS SDK verwenden. Dieses Thema enthält auch Informationen zu den ersten Schritten und Details zu früheren SDK-Versionen.

Verwendung **`UpdateFunctionConfiguration`** mit einem AWS SDK oder CLI

Die folgenden Code-Beispiele zeigen, wie `UpdateFunctionConfiguration` verwendet wird.

Beispiele für Aktionen sind Codeauszüge aus größeren Programmen und müssen im Kontext ausgeführt werden. Im folgenden Codebeispiel können Sie diese Aktion im Kontext sehen:

- Kennenlernen der Grundlagen

.NET

SDK für .NET

> **Note**
>
> Es gibt noch mehr dazu GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
   /// <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;
    }
```

- Einzelheiten zur API finden Sie UpdateFunctionConfigurationin der AWS SDK für .NET API-Referenz.

C++

SDK für C++

> **Note**
>
> Es gibt noch mehr dazu GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
       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;
        }
```

- Einzelheiten zur API finden Sie UpdateFunctionConfigurationin der AWS SDK für C++ API-Referenz.

CLI

AWS CLI

Die Konfiguration einer Funktion ändern

Im folgenden Beispiel für `update-function-configuration` wird die Speichergröße für die unveröffentlichte Version ($LATEST) der Funktion `my-function` auf 256 MB geändert.

```
aws lambda update-function-configuration \
    --function-name  my-function \
    --memory-size 256
```

Ausgabe:

```
{
    "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"
}
```

Weitere Informationen finden Sie unter Konfigurieren von AWS -Lambda-Funktionen im AWS -Lambda-Entwicklerhandbuch.

- Einzelheiten zur API finden Sie UpdateFunctionConfigurationin der AWS CLI Befehlsreferenz.

Go

SDK für Go V2

> **Note**
>
> Es gibt noch mehr dazu GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
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)
 }
}
```

- Einzelheiten zur API finden Sie UpdateFunctionConfigurationin der AWS SDK für Go API-Referenz.

Java

SDK für Java 2.x

> **Note**
>
> Es gibt noch mehr dazu GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
    /**
     * 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);
        }
    }
```

- Einzelheiten zur API finden Sie UpdateFunctionConfigurationin der AWS SDK for Java 2.x API-Referenz.

JavaScript

SDK für JavaScript (v3)

> **Note**
>
> Es gibt noch mehr dazu GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
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;
};
```

- Einzelheiten zur API finden Sie UpdateFunctionConfigurationin der AWS SDK für JavaScript API-Referenz.

PHP

SDK für PHP

> **Note**
>
> Es gibt noch mehr dazu GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
   public function updateFunctionConfiguration($functionName, $handler,
$environment = '')
   {
       return $this->lambdaClient->updateFunctionConfiguration([
           'FunctionName' => $functionName,
           'Handler' => "$handler.lambda_handler",
           'Environment' => $environment,
       ]);
   }
```

- Einzelheiten zur API finden Sie UpdateFunctionConfigurationin der AWS SDK für PHP API-Referenz.

PowerShell

Tools für PowerShell V4

Beispiel 1: Dieses Beispiel aktualisiert die vorhandene Lambda-Funktionskonfiguration

```
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
```

- Einzelheiten zur API finden Sie unter UpdateFunctionConfiguration AWS -Tools für PowerShellCmdlet-Referenz (V4).

Tools für V5 PowerShell

Beispiel 1: Dieses Beispiel aktualisiert die vorhandene Lambda-Funktionskonfiguration

```
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
```

- Einzelheiten zur API finden Sie unter UpdateFunctionConfiguration AWS -Tools für PowerShellCmdlet-Referenz (V5).

Python

SDK für Python (Boto3)

> **Note**
>
> Es gibt noch mehr dazu. GitHub Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
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
```

- Einzelheiten zur API finden Sie UpdateFunctionConfigurationin AWS SDK for Python (Boto3) API Reference.

Ruby

SDK für Ruby

> **Note**
>
> Es gibt noch mehr dazu. GitHub Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
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
```

- Einzelheiten zur API finden Sie UpdateFunctionConfigurationin der AWS SDK für Ruby API-Referenz.

Rust

SDK für Rust

> **Note**
>
> Es gibt noch mehr dazu GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
/** 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)
}
```

- Einzelheiten zur API finden Sie UpdateFunctionConfigurationin der API-Referenz zum AWS SDK für Rust.

SAP ABAP

SDK für SAP ABAP

> **Note**
>
> Es gibt noch mehr dazu. GitHub Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
   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.
```

- Einzelheiten zur API finden Sie UpdateFunctionConfigurationin der API-Referenz zum AWS SDK für SAP ABAP.

Swift

SDK für Swift

> **Note**
>
> Es gibt noch mehr dazu GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

```
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
    }
}
```

- Einzelheiten zur API finden Sie UpdateFunctionConfigurationin der API-Referenz zum AWS SDK für Swift.

Eine vollständige Liste der AWS SDK-Entwicklerhandbücher und Codebeispiele finden Sie unterLambda mit einem AWS SDK verwenden. Dieses Thema enthält auch Informationen zu den ersten Schritten und Details zu früheren SDK-Versionen.

Szenarien für die Verwendung von Lambda AWS SDKs

Die folgenden Codebeispiele zeigen Ihnen, wie Sie gängige Szenarien in Lambda mit AWS SDKs implementieren. Diese Szenarien zeigen Ihnen, wie Sie bestimmte Aufgaben erledigen, indem Sie mehrere Funktionen innerhalb von Lambda oder in Kombination mit anderen AWS-Services aufrufen.

Jedes Szenario enthält einen Link zum vollständigen Quell-Code, wo Sie Anleitungen zum Einrichten und Ausführen des Codes finden.

Szenarien zielen auf eine mittlere Erfahrungsebene ab, um Ihnen zu helfen, Service-Aktionen im Kontext zu verstehen.

Beispiele

- Bestätigen Sie bekannte Amazon Cognito Cognito-Benutzer automatisch mit einer Lambda-Funktion mithilfe eines SDK AWS
- Automatisches Migrieren bekannter Amazon Cognito Cognito-Benutzer mit einer Lambda-Funktion mithilfe eines SDK AWS
- Erstellen einer API-Gateway-REST-API zur Verfolgung von COVID-19-Daten
- Leihbibliothek-REST-API erstellen
- Erstellen einer Messenger-Anwendung mit Step Functions
- Eine Anwendung für Foto-Asset-Management erstellen, mit der Benutzer Fotos mithilfe von Labels verwalten können
- Erstellen einer Websocket-Chat-Anwendung mit API Gateway
- Erstellen einer Anwendung, die Kundenfeedback analysiert und Audio generiert
- Aufrufen einer Lambda-Funktion von einem Browser aus
- Transformieren von Daten für Ihre Anwendung mit S3 Object Lambda
- Verwenden von API Gateway zum Aufrufen einer Lambda-Funktion
- Verwenden von Step Functions, um Lambda-Funktionen aufzurufen
- Verwendung geplanter Ereignisse zum Aufrufen einer Lambda-Funktion
- Verwenden der Amazon Neptune-API zur Entwicklung einer Lambda-Funktion, die Graphdaten abfragt
- Schreiben Sie benutzerdefinierte Aktivitätsdaten mit einer Lambda-Funktion nach der Amazon Cognito Cognito-Benutzerauthentifizierung mithilfe eines SDK AWS

Bestätigen Sie bekannte Amazon Cognito Cognito-Benutzer automatisch mit einer Lambda-Funktion mithilfe eines SDK AWS

Die folgenden Codebeispiele zeigen, wie bekannte Amazon Cognito-Benutzer mit einer Lambda-Funktion automatisch bestätigt werden können.

- Konfigurieren Sie einen Benutzerpool, um eine Lambda-Funktion für den `PreSignUp`-Trigger aufzurufen.
- Registrieren eines Benutzers bei Amazon Cognito.
- Die Lambda-Funktion scannt eine DynamoDB-Tabelle und bestätigt automatisch bekannte Benutzer.
- Melden Sie sich als neuer Benutzer an und bereinigen Sie dann die Ressourcen.

Go

SDK für Go V2

> **Note**
>
> Es gibt noch mehr dazu GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

Führen Sie ein interaktives Szenario an einem Prompt aus.

```
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))
}
```

Behandeln Sie den `PreSignUp`-Trigger mit einer Lambda-Funktion.

```
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)
}
```

Erstellen Sie eine Struktur, die allgemeine Aufgaben ausführt.

```
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))
}
```

Erstellen Sie eine Struktur, die Amazon Cognito-Aktionen umschließt.

```
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
}
```

Erstellen Sie eine Struktur, die DynamoDB-Aktionen umhüllt.

```
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
}
```

Erstellen Sie eine Struktur, die CloudWatch Logs-Aktionen umschließt.

```
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
}
```

Erstellen Sie eine Struktur, die Aktionen umschließt. 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
}
```

Bereinigen von Ressourcen.

```
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!")
 }
}
```

- Weitere API-Informationen finden Sie in den folgenden Themen der AWS SDK für Go -API-Referenz.
  - DeleteUser
  - InitiateAuth
  - SignUp
  - UpdateUserPool

JavaScript

SDK für JavaScript (v3)

> **Note**
>
> Es gibt noch mehr dazu GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

Konfigurieren Sie einen interaktiven „Szenario“-Lauf. Die JavaScript (v3) -Beispiele verwenden gemeinsam einen Szenario-Runner, um komplexe Beispiele zu vereinfachen. Der komplette Quellcode ist aktiviert 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.",
  });
}
```

Dieses Szenario demonstriert die automatische Bestätigung eines bekannten Benutzers. Es orchestriert die Beispielschritte.

```
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,
  );
```

Dies sind Schritte, die mit anderen Szenarien geteilt werden.

```
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 },
);
```

Ein Handler für den `PreSignUp`-Trigger mit einer Lambda-Funktion.

```
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);
};
```

Modul zur CloudWatch Protokollierung von Aktionen.

```
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];
  }
};
```

Modul für Amazon-Cognito-Aktionen.

```
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];
  }
};
```

Modul für DynamoDB-Aktionen.

```
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];
  }
};
```

- Weitere API-Informationen finden Sie in den folgenden Themen der AWS SDK für JavaScript -API-Referenz.
  - DeleteUser
  - InitiateAuth
  - SignUp
  - UpdateUserPool

Eine vollständige Liste der AWS SDK-Entwicklerhandbücher und Codebeispiele finden Sie unterLambda mit einem AWS SDK verwenden. Dieses Thema enthält auch Informationen zu den ersten Schritten und Details zu früheren SDK-Versionen.

Automatisches Migrieren bekannter Amazon Cognito Cognito-Benutzer mit einer Lambda-Funktion mithilfe eines SDK AWS

Das folgende Codebeispiel zeigt, wie bekannte Amazon Cognito-Benutzer mit einer Lambda-Funktion automatisch migriert werden können.

- Konfigurieren Sie einen Benutzerpool, um eine Lambda-Funktion für den `MigrateUser`-Trigger aufzurufen.
- Melden Sie sich bei Amazon Cognito mit einem Benutzernamen und einer E-Mail-Adresse an, die sich nicht im Benutzerpool befinden.
- Die Lambda-Funktion durchsucht eine DynamoDB-Tabelle und migriert bekannte Benutzer automatisch in den Benutzerpool.
- Führen Sie den Ablauf „Passwort vergessen“ aus, um das Passwort für den migrierten Benutzer zurückzusetzen.
- Melden Sie sich als neuer Benutzer an und bereinigen Sie dann die Ressourcen.

Go

SDK für Go V2

> **Note**
>
> Es gibt noch mehr dazu GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

Führen Sie ein interaktives Szenario an einem Prompt aus.

```
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))
}
```

Behandeln Sie den `MigrateUser`-Trigger mit einer Lambda-Funktion.

```
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)
}
```

Erstellen Sie eine Struktur, die allgemeine Aufgaben ausführt.

```
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))
}
```

Erstellen Sie eine Struktur, die Amazon Cognito-Aktionen umschließt.

```
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
}
```

Erstellen Sie eine Struktur, die DynamoDB-Aktionen umhüllt.

```
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
}
```

Erstellen Sie eine Struktur, die CloudWatch Logs-Aktionen umschließt.

```
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
}
```

Erstellen Sie eine Struktur, die Aktionen umschließt. 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
}
```

Bereinigen von Ressourcen.

```
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!")
  }
}
```

- Weitere API-Informationen finden Sie in den folgenden Themen der AWS SDK für Go -API-Referenz.
  - ConfirmForgotPassword

- DeleteUser
- ForgotPassword
- InitiateAuth
- SignUp
- UpdateUserPool

Eine vollständige Liste der AWS SDK-Entwicklerhandbücher und Codebeispiele finden Sie unterLambda mit einem AWS SDK verwenden. Dieses Thema enthält auch Informationen zu den ersten Schritten und Details zu früheren SDK-Versionen.

Erstellen einer API-Gateway-REST-API zur Verfolgung von COVID-19-Daten

Das folgende Codebeispiel zeigt, wie eine REST-API erstellt wird, die ein System zur Verfolgung der täglichen COVID-19-Fälle in den Vereinigten Staaten unter Verwendung fiktiver Daten simuliert.

Python

SDK für Python (Boto3)

Zeigt, wie AWS Chalice mit dem verwendet wird AWS SDK für Python (Boto3) , um eine serverlose REST-API zu erstellen, die Amazon API Gateway und Amazon AWS Lambda DynamoDB verwendet. Die REST-API simuliert ein System, das die täglichen COVID-19-Fälle in den Vereinigten Staaten unter Verwendung fiktiver Daten simuliert. Lernen Sie Folgendes:

- Verwenden Sie AWS Chalice, um Routen in Lambda-Funktionen zu definieren, die aufgerufen werden, um REST-Anfragen zu bearbeiten, die über API Gateway eingehen.
- Verwenden Sie Lambda-Funktionen zum Abrufen und Speichern von Daten in einer DynamoDB-Tabelle, um REST-Anforderungen zu bearbeiten.
- Definieren Sie die Tabellenstruktur und die Ressourcen für Sicherheitsrollen in einer AWS CloudFormation Vorlage.
- Verwenden Sie AWS Chalice und CloudFormation , um alle erforderlichen Ressourcen zu verpacken und bereitzustellen.
- Wird verwendet CloudFormation , um alle erstellten Ressourcen zu bereinigen.

Den vollständigen Quellcode und Anweisungen zur Einrichtung und Ausführung finden Sie im vollständigen Beispiel unter GitHub.

In diesem Beispiel verwendete Dienste

- API Gateway
- CloudFormation
- DynamoDB
- Lambda

Eine vollständige Liste der AWS SDK-Entwicklerhandbücher und Codebeispiele finden Sie unterLambda mit einem AWS SDK verwenden. Dieses Thema enthält auch Informationen zu den ersten Schritten und Details zu früheren SDK-Versionen.

Leihbibliothek-REST-API erstellen

Im folgenden Codebeispiel wird veranschaulicht, wie man eine Leihbibliothek erstellt, in der Kunden Bücher mithilfe einer REST-API ausleihen und zurückgeben können, die von einer Amazon-Aurora-Datenbank unterstützt wird.

Python

SDK für Python (Boto3)

Zeigt, wie die AWS SDK für Python (Boto3) mit der Amazon Relational Database Service (Amazon RDS) API und AWS Chalice verwendet wird, um eine REST-API zu erstellen, die von einer Amazon Aurora Aurora-Datenbank unterstützt wird. Der Webservice ist vollständig Serverless und stellt eine einfache Leihbibliothek dar, in der die Kunden Bücher ausleihen und zurückgeben können. Lernen Sie Folgendes:

- Erstellen und verwalten Sie einen Serverless-Aurora-Datenbank-Cluster.
- Wird AWS Secrets Manager zur Verwaltung von Datenbankanmeldedaten verwendet.
- Implementieren Sie einen Datenspeicher-Layer, der Amazon RDS verwendet, um Daten in die und aus der Datenbank zu verschieben.
- Verwenden Sie AWS Chalice, um eine serverlose REST-API für Amazon API Gateway bereitzustellen und. AWS Lambda
- Verwenden Sie das Anforderungspaket, um Anfragen an den Webservice zu senden.

Den vollständigen Quellcode und Anweisungen zur Einrichtung und Ausführung finden Sie im vollständigen Beispiel unter. GitHub

In diesem Beispiel verwendete Dienste

- API Gateway
- Aurora
- Lambda
- Secrets Manager

Eine vollständige Liste der AWS SDK-Entwicklerhandbücher und Codebeispiele finden Sie unterLambda mit einem AWS SDK verwenden. Dieses Thema enthält auch Informationen zu den ersten Schritten und Details zu früheren SDK-Versionen.

Erstellen einer Messenger-Anwendung mit Step Functions

Das folgende Codebeispiel zeigt, wie eine AWS Step Functions Messenger-Anwendung erstellt wird, die Nachrichtendatensätze aus einer Datenbanktabelle abruft.

Python

SDK für Python (Boto3)

Zeigt, wie AWS SDK für Python (Boto3) mit AWS Step Functions dem with eine Messenger-Anwendung erstellt wird, die Nachrichtendatensätze aus einer Amazon DynamoDB-Tabelle abruft und sie mit Amazon Simple Queue Service (Amazon SQS) sendet. Die Zustandsmaschine ist mit einer AWS Lambda Funktion integriert, mit der die Datenbank nach nicht gesendeten Nachrichten durchsucht werden kann.

- Erstellen Sie einen Zustandsautomaten, der Nachrichtendatensätze aus einer Amazon-DynamoDB-Tabelle abruft und aktualisiert.
- Aktualisieren Sie die Definition des Zustandsautomaten, um auch Nachrichten an Amazon Simple Queue Service (Amazon SQS) zu senden.
- Starten und stoppen Sie Ausführungen des Zustandsautomaten.
- Stellen Sie vom Zustandsautomaten aus über Serviceintegrationen eine Verbindung zu Lambda, DynamoDB und Amazon SQS her.

Den vollständigen Quellcode und Anweisungen zur Einrichtung und Ausführung finden Sie im vollständigen Beispiel unter GitHub.

In diesem Beispiel verwendete Dienste

- DynamoDB

- Lambda
- Amazon SQS
- Step Functions

Eine vollständige Liste der AWS SDK-Entwicklerhandbücher und Codebeispiele finden Sie unterLambda mit einem AWS SDK verwenden. Dieses Thema enthält auch Informationen zu den ersten Schritten und Details zu früheren SDK-Versionen.

Eine Anwendung für Foto-Asset-Management erstellen, mit der Benutzer Fotos mithilfe von Labels verwalten können

Die folgenden Codebeispiele zeigen, wie eine Serverless-Anwendung erstellt wird, mit der Benutzer Fotos mithilfe von Labels verwalten können.

.NET

SDK für .NET

Zeigt, wie eine Anwendung zur Verwaltung von Fotobeständen entwickelt wird, die mithilfe von Amazon Rekognition Labels in Bildern erkennt und sie für einen späteren Abruf speichert.

Den vollständigen Quellcode und Anweisungen zur Einrichtung und Ausführung finden Sie im vollständigen Beispiel unter GitHub.

Einen tiefen Einblick in den Ursprung dieses Beispiels finden Sie im Beitrag in der AWS - Community.

In diesem Beispiel verwendete Dienste

- API Gateway
- DynamoDB
- Lambda
- Amazon Rekognition
- Amazon S3
- Amazon SNS

C++

SDK für C++

Zeigt, wie eine Anwendung zur Verwaltung von Fotobeständen entwickelt wird, die mithilfe von Amazon Rekognition Labels in Bildern erkennt und sie für einen späteren Abruf speichert.

Den vollständigen Quellcode und Anweisungen zur Einrichtung und Ausführung finden Sie im vollständigen Beispiel unter GitHub.

Einen tiefen Einblick in den Ursprung dieses Beispiels finden Sie im Beitrag in der AWS - Community.

In diesem Beispiel verwendete Dienste

- API Gateway
- DynamoDB
- Lambda
- Amazon Rekognition
- Amazon S3
- Amazon SNS

Java

SDK für Java 2.x

Zeigt, wie eine Anwendung zur Verwaltung von Fotobeständen entwickelt wird, die mithilfe von Amazon Rekognition Labels in Bildern erkennt und sie für einen späteren Abruf speichert.

Den vollständigen Quellcode und Anweisungen zur Einrichtung und Ausführung finden Sie im vollständigen Beispiel unter GitHub.

Einen tiefen Einblick in den Ursprung dieses Beispiels finden Sie im Beitrag in der AWS - Community.

In diesem Beispiel verwendete Dienste

- API Gateway
- DynamoDB
- Lambda

- Amazon Rekognition
- Amazon S3
- Amazon SNS

JavaScript

SDK für JavaScript (v3)

Zeigt, wie eine Anwendung zur Verwaltung von Fotobeständen entwickelt wird, die mithilfe von Amazon Rekognition Labels in Bildern erkennt und sie für einen späteren Abruf speichert.

Den vollständigen Quellcode und Anweisungen zur Einrichtung und Ausführung finden Sie im vollständigen Beispiel unter GitHub.

Einen tiefen Einblick in den Ursprung dieses Beispiels finden Sie im Beitrag in der AWS - Community.

In diesem Beispiel verwendete Dienste

- API Gateway
- DynamoDB
- Lambda
- Amazon Rekognition
- Amazon S3
- Amazon SNS

Kotlin

SDK für Kotlin

Zeigt, wie eine Anwendung zur Verwaltung von Fotobeständen entwickelt wird, die mithilfe von Amazon Rekognition Labels in Bildern erkennt und sie für einen späteren Abruf speichert.

Den vollständigen Quellcode und Anweisungen zur Einrichtung und Ausführung finden Sie im vollständigen Beispiel unter GitHub.

Einen tiefen Einblick in den Ursprung dieses Beispiels finden Sie im Beitrag in der AWS - Community.

In diesem Beispiel verwendete Dienste

- API Gateway
- DynamoDB
- Lambda
- Amazon Rekognition
- Amazon S3
- Amazon SNS

PHP

SDK für PHP

Zeigt, wie eine Anwendung zur Verwaltung von Fotobeständen entwickelt wird, die mithilfe von Amazon Rekognition Labels in Bildern erkennt und sie für einen späteren Abruf speichert.

Den vollständigen Quellcode und Anweisungen zur Einrichtung und Ausführung finden Sie im vollständigen Beispiel unter GitHub.

Einen tiefen Einblick in den Ursprung dieses Beispiels finden Sie im Beitrag in der AWS - Community.

In diesem Beispiel verwendete Dienste

- API Gateway
- DynamoDB
- Lambda
- Amazon Rekognition
- Amazon S3
- Amazon SNS

Rust

SDK für Rust

Zeigt, wie eine Anwendung zur Verwaltung von Fotobeständen entwickelt wird, die mithilfe von Amazon Rekognition Labels in Bildern erkennt und sie für einen späteren Abruf speichert.

Den vollständigen Quellcode und Anweisungen zur Einrichtung und Ausführung finden Sie im vollständigen Beispiel unter GitHub.

Einen tiefen Einblick in den Ursprung dieses Beispiels finden Sie im Beitrag in der AWS - Community.

In diesem Beispiel verwendete Dienste

- API Gateway
- DynamoDB
- Lambda
- Amazon Rekognition
- Amazon S3
- Amazon SNS

Eine vollständige Liste der AWS SDK-Entwicklerhandbücher und Codebeispiele finden Sie unterLambda mit einem AWS SDK verwenden. Dieses Thema enthält auch Informationen zu den ersten Schritten und Details zu früheren SDK-Versionen.

Erstellen einer Websocket-Chat-Anwendung mit API Gateway

Das folgende Codebeispiel zeigt, wie eine Chat-Anwendung erstellt wird, die von einer auf Amazon API Gateway basierenden Websocket-API bereitgestellt wird.

Python

SDK für Python (Boto3)

Zeigt, wie Sie AWS SDK für Python (Boto3) mit Amazon API Gateway V2 eine Websocket-API erstellen, die in Amazon DynamoDB integriert AWS Lambda werden kann.

- Erstellen Sie eine WebSocket-API, die von API Gateway bereitgestellt wird.
- Definieren Sie einen Lambda-Handler, der Verbindungen in DynamoDB speichert und Nachrichten an andere Chat-Teilnehmer sendet.
- Stellen Sie eine Verbindung zur Websocket-Chat-Anwendung her und senden Sie Nachrichten mit dem Websockets-Paket.

Den vollständigen Quellcode und Anweisungen zur Einrichtung und Ausführung finden Sie im vollständigen Beispiel unter. GitHub

In diesem Beispiel verwendete Dienste

- API Gateway
- DynamoDB
- Lambda

Eine vollständige Liste der AWS SDK-Entwicklerhandbücher und Codebeispiele finden Sie unterLambda mit einem AWS SDK verwenden. Dieses Thema enthält auch Informationen zu den ersten Schritten und Details zu früheren SDK-Versionen.

Erstellen einer Anwendung, die Kundenfeedback analysiert und Audio generiert

Die folgenden Codebeispiele zeigen, wie Sie eine Anwendung erstellen, die Kundenkommentarkarten analysiert, sie aus ihrer Originalsprache übersetzt, ihre Stimmung ermittelt und aus dem übersetzten Text eine Audiodatei generiert.

.NET

SDK für .NET

Diese Beispielanwendung analysiert und speichert Kundenfeedback-Karten. Sie ist auf die Anforderungen eines fiktiven Hotels in New York City zugeschnitten. Das Hotel erhält Feedback von Gästen in Form von physischen Kommentarkarten in verschiedenen Sprachen. Dieses Feedback wird über einen Webclient in die App hochgeladen. Nachdem ein Bild einer Kommentarkarte hochgeladen wurde, werden folgende Schritte ausgeführt:

- Der Text wird mithilfe von Amazon Textract aus dem Bild extrahiert.
- Amazon Comprehend ermittelt die Stimmung und die Sprache des extrahierten Textes.
- Der extrahierte Text wird mithilfe von Amazon Translate ins Englische übersetzt.
- Amazon Polly generiert auf der Grundlage des extrahierten Texts eine Audiodatei.

Die vollständige App kann mithilfe des AWS CDK bereitgestellt werden. Den Quellcode und Anweisungen zur Bereitstellung finden Sie im Projekt unter GitHub.

In diesem Beispiel verwendete Dienste

- Amazon Comprehend
- Lambda

- Amazon Polly
- Amazon Textract
- Amazon Translate

Java

SDK für Java 2.x

Diese Beispielanwendung analysiert und speichert Kundenfeedback-Karten. Sie ist auf die Anforderungen eines fiktiven Hotels in New York City zugeschnitten. Das Hotel erhält Feedback von Gästen in Form von physischen Kommentarkarten in verschiedenen Sprachen. Dieses Feedback wird über einen Webclient in die App hochgeladen. Nachdem ein Bild einer Kommentarkarte hochgeladen wurde, werden folgende Schritte ausgeführt:

- Der Text wird mithilfe von Amazon Textract aus dem Bild extrahiert.
- Amazon Comprehend ermittelt die Stimmung und die Sprache des extrahierten Textes.
- Der extrahierte Text wird mithilfe von Amazon Translate ins Englische übersetzt.
- Amazon Polly generiert auf der Grundlage des extrahierten Texts eine Audiodatei.

Die vollständige App kann mithilfe des AWS CDK bereitgestellt werden. Den Quellcode und Anweisungen zur Bereitstellung finden Sie im Projekt unter GitHub.

In diesem Beispiel verwendete Dienste

- Amazon Comprehend
- Lambda
- Amazon Polly
- Amazon Textract
- Amazon Translate

JavaScript

SDK für JavaScript (v3)

Diese Beispielanwendung analysiert und speichert Kundenfeedback-Karten. Sie ist auf die Anforderungen eines fiktiven Hotels in New York City zugeschnitten. Das Hotel erhält Feedback von Gästen in Form von physischen Kommentarkarten in verschiedenen Sprachen.

Dieses Feedback wird über einen Webclient in die App hochgeladen. Nachdem ein Bild einer Kommentarkarte hochgeladen wurde, werden folgende Schritte ausgeführt:

- Der Text wird mithilfe von Amazon Textract aus dem Bild extrahiert.
- Amazon Comprehend ermittelt die Stimmung und die Sprache des extrahierten Textes.
- Der extrahierte Text wird mithilfe von Amazon Translate ins Englische übersetzt.
- Amazon Polly generiert auf der Grundlage des extrahierten Texts eine Audiodatei.

Die vollständige App kann mithilfe des AWS CDK bereitgestellt werden. Quellcode und Anweisungen zur Bereitstellung finden Sie im Projekt unter GitHub. Die folgenden Auszüge zeigen, wie der innerhalb von Lambda-Funktionen verwendet AWS SDK für JavaScript wird.

```
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 };
};
```

In diesem Beispiel verwendete Dienste

- Amazon Comprehend
- Lambda
- Amazon Polly
- Amazon Textract
- Amazon Translate

Ruby

SDK für Ruby

Diese Beispielanwendung analysiert und speichert Kundenfeedback-Karten. Sie ist auf die Anforderungen eines fiktiven Hotels in New York City zugeschnitten. Das Hotel erhält Feedback von Gästen in Form von physischen Kommentarkarten in verschiedenen Sprachen.

Dieses Feedback wird über einen Webclient in die App hochgeladen. Nachdem ein Bild einer Kommentarkarte hochgeladen wurde, werden folgende Schritte ausgeführt:

- Der Text wird mithilfe von Amazon Textract aus dem Bild extrahiert.
- Amazon Comprehend ermittelt die Stimmung und die Sprache des extrahierten Textes.
- Der extrahierte Text wird mithilfe von Amazon Translate ins Englische übersetzt.
- Amazon Polly generiert auf der Grundlage des extrahierten Texts eine Audiodatei.

Die vollständige App kann mithilfe des AWS CDK bereitgestellt werden. Quellcode und Anweisungen zur Bereitstellung finden Sie im Projekt unter. GitHub

In diesem Beispiel verwendete Dienste

- Amazon Comprehend
- Lambda
- Amazon Polly
- Amazon Textract
- Amazon Translate

Eine vollständige Liste der AWS SDK-Entwicklerhandbücher und Codebeispiele finden Sie unterLambda mit einem AWS SDK verwenden. Dieses Thema enthält auch Informationen zu den ersten Schritten und Details zu früheren SDK-Versionen.

Aufrufen einer Lambda-Funktion von einem Browser aus

Das folgende Codebeispiel zeigt, wie eine AWS Lambda Funktion von einem Browser aus aufgerufen wird.

JavaScript

SDK für JavaScript (v2)

Sie können eine browserbasierte Anwendung erstellen, die eine AWS Lambda Funktion verwendet, um eine Amazon DynamoDB-Tabelle mit Benutzerauswahlen zu aktualisieren.

Den vollständigen Quellcode und Anweisungen zur Einrichtung und Ausführung finden Sie im vollständigen Beispiel unter. GitHub

In diesem Beispiel verwendete Dienste

- DynamoDB

- Lambda

SDK für JavaScript (v3)

Sie können eine browserbasierte Anwendung erstellen, die eine AWS Lambda Funktion verwendet, um eine Amazon DynamoDB-Tabelle mit Benutzerauswahlen zu aktualisieren. Diese App verwendet v3. AWS SDK für JavaScript

Den vollständigen Quellcode und Anweisungen zur Einrichtung und Ausführung finden Sie im vollständigen Beispiel unter GitHub.

In diesem Beispiel verwendete Dienste

- DynamoDB
- Lambda

Eine vollständige Liste der AWS SDK-Entwicklerhandbücher und Codebeispiele finden Sie unterLambda mit einem AWS SDK verwenden. Dieses Thema enthält auch Informationen zu den ersten Schritten und Details zu früheren SDK-Versionen.

Transformieren von Daten für Ihre Anwendung mit S3 Object Lambda

Das folgende Codebeispiel zeigt, wie Sie Daten für Ihre Anwendung mit S3 Object Lambda transformieren können.

.NET

SDK für .NET

Zeigt, wie Sie benutzerdefinierten Code zu standardmäßigen S3-GET-Anforderungen hinzufügen, um das angeforderte Objekt, das von S3 abgerufen wurde, so zu ändern, dass das Objekt den Anforderungen des anfordernden Clients oder der anfordernden Anwendung entspricht.

Den vollständigen Quellcode und Anweisungen zur Einrichtung und Ausführung finden Sie im vollständigen Beispiel unter GitHub.

In diesem Beispiel verwendete Dienste

- Lambda

- Amazon S3

Eine vollständige Liste der AWS SDK-Entwicklerhandbücher und Codebeispiele finden Sie unterLambda mit einem AWS SDK verwenden. Dieses Thema enthält auch Informationen zu den ersten Schritten und Details zu früheren SDK-Versionen.

Verwenden von API Gateway zum Aufrufen einer Lambda-Funktion

Die folgenden Codebeispiele zeigen, wie eine von Amazon API Gateway aufgerufene AWS Lambda Funktion erstellt wird.

Java

SDK für Java 2.x

Zeigt, wie eine AWS Lambda Funktion mithilfe der Lambda Java Runtime API erstellt wird. In diesem Beispiel werden verschiedene AWS Dienste aufgerufen, um einen bestimmten Anwendungsfall auszuführen. Dieses Beispiel zeigt, wie man eine Lambda-Funktion erstellt, die von Amazon API Gateway aufgerufen wird und eine Amazon-DynamoDB-Tabelle nach Arbeitsjubiläen durchsucht und Amazon Simple Notification Service (Amazon SNS) verwendet, um eine Textnachricht an Ihre Mitarbeiter zu senden, die ihnen zu ihrem einjährigen Jubiläum gratuliert.

Den vollständigen Quellcode und Anweisungen zur Einrichtung und Ausführung finden Sie im vollständigen Beispiel unter GitHub.

In diesem Beispiel verwendete Dienste

- API Gateway
- DynamoDB
- Lambda
- Amazon SNS

JavaScript

SDK für JavaScript (v3)

Zeigt, wie eine AWS Lambda Funktion mithilfe der JavaScript Lambda-Laufzeit-API erstellt wird. In diesem Beispiel werden verschiedene AWS Dienste aufgerufen, um einen bestimmten

Anwendungsfall auszuführen. Dieses Beispiel zeigt, wie man eine Lambda-Funktion erstellt, die von Amazon API Gateway aufgerufen wird und eine Amazon-DynamoDB-Tabelle nach Arbeitsjubiläen durchsucht und Amazon Simple Notification Service (Amazon SNS) verwendet, um eine Textnachricht an Ihre Mitarbeiter zu senden, die ihnen zu ihrem einjährigen Jubiläum gratuliert.

Den vollständigen Quellcode und Anweisungen zur Einrichtung und Ausführung finden Sie im vollständigen Beispiel unter GitHub.

Dieses Beispiel ist auch verfügbar im AWS SDK für JavaScript Entwicklerhandbuch für v3.

In diesem Beispiel verwendete Dienste

- API Gateway
- DynamoDB
- Lambda
- Amazon SNS

Python

SDK für Python (Boto3)

Dieses Beispiel veranschaulicht, wie eine REST-API für Amazon API Gateway erstellt und verwendet wird, die auf eine AWS Lambda -Funktion verweist. Der Lambda-Handler veranschaulicht, wie basierend auf HTTP-Methoden weitergeleitet wird, wie Daten aus der Abfragezeichenfolge, dem Header und dem Text abgerufen werden und wie eine JSON-Antwort zurückgegeben wird.

- Stellen Sie eine Lambda-Funktion bereit.
- REST-API für API Gateway erstellen
- Erstellen Sie eine REST-Ressource, die auf die Lambda-Funktion verweist.
- Erteilen Sie API Gateway die Berechtigung, die Lambda-Funktion aufzurufen.
- Verwenden Sie das Anforderungspaket, um Anforderungen an die REST-API zu senden.
- Bereinigen Sie alle Ressourcen, die während der Demo erstellt wurden.

Dieses Beispiel lässt sich am besten auf ansehen GitHub. Den vollständigen Quellcode und Anweisungen zur Einrichtung und Ausführung finden Sie im vollständigen Beispiel unter GitHub.

In diesem Beispiel verwendete Dienste

- API Gateway
- DynamoDB
- Lambda
- Amazon SNS

Eine vollständige Liste der AWS SDK-Entwicklerhandbücher und Codebeispiele finden Sie unterLambda mit einem AWS SDK verwenden. Dieses Thema enthält auch Informationen zu den ersten Schritten und Details zu früheren SDK-Versionen.

Verwenden von Step Functions, um Lambda-Funktionen aufzurufen

Das folgende Codebeispiel zeigt, wie Sie eine AWS Step Functions Zustandsmaschine erstellen, die nacheinander AWS Lambda Funktionen aufruft.

Java

SDK für Java 2.x

Zeigt, wie Sie mithilfe von AWS Step Functions und einen AWS serverlosen Workflow erstellen. AWS SDK for Java 2.x Jeder Workflow-Schritt wird mithilfe einer AWS Lambda Funktion implementiert.

Den vollständigen Quellcode und Anweisungen zur Einrichtung und Ausführung finden Sie im vollständigen Beispiel unter GitHub.

In diesem Beispiel verwendete Dienste

- DynamoDB
- Lambda
- Amazon SES
- Step Functions

Eine vollständige Liste der AWS SDK-Entwicklerhandbücher und Codebeispiele finden Sie unterLambda mit einem AWS SDK verwenden. Dieses Thema enthält auch Informationen zu den ersten Schritten und Details zu früheren SDK-Versionen.

Verwendung geplanter Ereignisse zum Aufrufen einer Lambda-Funktion

Die folgenden Codebeispiele zeigen, wie eine AWS Lambda Funktion erstellt wird, die durch ein von Amazon EventBridge geplantes Ereignis aufgerufen wird.

Java

SDK für Java 2.x

Zeigt, wie ein von Amazon EventBridge geplantes Ereignis erstellt wird, das eine AWS Lambda Funktion aufruft. Konfigurieren Sie so EventBridge , dass ein Cron-Ausdruck verwendet wird, um zu planen, wann die Lambda-Funktion aufgerufen wird. In diesem Beispiel erstellen Sie eine Lambda-Funktion mithilfe der Lambda-Java-Laufzeit-API. In diesem Beispiel werden verschiedene AWS Dienste aufgerufen, um einen bestimmten Anwendungsfall auszuführen. Dieses Beispiel zeigt, wie man eine App erstellt, die eine mobile Textnachricht an Ihre Mitarbeiter sendet, um ihnen zum einjährigen Jubiläum zu gratulieren.

Den vollständigen Quellcode und Anweisungen zur Einrichtung und Ausführung finden Sie im vollständigen Beispiel unter GitHub.

In diesem Beispiel verwendete Dienste

- CloudWatch Logs
- DynamoDB
- EventBridge
- Lambda
- Amazon SNS

JavaScript

SDK für JavaScript (v3)

Zeigt, wie ein von Amazon EventBridge geplantes Ereignis erstellt wird, das eine AWS Lambda Funktion aufruft. Konfigurieren Sie so EventBridge , dass ein Cron-Ausdruck verwendet wird, um zu planen, wann die Lambda-Funktion aufgerufen wird. In diesem Beispiel erstellen Sie eine Lambda-Funktion mithilfe der JavaScript Lambda-Laufzeit-API. In diesem Beispiel werden verschiedene AWS Dienste aufgerufen, um einen bestimmten Anwendungsfall auszuführen. Dieses Beispiel zeigt, wie man eine App erstellt, die eine mobile Textnachricht an Ihre Mitarbeiter sendet, um ihnen zum einjährigen Jubiläum zu gratulieren.

Den vollständigen Quellcode und Anweisungen zur Einrichtung und Ausführung finden Sie im vollständigen Beispiel unter GitHub.

Dieses Beispiel ist auch verfügbar im AWS SDK für JavaScript Entwicklerhandbuch für v3.

In diesem Beispiel verwendete Dienste

- CloudWatch Logs
- DynamoDB
- EventBridge
- Lambda
- Amazon SNS

Python

SDK für Python (Boto3)

Dieses Beispiel zeigt, wie eine AWS Lambda Funktion als Ziel einer geplanten EventBridge Amazon-Veranstaltung registriert wird. Der Lambda-Handler schreibt eine freundliche Nachricht und die vollständigen Ereignisdaten für den späteren Abruf in Amazon CloudWatch Logs.

- Stellt eine Lambda-Funktion bereit.
- Erzeugt ein EventBridge geplantes Ereignis und macht die Lambda-Funktion zum Ziel.
- Erteilt die Erlaubnis, die EventBridge Lambda-Funktion aufrufen zu lassen.
- Druckt die neuesten Daten aus CloudWatch Logs, um das Ergebnis der geplanten Aufrufe anzuzeigen.
- Bereinigt alle Ressourcen, die während der Demo erstellt wurden.

Dieses Beispiel lässt sich am besten auf ansehen. GitHub Den vollständigen Quellcode und Anweisungen zur Einrichtung und Ausführung finden Sie im vollständigen Beispiel unter GitHub.

In diesem Beispiel verwendete Dienste

- CloudWatch Logs
- DynamoDB
- EventBridge
- Lambda

- Amazon SNS

Eine vollständige Liste der AWS SDK-Entwicklerhandbücher und Codebeispiele finden Sie unterLambda mit einem AWS SDK verwenden. Dieses Thema enthält auch Informationen zu den ersten Schritten und Details zu früheren SDK-Versionen.

Verwenden der Amazon Neptune-API zur Entwicklung einer Lambda-Funktion, die Graphdaten abfragt

Das folgende Codebeispiel zeigt, wie Sie mit der Neptune-API Grafikdaten abfragen.

Java

SDK für Java 2.x

Zeigt, wie die Java-API von Amazon Neptune verwendet wird, um eine Lambda-Funktion zu erstellen, die Grafikdaten innerhalb der VPC abfragt.

Den vollständigen Quellcode und Anweisungen zur Einrichtung und Ausführung finden Sie im vollständigen Beispiel unter GitHub.

In diesem Beispiel verwendete Dienste

- Lambda
- Neptune

Eine vollständige Liste der AWS SDK-Entwicklerhandbücher und Codebeispiele finden Sie unterLambda mit einem AWS SDK verwenden. Dieses Thema enthält auch Informationen zu den ersten Schritten und Details zu früheren SDK-Versionen.

Schreiben Sie benutzerdefinierte Aktivitätsdaten mit einer Lambda-Funktion nach der Amazon Cognito Cognito-Benutzerauthentifizierung mithilfe eines SDK AWS

Das folgende Codebeispiel veranschaulicht, wie benutzerdefinierte Aktivitätsdaten mit einer Lambda-Funktion nach der Amazon-Cognito-Benutzerauthentifizierung geschrieben werden.

- Verwenden Sie Administratorfunktionen, um einen Benutzer zu einem Benutzerpool hinzuzufügen.

- Konfigurieren Sie einen Benutzerpool, um eine Lambda-Funktion für den `PostAuthentication`-Trigger aufzurufen.
- Melden Sie den neuen Benutzer bei Amazon Cognito an.
- Die Lambda-Funktion schreibt benutzerdefinierte Informationen in CloudWatch Logs und in eine DynamoDB-Tabelle.
- Ermitteln Sie benutzerdefinierte Daten aus der DynamoDB-Tabelle, zeigen Sie diese an und bereinigen Sie dann die Ressourcen.

Go

SDK für Go V2

> **Note**
>
> Es gibt noch mehr dazu GitHub. Hier finden Sie das vollständige Beispiel und erfahren, wie Sie das AWS -Code-Beispiel- einrichten und ausführen.

Führen Sie ein interaktives Szenario an einem Prompt aus.

```
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))
}
```

Behandeln Sie den `PostAuthentication`-Trigger mit einer Lambda-Funktion.

```
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)
```

```
}
```

Erstellen Sie eine Struktur, die allgemeine Aufgaben ausführt.

```
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))
}
```

Erstellen Sie eine Struktur, die Amazon Cognito-Aktionen umschließt.

```
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
}
```

Erstellen Sie eine Struktur, die DynamoDB-Aktionen umhüllt.

```
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
}
```

Erstellen Sie eine Struktur, die CloudWatch Logs-Aktionen umschließt.

```
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
}
```

Erstellen Sie eine Struktur, die Aktionen umschließt. 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
}
```

Bereinigen von Ressourcen.

```
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!")
 }
}
```

- Weitere API-Informationen finden Sie in den folgenden Themen der AWS SDK für Go -API-Referenz.
  - AdminCreateUser
  - AdminSetUserPassword
  - DeleteUser
  - InitiateAuth
  - UpdateUserPool

Eine vollständige Liste der AWS SDK-Entwicklerhandbücher und Codebeispiele finden Sie unterLambda mit einem AWS SDK verwenden. Dieses Thema enthält auch Informationen zu den ersten Schritten und Details zu früheren SDK-Versionen.

Serverless-Beispiele für Lambda

Die folgenden Codebeispiele zeigen, wie Lambda mit AWS SDKs verwendet wird.

Beispiele

- Herstellen einer Verbindung mit einer Amazon-RDS-Datenbank in einer Lambda-Funktion
- Aufrufen einer Lambda-Funktion über einen Kinesis-Auslöser
- Aufrufen einer Lambda-Funktion über einen DynamoDB-Auslöser
- Aufrufen einer Lambda-Funktion über einen Amazon DocumentDB-Auslöser
- Aufrufen einer Lambda-Funktion über einen Amazon-MSK-Auslöser
- Aufrufen einer Lambda-Funktion über einen Amazon-S3-Auslöser
- Eine Lambda-Funktion über einen Amazon-SNS-Trigger aufrufen
- Aufrufen einer Lambda-Funktion über einen Amazon-SQS-Auslöser
- Melden von Batch-Elementfehlern für Lambda-Funktionen mit einem Kinesis-Auslöser
- Melden von Batch-Elementfehlern für Lambda-Funktionen mit einem DynamoDB-Auslöser

- Melden von Batch-Elementfehlern für Lambda-Funktionen mit einem Amazon-SQS-Auslöser

Herstellen einer Verbindung mit einer Amazon-RDS-Datenbank in einer Lambda-Funktion

Die folgenden Codebeispiele veranschaulichen, wie eine Lambda-Funktion implementiert wird, die eine Verbindung zu einer RDS-Datenbank herstellt. Die Funktion stellt eine einfache Datenbankanfrage und gibt das Ergebnis zurück.

.NET

SDK für .NET

> **Note**
>
> Es gibt noch mehr dazu GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Herstellen einer Verbindung mit einer Amazon-RDS-Datenbank in einer Lambda-Funktion mithilfe von .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

SDK für Go V2

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Herstellen einer Verbindung zu einer Amazon-RDS-Datenbank in einer Lambda-Funktion mit 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 für Java 2.x

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Herstellen einer Verbindung zu einer Amazon-RDS-Datenbank in einer Lambda-Funktion mit 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 für JavaScript (v3)

> **Note**
>
> Es gibt noch mehr dazu GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Herstellen einer Verbindung zu einer Amazon RDS-Datenbank in einer Lambda-Funktion mithilfe von 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)
  }
};
```

Herstellen einer Verbindung zu einer Amazon RDS-Datenbank in einer Lambda-Funktion mithilfe von 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

SDK für PHP

> **Note**
>
> Es gibt noch mehr dazu. GitHub Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Herstellen einer Verbindung zu einer Amazon-RDS-Datenbank in einer Lambda-Funktion mit 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

SDK für Python (Boto3)

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Herstellen einer Verbindung zu einer Amazon-RDS-Datenbank in einer Lambda-Funktion mit 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

SDK für Ruby

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Herstellen einer Verbindung zu einer Amazon-RDS-Datenbank in einer Lambda-Funktion mit 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 für Rust

> **Note**
>
> Es gibt noch mehr dazu GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Herstellen einer Verbindung zu einer Amazon-RDS-Datenbank in einer Lambda-Funktion mit 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}")
    }))
}
```

Eine vollständige Liste der AWS SDK-Entwicklerhandbücher und Codebeispiele finden Sie unterLambda mit einem AWS SDK verwenden. Dieses Thema enthält auch Informationen zu den ersten Schritten und Details zu früheren SDK-Versionen.

Aufrufen einer Lambda-Funktion über einen Kinesis-Auslöser

Die folgenden Codebeispiele veranschaulichen, wie eine Lambda-Funktion implementiert wird, die ein durch den Empfang von Datensätzen aus einem Kinesis-Stream ausgelöstes Ereignis empfängt. Die Funktion ruft die Kinesis-Nutzlast ab, dekodiert von Base64 und protokolliert den Datensatzinhalt.

.NET

SDK für .NET

> **Note**
>
> Es gibt noch mehr dazu GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines Kinesis-Ereignisses mit Lambda unter Verwendung von .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

SDK für Go V2

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines Kinesis-Ereignisses mit Lambda unter Verwendung von 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 für Java 2.x

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines Kinesis-Ereignisses mit Lambda unter Verwendung von 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 für JavaScript (v3)

> **Note**
>
> Es gibt noch mehr dazu GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Ein Kinesis-Ereignis mit Lambda verwenden. 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;
}
```

Ein Kinesis-Ereignis mit Lambda verwenden. 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

SDK für PHP

> **Note**
>
> Es gibt noch mehr dazu. GitHub Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines Kinesis-Ereignisses mit Lambda unter Verwendung von 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

SDK für Python (Boto3)

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines Kinesis-Ereignisses mit Lambda unter Verwendung von 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

SDK für Ruby

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines Kinesis-Ereignisses mit Lambda unter Verwendung von 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 für Rust

> **Note**
>
> Es gibt noch mehr dazu GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines Kinesis-Ereignisses mit Lambda unter Verwendung von 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
}
```

Eine vollständige Liste der AWS SDK-Entwicklerhandbücher und Codebeispiele finden Sie unterLambda mit einem AWS SDK verwenden. Dieses Thema enthält auch Informationen zu den ersten Schritten und Details zu früheren SDK-Versionen.

Aufrufen einer Lambda-Funktion über einen DynamoDB-Auslöser

Die folgenden Codebeispiele veranschaulichen, wie eine Lambda-Funktion implementiert wird, die ein durch den Empfang von Datensätzen aus einem DynamoDB-Stream ausgelöstes Ereignis empfängt. Die Funktion ruft die DynamoDB-Nutzdaten ab und protokolliert den Inhalt des Datensatzes.

.NET

SDK für .NET

> **Note**
>
> Es gibt noch mehr dazu GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines DynamoDB-Ereignisses mit Lambda unter Verwendung von .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

SDK für Go V2

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines DynamoDB-Ereignisses mit Lambda unter Verwendung von 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 für Java 2.x

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines DynamoDB-Ereignisses mit Lambda unter Verwendung von 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 für JavaScript (v3)

> **Note**
>
> Es gibt noch mehr dazu GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Konsumieren eines DynamoDB-Ereignisses mit Lambda unter Verwendung. 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)}`);
};
```

Konsumieren eines DynamoDB-Ereignisses mit Lambda unter Verwendung. 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

SDK für PHP

> **Note**
>
> Es gibt noch mehr dazu. GitHub Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines DynamoDB-Ereignisses mit Lambda unter Verwendung von 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

SDK für Python (Boto3)

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines DynamoDB-Ereignisses mit Lambda unter Verwendung von 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

SDK für Ruby

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines DynamoDB-Ereignisses mit Lambda unter Verwendung von 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 für Rust

> **Note**
>
> Es gibt noch mehr dazu GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines DynamoDB-Ereignisses mit Lambda unter Verwendung von 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(())

}
```

Eine vollständige Liste der AWS SDK-Entwicklerhandbücher und Codebeispiele finden Sie unterLambda mit einem AWS SDK verwenden. Dieses Thema enthält auch Informationen zu den ersten Schritten und Details zu früheren SDK-Versionen.

Aufrufen einer Lambda-Funktion über einen Amazon DocumentDB-Auslöser

Die folgenden Codebeispiele veranschaulichen, wie eine Lambda-Funktion implementiert wird, die ein durch den Empfang von Datensätzen aus einem DocumentDB-Änderungsstream ausgelöstes Ereignis empfängt. Die Funktion ruft die DocumentDB-Nutzdaten ab und protokolliert den Inhalt des Datensatzes.

.NET

SDK für .NET

> **Note**
>
> Es gibt noch mehr dazu GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines Amazon DocumentDB-Ereignisses mit Lambda unter Verwendung von .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

SDK für Go V2

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines Amazon DocumentDB-Ereignisses mit Lambda unter Verwendung von 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 für Java 2.x

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines Amazon DocumentDB-Ereignisses mit Lambda unter Verwendung von 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 für JavaScript (v3)

> **Note**
>
> Es gibt noch mehr dazu GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Ein Amazon DocumentDB DocumentDB-Ereignis mit Lambda verwenden. 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));
};
```

Ein Amazon DocumentDB DocumentDB-Ereignis mit Lambda verwenden 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

SDK für PHP

> **Note**
>
> Es gibt noch mehr dazu. GitHub Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines Amazon DocumentDB-Ereignisses mit Lambda unter Verwendung von 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

SDK für Python (Boto3)

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines Amazon DocumentDB-Ereignisses mit Lambda unter Verwendung von 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

SDK für Ruby

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines Amazon DocumentDB-Ereignisses mit Lambda unter Verwendung von 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 für Rust

> **Note**
>
> Es gibt noch mehr dazu GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines Amazon DocumentDB-Ereignisses mit Lambda unter Verwendung von 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(())

}
```

Eine vollständige Liste der AWS SDK-Entwicklerhandbücher und Codebeispiele finden Sie unterLambda mit einem AWS SDK verwenden. Dieses Thema enthält auch Informationen zu den ersten Schritten und Details zu früheren SDK-Versionen.

Aufrufen einer Lambda-Funktion über einen Amazon-MSK-Auslöser

Die folgenden Codebeispiele zeigen, wie eine Lambda-Funktion implementiert wird, die ein Ereignis empfängt, das durch den Empfang von Datensätzen von einem Amazon MSK-Cluster ausgelöst wird. Die Funktion ruft die MSK-Nutzdaten ab und protokolliert den Inhalt des Datensatzes.

.NET

SDK für .NET

> **Note**
>
> Es gibt noch mehr dazu GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines Amazon MSK-Ereignisses mit Lambda unter Verwendung von .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

SDK für Go V2

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines Amazon MSK-Ereignisses mit Lambda unter Verwendung von 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 für Java 2.x

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines Amazon MSK-Ereignisses mit Lambda unter Verwendung von 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 für JavaScript (v3)

> **Note**
>
> Es gibt noch mehr dazu GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Ein Amazon MSK-Ereignis mit Lambda verwenden. JavaScript

```
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)
      })
    }
}
```

Ein Amazon MSK-Ereignis mit Lambda verwenden. TypeScript

```
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

SDK für PHP

> **Note**
>
> Es gibt noch mehr dazu. GitHub Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines Amazon MSK-Ereignisses mit Lambda unter Verwendung von 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

SDK für Python (Boto3)

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines Amazon MSK-Ereignisses mit Lambda unter Verwendung von 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

SDK für Ruby

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines Amazon MSK-Ereignisses mit Lambda unter Verwendung von 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 für Rust

> **Note**
>
> Es gibt noch mehr dazu GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines Amazon MSK-Ereignisses mit Lambda unter Verwendung von 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
}
```

Eine vollständige Liste der AWS SDK-Entwicklerhandbücher und Codebeispiele finden Sie unterLambda mit einem AWS SDK verwenden. Dieses Thema enthält auch Informationen zu den ersten Schritten und Details zu früheren SDK-Versionen.

Aufrufen einer Lambda-Funktion über einen Amazon-S3-Auslöser

In den folgenden Codebeispiele wird die Implementierung einer Lambda-Funktion gezeigt, die ein Ereignis empfängt, das durch Hochladen eines Objekts in einen S3-Bucket ausgelöst wird. Die Funktion ruft den Namen des S3-Buckets sowie den Objektschlüssel aus dem Ereignisparameter ab und ruft die Amazon-S3-API auf, um den Inhaltstyp des Objekts abzurufen und zu protokollieren.

.NET

SDK für .NET

> **Note**
>
> Es gibt noch mehr dazu GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines S3-Ereignisses mit Lambda unter Verwendung von .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

SDK für Go V2

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines S3-Ereignisses mit Lambda unter Verwendung von 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 für Java 2.x

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines S3-Ereignisses mit Lambda unter Verwendung von Java

```
// 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 für JavaScript (v3)

> **Note**
>
> Es gibt noch mehr dazu GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Konsumieren eines S3-Ereignisses mit Lambda unter Verwendung 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);
    }
};
```

Konsumieren eines S3-Ereignisses mit Lambda unter Verwendung 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

SDK für PHP

> **Note**
>
> Es gibt noch mehr dazu. GitHub Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines S3-Ereignisses mit Lambda unter Verwendung von 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

SDK für Python (Boto3)

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines S3-Ereignisses mit Lambda unter Verwendung von 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

SDK für Ruby

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines S3-Ereignisses mit Lambda unter Verwendung von 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 für Rust

> **Note**
>
> Es gibt noch mehr dazu GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines S3-Ereignisses mit Lambda unter Verwendung von 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(())
```

```
}
```

Eine vollständige Liste der AWS SDK-Entwicklerhandbücher und Codebeispiele finden Sie unterLambda mit einem AWS SDK verwenden. Dieses Thema enthält auch Informationen zu den ersten Schritten und Details zu früheren SDK-Versionen.

Eine Lambda-Funktion über einen Amazon-SNS-Trigger aufrufen

In den folgenden Codebeispiele wird die Implementierung einer Lambda-Funktion veranschaulicht, die ein Ereignis empfängt, das durch das Empfangen von Nachrichten aus einem SNS-Thema ausgelöst wird. Die Funktion ruft die Nachrichten aus dem Ereignisparameter ab und protokolliert den Inhalt jeder Nachricht.

.NET

SDK für .NET

> **Note**
>
> Es gibt noch mehr dazu GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines SNS-Ereignisses mit Lambda unter Verwendung von .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

SDK für Go V2

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines SNS-Ereignisses mit Lambda unter Verwendung von 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 für Java 2.x

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines SNS-Ereignisses mit Lambda unter Verwendung von 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 für JavaScript (v3)

> **Note**
>
> Es gibt noch mehr dazu GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Konsumieren eines SNS-Ereignisses mit Lambda unter Verwendung. 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) {
    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;
  }
}
```

Konsumieren eines SNS-Ereignisses mit Lambda unter Verwendung. TypeScript

```
// 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

SDK für PHP

> **Note**
>
> Es gibt noch mehr dazu. GitHub Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines SNS-Ereignisses mit Lambda unter Verwendung von 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

SDK für Python (Boto3)

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines SNS-Ereignisses mit Lambda unter Verwendung von 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

SDK für Ruby

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines SNS-Ereignisses mit Lambda unter Verwendung von 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 für Rust

> **Note**
>
> Es gibt noch mehr dazu GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines SNS-Ereignisses mit Lambda unter Verwendung von 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
}
```

Eine vollständige Liste der AWS SDK-Entwicklerhandbücher und Codebeispiele finden Sie unterLambda mit einem AWS SDK verwenden. Dieses Thema enthält auch Informationen zu den ersten Schritten und Details zu früheren SDK-Versionen.

Aufrufen einer Lambda-Funktion über einen Amazon-SQS-Auslöser

Die folgenden Codebeispiele zeigen, wie eine Lambda-Funktion implementiert wird, die ein Ereignis empfängt, das durch den Empfang von Nachrichten aus einer SQS-Warteschlange ausgelöst wird. Die Funktion ruft die Nachrichten aus dem Ereignisparameter ab und protokolliert den Inhalt jeder Nachricht.

.NET

SDK für .NET

> **Note**
>
> Es gibt noch mehr dazu GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines SQS-Ereignisses mit Lambda unter Verwendung von .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

SDK für Go V2

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines SQS-Ereignisses mit Lambda unter Verwendung von 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 für Java 2.x

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines SQS-Ereignisses mit Lambda unter Verwendung von 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 für JavaScript (v3)

> **Note**
>
> Es gibt noch mehr dazu GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Konsumieren eines SQS-Ereignisses mit Lambda unter Verwendung. JavaScript

```
// 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;
  }
}
```

Konsumieren eines SQS-Ereignisses mit Lambda unter Verwendung. TypeScript

```
// 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

SDK für PHP

> **Note**
>
> Es gibt noch mehr dazu. GitHub Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines SQS-Ereignisses mit Lambda unter Verwendung von 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

SDK für Python (Boto3)

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines SQS-Ereignisses mit Lambda unter Verwendung von 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

SDK für Ruby

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines SQS-Ereignisses mit Lambda unter Verwendung von 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 für Rust

> **Note**
>
> Es gibt noch mehr dazu GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Nutzen eines SQS-Ereignisses mit Lambda unter Verwendung von 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
}
```

Eine vollständige Liste der AWS SDK-Entwicklerhandbücher und Codebeispiele finden Sie unterLambda mit einem AWS SDK verwenden. Dieses Thema enthält auch Informationen zu den ersten Schritten und Details zu früheren SDK-Versionen.

Melden von Batch-Elementfehlern für Lambda-Funktionen mit einem Kinesis-Auslöser

Die folgenden Codebeispiele zeigen, wie eine teilweise Batch-Antwort für Lambda-Funktionen implementiert wird, die Ereignisse von einem Kinesis-Stream empfangen. Die Funktion meldet die

Batch-Elementfehler in der Antwort und signalisiert Lambda, diese Nachrichten später erneut zu versuchen.

.NET

SDK für .NET

> **Note**
>
> Es gibt noch mehr dazu GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Melden von Fehlern bei Kinesis-Batchelementen mit Lambda unter Verwendung von .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

SDK für Go V2

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Melden von Fehlern bei Kinesis-Batchelementen mit Lambda unter Verwendung von 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 für Java 2.x

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Melden von Fehlern bei Kinesis-Batchelementen mit Lambda unter Verwendung von 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 für JavaScript (v3)

> **Note**
>
> Es gibt noch mehr dazu GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Melden von Fehlern bei Kinesis-Batchelementen mit Lambda unter Verwendung von 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;
}
```

Melden von Fehlern Kinesis Kinesis-Batch-Elementen mit Lambda unter Verwendung von. 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

SDK für PHP

> **Note**
>
> Es gibt noch mehr dazu. GitHub Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Melden von Fehlern bei Kinesis-Batchelementen mit Lambda unter Verwendung von 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

SDK für Python (Boto3)

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Melden von Fehlern bei Kinesis-Batchelementen mit Lambda unter Verwendung von 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

SDK für Ruby

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Melden von Fehlern bei Kinesis-Batchelementen mit Lambda unter Verwendung von 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 für Rust

> **Note**
>
> Es gibt noch mehr dazu GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Melden von Fehlern bei Kinesis-Batchelementen mit Lambda unter Verwendung von 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
}
```

Eine vollständige Liste der AWS SDK-Entwicklerhandbücher und Codebeispiele finden Sie unterLambda mit einem AWS SDK verwenden. Dieses Thema enthält auch Informationen zu den ersten Schritten und Details zu früheren SDK-Versionen.

Melden von Batch-Elementfehlern für Lambda-Funktionen mit einem DynamoDB-Auslöser

Die folgenden Codebeispiele zeigen, wie eine teilweise Batch-Antwort für Lambda-Funktionen implementiert wird, die Ereignisse von einem DynamoDB-Stream empfangen. Die Funktion meldet die Batch-Elementfehler in der Antwort und signalisiert Lambda, diese Nachrichten später erneut zu versuchen.

.NET

SDK für .NET

> **Note**
>
> Es gibt noch mehr dazu GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Melden von DynamoDB-Batchelementfehlern mit Lambda unter Verwendung von .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

SDK für Go V2

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Melden von DynamoDB-Batchelementfehlern mit Lambda unter Verwendung von 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 für Java 2.x

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Melden von DynamoDB-Batchelementfehlern mit Lambda unter Verwendung von 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 für JavaScript (v3)

> **Note**
>
> Es gibt noch mehr dazu GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Melden von DynamoDB-Batchelementfehlern mit Lambda unter Verwendung von. 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: [] };
};
```

Melden von DynamoDB-Batchelementfehlern mit Lambda unter Verwendung von. 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

SDK für PHP

> **Note**
>
> Es gibt noch mehr dazu. GitHub Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Melden von DynamoDB-Batchelementfehlern mit Lambda unter Verwendung von 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

SDK für Python (Boto3)

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Melden von DynamoDB-Batchelementfehlern mit Lambda unter Verwendung von 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

SDK für Ruby

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Melden von DynamoDB-Batchelementfehlern mit Lambda unter Verwendung von 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 für Rust

> **Note**
>
> Es gibt noch mehr dazu GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Melden von DynamoDB-Batchelementfehlern mit Lambda unter Verwendung von 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
}
```

Eine vollständige Liste der AWS SDK-Entwicklerhandbücher und Codebeispiele finden Sie unterLambda mit einem AWS SDK verwenden. Dieses Thema enthält auch Informationen zu den ersten Schritten und Details zu früheren SDK-Versionen.

Melden von Batch-Elementfehlern für Lambda-Funktionen mit einem Amazon-SQS-Auslöser

Die folgenden Codebeispiele zeigen, wie eine teilweise Batch-Antwort für Lambda-Funktionen implementiert wird, die Ereignisse aus einer SQS-Warteschlange empfangen. Die Funktion meldet die Batch-Elementfehler in der Antwort und signalisiert Lambda, diese Nachrichten später erneut zu versuchen.

.NET

SDK für .NET

> **Note**
>
> Es gibt noch mehr dazu GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Melden von SQS-Batchelementfehlern mit Lambda unter Verwendung von .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

SDK für Go V2

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Melden von SQS-Batchelementfehlern mit Lambda unter Verwendung von 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 für Java 2.x

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Melden von Fehlern bei SQS-Batchelementen mit Lambda unter Verwendung von 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 für JavaScript (v3)

> **Note**
>
> Es gibt noch mehr dazu GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Melden von SQS-Batch-Elementfehlern mit Lambda unter Verwendung von. JavaScript

```
// 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}`);
}
```

Melden von SQS-Batch-Elementfehlern mit Lambda unter Verwendung von. TypeScript

```
// 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

SDK für PHP

> **Note**
>
> Es gibt noch mehr dazu. GitHub Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Melden von SQS-Batchelementfehlern mit Lambda unter Verwendung von 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

SDK für Python (Boto3)

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Melden von Fehlern bei SQS-Batchelementen mit Lambda unter Verwendung von 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

SDK für Ruby

> **Note**
>
> Es gibt noch mehr GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Melden von Fehlern bei SQS-Batchelementen mit Lambda unter Verwendung von 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 für Rust

> **Note**
>
> Es gibt noch mehr dazu GitHub. Das vollständige Beispiel sowie eine Anleitung zum Einrichten und Ausführen finden Sie im Repository mit Serverless-Beispielen.

Melden von Fehlern bei SQS-Batchelementen mit Lambda unter Verwendung von 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
}
```

Eine vollständige Liste der AWS SDK-Entwicklerhandbücher und Codebeispiele finden Sie unterLambda mit einem AWS SDK verwenden. Dieses Thema enthält auch Informationen zu den ersten Schritten und Details zu früheren SDK-Versionen.

AWS Community-Beiträge für Lambda

AWS Community-Beiträge sind Beispiele, die von mehreren Teams erstellt wurden und von ihnen verwaltet werden AWS. Um Feedback zu geben, verwenden Sie den Mechanismus, der in den verlinkten Repositorys bereitgestellt wird.

Beispiele

- Erstellen und Testen einer Serverless-Anwendung

Erstellen und Testen einer Serverless-Anwendung

Die folgenden Codebeispiele zeigen, wie eine Serverless-Anwendung mithilfe von API Gateway mit Lambda und DynamoDB erstellt und getestet wird.

.NET

SDK für .NET

Es wird gezeigt, wie eine Serverless-Anwendung, bestehend aus einem API Gateway mit Lambda und DynamoDB, mithilfe des .NET SDK erstellt und getestet wird.

Den vollständigen Quellcode und Anweisungen zur Einrichtung und Ausführung finden Sie im vollständigen Beispiel unter GitHub.

In diesem Beispiel verwendete Dienste

- API Gateway
- DynamoDB

- Lambda

Go

SDK für Go V2

Es wird gezeigt, wie eine Serverless-Anwendung, bestehend aus einem API Gateway mit Lambda und DynamoDB, mithilfe des Go SDK erstellt und getestet wird.

Den vollständigen Quellcode und Anweisungen zur Einrichtung und Ausführung finden Sie im vollständigen Beispiel unter GitHub.

In diesem Beispiel verwendete Dienste

- API Gateway
- DynamoDB
- Lambda

Java

SDK für Java 2.x

Es wird gezeigt, wie eine Serverless-Anwendung, bestehend aus einem API Gateway mit Lambda und DynamoDB, mithilfe des Java SDK erstellt und getestet wird.

Den vollständigen Quellcode und Anweisungen zur Einrichtung und Ausführung finden Sie im vollständigen Beispiel unter GitHub.

In diesem Beispiel verwendete Dienste

- API Gateway
- DynamoDB
- Lambda

Rust

SDK für Rust

Es wird gezeigt, wie eine Serverless-Anwendung, bestehend aus einem API Gateway mit Lambda und DynamoDB, mithilfe des Rust SDK erstellt und getestet wird.

Den vollständigen Quellcode und Anweisungen zur Einrichtung und Ausführung finden Sie im vollständigen Beispiel unter GitHub.

In diesem Beispiel verwendete Dienste

- API Gateway
- DynamoDB
- Lambda

Eine vollständige Liste der AWS SDK-Entwicklerhandbücher und Codebeispiele finden Sie unterLambda mit einem AWS SDK verwenden. Dieses Thema enthält auch Informationen zu den ersten Schritten und Details zu früheren SDK-Versionen.

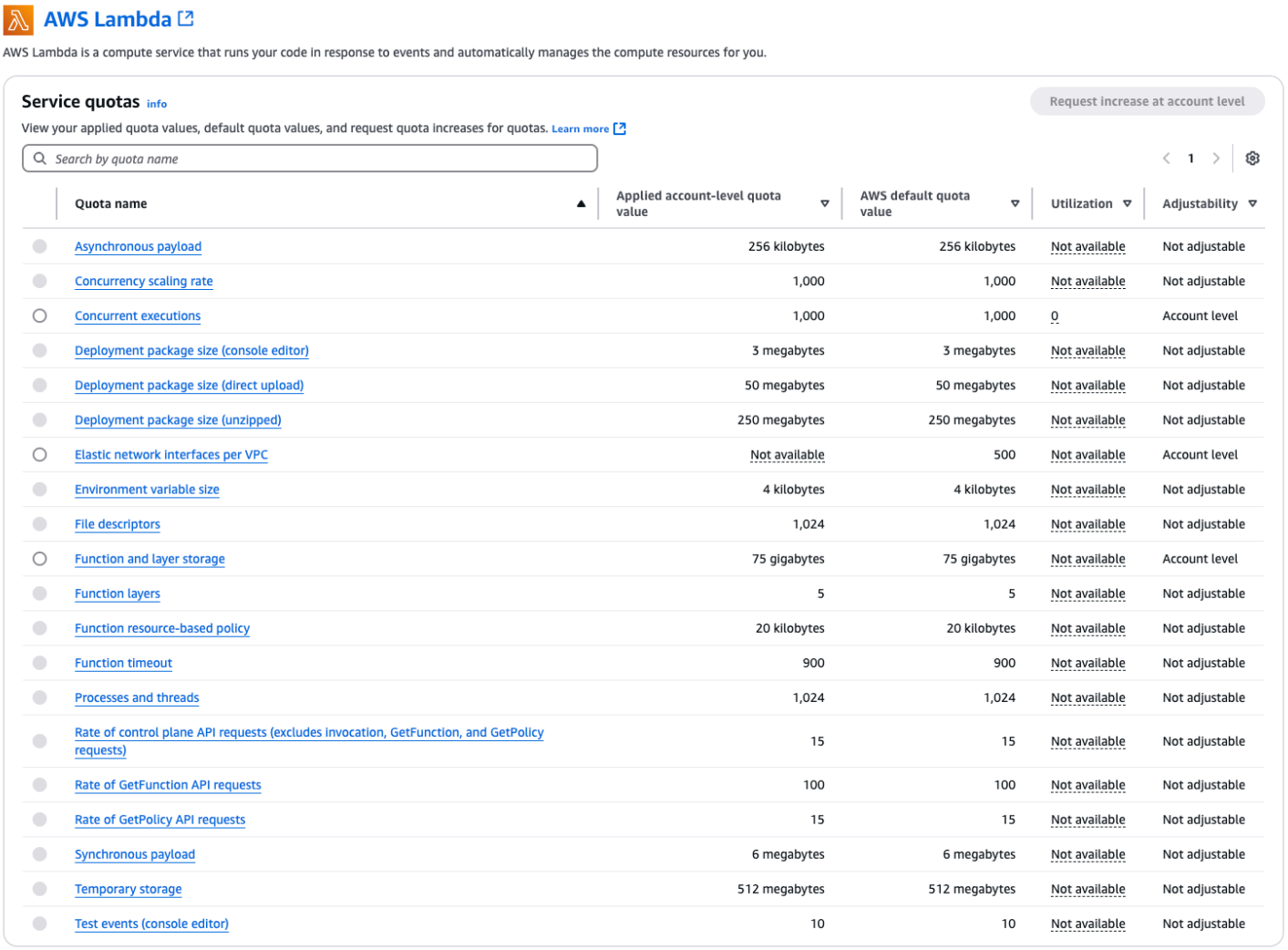
Lambda-Kontingente

> **Important**
>
> Neue AWS-Konten haben die Parallelität und die Speicherkontingente reduziert. AWS erhöht diese Kontingente automatisch auf der Grundlage Ihrer Nutzung.

AWS Lambda ist so konzipiert, dass es schnell skaliert werden kann, um den Bedarf zu decken, sodass Ihre Funktionen skaliert werden können, um den Datenverkehr in Ihrer Anwendung zu bedienen. Lambda ist für kurzlebige Rechenaufgaben konzipiert, die den Status zwischen Aufrufen nicht beibehalten oder nicht darauf angewiesen sind. Code kann in einem einzigen Aufruf bis zu 15 Minuten lang ausgeführt werden und eine einzelne Funktion kann bis zu 10 240 MB Speicher beanspruchen.

Es ist wichtig, die Schutzmaßnahmen zu verstehen, die zum Schutz Ihres Kontos und der Workloads anderer Kunden eingerichtet wurden. Dienstkontingente sind für alle AWS Dienste verfügbar und bestehen aus festen Grenzwerten, die Sie nicht ändern können, und weichen Grenzwerten, für die Sie Erhöhungen beantragen können. Standardmäßig wird allen neuen Konten ein Kontingentprofil zugewiesen, das die Erkundung von AWS Diensten ermöglicht.

Um die Kontingente zu sehen, die für Ihr Konto gelten, navigieren Sie zum Service-Quotas-Dashboard. Hier können Sie Service Quotas anzeigen, eine Erhöhung des Service-Kontingents anfordern und die aktuelle Auslastung anzeigen. Von hier aus können Sie zu einem bestimmten AWS Dienst wie Lambda aufschlüsseln:



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

In folgenden Abschnitten sind Standardkontingente und Grenzwerte in Lambda nach Kategorien aufgeführt.

Themen

- Datenverarbeitung und Speicherung
- Funktionskonfiguration, -bereitstellung und -ausführung
- Lambda-API-Anforderungen
- Sonstige Services

Datenverarbeitung und Speicherung

Lambda legt Kontingente für die Menge an Datenverarbeitung und Speicherressourcen, die Sie verwenden können, um Funktionen auszuführen und zu speichern. Kontingente für gleichzeitige Ausführungen und Speicherung gelten pro AWS-Region. Die Elastic-Network-Schnittstelle (ENI)-Kontingente gelten für jede Virtual Private Cloud (VPC), unabhängig von der Region. Die folgenden Kontingente können gegenüber ihren Standardwerten erhöht werden. Weitere Informationen finden Sie unter Beantragen einer Kontingenterhöhung im Service-Quotas-Benutzerhandbuch.

Ressource	Standardkontingent	Kann erhöht werden bis zu
Gleichzeitige Ausführungen	1.000	Zehntausende
Speicher für hochgeladene Funktionen (.zip-Datei-Archive) und Ebenen. Jede Funktionsversion und Ebenenversion verbraucht Speicher. Bewährte Methoden für die Verwaltung Ihres Codespeichers finden Sie bei Serverless Land unter Monitoring Lambda code storage.	75 GB	Terabytes
Speicher für als Container-Images definierten Funktionen Diese Bilder werden in Amazon ECR gespeichert.	Siehe Amazon-ECR-Service kontingente.	
Elastic Network-Schnittstellen in der Virtual Private Cloud (VPC) **Note** Dieses Kontingent wird mit anderen Services wie Amazon Elastic File System (Amazon EFS) geteilt. Siehe Amazon-VPC-Kontingente.	500	Tausende

Weitere Details zur Gleichzeitigkeit und zur datenverkehrbasierten Skalierung der Funktionsgleichzeitigkeit von Lambda finden Sie unter Verstehen der Skalierung von Lambda-Funktionen.

Funktionskonfiguration, -bereitstellung und -ausführung

Die folgenden Kontingente gelten für die Konfiguration, Bereitstellung und Ausführung von Funktionen. Sofern nicht anders angegeben, können sie nicht geändert werden.

> **Note**
>
> Die Lambda-Dokumentation, die Protokollmeldungen und die Konsole verwenden die Abkürzung MB (anstelle von MiB), um auf 1 024 KB zu verweisen.

Ressource	Quota
Funktion Speicherzuweisung	128 MB bis 10.240 MB (in Schritten von 1 MB). Hinweis: Lambda weist die CPU-Leistung proportional zur Menge des konfigurierten Arbeitsspeichers zu. Sie können den Arbeitsspeicher und die CPU-Leistung, die Ihrer Funktion zugewiesen sind, mit der Einstellung Arbeitsspeicher (MB) erhöhen oder verringern. Bei 1 769 MB hat eine Funktion das Äquivalent von einer vCPU.
Funktion Zeitüberschreitung	900 Sekunden (15 Minuten)
Funktion Umgebungsvariablen	4 KB, für alle Umgebungsvariablen, die mit der Funktion verknüpft sind, im Aggregat
Funktion ressourcenbasierte Richtlinie	20 KB

Ressource	Quota
Funktionsebenen	5 Ebenen
Funktion – Limit für Gleichzeitigkeitsskalierung	Für jede Funktion 1 000 Ausführungsumgebungen alle 10 Sekunden
Aufrufnutzlast (Anfrage und Antwort)	Jeweils 6 MB für Anfrage und Antwort (synchron) 200 MB für jede gestreamte Antwort (synchron) 1 MB (asynchron) 1 MB für die kombinierte Gesamtgröße von Anforderungszeile und Kopfdaten
Bandbreite für gestreamte Antworten	Unbegrenzt für die ersten 6 MB der Antwort Ihrer Funktion Für Antworten, die größer als 6 MB sind, 2 MBps für den Rest der Antwort
Größe des Bereitstellungspakets (ZIP-Dateiarchiv)	50 MB (gezippt, wenn über die Lambda-API hochgeladen oder SDKs). Laden Sie größere Dateien mit Amazon S3 hoch. 50 MB (beim Hochladen über die Lambda-Konsole) 250 MB Die maximale Größe des Inhalts eines Bereitstellungspakets, einschließlich Ebenen und benutzerdefinierter Laufzeiten. (ungezippt)

Ressource	Quota
Größe der Container-Image-Einstellungen	16 KB
Codepaketgröße des Container-Images	10 GB (maximale unkomprimierte Image-Größe, einschließlich aller Ebenen)
Testereignisse (Konsoleneditor)	10
`/tmp`-Verzeichnisspeicher	Zwischen 512 MB und 10 240 MB, in 1-MB-Schritten.
Dateibeschreibungen	1,024 **Note** Lambda Managed Instances verwenden die standardmäßigen Dateideskriptorgrenzwerte von Bottlerocket. Weitere Informationen finden Sie unter Grundlegendes zur Ausführungsumgebung von Lambda Managed Instances.

Ressource	Quota
Ausführungsprozesse/-Threads	1,024 **Note** Lambda Managed Instances verwenden die standardmäßigen Prozess- und Threadlimits von Bottlerocket. Weitere Informationen finden Sie unter Grundlegendes zur Ausführungsumgebung von Lambda Managed Instances.

Lambda-API-Anforderungen

Die folgenden Kontingente sind Lambda-API-Anfragen zugeordnet.

Ressource	Kontingent
Aufrufanfragen pro Funktion pro Region (synchron)	Jede Instance Ihrer Ausführungsumgebung kann bis zu 10 Anfragen pro Sekunde bearbeiten. Mit anderen Worten, das Gesamtauflimit beträgt das 10-fache Ihres Gleichzeitigkeitslimits. Siehe Verstehen der Skalierung von Lambda-Funktionen.
Aufrufanfragen pro Funktion pro Region (asynchron)	Jede Instance Ihrer Ausführungsumgebung kann eine unbegrenzte Anzahl an Anfragen bearbeiten. Mit anderen Worten, das Gesamtlimit für Aufrufe basiert nur auf der für Ihre Funktion verfügbaren

Ressource	Kontingent
	Gleichzeitigkeit. Siehe Verstehen der Skalierung von Lambda-Funktionen.
Aufrufanforderungen pro Funktionsversion oder Alias (Anfragen pro Sekunde)	10 x zugewiesene Provisioned Concurrency **Note** Dieses Kontingent gilt nur für Funktionen, die Provisioned Concurrency verwenden.
GetFunction-API-Anforderungen	100 Anforderungen pro Sekunde. Kann nicht erhöht werden.
GetPolicy-API-Anforderungen	15 Anforderungen pro Sekunde. Kann nicht erhöht werden.
Restliche API-Anfragen der Kontrollebene (ohne Aufrufe und Anfragen) GetFunction GetPolicy	Insgesamt 15 Anfragen pro Sekunde APIs (nicht 15 Anfragen pro Sekunde pro API). Kann nicht erhöht werden.

Sonstige Services

Kontingente für andere Dienste wie AWS Identity and Access Management (IAM), Amazon CloudFront (Lambda @Edge) und Amazon Virtual Private Cloud (Amazon VPC) können sich auf Ihre Lambda-Funktionen auswirken. Weitere Informationen finden Sie unter AWS-Service -Kontingent im Allgemeine Amazon Web Services-Referenz und Lambda mit Ereignissen aus anderen Diensten aufrufen AWS.

Viele Anwendungen, an denen Lambda beteiligt ist, verwenden mehrere AWS Dienste. Da verschiedene Services unterschiedliche Kontingente für verschiedene Funktionen haben, kann es schwierig sein, diese Kontingente für Ihre gesamte Anwendung zu verwalten. API Gateway hat z. B. eine standardmäßige Drosselungsgrenze von 10.000 Anforderungen pro Sekunde, während Lambda eine standardmäßige Gleichzeitigkeitsbeschränkung von 1.000 Anforderungen pro Sekunde

hat. Aufgrund dieser Diskrepanz ist es möglich, dass mehr eingehende Anfragen von API Gateway eingehen, als Lambda verarbeiten kann. Sie können dies beheben, indem Sie eine Erhöhung der Lambda-Gleichzeitigkeitsbeschränkung auf das erwartete Datenverkehrsaufkommen beantragen.

Durch Auslastungstests können Sie die Leistung Ihrer Anwendung überwachen, end-to-end bevor Sie sie in der Produktion einsetzen. Während eines Lasttests können Sie alle Kontingente ermitteln, die einen begrenzenden Faktor für das erwartete Verkehrsaufkommen darstellen und entsprechende Maßnahmen ergreifen.

Dokumentverlauf

In der folgenden Tabelle werden die wichtigen Änderungen am AWS Lambda Entwicklerhandbuch seit Mai 2018 beschrieben. Um Benachrichtigungen über Aktualisierungen dieser Dokumentation zu erhalten, können Sie den RSS-Feed abonnieren.

Änderung	Beschreibung	Datum
AWS verwaltete Richtlinienaktualisierungen	Lambda hat eine neue verwaltete Richtlinie veröffentlicht `AWSLambdaBasicDurableExecutionRolePolicy`, die Schreibberechtigungen für CloudWatch Protokolle und read/write Berechtigungen für die dauerhafte Ausführung bereitstellt, die von dauerhaften Lambda-Funktionen APIs verwendet werden.	1. Dezember 2025
Servicebezogene Rolle für Lambda	Lambda unterstützt jetzt eine serviceverknüpfte Rolle (`AWSServiceRoleForLambda`), die es Lambda ermöglicht, Instanzen zu beenden, die als Teil von Lambda-Kapazitätsanbietern verwaltet werden.	30. November 2025
Von Lambda verwaltete Instanzen	Lambda unterstützt jetzt Managed Instances, die dedizierte Rechenkapazität für Ihre Funktionen mit verbesserter Leistung, Sicherheit und Kontrolle bieten. Einzelhei	30. November 2025

	ten finden Sie unter Lambda Managed Instances.	
AWS verwaltete Richtlinienaktualisierungen	Lambda hat eine neue AWS verwaltete Richtlinie hinzugefügt (`AWSLambdaBasicDurableExecutionRolePolicy`).	30. November 2025
AWS verwaltete Richtlinienaktualisierungen	Lambda hat die bestehende AWS verwaltete Richtlinie aktualisiert`AWSLambda_FullAccess`) `kms:DescribeKey` und die `iam:CreateServiceLinkedRole` Aktionen zulassen.	30. November 2025
AWS verwaltetes Richtlinienupdate	Lambda hat eine neue AWS verwaltete Richtlinie (`AWSLambdaManagedEC2ResourceOperator`) hinzugefügt, um die automatisierte EC2 Amazon-Instanzverwaltung für Lambda-Kapazitätsanbieter zu ermöglichen. Einzelheiten finden Sie unter Lambda-Updates für AWS verwaltete Richtlinien.	30. November 2025

AWS verwaltetes Richtlinienupdate	Lambda hat eine neue AWS verwaltete Richtlinie hinzugefügt (`AWSLambdaServiceRolePolicy`). Einzelheiten finden Sie unter Lambda-Updates für AWS verwaltete Richtlinien.	30. November 2025
AWS verwaltete Richtlinienaktualisierungen	Lambda hat zwei bestehende AWS verwaltete Richtlinien aktualisiert (`AWSLambda_ReadOnlyAccess` und `AWSLambda_FullAccess`).	20. Februar 2025
Node.js-22.x-Laufzeit	Lambda unterstützt jetzt Node.js 22 als Image für die verwaltete Laufzeit und Container-Basis-Image (`nodejs22.x`).	22. November 2024
SnapStart Unterstützung für Python und .NET	Lambda SnapStart ist jetzt für verwaltete Python- und .NET-Laufzeiten verfügbar, beginnend mit `python3.12` und `dotnet8`.	18. November 2024
Python-3.13-Laufzeit	Lambda unterstützt jetzt Python 3.13 als verwaltete Laufzeit und Container-Basis-Image.	13. November 2024
Vom Kunden verwaltete Verschlüsselung für .zip-Bereitstellungspakete	Lambda unterstützt jetzt die vom AWS KMS Kunden verwaltete Schlüsselverschlüsselung für .zip-Bereitstellungspakete.	8. November 2024

Support SnapStart in neuen Regionen	Lambda SnapStartist jetzt in den folgenden Regionen erhältlich: Europa (Spanien) , Europa (Zürich), Asien-Paz ifik (Melbourne), Asien-Pazifik (Hyderabad) und Naher Osten (VAE).	12. Januar 2024
AWS verwaltete Richtlini enaktualisierungen	Lambda hat eine bestehend e AWS verwaltete Richtlini e aktualisiert (`AWSLambda VPCAccessExecution Role`).	5. Januar 2024
Python-3.12-Laufzeit	Lambda unterstützt jetzt Python 3.12 als verwaltete Laufzeit und Container-Basis-Image. Weitere Informationen finden Sie unter Python 3.12 Runtime now available AWS Lambda im AWS Compute-B log.	14. Dezember 2023
Java-21-Laufzeit	Lambda unterstützt jetzt Java 21 als Image für die verwaltete Laufzeit und Container-Basis-Image (`java21`).	16. November 2023

Node.js-20.x-Laufzeit	Lambda unterstützt jetzt Node.js 20 als Image für die verwaltete Laufzeit und Container-Basis-Image (`nodejs20.x`). Weitere Informationen finden Sie im AWS LambdaAWS Compute-Blog unter Node.js 20.x-Laufzeit jetzt verfügbar.	14. November 2023
Laufzeit „provided.al2023“	Lambda unterstützt jetzt Amazon Linux 2023 als Image für die verwaltete Laufzeit und Container-Basis-Image. Weitere Informationen finden Sie unter Einführung in die Amazon Linux 2023 Runtime for AWS Lambda im AWS Compute-Blog.	9. November 2023
IPv6 Unterstützung für Dual-Stack-Subnetze	Lambda unterstützt jetzt ausgehenden IPv6 Datenverkehr zu Dual-Stack-Subnetzen. Weitere Informationen finden Sie unter IPv6 Support.	12. Oktober 2023

Serverless-Funktionen und Anwendungen testen	Erfahren Sie mehr über Techniken zum Debuggen und Automatisieren von Tests von Serverless-Funktionen in der Cloud. In den Abschnitten zur Python- und Typescript-Sprache gibt es jetzt ein Kapitel zu Tests sowie Ressourcen. Einzelheiten finden Sie unter Serverless-Funktionen und Anwendungen testen.	16. Juni 2023
Ruby 3.2-Laufzeit	Lambda unterstützt jetzt eine neue Laufzeit für Ruby 3.2. Weitere Informationen finden Sie unter Erstellen von Lambda-Funktionen mit Ruby.	7. Juni 2023
Antwort-Streaming	Lambda unterstützt jetzt Streaming-Antworten von Funktionen. Weitere Informationen finden Sie unter Konfigurieren einer Lambda-Funktion zum Streamen von Antworten.	06. April 2023
Asynchrone Aufrufmetriken	Lambda veröffentlicht asynchrone Aufrufmetriken. Weitere Informationen finden Sie unter Asynchrone Aufrufmetriken.	9. Februar 2023

Kontrollen der Laufzeitversion	Lambda veröffentlicht neue Laufzeitversionen, die Sicherheitsupdates, Fehlerbehebungen und neue Funktionen enthalten. Sie können jetzt steuern, wann Ihre Funktionen auf die neuen Laufzeitversionen aktualisiert werden sollen. Weitere Informationen finden Sie unter Lambda-Laufzeitaktualisierungen.	23. Januar 2023
Lambda SnapStart	Verwenden Sie Lambda SnapStart , um die Startzeit für Java-Funktionen zu reduzieren, ohne zusätzliche Ressourcen bereitzustellen oder komplexe Leistungsoptimierungen zu implementieren. Weitere Informationen finden Sie unter Verbessern der Startleistung mit Lambda SnapStart.	28. November 2022
Node.js-18-Laufzeit	Lambda unterstützt jetzt eine neue Laufzeit für Node.js 18. Node.js 18 verwendet Amazon Linux 2. Weitere Details finden Sie unter Erstellen von Lambda-Funktionen mit Node.js.	18. November 2022

Lambda: Bedingungsschlüssel SourceFunctionArn	Bei einer AWS Ressource filtert der `lambda:SourceFunctionArn` Bedingungsschlüssel den Zugriff auf die Ressource nach dem ARN einer Lambda-Funktion. Einzelheiten dazu finden Sie unter Arbeiten mit Anmeldeinformationen der Lambda-Ausführungsumgebung.	01. Juli 2022
Node.js-16-Laufzeit	Lambda unterstützt jetzt eine neue Laufzeit für Node.js 16. Node.js 16 verwendet Amazon Linux 2. Weitere Details finden Sie unter Erstellen von Lambda-Funktionen mit Node.js.	11. Mai 2022
Lambda-Funktion URLs	Lambda unterstützt jetzt Funktionen URLs, bei denen es sich um dedizierte HTTP (S) -Endpunkte für Lambda-Funktionen handelt. Einzelheiten finden Sie unter Lambda-Funktion URLs.	6. April 2022
Geteilte Testereignisse in der Konsole AWS Lambda	Lambda unterstützt jetzt das Teilen von Testereignissen mit anderen Benutzern im selben AWS-Konto. Details finden Sie unter Testen von Lambda-Funktionen mit Hilfe der Konsole.	16. März 2022

PrincipalOrgId in ressourcenbasierten Richtlinien	Lambda unterstützt jetzt das Erteilen von Berechtigungen für eine Organisation in AWS Organizations. Details dazu finden Sie unter Verwenden ressourcenbasierter Richtlinien für AWS Lambda.	11. März 2022
.NET-6-Laufzeit	Lambda unterstützt jetzt eine neue Laufzeit für .NET 6. Weitere Details finden Sie unter Lambda-Laufzeiten.	23. Februar 2022
Ereignisfilterung für Kinesis-, DynamoDB- und Amazon-SQS-Ereignisquellen	Lambda unterstützt jetzt die Ereignisfilterung für Kinesis-, DynamoDB- und Amazon-SQS-Ereignisquellen. Details dazu finden Sie unter Lambda-Ereignisfilterung.	24. November 2021
mTLS-Authentifizierung für Amazon MSK und selbstverwaltete Apache-Kafka-Ereignisquellen	Lambda unterstützt ab sofort die mTLS-Authentifizierung für Amazon MSK und selbstverwaltete Apache-Kafka-Ereignisquellen. Weitere Details finden Sie unter Verwendung von Lambda mit Amazon MSK.	19. November 2021
Lambda auf Graviton2	Lambda unterstützt jetzt Graviton2 für Funktionen, die arm64-Architektur verwenden. Details dazu finden Sie unter .Lambda-Befehlssatz-Architekturenaus.	29. September 2021

Python-3.9-Laufzeit	Lambda unterstützt jetzt eine neue Laufzeit für Python 3.9. Weitere Details finden Sie unter Lambda-Laufzeiten.	16. August 2021
Neue Laufzeitversionen für Node.js, Python und Java	Neue Laufzeiten sind für Node.js, Python und Java verfügbar. Weitere Details finden Sie unter Lambda-Laufzeiten.	21. Juli 2021
Support für RabbitMQ als Ereignisquelle für Lambda	Lambda bietet jetzt Unterstützung für Amazon MQ für RabbitMQ als Ereignisquelle. Amazon MQ ist ein verwalteter Message-Broker-Service für Apache ActiveMQ und RabbitMQ, der das Einrichten und Betreiben von Message-Brokern in der Cloud vereinfacht. Weitere Details finden Sie unter Verwendung von Lambda mit Amazon MQ.	7. Juli 2021

SASL/PLAIN-Authentifizierung für selbstverwaltete Kafka in Lambda	SASL/PLAIN is now a supported authentication mechanism for self-managed Kafka event sources on Lambda Customers already using SASL/PLAINauf ihrem selbstverwalteten Kafka-Cluster können Lambda nun problemlos zum Erstellen von Verbraucheranwendungen verwenden, ohne die Art der Authentifizierung ändern zu müssen. Details dazu finden Sie unter Verwenden von Lambda mit selbstverwaltetem Apache Kafka.	29. Juni 2021
Lambda-Erweiterungs-API	Allgemeine Verfügbarkeit von Lambda-Erweiterungen. Erweiterungen verwenden, um Ihre Lambda-Funktionen zu erweitern. Sie können Erweiterungen verwenden, die von Lambda-Partnern bereitgestellt werden, oder Sie können Ihre eigenen Lambda-Erweiterungen erstellen. Weitere Details finden Sie unter Lambda-Erweiterungs-API.	24. Mai 2021
Neues Lambda-Konsolenerlebnis	Die Lambda-Konsole wurde überarbeitet, um die Leistung und Konsistenz zu verbessern.	2. März 2021

Node.js-14-Laufzeit	Lambda unterstützt jetzt eine neue Laufzeit für Node.js 14. Node.js 14 verwendet Amazon Linux 2. Weitere Details finden Sie unter Erstellen von Lambda-Funktionen mit Node.js.	27. Januar 2021
Lambda-Container-Images	Lambda unterstützt jetzt als Container-Images definiert e Funktionen. Sie können die Flexibilität von Container -Tooling mit der Agilität und der betrieblichen Einfachhe it von Lambda kombinier en, um Anwendungen zu bauen. Weitere Details finden Sie unter Verwenden von Container-Images mit Lambda.	1. Dezember 2020

Codesignatur für Lambda-Funktionen	Lambda unterstützt jetzt Codesignatur. Administratoren können Lambda-Funktionen so konfigurieren, dass sie bei der Bereitstellung nur signierten Code akzeptieren. Lambda überprüft die Signaturen, um sicherzustellen, dass der Code nicht verändert oder manipuliert wird. Zusätzlich sorgt Lambda dafür, dass der Code von vertrauenswürdigen Entwicklern unterzeichnet wurde, bevor die Bereitstellung zugelassen wird. Weitere Details finden Sie unter Konfigurieren von Codesignatur für Lambda.	23. November 2020
Vorschau: Lambda Runtime Logs API	Lambda unterstützt jetzt die Runtime Logs API. Lambda-Erweiterungen können die Logs-API verwenden, um Protokollstreams in der Ausführungsumgebung zu abonnieren. Weitere Details finden Sie unter Lambda Runtime Logs API.	12. November 2020

Neue Ereignisquelle für Amazon MQ	Lambda bietet jetzt Unterstützung für Amazon MQ als Ereignisquelle. Verwenden Sie eine Lambda-Funktion, um Datensätze von Ihrem Amazon-MQ-Message-Broker zu verarbeiten. Weitere Details finden Sie unter Verwendung von Lambda mit Amazon MQ.	5. November 2020
Vorschau: Lambda-Erweiterungs-API	Lambda-Erweiterungen verwenden, um Ihre Lambda-Funktionen zu erweitern. Sie können Erweiterungen verwenden, die von Lambda-Partnern bereitgestellt werden, oder Sie können Ihre eigenen Lambda-Erweiterungen erstellen. Weitere Details finden Sie unter Lambda-Erweiterungs-API.	8. Oktober 2020
Support für Java 8 und benutzerdefinierte Laufzeiten auf AL2	Lambda unterstützt jetzt Java 8 und benutzerdefinierte Laufzeiten unter Amazon Linux 2. Weitere Details finden Sie unter Lambda-Laufzeiten.	12. August 2020

Neue Ereignisquelle für Amazon Managed Streaming for Apache Kafka	Lambda bietet jetzt Unterstützung für Amazon MSK als Ereignisquelle. Verwenden Sie eine Lambda-Funktion mit Amazon MSK, um Datensätze in einem Kafka-Thema zu verarbeiten. Weitere Details finden Sie unter Verwendung von Lambda mit Amazon MSK.	11. August 2020
Verwenden von IAM-Bedingungsschlüsseln für Amazon-VPC-Einstellungen	Sie können nun Lambda-spezifische Bedingungsschlüssel für VPC-Einstellungen verwenden. Sie können beispielsweise festlegen, dass alle Funktionen in Ihrer Organisation mit einer VPC sein müssen. Sie können auch die Subnetze und Sicherheitsgruppen angeben, die die Benutzer der Funktion verwenden können bzw. nicht verwenden können. Weitere Details finden Sie unter Konfigurieren einer VPC für IAM-Funktionen.	10. August 2020

Einstellungen für Parallelbetrieb für Kinesis-HTTP/2-Stream-Konsumenten	Sie können jetzt die folgenden Parallelitätseinstellungen für Kinesis-Verbraucher mit erweitertem Fan-Out (HTTP/2-Streams) verwenden : ParallelizationFactor,, MaximumRetryAttempts, MaximumRecordAgeIn Seconds und. DestinationConfig BisectBatchOnFunctionError Einzelheiten finden Sie unter Verwenden AWS Lambda mit Amazon Kinesis.	7. Juli 2020
Batch-Fenster für Kinesis HTTP/2-Stream-Konsumenten	Sie können jetzt ein Batch-Fenster (MaximumBatchingWindowInSeconds) für HTTP/2-Streams konfigurieren. Lambda liest Datensätze aus dem Stream, bis es einen vollständigen Batch gesammelt hat oder bis das Batchfenster abläuft. Einzelheiten finden Sie unter Verwenden AWS Lambda mit Amazon Kinesis.	18. Juni 2020
Support für Amazon-EFS-Dateisysteme	Sie können nun ein Amazon-EFS-Dateisystem mit Ihren Lambda-Funktionen für den freigegebenen Netzwerkdateizugriff verbinden. Weitere Details finden Sie unter Konfigurieren des Dateisystemzugriffs für Lambda-Funktionen.	16. Juni 2020

AWS CDK Beispielanwendungen in der Lambda-Konsole	Die Lambda-Konsole enthält jetzt Beispielanwendungen, die das AWS Cloud Development Kit (AWS CDK) for TypeScript verwenden. Das AWS CDK ist ein Framework, mit dem Sie Ihre Anwendungsressourcen in Python TypeScript, Java oder .NET definieren können.	1. Juni 2020
Support für.NET Core 3.1.0 Runtime in AWS Lambda	AWS Lambda unterstützt jetzt die.NET Core 3.1.0-Runtime. Weitere Details finden Sie unter .NET Core CLI.	31. März 2020
Support für API Gateway HTTP APIs	Aktualisierte und erweiterte Dokumentation für die Verwendung von Lambda mit API Gateway, einschließlich Unterstützung für HTTP APIs. Es wurde eine Beispielanwendung hinzugefügt, die eine API erstellt und mit CloudFormation funktioniert. Weitere Details finden Sie unter Verwendung von Lambda mit Amazon API Gateway.	23. März 2020

Ruby 2.7	Für Ruby 2.7 steht eine neue Laufzeit, ruby2.7, zur Verfügung. Dies ist die erste Ruby-Laufzeitumgebung, die Amazon Linux 2 verwendet . Weitere Details finden Sie unter Erstellen von Lambda-Funktionen mit Ruby.	19. Februar 2020
Gleichzeitigkeitsmetriken	Lambda meldet nun die Metrik `ConcurrentExecutions` für alle Funktionen, Aliase und Versionen. Sie können ein Diagramm für diese Metrik auf der Überwachungsseite für Ihre Funktion anzeigen. Bisher wurde `ConcurrentExecutions` nur auf Kontoebene und für Funktionen gemeldet, die reservierte Gleichzeitigkeit verwenden . Details hierzu finden Sie unter AWS Lambda -Funktionsmetriken.	18. Februar 2020

Aktualisieren auf Funktionszustände	Funktionszustände werden jetzt standardmäßig für alle Funktionen erzwungen. Wenn Sie eine Funktion mit einer VPC verbinden, erstellt Lambda gemeinsame Elastic-Network-Schnittstellen. Dadurch kann Ihre Funktion skaliert werden, ohne zusätzliche Netzwerkschnittstellen erstellen zu müssen. Während dieser Zeit können Sie keine zusätzlichen Operationen für die Funktion ausführen, einschließlich der Aktualisierung der Konfiguration und der Veröffentlichung von Versionen. In einigen Fällen ist der Aufruf ebenfalls betroffen. Details zum aktuellen Status einer Funktion sind über die Lambda-API verfügbar. Diese Aktualisierung wird phasenweise veröffentlicht. Einzelheiten finden Sie im AWS Compute-Blog unter Aktualisierter Lambda-States-Lebenszyklus für VPC-Netzwerke. Weitere Informationen zu Zuständen finden Sie unter AWS Lambda -Funktionszustände.	24. Januar 2020

Aktualisierungen der API-Ausgabe der Funktionskonfiguration	Ursachencodes zu StateReasonCode(InvalidSubnet, InvalidSecurityGroup) und LastUpdateStatusReasonCode (SubnetOutOfIPAddresses, InvalidSubnet, InvalidSecurityGroup) für Funktionen hinzugefügt, die eine Verbindung zu einer VPC herstellen. Weitere Informationen zu Zuständen finden Sie unter AWS Lambda -Funktionszustände.	20. Januar 2020
Bereitgestellte Gleichzeitigkeit	Sie haben nun die Möglichkeit, bereitgestellte Gleichzeitigkeit (Provisioned Concurrency) für eine Funktionsversion oder einen Alias zuzuweisen. Mit der bereitgestellten Gleichzeitigkeit kann eine Funktion ohne Schwankungen der Latenz skalieren. Weitere Details finden Sie unter Verwalten von Gleichzeitigkeit für eine Lambda-Funktion.	03. Dezember 2019

Erstellen eines Datenbank-Proxys	Sie können nun mit der Lambda-Konsole einen Datenbank-Proxy für eine Lambda-Funktion erstellen . Ein Datenbank-Proxy ermöglicht es einer Funktion, hohe Gleichzeitigkeitsstufen ohne überlastende Datenbank verbindungen zu erreichen . Weitere Details finden Sie unter Konfigurieren des Datenbankzugriffs für eine Lambda-Funktion.	03. Dezember 2019
Perzentilunterstützung für die Dauer-Metrik	Sie können nun die Dauer-Metrik basierend auf Perzentil en filtern. Weitere Details finden Sie unter AWS Lambda -Metriken.	26. November 2019
Höhere Gleichzeitigkeit für Stream-Ereignisquellen	Eine neue Option für das DynamoDB-Stream- und Kinesis-Stream-Ereignis quellen-Mapping ermöglich t es Ihnen, von jedem Shard mehrere Batches gleichzeitig zu verarbeiten. Wenn Sie die Anzahl gleichzeitiger Stapel pro Shard erhöhen, kann die Gleichzeitigkeit der Funktion bis zu zehnmal so hoch sein wie die Anzahl der Shards in Ihrem Stream. Weitere Details finden Sie unter Lambda-Er eignisquellen-Zuweisung.	25. November 2019

Funktionszustände	Wenn Sie eine Funktion erstellen oder aktualisieren, wechselt diese in einen ausstehenden Zustand, während Lambda Ressourcen zur Unterstützung bereitstellt. Wenn Sie Ihre Funktion mit einer VPC verbinden, kann Lambda sofort eine gemeinsame Elastic-Network-Schnittstelle erstellen, anstatt Netzwerkschnittstellen zu erstellen, wenn Ihre Funktion aufgerufen wird. Dies verbessert die Leistung von mit der VPC verbundenen Funktionen, erfordert aber möglicherweise eine Aktualisierung Ihrer Automatisierung. Details hierzu finden Sie unter AWS Lambda -Funktionszustände.	25. November 2019
Fehlerbehandlungsoptionen für den asynchronen Aufruf	Für den asynchronen Aufruf stehen neue Konfigurationsoptionen zur Verfügung. Sie können Lambda so konfigurieren, dass Wiederholversuche begrenzt und ein maximales Ereignisalter festgelegt werden. Weitere Details finden Sie unter Konfigurieren der Fehlerbehandlung für den asynchronen Aufruf.	25. November 2019

Fehlerbehandlung für Stream-Ereignisquellen	Es sind neue Konfigurationsoptionen für Ereignisquellen-Zuweisungen verfügbar, die aus Streams gelesen werden. Sie können DynamoDB-Stream- und Kinesis-Stream-Ereignisquellen-Zuweisung konfigurieren, um Wiederholversuche zu begrenzen und ein maximales Datensatzalter festzulegen. Wenn Fehler auftreten, können Sie das Ereignisquellen-Mapping so konfigurieren, dass Batches vor dem erneuten Versuch aufgeteilt und Aufrufdatensätze für fehlgeschlagene Batches an eine Warteschlange oder an ein Thema gesendet werden. Weitere Details finden Sie unter Lambda-Ereignisquellen-Mapping.	25. November 2019

Ziele für den asynchronen Aufruf	Sie können Lambda jetzt so konfigurieren, dass Datensätze asynchroner Aufrufe an einen anderen Service gesendet werden. Aufrufdatensätze enthalten Details zu Ereignis, Kontext und Antwort der Funktion. Sie können Aufrufdatensätze an eine SQS-Warteschlange, ein SNS-Thema, eine Lambda-Funktion oder einen Ereignisbus senden. EventBridge Weitere Details finden Sie unter Konfigurieren der Ziele für asynchronen Aufruf.	25. November 2019
Neue Laufzeiten für Node.js, Python und Java	Neue Laufzeiten sind für Node.js 12, Python 3.8 und Java 11 verfügbar. Weitere Details finden Sie unter Lambda-Laufzeiten.	18. November 2019
FIFO-Warteschlangenunterstützung für Amazon-SQS-Ereignisquellen	Sie können jetzt ein Ereignisquellen-Mapping erstellen, die aus einer FIFO (First-in-First-out)-Warteschlange liest. Bisher wurden nur Standardwarteschlangen unterstützt. Weitere Details finden Sie unter Verwendung von Lambda mit Amazon SQS.	18. November 2019

Erstellen von Anwendungen in der Lambda-Konsole	Die Anwendungserstellung in der Lambda-Konsole ist jetzt allgemein verfügbar. Anweisungen finden Sie unter Anwendungen in der Lambda-Konsole verwalten.	31. Oktober 2019
Erstellen von Anwendungen in der Lambda-Konsole (Beta)	Sie können nun eine Lambda-Anwendung mit einer integrierten fortlaufenden Bereitstellungs-Pipeline in der Lambda-Konsole erstellen. Die Konsole stellt Beispielanwendungen bereit, die Sie als Ausgangspunkt für Ihr eigenes Projekt verwenden können. Wählen Sie zwischen AWS CodeCommit und GitHub für die Quellcodeverwaltung. Jedes Mal, wenn Sie Änderungen zu Ihrem Repository übertragen, wird die enthaltene Pipeline automatisch erstellt und bereitgestellt. Anweisungen finden Sie unter Anwendungen in der Lambda-Konsole verwalten.	3. Oktober 2019

Leistungsverbesserungen für VPC-verbundene Funktionen	Lambda verwendet jetzt eine neue Art von Elastic-Network-Schnittstelle, die von allen Funktionen in einem Virtual-Private-Cloud-(VPC)-Subnetz gemeinsam genutzt wird. Wenn Sie eine Funktion mit einer VPC verbinden, erstellt Lambda eine Netzwerkschnittstelle für jede Kombination aus Sicherheitsgruppe und Subnetz, die Sie auswählen. Wenn die freigegebenen Netzwerkschnittstellen verfügbar sind, muss die Funktion bei der Skalierung nach oben keine zusätzlichen Netzwerkschnittstellen erstellen. Dies verbessert die Startup-Zeiten erheblich. Weitere Details finden Sie unter Konfigurieren einer Lambda-Funktion für den Zugriff auf Ressourcen in einer VPC.	03. September 2019

Stream-Batch-Einstellungen	Sie können jetzt ein Batch-Fenster für Amazon-DynamoDB- und Amazon-Kinesis-Ereignisquellen-Mappings konfigurieren. Konfigurieren Sie ein Batch-Fenster von bis zu fünf Minuten, um eingehende Datensätze zu puffern, bis ein vollständiger Batch verfügbar ist. Dies reduziert die Anzahl der Aufrufe Ihrer Funktion, wenn der Stream weniger aktiv ist.	29. August 2019
CloudWatch Integration von Logs & Insights	Die Überwachungsseite in der Lambda-Konsole enthält jetzt Berichte von Amazon CloudWatch Logs Insights.	18. Juni 2019
Amazon Linux 2018.03	Die Lambda-Ausführungsumgebung wurde aktualisiert und verwendet jetzt Amazon Linux 2018.03. Weitere Details finden Sie unter Ausführungsumgebung.	21. Mai 2019

Node.js 10	Eine neue Laufzeit ist verfügbar für Node.js 10, nodejs10.x. Diese Laufzeit verwendet Node.js 10.15 und wird regelmäßig mit der neuesten Version von Node.js 10 aktualisiert. Node.js 10 ist auch die erste Laufzeit, die Amazon Linux 2 verwendet . Weitere Details finden Sie unter Erstellen von Lambda-Funktionen mit Node.js.	13. Mai 2019
GetLayerVersionByArn API	Verwenden Sie die GetLayerVersionByArnAPI, um Layer-Versionsinformationen mit dem Versions-ARN als Eingabe herunterzuladen. Im Vergleich zu GetLayerVersionByArn können Sie den ARN direkt verwenden GetLayerVersion, anstatt ihn zu analysieren, um den Layer-Namen und die Versionsnummer abzurufen.	25. April 2019
Ruby	AWS Lambda unterstützt jetzt Ruby 2.5 mit einer neuen Runtime. Weitere Details finden Sie unter Erstellen von Lambda-Funktionen mit Ruby.	29. November 2018

Ebenen	Mit Lambda-Ebenen können Sie Bibliotheken, benutzerdefinierte Laufzeiten und andere Abhängigkeiten getrennt von Ihrem Funktionscode verpacken und bereitstellen. Geben Sie Ihre Ebenen für andere Konten oder die ganze Welt frei. Weitere Details finden Sie unter Lambda-Ebenen.	29. November 2018
Benutzerdefinierte Laufzeiten	Erstellen Sie eine benutzerdefinierte Laufzeit zur Ausführung von Lambda-Funktionen in Ihrer bevorzugten Programmiersprache. Weitere Details finden Sie unter Benutzerdefinierte Lambda-Laufzeiten.	29. November 2018
Application-Load-Balancer-Auslöser	Elastic Load Balancing unterstützt nun Lambda-Funktionen als Ziel für Application Load Balancers. Details finden Sie unter Verwenden von Lambda mit Application Load Balancern.	29. November 2018

Verwenden von Kinesis-HTTP/2-Stream-Konsumenten als Auslöser	Sie können mit Kinesis HTTP/2-Daten-Stream-Konsumenten Ereignisse an sende AWS Lambda. Stream-Konsumenten haben einen dedizierten Lesedurchsatz aus jedem Shard in Ihrem Daten-Stream und verwenden HTTP/2, um die Latenz zu minimieren. Details finden Sie unter Verwenden von Lambda mit Kinesis.	19. November 2018
Python 3.7	AWS Lambda unterstützt jetzt Python 3.7 mit einer neuen Runtime. Weitere Informationen finden Sie unter Erstellen von Lambda-Funktionen mit Python.	19. November 2018
Erhöhung des Nutzlastlimits für asynchronen Funktionsaufruf	Die maximale Nutzlastgröße für asynchrone Aufrufe wurde von 128 KB auf 256 KB erhöht, was der maximale Nachrichtengröße eines Amazon-SNS-Auslösers entspricht. Weitere Details finden Sie unter Lambda-Kontingente.	16. November 2018
AWS GovCloud Region (USA-Ost)	AWS Lambda ist jetzt in der Region AWS GovCloud (USA-Ost) verfügbar.	12. November 2018

Die AWS SAM Themen wurden in ein separates Entwicklerhandbuch verschoben	Eine Reihe von Themen befasste sich mit der Entwicklung serverloser Anwendungen mithilfe von AWS Serverless Application Model (AWS SAM). Diese Themen wurden in das AWS Serverless Application Model -Entwicklerhandbuch verschoben.	25. Oktober 2018
Anzeigen von Lambda-Anwendungen in der Konsole	Sie können den Status Ihrer Lambda-Anwendungen auf der Seite Anwendungen in der Lambda-Konsole anzeigen. Auf dieser Seite wird der Status des CloudFormation Stacks angezeigt. Sie enthält Links zu Seiten mit weiteren Informationen zu den Ressourcen im Stack. Sie können auch aggregierte Metriken für die Anwendung anzeigen und benutzerdefinierte Überwachungs-Dashboards erstellen.	11. Oktober 2018
Timeout-Limit für die Funktionsausführung	Für langlaufende Funktionen wurde das maximal konfigurierbare Ausführungs-Timeout von 5 Minuten auf 15 Minuten erhöht. Weitere Details finden Sie unter Lambda-Limits.	10. Oktober 2018

Support für die PowerShell Kernsprache in AWS Lambda	AWS Lambda unterstützt jetzt die PowerShell Core-Sprache. Weitere Informationen finden Sie unter Programmiermodell für die Erstellung von Lambda-Funktionen in. PowerShell	11. September 2018
Support für.NET Core 2.1.0 Runtime in AWS Lambda	AWS Lambda unterstützt jetzt die.NET Core 2.1.0 Runtime. Weitere Informationen finden Sie unter .NET Core-CLI.	9. Juli 2018
Updates jetzt über RSS verfügbar	Sie können jetzt einen RSS-Feed abonnieren, um Releases für dieses Handbuch nachzufolgen.	5. Juli 2018
Unterstützung für Amazon SQS als Ereignisquelle	AWS Lambda unterstützt jetzt Amazon Simple Queue Service (Amazon SQS) als Ereignisquelle. Weitere Informationen finden Sie unter Aufrufen von Lambda-Funktionen.	28. Juni 2018
China (Ningxia)-Region	AWS Lambda ist jetzt in der Region China (Ningxia) verfügbar. Weitere Informationen zu Lambda-Regionen und -Endpunkten finden Sie unter Regionen und Endpunkte in der Allgemeine AWS-Referenz.	28. Juni 2018

Frühere Updates

In der folgenden Tabelle sind die wichtigen Änderungen in jeder Version des AWS Lambda - Entwicklerhandbuchs vor Juni 2018 beschrieben.

Änderung	Beschreibung	Datum
Laufzeitunterstützung für Node.js-Laufzeitversion 8.10	AWS Lambda unterstützt jetzt die Runtime-Version 8.10 von Node.js. Weitere Informationen finden Sie unter Erstellen von Lambda-Funktionen mit Node.js.	2. April 2018
Funktions- und Alias-Revision IDs	AWS Lambda unterstützt jetzt die Revision IDs Ihrer Funktionsversionen und Aliase. Sie können diese verwenden IDs , um bedingte Aktualisierungen nachzuverfolgen und anzuwenden, wenn Sie Ihre Funktionsversion oder Aliasressourcen aktualisieren.	25. Januar 2018
Unterstützung für Go und .NET 2.0 bei Laufzeit	AWS Lambda hat Runtime-Unterstützung für Go und .NET 2.0 hinzugefügt. Weitere Informationen erhalten Sie unter Erstellen von Lambda-Funktionen mit Go und Erstellen von Lambda-Funktionen mit C#.	15. Januar 2018
Umgestaltung der Konsole	AWS Lambda hat eine neue Lambda-Konsole eingeführt, um Ihre Erfahrung zu vereinfachen, und einen Cloud9-Code-Editor hinzugefügt, mit dem Sie Ihren Funktionscode besser debuggen und überarbeiten können.	30. November 2017
Festlegen von Gleichzeitigkeitslimits für einzelne Funktionen	AWS Lambda unterstützt jetzt das Setzen von Parallelitätslimits für einzelne Funktionen. Weitere Informationen finden Sie unter Konfigurieren reservierter Gleichzeitigkeit für eine Funktion.	30. November 2017
Verschiebung von Datenverkehr mithilfe von Aliassen	AWS Lambda unterstützt jetzt die Verlagerung von Traffic mit Aliasnamen. Weitere Informationen finden Sie unter Erstellen Sie eine fortlaufende Bereitstellung mit gewichteten Aliasnamen.	28. November 2017

Änderung	Beschreibung	Datum
Graduelle Code-Bereitstellung	AWS Lambda unterstützt jetzt die sichere Bereitstellung neuer Versionen Ihrer Lambda-Funktion mithilfe von Code Deploy. Weitere Informationen finden Sie unter Graduelle Code-Bereitstellung.	28. November 2017
Region China (Peking)	AWS Lambda ist jetzt in der Region China (Peking) verfügbar. Weitere Informationen zu Lambda-Regionen und -Endpunkten finden Sie unter Regionen und Endpunkte in der Allgemeine AWS-Referenz.	9. November 2017
Einführung von SAM Local	AWS Lambda stellt SAM Local (jetzt bekannt als SAM CLI) vor, ein AWS CLI Tool, das Ihnen eine Umgebung bietet, in der Sie Ihre serverlosen Anwendungen lokal entwickeln, testen und analysieren können, bevor Sie sie in die Lambda-Laufzeit hochladen. Weitere Informationen finden Sie unter Testen und Debuggen von Serverless Anwendungen.	11. August 2017
Region Kanada (Zentral)	AWS Lambda ist jetzt in der Region Kanada (Zentral) verfügbar. Weitere Informationen zu Lambda-Regionen und -Endpunkten finden Sie unter Regionen und Endpunkte in der Allgemeine AWS-Referenz.	22. Juni 2017
South America (São Paulo) Region	AWS Lambda ist jetzt in der Region Südamerika (São Paulo) verfügbar. Weitere Informationen zu Lambda-Regionen und -Endpunkten finden Sie unter Regionen und Endpunkte in der Allgemeine AWS-Referenz.	6. Juni 2017
AWS Lambda Unterstützung für AWS X-Ray.	In Lambda wird die Unterstützung von X-Ray eingeführt; damit können Sie Leistungsprobleme bei Ihren Lambda-Anwendungen erkennen, analysieren und optimieren. Weitere Informationen finden Sie unter Visualisieren Sie Lambda-Funktionsaufrufe mit AWS X-Ray.	19. April 2017

Änderung	Beschreibung	Datum
Region Asien-Pazifik (Mumbai)	AWS Lambda ist jetzt in der Region Asien-Pazifik (Mumbai) verfügbar. Weitere Informationen zu Lambda-Regionen und -Endpunkten finden Sie unter Regionen und Endpunkte in der Allgemeine AWS-Referenz.	28. März 2017
AWS Lambda unterstützt jetzt Node.js Runtime v6.10	AWS Lambda Unterstützung für Node.js Runtime v6.10 hinzugefügt. Weitere Informationen finden Sie unter Erstellen von Lambda-Funktionen mit Node.js.	22. März 2017
Region Europa (London)	AWS Lambda ist jetzt in der Region Europa (London) verfügbar. Weitere Informationen zu Lambda-Regionen und -Endpunkten finden Sie unter Regionen und Endpunkte in der Allgemeine AWS-Referenz.	1. Februar 2017
AWS Lambda Unterstützung für.NET-Runtime, Lambda @Edge (Preview), Dead Letter Queues und automatisiertes Deployment serverloser Anwendungen.	AWS Lambda Unterstützung für C# hinzugefügt. Weitere Informationen finden Sie unter Erstellen von Lambda-Funktionen mit C#. Mit Lambda @Edge können Sie Lambda-Funktionen an den AWS Edge-Standorten als Reaktion auf CloudFront Ereignisse ausführen. Weitere Informationen finden Sie unter Anpassen am Edge mit Lambda @Edge.	3. Dezember 2016
AWS Lambda fügt Amazon Lex als unterstützte Ereignisquelle hinzu.	Mit Lambda und Amazon Lex können Sie Chatbots für verschiedene Services wie Slack und Facebook schnell erstellen.	30. November 2016
US West (N. California) Region	AWS Lambda ist jetzt in der Region USA West (Nordkalifornien) verfügbar. Weitere Informationen zu Lambda-Regionen und -Endpunkten finden Sie unter Regionen und Endpunkte in der Allgemeine AWS-Referenz.	21. November 2016

Änderung	Beschreibung	Datum
Einführung der Einstellungen AWS SAM für die Erstellung und Bereitstellung von Lambda-basierten Anwendungen und die Verwendung von Umgebungsvariablen für die Konfiguration von Lambda-Funktionen.	AWS SAM: Sie können jetzt den verwenden, AWS SAM um die Syntax für das Ausdrücken von Ressourcen in einer serverlosen Anwendung zu definieren. Sie stellen Ihre Anwendung bereit, indem Sie die benötigten Ressourcen einfach zusammen mit den zugehörigen Berechtigungsrichtlinien als Teil der Anwendung in einer AWS CloudFormation -Vorlagendatei angeben (die in JSON oder YAML geschrieben ist), Ihre Bereitstellungsartefakte paketieren und die Vorlage bereitstellen. Umgebungsvariablen: Sie können Umgebungsvariablen verwenden, um Konfigurationseinstellungen für Ihre Lambda-Funktion vorzunehmen, ohne dabei den Code für die Funktion ändern zu müssen. Weitere Informationen finden Sie unter Arbeiten mit Lambda-Umgebungsvariablen.	18. November 2016
Region Asien-Pazifik (Seoul)	AWS Lambda ist jetzt in der Region Asien-Pazifik (Seoul) verfügbar. Weitere Informationen zu Lambda-Regionen und -Endpunkten finden Sie unter Regionen und Endpunkte in der Allgemeine AWS-Referenz.	29. August 2016
Asia Pacific (Sydney) Region	Lambda ist jetzt in der Region Asien-Pazifik (Sydney) verfügbar. Weitere Informationen zu Lambda-Regionen und -Endpunkten finden Sie unter Regionen und Endpunkte in der Allgemeine AWS-Referenz.	23. Juni 2016
Aktualisierungen für die Lambda-Konsole	Die Lambda-Konsole wurde aktualisiert, um das Verfahren für die Erstellung von Rollen zu vereinfachen.	23. Juni 2016
AWS Lambda unterstützt jetzt Node.js Runtime v4.3	AWS Lambda Unterstützung für Node.js Runtime v4.3 hinzugefügt. Weitere Informationen finden Sie unter Erstellen von Lambda-Funktionen mit Node.js.	07. April 2016

Änderung	Beschreibung	Datum
Region Europa (Frankfurt)	Lambda ist jetzt in der Region Europa (Frankfurt) verfügbar . Weitere Informationen zu Lambda-Regionen und -Endpunkten finden Sie unter Regionen und Endpunkte in der Allgemeine AWS-Referenz.	14. März 2016
VPC-Unterstützung	Sie können eine Lambda-Funktion jetzt für den Zugriff auf Ressourcen in Ihrer VPC konfigurieren. Weitere Informati onen finden Sie unter Lambda-Funktionen Zugriff auf Ressourcen in einer Amazon VPC gewähren.	11. Februar 2016
Die Lambda-La ufzeitumgebung wurde aktualisiert.	Die Ausführungsumgebung wurde aktualisiert.	4. November 2015

Änderung	Beschreibung	Datum
Unterstützung der Versioning, Python für die Entwicklung von Code für Lambda-Funktionen, geplante Ereignisse und Erhöhung der Ausführungszeit	Sie können den Code für Ihre Lambda-Funktionen jetzt mit Python entwickeln. Weitere Informationen finden Sie unter Erstellen von Lambda-Funktionen mit Python. Versioning: Sie können eine oder mehrere Versionen einer Lambda-Funktion pflegen. Mit der Versioning können Sie steuern, welche Versionen einer Lambda-Funktion in verschiedenen Umgebungen ausgeführt werden (z. B. in einer Entwicklungs-, Test- oder produktiven Umgebung). Weitere Informationen finden Sie unter Versionen der Lambda-Funktion verwalten. Geplante Ereignisse: Außerdem können Sie Lambda mithilfe der Lambda-Konsole so einrichten, dass Ihr Code regelmäßig und nach Plan abgerufen wird. Sie können einen festen Takt für die Ausführung (eine Anzahl von Stunden, Tagen oder Wochen) oder einen Cron-Ausdruck angeben. Weitere Informationen finden Sie unter Aufrufen einer Lambda-Funktion nach einem Zeitplan. Erhöhung der Ausführungszeit: Sie können Ihre Lambda-Funktion jetzt so einrichten, dass sie bis zu fünf Minuten lang ausgeführt wird; dadurch werden Funktionen mit längeren Laufzeiten ermöglicht, z. B. Aufträge für die Erfassung oder Verarbeitung großer Datenvolumen.	08. Oktober 2015

Änderung	Beschreibung	Datum
Unterstützung für DynamoDB Streams	DynamoDB Streams ist jetzt allgemein verfügbar; Sie können die Funktion in allen Regionen verwenden, in denen DynamoDB verfügbar ist. Sie können DynamoDB Streams für Ihre Tabelle aktivieren und eine Lambda-Funktion als Auslöser für die Tabelle verwenden. Auslöser sind benutzerdefinierte Aktionen, die als Reaktion auf Aktualisierungen der DynamoDB-Tabelle ausgeführt werden. Ein Beispiel-Walkthrough finden Sie unte Tutorial: Verwendung AWS Lambda mit Amazon DynamoDB DynamoDB-Streams .	14. Juli 2015
Lambda unterstützt jetzt den Aufruf von Lambda-Funktionen mit REST-kompatiblen Clients.	Bisher benötigten Sie, um Ihre Lambda-Funktion von Ihrer Web-, Mobil- oder IoT-Anwendung aus aufzurufen AWS SDKs (z. B. AWS SDK for Java, AWS SDK for Android oder AWS SDK for iOS). Lambda unterstützt jetzt Aufrufe einer Lambda-Funktion mit REST-kompatiblen Clients über eine benutzerdefinierte API, die Sie mithilfe des Amazon API Gateway erstellen können. Sie können Anfragen an die Endpunkt-URL Ihrer Lambda-Funktion senden. Sie können die Sicherheitseinstellungen an dem Endpunkt für einen offenen Zugriff konfigurieren, den Zugriff mithilfe von AWS Identity and Access Management (IAM) autorisieren oder API-Schlüssel verwenden, um den Zugriff auf Lambda-Funktionen durch andere Benutzer zu erfassen. Ein Beispielübung mit ersten Schritten finden Sie unter Aufrufen einer Lambda-Funktion über einen Amazon API Gateway-Endpunkt.	09. Juli 2015

Änderung	Beschreibung	Datum
In der Lambda-Konsole werden jetzt Vorlagen bereitgestellt, mit denen Sie Lambda-Funktionen einfach erstellen und testen können.	In der Lambda-Konsole wird eine Reihe von Vorlagen bereitgestellt. Jede Vorlage enthält eine Beispielkonfiguration für eine Ereignisquelle sowie Beispielcode für eine Lambda-Funktion, mit dem Sie Lambda-basierte Anwendungen einfach erstellen können. Die Vorlagen sind jetzt in allen Lambda-Übungen mit ersten Schritten enthalten.	09. Juli 2015
Lambda unterstützt jetzt Lambda-Funktionen, die in Java geschrieben wurden.	Sie können Lambda-Code jetzt in Java verfassen. Weitere Informationen finden Sie unter Erstellen von Lambda-Funktionen mit Java.	15. Juni 2015
Lambda unterstützt bei der Erstellung oder Aktualisierung einer Lambda-Funktion jetzt die Angabe eines Amazon-S3-Objekts als ZIP-Datei.	Sie können das Bereitstellungspaket für eine Lambda-Funktion (ZIP-Datei) in einen Amazon-S3-Bucket in derselben Region uploaden, in der Sie die Lambda-Funktion erstellen möchten. Danach können Sie bei der Erstellung oder Aktualisierung der Lambda-Funktion den Bucket-Namen und Objektschlüsselnamen angeben.	28. Mai 2015

Änderung	Beschreibung	Datum
In Lambda wurde die allgemeine Unterstützung von mobilen Backends hinzugefügt.	Lambda ist jetzt allgemein für den produktiven Einsatz verfügbar. In dieser Version werden außerdem neue Funktionen eingeführt, mit denen die Erstellung von Backends für Mobiltelefone, Tablets und das Internet of Things (IoT) vereinfacht wird; diese können mit Lambda automatisch skaliert werden, ohne dass eine Infrastruktur bereitgestellt oder verwaltet werden muss. Lambda unterstützt jetzt sowohl synchrone (in Echtzeit) als auch asynchrone Ereignisse. Es sind zusätzliche Funktionen enthalten, u. a. für eine einfachere Konfiguration und Verwaltung von Ereignisquellen. Das Berechtigungsmodell und das Programmiermodell wurden durch die Einführung von Ressourcenrichtlinien für Lambda-Funktionen vereinfacht.	9. April 2015
Vorversion	Vorversion des AWS Lambda -Entwicklerhandbuchs.	13. November 2014

Die vorliegende Übersetzung wurde maschinell erstellt. Im Falle eines Konflikts oder eines Widerspruchs zwischen dieser übersetzten Fassung und der englischen Fassung (einschließlich infolge von Verzögerungen bei der Übersetzung) ist die englische Fassung maßgeblich.