ベストプラクティスガイド

Amazon EKS



Copyright © 2026 2024. Review license at https://github.com/aws/aws-eks-best-practices/blob/master/LICENSE

Amazon EKS: ベストプラクティスガイド

Copyright © 2026 2024. Review license at https://github.com/aws/aws-eks-best-practices/blob/master/LICENSE

Amazon の商標およびトレードドレスは Amazon 以外の製品およびサービスに使用することはできません。また、お客様に誤解を与える可能性がある形式で、または Amazon の信用を損なう形式で使用することもできません。Amazon が所有していない他のすべての商標は、それぞれの所有者の所有物であり、Amazon に関連、関連、またはスポンサーされている場合があります。

Table of Contents

序章 1
関連ガイド 1
貢献 2
セキュリティ 3
このガイドの使い方 3
の責任共有モデルを理解する 3
序章 1
フィードバック 5
詳細情報 6
ツールとリソース 6
EKS 自動モード - セキュリティ 6
セキュリティアーキテクチャ 7
よくある質問 (FAQ) 17
Identity and Access Management 19
EKS クラスターへのアクセスの制御 20
クラスターアクセスに関する推奨事項 27
EKS ポッドの ID と認証情報 33
EKS ポッドの ID と認証情報に関する推奨事項 42
ツールとリソース 51
ポッドのセキュリティ 51
Linux の機能 52
ポッドセキュリティソリューション 53
推奨事項 62
ツールとリソース 51
マルチテナンシー 69
ソフトマルチテナンシー 69
Kubernetes コンストラクト 72
コントロールの軽減 74
ハードマルチテナンシー 81
今後の方向性 82
マルチクラスター管理ツールとリソース 82
検出コントロール 82
推奨事項 62
ツールとリソース 51

ネットワークセキュリティ 90
トラフィックコントロール 91
ネットワーク暗号化 91
ネットワークポリシー 91
推奨事項 62
セキュリティグループ 96
Pod のネットワークポリシーとセキュリティグループを使用するタイミング 99
Service Mesh ポリシーの強制または Kubernetes ネットワークポリシー 100
ThirdPartyのネットワークポリシーエンジン 101
転送中の暗号化 103
ツールとリソース 51
データ暗号化とシークレット管理 113
保管中の暗号化 113
シークレットの管理 115
ツールとリソース 51
ランタイムセキュリティ 118
セキュリティコンテキストと組み込みの Kubernetes コントロール 118
推奨事項 62
ツールとリソース 122
インフラストラクチャセキュリティ 122
推奨事項 62
代替案 126
ツールとリソース 51
規制コンプライアンス 130
左へのシフト 132
ツールとリソース 51
インシデント対応とフォレンジック 133
インシデント対応計画の例 133
推奨事項 62
ツールとリソース 51
イメージのセキュリティ 138
推奨事項 62
ツールとリソース 51
マルチアカウント戦略 149
マルチテナントクラスターのマルチワークロードアカウント戦略の計画 149
一元化された EKS クラスター 150

分散型 EKS クラスター 156
一元化された EKS クラスターと分散された EKS クラスター 158
クラスターアクセス管理 160
EKS アクセス管理オプション 160
オプション 1: クラスターアクセス管理 (CAM) API を使用する AWS IAM アイデンティティセンター 161
オプション 2: Kubernetes グループにマッピングされた AWS IAM ユーザー/ロール 163
オプション 3: OIDC プロバイダー 164
ワークロードの AWS EKS Pod Identity と IRSA 164
推奨事項 165
クラスターの自動スケーリング 166
EKS 自動モード 166
自動モードを使用する理由 167
よくある質問 167
Karpenter 173
推奨事項 62
Karpenter のベストプラクティス 174
NodePools の作成 178
ポッドのスケジューリング 182
CoreDNS レコメンデーション 185
Karpenter の設計図 186
その他のリソース 90
Cluster Autoscaler 186
概要: 186
パフォーマンスとスケーラビリティの最適化 191
コストと可用性の最適化 193
高度なユースケース 196
追加のパラメータ 199
その他のリソース 90
リファレンス 81
信頼性 203
このガイドの使い方 3
序章 1
フィードバック 5
アプリケーション 206
推奨事項 62

水平ポッドオートスケーラー (HPA) 211
垂直ポッドオートスケーラー (VPA) 212
アプリケーションの更新 213
ヘルスチェックと自己修復 215
推奨事項 215
中断への対応 217
推奨事項 218
オブザーバビリティ 220
推奨事項 220
コントロールプレーン 223
EKS アーキテクチャ 223
推奨事項 224
コントロールプレーンメトリクスのモニタリング 224
クラスター認証 228
アドミッションウェブフック 230
クラスターのアップグレードの処理 231
クラスターエンドポイントの接続 232
大規模なクラスターの実行 233
その他のリソース: 233
データプレーン 233
推奨事項 62
ネットワーク 241
Kubernetes ネットワーキングモデル 241
コンテナネットワークインターフェイス (CNI) 242
Amazon Virtual Private Cloud (VPC) CNI 243
サブネット計算ツール 244
VPC とサブネットに関する考慮事項 245
概要: 186
推奨事項 62
Amazon VPC CNI 256
概要: 186
推奨事項 62
IP アドレス使用率の最適化 269
ノードレベルの IP 消費を最適化する 269
IP の枯渇問題を軽減する 269
IPv6 クラスターの実行 276

概要: ... 186
推奨事項 ... 62
カスタムネットワーキング ... 286
構成例 ... 287
推奨事項 ... 62
Linux のプレフィックスモード ... 294
推奨事項 ... 62
Windows のプレフィックスモード ... 299
推奨事項 ... 62
ポッドあたりのセキュリティグループ ... 305
推奨事項 ... 62
ロードバランシング ... 314
Load Balancerタイプの選択 ... 315
ロードバランサーのプロビジョニング ... 316
Load Balancerのターゲットタイプの選択 ... 317
Load Balancer ヘルスチェックの設定 ... 319
可用性とポッドのライフサイクル ... 320
リファレンス ... 81
付録 ... 325
ネットワークパフォーマンスの問題のモニタリング ... 326
DNS スロットリングの問題に対する CoreDNS トラフィックのモニタリング ... 326
Conntrack メトリクスを使用した DNS クエリの遅延のモニタリング ... 327
その他の重要なネットワークパフォーマンスメトリクス ... 328
ネットワークパフォーマンスの問題がないかワークロードをモニタリングするためのメトリクスのキャプチャ ... 328
IPVS モードで kube-proxy を実行する ... 336
概要: ... 186
スケーラビリティ ... 341
このガイドの使い方 ... 341
スケーリングディメンションについて ... 341
超大規模スケーリング ... 342
コントロールプレーン ... 342
ワークロードとノードのバーストを制限する ... 342
ノードとポッドを安全にスケールダウンする ... 343
Kubectl の実行時にクライアント側のキャッシュを使用する ... 344
kubectl 圧縮を無効にする ... 344

シャードクラスターオートスケーラー 345
API の優先度と公平性 346
API サーバーでのリソースの取得 353
データプレーン 360
自動ノード自動スケーリング 360
さまざまな EC2 インスタンスタイプを使用する 360
API サーバーの負荷を軽減するためにより大きなノードを優先する 361
ワークロードのパフォーマンスを一定にするために同様のノードサイズを使用する 362
コンピューティングリソースを効率的に使用する 363
Amazon マシンイメージ (AMI) の更新を自動化する 363
コンテナに複数の EBS ボリュームを使用する 364
ワークロードが EBS ボリュームを使用している場合、EBS アタッチ制限が低いインスタンスを避ける 366
ディスクへの不要なログ記録を無効にする 366
OS 更新速度が必要なときにインスタンスにパッチを適用する 366
クラスターサービス 367
CoreDNS のスケーリング 368
Kubernetes メトリクスサーバーを垂直方向にスケールする 369
CoreDNS lameduck の期間 369
CoreDNS 準備状況プローブ 370
エージェントのログ記録とモニタリング 370
ワークロード 371
ポッドネットワークに IPv6 を使用する 372
名前空間あたりのサービス数の制限 372
Elastic Load Balancer のクォータを理解する 372
Route 53、Global Accelerator、または CloudFront を使用する 373
エンドポイントの代わりに EndpointSlices を使用する 374
可能であれば、イミュータブルシークレットと外部シークレットを使用する 374
デプロイ履歴の制限 375
enableServiceLinks をデフォルトで無効にする 375
リソースごとに動的アドミッションウェブフックを制限する 375
複数のクラスター間でワークロードを比較する 376
スケーリングの背後にある理論 377
ノードとチャーンレート 377
1 秒あたりのクエリ数で考える 377
分散コンポーネントのスケーリング 378

アップストリームとダウンストリームのボトルネック 379
メトリクスによるスケーリング 380
コントロールプレーンのモニタリング 383
API サーバー 383
問題はどこにありますか? 383
スケジューラー 389
Kube コントローラーマネージャー 390
ETCD 392
ノードの効率とスケーリング 393
ノードの選択 394
ノードビンのパッキング 395
使用率と飽和度 403
CPU 制限の設定 411
Kubernetes SLOs 414
Kubernetes SLOs 415
Kubernetes SLI メトリクス 418
クラスターSLOs 421
既知の制限とService Quotasタ 422
その他の AWS Service Quotas 423
AWS リクエストスロットリング 429
その他の既知の制限 430
クラスターのアップグレード 431
概要: 431
アップグレード前 431
クラスターup-to-dateに保つ 432
EKS リリースカレンダーを確認する 433
責任共有モデルがクラスターのアップグレードにどのように適用されるかを理解する 433
クラスターをインプレースでアップグレードする 433
コントロールプレーンとデータプレーンを順番にアップグレードする 434
EKS ドキュメントを使用してアップグレードチェックリストを作成する 435
Kubernetes API を使用してアドオンとコンポーネントをアップグレードする 435
アップグレード前に基本的な EKS 要件を確認する 437
使用可能な IP アドレスを検証する 437
EKS IAM ロールを検証する 438
EKS アドオンへの移行 439

コントロールプレーンをアップグレードする前に、削除された API の使用状況を特定して修正する 440
クラスターインサイト 440
Kube-no-trouble 442
冥王星 443
リソース 443
Kubernetes ワークロードを更新します。kubectl-convert を使用してマニフェストを更新する 444
データプレーンのアップグレード中にワークロードの可用性を確保するために、PodDisruptionBudgets と topologySpreadConstraints を設定する 445
Managed Node Groups または Karpenter を使用してデータプレーンのアップグレードを簡素化する 446
既存のノードとコントロールプレーンとのバージョン互換性を確認する 447
Karpenter マネージドノードのノード有効期限を有効にする 447
Karpenter マネージドノードにドリフト機能を使用する 448
eksctl を使用してセルフマネージド型ノードグループのアップグレードを自動化する 448
アップグレード前にクラスターをバックアップする 449
コントロールプレーンのアップグレード後に Fargate デプロイを再起動する 449
インプレースクラスターアップグレードの代わりに Blue/Green クラスターを評価する 449
Kubernetes プロジェクトで計画された主要な変更を追跡する — 先を考える 450
機能の削除に関する具体的なガイダンス 451
1.25 での Dockershim の削除 - Docker ソケット (DDS) に Detector を使用する 451
1.25 での PodSecurityPolicy の削除 - Pod セキュリティ標準または policy-as-code ソリューションへの移行 451
1.23 でのツリー内ストレージドライバーの廃止 - Container Storage Interface (CSI) ドライバーへの移行 451
その他のリソース 452
ClowdHaus EKS アップグレードガイダンス 452
GoNoGo 452
コスト最適化 453
一般的なガイドライン 453
EKS コスト最適化のベストプラクティス 453
このガイドの使い方 3
AWS サービスと Kubernetes の主な機能 454
フィードバック 5
フレームワーク 455

「」の柱: 測定と説明責任 455
Save の柱: コスト最適化 456
計画の柱: 計画と予測 457
実行の柱 457
リファレンス 81
認識度 458
推奨事項 62
その他のツール 465
コンピューティング 466
ワークロードの適切なサイズ設定 467
消費量の削減 467
未使用の容量を減らす 468
コンピューティングキャパシティタイプを最適化する 474
コンピューティング使用量の最適化 478
Network 479
ポッド間の通信 480
Load Balancerからポッドへの通信 492
コンテナレジストリからのデータ転送 494
インターネットおよび AWS サービスへのデータ転送 495
VPCs間のデータ転送 498
Service Mesh の使用 501
その他のリソース 90
[Storage (ストレージ)] 509
概要 186
エフェメラルボリューム 509
永続ボリューム 510
その他の考慮事項 517
オブザーバビリティ 518
序章 518
ロギング 518
EKS コントロールプレーン 518
EKS データプレーン 519
メトリクス 523
トレース 527
その他のリソース： 90
Server 530

AMI 管理 530
独自の Amazon EKS 最適化 Windows AMI の管理 531
カスタム EKS 最適化 AMIs の高速起動の設定 532
カスタム AMIs での Windows ベースレイヤーのキャッシュ 533
ブログ記事 534
Windows コンテナ用 gMSA 534
gMSA アカウントとは 534
Windows コンテナと gMSA のユースケース 535
Windows Server の強化 537
Windows Server Core による攻撃対象領域の削減 537
RDP 接続の回避 538
Amazon Inspector 539
Amazon GuardDuty 540
Amazon EC2 for Windows のセキュリティ 540
Windows イメージのスキャン 541
Windows のバージョンとライセンス 541
Windows サーバー バージョン 541
ライセンス 542
ログ記録 542
レコメンデーションのログ記録 542
Windows コンテナのモニタリング 543
Windows ネットワーク 548
Windows コンテナネットワークの概要 548
IP アドレス管理 549
コンテナネットワークインターフェイス (CNI) オプション 552
ネットワークポリシー 552
メモリとシステム管理 553
システムと kubelet メモリの予約 553
Windows コンテナのメモリ要件 554
結論 554
インフラストラクチャ管理 554
Windows イメージのプッシュとプル 555
リファレンス 557
スケジューリング 557
ノードへの PODs の割り当てのベストプラクティス 557
OS 固有のワークロードを適切なコンテナホストに配置する 558

同じクラスターでの複数の Windows ビルドの処理 559
RuntimeClass を使用して Pod マニフェストの NodeSelector と許容範囲を簡素化する 560
マネージド型ノードグループのサポート 561
その他のドキュメント 562
Windows コンテナのポッドセキュリティ 562
ストレージオプション 565
ツリー内ボリュームプラグインとout-of-treeボリュームプラグインとは 565
Windows 用ツリー内ボリュームプラグイン 565
Windows Out-of-tree 569
Amazon FSx for Windows File Server 570
Windows コンテナイメージの強化 571
1. ローカルセキュリティポリシーとレジストリを使用してアカウントポリシー (パスワードまたはロックアウト) を設定する 572
2. 監査ポリシー 574
3. Windows コンテナの IIS セキュリティのベストプラクティス 577
4. 最小権限の原則 582
最終的な考え方: Windows コンテナの保護が今日の脅威の状況で必須である理由 584
ハイブリッド 586
ネットワークの切断 586
ベストプラクティス 587
Kubernetes ポッドのフェイルオーバー 592
アプリケーションネットワークトラフィック 598
ホスト認証情報 603
AI/ML 606
フィードバック 606
コンピューティング 606
GPU リソースの最適化とコスト管理 606
ノードの耐障害性とトレーニングジョブの管理 620
アプリケーションのスケーリングとパフォーマンス 623
高度な GPU 管理のための動的リソース割り当て 625
ネットワーク 671
ノード間通信が高いアプリケーションには、より高いネットワーク帯域幅または Elastic Fabric Adapter を検討する 671
ポッドの起動時間を短縮するために使用可能な IP アドレスの数を増やす 672
セキュリティ 673
セキュリティおよびコンプライアンス 673

[Storage (ストレージ)] 674
データ管理とストレージ 674
オブザーバビリティ 687
モニタリングとオブザーバビリティ 687
オブザーバビリティとメトリクス 688
パフォーマンス 704
ML アーティファクト、サービングフレームワーク、スタートアップ最適化の管理 704
コンテナイメージのプル時間の最適化 706
kubelet ストレージと containerd ストレージの NVMe を検討する 709
寄稿 710
既存の寄稿者の概要 710
ローカル編集環境のセットアップ 710
リポジトリをフォークしてクローンする 710
VS Code Workspace を開く 711
ファイルの編集 711
プルリクエストを送信する 711
github.dev ウェブベースのエディタを使用する 712
1 ページの編集 712
ページの ID を表示および設定する 712
ページ ID を設定する 713
新しいページを作成する 713
ページメタデータを作成する 713
目次に追加する 714
イメージを挿入する 714
Vale でスタイルを確認する 715
ローカルプレビューを構築する 715
AsciiDoc チートシート 715
基本的なフォーマット 715
ヘッダー 716
Lists 716
Links 716
イメージ 716
コードブロック 717
テーブル 717
警告 717
含まれるもの 718

..... dccxix

Amazon EKS ベストプラクティスガイド

EKS ベストプラクティスガイドへようこそ。このプロジェクトの主な目標は、Amazon EKS の 2 日目のオペレーションに関する一連のベストプラクティスを提供することです。このガイダンスを GitHub に公開することを選択しました。これにより、迅速に反復し、さまざまな懸念に対してタイムリーで効果的なレコメンデーションを提供し、より広範なコミュニティからの提案を簡単に取り入れることができます。

現在、以下のトピックに関するガイドを公開しています。

- セキュリティのベストプラクティス
- 信頼性のベストプラクティス
- クラスターの自動スケーリングのベストプラクティス: Karpenter
- クラスターの自動スケーリングのベストプラクティス: cluster-autoscaler
- クラスターの自動スケーリングのベストプラクティス: EKS Auto Mode
- ネットワーキングのベストプラクティス
- スケーラビリティのベストプラクティス
- クラスターのアップグレードのベストプラクティス
- コスト最適化のベストプラクティス
- Windows コンテナを実行するためのベストプラクティス
- ハイブリッドデプロイのベストプラクティス
- AI/ML ワークロードを実行するためのベストプラクティス

また、ハードエンクと呼ばれる Python ベースの CLI (コマンドラインインターフェイス) をオープンソース化し、このガイドの推奨事項の一部を確認しました。

今後は、パフォーマンス、コスト最適化、運用上の優秀性に関するベストプラクティスガイダンスを公開する予定です。

関連ガイド

EKS ユーザーガイドに加えて、AWS は EKS の実装に役立つ他のいくつかのガイドを公開しています。

- EMR コンテナのベストプラクティスガイド

- EKS 上のデータ
- AWS オブザーバビリティのベストプラクティス
- Amazon EKS Blueprints for Terraform
- Amazon EKS 設計図クイックスタート

貢献

これらのガイドに寄稿することをお勧めします。有効であることが証明されているプラクティスを実装している場合は、問題またはプルリクエストを開いて共有してください。同様に、すでに公開したガイダンスでエラーや欠陥を発見した場合は、PR を送信して修正してください。PRs を送信するためのガイドラインは、寄稿ガイドラインに記載されています。

セキュリティのベストプラクティス

このガイドでは、リスク評価と緩和戦略を通じてビジネス価値を提供しながら、EKS に依存する情報、システム、アセットの保護に関するアドバイスを提供します。本書のガイダンスは、お客様がベストプラクティスに従って EKS を実装できるように AWS が公開している一連のベストプラクティスガイドの一部です。パフォーマンス、運用上の優秀性、コスト最適化、信頼性に関するガイドは、今後数か月で利用可能になります。

このガイドの使い方

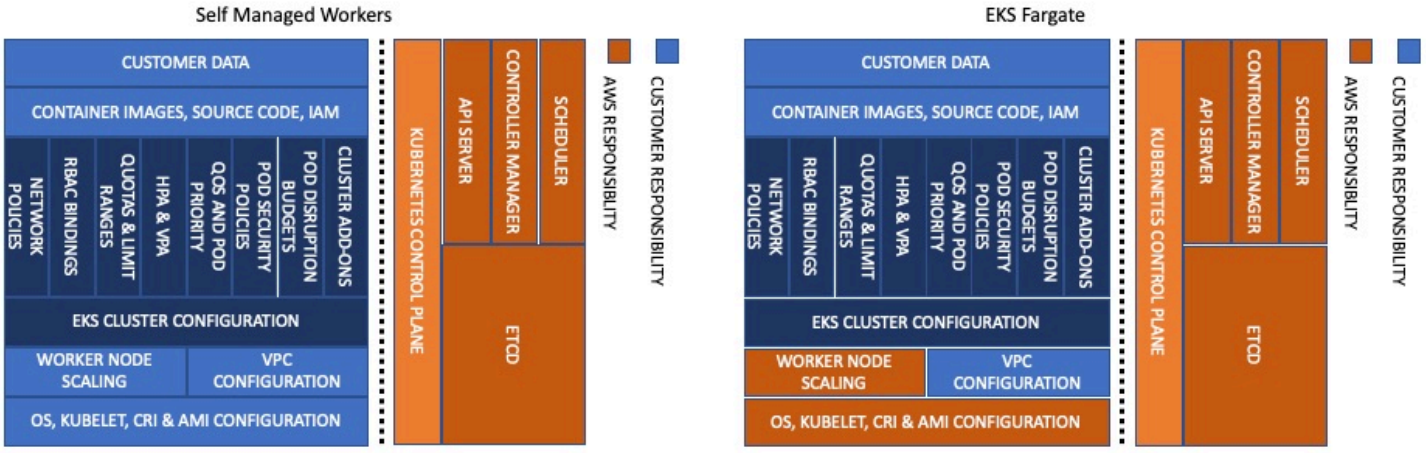
このガイドは、EKS クラスターとサポートするワークロードのセキュリティコントロールの有効性の実装とモニタリングを担当するセキュリティ実務者を対象としています。このガイドは、使いやすいようにさまざまなトピック領域にまとめられています。各トピックは簡単な概要から始まり、EKS クラスターを保護するための推奨事項とベストプラクティスのリストが続きます。トピックは特定の順序で読み取る必要はありません。

の責任共有モデルを理解する

EKS などのマネージドサービスを使用する場合、セキュリティとコンプライアンスは責任共有と見なされます。一般的に、AWS はクラウドのセキュリティ「of」を担当し、お客様はクラウドのセキュリティ「in」を担当します。EKS では、AWS が EKS マネージド Kubernetes コントロールプレーンの管理を担当します。これには、Kubernetes コントロールプレーンノード、ETCD データベース、AWS が安全で信頼性の高いサービスを提供するために必要なその他のインフラストラクチャが含まれます。EKS のコンシューマーは、 IAM、ポッドセキュリティ、ランタイムセキュリティ、ネットワークセキュリティなど、このガイドのトピックに主に責任を負います。

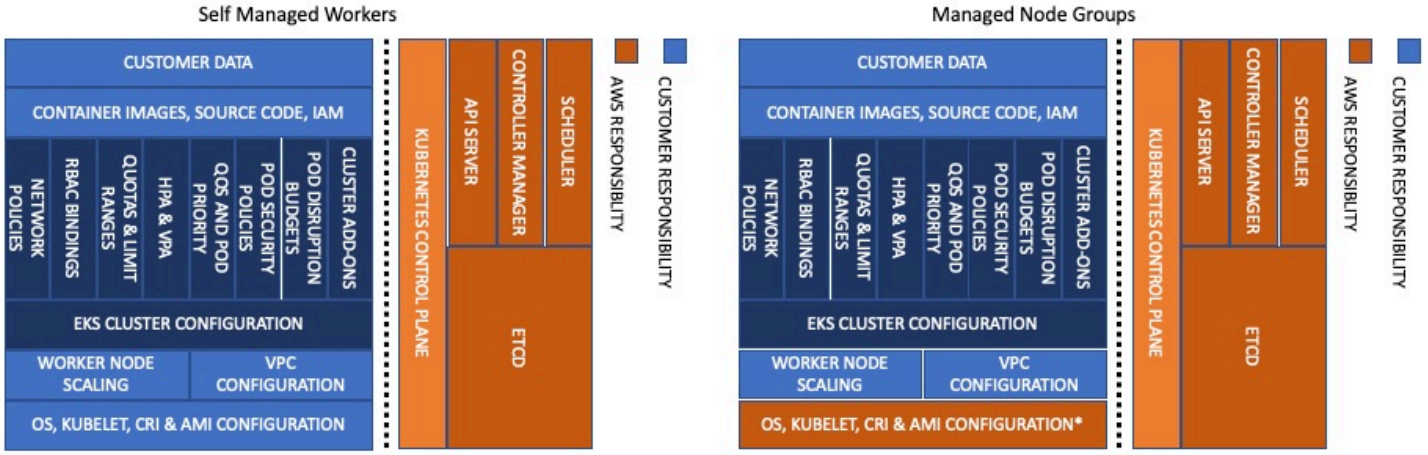
インフラストラクチャのセキュリティに関して、AWS はセルフマネージドワーカーからマネージド型ノードグループ、Fargate に移行する際、追加の責任を負います。例えば、Fargate では、AWS が Pod の実行に使用される基盤となるインスタンス/ランタイムの保護を担当します。

責任共有モデル - Fargate



AWS は、Kubernetes パッチバージョンとセキュリティパッチを使用して EKS 最適化 AMI を最新の状態に保つ責任も負います。Managed Node Groups (MNG) を使用しているお客様は、EKS API、CLI、Cloudformation、または AWS コンソールを介してノードグループを最新の AMI にアップグレードする責任があります。また、Fargate とは異なり、MNGsはインフラストラクチャ/クラスターを自動的にスケールしません。これは、クラスターオートスケーラー、または Karpenter、ネイティブ AWS Auto Scaling、SpotInst の Ocean、Atlassian の Escalator などの他のテクノロジーで処理できます。

責任共有モデル - MNG



システムを設計する前に、責任とサービスのプロバイダー (AWS) の間の境界線がどこにあるかを知ることが重要です。

責任共有モデルの詳細については、https://aws.amazon.com/compliance/shared-responsibility-model/ を参照してください。

序章

EKS などのマネージド Kubernetes サービスを使用する際に関連するセキュリティのベストプラクティス領域がいくつかあります。

- Identity and Access Management
- ポッドのセキュリティ
- ランタイムセキュリティ
- ネットワークセキュリティ
- マルチテナンシー
- マルチテナンシーのマルチアカウント
- 検出コントロール
- インフラストラクチャセキュリティ
- データ暗号化とシークレット管理
- 規制コンプライアンス
- インシデント対応とフォレンジック
- イメージのセキュリティ

システムの設計の一環として、セキュリティへの影響と、セキュリティ体制に影響を与える可能性のあるプラクティスを考慮する必要があります。たとえば、リソースセットに対してアクションを実行できるユーザーを制御する必要があります。また、セキュリティインシデントを迅速に特定し、システムやサービスを不正アクセスから保護し、データ保護を通じてデータの機密性と完全性を維持する機能も必要です。セキュリティインシデントに対応するための一連のプロセスを明確に定義してリハーサルすることで、セキュリティ体制も向上します。これらのツールやテクニックは、金銭的な損失の予防や規制遵守という目的を達成するためにも重要です。

AWS は、さまざまなセキュリティ意識の高いお客様からのフィードバックに基づいて進化した豊富なセキュリティサービスを提供することで、組織がセキュリティとコンプライアンスの目標を達成できるよう支援します。非常に安全な基盤を提供することで、お客様は「差別化されていない重労働」に費やす時間を減らし、ビジネス目標の達成により多くの時間を費やすことができます。

フィードバック

このガイドは、より広範な EKS/Kubernetes コミュニティから直接フィードバックや提案を収集するために GitHub でリリースされています。ガイドに含める必要があると思われるベストプラクティス

がある場合は、問題をファイルするか、GitHub リポジトリに PR を送信してください。新しい機能がサービスに追加されるとき、または新しいベストプラクティスが進化したときに、ガイドを定期的に更新することを意図しています。

詳細情報

このホワイトペーパーでは、セキュリティ監査ワーキンググループが主催する Kubernetes セキュリティホワイトペーパーで、セキュリティ実務者が健全な設計と実装の決定を行うのを支援することを目的として、Kubernetes アタックサーフェスとセキュリティアーキテクチャの主要な側面について説明します。

CNCF は、クラウドネイティブセキュリティに関するホワイトペーパーも公開しました。このホワイトペーパーでは、テクノロジー環境がどのように進化したかを調べ、DevOps プロセスとアジャイル手法に沿ったセキュリティプラクティスの導入を提唱しています。

ツールとリソース

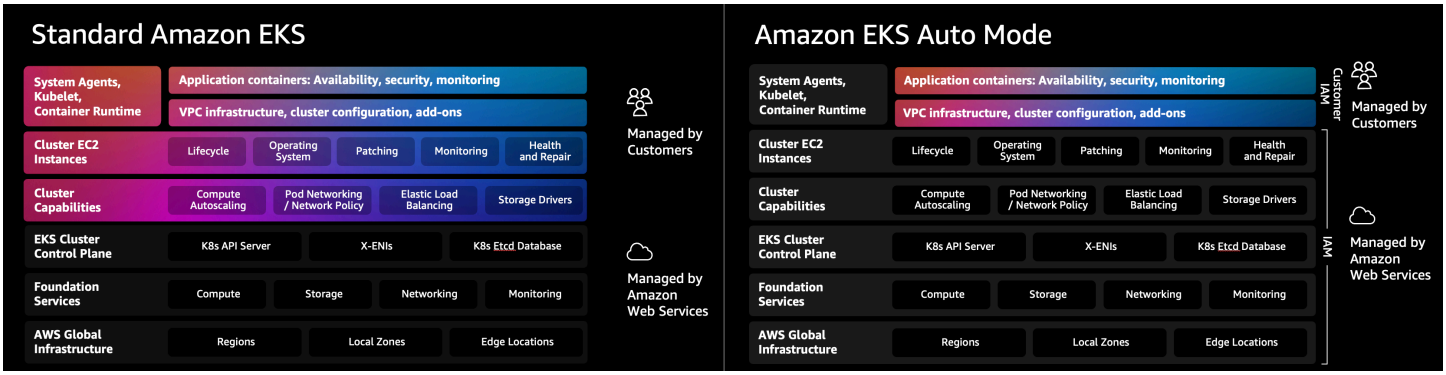
Amazon EKS セキュリティイマージョンワークショップ

EKS 自動モード - セキュリティ

> **Tip**
>
> Amazon EKS Auto Mode の実践的な経験に興味がありますか? AWS Connected Community にサインアップして、AWS の専門家が主催する次回の仮想 Kubernetes ワークショップに参加してください。

Amazon EKS Auto Mode は、AWS のセキュリティ管理をコントロールプレーンを超えて拡張し、ワーカーノードとコアクラスターコンポーネントを含めることで、強化されたセキュリティ機能を導入します。この包括的なセキュリティモデルは、組織が運用上のオーバーヘッドを削減しながら、強力なセキュリティ体制を維持するのに役立ちます。

責任共有モデル - EKS 自動モード



EKS Auto Mode の主なセキュリティ強化点は次のとおりです。

- アタックサーフェスを減らした最小コンテナ最適化 OS
- EC2 マネージドインスタンスを通じてセキュリティのベストプラクティスを強制する
- 必須のノードローテーションによるセキュリティパッチの自動管理
- 組み込みノードの分離とセキュリティの境界
- 最小限のアクセス許可で IAM 統合を合理化

EKS Auto Mode は、これらのセキュリティコントロールをデフォルトで適用するため、組織はセキュリティとコンプライアンスの要件を満たすと同時に、クラスターオペレーションを簡素化できます。このアプローチはdefense-in-depthの原則に沿っており、インフラストラクチャ、ノード、ワークロードレベルで複数のセキュリティコントロールレイヤーを提供します。

> **Important**
>
> EKS Auto Mode は強化されたセキュリティ機能を提供しますが、組織は引き続きアプリケーションレイヤーに適切なセキュリティコントロールを実装し、クラスターで実行されているワークロードのセキュリティのベストプラクティスに従う必要があります。

セキュリティアーキテクチャ

EKS Auto Mode は、コントロールプレーンから個々のノードまで、EKS インフラストラクチャの複数のレイヤーにセキュリティコントロールを実装します。このアーキテクチャを理解することは、EKS クラスターを効果的に管理および保護するために不可欠です。

コントロールプレーンのセキュリティ

EKS Auto Mode の EKS コントロールプレーンは、新しいセキュリティ機能を追加しながら、従来の EKS クラスターと同じ高いセキュリティ標準を維持します。

- エンベロープ暗号化: すべての Kubernetes API データはエンベロープ暗号化を使用して自動的に暗号化されます。
- KMS 統合: Kubernetes KMS プロバイダー v2 で AWS KMS を使用し、AWS 所有のキーまたはカスタマーマネージドキー (CMK) のオプションを使用します。
- 拡張コンポーネント管理: 自動スケーリング、ENI 管理、EBS コントローラーなどの重要なコンポーネントはクラスター外に移動され、AWS によって管理されます。
- セキュリティコントロールと監査機能の向上: EKS Auto Mode に必要なアクセス許可は、標準の EKS クラスターを超えて、個々のノードロールではなくクラスター IAM ロールを通じて完全に管理されます。

IAM 統合とアクセス管理

EKS Auto Mode は、EKS アクセスエントリと EKS Pod Identity を通じて AWS Identity and Access Management (IAM) との統合を強化します。

クラスターアクセス管理

EKS Auto Mode では、Cluster Access Management (CAM) API によるクラスターアクセス管理の改善が導入されています。

- による標準化された認証モード `EKS_API`
- API ベースのアクセス管理によるセキュリティの強化
- アクセスエントリとアクセスポリシーを使用したアクセスコントロールの簡素化

アクセスエントリを作成して、クラスターアクセスを管理できます。

```
aws eks create-access-entry \
    --cluster-name ${EKS_CLUSTER_NAME} \
    --principal-arn arn:aws:iam::${ACCOUNT_ID}:role/${IAM_ROLE_NAME} \
    --type STANDARD
```

> **Important**
>
> CONFIG_MAP_AND_API 認証モードで EKS Auto Mode クラスターを作成することは可能ですが、これは標準アプローチではなく、新しいクラスターにはデフォルトのAPI認証モードを使用することを強くお勧めします。 APIベースの認証は、従来の ConfigMap ベースのアプローチと比較して、セキュリティを強化し、アクセス管理を簡素化します。

EKS Pod Identity

EKS Auto Mode には Pod Identity Agent が既にデプロイされているため、ポッドに AWS IAM アクセス許可を付与する方法が合理化されています。

- OIDC プロバイダー設定なしで IAM アクセス許可管理を簡素化
- IRSA と比較した運用オーバーヘッドの削減
- セッションタグ付けと ABAC サポートによるセキュリティの強化

```
aws eks create-pod-identity-association \
  --cluster-name ${EKS_CLUSTER_NAME} \
  --role-arn arn:aws:iam::${AWS_ACCOUNT_ID}:role/${IAM_ROLE_NAME} \
  --namespace ${NAMESPACE} \
  --service-account ${SERVICE_ACCOUNT_NAME}
```

> **Important**
>
> Pod Identity は、IRSA と比較してセキュリティ機能を強化し、管理を簡素化するため、EKS Auto Mode の IAM アクセス許可に推奨されるアプローチです。

ノード IAM ロール

EKS Auto Mode は、EKS Auto Mode ノードが動作するために必要なアクセス許可のみAmazonEKSWorkerNodeMinimalPolicyを提供する新しい を使用します。これらのアクセス許可:

- 従来のノードポリシーと比較して、一連のアクセス許可を削減
- 最小特権の原則に従う

- AWS によって自動的に管理および更新されます

この最小限のポリシーアプローチは、ノードとそのワークロードで使用できるアクセス許可を制限することで、セキュリティ体制を改善するのに役立ちます。

ノードセキュリティ

EKS Auto Mode では、ノードレベルでいくつかの大幅なセキュリティ強化が導入されています。

EC2 マネージドインスタンスのセキュリティ

EKS Auto Mode ノードは、セキュリティプロパティが強化された Amazon EC2 マネージドインスタンスを使用します。

- AWS のノード運用能力を損なう可能性のあるオペレーションを妨げる IAM による制限
- 設定変更でノードの交換が必要なイミュータブルなインフラストラクチャパターン
- 定期的なセキュリティ更新を確保するために 21 日以内にノードを置き換える必要がある
- IMDSv2 を使用したインスタンスメタデータへの制限付きアクセスと制御されたホップ制限

オペレーティングシステムのセキュリティ

オペレーティングシステムは Bottlerocket のカスタムバリアントで、EKS Auto Mode 用に最適化されており、以下が含まれています。

- 読み取り専用ルートファイルシステム
- 必須のアクセスコントロールでデフォルトで有効になっている SELinux
- 一意の SELinux MCS ラベルを使用した自動ポッド分離
- 不要なサービスの SSH アクセスと削除を無効にしました
- ノードローテーションによる自動セキュリティパッチ

ノードコンポーネントのセキュリティ

ノードコンポーネントは、セキュリティのベストプラクティスを使用して設定されます。

- 安全なデフォルトで設定された Kubelet
- コンテナランタイム強化設定
- 証明書の自動管理とローテーション

- node-to-control-plane間の制限付き通信

ネットワークセキュリティ

EKS Auto Mode は、クラスター内および外部リソースとの安全な通信を確保するために、いくつかのネットワークセキュリティ機能を実装します。

VPC CNI ネットワークポリシー

EKS Auto Mode は、Amazon VPC CNI プラグインのネイティブ Kubernetes ネットワークポリシーサポートを活用します。

- アップストリームの Kubernetes ネットワークポリシー API と統合
- pod-to-pod通信をきめ細かく制御できます
- イングレスルールとエグレスルールの両方をサポート

EKS Auto Mode でネットワークポリシーのサポートを有効にするには、マconfigMap二フェストを使用して VPC CNI アドオンを設定する必要があります。以下がその例です。

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: amazon-vpc-cni
  namespace: kube-system
data:
  enable-network-policy: "true"
```

また、次に示すように、ネットワークポリシーのサポートがノードクラスで設定されていることを定義する必要があります。

```
apiVersion: eks.amazonaws.com/v1
kind: NodeClass
metadata:
  name: example-node-class
spec:
  networkPolicy: DefaultAllow
  networkPolicyEventLogs: Enabled
```

有効にすると、ネットワークポリシーを作成してトラフィックを制御できます。

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: default-deny
spec:
  podSelector: {}
  policyTypes:
  - Ingress
  - Egress
```

拡張 ENI 管理

EKS Auto Mode は、Elastic Network Interface (ENI) 管理のセキュリティを向上させます。

- AWS 管理の ENI アタッチメントと設定
- データトラフィックからのコントロールトラフィックの分離
- ノードに必要な権限を減らした自動 IP アドレス管理

ストレージセキュリティ

EKS Auto Mode は、エフェメラルストレージと永続ストレージの両方に強化されたセキュリティ機能を提供します。

エフェメラルストレージ

- エフェメラルボリュームに書き込まれたすべてのデータは自動的に暗号化されます
- 業界標準の AES-256 暗号化アルゴリズムを使用
- サービスによってシームレスに処理される暗号化と復号

EBS ボリューム

- ルートとデータ EBS ボリュームは常に暗号化されます
- ボリュームは、インスタンスの終了時に削除されるように設定されます。
- 暗号化にカスタム KMS キーを指定するオプションがあります

EFS 統合

- EFS を使用した転送中の暗号化のサポート

- EFS ファイルシステムの保管時の自動暗号化
- EFS アクセスポイントとの統合によるアクセス制御の強化

> **Important**
>
> EKS Auto Mode で EFS を使用する場合は、EFS ファイルシステムレベルで適切な暗号化設定が設定されていることを確認してください。EFS Auto Mode は EFS 暗号化を直接管理しないためです。

モニタリングとロギング

EKS Auto Mode は、モニタリングとログ記録の機能を強化し、クラスターのセキュリティ体制と運用状態を可視化するのに役立ちます。

コントロールプレーンのログ記録

EKS Auto Mode は、標準の EKS と同じコントロールプレーンログ記録機能を維持しますが、デフォルトですべてのログを有効にしてモニタリングを強化します。

- ログは Amazon CloudWatch Logs に送信されます
- デフォルトでは、EKS Auto Mode は API サーバー、監査、認証、コントローラーマネージャー、スケジューラのすべてのコントロールプレーンログを有効にします。
- EKS Auto Mode により、クラスターオペレーションとセキュリティイベントを詳細に可視化

> **Important**
>
> コントロールプレーンのログ記録では、CloudWatch のログストレージに追加のコストが発生します。セキュリティニーズとコスト管理のバランスを取るために、ロギング戦略を慎重に検討してください。

ノードレベルのログ記録

EKS Auto Mode はノードレベルのログ記録を強化します。

- システムログは自動的に収集され、CloudWatch Logs 経由でアクセスできます
- ノードログはノード終了後も保持され、インシデント後の分析に役立ちます

- ノードレベルのセキュリティイベントと運用上の問題の可視性を強化

Amazon GuardDuty の統合

EKS Auto Mode クラスターは、脅威検出を強化するために Amazon GuardDuty とシームレスに統合されます。次の機能があります。

- コントロールプレーン監査ログの自動スキャン
- ワークロードモニタリングで有効にできるランタイムモニタリング
- 既存の GuardDuty の検出結果とアラートメカニズムとの統合

Amazon GuardDuty for Kubernetes 監査ログで EKS Auto Mode 保護を有効にするには、次のコマンドを実行します。

```
aws guardduty update-detector \
    --detector-id 12abc34d567e8fa901bc2d34e56789f0 \
    --data-sources '{"Kubernetes":{"AuditLogs":{"Enable":true}}}'
```

ランタイムセキュリティのための Amazon GuardDuty 統合

Amazon GuardDuty は、EKS Auto Mode クラスターに不可欠なランタイムセキュリティモニタリングを提供し、包括的な脅威検出とセキュリティモニタリング機能を提供します。この統合は、潜在的なセキュリティ脅威と悪意のあるアクティビティをリアルタイムで特定するのに役立つため、特に重要です。

EKS Auto Mode の主な GuardDuty 機能

- ランタイムモニタリング:
  - ランタイム動作の継続的なモニタリング
  - 悪意のあるアクティビティまたは疑わしいアクティビティの検出
  - 潜在的なコンテナエスケープ試行の特定
  - 異常なプロセス実行またはネットワーク接続のモニタリング
- Kubernetes 固有の脅威検出:
  - 疑わしいポッドデプロイの試行の特定
  - 侵害されたコンテナの検出
  - 特権コンテナの起動のモニタリング

  - 疑わしいサービスアカウントの使用状況の特定
- 包括的な検出結果タイプ:
  - ポリシー:Kubernetes/* - セキュリティのベストプラクティスの違反を検出
  - Impact:Kubernetes/* - 影響を受ける可能性のあるリソースを特定します
  - Discovery:Kubernetes/* - 偵察アクティビティを検出します
  - Execution:Kubernetes/* - 疑わしい実行パターンを識別します
  - 永続性:Kubernetes/* - 潜在的な永続的な脅威を検出します

Amazon GuardDuty for Kubernetes 監査ログとランタイムモニタリングで EKS Auto Mode 保護を有効にするには、次のコマンドを実行します。

```
aws guardduty update-detector \
    --detector-id 12abc34d567e8fa901bc2d34e56789f0 \
    --data-sources '{
        "Kubernetes": {
            "AuditLogs": {"Enable": true},
            "RuntimeMonitoring": {"Enable": true}
        }
    }'
```

> **Important**
>
> GuardDuty Runtime Monitoring は EKS Auto Mode クラスターで自動的にサポートされるため、ノードレベルで追加の設定を行うことなく、セキュリティの可視性が向上します。

GuardDuty の結果の統合

GuardDuty の検出結果は、自動応答のために他の AWS サービスと統合できます。

- EventBridge ルール:

```
{
  "source": ["aws.guardduty"],
  "detail-type": ["GuardDuty Finding"],
  "detail": {
    "type": ["Runtime:Container/*", "Runtime:Kubernetes/*"],
```

```
    "severity": [4, 5, 6, 7, 8]
  }
}
```

- Security Hub の統合:

```
# Enable Security Hub integration
aws securityhub enable-security-hub \
    --enable-default-standards \
    --tags '{"Environment":"Production"}' \
    --region us-west-2
```

EKS Auto Mode を使用した GuardDuty のベストプラクティス

1. すべての検出結果タイプを有効にする:
   - Kubernetes 監査ログモニタリングとランタイムモニタリングの両方を有効にする
   - すべての重要度レベルの検出結果を設定する
2. 自動レスポンスの実装:
   - 重要度の高い検出結果の EventBridge ルールを作成する
   - AWS Security Hub との統合による一元的なセキュリティ管理
   - 必要に応じて自動修復アクションを設定する
3. 定期的なレビューとチューニング:
   - GuardDuty の検出結果を定期的に確認する
   - 環境に基づいて検出しきい値を調整する
   - 新しい検出結果タイプに基づいてレスポンス手順を更新する
4. クロスアカウント管理:
   - 一元管理に GuardDuty 管理者アカウントを使用することを検討する
   - 複数のアカウントで結果の集約を有効にする

> **Warning**
>
> GuardDuty は包括的なセキュリティモニタリングを提供しますが、ネットワークポリシー、ポッドセキュリティ標準、適切な RBAC 設定などの他のセキュリティコントロールを含む defense-in-depth戦略の一部である必要があります。

よくある質問 (FAQ)

Q: EKS Auto Mode は、セキュリティの点で標準の EKS とどのように異なりますか? A: EKS Auto Mode は、EC2 マネージドインスタンス、自動パッチ適用、必須ノードローテーション、組み込みセキュリティコントロールを通じてセキュリティを強化します。AWS がより多くのセキュリティ面を管理できるようにすることで、強力なセキュリティ体制を維持しながら運用上のオーバーヘッドを削減します。

Q: EKS Auto Mode で既存のセキュリティツールとポリシーを引き続き使用できますか? A: はい、EKS Auto Mode はほとんどの既存のセキュリティツールやポリシーと互換性があります。ただし、一部のノードレベルのセキュリティツールでは、EKS Auto Mode ノードのマネージド型の性質上、適応が必要になる場合があります。

Q: セキュリティエージェントとモニタリングツールを EKS Auto Mode にデプロイするにはどうすればよいですか? A: EKS Auto Mode では、セキュリティエージェントとモニタリングツールは、ノード OS に直接インストールするのではなく、Kubernetes ワークロード (通常は DaemonSets。デフォルトではすべてのノードに Pod の 1 つのインスタンスをデプロイします) としてデプロイする必要があります。このアプローチは、EKS Auto Mode のイミュータブルインフラストラクチャモデルと一致しています。例:

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: security-agent
  namespace: security
spec:
  selector:
    matchLabels:
      app: security-agent
  template:
    metadata:
      labels:
        app: security-agent
    spec:
      containers:
      - name: security-agent
        image: security-vendor/agent:latest
        securityContext:
          privileged: false
          # Use specific capabilities instead of privileged mode
          capabilities:
```

```
            add: ["NET_ADMIN", "SYS_ADMIN"]
```

Q: サードパーティーのセキュリティソリューションは EKS Auto Mode と互換性がありますか? A: EKS Auto Mode をサポートするために多くの一般的なサードパーティーのセキュリティソリューションが更新されましたが、EKS Auto Mode のサポートには更新バージョンまたは特定のデプロイ設定が必要になる可能性があるため、セキュリティベンダーに特定のバージョンとデプロイ要件を確認することが常に推奨されます。

Q: EKS Auto Mode のセキュリティエージェントにはどのような制限がありますか? A: 主な制限事項は次のとおりです。

- ノードのオペレーティングシステムを変更するための直接アクセスがない
- ノードローテーション間で永続性がない
- コンテナベースのデプロイと互換性がある必要があります
- ノードのイミュータビリティを尊重する必要がある
- 異なる権限設定が必要になる場合があります
- ノードへの永続的な変更は、 `NodePools`および `NodeClasses`リソースを通じて行う必要があります。

> **Note**
>
> EKS Auto Mode では、セキュリティツールのデプロイ戦略の調整が必要になる場合がありますが、これらの変更により、クラウドネイティブのベストプラクティスに沿った、より保守的で安全な設定が得られることがよくあります。EKS Auto Mode は、管理する機能のほとんどを完全に引き継ぐことを期待しています。したがって、これらの機能に加えた手動の変更は、アクセス可能な場合、EKS Auto Mode によって上書きまたは破棄される可能性があります。

Q: EKS Auto Mode でカスタム AMIs を使用できますか? A: 現時点では、EKS Auto Mode はカスタム AMIs をサポートしていません。これは、AWS が責任共有モデルの一部としてノードのセキュリティ、パッチ適用、メンテナンスを管理するため、設計上行われます。EKS Auto Mode ノードは、AWS によって最適化および保守されている Bottlerocket の特殊なバリアントを使用します。

Q: EKS Auto Mode でノードが自動的にローテーションされる頻度はどのくらいですか? A: EKS Auto Mode のノードの最大有効期間は 21 日です。この制限の前に自動的に置き換えられるため、定期的なセキュリティ更新とパッチ適用が保証されます。

Q: トラブルシューティングのために EKS Auto Mode ノードに SSH 接続できますか? A: いいえ。直接 SSH アクセスは EKS Auto Mode では使用できません。代わりに、NodeDiagnostic Custom Resource Definition (CRD) を使用してシステムログを収集し、情報をデバッグできます。

Q: ネットワークポリシーのサポートは EKS Auto Mode でデフォルトで有効になっていますか? A: 現時点では、ネットワークポリシーのサポートは VPC CNI アドオン設定を使用して明示的に有効にする必要があります。有効にすると、標準の Kubernetes ネットワークポリシーを使用できます。

Q: EKS Auto Mode で IRSA または Pod Identity を使用する必要がありますか? A: どちらもサポートされていますが、EKS Auto Mode では Pod Identity が推奨されるアプローチです。Pod Identity Security エージェントアドオンが既に含まれており、セキュリティ機能の強化と管理の簡素化が図られています。

Q: EKS Auto Mode で aws-auth ConfigMap を引き続き使用できますか? A: `aws-auth` ConfigMap は非推奨の機能です。セキュリティを強化し、アクセス管理を簡素化するために、API ベースの認証のデフォルトのアプローチを使用することをお勧めします。

Q: EKS Auto Mode でセキュリティイベントをモニタリングするにはどうすればよいですか? A: EKS Auto Mode は、GuardDuty、CloudWatch、CloudTrail などの複数のモニタリングソリューションと統合されています。GuardDuty は、EKS ワークロード専用の拡張ランタイムセキュリティモニタリングを提供します。

Q: EKS Auto Mode ノードからログを収集するにはどうすればよいですか? A: S3 バケットにログを自動的にアップロードする NodeDiagnostic CRD を使用します。CloudWatch Container Insights と AWS Distro for OpenTelemetry を使用することもできます。

> **Note**
>
> このよくある質問セクションは、新機能が EKS Auto Mode に追加され、お客様からよくある質問が寄せられると、定期的に更新されます。

Identity and Access Management

Identity and Access Management (IAM) は、認証と認可の 2 つの重要な機能を実行する AWS のサービスです。認証には ID の検証が含まれますが、認可は AWS リソースが実行できるアクションを管理します。AWS 内では、リソースは EC2 などの別の AWS サービスでも、IAM ユーザーやロールなどの AWS プリンシパルでもかまいません。リソースが実行できるアクションを管理するルールは、IAM ポリシーとして表されます。

EKS クラスターへのアクセスの制御

Kubernetes プロジェクトは、ベアラートークン、X.509 証明書、OIDC など、kube-apiserver サービスへのリクエストを認証するためのさまざまな戦略をサポートしています。EKS は現在、ウェブフックトークン認証、サービスアカウントトークン、および 2021 年 2 月 21 日現在の OIDC 認証をネイティブにサポートしています。

ウェブフック認証戦略は、ベアラートークンを検証するウェブフックを呼び出します。EKS では、これらのベアラートークンは、kubectlコマンドの実行時に AWS CLI または aws-iam-authenticator クライアントによって生成されます。コマンドを実行すると、トークンは kube-apiserver に渡され、認証ウェブフックに転送されます。リクエストの形式が正しい場合、ウェブフックはトークンの本文に埋め込まれた署名付き URL を呼び出します。この URL はリクエストの署名を検証し、ユーザーのアカウント、Arn、UserId などのユーザーに関する情報を kube-apiserver に返します。

認証トークンを手動で生成するには、ターミナルウィンドウで次のコマンドを入力します。

```
aws eks get-token --cluster-name <cluster_name> --region <region>
```

出力は次のようになります。

```
{
    "kind": "ExecCredential",
    "apiVersion": "client.authentication.k8s.io/v1alpha1",
    "spec": {},
    "status": {
        "expirationTimestamp": "2024-12-20T17:38:48Z",
        "token": "k8s-aws-
v1.aHR0cHM6Ly9zdHMudXMtd2VzdC0yLmFtYXpvbmF3cy5jb20vP0FjdGlvbj1HZ...."
    }
}
```

プログラムでトークンを取得することもできます。Go で書かれた例を次に示します。

```
package main

import (
  "fmt"
  "log"
  "sigs.k8s.io/aws-iam-authenticator/pkg/token"
)
```

```
func main()  {
  g, _ := token.NewGenerator(false, false)
  tk, err := g.Get("<cluster_name>")
  if err != nil {
    log.Fatal(err)
  }
  fmt.Println(tk)
}
```

出力は次のようになります。

```
{
  "kind": "ExecCredential",
  "apiVersion": "client.authentication.k8s.io/v1alpha1",
  "spec": {},
  "status": {
    "expirationTimestamp": "2020-02-19T16:08:27Z",
    "token": "k8s-aws-
v1.aHR0cHM6Ly9zdHMuYW1hem9uYXdzLmNvbS8_QWN0aW9uPUdldENhbGxlcklkZW50aXR5JlZlcnNpb249MjAxMS0wNi0x
  }
}
```

各トークンは で始まり、base64 でエンコードされた文字列k8s-aws-v1.が続きます。文字列はデコードされると、次のようなようになります。

```
https://sts.amazonaws.com/?Action=GetCallerIdentity&Version=2011-06-15&X-
Amz-Algorithm=AWS4-HMAC-SHA256&X-Amz-Credential=XXXXJPFRILKNSRC2W5QA
%2F20200219%2Fus-xxxx-1%2Fsts%2Faws4_request&X-Amz-Date=20200219T155427Z&X-
Amz-Expires=60&X-Amz-SignedHeaders=host%3Bx-k8s-aws-id&X-Amz-
Signature=XXXf8f3285e320ddb5e683a5c9a405301ad76546f24f28111fdad09cf648a393
```

トークンは、Amazon 認証情報と署名を含む署名付き URL で構成されます。詳細については、https://docs.aws.amazon.com/STS/latest/APIReference/API_GetCallerIdentity.html を参照してください。

トークンの有効期間 (TTL) は 15 分です。その後、新しいトークンを生成する必要があります。これはkubectl、 などのクライアントを使用する場合に自動的に処理されますが、Kubernetes ダッシュボードを使用している場合は、トークンの有効期限が切れるたびに新しいトークンを生成し、再認証する必要があります。

ユーザーの ID が AWS IAM サービスによって認証されると、kube-apiserver はkube-system名前空間の aws-auth ConfigMap を読み取り、ユーザーに関連付ける RBAC グループを決定します。ConfigMap aws-auth は、IAM ユーザーとロール、Kubernetes RBAC グループなど、IAM プリンシパル間の静的マッピングを作成するために使用されます。RBAC グループは、Kubernetes RoleBindings または ClusterRoleBindings で参照できます。これらは、Kubernetes リソース (オブジェクト) のコレクションに対して実行できる一連のアクション (動詞) を定義するという点で IAM ロールに似ています。

グローバル STS エンドポイントにリクエストを送信するクライアントをユーザーが識別するのに役立つ CloudWatch クエリ

以下の CloudWatch クエリを実行して、sts エンドポイントを取得します。stsendpoint が「sts.amazonaws.com」と等しい場合、これはグローバル STS エンドポイントです。stsendpoint が「sts.<region>.amazonaws.com」と等しい場合、これはリージョン STS エンドポイントです。

```
fields @timestamp, @message, @logStream, @log,stsendpoint
| filter @logStream like /authenticator/
| filter @message like /stsendpoint/
| sort @timestamp desc
| limit 10000
```

クラスターアクセスマネージャー

Cluster Access Manager は、AWS IAM プリンシパルの Amazon EKS クラスターへのアクセスを管理するために推奨される方法であり、AWS API の機能であり、EKS v1.23 以降のクラスター (新規または既存) のオプトイン機能です。これにより、AWS IAM と Kubernetes RBACs 間の ID マッピングが簡素化され、AWS と Kubernetes APIsなり、アクセス管理のために aws-auth ConfigMap を編集し、運用上のオーバーヘッドを減らし、設定ミスに対処するのに役立ちます。このツールを使用すると、クラスター管理者はクラスターの作成に使用した AWS IAM プリンシパルに自動的に付与されたcluster-adminアクセス許可を取り消したり絞り込んだりすることもできます。

この API は、次の 2 つの概念に基づいています。

- アクセスエントリ: Amazon EKS クラスターへの認証が許可されている AWS IAM プリンシパル (ユーザーまたはロール) に直接リンクされたクラスター ID。
- アクセスポリシー: Amazon EKS クラスターでアクションを実行するためのアクセスエントリの認可を提供する Amazon EKS 固有のポリシーです。

起動時に、Amazon EKS は事前定義された および AWS 管理ポリシーのみをサポートします。アクセスポリシーは IAM エンティティではなく、Amazon EKS によって定義および管理されます。

Cluster Access Manager では、アップストリーム RBAC と、API サーバーリクエストに関する Kubernetes AuthZ の決定に対する許可と合格 (拒否ではない) をサポートするアクセスポリシーを組み合わせることができます。拒否の決定は、アップストリーム RBAC オーソライザーと Amazon EKS オーソライザーの両方がリクエスト評価の結果を判断できない場合に発生します。

この機能を使用すると、Amazon EKS は 3 つの認証モードをサポートしています。

1. CONFIG_MAP configMap aws-auth を排他的に使用し続けるには。
2. API_AND_CONFIG_MAP は、EKS アクセスエントリ APIsと aws-auth configMap の両方から認証された IAM プリンシパルをソースとし、アクセスエントリを優先します。アクセスエントリへの既存のアクセスaws-auth許可の移行に最適です。
3. API は、EKS アクセスエントリ APIs。これは新しい推奨アプローチです。

開始するために、クラスター管理者は Amazon EKS クラスターを作成または更新し、優先認証を API_AND_CONFIG_MAPまたは APIメソッドに設定し、アクセスエントリを定義して、目的の AWS IAM プリンシパルへのアクセスを許可できます。

```
$ aws eks create-cluster \
    --name <CLUSTER_NAME> \
    --role-arn <CLUSTER_ROLE_ARN> \
    --resources-vpc-config
 subnetIds=<value>,endpointPublicAccess=true,endpointPrivateAccess=true \
    --logging '{"clusterLogging":[{"types":
["api","audit","authenticator","controllerManager","scheduler"],"enabled":true}]}' \
    --access-config
 authenticationMode=API_AND_CONFIG_MAP,bootstrapClusterCreatorAdminPermissions=false
```

上記のコマンドは、クラスター作成者の管理者アクセス許可なしで既に Amazon EKS クラスターを作成する例です。

update-cluster-config コマンドを使用して API authenticationMode を有効にするように Amazon EKS クラスター設定を更新できます。 を使用して既存のクラスターでこれを行うには、まず に更新API_AND_CONFIG_MAPしてから に更新CONFIG_MAPする必要がありますAPI。これらのオペレーションを元に戻すことはできません。つまり、 を から APIAPI_AND_CONFIG_MAPまたは に切り替えることもCONFIG_MAP、 から API_AND_CONFIG_MAPに切り替えることもできませんCONFIG_MAP。

```
$ aws eks update-cluster-config \
    --name <CLUSTER_NAME> \
    --access-config authenticationMode=API
```

API は、クラスターへのアクセスを追加および取り消すコマンドをサポートし、指定されたクラスターの既存のアクセスポリシーとアクセスエントリを検証します。デフォルトポリシーは、次のように Kubernetes RBACs と一致するように作成されます。

EKS アクセスポリシー	Kubernetes RBAC
AmazonEKSClusterAdminPolicy	クラスター管理者
AmazonEKSAdminPolicy	admin
AmazonEKSEditPolicy	edit (編集)
AmazonEKSViewPolicy	表示

```
$ aws eks list-access-policies
{
    "accessPolicies": [
        {
            "name": "AmazonEKSAdminPolicy",
            "arn": "arn:aws:eks::aws:cluster-access-policy/AmazonEKSAdminPolicy"
        },
        {
            "name": "AmazonEKSClusterAdminPolicy",
            "arn": "arn:aws:eks::aws:cluster-access-policy/AmazonEKSClusterAdminPolicy"
        },
        {
            "name": "AmazonEKSEditPolicy",
            "arn": "arn:aws:eks::aws:cluster-access-policy/AmazonEKSEditPolicy"
        },
        {
            "name": "AmazonEKSViewPolicy",
            "arn": "arn:aws:eks::aws:cluster-access-policy/AmazonEKSViewPolicy"
        }
    ]
}
```

```
$ aws eks list-access-entries --cluster-name <CLUSTER_NAME>

{
    "accessEntries": []
}
```

クラスター作成者の管理者権限なしでクラスターを作成すると、アクセスエントリは使用できません。これはデフォルトで作成される唯一のエントリです。

aws-auth ConfigMap (廃止)

Kubernetes と AWS 認証を統合する方法の 1 つは、 kube-system 名前空間に存在する aws-auth ConfigMap 経由です。AWS IAM アイデンティティ (ユーザー、グループ、ロール) 認証を Kubernetes ロールベースのアクセスコントロール (RBAC) 認可にマッピングします。ConfigMap aws-auth は、プロビジョニングフェーズ中に Amazon EKS クラスターに自動的に作成されます。最初はノードがクラスターに参加できるように作成されましたが、前述のように、この ConfigMap を使用して IAM プリンシパルに RBACs アクセスを追加することもできます。

クラスターの ConfigMap aws-auth を確認するには、次のコマンドを使用できます。

```
kubectl -n kube-system get configmap aws-auth -o yaml
```

これは、ConfigMap aws-auth のデフォルト設定のサンプルです。

```
apiVersion: v1
data:
  mapRoles: |
    - groups:
      - system:bootstrappers
      - system:nodes
      - system:node-proxier
      rolearn: arn:aws:iam::<AWS_ACCOUNT_ID>:role/kube-system-<SELF_GENERATED_UUID>
      username: system:node:{{SessionName}}
kind: ConfigMap
metadata:
  creationTimestamp: "2023-10-22T18:19:30Z"
  name: aws-auth
  namespace: kube-system
```

この ConfigMap のメインセッションは、基本的に 3 つのパラメータで構成される mapRolesブロックdataの にあります。

- groups: IAM ロールをマッピングする Kubernetes グループ。これは、デフォルトグループでも、または で指定されたカスタムグループclusterrolebindingでもかまいませんrolebinding。上記の例では、システムグループのみが宣言されています。
- rolearn: 次の形式 を使用して、Kubernetes グループ/グループが追加する AWS IAM ロールの ARN にマッピングされますarn:<PARTITION>:iam::<AWS_ACCOUNT_ID>:role/role-name。
- username: AWS IAM ロールにマッピングする Kubernetes 内のユーザー名。これは任意のカスタム名にすることができます。

また、AWS IAM ユーザーのアクセス許可をマッピングし、 の新しい設定ブロックを ConfigMap aws-auth dataの で定義してmapUsers、ユーザーarn の rolearn パラメータを置き換えることもできますが、ベストプラクティスとしてmapRoles、代わりに ユーザーに常にお勧めします。

アクセス許可を管理するには、ConfigMap aws-auth を編集して Amazon EKS クラスターへのアクセスを追加または削除します。aws-auth ConfigMap を手動で編集することは可能ですが、これは非常にセンチブな設定でありeksctl、不正確な設定は Amazon EKS クラスターの外部をロックする可能性があるため、 などのツールを使用することをお勧めします。詳細については、「」のサブセクションを確認してください。ツールを使用して、以下の aws-auth ConfigMap を変更します。

ConfigMap ベースのアクセス管理の利点

1. 設定ミスのリスクを軽減: API ベースの直接管理により、ConfigMap の手動編集に関連する一般的なエラーを排除します。これにより、クラスターからユーザーをロックアウトする可能性のある誤った削除や構文エラーを防ぐことができます。
2. 最小特権の原則を強化: クラスター作成者 ID からクラスター管理者アクセス許可が不要になり、より詳細で適切なアクセス許可の割り当てが可能になります。このアクセス許可は、Break Glass ユースケースに追加できます。
3. 拡張セキュリティモデル: アクセスエントリが適用される前に、組み込みの検証を提供します。さらに、 は認証のために AWS IAM とのより緊密な統合を提供します。
4. オペレーションの合理化: AWS ネイティブツールを使用してアクセス許可を管理するより直感的な方法を提供します。

クラスターアクセスに関する推奨事項

IAM Identity Center と CAM API を組み合わせる

- 管理の簡素化: クラスターアクセス管理 API を IAM アイデンティティセンターと組み合わせて使用すると、管理者は他の AWS のサービスと共に EKS クラスターアクセスを管理できるため、異なるインターフェイスを切り替えるか、ConfigMaps を手動で編集する必要がなくなります。
- アクセスエントリを使用して、クラスター外から IAM プリンシパルの Kubernetes アクセス許可を管理します。EKS API、AWS コマンドラインインターフェイス、AWS SDKs、AWS CloudFormation、AWS マネジメントコンソールを使用して、クラスターへのアクセスを追加および管理できます。つまり、クラスターを作成したのと同じツールでユーザーを管理できるということです。
- この例に示すオートメーションを活用して、エントリポイントとして CAM API を持つ IdP として AWS IAM Identity Center を使用してクラスターをデプロイします。
- 詳細な Kubernetes アクセス許可は、アクセスエントリとアクセスポリシーを介して SSO ID に関連付けられた IAM プリンシパルを持つ Kubernetes ユーザーまたはグループをマッピングすることで適用できます。
- 開始するには、「アクセスエントリを使用するように認証モードを変更する」に従い、次に「既存の aws-auth ConfigMap エントリをアクセスエントリに移行する」に従います。

EKS クラスターエンドポイントをプライベートにする

デフォルトでは、EKS クラスターをプロビジョニングすると、API クラスターエンドポイントはパブリックに設定されます。つまり、インターネットからアクセスできます。インターネットからアクセス可能であっても、エンドポイントは引き続き安全と見なされます。これは、すべての API リクエストを IAM で認証し、Kubernetes RBAC で承認する必要があるためです。つまり、企業のセキュリティポリシーで、インターネットからの API へのアクセスを制限することが義務付けられている場合、またはクラスター VPC の外部にトラフィックをルーティングできない場合は、次のことができます。

- EKS クラスターエンドポイントをプライベートに設定します。このトピックの詳細については、「クラスターエンドポイントアクセスの変更」を参照してください。
- クラスターエンドポイントをパブリックのままにして、クラスターエンドポイントと通信できる CIDR ブロックを指定します。ブロックは、実質的にクラスターエンドポイントへのアクセスが許可されているパブリック IP アドレスのホワイトリストに登録されたセットです。

- ホワイトリストに登録された CIDR ブロックのセットを使用してパブリックアクセスを設定し、プライベートエンドポイントアクセスを有効に設定します。これにより、コントロールプレーンのプロビジョニング時にクラスター VPC にプロビジョニングされるクロスアカウント ENIs を介して kubelets (ワーカー) と Kubernetes API 間のすべてのネットワークトラフィックを強制しながら、特定の範囲のパブリック IPs からのパブリックアクセスが可能になります。

認証にサービスアカウントトークンを使用しない

サービスアカウントトークンは、存続期間の長い静的認証情報です。侵害、紛失、盗難された場合、攻撃者はサービスアカウントが削除されるまで、そのトークンに関連付けられたすべてのアクションを実行できる可能性があります。CI/CD パイプラインアプリケーションなど、クラスター外から Kubernetes API を消費する必要があるアプリケーションには例外を付与する必要がある場合があります。このようなアプリケーションが EC2 インスタンスなどの AWS インフラストラクチャで実行されている場合は、インスタンスプロファイルを使用して Kubernetes RBAC ロールにマッピングすることを検討してください。

AWS リソースへの最小特権アクセスを採用する

IAM ユーザーは、Kubernetes API にアクセスするために AWS リソースに権限を割り当てる必要はありません。EKS クラスターへのアクセス権を IAM ユーザーに付与する必要がある場合は、特定の Kubernetes RBAC `aws-auth` グループにマッピングするエントリをそのユーザーの ConfigMap に作成します。

クラスター作成者プリンシパルから cluster-admin アクセス許可を削除する

デフォルトでは、Amazon EKS クラスターは、クラスター作成者プリンシパルにバインドされた永続的な`cluster-admin`アクセス許可で作成されます。Cluster Access Manager API では、 `API_AND_CONFIG_MAP` または `API`認証モードを使用する場合`--access-config bootstrapClusterCreatorAdminPermissions`false、このアクセス許可なしでクラスターを作成できます。このアクセスを取り消すことは、クラスター設定への望ましくない変更を避けるためのベストプラクティスと考えられています。このアクセスを取り消すプロセスは、クラスターへの他のアクセスを取り消す同じプロセスに従います。

API を使用すると、IAM プリンシパルとアクセスポリシーの関連付けを解除する柔軟性が得られます。この場合、 です`AmazonEKSClusterAdminPolicy`。

```
$ aws eks list-associated-access-policies \
    --cluster-name <CLUSTER_NAME> \
    --principal-arn <IAM_PRINCIPAL_ARN>
```

```
$ aws eks disassociate-access-policy --cluster-name <CLUSTER_NAME> \
    --principal-arn <IAM_PRINCIPAL_ARN. \
    --policy-arn arn:aws:eks::aws:cluster-access-policy/AmazonEKSClusterAdminPolicy
```

または、 アクセスcluster-admin許可に関連付けられたアクセスエントリを完全に削除します。

```
$ aws eks list-access-entries --cluster-name <CLUSTER_NAME>

{
    "accessEntries": []
}

$ aws eks delete-access-entry --cluster-name <CLUSTER_NAME> \
  --principal-arn <IAM_PRINCIPAL_ARN>
```

このアクセスは、クラスターにアクセスできないインシデント、緊急、またはブレークグラスのシナリオ中に必要に応じて再度許可できます。

クラスターがCONFIG_MAP認証方法でまだ設定されている場合、追加のすべてのユーザーに ConfigMap aws-auth 経由でクラスターへのアクセスを許可する必要があります。ConfigMap aws-auth を設定した後、クラスターを作成したエンティティに割り当てられたロールを削除して再作成できるのは、インシデント、緊急、またはブレークグラスシナリオが発生した場合、または ConfigMap aws-auth が破損していてクラスターにアクセスできない場合のみです。これは、本番稼働用クラスターで特に役立ちます。

複数のユーザーがクラスターへの同一のアクセスを必要とする場合の IAM ロールの使用

個々の IAM ユーザーごとにエントリを作成するのではなく、それらのユーザーが IAM ロールを引き受け、そのロールを Kubernetes RBAC グループにマッピングできるようにします。これは、特にアクセスを必要とするユーザーの数が増えるにつれて、保守が容易になります。

> **Important**
>
> ConfigMap によってマッピングされた IAM エンティティを使用して EKS aws-auth クラスターにアクセスする場合、記述されたユーザー名は Kubernetes 監査ログのユーザーフィールドに記録されます。IAM ロールを使用している場合、そのロールを引き受ける実際のユーザーは記録されず、監査もできません。

認証方法として configMap aws-auth を引き続き使用する場合、IAM ロールに K8s RBAC アクセス許可を割り当てるときは、ユーザー名に \{{SessionName}} を含める必要があります。これにより、監査ログにセッション名が記録されるため、実際のユーザーがこのロールを引き受けるユーザーを CloudTrail ログとともに追跡できます。

```
- rolearn: arn:aws:iam::XXXXXXXXXXXX:role/testRole
  username: testRole:{{SessionName}}
  groups:
    - system:masters
```

> Kubernetes 1.20 以降`user.extra.sessionName.0`では、 が Kubernetes 監査ログに追加されたため、この変更は不要になりました。

RoleBindings と ClusterRoleBindings の作成時に最小特権アクセスを使用する

AWS リソースへのアクセス許可の付与に関する前のポイントと同様に、RoleBindings と ClusterRoleBindings には、特定の関数の実行に必要な一連のアクセス許可のみを含める必要があります。絶対に必要でない限り、ロールと ClusterRoles `["*"]`で を使用しないでください。割り当てるアクセス許可が不明な場合は、audit2rbac などのツールを使用して、Kubernetes 監査ログで観測された API コールに基づいてロールとバインドを自動的に生成することを検討してください。

自動プロセスを使用してクラスターを作成する

前の手順で説明したように、Amazon EKS クラスターを作成するときに、`API_AND_CONFIG_MAP`または `API`認証モードを使用せず、クラスター作成者にアクセス`cluster-admin`許可を委任することをオプトアウトしない場合、クラスターを作成するフェデレーティッドユーザーなどの IAM エンティティユーザーまたはロールには、クラスターの RBAC 設定でアクセス`system:masters`許可が自動的に付与されます。このアクセス許可を削除するベストプラクティスであっても、ここで説明するように、`CONFIG_MAP`認証方法を使用して ConfigMap aws-auth に依存する場合、このアクセスを取り消すことはできません。したがって、専用の IAM ロールに関連付けられたインフラストラクチャ自動化パイプラインを使用してクラスターを作成し、他のユーザーやエンティティが引き受けるアクセス許可なしで、このロールのアクセス許可、ポリシー、パイプラインをトリガーするアクセス権を持つユーザーを定期的に監査することをお勧めします。また、このロールはクラスターでルーチンアクションを実行するために使用しないでください。また、たとえば SCM コードの変更を介してパイプラインによってトリガーされるクラスターレベルのアクションにのみ使用してください。

専用 IAM ロールを使用してクラスターを作成する

Amazon EKS クラスターを作成すると、クラスターを作成するフェデレーティッドユーザーなどの IAM エンティティユーザーまたはロールに、クラスターの RBAC 設定の`system:masters`アクセス許可が自動的に付与されます。このアクセスは削除できず、ConfigMap `aws-auth` を通じて管理されません。したがって、専用の IAM ロールを使用してクラスターを作成し、このロールを引き受けることができるユーザーを定期的に監査することをお勧めします。このロールは、クラスターでルーチンアクションを実行するために使用しないでください。代わりに、この目的のために ConfigMap `aws-auth` を介してクラスターへのアクセスを追加のユーザーに許可する必要があります。ConfigMap `aws-auth` を設定したら、ロールを保護し、クラスターにアクセスできないシナリオの一時的な昇格特権モード/ブレークグラスでのみ使用する必要があります。これは、直接ユーザーアクセスが設定されていないクラスターで特に便利です。

クラスターへのアクセスを定期的に監査する

アクセスを必要とするユーザーは、時間の経過とともに変化する可能性があります。ConfigMap `aws-auth` を定期的に監査して、アクセス権が付与されたユーザーと割り当てられた権限を確認する計画を立てます。kubectl-who-can や rbac-lookup などのオープンソースツールを使用して、特定のサービスアカウント、ユーザー、またはグループにバインドされたロールを調べることもできます。このトピックは、監査に関するセクションに進むときに詳しく説明します。その他のアイデアは、NCC グループのこの記事で確認できます。

configMap **`aws-auth`** に依存する場合は、ツールを使用して変更を行います。

正しくフォーマットされていない aws-auth ConfigMap を使用すると、クラスターにアクセスできなくなる可能性があります。ConfigMap を変更する必要がある場合は、 ツールを使用します。

eksctl `eksctl`CLI には、aws-auth ConfigMap に ID マッピングを追加するためのコマンドが含まれています。

CLI ヘルプの表示:

```
$ eksctl create iamidentitymapping --help
...
```

Amazon EKS クラスターにマッピングされた ID を確認します。

```
$ eksctl get iamidentitymapping --cluster $CLUSTER_NAME --region $AWS_REGION
```

```
ARN                                                                    USERNAME
               GROUPS                                                   ACCOUNT
arn:aws:iam::788355785855:role/kube-system-<SELF_GENERATED_UUID>      system:node:
{{SessionName}}     system:bootstrappers,system:nodes,system:node-proxier
```

IAM ロールをクラスター管理者にする:

```
$ eksctl create iamidentitymapping --cluster  <CLUSTER_NAME> --region=<region> --arn
 arn:aws:iam::123456:role/testing --group system:masters --username admin
...
```

詳細については、eksctlドキュメントを参照してください。

keikoproj による aws-auth

aws-auth の keikoproj には、cli と go ライブラリの両方が含まれています。

CLI ヘルプをダウンロードして表示する:

```
$ go get github.com/keikoproj/aws-auth
...
$ aws-auth help
...
```

または、kubectl 用の krew プラグインマネージャーaws-authを使用して をインストールします。

```
$ kubectl krew install aws-auth
...
$ kubectl aws-auth
...
```

go ライブラリなどの詳細については、GitHub の aws-auth ドキュメントを参照してください。

AWS IAM Authenticator CLI

aws-iam-authenticator プロジェクトには、ConfigMap を更新するための CLI が含まれています。

GitHub でリリースをダウンロードします。

クラスターのアクセス許可を IAM ロールに追加します。

```
$ ./aws-iam-authenticator add role --rolearn arn:aws:iam::185309785115:role/lil-dev-
role-cluster --username lil-dev-user --groups system:masters --kubeconfig ~/.kube/
config
...
```

認証とアクセス管理の代替アプローチ

IAM は EKS クラスターへのアクセスを必要とするユーザーを認証するために推奨される方法ですが、認証プロキシと Kubernetes 偽装を使用して GitHub などの OIDC ID プロバイダーを使用できます。このような 2 つのソリューションの投稿は、AWS オープンソースブログで公開されています。

- テレポートでの GitHub 認証情報を使用した EKS への認証
- kube-oidc-proxy を使用した複数の EKS クラスター間の一貫した OIDC 認証

> **Important**
>
> EKS は、プロキシを使用せずに OIDC 認証をネイティブにサポートします。詳細については、起動ブログ「Amazon EKS の OIDC ID プロバイダー認証の紹介」を参照してください。さまざまな認証方法用のコネクタを備えた一般的なオープンソース OIDC プロバイダーである Dex で EKS を設定する方法の例については、「Dex & dex-k8s-authenticator を使用した Amazon EKS への認証」を参照してください。ブログで説明されているように、OIDC プロバイダーによって認証されたユーザーのユーザー名/グループが Kubernetes 監査ログに表示されます。

AWS SSO を使用して、AWS を Azure AD などの外部 ID プロバイダーとフェデレーションすることもできます。これを使用する場合、AWS CLI v2.0 には、SSO セッションを現在の CLI セッションに簡単に関連付けて IAM ロールを引き受けることができる名前付きプロファイルを作成するオプションが含まれています。ユーザーの Kubernetes RBAC グループを決定するために IAM ロールが使用されkubectlるため、 を実行する前にロールを引き受ける必要があることに注意してください。

EKS ポッドの ID と認証情報

Kubernetes クラスター内で実行される特定のアプリケーションでは、適切に機能するために Kubernetes API を呼び出すアクセス許可が必要です。たとえば、AWS Load Balancer Controller はサービスのエンドポイントを一覧表示できる必要があります。また、コントローラーは AWS

APIs を呼び出して ALB をプロビジョニングおよび設定できる必要があります。このセクションでは、Pod に権限を割り当てるためのベストプラクティスについて説明します。

Kubernetes サービスアカウント

サービスアカウントは、ポッドに Kubernetes RBAC ロールを割り当てることができる特殊なタイプのオブジェクトです。デフォルトのサービスアカウントは、クラスター内の名前空間ごとに自動的に作成されます。特定のサービスアカウントを参照せずにポッドを名前空間にデプロイすると、その名前空間のデフォルトのサービスアカウントがポッドに自動的に割り当てられ、シークレット、つまりそのサービスアカウントのサービスアカウント (JWT) トークンが のボリュームとしてポッドにマウントされます/var/run/secrets/kubernetes.io/serviceaccount。そのディレクトリのサービスアカウントトークンをデコードすると、次のメタデータが表示されます。

```
{
  "iss": "kubernetes/serviceaccount",
  "kubernetes.io/serviceaccount/namespace": "default",
  "kubernetes.io/serviceaccount/secret.name": "default-token-5pv4z",
  "kubernetes.io/serviceaccount/service-account.name": "default",
  "kubernetes.io/serviceaccount/service-account.uid":
 "3b36ddb5-438c-11ea-9438-063a49b60fba",
  "sub": "system:serviceaccount:default:default"
}
```

デフォルトのサービスアカウントには、Kubernetes API に対する次のアクセス許可があります。

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  annotations:
    rbac.authorization.kubernetes.io/autoupdate: "true"
  creationTimestamp: "2020-01-30T18:13:25Z"
  labels:
    kubernetes.io/bootstrapping: rbac-defaults
  name: system:discovery
  resourceVersion: "43"
  selfLink: /apis/rbac.authorization.k8s.io/v1/clusterroles/system%3Adiscovery
  uid: 350d2ab8-438c-11ea-9438-063a49b60fba
rules:
- nonResourceURLs:
  - /api
  - /api/*
```

```
  - /apis
  - /apis/*
  - /healthz
  - /openapi
  - /openapi/*
  - /version
  - /version/
  verbs:
  - get
```

このロールは、認証されていないユーザーと認証されたユーザーが API 情報を読み取ることを許可し、パブリックにアクセス可能であると安全と見なされます。

Pod 内で実行されているアプリケーションが Kubernetes APIs を呼び出す場合、それらの APIs を呼び出すアクセス許可を明示的に付与するサービスアカウントを Pod に割り当てる必要があります。ユーザーアクセスのガイドラインと同様に、サービスアカウントにバインドされたロールまたは ClusterRole は、アプリケーションが機能するために必要な API リソースとメソッドに制限する必要があります。デフォルト以外のサービスアカウントを使用するには、Pod の `spec.serviceAccountName`フィールドを、使用するサービスアカウントの名前に設定します。サービスアカウントの作成の詳細については、https://kubernetes.io/docs/reference/access-authn-authz/rbac/#service-account-permissions を参照してください。

> **Note**
>
> Kubernetes 1.24 以前は、Kubernetes はサービスアカウントごとにシークレットを自動的に作成していました。このシークレットは /var/run/secrets/kubernetes.io/serviceaccount でポッドにマウントされ、ポッドによって Kubernetes API サーバーへの認証に使用されます。Kubernetes 1.24 では、サービスアカウントトークンはポッドの実行時に動的に生成され、デフォルトでは 1 時間のみ有効です。サービスアカウントのシークレットは作成されません。Jenkins など、Kubernetes API に対して認証する必要があるクラスターの外部で実行されるアプリケーションがある場合は、 のようなサービスアカウントを参照する注釈`kubernetes.io/service-account-token`とともに、 タイプのシークレットを作成する必要があります`metadata.annotations.kubernetes.io/service-account.name: <SERVICE_ACCOUNT_NAME>`。この方法で作成されたシークレットは期限切れになりません。

サービスアカウントの IAM ロール (IRSA)

IRSA は、Kubernetes サービスアカウントに IAM ロールを割り当てることができる機能です。これは、サービスアカウントトークンボリューム射影と呼ばれる Kubernetes 機能を活用することで機能します。ポッドが IAM ロールを参照するサービスアカウントで設定されている場合、Kubernetes API サーバーは起動時にクラスターのパブリック OIDC 検出エンドポイントを呼び出します。エンドポイントは、Kubernetes によって発行された OIDC トークンと、ボリュームとしてマウントされた結果のトークンを暗号的に署名します。この署名付きトークンにより、ポッドは AWS APIs に関連付けられた IAM ロールを呼び出すことができます。AWS API が呼び出されると、AWS SDK`sts:AssumeRoleWithWebIdentity`。トークンの署名を検証すると、IAM は Kubernetes が発行したトークンを一時的な AWS ロール認証情報と交換します。

IRSA を使用する場合は、AWS STS への不要な呼び出しを避けるために、AWS SDK セッションを再利用することが重要です。

IRSA の (JWT) トークンをデコードすると、次のような出力が生成されます。

```
{
  "aud": [
    "sts.amazonaws.com"
  ],
  "exp": 1582306514,
  "iat": 1582220114,
  "iss": "https://oidc.eks.us-west-2.amazonaws.com/id/
D43CF17C27A865933144EA99A26FB128",
  "kubernetes.io": {
    "namespace": "default",
    "pod": {
      "name": "alpine-57b5664646-rf966",
      "uid": "5a20f883-5407-11ea-a85c-0e62b7a4a436"
    },
    "serviceaccount": {
      "name": "s3-read-only",
      "uid": "a720ba5c-5406-11ea-9438-063a49b60fba"
    }
  },
  "nbf": 1582220114,
  "sub": "system:serviceaccount:default:s3-read-only"
}
```

この特定のトークンは、IAM ロールを引き受けることで、ポッドに S3 へのビューのみの権限を付与します。アプリケーションが S3 からの読み取りを試みると、トークンは次のような一時的な IAM 認証情報のセットと交換されます。

```
{
    "AssumedRoleUser": {
        "AssumedRoleId": "AROA36C6WWEJULFUYMPB6:abc",
        "Arn": "arn:aws:sts::123456789012:assumed-role/eksctl-winterfell-addon-
iamserviceaccount-de-Role1-1D61LT75JH3MB/abc"
    },
    "Audience": "sts.amazonaws.com",
    "Provider": "arn:aws:iam::123456789012:oidc-provider/oidc.eks.us-
west-2.amazonaws.com/id/D43CF17C27A865933144EA99A26FB128",
    "SubjectFromWebIdentityToken": "system:serviceaccount:default:s3-read-only",
    "Credentials": {
        "SecretAccessKey": "wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY",
        "SessionToken": "FwoGZXIvYXdzEGMaDMLxAZkuLpmSwYXShiL9A1S0X87VBC1mHCrRe/
pB2oesl1eXxUYnPJyC9ayOoXMvqXQsomq0xs6OqZ3vaa5Iw1HIyA4Cv1suLaOCoU3hNvOIJ6C94H1vU0siQYk7DIq9Av5RZ
Zv2A6zp5xGz9cWj2f0aD9v66vX4bexOs5t/YYhwuwAvkkJPSIGvxja0xRThnceHyFHKtj0Hbi/
PWAtlI8YJcDX69cM30JAHDdQHltm/4scFptW1hlvMaPWReCAaCrsHrATyka7ttw5YlUyvZ8EPogj6fwHlxmrXM9h1Bqdiko
LgkWsCTGlYcY8z3zkigJMbYn07ewTL5Ss7LazTJJa758I7PZan/
v3xQHd5DEc5WBneiV3iOznDFgup0VAMkIviVjVCkszaPSVEdK2NU7jtrh6Jfm7bU/3P6ZGCkyDLIa8MBn9KPXeJd/
yjTk5IifIwO/mDpGNUribg6TPxhzZ8b/XdZO1kS1gVgqjXyVCM+BRBh6C4H21w/eMzjCtDIpoxt5rGKL6Nu/
IFMipoC4fgx6LIIHwtGYMG7SWQi7OsMAkiwZRg0n68/RqWgLzBt/4pfjSRYuk=",
        "Expiration": "2020-02-20T18:49:50Z",
        "AccessKeyId": "ASIAIOSFODNN7EXAMPLE"
    }
}
```

EKS コントロールプレーンの一部として実行される変更ウェブフックは、AWS ロール ARN とウェブ ID トークンファイルへのパスを環境変数として Pod に挿入します。これらの値は手動で指定することもできます。

```
AWS_ROLE_ARN=arn:aws:iam::AWS_ACCOUNT_ID:role/IAM_ROLE_NAME
AWS_WEB_IDENTITY_TOKEN_FILE=/var/run/secrets/eks.amazonaws.com/serviceaccount/token
```

kubelet は、合計 TTL の 80% より古い場合、または 24 時間後に、射影されたトークンを自動的にローテーションします。AWS SDKs は、ローテーション時にトークンを再ロードする責任があります。IRSA の詳細については、https://docs.aws.amazon.com/eks/latest/userguide/iam-roles-for-service-accounts-technical-overview.html を参照してください。

EKS Pod Identity

EKS Pod Identity は re:Invent 2023 で起動された機能で、AWS アカウント内のクラスターごとに Open Id Connect (OIDC) ID プロバイダー (IDP) を設定することなく、kubernetes サービスアカウントに IAM ロールを割り当てることができます。EKS Pod Identity を使用するには、対象となるすべてのワーカーノードで DaemonSet ポッドとして実行されるエージェントをデプロイする必要があります。このエージェントは EKS アドオンとして利用でき、EKS Pod Identity 機能を使用するための前提条件です。この機能を使用するには、アプリケーションがサポートされているバージョンの AWS SDK を使用する必要があります。

EKS Pod Identity がポッド用に設定されている場合、EKS はポッド ID トークンを にマウントして更新します`/var/run/secrets/pods.eks.amazonaws.com/serviceaccount/eks-pod-identity-token`。このトークンは、AWS SDK が EKS Pod Identity Agent と通信するために使用します。EKS Pod Identity Agent は、ポッド ID トークンとエージェントの IAM ロールを使用して、AssumeRoleForPodIdentity API を呼び出してポッドの一時的な認証情報を作成します。ポッドに配信されるポッド ID トークンは、EKS クラスターから発行され、暗号化によって署名された JWT であり、EKS Pod ID で使用するための適切な JWT クレームがあります。

EKS Pod ID の詳細については、このブログを参照してください。

EKS Pod ID を使用するには、アプリケーションコードを変更する必要はありません。サポートされている AWS SDK バージョンでは、認証情報プロバイダーチェーンを使用して、EKS Pod Identity で使用できる認証情報が自動的に検出されます。IRSA と同様に、EKS ポッド ID はポッド内の変数を設定して、AWS 認証情報の検索方法を指示します。

EKS Pod ID の IAM ロールの使用

- EKS Pod Identity は、EKS クラスターと同じ AWS アカウントに属する IAM ロールのみを直接引き受けることができます。別の AWS アカウントの IAM ロールにアクセスするには、SDK 設定またはアプリケーションのコードでプロファイルを設定して、そのロールを引き受ける必要があります。 https://docs.aws.amazon.com/IAM/latest/UserGuide/sts_example_sts_AssumeRole_section.html
- EKS Pod Identity がサービスアカウント用に設定されている場合、Pod Identity Association を設定するユーザーまたはプロセスには、そのロールの使用`iam:PassRole`権限が必要です。
- 各サービスアカウントには、EKS Pod ID を介して関連付けられる IAM ロールが 1 つだけありますが、同じ IAM ロールを複数のサービスアカウントに関連付けることができます。

- EKS Pod Identity で使用される IAM ロールは、pods.eks.amazonaws.comサービスプリンシパルがそれらを引き受け、セッションタグを設定できるようにする必要があります。以下は、EKS Pod ID に IAM ロールの使用を許可するロール信頼ポリシーの例です。

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "pods.eks.amazonaws.com"
      },
      "Action": [
        "sts:AssumeRole",
        "sts:TagSession"
      ],
      "Condition": {
        "StringEquals": {
          "aws:SourceOrgId": "${aws:ResourceOrgId}"
        }
      }
    }
  ]
}
```

AWS ではaws:SourceOrgId、 のような条件キーを使用して、サービス間の混乱した代理問題から保護することをお勧めします。上記のロール信頼ポリシーの例では、 ResourceOrgIdは、AWS アカウントが属する AWS Organizations の AWS Organizations Organization ID と等しい変数です。EKS は、EKS Pod Identity でロールを引き受けるときaws:SourceOrgIdと同じ の値を渡します。

ABAC および EKS ポッド ID

EKS Pod Identity が IAM ロールを引き受けると、次のセッションタグが設定されます。

EKS ポッド ID セッションタグ	値
kubernetes-namespace	EKS Pod ID に関連付けられたポッドが実行される名前空間。

EKS ポッド ID セッションタグ	値
kubernetes-service-account	EKS Pod ID に関連付けられた kubernetes サービスアカウントの名前
eks-cluster-arn	EKS クラスターの ARN。例: `arn:${Partition}:eks:${Region}:${Account}:cluster/${ClusterName}` 。クラスター ARN は一意ですが、クラスターが削除され、同じ AWS アカウント内で同じ名前の同じリージョンで再作成されると、同じ ARN になります。
eks-cluster-name	EKS クラスターの名前。EKS クラスター名は、AWS アカウント内、および他の AWS アカウントの EKS クラスター内で同じにすることができます。
kubernetes-pod-name	EKS のポッドの名前。
kubernetes-pod-uid	EKS のポッドの UID。

これらのセッションタグを使用すると、属性ベースのアクセスコントロール (ABAC) を使用して、AWS リソースへのアクセスを特定の kubernetes サービスアカウントにのみ付与できます。その場合、kubernetes サービスアカウントは名前空間内でのみ一意であり、kubernetes 名前空間は EKS クラスター内でのみ一意であることを理解することが重要です。これらのセッションタグは、などの`aws:PrincipalTag/<tag-key>`グローバル条件キーを使用して AWS ポリシーでアクセスできます。 `aws:PrincipalTag/eks-cluster-arn`

例えば、IAM またはリソースポリシーを使用してアカウント内の AWS リソースにアクセスするためのアクセスを特定のサービスアカウントのみに付与する場合は、 `eks-cluster-arn` と `kubernetes-namespace` タグに加えて をチェックして、目的のクラスターのサービスアカウントのみがそのリソースにアクセスできる`kubernetes-service-account`ことを確認する必要があります。他のクラスターは `kubernetes-service-accounts`と を同じにすることができます`kubernetes-namespaces`。

この S3 バケットポリシーの例では、EKS クラスターが AWS アカウント でホストされている kubernetes-service-account、 kubernetes-namespaceの場合にのみeks-cluster-arn、アタッチされている S3 バケット内のオブジェクトへのアクセスを許可します111122223333。

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": {
                "AWS": "arn:aws:iam::111122223333:root"
            },
            "Action": "s3:*",
            "Resource":             [
                "arn:aws:s3:::ExampleBucket/*"
            ],
            "Condition": {
                "StringEquals": {
                    "aws:PrincipalTag/kubernetes-service-account": "s3objectservice",
                    "aws:PrincipalTag/eks-cluster-arn": "arn:aws:eks:us-
west-2:111122223333:cluster/ProductionCluster",
                    "aws:PrincipalTag/kubernetes-namespace": "s3datanamespace"
                }
            }
        }
    ]
}
```

IRSA と比較した EKS Pod ID

EKS ポッドに一時的な AWS 認証情報を配信するには、EKS ポッド ID と IRSA の両方をお勧めします。IRSA の特定のユースケースがない限り、EKS を使用するときは EKS Pod ID を使用することをお勧めします。この表は、2 つの機能の比較に役立ちます。

#	EKS Pod Identity	IRSA
AWS アカウントに OIDC IDP を作成するアクセス許可が必要ですか?	不可	はい

#	EKS Pod Identity	IRSA
クラスターごとに一意の IDP 設定が必要です	不可	はい
ABAC で使用する関連セッションタグを設定します	はい	なし
iam:PassRole チェックが必要ですか?	はい	なし
AWS アカウントの AWS STS Quota を使用しますか?	不可	はい
他の AWS アカウントにアクセスできる	ロールの連鎖により間接的に	sts:AssumeRoleWithWebIdentity で直接
AWS SDKs との互換性	はい	はい
ノードに Pod Identity Agent デーモンセットが必要ですか?	はい	なし

EKS ポッドの ID と認証情報に関する推奨事項

IRSA を使用するように aws-node デーモンセットを更新する

現在、aws-node デーモンセットは、EC2 インスタンスに割り当てられたロールを使用してポッドに IPs を割り当てるように設定されています。このロールには、AmazonEKS_CNI_Policy や EC2ContainerRegistryReadOnly など、複数の AWS 管理ポリシーが含まれており、ノードで実行されているすべてのポッドが ENIs、IP アドレスの割り当て/割り当て解除、または ECR からのイメージのプルを効果的に許可します。これはクラスターにリスクをもたらすため、IRSA を使用するように aws-node デーモンセットを更新することをお勧めします。これを行うためのスクリプトは、このガイドの リポジトリにあります。

aws-node デーモンセットは、バージョン v1.15.5 以降の EKS Pod ID をサポートしています。

ワーカーノードに割り当てられたインスタンスプロファイルへのアクセスを制限する

IRSA または EKS Pod ID を使用する場合、最初に IRSA または EKS Pod ID を使用するようにポッドの認証情報チェーンを更新しますが、ポッドはワーカーノードに割り当てられたインスタンスプロファイルの権限を継承できます。これらのアクセス許可を必要としないポッドの場合、インスタンスメタデータへのアクセスをブロックして、アプリケーションに必要なアクセス許可のみを持ち、ノードを持たないようにすることができます。

> ⚠ Warning
>
> インスタンスメタデータへのアクセスをブロックすると、IRSA または EKS Pod ID を使用しないポッドがワーカーノードに割り当てられたロールを継承できなくなります。

以下の例のように、インスタンスに IMDSv2 のみの使用を要求し、ホップ数を 1 に更新することで、インスタンスメタデータへのアクセスをブロックできます。これらの設定をノードグループの起動テンプレートに含めることもできます。インスタンスメタデータを無効にしないでください。これにより、ノード終了ハンドラーなどのコンポーネントや、インスタンスメタデータに依存する他のモノが正しく動作しなくなります。

```
$ aws ec2 modify-instance-metadata-options --instance-id <value> --http-tokens required
 --http-put-response-hop-limit 1
...
```

Terraform を使用して Managed Node Groups で使用する起動テンプレートを作成する場合は、メタデータブロックを追加して、次のコードスニペットに示すようにホップ数を設定します。

```
tf hl_lines="7" resource "aws_launch_template" "foo" { name = "foo" …
 metadata_options { http_endpoint = "enabled" http_tokens = "required"
http_put_response_hop_limit = 1 instance_metadata_tags = "enabled" } …
```

ノードで iptables を操作することで、EC2 メタデータへのポッドのアクセスをブロックすることもできます。この方法の詳細については、「インスタンスメタデータサービスへのアクセスの制限」を参照してください。

IRSA または EKS Pod ID をサポートしていない古いバージョンの AWS SDK を使用しているアプリケーションがある場合は、SDK バージョンを更新する必要があります。

IRSA ロールの IAM ロール信頼ポリシーをサービスアカウント名、名前空間、クラスターにスコープする

信頼ポリシーの範囲は、名前空間または名前空間内の特定のサービスアカウントに限定できます。IRSA を使用する場合は、サービスアカウント名を含めて、ロールの信頼ポリシーをできるだけ明確にすることをお勧めします。これにより、同じ名前空間内の他のポッドがロールを引き受けるのを効果的に防ぐことができます。これを使用してサービスアカウント/IAM ロールを作成すると、CLI `eksctl`は自動的にこれを行います。詳細については、https://eksctl.io/usage/iamserviceaccounts/ を参照してください。

IAM を直接使用すると、条件を使用して`:sub`クレームが想定する名前空間とサービスアカウントであることを確認する条件がロールの信頼ポリシーに追加されます。例えば、「system:serviceaccount:default:s3-read-only」というサブクレームを持つ IRSA トークンがある前は、これは`default`名前空間で、サービスアカウントは ですs3-read-only。クラスターの特定の名前空間内のサービスアカウントのみがそのロールを引き受けられるようにするには、次のような条件を使用します。

```
 "Condition": {
     "StringEquals": {
         "oidc.eks.us-west-2.amazonaws.com/id/D43CF17C27A865933144EA99A26FB128:aud":
"sts.amazonaws.com",
         "oidc.eks.us-west-2.amazonaws.com/id/D43CF17C27A865933144EA99A26FB128:sub":
"system:serviceaccount:default:s3-read-only"
     }
 }
```

アプリケーションごとに 1 つの IAM ロールを使用する

IRSA と EKS Pod Identity の両方を使用する場合、各アプリケーションに独自の IAM ロールを付与するのがベストプラクティスです。これにより、別のアプリケーションに影響を与えることなく 1 つのアプリケーションを変更できるため、分離が改善され、必要なアクセス許可のみをアプリケーションに付与することで、最小特権のプリンシパルを適用できます。

EKS Pod Identity で ABAC を使用する場合、複数のサービスアカウントで共通の IAM ロールを使用し、それらのセッション属性を使用してアクセス制御を行うことができます。これは、ABAC ではより少ない IAM ロールで運用できるため、大規模に運用する場合に特に便利です。

アプリケーションが IMDS にアクセスする必要がある場合は、IMDSv2 を使用し、EC2 インスタンスのホップ制限を 2 に引き上げます。

IMDSv2 では、PUT リクエストを使用してセッショントークンを取得する必要があります。最初の PUT リクエストには、セッショントークンの TTL を含める必要があります。AWS SDKs の新しいバージョンでは、これとトークンの更新が自動的に処理されます。また、IP 転送を防ぐために、EC2 インスタンスのデフォルトのホップ制限が意図的に 1 に設定されていることに注意してください。その結果、EC2 インスタンスで実行されるセッショントークンをリクエストするポッドは、最終的にタイムアウトし、IMDSv1 データフローの使用にフォールバックする可能性があります。EKS は、v1 と v2 の両方を有効にし、eksctl によってプロビジョニングされたノードまたは公式の CloudFormation テンプレートを使用してホップ制限を 2 に変更することで、IMDSv2 のサポートを追加します。 CloudFormation

サービスアカウントトークンの自動マウントを無効にする

アプリケーションが Kubernetes API を呼び出す必要がない場合は、アプリケーションの PodSpec `automountServiceAccountToken` 属性を `false` に設定するか、各名前空間のデフォルトのサービスアカウントにパッチを適用して、ポッドに自動的にマウントされなくなります。例えば、次のようになります。

```
kubectl patch serviceaccount default -p $'automountServiceAccountToken: false'
```

アプリケーションごとに専用サービスアカウントを使用する

各アプリケーションには、独自の専用サービスアカウントが必要です。これは、Kubernetes API のサービスアカウントと IRSA および EKS Pod Identity に適用されます。

> **Important**
>
> IRSA の使用時にインプレースクラスターアップグレードを実行する代わりに、クラスターのアップグレードにブルー/グリーンアプローチを採用する場合は、新しいクラスターの OIDC エンドポイントを使用して、各 IRSA IAM ロールの信頼ポリシーを更新する必要があります。ブルー/グリーンクラスターのアップグレードでは、古いクラスターとともに新しいバージョンの Kubernetes を実行するクラスターを作成し、ロードバランサーまたはサービスメッシュを使用して、古いクラスターで実行されているサービスから新しいクラスターにトラフィックをシームレスに移行します。EKS Pod Identity で Blue/Green クラスターのアップグレードを使用する場合は、新しいクラスター内の IAM ロールとサービスアカウントの間

にポッド ID の関連付けを作成します。また、sourceArn条件がある場合は、IAM ロールの信頼ポリシーを更新します。

ルート以外のユーザーとしてアプリケーションを実行する

デフォルトでは、コンテナはルートとして実行されます。これにより、ウェブ ID トークンファイルを読み取ることができますが、コンテナをルートとして実行することはベストプラクティスとは見なされません。別の方法として、 spec.securityContext.runAsUser 属性を PodSpec に追加することを検討してください。の値は任意の値runAsUserです。

次の例では、Pod 内のすべてのプロセスは、 runAsUserフィールドで指定されたユーザー ID で実行されます。

```
apiVersion: v1
kind: Pod
metadata:
  name: security-context-demo
spec:
  securityContext:
    runAsUser: 1000
    runAsGroup: 3000
  containers:
  - name: sec-ctx-demo
    image: busybox
    command: [ "sh", "-c", "sleep 1h" ]
```

コンテナを非ルートユーザーとして実行すると、トークンにはデフォルトで 0600 [ルート] アクセス許可が割り当てられるため、コンテナが IRSA サービスアカウントトークンを読み取ることができなくなります。コンテナの securityContext を更新して fsgroup=65534 [Nobody] を含めると、コンテナはトークンを読み取ることができます。

```
spec:
  securityContext:
    fsGroup: 65534
```

Kubernetes 1.19 以降では、この変更は不要になり、アプリケーションは Nobody グループに追加せずに IRSA サービスアカウントトークンを読み取ることができます。

アプリケーションへの最小特権アクセスを付与する

Action Hero は、アプリケーションと一緒に実行して、アプリケーションが正しく機能するために必要な AWS API コールと対応する IAM アクセス許可を特定できるユーティリティです。これは、アプリケーションに割り当てられた IAM ロールの範囲を徐々に制限するのに役立つという点で、IAM Access Advisor に似ています。詳細については、AWS リソースへの最小特権アクセスの付与に関するドキュメントを参照してください。

IRSA および Pod ID で使用される IAM ロールに対するアクセス許可の境界を設定することを検討してください。アクセス許可の境界を使用して、IRSA または Pod ID で使用されるロールがアクセス許可の最大レベルを超えないようにすることができます。アクセス許可の境界ポリシーの例を使用してアクセス許可の境界を開始するためのガイドの例については、この github リポジトリを参照してください。

EKS クラスターへの不要な匿名アクセスを確認して取り消す

理想的には、すべての API アクションで匿名アクセスを無効にする必要があります。匿名アクセスは、Kubernetes 組み込みユーザーシステムの RoleBinding または ClusterRoleBinding:anonymous を作成することによって付与されます。rbac-lookup ツールを使用して、system:anonymous ユーザーがクラスターに対して持っているアクセス許可を特定できます。

```
./rbac-lookup | grep -P 'system:(anonymous)|(unauthenticated)'
system:anonymous               cluster-wide        ClusterRole/system:discovery
system:unauthenticated         cluster-wide        ClusterRole/system:discovery
system:unauthenticated         cluster-wide        ClusterRole/system:public-info-
viewer
```

system:public-info-viewer 以外のロールまたは ClusterRole は、system:anonymous user または system:unauthenticated グループにバインドしないでください。

特定の APIs で匿名アクセスを有効にするには、いくつかの正当な理由が考えられます。クラスターの場合、匿名ユーザーがこれらの特定の APIs のみにアクセスでき、認証なしでこれらの APIs を公開しても、クラスターが脆弱になることはありません。

Kubernetes/EKS バージョン 1.14 以前は、system:unauthenticated グループはデフォルトで system:discovery および system:basic-user ClusterRoles に関連付けられていました。クラスターをバージョン 1.14 以降に更新した場合でも、クラスターの更新によってこれらのアクセス許可が取り消されないため、これらのアクセス許可はクラスターで引き続き有効になる場合があります。system:public-info-viewer を除く「system:unauthenticated」を持つ ClusterRoles を確認するには、次のコマンドを実行します (jq util が必要です) 。

```
kubectl get ClusterRoleBinding -o json | jq -r '.items[] | select(.subjects[]?.name
 =="system:unauthenticated") | select(.metadata.name != "system:public-info-viewer")
 | .metadata.name'
```

また、「system:unauthenticated」は、以下を使用して「system:public-info-viewer」を除くすべてのロールから削除できます。

```
kubectl get ClusterRoleBinding -o json | jq -r '.items[] | select(.subjects[]?.name
 =="system:unauthenticated") | select(.metadata.name != "system:public-info-viewer") |
 del(.subjects[] | select(.name =="system:unauthenticated"))' | kubectl apply -f -
```

または、kubectl describe と kubectl edit を使用して手動でチェックおよび削除することもできます。system:unauthenticated グループにクラスターに対する system:discovery アクセス許可があるかどうかを確認するには、次のコマンドを実行します。

```
kubectl describe clusterrolebindings system:discovery

Name:         system:discovery
Labels:       kubernetes.io/bootstrapping=rbac-defaults
Annotations:  rbac.authorization.kubernetes.io/autoupdate: true
Role:
  Kind:  ClusterRole
  Name:  system:discovery
Subjects:
  Kind   Name                   Namespace
  ----   ----                   ---------
  Group  system:authenticated
  Group  system:unauthenticated
```

system:unauthenticated グループにクラスターに対する system:basic-user アクセス許可があるかどうかを確認するには、次のコマンドを実行します。

```
kubectl describe clusterrolebindings system:basic-user

Name:         system:basic-user
Labels:       kubernetes.io/bootstrapping=rbac-defaults
Annotations:  rbac.authorization.kubernetes.io/autoupdate: true
Role:
  Kind:  ClusterRole
  Name:  system:basic-user
Subjects:
```

```
  Kind   Name                     Namespace
  ----   ----                     ---------
  Group  system:authenticated
  Group  system:unauthenticated
```

system:unauthenticated グループがクラスターの system:discovery および/または system:basic-user ClusterRoles にバインドされている場合は、system:unauthenticated グループからこれらのロールの関連付けを解除する必要があります。次のコマンドを使用して、system:discovery ClusterRoleBinding を編集します。

```
kubectl edit clusterrolebindings system:discovery
```

上記のコマンドは、以下に示すように、エディタで system:discovery ClusterRoleBinding の現在の定義を開きます。

```
# Please edit the object below. Lines beginning with a '#' will be ignored,
# and an empty file will abort the edit. If an error occurs while saving this file will
 be
# reopened with the relevant failures.
#
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  annotations:
    rbac.authorization.kubernetes.io/autoupdate: "true"
  creationTimestamp: "2021-06-17T20:50:49Z"
  labels:
    kubernetes.io/bootstrapping: rbac-defaults
  name: system:discovery
  resourceVersion: "24502985"
  selfLink: /apis/rbac.authorization.k8s.io/v1/clusterrolebindings/system%3Adiscovery
  uid: b7936268-5043-431a-a0e1-171a423abeb6
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: system:discovery
subjects:
- apiGroup: rbac.authorization.k8s.io
  kind: Group
  name: system:authenticated
- apiGroup: rbac.authorization.k8s.io
  kind: Group
```

```
  name: system:unauthenticated
```

上記のエディタ画面の「件名」セクションから system:unauthenticated グループのエントリを削除します。

system:basic-user ClusterRoleBinding に対して同じ手順を繰り返します。

IRSA で AWS SDK セッションを再利用する

IRSA を使用する場合、AWS SDK を使用して記述されたアプリケーションは、ポッドに配信されたトークンを使用して を呼び出しsts:AssumeRoleWithWebIdentity、一時的な AWS 認証情報を生成します。これは、コンピューティングサービスが一時的な AWS 認証情報を Lambda 関数などの AWS コンピューティングリソースに直接配信する他の AWS コンピューティングサービスとは異なります。つまり、AWS SDK セッションが初期化されるたびに、 の AWS STS への呼び出しAssumeRoleWithWebIdentityが行われます。アプリケーションが急速にスケールし、多くの AWS SDK セッションを初期化する場合、コードが を何度も呼び出すため、AWS STS からのスロットリングが発生する可能性がありますAssumeRoleWithWebIdentity。

このシナリオを回避するには、 への不要な呼び出しが行われAssumeRoleWithWebIdentityないように、アプリケーション内で AWS SDK セッションを再利用することをお勧めします。

次のコード例では、boto3 Python SDK を使用してセッションが作成され、同じセッションを使用してクライアントを作成し、Amazon S3 と Amazon SQS の両方とやり取りします。 AssumeRoleWithWebIdentityは 1 回だけ呼び出され、AWS SDK は有効期限が切れmy_sessionると の認証情報を自動的に更新します。

```
import boto3

= Create your own session

my_session = boto3.session.Session()

= Now we can create low-level clients from our session

sqs = my_session.client(``sqs``) s3 = my_session.client(``s3``)

s3response = s3.list_buckets() sqsresponse = sqs.list_queues()

#print the response from the S3 and SQS APIs print("`s3 response:`")
print(s3response) print("`—`") print("`sqs response:`")
print(sqsresponse)
```

EC2 などの別の AWS コンピューティングサービスから IRSA を使用して EKS にアプリケーションを移行する場合、これは特に重要な詳細です。AWS SDK セッションを初期化する他のコンピューティングサービスでは、指示がない限り、AWS STS は呼び出されません。

代替アプローチ

IRSA および EKS Pod ID はポッドに AWS ID を割り当てるための推奨方法ですが、アプリケーションに最新バージョンの AWS SDKs を含める必要があります。現在 IRSA をサポートしている SDKs の完全なリストについては、https://docs.aws.amazon.com/eks/latest/userguide/iam-roles-for-service-accounts-minimum-sdk.html を参照してください。EKS Pod ID については、https://docs.aws.amazon.com/eks/latest/userguide/pod-id-minimum-sdk.html を参照してください。互換性のある SDK ですぐに更新できないアプリケーションがある場合は、kube2iam や kiam など、Kubernetes ポッドに IAM ロールを割り当てるためのコミュニティ構築のソリューションがいくつかあります。AWS はこれらのソリューションの使用を推奨、容認、サポートしていませんが、IRSA や EKS Pod ID と同様の結果を得るためにコミュニティ全体で頻繁に使用されています。

これらの aws 提供以外のソリューションの 1 つを使用する必要がある場合は、デューデリジェンスを行い、そのために生じるセキュリティ上の影響を理解していることを確認してください。

ツールとリソース

- Amazon EKS セキュリティイマージョンワークショップ - Identity and Access Management
- Terraform EKS 設計図パターン - 完全プライベート Amazon EKS クラスター
- Terraform EKS ブループリントパターン - Amazon EKS クラスターの IAM アイデンティティセンターシングルサインオン
- Terraform EKS ブループリントパターン - Amazon EKS クラスター用の Okta シングルサインオン
- audit2rbac
- rbac.dev Kubernetes RBAC のブログやツールを含む追加リソースのリスト
- アクションのヒーロー
- kube2iam
- キアム

ポッドのセキュリティ

ポッド仕様には、全体的なセキュリティ体制を強化または弱める可能性のあるさまざまな属性が含まれています。Kubernetes の実務者として、コンテナで実行されているプロセスがコンテナランタイ

ムの分離境界からエスケープし、基盤となるホストにアクセスできないようにすることが主な懸念事項です。

Linux の機能

コンテナ内で実行されるプロセスは、デフォルトで [Linux] ルートユーザーのコンテキストで実行されます。コンテナ内のルートのアクションは、コンテナランタイムがコンテナに割り当てる一連の Linux 機能によって部分的に制約されますが、これらのデフォルトの権限により、攻撃者は権限をエスカレートしたり、Secrets や ConfigMaps などのホストにバインドされた機密情報にアクセスしたりできます。以下は、コンテナに割り当てられたデフォルトの機能のリストです。各機能の詳細については、http://man7.org/linux/man-pages/man7/capabilities.7.html を参照してください。

```
CAP_AUDIT_WRITE, CAP_CHOWN, CAP_DAC_OVERRIDE, CAP_FOWNER, CAP_FSETID,
CAP_KILL, CAP_MKNOD, CAP_NET_BIND_SERVICE, CAP_NET_RAW, CAP_SETGID,
CAP_SETUID, CAP_SETFCAP, CAP_SETPCAP, CAP_SYS_CHROOT
```

Example

EC2 ポッドと Fargate ポッドには、デフォルトで前述の機能が割り当てられます。さらに、Linux 機能は Fargate ポッドからのみ削除できます。

特権として実行されるポッドは、ホスト上のルートに関連付けられたすべての Linux 機能を継承します。これは、可能であれば避けてください。

ノード認可

すべての Kubernetes ワーカーノードは、ノード認可と呼ばれる認可モードを使用します。Node Authorization は、kubelet から送信されるすべての API リクエストを承認し、ノードが次のアクションを実行できるようにします。

読み取りオペレーション:

- サービス
- エンドポイント
- ノード
- ポッド
- kubelet のノードにバインドされたポッドに関連するシークレット、設定マップ、永続ボリュームクレーム、永続ボリューム

書き込みオペレーション:

- ノードとノードステータス (アドNodeRestrictionミッションプラグインを有効にして kubelet を制限し、独自のノードを変更)
- ポッドとポッドステータス (アドNodeRestrictionミッションプラグインを有効にして kubelet を制限し、それ自体にバインドされたポッドを変更)
- イベント

認証関連のオペレーション:

- TLS ブートストラップ用の CertificateSigningRequest (CSR) API への読み取り/書き込みアクセス
- 委任認証/認可チェックのために TokenReview と SubjectAccessReview を作成する機能

EKS はノード制限アドミッションコントローラーを使用します。これにより、ノードはノードにバインドされているノード属性とポッドオブジェクトの限定されたセットのみ変更できます。ただし、ホストへのアクセスを管理する攻撃者は、Kubernetes API から環境に関する機密情報を収集できるため、クラスター内で横方向に移動できます。

ポッドセキュリティソリューション

ポッドセキュリティポリシー (PSP)

以前は、ポッドセキュリティポリシー (PSP) リソースを使用して、ポッドを作成する前に満たす必要がある一連の要件を指定していました。Kubernetes バージョン 1.21 以降、PSP は廃止されました。これらは Kubernetes バージョン 1.25 で削除される予定です。

> **Important**
>
> PSPs は Kubernetes バージョン 1.21 では廃止されました。バージョン 1.25 まで、または代替 への移行には約 2 年かかります。このドキュメントでは、この廃止の動機について説明します。

新しいポッドセキュリティソリューションへの移行

Kubernetes v1.25 の時点で PSPs が削除されているため、クラスター管理者とオペレーターはこれらのセキュリティコントロールを置き換える必要があります。このニーズを満たすには、次の 2 つのソリューションがあります。

- Kubernetes エコシステムからの Policy-as-code (PAC) ソリューション
- Kubernetes Pod セキュリティ標準 (PSS)

PAC と PSS の両方のソリューションは PSP と共存できます。PSP を削除する前にクラスターで使用できます。これにより、PSP から移行する際の導入が容易になります。PSP から PSS への移行を検討する際は、このドキュメントを参照してください。

以下に概説する PAC ソリューションの 1 つである Kyverno には、類似ポリシー、機能比較、移行手順など、PSPsからソリューションに移行する際の具体的なガイダンスがブログ記事で説明されています。Pod Security Admission (PSA) に関する Kyverno への移行に関する追加情報とガイダンスは、AWS ブログで公開されています。

Policy-as-code (PAC)

Policy-as-code (PAC) ソリューションは、クラスターユーザーをガイドし、規定の自動化されたコントロールを通じて望ましくない動作を防ぐガードレールを提供します。PAC は Kubernetes Dynamic Admission Controller を使用して、ウェブフック呼び出しを介して Kubernetes API サーバーリクエストフローを傍受し、コードとして記述および保存されたポリシーに基づいてリクエストペイロードを変更および検証します。ミューテーションと検証は、API サーバーリクエストによってクラスターが変更される前に行われます。PAC ソリューションは、ポリシーを使用して、分類と値に基づいて API サーバーリクエストペイロードを照合して処理します。

Kubernetes で使用できるオープンソースの PAC ソリューションがいくつかあります。これらのソリューションは Kubernetes プロジェクトの一部ではなく、Kubernetes エコシステムから調達されています。PAC ソリューションの一部を以下に示します。

- OPA/ゲートキーパー
- オープンポリシーエージェント (OPA)
- キバーノ
- クベワーデン
- jsPolicy

PAC ソリューションの詳細と、ニーズに合わせて適切なソリューションを選択する方法については、以下のリンクを参照してください。

- Kubernetes のポリシーベースの対策 – パート 1
- Kubernetes のポリシーベースの対策 – パート 2

Pod Security Standards (PSS) と Pod Security Admission (PSA)

PSP の非推奨と、組み込みの Kubernetes ソリューションを使用してポッドセキュリティout-of-the-box制御する継続的な必要性に応じて、Kubernetes Auth Special Interest Group は Pod Security Standards (PSS) と Pod Security Admission (PSA) を作成しました。PSA エフォートには、PSS で定義されたコントロールを実装するアドミッションコントローラーウェブフックプロジェクトが含まれます。このアドミッションコントローラーのアプローチは、PAC ソリューションで使用された に似ています。

Kubernetes ドキュメントによると、PSS は「」3 つの異なるポリシーを定義して、セキュリティ範囲を広くカバーしています。これらのポリシーは累積的であり、高許可から高制限まで多岐にわたります。」

これらのポリシーは次のように定義されます。

- 特権: 無制限 (安全でない) ポリシー。可能な限り幅広いレベルのアクセス許可を提供します。このポリシーは、既知の特権のエスカレーションを許可します。これはポリシーがないことです。これは、ログ記録エージェント、CNIs、ストレージドライバー、および特権アクセスを必要とするその他のシステム全体のアプリケーションに適しています。
- ベースライン: 既知の特権のエスカレーションを防止する最小限の制限ポリシー。デフォルトの (最小限に指定された) Pod 設定を許可します。ベースラインポリシーでは、hostNetwork、hostPID、hostIPC、hostPath、hostPort、Linux 機能を追加できないこと、および他のいくつかの制限の使用を禁止しています。
- 制限付き: 現在の Pod 強化のベストプラクティスに従って、厳しく制限されたポリシー。このポリシーはベースラインから継承され、ルートまたはルートグループとして実行できないなどの制限を追加します。制限付きポリシーは、アプリケーションの機能に影響を与える可能性があります。これらは主にセキュリティクリティカルなアプリケーションの実行を対象としています。

これらのポリシーは、ポッド実行のプロファイルを定義し、3 つのレベルの特権アクセスと 制限付きアクセスに分かれています。

PSS で定義されたコントロールを実装するために、PSA は 3 つのモードで動作します。

- enforce: ポリシー違反により、ポッドが拒否されます。
- 監査: ポリシー違反は、監査ログに記録されたイベントへの監査注釈の追加をトリガーしますが、それ以外の場合は許可されます。
- 警告: ポリシー違反はユーザー向けの警告をトリガーしますが、それ以外の場合は許可されます。

これらのモードとプロファイル (制限) レベルは、次の例に示すように、ラベルを使用して Kubernetes 名前空間レベルで設定されます。

```
apiVersion: v1
kind: Namespace
metadata:
  name: policy-test
  labels:
    pod-security.kubernetes.io/enforce: restricted
```

これらのオペレーションモードを個別に使用すると、レスポンスが異なるため、ユーザーエクスペリエンスも異なります。適用モードでは、それぞれの podSpecs が設定された制限レベルに違反した場合、ポッドが作成されなくなります。ただし、このモードでは、Deployments などのポッドを作成するポッド以外の Kubernetes オブジェクトは、podSpecックが適用された PSS に違反していても、クラスターに適用されないようにします。この場合、デプロイが適用されますが、ポッド (複数可) は適用されません。

これは難しいユーザーエクスペリエンスです。正常に適用されたデプロイオブジェクトがポッドの作成に失敗したとすぐには判断されないためです。問題のある podSpecs はポッドを作成しません。を使用してデプロイリソースを検査`kubectl get deploy <DEPLOYMENT_NAME> -oyaml`すると、以下に示すように、失敗したポッド (複数可) `.status.conditions`要素からのメッセージが公開されます。

```
...
status:
  conditions:
    - lastTransitionTime: "2022-01-20T01:02:08Z"
      lastUpdateTime: "2022-01-20T01:02:08Z"
      message: 'pods "test-688f68dc87-tw587" is forbidden: violates PodSecurity
 "restricted:latest":
        allowPrivilegeEscalation != false (container "test" must set
 securityContext.allowPrivilegeEscalation=false),
        unrestricted capabilities (container "test" must set
 securityContext.capabilities.drop=["ALL"]),
        runAsNonRoot != true (pod or container "test" must set
 securityContext.runAsNonRoot=true),
        seccompProfile (pod or container "test" must set
 securityContext.seccompProfile.type
        to "RuntimeDefault" or "Localhost")'
      reason: FailedCreate
      status: "True"
```

```
      type: ReplicaFailure
...
```

監査モードと警告モードの両方で、ポッドの制限により、違反しているポッドの作成と起動が妨げられることはありません。ただし、これらのモードでは、API サーバーの監査ログイベントに対する監査注釈と、kubectl などの API サーバークライアントへの警告がそれぞれトリガーされます。ポッドとポッドを作成するオブジェクトに、違反のある podSpecsが含まれている場合です。kubectl 警告メッセージが以下に表示されます。

```
Warning: would violate PodSecurity "restricted:latest":
 allowPrivilegeEscalation != false (container "test" must set
 securityContext.allowPrivilegeEscalation=false), unrestricted capabilities (container
 "test" must set securityContext.capabilities.drop=["ALL"]), runAsNonRoot != true (pod
 or container "test" must set securityContext.runAsNonRoot=true), seccompProfile (pod
 or container "test" must set securityContext.seccompProfile.type to "RuntimeDefault"
 or "Localhost")
deployment.apps/test created
```

PSA 監査モードと警告モードは、クラスターオペレーションに悪影響を及ぼすことなく PSS を導入する場合に役立ちます。

PSA オペレーションモードは相互に排他的ではなく、累積的に使用できます。以下に示すように、複数のモードを 1 つの名前空間に設定できます。

```
apiVersion: v1
kind: Namespace
metadata:
  name: policy-test
  labels:
    pod-security.kubernetes.io/audit: restricted
    pod-security.kubernetes.io/enforce: restricted
    pod-security.kubernetes.io/warn: restricted
```

上記の例では、デプロイの適用時にユーザーフレンドリーな警告と監査注釈が提供され、違反の強制もポッドレベルで提供されます。実際、以下に示すように、複数の PSA ラベルで異なるプロファイルレベルを使用できます。

```
apiVersion: v1
kind: Namespace
metadata:
  name: policy-test
```

```
  labels:
    pod-security.kubernetes.io/enforce: baseline
    pod-security.kubernetes.io/warn: restricted
```

上記の例では、ベースラインプロファイルレベルを満たすすべてのポッドの作成を許可し、制限されたプロファイルレベルに違反するポッド (およびポッドを作成するオブジェクト) に警告するように PSA が設定されています。これは、ベースラインから制限付きプロファイルに変更した場合に考えられる影響を判断するのに役立つアプローチです。

既存のポッド

既存のポッドの名前空間が、より制限の厳しい PSS プロファイルを使用するように変更された場合、監査モードと警告モードは適切なメッセージを生成しますが、強制モードはポッドを削除しません。警告メッセージを以下に示します。

```
Warning: existing pods in namespace "policy-test" violate the new PodSecurity enforce
 level "restricted:latest"
Warning: test-688f68dc87-htm8x: allowPrivilegeEscalation != false, unrestricted
 capabilities, runAsNonRoot != true, seccompProfile
namespace/policy-test configured
```

免除

PSA は免除を使用して、適用されていたポッドに対する違反の適用を除外します。これらの免除事項を以下に示します。

- ユーザー名: 認証された (または偽装された) ユーザー名が除外されたユーザーからのリクエストは無視されます。
- RuntimeClassNames: 除外ランタイムクラス名を指定するポッドとワークロードリソースは無視されます。
- 名前空間: 除外名前空間のポッドとワークロードリソースは無視されます。

これらの免除は、API サーバー設定の一部として PSA アドミッションコントローラー設定で静的に適用されます。

検証ウェブフックの実装では、pod-security-webhookコンテナにボリュームとしてマウントされる Kubernetes ConfigMap リソース内で免除を設定できます。

```
apiVersion: v1
```

```
kind: ConfigMap
metadata:
  name: pod-security-webhook
  namespace: pod-security-webhook
data:
  podsecurityconfiguration.yaml: |
    apiVersion: pod-security.admission.config.k8s.io/v1
    kind: PodSecurityConfiguration
    defaults:
      enforce: "restricted"
      enforce-version: "latest"
      audit: "restricted"
      audit-version: "latest"
      warn: "restricted"
      warn-version: "latest"
    exemptions:
      # Array of authenticated usernames to exempt.
      usernames: []
      # Array of runtime class names to exempt.
      runtimeClasses: []
      # Array of namespaces to exempt.
      namespaces: ["kube-system","policy-test1"]
```

上記の ConfigMap YAML に示すように、クラスター全体のデフォルトの PSS レベルは、すべての PSA モード、監査、強制、および警告に対して制限されています。これは、除外されたものを除くすべての名前空間に影響します: `namespaces: ["kube-system","policy-test1"]`。さらに、以下に示す ValidatingWebhookConfiguration リソースでは、pod-security-webhook 名前空間も設定された PSS から除外されます。

```
...
webhooks:
  # Audit annotations will be prefixed with this name
  - name: "pod-security-webhook.kubernetes.io"
    # Fail-closed admission webhooks can present operational challenges.
    # You may want to consider using a failure policy of Ignore, but should
    # consider the security tradeoffs.
    failurePolicy: Fail
    namespaceSelector:
      # Exempt the webhook itself to avoid a circular dependency.
      matchExpressions:
        - key: kubernetes.io/metadata.name
          operator: NotIn
          values: ["pod-security-webhook"]
```

```
...
```

> **Important**
>
> Pod Security Admissions は、Kubernetes v1.25 で安定するように卒業しました。デフォルトでは有効にする前に Pod Security Admission 機能を使用する場合は、動的アドミッションコントローラー (ウェブフックの変更) をインストールする必要があります。ウェブフックのインストールと設定の手順については、こちらを参照してください。

policy-as-code と Pod セキュリティ標準の選択

Pod Security Standards (PSS) は、Kubernetes に組み込み、Kubernetes エコシステムからのソリューションを必要としないソリューションを提供することで、Pod Security Policy (PSP) を置き換えるために開発されました。つまり、コードpolicy-as-code (PAC) ソリューションははるかに柔軟です。

以下の Pros and Cons のリストは、ポッドセキュリティソリューションについてより情報に基づいた意思決定を行うのに役立ちます。

Policy-as-code (ポッドセキュリティ標準との比較)

メリット:

- より柔軟で詳細 (必要に応じてリソースの属性まで)
- ポッドだけでなく、さまざまなリソースやアクションに対して使用できます。
- 名前空間レベルで適用されるだけでなく
- Pod セキュリティ標準よりも成熟している
- 決定は、API サーバーリクエストペイロード内の任意のもの、および既存のクラスターリソースと外部データ (ソリューション依存) に基づいて行うことができます。
- 検証前の API サーバーリクエストの変更をサポート (ソリューション依存)
- 補完ポリシーと Kubernetes リソースを生成できます (ソリューション依存 - ポッドポリシーから、Kyverno はデプロイなどの上位レベルのコントローラーのポリシーを自動生成できます。Kyverno は、ルールの生成を使用して、「新しいリソースの作成時またはソースの更新時」に追加の Kubernetes リソースを生成することもできます。)
- Kubernetes API サーバーを呼び出す前に、左から CICD パイプラインにシフトするために使用できます (ソリューション依存)

- ベストプラクティス、組織標準など、セキュリティに必ずしも関連しない動作を実装するために使用できます。
- Kubernetes 以外のユースケースで使用可能 (ソリューション依存)
- 柔軟性があるため、ユーザーエクスペリエンスはユーザーのニーズに合わせて調整できます。

デメリット:

- Kubernetes に組み込まれていない
- より複雑な学習、設定、サポート
- ポリシーの作成には、新しいskills/languages/capabilitiesが必要になる場合があります

Pod Security Admission (policy-as-code と比較)

メリット:

- Kubernetes に組み込まれている
- 設定が簡単
- または ポリシーを使用して作成する新しい言語がない
- クラスターのデフォルトのアドミッションレベルが特権レベルに設定されている場合、名前空間ラベルを使用して名前空間をポッドセキュリティプロファイルにオプトインできます。

デメリット:

- policy-as-codeほど柔軟でも詳細でもない
- 3 つのレベルの制限のみ
- 主にポッドに重点を置く

概要

現在、PSP 以外のポッドセキュリティソリューションがなく、必要なポッドセキュリティ体制が Pod Security Standards (PSS) で定義されているモデルに適合している場合、policy-as-codeソリューションの代わりに PSS を採用するのが簡単な方法かもしれません。ただし、ポッドのセキュリティ体制が PSS モデルに適合しない場合、または PSS で定義されたコントロールを超えて追加のコントロールを追加することを想定している場合、policy-as-codeソリューションの方が適しているように見えます。

推奨事項

複数の Pod Security Admission (PSA) モードを使用してユーザーエクスペリエンスを向上させる

前述のように、PSA 強制モードでは、PSS 違反のあるポッドは適用されませんが、デプロイなどの上位レベルのコントローラーは停止しません。実際、デプロイはポッドの適用に失敗したことを示すことなく正常に適用されます。kubectl を使用してデプロイオブジェクトを検査し、PSA から失敗したポッドメッセージを検出できますが、ユーザーエクスペリエンスは向上する可能性があります。ユーザーエクスペリエンスを向上させるには、複数の PSA モード (監査、強制、警告) を使用する必要があります。

```
apiVersion: v1
kind: Namespace
metadata:
  name: policy-test
  labels:
    pod-security.kubernetes.io/audit: restricted
    pod-security.kubernetes.io/enforce: restricted
    pod-security.kubernetes.io/warn: restricted
```

上記の例では、強制モードが定義されているため、それぞれの podSpec で PSS 違反のあるデプロイマニフェストを Kubernetes API サーバーに適用しようとすると、デプロイは正常に適用されますが、ポッドは適用されません。また、監査モードと警告モードも有効であるため、API サーバークライアントは警告メッセージを受信し、API サーバーの監査ログイベントにもメッセージで注釈が付けられます。

特権として実行できるコンテナを制限する

前述のように、特権として実行されるコンテナは、ホストのルートに割り当てられたすべての Linux 機能を継承します。コンテナが適切に機能するために、これらのタイプの権限が必要になることはほとんどありません。コンテナのアクセス許可と機能を制限するために使用できる方法は複数あります。

> **Important**
>
> Fargate は、ポッドのコンテナが AWS が管理するインフラストラクチャで実行される「サーバーレス」コンテナ (複数可) を実行できるようにする起動タイプです。Fargate で

は、特権コンテナを実行したり、hostNetwork または hostPort を使用するようにポッドを設定したりすることはできません。

コンテナ内のプロセスをルートとして実行しない

デフォルトでは、すべてのコンテナは root として実行されます。これは、攻撃者がアプリケーションの脆弱性を悪用し、実行中のコンテナへのシェルアクセスを取得できる場合に問題になる可能性があります。このリスクを軽減するには、さまざまな方法があります。まず、コンテナイメージからシェルを削除します。次に、USER ディレクティブを Dockerfile に追加するか、ポッド内のコンテナを非ルートユーザーとして実行します。Kubernetes podSpec には、アプリケーションを実行するユーザーやグループを指定できる`spec.securityContext`、 の一連のフィールドが含まれています。これらのフィールド`runAsGroup`はそれぞれ `runAsUser`および です。

Kubernetes podSpec 内で `spec.securityContext`とその関連要素の使用を強制するために、policy-as-code または Pod セキュリティ標準をクラスターに追加できます。これらのソリューションを使用すると、インバウンド Kubernetes API サーバーリクエストペイロードを etcd に保持する前に検証できるポリシーまたはプロファイルを記述および/または使用できます。さらに、policy-as-code ソリューションは、インバウンドリクエストを変更したり、場合によっては新しいリクエストを生成したりできます。

Docker で Docker を実行したり、コンテナにソケットをマウントしたりしないでください。

これにより、Docker コンテナでイメージを構築/実行できますが、基本的には、コンテナで実行されているプロセスにノードの完全な制御を放棄します。Kubernetes でコンテナイメージを構築する必要がある場合は、代わりに Kaniko、buildah、または CodeBuild などのビルドサービスを使用します。

> **Note**
>
> コンテナイメージの構築など、CICD 処理に使用される Kubernetes クラスターは、より一般的なワークロードを実行しているクラスターから分離する必要があります。

hostPath の使用を制限するか、hostPath が必要な場合は、使用できるプレフィックスを制限し、ボリュームを読み取り専用として設定します。

hostPath は、ホストからコンテナに直接ディレクトリをマウントするボリュームです。ポッドにこのタイプのアクセスが必要になることはまれですが、必要になった場合はリスクに注意する必要があります。デフォルトでは、root として実行されるポッドは hostPath によって公開されるファイルシステムへの書き込みアクセス権を持ちます。これにより、攻撃者は kubelet 設定を変更したり、/etc /shadow などの hostPath によって直接公開されていないディレクトリやファイルへのシンボリックリンクを作成したり、ssh キーをインストールしたり、ホストにマウントされたシークレットを読み取ったり、その他の悪意のあるものを行ったりすることができます。hostPath によるリスクを軽減するには、 `readOnly`を `spec.containers.volumeMounts`として設定します。次に例を示します。

```
volumeMounts:
- name: hostPath-volume
    readOnly: true
    mountPath: /host-path
```

また、policy-as-code ソリューションを使用して、`hostPath`ボリュームで使用できるディレクトリを制限したり、`hostPath`使用を完全に防止したりする必要があります。Pod Security Standards Baseline または制限付きポリシーを使用して、 の使用を防ぐことができます`hostPath`。

特権エスカレーションの危険性の詳細については、Seth Art のブログ「Bad Pods: Kubernetes Pod Privilege Escalation」を参照してください。

リソースの競合や DoS 攻撃を避けるために、コンテナごとにリクエストと制限を設定する

リクエストや制限のないポッドは、理論的にはホストで使用できるすべてのリソースを消費できます。追加のポッドがノードにスケジュールされると、ノードに CPU またはメモリの負荷が発生し、Kubelet がノードからポッドを終了または削除する可能性があります。これらがすべて同時に発生するのを防ぐことはできませんが、リクエストと制限を設定すると、リソースの競合を最小限に抑え、大量のリソースを消費する記述が不十分なアプリケーションによるリスクを軽減できます。

`podSpec` では、CPU とメモリのリクエストと制限を指定できます。CPU はオーバーサブスクライブされる可能性があるため、圧縮可能なリソースと見なされます。メモリは非圧縮 です。つまり、複数のコンテナ間で共有することはできません。

CPU またはメモリのリクエストを指定する場合、基本的にコンテナが取得するメモリの量を指定します。Kubernetes は、ポッド内のすべてのコンテナのリクエストを集約して、ポッドをスケジュールするノードを決定します。コンテナが要求されたメモリ量を超えると、ノードにメモリ負荷がある場合に終了する可能性があります。

制限は、コンテナが消費できる CPU およびメモリリソースの最大量であり、コンテナ用に作成された cgroup `memory.limit_in_bytes`の値に直接対応します。メモリ制限を超えるコンテナは OOM が強制終了します。コンテナが CPU 制限を超えると、スロットリングされます。

> **Note**
>
> コンテナを使用する場合は`resources.limits`、コンテナリソースの使用 (別名: リソースフットプリント) が負荷テストに基づいてデータ駆動型で正確であることを強くお勧めします。正確で信頼できるリソースフットプリントがなければ、コンテナをパディング`resources.limits`できます。例えば、 は、潜在的なメモリリソース制限の不正確性を考慮して、観測可能な最大値より 20～30% 高くパディング`resources.limits.memory`できます。

Kubernetes は 3 つの Quality of Service (QoS) クラスを使用して、ノードで実行されているワークロードに優先順位を付けます。具体的には次のとおりです。

- 保証
- バースト可能
- ベストエフォート

制限とリクエストが設定されていない場合、ポッドはベストエフォート (最低優先度) として設定されます。ベストエフォートポッドは、メモリ不足時に最初に強制終了されます。ポッド内のすべてのコンテナに制限が設定されている場合、またはリクエストと制限が同じ値に設定され、0 に等しくない場合、ポッドは保証 (最高優先度) として設定されます。保証されたポッドは、設定されたメモリ制限を超えない限り強制終了されません。制限とリクエストが異なる値で 0 に等しくない、またはポッド内の 1 つのコンテナが制限を設定し、他のコンテナが異なるリソースに対して制限を設定していない場合、ポッドはバースト可能 (中優先度) として設定されます。これらのポッドにはいくつかのリソース保証がありますが、リクエストされたメモリを超えたら強制終了できます。

> **Important**
>
> リクエストはコンテナの cgroup `memory_limit_in_bytes`の値には影響しません。cgroup の制限はホストで使用可能なメモリ量に設定されます。それでも、リクエスト値の設定が低すぎると、ノードがメモリ負荷を受けると、ポッドが kubelet による終了の対象となる可能性があります。

Class	優先度	Condition	強制終了条件
保証付き	highest	制限 = リクエスト != 0	メモリ制限のみを超える
バースト可能	medium	limit != リクエスト != 0	リクエストメモリを超えた場合、強制終了可能
ベストエフォート	lowest	制限とリクエストが設定されていません	メモリ不足時に最初に強制終了する

リソース QoS の詳細については、Kubernetes ドキュメントを参照してください。

リクエストと制限の使用を強制するには、名前空間にリソースクォータを設定するか、制限範囲を作成します。リソースクォータを使用すると、名前空間に割り当てられた CPU や RAM などのリソースの合計量を指定できます。名前空間に適用すると、その名前空間にデプロイされたすべてのコンテナのリクエストと制限を強制的に指定します。対照的に、制限範囲を使用すると、リソースの割り当てをより詳細に制御できます。制限範囲を使用すると、ポッドごと、または名前空間内のコンテナごとに CPU リソースとメモリリソースの最小値/最大値を指定できます。何も指定されていない場合は、これらを使用してデフォルトのリクエスト/制限値を設定することもできます。

Policy-as-code ソリューションを使用すると、リクエストと制限を適用できます。 または を使用して、名前空間の作成時にリソースクォータと制限範囲を作成することもできます。

特権エスカレーションを許可しない

特権エスカレーションでは、プロセスが実行されているセキュリティコンテキストを変更できます。Sudo は、SUID または SGID ビットを持つバイナリであるため、この良い

例です。特権エスカレーションは基本的に、ユーザーが別のユーザーまたはグループのアクセス許可を持つファイルを実行する方法です。を に設定する policy-as-code 変更ポリシーを実装するか、 securityContext.allowPrivilegeEscalationで を設定allowPrivilegeEscalationfalseすることで、コンテナが特権エスカレーションを使用することを防ぐことができますpodSpec。Policy-as-code ポリシーを使用して、誤った設定が検出された場合に API サーバーリクエストが成功しないようにすることもできます。ポッドセキュリティ標準を使用して、ポッドが特権エスカレーションを使用できないようにすることもできます。

ServiceAccount トークンマウントを無効にする

Kubernetes API にアクセスする必要がないポッドの場合、ポッド仕様、または特定の ServiceAccount を使用するすべてのポッドで ServiceAccount トークンの自動マウントを無効にすることができます。

Example

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-no-automount
spec:
  automountServiceAccountToken: false
```

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: sa-no-automount
automountServiceAccountToken: false
```

サービス検出を無効にする

クラスター内サービスを検索または呼び出す必要がないポッドの場合、ポッドに提供される情報の量を減らすことができます。CoreDNS を使用しないCoreDNSポリシーを設定し、ポッドの名前空間内のサービスを環境変数として公開しないように設定できます。サービスリンクの詳細については、環境変数に関する Kubernetes ドキュメントを参照してください。ポッドの DNS ポリシーのデフォルト値は、クラスター内 DNS を使用するClusterFirst」で、デフォルト値以外の「Default」は基盤となるノードの DNS 解決を使用します。詳細については、ポッド DNS ポリシーの Kubernetes ドキュメントを参照してください。

Example

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-no-service-info
spec:
    dnsPolicy: Default # "Default" is not the true default value
    enableServiceLinks: false
```

読み取り専用ルートファイルシステムでイメージを設定する

読み取り専用のルートファイルシステムでイメージを設定すると、攻撃者はアプリケーションが使用するファイルシステムでバイナリを上書きできなくなります。アプリケーションがファイルシステムに書き込む必要がある場合は、一時ディレクトリに書き込むか、ボリュームをアタッチしてマウントすることを検討してください。これを強制するには、次のようにポッドの SecurityContext を設定します。

```
...
securityContext:
  readOnlyRootFilesystem: true
...
```

Policy-as-code および Pod セキュリティ標準を使用して、この動作を適用できます。

Example

Kubernetes の Windows コンテナは、Windows で実行されているコンテナtrueに対して に設定securityContext.readOnlyRootFilesystemすることはできません。これは、レジストリとシステムプロセスがコンテナ内で実行するために書き込みアクセスが必要なためです。

ツールとリソース

- Amazon EKS セキュリティイマージョンワークショップ - ポッドセキュリティ
- open-policy-agent/gatekeeper-library: OPA Gatekeeper ポリシーライブラリは、PSPs。
- Kyverno ポリシーライブラリ
- EKS の一般的な OPA および Kyverno ポリシーのコレクション。
- ポリシーベースの対策: パート 1

- ポリシーベースの対策: パート 2
- Pod Security Policy Migrator PSPs を OPA/Gatekeeper、KubeWarden、または Kyverno ポリシーに変換するツール
- SUSE オープンソースのゼロトラストコンテナセキュリティプラットフォームによる NeuVector は、プロセスポリシーとファイルシステムポリシー、アドミッションコントロールルールを提供します。

テナントの分離

マルチテナンシーを考えた場合、多くの場合、共有インフラストラクチャで実行されている他のユーザーやアプリケーションからユーザーやアプリケーションを分離する必要があります。

Kubernetes は単一テナントオーケストレーターです。つまり、コントロールプレーンの 単一インスタンスがクラスター内のすべてのテナント間で共有されます。ただし、マルチテナンシーのアセンブリを作成するために使用できるさまざまな Kubernetes オブジェクトがあります。例えば、名前空間とロールベースのアクセスコントロール (RBAC) を実装して、テナントを相互に論理的に分離できます。同様に、クォータと制限範囲を使用して、各テナントが消費できるクラスターリソースの量を制御できます。ただし、クラスターは強力なセキュリティ境界を提供する唯一のコンストラクトです。これは、 を管理する攻撃者が、そのホストにマウントされたすべてのシークレット、ConfigMapsボリュームを取得できるためです。また、Kubelet を偽装して、ノードの属性を操作したり、クラスター内で横方向に移動したりすることもできます。

以下のセクションでは、Kubernetes などの単一テナントオーケストレーターを使用するリスクを軽減しながら、テナント分離を実装する方法について説明します。

ソフトマルチテナンシー

ソフトマルチテナンシーでは、 名前空間、ロールとロールのバインディング、ネットワークポリシーなどのネイティブ Kubernetes コンストラクトを使用して、テナント間で論理的な分離を作成します。例えば、RBAC はテナントが互いのリソースにアクセスまたは操作できないようにすることができます。クォータと制限範囲は、各テナントが消費できるクラスターリソースの量を制御しますが、ネットワークポリシーは、異なる名前空間にデプロイされたアプリケーションが相互に通信するのを防ぐのに役立ちます。

ただし、これらのコントロールのいずれも、異なるテナントのポッドがノードを共有できないようにします。より強力な分離が必要な場合は、ノードセレクタ、アンチアフィニティルール、テイントや許容値を使用して、異なるテナントのポッドを別々のノードに強制的にスケジュールできます。これ

は、多くの場合、唯一のテナントノードと呼ばれます。テナントが多い環境では、これはかなり複雑でコストがかかります。

> **Important**
>
> 名前空間で実装されたソフトマルチテナンシーでは、名前空間がグローバルスコープ型であるため、フィルタリングされた名前空間のリストをテナントに提供できません。テナントが特定の名前空間を表示できる場合は、クラスター内のすべての名前空間を表示できます。

> **Warning**
>
> soft-multi-tenancyでは、テナントはデフォルトでクラスター内で実行されるすべてのサービスの CoreDNS をクエリできます。攻撃者は、クラスター内の任意のポッド `..svc.cluster.local`から dig SRV を実行することで、これを悪用する可能性があります。クラスター内で実行されるサービスの DNS レコードへのアクセスを制限する必要がある場合は、 CoreDNS のファイアウォールまたはポリシープラグインの使用を検討してください。詳細については、https://github.com/coredns/policy#kubernetes-metadata-multi-tenancy-policy を参照してください。

Kiosk は、ソフトマルチテナンシーの実装に役立つオープンソースプロジェクトです。これは、以下の機能を提供する一連の CRDsとコントローラーとして実装されています。

- 共有 Kubernetes クラスター内のテナントを分離するアカウントとアカウントユーザー
- アカウントユーザーのセルフサービス名前空間プロビジョニング
- クラスターを共有するときにサービスの品質と公平性を確保するためのアカウント制限
- 安全なテナント分離とセルフサービス名前空間の初期化のための名前空間テンプレート

Loft は Kiosk と DevSpace の保守者からの商用サービスで、次の機能を追加します。

- 異なるクラスター内のスペースへのアクセスを許可するためのマルチクラスターアクセス
- スリープモードは、非アクティブ期間中にスペース内のデプロイをスケールダウンします。
- GitHub などの OIDC 認証プロバイダーによるシングルサインオン

ソフトマルチテナンシーで対処できる主なユースケースは 3 つあります。

エンタープライズ設定

1 つ目は、「テナント」が従業員、請負業者、または組織によって承認されているという点で半信頼されているエンタープライズ設定です。各テナントは通常、部門やチームなどの管理部門と連携します。

このタイプの設定では、通常、クラスター管理者が名前空間の作成とポリシーの管理を担当します。また、特定の個人が名前空間を監督される委任管理モデルを実装し、デプロイ、サービス、ポッド、ジョブなど、ポリシーに関連しないオブジェクトに対して CRUD オペレーションを実行できるようにする場合もあります。

コンテナランタイムによって提供される分離は、この設定内で許容される場合もあれば、ポッドセキュリティの追加のコントロールで拡張する必要がある場合もあります。また、より厳密な分離が必要な場合は、異なる名前空間内のサービス間の通信を制限する必要がある場合もあります。

Kubernetes as a Service

対照的に、Kubernetes as a Service (KaaS) を提供する設定では、ソフトマルチテナンシーを使用できます。KaaS では、アプリケーションは一連の PaaS サービスを提供するコントローラーと CRDs のコレクションとともに共有クラスターでホストされます。テナントは Kubernetes API サーバーと直接やり取りし、ポリシー以外のオブジェクトに対して CRUD オペレーションを実行できます。には、テナントが独自の名前空間を作成および管理できるセルフサービスの要素もあります。このタイプの環境では、テナントは信頼できないコードを実行していると見なされます。

このタイプの環境でテナントを分離するには、多くの場合、厳密なネットワークポリシーとポッドサンドボックスを実装する必要があります。サンドボクシングは、Firecracker やユーザースペースカーネルなどのマイクロ VM 内でポッドのコンテナを実行する場所です。現在、EKS Fargate を使用してサンドボックスポッドを作成できます。

Software as a Service (SaaS)

ソフトマルチテナンシーの最終的なユースケースは、Software-as-a-Service (SaaS) 設定です。この環境では、各テナントはクラスター内で実行されているアプリケーションの特定のインスタンスに関連付けられます。多くの場合、各インスタンスには独自のデータがあり、通常は Kubernetes RBAC から独立した個別のアクセスコントロールを使用します。

他のユースケースとは異なり、SaaS 設定のテナントは Kubernetes API と直接インターフェイスしません。代わりに、SaaS アプリケーションは Kubernetes API とやり取りして、各テナントをサポートするために必要なオブジェクトを作成します。

Kubernetes コンストラクト

これらの各インスタンスでは、次のコンストラクトを使用してテナントを分離します。

名前空間

名前空間は、ソフトマルチテナンシーを実装するための基本です。これにより、クラスターを論理パーティションに分割できます。マルチテナンシーの実装に必要なクォータ、ネットワークポリシー、サービスアカウント、およびその他のオブジェクトは、名前空間に限定されます。

ネットワークポリシー

デフォルトでは、Kubernetes クラスター内のすべてのポッドは相互に通信できます。この動作は、ネットワークポリシーを使用して変更できます。

ネットワークポリシーは、ラベルまたは IP アドレス範囲を使用してポッド間の通信を制限します。テナント間の厳密なネットワーク分離が必要なマルチテナント環境では、ポッド間の通信を拒否するデフォルトのルールと、すべてのポッドが名前解決のために DNS サーバーをクエリできるようにする別のルールから始めることをお勧めします。そうすることで、名前空間内での通信を許可する、より寛容なルールの追加を開始できます。これは、必要に応じてさらに絞り込むことができます。

> **Note**
>
> Amazon VPC CNI は、Kubernetes ネットワークポリシーをサポートし、AWS で Kubernetes を実行するときに機密性の高いワークロードを分離して不正アクセスから保護できるポリシーを作成できるようになりました。つまり、Amazon EKS クラスター内でネットワークポリシー API のすべての機能を使用できます。このレベルのきめ細かな制御により、最小特権の原則を実装できます。これにより、承認されたポッドのみが相互に通信できるようになります。

> **Important**
>
> ネットワークポリシーは必須ですが、十分ではありません。ネットワークポリシーの適用には、Calico や Cilium などのポリシーエンジンが必要です。

ロールベースのアクセスコントロール (RBAC)

ロールとロールバインディングは、Kubernetes でロールベースのアクセスコントロール (RBAC) を適用するために使用される Kubernetes オブジェクトです。ロールには、クラスター内のオブジェクトに対して実行できるアクションのリストが含まれます。ロールバインディングは、ロールが適用される個人またはグループを指定します。エンタープライズおよび KaaS 設定では、RBAC を使用して、選択したグループまたは個人によるオブジェクトの管理を許可できます。

クォータ

クォータは、クラスターでホストされているワークロードの制限を定義するために使用されます。クォータでは、ポッドが消費できる CPU とメモリの最大量を指定するか、クラスターまたは名前空間に割り当てることができるリソースの数を制限できます。制限範囲を使用すると、各制限の最小値、最大値、デフォルト値を宣言できます。

共有クラスターでリソースを上書きすると、リソースを最大化できるため、多くの場合有益です。ただし、クラスターへの無制限のアクセスはリソース不足を引き起こし、パフォーマンスの低下やアプリケーションの可用性の低下につながる可能性があります。ポッドのリクエストの設定が低すぎて、実際のリソース使用率がノードの容量を超えた場合、ノードは CPU またはメモリ負荷を経験し始めます。この場合、ポッドが再起動されたり、ノードから削除されたりすることがあります。

これを防ぐには、マルチテナント環境の名前空間にクォータを課して、テナントがクラスターでポッドをスケジュールするときにリクエストと制限を指定するよう強制する計画を立てる必要があります。また、ポッドが消費できるリソースの量を制限することで、潜在的なサービス拒否を軽減します。

クォータを使用して、テナントの支出に合わせてクラスターのリソースを割り当てることもできます。これは、KaaS シナリオで特に役立ちます。

ポッドの優先度とプリエンプション

Pod の優先度とプリエンプションは、他の Pod よりも Pod を重視する場合に役立ちます。例えば、ポッドの優先度では、顧客 A のポッドを顧客 B よりも高い優先度で実行するように設定できます。使用可能な容量が不足すると、スケジューラは顧客 B の優先度の低いポッドを削除して、顧客 A の優先度の高いポッドに対応します。これは、顧客がプレミアムの支払いを希望する SaaS 環境で特に便利です。

> **Important**
>
> ポッドの優先度は、優先度の低い他のポッドに望ましくない影響を与える可能性があります。たとえば、犠牲者ポッドは正常に終了されますが、PodDisruptionBudget は保証されません。これにより、ポッドのクォーラムに依存する優先度の低いアプリケーションが破損する可能性があります。「優先の制限」を参照してください。

コントロールの軽減

マルチテナント環境の管理者としての主な懸念は、攻撃者が基盤となるホストにアクセスできないようにすることです。このリスクを軽減するには、次のコントロールを検討する必要があります。

コンテナのサンドボックス化された実行環境

サンドボクシングは、各コンテナを独自の分離された仮想マシンで実行する手法です。ポッドサンドボックスを実行するテクノロジーには、Firecracker や Weave's Firekube などがあります。

Firecracker を EKS でサポートされているランタイムにする作業の詳細については、https://threadreaderapp.com/thread/1238496944684597248.html を参照してください。

オープンポリシーエージェント (OPA) とゲートキーパー

Gatekeeper は、OPA で作成されたポリシーを適用する Kubernetes アドミッションコントローラーです。OPA を使用すると、別のインスタンスまたは他のテナントよりも高い優先度でテナントからポッドを実行するポリシーを作成できます。一般的な OPA ポリシーのコレクションは、このプロジェクトの GitHub リポジトリにあります。

CoreDNS の実験的な OPA プラグインもあり、OPA を使用して CoreDNS から返されたレコードをフィルタリング/制御できます。

キバーノ

Kyverno は、Kubernetes リソースとしてポリシーを使用して設定を検証、変更、生成できる Kubernetes ネイティブポリシーエンジンです。Kyverno は検証に Kustomize スタイルのオーバーレイを使用し、ミューテーション用の JSON パッチと戦略的マージパッチをサポートし、柔軟なトリガーに基づいて名前空間間でリソースをクローンできます。

Kyverno を使用して、名前空間の分離、ポッドセキュリティやその他のベストプラクティスの適用、ネットワークポリシーなどのデフォルト設定の生成を行うことができます。このプロジェクトの

GitHub リポジトリには、いくつかの例が含まれています。その他多くの は、Kyverno ウェブサイトのポリシーライブラリに含まれています。

テナントワークロードを特定のノードに分離する

テナントワークロードを特定のノードで実行するように制限することで、ソフトマルチテナンシーモデルの分離を強化できます。このアプローチでは、テナント固有のワークロードは、それぞれのテナント用にプロビジョニングされたノードでのみ実行されます。この分離を実現するために、ネイティブ Kubernetes プロパティ (ノードアフィニティ、テイントと許容範囲) を使用して、ポッドスケジューリングのために特定のノードをターゲットにし、他のテナントからのポッドがテナント固有のノードでスケジュールされないようにします。

パート 1 - ノードアフィニティ

Kubernetes ノードアフィニティは、ノードラベルに基づいてスケジューリングするノードをターゲットにするために使用されます。ノードアフィニティルールでは、ポッドはセレクタ条件に一致する特定のノードに引き付けられます。以下のポッド仕様では、`requiredDuringSchedulingIgnoredDuringExecution`ノードのアフィニティがそれぞれのポッドに適用されます。その結果、ポッドは次のキー/値 でラベル付けされたノードをターゲットにします`node-restriction.kubernetes.io/tenant: tenants-x`。

```
...
spec:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
        - matchExpressions:
          - key: node-restriction.kubernetes.io/tenant
            operator: In
            values:
            - tenants-x
...
```

このノードアフィニティでは、ラベルはスケジューリング中に必要になりますが、実行時には必要ではありません。基になるノードのラベルが変更された場合、そのラベルが変更されてもポッドは削除されません。ただし、将来のスケジューリングに影響する可能性があります。

> ⚠ Warning
>
> のラベルプレフィックスnode-restriction.kubernetes.io/は、Kubernetes で特別な意味を持ちます。EKS クラスターに対して有効になっている NodeRestriction は、このプレフィックスを持つラベルadding/removing/updatingkubeletすることを に禁止します。攻撃者は、これらのラベルを変更kubeletできないkubelet's credentials to update the node object or modify the system setup to pass these labels into `kubeletため、 を使用できません。このプレフィックスをすべてのポッド間スケジューリングに使用すると、攻撃者がノードラベルを変更してノードに別のワークロードセットを引き付けるシナリオを防ぐことができます。

Example

ノードアフィニティの代わりに、ノードセレクタを使用できます。ただし、ノードアフィニティはより表現力が高く、ポッドスケジューリング中により多くの条件を考慮できます。相違点とより高度なスケジューリングの選択肢の詳細については、アドバンスト Kubernetes ポッドからノードへのスケジューリングに関するこの CNCF ブログ記事を参照してください。

パート 2 - テイントと許容範囲

ポッドをノードに引き付けることは、この 3 つの部分からなるアプローチの最初の部分にすぎません。このアプローチを機能させるには、ポッドが承認されていないノードにポッドがスケジューリングされないようにする必要があります。不要なポッドや不正なポッドを撃退するために、Kubernetes はノードテイントを使用します。テイントは、ポッドのスケジュールを妨げる条件をノードに配置するために使用されます。以下のテイントでは、 のキーと値のペアを使用しますtenant: tenants-x。

```
...
    taints:
      - key: tenant
        value: tenants-x
        effect: NoSchedule
...
```

上記のノード の場合taint、テイントを許容するポッドのみをノードでスケジュールできます。承認されたポッドをノードにスケジュールできるようにするには、以下に示すように、それぞれのポッド仕様にテイントtolerationに を含める必要があります。

```
...
  tolerations:
  - effect: NoSchedule
    key: tenant
    operator: Equal
    value: tenants-x
...
```

上記のポッドは、少なくともその特定のテイントのために、ノードでのスケジューリングを停止tolerationされません。テイントは、ノードリソースのプレッシャーなど、特定の条件下でポッドスケジューリングを一時的に停止するために Kubernetes でも使用されます。ノードアフィニティ、テイント、許容度により、目的のポッドを特定のノードに効果的に引き付け、不要なポッドを反発できます。

> **Important**
>
> すべてのノードで を実行するには、特定の Kubernetes ポッドが必要です。これらのポッドの例は、コンテナネットワークインターフェイス (CNI) と kube-proxy デーモンセットによって開始されるポッドです。そのために、これらのポッドの仕様には、さまざまなテイントを許容するための非常に寛容な許容範囲が含まれています。これらの許容範囲を変更しないように注意する必要があります。これらの許容値を変更すると、クラスターオペレーションが正しくない可能性があります。さらに、OPA/Gatekeeper や Kyverno などのポリシー管理ツールを使用して、不正なポッドがこれらの許容値を使用することを防ぐ検証ポリシーを記述できます。

パート 3 - ノード選択のポリシーベースの管理

CICD パイプラインでのルールの適用など、ポッド仕様のノードアフィニティと許容度を管理するのに役立つツールがいくつかあります。ただし、分離の適用は Kubernetes クラスターレベルで行う必要があります。この目的のために、ポリシー管理ツールを使用して、リクエストペイロードに基づいてインバウンド Kubernetes API サーバーリクエストを変更し、上記のそれぞれのノードアフィニティルールと許容値を適用できます。

例えば、tentains-x 名前空間宛てのポッドには、tentens-x ノードでスケジューリングできるように、正しいノードアフィニティと許容値でスタンプを付けることができます。Kubernetes Mutating Admission Webhook を使用して設定されたポリシー管理ツールを使用すると、ポリシーを使用してインバウンドポッドの仕様を変更できます。ミューテーションは、必要なスケジューリングを可能に

するために必要な要素を追加します。ノードアフィニティを追加する OPA/Gatekeeper ポリシーの例を以下に示します。

```
apiVersion: mutations.gatekeeper.sh/v1alpha1
kind: Assign
metadata:
  name: mutator-add-nodeaffinity-pod
  annotations:
    aws-eks-best-practices/description: >-
      Adds Node affinity - https://kubernetes.io/docs/concepts/scheduling-eviction/
assign-pod-node/#node-affinity
spec:
  applyTo:
  - groups: [""]
    kinds: ["Pod"]
    versions: ["v1"]
  match:
    namespaces: ["tenants-x"]
  location:
 "spec.affinity.nodeAffinity.requiredDuringSchedulingIgnoredDuringExecution.nodeSelectorTerms"
  parameters:
    assign:
      value:
        - matchExpressions:
          - key: "tenant"
            operator: In
            values:
            - "tenants-x"
```

上記のポリシーは、ポッドを tenants-x 名前空間に適用する Kubernetes API サーバーリクエストに適用されます。ポリシーは`requiredDuringSchedulingIgnoredDuringExecution`ノードアフィニティルールを追加して、ポッドが `tenant: tenants-x`ラベルを持つノードに引き付けられるようにします。

以下に示す 2 番目のポリシーでは、ターゲット名前空間とグループ、種類、バージョンの同じ一致基準を使用して、同じポッド仕様に許容範囲を追加します。

```
apiVersion: mutations.gatekeeper.sh/v1alpha1
kind: Assign
metadata:
  name: mutator-add-toleration-pod
  annotations:
```

```
    aws-eks-best-practices/description: >-
      Adds toleration - https://kubernetes.io/docs/concepts/scheduling-eviction/taint-
and-toleration/
spec:
  applyTo:
  - groups: [""]
    kinds: ["Pod"]
    versions: ["v1"]
  match:
    namespaces: ["tenants-x"]
  location: "spec.tolerations"
  parameters:
    assign:
      value:
      - key: "tenant"
        operator: "Equal"
        value: "tenants-x"
        effect: "NoSchedule"
```

上記のポリシーはポッドに固有です。これは、ポリシーの 要素の変更されたlocation要素へのパスが原因です。デプロイやジョブのリソースなど、ポッドを作成するリソースを処理する追加のポリシーを記述できます。リストされたポリシーやその他の例は、このガイドのコンパニオン GitHub プロジェクトで確認できます。

これらの 2 つのミューテーションの結果、ポッドは目的のノードに引き付けられますが、同時に特定のノードテイントによってリピートされません。これを検証するには、2 つのkubectl呼び出しからの出力スニペットを表示して、 というラベルが付いたノードを取得しtenant=tenants-x、tenants-x名前空間のポッドを取得します。

```
kubectl get nodes -l tenant=tenants-x
NAME
ip-10-0-11-255...
ip-10-0-28-81...
ip-10-0-43-107...

kubectl -n tenants-x get pods -owide
NAME                                  READY   STATUS    RESTARTS   AGE   IP
 NODE
tenant-test-deploy-58b895ff87-2q7xw   1/1     Running   0          13s   10.0.42.143
 ip-10-0-43-107...
tenant-test-deploy-58b895ff87-9b6hg   1/1     Running   0          13s   10.0.18.145
 ip-10-0-28-81...
```

```
tenant-test-deploy-58b895ff87-nxvw5   1/1     Running   0          13s   10.0.30.117
 ip-10-0-28-81...
tenant-test-deploy-58b895ff87-vw796   1/1     Running   0          13s   10.0.3.113
 ip-10-0-11-255...
tenant-test-pod                       1/1     Running   0          13s   10.0.35.83
 ip-10-0-43-107...
```

上記の出力からわかるように、すべてのポッドは というラベルの付いたノードでスケジュールされますtenant=tenants-x。簡単に言うと、ポッドは目的のノードでのみ実行され、他のポッド (必要なアフィニティと許容値なし) は実行されません。テナントワークロードは効果的に分離されます。

変更後のポッド仕様の例を以下に示します。

```
apiVersion: v1
kind: Pod
metadata:
  name: tenant-test-pod
  namespace: tenants-x
spec:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
        - matchExpressions:
          - key: tenant
            operator: In
            values:
            - tenants-x
...
  tolerations:
  - effect: NoSchedule
    key: tenant
    operator: Equal
    value: tenants-x
...
```

Important

Kubernetes API サーバーリクエストフローに統合されているポリシー管理ツールは、変更と検証アドミッションウェブフックを使用して、指定された期間内に API サーバーのリクエストに応答するように設計されています。これは通常 3 秒以下です。ウェブフック呼び出

しが設定された時間内にレスポンスを返さない場合、インバウンド API サーバーリクエストのミューテーションや検証が発生する場合と発生しない場合があります。この動作は、アドミッションウェブフック設定が Fail Open または Fail Close に設定されているかどうかに基づいています。

上記の例では、OPA/Gatekeeper 用に記述されたポリシーを使用しました。ただし、ノード選択のユースケースを処理する他のポリシー管理ツールもあります。例えば、この Kyverno ポリシーを使用してノードアフィニティミューテーションを処理できます。

> **Note**
>
> 正しく動作している場合、ポリシーを変更すると、インバウンド API サーバーのリクエストペイロードに必要な変更が反映されます。ただし、変更の保持を許可する前に、ポリシーの検証を含めて、必要な変更が行われることを確認する必要があります。これは、tenant-to-node分離にこれらのポリシーを使用する場合に特に重要です。また、監査ポリシーを含めて、クラスターに不要な設定がないか定期的にチェックすることをお勧めします。

リファレンス

- k-rail 特定のポリシーを適用してマルチテナント環境を保護するように設計されています。
- Amazon EKS を使用した MultiTenant SaaS アプリケーションのセキュリティプラクティス

ハードマルチテナンシー

ハードマルチテナンシーは、テナントごとに個別のクラスターをプロビジョニングすることで実装できます。これはテナント間で非常に強力な分離を提供しますが、いくつかの欠点があります。

まず、テナントが多い場合、このアプローチはすぐに高価になる可能性があります。各クラスターのコントロールプレーンのコストを支払う必要があるだけでなく、クラスター間でコンピューティングリソースを共有することはできません。これにより、最終的に断片化が発生し、クラスターのサブセットが十分に活用されず、他のサブセットが過剰に活用されます。

次に、これらのすべてのクラスターを管理するための特別なツールを購入または構築する必要があります。時間が経つと、数百または数千のクラスターを管理するだけでは扱いにくくなる可能性があります。

最後に、テナントごとのクラスターの作成は、名前空間の作成と比較して遅くなります。ただし、規制の厳しい業界や強力な分離が必要な SaaS 環境では、ハードテナンシーアプローチが必要になる場合があります。

今後の方向性

Kubernetes コミュニティは、ソフトマルチテナンシーの現在の欠点とハードマルチテナンシーの課題を認識しています。マルチテナンシー特別インタレストグループ (SIG) は、階層名前空間コントローラー (HNC) や仮想クラスターなど、いくつかの改良プロジェクトを通じてこれらの欠点に対処しようとしています。

HNC 提案 (KEP) では、〔ポリシー] オブジェクト継承を持つ名前空間間に親子関係を作成する方法と、テナント管理者がサブ名前空間を作成する機能について説明します。

仮想クラスター提案では、クラスター内の各テナント (Kubernetes on Kubernetes とも呼ばれます) に対して、API サーバー、コントローラーマネージャー、スケジューラなど、コントロールプレーンサービスの個別のインスタンスを作成するメカニズムについて説明します。

マルチテナンシーベンチマーク提案では、分離とセグメンテーションに名前空間を使用してクラスターを共有するためのガイドラインと、ガイドラインへの準拠を検証するためのコマンドラインツール kubectl-mtb を提供します。

マルチクラスター管理ツールとリソース

- バンザイ雲
- コマンダー
- レンズ
- Nirmata
- Rafay
- ランチャー
- ウィーブフラックス

監査と口グ記録

[監査] ログの収集と分析は、さまざまな理由で役立ちます。口グは、根本原因の分析と属性、つまり特定のユーザーに変更を 書き込むのに役立ちます。十分な口グが収集されると、異常な動作を

検出するためにも使用できます。EKS では、監査ログは Amazon Cloudwatch Logs に送信されます。EKS の監査ポリシーは次のとおりです。

```
apiVersion: audit.k8s.io/v1beta1
kind: Policy
rules:
  # Log full request and response for changes to aws-auth ConfigMap in kube-system
 namespace
  - level: RequestResponse
    namespaces: ["kube-system"]
    verbs: ["update", "patch", "delete"]
    resources:
      - group: "" # core
        resources: ["configmaps"]
        resourceNames: ["aws-auth"]
    omitStages:
      - "RequestReceived"
  # Do not log watch operations performed by kube-proxy on endpoints and services
  - level: None
    users: ["system:kube-proxy"]
    verbs: ["watch"]
    resources:
      - group: "" # core
        resources: ["endpoints", "services", "services/status"]
  # Do not log get operations performed by kubelet on nodes and their statuses
  - level: None
    users: ["kubelet"] # legacy kubelet identity
    verbs: ["get"]
    resources:
      - group: "" # core
        resources: ["nodes", "nodes/status"]
  # Do not log get operations performed by the system:nodes group on nodes and their
 statuses
  - level: None
    userGroups: ["system:nodes"]
    verbs: ["get"]
    resources:
      - group: "" # core
        resources: ["nodes", "nodes/status"]
  # Do not log get and update operations performed by controller manager, scheduler,
 and endpoint-controller on endpoints in kube-system namespace
  - level: None
    users:
```

```
      - system:kube-controller-manager
      - system:kube-scheduler
      - system:serviceaccount:kube-system:endpoint-controller
    verbs: ["get", "update"]
    namespaces: ["kube-system"]
    resources:
      - group: "" # core
        resources: ["endpoints"]
  # Do not log get operations performed by apiserver on namespaces and their statuses/
finalizations
  - level: None
    users: ["system:apiserver"]
    verbs: ["get"]
    resources:
      - group: "" # core
        resources: ["namespaces", "namespaces/status", "namespaces/finalize"]
  # Do not log get and list operations performed by controller manager on
 metrics.k8s.io resources
  - level: None
    users:
      - system:kube-controller-manager
    verbs: ["get", "list"]
    resources:
      - group: "metrics.k8s.io"
  # Do not log access to health, version, and swagger non-resource URLs
  - level: None
    nonResourceURLs:
      - /healthz*
      - /version
      - /swagger*
  # Do not log events resources
  - level: None
    resources:
      - group: "" # core
        resources: ["events"]
  # Log request for updates/patches to nodes and pods statuses by kubelet and node
 problem detector
  - level: Request
    users: ["kubelet", "system:node-problem-detector", "system:serviceaccount:kube-
system:node-problem-detector"]
    verbs: ["update", "patch"]
    resources:
      - group: "" # core
        resources: ["nodes/status", "pods/status"]
```

```
    omitStages:
      - "RequestReceived"
  # Log request for updates/patches to nodes and pods statuses by system:nodes group
  - level: Request
    userGroups: ["system:nodes"]
    verbs: ["update", "patch"]
    resources:
      - group: "" # core
        resources: ["nodes/status", "pods/status"]
    omitStages:
      - "RequestReceived"
  # Log delete collection requests by namespace-controller in kube-system namespace
  - level: Request
    users: ["system:serviceaccount:kube-system:namespace-controller"]
    verbs: ["deletecollection"]
    omitStages:
      - "RequestReceived"
  # Log metadata for secrets, configmaps, and tokenreviews to protect sensitive data
  - level: Metadata
    resources:
      - group: "" # core
        resources: ["secrets", "configmaps"]
      - group: authentication.k8s.io
        resources: ["tokenreviews"]
    omitStages:
      - "RequestReceived"
  # Log requests for serviceaccounts/token resources
  - level: Request
    resources:
      - group: "" # core
        resources: ["serviceaccounts/token"]
  # Log get, list, and watch requests for various resource groups
  - level: Request
    verbs: ["get", "list", "watch"]
    resources:
      - group: "" # core
      - group: "admissionregistration.k8s.io"
      - group: "apiextensions.k8s.io"
      - group: "apiregistration.k8s.io"
      - group: "apps"
      - group: "authentication.k8s.io"
      - group: "authorization.k8s.io"
      - group: "autoscaling"
      - group: "batch"
```

```
      - group: "certificates.k8s.io"
      - group: "extensions"
      - group: "metrics.k8s.io"
      - group: "networking.k8s.io"
      - group: "policy"
      - group: "rbac.authorization.k8s.io"
      - group: "scheduling.k8s.io"
      - group: "settings.k8s.io"
      - group: "storage.k8s.io"
    omitStages:
      - "RequestReceived"
  # Default logging level for known APIs to log request and response
  - level: RequestResponse
    resources:
      - group: "" # core
      - group: "admissionregistration.k8s.io"
      - group: "apiextensions.k8s.io"
      - group: "apiregistration.k8s.io"
      - group: "apps"
      - group: "authentication.k8s.io"
      - group: "authorization.k8s.io"
      - group: "autoscaling"
      - group: "batch"
      - group: "certificates.k8s.io"
      - group: "extensions"
      - group: "metrics.k8s.io"
      - group: "networking.k8s.io"
      - group: "policy"
      - group: "rbac.authorization.k8s.io"
      - group: "scheduling.k8s.io"
      - group: "settings.k8s.io"
      - group: "storage.k8s.io"
    omitStages:
      - "RequestReceived"
  # Default logging level for all other requests to log metadata only
  - level: Metadata
    omitStages:
      - "RequestReceived"
```

推奨事項

監査ログを有効にする

監査ログは、EKS によって管理される EKS マネージド Kubernetes コントロールプレーンログの一部です。Kubernetes API サーバー、コントローラーマネージャー、スケジューラのログと監査ログを含むコントロールプレーンログを有効または無効にする手順については、https://docs.aws.amazon.com/eks/latest/userguide/control-plane-logs.html#enabling-control-plane-log-export を参照してください。

> **Note**
>
> コントロールプレーンのログ記録を有効にすると、CloudWatch にログを保存するためのコストが発生します。これにより、継続的なセキュリティコストに関するより広範な問題が発生します。最終的には、これらのコストと、 財務上の損失、評判の低下などのセキュリティ違反のコストを比較検討する必要があります。このガイドの推奨事項の一部のみを実装することで、環境を適切に保護できる場合があります。

> **Warning**
>
> CloudWatch Logs エントリの最大サイズは 1MB ですが、Kubernetes API リクエストの最大サイズは 1.5 MiB です。1MB を超えるログエントリは切り捨てられるか、リクエストメタデータのみが含まれます。

監査メタデータを使用する

Kubernetes 監査ログには、リクエストが承認されたかどうか`authorization.k8s.io/decision`と、決定の理由を示す 2 つの注釈が含まれています`authorization.k8s.io/reason`。これらの属性を使用して、特定の API コールが許可された理由を確認します。

疑わしいイベントのアラームを作成する

アラームを作成して、403 Forbidden レスポンスと 401 Unauthorized レスポンスが増加した場合に自動的にアラートを送信し、`host`、`sourceIPs`、 などの属性を使用してそれらのリクエストの送信元`k8s_user.username`を確認します。

Log Insights を使用してログを分析する

CloudWatch Log Insights を使用して、 ロール、RoleBindings、ClusterRoles、ClusterRoleBindings などの RBAC オブジェクトへの変更をモニタリングします。いくつかのサンプルクエリが以下に表示されます。

ConfigMap aws-auth の更新を一覧表示します。

```
fields @timestamp, @message
| filter @logStream like "kube-apiserver-audit"
| filter verb in ["update", "patch"]
| filter objectRef.resource = "configmaps" and objectRef.name = "aws-auth" and
 objectRef.namespace = "kube-system"
| sort @timestamp desc
```

検証ウェブフックの新規作成または変更を一覧表示します。

```
fields @timestamp, @message
| filter @logStream like "kube-apiserver-audit"
| filter verb in ["create", "update", "patch"] and responseStatus.code = 201
| filter objectRef.resource = "validatingwebhookconfigurations"
| sort @timestamp desc
```

ロールへの作成、更新、削除オペレーションを一覧表示します。

```
fields @timestamp, @message
| sort @timestamp desc
| limit 100
| filter objectRef.resource="roles" and verb in ["create", "update", "patch", "delete"]
```

RoleBindings への作成、更新、削除オペレーションを一覧表示します。

```
fields @timestamp, @message
| sort @timestamp desc
| limit 100
| filter objectRef.resource="rolebindings" and verb in ["create", "update", "patch",
 "delete"]
```

ClusterRoles への作成、更新、削除オペレーションを一覧表示します。

```
fields @timestamp, @message
```

```
| sort @timestamp desc
| limit 100
| filter objectRef.resource="clusterroles" and verb in ["create", "update", "patch",
 "delete"]
```

ClusterRoleBindings への作成、更新、削除オペレーションを一覧表示します。

```
fields @timestamp, @message
| sort @timestamp desc
| limit 100
| filter objectRef.resource="clusterrolebindings" and verb in ["create", "update",
 "patch", "delete"]
```

シークレットに対する不正な読み取りオペレーションをプロットします。

```
fields @timestamp, @message
| sort @timestamp desc
| limit 100
| filter objectRef.resource="secrets" and verb in ["get", "watch", "list"] and
 responseStatus.code="401"
| stats count() by bin(1m)
```

失敗した匿名リクエストのリスト:

```
fields @timestamp, @message, sourceIPs.0
| sort @timestamp desc
| limit 100
| filter user.username="system:anonymous" and responseStatus.code in ["401", "403"]
```

CloudTrail ログの監査

サービスアカウントの IAM ロール (IRSA) を使用しているポッドによって呼び出される AWS APIs は、サービスアカウントの名前とともに CloudTrail に自動的に記録されます。API の呼び出しを明示的に許可されていないサービスアカウントの名前がログに表示された場合は、IAM ロールの信頼ポリシーの設定が間違っている可能性があります。一般的に、Cloudtrail は特定の IAM プリンシパルに AWS API コールを記述する優れた方法です。

CloudTrail Insights を使用して疑わしいアクティビティを発掘する

CloudTrail インサイトは、CloudTrail 証跡からの書き込み管理イベントを自動的に分析し、異常なアクティビティを警告します。これにより、IRSA を使用して IAM ロールを引き受けるポッドな

ど、AWS アカウントの書き込み APIs で呼び出しボリュームが増加するタイミングを特定できます。詳細については、CloudTrail Insights の発表: 異常な API アクティビティの特定と対応」を参照してください。

その他のリソース

ログの量が増えると、Log Insights または別のログ分析ツールを使用してログを解析およびフィルタリングすると、無効になる可能性があります。別の方法として、Sysdig Falco と ekscloudwatch の実行を検討することもできます。Falco は監査ログを分析し、長期間にわたって異常や不正にフラグを立てます。ekscloudwatch プロジェクトは、監査ログイベントを CloudWatch から Falco に転送して分析します。Falco は、デフォルトの監査ルールのセットと、独自の監査ルールを追加する機能を提供します。

さらに別のオプションとして、監査ログを S3 に保存し、SageMaker ランダムカットフォレストアルゴリズムを使用して、さらなる調査を必要とする異常な動作を行うことが考えられます。

ツールとリソース

以下の商用プロジェクトとオープンソースプロジェクトを使用して、確立されたベストプラクティスとのクラスターの整合性を評価できます。

- Amazon EKS セキュリティイマージョンワークショップ - Detective Controls
- kubeaudit
- kube-scan Kubernetes Common Configuration Scoring System フレームワークに従って、クラスターで実行されているワークロードにリスクスコアを割り当てます。
- kubesec.io
- ポラリス
- 右舷
- Snyk
- Kubescape Kubescape は、クラスター、YAML ファイル、Helm チャートをスキャンするオープンソースの kubernetes セキュリティツールです。複数のフレームワーク (NSA-CISA や MITRE ATT&CK® など) に従って設定ミスを検出します。

ネットワークセキュリティ

ネットワークセキュリティにはいくつかのファセットがあります。1 つ目は、サービス間のネットワークトラフィックの流れを制限するルールの適用です。2 つ目は、転送中のトラフィックの暗号化

です。EKS にこれらのセキュリティ対策を実装するメカニズムはさまざまですが、多くの場合、次の項目が含まれます。

トラフィックコントロール

- ネットワークポリシー
- セキュリティグループ

ネットワーク暗号化

- サービスメッシュ
- コンテナネットワークインターフェイス (CNIs)
- Ingress Controllers と Load Balancer
- Nitro インスタンス
- cert-manager を使用する ACM プライベート CA

ネットワークポリシー

Kubernetes クラスター内では、すべての Pod 間通信がデフォルトで許可されています。この柔軟性は実験を促進するのに役立ちますが、安全とは見なされません。Kubernetes ネットワークポリシーは、ポッド間 (多くの場合、東部/西部トラフィック) およびポッドと外部サービス間のネットワークトラフィックを制限するメカニズムを提供します。Kubernetes ネットワークポリシーは、OSI モデルのレイヤー 3 と 4 で動作します。ネットワークポリシーは、ポッド、名前空間セレクタ、ラベルを使用して送信元ポッドと送信先ポッドを識別しますが、IP アドレス、ポート番号、プロトコル、またはこれらの組み合わせを含めることもできます。ネットワークポリシーは、多くの場合、イングレスルールとエグレスルールと呼ばれるポッドへのインバウンド接続とアウトバウンド接続の両方に適用できます。

Amazon VPC CNI プラグインのネイティブネットワークポリシーのサポートにより、ネットワークポリシーを実装して、kubernetes クラスター内のネットワークトラフィックを保護できます。これにより、アップストリームの Kubernetes ネットワークポリシー API と統合され、Kubernetes 標準との互換性と準拠が保証されます。アップストリーム API でサポートされている異なる識別子を使用してポリシーを定義できます。デフォルトでは、すべてのイングレストラフィックとエグレストラフィックがポッドに許可されます。policyType Ingress を持つネットワークポリシーが指定されている場合、ポッドへの許可された接続のみがポッドのノードからの接続であり、イングレスルールで許

可されている接続です。出力ルールにも同じが適用されます。複数のルールが定義されている場合、すべてのルールの結合が決定時に考慮されます。したがって、評価の順序はポリシーの結果には影響しません。

> **Important**
>
> EKS クラスターを初めてプロビジョニングする場合、VPC CNI ネットワークポリシー機能はデフォルトで有効になっていません。これを有効にするには、サポートされている VPC CNI アドオンバージョンをデプロイし、vpc-cni アドオン`true`で`ENABLE_NETWORK_POLICY`フラグを に設定します。詳細な手順については、Amazon EKS ユーザーガイドを参照してください。

推奨事項

ネットワークポリシーの開始方法 - 最小特権の原則に従う

デフォルトの拒否ポリシーを作成する

RBAC ポリシーと同様に、ネットワークポリシーでは最小特権のアクセス原則に従うことをお勧めします。まず、名前空間で を使用してすべてのインバウンドトラフィックとアウトバウンドトラフィックを制限するすべてのポリシーを拒否します。

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: default-deny
  namespace: default
spec:
  podSelector: {}
  policyTypes:
  - Ingress
  - Egress
```

default-deny

		Target		Source				
#	Policy name	Namespace	Pods	Namespace	Pods	Subnet	Ports	
1	default-deny	default	Any	Any	Any	Any	Any	

> **Note**
>
> 上記のイメージは、Tufin のネットワークポリシービューワーによって作成されました。

DNS クエリを許可するルールを作成する

すべてのルールをデフォルトで拒否すると、ポッドが名前解決のために CoreDNS をクエリできるようにするルールなど、追加のルールのレイヤーを開始できます。

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-dns-access
  namespace: default
spec:
  podSelector:
    matchLabels: {}
  policyTypes:
  - Egress
  egress:
  - to:
    - namespaceSelector:
        matchLabels:
          kubernetes.io/metadata.name: kube-system
      podSelector:
        matchLabels:
          k8s-app: kube-dns
    ports:
    - protocol: UDP
      port: 53
```

allow-dns-access

		Target		Destination				
#	Policy name	Namespace	Pods	Namespace	Pods	Subnet	Ports	
1	allow-dns-access	default	Any	name: kube-system	Any		UDP: 53	

名前空間/ポッド間のトラフィックフローを選択的に許可するルールを段階的に追加する

アプリケーションの要件を理解し、必要に応じてきめ細かな入出力ルールを作成します。以下の例は、ポート 80 の進入トラフィックを app-oneから に制限する方法を示していますclient-one。これにより、攻撃対象領域が最小限に抑えられ、不正アクセスのリスクが軽減されます。

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-ingress-app-one
  namespace: default
spec:
  podSelector:
    matchLabels:
      k8s-app: app-one
  policyTypes:
  - Ingress
  ingress:
  - from:
    - podSelector:
        matchLabels:
          k8s-app: client-one
    ports:
    - protocol: TCP
      port: 80
```

allow-ingress-app-one

		Target		Source				
#	Policy name	Namespace	Pods	Namespace	Pods	Subnet	Ports	
1	allow-ingress-app-one	default	k8s-app: app-one	default	k8s-app: client-one		TCP: 80	

ネットワークポリシーの適用のモニタリング

- ネットワークポリシーエディタを使用する
  - ネットワークポリシーエディタは、ネットワークフローログからの視覚化、セキュリティスコア、自動生成に役立ちます。
  - インタラクティブな方法でネットワークポリシーを構築する
- 監査ログ

  - EKS クラスターの監査ログを定期的に確認する
  - 監査ログは、ネットワークポリシーの変更など、クラスターで実行されたアクションに関する豊富な情報を提供します。
  - この情報を使用して、ネットワークポリシーへの経時的な変更を追跡し、不正または予期しない変更を検出します。
- 自動テスト
  - 本番環境をミラーリングするテスト環境を作成し、ネットワークポリシーに違反しようとするワークロードを定期的にデプロイすることで、自動テストを実装します。
- メトリクスのモニタリング
  - オブザーバビリティエージェントを設定して、VPC CNI ノードエージェントから prometheus メトリクスをスクレイプします。これにより、 はエージェントのヘルスと sdk エラーをモニタリングできます。
- ネットワークポリシーを定期的に監査する
  - ネットワークポリシーを定期的に監査して、現在のアプリケーション要件を満たしていることを確認します。アプリケーションが進化するにつれて、監査により冗長な進入、退出ルールを削除し、アプリケーションに過剰なアクセス許可がないことを確認する機会が得られます。
- Open Policy Agent (OPA) を使用してネットワークポリシーが存在することを確認する
  - 以下に示すような OPA ポリシーを使用して、アプリケーションポッドをオンボーディングする前にネットワークポリシーが常に存在することを確認します。このポリシーは、対応するネットワークポリシーが存在しない`k8s-app: sample-app`場合、ラベル付きの k8s ポッドのオンボーディングを拒否します。

```
package kubernetes.admission
import data.kubernetes.networkpolicies

deny[msg] {
    input.request.kind.kind == "Pod"
    pod_label_value := {v["k8s-app"] | v := input.request.object.metadata.labels}
    contains_label(pod_label_value, "sample-app")
    np_label_value := {v["k8s-app"] | v :=
 networkpolicies[_].spec.podSelector.matchLabels}
    not contains_label(np_label_value, "sample-app")
    msg:= sprintf("The Pod %v could not be created because it is missing an associated
 Network Policy.", [input.request.object.metadata.name])
}
contains_label(arr, val) {
```

```
    arr[_] == val
}
```

トラブルシューティング

vpc-network-policy-controller、node-agent ログをモニタリングする

EKS コントロールプレーンコントローラーマネージャーログを有効にして、ネットワークポリシー機能を診断します。コントロールプレーンログを CloudWatch ロググループにストリーミングし、CloudWatch Log Insights を使用して高度なクエリを実行できます。ログから、ネットワークポリシーに解決されるポッドエンドポイントオブジェクト、ポリシーの照合ステータスを表示し、ポリシーが期待どおりに動作している場合はデバッグできます。

さらに、Amazon VPC CNI では、EKS ワーカーノードから Amazon Cloudwatch へのポリシー適用ログの収集とエクスポートを有効にすることができます。有効にすると、CloudWatch Container Insights を活用して、ネットワークポリシーに関連する使用状況に関するインサイトを提供できます。

Amazon VPC CNI には、ノード上の eBPF プログラムとやり取りするためのインターフェイスを提供する SDK も付属しています。SDK は、 `aws-node`がノードにデプロイされたときにインストールされます。SDK バイナリは、ノードの `/opt/cni/bin` ディレクトリにインストールされています。起動時に、SDK は eBPF プログラムやマップの検査などの基本的な機能をサポートします。

```
sudo /opt/cni/bin/aws-eks-na-cli ebpf progs
```

ネットワークトラフィックメタデータのログ記録

AWS VPC フローログは、送信元と送信先の IP アドレスやポートなど、VPC を通過するトラフィックに関するメタデータを、受け入れられた/ドロップされたパケットとともにキャプチャします。この情報は、ポッドを含む VPC 内のリソース間の疑わしいアクティビティや異常なアクティビティを探すために分析できます。ただし、ポッドの IP アドレスは置き換えられると頻繁に変更されるため、フローログだけでは不十分である可能性があります。Calico Enterprise は、ポッドラベルやその他のメタデータを使用してフローログを拡張するため、ポッド間のトラフィックフローを簡単に解読できます。

セキュリティグループ

EKS は AWS VPC セキュリティグループ (SGs) を使用して、Kubernetes コントロールプレーンとクラスターのワーカーノード間のトラフィックを制御します。セキュリティグループは、ワーカーノード、他の VPC リソース、および外部 IP アドレス間のトラフィックを制御するためにも使用さ

れます。EKS クラスター (Kubernetes バージョン 1.14-eks.3 以降) をプロビジョニングすると、クラスターセキュリティグループが自動的に作成されます。このセキュリティグループは、EKS コントロールプレーンとマネージドノードグループからのノード間の無制限の通信を許可します。簡単にするために、アンマネージド型ノードグループを含むすべてのノードグループにクラスター SG を追加することをお勧めします。

Kubernetes バージョン 1.14 および EKS バージョン eks.3 以前は、EKS コントロールプレーンとノードグループに別々のセキュリティグループが設定されていました。コントロールプレーンとノードグループのセキュリティグループの最小ルールと推奨ルールは、https://docs.aws.amazon.com/eks/latest/userguide/sec-group-reqs.html にあります。コントロールプレーンセキュリティグループの最小ルールでは、ワーカーノード SG からのポート 443 のインバウンドを許可します。このルールにより、kubelets は Kubernetes API サーバーと通信できるようになります。また、ワーカーノード SG へのアウトバウンドトラフィック用のポート 10250 も含まれています。10250 は kubelets がリッスンするポートです。同様に、最小ノードグループルールでは、コントロールプレーン SG からのインバウンドポート 10250 とコントロールプレーン SG へのアウトバウンドポート 443 が許可されます。最後に、ノードグループ内のノード間の無制限の通信を許可するルールがあります。

クラスター内で実行されるサービス間の通信を制御し、RDS データベースなどのクラスター外で実行されるサービスをサービスする必要がある場合は、ポッドのセキュリティグループを検討してください。ポッドのセキュリティグループを使用すると、既存のセキュリティグループをポッドのコレクションに割り当てることができます。

> **Warning**
>
> ポッドの作成前に存在しないセキュリティグループを参照すると、ポッドはスケジュールされません。

`SecurityGroupPolicy` オブジェクトを作成し、 `PodSelector`または を指定することで、セキュリティグループに割り当てられるポッドを制御できます`ServiceAccountSelector`。セレクターを に設定すると`{}`、 で参照される SGs `SecurityGroupPolicy`が、名前空間内のすべてのポッドまたは名前空間内のすべてのサービスアカウントに割り当てられます。ポッドのセキュリティグループを実装する前に、すべての考慮事項を理解しておく必要があります。

> **Important**
>
> ポッドSGs を使用する場合は、クラスターセキュリティグループへのポート 53 アウトバウンドを許可する SGs を作成する必要があります。同様に、ポッドセキュリティグループから

のポート 53 インバウンドトラフィックを受け入れるようにクラスターセキュリティグループを更新する必要があります。

> **Important**
>
> セキュリティグループの制限は、ポッドのセキュリティグループを使用する場合にも適用されるため、慎重に使用してください。

> **Important**
>
> ポッド用に設定されたすべてのプローブについて、クラスターセキュリティグループ (kubelet) からのインバウンドトラフィックのルールを作成する必要があります。

> **Important**
>
> ポッドのセキュリティグループは、EC2 インスタンスの ENI 密度を高めるために作成された ENI トランキングと呼ばれる機能に依存しています。ポッドが SG に割り当てられると、VPC コントローラーはノードグループからのブランチ ENI をポッドに関連付けます。ポッドがスケジュールされた時点でノードグループで使用できる十分なブランチ ENIs がない場合、ポッドは保留状態のままになります。インスタンスがサポートできるブランチ ENIs の数は、インスタンスタイプ/ファミリーによって異なります。詳細については、https://docs.aws.amazon.com/eks/latest/userguide/security-groups-for-pods.html#supported-instance-types を参照してください。

ポッドのセキュリティグループでは、ポリシーデーモンのオーバーヘッドなしでクラスター内外のネットワークトラフィックを AWS ネイティブに制御できますが、他のオプションも使用できます。たとえば、Cilium ポリシーエンジンを使用すると、ネットワークポリシーで DNS 名を参照できます。Calico Enterprise には、ネットワークポリシーを AWS セキュリティグループにマッピングするためのオプションが含まれています。Istio などのサービスメッシュを実装している場合は、Egress Gateway を使用して、ネットワーク出力を特定の完全修飾ドメインまたは IP アドレスに制限できます。このオプションの詳細については、Istio の出力トラフィック制御に関する 3 つのパートシリーズをお読みください。

Pod のネットワークポリシーとセキュリティグループを使用するタイミング

Kubernetes ネットワークポリシーを使用するタイミング

- pod-to-podトラフィックの制御
  - クラスター内のポッド間のネットワークトラフィック (東西トラフィック) の制御に適しています。
- IP アドレスまたはポートレベルでトラフィックを制御する (OSI レイヤー 3 または 4)

ポッドに AWS セキュリティグループを使用するタイミング (SGP)

- 既存の AWS 設定を活用する
  - AWS サービスへのアクセスを管理する複雑な EC2 セキュリティグループのセットがすでにあり、EC2 インスタンスから EKS にアプリケーションを移行する場合、SGPs はセキュリティグループのリソースを再利用してポッドに適用するために非常に適しています。
- AWS サービスへのアクセスを制御する
  - EKS クラスター内で実行されているアプリケーションは、他の AWS サービス (RDS データベース) と通信し、ポッドから AWS SGPs を使用します。
- ポッドとノードのトラフィックの分離
  - ポッドトラフィックを残りのノードトラフィックから完全に分離する場合は、`POD_SECURITY_GROUP_ENFORCING_MODE=strict` モードで SGP を使用します。

ポッドとネットワークポリシーのセキュリティグループを使用するベストプラクティス

- レイヤードセキュリティ
  - レイヤードセキュリティアプローチに SGP ネットワークポリシーと kubernetes ネットワークポリシーの組み合わせを使用する
  - SGPs を使用してクラスターの一部ではない AWS サービスへのネットワークレベルのアクセスを制限し、kubernetes ネットワークポリシーはクラスター内のポッド間のネットワークトラフィックを制限できます。
- 最小権限の原則
  - ポッドまたは名前空間間の必要なトラフィックのみを許可する

- アプリケーションをセグメント化する
  - 可能な限り、アプリケーションをネットワークポリシーでセグメント化して、アプリケーションが侵害された場合に爆発半径を小さくします。
- ポリシーをシンプルかつ明確に保つ
  - Kubernetes ネットワークポリシーは、非常に詳細で複雑な場合があります。設定ミスのリスクを軽減し、管理オーバーヘッドを軽減するために、できるだけシンプルに保つのが最善です。
- アタックサーフェスを減らす
  - アプリケーションの露出を制限して攻撃対象領域を最小限に抑える

> **Important**
>
> ポッドのセキュリティグループには、 `strict`と の 2 つの強制モードがありますstandard。EKS クラスターのポッド機能にネットワークポリシーとセキュリティグループの両方を使用する場合は、 `standard` モードを使用する必要があります。

ネットワークセキュリティに関しては、多くの場合、レイヤードアプローチが最も効果的なソリューションです。kubernetes ネットワークポリシーと SGP を組み合わせて使用すると、EKS で実行されているアプリケーションに堅牢なdefense-in-depth戦略を提供できます。

Service Mesh ポリシーの強制または Kubernetes ネットワークポリシー

`service mesh` は、アプリケーションに追加できる専用のインフラストラクチャレイヤーです。これにより、独自のコードに追加することなく、オブザーバビリティ、トラフィック管理、セキュリティなどの機能を透過的に追加できます。

サービスメッシュは OSI モデルのレイヤー 7 (アプリケーション) でポリシーを適用しますが、kubernetes ネットワークポリシーはレイヤー 3 (ネットワーク) とレイヤー 4 (トランスポート) で動作します。このスペースには、AWS AppMesh、Istio、Linkerd など、多くのサービスがあります。

ポリシーの適用にサービスメッシュを使用するタイミング

- サービスメッシュへの既存の投資
- トラフィック管理、オブザーバビリティ、セキュリティなどのより高度な機能が必要
  - トラフィック制御、負荷分散、サーキットブレーク、レート制限、タイムアウトなど。

- サービスのパフォーマンスに関する詳細なインサイト (レイテンシー、エラー率、1 秒あたりのリクエスト数、リクエストボリュームなど)
- mTLS などのセキュリティ機能にサービスメッシュを実装して活用したい

よりシンプルなユースケースで Kubernetes ネットワークポリシーを選択する

- 相互に通信できるポッドを制限する
- ネットワークポリシーは、サービスメッシュよりも必要なリソースが少ないため、単純なユースケースや、サービスメッシュの実行と管理のオーバーヘッドが正当化されない可能性がある小規模なクラスターに適しています。

> **Note**
>
> ネットワークポリシーとサービスメッシュを一緒に使用することもできます。ネットワークポリシーを使用してポッド間のセキュリティと分離のベースラインレベルを提供し、サービスメッシュを使用してトラフィック管理、オブザーバビリティ、セキュリティなどの追加機能を追加します。

ThirdPartyのネットワークポリシーエンジン

グローバルネットワークポリシー、DNS ホスト名ベースのルール、レイヤー 7 ルール、ServiceAccount ベースのルール、明示的な拒否/ログアクションなどの高度なポリシー要件がある場合は、サードパーティーのネットワークポリシーエンジンを検討してください。Calico は、EKS と連携する Tigera のオープンソースポリシーエンジンです。Calico は、Kubernetes ネットワークポリシー機能のフルセットの実装に加えて、Istio と統合した場合の HTTP などのレイヤー 7 ルールのサポートなど、より豊富な機能を備えた拡張ネットワークポリシーをサポートしています。Calico ポリシーの範囲は、名前空間、ポッド、サービスアカウント、またはグローバルに設定できます。ポリシーがサービスアカウントにスコープされている場合、一連の入出力ルールがそのサービスアカウントに関連付けられます。適切な RBAC ルールを設定すると、チームがこれらのルールを上書きするのを防ぎ、IT セキュリティプロフェッショナルが名前空間の管理を安全に委任できるようになります。同様に、Cilium の保守者もネットワークポリシーを拡張して、 HTTP などのレイヤー 7 ルールの部分的なサポートを含めました。Cilium は DNS ホスト名もサポートしています。これは、Kubernetes サービス/ポッドと、VPC 内外で実行されるリソース間のトラフィックを制限するのに役立ちます。対照的に、Calico Enterprise には、Kubernetes ネットワークポリシーを AWS セキュリティグループと DNS ホスト名にマッピングできる機能が含まれています。

一般的な Kubernetes ネットワークポリシーのリストは、https://github.com/ahmetb/kubernetes-network-policy-recipes にあります。Calico の同様のルールのセットは、https://docs.projectcalico.org/security/calico-network-policy で入手できます。

Amazon VPC CNI ネットワークポリシーエンジンへの移行

一貫性を維持し、予期しないポッド通信動作を回避するには、クラスターに 1 つのネットワークポリシーエンジンのみをデプロイすることをお勧めします。3P から VPC CNI ネットワークポリシーエンジンに移行する場合は、VPC CNI ネットワークポリシーのサポートを有効にする前に、既存の 3P NetworkPolicy CRDs を Kubernetes NetworkPolicy リソースに変換することをお勧めします。また、移行したポリシーを本番環境に適用する前に、別のテストクラスターでテストします。これにより、ポッド通信動作の潜在的な問題や不整合を特定して対処できます。

移行ツール

移行プロセスを支援するために、既存の Calico/Cilium ネットワークポリシー CRD を Kubernetes ネイティブネットワークポリシーに変換する K8s Network Policy Migrator と呼ばれるツールを開発しました。 CRDs 変換後、VPC CNI ネットワークポリシーコントローラーを実行している新しいクラスターで、変換されたネットワークポリシーを直接テストできます。このツールは、移行プロセスを合理化し、スムーズに移行できるように設計されています。

> **Important**
>
> 移行ツールは、ネイティブの kubernetes ネットワークポリシー API と互換性のある 3P ポリシーのみを変換します。3P プラグインが提供する高度なネットワークポリシー機能を使用している場合、移行ツールはそれらをスキップしてレポートします。

移行ツールは、現在 AWS VPC CNI Network ポリシーエンジニアリングチームではサポートされておらず、ベストエフォートベースでお客様に提供されることに注意してください。このツールを使用して移行プロセスを容易にすることをお勧めします。ツールで問題やバグが発生した場合は、GitHub の問題を作成してください。お客様からのフィードバックは当社にとって貴重なものであり、当社のサービスの継続的な改善に役立ちます。

その他のリソース

- Kubernetes & Tigera: ネットワークポリシー、セキュリティ、監査
- Calico Enterprise

- Cilium
- NetworkPolicy Editor Cilium のインタラクティブポリシーエディタ
- Inspektor Gadget advise network-policy gadget ネットワークトラフィックの分析に基づいてネットワークポリシーを提案します

転送中の暗号化

PCI、HIPAA、またはその他の規制に準拠する必要があるアプリケーションは、転送中にデータを暗号化する必要がある場合があります。現在、TLS は、ワイヤ上のトラフィックを暗号化するための事実上の選択肢です。TLS は、以前の SSL と同様に、暗号化プロトコルを使用してネットワーク経由で安全な通信を提供します。TLS は対称暗号化を使用して、セッションの開始時にネゴシエートされた共有シークレットに基づいてデータを暗号化します。以下は、Kubernetes 環境でデータを暗号化する方法のいくつかです。

Nitro インスタンス

C5n, G4, I3en, M5dn, M5n, P3dn, R5dnR5n など、次の Nitro インスタンスタイプ間で交換されるトラフィックは、デフォルトで自動的に暗号化されます。トランジットゲートウェイやロードバランサーなどの中間ホップがある場合、トラフィックは暗号化されません。転送中の暗号化の詳細と、デフォルトでネットワーク暗号化をサポートするインスタンスタイプの完全なリストについては、「転送中の暗号化」を参照してください。

コンテナネットワークインターフェイス (CNIs)

WeaveNet は、パケットトラフィックには NaCl 暗号化を使用し、高速データパストラフィックには IPsec ESP を使用して、すべてのトラフィックを自動的に暗号化するように設定できます。

サービスメッシュ

転送中の暗号化は、App Mesh、Linkerd v2、Istio などのサービスメッシュで実装することもできます。AppMesh は、X.509 証明書または Envoy の Secret Discovery Service (SDS) による mTLS をサポートしています。Linkerd と Istio はどちらも mTLS をサポートしています。

aws-app-mesh-examples GitHub リポジトリは、Envoy コンテナで X.509 証明書と SPIRE を SDS プロバイダーとして使用して mTLS を設定するためのチュートリアルを提供します。

- X.509 証明書を使用した mTLS の設定
- SPIRE (SDS) を使用した TLS の設定

App Mesh は、AWS Certificate Manager (ACM) によって発行されたプライベート証明書、または仮想ノードのローカルファイルシステムに保存されている証明書による TLS 暗号化もサポートしています。

aws-app-mesh-examples GitHub リポジトリには、ACM によって発行された証明書と Envoy コンテナにパッケージ化された証明書を使用して TLS を設定するためのチュートリアルが用意されています。

- ファイル提供の TLS 証明書を使用した TLS の設定
- AWS Certificate Manager を使用した TLS の設定

Ingress Controllers と Load Balancer

イングレスコントローラーは、クラスター外から発せられる HTTP/S トラフィックを、クラスター内で実行されているサービスにインテリジェントにルーティングする方法です。多くの場合、これらのイングレスは Classic Load Balancer や Network Load Balancer (NLB) などのレイヤー 4 Load Balancerによって前面に配置されます。暗号化されたトラフィックは、 ロードバランサー、イングレスリソース、ポッドなど、ネットワーク内のさまざまな場所で終了できます。SSL 接続を終了する方法と場所は、最終的に組織のネットワークセキュリティポリシーによって決まります。たとえば、end-to-endの暗号化を必要とするポリシーがある場合は、Pod でトラフィックを復号化する必要があります。これにより、最初のハンドシェイクの確立にサイクルを費やす必要があるため、Pod に負担がかかります。全体的な SSL/TLS 処理は CPU を大量に消費します。したがって、柔軟性がある場合は、Ingress またはロードバランサーで SSL オフロードを実行してみてください。

AWS Elastic ロードバランサーで暗号化を使用する

AWS Application Load Balancer (ALB) と Network Load Balancer (NLB) はどちらもトランスポート暗号化 (SSL と TLS) をサポートしています。ALB の`alb.ingress.kubernetes.io/certificate-arn`注釈を使用すると、ALB に追加する証明書を指定できます。注釈を省略すると、コントローラーはホストフィールドを使用して使用可能な AWS Certificate Manager (ACM) 証明書を一致させることで、それを必要とする証明書をリスナーに追加しようとします。EKS v1.15 以降では、以下の例に示すように、NLB で `service.beta.kubernetes.io/aws-load-balancer-ssl-cert`注釈を使用できます。

```
apiVersion: v1
kind: Service
metadata:
  name: demo-app
  namespace: default
```

```
  labels:
    app: demo-app
  annotations:
     service.beta.kubernetes.io/aws-load-balancer-type: "nlb"
     service.beta.kubernetes.io/aws-load-balancer-ssl-cert: "<certificate ARN>"
     service.beta.kubernetes.io/aws-load-balancer-ssl-ports: "443"
     service.beta.kubernetes.io/aws-load-balancer-backend-protocol: "http"
spec:
  type: LoadBalancer
  ports:
  - port: 443
    targetPort: 80
    protocol: TCP
  selector:
    app: demo-app
//---
kind: Deployment
apiVersion: apps/v1
metadata:
  name: nginx
  namespace: default
  labels:
    app: demo-app
spec:
  replicas: 1
  selector:
    matchLabels:
      app: demo-app
  template:
    metadata:
      labels:
        app: demo-app
    spec:
      containers:
        - name: nginx
          image: nginx
          ports:
            - containerPort: 443
              protocol: TCP
            - containerPort: 80
              protocol: TCP
```

SSL/TLS 終了の追加の例を次に示します。

- Contour を使用した EKS Ingress の保護と GitOps の暗号化
- ACM を使用して Amazon EKS ワークロードの HTTPS トラフィックを終了するにはどうすればよいですか?

> **Important**
>
> AWS LB コントローラーなどの一部の Ingress は、Ingress Spec の一部としてではなく、注釈を使用して SSL/TLS を実装します。

cert-manager を使用した ACM プライベート CA

TLS と mTLS を有効にして、進入時、ポッド上、および ACM Private Certificate Authority (CA) と証明書を配布、更新、および取り消すための一般的な Kubernetes アドオンである cert-manager を使用するポッド間で EKS アプリケーションワークロードを保護できます。ACM Private CA は、高可用性で安全なマネージド CA であり、独自の CA を管理するための前払いとメンテナンスのコストはありません。デフォルトの Kubernetes 認証機関を使用している場合は、セキュリティを向上させ、ACM Private CA のコンプライアンス要件を満たす機会があります。ACM Private CA は、FIPS 140-2 Level 3 ハードウェアセキュリティモジュール (非常に安全) でプライベートキーを保護します。これに対して、メモリにエンコードされたキーを格納するデフォルトの CA (安全性が低い) と比較します。また、一元化された CA により、Kubernetes 環境内外の両方のプライベート証明書をより詳細に制御し、監査可能性を向上させることができます。

ワークロード間の相互 TLS の短縮 CA モード

EKS で mTLS に ACM プライベート CA を使用する場合は、存続期間の短い CA モードの存続期間の短い証明書を使用することをお勧めします。汎用 CA モードで有効期間の短い証明書を発行することは可能ですが、有効期間の短い CA モードを使用すると、新しい証明書を頻繁に発行する必要があるユースケースで、コスト効率が高くなります (一般モードよりも約 75% 安くなります) 。これに加えて、プライベート証明書の有効期間を EKS クラスター内のポッドの有効期間に合わせて調整する必要があります。ACM Private CA とその利点の詳細については、こちらを参照してください。

ACM セットアップ手順

まず、ACM Private CA 技術ドキュメントに記載されている手順に従って Private CA を作成します。プライベート CA を作成したら、通常のインストール手順に従って cert-manager をインストールします。cert-manager をインストールしたら、GitHub のセットアップ手順に従って Private

CA Kubernetes cert-manager プラグインをインストールします。プラグインを使用すると、cert-manager は ACM Private CA にプライベート証明書をリクエストできます。

Private CA と cert-manager とプラグインがインストールされた EKS クラスターができたので、アクセス許可を設定し、発行者を作成します。EKS ノードロールの IAM アクセス許可を更新して、ACM Private CA へのアクセスを許可します。<CA_ARN> をプライベート CA の値に置き換えます。

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "awspcaissuer",
            "Action": [
                "acm-pca:DescribeCertificateAuthority",
                "acm-pca:GetCertificate",
                "acm-pca:IssueCertificate"
            ],
            "Effect": "Allow",
            "Resource": "arn:aws:acm-pca:us-west-2:123456789012:certificate-
authority/12345678-1234-1234-1234-123456789012"
        }
    ]
}
```

IAM アカウントのサービスロール、または IRSA を使用することもできます。完全な例については、以下の追加のリソースセクションを参照してください。

次のテキストを含む cluster-issuer.yaml という名前のカスタムリソース定義ファイルを作成し、<CA_ARN>および <Region>情報をプライベート CA に置き換えて、Amazon EKS で発行者を作成します。

```
apiVersion: awspca.cert-manager.io/v1beta1
kind: AWSPCAClusterIssuer
metadata:
          name: demo-test-root-ca
spec:
          arn: <CA_ARN>
          region: <Region>
```

作成した発行者をデプロイします。

```
kubectl apply -f cluster-issuer.yaml
```

EKS クラスターは、プライベート CA に証明書をリクエストするように設定されています。上で作成したプライベート CA 発行者にissuerRefフィールドの値を変更することで、cert-manager のCertificateリソースを使用して証明書を発行できるようになりました。証明書リソースを指定してリクエストする方法の詳細については、cert-manager の証明書リソースガイドを参照してください。例については、こちらを参照してください。

Istio と cert-manager を使用する ACM プライベート CA

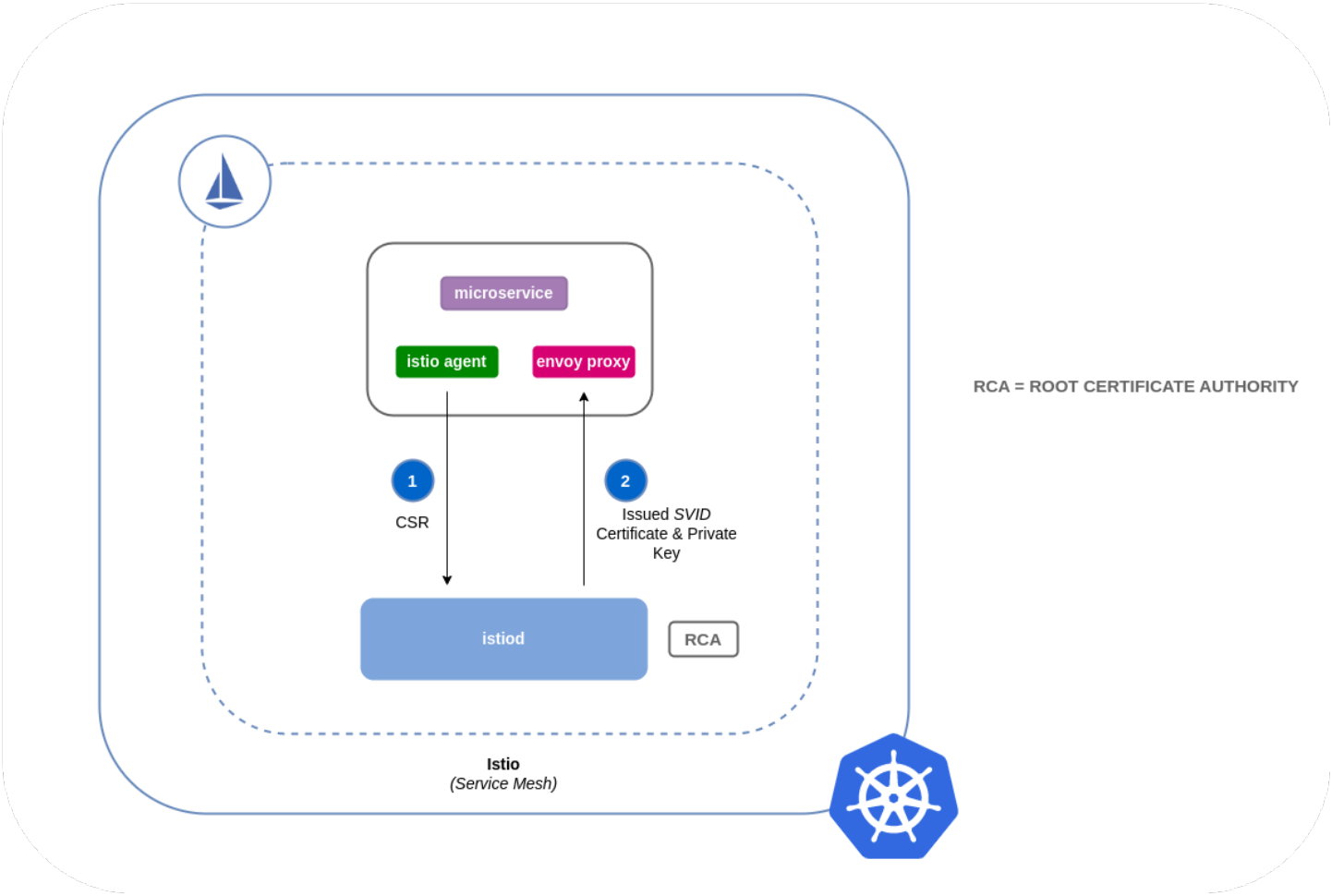
EKS クラスターで Istio を実行している場合は、Istio コントロールプレーン (特に istiod) をルート認証局 (CA) として機能させないようにし、ワークロード間の mTLS のルート CA として ACM プライベート CA を設定できます。このアプローチを使用する場合は、ACM プライベート CA で存続期間の短い CA モードを使用することを検討してください。詳細については、前のセクションとこのブログ記事を参照してください。

Istio での証明書署名の仕組み (デフォルト)

Kubernetes のワークロードは、サービスアカウントを使用して識別されます。サービスアカウントを指定しない場合、Kubernetes は自動的にサービスアカウントをワークロードに割り当てます。また、サービスアカウントは関連付けられたトークンを自動的にマウントします。このトークンは、Kubernetes API に対して認証するために、ワークロードのサービスアカウントによって使用されます。サービスアカウントは Kubernetes の ID として十分かもしれませんが、Istio には独自の ID 管理システムと CA があります。ワークロードがエンボイサイドカープロキシで起動するときは、信頼できると見なされ、メッシュ内の他のサービスと通信できるようにするために、Istio から割り当てられた ID が必要です。

Istio からこの ID を取得するために、 は証明書署名リクエスト (CSR) と呼ばれるリクエストを Istio コントロールプレーンistio-agentに送信します。この CSR には、処理前にワークロードの ID を検証できるように、サービスアカウントトークンが含まれています。この検証プロセスはistiod、登録機関 (または RA) と CA の両方として機能する によって処理されます。RA は、検証済みの CSR のみが CA に渡すようにするゲートキーパーとして機能します。CSR が検証されると、CA に転送され、サービスアカウントで SPIFFE ID を含む証明書が発行されます。この証明書は、SPIFFE 検証可能な ID ドキュメント (SVID) と呼ばれます。SVID は、識別および通信サービス間の転送中のトラフィックを暗号化するために、リクエスト元のサービスに割り当てられます。

Istio 証明書署名リクエストのデフォルトフロー:



ACM Private CA を使用した Istio での証明書署名の仕組み

Istio Certificate Signing Request エージェント (istio-csr) と呼ばれる cert-manager アドオンを使用して、Istio を ACM Private CA と統合できます。このエージェントは、Istio ワークロードとコントロールプレーンコンポーネントを証明書マネージャー発行者、この場合は ACM Private CA で保護できるようにします。istio-csr エージェントは、受信 CSR を検証するデフォルト設定で istiod が処理するのと同じサービスを公開します。 CSRs ただし、検証後、証明書マネージャーがサポートするリソース (外部 CA 発行者と の統合) にリクエストが変換されます。

ワークロードから CSR がある場合、その CSR は istio-csr に転送され、ACM Private CA に証明書がリクエストされます。istio-csr と ACM Private CA 間のこの通信は、AWS Private CA 発行者プラグインによって有効になります。Cert Manager は、このプラグインを使用して ACM Private CA に TLS 証明書をリクエストします。発行者プラグインは ACM Private CA サービスと通信して、ワークロードの署名付き証明書をリクエストします。証明書が署名されると、istio-csr に返されます。istio-csr は署名されたリクエストを読み取り、CSR を開始したワークロードに返します。

istio-csr を使用した Istio 証明書署名リクエストのフロー

image::istio-csr-with-acm-private-ca.png[istio-csr を使用した Istio 証明書署名リクエストのフロー]

Istio とプライベート CA のセットアップ手順

1. まず、このセクションの同じセットアップ手順に従って、以下を完了します。
2. プライベート CA を作成する
3. cert-manager をインストールする
4. 発行者プラグインをインストールする
5. アクセス許可を設定し、発行者を作成します。発行者は CA を表し、ワークロード証明書の署名istiodとメッシュに使用されます。ACM Private CA と通信します。
6. istio-system 名前空間を作成します。これは、 istiod certificateおよびその他の Istio リソースがデプロイされる場所です。
7. AWS Private CA Issuer Plugin で設定された Istio CSR をインストールします。ワークロードの証明書署名リクエストを保持して、承認および署名されていることを確認できます (preserveCertificateRequests=true)。

```
helm install -n cert-manager cert-manager-istio-csr jetstack/cert-manager-istio-csr \
--set "app.certmanager.issuer.group=awspca.cert-manager.io" \
--set "app.certmanager.issuer.kind=AWSPCAClusterIssuer" \
--set "app.certmanager.issuer.name=<the-name-of-the-issuer-you-created>" \
--set "app.certmanager.preserveCertificateRequests=true" \
--set "app.server.maxCertificateDuration=48h" \
--set "app.tls.certificateDuration=24h" \
--set "app.tls.istiodCertificateDuration=24h" \
--set "app.tls.rootCAFile=/var/run/secrets/istio-csr/ca.pem" \
--set "volumeMounts[0].name=root-ca" \
--set "volumeMounts[0].mountPath=/var/run/secrets/istio-csr" \
--set "volumes[0].name=root-ca" \
--set "volumes[0].secret.secretName=istio-root-ca"
```

8. カスタム設定を使用して Istio をインストールし、メッシュの証明書プロバイダーcert-manager istio-csrとして を istiodに置き換えます。このプロセスは、Istio Operator を使用して実行できます。

```
apiVersion: install.istio.io/v1alpha1
kind: IstioOperator
metadata:
  name: istio
  namespace: istio-system
spec:
```

```
profile: "demo"
hub: gcr.io/istio-release
values:
global:
  # Change certificate provider to cert-manager istio agent for istio agent
  caAddress: cert-manager-istio-csr.cert-manager.svc:443
components:
  pilot:
    k8s:
      env:
        # Disable istiod CA Sever functionality
      - name: ENABLE_CA_SERVER
        value: "false"
      overlays:
      - apiVersion: apps/v1
        kind: Deployment
        name: istiod
        patches:

          # Mount istiod serving and webhook certificate from Secret mount
        - path: spec.template.spec.containers.[name:discovery].args[7]
          value: "--tlsCertFile=/etc/cert-manager/tls/tls.crt"
        - path: spec.template.spec.containers.[name:discovery].args[8]
          value: "--tlsKeyFile=/etc/cert-manager/tls/tls.key"
        - path: spec.template.spec.containers.[name:discovery].args[9]
          value: "--caCertFile=/etc/cert-manager/ca/root-cert.pem"

        - path: spec.template.spec.containers.[name:discovery].volumeMounts[6]
          value:
            name: cert-manager
            mountPath: "/etc/cert-manager/tls"
            readOnly: true
        - path: spec.template.spec.containers.[name:discovery].volumeMounts[7]
          value:
            name: ca-root-cert
            mountPath: "/etc/cert-manager/ca"
            readOnly: true

        - path: spec.template.spec.volumes[6]
          value:
            name: cert-manager
            secret:
              secretName: istiod-tls
        - path: spec.template.spec.volumes[7]
```

```
            value:
              name: ca-root-cert
              configMap:
                defaultMode: 420
                name: istio-ca-root-cert
```

9. 作成した上記のカスタムリソースをデプロイします。

```
istioctl operator init
kubectl apply -f istio-custom-config.yaml
```

10.これで、ワークロードを EKS クラスターのメッシュにデプロイし、mTLS を適用できるようになりました。

Istio 証明書署名リクエスト

image::istio-csr-requests.png[Istio 証明書署名リクエスト]

ツールとリソース

- Amazon EKS セキュリティイマージョンワークショップ - ネットワークセキュリティ
- cert-manager と ACM Private CA プラグインを実装して EKS で TLS を有効にする方法。
- 新しい AWS Load Balancer Controller と ACM Private CA を使用して Amazon EKS でend-to-end の TLS 暗号化をセットアップします。
- GitHub の Private CA Kubernetes cert-manager プラグイン。
- プライベート CA Kubernetes 証明書マネージャープラグインユーザーガイド。
- AWS Private Certificate Authority の有効期間の短い証明書モードを使用する方法
- egress-operator プロトコル検査なしでクラスターからの出力トラフィックを制御する演算子と DNS プラグイン
- SUSE オープンソースのゼロトラストコンテナセキュリティプラットフォームによる NeuVector は、ポリシーネットワークルール、データ損失防止 (DLP)、ウェブアプリケーションファイアウォール (WAF)、ネットワーク脅威シグネチャを提供します。

データ暗号化とシークレット管理

保管中の暗号化

Kubernetes で使用できる AWS ネイティブストレージには、EBS、EFS、FSx for Lustre の 3 つの異なるオプションがあります。3 つのオプションはすべて、サービスマネージドキーまたはカスタマーマスターキー (CMK) を使用して保管時の暗号化を行います。EBS の場合は、ツリー内ストレージドライバーまたは EBS CSI ドライバーを使用できます。どちらも、ボリュームを暗号化し、CMK を提供するためのパラメータが含まれています。EFS の場合、EFS CSI ドライバーを使用できますが、EBS とは異なり、EFS CSI ドライバーは動的プロビジョニングをサポートしていません。EKS で EFS を使用する場合は、PV を作成する前に、ファイルシステムの保管時の暗号化をプロビジョニングして設定する必要があります。EFS ファイルの暗号化の詳細については、「保管中のデータの暗号化」を参照してください。保管時の暗号化の提供に加えて、EFS と FSx for Lustre には、転送中のデータを暗号化するためのオプションが含まれています。FSx for Lustre はデフォルトでこれを行います。EFS の場合、次の例のように `tls`パラメータを PV `mountOptions`の に追加することで、トランスポート暗号化を追加できます。

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: efs-pv
spec:
  capacity:
    storage: 5Gi
  volumeMode: Filesystem
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Retain
  storageClassName: efs-sc
  mountOptions:
    - tls
  csi:
    driver: efs.csi.aws.com
    volumeHandle: <file_system_id>
```

FSx CSI ドライバーは、Lustre ファイルシステムの動的プロビジョニングをサポートします。デフォルトではサービスマネージドキーでデータを暗号化しますが、次の例のように独自の CMK を指定するオプションがあります。

```
kind: StorageClass
```

```
apiVersion: storage.k8s.io/v1
metadata:
  name: fsx-sc
provisioner: fsx.csi.aws.com
parameters:
  subnetId: subnet-056da83524edbe641
  securityGroupIds: sg-086f61ea73388fb6b
  deploymentType: PERSISTENT_1
  kmsKeyId: <kms_arn>
```

> **Important**
>
> 2020 年 5 月 28 日現在、EKS Fargate ポッドのエフェメラルボリュームに書き込まれるすべてのデータは、業界標準の AES-256 暗号化アルゴリズムを使用してデフォルトで暗号化されています。暗号化と復号は サービスによってシームレスに処理されるため、アプリケーションに変更を加える必要はありません。

保管中のデータを暗号化する

保管中のデータの暗号化はベストプラクティスと見なされます。暗号化が必要かどうか不明な場合は、データを暗号化します。

CMKsを定期的にローテーションする

CMKs を自動的にローテーションするように KMS を設定します。これにより、年 1 回キーがローテーションされ、古いキーは無期限に保存されるため、データは復号できます。詳細については、「カスタマーマスターキーの更新」を参照してください。

EFS アクセスポイントを使用して共有データセットへのアクセスを簡素化する

異なる POSIX ファイルアクセス許可を持つ共有データセットがある場合、または異なるマウントポイントを作成して共有ファイルシステムの一部へのアクセスを制限する場合は、EFS アクセスポイントの使用を検討してください。アクセスポイントの操作の詳細については、https://docs.aws.amazon.com/efs/latest/ug/efs-access-points.html を参照してください。現在、アクセスポイント (AP) を使用する場合は、PV の `volumeHandle`パラメータで AP を参照する必要があります。

> **Important**
>
> 2021 年 3 月 23 日現在、EFS CSI ドライバーは EFS アクセスポイントの動的プロビジョニングをサポートしています。アクセスポイントはEFS ファイルシステムへのアプリケーション固有のエントリポイントであり、複数のポッド間でファイルシステムを簡単に共有できます。各 EFS ファイルシステムは、最大 120 PVsことができます。詳細については、「Amazon EFS CSI 動的プロビジョニングの紹介」を参照してください。

シークレットの管理

Kubernetes シークレットは、ユーザー証明書、パスワード、API キーなどの機密情報を保存するために使用されます。これらは、base64 でエンコードされた文字列として etcd に保持されます。EKS では、 etcd ノードの EBS ボリュームは EBS 暗号化で暗号化されます。ポッドは、 のシークレットを参照することで、Kubernetes シークレットオブジェクトを取得できますpodSpec。これらのシークレットは、環境変数にマッピングすることも、ボリュームとしてマウントすることもできます。シークレットの作成の詳細については、https://kubernetes.io/docs/concepts/configuration/secret/ を参照してください。

> **Warning**
>
> 特定の名前空間内のシークレットは、シークレットの名前空間内のすべてのポッドで参照できます。

> **Warning**
>
> ノードオーソライザーを使用すると、Kubelet はノードにマウントされたすべてのシークレットを読み取ることができます。

Kubernetes シークレットのエンベロープ暗号化に AWS KMS を使用する

これにより、一意のデータ暗号化キー (DEK) を使用してシークレットを暗号化できます。その後、DEK は AWS KMS のキー暗号化キー (KEK) を使用して暗号化され、定期的なスケジュールで自動的にローテーションできます。Kubernetes 用 KMS プラグインを使用すると、すべての Kubernetes シークレットはプレーンテキストではなく暗号化テキストの etcd に保存さ

れ、Kubernetes API サーバーによってのみ復号できます。詳細については、「EKS 暗号化プロバイダーのサポートを使用した多層防御」を参照してください。

Kubernetes シークレットの使用を監査する

EKS で、監査ログ記録を有効にし、シークレットが使用されたときに警告する CloudWatch メトリクスフィルターとアラームを作成します (オプション)。以下は、Kubernetes 監査ログ のメトリクスフィルターの例です`{($.verb="get") && ($.objectRef.resource="secret")}`。CloudWatch Log Insights では、次のクエリを使用することもできます。

```
fields @timestamp, @message
| sort @timestamp desc
| limit 100
| stats count(*) by objectRef.name as secret
| filter verb="get" and objectRef.resource="secrets"
```

上記のクエリには、特定の期間内にシークレットにアクセスされた回数が表示されます。

```
fields @timestamp, @message
| sort @timestamp desc
| limit 100
| filter verb="get" and objectRef.resource="secrets"
| display objectRef.namespace, objectRef.name, user.username, responseStatus.code
```

このクエリは、シークレットにアクセスしようとしたユーザーの名前空間とユーザー名、およびレスポンスコードとともにシークレットを表示します。

シークレットを定期的にローテーションする

Kubernetes はシークレットを自動的にローテーションしません。シークレットを更新する必要がある場合は、 Vault や AWS Secrets Manager などの外部シークレットストアの使用を検討してください。

異なるアプリケーションからシークレットを分離する方法として個別の名前空間を使用する

名前空間内のアプリケーション間で共有できないシークレットがある場合は、それらのアプリケーション用に別の名前空間を作成します。

環境変数の代わりにボリュームマウントを使用する

環境変数の値は、意図せずログに表示される可能性があります。ボリュームとしてマウントされたシークレットは tmpfs ボリューム (RAM ベースのファイルシステム) としてインスタンス化され、ポッドが削除されるとノードから自動的に削除されます。

外部シークレットプロバイダーを使用する

Kubernetes シークレットの使用には、AWS Secrets Manager や Hashicorp のボールトなど、いくつかの実行可能な選択肢があります。これらのサービスは、きめ細かなアクセスコントロール、強力な暗号化、Kubernetes Secrets では利用できないシークレットの自動ローテーションなどの機能を提供します。Bitnami のシールされたシークレットは、非対称暗号化を使用して「シールされたシークレット」を作成するもう 1 つのアプローチです。パブリックキーはシークレットを暗号化するために使用されますが、シークレットの復号に使用されるプライベートキーはクラスター内に保持されるため、Git などのソース管理システムにシールされたシークレットを安全に保存できます。詳細については、「シールされたシークレットを使用した Kubernetes でのシークレットデプロイの管理」を参照してください。

外部シークレットストアの使用が増えるにつれて、 はそれらを Kubernetes と統合する必要があります。Secret Store CSI ドライバーは、CSI ドライバーモデルを使用して外部シークレットストアからシークレットを取得するコミュニティプロジェクトです。現在、ドライバーは AWS Secrets Manager、Azure、Vault、および GCP をサポートしています。AWS プロバイダーは、AWS Secrets Manager と AWS Parameter Store の両方をサポートしています。また、有効期限が切れたときにシークレットをローテーションするように設定し、AWS Secrets Manager シークレットを Kubernetes シークレットに同期することもできます。シークレットの同期は、シークレットをボリュームから読み取るのではなく、環境変数として参照する必要がある場合に役立ちます。

> **Note**
>
> シークレットストア CSI ドライバーがシークレットを取得する必要がある場合、シークレットを参照するポッドに割り当てられた IRSA ロールを引き受けます。このオペレーションのコードは、こちらにあります。

AWS Secrets & Configuration Provider (ASCP) の詳細については、次のリソースを参照してください。

- Kubernetes Secret Store CSI ドライバーで AWS Secrets Configuration Provider を使用する方法

- Secrets Manager シークレットと Kubernetes Secrets Store CSI ドライバーの統合

external-secrets は、Kubernetes で外部シークレットストアを使用するもう 1 つの方法です。CSI ドライバーと同様に、外部シークレットは AWS Secrets Manager など、さまざまなバックエンドに対して機能します。違いは、外部シークレットストアからシークレットを取得するのではなく、外部シークレットはこれらのバックエンドからシークレットとして Kubernetes にシークレットをコピーすることです。これにより、任意のシークレットストアを使用してシークレットを管理し、Kubernetes ネイティブの方法でシークレットとやり取りできます。

ツールとリソース

- Amazon EKS セキュリティイマージョンワークショップ - データ暗号化とシークレット管理

ランタイムセキュリティ

ランタイムセキュリティは、コンテナの実行中にコンテナをアクティブに保護します。このアイデアは、コンテナ内で悪意のあるアクティビティが発生しないように検出および/または防止することです。これは、Linux 機能、セキュアコンピューティング (seccomp)、AppArmor、SELinux など、Kubernetes と統合された Linux カーネルまたはカーネル拡張機能のさまざまなメカニズムで実現できます。また、Linux カーネルメカニズムの手動設定を少なくして、ベースラインの確立や異常なアクティビティの検出に役立つ Amazon GuardDuty やサードパーティーツールなどのオプションもあります。

> **Important**
>
> Kubernetes は現在、seccomp、AppArmor、または SELinux プロファイルを Nodes にロードするためのネイティブメカニズムを提供していません。手動でロードするか、ブートストラップ時にノードにインストールする必要があります。これは、スケジューラがプロファイルを持つノードを認識しないため、Pod でそれらを参照する前に実行する必要があります。Security Profiles Operator などのツールが、ノードへのプロファイルのプロビジョニングを自動化する方法を以下に示します。

セキュリティコンテキストと組み込みの Kubernetes コントロール

多くの Linux ランタイムセキュリティメカニズムは Kubernetes と緊密に統合されており、Kubernetes セキュリティコンテキストを介して設定できます。このようなオプションの 1 つは

privilegedフラグです。これはfalseデフォルトで、有効になっている場合、ホストのルートと基本的に同等です。本番環境のワークロードで特権モードを有効にすることはほぼ常に適切ではありませんが、必要に応じてコンテナに詳細な権限を提供できるコントロールが他にも多数あります。

Linux の機能

Linux 機能を使用すると、ルートユーザーのすべての機能を提供することなく、ポッドまたはコンテナに特定の機能を付与できます。例としてはCAP_NET_ADMIN、ネットワークインターフェイスまたはファイアウォールの設定を許可する や、システムクロックの操作CAP_SYS_TIMEを許可する などがあります。

Seccomp

セキュアコンピューティング (seccomp) を使用すると、コンテナ化されたアプリケーションが基盤となるホストオペレーティングシステムのカーネルに対して特定の syscall を行うことを防ぐことができます。Linux オペレーティングシステムには数百のシステム呼び出しがありますが、コンテナを実行するために lion が共有する必要はありません。コンテナによって実行できる syscall を制限することで、アプリケーションのアタックサーフェスを効果的に減らすことができます。

Seccomp は、syscalls を傍受し、許可リストに登録されているもののみを通過できるようにすることで機能します。Docker には、ほとんどの汎用ワークロードに適したデフォルトの seccomp プロファイルがあり、containerd などの他のコンテナランタイムは同等のデフォルトを提供します。以下を Pod 仕様の securityContextセクションに追加することで、コンテナランタイムのデフォルトの seccomp プロファイルを使用するようにコンテナまたは Pod を設定できます。

```
securityContext:
  seccompProfile:
    type: RuntimeDefault
```

1.22 以降 (アルファでは 1.27 以降安定) 、上記RuntimeDefaultは単一の kubelet フラグ を使用してノード上のすべての Pod に使用できます--seccomp-default。次に、 で指定されたプロファイルsecurityContextは、他のプロファイルにのみ必要です。

追加の権限を必要とするモノに対して独自のプロファイルを作成することもできます。これは手動で行うと非常に面倒ですが、Inspektor Gadget (ネットワークポリシーを生成するためのネットワークセキュリティセクションでも推奨) や Security Profiles Operator などのツールがあり、eBPF や ログなどのツールを使用してベースライン特権要件を seccomp プロファイルとして記録できます。Security Profiles Operator では、ポッドとコンテナで使用できるように、記録されたプロファイルのノードへのデプロイを自動化できます。

AppArmor と SELinux

AppArmor と SELinux は、必須アクセスコントロールまたは MAC システムと呼ばれます。概念は seccomp に似ていますが、APIs や機能が異なるため、 特定のファイルシステムパスやネットワークポートなどのアクセスコントロールが可能になります。これらのツールのサポートは Linux ディストリビューションに依存し、Debian/Ubuntu は AppArmor をサポートし、RHEL/CentOS/Bottlerocket/Amazon Linux 2023 は SELinux をサポートします。SELinux の詳細については、「インフラストラクチャセキュリティ」セクションも参照してください。

AppArmor と SELinux はどちらも Kubernetes と統合されていますが、Kubernetes 1.28 以降では AppArmor プロファイルを注釈で指定する必要があります。SELinux ラベルはセキュリティコンテキストの SELinuxOptions フィールドから直接設定できます。

seccomp プロファイルと同様に、上記の Security Profiles Operator は、クラスター内のノードへのプロファイルのデプロイに役立ちます。(将来、プロジェクトは seccomp の場合と同様に AppArmor と SELinux のプロファイルを生成することも目標としています)。

推奨事項

Amazon GuardDuty を使用したランタイムのモニタリングと EKS 環境への脅威の検出

現在、EKS ランタイムを継続的にモニタリングし、EKS 監査ログを分析し、マルウェアやその他の疑わしいアクティビティをスキャンするソリューションがない場合、Amazon では、AWS 環境を保護するためのシンプルで高速、セキュア、スケーラブル、および費用対効果の高いワンクリック方法を必要とするお客様に Amazon GuardDuty を使用することを強くお勧めします。Amazon GuardDuty は、AWS CloudTrail 管理イベント、AWS CloudTrail イベントログ、VPC フローログ (Amazon EC2 インスタンスから)、Kubernetes 監査ログ、DNS ログなどの基本的なデータソースを分析して処理するセキュリティモニタリングサービスです。また、EKS ランタイムモニタリングも含まれています。悪意のある IP アドレスやドメインのリストなどの脅威インテリジェンスフィードを継続的に更新し、機械学習を使用して、AWS 環境内の予期しないアクティビティ、不正なアクティビティ、悪意のあるアクティビティを特定します。これには、権限のエスカレーション、公開された認証情報の使用、悪意のある IP アドレス、ドメインとの通信、Amazon EC2 インスタンスと EKS コンテナワークロードでのマルウェアの存在、疑わしい API アクティビティの検出などの問題が含まれます。GuardDuty は、GuardDuty コンソールまたは Amazon EventBridge で表示できるセキュリティ検出結果を生成することで、AWS 環境のステータスを通知します。GuardDuty は、検出結果を Amazon Simple Storage Service (S3) バケットにエクスポートし、AWS Security Hub や Detective などの他のサービスと統合するためのサポートも提供します。

この AWS Online Tech Talk「Amazon GuardDuty で Amazon EKS の脅威検出を強化 - AWS Online Tech Talks」を見て、これらの追加の EKS セキュリティ機能を数分でstep-by-step有効にする方法を確認してください。

オプション: ランタイムモニタリングにサードパーティーソリューションを使用する

Linux セキュリティに慣れていない場合、seccomp プロファイルと Apparmor プロファイルの作成と管理は難しい場合があります。習熟する時間がない場合は、サードパーティーの商用ソリューションの使用を検討してください。その多くは Apparmor や seccomp などの静的プロファイルを超えて移動し、機械学習を使用して疑わしいアクティビティをブロックまたは警告し始めています。これらのソリューションのいくつかは、 ツールセクションで以下にあります。その他のオプションについては、AWS Marketplace for Containers を参照してください。

seccomp ポリシーを作成する前に Linux 機能の追加/削除を検討する

機能には、syscalls が到達可能なカーネル関数のさまざまなチェックが含まれます。チェックが失敗した場合、syscall は通常エラーを返します。チェックは、特定の syscall の最初、または複数の異なる syscall を通じて到達可能な領域 (特定の特権ファイルへの書き込みなど) のカーネルのより深い方で行うことができます。一方、Seccomp は、実行前にすべての syscall に適用される syscall フィルターです。プロセスは、特定の syscall を実行する権利、または特定の syscall の特定の引数を取り消すことができるフィルターを設定できます。

seccomp を使用する前に、Linux 機能を追加/削除すると、必要なコントロールが得られるかどうかを検討してください。詳細については、「 コンテナの機能の設定」を参照してください。

Pod セキュリティポリシー (PSPs) を使用して目標を達成できるかどうか確認する

ポッドセキュリティポリシーは、過度に複雑になることなくセキュリティ体制を改善するさまざまな方法を提供します。seccomp プロファイルと Apparmor プロファイルを構築する前にPSPs で使用できるオプションを確認してください。

> **Warning**
>
> Kubernetes 1.25 以降、PSPsは削除され、Pod Security Admission Controller に置き換えられています。既存のサードパーティーの代替手段には、OPA/Gatekeeper や Kyverno などがあります。PSPs で一般的に見られるポリシーを実装するための Gatekeeper 制約と制約テンプレートのコレクションは、GitHub の Gatekeeper ライブラリリポジトリから取得できます。またPSPsの多くの置き換えは、ポッドセキュリティ標準の完全なコレクションを含む Kyverno ポリシーライブラリにあります。

ツールとリソース

- 開始する前に知っておくべき 7 つのこと
- AppArmor ローダー
- プロファイルを使用したノードのセットアップ
- Security Profiles Operator は、ユーザーが Kubernetes クラスターで SELinux、seccomp、AppArmor を簡単に使用できるようにする Kubernetes の機能強化です。実行中のワークロードからプロファイルを生成し、Pod で使用する Kubernetes ノードにプロファイルをロードする機能の両方を提供します。
- Inspektor Gadget を使用すると、Seccomp プロファイルの生成の支援など、Kubernetes でのランタイム動作のさまざまな側面を検査、トレース、プロファイリングできます。
- Aqua
- クォーリス
- スタックロックス
- Sysdig Secure
- プリズム
- SUSE オープンソースのゼロトラストコンテナセキュリティプラットフォームによる NeuVector は、プロセスプロファイルルールとファイルアクセスルールを提供します。

インフラストラクチャ (ホスト) の保護

コンテナイメージを保護することは重要ですが、コンテナイメージを実行するインフラストラクチャを保護することも同様に重要です。このセクションでは、ホストに対して直接起動される攻撃によるリスクを軽減するさまざまな方法について説明します。これらのガイドラインは、「ランタイムセキュリティ」セクションで説明されているガイドラインと組み合わせて使用する必要があります。

推奨事項

コンテナの実行に最適化された OS を使用する

Linux コンテナの実行用に設計された AWS の専用 OS である Flatcar Linux、Project Atomic、RancherOS、Bottlerocket の使用を検討してください。これには、攻撃対象領域の削減、起動時に検証されるディスクイメージ、SELinux を使用した強制的なアクセス許可の境界が含まれます。

または、Kubernetes ワーカーノードに EKS 最適化 AMI を使用します。EKS 最適化 AMI は定期的にリリースされ、コンテナ化されたワークロードを実行するために必要な最小限の OS パッケージとバイナリのセットが含まれています。

Hashicorp Packer を使用して Red Hat Enterprise Linux で実行されているカスタム Amazon EKS AMI を構築するために使用できるサンプル設定スクリプトについては、Amazon EKS AMI RHEL ビルド仕様を参照してください。このスクリプトをさらに活用して、STIG 準拠の EKS カスタム AMIs を構築できます。

ワーカーノード OS を最新の状態に保つ

Bottlerocket などのコンテナ最適化ホスト OS を使用するかどうかにかかわらず、EKS 最適化 AMIs などの Amazon マシンイメージでは、これらのホスト OS イメージを最新のセキュリティパッチで最新の状態に保つことをお勧めします。

EKS 最適化 AMIs の場合、CHANGELOG および/またはリリースノートチャネルを定期的にチェックし、更新されたワーカーノードイメージのクラスターへのロールアウトを自動化します。

インフラストラクチャをイミュータブルとして扱い、ワーカーノードの交換を自動化する

インプレースアップグレードを実行するのではなく、新しいパッチまたは更新が利用可能になったときにワーカーを置き換えます。これは、いくつかの方法でアプローチできます。グループ内のすべてのノードが最新の AMI に置き換えられるまで、ノードを順次コーディングおよびドレインしながら、最新の AMI を使用して既存の Auto Scaling グループにインスタンスを追加できます。または、すべてのノードが置き換えられるまで、古いノードグループからノードを順次接続およびドレインしながら、新しいノードグループにインスタンスを追加できます。EKS マネージド型ノードグループは最初のアプローチを使用し、新しい AMI が利用可能になると、 コンソールにワーカーをアップグレードするためのメッセージが表示されます。 には、最新の AMI を使用してノードグループを作成し、インスタンスが終了する前にノードグループからポッドを適切に分離およびドレインするためのメカニズム`eksctl`もあります。ワーカーノードの置き換えに別の方法を使用する場合は、新しい更新/パッチがリリースされ、コントロールプレーンがアップグレードされたときにワーカーを定期的に置き換える必要がある可能性が高いため、人的監視を最小限に抑えるプロセスを自動化することを強くお勧めします。

EKS Fargate を使用すると、AWS は更新が利用可能になると基盤となるインフラストラクチャを自動的に更新します。多くの場合、これはシームレスに行うことができますが、更新によってポッドが再スケジュールされることがあります。したがって、Fargate ポッドとしてアプリケーションを実行するときは、複数のレプリカを使用してデプロイを作成することをお勧めします。

kube-bench を定期的に実行して Kubernetes の CIS ベンチマークへの準拠を検証する

kube-bench は、Kubernetes の CIS ベンチマークに照らしてクラスターを評価する Aqua のオープンソースプロジェクトです。このベンチマークでは、アンマネージド Kubernetes クラスターを保護するためのベストプラクティスについて説明します。CIS Kubernetes ベンチマークには、コントロールプレーンとデータプレーンが含まれます。Amazon EKS はフルマネージド型のコントロールプレーンを提供するため、CIS Kubernetes Benchmark のすべての推奨事項が適用されるわけではありません。このスコープが Amazon EKS の実装方法を反映するように、AWS は CIS Amazon EKS Benchmark を作成しました。EKS ベンチマークは CIS Kubernetes Benchmark から継承され、EKS クラスターに関する特定の設定上の考慮事項を含むコミュニティからの追加の入力があります。

EKS クラスターに対して kube-bench を実行する場合は、Aqua Security の手順に従ってください。詳細については、「CIS Amazon EKS Benchmark の紹介」を参照してください。

ワーカーノードへのアクセスを最小限に抑える

SSH アクセスを有効にする代わりに、ホストにリモート接続する必要がある場合は SSM Session Manager を使用します。Session Manager では、紛失、コピー、共有される可能性のある SSH キーとは異なり、IAM を使用して EC2 インスタンスへのアクセスを制御できます。さらに、インスタンスで実行されたコマンドの監査証跡とログを提供します。

2020 年 8 月 19 日現在、マネージド型ノードグループはカスタム AMIs と EC2 起動テンプレートをサポートしています。これにより、SSM エージェントを AMI に埋め込むか、ワーカーノードがブートストラップされるときにインストールできます。Optimized AMI または ASG の起動テンプレートを変更しない場合は、この例のように DaemonSet を使用して SSM エージェントをインストールできます。

SSM ベースの SSH アクセスの最小 IAM ポリシー

`AmazonSSMManagedInstanceCore` AWS 管理ポリシーには、SSH アクセスを回避するだけの場合は、SSM Session Manager / SSM RunCommand に不要なアクセス許可が多数含まれています。特に懸念されるのは、 `ssm:GetParameter(s)`が パラメータストア内のすべてのパラメータ (AWS マネージド KMS キーが設定された SecureStrings を含む) にロールがアクセスできるようにするアクセス*許可です。

次の IAM ポリシーには、SSM Systems Manager を介したノードアクセスを有効にするための最小限のアクセス許可のセットが含まれています。

```
{
```

```
  "Version":"2012-10-17",
  "Statement": [
    {
      "Sid": "EnableAccessViaSSMSessionManager",
      "Effect": "Allow",
      "Action": [
        "ssmmessages:OpenDataChannel",
        "ssmmessages:OpenControlChannel",
        "ssmmessages:CreateDataChannel",
        "ssmmessages:CreateControlChannel",
        "ssm:UpdateInstanceInformation"
      ],
      "Resource": "*"
    },
    {
      "Sid": "EnableSSMRunCommand",
      "Effect": "Allow",
      "Action": [
        "ssm:UpdateInstanceInformation",
        "ec2messages:SendReply",
        "ec2messages:GetMessages",
        "ec2messages:GetEndpoint",
        "ec2messages:FailMessage",
        "ec2messages:DeleteMessage",
        "ec2messages:AcknowledgeMessage"
      ],
      "Resource": "*"
    }
  ]
}
```

このポリシーを設定し、Session Manager プラグインをインストールすると、 を実行できます。

```
aws ssm start-session --target [INSTANCE_ID_OF_EKS_NODE]
```

ノードにアクセスします。

> **Note**
>
> また、Session Manager の口グ記録を有効にするためのアクセス許可を追加することを検討することもできます。

プライベートサブネットにワーカーをデプロイする

ワーカーをプライベートサブネットにデプロイすることで、攻撃が頻繁に発生するインターネットへのワーカーの露出を最小限に抑えます。2020 年 4 月 22 日以降、マネージド型ノードグループのノードへのパブリック IP アドレスの割り当ては、デプロイ先のサブネットによって制御されます。これ以前は、マネージド型ノードグループのノードにはパブリック IP が自動的に割り当てられていました。ワーカーノードをパブリックサブネットにデプロイする場合は、AWS セキュリティグループルールを制限して公開を制限します。

Amazon Inspector を実行して、ホストの露出、脆弱性、ベストプラクティスからの逸脱を評価します。

Amazon Inspector を使用して、ノードへの意図しないネットワークアクセスや、基盤となる Amazon EC2 インスタンスの脆弱性を確認できます。

Amazon Inspector は、Amazon EC2 Systems Manager (SSM) エージェントがインストールされ、有効になっている場合にのみ、Amazon EC2 インスタンスに一般的な脆弱性と露出 (CVE) データを提供できます。このエージェントは、EKS 最適化 Amazon Linux AMIs を含む複数の Amazon マシンイメージ (AMI) にプリインストールされています。 AMIs SSM エージェントのステータスに関係なく、すべての Amazon EC2 インスタンスがスキャンされ、ネットワーク到達可能性の問題が検出されます。Amazon EC2 のスキャンの設定の詳細については、Amazon EC2 インスタンスのスキャン」を参照してください。

> **Important**
>
> Inspector は、Fargate ポッドの実行に使用されるインフラストラクチャでは実行できません。

代替案

SELinux を実行する

> **Note**
>
> Red Hat Enterprise Linux (RHEL)、CentOS、Bottlerocket、Amazon Linux 2023 で利用可能

SELinux は、コンテナを相互およびホストから分離するための追加のセキュリティレイヤーを提供します。SELinux を使用すると、管理者はすべてのユーザー、アプリケーション、プロセス、ファイルに必須のアクセスコントロール (MAC) を適用できます。これは、一連のラベルに基づいて特定のリソースに対して実行できるオペレーションを制限するバックストップと考えてください。EKS では、SELinux を使用してコンテナが互いのリソースにアクセスできないようにできます。

Container SELinux ポリシーは container-selinux パッケージで定義されます。Docker CE では、Docker (または他のコンテナランタイム) によって作成されたプロセスとファイルが制限されたシステムアクセスで実行されるように、このパッケージ (およびその依存関係) が必要です。コンテナは、 のエイリアスである`container_t`ラベルを使用します`svirt_lxc_net_t`。これらのポリシーにより、コンテナがホストの特定の機能にアクセスできなくなります。

Docker 用に SELinux を設定すると、Docker は自動的にワークロードをタイプ`container_t`としてラベル付けし、各コンテナに一意の MCS レベルを付与します。これにより、コンテナが互いに分離されます。より緩い制限が必要な場合は、SElinux で独自のプロファイルを作成し、ファイルシステムの特定の領域にコンテナアクセス許可を付与できます。これは、コンテナ/ポッドごとに異なるプロファイルを作成できるという点で PSPs に似ています。たとえば、一連の制限付きコントロールを持つ一般的なワークロードのプロファイルと、特権アクセスを必要とするモノのプロファイルを作成できます。

SELinux for Containers には、デフォルトの制限を変更するように設定できる一連のオプションがあります。必要に応じて、次の SELinux ブール値を有効または無効にできます。

ブール値	デフォルト	説明
`container_connect_any`	`off`	コンテナがホスト上の特権ポートにアクセスできるようにします。たとえば、ホスト上のポートを 443 または 80 にマッピングする必要があるコンテナがある場合です。
`container_manage_cgroup`	`off`	コンテナが cgroup 設定を管理できるようにします。たとえば、systemd を実行しているコンテナでは、こ

ブール値	デフォルト	説明
		れを有効にする必要があります。
container_use_cephfs	off	コンテナに ceph ファイルシステムの使用を許可します。

デフォルトでは、コンテナは で読み取り/実行/usrし、 からほとんどのコンテンツを読み取ることができます/etc。/var/lib/docker と の下のファイル/var/lib/containersにはラベルが付いていますcontainer_var_lib_t。デフォルトの完全なリストを表示するには、ラベルの container.fc ファイルを参照してください。

```
docker container run -it \
  -v /var/lib/docker/image/overlay2/repositories.json:/host/repositories.json \
  centos:7 cat /host/repositories.json
# cat: /host/repositories.json: Permission denied

docker container run -it \
  -v /etc/passwd:/host/etc/passwd \
  centos:7 cat /host/etc/passwd
# cat: /host/etc/passwd: Permission denied
```

でラベル付けされたファイルはcontainer_file_t、コンテナによって書き込み可能な唯一のファイルです。ボリュームマウントを書き込み可能にする場合は、最後に :zまたは :Z を指定する必要があります。

- :z は、コンテナが読み書きできるようにファイルのラベルを変更します。
- :Z は、コンテナのみが読み書きできるようにファイルのラベルを再付けします。

```
ls -Z /var/lib/misc
# -rw-r--r--. root root system_u:object_r:var_lib_t:s0   postfix.aliasesdb-stamp

docker container run -it \
  -v /var/lib/misc:/host/var/lib/misc:z \
  centos:7 echo "Relabeled!"

ls -Z /var/lib/misc
```

```
#-rw-r--r--. root root system_u:object_r:container_file_t:s0 postfix.aliasesdb-stamp
```

```
docker container run -it \
  -v /var/log:/host/var/log:Z \
  fluentbit:latest
```

Kubernetes では、ラベル変更が若干異なります。Docker でファイルに自動的にラベルを変更するのではなく、カスタム MCS ラベルを指定してポッドを実行できます。再ラベル付けをサポートするボリュームは、アクセスできるように自動的に再ラベル付けされます。一致する MCS ラベルを持つポッドは、ボリュームにアクセスできます。厳密な分離が必要な場合は、ポッドごとに異なる MCS ラベルを設定します。

```
securityContext:
  seLinuxOptions:
    # Provide a unique MCS label per container
    # You can specify user, role, and type also
    # enforcement based on type and level (svert)
    level: s0:c144:c154
```

この例では、コンテナがアクセスできるファイルに割り当てられた MCS ラベル`s0:c144:c154`に対応しています。

EKS では、FluentD などの特権コンテナの実行を許可するポリシーを作成し、ホストディレクトリのラベルを変更することなく、ホスト上の /var/log からの読み取りを許可する SELinux ポリシーを作成できます。同じラベルを持つポッドは、同じホストボリュームにアクセスできます。

CentOS 7 と RHEL 7 で SELinux が設定された Amazon EKS のサンプル AMIs を実装しました。これらの AMIs規制の厳しいお客様の要件を満たすサンプル実装を示すために開発されました。

> **Warning**
>
> SELinux は、タイプが閉じられていないコンテナを無視します。

ツールとリソース

- オンプレミスアプリケーションの SELinux Kubernetes RBAC と配送セキュリティポリシー
- Kubernetes の反復強化
- Audit2Allow

- SEAlert
- Udica を使用してコンテナの SELinux ポリシーを生成する は、Linux の機能、ポート、マウントポイントのコンテナ仕様ファイルを調べ、コンテナを適切に実行できるようにする一連の SELinux ルールを生成するツールについて説明します。
- さまざまな規制要件を満たすために OS を強化するための AMI 強化プレイブック
- Keiko Upgrade Manager は、ワーカーノードのローテーションを調整する Intuit のオープンソースプロジェクトです。
- Sysdig Secure
- eksctl

コンプライアンス

コンプライアンスは、AWS とそのサービスのコンシューマー間の責任共有です。一般的に、AWS は「クラウドのセキュリティ」を担当し、ユーザーは「クラウドのセキュリティ」を担当します。AWS とそのユーザーが何を担当しているかを示す行は、サービスによって異なります。例えば、Fargate では、AWS はデータセンター、ハードウェア、仮想インフラストラクチャ (Amazon EC2)、コンテナランタイム (Docker) の物理的なセキュリティを管理します。Fargate のユーザーは、コンテナイメージとそのアプリケーションを保護する責任があります。コンプライアンス標準に準拠する必要があるワークロードを実行する際の重要な考慮事項は何かについて、誰が責任を負うかを把握します。

次の表は、さまざまなコンテナサービスが準拠するコンプライアンスプログラムを示しています。

コンプライアンスプログラム	Amazon ECS オーケストレーター	Amazon EKS オーケストレーター	ECS Fargate	Amazon ECR
PCI DSS レベル 1	1	1	1	1
HIPAA 適格	1	1	1	1
SOC I	1	1	1	1
SOC II	1	1	1	1
SOC III	1	1	1	1

コンプライアンスプログラム	Amazon ECS オーケストレーター	Amazon EKS オーケストレーター	ECS Fargate	Amazon ECR
ISO 27001:2013	1	1	1	1
ISO 9001:2015	1	1	1	1
ISO 27017:2015	1	1	1	1
ISO 27018:2019	1	1	1	1
IRAP	1	1	1	1
FedRAMP Moderate (East/West)	1	1	0	1
FedRAMP High (GovCloud)	1	1	0	1
DOD CC SRG	1	DISA レビュー (IL5)	0	1
HIPAA BAA	1	1	1	1
MTCS	1	1	0	1
C5	1	1	0	1
K-ISMS	1	1	0	1
ENS High	1	1	0	1
OSPAR	1	1	0	1
HITRUST CSF	1	1	1	1

コンプライアンスステータスは時間の経過とともに変化します。最新のステータスについては、常に https://aws.amazon.com/compliance/services-in-scope/ を参照してください。

クラウド認定モデルとベストプラクティスの詳細については、AWS ホワイトペーパー「安全なクラウド導入のための認定モデル」を参照してください。

左へのシフト

左へのシフトの概念には、ソフトウェア開発ライフサイクルの早い段階でポリシー違反やエラーをキャッチすることが含まれます。セキュリティの観点から見ると、これは非常に有益です。たとえば、開発者は、アプリケーションがクラスターにデプロイされる前に、設定の問題を修正できます。このようなミスをキャッチすると、ポリシーに違反する設定がデプロイされるのを防ぐことができます。

コードとしてのポリシー

ポリシーは、 許可されている動作や禁止されている動作など、動作を管理するための一連のルールと考えることができます。たとえば、すべての Dockerfiles に、コンテナを非ルートユーザーとして実行する USER ディレクティブを含める必要があるというポリシーがあるとします。ドキュメントとして、このようなポリシーを検出して適用するのは難しい場合があります。また、要件の変化に応じて古くなることもあります。Policy as Code (PaC) ソリューションを使用すると、既知および永続的な脅威を検出、防止、削減、対処するセキュリティ、コンプライアンス、プライバシー制御を自動化できます。さらに、ポリシーをコーディングし、他のコードアーティファクトと同様に管理するためのメカニズムも提供します。このアプローチの利点は、DevOps および GitOps 戦略を再利用して、Kubernetes クラスターのフリート間でポリシーを管理および一貫して適用できることです。PaC オプションと PSP の将来については、「Pod Security」を参照してください。 PSPs

パイプラインで policy-as-code ツールを使用して、デプロイ前に違反を検出する

- OPA は、CNCF の一部であるオープンソースのポリシーエンジンです。これはポリシーの決定に使用され、言語ライブラリやサービスなど 、さまざまな方法で実行できます。OPA ポリシーは、Rego と呼ばれるドメイン固有言語 (DSL) で記述されます。多くの場合、Gatekeeper プロジェクトとして Kubernetes Dynamic Admission Controller の一部として実行されますが、OPA を CI/CD パイプラインに組み込むこともできます。これにより、デベロッパーはリリースサイクルの早い段階で設定に関するフィードバックを取得できます。これにより、本番環境に移行する前に問題を解決できます。一般的な OPA ポリシーのコレクションは、このプロジェクトの GitHub リポジトリにあります。
- Conftest は OPA 上に構築されており、Kubernetes 設定をテストするための開発者向けのエクスペリエンスを提供します。
- Kyverno は、Kubernetes 用に設計されたポリシーエンジンです。Kyverno では、ポリシーは Kubernetes リソースとして管理され、ポリシーを記述するために新しい言語は必要ありません。

これにより、kubectl、git、kustomize などの使い慣れたツールを使用してポリシーを管理できます。Kyverno ポリシーは、Kubernetes リソースを検証、変更、生成できるだけでなく、OCI イメージサプライチェーンのセキュリティを確保できます。Kyverno CLI を使用して、CI/CD パイプラインの一部としてポリシーをテストし、リソースを検証できます。すべての Kyverno コミュニティポリシーは Kyverno ウェブサイトにあります。Kyverno CLI を使用してパイプラインでテストを記述する例については、ポリシーリポジトリを参照してください。

ツールとリソース

- Amazon EKS セキュリティイマージョンワークショップ - 規制コンプライアンス
- kube-bench
- docker-bench-security
- AWS Inspector
- Kubernetes セキュリティレビュー Kubernetes 1.13.4 のサードパーティーセキュリティ評価 (2019)
- SUSE オープンソースのゼロトラストコンテナセキュリティプラットフォームによる NeuVector は、コンプライアンスレポートとカスタムコンプライアンスチェックを提供します。

インシデント対応とフォレンジック

インシデントに迅速に対応できれば、違反による損害を最小限に抑えることができます。疑わしい動作を警告できる信頼性の高いアラートシステムを持つことは、適切なインシデント対応計画の最初のステップです。インシデントが発生した場合は、影響を受けたコンテナを破棄して置き換えるか、コンテナを分離して検査するかをすばやく決定する必要があります。フォレンジック調査と根本原因分析の一環としてコンテナを分離することを選択した場合は、次の一連のアクティビティに従う必要があります。

インシデント対応計画の例

問題のある Pod とワーカーノードを特定する

最初のアクションは、ダメージを分離することです。まず、侵害が発生した場所を特定し、その Pod とそのノードを残りのインフラストラクチャから分離します。

ワークロード名を使用して問題のある Pod とワーカーノードを特定する

問題のあるポッドの名前と名前空間がわかっている場合は、次のようにポッドを実行しているワーカーノードを特定できます。

```
kubectl get pods <name> --namespace <namespace> -o=jsonpath='{.spec.nodeName}{"\n"}'
```

デプロイなどのワークロードリソースが侵害された場合、ワークロードリソースの一部であるすべてのポッドが侵害されている可能性があります。次のコマンドを使用して、ワークロードリソースのすべてのポッドとそれらが実行されているノードを一覧表示します。

```
selector=$(kubectl get deployments <name> \
 --namespace <namespace> -o json | jq -j \
'.spec.selector.matchLabels | to_entries | .[] | "\(.key)=\(.value)"')

kubectl get pods --namespace <namespace> --selector=$selector \
-o json | jq -r '.items[] | "\(.metadata.name) \(.spec.nodeName)"'
```

上記のコマンドはデプロイ用です。レプリカセット、ステートフルセットなど、他のワークロードリソースに対しても同じコマンドを実行できます。

サービスアカウント名を使用して問題のある Pod とワーカーノードを特定する

場合によっては、サービスアカウントが侵害されていることを特定できます。識別されたサービスアカウントを使用するポッドが侵害されている可能性があります。次のコマンドを使用して、サービスアカウントとそれらが実行されているノードを使用して、すべてのポッドを識別できます。

```
kubectl get pods -o json --namespace <namespace> | \
    jq -r '.items[] |
    select(.spec.serviceAccount == "<service account name>") |
    "\(.metadata.name) \(.spec.nodeName)"'
```

脆弱または侵害されたイメージとワーカーノードを持つ Pod を特定する

場合によっては、クラスターのポッドで使用されているコンテナイメージが悪意のあるものであったり、侵害されていることが判明することがあります。コンテナイメージにマルウェアが含まれていることが判明した場合、既知の不正なイメージであるか、悪用された CVE がある場合、コンテナイメージは悪意のあるイメージであるか、侵害されています。コンテナイメージが侵害されたすべてのポッドを考慮する必要があります。次のコマンドを使用して、実行中のイメージとノードを使用してポッドを識別できます。

```
IMAGE=<Name of the malicious/compromised image>

kubectl get pods -o json --all-namespaces | \
    jq -r --arg image "$IMAGE" '.items[] |
    select(.spec.containers[] | .image == $image) |
    "\(.metadata.name) \(.metadata.namespace) \(.spec.nodeName)"'
```

ポッドへのすべての入出力トラフィックを拒否するネットワークポリシーを作成してポッドを分離する

すべてのトラフィック拒否ルールは、ポッドへのすべての接続を切断することで、すでに進行中の攻撃を停止するのに役立ちます。次のネットワークポリシーは、 というラベルのポッドに適用されますapp=web。

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: default-deny
spec:
  podSelector:
    matchLabels:
      app: web
  policyTypes:
  - Ingress
  - Egress
```

> **Important**
>
> 攻撃者が基盤となるホストにアクセスした場合、ネットワークポリシーは無効である可能性があります。が発生したと思われる場合は、AWS セキュリティグループを使用して、侵害されたホストを他のホストから分離できます。ホストのセキュリティグループを変更する場合、そのホストで実行されているすべてのコンテナに影響することに注意してください。

必要に応じて、ポッドまたはワーカーノードに割り当てられた一時的なセキュリティ認証情報を取り消す

ワーカーノードに Pod が他の AWS リソースにアクセスできるようにする IAM ロールが割り当てられている場合は、それらのロールをインスタンスから削除して、攻撃によるさらなる損害を防ぎま

す。同様に、ポッドに IAM ロールが割り当てられている場合は、他のワークロードに影響を与えることなく、ロールから IAM ポリシーを安全に削除できるかどうかを評価します。

ワーカーノードをコーディングする

影響を受けるワーカーノードを遮断することで、影響を受けるノードにポッドをスケジュールしないようにスケジューラに通知します。これにより、他のワークロードを中断することなく、フォレンジック調査用のノードを削除できます。

> **Note**
>
> このガイダンスは、各 Fargate ポッドが独自のサンドボックス環境で実行される Fargate には適用されません。グルーニングの代わりに、すべての入出力トラフィックを拒否するネットワークポリシーを適用して、影響を受ける Fargate ポッドを隔離します。

影響を受けるワーカーノードで終了保護を有効にする

攻撃者は、影響を受けたノードを終了することで、自分の不正行為を消去しようとする可能性があります。終了保護を有効にすると、これを防ぐことができます。インスタンスのスケールイン保護は、スケールインイベントからノードを保護します。

> **Warning**
>
> スポットインスタンスで終了保護を有効にすることはできません。

問題のある Pod/Node に、アクティブな調査の一部であることを示すラベルを付けます。

これは、クラスター管理者が調査が完了するまで、影響を受ける Pods/Nodes を改ざんしないように警告するのに役立ちます。

ワーカーノードで揮発性アーティファクトをキャプチャする

- オペレーティングシステムのメモリをキャプチャします。これにより、Docker デーモン (または他のコンテナランタイム) とそのサブプロセスがコンテナごとにキャプチャされます。これは、LiME や Volatility などのツール、またはそれら上に構築される Amazon EC2 用自動フォレンジックオーケストレーターなどの高レベルのツールを使用して実現できます。

- 実行中のプロセスと開いているポートの netstat ツリーダンプを実行します。これにより、コンテナごとに docker デーモンとそのサブプロセスがキャプチャされます。
- 証拠が変更される前に、コマンドを実行してコンテナレベルの状態を保存します。コンテナランタイムの機能を使用して、現在実行中のコンテナに関する情報をキャプチャできます。例えば、Containerd では、以下を実行できます。
  - `crictl ps` 実行中のプロセス用。
  - `crictl logs CONTAINER` デーモンレベルの保持ログの 。

    代わりに nerdctl CLI を使用して containerd でも同じことが実現できます `docker` (例: `nerdctl inspect`)。コンテナランタイムに応じて、いくつかの追加のコマンドを使用できます。たとえば、Docker はコンテナファイルシステムの変更を確認したり`docker checkpoint`、揮発性メモリ (RAM) を含むすべてのコンテナ状態を保存`docker diff`したりする必要があります。containerd または CRI-O ランタイムの同様の機能の詳細については、この Kubernetes ブログ記事を参照してください。
- フォレンジックキャプチャのためにコンテナを一時停止します。
- インスタンスの EBS ボリュームをスナップショットします。

侵害された Pod またはワークロードリソースを再デプロイする

フォレンジック分析用にデータを収集したら、侵害されたポッドまたはワークロードリソースを再デプロイできます。

まず、侵害された脆弱性の修正をロールアウトし、新しい代替ポッドを開始します。次に、脆弱なポッドを削除します。

脆弱なポッドが上位レベルの Kubernetes ワークロードリソース (デプロイや DaemonSet など) によって管理されている場合、それらを削除すると新しいポッドがスケジュールされます。そのため、脆弱なポッドは再び起動されます。その場合は、脆弱性を修正した後に新しい代替ワークロードリソースをデプロイする必要があります。次に、脆弱なワークロードを削除する必要があります。

推奨事項

AWS セキュリティインシデント対応ホワイトペーパーを確認する

このセクションでは、セキュリティ違反の疑いに対処するための簡単な概要といくつかの推奨事項を示しますが、このトピックはホワイトペーパー「AWS セキュリティインシデント対応」で網羅されています。

セキュリティゲームデーを練習する

セキュリティ担当者を赤と青の 2 つのチームに分割します。赤チームはさまざまなシステムの脆弱性を調べることに集中し、青チームはそれらに対する防御を担当します。個別のチームを作成するのに十分なセキュリティ担当者がない場合は、Kubernetes の悪用に関する知識を持つ外部エンティティの採用を検討してください。

Kubesploit は、CyberArk の侵入テストフレームワークで、ゲームデーの実行に使用できます。クラスターの脆弱性をスキャンする他のツールとは異なり、kubesploit は実際の攻撃をシミュレートします。これにより、ブルーチームは攻撃への対応を練習し、その有効性を評価する機会が得られます。

クラスターに対してペネトレーションテストを実行する

独自のクラスターを定期的に攻撃すると、脆弱性や設定ミスを検出するのに役立ちます。開始する前に、クラスターに対してテストを実行する前に、ペネトレーションテストのガイドラインに従ってください。

ツールとリソース

- Kubernetes の侵入テストツールである kube-hunter。
- Gremlin は、アプリケーションやインフラストラクチャに対する攻撃をシミュレートするために使用できるカオスエンジニアリングツールキットです。
- Kubernetes インストールの攻撃と防御
- kubesploit
- SUSE オープンソースのゼロトラストコンテナセキュリティプラットフォームによる NeuVector は、脆弱性とリスクのレポートとセキュリティイベント通知を提供します。
- 高度な永続的な脅威
- Kubernetes の実用的な攻撃と防御
- RBAC アクセス許可を利用して Kubernetes クラスターを侵害する

イメージのセキュリティ

コンテナイメージは、攻撃に対する防御の最前線と見なしてください。安全性が低く、構築が不十分なイメージがあると、攻撃者はコンテナの境界を抜け出し、ホストにアクセスできるようになる可能性があります。ホストになると、攻撃者は機密情報にアクセスしたり、クラスター内または AWS ア

カウントで横方向に移動したりできます。以下のベストプラクティスは、このような状況が発生するリスクを軽減するのに役立ちます。

推奨事項

最小イメージを作成する

まず、コンテナイメージから不要なバイナリをすべて削除します。Dockerhub の不慣れなイメージを使用している場合は、コンテナの各レイヤーの内容を表示できる Dive などのアプリケーションを使用してイメージを検査します。SETUID ビットと SETGID ビットを持つすべてのバイナリは、権限の昇格に使用できるため削除し、悪意のある目的に使用できる nc や curl などのシェルやユーティリティをすべて削除することを検討してください。SETUID ビットと SETGID ビットを持つファイルは、次のコマンドで確認できます。

```
find / -perm /6000 -type f -exec ls -ld {} \;
```

これらのファイルから特別なアクセス許可を削除するには、コンテナイメージに次のディレクティブを追加します。

```
RUN find / -xdev -perm /6000 -type f -exec chmod a-s {} \; || true
```

これとは逆に、イメージのファング解除と呼ばれます。

マルチステージビルドを使用する

マルチステージビルドを使用することは、最小限のイメージを作成する方法です。多くの場合、マルチステージビルドは継続的インテグレーションサイクルの一部を自動化するために使用されます。たとえば、マルチステージビルドを使用してソースコードをリントしたり、静的コード分析を実行したりできます。これにより、デベロッパーはパイプラインの実行を待つ代わりに、すぐにフィードバックを得ることができます。マルチステージビルドは、コンテナレジストリにプッシュされる最終イメージのサイズを最小限に抑えることができるため、セキュリティの観点から魅力的です。コンテナイメージからビルドツールやその他の無関係なバイナリを省くことで、イメージのアタックサーフェスが減り、セキュリティ対策を強化することができます。マルチステージビルドの詳細については、Docker のマルチステージビルドドキュメントを参照してください。

コンテナイメージのソフトウェア部品表 (SBOMsを作成する

「ソフトウェア部品表」 (SBOM) は、コンテナイメージを構成するソフトウェアアーティファクトのネストされたインベントリです。SBOM は、ソフトウェアセキュリティとソフトウェアサプラ

イチェーンのリスク管理における重要な構成要素です。SBOM を生成して中央リポジトリに保存し、SBOMs脆弱性をスキャンすると、次の問題に対処できます。

- 可視性: コンテナイメージを構成するコンポーネントを理解します。中央リポジトリに保存すると、デプロイ後であってもいつでも SBOMs監査してスキャンし、ゼロデイ脆弱性などの新しい脆弱性を検出して対応できます。
- Provenance Verification: アーティファクトの発生場所と発生方法に関する既存の前提が真であり、ビルドまたは配信プロセス中にアーティファクトまたは付随するメタデータが改ざんされていないことを確認します。
- 信頼性: 特定のアーティファクトとそのコンテンツが、意図したとおりに動作するために信頼できることを保証します。つまり、 は目的に適しています。これには、コードを安全に実行できるかどうかの判断と、コードの実行に関連するリスクに関する情報に基づいた意思決定が含まれます。信頼性は、認証済みのパイプライン実行レポートと、認証済みの SBOM および認証済みの CVE スキャンレポートを作成して、このイメージが安全なコンポーネントで安全な手段 (パイプライン) を介して実際に作成されることをイメージのコンシューマーに保証することで保証されます。
- 依存関係の信頼検証: アーティファクトの依存関係ツリーを再帰的にチェックして、使用するアーティファクトの信頼性と出所を確認します。SBOMs のドリフトは、不正、信頼できない依存関係、侵入の試みなどの悪意のあるアクティビティを検出するのに役立ちます。

次のツールを使用して SBOM を生成できます。

- Amazon Inspector を使用して SBOMs を作成およびエクスポートできます。
- Anchore からの Syft は、SBOM の生成にも使用できます。脆弱性スキャンを迅速化するために、コンテナイメージ用に生成された SBOM をスキャンする入力として使用できます。SBOM とスキャンレポートは、レビューと監査の目的で Amazon ECR などの中央 OCI リポジトリにイメージをプッシュする前に、イメージにアタッチされます。

ソフトウェアサプライチェーンの保護の詳細については、CNCF Software Supply Chain Best Practices ガイドを参照してください。

イメージの脆弱性を定期的にスキャンする

仮想マシンと同様に、コンテナイメージには脆弱性のあるバイナリとアプリケーションライブラリが含まれているか、時間の経過とともに脆弱性が発生する可能性があります。エクスプロイトから保護する最善の方法は、イメージスキャナーでイメージを定期的にスキャンすることです。Amazon ECR に保存されているイメージは、プッシュまたはオンデマンド (24 時間に 1 回) でスキャンで

きます。ECR は現在、Basic と Enhanced の 2 種類のスキャンをサポートしています。ベーシックスキャンは、オープンソースのイメージスキャンソリューションである Clair を無償で活用します。拡張スキャンでは、Amazon Inspector を使用して、追加のコストで自動連続スキャンを提供します。イメージがスキャンされると、結果は EventBridge の ECR のイベントストリームに記録されます。ECR コンソール内からスキャンの結果を表示することもできます。高脆弱性または重大な脆弱性があるイメージは、削除または再構築する必要があります。デプロイされたイメージに脆弱性が生じた場合は、できるだけ早く交換する必要があります。

脆弱性のあるイメージがデプロイされている場所を知ることは、環境を安全に保つために不可欠です。イメージ追跡ソリューションを自分で構築することは可能ですが、この機能やその他の高度な機能をすぐに利用できる商用サービスが既にいくつかあります。

- Grype
- Palo Alto - Prisma Cloud (twistcli)
- Aqua
- Kubei
- Trivy
- Snyk

Kubernetes 検証ウェブフックを使用して、イメージに重大な脆弱性がないことを検証することもできます。検証ウェブフックは、Kubernetes API の前に呼び出されます。通常、ウェブフックで定義された検証基準に準拠していないリクエストを拒否するために使用されます。これは、ECR describeImageScanFindings API を呼び出して、ポッドが重大な脆弱性を持つイメージをプルしているかどうかを判断するサーバーレスウェブフックの例です。脆弱性が見つかった場合、ポッドは拒否され、CVEsされます。

認証を使用してアーティファクトの整合性を検証する

認証とは、パイプラインの実行、SBOM、脆弱性スキャンレポートなどの「述語」という何か を主張する暗号で署名された「ステートメント」であり、別のモノ、「件名」、つまり コンテナイメージについて当てはまります。

認証は、ユーザーがアーティファクトがソフトウェアサプライチェーンの信頼できるソースから来ていることを検証するのに役立ちます。例えば、コンテナイメージには、そのイメージに含まれるすべてのソフトウェアコンポーネントや依存関係がわからない場合があります。ただし、コンテナイメージのプロデューサーがどのソフトウェアが存在するかを信頼する場合、プロデューサーの認証を使用

してそのアーティファクトを信頼できます。つまり、自分で分析を行う代わりに、ワークフローでアーティファクトを安全に使用できます。

- 認証は、AWS Signer または Sigstore 共同署名を使用して作成できます。
- Kyverno などの Kubernetes アドミッションコントローラーを使用して認証を検証できます。
- コンテナイメージへの認証の作成やアタッチなどのトピックを含むオープンソースツールを使用した AWS でのソフトウェアサプライチェーン管理のベストプラクティスの詳細については、このワークショップを参照してください。

ECR リポジトリの IAM ポリシーを作成する

現在、組織が共有 AWS アカウント内で複数の開発チームを独立して運用することは珍しくありません。これらのチームがアセットを共有する必要がない場合は、各チームが操作できるリポジトリへのアクセスを制限する一連の IAM ポリシーを作成できます。これを実装する良い方法は、ECR 名前空間を使用することです。名前空間は、同様のリポジトリをグループ化する方法です。たとえば、チーム A のすべてのレジストリには team-a/ というプレフィックスを付けることができますが、チーム B のレジストリには team-b/ というプレフィックスを使用できます。アクセスを制限するポリシーは次のようになります。

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Sid": "AllowPushPull",
      "Effect": "Allow",
      "Action": [
        "ecr:GetDownloadUrlForLayer",
        "ecr:BatchGetImage",
        "ecr:BatchCheckLayerAvailability",
        "ecr:PutImage",
        "ecr:InitiateLayerUpload",
        "ecr:UploadLayerPart",
        "ecr:CompleteLayerUpload"
      ],
      "Resource": [
        "arn:aws:ecr:us-east-1:123456789012:repository/team-a/*"
      ]
    }
  ]
}
```

ECR プライベートエンドポイントの使用を検討する

ECR API にはパブリックエンドポイントがあります。したがって、リクエストが IAM によって認証および承認されている限り、ECR レジストリにはインターネットからアクセスできます。クラスター VPC にインターネットゲートウェイ (IGW) がないサンドボックス環境で操作する必要があるユーザーには、ECR のプライベートエンドポイントを設定できます。プライベートエンドポイントを作成すると、インターネット経由でトラフィックをルーティングするのではなく、プライベート IP アドレスを介して ECR API にプライベートにアクセスできます。このトピックの詳細については、「Amazon ECR インターフェイス VPC エンドポイント」を参照してください。

ECR のエンドポイントポリシーを実装する

のデフォルトのエンドポイントポリシーは、リージョン内のすべての ECR リポジトリへのアクセスを許可します。これにより、攻撃者/インサイダーはデータをコンテナイメージとしてパッケージ化し、別の AWS アカウントのレジストリにプッシュすることで、データを流出させることができます。このリスクを軽減するには、ECR リポジトリへの API アクセスを制限するエンドポイントポリシーを作成する必要があります。たとえば、次のポリシーでは、アカウント内のすべての AWS プリンシパルが に対してすべてのアクションを実行し、ECR リポジトリに対してのみアクションを実行することを許可します。

```
{
  "Statement": [
    {
      "Sid": "LimitECRAccess",
      "Principal": "*",
      "Action": "*",
      "Effect": "Allow",
      "Resource": "arn:aws:ecr:<region>:<account_id>:repository/*"
    }
  ]
}
```

AWS Organization の一部ではない IAM 原則によるイメージのプッシュ/プルを防ぐ新しい`PrincipalOrgID`属性を使用する条件を設定することで、これをさらに強化できます。詳細については、「aws:PrincipalOrgID」を参照してください。`com.amazonaws.<region>.ecr.dkr` と `com.amazonaws.<region>.ecr.api` エンドポイントの両方に同じポリシーを適用することをお勧めします。EKS は ECR から kube-proxy、coredns、aws-node のイメージをプルするため、レジストリのアカウント ID をエンドポイントポリシーのリソースのリストに追加するか、 `602401143452.dkr.ecr.us-west-2.amazonaws.com/` ポリシーを変更してアカウント

ID からのプルを許可し、プッシュを制限する必要があります。次の表は、EKS イメージの提供元となる AWS アカウントとクラスターリージョン間のマッピングを示しています。

アカウント番号	リージョン
602401143452	以下に示すものを除くすべての商用リージョン
—	—
800184023465	ap-east-1 - アジアパシフィック (香港)
558608220178	me-south-1 - 中東 (バーレーン)
918309763551	cn-north-1 - 中国 (北京)
961992271922	cn-northwest-1 - 中国 (寧夏)

エンドポイントポリシーの使用の詳細については、「VPC エンドポイントポリシーを使用して Amazon ECR アクセスを制御する」を参照してください。

ECR のライフサイクルポリシーを実装する

NIST アプリケーションコンテナセキュリティガイドでは、「レジストリ内の古いイメージ」のリスクについて警告しています。誤ったデプロイや露出を防ぐために、古くて脆弱でout-of-dateソフトウェアパッケージを持つ古いイメージは、時間の経過とともに削除する必要があることに注意してください。各 ECR リポジトリには、イメージの有効期限のルールを設定するライフサイクルポリシーを設定できます。AWS の公式ドキュメントでは、テストルールをセットアップし、評価して適用する方法について説明します。公式ドキュメントには、リポジトリ内のイメージをフィルタリングするさまざまな方法を示すライフサイクルポリシーの例がいくつかあります。

- イメージの経過時間または数によるフィルタリング
- タグ付きイメージまたはタグなしイメージによるフィルタリング
- 複数のルールまたは単一のルールでのイメージタグによるフィルタリング

???+ 警告 長時間実行されているアプリケーションのイメージが ECR から消去されると、アプリケーションの再デプロイ時または水平スケーリング時にイメージのプルエラーが発生する可能性があります。イメージライフサイクルポリシーを使用する場合は、デプロイとそれらが参照するイメージ

を最新の状態に保ち、リリース/デプロイの実行頻度を考慮した [イメージ] 有効期限ルールを常に作成するために、適切な CI/CD プラクティスが設定されていることを確認してください。

キュレーションされたイメージのセットを作成する

デベロッパーが独自のイメージを作成できるようにするのではなく、組織内のさまざまなアプリケーションスタック用に一連の審査済みイメージを作成することを検討してください。そうすることで、開発者は Dockerfile の作成方法を学ぶ必要がなくなり、コードの記述に集中できます。変更が Master にマージされると、CI/CD パイプラインはアセットを自動的にコンパイルし、アーティファクトリポジトリに保存して、そのアーティファクトを適切なイメージにコピーしてから、ECR などの Docker レジストリにプッシュできます。少なくとも、開発者が独自の Dockerfile を作成する一連のベースイメージを作成する必要があります。理想的には、Dockerhub からイメージをプルしないようにします。イメージ内の内容が 1/ に常にわかっているわけではなく、上位 1000 イメージの 2/約 5 に脆弱性があるためです。これらのイメージとその脆弱性のリストは、こちらで確認できます。

非ルートユーザーとして実行する USER ディレクティブを Dockerfiles に追加する

ポッドセキュリティセクションで説明したように、コンテナをルートとして実行することは避けてください。これは podSpec の一部として設定できますが、Dockerfiles への USERディレクティブを使用することをお勧めします。USER ディレクティブは、USER ディレクティブの後に表示される RUN、ENTRYPOINT、または CMD命令を実行するときに使用する UID を設定します。

Dockerfiles をリントする

リンティングを使用して、Dockerfiles が一連の事前定義されたガイドラインに従っていることを確認できます。たとえば、 USER ディレクティブの 包含や、すべてのイメージにタグを付けるという要件などです。dockerfile_lint は、一般的なベストプラクティスを検証する RedHat のオープンソースプロジェクトであり、Dockerfiles をリントするための独自のルールを構築するために使用できるルールエンジンが含まれています。これは CI パイプラインに組み込むことができます。ルールに違反する Dockerfile を使用してビルドすると、自動的に失敗します。

Scratch からイメージを構築する

イメージを構築するときは、コンテナイメージのアタックサーフェスを減らすことを主な目的とする必要があります。これを行う理想的な方法は、脆弱性を悪用するために使用できるバイナリのない最小限のイメージを作成することです。幸い、Docker には からイメージを作成するメカニズムがありますscratch。Go などの言語では、静的にリンクされたバイナリを作成し、次の例のように Dockerfile で参照できます。

```
############################
# STEP 1 build executable binary
############################
FROM golang:alpine AS builder# Install git.
# Git is required for fetching the dependencies.
RUN apk update && apk add --no-cache gitWORKDIR $GOPATH/src/mypackage/myapp/COPY . . #
 Fetch dependencies.
# Using go get.
RUN go get -d -v# Build the binary.
RUN go build -o /go/bin/hello

############################
# STEP 2 build a small image
############################
FROM scratch# Copy our static executable.
COPY --from=builder /go/bin/hello /go/bin/hello# Run the hello binary.
ENTRYPOINT ["/go/bin/hello"]
```

これにより、アプリケーションとそれ以外で構成されるコンテナイメージが作成され、非常に安全になります。

ECR でイミュータブルタグを使用する

イミュータブルタグを使用すると、イメージリポジトリへのプッシュごとにイメージタグを強制的に更新できます。これにより、攻撃者がイメージのタグを変更せずに悪意のあるバージョンでイメージを上書きするのを防ぐことができます。さらに、イメージを簡単かつ一意に識別する方法も提供します。

イメージ、SBOMs、パイプライン実行、脆弱性レポートに署名する

Docker が最初に導入されたとき、コンテナイメージを検証するための暗号化モデルはありませんでした。v2 では、Docker はイメージマニフェストにダイジェストを追加しました。これにより、イメージの設定をハッシュし、ハッシュを使用してイメージの ID を生成できるようになりました。イメージ署名を有効にすると、Docker エンジンはマニフェストの署名を検証し、コンテンツが信頼できるソースから生成され、改ざんが行われていないことを確認します。各レイヤーがダウンロードされると、エンジンはレイヤーのダイジェストを検証し、コンテンツがマニフェストで指定されたコンテンツと一致することを確認します。イメージ署名を効果的に行うと、イメージに関連付けられたデジタル署名の検証を通じて、安全なサプライチェーンを作成できます。

AWS Signer または Sigstore Cosign を使用して、コンテナイメージの署名、SBOMs の認証の作成、脆弱性スキャンレポート、パイプライン実行レポートを作成できます。これらの認証により、イ

メージの信頼性と完全性が保証され、実際には干渉や改ざんなしで信頼できるパイプラインによって作成され、イメージパブリッシャーによって検証および信頼された (SBOM 内の) 文書化されたソフトウェアコンポーネントのみが含まれていることが保証されます。これらの認証は、コンテナイメージにアタッチし、リポジトリにプッシュできます。

次のセクションでは、監査とアドミッションコントローラーの検証に認証されたアーティファクトを使用する方法について説明します。

Kubernetes アドミッションコントローラーを使用したイメージの整合性の検証

動的アドミッションコントローラーを使用してターゲット Kubernetes クラスターにイメージをデプロイする前に、イメージ署名、認証済みアーティファクトを自動的に検証し、アーティファクトのセキュリティメタデータがアドミッションコントローラーポリシーに準拠している場合にのみデプロイを承認できます。

例えば、イメージの署名、認証済み SBOM、認証済みパイプライン実行レポート、または認証済み CVE スキャンレポートを暗号化して検証するポリシーを作成できます。CVE スキャンに重要な CVE がないなど、ポリシーに条件を記述してレポート内のデータを確認できます CVEs 。これらの条件を満たすイメージに対してのみデプロイが許可され、他のすべてのデプロイはアドミッションコントローラーによって拒否されます。

アドミッションコントローラーの例は次のとおりです。

- キバーノ
- OPA ゲートキーパー
- ポーティリス語
- https://github.com/deislabs/ratify批准
- クローン病
- Grafeas チュートリアル
- 身分証明

コンテナイメージのパッケージを更新する

イメージ`apt-get update && apt-get upgrade`内のパッケージをアップグレードするには、Dockerfiles に RUN を含める必要があります。アップグレードではルートとして実行する必要がありますが、これはイメージ構築フェーズ中に発生します。アプリケーションは root として実行する必要はありません。更新をインストールし、USER ディレクティブを使用して別のユーザーに切

り替えることができます。ベースイメージが非ルートユーザーとして実行されている場合は、root と back に切り替えます。最新のセキュリティ更新プログラムのインストールは、ベースイメージのメンテナーだけに依存しないでください。

apt-get clean を実行して、インストーラファイルを から削除します/var/cache/apt/archives/。パッケージをインストールrm -rf /var/lib/apt/lists/*した後に を実行することもできます。これにより、インデックスファイルまたはインストール可能なパッケージのリストが削除されます。これらのコマンドはパッケージマネージャーごとに異なる場合があることに注意してください。例えば、次のようになります。

```
RUN apt-get update && apt-get install -y \
    curl \
    git \
    libsqlite3-dev \
    && apt-get clean && rm -rf /var/lib/apt/lists/*
```

ツールとリソース

- Amazon EKS セキュリティイマージョンワークショップ - イメージセキュリティ
- docker-slim 安全な最小限のイメージを構築する
- dockle Dockerfile が安全なイメージを作成するためのベストプラクティスと一致していることを確認します。
- Dockerfiles の dockerfile-lint ルールベースの linter
- hadolint スマート dockerfile linter
- Gatekeeper と OPA ポリシーベースのアドミッションコントローラー
- Kyverno Kubernetes ネイティブポリシーエンジン
- in-toto サプライチェーン内のステップの実行が意図されているかどうか、およびステップが適切なアクターによって実行されたかどうかをユーザーが確認できるようにします
- 公証人 コンテナイメージに署名するためのプロジェクト
- 公証人 v2
- Grafeas ソフトウェアサプライチェーンを監査および管理するためのオープンアーティファクトメタデータ API
- SUSE オープンソースのゼロトラストコンテナセキュリティプラットフォームによる NeuVector は、脆弱性、シークレット、コンプライアンスのコンテナ、イメージ、レジストリスキャンを提供します。

マルチアカウント戦略

AWS では、マルチアカウント戦略と AWS 組織を使用して、ビジネスアプリケーションとデータを分離および管理することをお勧めします。マルチアカウント戦略を使用することには多くの利点があります。

- AWS API サービスクォータの引き上げ。クォータは AWS アカウントに適用され、ワークロードに複数のアカウントを使用すると、ワークロードで使用できる全体的なクォータが増加します。
- よりシンプルな Identity and Access Management (IAM) ポリシー。ワークロードとワークロードをサポートするオペレーターに独自の AWS アカウントのみへのアクセスを許可すると、最小特権の原則を達成するためにきめ細かな IAM ポリシーを作成する時間が短縮されます。
- AWS リソースの分離が改善されました。設計上、アカウント内でプロビジョニングされたすべてのリソースは、他のアカウントでプロビジョニングされたリソースから論理的に分離されます。この分離境界により、アプリケーション関連の問題、設定ミス、または悪意のあるアクションのリスクを制限する方法が提供されます。1 つのアカウント内で問題が発生した場合に、他のアカウントに含まれるワークロードへの影響を軽減または排除することができます。
- AWS マルチアカウント戦略ホワイトペーパーで説明されているその他の利点

以下のセクションでは、集中型または分散型の EKS クラスターアプローチを使用して EKS ワークロードにマルチアカウント戦略を実装する方法について説明します。

マルチテナントクラスターのマルチワークロードアカウント戦略の計画

マルチアカウント AWS 戦略では、S3 バケット、ElastiCache クラスター、DynamoDB テーブルなどの特定のワークロードに属するリソースはすべて、そのワークロードのすべてのリソースを含む AWS アカウントに作成されます。これらはワークロードアカウントと呼ばれ、EKS クラスターはクラスターアカウントと呼ばれるアカウントにデプロイされます。クラスターアカウントについては、次のセクションで説明します。専用ワークロードアカウントにリソースをデプロイすることは、kubernetes リソースを専用名前空間にデプロイする場合と似ています。

その後、ワークロードアカウントは、必要に応じてソフトウェア開発ライフサイクルやその他の要件別にさらに分類できます。たとえば、特定のワークロードには、本番稼働用アカウント、開発用アカウント、または特定のリージョンでそのワークロードのインスタンスをホストするためのアカウントを含めることができます。詳細については、この AWS ホワイトペーパーを参照してください。

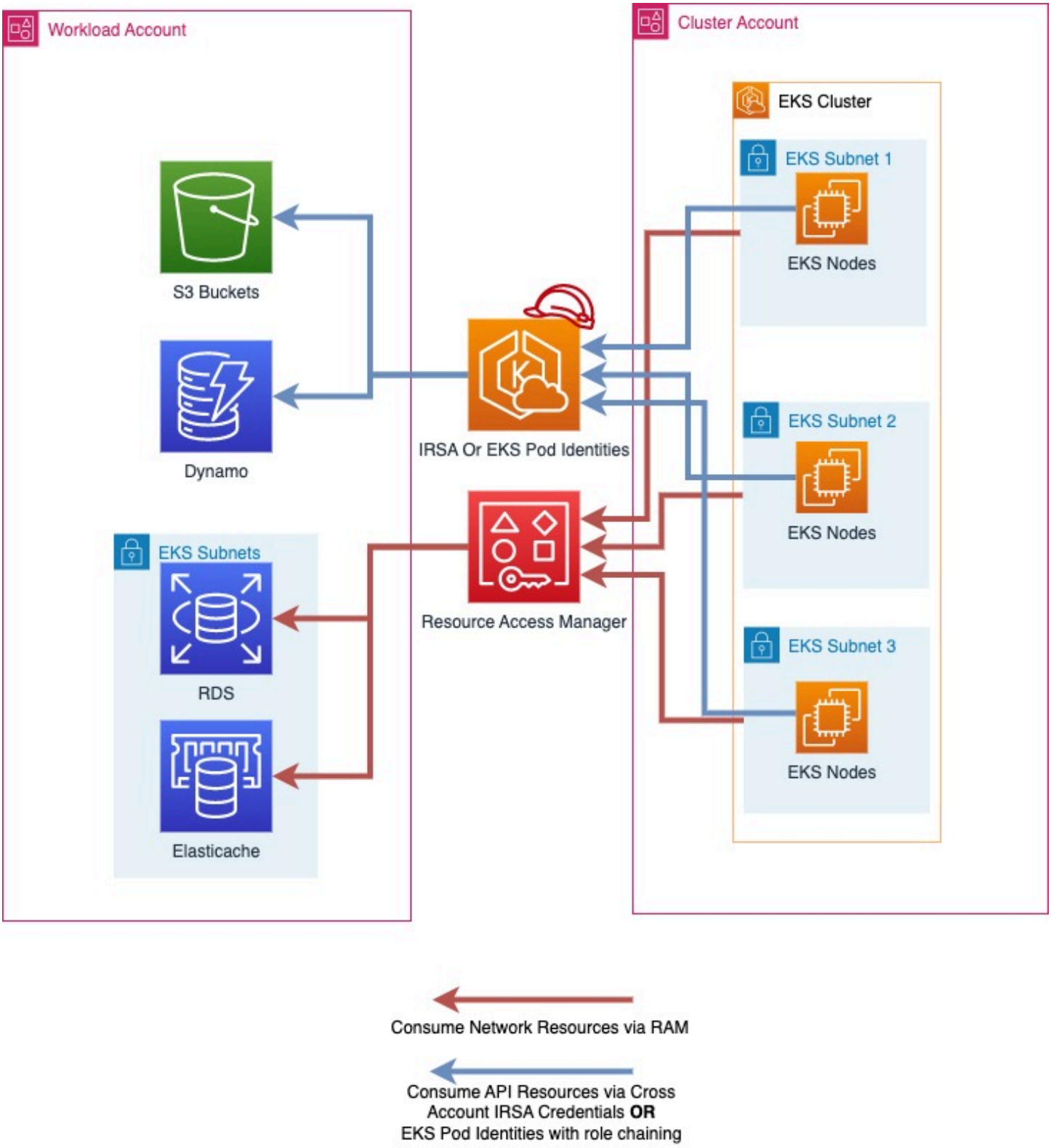
EKS マルチアカウント戦略を実装するときは、次のアプローチを採用できます。

一元化された EKS クラスター

このアプローチでは、EKS クラスターは という 1 つの AWS アカウントにデプロイされますCluster Account。サービスアカウント (IRSA) または EKS Pod ID の IAM ロールを使用して一時的な AWS 認証情報を配信し、AWS Resource Access Manager (RAM) を使用してネットワークアクセスを簡素化することで、マルチテナント EKS クラスターにマルチアカウント戦略を採用できます。 https://docs.aws.amazon.com/eks/latest/userguide/pod-identities.htmlクラスターアカウントには、VPC、サブネット、EKS クラスター、EC2/Fargate コンピューティングリソース (ワーカーノード) 、および EKS クラスターの実行に必要な追加のネットワーク設定が含まれます。

マルチテナントクラスターのマルチワークロードアカウント戦略では、AWS アカウントは通常、リソースグループを分離するためのメカニズムとして kubernetes 名前空間と一致します。マルチテナント EKS クラスターにマルチアカウント戦略を実装する場合は、EKS クラスター内のテナント分離のベストプラクティスに従う必要があります。

Cluster Accounts AWS 組織に複数の を持つことは可能であり、ソフトウェア開発ライフサイクルのニーズに合った複数の を持つことCluster Accountsがベストプラクティスです。非常に大規模なワークロードでは、すべてのワークロードCluster Accountsで使用できる十分な kubernetes と AWS サービスクォータを確保するために、複数の が必要になる場合があります。



|上記の図では、AWS RAM を使用してクラスターアカウントからワークロードアカウントにサブネットを共有しています。次に、EKS ポッドで実行されているワークロードは、IRSA または EKS

Pod Identity とロールチェーンを使用してワークロードアカウントのロールを引き受け、AWS リソースにアクセスします。

マルチテナントクラスターのマルチワークロードアカウント戦略の実装

AWS Resource Access Manager とのサブネットの共有

AWS Resource Access Manager (RAM) を使用すると、AWS アカウント間でリソースを共有できます。

AWS Organization で RAM が有効になっている場合は、クラスターアカウントからワークロードアカウントに VPC サブネットを共有できます。これにより、Amazon ElastiCache クラスターや Amazon Relational Database Service (RDS) データベースなどのワークロードアカウントが所有する AWS リソースを EKS クラスターと同じ VPC にデプロイし、EKS クラスターで実行されているワークロードで使用できるようになります。

RAM 経由でリソースを共有するには、クラスターアカウントの AWS コンソールで RAM を開き、「リソース共有」と「リソース共有の作成」を選択します。リソース共有に名前を付け、共有するサブネットを選択します。次へを再度選択し、サブネットを共有するワークロードアカウントの 12 桁のアカウント IDs を入力し、次へを再度選択して、リソース共有の作成をクリックして終了します。このステップの後、ワークロードアカウントはそれらのサブネットにリソースをデプロイできます。

RAM 共有は、プログラムで作成することも、Infrastructure as Code を使用して作成することもできます。

EKS ポッド ID と IRSA の選択

re:Invent 2023 で、AWS は一時的な AWS 認証情報を EKS 上のポッドに配信する簡単な方法として EKS Pod ID を起動しました。IRSA と EKS Pod ID はどちらも、一時的な AWS 認証情報を EKS ポッドに配信するための有効な方法であり、引き続きサポートされます。ニーズに最適な配信方法を検討する必要があります。

EKS クラスターと複数の AWS アカウントを使用する場合、IRSA は EKS クラスターが直接ホストされているアカウント以外の AWS アカウントでロールを直接引き受けることができますが、EKS Pod ID ではロールの連鎖を設定する必要があります。詳細な比較については、EKS ドキュメントを参照してください。

サービスアカウントの IAM ロールを使用した AWS API リソースへのアクセス

サービスアカウントの IAM ロール (IRSA) を使用すると、EKS で実行されているワークロードに一時的な AWS 認証情報を配信できます。IRSA を使用して、クラスターアカウントからワークロードアカウントの IAM ロールの一時的な認証情報を取得できます。これにより、クラスターアカウントの EKS クラスターで実行されているワークロードは、ワークロードアカウントでホストされている S3 バケットなどの AWS API リソースをシームレスに消費し、Amazon RDS データベースや Amazon EFS FileSystems などのリソースに IAM 認証を使用できます。

ワークロードアカウントで IAM 認証を使用する AWS API リソースおよびその他のリソースには、クロスアカウントアクセスが可能であり、明示的が有効になっている場合を除き、同じワークロードアカウントの IAM ロールの認証情報でのみアクセスできます。

クロスアカウントアクセスの IRSA の有効化

クラスターアカウントのワークロードの IRSA がワークロードアカウントのリソースにアクセスできるようにするには、まずワークロードアカウントに IAM OIDC ID プロバイダーを作成する必要があります。これは、ID プロバイダーがワークロードアカウントで作成されることを除いて、IRSA の設定と同じ手順で行うことができます。

次に、EKS でワークロードに IRSA を設定するときに、 ドキュメントと同じステップを実行できますが、「別のアカウントのクラスターから ID プロバイダーを作成する」セクションで説明されているように、ワークロードアカウントの 12 桁のアカウント ID を使用します。

これを設定すると、EKS で実行されているアプリケーションは、そのサービスアカウントを直接使用してワークロードアカウントのロールを引き受け、その中のリソースを使用できます。

EKS Pod ID を使用した AWS API リソースへのアクセス

EKS Pod Identity は、EKS で実行されているワークロードに AWS 認証情報を配信する新しい方法です。EKS ポッド ID は、EKS 上のポッドに AWS 認証情報を配信するために OIDC 設定を管理する必要がなくなるため、AWS リソースの設定を簡素化します。

クロスアカウントアクセスの EKS Pod ID の有効化

IRSA とは異なり、EKS Pod Identity は、EKS クラスターと同じアカウントのロールへのアクセスを直接付与するためにのみ使用できます。別の AWS アカウントのロールにアクセスするには、EKS Pod ID を使用するポッドでロールの連鎖を実行する必要があります。

ロールの連鎖は、さまざまな AWS SDKs で使用できる Process Credentials Provider を使用して、aws 設定ファイルを使用してアプリケーションプロファイルで設定できます。

credential_processは、次のようなプロファイルを設定するときに認証情報ソースとして使用できます。

```
# Content of the AWS Config file
[profile account_b_role]
source_profile = account_a_role
role_arn = arn:aws:iam::444455556666:role/account-b-role

[profile account_a_role]
credential_process = /eks-credential-processrole.sh
```

credentials_process によって呼び出されるスクリプトのソース:

```
#!/bin/bash
# Content of the eks-credential-processrole.sh
# This will retreive the credential from the pod identities agent,
# and return it to the AWS SDK when referenced in a profile
curl -H "Authorization: $(cat $AWS_CONTAINER_AUTHORIZATION_TOKEN_FILE)"
 $AWS_CONTAINER_CREDENTIALS_FULL_URI | jq -c '{AccessKeyId: .AccessKeyId,
 SecretAccessKey: .SecretAccessKey, SessionToken: .Token, Expiration: .Expiration,
 Version: 1}'
```

上記のように、アカウント A ロールと B ロールの両方を使用して aws 設定ファイルを作成し、ポッド仕様で AWS_CONFIG_FILE および AWS_PROFILE env vars を指定できます。EKS Pod ID ウェブフックは、env vars がポッド仕様に既に存在する場合、上書きされません。

```
# Snippet of the PodSpec
containers:
  - name: container-name
    image: container-image:version
    env:
    - name: AWS_CONFIG_FILE
      value: path-to-customer-provided-aws-config-file
    - name: AWS_PROFILE
      value: account_b_role
```

EKS ポッドアイデンティティを使用してロールを連鎖させるロール信頼ポリシーを設定する場合、EKS 固有の属性をセッションタグとして参照し、属性ベースのアクセスコントロール (ABAC) を使用して、ポッドが属する Kubernetes サービスアカウントなど、IAM ロールへのアクセスを特定の EKS ポッドアイデンティティセッションのみに制限できます。

これらの属性の一部は汎用的に一意ではない場合があることに注意してください。たとえば、2 つの EKS クラスターが同じ名前空間を持ち、1 つのクラスターが名前空間間で同じ名前のサービスアカウントを持つ場合があります。したがって、EKS Pod ID と ABAC を介してアクセスを許可する場合は、サービスアカウントへのアクセスを許可するときにクラスター ARN と名前空間を常に考慮することをお勧めします。

クロスアカウントアクセス用の ABAC および EKS Pod ID

マルチアカウント戦略の一環として EKS Pod Identity を使用して他のアカウントのロール (ロールの連鎖) を引き受ける場合は、別のアカウントにアクセスする必要があるサービスアカウントごとに一意の IAM ロールを割り当てるか、複数のサービスアカウントで共通の IAM ロールを使用し、ABAC を使用してアクセスできるアカウントを制御できます。

ABAC を使用して、ロールチェーンを使用してロールを別のアカウントに引き受けることができるサービスアカウントを制御するには、ロール信頼ポリシーステートメントを作成します。このステートメントでは、想定値が存在する場合にのみ、ロールセッションでロールを引き受けることができます。次のロール信頼ポリシーでは、 `eks-cluster-arn`および `kubernetes-namespace` タグがすべて想定値を持つ場合にのみ`kubernetes-service-account`、EKS クラスターアカウント (アカウント ID 111122223333) のロールがロールを引き受けることを許可します。

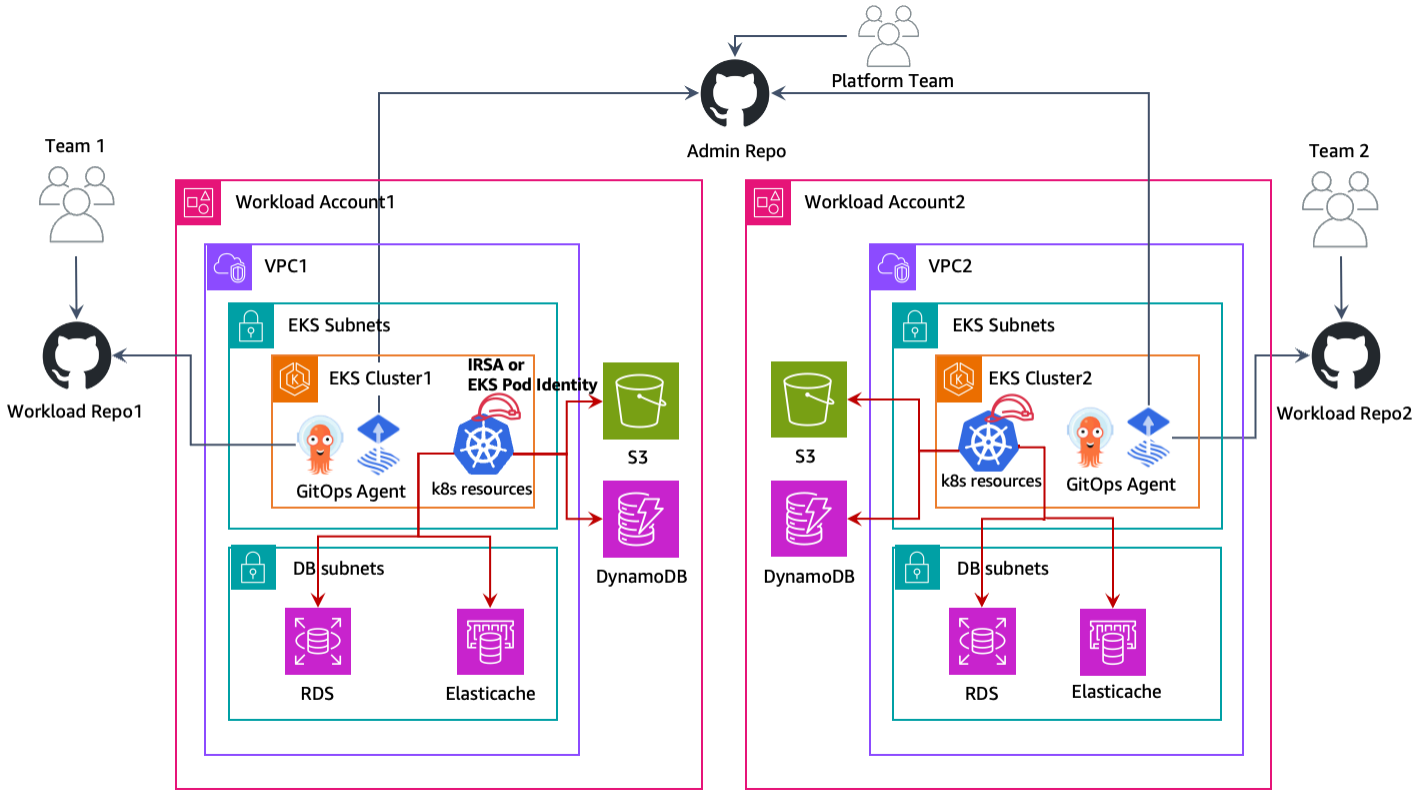
```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": {
                "AWS": "arn:aws:iam::111122223333:root"
            },
            "Action": "sts:AssumeRole",
            "Condition": {
                "StringEquals": {
                    "aws:PrincipalTag/kubernetes-service-account":
 "PayrollApplication",
                    "aws:PrincipalTag/eks-cluster-arn": "arn:aws:eks:us-
east-1:111122223333:cluster/ProductionCluster",
                    "aws:PrincipalTag/kubernetes-namespace": "PayrollNamespace"
                }
            }
        }
    ]
}
```

この戦略を使用する場合は、共通の IAM ロールに アクセスsts:AssumeRole許可のみがあり、他の AWS アクセスがないことを確認することがベストプラクティスです。

ABAC を使用する場合は、IAM ロールとユーザーにタグ付けできるユーザーを、厳密に必要なユーザーのみに制御することが重要です。IAM ロールまたはユーザーにタグを付けることができるユーザーは、EKS Pod Identity によって設定されるものと同じロール/ユーザーにタグを設定でき、権限をエスカレーションできる可能性があります。IAM ポリシー、またはサービスコントロールポリシー (SCP) を使用して、IAM ロールとユーザーのタグkubernetes-とeks-タグを設定するアクセス権を持つユーザーを制限できます。

分散型 EKS クラスター

このアプローチでは、EKS クラスターは各ワークロードの AWS アカウントにデプロイされ、Amazon S3 バケット、VPCs、Amazon DynamoDB テーブルなどの他の AWS リソースと一緒に動作します。各ワークロードアカウントは独立しており、自己完結性があり、各ビジネスユニット/アプリケーションチームによって運用されています。このモデルでは、 AI/ML クラスター 、バッチ処理、汎用など、さまざまなクラスター機能の再利用可能なブループリントを作成し、アプリケーションチームの要件に基づいてクラスターを 供給できます。アプリケーションチームとプラットフォームチームの両方がそれぞれの GitOps リポジトリから運用され、ワークロードクラスターへのデプロイを管理します。



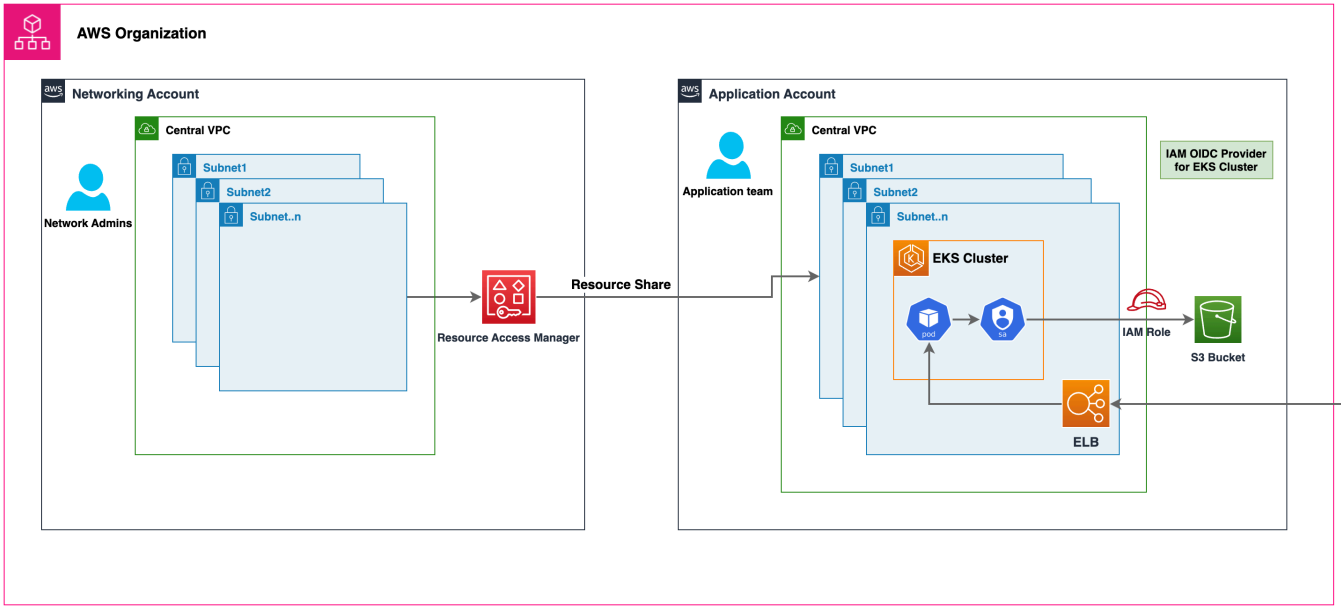
上記の図では、Amazon EKS クラスターおよびその他の AWS リソースがそれぞれのワークロードアカウントにデプロイされています。次に、EKS ポッドで実行されているワークロードは、IRSA または EKS Pod ID を使用して AWS リソースにアクセスします。

GitOps は、Git リポジトリでシステム全体が宣言的に記述されるように、アプリケーションとインフラストラクチャのデプロイを管理する方法です。これは、バージョン管理、イミュータブルアーティファクト、自動化のベストプラクティスを使用して、複数の Kubernetes クラスターの状態を管理できる運用モデルです。このマルチクラスターモデルでは、各ワークロードクラスターは複数の Git リポジトリでブートストラップされるため、各チーム (アプリケーション、プラットフォーム、セキュリティなど) はそれぞれの変更をクラスターにデプロイできます。

各アカウントのサービスアカウント (IRSA) または EKS Pod ID の IAM ロールを使用して、EKS ワークロードが一時的な aws 認証情報を取得して他の AWS リソースに安全にアクセスできるようにします。 https://docs.aws.amazon.com/eks/latest/userguide/pod-identities.htmlIAM ロールは、それぞれのワークロード AWS アカウントで作成され、一時的な IAM アクセスを提供するために k8s サービスアカウントにマッピングされます。したがって、このアプローチではクロスアカウントアクセスは必要ありません。IRSA の各ワークロードで を設定する方法については、サービスアカウントの IAM ロールのドキュメント、各アカウントの EKS ポッド ID を設定する方法については、EKS ポッド ID のドキュメントに従ってください。

一元化されたネットワーク

AWS RAM を使用して VPC サブネットをワークロードアカウントと共有し、Amazon EKS クラスターやその他の AWS リソースを起動することもできます。これにより、一元化されたネットワーク管理、簡素化されたネットワーク接続、および分散された EKS クラスターが可能になります。このアプローチの詳細なウォークスルーと考慮事項については、この AWS ブログを参照してください。



上記の図では、AWS RAM を使用して、中央ネットワークアカウントからワークロードアカウントにサブネットを共有しています。次に、EKS クラスターおよびその他の AWS リソースが、それぞれのワークロードアカウントのサブネットで起動されます。EKS ポッドは IRSA または EKS Pod ID を使用して AWS リソースにアクセスします。

一元化された EKS クラスターと分散された EKS クラスター

集中型または分散型で実行するかどうかは、要件によって異なります。この表は、各戦略の主な違いを示しています。

#	一元化された EKS クラスター	分散型 EKS クラスター
クラスター管理:	1 つの EKS クラスターの管理は、複数のクラスターを管理するよりも簡単です。	複数の EKS クラスターを管理する運用オーバーヘッドを減らすには、効率的なクラスター管理の自動化が必要です。
コスト効率:	EKS クラスターとネットワークリソースの再利用を許可し、コスト効率を高める	ワークロードごとにネットワークとクラスターのセットアップが必要で、追加のリソースが必要です

#	一元化された EKS クラスター	分散型 EKS クラスター
耐障害性:	クラスターに障害が発生した場合、集中型クラスター上の複数のワークロードが影響を受ける可能性があります。	クラスターに障害が発生した場合、損害はそのクラスターで実行されるワークロードのみに制限されます。他のすべてのワークロードは影響を受けません
分離とセキュリティ:	分離/ソフトマルチテナンシーは、 のような k8s ネイティブコンストラクトを使用して実現されますNamespaces 。ワークロードは、CPU、メモリなどの基盤となるリソースを共有する場合があります。AWS リソースは独自のワークロードアカウントに分離され、デフォルトでは他の AWS アカウントからはアクセスできません。	ワークロードがリソースを共有しない個々のクラスターとノードで実行されるため、コンピューティングリソースをより強力に分離できます。AWS リソースは独自のワークロードアカウントに分離され、デフォルトでは他の AWS アカウントからはアクセスできません。
パフォーマンスとスケーラビリティ:	ワークロードが非常に大規模になると、クラスターアカウントで kubernetes と AWS サービスクォータが発生する可能性があります。追加のクラスターアカウントをデプロイしてさらにスケールできます	より多くのクラスターと VPCsと、各ワークロードの k8 と AWS サービスクォータが利用可能になります。
ネットワーク:	クラスターごとに 1 つの VPC が使用されるため、そのクラスター上のアプリケーションの接続が簡単になります。	分散 EKS クラスター VPCs 間でルーティングを確立する必要があります

#	一元化された EKS クラスター	分散型 EKS クラスター
Kubernetes アクセス管理:	すべてのワークロードチームへのアクセスを提供し、kubernetes リソースが適切に分離されるように、クラスター内でさまざまなロールとユーザーを維持する必要があります。	各クラスターがワークロード/チーム専用であるため、アクセス管理を簡素化
AWS アクセス管理:	AWS リソースは、ワークロードアカウントの IAM ロールでのみデフォルトでアクセスできる独自のアカウントにデプロイされます。ワークロードアカウントの IAM ロールは、IRSA または EKS Pod ID を使用してクロスアカウントとして引き受けられます。	AWS リソースは、ワークロードアカウントの IAM ロールでのみデフォルトでアクセスできる独自のアカウントにデプロイされます。ワークロードアカウントの IAM ロールは、IRSA または EKS Pod ID を持つポッドに直接配信されます。

クラスターアクセス管理

Amazon EKS クラスターのセキュリティと整合性を維持するには、効果的なアクセス管理が不可欠です。このガイドでは、AWS IAM Identity Center (以前の AWS SSO) の使用に焦点を当てて、EKS アクセス管理のさまざまなオプションについて説明します。さまざまなアプローチを比較し、そのトレードオフについて話し合い、既知の制限と考慮事項を強調します。

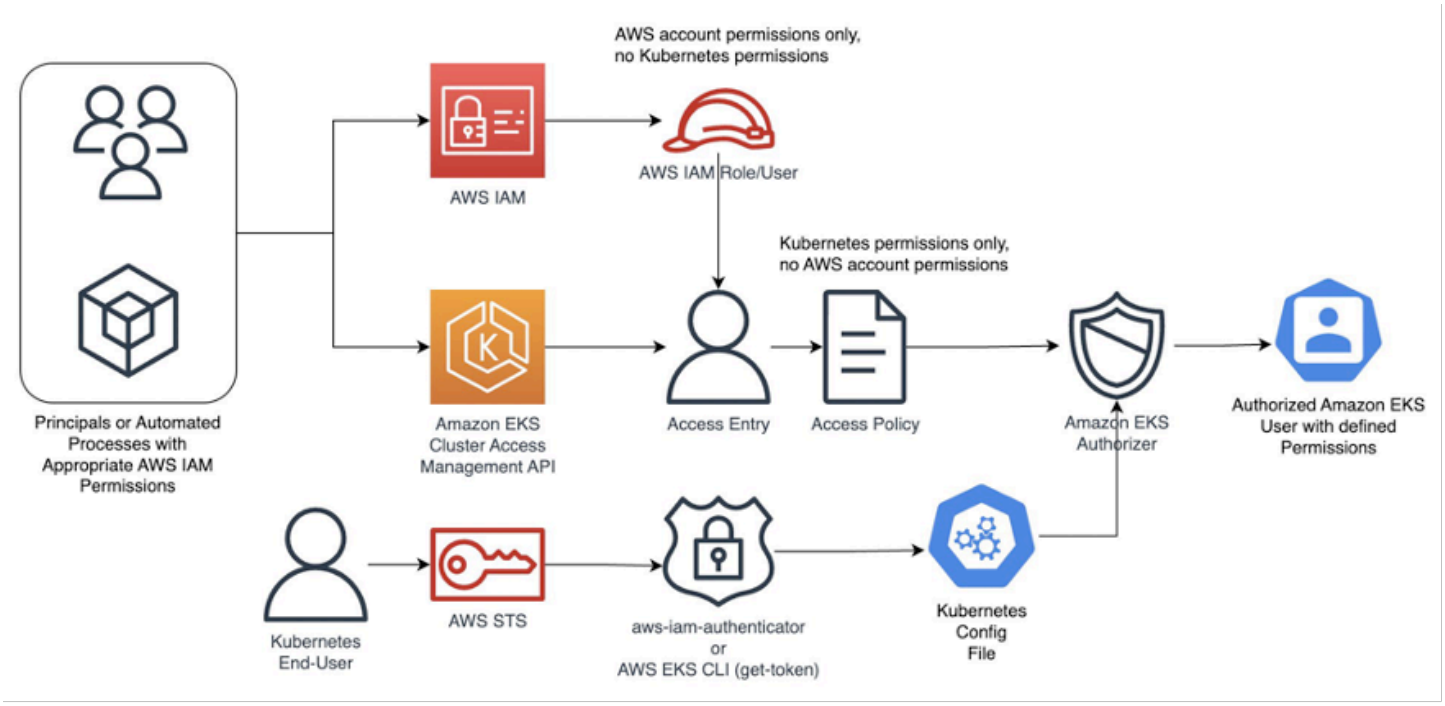
EKS アクセス管理オプション

> **Note**
>
> ConfigMap ベースのアクセス管理 (aws-auth ConfigMap) は廃止され、Cluster Access Management (CAM) API に置き換えられます。新しい EKS クラスターの場合は、CAM API を実装してクラスターアクセスを管理します。aws-auth ConfigMap を使用する既存のクラスターの場合は、CAM API を使用して に移行します。

オプション 1: クラスターアクセス管理 (CAM) API を使用する AWS IAM アイデンティティセンター

- 一元化されたユーザーとアクセス許可の管理
- 既存の ID プロバイダーとの統合 (Microsoft AD、Okta、PingId など)
- CAM API はアクセスエントリを使用して、AWS IAM プリンシパル (ユーザーまたはロール) を EKS クラスターにリンクします。これらのエントリは IAM Identity Center のマネージド ID と連携するため、管理者は Identity Center で定義されたユーザーとグループのクラスターアクセスを制御できます。

EKS クラスター認証フロー:



1. プリンシパル (ヒューマンユーザー) または自動プロセスは、適切な AWS アカウントのアクセス許可を提示することで AWS IAM を介して認証します。このステップでは、適切な AWS IAM プリンシパル (ロールまたはユーザー) にマッピングされます。
2. 次に、EKS アクセスエントリは、Kubernetes アクセス許可のみを含む適切なアクセスポリシーを定義することで、この IAM プリンシパルを Kubernetes RBAC プリンシパル (ユーザーまたはグループ) にマッピングします。

3. Kubernetes エンドユーザーがクラスターにアクセスしようとすると、その認証リクエストは aws-iam-authenticator または AWS EKS CLI によって処理され、kubeconfig ファイルのクラスターコンテキストに対して検証されます。
4. 最後に、EKS オーソライザーは、認証されたユーザーのアクセスエントリに関連付けられたアクセス許可を検証し、それに応じてアクセスを許可または拒否します。
   - API は Amazon EKS 固有のアクセスポリシーを使用して、各アクセスエントリの認可レベルを定義します。これらのポリシーは、IAM Identity Center で設定されたロールとアクセス許可にマッピングできるため、AWS のサービスおよび EKS クラスター間で一貫したアクセスコントロールを確保できます。

ConfigMap ベースのアクセス管理の利点:

1. 設定ミスのリスクを軽減: API ベースの直接管理により、ConfigMap の手動編集に関連する一般的なエラーを排除します。これにより、クラスターからユーザーをロックアウトする可能性のある誤った削除や構文エラーを防ぐことができます。
2. 最小特権の原則を強化: クラスター作成者 ID からクラスター管理者アクセス許可が不要になり、より詳細で適切なアクセス許可の割り当てが可能になります。このアクセス許可は、Break Glass ユースケースに追加できます。
3. 拡張セキュリティモデル: アクセスエントリが適用される前に、組み込みの検証を提供します。さらに、 は認証のために AWS IAM とのより緊密な統合を提供します。
4. オペレーションの合理化: AWS ネイティブツールを使用してアクセス許可を管理する、より直感的な方法を提供します。

ベストプラクティス:

1. AWS Organizations を使用して複数のアカウントを管理し、サービスコントロールポリシー (SCPs。
2. 最小特権の原則を実装するには、さまざまな EKS ロール (管理者、開発者、読み取り専用など) の特定のアクセス許可セットを作成します。
3. 属性ベースのアクセスコントロール (ABAC) を使用して、ユーザー属性に基づいてポッドにアクセス許可を動的に割り当てます。
4. アクセス許可を定期的に監査して確認します。

考慮事項/制限事項:

1. Identity Center によって生成されたロール ARNs にはランダムなサフィックスがあるため、静的設定での使用が困難です。
2. Kubernetes リソースレベルでのきめ細かなアクセス許可のサポートが制限されました。カスタム Kubernetes RBAC ロールには追加の設定が必要です。Kubernetes ネイティブ RBAC に加えて、EKS クラスターでの高度なアクセス許可管理に Kyverno を使用することを検討してください。

オプション 2: Kubernetes グループにマッピングされた AWS IAM ユーザー/ロール

メリット:

1. IAM アクセス許可をきめ細かく制御します。
2. 予測可能なロール ARN と静的ロール ARNs

デメリット:

1. ユーザーアカウントの管理オーバーヘッドの増加
2. 一元化された ID 管理の欠如
3. IAM エンティティの拡散の可能性

ベストプラクティス:

1. セキュリティと管理性を向上させるために、IAM ユーザーの代わりに IAM ロールを使用する
2. 一貫性と管理の容易性を確保するために、ロールの命名規則を実装する
3. IAM ポリシー条件を使用して、タグやその他の属性に基づいてアクセスを制限します。
4. アクセスキーを定期的にローテーションし、アクセス許可を確認します。

考慮事項/制限事項:

1. 多数のユーザーまたはロールを管理する際のスケーラビリティの問題
2. 組み込みのシングルサインオン機能がない

オプション 3: OIDC プロバイダー

メリット:

1. 既存の ID 管理システムとの統合
2. ユーザーアカウントの管理オーバーヘッドの削減

デメリット:

1. 追加の設定の複雑さ
2. 認証中のレイテンシーが増加する可能性
3. 外部 ID プロバイダーへの依存関係

ベストプラクティス:

1. OIDC プロバイダーを慎重に設定して、安全なトークン検証を確保します。
2. 有効期間の短いトークンを使用し、トークン更新メカニズムを実装します。
3. OIDC 設定を定期的に監査および更新します。

外部シングルサインオンプロバイダーを Amazon EKS と統合するリファレンス実装については、このガイドを参照してください。

考慮事項/制限事項:

1. IAM と比較して、AWS サービスとのネイティブ統合が制限されました。
2. EKS が署名キーを検出するには、OIDC プロバイダーの発行者 URL がパブリックにアクセス可能である必要があります。

ワークロードの AWS EKS Pod Identity と IRSA

Amazon EKS には、Amazon EKS クラスターで実行されるワークロードに AWS IAM アクセス許可を付与する 2 つの方法があります。サービスアカウント (IRSA) の IAM ロールと EKS ポッドアイデンティティです。

IRSA と EKS Pod Identity の両方が最小特権アクセス、認証情報の分離、監査可能性の利点を提供しますが、EKS Pod Identity はワークロードにアクセス許可を付与するために推奨される方法です。

EKS ポッドのアイデンティティと認証情報に関する詳細なガイダンスについては、セキュリティのベストプラクティスの「アイデンティティと認証情報」セクションを参照してください。

推奨事項

IAM Identity Center と CAM API を組み合わせる

- 管理の簡素化: クラスターアクセス管理 API を IAM アイデンティティセンターと組み合わせて使用すると、管理者は他の AWS のサービスと共に EKS クラスターアクセスを管理できるため、異なるインターフェイスを切り替えるか、ConfigMaps を手動で編集する必要がなくなります。
- アクセスエントリを使用して、クラスター外から IAM プリンシパルの Kubernetes アクセス許可を管理します。EKS API、AWS コマンドラインインターフェイス、AWS SDKs、AWS CloudFormation、AWS マネジメントコンソールを使用して、クラスターへのアクセスを追加および管理できます。つまり、クラスターを作成したのと同じツールでユーザーを管理できるということです。
- 詳細な Kubernetes アクセス許可は、アクセスエントリとアクセスポリシーを介して SSO ID に関連付けられた IAM プリンシパルを持つ Kubernetes ユーザーまたはグループをマッピングすることで適用できます。
- 開始するには、「アクセスエントリを使用するように認証モードを変更する」に従い、次に「既存の aws-auth ConfigMap エントリをアクセスエントリに移行する」に従います。

クラスターの自動スケーリングのベストプラクティス

このガイドでは、Auto Mode、Karpenter、Kubernetes Cluster Autoscaler のガイダンスなど、クラスターの自動スケーリングに関するアドバイスを提供します。

トピック

- EKS 自動モード
- Karpenter
- Cluster Autoscaler

EKS 自動モード

> **Tip**
>
> Amazon EKS Auto Mode の実践的な経験に興味がありますか? AWS Connected Community にサインアップして、AWS の専門家による次回の仮想 Kubernetes ワークショップに参加してください。

Amazon EKS Auto Mode は、Kubernetes インフラストラクチャ管理における大きな進化を表し、安全でスケーラブルなクラスターインフラストラクチャと AWS によって管理される統合された Kubernetes 機能を組み合わせています。このサービスは、フルマネージド型のワーカーノードオペレーションを提供するため、お客様が Managed Node Groups または AutoScaling グループ を設定する必要がなくなります。

EKS Auto Mode の主なアーキテクチャ上の違いは、ポッドリクエスト に応じて EC2 インスタンスを自動的にプロビジョニングする Karpenter ベースのシステムを使用することです。これらのインスタンスは、EBS CSI AMIs で実行されるため、インフラストラクチャは AWS によって真に管理されます。従来のスケーリング方法とは対照的に、次のようになります。

- 従来の Cluster Autoscaler (CAS) では、ノードグループを手動で管理する必要があり、ノードグループごとに 1 つのインスタンスタイプを持つノードのみを作成できます。
- セルフマネージド Karpenter は、EC2 フリート API を使用して柔軟性を高め、さまざまなインスタンスタイプをプロビジョニングできますが、カスタマー管理が必要です。
- EKS Auto Mode は、マネージド NodePools と NodeClasses を介してすべてのスケーリングオペレーションを自動的に処理します。

新しいシステムでは、いくつかの運用上の改善が導入されています。

- 手動ノードグループ設定なしの自動ポッド駆動型スケーリング
- Ingress リソースに基づいて ALB/NLB を自動的に作成する組み込みマネージドロードバランサーコントローラー
- 事前設定された Pod ID と統合されたセキュリティ機能
- 自動交換によるノードの最大ランタイムは 21 日

コストの観点からは、EKS Auto Mode は標準の EC2 料金を維持しながら、Auto Mode が管理するノードに対してのみ管理料金を追加します。重要なのは、Auto Mode マネージドノードを同じクラスター内のセルフマネージドノードと混在させることです。

AWS はほとんどの運用面を処理しますが、お客様はクラスターバージョン管理の責任を負い、ワーカーノードのローリング更新をトリガーする制御されたアップグレードを実行できます。

自動モードを使用する理由

Auto Mode は、Kubernetes と EKS の利点を必要とするユーザーを対象としていますが、Auto Scaling、ロードバランシング、ストレージなどの重要なプラットフォーム部分のアップグレードやインストール/メンテナンスなど、Kubernetes に関する運用上の負担を最小限に抑える必要があります。Auto Mode は、EKS を Kubernetes メンテナンスに伴う未分化の重労働の最小化にさらに踏み込む

よくある質問

EKS Auto Mode とオープンソース Karpenter の違いは何ですか?

EKS Auto Mode は、本稼働グレードの Kubernetes の実行をシンプルにする大規模な機能スイートです。これらの機能の 1 つは、フルマネージド型の Karpenter の自動スケーリングの利点です。オペレーションの観点から見ると、唯一の違いは、Karpenter ポッド自体のデプロイ、スケーリング、アップグレードを管理する必要がない EKS Auto Mode です。マネージド NodeClasses や NodePools などの他のすべてのオペレーションは、オープンソースの Karpenter と同様に機能します。

マネージド型ノードグループを Auto Mode マネージド型ノードと一緒に実行できますか?

はい、Auto Mode で提供されている Auto Scaling ノードとともに、マネージド型ノードグループを介して静的ノードを実行できます

クラスターを標準 EKS から EKS Auto Mode に移行できますか?

はい。既存のクラスターで EKS Auto Mode を有効にする手順については、AWS の公式ドキュメントを参照してください。

注意点: 1。Auto Mode を有効にしたら、Karpenter や AWS Load Balancer Controller 2 など、Auto Mode で現在管理されているインストール済みのコンポーネントをアンインストールします。インストールされているアドオンがup-to-dateであることを確認する必要があります。ドキュメントを参照してください。

EKS Auto Mode で NodePools を設定する方法

新しいクラスターは 2 つの NodePoolsで事前設定されます

汎用

```yaml
apiVersion: karpenter.sh/v1
kind: NodePool
metadata:
  generation: 1
  labels:
    app.kubernetes.io/managed-by: eks
  name: general-purpose
spec:
  disruption:
    budgets:
      - nodes: 10%
    consolidateAfter: 30s
    consolidationPolicy: WhenEmptyOrUnderutilized
  template:
    spec:
      expireAfter: 336h
      nodeClassRef:
        group: eks.amazonaws.com
        kind: NodeClass
        name: default
      requirements:
        - key: karpenter.sh/capacity-type
          operator: In
          values:
            - on-demand
        - key: eks.amazonaws.com/instance-category
          operator: In
          values:
            - c
            - m
            - r
        - key: eks.amazonaws.com/instance-generation
          operator: Gt
          values:
            - '4'
        - key: kubernetes.io/arch
          operator: In
          values:
            - amd64
        - key: kubernetes.io/os
          operator: In
          values:
            - linux
      terminationGracePeriod: 24h0m0s
```

この NodePool は、次の特性を持つノードを起動するように Karpenter に指示します。

1. 「オンデマンド」のキャパシティタイプ
2. C、M、または R のインスタンスタイプ
3. 4 のインスタンス生成
4. AMD アーキテクチャ
5. Linux OS

また、すべてのノードの 10% のみが特定の時点で中断状態になる可能性があること、およびノードが空であるか使用率が低い場合にのみ統合が行われることを宣言することで、スケールダウンロジックを定義します。

アタッチ

```
apiVersion: karpenter.sh/v1
kind: NodePool
metadata:
  generation: 1
  labels:
    app.kubernetes.io/managed-by: eks
  name: system
spec:
  disruption:
    budgets:
      - nodes: 10%
    consolidateAfter: 30s
    consolidationPolicy: WhenEmptyOrUnderutilized
  template:
    spec:
      expireAfter: 336h
      nodeClassRef:
        group: eks.amazonaws.com
        kind: NodeClass
        name: default
      requirements:
        - key: karpenter.sh/capacity-type
          operator: In
          values:
            - on-demand
        - key: eks.amazonaws.com/instance-category
          operator: In
          values:
            - c
            - m
            - r
        - key: eks.amazonaws.com/instance-generation
          operator: Gt
          values:
            - '4'
        - key: kubernetes.io/arch
          operator: In
          values:
            - amd64
            - arm64
        - key: kubernetes.io/os
          operator: In
          values:
            - linux
      taints:
        - effect: NoSchedule
```

この NodePool は、以下の違いを除いて「汎用」に似ています。

1. これにより、ARM アーキテクチャと AMD アーキテクチャを持つノードが可能になります。
2. CriticalAddonsOnly」に許容範囲がない限り、これらのノードを NoSchedule で汚染します。これは EKS アドオンによる内部使用を目的としています。

カスタム

必要に応じて、独自のカスタム NodePools を作成できます。NodePools の詳細については、Karpenter ドキュメントを参照してください。

新しいノードの起動時に Auto Mode で使用される AMI をカスタマイズできますか?

いいえ、現在サポートされている AMIs は Amazon が提供する Bottlerocket のみです

Kubernetes ホストにカスタムツールまたはエージェントをインストールするにはどうすればよいですか?

AMI のカスタマイズはサポートされていないため、セキュリティスキャンなどのホストレベルのソフトウェアが必要な場合は、ワークロードを Kubernetes DaemonSet としてデプロイする必要があります。

新しい EKS Auto Mode クラスターをプロビジョニングするときに、クラスターデータプレーンで実行されているコンポーネントは何ですか?

eksctl または AWS コンソールを使用してクラスターを作成する場合、EKS Auto Mode クラスターで実行されているポッドは Kubernetes Metrics Server ポッドのみです。Karpenter、AWS Load Balancer Controller、EBS CSI ドライバーなどの EKS Auto Mode の他のコンポーネントはすべてクラスター外で実行および管理されています。

新しい EKS Auto Mode クラスターをサポートするために実行されているマネージドコンポーネントは何ですか?

EKS Auto Mode は、本稼働グレードの Kubernetes に必要なデータプレーンのほとんどのデプロイを完全に自動化します。これには、以下が含まれます。

- Karpenter、クラスターのコンピューティングを自動スケーリング
- 自動 Elastic Load Balancer 統合を介して Kubernetes サービスを簡単に公開できるようにする AWS Load Balancer Controller

- EBS CSI
- VPC CNI
- EKS Pod Identity エージェント

がクラスターでポッドとして実行していた Auto Mode のコンポーネントをトラブルシューティングするにはどうすればよいですか?

EKS Auto Mode では、AWS Load Balancer Controller や Karpenter などの多くのコンポーネントがクラスターの外部で管理されるため、自己管理時と同じログを可視化することはできません。自動モード機能の機能をトラブルシューティングする必要がある場合は、AWS サポートチケットを作成します。

Karpenter

Karpenter は、Kubernetes クラスター内のノードライフサイクル管理を強化するように設計されたオープンソースプロジェクトです。ポッドの特定のスケジューリングニーズに基づいてノードのプロビジョニングとプロビジョニング解除を自動化し、効率的なスケーリングとコスト最適化を可能にします。主な機能は次のとおりです。

- リソースの制約により Kubernetes スケジューラがスケジュールできないポッドをモニタリングします。
- スケジューリング不可能なポッドのスケジューリング要件 (リソースリクエスト、ノードセレクタ、アフィニティ、許容範囲など) を評価します。
- これらのポッドの要件を満たす新しいノードをプロビジョニングします。
- 不要になったノードを削除します。

Karpenter を使用すると、テイント、ラベル、要件 (インスタンスタイプ、ゾーンなど) 、プロビジョニングされたリソースの合計の制限など、ノードのプロビジョニングに制約がある NodePools を定義できます。ワークロードをデプロイするときは、リソースリクエスト/制限、ノードセレクタ、ノード/ポッドアフィニティ、許容範囲、トポロジースプレッド制約など、ポッド仕様でさまざまなスケジューリング制約を指定できます。Karpenter は、これらの仕様に基づいて適切なサイズのノードをプロビジョニングします。

Karpenter を使用する理由

Karpenter のリリース前は、Kubernetes ユーザーは主に Amazon EC2 Auto Scaling グループと Kubernetes Cluster Autoscaler (CAS) に依存して、クラスターのコンピューティング性能を動的に調整していました。Karpenter を使用すると、Karpenter で得られる柔軟性と多様性を実現するために、数十のノードグループを作成する必要はありません。CAS とは異なり、Karpenter は Kubernetes バージョンにそれほど緊密に結合されておらず、AWS API と Kubernetes APIs の間でジャンプする必要はありません。

Karpenter は、インスタンスオーケストレーションの責任を 1 つのシステムに統合します。これにより、よりシンプルで安定したクラスター対応が可能になります。Karpenter は、以下の簡単な方法を提供することで、Cluster Autoscaler が提供するいくつかの課題を克服するように設計されています。

- ワークロードの要件に基づいてノードをプロビジョニングします。
- 柔軟な NodePool オプションを使用して、インスタンスタイプごとに多様なノード設定を作成します。Karpenter は、多くの特定のカスタムノードグループを管理する代わりに、単一の柔軟な NodePool で多様なワークロード容量を管理できます。
- ノードをすばやく起動し、ポッドをスケジューリングすることで、大規模なポッドスケジューリングを改善します。

Karpenter の使用に関する情報とドキュメントについては、karpenter.sh サイトを参照してください。

推奨事項

ベストプラクティスは、Karpenter 自体、NodePools、ポッドスケジューリングのセクションに分かれています。

Karpenter のベストプラクティス

以下のベストプラクティスでは、Karpenter 自体に関連するトピックについて説明します。

本番稼働用クラスターの AMIsロックダウンする

Karpenter が本番稼働用クラスターに使用する有名な Amazon マシンイメージ (AMIs) を固定することを強くお勧めします。エイリアス`amiSelector`を に設定して を使用するか`@latest`、リリース時にテストされていない AMIs をデプロイする他の方法を使用すると、本番稼働用クラスターでワークロードの障害やダウンタイムが発生するリスクがあります。そのため、非本番クラスターで新しい

バージョンをテストしている間は、本番クラスターのテスト済み作業バージョンの AMIs を固定することを強くお勧めします。たとえば、次のように NodeClass にエイリアスを設定できます。

```
amiSelectorTerms
  - alias: al2023@v20240807
```

Karpenter での AMIs の管理とピンダウンの詳細については、Karpenter ドキュメントの「AMI の管理 AMIs」を参照してください。

容量のニーズが変化するワークロードに Karpenter を使用する

Karpenter は、Autoscaling Groups (ASGs) や Managed Node Groups (MNGs) よりも Kubernetes ネイティブ APIs に近いスケーリング管理を提供します。 https://aws.amazon.com/blogs/containers/amazon-eks-cluster-multi-zone-auto-scaling-groups/ASGs と MNGsは、EC2 CPU 負荷などの AWS レベルのメトリクスに基づいてスケーリングがトリガーされる AWS ネイティブ抽象化です。Cluster Autoscaler は Kubernetes 抽象化を AWS 抽象化にブリッジしますが、特定のアベイラビリティーゾーンのスケジュールなど、そのための柔軟性が失われます。

Karpenter は AWS 抽象化のレイヤーを削除して、柔軟性の一部を Kubernetes に直接取り込みます。Karpenter は、需要が急増したり、コンピューティング要件が多様化したワークロードを持つクラスターに最適です。MNGsと ASGs は、より静的で整合性のあるワークロードを実行するクラスターに適しています。要件に応じて、動的に管理されたノードと静的に管理されたノードを組み合わせて使用できます。

次の場合は、他の Auto Scaling プロジェクトを検討してください。

Karpenter でまだ開発中の機能が必要です。Karpenter は比較的新しいプロジェクトであるため、Karpenter にまだ含まれていない機能が必要な場合は、当面は他の Auto Scaling プロジェクトを検討してください。

EKS Fargate またはノードグループに属するワーカーノードで Karpenter コントローラーを実行する

Karpenter は Helm チャートを使用してインストールされます。Helm チャートは、クラスターのスケーリングにコントローラーを使用する前に実行する必要があるデプロイとして Karpenter コントローラーとウェブフックポッドをインストールします。少なくとも 1 つのワーカーノードを持つ少なくとも 1 つの小さなノードグループをお勧めします。別の方法として、`karpenter`名前空間の Fargate プロファイルを作成して、これらのポッドを EKS Fargate で実行することもできます。これ

により、この名前空間にデプロイされたすべてのポッドが EKS Fargate で実行されます。Karpenter が管理するノードで Karpenter を実行しないでください。

Karpenter でのカスタム起動テンプレートのサポートなし

v1 APIs でのカスタム起動テンプレートのサポートはありません。カスタムユーザーデータを使用するか、EC2NodeClass でカスタム AMIs を直接指定できます。これを行う方法の詳細については、NodeClasses を参照してください。

ワークロードに合わないインスタンスタイプを除外する

クラスターで実行されているワークロードで必要でない場合は、 `node.kubernetes.io/instance-type`キーを使用して特定のインスタンスタイプを除外することを検討してください。

次の例は、大きな Graviton インスタンスのプロビジョニングを回避する方法を示しています。

```
- key: node.kubernetes.io/instance-type
  operator: NotIn
  values:
  - m6g.16xlarge
  - m6gd.16xlarge
  - r6g.16xlarge
  - r6gd.16xlarge
  - c6g.16xlarge
```

スポットの使用時に中断処理を有効にする

Karpenter はネイティブの中断処理をサポートし、スポットインスタンスの中断、スケジュールされたメンテナンスイベント、ワークロードを中断する可能性のあるインスタンスの終了/停止イベントなどの非自発的な中断イベントを処理できます。Karpenter は、ノードのこのようなイベントを検出すると、影響を受けるノードを事前に自動的に汚染、ドレイン、終了して、中断前にワークロードの正常なクリーンアップを開始します。2 分前の通知によるスポット中断の場合、Karpenter は新しいノードをすばやく起動し、インスタンスが再利用される前にポッドを移動できるようにします。中断処理を有効にするには、この目的のためにプロビジョニングされた `--interruption-queue` SQS キューの名前で CLI 引数を設定します。ここで説明するように、ノード終了ハンドラーと一緒に Karpenter 中断処理を使用することはお勧めしませんhttps://karpenter.sh/docs/faq/#interruption-handling。

チェックポイントやその他の形式の正常なドレイニングが必要で、シャットダウン前に 2 分かかるポッドは、クラスターで Karpenter の中断処理を有効にする必要があります。

アウトバウンドインターネットアクセスのない Amazon EKS プライベートクラスター

インターネットへのルートがない VPC に EKS クラスターをプロビジョニングする場合は、EKS ドキュメントに記載されているプライベートクラスターの要件に従って環境が設定されていることを確認する必要があります。さらに、VPC に STS VPC リージョンエンドポイントが作成されていることを確認する必要があります。そうでない場合、次のような エラーが表示されます。

```
{"level":"FATAL","time":"2024-02-29T14:28:34.392Z","logger":"controller","message":"Checking
 EC2 API connectivity, WebIdentityErr: failed to retrieve credentials
\ncaused by: RequestError: send request failed\ncaused by: Post
 \"https://sts.<region>.amazonaws.com/\": dial tcp 54.239.32.126:443: i/o
 timeout","commit":"596ea97"}
```

Karpenter Controller はサービスアカウント (IRSA) に IAM ロールを使用するため、これらの変更はプライベートクラスターで必要です。IRSA で設定されたポッドは、AWS Security Token Service (AWS STS) API を呼び出して認証情報を取得します。アウトバウンドインターネットアクセスがない場合は、VPC で AWS STS VPC エンドポイントを作成して使用する必要があります。

プライベートクラスターでは、SSM の VPC エンドポイントも作成する必要があります。Karpenter が新しいノードのプロビジョニングを試みると、起動テンプレートの設定と SSM パラメータをクエリします。VPC に SSM VPC エンドポイントがない場合、次のエラーが発生します。

```
{"level":"ERROR","time":"2024-02-29T14:28:12.889Z","logger":"controller","message":"Unable
 to hydrate the AWS launch template cache, RequestCanceled: request context canceled
\ncaused by: context canceled","commit":"596ea97","tag-key":"karpenter.k8s.aws/
cluster","tag-value":"eks-workshop"}
...
{"level":"ERROR","time":"2024-02-29T15:08:58.869Z","logger":"controller.nodeclass","message":"d
 amis from ssm, getting ssm parameter \"/aws/service/eks/optimized-ami/1.27/amazon-
linux-2/recommended/image_id\", RequestError: send request failed\ncaused by:
 Post \"https://ssm.<region>.amazonaws.com/\": dial tcp 67.220.228.252:443: i/o
 timeout","commit":"596ea97","ec2nodeclass":"default","query":"/aws/service/eks/
optimized-ami/1.27/amazon-linux-2/recommended/image_id"}
```

Price List Query API の VPC エンドポイントはありません。その結果、料金データは時間の経過とともに古くなります。Karpenter は、オンデマンドの料金データをバイナリに含めることでこれを回避しますが、Karpenter のアップグレード時にのみそのデータを更新します。料金データのリクエストに失敗すると、次のエラーメッセージが表示されます。

```
{"level":"ERROR","time":"2024-02-29T15:08:58.522Z","logger":"controller.pricing","message":"ret
 on-demand pricing data, RequestError: send request failed\ncaused by: Post
```

```
 \"https://api.pricing.<region>.amazonaws.com/\": dial tcp 18.196.224.8:443:
 i/o timeout; RequestError: send request failed\ncaused by: Post \"https://
api.pricing.<region>.amazonaws.com/\": dial tcp 18.185.143.117:443: i/o
 timeout","commit":"596ea97"}
```

完全にプライベート EKS クラスターで Karpenter を使用し、作成する VPC エンドポイントを確認するには、このドキュメントを参照してください。

NodePools の作成

以下のベストプラクティスでは、NodePools の作成に関連するトピックについて説明します。

次の場合に複数の NodePools を作成します。

異なるチームがクラスターを共有していて、異なるワーカーノードでワークロードを実行する必要がある場合、または異なる OS またはインスタンスタイプの要件がある場合は、複数の NodePools を作成します。たとえば、あるチームは Bottlerocket を使用し、別のチームは Amazon Linux を使用する場合があります。同様に、あるチームは、別のチームが必要としない高価な GPU ハードウェアにアクセスできる可能性があります。複数の NodePools を使用すると、各チームが最適なアセットを利用できるようになります。

相互に排他的または重み付けされた NodePoolsを作成する

一貫したスケジューリング動作を提供するために、相互に排他的または重み付けされた NodePools を作成することをお勧めします。一致しない場合、複数の NodePoolsが一致すると、Karpenter は使用するものをランダムに選択し、予期しない結果を引き起こします。複数の NodePoolsを作成するための便利な例は次のとおりです。

GPU を使用して NodePool を作成し、特別なワークロードをこれらの (高価な) ノードでのみ実行できるようにします。

```
# NodePool for GPU Instances with Taints
apiVersion: karpenter.sh/v1
kind: NodePool
metadata:
  name: gpu
spec:
  disruption:
    consolidateAfter: 1m
    consolidationPolicy: WhenEmptyOrUnderutilized
  template:
```

```
    metadata: {}
    spec:
      nodeClassRef:
        group: karpenter.k8s.aws
        kind: EC2NodeClass
        name: default
      expireAfter: Never
      requirements:
      - key: node.kubernetes.io/instance-type
        operator: In
        values:
        - p3.8xlarge
        - p3.16xlarge
      - key: kubernetes.io/os
        operator: In
        values:
        - linux
      - key: kubernetes.io/arch
        operator: In
        values:
        - amd64
      - key: karpenter.sh/capacity-type
        operator: In
        values:
        - on-demand
      taints:
      - effect: NoSchedule
        key: nvidia.com/gpu
        value: "true"
```

テイントの許容値によるデプロイ:

```
# Deployment of GPU Workload will have tolerations defined
apiVersion: apps/v1
kind: Deployment
metadata:
  name: inflate-gpu
spec:
    spec:
      tolerations:
      - key: "nvidia.com/gpu"
        operator: "Exists"
        effect: "NoSchedule"
```

別のチームの一般的なデプロイでは、NodePool 仕様に nodeAffinity を含めることができます。その後、デプロイは nodeSelectorTerms を使用して を照合できますbilling-team。

```
# NodePool for regular EC2 instances
apiVersion: karpenter.sh/v1
kind: NodePool
metadata:
  name: generalcompute
spec:
  template:
    metadata:
      labels:
        billing-team: my-team
    spec:
      nodeClassRef:
        group: karpenter.k8s.aws
        kind: EC2NodeClass
        name: default
      expireAfter: Never
      requirements:
      - key: node.kubernetes.io/instance-type
        operator: In
        values:
        - m5.large
        - m5.xlarge
        - m5.2xlarge
        - c5.large
        - c5.xlarge
        - c5a.large
        - c5a.xlarge
        - r5.large
        - r5.xlarge
      - key: kubernetes.io/os
        operator: In
        values:
        - linux
      - key: kubernetes.io/arch
        operator: In
        values:
        - amd64
      - key: karpenter.sh/capacity-type
        operator: In
        values:
```

```
        - on-demand
```

nodeAffinity を使用したデプロイ:

```
# Deployment will have spec.affinity.nodeAffinity defined
kind: Deployment
metadata:
  name: workload-my-team
spec:
  replicas: 200
    spec:
      affinity:
        nodeAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            nodeSelectorTerms:
              - matchExpressions:
                - key: "billing-team"
                  operator: "In"
                  values: ["my-team"]
```

タイマー (TTL) を使用してクラスターからノードを自動的に削除する

プロビジョニングされたノードのタイマーを使用して、ワークロードポッドがないノードや有効期限に達したノードを削除するタイミングを設定できます。ノードの有効期限はアップグレードの手段として使用できるため、ノードは廃止され、更新されたバージョンに置き換えられます。を使用してノードの有効期限を設定する方法については、Karpenter ドキュメントの「有効期限`spec.template.spec`」を参照してください。

Karpenter がプロビジョニングできるインスタンスタイプを過度に制約しないようにします。特にスポットを利用する場合です。

スポットを使用する場合、Karpenter は Price Capacity Optimized 配分戦略を使用して EC2 インスタンスをプロビジョニングします。この戦略では、起動するインスタンスの数について最も深いプールからインスタンスをプロビジョニングし、中断のリスクを最小限に抑えるように EC2 に指示します。次に、EC2 フリートは、これらのプールのうち最低価格のスポットインスタンスをリクエストします。Karpenter が利用できるインスタンスタイプが多いほど、EC2 はスポットインスタンスのランタイムを最適化できます。デフォルトでは、Karpenter はクラスターがデプロイされているリージョンとアベイラビリティーゾーンで EC2 が提供するすべてのインスタンスタイプを使用します。Karpenter は、保留中のポッドに基づいてすべてのインスタンスタイプのセットからインテリジェントに選択し、ポッドが適切なサイズで装備されたインスタンスにスケジュールされていること

を確認します。たとえば、ポッドに GPU が必要ない場合、Karpenter は GPU をサポートする EC2 インスタンスタイプにポッドをスケジュールしません。使用するインスタンスタイプが不明な場合は、Amazon ec2-instance-selector を実行して、コンピューティング要件に一致するインスタンスタイプのリストを生成できます。たとえば、CLI はメモリ vCPU、アーキテクチャ、リージョンを入力パラメータとして受け取り、これらの制約を満たす EC2 インスタンスのリストを提供します。

```
$ ec2-instance-selector --memory 4 --vcpus 2 --cpu-architecture x86_64 -r ap-
southeast-1
c5.large
c5a.large
c5ad.large
c5d.large
c6i.large
t2.medium
t3.medium
t3a.medium
```

スポットインスタンスを使用するときは、Karpenter にあまり多くの制約を課さないでください。そうすると、アプリケーションの可用性に影響する可能性があるためです。たとえば、特定のタイプのすべてのインスタンスが再利用され、それらを置き換える適切な代替手段がないとします。ポッドは、設定されたインスタンスタイプのスポット容量が補充されるまで保留状態のままになります。スポットプールは AZs 間で異なるため、インスタンスを異なるアベイラビリティーゾーンに分散することで容量不足エラーのリスクを減らすことができます。ただし、一般的なベストプラクティスは、Karpenter が スポットを使用するときにさまざまなインスタンスタイプのセットを使用できるようにすることです。

ポッドのスケジューリング

以下のベストプラクティスは、ノードのプロビジョニングに Karpenter を使用してクラスターにポッドをデプロイすることに関連しています。

EKS のベストプラクティスに従って高可用性を実現する

可用性の高いアプリケーションを実行する必要がある場合は、一般的な EKS ベストプラクティスの推奨事項に従ってください。ノードとゾーンにポッドを分散する方法の詳細については、Karpenter ドキュメントの「トポロジースプレッド」を参照してください。Disruption Budgets を使用して、ポッドの削除または削除を試みた場合に備えて、メンテナンスが必要な利用可能な最小ポッドを設定します。

レイヤード制約を使用して、クラウドプロバイダーから利用できるコンピューティング機能を制限する

Karpenter のレイヤード制約のモデルを使用すると、複雑な NodePool とポッドのデプロイ制約のセットを作成して、ポッドスケジューリングに最適なマッチングを取得できます。ポッド仕様がリクエストできる制約の例は次のとおりです。

- 特定のアプリケーションのみが利用可能なアベイラビリティーゾーンで を実行する必要があります。たとえば、特定のアベイラビリティーゾーンにある EC2 インスタンスで実行される別のアプリケーションと通信する必要があるポッドがあるとします。VPC 内のクロス AZ トラフィックを減らすことが目的の場合は、EC2 インスタンスが配置されている AZ にポッドを共同配置できます。このようなターゲット設定は、多くの場合、ノードセレクタを使用して行われます。ノードセレクタの詳細については、Kubernetes ドキュメントを参照してください。
- 特定の種類のプロセッサやその他のハードウェアが必要です。ポッドを GPU で実行する必要があるポッド仕様の例については、Karpenter ドキュメントの Accelerators セクションを参照してください。

コンピューティング支出をモニタリングする請求アラームを作成する

クラスターを自動的にスケーリングするように設定する場合は、支出がしきい値を超えたときに警告する請求アラームを作成し、Karpenter 設定にリソース制限を追加する必要があります。Karpenter でリソース制限を設定することは、Karpenter NodePool によってインスタンス化できるコンピューティングリソースの最大量を表すという点で、AWS Auto Scaling グループの最大容量を設定することに似ています。

> **Note**
>
> クラスター全体のグローバル制限を設定することはできません。制限は特定の NodePools に適用されます。

以下のスニペットでは、最大 1000 個の CPU コアと 1000Gi のメモリのみをプロビジョニングするように Karpenter に指示しています。Karpenter は、制限が満たされたか超えた場合にのみ容量の追加を停止します。制限を超えると、Karpenter コントローラーはコントローラーのログに`memory resource usage of 1001 exceeds limit of 1000`同様のメッセージを書き込みます。コンテナログを CloudWatch Logs にルーティングする場合は、メトリクスフィルターを作成して口

グ内の特定のパターンや用語を検索し、設定されたメトリクスのしきい値を超えたときに警告する CloudWatch アラームを作成できます。

Karpenter での制限の詳細については、Karpenter ドキュメントの「リソース制限の設定」を参照してください。

```
spec:
  limits:
    cpu: 1000
    memory: 1000Gi
```

Karpenter がプロビジョニングできるインスタンスタイプを制限または制限しない場合、Karpenter は必要に応じてクラスターにコンピューティング容量を追加し続けます。この方法で Karpenter を設定すると、クラスターを自由にスケールできますが、コストに大きな影響を与える可能性もあります。このため、請求アラームを設定することをお勧めします。請求アラームを使用すると、アカウントの計算された推定請求額が定義されたしきい値を超えたときに、アラートと事前通知を受け取ることができます (複数可) 。詳細については、「推定請求額をプロアクティブにモニタリングするための Amazon CloudWatch 請求アラームの設定」を参照してください。

また、コスト異常検出を有効にすることもできます。これは、機械学習を使用してコストと使用状況を継続的にモニタリングし、異常な支出を検出する AWS コスト管理機能です。詳細については、AWS コスト異常検出入門ガイドを参照してください。AWS Budgets で予算を作成するまで行った場合は、特定のしきい値を超えたときに通知するようにアクションを設定することもできます。予算アクションを使用すると、E メールを送信したり、SNS トピックにメッセージを投稿したり、Slack などのチャットボットにメッセージを送信したりできます。詳細については、「AWS Budgets アクションの設定」を参照してください。

karpenter.sh/do-not-disrupt 注釈を使用して、Karpenter がノードのプロビジョニングを解除できないようにします。

長時間実行されているバッチジョブやステートフルアプリケーションなど、Karpenter がプロビジョニングしたノードで重要なアプリケーションを実行していて、ノードの TTL の有効期限が切れている場合、インスタンスが終了するとアプリケーションは中断されます。ポッドに`karpenter.sh/do-not-disrupt`注釈を追加することで、ポッドが終了するか`karpenter.sh/do-not-disrupt`注釈が削除されるまでノードを保持するように Karpenter に指示します。詳細については、「Distruption documentation」を参照してください。

ノードに残っているデーモンセット以外のポッドのみがジョブに関連付けられている場合、Karpenter はジョブのステータスが成功または失敗する限り、それらのノードをターゲットにして終了できます。

統合を使用する場合、すべての非 CPU リソースの requests=limits を設定する

ポッドリソースリクエストとノード上の割り当て可能なリソースの量を比較することで、一般的な作業の統合とスケジューリングを行います。リソースの制限は考慮されません。たとえば、メモリ制限がメモリリクエストより大きいポッドは、リクエストを超えてバーストする可能性があります。同じノード上の複数のポッドが同時にバーストすると、メモリ不足 (OOM) 状態が原因で一部のポッドが終了する可能性があります。統合すると、リクエストのみを考慮してポッドをノードにパックする機能があるため、これが発生する可能性が高くなります。

LimitRanges を使用してリソースリクエストと制限のデフォルトを設定する

Kubernetes はデフォルトのリクエストや制限を設定しないため、コンテナが基盤となるホスト、CPU、メモリからリソースを消費することはバインドされません。Kubernetes スケジューラは、ポッドの合計リクエスト数 (ポッドのコンテナからの合計リクエスト数またはポッドの Init コンテナからの合計リソースのうち高い方) を調べて、ポッドをスケジュールするワーカーノードを決定します。同様に、Karpenter はポッドのリクエストを考慮して、プロビジョニングするインスタンスのタイプを決定します。リソースリクエストが一部のポッドで指定されていない場合は、制限範囲を使用して名前空間に適切なデフォルトを適用できます。

「名前空間のデフォルトのメモリリクエストと制限を設定する」を参照してください。

すべてのワークロードに正確なリソースリクエストを適用する

Karpenter は、ワークロードの要件に関する情報が正確であれば、ワークロードに最適なノードを起動できます。これは、Karpenter の統合機能を使用する場合に特に重要です。

「すべてのワークロードのリソースリクエスト/制限の設定とサイズ設定」を参照してください。

CoreDNS レコメンデーション

CoreDNS の設定を更新して信頼性を維持する

Karpenter が管理するノードに CoreDNS ポッドをデプロイする場合、需要に合わせて新しいノードを迅速に終了/作成するという Karpenter の動的な性質を考慮すると、次のベストプラクティスに従うことをお勧めします。

CoreDNS lameduck の期間

CoreDNS 準備状況プローブ

これにより、DNS クエリが、まだ準備中または終了済みの CoreDNS Pod に転送されないようになります。

Karpenter の設計図

Karpenter はアプリケーションファーストのアプローチで のコンピューティングキャパシティを Kubernetes データプレーンにプロビジョニングするため、ワークロードを適切に設定する方法を疑問視する一般的なシナリオがあります。Karpenter ブループリントは、ここで説明するベストプラクティスに従った一般的なワークロードシナリオのリストを含むリポジトリです。Karpenter が設定された EKS クラスターを作成し、リポジトリに含まれる各ブループリントをテストするために必要なすべてのリソースがあります。さまざまな設計図を組み合わせて、ワークロードに必要な設計図を最終的に作成できます (複数可) 。

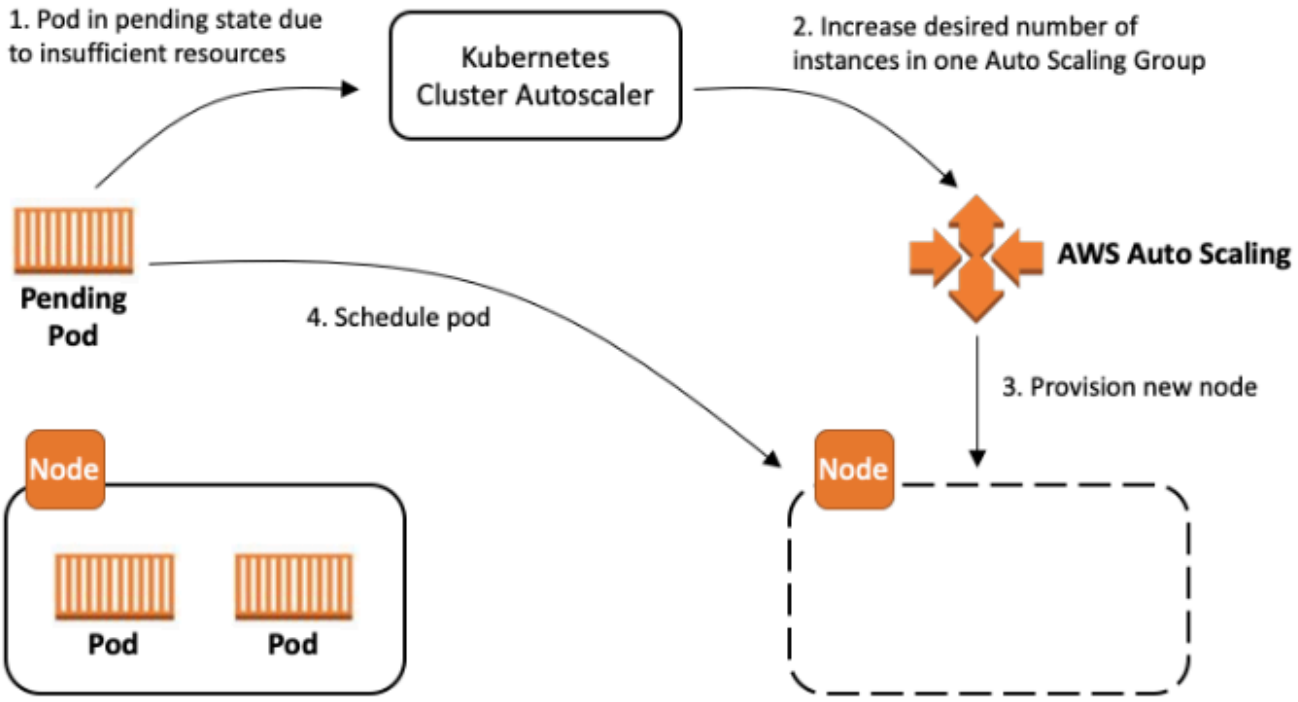
その他のリソース

- Karpenter Immersion Day ワークショップ
- Karpenter コスト最適化ワークショップ
- EKS ワークショップ - Karpenter
- Karpenter と Cluster Autoscaler
- re:Invent 2023 での Karpenter セッション
- チュートリアル: Amazon EC2 スポットと Karpenter を使用して Kubernetes クラスターをより少ない時間実行する

Cluster Autoscaler

概要:

Kubernetes Cluster Autoscaler は、SIG Autoscaling によって維持される一般的な Cluster Autoscaling ソリューションです。リソースを浪費することなく、ポッドをスケジュールするのに十分なノードがクラスターにあることを確認する責任があります。スケジュールに失敗したポッドと使用率の低いノードを監視します。次に、変更をクラスターに適用する前に、ノードの追加または削除をシミュレートします。Cluster Autoscaler 内の AWS クラウドプロバイダーの実装は、EC2 Auto Scaling グループの `.DesiredReplicas`フィールドを制御します。



このガイドでは、Cluster Autoscaler を設定し、組織の要件を満たす最適なトレードオフのセットを選択するためのメンタルモデルを提供します。最適な設定は 1 つではありませんが、パフォーマンス、スケーラビリティ、コスト、可用性をトレードオフできる一連の設定オプションがあります。さらに、このガイドでは、AWS の設定を最適化するためのヒントとベストプラクティスを提供します。

用語集

このドキュメントでは、以下の用語が頻繁に使用されます。これらの用語は幅広い意味を持つことができますが、本書の目的上、以下の定義に限定されます。

スケーラビリティとは、Kubernetes クラスターのポッドとノードの数が増えるにつれて Cluster Autoscaler がどの程度うまく機能するかを指します。スケーラビリティの制限に達すると、Cluster Autoscaler のパフォーマンスと機能が低下します。Cluster Autoscaler がスケーラビリティの制限を超えると、クラスター内のノードを追加または削除できなくなる可能性があります。

パフォーマンスとは、Cluster Autoscaler がスケーリングの決定を行い、実行できる速度を指します。完全にパフォーマンスの高い Cluster Autoscaler は、ポッドがスケジュール不可能になるなど、刺激に応じて即座に決定を行い、スケーリングアクションをトリガーします。

可用性とは、ポッドを迅速かつ中断することなくスケジュールできることを意味します。これには、新しく作成されたポッドをスケジュールする必要がある場合や、スケールダウンされたノードがスケジュールされた残りのポッドを終了する場合が含まれます。

コストは、イベントのスケールアウトとスケールインの背後にある決定によって決まります。既存のノードが十分に活用されていない場合、または受信ポッドに対して大きすぎる新しいノードが追加された場合、リソースは浪費されます。ユースケースによっては、積極的なスケールダウン決定により、ポッドの早期終了に関連するコストが発生する可能性があります。

Node Groups は、クラスター内のノードのグループの抽象的な Kubernetes 概念です。これは真の Kubernetes リソースではありませんが、Cluster Autoscaler、Cluster API、およびその他のコンポーネントに抽象化として存在します。ノードグループ内のノードは、ラベルやテイントなどのプロパティを共有しますが、複数のアベイラビリティーゾーンまたはインスタンスタイプで構成される場合があります。

EC2 Auto Scaling Groups は、EC2 でのノードグループの実装として使用できます。EC2 Auto Scaling グループは、Kubernetes クラスターを自動的に結合し、Kubernetes API の対応するノードリソースにラベルとテイントを適用するインスタンスを起動するように設定されています。

EC2 マネージド型ノードグループは、EC2 でのノードグループの別の実装です。EC2 Autoscaling Scaling Groups を手動で設定する複雑さを抽象化し、ノードバージョンのアップグレードや正常なノード終了などの追加の管理機能を提供します。

クラスターオートスケーラーの操作

Cluster Autoscaler は、通常、クラスター内のデプロイとしてインストールされます。リーダーの選択を使用して高可用性を確保しますが、作業は一度に 1 つのレプリカによって行われます。水平方向にスケーラブルではありません。基本的なセットアップの場合、デフォルトでは、提供されているインストール手順を使用してすぐに機能しますが、注意すべき点がいくつかあります。

以下を確認してください。

- Cluster Autoscaler のバージョンがクラスターのバージョンと一致します。クロスバージョン互換性はテストもサポートもされていません。
- このモードの使用を妨げる特定の高度なユースケースがない限り、自動検出は有効です。

IAM ロールへの最小特権アクセスを採用する

Auto Discovery を使用する場合は、現在のクラスターに限定されたアクション`autoscaling:SetDesiredCapacity`と Auto Scaling グループを制限し

てautoscaling:TerminateInstanceInAutoScalingGroup、最小特権アクセスを採用することを強くお勧めします。

これにより、--node-group-auto-discovery引数がタグ (など) を使用してクラスターのノードグループにスコープダウンされていない場合でも、1 つのクラスターで実行されているCluster Autoscaler が別のクラスターのノードグループを変更できなくなりますk8s.io/cluster-autoscaler/<cluster-name>。

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "autoscaling:SetDesiredCapacity",
                "autoscaling:TerminateInstanceInAutoScalingGroup"
            ],
            "Resource": "*",
            "Condition": {
                "StringEquals": {
                    "aws:ResourceTag/k8s.io/cluster-autoscaler/enabled": "true",
                    "aws:ResourceTag/k8s.io/cluster-autoscaler/my-cluster": "owned"
                }
            }
        },
        {
            "Effect": "Allow",
            "Action": [
                "autoscaling:DescribeAutoScalingGroups",
                "autoscaling:DescribeAutoScalingInstances",
                "autoscaling:DescribeLaunchConfigurations",
                "autoscaling:DescribeScalingActivities",
                "autoscaling:DescribeTags",
                "ec2:DescribeImages",
                "ec2:DescribeInstanceTypes",
                "ec2:DescribeLaunchTemplateVersions",
                "ec2:GetInstanceTypesFromInstanceRequirements",
                "eks:DescribeNodegroup"
            ],
            "Resource": "*"
        }
    ]
```

```
}
```

ノードグループの設定

効果的な自動スケーリングは、クラスターの一連のノードグループを正しく設定することから始まります。適切なノードグループのセットを選択することは、ワークロード全体の可用性を最大化し、コストを削減するために重要です。AWS は EC2 Auto Scaling Groups を使用してノードグループを実装します。これは、多数のユースケースに柔軟に対応します。ただし、Cluster Autoscaler はノードグループについていくつかの仮定を行います。EC2 Auto Scaling グループ設定をこれらの前提と一致させることで、望ましくない動作を最小限に抑えることができます。

以下を確認してください。

- ノードグループ内の各ノードには、ラベル、テイント、リソースなど、同じスケジューリングプロパティがあります。
  - MixedInstancePolicies の場合、インスタンスタイプは CPU、メモリ、GPU で同じ形状である必要があります
  - ポリシーで指定された最初のインスタンスタイプは、スケジューリングをシミュレートするために使用されます。
  - ポリシーにより多くのリソースを持つ追加のインスタンスタイプがある場合、スケールアウト後にリソースが浪費される可能性があります。
  - ポリシーに追加のインスタンスタイプがあり、リソースが少ない場合、ポッドはインスタンスでスケジュールに失敗する可能性があります。
- ノード数が多いノードグループは、ノード数が少ない多くのノードグループよりも優先されます。これはスケーラビリティに最も大きな影響を与えます。
- 可能な限り、両方のシステムがサポートを提供する場合は EC2 機能 (リージョン、MixedInstancePolicy など) を優先します。

> **Note**
>
> EKS Managed Node Groups を使用することをお勧めします。マネージド型ノードグループには、EC2 Auto Scaling グループの自動検出や正常なノード終了などの Cluster Autoscaler の機能など、強力な管理機能があります。

パフォーマンスとスケーラビリティの最適化

Auto Scaling アルゴリズムのランタイムの複雑さを理解することで、1,000 を超えるノードを持つ大規模なクラスターでスムーズに動作し続けるように Cluster Autoscaler を調整することができます。

Cluster Autoscaler のスケーラビリティを調整するための主なつまみは、プロセスに提供されるリソース、アルゴリズムのスキャン間隔、クラスター内のノードグループの数です。このアルゴリズムの真のランタイムの複雑さには、プラグインの複雑さやポッドの数のスケジューリングなど、他の要因があります。これらはクラスターのワークロードにとって自然であり、簡単に調整できないため、設定不可能なパラメータと見なされます。

Cluster Autoscaler は、ポッド、ノード、ノードグループなど、クラスターの状態全体をメモリにロードします。スキャン間隔ごとに、アルゴリズムはスケジュール不可能なポッドを識別し、各ノードグループのスケジューリングをシミュレートします。これらの要素の調整にはさまざまなトレードオフがあるため、ユースケースに合わせて慎重に検討する必要があります。

Cluster Autoscaler の垂直的自動スケーリング

Cluster Autoscaler を大規模なクラスタにスケーリングする最も簡単な方法は、デプロイのリソースに対するリクエストを増やすことです。大規模なクラスターでは、メモリと CPU の両方を増やす必要があります。ただし、この数値はクラスターのサイズによって大きく異なります。自動スケーリングアルゴリズムは、すべてのポッドとノードをメモリに保存するため、場合によっては 1 ギガバイトを超えるメモリフットプリントが発生する可能性があります。リソースの増加は通常、手動で行われます。定常的なリソース調整作業が運用上の負担となる場合には、Addon Resizer または Vertical Pod Autoscaler の使用を検討してください。

ノードグループの数を減らす

ノードグループの数を最小限に抑えることは、Cluster Autoscaler が大規模なクラスターで引き続き優れたパフォーマンスを発揮できるようにするための 1 つの方法です。これは、チームごとまたはアプリケーションごとにノードグループを構築する一部の組織では難しい場合があります。これは Kubernetes API で完全にサポートされていますが、スケーラビリティに影響する Cluster Autoscaler アンチパターンと見なされます。複数のノードグループ (スポットや GPUs など) を使用する理由は多数ありますが、多くの場合、少数のグループを使用しながら同じ効果を実現する代替設計があります。

以下を確認してください。

- ポッドの分離は、ノードグループではなく名前空間を使用して行われます。
  - この処理は、低信頼のマルチテナントクラスターでは不可能な場合があります。

  - Pod ResourceRequests と ResourceLimits は、リソースの競合を避けるために適切に設定されています。
  - インスタンスタイプが大きいほど、ビンパッキングがより最適化され、システムポッドのオーバーヘッドが削減されます。
- NodeTaints または NodeSelectors は、ルールではなく例外としてポッドをスケジュールするために使用されます。
- リージョンリソースは、複数のアベイラビリティーゾーンを持つ単一の EC2 Auto Scaling グループとして定義されます。

スキャン間隔の短縮

スキャン間隔を短くすると (10 秒など) 、ポッドがスケジュール不能になったときに Cluster Autoscaler ができるだけ早く応答します。ただし、各スキャンでは、Kubernetes API および EC2 Auto Scaling グループまたは EKS Managed Node Group API への多くの APIs コールが発生します。これらの API コールは、Kubernetes コントロールプレーンのレート制限やサービス利用不能につながる可能性があります。

デフォルトのスキャン間隔は 10 秒ですが、AWS では、ノードの起動に新しいインスタンスの起動にかなり時間がかかります。つまり、全体的なスケールアップ時間を大幅に増やすことなく、スキャン間隔を増やすことが可能だということです。たとえば、ノードの起動に 2 分かかる場合、間隔を 1 分に変更すると、API コールが 6 倍減少し、スケールアップが 38% 遅くなります。

ノードグループ間のシャーディング

Cluster Autoscaler は、特定のノードグループのセットで動作するように設定できます。この機能を使用すると、Cluster Autoscaler の複数のインスタンスをデプロイできます。それぞれが異なるノードグループのセットで動作するように設定されています。この戦略により、任意の数のノードグループを使用し、スケーラビリティのためにコストを取引できます。パフォーマンスを向上させるための最後の手段としてのみ、これを使用することをお勧めします。

Cluster Autoscaler はもともとこの設定用に設計されていないため、いくつかの副作用があります。シャードは通信しないため、複数のオートスケーラーがスケジュール不可能なポッドをスケジュールしようとする可能性があります。これにより、複数のノードグループから不要なスケールアウトが発生する可能性があります。これらの追加のノードは、 の後にスケールバックされますscale-down-delay。

```
metadata:
```

```
  name: cluster-autoscaler
  namespace: cluster-autoscaler-1

...

--nodes=1:10:k8s-worker-asg-1
--nodes=1:10:k8s-worker-asg-2

---

metadata:
  name: cluster-autoscaler
  namespace: cluster-autoscaler-2

...

--nodes=1:10:k8s-worker-asg-3
--nodes=1:10:k8s-worker-asg-4
```

以下を確認してください。

- 各シャードは、EC2 Auto Scaling グループの一意のセットを指すように設定されています。
- リーダー選挙の競合を避けるため、各シャードは個別の名前空間にデプロイされます。

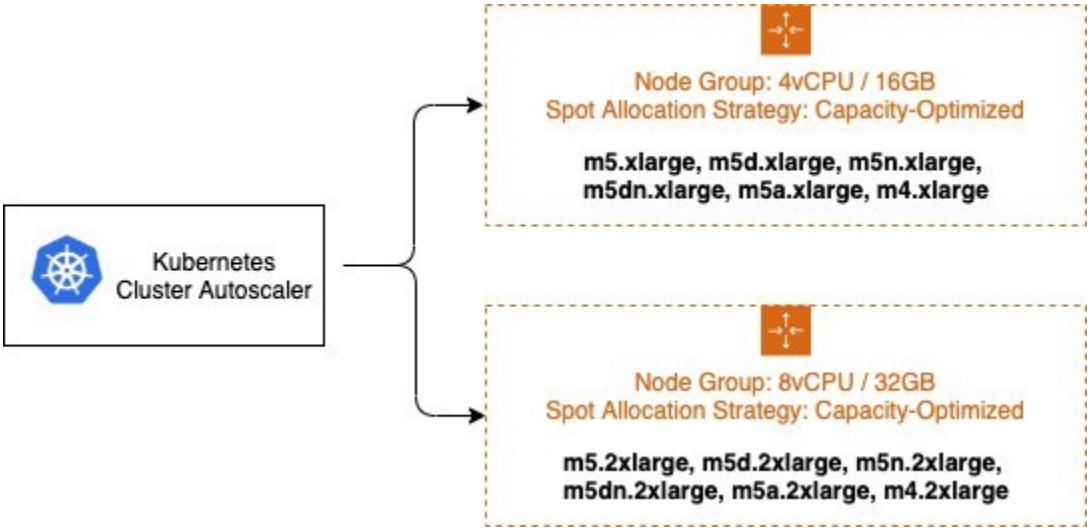
コストと可用性の最適化

スポットインスタンス

ノードグループでスポットインスタンスを使用し、オンデマンド料金から最大 90% を節約できます。EC2 で容量を戻す必要がある場合は、いつでもスポットインスタンスを中断できます。キャパシティ不足エラーは、EC2 Auto Scaling グループが使用可能なキャパシティ不足のためにスケールアップできない場合に発生します。多くのインスタンスファミリーを選択して多様性を最大化すると、多くのスポットキャパシティープールを活用して希望するスケールを達成できる可能性が高くなり、クラスターの可用性に対するスポットインスタンスの中断の影響が軽減されます。スポットインスタンスにより多種類のインスタンスポリシーを使用することは、ノードグループの数を増やすことなく多様性を高める優れた方法です。保証されたリソースが必要な場合は、スポットインスタンスの代わりにオンデマンドインスタンスを使用してください。

混合インスタンスポリシーを設定するときは、すべてのインスタンスタイプに同様のリソース容量があることが重要です。オートスケーラーのスケジューリングシミュレーター

は、MixedInstancePolicy の最初の InstanceType InstanceType を使用します。後続のインスタンスタイプが大きい場合、スケールアップ後にリソースが浪費される可能性があります。後続がより小さい場合には、容量不足のために、ポッドは新しいインスタンスでのスケジュールに失敗することがあります。例えば、M4, M5, M5a、M5n インスタンスはすべて同じ量の CPU とメモリを持ち、MixedInstancePolicy の候補として最適です。EC2 インスタンスセレクタツールは、同様のインスタンスタイプを識別するのに役立ちます。



オンデマンドキャパシティとスポットキャパシティを別々の EC2 Auto Scaling グループに分離することをお勧めします。これは、スケジューリングプロパティが根本的に異なるため、基本容量戦略を使用するよりも優先されます。スポットインスタンスはいつでも中断されるため (EC2 が容量を戻す必要がある場合) 、ユーザーは多くの場合、プリエンプション可能なノードをテイントし、プリエンプション動作に明示的なポッドトレレーションを必要とします。これらのテイントはノードのスケジューリングプロパティが異なるため、複数の EC2 Auto Scaling グループに分割する必要があります。

Cluster Autoscaler には、スケーリングするノードグループを選択するためのさまざまな戦略を提供するエクスパンダーの概念があります。戦略`--expander=least-waste`は優れた汎用デフォルトであり、スポットインスタンスの分散に複数のノードグループを使用する場合 (上の図を参照) 、スケーリングアクティビティの後に最もよく利用されるグループをスケーリングすることで、ノードグループをさらにコスト最適化できます。

ノードグループ/ASG の優先順位付け

Priority エクスパンダーを使用して優先度ベースの自動スケーリングを設定することもできます。は、クラスターがノードグループ/ASG に優先順位を付けること`--expander=priority`を可能にし、何らかの理由でスケールできない場合は、優先順位リストで次のノードグループを選択します。これは、GPU がワークロードに最適なパフォーマンスを提供するため、P3 インスタンスタイプを使

用する場合などに便利ですが、2 番目のオプションとして P2 インスタンスタイプを使用することもできます。

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: cluster-autoscaler-priority-expander
  namespace: kube-system
data:
  priorities: |-
    10:
      - .*p2-node-group.*
    50:
      - .*p3-node-group.*
```

Cluster Autoscaler は、p3-node-group という名前に一致する EC2 Auto Scaling グループをスケールアップしようとします。 このオペレーションが 内で成功しない場合`--max-node-provision-time`、名前 p2-node-group に一致する EC2 Auto Scaling グループのスケーリングを試みます。 この値のデフォルトは 15 分で、これを短くすることで、応答性の高いノードグループを選択することもできます。ただし、値を小さくしすぎると、不要なスケールアウトを発生させる可能性があります。

オーバープロビジョニング

Cluster Autoscaler は、必要性がある場合にのみノードがクラスタに追加され、使用されなくなった際には削除されるようにすることで、コストを最小限に抑えます。この構成では、スケジューリングされる前の多くのポッドが、ノードのスケールアップを待たせざるを得ないため、デプロイのレイテンシーに大きな影響を与えます。ノードが使用可能になるまでに数分かかることがあり、ポッドのスケジューリングのレイテンシーが桁違いに増加する可能性があります。

この問題は、オーバープロビジョニングにより、スケジューリングのレイテンシーをコストとトレードオフすることで緩和できます。オーバープロビジョニングは、クラスター内のスペースを占める負の優先度の一時的なポッドを使用して実装されます。新しく作成されたポッドがスケジュール不可能で優先度が高い場合、一時ポッドはスペースを確保するために優先されます。その後、一時ポッドはスケジュール不可能になり、Cluster Autoscaler が新しいオーバープロビジョニングされたノードをスケールアウトするようにトリガーされます。

オーバープロビジョニングには、他にあまり明確でない利点があります。過剰プロビジョニングを行わないと、使用率の高いクラスターの副作用の 1 つは、ポッドが Pod または Node Affinity の`preferredDuringSchedulingIgnoredDuringExecution`ルールを使用して最適なスケ

ジューリング決定を下さないことです。これの一般的なユースケースは、AntiAffinity を使用して可用性の高いアプリケーションのポッドをアベイラビリティーゾーン間で分離することです。オーバープロビジョニングは、正しいゾーンのノードが使用可能になる可能性を大幅に増やす可能性があります。

過剰にプロビジョニングされた容量の量は、組織にとって慎重なビジネス上の決定です。その中核となるのは、パフォーマンスとコストのトレードオフです。この決定を行う 1 つの方法は、平均スケールアップ頻度を決定し、それを新しいノードのスケールアップにかかる時間で割ることです。例えば、平均して 30 秒ごとに新しいノードが必要で、EC2 が新しいノードをプロビジョニングするのに 30 秒かかる場合、オーバープロビジョニングの 1 つのノードで常に追加のノードが利用可能になり、1 つの EC2 インスタンスのコストでスケジューリングのレイテンシーを 30 秒短縮できます。ゾーンスケジューリングの決定を改善するには、EC2 Auto Scaling グループのアベイラビリティーゾーンの数と等しい数のノードをオーバープロビジョニングして、スケジューラが受信ポッドに最適なゾーンを選択できるようにします。

スケールダウンエビクションの防止

一部のワークロードでは、削除にコストがかかります。ビッグデータ分析、機械学習タスク、テストランナーは最終的に完了しますが、中断した場合は再起動する必要があります。Cluster Autoscaler は、scale-down-utilization-thresholdの下にあるノードをスケールダウンしようとします。これにより、ノード上の残りのポッドが中断されます。これは、削除にコストがかかるポッドが、Cluster Autoscaler によって認識されるラベルによって保護されるようにすることで防止できます。

以下を確認してください。

- ポッドの削除にコストがかかると、 注釈が付けられます。 `cluster-autoscaler.kubernetes.io/safe-to-evict=false`

高度なユースケース

EBS ボリューム

データベースや分散キャッシュなどのステートフルアプリケーションを構築するには、永続的ストレージが不可欠です。EBS ボリュームは、Kubernetes でこのユースケースを有効にしますが、特定のゾーンに限定されます。これらのアプリケーションは、AZ ごとに個別の EBS ボリュームを使用して複数の AZs にまたがってシャーディングする場合に可用性を高めることができます。その後、Cluster Autoscaler は EC2 Autoscaling グループのスケーリングのバランスをとることができます。

以下を確認してください。

- `balance-similar-node-groups=true` を設定することで、ノードグループのバランシングが有効化されている。
- ノードグループは、異なるアベイラビリティーゾーンと EBS ボリュームを除いて、同じ設定で設定されます。

共同スケジューリング

機械学習の分散型トレーニングジョブでは、共通のゾーンを使用するノード構成によりレイテンシーを最小化できるので、大きなメリットが得られます。これらのワークロードは、複数のポッドを特定のゾーンにデプロイします。これは、 を使用して、すべての共同スケジュールされたポッドまたは Node Affinity に Pod Affinity を設定することで実現できますtopologyKey: failure-domain.beta.kubernetes.io/zone。その後、Cluster Autoscaler は需要に合わせて特定のゾーンをスケールアウトします。複数の EC2 Auto Scaling グループをアベイラビリティーゾーンごとに 1 つずつ割り当てて、共同スケジュールされたワークロード全体のフェイルオーバーを有効にできます。

以下を確認してください。

- `balance-similar-node-groups=false` を設定することで、ノードグループのバランシングが有効化されている。
- Node Affinity や Pod Preemption は、クラスターにリージョンノードグループとゾーンノードグループの両方が含まれている場合に使用されます。
  - Node Affinity を使用して、ゾーンノードグループを回避するためにリージョンポッドを強制または奨励します。その逆も同様です。
  - ゾーンポッドがリージョン別ノードグループにスケジュールされている場合、リージョン別ポッドの容量が不均衡になります。
  - ゾーンワークロードが中断と再配置を許容できる場合は、リージョン別にスケーリングされたポッドが競合の少ないゾーンでプリエンプションと再スケジュールを強制できるように Pod Preemption を設定します。

アクセラレータ

一部のクラスターでは、GPU などの特殊なハードウェアアクセラレーターを利用しています。スケールアウトが行われた場合、アクセラレーターデバイスのプラグインがリソースをクラスターに

アドバタイズするのに、数分かかることがあります。Cluster Autoscaler は、このノードにアクセラレーターがあることをシミュレートしましたが、アクセラレーターの準備が整い、ノードの使用可能なリソースが更新されるまで、保留中のポッドをノードでスケジュールすることはできません。これは、不要なスケールアウトが繰り返されることにつがなります。

さらに、アクセラレーターがあり、CPU またはメモリ使用率が高いノードは、アクセラレーターが未使用であってもスケールダウンの対象として考慮されません。この動作は、アクセラレーターの相対コストのため、高価になる可能性があります。代わりに、Cluster Autoscaler は特別なルールを適用して、占有されていないアクセラレーターがある場合にスケールダウンするノードを考慮できます。

このような場合の正しい動作を確保するために、アクセラレーターノードで kubelet を設定して、クラスターに参加する前にノードにラベルを付けることができます。Cluster Autoscaler は、このラベルセレクタを使用してアクセラレーターに最適化された動作をトリガーします。

以下を確認してください。

- GPU ノードの Kubelet は で設定されます。 `--node-labels k8s.amazonaws.com/accelerator=$ACCELERATOR_TYPE`
- Accelerator を使用するノードは、上記の同じスケジューリングプロパティルールに従います。

0 からのスケーリング

Cluster Autoscaler は、ゼロとの間でノードグループをスケーリングできるため、大幅なコスト削減につながります。LaunchConfiguration または LaunchTemplate で指定された InstanceType を調べることで、Auto Scaling グループの CPU、メモリ、GPU リソースを検出します。一部のポッドには、LaunchConfiguration から検出できない `PrivateIPv4Address` `WindowsENI`や特定の NodeSelectors や Taints などの追加のリソースが必要です。Cluster Autoscaler はEC2 Auto Scaling グループのタグから検出することで、これらの要因を考慮できます。例えば、次のようになります。

```
Key: k8s.io/cluster-autoscaler/node-template/resources/$RESOURCE_NAME
Value: 5
Key: k8s.io/cluster-autoscaler/node-template/label/$LABEL_KEY
Value: $LABEL_VALUE
Key: k8s.io/cluster-autoscaler/node-template/taint/$TAINT_KEY
Value: NoSchedule
```

> **Note**
>
> ゼロにスケーリングすると、容量は EC2 に返され、将来使用できなくなる可能性があることに注意してください。

追加のパラメータ

Cluster Autoscaler には、動作とパフォーマンスを調整するために使用できる多くの設定オプションが用意されています。パラメータの完全なリストは、GitHub で入手できます。

パラメータ	説明	デフォルト
スキャン間隔	スケールアップまたはスケールダウンのためにクラスターが再評価される頻度	10 秒
max-empty-bulk-delete	同時に削除できる空のノードの最大数。	10
scale-down-delay-after-add	スケールアップ後のスケールダウン評価の再開時間	10 分
scale-down-delay-after-delete	評価をスケールダウンするノードの削除が再開されてから、デフォルトではスキャン間隔になります。	スキャン間隔
scale-down-delay-after-failure	スケールダウン失敗後にスケールダウン評価が再開される期間	3 分
scale-down-unneeded-time	スケールダウンの対象となるまでにノードが必要でない期間	10 分

パラメータ	説明	デフォルト
scale-down-unready-time	スケールダウンの対象となるまでに、準備されていないノードが不要になる期間	20 分
scale-down-utilization-threshold	リクエストされたリソースの合計を容量で割ったノード使用率。これを下回ると、ノードのスケールダウンを検討できます。	0.5
scale-down-non-empty-candidates-count	ドレインによるスケールダウンの候補として 1 回の反復で考慮される空でないノードの最大数。値が低いほど、CA の応答性は向上しますが、スケールダウンレイテンシーが遅くなる可能性があります。値を大きくすると、大きなクラスター (数百のノード) の CA パフォーマンスに影響する可能性があります。このヒューリスティックをオフにするには、非正の値に設定します。CA は考慮するノードの数を制限しません。	30

パラメータ	説明	デフォルト
scale-down-candidates-pool-ratio	以前のイテレーションの一部の候補が有効でなくなった場合に、スケールダウンの空の候補ではない追加のノードの比率。値が低いほど、CA の応答性は向上しますが、スケールダウンレイテンシーが遅くなる可能性があります。値を大きくすると、大きなクラスター (数百のノード) の CA パフォーマンスに影響する可能性があります。このヒューリスティックをオフにするには 1.0 に設定します。CA はすべてのノードを追加候補として受け取ります。	0.1
scale-down-candidates-pool-min-count	以前のイテレーションの一部の候補が有効でなくなった場合に、スケールダウンの空の候補ではない追加のノードの最小数。追加の候補のプールサイズを計算する場合 `max(#nodes * scale-down-candidates-pool-ratio, scale-down-candidates-pool-min-count)`	50

その他のリソース

このページには、Cluster Autoscaler のプレゼンテーションとデモのリストが含まれています。ここでプレゼンテーションまたはデモを追加する場合は、プルリクエストを送信してください。

プレゼンテーション/デモ	プレゼンター
Kubernetes での Auto Scaling とコスト最適化: 0～100	Guy Templeton、Skyscanner & Jiaxin Shan、Amazon
SIG-Autoscaling の詳細	Maciek Pytel & Marcin Wielgus

リファレンス

- https://github.com/kubernetes/autoscaler/blob/master/cluster-autoscaler/FAQ.md
- https://github.com/kubernetes/autoscaler/blob/master/cluster-autoscaler/cloudprovider/aws/README.md
- https://github.com/aws/amazon-ec2-instance-selector
- https://github.com/aws/aws-node-termination-handler

信頼性のベストプラクティス

このセクションでは、EKS で実行されているワークロードを回復力と高可用性にするためのガイダンスを提供します。

このガイドの使い方

このガイドは、EKS で可用性と耐障害性に優れたサービスを開発して運用したい開発者やアーキテクトを対象としています。このガイドは、使いやすいようにさまざまなトピック領域にまとめられています。各トピックは簡単な概要から始まり、EKS クラスターの信頼性に関する推奨事項とベストプラクティスのリストが続きます。

序章

EKS の信頼性のベストプラクティスは、以下のトピックでグループ化されています。

- アプリケーション
- コントロールプレーン
- データプレーン

システムの信頼性を高めるものは何ですか? 一定期間の環境の変化にもかかわらず、システムが一貫して機能し、需要を満たすことができる場合は、信頼性と呼ぶことができます。これを実現するには、システムは障害を検出し、自動的に修復し、需要に基づいてスケーリングできる必要があります。

お客様は、ミッションクリティカルなアプリケーションとサービスを確実に運用するための基盤として Kubernetes を使用できます。ただし、コンテナベースのアプリケーション設計原則を組み込むことに加えて、ワークロードを確実に実行するには、信頼性の高いインフラストラクチャも必要です。Kubernetes では、インフラストラクチャはコントロールプレーンとデータプレーンで構成されます。

EKS は、可用性と耐障害性に優れた本番稼働グレードの Kubernetes コントロールプレーンを提供します。

EKS では、AWS が Kubernetes コントロールプレーンの信頼性を担当します。EKS は、AWS リージョンの 3 つのアベイラビリティーゾーンで Kubernetes コントロールプレーンを実行しま

す。Kubernetes API サーバーと etcd クラスターの可用性とスケーラビリティを自動的に管理します。

データプレーンの信頼性に対する責任は、ユーザー、顧客、AWS の間で共有されます。EKS には、Kubernetes データプレーンをデプロイするための 4 つのワーカーノードオプションが用意されています。

最も管理されたオプションである EKS Auto Mode は、データプレーンのプロビジョニング、スケーリング、更新を処理し、マネージド型のコンピューティング、ネットワーキング、ストレージ機能を提供します。Auto Mode AMIs は頻繁にリリースされ、クラスターは CVE 修正とセキュリティパッチをデプロイするために最新の AMI に自動的に更新されます。自動モード NodePools で中断コントロールを設定することで、これが発生するタイミングを制御できます。

Fargate は、ノードごとに 1 つの Pod を実行することで、データプレーンのプロビジョニングとスケーリングを処理します。3 番目のオプションであるマネージドノードグループは、データプレーンのプロビジョニングと更新を処理します。最後に、セルフマネージド型ノードはデータプレーンの最小マネージド型オプションです。使用する AWS マネージドデータプレーンが多いほど、責任は軽減されます。

マネージド型ノードグループは、EC2 ノードのプロビジョニングとライフサイクル管理を自動化します。EKS API (EKS コンソール、AWS API、AWS CLI、CloudFormation、Terraform、または を使用`eksctl`) を使用して、マネージドノードを作成、スケーリング、アップグレードできます。マネージドノードは、アカウントで EKS 最適化 Amazon Linux 2 EC2 インスタンスを実行し、SSH アクセスを有効にすることでカスタムソフトウェアパッケージをインストールできます。マネージドノードをプロビジョニングすると、複数のアベイラビリティーゾーンにまたがることができる EKS マネージド Auto Scaling グループの一部として実行されます。これは、マネージドノードの作成時に指定したサブネットを介して制御します。また、EKS はマネージドノードに自動的にタグを付け、Cluster Autoscaler で使用できるようにします。

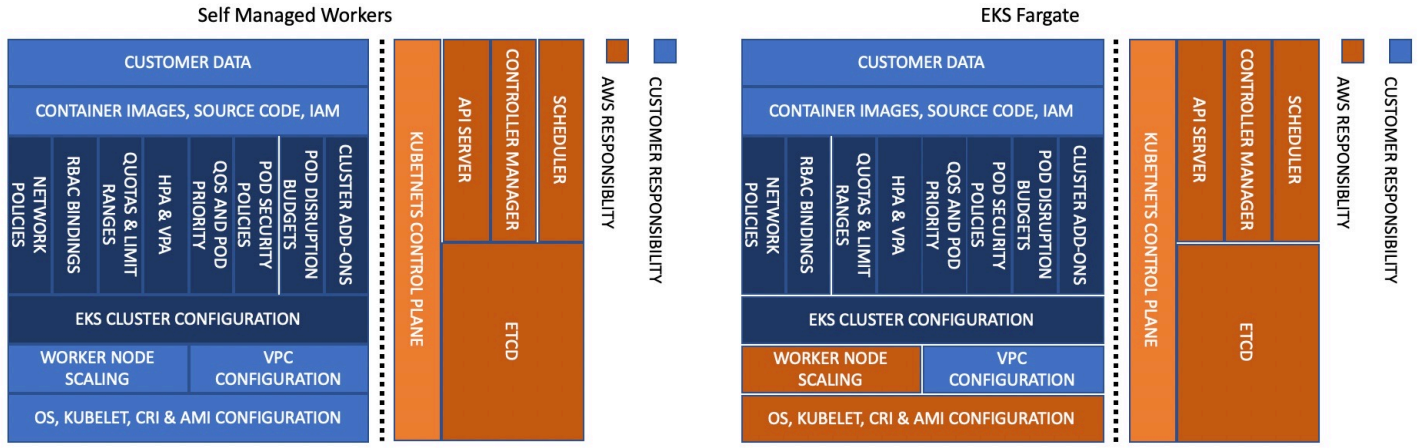
Amazon EKS は、マネージド型ノードグループで CVE およびセキュリティパッチの責任共有モデルに従います。マネージドノードは Amazon EKS 最適化 AMIs を実行するため、Amazon EKS はバグ修正時にこれらの AMIs のパッチ適用されたバージョンを構築する責任があります。ただし、これらのパッチが適用された AMI バージョンのマネージド型ノードグループへのデプロイはユーザーが担当します。

EKS はノードの更新も管理しますが、更新プロセスを開始する必要があります。マネージドノードを更新するプロセスは、EKS ドキュメントで説明されています。

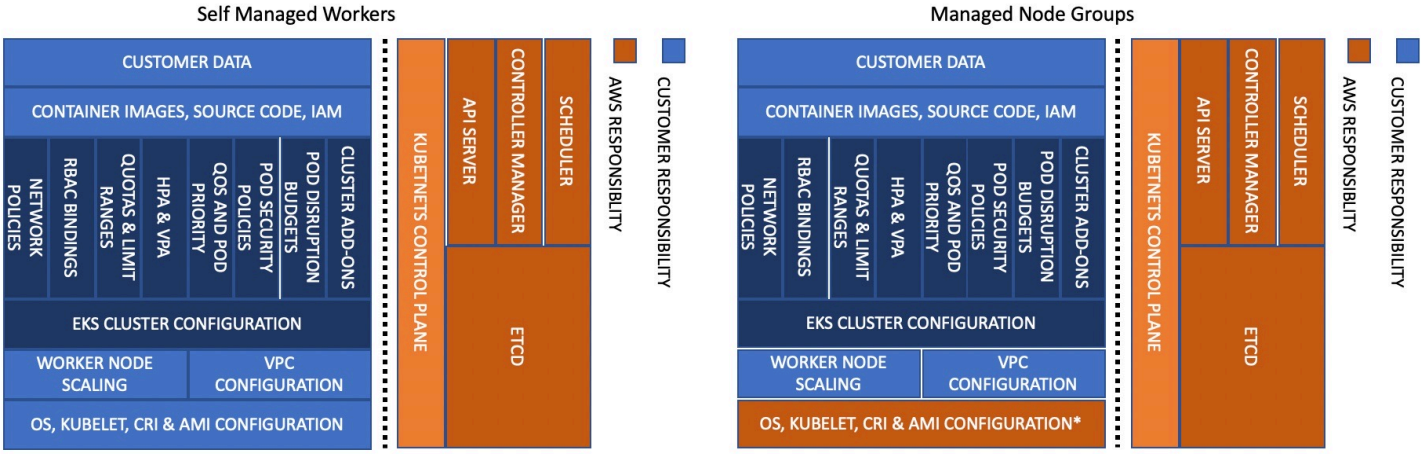
セルフマネージド型ノードを実行する場合は、Amazon EKS 最適化 Linux AMI を使用してワーカーノードを作成できます。AMI とノードのパッチ適用とアップグレードはお客様の責任となり

ます。、eksctlCloudFormation、または infrastructure をコードツールとして使用してセルフマネージドノードをプロビジョニングするのがベストプラクティスです。これにより、セルフマネージドノードのアップグレードが容易になります。ワーカーノードを更新するときは、新しいノードへの移行を検討してください。移行プロセスによって古いノードグループが としてテイントNoScheduleされ、新しいスタックが既存のポッドワークロードを受け入れる準備ができたらノードがドレインされるためです。ただし、セルフマネージドノードのインプレースアップグレードを実行することもできます。

責任共有モデル - Fargate



責任共有モデル - MNG



このガイドには、EKS データプレーン、Kubernetes コアコンポーネント、アプリケーションの信頼性を向上させるために使用できる推奨事項のセットが含まれています。

フィードバック

このガイドは、より広範な EKS/Kubernetes コミュニティから直接フィードバックや提案を収集するために GitHub でリリースされています。ガイドに含める必要があると思われるベストプラクティスがある場合は、GitHub リポジトリに問題をファイルするか、PR を送信してください。サービスに新機能が追加されたとき、または新しいベストプラクティスが進化したときに、ガイドを定期的に更新する予定です。

高可用性アプリケーションの実行

お客様は、変更を行う場合や特にトラフィックの急増時など、アプリケーションを常に利用できることを期待しています。スケーラブルで回復力のあるアーキテクチャにより、アプリケーションやサービスを中断することなく実行し、ユーザーを満足させます。スケーラブルなインフラストラクチャは、ビジネスのニーズに基づいて増減します。単一障害点を排除することは、アプリケーションの可用性を向上させ、回復力を高めるための重要なステップです。

Kubernetes を使用すると、アプリケーションを運用し、可用性と回復力に優れた方法で実行できます。宣言型管理により、アプリケーションを設定すると、Kubernetes は引き続き現在の状態を目的の状態と照合しようとします。

推奨事項

Pod Disruption Budgets を設定する

Pod Disruption Budgets は、アプリケーションが同時に発生する中断の量を制限するために使用されます。ワークロードの一部を常に利用できるようにすることが重要な場合は、ワークロード用に設定する必要があります。EKS Auto Mode、Karpenter、Cluster Autoscaler は、スケールダウン時に設定された Pod Disruption Budgets を認識し、準拠します。EKS Auto Mode、Karpenter、マネージド型ノードグループは、ノードの更新時に Pod Disruption Budgets にも準拠します。

シングルトンポッドの実行を避ける

アプリケーション全体が 1 つの Pod で実行されている場合、その Pod が終了するとアプリケーションは使用できなくなります。個々のポッドを使用してアプリケーションをデプロイする代わりに、デプロイを作成します。デプロイによって作成されたポッドが失敗または終了した場合、デプロイコントローラーは新しいポッドを起動して、指定された数のレプリカポッドが常に実行されていることを確認します。

複数のレプリカを実行する

デプロイを使用してアプリケーションの複数のレプリカポッドを実行すると、可用性の高い方法で実行できます。1 つのレプリカが失敗しても、残りのレプリカは機能します。ただし、Kubernetes が損失を補うために別の Pod を作成するまで、容量が減ります。さらに、Horizontal Pod Autoscaler を使用して、ワークロードの需要に基づいてレプリカを自動的にスケーリングできます。

ノード間でレプリカをスケジュールする

複数のレプリカを実行することは、すべてのレプリカが同じノードで実行されていて、ノードが使用できなくなった場合にはあまり役に立ちません。ポッドアンチアフィニティまたはポッドトポロジスプレッド制約を使用して、デプロイのレプリカを複数のワーカーノードに分散することを検討してください。

複数の AZs で実行することで、一般的なアプリケーションの信頼性をさらに向上させることができます。

Pod アンチアフィニティルールの使用

以下のマニフェストは、ポッドを別々のノードと AZ に配置することを Kubernetes スケジューラに指示します。 AZs 個別のノードや AZ は必要ありません。そうすると、各 AZ でポッドが実行されると、Kubernetes はポッドをスケジュールできなくなります。アプリケーションに必要なレプリカが 3 つしかない場合は、 `requiredDuringSchedulingIgnoredDuringExecution`に を使用できます。Kubernetes `topologyKey: topology.kubernetes.io/zone`スケジューラは、同じ AZ で 2 つのポッドをスケジュールしません。

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: spread-host-az
  labels:
    app: web-server
spec:
  replicas: 4
  selector:
    matchLabels:
      app: web-server
  template:
    metadata:
      labels:
        app: web-server
    spec:
```

```
      affinity:
        podAntiAffinity:
          preferredDuringSchedulingIgnoredDuringExecution:
          - podAffinityTerm:
              labelSelector:
                matchExpressions:
                - key: app
                  operator: In
                  values:
                  - web-server
              topologyKey: topology.kubernetes.io/zone
            weight: 100
          - podAffinityTerm:
              labelSelector:
                matchExpressions:
                - key: app
                  operator: In
                  values:
                  - web-server
              topologyKey: kubernetes.io/hostname
            weight: 99
      containers:
      - name: web-app
        image: nginx:1.16-alpine
```

ポッドトポロジの分散制約の使用

ポッドのアンチアフィニティルールと同様に、ポッドトポロジの分散制約により、ホストや AZs などのさまざまな障害 (またはトポロジ) ドメインでアプリケーションを使用できるようになります。このアプローチは、異なるトポロジドメインのそれぞれに複数のレプリカを配置することで、耐障害性と可用性を確保しようとする場合に非常に効果的です。一方、ポッドのアンチアフィニティルールは、互いにアンチアフィニティを持つポッドが反発効果を持つため、トポロジドメインに 1 つのレプリカがある場合に簡単に結果を生成できます。このような場合、専用ノード上の 1 つのレプリカは耐障害性には適していません。また、リソースの適切な使用にも適していません。トポロジの分散制約を使用すると、スケジューラがトポロジドメイン全体に適用しようとするスプレッドまたはディストリビューションをより細かく制御できます。このアプローチで使用する重要なプロパティをいくつか紹介します。

1. maxSkew は、トポロジドメイン間でモノが不均等になる最大ポイントを制御または決定するために使用します。たとえば、アプリケーションに 10 個のレプリカがあり、3 つの AZs にデプロイされている場合、均等に分散することはできませんが、ディストリビューションの不均等さに影

響を与えることができます。この場合、 は 1～10 のいずれかmaxSkewになります。値 1 は、 のようなスプレッド4,3,3、3,4,3または 3 つの AZs 3,3,4にまたがるスプレッドになる可能性があることを意味します。対照的に、値が 10 の場合、 のようなスプレッド10,0,0、0,10,0または 3 つの AZs 0,0,10にまたがるスプレッドになる可能性があります。

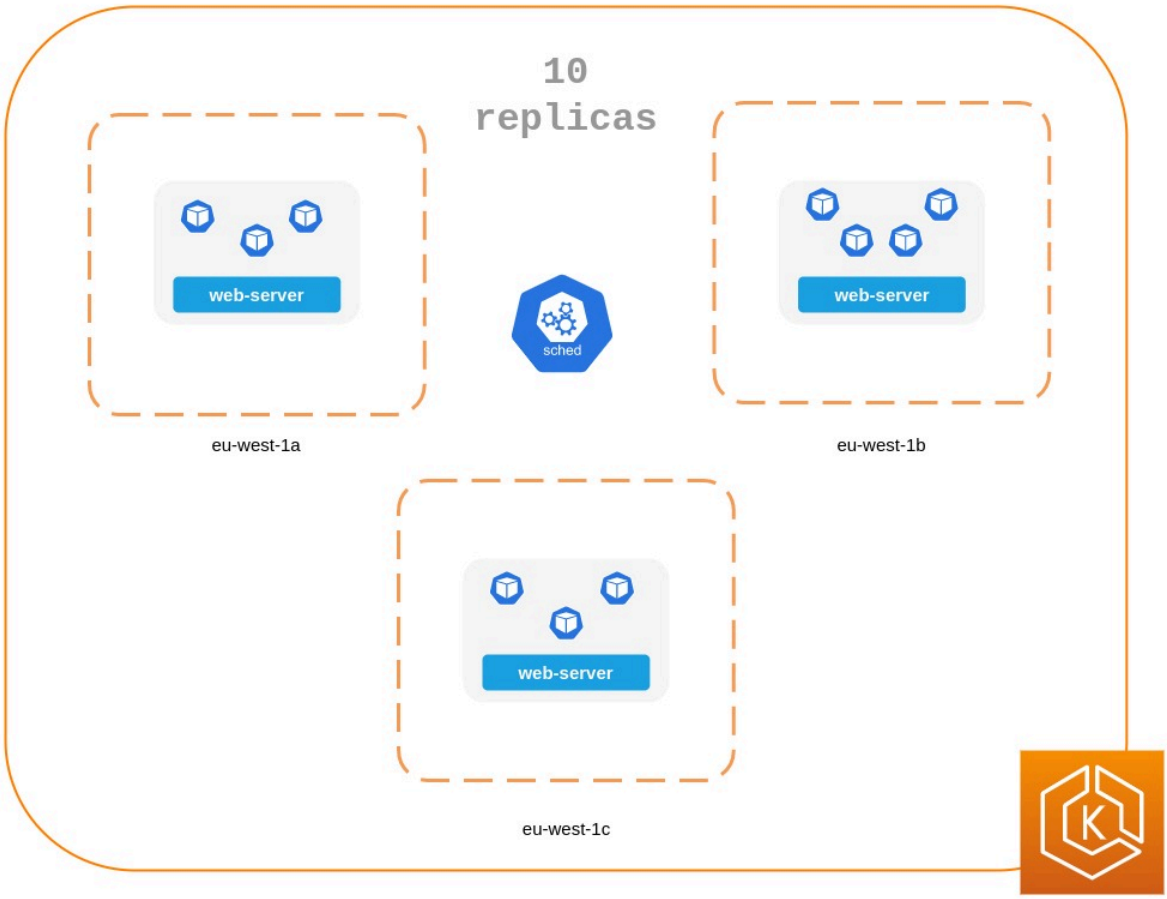
2. topologyKey はノードラベルの 1 つのキーであり、ポッドディストリビューションに使用するトポロジドメインのタイプを定義します。たとえば、ゾーンスプレッドには次のキーと値のペアがあります。

```
topologyKey: "topology.kubernetes.io/zone"
```

3. whenUnsatisfiable プロパティは、必要な制約が満たされない場合にスケジューラがどのように応答するかを決定するために使用されます。
4. labelSelector は、指定した制約に従ってポッドを配置する場所を決定するときにスケジューラがポッドを認識できるように、一致するポッドを見つけるために使用されます。

上記に加えて、Kubernetes ドキュメントでさらに読むことができる他のフィールドがあります。

ポッドトポロジーが 3 つの AZs




```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: spread-host-az
  labels:
    app: web-server
spec:
  replicas: 10
  selector:
    matchLabels:
      app: web-server
  template:
    metadata:
      labels:
        app: web-server
    spec:
      topologySpreadConstraints:
```

```
    - maxSkew: 1
      topologyKey: "topology.kubernetes.io/zone"
      whenUnsatisfiable: ScheduleAnyway
      labelSelector:
        matchLabels:
          app: express-test
  containers:
  - name: web-app
    image: nginx:1.16-alpine
```

Kubernetes メトリクスサーバーを実行する

Kubernetes メトリクスサーバーをインストールして、アプリケーションのスケーリングに役立てます。HPA や VPA などの Kubernetes Autoscaler アドオンは、アプリケーションのメトリクスを追跡してスケーリングする必要があります。metrics-server は、スケーリングの決定に使用できるリソースメトリクスを収集します。メトリクスは kubelets から収集され、Metrics API 形式で提供されます。

メトリクスサーバーはデータを保持せず、モニタリングソリューションではありません。その目的は、CPU とメモリの使用状況メトリクスを他のシステムに公開することです。アプリケーションの状態を経時的に追跡する場合は、Prometheus や Amazon CloudWatch などのモニタリングツールが必要です。

EKS ドキュメントに従って、EKS クラスターに metrics-server をインストールします。

水平ポッドオートスケーラー (HPA)

HPA は、需要に応じてアプリケーションを自動的にスケーリングし、ピークトラフィック中に顧客に影響を与えないようにできます。これは、リソースメトリクスを提供する APIsする Kubernetes のコントロールループとして実装されます。

HPA は、次の APIs からメトリクスを取得できます。1. Resource Metrics API `metrics.k8s.io` とも呼ばれる — ポッド 2 の CPU とメモリの使用量を提供します。 `custom.metrics.k8s.io`— Prometheus などの他のメトリクスコレクターからのメトリクスを提供します。これらのメトリクスは Kubernetes クラスターの内部にあります。3。 `external.metrics.k8s.io`— Kubernetes クラスターの外部にあるメトリクスを提供します (SQS キューの深さ、ELB レイテンシーなど) 。

これらの 3 つの APIs のいずれかを使用して、アプリケーションをスケールするための メトリクスを指定する必要があります。

カスタムメトリクスまたは外部メトリクスに基づくアプリケーションのスケーリング

カスタムメトリクスまたは外部メトリクスを使用して、CPU またはメモリ使用率以外のメトリクスでアプリケーションをスケーリングできます。カスタムメトリクス API サーバーは、HPA がアプリケーションの自動スケーリングに使用できる `custom-metrics.k8s.io` API を提供します。

Prometheus Adapter for Kubernetes Metrics APIs を使用して Prometheus からメトリクスを収集し、HPA で を使用できます。この場合、Prometheus アダプターは Prometheus メトリクスを Metrics API 形式で公開します。

Prometheus Adapter をデプロイしたら、kubectl を使用してカスタムメトリクスをクエリできます。`kubectl get –raw /apis/custom.metrics.k8s.io/v1beta1/`

外部メトリクスは、名前が示すように、Kubernetes クラスターの外部にあるメトリクスを使用してデプロイをスケールする機能を Horizontal Pod Autoscaler に提供します。たとえば、バッチ処理ワークロードでは、SQS キューで処理中のジョブの数に基づいてレプリカの数をスケールするのが一般的です。

Kubernetes ワークロードをオートスケーリングするには、KEDA (Kubernetes Event-driven Autoscaling) を使用できます。これは、多数のカスタムイベントに基づいてコンテナスケーリングを駆動できるオープンソースプロジェクトです。この AWS ブログでは、Amazon Managed Service for Prometheus for Kubernetes ワークロードの自動スケーリングを使用する方法について説明します。

垂直ポッドオートスケーラー (VPA)

VPA は Pod の CPU とメモリの予約を自動的に調整し、アプリケーションの「適切なサイズ」を支援します。リソース割り当てを増やすことで実行される垂直スケーリングが必要なアプリケーションの場合、VPA を使用して Pod レプリカを自動的にスケーリングしたり、スケーリングに関する推奨事項を提供したりできます。

VPA の現在の実装では Pod にインプレース調整が実行されないため、VPA がスケーリングする必要がある場合、アプリケーションは一時的に使用できなくなる可能性があります。代わりに、スケーリングする必要がある Pod を再作成します。

EKS ドキュメントには、VPA を設定するためのチュートリアルが含まれています。

Fairwinds Goldilocks プロジェクトは、CPU およびメモリのリクエストと制限に関する VPA レコメンデーションを視覚化するためのダッシュボードを提供します。VPA 更新モードでは、VPA レコメンデーションに基づいて Pod を自動スケーリングできます。

アプリケーションの更新

最新のアプリケーションには、高い安定性と可用性を備えた迅速なイノベーションが必要です。Kubernetes は、顧客を中断することなくアプリケーションを継続的に更新するためのツールを提供します。

可用性を犠牲にすることなく変更を迅速にデプロイできるようにするベストプラクティスをいくつか見てみましょう。

ロールバックを実行するメカニズムがある

元に戻すボタンを使用すると、災害を回避できます。本番環境クラスターを更新する前に、別の下位環境 (テスト環境または開発環境) でデプロイをテストするのがベストプラクティスです。CI/CD パイプラインを使用すると、デプロイの自動化とテストに役立ちます。継続的なデプロイパイプラインを使用すると、アップグレードに欠陥が発生した場合に、古いバージョンにすばやく戻すことができます。

デプロイを使用して、実行中のアプリケーションを更新できます。これは通常、コンテナイメージを更新することによって行われます。kubectl を使用して、次のようなデプロイを更新できます。

```
kubectl --record deployment.apps/nginx-deployment set image nginx-deployment
 nginx=nginx:1.16.1
```

--record 引数はデプロイの変更を記録し、ロールバックを実行する必要がある場合に役立ちます。 は、クラスター内のデプロイに記録された変更kubectl rollout history deploymentを表示します。を使用して変更をロールバックできますkubectl rollout undo deployment <DEPLOYMENT_NAME>。

デフォルトでは、ポッドの再作成を必要とするデプロイを更新すると、デプロイはローリング更新を実行します。つまり、Kubernetes はデプロイで実行中のポッドの一部のみを更新し、すべてのポッドを一度に更新するわけではありません。Kubernetes が RollingUpdateStrategyプロパティを使用してローリング更新を実行する方法を制御できます。

デプロイのローリング更新を実行するときは、 Max Unavailableプロパティを使用して、更新中に使用できないポッドの最大数を指定できます。Deployment の Max Surgeプロパティを使用すると、必要な数のポッドに対して作成できるポッドの最大数を設定できます。

ロールアウトが顧客max unavailableを混乱させないように調整することを検討してください。例えば、Kubernetes max unavailableはデフォルトで 25% を設定します。つまり、100 個のポッド

がある場合、ロールアウト中にアクティブに動作しているポッドは 75 個のみです。アプリケーションで最低 80 個の Pod が必要な場合は、このロールアウトが中断される可能性があります。代わりに、 を 20% `max unavailable`に設定して、ロールアウト全体で少なくとも 80 個の機能ポッドがあることを確認できます。

ブルー/グリーンデプロイを使用する

変更には本質的にリスクが伴いますが、元に戻すことができない変更は致命的になる可能性があります。ロールバックを通じて時間を効果的に戻すことができる手順を変更すると、機能強化と実験の安全性が向上します。ブルー/グリーンデプロイでは、問題が発生した場合に変更をすばやく取り消す方法が提供されます。このデプロイ戦略では、新しいバージョンの環境を作成します。この環境は、更新されるアプリケーションの最新バージョンと同じです。新しい環境がプロビジョニングされると、トラフィックは新しい環境にルーティングされます。新しいバージョンがエラーを発生させずに必要な結果を生成する場合、古い環境は終了します。それ以外の場合、トラフィックは古いバージョンに復元されます。

Kubernetes でブルー/グリーンデプロイを実行するには、既存のバージョンのデプロイと同じ新しいデプロイを作成します。新しいデプロイのポッドがエラーなしで実行されていることを確認したら、アプリケーションのポッドにトラフィックをルーティングするサービスの`selector`仕様を変更することで、新しいデプロイへのトラフィックの送信を開始できます。

Flux、Jenkins、" などの多くの継続的統合ツールを使用すると、ブルー/グリーンデプロイを自動化できます。AWS Containers ブログには、AWS Load Balancer Controller を使用したチュートリアルが含まれています。AWS Load Balancer Controller を使用したブルー/グリーンデプロイ、Canary デプロイ、A/B テスト

Canary デプロイを使用する

Canary デプロイは、変更からリスクを大幅に排除できる Blue/Green デプロイのバリアントです。このデプロイ戦略では、古いデプロイと並行してポッド数が少ない新しいデプロイを作成し、トラフィックのごく一部を新しいデプロイに振り向けます。メトリクスで、新しいバージョンのパフォーマンスが既存のバージョンと同じかそれ以上であることが示された場合は、新しいデプロイへのトラフィックを徐々に増やし、すべてのトラフィックが新しいデプロイに転送されるまでスケールアップします。問題が発生した場合は、すべてのトラフィックを古いデプロイにルーティングし、新しいデプロイへのトラフィックの送信を停止できます。

Kubernetes には Canary デプロイを実行するネイティブな方法はありませんが、Istio で Flagger などのツールを使用できます。

ヘルスチェックと自己修復

バグのないソフトウェアはありませんが、Kubernetes はソフトウェア障害の影響を最小限に抑えるのに役立ちます。以前は、アプリケーションがクラッシュした場合、アプリケーションを手動で再起動して状況を修復する必要がありました。Kubernetes を使用すると、ポッド内のソフトウェア障害を検出し、新しいレプリカに自動的に置き換えることができます。Kubernetes を使用すると、アプリケーションの状態をモニタリングし、異常なインスタンスを自動的に置き換えることができます。

Kubernetes は 3 種類のヘルスチェックをサポートしています。

1. ライブネスプローブ
2. スタートアッププローブ (Kubernetes バージョン 1.16 以降でサポート)
3. 準備状況プローブ

Kubernetes エージェントである Kubelet は、上記のすべてのチェックを実行する責任があります。Kubelet は Pod の状態を 3 つの方法で確認できます。kubelet は、Pod のコンテナ内でシェルコマンドを実行するか、コンテナに HTTP GET リクエストを送信するか、指定されたポートで TCP ソケットを開くことができます。

コンテナ内でシェルスクリプトを実行する `exec`ベースのプローブを選択した場合は、`timeoutSeconds`値の有効期限が切れる前にシェルコマンドが終了していることを確認します。そうしないと、ノードに`<defunct>`プロセスが発生し、ノードに障害が発生します。

推奨事項

Liveness Probe を使用して異常なポッドを削除する

Liveness プローブは、プロセスが実行され続けるがアプリケーションが応答しなくなるデッドロック状態を検出できます。たとえば、ポート 80 でリッスンするウェブサービスを実行している場合は、Pod のポート 80 で HTTP GET リクエストを送信するように Liveness プローブを設定できます。Kubelet は定期的に GET リクエストを Pod に送信し、レスポンスを期待します。Pod が 200 ~ 399 の間に応答した場合、kubelet は Pod が正常であると見なします。そうしないと、Pod は異常としてマークされます。Pod がヘルスチェックに継続的に失敗した場合、kubelet はそれを終了します。

を使用して`initialDelaySeconds`、最初のプローブを遅延させることができます。

Liveness Probe を使用する場合は、Kubernetes がすべての Pod を置き換えようとし、アプリケーションがオフラインになるため、すべての Pod が同時に Liveness Probe に失敗する状況でアプリ

ケーションが実行されないようにしてください。さらに、Kubernetes は引き続き Liveness Probes にも失敗する新しい Pod を作成し、コントロールプレーンに不要な負荷をかけます。外部データベースなど、Pod の外部にある要素に依存するように Liveness Probe を設定しないでください。つまり、応答しない external-to-your-Pod データベースは、Pod が Liveness Probes に失敗しないようにする必要があります。

Sandor Szücs の投稿「LIVENESS PROBES ARE DANGEROUS」では、プローブの設定ミスによって発生する可能性のある問題について説明しています。

起動に時間がかかるアプリケーションに Startup Probe を使用する

アプリの起動にさらに時間が必要な場合は、Startup Probe を使用して Liveness and Readiness Probe を遅延させることができます。たとえば、データベースからキャッシュをハイドレートする必要がある Java アプリケーションは、完全に機能するまでに最大 2 分かかる場合があります。Liveness または Readiness Probe が完全に機能するまでは失敗する可能性があります。Startup Probe を設定すると、Liveness または Readiness Probe が実行される前に Java アプリが正常になります。

Startup Probe が成功するまで、他のすべてのプローブは無効になります。Kubernetes がアプリケーションの起動を待機する最大時間を定義できます。最大設定時間が経過しても Pod が起動プローブに失敗した場合、Pod は終了し、新しい Pod が作成されます。

Startup Probe は Liveness Probe に似ています。失敗すると、Pod が再作成されます。Ricardo A. が記事「Fantastic Probes and How To Configure Them」で説明しているように、アプリケーションの起動時間が予測不可能な場合は、スタートアッププローブを使用する必要があります。アプリケーションを起動するのに 10 秒かかることがわかっている場合は、`initialDelaySeconds`代わりにで Liveness/Readiness Probe を使用する必要があります。

Readiness Probe を使用して部分的な利用不能を検出する

Liveness プローブは、Pod を終了 (したがってアプリを再起動) して解決されたアプリの障害を検出しますが、Readiness Probe はアプリが一時的に使用できない可能性のある状態を検出します。このような状況では、アプリが一時的に応答しなくなることがありますが、このオペレーションが完了すると再び正常であることが予想されます。

たとえば、強力なディスク I/O オペレーション中に、アプリケーションが一時的にリクエストを処理できない場合があります。ここでは、アプリケーションの Pod を終了することは解決策ではありません。同時に、Pod に送信される追加のリクエストが失敗する可能性があります。

Readiness Probe を使用すると、アプリで一時的な使用不能を検出し、再び機能するまで Pod へのリクエストの送信を停止できます。ポッドの再作成に失敗する Liveness Probe とは異なり、Readiness Probe が失敗すると、ポッドは Kubernetes Service からトラフィックを受信しません。準備プローブが成功すると、Pod はサービスからのトラフィックの受信を再開します。

Liveness Probe と同様に、Pod の外部にあるリソース (データベースなど) に依存する Readiness Probe を設定しないでください。設定の不十分な準備状況によってアプリケーションが機能しなくなるシナリオを次に示します。アプリケーションのデータベースに到達できない場合に Pod の準備状況プローブが失敗すると、同じヘルスチェック基準を共有するため、他の Pod レプリカも同時に失敗します。この方法でプローブを設定すると、データベースが使用できないたびに Pod の準備プローブが失敗し、Kubernetes はすべての Pod へのトラフィックの送信を停止します。

Readiness Probes を使用すると、デプロイの更新にかかる時間が長くなる可能性があります。準備プローブが成功しない限り、新しいレプリカはトラフィックを受信しません。それまでは、古いレプリカは引き続きトラフィックを受信します。

中断への対応

Pod の有効期間は限られています。長時間稼働している Pod がある場合でも、Pod が正常に終了することを確認するのが賢明です。アップグレード戦略によっては、Kubernetes クラスターのアップグレードで新しいワーカーノードの作成が必要になる場合があります。そのため、すべての Pod を新しいノードで再作成する必要があります。適切な終了処理と Pod Disruption Budgets を使用すると、Pod が古いノードから削除され、新しいノードで再作成されるため、サービスの中断を回避できます。

ワーカーノードをアップグレードする推奨方法は、新しいワーカーノードを作成し、古いノードを終了することです。ワーカーノードを終了する前に、drain終了する必要があります。ワーカーノードがドレインされると、そのすべてのポッドは安全に削除されます。ここでのキーワードは安全です。ワーカーのポッドが削除されると、単にSIGKILLシグナルが送信されるわけではありません。代わりに、削除されるポッド内の各コンテナのメインプロセス (PID 1) にSIGTERMシグナルが送信されます。SIGTERM シグナルが送信されると、Kubernetes はSIGKILLシグナルが送信される前にプロセスに時間 (猶予期間) を与えます。この猶予期間はデフォルトで 30 秒です。kubectl の grace-periodフラグを使用するか、Podspec terminationGracePeriodSecondsで宣言することで、デフォルトを上書きできます。

```
kubectl delete pod <pod name> –grace-period=<seconds>
```

通常、メインプロセスに PID 1 がないコンテナがあります。Python ベースのサンプルコンテナを考えてみましょう。

```
$ kubectl exec python-app -it ps
 PID USER TIME COMMAND
 1   root 0:00 {script.sh} /bin/sh ./script.sh
 5   root 0:00 python app.py
```

この例では、シェルスクリプトは SIGTERMを受け取ります。この例では Python アプリケーションであるメインプロセスはSIGTERMシグナルを取得しません。Pod が終了すると、Python アプリケーションは突然強制終了されます。これは、コンテナENTRYPOINTの を変更して Python アプリケーションを起動することで修正できます。または、dumb-init などのツールを使用して、アプリケーションがシグナルを処理できるようにすることもできます。

コンテナフックを使用して、コンテナの開始または停止時にスクリプトまたは HTTP リクエストを実行することもできます。PreStop フックアクションは、コンテナがSIGTERMシグナルを受信する前に実行され、このシグナルが送信される前に完了する必要があります。terminationGracePeriodSeconds 値は、SIGTERMシグナルが送信されたときではなく、PreStopフックアクションの実行が開始されたときから適用されます。

推奨事項

Pod Disruption Budgets で重要なワークロードを保護する

アプリケーションのレプリカ数が宣言されたしきい値を下回った場合、Pod Disruption Budget または PDB は一時的に削除プロセスを停止できます。使用可能なレプリカの数がしきい値を超えると、削除プロセスは続行されます。PDB を使用して、 minAvailableとレプリカmaxUnavailableの数を宣言できます。たとえば、アプリのコピーを 3 つ以上使用できるようにする場合は、PDB を作成できます。

```
apiVersion: policy/v1beta1
kind: PodDisruptionBudget
metadata:
  name: my-svc-pdb
spec:
  minAvailable: 3
  selector:
    matchLabels:
      app: my-svc
```

上記の PDB ポリシーは、3 つ以上のレプリカが使用可能になるまでエビクションプロセスを停止するよう Kubernetes に指示します。ノードドレイニングは を尊重しますPodDisruptionBudgets。EKS マネージド型ノードグループのアップグレード中、ノードは 15

分のタイムアウトでドレインされます。15 分後、更新が強制されない場合 (オプションは EKS コンソールでローリング更新と呼ばれます)、更新は失敗します。更新が強制されると、ポッドは削除されます。

セルフマネージド型ノードの場合、AWS Node Termination Handler などのツールを使用して、EC2 メンテナンスイベントや EC2 スポットの中断などEC22 インスタンスが使用できなくなる可能性のあるイベントに Kubernetes コントロールプレーンが適切に応答するようにすることもできます。EC2 Kubernetes API を使用してノードをコーディングし、新しい Pod がスケジュールされていないことを確認してからドレインし、実行中の Pod を終了します。

Pod anti-affinity を使用して、異なるノードでデプロイの Pod をスケジュールし、ノードのアップグレード中の PDB 関連の遅延を回避できます。

カオスエンジニアリングの練習

カオスエンジニアリングは、本番環境の乱れに耐えられるシステムの能力の信頼性を構築するために、分散システムで実験する分野です。

Dominik Tornow は、ブログで、Kubernetes は「ユーザーがシステムの望ましい状態をシステムに提供する」宣言システムであると説明しています。次に、システムは現在の状態と望ましい状態を考慮して、現在の状態から望ましい状態に移行するコマンドのシーケンスを決定します。「�� つまり、Kubernetes は常に目的の状態を保存し、システムが逸脱すると、Kubernetes は状態を復元するアクションを実行します。たとえば、ワーカーノードが使用できなくなった場合、Kubernetes はポッドを別のワーカーノードに再スケジュールします。同様に、`replica`クラッシュした場合、Deployment Contoller は新しい を作成します`replica`。このようにして、Kubernetes コントローラーは障害を自動的に修正します。

Gremlin などのカオスエンジニアリングツールは、Kubernetes クラスターの耐障害性をテストし、単一障害点を特定するのに役立ちます。クラスター (およびそれ以降) に人為的カオスを導入するツールは、システムの弱点を発見し、ボトルネックや設定ミスを特定し、制御された環境で問題を修正する機会を与える可能性があります。カオスエンジニアリングの哲学は、予期しないダウンタイムを最小限に抑えるために、意図的にモノを破壊し、インフラストラクチャをストレステストすることを提唱しています。

Service Mesh を使用する

サービスメッシュを使用して、アプリケーションの耐障害性を向上させることができます。サービスメッシュによりservice-to-service通信が可能になり、マイクロサービスネットワークのオブザーバビリティが向上します。ほとんどのサービスメッシュ製品は、アプリケーションのネットワークトラフィックを傍受して検査する各サービスと一緒に小さなネットワークプロキシを実行することで機能

します。アプリケーションを変更せずに、アプリケーションをメッシュに配置することができます。サービスプロキシの組み込み機能を使用すると、ネットワーク統計の生成、アクセスログの作成、分散トレースのアウトバウンドリクエストへの HTTP ヘッダーの追加を行うことができます。

サービスメッシュは、自動リクエストの再試行、タイムアウト、サーキットブレーク、レート制限などの機能により、マイクロサービスの耐障害性を高めるのに役立ちます。

複数のクラスターを操作する場合は、サービスメッシュを使用してクラスター間のservice-to-service通信を有効にできます。

サービスメッシュ

- Istio
- LinkerD
- コンサル

オブザーバビリティ

オブザーバビリティは、モニタリング、ログ記録、トレースを含む包括的な用語です。マイクロサービスベースのアプリケーションは、本質的に分散されます。単一のシステムのモニタリングで十分なモノリシックアプリケーションとは異なり、分散アプリケーションアーキテクチャでは、各コンポーネントのパフォーマンスをモニタリングする必要があります。クラスターレベルのモニタリング、ログ記録、分散トレースシステムを使用して、顧客を混乱させる前にクラスター内の問題を特定できます。

トラブルシューティングとモニタリング用の Kubernetes 組み込みツールは限られています。metrics-server はリソースメトリクスを収集してメモリに保存しますが、保持しません。kubectl を使用して Pod のログを表示できますが、Kubernetes は自動的にログを保持しません。また、分散トレースの実装は、アプリケーションコードレベルまたはサービスメッシュを使用して行われます。

Kubernetes の拡張性はここで光ります。Kubernetes を使用すると、任意の一元的なモニタリング、ログ記録、トレースソリューションを導入できます。

推奨事項

アプリケーションをモニタリングする

最新のアプリケーションでモニタリングする必要があるメトリクスの数は継続的に増加しています。これは、お客様の課題の解決に集中できるように、アプリケーションを自動的に追跡する方法がある

場合に役立ちます。Prometheus や CloudWatch Container Insights などのクラスター全体のモニタリングツールは、クラスターとワークロードをモニタリングし、問題が発生するとき、または問題が発生する前にシグナルを提供できます。

モニタリングツールを使用すると、運用チームがサブスクライブできるアラートを作成できます。イベントのアラームをアクティブ化するルールを検討してください。イベントが悪化すると、停止したり、アプリケーションのパフォーマンスに影響を与えたりする可能性があります。

モニタリングすべきメトリクスが不明な場合は、次の方法からヒントを得ることができます。

- RED メソッド。リクエスト、エラー、期間を表します。
- USE メソッド。使用率、飽和度、エラーを表します。

Sysdig の記事 Kubernetes でのアラートのベストプラクティスには、アプリケーションの可用性に影響を与える可能性のあるコンポーネントの包括的なリストが含まれています。

Prometheus クライアントライブラリを使用してアプリケーションメトリクスを公開する

アプリケーションの状態をモニタリングし、標準メトリクスを集約することに加えて、Prometheus クライアントライブラリを使用してアプリケーション固有のカスタムメトリクスを公開し、アプリケーションのオブザーバビリティを向上させることもできます。

一元的なログ記録ツールを使用してログを収集し、保持する

EKS でのログ記録は、コントロールプレーンログとアプリケーションログの 2 つのカテゴリに分類されます。EKS コントロールプレーンのログ記録は、コントロールプレーンからアカウントの CloudWatch Logs に直接監査ログと診断ログを提供します。アプリケーションログは、クラスター内で実行されている Pod によって生成されるログです。アプリケーションログには、ビジネスロジックアプリケーションを実行する Pod と、CoreDNS、Cluster Autoscaler、Prometheus などの Kubernetes システムコンポーネントによって生成されたログが含まれます。

EKS には 5 種類のコントロールプレーンログが用意されています。

1. Kubernetes API サーバーコンポーネントログ
2. 監査
3. オーセンティケーター
4. コントローラーマネージャー
5. スケジューラー

コントローラーマネージャーとスケジューラのログは、ボトルネックやエラーなどのコントロールプレーンの問題の診断に役立ちます。デフォルトでは、EKS コントロールプレーンログは CloudWatch Logs に送信されません。コントロールプレーンのログ記録を有効にし、アカウント内のクラスターごとにキャプチャする EKS コントロールプレーンログのタイプを選択できます。

アプリケーションログを収集するには、Fluent Bit、Fluentd、CloudWatch Container Insights などのログアグリゲータツールをクラスターにインストールする必要があります。

Kubernetes ログアグリゲータツールは DaemonSets として実行され、ノードからコンテナログをスクレイプします。その後、アプリケーションログは一元管理された保存先に送信されます。例えば、CloudWatch Container Insights は Fluent Bit または Fluentd を使用してログを収集し、CloudWatch Logs に送信して保存できます。Fluent Bit と Fluentd は、Elasticsearch や InfluxDB などの多くの一般的なログ分析システムをサポートしているため、Fluent ビットや Fluentd のログ設定を変更することで、ログのストレージバックエンドを変更できます。

分散トレースシステムを使用してボトルネックを特定する

一般的な最新のアプリケーションには、ネットワーク上に分散されたコンポーネントがあり、その信頼性は、アプリケーションを構成する各コンポーネントが適切に機能するかどうかによって異なります。分散トレースソリューションを使用して、リクエストのフローとシステムの通信方法を理解できます。トレースは、アプリケーションネットワークにボトルネックが存在する場所を示し、カスケード障害を引き起こす可能性のある問題を防ぐことができます。

アプリケーションにトレースを実装するには、2 つのオプションがあります。共有ライブラリを使用してコードレベルで分散トレースを実装するか、サービスメッシュを使用できます。

コードレベルでトレースを実装すると、不利になる可能性があります。この方法では、コードを変更する必要があります。Polyglot アプリケーションがある場合、これはさらに複雑になります。また、サービス全体でさらに別のライブラリを維持する責任もあります。

LinkerD や Istio などの Service Mesh を使用して、アプリケーションコードへの最小限の変更でアプリケーションに分散トレースを実装できます。サービスメッシュを使用して、メトリクスの生成、ログ記録、トレースを標準化できます。

AWS X-Ray などのトレースツールである Jaeger は、共有ライブラリとサービスメッシュの両方の実装をサポートしています。

(共有ライブラリとサービスメッシュ) の両方の実装をサポートする AWS X-Ray や Jaeger などのトレースツールを使用することを検討してください。これにより、後でサービスメッシュを採用する場合にツールを切り替える必要がなくなります。

EKS コントロールプレーン

Amazon Elastic Kubernetes Service (EKS) は、独自の Kubernetes コントロールプレーンまたはワーカーノードをインストール、運用、保守することなく、AWS で Kubernetes を簡単に実行できるようにするマネージド Kubernetes サービスです。アップストリーム Kubernetes を実行し、Kubernetes 準拠の認定を受けています。この準拠により、EC2 またはオンプレミスにインストールできるオープンソースのコミュニティバージョンと同様に、EKS は Kubernetes APIs を確実にサポートします。アップストリーム Kubernetes で実行されている既存のアプリケーションは、Amazon EKS と互換性があります。

EKS は、Kubernetes コントロールプレーンノードの可用性とスケーラビリティを自動的に管理し、異常なコントロールプレーンノードを自動的に置き換えます。

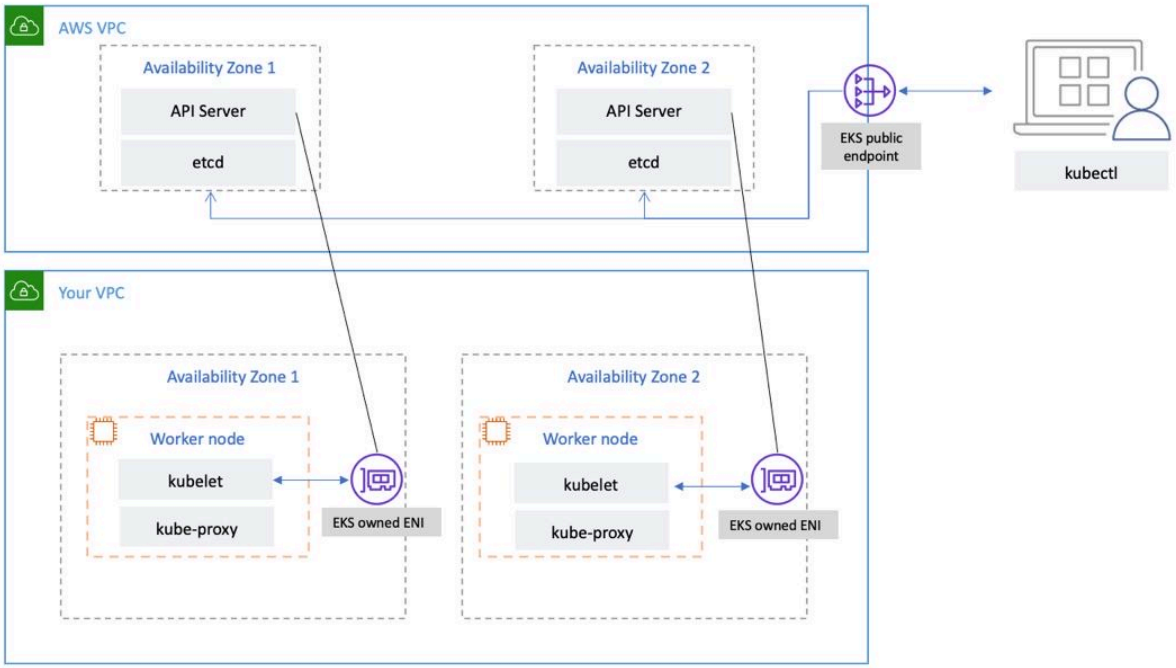
EKS アーキテクチャ

EKS アーキテクチャは、Kubernetes コントロールプレーンの可用性と耐久性を損なう可能性のある単一障害点を排除するように設計されています。

EKS によって管理される Kubernetes コントロールプレーンは、EKS マネージド VPC 内で実行されます。EKS コントロールプレーンは、Kubernetes API サーバーノード etcd クラスターで構成されます。API サーバー、スケジューラなどのコンポーネントを実行し、自動スケーリンググループで`kube-controller-manager`実行する Kubernetes API サーバーノード。EKS は、AWS リージョン内の個別のアベイラビリティーゾーン (AZs) で最低 2 つの API サーバーノードを実行します。同様に、耐久性のために、 etcd サーバーノードは 3 つの AZs にまたがる Auto Scaling グループでも実行されます。EKS は各 AZ で NAT ゲートウェイを実行し、API サーバーと etcd サーバーはプライベートサブネットで実行されます。このアーキテクチャにより、単一の AZ のイベントが EKS クラスターの可用性に影響を与えなくなります。

新しいクラスターを作成すると、Amazon EKS はクラスターとの通信に使用するマネージド Kubernetes API サーバーの高可用性エンドポイントを作成します (などのツールを使用`kubectl`)。マネージドエンドポイントは NLB を使用して Kubernetes API サーバーの負荷分散を行います。EKS は、ワーカーノードとの通信を容易にするために、異なる AZs に 2 つの ENI もプロビジョニングします。

EKS データプレーンネットワーク接続



Kubernetes クラスターの API サーバーがパブリックインターネット (パブリックエンドポイントを使用) から到達可能か、VPC (EKS マネージド ENIs) または両方から到達可能かを設定できます。

ユーザーとワーカーノードがパブリックエンドポイントまたは EKS マネージド ENI を使用して API サーバーに接続するかどうかにかかわらず、接続には冗長パスがあります。

推奨事項

以下の推奨事項を確認してください。

コントロールプレーンメトリクスのモニタリング

Kubernetes API メトリクスをモニタリングすると、コントロールプレーンのパフォーマンスに関するインサイトが得られ、問題を特定できます。異常なコントロールプレーンは、クラスター内で実行されているワークロードの可用性を損なう可能性があります。たとえば、コントローラーの書き込みが不十分な場合、API サーバーが過負荷になり、アプリケーションの可用性に影響する可能性があります。

Kubernetes は`/metrics`、エンドポイントでコントロールプレーンメトリクスを公開します。

を使用して公開されているメトリクスを表示できます`kubectl`。

```
kubectl get --raw /metrics
```

これらのメトリクスは Prometheus テキスト形式で表されます。

Prometheus を使用して、これらのメトリクスを収集して保存できます。2020 年 5 月、CloudWatch は CloudWatch Container Insights で Prometheus メトリクスをモニタリングするためのサポートを追加しました。そのため、Amazon CloudWatch を使用して EKS コントロールプレーンをモニタリングすることもできます。チュートリアルを使用して、新しい Prometheus スクレイプターゲットの追加: Prometheus KPI サーバーメトリクスを使用してメトリクスを収集し、CloudWatch ダッシュボードを作成してクラスターのコントロールプレーンをモニタリングできます。

Kubernetes API サーバーのメトリクスは、こちらで確認できます。たとえば、`apiserver_request_duration_seconds` は API リクエストの実行にかかる時間を指定できます。

これらのコントロールプレーンメトリクスのモニタリングを検討してください。

API サーバー

メトリクス	説明
`apiserver_request_total`	verb、dry run 値、グループ、バージョン、リソース、スコープ、コンポーネント、HTTP レスポンスコードごとに分割された apiserver リクエストのカウンター。
`apiserver_request_duration_seconds*`	動詞、ドライラン値、グループ、バージョン、リソース、サブリソース、スコープ、コンポーネントごとのレスポンスレイテンシーヒストグラムを秒単位で表示します。
`apiserver_admission_controller_admission_duration_seconds*`	名前で識別され、オペレーションと API リソースとタイプ (検証または承認) ごとに分割された、秒単位のアドミッションコントローラーレイテンシーヒストグラム。
`apiserver_admission_webhook_rejection_count`	アドミッションウェブフック拒否の数。名前、オペレーション、拒否コード、タイプ (検証または承認) 、エラータイプ (calling_webhook_e

メトリクス	説明
	rror、apiserver_internal_error、no_error) で識別されます
rest_client_request_duration_seconds*	レイテンシーヒストグラムを秒単位でリクエストします。動詞と URL によって分類されます。
rest_client_requests_total	ステータスコード、メソッド、ホストでパーティション分割された、HTTP リクエスト数。

- ヒストグラムメトリクスには、_bucket、_sum、および _count サフィックスが含まれます。

etcd

メトリクス	説明
etcd_request_duration_seconds*	Etcd リクエストのレイテンシーヒストグラ

メトリクス	説明
	ムは、オペレーションとオブジェクトタイプごとに秒単位です。

メトリクス	説明
`apiserver_storage_db_total_size_in_bytes` または `apiserver_storage_size_bytes` (EKS v1.28 以降)	Etcd データベースサイズ。

- ヒストグラムメトリクスには、_bucket、_sum、および _count サフィックスが含まれます。

Kubernetes モニタリング概要ダッシュボードを使用して、Kubernetes API サーバーのリクエストとレイテンシーおよび etcd レイテンシーメトリクスを視覚化およびモニタリングすることを検討してください。

> **Important**
>
> データベースのサイズ制限を超えると、 etcd はスペースなしアラームを発行し、それ以降の書き込みリクエストの実行を停止します。つまり、クラスターは読み取り専用になり、新しいポッドの作成、デプロイのスケーリングなどのオブジェクトを変更するすべてのリクエストは、クラスターの API サーバーによって拒否されます。

クラスター認証

EKS は現在、ベアラー/サービスアカウントトークンとウェブフックトークン認証を使用する IAM 認証の 2 種類の認証をサポートしています。ユーザーが Kubernetes API を呼び出すと、ウェブフックはリクエストに含まれる認証トークンを IAM に渡します。トークンは、ベース 64 署名付き URL であり、AWS コマンドラインインターフェイス (AWS CLI) によって生成されます。

EKS クラスターを作成する IAM ユーザーまたはロールは、クラスターへのフルアクセスを自動的に取得します。EKS クラスターへのアクセスを管理するには、aws-auth 設定マップを編集します。

aws-auth 設定マップを誤って設定し、クラスターにアクセスできなくなった場合でも、クラスター作成者のユーザーまたはロールを使用して EKS クラスターにアクセスできます。

万一、AWS リージョンで IAM サービスを使用できない場合は、Kubernetes サービスアカウントのベアラートークンを使用してクラスターを管理することもできます。

クラスター内のすべてのアクションを実行できるsuper-adminアカウントを作成します。

```
kubectl -n kube-system create serviceaccount super-admin
```

super-admin cluster-admin ロールを付与するロールバインディングを作成します。

```
kubectl create clusterrolebinding super-admin-rb --clusterrole=cluster-admin --
serviceaccount=kube-system:super-admin
```

サービスアカウントのシークレットを取得します。

```
SECRET_NAME=`kubectl -n kube-system get serviceaccount/super-admin -o
 jsonpath='{.secrets[0].name}'`
```

シークレットに関連付けられたトークンを取得します。

```
TOKEN=`kubectl -n kube-system get secret $SECRET_NAME -o jsonpath='{.data.token}'|
 base64 --decode`
```

サービスアカウントとトークンを に追加しますkubeconfig。

```
kubectl config set-credentials super-admin --token=$TOKEN
```

スーパー管理者アカウントを使用するにはkubeconfig、 で現在のコンテキストを設定します。

```
kubectl config set-context --current --user=super-admin
```

最終kubeconfig版は次のようになります。

```
apiVersion: v1
clusters:
```

```
- cluster:
    certificate-authority-data:<REDACTED>
    server: https://<CLUSTER>.gr7.us-west-2.eks.amazonaws.com
  name: arn:aws:eks:us-west-2:<account number>:cluster/<cluster name>
contexts:
- context:
    cluster: arn:aws:eks:us-west-2:<account number>:cluster/<cluster name>
    user: super-admin
  name: arn:aws:eks:us-west-2:<account number>:cluster/<cluster name>
current-context: arn:aws:eks:us-west-2:<account number>:cluster/<cluster name>
kind: Config
preferences: {}
users:
#- name: arn:aws:eks:us-west-2:<account number>:cluster/<cluster name>
#  user:
#    exec:
#      apiVersion: client.authentication.k8s.io/v1beta1
#      args:
#      - --region
#      - us-west-2
#      - eks
#      - get-token
#      - --cluster-name
#      - <<cluster name>>
#      command: aws
#      env: null
- name: super-admin
  user:
    token: <<super-admin sa's secret>>
```

アドミッションウェブフック

Kubernetes には、アドミッションウェブフックの検証とアドミッションウェブフックの変更の 2 種類のアドミッションウェブフックがあります。これにより、ユーザーは kubernetes API を拡張し、API によって受け入れられる前にオブジェクトを検証または変更できます。これらのウェブフックの設定が悪いと、クラスターの重要なオペレーションがブロックされ、EKS コントロールプレーンが不安定になる可能性があります。

クラスターの重要なオペレーションに影響を与えないようにするには、次のように「キャッチオール」ウェブフックを設定しないでください。

```
- name: "pod-policy.example.com"
```

```
  rules:
  - apiGroups:   ["*"]
    apiVersions: ["*"]
    operations:  ["*"]
    resources:   ["*"]
    scope: "*"
```

または、タイムアウトが 30 秒未満のフェイルオープンポリシーがウェブフックにあることを確認し、ウェブフックが使用できない場合にクラスターの重要なワークロードが損なわれないようにします。

安全でないポッドをブロックする sysctls

Sysctl は、ランタイム中にユーザーがカーネルパラメータを変更できるようにする Linux ユーティリティです。これらのカーネルパラメータは、ネットワーク、ファイルシステム、仮想メモリ、プロセス管理など、オペレーティングシステムの動作のさまざまな側面を制御します。

Kubernetes では、ポッドにsysctlプロファイルを割り当てることができます。Kubernetes は、安全と安全でないsystclsに分類されます。Safe sysctlsはコンテナまたは Pod に名前空間化され、設定してもノードまたはノード自体の他の Pod には影響しません。一方、安全でない sysctl は、他の Pod を中断したり、ノードを不安定にしたりする可能性があるため、デフォルトでは無効になっています。

安全でない sysctlsはデフォルトで無効になっているため、kubelet は安全でないsysctlプロファイルを持つ Pod を作成しません。このような Pod を作成すると、スケジューラはそのような Pod をノードに繰り返し割り当てますが、ノードは起動に失敗します。この無限ループは最終的にクラスターコントロールプレーンに負担をかけ、クラスターを不安定にします。

OPA Gatekeeper または Kyverno を使用して、安全でない のポッドを拒否することを検討してくださいsysctls。

クラスターのアップグレードの処理

2021 年 4 月以降、Kubernetes リリースサイクルは年間 4 つのリリース (四半期に 1 回) から年間 3 つのリリースに変更されました。新しいマイナーバージョン (1.21 や 1.22 など) が約 15 週間ごとにリリースされます。Kubernetes 1.19 以降、各マイナーバージョンは最初のリリースから約 12 か月間サポートされています。Kubernetes v1.28 の登場により、コントロールプレーンとワーカーノード間の互換性スキューが n-2 から n-3 のマイナーバージョンに拡張されました。詳細については、「クラスターアップグレードのベストプラクティス」を参照してください。

クラスターエンドポイントの接続

Amazon EKS (Elastic Kubernetes Service) を使用する場合、Kubernetes コントロールプレーンのスケーリングやパッチ適用などのイベント中に接続のタイムアウトやエラーが発生することがあります。これらのイベントにより、kube-apiserver インスタンスが置き換えられ、FQDN を解決するときに異なる IP アドレスが返される可能性があります。このドキュメントでは、Kubernetes API コンシューマーが信頼性の高い接続を維持するためのベストプラクティスの概要を説明します。

> **Note**
>
> これらのベストプラクティスを実装するには、新しい DNS 再解決および再試行戦略を効果的に処理するために、クライアント設定またはスクリプトの更新が必要になる場合があります。

主な問題は、DNS クライアント側のキャッシュと、EKS エンドポイントの古い IP アドレス - パブリックエンドポイントの場合はパブリック NLB、プライベートエンドポイントの場合は X-ENI の可能性です。kube-apiserver インスタンスが置き換えられると、完全修飾ドメイン名 (FQDN) が新しい IP アドレスに解決される場合があります。ただし、AWS マネージド Route 53 ゾーンで 60 秒に設定されている DNS 有効期限 (TTL) の設定により、クライアントは短期間、古い IP アドレスを引き続き使用できます。

これらの問題を軽減するには、Kubernetes API コンシューマー (kubectl、CI/CD パイプライン、カスタムアプリケーションなど) に次のベストプラクティスを実装する必要があります。

- DNS 再解決の実装
- バックオフとジッターを使用して再試行を実装します。例えば、この記事の「障害発生」を参照してください。
- クライアントタイムアウトを実装します。長時間実行されるリクエストがアプリケーションをブロックしないように、適切なタイムアウトを設定します。一部の Kubernetes クライアントライブラリ、特に OpenAPI ジェネレーターによって生成されたライブラリでは、カスタムタイムアウトを簡単に設定できない場合があります。
  - kubectl を使用した例 1:

```
kubectl get pods --request-timeout 10s # default: no timeout
```

  - Python を使用した例 2: Kubernetes クライアントが _request_timeout パラメータを提供する

これらのベストプラクティスを実装することで、Kubernetes API を操作する際のアプリケーションの信頼性と耐障害性を大幅に向上させることができます。これらの実装は、特にシミュレートされた障害条件下で徹底的にテストし、実際のスケーリングまたはパッチ適用イベント中に期待どおりに動作することを確認してください。

大規模なクラスターの実行

EKS は、コントロールプレーンインスタンスの負荷をアクティブにモニタリングし、高性能を確保するために自動的にスケーリングします。ただし、大規模なクラスターを実行する場合は、Kubernetes 内の潜在的なパフォーマンスの問題と制限、および AWS サービスのクォータを考慮する必要があります。

- ProjectCalico チームによって実行されたテストに従ってkube-proxy、 を iptables モードで使用すると、1000 を超えるサービスを持つクラスターでネットワークレイテンシーが発生する可能性があります。解決策は、 ipvskube-proxyモードでの実行に切り替えることです。
- また、CNI が Pod の IP アドレスをリクエストする必要がある場合や、新しい EC2 インスタンスを頻繁に作成する必要がある場合に、EC2 API リクエストのスロットリングが発生することがあります。 EC2 IP アドレスをキャッシュするように CNI を設定することで、EC2 API の呼び出しを減らすことができます。より大きな EC2 インスタンスタイプを使用して EC2 スケーリングイベントを減らすことができます。

その他のリソース:

- Amazon EKS ワーカーノードのクラスターネットワークの説明を解除する
- Amazon EKS クラスターエンドポイントのアクセスコントロール
- AWS re:Invent 2019: Amazon EKS under the hood (CON421-R1)

EKS データプレーン

高可用性で回復力のあるアプリケーションを運用するには、高可用性で回復力のあるデータプレーンが必要です。伸縮自在なデータプレーンにより、Kubernetes はアプリケーションを自動的にスケーリングして修復できます。回復力のあるデータプレーンは、2 つ以上のワーカーノードで構成され、ワークロードに合わせて増減し、障害から自動的に回復できます。

EKS を使用するワーカーノードには、EKS Auto Mode マネージドノード、EC2 インスタンス、Fargate の複数の選択肢があります。 https://docs.aws.amazon.com/eks/latest/userguide/fargate.html

EKS Auto Mode は、回復力のあるデータプレーンへの最も簡単なパスを提供します。Auto Mode は、Kubernetes クラスターの AWS 管理をクラスター自体を超えて拡張し、AWS がワークロードのスムーズな運用を可能にするインフラストラクチャを設定および管理できるようにします。Auto Mode は、Kubernetes が Pod をスケーリングするときにデータプレーンを自動的にスケールアップまたはスケールダウンし、クラスター内のノードが現在実行中のワークロードに対して適切かつ費用対効果の高いサイズになるように継続的に機能します。

EC2 インスタンスを選択した場合は、ワーカーノードを自分で管理することも、EKS マネージド型ノードグループを使用することもできます。Auto Mode、マネージドワーカーノード、セルフマネージドワーカーノード、Fargate が混在したクラスターを持つことができます。

Fargate は、分離されたコンピューティング環境で各 Pod を実行します。Fargate で実行されている各 Pod は、独自のワーカーノードを取得します。Fargate は、Kubernetes がポッドをスケーリングするときにデータプレーンを自動的にスケーリングします。水平ポッドオートスケーラーを使用して、データプレーンとワークロードの両方をスケールできます。

EC2 ワーカーノードをスケールする (AWS によって自動的に実行される EKS Auto Mode を使用しない場合) 推奨方法は、Karpenter、Kubernetes Cluster Autoscaler、または EC2 Auto Scaling グループを使用することです。

推奨事項

ワーカーノードとワークロードを複数の AZs

ワーカーノードと Pod を複数の AZ で実行することで、個々の AZs の障害からワークロードを保護できます。ノードを作成するサブネットを使用して、ワーカーノードが作成される AZ を制御できます。

AZs 間でポッドを分散するための推奨方法は、ポッドにトポロジースプレッド制約を使用することです。EKS Auto Mode や Karpenter などの自動スケーリング機能は、トポロジの分散制約を認識し、制約を満たすために正しい AZs でノードを自動的に起動します。

以下のデプロイでは、可能な場合はポッドを AZs に分散し、そうでない場合はとにかくポッドを実行できるようにします。

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: web-server
spec:
```

```
  replicas: 3
  selector:
    matchLabels:
      app: web-server
  template:
    metadata:
      labels:
        app: web-server
    spec:
      topologySpreadConstraints:
        - maxSkew: 1
          whenUnsatisfiable: ScheduleAnyway
          topologyKey: topology.kubernetes.io/zone
          labelSelector:
            matchLabels:
              app: web-server
      containers:
      - name: web-app
        image: nginx
        resources:
          requests:
            cpu: 1
```

Note

kube-scheduler は、それらのラベルを持つノードを介してのみトポロジドメインを認識します。上記のデプロイが 1 つのゾーンにのみノードを持つクラスターにデプロイされる場合、すべてのポッドkube-schedulerは他のゾーンを認識しないため、それらのノードでスケジュールされます。このトポロジがスケジューラで想定どおりに機能するには、ノードがすべてのゾーンに既に存在している必要があります。トポロジスプレッド制約のminDomainsプロパティは、この問題を回避するために実行されているノードがある場合でも、対象となるドメインの数をスケジューラに通知するために使用されます。

Warning

whenUnsatisfiable を に設定するとDoNotSchedule、トポロジの分散制約が満たされない場合、ポッドはスケジュールできなくなります。これは、トポロジの分散制約に違反するのではなく、ポッドを実行しない方が望ましい場合にのみ設定する必要があります。

古いバージョンの Kubernetes では、ポッドアンチアフィニティルールを使用して、複数の AZs 間でポッドをスケジュールできます。以下のマニフェストは、異なる AZ でポッドをスケジュールすることを Kubernetes スケジューラに指示します。 AZs

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: web-server
  labels:
    app: web-server
spec:
  replicas: 4
  selector:
    matchLabels:
      app: web-server
  template:
    metadata:
      labels:
        app: web-server
    spec:
      affinity:
        podAntiAffinity:
          preferredDuringSchedulingIgnoredDuringExecution:
          - podAffinityTerm:
              labelSelector:
                matchExpressions:
                - key: app
                  operator: In
                  values:
                  - web-server
              topologyKey: failure-domain.beta.kubernetes.io/zone
            weight: 100
      containers:
      - name: web-app
        image: nginx
```

> **Warning**
>
> 異なる AZs 間でポッドをスケジュールする必要はありません。そうしないと、デプロイ内のポッドの数が AZs の数を超えることはありません。

EBS ボリュームを使用するときに、各 AZ でノードを起動する機能を確保する

Amazon EBS を使用して永続ボリュームを提供する場合は、ポッドと関連する EBS ボリュームが同じ AZ にあることを確認する必要があります。ポッドは、別の AZ にある EBS-backed 永続ボリュームにアクセスできません。Kubernetes スケジューラは、ノード上のラベルからワーカーノードが配置されている AZ を認識し、常にボリュームと同じ AZ に EBS ボリュームを必要とするポッドをスケジュールします。ただし、ボリュームが配置されている AZ にワーカーノードがない場合は、Pod をスケジュールできません。

EKS Auto Mode または Karpenter を使用する場合は、NodeClass が各 AZ のサブネットを選択することを確認する必要があります。マネージド型ノードグループを使用する場合は、各 AZ にノードグループがあることを確認する必要があります。

EBS ストレージ機能は EKS Auto Mode に組み込まれていますが、Karpenter または Managed Node Groups を使用している場合は、EBS CSI もインストールする必要があります。

EKS Auto Mode を使用してワーカーノードを管理する

EKS Auto Mode は、運用上のオーバーヘッドを最小限に抑えながら、本番環境対応のクラスターを提供することで、EKS 管理を合理化します。Auto Mode は、クラスターで実行されている Pod に応じて、ノードの数をスケールアップまたはスケールダウンします。ノードはソフトウェアパッチと修正によって自動的に最新の状態に保たれ、更新は設定された NodePool 中断設定と Pod Disruption Budgets に従って実行されます。

ノードモニタリングエージェントを実行する

Node Monitoring Agent は、Kubernetes イベントを公開し、ノードのステータス条件を更新することで、ノードのヘルス問題をモニタリングして対応します。Node Monitoring Agent は EKS Auto Mode ノードに含まれており、Auto Mode で管理されていないノードの EKS アドオンとしてインストールできます。

EKS Auto Mode、マネージドノードグループ、Karpenter はすべて、Node Monitoring Agent によって報告された致命的なノード状態を検出し、それらの状態が発生したときにそれらのノードを自動的に修復できます。

QoS の実装

重要なアプリケーションの場合は、Pod のコンテナに `requests=limits` を定義することを検討してください。これにより、別の Pod がリソースをリクエストした場合にコンテナが強制終了されることはありません。

すべてのコンテナに CPU とメモリの制限を実装することがベストプラクティスです。これは、コンテナがシステムリソースを誤って消費して、他のコロケーションプロセスの可用性に影響を与えるのを防ぐためです。

すべてのワークロードのリソースリクエスト/制限を設定およびサイズ設定する

ワークロードのリソースリクエストと制限のサイズ設定には、いくつかの一般的なガイダンスを適用できます。

- CPU のリソース制限を指定しないでください。制限がない場合、リクエストは相対的な CPU 時間コンテナが取得する量の重みとして機能します。これにより、ワークロードは人為的な制限や飢餓なしに CPU 全体を使用できます。
- CPU 以外のリソースの場合、 `requests= limits`を設定すると最も予測可能な動作が得られます。`requests!=` の場合`limits`、コンテナの QOS も保証済みからバースト可能に減少するため、ノードの圧力が発生した場合に削除される可能性が高くなります。
- CPU 以外のリソースの場合、リクエストよりもはるかに大きい制限を指定しないでください。より大きいほど、 に対して設定`limits`され`requests`、ノードがオーバーコミットされる可能性が高くなり、ワークロードが中断される可能性が高くなります。
- Karpenter や Cluster AutoScaler などのノードの自動スケーリングソリューションを使用する場合、リクエストのサイズを正しく設定することは特に重要です。これらのツールは、ワークロードリクエストを調べて、プロビジョニングするノードの数とサイズを決定します。リクエストが小さすぎて制限が大きい場合、ノードに密集している場合、ワークロードが削除されたり、OOM が強制終了されたりすることがあります。

リソースリクエストの判断は難しい場合がありますが、Vertical Pod Autoscaler などのツールは、実行時にコンテナリソースの使用状況を観察することで、リクエストの「適切なサイズ」に役立ちます。リクエストサイズの決定に役立つその他のツールは次のとおりです。

- ゴールドロック
- パーカ
- Prodfiler
- rsg

名前空間のリソースクォータを設定する

名前空間は、複数のチームまたはプロジェクト間に分散された多くのユーザーがいる環境での使用を対象としています。これらは名前の範囲を提供し、複数のチーム、プロジェクト、ワークロード間でクラスターリソースを分割する方法です。名前空間のリソース消費の集計を制限できます。ResourceQuota オブジェクトは、名前空間で作成できるオブジェクトの数量をタイプ別に制限できます。また、そのプロジェクトのリソースによって消費される可能性のあるコンピューティングリソースの合計量も制限できます。特定の名前空間でリクエストできるストレージやコンピューティング (CPU およびメモリ) リソースの合計を制限できます。

CPU やメモリなどのコンピューティングリソースの名前空間でリソースクォータが有効になっている場合、ユーザーはその名前空間内のコンテナごとにリクエストまたは制限を指定する必要があります。

名前空間ごとにクォータを設定することを検討してください。を使用してLimitRanges、名前空間内のコンテナに事前設定された制限を自動的に適用することを検討してください。

名前空間内のコンテナリソースの使用を制限する

Resource Quotas は、名前空間で使用できるリソースの量を制限するのに役立ちます。LimitRange オブジェクトは、コンテナがリクエストできる最小リソースと最大リソースを実装するのに役立ちます。LimitRange を使用すると、コンテナのデフォルトのリクエストと制限を設定できます。これは、コンピューティングリソースの制限を設定することが組織内の標準プラクティスではない場合に役立ちます。名前が示すように、 LimitRangeは名前空間の Pod またはコンテナあたりのコンピューティングリソース使用量の最小値と最大値を適用できます。また、名前空間で PersistentVolumeClaim ごとに最小ストレージリクエストと最大ストレージリクエストを適用します。

を LimitRangeと組み合わせて使用ResourceQuotaして、コンテナレベルと名前空間レベルで制限を適用することを検討してください。これらの制限を設定すると、コンテナまたは名前空間がクラスター内の他のテナントが使用するリソースに影響を与えなくなります。

NodeLocal DNSCache を使用する

NodeLocal DNSCache を実行することで、クラスター DNS のパフォーマンスを向上させることができます。この機能は、クラスターノードで DNS キャッシュエージェントを DaemonSet として実行します。すべてのポッドは、kube-dnsサービスを使用する代わりに、ノードで実行されている DNS キャッシュエージェントを使用して名前を解決します。この機能は EKS Auto Mode に自動的に含まれます。

自動スケーリング CoreDNS を設定する

クラスター DNS のパフォーマンスを向上させるもう 1 つの方法は、CoreDNS ポッドの組み込み自動スケーリングを有効にすることです。

この機能は、ノード数や CPU コア数など、クラスターの状態を継続的にモニタリングします。この情報に基づいて、コントローラーは EKS クラスター内の CoreDNS デプロイのレプリカ数を動的に調整します。

ネットワーキングのベストプラクティス

クラスターとアプリケーションを効率的に運用するには、Kubernetes ネットワークを理解することが重要です。ポッドネットワークはクラスターネットワークとも呼ばれ、Kubernetes ネットワークの中心です。Kubernetes は、クラスターネットワーキング用の Container Network Interface (CNI) プラグインをサポートしています。

Amazon EKS は、Kubernetes Pod ネットワーキングを実装するために、Amazon Virtual Private Cloud (VPC) CNI プラグインを正式にサポートしています。VPC CNI は AWS VPC とのネイティブ統合を提供し、アンダーレイモードで動作します。アンダーレイモードでは、ポッドとホストは同じネットワークレイヤーに配置され、ネットワーク名前空間を共有します。Pod の IP アドレスは、クラスターと VPC の観点から一貫しています。

このガイドでは、Kubernetes クラスターネットワーキングのコンテキストで Amazon VPC Container Network Interface (VPC CNI) を紹介します。VPC CNI は EKS でサポートされているデフォルトのネットワークプラグインであるため、ガイドの焦点です。VPC CNI は、さまざまなユースケースをサポートするように高度に設定できます。このガイドには、さまざまな VPC CNI ユースケース、操作モード、サブコンポーネント、推奨事項に関する専用セクションも含まれています。

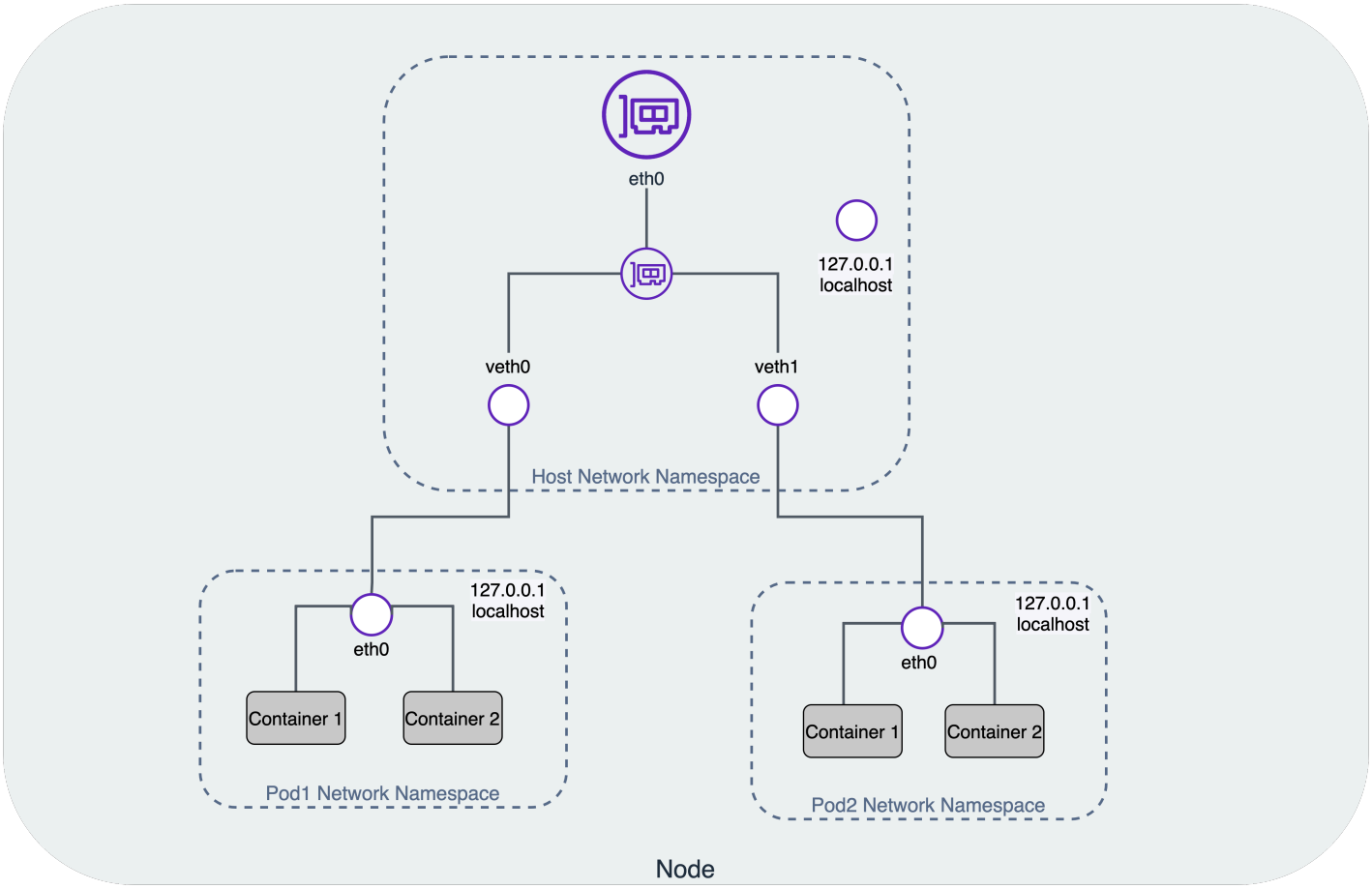
Amazon EKS はアップストリーム Kubernetes を実行し、Kubernetes 準拠の認定を受けています。代替 CNI プラグインを使用できますが、このガイドでは代替 CNIs を管理するための推奨事項は提供していません。EKS Alternate CNI ドキュメントで、代替 CNI CNIs を効果的に管理するためのパートナーとリソースのリストを確認してください。

Kubernetes ネットワーキングモデル

Kubernetes は、クラスターネットワークに次の要件を設定します。

- 同じノードでスケジュールされたポッドは、NAT (ネットワークアドレス変換) を使用せずに他のポッドと通信できる必要があります。
- 特定のノードで実行されているすべてのシステムデーモン (kubelet などのバックグラウンドプロセス) は、同じノードで実行されている Pod と通信できます。
- ホストネットワークを使用するポッドは、NAT を使用せずに、他のすべてのノードの他のすべてのポッドに接続できる必要があります。

Kubernetes が互換性のあるネットワーク実装に期待することの詳細については、Kubernetes ネットワークモデルを参照してください。次の図は、Pod ネットワーク名前空間とホストネットワーク名前空間の関係を示しています。



コンテナネットワークインターフェイス (CNI)

Kubernetes は、Kubernetes ネットワークモデルを実装するための CNI 仕様とプラグインをサポートしています。CNI は、仕様 (現行バージョン 1.0.0) と、コンテナ内のネットワークインターフェイスを設定するためのプラグインを記述するためのライブラリと、サポートされている複数のプラグインで構成されます。CNI は、コンテナのネットワーク接続と、コンテナが削除されたときに割り当てられたリソースの削除にのみ関係します。

CNI プラグインは、kubelet に`--network-plugin=cni`コマンドラインオプションを渡すことで有効になります。Kubelet は `--cni-conf-dir` (デフォルトは /etc/cni/net.d) からファイルを読み取り、そのファイルからの CNI 設定を使用して各 Pod のネットワークを設定します。CNI 設定ファイルは CNI 仕様 (最小 v0.4.0) と一致する必要があり、設定で参照される必要な CNI プラグインは `--cni-bin-dir` ディレクトリに存在する必要があります (デフォルトは /opt/cni/bin)。ディレクトリに

複数の CNI 設定ファイルがある場合、kubelet は名前が最初に来る設定ファイルを辞書順に使用します。

Amazon Virtual Private Cloud (VPC) CNI

AWS が提供する VPC CNI は、EKS クラスターのデフォルトのネットワークアドオンです。VPC CNI アドオンは、EKS クラスターをプロビジョニングするときにデフォルトでインストールされます。VPC CNI は Kubernetes ワーカーノードで実行されます。VPC CNI アドオンは、CNI バイナリと IP アドレス管理 (ipamd) プラグインで構成されます。CNI は VPC ネットワークから Pod に IP アドレスを割り当てます。ipamd は、各 Kubernetes ノードへの AWS Elastic Networking Interface (ENIs) を管理し、IPsのウォームプールを維持します。VPC CNI には、ポッドの起動時間を短縮するための ENIsと IP アドレスを事前に割り当てるための設定オプションが用意されています。推奨されるプラグイン管理のベストプラクティスについては、「Amazon VPC CNI」を参照してください。

Amazon EKS では、クラスターの作成時に少なくとも 2 つのアベイラビリティーゾーンにサブネットを指定することをお勧めします。Amazon VPC CNI は、ノードサブネットから Pod に IP アドレスを割り当てます。サブネットで使用可能な IP アドレスを確認することを強くお勧めします。EKS クラスターをデプロイする前に、VPC とサブネットの推奨事項を検討してください。

Amazon VPC CNI は、ノードのプライマリ ENIs にアタッチされたサブネットから ENI とセカンダリ IP アドレスのウォームプールを割り当てます。この VPC CNI モードはセカンダリ IP モードと呼ばれます。IP アドレスの数、つまりポッドの数 (Pod 密度) は、インスタンスタイプで定義されている ENIs の数と ENI あたりの IP アドレス (制限) によって定義されます。セカンダリモードはデフォルトであり、インスタンスタイプが小さい小さなクラスターに適しています。ポッド密度の問題が発生した場合は、プレフィックスモードの使用を検討してください。ENIs にプレフィックスを割り当てることで、ポッドのノードで使用可能な IP アドレスを増やすこともできます。

Amazon VPC CNI は AWS VPC とネイティブに統合され、ユーザーは既存の AWS VPC ネットワークとセキュリティのベストプラクティスを適用して Kubernetes クラスターを構築できます。これには、VPC フローログ、VPC ルーティングポリシー、セキュリティグループを使用してネットワークトラフィックを分離する機能が含まれます。デフォルトでは、Amazon VPC CNI はノード上のプライマリ ENI に関連付けられたセキュリティグループを Pod に適用します。Pod に異なるネットワークルールを割り当てる場合は、Pod のセキュリティグループを有効にすることを検討してください。

デフォルトでは、VPC CNI はノードのプライマリ ENI に割り当てられたサブネットから Pod に IP アドレスを割り当てます。数千のワークロードで大規模なクラスターを実行すると、IPv4 アドレスが不足することがよくあります。AWS VPC では、IPs4 CIDRs ブロックの枯渇を回避する

セカンダリ CIDR を割り当てることで、使用可能な IP を拡張できます。 IPv4 AWS VPC CNI では、ポッドに別のサブネット CIDR 範囲を使用できます。VPC CNI のこの機能では、カスタムネットワーキングと呼ばれます。カスタムネットワークを使用して EKS で `100.64.0.0/10`および `198.19.0.0/16` CIDRs (CG-NAT) を使用することを検討してください。これにより、Pod が VPC から RFC1918 IP アドレスを消費しなくなる環境を効果的に作成できます。

カスタムネットワークは IPv4 アドレスの枯渇問題に対処するための 1 つのオプションですが、運用上のオーバーヘッドが必要です。この問題を解決するには、カスタムネットワークではなく IPv6 クラスターを使用することをお勧めします。具体的には、VPC で使用可能なすべての IPv6 クラスターに移行することをお勧めします。 IPv4 IPv6 をサポートする組織の計画を評価し、IPv6 への投資により長期的な価値があるかどうかを検討します。

EKS の IPv6 のサポートは、制限された IPv4 アドレス空間に起因する IP 枯渇の問題の解決に焦点を当てています。IPv4 の枯渇に関するお客様の問題に応じて、EKS はデュアルスタックポッドよりも IPv6-onlyポッドを優先しています。つまり、ポッドは IPv4 リソースにアクセスできますが、VPC CIDR 範囲からの IPv4 アドレスは割り当てられません。VPC CNI は、AWS マネージド VPC IPv6 CIDR ブロックから IPv6 アドレスを Pod に割り当てます。

サブネット計算ツール

このプロジェクトには、サブネット計算ツール Excel ドキュメントが含まれています。この計算ドキュメントは、 `WARM_IP_TARGET`や など、さまざまな ENI 設定オプションで指定されたワークロードの IP アドレス消費をシミュレートします`WARM_ENI_TARGET`。このドキュメントには、2 つのシートが含まれています。1 つはウォーム ENI モード用、もう 1 つはウォーム IP モード用です。これらのモードの詳細については、VPC CNI ガイダンスを参照してください。

入力:

- サブネット CIDR サイズ
- ウォーム ENI ターゲットまたはウォーム IP ターゲット
- インスタンスのリスト
  - インスタンスごとにスケジュールされたワークロードポッドのタイプ、数、数

出力:

- ホストされているポッドの合計数
- 消費IPs の数

- 残りのサブネット IPsの数
- インスタンスレベルの詳細
  - インスタンスあたりのウォーム IPs/ENIsの数
  - インスタンスあたりのアクティブな IPs/ENIsの数

VPC とサブネットに関する考慮事項

EKS クラスターを操作するには、Kubernetes ネットワークに加えて、AWS VPC ネットワークに関する知識が必要です。

VPC の設計または既存の VPC へのクラスターのデプロイを開始する前に、EKS コントロールプレーンVPCs通信メカニズムを理解しておくことをお勧めします。

EKS で使用する VPC とサブネットを設計する場合は、クラスター VPC に関する考慮事項と Amazon EKS セキュリティグループの考慮事項を参照してください。 https://docs.aws.amazon.com/eks/latest/userguide/sec-group-reqs.html

概要:

EKS クラスターアーキテクチャ

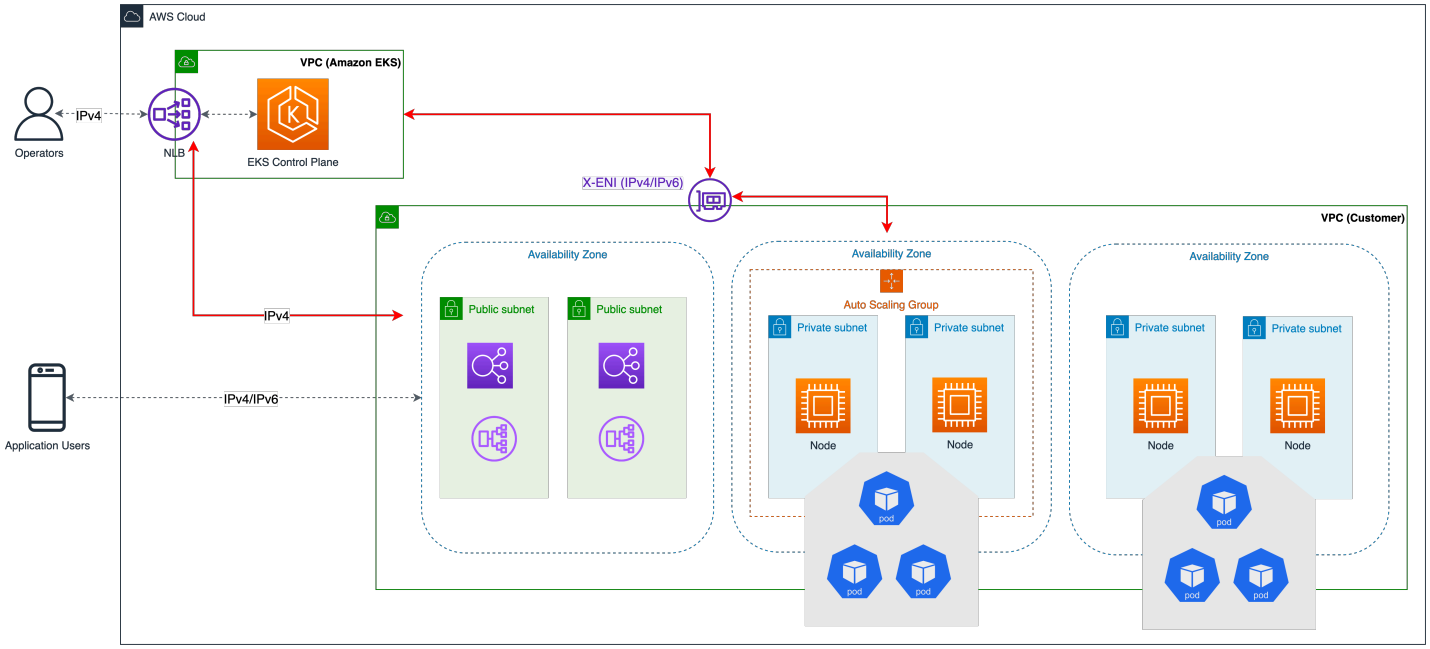
EKS クラスターは 2 つの VPCsで構成されます。

- Kubernetes コントロールプレーンをホストする AWS マネージド VPC。この VPC は顧客アカウントには表示されません。
- Kubernetes ノードをホストするカスタマーマネージド VPC。これは、コンテナが実行される場所であり、クラスターで使用されるロードバランサーなどの他のカスタマー管理の AWS インフラストラクチャも同様です。この VPC は顧客アカウントに表示されます。クラスターを作成する前に、カスタマーマネージド VPC を作成する必要があります。eksctl は、指定しない場合に VPC を作成します。

カスタマー VPC のノードには、AWS VPC のマネージド API サーバーエンドポイントに接続する機能が必要です。これにより、ノードは Kubernetes コントロールプレーンに登録され、アプリケーションポッドを実行するリクエストを受信できます。

ノードは、(a) EKS パブリックエンドポイント、または (b) EKS によって管理されるクロスアカウント Elastic Network Interface (X-ENI) を介して EKS コントロールプレーンに接続します。クラスター

を作成するときは、少なくとも 2 つの VPC サブネットを指定する必要があります。EKS は、クラスターの作成時に指定された各サブネット (クラスターサブネットとも呼ばれます) に X-ENI を配置します。Kubernetes API サーバーは、これらのクロスアカウント ENIs を使用して、カスタマーマネージドクラスター VPC サブネットにデプロイされたノードと通信します。



ノードが起動すると、EKS ブートストラップスクリプトが実行され、Kubernetes ノード設定ファイルがインストールされます。各インスタンスの起動プロセスの一環として、コンテナランタイムエージェント、kubelet、Kubernetes ノードエージェントを起動します。

ノードを登録するため、Kubelet は Kubernetes クラスターエンドポイントに連絡します。VPC 外のパブリックエンドポイントまたは VPC 内のプライベートエンドポイントとの接続を確立します。Kubelet は API 指示を受け取り、ステータスの更新とハートビートを定期的にエンドポイントに提供します。

EKS コントロールプレーン通信

EKS には、クラスターエンドポイントへのアクセスを制御する 2 つの方法があります。エンドポイントアクセスコントロールを使用すると、パブリックインターネットからエンドポイントにアクセスできるか、VPC 経由でのみエンドポイントにアクセスできるかを選択できます。パブリックエンドポイント (デフォルト) 、プライベートエンドポイント、またはその両方を一度にオンにできます。

クラスター API エンドポイントの設定によって、ノードがコントロールプレーンと通信するためのパスが決まります。これらのエンドポイント設定は、EKS コンソールまたは API を使用していつでも変更できます。

パブリックエンドポイント

これは新しい Amazon EKS クラスターのデフォルトの動作です。クラスターのパブリックエンドポイントのみが有効になっている場合、クラスターの VPC 内 (ワーカーノードからコントロールプレーンへの通信など) から送信される Kubernetes API リクエストは VPC を離れますが、Amazon のネットワークは離れません。ノードをコントロールプレーンに接続するには、パブリック IP アドレスとインターネットゲートウェイへのルート、または NAT ゲートウェイのパブリック IP アドレスを使用できる NAT ゲートウェイへのルートが必要です。

パブリックエンドポイントとプライベートエンドポイント

パブリックエンドポイントとプライベートエンドポイントの両方が有効になっている場合、VPC 内からの Kubernetes API リクエストは、VPC 内の X-ENIs を介してコントロールプレーンと通信します。クラスター API サーバーにはインターネットからアクセスできます。

プライベートエンドポイント

プライベートエンドポイントのみが有効になっている場合、インターネットから API サーバーへのパブリックアクセスはありません。クラスター API サーバーへのすべてのトラフィックはクラスターの VPC または接続されたネットワーク内から送信する必要があります。ノードは、VPC 内の X-ENIs を介して API サーバーと通信します。クラスター管理ツールにはプライベートエンドポイントへのアクセス権が必要です。Amazon VPC の外部からプライベート Amazon EKS クラスターエンドポイントに接続する方法について説明します。

クラスターの API サーバーエンドポイントは、パブリック DNS サーバーによって VPC のプライベート IP アドレスに解決されることに注意してください。これまではエンドポイントは VPC 内からしか解決できませんでした。

VPC 設定

Amazon VPC では、IPv4 と IPv6 アドレス設定がサポートされています。Amazon EKS はデフォルトで IPv4 をサポートしています。VPC には、それに関連付けられた IPv4 CIDR ブロックがある必要があります。オプションで、複数の IPv4 クラスレスドメイン間ルーティング (CIDR) ブロックと複数の IPv6 CIDR ブロックを VPC に関連付けることができます。VPC を作成するときは、RFC 1918 で指定されているように、プライベート IPv4 アドレス範囲から VPC の IPv4 CIDR ブロックを指定する必要があります。 http://www.faqs.org/rfcs/rfc1918.html許可されるブロックサイズは、/16プレフィックス (65,536 IP アドレス) と/28プレフィックス (16 IP アドレス) の間です。

新しい VPC を作成するときは、単一の IPv6 CIDR ブロックをアタッチし、既存の VPC を変更するときは最大 5 つまでアタッチできます。IPv6 CIDR ブロックサイズのプレフィックスの長さは /44

~ /60 で、IPv6 サブネットの場合は /44/ ~ /64 です。Amazon が管理する IPv6 アドレスのプールから IPv6 CIDR ブロックをリクエストできます。詳細については、「VPC ユーザーガイド」の「VPC CIDR ブロック」セクションを参照してください。

Amazon EKS クラスターは、IPv4 と IPv6 の両方をサポートします。デフォルトでは、EKS クラスターは IPv4 IP を使用します。クラスターの作成時に IPv6 を指定すると、IPv6 クラスターの使用が有効になります。IPv6 クラスターには、デュアルスタック VPCs とサブネットが必要です。

Amazon EKS では、クラスターの作成時に異なるアベイラビリティーゾーンにある少なくとも 2 つのサブネットを使用することをお勧めします。クラスターの作成時に渡すサブネットは、クラスターサブネットと呼ばれます。クラスターを作成すると、Amazon EKS は指定したサブネットに最大 4 つのクロスアカウント (x アカウントまたは x ENIs) ENIs を作成します。x-ENIs は常にデプロイされ、ログ配信、exec、プロキシなどのクラスター管理トラフィックに使用されます。VPC とサブネットの要件の詳細については、「EKS ユーザーガイド」を参照してください。

Kubernetes ワーカーノードはクラスターサブネットで実行できますが、推奨されません。クラスターのアップグレード中、Amazon EKS はクラスターサブネットに追加の ENIs をプロビジョニングします。クラスターがスケールアウトすると、ワーカーノードとポッドがクラスターサブネットで使用可能な IPs を消費する可能性があります。したがって、/28 ネットマスクで専用クラスターサブネットを使用することを検討 IPs することをお勧めします。

Kubernetes ワーカーノードは、パブリックサブネットまたはプライベートサブネットで実行できます。サブネットがパブリックかプライベートかとは、サブネット内のトラフィックがインターネットゲートウェイ経由でルーティングされるかどうかを指します。パブリックサブネットには、インターネットゲートウェイを介してインターネットへのルートテーブルエントリがありますが、プライベートサブネットにはアクセスできません。

別の場所から発信され、ノードに到達するトラフィックはイングレスと呼ばれます。ノードから発信され、ネットワークから出るトラフィックは、エグレスと呼ばれます。インターネットゲートウェイで設定されたサブネット内のパブリック IP アドレスまたは Elastic IP アドレス (EIPs) を持つノードは、VPC の外部からの進入を許可します。プライベートサブネットには通常、NAT ゲートウェイへのルーティングがあります。これにより、VPC の外部からサブネット内のノードへの進入トラフィックは許可されませんが、ノードからのトラフィックが VPC (進入) から出ることは許可されます。

IPv6 の世界では、すべてのアドレスがインターネットでルーティング可能です。ノードとポッドに関連付けられた IPv6 アドレスはパブリックです。プライベートサブネットは、VPC に Egress-Only インターネットゲートウェイ (EIGW) を実装することでサポートされ、すべての着信トラフィックを

ブロックしながらアウトバウンドトラフィックを許可します。IPv6 サブネットを実装するためのベストプラクティスについては、VPC ユーザーガイドを参照してください。

VPC とサブネットは 3 つの異なる方法で設定できます。

パブリックサブネットのみを使用する

同じパブリックサブネットに、ノードとイングレスリソース (ロードバランサーなど) の両方が作成されます。パブリックサブネットに をタグ付けkubernetes.io/role/elbして、インターネットに面するロードバランサーを構築します。この設定では、クラスターエンドポイントをパブリック、プライベート、またはその両方 (パブリックとプライベート) に設定できます。

プライベートサブネットとパブリックサブネットの使用

ノードはプライベートサブネットに作成されますが、イングレスリソースはパブリックサブネットにインスタンス化されます。クラスターエンドポイントへのパブリックアクセス、プライベートアクセス、またはその両方 (パブリックとプライベート) アクセスを有効にできます。クラスターエンドポイントの設定に応じて、ノードトラフィックは NAT ゲートウェイまたは ENI を介して入ります。

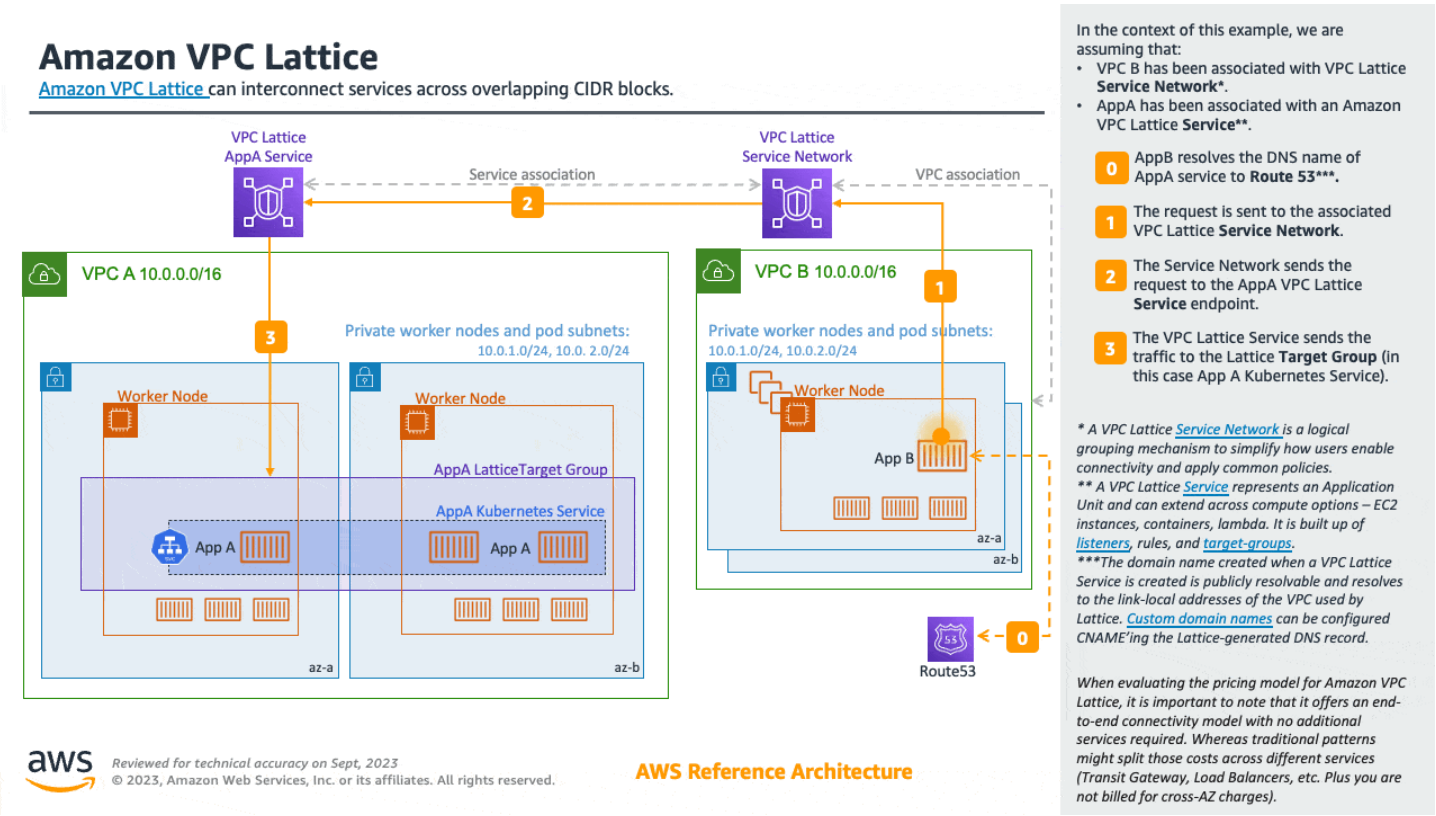
プライベートサブネットのみを使用する

ノードと進入の両方がプライベートサブネットに作成されます。kubernetes.io/role/internal-elb サブネットタグを使用して内部ロードバランサーを作成します。クラスターのエンドポイントにアクセスするには、VPN 接続が必要です。EC2 およびすべての Amazon ECR および S3 リポジトリに対して AWS PrivateLink を有効にする必要があります。クラスターのプライベートエンドポイントのみを有効にする必要があります。プライベートクラスターをプロビジョニングする前に、EKS プライベートクラスターの要件を確認することをお勧めします。

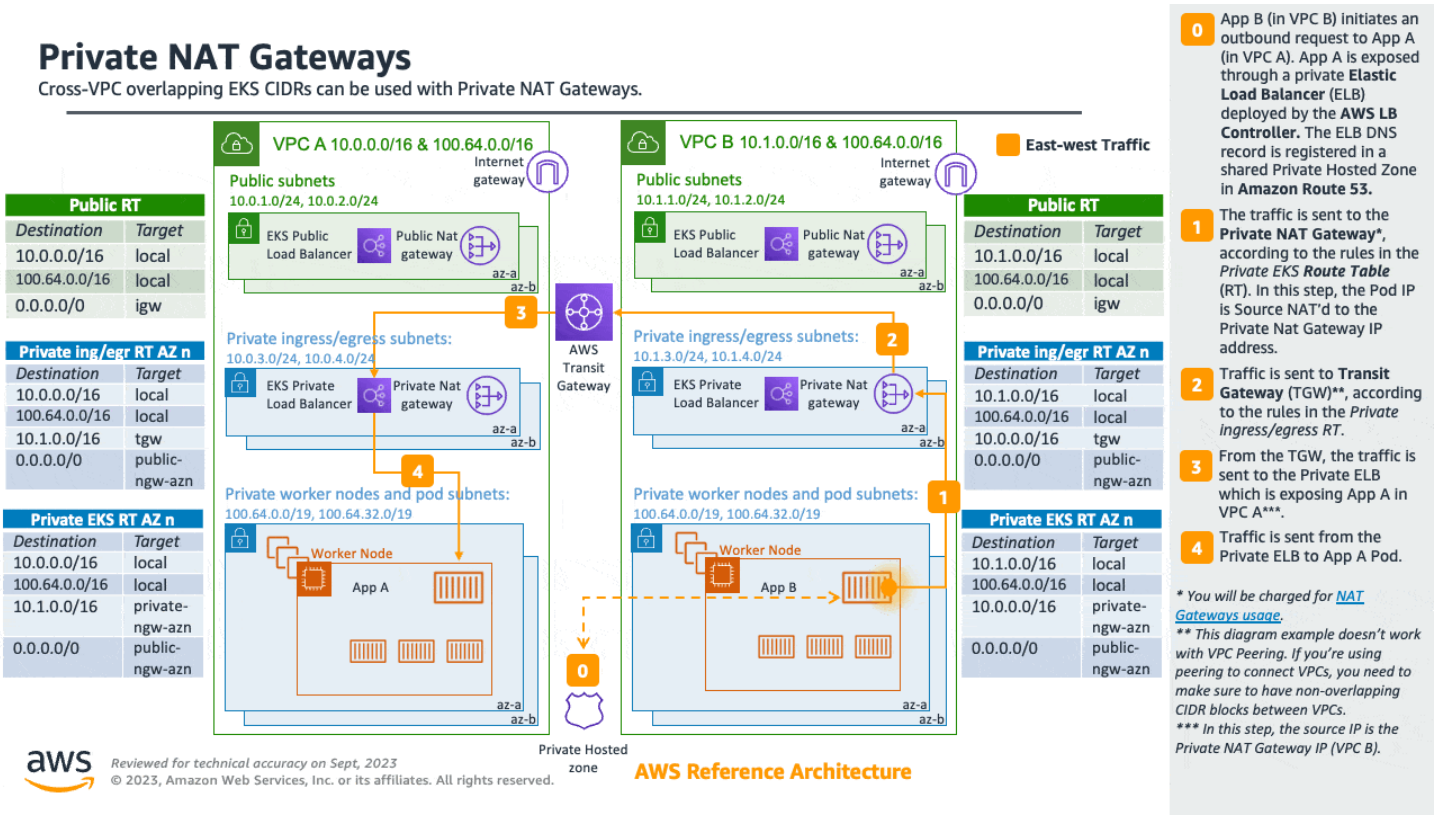
VPCs間の通信

複数の VPCs と個別の EKS クラスターをこれらの VPCs にデプロイする必要がある場合は、多くのシナリオがあります。

Amazon VPC Lattice を使用すると、複数の VPCs とアカウントにサービスを一貫して安全に接続できます (VPC ピアリング、AWS PrivateLink、AWS Transit Gateway などのサービスによって追加の接続が提供される必要はありません) 。詳細については、こちらを参照してください。



Amazon VPC Lattice は IPv4 と IPv6 のリンクローカルアドレス空間で動作し、IPv4 アドレスが重複している可能性のあるサービス間の接続を提供します。運用効率を高めるために、EKS クラスターとノードを重複しない IP 範囲にデプロイすることを強くお勧めします。インフラストラクチャに重複する IP 範囲を持つ VPCs が含まれている場合は、それに応じてネットワークを設計する必要があります。プライベート NAT ゲートウェイ、またはカスタムネットワーキングモードの VPC CNI をトランジットゲートウェイと組み合わせて EKS 上のワークロードを統合し、ルーティング可能な RFC1918 IP アドレスを維持しながら、重複する CIDR の課題を解決することをお勧めします。

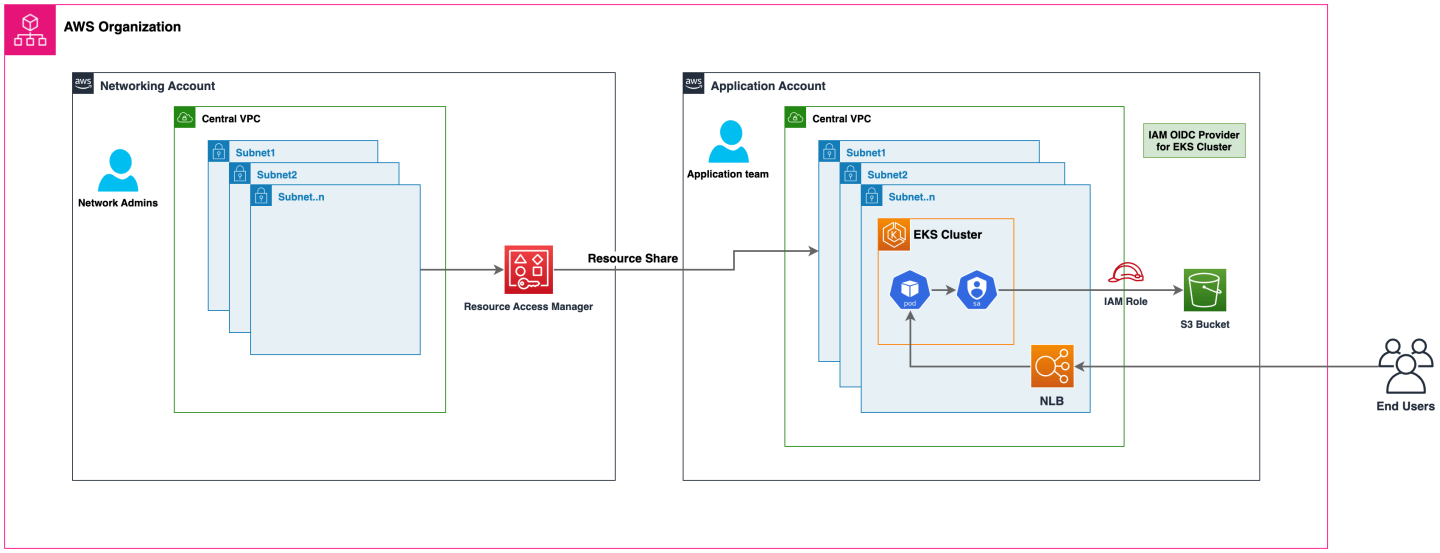


お客様がサービスプロバイダーであり、Kubernetes サービスと進入 (ALB または NLB) を個別のアカウントの顧客 VPC と共有する場合は、エンドポイントサービスとも呼ばれる AWS PrivateLink の使用を検討してください。

複数のアカウント間で VPC を共有する

多くの企業は、ネットワーク管理を合理化し、コストを削減し、AWS Organization 内の複数の AWS アカウントでセキュリティを向上させる手段として、共有 Amazon VPCs を採用しました。AWS Resource Access Manager (RAM) を使用して、サポートされている AWS リソースを個々の AWS アカウント、組織単位 (OUs)、または AWS Organization 全体と安全に共有します。

AWS RAM を使用して、別の AWS アカウントの共有 VPC サブネットに Amazon EKS クラスター、マネージド型ノードグループ、その他のサポート対象の AWS リソース (LoadBalancers、セキュリティグループ、エンドポイントなど) をデプロイできます。以下の図は、高レベルアーキテクチャの例を示しています。これにより、中央ネットワークチームは VPCs、サブネットなどのネットワーク構造を制御し、アプリケーションまたはプラットフォームチームはそれぞれの AWS アカウントに Amazon EKS クラスターをデプロイできます。このシナリオの完全なチュートリアルは、この github リポジトリで入手できます。



共有サブネットを使用する場合の考慮事項

- Amazon EKS クラスターとワーカーノードは、すべて同じ VPC の一部である共有サブネット内に作成できます。Amazon EKS は、複数の VPCsクラスターの作成をサポートしていません。
- Amazon EKS は AWS VPC セキュリティグループ (SGs) を使用して、Kubernetes コントロールプレーンとクラスターのワーカーノード間のトラフィックを制御します。セキュリティグループは、ワーカーノード、他の VPC リソース、および外部 IP アドレス間のトラフィックを制御するためにも使用されます。これらのセキュリティグループは、アプリケーション/参加者アカウントで作成する必要があります。ポッドに使用するセキュリティグループも参加者アカウントにあることを確認します。セキュリティグループ内のインバウンドルールとアウトバウンドルールを設定して、中央 VPC アカウントにあるセキュリティグループとの間で必要なトラフィックを許可できます。
- Amazon EKS クラスターが存在する参加者アカウント内に IAM ロールと関連するポリシーを作成します。これらの IAM ロールとポリシーは、Amazon EKS によって管理される Kubernetes クラスター、および Fargate で実行されているノードとポッドに必要なアクセス許可を付与するために不可欠です。アクセス許可により、Amazon EKS はユーザーに代わって他の AWS サービスを呼び出すことができます。
- 次のアプローチに従って、k8s ポッドから Amazon S3 バケット、Dynamodb テーブルなどの AWS リソースへのクロスアカウントアクセスを許可できます。
  - リソースベースのポリシーアプローチ: AWS サービスがリソースポリシーをサポートしている場合は、適切なリソースベースのポリシーを追加して、kubernetes ポッドに割り当てられた IAM ロールへのクロスアカウントアクセスを許可できます。このシナリオでは、OIDC プロバイダー、IAM ロール、およびアクセス許可ポリシーがアプリケーションアカウントに存在しま

す。リソースベースのポリシーをサポートする AWS サービスを検索するには、IAM と連携する AWS のサービスを参照し、リソースベースの列で「はい」があるサービスを探します。

  - OIDC プロバイダーアプローチ: OIDC プロバイダー、IAM ロール、アクセス許可、信頼ポリシーなどの IAM リソースは、リソースが存在する他の参加者の AWS アカウントに作成されます。これらのロールは、クロスアカウントリソースにアクセスできるように、アプリケーションアカウントの Kubernetes ポッドに割り当てられます。このアプローチの完全なウォークスルーについては、Kubernetes サービスアカウントのクロスアカウント IAM ロールのブログを参照してください。

- Amazon Elastic Loadbalancer (ELB) リソース (ALB または NLB) をデプロイして、アプリケーションまたは中央ネットワークアカウントの k8s ポッドにトラフィックをルーティングできます。中央ネットワークアカウントに ELB リソースをデプロイする詳細な手順については、「クロスアカウントLoad Balancerによる Amazon EKS ポッドの公開」のチュートリアルを参照してください。このオプションは、ロードLoad Balancerリソースのセキュリティ設定を完全に制御できるため、柔軟性が向上します。
- `custom networking feature` Amazon VPC CNI を使用する場合は、中央ネットワークアカウントに記載されているアベイラビリティーゾーン (AZ) ID マッピングを使用して、各 を作成する必要がありますENIConfig。これは、物理 AZs を各 AWS アカウントの AZ 名にランダムにマッピングするためです。

セキュリティグループ

セキュリティグループは、関連付けられたリソースに到達できるトラフィックおよびリソースから発信できるトラフィックを制御します。Amazon EKS はセキュリティグループを使用して、コントロールプレーンとノード間の通信を管理します。クラスターを作成すると、Amazon EKS により `eks-cluster-sg-my-cluster-uniqueID` という名前のセキュリティグループが作成されます。EKS は、これらのセキュリティグループをマネージド ENIsとノードに関連付けます。デフォルトのルールでは、すべてのトラフィックがクラスターとノード間で自由に行き来することができます。また、任意の送信先へのすべてのアウトバウンドトラフィックが許可されています。

クラスターを作成するときに、独自のセキュリティグループを指定できます。独自のセキュリティグループを指定する場合は、セキュリティグループの推奨事項を参照してください。

推奨事項

マルチ AZ 配置を検討する

AWS リージョンは、複数の物理的に分離および分離されたアベイラビリティーゾーン (AZ) を提供します。AZ は、低レイテンシー、高スループット、および高度に冗長なネットワークで接続されます。アベイラビリティーゾーンを使用すると、中断することなくアベイラビリティーゾーン間で自動的にフェイルオーバーするアプリケーションを設計および運用できます。Amazon EKS では、複数のアベイラビリティーゾーンに EKS クラスターをデプロイすることを強くお勧めします。クラスターを作成するときは、少なくとも 2 つのアベイラビリティーゾーンにサブネットを指定することを検討してください。

ノードで実行されている Kubelet は、 などのノードオブジェクトにラベルを自動的に追加しますtopology.kubernetes.io/region=us-west-2。ノードラベルを Pod トポロジーの分散制約と組み合わせて使用して、ゾーン間で Pod がどのように分散されるかを制御することをお勧めします。これらのヒントにより、Kubernetes スケジューラはポッドを配置して可用性を期待し、相関障害がワークロード全体に影響を与えるリスクを軽減できます。ノードセレクタと AZ スプレッド制約の例については、「ポッドへのノードの割り当て」を参照してください。

ノードの作成時にサブネットまたはアベイラビリティーゾーンを定義できます。サブネットが設定されていない場合、ノードはクラスターサブネットに配置されます。マネージド型ノードグループの EKS サポートは、利用可能な容量で複数のアベイラビリティーゾーンにノードを自動的に分散します。Karpenter は、ワークロードがトポロジーのスプレッド制限を定義する場合、ノードを指定された AZ にスケーリングすることで AZs スプレッドプレイスメントを尊重します。

AWS Elastic Load Balancer は、Kubernetes クラスターの AWS Load Balancer Controller によって管理されます。Kubernetes Ingress リソース用の Application Load Balancer (ALB) と Load Balancer タイプの Kubernetes サービス用の Network Load Balancer (NLB) をプロビジョニングします。Elastic Load Balancer コントローラーはタグを使用してサブネットを検出します。ELB コントローラーでは、進入リソースを正常にプロビジョニングするために、最低 2 つのアベイラビリティーゾーン (AZs) が必要です。地理的冗長性の安全性と信頼性を活用するには、少なくとも 2 つの AZs にサブネットを設定することを検討してください。

プライベートサブネットにノードをデプロイする

プライベートサブネットとパブリックサブネットの両方を含む VPC は、Kubernetes ワークロードを EKS にデプロイするのに最適な方法です。2 つの異なるアベイラビリティーゾーンに 2 つ以上のパブリックサブネットと 2 つのプライベートサブネットを設定することを検討してください。パブ

リックサブネットの関連ルートテーブルには、インターネットゲートウェイ へのルートが含まれています。ポッドは NAT ゲートウェイを介してインターネットとやり取りできます。プライベートサブネットは、IPv6 環境 (EIGW) の Egress-Only インターネットゲートウェイでサポートされています。

プライベートサブネット内のノードをインスタンス化することで、ノードへのトラフィックを最大限制御でき、ほとんどの Kubernetes アプリケーションに効果的です。イングレスリソース (ロードバランサーなど) はパブリックサブネットでインスタンス化され、プライベートサブネットで動作するポッドにトラフィックをルーティングします。

セキュリティとネットワークの厳密な分離が必要な場合は、プライベート専用モードを検討してください。この設定では、AWS リージョンの VPC 内の個別のアベイラビリティーゾーンに 3 つのプライベートサブネットがデプロイされます。サブネットにデプロイされたリソースは、インターネットにアクセスすることも、サブネット内のリソースにもアクセスすることもできません。Kubernetes アプリケーションが他の AWS のサービスにアクセスするには、PrivateLink インターフェイスやゲートウェイエンドポイントを設定する必要があります。AWS Load Balancer Controller を使用してトラフィックを Pod にリダイレクトするように内部Load Balancerを設定できます。コントローラーがロードバランサーをプロビジョニングするには、プライベートサブネットにタグ (`kubernetes.io/role/internal-elb: 1`) を付ける必要があります。ノードをクラスターに登録するには、クラスターエンドポイントをプライベートモードに設定する必要があります。要件と考慮事項の詳細については、プライベートクラスターガイドを参照してください。

クラスターエンドポイントのパブリックモードとプライベートモードを検討する

Amazon EKS には、パブリック専用、public-and-privateのクラスターエンドポイントモードが用意されています。デフォルトモードはパブリック専用ですが、クラスターエンドポイントをパブリックモードとプライベートモードで設定することをお勧めします。このオプションを使用すると、クラスターの VPC 内の Kubernetes API コール (node-to-control-plane通信など) がプライベート VPC エンドポイントとトラフィックを利用してクラスターの VPC 内に留まることができます。一方、クラスター API サーバーにはインターネットからアクセスできます。ただし、パブリックエンドポイントを使用できる CIDR ブロックを制限することを強くお勧めします。CIDR ブロックの制限など、パブリックエンドポイントとプライベートエンドポイントのアクセスを設定する方法について説明します。

セキュリティとネットワークの分離が必要な場合は、プライベート専用のエンドポイントをお勧めします。EKS ユーザーガイドに記載されているオプションのいずれかを使用して、API サーバーにプライベートに接続することをお勧めします。

セキュリティグループを慎重に設定する

Amazon EKS は、カスタムセキュリティグループの使用をサポートしています。カスタムセキュリティグループでは、ノードと Kubernetes コントロールプレーン間の通信を許可する必要があります。組織がオープンな通信を許可していない場合は、ポートの要件を確認し、ルールを手動で設定してください。

EKS は、クラスターの作成時に指定したカスタムセキュリティグループをマネージドインターフェイス (X-ENIs。ただし、ノードにすぐには関連付けられません。ノードグループを作成するときは、カスタムセキュリティグループを手動で関連付けることを強くお勧めします。securityGroupSelectorTerms を有効にして、ノードの自動スケーリング中にカスタムセキュリティグループの Karpenter ノードテンプレート検出を有効にすることを検討してください。

すべてのノード間通信トラフィックを許可するセキュリティグループを作成することを強くお勧めします。ブートストラッププロセス中、ノードはクラスターエンドポイントにアクセスするためにアウトバウンドインターネット接続を必要とします。オンプレミス接続やコンテナレジストリアクセスなどの外部アクセス要件を評価し、ルールを適切に設定します。本番環境に変更を加える前に、開発環境で接続を慎重に確認することを強くお勧めします。

各アベイラビリティーゾーンに NAT ゲートウェイをデプロイする

プライベートサブネット (IPv4 および IPv6) にノードをデプロイする場合は、各アベイラビリティーゾーン (AZ) に NAT ゲートウェイを作成して、ゾーンに依存しないアーキテクチャを確保し、AZ 間の支出を削減することを検討してください。AZ 内の各 NAT ゲートウェイは冗長性を持って実装されます。

Amazon VPC CNI

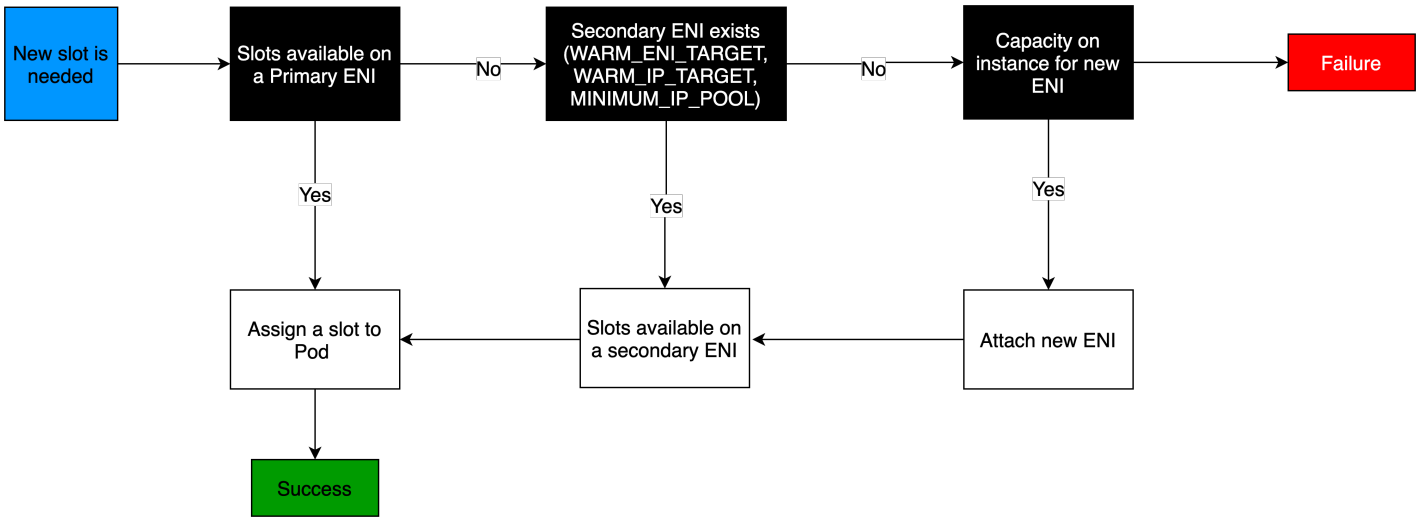
Amazon EKS は、VPC CNI とも呼ばれる Amazon VPC コンテナネットワークインターフェイスプラグインを介してクラスターネットワークを実装します。CNI プラグインを使用すると、Kubernetes Pod は VPC ネットワークと同じ IP アドレスを持つことができます。具体的には、Pod 内のすべてのコンテナはネットワーク名前空間を共有し、ローカルポートを使用して相互に通信できます。

Amazon VPC CNI には 2 つのコンポーネントがあります。

- CNI バイナリ。Pod 間の通信を有効にするように Pod-to-Pod ネットワークを設定します。CNI バイナリはノードルートファイルシステムで実行され、新しい Pod がノードに追加されるか、ノードから削除されたときに kubelet によって呼び出されます。
- ipamd は、長時間実行されるノードローカル IP アドレス管理 (IPAM) デーモンであり、以下を担当します。
  - ノードでの ENIsの管理
  - 使用可能な IP アドレスまたはプレフィックスのウォームプールの維持

インスタンスが作成されると、EC2 はプライマリサブネットに関連付けられたプライマリ ENI を作成してアタッチします。プライマリサブネットはパブリックサブネットでもプライベートサブネットでもかまいません。hostNetwork モードで実行される Pod は、ノードのプライマリ ENI に割り当てられたプライマリ IP アドレスを使用し、ホストと同じネットワーク名前空間を共有します。

CNI プラグインは、ノード上の Elastic Network Interface (ENI) を管理します。ノードがプロビジョニングされると、CNI プラグインはノードのサブネットからプライマリ ENI にスロットのプール (IPs またはプレフィックス) を自動的に割り当てます。このプールはウォームプールと呼ばれ、そのサイズはノードのインスタンスタイプによって決まります。CNI 設定によっては、スロットが IP アドレスまたはプレフィックスである場合があります。ENI のスロットが割り当てられると、CNI はスロットのウォームプールを持つ追加の ENIs をノードにアタッチできます。これらの追加の ENIs はセカンダリ ENIsと呼ばれます。各 ENI は、インスタンスタイプに基づいて、特定の数のスロットのみをサポートできます。CNI は、必要なスロット数に基づいてより多くの ENIs をインスタンスにアタッチします。これは通常、Pod の数に対応します。このプロセスは、ノードが追加の ENI をサポートできなくなるまで続行されます。CNI は、ポッドの起動を高速化するために、「ウォームENIs とスロットも事前に割り当てます。各インスタンスタイプには、アタッチできる ENIs の最大数があることに注意してください。これは、コンピューティングリソースに加えて、ポッド密度 (ノードあたりのポッド数) に関する 1 つの制約です。



ネットワークインターフェイスの最大数と使用できるスロットの最大数は、EC2 インスタンスのタイプによって異なります。各 Pod はスロットの IP アドレスを消費するため、特定の EC2 インスタンスで実行できる Pod の数は、アタッチできる ENIs の数と、各 ENI がサポートするスロットの数によって異なります。インスタンスの CPU とメモリリソースが枯渇しないように、EKS ユーザーガイドあたりの最大ポッド数を設定することをお勧めします。を使用するポッドhostNetworkは、この計算から除外されます。max-pod-calculator.sh というスクリプトを使用して、特定のインスタンスタイプの EKS の推奨最大ポッド数を計算することを検討できます。

概要:

セカンダリ IP モードは、VPC CNI のデフォルトモードです。このガイドでは、セカンダリ IP モードが有効になっている場合の VPC CNI の動作の一般的な概要を説明します。ipamd (IP アドレスの割り当て) の機能は、、、 the section called “ポッドあたりのセキュリティグループ”などの VPC CNI the section called “Linux のプレフィックスモード”の設定によって異なりますthe section called “カスタムネットワーキング”。

Amazon VPC CNI は、ワーカーノードに aws-node という名前の Kubernetes デーモンセットとしてデプロイされます。ワーカーノードがプロビジョニングされると、プライマリ ENI と呼ばれるデフォルトの ENI がアタッチされます。CNI は、ノードのプライマリ ENIs にアタッチされたサブネットから ENI とセカンダリ IP アドレスのウォームプールを割り当てます。デフォルトでは、ipamd はノードに追加の ENI の割り当てを試みます。IPAMD は、単一のポッドがスケジュールされ、プライマリ ENI からセカンダリ IP アドレスが割り当てられると、追加の ENI を割り当てます。この「ウォーム」 ENI により、Pod ネットワーキングが高速化されます。セカンダリ IP アドレスのプールがなくなると、CNI は別の ENI を追加してさらに割り当てます。

プール内の ENIs と IP アドレスの数
は、WARM_ENI_TARGET、WARM_IP_TARGET、MINIMUM_IP_TARGET と呼ばれる環境変数を介して設定されます。aws-node デーモンセットは、十分な数の ENIs がアタッチされていることを定期的に確認します。、、 WARM_IP_TARGET のすべてのMINIMUM_IP_TARGET条件が満たされるとWARM_ENI_TARGET、十分な数の ENIs がアタッチされます。アタッチENIs が不十分な場合、CNI は EC2 に API コールを行い、MAX_ENI制限に達するまでさらにアタッチします。

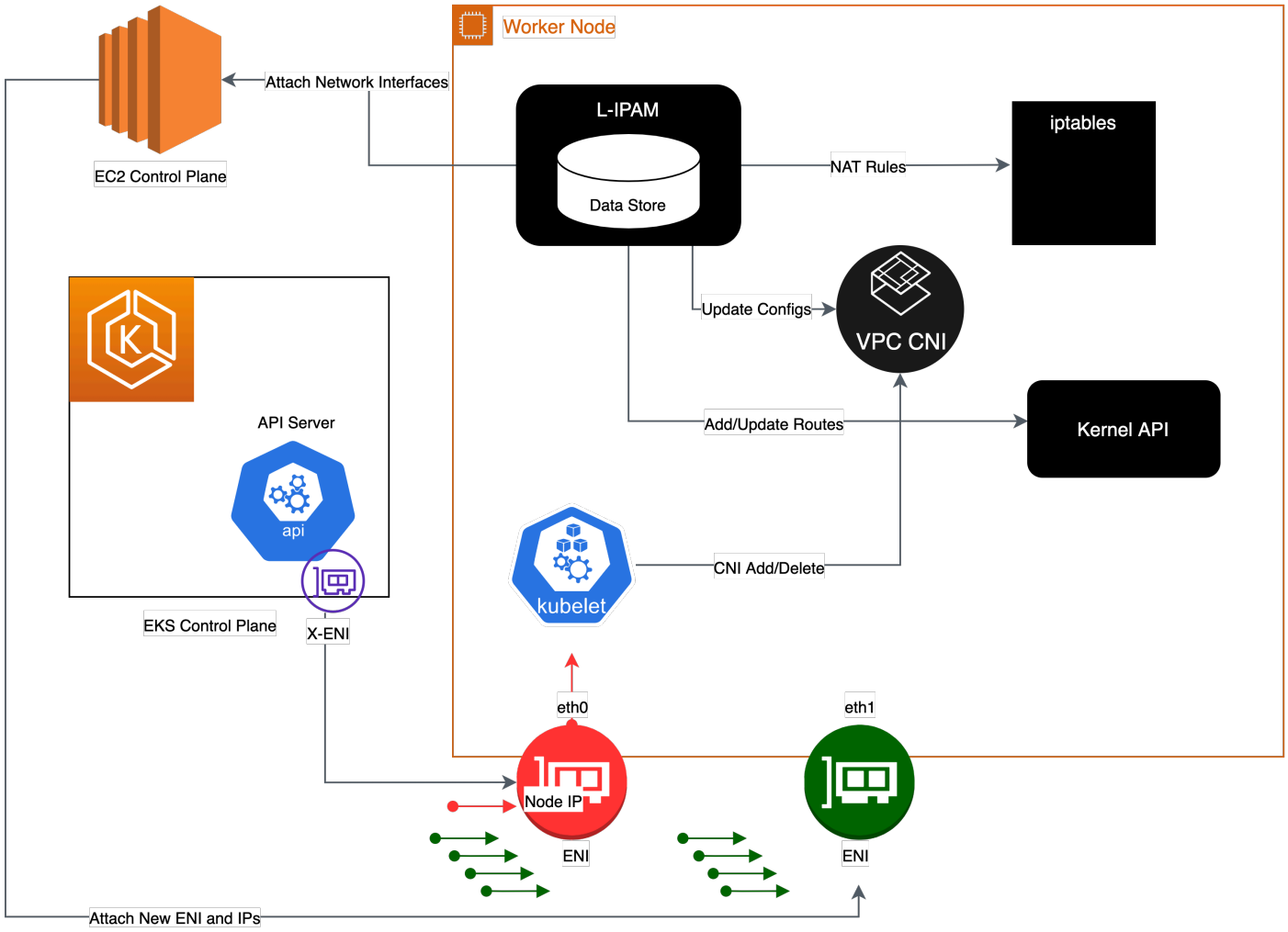
- WARM_ENI_TARGET - 整数、0 より大きい値は要件が有効であることを示します
  - 維持するウォーム ENIsの数。ENI は、セカンダリ ENI としてノードにアタッチされている場合、「ウォーム」ですが、どの Pod でも使用されません。具体的には、ENI の IP アドレスが Pod に関連付けられていません。
  - 例: 2 つの ENIs を持つインスタンスを考えてみましょう。各 ENI は 5 つの IP アドレスをサポートします。WARM_ENI_TARGET は 1 に設定されています。インスタンスに 5 つの IP アドレスが関連付けられている場合、CNI はインスタンスにアタッチされた 2 つの ENIs を維持します。最初の ENI が使用されており、この ENI の 5 つの可能な IP アドレスがすべて使用されます。2 番目の ENI は、プール内の 5 つの IP アドレスすべてを持つ「ウォーム」です。インスタンスで別の Pod を起動する場合は、6 番目の IP アドレスが必要です。CNI は、この 6 番目の Pod に、2 番目の ENI とプールの 5 つの IPs アドレスを割り当てます。2 番目の ENI は現在使用中であり、「ウォーム」ステータスではなくなりました。CNI は 3 番目の ENI を割り当てて、少なくとも 1 つのウォーム ENI を維持します。

> **Note**
>
> ウォーム ENIs VPC の CIDR から IP アドレスを消費します。IP アドレスは、ポッドなどのワークロードに関連付けられるまで、「未使用」または「ウォーム」です。

- WARM_IP_TARGET、整数、0 より大きい値は要件が有効であることを示します
  - 維持するウォーム IP アドレスの数。ウォーム IP はアクティブにアタッチされた ENI で使用できますが、ポッドに割り当てられていません。つまり、使用可能なウォーム IPs の数は、追加の ENI を必要とせずに Pod に割り当てることができる IPs の数です。
- 例: 1 つの ENI を持つインスタンスを考えてみましょう。各 ENI は 20 個の IP アドレスをサポートします。WARM_IP_TARGET は 5 に設定されています。WARM_ENI_TARGET は 0 に設定されています。16 番目の IP アドレスが必要になるまで、1 つの ENI のみがアタッチされます。次に、CNI はサブネット CIDR から 20 個のアドレスを消費する 2 番目の ENI をアタッチします。
- MINIMUM_IP_TARGET、整数、0 より大きい値は要件が有効であることを示します

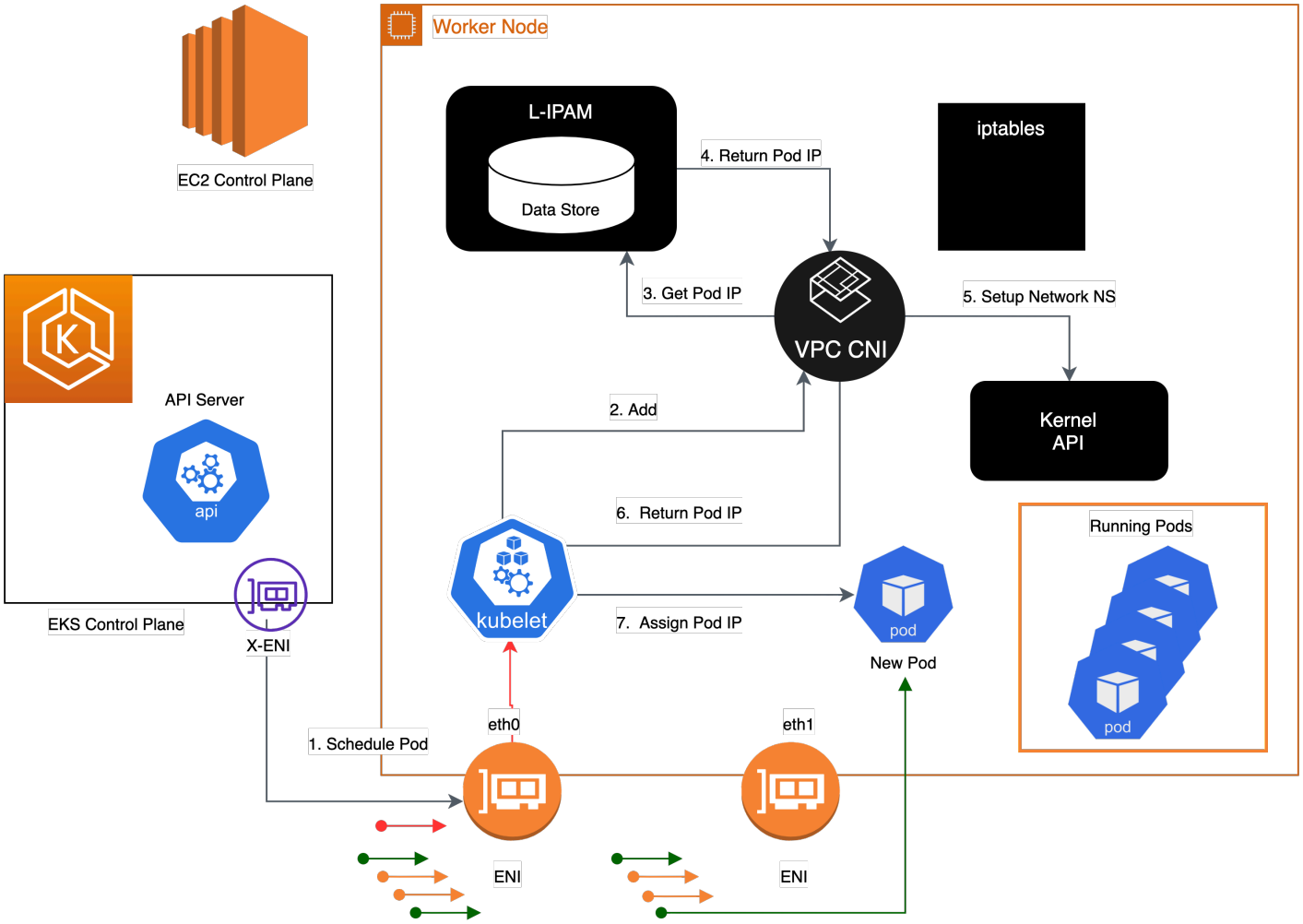
- いつでも割り当てられる IP アドレスの最小数。これは一般的に、インスタンスの起動時に複数の ENIs の割り当てをフロントロードするために使用されます。
- 例: 新しく起動したインスタンスを考えてみましょう。1 つの ENI があり、各 ENI は 10 個の IP アドレスをサポートします。MINIMUM_IP_TARGET は 100 に設定されています。ENI は直ちにさらに 9 つの ENIsし、合計 100 個のアドレスをアタッチします。これは、WARM_IP_TARGET 値または WARM_ENI_TARGET 値に関係なく発生します。

このプロジェクトには、サブネット計算ツール Excel ドキュメントが含まれています。この計算ドキュメントは、 WARM_IP_TARGETや など、さまざまな ENI 設定オプションで指定されたワークロードの IP アドレス消費をシミュレートしますWARM_ENI_TARGET。

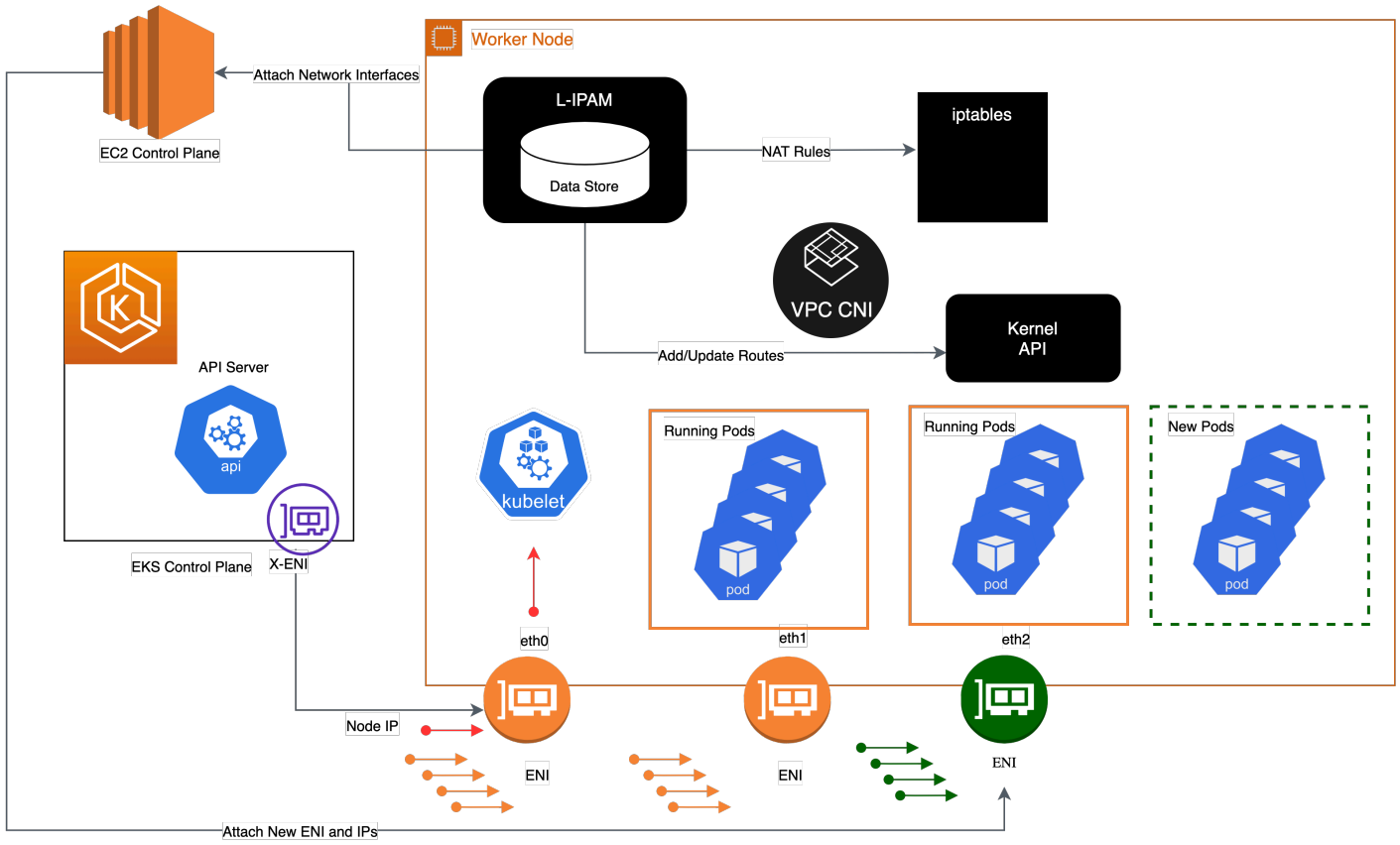


Kubelet が Pod の追加リクエストを受信すると、CNI バイナリは使用可能な IP アドレスを ipamd にクエリし、ipamd はそれを Pod に提供します。CNI バイナリはホストと Pod ネットワークをワイヤリングします。

ノードにデプロイされたポッドは、デフォルトでプライマリ ENI と同じセキュリティグループに割り当てられます。または、ポッドを異なるセキュリティグループで設定することもできます。



IP アドレスのプールが枯渇すると、プラグインは別の Elastic Network Interface をインスタンスに自動的にアタッチし、別のセカンダリ IP アドレスのセットをそのインターフェイスに割り当てます。このプロセスは、ノードが追加の Elastic Network Interface をサポートできなくなるまで続きます。



ポッドが削除されると、VPC CNI はポッドの IP アドレスを 30 秒のクールダウンキャッシュに配置します。クールダウンキャッシュIPs は新しい Pod に割り当てられません。クーリングオフ期間が終了すると、VPC CNI は Pod IP をウォームプールに戻します。クーリングオフ期間は、Pod IP アドレスが早期にリサイクルされるのを防ぎ、すべてのクラスターノードで kube-proxy が iptables ルールの更新を完了できるようにします。IPs または ENIs の数がウォームプール設定の数を超えると、ipamd プラグインIPs と ENIs を VPC に返します。

上記のセカンダリ IP モードで説明したように、各 Pod はインスタンスにアタッチされた ENIs の 1 つからセカンダリプライベート IP アドレスを 1 つ受け取ります。各 Pod は IP アドレスを使用するため、特定の EC2 インスタンスで実行できる Pod の数は、アタッチできる ENIsの数とサポートする IP アドレスの数によって異なります。VPC CNI は制限ファイルをチェックして、インスタンスのタイプごとに許可される ENIs と IP アドレスの数を確認します。

次の式を使用して、ノードにデプロイできる Pod の最大数を決定できます。

```
(Number of network interfaces for the instance type * (the number of IP addresses per
 network interface - 1)) + 2
```

+2 は、kube-proxy や VPC CNI などのホストネットワークを必要とするポッドを示します。Amazon EKS では、kube-proxy と VPC CNI が各ノードで動作する必要があり、これらの要件は max-pods 値に考慮されます。追加のホストネットワーキングポッドを実行する場合は、最大ポッド値を更新することを検討してください。起動テンプレートでユーザーデータ`--kubelet-extra-args "--max-pods=110"`として を指定できます。

例えば、3 つの c5.large ノード (ENIsつ、最大 10 IPs) を持つクラスターでは、クラスターが起動し、2 つの CoreDNS ポッドがある場合、CNI は 50 個の IP アドレスを消費し、43 個の IPsに保持します。ウォームプールを使用すると、アプリケーションのデプロイ時に Pod の起動を高速化できます。

ノード 1 (CoreDNS ポッドを使用): 2 ENIs、20 IPsが割り当てられました

ノード 2 (CoreDNS ポッドを使用): 2 ENIs、20 IPsが割り当てられました

ノード 3 (ポッドなし): 1 ENI。10 IPs が割り当てられました。

ノード 1 とノード 2 (同一設定):

- ENIs × 10 IPs = 合計 20 IPs
- プライマリ IPs (ENI ごとに 1 つ) = 18 IPs
- CoreDNS ポッドの IP を 1 つ減算 = 使用可能な IPs
- したがって、これらの各ノードにはウォームプールに 17 IPs があります。

ノード 3 の場合:

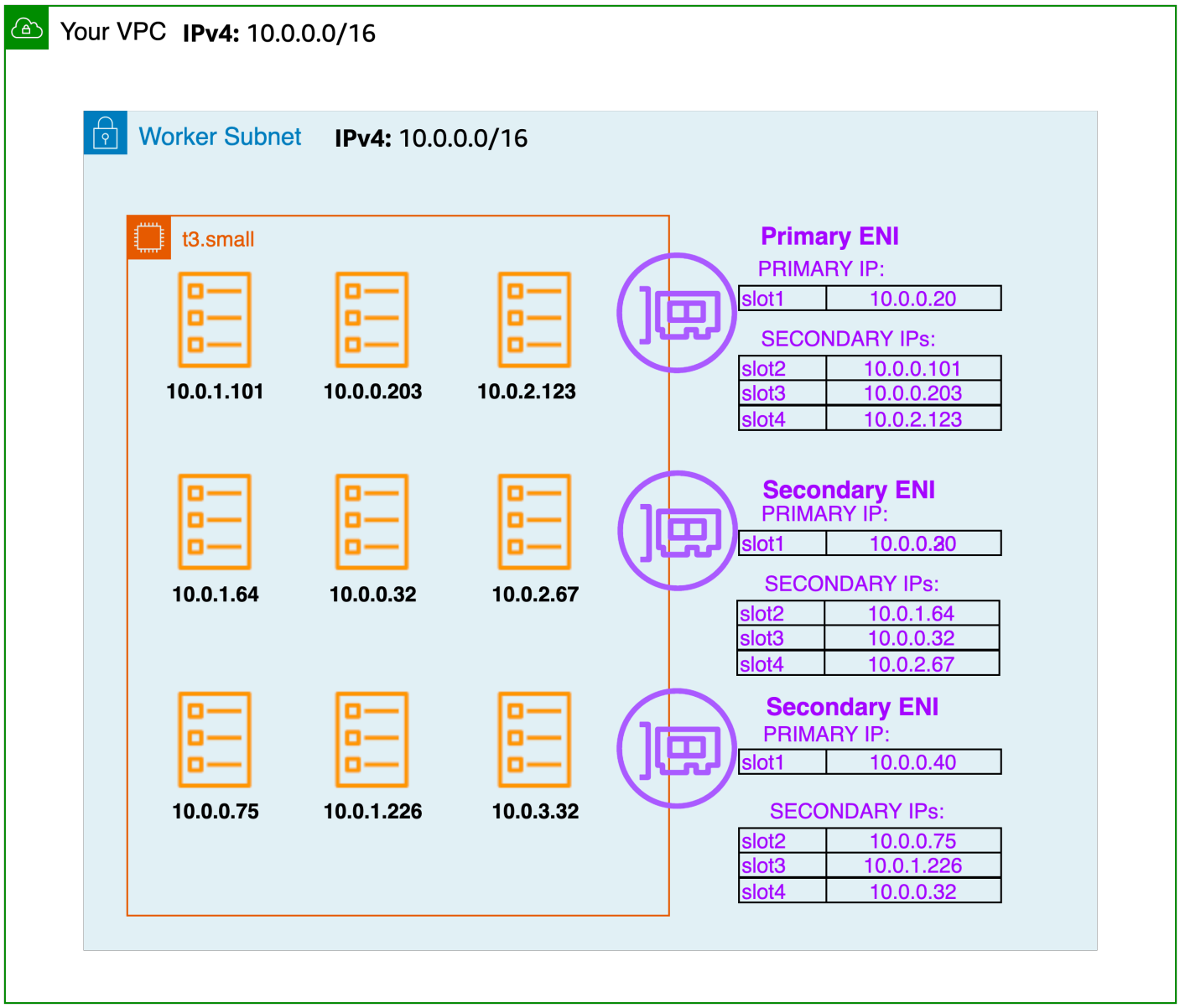
- 1 ENI × 10 IPs = 合計 10 IPs
- 1 つのプライマリ IP = ウォームプールで使用できる 9 IPs を減算する

ウォームプールの合計計算: - 17 (ノード 1) + 17 (ノード 2) + 9 (ノード 3) = 43 IPs

多くの場合デーモンセットとして実行されるインフラストラクチャポッドは、それぞれ最大ポッド数に影響することに注意してください。これらには以下が含まれます。

- CoreDNS
- Amazon Elastic LoadBalancer
- metrics-server の運用ポッド

これらのポッドの容量を組み合わせてインフラストラクチャを計画することをお勧めします。各インスタンスタイプでサポートされるポッドの最大数のリストについては、GitHub のeni-max-Pods.txt」を参照してください。



推奨事項

自動モードで EKS クラスターをデプロイする

EKS Auto Mode を使用してクラスターを作成すると、AWS はクラスターの VPC Container Network Interface (CNI) 設定を管理します。Amazon EKS 自動モード ではネットワーク形成のアドオンをイ

ンストールまたはアップグレードする必要はありません。ただし、ワークロードがマネージド VPC CNI 設定と互換性があることを確認してください。

VPC CNI マネージドアドオンをデプロイする

クラスターをプロビジョニングすると、Amazon EKS は VPC CNI を自動的にインストールします。ただし、Amazon EKS は、クラスターがコンピューティング、ストレージ、ネットワークなどの基盤となる AWS リソースとやり取りできるようにするマネージドアドオンをサポートしています。VPC CNI などのマネージドアドオンを使用してクラスターをデプロイすることを強くお勧めします。

Amazon EKS マネージドアドオンは、Amazon EKS クラスターの VPC CNI のインストールと管理を提供します。Amazon EKS アドオンには最新のセキュリティパッチ、バグ修正が含まれており、Amazon EKS と連携するために AWS によって検証されます。VPC CNI アドオンを使用すると、Amazon EKS クラスターのセキュリティと安定性を継続的に確保し、アドオンのインストール、設定、更新に必要な労力を削減できます。さらに、マネージドアドオンは、Amazon EKS API、AWS マネジメントコンソール、AWS CLI、および eksctl を介して追加、更新、または削除できます。

`kubectl get` コマンドの `--show-managed-fields`フラグを使用して、VPC CNI のマネージドフィールドを見つけることができます。

```
kubectl get daemonset aws-node --show-managed-fields -n kube-system -o yaml
```

マネージドアドオンは、15 分ごとに設定を自動的に上書きすることで、設定のずれを防ぎます。つまり、アドオンの作成後に Kubernetes API を介して行われたマネージドアドオンへの変更は、自動ドリフト防止プロセスによって上書きされ、アドオンの更新プロセス中にデフォルトに設定されます。

EKS によって管理されるフィールドは、マネージャーを EKS とする managedFields の下に一覧表示されます。EKS によって管理されるフィールドには、サービスアカウント、イメージ、イメージ URL、ライブネスプローブ、準備状況プローブ、ラベル、ボリューム、ボリュームマウントが含まれます。

> **Note**
>
> WARM_ENI_TARGET、WARM_IP_TARGET、MINIMUM_IP_TARGET などの最も頻繁に使用されるフィールドは管理されず、調整されません。これらのフィールドへの変更は、アドオンの更新時に保持されます。

本番稼働用クラスターを更新する前に、非本番稼働用クラスターで特定の設定のアドオン動作をテストすることをお勧めします。さらに、アドオン設定については、EKS ユーザーガイドの手順に従ってください。

マネージドアドオンへの移行

バージョンの互換性を管理し、セルフマネージド VPC CNI のセキュリティパッチを更新します。セルフマネージド型アドオンを更新するには、EKS ユーザーガイドで説明されている Kubernetes APIs と手順を使用する必要があります。既存の EKS クラスターのマネージドアドオンに移行し、移行前に現在の CNI 設定のバックアップを作成することを強くお勧めします。マネージドアドオンを設定するには、Amazon EKS API、AWS マネジメントコンソール、または AWS コマンドラインインターフェイスを使用できます。

```
kubectl apply view-last-applied daemonset aws-node -n kube-system  aws-k8s-cni-old.yaml
```

フィールドがデフォルト設定で管理対象としてリストされている場合、Amazon EKS は CNI 設定を置き換えます。マネージドフィールドを変更しないように注意してください。アドオンは、ウォーム環境変数や CNI モードなどの設定フィールドを調整しません。マネージド CNI への移行中は、Pod とアプリケーションは引き続き実行されます。

更新前のバックアップ CNI 設定

VPC CNI はカスタマーデータプレーン (ノード) で実行されるため、Amazon EKS は、新しいバージョンがリリースされたとき、またはクラスターを新しい Kubernetes マイナーバージョンに更新した後に、アドオン (マネージド型およびセルフマネージド型) を自動的に更新しません。既存のクラスターのアドオンを更新するには、update-addon API を使用して更新をトリガーするか、アドオンの EKS コンソールで update now リンクをクリックする必要があります。セルフマネージドアドオンをデプロイした場合は、「セルフマネージド VPC CNI アドオンの更新」に記載されているステップに従います。

一度に 1 つのマイナーバージョンを更新することを強くお勧めします。例えば、現在のマイナーバージョン `1.9` を `1.11` に更新する場合、まず `1.10` での最新のパッチバージョンに更新した上で、次に `1.11` での最新のパッチバージョンに更新します。

Amazon VPC CNI を更新する前に、aws-node デーモンセットの検査を実行します。既存の設定のバックアップを取ります。マネージドアドオンを使用する場合は、Amazon EKS が上書きする可能性のある設定を更新していないことを確認します。自動化ワークフローの更新後フック、またはアドオンの更新後の手動適用ステップをお勧めします。

```
kubectl apply view-last-applied daemonset aws-node -n kube-system  aws-k8s-cni-old.yaml
```

セルフマネージド型アドオンの場合は、GitHub `releases`の とバックアップを比較して利用可能なバージョンを確認し、更新するバージョンの変更について理解します。Helm を使用してセルフマネージドアドオンを管理し、値ファイルを活用して設定を適用することをお勧めします。デーモンセットの削除を伴う更新操作は、アプリケーションのダウンタイムにつながるため、回避する必要があります。

セキュリティコンテキストを理解する

VPC CNI を効率的に管理するために設定されたセキュリティコンテキストを理解することを強くお勧めします。Amazon VPC CNI には、CNI バイナリと ipamd (aws-node) デーモンセットの 2 つのコンポーネントがあります。CNI はノードでバイナリとして実行され、ノードルートファイルシステムにアクセスできます。また、ノードレベルで iptables を処理するため、特権アクセスもあります。CNI バイナリは、ポッドが追加または削除されると kubelet によって呼び出されます。

aws-node デーモンセットは、ノードレベルでの IP アドレス管理を担当する長時間実行されるプロセスです。aws-node は `hostNetwork`モードで実行され、ループバックデバイスへのアクセスと、同じノード上の他のポッドのネットワークアクティビティを許可します。aws-node init-container は特権モードで実行され、CRI ソケットをマウントして、Daemonset がノードで実行されているポッドによる IP 使用率をモニタリングできるようにします。Amazon EKS は、aws-node init コンテナの特権要件を削除しようとしています。さらに、aws-node は NAT エントリを更新し、iptables モジュールをロードする必要があるため、NET_ADMIN 権限で実行されます。

Amazon EKS では、ポッドとネットワーク設定の IP 管理用に aws-node マニフェストで定義されているセキュリティポリシーをデプロイすることをお勧めします。VPC CNI の最新バージョンへの更新を検討してください。さらに、特定のセキュリティ要件がある場合は、GitHub の問題を開くことを検討してください。

CNI に個別の IAM ロールを使用する

AWS VPC CNI には、AWS Identity and Access Management (IAM) アクセス許可が必要です。IAM ロールを使用する前に、CNI ポリシーを設定する必要があります。IPv4 クラスターの AWS 管理ポリシー`AmazonEKS_CNI_Policy`である を使用できます。AmazonEKS CNI 管理ポリシーには、IPv4 クラスターに対するアクセス許可のみがあります。IPv6 クラスターには、ここにリストされているアクセス許可を持つ個別の IAM ポリシーを作成する必要があります。

デフォルトでは、VPC CNI は Amazon EKS ノード IAM ロール (マネージド型ノードグループとセルフマネージド型ノードグループの両方) を継承します。

Amazon VPC CNI の関連ポリシーを使用して別の IAM ロールを設定することを強くお勧めします。そうでない場合、Amazon VPC CNI のポッドはノード IAM ロールに割り当てられたアクセス許可を取得し、ノードに割り当てられたインスタンスプロファイルにアクセスできます。

VPC CNI プラグインは、aws-node というサービスアカウントを作成して設定します。デフォルトでは、サービスアカウントは Amazon EKS CNI ポリシーがアタッチされた Amazon EKS ノード IAM ロールにバインドされます。別の IAM ロールを使用するには、Amazon EKS CNI ポリシーがアタッチされた新しいサービスアカウントを作成することをお勧めします。新しいサービスアカウントを使用するには、CNI ポッドを再デプロイする必要があります。新しいクラスターを作成するときは、VPC CNI マネージドアドオン`--service-account-role-arn`の を指定することを検討してください。IPv4 と IPv6 の両方の Amazon EKS CNI ポリシーを Amazon EKS ノードロールから削除してください。

セキュリティ違反の爆発範囲を最小限に抑えるために、アクセスインスタンスメタデータをブロックすることをお勧めします。

Liveness/Readiness Probe の障害に対処する

EKS 1.20 以降のクラスターのライブネスと準備状況プローブのタイムアウト値 (デフォルトは `timeoutSeconds: 10`) を増やすことをお勧めします。これにより、アプリケーションの Pod が containerCreating 状態でスタックするのを防ぐことができます。この問題は、データ集約型クラスターとバッチ処理クラスターで発生しています。CPU 使用率が高いと、aws-node プローブのヘルス障害が発生し、Pod の CPU リクエストが満たされなくなります。プローブタイムアウトの変更に加えて、aws-node の CPU リソースリクエスト (デフォルト `CPU: 25m`) が正しく設定されていることを確認します。ノードに問題がある場合を除き、設定を更新することはお勧めしません。

Amazon EKS サポートを利用する間は、ノード`bash /opt/cni/bin/aws-cni-support.sh`で sudo を実行することを強くお勧めします。このスクリプトは、ノードの kubelet ログとメモリ使用率の評価に役立ちます。スクリプトを実行するには、Amazon EKS ワーカーノードに SSM エージェントをインストールすることを検討してください。

EKS 最適化されていない AMI インスタンスで IPTables 転送ポリシーを設定する

カスタム AMI を使用している場合は、kubelet.service で iptables 転送ポリシーを ACCEPT に設定してください。多くのシステムでは、iptables 転送ポリシーを DROP に設定します。HashiCorp Packer と、AWS GitHub の Amazon EKS AMI リポジトリのリソースと設定スクリプトを使用したビルド仕様を使用して、カスタム AMI を構築できます。kubelet.service を更新し、ここで指定した手順に従ってカスタム AMI を作成できます。

CNI バージョンを定期的にアップグレードする

VPC CNI には下位互換性があります。最新バージョンは、Amazon EKS がサポートするすべての Kubernetes バージョンで動作します。さらに、VPC CNI は EKS アドオンとして提供されます (上記の「VPC CNI マネージドアドオンのデプロイ」を参照) 。EKS アドオンはアドオンのアップグレードをオーケストレーションしますが、データプレーンで実行されるため、CNI などのアドオンは自動的にアップグレードされません。マネージド型およびセルフマネージド型のワーカーノードのアップグレード後に VPC CNI アドオンをアップグレードするのはお客様の責任です。

IP アドレス使用率の最適化

アプリケーションのモダナイゼーションにより、コンテナ化された環境は急速に拡大しています。つまり、ますます多くのワーカーノードとポッドがデプロイされています。

Amazon VPC CNI プラグインは、VPC の CIDR (複数可) から各ポッドに IP アドレスを割り当てます。このアプローチは、VPC フローログやその他のモニタリングソリューションなどのツールを使用して、Pod アドレスを完全に可視化します。ワークロードタイプによっては、大量の IP アドレスがポッドによって消費される可能性があります。

AWS ネットワークアーキテクチャを設計するときは、VPC およびノードレベルで Amazon EKS IP 消費量を最適化することが重要です。これにより、IP の枯渇の問題を軽減し、ノードあたりのポッド密度を向上させることができます。

このセクションでは、これらの目標の達成に役立つテクニックについて説明します。

ノードレベルの IP 消費を最適化する

プレフィックス委任は、Amazon Virtual Private Cloud (Amazon VPC) の機能であり、Amazon Elastic Compute Cloud (Amazon EC2) インスタンスに IPv4 または IPv6 プレフィックスを割り当てることができます。これにより、ネットワークインターフェイス (ENI) あたりの IP アドレスが増加し、ノードあたりのポッド密度が増加し、コンピューティング効率が向上します。プレフィックスの委任は、カスタムネットワーキングでもサポートされています。

詳細については、「Linux ノードでのプレフィックス委任」および「Windows ノードでのプレフィックス委任」セクションを参照してください。

IP の枯渇問題を軽減する

クラスターが使用可能なすべての IP アドレスを消費しないように、VPCs とサブネットのサイズを増加を考慮して設定することを強くお勧めします。

IPv6 を採用することは、これらの問題を最初から回避する優れた方法です。ただし、スケーラビリティのニーズが初期計画を超え、IPv6 を採用できない組織では、IP アドレスの枯渇に対する VPC 設計の改善が推奨されます。Amazon EKS のお客様で最も一般的に使用される手法は、ルーティング不可能なセカンダリ CIDRs を VPC に追加し、IP アドレスを Pod に割り当てるときにこの追加の IP スペースを使用するように VPC CNI を設定することです。これは一般的にカスタムネットワークと呼ばれます。

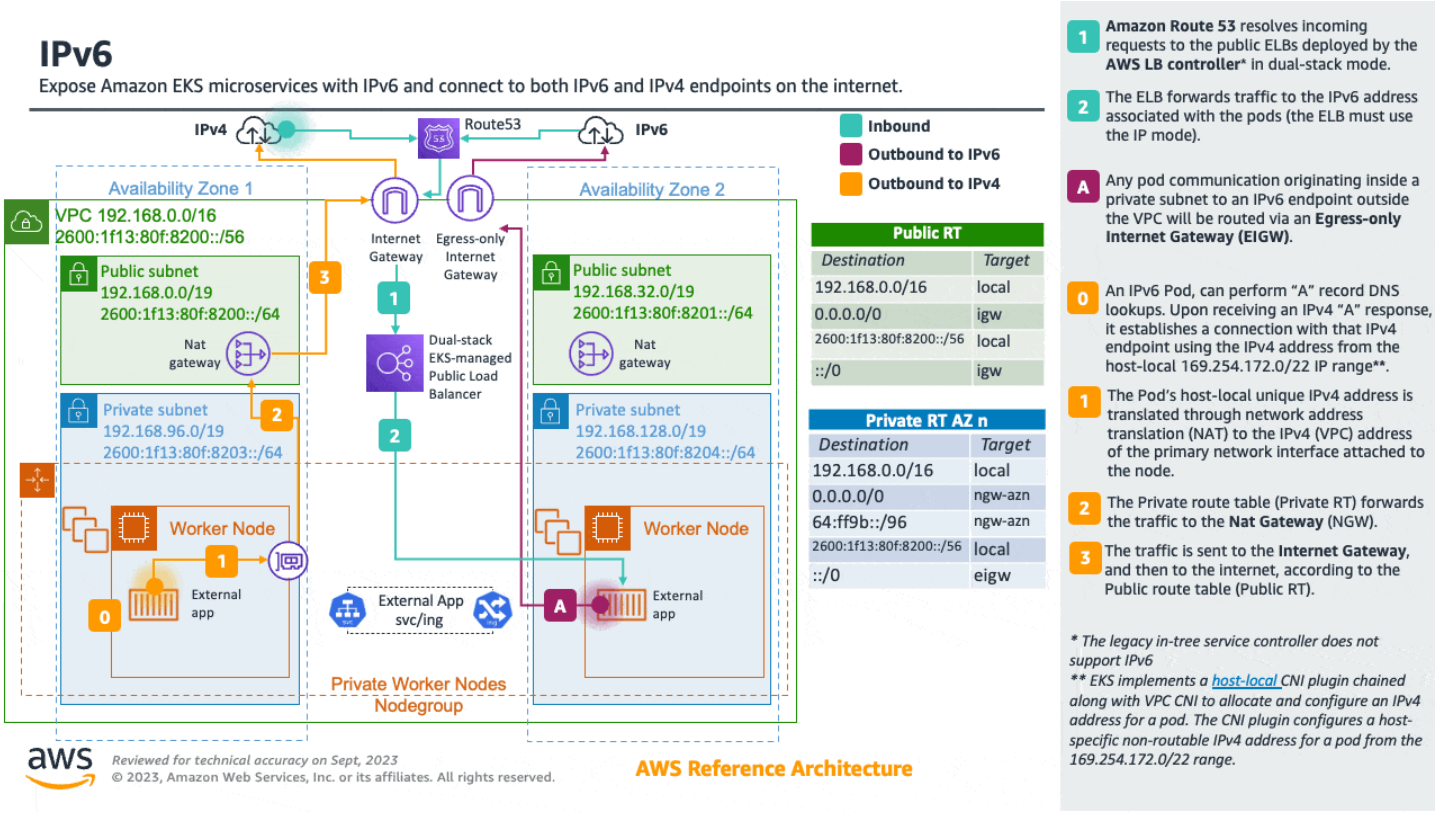
ノードに割り当てられた IPs のウォームプールを最適化するために使用できる Amazon VPC CNI の変数について説明します。このセクションは、Amazon EKS に組み込まれていないが、IP の枯渇を軽減するのに役立つ他のアーキテクチャパターンで終了します。

IPv6 を使用する (推奨)

IPv6 の採用はRFC1918 の制限を回避する最も簡単な方法です。ネットワークアーキテクチャを選択するときは、IPv6 を最初のオプションとして採用することを強くお勧めします。IPv6 は IP アドレス領域全体を大幅に拡大し、クラスター管理者は IPv4 の制限を回避するために労力を費やすことなく、アプリケーションの移行とスケーリングに集中できます。

Amazon EKS クラスターは、IPv4 と IPv6 の両方をサポートしています。デフォルトでは、EKS クラスターは IPv4 アドレス空間を使用します。クラスター作成時に IPv6 ベースのアドレス空間を指定すると、IPv6 を使用できます。IPv6 EKS クラスターでは、ポッドとサービスは IPv6 アドレスを受け取りますが、レガシー IPv4 エンドポイントが IPv6 クラスターで実行されているサービスに接続し、その逆も可能です。クラスター内のすべてのpod-to-pod通信は、常に IPv6 経由で行われます。VPC (/56) 内では、IPv6 サブネットの IPv6 CIDR ブロックサイズは /64 に固定されます。これにより、2^64 (約 18 50 億) の IPv6 アドレスが提供され、 は EKS でのデプロイをスケーリングできます。

詳細については、IPv6 EKS クラスターの実行」セクションを参照してください。実践的な経験については、「Get hands-on with IPv6 workshop」の「Understanding IPv6 on Amazon EKS」セクションを参照してください。 IPv6



IPv4 クラスターの IP 消費を最適化する

このセクションは、レガシーアプリケーションを実行しているお客様、および/または IPv6 に移行する準備ができていないお客様専用です。すべての組織ができるだけ早く IPv6 に移行することをお勧めしますが、IPv4 を使用してコンテナワークロードをスケールするための代替アプローチを検討する必要がある場合もあります。このため、Amazon EKS クラスターで IPv4 (RFC1918) アドレス空間の消費量を最適化するためのアーキテクチャパターンについても説明します。

成長計画

IP の枯渇に対する第 1 の防御策として、クラスターが使用可能なすべての IP アドレスを消費しないように、IPv4 VPCs とサブネットのサイズを増やすことを強くお勧めします。サブネットに十分な使用可能な IP アドレスがない場合、新しい Pod またはノードを作成することはできません。

VPC とサブネットを構築する前に、必要なワークロードスケールから逆算することをお勧めします。例えば、eksctl (EKS でクラスターを作成および管理するためのシンプルな CLI ツール) を使用してクラスターを構築する場合、/19 サブネットはデフォルトで作成されます。/19 のネットマスクは、8000 を超えるアドレスを割り当てることができるほとんどのワークロードタイプに適しています。

> **Important**
>
> VPCs とサブネットのサイズを設定する場合、ロードバランサー、RDS データベース、その他の VPC 内サービスなど、IP アドレスを消費できる要素 (ポッドとノードを除く) が多数存在する可能性があります。

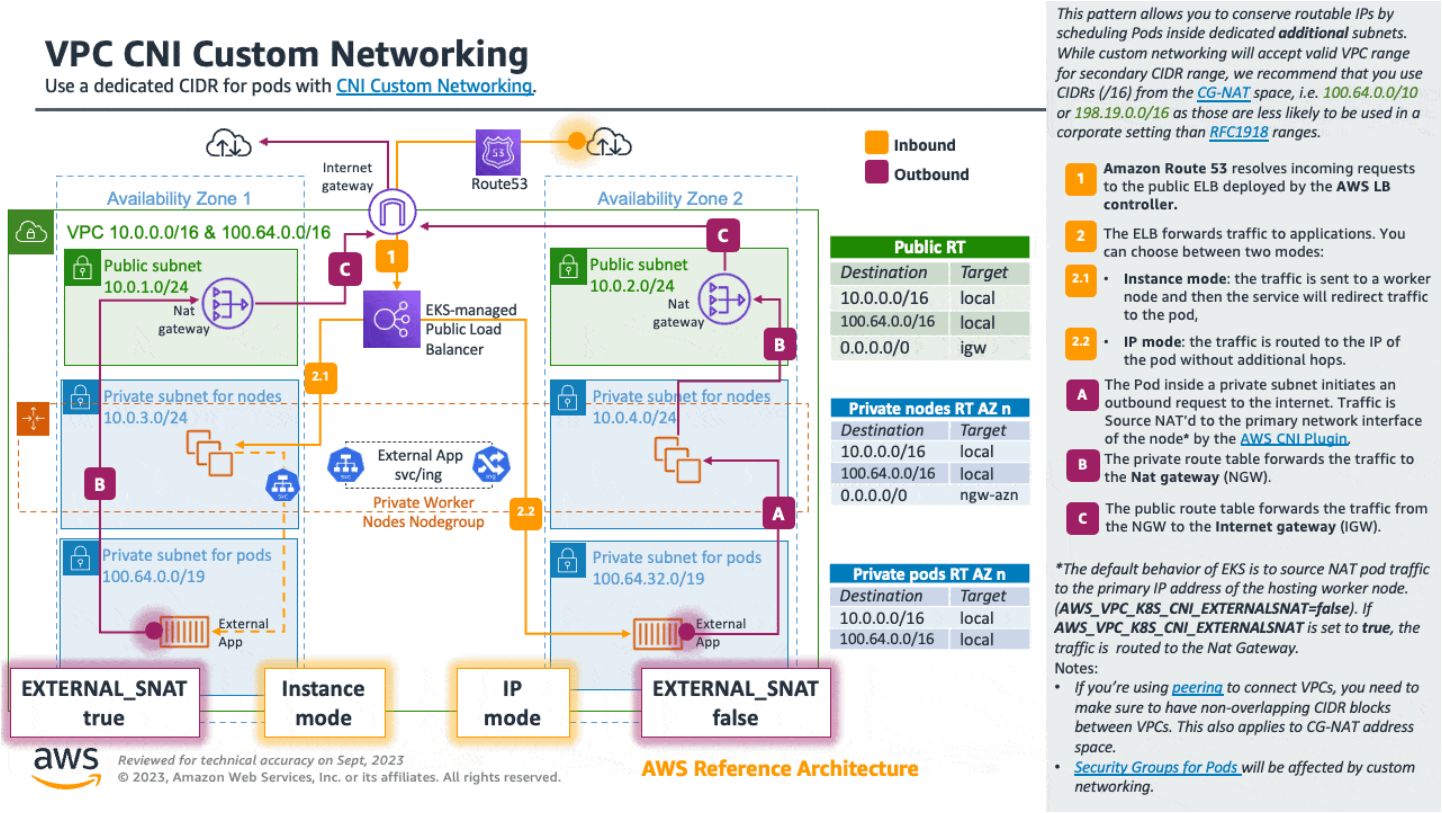
さらに、Amazon EKS は、コントロールプレーンへの通信を可能にするために必要な最大 4 つの Elastic Network Interface (X-ENI) を作成できます (詳細については、こちら)。クラスターのアップグレード中、Amazon EKS は新しい X-ENIs を作成し、アップグレードが成功すると古い X-ENI を削除します。このため、EKS クラスターに関連付けられたサブネットには、少なくとも /28 (16 IP アドレス) のネットマスクをお勧めします。

サンプル EKS Subnet Calculator スプレッドシートを使用して、ネットワークの計画を立てることができます。スプレッドシートは、ワークロードと VPC ENI 設定に基づいて IP 使用量を計算します。IP 使用量を IPv4 サブネットと比較して、設定とサブネットサイズがワークロードに十分かどうかを判断します。VPC 内のサブネットで使用可能な IP アドレスが不足している場合は、VPC の元の CIDR ブロックを使用して新しいサブネットを作成することをお勧めします。Amazon EKS がクラスターサブネットとセキュリティグループの変更を許可するようになりました。

カスタムネットワーキング

RFC1918 IP スペースを使い果たそうとする場合は、カスタムネットワーキングパターンを使用して、専用の追加サブネット内でポッドをスケジュールすることで、ルーティング可能な IPs を節約できます。カスタムネットワーキングはセカンダリ CIDR 範囲の有効な VPC 範囲を受け入れますが、CG-NAT スペースの CIDRs (/16) を使用することをお勧めします。つまり、RFC1918 範囲よりも企業設定で使用される可能性が低い`100.64.0.0/10``198.19.0.0/16`ためです。

詳細については、カスタムネットワーキングの専用セクションを参照してください。



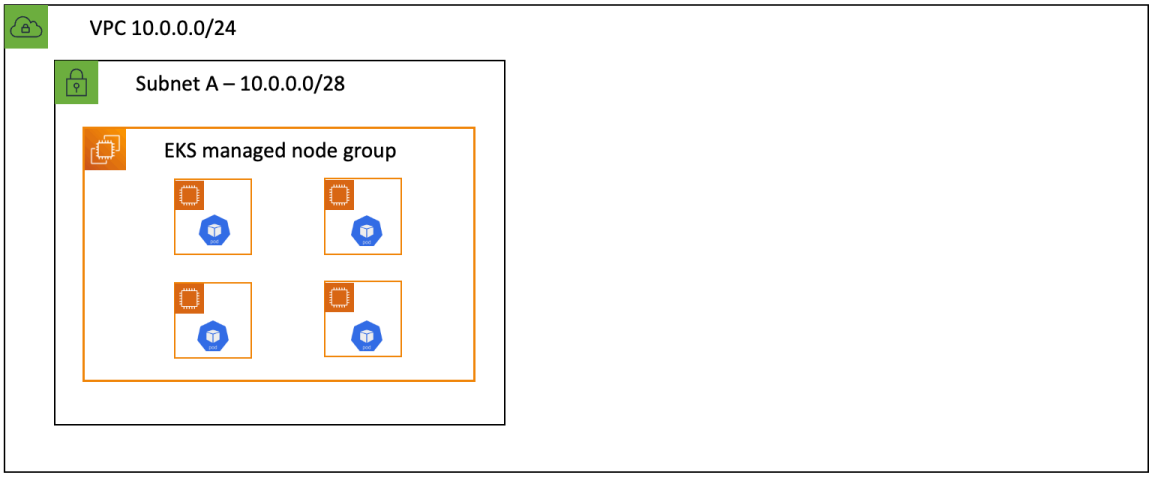
拡張サブネット検出

拡張サブネット検出は、Amazon VPC CNI で検出できるように新しいサブネットにタグを付けることで、IP を使い果たすための効率的なネットワーク設定の代替手段を提供します。拡張サブネット検出を使用すると、現在のワークロードが同じサブネットで実行され続け、Amazon Elastic Kubernetes Service (Amazon EKS) は新しい「使用可能なサブネット (複数可) 」で追加のポッドをスケジュールできるようになりました。

クラスターの現在のサブネットで IP アドレスが不足している場合は、次のように Amazon EKS クラスターにサブネットを追加するだけです。

1. 新しい CIDR ブロックを VPC に関連付けます。
2. 新しい CIDR ブロックに新しいサブネットを作成し、「kubernetes.io/role/cni」 =「1」でタグ付けします。
3. Amazon VPC CNI アドオンの「true」への ENABLE_SUBNET_DISCOVERY 設定を有効にします (バージョン 1.18.0 以降のデフォルト) 。

VPC および Amazon EKS クラスターで拡張サブネット検出を有効にすると、次の図に示すように、新しい Elastic Network Interface (ENIs) が Amazon EKS ノードにアタッチされます。



詳細については、AWS コンテナブログの「Amazon VPC CNI が拡張サブネット検出を導入する」を参照してください。

IPsムプールを最適化する

デフォルト設定では、VPC CNI は ENI 全体 (および関連する IPs) をウォームプールに保持します。これにより、特に大規模なインスタンスタイプでは、大量の IPs消費される可能性があります。

クラスターサブネットで使用できる IP アドレスの数に制限がある場合は、以下の VPC CNI 設定環境変数を精査します。

- `WARM_IP_TARGET`
- `MINIMUM_IP_TARGET`
- `WARM_ENI_TARGET`

ノードで実行する予定の Pod の数に厳密に一致する`MINIMUM_IP_TARGET`ように、 の値を設定できます。これにより、Pod が作成され、CNI は EC2 API を呼び出すことなくウォームプールから IP アドレスを割り当てることができます。

の値の設定が低WARM_IP_TARGETすぎると、EC2 API への追加の呼び出しが発生し、リクエストのスロットリングが発生する可能性があることに注意してください。大規模なクラスターでは、リクエストのスロットリングを避けるためMINIMUM_IP_TARGET、 とともに を使用します。

これらのオプションを設定するには、aws-k8s-cni.yamlマニフェストをダウンロードし、環境変数を設定します。執筆時点で、最新のリリースはこちらにあります。設定値のバージョンが、インストールされている VPC CNI バージョンと一致していることを確認します。

> **Warning**
>
> これらの設定は、CNI を更新するとデフォルトにリセットされます。更新する前に、CNI のバックアップを取得してください。設定を確認して、更新が成功した後に再適用する必要があるかどうかを確認します。

既存のアプリケーションのダウンタイムなしで CNI パラメータをその場で調整できますが、スケーラビリティのニーズをサポートする値を選択する必要があります。たとえば、バッチワークロードを使用している場合は、Pod WARM_ENI_TARGET スケールのニーズに合わせてデフォルトを更新することをお勧めします。を高い値WARM_ENI_TARGETに設定すると、大きなバッチワークロードの実行に必要なウォーム IP プールが常に維持されるため、データ処理の遅延を回避できます。

> **Warning**
>
> VPC 設計の改善は、IP アドレスの枯渇に対する推奨応答です。IPv6 やセカンダリ CIDRs。これらの値を調整してウォーム IPs の数を最小限に抑えることは、他のオプションを除外した後の一時的な解決策である必要があります。これらの値を誤って設定すると、クラスターオペレーションが妨げられる可能性があります。本番稼働用システムに変更を加える前に、このページの考慮事項を確認してください。

IP アドレスインベントリのモニタリング

上記のソリューションに加えて、IP 使用率を可視化することも重要です。CNI メトリクスヘルパーを使用して、サブネットの IP アドレスインベントリをモニタリングできます。使用可能なメトリクスには、次のようなものがあります。

- クラスターがサポートできる ENIs の最大数
- 既に割り当てられている ENIsの数

- Pod に現在割り当てられている IP アドレスの数
- 使用可能な IP アドレスの合計数と最大数

サブネットの IP アドレスが不足している場合に通知を受け取るように CloudWatch アラームを設定することもできます。

> ⚠ Warning
>
> VPC CNI の `DISABLE_METRICS`変数が false に設定されていることを確認します。

その他の考慮事項

IP の枯渇に役立つ Amazon EKS には組み込まれていない他のアーキテクチャパターンがあります。例えば、VPCs 間の通信を最適化したり、複数のアカウント間で VPC を共有して IPv4 アドレスの割り当てを制限したりできます。

これらのパターンの詳細については、以下を参照してください。

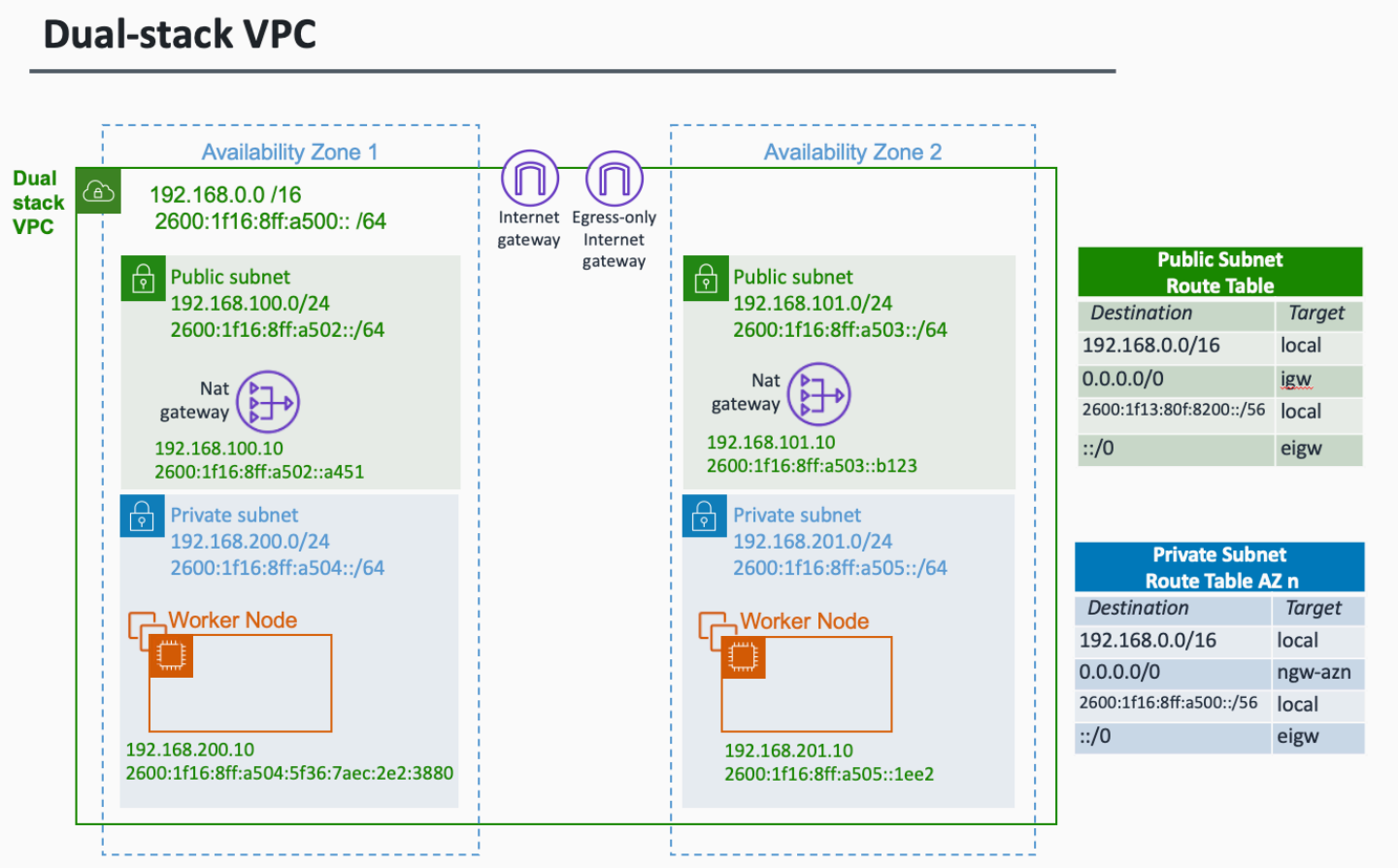
- ハイパースケール Amazon VPC ネットワークの設計
- Amazon VPC Lattice との安全なマルチアカウントマルチ VPC 接続を構築します。

IPv6 EKS クラスターの実行

IPv6 モードの EKS は、大規模な EKS クラスターでよく現れる IPv4 の枯渇の問題を解決します。EKS の IPv6 のサポートは、IPv4 アドレス空間のサイズが限られていることに起因する IPv4 の枯渇問題の解決に焦点を当てています。これは多くのお客様から寄せられる重大な懸念事項であり、Kubernetes IPv4/IPv6 デュアルスタック機能とは異なります。EKS/IPv6 は、IPv6 CIDRs を使用してネットワーク境界を相互接続する柔軟性も提供するため、CIDR の重複が発生する可能性を最小限に抑え、2-Foldの問題を解決します (クラスター内、クラスター間) 。EKS クラスターを IPv6 モード (--ip-family ipv6) でデプロイする場合、アクションは元に戻すことはできません。簡単に言うと、EKS IPv6 サポートはクラスターの存続期間中有効になります。

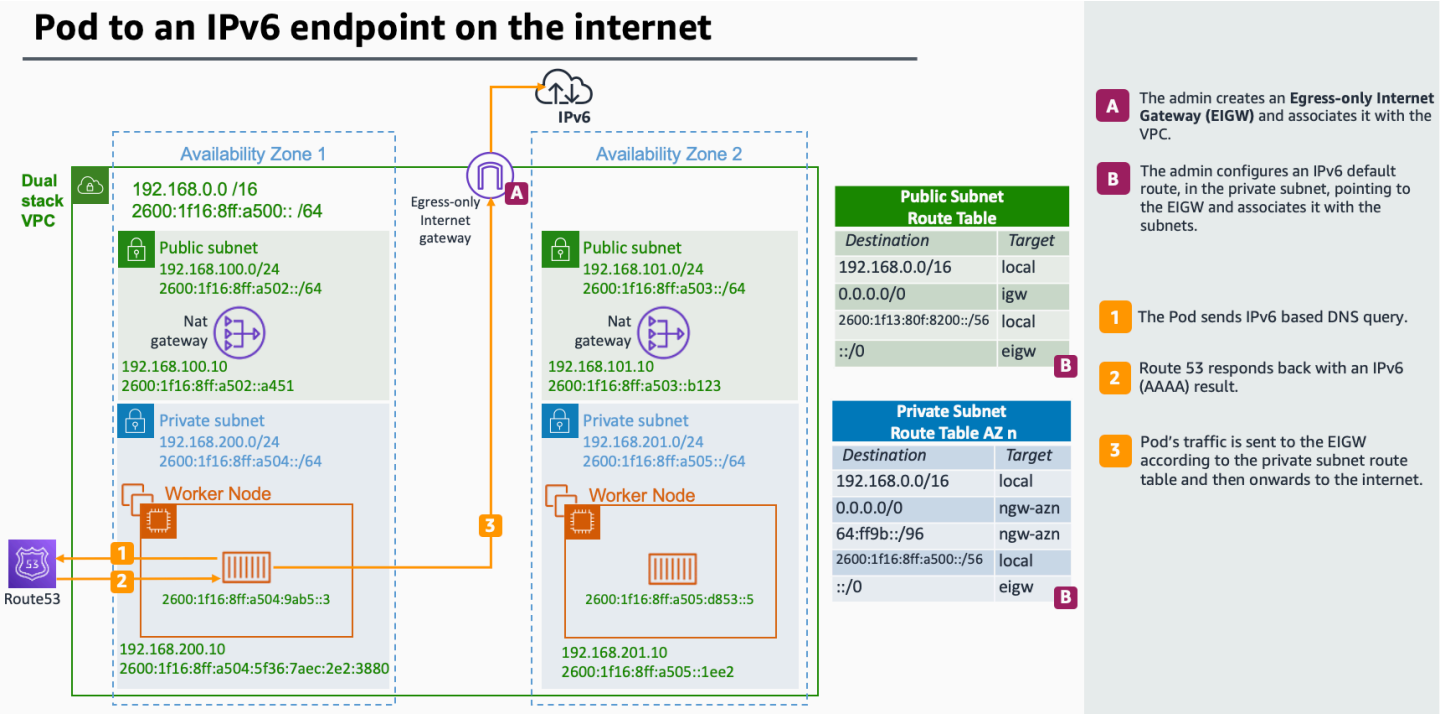
IPv6 EKS クラスターでは、ポッドとサービスはレガシー IPv4 エンドポイントとの互換性を維持しながら IPv6 アドレスを受け取ります。 IPv4 これには、外部 IPv4 エンドポイントがクラスター内サービスにアクセスしたり、ポッドが外部 IPv4 エンドポイントにアクセスしたりするための機能が含まれます。

Amazon EKS IPv6 サポートは、ネイティブ VPC IPv6 機能を活用します。各 VPC には、IPv4 アドレスプレフィックス (CIDR ブロックサイズは /16 ~ /28) と、Amazon の GUA (グローバルユニキャストアドレス) 内からの一意の /56 IPv6 アドレスプレフィックス (固定) が割り当てられます。VPC 内の各サブネットに /64 アドレスプレフィックスを割り当てることができます。ルートテーブル、ネットワークアクセスコントロールリスト、ピアリング、DNS 解決などの IPv4 機能は、IPv6 対応 VPC でも同じように動作します。次に、VPC はデュアルスタック VPC と呼ばれ、デュアルスタックサブネットに続き、次の図は EKS/IPV4IPv6IPv6 VPC 基盤パターンを示しています。



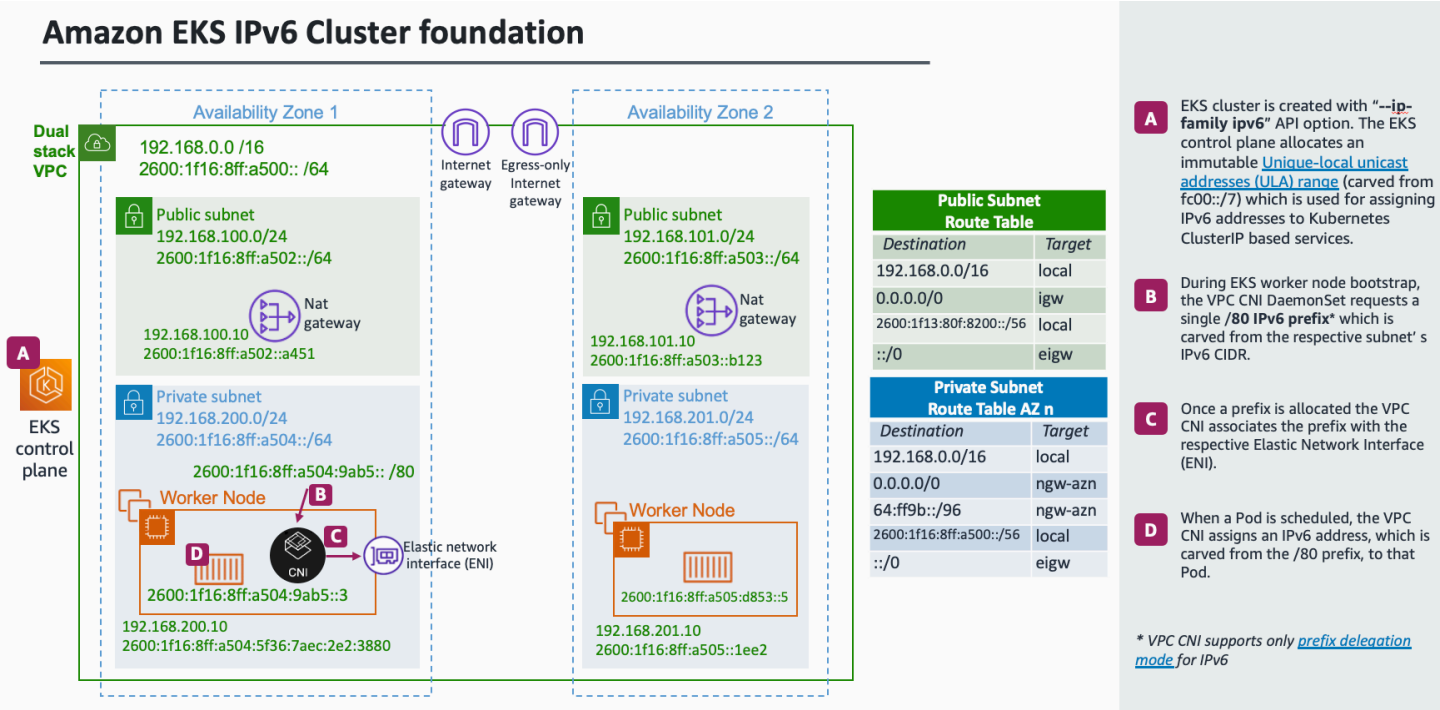
IPv6 の世界では、すべてのアドレスがインターネットでルーティング可能です。デフォルトでは、VPC はパブリック GUA 範囲から IPv6 CIDR を割り当てます。ただし、2024 年 8 月以降は、Amazon VPCs IP Address Manager (IPAM) で VPC とサブネットにプライベート IPv6 アドレス指定を使用することもできます。詳細については、この AWS Networking ブログ記事と VPC ドキュメントを参照してください。

次の図は、EKS/IPv6 クラスター内の Pod IPv6 インターネット出力フローを示しています。



IPv6 サブネットを実装するためのベストプラクティスについては、VPC ユーザーガイドを参照してください。

IPv6 EKS クラスターでは、ノードとポッドはパブリック IPv6 アドレスを受け取ります。EKS は、一意のローカル IPv6 ユニキャストアドレス (ULA) に基づいて、サービスに IPv6 アドレスを割り当てます。IPv6 クラスターの ULA サービス CIDR は、クラスター作成段階で自動的に割り当てられ、IPv4 とは異なり、指定することはできません。次の図は、EKS/IPv6 ベースのクラスターコントロールプレーンデータプラン基盤パターンを示しています。



概要:

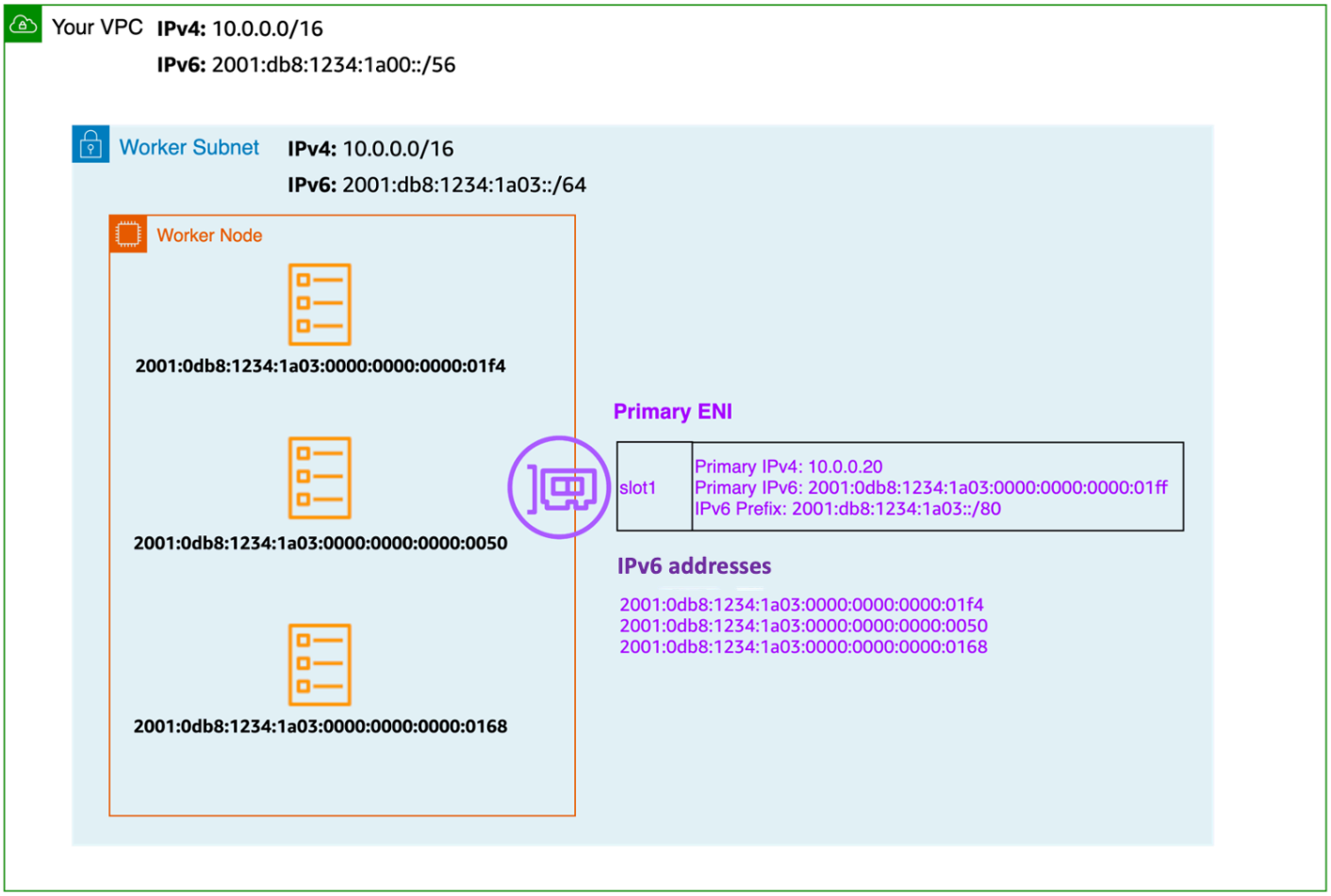
EKS/IPv6 はプレフィックスモード (VPC-CNI プラグイン ENI IP 割り当てモード) でのみサポートされています。プレフィックスモードの詳細については、「」を参照してください。

プレフィックスの割り当ては Nitro ベースの EC2 インスタンスでのみ機能するため、EKS/IPv6 はクラスターデータプレーンが EC2 Nitro ベースのインスタンスを使用する場合にのみサポートされます。

簡単に言うと、IPv6 プレフィックス /80 (ワーカーノードあたり) は最大 10^14 個の IPv6 アドレスを生成するため、制限要因は IPs ではなくポッド密度 (リソース単位) になります。

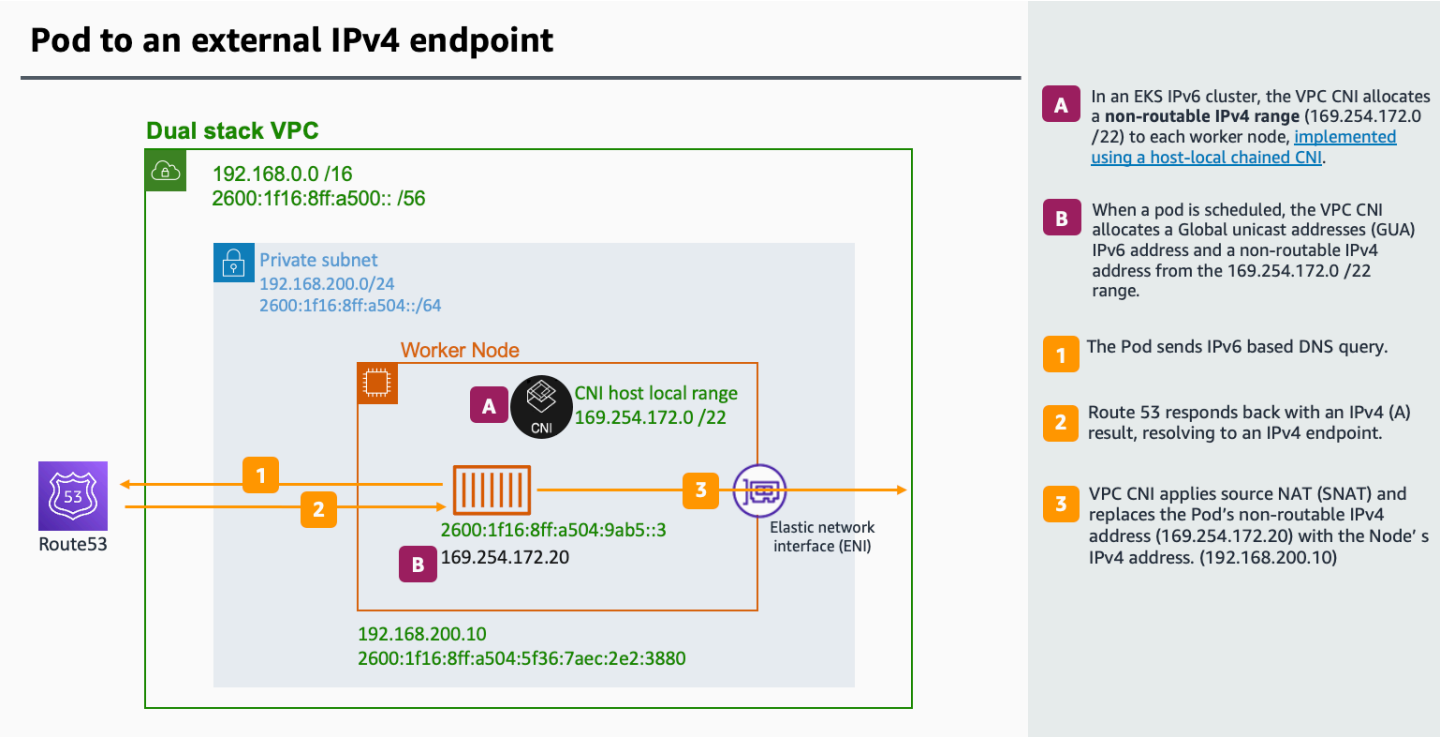
IPv6 プレフィックスの割り当ては、EKS ワーカーノードブートストラップ時にのみ発生します。この動作は、プライベート IPv4 アドレスを適時に割り当てることを目的とした VPC CNI プラグイン (ipamd) によって生成されたスロットリングされた API コールが原因で、Pod のスケジューリングで高 Pod チャーン EKS/IPv4 クラスターが遅延することがよくあるシナリオを軽減することが知られています。また、VPC-CNI プラグインの高度なつまみが WARM_IP/ENI、MINIMUM_IP を不必要にチューニングすることも知られています。

次の図は、IPv6 ワーカーノードの Elastic Network Interface (ENI) を拡大したものです。



すべての EKS ワーカーノードには、対応する DNS エントリとともに IPv4 アドレスと IPv6 アドレスが割り当てられます。特定のワーカーノードでは、デュアルスタックサブネットの単一の IPv4 アドレスのみが消費されます。IPv6 の EKS サポートにより、高度に評価された Egress-Only IPv4 モデルを通じて IPv4 エンドポイント (AWS、オンプレミス、インターネット) と通信できます。EKS は、VPC CNI プラグインのセカンダリであるホストローカル CNI プラグインを実装し、ポッドの IPv4 アドレスを割り当てて設定します。CNI プラグインは、169.254.172.0/22 の範囲から Pod のホスト固有のルーティング不可能な IPv4 アドレスを設定します。Pod に割り当てられた IPv4 アドレスはワーカーノードに固有であり、ワーカーノードを超えてアドバタイズされません。169.254.172.0/22 は、大規模なインスタンスタイプをサポートできる最大 1024 個の一意の IPv4 アドレスを提供します。

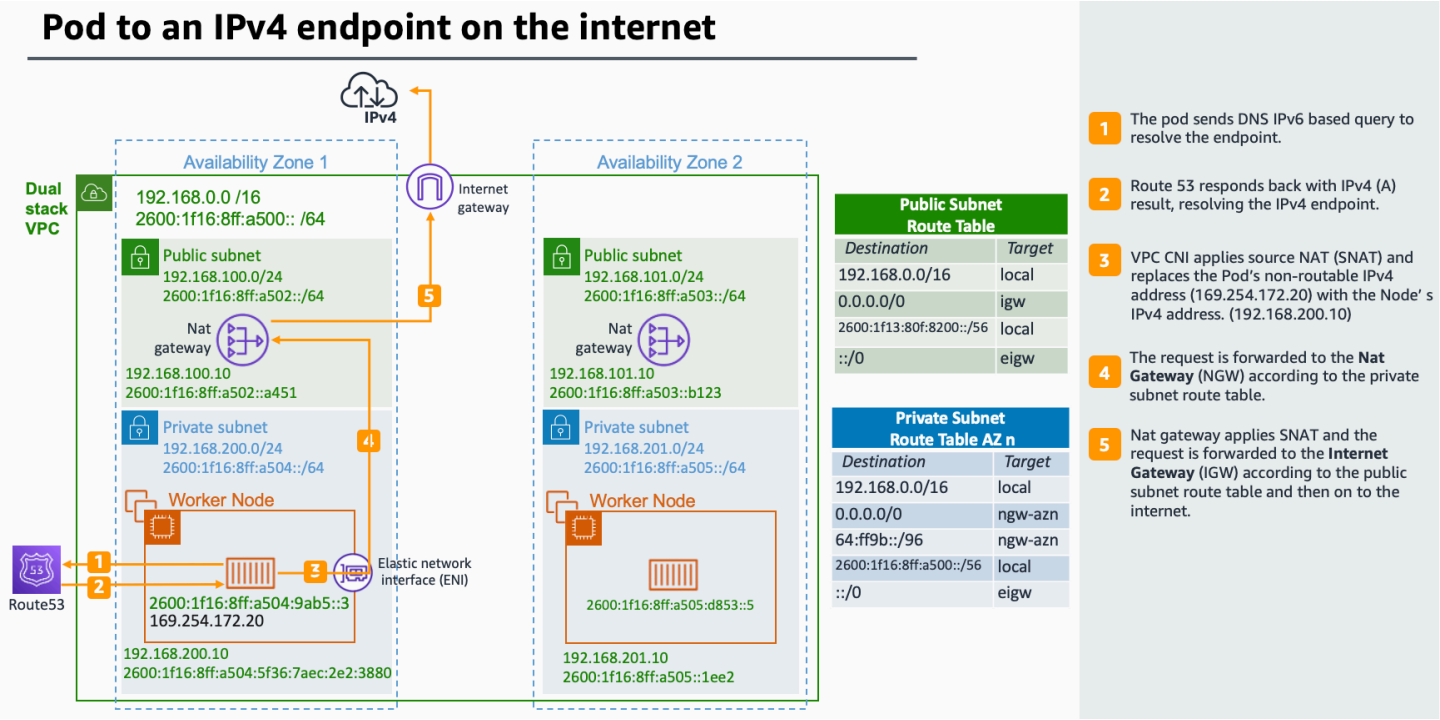
次の図は、クラスター境界 (非インターネット) 外の IPv4 エンドポイントに接続する IPv6 Pod のフローを示しています。 IPv4



上記の図では、Pod はエンドポイントの DNS ルックアップを実行し、IPv4 の「A」レスポンスを受信すると、Pod のノードのみの一意の IPv4 アドレスは、ソースネットワークアドレス変換 (SNAT) を介して EC2 ワーカーノードにアタッチされたプライマリネットワークインターフェイスのプライベート IPv4 (VPC) アドレスに変換されます。

> **Note**
>
> 上記のパターンでは、EKS/IPv6 Pod が実行されているサブネットで DNS64 を無効にする必要があります。IPv6 DNS64 を有効にすると、DNS リゾルバーは IPv4 専用エンドポイントの合成された IPv6 アドレスと IPv4 アドレスを返します。 IPv4-only その結果、トラフィックは、上記のパターンに示すように VPC 内に留まる代わりに、NAT ゲートウェイの (アーキテクチャに含まれている場合) NAT64 機能を介してルーティングされます。これにより、予期しない NAT Gateway の使用と関連コストが発生する可能性があります。

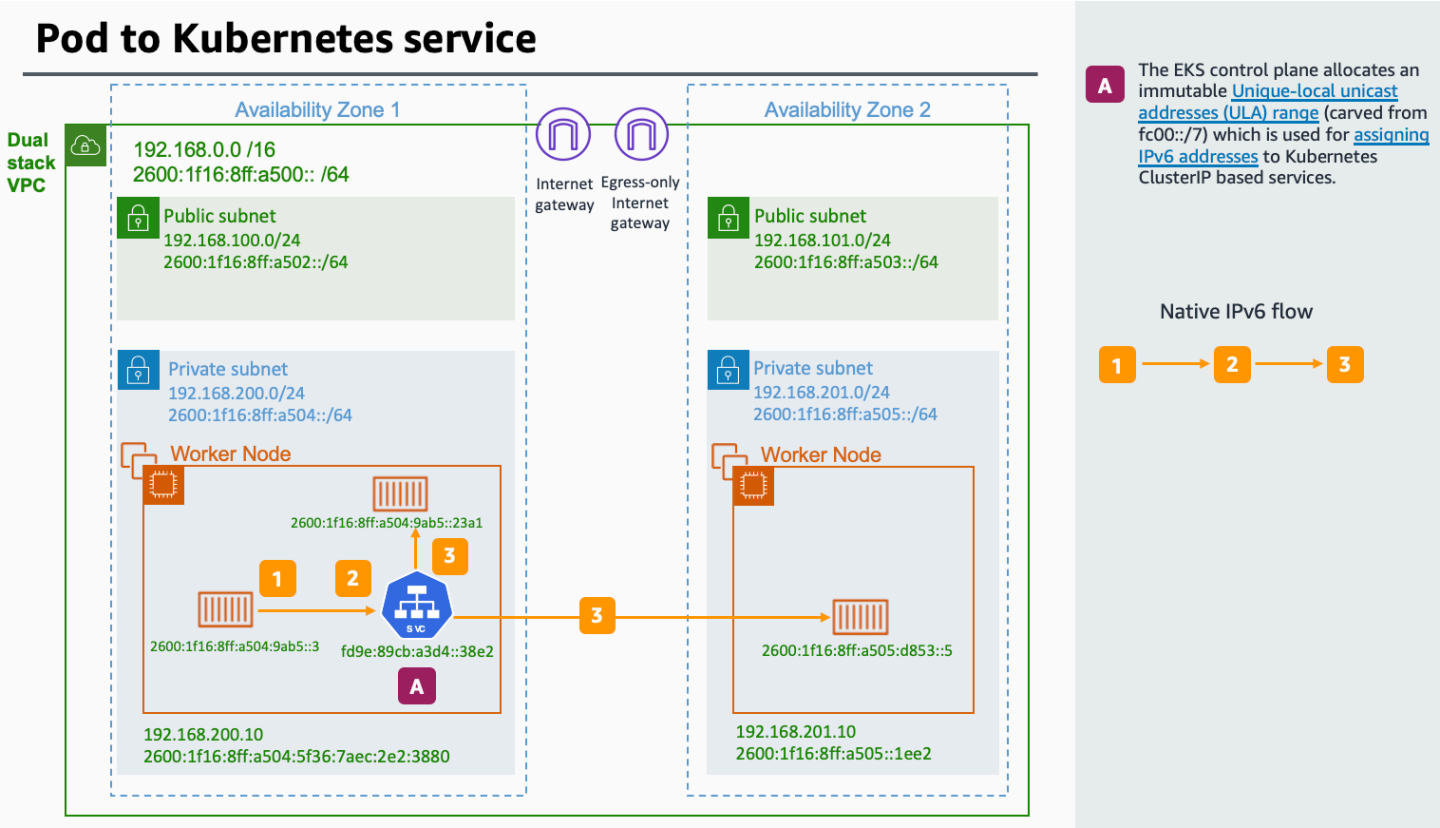
EKS/IPv6 Pod は、同様のフローが存在するように、パブリック IPv4 アドレスを使用してインターネット経由で IPv4 エンドポイントに接続する必要があります。次の図は、クラスター境界外の IPv4 エンドポイントに接続する IPv6 Pod のフローを示しています (インターネットをルーティング可能) 。 IPv4



上記の図では、Pod はエンドポイントの DNS ルックアップを実行し、IPv4 の「A」レスポンスを受信すると、Pod のノードのみの一意の IPv4 アドレスは、ソースネットワークアドレス変換 (SNAT) を介して EC2 ワーカーノードにアタッチされたプライマリネットワークインターフェイスのプライベート IPv4 (VPC) アドレスに変換されます。その後、ポッド IPv4 アドレス (ソース IPv4: EC2 プライマリ IP) は IPv4 NAT ゲートウェイにルーティングされ、EC2 プライマリ IP は有効なインターネットルーティング可能な IPv4 パブリック IP アドレス (NAT ゲートウェイ割り当てパブリック IP) に変換されます。

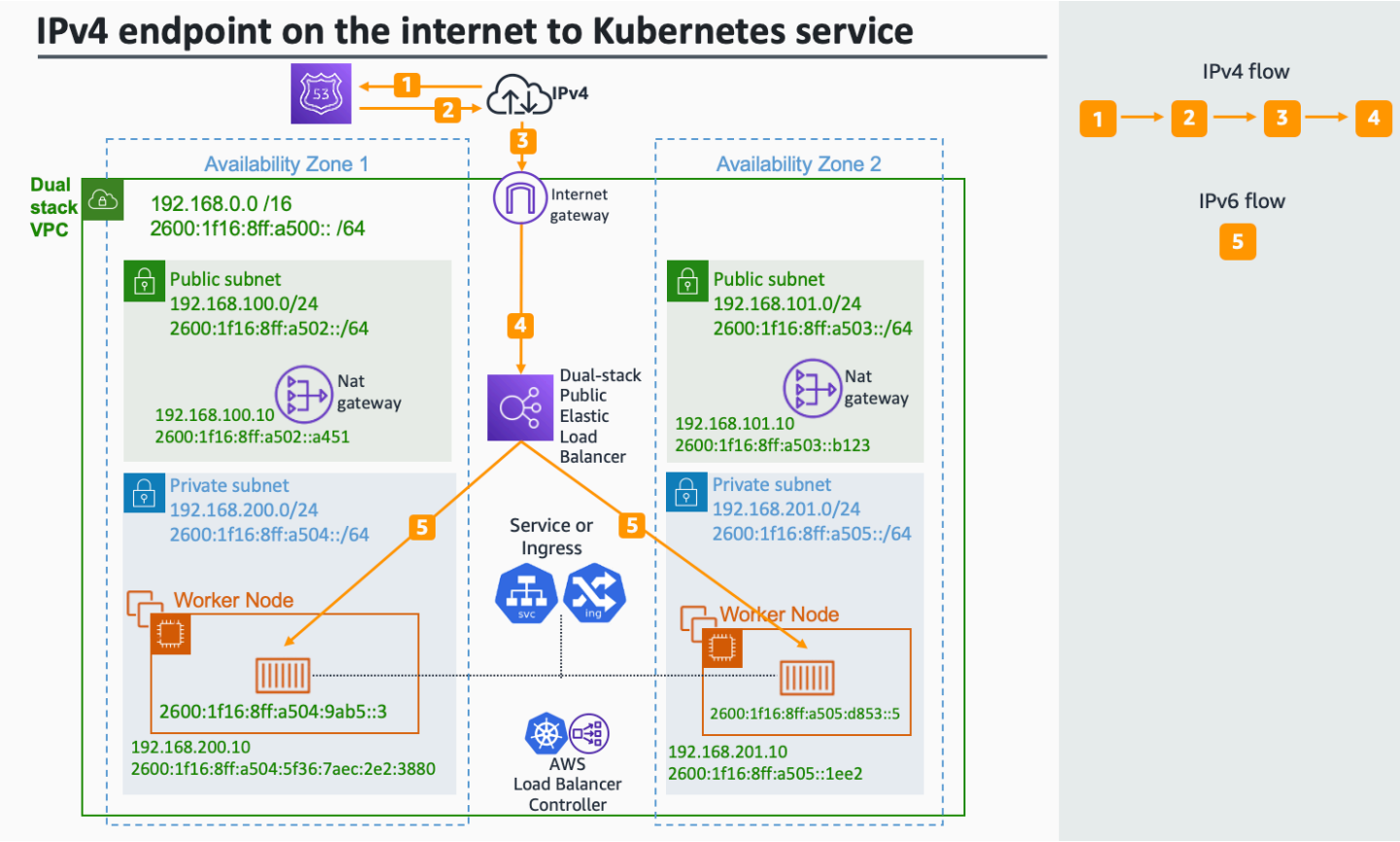
ノード間の Pod-to-Pod通信では、常に IPv6 アドレスが使用されます。VPC CNI は、IPv4 接続をブロックしながら IPv6 を処理するように iptables を設定します。 IPv4

Kubernetes サービスは、一意のローカル IPv6 ユニキャストアドレス (ULA) から IPv6 アドレス (ClusterIP) のみを受け取ります。 IPv6 IPv6 クラスターの ULA サービス CIDR は、EKS クラスターの作成段階で自動的に割り当てられ、変更することはできません。次の図は、ポッドから Kubernetes サービスへのフローを示しています。



サービスは、AWS ロードバランサーを使用してインターネットに公開されます。ロードバランサーは、パブリック IPv4 アドレスと IPv6 アドレス、a.k.a デュアルスタックロードバランサーを受け取ります。IPv6 クラスター kubernetes サービスにアクセスする IPv64 クライアントの場合、ロードバランサーは IPv4 から IPv6 への変換を行います。

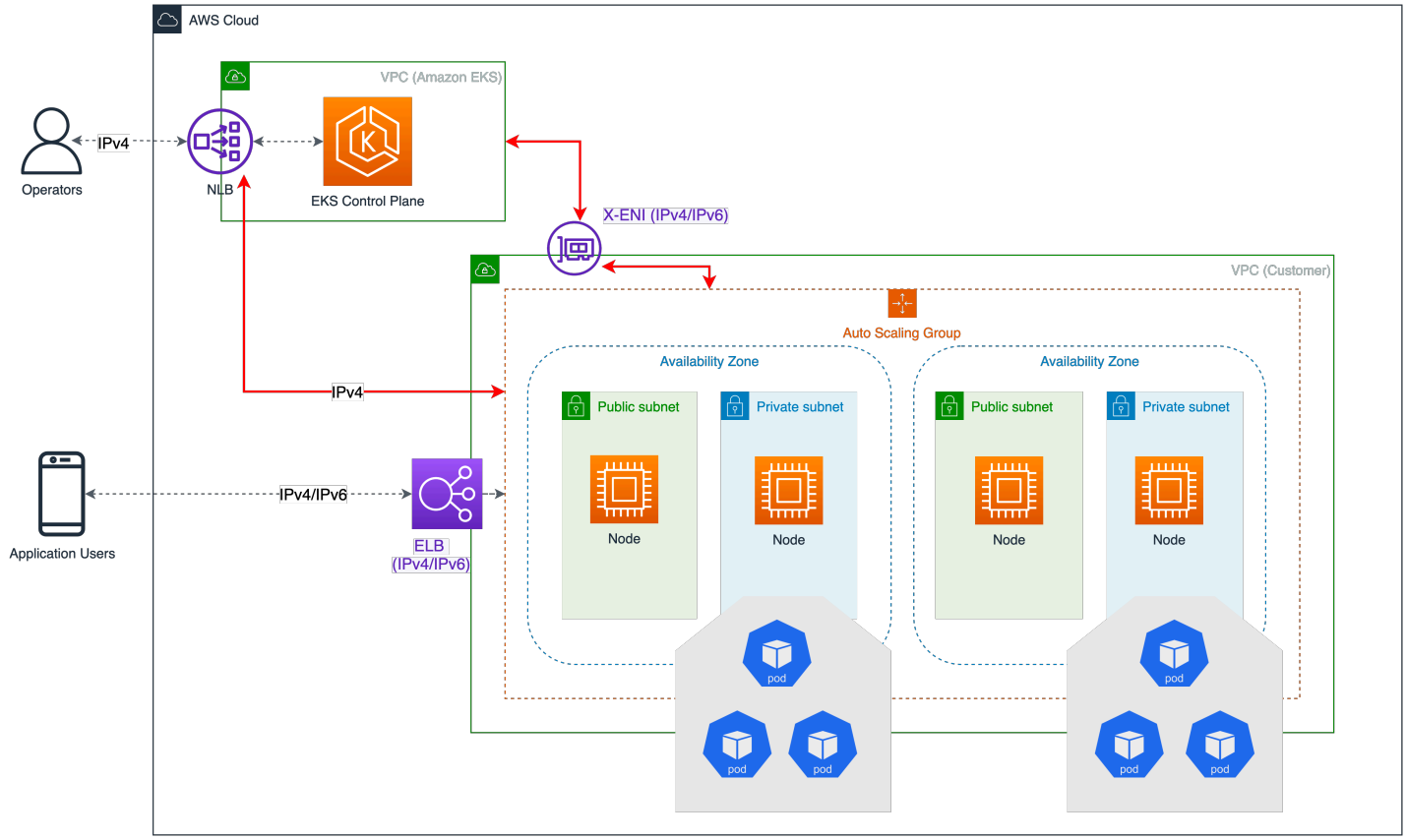
Amazon EKS では、プライベートサブネットでワーカーノードとポッドを実行することをお勧めします。パブリックサブネットにパブリックロードバランサーを作成して、プライベートサブネットにあるノードで実行されている Pod へのトラフィックをロードバランシングできます。次の図は、EKS/IPv6 Ingress ベースのサービスにアクセスするインターネット IPv4 ユーザーを示しています。IPv6



> **Note**
>
> 上記のパターンでは、AWS ロードバランサーコントローラーの最新バージョンをデプロイする必要があります

EKS コントロールプレーンデータプレーン通信

EKS は、デュアルスタックモード (IPv4/IPv6) でクロスアカウント ENIs (X-ENIs) をプロビジョニングします。kubelet や kube-proxy などの Kubernetes ノードコンポーネントは、デュアルスタックをサポートするように設定されています。Kubelet と kube-proxy は hostNetwork モードで実行され、ノードのプライマリネットワークインターフェイスにアタッチされた IPv4 アドレスと IPv6 アドレスの両方にバインドされます。Kubernetes API サーバーは、X-ENIs を介して Pod とノードコンポーネントと通信しますIPv6 ベースです。ポッドは X-ENIs を介して API サーバーと通信し、ポッドから API サーバーへの通信は常に IPv6 モードを使用します。



推奨事項

コンピューティングリソースに基づくスケジュール

1 つのノードで多数の Pod を実行するには、1 つの IPv6 プレフィックスで十分です。これにより、ノード上のポッドの最大数に対する ENI と IP の制限も効果的に削除されます。IPv6 は max-Pod への直接的な依存関係を削除しますが、m5.large などの小さなインスタンスタイプでプレフィックスアタッチメントを使用する場合、IP アドレスを使い果たす前にインスタンスの CPU リソースとメモリリソースを使い果たす可能性があります。セルフマネージド型ノードグループまたはカスタム AMI ID を持つマネージド型ノードグループを使用している場合は、EKS が推奨する最大 Pod 値を手動で設定する必要があります。

次の式を使用して、IPv6 EKS クラスターのノードにデプロイできる Pod の最大数を決定できます。

```
((Number of network interfaces for instance type (number of prefixes per network
 interface-1)* 16) + 2
```

```
((3 ENIs)_((10 secondary IPs per ENI-1)_ 16)) + 2 = 460 (real)
```

マネージド型ノードグループは、Pod の最大数を自動的に計算します。リソースの制限によるポッドスケジューリングの失敗を避けるため、ポッドの最大数に対する EKS の推奨値を変更しないでください。

既存のカスタムネットワークの目的を評価する

カスタムネットワークが現在有効になっている場合、Amazon EKS では IPv6 を使用してそのネットワークの必要性を再評価することをお勧めします。カスタムネットワークを使用して IPv4 の枯渇問題に対処することを選択した場合、IPv6 では不要になります。ノードとポッドの個別のネットワークなど、セキュリティ要件を満たすためにカスタムネットワークを使用している場合は、EKS ロードマップリクエストを送信することをお勧めします。

EKS/IPv6 クラスターの Fargate Pod

EKS は、Fargate で実行されているポッドの IPv6 をサポートしています。Fargate で実行されているポッドは、VPC CIDR 範囲 (IPv6) から切り取られた IPv4IPv6 アドレスを使用します。 IPv4 簡単に言うと、EKS/Fargate Pods クラスター全体の密度は、使用可能な IPv4 アドレスと IPv6 アドレスに制限されます。将来の成長に備えて、デュアルスタックサブネット/VPC CIDRsのサイズを設定することをお勧めします。基盤となるサブネットに使用可能な IPv6 アドレスに関係なく、使用可能な IPv4 アドレスが含まれていない場合、新しい Fargate Pod をスケジュールすることはできませんIPv6。

AWS Load Balancer Controller (LBC) をデプロイする

アップストリームのツリー内 Kubernetes サービスコントローラーは IPv6 をサポートしていません。AWS Load Balancer Controller アドオンの最新バージョンを使用することをお勧めします。LBC は、次の注釈が付いた対応する kubernetes サービス/イングレス定義を消費した場合にのみ、デュアルスタック NLB またはデュアルスタック ALB をデプロイ`"alb.ingress.kubernetes.io/ip-address-type: dualstack"`します。 `"alb.ingress.kubernetes.io/target-type: ip"`

カスタムネットワーキング

デフォルトでは、Amazon VPC CNI はプライマリサブネットから選択された IP アドレスをポッドに割り当てます。プライマリサブネットは、プライマリ ENI がアタッチされているサブネット CIDR で、通常はノード/ホストのサブネットです。

サブネット CIDR が小さすぎる場合、CNI は Pod に割り当てるのに十分なセカンダリ IP アドレスを取得できない可能性があります。これは EKS IPv4 クラスターの一般的な課題です。

カスタムネットワークは、この問題の解決策の 1 つです。

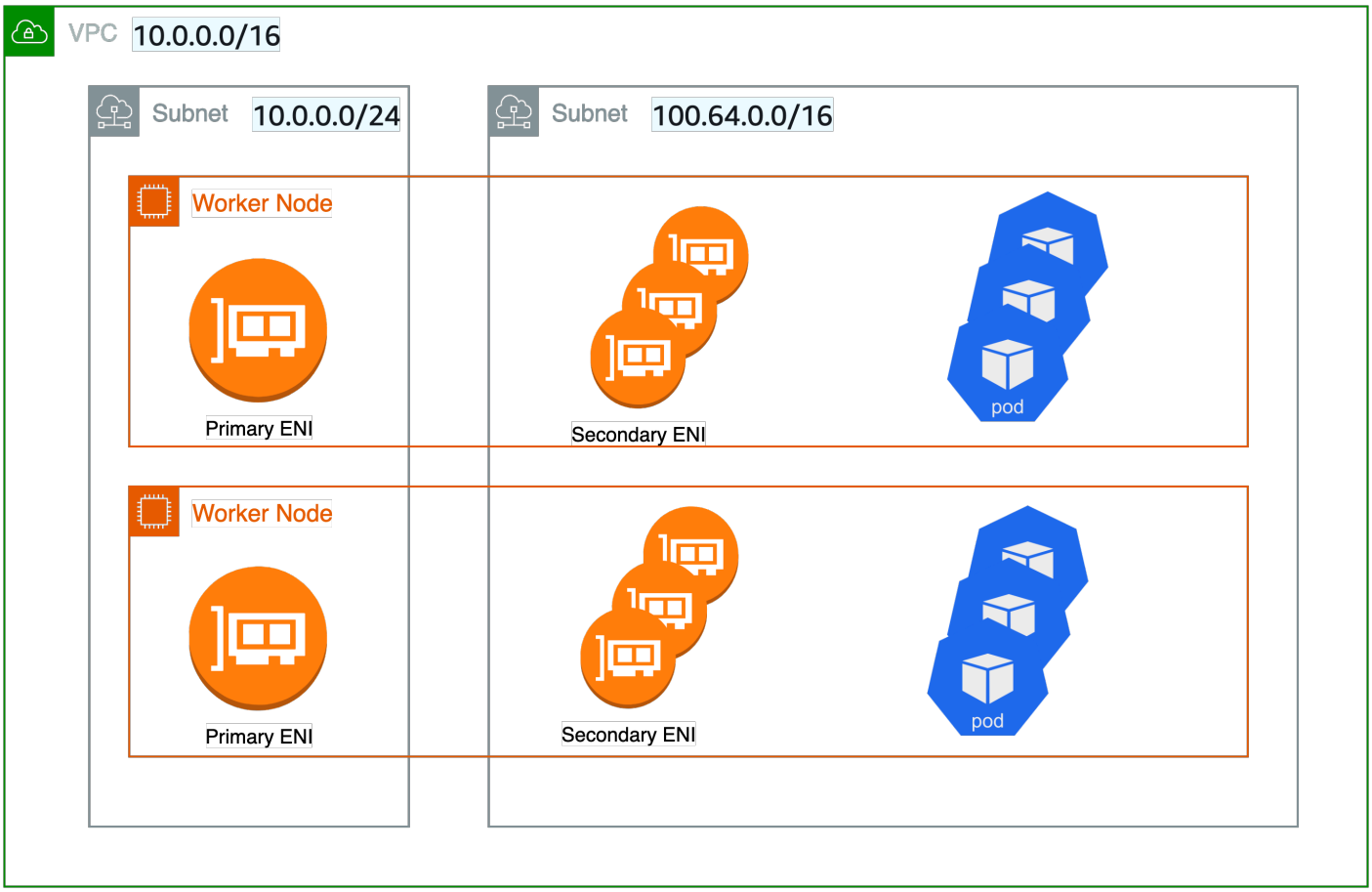
カスタムネットワークは、セカンダリ VPC アドレススペース (CIDR) からノード IP と Pod IPs を割り当てることで、IP の枯渇の問題に対処します。カスタムネットワークサポートはENIConfig カスタムリソースをサポートします。ENIConfig には、ポッドが属するセキュリティグループ (複数可) とともに、代替サブネット CIDR 範囲 (セカンダリ VPC CIDR から取得) が含まれます。カスタムネットワーキングを有効にすると、VPC CNI は ENIs ENIConfig を作成します。CNI は、ENIConfig CRD で定義された CIDR 範囲から IP アドレスをポッドに割り当てます。

プライマリ ENI はカスタムネットワークでは使用されないため、ノードで実行できる Pod の最大数は少なくなります。ホストネットワークポッドは、プライマリ ENI に割り当てられた IP アドレスを引き続き使用します。さらに、プライマリ ENI はソースネットワーク変換を処理し、Pod トラフィックをノード外にルーティングするために使用されます。

構成例

カスタムネットワーキングはセカンダリ CIDR 範囲に有効な VPC 範囲を受け入れますが、CG-NAT スペースの CIDRs (/16) を使用することをお勧めします。つまり、100.64.0.0/10 または 198.19.0.0/16 は、他の RFC1918 範囲よりも企業設定で使用される可能性が低いためです。VPC で使用できる許可および制限された CIDR ブロックの関連付けの詳細については、VPC ドキュメントの「VPC とサブネットのサイズ設定」セクションのIPv4 CIDR ブロックの関連付けの制限」を参照してください。

次の図に示すように、ワーカーノードのプライマリ Elastic Network Interface (ENI) は引き続きプライマリ VPC CIDR 範囲 (この場合は 10.0.0.0/16) を使用しますがENIs はセカンダリ VPC CIDR 範囲 (この場合は 100.64.0.0/16) を使用します。ここで、ポッドで 100.64.0.0/16 CIDR 範囲を使用するには、カスタムネットワークを使用するように CNI プラグインを設定する必要があります。ここに記載されている手順を実行できます。



CNI でカスタムネットワーキングを使用する場合は、AWS_VPC_K8S_CNI_CUSTOM_NETWORK_CFG環境変数を に設定しますtrue。

```
kubectl set env daemonset aws-node -n kube-system
 AWS_VPC_K8S_CNI_CUSTOM_NETWORK_CFG=true
```

の場合AWS_VPC_K8S_CNI_CUSTOM_NETWORK_CFG=true、CNI は で定義されたサブネットから Pod IP アドレスを割り当てますENIConfig。ENIConfig カスタムリソースは、ポッドがスケジュールされるサブネットを定義するために使用されます。

```
apiVersion : crd.k8s.amazonaws.com/v1alpha1
kind : ENIConfig
metadata:
  name: us-west-2a
spec:
  securityGroups:
    - sg-0dff111a1d11c1c11
```

```
  subnet: subnet-011b111c1f11fdf11
```

ENIconfig カスタムリソースを作成するときは、新しいワーカーノードを作成し、既存のノードをドレインする必要があります。既存のワーカーノードとポッドは影響を受けません。

推奨事項

のときにカスタムネットワーキングを使用する

IPv4 の枯渇に対処していて、まだ IPv6 を使用できない場合は、カスタムネットワークを検討することをお勧めします。Amazon EKS による RFC6598 スペースのサポートにより、RFC1918 を超えて Pod をスケールし、枯渇の問題に対処できます。ノードの Pods 密度を高めるために、カスタムネットワークでプレフィックス委任を使用することを検討してください。

セキュリティグループの要件が異なる別のネットワークで Pod を実行するセキュリティ要件がある場合は、カスタムネットワークを検討してください。カスタムネットワークを有効にすると、ポッドはノードのプライマリネットワークインターフェイスとは異なるサブネットまたはセキュリティグループを ENIConfig で定義して使用します。

カスタムネットワーキングは、複数の EKS クラスターとアプリケーションをデプロイしてオンプレミスのデータセンターサービスを接続するのに最適なオプションです。Amazon Elastic Load Balancing や NAT-GW などのサービスの VPC 内の EKS にアクセスできるプライベートアドレス (RFC1918) の数を増やすことができますが、複数のクラスターで Pod にルーティング不可能な CG-NAT スペースを使用します。トランジットゲートウェイと共有サービス VPC (高可用性のために複数のアベイラビリティーゾーンにまたがる NAT ゲートウェイを含む) とのカスタムネットワーキングにより、スケーラブルで予測可能なトラフィックフローを提供できます。このブログ記事では、カスタムネットワークを使用して EKS Pod をデータセンターネットワークに接続するための最も推奨される方法の 1 つであるアーキテクチャパターンについて説明します。

カスタムネットワーキングを避ける

IPv6 を実装する準備ができました

カスタムネットワークは IP の枯渇の問題を軽減できますが、追加の運用オーバーヘッドが必要です。現在デュアルスタック (IPv4/IPv6) VPC をデプロイしている場合、またはプランに IPv6 サポートが含まれている場合は、代わりに IPv6 クラスターを実装することをお勧めします。IPv6 EKS クラスターをセットアップし、アプリケーションを移行できます。IPv6 EKS クラスターでは、Kubernetes と Pod の両方が IPv6 アドレスを取得し、IPv4 エンドポイントと IPv6 エンドポイ

ントの両方と通信できます。IPv6 EKS クラスターを実行するためのベストプラクティスを確認してください。

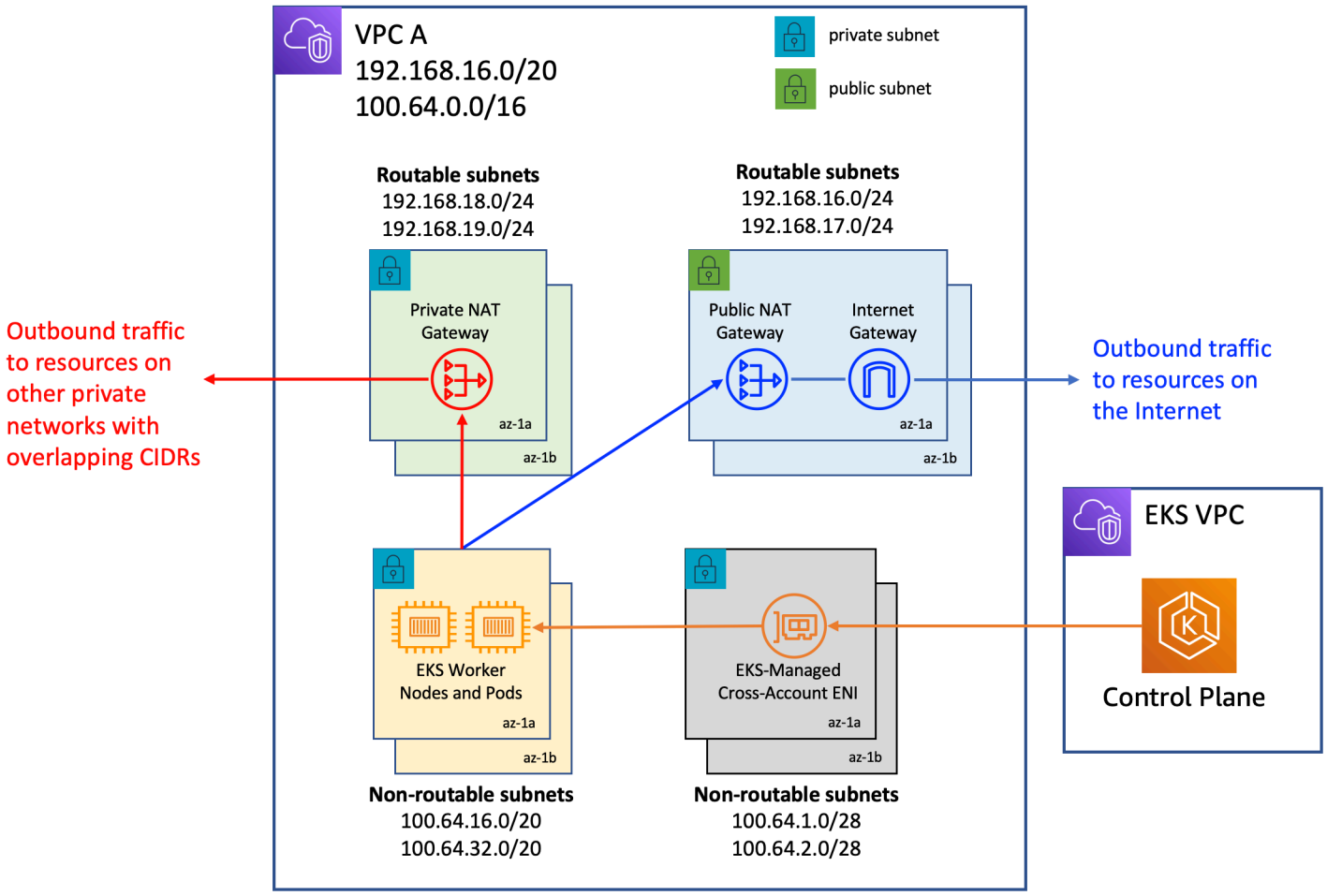
枯渇した CG-NAT スペース

さらに、現在 CG-NAT スペースの CIDRs を使用しているか、セカンダリ CIDR をクラスター VPC にリンクできない場合は、代替 CNI を使用するなど、他のオプションを検討する必要があります。商用サポートを受けるか、オープンソースの CNI プラグインプロジェクトにパッチをデバッグして送信するための社内知識を持つことを強くお勧めします。詳細については、「代替 CNI プラグインユーザーガイド」を参照してください。

プライベート NAT ゲートウェイを使用する

Amazon VPC でプライベート NAT ゲートウェイ機能が提供されるようになりました。Amazon のプライベート NAT Gateway を使用すると、プライベートサブネット内のインスタンスは、重複する CIDR を持つ他の VPCs およびオンプレミスネットワークに接続できます。 CIDRs このブログ記事で説明されている方法を使用して、プライベート NAT ゲートウェイを採用して、CIDRsこれは、クライアントによって表明された重大な苦情です。カスタムネットワークは、重複する CIDR の問題に単独で対処できず、設定の課題に追加されます。

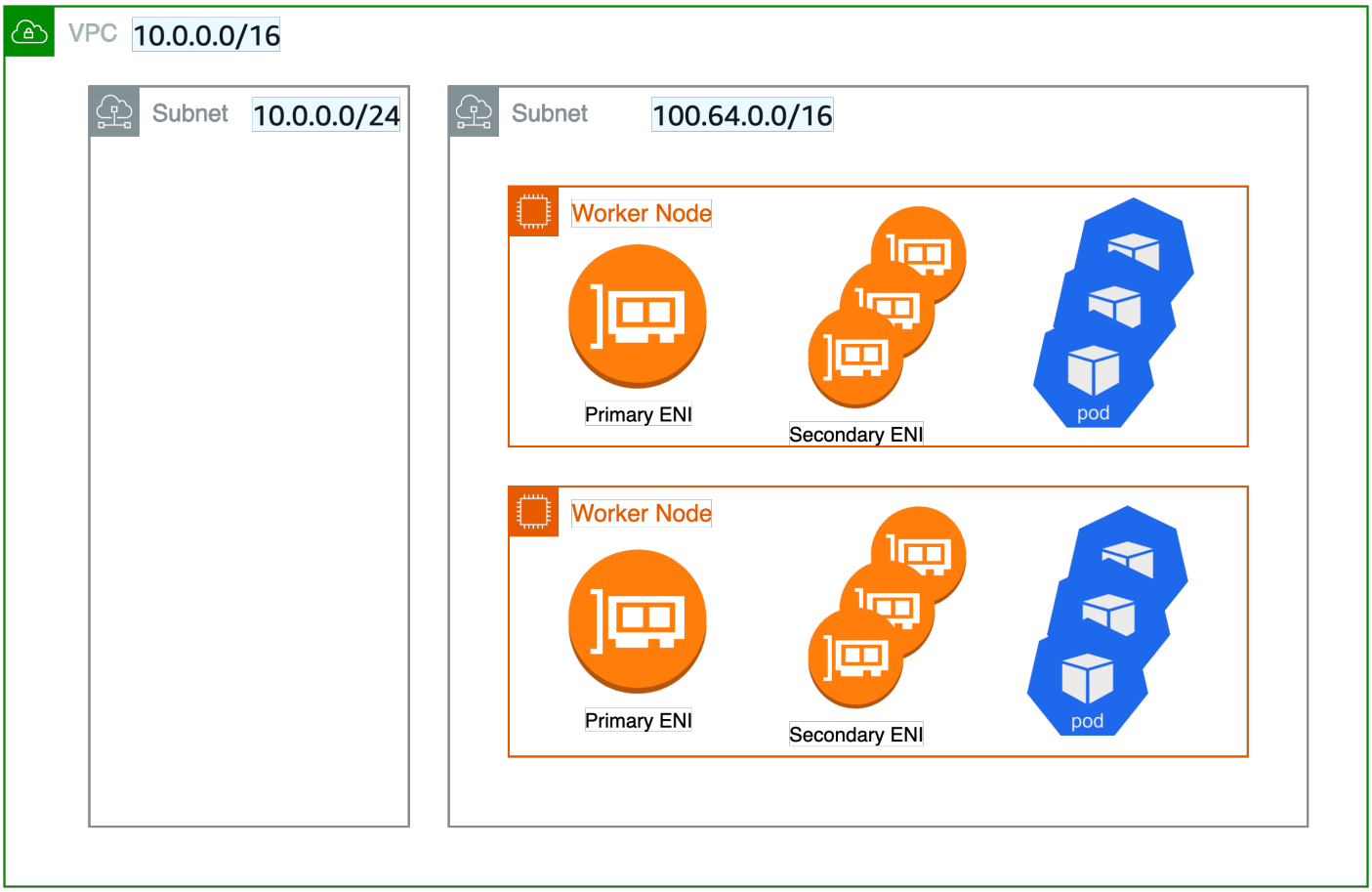
このブログ投稿実装で使用されるネットワークアーキテクチャは、Amazon VPC ドキュメントの「重複するネットワーク間の通信を有効にする」の推奨事項に従います。このブログ記事で示すように、RFC6598 アドレスと組み合わせてプライベート NAT ゲートウェイの使用を拡張して、顧客のプライベート IP 枯渇の問題を管理することができます。EKS クラスター、ワーカーノードはルーティング不可能な 100.64.0.0/16 VPC セカンダリ CIDR 範囲にデプロイされますが、プライベート NAT ゲートウェイ、NAT ゲートウェイはルーティング可能な RFC1918 CIDR 範囲にデプロイされます。このブログでは、ルーティング不可能な CIDR 範囲が重複する VPCs 間の通信を容易にするために、トランジットゲートウェイを使用して VPCs を接続する方法について説明します。VPC のルーティング不可能なアドレス範囲の EKS リソースが重複するアドレス範囲を持たない他の VPCs と通信する必要があるユースケースでは、VPC ピアリングを使用してそのような VPCs を相互接続できます。この方法は、VPC ピアリング接続を介したアベイラビリティーゾーン内のすべてのデータ転送が無料になったため、潜在的なコスト削減につながる可能性があります。



ノードとポッドの一意のネットワーク

セキュリティ上の理由からノードとポッドを特定のネットワークに分離する必要がある場合は、より大きなセカンダリ CIDR ブロック (100.64.0.0/8 など) からサブネットにノードとポッドをデプロイすることをお勧めします。VPC に新しい CIDR をインストールした後、セカンダリ CIDR を使用して別のノードグループをデプロイし、元のノードをドレインして、ポッドを新しいワーカーノードに自動的に再デプロイできます。これを実装する方法の詳細については、このブログ記事を参照してください。

カスタムネットワークは、次の図に示すセットアップでは使用されません。代わりに、Kubernetes ワーカーノードは、100.64.0.0/10 など、VPC のセカンダリ VPC CIDR 範囲のサブネットにデプロイされます。EKS クラスターの実行は維持できますが (コントロールプレーンは元のサブネットに残ります) 、ノードとポッドはセカンダリサブネットに移動されます。これは、VPC で IP が枯渇するリスクを軽減するための、従来の手法ではありませんが、もう 1 つの手法です。ポッドを新しいワーカーノードに再デプロイする前に、古いノードをドレインすることをお勧めします。



アベイラビリティーゾーンラベルを使用して設定を自動化する

Kubernetes を有効にして、ワーカーノードのアベイラビリティーゾーン (AZ) に対応する ENIConfig を自動的に適用できます。

Kubernetes はワーカーノード[topology.kubernetes.io/zone](topology.kubernetes.io/zone)に タグを自動的に追加します。Amazon EKS では、AZ ごとにセカンダリサブネット (代替 CIDR) が 1 つしかない場合、ENI 設定名としてアベイラビリティーゾーンを使用することをお勧めします。その後、ENI 設定名を検出するために使用されるラベルを に設定できますtopology.kubernetes.io/zone。タグfailure-domain.beta.kubernetes.io/zoneは廃止され、タグ に置き換えられることに注意してくださいtopology.kubernetes.io/zone。

1. name フィールドを VPC のアベイラビリティーゾーンに設定します。
2. 次のコマンドを使用して自動設定を有効にする
3. 次のコマンドを使用して設定ラベルを設定します。

```
kubectl set env daemonset aws-node -n kube-system
 "AWS_VPC_K8S_CNI_CUSTOM_NETWORK_CFG=true"
kubectl set env daemonset aws-node -n kube-system
 "ENI_CONFIG_LABEL_DEF=topology.kubernetes.io/zone"
```

アベイラビリティーゾーンごとに複数のセカンダリサブネットがある場合は、特定の を作成する必要がありますENI_CONFIG_LABEL_DEF。k8s.amazonaws.com/eniConfig=us-west-2a-subnet-1 を ENI_CONFIG_LABEL_DEFとして設定k8s.amazonaws.com/eniConfigし、や などのカスタム eniConfig 名を使用してノードにラベルを付けることを検討してくださいk8s.amazonaws.com/eniConfig=us-west-2a-subnet-2。

セカンダリネットワークの設定時にポッドを置き換える

カスタムネットワークを有効にしても、既存のノードは変更されません。カスタムネットワーキングは破壊的なアクションです。カスタムネットワーキングを有効にした後にクラスター内のすべてのワーカーノードをローリング置換するのではなく、EKS 入門ガイドの AWS CloudFormation テンプレートを、Lambda 関数を呼び出して環境変数で aws-node Daemonset を更新し、ワーカーノードがプロビジョニングされる前にカスタムネットワーキングを有効にするカスタムリソースで更新することをお勧めします。

カスタム CNI ネットワーク機能に切り替える前に、クラスター内に実行中の Pod を持つノードがある場合は、ノードをコーディングしてドレインし、Pod を適切にシャットダウンしてからノードを終了する必要があります。ENIConfig ラベルまたは注釈に一致する新しいノードのみがカスタムネットワークを使用するため、これらの新しいノードでスケジュールされた Pod にセカンダリ CIDR の IP を割り当てることができます。

ノードあたりの最大ポッド数を計算する

ノードのプライマリ ENI は Pod IP アドレスの割り当てに使用されなくなったため、特定の EC2 インスタンスタイプで実行できる Pod の数が減少します。この制限を回避するには、カスタムネットワーキングでプレフィックス割り当てを使用できます。プレフィックスを割り当てると、各セカンダリ IP はセカンダリ ENIs。

カスタムネットワーキングを使用する m5.large インスタンスの Pod の最大数を検討してください。

プレフィックスの割り当てなしで実行できるポッドの最大数は 29 です。

- `3 ENIs - 1) * (10 secondary IPs per ENI - 1 + 2 = 20`

プレフィックスアタッチメントを有効にすると、ポッドの数は 290 に増加します。

- `(3 ENIs - 1) * ((10 secondary IPs per ENI - 1) * 16 + 2 = 290`

ただし、インスタンスの仮想 CPUs の数はかなり少ないため、max-pods を 290 ではなく 110 に設定することをお勧めします。より大きなインスタンスでは、EKS は最大ポッド値を 250 にすることをお勧めします。より小さなインスタンスタイプ (m5.large など) のプレフィックスアタッチメントを使用する場合、インスタンスの CPU リソースとメモリリソースを IP アドレスのかなり前に使い果たす可能性があります。

> **Note**
>
> CNI プレフィックスが /28 プレフィックスを ENI に割り当てる場合、IP アドレスの連続したブロックである必要があります。プレフィックスの生成元のサブネットが高度に断片化されている場合、プレフィックスアタッチメントが失敗する可能性があります。これを回避するには、クラスター用の新しい専用 VPC を作成するか、サブネットに一連の CIDR をプレフィックスアタッチメント専用に予約します。このトピックの詳細については、「サブネット CIDR 予約」を参照してください。

CG-NAT スペースの既存の使用状況を特定する

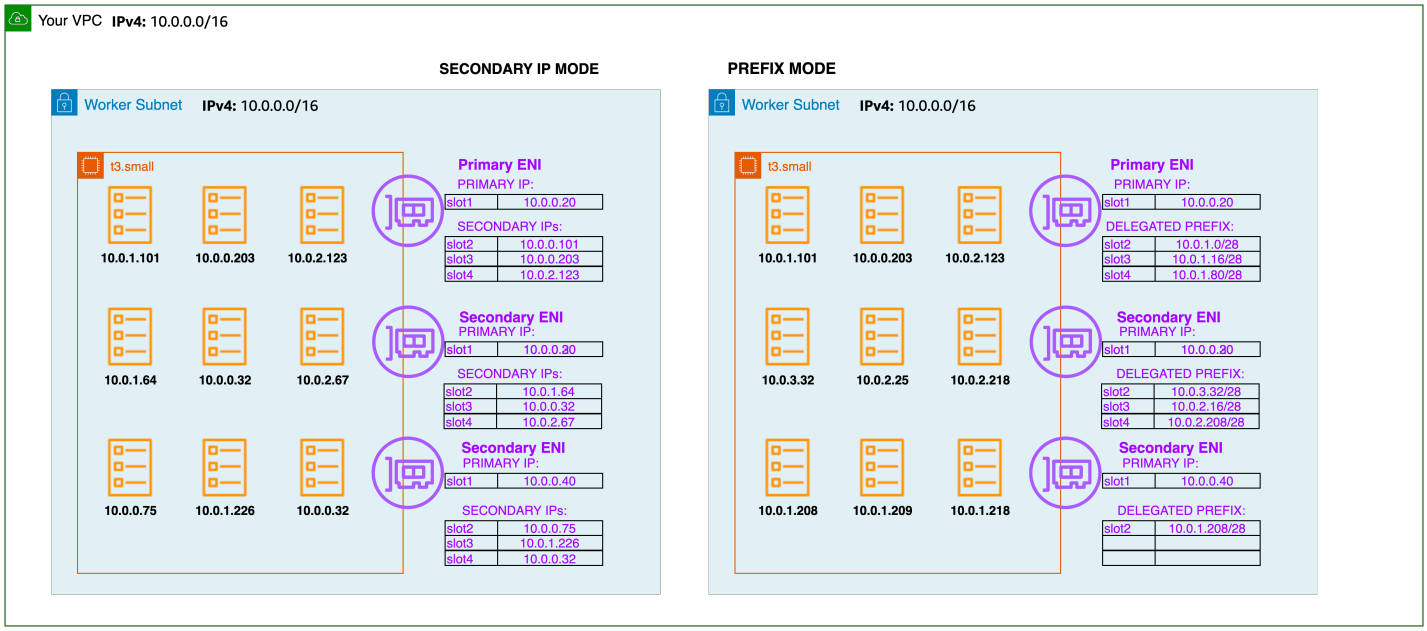
カスタムネットワークを使用すると、IP の枯渇の問題を軽減できますが、すべての課題を解決することはできません。クラスターにすでに CG-NAT スペースを使用している場合、または単にセカンダリ CIDR をクラスター VPC に関連付ける機能がない場合は、代替 CNI の使用や IPv6 クラスターへの移行など、他のオプションを検討することをお勧めします。

Linux のプレフィックスモード

Amazon VPC CNI は、Amazon EC2 ネットワークインターフェイスにネットワークプレフィックスを割り当てて、ノードで使用できる IP アドレスの数を増やし、ノードあたりのポッド密度を向上させます。個々のセカンダリ IP アドレスをネットワークインターフェイスに割り当てる代わりに、Amazon VPC CNI アドオンのバージョン 1.9.0 以降を設定して IPv4 および IPv6 CIDRs を割り当てることができます。

プレフィックスモードは IPv6 クラスターでデフォルトで有効になっており、サポートされている唯一のオプションです。VPC CNI は、ENI のスロットに /80 IPv6 プレフィックスを割り当てます。詳細については、このガイドの IPv6 セクションを参照してください。

プレフィックス割り当てモードでは、インスタンスタイプあたりの Elastic Network Interface の最大数は変わりませんが、個々の IPv4 アドレスをネットワークインターフェイスのスロットに割り当てる代わりに、/28 (16 IP アドレス) の IPv4 アドレスプレフィックスを割り当てるように Amazon VPC CNI を設定できるようになりました。ENABLE_PREFIX_DELEGATION が true に設定されている場合、VPC CNI は ENI に割り当てられたプレフィックスから Pod に IP アドレスを割り当てます。EKS ユーザーガイドに記載されている手順に従って、プレフィックス IP モードを有効にしてください。



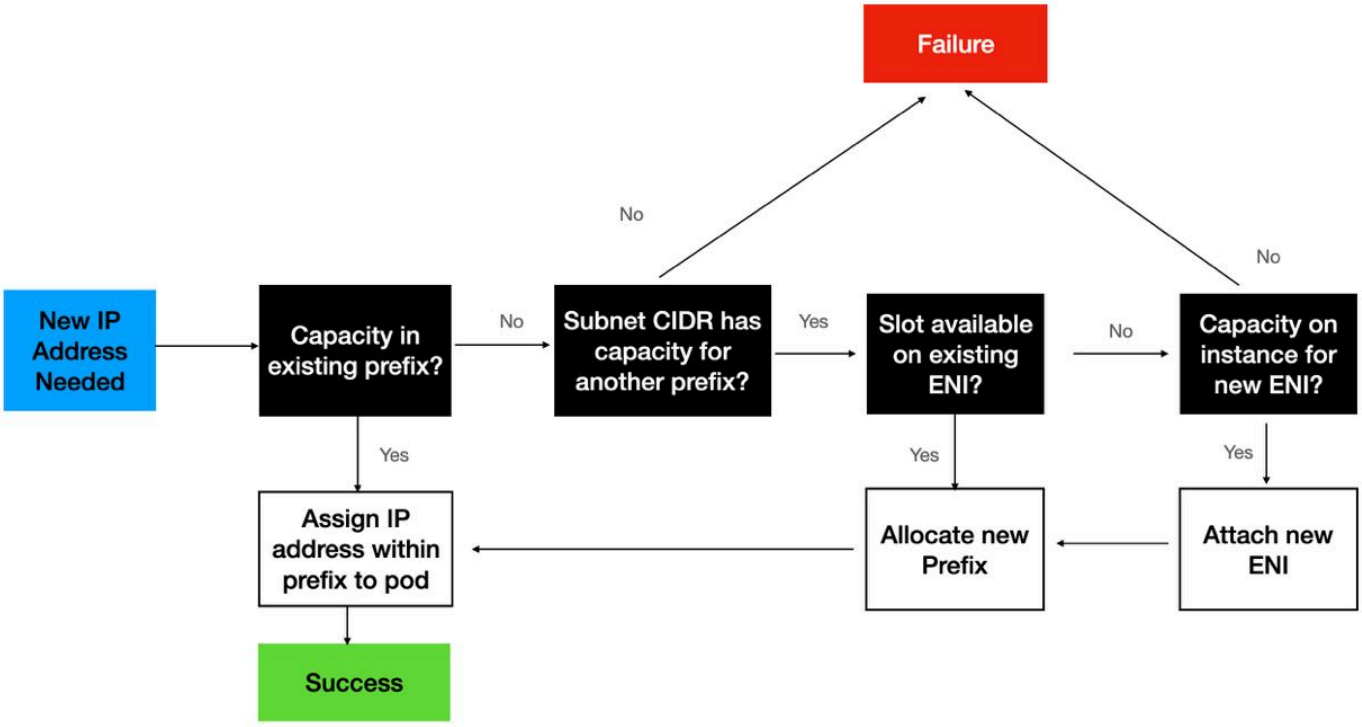
ネットワークインターフェイスに割り当てることができる IP アドレスの数はインスタンスタイプによって異なります。ネットワークインターフェイスに割り当てる各プレフィクスが、1 つの IP アドレスとしてカウントされます。例えば、`c5.large` インスタンスで、ネットワークインターフェイスあたりの IPv4 アドレス数が `10` に制限されているとします。このインスタンスの各ネットワークインターフェイスに、プライマリ IPv4 アドレスが存在します。ネットワークインターフェイスにセカンダリ IPv4 アドレスがない場合はネットワークインターフェイスに最大 9 つのプレフィクスを割り当てることができます。ネットワークインターフェイスに割り当てる IPv4 アドレスを追加する度に、ネットワークインターフェイスに割り当てるプレフィクスの数が 1 つ少なくなります。インスタンスタイプごとのネットワークインターフェイスごとの IP アドレスと、ネットワークインターフェイスへのプレフィックスの割り当てに関する AWS EC2 ドキュメントを確認してください。
https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ec2-prefix-eni.html

ワーカーノードの初期化中、VPC CNI はプライマリ ENI に 1 つ以上のプレフィックスを割り当てます。CNI は、ウォームプールを維持することで、ポッドの起動を高速化するためのプレフィックスを

事前に割り当てます。ウォームプールに保持されるプレフィックスの数は、環境変数を設定することで制御できます。

- WARM_PREFIX_TARGET、現在のニーズを超えて割り当てられるプレフィックスの数。
- WARM_IP_TARGET、現在のニーズを超えて割り当てられる IP アドレスの数。
- MINIMUM_IP_TARGET: いつでも使用できる IP アドレスの最小数。
- WARM_IP_TARGET と MINIMUM_IP_TARGET を設定すると、 が上書きされますWARM_PREFIX_TARGET。

スケジュールされたポッドが増えると、既存の ENI に追加のプレフィックスがリクエストされます。まず、VPC CNI は既存の ENI に新しいプレフィックスの割り当てを試みます。ENI が容量にある場合、VPC CNI は新しい ENI をノードに割り当てようとします。新しい ENIsは、最大 ENI 制限 (インスタンスタイプで定義) に達するまでアタッチされます。新しい ENI がアタッチされると、ipamd は WARM_PREFIX_TARGET、、WARM_IP_TARGETおよび MINIMUM_IP_TARGET設定を維持するために必要な 1 つ以上のプレフィックスを割り当てます。



推奨事項

プレフィックスモードを使用する

ワーカーノードで Pod 密度の問題が発生した場合は、プレフィックスモードを使用します。VPC CNI エラーを回避するには、プレフィックスモードに移行する前に、サブネットで /28 プレフィックスの連続するアドレスブロックを調べることをお勧めします。サブネット予約の詳細については、「サブネット予約を使用してサブネットの断片化 (IPv4) を回避する」セクションを参照してください。

下位互換性のために、最大ポッド制限はセカンダリ IP モードをサポートするように設定されています。ポッド密度を増やすには、Kubelet にmax-pods値を指定し、ノードのユーザーデータ--use-max-pods=falseとして を指定します。max-pod-calculator.sh スクリプトを使用して、特定のインスタンスタイプで EKS が推奨するポッドの最大数を計算することを検討できます。ユーザーデータの例については、EKS ユーザーガイドを参照してください。

```
./max-pods-calculator.sh --instance-type m5.large --cni-version ``1.9``.0 --cni-prefix-delegation-enabled
```

プレフィックス割り当てモードは、プライマリ ENI がポッドに使用されていない CNI カスタムネットワーキングのユーザーにとって特に重要です。プレフィックスを割り当てることで、ポッドに使用されるプライマリ ENI がない場合でも、ほぼすべての Nitro インスタンスタイプにより多くの IPs をアタッチできます。

プレフィックスモードを回避する

サブネットが非常にフラグメント化されていて、/28 プレフィックスを作成するための使用可能な IP アドレスが不十分な場合は、プレフィックスモードを使用しないでください。プレフィックスが生成されるサブネット (分散セカンダリ IP アドレスを持つ頻繁に使用されるサブネット) がフラグメント化されている場合、プレフィックスアタッチメントが失敗することがあります。この問題は、新しいサブネットを作成し、プレフィックスを予約することで回避できます。

プレフィックスモードでは、ワーカーノードに割り当てられたセキュリティグループは Pod によって共有されます。共有コンピューティングリソースでさまざまなネットワークセキュリティ要件を持つアプリケーションを実行してコンプライアンスを達成するためのセキュリティ要件がある場合は、Pod のセキュリティグループの使用を検討してください。

同じノードグループで同様のインスタンスタイプを使用する

ノードグループには、多くのタイプのインスタンスが含まれている場合があります。インスタンスの最大ポッド数が少ない場合、その値はノードグループ内のすべてのノードに適用されます。ノードグループで同様のインスタンスタイプを使用して、ノードの使用を最大化することを検討してください。自動ノードスケーリングに Karpenter を使用している場合は、プロビジョナー API の要件部分で node.kubernetes.io/instance-type を設定することをお勧めします。

> **Warning**
>
> 特定のノードグループ内のすべてのノードの最大ポッド数は、ノードグループ内の任意の 1 つのインスタンスタイプの最小最大ポッド数によって定義されます。

IPv4 アドレスを節約WARM_PREFIX_TARGETするように を設定する

のインストールマニフェストのデフォルト値WARM_PREFIX_TARGETは 1 です。ほとんどの場合、の推奨値 1 WARM_PREFIX_TARGETは、インスタンスに割り当てられた未使用の IP アドレスを最小限に抑えながら、ポッドの高速起動時間を適切に組み合わせます。

ノードごとの IPv4 アドレスの使用WARM_IP_TARGETとMINIMUM_IP_TARGET設定をさらに節約する必要がある場合は、設定WARM_PREFIX_TARGET時に上書きされます。WARM_IP_TARGET を 16 未満の値に設定することで、CNI が余分なプレフィックス全体をアタッチしないようにできます。

新しい ENI をアタッチするよりも新しいプレフィックスを割り当てることを優先する

既存の ENI に追加のプレフィックスを割り当てることは、新しい ENI を作成してインスタンスにアタッチするよりも高速な EC2 API オペレーションです。プレフィックスを使用すると、IPv4 アドレス割り当てが不足している間のパフォーマンスが向上します。プレフィックスのアタッチは通常 1 秒未満で完了しますが、新しい ENI のアタッチには最大 10 秒かかる場合があります。ほとんどのユースケースでは、CNI はプレフィックスモードで実行する場合、ワーカーノードごとに 1 つの ENI のみを必要とします。ノードあたり最大 15 IPs を (最悪の場合) 確保できる場合は、新しいプレフィックス割り当てネットワークモードを使用し、それに伴うパフォーマンスと効率の向上を実現することを強くお勧めします。

サブネット予約を使用してサブネットの断片化を回避する (IPv4)

EC2 が /28 IPv4 プレフィックスを ENI に割り当てる場合、サブネットからの IP アドレスの連続したブロックである必要があります。プレフィックスが生成されるサブネットがフラグメント化されて

いる場合 (分散セカンダリ IP アドレスを持つ高度に使用されているサブネット) 、プレフィックスアタッチメントが失敗し、VPC CNI ログに次のエラーメッセージが表示されます。

```
failed to allocate a private IP/Prefix address: InsufficientCidrBlocks: There are not
 enough free cidr blocks in the specified subnet to satisfy the request.
```

フラグメント化を回避し、プレフィックスを作成するのに十分な連続スペースを確保するために、VPC サブネット CIDR 予約を使用して、サブネット内の IP スペースを予約し、プレフィックスによって排他的に使用することができます。予約を作成すると、VPC CNI プラグインは EC2 APIsを呼び出して、予約済みスペースから自動的に割り当てられるプレフィックスを割り当てます。

新しいサブネットを作成し、プレフィックスのスペースを予約し、そのサブネットで実行されているワーカーノードの VPC CNI でプレフィックス割り当てを有効にすることをお勧めします。新しいサブネットが VPC CNI プレフィックス割り当てが有効になっている EKS クラスターで実行されている Pod 専用である場合は、プレフィックス予約ステップをスキップできます。

VPC CNI のダウングレードを避ける

プレフィックスモードは、VPC CNI バージョン 1.9.0 以降で動作します。プレフィックスモードが有効になり、プレフィックスが ENIs に割り当てられたら、Amazon VPC CNI アドオンを 1.9.0 より前のバージョンにダウングレードすることは避ける必要があります。VPC CNI をダウングレードする場合は、ノードを削除して再作成する必要があります。

プレフィックス委任への移行中にすべてのノードを置き換える

既存のワーカーノードをローリング置換するのではなく、使用可能な IP アドレスの数を増やすために新しいノードグループを作成することを強くお勧めします。既存のすべてのノードを遮断およびドレインして、既存のすべての Pod を安全に削除します。サービスの中断を防ぐために、重要なワークロードの本番稼働用クラスターに Pod Disruption Budgets を実装することをお勧めします。新しいノードのポッドには、ENI に割り当てられたプレフィックスから IP が割り当てられます。Pod が実行されていることを確認したら、古いノードとノードグループを削除できます。マネージド型ノードグループを使用している場合は、ここに記載されている手順に従ってノードグループを安全に削除してください。

Windows のプレフィックスモード

Amazon EKS では、Windows ホストで実行される各 Pod には、デフォルトで VPC リソースコントローラーによってセカンダリ IP アドレスが割り当てられます。この IP アドレスは、ホストのサブネットから割り当てられた VPC ルーティング可能なアドレスです。Linux では、インスタンスにア

タッチされた各 ENI には、セカンダリ IP アドレスまたは /28 CIDR (プレフィックス) で入力できる複数のスロットがあります。ただし、Windows ホストは 1 つの ENI とその使用可能なスロットのみをサポートします。セカンダリ IP アドレスのみを使用すると、割り当て可能な IP アドレスが多数ある場合でも、Windows ホストで実行できるポッドの数を実質的に制限できます。

Windows ホストのポッド密度を高めるために、特に小さなインスタンスタイプを使用する場合は、Windows ノードのプレフィックス委任を有効にできます。プレフィックス委任が有効になっている場合、/28 IPv4 プレフィックスはセカンダリ IP アドレスではなく ENI スロットに割り当てられます。プレフィックスの委任を有効にするには、設定マップに `amazon-vpc-cni enable-windows-prefix-delegation: "true"`エントリを追加します。これは、Windows サポートを有効にするために`enable-windows-ipam: "true"`エントリを設定する必要がある設定マップと同じです。

EKS ユーザーガイドに記載されている手順に従って、Windows ノードのプレフィックス委任モードを有効にしてください。

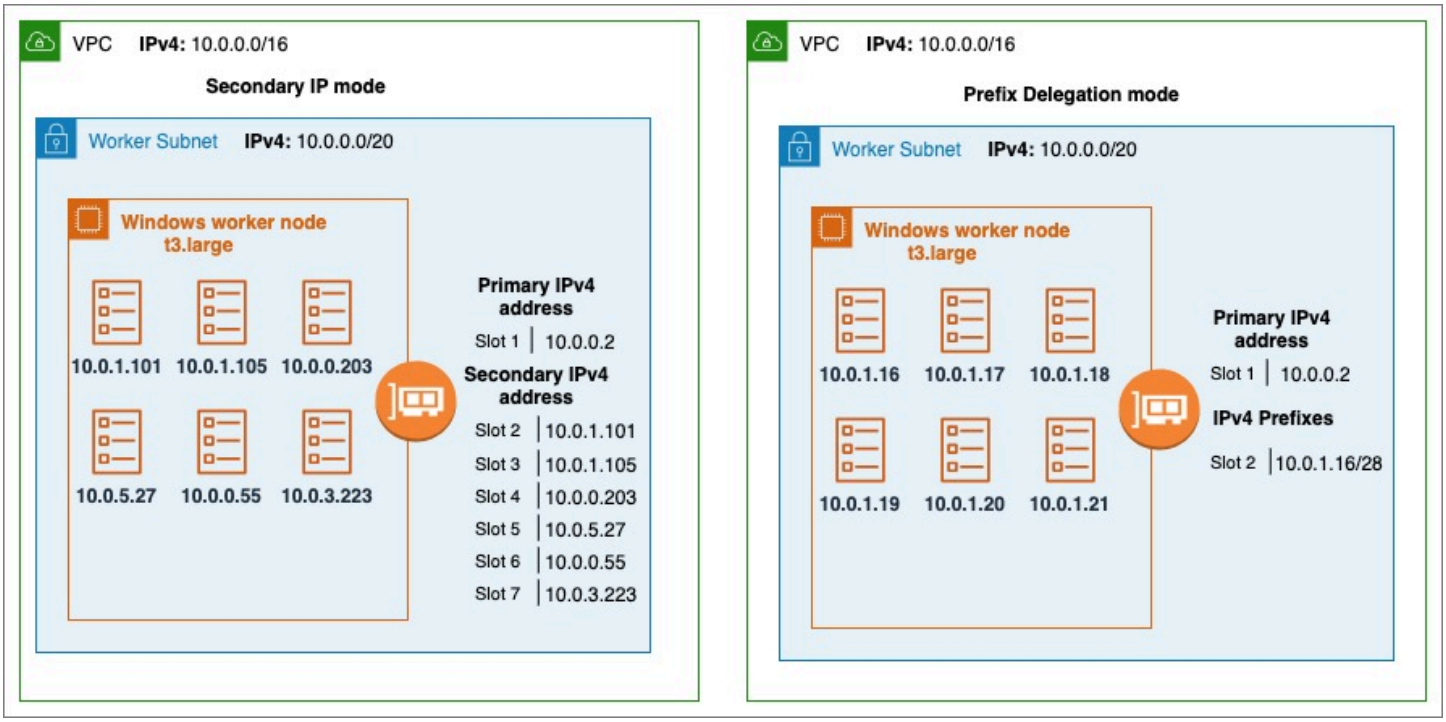


図: セカンダリ IP モードとプレフィックス委任モードの比較

ネットワークインターフェイスに割り当てることができる IP アドレスの最大数は、インスタンスタイプとそのサイズによって異なります。ネットワークインターフェイスに割り当てられた各プレフィックスは、使用可能なスロットを消費します。たとえば、`c5.large`インスタンスにはネットワークインターフェイスあたりの10スロット数の制限があります。ネットワークインターフェイスの最初のスロットは常にインターフェイスのプライマリ IP アドレスによって消費され、プレフィッ

クスやセカンダリ IP アドレス用の 9 つのスロットが残ります。これらのスロットにプレフィックスが割り当てられている場合、ノードは (9 * 16) 144 個の IP アドレスをサポートできますが、セカンダリ IP アドレスが割り当てられている場合、サポートできる IP アドレスは 9 個のみです。詳細については、インスタンスタイプごとのネットワークインターフェイスごとの IP アドレスと、ネットワークインターフェイスへのプレフィックスの割り当てに関するドキュメントを参照してください。

ワーカーノードの初期化中、VPC リソースコントローラーは IP アドレスのウォームプールを維持することで、ポッドの起動を高速化するためにプライマリ ENI に 1 つ以上のプレフィックスを割り当てます。ウォームプールに保持されるプレフィックスの数は、`amazon-vpc-cni`設定マップで次の設定パラメータを設定することで制御できます。

- `warm-prefix-target`、現在のニーズを超えて割り当てられるプレフィックスの数。
- `warm-ip-target`、現在のニーズを超えて割り当てられる IP アドレスの数。
- `minimum-ip-target`: いつでも使用できる IP アドレスの最小数。
- `warm-ip-target` および/または`minimum-ip-target`、設定されている場合は が上書きされます`warm-prefix-target`。

ノードでより多くの Pod がスケジュールされると、既存の ENI に追加のプレフィックスがリクエストされます。Pod がノードでスケジュールされている場合、VPC Resource Controller はまずノードの既存のプレフィックスから IPv4 アドレスを割り当てようとします。これが不可能な場合は、サブネットに必要な容量がある限り、新しい IPv4 プレフィックスがリクエストされます。

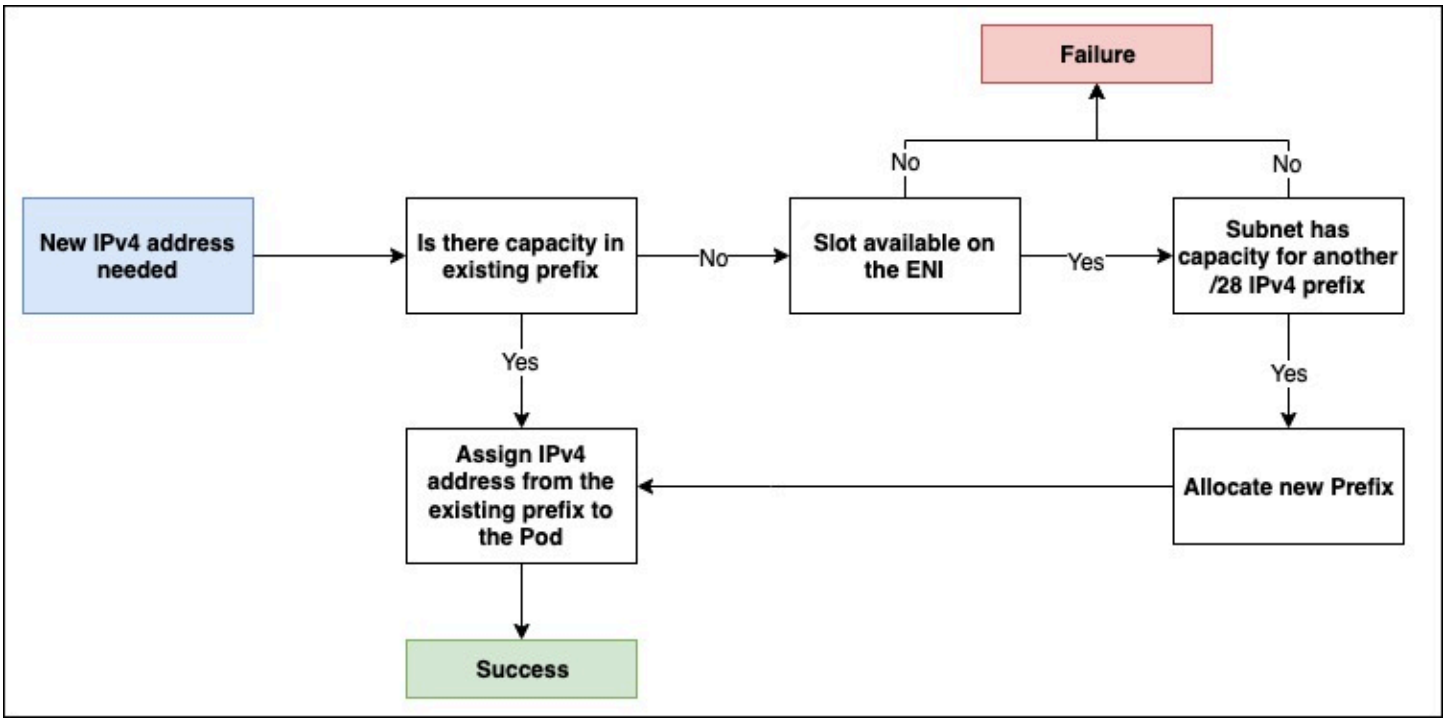


図: Pod への IPv4 アドレスの割り当て時のワークフロー

推奨事項

プレフィックスの委任を使用する

ワーカーノードで Pod 密度の問題が発生した場合は、プレフィックス委任を使用します。エラーを回避するには、プレフィックスモードに移行する前に、サブネットで /28 プレフィックスの連続したアドレスブロックを調べることをお勧めします。サブネット予約の詳細については、「サブネット予約を使用してサブネットの断片化 (IPv4) を回避する」セクションを参照してください。

デフォルトでは、Windows ノードmax-podsの は に設定されます110。ほとんどのインスタンスタイプでは、これで十分です。この制限を増減する場合は、ユーザーデータのブートストラップコマンドに以下を追加します。

```
-KubeletExtraArgs '--max-pods=example-value'
```

Windows ノードのブートストラップ設定パラメータの詳細については、こちらのドキュメントを参照してください。

プレフィックスの委任を避ける

サブネットが非常にフラグメント化されていて、/28 プレフィックスを作成するための使用可能な IP アドレスが不十分な場合は、プレフィックスモードを使用しないでください。プレフィックスが生成されるサブネット (分散セカンダリ IP アドレスを持つ頻繁に使用されるサブネット) がフラグメント化されている場合、プレフィックスアタッチメントが失敗することがあります。この問題は、新しいサブネットを作成し、プレフィックスを予約することで回避できます。

IPv4 アドレスを節約するためにプレフィックス委任のパラメータを設定する

warm-prefix-target、warm-ip-target、および を使用して、プリスケーリングと動的スケーリングの動作をプレフィックスで微調整minimum-ip-targetできます。デフォルトでは、次の値が使用されます。

```
warm-ip-target: "1"
minimum-ip-target: "3"
```

これらの設定パラメータを微調整することで、IP アドレスを節約し、IP アドレスの割り当てによる Pod レイテンシーを低減する最適なバランスを実現できます。これらの設定パラメータの詳細については、こちらのドキュメントを参照してください。

サブネット予約を使用してサブネットの断片化を回避する (IPv4)

EC2 が /28 IPv4 プレフィックスを ENI に割り当てる場合、サブネットからの IP アドレスの連続したブロックである必要があります。プレフィックスが生成されるサブネットがフラグメント化されている場合 (分散セカンダリ IP アドレスを持つ高度に使用されているサブネット) 、プレフィックスアタッチメントが失敗する可能性があり、次のノードイベントが表示されます。

```
InsufficientCidrBlocks: The specified subnet does not have enough free cidr blocks to
 satisfy the request
```

フラグメント化を回避し、プレフィックスを作成するための十分な連続スペースを確保するには、VPC サブネット CIDR 予約を使用して、サブネット内の IP スペースを予約し、プレフィックスによって排他的に使用できるようにします。予約を作成すると、リザーブドブロックの IP アドレスは他のリソースに割り当てられません。これにより、VPC Resource Controller はノード ENI への割り当て呼び出し中に使用可能なプレフィックスを取得できます。

新しいサブネットを作成し、プレフィックスのスペースを予約し、そのサブネットで実行されているワーカーノードのプレフィックス割り当てを有効にすることをお勧めします。新しいサブネットが、プレフィックスの委任が有効になっている EKS クラスターで実行されている Pod 専用である場合は、プレフィックスの予約ステップをスキップできます。

セカンダリ IP モードからプレフィックス委任モードに移行するとき、またはその逆に移行するときにすべてのノードを置き換える

既存のワーカーノードをローリング置換するのではなく、使用可能な IP アドレスの数を増やすために新しいノードグループを作成することを強くお勧めします。

セルフマネージド型ノードグループを使用する場合、移行のステップは次のとおりです。

- 新しいノードがワークロードに対応できるようにクラスターの容量を増やす
- Windows のプレフィックス委任機能の有効化/無効化
- 既存のすべてのノードを遮断してドレインし、既存のすべての Pod を安全に削除します。サービスの中断を防ぐために、重要なワークロードの本番稼働用クラスターに Pod Disruption Budgets を実装することをお勧めします。
- Pod が実行されていることを確認したら、古いノードとノードグループを削除できます。新しいノードのポッドには、ノード ENI に割り当てられたプレフィックスから IPv4 アドレスが割り当てられます。

マネージド型ノードグループを使用する場合、移行のステップは次のとおりです。

- Windows のプレフィックス委任機能の有効化/無効化
- ここで説明するステップを使用して、ノードグループを更新します。これは上記と同様のステップを実行しますが、EKS によって管理されます。

> ⚠ Warning
>
> ノード上のすべての Pod を同じモードで実行する

Windows では、セカンダリ IP モードとプレフィックス委任モードの両方で Pod を同時に実行しないことをお勧めします。このような状況は、セカンダリ IP モードからプレフィックス委任モードに移行するか、実行中の Windows ワークロードでその逆に移行すると発生する可能性があります。

これは実行中のポッドには影響しませんが、ノードの IP アドレス容量に関して不整合が生じる可能性があります。たとえば、セカンダリ IPv4 アドレス用に 14 個のスロットを持つ t3.xlarge ノードがあるとします。10 個の Pod を実行している場合、ENI の 10 個のスロットがセカンダリ IP アドレスによって消費されます。プレフィックスの委任を有効にすると、kube-api サーバーにアドバタイズされる容量は (14 スロット x プレフィックスあたり 16 ip アドレス) 244 ですが、その時点での実際の容量は (残りの 4 スロット x プレフィックスあたり 16 アドレス) 64 になります。アドバタイズされる容量と実際の容量 (残りのスロット) の間にこの不整合があると、割り当て可能な IP アドレスよりも多くの Pod を実行すると、問題が発生する可能性があります。

つまり、上記の移行戦略を使用して、ポッドをセカンダリ IP アドレスからプレフィックスから取得したアドレスに安全に移行できます。モードを切り替えると、Pod は通常どおり実行され続けます。

- セカンダリ IP モードからプレフィックス委任モードに切り替える場合、実行中のポッドに割り当てられたセカンダリ IP アドレスは解放されません。プレフィックスは空きスロットに割り当てられます。ポッドが終了すると、使用していたセカンダリ IP とスロットが解放されます。
- プレフィックス委任モードからセカンダリ IP モードに切り替えると、範囲内のすべての IPs がポッドに割り当てられなくなると、プレフィックスが解放されます。プレフィックスの IP がポッドに割り当てられている場合、そのプレフィックスはポッドが終了するまで保持されます。

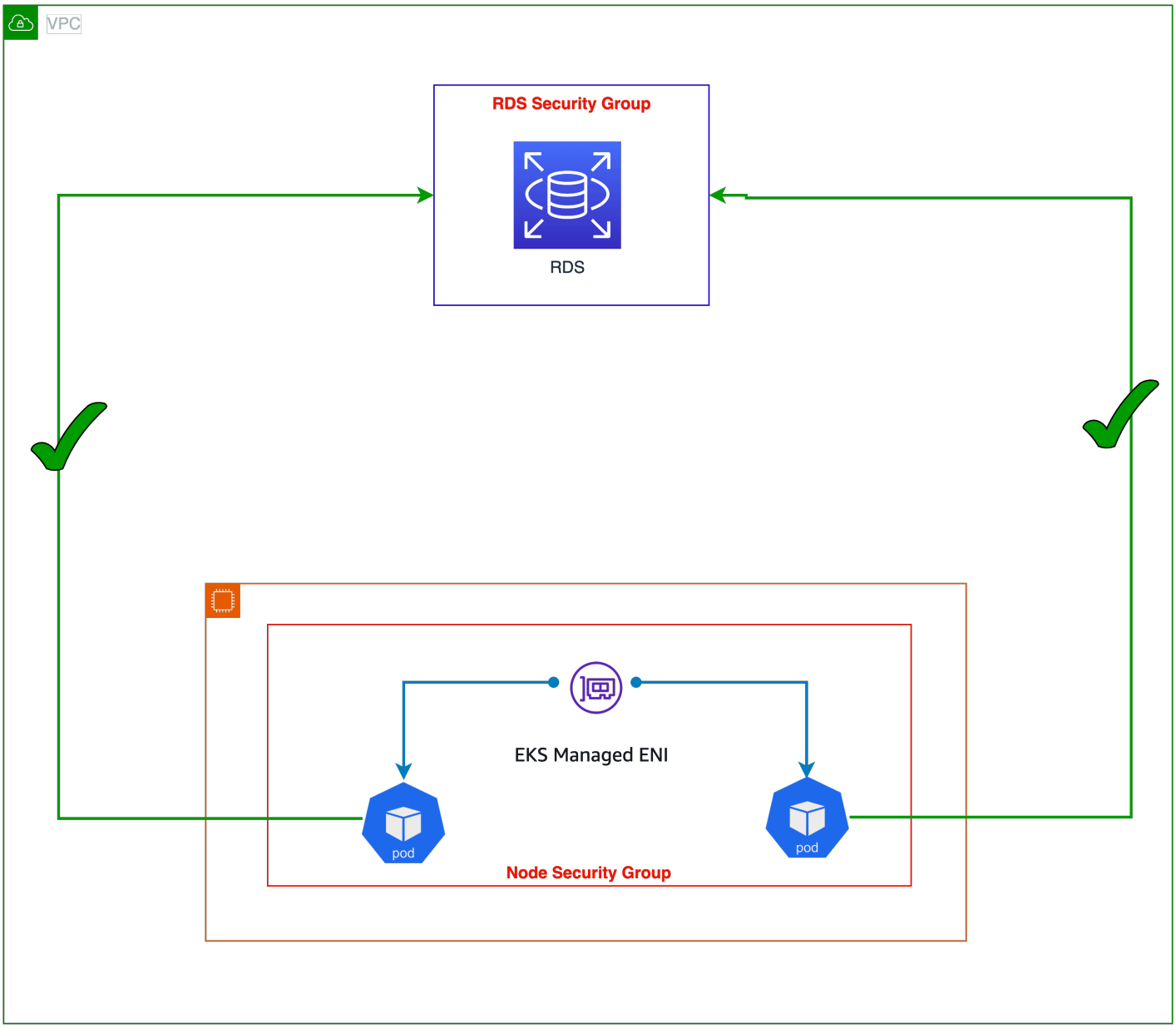
プレフィックスの委任による問題のデバッグ

ここでデバッグガイドを使用して、Windows でプレフィックスの委任で直面している問題を詳しく調べることができます。

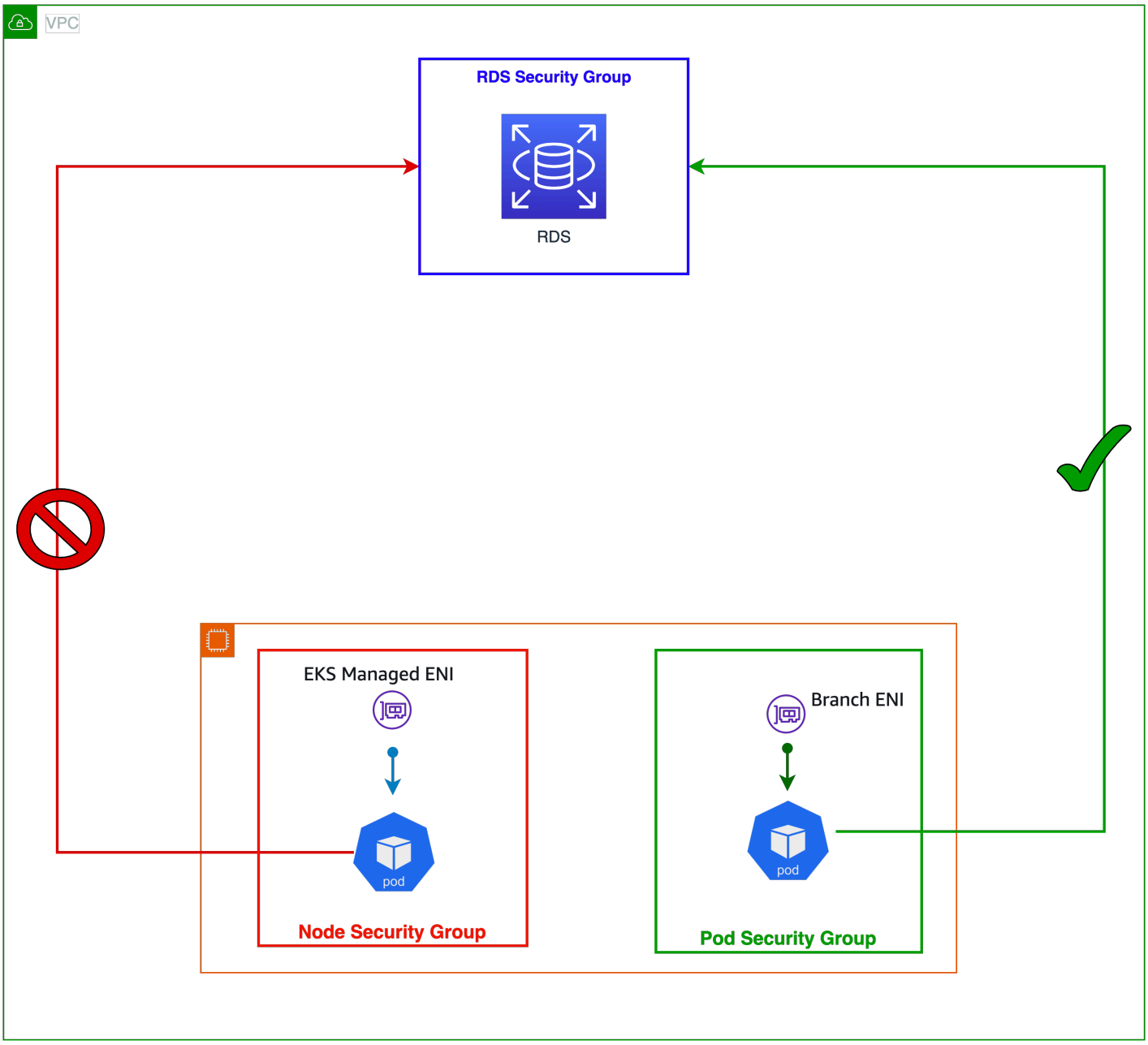
ポッドあたりのセキュリティグループ

AWS セキュリティグループは、EC2 インスタンスの仮想ファイアウォールとして機能し、インバウンドトラフィックとアウトバウンドトラフィックを制御します。デフォルトでは、Amazon VPC CNI はノード上のプライマリ ENI に関連付けられたセキュリティグループを使用します。具体的には、インスタンスに関連付けられているすべての ENI に同じ EC2 セキュリティグループがあります。したがって、ノード上のすべての Pod は、実行されるノードと同じセキュリティグループを共有します。

次の図に示すように、ワーカーノードで動作しているすべてのアプリケーションポッドは RDS データベースサービスにアクセスできます (RDS インバウンドではノードセキュリティグループが許可されます) 。セキュリティグループは、ノードで実行されているすべての Pod に適用されるため、粗すぎます。Pod のセキュリティグループは、ワークロードのネットワークセグメンテーションを提供します。これは、多層防御戦略にとって不可欠な部分です。

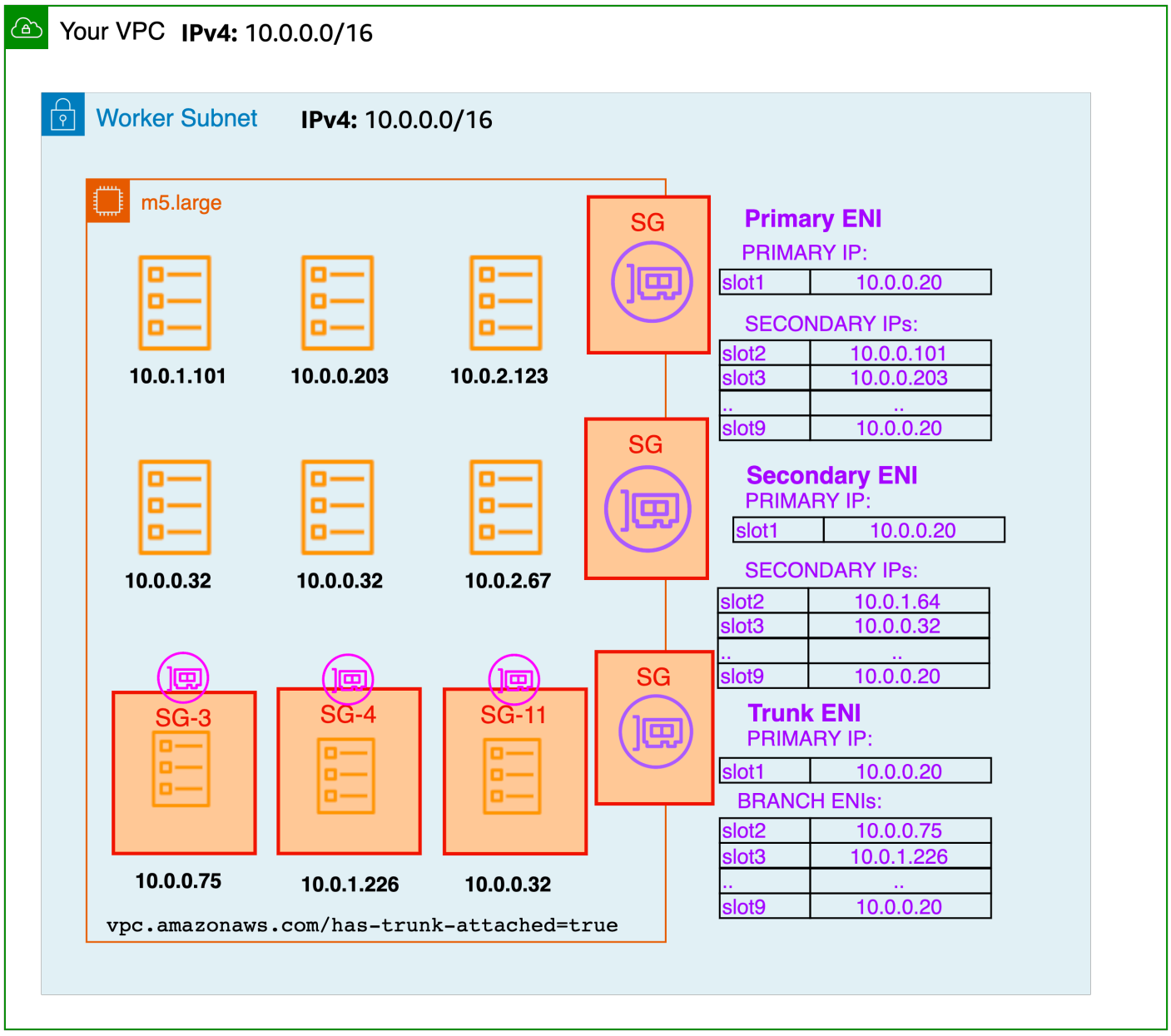


Pod のセキュリティグループを使用すると、共有コンピューティングリソースでさまざまなネットワークセキュリティ要件を持つアプリケーションを実行することで、コンピューティング効率を向上させることができます。Pod-to-Pod や Pod-to-External AWS サービスなど、複数のタイプのセキュリティルールを EC2 セキュリティグループで 1 か所で定義し、Kubernetes ネイティブ APIs を使用するワークロードに適用できます。以下の図は、Pod レベルで適用されるセキュリティグループと、アプリケーションのデプロイとノードアーキテクチャを簡素化する方法を示しています。Pod が Amazon RDS データベースにアクセスできるようになりました。



ENABLE_POD_ENI=true VPC CNI の を設定することで、Pod のセキュリティグループを有効にできます。有効にすると、コントロールプレーン (EKS によって管理) で実行されている VPC リソースコントローラーは、「aws-k8s-trunk-eni」というトランクインターフェイスを作成してノードにアタッチします。トランクインターフェイスは、インスタンスにアタッチされた標準ネットワークインターフェイスとして機能します。トランクインターフェイスを管理するには、Amazon EKS クラスターに含まれるクラスターロールに AmazonEKSVPCResourceController管理ポリシーを追加する必要があります。

また、コントローラーは「aws-k8s-branch-eni」という名前のブランチインターフェイスを作成し、トランクインターフェイスに関連付けます。ポッドには SecurityGroupPolicy カスタムリソースを使用してセキュリティグループが割り当てられ、ブランチインターフェイスに関連付けられます。セキュリティグループはネットワークインターフェイスで指定されているため、これらの追加のネットワークインターフェイスで特定のセキュリティグループを必要とするポッドをスケジュールできるようになりました。デプロイの前提条件を含め、ポッドのセキュリティグループに関する EKS ユーザーガイドセクションを確認してください。



ブランチインターフェイス容量は、セカンダリ IP アドレスの既存のインスタンスタイプの制限に追加されます。セキュリティグループを使用するポッドは、max-pods 式では考慮されません。ポッド

にセキュリティグループを使用する場合は、max-pods 値を上げることを検討するか、ノードが実際にサポートできる数よりも少ないポッドの実行で問題ありません。

m5.large には、最大 9 つのブランチネットワークインターフェイスと最大 27 のセカンダリ IP アドレスを標準ネットワークインターフェイスに割り当てることができます。以下の例に示すように、m5.large のデフォルトの最大ポッドは 29 で、EKS はセキュリティグループを使用するポッドを最大ポッドにカウントします。ノードの最大ポッドを変更する方法については、EKS ユーザーガイドを参照してください。

Pod のセキュリティグループをカスタムネットワークと組み合わせて使用する場合、ポッドのセキュリティグループで定義されたセキュリティグループは、ENIConfig で指定されたセキュリティグループではなく使用されます。そのため、カスタムネットワーキングが有効になっている場合は、Pod ごとにセキュリティグループを使用しながら、セキュリティグループの順序を慎重に評価してください。

推奨事項

Liveness Probe の TCP Early Demux を無効にする

ライブネスプローブまたは準備状況プローブを使用している場合は、kubelet が TCP 経由でブランチネットワークインターフェイス上の Pod に接続できるように、TCP 早期デマックスを無効にする必要もあります。これは strict モードでのみ必要です。これを行うには、次のコマンドを実行します。

```
kubectl edit daemonset aws-node -n kube-system
```

`initContainer` セクションで、 の値を `DISABLE_TCP_EARLY_DEMUX` に変更します。 `true`.

Security Group For Pods を使用して、既存の AWS 設定投資を活用します。

セキュリティグループを使用すると、RDS データベースや EC2 インスタンスなどの VPC リソースへのネットワークアクセスを簡単に制限できます。Pod あたりのセキュリティグループの明らかな利点の 1 つは、既存の AWS セキュリティグループリソースを再利用できることです。セキュリティグループをネットワークファイアウォールとして使用して AWS サービスへのアクセスを制限する場合は、ブランチ ENIs を使用してセキュリティグループを Pod に適用することをお勧めします。EC2 インスタンスから EKS にアプリケーションを転送し、セキュリティグループを使用して他の AWS サービスへのアクセスを制限する場合は、Pod のセキュリティグループの使用を検討してください。

ポッドセキュリティグループの強制モードを設定する

Amazon VPC CNI プラグインバージョン 1.11 に、 POD_SECURITY_GROUP_ENFORCING_MODE (「強制モード」) という名前の新しい設定が追加されました。強制モードは、ポッドに適用するセキュリティグループと、ソース NAT が有効になっているかどうかの両方を制御します。強制モードは、厳格または標準として指定できます。Strict はデフォルトであり、 を ENABLE_POD_ENIに設定した場合の VPC CNI の以前の動作を反映していますtrue。

Strict Mode では、ブランチ ENI セキュリティグループのみが強制されます。ソース NAT も無効になっています。

標準モードでは、プライマリ ENI とブランチ ENI (ポッドに関連付けられている) の両方に関連付けられたセキュリティグループが適用されます。ネットワークトラフィックは、両方のセキュリティグループに準拠する必要があります。

> **Warning**
>
> モードの変更は、新しく起動された Pod にのみ影響します。既存のポッドは、ポッドの作成時に設定されたモードを使用します。トラフィック動作を変更する場合は、セキュリティグループを持つ既存の Pod をリサイクルする必要があります。

強制モード: ポッドとノードのトラフィックを分離するには、厳密なモードを使用します。

デフォルトでは、Pod のセキュリティグループは「厳格モード」に設定されています。Pod トラフィックをノードの残りのトラフィックから完全に分離する必要がある場合は、この設定を使用します。厳格モードでは、ソース NAT がオフになっているため、ブランチ ENI アウトバウンドセキュリティグループを使用できます。

> **Warning**
>
> Strict モードを有効にすると、ポッドからのすべてのアウトバウンドトラフィックがノードを離れ、VPC ネットワークに入ります。同じノード上のポッド間のトラフィックは VPC を経由します。これにより、VPC トラフィックが増加し、ノードベースの機能が制限されます。NodeLocal DNSCache は strict モードではサポートされていません。

強制モード: 次の状況では標準モードを使用します。

Pod のコンテナに表示されるクライアントソース IP

Pod のコンテナでクライアントソース IP を表示したままにする必要がある場合は、`POD_SECURITY_GROUP_ENFORCING_MODE`を に設定することを検討してください`standard`。Kubernetes サービスは externalTrafficPolicy=local をサポートし、クライアントソース IP (デフォルトタイプクラスター) の保存をサポートします。標準モードで externalTrafficPolicy が Local に設定されているインスタンスターゲットを使用して、NodePort および LoadBalancer タイプの Kubernetes サービスを実行できるようになりました。 はクライアントソース IP `Local`を保持し、LoadBalancer および NodePort タイプのサービスの 2 番目のホップを回避します。

NodeLocal DNSCache のデプロイ

ポッドにセキュリティグループを使用する場合は、NodeLocal DNSCache を使用するポッドをサポートするように標準モードを設定します。NodeLocal DNSCache は、クラスターノードで DNS キャッシュエージェントを DaemonSet として実行することで、クラスター DNS のパフォーマンスを向上させます。これにより、DNS QPS 要件が最も高いポッドはローカルキャッシュを持つローカル kube-dns/CoreDNS をクエリできるため、レイテンシーが向上します。

NodeLocal DNSCache は、ノードへのすべてのネットワークトラフィックが VPC に入るため、厳密なモードではサポートされません。

Kubernetes ネットワークポリシーのサポート

セキュリティグループが関連付けられている Pod でネットワークポリシーを使用する場合は、標準強制モードを使用することをお勧めします。

ポッドのセキュリティグループを利用して、クラスターに含まれていない AWS のサービスへのネットワークレベルのアクセスを制限することを強くお勧めします。クラスター内のポッド間のネットワークトラフィックを制限するネットワークポリシーを検討してください。これは、多くの場合、東部/西部トラフィックと呼ばれます。

ポッドごとにセキュリティグループとの非互換性を特定する

Windows ベースのインスタンスと Nitro 以外のインスタンスは、Pod のセキュリティグループをサポートしていません。Pod でセキュリティグループを利用するには、インスタンスに isTrunkingEnabled というタグを付ける必要があります。Pod が VPC 内外の AWS のサービスに依存していない場合は、セキュリティグループではなくネットワークポリシーを使用して Pod 間のアクセスを管理します。

ポッドあたりのセキュリティグループを使用して AWS サービスへのトラフィックを効率的に制御する

EKS クラスター内で実行されているアプリケーションが RDS データベースなどの VPC 内の別のリソースと通信する必要がある場合は、ポッドSGs を使用することを検討してください。CIDR または DNS 名を指定できるポリシーエンジンはありますが、VPC 内に存在するエンドポイントを持つ AWS サービスと通信するときは最適ではありません。

対照的に、Kubernetes ネットワークポリシーは、クラスター内外の両方の入出力トラフィックを制御するメカニズムを提供します。アプリケーションが他の AWS のサービスへの依存関係を制限している場合は、Kubernetes ネットワークポリシーを検討する必要があります。SGs などの AWS ネイティブセマンティクスではなく、CIDR 範囲に基づいて出力ルールを指定するネットワークポリシーを設定して、AWS のサービスへのアクセスを制限できます。Kubernetes ネットワークポリシーを使用して、ポッド間 (多くの場合、東部/西部トラフィック) およびポッドと外部サービス間のネットワークトラフィックを制御できます。Kubernetes ネットワークポリシーは OSI レベル 3 および 4 で実装されます。

Amazon EKS では、Calico や Cilium などのネットワークポリシーエンジンを使用できます。デフォルトでは、ネットワークポリシーエンジンはインストールされません。のセットアップ方法については、それぞれのインストールガイドを参照してください。ネットワークポリシーの使用方法の詳細については、「EKS Security best practices」を参照してください。DNS ホスト名機能は、エンタープライズバージョンのネットワークポリシーエンジンで使用できます。これは、Kubernetes サービス/ポッドと AWS の外部で実行されるリソース間のトラフィックを制御するのに役立ちます。また、デフォルトでセキュリティグループをサポートしていない AWS サービスの DNS ホスト名のサポートを検討することもできます。

AWS Loadbalancer Controller を使用するように単一のセキュリティグループにタグ付けする

Pod に多くのセキュリティグループが割り当てられている場合、Amazon EKS では`kubernetes.io/cluster/$name`、共有または所有されている単一のセキュリティグループにタグ付けすることをお勧めします。タグを使用すると、AWS Loadbalancer Controller はセキュリティグループのルールを更新して、トラフィックを Pod にルーティングできます。Pod にセキュリティグループが 1 つだけ指定されている場合、タグの割り当てはオプションです。セキュリティグループで設定されたアクセス許可は付加的であるため、ロードバランサーコントローラーがルールを見つけて調整するには、1 つのセキュリティグループにタグ付けするだけで十分です。また、セキュリティグループで定義されているデフォルトのクォータに従うのにも役立ちます。

アウトバウンドトラフィックの NAT を設定する

ソース NAT は、セキュリティグループが割り当てられた Pod からのアウトバウンドトラフィックに対して無効になります。NAT ゲートウェイまたはインスタンスで設定されたプライベートサブネット上のインターネット起動ワーカーノードへのアクセスを必要とするセキュリティグループを使用するポッドの場合、CNI で外部 SNAT を有効にします。

```
kubectl set env daemonset -n kube-system aws-node AWS_VPC_K8S_CNI_EXTERNALSNAT=true
```

セキュリティグループを持つポッドをプライベートサブネットにデプロイする

セキュリティグループが割り当てられているポッドは、プライベートサブネットにデプロイされているノードで実行する必要があります。パブリックサブネットにデプロイされたセキュリティグループが割り当てられたポッドは、インターネットにアクセスできないことに注意してください。

Pod 仕様ファイルで terminationGracePeriodSeconds を検証する

Pod 仕様ファイルで `terminationGracePeriodSeconds`がゼロ以外であることを確認します (デフォルトは 30 秒) 。これは、Amazon VPC CNI がワーカーノードから Pod ネットワークを削除するために不可欠です。ゼロに設定すると、CNI プラグインはホストから Pod ネットワークを削除せず、ブランチ ENI は効果的にクリーンアップされません。

Fargate でのポッドのセキュリティグループの使用

Fargate で実行される Pod のセキュリティグループは、EC2 ワーカーノードで実行される Pod と非常によく似ています。たとえば、Fargate Pod に関連付ける SecurityGroupPolicy でセキュリティグループを参照する前に、セキュリティグループを作成する必要があります。デフォルトでは、SecurityGroupPolicy を Fargate Pod に明示的に割り当てない場合、クラスターセキュリティグループはすべての Fargate Pod に割り当てられます。わかりやすくするために、クラスターセキュリティグループを Fagate Pod の SecurityGroupPolicy に追加すると、セキュリティグループに最小限のセキュリティグループルールを追加する必要があります。describe-cluster API を使用してクラスターセキュリティグループを見つけることができます。

```
 aws eks describe-cluster --name CLUSTER_NAME --query
 'cluster.resourcesVpcConfig.clusterSecurityGroupId'
```

```
cat >my-fargate-sg-policy.yaml <<EOF
apiVersion: vpcresources.k8s.aws/v1beta1
kind: SecurityGroupPolicy
metadata:
```

```
  name: my-fargate-sg-policy
  namespace: my-fargate-namespace
spec:
  podSelector:
    matchLabels:
      role: my-fargate-role
  securityGroups:
    groupIds:
      - cluster_security_group_id
      - my_fargate_pod_security_group_id
EOF
```

セキュリティグループの最小ルールを以下に示します。これらのルールにより、Fargate Pod は kube-apiserver、kubelet、CoreDNS などのクラスター内サービスと通信できます。また、Fargate Pod との間のインバウンド接続とアウトバウンド接続を許可するルールを追加する必要があります。これにより、Pod は VPC 内の他の Pod またはリソースと通信できるようになります。さらに、Fargate が Amazon ECR または DockerHub などの他のコンテナレジストリからコンテナイメージをプルするためのルールを含める必要があります。詳細については、AWS 全般のリファレンスの「AWS IP アドレス範囲」を参照してください。

次のコマンドを使用して、Fargate Pod に適用されているセキュリティグループを検索できます。

```
kubectl get pod FARGATE_POD -o jsonpath='{.metadata.annotations.fargate\.amazonaws
\.com/pod-sg}{"\n"}'
```

上記のコマンドの enild を書き留めます。

```
aws ec2 describe-network-interfaces --network-interface-ids ENI_ID --query
 'NetworkInterfaces[*].Groups[*]'
```

新しいセキュリティグループを適用するには、既存の Fargate ポッドを削除して再作成する必要があります。たとえば、次のコマンドは example-app のデプロイを開始します。特定のポッドを更新するには、次のコマンドで名前空間とデプロイ名を変更できます。

```
kubectl rollout restart -n example-ns deployment example-pod
```

ロードバランシング

ロードバランサーは受信トラフィックを受信し、EKS クラスターでホストされている目的のアプリケーションのターゲットに分散します。これにより、アプリケーションの耐障害性が向上しま

す。EKS クラスターにデプロイすると、AWS Load Balancer コントローラーはそのクラスターの AWS Elastic Load Balancer を作成および管理します。LoadBalancer タイプの Kubernetes サービスが作成されると、AWS Load Balancer Controller は、受信したトラフィックを OSI モデルのレイヤー 4 で負荷分散する Network Load Balancer (NLB) を作成します。Kubernetes Ingress オブジェクトを作成すると、AWS Load Balancer Controller は、OSI モデルのレイヤー 7 でトラフィックをロードバランシングする Application Load Balancer (ALB) を作成します。

Load Balancerタイプの選択

AWS Elastic Load Balancing (ELB) ポートフォリオは、Application Load Balancer (ALB)、Network Load Balancer (NLB)、Gateway Load Balancer (GWLB)、Classic Load Balancer (CLB) のロードバランサーをサポートしています。このベストプラクティスセクションでは、EKS クラスターに最も関連性の高い ALB と NLB に焦点を当てます。

ロードバランサーのタイプを選択する際の主な考慮事項は、ワークロードの要件です。

詳細については、「製品比較」を参照してください。

ワークロードが HTTP/HTTPS の場合は Application Load Balancer (ALB) を選択します。

ワークロードが OSI モデルのレイヤー 7 でロードバランシングを必要とする場合、AWS Load Balancer Controller を使用して ALB をプロビジョニングできます。プロビジョニングについては、次のセクションで説明します。ALB は、前述の Ingress リソースによって制御および設定され、HTTP または HTTPS トラフィックをクラスター内の異なる Pod にルーティングします。ALB では、アプリケーショントラフィックルーティングアルゴリズムを柔軟に変更することができます。デフォルトのルーティングアルゴリズムはラウンドロビンで、未処理のリクエストが最も少ないルーティングアルゴリズムもあります。

ワークロードが TCP の場合、またはワークロードでクライアントのソース IP 保存が必要な場合は、Network Load Balancer (NLB) を選択します。

Network Load Balancer は、Open Systems Interconnection (OSI) モデルの 4 番目のレイヤー (トランスポート) で機能します。TCP および UDP ベースのワークロードに適しています。Network Load Balancer は、デフォルトで、ポッドにトラフィックを提示するときに、クライアントのアドレスのソース IP も保持します。

ワークロードが DNS を利用できない場合は、Network Load Balancer (NLB) を選択します。

NLB を使用するもう 1 つの重要な理由は、クライアントが DNS を利用できない場合です。この場合、Network Load Balancer の IPs が静的であるため、NLB はワークロードにより適している可能性があります。クライアントは、ロードバランサーへの接続時にドメイン名を IP アドレスに解決するときに DNS を使用することをお勧めしますが、クライアントのアプリケーションが DNS 解決をサポートしておらず、ハードコードされた IPs のみを受け入れる場合、NLB は IPs が静的であり、NLB の存続期間中は同じであるため、NLB の方が適しています。

ロードバランサーのプロビジョニング

ワークロードに最適なLoad Balancerを決定した後、お客様はロードバランサーをプロビジョニングするためのいくつかのオプションがあります。

AWS Load Balancer Controller をデプロイしてLoad Balancerをプロビジョニングする

EKS クラスター内でロードバランサーをプロビジョニングするには、2 つの主要な方法があります。

- AWS クラウドプロバイダーでのサービスコントローラーの活用 (レガシー)
- AWS Load Balancer Controller の活用 (推奨)

デフォルトでは、Kubernetes Service Controller はツリー内コントローラーとも呼ばれ、LoadBalancer タイプの Kubernetes Service リソースを調整します。このコントローラーは、Kubernetes Cloud Controller Manager として機能する AWS Cloud Provider コンポーネントに組み込まれています。 https://kubernetes.io/docs/concepts/architecture/cloud-controller/

プロビジョニングされた Elastic Load Balancer の設定は、Kubernetes Service マニフェストに追加する必要がある注釈によって制御されます。Service Controller と AWS Load Balancer Controller で使用される注釈は異なります。

Service Controller はレガシーであり、現在重要なバグ修正のみを受けています。LoadBalancer タイプの Kubernetes サービスを作成すると、Service Controller はデフォルトで AWS CLB を作成しますが、適切な注釈を使用する場合は AWS NLB を作成することもできます。Service Controller は Kubernetes Ingress リソースをサポートしておらず、IPv6 もサポートしていないことに注意してください。

EKS クラスターで AWS Load Balancer Controller を使用して、Kubernetes サービスと Ingress リソースを照合することをお勧めします。AWS Load Balancer Controller が調整プロセスを所有するように、Kubernetes Service または Ingress マニフェストで適切な注釈を使用する必要があります (Service Controller の代わりに)。

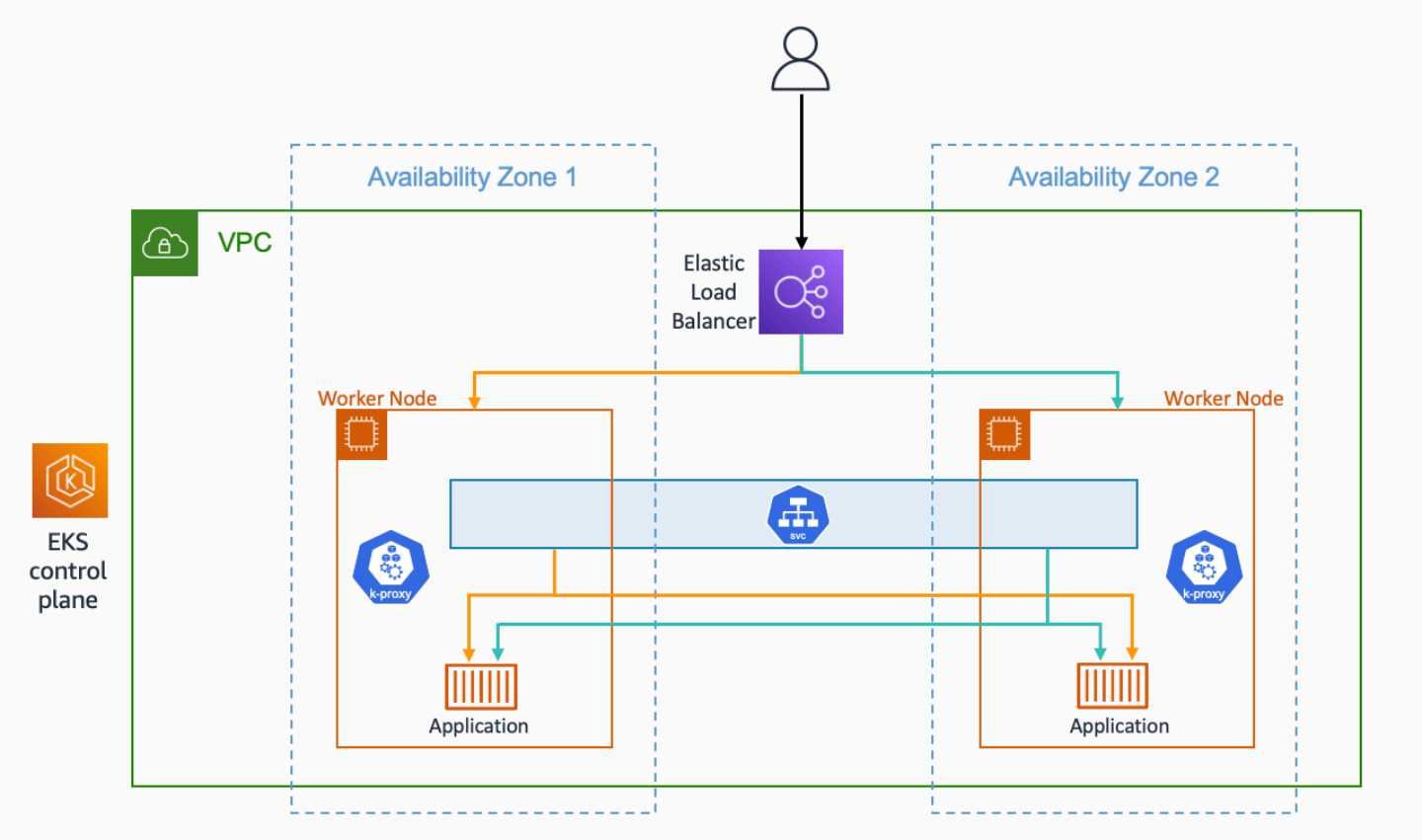
EKS Auto Mode を使用している場合は、AWS Load Balancer Controller が自動的に提供されます。インストールは必要ありません。

Load Balancerのターゲットタイプの選択

IP Target-Type を使用して Pod をターゲットとして登録する

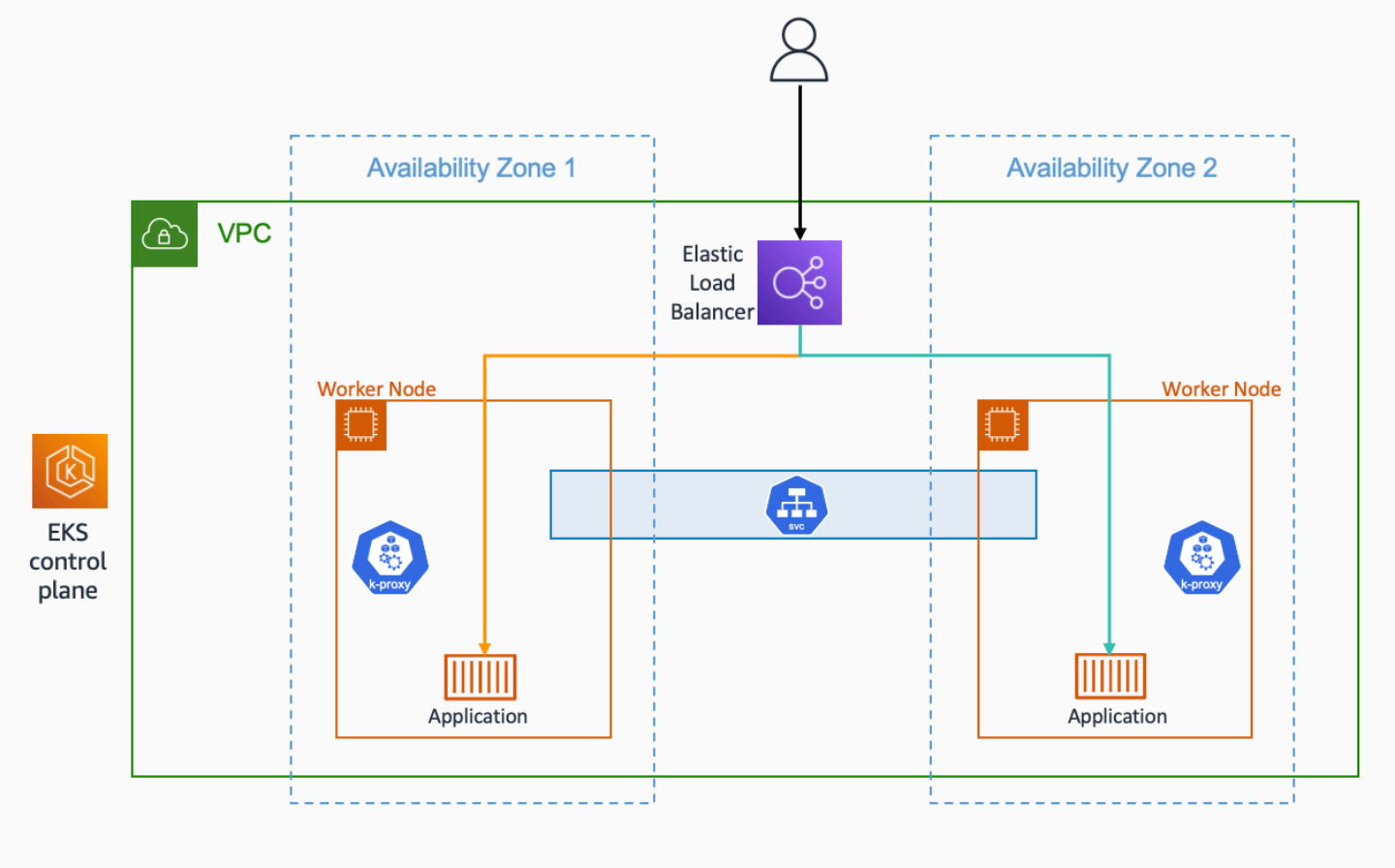
AWS Elastic Load Balancer: Network & Application は、受信したトラフィックをターゲットグループ内の登録済みターゲットに送信します。EKS クラスターには、ターゲットグループに登録できるターゲットのタイプが 2 つあります。インスタンスと IP です。どのターゲットタイプが使用されるかは、登録される内容と、Load Balancerからポッドへのトラフィックのルーティング方法に影響します。デフォルトでは、AWS Load Balancer コントローラーは「インスタンス」タイプを使用してターゲットを登録し、このターゲットはワーカーノードの IP と NodePort になります。これには以下が含まれます。

- Load Balancer からのトラフィックは NodePort のワーカーノードに転送され、これは iptables ルール (ノードで実行されている kube-proxy によって設定) によって処理され、ClusterIP の サービス (まだノード上) に転送されます。最後に、サービスは登録されたポッドをランダムに選択してトラフィックを転送します。このフローには複数のホップが含まれ、特に追加のレイテンシーが発生する可能性があります。これは、サービスが別の AZ にある可能性のある別のワーカーノードで実行されているポッドを選択する場合があるためです。
- Load Balancer はワーカーノードをターゲットとして登録するため、ターゲットに送信されるヘルスチェックはポッドによって直接受信されるのではなく、NodePort 上のワーカーノードによって受信され、ヘルスチェックトラフィックは上記の同じパスに従います。
- Load Balancer によって転送されるトラフィックはポッドに直接送信されないため、モニタリングとトラブルシューティングはより複雑です。適切なトラブルシューティングのために、ワーカーノードで受信したパケットをサービスClusterIP に慎重に関連付け、最終的にはポッドがパケットのパスを完全にend-to-endで可視化できるようにする必要があります。



これとは対照的に、ターゲットタイプを「IP」として設定する場合、意味は次のとおりです。

- Load Balancer からのトラフィックはポッドに直接転送されます。これにより、ワーカーノードとサービスクラスター IP の以前の追加ホップをバイパスするネットワークパスが簡素化され、サービスが別の AZ のポッドにトラフィックを転送し、最後にワーカーノードで iptables ルールのオーバーヘッド処理を削除した場合に発生するレイテンシーが短縮されます。
- Load Balancer のヘルスチェックはポッドによって直接受信され、応答されます。つまり、ターゲットステータス「正常」または「異常」はポッドのヘルスステータスを直接表したものです。
- モニタリングとトラブルシューティングが簡単で、パケット IP アドレスをキャプチャするツールを使用すると、送信元フィールドと送信先フィールドの Load Balancer とポッド間の双方向トラフィックが直接明らかになります。



追加した IP ターゲットを使用する AWS Elastic Load Balancing を作成するには:

- `alb.ingress.kubernetes.io/target-type: ip` Kubernetes Ingress (Application Load Balancer) を設定する際の Ingress のマニフェストへの注釈
- `service.beta.kubernetes.io/aws-load-balancer-nlb-target-type: ip` LoadBalancer (Network Load Balancer) タイプの Kubernetes サービスを設定するときのサービスのマニフェストへの 注釈。

Load Balancer ヘルスチェックの設定

Kubernetes は独自のヘルスチェックメカニズムを提供しますが (次のセクションで詳しく説明します)、Kubernetes コントロールプレーン外で機能する補完的な保護手段として ELB ヘルスチェックを実装することをお勧めします。この独立したレイヤーは、次の場合でもアプリケーションのモニタリングを継続します。

- Kubernetes コントロールプレーンの中断
- プローブ実行の遅延

- kubelet とポッド間のネットワークパーティション

上記のシナリオで最大可用性の高速復旧を必要とする重要なワークロードの場合、ELB ヘルスチェックは、Kubernetes のネイティブメカニズムの代わりにではなく、 と連携する重要な安全ネットを提供します。

ELB でヘルスチェックを設定して微調整するには、Service Controller または AWS Load Balancer Controller によって調整される Kubernetes Service または Ingress マニフェストで注釈を使用する必要があります。

可用性とポッドのライフサイクル

アプリケーションのアップグレード中は、ユーザーがダウンタイムを経験しないように、アプリケーションが常にリクエストを処理できることを確認する必要があります。このシナリオの一般的な課題の 1 つは、Kubernetes レイヤーとインフラストラクチャ、例えば外部 Load Balancer 間でワークロードの可用性ステータスを同期することです。次のいくつかのセクションでは、このようなシナリオに対処するためのベストプラクティスに焦点を当てます。

> **Note**
>
> 以下の説明は、Kubernetes のEndpointSlices に基づいています。 https://kubernetes.io/docs/concepts/services-networking/service/#endpoints以下のシナリオでは、両者の違いはごくわずかです。AWS Load Balancer Controller はデフォルトでエンドポイントを消費し、コントローラーで enable-endpoint-sliceflag を有効にすることで EndpointSlices を有効にできます。 enable-endpoint-sliceflag

ヘルスチェックを使用する

デフォルトでは、Kubernetes はプロセスヘルスチェックを実行し、ノード上の kubelet プロセスがコンテナのメインプロセスが実行されているかどうかを確認します。そうでない場合は、デフォルトでそのコンテナを再起動します。ただし、コンテナプロセスが実行されているがデッドロック状態にある場合、またはアプリケーションが正常に開始されたかどうかを識別するように Kubernetes プローブを設定することもできます。プローブは、exec、grpc、httpGet、および tcpSocket メカニズムに基づくことができます。プローブのタイプと結果に基づいて、コンテナを再起動できます。

ポッド作成プロセスの一連のイベントについては、以下の付録セクションのポッド作成を参照してください。

準備状況プローブを使用する

デフォルトでは、Pod 内のすべてのコンテナが Pod 条件を実行している場合、「準備完了」と見なされます。 https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/#pod-conditionsただし、アプリケーションは引き続きクライアントリクエストを処理できない場合があります。たとえば、アプリケーションがリクエストを処理できるようにするには、外部リソースから一部のデータまたは設定をプルする必要がある場合があります。このような状態では、アプリケーションを強制終了したり、リクエストを転送したりしません。準備状況プローブを使用すると、ポッドが「準備完了」と見なされないようにできます。つまり、プローブ結果が になるまで EndpointSlice オブジェクトに追加されませんsuccess。一方、プローブが行をさらに下回ると、Pod は EndpointSlice オブジェクトから削除されます。各コンテナの Pod マニフェストで準備状況プローブを設定できます。各ノードの kubeletプロセスは、そのノードのコンテナに対して準備状況プローブを実行します。

Pod の準備状況ゲートを使用する

準備状況プローブの 1 つの側面は、外部フィードバック/影響メカニズムがないことです。ノードの kubelet プロセスはプローブを実行し、プローブの状態を定義します。EndpointSlice Controller はエンドポイント (Pod) のリストを常に最新の状態に保つため、Kubernetes レイヤー (東西トラフィック) 内のマイクロサービス自体間のリクエストには影響しません。外部メカニズムが必要な理由とタイミング

Kubernetes Service タイプの Load Balancer または Kubernetes Ingress (北から南へのトラフィックの場合) を使用してアプリケーションを公開する場合、ロードバランサーが最新のリストターゲットも持つように、各 Kubernetes Service の Pod IPs のリストを外部インフラストラクチャロードバランサーに伝播する必要があります。AWS Load Balancer Controller は、ここでギャップを埋めます。AWS Load Balancer Controller を使用して を利用する場合target group: IP、AWS Load Balancer Controller kube-proxy も更新を受信し (経由watch)、ELB API と通信して Pod IP を ELB のターゲットとして設定し、登録を開始します。

デプロイのローリング更新を実行すると、新しいポッドが作成され、新しいポッドの条件が「準備完了」になるとすぐに、古い/既存のポッドが終了します。このプロセス中、Kubernetes EndpointSlice オブジェクトは、ELB が新しいポッドをターゲットとして登録するのにかかる時間よりも速く更新されます。「ターゲット登録」を参照してください。しばらくすると、Kubernetes レイヤーとクライアントリクエストを削除できるインフラストラクチャレイヤーの状態が一致しなくなる可能性があります。Kubernetes レイヤー内のこの期間中、新しい Pod はリクエストを処理する準備ができていますが、ELB の観点からは処理できません。

Pod Readiness Gates を使用すると、Pod 条件が「準備完了」と見なされる前に満たす必要がある追加要件を定義できます。AWS ELB の場合、AWS Load Balancer Controller は AWS ELB のター

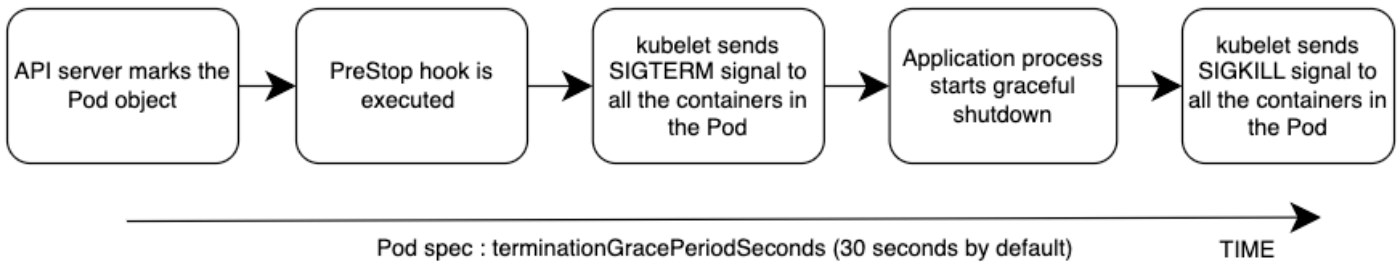
ゲット (ポッド) のステータスをモニタリングし、ターゲット登録が完了し、そのステータスが「正常」になると、コントローラーはポッドの条件を「準備完了」に更新します。このアプローチでは、AWS ELB のターゲットステータスである外部ネットワークの状態に基づいて Pod 条件に影響します。Pod Readiness Gates はローリング更新シナリオで重要です。これにより、デプロイのローリング更新が古いポッドを終了し、新しく作成されたポッドのターゲットステータスが AWS ELB で「正常」になるのを防ぐことができます。

アプリケーションを適切にシャットダウンする

クライアントがダウンタイムを経験しないように、アプリケーションは正常なシャットダウンを開始して SIGTERM シグナルに応答する必要があります。つまり、アプリケーションはデータの保存、ファイル記述子の閉鎖、データベース接続の閉鎖、処理中のリクエストの正常な完了、ポッド終了リクエストを満たすためのタイムリーな終了などのクリーンアップ手順を実行する必要があります。クリーンアップを完了できるように、猶予期間を十分な長さに設定する必要があります。SIGTERM シグナルに応答する方法については、アプリケーションで使用する各プログラミング言語のリソースを参照してください。

SIGTERM シグナルを受信したときにアプリケーションが正常にシャットダウンできない場合、またはシグナルを無視/受信しない場合、代わりに PreStop フックを使用してアプリケーションの正常なシャットダウンを開始できます。プリストップフックは、SIGTERM シグナルが送信される直前に実行され、アプリケーションコード自体にこれらのオペレーションを実装することなく、任意のオペレーションを実行できます。

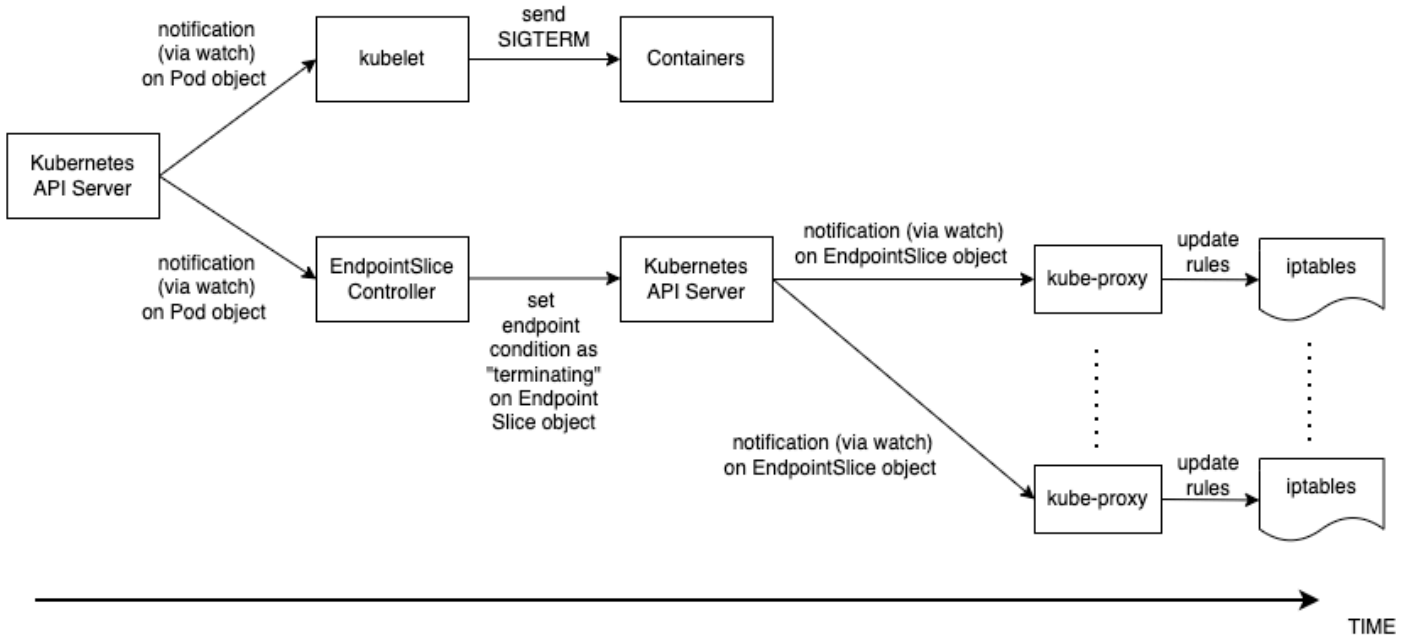
イベントの全体的なシーケンスを次の図に示します。注: アプリケーションの正常なシャットダウン手順の結果、または PreStop フックの結果に関係なく、アプリケーションコンテナは最終的に SIGKILL を介して猶予期間の終了時に終了します。



ポッド削除プロセスのイベントのシーケンスを確認するには、以下の付録セクションのポッド削除を参照してください。

クライアントリクエストを適切に処理する

Pod 削除のイベントのシーケンスは、Pod の作成とは異なります。Pod が作成される と、Kubernetes API の Pod IP `kubelet`が更新され、EndpointSlice オブジェクトのみが更新されます。一方、ポッドが終了すると、Kubernetes API は kubelet と EndpointSlice コントローラーの両方に同時に通知します。イベントのシーケンスを示す次の図を慎重に検査します。



状態が API サーバーから上記で説明したノードの iptables ルールまで伝播する方法は、興味深い競合状態を作成します。各ノードの kube-proxy がローカル iptables ルールを更新するよりもかなり早くコンテナが SIGKILL シグナルを受信する可能性が高いためです。このような場合、言及する価値のある 2 つのシナリオは です。

- アプリケーションが SIGTERM を受信したときに処理中のリクエストと接続をすぐに急に切断した場合、クライアントはその場で 50 倍のエラーが発生することになります。
- アプリケーションが SIGTERM を受信したときにすべての処理中のリクエストと接続が完全に処理されるようにしても、猶予期間中は、iptables ルールがまだ更新されていない可能性があるため、新しいクライアントリクエストがアプリケーションコンテナに送信されます。クリーンアップ手順がコンテナのサーバーソケットを閉じるまで、これらの新しいリクエストは新しい接続になります。猶予期間が終了すると、SIGTERM の後に確立された接続は、SIGKILL が送信されてから無条件に削除されます。

Pod 仕様で猶予期間を十分長く設定すると、この課題に対処できますが、伝播の遅延と実際のクライアントリクエストの数によっては、アプリケーションが接続を適切に終了するまでにかかる時間を予測するのが困難です。したがって、ここではあまり完全ではなく、最も実現可能なアプローチは、PreStop フックを使用して iptables ルールが更新されるまで SIGTERM シグナルを遅らせ、既存の接続のみが継続されるのではなく、新しいクライアントリクエストがアプリケーションに送信されないようにすることです。PreStop フックは、 などのシンプルな Exec ハンドラーにすることができますsleep 10。

上記の動作と推奨事項は、AWS Load Balancer Controller を使用して Kubernetes Service タイプの Load Balancer または Kubernetes Ingress (北 - 南トラフィック用) を使用してアプリケーションを公開しtarget group: IP、 を利用する場合も同様に適用されます。 Load Balancer kube-proxy AWS Load Balancer Controller と同様に、EndpointSlice オブジェクトで (監視を介して) 更新も受信し、ELB API と通信して ELB から Pod IP の登録解除を開始するためです。ただし、Kubernetes API または ELB API の負荷によっては時間がかかる場合があり、SIGTERM はすでにアプリケーションに送信されている可能性があります。ELB がターゲットの登録解除を開始すると、そのターゲットへのリクエストの送信が停止されるため、アプリケーションは新しいリクエストを受信せず、ELB はデフォルトで 300 秒の登録解除の遅延も開始します。登録解除プロセス中、ターゲットは基本的に ELB drainingがそのターゲットへの処理中のリクエスト/既存の接続がドレインするのを待機します。登録解除の遅延が期限切れになると、ターゲットは使用されず、そのターゲットへの実行中のリクエストは強制的に削除されます。

Pod 中断予算を使用する

アプリケーションのポッド中断予算 (PDB) を設定します。PDBlimitsします。これにより、StatefulSet またはデプロイでポッドの最小数または割合が維持されます。たとえば、クォーラムベースのアプリケーションでは、実行中のレプリカの数がクォーラムに必要な数を下回ることがないようにする必要があります。または、ウェブフロントエンドにより、ロードを処理するレプリカの数が合計の特定の割合を下回ることがないようにすることができます。PDB は、ドレインされるノードやデプロイの新しいバージョンがロールアウトされるなどのアクションからアプリケーションを保護します。PDB の は、ノードオペレーティングシステムの障害やネットワーク接続の喪失などの意図しない中断からアプリケーションを保護しないことに注意してください。詳細については、「Kubernetes でのアプリケーションの中断予算の指定」のドキュメントを参照してください。

リファレンス

- KubeCon Europe 2019 セッション - 準備完了? サービスヘルスのためのポッド準備ゲートの詳細
- 書籍 - Kubernetes in Action

- AWS ブログ - ALB on EKS (トラフィックを失うことなく) を使用してアプリケーションを迅速にスケーリングする方法

付録

ポッドの作成

ポッドがデプロイされ、クライアントリクエストを受信して処理する準備が整うシナリオでは、イベントのシーケンスを理解することが不可欠です。イベントのシーケンスについて説明します。

1. ポッドは Kubernetes コントロールプレーンに作成されます (kubectl コマンド、デプロイの更新、スケーリングアクション など)。
2. `kube-scheduler` は Pod をクラスター内のノードに割り当てます。
3. 割り当てられたノードで実行されている kubelet プロセスは (を介して`watch`) 更新を受け取り、コンテナランタイムと通信して Pod 仕様で定義されたコンテナを開始します。
4. コンテナの実行が開始されると、kubelet は Kubernetes API の Pod オブジェクトのように Pod 条件を更新します。 `Ready`
5. EndpointSlice Controller は Pod 条件の更新を (経由で`watch`) 受信し、Pod IP/Port を新しいエンドポイントとして各 Kubernetes サービスの EndpointSlice オブジェクト (Pod IPs) に追加します。
6. 各ノードの kube-proxy プロセスは、EndpointSlice オブジェクトの更新を (を介して`watch`) 受信し、各ノードの iptables ルールを新しい Pod IP/ポートで更新します。

ポッドの削除

ポッドの作成と同様に、ポッドの削除中のイベントのシーケンスを理解することが重要です。イベントのシーケンスについて説明します。

1. ポッド削除リクエストが Kubernetes API サーバーに送信されます (`kubectl`コマンド、デプロイの更新、スケーリングアクション など)。
2. Kubernetes API サーバーは、Pod オブジェクトの deletionTimestamp フィールドを設定することで、デフォルトで 30 秒の猶予期間を開始します。(猶予期間は を介してポッド仕様で設定できます`terminationGracePeriodSeconds`)
3. ノードで実行されている`kubelet`プロセスは、Pod オブジェクトで (監視を介して) 更新を受け取り、その Pod の各コンテナ内のプロセス識別子 1 (PID 1) に SIGTERM シグナルを送信します。次に、 を監視します`terminationGracePeriodSeconds`。

4. EndpointSlice Controller は、ステップ 2 から (を介してwatch) 更新を受け取り、各 Kubernetes サービスの EndpointSlice オブジェクト (ポッド IPs」に設定します。
5. 各ノードの kube-proxy プロセスは、EndpointSlice オブジェクトの更新 (経由watch) を受け取り、各ノードの iptables ルールが kube-proxy によって更新されて、Pod へのクライアントリクエストの転送を停止します。
6. terminationGracePeriodSeconds の有効期限が切れると、 は Pod 内の各コンテナの親プロセスに SIGKILL シグナルkubeletを送信し、強制的に終了します。
7. TheEndpointSliceは、EndpointSlice オブジェクトからエンドポイントを削除します。
8. API サーバーは Pod オブジェクトを削除します。

ネットワークパフォーマンスの問題に関する EKS ワークロードのモニタリング

DNS スロットリングの問題に対する CoreDNS トラフィックのモニタリング

DNS 集約型ワークロードを実行すると、DNS スロットリングが原因で断続的な CoreDNS 障害が発生することがあります。また、UnknownHostException エラーが時折発生するアプリケーションに影響を与える可能性があります。

Deployment for CoreDNS には、クラスター内の個別のワーカーノードで CoreDNS のインスタンスを実行するように Kubernetes スケジューラに指示するアンチアフィニティポリシーがあります。つまり、同じワーカーノードにレプリカが共存しないようにする必要があります。これにより、各レプリカからのトラフィックは異なる ENI を介してルーティングされるため、ネットワークインターフェイスあたりの DNS クエリの数を効果的に削減できます。1 秒あたり 1024 パケットの制限が原因で DNS クエリがスロットリングされていることに気付いた場合は、1) CoreDNS レプリカの数を増やすか、2) NodeLocal DNSCache を実装できます。詳細については、CoreDNS メトリクスのモニタリング」を参照してください。

Challenge

- パケットドロップは数秒で発生し、DNS スロットリングが実際に行われているかどうかを判断するために、これらのパターンを適切にモニタリングするのは難しい場合があります。
- DNS クエリは、Elastic Network Interface レベルでスロットリングされます。したがって、スロットリングされたクエリはクエリログには表示されません。

- フローログですべての IP トラフィックはキャプチャされません。例: Amazon DNS サーバーに接続したときにインスタンスによって生成されるトラフィック。独自の DNS サーバーを使用する場合、その DNS サーバーへのすべてのトラフィックがログに記録されます。

ソリューション

ワーカーノードの DNS スロットリングの問題を特定する簡単な方法は、`linklocal_allowance_exceeded`メトリクスをキャプチャすることです。linklocal_allowance_exceeded は、ローカルプロキシサービスへのトラフィックの PPS がネットワークインターフェイスの最大値を超えたためにドロップされたパケットの数です。これは、DNS サービス、インスタンスメタデータサービス、および Amazon Time Sync Service へのトラフィックに影響します。このイベントをリアルタイムで追跡する代わりに、このメトリクスを Amazon Managed Service for Prometheus にもストリーミングし、Amazon Managed Grafana で視覚化できます。

Conntrack メトリクスを使用した DNS クエリの遅延のモニタリング

CoreDNS スロットリング/クエリ遅延のモニタリングに役立つもう 1 つのメトリクスは、 `conntrack_allowance_available`および `conntrack_allowance_exceeded`です。Connections Tracked 許容値を超えたことによる接続障害は、他の許容値を超えたことによる接続障害よりも大きな影響を与える可能性があります。TCP を使用してデータを転送する場合、帯域幅、PPS などの EC2 インスタンスネットワーク許容量を超えたためにキューに入れられたパケットやドロップされたパケットは通常、TCP の輻輳制御機能のおかげで適切に処理されます。影響を受けるフローは遅くなり、失われたパケットは再送信されます。ただし、インスタンスが Connections Tracked 許容量を超えると、既存の接続の一部が閉じられて新しい接続のためのスペースが確保されるまで、新しい接続を確立することはできません。

`conntrack_allowance_available` と `conntrack_allowance_exceeded`は、インスタンスごとに異なる、追跡された接続の許容量をモニタリングするのに役立ちます。これらのネットワークパフォーマンスメトリクスにより、ネットワーク帯域幅、パケットPackets-Per-Second (PPS)、追跡された接続、リンクローカルサービスアクセス (Amazon DNS、インスタンスメタデータサービス、Amazon Time Sync) など、インスタンスのネットワーク許容量を超えたときにキューに入れられたパケットまたはドロップされたパケットの数を可視化できます。

`conntrack_allowance_available` は、そのインスタンスタイプの Connections Tracked 許容量に達する前にインスタンスによって確立できる追跡接続の数です (二トロベースのインスタンスでのみサポートされます)。 `conntrack_allowance_exceeded`は、接続追跡がインスタンスの最大数を超え、新しい接続を確立できなかったためにドロップされたパケットの数です。

その他の重要なネットワークパフォーマンスメトリクス

その他の重要なネットワークパフォーマンスメトリクスは次のとおりです。

bw_in_allowance_exceeded (メトリクスの理想的な値は 0 である必要があります) は、インバウンド集計帯域幅がインスタンスの最大値を超えたためにキューに入れられた、またはドロップされたパケットの数です。

bw_out_allowance_exceeded (メトリクスの理想的な値は 0 である必要があります) は、アウトバウンド集計帯域幅がインスタンスの最大値を超えたためにキューに入れられた、またはドロップされたパケットの数です。

pps_allowance_exceeded (メトリクスの理想的な値は 0 である必要があります) は、双方向 PPS がインスタンスの最大値を超えたためにキューに入れられた、またはドロップされたパケットの数です。

ネットワークパフォーマンスの問題がないかワークロードをモニタリングするためのメトリクスのキャプチャ

Elastic Network Adapter (ENA) ドライバーは、有効になっているインスタンスから上記のネットワークパフォーマンスメトリクスを発行します。すべてのネットワークパフォーマンスメトリクスは、CloudWatch エージェントを使用して CloudWatch に発行できます。詳細については、ブログを参照してください。

ここで、上記のメトリクスをキャプチャし、Amazon Managed Service for Prometheus に保存して、Amazon Managed Grafana を使用して視覚化しましょう。

前提条件

- ethtool - ワーカーノードに ethtool がインストールされていることを確認します
- AWS アカウントで設定された AMP ワークスペース。手順については、「AMP ユーザーガイド」の「ワークスペースの作成」を参照してください。
- Amazon Managed Grafana Workspace

Prometheus Node Exporterのデプロイ

ワーカーノードの ethtool メトリクスを収集するには、メトリクスを収集して Prometheus 形式で公開するコレクターをデプロイする必要があります。次のコマンドは、prometheus-

communityHelm リポジトリを追加し、ethtoolコレクターを有効にしてprometheus-node-exporterチャートをインストールします。

```
helm repo add prometheus-community https://prometheus-community.github.io/helm-charts

helm upgrade -i prometheus-node-exporter prometheus-community/prometheus-node-exporter
 \
  --set extraArgs[0]="--collector.ethtool" \
  --set extraArgs[1]="--collector.ethtool.device-include=(eth|em|eno|ens|enp)[0-9s]+" \
  --set extraArgs[2]="--collector.ethtool.metrics-include=.*" \
  --set podAnnotations."prometheus\.io/scrape='true'" \
  --set podAnnotations."prometheus\.io/port='9100'"
```

ADOT コレクターをデプロイして ethtool メトリクスをスクレイプし、Amazon Managed Service for Prometheus ワークスペースに保存します。

AWS Distro for OpenTelemetry (ADOT) をインストールする各クラスターには、メトリクスを Amazon Managed Service for Prometheus に保存するためのアクセス許可を AWS サービスアカウントに付与するために、このロールが必要です。IRSA を使用して IAM ロールを作成し、それを Amazon EKS のサービスアカウントに関連付けるには、以下の手順に従います。

```
eksctl create iamserviceaccount --name adot-collector --namespace default
 --cluster <CLUSTER_NAME> --attach-policy-arn arn:aws:iam::aws:policy/
AmazonPrometheusRemoteWriteAccess --attach-policy-arn arn:aws:iam::aws:policy/
AWSXrayWriteOnlyAccess --attach-policy-arn arn:aws:iam::aws:policy/
CloudWatchAgentServerPolicy --region <REGION> --approve  --override-existing-
serviceaccounts
```

ADOT コレクターをデプロイして prometheus ethtool エクスポーターからメトリクスをスクレイプし、Amazon Managed Service for Prometheus に保存しましょう

次の手順では、デプロイをモード値とする YAML ファイルの例を使用します。これはデフォルトモードであり、スタンドアロンアプリケーションと同様に ADOT Collector をデプロイします。この設定は、サンプルアプリケーションから OTLP メトリクスを受け取り、クラスター上のポッドからスクレイプされた Amazon Managed Service for Prometheus メトリクスを受け取ります。

```
curl -o collector-config-amp.yaml https://raw.githubusercontent.com/aws-observability/
aws-otel-community/master/sample-configs/operator/collector-config-amp.yaml
```

collector-config-amp.yaml で、以下を独自の値に置き換えます。

- モード: デプロイ
- serviceAccount: adot-collector
- エンドポイント: <YOUR_REMOTE_WRITE_ENDPOINT>
- リージョン: <YOUR_AWS_REGION>
- 名前: adot-collector

```
kubectl apply -f collector-config-amp.yaml
```

adot コレクターがデプロイされると、メトリクスは Amazon Prometheus に正常に保存されます。

Amazon Managed Service for Prometheus でアラートマネージャーを設定して通知を送信する

Amazon Managed Service for Prometheus のアラートマネージャーを使用して重要なアラートのアラートルールを設定し、Amazon SNS トピックに通知を送信できます。これまでに説明したメトリクスをチェックするように、記録ルールとアラートルールを設定しましょう。

Amazon Managed Service for Prometheus の ACK Controller を使用して、アラートルールと記録ルールをプロビジョニングします。

Amazon Managed Service for Prometheus サービスの ACL コントローラーをデプロイしましょう。

```
export SERVICE=prometheusservice
export RELEASE_VERSION=`curl -sL https://api.github.com/repos/aws-controllers-k8s/
$SERVICE-controller/releases/latest | grep '"tag_name":' | cut -d'"' -f4`
export ACK_SYSTEM_NAMESPACE=ack-system
export AWS_REGION=us-east-1
aws ecr-public get-login-password --region us-east-1 | helm registry login --username
 AWS --password-stdin public.ecr.aws
helm install --create-namespace -n $ACK_SYSTEM_NAMESPACE ack-$SERVICE-controller \
oci://public.ecr.aws/aws-controllers-k8s/$SERVICE-chart --version=$RELEASE_VERSION --
set=aws.region=$AWS_REGION
```

コマンドを実行し、しばらくすると次のメッセージが表示されます。

```
You are now able to create Amazon Managed Service for Prometheus (AMP) resources!

The controller is running in "cluster" mode.
```

```
The controller is configured to manage AWS resources in region: "us-east-1"

The ACK controller has been successfully installed and ACK can now be used to provision
 an Amazon Managed Service for Prometheus workspace.
```

ここで、アラートマネージャー定義とルールグループをプロビジョニングするための yaml ファイルを作成します。以下のファイルを として保存します。 rulegroup.yaml

```
apiVersion: prometheusservice.services.k8s.aws/v1alpha1
kind: RuleGroupsNamespace
metadata:
   name: default-rule
spec:
   workspaceID: <Your WORKSPACE-ID>
   name: default-rule
   configuration: |
     groups:
     - name: ppsallowance
       rules:
       - record: metric:pps_allowance_exceeded
         expr: rate(node_net_ethtool{device="eth0",type="pps_allowance_exceeded"}[30s])
       - alert: PPSAllowanceExceeded
         expr: rate(node_net_ethtool{device="eth0",type="pps_allowance_exceeded"}
 [30s]) > 0
         labels:
           severity: critical

         annotations:
           summary: Connections dropped due to total allowance exceeding for the
 (instance {{ $labels.instance }})
           description: "PPSAllowanceExceeded is greater than 0"
     - name: bw_in
       rules:
       - record: metric:bw_in_allowance_exceeded
         expr: rate(node_net_ethtool{device="eth0",type="bw_in_allowance_exceeded"}
[30s])
       - alert: BWINAllowanceExceeded
         expr: rate(node_net_ethtool{device="eth0",type="bw_in_allowance_exceeded"}
 [30s]) > 0
         labels:
           severity: critical
```

```
        annotations:
          summary: Connections dropped due to total allowance exceeding for the
 (instance {{ $labels.instance }})
          description: "BWInAllowanceExceeded is greater than 0"
    - name: bw_out
      rules:
      - record: metric:bw_out_allowance_exceeded
        expr: rate(node_net_ethtool{device="eth0",type="bw_out_allowance_exceeded"}
[30s])
      - alert: BWOutAllowanceExceeded
        expr: rate(node_net_ethtool{device="eth0",type="bw_out_allowance_exceeded"}
 [30s]) > 0
        labels:
          severity: critical

        annotations:
          summary: Connections dropped due to total allowance exceeding for the
 (instance {{ $labels.instance }})
          description: "BWoutAllowanceExceeded is greater than 0"
    - name: conntrack
      rules:
      - record: metric:conntrack_allowance_exceeded
        expr: rate(node_net_ethtool{device="eth0",type="conntrack_allowance_exceeded"}
[30s])
      - alert: ConntrackAllowanceExceeded
        expr: rate(node_net_ethtool{device="eth0",type="conntrack_allowance_exceeded"}
 [30s]) > 0
        labels:
          severity: critical

        annotations:
          summary: Connections dropped due to total allowance exceeding for the
 (instance {{ $labels.instance }})
          description: "ConnTrackAllowanceExceeded is greater than 0"
    - name: linklocal
      rules:
      - record: metric:linklocal_allowance_exceeded
        expr: rate(node_net_ethtool{device="eth0",type="linklocal_allowance_exceeded"}
[30s])
      - alert: LinkLocalAllowanceExceeded
        expr: rate(node_net_ethtool{device="eth0",type="linklocal_allowance_exceeded"}
 [30s]) > 0
        labels:
          severity: critical
```

```
        annotations:
          summary: Packets dropped due to PPS rate allowance exceeded for local
 services  (instance {{ $labels.instance }})
          description: "LinkLocalAllowanceExceeded is greater than 0"
```

WORKSPACE-ID を、使用しているワークスペースの Workspace ID に置き換えます。

ここで、アラートマネージャーの定義を設定します。以下の fie を として保存します。`alertmanager.yaml`

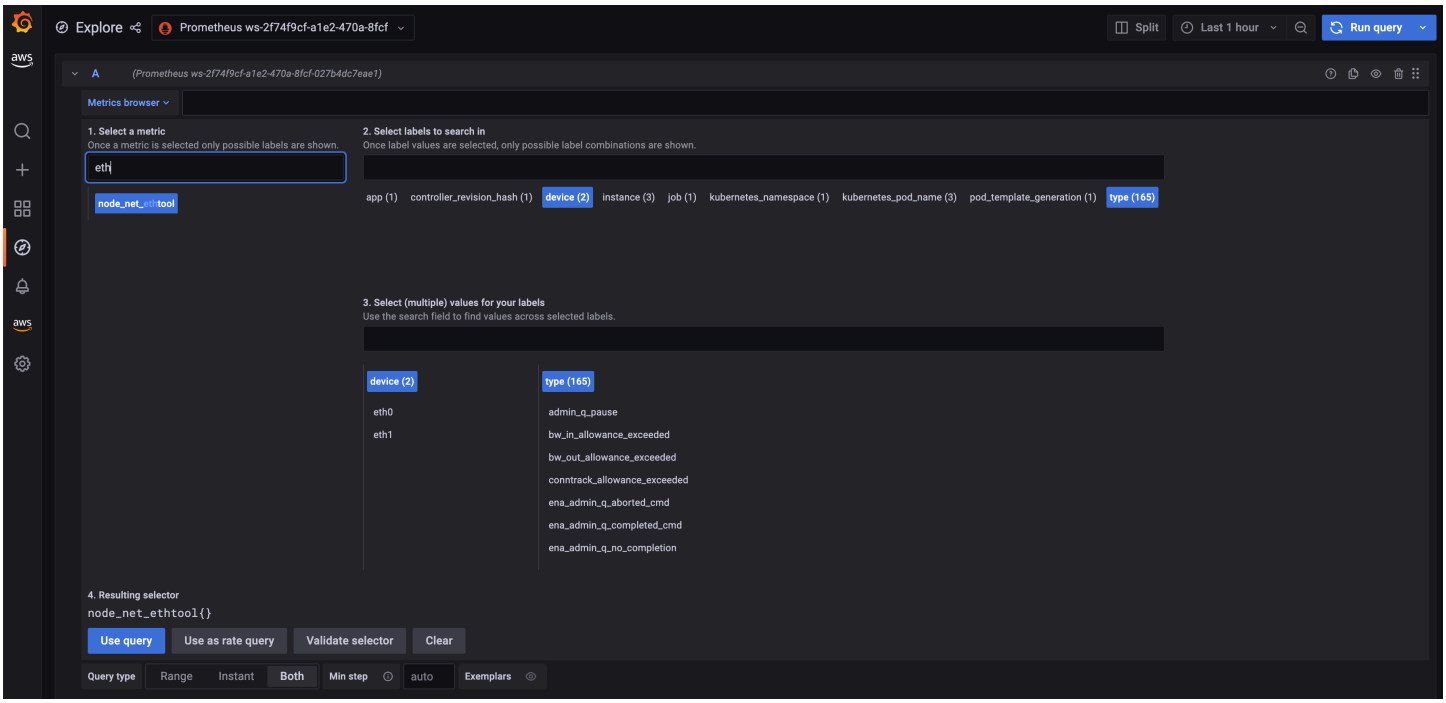
```
apiVersion: prometheusservice.services.k8s.aws/v1alpha1
kind: AlertManagerDefinition
metadata:
  name: alert-manager
spec:
  workspaceID: <Your WORKSPACE-ID >
  configuration: |
    alertmanager_config: |
      route:
         receiver: default_receiver
       receivers:
       - name: default_receiver
          sns_configs:
          - topic_arn: TOPIC-ARN
            sigv4:
              region: REGION
            message: |
              alert_type: {{ .CommonLabels.alertname }}
              event_type: {{ .CommonLabels.event_type }}
```

You WORKSPACE-ID を新しいワークスペースの Workspace ID に、TOPIC-ARN をアラートを送信する Amazon Simple Notification Service トピックの ARN に、REGION をワークロードの現在のリージョンに置き換えます。ワークスペースに Amazon SNS にメッセージを送信するアクセス許可があることを確認します。

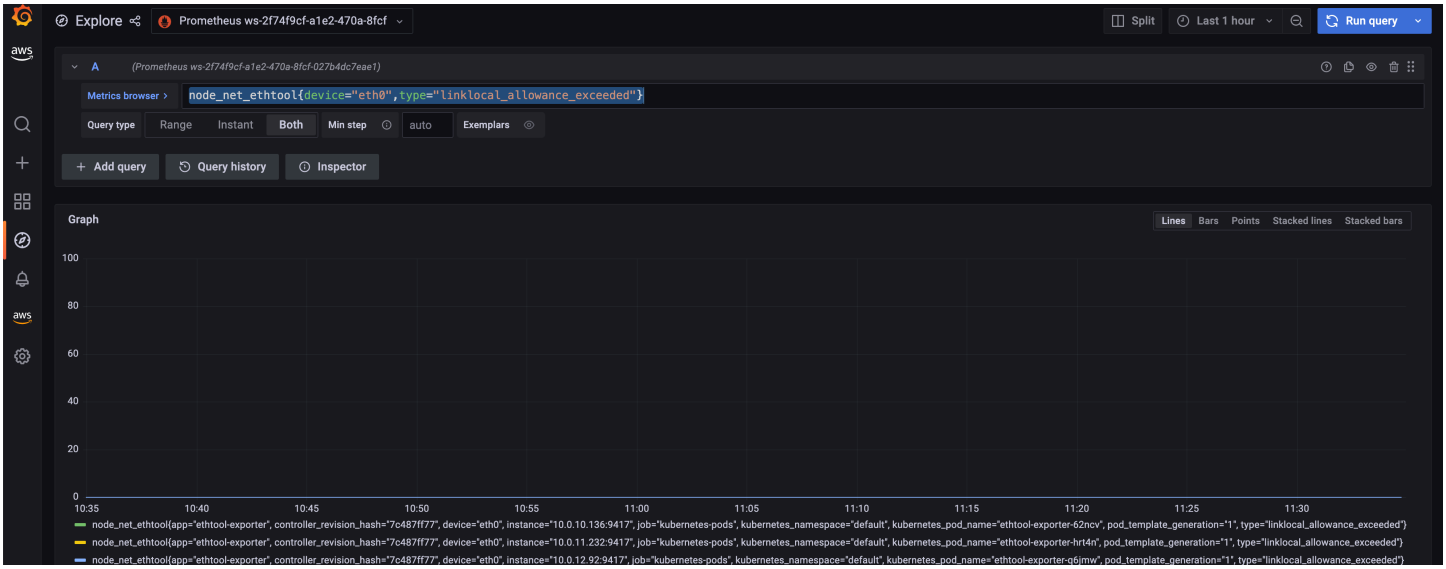
Amazon Managed Grafana で ethtool メトリクスを視覚化する

Amazon Managed Grafana 内のメトリクスを視覚化し、ダッシュボードを構築しましょう。Amazon Managed Service for Prometheus を Amazon Managed Grafana コンソール内のデータソースとして設定します。手順については、「データソースとして Amazon Prometheus を追加する」を参照してください。

ここで Amazon Managed Grafana のメトリクスを見てみましょう。探索ボタンをクリックして、ethtool を検索します。



クエリ を使用して、linklocal_allowance_exceeded メトリクスのダッシュボードを構築しましょうrate(node_net_ethtool{device="eth0",type="linklocal_allowance_exceeded"}[30s])。これにより、以下のダッシュボードが表示されます。

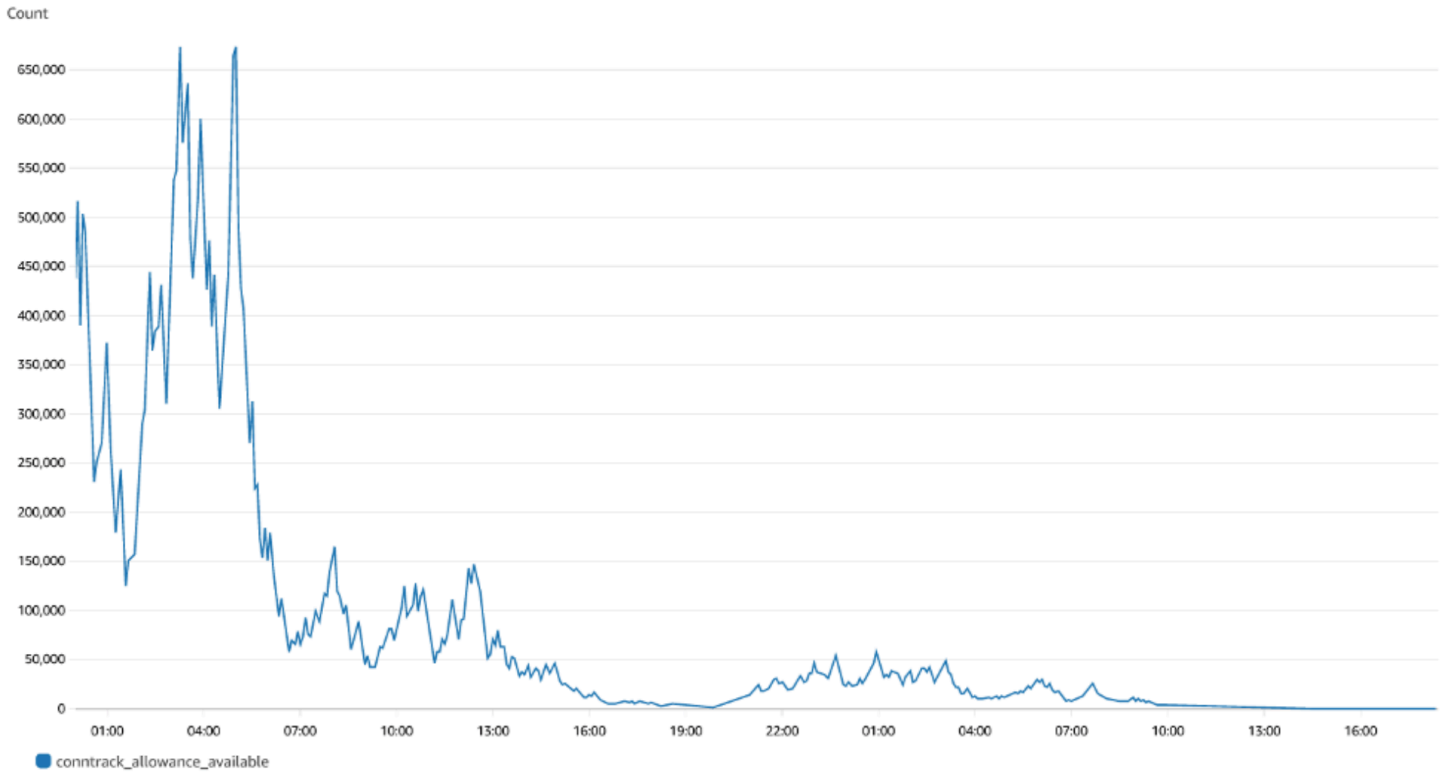


値が 0 であるため、パケットがドロップされていないことがわかります。

クエリ を使用して、conntrack_allowance_exceeded メトリクスのダッシュボードを構築しましょうrate(node_net_ethtool{device="eth0",type="conntrack_allowance_exceeded"}[30s])。これにより、以下のダッシュボードが表示されます。



メトリクスは、CloudWatch CloudWatch、CloudWatch で視覚化conntrack_allowance_exceededできます。CloudWatch の結果ダッシュボードは次のようになります。



値がゼロであるため、パケットがドロップされていないことがわかります。Nitro ベースのインスタンスを使用している場合は、 用の同様のダッシュボードを作成しconntrack_allowance_available、EC2 インスタンスの接続をプロアクティブにモニタリングできます。これをさらに拡張するには、Slack、SNS、Pagerduty などに通知を送信するように Amazon Managed Grafana でアラートを設定します。

IPVS モードで kube-proxy を実行する

IP Virtual Server (IPVS) モードでの EKS は、レガシー iptables モードで実行されている kube-proxy を使用して 1,000 を超えるサービスで大規模なクラスターを実行するときによく見られるネットワークレイテンシーの問題を解決します。このパフォーマンスの問題は、パケットごとに iptables パケットフィルタリングルールを順次処理した結果です。このレイテンシーの問題は、iptables の後継である nftables で解決されています。ただし、この執筆時点では、kube-proxy は nftables を活用するためにまだ開発中です。この問題を回避するには、クラスターを kube-proxy IPVS モードで実行するように設定できます。

概要:

Kubernetes バージョン 1.11 から GA となっていた IPVS は、線形検索ではなくハッシュテーブルを使用してパケットを処理し、数千のノードとサービスを持つクラスターに効率を提供します。IPVS はロードバランシング用に設計されており、Kubernetes ネットワークパフォーマンスの問題に適したソリューションです。

IPVS には、バックエンドポッドにトラフィックを分散するためのいくつかのオプションがあります。各オプションの詳細については、Kubernetes の公式ドキュメントに記載されていますが、以下に簡単なリストを示します。ラウンドロビン接続と最小接続は、Kubernetes の IPVS ロードバランシングオプションで最も人気のある選択肢の 1 つです。

```
- rr (Round Robin)
- wrr (Weighted Round Robin)
- lc (Least Connections)
- wlc (Weighted Least Connections)
- lblc (Locality Based Least Connections)
- lblcr (Locality Based Least Connections with Replication)
- sh (Source Hashing)
- dh (Destination Hashing)
- sed (Shortest Expected Delay)
- nq (Never Queue)
```

実装

EKS クラスターで IPVS を有効にするために必要なステップはわずかです。まず、EKS ワーカーノードイメージに Linux Virtual Server 管理`ipvsadm`パッケージがインストールされていることを確認します。Amazon Linux 2023 などの Fedora ベースのイメージにこのパッケージをインストールするには、ワーカーノードインスタンスで次のコマンドを実行します。

```
sudo dnf install -y ipvsadm
```

Ubuntu などの Debian ベースのイメージでは、インストールコマンドは次のようになります。

```
sudo apt-get install ipvsadm
```

次に、上記の IPVS 設定オプションのカーネルモジュールをロードする必要があります。これらのモジュールは、再起動後も存続するように、 `/etc/modules-load.d/` ディレクトリ内のファイルに書き込むことをお勧めします。

```
sudo sh -c 'cat << EOF > /etc/modules-load.d/ipvs.conf
ip_vs
ip_vs_rr
ip_vs_wrr
ip_vs_lc
ip_vs_wlc
ip_vs_lblc
ip_vs_lblcr
ip_vs_sh
ip_vs_dh
ip_vs_sed
ip_vs_nq
nf_conntrack
EOF'
```

次のコマンドを実行して、既に実行されているマシンにこれらのモジュールをロードできます。

```
sudo modprobe ip_vs
sudo modprobe ip_vs_rr
sudo modprobe ip_vs_wrr
sudo modprobe ip_vs_lc
sudo modprobe ip_vs_wlc
sudo modprobe ip_vs_lblc
```

```
sudo modprobe ip_vs_lblcr
sudo modprobe ip_vs_sh
sudo modprobe ip_vs_dh
sudo modprobe ip_vs_sed
sudo modprobe ip_vs_nq
sudo modprobe nf_conntrack
```

> **Note**
>
> これらのワーカーノードステップは、ワーカーノードのブートストラッププロセスの一環として、ユーザーデータスクリプトを介して、またはカスタムワーカーノード AMI を構築するために実行されるビルドスクリプトで実行することを強くお勧めします。

次に、IPVS モードで実行するようにクラスターの kube-proxy DaemonSet を設定します。これを行うには、 kube-proxymodeを に設定ipvsし、 ipvs schedulerを上記のロードバランシングオプションのいずれかに設定します。例えば、ラウンドロビンrrの場合です。

> **Warning**
>
> これは破壊的な変更であり、時間外に実行する必要があります。影響を最小限に抑えるために、最初の EKS クラスターの作成時にこれらの変更を行うことをお勧めします。

kube-proxy EKS アドオンを更新することで、AWS CLI コマンドを発行して IPVS を有効にできます。

```
aws eks update-addon --cluster-name $CLUSTER_NAME --addon-name kube-proxy \
  --configuration-values '{"ipvs": {"scheduler": "rr"}, "mode": "ipvs"}' \
  --resolve-conflicts OVERWRITE
```

または、クラスターの ConfigMap kube-proxy-config を変更することでこれを行うことができます。

```
kubectl -n kube-system edit cm kube-proxy-config
```

の scheduler設定を見つけipvs、 値を上記の ipvs ロードバランシングオプションの 1 つに設定します。例えば、ラウンドロビンrrの場合です。デフォルト設定が である mode設定を見つ

けiptables、値を に変更しますipvs。どちらのオプションの結果も、次の設定のようになります。

```
  iptables:
    masqueradeAll: false
    masqueradeBit: 14
    minSyncPeriod: 0s
    syncPeriod: 30s
  ipvs:
    excludeCIDRs: null
    minSyncPeriod: 0s
    scheduler: "rr"
    syncPeriod: 30s
  kind: KubeProxyConfiguration
  metricsBindAddress: 0.0.0.0:10249
  mode: "ipvs"
  nodePortAddresses: null
  oomScoreAdj: -998
  portRange: ""
  udpIdleTimeout: 250ms
```

これらの変更を行う前にワーカーノードがクラスターに結合されている場合は、kube-proxy DaemonSet を再起動する必要があります。

```
kubectl -n kube-system rollout restart ds kube-proxy
```

検証

いずれかのワーカーノードで次のコマンドを発行することで、クラスターとワーカーノードが IPVS モードで実行されていることを検証できます。

```
sudo ipvsadm -L
```

少なくとも、 の Kubernetes API Server サービスのエントリ10.100.0.1と の CoreDNS サービスのエントリを示す、次のような結果が表示されます10.100.0.10。

```
IP Virtual Server version 1.2.1 (size=4096)
Prot LocalAddress:Port Scheduler Flags
  -> RemoteAddress:Port           Forward Weight ActiveConn InActConn
TCP  ip-10-100-0-1.us-east-1. rr
  -> ip-192-168-113-81.us-eas Masq        1      0          0
```

```
  -> ip-192-168-162-166.us-ea Masq        1      1          0
TCP  ip-10-100-0-10.us-east-1 rr
  -> ip-192-168-104-215.us-ea Masq        1      0          0
  -> ip-192-168-123-227.us-ea Masq        1      0          0
UDP  ip-10-100-0-10.us-east-1 rr
  -> ip-192-168-104-215.us-ea Masq        1      0          0
  -> ip-192-168-123-227.us-ea Masq        1      0          0
```

> **Note**
>
> この出力例は、サービス IP アドレス範囲が の EKS クラスターから取得されます`10.100.0.0/16`。

EKS スケーラビリティのベストプラクティス

このガイドでは、EKS クラスターのスケーリングに関するアドバイスを提供します。EKS クラスターのスケーリングの目的は、単一のクラスターが実行できる作業量を最大化することです。単一の大規模な EKS クラスターを使用すると、複数のクラスターを使用する場合と比較して運用上の負荷を軽減できますが、マルチリージョンデプロイ、テナント分離、クラスターのアップグレードなどにはトレードオフがあります。このドキュメントでは、単一のクラスターで最大限のスケーラビリティを実現する方法に焦点を当てます。

このガイドの使い方

このガイドは、AWS での EKS クラスターの作成と管理を担当する開発者と管理者を対象としています。一般的な Kubernetes スケーリングプラクティスに焦点を当てていますが、EKS Anywhere を使用して AWS リージョンの外部で実行されるセルフマネージド Kubernetes クラスターやクラスターに固有のものはありません。

各トピックには簡単な概要があり、EKS クラスターを大規模に運用するための推奨事項とベストプラクティスが続きます。トピックは特定の順序で読み取る必要はなく、推奨事項はクラスターで機能するテストと検証を行わずに適用しないでください。

スケーリングディメンションについて

スケーラビリティはパフォーマンスや信頼性とは異なり、クラスターやワークロードのニーズを計画する際には 3 つすべてを考慮する必要があります。クラスターがスケールするときはモニタリングする必要がありますが、このガイドではモニタリングのベストプラクティスについては説明しません。EKS は大きなサイズにスケールできますが、クラスターを 300 ノードまたは 5,000 ポッド以上にスケールする方法を計画する必要があります。これらは絶対数ではありませんが、このガイドを複数のユーザー、エンジニア、サポートプロフェッショナルと共同で作成したものです。

Kubernetes でのスケーリングは多次元であり、あらゆる状況で機能する特定の設定や推奨事項はありません。スケーリングのガイダンスを提供できる主な領域は次のとおりです。

EKS クラスターの Kubernetes コントロールプレーンには、AWS が実行およびスケーリングするすべてのサービス (Kubernetes API サーバーなど) が含まれます。コントロールプレーンのスケーリングは AWS の責任ですが、責任を持ってコントロールプレーンを使用するのはお客様の責任です。

Kubernetes Data Plane スケーリングは、クラスターとワークロードに必要な AWS リソースを処理しますが、EKS コントロールプレーンの外部にあります。EC2 インスタンス、kubelet、ストレージなどのリソースはすべて、クラスターのスケールに応じてスケールする必要があります。

クラスターサービスは、クラスター内で実行され、クラスターとワークロードに機能を提供する Kubernetes コントローラーとアプリケーションです。これらは、EKS アドオン、およびコンプライアンスと統合のためにインストールする他の サービスや Helm チャートです。これらのサービスはワークロードによって依存されることが多く、ワークロードがスケールするにつれて、クラスターサービスはそれらに合わせてスケールする必要があります。

ワークロードはクラスターがある理由であり、クラスターと水平方向にスケーリングする必要があります。Kubernetes には、クラスターのスケールに役立つワークロードの統合と設定があります。名前空間やサービスなどの Kubernetes 抽象化には、アーキテクチャ上の考慮事項もあります。

超大規模スケーリング

1,000 個のノードまたは 50,000 個の Pod を超えて 1 つのクラスターをスケーリングする場合は、ご連絡いたします。サポートチームまたはテクニカルアカウントマネージャーに連絡して、このガイドに記載されている情報を超えて計画とスケーリングを支援できるスペシャリストに連絡することをお勧めします。Amazon EKS は、オンボーディング対象として選択されている場合、1 つのクラスターで最大 100,000 個のノードをサポートできます。

Kubernetes コントロールプレーン

Kubernetes コントロールプレーンは、Kubernetes API Server、Kubernetes Controller Manager、スケジューラ、および Kubernetes が機能するために必要なその他のコンポーネントで構成されます。これらのコンポーネントのスケーラビリティ制限は、クラスターで実行している内容によって異なりますが、スケーリングに最も影響がある領域には、Kubernetes のバージョン、使用率、個々のノードスケーリングが含まれます。

ワークロードとノードのバーストを制限する

> **Important**
>
> コントロールプレーンの API 制限に達しないようにするには、クラスターサイズを一度に 2 桁のパーセンテージで増やすスケーリングスパイクを制限する必要があります (たとえば、1,000 ノードから 1,100 ノード、4,000 ポッドから 4,500 ポッドに一度に増やすなど)。

EKS コントロールプレーンはクラスターが大きくなると自動的にスケールされますが、スケーリング速度には制限があります。EKS クラスターを初めて作成すると、コントロールプレーンはすぐに数百のノードや数千のポッドにスケールできなくなります。EKS によるスケーリングの改善の詳細については、このブログ記事を参照してください。

大規模なアプリケーションのスケーリングには、インフラストラクチャが完全に準備できるように適応する必要があります (ロードバランサーのウォームなど) 。スケーリングの速度を制御するには、アプリケーションに適したメトリクスに基づいてスケーリングしていることを確認してください。CPU とメモリのスケーリングでは、アプリケーションの制約を正確に予測できない場合があります。Kubernetes Horizontal Pod Autoscaler (HPA) でカスタムメトリクス (1 秒あたりのリクエスト数など) を使用すると、スケーリングオプションが適している可能性があります。

カスタムメトリクスを使用するには、Kubernetes ドキュメントの例を参照してください。より高度なスケーリングニーズがある場合、または外部ソース (AWS SQS キューなど) に基づいてスケーリングする必要がある場合は、イベントベースのワークロードスケーリングに KEDA を使用します。

ノードとポッドを安全にスケールダウンする

長時間実行されているインスタンスを置き換える

ノードを定期的に置き換えることで、設定のドリフトや、長時間の稼働後にのみ発生する問題 (メモリリークの遅延など) を回避することで、クラスターを正常に保つことができます。自動交換は、ノードのアップグレードとセキュリティパッチ適用の適切なプロセスとプラクティスを提供します。クラスター内のすべてのノードが定期的に置き換えられる場合、継続的なメンテナンスのために個別のプロセスを維持するのに必要な労力が少なくなります。

Karpenter の有効期限 (TTL) 設定を使用して、指定した時間実行されたインスタンスを置き換えます。セルフマネージド型ノードグループは、 `max-instance-lifetime`設定を使用してノードを自動的にサイクルできます。マネージド型ノードグループには現在この機能はありませんが、GitHub でリクエストを追跡できます。

使用率の低いノードを削除する

Kubernetes Cluster Autoscaler のスケールダウンしきい値を使用して実行中のワークロードがない場合、ノードを削除できます。`--scale-down-utilization-threshold`または、Karpenter では、`ttlSecondsAfterEmpty`プロビジョナー設定を使用できます。

ポッドの中断予算と安全なノードのシャットダウンを使用する

Kubernetes クラスターからポッドとノードを削除するには、コントローラーが複数のリソース (EndpointSlices など) を更新する必要があります。これを頻繁に行うか、速すぎると、変更がコントローラーに伝達されるにつれて、API サーバーのスロットリングやアプリケーションの停止が発生する可能性があります。Pod Disruption Budgets は、クラスター内でノードが削除または再スケジュールされたときにワークロードの可用性を保護するために、解約を遅らせるためのベストプラクティスです。

Kubectl の実行時にクライアント側のキャッシュを使用する

kubectl コマンドを非効率的に使用すると、Kubernetes API Server に追加の負荷がかかる可能性があります。kubectl を繰り返し使用するスクリプトやオートメーション (ループ用 など) を実行したり、ローカルキャッシュなしでコマンドを実行したりしないでください。

kubectl には、クラスターから検出情報をキャッシュして、必要な API コールの量を減らすクライアント側のキャッシュがあります。キャッシュはデフォルトで有効になっており、10 分ごとに更新されます。

コンテナから、またはクライアント側のキャッシュなしで kubectl を実行すると、API スロットリングの問題が発生する可能性があります。不要な API コールが発生しないように--cache-dir、 をマウントしてクラスターキャッシュを保持することをお勧めします。

kubectl 圧縮を無効にする

kubeconfig ファイルで kubectl 圧縮を無効にすると、API とクライアントの CPU 使用率が低下する可能性があります。デフォルトでは、サーバーはクライアントに送信されたデータを圧縮してネットワーク帯域幅を最適化します。これにより、すべてのリクエストに対してクライアントとサーバーに CPU 負荷がかかり、十分な帯域幅がある場合、圧縮を無効にするとオーバーヘッドとレイテンシーが軽減されます。圧縮を無効にするには、 --disable-compression=trueフラグを使用するか、kubeconfig ファイルdisable-compression: trueで を設定します。

```
apiVersion: v1
clusters:
- cluster:
    server: serverURL
    disable-compression: true
  name: cluster
```

シャードクラスターオートスケーラー

Kubernetes Cluster Autoscaler は、最大 1000 ノードまでスケールアップするようにテストされています。1000 個を超えるノードを持つ大規模なクラスターでは、Cluster Autoscaler の複数のインスタンスをシャードモードで実行することをお勧めします。各 Cluster Autoscaler インスタンスは、一連のノードグループをスケールするように設定されています。次の例は、各スケール 4 ノードグループに設定された 2 つのクラスターの自動スケーリング設定を示しています。

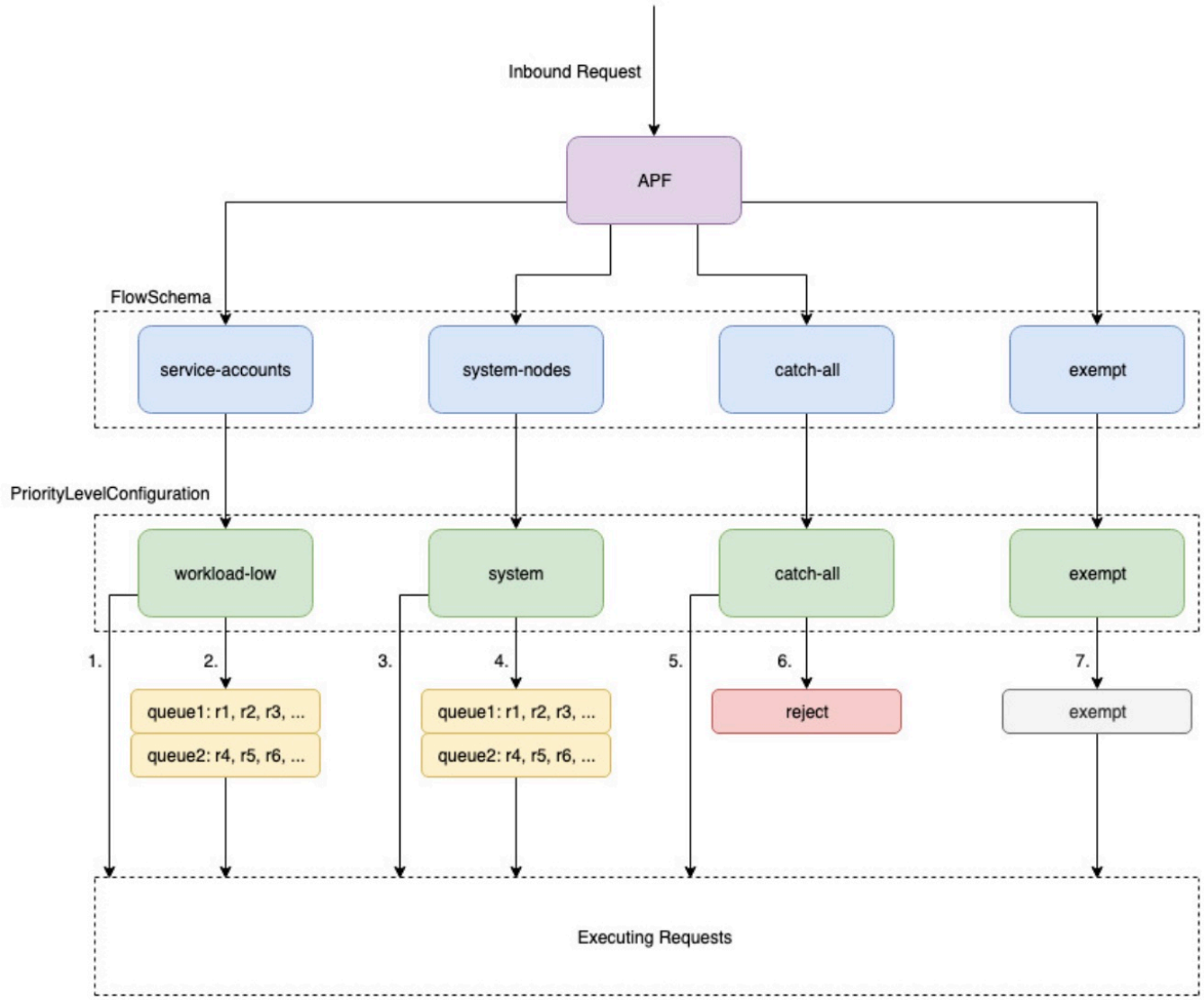
ClusterAutoscaler-1

```
autoscalingGroups:
- name: eks-core-node-grp-20220823190924690000000011-80c1660e-030d-476d-cb0d-
d04d585a8fcb
  maxSize: 50
  minSize: 2
- name: eks-data_m1-20220824130553925600000011-5ec167fa-ca93-8ca4-53a5-003e1ed8d306
  maxSize: 450
  minSize: 2
- name: eks-data_m2-20220824130733258600000015-aac167fb-8bf7-429d-d032-e195af4e25f5
  maxSize: 450
  minSize: 2
- name: eks-data_m3-20220824130553914900000003-18c167fa-ca7f-23c9-0fea-f9edefbda002
  maxSize: 450
  minSize: 2
```

ClusterAutoscaler-2

```
autoscalingGroups:
- name: eks-data_m4-20220824130553925500000000f-5ec167fa-ca86-6b83-ae9d-1e07ade3e7c4
  maxSize: 450
  minSize: 2
- name: eks-data_m5-20220824130744542100000017-02c167fb-a1f7-3d9e-a583-43b4975c050c
  maxSize: 450
  minSize: 2
- name: eks-data_m6-20220824130553924300000000d-9cc167fa-ca94-132a-04ad-e43166cef41f
  maxSize: 450
  minSize: 2
- name: eks-data_m7-20220824130553921000000009-96c167fa-ca91-d767-0427-91c879ddf5af
  maxSize: 450
  minSize: 2
```

API の優先度と公平性



1. If workload-low has not reached its concurrency limit, the request will be dispatched immediately.
2. If workload-low has reached its concurrency limit, the request will be queued prior to dispatch.
3. If system has not reached its concurrency limit, the request will be dispatched immediately.
4. If system has reached its concurrency limit, the request will be queued prior to dispatch.
5. If catch-all has not reached its concurrency limit, the request will be dispatched immediately.
6. If catch-all has reached its concurrency limit, the request will be dropped.
7. Requests mapping to exempt will always be dispatched.

概要:

リクエストが増加している間に自分自身が過負荷にならないように、API Server は、特定の時点で未処理の可能性がある処理中のリクエストの数を制限します。この制限を超えると、API Server は

リクエストの拒否を開始し、「リクエストが多すぎます」の 429 HTTP レスポンスコードをクライアントに返します。サーバーがリクエストを削除し、後でクライアントに再試行させる場合は、リクエスト数にサーバー側の制限を課さず、コントロールプレーンをオーバーロードして、パフォーマンスが低下したり、利用できなくなったりするのが推奨されます。

これらの処理中のリクエストをさまざまなリクエストタイプに分割する方法を設定するために Kubernetes が使用するメカニズムは、API Priority and Fairness と呼ばれます。API Server は、 フラグ`--max-requests-inflight`と `--max-mutating-requests-inflight`フラグで指定された値を合計することで、受け入れることができる処理中のリクエストの合計数を設定します。EKS は、これらのフラグに対して 400 リクエストと 200 リクエストのデフォルト値を使用し、一度に合計 600 リクエストをディスパッチできます。ただし、使用率とワークロードのチャーンの増加に応じてコントロールプレーンをより大きなサイズにスケールするため、それに応じて 2000 年までインフライトリクエストクォータが増加します (変更される可能性があります) 。APF は、これらの処理中のリクエストクォータをさまざまなリクエストタイプにさらに分割する方法を指定します。EKS コントロールプレーンは、各クラスターに登録された少なくとも 2 つの API Server で高可用性であることに注意してください。つまり、クラスターが処理できる処理中のリクエストの総数は、kube-apiserver ごとに設定された処理中のクォータの 2 倍 (水平方向にスケールアウトしている場合はそれ以上) になります。これは、最大の EKS クラスターで 1 秒あたり数千のリクエストに相当します。

PriorityLevelConfigurations と FlowSchemas と呼ばれる 2 種類の Kubernetes オブジェクトは、リクエストの合計数を異なるリクエストタイプに分割する方法を設定します。これらのオブジェクトは API Server によって自動的に維持され、EKS は指定された Kubernetes マイナーバージョンのこれらのオブジェクトのデフォルト設定を使用します。PriorityLevelConfigurations は、許可されたリクエストの合計数の一部を表します。たとえば、ワークロードの高い PriorityLevelConfiguration には、合計 600 件のリクエストのうち 98 件が割り当てられます。すべての PriorityLevelConfigurations に割り当てられたリクエストの合計は 600 に等しくなります (または、特定のレベルにリクエストの一部が付与されると API サーバーが切り上げられるため、600 をわずかに上回ります) 。クラスター内の PriorityLevelConfigurations と、それぞれに割り当てられたリクエストの数を確認するには、次のコマンドを実行します。EKS 1.32 のデフォルトは次のとおりです。

```
$ kubectl get --raw /metrics | grep apiserver_flowcontrol_nominal_limit_seats
apiserver_flowcontrol_nominal_limit_seats{priority_level="catch-all"} 13
apiserver_flowcontrol_nominal_limit_seats{priority_level="exempt"} 0
apiserver_flowcontrol_nominal_limit_seats{priority_level="global-default"} 49
apiserver_flowcontrol_nominal_limit_seats{priority_level="leader-election"} 25
apiserver_flowcontrol_nominal_limit_seats{priority_level="node-high"} 98
apiserver_flowcontrol_nominal_limit_seats{priority_level="system"} 74
apiserver_flowcontrol_nominal_limit_seats{priority_level="workload-high"} 98
```

```
apiserver_flowcontrol_nominal_limit_seats{priority_level="workload-low"} 245
```

オブジェクトの 2 番目のタイプは FlowSchemas です。特定のプロパティセットを持つ API サーバーリクエストは、同じ FlowSchema で分類されます。これらのプロパティには、認証されたユーザー、または API グループ、名前空間、リソースなどのリクエストの属性が含まれます。FlowSchema は、このタイプのリクエストをマッピングする PriorityLevelConfiguration も指定します。2 つのオブジェクトは、「このタイプのリクエストをこの処理中のリクエストのシェアにカウントします」と言います。リクエストが API サーバーに到達すると、必要なすべてのプロパティに一致するものが見つかるまで、各 FlowSchemas がチェックされます。複数の FlowSchemas がリクエストに一致する場合、API サーバーは一致する優先順位が最も小さい FlowSchema を選択します。この優先順位は オブジェクトのプロパティとして指定されます。

FlowSchemas と PriorityLevelConfigurations のマッピングは、次のコマンドを使用して表示できます。

```
$ kubectl get flowschemas
NAME                           PRIORITYLEVEL     MATCHINGPRECEDENCE
 DISTINGUISHERMETHOD   AGE     MISSINGPL
exempt                         exempt            1                  <none>
     7h19m   False
eks-exempt                     exempt            2                  <none>
     7h19m   False
probes                         exempt            2                  <none>
     7h19m   False
system-leader-election         leader-election   100                ByUser
     7h19m   False
endpoint-controller            workload-high     150                ByUser
     7h19m   False
workload-leader-election       leader-election   200                ByUser
     7h19m   False
system-node-high               node-high         400                ByUser
     7h19m   False
system-nodes                   system            500                ByUser
     7h19m   False
kube-controller-manager        workload-high     800                ByNamespace
     7h19m   False
kube-scheduler                 workload-high     800                ByNamespace
     7h19m   False
kube-system-service-accounts   workload-high     900                ByNamespace
     7h19m   False
eks-workload-high              workload-high     1000               ByUser
     7h14m   False
```

```
service-accounts              workload-low      9000                ByUser
     7h19m   False
global-default                global-default    9900                ByUser
     7h19m   False
catch-all                     catch-all         10000               ByUser
     7h19m   False
```

PriorityLevelConfigurations には、キュー、拒否、または免除のタイプを指定できます。Queue 型と Reject 型では、その優先度レベルの処理中のリクエストの最大数に制限が適用されますが、その制限に達すると動作は異なります。たとえば、ワークロードの高い PriorityLevelConfiguration は Queue 型を使用し、コントローラーマネージャー、エンドポイントコントローラー、スケジューラ、eks 関連コントローラー、および kube-system 名前空間で実行されているポッドから 98 個のリクエストを使用できます。Queue 型が使用されるため、API Server はリクエストをメモリに保持しようとし、これらのリクエストがタイムアウトする前に処理中のリクエスト数が 98 を下回ることを期待します。特定のリクエストがキューでタイムアウトした場合、または既にキューに入れられているリクエストが多すぎる場合、API Server はリクエストを削除してクライアントに 429 を返します。キューイングによりリクエストが 429 を受信できなくなる場合がありますが、リクエストに対するend-to-endのレイテンシーの増加というトレードオフが伴います。

次に、タイプが Reject のキャッチオール PriorityLevelConfiguration にマッピングされるキャッチオール FlowSchema について考えます。クライアントが 13 件の処理中のリクエストの上限に達した場合、API Server はキューイングを実行せず、429 レスポンスコードを使用してリクエストを即座に削除します。最後に、Exempt 型の PriorityLevelConfiguration へのマッピングリクエストは 429 を受信せず、常にすぐにディスパッチされます。これは、healthz リクエストや system:masters グループからのリクエストなど、優先度の高いリクエストに使用されます。

APF およびドロップされたリクエストのモニタリング

APF が原因でリクエストが削除されているかどうかを確認するために、 の API Server メトリクス`apiserver_flowcontrol_rejected_requests_total`をモニタリングして、影響を受ける FlowSchemas と PriorityLevelConfigurations を確認できます。例えば、このメトリクスは、サービスアカウント FlowSchema からの 100 件のリクエストが、ワークロードの低いキューでリクエストがタイムアウトしたために削除されたことを示しています。

```
% kubectl get --raw /metrics | grep apiserver_flowcontrol_rejected_requests_total
apiserver_flowcontrol_rejected_requests_total{flow_schema="service-
accounts",priority_level="workload-low",reason="time-out"} 100
```

特定の PriorityLevelConfiguration が 429 秒の受信またはキューイングによるレイテンシーの増加にどれだけ近いかを確認するには、同時実行数の制限と使用中の同時実行数の差を比較できます。この例では、100 リクエストのバッファがあります。

```
% kubectl get --raw /metrics | grep
 'apiserver_flowcontrol_nominal_limit_seats.*workload-low'
apiserver_flowcontrol_nominal_limit_seats{priority_level="workload-low"} 245

% kubectl get --raw /metrics | grep
 'apiserver_flowcontrol_current_executing_seats.*workload-low'
apiserver_flowcontrol_current_executing_seats{flow_schema="service-
accounts",priority_level="workload-low"} 145
```

特定の PriorityLevelConfiguration でキューイングが発生しているが、必ずしもリクエストが削除されていないかどうかを確認するには、 の メトリクスを参照apiserver_flowcontrol_current_inqueue_requestsできます。

```
% kubectl get --raw /metrics | grep
 'apiserver_flowcontrol_current_inqueue_requests.*workload-low'
apiserver_flowcontrol_current_inqueue_requests{flow_schema="service-
accounts",priority_level="workload-low"} 10
```

その他の便利な Prometheus メトリクスは次のとおりです。

- apiserver_flowcontrol_dispatched_requests_total
- apiserver_flowcontrol_request_execution_seconds
- apiserver_flowcontrol_request_wait_duration_seconds

APF メトリクスの完全なリストについては、アップストリームのドキュメントを参照してください。

ドロップされたリクエストの防止

ワークロードを変更して 429 を防ぐ

特定の PriorityLevelConfiguration が許可される処理中のリクエストの最大数を超えたために APF がリクエストを削除している場合、影響を受ける FlowSchemas のクライアントは、特定の時間に実行されるリクエストの数を減らすことができます。これは、429 秒間に行われたリクエストの合計数を

減らすことで実現できます。高価なリスト呼び出しなどの長時間実行されるリクエストは、実行されている期間全体の処理中のリクエストとしてカウントされるため、特に問題があることに注意してください。これらの高価なリクエストの数を減らしたり、これらのリスト呼び出しのレイテンシーを最適化したり (たとえば、リクエストごとにフェッチされるオブジェクトの数を減らしたり、監視リクエストを使用して に切り替えるなど) 、特定のワークロードに必要な合計同時実行数を減らすことができます。

APF 設定を変更して 429 を防ぐ

> **Warning**
>
> デフォルトの APF 設定を変更するのは、実行していることがわかっている場合のみです。APF 設定を誤って設定すると、API Server リクエストがドロップされ、ワークロードが大幅に中断される可能性があります。

リクエストの削除を防ぐもう 1 つの方法は、EKS クラスターにインストールされているデフォルトの FlowSchemas または PriorityLevelConfigurations を変更することです。EKS は、特定の Kubernetes マイナーバージョンの FlowSchemas と PriorityLevelConfigurations のアップストリームデフォルト設定をインストールします。API Server は、オブジェクトの次の注釈が false に設定されない限り、変更されると、これらのオブジェクトを自動的にデフォルトに調整します。

```
metadata:
  annotations:
    apf.kubernetes.io/autoupdate-spec: "false"
```

大まかに言うと、APF 設定は次のいずれかに変更できます。

- 関心のあるリクエストに、より多くの処理中の容量を割り当てます。
- 他のリクエストタイプの容量を枯渇させる可能性のある、重要でないリクエストや高価なリクエストを分離します。

これは、デフォルトの FlowSchemas と PriorityLevelConfigurations を変更するか、これらのタイプの新しいオブジェクトを作成することによって実現できます。オペレーターは、関連する PriorityLevelConfigurations オブジェクトの assuredConcurrencyShares の値を増やして、割り当てられた処理中のリクエストの割合を増やすことができます。さらに、リクエストがディスパッチされる前にキューに入れられたことによる追加のレイテンシーをアプリケーションが処理できる場合、特定の時間にキューに入れることができるリクエストの数を増やすこともできます。

または、顧客のワークロードに固有の新しい FlowSchema オブジェクトと PriorityLevelConfigurations オブジェクトを作成することもできます。より assuredConcurrencyShares を既存の PriorityLevelConfigurations または新しい PriorityLevelConfigurations に割り当てると、API Server ごとに全体的な制限が 600 インフライトのままになるため、他のバケットで処理できるリクエストの数が減少します。

APF のデフォルトを変更するときは、これらのメトリクスを非本番クラスターでモニタリングして、設定を変更しても意図しない 429 が発生しないようにする必要があります。

1. のすべての FlowSchemas の メトリクスをモニタリングして、バケットがリクエストの削除を開始しないようにapiserver_flowcontrol_rejected_requests_totalする必要があります。
2. apiserver_flowcontrol_nominal_limit_seats と の値を比較して、使用中の同時実行がその優先度レベルの制限を超過するリスクがないことを確認するapiserver_flowcontrol_current_executing_seats必要があります。

新しい FlowSchema と PriorityLevelConfiguration を定義するための一般的なユースケースの 1 つは、分離用です。長時間実行されるリストイベント呼び出しをポッドから独自のリクエスト共有に分離するとします。これにより、既存のサービスアカウント FlowSchema を使用するポッドからの重要なリクエストが 429 を受信し、リクエスト容量が枯渇するのを防ぐことができます。処理中のリクエストの合計数は有限ですが、この例では、APF 設定を変更して特定のワークロードのリクエスト容量をより適切に分割できることを示しています。

リストイベントリクエストを分離する FlowSchema オブジェクトの例:

```
apiVersion: flowcontrol.apiserver.k8s.io/v1
kind: FlowSchema
metadata:
  name: list-events-default-service-accounts
spec:
  distinguisherMethod:
    type: ByUser
  matchingPrecedence: 8000
  priorityLevelConfiguration:
    name: catch-all
  rules:
  - resourceRules:
    - apiGroups:
      - '*'
      namespaces:
```

```
    - default
    resources:
    - events
    verbs:
    - list
  subjects:
  - kind: ServiceAccount
    serviceAccount:
      name: default
      namespace: default
```

- この FlowSchema は、デフォルトの名前空間のサービスアカウントによって行われたすべてのリストイベント呼び出しをキャプチャします。
- 一致する優先順位 8000 は、既存の service-accounts FlowSchema で使用される 9000 の値よりも小さいため、これらのリストイベント呼び出しは service-accounts ではなく list-events-default-service-accounts と一致します。
- キャッチオール PriorityLevelConfiguration を使用して、これらのリクエストを分離しています。このバケットでは、これらの長時間実行されるリストイベント呼び出しで使用できる処理中のリクエストは 13 個のみです。ポッドは、13 を超えるリクエストを同時に発行しようとするとすぐに 429 を受信し始めます。

API サーバーでのリソースの取得

API サーバーから情報を取得することは、あらゆるサイズのクラスターで想定される動作です。クラスター内のリソース数をスケールすると、リクエストの頻度とデータ量がすぐにコントロールプレーンのボトルネックになり、API のレイテンシーと速度低下につながる可能性があります。レイテンシーの重要度によっては、注意しないと予期しないダウンタイムが発生します。

リクエストしている内容と、これらのタイプの問題を回避するための最初のステップの頻度を認識します。スケーリングのベストプラクティスに基づいてクエリの量を制限するガイダンスを次に示します。このセクションの提案は、最適なスケーリングが知られているオプションから順に提供されています。

共有インフォマーを使用する

Kubernetes API と統合するコントローラーとオートメーションを構築する場合、多くの場合、Kubernetes リソースから情報を取得する必要があります。これらのリソースを定期的にポーリングすると、API サーバーに大きな負荷がかかる可能性があります。

client-go ライブラリのインフォマーを使用すると、変更をポーリングするのではなく、イベントに基づいてリソースの変更を監視する利点があります。インフォマーは、イベントと変更に共有キャッシュを使用して負荷をさらに軽減するため、同じリソースを監視する複数のコントローラーが負荷を追加することはありません。

コントローラーは、特に大規模なクラスターでは、ラベルやフィールドセレクタなしでクラスター全体のリソースをポーリングしないようにする必要があります。フィルタリングされていないポーリングごとに、API サーバーを介して etcd から送信され、クライアントによってフィルタリングされる不要なデータが大量に必要です。ラベルと名前空間に基づいてフィルタリングすることで、API サーバーがクライアントに送信されるリクエストとデータをフルフィルするために実行する必要がある作業の量を減らすことができます。

Kubernetes API の使用を最適化する

カスタムコントローラーまたはオートメーションを使用して Kubernetes API を呼び出す場合は、必要なリソースのみに呼び出しを制限することが重要です。制限がないと、API サーバーや etcd に不要な負荷が発生する可能性があります。

可能な限り、Watch 引数を使用することをお勧めします。引数がない場合、デフォルトの動作はオブジェクトを一覧表示することです。リストの代わりに watch を使用するには、 API リクエストの末尾`?watch=true`に を追加できます。たとえば、ウォッチを使用してデフォルトの名前空間内のすべてのポッドを取得するには、次のようにします。

```
/api/v1/namespaces/default/pods?watch=true
```

オブジェクトを一覧表示する場合は、一覧表示するオブジェクトの範囲と返されるデータの量を制限する必要があります。リクエストに引`limit=500`数を追加することで、返されるデータを制限できます。`fieldSelector` 引数と`/namespace/`パスは、リストの範囲を必要に応じて絞り込むのに役立ちます。たとえば、デフォルトの名前空間で実行中のポッドのみを一覧表示するには、次の API パスと引数を使用します。

```
/api/v1/namespaces/default/pods?fieldSelector=status.phase=Running&limit=500
```

または、 で実行されているすべてのポッドを一覧表示します。

```
/api/v1/pods?fieldSelector=status.phase=Running&limit=500
```

監視呼び出しやリストされたオブジェクトを制限するもう 1 つのオプション
は、`resourceVersions`Kubernetes ドキュメントで参照できる を使用することで

す。resourceVersion 引数がないと、利用可能な最新バージョンが返されます。このバージョンでは、データベースの読み込みが最も高価で低速な etcd クォーラム読み込みが必要になります。resourceVersion は、クエリしようとしているリソースによって異なり、metadata.resourseVersionフィールドにあります。これは、単に通話を一覧表示するのではなく、ウォッチコールを使用する場合にも推奨されます。

API サーバーキャッシュから結果を返す特別な がresourceVersion=0利用可能です。これにより、 etcd 負荷を減らすことができますが、ページ分割はサポートされていません。

```
/api/v1/namespaces/default/pods?resourceVersion=0
```

resourceVersion を前述のリストまたは watch から受信した最新の既知の値に設定して watch を使用することをお勧めします。これは client-go で自動的に処理されます。ただし、他の言語で k8s クライアントを使用している場合は、再確認することをお勧めします。

```
/api/v1/namespaces/default/pods?watch=true&resourceVersion=362812295
```

引数なしで API を呼び出すと、API サーバーや etcd にとって最もリソースを大量に消費します。この呼び出しでは、ページ分割やスコープの制限を行わずにすべての名前空間内のすべてのポッドを取得し、etcd からのクォーラム読み取りが必要になります。

```
/api/v1/pods
```

DaemonSet が群れを雷撃するのを防ぐ

DaemonSet は、すべての (または一部の) ノードがポッドのコピーを実行するようにします。ノードがクラスターに参加すると、daemonset-controller はそれらのノードのポッドを作成します。ノードがクラスターを離れると、それらのポッドはガベージコレクションされます。DaemonSet を削除すると、作成したポッドがクリーンアップされます。

DaemonSet の一般的な用途は次のとおりです。

- すべてのノードでクラスターストレージデーモンを実行する
- すべてのノードでログ収集デーモンを実行する
- すべてのノードでノードモニタリングデーモンを実行する

数千のノードがあるクラスターでは、新しい DaemonSet の作成、DaemonSet の更新、またはノード数の増加により、コントロールプレーンに高い負荷がかかる可能性があります。DaemonSet ポッ

ドがポッドの起動時に高価な API サーバーリクエストを発行すると、多数の同時リクエストからコントロールプレーンで大量のリソースが使用される可能性があります。

通常のオペレーションでは、 を使用してRollingUpdate、新しい DaemonSet ポッドを段階的にロールアウトできます。RollingUpdate 更新戦略では、DaemonSet テンプレートを更新すると、コントローラーは古い DaemonSet ポッドを強制終了し、制御された方法で新しい DaemonSet ポッドを自動的に作成します。DaemonSet の最大 1 つのポッドは、更新プロセス全体で各ノードで実行されます。を 1、0、60 maxSurge maxUnavailableに設定することで、段階的なロールアウトを実行できますminReadySeconds。更新戦略を指定しない場合、Kubernetes はデフォルトで を 1、0、minReadySeconds0 maxSurge RollingUpdatemaxUnavailableとして作成します。

```
minReadySeconds: 60
strategy:
  type: RollingUpdate
  rollingUpdate:
    maxSurge: 0
    maxUnavailable: 1
```

DaemonSet が既に作成されており、すべてのノードで予想される数のポッドがある場合、 は新しい DaemonSet Ready ポッドを段階的にロールアウトRollingUpdateします。群れの問題は、RollingUpdate戦略の対象ではない特定の条件下で発生する可能性があります。

DaemonSet の作成時に群れが騙されるのを防ぐ

デフォルトでは、RollingUpdate設定に関係なく、kube-controller-manager の daemonset-controller は、新しい DaemonSet を作成するときに、一致するすべてのノードのポッドを同時に作成します。DaemonSet の作成後にポッドを段階的にロールアウトするには、 NodeSelectorまたは を使用できますNodeAffinity。これにより、ゼロノードに一致する DaemonSet が作成され、ノードを徐々に更新して、DaemonSet からポッドを制御されたレートで実行できるようになります。このアプローチに従うことができます。

- のすべてのノードにラベルを追加しますrun-daemonset=false。

```
kubectl label nodes --all run-daemonset=false
```

- run-daemonset=false ラベルのないノードに一致するNodeAffinity設定で DaemonSet を作成します。これにより、最初は DaemonSet に対応するポッドがなくなります。

```
affinity:
  nodeAffinity:
    requiredDuringSchedulingIgnoredDuringExecution:
      nodeSelectorTerms:
      - matchExpressions:
        - key: run-daemonset
          operator: NotIn
          values:
          - "false"
```

- 制御されたレートでノードからrun-daemonset=falseラベルを削除します。この bash スクリプトを例として使用できます。

```
#!/bin/bash

nodes=$(kubectl get --raw "/api/v1/nodes" | jq -r '.items | .[].metadata.name')

for node in ${nodes[@]}; do
   echo "Removing run-daemonset label from node $node"
   kubectl label nodes $node run-daemonset-
   sleep 5
done
```

- 必要に応じて、DaemonSet オブジェクトから NodeAffinity設定を削除します。また、これにより がトリガーRollingUpdateされ、DaemonSet テンプレートが変更されたため、既存のすべての DaemonSet ポッドが徐々に置き換えられることに注意してください。

ノードスケールアウトで群れが雷鳴するのを防ぐ

DaemonSet の作成と同様に、新しいノードを高速で作成すると、多数の DaemonSet ポッドが同時に起動する可能性があります。コントローラーがこの同じレートで DaemonSet ポッドを作成できるように、制御されたレートで新しいノードを作成する必要があります。これが不可能な場合は、 を使用して、新しいノードを最初に既存の DaemonSet の対象外にすることができますNodeAffinity。次に、新しいノードにラベルを徐々に追加して、daemonset-controller が制御されたレートでポッドを作成できるようにします。このアプローチに従うことができます。

- のすべての既存のノードにラベルを追加する run-daemonset=true

```
kubectl label nodes --all run-daemonset=true
```

- DaemonSet を、run-daemonset=trueラベルを持つ任意のノードと一致するようにNodeAffinity設定で更新します。また、これにより がトリガーRollingUpdateされ、DaemonSet テンプレートが変更されたため、既存のすべての DaemonSet ポッドが徐々に置き換えられることに注意してください。次のステップに進む前にRollingUpdate、 が完了するまで待つ必要があります。

```
affinity:
  nodeAffinity:
    requiredDuringSchedulingIgnoredDuringExecution:
      nodeSelectorTerms:
      - matchExpressions:
        - key: run-daemonset
          operator: In
          values:
          - "true"
```

- クラスターに新しいノードを作成します。これらのノードには run-daemonset=trueラベルがないため、DaemonSet はそれらのノードと一致しないことに注意してください。
- run-daemonset=true ラベルを新しいノード (現在 run-daemonsetラベルがありません) に制御されたレートで追加します。この bash スクリプトを例として使用できます。

```
#!/bin/bash

nodes=$(kubectl get --raw "/api/v1/nodes?labelSelector=%21run-daemonset" | jq -r
 '.items | .[].metadata.name')

for node in ${nodes[@]}; do
   echo "Adding run-daemonset=true label to node $node"
   kubectl label nodes $node run-daemonset=true
   sleep 5
done
```

- 必要に応じて、DaemonSet オブジェクトから NodeAffinity設定を削除し、すべてのノードから run-daemonsetラベルを削除します。

DaemonSet の更新で群れが雷鳴するのを防ぐ

RollingUpdate ポリシーは、 である DaemonSet ポッドmaxUnavailableの設定のみを尊重しますReady。DaemonSet にNotReadyポッドのみまたはNotReadyポッドの大部分があり、テンプレートを更新すると、daemonset-controller は任意のポッドに対して同時に新しいNotReadyポッドを作成します。これにより、NotReady大量のポッドがある場合に群れの問題が発生する可能性があります。たとえば、ポッドが継続的にクラッシュループしている場合や、イメージのプルに失敗している場合などです。

DaemonSet を更新したときにポッドを段階的にロールアウトするには、DaemonSet NotReady の更新戦略を から RollingUpdateに一時的に変更できますOnDelete。ではOnDelete、DaemonSet テンプレートを更新すると、コントローラーは古いポッドを手動で削除した後に新しいポッドを作成し、新しいポッドのロールアウトを制御できるようにします。このアプローチに従うことができます。

- DaemonSet NotReady にポッドがあるかどうかを確認します。
- そうでない場合は、DaemonSet テンプレートを安全に更新でき、RollingUpdate戦略によって段階的なロールアウトが保証されます。
- 「はい」の場合は、まず DaemonSet を更新して OnDelete戦略を使用する必要があります。

```
updateStrategy:
  type: OnDelete
```

- 次に、必要な変更で DaemonSet テンプレートを更新します。
- この更新後、管理されたレートでポッド削除リクエストを発行することで、古い DaemonSet ポッドを削除できます。この bash スクリプトは、kube-system 名前空間で DaemonSet 名が fluentd-elasticsearch である例として使用できます。

```
#!/bin/bash

daemonset_pods=$(kubectl get --raw "/api/v1/namespaces/kube-system/pods?
labelSelector=name%3Dfluentd-elasticsearch" | jq -r '.items | .[].metadata.name')

for pod in ${daemonset_pods[@]}; do
   echo "Deleting pod $pod"
   kubectl delete pod $pod -n kube-system
   sleep 5
```

```
done
```

- 最後に、DaemonSet を以前の`RollingUpdate`戦略に更新できます。

Kubernetes データプレーン

EC2 インスタンスタイプの選択は、複数のワークロードを持つクラスターでお客様が直面する最も難しい決定の 1 つである可能性があります。one-size-fitsソリューションはありません。スケーリングコンピューティングの一般的な落とし穴を回避するのに役立つヒントをいくつか紹介します。

自動ノード自動スケーリング

負荷を減らし、Kubernetes と深く統合するノードの自動スケーリングを使用することをお勧めします。マネージド型ノードグループと Karpenter は、大規模なクラスターに推奨されます。

マネージド型ノードグループは、Amazon EC2 Auto Scaling グループの柔軟性と、マネージド型アップグレードと設定の利点を提供します。Kubernetes Cluster Autoscaler を使用してスケーリングでき、さまざまなコンピューティングニーズを持つクラスターの一般的なオプションです。

Karpenter は、AWS によって作成されたオープンソースのワークロードネイティブノードオートスケーラーです。ノードグループを管理することなく、リソース (GPU など) のワークロード要件とテイントと許容範囲 (ゾーンスプレッドなど) に基づいてクラスター内のノードをスケーリングします。ノードは EC2 から直接作成されるため、デフォルトのノードグループクォータである 1 グループあたり 450 ノードが回避され、運用オーバーヘッドが少なく、インスタンス選択の柔軟性が向上します。可能な場合は Karpenter を使用することをお勧めします。

さまざまな EC2 インスタンスタイプを使用する

各 AWS リージョンでは、インスタンスタイプごとに使用できるインスタンスの数に制限があります。1 つのインスタンスタイプのみを使用するクラスターを作成し、そのリージョンの容量を超えてノード数をスケールすると、使用可能なインスタンスがないというエラーが表示されます。この問題を回避するには、クラスターで使用できるインスタンスのタイプを任意に制限しないでください。

Karpenter は、デフォルトで互換性のあるインスタンスタイプの幅広いセットを使用し、保留中のワークロード要件、可用性、コストに基づいてプロビジョニング時にインスタンスを選択します。NodePools の `karpenter.k8s.aws/instance-category`キーで使用されるインスタンスタイプのリストを拡張できます。

Kubernetes Cluster Autoscaler では、ノードグループを一貫してスケーリングできるように、同じサイズにする必要があります。CPU とメモリのサイズに基づいて複数のグループを作成し、個別にスケーリングする必要があります。ec2-instance-selector を使用して、ノードグループ用に同じサイズのインスタンスを識別します。

```
ec2-instance-selector --service eks --vcpus-min 8 --memory-min 16
a1.2xlarge
a1.4xlarge
a1.metal
c4.4xlarge
c4.8xlarge
c5.12xlarge
c5.18xlarge
c5.24xlarge
c5.2xlarge
c5.4xlarge
c5.9xlarge
c5.metal
```

API サーバーの負荷を軽減するためにより大きなノードを優先する

使用するインスタンスタイプを決定する場合、Kubernetes コントロールプレーンの負荷が少なくなります。kubelets と DaemonSets の実行が少なくなるためです。ただし、大きなノードは小さなノードのように完全には使用できない場合があります。ノードサイズは、ワークロードの可用性とスケール要件に基づいて評価する必要があります。

3 つの u-24tb1.metal インスタンス (24 TB のメモリと 448 コア) を持つクラスターには 3 つの kubelets があり、デフォルトでノードあたり 110 ポッドに制限されます。ポッドがそれぞれ 4 つのコアを使用している場合は、これが想定される可能性があります (4 つのコア x 110 = 440 コア/ノード)。3 ノードクラスターでは、1 つのインスタンスの停止がクラスターの 1/3 に影響を与える可能性があるため、インスタンスインシデントを処理する機能は低くなります。Kubernetes スケジューラがワークロードを適切に配置できるように、ワークロードにノード要件とポッドスプレッドを指定する必要があります。

ワークロードは、必要なリソースと、テイント、許容範囲、PodTopologySpread を介して必要な可用性を定義する必要があります。コントロールプレーンの負荷を軽減し、運用を削減し、コストを削減するには、完全に活用でき、可用性の目標を達成できる最大のノードを優先する必要があります。

リソースが利用可能な場合、Kubernetes スケジューラは自動的にワークロードをアベイラビリティーゾーンとホストに分散しようとします。使用可能な容量がない場合、Kubernetes Cluster

Autoscaler は各アベイラビリティーゾーンにノードを均等に追加しようとします。Karpenter は、ワークロードが他の要件を指定しない限り、できるだけ迅速かつ安価にノードを追加しようとします。

スケジューラと新しいノードを使用してワークロードをアベイラビリティーゾーンに分散させるには、topologySpreadConstraints を使用する必要があります。

```
spec:
  topologySpreadConstraints:
    - maxSkew: 3
      topologyKey: "topology.kubernetes.io/zone"
      whenUnsatisfiable: ScheduleAnyway
      labelSelector:
        matchLabels:
          dev: my-deployment
    - maxSkew: 2
      topologyKey: "kubernetes.io/hostname"
      whenUnsatisfiable: ScheduleAnyway
      labelSelector:
        matchLabels:
          dev: my-deployment
```

ワークロードのパフォーマンスを一定にするために同様のノードサイズを使用する

ワークロードは、一貫したパフォーマンスと予測可能なスケーリングを可能にするために実行する必要があるノードのサイズを定義する必要があります。5 億 CPU をリクエストするワークロードは、4 コアのインスタンスと 16 コアのインスタンスで動作が異なります。T シリーズインスタンスなどのバースト CPU CPUs。

ワークロードの一貫したパフォーマンスを確保するために、ワークロードはサポートされている Karpenter ラベルを使用して特定のインスタンスサイズをターゲットにできます。

```
kind: deployment
...
spec:
  template:
    spec:
    containers:
    nodeSelector:
```

```
      karpenter.k8s.aws/instance-size: 8xlarge
```

Kubernetes Cluster Autoscaler を使用してクラスターでスケジュールされているワークロードは、ラベルマッチングに基づいてノードセレクタをノードグループと一致させる必要があります。

```
spec:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
        - matchExpressions:
          - key: eks.amazonaws.com/nodegroup
            operator: In
            values:
            - 8-core-node-group    # match your node group name
```

コンピューティングリソースを効率的に使用する

コンピューティングリソースには、EC2 インスタンスとアベイラビリティーゾーンが含まれます。コンピューティングリソースを効果的に使用すると、スケーラビリティ、可用性、パフォーマンスが向上し、総コストを削減できます。複数のアプリケーションを持つ Auto Scaling 環境では、効率的なリソース使用量を予測することは非常に困難です。Karpenter は、ワークロードのニーズに基づいてオンデマンドでインスタンスをプロビジョニングし、使用率と柔軟性を最大化するために作成されました。

Karpenter を使用すると、ワークロードはまずノードグループを作成したり、特定のノードのラベルテイントを設定したりすることなく、必要なコンピューティングリソースのタイプを宣言できます。詳細については、Karpenter のベストプラクティスを参照してください。Karpenter プロビジョナーで統合を有効にして、使用率の低いノードを置き換えることを検討してください。

Amazon マシンイメージ (AMI) の更新を自動化する

ワーカーノードコンポーネントを最新の状態に保つことで、Kubernetes API と互換性のある最新のセキュリティパッチと機能を確保できます。kubelet の更新は Kubernetes 機能の最も重要なコンポーネントですが、OS、カーネル、およびローカルにインストールされたアプリケーションパッチを自動化することで、スケーリングに伴うメンテナンスを減らすことができます。

ノードイメージには、最新の Amazon EKS 最適化 Amazon Linux 2 または Amazon EKS 最適化 Bottlerocket AMI を使用することをお勧めします。Karpenter は、利用可能な最新の AMI を自動的に使用して、クラスター内に新しいノードをプロビジョニングします。マネージド型ノードグループ

は、ノードグループの更新中に AMI を更新しますが、ノードのプロビジョニング時に AMI ID は更新されません。

Managed Node Groups の場合、Auto Scaling Group (ASG) 起動テンプレートがパッチリリースに使用可能になったら、新しい AMI IDs で更新する必要があります。AMI マイナーバージョン (1.23.5 から 1.24.3 など) は、ノードグループのアップグレードとして EKS コンソールと API で使用できます。パッチリリースバージョン (1.23.5 から 1.23.6 など) は、ノードグループのアップグレードとして表示されません。AMI パッチリリースでノードグループを最新の状態に保つ場合は、新しい起動テンプレートバージョンを作成し、ノードグループにインスタンスを新しい AMI リリースに置き換えさせる必要があります。

利用可能な最新の AMI は、このページから検索することも、AWS CLI を使用することもできます。

```
aws ssm get-parameter \
  --name /aws/service/eks/optimized-ami/1.24/amazon-linux-2/recommended/image_id \
  --query "Parameter.Value" \
  --output text
```

コンテナに複数の EBS ボリュームを使用する

EBS ボリュームには、ボリュームのタイプ (gp3 など) とディスクのサイズに基づいて入出力 (I/O) クォータがあります。アプリケーションがホストと単一の EBS ルートボリュームを共有する場合、ホスト全体のディスククォータを使い果たし、他のアプリケーションが使用可能な容量を待機する可能性があります。アプリケーションは、オーバーレイパーティションにファイルを書き込んだり、ホストからローカルボリュームをマウントしたり、使用するログ記録エージェントに応じて標準出力 (STDOUT) にログアウトしたりすると、ディスクに書き込みます。

ディスク I/O の枯渇を回避するには、2 番目のボリュームをコンテナ状態フォルダ (/run/containerd など) にマウントし、ワークロードストレージに個別の EBS ボリュームを使用し、不要なローカルログ記録を無効にする必要があります。

eksctl を使用して EC2 インスタンスに 2 番目のボリュームをマウントするには、この設定でノードグループを使用できます。

```
managedNodeGroups:
  - name: al2-workers
    amiFamily: AmazonLinux2
    desiredCapacity: 2
    volumeSize: 80
    additionalVolumes:
```

```
      - volumeName: '/dev/sdz'
        volumeSize: 100
    preBootstrapCommands:
    - |
      "systemctl stop containerd"
      "mkfs -t ext4 /dev/nvme1n1"
      "rm -rf /var/lib/containerd/*"
      "mount /dev/nvme1n1 /var/lib/containerd/"
      "systemctl start containerd"
```

terraform を使用してノードグループをプロビジョニングする場合は、「Terraform の EKS 設計図」の例を参照してください。Karpenter を使用してノードをプロビジョニングする場合は、ノードのユーザーデータblockDeviceMappingsで を使用してボリュームを追加できます。

EBS ボリュームをポッドに直接マウントするには、AWS EBS CSI ドライバーを使用し、ストレージクラスでボリュームを消費する必要があります。

```
---
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: ebs-sc
provisioner: ebs.csi.aws.com
volumeBindingMode: WaitForFirstConsumer
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: ebs-claim
spec:
  accessModes:
    - ReadWriteOnce
  storageClassName: ebs-sc
  resources:
    requests:
      storage: 4Gi
---
apiVersion: v1
kind: Pod
metadata:
  name: app
spec:
  containers:
```

```
  - name: app
    image: public.ecr.aws/docker/library/nginx
    volumeMounts:
    - name: persistent-storage
      mountPath: /data
  volumes:
  - name: persistent-storage
    persistentVolumeClaim:
      claimName: ebs-claim
```

ワークロードが EBS ボリュームを使用している場合、EBS アタッチ制限が低いインスタンスを避ける

EBS は、ワークロードに永続的ストレージを持たせる最も簡単な方法の 1 つですが、スケーラビリティにも制限があります。各インスタンスタイプには、アタッチできる EBS ボリュームの最大数があります。ワークロードは、実行するインスタンスタイプを宣言し、Kubernetes テイントを持つ 1 つのインスタンスでレプリカの数を制限する必要があります。

ディスクへの不要なログ記録を無効にする

本番環境でデバッグログ記録を使用してアプリケーションを実行せず、ディスクの読み取りと書き込みを頻繁に行うログ記録を無効にすることで、不要なローカルログ記録を回避できます。Journald は、ログバッファをメモリに保持し、定期的にディスクにフラッシュするローカルログ記録サービスです。Journald は、すべての行をディスクにすぐにログに記録する syslog よりも優先されます。syslog を無効にすると、必要なストレージの合計量も減り、複雑なログローテーションルールが不要になります。syslog を無効にするには、次のスニペットを cloud-init 設定に追加します。

```
runcmd:
  - [ systemctl, disable, --now, syslog.service ]
```

OS 更新速度が必要なときにインスタンスにパッチを適用する

> **Important**
>
> インスタンスのパッチ適用は、必要な場合にのみ行う必要があります。Amazon では、インフラストラクチャをイミュータブルとして扱い、アプリケーションと同じように低い環境を通じて昇格される更新を徹底的にテストすることをお勧めします。このセクションは、それが不可能な場合に適用されます。

コンテナ化されたワークロードを中断することなく、既存の Linux ホストにパッケージをインストールするには数秒かかります。パッケージは、インスタンスを接続、ドレイン、または置き換えることなくインストールおよび検証できます。

インスタンスを置き換えるには、まず新しい AMIs を作成、検証、配布する必要があります。インスタンスには置換が作成されている必要があり、古いインスタンスは接続およびドレインされている必要があります。次に、新しいインスタンスでワークロードを作成し、検証して、パッチを適用する必要があるすべてのインスタンスに対して繰り返す必要があります。ワークロードを中断することなくインスタンスを安全に置き換えるには、数時間、数日、数週間かかります。

Amazon では、自動化された宣言型システムから構築、テスト、昇格されるイミュータブルなインフラストラクチャの使用を推奨していますが、システムへの迅速なパッチ適用が必要な場合は、システムへのパッチ適用を行い、新しい AMIs が利用可能になったら置き換える必要があります。パッチ適用とシステムの置き換えには大きな時間差があるため、必要に応じて AWS Systems Manager Patch Manager を使用してパッチ適用ノードを自動化することをお勧めします。

ノードにパッチを適用すると、AMI の更新後にセキュリティ更新をすばやくロールアウトし、定期的にインスタンスを置き換えることができます。Flatcar Container Linux や Bottlerocket OS などの読み取り専用ルートファイルシステムでオペレーティングシステムを使用している場合は、それらのオペレーティングシステムで動作する更新演算子を使用することをお勧めします。Flatcar Linux 更新演算子と Bottlerocket 更新演算子はインスタンスを再起動して、ノードを自動的に最新の状態に保ちます。

クラスターサービス

クラスターサービスは EKS クラスター内で実行されますが、ユーザーワークロードではありません。Linux サーバーを使用している場合は、ワークロードをサポートするために、NTP、syslog、コンテナランタイムなどのサービスを実行する必要があります。クラスターサービスは似ており、クラスターの自動化と運用に役立つサービスをサポートしています。Kubernetes では、これらは通常 kube-system 名前空間で実行され、一部は DaemonSets として実行されます。

クラスターサービスは稼働時間が長く、停止中やトラブルシューティングに不可欠なことがよくあります。コアクラスターサービスが利用できない場合、停止 (ディスク使用率が高いなど) の回復や防止に役立つデータにアクセスできなくなる可能性があります。これらは、別のノードグループや AWS Fargate などの専用コンピューティングインスタンスで実行する必要があります。これにより、クラスターサービスが、スケールアップ中またはより多くのリソースを使用しているワークロードによって共有インスタンスに影響されないようになります。

CoreDNS のスケーリング

CoreDNS のスケーリングには 2 つの主要なメカニズムがあります。CoreDNS サービスへの呼び出しの数を減らし、レプリカの数を増やします。

ndot を下げて外部クエリを減らす

ndots 設定は、DNS のクエリを回避するのに十分なドメイン名の期間 (a.k.a. "dots") の数を指定します。アプリケーションの ndots 設定が 5 (デフォルト) で、api.example.com (2 つのドット) などの外部ドメインからリソースをリクエストすると、CoreDNS は、より具体的なドメインの /etc/resolv.conf で定義された検索ドメインごとにクエリされます。デフォルトでは、外部リクエストを行う前に次のドメインが検索されます。

```
api.example.<namespace>.svc.cluster.local
api.example.svc.cluster.local
api.example.cluster.local
api.example.<region>.compute.internal
```

namespace と のregion値は、ワークロードの名前空間とコンピューティングリージョンに置き換えられます。クラスターの設定に基づいて、追加の検索ドメインがある場合があります。

CoreDNS へのリクエストの数を減らすには、ワークロードの ndots オプションを下げるか、末尾にを含めてドメインリクエストを完全に認定します (例: `api.example.com.`)。ワークロードが DNS 経由で外部サービスに接続する場合は、ワークロードが不要なクラスター DNS クエリをクラスター内で実行しないように、ndot を 2 に設定することをお勧めします。ワークロードがクラスター内のサービスへのアクセスを必要としない場合は、別の DNS サーバーと検索ドメインを設定できます。

```
spec:
  dnsPolicy: "None"
  dnsConfig:
    options:
      - name: ndots
        value: "2"
      - name: edns0
```

ndot を低すぎる値に下げた場合、または接続するドメインに十分な特異度 (末尾の を含む) が含まれていない場合、DNS ルックアップは失敗する可能性があります。この設定がワークロードにどのように影響するかをテストしてください。

CoreDNS を水平方向にスケールする

CoreDNS インスタンスは、デプロイにレプリカを追加することでスケールできます。NodeLocal DNS またはクラスター比例オートスケーラーを使用して CoreDNS をスケールすることをお勧めします。

NodeLocal DNS では、クラスター内のより多くのコンピューティングリソースを必要とする DaemonSet としてノードごとに 1 つのインスタンスを実行する必要がありますが、DNS リクエストの失敗を回避し、クラスター内の DNS クエリの応答時間を短縮できます。クラスター比例オートスケーラーは、クラスター内のノードまたはコアの数に基づいて CoreDNS をスケーリングします。これはクエリをリクエストするための直接的な相関関係ではありませんが、ワークロードとクラスターサイズによっては便利です。デフォルトの比例スケールでは、クラスター内の 256 コアまたは 16 ノードのいずれか早い方ごとにレプリカを追加します。

CoreDNS EKS アドオンを使用する場合は、自動スケーリングオプションを有効にすることを検討してください。CoreDNS オートスケーラーは、(ノード÷16) または (CPU コア÷256) を最大とする式を使用してノード数と CPU コアをモニタリングすることで CoreDNS レプリカの数を動的に調整し、必要に応じてすぐにスケールアップし、安定性を維持するために徐々にスケールダウンします。

Kubernetes メトリクスサーバーを垂直方向にスケールする

Kubernetes メトリクスサーバーは、水平スケーリングと垂直スケーリングをサポートしています。Metrics Server を水平方向にスケーリングすることで可用性は高くなりますが、より多くのクラスターメトリクスを処理するために水平方向にスケーリングされることはありません。ノードと収集されたメトリクスがクラスターに追加されると、メトリクスサーバーは推奨事項に基づいて垂直方向にスケールする必要があります。

Metrics Server は、収集、集約、処理するデータをメモリに保持します。クラスターが大きくなると、Metrics Server が保存するデータの量が増えます。大規模なクラスターでは、メトリクスサーバーは、デフォルトのインストールで指定されたメモリと CPU 予約よりも多くのコンピューティングリソースを必要とします。Vertical Pod Autoscaler (VPA) または Addon Resizer を使用して、メトリクスサーバーをスケールできます。Addon Resizer はワーカーノードに比例して垂直方向にスケールし、VPA は CPU とメモリの使用量に基づいてスケールします。

CoreDNS lameduck の期間

ポッドは名前解決に kube-dnsサービスを使用します。Kubernetes は送信先 NAT (DNAT) を使用して、ノードから CoreDNS バックエンドポッドにkube-dnsトラフィックをリダイレクトします。CoreDNS デプロイをスケールすると、 はノードの iptables ルールとチェーンkube-proxyを更

新して、DNS トラフィックを CoreDNS ポッドにリダイレクトします。CoreDNS をスケールアップするときに新しいエンドポイントを伝播し、スケールダウンするときにルールを削除すると、クラスターのサイズに応じて 1 ~ 10 秒かかることがあります。

この伝達遅延により、CoreDNS ポッドが終了してもノードの iptables ルールが更新されていない場合、DNS ルックアップが失敗する可能性があります。このシナリオでは、ノードは終了した CoreDNS Pod に DNS クエリを送信し続ける場合があります。

CoreDNS ポッドで Lameduck 期間を設定することで、DNS ルックアップの失敗を減らすことができます。 CoreDNS Lameduck モードの間、CoreDNS は処理中のリクエストに引き続き応答します。Lameduck 期間を設定すると、CoreDNS シャットダウンプロセスが遅延し、ノードが iptables ルールとチェーンを更新するために必要な時間を確保できます。

CoreDNS lameduck 期間を 30 秒に設定することをお勧めします。

CoreDNS 準備状況プローブ

CoreDNS の準備状況プローブ`/health`に `/ready` の代わりに を使用することをお勧めします。

ラメダック期間を 30 秒に設定し、ポッド終了前にノードの iptables ルールを更新するのに十分な時間を確保し、 CoreDNS 準備状況プローブ`/ready/health`の代わりに を使用することで、起動時に CoreDNS ポッドが DNS リクエストに迅速に応答する準備が完全に整います。

```
readinessProbe:
  httpGet:
    path: /ready
    port: 8181
    scheme: HTTP
```

CoreDNS Ready プラグインの詳細については、https://coredns.io/plugins/ready/ を参照してください。

エージェントのログ記録とモニタリング

エージェントが API サーバーにクエリを実行してワークロードメタデータでログとメトリクスを強化するため、エージェントのログ記録とモニタリングによってクラスターコントロールプレーンに多大な負荷がかかる可能性があります。ノード上のエージェントは、コンテナやプロセス名などを表示するために、ローカルノードリソースにのみアクセスできます。API サーバーをクエリすると、Kubernetes デプロイ名やラベルなどの詳細を追加できます。これはトラブルシューティングに非常に役立ちますが、スケーリングに悪影響を及ぼす可能性があります。

ログ記録とモニタリングにはさまざまなオプションがあるため、すべてのプロバイダーの例を表示することはできません。fluentbit では、Use_Kubelet を有効にして Kubernetes API Server ではなくローカル kubelet からメタデータを取得し、データをキャッシュできるときの繰り返し呼び出しを減らす数値 (60 など) Kube_Meta_Cache_TTLに設定することをお勧めします。

スケーリングのモニタリングとログ記録には、次の 2 つの一般的なオプションがあります。

- 統合を無効にする
- サンプリングとフィルタリング

ログメタデータが失われるため、統合を無効にすることはオプションではないことがよくあります。これにより、API スケーリングの問題はなくなりますが、必要に応じて必要なメタデータがないため、他の問題が発生します。

サンプリングとフィルタリングは、収集されるメトリクスとログの数を減らします。これにより、Kubernetes API へのリクエストの量が減り、収集されるメトリクスとログに必要なストレージの量が減ります。ストレージコストを削減することで、システム全体のコストを削減できます。

サンプリングを設定する機能はエージェントソフトウェアに依存し、さまざまな取り込みポイントに実装できます。API サーバーの呼び出しが発生する可能性が高いため、可能な限りエージェントの近くにサンプリングを追加することが重要です。サンプリングサポートの詳細については、プロバイダーにお問い合わせください。

CloudWatch と CloudWatch Logs を使用している場合は、ドキュメントで説明されているパターンを使用してエージェントフィルタリングを追加できます。

ログとメトリクスが失われないようにするには、受信エンドポイントで停止した場合にデータをバッファできるシステムにデータを送信する必要があります。fluentbit を使用すると、Amazon Kinesis Data Firehose を使用してデータを一時的に保持できるため、最終的なデータストレージの場所が過負荷になる可能性が低くなります。

ワークロード

ワークロードは、クラスターのサイズに影響します。Kubernetes APIs を大量に使用するワークロードは、1 つのクラスターに保持できるワークロードの合計量を制限しますが、負荷を軽減するために変更できるデフォルトがいくつかあります。

Kubernetes クラスターのワークロードは、Kubernetes API と統合する機能 (Secrets や ServiceAccounts など) にアクセスできますが、これらの機能は必ずしも必要ではなく、使用されて

いない場合は無効にする必要があります。ワークロードのアクセスと Kubernetes コントロールプレーンへの依存を制限すると、クラスターで実行できるワークロードの数が増え、ワークロードへの不要なアクセスが削除され、最小特権のプラクティスが実装されるため、クラスターのセキュリティが向上します。詳細については、セキュリティのベストプラクティスをお読みください。

ポッドネットワークに IPv6 を使用する

VPC を IPv4 から IPv6 に移行できないため、クラスターをプロビジョニングする前に IPv6 を有効にすることが重要です。VPC で IPv6 を有効にしても、IPv6 を使用する必要はなく、ポッドやサービスで IPv6 を使用している場合でもIPv4 アドレスとの間でトラフィックをルーティングできます。詳細については、EKS ネットワークのベストプラクティスを参照してください。

クラスターで IPv6 を使用すると、最も一般的なクラスターとワークロードのスケーリング制限の一部を回避できます。IPv6 は、使用可能な IP アドレスがないため、ポッドとノードを作成できない IP アドレスの枯渇を回避します。また、ノードあたりの ENI アタッチメントの数を減らすことで、ポッドが IP アドレスをより速く受け取るため、ノードごとのパフォーマンスも向上します。VPC CNI で IPv4 プレフィックスモードを使用することで、同様のノードパフォーマンスを実現できますが、VPC で使用可能な十分な IP アドレスがあることを確認する必要があります。

名前空間あたりのサービス数の制限

名前空間内のサービスの最大数は 5,000 で、クラスター内のサービスの最大数は 10,000 です。ワークロードとサービスを整理し、パフォーマンスを向上させ、名前空間スコープリソースへのカスケードの影響を回避するには、名前空間あたりのサービス数を 500 に制限することをお勧めします。

kube-proxy を使用してノードごとに作成される IP テーブルルールの数は、クラスター内のサービスの合計数に応じて増加します。数千の IP テーブルルールを生成し、それらのルールを介してパケットをルーティングすると、ノードのパフォーマンスに影響し、ネットワークレイテンシーが増加します。

名前空間あたりのサービス数が 500 未満である限り、単一のアプリケーション環境を含む Kubernetes 名前空間を作成します。これにより、サービス検出の制限を回避するためにサービス検出を小さく保ち、サービス名の衝突を回避することもできます。アプリケーション環境 (開発、テスト、製品など) では、名前空間の代わりに個別の EKS クラスターを使用する必要があります。

Elastic Load Balancer のクォータを理解する

サービスを作成するときは、使用するロードバランシングのタイプ (Network Load Balancer (NLB) や Application Load Balancer (ALB) など) を検討してください。ロードバランサーのタイプごとに機能やクォータが異なります。デフォルトのクォータの一部は調整できますが、変更できないクォータ

の上限があります。アカウントのクォータと使用状況を表示するには、AWS コンソールで Service Quotas ダッシュボードを表示します。

たとえば、デフォルトの ALB ターゲットは 1000 です。エンドポイントが 1000 を超えるサービスがある場合は、クォータを増やすか、サービスを複数の ALBs に分割するか、Kubernetes Ingress を使用する必要があります。デフォルトの NLB ターゲットは 3000 ですが、AZ あたり 500 ターゲットに制限されています。クラスターが NLB サービスに対して 500 個を超えるポッドを実行している場合は、複数の AZs を使用するか、クォータ制限の引き上げをリクエストする必要があります。

サービスに結合されたロードバランサーを使用する代わりに、イングレスコントローラーを使用します。AWS Load Balancer コントローラーはイングレスリソースの ALBs を作成できますが、クラスターで専用コントローラーを実行することを検討することもできます。クラスター内イングレスコントローラーを使用すると、クラスター内でリバースプロキシを実行して、単一のロードバランサーから複数の Kubernetes サービスを公開できます。コントローラーには Gateway API のサポートなど、さまざまな機能があり、ワークロードの数と規模に応じてメリットが得られる場合があります。

Route 53、Global Accelerator、または CloudFront を使用する

複数のロードバランサーを使用するサービスを単一のエンドポイントとして利用できるようにするには、Amazon CloudFront、AWS Global Accelerator、または Amazon Route 53 を使用して、すべてのロードバランサーを単一の顧客向けエンドポイントとして公開する必要があります。各オプションにはさまざまな利点があり、ニーズに応じて個別に使用することも、一緒に使用することもできます。

Route 53 は、共通の名前で複数のロードバランサーを公開し、割り当てられた重みに基づいて各ロードバランサーにトラフィックを送信できます。DNS の重みの詳細については、 ドキュメントを参照してください。また、AWS Load Balancer Controller ドキュメントの Kubernetes 外部 DNS コントローラーを使用してそれらを実装する方法も確認できます。

Global Accelerator は、リクエスト IP アドレスに基づいて、最も近いリージョンにワークロードをルーティングできます。これは、複数のリージョンにデプロイされるワークロードに便利ですが、1 つのリージョンの 1 つのクラスターへのルーティングは改善されません。Route 53 を Global Accelerator と組み合わせて使用すると、AZ が利用できない場合はヘルスチェックや自動フェイルオーバーなどの追加の利点があります。このブログ記事では、Route 53 で Global Accelerator を使用する例を示しています。

CloudFront は、Route 53 および Global Accelerator で使用するか、単独で複数の宛先にトラフィックをルーティングできます。CloudFront は、オリジンソースから提供されるアセットをキャッシュするため、サービス内容に応じて帯域幅要件が軽減される可能性があります。

エンドポイントの代わりに EndpointSlices を使用する

サービスラベルに一致するポッドを検出するときは、エンドポイントの代わりに EndpointSlices を使用する必要があります。エンドポイントは、サービスを小規模に公開する簡単な方法ですが、自動スケーリングまたは更新を行う大規模なサービスでは、Kubernetes コントロールプレーンで大量のトラフィックが発生します。EndpointSlices には、トポロジ対応ヒントなどを有効にする自動グループ化があります。

デフォルトでは、すべてのコントローラーが EndpointSlices を使用するわけではありません。コントローラーの設定を確認し、必要に応じて有効にする必要があります。AWS Load Balancer Controller では、`--enable-endpoint-slices`オプションの フラグを有効にして EndpointSlices を使用する必要があります。

可能であれば、イミュータブルシークレットと外部シークレットを使用する

kubelet は、そのノード上のポッドのボリュームで使用されるシークレットの現在のキーと値のキャッシュを保持します。kubelet は、変更を検出するためにシークレットに監視を設定します。クラスターがスケールすると、監視の数が増えると API サーバーのパフォーマンスに悪影響を及ぼす可能性があります。

Secrets のウォッチの数を減らすには、次の 2 つの戦略があります。

- Kubernetes リソースへのアクセスを必要としないアプリケーションの場合、automountServiceAccountToken: false を設定することで、サービスアカウントシークレットの自動マウントを無効にすることができます。
- アプリケーションのシークレットが静的で、今後変更されない場合は、シークレットをイミュータブルとしてマークします。kubelet は、イミュータブルシークレットの API 監視を維持しません。

サービスアカウントのポッドへの自動マウントを無効にするには、ワークロードで次の設定を使用します。特定のワークロードにサービスアカウントが必要な場合は、これらの設定を上書きできます。

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: app
automountServiceAccountToken: true
```

制限の 10,000 を超える前に、クラスター内のシークレットの数をモニタリングします。次のコマンドを使用すると、クラスター内のシークレットの合計数を確認できます。この制限は、クラスターモニタリングツールでモニタリングする必要があります。

```
kubectl get secrets -A | wc -l
```

この制限に達する前に、クラスター管理者に警告するようにモニタリングを設定する必要があります。Secrets Store CSI ドライバーで AWS Key Management Service (AWS KMS) や Hashicorp Vault などの外部シークレット管理オプションを使用することを検討してください。 https://secrets-store-csi-driver.sigs.k8s.io/

デプロイ履歴の制限

古いオブジェクトはクラスター内で追跡されるため、ポッドの作成、更新、削除に時間がかかる場合があります。デプロイ`revisionHistoryLimit`の を減らして、古い ReplicaSets をクリーンアップできます。これにより、Kubernetes Controller Manager によって追跡されるオブジェクトの合計量が減ります。10 のデプロイのデフォルトの履歴制限。

クラスターが CronJobs またはその他のメカニズムを使用して多数のジョブオブジェクトを作成する場合は、 `ttlSecondsAfterFinished`設定を使用してクラスター内の古いポッドを自動的にクリーンアップする必要があります。これにより、指定した時間が経過すると、正常に実行されたジョブがジョブ履歴から削除されます。

enableServiceLinks をデフォルトで無効にする

ポッドがノードで実行されると、kubelet はアクティブなサービスごとに一連の環境変数を追加します。Linux プロセスには環境の最大サイズがあり、名前空間にサービスが多すぎる場合に到達できます。名前空間あたりのサービス数は 5,000 を超えることはできません。その後、サービス環境変数の数はシェルの制限を超過し、起動時にポッドがクラッシュします。

ポッドがサービス検出にサービス環境変数を使用しない理由は他にもあります。環境変数名の競合、漏洩しているサービス名、および合計環境サイズがいくつかあります。サービスエンドポイントの検出には CoreDNS を使用する必要があります。

リソースごとに動的アドミッションウェブフックを制限する

動的アドミッションウェブフックには、アドミッションウェブフックとミューテーションウェブフックが含まれます。これらは、リソースが Kubernetes API に送信されるときに順番に呼び出される Kubernetes コントロールプレーンの一部ではない API エンドポイントです。各ウェブフックのデ

フォルトのタイムアウトは 10 秒で、複数のウェブフックまたはいずれかのタイムアウトがある場合、API リクエストにかかる時間を増やすことができます。

特に AZ インシデント中に、ウェブフックの可用性が高く、 failurePolicy が適切に設定され、リソースを拒否するか、障害を無視することを確認します。--dry-run kubectl コマンドがウェブフックをバイパスできるようにすることで、不要な場合はウェブフックを呼び出さないでください。

```
apiVersion: admission.k8s.io/v1
kind: AdmissionReview
request:
  dryRun: False
```

ウェブフックをミューテーションすると、リソースが頻繁に連続して変更される可能性があります。5 つの変更ウェブフックがあり、50 のリソースをデプロイする場合、etd は、変更したリソースの古いバージョンを削除するために圧縮が 5 分ごとに実行されるまで、各リソースのすべてのバージョンを保存します。etcd が置き換えられたリソースを削除するこのシナリオでは、etcd から 200 個のリソースバージョンが削除され、リソースのサイズによっては、デフラグが 15 分ごとに実行されるまで、etcd ホストでかなりのスペースを使用する場合があります。

このデフラグメンテーションにより etcd が一時停止し、Kubernetes API とコントローラーに他の影響を与える可能性があります。大規模なリソースを頻繁に変更したり、数百のリソースをすばやく連続して変更したりすることは避けてください。

複数のクラスター間でワークロードを比較する

同様のパフォーマンスを持つ必要があるが、そうでないクラスターが 2 つある場合は、メトリクスを比較して理由を特定してみてください。

例えば、クラスターのレイテンシーを比較するのは一般的な問題です。これは通常、API リクエストの量の違いが原因です。次の CloudWatch LogInsight クエリを実行して、違いを理解できます。

```
filter @logStream like "kube-apiserver-audit"
| stats count(*) as cnt by objectRef.apiGroup, objectRef.apiVersion,
 objectRef.resource, userAgent, verb, responseStatus.code
| sort cnt desc
| limit 1000
```

フィルターを追加して絞り込むことができます。例えば、 からのすべてのリストリクエストに焦点を当てますfoo。

```
filter @logStream like "kube-apiserver-audit"
| filter verb = "list"
| filter user.username like "foo"
| stats count(*) as cnt by objectRef.apiGroup, objectRef.apiVersion,
 objectRef.resource, responseStatus.code
| sort cnt desc
| limit 1000
```

Kubernetes スケーリング理論

ノードとチャーンレート

Kubernetes のスケーラビリティについて議論するときは、多くの場合、1 つのクラスターに存在するノードの数の観点から説明します。興味深いことに、これはスケーラビリティを理解するために最も有用なメトリクスになることはほとんどありません。たとえば、ポッド数が多いが固定数の 5,000 個のノードクラスターでは、初期設定後にコントロールプレーンに大きなストレスがかかりません。ただし、1,000 個のノードクラスターを使用し、1 分以内に 10,000 個の有効期間の短いジョブを作成しようとすると、コントロールプレーンに大きな負担がかかります。

単純にノード数を使用してスケーリングを理解すると、誤解を招く可能性があります。特定の期間内に発生する変化率について考えることをお勧めします (Prometheus クエリがデフォルトで通常使用するものであるため、このディスカッションでは 5 分間隔で行います) 。変更率の観点から問題をフレーミングすることで、希望する規模を達成するために何を調整すべきかをよりよく理解できる理由を見てみましょう。

1 秒あたりのクエリ数で考える

Kubernetes には、Kubernetes チェーン内の次のリンクが圧倒されないように、Kubelet、Scheduler、Kube Controller Manager、API サーバーなど、コンポーネントごとに多くの保護メカニズムがあります。たとえば、Kubelet には API サーバーへの呼び出しを一定のレートでスロットリングするフラグがあります。これらの保護メカニズムは、通常、1 秒あたりに許可されるクエリまたは QPS で表されますが、必ずしも表されるわけではありません。

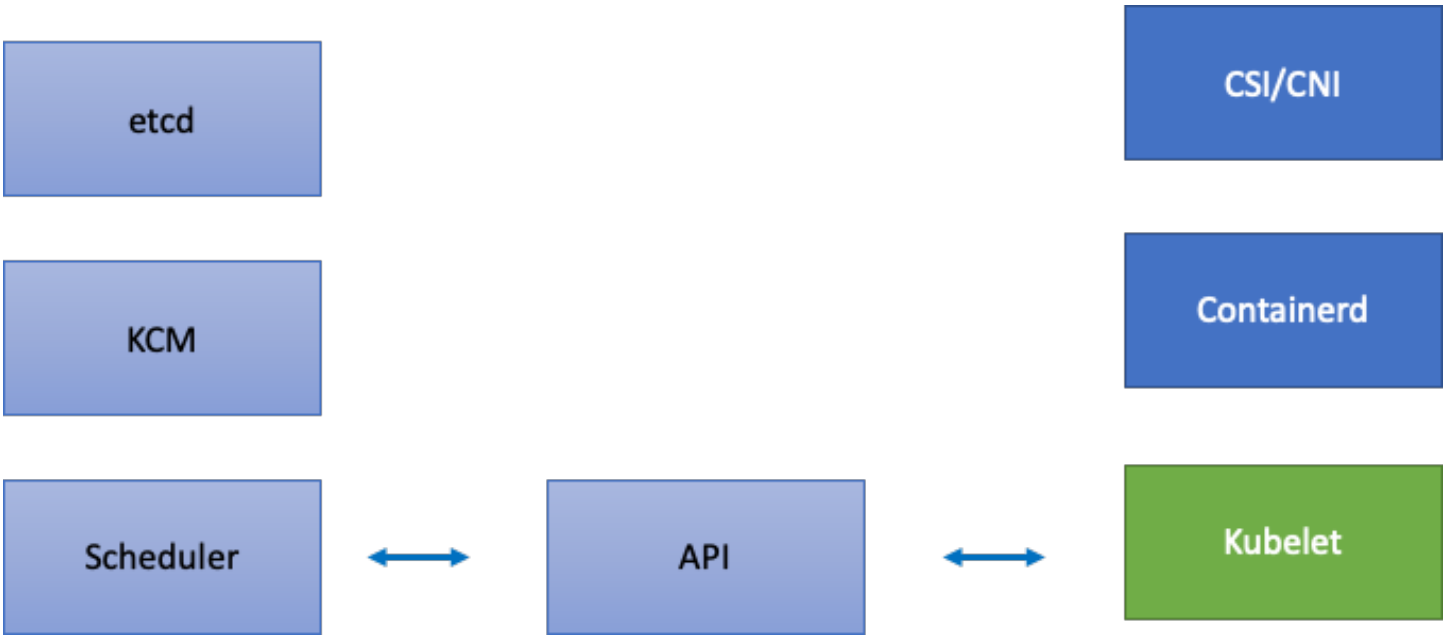
これらの QPS 設定を変更するときは、細心の注意が必要です。Kubelet での 1 秒あたりのクエリなど、1 つのボトルネックを削除すると、他のダウンストリームコンポーネントに影響します。これにより、システムが特定のレートを超える可能性があるため、Kubernetes でワークロードを正常にスケーリングするには、サービスチェーンの各部分を理解してモニタリングすることが重要です。

> **Note**
>
> API サーバーには、API Priority and Fairness が導入された、より複雑なシステムがあります。これについては、別途説明します。

> **Note**
>
> 注意点として、一部のメトリクスは適切であるように見えますが、実際には別の何かを測定しています。例として、 は Kubelet から apiserver へのリクエストの数ではなく、Kubelet のメトリクスサーバーにのみ`kubelet_http_inflight_requests`関連します。これにより、Kubelet の QPS フラグの設定が誤る可能性があります。特定の Kubelet の監査ログに対するクエリは、メトリクスをチェックするより信頼性の高い方法です。

分散コンポーネントのスケーリング

EKS はマネージドサービスであるため、Kubernetes コンポーネントを etcd、Kube Controller Manager、スケジューラ (図の左側) を含む AWS マネージドコンポーネントと、Kubelet、Container Runtime、ネットワークおよびストレージドライバー (図の右側) などの AWS APIs を呼び出すさまざまな演算子などのカスタマー設定可能なコンポーネントという 2 つのカテゴリに分割しましょう。API サーバーの優先度と公平性の設定はお客様が設定できるため、AWS が管理している場合でも API サーバーを中間のままにします。



アップストリームとダウンストリームのボトルネック

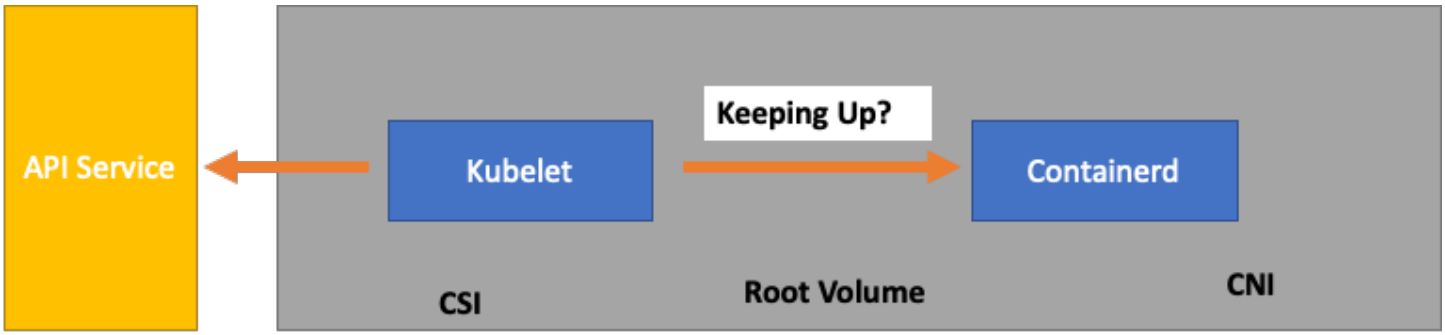
各サービスをモニタリングするときは、ボトルネックを探すために、双方向のメトリクスを確認することが重要です。これを行う方法については、例として Kubelet を使用します。Kubelet は API サーバーとコンテナランタイムの両方と通信します。いずれかのコンポーネントで問題が発生しているかどうかを検出するには、どのように、何をモニタリングする必要がありますか?

ノードあたりのポッド数

ノードで実行できるポッドの数などのスケーリング番号を見ると、アップストリームが顔の値でサポートするノードごとに 110 個のポッドを取ることができます。

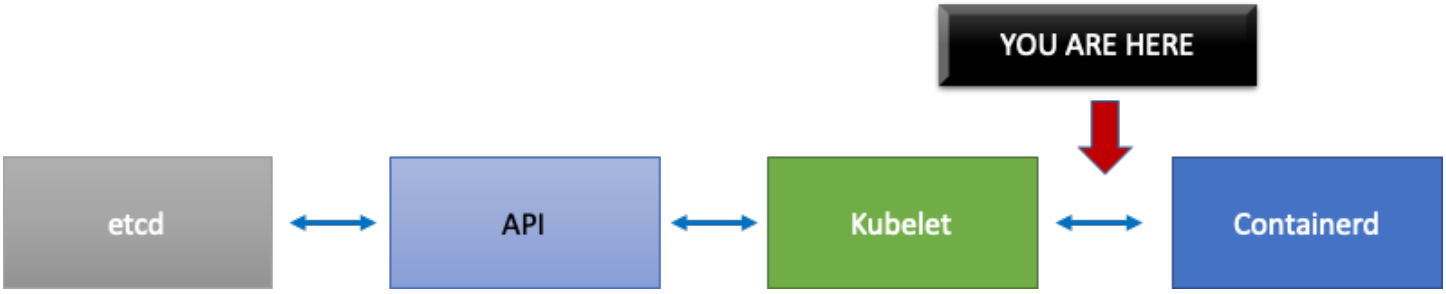
> **Note**
>
> https://kubernetes.io/docs/setup/best-practices/cluster-large/

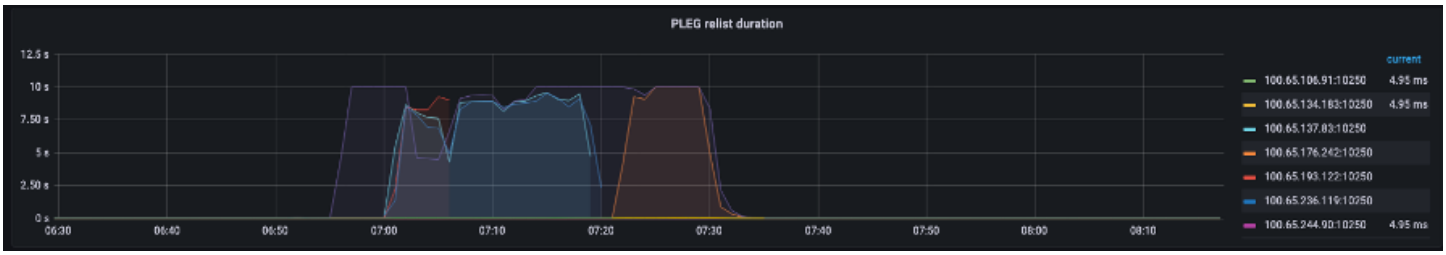
ただし、ワークロードは、アップストリームのスケーラビリティテストでテストされたものよりも複雑である可能性があります。本番環境で実行するポッドの数を確実に処理できるように、Kubelet が Containerd ランタイムを「維持」していることを確認します。



シンプルにするために、Kubelet はコンテナランタイム (この場合は Containerd) からポッドのステータスを取得しています。ステータスの変更が速すぎるポッドが多すぎた場合はどうなりますか? 変更率が高すぎると、〔コンテナランタイムへの] リクエストがタイムアウトする可能性があります。

> **Note**
>
> Kubernetes は常に進化しており、このサブシステムは現在変更中です。https://github.com/kubernetes/enhancements/issues/3386





上記のグラフには、ポッドライフサイクルイベント生成期間メトリクスのタイムアウト値に達したことを示すフラットな線が表示されます。これを独自のクラスターで表示する場合は、次の PromQL 構文を使用できます。

```
increase(kubelet_pleg_relist_duration_seconds_bucket{instance="$instance"}
[$__rate_interval])
```

このタイムアウト動作を目撃した場合、ノードが可能な制限を超えてプッシュされたことがわかります。先に進む前に、タイムアウトの原因を修正する必要があります。これは、ノードあたりのポッド数を減らすか、大量の再試行を引き起こしている可能性のあるエラー (解約率に影響する) を探すことで実現できます。重要なポイントとして、ノードが割り当てられたポッドの解約率と固定数を使用して解約率を処理できるかどうかを把握するには、メトリクスが最善の方法です。

メトリクスによるスケーリング

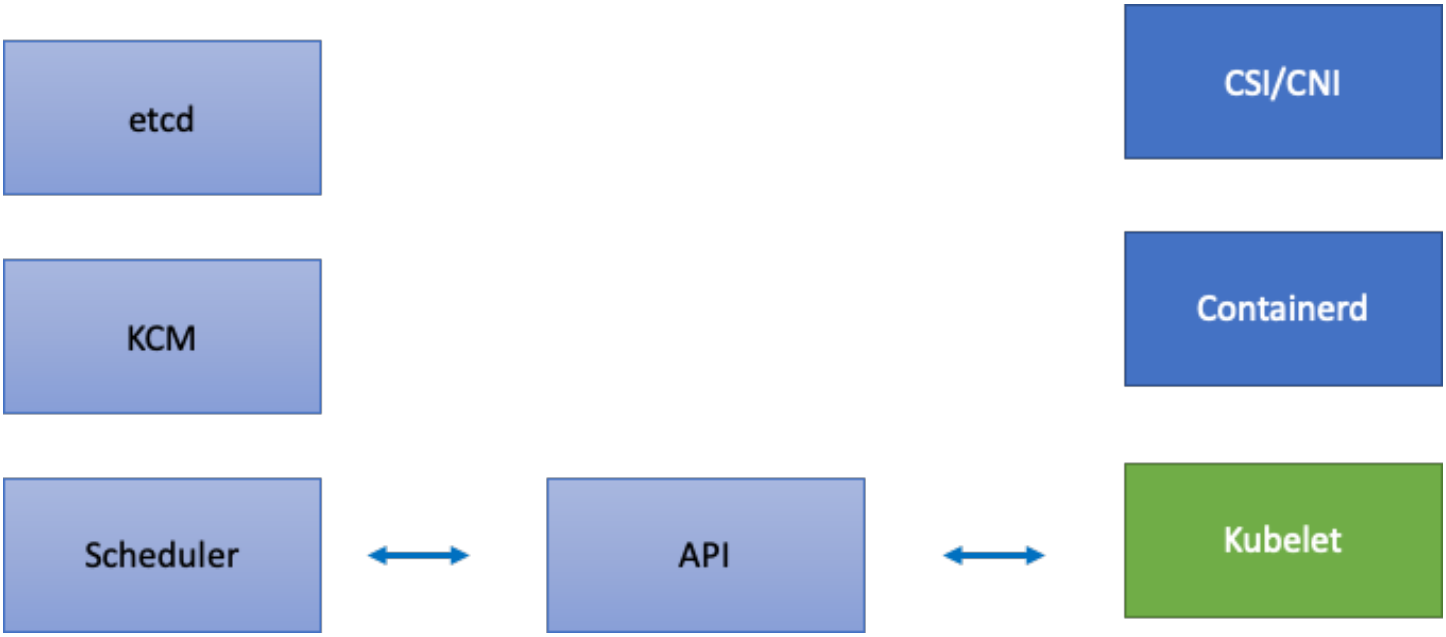
メトリクスを使用してシステムを最適化するという概念は古いものですが、Kubernetes ジャーニーを開始すると見落とされがちです。特定の数値 (ノードあたり 110 個のポッド) に焦点を当てるのではなく、システム内のボトルネックを見つけるのに役立つメトリクスを見つけることに重点を置いています。これらのメトリクスの適切なしきい値を理解することで、システムが最適に構成されているという高い信頼性が得られます。

変更の影響

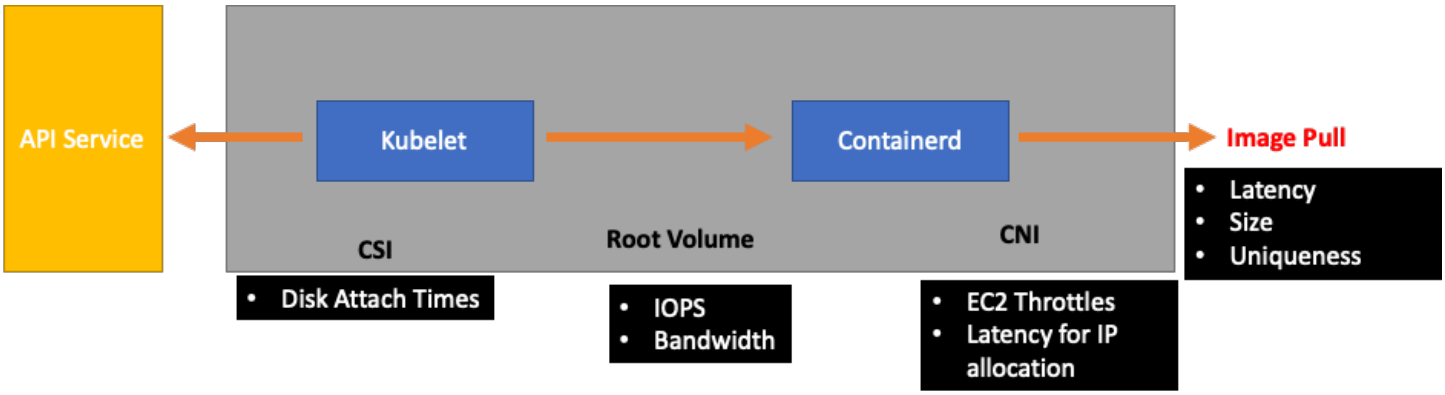
問題が発生する可能性がある一般的なパターンは、疑わしい最初のメトリクスまたはログエラーに焦点を当てることです。以前に Kubelet がタイムアウトしていたことがわかったら、Kubelet が送信で

きる 1 秒あたりのレートを増やすなど、ランダムなことを試すことができます。ただし、最初に見つかったエラーの下流にあるすべての全体像を見るのが賢明です。目的とデータに基づいて各変更を行います。

Kubelet のダウンストリームは、Containerd ランタイム (ポッドエラー) 、ストレージドライバー (CSI) などの DaemonSets、EC2 API と通信するネットワークドライバー (CNI) などです。



前述の Kubelet がランタイムに追いついていない例を続行しましょう。ノードを密に格納してエラーをトリガーできる点がいくつかあります。



ワークロードに適したノードサイズを設計する場合、これらはeasy-to-overlookシグナルであり、システムに不要な負荷がかかり、スケールとパフォーマンスの両方が制限されている可能性があります。

不要なエラーのコスト

Kubernetes コントローラーは、エラー状態が発生した場合の再試行に優れていますが、これにはコストがかかります。これらの再試行により、Kube Controller Manager などのコンポーネントへの負荷が増加する可能性があります。このようなエラーをモニタリングすることは、スケールテストの重要なテナントです。

エラーが少なくなると、システム内の問題を見つけやすくなります。大規模なオペレーション (アップグレードなど) の前にクラスターにエラーがないことを定期的に確認することで、予期しないイベントが発生した場合のトラブルシューティングログを簡素化できます。

ビューの展開

何千ものノードを持つ大規模なクラスターでは、ボトルネックを個別に探したくないと考えています。PromQL では、topk という関数を使用してデータセット内の最大値を見つけることができます。K は必要な項目数を配置する変数です。ここでは、3 つのノードを使用して、クラスター内のすべての Kubelets が飽和しているかどうかを把握します。ここまでレイテンシーを調べてきました。次に、Kubelet がイベントを破棄しているかどうかを見てみましょう。

```
topk(3, increase(kubelet_pleg_discard_events{}[$__rate_interval]))
```

このステートメントを破棄します。

- Grafana 変数を使用して`$__rate_interval`、必要な 4 つのサンプルを取得します。これにより、シンプルな変数を使用したモニタリングで複雑なトピックがバイパスされます。
- `topk` は上位の結果のみを提供し、数値 3 はそれらの結果を 3 に制限します。これは、クラスター全体のメトリクスに便利な関数です。
- `{}` にフィルターがないことを知らせます。通常は、スクレイピングルールのジョブ名を入力しますが、これらの名前は異なるため、空白のままにします。

問題を半分に分割する

システムのボトルネックに対処するために、アップストリームまたはダウンストリームに問題があることを示すメトリクスを見つけるアプローチを取ります。これにより、問題を半分に分割できます。また、メトリクスデータの表示方法の中核となる原則でもあります。

このプロセスを開始するのに適した場所は API サーバーです。クライアントアプリケーションまたはコントロールプレーンに問題があるかどうかを確認することができます。

コントロールプレーンのモニタリング

API サーバー

API サーバーを見るときは、その関数の 1 つがインバウンドリクエストをスロットリングしてコントロールプレーンの過負荷を防ぐことを覚えておくことが重要です。API サーバーレベルでボトルネックのように見えるのは、実際にはより深刻な問題から保護することかもしれません。システムを移動するリクエストの量を増やすことの長所と短所を考慮する必要があります。API サーバーの値を増やす必要があるかどうかを判断するために、注意すべき事項の小さなサンプリングを次に示します。

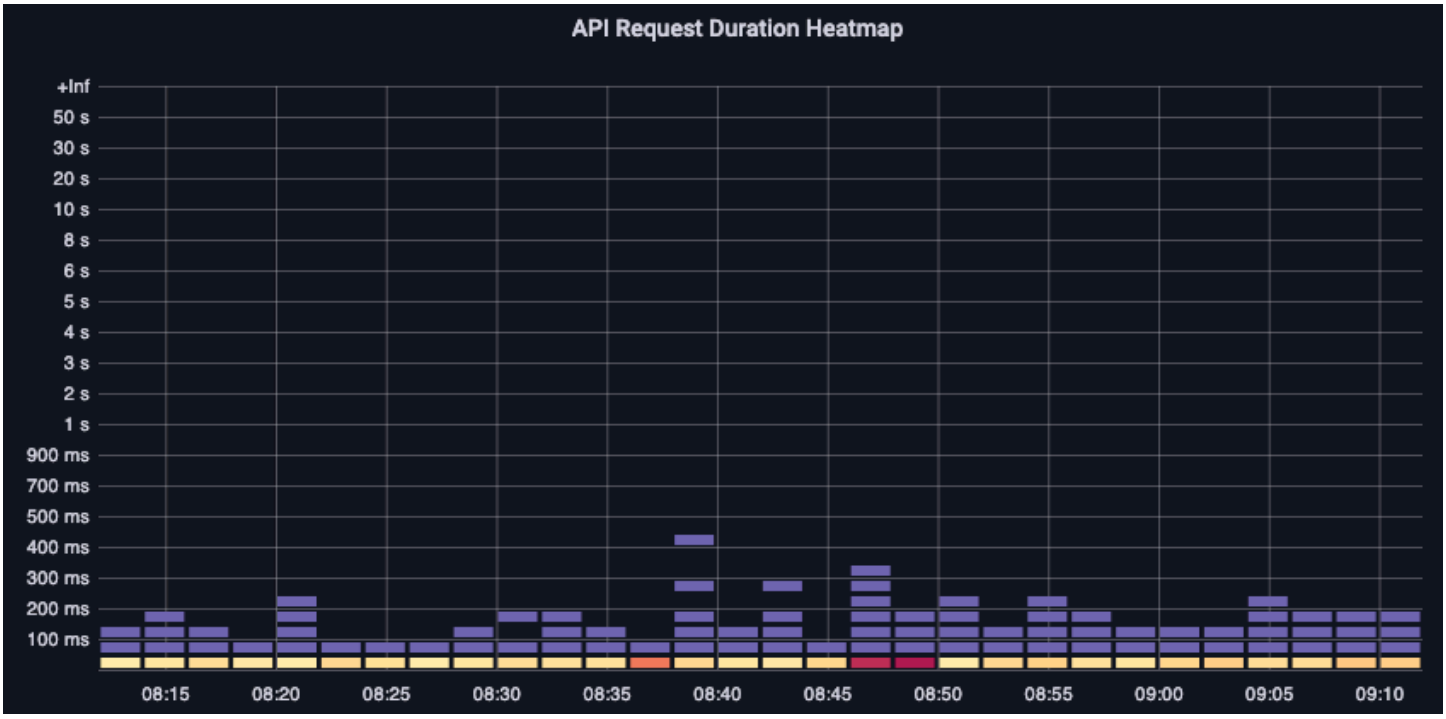
1. システムを通過するリクエストのレイテンシーはどれくらいですか?
2. このレイテンシーは API サーバー自体ですか、それとも etcd のような「ダウンストリーム」ですか?
3. API サーバーキューの深さは、このレイテンシーの要因ですか?
4. API Priority and Fairness (APF) キューは、必要な API コールパターンに対して正しく設定されていますか?

問題はどこにありますか?

まず、 API レイテンシーの メトリクスを使用して、API サーバーがサービスリクエストにどれだけかかるかを把握できます。以下の PromQL と Grafana ヒートマップを使用して、このデータを表示しましょう。

```
max(increase(apiserver_request_duration_seconds_bucket{subresource!
="status",subresource!="token",subresource!="scale",subresource!="/
healthz",subresource!="binding",subresource!="proxy",verb!="WATCH"}[$__rate_interval]))
 by (le)
```

> **Note**
>
> この記事で使用されている API ダッシュボードを使用して API サーバーをモニタリングする方法の詳細については、次のブログを参照してください。



これらのリクエストはすべて 1 秒単位です。これは、コントロールプレーンがリクエストをタイムリーに処理していることを示しています。しかし、そのようにならなかったらどうでしょうか。

上記の API リクエスト期間で使用している形式はヒートマップです。ヒートマップ形式の良い点は、デフォルトで API のタイムアウト値 (60 秒) を示すことです。ただし、実際に知っておく必要があるのは、タイムアウトしきい値に達する前に、この値をどのしきい値にする必要があるかということです。許容されるしきい値の大まかなガイドラインについては、こちらにあるアップストリーム Kubernetes SLO を使用できます。

> **Note**
>
> このステートメントの最大関数に注目しますか? 複数のサーバー (デフォルトでは EKS 上の 2 つの API サーバー) を集約するメトリクスを使用する場合は、それらのサーバーをまとめて平均化しないことが重要です。

非対称トラフィックパターン

1 つの API サーバー [pod] が軽くロードされ、もう 1 つの API サーバーが頻繁にロードされた場合どうなりますか? これら 2 つの数値を一緒に平均すると、何が起こっているのかを誤って解釈する可能性があります。たとえば、3 つの API サーバーがありますが、すべての負荷はこれらの API サー

バーの 1 つにあります。スケールとパフォーマンスの問題を投資するときは、原則として、 etcd サーバーや API サーバーなど、複数のサーバーを持つものをすべて分割する必要があります。



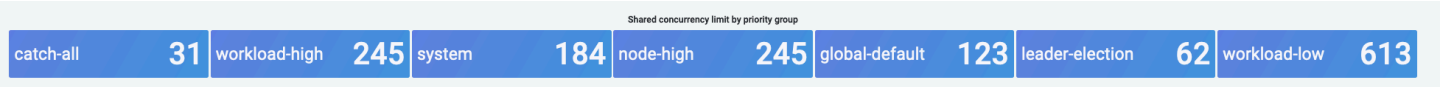
API Priority and Fairness への移行では、システム上のリクエストの総数は、API サーバーがオーバーサブスクライブされているかどうかを確認する 1 つの要素にすぎません。システムは一連のキューで動作するようになったため、これらのキューのいずれかが満杯かどうか、およびそのキューのトラフィックがドロップされているかどうかを確認する必要があります。

次のクエリを使用して、これらのキューを見てみましょう。

```
max without(instance)(apiserver_flowcontrol_nominal_limit_seats{})
```

> **Note**
>
> API A&F の仕組みの詳細については、以下のベストプラクティスガイドを参照してください。

ここでは、クラスターでデフォルトで提供される 7 つの異なる優先度グループを示します。



次に、その優先度グループの何パーセントが使用されているかを確認し、特定の優先度レベルが飽和しているかどうかを理解できるようにします。ワークロードの低いレベルでリクエストをスロットリングするのが望ましいかもしれませんが、リーダー選挙レベルの低下は望ましくないでしょう。

API Priority and Fairness (APF) システムには多くの複雑なオプションがあり、これらのオプションの一部は意図しない結果をもたらす可能性があります。フィールドに表示され

る一般的な問題は、不要なレイテンシーの追加を開始する時点までキューの深さを増やすことです。この問題は、 `apiserver_flowcontrol_current_inqueue_request`メトリクスを使用してモニタリングできます。を使用してドロップを確認できますapiserver_flowcontrol_rejected_requests_total。これらのメトリクスは、バケットが同時実行数を超えた場合、ゼロ以外の値になります。



キューの深さを増やすと、API Server がレイテンシーの大きな原因になる可能性があるため、注意が必要です。作成されたキューの数を慎重に検討することをお勧めします。例えば、EKS システム上の共有数は 600 です。キューの作成数が多すぎると、リーダー選択キューやシステムキューなどのスループットを必要とする重要なキューの共有を減らすことができます。余分なキューを作成すると、キューのサイズを正しく設定することが困難になる可能性があります。

APF で実行できる単純な影響のある変更に集中するには、使用率の低いバケットから共有を取得し、最大使用量のバケットのサイズを増やします。これらのバケット間で共有をインテリジェントに再分散することで、ドロップの可能性を減らすことができます。

詳細については、「EKS ベストプラクティスガイド」の「API Priority and Fairness settings」を参照してください。

API と etcd のレイテンシー

API サーバーのメトリクス/ログを使用して、API サーバーに問題があるか、API サーバーのアップストリーム/ダウンストリームに問題があるか、またはその両方の組み合わせであるかどうかを判断する方法。これをよりよく理解するために、API Server と etcd の関連性と、間違ったシステムのトラブルシューティングがどれほど簡単かを見てみましょう。

次の図では、API サーバーのレイテンシーを示していますが、グラフのバーにレイテンシーのほとんどが etcd レベルで表示されるため、このレイテンシーの多くは etcd サーバーと相関しています。20 秒の API サーバーのレイテンシーがあると同時に 15 秒の etcd レイテンシーがある場合、レイテンシーの大部分は実際には etcd レベルになります。

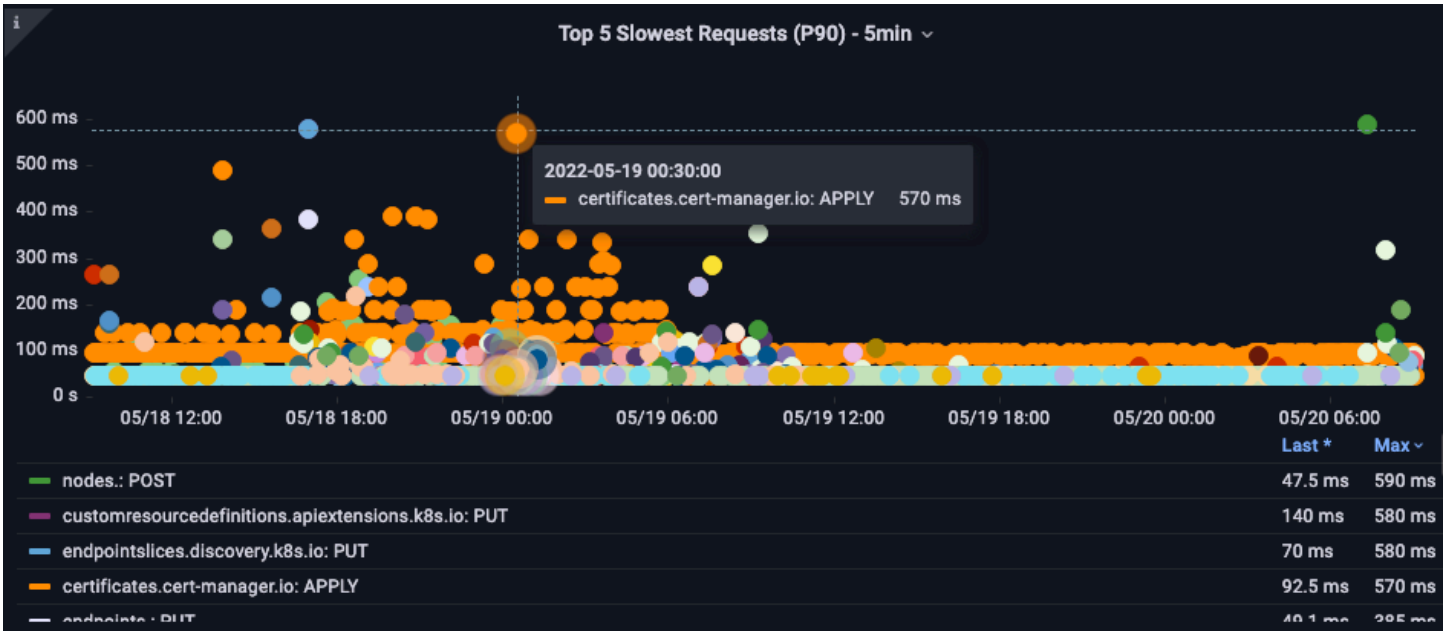
フロー全体を見ると、API Server のみに焦点を当てるのではなく、etcd が圧縮されている (適用カウンターの増加が遅い) ことを示すシグナルを探すことが賢明であることがわかります。一目で適切な問題領域にすばやく移動できるため、ダッシュボードは強力になります。

> **Note**
>
> セクションのダッシュボードは、https://github.com/RiskyAdventure/Troubleshooting-Dashboards/blob/main/api-troubleshooter.json にあります。



コントロールプレーンとクライアント側の問題

このグラフでは、その期間に完了するのに最も時間がかかった API コールを探しています。この場合、カスタムリソース (CRD) が 05:40 の間に最も潜在的な呼び出しである APPLY 関数を呼び出していることがわかります。



このデータを使用して、Ad-Hoc PromQL または CloudWatch Insights クエリを使用して、その期間中に監査ログから LIST リクエストをプルし、これがどのアプリケーションであるかを確認できます。

CloudWatch を使用したソースの検索

メトリクスは、調査する問題領域を見つけ、問題の時間枠と検索パラメータの両方を絞り込むのに最適です。このデータを取得したら、より詳細な時間とエラーのログに移行します。これを行うには、CloudWatch Logs Insights を使用してログをメトリクスに変換します。

例えば、上記の問題を調査するために、次の CloudWatch Logs Insights クエリを使用して userAgent と requestURI をプルし、このレイテンシーの原因となっているアプリケーションをピンダウンできるようにします。

> **Note**
>
> Watch で通常のリスト/再同期動作をプルしないようにするには、適切な カウントを として使用する必要があります。

```
fields @timestamp, @message
| filter @logStream like "kube-apiserver-audit"
| filter ispresent(requestURI)
| filter verb = "list"
| parse requestReceivedTimestamp /\d+-\d+-(?<StartDay>\d+)T(?<StartHour>\d+):(?
<StartMinute>\d+):(?<StartSec>\d+).(?<StartMsec>\d+)Z/
| parse stageTimestamp /\d+-\d+-(?<EndDay>\d+)T(?<EndHour>\d+):(?<EndMinute>\d+):(?
<EndSec>\d+).(?<EndMsec>\d+)Z/
| fields (StartHour * 3600 + StartMinute * 60 + StartSec + StartMsec / 1000000) as
 StartTime, (EndHour * 3600 + EndMinute * 60 + EndSec + EndMsec / 1000000) as EndTime,
 (EndTime - StartTime) as DeltaTime
| stats avg(DeltaTime) as AverageDeltaTime, count(*) as CountTime by requestURI,
 userAgent
| filter CountTime >=50
| sort AverageDeltaTime desc
```

このクエリを使用すると、2 つの異なるエージェントが多数の高レイテンシーリストオペレーションを実行していることがわかりました。Splunk および CloudWatch エージェント。データを使用して、このコントローラーを削除、更新、または別のプロジェクトに置き換えることができます。

requestURI	AverageDeltaTime	AverageD	CountTime
/api/v1/pods	rest-client/2.1.0 (linux-gnu x86_64) ruby/2.5.5p157	9.1395	87
/api/v1/pods?limit=500&resourceVersion=0	amazon-cloudwatch-agent/v0.0.0 (linux/amd64) kubernetes/$Format	5.3976	929

> **Note**
>
> このトピックの詳細については、次のブログを参照してください。

スケジューラー

EKS コントロールプレーンインスタンスは別の AWS アカウントで実行されるため、これらのコンポーネントをメトリクス用にスクレイピングすることはできません (API サーバーは例外です)。ただし、これらのコンポーネントの監査ログにアクセスできるため、これらのログをメトリクスに変換して、いずれかのサブシステムがスケーリングのボトルネックを引き起こしているかどうかを確認することができます。CloudWatch Logs Insights を使用して、スケジューラキューにあるスケジュールされていないポッドの数を確認しましょう。

スケジューラログのスケジュールされていないポッド

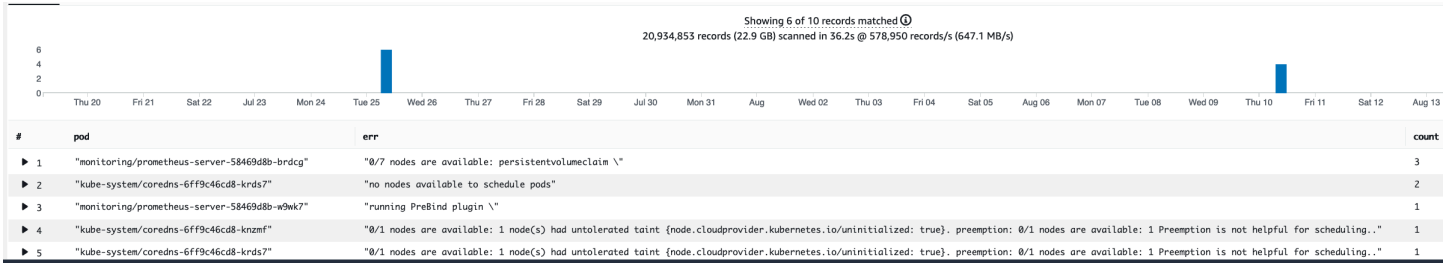
セルフマネージド Kubernetes (Kops など) でスケジューラメトリクスを直接スクレイプするアクセス権がある場合、次の PromQL を使用してスケジューラのバックログを理解します。

```
max without(instance)(scheduler_pending_pods)
```

EKS では上記のメトリクスにアクセスできないため、以下の CloudWatch Logs Insights クエリを使用して、特定の期間中にスケジュールできなかったポッドの数を確認してバックログを確認します。その後、ピーク時にメッセージをさらに詳しく調べて、ボトルネックの性質を理解できます。たとえば、ノードのスピンアップ速度が十分でない、またはスケジューラ自体のレートリミッターなどです。

```
fields timestamp, pod, err, @message
| filter @logStream like "scheduler"
| filter @message like "Unable to schedule pod"
| parse @message  /^.(?<date>\d{4})\s+(?<timestamp>\d+:\d+:\d+\.\d+)\s+\S*\s+\S+\]\s
\"(.*?)\"\s+pod=(?<pod>\"(.*?)\")\s+err=(?<err>\"(.*?)\")/
| count(*) as count by pod, err
| sort count desc
```

ここでは、ストレージ PVC が使用できないためにポッドがデプロイされなかったというスケジューラのエラーが表示されます。



#	pod	err	count
▶ 1	"monitoring/prometheus-server-58469d8b-brdcg"	"0/7 nodes are available: persistentvolumeclaim \"	3
▶ 2	"kube-system/coredns-6ff9c46cd8-krds7"	"no nodes available to schedule pods"	2
▶ 3	"monitoring/prometheus-server-58469d8b-w9wk7"	"running PreBind plugin \"	1
▶ 4	"kube-system/coredns-6ff9c46cd8-knzmf"	"0/1 nodes are available: 1 node(s) had untolerated taint {node.cloudprovider.kubernetes.io/uninitialized: true}. preemption: 0/1 nodes are available: 1 Preemption is not helpful for scheduling.."	1
▶ 5	"kube-system/coredns-6ff9c46cd8-krds7"	"0/1 nodes are available: 1 node(s) had untolerated taint {node.cloudprovider.kubernetes.io/uninitialized: true}. preemption: 0/1 nodes are available: 1 Preemption is not helpful for scheduling.."	1

> **Note**
>
> この関数を有効にするには、コントロールプレーンで監査ログ記録を有効にする必要があります。また、時間の経過とともに不必要にコストが増加しないように、ログの保持を制限することもベストプラクティスです。以下の EKSCTL ツールを使用してすべてのログ記録関数を有効にする例。

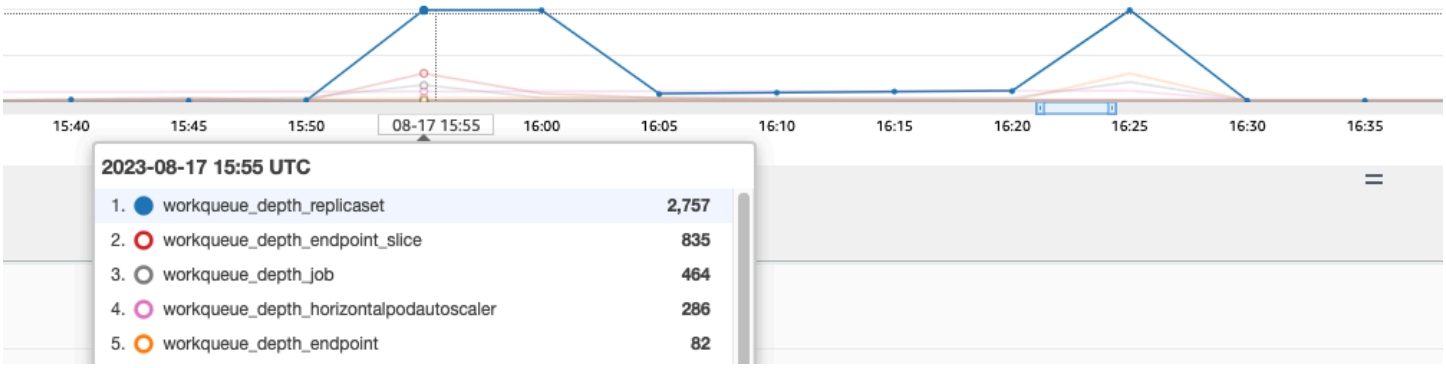
```
cloudWatch:
  clusterLogging:
    enableTypes: ["*"]
    logRetentionInDays: 10
```

Kube コントローラーマネージャー

Kube Controller Manager には、他のすべてのコントローラーと同様に、一度に実行できるオペレーションの数に制限があります。これらのパラメータを設定できる KOPS 設定を見て、これらのフラグの一部を確認しましょう。

```
  kubeControllerManager:
    concurrentEndpointSyncs: 5
    concurrentReplicasetSyncs: 5
    concurrentNamespaceSyncs: 10
    concurrentServiceaccountTokenSyncs: 5
    concurrentServiceSyncs: 5
    concurrentResourceQuotaSyncs: 5
    concurrentGcSyncs: 20
    kubeAPIBurst: 30
    kubeAPIQPS: "20"
```

これらのコントローラーには、クラスターの解約率が高いときにいっぱいになるキューがあります。この場合、レプリカセットセットコントローラーのキューには大きなバックログがあります。



このような状況に対処するには、2 つの異なる方法があります。セルフマネージドを実行すると、同時ゴルーチンを増やすことができますが、KCM でより多くのデータを処理することで etcd に影響します。もう 1 つのオプションは、デプロイ `.spec.revisionHistoryLimit` で を使用するレプリカセットオブジェクトの数を減らして、ロールバックできるレプリカセットオブジェクトの数を減らし、このコントローラーへの負荷を軽減することです。

```
spec:
  revisionHistoryLimit: 2
```

他の Kubernetes 機能は、高チャーンレートシステムの負荷を軽減するために調整またはオフにすることができます。たとえば、ポッド内のアプリケーションが k8s API に直接話す必要がない場合、それらのポッドに射影されたシークレットをオフにすると、ServiceaccountTokenSyncs の負荷が軽減されます。これは、可能であればこのような問題に対処するより望ましい方法です。

```
kind: Pod
spec:
  automountServiceAccountToken: false
```

メトリクスにアクセスできないシステムでは、ログを再度確認して競合を検出できます。コントローラーごとまたは集計レベルで処理されているリクエストの数を確認するには、次の CloudWatch Logs Insights クエリを使用します。

KCM によって処理された合計ボリューム

```
# Query to count API qps coming from kube-controller-manager, split by controller type.
```

```
# If you're seeing values close to 20/sec for any particular controller, it's most
 likely seeing client-side API throttling.
fields @timestamp, @logStream, @message
| filter @logStream like /kube-apiserver-audit/
| filter userAgent like /kube-controller-manager/
# Exclude lease-related calls (not counted under kcm qps)
| filter requestURI not like "apis/coordination.k8s.io/v1/namespaces/kube-system/
leases/kube-controller-manager"
# Exclude API discovery calls (not counted under kcm qps)
| filter requestURI not like "?timeout=32s"
# Exclude watch calls (not counted under kcm qps)
| filter verb != "watch"
# If you want to get counts of API calls coming from a specific controller, uncomment
 the appropriate line below:
# | filter user.username like "system:serviceaccount:kube-system:job-controller"
# | filter user.username like "system:serviceaccount:kube-system:cronjob-controller"
# | filter user.username like "system:serviceaccount:kube-system:deployment-controller"
# | filter user.username like "system:serviceaccount:kube-system:replicaset-controller"
# | filter user.username like "system:serviceaccount:kube-system:horizontal-pod-
autoscaler"
# | filter user.username like "system:serviceaccount:kube-system:persistent-volume-
binder"
# | filter user.username like "system:serviceaccount:kube-system:endpointslice-
controller"
# | filter user.username like "system:serviceaccount:kube-system:endpoint-controller"
# | filter user.username like "system:serviceaccount:kube-system:generic-garbage-
controller"
| stats count(*) as count by user.username
| sort count desc
```

ここで重要なのは、スケーラビリティの問題を調べるとき、詳細なトラブルシューティングフェーズに進む前にパスのすべてのステップ (API、スケジューラ、KCM など) を調べることです。多くの場合、本番環境では、Kubernetes の複数の部分を調整して、システムが最もパフォーマンスの高い機能を発揮できるようにします。はるかに大きなボトルネックの単なる症状 (ノードタイムアウトなど) を誤ってトラブルシューティングするのは簡単です。

ETCD

etcd はメモリマップファイルを使用してキーと値のペアを効率的に保存します。2、4、8 8GB の制限で一般的に設定されるこのメモリ容量のサイズを設定する保護メカニズムがあります。データベース内のオブジェクトが少ないということは、オブジェクトが更新され、古いバージョンをクリーンアウトする必要がある場合に行う必要があるクリーンアップ etcd が少なくなることを意味します。オ

ブジェクトの古いバージョンをクリーンアウトするこのプロセスは、圧縮と呼ばれます。多数の圧縮操作の後、特定のしきい値を超える、または一定のスケジュールで発生する、デフラグと呼ばれる使用可能なスペースを回復する後続のプロセスがあります。

Kubernetes 内のオブジェクトの数を制限し、圧縮とデフラグの両方のプロセスの影響を減らすために実行できるユーザー関連の項目がいくつかあります。たとえば、Helm は高い を保持しますrevisionHistoryLimit。これにより、システム上の ReplicaSets などの古いオブジェクトがロールバックを実行できるようになります。履歴制限を 2 に設定することで、オブジェクト (ReplicaSets など) の数を 10 個から 2 個に減らすことができ、システムへの負荷を軽減できます。

```
apiVersion: apps/v1
kind: Deployment
spec:
  revisionHistoryLimit: 2
```

モニタリングの観点から、システムレイテンシーの急増が時間で区切られた設定パターンで発生した場合は、このデフラグプロセスがソースであるかどうかを確認してください。これは CloudWatch Logs を使用して確認できます。

デフラグの開始/終了時刻を確認するには、次のクエリを使用します。

```
fields @timestamp, @message
| filter @logStream like /etcd-manager/
| filter @message like /defraging|defraged/
| sort @timestamp asc
```

@message	@logStream
Oct 6 19:19:52 ip-172-16-176-9 etcdnanny: {"level":"info","ts":"2022-10-06T19:19:52.826Z","caller":"defrag/defrag.go:312","msg":"defraging","…	etcd-manager-i-0101a20d88e74fa5b
Oct 6 19:20:28 ip-172-16-176-9 etcdnanny: {"level":"info","ts":"2022-10-06T19:20:28.836Z","caller":"defrag/defrag.go:322","msg":"defraged"}	etcd-manager-i-0101a20d88e74fa5b
Oct 6 19:25:57 ip-172-16-101-191 etcdnanny: {"level":"info","ts":"2022-10-06T19:25:57.650Z","caller":"defrag/defrag.go:312","msg":"defraging"…	etcd-manager-i-0e3aa5c2ae801af44
Oct 6 19:26:34 ip-172-16-101-191 etcdnanny: {"level":"info","ts":"2022-10-06T19:26:34.457Z","caller":"defrag/defrag.go:322","msg":"defraged"}	etcd-manager-i-0e3aa5c2ae801af44
Oct 6 19:32:26 ip-172-16-50-136 etcdnanny: {"level":"info","ts":"2022-10-06T19:32:26.592Z","caller":"defrag/defrag.go:312","msg":"defraging",…	etcd-manager-i-025ebb6b463ee0f21
Oct 6 19:33:09 ip-172-16-50-136 etcdnanny: {"level":"info","ts":"2022-10-06T19:33:09.575Z","caller":"defrag/defrag.go:322","msg":"defraged"}	etcd-manager-i-025ebb6b463ee0f21

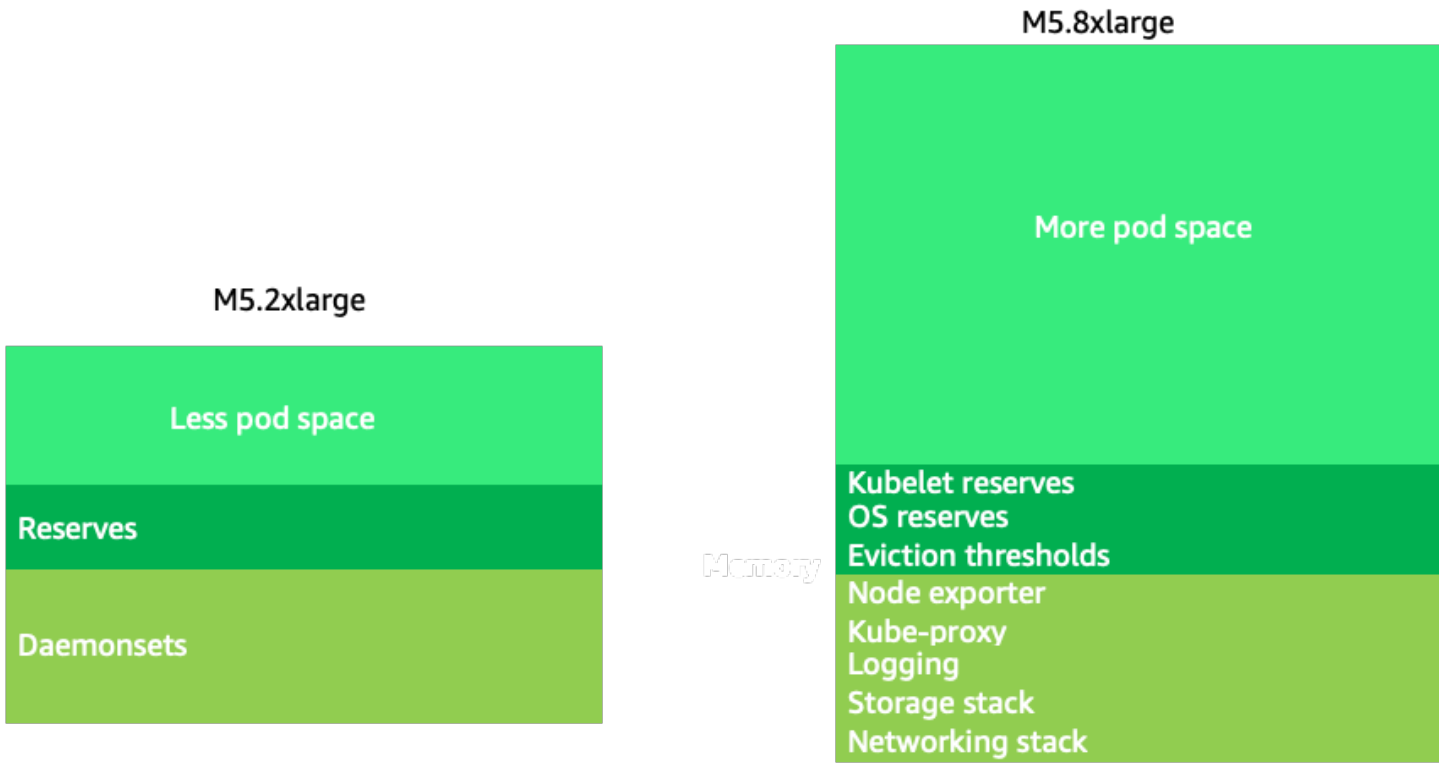
ノードとワークロードの効率

ワークロードとノードを効率的に使用すると、パフォーマンスとスケールを向上させながら、複雑さとコストを削減できます。この効率を計画するときは多くの要素を考慮する必要があり、トレードオフと各機能に対する 1 つのベストプラクティス設定の観点から考えることが最も簡単です。これらのトレードオフについては、次のセクションで詳しく見てみましょう。

ノードの選択

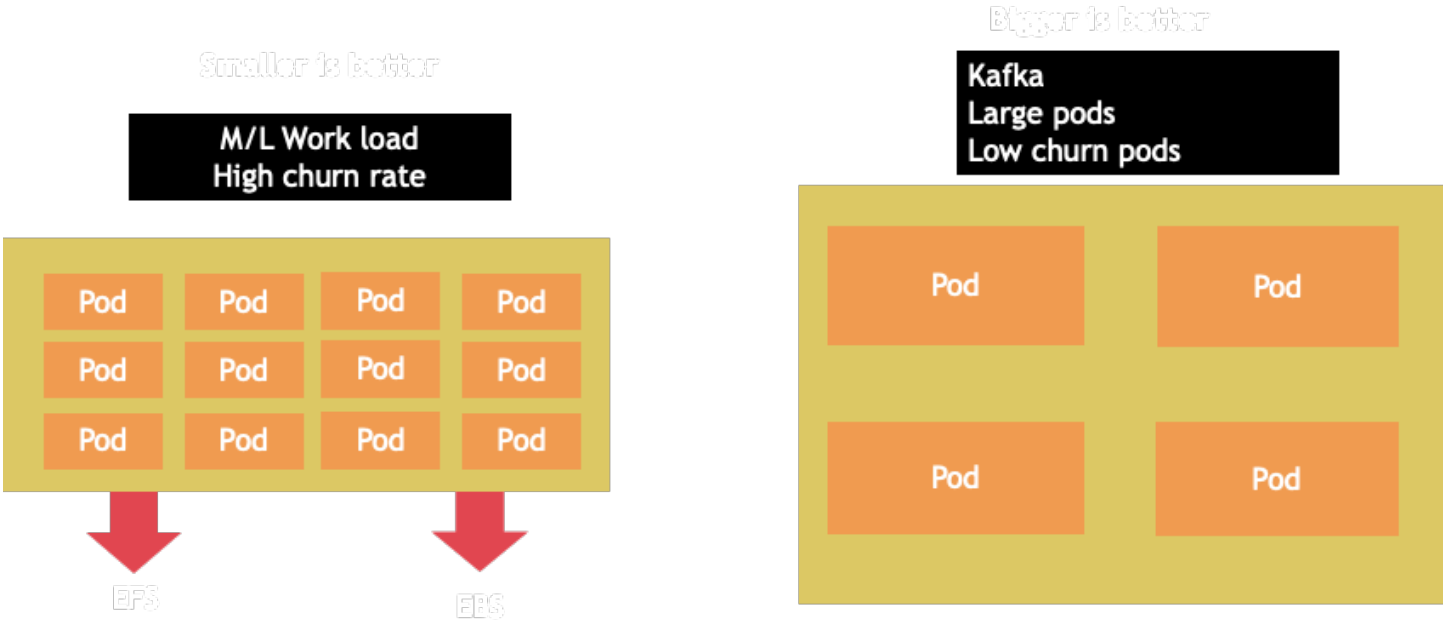
ノードサイズをわずかに大きく (4 ~ 12xlarge) 使用すると、システムコンポーネントの DaemonSets やリザーブなど、「オーバーヘッド」に使用されるノードの割合が減るため、ポッドの実行に使用できる領域が増えます。以下の図では、DaemonSets が中程度の数の 2xlarge システムと 8xlarge システムの使用可能なスペースの違いを示しています。

> **Note**
>
> k8s は一般的なルールとして水平方向にスケールするため、ほとんどのアプリケーションでは NUMA サイズノードのパフォーマンスへの影響を考慮することは意味がないため、そのノードサイズを下回る範囲を推奨します。



ノードサイズが大きいと、ノードあたりの使用可能なスペースの割合が高くなります。ただし、このモデルは、エラーが発生したり、ノードを飽和させたりするほど多くのポッドでノードをパッキングすることで極端に処理できます。ノード飽和度のモニタリングは、より大きなノードサイズを正常に使用する上で重要です。

ノードの選択がone-size-fits-all提案になることはほとんどありません。多くの場合、解約率が劇的に異なるワークロードを異なるノードグループに分割するのが最善です。チャーンレートが高い小規模なバッチワークロードは 4xlarge ファミリーのインスタンスが最適です。一方、8 vCPU を使用し、チャーンレートが低い Kafka などの大規模なアプリケーションは 12xlarge ファミリーが最適です。



> **Note**
>
> 非常に大きなノードサイズで考慮すべきもう 1 つの要因は、CGROUPS がコンテナ化されたアプリケーションから vCPU の総数を非表示にしないことです。動的ランタイムでは、意図しない数の OS スレッドが発生し、トラブルシューティングが困難なレイテンシーが発生することがよくあります。これらのアプリケーションでは、CPU ピン留めをお勧めします。トピックの詳細については、次の動画 https://www.youtube.com/watch?v=NqtfDy_KAqg を参照してください。

ノードビンのパッキング

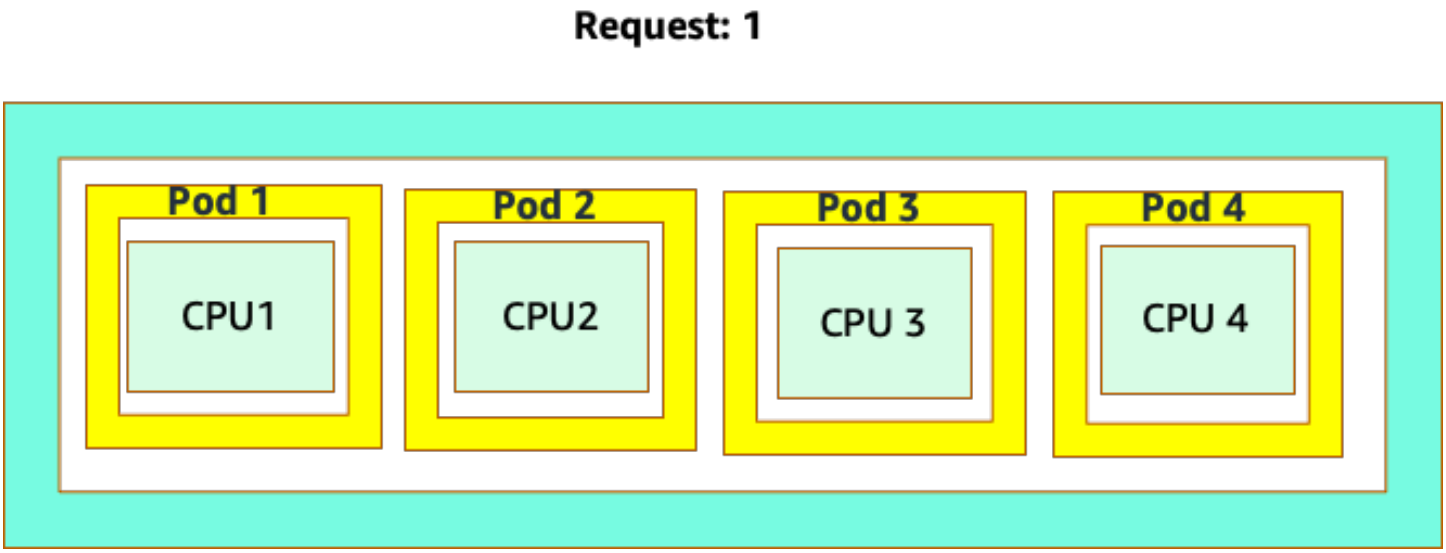
Kubernetes ルールと Linux ルール

Kubernetes でワークロードを処理するときに注意する必要があるルールは 2 つあります。Kubernetes スケジューラのルール。リクエスト値を使用してノードでポッドをスケジュールし、ポッドがスケジュールされた後に何が起こるかを示します。これは Kubernetes ではなく Linux の領域です。

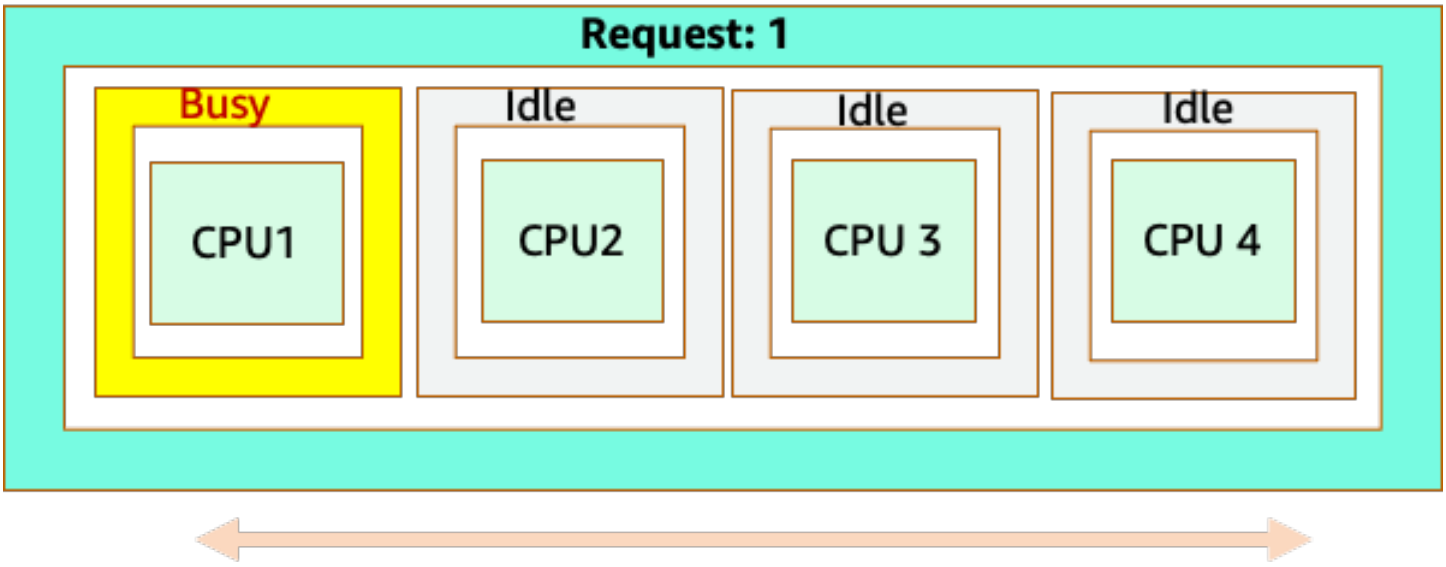
Kubernetes スケジューラが完了すると、新しいルールセットが Linux Completely Fair Scheduler (CFS) を引き継ぎます。重要な点は、Linux CFS にはコアの概念がないということです。コアでの思考が、スケールに合わせてワークロードを最適化する際の大きな問題につながる理由について説明します。

コアでの思考

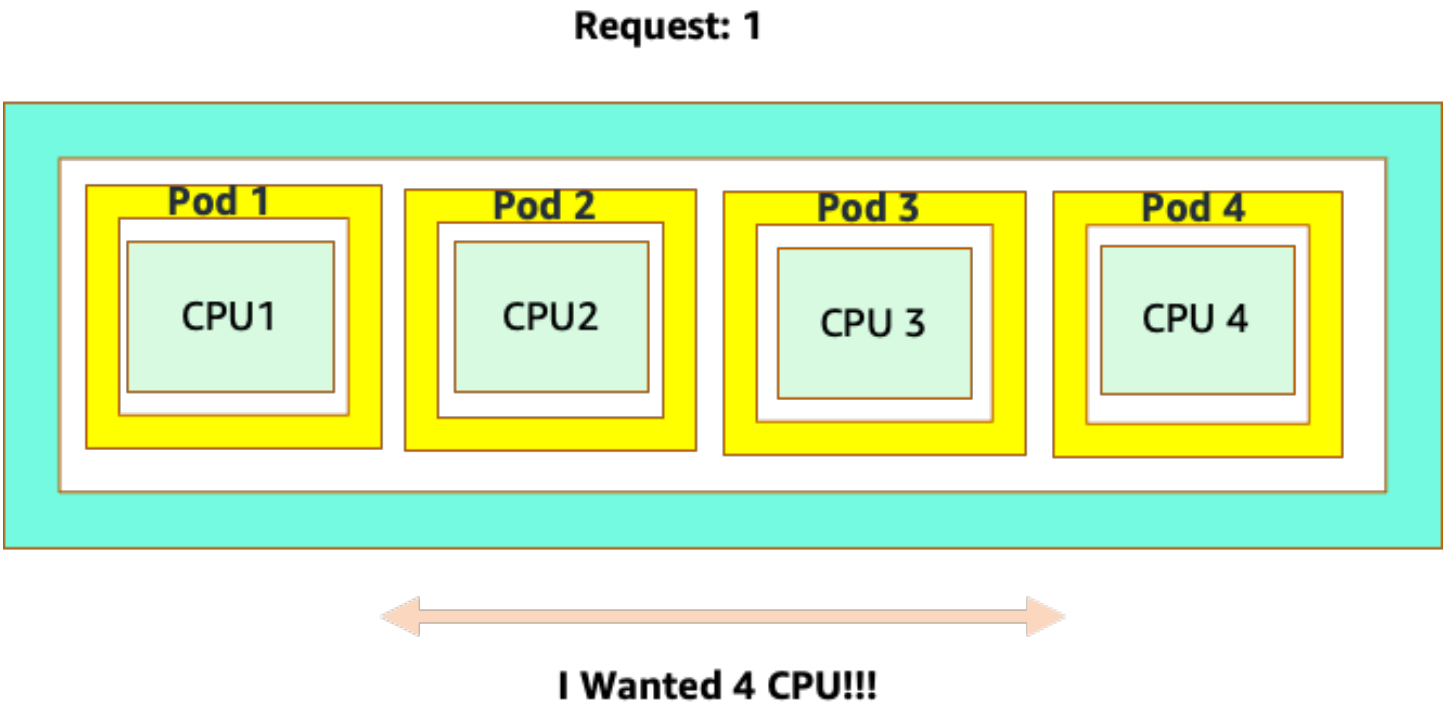
Kubernetes スケジューラにはコアの概念があるため、混乱が始まります。Kubernetes スケジューラの観点から見ると、それぞれが 1 つのコアセットのリクエストを持つ 4 つの NGINX ポッドを持つノードを見ると、ノードは次のようになります。



ただし、これが Linux CFS の観点からどのように異なるかを考えてみましょう。Linux CFS システムを使用する際に覚えておくべき最も重要なことは、共有システムにカウントされるコンテナはビジーコンテナ (CGROUPS) のみです。この場合、最初のコンテナのみがビジー状態であるため、ノード上の 4 つのコアすべてを使用できます。



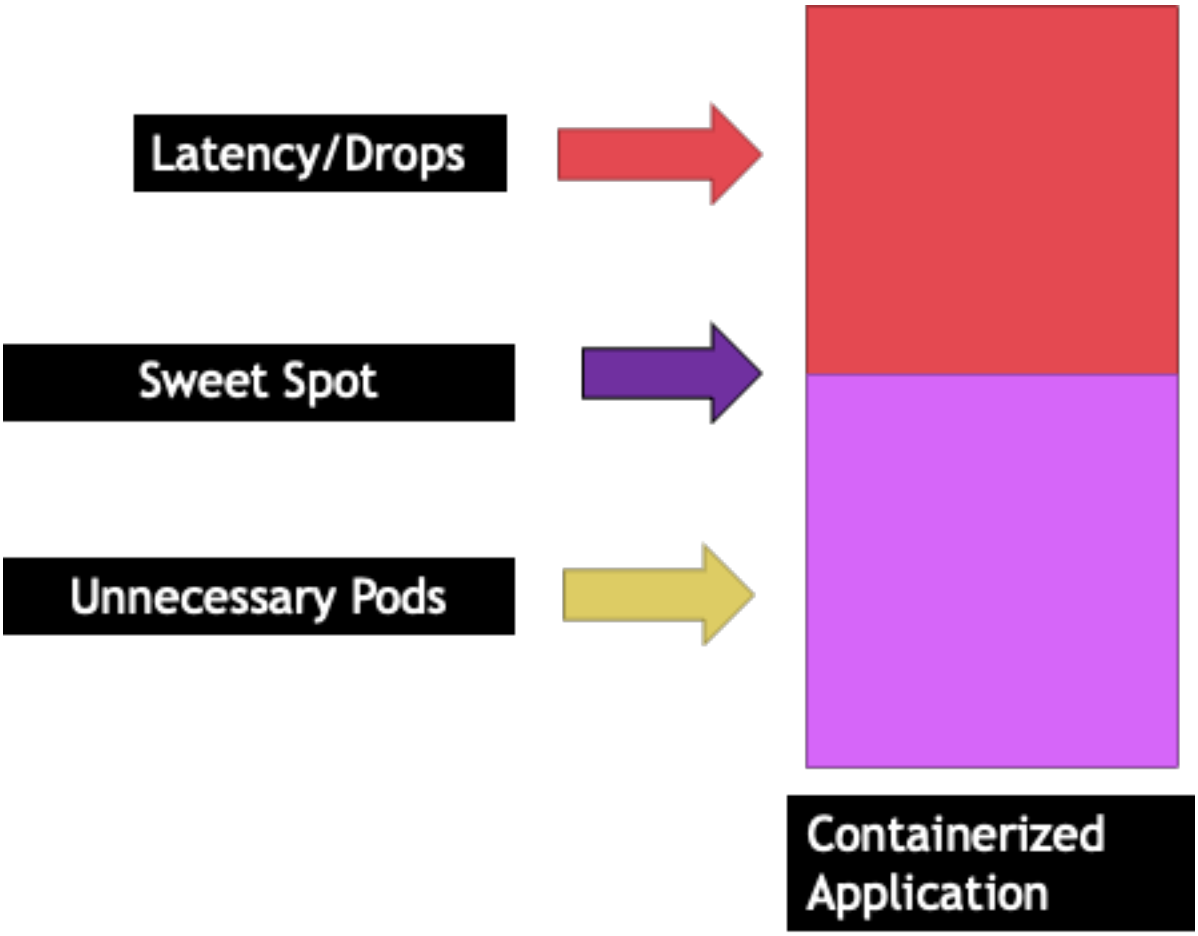
なぜこれが重要なのですか? NGINX アプリケーションがそのノードで唯一のビジーコンテナであった開発クラスターでパフォーマンステストを実行したとします。アプリケーションを本番環境に移行すると、NGINX アプリケーションは 4 つの vCPU のリソースを必要としますが、ノード上の他のすべてのポッドがビジー状態であるため、アプリケーションのパフォーマンスは制限されます。



このような状況では、アプリケーションが「スイートスポット」にスケールできないため、不必要にコンテナを追加することになります。の重要な概念"sweet spot"をもう少し詳しく見てみましょう。

アプリケーションの適切なサイジング

各アプリケーションには、トラフィックを取ることができない特定のポイントがあります。このポイントを超えると、処理時間が長くなり、このポイントを大幅に超えてプッシュするとトラフィックがドロップされる可能性があります。これは、アプリケーションの飽和ポイントと呼ばれます。スケーリングの問題を回避するには、飽和ポイントに達する前にアプリケーションのスケーリングを試みる必要があります。これをスイートスポットと呼びましょう。



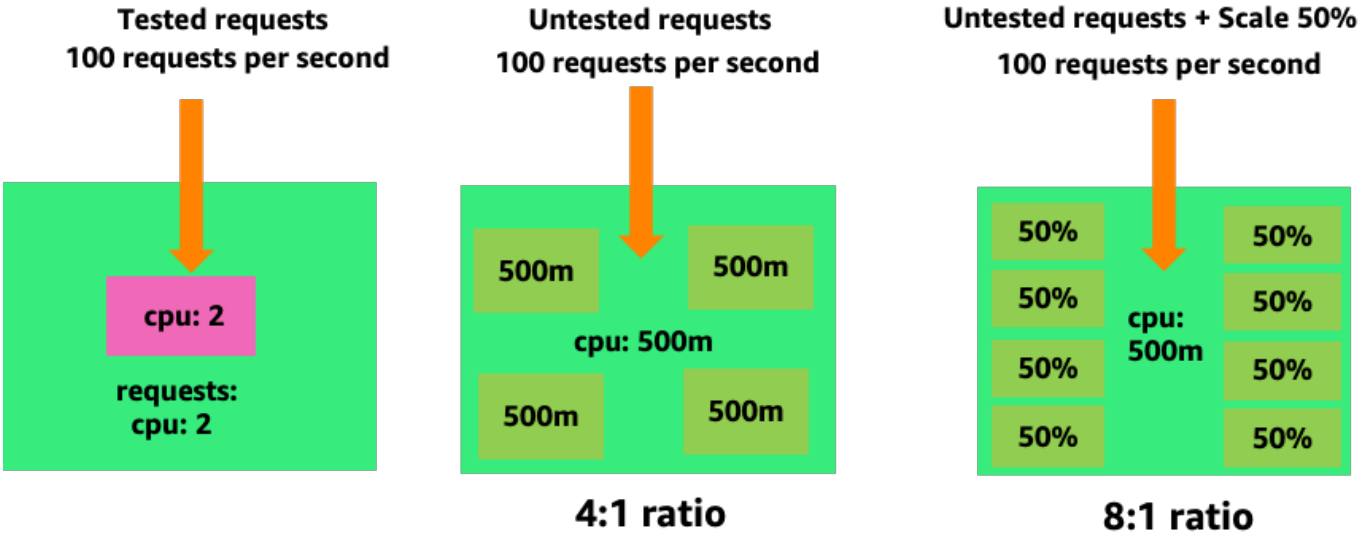
各アプリケーションをテストして、そのスイートスポットを理解する必要があります。アプリケーションごとに異なるため、ここでは一般的なガイダンスはありません。このテストでは、アプリケーションの飽和ポイントを示す最適なメトリクスを理解しようとしています。多くの場合、使用率メトリクスはアプリケーションが飽和していることを示すために使用されますが、これはすぐにスケーリングの問題につながる可能性があります (このトピックについては、後のセクションで詳しく説明し

ます)。この「スイートスポット」を取得したら、これを使用してワークロードを効率的にスケーリングできます。

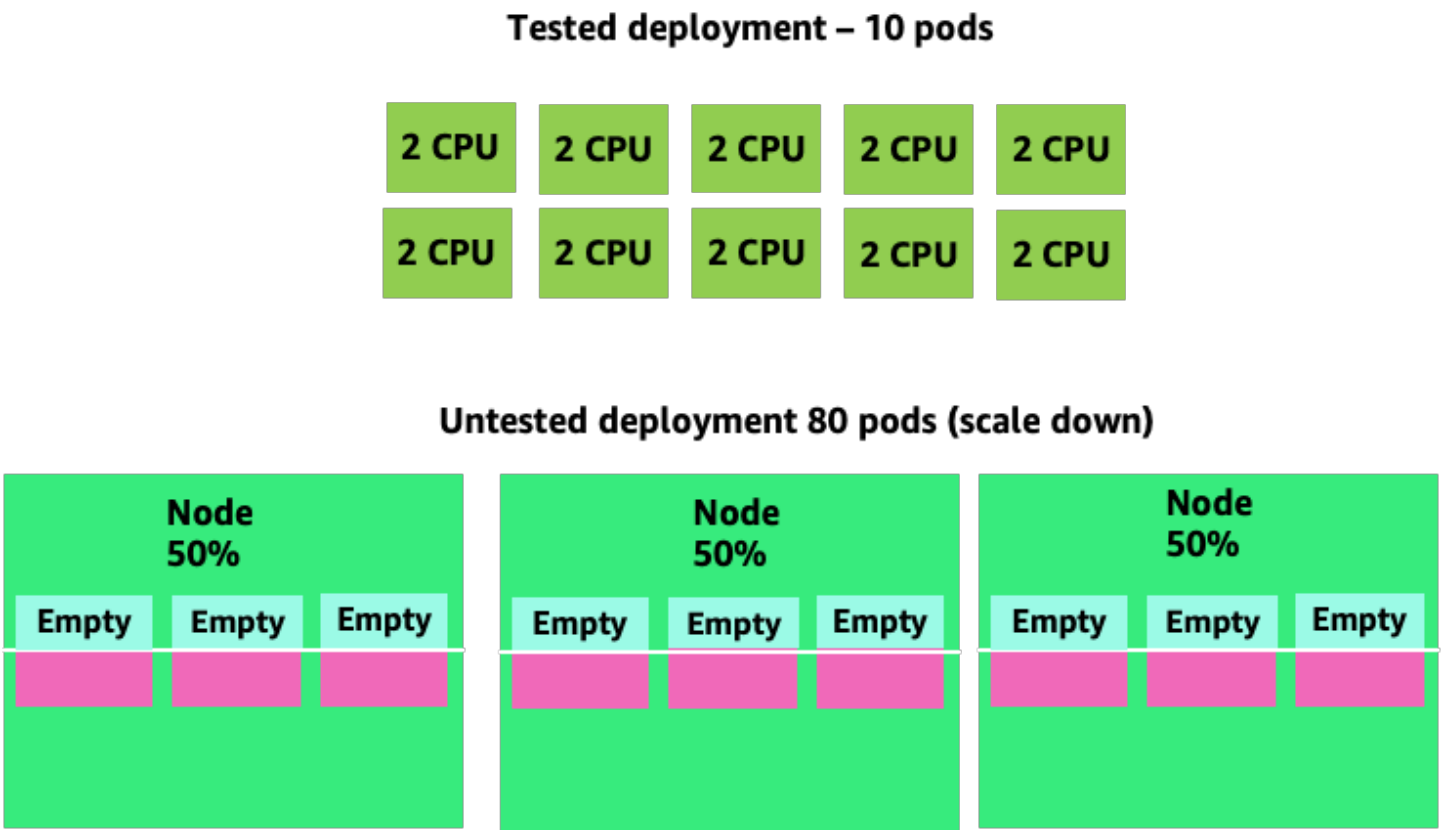
逆に、スイートスポットのかなり前にスケールアップして不要なポッドを作成した場合はどうなりますか? これについては、次のセクションで説明します。

ポッドスプロール

不要なポッドをすばやく作成する方法を確認するには、左側の最初の例を見てみましょう。このコンテナの正しい垂直スケールは、1 秒あたり 100 件のリクエストを処理するときに、約 2 つの vCPUs の使用率を占めます。ただし、リクエストを半分のコアに設定してリクエスト値をプロビジョニング不足にした場合、実際に必要な 1 つのポッドごとに 4 つのポッドが必要になります。この問題をさらに悪化させると、HPA がデフォルトの 50% CPU に設定されている場合、それらのポッドは半分空にスケールされ、8:1 の比率になります。

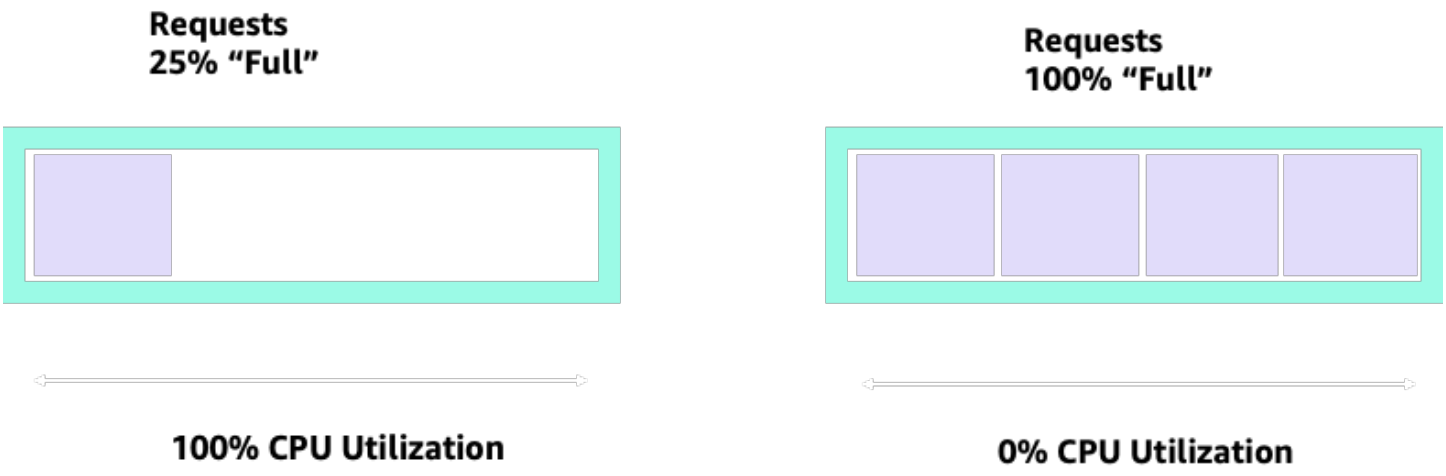


この問題を拡大すると、この問題がどのように機能しなくなるかをすばやく確認できます。スウィートスポットが正しく設定されていない 10 個のポッドをデプロイすると、80 個のポッドと、それらを実行するために必要な追加のインフラストラクチャにすばやくスパイラルされる可能性があります。



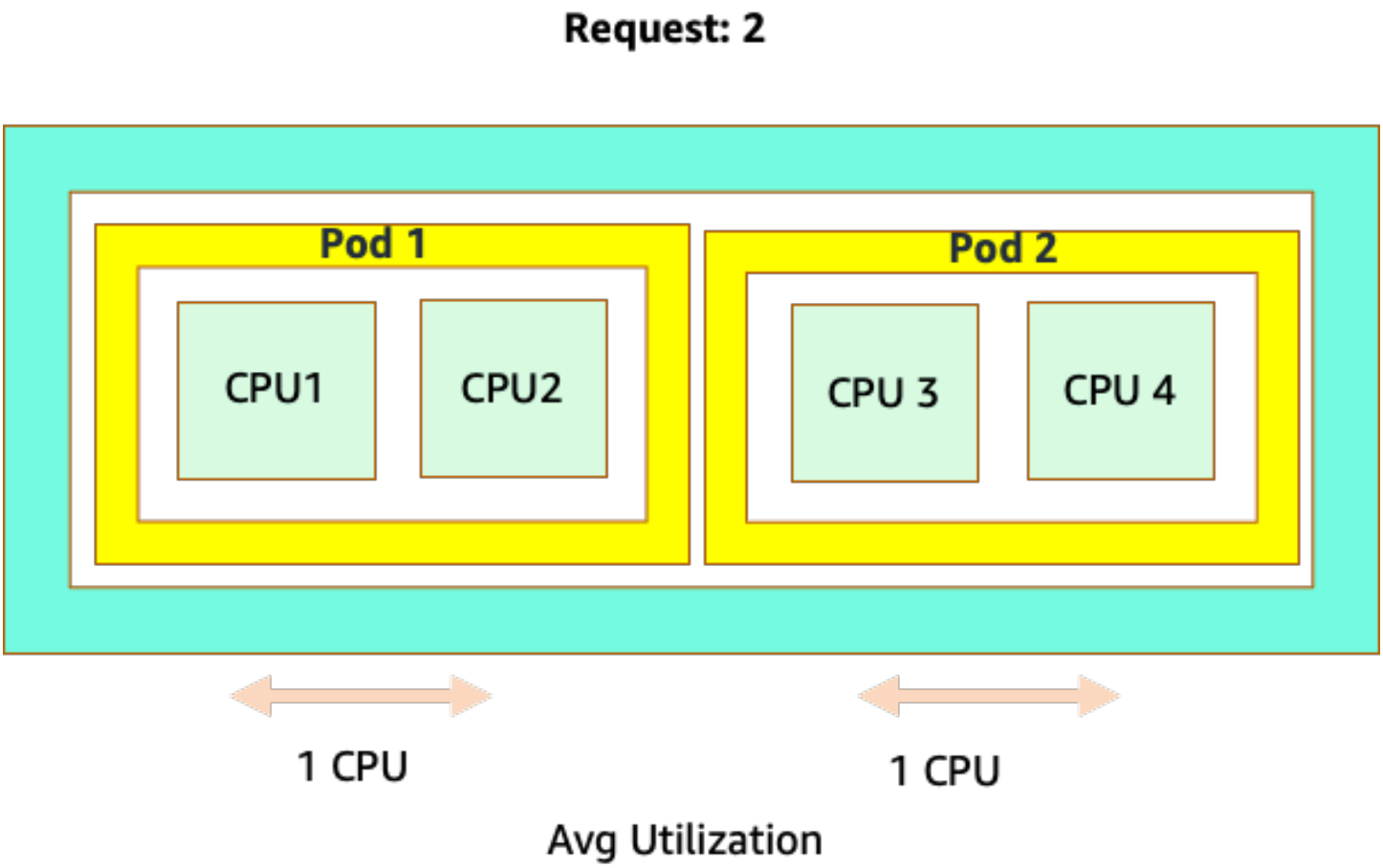
アプリケーションがスウィートスポットで動作できないことによる影響を理解したところで、ノードレベルに戻り、Kubernetes スケジューラと Linux CFS の違いがなぜそれほど重要なのかを尋ねてみましょう。

HPA でスケールアップまたはスケールダウンする場合、より多くのポッドを割り当てるのに十分なスペースがあるシナリオが考えられます。左に示されているノードはすでに 100% の CPU 使用率であるため、これは不適切な決定です。非現実的ではあるが理論上可能なシナリオでは、ノードが完全に満杯であっても CPU 使用率がゼロであるという極端なシナリオが発生する可能性があります。

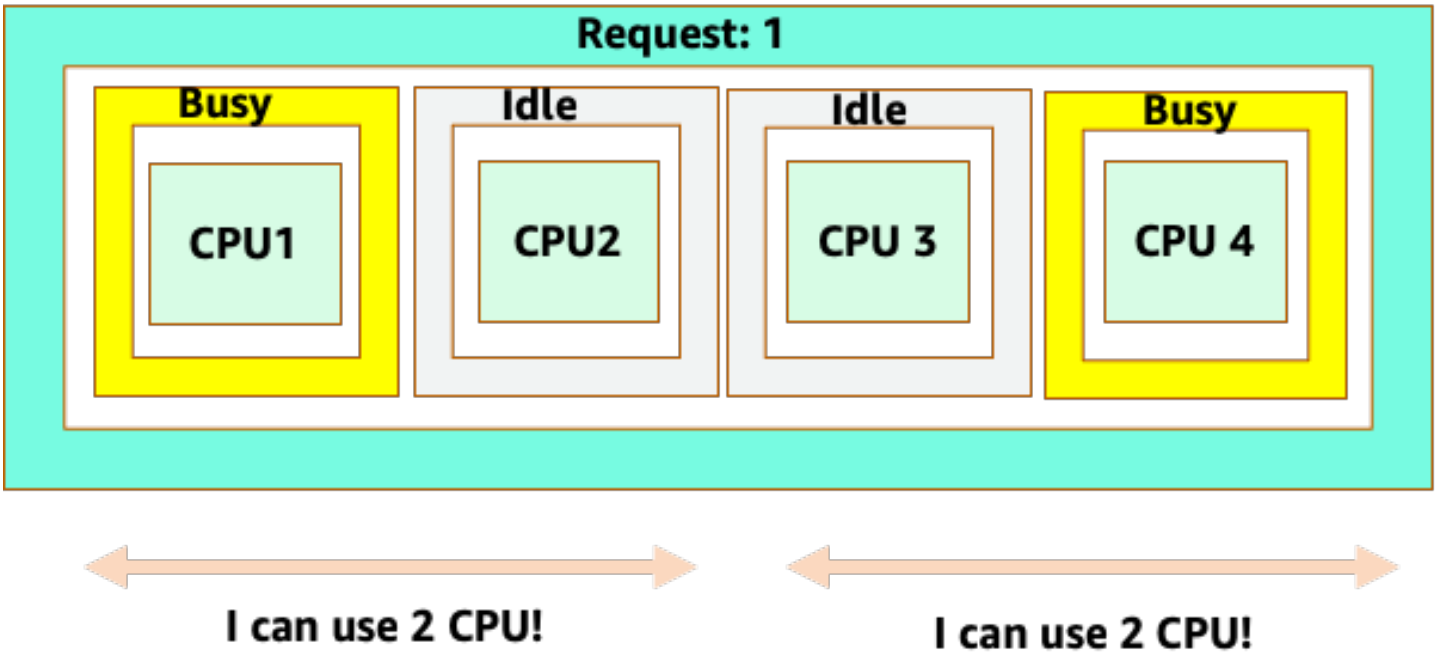


リクエストの設定

そのアプリケーションの「スイートスポット」値にリクエストを設定しようとしますが、次の図に示すように非効率になります。ここでは、リクエスト値を 2 vCPU に設定していますが、これらのポッドの平均使用率はほとんどの場合 1 CPU しか実行されません。この設定により、CPU サイクルの 50% が無駄になるため、許容できません。



これにより、問題に対する複雑な回答が得られます。コンテナ使用率はバキュームでは考慮できません。ノードで実行されている他のアプリケーションを考慮する必要があります。次の例では、本質的にバースト性のコンテナは、メモリが制限されている可能性のある 2 つの CPU 使用率の低いコンテナと で混合されます。このようにして、コンテナがノードに課税することなくスイートスポットに到達できるようにします。



これらすべてから取り除く重要な概念は、Kubernetes スケジューラのコアの概念を使用して Linux コンテナのパフォーマンスを理解することで、関連付けられていないため、意思決定が不十分になる可能性があることです。

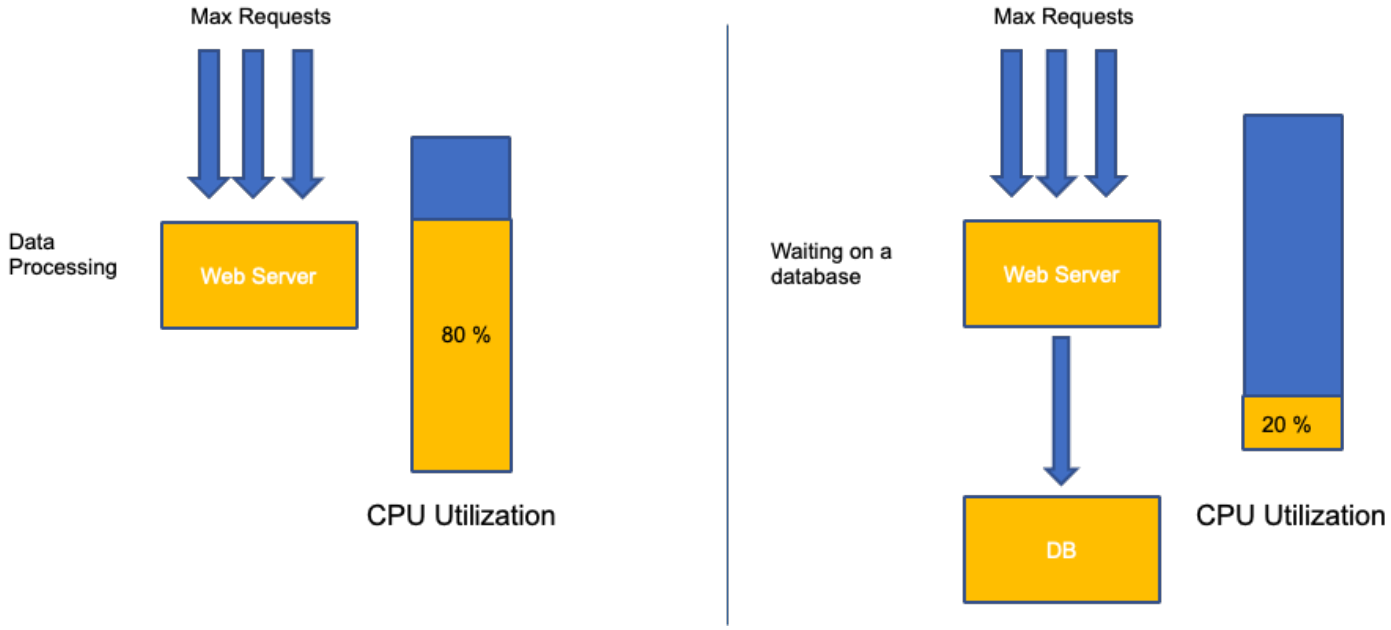
> **Note**
>
> Linux CFS には強力なポイントがあります。これは、I/O ベースのワークロードに特に当てはまります。ただし、アプリケーションがサイドカーなしでフルコアを使用し、I/O 要件がない場合、CPU ピン留めはこのプロセスからかなりの複雑さを取り除く可能性があるため、これらの注意点をお勧めします。

使用率と飽和度

アプリケーションスケーリングでよくある間違いは、スケーリングメトリクスに CPU 使用率のみを使用することです。複雑なアプリケーションでは、これはほとんどの場合、アプリケーションが実際にリクエストで飽和していることを示す不適切な指標です。左側の例では、すべてのリクエストが実際にウェブサーバーに達しているため、CPU 使用率は飽和度とともに適切に追跡されています。

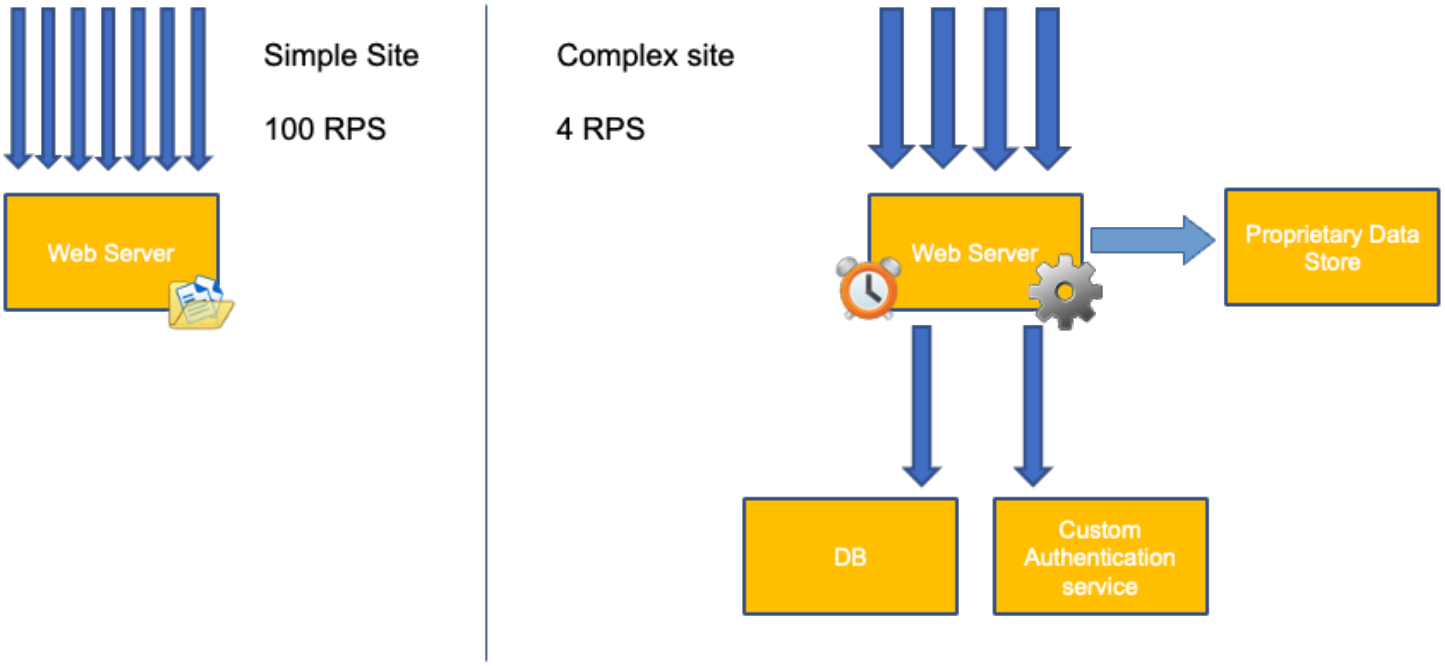
実際のアプリケーションでは、これらのリクエストの一部がデータベースレイヤーや認証レイヤーなどによって処理される可能性があります。このより一般的なケースでは、リクエストが他のエンティ

ティによって処理されているため、CPU が飽和状態で追跡されていないことに注意してください。この場合、CPU は飽和度を示す非常に低い指標です。



アプリケーションのパフォーマンスに間違ったメトリクスを使用することが、Kubernetes での不要で予測不可能なスケーリングの最大の理由です。使用しているアプリケーションの種類に適した飽和メトリクスを選択するには、細心の注意が必要です。指定できるすべてのレコメンデーションに適合するサイズは 1 つではないことに注意してください。使用する言語と問題のアプリケーションの種類に応じて、飽和度に関するさまざまなメトリクスのセットがあります。

この問題は CPU 使用率でのみ発生する可能性がありますが、1 秒あたりのリクエスト数などの他の一般的なメトリクスも、上記で説明したのとまったく同じ問題になる可能性があります。リクエストは DB レイヤー、認証レイヤーにも送信され、ウェブサーバーによって直接サービスされないため、ウェブサーバー自体の真の飽和のメトリクスとしては不十分なことに注意してください。



残念ながら、適切な飽和メトリクスを選択する場合、簡単な回答はありません。考慮すべきガイドラインをいくつか示します。

- 言語ランタイムを理解する - 複数の OS スレッドを持つ言語は、シングルスレッドアプリケーションと反応が異なるため、ノードへの影響は異なります。
- 正しい垂直スケールを理解します。新しいポッドをスケーリングする前に、アプリケーションの垂直スケールにどのくらいのバッファが必要ですか?
- アプリケーションの飽和度を真に反映するメトリクス - Kafka プロデューサーの飽和メトリクスは、複雑なウェブアプリケーションとは大きく異なります。
- ノード上の他のすべてのアプリケーションが相互にどのような影響を与えるか - アプリケーションのパフォーマンスは、ノード上の他のワークロードが大きな影響を与えるバキュームでは行われません。

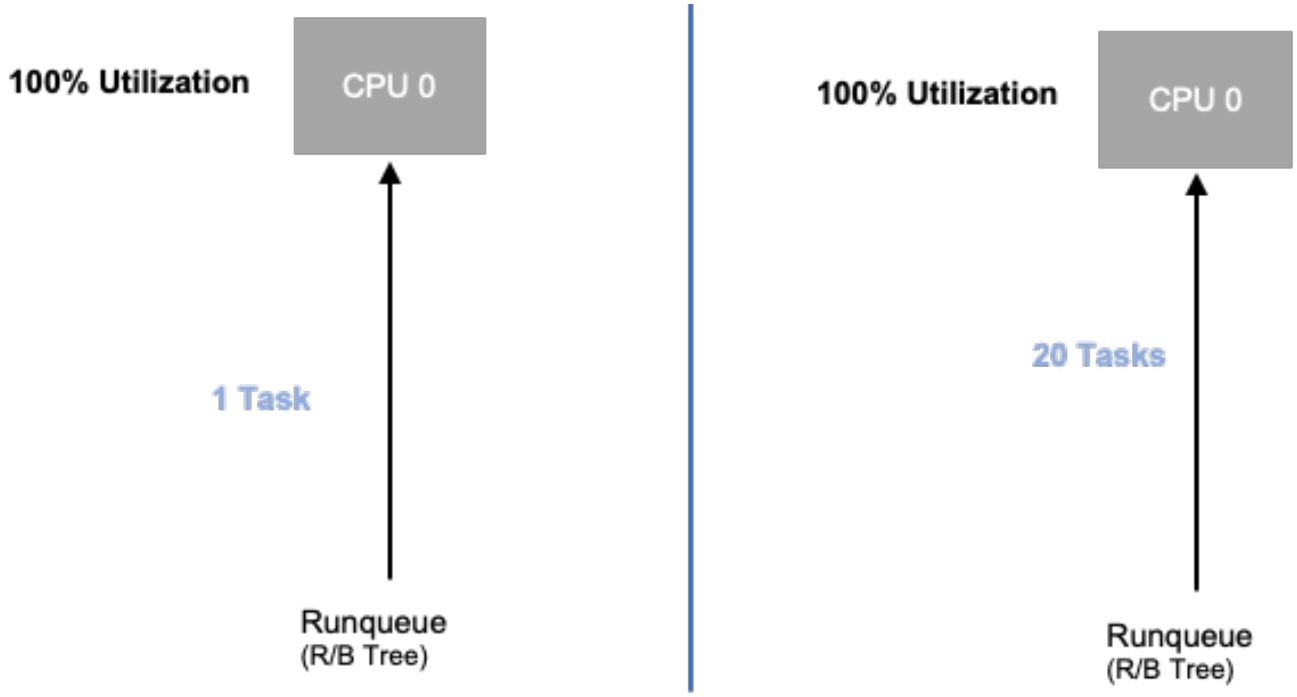
このセクションを終了するには、上記を過度に複雑で不要なものとして却下するのが簡単です。多くの場合、問題が発生していますが、間違ったメトリクスを調べているため、問題の本質に気付いていません。次のセクションでは、これがどのように行われるかを見ていきます。

ノード飽和

アプリケーションの飽和度を調べたので、ノードの観点からこの同じ概念を見てみましょう。使用率と飽和度の違いを確認するために 100% が使用されている 2 つの CPUs を見てみましょう。

左側の vCPU は 100% 使用されていますが、この vCPU で実行するのを待っているタスクは他にないため、純粋に理論的な意味では、これは非常に効率的です。一方、2 番目の例では、vCPU によって処理されるのを待っているシングルスレッドアプリケーションが 20 個あります。20 個のアプリケーションすべてが、vCPU によって処理されるターンを待っている間に、何らかのレイテンシーが発生するようになりました。つまり、右側の vCPU は飽和状態です。

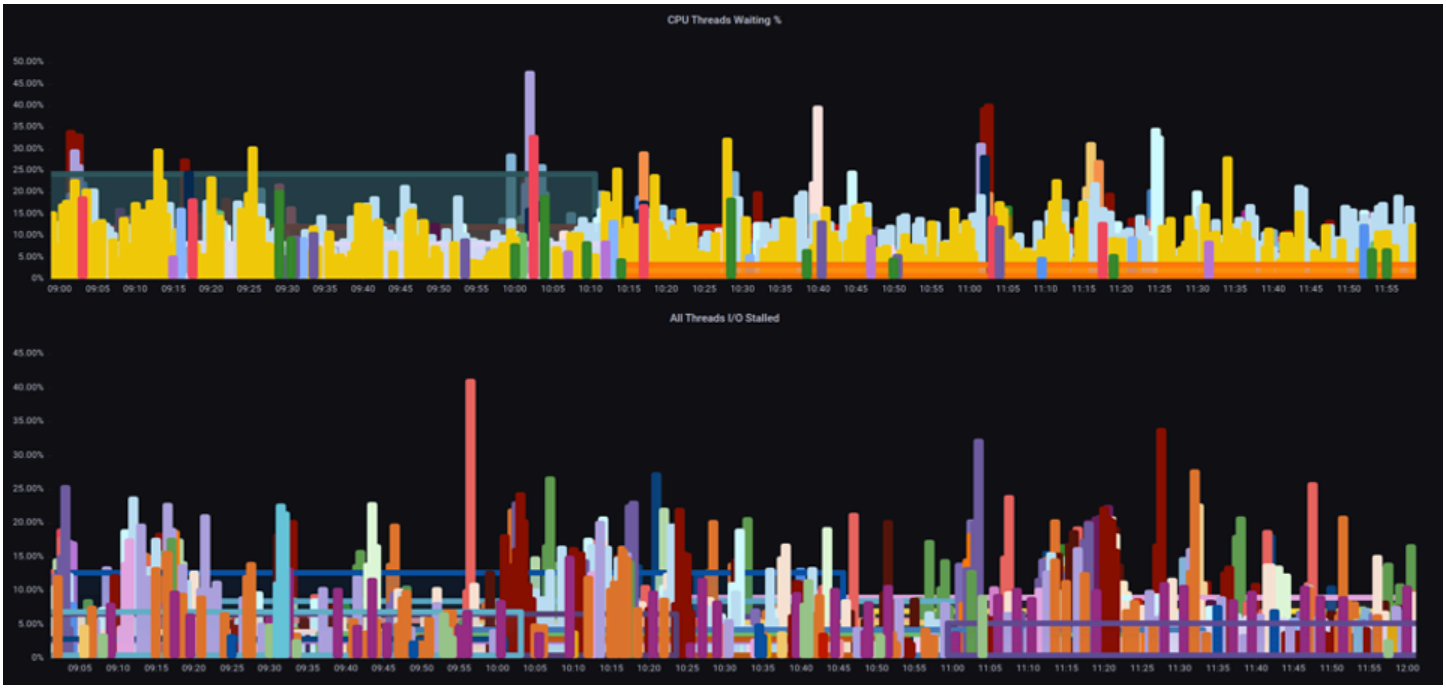
使用率だけを見ると、この問題は発生しませんが、このレイテンシーはネットワークなど無関係なものに起因する可能性があり、誤った経路をたどる可能性があります。



ノードで実行されているポッドの総数をいつでも増やすときは、使用率メトリクスだけでなく、飽和メトリクスを表示することが重要です。これは、ノードが過剰に飽和しているという事実を簡単に見逃す可能性があるためです。このタスクでは、次のグラフに示すように、プレッシャーストール情報メトリクスを使用できます。

PromQL - I/O の停止

```
topk(3, ((irate(node_pressure_io_stalled_seconds_total[1m])) * 100))
```



> **Note**
>
> 圧力ストールメトリクスの詳細については、https://facebookmicrosites.github.io/psi/docs/overview* を参照してください。

これらのメトリクスを使用すると、スレッドが CPU を待っているかどうか、またはボックス上のすべてのスレッドがメモリや I/O などのリソースを待っているときに停止しているかどうかを確認できます。 例えば、1 分間にインスタンス上のすべてのスレッドが I/O を待つのが停止した割合を確認できます。

```
topk(3, ((irate(node_pressure_io_stalled_seconds_total[1m])) * 100))
```

このメトリクスを使用すると、上のグラフでボックス上のすべてのスレッドがハイウォーターマークで I/O を待っている時間の 45% 停止していたことがわかります。つまり、その 1 分間にこれらの CPU サイクルをすべて破棄していました。これが起こっていることを理解することで、vCPU のかなりの時間を再利用できるため、スケーリングがより効率的になります。

HPA V2

HPA API の Auto Scaling/v2 バージョンを使用することをお勧めします。HPA API の古いバージョンでは、特定のエッジケースでスケーリングが停止する可能性があります。また、各スケーリングス

テップ中にポッドが 2 倍になるだけでした。これにより、迅速なスケーリングが必要な小規模なデプロイに問題が発生しました。

Auto Scaling/v2 を使用すると、スケールオンする複数の基準を含める柔軟性が向上し、カスタムメトリクスと外部メトリクス (K8s 以外のメトリクス) を使用するときに大きな柔軟性が得られます。

例として、3 つの値のうち最も高い値でスケーリングできます (以下を参照) 。すべてのポッドの平均使用率が 50% を超える場合、カスタムメトリクスの Ingress の 1 秒あたりのパケット数が平均 1,000 を超える場合、または Ingress オブジェクトが 1 秒あたりのリクエスト数が 10K000 を超える場合にスケーリングされます。

> **Note**
>
> これは、Auto Scaling API の柔軟性を示すためだけに、本番環境でのトラブルシューティングが困難な、過度に複雑なルールには反対することをお勧めします。

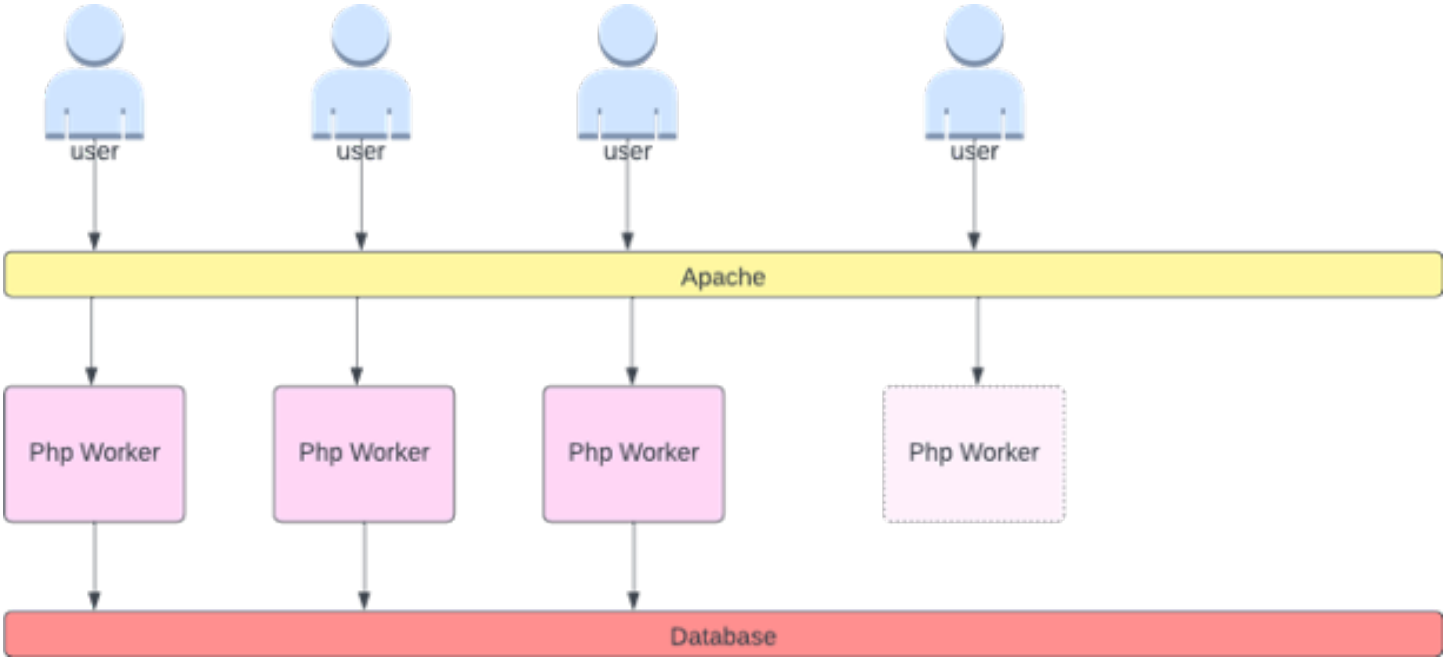
```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: php-apache
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: php-apache
  minReplicas: 1
  maxReplicas: 10
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 50
  - type: Pods
    pods:
      metric:
        name: packets-per-second
      target:
        type: AverageValue
```

```
        averageValue: 1k
  - type: Object
    object:
      metric:
        name: requests-per-second
      describedObject:
        apiVersion: networking.k8s.io/v1
        kind: Ingress
        name: main-route
      target:
        type: Value
        value: 10k
```

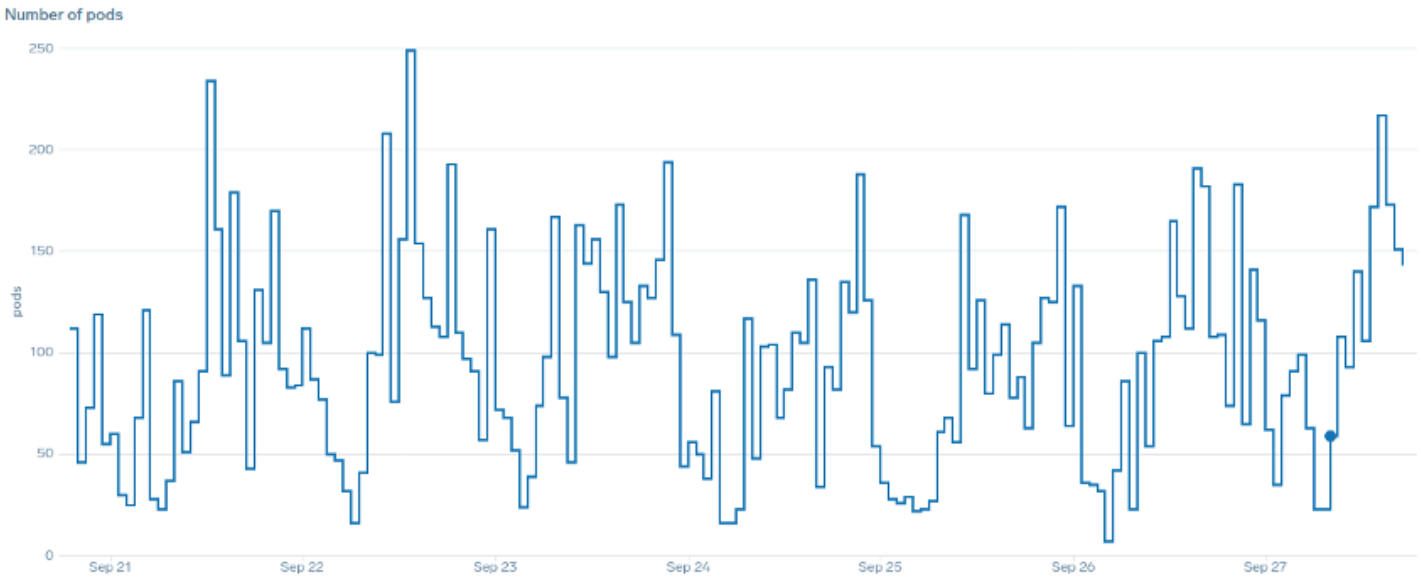
ただし、このようなメトリクスを複雑なウェブアプリケーションに使用する危険性について学びました。この場合、アプリケーションの飽和度と使用率を正確に反映したカスタムメトリクスまたは外部メトリクスを使用することで、より良いサービスを提供できます。HPAv2 では、任意のメトリクスに従ってスケーリングできますが、そのメトリクスを見つけて Kubernetes に エクスポートして使用する必要があります。

例えば、Apache でアクティブなスレッドキュー数を確認できます。これにより、「スムーズ」なスケーリングプロファイルが作成されることがよくあります (近い将来、その期間の詳細を参照) 。スレッドがアクティブな場合、そのスレッドがデータベースレイヤーを待っているか、ローカルでリクエストを処理しているかは関係ありません。すべてのアプリケーションスレッドが使用されている場合、アプリケーションが飽和していることは大きな兆候です。

このスレッドの枯渇をシグナルとして使用して、完全に利用可能なスレッドプールを持つ新しいポッドを作成できます。これにより、トラフィックが多い時間帯にアプリケーションが吸収するバッファの大きさも制御できます。例えば、スレッドプールの合計が 10 の場合、4 つのスレッドを使用したスケーリングと 8 つのスレッドを使用したスケーリングは、アプリケーションのスケーリング時に使用できるバッファに大きな影響を与えます。4 の設定は、負荷の高い状況で迅速にスケーリングする必要があるアプリケーションにとって理にかなっています。リクエストの数が時間の経過とともに急激に増加するため、スケーリングに十分な時間があれば、リソースに対して 8 の設定の方が効率的です。

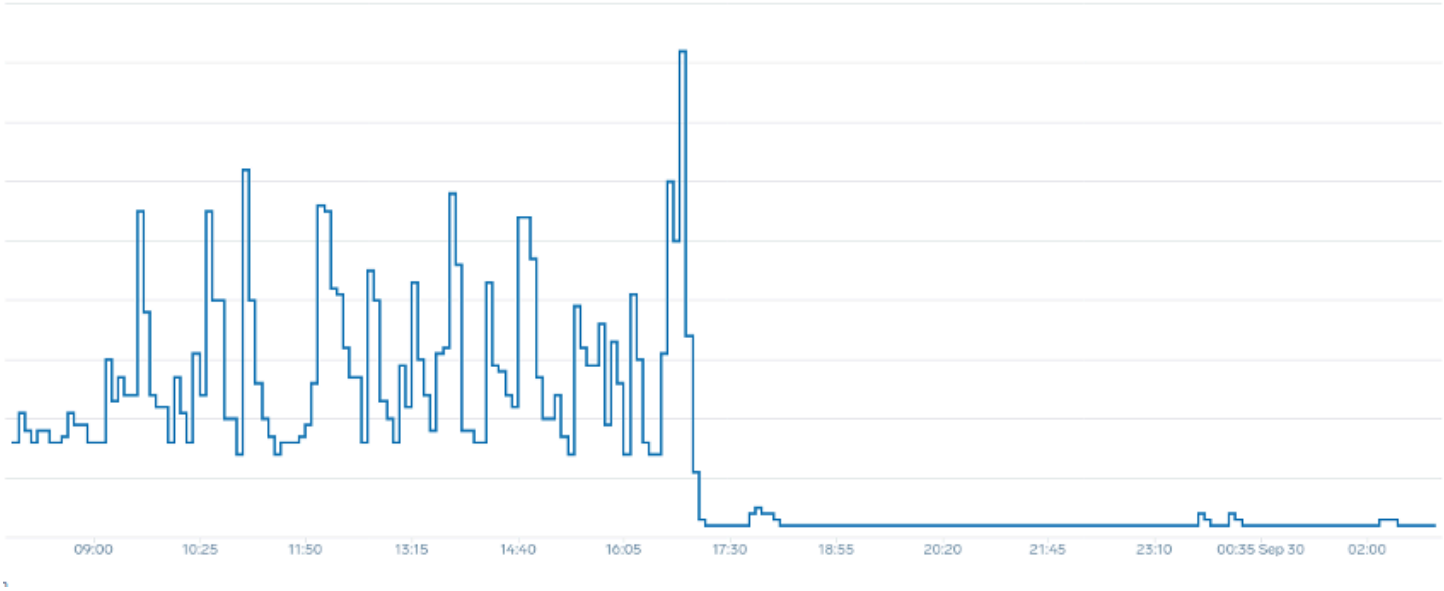


スケーリングとは、「スムーズ」という意味ですか? CPU をメトリクスとして使用している次のグラフに注意してください。このデプロイのポッドは、すぐに再びスケールダウンするために、50 個のポッドから最大 250 個のポッドまで短時間でスパイクします。これは非常に非効率的なスケーリングであり、クラスターのチャーンの主な原因です。



アプリケーションの正しいスイートスポット (グラフの中間) を反映するメトリクスに変更した後、どのようにスムーズにスケールできるかに注目してください。スケーリングが効率的になり、ポッドはリクエスト設定を調整することで指定したヘッドルームに合わせて完全にスケーリングできるようになりました。現在、少数のポッドグループが、数百のポッドが以前に実行していた作業を行ってい

ます。実世界のデータは、これが Kubernetes クラスターのスケーラビリティの最も重要な要素であることを示しています。

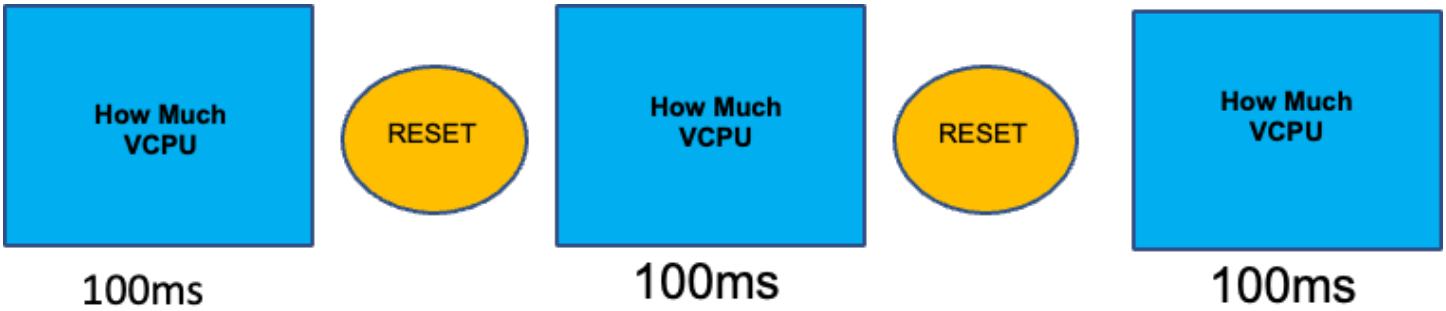


重要なポイントは、CPU 使用率はアプリケーションとノードの両方のパフォーマンスの 1 つのディメンションにすぎないことです。ノードとアプリケーションの唯一のヘルス指標として CPU 使用率を使用すると、スケーリング、パフォーマンス、コストに問題が生じます。これらはすべて密接にリンクされた概念です。アプリケーションとノードのパフォーマンスが高いほど、スケーリングの必要が少なくなり、コストが削減されます。

特定のアプリケーションのスケーリングに正しい飽和メトリクスを見つけて使用すると、そのアプリケーションの真のボトルネックをモニタリングしてアラームを発することもできます。この重要なステップをスキップすると、不可能ではないにしても、パフォーマンスの問題のレポートを理解するのは困難になります。

CPU 制限の設定

誤解されたトピックについてこのセクションをまとめるために、CPU の制限について説明します。つまり、制限は、100 ミリ秒ごとにリセットされるカウンターを持つコンテナに関連付けられたメタデータです。これにより、Linux は 100 ミリ秒の期間に特定のコンテナによってノード全体で使用されている CPU リソースの数を追跡できます。



設定制限の一般的なエラーは、アプリケーションがシングルスレッドで、「割り当てられた」vCPU でのみ実行されていると仮定することです。上のセクションでは、CFS がコアを割り当てず、実際には大きなスレッドプールを実行しているコンテナがボックス上の使用可能なすべての vCPU でスケジュールされることを学習しました。

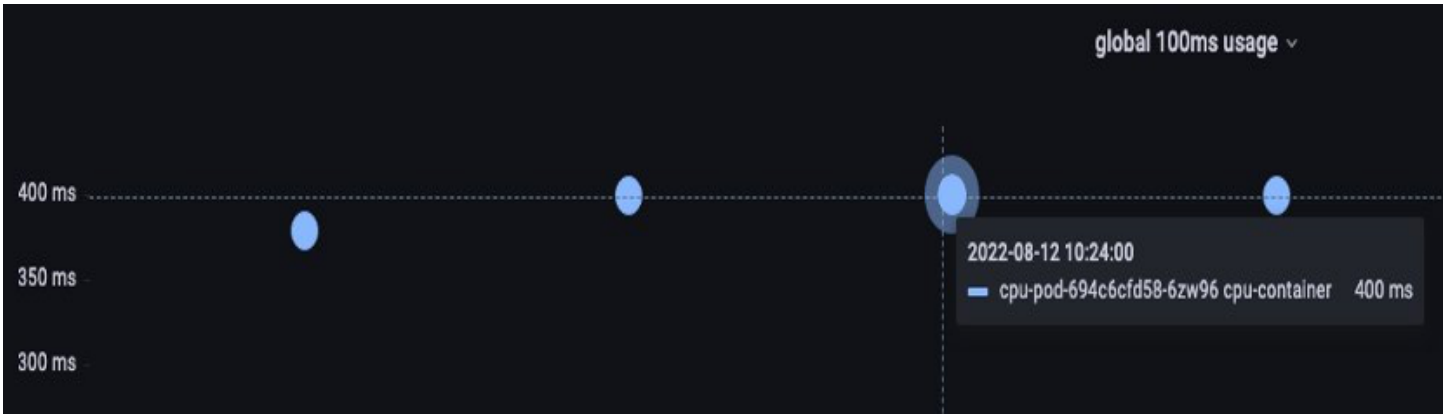
64 個の OS スレッドが 64 個の使用可能なコアで実行されている場合 (Linux ノードの観点から見ると)、これら 64 個のコアすべてで実行されている時間が追加された後、100 ミリ秒の期間の合計使用 CPU 時間料金が発生します。これはガベージコレクションプロセス中にのみ発生する可能性があるため、このようなことを見逃すのは非常に簡単です。そのため、制限を設定する前に、メトリクスを使用して、時間の経過とともに正しい使用量を確保する必要があります。

幸い、アプリケーション内のすべてのスレッドで使用されている vCPU の量を正確に確認する方法があります。`container_cpu_usage_seconds_total` この目的のために メトリクスを使用します。

スロットリングロジックは 100 ミリ秒ごとに発生し、このメトリクスは 1 秒あたりのメトリクスであるため、この 100 ミリ秒の期間に一致するように PromQL を使用します。この PromQL ステートメントの仕組みについて詳しく知りたい場合は、次のブログを参照してください。

PromQL クエリ:

```
topk(3, max by (pod, container)(rate(container_cpu_usage_seconds_total{image!="",
 instance="$instance"}[$__rate_interval]))) / 10
```



適切な値があるとわかったら、制限を本番環境に置くことができます。その後、予期しない理由でアプリケーションがスロットリングされているかどうかを確認する必要があります。これを行うには、以下を確認します。 container_cpu_throttled_seconds_total

```
topk(3, max by (pod, container)(rate(container_cpu_cfs_throttled_seconds_total{image!
=``""``, instance=``"$instance"``}[$__rate_interval]))) / 10
```



メモリ

メモリ割り当ては、Linux CGroup 動作の Kubernetes スケジューリング動作を簡単に混同できる別の例です。これは、CGroup v2 が Linux と Kubernetes でメモリを処理する方法に大きな変更があったため、より微妙なトピックです。詳細については、このブログを参照してください。

CPU リクエストとは異なり、スケジューリングプロセスが完了すると、メモリリクエストは使用されません。これは、CPU と同じ方法で CGroup v1 のメモリを圧縮できないためです。これにより、メモリ制限だけで済みます。メモリ制限は、ポッドを完全に終了することでメモリリークのフェイルセーフとして機能するように設計されています。これはオールスタイルまたはノースタイルの提案ですが、この問題に対処する新しい方法が提供されました。

まず、コンテナに適切な量のメモリを設定することは、表示されるとおり簡単なことではないことを理解することが重要です。Linux のファイルシステムは、パフォーマンスを向上させるためにキャッシュとしてメモリを使用します。このキャッシュは時間の経過とともに増加し、キャッシュに適したメモリ量を知ることは困難ですが、アプリケーションのパフォーマンスに大きな影響を与えることなく再利用できます。これにより、多くの場合、メモリ使用量が誤って解釈されます。

メモリを「圧縮」する機能は、CGroup v2 の背後にある主要なドライバーの 1 つです。CGroup V2 が必要だった理由の詳細については、LISA21 での Chris Down のプレゼンテーションを参照してください。ここでは、最小メモリを正しく設定できない理由が、CGroup v2 とプレッシャーストールメトリクスの作成につながった理由の 1 つです。

幸い、Kubernetes には `memory.min`および `memory.high` の概念がありますが、`requests.memory` これにより、このキャッシュされたメモリを積極的に解放して、他のコンテナを使用できるようになります。コンテナがメモリの上限に達すると、カーネルはコンテナのメモリを `memory.min` に設定された値まで積極的に再利用できます。したがって、ノードにメモリ負荷がかかった場合の柔軟性が向上します。

キーの質問が になり、どの値を `memory.min` に設定しますか? これは、メモリプレッシャーのストールメトリクスが関係する場所です。これらのメトリクスを使用して、コンテナレベルでメモリの「スラッシュ」を検出できます。その後、fbtax などのコントローラーを使用して、このメモリスラッシングを探して `memory.min` の正しい値を検出し、`memory.min` 値をこの設定に動的に設定できます。

概要

セクションを要約すると、以下の概念を簡単にまとめることができます。

- 使用率と飽和度
- Kubernetes スケジューラロジックを使用した Linux パフォーマンスルール

これらの概念を分離するには、細心の注意が必要です。パフォーマンスとスケールはディープレベルでリンクされます。不要なスケーリングではパフォーマンスの問題が発生し、スケーリングの問題が発生します。

Kubernetes アップストリーム SLOs

Amazon EKS はアップストリーム Kubernetes リリースと同じコードを実行し、EKS クラスターが Kubernetes コミュニティで定義された SLOs 内で動作するようにします。Kubernetes Scalability Special Interest Group (SIG) は、スケーラビリティの目標を定義し、SLIs と SLOs。

SLIs は、システムの実行状況を判断するために使用できるメトリクスやメジャーなどのシステムを測定する方法です。例えば、リクエストのレイテンシーやカウントなどです。SLOs「well」を実行しているときに予想される値を定義します。たとえば、リクエストのレイテンシーは 3 秒未満のままです。Kubernetes SLOs と SLIs は Kubernetes コンポーネントのパフォーマンスに焦点を当て、EKS クラスターエンドポイントの可用性に焦点を当てる Amazon EKS Service SLAs とは完全に独立しています。

Kubernetes には、CSI ドライバー、アドミッションウェブフック、自動スケーラーなどのカスタムアドオンまたはドライバーを使用してシステムを拡張できる多くの機能があります。これらの拡張機能は、さまざまな方法で Kubernetes クラスターのパフォーマンスに大きな影響を与える可能性があります。つまり、 を使用するアドミッションウェブフックは、ウェブフックターゲットが使用できない場合に K8s API リクエストにレイテンシーを追加する`failurePolicy=Ignore`可能性があります。Kubernetes Scalability SIG は、「お約束します」フレームワークを使用してスケーラビリティを定義します。

> 次のことを約束する場合: - クラスターを正しく設定する - 「合理的に」拡張機能を使用する - クラスター内の負荷を推奨制限内に維持する
>
> 次に、クラスターがスケールすることを約束します。つまり、すべての SLOsが満たされます。

Kubernetes SLOs

Kubernetes SLOs は、ワーカーノードのスケーリングやアドミッションウェブフックなど、クラスターに影響を与える可能性のあるプラグインや外部の制限をすべて考慮しているわけではありません。これらの SLOs Kubernetes コンポーネントに焦点を当て、Kubernetes アクションとリソースが期待どおりに動作していることを確認します。SLOsKubernetes 開発者が Kubernetes コードを変更してもシステム全体のパフォーマンスが低下しないようにするのに役立ちます。

Kuberntes Scalability SIG では、以下の公式 SLO/SLIsを定義します。Amazon EKS チームは、これらの SLOs/SLIs の EKS クラスターでスケーラビリティテストを定期的に実行し、変更が行われ、新しいバージョンがリリースされるとパフォーマンスが低下するかどうかをモニタリングします。

目的	定義	SLO
API リクエストのレイテンシー (変更)	過去 5 分間の 99 パーセンタイルとして測定される、すべての (リソース、動詞) ペアの	デフォルトの Kubernetes インストールでは、仮想リソースと集計リソースおよびカスタ

目的	定義	SLO
	単一オブジェクトに対する変更 API コールの処理のレイテンシー	ムリソース定義を除くすべての (リソース、動詞) ペアについて、クラスター日あたり 99 パーセンタイル <= 1
API リクエストのレイテンシー (読み取り専用)	すべての (リソース、スコープ) ペアの非ストリーミング読み取り専用 API コールを処理するレイテンシー。過去 5 分間の 99 パーセンタイルとして測定されます。	デフォルトの Kubernetes インストールでは、仮想リソースと集計リソースおよびカスタムリソース定義を除くすべての (リソース、スコープ) ペアについて、クラスター日あたり 99 パーセンタイル: (a) <= `scope=resource` (b) <= 30s それ以外の場合は <= 1s (`scope=namespace` または の場合`scope=cluster`)
ポッドの起動レイテンシー	スケジュール可能なステートレスポッドの起動レイテンシー。ただし、イメージをプルして init コンテナを実行する時間は除きます。これは、ポッド作成タイムスタンプから、すべてのコンテナが開始と報告され、監視によって観測されるまでの時間まで測定され、過去 5 分間で 99 パーセンタイルと測定されます。	デフォルトの Kubernetes インストールでは、クラスター日あたり 99 パーセンタイル <= 5s

API リクエストのレイテンシー

`kube-apiserver` は`1m0s`デフォルトで と`--request-timeout`定義されています。つまり、リクエストはタイムアウトしてキャンセルされる前に最大 1 分 (60 秒) 実行できます。レイテンシー用に定義された SLOs は、変更または読み取り専用のリクエストのタイプによって分類されます。

変異

Kubernetes でリクエストをミューテーションすると、作成、削除、更新など、リソースが変更されます。これらのリクエストは、更新されたオブジェクトが返される前に etcd バックエンドに書き込む必要があるため、コストがかかります。Etcd は、すべての Kubernetes クラスターデータに使用される分散キーバリューストアです。

このレイテンシーは、Kubernetes リソースの (リソース、動詞) ペアの 5 分間で 99 パーセンタイルとして測定されます。たとえば、これにより、ポッドの作成リクエストとノードの更新リクエストのレイテンシーが測定されます。SLO を満たすには、リクエストのレイテンシーが <= 1 秒である必要があります。

[Read-only]

読み取り専用リクエストは、単一のリソース (ポッド X の取得など) またはコレクション (「名前空間 X からすべてのポッドを取得する」など) を取得します。はオブジェクトのキャッシュkube-apiserverを保持するため、リクエストされたリソースはキャッシュから返されるか、最初に etcd から取得する必要がある場合があります。これらのレイテンシーも 5 分間で 99 パーセンタイルで測定されますが、読み取り専用リクエストには個別のスコープを含めることができます。SLO は 2 つの異なる目標を定義します。

- 単一のリソース (など) `kubectl get pod -n mynamespace my-controller-xxx` に対して行われたリクエストの場合、リクエストのレイテンシーは <= 1 秒のままにする必要があります。
- 名前空間またはクラスター (などkubectl get pods -A) 内の複数のリソースに対して行われたリクエストの場合、レイテンシーは <= 30 秒のままにする必要があります

Kubernetes リソースのリストに対して行われたリクエストは、リクエスト内のすべてのオブジェクトの詳細が SLO 内で返されることを想定しているため、SLO にはリクエストスコープごとに異なるターゲット値があります。大規模なクラスターや大量のリソースコレクションでは、レスポンスサイズが大きくなり、返すまでに時間がかかる場合があります。例えば、JSON でエンコードされた各ポッドが約 1 KiB である数万個のポッドを実行しているクラスターでは、クラスター内のすべてのポッドを返すと 10MB 以上になります。Kubernetes クライアントは、APIListChunking を使用してリソースの大規模なコレクションを取得することで、このレスポンスサイズを減らすのに役立ちます。

ポッドの起動レイテンシー

この SLO は主に、Pod の作成からその Pod 内のコンテナが実際に実行を開始するまでの時間に関係します。これを測定するために、Pod に記録された作成タイムスタンプとの差を測定し、その Pod

の WATCH がコンテナの開始をレポートすると、コンテナが計算されます (コンテナイメージのプル時間と初期化コンテナの実行時間を除く)。SLO を満たすには、このポッド起動レイテンシーのクラスター日あたりの 99 パーセンタイルが <=5 秒のままである必要があります。

この SLO は、ワーカーノードがこのクラスターに既に存在しており、ポッドをスケジュールする準備が整っていることを前提としていることに注意してください。この SLO は、イメージのプルや初期化コンテナの実行を考慮しておらず、永続的ストレージプラグインを活用しない「ステートレスポッド」にテストを制限します。

Kubernetes SLI メトリクス

Kubernetes は、これらの SLIs を経時的に追跡する Kubernetes コンポーネントに Prometheus メトリクスを追加することでSLIs に関するオブザーバビリティも改善しています。Prometheus クエリ言語 (PromQL) を使用すると、Prometheus や Grafana ダッシュボードなどのツールで SLI パフォーマンスを経時的に表示するクエリを構築できます。以下に、上記の SLOs の例をいくつか示します。

API サーバーリクエストのレイテンシー

メトリクス	定義
apiserver_request_sli_duration_seconds	動詞、グループ、バージョン、リソース、サブリソース、スコープ、コンポーネントごとのレスポンスレイテンシー分布 (ウェブフック期間、優先度と公平性キューの待機時間はカウントされません)。
apiserver_request_duration_seconds	verb、dry run 値、グループ、バージョン、リソース、サブリソース、スコープ、コンポーネントごとの、秒単位の応答遅延分布。

> **Note**
>
> `apiserver_request_sli_duration_seconds` メトリクスは Kubernetes 1.27 以降で使用できます。

これらのメトリクスを使用して、API Server の応答時間と、Kubernetes コンポーネントや他のプラグイン/コンポーネントにボトルネックがあるかどうかを調査できます。以下のクエリは、コミュニティ SLO ダッシュボードに基づいています。

API リクエストレイテンシー SLI (変更) - 今回は、ウェブフックの実行やキューでの待機時間は含まれません。 `histogram_quantile(0.99, sum(rate(apiserver_request_sli_duration_seconds_bucket{verb=~"CREATE|DELETE|PATCH|POST|PUT", subresource!~"proxy|attach|log|exec|portforward"}[5m])) by (resource, subresource, verb, scope, le)) > 0`

API リクエストレイテンシー合計 (変更) - これはリクエストが API サーバーにかかった合計時間です。ウェブフック実行と API Priority and Fairness 待機時間が含まれているため、この時間は SLI 時間よりも長くなる可能性があります。 `histogram_quantile(0.99, sum(rate(apiserver_request_duration_seconds_bucket{verb=~"CREATE|DELETE|PATCH|POST|PUT", subresource!~"proxy|attach|log|exec|portforward"}[5m])) by (resource, subresource, verb, scope, le)) > 0`

これらのクエリでは、 `kubectl port-forward`や リクエスト (`subresource!~"proxy|attach|log|exec|portforward"`) など、すぐに返されないストリーミング API `kubectl exec`リクエストを除外し、オブジェクトを変更する Kubernetes 動詞 () のみをフィルタリングします`verb=~"CREATE|DELETE|PATCH|POST|PUT"`。次に、過去 5 分間のレイテンシーの 99 パーセンタイルを計算します。

読み取り専用 API リクエストに同様のクエリを使用できます。フィルタリングする動詞を変更して、読み取り専用アクション `LIST`と を含めるだけです`GET`。1 つのリソースの取得や多数のリソースの一覧表示など、リクエストの範囲に応じて異なる SLO しきい値もあります。

API リクエストレイテンシー SLI (読み取り専用) - 今回は、ウェブフックの実行やキューでの待機時間は含まれません。単一のリソースの場合 (scope=resource、 threshold=1s) `histogram_quantile(0.99, sum(rate(apiserver_request_sli_duration_seconds_bucket{verb=~"GET", scope=~"resource"}[5m])) by (resource, subresource, verb, scope, le))`

同じ名前空間内のリソースのコレクション (scope=namespace、 threshold=5s) `histogram_quantile(0.99, sum(rate(apiserver_request_sli_duration_seconds_bucket{verb=~"LIST", scope=~"namespace"}[5m])) by (resource, subresource, verb, scope, le))`

クラスター全体のリソースのコレクションの場合 (スコープ = クラスター、しきい値 = 30 秒) `histogram_quantile(0.99,`

```
sum(rate(apiserver_request_sli_duration_seconds_bucket{verb=~"LIST",
scope=~"cluster"}[5m])) by (resource, subresource, verb, scope, le))
```

API リクエストレイテンシー合計 (読み取り専用) - これはリクエストが API サーバーにかかった合計時間であり、ウェブフックの実行と待機時間が含まれているため、この時間は SLI 時間よりも長くなる可能性があります。単一のリソースの場合 (scope=resource、 threshold=1s) `histogram_quantile(0.99, sum(rate(apiserver_request_duration_seconds_bucket{verb=~"GET", scope=~"resource"}[5m])) by (resource, subresource, verb, scope, le))`

同じ名前空間内のリソースのコレクション (scope=namespace、 threshold=5s) `histogram_quantile(0.99, sum(rate(apiserver_request_duration_seconds_bucket{verb=~"LIST", scope=~"namespace"}[5m])) by (resource, subresource, verb, scope, le))`

クラスター全体のリソースのコレクションの場合 (スコープ = クラスター、しきい値 = 30 秒) `histogram_quantile(0.99, sum(rate(apiserver_request_duration_seconds_bucket{verb=~"LIST", scope=~"cluster"}[5m])) by (resource, subresource, verb, scope, le))`

SLI メトリクスは、リクエストが API Priority and Fairness キューで待機したり、アドミッションウェブフックやその他の Kubernetes 拡張機能を使用したりする時間を除外することで、Kubernetes コンポーネントのパフォーマンスに関するインサイトを提供します。合計メトリクスは、アプリケーションが API サーバーからのレスポンスを待っている時間を反映するため、より包括的なビューを提供します。これらのメトリクスを比較すると、リクエスト処理の遅延が発生する場所を把握できます。

ポッドの起動レイテンシー

メトリクス	定義
kubelet_pod_start_sli_duration_seconds	ポッドの作成タイムスタンプからすべてのコンテナが開始と報告され、監視によって観測されるまでに測定される、イメージをプルして init コンテナを実行する時間を除く、ポッドを開始するまでの秒単位の期間
kubelet_pod_start_duration_seconds	kubelet がポッドを初めて確認してからポッドの実行が開始されるまでの秒単位の期間。これ

メトリクス	定義
	には、ポッドをスケジュールしたり、ワーカーノード容量をスケールアウトしたりする時間は含まれません。

> **Note**
>
> kubelet_pod_start_sli_duration_seconds は Kubernetes 1.27 以降で利用可能です。

上記のクエリと同様に、これらのメトリクスを使用して、Kubelet アクションと比較して、ノードスケーリング、イメージプル、初期化コンテナがポッドの起動を遅延させている時間に関するインサイトを得ることができます。

ポッドの起動レイテンシー SLI - これは、ポッドが作成されてからアプリケーションコンテナが実行中と報告されるまでの時間です。これには、ワーカーノードの容量が利用可能になり、ポッドがスケジュールされるまでにかかる時間が含まれますが、イメージのプルや init コンテナの実行にかかる時間は含まれません。 `histogram_quantile(0.99, sum(rate(kubelet_pod_start_sli_duration_seconds_bucket[5m])) by (le))`

ポッドの起動レイテンシー合計 - これは、kubelet がポッドを初めて起動するのにかかる時間です。これは、kubelet が WATCH 経由でポッドを受信したときから測定されます。これには、ワーカーノードのスケーリングまたはスケジューリングの時間は含まれません。これには、実行するイメージと初期化コンテナをプルする時間が含まれます。 `histogram_quantile(0.99, sum(rate(kubelet_pod_start_duration_seconds_bucket[5m])) by (le))`

クラスターSLOs

EKS クラスターの Kubernetes リソースから Prometheus メトリクスを収集している場合は、Kubernetes コントロールプレーンコンポーネントのパフォーマンスに関するより深いインサイトを得ることができます。

パフォーマンステストリポジトリには、テスト中のクラスターのレイテンシーと重要なパフォーマンスメトリクスを表示する Grafana ダッシュボードが含まれています。perf-tests 設定は、Kubernetes メトリクスを収集するように設定されたオープンソースプロジェクトである kube-prometheus-stack

を活用しますが、Amazon Managed Prometheus と Amazon Managed Grafana を使用することもできます。

kube-prometheus-stack または同様の Prometheus ソリューションを使用している場合は、同じダッシュボードをインストールしてクラスターの SLOs をリアルタイムで監視できます。

1. まず、 でダッシュボードで使用される Prometheus ルールをインストールする必要がありますkubectl apply -f prometheus-rules.yaml。ルールのコピーは、https://github.com/kubernetes/perf-tests/blob/master/clusterloader2/pkg/prometheus/manifests/prometheus-rules.yaml からダウンロードできます。
   a. ファイルの名前空間が環境と一致することを確認してください。
   b. を使用している場合は、ラベルが prometheus.prometheusSpec.ruleSelector helm 値と一致することを確認します。 kube-prometheus-stack
2. その後、Grafana にダッシュボードをインストールできます。生成する JSON ダッシュボードと Python スクリプトは、https://github.com/kubernetes/perf-tests/tree/master/clusterloader2/pkg/prometheus/manifests/dashboards から入手できます。
   a. slo.json ダッシュボードには、Kubernetes SLOs

SLOs はクラスター内の Kubernetes コンポーネントのパフォーマンスに焦点を当てていますが、クラスターにさまざまな視点やインサイトを提供する追加のメトリクスを確認できます。Kube-state-metrics などの Kubernetes コミュニティプロジェクトは、クラスターの傾向をすばやく分析するのに役立ちます。Kubernetes コミュニティの最も一般的なプラグインとドライバーも Prometheus メトリクスを出力するため、オートスケーラーやカスタムスケジューラなどを調査することができます。

オブザーバビリティのベストプラクティスガイドには、さらなるインサイトを得るために使用できる他の Kubernetes メトリクスの例が記載されています。

既知の制限とService Quotasタ

Amazon EKS はさまざまなワークロードに使用でき、さまざまな AWS サービスとやり取りできます。また、お客様のワークロードで同様の範囲の AWS サービスクォータや、スケーラビリティを妨げるその他の問題が発生しています。

AWS アカウントにはデフォルトのクォータがあります (チームがリクエストできる各 AWS リソースの数の上限) 。各 AWS サービスは独自のクォータを定義し、クォータは通常リージョン固有です。一部のクォータ (ソフト制限) の引き上げをリクエストでき、他のクォータは引き上げることが

できません (ハード制限) 。アプリケーションを設計するときは、これらの値を考慮する必要があります。これらのサービス制限を定期的に確認し、 中にアプリケーション設計に組み込むことを検討してください。

アカウントの使用状況を確認し、AWS Service Quotas コンソールまたは AWS CLI を使用してクォータ引き上げリクエストを開くことができます。Service Quotas の詳細と、その引き上げに関するその他の制限や通知については、各 AWS Service の AWS ドキュメントを参照してください。

> **Note**
>
> Amazon EKS Service Quotas は、サービスクォータを一覧表示し、利用可能な場合は引き上げをリクエストするためのリンクがあります。

その他の AWS Service Quotas

EKS のお客様は、他の AWS のサービスについて、以下に示すクォータの影響を受けています。これらの一部は特定のユースケースまたは設定にのみ適用される場合がありますが、ソリューションがスケーリングするときにこれらのいずれかに遭遇するかどうかを検討してください。クォータはサービス別に整理されており、各クォータには L-XXXXXXXX 形式の ID があり、AWS Service Quotas コンソールで検索するために使用できます。

サービス	クォータ (L-xxxxx)	Impact	ID (L-xxxxx)	デフォルト
IAM	アカウントあたりのロール数	アカウントのクラスターまたは IRSA ロールの数を制限できます。	L-FE177D64	1,000
IAM	アカウントあたりの OpenId 接続プロバイダー	アカウントあたりのクラスター数を制限できます。OpenID Connect は IRSA によって使用されます	L-858F3967	100

サービス	クォータ (L-xxxxx)	Impact	ID (L-xxxxx)	デフォルト
IAM	ロールの信頼ポリシーの長さ	IAM ロールが for IRSA に関連付けられているクラスターの数を制限できます	L-C07B4B0D	2,048
VPC	ネットワークインターフェイス当たりのセキュリティグループ	クラスターのネットワーク制御または接続を制限できる	L-2AFB9258	5
VPC	VPC 当たりの IPv4 CIDR ブロック	EKS ワーカーノードの数を制限できる	L-83CA0A9D	5
VPC	ルートテーブルあたりのルート	クラスターのネットワーク制御または接続を制限できる	L-93826ACB	50
VPC	VPC 当たりのアクティブな VPC ピアリング接続	クラスターのネットワーク制御または接続を制限できる	L-7E9ECCDB	50
VPC	セキュリティグループあたりのインバウンドルールまたはアウトバウンドルール。	クラスターのネットワークの制御または接続を制限できます。EKS の一部のコントローラーは新しいルールを作成します。	L-0EA8095F	50

サービス	クォータ (L-xxxxx)	Impact	ID (L-xxxxx)	デフォルト
VPC	リージョンあたりの VPC	アカウントあたりのクラスターの数、またはクラスターのネットワークの制御または接続を制限できます	L-F678F1CE	5
VPC	リージョンあたりのインターネットゲートウェイ	アカウントあたりのクラスターの数、またはクラスターのネットワークの制御または接続を制限できます	L-A4707A72	5
VPC	リージョンあたりのネットワークインターフェイス	EKS ワーカーノードの数、または Impact EKS コントロールプレーンのスケーリング/更新アクティビティを制限できます。	L-DF5E4CA3	5,000
VPC	ネットワークアドレスの使用量	アカウントあたりのクラスターの数、またはクラスターのネットワークの制御または接続を制限できます	L-BB24F6E5	64,000

サービス	クォータ (L-xxxxx)	Impact	ID (L-xxxxx)	デフォルト
VPC	ピアリングされたネットワークアドレスの使用量	アカウントあたりのクラスターの数、またはクラスターのネットワークの制御または接続を制限できます	L-CD17FD4B	128,000
ELB	Network Load Balancer あたりのリスナー	クラスターへのトラフィック進入の制御を制限できます。	L-57A373D6	50
ELB	リージョンあたりのターゲットグループ	クラスターへのトラフィック進入の制御を制限できます。	L-B22855CB	3,000
ELB	Application Load Balancer あたりのターゲット	クラスターへのトラフィック進入の制御を制限できます。	L-7E6692B2	1,000
ELB	Network Load Balancer あたりのターゲット	クラスターへのトラフィック進入の制御を制限できます。	L-EEF1AD04	3,000
ELB	Network Load Balancer ごとのアベイラビリティーゾーンあたりのターゲット	クラスターへのトラフィック進入の制御を制限できます。	L-B211E961	500

サービス	クォータ (L-xxxxx)	Impact	ID (L-xxxxx)	デフォルト
ELB	リージョンごとのターゲットグループあたりのターゲット	クラスターへのトラフィック進入の制御を制限できます。	L-A0D0B863	1,000
ELB	リージョンあたりの Application Load Balancer	クラスターへのトラフィック進入の制御を制限できます。	L-53DA6B97	50
ELB	リージョンあたりの Classic Load Balancer	クラスターへのトラフィック進入の制御を制限できます。	L-E9E9831D	20
ELB	リージョンあたりの Network Load Balancer	クラスターへのトラフィック進入の制御を制限できます。	L-69A177A2	50
EC2	オンデマンドスタンダード (A、C、D、H、I、M、R、T、Z) インスタンスの実行 (最大 vCPU 数として)	EKS ワーカーノードの数を制限できる	L-1216C47A	5
EC2	すべての標準 (A、C、D、H、I、M、R、T、Z) スポットインスタンスリクエスト (最大 vCPU 数)	EKS ワーカーノードの数を制限できる	L-34B43A08	5

サービス	クォータ (L-xxxxx)	Impact	ID (L-xxxxx)	デフォルト
EC2	EC2-VPC Elastic IP	NAT GWs (つまり VPCs) の数を制限でき、リージョン内のクラスターの数を制限できる	L-0263D0A3	5
EBS	リージョンあたりのスナップショット	ステートフルワークロードのバックアップ戦略を制限できる	L-309BACF6	100,000
EBS	汎用 SSD (gp3) ボリュームのストレージ (TiB)	EKS ワーカーノードまたはPersistentVolumeストレージの数を制限できます	L-7A658B76	50
EBS	汎用 SSD (gp2) ボリュームのストレージ (TiB)	EKS ワーカーノードまたはPersistentVolumeストレージの数を制限できます	L-D18FCD1D	50
ECR	登録済みリポジトリ	クラスター内のワークロードの数を制限できる	L-CFEB8E8D	100,000
ECR	リポジトリあたりのイメージ数	クラスター内のワークロードの数を制限できる	L-03A36CE1	20,000

サービス	クォータ (L-xxxxx)	Impact	ID (L-xxxxx)	デフォルト
SecretsManager	リージョンあたりのシークレット	クラスター内のワークロードの数を制限できる	L-2F66C23C	500,000

AWS リクエストスロットリング

また、AWS サービスはリクエストスロットリングを実装して、パフォーマンスを維持し、すべてのお客様が利用できるようにします。Service Quotas と同様に、各 AWS サービスは独自のリクエストスロットリングしきい値を維持します。ワークロードが大量の API コールを迅速に発行する必要がある場合や、アプリケーションでリクエストスロットリングエラーに気付いた場合は、それぞれの AWS サービスのドキュメントを確認することを検討してください。

EC2 ネットワークインターフェイスまたは IP アドレスのプロビジョニングに関する EC2 API リクエストでは、大規模なクラスターやクラスターの大幅なスケール時にリクエストスロットリングが発生する可能性があります。次の表は、顧客がリクエストスロットリングに遭遇した API アクションの一部を示しています。EC2 レート制限のデフォルトと、レートスロットリングに関する EC2 ドキュメントでレート制限の引き上げをリクエストする手順を確認できます。

アクションのミューテーション	読み取り専用アクション
AssignPrivateIpAddresses	DescribeDhcpOptions
AttachNetworkInterface	DescribeInstances
CreateNetworkInterface	DescribeNetworkInterfaces
DeleteNetworkInterface	DescribeSecurityGroups
DeleteTags	DescribeTags
DetachNetworkInterface	DescribeVpcs
ModifyNetworkInterfaceAttribute	DescribeVolumes
UnassignPrivateIpAddresses	

その他の既知の制限

- Route 53 には、Route 53 API への 1 秒あたり 5 リクエストというかなり低いレート制限もあります。外部 DNS などのプロジェクトで更新するドメインが多数ある場合、レートスロットリングとドメインの更新の遅延が発生することがあります。
  - 一部の Nitro インスタンスタイプでは、Amazon EBS ボリューム、ネットワークインターフェイス、および NVMe インスタンスストアボリューム間で共有されるボリュームアタッチメント制限が 28 です。 NVMe ワークロードが多数の EBS ボリュームをマウントしている場合、これらのインスタンスタイプで達成できるポッド密度に制限が発生する可能性があります。
  - Ec2 インスタンスごとに追跡できる接続の最大数があります。ワークロードが多数の接続を処理している場合、この最大値に達したため、通信の失敗やエラーが発生する可能性があります。`conntrack_allowance_available` および `conntrack_allowance_exceeded`ネットワークパフォーマンスメトリクスを使用して、EKS ワーカーノードで追跡された接続の数をモニタリングできます。
  - EKS 環境では、アップストリームガイダンスに従って etcd ストレージの制限は 8 GiB です。etcd db サイズを追跡`apiserver_storage_size_bytes`するには、メトリクスをモニタリングしてください。アラートルール`etcdBackendQuotaLowSpace`と を参照して`etcdExcessiveDatabaseGrowth`、このモニタリングを設定できます。

クラスターのアップグレードのベストプラクティス

このガイドでは、クラスター管理者が Amazon EKS アップグレード戦略を計画および実行する方法について説明します。また、セルフマネージド型ノード、マネージド型ノードグループ、Karpenter ノード、Fargate ノードをアップグレードする方法についても説明します。EKS Anywhere、セルフマネージド Kubernetes、AWS Outposts、または AWS Local Zones に関するガイダンスは含まれていません。

概要:

Kubernetes バージョンには、コントロールプレーンとデータプレーンの両方が含まれます。スムーズなオペレーションを実現するには、コントロールプレーンとデータプレーンの両方が 1.24 などの同じ Kubernetes マイナーバージョンを実行する必要があります。AWS はコントロールプレーンを管理およびアップグレードしますが、データプレーンのワーカーノードを更新するのはユーザーの責任です。

- コントロールプレーン — コントロールプレーンのバージョンは、Kubernetes API サーバーによって決まります。Amazon EKS クラスターでは、AWS がこのコンポーネントを管理します。コントロールプレーンのアップグレードは、AWS API を介して開始できます。
- データプレーン — データプレーンバージョンは、個々のノードで実行されている Kubelet バージョンに関連付けられます。同じクラスター内のノードが異なるバージョンを実行している可能性があります。を実行することで、すべてのノードのバージョンを確認できます`kubectl get nodes`。

アップグレード前

Amazon EKS で Kubernetes バージョンをアップグレードする場合は、アップグレードを開始する前にいくつかの重要なポリシー、ツール、手順を実施する必要があります。

- 非推奨ポリシーを理解する — Kubernetes 非推奨ポリシーの仕組みを深く理解します。既存のアプリケーションに影響を与える可能性のある今後の変更に注意してください。Kubernetes の新しいバージョンでは、多くの場合、特定の APIsや機能が段階的に廃止されるため、アプリケーションの実行に問題が発生する可能性があります。
- Kubernetes 変更ログの確認 — Amazon EKS Kubernetes バージョンとともに Kubernetes 変更ログを徹底的に確認し、ワークロードに影響を与える可能性のある変更の中断など、クラスターへの影響の可能性を理解します。

- クラスターアドオンの互換性の評価 — Amazon EKS は、新しいバージョンがリリースされたとき、またはクラスターを新しい Kubernetes マイナーバージョンに更新した後、アドオンを自動的に更新しません。アドオンの更新を確認して、アップグレード先のクラスターバージョンとの既存のクラスターアドオンの互換性を理解します。
- コントロールプレーンのログ記録を有効にする — コントロールプレーンのログ記録を有効にして、アップグレードプロセス中に発生する可能性のあるログ、エラー、または問題をキャプチャします。異常がないか、これらのログを確認することを検討してください。非本番環境でクラスターのアップグレードをテストするか、自動テストを継続的な統合ワークフローに統合して、アプリケーション、コントローラー、カスタム統合とのバージョン互換性を評価します。
- クラスター管理用の eksctl を調べる — eksctl を使用して EKS クラスターを管理することを検討してください。これにより、コントロールプレーンの更新、アドオンの管理、ワーカーノードの更新の処理out-of-the-box行うことができます。
- 自動モードまたはマネージド型ノードグループのオプト — 自動モードまたは EKS マネージド型ノードグループを使用して、ワーカーノードのアップグレードを合理化および自動化します。これらのオプションはプロセスを簡素化し、手動による介入を減らします。
- kubectl 変換プラグインを利用する — kubectl 変換プラグインを活用して、さまざまな API バージョン間の Kubernetes マニフェストファイルの変換を容易にします。これにより、設定が新しい Kubernetes バージョンと互換性を保つことができます。

クラスターup-to-dateに保つ

Kubernetes の更新を最新の状態に保つことは、Amazon EKS の責任共有モデルを反映した、安全で効率的な EKS 環境にとって最優先事項です。これらの戦略を運用ワークフローに統合することで、最新の機能と改善を最大限に活用するup-to-date安全なクラスターを維持できます。戦術:

- サポートされているバージョンポリシー — Kubernetes コミュニティに合わせて、Amazon EKS は通常 3 つのアクティブな Kubernetes バージョンを提供します。Kubernetes マイナーバージョンは、リリース後最初の 14 か月間、Amazon EKS で標準サポートされています。標準サポート終了日を過ぎたバージョンは次の 12 か月間の延長サポートに入ります。非推奨通知は、バージョンが標準サポート終了日に達する少なくとも 60 日前に発行されます。詳細については、「EKS バージョンライフサイクルドキュメント」を参照してください。
- 自動アップグレードポリシー — EKS クラスターの Kubernetes 更新と同期することを強くお勧めします。26 か月のライフサイクル (14 か月の標準サポートと 12 か月の延長サポート) が終了した Kubernetes バージョンで実行されているクラスターは、次のバージョンに自動的にアップグレードされます。延長サポートは無効にできることに注意してください。バージョンend-of-lifeる前に

プロアクティブにアップグレードしないと、自動アップグレードがトリガーされ、ワークロードとシステムが中断される可能性があります。詳細については、「EKS バージョンに関するFAQs」を参照してください。

- アップグレードランブックの作成 — アップグレードを管理するための文書化されたプロセスを確立します。プロアクティブアプローチの一環として、アップグレードプロセスに合わせたランブックと特殊なツールを開発します。これにより、準備が強化されるだけでなく、複雑な移行も簡素化されます。クラスターを少なくとも 1 年に 1 回アップグレードするのが標準的な方法です。このプラクティスにより、継続的な技術的進歩に合わせて、環境の効率とセキュリティが向上します。

EKS リリースカレンダーを確認する

EKS Kubernetes リリースカレンダーを確認して、新しいバージョンがいつリリースされるか、および特定のバージョンのサポートが終了するかを確認してください。一般的に、EKS は毎年 3 つのマイナーバージョンの Kubernetes をリリースし、各マイナーバージョンは約 14 か月間サポートされています。

さらに、アップストリームの Kubernetes リリース情報を確認します。

責任共有モデルがクラスターのアップグレードにどのように適用されるかを理解する

クラスターコントロールプレーンとデータプレーンの両方のアップグレードを開始する責任があります。アップグレードを開始する方法について説明します。クラスターのアップグレードを開始すると、AWS はクラスターコントロールプレーンのアップグレードを管理します。Fargate ポッドやアドオンなど、データプレーンのアップグレードはお客様の責任となります。クラスターで実行されているワークロードのアップグレードを検証して計画し、クラスターのアップグレード後に可用性とオペレーションに影響がないことを確認する必要があります。

クラスターをインプレースでアップグレードする

EKS は、インプレースクラスターアップグレード戦略をサポートしています。これにより、クラスターリソースが維持され、クラスター設定の一貫性が維持されます (API エンドポイント、OIDC、ENIs、ロードバランサーなど) 。これはクラスターユーザーにとってそれほど破壊的ではなく、ワークロードを再デプロイしたり、外部リソース (DNS、ストレージなど) を移行したりすることなく、クラスター内の既存のワークロードとリソースを使用します。

インプレースクラスターアップグレードを実行する場合、一度に実行できるマイナーバージョンアップグレードは 1 つだけであることに注意してください (例: 1.24 から 1.25)。

つまり、複数のバージョンを更新する必要がある場合は、一連のシーケンシャルアップグレードが必要になります。シーケンシャルアップグレードの計画はより複雑で、ダウンタイムのリスクが高くなります。このような場合は、「」を参照してくださいthe section called “インプレースクラスターアップグレードの代わりに Blue/Green クラスターを評価する”。

コントロールプレーンとデータプレーンを順番にアップグレードする

クラスターをアップグレードするには、次のアクションを実行する必要があります。

1. Kubernetes および EKS リリースノートを確認します。
2. クラスターのバックアップを取ります (オプション)。
3. ワークロードでの廃止および削除された API 使用状況を特定して修正します。
4. Managed Node Groups が使用されている場合は、コントロールプレーンと同じ Kubernetes バージョンにあることを確認します。EKS Fargate Profiles によって作成された EKS マネージド型ノードグループとノードは、Kubernetes バージョン 1.27 以下のコントロールプレーンとデータプレーン間の 2 つのマイナーバージョンスキューをサポートしています。1.28 以降、EKS Fargate Profiles によって作成された EKS マネージド型ノードグループとノードは、コントロールプレーンとデータプレーン間で 3 つのマイナーバージョンスキューをサポートしています。たとえば、EKS コントロールプレーンのバージョンが 1.28 の場合、1.25 以前の kubelet バージョンを安全に使用できます。EKS バージョンが 1.27 の場合、使用できる最も古い kubelet バージョンは 1.25 です。
5. AWS コンソールまたは CLI を使用してクラスターコントロールプレーンをアップグレードします。
6. アドオンの互換性を確認します。必要に応じて、Kubernetes アドオンとカスタムコントローラーをアップグレードします。
7. kubectl を更新します。
8. クラスターデータプレーンをアップグレードします。ノードをアップグレードしたクラスターと同じ Kubernetes マイナーバージョンにアップグレードします。

> **Tip**
>
> クラスターが EKS Auto Mode を使用して作成された場合は、クラスターデータプレーンをアップグレードする必要はありません。コントロールプレーンをアップグレードすると、EKS Auto Mode はすべてのポッド中断予算を尊重しながら、マネージドノードの段階的な更新を開始します。運用要件への準拠を確認するために、必ずこれらの更新をモニタリングしてください。

EKS ドキュメントを使用してアップグレードチェックリストを作成する

EKS Kubernetes バージョンドキュメントには、各バージョンの変更の詳細なリストが含まれています。アップグレードごとにチェックリストを作成します。

特定の EKS バージョンアップグレードガイダンスについては、ドキュメントで各バージョンにおける重要な変更点と考慮事項を確認してください。

- 標準サポートで Kubernetes バージョンのリリースノートを確認する
- 延長サポートに関する Kubernetes バージョンのリリースノートを確認する

Kubernetes API を使用してアドオンとコンポーネントをアップグレードする

クラスターをアップグレードする前に、使用している Kubernetes コンポーネントのバージョンを理解しておく必要があります。クラスターコンポーネントをインベントリし、Kubernetes API を直接使用するコンポーネントを特定します。これには、モニタリングおよびログ記録エージェント、クラスターオートスケーラー、コンテナストレージドライバー (EBS CSI、EFS CSI など) 、イングレスコントローラー、Kubernetes API に直接依存するその他のワークロードやアドオンなどの重要なクラスターコンポーネントが含まれます。

> **Tip**
>
> 重要なクラスターコンポーネントは、多くの場合、`*-system`名前空間にインストールされます。

```
kubectl get ns | grep '-system'
```

Kubernetes API に依存するコンポーネントを特定したら、そのドキュメントでバージョンの互換性とアップグレード要件を確認してください。たとえば、バージョンの互換性については、AWS Load Balancer Controller のドキュメントを参照してください。一部のコンポーネントは、クラスターのアップグレードに進む前にアップグレードまたは設定の変更が必要になる場合があります。チェックすべき重要なコンポーネントには、CoreDNS、kube-proxy、VPC CNI、ストレージドライバーなどがあります。

クラスターには、Kubernetes API を使用する多くのワークロードが含まれており、イングレスコントローラー、継続的デリバリーシステム、モニタリングツールなどのワークロード機能に必要です。EKS クラスターをアップグレードするときは、アドオンとサードパーティーツールもアップグレードして、互換性があることを確認する必要があります。

一般的なアドオンの以下の例と、関連するアップグレードドキュメントを参照してください。

- Amazon VPC CNI: 各クラスターバージョンの Amazon VPC CNI アドオンの推奨バージョンについては、「Amazon VPC CNI plugin for Kubernetes セルフマネージドアドオンの更新」を参照してください。Amazon EKS アドオンとしてインストールする場合、一度にアップグレードできるマイナーバージョンは 1 つだけです。
- kube-proxy:「Kubernetes kube-proxy セルフマネージドアドオンの更新」を参照してください。
- CoreDNS: CoreDNS セルフマネージドアドオンの更新を参照してください。
- AWS Load Balancer Controller: AWS Load Balancer Controller は、デプロイした EKS バージョンと互換性がある必要があります。詳細については、インストールガイドを参照してください。
- Amazon Elastic Block Store (Amazon EBS) Container Storage Interface (CSI) ドライバー: インストールおよびアップグレード情報については、「Amazon EKS アドオンとしての Amazon EBS CSI ドライバーの管理」を参照してください。
- Amazon Elastic File System (Amazon EFS) Container Storage Interface (CSI) ドライバー: インストールとアップグレードの情報については、「Amazon EFS CSI ドライバー」を参照してください。
- Kubernetes メトリクスサーバー: 詳細については、GitHub の「メトリクスサーバー」を参照してください。
- Kubernetes Cluster Autoscaler: Kubernetes Cluster Autoscaler のバージョンをアップグレードするには、デプロイ内のイメージのバージョンを変更します。Cluster Autoscaler は Kubernetes スケジューラと緊密に結合されています。クラスターをアップグレードするときは、常にアップグレー

ドする必要があります。GitHub リリースを確認して、Kubernetes マイナーバージョンに対応する最新リリースのアドレスを見つけます。

- Karpenter: インストールとアップグレードの情報については、Karpenter ドキュメントを参照してください。

> **Tip**
>
> コンピューティングの自動スケーリング、ブロックストレージ、ロードバランシング機能など、Amazon EKS Auto Mode の機能を手動でアップグレードする必要はありません。

アップグレード前に基本的な EKS 要件を確認する

AWS では、アップグレードプロセスを完了するために、アカウント内の特定のリソースが必要です。これらのリソースが存在しない場合、クラスターをアップグレードすることはできません。コントロールプレーンのアップグレードには、次のリソースが必要です。

1. 使用可能な IP アドレス: Amazon EKS では、クラスターを更新するためにクラスターの作成時に指定したサブネットから最大 5 つの使用可能な IP アドレスが必要です。そうでない場合は、バージョン更新を実行する前に、クラスター設定を更新して新しいクラスターサブネットを含めます。
2. EKS IAM ロール: コントロールプレーン IAM ロールは、必要なアクセス許可を持つアカウントにまだ存在します。
3. クラスターでシークレット暗号化が有効になっている場合は、クラスター IAM ロールに AWS Key Management Service (AWS KMS) キーを使用するアクセス許可があることを確認してください。

使用可能な IP アドレスを検証する

クラスターをアップグレードする際、Amazon EKS には、クラスターの作成時に指定したサブネット内に、最大で 5 つの使用可能な IP アドレスが必要となります。

サブネットにクラスターをアップグレードするのに十分な IP アドレスがあることを確認するには、次のコマンドを実行します。

```
CLUSTER=<cluster name>
```

```
aws ec2 describe-subnets --subnet-ids \
  $(aws eks describe-cluster --name ${CLUSTER} \
  --query 'cluster.resourcesVpcConfig.subnetIds' \
  --output text) \
  --query 'Subnets[*].[SubnetId,AvailabilityZone,AvailableIpAddressCount]' \
  --output table

-------------------------------------------------------
|                  DescribeSubnets                    |
+---------------------------+--------------+-------+
|  subnet-067fa8ee8476abbd6 |  us-east-1a  |  8184 |
|  subnet-0056f7403b17d2b43 |  us-east-1b  |  8153 |
|  subnet-09586f8fb3addbc8c |  us-east-1a  |  8120 |
|  subnet-047f3d276a22c6bce |  us-east-1b  |  8184 |
+---------------------------+--------------+-------+
```

VPC CNI メトリクスヘルパーを使用して、VPC メトリクスの CloudWatch ダッシュボードを作成できます。Amazon EKS では、クラスターの作成時に最初に指定されたサブネットの IP アドレスが不足している場合、Kubernetes バージョンのアップグレードを開始する前に UpdateClusterConfiguration」 API を使用してクラスターサブネットを更新することをお勧めします。新しいサブネットが提供されることを確認してください。

- クラスターの作成時に選択された同じ AZsします。
- クラスターの作成時に提供されたのと同じ VPC に属している

既存の VPC CIDR ブロックの IP アドレスが不足する場合は、追加の CIDR ブロックの関連付けを検討してください。AWS では、追加の CIDR ブロックを既存のクラスター VPC に関連付けることができ、IP アドレスプールを効果的に拡張できます。この拡張は、追加のプライベート IP 範囲 (RFC 1918)、または必要に応じてパブリック IP 範囲 (RFC 1918 以外) を導入することで実現できます。Amazon EKS が新しい CIDR を使用するには、新しい VPC CIDR ブロックを追加し、VPC 更新の完了を許可する必要があります。その後、新しく設定された CIDR ブロックに基づいてサブネットを VPC に更新できます。

EKS IAM ロールを検証する

IAM ロールが利用可能で、アカウントに正しい継承ロールポリシーがあることを確認するには、次のコマンドを実行します。

```
CLUSTER=<cluster name>
ROLE_ARN=$(aws eks describe-cluster --name ${CLUSTER} \
```

```
  --query 'cluster.roleArn' --output text)
aws iam get-role --role-name ${ROLE_ARN##*/} \
  --query 'Role.AssumeRolePolicyDocument'

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

EKS アドオンへの移行

Amazon EKS は、クラスターごとに Amazon VPC CNI plugin for Kubernetes、kube-proxy、CoreDNS などのアドオンを自動的にインストールします。アドオンはセルフマネージド型でも、Amazon EKS アドオンとしてインストールすることもできます。Amazon EKS アドオンは、EKS API を使用してアドオンを管理する代替方法です。

Amazon EKS アドオンを使用して、1 つのコマンドでバージョンを更新できます。例:

```
aws eks update-addon –cluster-name my-cluster –addon-name vpc-cni –addon-version
 version-number \
--service-account-role-arn arn:aws:iam::111122223333:role/role-name –configuration-
values '{}' –resolve-conflicts PRESERVE
```

次の EKS アドオンがあるかどうかを確認します。

```
aws eks list-addons --cluster-name <cluster name>
```

> **Warning**
>
> EKS アドオンは、コントロールプレーンのアップグレード中に自動的にアップグレードされません。EKS アドオンの更新を開始し、目的のバージョンを選択する必要があります。

- 利用可能なすべてのバージョンから互換性のあるバージョンを選択する責任があります。アドオンバージョンの互換性に関するガイダンスを確認してください。
- Amazon EKS アドオンは、一度に 1 つのマイナーバージョンしかアップグレードできません。

EKS アドオンとして利用できるコンポーネントと、使用を開始する方法について説明します。

EKS アドオンにカスタム設定を提供する方法について説明します。

コントロールプレーンをアップグレードする前に、削除された API の使用状況を特定して修正する

EKS コントロールプレーンをアップグレードする前に、削除APIs 使用状況を特定する必要があります。そのためには、実行中のクラスターまたは静的にレンダリングされた Kubernetes マニフェストファイルをチェックできるツールを使用することをお勧めします。

静的マニフェストファイルに対してチェックを実行する方が一般的に正確です。ライブクラスターに対して実行すると、これらのツールは誤検出を返す可能性があります。

非推奨の Kubernetes API は、API が削除されたことを意味するものではありません。Kubernetes 廃止ポリシーを確認して、API の削除がワークロードにどのように影響するかを理解する必要があります。

クラスターインサイト

Cluster Insights は、EKS クラスターを新しいバージョンの Kubernetes にアップグレードする機能に影響を与える可能性のある問題に関する検出結果を提供する機能です。これらの検出結果は Amazon EKS によってキュレートおよび管理され、それらを修復する方法に関する推奨事項を提供します。Cluster Insights を活用することで、新しい Kubernetes バージョンへのアップグレードに費やす労力を最小限に抑えることができます。

EKS クラスターのインサイトを表示するには、 コマンドを実行します。

```
aws eks list-insights --region <region-code> --cluster-name <my-cluster>

{
    "insights": [
        {
            "category": "UPGRADE_READINESS",
            "name": "Deprecated APIs removed in Kubernetes v1.29",
```

```
            "insightStatus": {
                "status": "PASSING",
                "reason": "No deprecated API usage detected within the last 30 days."
            },
            "kubernetesVersion": "1.29",
            "lastTransitionTime": 1698774710.0,
            "lastRefreshTime": 1700157422.0,
            "id": "123e4567-e89b-42d3-a456-579642341238",
            "description": "Checks for usage of deprecated APIs that are scheduled for
 removal in Kubernetes v1.29. Upgrading your cluster before migrating to the updated
 APIs supported by v1.29 could cause application impact."
        }
    ]
}
```

受信したインサイトに関するより詳細な出力については、 コマンドを実行できます。

```
aws eks describe-insight --region <region-code> --id <insight-id> --cluster-name <my-
cluster>
```

Amazon EKS コンソールでインサイトを表示することもできます。クラスターリストからクラスターを選択すると、インサイトの検出結果は Upgrade Insightsタブの下に表示されます。

でクラスターインサイトが見つかった場合は"status": ERROR、クラスターのアップグレードを実行する前に問題に対処する必要があります。次の修復アドバイスを共有する aws eks describe-insight コマンドを実行します。

影響を受けるリソース:

```
"resources": [
      {
        "insightStatus": {
          "status": "ERROR"
        },
        "kubernetesResourceUri": "/apis/policy/v1beta1/podsecuritypolicies/null"
      }
]
```

廃止された APIs:

```
"deprecationDetails": [
      {
```

```
        "usage": "/apis/flowcontrol.apiserver.k8s.io/v1beta2/flowschemas",
        "replacedWith": "/apis/flowcontrol.apiserver.k8s.io/v1beta3/flowschemas",
        "stopServingVersion": "1.29",
        "clientStats": [],
        "startServingReplacementVersion": "1.26"
      }
]
```

推奨されるアクション:

```
"recommendation": "Update manifests and API clients to use newer Kubernetes APIs if
 applicable before upgrading to Kubernetes v1.26."
```

EKS コンソールまたは CLI を使用してクラスターインサイトを利用すると、EKS クラスターバージョンを正常にアップグレードするプロセスが高速化されます。詳細については、* 公式 EKS Docs * Cluster Insights 起動ブログを参照してください。

Kube-no-trouble

Kube-no-trouble は、コマンド を使用するオープンソースのコマンドラインユーティリティですkubent。引数kubentなしで を実行すると、現在の KubeConfig コンテキストを使用してクラスターをスキャンし、廃止および削除する APIsします。

```
kubent

4:17PM INF >>> Kube No Trouble `kubent` <<<
4:17PM INF version 0.7.0 (git sha d1bb4e5fd6550b533b2013671aa8419d923ee042)
4:17PM INF Initializing collectors and retrieving data
4:17PM INF Target K8s version is 1.24.8-eks-ffeb93d
4:1 INF Retrieved 93 resources from collector name=Cluster
4:17PM INF Retrieved 16 resources from collector name="Helm v3"
4:17PM INF Loaded ruleset name=custom.rego.tmpl
4:17PM INF Loaded ruleset name=deprecated-1-16.rego
4:17PM INF Loaded ruleset name=deprecated-1-22.rego
4:17PM INF Loaded ruleset name=deprecated-1-25.rego
4:17PM INF Loaded ruleset name=deprecated-1-26.rego
4:17PM INF Loaded ruleset name=deprecated-future.rego
_____________________________________________________________________________________
>>> Deprecated APIs removed in 1.25 <<<
-------------------------------------------------------------------------------------
KIND                NAMESPACE     NAME             API_VERSION      REPLACE_WITH
 (SINCE)
```

```
PodSecurityPolicy    <undefined>    eks.privileged    policy/v1beta1    <removed> (1.21.0)
```

また、静的マニフェストファイルと helm パッケージをスキャンするためにも使用できます。マニフェストがデプロイされる前に問題を特定するために、継続的インテグレーション (CI) プロセスkubentの一部として を実行することをお勧めします。マニフェストのスキャンは、ライブクラスターのスキャンよりも正確です。

Kube-no-trouble は、クラスターをスキャンするための適切なアクセス許可を持つサンプルサービスアカウントとロールを提供します。

冥王星

もう 1 つのオプションは、ライブクラスター、マニフェストファイル、helm チャートのスキャンをサポートし、CI プロセスに含めることができる GitHub アクションがあるため、 に似kubentた pluto です。

```
pluto detect-all-in-cluster

NAME             KIND                VERSION          REPLACEMENT   REMOVED
 DEPRECATED   REPL AVAIL
eks.privileged   PodSecurityPolicy   policy/v1beta1                 false     true
    true
```

リソース

アップグレード前にクラスターが非推奨APIs を使用していないことを確認するには、以下をモニタリングする必要があります。

- Kubernetes v1.19 `apiserver_requested_deprecated_apis`以降の メトリクス:

```
kubectl get --raw /metrics | grep apiserver_requested_deprecated_apis

apiserver_requested_deprecated_apis{group="policy",removed_release="1.25",resource="podsecurity
 1
```

- が `k8s.io/deprecated`に設定されている監査ログの イベント`true`:

```
CLUSTER="<cluster_name>"
```

```
QUERY_ID=$(aws logs start-query \
 --log-group-name /aws/eks/${CLUSTER}/cluster \
 --start-time $(date -u --date="-30 minutes" "+%s") # or date -v-30M "+%s" on MacOS \
 --end-time $(date "+%s") \
 --query-string 'fields @message | filter `annotations.k8s.io/deprecated`="true"' \
 --query queryId --output text)

echo "Query started (query id: $QUERY_ID), please hold ..." && sleep 5 # give it some
 time to query

aws logs get-query-results --query-id $QUERY_ID
```

非推奨の APIsは行を出力します。

```
{
    "results": [
        [
            {
                "field": "@message",
                "value": "{\"kind\":\"Event\",\"apiVersion\":\"audit.k8s.io/v1\",
\"level\":\"Request\",\"auditID\":\"8f7883c6-b3d5-42d7-967a-1121c6f22f01\",\"stage
\":\"ResponseComplete\",\"requestURI\":\"/apis/policy/v1beta1/podsecuritypolicies?
allowWatchBookmarks=true\\u0026resourceVersion=4131\\u0026timeout=9m19s\
\u0026timeoutSeconds=559\\u0026watch=true\",\"verb\":\"watch\",\"user\":{\"username
\":\"system:apiserver\",\"uid\":\"8aabfade-da52-47da-83b4-46b16cab30fa\",
\"groups\":[\"system:masters\"]},\"sourceIPs\":[\"::1\"],\"userAgent\":\"kube-
apiserver/v1.24.16 (linux/amd64) kubernetes/af930c1\",\"objectRef\":{\"resource
\":\"podsecuritypolicies\",\"apiGroup\":\"policy\",\"apiVersion\":\"v1beta1\"},
\"responseStatus\":{\"metadata\":{},\"code\":200},\"requestReceivedTimestamp\":
\"2023-10-04T12:36:11.849075Z\",\"stageTimestamp\":\"2023-10-04T12:45:30.850483Z\",
\"annotations\":{\"authorization.k8s.io/decision\":\"allow\",\"authorization.k8s.io/
reason\":\"\",\"k8s.io/deprecated\":\"true\",\"k8s.io/removed-release\":\"1.25\"}}"
            },
[...]
```

Kubernetes ワークロードを更新します。kubectl-convert を使用してマニフェストを更新する

更新する必要があるワークロードとマニフェストを特定したら、マニフェストファイルのリソースタイプ (PodSecurityPolicies を PodSecurityStandards など) を変更する必要があります。これには、置き換えるリソースに応じて、リソース仕様を更新し、追加の調査を行う必要があります。

リソースタイプが同じままで API バージョンを更新する必要がある場合は、 kubectl-convert コマンドを使用してマニフェストファイルを自動的に変換できます。たとえば、古いデプロイを に変換しますapps/v1。詳細については、Kubernetes ウェブサイトの「kubectl convert plugin のインストール」を参照してください。

```
kubectl-convert -f <file> --output-version <group>/<version>
```

データプレーンのアップグレード中にワークロードの可用性を確保するために、PodDisruptionBudgets と topologySpreadConstraints を設定する

データプレーンのアップグレード中にワークロードの可用性を確保するために、ワークロードに適切な PodDisruptionBudgets と topologySpreadConstraints があることを確認します。すべてのワークロードが同じレベルの可用性を必要とするわけではないため、ワークロードのスケールと要件を検証する必要があります。

ワークロードが複数のアベイラビリティーゾーンに分散され、トポロジーが分散された複数のホストに分散されていることを確認すると、ワークロードがインシデントなしで新しいデータプレーンに自動的に移行する信頼度が高まります。

レプリカの 80% が常に利用可能で、ゾーンとホスト間でレプリカを分散するワークロードの例を次に示します。

```
apiVersion: policy/v1
kind: PodDisruptionBudget
metadata:
  name: myapp
spec:
  minAvailable: "80%"
  selector:
    matchLabels:
      app: myapp
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp
spec:
  replicas: 10
  selector:
```

```
    matchLabels:
      app: myapp
  template:
    metadata:
      labels:
        app: myapp
    spec:
      containers:
      - image: public.ecr.aws/eks-distro/kubernetes/pause:3.2
        name: myapp
        resources:
          requests:
            cpu: "1"
            memory: 256M
      topologySpreadConstraints:
      - labelSelector:
          matchLabels:
            app: host-zone-spread
        maxSkew: 2
        topologyKey: kubernetes.io/hostname
        whenUnsatisfiable: DoNotSchedule
      - labelSelector:
          matchLabels:
            app: host-zone-spread
        maxSkew: 2
        topologyKey: topology.kubernetes.io/zone
        whenUnsatisfiable: DoNotSchedule
```

AWS Resilience Hub は、サポートされているリソースとして Amazon Elastic Kubernetes Service (Amazon EKS) を追加しました。Resilience Hub は、アプリケーションの耐障害性を定義、検証、追跡するための単一の場所を提供するため、ソフトウェア、インフラストラクチャ、または運用の中断による不要なダウンタイムを回避できます。

Managed Node Groups または Karpenter を使用してデータプレーンのアップグレードを簡素化する

マネージド型ノードグループと Karpenter はどちらもノードのアップグレードを簡素化しますが、アプローチは異なります。

マネージド型ノードグループは、ノードのプロビジョニングとライフサイクル管理を自動化します。つまり、1 回のオペレーションでノードを作成、自動更新、または終了できます。

デフォルト設定では、Karpenter は互換性のある最新の EKS Optimized AMI を使用して新しいノードを自動的に作成します。EKS が更新された EKS 最適化 AMIsリリースするか、クラスターがアップグレードされると、Karpenter は自動的にこれらのイメージの使用を開始します。Karpenter は Node Expiry を実装してノードを更新します。

Karpenter は、カスタム AMIs を使用するように設定できます。Karpenter でカスタム AMIs を使用する場合、kubelet のバージョンはお客様の責任となります。

既存のノードとコントロールプレーンとのバージョン互換性を確認する

Amazon EKS で Kubernetes アップグレードを進める前に、マネージド型ノードグループ、セルフマネージド型ノード、コントロールプレーン間の互換性を確保することが重要です。互換性は、使用している Kubernetes バージョンによって決まり、シナリオによって異なります。戦術:

- Kubernetes v1.28+ — ** Kubernetes バージョン 1.28 以降では、コアコンポーネントのより寛容なバージョンポリシーがあります。具体的には、Kubernetes API サーバーと kubelet 間のサポートされているスキューは、n-2 から n-3 の 1 つのマイナーバージョンで拡張されています。たとえば、EKS コントロールプレーンのバージョンが 1.28 の場合、1.25 以前の kubelet バージョンを安全に使用できます。このバージョンスキューは、AWS Fargate、マネージド型ノードグループ、セルフマネージド型ノードでサポートされています。セキュリティ上の理由から、Amazon マシンイメージ (AMI) up-to-date状態に保つことを強くお勧めします。古い kubelet バージョンは、潜在的な共通脆弱性識別子 (CVEs) が原因でセキュリティリスクをもたらす可能性があります。これは、古い kubelet バージョンを使用する利点を上回る可能性があります。
- Kubernetes < v1.28 — v1.28 より前のバージョンを使用している場合、API サーバーと kubelet の間でサポートされているスキューは n-2 です。たとえば、EKS バージョンが 1.27 の場合、使用できる最も古い kubelet バージョンは 1.25 です。このバージョンスキューは、AWS Fargate、マネージド型ノードグループ、セルフマネージド型ノードに適用されます。

Karpenter マネージドノードのノード有効期限を有効にする

Karpenter がノードのアップグレードを実装する 1 つの方法は、ノードの有効期限の概念を使用することです。これにより、ノードのアップグレードに必要な計画が削減されます。プロビジョナーで ttlSecondsUntilExpired の値を設定すると、ノードの有効期限が有効になります。ノードが秒単位で定義された経過時間に達すると、ノードは安全にドレインおよび削除されます。これは使用中であっても当てはまります。ノードを新しくプロビジョニングされたアップグレードされたインスタンスに

置き換えることができます。ノードが置き換えられると、Karpenter は最新の EKS 最適化 AMIs を使用します。詳細については、Karpenter ウェブサイトの「プロビジョニング解除」を参照してください。

Karpenter は、この値にジッターを自動的に追加しません。ワークロードの過剰な中断を防ぐには、Kubernetes ドキュメントに示すように、ポッドの中断予算を定義します。

プロビジョナーで ttlSecondsUntilExpired を設定する場合、これはプロビジョナーに関連付けられた既存のノードに適用されます。

Karpenter マネージドノードにドリフト機能を使用する

Karpenter のドリフト機能は、Karpenter でプロビジョニングされたノードを自動的にアップグレードして、EKS コントロールプレーンと同期させることができます。Karpenter Drift は現在、機能ゲートを使用して有効にする必要があります。Karpenter のデフォルト設定では、EKS クラスターのコントロールプレーンと同じメジャーバージョンとマイナーバージョンで最新の EKS 最適化 AMI が使用されます。

EKS クラスターのアップグレードが完了すると、Karpenter のドリフト機能は、Karpenter でプロビジョニングされたノードが以前のクラスターバージョンで EKS 最適化 AMIs を使用していることを検出し、それらのノードを自動的に接続、ドレイン、および置き換えます。新しいノードへのポッドの移動をサポートするには、適切なポッドリソースクォータを設定し、ポッド中断予算 (PDB) を使用して Kubernetes のベストプラクティスに従います。Karpenter のプロビジョニング解除は、ポッドリソースリクエストに基づいて代替ノードを事前にスピンアップし、ノードのプロビジョニング解除時に PDBs を尊重します。

eksctl を使用してセルフマネージド型ノードグループのアップグレードを自動化する

セルフマネージド型ノードグループは、アカウントにデプロイされ、EKS サービス外のクラスターにアタッチされた EC2 インスタンスです。これらは通常、何らかの自動化ツールによってデプロイおよび管理されます。セルフマネージド型ノードグループをアップグレードするには、ツールのドキュメントを参照してください。

たとえば、eksctl はセルフマネージド型ノードの削除とドレインをサポートしています。

一般的なツールには、次のようなものがあります。

- eksctl

- kOps
- EKS 設計図

アップグレード前にクラスターをバックアップする

Kubernetes の新しいバージョンでは、Amazon EKS クラスターに大幅な変更が導入されています。クラスターをアップグレードした後は、ダウングレードできません。

Velero は、コミュニティがサポートするオープンソースのツールで、既存のクラスターのバックアップを取得し、新しいクラスターにバックアップを適用するために使用できます。

EKS で現在サポートされている Kubernetes バージョンの新しいクラスターのみを作成できます。クラスターが現在実行されているバージョンがまだサポートされていて、アップグレードが失敗した場合、元のバージョンで新しいクラスターを作成し、データプレーンを復元できます。IAM を含む AWS リソースは、Velero によるバックアップに含まれていないことに注意してください。これらのリソースは再作成する必要があります。

コントロールプレーンのアップグレード後に Fargate デプロイを再起動する

Fargate データプレーンノードをアップグレードするには、ワークロードを再デプロイする必要があります。`-o wide` オプションを使用してすべてのポッドを一覧表示することで、Fargate ノードで実行されているワークロードを特定できます。で始まるノード名`fargate-`は、クラスターに再デプロイする必要があります。

インプレースクラスターアップグレードの代わりに Blue/Green クラスターを評価する

一部のお客様は、ブルー/グリーンアップグレード戦略を優先します。これには利点がありますが、考慮すべき欠点も含まれます。

その他の利点には以下が含まれます。

- 複数の EKS バージョンを一度に変更可能 (例: 1.23 から 1.25)
- 古いクラスターに戻すことができます
- 新しいシステム (Terraform など) で管理できる新しいクラスターを作成します。

- ワークロードは個別に移行できます

次のような欠点があります。

- コンシューマーの更新を必要とする API エンドポイントと OIDC の変更 (kubectl や CI/CD など)
- 移行中に 2 つのクラスターを並行して実行する必要があるため、コストがかかり、リージョンの容量が制限される可能性があります
- ワークロードが相互に依存して移行する場合は、より多くの調整が必要です
- ロードバランサーと外部 DNS が複数のクラスターに簡単にまたがることができない

この戦略は可能ですが、インプレースアップグレードよりもコストがかかり、調整とワークロードの移行により多くの時間が必要です。状況によっては必要になる場合があり、慎重に計画する必要があります。

GitOps などの高度な自動化および宣言型システムでは、これが容易になる可能性があります。データをバックアップして新しいクラスターに移行するには、ステートフルワークロードに対して追加の予防措置を講じる必要があります。

詳細については、以下のブログ記事を参照してください。

- Kubernetes クラスターのアップグレード: Blue-Green デプロイ戦略
- ステートレス ArgoCD ワークロードの Blue/Green または Canary Amazon EKS クラスターの移行

Kubernetes プロジェクトで計画された主要な変更を追跡する — 先を考える

次のバージョンだけを見ないでください。リリースされた Kubernetes の新しいバージョンを確認し、主な変更を特定します。例えば、一部のアプリケーションは docker API を直接使用し、Docker (Dockershim とも呼ばれます) の Container Runtime Interface (CRI) のサポートは Kubernetes で削除されました`1.24`。このような変更の準備にはさらに時間がかかります。

アップグレード先のバージョンの文書化されたすべての変更を確認し、必要なアップグレード手順を書き留めます。また、Amazon EKS マネージドクラスターに固有の要件や手順にも注意してください。

- Kubernetes 変更ログ

機能の削除に関する具体的なガイダンス

1.25 での Dockershim の削除 - Docker ソケット (DDS) に Detector を使用する

EKS Optimized AMI for 1.25 には、Dockershim のサポートが含まれなくなりました。Docker ソケットをマウント するなど、Dockershim に依存関係がある場合は、ワーカーノードを 1.25 にアップグレードする前に、それらの依存関係を削除する必要があります。

1.25 にアップグレードする前に、Docker ソケットに依存しているインスタンスを見つけます。Detector for Docker Socket (DDS)、kubectl プラグインを使用することをお勧めします。

1.25 での PodSecurityPolicy の削除 - Pod セキュリティ標準または policy-as-code ソリューションへの移行

`PodSecurityPolicy` は Kubernetes 1.21 では廃止され、Kubernetes 1.25 では削除されました。クラスターで PodSecurityPolicy を使用している場合は、ワークロードの中断を避けるため、クラスターをバージョン 1.25 にアップグレードする前に、組み込みの Kubernetes Pod セキュリティ標準 (PSS) または policy-as-code ソリューションに移行する必要があります。

AWS は、EKS ドキュメントで詳細なよくある質問を発行しました。

Pod Security Standards (PSS) と Pod Security Admission (PSA) のベストプラクティスを確認してください。

Kubernetes ウェブサイトの PodSecurityPolicy 廃止ブログ記事を参照してください。

1.23 でのツリー内ストレージドライバーの廃止 - Container Storage Interface (CSI) ドライバーへの移行

Container Storage Interface (CSI) は、Kubernetes が既存のツリー内ストレージドライバーメカニズムを置き換えるのに役立つように設計されています。Amazon EBS コンテナストレージインターフェイス (CSI) の移行機能は、Amazon EKS `1.23` 以降のクラスターではデフォルトで有効になっています。バージョン `1.22`以前のクラスターでポッドを実行している場合は、サービスの中断を避けるため、クラスターをバージョン に更新する前に Amazon EBS CSI ドライバーをインストールする必要があります。 `1.23`

Amazon EBS CSI 移行に関するよくある質問を確認してください。

その他のリソース

ClowdHaus EKS アップグレードガイダンス

ClowdHaus EKS アップグレードガイダンスは、Amazon EKS クラスターのアップグレードに役立つ CLI です。クラスターを分析して、アップグレード前に修正すべき潜在的な問題がないか調べることができます。

GoNoGo

GoNoGo は、クラスターアドオンのアップグレードの信頼性を判断するためのアルファステージツールです。

コスト最適化のベストプラクティス

コスト最適化は、最低料金でビジネス成果を達成しています。このガイドのドキュメントに従うことで、Amazon EKS ワークロードを最適化できます。

一般的なガイドライン

クラウドには、マイクロサービスのコスト最適化を実現するのに役立つ一般的なガイドラインがいくつかあります。

- Amazon EKS で実行されているワークロードが、コンテナを実行するための特定のインフラストラクチャタイプから独立していることを確認します。これにより、最も安価なインフラストラクチャタイプで実行する際の柔軟性が高まります。EC2 で Amazon EKS を使用する場合、ワークロードの性質上、GPU やその他のインスタンスタイプを要求するなど、特定のタイプの EC2 インスタンスタイプを必要とするワークロードがある場合、例外が発生する可能性があります。
- 最適にプロファイリングされたコンテナインスタンスを選択する — Amazon Amazon CloudWatch EKS などのサービスや Kubernetes エコシステムで利用可能なサードパーティーツールを使用して、本番稼働環境または本番稼働前環境をプロファイリングし、CPU やメモリなどの重要なメトリクスをモニタリングします。これにより、適切な量のリソースを割り当て、リソースの浪費を回避できます。
- オンデマンド、スポット、Savings Plans など、EC2 で EKS を実行するために AWS で使用できるさまざまな購入オプションを活用します。

EKS コスト最適化のベストプラクティス

クラウドでのコスト最適化には、次の 3 つの一般的なベストプラクティス領域があります。

- 費用対効果の高いリソース (Auto Scaling、Down Scaling、ポリシー、購入オプション)
- 支出認識 (AWS およびサードパーティー製ツールを使用)
- 時間の経過に伴う最適化 (適切なサイズ設定)

他のガイダンスと同様に、トレードオフがあります。組織と協力して、このワークロードの優先順位と、どのベストプラクティスが最も重要かを理解してください。

このガイドの使い方

このガイドは、EKS クラスターとサポートするワークロードの実装と管理を担当する開発チームを対象としています。このガイドは、使用しやすいようにさまざまなベストプラクティス領域に整理されています。各トピックには、EKS クラスターのコスト最適化に使用する推奨事項、ツール、ベストプラクティスのリストがあります。トピックは、特定の順序で読み込む必要はありません。

AWS サービスと Kubernetes の主な機能

コストの最適化は、以下の AWS のサービスと機能でサポートされています。

- EC2 インスタンスタイプ、Savings Plan (およびリザーブドインスタンス) 、スポットインスタンスを異なる料金で利用できます。
- Auto Scaling と Kubernetes ネイティブ Auto Scaling ポリシー。予測可能なワークロードには、Savings Plan s(以前のリザーブドインスタンス) を検討してください。EBS や EFS などのマネージドデータストアを使用して、アプリケーションデータの伸縮性と耐久性を実現します。
- Billing and Cost Management コンソールダッシュボードと AWS Cost Explorer には、AWS の使用状況の概要が表示されます。AWS Organizations を使用して、請求の詳細を細かくします。いくつかのサードパーティー製ツールの詳細も共有されています。
- Amazon CloudWatch Container Metrics は、EKS クラスターによるリソースの使用状況に関するメトリクスを提供します。Kubernetes ダッシュボードに加えて、Kubernetes エコシステムには、無駄を減らすために使用できるツールがいくつかあります。

このガイドには、Amazon EKS クラスターのコスト最適化を向上させるために使用できる推奨事項のセットが含まれています。

フィードバック

このガイドは、より広範な EKS/Kubernetes コミュニティから直接的なフィードバックと提案を収集するために、GitHub でリリースされています。ガイドに含める必要があると思われるベストプラクティスがある場合は、GitHub リポジトリで問題を申請するか、PR を送信してください。新しい機能がサービスに追加されるとき、または新しいベストプラクティスが進化したときに、ガイドを定期的に更新する予定です。

コスト最適化フレームワーク

AWS Cloud Economics は、Amazon EKS などの最新のコンピューティングテクノロジーを採用することで、お客様の効率を高め、コストを削減するのに役立つ分野です。この規律では、4 つの柱で構成される「クラウド財務管理 (CFM) フレームワーク」と呼ばれる方法論に従うことを推奨しています。



「」の柱: 測定と説明責任

See の柱は、クラウド支出の測定、モニタリング、説明責任の作成方法を定義する基本的なアクティビティとテクノロジーのセットです。多くの場合、「オブザーバビリティ」、「計測」、または「テレメトリ」と呼ばれます。「オブザーバビリティ」インフラストラクチャの機能と制限により、何を最適化できるかが決まります。コストを明確に把握することは、どこから始めているかを知る必要があるため、コスト最適化の最初の重要なステップです。このタイプの可視性は、環境をさらに最適化するために実行する必要があるアクティビティのタイプもガイドします。

「柱を参照」のベストプラクティスの簡単な概要を次に示します。

- ワークロードのタグ付け戦略を定義して維持します。
  - インスタンスタグ付けを使用すると、EKS クラスターにタグ付けすることで、個々のクラスターコストを表示し、コストと使用状況レポートに割り当てることができます。
- Kubecost などのテクノロジーを使用して、EKS 使用状況のレポートとモニタリングを確立します。

  - リソースが適切にタグ付けされ、視覚化を使用することで、クラウドインテリジェンスダッシュボードを有効にし、コストを測定および見積もることができます。
- クラウドコストをアプリケーション、基幹業務 (LoBs)、収益ストリームに割り当てます。
- 効率/価値 KPIsに伝達します。たとえば、トランザクションあたりのコストを測定する「単位メトリクス」KPI を作成します。例えば、乗車共有サービスには「乗車あたりのコスト」の KPI がある場合があります。

この柱に関連する推奨されるテクノロジーとアクティビティの詳細については、このガイドの「コスト最適化 - オブザーバビリティ」セクションを参照してください。

Save の柱: コスト最適化

この柱は、「」の柱で開発されたテクノロジーと機能に基づいています。通常、以下のアクティビティがこの柱に該当します。

- 環境内の無駄を特定して排除します。
- コスト効率のためのアーキテクトと設計。
- オンデマンドインスタンスとスポットインスタンスなど、最適な購入オプションを選択します。
- サービスが進化するにつれて適応する: AWS のサービスが進化するにつれて、それらのサービスを効率的に使用する方法が変わる可能性があります。これらの変更を考慮して進んで適応してください。

これらのアクティビティは運用されているため、環境の特性に大きく依存します。コストの主な要因は何ですか？ さまざまな環境にはどのようなビジネス価値がありますか？ インスタンスファミリータイプなど、各環境に最適な購入オプションとインフラストラクチャの選択肢は何ですか？

以下は、EKS クラスターの最も一般的なコスト要因の優先順位付けされたリストです。

1. コンピューティングコスト： 複数のタイプのインスタンスファミリー、購入オプション、スケーラビリティと可用性のバランスをとるには、慎重に検討する必要があります。詳細については、このガイドの「コスト最適化 - コンピューティング」セクションの推奨事項を参照してください。
2. ネットワークコスト： EKS クラスターに 3 つの AZs を使用すると、AZ 間のトラフィックコストが増加する可能性があります。HA 要件とネットワークトラフィックのコストを抑える方法に関する推奨事項については、このガイドの「コスト最適化 - ネットワーキング」セクションを参照してください。

3. ストレージコスト： EKS クラスター内のワークロードのステートフル/ステートレスな性質、およびさまざまなストレージタイプの使用方法に応じて、ストレージをワークロードの一部として検討できます。EKS ストレージコストに関する考慮事項については、このガイドの「コスト最適化 - ストレージ」セクションを参照してください。

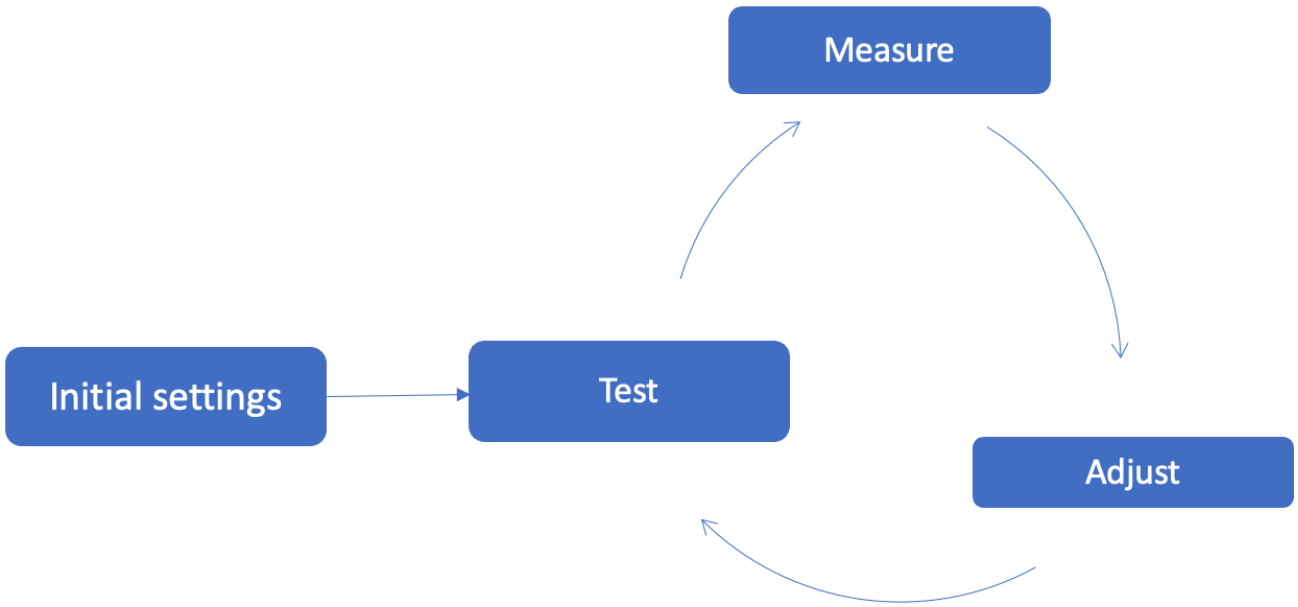
計画の柱: 計画と予測

See の柱の推奨事項が実装されると、クラスターは継続的に最適化されます。クラスターを効率的に運用する経験が積まれると、計画と予測のアクティビティは次のことに集中できます。

- クラウドのコストを動的に予算化および予測します。
- EKS コンテナサービスによって提供されるビジネス価値の定量化。
- EKS クラスターのコスト管理を IT 財務管理計画と統合します。

実行の柱

コスト最適化は継続的なプロセスであり、段階的な改善のフライホイールが含まれます。



EKS クラスターの最適化を組織のFinOps」の取り組みに統合するには、この種のアクティビティに対するエグゼクティブスポンサーシップを確保することが重要です。これにより、EKS クラスターのコストを共有し、EKS クラスターのコストガードレールを実装し、ツール、オートメーション、

アクティビティが組織のニーズに応じて進化するようにすることで、ステークホルダーの調整が可能になります。

リファレンス

- AWS Cloud Economics、Cloud Financial Management

支出認識

支出認識とは、誰が、どこで、何が EKS クラスターで支出を引き起こしているかを理解することです。このデータを正確に把握することで、支出に対する認識を高め、修復すべき領域を強調できます。

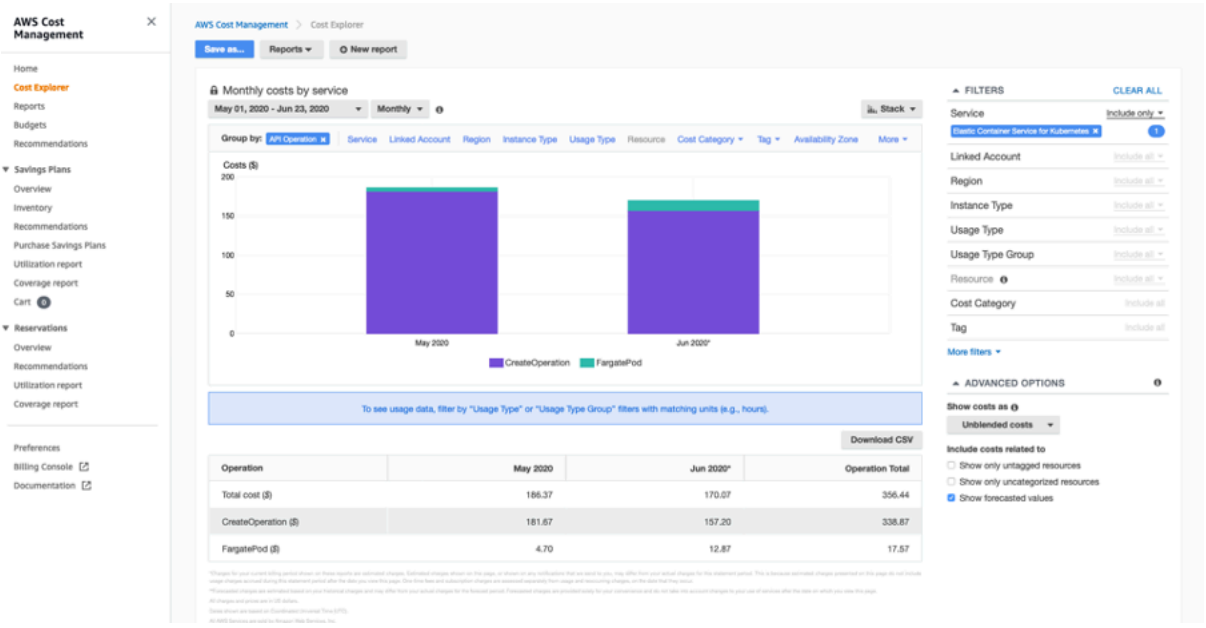
推奨事項

Cost Explorer を使用する

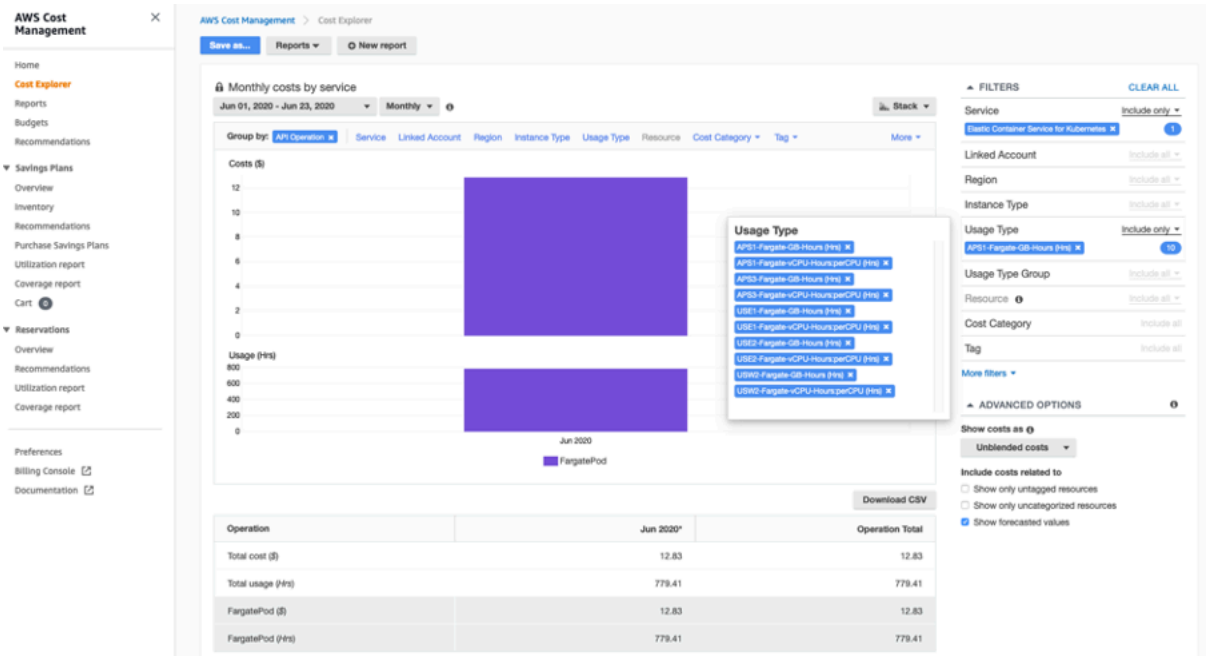
AWS Cost Explorer には、easy-to-useインターフェイスがあります。Cost Explorer で利用可能なフィルターを使用して、コストと使用状況のデータをさまざまなレベルで分析できます。

EKS コントロールプレーンと EKS Fargate のコスト

フィルターを使用すると、次の図に示すように、コントロールプレーンと Fargate Pod で EKS コストについて発生したコストをクエリできます。



フィルターを使用すると、EKS のリージョン全体で Fargate Pod に発生した総コストをクエリできます。これには、次の図に示すように、CPU あたりの vCPU 時間数と GB 時間の両方が含まれます。



リソースのタグ付け

Amazon EKS は、Amazon EKS クラスターへの AWS タグの追加をサポートしています。これにより、クラスターを管理するための EKS API へのアクセスを簡単に制御できます。EKS クラスターに追加されたタグは AWS EKS クラスターリソースに固有であり、EC2 インスタンスやロードバランサーなど、クラスターで使用される他の AWS リソースには伝達されません。現在、クラスターのタグ付けは、AWS API、コンソール、および SDKs。

AWS Fargate は、[コンテナ] に適切なサイズのコンピューティング能力をオンデマンドで提供するテクノロジーです。クラスターの Fargate でポッドをスケジューリングする前に、起動時に Fargate を使用するポッドを指定する Fargate プロファイルを少なくとも 1 つ定義する必要があります。

EKS クラスターへのタグの追加と一覧表示:

```
$ aws eks tag-resource --resource-arn arn:aws:eks:us-west-2:xxx:cluster/ekscluster1 --
tags team=devops,env=staging,bu=cio,costcenter=1234
$ aws eks list-tags-for-resource --resource-arn arn:aws:eks:us-west-2:xxx:cluster/
ekscluster1
{
    "tags": {
        "bu": "cio",
        "env": "staging",
```

```
        "costcenter": "1234",
        "team": "devops"
    }
}
```

AWS Cost Explorer でコスト配分タグをアクティブ化すると、AWS はコスト配分タグを使用してコスト配分レポートのリソースコストを整理し、AWS コストの分類と追跡を容易にします。

タグには Amazon EKS に関する意味論的な意味はなく、完全に文字列として解釈されます。例えば Amazon EKS クラスターに一連のタグを定義して、各クラスターの所有者とスタックレベルを追跡できます。

AWS Trusted Advisor を使用する

AWS Trusted Advisor は、コスト最適化、セキュリティ、耐障害性、パフォーマンス、サービス制限の 5 つのカテゴリにわたって、さまざまなベストプラクティスチェックと推奨事項を提供します。

コスト最適化のために、Trusted Advisor は未使用のリソースとアイドル状態のリソースを排除し、リザーブドキャパシティーへのコミットメントを推奨します。Amazon EKS に役立つ主なアクション項目は、使用率の低い EC2 インスタンス、関連付けられていない Elastic IP アドレス、アイドルロードバランサー、使用率の低い EBS ボリュームなどです。チェックの完全なリストは、https://aws.amazon.com/premiumsupport/technology/trusted-advisor/best-practice-checklist/ で入手できます。

Trusted Advisor はEC2 インスタンスと Fargate の Savings Plans とリザーブドインスタンスのレコメンデーションも提供します。これにより、割引料金と引き換えに一貫した使用量にコミットできます。

> **Note**
>
> Trusted Advisor からの推奨事項は一般的な推奨事項であり、EKS に固有のものではありません。

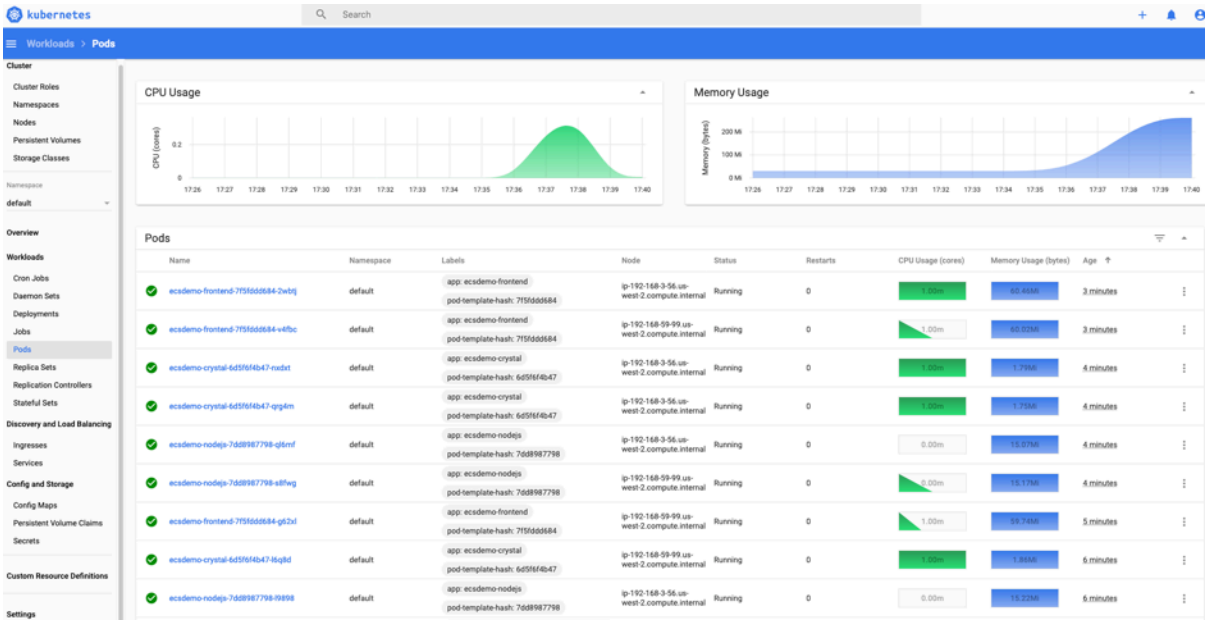
Kubernetes ダッシュボードを使用する

Kubernetes ダッシュボード

Kubernetes Dashboard は、Kubernetes クラスター用の汎用ウェブベースの UI で、クラスター、ノード、ポッドレベルでのリソース使用量など、Kubernetes クラスターに関する情報を提供しま

す。Amazon EKS クラスターへの Kubernetes ダッシュボードのデプロイについては、Amazon EKS ドキュメントで説明されています。

ダッシュボードには、各ノードとポッドのリソース使用状況の内訳と、ポッド、サービス、デプロイ、その他の Kubernetes オブジェクトに関する詳細なメタデータが表示されます。この統合された情報により、Kubernetes 環境を可視化できます。



kubectl 上部と describe コマンド

kubectl top および kubectl describe コマンドを使用したリソース使用状況メトリクスの表示。kubectl top は、クラスター全体のポッドまたはノード、または特定のポッドまたはノードの現在の CPU とメモリの使用状況を表示します。kubectl describe コマンドは、特定のノードまたはポッドに関する詳細情報を提供します。

```
$ kubectl top pods
$ kubectl top nodes
$ kubectl top pod pod-name --namespace mynamespace --containers
```

トップコマンドを使用すると、出力にはノードが使用している CPU (コア単位) とメモリ (MiB 単位) の合計量と、それらの数値が表すノードの割り当て可能な容量の割合が表示されます。その後、--containers フラグを追加して、ポッド内の次のレベルのコンテナレベルにドリルダウンできます。

```
$ kubectl describe node <node>
$ kubectl describe pod <pod>
```

kubectl describe は、各リソースリクエストまたは制限が表す使用可能な合計容量の割合を返します。

kubectl は、kubernetes ポッド、ノード、コンテナ間で CPU、メモリ、ストレージなどの重要なリソースの使用率と可用性を追跡し、説明します。この認識は、リソースの使用状況を理解し、コストを制御するのに役立ちます。

CloudWatch Container Insights を使用する

CloudWatch Container Insights を使用して、コンテナ化されたアプリケーションとマイクロサービスからメトリクスとログを収集、集約、要約します。Container Insights は、EC2 上の Amazon Elastic Kubernetes Service、および Amazon EC2 上の Kubernetes プラットフォームで使用できます。このメトリクスには、CPU、メモリ、ディスク、ネットワークなどのリソース使用率が含まれます。

インサイトのインストールについては、 ドキュメントを参照してください。

CloudWatch は、クラスター、ノード、ポッド、タスク、サービスレベルで CloudWatch メトリクスとして集計メトリクスを作成します。

次のクエリは、平均ノード CPU 使用率でソートされたノードのリストを示しています。

```
STATS avg(node_cpu_utilization) as avg_node_cpu_utilization by NodeName
| SORT avg_node_cpu_utilization DESC
```

コンテナ名別の CPU 使用率

```
stats pct(container_cpu_usage_total, 50) as CPUPercMedian by kubernetes.container_name
| filter Type="Container"
```

コンテナ名別のディスク使用量

```
stats floor(avg(container_filesystem_usage/1024)) as container_filesystem_usage_avg_kb
 by InstanceId, kubernetes.container_name, device
| filter Type="ContainerFS"
| sort container_filesystem_usage_avg_kb desc
```

Container Insights ドキュメントでは、さらに多くのサンプルクエリが提供されています。

この認識は、リソースの使用状況を理解し、コストを制御するのに役立ちます。

Kubecost を使用した支出の認識とガイダンス

kubecost などのサードパーティーツールを Amazon EKS にデプロイして、Kubernetes クラスターの実行コストを可視化することもできます。Kubecost を使用したコストの追跡については、この AWS ブログを参照してください。

Helm 3 を使用した kubecost のデプロイ:

```
$ curl -sSL https://raw.githubusercontent.com/helm/helm/master/scripts/get-helm-3 |
 bash
$ helm version --short
v3.2.1+gfe51cd1
$ helm repo add stable https://kubernetes-charts.storage.googleapis.com/
$ helm repo add stable https://kubernetes-charts.storage.googleapis.com/c^C
$ kubectl create namespace kubecost
namespace/kubecost created
$ helm repo add kubecost https://kubecost.github.io/cost-analyzer/
"kubecost" has been added to your repositories

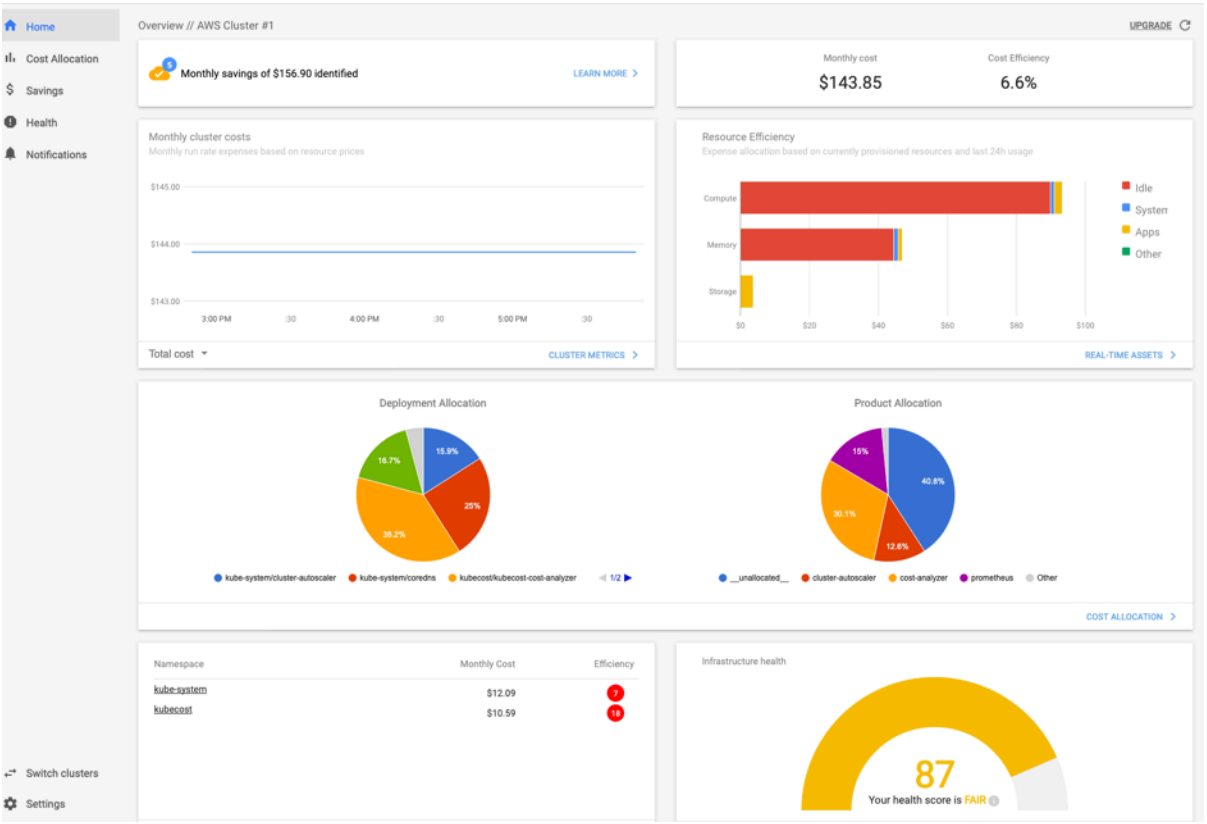
$ helm install kubecost kubecost/cost-analyzer --namespace kubecost --set
 kubecostToken="aGRoZEBqc2pzLmNvbQ==xm343yadf98"
NAME: kubecost
LAST DEPLOYED: Mon May 18 08:49:05 2020
NAMESPACE: kubecost
STATUS: deployed
REVISION: 1
TEST SUITE: None
NOTES:
----------------------------------------------------Kubecost has been successfully
 installed. When pods are Ready, you can enable port-forwarding with the following
 command:

    kubectl port-forward --namespace kubecost deployment/kubecost-cost-analyzer 9090

Next, navigate to http://localhost:9090 in a web browser.
$ kubectl port-forward --namespace kubecost deployment/kubecost-cost-analyzer 9090

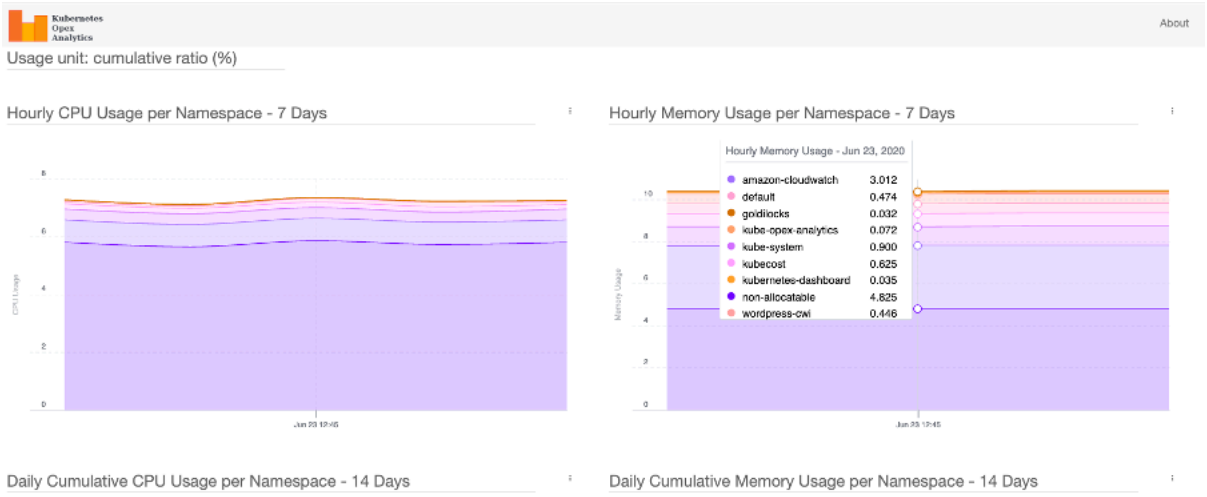
NOTE: If you are using Cloud 9 or have a need to forward it to a different port like
 8080, issue the following command
$ kubectl port-forward --namespace kubecost deployment/kubecost-cost-analyzer 8080:9090
```

Kubecost ダッシュボード -



Kubernetes コスト配分とキャパシティプランニング分析ツールを使用する

Kubernetes Opex Analytics は、組織が Kubernetes クラスターで消費されているリソースを追跡して過払いを防ぐのに役立つツールです。そのために、短期 (7 日)、中期 (14 日)、長期 (12 か月) の使用状況レポートを生成し、各プロジェクトが時間の経過とともに費やしているリソースの量に関する関連インサイトを示します。



ヨータスケール

Yotascale は、Kubernetes コストを正確に配分するのに役立ちます。Yotascale Kubernetes Cost Allocation 機能は、一般的な市場レートの見積もりではなく、リザーブドインスタンスの割引とスポットインスタンスの料金を含む実際のコストデータを利用して、Kubernetes のコストフットプリントの合計を通知します。

詳細については、ウェブサイトを参照してください。

Alcide Advisor

Alcide は AWS パートナーネットワーク (APN) アドバンストテクノロジーパートナーです。Alcide Advisor は、Amazon EKS クラスター、ノード、ポッドの設定を、セキュリティのベストプラクティスと内部ガイドラインに従って実行するように調整するのに役立ちます。Alcide Advisor は、Kubernetes 監査とコンプライアンスのためのエージェントレスサービスであり、本番環境に移行する前に開発段階を強化することで、摩擦がなく安全な DevSecOps フローを保証するように構築されています。

詳細については、このブログ記事を参照してください。

その他のツール

Kubernetes ガベージコレクション

Kubernetes ガベージコレクターの役割は、以前は所有者を持っていたが、もう所有者を持たない特定のオブジェクトを削除することです。

Fargate の数

Fargatecount は便利なツールです。AWS のお客様は、カスタム CloudWatch メトリクスを使用して、特定のアカウントの特定のリージョンで Fargate にデプロイされた EKS ポッドの総数を追跡できます。これにより、EKS クラスター全体で実行されているすべての Fargate ポッドを追跡できます。

Popeye - Kubernetes クラスターサニタイザー

ポップアイ - Kubernetes クラスターサニタイザーは、ライブ Kubernetes クラスターをスキャンし、デプロイされたリソースと設定に関する潜在的な問題をレポートするユーティリティです。ディスクに配置されているものではなく、デプロイされたものに基づいてクラスターをサニタイズします。クラスターをスキャンすることで、設定ミスを検出し、ベストプラクティスを確実に実施できます。

リソース

コスト最適化のベストプラクティスの詳細については、以下のリソースを参照してください。

ドキュメントとブログ

- Amazon EKS がタグ付けをサポート

ツール

- AWS Billing and Cost Management とは
- Amazon CloudWatch Container Insights
- Kubecost を使用してマルチテナント Amazon EKS クラスターのコストを追跡する方法
- Kubecost
- Kube Opsview
- Kubernetes Opex 分析

コンピューティングと自動スケーリング

開発者は、CPU やメモリなどのアプリケーションのリソース要件について見積もりを行いますが、継続的に調整しないと古くなり、コストが増加し、パフォーマンスと信頼性が低下する可能性があります。アプリケーションのリソース要件を継続的に調整することは、初めて正しく設定するよりも重要です。

以下に説明するベストプラクティスは、コストを最小限に抑え、組織が投資収益率を最大化しながら、ビジネス成果を達成するコスト対応ワークロードを構築して運用するのに役立ちます。クラスターのコンピューティングコストを最適化するための重要度の高い順序は次のとおりです。

1. 適切なサイズのワークロード
2. 未使用の容量を減らす
3. コンピューティングキャパシティタイプ (スポットなど) とアクセラレーター (GPUs) を最適化する

ワークロードの適切なサイズ設定

ほとんどの EKS クラスターでは、コストの大部分は、コンテナ化されたワークロードの実行に使用される EC2 インスタンスから発生します。ワークロードの要件を理解しないと、コンピューティングリソースのサイズを適正化することはできません。そのため、適切なリクエストと制限を使用し、必要に応じてそれらの設定を調整することが不可欠です。さらに、インスタンスサイズやストレージの選択などの依存関係はワークロードのパフォーマンスに影響を与える可能性があり、コストや信頼性に意図しないさまざまな影響を与える可能性があります。

リクエストは実際の使用率と一致する必要があります。コンテナのリクエストが高すぎると、未使用の容量が発生し、クラスターの合計コストの大きな要因になります。アプリケーションやサイドカーなど、ポッド内の各コンテナには、ポッドの集約制限が可能な限り正確になるように独自のリクエストと制限を設定する必要があります。

コンテナのリソースリクエストと制限を見積もる、Goldilocks、KRR、Kubecost などのツールを使用します。アプリケーションの性質、パフォーマンス/コスト要件、複雑さに応じて、スケーリングに最適なメトリクス、アプリケーションのパフォーマンスが低下する時点 (飽和点) 、それに応じてリクエストと制限を調整する方法を評価する必要があります。このトピックの詳細については、「アプリケーションの適切なサイズ設定」を参照してください。

Horizontal Pod Autoscaler (HPA) を使用して、実行するアプリケーションのレプリカの数を制御し、Vertical Pod Autoscaler (VPA) を使用してレプリカごとにアプリケーションに必要なリクエストの数を調整して制限し、Karpenter や Cluster Autoscaler などのノードオートスケーラーを使用してクラスター内のノードの合計数を継続的に調整することをお勧めします。Karpenter と Cluster Autoscaler を使用したコスト最適化の手法については、このドキュメントの後半セクションで説明します。

Vertical Pod Autoscaler は、コンテナに割り当てられたリクエストと制限を調整して、ワークロードが最適に実行されるようにできます。ポッドを自動的に変更して再起動しないように、VPA を監査モードで実行する必要があります。観測されたメトリクスに基づいて変更を提案します。本番稼働用ワークロードに影響する変更は、アプリケーションの信頼性とパフォーマンスに影響を与える可能性があるため、本番稼働用以外の環境で最初に変更を確認してテストする必要があります。

消費量の削減

コストを削減する最善の方法は、より少ないリソースをプロビジョニングすることです。そのための 1 つの方法は、現在の要件に基づいてワークロードを調整することです。コスト最適化の取り組みは、ワークロードが要件を定義し、動的にスケールするようにすることから始める必要があります。そのためには、アプリケーションからメトリクスを取得し、アプリケーションが動的に安全にスケー

ルアップおよびスケールダウンできるように、 PodDisruptionBudgets や Pod Readiness Gates などの設定を行う必要があります。Cluster Autoscaler と Karpenter の両方が PodDisruptionBudgets を尊重しているため、制限的な PodDisruptionBudgets は Cluster Autoscaler と Karpenter がノードをスケールダウンするのを防ぐことができることを考慮することが重要です。PodDisruptionBudget のminAvailable」値は、常にデプロイ内のポッドの数よりも小さく、2 つのポッドの間に適切なバッファを保持する必要があります。例えば、少なくとも 4 つのポッドを常に実行する必要がある 6 つのポッドのデプロイでは、PodDisruptionBidget のminAvailable」を 4 に設定します。これにより、Cluster Autoscaler と Karpenter は、ノードのスケールダウンイベント中に、使用率の低いノードからポッドを安全にドレインおよび削除できます。Cluster Autoscaler のよくある質問ドキュメントを参照してください。

Horizontal Pod Autoscaler は、アプリケーションのパフォーマンスと信頼性の要件を満たすために必要なレプリカの数を調整できる柔軟なワークロードオートスケーラーです。CPU、メモリ、キューの深さ、ポッドへの接続数などのカスタムメトリクスなど、さまざまなメトリクスに基づいてスケールアップ/ダウンするタイミングを定義する柔軟なモデルがあります。

Kubernetes メトリクスサーバーは、CPU やメモリの使用量などの組み込みメトリクスに応じてスケーリングを有効にしますが、Amazon CloudWatch や SQS キューの深さなどの他のメトリクスに基づいてスケーリングする場合は、KEDA などのイベント駆動型自動スケーリングプロジェクトを検討する必要があります。CloudWatch メトリクスで KEDA を使用する方法については、このブログ記事を参照してください。モニタリングおよびスケーリングの基準となるメトリクスが不明な場合は、重要なメトリクスのモニタリングに関するベストプラクティスを確認してください。

ワークロードの消費量を減らすと、クラスターの容量が過剰になり、適切な自動スケーリング設定により、ノードを自動的にスケールダウンして総支出を削減できます。コンピューティング容量を手動で最適化しないことをお勧めします。Kubernetes スケジューラとノードオートスケーラーは、このプロセスを処理するように設計されています。

未使用の容量を減らす

アプリケーションの正しいサイズを決定し、過剰なリクエストを減らしたら、プロビジョニングされたコンピューティング容量を減らし始めることができます。上記のセクションからワークロードを正しくサイズ設定する時間がかかっている場合は、これを動的に実行できます。AWS の Kubernetes で使用されるプライマリノードオートスケーラーは 2 つあります。

Karpenter と Cluster Autoscaler

Karpenter と Kubernetes Cluster Autoscaler の両方が、ポッドの作成または削除とコンピューティング要件の変更に応じて、クラスター内のノード数をスケーリングします。両方の主な目標は同じです

が、Karpenter はノード管理のプロビジョニングとプロビジョニング解除に異なるアプローチを採用しているため、コストを削減し、クラスター全体の使用量を最適化できます。

クラスターのサイズが大きくなり、ワークロードの種類が増えるにつれて、ノードグループとインスタンスを事前設定することが難しくなります。ワークロードリクエストと同様に、初期ベースラインを設定し、必要に応じて継続的に調整することが重要です。

Cluster Autoscaler を使用している場合、各 Auto Scaling グループ (ASG) の「最小」値と「最大」値を尊重し、「希望する」値のみを調整します。Cluster Autoscaler は「最小」数を超えて ASG をスケールダウンできないため、基盤となる ASG にこれらの値を設定するときは注意することが重要です。「希望する」カウントを通常の営業時間に必要なノード数として設定し、「最小」を営業時間外に必要なノード数として設定します。Cluster Autoscaler のよくある質問ドキュメントを参照してください。

クラスターオートスケーラー Priority Expander

Kubernetes Cluster Autoscaler は、ノードグループと呼ばれるノードのグループをスケールアップおよびスケールダウンすることで機能します。ワークロードを動的にスケーリングしていない場合、Cluster Autoscaler はコスト削減には役立ちません。Cluster Autoscaler では、ワークロードが消費されるように、クラスター管理者が事前にノードグループを作成する必要があります。ノードグループは、同じ「プロファイル」を持つインスタンスを使用するように設定する必要があります。つまり、CPU とメモリの量はほぼ同じです。

複数のノードグループを持つことができ、Cluster Autoscaler は優先度スケーリングレベルを設定するように設定でき、各ノードグループには異なるサイズのノードを含めることができます。ノードグループは異なる容量タイプを持つことができ、優先度エクスパンダーを使用して、安価なグループを最初にスケーリングできます。

以下は、 を使用してオンデマンドインスタンスを使用する前にリザーブドキャパシティ`ConfigMap`に優先順位を付けるクラスター設定のスニペットの例です。同じ手法を使用して、他のタイプよりも Graviton インスタンスまたはスポットインスタンスに優先順位を付けることができます。

```
apiVersion: eksctl.io/v1alpha5
kind: ClusterConfig
metadata:
  name: my-cluster
managedNodeGroups:
  - name: managed-ondemand
    minSize: 1
```

```
    maxSize: 7
    instanceType: m5.xlarge
  - name: managed-reserved
    minSize: 2
    maxSize: 10
    instanceType: c5.2xlarge
```

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: cluster-autoscaler-priority-expander
  namespace: kube-system
data:
  priorities: |-
    10:
      - .*ondemand.*
    50:
      - .*reserved.*
```

ノードグループを使用すると、基盤となるコンピューティングリソースがデフォルトで期待どおりのことを行うのに役立ちます。たとえば、ノードを AZs に分散できますが、すべてのワークロードが同じ要件や期待値を持つわけではないため、アプリケーションに要件を明示的に宣言させることをお勧めします。Cluster Autoscaler の詳細については、「ベストプラクティス」セクションを参照してください。

デスケジューラ

Cluster Autoscaler は、スケジュールする必要がある新しいポッド、またはノードが十分に活用されていないことに基づいて、クラスターからノード容量を追加および削除できます。ノードにスケジュールされた後、ポッド配置の完全リストビューは表示されません。Cluster Autoscaler を使用している場合は、クラスターの容量の浪費を避けるために、Kubernetes デスケジューラも確認する必要があります。

クラスターに 10 個のノードがあり、各ノードが 60% 使用されている場合、クラスター内のプロビジョニングされた容量の 40% を使用していません。Cluster Autoscaler では、ノードあたりの使用率しきい値を 60% に設定できますが、使用率が 60% を下回った後にのみ、1 つのノードをスケールダウンしようとします。

デスケジューラを使用すると、ポッドがスケジュールされた後、またはノードがクラスターに追加された後に、クラスターの容量と使用率を確認できます。クラスターの合計容量を指定されたしきい値を超えて維持しようとします。また、ノードテイントまたはクラスターに参加する新しいノードに基

づいてポッドを削除して、ポッドが最適なコンピューティング環境で実行されていることを確認することもできます。デスケジューラは削除されたポッドの置き換えをスケジュールしませんが、そのためにデフォルトのスケジューラに依存することに注意してください。

Karpenter の統合

Karpenter はノード管理に「グループレス」アプローチを採用しています。このアプローチは、さまざまなワークロードタイプに対してより柔軟であり、クラスター管理者の事前設定が少なくて済みます。Karpenter は、グループを事前定義し、ワークロードのニーズに応じて各グループをスケーリングする代わりに、プロビジョナーとノードテンプレートを使用して、作成できる EC2 インスタンスのタイプと、作成時にインスタンスに関する設定を広く定義します。

ビンパッキングは、より多くのワークロードをより少ない最適なサイズのインスタンスにパッキングすることで、インスタンスのリソースをさらに活用する方法です。これにより、ワークロードが使用するリソースのみをプロビジョニングすることでコンピューティングコストを削減できますが、トレードオフがあります。特に大規模なスケーリングイベント中に容量をクラスターに追加する必要があるため、新しいワークロードの開始に時間がかかる場合があります。ビンパッキングを設定するときは、コストの最適化、パフォーマンス、可用性のバランスを考慮してください。

Karpenter は、インスタンスリソースの使用率を向上させ、コンピューティングコストを削減するために、継続的にモニタリングとバイナリパックを行うことができます。Karpenter は、ワークロードによりコスト効率の高いワーカーノードを選択することもできます。これは、プロビジョナーで「統合」フラグを true にすることで実現できます (以下のサンプルコードスニペット)。以下の例は、統合を有効にするプロビジョナーの例を示しています。このガイドを書いている時点で、Karpenter は実行中のスポットインスタンスをより安価なスポットインスタンスに置き換えません。Karpenter の統合の詳細については、このブログを参照してください。

```
apiVersion: karpenter.sh/v1
kind: Provisioner
metadata:
  name: enable-binpacking
spec:
  consolidation:
    enabled: true
```

チェックポイントなしで長時間実行されるバッチジョブなど、中断できないワークロードの場合は、ポッドに注釈を付けることを検討してくださいdo-not-evict。ポッドを エビクションからオプトアウトすることで、このポッドを含むノードを自発的に削除しないように Karpenter に伝えます。ただし、ノードのドレイン中にdo-not-evictポッドがノードに追加された場合、残りのポッドは引

き続き削除されますが、そのポッドは削除されるまで終了をブロックします。いずれの場合も、ノードは、ノードで追加の作業がスケジュールされないように接続されます。注釈の設定方法の例を以下に示します。

```
apiVersion: v1
kind: Pod
metadata:
  name: label-demo
  labels:
    environment: production
  annotations: +
    "karpenter.sh/do-not-evict": "true"
spec:
  containers:

* name: nginx
image: nginx
ports:
 ** containerPort: 80
```

Cluster Autoscaler パラメータを調整して使用率の低いノードを削除する

ノード使用率は、リクエストされたリソースの合計を容量で割ったものとして定義されます。デフォルトでは、 scale-down-utilization-thresholdは 50% に設定されています。このパラメータは、 および とともに使用できます。これによりscale-down-unneeded-time、スケールダウンの対象となる前にノードが不要になる時間を決定できます。デフォルトは 10 分です。スケールダウンされたノードでまだ実行されているポッドは、kube-scheduler によって他のノードでスケジュールされます。これらの設定を調整すると、使用率の低いノードを削除できますが、クラスターを強制的に早期にスケールダウンしないように、まずこれらの値をテストすることが重要です。

削除にコストがかかるポッドが Cluster Autoscaler によって認識されるラベルで保護されるようにすることで、スケールダウンの発生を防ぐことができます。これを行うには、削除にコストがかかるポッドに注釈 があることを確認しますcluster-autoscaler.kubernetes.io/safe-to-evict=false。以下は、注釈を設定するための yaml の例です。

```
apiVersion: v1
kind: Pod
metadata:
  name: label-demo
  labels:
    environment: production
```

```
  annotations: +
    "cluster-autoscaler.kubernetes.io/safe-to-evict": "false"
spec:
  containers:

* name: nginx
image: nginx
ports:
 ** containerPort: 80
```

Cluster Autoscaler と Karpenter を使用してノードにタグを付ける

AWS リソースタグは、リソースを整理し、AWS のコストを詳細レベルで追跡するために使用されます。コスト追跡のために Kubernetes ラベルと直接相関することはありません。Kubernetes リソースラベル付けから開始し、Kubecost などのツールを使用して、ポッドや名前空間の Kubernetes ラベルに基づいてインフラストラクチャのコストレポートを取得することをお勧めします。

ワーカーノードには、AWS Cost Explorer で請求情報を表示するためのタグが必要です。Cluster Autoscaler では、起動テンプレートを使用してマネージド型ノードグループ内のワーカーノードにタグを付けます。セルフマネージド型ノードグループの場合は、EC2 自動スケーリンググループを使用してインスタンスにタグを付けます。Karpenter によってプロビジョニングされたインスタンスの場合は、ノードテンプレートの spec.tags を使用してタグ付けします。

マルチテナントクラスター

異なるチームによって共有されているクラスターを操作する場合、同じノードで実行されている他のワークロードを可視化できない場合があります。リソースリクエストは、CPU 共有などの「ノイズの多い近隣」の懸念を分離するのに役立ちますが、ディスク I/O の枯渇などのすべてのリソース境界を分離するとは限りません。ワークロードのすべての使用可能なリソースを分離または制限できるわけではありません。共有リソースを他のワークロードよりも高い割合で消費するワークロードは、ノードのテイントと許容度によって分離する必要があります。このようなワークロードのもう 1 つの高度な手法は、コンテナの共有 CPU ではなく排他的 CPU を確保する CPU ピン留めです。

ノードレベルでワークロードを分離するとコストがかかる場合がありますが、リザーブドインスタンス、Graviton プロセッサhttps://aws.amazon.com/ec2/graviton/、またはスポットを使用して、BestEffort ジョブをスケジュールしたり、追加の節約を活用したりできます。

共有クラスターには、IP の枯渇、Kubernetes サービスの制限、API スケーリングリクエストなどのクラスターレベルのリソース制約もあります。スケーラビリティのベストプラクティスガイドを確認して、クラスターがこれらの制限を回避する必要があります。

リソースは、名前空間または Karpenter プロビジョナーレベルで分離できます。Resource Quotas は、名前空間内のリソースワークロードが消費できるリソース数の制限を設定する方法を提供します。これは適切な初期ガードレールですが、ワークロードのスケーリングを人為的に制限しないように継続的に評価する必要があります。

Karpenter プロビジョナーは、クラスター内の一部の使用可能なリソース (CPU、GPU など) に制限を設定できますが、適切なプロビジョナーを使用するようにテナントアプリケーションを設定する必要があります。これにより、1 つのプロビジョナーがクラスター内に作成するノードが多すぎるのを防ぐことができますが、制限が低すぎることがないように継続的に評価し、ワークロードのスケーリングを防ぐ必要があります。

スケジュールされた Auto Scaling

週末やオフ時間にクラスターをスケールダウンする必要がある場合があります。これは、使用されていないときにゼロにスケールダウンするテストクラスターと非本番クラスターに特に関連します。cluster-turndown などのソリューションは、cron スケジュールに基づいてレプリカをゼロにスケールダウンできます。次の AWS ブログで説明されているように、Karpenter でこれを取得することもできます。

コンピューティングキャパシティタイプを最適化する

クラスター内のコンピューティングの合計容量を最適化し、ビンパッキングを利用したら、クラスターでプロビジョニングしたコンピューティングのタイプと、それらのリソースに対する支払い方法を確認する必要があります。AWS には、コンピューティングのコストを削減できる Compute Savings プランがあり、これを次のキャパシティタイプに分類します。

- Spot
- 積立プラン
- オンデマンド
- Fargate

キャパシティタイプごとに管理オーバーヘッド、可用性、長期コミットメントのトレードオフが異なるため、どちらが環境に適しているかを決定する必要があります。1 つのキャパシティタイプに依存する環境はなく、1 つのクラスターに複数の実行タイプを組み合わせて、特定のワークロード要件とコストを最適化できます。

Spot

スポットキャパシティタイプは、アベイラビリティーゾーンの予備のキャパシティから EC2 インスタンスをプロビジョニングします。スポットは最大 90% の割引を提供しますが、これらのインスタンスは他の場所で必要になると中断される可能性があります。さらに、新しいスポットインスタンスをプロビジョニングする容量が常にあるとは限らず、既存のスポットインスタンスは 2 分間の中断通知で再利用できます。アプリケーションの起動またはシャットダウンプロセスが長い場合、スポットインスタンスが最適なオプションではない可能性があります。

スポットコンピューティングでは、スポット容量が利用できない可能性を減らすために、さまざまなインスタンスタイプを使用する必要があります。インスタンスの中断は、ノードを安全にシャットダウンするために処理する必要があります。Karpenter または Managed Node Group の一部でプロビジョニングされたノードは、インスタンスの中断通知を自動的にサポートします。セルフマネージド型ノードを使用している場合は、スポットインスタンスを適切にシャットダウンするために、ノード終了ハンドラーを個別に実行する必要があります。

スポットインスタンスとオンデマンドインスタンスを 1 つのクラスターでバランスさせることができます。Karpenter を使用すると、加重プロビジョナーを作成して、さまざまな容量タイプのバランスを実現できます。Cluster Autoscaler を使用すると、スポットインスタンスとオンデマンドインスタンスまたはリザーブドインスタンスを含む混合ノードグループを作成できます。

以下は、Karpenter を使用してオンデマンドインスタンスよりもスポットインスタンスを優先する例です。プロビジョナーを作成するときは、スポット、オンデマンド、またはその両方を指定できます (以下を参照)。両方を指定し、ポッドがスポットとオンデマンドのどちらを使用する必要があるかを明示的に指定しない場合、Karpenter は price-capacity-optimization 配分戦略 を使用してノードをプロビジョニングするときにスポットを優先します。

```
apiVersion: karpenter.sh/v1
kind: Provisioner
metadata:
  name: spot-prioritized
spec:
  requirements:
    - key: "karpenter.sh/capacity-type"
      operator: In
        values: ["spot", "on-demand"]
```

Savings Plans、リザーブドインスタンス、AWS EDP

コンピューティング削減プランを使用すると、コンピューティング支出を削減できます。Savings Plans は、1 年または 3 年のコンピューティング使用量のコミットメントに対して割引価格を提供します。使用量は EKS クラスター内の EC2 インスタンスに適用できますが、Lambda や Fargate などのコンピューティング使用量にも適用されます。Savings Plans を使用すると、コストを削減し、コミットメント期間中に任意の EC2 インスタンスタイプを選択することができます。

コンピューティング節約プランは、使用するインスタンスタイプ、ファミリー、またはリージョンに関するコミットメントを必要とせずに、EC2 コストを最大 66% 削減できます。節約は、使用するインスタンスに自動的に適用されます。

EC2 Instance Savings Plans は、コンピューティングを最大 72% 削減し、C ファミリーのインスタンスなど、特定のリージョンと EC2 ファミリーでの使用を約束します。リージョン内の任意の AZ に使用状況をシフトしたり、c5 や c6 などの任意の世代のインスタンスファミリーを使用したり、ファミリー内の任意のサイズのインスタンスを使用したりできます。割引は、Savings Plans 基準に一致するアカウント内のすべてのインスタンスに自動的に適用されます。

リザーブドインスタンスは EC2 Instance Savings Plan sに似ていますが、アベイラビリティーゾーンまたはリージョンの容量を保証し、オンデマンドインスタンスよりもコストを最大 72% 削減します。必要な予約容量を計算したら、予約期間 (1 年または 3 年) を選択できます。割引は、アカウントでこれらの EC2 インスタンスを実行すると自動的に適用されます。

お客様は、AWS とのエンタープライズ契約に登録することもできます。エンタープライズ契約では、ニーズに最適な契約を調整できます。お客様は、AWS EDP (エンタープライズ割引プログラム) に基づく料金で割引を受けることができます。エンタープライズ契約の詳細については、AWS 販売担当者にお問い合わせください。

オンデマンド

オンデマンド EC2 インスタンスには、Savings Plans と比較して、スポットと比較して中断のない可用性という利点があり、長期のコミットメントはありません。クラスターのコストを削減する場合は、オンデマンド EC2 インスタンスの使用量を減らす必要があります。

ワークロード要件を最適化したら、クラスターの最小容量と最大容量を計算できるはずです。この数は時間の経過とともに変化する可能性がありますが、ほとんど減少しません。最小限のものすべてに Savings Plan sを使用することを検討し、アプリケーションの可用性に影響を与えない容量を見つけます。継続的に使用されない、または可用性のために必要になるその他のものは、オンデマンドで使用できます。

このセクションで説明したように、使用量を減らす最善の方法は、より少ないリソースを消費し、プロビジョニングしたリソースを可能な限り最大限活用することです。Cluster Autoscaler を使用すると、 `scale-down-utilization-threshold`設定で使用率の低いノードを削除できます。Karpenter では、統合を有効にすることをお勧めします。

ワークロードで使用できる EC2 インスタンスタイプを手動で識別するには、ec2-instance-selector を使用する必要があります。ec2-instance-selector は、各リージョンで使用できるインスタンスと EKS と互換性のあるインスタンスを表示できます。x86 プロセスアーキテクチャ、4 Gb のメモリ、2 つの vCPUsus-east-1 リージョンで利用可能なインスタンスの使用例。

```
ec2-instance-selector --memory 4 --vcpus 2 --cpu-architecture x86_64 \
  -r us-east-1 --service eks
c5.large
c5a.large
c5ad.large
c5d.large
c6a.large
c6i.large
t2.medium
t3.medium
t3a.medium
```

非本番環境では、夜間や週末などの未使用の時間帯にクラスターを自動的にスケールダウンできます。kubecost プロジェクトの cluster-turndown は、設定されたスケジュールに基づいてクラスターを自動的にスケールダウンできるコントローラーの例です。

Fargate コンピューティング

Fargate コンピューティングは、EKS クラスター用のフルマネージド型のコンピューティングオプションです。Kubernetes クラスターのノードごとに 1 つのポッドをスケジュールすることで、ポッドを分離できます。これにより、コンピューティングノードをワークロードの CPU および RAM 要件に合わせてサイズ設定し、クラスター内のワークロード使用率を厳密に制御できます。

Fargate は、0.5 GB のメモリを搭載した .25 vCPU から、120 GB のメモリを搭載した 16 vCPU までワークロードをスケールできます。使用可能なポッドサイズのバリエーションには制限があり、ワークロードが Fargate 設定にどのように最適に適合するかを理解する必要があります。たとえば、ワークロードに 0.5 GB のメモリを持つ 1 つの vCPU が必要な場合、最小の Fargate ポッドは 2 GB のメモリを持つ 1 つの vCPU になります。

Fargate には EC2 インスタンスやオペレーティングシステム管理がないなど多くの利点がありますが、デプロイされたすべてのポッドがクラスター内の個別のノードとして分離されているため、従

来の EC2 インスタンスよりも多くのコンピューティング容量が必要になる場合があります。そのためには、Kubelet、ロギングエージェント、通常ノードにデプロイする DaemonSets などに対して、さらに重複する必要があります。DaemonSets は Fargate ではサポートされていないため、ポッド「サイドカー」に変換してアプリケーションと一緒に実行する必要があります。

Fargate は、ワークロードの境界がバースト不可能またはワークロード間で共有できないノードであるため、プロビジョニングよりもビンパッキングや CPU のメリットを享受できません。Fargate は、それ自体にコストがかかる EC2 インスタンス管理時間を節約しますが、CPU とメモリのコストは他の EC2 容量タイプよりも高価になる可能性があります。Fargate ポッドは、コンピューティング削減プランを活用してオンデマンドコストを削減できます。

コンピューティング使用量の最適化

コンピューティングインフラストラクチャのコストを削減するもう 1 つの方法は、ワークロードに対してより効率的なコンピューティングを使用することです。これは、x86 よりも最大 20% 安価でエネルギー効率が 60% 高い Graviton プロセッサなどのパフォーマンスの高い汎用コンピューティングやGPUs や FPGAs などのワークロード固有のアクセラレーターから発生する可能性があります。アームアーキテクチャで実行できるコンテナを構築し、ワークロードに適したアクセラレーターでノードを設定する必要があります。

EKS には、混合アーキテクチャ (amd64 や arm64 など) でクラスターを実行する機能があり、コンテナが複数のアーキテクチャ用にコンパイルされている場合は、プロビジョナーで両方のアーキテクチャを許可することで、Karpenter で Graviton プロセッサを活用できます。ただし、一貫したパフォーマンスを維持するには、各ワークロードを 1 つのコンピューティングアーキテクチャに保持し、使用可能な追加の容量がない場合にのみ異なるアーキテクチャを使用することをお勧めします。

プロビジョナーは複数のアーキテクチャで設定でき、ワークロードはワークロード仕様で特定のアーキテクチャをリクエストすることもできます。

```
apiVersion: karpenter.sh/v1
kind: Provisioner
metadata:
  name: default
spec:
  requirements:
  - key: "kubernetes.io/arch"
    operator: In
    values: ["arm64", "amd64"]
```

Cluster Autoscaler では、Graviton インスタンス用のノードグループを作成し、新しい容量を利用するためにワークロードにノードの許容範囲を設定する必要があります。

GPUsと FPGAsはワークロードのパフォーマンスを大幅に向上させることができますが、アクセラレーターを使用するようにワークロードを最適化する必要があります。機械学習と人工知能の多くのワークロードタイプはコンピューティングに GPUs を使用でき、インスタンスはリソースリクエストを使用してクラスターに追加し、ワークロードにマウントできます。

```
spec:
  template:
    spec:
    - containers:
      ...
      resources:
          limits:
            nvidia.com/gpu: "1"
```

一部の GPU ハードウェアは複数のワークロード間で共有できるため、1 つの GPU をプロビジョニングして使用できます。ワークロード GPU 共有を設定する方法については、仮想 GPU デバイスプラグインを参照してください。以下のブログも参照できます。

- GPU sharing on Amazon EKS with NVIDIA time-slicing and accelerated EC2 instances
- Maximizing GPU utilization with NVIDIA's Multi-Instance GPU (MIG) on Amazon EKS: Running more pods per GPU for enhanced performance

コスト最適化 - ネットワーク

高可用性 (HA) のシステム設計は、耐障害性と耐障害性を実現するためにベストプラクティスです。実際には、これはワークロードと基盤となるインフラストラクチャを特定の AWS リージョンの複数のアベイラビリティーゾーン (AZs) に分散することを意味します。Amazon EKS 環境にこれらの特性があることを確認することで、システムの全体的な信頼性が向上します。これと組み合わせて、EKS 環境は、さまざまなコンストラクト (VPCs など) 、コンポーネント (ELBs など) 、統合 (ECR やその他のコンテナレジストリなど) で構成される可能性があります。

高可用性システムとその他のユースケース固有のコンポーネントの組み合わせは、データの転送と処理方法に大きな役割を果たします。これにより、データ転送と処理によって発生するコストにも影響します。

以下に詳述するプラクティスは、さまざまなドメインやユースケースでコスト効率を実現するために、EKS 環境を設計および最適化するのに役立ちます。

ポッド間の通信

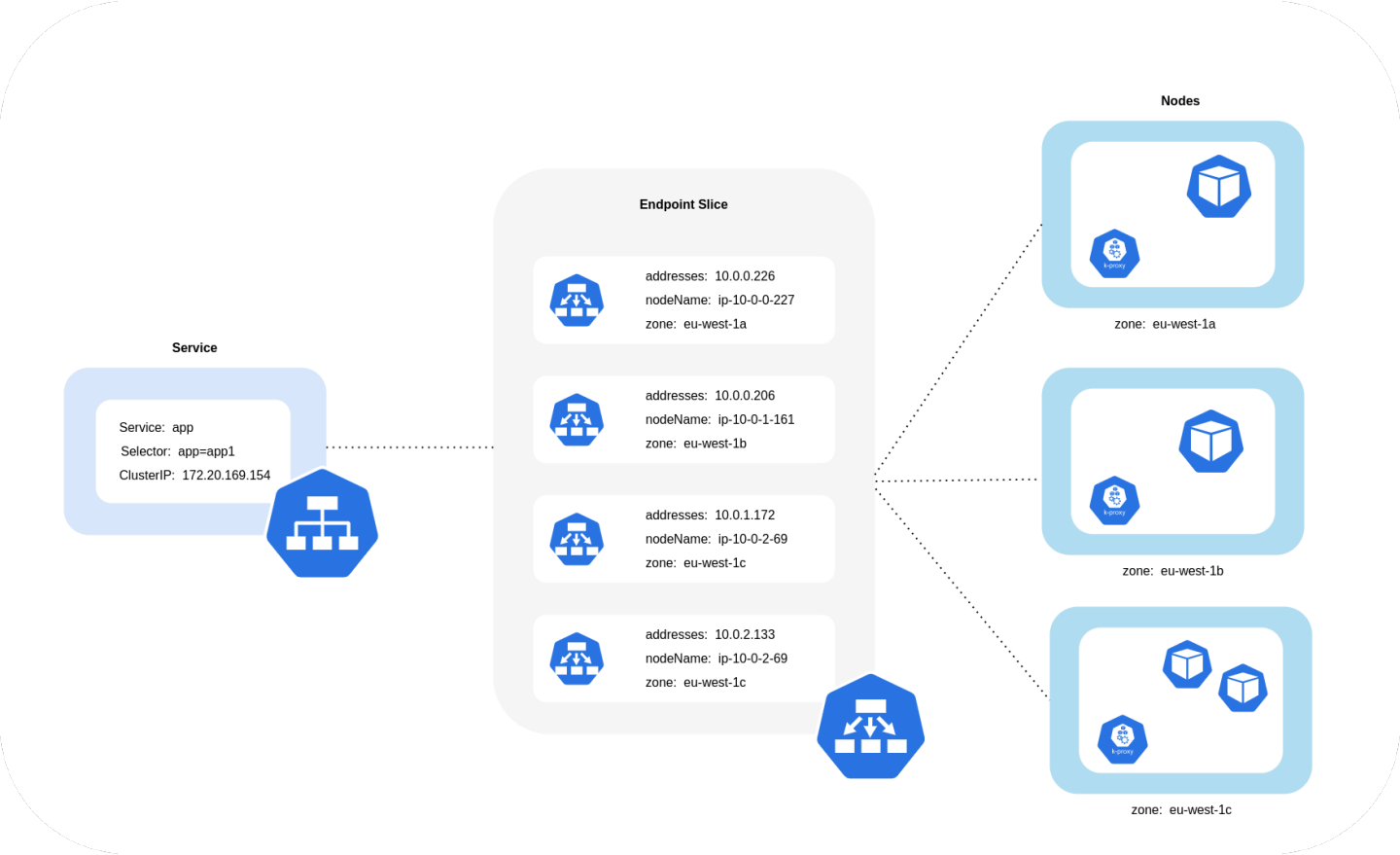
設定によっては、Pod 間のネットワーク通信とデータ転送が Amazon EKS ワークロードの実行にかかる全体的なコストに大きな影響を与える可能性があります。このセクションでは、高可用性 (HA) アーキテクチャ、アプリケーションのパフォーマンス、耐障害性を考慮しながら、ポッド間通信に関連するコストを削減するためのさまざまな概念とアプローチについて説明します。

アベイラビリティーゾーンへのトラフィックの制限

Kubernetes プロジェクトは、障害ドメイン間のワークロード分散やトポロジ対応ボリュームプロビジョナーなどの機能を有効にするために、ノードに割り当てられた kubernetes.io/hostname、topology.kubernetes.io/region、topology.kubernetes.io/zone などのラベルを含むトポロジ対応コンストラクトの開発を早期に開始しました。Kubernetes 1.17 を卒業したことで、ラベルは Pod 間通信のトポロジ対応ルーティング機能を有効にするためにも活用されました。

以下は、コストを削減し、レイテンシーを最小限に抑えるために、EKS クラスター内のポッド間のクロス AZ トラフィックの量を制御する方法に関するいくつかの戦略です。

クラスター内のポッド間のクロス AZ トラフィックの量 (バイト単位で転送されるデータ量など) を詳細に把握したい場合は、この記事を参照してください。



上記の図に示すように、サービスは Pod 宛てのトラフィックを受信する安定したネットワーク抽象化レイヤーです。サービスが作成されると、複数の EndpointSlices が作成されます。各 EndpointSlice には、ポッドアドレスのサブセットと、それらが実行されているノード、および追加のトポロジ情報を含むエンドポイントのリストがあります。すべてのノードで実行されているデーモンセットである Amazon VPC CNI を使用する場合、kube-proxy はネットワークルールを維持し、Pod 通信とサービス検出を有効にします (代替 eBPF ベースの CNIs は kube-proxy を使用しない場合がありますが、同等の動作を提供します)。内部ルーティングのロールは果たされますが、作成された EndpointSlices の消費量に基づいて行われます。

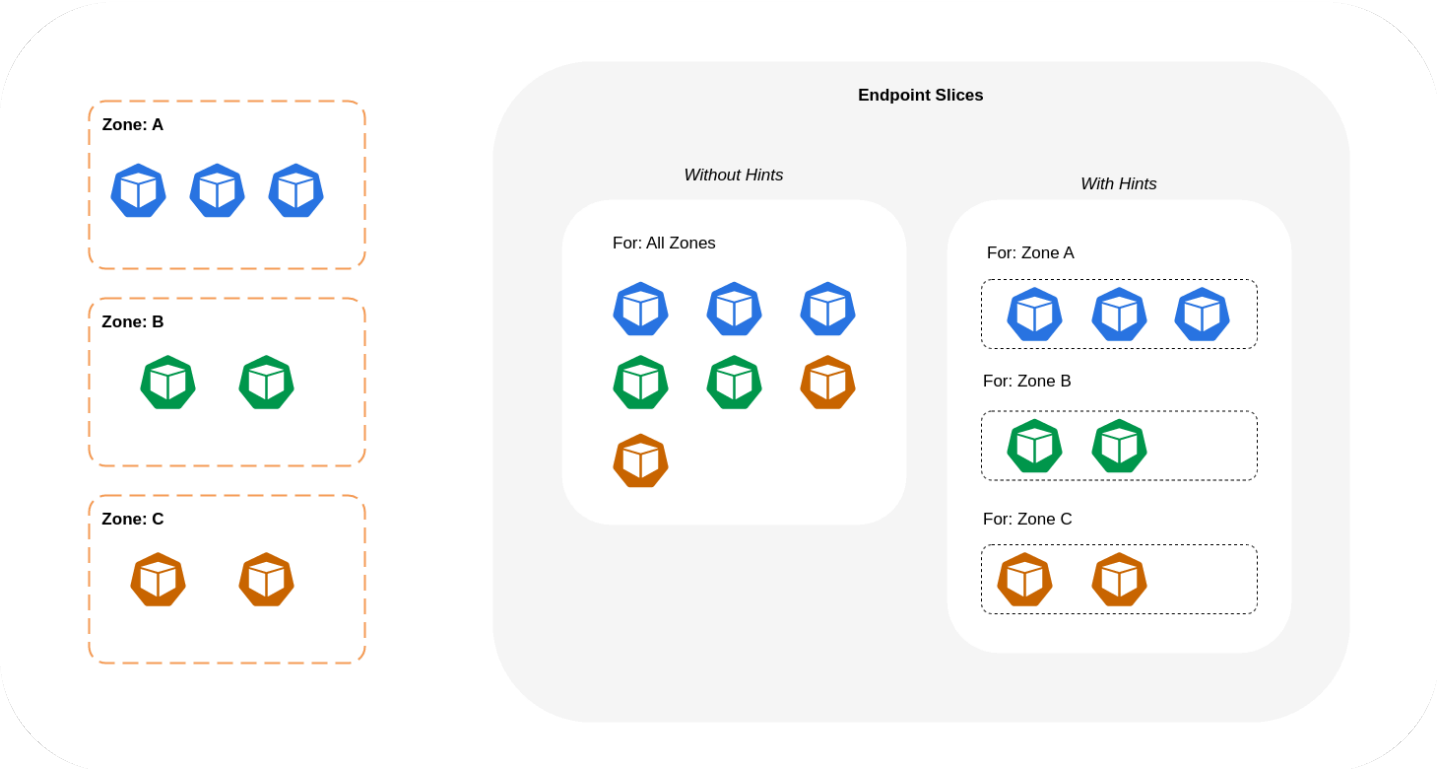
EKS では、kube-proxy は、ノードまたは AZ 配置に関係なく、クラスター内のすべてのポッドへのトラフィック分散に iptables NAT ルール (または IPVS、NFTables) を主に使用します。このデフォルトのディストリビューションは、AZ 間のトラフィックルーティングにつながる可能性があり、大規模なデプロイでは、機密性の高いアプリケーションのレイテンシーが増加し、AZ 間のデータ転送料金が発生する可能性があります。

トポロジ対応ルーティングの使用 (以前はトポロジ対応ヒントと呼ばれていました)

トポロジ対応ルーティングを有効にして Kubernetes サービスに実装すると、EndpointSlice コントローラーはクラスターが分散されているさまざまなゾーンにエンドポイントを比例して割り当てま

す。これらのエンドポイントごとに、EndpointSlice コントローラーはゾーンのヒントも設定します。ヒントは、エンドポイントがトラフィックを処理するゾーンを記述します。 kube-proxyは、適用されるヒントに基づいて、ゾーンからエンドポイントにトラフィックをルーティングします。

以下の図は、ヒントを含む EndpointSlices が、ゾーンのオリジンポイントに基づいてどの宛先に移動するかkube-proxyがわかるように編成されている方法を示しています。ヒントがない場合、このような割り当てや組織は存在せず、トラフィックは送信元に関係なく異なるゾーンの宛先にプロキシされます。



場合によっては、EndpointSlice コントローラーは別のゾーンにヒントを適用することがあります。つまり、エンドポイントは別のゾーンから発信されるトラフィックを処理する可能性があります。これは、異なるゾーンのエンドポイント間でトラフィックの均等な分散を維持しようとするためです。

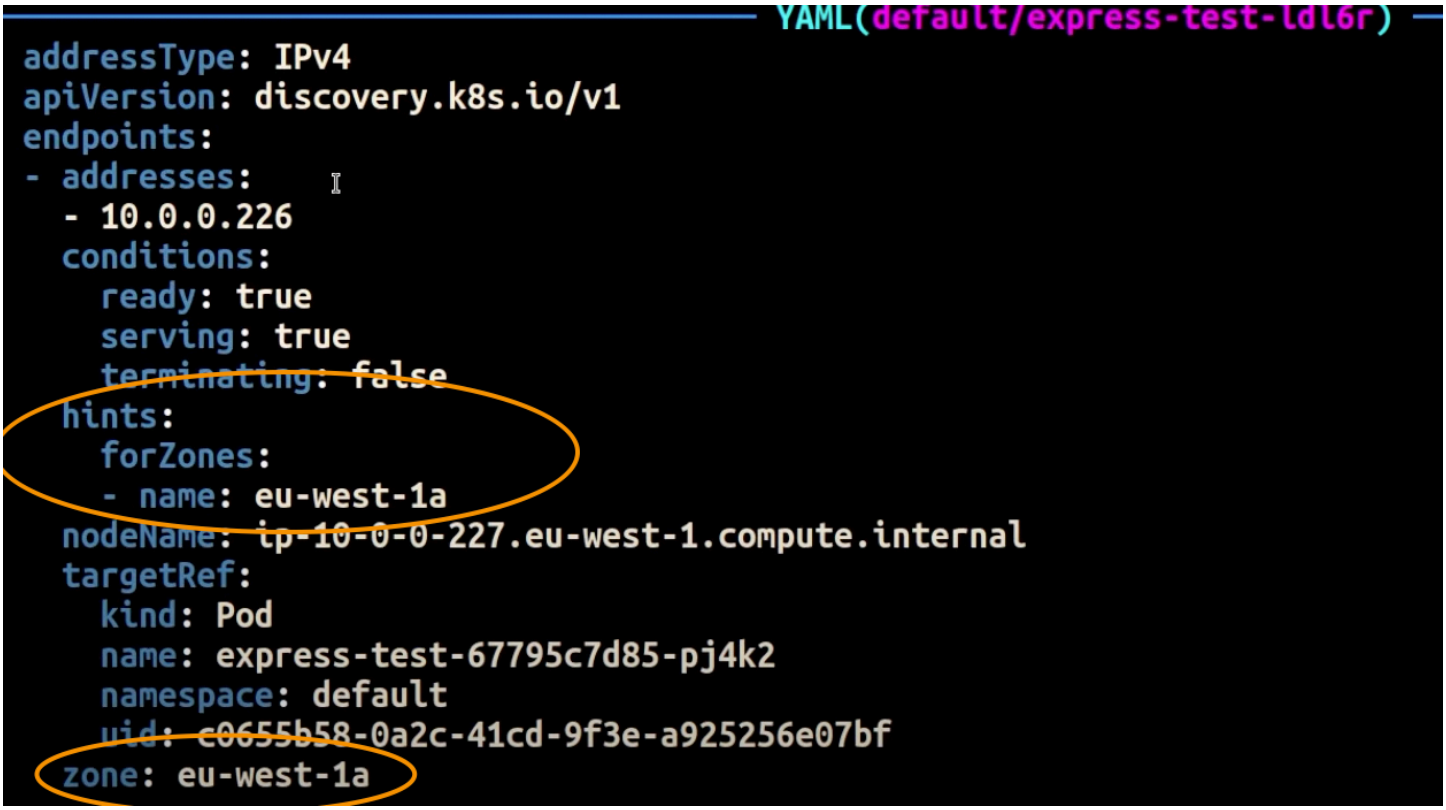
以下は、サービスのトポロジ対応ルーティングを有効にする方法に関するコードスニペットです。

```
apiVersion: v1
kind: Service
metadata:
  name: orders-service
  namespace: ecommerce
  annotations:
    service.kubernetes.io/topology-mode: Auto
spec:
```

```
  selector:
    app: orders
  type: ClusterIP
  ports:

* protocol: TCP
port: 3003
targetPort: 3003
```

以下のスクリーンショットは、EndpointSlice コントローラーが AZ で実行されているポッドレプリカのエンドポイントにヒントを正常に適用した結果を示していますeu-west-1a。



> **Note**
>
> トポロジ対応ルーティングはまだベータ版であることに注意してください。この機能は、クラスタートポロジ全体で均等に分散されたワークロードでより予測的に動作します。コントローラーは、ゾーン間でエンドポイントを比例的に割り当てますが、ゾーン内のノードリソースのバランスが不均衡すぎて過剰なオーバーロードを回避すると、ヒントの割り当てをスキップする可能性があります。したがって、ポッドトポロジのスプレッド制約などのアプリケーションの可用性を高めるスケジューリング制約と組み合わせて使用することを強くお勧めします。また、Amazon EC2 スポットインスタンスを使用する場合など、ゾーン間で容

量が変動する場合は、比例分布を計算するときに中断や置換がリアルタイムで検出されないため、ヒントが割り当てられない場合があることに注意してください。

トラフィック分散の使用

Kubernetes 1.30 で導入され、1.33 で一般公開されたトラフィックディストリビューションは、同じゾーンのトラフィック設定のトポロジ対応ルーティングよりも簡単な代替手段を提供します。トポロジ対応ルーティングは、エンドポイントの過負荷を回避するためにトラフィックルーティングにインテリジェントなアプローチを使用しようとしますが、予期しない動作が発生しました。トラフィック分散は、代わりに予測可能性を優先します。PreferClose オプションは、EndpointSlice Controller によって設定されたゾーンヒントに基づいて、最初に同じゾーンエンドポイントにトラフィックをルーティングするルールを作成するように kube-proxy に指示します。同じゾーンのエンドポイントが利用できない場合、サービスのクラスターエンドポイント間でトラフィックを分散させることにフォールバックします。この機能は、トポロジ対応ルーティングが提供する負荷の均等分散を試みるのではなく、近接性の最適化のトレードオフを受け入れるワークロード向けに設計されています。

以下は、サービスのトラフィック分散を有効にする方法に関するコードスニペットです。

```
apiVersion: v1
kind: Service
metadata:
  name: orders-service
  namespace: ecommerce
spec:
  trafficDistribution: PreferClose
  selector:
    app: orders
  type: ClusterIP
  ports:

* protocol: TCP
port: 3003
targetPort: 3003
```

トラフィック分散を有効にすると、共通の課題が発生します。ほとんどのトラフィックが同じゾーンから発信された場合、単一の AZ 内のエンドポイントが過負荷になる可能性があります。このオーバーロードは重大な問題を引き起こす可能性があります。

- マルチ AZ 配置を管理する 1 つの Horizontal Pod Autoscaler (HPA) は、異なる AZs。ただし、このアクションは、影響を受けるゾーンで増加した負荷に効果的に対処できません。
- この状況は、リソースの非効率性につながる可能性があります。Karpenter などのクラスターオートスケーラーがさまざまな AZs、影響を受けていない AZs に追加のノードをプロビジョニングし、不要なリソース割り当てが発生する可能性があります。

この課題を克服するには:

- ゾーンごとに個別のデプロイを作成し、独自の HPAs を使用して互いに独立してスケーリングします。
- トポロジのスプレッド制約を活用して、クラスター全体のワークロード分散を確保します。これにより、トラフィックの多いゾーンでのエンドポイントの過負荷を防ぐことができます。

Autoscaler の使用: 特定の AZ にノードをプロビジョニングする

複数の AZ にまたがる高可用性環境でワークロードを実行することを強くお勧めAZs。これにより、特に AZ に問題がある場合に、アプリケーションの信頼性が向上します。ネットワーク関連のコストを削減するために信頼性を犠牲にする場合は、ノードを 1 つの AZ に制限できます。

同じ AZ ですべての Pod を実行するには、同じ AZ でワーカーノードをプロビジョニングするか、同じ AZ で実行されているワーカーノードで Pod をスケジュールします。単一の AZ 内にノードをプロビジョニングするには、Cluster Autoscaler (CA) で同じ AZ に属するサブネットを持つノードグループを定義します。Karpenter では、 を使用して`topology.kubernetes.io/zone`、ワーカーノードを作成する AZ を指定します。たとえば、次の Karpenter プロビジョナースニペットは、us-west-2a AZ のノードをプロビジョニングします。

Karpenter

```
apiVersion: karpenter.sh/v1
kind: Provisioner
metadata:
name: single-az
spec:
  requirements:

* key: "topology.kubernetes.io/zone"`
operator: In
values: ["us-west-2a"]
```

クラスターオートスケーラー (CA)

```
apiVersion: eksctl.io/v1alpha5
kind: ClusterConfig
metadata:
  name: my-ca-cluster
  region: us-east-1
  version: "1.21"
availabilityZones:

* us-east-1a
managedNodeGroups:
* name: managed-nodes
labels:
  role: managed-nodes
instanceType: t3.medium
minSize: 1
maxSize: 10
desiredCapacity: 1
...
```

ポッド割り当てとノードアフィニティの使用

または、複数の AZs で実行されているワーカーノードがある場合、各ノードには AZ の値 (us-west-2a や us-west-2b など) を含む topology.kubernetes.io/zone というラベルが付けられます。nodeSelector または を使用してnodeAffinity、ポッドを単一の AZ のノードにスケジュールできます。たとえば、次のマニフェストファイルは、AZ us-west-2a で実行されているノード内で Pod をスケジュールします。

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
    env: test
spec:
  nodeSelector:
    topology.kubernetes.io/zone: us-west-2a
  containers:

* name: nginx
image: nginx
```

```
imagePullPolicy: IfNotPresent
```

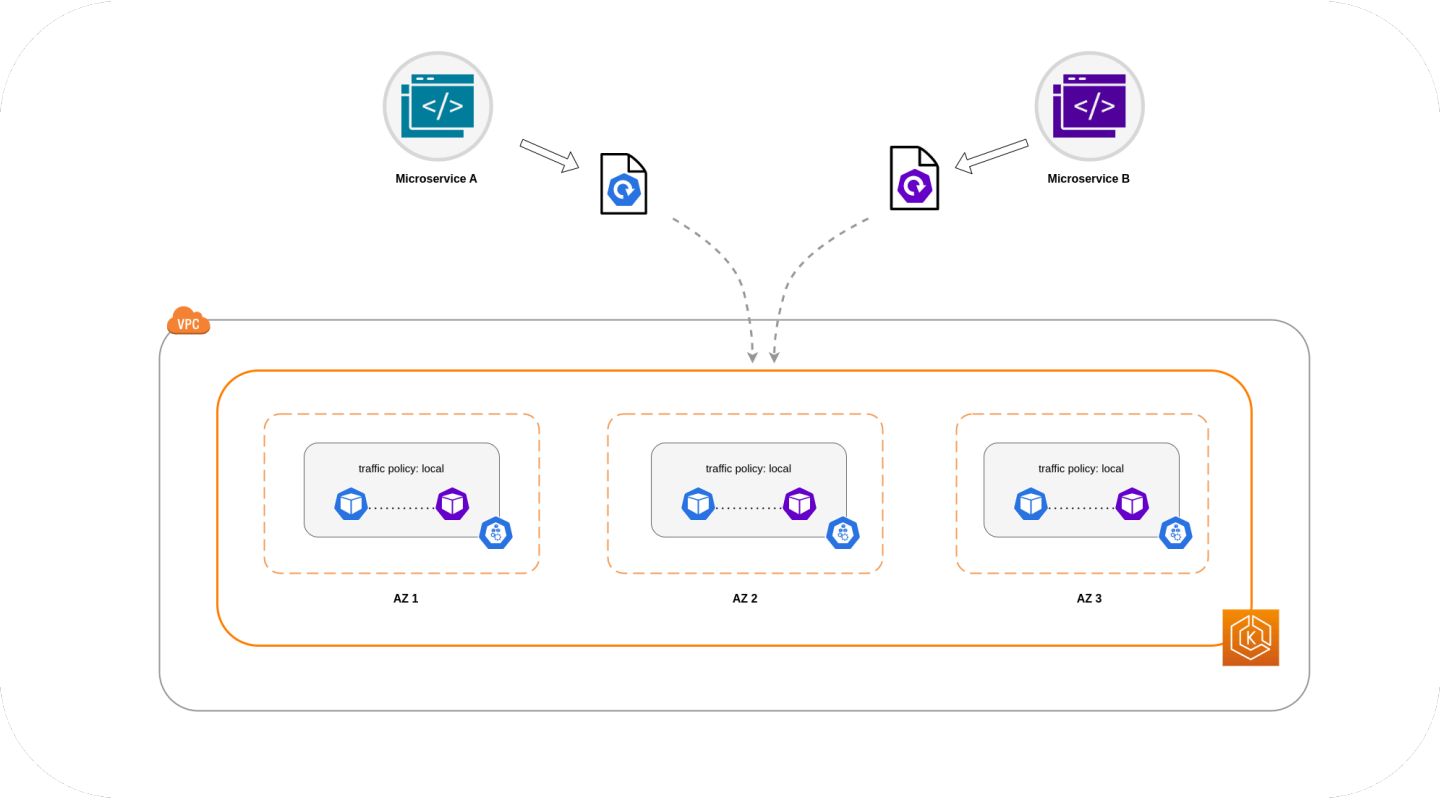
ノードへのトラフィックの制限

ゾーンレベルでのトラフィックの制限では不十分な場合があります。コストを削減するだけでなく、頻繁に相互通信を行う特定のアプリケーション間のネットワークレイテンシーを減らすという要件も追加される場合があります。最適なネットワークパフォーマンスを実現し、コストを削減するには、特定のノードへのトラフィックを制限する方法が必要です。例えば、マイクロサービス A は、高可用性 (HA) セットアップであっても、ノード 1 のマイクロサービス B と常に通信する必要があります。ノード 1 のマイクロサービス A がノード 2 のマイクロサービス B と通信すると、特にノード 2 が別の AZ にある場合、この種のアプリケーションに必要なパフォーマンスに悪影響を及ぼす可能性があります。

サービス内部トラフィックポリシーの使用

Pod ネットワークトラフィックをノードに制限するには、サービス内部トラフィックポリシー を使用できます。デフォルトでは、ワークロードの サービスに送信されるトラフィックは、生成されたさまざまなエンドポイントにランダムに分散されます。したがって、HA アーキテクチャでは、マイクロサービス A からのトラフィックは、異なる AZs 間で特定のノード上のマイクロサービス B のレプリカに移動する可能性があります。ただし、サービスの内部トラフィックポリシーを に設定するとLocal、トラフィックはトラフィックの送信元のノード上のエンドポイントに制限されます。このポリシーは、ノードローカルエンドポイントの排他的使用を指示します。つまり、そのワークロードのネットワークトラフィック関連のコストは、ディストリビューションがクラスター全体であった場合よりも低くなります。また、レイテンシーが低くなり、アプリケーションのパフォーマンスが向上します。

> **Note**
>
> この機能は、Kubernetes のトポロジ対応ルーティングと組み合わせることはできません。



以下は、サービスの内部トラフィックポリシーを設定する方法に関するコードスニペットです。

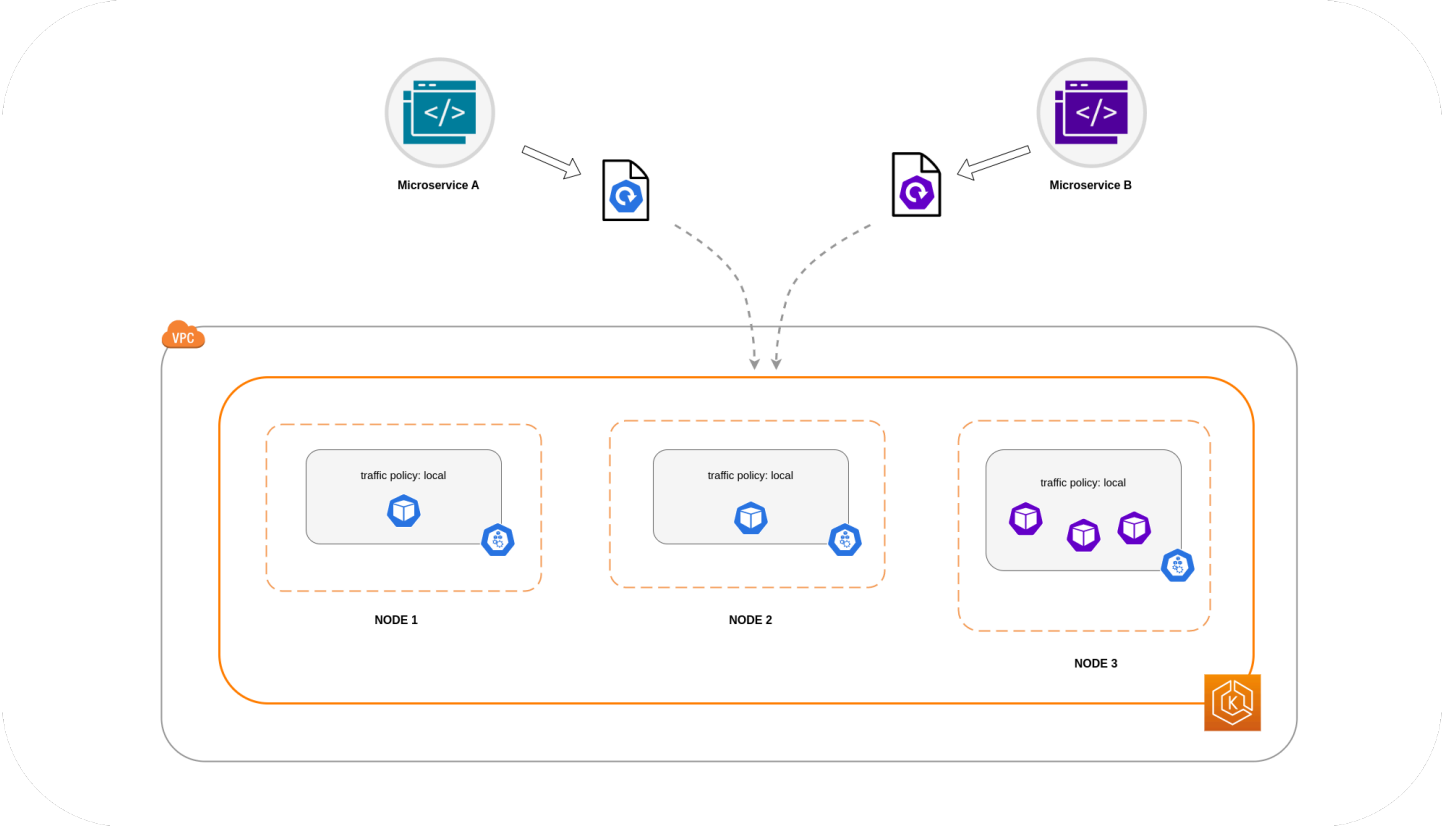
```
apiVersion: v1
kind: Service
metadata:
  name: orders-service
  namespace: ecommerce
spec:
  selector:
    app: orders
  type: ClusterIP
  ports:

* protocol: TCP
port: 3003
targetPort: 3003
  internalTrafficPolicy: Local
```

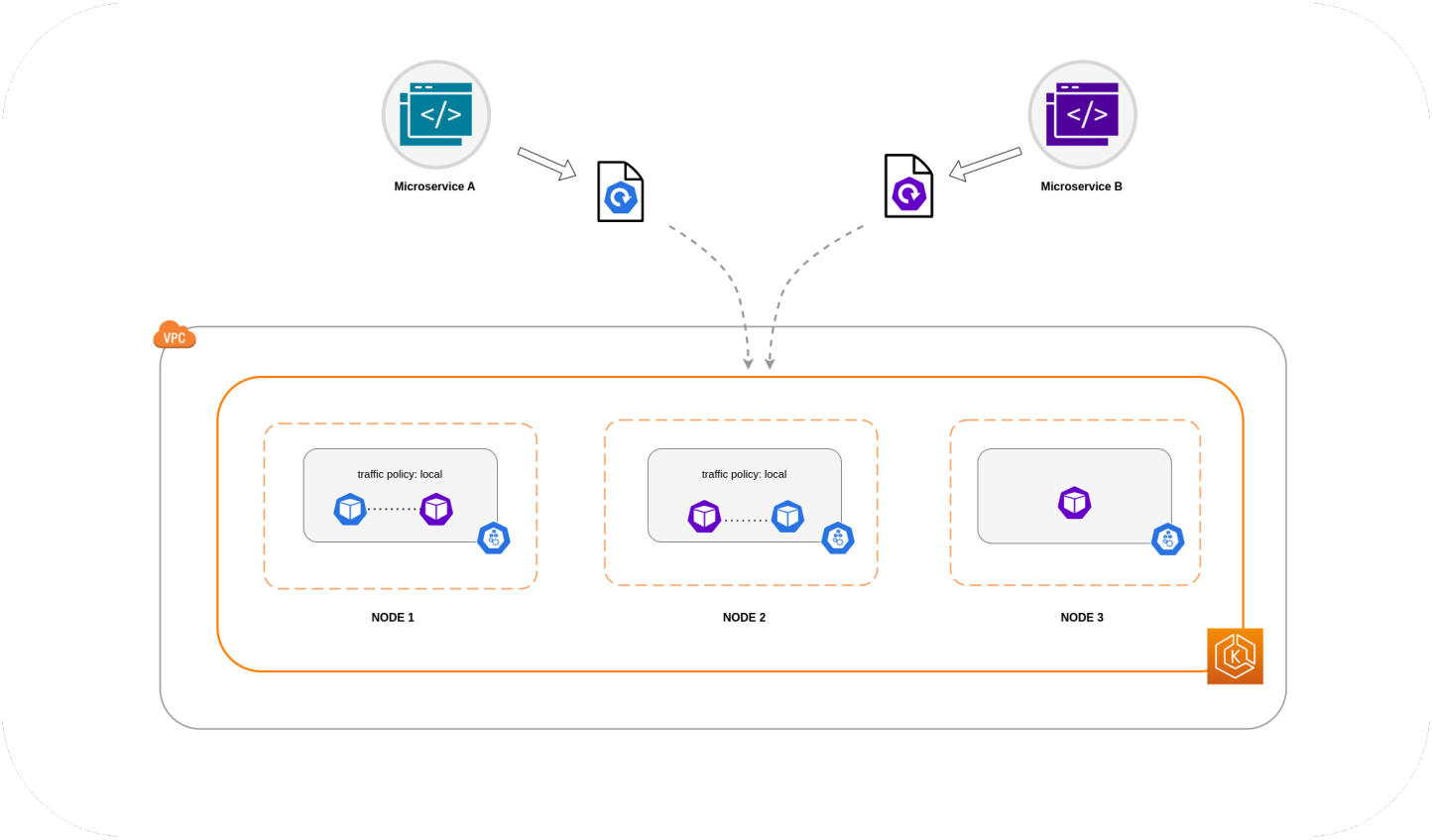
トラフィックの減少によるアプリケーションからの予期しない動作を回避するには、次のアプローチを検討する必要があります。

- 通信ポッドごとに十分なレプリカを実行する
- トポロジ分散制約を使用してポッドを比較的均等に分散させる
- 通信ポッドのコロケーションにポッドアフィニティルールを使用する

この例では、マイクロサービス A のレプリカが 2 つ、マイクロサービス B のレプリカが 3 つあります。マイクロサービス A のレプリカがノード 1 と 2 の間に分散されていて、マイクロサービス B のレプリカがノード 3 にある場合、Local内部トラフィックポリシーのため通信できません。使用可能なノードローカルエンドポイントがない場合、トラフィックはドロップされます。



Microservice B にノード 1 と 2 の 3 つのレプリカのうち 2 つがある場合、ピアアプリケーション間の通信が行われます。ただし、マイクロサービス B の分離されたレプリカには、通信するピアレプリカはありません。



> **Note**
>
> シナリオによっては、上記の図に示されているような分離されたレプリカが目的 (外部受信トラフィックからのリクエストの処理など) を果たしても、懸念の原因にならない場合があります。

トポロジのスプレッド制約によるサービス内部トラフィックポリシーの使用

内部トラフィックポリシーをトポロジの分散制約と組み合わせて使用すると、異なるノードでマイクロサービスを通信するための適切な数のレプリカを確保できます。

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: express-test
spec:
  replicas: 6
  selector:
    matchLabels:
```

```
      app: express-test
  template:
    metadata:
      labels:
        app: express-test
        tier: backend
    spec:
      topologySpreadConstraints:
      - maxSkew: 1
        topologyKey: "topology.kubernetes.io/zone"
        whenUnsatisfiable: ScheduleAnyway
        labelSelector:
          matchLabels:
            app: express-test
```

ポッドアフィニティルールでのサービス内部トラフィックポリシーの使用

もう 1 つの方法は、サービス内部トラフィックポリシーを使用するときに Pod アフィニティルールを使用することです。Pod アフィニティを使用すると、頻繁に通信するため、スケジューラに影響を与えて特定の Pod をコロケーションできます。特定の Pod に厳密なスケジューリング制約 (`requiredDuringSchedulingIgnoredDuringExecution`) を適用することで、スケジューラがノードに Pod を配置するときの Pod コロケーションの結果が向上します。

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: graphql
  namespace: ecommerce
  labels:
    app.kubernetes.io/version: "0.1.6"
    ...
    spec:
      serviceAccountName: graphql-service-account
      affinity:
        podAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
          - labelSelector:
              matchExpressions:
              - key: app
                operator: In
                values:
                - orders
```

```
            topologyKey: "kubernetes.io/hostname"
```

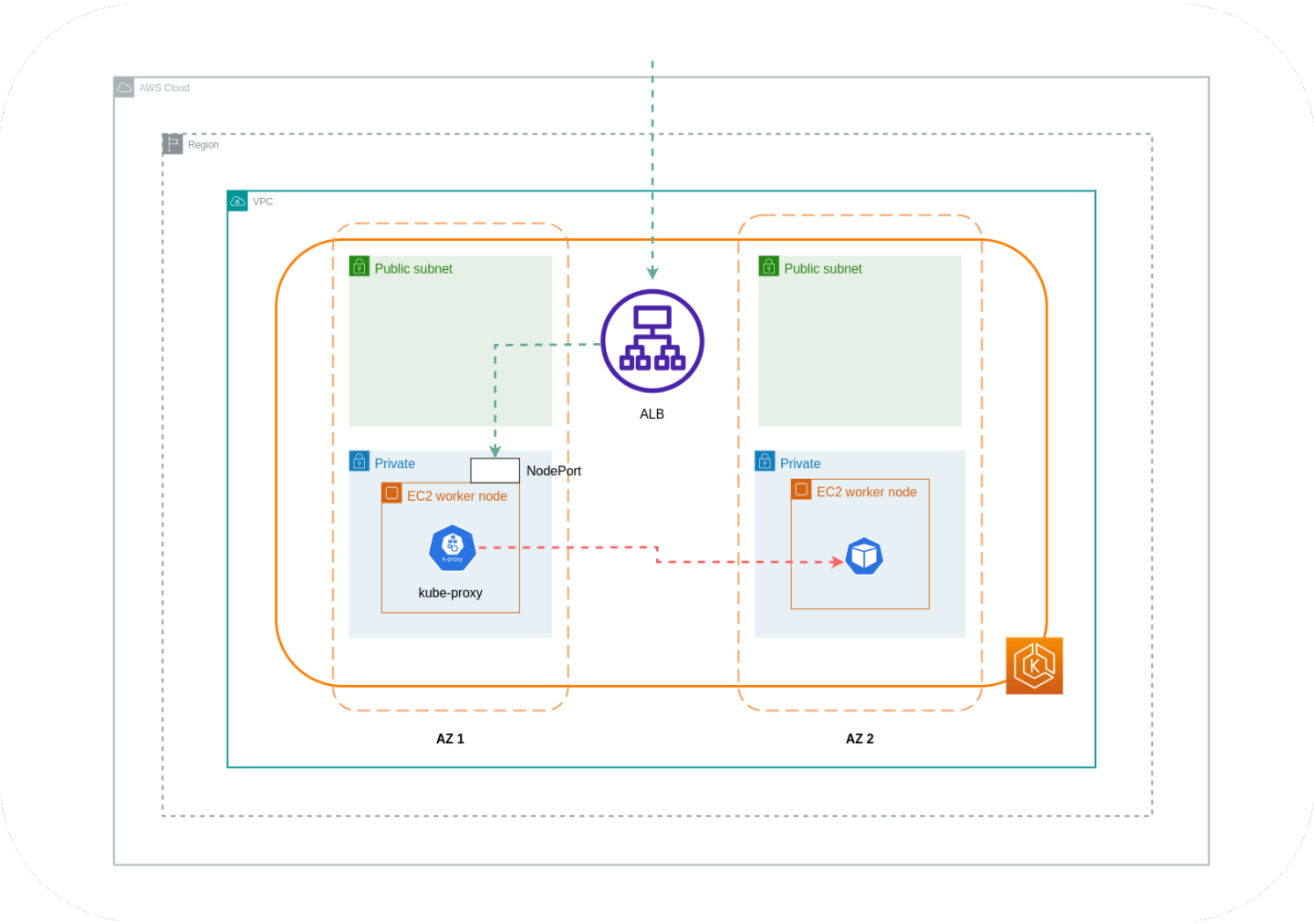
Load Balancerからポッドへの通信

EKS ワークロードは通常、EKS クラスター内の関連する Pod にトラフィックを分散するロードバランサーによって前面に配置されます。アーキテクチャは、内部および/または外部向けロードバランサーで構成される場合があります。アーキテクチャとネットワークトラフィックの設定によっては、ロードバランサーと Pod 間の通信がデータ転送料金に大きな影響を与える可能性があります。

AWS Load Balancer Controller を使用して、ELB リソース (ALB および NLB) の作成を自動的に管理できます。このようなセットアップで発生するデータ転送料金は、ネットワークトラフィックによって取得されるパスによって異なります。AWS Load Balancer Controller は、インスタンスモードと ip モードの 2 つのネットワークトラフィックモードをサポートしています。

インスタンスモードを使用すると、EKS クラスターの各ノードで NodePort が開きます。その後、ロードバランサーはノード間でトラフィックを均等にプロキシします。ノードで送信先 Pod が実行されている場合、データ転送コストは発生しません。ただし、送信先 Pod が別のノードにあり、トラフィックを受信する NodePort とは異なる AZ にある場合、kube-proxy から送信先 Pod への追加のネットワークホップが発生します。このようなシナリオでは、AZ 間のデータ転送料金が発生します。ノード間でトラフィックが均等に分散されるため、kube-proxies から関連する送信先 Pod へのクロスゾーンネットワークトラフィックホップに関連する追加のデータ転送料金が発生する可能性が高くなります。

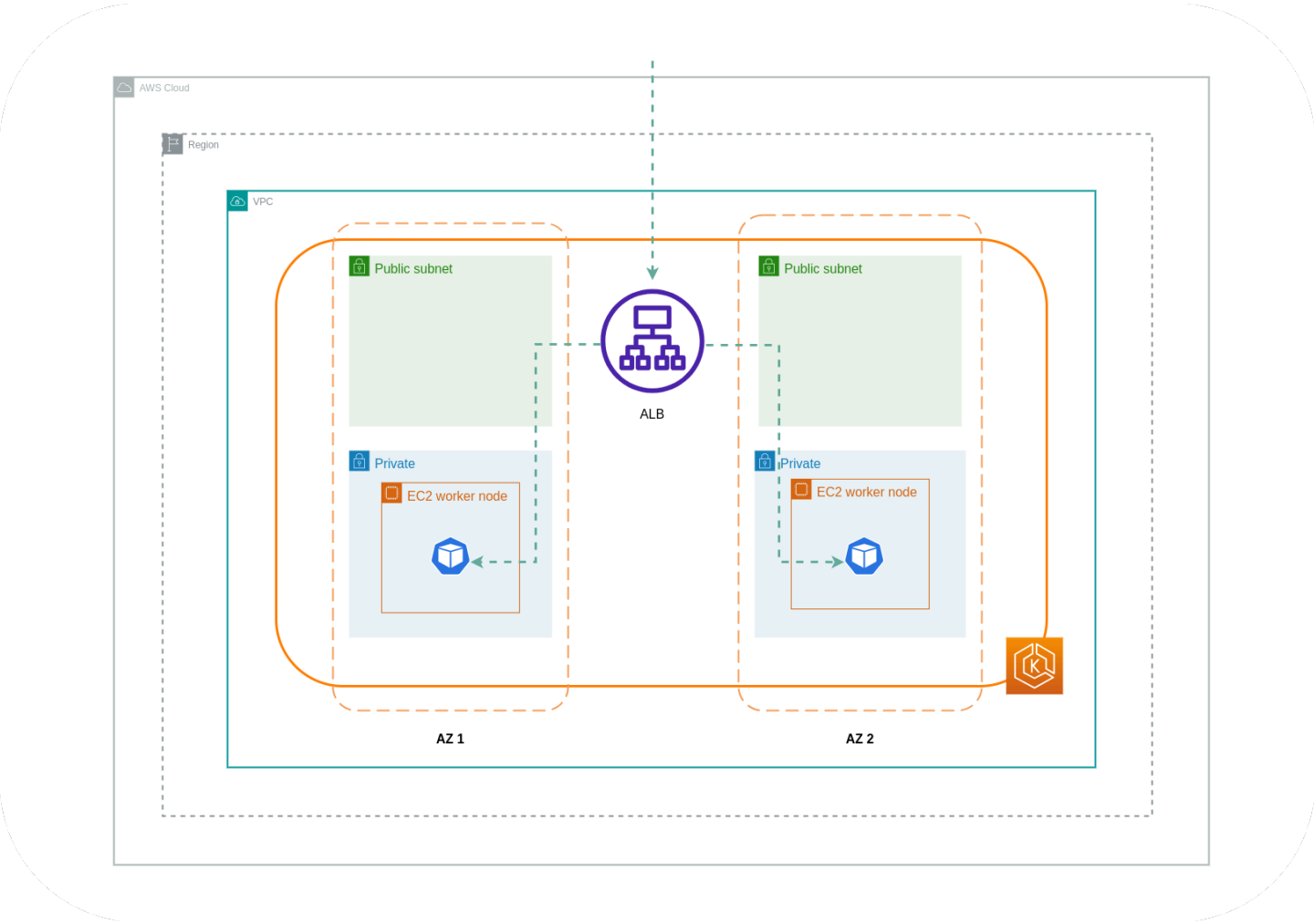
次の図は、ロードバランサーから NodePort に、その後 から別の AZ の別のノードの宛先 Pod kube-proxyに流れるトラフィックのネットワークパスを示しています。これは、インスタンスモード設定の例です。



ip モードを使用する場合、ネットワークトラフィックはロードバランサーから送信先 Pod に直接プロキシされます。そのため、このアプローチにはデータ転送料金はかかりません。

> **Note**
>
> データ転送料金を削減するために、ロードバランサーを ip トラフィックモードに設定することをお勧めします。この設定では、ロードバランサーが VPC 内のすべてのサブネットにデプロイされていることを確認することも重要です。

以下の図は、ロードバランサーからネットワーク IP モードで Pod に流れるトラフィックのネットワークパスを示しています。



コンテナレジストリからのデータ転送

Amazon ECR

Amazon ECR プライベートレジストリへのデータ転送は無料です。リージョン内データ転送には料金はかかりませんが、インターネットへのデータ転送とリージョン間のデータ転送には、転送の両側でインターネットデータ転送料金が請求されます。

ECRs の組み込みイメージレプリケーション機能を使用して、関連するコンテナイメージをワークロードと同じリージョンにレプリケートする必要があります。これにより、レプリケーションは 1 回課金され、同じリージョン (リージョン内) のイメージプルはすべて無料になります。

インターフェイス VPC エンドポイントを使用してリージョン内の ECR リポジトリに接続することで、ECR からのイメージのプル (データ転送) に関連するデータ転送コストをさらに削減できます。ECR のパブリック AWS エンドポイントに接続する別の方法 (NAT ゲートウェイとインター

ネットゲートウェイ経由) では、データ処理と転送のコストが高くなります。次のセクションでは、ワークロードと AWS サービス間のデータ転送コストの削減について詳しく説明します。

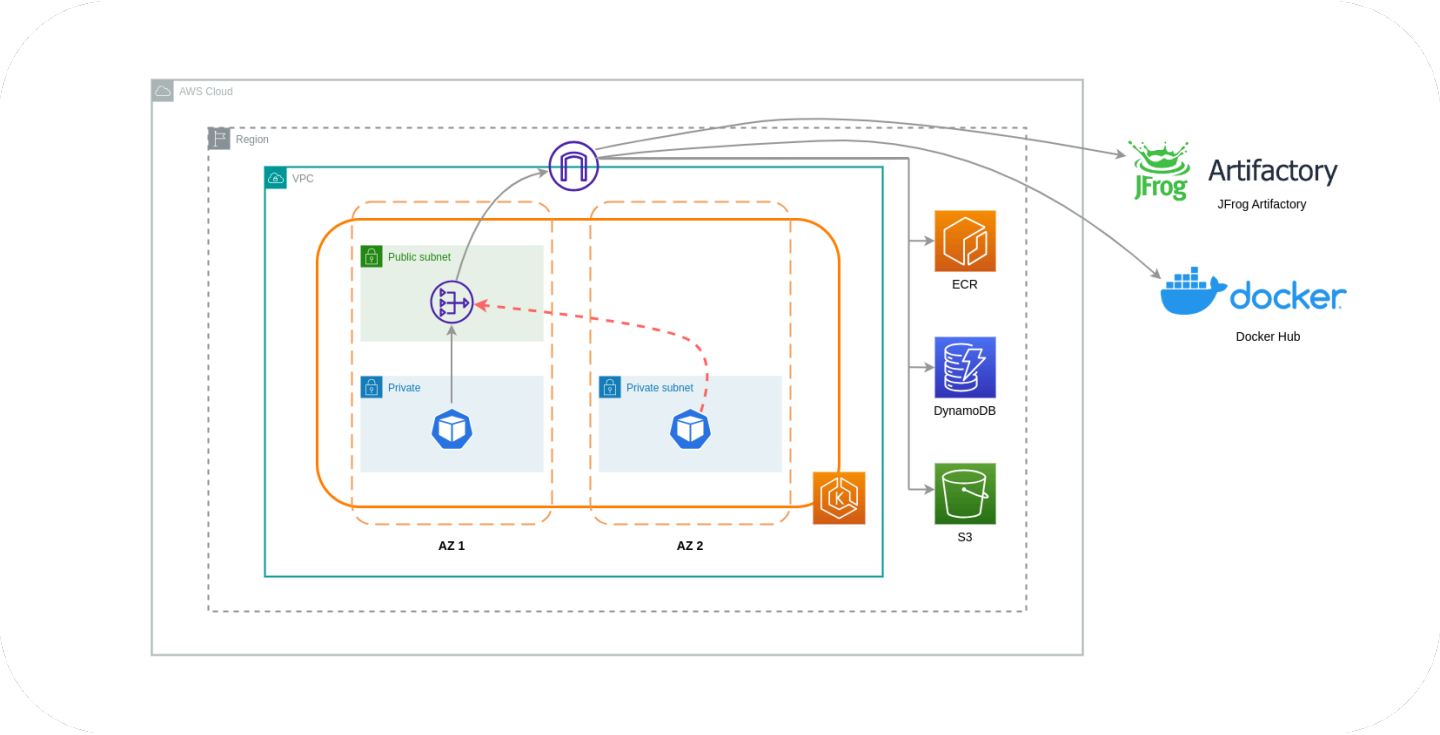
特に大きなイメージでワークロードを実行している場合は、キャッシュされたコンテナイメージを使用して独自のカスタム Amazon マシンイメージ (AMIs) を構築できます。これにより、コンテナレジストリから EKS ワーカーノードへの初期イメージのプル時間と潜在的なデータ転送コストを削減できます。

インターネットおよび AWS サービスへのデータ転送

インターネット経由で Kubernetes ワークロードを他の AWS のサービスやサードパーティーのツールやプラットフォームと統合するのが一般的な方法です。関連する送信先との間でトラフィックをルーティングするために使用される基盤となるネットワークインフラストラクチャは、データ転送プロセスで発生するコストに影響を与える可能性があります。

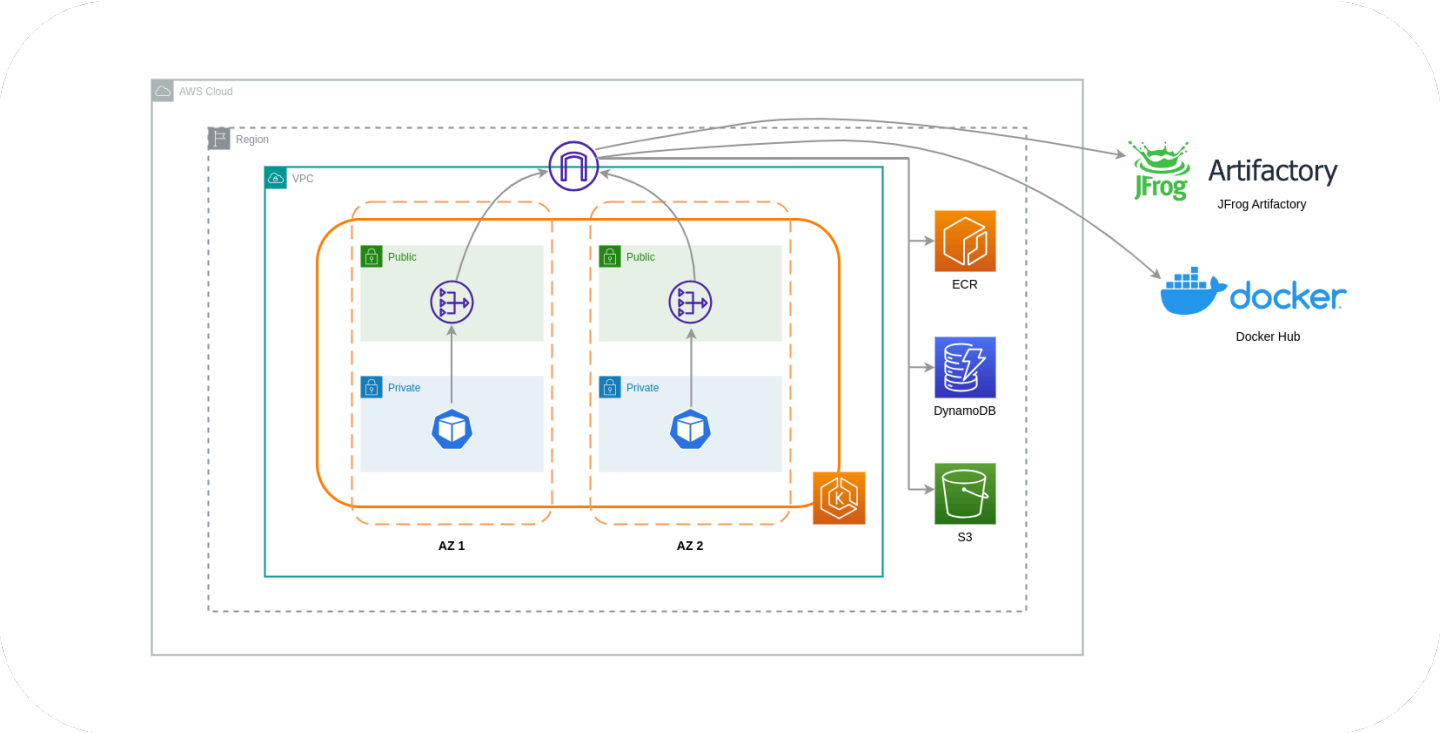
NAT ゲートウェイの使用

NAT ゲートウェイは、ネットワークアドレス変換 (NAT) を実行するネットワークコンポーネントです。次の図は、他の AWS サービス (Amazon ECR、DynamoDB、S3) およびサードパーティープラットフォームと通信する EKS クラスター内の Pod を示しています。この例では、ポッドは個別の AZs。インターネットからトラフィックを送受信するために、NAT ゲートウェイが 1 つの AZ のパブリックサブネットにデプロイされ、プライベート IP アドレスを持つすべてのリソースが 1 つのパブリック IP アドレスを共有してインターネットにアクセスできるようになります。この NAT Gateway は、インターネットゲートウェイコンポーネントと通信し、パケットを最終送信先に送信できるようにします。



このようなユースケースで NAT ゲートウェイを使用する場合、各 AZ に NAT ゲートウェイをデプロイすることで、データ転送コストを最小限に抑えることができます。これにより、インターネットにルーティングされるトラフィックは同じ AZ の NAT ゲートウェイを通過し、AZ 間のデータ転送を回避できます。ただし、AZ 間のデータ転送のコストを節約できますが、この設定はアーキテクチャに追加の NAT ゲートウェイのコストが発生することを意味します。

この推奨アプローチを次の図に示します。



VPC エンドポイントの使用

このようなアーキテクチャのコストをさらに削減するには、VPC エンドポイントを使用してワークロードと AWS サービス間の接続を確立する必要があります。VPC エンドポイントを使用すると、データ/ネットワークパケットがインターネットを通過することなく、VPC 内から AWS のサービスにアクセスできます。すべてのトラフィックは内部であり、AWS ネットワーク内にとどまります。VPC エンドポイントには、インターフェイス VPC エンドポイント (多くの AWS サービスでサポート) とゲートウェイ VPC エンドポイント (S3 と DynamoDB でのみサポート) の 2 種類があります。

ゲートウェイ VPC エンドポイント

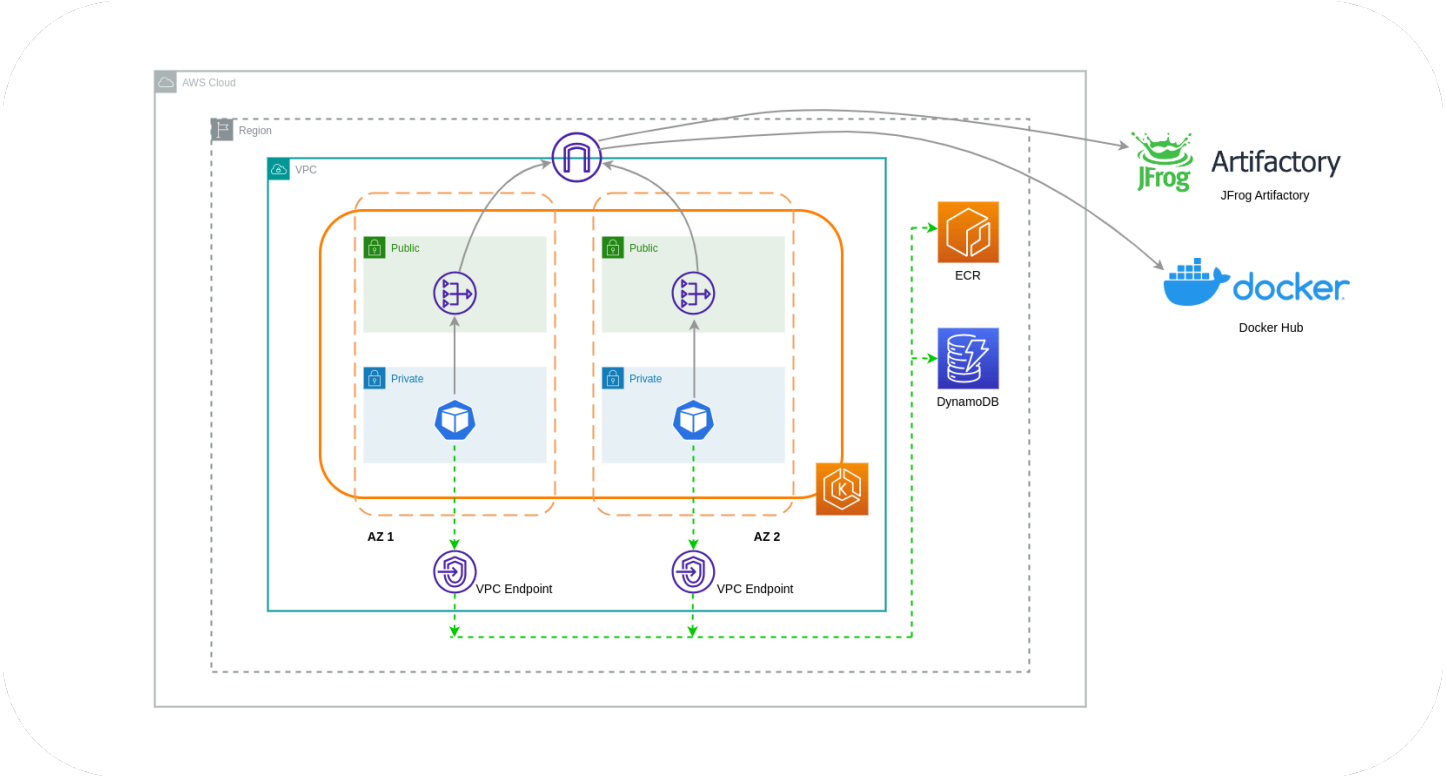
Gateway VPC エンドポイントに関連する時間単位のコストやデータ転送コストはありません。Gateway VPC エンドポイントを使用する場合は、VPC の境界を越えて拡張できないことに注意してください。VPC ピアリング、VPN ネットワーク、または Direct Connect 経由で使用することはできません。

インターフェイス VPC エンドポイント

VPC エンドポイントには時間単位の料金がかかり、基盤となる ENI を介したデータ処理には追加料金がかかります。AZ 間のデータ転送は [非課金] (https://aws.amazon.com/about-aws/whats-

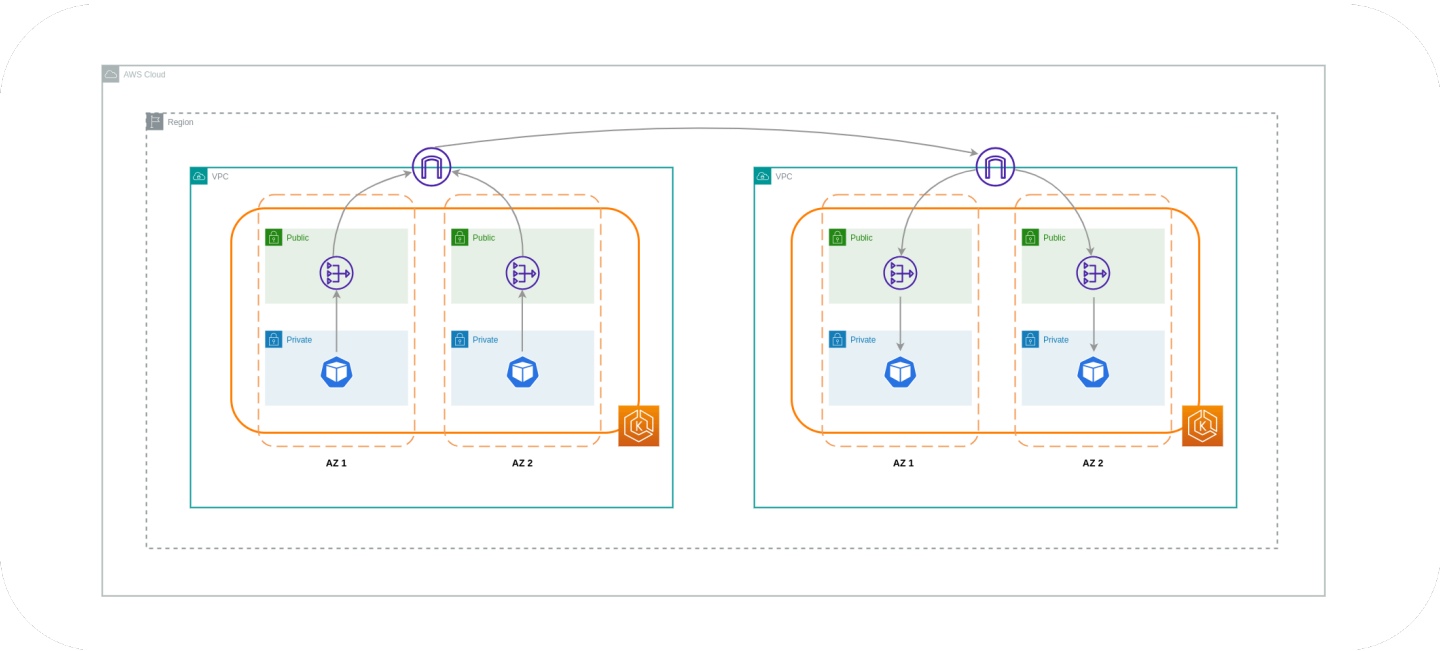
new/2022/04/aws-data-transfer-price-reduction-privatelink-transit-gateway-client-vpn-services/) であることに注意してください。

次の図は、VPC エンドポイントを介して AWS サービスと通信するポッドを示しています。



VPCs間のデータ転送

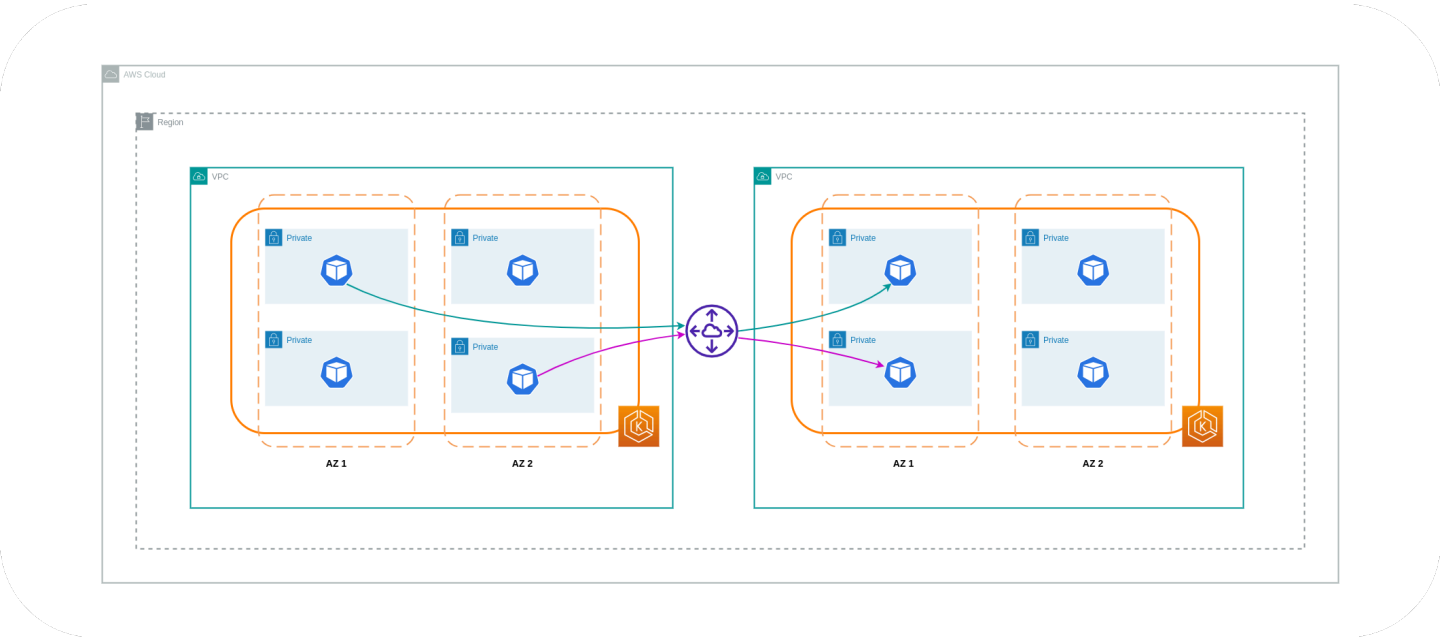
場合によっては、異なる VPCs (同じ AWS リージョン内) にワークロードがあり、相互に通信する必要がある場合があります。これは、トラフィックがそれぞれの VPCs にアタッチされたインターネットゲートウェイを介してパブリックインターネットを通過できるようにすることで実現できます。このような通信を有効にするには、EC2 インスタンス、NAT ゲートウェイ、NAT インスタンスなどのインフラストラクチャコンポーネントをパブリックサブネットにデプロイします。ただし、これらのコンポーネントを含むセットアップでは、VPCs との間でデータを処理/転送するための料金が発生します。個別の VPCs との間のトラフィックが AZs 間で移動する場合は、データ転送に追加料金が発生します。以下の図は、NAT ゲートウェイとインターネットゲートウェイを使用して、異なる VPCs 内のワークロード間の通信を確立するセットアップを示しています。



VPC ピアリング接続

このようなユースケースのコストを削減するには、VPC ピアリングを使用できます。VPC ピアリング接続では、同じ AZ 内に留まるネットワークトラフィックに対してデータ転送料金は発生しません。トラフィックが AZs を超えると、コストが発生します。ただし、同じ AWS リージョン内の別々の VPC 内のワークロード間のコスト効率の高い通信にはVPCs ピアリングアプローチが推奨されます。ただし、VPC ピアリングは推移的なネットワークを許可しないため、主に 1:1 VPC 接続に有効であることに注意してください。

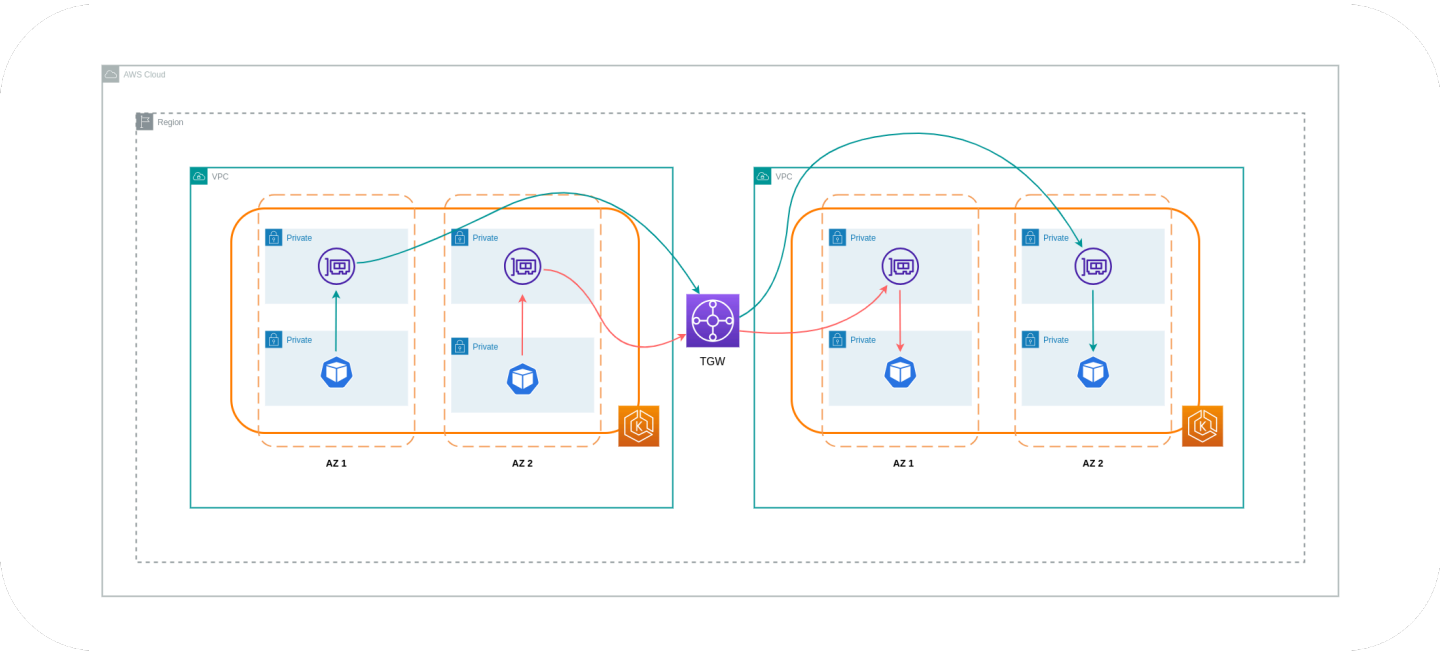
次の図は、VPC ピアリング接続を介したワークロード通信の概要を示しています。



推移的なネットワーク接続

前のセクションで説明したように、VPC ピアリング接続では推移的なネットワーク接続は許可されません。3 つ以上の VPCs を推移的なネットワーク要件で接続する場合は、Transit Gateway (TGW) を使用する必要があります。これにより、VPC ピアリングの制限や、複数の VPC 間の複数の VPC ピアリング接続に関連する運用オーバーヘッドVPCs克服できます。TGW に送信されたデータに対して、時間単位で課金されます。TGW を通過する AZ 間トラフィックに関連する送信先コストはありません。

次の図は、異なる VPCs 内の同じ AWS リージョン内のワークロード間で TGW を通過する AZ 間トラフィックを示しています。



Service Mesh の使用

サービスメッシュは、EKS クラスター環境のネットワーク関連コストを削減するために使用できる強力なネットワーク機能を提供します。ただし、サービスメッシュを採用すると、サービスメッシュが環境に導入する運用タスクと複雑さを慎重に検討する必要があります。

トラフィックをアベイラビリティーゾーンに制限する

Istio の Locality 加重ディストリビューションの使用

Istio を使用すると、ルーティングの発生後にネットワークポリシーをトラフィックに適用できます。これは、ローカリティ加重分散などの送信先ルールを使用して行われます。この機能を使用すると、送信元に基づいて特定の送信先に移動できるトラフィックの重み (パーセンテージで表される) を制御できます。このトラフィックのソースは、外部 (またはパブリック向け) ロードバランサーまたはクラスター自体内の Pod のいずれかです。すべての Pod エンドポイントが利用可能になると、加重ラウンドロビン負荷分散アルゴリズムに基づいてローカリティが選択されます。特定のエンドポイントが異常または使用できない場合、利用可能なエンドポイントにこの変更を反映するように、ローカリティの重みが自動的に調整されます。

> **Note**
>
> ローカリティ加重分散を実装する前に、まずネットワークトラフィックパターンと、送信先ルールポリシーがアプリケーションの動作に与える影響を理解する必要があります。そのた

め、AWS X-Ray や Jaeger などのツールを使用して分散トレースメカニズムを導入することが重要です。

上記の Istio 送信先ルールは、ロードバランサーから EKS クラスター内の Pod へのトラフィックを管理するためにも適用できます。ローカル加重分散ルールは、高可用性ロードバランサー (特に Ingress Gateway) からトラフィックを受信するサービスに適用できます。これらのルールにより、ゾーンオリジン - この場合のロードバランサーに基づいて、 へのトラフィック量を制御できます。正しく設定すると、異なる AZs のポッドレプリカにトラフィックを均等またはランダムに分散するロードバランサーと比較して、エグレスクロスゾーントラフィックが少なくなります。

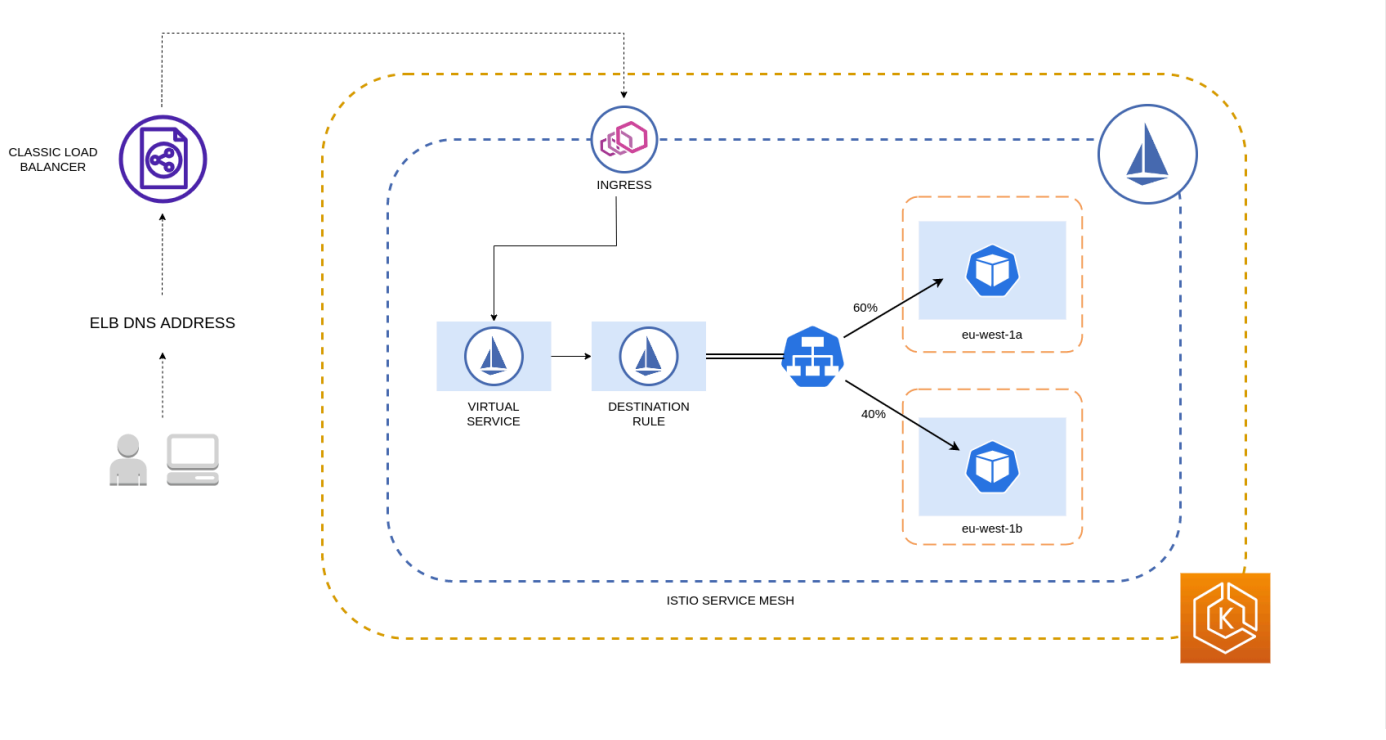
以下は、Istio の送信先ルールリソースのコードブロックの例です。以下に示すように、このリソースは、eu-west-1リージョン内の 3 つの異なる AZs からの受信トラフィックの加重設定を指定します。これらの設定は、特定の AZ からの受信トラフィックの大部分 (この場合は 70%) を、送信元と同じ AZ の送信先にプロキシする必要があることを宣言します。

```
apiVersion: networking.istio.io/v1beta1
kind: DestinationRule
metadata:
  name: express-test-dr
spec:
  host: express-test.default.svc.cluster.local
  trafficPolicy:
    loadBalancer:                      +
      localityLbSetting:
        distribute:
        - from: eu-west-1/eu-west-1a/  +
          to:
            "eu-west-1/eu-west-1a/_": 70
            "eu-west-1/eu-west-1b/_": 20
            "eu-west-1/eu-west-1c/_": 10
        - from: eu-west-1/eu-west-1b/_  +
          to:
            "eu-west-1/eu-west-1a/_": 20
            "eu-west-1/eu-west-1b/_": 70
            "eu-west-1/eu-west-1c/_": 10
        - from: eu-west-1/eu-west-1c/_  +
          to:
            "eu-west-1/eu-west-1a/_": 20
            "eu-west-1/eu-west-1b/_": 10
            "eu-west-1/eu-west-1c/*": 70**
```

```
    connectionPool:
      http:
        http2MaxRequests: 10
        maxRequestsPerConnection: 10
    outlierDetection:
      consecutiveGatewayErrors: 1
      interval: 1m
      baseEjectionTime: 30s
```

> **Note**
>
> 分散先として指定できる最小重みは 1% です。これは、メイン送信先のエンドポイントが異常または使用できなくなった場合に備えて、フェイルオーバーリージョンとゾーンを維持するためです。

次の図は、eu-west-1 リージョンに高可用性ロードバランサーがあり、ローカリティ加重分散が適用されるシナリオを示しています。この図の送信先ルールポリシーは、eu-west-1a からのトラフィックの 60% を同じ AZ の Pod に送信するように設定されていますが、eu-west-1a からのトラフィックの 40% は eu-west-1b の Pod に送信する必要があります。



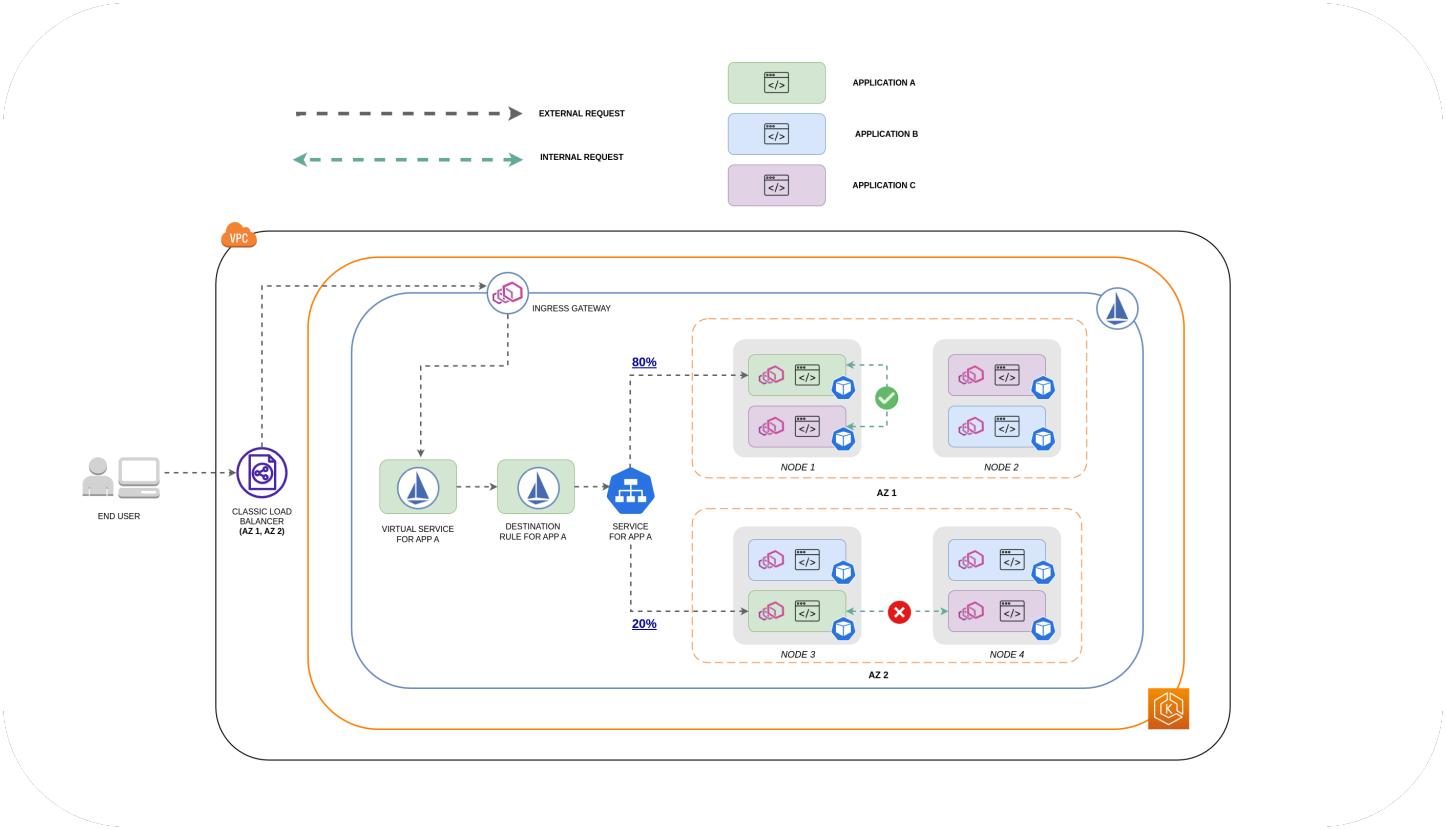
アベイラビリティーゾーンとノードへのトラフィックの制限

Istio での サービス内部トラフィックポリシーの使用

Pod 間の外部着信トラフィックと内部トラフィックに関連するネットワークコストを削減するために、Istio の送信先ルールと Kubernetes Service の内部トラフィックポリシーを組み合わせることができます。Istio 送信先ルールをサービス内部トラフィックポリシーと組み合わせる方法は、主に 3 つの要素によって異なります。

- マイクロサービスの役割
- マイクロサービス全体のネットワークトラフィックパターン
- Kubernetes クラスタートポロジ全体にマイクロサービスをデプロイする方法

次の図は、ネストされたリクエストの場合のネットワークフローと、前述のポリシーがトラフィックをどのように制御するかを示しています。



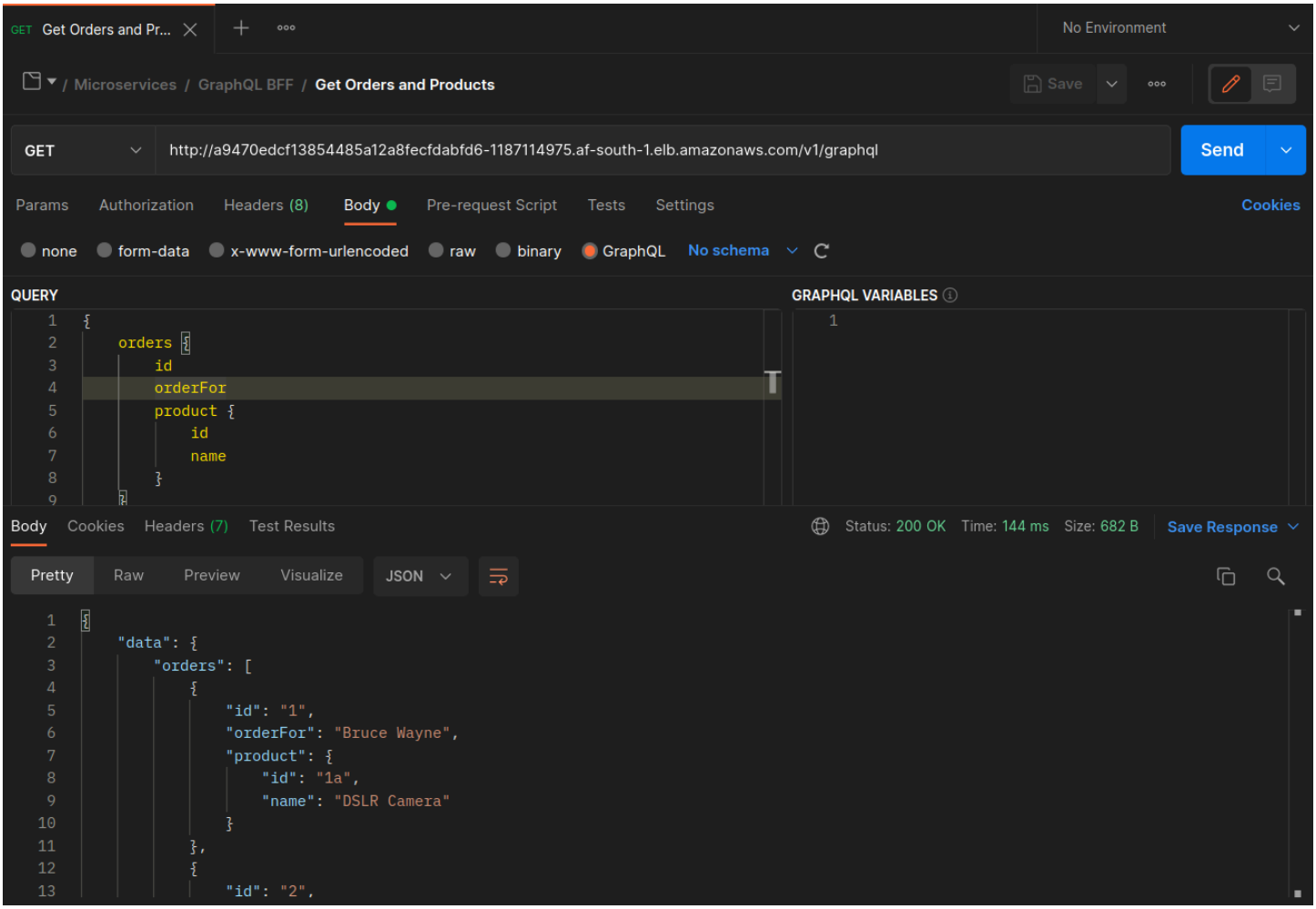
1. エンドユーザーは APP A にリクエストを行い、APP C にネストされたリクエストを行います。このリクエストは、最初に高可用性ロードバランサーに送信されます。ロードバランサーには、上記の図に示すように、AZ 1 および AZ 2 にインスタンスがあります。

2. その後、外部受信リクエストは Istio Virtual Service によって正しい宛先にルーティングされます。
3. リクエストがルーティングされると、Istio 送信先ルールは、送信元 (AZs 1 または AZ 2) に基づいて、各 AZ へのトラフィック量を制御します。
4. その後、トラフィックは APP A のサービスに送信され、それぞれの Pod エンドポイントにプロキシされます。図に示すように、着信トラフィックの 80% が AZ 1 の Pod エンドポイントに送信され、着信トラフィックの 20% が AZ 2 に送信されます。
5. 次に、APP A は APP C に内部リクエストを行います。 APP C のサービスでは、内部トラフィックポリシーが有効になっています (`internalTrafficPolicy`: Local`)。
6. APP C のノードローカルエンドポイントが使用可能であるため、APP A (NODE 1) から APP Cへの内部リクエストは成功しました。
7. APP C に使用可能なノードローカルエンドポイントがないため、APP A (NODE 3) から APP C への内部リクエストは失敗します。 図に示すように、APP C には NODE 3 にレプリカがありません。 **

以下のスクリーンショットは、このアプローチのライブ例からキャプチャされています。最初のスクリーンショットのセットは、 への外部リクエストの成功`graphql`と、 からノード 上の同じ場所にある`orders`レプリカ`graphql`へのネストされたリクエストの成功を示しています`ip-10-0-0-151.af-south-1.compute.internal`。

test-results.txt

```
kubectl get pods -n ecommerce -o wide --field-selector spec.nodeName=ip-10-0-0-147.af-south-1.compute.i
NAME                        READY   STATUS    RESTARTS   AGE   IP          NODE
products-8567b458c8-brgdp   2/2     Running   0          64m   10.0.0.83   ip-10-0-0-147.af-south-1.com
--------------------------------------------------------------------------
kubectl get pods -n ecommerce -o wide --field-selector spec.nodeName=ip-10-0-1-153.af-south-1.compute.i
NAME                        READY   STATUS    RESTARTS   AGE   IP         NODE
products-8567b458c8-5shq8   2/2     Running   0          65m   10.0.1.4   ip-10-0-1-153.af-south-1.comp
--------------------------------------------------------------------------
kubectl get pods -n ecommerce -o wide --field-selector spec.nodeName=ip-10-0-1-68.af-south-1.compute.in
NAME                      READY   STATUS    RESTARTS   AGE   IP           NODE
orders-684bfbd6d9-5dxcf   2/2     Running   0          22m   10.0.1.145   ip-10-0-1-68.af-south-1.compu
--------------------------------------------------------------------------
kubectl get pods -n ecommerce -o wide --field-selector spec.nodeName=ip-10-0-0-151.af-south-1.compute.i
NAME                      READY   STATUS    RESTARTS   AGE   IP            NODE
graphql-8f877c686-k7k9b   2/2     Running   0          18m   10.0.0.239    ip-10-0-0-151.af-south-1.comp
orders-684bfbd6d9-rc2zp   2/2     Running   0          18m   10.0.0.182    ip-10-0-0-151.af-south-1.comp
```



Istio を使用すると、プロキシが認識している [アップストリームクラスター](https://www.envoyproxy.io/docs/envoy/latest/intro/arch_overview/intro/terminology) とエンドポイントの統計を検証してエクスポートできます。これは、ネットワークフローの全体像と、ワークロードのサービス間の分散の割合を提供するのに役立ちます。同じ例を続けると、`graphql`プロキシが認識している`orders`エンドポイントは、次のコマンドを使用して取得できます。

```
kubectl exec -it deploy/graphql -n ecommerce -c istio-proxy -- curl localhost:15000/
clusters | grep orders
```

```
...
orders-service.ecommerce.svc.cluster.local::10.0.1.33:3003::**rq_error::0**
orders-service.ecommerce.svc.cluster.local::10.0.1.33:3003::**rq_success::119**
orders-service.ecommerce.svc.cluster.local::10.0.1.33:3003::**rq_timeout::0**
orders-service.ecommerce.svc.cluster.local::10.0.1.33:3003::**rq_total::119**
orders-service.ecommerce.svc.cluster.local::10.0.1.33:3003::**health_flags::healthy**
orders-service.ecommerce.svc.cluster.local::10.0.1.33:3003::**region::af-south-1**
```

```
orders-service.ecommerce.svc.cluster.local::10.0.1.33:3003::**zone::af-south-1b**
...
```

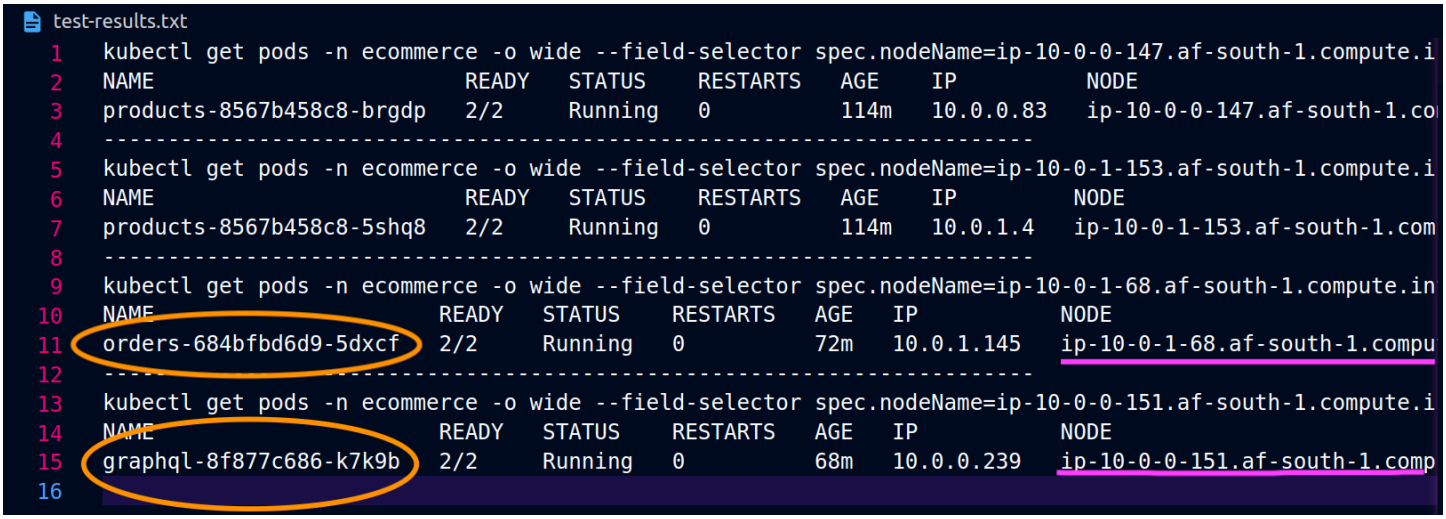
この場合、graphqlプロキシはノードを共有するレプリカのordersエンドポイントのみを認識します。注文サービスからinternalTrafficPolicy: Local設定を削除し、上記のようなコマンドを再実行すると、結果は異なるノードに分散されたレプリカのすべてのエンドポイントを返します。さらに、それぞれのエンドポイントrq_totalの を調べると、ネットワークディストリビューションのシェアが比較的均等であることがわかります。したがって、エンドポイントが異なる AZs で実行されているアップストリームサービスに関連付けられている場合、ゾーン間でのこのネットワーク分散によりコストが高くなります。

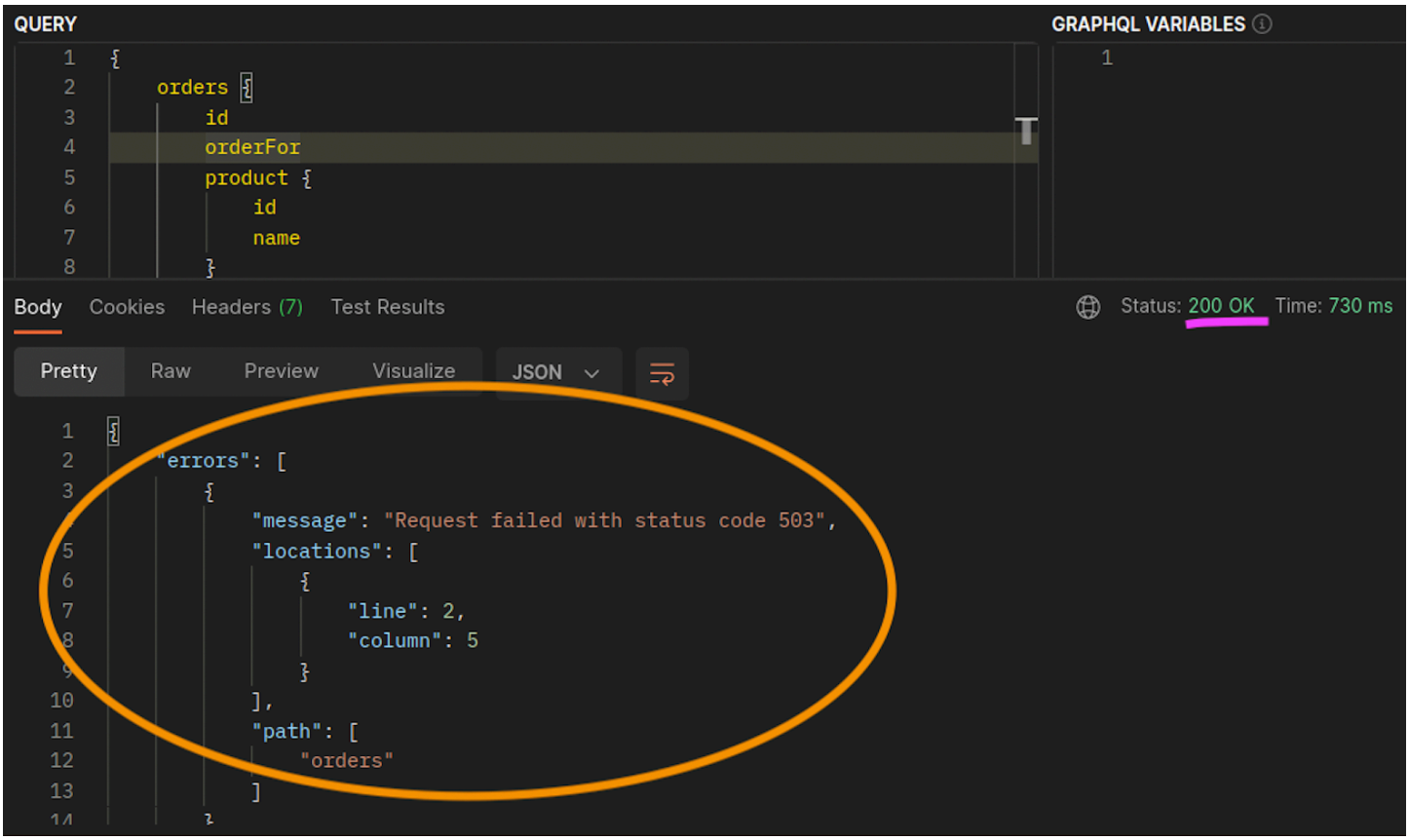
上記の前のセクションで説明したように、ポッドアフィニティを使用することで、頻繁に通信するポッドを共同配置できます。

```
...
spec:
...
  template:
    metadata:
      labels:
        app: graphql
        role: api
        workload: ecommerce
    spec:
      affinity:
        podAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
          - labelSelector:
              matchExpressions:
              - key: app
                operator: In
                values:
                - orders
            topologyKey: "kubernetes.io/hostname"
      nodeSelector:
        managedBy: karpenter
        billing-team: ecommerce
...
```

graphql と orders レプリカが同じノード (ip-10-0-0-151.af-south-1.compute.internal) に共存しない場合、 への最初のリクエストgraphqlは、以下

の Postman スクリーンショット200 response codeの「」に記載されているように成功しますが、 から graphqlへのネストされた 2 番目のリクエストは でordersが失敗します503 response code。





その他のリソース

- Istio を使用して EKS のレイテンシーとデータ転送コストに対処する

- Amazon Elastic Kubernetes Service のネットワークトラフィックに対するトポロジ対応ヒントの影響を調べる
- Amazon EKS Cross-AZ ポッドをポッドネットワークバイトに可視化する
- Istio で AZ トラフィックを最適化する
- トポロジ対応ルーティングで AZ トラフィックを最適化する
- サービス内部トラフィックポリシーによる Kubernetes のコストとパフォーマンスの最適化
- Istio とサービスの内部トラフィックポリシーを使用して Kubernetes のコストとパフォーマンスを最適化する
- 一般的なアーキテクチャでのデータ転送コストの概要
- AWS コンテナサービスのデータ転送コストについて

[Storage (ストレージ)]

概要

短期または長期でデータを保存する必要があるアプリケーションを実行するシナリオがあります。このようなユースケースでは、ポッドによってボリュームを定義およびマウントして、コンテナが異なるストレージメカニズムを利用できるようにします。Kubernetes は、エフェメラルストレージと永続ストレージのさまざまなタイプのボリュームをサポートしています。ストレージの選択は、アプリケーションの要件に大きく依存します。各アプローチにはコストへの影響があり、以下に説明するプラクティスは、EKS 環境で何らかの形式のストレージを必要とするワークロードのコスト効率を実現するのに役立ちます。

エフェメラルボリューム

エフェメラルボリュームは、一時的なローカルボリュームを必要とするが、再起動後にデータを保持する必要がないアプリケーション用です。これには、スクラッチスペース、キャッシュ、設定データやシークレットなどの読み取り専用入力データの要件が含まれます。Kubernetes エフェメラルボリュームの詳細については、こちらを参照してください。ほとんどのエフェメラルボリューム (例: emptyDir、configMap、downwardAPI、シークレット、ホストパス) は、ローカルにアタッチされた書き込み可能デバイス (通常はルートディスク) または RAM によってバックアップされるため、最もコスト効率が高くパフォーマンスの高いホストボリュームを選択することが重要です。

EBS ボリュームの使用

ホストルートボリュームとして gp3 から開始することをお勧めします。これは Amazon EBS が提供する最新の汎用 SSD ボリュームであり、gp2 ボリュームと比較して GB あたり低価格 (最大 20%) を提供します。

Amazon EC2 インスタンスストアの使用

Amazon EC2 インスタンスストアは、EC2 インスタンスに一時的なブロックレベルのストレージを提供します。EC2 インスタンスストアが提供するストレージは、ホストに物理的にアタッチされたディスクからアクセスできます。Amazon EBS とは異なり、インスタンスストアボリュームはインスタンスの起動時にのみアタッチでき、これらのボリュームはインスタンスの存続期間中にのみ存在します。他のインスタンスにデタッチおよび再アタッチすることはできません。Amazon EC2 インスタンスストアの詳細については、こちらを参照してください。インスタンスストアボリュームには追加料金はかかりません。これにより、EBS ボリュームが大きい一般的な EC2 インスタンスよりもコスト効率が高くなります (インスタンスストアボリューム) 。

Kubernetes でローカルストアボリュームを使用するには、Amazon EC2 ユーザーデータを使用してディスクをパーティション化、設定、フォーマットし、ポッド仕様でボリュームを HostPath としてマウントできるようにする必要があります。または、Local Persistent Volume Static Provisioner を活用して、ローカルストレージ管理を簡素化することもできます。Local Persistent Volume 静的プロビジョナーを使用すると、標準の Kubernetes PersistentVolumeClaim (PVC) インターフェイスを介してローカルインスタンスストアボリュームにアクセスできます。さらに、ノードアフィニティ情報を含む PersistentVolumes (PVs) をプロビジョニングして、Pod を正しいノードにスケジュールします。Kubernetes PersistentVolumes を使用していますが、EC2 インスタンスストアボリュームは本質的にエフェメラルです。エフェメラルディスクに書き込まれたデータは、インスタンスの存続期間中にのみ使用できます。インスタンスが終了すると、 もデータになります。詳細については、このブログを参照してください。

Amazon EC2 インスタンスストアボリュームを使用する場合、合計 IOPS 制限はホストと共有され、ポッドを特定のホストにバインドすることに注意してください。Amazon EC2 インスタンスストアボリュームを採用する前に、ワークロードの要件を徹底的に確認する必要があります。

永続ボリューム

Kubernetes は通常、実行中のステートレスアプリケーションに関連付けられます。ただし、1 つのリクエストから次のリクエストまで永続的なデータや情報を保持する必要があるマイクロサービスを実行するシナリオがあります。データベースは、このようなユースケースの一般的な例です。ただし、ポッドとその中のコンテナまたはプロセスは、本質的にエフェメラルです。Pod の存続期間を

超えてデータを保持するには、PVs を使用して、Pod から独立した特定の場所にあるストレージへのアクセスを定義できます。PVs に関連するコストは、使用されているストレージのタイプとアプリケーションによる消費方法に大きく依存します。

Amazon EKS の Kubernetes PVs をサポートするストレージオプションには、さまざまなタイプがあります。以下に説明するストレージオプションは、Amazon EBS、Amazon EFS、Amazon FSx for Lustre、Amazon FSx for NetApp ONTAP です。

Amazon Elastic Block Store (EBS) ボリューム

Amazon EBS ボリュームは、Kubernetes PVs として使用して、ブロックレベルのストレージボリュームを提供できます。これらは、ランダムな読み取りと書き込みに依存するデータベースや、長時間の継続的な読み取りと書き込みを実行するスループットを多用するアプリケーションに適しています。Amazon Elastic Block Store Container Storage Interface (CSI) ドライバーを使用すると、Amazon EKS クラスターは永続的ボリュームの Amazon EBS ボリュームのライフサイクルを管理できます。Container Storage Interface は、Kubernetes とストレージシステム間のインタラクションを有効にし、容易にします。CSI ドライバーを EKS クラスターにデプロイすると、永続ボリューム (PVs)、永続ボリュームクレーム (PVCs)、ストレージクラス (SCs) などのネイティブ Kubernetes ストレージリソースを介してその機能にアクセスできます。このリンクでは、Amazon EBS ボリュームを Amazon EBS CSI ドライバーとやり取りする方法の実例を示します。

適切なボリュームの選択

価格とパフォーマンスの適切なバランスを実現するため、最新世代のブロックストレージ (gp3) を使用することをお勧めします。また、追加のブロックストレージ容量をプロビジョニングすることなく、ボリュームサイズに関係なくボリューム IOPS とスループットをスケールすることもできます。現在 gp2 ボリュームを使用している場合は、gp3 ボリュームへの移行を強くお勧めします。ブログ記事「Amazon EKS クラスターを gp2 から gp3 EBS ボリュームに移行する」では、アプリケーションのダウンタイムを必要とする CSI ボリュームスナップショット機能を使用して、バックアップと復元を使用して Amazon EKS クラスターの gp3 上の gp2 から移行する方法について説明します。

Amazon EBS では、ボリュームサイズ、IOPS、スループットなどのボリューム特性をオンラインで変更できます。この機能を使用すると、このブログで説明されているように、EBS CSI ドライバー v1.19.0 以降を必要とする PVC 注釈を使用するか、ここで説明されている VolumeAttributesClass API を使用して Amazon EKS v1.31 および EBS CSI ドライバー 1.35 から始めることで、アプリケーションのダウンタイムなしで gp3 上の gp2 から移行できます。

1 つの gp3 ボリュームがサポートできる以上のパフォーマンスとボリュームを必要とするアプリケーションがある場合は、io2 ブロックエクスプレスの使用を検討する必要があります。このタイプのストレージは、SAP HANA や低レイテンシー要件の他の大規模なデータベースなど、最大、最

も I/O 負荷の高いミッションクリティカルなデプロイに最適です。インスタンスの EBS パフォーマンスはインスタンスのパフォーマンス制限によって制限されるため、すべてのインスタンスが io2 Block Express ボリュームをサポートしているわけではありません。このドキュメントでは、サポートされているインスタンスタイプやその他の考慮事項を確認できます。

単一の gp3 ボリュームは、最大 16,000 IOPS、最大スループット 1,000 MiB/秒、最大 16TiB をサポートできます。最大 256,000 IOPS、4,000 MiB/秒、スループット、64TiB を提供する最新世代のプロビジョンド IOPS SSD ボリューム。

これらのオプションのうち、ストレージのパフォーマンスとコストをアプリケーションのニーズに最も合わせる必要があります。

時間の経過に伴うモニタリングと最適化

アプリケーションのベースラインパフォーマンスを理解し、選択したボリュームを監視して、それが要件/期待を満たしているかどうか、または過剰にプロビジョニングされているかどうかを確認することが重要です (プロビジョニングされた IOPS が完全に活用されていないシナリオなど)。

最初から大量のボリュームを割り当てる代わりに、データを蓄積するにつれてボリュームのサイズを徐々に増やすことができます。Amazon Elastic Block Store CSI ドライバー (aws-ebs-csi-driver) のボリュームサイズ変更機能を使用して、ボリュームのサイズを動的に変更できます。EBS ボリュームサイズは増やすことしかできないことに注意してください。

ダングリング EBS ボリュームを特定して削除するには、AWS の信頼されたアドバイザーのコスト最適化カテゴリを使用できます。この機能は、アタッチされていないボリュームや、一定期間の書き込みアクティビティが非常に少ないボリュームを特定するのに役立ちます。ライブ Kubernetes クラスターをスキャンし、デプロイされたリソースと設定に関する潜在的な問題をレポートする、Popeye と呼ばれるクラウドネイティブのオープンソースの読み取り専用ツールがあります。たとえば、未使用の PVs と PVCs をスキャンして、それらがバインドされているかどうか、またはボリュームマウントエラーがあるかどうかを確認できます。

モニタリングの詳細については、EKS コスト最適化オブザーバビリティガイドを参照してください。

考慮すべきもう 1 つのオプションは、AWS Compute Optimizer Amazon EBS ボリュームのレコメンデーションです。このツールは、最適なボリューム設定と必要な適切なパフォーマンスレベルを自動的に特定します。例えば、過去 14 日間の最大使用率に基づいて、プロビジョニングされた IOPS、ボリュームサイズ、EBS ボリュームのタイプに関する最適な設定に使用できます。また、レコメンデーションから得られる月間コスト削減の可能性も定量化します。詳細については、このブログを参照してください。

バックアップの保持ポリシー

point-in-timeスナップショットを作成することで、Amazon EBS ボリュームのデータをバックアップできます。Amazon EBS CSI ドライバーはボリュームスナップショットをサポートします。ここで説明するステップを使用して、スナップショットを作成し、EBS PV を復元する方法について説明しますhttps://github.com/kubernetes-sigs/aws-ebs-csi-driver/blob/master/examples/kubernetes/snapshot/README.md。

後続のスナップショットは増分バックアップです。つまり、最新のスナップショットの後に変更されたデバイスのブロックのみが保存されます。これにより、スナップショットを作成するのに要する時間が最小限に抑えられ、データを複製しないことで、ストレージコストが節約されます。ただし、適切な保持ポリシーなしで古い EBS スナップショットの数を増やすと、大規模な運用時に予期しないコストが発生する可能性があります。AWS API を使用して Amazon EBS ボリュームを直接バックアップする場合は、Amazon Elastic Block Store (EBS) スナップショットと EBS-backed Amazon マシンイメージ (AMIs) 用の自動化されたポリシーベースのライフサイクル管理ソリューションを提供する Amazon Data Lifecycle Manager (DLM) を活用できます。コンソールを使用すると、EBS スナップショットと AMIs。

> **Note**
>
> 現在、Amazon EBS CSI ドライバーを介して Amazon DLM を使用する方法はありません。

Kubernetes 環境では、Velero というオープンソースツールを使用して EBS 永続ボリュームをバックアップできます。バックアップの有効期限が切れるようにジョブをスケジュールするときに TTL フラグを設定できます。以下は、例として Velero からのガイドです。

Amazon Elastic File System (EFS)

Amazon Elastic File System (EFS) は、サーバーレスで完全に伸縮自在なファイルシステムです。標準のファイルシステムインターフェイスとファイルシステムのセマンティクスを使用してファイルデータを共有し、幅広いワークロードやアプリケーションに対応します。ワークロードやアプリケーションの例としては、Wordpress や Drupal、JIRA や Git などの開発者ツール、Jupyter などの共有ノートブックシステム、ホームディレクトリなどがあります。

Amazon EFS の主な利点の 1 つは、複数のノードと複数のアベイラビリティーゾーンにまたがる複数のコンテナにマウントできることです。もう 1 つの利点は、使用したストレージに対してのみ料金を支払うことです。EFS ファイルシステムは、ファイルを追加および削除すると自動的に拡張および縮小されるため、容量計画が不要になります。

Kubernetes で Amazon EFS を使用するには、Amazon Elastic File System Container Storage Interface (CSI) ドライバー、aws-efs-csi-driver を使用する必要があります。現在、ドライバーはアクセスポイントを動的に作成できます。ただし、Amazon EFS ファイルシステムを最初にプロビジョニングし、Kubernetes ストレージクラスパラメータへの入力として提供する必要があります。

適切な EFS ストレージクラスの選択

Amazon EFS には 4 つのストレージクラスがあります。

2 つの標準ストレージクラス：

- Amazon EFS 標準
- Amazon EFS Standard-Infrequent Access (EFS Standard-IA)

2 つの 1 ゾーンストレージクラス：

- Amazon EFS 1 ゾーン
- Amazon EFS 1 ゾーン低頻度アクセス (EFS 1 ゾーン - IA)

低頻度アクセス (IA) ストレージクラスは、毎日アクセスされないファイルに対してコスト最適化されています。Amazon EFS ライフサイクル管理を使用すると、ライフサイクルポリシーの期間 (7、14、30、60、または 90 日間) アクセスされなかったファイルを IA ストレージクラスに移動できます。これにより、EFS 標準ストレージクラスと EFS 1 ゾーンストレージクラスと比較して、ストレージコストがそれぞれ最大 92% EFS 削減されます。

EFS Intelligent-Tiering を使用すると、ライフサイクル管理はファイルシステムのアクセスパターンを監視し、ファイルを最適なストレージクラスに自動的に移動します。

> **Note**
>
> aws-efs-csi-driver は現在、ストレージクラス、ライフサイクル管理、または Intelligent-Tiering の変更を制御できません。これらは、AWS コンソールまたは EFS APIs を使用して手動でセットアップする必要があります。

> **Note**
>
> aws-efs-csi-driver は、Window ベースのコンテナイメージと互換性がありません。

> **Note**
>
> ファイルシステムのサイズに比例する量のメモリを消費する DiskUsage 関数が原因で、vol-metrics-opt-in (ボリュームメトリクスを出力するため) が有効になっている場合、メモリに既知の問題があります。現在、大量のメモリを消費しないように、大きなファイルシステムでは「--vol-metrics-opt-in」オプションを無効にすることをお勧めします。vol-metrics-opt-in 詳細については、GitHub の問題リンクを参照してください。

Amazon FSx for Lustre

Lustre は、最大数百 GB/秒のスループットとオペレーションごとのミリ秒未満のレイテンシーを必要とするワークロードで一般的に使用される高性能の並列ファイルシステムです。これは、機械学習トレーニング、財務モデリング、HPC、ビデオ処理などのシナリオに使用されます。Amazon FSx for Lustre は、スケーラビリティとパフォーマンスを備えたフルマネージド共有ストレージを提供し、Amazon S3 とシームレスに統合されています。

Amazon EKS の FSx for Lustre CSI ドライバーまたは AWS のセルフマネージド Kubernetes クラスターを使用して、FSx for Lustre でバックアップされた Kubernetes 永続ストレージボリュームを使用できます。詳細と例については、Amazon EKS ドキュメントを参照してください。

Amazon S3 へのリンク

Amazon S3 に存在する耐久性の高い長期データリポジトリを FSx for Lustre ファイルシステムにリンクすることをお勧めします。リンクされると、大きなデータセットは必要に応じて Amazon S3 から FSx for Lustre ファイルシステムに遅延ロードされます。分析と結果を S3 に戻し、Lustre ファイルシステムを削除することもできます。

適切なデプロイとストレージオプションの選択

FSx for Lustre にはさまざまなデプロイオプションがあります。最初のオプションはスクラッチと呼ばれ、データはレプリケートされません。2 番目のオプションは永続と呼ばれ、名前が示すように、データは保持されます。

最初のオプション (スクラッチ) を使用して、一時的な短期データ処理のコストを削減できます。永続デプロイオプションは、AWS アベイラビリティーゾーン内のデータを自動的にレプリケートする長期ストレージ用に設計されています。また、SSD ストレージと HDD ストレージの両方をサポートしています。

必要なデプロイタイプは、FSx for lustre ファイルシステムの Kubernetes StorageClass のパラメータで設定できます。サンプルテンプレートを提供するリンクを次に示します。

> **Note**
>
> レイテンシーの影響を受けやすいワークロードや、最高レベルの IOPS/スループットを必要とするワークロードの場合は、SSD ストレージを選択する必要があります。レイテンシーの影響を受けないスループット重視のワークロードの場合は、HDD ストレージを選択する必要があります。

データ圧縮を有効にする

また、「LZ4」をデータ圧縮タイプとして指定することで、ファイルシステムでデータ圧縮を有効にすることもできます。有効にすると、新しく書き込まれたすべてのファイルは、ディスクに書き込まれる前に FSx for Lustre で自動的に圧縮され、読み取り時に解凍されます。LZ4 データ圧縮アルゴリズムは可逆であるため、元のデータは圧縮されたデータから完全に再構築できます。

データ圧縮タイプは、FSx for lustre ファイルシステムの Kubernetes StorageClass のパラメータで LZ4 として設定できます。値がデフォルトで NONE に設定されている場合、圧縮は無効になります。このリンクにはサンプルテンプレートが用意されています。

> **Note**
>
> Amazon FSx for Lustre は、Window ベースのコンテナイメージと互換性がありません。

Amazon FSx for NetApp ONTAP

Amazon FSx for NetApp ONTAP は、NetApp の ONTAP ファイルシステム上に構築されたフルマネージド共有ストレージです。FSx for ONTAP は、AWS またはオンプレミスで実行されている Linux、Windows、macOS コンピューティングインスタンスから幅広くアクセスできる、機能豊富で高速、柔軟な共有ファイルストレージを提供します。

Amazon FSx for NetApp ONTAP は、1/プライマリ階層と 2/キャパシティプール階層の 2 つのストレージ階層をサポートしています。

プライマリ階層は、アクティブでレイテンシーの影響を受けやすいデータ用のプロビジョニングされた高性能 SSD ベースの階層です。完全に伸縮自在なキャパシティプール階層は、アクセス頻度の低いデータに対してコスト最適化されており、データが階層化されると自動的にスケールされ、事実上

無制限のペタバイトの容量を提供します。キャパシティープールストレージでデータ圧縮と重複排除を有効にして、データが消費するストレージ容量をさらに削減できます。NetApp のポリシーベースのネイティブ FabricPool 機能は、データアクセスパターンを継続的にモニタリングし、ストレージ層間で双方向にデータを自動的に転送して、パフォーマンスとコストを最適化します。

NetApp の Astra Trident は、CSI ドライバーを使用して動的ストレージオーケストレーションを提供します。これにより、Amazon EKS クラスターは、Amazon FSx for NetApp ONTAP ファイルシステムによってバックアップされた永続ボリューム PVs のライフサイクルを管理できます。開始するには、Astra Trident ドキュメントの「Amazon FSx for NetApp ONTAP で Astra Trident を使用」を参照してください。

その他の考慮事項

コンテナイメージのサイズを最小限に抑える

コンテナがデプロイされると、コンテナイメージは複数のレイヤーとしてホストにキャッシュされます。イメージのサイズを減らすことで、ホストに必要なストレージの量を減らすことができます。

最初からスクラッチイメージやディストリビューションコンテナイメージ (アプリケーションとそのランタイム依存関係のみを含む) などのスラムダウンされたベースイメージを使用することで、攻撃領域の削減やイメージのプル時間の短縮など、その他の補助的な利点に加えて、ストレージコストを削減できます。

また、最小限のイメージを作成する簡単で安全な方法を提供する Slim.ai などのオープンソースツールの使用も検討する必要があります。

パッケージ、ツール、アプリケーションの依存関係、ライブラリの複数のレイヤーは、コンテナイメージサイズを簡単に肥大化できます。マルチステージビルドを使用すると、あるステージから別のステージにアーティファクトを選択的にコピーでき、最終イメージから不要なものをすべて除外できます。イメージ構築のベストプラクティスについては、こちらを参照してください。

もう 1 つの考慮事項は、キャッシュされたイメージを保持する期間です。一定量のディスクが使用されているときに、古いイメージをイメージキャッシュからクリーンアップできます。これにより、ホストのオペレーションに十分なスペースを確保できます。デフォルトでは、kubelet は未使用のイメージのガベージコレクションを 5 分ごとに、未使用のコンテナのガベージコレクションを 1 分ごとに実行します。

未使用のコンテナとイメージガベージコレクションのオプションを設定するには、設定ファイルを使用して kubelet を調整し、*KubeletConfiguration*リソースタイプを使用してガベージコレクションに関連するパラメータを変更します。

詳細については、Kubernetes ドキュメントを参照してください。

オブザーバビリティ

序章

オブザーバビリティツールは、ワークロードを効率的に検出、修復、調査するのに役立ちます。テレメトリデータのコストは、EKS の使用が増えるにつれて自然に増加します。運用上のニーズのバランスを取り、ビジネスにとって何が重要かを測定し、オブザーバビリティコストを抑えるのは難しい場合があります。このガイドでは、オブザーバビリティの 3 つの柱であるログ、メトリクス、トレースのコスト最適化戦略に焦点を当てています。これらのベストプラクティスはそれぞれ、組織の最適化目標に合わせて個別に適用できます。

ロギング

ログ記録は、クラスター内のアプリケーションのモニタリングとトラブルシューティングに重要な役割を果たします。ログ記録コストを最適化するために使用できる戦略はいくつかあります。以下に示すベストプラクティス戦略には、ログ保持ポリシーを調べてログデータを保持する期間をきめ細かく制御すること、重要度に基づいてログデータをさまざまなストレージオプションに送信すること、ログフィルタリングを使用して保存されるログメッセージのタイプを絞り込むことなどがあります。ログテレメトリを効率的に管理することで、環境のコスト削減につながります。

EKS コントロールプレーン

コントロールプレーンログの最適化

Kubernetes コントロールプレーンは、クラスターを管理するコンポーネントのセットであり、これらのコンポーネントはさまざまなタイプの情報をログストリームとして Amazon CloudWatch のロググループに送信します。すべてのコントロールプレーンログタイプを有効にするとメリットがありますが、各ログの情報と、すべてのログテレメトリを保存するための関連コストに注意する必要があります。クラスターから Amazon CloudWatch Logs に送信されたログの標準の CloudWatch Logs データインジェストとストレージコストに対して課金されます。 Amazon CloudWatch 有効にする前に、各ログストリームが必要かどうかを評価します。

例えば、本番稼働用以外のクラスターでは、API サーバーログなどの特定のログタイプを分析のためにのみ選択的に有効にし、後で無効にします。ただし、イベントを再現できず、問題を解決するにはより多くのログ情報が必要な本番稼働用クラスターでは、すべてのログタイプを有効にできます。コントロールプレーンのコスト最適化の実装の詳細については、このブログ記事を参照してください。

S3 にログをストリーミングする

もう 1 つのコスト最適化のベストプラクティスは、CloudWatch Logs サブスクリプションを介してコントロールプレーンログを S3 にストリーミングすることです。CloudWatch Logs サブスクリプションを使用すると、ログを S3 に選択的に転送できるため、CloudWatch でログを無期限に保持するよりもコスト効率の高い長期ストレージが提供されます。例えば、本番クラスターの場合、重要なロググループを作成し、サブスクリプションを活用して 15 日後にこれらのログを S3 にストリーミングできます。これにより、分析のために口グにすばやくアクセスできますが、ログをよりコスト効率の高いストレージに移動することでコストを削減できます。

> **Important**
>
> 9/5/2023 の時点で、EKS ログは Amazon CloudWatch Logs で Vended Logs として分類されます。提供されたログは、AWS のサービスが顧客に代わってネイティブに発行する特定の AWS サービスログで、ボリューム割引料金で利用できます。Vended Logs の料金の詳細については、Amazon CloudWatch の料金ページを参照してください。

EKS データプレーン

ログの保持

Amazon CloudWatch のデフォルトの保持ポリシーでは、ログを無期限に保持し、有効期限が切れないようにすることで、AWS リージョンに適用されるストレージコストが発生します。ストレージコストを削減するために、ワークロードの要件に基づいて各ロググループの保持ポリシーをカスタマイズできます。

開発環境では、保持期間を長くする必要がない場合があります。ただし、本番環境では、トラブルシューティング、コンプライアンス、キャパシティプランニングの要件を満たすために、より長い保持ポリシーを設定できます。例えば、ピークの祝祭日シーズンに e コマースアプリケーションを実行していて、システムが負荷が高くなり、すぐには気付かないような問題が発生する可能性がある場合は、詳細なトラブルシューティングとイベント後分析のために、より長いログ保持期間を設定する必要があります。

AWS CloudWatch コンソールまたは AWS API で、各ロググループに基づいて 1 日から 10 年の保持期間を設定できます。保持期間を柔軟に設定することで、重要なログを維持しながら、ログのストレージコストを節約できます。

ログストレージオプション

ストレージはオブザーバビリティコストの大きな要因であるため、ログストレージ戦略を最適化することが重要です。戦略は、パフォーマンスとスケーラビリティを維持しながら、ワークロードの要件と一致する必要があります。ログの保存コストを削減する 1 つの戦略は、AWS S3 バケットとそのさまざまなストレージ階層を活用することです。

S3 に直接ログを転送する

開発環境など、重要度の低いログは、Cloudwatch ではなく S3 に直接転送することを検討してください。これにより、ログストレージコストにすぐに影響する可能性があります。1 つのオプションは、Fluentbit を使用してログを S3 に直接転送することです。これは、FluentBit が保持のためにコンテナログを送信する送信先である [OUTPUT]セクションで定義します。追加の設定パラメータについては、こちらを参照してください。

```
[OUTPUT]
        Name eks_to_s3
        Match application.*
        bucket $S3_BUCKET name
        region us-east-2
        store_dir /var/log/fluentbit
        total_file_size 30M
        upload_timeout 3m
```

短期分析のためにのみ CloudWatch にログを転送する

データの即時分析が必要な本番環境など、より重要なログについては、ログを CloudWatch に転送することを検討してください。これは、FluentBit が保持のためにコンテナログを送信する送信先である [OUTPUT]セクションで定義します。追加の設定パラメータについては、こちらを参照してください。

```
[OUTPUT]
        Name eks_to_cloudwatch_logs
        Match application.*
        region us-east-2
        log_group_name fluent-bit-cloudwatch
        log_stream_prefix from-fluent-bit-
        auto_create_group On
```

ただし、これはコスト削減にすぐには影響しません。さらに節約するには、これらのログを Amazon S3 にエクスポートする必要があります。

CloudWatch から Amazon S3 にエクスポートする

Amazon CloudWatch logsを長期間保存する場合は、Amazon EKS CloudWatch ログを Amazon Simple Storage Service (Amazon S3) にエクスポートすることをお勧めします。コンソールまたは API を使用してエクスポートタスクを作成することで、ログを Amazon S3 バケットに転送できます。これを行うと、Amazon S3 はコストをさらに削減するための多くのオプションを提供します。独自の Amazon S3 ライフサイクルルールを定義して、ニーズに合ったストレージクラスにログを移動したり、Amazon S3 Intelligent-Tiering ストレージクラスを活用して、AWS が使用パターンに基づいてデータを長期ストレージに自動的に移動させることができます。詳細については、このブログを参照してください。例えば、本番環境のログは CloudWatch に 30 日以上保存され、Amazon S3 バケットにエクスポートされます。その後、後でログを参照する必要がある場合は、Amazon Athena を使用して Amazon S3 バケット内のデータをクエリできます。

ログレベルを減らす

アプリケーションの選択的ログ記録を練習します。アプリケーションとノードの両方がデフォルトでログを出力します。アプリケーションログについては、ワークロードと環境の重要性に合わせてログレベルを調整します。例えば、以下の Java アプリケーションは、一般的なデフォルトのアプリケーション設定であるINFOログを出力しており、コードによっては大量のログデータが発生する可能性があります。

```
import org.apache.log4j.*;

public class LogClass {
   private static org.apache.log4j.Logger log = Logger.getLogger(LogClass.class);

public static void main(String[] args) {
      log.setLevel(Level.INFO);

   log.debug("This is a DEBUG message, check this out!");
   log.info("This is an INFO message, nothing to see here!");
   log.warn("This is a WARN message, investigate this!");
   log.error("This is an ERROR message, check this out!");
   log.fatal("This is a FATAL message, investigate this!");    } }
```

開発環境では、ログレベルを に変更します。これによりDEBUG、問題をデバッグしたり、本番環境に入る前に潜在的な問題を捕捉したりすることができます。

```
      log.setLevel(Level.DEBUG);
```

本番環境では、ログレベルを ERRORまたは に変更することを検討してくださいFATAL。これにより、アプリケーションにエラーがある場合にのみログが出力され、ログ出力が減少し、アプリケーションのステータスに関する重要なデータに集中するのに役立ちます。

```
      log.setLevel(Level.ERROR);
```

さまざまな Kubernetes コンポーネントのログレベルを微調整できます。たとえば、EKS Node オペレーティングシステムとして Bottlerocket を使用している場合、kubelet プロセスログレベルを調整できる構成設定があります。この設定のスニペットを以下に示します。kubelet プロセスのログ詳細度を調整するデフォルトのログレベル 2 に注意してください。

```
[settings.kubernetes]
log-level = "2"
image-gc-high-threshold-percent = "85"
image-gc-low-threshold-percent = "80"
```

開発環境では、追加のイベントを表示するためにログレベルを 2 より大きく設定できます。これはデバッグに適しています。本番環境では、レベルを 0 に設定して、重要なイベントのみを表示できます。

フィルターの活用

デフォルトの EKS Fluentbit 設定を使用して Cloudwatch にコンテナログを送信する場合、FluentBit は、以下の[INPUT]設定ブロックに示すように、Kubernetes メタデータで強化されたすべてのアプリケーションコンテナログをキャプチャして Cloudwatch に送信します。

```
 [INPUT]
     Name                tail
     Tag                 application.*
     Exclude_Path        /var/log/containers/cloudwatch-agent*, /var/log/containers/
fluent-bit*, /var/log/containers/aws-node*, /var/log/containers/kube-proxy*
     Path                /var/log/containers/*.log
     Docker_Mode         On
     Docker_Mode_Flush   5
     Docker_Mode_Parser  container_firstline
     Parser              docker
     DB                  /var/fluent-bit/state/flb_container.db
     Mem_Buf_Limit       50MB
     Skip_Long_Lines     On
     Refresh_Interval    10
```

```
    Rotate_Wait         30
    storage.type        filesystem
    Read_from_Head      ${READ_FROM_HEAD}
```

上記の[INPUT]セクションでは、すべてのコンテナログを取り込んでいます。これにより、必要のない大量のデータが生成されます。このデータをフィルタリングすると、CloudWatch に送信されるログデータの量を減らすことができるため、コストを削減できます。CloudWatch に出力する前に、ログにフィルターを適用できます。Fluentbit は、 [FILTER]セクションでこれを定義します。例えば、Kubernetes メタデータがログイベントに追加されないようにフィルタリングすると、ログボリュームを減らすことができます。

```
[FILTER]
    Name                nest
    Match               application.*
    Operation           lift
    Nested_under        kubernetes
    Add_prefix          Kube.

[FILTER]
    Name                modify
    Match               application.*
    Remove              Kube.<Metadata_1>
    Remove              Kube.<Metadata_2>
    Remove              Kube.<Metadata_3>

[FILTER]
    Name                nest
    Match               application.*
    Operation           nest
    Wildcard            Kube.*
    Nested_under        kubernetes
    Remove_prefix       Kube.
```

メトリクス

メトリクスは、システムのパフォーマンスに関する貴重な情報を提供します。システム関連または利用可能なすべてのリソースメトリクスを一元的な場所に統合することで、パフォーマンスデータを比較および分析できます。この一元化されたアプローチにより、リソースのスケールアップやスケールダウンなど、より多くの情報に基づいた戦略的意思決定を行うことができます。さらに、 メトリクスはリソースの正常性を評価する上で重要な役割を果たし、必要に応じてプロアクティブに対策を講じることができます。一般的に、オブザーバビリティのコストは、テレメトリデータの収集と保持

に応じてスケールされます。以下は、メトリクステレメトリのコストを削減するために実装できるいくつかの戦略です。重要なメトリクスのみを収集し、テレメトリデータの基数を減らし、テレメトリデータ収集の詳細度を微調整します。

重要なものをモニタリングし、必要なもののみを収集する

最初のコスト削減戦略は、収集するメトリクスの数を減らし、保持コストを削減することです。

1. まず、自分やステークホルダーの要件から逆算して、最も重要なメトリクスを決定します。成功メトリクスは、すべてのユーザーで異なります。良いものがどのように見えるかを把握し、それを測定します。
2. サポートしているワークロードを深く掘り下げ、その主要業績評価指標 (KPIs「ゴールデンシグナル」を特定することを検討してください。これらは、ビジネス要件とステークホルダー要件と一致する必要があります。Amazon CloudWatch と Metric Math を使用して SLIs、SLOs、SLAs を計算することは、サービスの信頼性を管理するために不可欠です。このガイドで説明されているベストプラクティスに従って、EKS 環境のパフォーマンスを効果的にモニタリングおよび維持します。
3. 次に、さまざまなインフラストラクチャレイヤーを続行して、EKS クラスター、ノード、追加のインフラストラクチャメトリクスをワークロード KPIs に接続して関連付けます。ビジネスメトリクスと運用メトリクスを相互に関連付けて、観察された影響に基づいて結論を導き出すことができるシステムに保存します。
4. EKS は、コントロールプレーン、クラスター kube-state-metrics、ポッド、ノードからのメトリクスを公開します。これらのすべてのメトリクスの関連性はニーズによって異なりますが、異なるレイヤーにまたがるすべてのメトリクスは必要ない可能性があります。この EKS 必須メトリクスガイドは、EKS クラスターとワークロードの全体的な状態をモニタリングするためのベースラインとして使用できます。

kubelet メトリクスのみを保持し、すべてのコンテナメトリクスを削除`relabel_config`するためにを使用する prometheus スクレイプ設定の例`metric_relabel_config`を次に示します。

```
kubernetes_sd_configs:
- role: endpoints
  namespaces:
    names:
    - kube-system
bearer_token_file: /var/run/secrets/kubernetes.io/serviceaccount/token
tls_config:
  insecure_skip_verify: true
```

```
  relabel_configs:
  - source_labels: [__meta_kubernetes_service_label_k8s_app]
    regex: kubelet
    action: keep

  metric_relabel_configs:
  - source_labels: [__name__]
    regex: container_(network_tcp_usage_total|network_udp_usage_total|tasks_state|
cpu_load_average_10s)
    action: drop
```

必要に応じて濃度を下げる

カーディナリティとは、特定のメトリクスセットのディメンション (prometheus ラベルなど) と組み合わせたデータ値の一意性を指します。高基数メトリクスには多くのディメンションがあり、各ディメンションメトリクスの組み合わせの一意性は高くなります。カーディナリティが高いほど、メトリクステレメトリのデータサイズとストレージのニーズが大きくなり、コストが増加します。

以下の高基数例では、メトリクス、レイテンシー、ディメンション、RequestID、CustomerID、サービスがあり、各ディメンションに多くの一意の値があることがわかります。カーディナリティは、ディメンションあたりの可能な値の数の組み合わせの尺度です。Prometheus では、一意のディメンション/ラベルの各セットが新しいメトリクスと見なされるため、カーディナリティが高いほどメトリクスが多くなります。

メトリクスごとに多くのメトリクスとディメンション/ラベル (クラスター、名前空間、サービス、ポッド、コンテナなど) を持つ EKS 環境では、基数が増加する傾向があります。コストを最適化するには、収集するメトリクスの基数を慎重に検討してください。例えば、クラスターレベルで視覚化する特定のメトリクスを集約する場合、名前空間ラベルなどの下位レイヤーにある追加のラベルを削除できます。

prometheus で高基数メトリクスを識別するには、次の PROMQL クエリを実行して、メトリクス (基数) の数が最も多いスクレイプターゲットを判断できます。

```
topk_max(5, max_over_time(scrape_samples_scraped[1h]))
```

および次の PROMQL クエリは、どのスクレイプターゲットが最も高いメトリクスチャーン (特定のスクレイプで作成された新しいメトリクスシリーズの数) 率を持つかを判断するのに役立ちます。

```
topk_max(5, max_over_time(scrape_series_added[1h]))
```

grafana を使用している場合は、Grafana Lab の Mimirtool を使用して grafana ダッシュボードと prometheus ルールを分析し、未使用の高濃度メトリクスを特定できます。ダッシュボードで参照されていないアクティブなメトリクスを特定するには、`mimirtool analyze`および `mimirtool analyze prometheus` コマンドを使用する方法に関するこのガイドに従ってください。

メトリクスの詳細度を考慮する

毎秒のように、毎分よりも細かくメトリクスを収集すると、テレメトリを収集して保存する量に大きな影響を与え、コストが増加する可能性があります。一時的な問題を確認するのに十分な詳細度と、費用対効果を高めるために十分な低度のバランスをとる、適切なスクレイプまたはメトリクス収集間隔を決定します。キャパシティプランニングとより大きな時間枠分析に使用されるメトリクスの詳細度を下げます。

以下は、デフォルトの AWS Distro for Opentelemetry (ADOT) EKS Addon Collector 設定のスニペットです。

> **Important**
>
> グローバル prometheus スクレイプ間隔は 15 秒に設定されています。このスクレイプ間隔を長くすると、prometheus で収集されるメトリクスデータの量が減少します。

```
apiVersion: opentelemetry.io/v1alpha1
kind: OpenTelemetryCollector
metadata:
  name: my-collector-amp

...

config: |
    extensions:
      sigv4auth:
        region: "+++<YOUR_AWS_REGION>+++" service: "aps"+++</YOUR_AWS_REGION>+++

 receivers:
   #
   # Scrape configuration for the Prometheus Receiver
   # This is the same configuration used when Prometheus is installed using the
 community Helm chart
   #
   prometheus:
```

```
    config:
      global:   scrape_interval: 15s
        scrape_timeout: 10s
```

トレース

トレースに関連する主なコストは、トレースストレージの生成によるものです。トレースでは、パフォーマンスの側面を診断して理解するための十分なデータを収集することが目的です。ただし、X-Ray トレースのコストは X-Ray に転送されたデータに基づいているため、転送後にトレースを消去してもコストは削減されません。適切な分析を実行するためにデータを維持しながら、トレースのコストを削減する方法を確認しましょう。

サンプリングルールを適用する

X-Ray のサンプリングレートはデフォルトで控えめです。収集するデータ量を制御できるサンプリングルールを定義します。これにより、コストを削減しながらパフォーマンス効率が向上します。サンプリングレートを減らすことで、より低いコスト構造を維持しながら、ワークロードに必要なものだけをリクエストからトレースを収集できます。

例えば、問題のある 1 つのルートに対するすべてのリクエストのトレースをデバッグする Java アプリケーションがあるとします。

JSON ドキュメントからサンプリングルールをロードするように SDK 経由で を設定する

```
{
"version": 2,
  "rules": [
    {
"description": "debug-eks",
      "host": "*",
      "http_method": "PUT",
      "url_path": "/history/*",
      "fixed_target": 0,
      "rate": 1,
      "service_type": "debug-eks"
    }
  ],
  "default": {
"fixed_target": 1,
    "rate": 0.1
  }
```

```
}
```

コンソール経由

AWS Distro for OpenTelemetry (ADOT) によるテールサンプリングの適用

ADOT Tail Sampling を使用すると、サービスに取り込まれたトレースのボリュームを制御できます。ただし、テールサンプリングでは、リクエスト内のすべてのスパンが完了した後に、最初ではなくサンプリングポリシーを定義できます。これにより、CloudWatch に転送される raw データの量がさらに制限されるため、コストが削減されます。

例えば、ランディングページへのトラフィックの 1% と支払いページへのリクエストの 10% をサンプリングする場合、30 分間に 300 トレースが残る可能性があります。特定のエラーをフィルタリングする の ADOT Tail Sampling ルールでは、200 個のトレースを残して保存されるトレースの数を減らすことができます。

```
processors:
  groupbytrace:
    wait_duration: 10s
    num_traces: 300
    tail_sampling:
    decision_wait: 1s # This value should be smaller than wait_duration
    policies:
      - ..... # Applicable policies**
  batch/tracesampling:
    timeout: 0s # No need to wait more since this will happen in previous processors
    send_batch_max_size: 8196 # This will still allow us to limit the size of the
 batches sent to subsequent exporters

service:
  pipelines:
    traces/tailsampling:
      receivers: [otlp]
      processors: [groupbytrace, tail_sampling, batch/tracesampling]
      exporters: [awsxray]
```

Amazon S3 ストレージオプションを活用する

トレースを保存するには、AWS S3 バケットとそのさまざまなストレージクラスを活用する必要があります。保持期間が終了する前にトレースを S3 にエクスポートします。Amazon S3 ライフサイクルルールを使用して、要件を満たすストレージクラスにトレースデータを移動します。

例えば、90 日経過のトレースがある場合、Amazon S3 Intelligent-Tiering は使用パターンに基づいてデータを長期ストレージに自動的に移動できます。後でトレースを参照する必要がある場合は、Amazon Athena を使用して Amazon S3 内のデータをクエリできます。これにより、分散トレースのコストをさらに削減できます。

その他のリソース：

- オブザーバビリティのベストプラクティスガイド
- ベストプラクティスメトリクスの収集
- AWS re:Invent 2022 - Amazon でのオブザーバビリティのベストプラクティス (COP343)
- AWS re:Invent 2022 - Observability: Best practices for modern applications (COP344)

Windows のベストプラクティス

このガイドでは、Windows コンテナとノードの実行に関するアドバイスを提供します。

トピック

- Amazon EKS 最適化 Windows AMI 管理
- Windows Pod とコンテナの gMSA を設定する
- Windows ワーカーノードの強化
- コンテナイメージスキャン
- Windows Server のバージョンとライセンス
- ログ記録
- モニタリング
- Windows ネットワーク
- OOM エラーの回避
- Windows サーバーとコンテナへのパッチ適用
- 異種ワークロードの実行
- ポッドのセキュリティコンテキスト
- 永続的ストレージオプション
- Windows コンテナイメージの強化

Amazon EKS 最適化 Windows AMI 管理

Windows Amazon EKS 最適化 AMI は Windows Server 2019 と Windows Server 2022 上に構築されています。Amazon EKS ノードのベースイメージとして機能するように構成されています。デフォルトでは、AMI には次のコンポーネントが含まれています。

- kubelet
- kube-proxy
- AWS IAM Authenticator for Kubernetes
- csi-proxy
- containerd

AWS Systems Manager Parameter Store API をクエリすることで、Amazon EKS 最適化 AMI の Amazon マシンイメージ (AMI) ID をプログラム的に取得できます。この API から提供されるパラメータにより、Amazon EKS 最適化 AMI ID を手動で検索する必要がなくなります。Systems Manager Parameter Store API の詳細については、GetParameter を参照してください。ユーザーアカウントには、Amazon EKS 最適化 AMI メタデータを取得するための ssm:GetParameter IAM アクセス許可が必要です。

次の の例では、Windows Server 2019 LTSC Core 用の最新の Amazon EKS 最適化 AMI の AMI ID を取得します。AMI 名にリストされているバージョン番号は、準備されている対応する Kubernetes ビルドに関連しています。

```
aws ssm get-parameter --name /aws/service/ami-windows-latest/Windows_Server-2019-
English-Core-EKS_Optimized-1.21/image_id --region us-east-1 --query "Parameter.Value"
 --output text
```

出力例:

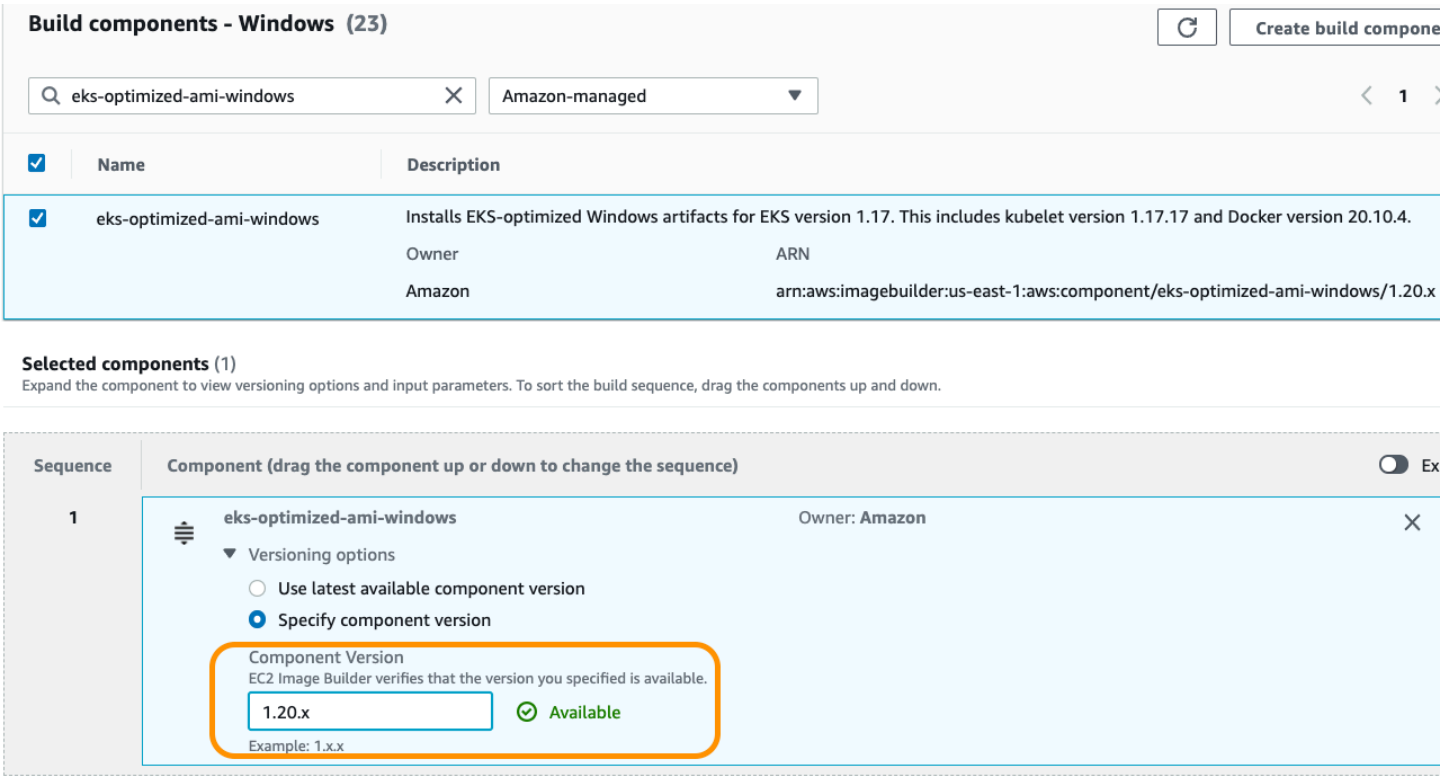
```
ami-09770b3eec4552d4e
```

独自の Amazon EKS 最適化 Windows AMI の管理

本番環境に不可欠なステップは、Amazon EKS クラスター全体で同じ Amazon EKS 最適化 Windows AMI と kubelet バージョンを維持することです。

Amazon EKS クラスター全体で同じバージョンを使用すると、トラブルシューティングにかかる時間が短縮され、クラスターの一貫性が向上します。Amazon EC2 Image Builder は、Amazon EKS クラスター全体で使用するカスタム Amazon EKS 最適化 Windows AMIs の作成と維持に役立ちます。

Amazon EC2 Image Builder を使用して、Windows Server バージョン、AWS Windows Server AMI リリース日、OS ビルドバージョンのいずれかを選択します。ビルドコンポーネントのステップでは、既存の EKS 最適化 Windows アーティファクトと kubelet バージョンを選択できます。詳細については、https://docs.aws.amazon.com/eks/latest/userguide/eks-custom-ami-windows.html を参照してください。



注: ベースイメージを選択する前に、リリースチャネルの更新に関する重要な詳細については、Windows Server のバージョンとライセンスのセクションを参照してください。

カスタム EKS 最適化 AMIs の高速起動の設定

カスタム Windows Amazon EKS 最適化 AMI を使用する場合、高速起動機能を有効にすることで、Windows ワーカーノードの起動を最大 65% 高速化できます。この機能は、Sysprep 専門分野、Windows Out of Box Experience (OOBE) ステップ、必要な再起動がすでに完了している事前プロビジョニングされたスナップショットのセットを維持します。これらのスナップショットは後続の起動で使用され、スケールアウトまたはノードの置き換えにかかる時間が短縮されます。高速起動は、EC2 コンソールまたは AWS CLI で所有している AMIs に対してのみ有効にでき、維持されているスナップショットの数は設定可能です。

注: Fast Launch は、Amazon が提供するデフォルトの EKS 最適化 AMI と互換性がありません。有効にする前に、上記のようにカスタム AMI を作成します。

詳細については、AWS Windows AMIs - AMI をより高速に起動するように設定してください。

カスタム AMIs での Windows ベースレイヤーのキャッシュ

Windows コンテナイメージは Linux コンテナイメージよりも大きくなっています。コンテナ化された .NET Framework ベースのアプリケーションを実行している場合、平均イメージサイズは約 8.24 GB です。ポッドのスケジューリング中、ポッドが実行中ステータスに達する前に、コンテナイメージを完全にプルしてディスクに抽出する必要があります。

このプロセス中、コンテナランタイム (コンテナ) はディスク内のコンテナイメージ全体をプルして抽出します。プルオペレーションは並列プロセスです。つまり、コンテナランタイムはコンテナイメージレイヤーを並列にプルします。対照的に、抽出オペレーションはシーケンシャルプロセスで行われ、I/O を大量に消費します。そのため、コンテナイメージが完全に抽出され、コンテナランタイム (コンテナ) で使用できるようになるまでに 8 分以上かかる場合があり、その結果、ポッドの起動に数分かかる場合があります。

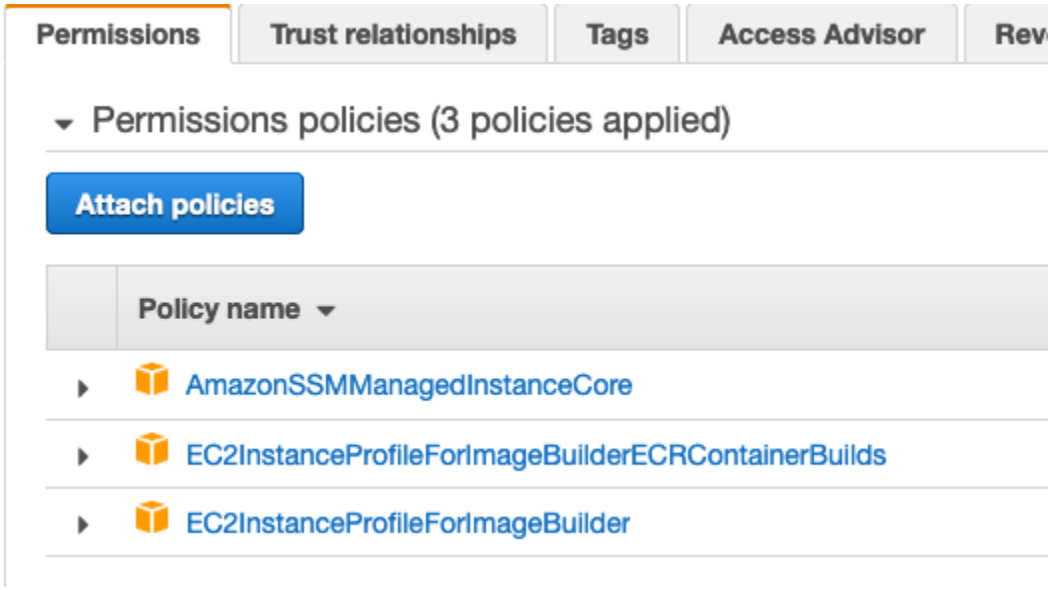
「Windows Server とコンテナのパッチ適用」トピックで説明したように、EKS でカスタム AMI を構築するオプションがあります。AMI の準備中に、EC2 Image Builder コンポーネントを追加して、必要なすべての Windows コンテナイメージをローカルにプルし、AMI を生成できます。この戦略により、ポッドが実行中のステータスに達する時間が大幅に短縮されます。

Amazon EC2 Image Builder で、コンポーネントを作成して必要なイメージをダウンロードし、Image recipe にアタッチします。次の の例では、ECR リポジトリから特定のイメージをプルします。

```
name: ContainerdPull
description: This component pulls the necessary containers images for a cache strategy.
schemaVersion: 1.0

phases:
  - name: build
    steps:
      - name: containerdpull
        action: ExecutePowerShell
        inputs:
          commands:
            - Set-ExecutionPolicy Unrestricted -Force
            - (Get-ECRLoginCommand).Password | docker login --username AWS --password-
stdin 111000111000.dkr.ecr.us-east-1.amazonaws.com
            - ctr image pull mcr.microsoft.com/dotnet/framework/aspnet:latest
            - ctr image pull 111000111000.dkr.ecr.us-east-1.amazonaws.com/
myappcontainerimage:latest
```

次のコンポーネントが正常に動作することを確認するには、EC2 Image Builder (EC2InstanceProfileForImageBuilder) で使用される IAM ロールにポリシーがアタッチされているかどうかを確認します。



ブログ記事

次のブログ記事では、カスタム Amazon EKS Windows AMIs のキャッシュ戦略を実装する方法をステップバイステップで説明します。

EC2 Image Builder とイメージキャッシュ戦略を使用して Windows コンテナの起動時間を短縮する

Windows Pod とコンテナの gMSA を設定する

gMSA アカウントとは

.NET アプリケーションなどの Windows ベースのアプリケーションは、多くの場合、Active Directory を ID プロバイダーとして使用し、NTLM または Kerberos プロトコルを使用した認可/認証を提供します。

Kerberos チケットを Active Directory と交換するアプリケーションサーバーは、ドメインに参加している必要があります。Windows コンテナはドメイン結合をサポートしておらず、コンテナがエフェメラルリソースであるためあまり意味がないため、Active Directory RID プールに負担がかかります。

ただし、管理者は gMSA Active Directory アカウントを活用して、Windows コンテナ、NLB、サーバーファームなどのリソースの Windows 認証をネゴシエートできます。

Windows コンテナと gMSA のユースケース

Windows 認証を活用し、Windows コンテナとして実行されるアプリケーションは、Windows ノードがコンテナに代わって Kerberos チケットの交換に使用されるため、gMSA の恩恵を受けます。gMSA 統合をサポートするように Windows ワーカーノードをセットアップするには、次の 2 つのオプションがあります。

1 - ドメイン結合 Windows ワーカーノード

この設定では、Windows ワーカーノードは Active Directory ドメインにドメイン結合され、Windows ワーカーノードの AD コンピュータアカウントは Active Directory に対する認証と、ポッドで使用する gMSA ID の取得に使用されます。

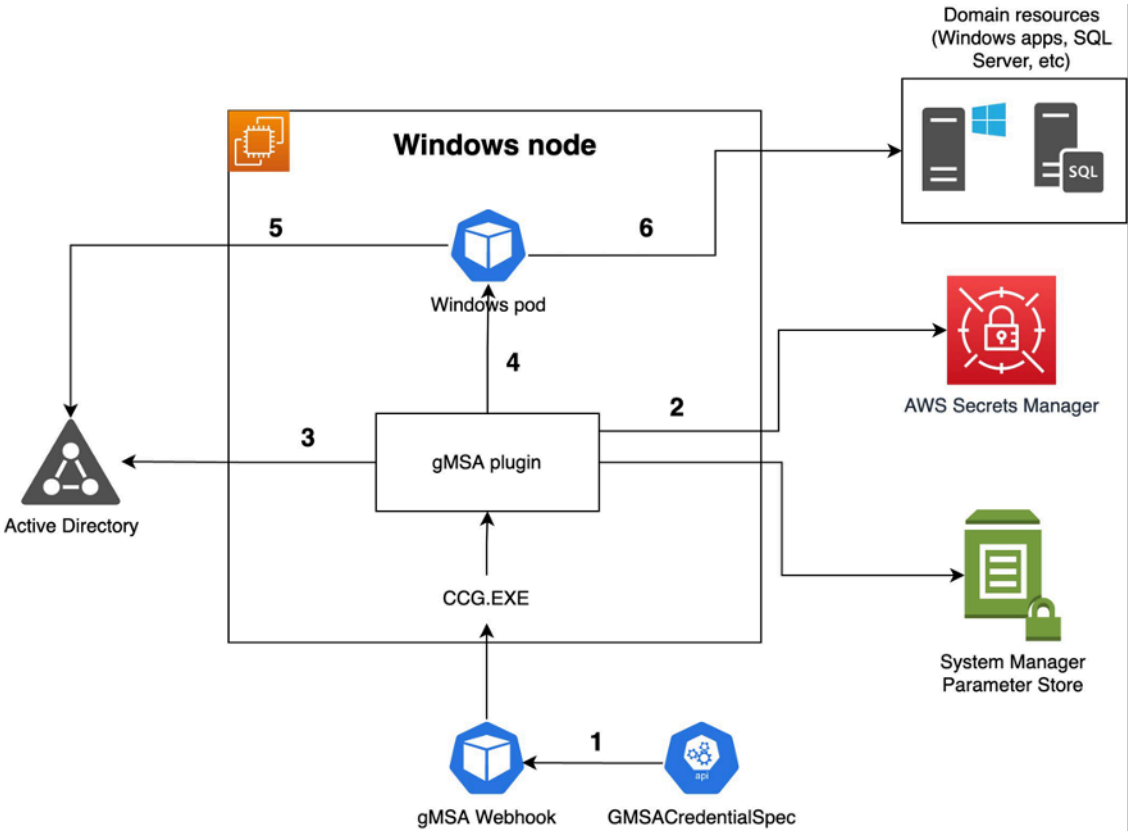
ドメイン結合アプローチでは、既存の Active Directory GPOs を使用して Windows ワーカーノードを簡単に管理および強化できます。ただし、Kubernetes クラスターで Windows ワーカーノードを結合すると、追加の運用オーバーヘッドと遅延が発生します。これは、ノードの起動時に追加の再起動が必要になり、Kubernetes クラスターがノードを終了した後に Active Directory ガレージをクリーンアップする必要があるためです。

次のブログ記事では、ドメイン結合 Windows ワーカーノードアプローチを実装する方法について詳しくstep-by-step。

Amazon EKS Windows ポッドでの Windows 認証

2 - ドメインレス Windows ワーカーノード

この設定では、Windows ワーカーノードは Active Directory ドメインに参加せず、「ポータブル」アイデンティティ (ユーザー/パスワード) を使用して Active Directory に対して認証し、ポッドで使用する gMSA アイデンティティを取得します。



ポータブル ID は Active Directory ユーザーです。ID (ユーザー/パスワード) は AWS Secrets Manager または AWS System Manager パラメータストアに保存され、ccg_plugin と呼ばれる AWS が開発したプラグインを使用して AWS Secrets Manager または AWS System Manager パラメータストアからこの ID を取得し、それを containerd に渡して gMSA ID を取得し、ポッドで使用できるようにします。

このドメインレスアプローチでは、gMSA を使用するときに Windows ワーカーノードの起動中に Active Directory とやり取りせず、Active Directory 管理者の運用オーバーヘッドを減らすことができます。

次のブログ記事では、Domainless Windows ワーカーノードアプローチを実装する方法の詳細な step-by-stepについて説明します。

Amazon EKS Windows ポッドのドメインレス Windows 認証

重要な注意事項

ポッドは gMSA アカウントを使用できますが、Windows 認証をサポートするように Microsoft IIS を設定するには、Dockerfile を使用してアプリケーションまたはサービスをセットアップする必要があります。

```
RUN Install-WindowsFeature -Name Web-Windows-Auth -IncludeAllSubFeature
RUN Import-Module WebAdministration; Set-ItemProperty 'IIS:\AppPools\SiteName' -name
 processModel.identityType -value 2
RUN Import-Module WebAdministration; Set-WebConfigurationProperty -Filter '/
system.webServer/security/authentication/anonymousAuthentication' -Name Enabled -Value
 False -PSPath 'IIS:\' -Location 'SiteName'
RUN Import-Module WebAdministration; Set-WebConfigurationProperty -Filter '/
system.webServer/security/authentication/windowsAuthentication' -Name Enabled -Value
 True -PSPath 'IIS:\' -Location 'SiteName'
```

Windows ワーカーノードの強化

OS 強化は、OS 設定、パッチ適用、不要なソフトウェアパッケージの削除を組み合わせたもので、システムをロックダウンし、攻撃対象領域を減らすことを目的としています。会社に必要な強化設定を使用して、独自の EKS 最適化 Windows AMI を準備するのがベストプラクティスです。

AWS は、最新の Windows Server セキュリティパッチを含む新しい EKS 最適化 Windows AMI を毎月提供します。ただし、セルフマネージド型ノードグループとマネージド型ノードグループのどちらを使用するかにかかわらず、必要な OS 設定を適用して AMI を強化するのはユーザーの責任です。

Microsoft は、セキュリティポリシーのニーズに基づいて強化を実現するのに役立つ Microsoft Security Compliance Toolkit や Security Baselines などのさまざまなツールを提供しています。CIS Benchmarks も利用可能であり、本番環境用の Amazon EKS Optimized Windows AMI 上に実装する必要があります。

Windows Server Core による攻撃対象領域の削減

Windows Server Core は、EKS 最適化 Windows AMI の一部として使用できる最小限のインストールオプションです。Windows Server Core をデプロイすると、いくつかの利点があります。まず、ディスクフットプリントが比較的小さく、Server Core では 6GB、デスクトップエクスペリエンスを備えた Windows Server では 10GB です。次に、コードベースと使用可能な APIs。

AWS は、Amazon EKS がサポートするバージョンに関係なく、最新の Microsoft セキュリティパッチを含む新しい Amazon EKS 最適化 Windows AMIs を毎月お客様に提供します。ベストプラクティスとして、Windows ワーカーノードは最新の Amazon EKS 最適化 AMI に基づいて新しいノードに置き換える必要があります。更新を実施せずに 45 日以上実行されているノードや、ノードの交換には、セキュリティのベストプラクティスがありません。

RDP 接続の回避

リモートデスクトッププロトコル (RDP) は、ネットワーク経由で別の Windows コンピュータに接続するためのグラフィカルインターフェイスをユーザーに提供するために Microsoft によって開発された接続プロトコルです。

ベストプラクティスとして、Windows ワーカーノードはエフェメラルホストであるかのように扱う必要があります。つまり、管理接続、更新、トラブルシューティングはありません。変更と更新は新しいカスタム AMI として実装し、Auto Scaling グループの更新に置き換える必要があります。「Windows サーバーとコンテナへのパッチ適用」および「Amazon EKS 最適化 Windows AMI 管理」を参照してください。

次の例のように、ssh プロパティで false 値を渡して、デプロイ中に Windows ノードの RDP 接続を無効にします。

```
nodeGroups:
- name: windows-ng
  instanceType: c5.xlarge
  minSize: 1
  volumeSize: 50
  amiFamily: WindowsServer2019CoreContainer
  ssh:
    allow: false
```

Windows ノードへのアクセスが必要な場合は、AWS System Manager Session Manager を使用して、AWS コンソールと SSM エージェントを介して安全な PowerShell セッションを確立します。ソリューションを実装する方法については、AWS Systems Manager Session Manager を使用して Windows インスタンスに安全にアクセスする方法をご覧ください。

System Manager Session Manager を使用するには、Windows ワーカーノードの起動に使用される IAM ロールに追加の IAM ポリシーを適用する必要があります。以下は、AmazonSSMManagedInstanceCore が`eksctl`クラスターマニフェストで指定されている例です。

```
 nodeGroups:
- name: windows-ng
  instanceType: c5.xlarge
  minSize: 1
  volumeSize: 50
  amiFamily: WindowsServer2019CoreContainer
```

```
  ssh:
    allow: false
  iam:
    attachPolicyARNs:
      - arn:aws:iam::aws:policy/AmazonEKSWorkerNodePolicy
      - arn:aws:iam::aws:policy/AmazonEKS_CNI_Policy
      - arn:aws:iam::aws:policy/ElasticLoadBalancingFullAccess
      - arn:aws:iam::aws:policy/AmazonEC2ContainerRegistryReadOnly
      - arn:aws:iam::aws:policy/AmazonSSMManagedInstanceCore
```

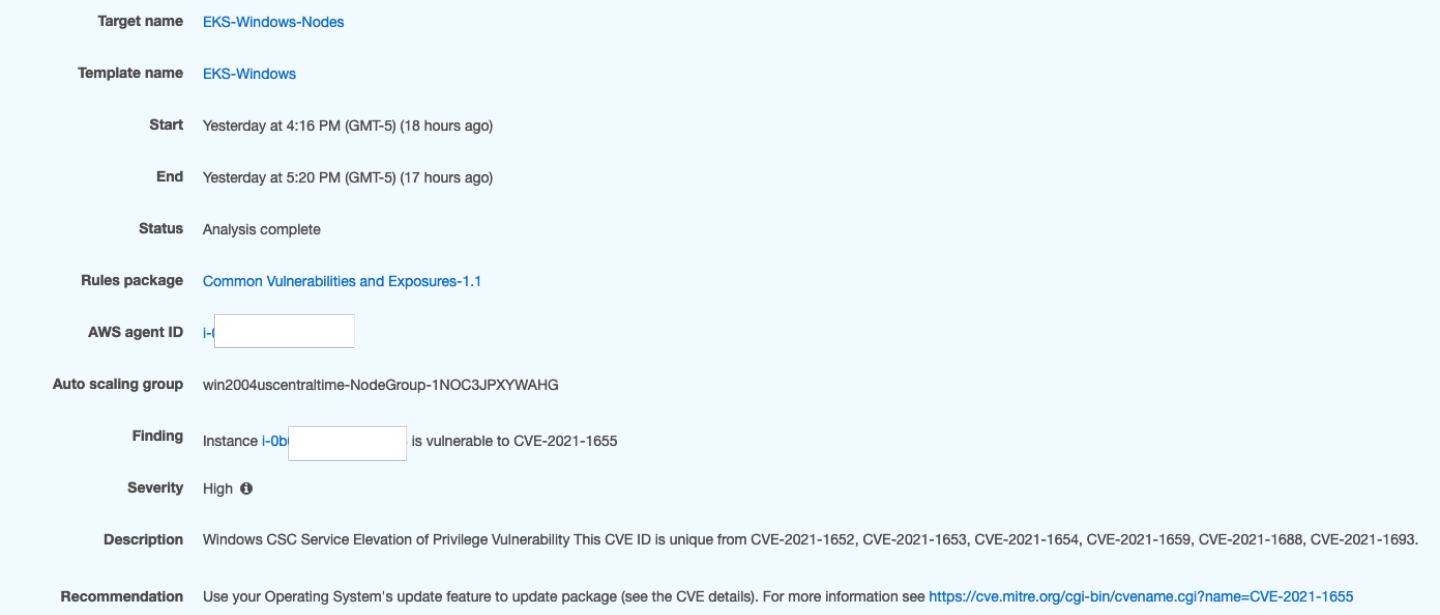
Amazon Inspector

> Amazon Inspector は、AWS にデプロイされたアプリケーションのセキュリティとコンプライアンスを向上させるのに役立つ自動セキュリティ評価サービスです。Amazon Inspector は、アプリケーションの露出、脆弱性、ベストプラクティスからの逸脱を自動的に評価します。評価を実行した後、Amazon Inspector は、重要度のレベルごとに優先順位が付けられたセキュリティ結果の詳細なリストを作成します。これらの検出結果は、直接レビューすることも、Amazon Inspector コンソールまたは API から入手できる詳細な評価レポートの一部としてレビューすることもできます。

Amazon Inspector は、Windows ワーカーノードで CIS ベンチマーク評価を実行するために使用でき、次のタスクを実行して Windows Server Core にインストールできます。

1. 次の .exe ファイルをダウンロードします。https://inspector-agent.amazonaws.com/windows/installer/latest/AWSAgentInstall.exe
2. エージェントを Windows ワーカーノードに転送します。
3. PowerShell で次のコマンドを実行して、Amazon Inspector エージェントをインストールします。
   `.\AWSAgentInstall.exe /install`

以下は、初回実行後の出力です。ご覧のとおり、CVE データベースに基づいて検出結果が生成されました。これを使用して、ワーカーノードを強化したり、強化された設定に基づいて AMI を作成したりできます。



Amazon Inspector エージェントのインストール方法、CIS ベンチマーク評価の設定方法、レポートの生成方法など、Amazon Inspector の詳細については、Amazon Inspector による Windows ワークロードのセキュリティとコンプライアンスの向上」ビデオを参照してください。

Amazon GuardDuty

> Amazon GuardDuty は、Amazon S3 に保存されている AWS アカウント、ワークロード、データを保護するために、悪意のあるアクティビティや不正な動作を継続的にモニタリングする脅威検出サービスです。クラウドを使用すると、アカウントとネットワークアクティビティの収集と集約が簡素化されますが、セキュリティチームが潜在的な脅威についてイベントログデータを継続的に分析するのに時間がかかる場合があります。

Amazon GuardDuty を使用すると、RDP ブルートフォースやポートプローブ攻撃など、Windows ワーカーノードに対する悪意のあるアクティビティを可視化できます。

Amazon GuardDuty を使用した Windows ワークロードの脅威検出ビデオで、最適化された EKS Windows AMI で CIS ベンチマークを実装および実行する方法について説明します。

Amazon EC2 for Windows のセキュリティ

Amazon EC2 Windows インスタンスのセキュリティのベストプラクティスを読み、すべてのレイヤーにセキュリティコントロールを実装します。

コンテナイメージスキャン

イメージスキャンは、さまざまなオペレーティングシステムの脆弱性をスキャンして、アプリケーションのコンテナイメージのセキュリティを向上させるのに役立つ自動脆弱性評価機能です。

現在、Amazon Elastic Container Registry (ECR) は Linux コンテナイメージの脆弱性のみをスキャンできます。ただし、Windows コンテナイメージスキャン用の既存の CI/CD パイプラインと統合できるサードパーティー製ツールがあります。

- アンカー
- PaloAlto Prisma クラウド
- Trend Micro - Deep Security スマートチェック

これらのソリューションを Amazon Elastic Container Repository (ECR) と統合する方法の詳細については、以下を確認してください。

- Amazon Elastic Container Registry (ECR) で画像をスキャンするアンカー
- PaloAlto、Amazon Elastic Container Registry (ECR) でイメージをスキャンする
- TrendMicro、Amazon Elastic Container Registry (ECR) でのイメージのスキャン

Windows Server のバージョンとライセンス

Windows サーバー バージョン

Amazon EKS 最適化 Windows AMI は、長期サービスチャネル (LTSC) の Windows Server 2019 および 2022 Datacenter Edition に基づいています。Datacenter バージョンには、ワーカーノードで実行されているコンテナの数に制限はありません。詳細については、https://docs.microsoft.com/en-us/virtualization/windowscontainers/about/faq を参照してください。

長期サービスチャネル (LTSC)

以前は「長期サービスブランチ」と呼ばれていましたが、これは既に使い慣れているリリースモデルであり、Windows Server の新しいメジャーバージョンが 2～3 年ごとにリリースされます。ユーザーは 5 年間のメインストリームサポートと 5 年間の延長サポートを受けることができます。

ライセンス

Windows Server ベースの AMI で Amazon EC2 インスタンスを起動する場合、Amazon はライセンスコストとライセンスコンプライアンスをカバーします。

ログ記録

コンテナ化されたアプリケーションは通常、アプリケーションログを STDOUT に送信します。コンテナランタイムはこれらのログをトラップし、何らかの操作を行います。通常、 は ファイルに書き込みます。これらのファイルが保存される場所は、コンテナのランタイムと設定によって異なります。

Windows ポッドの基本的な違いの 1 つは、STDOUT を生成しないことです。LogMonitor を実行して、実行中の Windows コンテナとパイプ形式のログ出力から STDOUT への ETW (Event Tracing for Windows)、Windows イベントログ、およびその他のアプリケーション固有のログを取得できます。これらのログは、fluent-bit を使用してストリーミングするか、Amazon CloudWatch などの目的の宛先に流暢にストリーミングできます。

ログ収集メカニズムは、Kubernetes ポッドから STDOUT/STDERR ログを取得します。DaemonSet は、コンテナからログを収集する一般的な方法です。これにより、アプリケーションとは独立してログのrouting/filtering/enrichmentを管理できます。流暢な DaemonSet を使用して、これらのログやその他のアプリケーション生成ログを目的のログアグリゲータにストリーミングできます。

Windows ワークロードから CloudWatch へのログストリーミングの詳細については、こちらを参照してください。

レコメンデーションのログ記録

Kubernetes で Windows ワークロードを運用する場合、一般的なログ記録のベストプラクティスに違いはありません。

- 構造化ログエントリ (JSON/SYSLOG) を常にログに記録します。これにより、このような構造化形式には事前に記述されたパーサーが多数あるため、ログエントリの処理が容易になります。
- ログを一元化する - 専用ログコンテナは、特にすべてのコンテナからログメッセージを収集して送信先に転送するために使用できます。
- デバッグ時を除いて、ログの詳細度を低く保ちます。冗長性により、ログ記録インフラストラクチャに大きなストレスがかかり、大きなイベントがノイズで失われる可能性があります。

- トレーサビリティのために、必ずトランザクション/リクエスト ID とともにアプリケーション情報を口グに記録します。Kubernetes オブジェクトにはアプリケーション名が含まれていないため、たとえば、口グをデバッグするときにポッド名に意味がないwindows-twryrqyw場合があります。これにより、集約口グを使用したアプリケーションのトレーサビリティとトラブルシューティングに役立ちます。

  これらのトランザクション/相関 ID の生成方法は、プログラミングコンストラクトによって異なります。ただし、非常に一般的なパターンは、口グ記録 Aspect/Interceptor を使用することです。これにより、MDC (マッピングされた診断コンテキスト) を使用して、次のようなすべての受信リクエストに一意のトランザクション/相関 ID を挿入できます。

```
import org.slf4j.MDC;
import java.util.UUID;
Class LoggingAspect { //interceptor

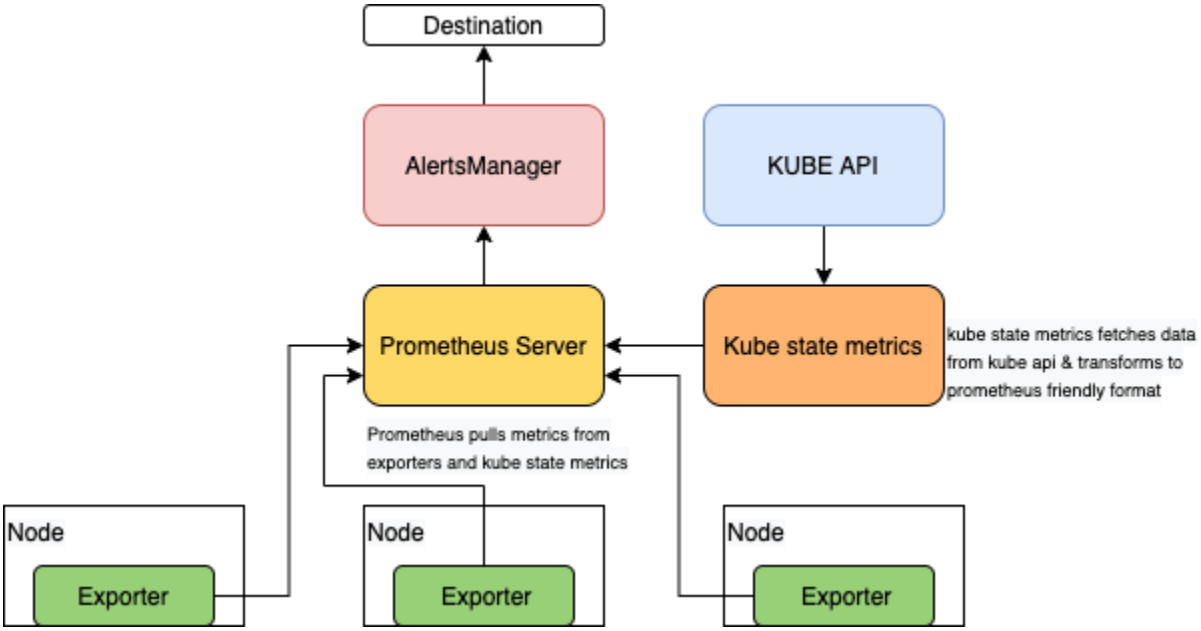
    @Before(value = "execution(* *.*(..))")
    func before(...) {
        transactionId = generateTransactionId();
        MDC.put(CORRELATION_ID, transactionId);
    }

    func generateTransactionId() {
        return UUID.randomUUID().toString();
    }
}
```

モニタリング

段階的な CNCF プロジェクトである Prometheus は、Kubernetes へのネイティブ統合を備えた最も人気のあるモニタリングシステムです。Prometheus は、コンテナ、ポッド、ノード、クラスターに関するメトリクスを収集します。さらに、Prometheus は AlertsManager を活用します。これにより、クラスター内の問題が発生した場合に警告するようにアラートをプログラムできます。Prometheus は、メトリクス名とキーと値のペアによって識別される時系列データとしてメトリクスデータを保存します。Prometheus には、Prometheus Query Language の略語である PromQL という言語を使用したクエリの実行が含まれています。

Prometheus メトリクスコレクションの大まかなアーキテクチャを以下に示します。



Prometheus はプルメカニズムを使用し、エクスポーターを使用してターゲットから、および kube 状態メトリクスを使用して Kubernetes API からメトリクスをスクレイプします。つまり、アプリケーションとサービスは Prometheus 形式のメトリクスを含む HTTP(S) エンドポイントを公開する必要があります。その後、Prometheus は設定に従って、これらの HTTP(S) エンドポイントから定期的にメトリクスを取得します。

エクスポーターを使用すると、サードパーティーのメトリクスを Prometheus 形式のメトリクスとして使用できます。Prometheus エクスポーターは通常、各ノードにデプロイされます。エクスポーターの完全なリストについては、Prometheus エクスポーターを参照してください。ノードエクスポーターは Linux ノードのホストハードウェアと OS メトリクスのエクスポートに適していますが、Windows ノードでは機能しません。

安定した Prometheus helm チャートを使用する場合、Windows ノードを持つ混合ノード EKS クラスターでは、このエクスポーターは Windows 向けではないため、Windows ノードに失敗したポッドが表示されます。Windows ワーカープールを個別に扱い、代わりに Windows ワーカーノードグループに Windows エクスポーターをインストールする必要があります。

Windows ノードの Prometheus モニタリングをセットアップするには、Windows サーバー自体に WMI エクスポーターをダウンロードしてインストールし、Prometheus 設定ファイルのスクレイプ設定内でターゲットを設定する必要があります。リリースページには、使用可能なすべての .msi インストーラと、それぞれの機能セットとバグ修正が表示されます。インストーラは Windows_exporter を Windows サービスとして設定し、Windows ファイアウォールに例外を作成します。インストーラがパラメータなしで実行されている場合、エクスポーターは有効なコレクター、ポートなどのデフォルト設定で実行されます。

このガイドのスケジューリングのベストプラクティスセクションでは、テイント/許容範囲または RuntimeClass を使用してノードエクスポーターを linux ノードにのみ選択的にデプロイし、Windows エクスポーターはノードのブートストラップ時または任意の設定管理ツール (Chef、Ansible、SSM など) を使用して Windows ノードにインストールすることを提案しています。

ノードエクスポーターがデーモンセット としてインストールされている Linux ノードとは異なり、Windows ノードでは WMI エクスポーターがホスト自体にインストールされていることに注意してください。エクスポーターは、CPU 使用率、メモリ、ディスク I/O 使用率などのメトリクスをエクスポートし、IIS サイトとアプリケーション、ネットワークインターフェイス、サービスのモニタリングにも使用できます。

Windows_exporter は、デフォルトで有効なコレクターのすべてのメトリクスを公開します。これは、エラーを回避するためにメトリクスを収集する推奨方法です。ただし、高度な使用のために、メトリクスをフィルタリングするコレクターのオプションのリストを windows_exporter に渡すことができます。これを行うには、Prometheus 設定の collect[] パラメータを使用します。

Windows のデフォルトのインストール手順には、フィルタリングするコレクターなどの引数を使用してブートストラッププロセス中にエクスポーターをサービスとしてダウンロードおよび開始することが含まれます。

```
> Powershell Invoke-WebRequest https://github.com/prometheus-community/
windows_exporter/releases/download/v0.13.0/windows_exporter-0.13.0-amd64.msi -OutFile
 <DOWNLOADPATH>

> msiexec /i <DOWNLOADPATH>
 ENABLED_COLLECTORS="cpu,cs,logical_disk,net,os,system,container,memory"
```

デフォルトでは、メトリクスはポート 9182 の /metrics エンドポイントでスクレイピングできます。この時点で、Prometheus は Prometheus 設定に次の scrape_config を追加することでメトリクスを消費できます。

```
scrape_configs:
    - job_name: "prometheus"
      static_configs:
        - targets: ['localhost:9090']
    ...
    - job_name: "wmi_exporter"
      scrape_interval: 10s
      static_configs:
```

```
        - targets: ['<windows-node1-ip>:9182', '<windows-node2-ip>:9182', ...]
```

Prometheus 設定は を使用して再ロードされます

```
> ps aux | grep prometheus
> kill HUP <PID>
```

ターゲットを追加するより適切で推奨される方法は、ServiceMonitor オブジェクトの定義を提供する ServiceMonitor というカスタムリソース定義と、定義した ServiceMonitors をアクティブ化し、必要な Prometheus 設定を自動的に構築するコントローラーを Prometheus 演算子の一部として] を使用することです。

Kubernetes サービスのグループをモニタリングする方法を宣言的に指定する ServiceMonitor は、Kubernetes 内からメトリクスをスクレイプするアプリケーションを定義するために使用されます。ServiceMonitor 内では、オペレータが Kubernetes サービスを識別するために使用できる Kubernetes ラベルを指定します。これにより、モニタリングするポッドが識別されます。

ServiceMonitor を活用するには、特定の Windows ターゲットを指すエンドポイントオブジェクト、ヘッドレスサービス、Windows ノード用の ServiceMontor を作成します。

```
apiVersion: v1
kind: Endpoints
metadata:
  labels:
    k8s-app: wmiexporter
  name: wmiexporter
  namespace: kube-system
subsets:
- addresses:
  - ip: NODE-ONE-IP
    targetRef:
      kind: Node
      name: NODE-ONE-NAME
  - ip: NODE-TWO-IP
    targetRef:
      kind: Node
      name: NODE-TWO-NAME
  - ip: NODE-THREE-IP
    targetRef:
      kind: Node
      name: NODE-THREE-NAME
  ports:
```

```
  - name: http-metrics
    port: 9182
    protocol: TCP

---
apiVersion: v1
kind: Service ##Headless Service
metadata:
  labels:
    k8s-app: wmiexporter
  name: wmiexporter
  namespace: kube-system
spec:
  clusterIP: None
  ports:
  - name: http-metrics
    port: 9182
    protocol: TCP
    targetPort: 9182
  sessionAffinity: None
  type: ClusterIP

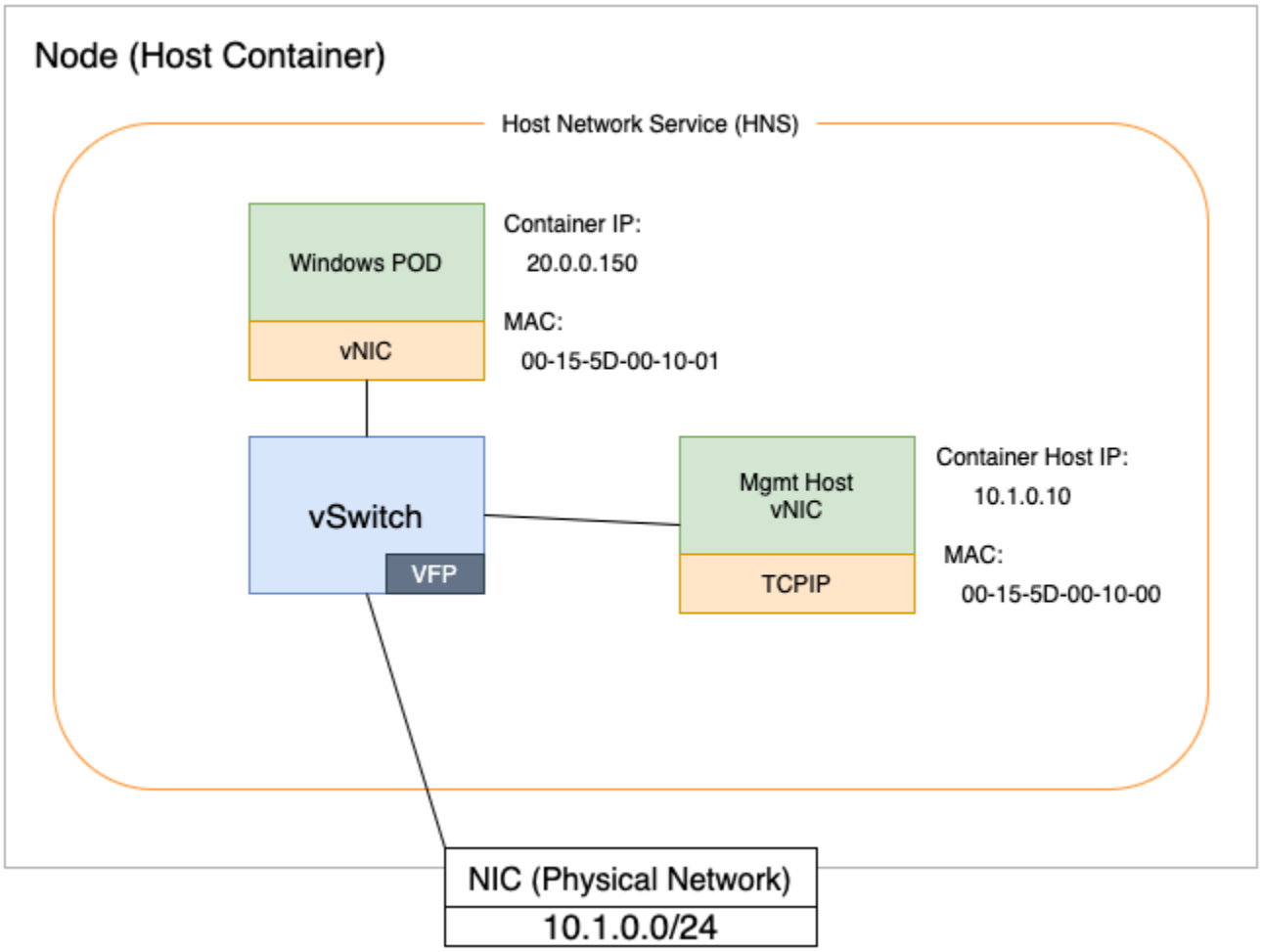
---
apiVersion: monitoring.coreos.com/v1
kind: ServiceMonitor ##Custom ServiceMonitor Object
metadata:
  labels:
    k8s-app: wmiexporter
  name: wmiexporter
  namespace: monitoring
spec:
  endpoints:
  - interval: 30s
    port: http-metrics
  jobLabel: k8s-app
  namespaceSelector:
    matchNames:
    - kube-system
  selector:
    matchLabels:
      k8s-app: wmiexporter
```

オペレーターと ServiceMonitor の使用の詳細については、オペレーターの公式ドキュメントを参照してください。Prometheus は、多くのサービス検出オプションを使用した動的ターゲット検出をサポートしていることに注意してください。

Windows ネットワーク

Windows コンテナネットワークの概要

Windows コンテナは Linux コンテナとは根本的に異なります。Linux コンテナは、名前空間、ユニオンファイルシステム、cgroup などの Linux コンストラクトを使用します。Windows では、これらのコンストラクトは Host Compute Service (HCS) によってコンテナから抽象化されます。HCS は、Windows でのコンテナ実装の上にある API レイヤーとして機能します。Windows コンテナは、ノードのネットワークトポロジを定義するホストネットワークサービス (HNS) も活用します。



ネットワークの観点からは、HCS と HNS は Windows コンテナを仮想マシンのように機能させます。たとえば、各コンテナには、上の図に示すように、Hyper-V 仮想スイッチ (vSwitch) に接続されている仮想ネットワークアダプタ (vNIC) があります。

IP アドレス管理

Amazon EKS のノードは、そのノードの Elastic Network Interface (ENI) を使用して AWS VPC ネットワークに接続します。現在、Windows ワーカーノードごとに 1 つの ENI のみがサポートされています。Windows ノードの IP アドレス管理は、コントロールプレーンで実行される VPC Resource Controller によって実行されます。Windows ノードの IP アドレス管理のワークフローの詳細については、こちらを参照してください。

Windows ワーカーノードがサポートできるポッドの数は、ノードのサイズと使用可能な IPv4 アドレスの数によって決まります。ノードで使用できる IPv4 アドレスは、次のように計算できます。

- デフォルトでは、セカンダリ IPv4 アドレスのみが ENI に割り当てられます。このような場合:

  ```
  Total IPv4 addresses available for Pods = Number of supported IPv4 addresses in the
   primary interface - 1
  ```

  1 つの IPv4 アドレスが ENI のプライマリアドレスとして使用されるため、ポッドに割り当てることができないため、合計数から 1 つを減算します。
- プレフィックス委任機能を有効にして、クラスターがポッド密度が高いように設定されている場合、

  ```
  Total IPv4 addresses available for Pods = (Number of supported IPv4 addresses in the
   primary interface - 1) * 16
  ```

  ここでは、セカンダリ IPv4 アドレスを割り当てる代わりに、VPC Resource Controller が割り当て/28 prefixesるため、使用可能な IPv4 アドレスの総数は 16 回ブーストされます。

上記の式を使用して、次のように m5.large インスタンスに基づいてノード化された Windows ワーカーの最大ポッド数を計算できます。

- デフォルトでは、セカンダリ IP モードで実行する場合 -

  ```
  10 secondary IPv4 addresses per ENI - 1 = 9 available IPv4 addresses
  ```
- を使用する場合 prefix delegation-

```
(10 secondary IPv4 addresses per ENI - 1) * 16 = 144 available IPv4 addresses
```

インスタンスタイプがサポートできる IP アドレスの数の詳細については、「インスタンスタイプごとのネットワークインターフェイスあたりの IP アドレス」を参照してください。

もう 1 つの重要な考慮事項は、ネットワークトラフィックの流れです。Windows では、100 を超えるサービスを持つノードでポートが枯渇するリスクがあります。この条件が発生すると、ノードは次のメッセージでエラーのスローを開始します。

「ポリシーの作成に失敗しました: hcnCreateLoadBalancer failed in Win32: The specified port already exists」

この問題に対処するために、Direct Server Return (DSR) を活用します。DSR は、非対称ネットワーク負荷分散の実装です。つまり、リクエストトラフィックとレスポンストラフィックは異なるネットワークパスを使用します。この機能はポッド間の通信を高速化し、ポートが枯渇するリスクを軽減します。したがって、Windows ノードで DSR を有効にすることをお勧めします。

Windows Server SAC EKS 最適化 AMIs。Windows Server 2019 LTSC EKS Optimized AMIs の場合、インスタンスのプロビジョニング中に、以下のスクリプトを使用し、eksctlnodeGroup で Windows Server 2019 Full または Core を amiFamily として使用して有効にする必要があります。詳細については、「eksctl custom AMI」を参照してください。

```
nodeGroups:
- name: windows-ng
  instanceType: c5.xlarge
  minSize: 1
  volumeSize: 50
  amiFamily: WindowsServer2019CoreContainer
  ssh:
    allow: false
```

Windows Server 2019 以降で DSR を使用するには、インスタンスの起動時に次の kube-proxy フラグを指定する必要があります。これを行うには、セルフマネージド型ノードグループの起動テンプレートに関連付けられたユーザーデータスクリプトを調整します。

```
<powershell>
[string]$EKSBinDir = "$env:ProgramFiles\Amazon\EKS"
```

```
[string]$EKSBootstrapScriptName = 'Start-EKSBootstrap.ps1'
[string]$EKSBootstrapScriptFile = "$EKSBinDir\$EKSBootstrapScriptName"
(Get-Content $EKSBootstrapScriptFile).replace('"--proxy-mode=kernelspace",', '"--proxy-
mode=kernelspace", "--feature-gates WinDSR=true", "--enable-dsr",') | Set-Content
 $EKSBootstrapScriptFile
& $EKSBootstrapScriptFile -EKSClusterName "eks-windows" -APIServerEndpoint "https://
<REPLACE-EKS-CLUSTER-CONFIG-API-SERVER>" -Base64ClusterCA "<REPLACE-EKSCLUSTER-
CONFIG-DETAILS-CA>" -DNSClusterIP "172.20.0.10" -KubeletExtraArgs "--node-
labels=alpha.eksctl.io/cluster-name=eks-windows,alpha.eksctl.io/nodegroup-name=windows-
ng-ltsc2019 --register-with-taints=" 3>&1 4>&1 5>&1 6>&1
</powershell>
```

DSR の有効化は、Microsoft Networking ブログと AWS Lab の Windows コンテナの指示に従って検証できます。

```
PS C:\temp> (Get-Content C:\temp\pmsx1eau.0ia\policy.txt | ConvertFrom-JSON).Policies


ExternalPort : 53
InternalPort : 53
IsDSR        : True
Protocol     : 17
Type         : ELB
VIPs         : {172.20.0.10}

ExternalPort : 53
InternalPort : 53
IsDSR        : True
Protocol     : 6
Type         : ELB
VIPs         : {172.20.0.10}

ExternalPort : 443
InternalPort : 443
IsDSR        : True
Protocol     : 6
Type         : ELB
VIPs         : {172.20.0.1}
```

サブネットで使用可能な IPv4 アドレスを保持し、無駄を最小限に抑えることが重要な場合は、Windows のプレフィックスモード - 避けるタイミングで説明されているように、プレフィックス委任モードを使用しないことをお勧めします。引き続きプレフィックス委任を使用する場合は、サブネットの IPv4 アドレス使用率を最適化する手順を実行できます。IPv4 アドレスのリクエストと割り当てプロセスを微調整する方法の詳細については、「プレフィックス委任のパラメータの設定」を参照してください。これらの設定を調整すると、IPv4 アドレスの節約とプレフィックス委任によるポッド密度の利点のバランスを取ることができます。

セカンダリ IPv4 アドレスを割り当てるというデフォルト設定を使用する場合、現在、VPC Resource Controller が IPv4 アドレスをリクエストして割り当てる方法を操作するためのサポートされている設定はありません。具体的には、 `minimum-ip-target`と `warm-ip-target`はプレフィックス委任モードでのみサポートされています。また、セカンダリ IP モードでは、インターフェイスで使用可能な IP アドレスに応じて、VPC リソースコントローラーは通常、ポッドの起動時間を短縮するためにウォーム IP を維持するために、ユーザーに代わって 3 つの未使用の IPv4 IPsをノードに割り当てます。未使用のウォーム IP アドレスの IP 浪費を最小限に抑える場合は、ENI の IP アドレス容量をできるだけ多く使用できるように、特定の Windows ノードでより多くのポッドをスケジュールすることを目指します。より明示的には、ENI 上のすべての IPs アドレスがノードと実行中のポッドで既に使用されている場合、ウォーム未使用 IP を避けることができます。サブネット内の IP アドレスの可用性に関する制約を解決するためのもう 1 つの回避策は、サブネットサイズを増やすか、Windows ノードを独自の専用サブネットに分離することです (複数可) 。

さらに、現時点では IPv6 は Windows ノードではサポートされていないことに注意してください。

コンテナネットワークインターフェイス (CNI) オプション

AWSVPC CNI は、Windows および Linux ワーカーノード用の事実上の CNI プラグインです。AWSVPC CNI は多くのお客様のニーズを満たす一方で、IP の枯渇を避けるためにオーバーレイネットワークなどの代替案を検討する必要がある場合があります。このような場合、Calico CNI を AWSVPC CNI の代わりに使用できます。Project Calico は、Tigera によって開発されたオープンソースソフトウェアです。このソフトウェアには、EKS で動作する CNI が含まれています。Calico CNI を EKS にインストールする手順は、Project Calico EKS のインストールページにあります。

ネットワークポリシー

ベストプラクティスとして、Kubernetes クラスター上のポッド間のオープン通信のデフォルトモードから、ネットワークポリシーに基づいてアクセスを制限するモードに変更します。オープンソースの Project Calico は、Linux ノードと Windows ノードの両方で動作するネットワークポリシーを強力にサポートしています。この機能は別個であり、Calico CNI の使用には依存しません。したがって、Calico をインストールし、ネットワークポリシー管理に使用することをお勧めします。

Calico を EKS にインストールする手順は、Amazon EKS の「Calico のインストール」ページで確認できます。

さらに、「Amazon EKS Best Practices Guide for Security - Network Section」で提供されるアドバイスは、Windows ワーカーノードを持つ EKS クラスターにも等しく適用されますが、現時点では「Security Groups for Pods」などの一部の機能は Windows ではサポートされていません。

OOM エラーの回避

Windows には、Linux out-of-memoryのプロセスキラーはありません。Windows は常にすべてのユーザーモードのメモリ割り当てを仮想として扱い、ページファイルは必須です。最終的な効果は、Windows が Linux と同じようにメモリ不足状態にならないことです。プロセスは、メモリ不足 (OOM) の終了ではなく、ディスクにページングされます。メモリが過剰にプロビジョニングされ、すべての物理メモリが枯渇した場合、ページングによってパフォーマンスが低下する可能性があります。

システムと kubelet メモリの予約

kubelet、コンテナランタイムなどの kubernetes システムデーモンの`--kubelet-reserve`リソース予約をキャプチャし、sshd、udev などの OS システムデーモンのリソース予約を`--system-reserve`キャプチャする Linux とは異なります。Windows では、これらのフラグは kubelet またはノードで実行されているプロセスでメモリ制限をキャプチャして設定しません。

ただし、これらのフラグを組み合わせて NodeAllocatable を管理し、ポッドマニフェストのメモリリソース制限を使用してノードの容量を減らし、ポッドあたりのメモリ割り当てを制御できます。この戦略を使用すると、メモリ割り当てをより適切に制御できるだけでなく、Windows ノードのout-of-memory (OOM) を最小限に抑えるメカニズムも得られます。

Windows ノードでは、ベストプラクティスとして、OS とプロセス用に少なくとも 2GB のメモリを予約します。`--kubelet-reserve` および/または を使用して `--system-reserve` NodeAllocatable を減らします。

Amazon EKS セルフマネージド Windows ノードドキュメントに従って、CloudFormation テンプレートを使用して、kubelet 設定をカスタマイズした新しい Windows ノードグループを起動します。CloudFormation には、 と同じ `BootstrapArguments` という要素がありますKubeletExtraArgs。次のフラグと値で を使用します。

```
--kube-reserved memory=0.5Gi,ephemeral-storage=1Gi --system-
reserved memory=1.5Gi,ephemeral-storage=1Gi --eviction-hard
 memory.available<200Mi,nodefs.available<10%"
```

eksctl がデプロイツールである場合は、次のドキュメントを確認して kubelet 設定をカスタマイズします。https://eksctl.io/usage/customizing-the-kubelet/

Windows コンテナのメモリ要件

Microsoft のドキュメントによると、NANO 用の Windows Server ベースイメージには少なくとも 30MB が必要ですが、Server Core には 45MB が必要です。これらの数は、.NET Framework、Web Services as IIS、アプリケーションなどの Windows コンポーネントを追加するにつれて増加します。

Windows コンテナイメージに必要な最小メモリ量、つまりベースイメージとそのアプリケーションレイヤーを把握し、ポッド仕様でコンテナのリソース/リクエストとして設定することが重要です。また、アプリケーションの問題が発生した場合にポッドが使用可能なすべてのノードメモリを消費しないように制限を設定する必要があります。

以下の例では、Kubernetes スケジューラがノードにポッドを配置しようとすると、ポッドのリクエストを使用して、スケジューリングに十分なリソースがあるノードが決定されます。

```
spec:
 - name: iis
   image: mcr.microsoft.com/windows/servercore/iis:windowsservercore-ltsc2019
   resources:
     limits:
       cpu: 1
       memory: 800Mi
     requests:
       cpu: .1
       memory: 128Mi
```

結論

このアプローチを使用すると、メモリ枯渇のリスクは最小限に抑えられますが、それを防ぐことはできません。Amazon CloudWatch Metrics を使用すると、メモリが枯渇した場合のアラートと修復を設定できます。

Windows サーバーとコンテナへのパッチ適用

Windows Server へのパッチ適用は、Windows 管理者向けの標準管理タスクです。これは、Amazon System Manager - Patch Manager、WSUS、System Center Configuration Manager など、さまざまなツールを使用して実現できます。ただし、Amazon EKS クラスター内の Windows ノードを通常の Windows サーバーとして扱うことはできません。これらはイミュータブルサーバーとして扱う必要

があります。簡単に言うと、既存のノードを更新せずに、新しく更新された AMI に基づいて新しいノードを起動します。

EC2 Image Builder を使用すると、レシピを作成し、コンポーネントを追加することで、AMIs の構築を自動化できます。

次の例は、AWS (Amazon マネージド) によって構築された既存のコンポーネントと、作成したコンポーネント (私が所有する) を示しています。update-windows と呼ばれる Amazon が管理するコンポーネントに細心の注意を払ってください。これにより、EC2 Image Builder パイプラインを介して AMI を生成する前に Windows Server が更新されます。

Associated components

Search components

1

Component name	Version	Description	Date created	Platform	Owner	ARN
eks-optimized-ami-windows	1.20.0	Installs EKS-optimized Windows artifacts for EKS version 1.20. This includes kubelet version 1.20.4 and Docker version 20.10.6.	Aug 18, 2021 6:33 PM	Windows	Amazon	arn:aws:imagebuilder:us-east-1:aws:component/eks-optimized-ami-windows/1.20.0/1
update-windows	1.0.1	Updates Windows with the latest security updates.	Aug 18, 2021 6:33 PM	Windows	Amazon	arn:aws:imagebuilder:us-east-1:aws:component/update-windows/1.0.1/1
Docker Pull	1.0.1	-	Dec 22, 2021 12:08 PM	Windows	[redacted]	arn:aws:imagebuilder:us-east-[redacted]component/docker-pull/1.0.1/1
simple-boot-test-windows	1.0.0	Executes a simple boot test.	Nov 30, 2019 11:39 PM	Windows	Amazon	arn:aws:imagebuilder:us-east-1:aws:component/simple-boot-test-windows/1.0.0/1

EC2 Image Builder を使用すると、Amazon Managed Public AMI に基づいて AMIs、ビジネス要件を満たすようにカスタマイズできます。その後、これらの AMIs を起動テンプレートに関連付けることができます。これにより、新しい AMI を EKS ノードグループによって作成された Auto Scaling グループにリンクできます。これが完了すると、既存の Windows ノードの終了を開始でき、新しく更新された AMI に基づいて新しい Windows ノードが起動されます。

Windows イメージのプッシュとプル

Amazon は、キャッシュされた 2 つの Windows コンテナイメージを含む EKS 最適化 AMIs を公開します。

```
mcr.microsoft.com/windows/servercore
mcr.microsoft.com/windows/nanoserver
```

```
REPOSITORY                                                      TAG                       IMAGE ID       CREATED          SIZE
mcr.microsoft.com/windows/servercore                            ltsc2019                  617dd5ead5b8   6 weeks ago      5.74GB
mcr.microsoft.com/windows/nanoserver                            1809                      8334af2fa7ba   6 weeks ago      257MB
[redacted].dkr.ecr.us-east-1.amazonaws.com/fluentd-windows-coreltsc   latest                    721afca2c725   7 weeks ago      6.96GB
fluent/fluentd                                                  v1.14-windows-ltsc2019-1  721afca2c725   7 weeks ago      6.96GB
amazonaws.com/eks/pause-windows                                 latest                    6392f69ae6e7   10 months ago    255MB
```

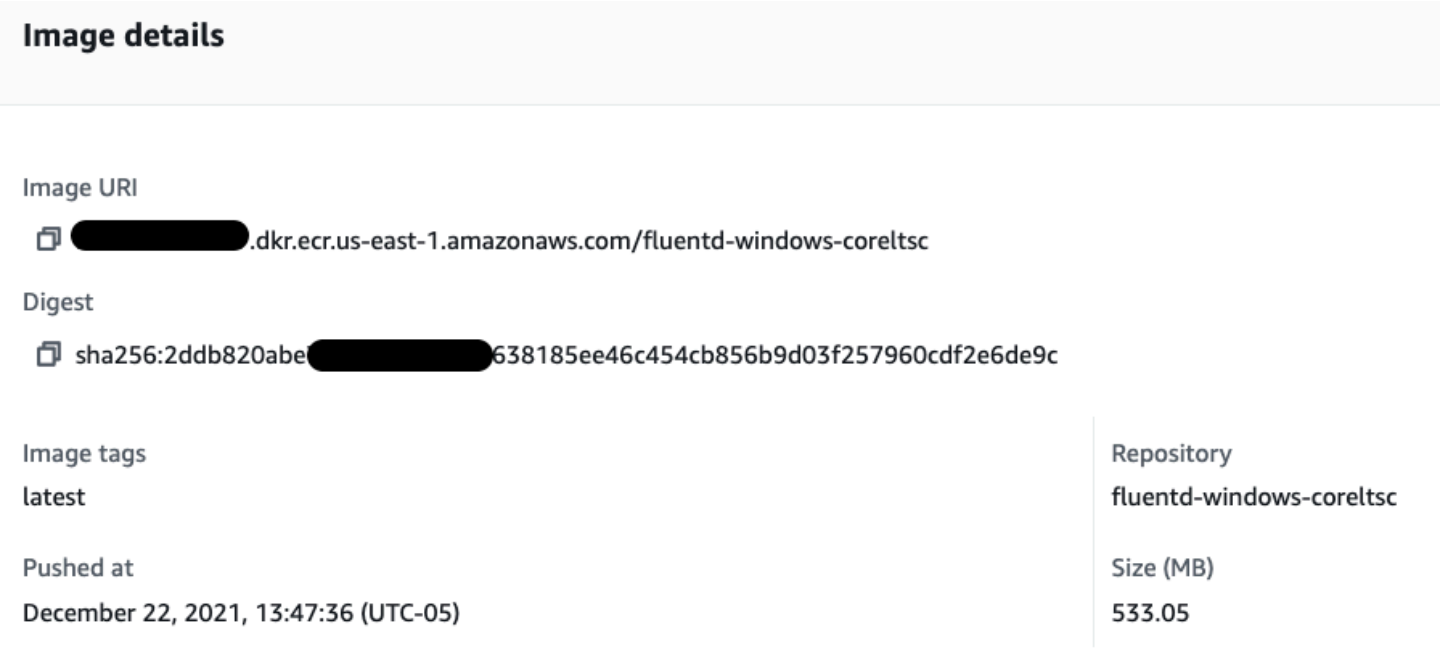
キャッシュされたイメージは、メイン OS の更新後に更新されます。Microsoft が Windows コンテナベースイメージに直接影響する新しい Windows 更新プログラムをリリースすると、その更新プログ

ラムはメイン OS で通常の Windows 更新プログラムとして起動されます。環境をup-to-dateに保つと、ノードおよびコンテナレベルでより安全な環境が提供されます。

Windows コンテナイメージのサイズはプッシュ/プルオペレーションに影響し、コンテナの起動時間が遅くなる可能性があります。Windows コンテナイメージをキャッシュすると、コンテナの起動ではなく AMI ビルドの作成時に高価な I/O オペレーション (ファイル抽出) を実行できます。その結果、必要なすべてのイメージレイヤーが AMI で抽出され、使用できるようになるため、Windows コンテナの起動時間が短縮され、トラフィックの受け入れを開始できます。プッシュオペレーション中、イメージを構成するレイヤーのみがリポジトリにアップロードされます。

次の例は、Amazon ECR の fluentd-windows-sac2004 イメージの 390.18 MB のみであることを示しています。これは、プッシュオペレーション中に発生したアップロードの量です。

次の例は、Amazon ECR リポジトリにプッシュされた流暢な Windows ltsc イメージを示しています。ECR に保存されるレイヤーのサイズは 533.05 MB です。



以下の からの出力ではdocker image ls、fluentd v1.14-windows-ltsc2019-1 のサイズはディスクで 6.96 GB ですが、その量のデータをダウンロードして抽出したわけではありません。

実際には、プルオペレーション中に圧縮された 533.05 MB のみがダウンロードおよび抽出されます。

```
REPOSITORY                                                                TAG
            IMAGE ID       CREATED         SIZE
```

```
111122223333.dkr.ecr.us-east-1.amazonaws.com/fluentd-windows-coreltsc   latest
            721afca2c725   7 weeks ago     6.96GB
fluent/fluentd                                                          v1.14-windows-
ltsc2019-1   721afca2c725   7 weeks ago     6.96GB
amazonaws.com/eks/pause-windows                                         latest
            6392f69ae6e7   10 months ago   255MB
```

サイズ列には、イメージの全体的なサイズ 6.96 GB が表示されます。内訳:

- Windows Server Core 2019 LTSC ベースイメージ = 5.74 GB
- Fluentd 非圧縮ベースイメージ = 6.96 GB
- ディスクの差 = 1.2 GB
- Fluentd 圧縮最終イメージ ECR = 533.05 MB

ベースイメージはローカルディスクに既に存在するため、ディスク上の合計容量はさらに 1.2 GB になります。次回、サイズ列に GBs の量が表示されるときは、心配しすぎないでください。70% 以上がキャッシュされたコンテナイメージとしてディスクに既に存在する可能性があります。

リファレンス

EC2 Image Builder とイメージキャッシュ戦略を使用して Windows コンテナの起動時間を短縮する

異種ワークロードの実行

Kubernetes は、同じクラスターに Linux ノードと Windows ノードを混在させることができる異種クラスターをサポートしています。そのクラスター内では、Linux で実行される Pod と Windows で実行される Pod を混在させることができます。同じクラスターで複数のバージョンの Windows を実行することもできます。ただし、この決定を行う際に考慮する必要があるいくつかの要因 (以下で説明) があります。

ノードへの PODs の割り当てのベストプラクティス

Linux および Windows ワークロードをそれぞれの OS 固有のノードに保持するには、ノードセレクタとテイント/許容範囲の組み合わせを使用する必要があります。異種環境でワークロードをスケジュールする主な目標は、既存の Linux ワークロードとの互換性が損なわれないようにすることです。

OS 固有のワークロードを適切なコンテナホストに配置する

ユーザーは、nodeSelectors を使用して Windows コンテナを適切なホストでスケジュールできます。現在、すべての Kubernetes ノードには次のデフォルトラベルがあります。

```
kubernetes.io/os = [windows|linux]
kubernetes.io/arch = [amd64|arm64|...]
```

Pod 仕様に のような nodeSelector が含まれていない場合`"kubernetes.io/os": windows`、Pod は任意のホスト、Windows、Linux でスケジュールできます。Windows コンテナは Windows でのみ実行でき、Linux コンテナは Linux でのみ実行できるため、これは問題になる可能性があります。

エンタープライズ環境では、Linux コンテナの既存のデプロイが多数あることも、Helm チャートなどのoff-the-shelf設定のエコシステムがあることも珍しくありません。このような状況では、デプロイの nodeSelectors を変更するのをためらうことがあります。代わりに、テイントを使用します。

例: `--register-with-taints='os=windows:NoSchedule'`

EKS を使用している場合、eksctl は clusterConfig を通じてテイントを適用する方法を提供します。

```
NodeGroups:
  - name: windows-ng
    amiFamily: WindowsServer2022FullContainer
    ...
    labels:
      nodeclass: windows2022
    taints:
      os: "windows:NoSchedule"
```

すべての Windows ノードにテイントを追加すると、スケジューラはテイントを許容しない限り、それらのノードでポッドをスケジュールしません。ポッドマニフェストの例:

```
nodeSelector:
    kubernetes.io/os: windows
tolerations:
    - key: "os"
      operator: "Equal"
      value: "windows"
      effect: "NoSchedule"
```

同じクラスターでの複数の Windows ビルドの処理

各ポッドで使用される Windows コンテナベースイメージは、ノードと同じカーネルビルドバージョンと一致する必要があります。同じクラスターで複数の Windows Server ビルドを使用する場合は、追加のノードラベル、nodeSelectors を設定するか、Windows-build というラベルを活用する必要があります。

Kubernetes 1.17 は、 という新しいラベルを自動的に追加node.kubernetes.io/windows-buildし、同じクラスター内の複数の Windows ビルドの管理を簡素化します。古いバージョンを実行している場合は、このラベルを手動で Windows ノードに追加することをお勧めします。

このラベルには、互換性のために一致する必要がある Windows メジャー、マイナー、ビルド番号が反映されています。以下は、Windows Server の各バージョンで現在使用されている値です。

Windows Server は、プライマリリリースチャネルとして長期サービスチャネル (LTSC) に移行していることに注意してください。Windows Server 半期チャネル (SAC) は 2022 年 8 月 9 日に廃止されました。Windows Server の今後の SAC リリースはありません。

[Product Name] (製品名)	ビルド番号 (複数可)
サーバーフル 2022 LTSC	10.0.20348
サーバーコア 2019 LTSC	10.0.17763

次のコマンドを使用して OS ビルドバージョンを確認できます。

```
kubectl get nodes -o wide
```

KERNEL-VERSION 出力は Windows OS ビルドバージョンと一致します。

```
NAME                            STATUS   ROLES    AGE   VERSION               INTERNAL-
IP   EXTERNAL-IP     OS-IMAGE                         KERNEL-VERSION
 CONTAINER-RUNTIME
ip-10-10-2-235.ec2.internal   Ready    <none>   23m   v1.24.7-eks-fb459a0
 10.10.2.235   3.236.30.157    Windows Server 2022 Datacenter   10.0.20348.1607
         containerd://1.6.6
ip-10-10-31-27.ec2.internal   Ready    <none>   23m   v1.24.7-eks-fb459a0
 10.10.31.27   44.204.218.24   Windows Server 2019 Datacenter   10.0.17763.4131
         containerd://1.6.6
```

```
ip-10-10-7-54.ec2.internal    Ready    <none>   31m   v1.24.11-eks-a59e1f0   10.10.7.54
    3.227.8.172     Amazon Linux 2                   5.10.173-154.642.amzn2.x86_64
 containerd://1.6.19
```

次の例では、異なる Windows ノードグループ OS バージョンを実行するときに正しい Windows ビルドバージョンと一致するように、追加の nodeSelector をポッドマニフェストに適用します。

```
nodeSelector:
    kubernetes.io/os: windows
    node.kubernetes.io/windows-build: '10.0.20348'
tolerations:
    - key: "os"
    operator: "Equal"
    value: "windows"
    effect: "NoSchedule"
```

RuntimeClass を使用して Pod マニフェストの NodeSelector と許容範囲を簡素化する

RuntimeClass を使用して、テイントと許容範囲を使用するプロセスを簡素化することもできます。これは、これらのテイントと許容値をカプセル化するために使用する RuntimeClass オブジェクトを作成することで実現できます。

次のマニフェストを実行して RuntimeClass を作成します。

```
apiVersion: node.k8s.io/v1beta1
kind: RuntimeClass
metadata:
  name: windows-2022
handler: 'docker'
scheduling:
  nodeSelector:
    kubernetes.io/os: 'windows'
    kubernetes.io/arch: 'amd64'
    node.kubernetes.io/windows-build: '10.0.20348'
  tolerations:
  - effect: NoSchedule
    key: os
    operator: Equal
    value: "windows"
```

Runtimeclass が作成されたら、 を Pod マニフェストの仕様として使用して割り当てます。

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: iis-2022
  labels:
    app: iis-2022
spec:
  replicas: 1
  template:
    metadata:
      name: iis-2022
      labels:
        app: iis-2022
    spec:
      runtimeClassName: windows-2022
      containers:
      - name: iis
```

マネージド型ノードグループのサポート

お客様が Windows アプリケーションをより合理化された方法で実行できるように、AWS は 2022 年 12 月 15 日に Windows コンテナの Amazon EKS Managed Node Group (MNG) サポートを開始しました。運用チームの足並みを揃えるために、Windows MNGs は Linux MNGs。Windows Server 2019 および 2022 の完全およびコア AMI (Amazon マシンイメージ) ファミリーバージョンがサポートされています。

Managed Node Groups (MNG) では、以下の AMI ファミリーがサポートされています。

AMI ファミリー
WINDOWS_CORE_2019_x86_64
WINDOWS_FULL_2019_x86_64
WINDOWS_CORE_2022_x86_64
WINDOWS_FULL_2022_x86_64

その他のドキュメント

AWS 公式ドキュメント: https://docs.aws.amazon.com/eks/latest/userguide/windows-support.html

Pod Networking (CNI) の仕組みを理解するには、次のリンクを確認してください。https://docs.aws.amazon.com/eks/latest/userguide/pod-networking.html

AWS Blog on Deploying Managed Node Group for Windows on EKS: https://aws.amazon.com/blogs/containers/deploying-amazon-eks-windows-managed-node-groups/

ポッドのセキュリティコンテキスト

Pod Security Policies (PSP) と Pod Security Standards (PSS) は、Kubernetes でセキュリティを適用する主な 2 つの方法です。PodSecurityPolicy は Kubernetes v1.21 で廃止され、v1.25 で削除され、Pod Security Standard (PSS) は今後セキュリティを適用するために Kubernetes が推奨するアプローチであることに注意してください。

ポッドセキュリティポリシー (PSP) は、セキュリティポリシーを実装するための Kubernetes のネイティブソリューションです。PSP は、Pod 仕様のセキュリティに敏感な側面を制御するクラスターレベルのリソースです。ポッドセキュリティポリシーを使用すると、ポッドがクラスターに受け入れられるために満たす必要がある一連の条件を定義できます。PSP 機能は Kubernetes の初期から利用可能であり、特定のクラスターで誤って設定されたポッドが作成されるのを防ぐように設計されています。

Pod セキュリティポリシーの詳細については、Kubernetes ドキュメントを参照してください。Kubernetes 非推奨ポリシーによると、古いバージョンでは、この機能が非推奨になってから 9 か月後にサポートされなくなります。

一方、推奨されるセキュリティアプローチであり、通常はセキュリティコンテキストを使用して実装されるポッドセキュリティ標準 (PSS) は、ポッドマニフェストのポッドとコンテナの仕様の一部として定義されます。PSS は、ポッドのセキュリティ関連のベストプラクティスに対処するために Kubernetes プロジェクトチームが定義した公式標準です。ベースライン (最小制限、デフォルト) 、特権 (無制限) 、制限 (最も制限的) などのポリシーを定義します。

ベースラインプロファイルから開始することをお勧めします。PSS ベースラインプロファイルは、セキュリティと潜在的な摩擦の強固なバランスを提供し、例外のリストを最小限に抑える必要があり、ワークロードセキュリティの良い出発点として機能します。現在 PSPs、PSS に切り替えることをお勧めします。PSS ポリシーの詳細については、Kubernetes ドキュメントを参照してくだ

さい。これらのポリシーは、OPA や Kyverno のものを含む複数のツールで適用できます。たとえば、Kyverno はここで PSS ポリシーの完全なコレクションを提供します。

セキュリティコンテキスト設定を使用すると、プロセスを選択する権限を付与したり、プログラムプロファイルを使用して個々のプログラムに機能を制限したり、権限のエスカレーションを許可したり、システムコールをフィルタリングしたりできます。

Kubernetes の Windows ポッドには、セキュリティコンテキストに関して、標準の Linux ベースのワークロードといくつかの制限と差別化要因があります。

Windows は、システム名前空間フィルターを使用してコンテナごとにジョブオブジェクトを使用し、コンテナ内のすべてのプロセスを含み、ホストから論理的に分離します。名前空間フィルタリングが設定されていない Windows コンテナを実行する方法はありません。つまり、ホストのコンテキストではシステム権限をアサートできないため、Windows では特権コンテナを使用できません。

以下に`windowsOptions`示しているのは、文書化された Windows セキュリティコンテキストオプションのみですが、残りは一般的なセキュリティコンテキストオプションです。

Windows と Linux でサポートされているセキュリティコンテキスト属性のリストについては、こちらの公式ドキュメントを参照してください。

Pod 固有の設定は、すべてのコンテナに適用されます。指定しない場合、PodSecurityContext のオプションが使用されます。SecurityContext と PodSecurityContext の両方で設定されている場合、SecurityContext で指定された値が優先されます。

たとえば、Windows オプションであるポッドとコンテナの runAsUserName 設定は、Linux 固有の runAsUser 設定とほぼ同等であり、次のマニフェストでは、ポッド固有のセキュリティコンテキストがすべてのコンテナに適用されます。

```
apiVersion: v1
kind: Pod
metadata:
  name: run-as-username-pod-demo
spec:
  securityContext:
    windowsOptions:
      runAsUserName: "ContainerUser"
  containers:
  - name: run-as-username-demo

  nodeSelector:
```

```
    kubernetes.io/os: windows
```

一方、以下では、コンテナレベルのセキュリティコンテキストはポッドレベルのセキュリティコンテキストを上書きします。

```
apiVersion: v1
kind: Pod
metadata:
  name: run-as-username-container-demo
spec:
  securityContext:
    windowsOptions:
      runAsUserName: "ContainerUser"
  containers:
  - name: run-as-username-demo
    ..
    securityContext:
        windowsOptions:
            runAsUserName: "ContainerAdministrator"
  nodeSelector:
    kubernetes.io/os: windows
```

runAsUserName フィールドの許容値の例: ContainerAdministrator、ContainerUser、NT AUTHORITY\NETWORK SERVICE、NT AUTHORITY\LOCAL SERVICE

ContainerUser for Windows ポッドを使用してコンテナを実行することをお勧めします。ユーザーはコンテナとホスト間で共有されませんが、ContainerAdministrator にはコンテナ内の に対する追加の権限があります。注意すべきユーザー名の制限があることに注意してください。

ContainerAdministrator を使用するときの良い例は、PATH を設定することです。USER ディレクティブを使用すると、次のようなことができます。

```
USER ContainerAdministrator
RUN setx /M PATH "%PATH%;C:/your/path"
USER ContainerUser
```

また、シークレットはノードのボリュームにクリアテキストで書き込まれます (linux の tmpfs/in-memory と比較して) 。つまり、2 つのことを行う必要があります。

- ファイル ACLs を使用してシークレットファイルの場所を保護する
- BitLocker を使用したボリュームレベルの暗号化の使用

永続的ストレージオプション

ツリー内ボリュームプラグインとout-of-treeボリュームプラグインとは

Container Storage Interface (CSI) の導入前は、すべてのボリュームプラグインがツリー内にあり、コア Kubernetes バイナリで構築、リンク、コンパイル、出荷され、コア Kubernetes API を拡張していました。つまり、新しいストレージシステムを Kubernetes (ボリュームプラグイン) に追加するには、コア Kubernetes コードリポジトリにコードをチェックする必要があります。

Out-of-tree ボリュームプラグインは Kubernetes コードベースとは独立して開発され、拡張機能として Kubernetes クラスターにデプロイ (インストール) されます。これにより、ベンダーはドライバーをout-of-band、つまり Kubernetes リリースサイクルとは別に更新できます。これは、Kubernetes が k8 とやり取りする標準的な方法をベンダーに提供するストレージインターフェイスまたは CSI を作成しているため、ほとんどの場合可能です。

Amazon Elastic Kubernetes Services (EKS) ストレージクラスと CSI ドライバーの詳細については、https://docs.aws.amazon.com/eks/latest/userguide/storage.html を参照してください。

Windows 用ツリー内ボリュームプラグイン

Kubernetes ボリュームを使用すると、データ永続性要件を持つアプリケーションを Kubernetes にデプロイできます。永続的ボリュームの管理は、ボリュームのプロビジョニング/プロビジョニング解除/サイズ変更、Kubernetes ノードへのボリュームのアタッチ/デタッチ、ポッド内の個々のコンテナへのボリュームのマウント/デタッチで構成されます。特定のストレージバックエンドまたはプロトコルにこれらのボリューム管理アクションを実装するためのコードは、Kubernetes ボリュームプラグイン (ツリー内ボリュームプラグイン) の形式で出荷されます。Amazon Elastic Kubernetes Services (EKS) では、次のクラスの Kubernetes ボリュームプラグインが Windows でサポートされています。

ツリー内ボリュームプラグイン: awsElasticBlockStore

Windows ノードでツリー内ボリュームプラグインを使用するには、NTFS を fsType として使用する追加の StorageClass を作成する必要があります。EKS では、デフォルトの StorageClass はデフォルトの fsType として ext4 を使用します。

StorageClass は、管理者が提供するストレージの「クラス」を記述する方法を提供します。異なるクラスは、クラスター管理者によって決定されるquality-of-serviceレベル、バックアップポリシー、または任意のポリシーにマッピングされる場合があります。Kubernetes は、クラスが表すものにつ

いては考えていません。この概念は、他のストレージシステムでは「プロファイル」と呼ばれることもあります。

これを確認するには、次のコマンドを実行します。

```
kubectl describe storageclass gp2
```

出力:

```
Name:            gp2
IsDefaultClass:  Yes
Annotations:     kubectl.kubernetes.io/last-applied-
configuration={"apiVersion":"storage.k8s.io/v1","kind":"StorageClas
","metadata":{"annotations":{"storageclass.kubernetes.io/is-default-
class":"true"},"name":"gp2"},"parameters":{"fsType"
"ext4","type":"gp2"},"provisioner":"kubernetes.io/aws-
ebs","volumeBindingMode":"WaitForFirstConsumer"}
,storageclass.kubernetes.io/is-default-class=true
Provisioner:           kubernetes.io/aws-ebs
Parameters:            fsType=ext4,type=gp2
AllowVolumeExpansion:  <unset>
MountOptions:          <none>
ReclaimPolicy:         Delete
VolumeBindingMode:     WaitForFirstConsumer
Events:                <none>
```

NTFS をサポートする新しい StorageClass を作成するには、次のマニフェストを使用します。

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: gp2-windows
provisioner: kubernetes.io/aws-ebs
parameters:
  type: gp2
  fsType: ntfs
volumeBindingMode: WaitForFirstConsumer
```

次のコマンドを実行して StorageClass を作成します。

```
kubectl apply -f NTFSStorageClass.yaml
```

次のステップでは、永続ボリュームクレーム (PVC) を作成します。

PersistentVolume (PV) は、管理者によってプロビジョニングされたか、PVC を使用して動的にプロビジョニングされたクラスター内のストレージです。これは、ノードがクラスターリソースであるかのように、クラスター内のリソースです。この API オブジェクトは、NFS、iSCSI、またはcloud-provider-specificストレージシステムなど、ストレージの実装の詳細をキャプチャします。

PersistentVolumeClaim (PVC) は、ユーザーによるストレージのリクエストです。クレームは、特定のサイズとアクセスモードをリクエストできます (例えば、ReadWriteOnce、ReadOnlyMany、または ReadWriteMany をマウントできます)。

ユーザーには、さまざまなユースケースでパフォーマンスなどのさまざまな属性を持つ PersistentVolumes が必要です。クラスター管理者は、ユーザーがそれらのボリュームの実装方法の詳細を公開することなく、サイズやアクセスモードだけでなく、さまざまな方法で異なるさまざまな PersistentVolumes を提供できる必要があります。これらのニーズには、StorageClass リソースがあります。

以下の例では、名前空間ウィンドウ内に PVC が作成されています。

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: ebs-windows-pv-claim
  namespace: windows
spec:
  accessModes:
    - ReadWriteOnce
  storageClassName: gp2-windows
  resources:
    requests:
      storage: 1Gi
```

次のコマンドを実行して PVC を作成します。

```
kubectl apply -f persistent-volume-claim.yaml
```

次のマニフェストは Windows Pod を作成し、VolumeMount を として設定C:\Dataし、PVC を のアタッチされたストレージとして使用しますC:\Data。

```
apiVersion: apps/v1
```

```
kind: Deployment
metadata:
  name: windows-server-ltsc2019
  namespace: windows
spec:
  selector:
    matchLabels:
      app: windows-server-ltsc2019
      tier: backend
      track: stable
  replicas: 1
  template:
    metadata:
      labels:
        app: windows-server-ltsc2019
        tier: backend
        track: stable
    spec:
      containers:
      - name: windows-server-ltsc2019
        image: mcr.microsoft.com/windows/servercore:ltsc2019
        ports:
        - name: http
          containerPort: 80
        imagePullPolicy: IfNotPresent
        volumeMounts:
        - mountPath: "C:\\data"
          name: test-volume
      volumes:
        - name: test-volume
          persistentVolumeClaim:
            claimName: ebs-windows-pv-claim
      nodeSelector:
        kubernetes.io/os: windows
        node.kubernetes.io/windows-build: '10.0.17763'
```

PowerShell 経由で Windows ポッドにアクセスして結果をテストします。

```
kubectl exec -it podname powershell -n windows
```

Windows Pod 内で、以下を実行します。 `ls`

出力:

```
PS C:\> ls


    Directory: C:\


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----          3/8/2021   1:54 PM                data
d-----          3/8/2021   3:37 PM                inetpub
d-r---          1/9/2021   7:26 AM                Program Files
d-----          1/9/2021   7:18 AM                Program Files (x86)
d-r---          1/9/2021   7:28 AM                Users
d-----          3/8/2021   3:36 PM                var
d-----          3/8/2021   3:36 PM                Windows
-a----         12/7/2019   4:20 AM           5510 License.txt
```

データディレクトリは EBS ボリュームによって提供されます。

Windows Out-of-tree

CSI プラグインに関連付けられたコードは、通常コンテナイメージとして配布され、DaemonSets や StatefulSets などの標準の Kubernetes コンストラクトを使用してデプロイされるout-of-treeスクリプトおよびバイナリとして出荷されます。 StatefulSets CSI プラグインは、Kubernetes で幅広いボリューム管理アクションを処理します。CSI プラグインは通常、ノードプラグイン (DaemonSet として各ノードで実行される) とコントローラープラグインで構成されます。

CSI ノードプラグイン (特にブロックデバイスまたは共有ファイルシステムを介して公開される永続的ボリュームに関連付けられているもの) は、ディスクデバイスのスキャン、ファイルシステムのマウントなど、さまざまな特権オペレーションを実行する必要があります。これらのオペレーションはホストオペレーティングシステムごとに異なります。Linux ワーカーノードの場合、コンテナ化された CSI ノードプラグインは通常、特権コンテナとしてデプロイされます。Windows ワーカーノードの場合、コンテナ化された CSI ノードプラグインの特権オペレーションは、各 Windows ノードにプリインストールする必要があるコミュニティ管理のスタンドアロンバイナリである csi-proxy を使用してサポートされます。

Amazon EKS 最適化 Windows AMI には、2022 年 4 月以降の CSI プロキシが含まれています。お客様は、Windows ノードで SMB CSI ドライバーを使用して、Amazon FSx for Windows File Server、Amazon FSx for NetApp ONTAP SMB 共有、および/または AWS Storage Gateway — File Gateway にアクセスできます。

次のブログでは、Amazon FSx for Windows File Server を Windows Pods の永続的ストレージとして使用するように SMB CSI ドライバーを設定する方法の実装の詳細について説明しています。

Amazon FSx for Windows File Server

オプションとして、SMB グローバルマッピングと呼ばれる SMB 機能を使用して Amazon FSx for Windows File Server を使用する方法があります。これにより、ホストに SMB 共有をマウントし、その共有のディレクトリをコンテナに渡すことができます。コンテナは、特定のサーバー、共有、ユーザー名、パスワードで設定する必要はありません。すべてホストで処理されます。コンテナは、ローカルストレージがある場合と同じように機能します。

> SMB グローバルマッピングはオーケストレーターに対して透過的であり、HostPath を介してマウントされるため、安全な懸念が生じる可能性があります。

以下の例では、パス`G:\Directory\app-state`は Windows ノード上の SMB 共有です。

```
apiVersion: v1
kind: Pod
metadata:
  name: test-fsx
spec:
  containers:
  - name: test-fsx
    image: mcr.microsoft.com/windows/servercore:ltsc2019
    command:
      - powershell.exe
      - -command
      - "Add-WindowsFeature Web-Server; Invoke-WebRequest -UseBasicParsing
 -Uri 'https://dotnetbinaries.blob.core.windows.net/servicemonitor/2.0.1.6/
ServiceMonitor.exe' -OutFile 'C:\\ServiceMonitor.exe'; echo '<html><body><br/
><br/><marquee><H1>Hello EKS!!!<H1><marquee></body><html>' > C:\\inetpub\\wwwroot\
\default.html; C:\\ServiceMonitor.exe 'w3svc'; "
    volumeMounts:
      - mountPath: C:\dotnetapp\app-state
        name: test-mount
  volumes:
    - name: test-mount
      hostPath:
        path: G:\Directory\app-state
        type: Directory
  nodeSelector:
      beta.kubernetes.io/os: windows
```

```
      beta.kubernetes.io/arch: amd64
```

次のブログでは、Amazon FSx for Windows File Server を Windows Pod の永続ストレージとして設定する方法の実装の詳細について説明しています。

Windows コンテナイメージの強化

Windows コンテナイメージを強化していますか? 長年にわたり、私は世界中のお客様と連携し、レガシーワークロードをコンテナ、特に Windows ワークロードに移行できるようにしてきました。20 年以上の経験を持つ組織では、Windows Server の強化に多大な労力とリソースを注いでおり、CIS Benchmarks からランタイムウイルス対策保護まで、機密データを保護するためのすべてを実装しています。

ただし、懸念される傾向が浮上しています。これらの安全性の高い仮想マシンはコンテナにモダナイズされるため、多くの重要な強化プラクティスが見落とされています。ベースイメージ (OS) から IIS などのウェブサービスまで、Windows セキュリティのベストプラクティスは無視されることが多く、ほとんどがコンテナホストの保護のみに焦点を当てています。コンテナが分離された名前空間で動作している間も、カーネルプリミティブはホストと共有されることを認識することが重要です。攻撃者は通常、コンテナホストを直接ターゲットにするのではなく、横方向の動きに関心を持ち、弱いコンテナセキュリティ設定を悪用して機密データにアクセスできるようにします。

ドキュメントの目的は、IIS でASP.NETウェブサイトをホストする Windows コンテナに特に実装する必要がある重要なセキュリティ設定をいくつか紹介することです。次の 4 つの主要分野に焦点を当てます。

- アカウントセキュリティポリシー
- 監査ポリシー
- IIS セキュリティのベストプラクティス
- 最小権限の原則

まず、これらの各セキュリティ設定が Windows コンテナの保護に不可欠である理由を調べ、それらが軽減する特定のリスクとそれらが提供するセキュリティ上の利点を調べます。次に、Dockerfile にこれらの設定を正しく実装し、潜在的な脅威に対してコンテナを強化する方法を示すコードスニペットについて説明します。最後に、各設定を詳細に分類し、その機能、コンテナセキュリティへの影響、アプリケーションの保護にどのように貢献するかを包括的に説明します。このアプローチは、これらのベストプラクティスを適用する方法を示すだけでなく、コンテナ化された環境で堅牢なセキュリティ体制を維持するために必要不可欠な理由を理解するためのインサイトも提供します。

1. ローカルセキュリティポリシーとレジストリを使用してアカウントポリシー (パスワードまたはロックアウト) を設定する

Windows Server Core は、[EKS Optimized Windows AMI](https://docs.aws.amazon.com/eks/latest/userguide/eks-optimized-windows-ami.html) の一部として使用できる最小限のインストールオプションです。ローカルセキュリティポリシーとレジストリを使用してアカウントポリシー (パスワードまたはロックアウト) を設定すると、堅牢なパスワードとロックアウトルールを適用することで、システムセキュリティが強化されます。これらのポリシーでは、ユーザーは最小長と複雑さが定義された強力なパスワードを作成し、一般的なパスワード関連の攻撃から保護する必要があります。

パスワードの最大有効期間を設定することで、ユーザーはパスワードを定期的に更新するように求められ、認証情報が侵害される可能性が低くなります。ロックアウトポリシーは、指定した回数のログイン試行の失敗後にアカウントを一時的にロックすることで保護を強化し、ブルートフォース攻撃を防ぐのに役立ちます。Windows レジストリを使用してこれらの設定を行うと、管理者はこれらのセキュリティ対策をシステムレベルで適用できるため、組織全体の統一性とコンプライアンスが確保されます。これらのアカウントポリシーを Windows コンテナに適用することは、コンテナがエフェメラルで、分離されたワークロードを対象としている場合が多い場合でも、セキュリティの一貫性を維持する上で不可欠です。

セキュリティの整合性

- コンプライアンス: コンテナに一貫したパスワードポリシーとロックアウトルールを適用すると、特に厳格なアクセスコントロールを必要とする環境 (HIPAA、PCI-DSS などの規制コンプライアンスなど) でセキュリティコンプライアンスを維持するのに役立ちます。
- 強化されたコンテナ: これらの設定を適用すると、Windows コンテナが不正アクセスやパスワードベースの攻撃に対して強化され、コンテナのセキュリティ体制がより広範なシステムセキュリティポリシーと一致します。

ブルートフォース攻撃に対する保護

- アカウントロックアウト: これらの設定は、特定の回数のログイン試行が失敗した後にアカウントをロックすることで、ブルートフォースログイン試行を防ぐのに役立ちます。これにより、攻撃者は無制限のパスワードを試すことができません。
- パスワードの複雑さ: 十分な長さの複雑なパスワードを要求すると、隔離されたコンテナ化された環境でも弱いパスワードが悪用される可能性が低くなります。

マルチユーザーシナリオ

- コンテナ化されたアプリケーションが複数のユーザーを処理するように設計されている場合、またはユーザー認証が必要な場合、パスワードポリシーを適用すると、コンテナ内のユーザーアカウントが厳格なセキュリティルールに準拠し、アクセスが承認されたユーザーのみに制限されます。

永続的な Windows コンテナ

- コンテナは通常エフェメラルと見なされますが、特定の Windows コンテナは長期サービスを実行したり、ユーザー管理を処理したりできるため、通常の Windows サーバーと同様の適切なセキュリティポリシーを適用することが重要です。

ハイブリッド環境での整合性

- インフラストラクチャで仮想マシンとコンテナの両方を実行している場合、すべての環境に同じセキュリティポリシー (パスワード/ロックアウトポリシーなど) を適用すると、セキュリティ標準が統一され、ガバナンスと管理が簡素化されます。

要約すると、これらのアカウントポリシーを Windows コンテナ内に適用すると、コンテナがセキュリティ戦略の弱点にならず、パスワード攻撃から保護され、環境全体で整合性が強化されます。

Dockerfile:

```
# Configure account policies for password complexity and lockout
RUN powershell -Command \
      "Write-Output 'Configuring Account Policies (Password/Lockout)...'; \
      NET ACCOUNTS /MINPWLEN:14 /MAXPWAGE:60 /MINPWAGE:14 /LOCKOUTTHRESHOLD:5
```

説明:

このセクションでは、Windows レジストリを介してパスワードとロックアウト設定のアカウントポリシーを設定します。これらのポリシーは、パスワード要件とアカウントのロックアウトしきい値を制御することで、セキュリティを強化するのに役立ちます。

1. MinimumPasswordLength (MINPWLEN) = 14 この設定では、パスワードの最小文字数を定義します。範囲は 0 ~ 14 文字で、デフォルトは 6 文字です。

2. MaximumPasswordAge (MAXPWAGE) = 60 この設定では、パスワードが有効な最大日数を定義します。UNLIMITED を使用して制限は指定されません。/MAXPWAGE は /MINPWAGE より小さくすることはできません。範囲は 1～999 で、デフォルトは 90 日です。
3. ロックアウトしきい値 (LOCKOUTTHRESHOLD) = 5 この設定では、失敗したログイン試行のしきい値を定義します。5 回間違えると、アカウントはロックされます。

これらの設定は、強力なパスワードポリシーを適用し、ログイン試行が一定回数失敗した後にアカウントをロックアウトすることで、パスワードセキュリティを向上させ、ブルートフォース攻撃を防ぐのに役立ちます。

2. 監査ポリシー

監査ポリシーは Windows コンテナにとって重要です。これは、ログイン試行や特権の使用などのセキュリティイベントに対する重要な可視性を提供し、不正アクセスの検出、ユーザーアクティビティのモニタリング、規制標準への準拠の確保に役立つためです。コンテナのエフェメラルな性質であっても、監査ログは、インシデント調査、プロアクティブな脅威検出、コンテナ化された環境全体で一貫したセキュリティ体制の維持に不可欠です。

セキュリティのモニタリングとコンプライアンス:

- ユーザーアクティビティの追跡: 監査ポリシーにより、管理者はコンテナ内のログイン試行や特権の使用などのユーザーアクティビティをモニタリングできます。これは、不正アクセスや疑わしい動作を検出するために重要です。
- 規制コンプライアンス: 多くの組織は、HIPAA、PCI-DSS、GDPR などの規制に準拠するためにセキュリティイベントを記録する必要があります。コンテナで監査ポリシーを有効にすると、コンテナ化された環境でもこれらの要件を満たすことができます。

インシデント調査:

- フォレンジックと分析: コンテナ化されたアプリケーションまたはサービスが侵害された場合、監査ログはインシデント後の分析に役立つインサイトを提供します。セキュリティチームは、攻撃者が実行したアクションを追跡したり、違反がどのように発生したかを特定したりできます。
- リアルタイム検出: 監査ログを使用すると、管理者は重要なイベント (ログイン試行の失敗、特権エスカレーションなど) のリアルタイムアラートを設定できます。このプロアクティブモニタリングは、攻撃を早期に検出し、応答時間を短縮するのに役立ちます。

環境間の一貫性:

- 統一されたセキュリティ体制: レジストリを介してコンテナに監査ポリシーを適用することで、コンテナ化された環境とコンテナ化されていない環境の両方で一貫したセキュリティプラクティスを確保できます。これにより、コンテナがセキュリティモニタリングの死角になることを回避できます。
- ハイブリッド環境での可視性: 従来の Windows サーバーとコンテナの両方を実行している組織の場合、監査ポリシーはすべてのプラットフォームで同様の可視性と制御を提供し、管理をより簡単かつ効果的にします。

特権オペレーションの追跡:

- 特権使用監査: アプリケーションが昇格された特権で実行されるコンテナ環境や管理タスクが実行されるコンテナ環境では、特権オペレーションを監査することで説明責任が保証されます。コンテナ内で機密リソースにアクセスしたユーザーや重要なタスクを実行したユーザーを記録できます。
- 権限の濫用を防ぐ: 権限の使用をモニタリングすることで、権限のないユーザーが権限を昇格させたり、コンテナ内の制限された領域にアクセスしようとするタイミングを検出できるため、内部または外部の攻撃を防ぐことができます。

不正なアクセス試行の検出:

- 失敗したログオン試行: 失敗したログイン試行の監査ポリシーを有効にすると、コンテナ化されたアプリケーションにアクセスするためのブルートフォース攻撃や不正な試行を特定するのに役立ちます。これにより、システムにアクセスしようとしているユーザーと頻度を可視化できます。
- アカウントロックアウトのモニタリング: アカウントのロックアウトイベントを監査することで、管理者は疑わしいアクティビティや悪意のあるアクティビティによって発生する可能性のあるロックアウトを検出して調査できます。

エフェメラル環境でも永続的なセキュリティ:

- Ephemeral Yet Secure: コンテナはエフェメラルです。つまり、頻繁に削除および再作成できますが、監査は、コンテナの実行中にセキュリティイベントがキャプチャされるようにするための主要な役割を果たします。これにより、コンテナのライフサイクル中に重要なセキュリティイベントが記録されます。

集中ログ記録:

- 集中型システムへのログの転送: コンテナを集中型ログ記録システム (ELK スタック、AWS CloudWatch など) と統合して、複数のコンテナインスタンスから監査ログをキャプチャできます。これにより、インフラストラクチャ全体のセキュリティイベントのより良い分析と相関が可能になります。

Dockerfile:

```
# Configure audit policies for logging security events
RUN powershell -Command \
    "Write-Host 'Configuring Audit Policy..'; \
    Set-ItemProperty -Path 'HKLM:\\SYSTEM\\CurrentControlSet\\Control\\Lsa' -Name
 'SCENoApplyLegacyAuditPolicy' -Value 0; \
    auditpol /set /category:"Logon/Logoff" /subcategory:"Logon" /failure:enable

# Creates STDOUT on Windows Containers (check GitHub LogMonitor:: https://github.com/
microsoft/windows-container-tools/blob/main/LogMonitor/README.md)
COPY LogMonitor.exe LogMonitorConfig.json 'C:\\LogMonitor\\'
WORKDIR /LogMonitor
```

説明:

このセクションでは、レジストリの変更を使用して監査ポリシーを設定します。監査ポリシーは、Windows によって ログに記録されるセキュリティイベントを制御し、不正アクセスの試みのモニタリングと検出に役立ちます。

1. SCENoApplyLegacyAuditPolicy = 0 レガシー監査ポリシー形式を無効にし、新しいバージョンの Windows で導入されたより詳細な監査ポリシーを有効にします。これは最新の監査設定にとって重要です。
2. Auditpol Subcategory: "Logon" この設定により、成功ログオンイベントと失敗ログオンイベントの両方を監査できます。値 3 は、Windows がログオン試行の成功と失敗の両方を記録することを意味します。これにより、システムにアクセスしているユーザーをモニタリングし、失敗したログイン試行をキャッチできます。

これらの監査ポリシーは、ログイン試行や特権オペレーションの使用など、重要なセキュリティイベントの詳細なログを提供するため、セキュリティのモニタリングとコンプライアンスに不可欠です。

3. Windows コンテナの IIS セキュリティのベストプラクティス

Windows コンテナに IIS のベストプラクティスを実装することは、いくつかの理由で重要であり、アプリケーションの安全性、高性能、スケーラビリティを確保します。コンテナは分離と軽量な環境を提供しますが、脆弱性や運用上の問題を回避するために適切な設定が必要です。Windows コンテナでの IIS のベストプラクティスに従うことが重要です。

セキュリティ

- 一般的な脆弱性の防止: IIS は、多くの場合、クロスサイトスクリプティング (XSS)、クリックジャック、情報開示などの攻撃のターゲットです。セキュリティヘッダー (X-Content-Type-Options、X-Frame-Options、Strict-Transport-Security など) を実装すると、これらの脅威からアプリケーションを保護するのに役立ちます。
- 分離が不十分: コンテナは分離されていますが、IIS インスタンスの設定ミスにより、サーバーバージョンの詳細、ディレクトリのリスト、暗号化されていない通信などの機密情報が公開される可能性があります。ディレクトリブラウジングや IIS バージョンヘッダーの削除などの機能を無効にして、攻撃対象領域を最小限に抑えます。
- 暗号化と HTTPS: HTTPS 専用接続の強制などのベストプラクティスでは、転送中のデータが暗号化され、機密情報が傍受されないようにします。

パフォーマンス

- 効率的なリソース使用量: 動的圧縮と静的圧縮の有効化などの IIS のベストプラクティスにより、帯域幅の使用量が減少し、ロード時間が短縮されます。これらの最適化は、リソースがコンテナとホストシステム間で共有されるコンテナ化された環境で特に重要です。
- 最適化ログ記録: ログ記録を適切に設定 (X-Forwarded-For ヘッダーなど) することで、不要なログ記録のオーバーヘッドを最小限に抑えながら、クライアントアクティビティを追跡できます。これにより、パフォーマンスを低下させることなく、トラブルシューティングに関連するデータを収集できます。

スケーラビリティとメンテナンス性

- 環境間の整合性: ベストプラクティスに従うことで、IIS 設定が複数のコンテナインスタンス間で一貫していることを確認します。これにより、スケーリングが簡素化され、新しいコンテナがデプロイされたときに、同じセキュリティとパフォーマンスのガイドラインに準拠していることが保証されます。

- 自動設定: フォルダのアクセス許可の設定や不要な機能の無効化など、Dockerfiles のベストプラクティスでは、新しいコンテナごとに自動的に正しく設定されていることを確認します。これにより、手動による介入が減り、人為的ミスのリスクが軽減されます。

コンプライアンス

- 規制要件を満たす: 多くの業界には、暗号化通信 (HTTPS) やクライアントリクエストのログ記録など、特定のセキュリティ対策を義務付ける厳格な規制要件 (PCI-DSS、HIPAA など) があります。コンテナの IIS ベストプラクティスに従うことで、これらの標準を確実に順守できます。
- 監査可能性: 監査ポリシーと安全なログ記録を実装することで、監査で重要なイベントのトレーサビリティが可能になります。たとえば、X-Forwarded-For ヘッダーをログに記録すると、クライアント IP アドレスがプロキシベースのアーキテクチャに正しく記録されます。

共有環境でのリスクの最小化

- 設定ミスの回避: コンテナはホストのカーネルを共有し、互いに分離されている間、設定が不十分な IIS インスタンスは脆弱性を公開したり、パフォーマンスのボトルネックを発生させたりする可能性があります。ベストプラクティスにより、各 IIS インスタンスが最適に実行され、コンテナ間の問題のリスクが軽減されます。
- 最小特権アクセス: コンテナ内のフォルダとファイルに適切なアクセス許可を設定する (PowerShell で Set-Acl を使用するなど) と、コンテナ内のユーザーとプロセスに必要なアクセスのみが確保されるため、特権のエスカレーションやデータ改ざんのリスクが軽減されます。

エフェメラル環境の耐障害性

- コンテナのエフェメラルな性質: コンテナは短期間で頻繁に再構築されます。IIS のベストプラクティスを適用すると、再デプロイ回数に関係なく、各コンテナが安全かつ一貫して設定されます。これにより、時間の経過とともに設定ミスが発生するのを防ぐことができます。
- 潜在的な設定ミスの軽減: ベストプラクティスを自動的に適用することで (弱いプロトコルやヘッダーを無効にするなど) 、コンテナの再起動や更新中に設定ミスが発生するリスクが最小限に抑えられます。

Dockerfile:

```
# Enforce HTTPS (disable HTTP) -- Only if container is target for SSL termination
RUN powershell -Command \
```

```
    "$httpBinding = Get-WebBinding -Name 'Default Web Site' -Protocol http | Where-
Object { $_.bindingInformation -eq '*:80:' }; \
    if ($httpBinding) { Remove-WebBinding -Name 'Default Web Site' -Protocol http -Port
 80; } \
    $httpsBinding = Get-WebBinding -Name 'Default Web Site' -Protocol https | Where-
Object { $_.bindingInformation -eq '*:443:' }; \
    if (-not $httpsBinding) { New-WebBinding -Name 'Default Web Site' -Protocol https -
Port 443 -IPAddress '*'; }"

# Use secure headers
RUN powershell -Command \
    "Write-Host 'Adding security headers...'; \
    Add-WebConfigurationProperty -pspath 'MACHINE/WEBROOT/APPHOST' -filter
 'system.applicationHost/sites/siteDefaults/logFile/customFields' -name
 "." -value @{logFieldName='X-Forwarded-For';sourceName='X-Forwarded-
For';sourceType='RequestHeader'}; \
    Add-WebConfigurationProperty -pspath 'MACHINE/WEBROOT/APPHOST' -filter
 "system.webServer/httpProtocol/customHeaders" -name "." -value @{name='Strict-
Transport-Security';value='max-age=31536000; includeSubDomains'}; \
    Add-WebConfigurationProperty -pspath 'MACHINE/WEBROOT/APPHOST' -filter
 "system.webServer/httpProtocol/customHeaders" -name "." -value @{name='X-Content-Type-
Options';value='nosniff'}; \
    Add-WebConfigurationProperty -pspath 'MACHINE/WEBROOT/APPHOST' -filter
 "system.webServer/httpProtocol/customHeaders" -name "." -value @{name='X-XSS-
Protection';value='1; mode=block'}; \
    Add-WebConfigurationProperty -pspath 'MACHINE/WEBROOT/APPHOST' -filter
 "system.webServer/httpProtocol/customHeaders" -name "." -value @{name='X-Frame-
Options';value='DENY'};"

# Disable IIS version disclosure
RUN powershell -Command \
    "Write-Host 'Disabling IIS version disclosure...'; \
    Import-Module WebAdministration; \
    Set-WebConfigurationProperty -pspath 'MACHINE/WEBROOT/APPHOST' -filter
 "system.webServer/security/requestFiltering" -name "removeServerHeader" -value
 "true";"

# Set IIS Logging Best Practices
RUN powershell -Command \
    Set-WebConfigurationProperty -pspath 'MACHINE/WEBROOT/APPHOST' -filter
 "system.webServer/directoryBrowse" -name "enabled" -value "false"; \
    Set-WebConfigurationProperty -pspath 'MACHINE/WEBROOT/APPHOST' -filter
 "system.webServer/httpErrors" -name "existingResponse" -value "PassThrough"; \
```

```
# Enable IIS dynamic and static compression to optimize performance
RUN powershell -Command \
    "Write-Host 'Enabling IIS compression...'; \
    Enable-WindowsOptionalFeature -Online -FeatureName IIS-HttpCompressionDynamic; \
    Import-Module WebAdministration; \
    Set-WebConfigurationProperty -pspath 'MACHINE/WEBROOT/APPHOST' -filter
 "system.webServer/urlCompression" -name "doDynamicCompression" -value "true"; \
    Set-WebConfigurationProperty -pspath 'MACHINE/WEBROOT/APPHOST' -filter
 "system.webServer/urlCompression" -name "doStaticCompression" -value "true"

# Ensure proper folder permissions using PowerShell's Set-Acl

RUN powershell -Command \
    "Write-Host 'Setting folder permissions for IIS...'; \
    $path = 'C:\\inetpub\\wwwroot'; \
    $acl = Get-Acl $path; \
    $iusr = New-Object System.Security.Principal.NTAccount('IIS_IUSRS'); \
    $rule = New-Object System.Security.AccessControl.FileSystemAccessRule($iusr,
 'ReadAndExecute', 'ContainerInherit, ObjectInherit', 'None', 'Allow'); \
    $acl.SetAccessRule($rule); \
    $users = New-Object System.Security.Principal.NTAccount('Users'); \
    $rule2 = New-Object System.Security.AccessControl.FileSystemAccessRule($users,
 'ReadAndExecute', 'ContainerInherit, ObjectInherit', 'None', 'Allow'); \
    $acl.SetAccessRule($rule2); \
    Set-Acl -Path $path -AclObject $acl"
```

説明:

このコマンドは、X-Forwarded-For ヘッダーをログに記録するように IIS を設定します。これは、リクエストがプロキシまたはロードバランサーを通過するときに、元のクライアント IP アドレスをキャプチャするために一般的に使用されます。デフォルトでは、IIS はロードバランサーまたはリバースプロキシの IP アドレスのみをログに記録するため、このカスタムログフィールドを追加すると、セキュリティ監査、分析、トラブルシューティングのために真のクライアント IP を追跡するのに役立ちます。

1. X-Forwarded-For ヘッダー。リクエストがプロキシまたはロードバランサーを通過するときに、元のクライアント IP アドレスをキャプチャするために一般的に使用されます。デフォルトでは、IIS はロードバランサーまたはリバースプロキシの IP アドレスのみをログに記録するため、このカスタムログフィールドを追加すると、セキュリティ監査、分析、トラブルシューティングのために真のクライアント IP を追跡するのに役立ちます。

2. Strict-Transport-Security (HSTS) ブラウザが HTTPS 経由でのみ通信するようにします。max-age=31536000 は、このポリシーを 1 年間適用することを指定し、includeSubDomains はすべてのサブドメインにポリシーを適用します。
3. X-Content-Type-Options ブラウザが宣言された Content-Type からレスポンスを「MIME-sniffing」できないようにします。これにより、一部のタイプの攻撃を防ぐことができます。
4. X-XSS-Protection は、ブラウザでクロスサイトスクリプティング (XSS) 保護を有効にします。
5. X-Frame-Options ページが iframe に埋め込まれないようにし、クリックジャック攻撃から保護します。
6. IIS バージョン開示を無効にする このコマンドは、HTTP レスポンスの Server ヘッダーを無効にします。デフォルトでは、使用されている IIS のバージョンが表示されます。この情報を非表示にすると、攻撃者が IIS バージョンに固有の脆弱性を特定してターゲットにするリスクを軽減できます。
7. HTTPS 専用接続を有効にする この (コメントアウト) セクションでは、HTTPS 接続を強制し、HTTP を無効にします。コメントを解除すると、Dockerfile は、ポート 443 (HTTPS) でのみリッスンし、ポート 80 のデフォルトの HTTP バインディングを削除するように IIS を設定します。これは、コンテナ内で SSL を終了し、すべてのトラフィックが暗号化されるようにする場合に便利です。
8. ディレクトリ閲覧を無効にする デフォルトのドキュメントが存在しない場合に IIS がディレクトリリストを表示できないようにします。これにより、内部ファイル構造がユーザーに公開されるのを回避できます。
9. カスタムエラーページのパススルー アプリケーションに独自のエラー処理がある場合、IIS はデフォルトの IIS エラーページを表示するのではなく、アプリケーションのエラーページをパススルーします。
10. 詳細エラーモード IIS を設定して、ローカルリクエストのみの詳細なエラーメッセージを表示するため、開発者は機密情報を外部ユーザーに公開せずに問題を診断できます。
11. 適切なフォルダアクセス許可を確保する このブロックは、IIS ウェブルート (C:\inetpub\wwwroot) のフォルダアクセス許可を設定します。IIS_IUSRS および Users グループの読み取りおよび実行アクセス許可を設定し、これらのユーザーがフォルダにアクセスできるが、ファイルを変更できないようにします。適切なアクセス許可を設定すると、不正なアクセスやウェブサーバーによってホストされるファイルの改ざんのリスクが最小限に抑えられます。

Windows コンテナの IIS ベストプラクティスに従うことで、コンテナ化されたアプリケーションの安全性、パフォーマンス、スケーラビリティが確保されます。これらのプラクティスは、脆弱性の防止、リソースの使用の最適化、コンプライアンスの確保、コンテナインスタンス間の一貫性の維持に

役立ちます。コンテナは分離されるように設計されていますが、リスクを最小限に抑え、本番環境でのアプリケーションの信頼性を確保するために適切な設定が必要です。

4. 最小権限の原則

最小特権の原則 (PoLP) は、特にセキュリティを強化し、コンテナ化された環境内のリスクを最小限に抑える上で、Windows コンテナにとって重要ないくつかの理由で重要です。この原則は、システムまたはアプリケーションが適切に機能するために必要な最小限のレベルのアクセス許可で動作することを規定しています。Windows コンテナではこれが重要な理由は次のとおりです。

攻撃表面の最小化

- コンテナは、多くの場合、さまざまなシステムコンポーネントとやり取りするアプリケーションを実行し、アプリケーションが持つ権限が多いほど、それらのコンポーネントへのアクセスが広くなります。コンテナのアクセス許可を必要なもののみに制限することで、PoLP は攻撃対象領域を大幅に削減し、攻撃者がコンテナを侵害された場合に悪用することが困難になります。

侵害されたコンテナの影響の制限

- Windows コンテナが侵害された場合、過剰な権限 (管理者またはルートレベルのアクセスなど) を持つアプリケーションを実行すると、攻撃者が重要なシステムファイルを制御したり、コンテナホスト全体で権限を昇格させたりする可能性があります。PoLP を強制することで、コンテナが侵害された場合でも、攻撃者はできることが制限され、さらなる エスカレーションや機密性の高いリソースやその他のコンテナへのアクセスが防止されます。

マルチテナント環境での保護

- クラウド環境またはエンタープライズ環境では、複数のコンテナを同じ物理インフラストラクチャまたは仮想インフラストラクチャで実行できます。PoLP は、侵害されたコンテナが他のテナントに属するリソースやデータにアクセスできないようにします。この分離は、共有マルチテナント環境でセキュリティを維持し、コンテナ間の横方向の移動から保護するために不可欠です。

特権エスカレーションの軽減

- 高い権限で実行されるコンテナは、攻撃者がシステム内の権限を エスカレーションするために使用できます。PoLP は、コンテナのシステムリソースへのアクセスを制限することで、コンテナの環境を超えた不正なアクションや特権の エスカレーションを防止することで、このリスクを軽減します。

コンプライアンスと監査

- 多くの規制標準とセキュリティフレームワーク (PCI DSS、HIPAA、GDPR など) では、機密データへのアクセスを制限するためにシステムが PoLP に準拠する必要があります。制限された権限を持つ Windows コンテナを実行すると、組織はこれらの規制に準拠し、アプリケーションが特に必要なリソースにのみアクセスできるようになります。

設定ミスのリスクを軽減する

- コンテナが不要な権限で実行されると、軽微な設定ミスであっても、重大なセキュリティ脆弱性につながる可能性があります。例えば、管理者として実行されているコンテナが誤ってインターネットに公開された場合、攻撃者はシステムを制御できます。PoLP は、権限が制限されているため、このようなリスクを防ぐのに役立ちます。これにより、設定ミスのリスクが軽減されます。

コンテナのセキュリティ体制の改善

- PoLP に従うことで、コンテナは基盤となるホストシステムと相互により適切に分離されます。これにより、コンテナ化されたアプリケーションが定義された範囲外のシステムファイルやプロセスにアクセスまたは変更する可能性が低くなり、ホストオペレーティングシステムやその他のワークロードの整合性が維持されます。

Dockerfile:

```
# Strongly recommended that when deploying a Windows server container to any multi-
tenant environment that your application runs via the ContainerUser account
USER ContainerUser
```

説明:

このセクションでは、USER ContainerUser コマンドは、Windows コンテナ内のアプリケーションがデフォルトの管理者アカウントではなく ContainerUser アカウントで実行されるように指定します。

これが特にマルチテナント環境で重要である理由は次のとおりです。

1. 最小特権の原則: ContainerUser アカウントは、権限が制限された非管理者ユーザーです。このアカウントでアプリケーションを実行すると、最小特権の原則に従い、悪用のリスクを最小限に抑

えることができます。攻撃者がアプリケーションを侵害した場合、システムへのアクセスが制限され、潜在的な損害が軽減されます。

2. セキュリティの強化: マルチテナント環境では、コンテナは同じ基盤となるインフラストラクチャを共有できます。ContainerUser として実行すると、1 つのコンテナが侵害された場合でも、重要なシステムファイルやその他のコンテナにアクセスまたは変更する管理者権限がなくなります。これにより、攻撃対象領域が大幅に減少します。
3. ルートアクセスの回避: デフォルトでは、コンテナは昇格されたアクセス許可 (Linux コンテナのルートアクセスと同様) で実行される場合があり、悪用されると危険になる可能性があります。ContainerUser を使用すると、アプリケーションが不要な管理権限で実行されないため、攻撃者が権限をエスカレーションすることが困難になります。
4. マルチテナント環境のベストプラクティス: 複数のユーザーまたは組織が同じインフラストラクチャを共有する環境 (クラウドなど) では、セキュリティが重要です。アクセス許可が制限されたアプリケーションを実行すると、あるテナントのアプリケーションが他のテナントに影響を与えるのを防ぎ、プラットフォーム全体で機密データとリソースを保護します。

USER ContainerUser コマンドは、アプリケーションが最小限の権限で実行されるようにし、コンテナが侵害された場合に発生する可能性のある損害を制限することで、マルチテナント環境でのセキュリティを強化します。これは、コンテナ化された環境での不正アクセスや特権のエスカレーションを防ぐためのベストプラクティスです。

最小特権の原則は、セキュリティ違反の潜在的な影響を制限し、攻撃対象領域を減らし、重要なシステムコンポーネントへの不正アクセスを防ぐため、Windows コンテナにとって不可欠です。必要なアクセス許可のみを使用してコンテナ化されたアプリケーションを実行することで、組織はコンテナ環境、特にマルチテナントインフラストラクチャと共有インフラストラクチャのセキュリティと安定性を大幅に強化できます。

最終的な考え方: Windows コンテナの保護が今日の脅威の状況で必須である理由

脅威がより洗練され、豊富になる今日の急速に進化するデジタル世界では、Windows コンテナを保護することは単なる推奨事項ではなく、絶対的な必要性です。コンテナは、アプリケーションをパッケージ化してデプロイするための軽量で柔軟な方法を提供しますが、セキュリティの脆弱性の影響を受けません。インフラストラクチャを合理化するためにコンテナを採用する企業が増えるにつれて、適切に保護されなければ、サイバー攻撃の潜在的なターゲットにもなります。

インターネットには、パッチが適用されていない脆弱性をターゲットとする悪意のある攻撃者から、設定ミスの自動ボットスキャンまで、さまざまな脅威が溢れています。適切なセキュリティ対策が講

じられていないと、コンテナを悪用して機密データを公開したり、権限をエスカレートしたり、より広範なインフラストラクチャを侵害する可能性のある攻撃のエントリポイントとして機能したりする可能性があります。これにより、環境の他の部分を保護するのと同じくらいコンテナのセキュリティが重要になります。

Windows コンテナを使用する場合、多くの従来のセキュリティのベストプラクティスが引き続き適用されます。堅牢なアカウントポリシーの実装、IIS 設定の保護、HTTPS の適用、厳格なファイアウォールルールの使用、重要なファイルへの最小特権アクセスの適用はすべて、コンテナが攻撃に対して回復力を維持する重要な手段です。さらに、定期的な監査と口グ記録により、コンテナ内で何が起こっているかを可視化できるため、疑わしいアクティビティを、本格的なインシデントになる前に検出できます。

Windows コンテナの保護は、機密データの保護とアプリケーションの整合性の確保を義務付ける規制要件にも準拠しています。クラウドネイティブアーキテクチャとコンテナ化されたアーキテクチャが普及するにつれて、ベースイメージから実行中のコンテナまで、すべてのレイヤーでセキュリティを確保すると、運用を保護し、顧客の信頼を維持するのに役立ちます。

要約すると、コンテナ化されたアプリケーションの増加とサイバー脅威の増加により、コンテナセキュリティは最新のインフラストラクチャ管理の交渉不可能な側面になります。ベストプラクティスに従い、脆弱性を継続的にモニタリングすることで、企業はセキュリティを損なうことなく Windows コンテナの俊敏性と効率性を享受できます。この脅威の多い環境では、Windows コンテナの保護は単なるオプションではなく、必須です。

ハイブリッドデプロイのベストプラクティス

このガイドでは、EKS Hybrid Nodes または EKS Anywhere を使用してオンプレミス環境またはエッジ環境でデプロイを実行するためのガイダンスを提供します。

現在、以下のトピックに関するガイドを公開しています。

- EKS Hybrid Nodes とネットワーク切断のベストプラクティス

EKS Hybrid Nodes とネットワーク切断

EKS Hybrid Nodes アーキテクチャは、独自のデータセンターまたはエッジロケーションでローカル Kubernetes クラスターを完全に実行することに慣れているお客様にとって初めてのものです。EKS Hybrid Nodes では、Kubernetes コントロールプレーンは AWS リージョンで実行され、ノードのみがオンプレミスで実行されるため、Kubernetes クラスターアーキテクチャが「拡張」または「拡張」されます。

これは、「ノードが Kubernetes コントロールプレーンから切断されるとどうなりますか？」という一般的な質問につながります。

このガイドでは、以下のトピックを確認して、この質問に答えます。各アプリケーションの動作は依存関係、設定、環境によって異なる可能性があるため、ネットワークの切断を通じてアプリケーションの安定性と信頼性を検証することをお勧めします。EKS Hybrid Nodes と独自のアプリケーションでネットワーク切断をテストするために参照できるテストのセットアップ、手順、結果については、aws-samples/eks-hybrid-examples GitHub リポジトリを参照してください。GitHub リポジトリには、このガイドで説明されている動作の検証に使用されるテストの追加の詳細も含まれています。

- ネットワーク切断による安定性のベストプラクティス
- ネットワーク切断による Kubernetes ポッドのフェイルオーバー動作
- ネットワーク切断によるアプリケーションネットワークトラフィック
- ネットワーク切断による認証情報のホスト

ネットワーク切断による安定性のベストプラクティス

高可用性ネットワーク

ハイブリッドノードと Kubernetes コントロールプレーン間のネットワーク切断を回避する最善の方法は、オンプレミス環境と AWS との間の冗長で回復力のある接続を使用することです。これらのソリューションを使用した高可用性ハイブリッドネットワークの設計の詳細については、AWS Direct Connect Resiliency Toolkit と AWS Site-to-Site VPN のドキュメントを参照してください。

高可用性アプリケーション

アプリケーションを設計するときは、障害ドメインとさまざまなタイプの停止の影響を考慮してください。Kubernetes には、ノード、ゾーン、リージョンのドメイン間でアプリケーションレプリカをデプロイおよび維持する組み込みメカニズムが用意されています。これらのメカニズムの使用は、アプリケーションのアーキテクチャ、環境、可用性の要件によって異なります。例えば、ステートレスアプリケーションは多くの場合、複数のレプリカでデプロイでき、任意のホストとインフラストラクチャ容量を移動できます。また、ノードセレクタとトポロジ分散制約を使用して、異なるドメイン間でアプリケーションのインスタンスを実行できます。Kubernetes で回復力のあるアプリケーションを構築するためのアプリケーションレベルの手法の詳細については、「EKS ベストプラクティスガイド」を参照してください。

Kubernetes は、ポッドを他のノードに移動するかどうかを決定するときに、Kubernetes コントロールプレーンから切断されたノードのゾーン情報を評価します。ゾーン内のすべてのノードに到達できない場合、Kubernetes はそのゾーン内のノードのポッド エビクションをキャンセルします。ベストプラクティスとして、複数のデータセンターまたは物理的な ロケーションでノードが実行されているデプロイがある場合は、データセンターまたは物理的な ロケーションに基づいて各ノードにゾーンを割り当てます。クラウド内のノードで EKS を実行すると、このゾーンラベルは AWS cloud-controller-manager によって自動的に適用されます。ただし、cloud-controller-managerハイブリッドノードでは使用されないため、kubelet 設定を通じてこの情報を渡すことができます。ハイブリッドノードのノード設定でゾーンを設定する方法の例を以下に示します。ハイブリッドノード CLI () を使用してハイブリッドノードをクラスターに接続すると、設定が渡されますnodeadm。`topology.kubernetes.io/zone` ラベルの詳細については、Kubernetes ドキュメントを参照してください。ハイブリッドノード CLI の詳細については、「ハイブリッドノード nodeadm リファレンス」を参照してください。

```
apiVersion: node.eks.aws/v1alpha1
kind: NodeConfig
spec:
  cluster:
```

```
    name: my-cluster
    region: my-region
  kubelet:
    flags:
      - --node-labels=topology.kubernetes.io/zone=dc1
  hybrid:
    ...
```

ネットワークモニタリング

ハイブリッド接続に AWS Direct Connect または AWS Site-to-Site VPN を使用する場合は、CloudWatch アラーム、ログ、メトリクスを活用してハイブリッド接続の状態を観察し、問題を診断できます。詳細については、「AWS Direct Connect リソースのモニタリング」および「AWS Site-to-Site VPN 接続のモニタリング」を参照してください。

EKS コントロールプレーンで実行されている node-lifecycle-controller によって報告されたNodeNotReadyイベントのアラームを作成することをお勧めします。これにより、ハイブリッドノードでネットワークの切断が発生している可能性があります。Controller Manager の EKS コントロールプレーンのログ記録を有効にし、「ノードのステータス変更イベントメッセージの記録」メッセージのメトリクスフィルターを CloudWatch=NodeNotReady」で作成することで、このアラームを作成できます。メトリクスフィルターを作成したら、目的のしきい値に基づいてこのフィルターのアラームを作成できます。詳細については、CloudWatch ドキュメントの「ログのアラーム」を参照してください。

Transit Gateway (TGW) と Virtual Private Gateway (VGW) の組み込みメトリクスを使用して、TGW または VGW との間のネットワークトラフィックを監視できます。これらのメトリクスのアラームを作成して、ハイブリッドノードと EKS コントロールプレーン間の潜在的なネットワーク問題を示すネットワークトラフィックが通常のレベルを下回るシナリオを検出できます。TGW および VGW メトリクスを次の表に示します。

ゲートウェイ	メトリクス	説明
Transit Gateway	BytesIn	TGW がアタッチメント (EKS コントロールプレーンからハイブリッドノード) から受信したバイト数
Transit Gateway	BytesOut	TGW からアタッチメント (ハイブリッドノードから EKS コントロールプレーン) に送信されたバイト数

ゲートウェイ	メトリクス	説明
仮想プライベートゲートウェイ	TunnelDataIn	VPN トンネルを介して接続の AWS 側からカスタマーゲートウェイ (EKS コントロールプレーンからハイブリッドノード) に送信されたバイト数
仮想プライベートゲートウェイ	TunnelDataOut	VPN トンネルを介してカスタマーゲートウェイ (ハイブリッドノードから EKS コントロールプレーン) から接続の AWS 側で受信したバイト数

CloudWatch Network Monitor を使用してハイブリッド接続をより深く把握し、平均復旧時間を短縮し、ネットワークの問題の原因が AWS か環境かを判断することもできます。CloudWatch Network Monitor を使用して、ハイブリッドネットワーク接続のパケット損失とレイテンシーを視覚化し、アラートとしきい値を設定し、ネットワークパフォーマンスを向上させるためのアクションを実行できます。詳細については、Amazon CloudWatch Network Monitor の使用」を参照してください。

EKS には、クラスターとアプリケーションの正常性をモニタリングするためのオプションがいくつか用意されています。クラスターの状態については、EKS コンソールのオブザーバビリティダッシュボードを使用して、問題を迅速に検出、トラブルシューティング、修正できます。Amazon Managed Service for Prometheus、AWS Distro for Open Telemetry (ADOT)、CloudWatch を使用して、クラスター、アプリケーション、インフラストラクチャのモニタリングを行うこともできます。EKS オブザーバビリティオプションの詳細については、「クラスターのパフォーマンスをモニタリングし、ログを表示する」を参照してください。

ローカルトラブルシューティング

ハイブリッドノードと EKS コントロールプレーン間のネットワーク切断に備えるために、セカンダリモニタリングとログバックエンドを設定して、リージョンの AWS サービスに到達できない場合にアプリケーションのオブザーバビリティを維持できます。たとえば、AWS Distro for Open Telemetry (ADOT) コレクターを設定して、メトリクスとログを複数のバックエンドに送信できます。`crictl` CLI などのローカルツールを使用して、通常は Kubernetes API サーバーエンドポイントをクエリする`kubectl`他の Kubernetes API 互換クライアントの代わりとして、ポッドやコンテナをローカルで操作することもできます。の詳細については`crictl`、cri-tools GitHub の`crictl`ドキュメントを参照してください。以下に、いくつかの便利な`crictl`コマンドを示します。

ホストで実行されているポッドを一覧表示します。

```
crictl pods
```

ホストで実行されているコンテナを一覧表示します。

```
crictl ps
```

ホストで実行されているイメージを一覧表示します。

```
crictl images
```

ホストで実行されているコンテナのログを取得します。

```
crictl logs CONTAINER_NAME
```

ホストで実行されているポッドの統計を取得します。

```
crictl statsp
```

アプリケーションネットワークトラフィック

ハイブリッドノードを使用する場合は、アプリケーショントラフィックのネットワークフローと、アプリケーションをクラスターの外部に公開するために使用するテクノロジーを検討して理解することが重要です。アプリケーションの負荷分散と進入のテクノロジーは、ネットワークの切断時に動作が異なります。たとえば、アプリケーションの負荷分散に Cilium の BGP コントロールプレーン機能を使用している場合、ネットワークの切断中にポッドとサービスの BGP セッションがダウンする可能性があります。これは、BGP スピーカー機能が Cilium エージェントと統合されており、Kubernetes コントロールプレーンから切断されると Cilium エージェントが継続的に再起動するためです。再起動の理由は、そのヘルスが Kubernetes コントロールプレーンへのアクセスと組み合わされているため、Cilium のヘルスチェックが失敗したためです (「CFP: #31702 with an opt-in improvement in Cilium v1.17」を参照)。同様に、AWS リージョンが発信するアプリケーショントラフィックに Application Load Balancer (ALB) または Network Load Balancer (NLB) を使用している場合、オンプレミス環境が AWS リージョンへの接続を失った場合、そのトラフィックが一時的にダウンする可能性があります。本稼働環境にデプロイする前に、負荷分散と進入に使用するテクノロジーがネットワークの切断中も安定していることを検証することをお勧めします。aws-samples/eks-hybrid-examples GitHub リポジトリの例では、L2 モードでの負荷分散に MetalLB を使用しています。これは、ハイブリッドノードと EKS コントロールプレーン間のネットワーク切断中も安定しています。

リモート AWS サービスへの依存関係を確認する

ハイブリッドノードを使用する場合は、オンプレミス環境またはエッジ環境の外部にあるリージョンの AWS サービスに対する依存関係に注意してください。例としては、アプリケーションデータ用の Amazon S3 または Amazon RDS へのアクセス、メトリクスとログ用の Amazon Managed Service for Prometheus または CloudWatch の使用、リージョンが発信するトラフィック用の Application and Network Load Balancer の使用、Amazon Elastic Container Registry からのコンテナのプルなどがあります。これらのサービスは、オンプレミス環境と AWS 間のネットワーク切断中はアクセスできません。オンプレミス環境で AWS とのネットワーク切断が発生しやすい場合は、AWS サービスの使用を確認し、それらのサービスへの接続が失われてもアプリケーションの静的安定性が損なわれないことを確認します。

Kubernetes ポッドのフェイルオーバー動作を調整する

ホスト間で移植できないアプリケーションのネットワーク切断中、またはポッドフェイルオーバー用の予備容量がないリソースに制約のある環境では、ポッドフェイルオーバー動作を調整するオプションがあります。一般的に、アプリケーションのリソース要件を考慮し、ノードに障害が発生した場合にアプリケーションの 1 つ以上のインスタンスが別のホストにフェイルオーバーするのに十分な容量を持つことが重要です。

- オプション 1 - DaemonSets を使用する: このオプションは、クラスター内のすべてのノードで実行できるアプリケーションと実行する必要があるアプリケーションに適用されます。DaemonSets は到達不可能なテイントを許容するように自動的に設定されます。これにより、DaemonSet ポッドはネットワーク切断によってノードにバインドされます。
- オプション 2 - 到達不可能なテイントtolerationSecondsに合わせて調整する: ネットワークの切断中にポッドがノードにバインドされたままになる時間を調整できます。これを行うには、アプリケーションポッドを設定して、到達できないテイントを許容し、指定した期間 (tolerationSecondsアプリケーション仕様の) NoExecute効果を持たせます。このオプションでは、ネットワークが切断されると、アプリケーションポッドは tolerationSecondsの有効期限が切れるまでノードにバインドされたままになります。を使用して到達不可能なテイントtolerationSecondsに対して を増やすNoExecuteと、到達不可能なホストで実行されているポッドが他の到達可能な正常なホストに移動するのに時間がかかる可能性があるため、慎重に検討してください。
- オプション 3: カスタムコントローラー: NoExecuteKubernetes で到達不可能なテイントをモニタリングするカスタムコントローラー (またはその他のソフトウェア) を作成して実行できます。このテイントが検出されると、カスタムコントローラーはアプリケーション固有のメトリクスをチェックしてアプリケーションのヘルスを評価できます。アプリケーションが正常である場合、カ

スタムコントローラーは到達不可能なテイントを削除し、ネットワーク切断中のノードからのポッドのエビクションを防ぐことができます。

到達不可能なテイントtolerationSecondsに対して を使用してデプロイを設定する方法の例を以下に示します。この例では、 tolerationSecondsは 1800 (30 分) に設定されています。つまり、到達できないノードで実行されているポッドは、ネットワークの切断が 30 分以上続く場合にのみ削除されます。

```
apiVersion: apps/v1
kind: Deployment
metadata:
...
spec:
...
      tolerations:
      - key: "node.kubernetes.io/unreachable"
        operator: "Exists"
        effect: "NoExecute"
        tolerationSeconds: 1800
```

ネットワーク切断による Kubernetes ポッドのフェイルオーバー

まず、ノードと Kubernetes コントロールプレーン間のネットワーク切断時の Kubernetes の動作に影響する主要な概念、コンポーネント、設定を確認します。EKS はアップストリーム Kubernetes に準拠しているため、ここで説明するすべての Kubernetes の概念、コンポーネント、設定は、EKS および EKS Hybrid Nodes のデプロイに適用されます。

ネットワーク切断中のポッドフェイルオーバー動作を改善するために EKS に特に改善が加えられました。詳細については、アップストリーム Kubernetes リポジトリのGitHub の問題 #131294」と「#131481」を参照してください。

概念

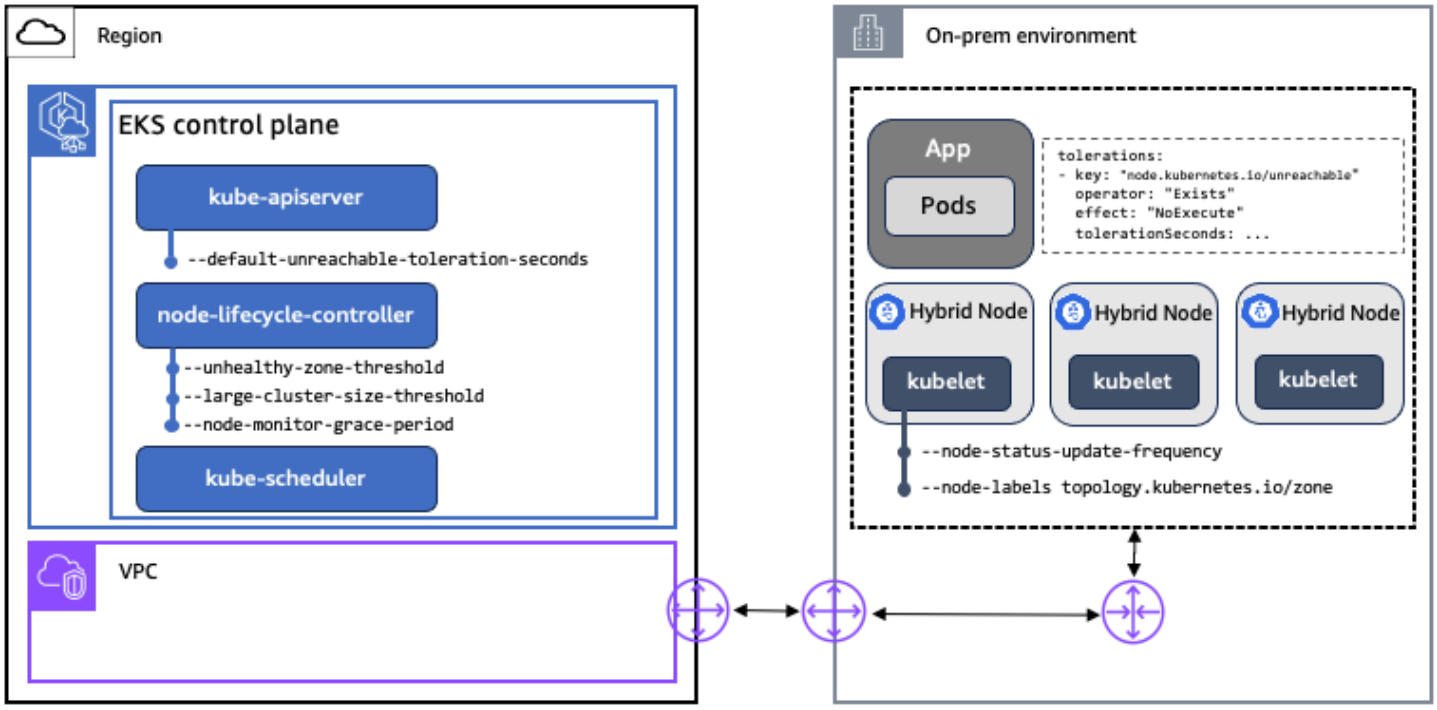
テイントと許容範囲: テイントと許容範囲は、ノードへのポッドのスケジュールを制御するために Kubernetes で使用されます。テイントは node-lifecycle-controller によって設定され、ノードがスケジューリングの対象ではないか、それらのノードのポッドを削除する必要があることを示します。ネットワーク切断が原因でノードに到達できない場合、node-lifecycle-controller は NoSchedule 効果で node.kubernetes.io/unreachable テイントを適用し、特定の条件が満たされた場合は NoExecute 効果を適用します。node.kubernetes.io/unreachable テイントは、NodeCondition Ready が不明であ

ることに対応します。ユーザーは、PodSpec のアプリケーションレベルでテイントの許容値を指定できます。

- NoSchedule: 許容値が一致しない限り、テイントされたノードに新しいポッドはスケジュールされません。ノードで既に実行されているポッドは削除されません。
- NoExecute: テイントを許容しないポッドはすぐに削除されます。(tolerationSeconds を指定せずに) テイントを許容するポッドは、永久にバインドされます。tolerationSeconds が指定されたテイントを許容するポッドは、指定された時間バインドされたままになります。その時間が経過すると、ノードライフサイクルコントローラーはノードからポッドを削除します。

ノードリース: Kubernetes は Lease API を使用して kubelet ノードハートビートを Kubernetes API サーバーに通信します。ノードごとに、一致する名前の Lease オブジェクトがあります。内部的には、各 kubelet ハートビートは Lease オブジェクトの spec.renewTime フィールドを更新します。Kubernetes コントロールプレーンは、このフィールドのタイムスタンプを使用してノードの可用性を判断します。ノードが Kubernetes コントロールプレーンから切断されている場合は、リースの spec.renewTime を更新できず、コントロールプレーンはそれを NodeCondition Ready が不明であると解釈します。

コンポーネント



コンポーネント	サブコンポーネント	説明
Kubernetes コントロールプレーン	kube-api-server	API サーバーは、Kubernetes API を公開する Kubernetes コントロールプレーンのコアコンポーネントです。
Kubernetes コントロールプレーン	node-lifecycle-controller	kube-controller-manager が実行するコントローラーの 1 つ。ノードの問題を検出して対応します。
Kubernetes コントロールプレーン	kube-scheduler	ノードが割り当てられていない新しく作成された Pod を監視し、実行するノードを選択するコントロールプレーンコンポーネント。
Kubernetes ノード	kubelet	クラスター内の各ノードで実行されるエージェント。kubelet は PodSpecs を監視し、それらの PodSpecs で説明されているコンテナが実行中で正常であることを確認します。

構成設定

コンポーネント	設定	説明	K8s デフォルト	EKS デフォルト	EKS で設定可能
kube-api-server	default-unreachable-toleration-seconds	このような許容範囲がまだないすべてのポッドにデフォルトで追加`unreachable:NoExecute` される の許容範囲`tolerationSeconds` の を示します。	300	300	不可
node-lifecycle-controller	node-monitor-grace-period	ノードが異常とマークされるまでに応答しない時間。kubelet の の N 倍にする必要があります。N は`nodeStatusUpdateFrequency` 、kubelet がノードステータス	40	40	不可

コンポーネント	設定	説明	K8s デフォルト	EKS デフォルト	EKS で設定可能
		をポストするために許可される再試行回数です。			
node-lifecycle-controller	large-cluster-size-threshold	node-lifecycle-controller がクラスターをエビクションロジック用に大規模として扱うノードの数。このサイズ以下のクラスターでは、 `--secondary-node-eviction-rate` は 0 に上書きされます。	50	100,000	不可
node-lifecycle-controller	unhealthy-zone-threshold	ゾーン内のノードのうち、そのゾーンが異常として扱われるために準備中である必要があるノードの割合。	55%	55%	不可
kubelet	node-status-update-frequency	kubelet がノードステータスをコントロールプレーンに投稿する頻度。node-lifecycle-controller `nodeMonitorGracePeriod` で と互換性がある必要があります。	10	10	[Yes (はい)]
kubelet	ノードラベル	クラスターにノードを登録するときに追加するラベル。ラベルはハイブリッドノードで指定して、ノードをゾーンにグループ化`topology.kubernetes.io/zone` できます。	なし	なし	はい

ネットワーク切断による Kubernetes ポッドのフェイルオーバー

ここで説明する動作は、ポッドがデフォルト設定で Kubernetes デプロイとして実行されており、EKS が Kubernetes プロバイダーとして使用されていることを前提としています。実際の動作は、環境、ネットワーク切断のタイプ、アプリケーション、依存関係、クラスター設定によって異な

る場合があります。このガイドのコンテンツは、特定のアプリケーション、クラスター設定、プラグインのサブセットを使用して検証されました。本番環境に移行する前に、独自の環境と独自のアプリケーションで動作をテストすることを強くお勧めします。

ノードと Kubernetes コントロールプレーンの間にネットワーク切断がある場合、切断された各ノードの kubelet は Kubernetes コントロールプレーンと通信できません。したがって、kubelet は接続が復元されるまでそれらのノードのポッドを削除できません。つまり、ネットワークの切断前にそれらのノードで実行されているポッドは、切断中に引き続き実行され、他の障害が原因でシャットダウンされないと仮定します。つまり、ノードと Kubernetes コントロールプレーン間のネットワーク切断中に静的安定性を実現できますが、接続が復元されるまでノードまたはワークロードで変更オペレーションを実行することはできません。

ネットワーク切断の性質に基づいて異なるポッドフェイルオーバー動作を生成する主なシナリオは 5 つあります。すべてのシナリオで、ノードが Kubernetes コントロールプレーンに再接続されると、クラスターはオペレーターの介入なしで再び正常になります。以下のシナリオでは、観測結果に基づいて期待される結果を概説しますが、これらの結果は、考えられるすべてのアプリケーションおよびクラスター設定に適用されるとは限りません。

シナリオ 1: クラスターの完全な中断

期待される結果: 到達できないノードのポッドは削除されず、それらのノードで実行され続けます。

クラスターの完全な中断とは、クラスター内のすべてのノードが Kubernetes コントロールプレーンから切断されることを意味します。このシナリオでは、コントロールプレーンの node-lifecycle-controller は、クラスター内のすべてのノードに到達できないことを検出し、ポッドの削除をキャンセルします。

クラスター管理者は、切断`Not Ready`中にステータスが のすべてのノードを表示します。Pod のステータスは変更されず、切断後の再接続中にノードに新しいポッドがスケジュールされることはありません。

シナリオ 2: フルゾーンの中断

期待される結果: 到達できないノードのポッドは削除されず、それらのノードで引き続き実行されます。

フルゾーン中断とは、ゾーン内のすべてのノードが Kubernetes コントロールプレーンから切断されることを意味します。このシナリオでは、コントロールプレーンの node-lifecycle-controller は、ゾーン内のすべてのノードに到達できないことを検出し、ポッドの削除をキャンセルします。

クラスター管理者は、切断Not Ready中にステータスが のすべてのノードを表示します。Pod のステータスは変更されず、切断後の再接続中にノードに新しいポッドがスケジュールされることはありません。

シナリオ 3: メジャーゾーンの中断

期待される結果: 到達できないノードのポッドは削除されず、それらのノードで実行され続けます。

過半数ゾーンの中断は、特定のゾーンのほとんどのノードが Kubernetes コントロールプレーンから切断されることを意味します。Kubernetes のゾーンは、同じtopology.kubernetes.io/zoneラベルを持つノードによって定義されます。クラスターにゾーンが定義されていない場合、大部分の中断はクラスター全体のノードの大部分が切断されることを意味します。デフォルトでは、過半数は node-lifecycle-controller の によって定義されunhealthy-zone-threshold、Kubernetes と EKS の両方で 55% に設定されます。EKS では large-cluster-size-thresholdが 100,000 に設定されているため、ゾーン内のノードの 55% 以上に到達できない場合、ポッドの削除はキャンセルされます (ほとんどのクラスターが 100,000 ノードよりはるかに小さい場合) 。

クラスター管理者は、切断Not Ready中にゾーン内のほとんどのノードのステータスが になりますが、ポッドのステータスは変更されず、他のノードでも再スケジュールされません。

上記の動作は、3 ノードを超えるクラスターにのみ適用されます。3 つ以下のノードのクラスターでは、到達できないノード上のポッドは削除がスケジュールされ、新しいポッドは正常なノードでスケジュールされます。

テスト中に、ゾーンのノードの大部分に到達できない場合でも、ネットワーク切断中にポッドが 1 つの到達不可能なノードから完全に削除されたことが確認されることがあります。この動作の原因として、Kubernetes node-lifecycle-controller で考えられる競合状態をまだ調査中です。

シナリオ 4: マイノリティゾーンの中断

期待される結果: ポッドは到達不可能なノードから削除され、新しいポッドは利用可能な適格なノードでスケジュールされます。

少数中断とは、ゾーン内のノードのごく一部が Kubernetes コントロールプレーンから切断されることを意味します。クラスターにゾーンが定義されていない場合、クラスター全体のノードの少数が切断されることを意味します。前述のように、少数は、デフォルトで 55% の node-lifecycle-controller unhealthy-zone-thresholdの設定によって定義されます。このシナリオでは、ネットワークの切断が default-unreachable-toleration-seconds (5 分) および node-monitor-grace-period (40 秒) より長く、ゾーン内のノードの 55% 未満に到達できない場合、新しいポッドは正常なノードでスケジュールされ、到達できないノードのポッドは削除対象としてマークされます。

クラスター管理者は正常なノードで作成された新しいポッドを表示し、切断されたノードのポッドはとして表示されますTerminating。切断されたノードのポッドには Terminatingステータスがありますが、ノードが Kubernetes コントロールプレーンに再接続するまで完全には削除されないことに注意してください。

シナリオ 5: ネットワーク中断中のノードの再起動

期待される結果: 到達不可能なノードのポッドは、ノードが Kubernetes コントロールプレーンに再接続されるまで開始されません。ポッドフェイルオーバーは、到達できないノードの数に応じて、シナリオ 1～3 で説明されているロジックに従います。

ネットワーク中断中のノードの再起動は、ネットワーク切断と同時にノードで別の障害 (電源サイクル、out-of-memoryイベント、その他の問題など) が発生したことを意味します。ネットワーク切断の開始時にそのノードで実行されていたポッドは、kubelet も再起動した場合、切断中に自動的に再起動されません。kubelet は、起動時に Kubernetes API サーバーにクエリを実行して、実行するポッドを確認します。ネットワークの切断が原因で kubelet が API サーバーに到達できない場合、ポッドの起動に必要な情報を取得できません。

このシナリオでは、CLI crictl などのローカルトラブルシューティングツールを使用して、ポッドを「ブレークグラス」メジャーとして手動で起動することはできません。Kubernetes は通常、失敗したポッドを削除し、既存のポッドを再起動するのではなく、新しいポッドを作成します (詳細については、containerd GitHub リポジトリの #10213 を参照してください) 。静的ポッドは、kubelet によって制御される唯一の Kubernetes ワークロードオブジェクトであり、これらのシナリオ中に再起動できます。ただし、アプリケーションのデプロイに静的ポッドを使用することは一般的にお勧めしません。代わりに、複数のレプリカを異なるホストにデプロイして、ノード障害やノードと Kubernetes コントロールプレーン間のネットワーク切断など、複数の同時障害が発生した場合にアプリケーションの可用性を確保します。

ネットワーク切断によるアプリケーションネットワークトラフィック

このページのトピックは、Kubernetes クラスターネットワークと、ノードと Kubernetes コントロールプレーン間のネットワーク切断中のアプリケーショントラフィックに関連しています。

Cilium

Cilium には、IP アドレス管理 (IPAM)、カプセル化、ロードバランシング、クラスタールーティングのためのいくつかのモードがあります。このガイドで検証されたモードは、クラスタースコープ IPAM、VXLAN オーバーレイ、BGP 負荷分散、kube-proxy を使用していました。Cilium も BGP 負荷分散なしで使用され、MetalLB L2 負荷分散に置き換えられました。

Cilium インストールのベースは、Cilium 演算子と Cilium エージェントで構成されます。Cilium 演算子はデプロイとして実行され、Cilium Custom Resource Definitions (CRDs を登録し、IPAM を管理し、クラスターオブジェクトを https://docs.cilium.io/en/stable/internals/cilium_operator/Kubernetes API サーバーと同期します。Cilium エージェントは、各ノードで DaemonSet として実行され、クラスターで実行されているワークロードのネットワークルールを制御する eBPF プログラムを管理します。

一般的に、Cilium によって設定されたクラスター内ルーティングは、ネットワークの切断中も引き続き利用可能であり、インプレースです。これは、ポッドネットワークのクラスター内トラフィックフローと IP テーブル (iptables) ルールを監視することで確認できます。

```
ip route show table all | grep cilium
```

```
10.86.2.0/26 via 10.86.3.16 dev cilium_host proto kernel src 10.86.3.16 mtu 1450
10.86.2.64/26 via 10.86.3.16 dev cilium_host proto kernel src 10.86.3.16 mtu 1450
10.86.2.128/26 via 10.86.3.16 dev cilium_host proto kernel src 10.86.3.16 mtu 1450
10.86.2.192/26 via 10.86.3.16 dev cilium_host proto kernel src 10.86.3.16 mtu 1450
10.86.3.0/26 via 10.86.3.16 dev cilium_host proto kernel src 10.86.3.16
10.86.3.16 dev cilium_host proto kernel scope link
...
```

ただし、ネットワークの切断中、Cilium オペレーターと Cilium エージェントは、ヘルスチェックと Kubernetes API サーバーへの接続の正常性が結合されているため、再起動します。ネットワークの切断中に Cilium オペレーターと Cilium エージェントのログに以下が表示されることが予想されます。ネットワークの切断中に、 CLI `crictl` などのツールを使用して、ログを含むこれらのコンポーネントの再起動を監視できます。

```
msg="Started gops server" address="127.0.0.1:9890" subsys=gops
msg="Establishing connection to apiserver" host="https://<k8s-cluster-ip>:443"
 subsys=k8s-client
msg="Establishing connection to apiserver" host="https://<k8s-cluster-ip>:443"
 subsys=k8s-client
msg="Unable to contact k8s api-server" error="Get \"https://<k8s-cluster-ip>:443/
api/v1/namespaces/kube-system\": dial tcp <k8s-cluster-ip>:443: i/o timeout"
 ipAddr="https://<k8s-cluster-ip>:443" subsys=k8s-client
msg="Start hook failed" function="client.(*compositeClientset).onStart
 (agent.infra.k8s-client)" error="Get \"https://<k8s-cluster-ip>:443/api/v1/namespaces/
kube-system\": dial tcp <k8s-cluster-ip>:443: i/o timeout"
msg="Start failed" error="Get \"https://<k8s-cluster-ip>:443/api/v1/namespaces/kube-
system\": dial tcp <k8s-cluster-ip>:443: i/o timeout" duration=1m5.003834026s
```

```
msg=Stopping
msg="Stopped gops server" address="127.0.0.1:9890" subsys=gops
msg="failed to start: Get \"https://<k8s-cluster-ip>:443/api/v1/namespaces/kube-system
\": dial tcp <k8s-cluster-ip>:443: i/o timeout" subsys=daemon
```

アプリケーションの負荷分散に Cilium の BGP コントロールプレーン機能を使用している場合、BGP スピーカー機能が Cilium エージェントと統合されており、Kubernetes コントロールプレーンから切断されると Cilium エージェントが継続的に再起動するため、ポッドとサービスの BGP セッションがネットワークの切断中にダウンする可能性があります。詳細については、Cilium ドキュメントの「Cilium BGP コントロールプレーンオペレーションガイド」を参照してください。さらに、電源サイクルやマシンの再起動などのネットワーク切断中に同時障害が発生した場合、Cilium ルートはこれらのアクションを通じて保持されませんが、ノードが Kubernetes コントロールプレーンに再接続して Cilium が再び起動すると、ルートは再作成されます。

Calico

近日公開

MetalLB

MetalLB には、L2 モードと BGP モードの L2 つのロードバランシングモードがあります。 https://metallb.universe.tf/concepts/bgp/これらの負荷分散モードの仕組みとその制限の詳細については、MetalLB ドキュメントを参照してください。このガイドの検証では、L2 モードで MetalLB を使用し、クラスター内の 1 台のマシンが Kubernetes サービスの所有権を取得し、IPv4 の ARP を使用してロードバランサーの IP アドレスをローカルネットワークで到達できるようにしました。MetalLB を実行する場合、IP 割り当てを担当するコントローラーと、割り当てられた IP アドレスを持つ広告サービスを担当する各ノードで実行されるスピーカーがあります。MetalLB コントローラーはデプロイとして実行され、MetalLB スピーカーは DaemonSet として実行されます。ネットワークの切断中、MetalLB コントローラーとスピーカーはクラスターリソースの Kubernetes API サーバーを監視できませんが、実行は続行されます。最も重要なのは、外部接続に MetalLB を使用しているサービスは、ネットワークの切断中も引き続き利用可能でアクセス可能です。

kube-proxy

EKS クラスターでは、kube-proxy は各ノードで DaemonSet として実行され、ネットワークルールを管理して、サービス IP アドレスを基盤となるポッドの IP アドレスに変換することで、サービスとポッド間の通信を有効にします。kube-proxy によって設定された IP テーブル (iptables) ルールは、ネットワークの切断中に維持され、クラスター内ルーティングは引き続き機能し、kube-proxy ポッドは引き続き実行されます。

kube-proxy ルールは、次の iptables コマンドで確認できます。最初のコマンドは、PREROUTINGチェーンを通過するパケットがKUBE-SERVICESチェーンに誘導されることを示しています。

```
iptables -t nat -L PREROUTING
```

```
Chain PREROUTING (policy ACCEPT)
target         prot opt source      destination
KUBE-SERVICES  all  --  anywhere    anywhere      /* kubernetes service portals */
```

KUBE-SERVICES チェーンを調べると、さまざまなクラスターサービスのルールを確認できます。

```
Chain KUBE-SERVICES (2 references)
target                     prot opt source      destination
KUBE-SVL-NZTS37XDTDNXGCKJ  tcp  --  anywhere    172.16.189.136  /* kube-system/hubble-
peer:peer-service cluster IP /
KUBE-SVC-2BINP2AXJOTI3HJ5  tcp  --  anywhere    172.16.62.72    / default/metallb-
webhook-service cluster IP /
KUBE-SVC-LRNEBRA3Z5YGJ4QC  tcp  --  anywhere    172.16.145.111  / default/redis-leader
 cluster IP /
KUBE-SVC-I7SKRZYQ7PWYV5X7  tcp  --  anywhere    172.16.142.147  / kube-system/eks-
extension-metrics-api:metrics-api cluster IP /
KUBE-SVC-JD5MR3NA4I4DYORP  tcp  --  anywhere    172.16.0.10     / kube-system/kube-
dns:metrics cluster IP /
KUBE-SVC-TCOU7JCQXEZGVUNU  udp  --  anywhere    172.16.0.10     / kube-system/kube-
dns:dns cluster IP /
KUBE-SVC-ERIFXISQEP7F7OF4  tcp  --  anywhere    172.16.0.10     / kube-system/kube-
dns:dns-tcp cluster IP /
KUBE-SVC-ENODL3HWJ5BZY56Q  tcp  --  anywhere    172.16.7.26     / default/frontend
 cluster IP /
KUBE-EXT-ENODL3HWJ5BZY56Q  tcp  --  anywhere    <LB-IP>    / default/frontend
 loadbalancer IP /
KUBE-SVC-NPX46M4PTMTKRN6Y  tcp  --  anywhere    172.16.0.1      / default/
kubernetes:https cluster IP /
KUBE-SVC-YU5RV2YQWHLZ5XPR  tcp  --  anywhere    172.16.228.76   / default/redis-
follower cluster IP /
KUBE-NODEPORTS             all  --  anywhere    anywhere        / kubernetes service
 nodeports; NOTE: this must be the last rule in this chain */
```

フロントエンドサービスのチェーンでアプリケーションを調べると、サービスを支えるポッド IP アドレスを確認できます。

```
iptables -t nat -L KUBE-SVC-ENODL3HWJ5BZY56Q
```

```
Chain KUBE-SVC-ENODL3HWJ5BZY56Q (2 references)
target                     prot opt source    destination
KUBE-SEP-EKXE7ASH7Y74BGBO  all  --  anywhere  anywhere    /* default/frontend ->
 10.86.2.103:80 / statistic mode random probability 0.33333333349
KUBE-SEP-GCY3OUXWSVMSEAR6  all  --  anywhere  anywhere    / default/frontend ->
 10.86.2.179:80 / statistic mode random probability 0.50000000000
KUBE-SEP-6GJJR3EF5AUP2WBU  all  --  anywhere  anywhere    / default/frontend ->
 10.86.3.47:80 */
```

次の kube-proxy ログメッセージは、Kubernetes API サーバーがノードおよびエンドポイントリソースの更新を監視しようとしたときに、ネットワークの切断中に予期されます。

```
"Unhandled Error" err="k8s.io/client-go/informers/factory.go:160: Failed to watch
 *v1.Node: failed to list *v1.Node: Get \"https://<k8s-endpoint>/api/v1/nodes?
fieldSelector=metadata.name%3D<node-name>&resourceVersion=2241908\": dial tcp <k8s-
ip>:443: i/o timeout" logger="UnhandledError"
"Unhandled Error" err="k8s.io/client-go/informers/factory.go:160: Failed to watch
 *v1.EndpointSlice: failed to list *v1.EndpointSlice: Get \"https://<k8s-endpoint>/
apis/discovery.k8s.io/v1/endpointslices?labelSelector=%21service.kubernetes.io
%2Fheadless%2C%21service.kubernetes.io%2Fservice-proxy-name&resourceVersion=2242090\":
 dial tcp <k8s-ip>:443: i/o timeout" logger="UnhandledError"
```

CoreDNS

デフォルトでは、EKS クラスターのポッドは、クラスター内 DNS クエリのネームサーバーとして CoreDNS クラスター IP アドレスを使用します。EKS クラスターでは、CoreDNS はノードのデプロイとして実行されます。ハイブリッドノードでは、ハイブリッドノードでローカルに実行されている CoreDNS レプリカがある場合、ポッドはネットワークの切断中に CoreDNS と通信し続けることができます。クラウドにノードを持つ EKS クラスターがあり、オンプレミス環境にハイブリッドノードがある場合は、各環境に少なくとも 1 つの CoreDNS レプリカを持つことをお勧めします。CoreDNS は、ネットワークの切断前に作成されたレコードの DNS クエリの処理を継続し、静的安定性のためにネットワーク再接続を通じて実行を継続します。

Kubernetes API サーバーからオブジェクトを一覧表示しようとすると、ネットワークの切断中に次の CoreDNS ログメッセージが予期されます。

```
Failed to watch *v1.Namespace: failed to list *v1.Namespace: Get "https://<k8s-cluster-
ip>:443/api/v1/namespaces?resourceVersion=2263964": dial tcp <k8s-cluster-ip>:443: i/o
 timeout
Failed to watch *v1.Service: failed to list *v1.Service: Get "https://<k8s-cluster-
ip>:443/api/v1/services?resourceVersion=2263966": dial tcp <k8s-cluster-ip>:443: i/o
 timeout
Failed to watch *v1.EndpointSlice: failed to list *v1.EndpointSlice: Get "https://<k8s-
cluster-ip>:443/apis/discovery.k8s.io/v1/endpointslices?resourceVersion=2263896": dial
 tcp <k8s-cluster-ip>: i/o timeout
```

ネットワーク切断による認証情報のホスト

EKS Hybrid Nodes は、AWS Systems Manager (SSM) ハイブリッドアクティベーションおよび AWS IAM Roles Anywhere と統合され、EKS コントロールプレーンでノードを認証するために使用される一時的な IAM 認証情報を提供します。SSM と IAM Roles Anywhere の両方が、オンプレミスホストで管理する一時的な認証情報を自動的に更新します。クラスター内のハイブリッドノード全体で単一の認証情報プロバイダーを使用することをお勧めします。SSM ハイブリッドアクティベーションまたは IAM Roles Anywhere のいずれかですが、両方を使用することはできません。

SSM ハイブリッドアクティベーション

SSM によってプロビジョニングされた一時的な認証情報は 1 時間有効です。SSM を認証情報プロバイダーとして使用する場合、認証情報の有効期間を変更することはできません。一時的な認証情報は、有効期限が切れる前に SSM によって自動的にローテーションされ、ローテーションはノードまたはアプリケーションのステータスには影響しません。ただし、SSM エージェントと SSM リージョンエンドポイントの間にネットワーク切断がある場合、SSM は認証情報を更新できず、認証情報の有効期限が切れる可能性があります。

SSM リージョンエンドポイントに接続できない場合、SSM は認証情報の更新の再試行にエクスポネンシャルバックオフを使用します。SSM エージェントバージョン `3.3.808.0` 以降 (2024 年 8 月リリース) では、エクスポネンシャルバックオフの上限は 30 分です。ネットワークの切断時間によっては、SSM が認証情報を更新するまでに最大 30 分かかる場合があり、ハイブリッドノードは認証情報が更新されるまで EKS コントロールプレーンに再接続しません。このシナリオでは、SSM エージェントを再起動して認証情報の更新を強制できます。現在の SSM 認証情報の更新動作の副作用として、ノードは、各ノードの SSM エージェントが認証情報の更新を管理するタイミングに応じて、異なるタイミングで再接続することがあります。このため、再接続済みのノードにまだ再接続されていないノードからのポッドフェイルオーバーが発生する可能性があります。

SSM エージェントのバージョンを取得します。SSM コンソールの Fleet Manager セクションを確認することもできます。

```
# AL2023, RHEL
yum info amazon-ssm-agent
# Ubuntu
snap list amazon-ssm-agent
```

SSM エージェントを再起動します。

```
# AL2023, RHEL
systemctl restart amazon-ssm-agent
# Ubuntu
systemctl restart snap.amazon-ssm-agent.amazon-ssm-agent
```

SSM エージェントログを表示します。

```
tail -f /var/log/amazon/ssm/amazon-ssm-agent.log
```

ネットワーク切断中に予想されるログメッセージ:

```
INFO [CredentialRefresher] Credentials ready
INFO [CredentialRefresher] Next credential rotation will be in 29.995040663666668
 minutes
ERROR [CredentialRefresher] Retrieve credentials produced error: RequestError: send
 request failed
INFO [CredentialRefresher] Sleeping for 35s before retrying retrieve credentials
ERROR [CredentialRefresher] Retrieve credentials produced error: RequestError: send
 request failed
INFO [CredentialRefresher] Sleeping for 56s before retrying retrieve credentials
ERROR [CredentialRefresher] Retrieve credentials produced error: RequestError: send
 request failed
INFO [CredentialRefresher] Sleeping for 1m24s before retrying retrieve credentials
```

IAM Roles Anywhere

IAM Roles Anywhere によってプロビジョニングされた一時的な認証情報は、デフォルトで 1 時間有効です。IAM Roles Anywhere プロファイルの `durationSeconds` フィールドを使用して、IAM Roles Anywhere で認証情報の有効期間を設定できます。認証情報の最大有効期間は 12 時間です。Hybrid Nodes IAM ロール `MaxSessionDuration` の設定は、IAM Roles Anywhere プロファイル `durationSeconds` の設定よりも大きくする必要があります。

ハイブリッドノードの認証情報プロバイダーとして IAM Roles Anywhere を使用する場合は、kubelet が `aws_signing_helper credential-process`を呼び出してオンデマンドで認証情報を取得するため、通常、ネットワークの切断後数秒以内に EKS コントロールプレーンに再接続します。ハイブリッドノードやネットワークの切断とは直接関係ありませんが、IAM Roles Anywhere の使用時に証明書の有効期限に関する通知とアラートを設定できます。詳細については、「IAM Roles Anywhere で通知設定をカスタマイズする」を参照してください。

AI/ML ワークロードを実行するためのベストプラクティス

> **Tip**
>
> 今後の Amazon EKS AI/ML イベントやワークショップについては、「Get Hands on with EKS」を参照してください。

EKS で AI/ML ワークロードを実行する際のベストプラクティスを実装することで、それらのワークロードのパフォーマンス、コスト効率、耐障害性、適切なリソースを確保できます。ベストプラクティスは、コンピューティング、ネットワーキング、ストレージ、オブザーバビリティ、パフォーマンスの一般的なセクションに分かれています。

フィードバック

このガイドは、より広範な EKS/Kubernetes コミュニティから直接フィードバックや提案を収集するために、GitHub でリリースされています。ガイドに含める必要があると思われるベストプラクティスがある場合は、GitHub リポジトリに問題をファイルするか、PR を送信してください。新しい機能がサービスに追加されるとき、または新しいベストプラクティスが進化したときに、ガイドを定期的に更新することを意図しています。

コンピューティングと自動スケーリング

GPU リソースの最適化とコスト管理

Well-Known ラベルを使用して GPU 要件でワークロードをスケジュールする

さまざまな GPU 特性 (GPU、GPU メモリなど) に敏感な AI/ML ワークロードでは、Karpenter およびマネージド型ノードグループで使用されるノードタイプでサポートされている既知のスケジューリングラベルを使用して GPU 要件を指定することをお勧めします。これらを定義しないと、GPU リソースが不十分なインスタンスでポッドがスケジュールされ、障害やパフォーマンスの低下が発生する可能性があります。nodeSelector または Node アフィニティを使用して、ポッドを実行するノードを指定し、ポッドのリソースセクションでコンピューティングリソース (CPU、メモリ、GPUs など) を設定することをお勧めします。

例

たとえば、Karpenter を使用するときに GPU ネームノードセレクタを使用するとします。

```
apiVersion: v1
kind: Pod
metadata:
  name: gpu-pod-example
spec:
  containers:
  - name: ml-workload
    image: <image>
    resources:
      limits:
        nvidia.com/gpu: 1  # Request one NVIDIA GPU
  nodeSelector:
    karpenter.k8s.aws/instance-gpu-name: "l40s"  # Run on nodes with NVIDIA L40S GPUs
```

Kubernetes Device Plugin を使用して GPUs

GPUs をノードに公開するには、NVIDIA GPU ドライバーをノードのオペレーティングシステムとコンテナランタイムにインストールし、Kubernetes スケジューラが使用可能な GPUs を持つノードにポッドを割り当てることができるように設定する必要があります。NVIDIA Kubernetes Device Plugin のセットアッププロセスは、使用している EKS Accelerated AMI によって異なります。

- Bottlerocket Accelerated AMI: この AMI には NVIDIA GPU ドライバーが含まれており、NVIDIA Kubernetes Device Plugin がプリインストールされ、すぐに使用できるため、GPU サポートがすぐに利用できます。GPUs を Kubernetes スケジューラに公開するために、追加の設定は必要ありません。
- AL2023 Accelerated AMI: この AMI には NVIDIA GPU ドライバーが含まれていますが、NVIDIA Kubernetes Device Plugin はプリインストールされていません。デバイスプラグインは、通常は DaemonSet を介して個別にインストールおよび設定する必要があります。eksctl を使用してクラスターを作成し、ClusterConfig で GPU インスタンスタイプ (など`g5.xlarge`) を指定すると、`eksctl` は自動的に適切な AMI を選択し、NVIDIA Kubernetes Device Plugin をインストールします。詳細については、eksctl ドキュメントの「GPU サポート」を参照してください。

代わりに EKS Accelerated AMIs と NVIDIA GPU 演算子を使用して NVIDIA Kubernetes デバイスプラグインなどのコンポーネントを管理する場合は、NVIDIA GPU ドライバーと NVIDIA Container ツールキットの管理を、インストール済みの NVIDIA GPU ドライバーと NVIDIA Container Toolkit NVIDIA ドキュメントに従って無効にしてください。

NVIDIA Device Plugin がアクティブで GPUsが正しく公開されていることを確認するには、以下を実行します。

```
kubectl describe node | grep nvidia.com/gpu
```

このコマンドは、nvidia.com/gpuリソースがノードの容量と割り当て可能なリソースにあるかどうかを確認します。たとえば、1 つの GPU を持つノードには が表示されますnvidia.com/gpu: 1。詳細については、Kubernetes GPU スケジューリングガイドを参照してください。

さまざまな EC2 インスタンスタイプを使用する

the section called “データプレーン”「」セクションで説明されているように、Amazon EKS でのスケーラビリティに関する重要なベストプラクティスは、できるだけ多くの異なる EC2 インスタンスタイプを使用することです。この推奨事項は、高速ハードウェア (GPUs など) を使用するインスタンスにも適用されます。1 つのインスタンスタイプのみを使用するクラスターを作成し、リージョンの容量を超えてノード数をスケーリングしようとすると、使用可能なインスタンスがないことを示す容量不足エラー (ICE) が表示されることがあります。任意に分散する前に、AI/ML ワークロードの固有の特性を理解することが重要です。EC2 Instance Type Explorer ツールを使用して使用可能なインスタンスタイプを確認し、特定のコンピューティング要件に一致するインスタンスタイプのリストを生成し、クラスターで使用できるインスタンスのタイプを任意に制限しないようにします。

高速コンピューティングインスタンスは、短期、中期、定常状態のワークロードに合わせて、さまざまな購入モデルで提供されています。予約を避けたい短期的で柔軟で耐障害性のあるワークロードについては、スポットインスタンスをご覧ください。キャパシティブロック、オンデマンドインスタンス、Saving Plans を使用すると、中長期のワークロード期間にわたって高速コンピューティングインスタンスをプロビジョニングできます。任意の購入オプションで必要な容量に正常にアクセスする可能性を高めるために、インスタンスタイプとアベイラビリティーゾーンのさまざまなリストを使用することをお勧めします。または、特定の購入モデルの ICEs が発生した場合は、別のモデルを使用して再試行します。

例 次の例は、Karpenter NodePool が世代 3 より大きい G および P インスタンス (p3 など) をプロビジョニングできるようにする方法を示しています。詳細については、「スケーラビリティ」のセクションを参照してください。

```
- key: karpenter.k8s.aws/instance-category
  operator: In
  values: ["g", "p"] # Diversifies across G-series and P-series
- key: karpenter.k8s.aws/instance-generation
  operator: Gt
  values: ["3"] # Selects instance generations greater than 3
```

GPUs、以下のAmazon EC2 スポットインスタンスを Karpenter で GPUs」を参照してください。

Karpenter で GPUs に Amazon EC2 スポットインスタンスを使用することを検討する

Amazon EC2 スポットインスタンスを使用すると、AWS クラウド内の未使用の EC2 容量を利用でき、オンデマンド料金と比較して最大 90% の割引で利用できます。Amazon EC2 スポットインスタンスは、EC2 が容量を戻す必要がある場合、2 分間の通知で中断できます。詳細については、Amazon EC2 ユーザーガイド」の「スポットインスタンス」を参照してください。Amazon EC2 スポットは、耐障害性、ステートレス、柔軟性 (時間およびインスタンスタイプ) の高いワークロードに最適です。スポットインスタンスを使用するタイミングの詳細については、EC2 スポットインスタンスのベストプラクティス」を参照してください。スポットフレンドリであれば、AI/ML ワークロードにスポットインスタンスを使用することもできます。

ユースケース

スポットフレンドリーなワークロードには、ビッグデータ、コンテナ化されたワークロード、CI/CD、ステートレスウェブサーバー、ハイパフォーマンスコンピューティング (HPC)、レンダリングワークロードなどがあります。スポットインスタンスは、柔軟性がない、ステートフル、耐障害性がある、またはインスタンスノード間で緊密に結合されているワークロードには適していません (例えば、計算のために相互に大きく依存する並列プロセスを持つワークロードで、計算流体力学や複雑な相互依存関係を持つ分散データベースなどの MPI ベースの高性能コンピューティングアプリケーションなど、一定のノード間通信が必要です) 。推奨される特定のユースケースを以下に示します (特定の順序はありません) 。

- リアルタイムオンライン推論: スポットインスタンスを使用して、ワークロードがスポットフレンドリである限り、リアルタイム推論ワークロードのコスト最適化スケーリングを行います。つまり、推論時間は 2 分未満で、アプリケーションは中断に対して耐障害性があり、さまざまなインスタンスタイプで実行できます。潜在的なスポット中断を処理するアプリケーションレベルの耐障害性を実装しながら、インスタンスの多様性 (複数のインスタンスタイプやアベイラビリティーゾーンなど) や予約を通じて高可用性を確保します。
- ハイパーパラメータ調整: スポットインスタンスを使用して、特に短時間の実験では、中断が大きな損失なしで許容されるため、探索的な調整ジョブを日和見的に実行します。
- データ拡張: スポットインスタンスを使用して、中断された場合にチェックポイントから再起動できるデータ前処理および拡張タスクを実行するため、スポットの可変可用性に最適です。
- ファインチューニングモデル: スポットインスタンスを使用して、堅牢なチェックポイントメカニズムでファインチューニングを行い、最後に保存された状態から再開して、インスタンスの中断の影響を最小限に抑えます。

- バッチ推論: スポットインスタンスを使用して、オフライン推論リクエストの大量のバッチをnon-real-time方法で処理します。ジョブを一時停止および再開でき、スポットのコスト削減と最適な連携を提供し、再試行や分散による潜在的な中断を処理します。
- オポチュニスティックトレーニングサブセット: わずかなトレーニングワークロードや実験的なトレーニングワークロード (1,000 万パラメータ未満の小規模なモデルなど) にはスポットインスタンスを使用します。この場合、中断は許容され、インスタンスタイプやリージョン間の分散などの効率の最適化を適用できますが、中断の可能性があるため、本番規模のトレーニングにはお勧めしません。

考慮事項

Amazon EKS の高速ワークロードにスポットインスタンスを使用するには、いくつかの重要な考慮事項があります (特定の順序はありません)。

- Karpenter を使用して、高度な統合を有効にしたスポットインスタンスを管理します。Karpenter NodePool で karpenter.sh/capacity-type を「スポット」として指定することで、Karpenter は追加の設定なしでデフォルトでスポットインスタンスをプロビジョニングします。ただし、使用率の低いスポットノードを低価格のスポット代替に置き換える高度なSpot-to-Spot統合を有効にするには、Karpenter コントローラー引数で --feature-gates SpotToSpotConsolidation=true を設定するか、FEATURE_GATES 環境変数を使用して SpotToSpotConsolidation 機能ゲートを有効にする必要があります。Karpenter は、price-capacity-optimized配分戦略を使用して EC2 インスタンスをプロビジョニングします。NodePool の要件とポッドの制約に基づいて、Karpenter はスケジュール不可能なポッドをパッケージ化し、さまざまなインスタンスタイプのセットを Amazon EC2 フリート API に送信します。EC2 Instance Type Explorer ツールを使用して、特定のコンピューティング要件に一致するインスタンスタイプのリストを生成できます。
- ワークロードがステートレスで耐障害性があり、柔軟性があることを確認します。ワークロードは、ステートレスで耐障害性があり、インスタンス/GPU サイズに関して柔軟である必要があります。これにより、スポットの中断後にシームレスに再開でき、インスタンスの柔軟性により、スポットに長時間とどまる可能性があります。Karpenter でスポット中断処理を有効にするには、AWS SQS キューの名前を使用して settings.interruptionQueue Helm 値を設定し、スポット中断イベントをキャッチします。例えば、Helm 経由でインストールする場合は、--set "settings.interruptionQueue=${CLUSTER_NAME}" を使用します。例については、「Karpenter 入門ガイド」を参照してください。Karpenter は、スポット中断イベントに気付くと、中断イベントの前にノードを自動的にコーディング、テイント、ドレイン、終了 (複数可) して、ポッドの終了猶予期間を最大化します。同時に、Karpenter はすぐに新しいノードを起動し、できるだけ早く準備できるようにします。

- インスタンスタイプの選択を過度に制限しないようにします。インスタンスタイプをできるだけ制限しないでください。インスタンスタイプを制限しないと、スポットインスタンスの中断頻度が低く、低コストで大規模なスポット容量を取得する可能性が高くなります。例えば、特定のタイプ (g5.xlarge など) に制限することは避けてください。karpenter.k8s.aws/instance-category や karpenter.k8s.aws/instance-generation などのキーを使用して、さまざまなインスタンスカテゴリと世代のセットを指定することを検討してください。Karpenter を使用すると、複数のインスタンスタイプとアベイラビリティーゾーン (AZs) 間でオンデマンドインスタンス容量とスポットインスタンス容量を簡単に分散できます。さらに、AI/ML ワークロードが特定のアクセラレーターまたは限られたアクセラレーターを必要とするが、リージョン間で柔軟性がある場合は、スポットプレイスメントスコアを使用して、起動前にワークロードをデプロイする最適なリージョンを動的に特定できます。
- NodePool の要件を拡張して、同様の EC2 インスタンスファミリーの数を増やします。すべてのスポットインスタンスプールは、特定のアベイラビリティーゾーン (AZ) 内の特定のインスタンスタイプの未使用の EC2 インスタンス容量で構成されます。Karpenter が新しいノードのプロビジョニングを試みると、NodePool の要件に一致するインスタンスタイプが選択されます。互換性のあるインスタンスタイプに AZ のスポット容量がない場合、プロビジョニングは失敗します。この問題を回避するには、GPU メモリやレイトレーシングなどのハードウェアのニーズを考慮しながら、サイズとアベイラビリティーゾーン (AZs) 間で NVIDIA のより広範な g シリーズインスタンス (世代 4 以降) を許可します。インスタンスのタイプは異なる可能性があるため、ワークロードが各タイプで実行でき、得られるパフォーマンスがニーズを満たすことを確認する必要があります。
- リージョン内のすべてのアベイラビリティーゾーンを活用します。使用可能な容量はアベイラビリティーゾーン (AZ) によって異なります。特定のインスタンスタイプは、ある AZ では使用できないものの、別の AZ では豊富である可能性があります。インスタンスタイプとアベイラビリティーゾーンの一意の組み合わせはそれぞれ、個別のスポットキャパシティープールを構成します。Karpenter NodePool 要件内のリージョン内のすべての AZs で容量をリクエストすることで、より多くのプールを一度に効果的に検索できます。これにより、スポットキャパシティープールの数を最大化できるため、スポットキャパシティーを取得する可能性が高くなります。これを実現するには、NodePool 設定で、topology.kubernetes.io/zone キーを完全に省略して、Karpenter がリージョンで使用可能なすべての AZs から選択できるようにするか、 演算子を使用して AZs を明示的に一覧表示します。 で値を指定します (us-west-2a など) 。
- スポットプレイスメントスコア (SPS) を使用して、スポットインスタンスを使用して必要な容量に正常にアクセスする可能性を可視化することを検討してください。スポットプレイスメントスコア (SPS) は、スポットリクエストが成功する可能性を評価するのに役立つスコアを提供するツールです。SPS を使用する場合は、まずスポットインスタンスのコンピューティング要件を指定し、Amazon EC2 はスポットリクエストが成功する可能性が高い上位 10 のリージョンまたはアベ

イラビリティーゾーン (AZs) を返します。リージョンとアベイラビリティーゾーンは1 ~ 10 のスケールで採点されます。スコアが 10 の場合、スポットリクエストは成功する可能性は高くなりますが、成功が保証されるわけではありません。スコア 1 はスポットリクエストがまったく成功する可能性がないことを示します。異なるリージョンまたはアベイラビリティーゾーンで同じスコアが返される場合があります。詳細については、「AWS でスポットプレイスメントスコアトラッカーダッシュボードを構築するためのガイダンス」を参照してください。スポット容量は常に変動するため、SPS は、ワークロードの制約 (柔軟性、パフォーマンス、サイズなど) に最適なインスタンスタイプ、AZs、リージョンの組み合わせを特定するのに役立ちます。AI/ML ワークロードが特定のアクセラレーターまたは限られたアクセラレーターを必要とするが、リージョン間で柔軟性がある場合は、スポットプレイスメントスコアを使用して、起動前にワークロードをデプロイする最適なリージョンを動的に特定できます。スポット容量を取得する可能性を自動的に確認できるように、SPS トラッカーダッシュボードを構築するためのガイダンスを提供します。このソリューションは、さまざまなセットアップ (GPUs) の YAML 設定を使用して SPS スコアを経時的にモニタリングし、CloudWatch にメトリクスを保存し、設定を比較するためのダッシュボードを提供します。ワークロードごとにダッシュボードを定義して vCPU、メモリ、GPU のニーズを評価し、他の AWS リージョンの使用を検討するなど、EKS クラスターの最適なセットアップを確保します。詳細については、「スポットプレイスメントスコアの仕組み」を参照してください。

- スポットの中断とテストを適切に処理します。終了期間が 2 分を超えるポッドの場合、古いノードはそれらのポッドが再スケジュールされる前に中断され、ワークロードの可用性に影響する可能性があります。アプリケーションを設計するときは、2 分間のスポット中断通知を検討してください。 長時間実行されるアプリケーション (例: Amazon S3) などの永続的ストレージに進行状況を保存して、中断後に再開します。 Pod 仕様で terminationGracePeriodSeconds (デフォルトは 30 秒) を拡張して、正常なシャットダウンにより多くの時間を確保します。 および preStop、クリーンアップ、 状態の保存、 および 接続クロージャ。スケーリング時間が重要で、アプリケーションのトラフィック処理に 2 分以上かかるリアルタイムワークロードの場合は、 と the section called “パフォーマンス”のベストプラクティスを確認して、コンテナの起動時間と ML モデルのロード時間を最適化することを検討the section called “[Storage (ストレージ)]”してください。代替ノードをテストするには、AWS Fault Injection Service (FIS) を使用してスポットの中断をシミュレートします。

これらのコアスポットのベストプラクティスに加えて、Amazon EKS で GPU ワークロードを管理する際には、これらの要素を考慮してください。CPU ベースのワークロードとは異なり、GPU ワークロードは GPU 機能や使用可能な GPU メモリなどのハードウェアの詳細に特に敏感です。GPU ワークロードは、使用できるインスタンスタイプによって制約される場合があり、CPUs。最初のステップとして、ワークロードがインスタンスに柔軟性があるかどうかを評価します。ワークロードで使用できるインスタンスタイプがわからない場合は、個別にテストして互換性と機能を確認します。分散

によってワークロードが動作し続け、パフォーマンスへの影響 (スループットや完了時間など) を把握しながら、可能な限り分散できる柔軟性を特定します。ワークロードの多様化の一環として、次の点を考慮してください。

- CUDA とフレームワークの互換性を確認します。GPU ワークロードは、特定のハードウェア、GPU タイプ (p3 の V100 と p4 の A100 など) 、または TensorFlow などのライブラリの特定の CUDA バージョン用に記述される場合があるため、ワークロードの互換性を確認してください。この互換性は、ランタイムエラー、クラッシュ、GPU アクセラレーションの障害 (PyTorch や TensorFlow などのフレームワークとの CUDA バージョンの不一致により実行が妨げられる可能性があるなど) や、FP16/INT8 精度などのハードウェア機能を活用する機能を防止するために不可欠です。
- GPU メモリ。DCGM Exporter などのツールを使用して、モデルのメモリ要件を評価し、ランタイム中にモデルのメモリ使用量をプロファイリングし、インスタンスタイプに必要な最小 GPU メモリを karpenter.k8s.aws/instance-gpu-memory などの既知のラベルに設定します。GPU VRAM はインスタンスタイプ (NVIDIA T4 が 16GB, A10G が 24GB, V100 が 16-32GB など) によって異なり、ML モデル (大規模言語モデルなど) が使用可能なメモリを超え、out-of-memory (OOM) エラーやクラッシュが発生する可能性があります。EKS のスポットインスタンスでは、分散が制限される場合があります。例えば、モデルが適合しない場合、低 VRAM タイプを含めることはできません。これにより、キャパシティプールへのアクセスが制限され、中断リスクが増加する可能性があります。単一 GPU の場合、単一ノード推論 (GPU リソースを利用するために同じノードで複数のポッドがスケジュールされているなど) では、スポット設定に十分な VRAM を持つインスタンスタイプのみを含めることができるため、分散が制限される場合があります。
- 浮動小数点の精度とパフォーマンス。すべての Nvidia GPU アーキテクチャが同じ浮動小数点精度 (FP16/INT8 など) を持つわけではありません。ワークロードに必要なコアタイプ (CUDA/Tensor/RT) のパフォーマンスと浮動小数点精度を評価します。低価格でパフォーマンスの低い GPU で実行しても、パフォーマンスは向上しないため、分散の影響を理解するために、特定の期間内に完了した作業の観点からパフォーマンスを評価することを検討してください。

シナリオ: リアルタイム推論ワークロードの分散

スポットインスタンスのリアルタイムのオンライン推論ワークロードの場合、互換性のある GPU インスタンスファミリーと世代を分散するように Karpenter NodePool を設定できます。このアプローチにより、GPU の機能、メモリ、アーキテクチャに対する制約を通じてパフォーマンスを維持しながら、複数のスポットプールから高可用性を確保できます。インスタンス容量が制限されている場合の代替手段の使用をサポートし、中断を最小限に抑え、推論レイテンシーを最適化します。この NodePool 状態の例では、GPU メモリが 20GB を超える 3 を超える g および p シリーズインスタンスを使用します。

例

```
apiVersion: karpenter.sh/v1
kind: NodePool
metadata:
  name: gpu-inference-spot
spec:
  template:
    metadata:
      labels:
        role: gpu-spot-worker
    spec:
      requirements:
        - key: karpenter.sh/capacity-type
          operator: In
          values: ["spot"] # Use Spot Instances
        - key: karpenter.k8s.aws/instance-category
          operator: In
          values: ["g", "p"] # Diversifies across G-series and P-series
        - key: karpenter.k8s.aws/instance-generation
          operator: Gt
          values: ["3"] # Selects instance generations greater than 3
        - key: kubernetes.io/arch
          operator: In
          values: ["amd64"] # Specifies AMD64 architecture, compatible with NVIDIA GPUs
        - key: karpenter.k8s.aws/instance-gpu-memory
          operator: Gt
          values: ["20480"] # Ensures more than 20GB (20480 MiB) total GPU memory
      taints:
        - key: nvidia.com/gpu
          effect: NoSchedule
      nodeClassRef:
        name: gpu-inference-ec2
        group: karpenter.k8s.aws
        kind: EC2NodeClass
      expireAfter: 720h
  limits:
    cpu: 100
    memory: 100Gi
  disruption:
    consolidationPolicy: WhenEmptyOrUnderutilized
    consolidateAfter: 5m # Enables consolidation of underutilized nodes after 5 minutes
```

長時間実行されるトレーニングジョブのチェックポイントを実装する

チェックポイントは、プロセスの状態を定期的に保存し、中断した場合に最後に保存したポイントから再開できるようにする耐障害性手法です。機械学習では、一般的にトレーニングに関連付けられており、長時間実行されるジョブによってモデルの重みとオプティマイザの状態が節約され、ハードウェアの問題やスポットインスタンスの中断などの障害後にトレーニングを再開できます。

チェックポイントを使用して、トレーニング中に機械学習 (ML) モデルの状態を保存します。チェックポイントはモデルのスナップショットであり、機械学習フレームワークのコールバック関数を使用して設定できます。保存されたチェックポイントを使用して、トレーニングジョブを最後に保存したチェックポイントから再開できます。チェックポイントを使用すると、トレーニングジョブまたはインスタンスの予期しない中断により、モデルスナップショットをトレーニング中に保存します。これにより、チェックポイントから将来モデルのトレーニングを再開できます。ノードレジリエンシーシステムの実装に加えて、チェックポイントを実装して、ハードウェア障害や Amazon EC2 スポットインスタンスの中断などによる中断の影響を軽減することをお勧めします。

チェックポイントがないと、中断により計算時間が無駄になり、進行状況が失われる可能性があり、長時間実行されるトレーニングジョブにはコストがかかります。チェックポイントを使用すると、ジョブは定期的に状態 (モデルの重みやオプティマイザの状態など) を保存し、中断後に最後のチェックポイント (最後に処理されたバッチ) から再開できます。チェックポイントを実装するには、大量のバッチでデータを処理し、トレーニングジョブの進行中に Mountpoint for Amazon S3 CSI ドライバーを介して Amazon S3 バケットなどの中間結果を永続的ストレージに保存するようにアプリケーションを設計します。 Amazon S3

ユースケース

チェックポイントは、耐障害性とパフォーマンスのオーバーヘッドのバランスをとるために、特定のシナリオで特に有益です。次の場合はチェックポイントの使用を検討してください。

- ジョブ時間が数時間を超える: 長時間実行されるトレーニングジョブの場合 (小規模モデルの場合は >1 ~ 2 時間、数十億のパラメータを持つ大規模基盤モデルの場合は日/週など) 、中断による進行状況の損失にはコストがかかります。ジョブを短くすると、I/O オーバーヘッドが正当化されない場合があります。
- スポットインスタンスまたはハードウェア障害の場合: EC2 スポット (2 分前の通知) やハードウェア障害 (GPU メモリエラーなど) などの中断が発生しやすい環境では、チェックポイントにより迅速な再開が可能になり、耐障害性のあるワークロードでスポットを有効にしてコストを削減できます。

- 大規模な分散トレーニング: 数百/数千のアクセラレーター (>100 GPUs) のセットアップの場合、障害間の平均時間はスケールに応じて直線的に減少します。モデル/データ並列処理に を使用してチェックポイントへの同時アクセスを処理し、完全な再起動を回避します。
- リソース需要の高い大規模モデル: クラスターサイズが原因で障害が避けられないペタバイト規模の LLM トレーニングでは、階層型アプローチ (一時的な場合は 5～30 分ごとに高速ローカル、重大な障害の場合は 1 時間ごとに耐久性) により、復旧時間と効率が最適化されます。

P インスタンスと Trainium インスタンスの容量保証に ML キャパシティブロックを使用する

ML のキャパシティブロックを使用すると、需要の高い GPU インスタンス、特に P インスタンス (p6-b200、p5、p5e、p5en、p4d、p4de など) と Trainium インスタンス (trn1、trn2 など) を予約して、短期間の機械学習 (ML) ワークロードをサポートするためにほぼ即時または将来の日付に開始できます。これらの予約は、モデルトレーニングやファインチューニングなどの計算負荷の高いタスクの容量を確保するのに最適です。EC2 キャパシティブロックの料金は、予約料金とオペレーティングシステム料金で構成されます。料金の詳細については、「ML 料金の EC2 キャパシティブロック」を参照してください。

予測可能なキャパシティ保証のために Amazon EKS の AI/ML ワークロード用の GPUs を予約するには、短期またはオンデマンドキャパシティ予約 (ODCR) に ML キャパシティブロックを活用して汎用キャパシティ保証を行うことをお勧めします。 ODCRs

- ODCRsを使用すると、特定のアベイラビリティーゾーンの EC2 インスタンス容量 (g5 や p5 などの GPU インスタンスなど) を一定期間予約できるため、需要が高い場合でも可用性を確保できます。ODCRsには長期的なコミットメントはありませんが、使用の有無にかかわらず、リザーブドキャパシティのオンデマンド料金を支払います。EKS では、ODCRs は Karpenter やマネージド型ノードグループなどのノードタイプでサポートされています。Karpenter で ODCRs に優先順位を付けるには、 `capacityReservationSelectorTerms`フィールドを使用するように NodeClass を設定します。Karpenter NodePools ドキュメントを参照してください。
- キャパシティブロックは、GPU (p5、p4d など) または Trainium (trn1、trn2) インスタンス専用の予約メカニズムであり、モデルトレーニング、ファインチューニング、実験などの短期 ML ワークロード向けに設計されています。将来の日付から始まる定義された期間 (通常は 24 時間～182 日) のキャパシティーを予約し、予約時間に対してのみ支払います。これらは前払いであり、容量のニーズに応じて事前に計画する必要があり、自動スケーリングをサポートしていませんが、低レイテンシーのネットワーキングのために EC2 UltraClusters にコロケーションされています。予約期間に対してのみ課金されます。詳細については、「キャパシティブロックの検索と購入」を参照す

るか、「Create a managed node group with Capacity Blocks for ML」の手順を使用してキャパシティブロックでマネージド型ノードグループを設定して開始してください。

AWS マネジメントコンソールを使用してキャパシティを予約し、ML キャパシティブロックを使用するようにノードを設定します。ワークロードスケジュールに基づいて予約を計画し、ステージングクラスターでテストします。詳細については、キャパシティブロックのドキュメントを参照してください。

G Amazon EC2 インスタンスのオンデマンド、Amazon EC2 スポット、またはオンデマンドキャパシティ予約 (ODCRs) を検討する

G Amazon EC2 インスタンスでは、オンデマンド、Amazon EC2 スポットインスタンス、オンデマンドキャパシティ予約とは異なる購入オプションを検討してください。ODCRsを使用すると、特定のアベイラビリティーゾーンの EC2 インスタンス容量を一定期間予約できるため、需要が高い場合でも可用性を確保できます。P インスタンスと Trainium インスタンスでのみ使用できる ML キャパシティブロックとは異なり、ODCRs は G インスタンスを含む幅広いインスタンスタイプに使用できるため、推論やグラフィックスなど、さまざまな GPU 機能を必要とするワークロードに適しています。Amazon EC2 スポットインスタンスを使用する場合、さまざまなインスタンスタイプ、サイズ、アベイラビリティーゾーンに分散できることが、スポットに長く留まるための鍵となります。

ODCRsには長期的なコミットメントはありませんが、使用の有無にかかわらず、リザーブドキャパシティのオンデマンド料金を支払います。ODCRs はすぐに使用できるように作成することも、将来の日付にスケジュールすることもできます。これにより、キャパシティプランニングを柔軟に行うことができます。Amazon EKS ODCRs は Karpenter やマネージド型ノードグループなどのノードタイプでサポートされています。Karpenter で ODCRs に優先順位を付けるには、`capacityReservationSelectorTerms`フィールドを使用するように NodeClass を設定します。Karpenter NodePools ドキュメントを参照してください。CLI コマンドを含む ODCRs「オンデマンドキャパシティ予約の開始方法」を参照してください。

他の高速インスタンスタイプとサイズを検討する

Amazon EKS の ML ワークロードのパフォーマンスとコストの両方を最適化するには、適切な高速インスタンスとサイズを選択することが不可欠です。例えば、異なる GPU インスタンスファミリーには、GPU メモリなどの異なるパフォーマンスと機能があります。最も価格パフォーマンスの高いオプションを選択するには、高速コンピューティングの EC2 インスタンスタイプページで使用可能な GPU インスタンスを確認してください。複数のインスタンスタイプとサイズを評価して、特定のワークロード要件に最適なものを見つけます。GPUs の数、メモリ、ネットワークパフォーマン

スなどの要因を考慮してください。適切な GPU インスタンスタイプとサイズを慎重に選択することで、EKS クラスターのリソース使用率とコスト効率を向上させることができます。

EKS ノードで GPU インスタンスを使用する場合、デフォルトでkube-systemは名前空間にnvidia-device-plugin-daemonsetポッドがあります。インスタンスで GPU (複数可) を最大限に活用しているかどうかをすばやく把握するには、次に示すように nvidia-smi を使用できます。

```
kubectl exec nvidia-device-plugin-daemonset-xxxxx \
  -n kube-system -- nvidia-smi \
  --query-
gpu=index,power.draw,power.limit,temperature.gpu,utilization.gpu,utilization.memory,memory.free
 \
  --format=csv -l 5
```

- utilization.memory が 100% に近い場合、コード (複数可) はメモリバインドである可能性があります。つまり、GPU (メモリ) は完全に利用されていますが、パフォーマンスの最適化をさらに調査する必要があることを示唆している可能性があります。
- utilization.gpu が 100% に近い場合、GPU が完全に使用されているとは限りません。より適切なメトリクスは、 power.drawと の比率ですpower.limit。この比率が 100% 以上の場合、コード (複数可) は GPU のコンピューティング性能を完全に活用しています。
- -l 5 フラグは、メトリクスを 5 秒ごとに出力するように に指示します。単一の GPU インスタンスタイプの場合、インデックスクエリフラグは必要ありません。

詳細については、AWS ドキュメントの「GPU インスタンス」を参照してください。

タイムスライス、MIG、フラクショナル GPU 割り当てを使用して GPU リソース割り当てを最適化する

Kubernetes の静的リソース制限 (CPU、メモリ、GPU 数など) は、特に推論などの動的な AI/ML ワークロードでは、過剰プロビジョニングや使用率の低下につながる可能性があります。適切な GPU を選択することは重要です。少量または急増するワークロードの場合、タイムスライスにより、コンピューティングリソースを共有することで複数のワークロードが単一の GPU を共有できるため、効率が向上し、無駄が削減される可能性があります。GPU 共有は、さまざまなオプションを通じて実現できます。

- ノードセレクタ/ノードアフィニティを活用してスケジューリングに影響を与える: プロビジョニングされたノードとポッドがワークロードに適した GPUs でスケジュールされていることを確認します (例: karpenter.k8s.aws/instance-gpu-name: "a100")

- タイムスライス: GPU のコンピューティングリソースを経時的に共有するようにワークロードをスケジュールし、物理パーティショニングなしで同時実行できるようにします。これは、コンピューティング要求が変化するワークロードに最適ですが、メモリの分離がない可能性があります。
- マルチインスタンス GPU (MIG): MIG では、1 つの NVIDIA GPU を複数の独立したインスタンスに分割でき、NVIDIA Ampere (A100 GPU など)、NVIDIA Hopper (H100 GPU など)、NVIDIA Blackwell (Blackwell GPUs など) GPUs。各 MIG インスタンスは専用のコンピューティングリソースとメモリリソースを受け取るため、マルチテナント環境やリソースの保証を必要とするワークロードでリソースを共有できます。これにより、タイムスライスによってバッチサイズが異なる複数のモデルを提供するシナリオなど、GPU リソースの使用率を最適化できます。
- 小数 GPU 割り当て: ソフトウェアベースのスケジューリングを使用して GPU のコンピューティングまたはメモリの一部をワークロードに割り当て、動的ワークロードに柔軟性を提供します。Run:ai プラットフォームの一部である NVIDIA KAI スケジューラは、ポッドが小数 GPU リソースをリクエストできるようにすることでこれを可能にします。

EKS でこれらの機能を有効にするには、GPUs をスケジュール可能なリソースとして公開し、タイムスライスと MIG をサポートする NVIDIA Device Plugin をデプロイできます。詳細については、「Kubernetes のタイムスライス GPUs」と「NVIDIA タイムスライスおよび高速 EC2 インスタンスを使用した Amazon EKS での GPU 共有」を参照してください。

例

たとえば、NVIDIA Device Plugin でタイムスライスを有効にするには:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: nvidia-device-plugin-config
  namespace: kube-system
data:
  config.yaml: |
    version: v1
    sharing:
      timeSlicing:
        resources:
        - name: nvidia.com/gpu
          replicas: 4  # Allow 4 pods to share each GPU
```

例

たとえば、KAI スケジューラを小数 GPU 割り当てに使用するには、NVIDIA GPU Operator と一緒にデプロイし、ポッド仕様で小数 GPU リソースを指定します。

```
apiVersion: v1
kind: Pod
metadata:
  name: fractional-gpu-pod-example
  annotations:
    gpu-fraction: "0.5"  # Annotation for 50% GPU
  labels:
    runai/queue: "default"  # Required queue assignment
spec:
  containers:
  - name: ml-workload
    image: nvcr.io/nvidia/pytorch:25.04-py3
    resources:
      limits:
        nvidia.com/gpu: 1
  nodeSelector:
    nvidia.com/gpu: "true"
  schedulerName: kai-scheduler
```

ノードの耐障害性とトレーニングジョブの管理

自動復旧によるノードヘルスチェックの実装

複数のノードにまたがるマルチ GPU モデルトレーニングなど、ノード間の頻繁な通信を必要とする Amazon EKS の分散トレーニングジョブでは、GPU や EFA の障害などのハードウェアの問題により、トレーニングジョブが中断される可能性があります。これらの中断により、特に安定したハードウェアに依存する長時間実行される AI/ML ワークロードでは、トレーニングの進行状況が失われ、コストが増加する可能性があります。

GPU ワークロードを実行している EKS クラスターの GPU 障害など、ハードウェア障害に対する回復力を高めるために、EKS Node Monitoring Agent with Auto Repair または Amazon SageMaker HyperPod のいずれかを利用することをお勧めします。自動修復を備えた EKS Node Monitoring Agent は、ノードのヘルスモニタリングや標準の Kubernetes メカニズムを使用した自動修復などの機能を提供しますが、SageMaker HyperPod は、ディープヘルスチェックや自動ジョブ再開など、大規模な ML トレーニング用に特別に設計された、ターゲットを絞った回復力と追加機能を提供します。

- Node Auto Repair を備えた EKS Node Monitoring Agent は、ログを読み取って NodeConditions を適用することでノードの状態を継続的にモニタリングします。これには、高速ハードウェアに固有の標準条件Readyや、GPU やネットワーク障害などの問題を特定するための条件が含まれます。ノードが異常であると判断された場合、ノードの自動修復はノードを遮断し、新しいノードに置き換えます。ポッドの再スケジュールとジョブの再起動は、標準の Kubernetes メカニズムとジョブの再起動ポリシーに依存します。
- SageMaker HyperPod のディープヘルスチェックとヘルスモニタリングエージェントは、GPU および Trainium ベースのインスタンスのヘルスステータスを継続的にモニタリングします。これは AI/ML ワークロードに合わせて調整され、ラベル (node-health-status など) を使用してノードの状態を管理します。ノードが異常と見なされると、HyperPod は GPUs。デフォルトでは、基本的なヘルスチェックを通じて EFA のネットワーク関連の障害を検出し、中断されたトレーニングジョブの自動再開をサポートしているため、最後のチェックポイントからジョブを続行できるため、大規模な ML タスクの中断を最小限に抑えることができます。

EFA を使用して Auto Repair を使用する EKS Node Monitoring Agent と SageMaker HyperPod クラスターの両方について、リモートダイレクトメモリアクセス (RDMA) エラーやパケットドロップなどの EFA 固有のメトリクスをモニタリングするには、AWS EFA ドライバーがインストールされていることを確認してください。さらに、CloudWatch Observability アドオンをデプロイするか、Prometheus と Grafana を備えた DCGM Exporter などのツールを使用して EFA、GPU、SageMaker HyperPod の機能に関連する特定のメトリクスをモニタリングすることをお勧めします。

中断の影響を受けやすいワークロードの Karpenter 統合を無効にする

処理、大規模な AI/ML 予測タスク、トレーニングなどの中断の影響を受けやすいワークロードの場合は、ジョブ実行中の中断を防ぐために Karpenter 統合ポリシーを調整することをお勧めします。Karpenter の統合機能は、使用率の低いノードを終了するか、低価格の代替ノードに置き換えることで、クラスターコストを自動的に最適化します。ただし、ワークロードが GPU をフル活用している場合でも、Karpenter はポッドの要件を満たす低価格の適正サイズのインスタンスタイプを識別し、ジョブの中断につながる場合、ノードを統合することがあります。

`WhenEmptyOrUnderutilized` 統合ポリシーはノードを早期に終了し、実行時間が長くなる可能性があります。たとえば、ポッドの再スケジュール、データの再ロードが原因でジョブの再開が中断されると、長時間実行されるバッチ推論ジョブにはコストがかかる可能性があります。これを軽減するには、 `consolidationPolicy`を に設定`WhenEmpty`し、ワークロードの急増時にノードを保持するように 1 時間などの`consolidateAfter`期間を設定できます。例:

```
disruption:
  consolidationPolicy: WhenEmpty
  consolidateAfter: 60m
```

このアプローチにより、リアルタイムのオンライン推論データ処理やモデルトレーニングなど、急増するバッチ推論ワークロードやその他の中断の影響を受けやすいジョブのポッド起動レイテンシーが改善され、中断のコストがコンピューティングコスト削減を上回るようになります。Karpenter NodePool Disruption Budgets は、Karpenter の中断を管理するためのもう 1 つの機能です。予算を使用すると、特定の時点で選択した NodePool で中断されるノード数が一定以下になるようにできます。また、中断予算を使用して、特定の時間 (ピーク時間など) にすべてのノードが中断されるのを防ぐこともできます。詳細については、「Karpenter 統合ドキュメント」を参照してください。

ttlSecondsAfterFinished を使用して Kubernetes ジョブを自動クリーンアップする

Amazon EKS の Kubernetes ジョブ`ttlSecondsAfterFinished`に対して を設定して、完了したジョブオブジェクトを自動的に削除することをお勧めします。リンガージョブオブジェクトは、API サーバーメモリなどのクラスターリソースを消費し、ダッシュボード (Grafana、Amazon CloudWatch など) を整理してモニタリングを複雑にします。例えば、TTL を 1 時間に設定すると、ジョブは完了直後に削除され、クラスターが整理されます。詳細については、「終了したジョブの自動クリーンアップ」を参照してください。

優先度の高いジョブ/ワークロード用に優先度の低いジョブのプリエンプションを設定する

Amazon EKS の混合優先度 AI/ML ワークロードでは、優先度の高いタスク (リアルタイム推論など) がリソースを迅速に受信するように、優先度の低いジョブのプリエンプションを設定できます。プリエンプションがない場合、バッチプロセス (バッチ推論、データ処理など) 、非バッチサービス (バックグラウンドタスク、cron ジョブなど) 、CPU/メモリ集約型ジョブ (ウェブサービスなど) などの優先度の低いワークロードは、ノードを占有することで重要なポッドを遅らせる可能性があります。プリエンプションにより、優先度の高いポッドにリソースが必要な場合に Kubernetes は優先度の低いポッドを削除できるため、GPUs、CPUs、またはメモリを備えたノードで効率的なリソース割り当てが可能になります。Kubernetes を使用して優先順位`PriorityClass`を割り当て`PodDisruptionBudget`、エビクション動作を制御することをお勧めします。

```
apiVersion: scheduling.k8s.io/v1
kind: PriorityClass
metadata:
  name: low-priority
value: 100
```

```
---
spec:
  priorityClassName: low-priority
```

詳細については、「Kubernetes Priority and Preemption Documentation」を参照してください。

アプリケーションのスケーリングとパフォーマンス

Karpenter または静的ノードを使用して ML ワークロードのコンピューティング容量を調整する

Amazon EKS での機械学習 (ML) ワークフローのコスト効率と応答性を確保するため、ワークロードの特性とコストコミットメントに合わせてノードプロビジョニング戦略を調整することをお勧めします。以下は、Karpenter just-in-timeスケーリングと、リザーブドキャパシティの静的ノードグループという 2 つの考慮すべきアプローチです。

- Karpenter のようなJust-in-timeデータプレーンスケーラー: コンピューティング需要が変化する動的 ML ワークフロー (GPU ベースの推論と CPU ベースのプロットなど) では、Karpenter のような just-in-timeデータプレーンスケーラーを使用することをお勧めします。
- 予測可能なワークロードに静的ノードグループを使用する: 予測可能な定常状態の ML ワークロードの場合、またはリザーブドインスタンスを使用する場合、EKS マネージド型ノードグループは、リザーブドキャパシティが完全にプロビジョニングおよび利用され、節約を最大化するのに役立ちます。このアプローチは、RIs または ODCRs を介してコミットされる特定のインスタンスタイプに最適です。

例

これは、インスタンス生成が 3 より大きい g Amazon EC2 インスタンスの起動を可能にする多様な Karpenter NodePool の例です。

```
apiVersion: karpenter.sh/v1
kind: NodePool
metadata:
  name: gpu-inference
spec:
  template:
    spec:
      nodeClassRef:
        group: karpenter.k8s.aws
        kind: EC2NodeClass
```

```
        name: default
      requirements:
        - key: karpenter.sh/capacity-type
          operator: In
          values: ["on-demand"]
        - key: karpenter.k8s.aws/instance-category
          operator: In
          values: ["g"]
        - key: karpenter.k8s.aws/instance-generation
          operator: Gt
          values: ["3"]
        - key: kubernetes.io/arch
          operator: In
          values: ["amd64"]
      taints:
        - key: nvidia.com/gpu
          effect: NoSchedule
  limits:
    cpu: "1000"
    memory: "4000Gi"
    nvidia.com/gpu: "10"  *# Limit the total number of GPUs to 10 for the NodePool*
  disruption:
    consolidationPolicy: WhenEmpty
    consolidateAfter: 60m
    expireAfter: 720h
```

例

トレーニングワークロードに静的ノードグループを使用する例:

```
apiVersion: eksctl.io/v1alpha5
kind: ClusterConfig
metadata:
  name: ml-cluster
  region: us-west-2
managedNodeGroups:
  - name: gpu-node-group
    instanceType: p4d.24xlarge
    minSize: 2
    maxSize: 2
    desiredCapacity: 2
    taints:
      - key: nvidia.com/gpu
```

```
        effect: NoSchedule
```

テイントと許容範囲を使用して、高速化されていないワークロードが高速インスタンスでスケジュールされないようにします。

GPU リソースで高速化されていないワークロードをスケジュールすることはコンピューティング効率が高くありません。高速化されていないワークロードポッドが不適切なノードでスケジュールされないように、テイントと許容値を使用することをお勧めします。詳細については、Kubernetes のドキュメントを参照してください。

モデルのパフォーマンスに基づいてスケールする

推論ワークロードの場合、Kubernetes Event-Driven Auto Scaling (KEDA) を使用して、推論リクエストやトークンスループットなどのモデルパフォーマンスメトリクスに基づいて適切なクールダウン期間でスケーリングすることをお勧めします。静的スケーリングポリシーでは、リソースのプロビジョニングが過剰または過小になり、コストとレイテンシーに影響する可能性があります。詳細については、KEDA ドキュメントを参照してください。

高度な GPU 管理のための動的リソース割り当て

動的リソース割り当て (DRA) は、Kubernetes GPU リソース管理の根本的な進歩を表します。DRA は従来のデバイスプラグインの制限を超えて、高度な GPU 共有、トポロジ対応、クロスノードリソース調整を可能にします。Amazon EKS バージョン 1.33 で利用可能な DRA は、以下を提供することで AI/ML ワークロードの重要な課題に対処します。

- きめ細かな GPU 割り当て
- マルチプロセスサービス (MPS) やマルチインスタンス GPU (MIG) などの高度な共有メカニズム
- NVIDIA GB200 UltraServers を含む次世代ハードウェアアーキテクチャのサポート

従来の GPU 割り当てでは、GPUs不透明な整数リソースとして扱われ、使用率が大幅に低下します (多くの場合、本番稼働用クラスターでは 30～40%)。これは、ワークロードが小数リソースのみを必要とする場合でもGPUs 全体への排他的アクセスを受け取るために発生します。DRA は、Kubernetes スケジューラにハードウェア特性とワークロード要件を完全に可視化する構造化された宣言型割り当てを導入することで、このモデルを変換します。これにより、インテリジェントな配置の決定と効率的なリソース共有が可能になります。

NVIDIA デバイスプラグインの代わりに DRA を使用する利点

NVIDIA デバイスプラグイン (バージョン 以降`0.12.0`) は、タイムスライス、MPS、MIG などの GPU 共有メカニズムをサポートしています。ただし、DRA が対処するアーキテクチャ上の制限があります。

NVIDIA デバイスプラグインの制限事項

- 静的設定: GPU 共有設定 (タイムスライスレプリカと MPS 設定) には、 を通じてクラスター全体の事前設定が必要です`ConfigMaps`。これにより、ワークロードごとに異なる共有戦略を提供することが困難になります。
- きめ細かな選択の制限: デバイスプラグインはノードラベルを介して GPU 特性を公開しますが、ワークロードはスケジューリングの決定の一環として特定の GPU 設定 (メモリサイズとコンピューティング機能) を動的にリクエストすることはできません。
- クロスノードリソースの調整なし: 複数のノードにまたがる分散 GPU リソースを管理したり、NVIDIA GB200 などのシステムの NVLink ドメインなどの複雑なトポロジ要件を表現したりすることはできません。
- スケジューラの制約: Kubernetes スケジューラは GPU リソースを不透明な整数として扱い、トポロジ対応の決定や複雑なリソース依存関係の処理能力を制限します。
- 設定の複雑さ: さまざまな共有戦略をセットアップするには`ConfigMaps`、複数の慎重なノードラベル付けが必要で、運用が複雑になります。

DRA を使用したソリューション

- 動的リソース選択: DRA を使用すると、ワークロードは を通じてリクエスト時に詳細な要件 (GPU メモリ、ドライバーバージョン、および特定の属性) を指定できます`resourceclaims`。これにより、より柔軟なリソースマッチングが可能になります。
- トポロジの認識: DRA は、構造化されたパラメータとデバイスセレクタを通じて、クロスノード GPU 通信やメモリコヒーレント相互接続などの複雑な要件を処理します。
- クロスノードリソース管理: 複数のノードにまたがる分散 GPU リソースの調整`computeDomains`を可能にします。これは、IMEX チャネルを使用する GB200 などのシステムにとって重要です。
- ワークロード固有の設定: それぞれが異なる共有戦略と設定`ResourceClaim`を指定し、クラスター全体の設定ではなくワークロードごとにきめ細かな制御を可能にします。
- スケジューラの統合の強化: DRA はスケジューラに詳細なデバイス情報を提供し、ハードウェアトポロジとリソースの特性に基づいて、よりインテリジェントな配置の決定を可能にします。

重要: DRA は NVIDIA デバイスプラグインを完全に置き換えるものではありません。NVIDIA DRA ドライバーは、デバイスプラグインと連携して機能を強化します。デバイスプラグインは引き続き基本的な GPU の検出と管理を処理し、DRA は高度な割り当てとスケジューリング機能を追加します。

DRA でサポートされているインスタンスとその機能

DRA のサポートは、次の表に示すように、Amazon EC2 インスタンスファミリーと GPU アーキテクチャによって異なります。

インスタンスファミリー	GPU タイプ	タイムスライス	MIG サポート	MPS サポート	IMEX サポート	ユースケース
G5	NVIDIA A10G	あり	なし	あり	いいえ	推論とグラフィックスのワークロード
G6	NVIDIA L4	あり	なし	あり	いいえ	AI 推論とビデオ処理
G6e	NVIDIA L40S	あり	なし	あり	いいえ	トレーニング、推論、グラフィックス
P4d/P4de	NVIDIA A100	はい	あり	あり	いいえ	大規模なトレーニングと HPC
P5	NVIDIA H100	はい	あり	あり	いいえ	基盤モデルトレーニング
P6	NVIDIA B200	はい	あり	あり	いいえ	数十億または数兆のパラメータモデル、分散トレーニング、推論
P6e	NVIDIA GB200	はい	あり	あり	あり	数十億または数兆のパラメータモデル、分散トレーニング、推論

以下は、 テーブルの各機能の説明です。

- タイムスライス: 複数のワークロードが GPU コンピューティングリソースを経時的に共有できるようにします。
- マルチインスタンス GPU (MIG): 分離された GPU インスタンスを作成するハードウェアレベルのパーティショニング。
- マルチプロセスサービス (MPS): 1 つの GPU で複数の CUDA プロセスの同時実行を有効にします。
- Internode Memory Exchange (IMEX): GB200 UltraServers のノード間のメモリコヒーレント通信。

その他のリソース

Kubernetes DRA および NVIDIA DRA ドライバーの詳細については、GitHub の以下のリソースを参照してください。

- Kubernetes dynamic-resource-allocation
- DRA の Kubernetes 拡張提案
- GPUs 用 NVIDIA DRA ドライバー
- NVIDIA DRA の例とクイックスタート

高度な GPU 管理のための動的リソース割り当てを設定する

次のトピックでは、高度な GPU 管理のために動的リソース割り当て (DRA) を設定する方法を示します。

前提条件

Amazon EKS に DRA を実装する前に、環境が次の要件を満たしていることを確認してください。

クラスターの設定

- バージョン `1.33` 以降を実行している Amazon EKS クラスター
- Amazon EKS マネージド型ノードグループ (DRA は現在、Karpenter ではなく、AL2023 および Bottlerocket NVIDIA 最適化 AMIs を持つマネージド型ノードグループでのみサポートされています) https://github.com/kubernetes-sigs/karpenter/issues/1231
- 適切なインスタンスタイプを持つ NVIDIA GPU 対応ワーカーノード

必要なコンポーネント

- NVIDIA デバイスプラグインバージョン `0.17.1` 以降
- NVIDIA DRA ドライバーバージョン `25.3.0`以降

ステップ 1: eksctl を使用して DRA 対応ノードグループでクラスターを作成する

1. という名前のクラスター設定ファイルを作成しますdra-eks-cluster.yaml。

```
---
apiVersion: eksctl.io/v1alpha5
kind: ClusterConfig

metadata:
  name: dra-eks-cluster
  region: us-west-2
  version: '1.33'

managedNodeGroups:
- name: gpu-dra-nodes
  amiFamily: AmazonLinux2023
  instanceType: g6.12xlarge
  desiredCapacity: 2
  minSize: 1
  maxSize: 3

  labels:
    node-type: "gpu-dra"
    nvidia.com/gpu.present: "true"

  taints:
  - key: nvidia.com/gpu
    value: "true"
    effect: NoSchedule
```

2. クラスターを作成します。

```
eksctl create cluster -f dra-eks-cluster.yaml
```

ステップ 2: NVIDIA デバイスプラグインをデプロイする

NVIDIA デバイスプラグインをデプロイして、基本的な GPU 検出を有効にします。

1. NVIDIA デバイスプラグイン Helm リポジトリを追加します。

```
helm repo add nvidia https://nvidia.github.io/k8s-device-plugin
helm repo update
```

2. デバイスプラグインのカスタム値を作成します。

```
cat <<EOF > nvidia-device-plugin-values.yaml
gfd:
  enabled: true
nfd:
  enabled: true
tolerations:
  - key: nvidia.com/gpu
    operator: Exists
    effect: NoSchedule
EOF
```

3. NVIDIA デバイスプラグインをインストールします。

```
helm install nvidia-device-plugin nvidia/nvidia-device-plugin \
 --namespace nvidia-device-plugin \
 --create-namespace \
 --version v0.17.1 \
 --values nvidia-device-plugin-values.yaml
```

ステップ 3: NVIDIA DRA ドライバー Helm チャートをデプロイする

1. DRA ドライバー`dra-driver-values.yaml`の値ファイルを作成します。

```
---
nvidiaDriverRoot: /

gpuResourcesEnabledOverride: true

resources:
  gpus:
    enabled: true
```

```
  computeDomains:
    enabled: true  # Enable for GB200 IMEX support

controller:
  tolerations:
  - key: nvidia.com/gpu
    operator: Exists
    effect: NoSchedule

kubeletPlugin:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
        - matchExpressions:
          - key: "nvidia.com/gpu.present"
            operator: In
            values: ["true"]
  tolerations:
  - key: nvidia.com/gpu
    operator: Exists
    effect: NoSchedule
```

2. NVIDIA NGC Helm リポジトリを追加します。

```
helm repo add nvidia https://helm.ngc.nvidia.com/nvidia
helm repo update
```

3. NVIDIA DRA ドライバーをインストールします。

```
helm install nvidia-dra-driver nvidia/nvidia-dra-driver-gpu \
 --version="25.3.0-rc.2" \
 --namespace nvidia-dra-driver \
 --create-namespace \
 --values dra-driver-values.yaml
```

ステップ 4: DRA のインストールを確認する

1. DRA API リソースが使用可能であることを確認します。

```
kubectl api-resources | grep resource.k8s.io/v1beta1
```

予想される出力は次のとおりです。

```
deviceclasses resource.k8s.io/v1beta1 false DeviceClass
resourceclaims resource.k8s.io/v1beta1 true ResourceClaim
resourceclaimtemplates resource.k8s.io/v1beta1 true ResourceClaimTemplate
resourceslices resource.k8s.io/v1beta1 false ResourceSlice
```

2. 使用可能なデバイスクラスを確認します。

```
kubectl get deviceclasses
```

以下は、予想される出力の例です。

```
NAME                                        AGE
compute-domain-daemon.nvidia.com            4h39m
compute-domain-default-channel.nvidia.com   4h39m
gpu.nvidia.com                              4h39m
mig.nvidia.com                              4h39m
```

新しく作成された G6 GPU インスタンスが DRA を有効にして Amazon EKS クラスターに参加すると、次のアクションが発生します。

- NVIDIA DRA ドライバーは A10G GPU を自動的に検出し、そのノードresourceslicesに 2 つを作成します。
- gpu.nvidia.com スライスは、物理 A10G GPU デバイスをその仕様 (メモリ、コンピューティング機能など) に登録します。
- A10G は MIG パーティショニングをサポートしていないため、compute-domain.nvidia.comスライスは GPU のコンピューティングコンテキスト全体を表す単一のコンピューティングドメインを作成します。
- 次にresourceslices、これらは Kubernetes API サーバーに発行され、GPU リソースが を通じてスケジューリングできるようになりますresourceclaims。

  DRA スケジューラは、 を介して GPU リソースをリクエストする Pod にこの GPU をインテリジェントに割り当てることができるようになりresourceclaimtemplates、従来のデバイスプラグインアプローチよりも柔軟なリソース管理が可能になりました。これは、手動操作なしで自動的に行われます。DRA ドライバーがリソースの検出と登録プロセスを完了すると、ノードは GPU ワークロードで使用できるようになります。

次のコマンドを実行する場合:

```
kubectl get resourceslices
```

以下は、予想される出力の例です。

```
NAME                                                                NODE
            DRIVER                          POOL                                AGE
ip-100-64-129-47.ec2.internal-compute-domain.nvidia.com-rwsts
 ip-100-64-129-47.ec2.internal    compute-domain.nvidia.com
 ip-100-64-129-47.ec2.internal    35m
ip-100-64-129-47.ec2.internal-gpu.nvidia.com-6kndg
 ip-100-64-129-47.ec2.internal    gpu.nvidia.com
 ip-100-64-129-47.ec2.internal    35m
```

「the section called “動的リソース割り当てを使用してシンプルな GPU ワークロードをスケジュールする”」に進みます。

動的リソース割り当てを使用してシンプルな GPU ワークロードをスケジュールする

動的リソース割り当て (DRA) を使用してシンプルな GPU ワークロードをスケジュールするには、次の手順を実行します。先に進む前に、 に従っていることを確認してくださいthe section called “高度な GPU 管理のための動的リソース割り当てを設定する”。

1. という名前のファイルを使用して GPU 割り当てResourceClaimTemplateの基本 を作成しますbasic-gpu-claim-template.yaml。

```
---
apiVersion: v1
kind: Namespace
metadata:
  name: gpu-test1

---
apiVersion: resource.k8s.io/v1beta1
kind: ResourceClaimTemplate
metadata:
  namespace: gpu-test1
  name: single-gpu
spec:
```

```
  spec:
    devices:
      requests:
      - name: gpu
        deviceClassName: gpu.nvidia.com
```

2. テンプレートを適用します。

```
kubectl apply -f basic-gpu-claim-template.yaml
```

3. ステータスを確認します。

```
kubectl get resourceclaimtemplates -n gpu-test1
```

出力例を次に示します。

```
NAME         AGE
single-gpu   9m16s
```

4. という名前のファイルResourceClaimTemplateで を使用する Pod を作成しますbasic-gpu-pod.yaml。

```
---
apiVersion: v1
kind: Pod
metadata:
  namespace: gpu-test1
  name: gpu-pod
  labels:
    app: pod
spec:
  containers:
  - name: ctr0
    image: ubuntu:22.04
    command: ["bash", "-c"]
    args: ["nvidia-smi -L; trap 'exit 0' TERM; sleep 9999 & wait"]
    resources:
      claims:
      - name: gpu0
  resourceClaims:
  - name: gpu0
    resourceClaimTemplateName: single-gpu
```

```
  nodeSelector:
    NodeGroupType: gpu-dra
    nvidia.com/gpu.present: "true"
  tolerations:
  - key: "nvidia.com/gpu"
    operator: "Exists"
    effect: "NoSchedule"
```

5. Pod を適用してモニタリングします。

```
kubectl apply -f basic-gpu-pod.yaml
```

6. Pod のステータスを確認します。

```
kubectl get pod -n gpu-test1
```

予想される出力の例を次に示します。

```
NAME      READY   STATUS    RESTARTS   AGE
gpu-pod   1/1     Running   0          13m
```

7. ResourceClaim ステータスを確認します。

```
kubectl get resourceclaims -n gpu-test1
```

予想される出力の例を次に示します。

```
NAME                 STATE                AGE
gpu-pod-gpu0-l76cg   allocated,reserved   9m6s
```

8. Pod ログを表示して GPU 情報を確認します。

```
kubectl logs gpu-pod -n gpu-test1
```

予想される出力の例を次に示します。

```
GPU 0: NVIDIA L4 (UUID: GPU-da7c24d7-c7e3-ed3b-418c-bcecc32af7c5)
```

DRA を使用したより高度な GPU 最適化手法the section called “動的リソース割り当てによる GPU 最適化手法”については、「」を参照してください。

動的リソース割り当てによる GPU 最適化手法

最新の GPU ワークロードでは、最適な使用率とコスト効率を実現するために、高度なリソース管理が必要です。DRA は、さまざまなユースケースとハードウェア機能に対応するいくつかの高度な最適化手法を可能にします。

- タイムスライスを使用すると、複数のワークロードで GPU コンピューティングリソースを経時的に共有できるため、GPU の使用量が散発的な推論ワークロードに最適です。例については、the section called “タイムスライスによる GPU ワークロードの最適化”を参照してください。
- マルチプロセスサービス (MPS) を使用すると、タイムスライスよりも優れた分離で、単一の GPU で複数の CUDA プロセスを同時に実行できます。例については、the section called “MPS で GPU ワークロードを最適化する”を参照してください。
- マルチインスタンス GPU (MIG) はハードウェアレベルのパーティショニングを提供し、専用のコンピューティングリソースとメモリリソースを備えた独立した GPU インスタンスを作成します。例については、the section called “マルチインスタンス GPU で GPU ワークロードを最適化する”を参照してください。
- Internode Memory Exchange (IMEX) を使用すると、ノード間のメモリコヒーレント通信が可能になり、NVIDIA GB200 システムでの分散トレーニングが可能になります。例については、the section called “GB200 P6e インスタンスを使用して IMEX で GPU ワークロードを最適化する”を参照してください。

これらの手法により、リソースの使用率が大幅に向上します。組織は、GPU 使用率が従来の割り当てで 30～40% から、最適化された共有戦略で 80～90% に増加すると報告しています。手法の選択は、ワークロードの特性、分離要件、ハードウェア機能によって異なります。

タイムスライスによる GPU ワークロードの最適化

タイムスライスを使用すると、複数のワークロードが同じ物理 GPU で順次実行されるようにスケジュールすることで、GPU コンピューティングリソースを共有できます。GPU を散発的に使用する推論ワークロードに最適です。

次の手順を実行します。

1. という名前のファイルを使用してタイムスライスResourceClaimTemplate用の を定義しますtimeslicing-claim-template.yaml。

```
---
apiVersion: v1
kind: Namespace
metadata:
  name: timeslicing-gpu

---
apiVersion: resource.k8s.io/v1beta1
kind: ResourceClaimTemplate
metadata:
  name: timeslicing-gpu-template
  namespace: timeslicing-gpu
spec:
  spec:
    devices:
      requests:
      - name: shared-gpu
        deviceClassName: gpu.nvidia.com
      config:
      - requests: ["shared-gpu"]
        opaque:
          driver: gpu.nvidia.com
          parameters:
            apiVersion: resource.nvidia.com/v1beta1
            kind: GpuConfig
            sharing:
              strategy: TimeSlicing
```

2. という名前のファイルでタイムスライスを使用してポッドを定義しますtimeslicing-pod.yaml。

```
---
# Pod 1 - Inference workload
apiVersion: v1
kind: Pod
metadata:
  name: inference-pod-1
  namespace: timeslicing-gpu
  labels:
    app: gpu-inference
spec:
  restartPolicy: Never
```

```
  containers:
  - name: inference-container
    image: nvcr.io/nvidia/pytorch:25.04-py3
    command: ["python", "-c"]
    args:
    - |
      import torch
      import time
      import os
      print(f"=== POD 1 STARTING ===")
      print(f"GPU available: {torch.cuda.is_available()}")
      print(f"GPU count: {torch.cuda.device_count()}")
      if torch.cuda.is_available():
          device = torch.cuda.current_device()
          print(f"Current GPU: {torch.cuda.get_device_name(device)}")
          print(f"GPU Memory:
 {torch.cuda.get_device_properties(device).total_memory / 1024**3:.1f} GB")
          # Simulate inference workload
          for i in range(20):
              x = torch.randn(1000, 1000).cuda()
              y = torch.mm(x, x.t())
              print(f"Pod 1 - Iteration {i+1} completed at {time.strftime('%H:%M:
%S')}")
              time.sleep(60)
      else:
          print("No GPU available!")
          time.sleep(5)
    resources:
      claims:
      - name: shared-gpu-claim
  resourceClaims:
  - name: shared-gpu-claim
    resourceClaimTemplateName: timeslicing-gpu-template
  nodeSelector:
    NodeGroupType: "gpu-dra"
    nvidia.com/gpu.present: "true"
  tolerations:
  - key: nvidia.com/gpu
    operator: Exists
    effect: NoSchedule


---
# Pod 2 - Training workload
```

```
apiVersion: v1
kind: Pod
metadata:
  name: training-pod-2
  namespace: timeslicing-gpu
  labels:
    app: gpu-training
spec:
  restartPolicy: Never
  containers:
  - name: training-container
    image: nvcr.io/nvidia/pytorch:25.04-py3
    command: ["python", "-c"]
    args:
    - |
      import torch
      import time
      import os
      print(f"=== POD 2 STARTING ===")
      print(f"GPU available: {torch.cuda.is_available()}")
      print(f"GPU count: {torch.cuda.device_count()}")
      if torch.cuda.is_available():
          device = torch.cuda.current_device()
          print(f"Current GPU: {torch.cuda.get_device_name(device)}")
          print(f"GPU Memory:
 {torch.cuda.get_device_properties(device).total_memory / 1024**3:.1f} GB")
          # Simulate training workload with heavier compute
          for i in range(15):
              x = torch.randn(2000, 2000).cuda()
              y = torch.mm(x, x.t())
              loss = torch.sum(y)
              print(f"Pod 2 - Training step {i+1}, Loss: {loss.item():.2f} at
 {time.strftime('%H:%M:%S')}")
              time.sleep(5)
      else:
          print("No GPU available!")
          time.sleep(60)
    resources:
      claims:
      - name: shared-gpu-claim-2
  resourceClaims:
  - name: shared-gpu-claim-2
    resourceClaimTemplateName: timeslicing-gpu-template
  nodeSelector:
```

```
    NodeGroupType: "gpu-dra"
    nvidia.com/gpu.present: "true"
  tolerations:
  - key: nvidia.com/gpu
    operator: Exists
    effect: NoSchedule
```

3. テンプレートと Pod を適用します。

```
kubectl apply -f timeslicing-claim-template.yaml
kubectl apply -f timeslicing-pod.yaml
```

4. リソースクレームをモニタリングします。

```
kubectl get resourceclaims -n timeslicing-gpu -w
```

出力例を次に示します。

```
NAME                                        STATE                AGE
inference-pod-1-shared-gpu-claim-9p97x      allocated,reserved   21s
training-pod-2-shared-gpu-claim-2-qghnb     pending              21s
inference-pod-1-shared-gpu-claim-9p97x      pending              105s
training-pod-2-shared-gpu-claim-2-qghnb     pending              105s
inference-pod-1-shared-gpu-claim-9p97x      pending              105s
training-pod-2-shared-gpu-claim-2-qghnb     allocated,reserved   105s
inference-pod-1-shared-gpu-claim-9p97x      pending              105s
```

最初のポッド (`inference-pod-1`)

- 状態: `allocated,reserved`
- 意味: DRA が使用可能な GPU を見つけ、この Pod 用に予約しました
- ポッドステータス: すぐに実行を開始します

2 番目のポッド (`training-pod-2`)

- 状態: `pending`
- 意味: DRA が同じ GPU でタイムスライスを設定するのを待つ
- ポッドのステータス: スケジュールの待機中

- 状態は から pending allocated,reservedに変わります running

MPS で GPU ワークロードを最適化する

マルチプロセスサービス (MPS) を使用すると、タイムスライスよりも優れた分離で、単一の GPU で複数の CUDA コンテキストを同時に実行できます。

次の手順を実行します。

1. という名前のファイルを使用して MPS ResourceClaimTemplateの を定義しますmps-claim-template.yaml。

```
---
apiVersion: v1
kind: Namespace
metadata:
  name: mps-gpu

---
apiVersion: resource.k8s.io/v1beta1
kind: ResourceClaimTemplate
metadata:
  name: mps-gpu-template
  namespace: mps-gpu
spec:
  spec:
    devices:
      requests:
      - name: shared-gpu
        deviceClassName: gpu.nvidia.com
      config:
      - requests: ["shared-gpu"]
        opaque:
          driver: gpu.nvidia.com
          parameters:
            apiVersion: resource.nvidia.com/v1beta1
            kind: GpuConfig
            sharing:
              strategy: MPS
```

2. という名前のファイルで MPS を使用してポッドを定義しますmps-pod.yaml。

```
---
# Single Pod with Multiple Containers sharing GPU via MPS
apiVersion: v1
kind: Pod
metadata:
  name: mps-multi-container-pod
  namespace: mps-gpu
  labels:
    app: mps-demo
spec:
  restartPolicy: Never
  containers:
  # Container 1 - Inference workload
  - name: inference-container
    image: nvcr.io/nvidia/pytorch:25.04-py3
    command: ["python", "-c"]
    args:
    - |
      import torch
      import torch.nn as nn
      import time
      import os

      print(f"=== INFERENCE CONTAINER STARTING ===")
      print(f"Process ID: {os.getpid()}")
      print(f"GPU available: {torch.cuda.is_available()}")
      print(f"GPU count: {torch.cuda.device_count()}")

      if torch.cuda.is_available():
          device = torch.cuda.current_device()
          print(f"Current GPU: {torch.cuda.get_device_name(device)}")
          print(f"GPU Memory:
 {torch.cuda.get_device_properties(device).total_memory / 1024**3:.1f} GB")

          # Create inference model
          model = nn.Sequential(
              nn.Linear(1000, 500),
              nn.ReLU(),
              nn.Linear(500, 100)
          ).cuda()

          # Run inference
          for i in range(1, 999999):
```

```
            with torch.no_grad():
                x = torch.randn(128, 1000).cuda()
                output = model(x)
                result = torch.sum(output)
                print(f"Inference Container PID {os.getpid()}: Batch {i}, Result:
{result.item():.2f} at {time.strftime('%H:%M:%S')}")
            time.sleep(2)
    else:
        print("No GPU available!")
        time.sleep(60)
  resources:
    claims:
    - name: shared-gpu-claim
      request: shared-gpu

 # Container 2 - Training workload
 - name: training-container
   image: nvcr.io/nvidia/pytorch:25.04-py3
   command: ["python", "-c"]
   args:
   - |
     import torch
     import torch.nn as nn
     import time
     import os

     print(f"=== TRAINING CONTAINER STARTING ===")
     print(f"Process ID: {os.getpid()}")
     print(f"GPU available: {torch.cuda.is_available()}")
     print(f"GPU count: {torch.cuda.device_count()}")

     if torch.cuda.is_available():
         device = torch.cuda.current_device()
         print(f"Current GPU: {torch.cuda.get_device_name(device)}")
         print(f"GPU Memory:
{torch.cuda.get_device_properties(device).total_memory / 1024**3:.1f} GB")

         # Create training model
         model = nn.Sequential(
             nn.Linear(2000, 1000),
             nn.ReLU(),
             nn.Linear(1000, 500),
             nn.ReLU(),
             nn.Linear(500, 10)
```

```
            ).cuda()

            criterion = nn.MSELoss()
            optimizer = torch.optim.Adam(model.parameters(), lr=0.001)

            # Run training
            for epoch in range(1, 999999):
                x = torch.randn(64, 2000).cuda()
                target = torch.randn(64, 10).cuda()

                optimizer.zero_grad()
                output = model(x)
                loss = criterion(output, target)
                loss.backward()
                optimizer.step()

                print(f"Training Container PID {os.getpid()}: Epoch {epoch}, Loss:
{loss.item():.4f} at {time.strftime('%H:%M:%S')}")
                time.sleep(3)
        else:
            print("No GPU available!")
            time.sleep(60)
      resources:
        claims:
        - name: shared-gpu-claim
          request: shared-gpu

    resourceClaims:
    - name: shared-gpu-claim
      resourceClaimTemplateName: mps-gpu-template

    nodeSelector:
      NodeGroupType: "gpu-dra"
      nvidia.com/gpu.present: "true"
    tolerations:
    - key: nvidia.com/gpu
      operator: Exists
      effect: NoSchedule
```

3. テンプレートを適用し、複数の MPS Pod を作成します。

```
kubectl apply -f mps-claim-template.yaml
kubectl apply -f mps-pod.yaml
```

4. リソースクレームをモニタリングします。

```
kubectl get resourceclaims -n mps-gpu -w
```

出力例を次に示します。

```
NAME                                                STATE                AGE
mps-multi-container-pod-shared-gpu-claim-2p9kx   allocated,reserved   86s
```

この設定は、動的リソース割り当て (DRA) による NVIDIA Multi-Process Service (MPS) を使用した真の GPU 共有を示しています。GPU を使用してワークロードが順番に変化するタイムスライスとは異なり、MPS では両方のコンテナを同じ物理 GPU で同時に実行できます。重要なインサイトは、DRA MPS 共有には、複数の個別の Pod ではなく、単一の Pod 内に複数のコンテナが必要であることです。デプロイされると、DRA ドライバーは Pod `ResourceClaim`に 1 つを割り当て、推論コンテナとトレーニングコンテナの両方を同時に実行できるように MPS を自動的に設定します。

各コンテナは、独自の独立した GPU メモリスペースとコンピューティングリソースを取得し、MPS デーモンが基盤となるハードウェアへのアクセスを調整します。次の操作を行うことで、これが機能していることを確認できます。

- をチェックすると`nvidia-smi`、両方のコンテナが同じ GPU デバイスを共有する M+C (`MPS + Compute`) プロセスとして表示されます。
- 両方のコンテナからのログをモニタリングすると、同時実行を証明するインターリーブタイムスタンプが表示されます。

このアプローチは、補完的なワークロードが 1 つのプロセスで十分に活用されないままにするのではなく、高価な GPU ハードウェアを効率的に共有できるようにすることで、GPU 使用率を最大化します。

Container1: **`inference-container`**

```
root@mps-multi-container-pod:/workspace# nvidia-smi
Wed Jul 16 21:09:30 2025
+-----------------------------------------------------------------------------------------
+
| NVIDIA-SMI 570.158.01             Driver Version: 570.158.01     CUDA Version: 12.9
   |
```

```
|-----------------------------------------+------------------------
+----------------------+
| GPU  Name                 Persistence-M | Bus-Id          Disp.A | Volatile Uncorr.
 ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |           Memory-Usage | GPU-Util  Compute
 M. |
|                                         |                        |               MIG
 M. |
|=========================================+========================
+======================|
|   0  NVIDIA L4                      On  |   00000000:35:00.0 Off |
 0 |
| N/A   48C    P0             28W /   72W |     597MiB /  23034MiB |      0%   E.
 Process |
|                                         |                        |
 N/A |
+-----------------------------------------+------------------------
+----------------------+

+-----------------------------------------------------------------------------------------
+
| Processes:
   |
|  GPU   GI   CI              PID   Type   Process name                        GPU
 Memory |
|        ID   ID                                                               Usage
   |
|
=========================================================================================|
|    0   N/A  N/A               1    M+C   python
 246MiB |
+-----------------------------------------------------------------------------------------
+
```

Container2: **training-container**

```
root@mps-multi-container-pod:/workspace# nvidia-smi
Wed Jul 16 21:16:00 2025
+-----------------------------------------------------------------------------------------
+
| NVIDIA-SMI 570.158.01             Driver Version: 570.158.01     CUDA Version: 12.9
   |
```

```
|-----------------------------------------+------------------------
+----------------------+
| GPU  Name                 Persistence-M | Bus-Id          Disp.A | Volatile Uncorr.
 ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |           Memory-Usage | GPU-Util  Compute
 M. |
|                                         |                        |               MIG
 M. |
|=========================================+========================
+======================|
|   0  NVIDIA L4                      On  |   00000000:35:00.0 Off |
 0 |
| N/A   51C    P0             28W /   72W |     597MiB /  23034MiB |      0%   E.
 Process |
|                                         |                        |
 N/A |
+-----------------------------------------+------------------------
+----------------------+

+-----------------------------------------------------------------------------------------
+
| Processes:
   |
|  GPU   GI   CI              PID   Type   Process name                        GPU
 Memory |
|        ID   ID                                                               Usage
   |
|
=========================================================================================|
|    0   N/A  N/A               1    M+C   python
 314MiB |
+-----------------------------------------------------------------------------------------
+
```

マルチインスタンス GPU で GPU ワークロードを最適化する

マルチインスタンス GPU (MIG) はハードウェアレベルのパーティショニングを提供し、専用のコンピューティングリソースとメモリリソースを備えた独立した GPU インスタンスを作成します。

さまざまなプロファイルで動的 MIG パーティショニングを使用するには、NVIDIA GPU Operator が必要です。NVIDIA GPU Operator は MIG Manager を使用して MIG プロファイルを作成し、P4D, P4De, P5, P6 などの GPU インスタンスを再起動して設定変更を適用します。GPU Operator には、ノードラベルの変更を監視し、適切な MIG 設定を自動的に適用する MIG Manager コンポーネント

を介した包括的な MIG 管理機能が含まれています。MIG プロファイルの変更がリクエストされると、オペレーターはすべての GPU クライアントを適切にシャットダウンし、新しいパーティションジオメトリを適用して、影響を受けるサービスを再起動します。このプロセスでは、GPU インスタンスのノードを再起動して、GPU の状態をクリーンに移行する必要があります。そのため、MIG Manager 設定`WITH0REBOOT=true`で を有効にすることは、MIG デプロイを成功させるために不可欠です。

Amazon EKS で MIG を使用するには、NVIDIA DRA ドライバーと NVIDIA GPU Operator の両方が必要です。NVIDIA Device Plugin と DCGM Exporter は NVIDIA GPU Operator の一部であるため、これに加えては必要ありません。EKS NVIDIA AMIs には NVIDIA ドライバーがプリインストールされているため、競合を回避し、インスタンスに既に存在する最適化されたドライバーを活用するために、GPU Operator によるドライバーのデプロイを無効にしました。NVIDIA DRA ドライバーは MIG インスタンスの動的リソース割り当てを処理し、GPU オペレーターは GPU ライフサイクル全体を管理します。これには、MIG 設定、デバイスプラグイン機能、DCGM によるモニタリング、ノード機能検出が含まれます。この統合アプローチは、ハードウェアレベルの分離と動的リソース割り当て機能を備えた、エンタープライズ GPU 管理の完全なソリューションを提供します。

ステップ 1: NVIDIA GPU Operator をデプロイする

1. NVIDIA GPU Operator リポジトリを追加します。

```
helm repo add nvidia https://nvidia.github.io/gpu-operator
helm repo update
```

2. `gpu-operator-values.yaml` ファイルを作成します。

```
driver:
  enabled: false

mig:
  strategy: mixed

migManager:
  enabled: true
  env:
    - name: WITH_REBOOT
      value: "true"
  config:
    create: true
    name: custom-mig-parted-configs
    default: "all-disabled"
```

```
    data:
      config.yaml: |-
        version: v1
        mig-configs:
          all-disabled:
            - devices: all
              mig-enabled: false

          # P4D profiles (A100 40GB)
          p4d-half-balanced:
            - devices: [0, 1, 2, 3]
              mig-enabled: true
              mig-devices:
                "1g.5gb": 2
                "2g.10gb": 1
                "3g.20gb": 1
            - devices: [4, 5, 6, 7]
              mig-enabled: false

          # P4DE profiles (A100 80GB)
          p4de-half-balanced:
            - devices: [0, 1, 2, 3]
              mig-enabled: true
              mig-devices:
                "1g.10gb": 2
                "2g.20gb": 1
                "3g.40gb": 1
            - devices: [4, 5, 6, 7]
              mig-enabled: false

devicePlugin:
  enabled: true
  config:
    name: ""
    create: false
    default: ""

toolkit:
  enabled: true

nfd:
  enabled: true

gfd:
```

```
  enabled: true

dcgmExporter:
  enabled: true
  serviceMonitor:
    enabled: true
    interval: 15s
    honorLabels: false
    additionalLabels:
      release: kube-prometheus-stack

nodeStatusExporter:
  enabled: false

operator:
  defaultRuntime: containerd
  runtimeClass: nvidia
  resources:
    limits:
      cpu: 500m
      memory: 350Mi
    requests:
      cpu: 200m
      memory: 100Mi

daemonsets:
  tolerations:
    - key: "nvidia.com/gpu"
      operator: "Exists"
      effect: "NoSchedule"
  nodeSelector:
    accelerator: nvidia
  priorityClassName: system-node-critical
```

3. `gpu-operator-values.yaml` ファイルを使用して GPU Operator をインストールします。

```
helm install gpu-operator nvidia/gpu-operator \
  --namespace gpu-operator \
  --create-namespace \
  --version v25.3.1 \
  --values gpu-operator-values.yaml
```

この Helm チャートは、以下のコンポーネントと複数の MIG プロファイルをデプロイします。

- Device Plugin (GPU リソーススケジューリング)
- DCGM Exporter (GPU メトリクスとモニタリング)
- Node Feature Discovery (NFD - ハードウェアラベル付け)
- GPU Feature Discovery (GFD - GPU 固有のラベル付け)
- MIG Manager (マルチインスタンス GPU パーティショニング)
- Container Toolkit (GPU コンテナランタイム)
- オペレーターコントローラー (ライフサイクル管理)

4. デプロイポッドを確認します。

```
kubectl get pods -n gpu-operator
```

出力例を次に示します。

```
NAME                                                          READY   STATUS
 RESTARTS        AGE
gpu-feature-discovery-27rdq                                   1/1     Running
 0               3h31m
gpu-operator-555774698d-48brn                                 1/1     Running
 0               4h8m
nvidia-container-toolkit-daemonset-sxmh9                      1/1     Running
 1 (3h32m ago)   4h1m
nvidia-cuda-validator-qb77g                                   0/1     Completed
 0               3h31m
nvidia-dcgm-exporter-cvzd7                                    1/1     Running
 0               3h31m
nvidia-device-plugin-daemonset-5ljm5                          1/1     Running
 0               3h31m
nvidia-gpu-operator-node-feature-discovery-gc-67f66fc557-q5wkt    1/1     Running
 0               4h8m
nvidia-gpu-operator-node-feature-discovery-master-5d8ffddcsl6s6   1/1     Running
 0               4h8m
nvidia-gpu-operator-node-feature-discovery-worker-6t4w7           1/1     Running
 1 (3h32m ago)   4h1m
nvidia-gpu-operator-node-feature-discovery-worker-9w7g8           1/1     Running
 0               4h8m
nvidia-gpu-operator-node-feature-discovery-worker-k5fgs           1/1     Running
 0               4h8m
nvidia-mig-manager-zvf54                                      1/1     Running
 1 (3h32m ago)   3h35m
```

5. MIG の例をテストするための p4De マネージドノードグループを使用して Amazon EKS クラスターを作成します。

```
apiVersion: eksctl.io/v1alpha5
kind: ClusterConfig

metadata:
  name: dra-eks-cluster
  region: us-east-1
  version: '1.33'

managedNodeGroups:
# P4DE MIG Node Group with Capacity Block Reservation
- name: p4de-mig-nodes
  amiFamily: AmazonLinux2023
  instanceType: p4de.24xlarge

  # Capacity settings
  desiredCapacity: 0
  minSize: 0
  maxSize: 1

  # Use specific subnet in us-east-1b for capacity reservation
  subnets:
    - us-east-1b

  # AL2023 NodeConfig for RAID0 local storage only
  nodeadmConfig:
    apiVersion: node.eks.aws/v1alpha1
    kind: NodeConfig
    spec:
      instance:
        localStorage:
          strategy: RAID0

  # Node labels for MIG configuration
  labels:
    nvidia.com/gpu.present: "true"
    nvidia.com/gpu.product: "A100-SXM4-80GB"
    nvidia.com/mig.config: "p4de-half-balanced"
    node-type: "p4de"
    vpc.amazonaws.com/efa.present: "true"
    accelerator: "nvidia"
```

```
  # Node taints
  taints:
    - key: nvidia.com/gpu
      value: "true"
      effect: NoSchedule

  # EFA support
  efaEnabled: true

  # Placement group for high-performance networking
  placementGroup:
    groupName: p4de-placement-group
    strategy: cluster

  # Capacity Block Reservation (CBR)
  # Ensure CBR ID matches the subnet AZ with the Nodegroup subnet
  spot: false
  capacityReservation:
    capacityReservationTarget:
      capacityReservationId: "cr-abcdefghij"  # Replace with your capacity
 reservation ID
```

NVIDIA GPU Operator はノードに追加されたラベルを使用しnvidia.com/mig.config: "p4de-half-balanced"、指定されたプロファイルで GPU をパーティション化します。

6. p4de インスタンスにログインします。
7. 次のコマンドを実行してください。

```
nvidia-smi -L
```

次の出力例が表示されます。

```
[root@ip-100-64-173-145 bin]# nvidia-smi -L
GPU 0: NVIDIA A100-SXM4-80GB (UUID: GPU-ab52e33c-be48-38f2-119e-b62b9935925a)
  MIG 3g.40gb     Device  0: (UUID: MIG-da972af8-a20a-5f51-849f-bc0439f7970e)
  MIG 2g.20gb     Device  1: (UUID: MIG-7f9768b7-11a6-5de9-a8aa-e9c424400da4)
  MIG 1g.10gb     Device  2: (UUID: MIG-498adad6-6cf7-53af-9d1a-10cfd1fa53b2)
  MIG 1g.10gb     Device  3: (UUID: MIG-3f55ef65-1991-571a-ac50-0dbf50d80c5a)
GPU 1: NVIDIA A100-SXM4-80GB (UUID: GPU-0eabeccc-7498-c282-0ac7-d3c09f6af0c8)
  MIG 3g.40gb     Device  0: (UUID: MIG-80543849-ea3b-595b-b162-847568fe6e0e)
  MIG 2g.20gb     Device  1: (UUID: MIG-3af1958f-fac4-59f1-8477-9f8d08c55029)
```

```
  MIG 1g.10gb     Device  2: (UUID: MIG-401088d2-716f-527b-a970-b1fc7a4ac6b2)
  MIG 1g.10gb     Device  3: (UUID: MIG-8c56c75e-5141-501c-8f43-8cf22f422569)
GPU 2: NVIDIA A100-SXM4-80GB (UUID: GPU-1c7a1289-243f-7872-a35c-1d2d8af22dd0)
  MIG 3g.40gb     Device  0: (UUID: MIG-e9b44486-09fc-591a-b904-0d378caf2276)
  MIG 2g.20gb     Device  1: (UUID: MIG-ded93941-9f64-56a3-a9b1-a129c6edf6e4)
  MIG 1g.10gb     Device  2: (UUID: MIG-6c317d83-a078-5c25-9fa3-c8308b379aa1)
  MIG 1g.10gb     Device  3: (UUID: MIG-2b070d39-d4e9-5b11-bda6-e903372e3d08)
GPU 3: NVIDIA A100-SXM4-80GB (UUID: GPU-9a6250e2-5c59-10b7-2da8-b61d8a937233)
  MIG 3g.40gb     Device  0: (UUID: MIG-20e3cd87-7a57-5f1b-82e7-97b14ab1a5aa)
  MIG 2g.20gb     Device  1: (UUID: MIG-04430354-1575-5b42-95f4-bda6901f1ace)
  MIG 1g.10gb     Device  2: (UUID: MIG-d62ec8b6-e097-5e99-a60c-abf8eb906f91)
  MIG 1g.10gb     Device  3: (UUID: MIG-fce20069-2baa-5dd4-988a-cead08348ada)
GPU 4: NVIDIA A100-SXM4-80GB (UUID: GPU-5d09daf0-c2eb-75fd-3919-7ad8fafa5f86)
GPU 5: NVIDIA A100-SXM4-80GB (UUID: GPU-99194e04-ab2a-b519-4793-81cb2e8e9179)
GPU 6: NVIDIA A100-SXM4-80GB (UUID: GPU-c1a1910f-465a-e16f-5af1-c6aafe499cd6)
GPU 7: NVIDIA A100-SXM4-80GB (UUID: GPU-c2cfafbc-fd6e-2679-e955-2a9e09377f78)
```

NVIDIA GPU Operator は P4DE インスタンスに p4de-half-balanced MIG プロファイルを正常に適用し、設定に従ってハードウェアレベルの GPU パーティションを作成しました。パーティショニングの仕組みは次のとおりです。

GPU Operator は、埋め込み MIG プロファイルからこの設定を適用しました。

```
p4de-half-balanced:
  - devices: [0, 1, 2, 3]        # First 4 GPUs: MIG enabled
    mig-enabled: true
    mig-devices:
      "1g.10gb": 2               # 2x small instances (10GB each)
      "2g.20gb": 1               # 1x medium instance (20GB)
      "3g.40gb": 1               # 1x large instance (40GB)
  - devices: [4, 5, 6, 7]        # Last 4 GPUs: Full GPUs
    mig-enabled: false
```

nvidia-smi -L 出力から、GPU オペレーターが作成した内容は次のとおりです。

- MIG 対応 GPUs (0 ~ 3): ハードウェアパーティション分割
  - GPU 0: NVIDIA A100-SXM4-80GB
    - MIG 3g.40gb デバイス 0 – 大規模なワークロード (40GB メモリ、42 SMs)
    - MIG 2g.20gb デバイス 1 – 中ワークロード (20GB メモリ、28 SMs)
    - MIG 1g.10gb デバイス 2 – 小規模なワークロード (10GB メモリ、14 SMs)

    - MIG 1g.10gb デバイス 3 – 小規模なワークロード (10GB メモリ、14 SMs)
  - GPU 1: NVIDIA A100-SXM4-80GB
    - MIG 3g.40gb デバイス 0 – 同一パーティションレイアウト
    - MIG 2g.20gb デバイス 1
    - MIG 1g.10gb デバイス 2
    - MIG 1g.10gb デバイス 3
  - GPU 2 および GPU 3 – GPU 0 および GPU 1 と同じパターン
- フル GPUs (4 ~ 7): MIG パーティショニングなし
  - GPU 4: NVIDIA A100-SXM4-80GB – フル 80GB GPU
  - GPU 5: NVIDIA A100-SXM4-80GB – フル 80GB GPU
  - GPU 6: NVIDIA A100-SXM4-80GB – フル 80GB GPU
  - GPU 7: NVIDIA A100-SXM4-80GB – フル 80GB GPU

NVIDIA GPU Operator が MIG パーティションを作成すると、NVIDIA DRA ドライバーはこれらのハードウェア分離インスタンスを自動的に検出し、Kubernetes での動的リソース割り当てを可能にします。DRA ドライバーは、特定のプロファイル (1g.10gb、2g.20gb、3g.40gb) を持つ各 MIG インスタンスを検出し、`mig.nvidia.com`デバイスクラスを介してスケジューリング可能なリソースとして公開します。

DRA ドライバーは MIG トポロジを継続的にモニタリングし、すべての GPUs で使用可能なインスタンスのインベントリを維持します。Pod が を介して特定の MIG プロファイルをリクエストすると`ResourceClaimTemplate`、DRA ドライバーは利用可能な GPU から適切な MIG インスタンスをインテリジェントに選択し、真のハードウェアレベルのマルチテナンシーを有効にします。この動的割り当てにより、複数の分離されたワークロードを同じ物理 GPU で同時に実行しながら、厳密なリソース境界とパフォーマンスの保証を維持できます。

ステップ 2: MIG リソース割り当てをテストする

次に、いくつかの例を実行して、DRA が MIG インスタンスをさまざまなワークロードに動的に割り当てる方法を示します。`resourceclaimtemplates` ポッドとテストポッドをデプロイして、DRA ドライバーが使用可能な MIG パーティションにワークロードを配置する方法を確認し、複数のコンテナがハードウェアレベルの分離で GPU リソースを共有できるようにします。

1. MIG を含める`mig-claim-template.yaml`ために を作成します`resourceclaimtemplates`。

```
apiVersion: v1
```

```
kind: Namespace
metadata:
  name: mig-gpu

---
# Template for 3g.40gb MIG instance (Large training)
apiVersion: resource.k8s.io/v1beta1
kind: ResourceClaimTemplate
metadata:
  name: mig-large-template
  namespace: mig-gpu
spec:
  spec:
    devices:
      requests:
      - name: mig-large
        deviceClassName: mig.nvidia.com
        selectors:
        - cel:
            expression: |
              device.attributes['gpu.nvidia.com'].profile == '3g.40gb'

---
# Template for 2g.20gb MIG instance (Medium training)
apiVersion: resource.k8s.io/v1beta1
kind: ResourceClaimTemplate
metadata:
  name: mig-medium-template
  namespace: mig-gpu
spec:
  spec:
    devices:
      requests:
      - name: mig-medium
        deviceClassName: mig.nvidia.com
        selectors:
        - cel:
            expression: |
              device.attributes['gpu.nvidia.com'].profile == '2g.20gb'

---
# Template for 1g.10gb MIG instance (Small inference)
apiVersion: resource.k8s.io/v1beta1
kind: ResourceClaimTemplate
```

```
metadata:
  name: mig-small-template
  namespace: mig-gpu
spec:
  spec:
    devices:
      requests:
      - name: mig-small
        deviceClassName: mig.nvidia.com
        selectors:
        - cel:
            expression: |
              device.attributes['gpu.nvidia.com'].profile == '1g.10gb'
```

2. 3 つのテンプレートを適用します。

```
kubectl apply -f mig-claim-template.yaml
```

3. 次のコマンドを実行してください。

```
kubectl get resourceclaimtemplates -n mig-gpu
```

出力例を次に示します。

```
NAME                 AGE
mig-large-template   71m
mig-medium-template  71m
mig-small-template   71m
```

4. を作成して`mig-pod.yaml`、この を活用する複数のジョブをスケジュールします`resourceclaimtemplates`。

```
---
# ConfigMap containing Python scripts for MIG pods
apiVersion: v1
kind: ConfigMap
metadata:
  name: mig-scripts-configmap
  namespace: mig-gpu
data:
  large-training-script.py: |
    import torch
```

```
    import torch.nn as nn
    import torch.optim as optim
    import time
    import os

    print(f"=== LARGE TRAINING POD (3g.40gb) ===")
    print(f"Process ID: {os.getpid()}")
    print(f"GPU available: {torch.cuda.is_available()}")
    print(f"GPU count: {torch.cuda.device_count()}")

    if torch.cuda.is_available():
        device = torch.cuda.current_device()
        print(f"Using GPU: {torch.cuda.get_device_name(device)}")
        print(f"GPU Memory: {torch.cuda.get_device_properties(device).total_memory /
1e9:.1f} GB")

        # Large model for 3g.40gb instance
        model = nn.Sequential(
            nn.Linear(2048, 1024),
            nn.ReLU(),
            nn.Linear(1024, 512),
            nn.ReLU(),
            nn.Linear(512, 256),
            nn.ReLU(),
            nn.Linear(256, 10)
        ).cuda()

        optimizer = optim.Adam(model.parameters())
        criterion = nn.CrossEntropyLoss()

        print(f"Model parameters: {sum(p.numel() for p in model.parameters())}")

        # Training loop
        for epoch in range(100):
            # Large batch for 3g.40gb
            x = torch.randn(256, 2048).cuda()
            y = torch.randint(0, 10, (256,)).cuda()

            optimizer.zero_grad()
            output = model(x)
            loss = criterion(output, y)
            loss.backward()
            optimizer.step()
```

```
            if epoch % 10 == 0:
                print(f"Large Training - Epoch {epoch}, Loss: {loss.item():.4f}, GPU
Memory: {torch.cuda.memory_allocated()/1e9:.2f}GB")
            time.sleep(3)

        print("Large training completed on 3g.40gb MIG instance")

 medium-training-script.py: |
   import torch
   import torch.nn as nn
   import torch.optim as optim
   import time
   import os

   print(f"=== MEDIUM TRAINING POD (2g.20gb) ===")
   print(f"Process ID: {os.getpid()}")
   print(f"GPU available: {torch.cuda.is_available()}")
   print(f"GPU count: {torch.cuda.device_count()}")

   if torch.cuda.is_available():
       device = torch.cuda.current_device()
       print(f"Using GPU: {torch.cuda.get_device_name(device)}")
       print(f"GPU Memory: {torch.cuda.get_device_properties(device).total_memory /
1e9:.1f} GB")

       # Medium model for 2g.20gb instance
       model = nn.Sequential(
           nn.Linear(1024, 512),
           nn.ReLU(),
           nn.Linear(512, 256),
           nn.ReLU(),
           nn.Linear(256, 10)
       ).cuda()

       optimizer = optim.Adam(model.parameters())
       criterion = nn.CrossEntropyLoss()

       print(f"Model parameters: {sum(p.numel() for p in model.parameters())}")

       # Training loop
       for epoch in range(100):
           # Medium batch for 2g.20gb
           x = torch.randn(128, 1024).cuda()
           y = torch.randint(0, 10, (128,)).cuda()
```

```
            optimizer.zero_grad()
            output = model(x)
            loss = criterion(output, y)
            loss.backward()
            optimizer.step()

            if epoch % 10 == 0:
                print(f"Medium Training - Epoch {epoch}, Loss: {loss.item():.4f}, GPU
Memory: {torch.cuda.memory_allocated()/1e9:.2f}GB")
            time.sleep(4)

        print("Medium training completed on 2g.20gb MIG instance")

 small-inference-script.py: |
   import torch
   import torch.nn as nn
   import time
   import os

   print(f"=== SMALL INFERENCE POD (1g.10gb) ===")
   print(f"Process ID: {os.getpid()}")
   print(f"GPU available: {torch.cuda.is_available()}")
   print(f"GPU count: {torch.cuda.device_count()}")

   if torch.cuda.is_available():
       device = torch.cuda.current_device()
       print(f"Using GPU: {torch.cuda.get_device_name(device)}")
       print(f"GPU Memory: {torch.cuda.get_device_properties(device).total_memory /
1e9:.1f} GB")

       # Small model for 1g.10gb instance
       model = nn.Sequential(
           nn.Linear(512, 256),
           nn.ReLU(),
           nn.Linear(256, 10)
       ).cuda()

       print(f"Model parameters: {sum(p.numel() for p in model.parameters())}")

       # Inference loop
       for i in range(200):
           with torch.no_grad():
               # Small batch for 1g.10gb
```

```
                x = torch.randn(32, 512).cuda()
                output = model(x)
                prediction = torch.argmax(output, dim=1)

                if i % 20 == 0:
                    print(f"Small Inference - Batch {i}, Predictions:
 {prediction[:5].tolist()}, GPU Memory: {torch.cuda.memory_allocated()/1e9:.2f}GB")
            time.sleep(2)

        print("Small inference completed on 1g.10gb MIG instance")

---
# Pod 1: Large training workload (3g.40gb)
apiVersion: v1
kind: Pod
metadata:
  name: mig-large-training-pod
  namespace: mig-gpu
  labels:
    app: mig-large-training
    workload-type: training
spec:
  restartPolicy: Never
  containers:
  - name: large-training-container
    image: nvcr.io/nvidia/pytorch:25.04-py3
    command: ["python", "/scripts/large-training-script.py"]
    volumeMounts:
    - name: script-volume
      mountPath: /scripts
      readOnly: true
    resources:
      claims:
      - name: mig-large-claim
  resourceClaims:
  - name: mig-large-claim
    resourceClaimTemplateName: mig-large-template
  nodeSelector:
    node.kubernetes.io/instance-type: p4de.24xlarge
    nvidia.com/gpu.present: "true"
  tolerations:
  - key: nvidia.com/gpu
    operator: Exists
    effect: NoSchedule
```

```
  volumes:
  - name: script-volume
    configMap:
      name: mig-scripts-configmap
      defaultMode: 0755

---
# Pod 2: Medium training workload (2g.20gb) - can run on SAME GPU as Pod 1
apiVersion: v1
kind: Pod
metadata:
  name: mig-medium-training-pod
  namespace: mig-gpu
  labels:
    app: mig-medium-training
    workload-type: training
spec:
  restartPolicy: Never
  containers:
  - name: medium-training-container
    image: nvcr.io/nvidia/pytorch:25.04-py3
    command: ["python", "/scripts/medium-training-script.py"]
    volumeMounts:
    - name: script-volume
      mountPath: /scripts
      readOnly: true
    resources:
      claims:
      - name: mig-medium-claim
  resourceClaims:
  - name: mig-medium-claim
    resourceClaimTemplateName: mig-medium-template
  nodeSelector:
    node.kubernetes.io/instance-type: p4de.24xlarge
    nvidia.com/gpu.present: "true"
  tolerations:
  - key: nvidia.com/gpu
    operator: Exists
    effect: NoSchedule
  volumes:
  - name: script-volume
    configMap:
      name: mig-scripts-configmap
      defaultMode: 0755
```

```
---
# Pod 3: Small inference workload (1g.10gb) - can run on SAME GPU as Pod 1 & 2
apiVersion: v1
kind: Pod
metadata:
  name: mig-small-inference-pod
  namespace: mig-gpu
  labels:
    app: mig-small-inference
    workload-type: inference
spec:
  restartPolicy: Never
  containers:
  - name: small-inference-container
    image: nvcr.io/nvidia/pytorch:25.04-py3
    command: ["python", "/scripts/small-inference-script.py"]
    volumeMounts:
    - name: script-volume
      mountPath: /scripts
      readOnly: true
    resources:
      claims:
      - name: mig-small-claim
  resourceClaims:
  - name: mig-small-claim
    resourceClaimTemplateName: mig-small-template
  nodeSelector:
    node.kubernetes.io/instance-type: p4de.24xlarge
    nvidia.com/gpu.present: "true"
  tolerations:
  - key: nvidia.com/gpu
    operator: Exists
    effect: NoSchedule
  volumes:
  - name: script-volume
    configMap:
      name: mig-scripts-configmap
      defaultMode: 0755
```

5. この仕様を適用し、3 つの Pod をデプロイします。

```
kubctl apply -f mig-pod.yaml
```

これらのポッドは DRA ドライバーによってスケジュールする必要があります。

6. DRA ドライバー Pod ログを確認すると、次のような出力が表示されます。

```
I0717 21:50:22.925811 1 driver.go:87] NodePrepareResource is called: number of
 claims: 1
I0717 21:50:22.932499 1 driver.go:129] Returning newly prepared devices
 for claim '933e9c72-6fd6-49c5-933c-a896407dc6d1': [&Device{RequestNames:
[mig-large],PoolName:ip-100-64-173-145.ec2.internal,DeviceName:gpu-0-
mig-9-4-4,CDIDeviceIDs:[k8s.gpu.nvidia.com/device=**gpu-0-mig-9-4-4**],}]
I0717 21:50:23.186472 1 driver.go:87] NodePrepareResource is called: number of
 claims: 1
I0717 21:50:23.191226 1 driver.go:129] Returning newly prepared devices
 for claim '61e5ddd2-8c2e-4c19-93ae-d317fecb44a4': [&Device{RequestNames:
[mig-medium],PoolName:ip-100-64-173-145.ec2.internal,DeviceName:gpu-2-
mig-14-0-2,CDIDeviceIDs:[k8s.gpu.nvidia.com/device=**gpu-2-mig-14-0-2**],}]
I0717 21:50:23.450024 1 driver.go:87] NodePrepareResource is called: number of
 claims: 1
I0717 21:50:23.455991 1 driver.go:129] Returning newly prepared devices
 for claim '1eda9b2c-2ea6-401e-96d0-90e9b3c111b5': [&Device{RequestNames:
[mig-small],PoolName:ip-100-64-173-145.ec2.internal,DeviceName:gpu-1-
mig-19-2-1,CDIDeviceIDs:[k8s.gpu.nvidia.com/device=**gpu-1-mig-19-2-1**],}]
```

7. を確認しresourceclaims、Pod のステータスを確認します。

```
kubectl get resourceclaims -n mig-gpu -w
```

出力例を次に示します。

```
NAME                                             STATE                AGE
mig-large-training-pod-mig-large-claim-6dpn8     pending              0s
mig-large-training-pod-mig-large-claim-6dpn8     pending              0s
mig-large-training-pod-mig-large-claim-6dpn8     allocated,reserved   0s
mig-medium-training-pod-mig-medium-claim-bk596   pending              0s
mig-medium-training-pod-mig-medium-claim-bk596   pending              0s
mig-medium-training-pod-mig-medium-claim-bk596   allocated,reserved   0s
mig-small-inference-pod-mig-small-claim-d2t58    pending              0s
mig-small-inference-pod-mig-small-claim-d2t58    pending              0s
mig-small-inference-pod-mig-small-claim-d2t58    allocated,reserved   0s
```

ご覧のとおり、すべての Pod は DRA ドライバーallocated,reservedによって保留中から に移動しました。

8. ノードnvidia-smiから を実行します。3 つの Python プロセッサが実行されていることがわかります。

```
root@ip-100-64-173-145 bin]# nvidia-smi
+-----------------------------------------------------------------------------------------
+
| NVIDIA-SMI 570.158.01 Driver Version: 570.158.01 CUDA Version: 12.8 |
|-----------------------------------------+------------------------
+----------------------+
| GPU Name Persistence-M | Bus-Id Disp.A | Volatile Uncorr. ECC |
| Fan Temp Perf Pwr:Usage/Cap | Memory-Usage | GPU-Util Compute M. |
| | | MIG M. |
|=========================================+========================
+======================|
| 0 NVIDIA A100-SXM4-80GB On | 00000000:10:1C.0 Off | On |
| N/A 63C P0 127W / 400W | 569MiB / 81920MiB | N/A Default |
| | | Enabled |
+-----------------------------------------+------------------------
+----------------------+
| 1 NVIDIA A100-SXM4-80GB On | 00000000:10:1D.0 Off | On |
| N/A 56C P0 121W / 400W | 374MiB / 81920MiB | N/A Default |
| | | Enabled |
+-----------------------------------------+------------------------
+----------------------+
| 2 NVIDIA A100-SXM4-80GB On | 00000000:20:1C.0 Off | On |
| N/A 63C P0 128W / 400W | 467MiB / 81920MiB | N/A Default |
| | | Enabled |
+-----------------------------------------+------------------------
+----------------------+
| 3 NVIDIA A100-SXM4-80GB On | 00000000:20:1D.0 Off | On |
| N/A 57C P0 118W / 400W | 249MiB / 81920MiB | N/A Default |
| | | Enabled |
+-----------------------------------------+------------------------
+----------------------+
| 4 NVIDIA A100-SXM4-80GB On | 00000000:90:1C.0 Off | 0 |
| N/A 51C P0 77W / 400W | 0MiB / 81920MiB | 0% Default |
| | | Disabled |
+-----------------------------------------+------------------------
+----------------------+
| 5 NVIDIA A100-SXM4-80GB On | 00000000:90:1D.0 Off | 0 |
```

```
| N/A 46C P0 69W / 400W | 0MiB / 81920MiB | 0% Default |
| | | Disabled |
+-----------------------------------------+-----------------------
+----------------------+
| 6 NVIDIA A100-SXM4-80GB On | 00000000:A0:1C.0 Off | 0 |
| N/A 52C P0 74W / 400W | 0MiB / 81920MiB | 0% Default |
| | | Disabled |
+-----------------------------------------+-----------------------
+----------------------+
| 7 NVIDIA A100-SXM4-80GB On | 00000000:A0:1D.0 Off | 0 |
| N/A 47C P0 72W / 400W | 0MiB / 81920MiB | 0% Default |
| | | Disabled |
+-----------------------------------------+-----------------------
+----------------------+



+-----------------------------------------------------------------------------------------------------------
+
| MIG devices: |
+------------------+----------------------------------+-----------
+-----------------------+
| GPU GI CI MIG | Memory-Usage | Vol| Shared |
| ID ID Dev | BAR1-Usage | SM Unc| CE ENC DEC OFA JPG |
| | | ECC| |
|==================+==================================+===========
+=======================|
| 0 2 0 0 | 428MiB / 40192MiB | 42 0 | 3 0 2 0 0 |
| | 2MiB / 32767MiB | | |
+------------------+----------------------------------+-----------
+-----------------------+
| 0 3 0 1 | 71MiB / 19968MiB | 28 0 | 2 0 1 0 0 |
| | 0MiB / 16383MiB | | |
+------------------+----------------------------------+-----------
+-----------------------+
| 0 9 0 2 | 36MiB / 9728MiB | 14 0 | 1 0 0 0 0 |
| | 0MiB / 8191MiB | | |
+------------------+----------------------------------+-----------
+-----------------------+
| 0 10 0 3 | 36MiB / 9728MiB | 14 0 | 1 0 0 0 0 |
| | 0MiB / 8191MiB | | |
+------------------+----------------------------------+-----------
+-----------------------+
| 1 1 0 0 | 107MiB / 40192MiB | 42 0 | 3 0 2 0 0 |
| | 0MiB / 32767MiB | | |
```

```
+------------------+-----------------------------------+-----------
+-----------------------+
| 1 5 0 1 | 71MiB / 19968MiB | 28 0 | 2 0 1 0 0 |
| | 0MiB / 16383MiB | | |
+------------------+-----------------------------------+-----------
+-----------------------+
| 1 13 0 2 | 161MiB / 9728MiB | 14 0 | 1 0 0 0 0 |
| | 2MiB / 8191MiB | | |
+------------------+-----------------------------------+-----------
+-----------------------+
| 1 14 0 3 | 36MiB / 9728MiB | 14 0 | 1 0 0 0 0 |
| | 0MiB / 8191MiB | | |
+------------------+-----------------------------------+-----------
+-----------------------+
| 2 1 0 0 | 107MiB / 40192MiB | 42 0 | 3 0 2 0 0 |
| | 0MiB / 32767MiB | | |
+------------------+-----------------------------------+-----------
+-----------------------+
| 2 5 0 1 | 289MiB / 19968MiB | 28 0 | 2 0 1 0 0 |
| | 2MiB / 16383MiB | | |
+------------------+-----------------------------------+-----------
+-----------------------+
| 2 13 0 2 | 36MiB / 9728MiB | 14 0 | 1 0 0 0 0 |
| | 0MiB / 8191MiB | | |
+------------------+-----------------------------------+-----------
+-----------------------+
| 2 14 0 3 | 36MiB / 9728MiB | 14 0 | 1 0 0 0 0 |
| | 0MiB / 8191MiB | | |
+------------------+-----------------------------------+-----------
+-----------------------+
| 3 1 0 0 | 107MiB / 40192MiB | 42 0 | 3 0 2 0 0 |
| | 0MiB / 32767MiB | | |
+------------------+-----------------------------------+-----------
+-----------------------+
| 3 5 0 1 | 71MiB / 19968MiB | 28 0 | 2 0 1 0 0 |
| | 0MiB / 16383MiB | | |
+------------------+-----------------------------------+-----------
+-----------------------+
| 3 13 0 2 | 36MiB / 9728MiB | 14 0 | 1 0 0 0 0 |
| | 0MiB / 8191MiB | | |
+------------------+-----------------------------------+-----------
+-----------------------+
| 3 14 0 3 | 36MiB / 9728MiB | 14 0 | 1 0 0 0 0 |
| | 0MiB / 8191MiB | | |
```

```
+------------------+-------------------------------------+-----------
+-----------------------+



+-----------------------------------------------------------------------------------------
+
| Processes: |
| GPU GI CI PID Type Process name GPU Memory |
| ID ID Usage |
|
=========================================================================================|
**| 0 2 0 64080 C python 312MiB |
| 1 13 0 64085 C python 118MiB |
| 2 5 0 64073 C python 210MiB |**
+-----------------------------------------------------------------------------------------
+
```

GB200 P6e インスタンスを使用して IMEX で GPU ワークロードを最適化する

IMEX (Internode Memory Exchange) は、NVIDIA GB200 UltraServers での分散トレーニングのために、ノード間のメモリコヒーレント通信を可能にします。

次の手順を実行します。

1. という名前のファイルを使用して、マルチノードトレーニングComputeDomain用の を定義しますimex-compute-domain.yaml。

```
apiVersion: resource.nvidia.com/v1beta1
kind: ComputeDomain
metadata:
  name: distributed-training-domain
  namespace: default
spec:
  numNodes: 2
  channel:
    resourceClaimTemplate:
      name: imex-channel-template
```

2. という名前のファイルで IMEX チャネルを使用してポッドを定義しますimex-pod.yaml。

```
apiVersion: v1
kind: Pod
```

```
metadata:
  name: imex-distributed-training
  namespace: default
  labels:
    app: imex-training
spec:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
        - matchExpressions:
          - key: nvidia.com/gpu.clique
            operator: Exists
  containers:
  - name: distributed-training
    image: nvcr.io/nvidia/pytorch:25.04-py3
    command: ["bash", "-c"]
    args:
    - |
      echo "=== IMEX Channel Verification ==="
      ls -la /dev/nvidia-caps-imex-channels/
      echo ""

      echo "=== GPU Information ==="
      nvidia-smi
      echo ""

      echo "=== NCCL Test (if available) ==="
      python -c "
      import torch
      import torch.distributed as dist
      import os

      print(f'CUDA available: {torch.cuda.is_available()}')
      print(f'CUDA device count: {torch.cuda.device_count()}')

      if torch.cuda.is_available():
          for i in range(torch.cuda.device_count()):
              print(f'GPU {i}: {torch.cuda.get_device_name(i)}')

      # Check for IMEX environment variables
      imex_vars = [k for k in os.environ.keys() if 'IMEX' in k or 'NVLINK' in k]
      if imex_vars:
          print('IMEX Environment Variables:')
```

```
          for var in imex_vars:
              print(f'  {var}={os.environ[var]}')

      print('IMEX channel verification completed')
      "

      # Keep container running for inspection
      sleep 3600
    resources:
      claims:
      - name: imex-channel-0
      - name: imex-channel-1
  resourceClaims:
  - name: imex-channel-0
    resourceClaimTemplateName: imex-channel-template
  - name: imex-channel-1
    resourceClaimTemplateName: imex-channel-template
  tolerations:
  - key: nvidia.com/gpu
    operator: Exists
    effect: NoSchedule
```

> **Note**
>
> これには P6e GB200 インスタンスが必要です。

3. ComputeDomain および テンプレートを適用して IMEX をデプロイします。

```
kubectl apply -f imex-claim-template.yaml
kubectl apply -f imex-compute-domain.yaml
kubectl apply -f imex-pod.yaml
```

4. ComputeDomain ステータスを確認します。

```
kubectl get computedomain distributed-training-domain
```

5. IMEX デーモンのデプロイをモニタリングします。

```
kubectl get pods -n nvidia-dra-driver -l resource.nvidia.com/computeDomain
```

6. Pod の IMEX チャネルを確認します。

```
kubectl exec imex-distributed-training -- ls -la /dev/nvidia-caps-imex-channels/
```

7. Pod ログを表示します。

```
kubectl logs imex-distributed-training
```

以下は、予想される出力の例です。

```
=== IMEX Channel Verification ===
total 0
drwxr-xr-x. 2 root root 80 Jul 8 10:45 .
drwxr-xr-x. 6 root root 380 Jul 8 10:45 ..
crw-rw-rw-. 1 root root 241, 0 Jul 8 10:45 channel0
crw-rw-rw-. 1 root root 241, 1 Jul 8 10:45 channel1
```

詳細については、GitHub の NVIDIA の例を参照してください。

ネットワーク

ノード間通信が高いアプリケーションには、より高いネットワーク帯域幅または Elastic Fabric Adapter を検討する

ノード間通信の需要が高い Amazon EKS の分散トレーニングワークロードの場合は、ネットワーク帯域幅が高いインスタンスまたは Elastic Fabric Adapter (EFA) を選択することを検討してください。ネットワークパフォーマンスが不十分な場合、データ転送がボトルネックになり、分散マルチ GPU トレーニングなどの機械学習タスクが遅くなる可能性があります。推論ワークロードでは通常、ノード間の通信は高くないことに注意してください。

例

たとえば、Karpenter を使用します。

```
apiVersion: v1
kind: Pod
metadata:
  name: ml-workload
spec:
```

```
  nodeSelector:
    karpenter.k8s.aws/instance-network-bandwidth: "100000"  # 100 Gbps in Mbps
    node.kubernetes.io/instance-type: p5.48xlarge  # EFA-enabled instance
  containers:
  - name: training-job
    image: `763104351884.dkr.ecr.us-west-2.amazonaws.com/pytorch-inference:2.6.0-gpu-
py312-cu124-ubuntu22.04-ec2-v1.6`
    resources:
      limits:
        vpc.amazonaws.com/efa: 1  # Requires EFA device plugin
```

トレーニングジョブに EFA を活用するために、MPI や NCCL などのツールがコンテナイメージにインストールされていることを確認します。

ポッドの起動時間を短縮するために使用可能な IP アドレスの数を増やす

EKS では、各ポッドに VPC CIDR ブロックの IP アドレスが必要です。クラスターがより多くのノードやポッドでスケールすると、IP アドレスの枯渇やパフォーマンスの低下のリスクがありますが、プレフィックスの委任を有効にすると、IP 範囲を事前に割り当てて EC2 API コールを減らすことで、これらの問題を軽減できるため、ポッドの起動時間が短縮され、スケーラビリティが向上します。

クラスターの作成後にプレフィックスの委任を有効にすると、VPC コンテナネットワークインターフェイス (CNI) は EC2 インスタンスのネットワークインターフェイスに IP プレフィックス (/28、それぞれ 16 個の IP アドレスを付与) を割り当てることができます。つまり、各ノードはより多くのポッドをサポートできるため、IP 不足のリスクが軽減されます。たとえば、`c5.4xlarge`インスタンスでは、プレフィックス委任を使用して最大 110 個のポッドをサポートできます。

多くの小さなポッドがある環境で IP 使用率を最適化するにはプレフィックスの委任が不可欠ですが、AI/ML ワークロードでは多くの場合、使用するポッドが少なくて大きいポッド (GPU ごとに 1 つのポッドなど) を使用します。プレフィックス委任を有効にすると、VPC CNI はウォームプールを維持することで、ポッドの起動を高速化するためにプレフィックスを事前に割り当てることができます。つまり、IP アドレスがすぐに使用可能になり、非プレフィックスモードでのオンデマンド割り当てと比較して、ポッドの初期化に必要な時間が短縮されます。このような場合、プレフィックス委任を有効にすることで IP を節約することで、AI/ML ワークロードのパフォーマンス上の利点が得られます。IP アドレス設定と IP 範囲の事前割り当てに必要な EC2 API コールの数を減らすことで、プレフィックス委任を使用すると、ポッドの起動時間が短縮され、AI/ML ワークロードを迅速にスケーリングするために特に役立ちます。

プレフィックスの委任を有効にするには：

```
kubectl set env daemonset/aws-node -n kube-system ENABLE_PREFIX_DELEGATION=true
```

特に大規模なデプロイでは、IP アドレスの枯渇を避けるために VPC サブネットを適切に計画し、VPCs 間での重複を避けるために CIDR ブロックを管理します。詳細については、「IP アドレス使用率の最適化」および「プレフィックス付きの Amazon EKS ノードへの IP アドレスの割り当て」を参照してください。

セキュリティ

セキュリティおよびコンプライアンス

暗号化準拠のストレージに S3 と KMS を検討する

特に指定しない限り、すべての S3 バケットはデフォルトで SSE-S3 を使用して保管中のオブジェクトを暗号化します。ただし、代わりに AWS Key Management Service (AWS KMS) キー (SSE-KMS) によるサーバー側の暗号化を使用するようにバケットを設定することもできます。AWS KMS のセキュリティ制御は、暗号化関連のコンプライアンス要件を満たすのに役立ちます。これらの KMS キーを使用して、Amazon S3 バケットのデータを保護できます。S3 バケットで SSE-KMS 暗号化を使用する場合、AWS KMS キーはバケットと同じリージョンに存在する必要があります。

SSE-KMS に S3 バケットキーを使用するように汎用バケットを設定し、Amazon S3 から AWS KMS へのリクエストトラフィックを減らすことで、AWS KMS リクエストコストを最大 99% 削減します。S3 バケットキーは、ディレクトリバケット内の および オペレーションで常に有効になっており、無効にすることはできません。 GET PUT

Amazon S3 Express One Zone は、S3 ディレクトリバケットと呼ばれる特定のタイプのバケットを使用することに注意してください。ディレクトリバケットは S3 Express One Zone ストレージクラス専用であり、高性能で低レイテンシーのアクセスを可能にします。S3 ディレクトリバケットでデフォルトのバケット暗号化を設定するには、次の例のように、AWS CLI を使用し、エイリアスではなく KMS キー ID または ARN を指定します。

Example

```
aws s3api put-bucket-encryption --bucket my-directory-bucket --server-side-encryption-
configuration \
   '{"Rules": [{"ApplyServerSideEncryptionByDefault": {"SSEAlgorithm": "aws:kms",
 "KMSMasterKeyID": "1234abcd-12ab-34cd-56ef-1234567890ab"}}]}'
```

EKS ポッドの IAM ロールに、暗号化されたオブジェクトにアクセスするための KMS アクセス許可 (などkms:Decrypt) があることを確認します。これをステージング環境でテストするには、サンプルモデルをバケットにアップロードし、ポッドにマウントし (Mountpoint S3 CSI ドライバー経由など)、ポッドが暗号化されたデータを エラーなしで読み取れることを確認します。AWS CloudTrail 経由でログを監査し、暗号化要件への準拠を確認します。セットアップの詳細とキー管理については、KMS ドキュメントを参照してください。

[Storage (ストレージ)]

データ管理とストレージ

CSI ドライバーを使用して AI モデルをポッドにデプロイする

AI/ML ワークロードでは、多くの場合、大規模なモデルアーティファクト (トレーニング済みの重み、設定など) にアクセスする必要があり、ポッドではコンテナイメージに埋め込まずにこれらにアクセスするための信頼性が高くスケーラブルな方法が必要であるため、イメージサイズとコンテナレジストリのプル時間が長くなる可能性があります。ボリュームマウントを管理する運用上のオーバーヘッドを減らすには、それぞれの CSI ドライバーを使用して Amazon ストレージサービス (S3、FSx for Lustre、FSx for OpenZFS、EFS など) を永続ボリューム (PVs) としてマウントして、ポッドに AI モデルをデプロイすることをお勧めします。実装の詳細については、このセクションの後続のトピックを参照してください。

EKS での ML モデルキャッシュのストレージの最適化

最適なストレージソリューションを活用することは、ポッドとアプリケーションの起動レイテンシーを最小限に抑え、メモリ使用量を削減し、必要なレベルのパフォーマンスを取得してワークロードを高速化し、ML ワークロードのスケーラビリティを確保するために不可欠です。ML ワークロードはモデルファイル (重み) に依存することが多いため、サイズが大きくなり、ポッドまたはノード間でデータへの共有アクセスが必要になる場合があります。最適なストレージソリューションの選択は、単一ノードの効率、マルチノードアクセス、レイテンシー要件、コスト制約、データ統合要件 (Amazon S3 データリポジトリなど) など、ワークロードの特性によって異なります。ワークロードでさまざまなストレージソリューションをベンチマークして、どのソリューションが要件を満たすかを把握することをお勧めします。ワークロードの要件に基づいて評価できるように、以下のオプションが用意されています。

EKS CSI ドライバーは、次の AWS ストレージサービスをサポートしています。それぞれに独自の CSI ドライバーがあり、AI および ML ワークフローに独自の強みがあります。

- Amazon S3 のマウントポイント
- Amazon FSx for Lustre
- Amazon FSx for OpenZFS
- Amazon EFS
- Amazon EBS

AWS ストレージサービスの選択は、デプロイアーキテクチャ、スケール、パフォーマンス要件、コスト戦略によって異なります。Storage CSI ドライバーは EKS クラスターにインストールする必要があります。これにより、CSI ドライバーは Pod のライフサイクル外で永続ボリューム (PV) を作成および管理できます。CSI ドライバーを使用すると、サポートされている AWS ストレージサービスの PV 定義を EKS クラスターリソースとして作成できます。その後、ポッドは PV の永続ボリュームクレーム (PVC) を作成して、データボリュームのこれらのストレージボリュームにアクセスできます。AWS ストレージサービスとデプロイシナリオに応じて、単一の PVC (および関連する PV) をワークロードの複数の Pod にアタッチできます。たとえば、ML トレーニングの場合、共有トレーニングデータは PV に保存され、複数の Pod によってアクセスされます。リアルタイムのオンライン推論の場合、LLM モデルは PV にキャッシュされ、複数の Pod によってアクセスされます。AWS ストレージサービス用のサンプル PV および PVC YAML ファイルを以下に示します。

パフォーマンスのモニタリング ディスクのパフォーマンスが低いと、コンテナイメージの読み取りが遅くなり、ポッドの起動レイテンシーが増加し、推論やトレーニングのスループットが低下する可能性があります。Amazon CloudWatch を使用して、AWS ストレージサービスのパフォーマンスメトリクスをモニタリングします。パフォーマンスのボトルネックを特定するときは、ストレージ設定パラメータを変更してパフォーマンスを最適化します。

シナリオ: 複数の GPU インスタンスのワークロード

Amazon FSx for Lustre: 分散トレーニングやモデル提供など、レイテンシーの影響を受けやすい高帯域幅スループットの動的ワークロードを持つ複数の EC2 GPU コンピューティングインスタンス環境があり、ネイティブの Amazon S3 データリポジトリ統合が必要な場合は、Amazon FSx for Lustre をお勧めします。FSx for Lustre は、高性能コンピューティング (HPC)、Machine Learningなどのコンピューティング集約型のワークロード向けに設計されたフルマネージド型の高性能並列ファイルシステムを提供します。

FSx for Lustre CSI ドライバーをインストールして FSx ファイルシステムを永続ボリューム (PV) として EKS にマウントし、FSx for Lustre ファイルシステムをスタンドアロンの高性能キャッシュとしてデプロイするか、S3-linkedファイルシステムとしてデプロイして S3 データの高性能キャッシュとして機能し、GPU コンピューティングインスタンス間のデータアクセスに高速 I/O と高スルー

プットを提供します。FSx for Lustre は、Scratch-SSD または Persistent-SSD ストレージオプションを使用してデプロイできます。

- スクラッチ SSD ストレージ: 一時的または短期間 (時間) のワークロードに推奨され、TiB あたりの固定スループット容量がプロビジョニングされます。
- 永続的 SSD ストレージ: HPC シミュレーション、ビッグデータ分析、Machine Learningトレーニングなど、最高レベルの可用性を必要とするミッションクリティカルで長時間実行されるワークロードに推奨されます。Persistent-SSD ストレージでは、必要なストレージ容量とスループット容量 (TiB あたり) の両方を設定できます。

パフォーマンスに関する考慮事項：

- FSx for Lustre ファイルシステムを管理する管理ポッド: lustre クライアントがインストールされ、FSx ファイルシステムがマウントされた「管理」ポッドを設定します。これにより、アクセスポイントは FSx ファイルシステムのファインチューニングを有効にできます。また、GPU コンピューティングインスタンスを起動する前に ML トレーニングデータまたは LLM モデルを使用して FSx ファイルシステムを事前ウォーミングする必要がある状況でも有効にできます。これは、アーキテクチャがスポットベースの Amazon EC2 GPU/コンピューティングインスタンスを使用している場合に特に重要です。ここでは、管理ポッドを使用して、必要なデータを FSx ファイルシステムに「ウォーム」または「プリロード」して、スポットベースの Amazon EC2 インスタンスの実行時にデータを処理できるようにすることができます。
- Elastic Fabric Adapter (EFA): Persistent-SSD ストレージデプロイタイプは Elastic Fabric Adapter (EFA) をサポートしています。EFA の使用は、高性能でスループットベースの GPU ベースのワークロードに最適です。FSx for Lustre は NVIDIA GPUDirect Storage (GDS) をサポートしています。ここで、GDS は、ローカルストレージまたはリモートストレージと GPU メモリ間の直接データパスを作成し、データアクセスを高速化するテクノロジーです。
- 圧縮: 圧縮できるファイルタイプがある場合は、ファイルシステムでデータ圧縮を有効にします。これにより、データ圧縮によって FSx for Lustre ファイルサーバーとストレージ間で転送されるデータ量が減るため、パフォーマンスを向上させることができます。
- Lustre ファイルシステムのストライピング設定：
  - データストライピング: FSx for Luster が Lustre ファイルシステム内の複数のオブジェクトストレージターゲット (OSTs) にファイルのデータを分散できるようにします。特に大規模な ML トレーニングジョブでは、並列アクセスとスループットを最大化します。
  - スタンドアロンファイルシステムのストライピング: デフォルトでは、FSx for Lustre のプログレッシブファイルレイアウト (PFL) 機能を介して 4 コンポーネントの Lustre ストライピング設定が作成されます。ほとんどのシナリオでは、デフォルトの PFL Lustre ストライプの数/サイズ

を更新する必要はありません。Lustre データストライピングを調整する必要がある場合は、FSx for Lustre ファイルシステムのストライピングパラメータを参照して、Lustre ストライピングを手動で調整できます。

- S3-Linkedファイルシステム: ネイティブ Amazon S3 統合 (データリポジトリ関連付けまたは DRA) を使用して FSx ファイルシステムにインポートされたファイルは、デフォルトの PFL レイアウトを使用するのではなく、ファイルシステムの `ImportedFileChunkSize`パラメータでレイアウトを使用します。より大きい S3-importedファイルは、`ImportedFileChunkSize`定義された値 (デフォルト`ImportedFileChunkSize`は 1GiB) に基づいてストライプカウントの複数の OSTs に保存されます。大きなファイルがある場合は、このパラメータをより高い値に調整することをお勧めします。
- 配置: FSx for Lustre ファイルシステムをコンピューティングノードまたは GPU ノードと同じアベイラビリティーゾーンにデプロイして、データへのレイテンシーを最小限に抑え、アベイラビリティーゾーン間のアクセスパターンを回避します。異なるアベイラビリティーゾーンに複数の GPU ノードがある場合は、低レイテンシーのデータアクセスのために、各アベイラビリティーゾーンに FSx ファイルシステムをデプロイすることをお勧めします。

例

静的プロビジョニング (FSx インスタンスが既にプロビジョニングされている場合) を使用した FSx for Lustre ファイルシステムの永続ボリューム (PV) 定義。

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: fsx-pv
spec:
  capacity:
    storage: 1200Gi
  volumeMode: Filesystem
  accessModes:
    - ReadWriteMany
  mountOptions:
    - flock
  persistentVolumeReclaimPolicy: Recycle
  csi:
    driver: fsx.csi.aws.com
    volumeHandle: [FileSystemId of FSx instance]
    volumeAttributes:
      dnsname: [DNSName of FSx instance]
```

```
      mountname: [MountName of FSx instance]
```

例

と呼ばれる PV の永続ボリュームクレーム定義fsx-pv：

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: fsx-claim
spec:
  accessModes:
    - ReadWriteMany
  storageClassName: ""
  resources:
    requests:
      storage: 1200Gi
  volumeName: fsx-pv
```

例

の永続ボリュームクレームを使用するようにポッドを設定しますfsx-claim。

```
apiVersion: v1
kind: Pod
metadata:
  name: fsx-app
spec:
  containers:
  - name: app
    image: amazonlinux:2023
    command: ["/bin/sh"]
    volumeMounts:
    - name: persistent-storage
      mountPath: /data
  volumes:
  - name: persistent-storage
    persistentVolumeClaim:
      claimName: fsx-claim
```

完全な例については、GitHub の FSx for Lustre ドライバーの例を参照してください。Amazon CloudWatch を使用して Amazon FSx for Lustre のパフォーマンスメトリクスをモニタリングします。パフォーマンスのボトルネックが特定されたら、必要に応じて設定パラメータを調整します。

シナリオ: 単一 GPU インスタンスのワークロード

Mountpoint for Amazon S3 with CSI Driver: Mountpoint for Amazon S3 CSI ドライバーを使用して、ポッドにボリュームとして S3 バケットをマウントできます。 Amazon S3 この方法では、特定の S3 バケットにアクセスできるポッドをきめ細かく制御できます。各ポッドには独自のマウントポイントインスタンスとローカルキャッシュ (5 ~ 10 GB) があり、ポッド間でモデルのロードと読み取りのパフォーマンスを分離します。この設定では、サービスアカウントの IAM ロール (IRSA) によるポッドレベルの認証と、さまざまなモデルまたは顧客の独立したモデルバージョニングがサポートされています。各ポッドは S3 API コールを発行し、独自のキャッシュを維持するため、トレードオフはメモリ使用量と API トラフィックの増加です。

例 CSI ドライバーを使用したポッドデプロイ YAML の部分的な例：

```
# CSI driver dynamically mounts the S3 bucket for each pod

volumes:
  - name: s3-mount
    csi:
      driver: s3.csi.aws.com
      volumeAttributes:
        bucketName: your-s3-bucket-name
        mountOptions: "--allow-delete"  # Optional
        region: us-west-2

containers:
  - name: inference
    image: your-inference-image
    volumeMounts:
      - mountPath: /models
        name: s3-mount
volumeMounts:
  - name: model-cache
    mountPath: /models
volumes:
  - name: model-cache
    hostPath:
      path: /mnt/s3-model-cache
```

パフォーマンスに関する考慮事項：

- データキャッシュ: Mountpoint for S3 はコンテンツをキャッシュしてコストを削減し、同じファイルへの繰り返し読み取りのパフォーマンスを向上させることができます。キャッシュオプションとパラメータについては、「キャッシュ設定」を参照してください。
- オブジェクトのパートサイズ: 72GB を超えるファイルを保存してアクセスする場合は、「Mountpoint パフォーマンスの設定」を参照して、データプロファイルとワークロード要件を満たすように `--read-part-size`および `--write-part-size` コマンドラインパラメータを設定する方法を理解します。
- 共有キャッシュは、最大 1MB のサイズのオブジェクト用に設計されています。大きなオブジェクトはサポートされていません。EKS ノードの NVMe または EBS ボリュームでオブジェクトをキャッシュするには、ローカルキャッシュオプションを使用します。
- API リクエスト料金: Mountpoint for S3 で多数のファイルオペレーションを実行すると、API リクエスト料金がストレージコストの一部になる可能性があります。これを軽減するには、強力な整合性が必要ない場合は、常にメタデータキャッシュを有効にし、API オペレーションの数を S3 に減らす`metadata-ttl`期間を設定します。

詳細については、Amazon EKS 公式ドキュメントの「Mountpoint for Amazon S3 CSI Driver」を参照してください。ボトルネックが発生した場合は、Amazon CloudWatch メトリクスを使用して Amazon S3 のパフォーマンスメトリクス Amazon CloudWatchをモニタリングし、必要に応じて設定を調整することをお勧めします。

Amazon FSx for OpenZFS 永続共有ストレージ

レイテンシーの影響を受けやすいワークロードを持つ複数の EC2 GPU コンピューティングインスタンスで、さまざまなアプリケーションに高可用性、高性能、コスト感度、複数のポッドデプロイを必要とするシナリオでは、Amazon FSx for OpenZFS をお勧めします。ワークロードの例には、リアルタイム推論、強化学習、生成攻撃者ネットワークのトレーニングなどがあります。FSx for OpenZFS は、小さな IO データアクセスパターンを使用する小さなファイルを持つ集中型ディレクトリ構造への高性能アクセスを必要とするワークロードに特に有益です。また、FSx for OpenZFS は、ストレージ容量とは別にパフォーマンスを柔軟にスケールできるため、必要なパフォーマンスレベルを維持しながら、ストレージサイズを実際のニーズに合わせることで、最適なコスト効率を実現できます。

ネイティブ FSx for OpenZFS CSI ドライバーを使用すると、複数のボリュームを作成して、単一のファイルシステムに複数の PVCs を作成できます。これにより、単一のファイルシステムに統合されたアプリケーションポッドをデプロイすることで、管理オーバーヘッドが削減され、ファイルシス

テムのスループットと IOPS の使用率が最大化されます。さらに、ゼロコピースナップショット、ゼロコピークローン、CSI ドライバーを介して動的にプロビジョニングできるユーザーおよびグループのクォータなどのエンタープライズ機能も含まれています。

FSx for OpenZFS は、作成時に 3 つの異なるデプロイタイプをサポートします。

- シングル AZ: ミリ秒未満のレイテンシーを持つ最低コストのオプションですが、ファイルシステムまたはアベイラビリティーゾーンレベルでは高可用性を提供しません。開発およびテストワークロード、またはアプリケーションレイヤーで高可用性を持つワークロードに推奨されます。
- シングル AZ (HA): ミリ秒未満のレイテンシーでファイルシステムレベルで高可用性を提供します。高可用性を必要とする最高のパフォーマンスのワークロードに推奨されます。
- マルチ AZ: ファイルシステムレベルおよびアベイラビリティーゾーン間で高可用性を提供します。アベイラビリティーゾーン間で追加の可用性を必要とする高性能ワークロードに推奨されます。

パフォーマンスに関する考慮事項：

- デプロイタイプ： アベイラビリティーゾーン間の追加の可用性が要件でない場合は、シングル AZ (HA) デプロイタイプの使用を検討してください。このデプロイタイプは、書き込みのスループットの最大 100% を提供し、ミリ秒未満のレイテンシーを維持します。Gen2 ファイルシステムには、頻繁にアクセスされるデータを最大 テラバイト保存するための追加の NVMe キャッシュがあります。マルチ AZ ファイルシステムは、書き込みのスループットの最大 75% を提供し、クロス AZ トラフィックに対応するためのレイテンシーが増加します。
- スループットと IOPS: ファイルシステム用に設定されたスループットと IOPS の両方をデプロイ後にスケールアップまたはスケールダウンできます。最大 10GB/秒のディスクスループットをプロビジョニングし、最大 21GB/秒のキャッシュされたデータアクセスを提供できます。IOPS はディスクから最大 400,000 までスケールでき、キャッシュは 100 万 IOPS 以上を提供できます。シングル AZ ファイルシステムのスループットスケーリングでは、高可用性が存在しないため、ファイルシステムが短時間停止することに注意してください。シングル AZ (HA) またはマルチ AZ ファイルシステムのスループットスケーリングは、中断することなく実行できます。SSD IOPS は 6 時間に 1 回スケールできます。
- ストレージクラス： FSx for OpenZFS は、SSD ストレージクラスと Intelligent-Tiering ストレージクラスの両方をサポートしています。AI/ML ワークロードでは、SSD ストレージクラスを使用して、CPU / GPU の をできるだけビジー状態に保つワークロードに一貫したパフォーマンスを提供することをお勧めします。

- 圧縮： 圧縮できるワークロードがある場合は、LZ4 圧縮アルゴリズムを有効にします。これにより、各ファイルがキャッシュで消費するデータの量が減少し、より多くのデータをネットワークスループットとしてキャッシュから直接提供でき、IOPS により SSD ディスクの負荷が軽減されます。
- レコードサイズ： ほとんどの AI/ML ワークロードは、デフォルトの 128KiB のレコードサイズのままにしておくとメリットがあります。この値は、データセットがアプリケーションから 128KiB 未満の一貫した小さなブロックアクセスを持つ大きなファイル (10GiB 128KiB 以上) で構成されている場合にのみ削減する必要があります。

ファイルシステムが作成されると、関連付けられたルートボリュームがサービスによって自動的に作成されます。ファイルシステムのルートボリュームの子ボリューム内にデータを保存するのがベストプラクティスです。FSx for OpenZFS CSI ドライバーを使用して、関連付けられた永続ボリュームクレームを作成し、子ボリュームを動的に作成します。

例:

FSx for OpenZFS ボリュームのストレージクラス (SC) 定義。既存のファイルシステムにルートボリュームの子ボリューム ($ROOT_VOL_ID) を作成し、NFS v4.2 プロトコルを使用してボリュームを VPC CIDR ($VPC_CIDR) にエクスポートするために使用されます。

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: fsxz-vol-sc
provisioner: fsx.openzfs.csi.aws.com
parameters:
  ResourceType: "volume"
  ParentVolumeId: '"$ROOT_VOL_ID"'
  CopyTagsToSnapshots: 'false'
  DataCompressionType: '"LZ4"'
  NfsExports: '[{"ClientConfigurations": [{"Clients": "$VPC_CIDR", "Options":
 ["rw","crossmnt","no_root_squash"]}]}]'
  ReadOnly: 'false'
  RecordSizeKiB: '128'
  Tags: '[{"Key": "Name", "Value": "AI-ML"}]'
  OptionsOnDeletion: '["DELETE_CHILD_VOLUMES_AND_SNAPSHOTS"]'
reclaimPolicy: Delete
allowVolumeExpansion: false
mountOptions:
  - nfsvers=4.2
```

```
  - rsize=1048576
  - wsize=1048576
  - timeo=600
  - nconnect=16
  - async
```

上記の fsxz-vol-sc に対して動的に作成された永続ボリューム要求 (PVC)。割り当てられたストレージ容量は 1Gi です。これは、CSI ドライバーのよくある質問に記載されているように、FSx for OpenZFS ボリュームに必要です。ボリュームには、この設定でファイルシステムにプロビジョニングされたフルキャパシティが提供されます。ボリューム容量を制限する必要がある場合は、ユーザーまたはグループのクォータを使用して制限できます。

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: dynamic-vol-pvc
  namespace: example
spec:
  accessModes:
    - ReadWriteMany
  storageClassName: fsxz-vol-sc
  resources:
    requests:
      storage: 1Gi
```

dynamic-vol-pvc の永続ボリュームクレーム (PVC) を使用してボリュームをマウントするようにポッドを設定します。

```
kind: Pod
apiVersion: v1
metadata:
  name: fsx-app
  namespace: example
spec:
  volumes:
    - name: dynamic-vol-pv
      persistentVolumeClaim:
        claimName: dynamic-vol-pvc
  containers:
    - name: app
      image: amazonlinux:2023
      command: ["/bin/sh"]
```

```
    volumeMounts:
      - mountPath: "/mnt/fsxz"
        name: dynamic-vol-pv
```

共有モデルキャッシュの Amazon EFS

複数の EC2 GPU コンピューティングインスタンス環境があり、複数のノードとアベイラビリティーゾーン間で共有モデルアクセスを必要とする動的なワークロード (Karpenter によるリアルタイムオンライン推論など) があり、パフォーマンスとスケーラビリティのニーズが中程度のシナリオでは、EFS CSI ドライバーを介して Amazon Elastic File System (EFS) ファイルシステムを永続的ボリュームとして使用することをお勧めします。Amazon EFS は、フルマネージドで可用性が高く、スケーラブルなクラウドベースの NFS ファイルシステムです。共有ファイルストレージを持つ EC2 インスタンスとコンテナを一貫したパフォーマンスで実現し、ストレージの事前プロビジョニングは必要ありません。EFS をモデルボリュームとして使用し、EKS クラスターで永続ボリュームを定義して、ボリュームを共有ファイルシステムとしてマウントします。EFS ファイルシステムによってバックアップされる各永続ボリュームクレーム (PVC) は、EFS EFS ファイルシステムへの EFS アクセスポイントとして作成されます。EFS を使用すると、複数のノードとポッドが同じモデルファイルにアクセスできるため、各ノードのファイルシステムにデータを同期する必要がなくなります。EFS CSI ドライバーをインストールしてEFS を EKS と統合します。

Amazon EFS ファイルシステムは、次のスループットモードでデプロイできます。

- バーストスループット: ファイルシステムサイズでスループットをスケーリングし、バーストが時折発生するさまざまなワークロードに適しています。
- プロビジョンドスループット: 専有スループット。予測可能なパフォーマンスニーズが制限内の一貫した ML トレーニングジョブに最適です。
- Elastic Throughput (ML に推奨) : ワークロード、さまざまな ML ワークロードの費用対効果に基づいて自動的にスケーリングします。

パフォーマンス仕様を確認するには、「Amazon EFS パフォーマンス仕様」を参照してください。

パフォーマンスに関する考慮事項 :

- さまざまなワークロードに Elastic スループットを使用します。
- アクティブな ML ワークロードには標準ストレージクラスを使用します。

Amazon EFS ファイルシステムを EKS クラスターとポッド内の永続的ボリュームとして使用する完全な例については、GitHub の EFS CSI ドライバーの例を参照してください。Amazon CloudWatch

を使用して Amazon EFS パフォーマンスメトリクスをモニタリングします。パフォーマンスのボトルネックが特定されたら、必要に応じて設定パラメータを調整します。

レイテンシーの影響を受けやすいオブジェクト指向ワークフローに S3 Express One Zone を使用する

大規模なモデルトレーニング、推論、高性能分析など、Amazon EKS のレイテンシーの影響を受けやすい AI/ML ワークロードの場合は、SS3 Express One Zone を使用して高性能モデルのストレージと取り出しを行うことをお勧めします。S3 Express One Zone は、ファイルシステムなどの階層名前空間を提供します。この名前空間では、ディレクトリバケットにアップロードするだけで、高速を維持しながらすべてを「チャッキングする」のに適しています。これは、オブジェクト指向ワークフローに慣れている場合に特に便利です。または、ファイルシステム (POSIX 準拠など) に慣れている場合は、Amazon FSx for Lustre または OpenZFS を使用することをお勧めします。Amazon S3 Express One Zone は、ディレクトリバケットを使用して単一のアベイラビリティーゾーン (AZ) にデータを保存し、複数の AZ にデータを分散する標準 S3 バケットよりも低いレイテンシーを提供します AZs 。最良の結果を得るには、EKS コンピューティングを Express One Zone バケットと同じ AZ にコロケーションしてください。S3 Express One Zone の違いの詳細については、「ディレクトリバケットの違い」を参照してください。

ファイルシステムのセマンティクスを使用して S3 Express One Zone にアクセスするには、ローカルファイルシステムとして S3 バケット (Express One Zone を含む) をマウントする Mountpoint S3 CSI ドライバーを使用することをお勧めします。 S3 これにより、ファイルオペレーション (オープン、読み取り、書き込みなど) が S3 API コールに変換され、複数のクライアントからの読み取り負荷の高いワークロードや新しいオブジェクトへのシーケンシャル書き込みに最適化された高スループットアクセスが提供されます。サポートされているオペレーションと制限 (完全な POSIX コンプライアンスがないが、Express One Zone でサポートされている名前を追加および変更するなど) の詳細については、Mountpoint セマンティクスドキュメントを参照してください。

パフォーマンス上の利点

- S3 Standard よりも最大 10 倍高速なデータアクセスを提供し、1 桁ミリ秒の一貫したレイテンシーと最大 80% の低いリクエストコストを実現します。
- ディレクトリバケットごとに 1 秒あたり数十万から数百万のリクエストを処理するようにスケールし、極度の負荷 (ネットワークを飽和させる数十から数十万の GPUs/CPUs を持つクラスターなど) 中に標準 S3 で見られるスロットリングやブラウンアウトを回避します。
- セッションベースの認証メカニズムを使用します。一度認証してセッショントークンを取得し、リクエストごとの認証オーバーヘッドなしで高速で繰り返しオペレーションを実行します。これは、頻繁なチェックポイントやデータロードなどのワークロードに最適化されています。

推奨されるユースケース

- キャッシュ: S3 Express One Zone で Mountpoint S3 CSI ドライバーを使用する主なユースケースの 1 つはキャッシュです。最初のインスタンスは S3 Standard (汎用) からデータを読み取り、低レイテンシーの Express One Zone にキャッシュします。他のクライアントによるそれ以降の読み取りはキャッシュされたデータにすばやくアクセスするため、複数の EKS ノードが同じデータ (共有トレーニングデータセットなど) を読み取るマルチノードシナリオに最適です。これにより、繰り返しアクセスする場合にパフォーマンスが最大 7 倍向上し、コンピューティングコストを削減できます。完全な POSIX コンプライアンス (ファイルロックやインプレース変更など) を必要とするワークロードでは、代わりに Amazon FSx for Lustre または OpenZFS を検討してください。
- 大規模な AI/ML トレーニングと推論: 汎用 S3 スロットリングによって遅延が発生し、GPUs など) を持つワークロードに最適です。例えば、LLM の研究者や日常的なモデルテスト/チェックポイントを実行する組織は、リージョン S3 を中断することなく、高速で信頼性の高いアクセスからメリットを得られます。小規模なワークロード (ノード数 10 個など) では、S3 Standard または他のストレージクラスで十分です。
- データパイプライン: モデルのロード/準備、トレーニングデータのアーカイブ、チェックポイントのストリーミングを行います。チームが従来のファイルシステムよりもオブジェクトストレージを優先する場合は (S3 に慣れているためなど)、FSx for Lustre などの POSIX 準拠のオプションを設計変更する代わりに、これを使用します。

考慮事項

- 耐障害性: シングル AZ 設計は、マルチ AZ クラス (99.99% の可用性) と比較して、99.999999999% の耐久性 (AZ 内の冗長性による標準 S3 と同じ) を提供しますが、可用性 (99.95% 設計、99.9% SLA) は低くなります。AZ 障害に対する耐障害性が低くなります。再利用可能なデータまたはキャッシュされたデータには、 を使用します。重要なワークロードには、マルチ AZ レプリケーションまたはバックアップを検討してください。
- API と機能のサポート: S3 APIs のサブセット (ライフサイクルポリシーやレプリケーションなしなど) をサポートします。セッション認証やオブジェクト処理には、アプリケーションのマイナーな変更が必要になる場合があります。
- EKS 統合: ネットワークレイテンシーを最小限に抑えるために、EKS ポッド/ノードをディレクトリバケットと同じ AZ に配置します。Mountpoint for Amazon S3 または CSI ドライバーを Kubernetes ネイティブアクセスに使用します。
- テスト： 非本番環境で取得レイテンシーをテストして、パフォーマンスの向上を検証します。標準 S3 シナリオ (GPU 飽和度が高いなど) のスロットリングをモニタリングし、比較します。

S3 Express One Zone ストレージクラスは複数のリージョンで利用でき、ストレージを待たずにオブジェクトへのアクセスを必要とするワークロードの EKS と統合されます。詳細については、S3 Express One Zone の開始方法」を参照してください。

オブザーバビリティ

モニタリングとオブザーバビリティ

GPU メトリクスの説明

GPU 使用率メトリクスは、GPU がサンプルウィンドウ中に何らかの作業を実行したかどうかを示します。このメトリクスは、GPU が少なくとも 1 つの命令を実行した時間の割合をキャプチャしますが、GPU がハードウェアをどの程度効率的に使用したかは明らかにしません。GPU には、命令を実行する並列処理ユニットである複数のストリーミングマルチプロセッサ (SMs) が含まれています。使用率が 100% の読み取りは、GPU がすべての SMs で大量の並列ワークロードを実行したことを意味するか、1 つの小さな命令がサンプル期間中に GPU をアクティブ化したことを意味する可能性があります。実際の使用率を理解するには、複数のレベルのハードウェアアーキテクチャで GPU メトリクスを調べる必要があります。各ストリーミングマルチプロセッサは異なるコアタイプから構築され、各レイヤーは異なるパフォーマンス特性を公開します。最上位メトリクス (GPU 使用率、メモリ使用率、GPU Power、GPU 温度、nvidia-smi で表示) は、デバイスがアクティブかどうかを示します。メトリクス (SM 使用率、SM アクティビティ、テンソルコア使用率) を深くすると、GPU がリソースをどの程度効率的に使用するかがわかります。

GPU の高電力使用量をターゲットにする

使用率の低い GPUs、ワークロードがすべての GPU コンポーネントを同時にエンゲージできないため、コンピューティング容量を浪費し、コストが増加します。Amazon EKS の AI/ML ワークロードの場合、実際の GPU アクティビティを識別するためのプロキシとして GPU 電力使用量を追跡します。GPU 使用率は、GPU がカーネルを実行する時間の割合を報告しますが、ストリーミングマルチプロセッサ、メモリコントローラー、テンソルコアがすべて同時にアクティブであるかどうかは明らかではありません。電力使用量は、完全にエンゲージされたハードウェアが、軽量カーネルを実行しているハードウェアやタスク間でアイドル状態になっているハードウェアよりも大幅に多くの電力を消費するため、このギャップを公開します。GPU の熱設計能力 (TDP) と消費電力を比較して使用率の低さを特定し、ワークロードが CPU 前処理、ネットワーク I/O、非効率的なバッチサイズによってボトルネックになっているかどうかを調査します。

Amazon EKS で CloudWatch Container Insights を設定して、GPU の電力消費量が低いポッド、ノード、またはワークロードを特定します。このツールは Amazon EKS と直接統合されており、GPU の

電力消費量をモニタリングし、電力使用量がターゲットレベルを下回ったときにポッドスケジューリングまたはインスタンスタイプを調整できます。高度な視覚化またはカスタムダッシュボードが必要な場合は、NVIDIA の DCGM-Exporter with Prometheus と Grafana for Kubernetes ネイティブモニタリングを使用します。どちらのアプローチも`nvidia_smi_power_draw`、(GPU の電力消費量) や `nvidia_smi_temperature_gpu` (GPU 温度) などの主要な NVIDIA メトリクスを表しています。メトリクスのリストについては、https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/CloudWatch-Agent-NVIDIA-GPU.htm を参照してください。特定の時間、または特定のジョブで一貫して低い電力使用量などのパターンを探します。これらの傾向は、ワークロードを統合する場所やリソース割り当てを調整する場所を特定するのに役立ちます。

Kubernetes の静的リソース制限 (CPU、メモリ、GPU 数など) は、特に需要が変動する推論などの動的な AI/ML ワークロードでは、過剰プロビジョニングや使用率の低下につながることがよくあります。使用率の傾向を分析し、ワークロードをより少ない GPUs に統合します。追加の GPU を割り当てる前に、各 GPU がフル使用率に達していることを確認します。このアプローチにより、無駄が減り、コストが削減されます。スケジューリングと共有戦略の最適化に関する詳細なガイダンスについては、「EKS Compute and Autoscaling best practices」を参照してください。

オブザーバビリティとメトリクス

AI/ML ワークロードのモニタリングおよびオブザーバビリティツールの使用

最新の AI/ML サービスでは、インフラストラクチャ、モデリング、アプリケーションロジック間の調整が必要です。プラットフォームエンジニアは、インフラストラクチャとオブザーバビリティスタックを管理します。メトリクスを収集、保存、視覚化します。AI/ML エンジニアは、モデル固有のメトリクスを定義し、さまざまな負荷とデータ分散の下でパフォーマンスをモニタリングします。アプリケーション開発者APIs を消費し、リクエストをルーティングし、サービスレベルのメトリクスとユーザーインタラクションを追跡します。統一されたオブザーバビリティプラクティスがない場合、これらのチームはサイロで作業し、システムのヘルスとパフォーマンスに関する重要なシグナルを見逃します。環境間で可視性を共有することで、すべてのステークホルダーが問題を早期に検出し、信頼性の高いサービスを維持できます。

AI/ML ワークロード用の Amazon EKS クラスターの最適化には、特に GPU メモリ管理に関する固有のモニタリング課題があります。適切なモニタリングを行わないと、組織はout-of-memory (OOM) エラー、リソースの非効率性、不要なコストに直面します。効果的なモニタリングにより、EKS 顧客のパフォーマンス、耐障害性が向上し、コストを削減できます。3 つのモニタリングレイヤーを組み合わせた包括的なアプローチを使用します。まず、NVIDIA DCGM Exporter を使用して詳細な GPU メトリクスをモニタリングし、GPU 電力使用量、GPU 温度、SM アクティビティ、SM 占有率、XID エラーを追跡します。次に、Ray や vLLM などの推論サービスフレームワークをモニタリン

グして、ネイティブメトリクスを通じて分散ワークロードのインサイトを取得します。3 つ目は、アプリケーションレベルのインサイトを収集して、ワークロードに固有のカスタムメトリクスを追跡することです。このレイヤードアプローチにより、ハードウェア使用率からアプリケーションパフォーマンスまでを可視化できます。

ツールとフレームワーク

いくつかのツールとフレームワークは、AI/ML ワークロードをモニタリングするためのネイティブ out-of-the-boxメトリクスを提供します。これらの組み込みメトリクスにより、カスタム計測が不要になり、セットアップ時間が短縮されます。メトリクスは、レイテンシー、スループット、トークン生成などのパフォーマンスの側面に焦点を当てており、推論の提供とベンチマークに不可欠です。ネイティブメトリクスを使用すると、カスタムコレクションパイプラインを構築せずに、すぐにモニタリングを開始できます。

- vLLM: リクエストレイテンシーやメモリ使用量などのネイティブメトリクスを提供する、大規模言語モデル (LLMs用の高スループットサービスエンジン。
- Ray: タスクの実行時間やリソース使用率など、スケーラブルな AI ワークロードのメトリクスを出力する分散コンピューティングフレームワーク。
- Hugging Face Text Generation Inference (TGI): LLMsデプロイして提供するためのツールキット。推論パフォーマンスのメトリクスが組み込まれています。
- NVIDIA genai-perf: 生成 AI モデルをベンチマークし、スループット、レイテンシー、特定の時間間隔で完了したリクエストなどの LLM 固有のメトリクスを測定するためのコマンドラインツールです。

オブザーバビリティメソッド

以下のいずれかの方法で追加のオブザーバビリティメカニズムを実装することをお勧めします。

CloudWatch Container Insights 組織が最小限のセットアップで AWS ネイティブツールを希望する場合は、CloudWatch Container Insights をお勧めします。NVIDIA DCGM Exporter と統合して GPU メトリクスを収集し、インサイトをすばやく得るための構築済みのダッシュボードを提供します。クラスターに CloudWatch Observability アドオンをインストールすることで、Container Insights は NVIDIA DCGM Exporter のライフサイクルをデプロイおよび管理し、Nvidia のドライバーから GPU メトリクスを収集して CloudWatch に公開します。

Container Insights をインストールすると、CloudWatch は環境内の NVIDIA GPUs を自動的に検出し、重要なヘルスとパフォーマンスのメトリクスを収集します。これらのメトリクスは、厳選された out-of-the-boxダッシュボードに表示されます。Unified CloudWatch エージェントを使用して Ray と

vLLM を CloudWatch と統合し、ネイティブメトリクスを送信することもできます。 CloudWatch この統合アプローチにより、EKS 環境でのオブザーバビリティが簡素化され、チームはモニタリングインフラストラクチャを構築するのではなく、パフォーマンスの調整とコストの最適化に集中できます。

使用可能なメトリクスの完全なリストについては、「Amazon EKS および Kubernetes Container Insights メトリクス」を参照してください。GPU モニタリングの実装に関するstep-by-stepガイダンスについては、Amazon CloudWatch Container Insights を使用した NVIDIA GPU ワークロードの運用上のインサイトの取得」を参照してください。推論レイテンシーを最適化する実用的な例については、「AI 応答性の最適化: Amazon Bedrock レイテンシー最適化推論の実用的なガイド」を参照してください。

Managed Prometheus と Grafana カスタマイズされたダッシュボードと高度な可視化機能が必要な場合は、NVIDIA DCGM-Exporter と Grafana for Kubernetes ネイティブモニタリングを使用して Prometheus をデプロイします。Prometheus は DCGM-Exporter から GPU メトリクスをスクレイピングして保存し、Grafana は柔軟な視覚化とアラート機能を提供します。このアプローチにより、CloudWatch Container Insights と比較して、ダッシュボードの設計とメトリクスの保持をより細かく制御できます。

Ray、vLLM Ray、vLLM などのオープンソースフレームワークを統合しhttps://awslabs.github.io/ai-on-eks/docs/blueprints/inference/GPUs/vLLM-rayserveてネイティブメトリクスを Prometheus にエクスポートすることで、このモニタリングスタックを拡張できます。Grafana を AWS X-Ray データソースに接続して分散トレースを視覚化し、推論パイプライン全体のパフォーマンスのボトルネックを特定することもできます。この組み合わせにより、アプリケーションレベルのリクエストフローを通じて GPU レベルのメトリクスからend-to-endの可視性が得られます。

このモニタリングスタックのデプロイに関するstep-by-stepガイダンスについては、「AWS マネージドオープンソースサービスを使用した Amazon EKS での GPU ワークロードのモニタリング」を参照してください。

コアトレーニングとファインチューニングメトリクスのモニタリングを検討する

コアトレーニングメトリクスをモニタリングして、Amazon EKS クラスターとそのクラスターで実行されている機械学習ワークロードの状態とパフォーマンスを追跡します。トレーニングワークロードは、長期間実行され、リソースの消費が異なり、モデルの収束とデータパイプラインの効率を可視化する必要があるため、推論ワークロードとは異なるモニタリング要件があります。以下のメトリクスは、ボトルネックを特定し、リソース割り当てを最適化し、トレーニングジョブを正常に完了させるのに役立ちます。このモニタリングアプローチを実装するためのstep-by-stepガイダンスについては、「Amazon EKS での機械学習ワークロードのモニタリングの概要」を参照してください。

リソース使用状況メトリクス

リソース使用状況メトリクスをモニタリングして、リソースが適切に消費されていることを確認します。これらのメトリクスは、ボトルネックと根本原因のパフォーマンス問題を特定するのに役立ちます。

- CPU、メモリ、ネットワーク、GPU 電力、GPU 温度 - これらのメトリクスをモニタリングして、割り当てられたリソースがワークロードの需要を満たし、最適化の機会を特定します。gpu_memory_usage_bytes などのメトリクスを追跡してメモリ消費パターンを特定し、ピーク使用量を検出します。95 パーセンタイル (P95) などのパーセンタイルを計算して、トレーニング中の最も高いメモリ需要を把握します。この分析は、モデルとインフラストラクチャを最適化して OOM エラーを回避し、コストを削減するのに役立ちます。
- SM 占有率、SM アクティビティ、FPxx アクティビティ - これらのメトリクスをモニタリングして、GPU の基盤となるリソースがどのように使用されているかを理解します。経験則としての SM アクティビティのターゲット 0.8。
- ノードとポッドのリソース使用率 - ノードとポッドレベルでリソースの使用状況を追跡して、リソースの競合と潜在的なボトルネックを特定します。ノードが容量制限に近づいているかどうかをモニタリングします。これにより、ポッドスケジューリングが遅延し、トレーニングジョブが遅くなる可能性があります。
- リソース使用率とリクエストおよび制限の比較 — 実際のリソース使用率を設定済みのリクエストおよび制限と比較して、クラスターが現在のワークロードを処理し、将来のワークロードに対応できるかどうかを判断します。この比較により、OOM エラーやリソースの無駄を避けるためにリソース割り当てを調整する必要があるかどうかがわかります。
- ML フレームワークの内部メトリクス - TensorFlow や PyTorch などの ML フレームワークから内部トレーニングと収束メトリクスを取得します。これらのメトリクスには、損失曲線、学習率、バッチ処理時間、トレーニングステップ期間が含まれます。TensorBoard または同様のツールを使用してこれらのメトリクスを視覚化し、モデルの収束を追跡し、トレーニングの非効率性を特定します。

モデルパフォーマンスメトリクス

モデルパフォーマンスメトリクスをモニタリングして、トレーニングプロセスが精度とビジネス要件を満たすモデルを生成することを確認します。これらのメトリクスは、トレーニングを停止するタイミングの決定、モデルバージョンの比較、パフォーマンスの低下の特定に役立ちます。

- 精度、精度、再現率、F1-score — これらのメトリクスを追跡して、モデルが検証データに対してどの程度うまく機能するかを理解します。各トレーニングエポックの後に検証セットの F1-score

を計算して、モデルが改善しているかどうか、および許容可能なパフォーマンスレベルに達したタイミングを評価します。

- ビジネス固有のメトリクスと KPIs — AI/ML イニシアチブのビジネス価値を直接測定するメトリクスを定義して追跡します。レコメンデーションシステムでは、クリックスルー率や変換率などのメトリクスを追跡して、モデルが意図したビジネス成果を促進するようにします。
- 経時的なパフォーマンス — モデルバージョンとトレーニング実行のパフォーマンスメトリクスを比較して、傾向を特定し、低下を検出します。新しいモデルバージョンが、ベースラインモデルと比較してパフォーマンスを維持または改善するかどうかを追跡します。この履歴比較は、新しいモデルをデプロイするか、トレーニングの問題を調査するかを決定するのに役立ちます。

データ品質とドリフトのメトリクス

データ品質とドリフトメトリクスをモニタリングして、トレーニングデータの一貫性と代表性を維持します。データドリフトにより、時間の経過とともにモデルのパフォーマンスが低下する可能性がありますが、データ品質の問題により、モデルの収束や信頼性の低い結果の生成が妨げられる可能性があります。

- 入力データの統計プロパティ — 入力特徴の平均、標準偏差、分布などの統計プロパティを経時的に追跡して、データのドリフトや異常を検出します。特徴分布がベーストレーニングデータから大幅にシフトするかどうかをモニタリングします。例えば、重要な機能の平均が 2 標準偏差以上変化した場合、データパイプラインが変更されたかどうか、または基盤となるデータソースがシフトしたかどうかを調査します。
- データドリフトの検出とアラート — トレーニングに影響を与える前に、データ品質の問題を検出して警告する自動メカニズムを実装します。Kolmogorov-Smirnov テストやカイ二乗テストなどの統計テストを使用して、現在のデータ分布を元のトレーニングデータと比較します。テストが重大なドリフトを検出したときにアラートを設定して、更新されたデータでモデルを再トレーニングしたり、データパイプラインの問題を調査したりできるようにします。

レイテンシーとスループットのメトリクス

レイテンシーとスループットのメトリクスをモニタリングして、トレーニングパイプラインのボトルネックを特定し、リソース使用率を最適化します。これらのメトリクスは、トレーニング中に費やす時間と最適化作業の焦点を理解するのに役立ちます。

- ML トレーニングパイプラインのEnd-to-Endのレイテンシー — データ取り込みからモデル更新まで、トレーニングパイプライン全体を通過するデータの合計時間を測定します。トレーニング実行全体でこのメトリクスを追跡して、パイプラインの変更によってパフォーマンスが向上するか低下

するかを特定します。レイテンシーが高い場合は、多くの場合、データのロード、前処理、ノード間のネットワーク通信のボトルネックを示します。

- トレーニングスループットと処理レート — トレーニングパイプラインが時間単位ごとに処理するデータの量を追跡して、効率的なリソース使用率を確保します。1 秒あたりに処理されたサンプルや 1 分あたりに完了したバッチなどのメトリクスをモニタリングします。ハードウェア容量に対するスループットが低い場合は、GPU サイクルを浪費するデータロード、前処理、またはモデル計算の非効率性が示されます。
- チェックポイントの保存と復元のレイテンシー – モデルチェックポイントをストレージ (S3、EFS、FSx) に保存し、ジョブを再開したり、障害から回復したりするときに GPU または CPU メモリに復元するために必要な時間をモニタリングします。チェックポイント操作が遅いと、ジョブの復旧時間が長くなり、コストが増加します。チェックポイントのサイズ、保存期間、復元期間、障害数を追跡して、圧縮やストレージ階層の高速化などの最適化の機会を特定します。
- データのロードと前処理時間 - ストレージからのデータのロードと前処理変換の適用にかかった時間を測定します。この時間をモデル計算時間と比較して、トレーニングがデータバインドかコンピューティングバインドかを判断します。データロードが総トレーニング時間の 20% 以上を消費する場合は、キャッシュ、プリフェッチ、または高速ストレージを使用してデータパイプラインを最適化することを検討してください。

エラー率と失敗

トレーニングパイプライン全体でエラー率と障害をモニタリングして、信頼性を維持し、無駄なコンピューティングリソースを防止します。検出されないエラーにより、トレーニングジョブがサイレントに失敗したり、無効なモデルを生成したり、問題に気付くまでに GPU 時間を浪費したりする可能性があります。

- パイプラインエラーモニタリング — データの前処理、モデルトレーニング、チェックポイントオペレーションなど、ML パイプラインのすべてのステージでエラーを追跡します。エラータイプ、頻度、影響を受けるコンポーネントをログに記録して、問題をすばやく特定します。一般的なエラーには、データ形式の不一致、前処理中のout-of-memory障害、ストレージ制限によるチェックポイント保存障害などがあります。エラー率がベースラインしきい値を超えたときにアラートを設定して、エラーがカスケードされる前に調査できるようにします。
- 定期的なエラー分析 — 定期的なエラーのパターンを特定して調査し、将来の障害を防ぎ、パイプラインの信頼性を向上させます。ログを分析して、特定のデータサンプル、バッチサイズ、またはトレーニング設定が一貫して障害を引き起こすかどうかを確認します。たとえば、特定の入力データ型が前処理エラーをトリガーする場合は、パイプラインの早い段階で検証チェックを追加する

か、データクリーニングロジックを更新します。平均障害間隔 (MTBF) を追跡して、パイプラインの信頼性が時間の経過とともに向上するかどうかを測定します。

Kubernetes および EKS 固有のメトリクス

Kubernetes および EKS メトリクスをモニタリングして、クラスターインフラストラクチャが正常であり、トレーニングワークロードをサポートできることを確認します。これらのメトリクスは、トレーニングジョブの失敗やパフォーマンスの低下を引き起こす前に、インフラストラクチャの問題を検出するのに役立ちます。

- Kubernetes クラスター状態メトリクス — ポッド、ノード、デプロイ、サービスなどの Kubernetes オブジェクトのヘルスとステータスをモニタリングします。ポッドのステータスを追跡して、保留中、失敗、またはクラッシュループ状態でスタックしているポッドを特定します。ノードの状態をモニタリングして、ディスク負荷、メモリ負荷、ネットワーク利用不能などの問題を検出します。kubectl またはモニタリングツールを使用してこれらのメトリクスを継続的にチェックし、ポッドの起動に失敗したり、ノードがスケジュールできなくなったりした場合にアラートを設定します。
- Training Pipeline Execution Metrics — パイプラインの実行の成功と失敗、ジョブ期間、ステップ完了時間、オーケストレーションエラーを追跡します。トレーニングジョブが想定期間内に完了したかどうか、および失敗率が時間の経過とともに増加するかどうかをモニタリングします。ジョブの成功率、平均ジョブ期間、失敗までの時間などのメトリクスを追跡します。これらのメトリクスは、インフラストラクチャの問題、設定の問題、またはデータ品質の問題がトレーニングの失敗を引き起こすかどうかを特定するのに役立ちます。
- AWS サービスメトリクス — EKS インフラストラクチャとトレーニングワークロードをサポートする AWS サービスのメトリクスを追跡します。リクエストのレイテンシー、エラー率、スループットなどの S3 メトリクスをモニタリングして、データのロードパフォーマンスが一貫していることを確認します。IOPS、スループット、キューの長さなどの EBS ボリュームメトリクスを追跡して、ストレージのボトルネックを検出します。VPC フローログとネットワークメトリクスをモニタリングして、ノード間または外部サービスへの接続の問題を特定します。
- Kubernetes コントロールプレーンメトリクス — API サーバー、スケジューラ、コントローラーマネージャー、および etcd データベースをモニタリングして、クラスターオペレーションに影響するパフォーマンスの問題や障害を検出します。API サーバーのリクエストレイテンシー、リクエストレート、エラー率を追跡して、コントロールプレーンがリクエストのスケジュールに迅速に応答できるようにします。etcd データベースのサイズ、コミット期間、リーダーの変更をモニタリングして、安定性の問題を検出します。API サーバーのレイテンシーが高い、または頻繁に etcd

リーダーを変更すると、ポッドのスケジュールが遅延し、トレーニングジョブの起動時間が長くなる可能性があります。

アプリケーションログとインスタンスログ

アプリケーションログとインスタンスログを収集して分析し、メトリクスだけでは説明できない問題を診断します。ログは、エラー、状態変更、システムイベントに関する詳細なコンテキストを提供し、トレーニングジョブが失敗したり、パフォーマンスが低下したりする理由を理解するのに役立ちます。ログとメトリクスを関連付けることで、根本原因をより迅速に特定できます。

- アプリケーションログ - トレーニングジョブ、データパイプライン、ML フレームワークからアプリケーションログを収集して、ボトルネックを特定し、障害を診断します。これらのログには、データのロードエラー、モデル初期化の失敗、チェックポイントの保存エラー、フレームワーク固有の警告など、ジョブの実行に関する詳細情報が記録されます。ログタイムスタンプをメトリクスのスパイクと関連付けて、パフォーマンスの低下や障害の原因を理解します。例えば、GPU 使用率が突然低下した場合、データパイプラインの停止や前処理の失敗を示すエラーがないか、アプリケーションログを確認します。CloudWatch Logs や Fluent Bit などの一元化されたログ記録ツールを使用して、すべてのポッドからログを集約し、検索できるようにします。
- インスタンスログ - システムジャーナルログや dmesg 出力などのインスタンスレベルのログを収集して、ハードウェアの問題やカーネルレベルの問題を検出します。これらのログには、GPU ドライバーのエラー、メモリ割り当ての失敗、ディスク I/O エラー、アプリケーションログに表示されない可能性のあるネットワークインターフェイスの問題が表示されます。インスタンスログをアプリケーションログとメトリクスと関連付けて、トレーニングの失敗がハードウェアの問題またはアプリケーションの問題に起因するかどうかを判断します。例えば、トレーニングジョブがout-of-memoryエラーで失敗した場合、システムがメモリ不足になったか、アプリケーションがコンテナ制限を超えたかどうかを示すカーネル OOM キラーメッセージの dmesg ログを確認します。GPU XID エラーやディスク障害などの重大なハードウェアエラーのアラートを設定して、障害が発生したインスタンスを、トレーニングが広範囲に中断される前に置き換えることができます。

以下のセクションでは、CloudWatch Container Insights と Amazon Managed Grafana を使用した Amazon Managed Prometheus の 2 つの AWS 推奨アプローチを使用して、上記のメトリクスを収集する方法を示します。最小限のセットアップと事前構築されたダッシュボードで AWS ネイティブツールを使用する場合は、CloudWatch Container Insights を選択します。カスタマイズされたダッシュボード、高度な視覚化機能が必要な場合、または既存の Prometheus ベースのモニタリングインフラストラクチャと統合する場合は、Amazon Managed Grafana で Amazon Managed Prometheus

を選択します。使用可能な Container Insights メトリクスの完全なリストについては、「Amazon EKS および Kubernetes Container Insights メトリクス」を参照してください。

リアルタイムオンライン推論メトリクスのモニタリングを検討する

リアルタイムシステムでは、ユーザーやその他の依存システムにタイムリーな応答を提供するには、低レイテンシーが不可欠です。レイテンシーが高いと、ユーザーエクスペリエンスが低下したり、パフォーマンス要件に違反する可能性があります。推論レイテンシーに影響するコンポーネントには、モデルのロード時間、前処理時間、実際の予測時間、後処理時間、ネットワーク送信時間などがあります。推論レイテンシーをモニタリングして、サービスレベルアグリーメント (SLAs) を満たす低レイテンシーのレスポンスを確保し、以下のカスタムメトリクスを開発することをお勧めします。予想される負荷下でテストし、ネットワークレイテンシーを含め、同時リクエストを考慮し、さまざまなバッチサイズでテストします。

- Time to First Token (TTFT) — ユーザーがリクエストを送信してからレスポンスの開始 (最初の単語、トークン、またはチャンク) を受け取るまでの時間。たとえば、チャットボットでは、ユーザーが質問をした後、最初の出力 (トークン) の生成にかかる時間を確認します。
- End-to-Endのレイテンシー — リクエストが受信されてからレスポンスが返送されるまでの合計時間です。たとえば、リクエストからレスポンスまでの時間を測定します。
- Output Tokens Per Second (TPS) — モデルが応答を開始した後に新しいトークンを生成する速度を示します。たとえば、チャットボットでは、ベースラインテキストの言語モデルの生成速度を追跡します。
- エラー率 — 失敗したリクエストを追跡します。これはパフォーマンスの問題を示している可能性があります。たとえば、大きなドキュメントや特定の文字の失敗したリクエストをモニタリングします。
- スループット — システムが時間単位ごとに処理できるリクエストまたはオペレーションの数を測定します。たとえば、ピーク負荷を処理するために 1 秒あたりのリクエストを追跡します。

K/V (キー/値) キャッシュは、推論レイテンシーの強力な最適化手法であり、特にトランスフォーマーベースのモデルに関連します。K/V キャッシュは、以前のトランスフォーマーレイヤー計算のキーテンソルと値テンソルを保存し、特に大規模言語モデル (LLMs。キャッシュ効率メトリクス (特に K/V またはセッションキャッシュの使用):

- キャッシュヒット/ミス率 — キャッシュを利用する推論設定 (K/V または埋め込みキャッシュ) の場合、キャッシュが役立つ頻度を測定します。ヒット率が低い場合は、最適ではないキャッシュ設定やワークロードの変更を示している可能性があり、どちらもレイテンシーが増加する可能性があります。

以降のトピックでは、上記のいくつかのメトリクスのデータを収集する方法について説明します。AWS が推奨する 2 つのアプローチの例を示します。AWS ネイティブの CloudWatch Container Insights と Amazon Managed Grafana を使用したオープンソースの Amazon Managed Prometheus です。 https://docs.aws.amazon.com/grafana/latest/userguide/getting-started-with-AMG.html全体的なオブザーバビリティのニーズに基づいて、これらのソリューションのいずれかを選択します。Container Insights メトリクスの完全なリストについては、「Amazon EKS および Kubernetes Container Insights メトリクス」を参照してください。

GPU メモリ使用量の追跡

the section called “コアトレーニングとファインチューニングメトリクスのモニタリングを検討する” トピックで説明したように、メモリ不足 (OOM) エラーを防ぎout-of-memoryの使用が不可欠です。次の例は、トレーニングアプリケーションを計測してカスタムヒストグラムメトリクス を公開しgpu_memory_usage_bytes、P95 メモリ使用量を計算してピーク消費量を特定する方法を示しています。ステージング環境でサンプルトレーニングジョブ (トランスフォーマーモデルの微調整など) を使用してテストしてください。

AWS ネイティブ CloudWatch Container Insights の例

このサンプルでは、AWS ネイティブアプローチを使用してトレーニングアプリケーションを計測してヒストグラムgpu_memory_usage_bytesとして公開する方法を示します。AI/ML コンテナは、CloudWatch Embedded Metrics Format (EMF) 形式で構造化ログを出力するように設定する必要があります。CloudWatch ログは EMF を解析し、メトリクスを公開します。トレーニングアプリケーションで aws_embedded_metrics を使用して、構造化ログを EMF 形式で CloudWatch Logs に送信します。CloudWatch Logs は GPU メトリクスを抽出します。

```
from aws_embedded_metrics import metric_scope
import torch
import numpy as np

memory_usage = []

@metric_scope
def log_gpu_memory(metrics):
    # Record current GPU memory usage
    mem = torch.cuda.memory_allocated()
    memory_usage.append(mem)

    # Log as histogram metric
    metrics.set_namespace("MLTraining/GPUMemory")
    metrics.put_metric("gpu_memory_usage_bytes", mem, "Bytes", "Histogram")
```

```
    # Calculate and log P95 if we have enough data points
    if len(memory_usage) >= 10:
        p95 = np.percentile(memory_usage, 95)
        metrics.put_metric("gpu_memory_p95_bytes", p95, "Bytes")
        print(f"Current memory: {mem} bytes, P95: {p95} bytes")

# Example usage in training loop
for epoch in range(20):
    # Your model training code would go here
    log_gpu_memory()
```

Prometheus と Grafana の例

このサンプルでは、Python の Prometheus クライアントライブラリを使用して、トレーニングアプリケーションを計測してヒストグラムgpu_memory_usage_bytes`として公開する方法を示します。

```
from prometheus_client import Histogram
from prometheus_client import start_http_server
import pynvml

start_http_server(8080)
memory_usage = Histogram(
    'gpu_memory_usage_bytes',
    'GPU memory usage during training',
    ['gpu_index'],
    buckets=[1e9, 2e9, 4e9, 8e9, 16e9, 32e9]
)

# Function to get GPU memory usage
def get_gpu_memory_usage():
    if torch.cuda.is_available():
        # Get the current GPU device
        device = torch.cuda.current_device()

        # Get memory usage in bytes
        memory_allocated = torch.cuda.memory_allocated(device)
        memory_reserved = torch.cuda.memory_reserved(device)

        # Total memory usage (allocated + reserved)
        total_memory = memory_allocated + memory_reserved
```

```
        return device, total_memory
    else:
        return None, 0

# Get GPU memory usage
gpu_index, memory_used = get_gpu_memory_usage()
```

リアルタイムオンライン推論の推論リクエスト期間を追跡する

the section called “コアトレーニングとファインチューニングメトリクスのモニタリングを検討する”トピックで説明したように、ユーザーやその他の依存システムにタイムリーな応答を提供するには、低レイテンシーが不可欠です。次の例は、推論アプリケーションによって公開されたカスタムヒストグラムメトリクス inference_request_duration_secondsを追跡する方法を示しています。95 パーセンタイル (P95) レイテンシーを計算して、最悪のシナリオに焦点を当て、ステージング環境で合成推論リクエスト (Locust 経由など) でテストし、アラートしきい値 (>500ms など) を設定して SLA 違反を検出します。

AWS ネイティブ CloudWatch Container Insights の例

このサンプルでは、AWS CloudWatch Embedded Metric Format を使用して、推論アプリケーションで inference_request_duration_seconds のカスタムヒストグラムメトリクスを作成する方法を示します。

```
import boto3
import time
from aws_embedded_metrics import metric_scope, MetricsLogger

cloudwatch = boto3.client('cloudwatch')

@metric_scope
def log_inference_duration(metrics: MetricsLogger, duration: float):
    metrics.set_namespace("ML/Inference")
    metrics.put_metric("inference_request_duration_seconds", duration, "Seconds",
 "Histogram")
    metrics.set_property("Buckets", [0.1, 0.5, 1, 2, 5])

@metric_scope
def process_inference_request(metrics: MetricsLogger):
    start_time = time.time()

    # Your inference processing code here
    # For example:
```

```
    # result = model.predict(input_data)

    duration = time.time() - start_time
    log_inference_duration(metrics, duration)

    print(f"Inference request processed in {duration} seconds")

# Example usage
process_inference_request()
```

Prometheus と Grafana の例

このサンプルでは、Python の Prometheus クライアントライブラリを使用して、推論アプリケーションで inference_request_duration_seconds のカスタムヒストグラムメトリクスを作成する方法を示します。

```
from prometheus_client import Histogram
from prometheus_client import start_http_server
import time

start_http_server(8080)
request_duration = Histogram(
    'inference_request_duration_seconds',
    'Inference request latency',
    buckets=[0.1, 0.5, 1, 2, 5]
)
start_time = time.time()
# Process inference request
request_duration.observe(time.time() - start_time)
```

Grafana で、 クエリを使用して P95 レイテンシーの傾向`histogram_quantile(0.95, sum(rate(inference_request_duration_seconds_bucket[5m])) by (le, pod))`を視覚化します。詳細については、「Prometheus ヒストグラムドキュメント」および「Prometheus クライアントドキュメント」を参照してください。

リアルタイムオンライン推論のトークンスループットを追跡する

the section called “コアトレーニングとファインチューニングメトリクスのモニタリングを検討する” トピックで説明したように、トークンの処理時間をモニタリングしてモデルのパフォーマンスを測定し、スケーリングの決定を最適化することをお勧めします。次の例は、推論アプリケーションによって公開されたカスタムヒストグラムメトリクス `token_processing_duration_seconds`を

追跡する方法を示しています。95 パーセンタイル (P95) の期間を計算して処理効率を分析し、非本番稼働用クラスターでシミュレートされたリクエストロード (100 ~ 1000 リクエスト/秒など) でテストし、KEDA トリガーを調整してスケーリングを最適化します。

AWS ネイティブ CloudWatch Container Insights の例

このサンプルでは、AWS CloudWatch Embedded Metric Format を使用して token_processing_duration_seconds の推論アプリケーションでカスタムヒストグラムメトリクスを作成する方法を示します。ディメンション (「set_dimension）with a custom `get_duration_bucket 関数」を使用して、期間をバケット (「⇐0.01」、「>1」など) に分類します。

```
import boto3
import time
from aws_embedded_metrics import metric_scope, MetricsLogger

cloudwatch = boto3.client('cloudwatch')

@metric_scope
def log_token_processing(metrics: MetricsLogger, duration: float, token_count: int):
    metrics.set_namespace("ML/TokenProcessing")
    metrics.put_metric("token_processing_duration_seconds", duration, "Seconds")
    metrics.set_dimension("ProcessingBucket", get_duration_bucket(duration))
    metrics.set_property("TokenCount", token_count)

def get_duration_bucket(duration):
    buckets = [0.01, 0.05, 0.1, 0.5, 1]
    for bucket in buckets:
        if duration <= bucket:
            return f"<={bucket}"
    return f">{buckets[-1]}"

@metric_scope
def process_tokens(input_text: str, model, tokenizer, metrics: MetricsLogger):
    tokens = tokenizer.encode(input_text)
    token_count = len(tokens)

    start_time = time.time()
    # Process tokens (replace with your actual processing logic)
    output = model(tokens)
    duration = time.time() - start_time

    log_token_processing(metrics, duration, token_count)
```

```
    print(f"Processed {token_count} tokens in {duration} seconds")
    return output
```

Prometheus と Grafana の例

このサンプルでは、Python の Prometheus クライアントライブラリを使用して token_processing_duration_seconds の推論アプリケーションでカスタムヒストグラムメトリクスを作成する方法を示します。

```
from prometheus_client import Histogram
from prometheus_client import start_http_server
import time

start_http_server(8080)
token_duration = Histogram(
    'token_processing_duration_seconds',
    'Token processing time per request',
    buckets=[0.01, 0.05, 0.1, 0.5, 1]
)
start_time = time.time()
# Process tokens
token_duration.observe(time.time() - start_time)
```

Grafana で、 クエリを使用して P95 処理時間の傾向`histogram_quantile(0.95, sum(rate(token_processing_duration_seconds_bucket[5m])) by (le, pod))`を視覚化します。詳細については、「Prometheus ヒストグラムドキュメント」および「Prometheus クライアントドキュメント」を参照してください。

チェックポイントの復元レイテンシーの測定

the section called “コアトレーニングとファインチューニングメトリクスのモニタリングを検討する” トピックで説明したように、チェックポイントレイテンシーはモデルライフサイクルの複数のフェーズにおける重要なメトリクスです。次の例は、アプリケーションによって公開されたカスタムヒストグラムメトリクス `checkpoint_restore_duration_seconds`を追跡する方法を示しています。95 パーセンタイル (P95) の期間を計算して復元パフォーマンスをモニタリングし、非本番クラスターでスポット中断をテストし、アラートしきい値 (<30 秒など) を設定して遅延を検出します。

AWS ネイティブ CloudWatch Container Insights の例

このサンプルは、CloudWatch Insights を使用して checkpoint_restore_duration_seconds をヒストグラムとして公開するようにバッチアプリケーションを計測する方法を示しています。

```
import boto3
import time
import torch
from aws_embedded_metrics import metric_scope, MetricsLogger

@metric_scope
def log_checkpoint_restore(metrics: MetricsLogger, duration: float):
    metrics.set_namespace("ML/ModelOperations")
    metrics.put_metric("checkpoint_restore_duration_seconds", duration, "Seconds",
 "Histogram")
    metrics.set_property("Buckets", [1, 5, 10, 30, 60])
    metrics.set_property("CheckpointSource", "s3://my-bucket/checkpoint.pt")

@metric_scope
def load_checkpoint(model, checkpoint_path: str, metrics: MetricsLogger):
    start_time = time.time()

    # Load model checkpoint
    model.load_state_dict(torch.load(checkpoint_path))

    duration = time.time() - start_time
    log_checkpoint_restore(metrics, duration)

    print(f"Checkpoint restored in {duration} seconds")
```

Prometheus と Grafana の例

このサンプルは、Python の Prometheus クライアントライブラリを使用して、バッチアプリケーションを計測してヒストグラム`checkpoint_restore_duration_seconds`として公開する方法を示しています。

```
from prometheus_client import Histogram
from prometheus_client import start_http_server
import torch

start_http_server(8080)
restore_duration = Histogram(
    'checkpoint_restore_duration_seconds',
    'Time to restore checkpoint',
    buckets=[1, 5, 10, 30, 60]
)
with restore_duration.time():
```

```
    model.load_state_dict(torch.load("s3://my-bucket/checkpoint.pt"))
```

Grafana で、 クエリを使用して P95 復元レイテンシーの傾向`histogram_quantile(0.95, sum(rate(checkpoint_restore_duration_seconds_bucket[5m]) by (le))`を視覚化します。詳細については、「Prometheus ヒストグラムドキュメント」および「Prometheus クライアントドキュメント」を参照してください。

アプリケーションのスケーリングとパフォーマンス

ML アーティファクト、サービングフレームワーク、スタートアップ最適化の管理

Amazon EKS に機械学習 (ML) モデルをデプロイするには、モデルをコンテナイメージとランタイム環境に統合する方法を慎重に検討する必要があります。これにより、スケーラビリティ、再現性、効率的なリソース使用率が確保されます。このトピックでは、ML モデルアーティファクトの処理、フレームワークの提供の選択、事前キャッシュなどの手法によるコンテナの起動時間の最適化に関するさまざまなアプローチについて説明します。これらはすべて、コンテナの起動時間を短縮するように調整されています。

デプロイでの ML モデルアーティファクトの処理

重要な決定は、ML モデルアーティファクト (重みや設定など) を自分で処理する方法です。この選択は、イメージサイズ、デプロイ速度、モデル更新頻度、運用オーバーヘッドに影響します。「モデル」の保存を参照するときは、モデルアーティファクト (トレーニングされたパラメータやモデルの重みなど) を参照することに注意してください。Amazon EKS で ML モデルアーティファクトを処理するには、さまざまなアプローチがあります。各 にはトレードオフがあり、最適なものはモデルのサイズ、更新頻度、インフラストラクチャのニーズによって異なります。最小から最も推奨されるアプローチを次に示します。

- モデルをコンテナイメージにベーキングする: イメージ構築中にモデルファイル (.safetensors、.pth、.h5 など) をコンテナイメージ (Dockerfile など) にコピーします。モデルはイミュータブルイメージの一部です。更新頻度の低い小規模なモデルには、このアプローチを使用することをお勧めします。これにより、一貫性と再現性が確保され、ロードの遅延がなくなり、依存関係の管理が簡素化されますが、イメージサイズが大きくなり、ビルドとプッシュが遅くなり、モデルの更新に再構築と再デプロイが必要になるため、レジストリプルスループットのため、大規模なモデルには適していません。

- 実行時にモデルをダウンロードする: アプリケーションは、init コンテナまたはエントリポイントのスクリプトを介して、外部ストレージ (S3 CSI ドライバー用の Mountpoint、Amazon S3 S3 など) からモデルをダウンロードします。 S3 更新が頻繁な大規模モデルでは、このアプローチから始めることをお勧めします。これにより、コンテナイメージはコード/ランタイムに集中し、再構築なしでモデルを簡単に更新でき、ストレージメタデータを介したバージョニングがサポートされますが、潜在的なネットワーク障害 (再試行ロジックが必要) が発生し、認証とキャッシュが必要になります。

詳細については、「AI on EKS Workshop」の「Accelerating pull process」を参照してください。

ML モデルの提供

Amazon EKS で機械学習 (ML) モデルをデプロイして提供するには、レイテンシー、スループット、スケーラビリティ、運用の簡素化を最適化するための適切なモデル提供アプローチを選択する必要があります。選択は、モデルタイプ (言語、ビジョンモデルなど) 、ワークロードの需要 (リアルタイム推論など) 、チームの専門知識によって異なります。一般的なアプローチには、プロトタイプ作成用の Python ベースのセットアップ、本番稼働用機能専用のモデルサーバー、高パフォーマンスと効率のための特殊な推論エンジンなどがあります。各方法には、セットアップの複雑さ、パフォーマンス、リソース使用率のトレードオフが含まれます。フレームワークを提供すると、依存関係が原因でコンテナイメージサイズ (複数 GBs) が増加し、起動時間に影響する可能性があることに注意してください。これを軽減するには、アーティファクト処理手法を使用したデカップリングを検討してください。オプションは、最も推奨されていないものから最も推奨されていないものまで一覧表示されます。

Python フレームワークを使用する (FastAPI、HuggingFace Transformers with PyTorch など) Python フレームワークを使用してカスタムアプリケーションを開発し、モデルファイル (重み、設定、トークナイザー) をコンテナ化されたノード設定に埋め込みます。

- 長所: 簡単なプロトタイプ、追加のインフラストラクチャのない Python 専用、すべての HuggingFace モデルと互換性がある、簡単な Kubernetes デプロイ。
- 短所: 単一のリクエスト/簡易バッチ処理、遅いトークン生成 (最適化されたカーネルなし) 、メモリ非効率、スケーリング/モニタリングの欠如に制限され、起動時間が長くなります。
- 推奨事項: カスタムロジック統合を必要とする初期プロトタイピングまたは単一ノードタスクに使用します。

専用モデル提供フレームワーク (TensorRT-LLM、TGI など) の使用 ML 推論に TensorRT-LLM や TGI などの専用サーバーを採用し、モデルのロード、ルーティング、最適化を管理します。これらの

は、セーフテンソルなどの形式と、オプションのコンパイルまたはプラグインをサポートしています。

- 長所: バッチ処理 (静的/進行中または継続的) 、量子化 (INT8、FP8、GPTQ)、ハードウェア最適化 (NVIDIA、AMD、Intel、Inferentia)、マルチ GPU サポート (Tensor/Pipeline Parallelism) を提供します。TensorRT-LLM は多様なモデル (LLMs、エンコーダーデコーダー) をサポートし、TGI は HuggingFace 統合を活用します。
- 欠点: TensorRT-LLM はコンパイルが必要で、NVIDIA 専用です。TGI はバッチ処理の効率が低い場合があります。どちらも設定オーバーヘッドを追加し、すべてのモデルタイプ (トランスフォーマー以外など) に適合しない場合があります。
- 推奨事項: A/B テストや互換性のあるハードウェアを使用した高スループットなどの本稼働機能を必要とする PyTorch/TensorFlow モデルに適しています。

特殊な高スループット推論エンジン (vLLM など) の使用 vLLM などの高度な推論エンジンを使用し、PagedAttention、インフライトバッチ処理、量子化 (INT8、FP8-KV、AWQ) を使用して LLM サービスを最適化し、EKS 自動スケーリングと統合できます。

- 長所: 高スループットとメモリ効率 (40～60% の VRAM 削減) 、動的リクエスト処理、トークンストリーミング、単一ノードの Tensor Parallel マルチ GPU サポート、幅広いハードウェア互換性。
- 短所: デコーダーのみのトランスフォーマー (LLaMA など) 用に最適化され、トランスフォーマー以外のモデルでは効果が低く、互換性のあるハードウェア (NVIDIA GPUs など) とセットアップ作業が必要です。
- 推奨事項: EKS でのハイボリュームで低レイテンシーの LLM 推論に最適な選択肢。スケーラビリティとパフォーマンスを最大化します。

コンテナイメージのプル時間の最適化

コンテナイメージが大きいと、ポッドの起動レイテンシーに影響するコールドスタートの遅延が発生する可能性があります。リアルタイム推論ワークロードを水平方向にスケーリングするなど、レイテンシーの影響を受けやすいワークロードでは、ポッドの迅速な起動が重要です。コンテナイメージのプル時間を最適化するには、次のアプローチを検討してください。

コンテナイメージサイズの縮小

起動時にコンテナイメージのサイズを減らすことも、イメージを小さくする方法の 1 つです。コンテナイメージのビルドプロセスの各ステップで削減できます。開始するには、必要な依存関係の数が最も少ないベースイメージを選択します。イメージの構築中に、必要な必須ライブラリと

アーティファクトのみを含めます。イメージを構築するときは、複数の RUNまたは COPY コマンドを組み合わせて、より多数の大きなレイヤーを作成してみてください。AI/ML フレームワークの場合、マルチステージビルドを使用してビルドとランタイムを分離し、必要なアーティファクトのみをコピーし (レジストリやローカルコンテキストCOPY –from=の場合は を介して)、ランタイム専用イメージなどのバリアントを選択します (例: 3.03 GB pytorch/pytorch:2.7.1-cuda11.8-cudnn9-runtime で、6.66 GB で開発)。詳細については、「AI on EKS Workshop」の「Reducing container image size」を参照してください。

SOCI スナップショッターを使用してイメージをプリプルする

簡単に最小化できない非常に大きなイメージでは、並列プルおよびアンパックモードで設定されたオープンソースの Seekable OCI (SOCI) スナップショッターを使用できます。このソリューションでは、ビルドパイプラインを再構築または変更することなく、既存のイメージを使用できます。このオプションは、非常に大きなイメージを持つワークロードを高性能 EC2 コンピューティングインスタンスにデプロイする場合に特に効果的です。スケールされた AI/ML ワークロードで一般的なように、高スループットのネットワーキングと高性能ストレージ設定でうまく機能します。

SOCI 並列プル/アンパックモードは、設定可能な並列化戦略を通じてend-to-endのイメージプルパフォーマンスを向上させます。イメージのプルと準備を高速化すると、新しいワークロードをデプロイし、クラスターを効率的にスケールする速度に直接影響します。イメージプルには主に 2 つのフェーズがあります。

1. レジストリからノードへのレイヤーの取得

   レイヤーフェッチの最適化のために、SOCI はレイヤーごとに複数の同時 HTTP 接続を作成し、単一接続の制限を超えてダウンロードスループットを乗算します。大きなレイヤーをチャンクに分割し、複数の接続間で同時にダウンロードします。このアプローチは、利用可能なネットワーク帯域幅を飽和させ、ダウンロード時間を大幅に短縮するのに役立ちます。これは、1 つのレイヤーを数ギガバイトにできる AI/ML ワークロードに特に役立ちます。

2. これらのレイヤーを解凍して準備し、コンテナを作成する

   レイヤーの解凍を最適化するために、SOCI は複数のレイヤーを同時に処理します。次のレイヤーを開始する前に、各レイヤーが完全に解凍されるのを待つ代わりに、使用可能な CPU コアを使用して複数のレイヤーを同時に解凍および抽出します。この並列処理は、従来の I/O バインド解凍フェーズを、使用可能なコアに合わせてスケールする CPU 最適化オペレーションに変換します。システムは、スループットを最大化しながらファイルシステムの一貫性を維持するために、この並列化を慎重に調整します。

SOCI 並列プルモードは、ダウンロードの同時実行と並列処理の解凍の両方に対して設定可能なパラメータを持つデュアルしきい値の管理システムを使用します。このきめ細かな制御により、特定のパフォーマンス要件と環境条件に合わせて SOCI の動作を微調整できます。これらのパラメータを理解することで、最適なプルパフォーマンスを得るためにランタイムを最適化できます。

リファレンス

- ソリューションとチューニングのトレードオフの詳細については、GitHub の SOCI プロジェクトリポジトリの機能ドキュメントを参照してください。
- Amazon EKS での Karpenter の実践的な例については、SOCI スナップショッター並列プル/アンパックモードを使用した Karpenter 設計図を参照してください。
- Bottlerocket を並列プル用に設定する方法については、Bottlerocket ドキュメントの「soci-snapshotter Parallel Pull Unpack Mode」を参照してください。

EBS スナップショットを使用したイメージのプリプル

キャッシュされたコンテナイメージの Amazon Elastic Block Store (EBS) スナップショットを作成し、このスナップショットを EKS ワーカーノードに再利用できます。これにより、ノードの起動時にイメージがローカルでプリフェッチされ、ポッドの初期化時間が短縮されます。詳細については、「マネージド型ノードグループの Karpenter と EKS Terraform ブループリントを使用して、Bottlerocket データボリュームで Amazon EKS のコンテナ起動時間を短縮する」を参照してください。 https://aws-ia.github.io/terraform-aws-eks-blueprints/patterns/machine-learning/ml-container-cache/

詳細については、「AI on EKS Workshop」の「Using containerd snapshotter and Preload container images into Bottlerocket data volumes with EBS Snapshots」を参照してください。

コンテナランタイムキャッシュを使用したイメージのプリプル

Kubernetes リソース (DaemonSet や Deployment など) を使用して、コンテナイメージをノードにプリプルして、ノードのコンテナランタイムキャッシュに入力できます。コンテナランタイムキャッシュは、コンテナランタイムによって管理されるローカルストレージです (例: レジストリからプルされた後にイメージが保存される containerd。 プル前により、イメージがローカルで使用可能になり、ポッドの起動時のダウンロード遅延を回避できます。 このアプローチは、イメージが頻繁に変更される場合 (頻繁な更新など) 、EBS スナップショットが事前設定されていない場合、EBS ボリュームの構築にコンテナレジストリから直接プルするよりも時間がかかる場合、またはノードがすでにクラスターにあり、可能なイメージのいずれかを使用してオンデマンドでポッドをスピンアップする必要がある場合に特に役立ちます。

すべてのバリアントをプリプルすることで、必要なイメージに関係なく、起動時間を短縮できます。例えば、10 の異なる手法を使用して構築された 100,000 個の小型モデルを必要とする超並列 ML ワークロードでは、大規模なクラスター (数千のノードなど) 全体で DaemonSet を介して 10 個のイメージをプリプルすることで、ポッドの起動時間が最小限に抑えられ、オンデマンドプルを回避することで 10 秒未満で完了できます。コンテナランタイムキャッシュアプローチを使用すると、EBS スナップショットを管理する必要がなくなり、DaemonSets で常に最新のコンテナイメージバージョンを取得できますが、ノードがスケールイン/スケールアウトするリアルタイム推論ワークロードの場合、Cluster Autoscaler などのツールによって追加された新しいノードは、プル前の DaemonSet がイメージプルを完了する前にワークロードポッドをスケジュールする場合があります。これにより、新しいノードの最初のポッドがプルをトリガーし、起動が遅れ、低レイテンシーの要件に影響を与える可能性があります。さらに、kubelet イメージガベージコレクションは、ディスク使用量が特定のしきい値を超えた場合、または設定された最大未使用期間を超えた場合に、未使用のイメージを削除することで、プル済みイメージに影響を与える可能性があります。スケールイン/アウトパターンでは、アイドル状態のノード上のイメージが削除される可能性があるため、その後のスケールアップ中に再プルし、バーストワークロードのキャッシュの信頼性を低下させる必要があります。

コンテナランタイムキャッシュにイメージをプルする例については、AWS GitHub リポジトリを参照してください。

kubelet ストレージと containerd ストレージの NVMe を検討する

エフェメラル NVMe インスタンスストレージディスクを使用してディスクパフォーマンスを向上させるcontainerdには、 kubeletと の設定を検討してください。コンテナプルプロセスでは、レジストリからコンテナイメージをダウンロードし、そのレイヤーを使用可能な形式に解凍します。解凍中に I/O オペレーションを最適化するには、コンテナホストのインスタンスタイプにより高いレベルの I/O パフォーマンスとスループットを提供するものを評価する必要があります。ローカルストレージを搭載した NVMe バックドインスタンスと EBS ボリューム IOPS/スループットです。NVMe ローカルストレージを使用する EC2 インスタンスの場合、ノードの基盤となるファイルシステムを kubelet (/var/lib/kubelet)、containerd (/var/lib/containerd)、Pod ログ (/var/log/pods) に設定して、エフェメラル NVMe インスタンスストレージディスクを使用して I/O パフォーマンスとスループットを向上させることを検討してください。

ノードのエフェメラルストレージは、ノードにダウンロードされるエフェメラルストレージとコンテナイメージをリクエストするポッド間で共有できます。Bottlerocket または AL2023 EKS 最適化 AMIs で Karpenter を使用する場合、これは instanceStorePolicy を RAID0 に設定するか、マネージドノードグループを使用する場合は、ユーザーデータの一部として NodeConfig の localStoragePolicy を設定することで ECEC2NodeClass で設定できます。

このガイドへの貢献

誰でもベストプラクティスガイドに貢献できます。EKS ベストプラクティスガイドは、GitHub の AsciiDoc 形式で書かれています。

既存の寄稿者の概要

- VS Code `bpg-docs.code-workspace`で を開き、AsciiDoc 拡張機能を自動的にインストールします。
  - Visual Studio Marketplace の AsciiDoc 拡張機能の詳細をご覧ください。
- AWS Docs ウェブサイトのソースファイルは に保存されます。 `latest/bpg`
- 構文はマークダウンとよく似ています。
  - AsciiDoctor ドキュメントの構文リファレンスを確認します。
- docs プラットフォームは のみをデプロイします`latest/bpg/images`。各ガイドセクションには、このディレクトリへのシンボリックリンクがあります。たとえば、 は を`latest/bpg/networking/images`指していますlatest/bpg/images。

ローカル編集環境のセットアップ

ガイドを頻繁に編集する場合は、ローカル編集環境を設定します。

リポジトリをフォークしてクローンする

`git`、`github`、およびテキストエディタに精通している必要があります。`git` および の開始方法についてはgithub、GitHub ドキュメントの「GitHub アカウントの開始方法」を参照してください。GitHub

1. GitHub で EKS ベストプラクティスガイドを参照してください。
2. プロジェクトリポジトリのフォークを作成します。GitHub ドキュメントでリポジトリをフォークする方法について説明します。
3. プロジェクトリポジトリのフォークをクローンします。分岐リポジトリのクローンを作成する方法について説明します。

VS Code Workspace を開く

AWS では、Microsoft の Visual Studio Code を使用してガイドを編集することをお勧めします。VS Code の詳細については、Visual Studio Code ドキュメントの「Visual Studio Code のダウンロード」および「Visual Studio Code の開始方法」を参照してください。

1. VS Code を開きます。
2. クローンされたリポジトリから `bpg-docs.code-workspace` ファイルを開きます。
3. このワークスペースを初めて開く場合は、AsciiDoc 拡張機能をインストールするプロンプトを受け入れます。この拡張機能は、AsciiDoc ファイルの構文をチェックし、ライブプレビューを生成します。
4. `latest/bpg` ディレクトリを参照します。このディレクトリは、AWS ドキュメントサイトにデプロイするソースファイルを保持します。ソースファイルは、「セキュリティ」や「ネットワーク」などのガイドセクション別に整理されています。

ファイルの編集

1. エディタでファイルを開きます。
   - 見出し、リンク、リストの作成方法については、AsciiDoc 構文を参照してください。
   - Markdown 構文を使用して、テキスト、リスト、見出しをフォーマットできます。Markdown 構文を使用してリンクを作成することはできません。
2. ページのライブプレビューを開きます。
   - まず、 `ctrl-k`または を押します `cmd-k` (キーボードによって異なります) 。次に、 を押しますv。これにより、分割ビューでプレビューが開きます。

AWS では、特徴量ブランチを使用して変更を整理することをお勧めします。git を使用してブランチを作成する方法について説明します。

プルリクエストを送信する

GitHub ウェブサイトまたは GitHub CLI からプルリクエストを作成できます。

GitHub ウェブサイトを使用してフォークからプルリクエストを作成する方法について説明します。

GitHub CLI を使用してプルリクエストを作成する方法について説明します。

github.dev ウェブベースのエディタを使用する

github.dev ウェブベースのエディタは VS Code に基づいています。これは、セットアップなしで複数のファイルを編集し、コンテンツをプレビューする優れた方法です。

AsciiDoc 拡張機能がサポートされています。GUI を使用して git オペレーションを実行できます。ウェブベースのエディタには、コマンドを実行するためのシェルやターミナルはありません。

GitHub アカウントが必要です。必要に応じてログインするように求められます。

GitHub ウェブベースのエディタを起動します。

1 ページの編集

GitHub を使用すると、個々のページをすばやく更新できます。各ページには、下部に#GitHub でこのページを編集する」リンクがあります。

1. 編集するこのガイドのページに移動します。
2. 下部にあるGitHub でこのページを編集する」リンクをクリックします。
3. GitHub ファイルビューワーの右上にある編集鉛筆アイコンをクリックするか、 e
4. ファイルを編集する
5. 「コミット変更...」ボタンを使用して変更を送信します。このボタンは GitHub プルリクエストを作成します。ガイドの管理者は、このプルリクエストを確認します。レビューワーはプルリクエストを承認するか、変更をリクエストします。

ページの ID を表示および設定する

このページでは、ページ ID を表示および設定する方法について説明します。

ページ ID は、ドキュメントサイトの各ページを識別する一意の文字列です。特定のページにいるときに、ブラウザのアドレスバーでページ ID を表示できます。ページ ID は、URL、ファイル名、およびクロスリファレンスリンクの作成に使用されます。

たとえば、このページを表示している場合、ブラウザのアドレスバーの URL は次のようになります。

```
https://docs.aws.amazon.com/view-set-page-id.html
```

URL (`view-set-page-id`) の最後の部分はページ ID です。

ページ ID を設定する

新しいページを作成するときは、ソースファイルでページ ID を設定する必要があります。ページ ID は、ページの内容を説明する簡潔なハイフン文字列である必要があります。

1. テキストエディタで新しいページのソースファイルを開きます。
2. ファイルの上部に、次の行を追加します。最初の見出しより上である必要があります。

   ```
   [#my-new-page]
   ```

   を新しいページのページ ID `my-new-page`に置き換えます。
3. ファイルを保存します。

> **Note**
>
> ページ IDsは、ドキュメントサイト全体で一意である必要があります。既存のページ ID を使用しようとすると、ビルドエラーが発生します。

新しいページを作成する

新しいページを作成し、ガイドの目次を更新する方法について説明します。

ページメタデータを作成する

1. ページタイトルとページショートタイトルを決定します。ページの短いタイトルはオプションですが、ページのタイトルが数語を超える場合は推奨されます。
2. ページの ID を決定します。これは、EKS ベストプラクティスガイド内で一意である必要があります。規則では、すべての小文字を使用し、 で単語を区切ります`-`。
3. 必要に応じてフォルダに新しい asciidoc ファイルを作成し、次のテキストをファイルに追加します。

   Example

   ["topic"] [#<page-id>] = <page-title> :info_titleabbrev: <page-short-title>

例えば、

Example

["topic"] [#scalability] = EKS Scalability のベストプラクティス :info_titleabbrev: Scalability

目次に追加する

1. 目次で親ページの ファイルを開きます。新しい最上位ガイドセクションの場合、親ファイルは ですbook.adoc。
2. 親ファイルの下部で、次のディレクティブを更新して挿入します。

Example

include::<new-filename>[leveloffset=+1]

例えば、

Example

include::dataplane.adoc[leveloffset=+1]

イメージを挿入する

1. 編集するページのイメージプレフィックスを見つけます。ファイルの見出しの :imagesdir:プロパティを確認します。例については、 `:imagesdir: images/reliability/
2. イメージを次のようにこのパスに配置します。 latest/bpg/images/reliability
3. イメージに適した alt-text を決定します。イメージの簡単な概要を記述します。たとえば、「3 つのアベイラビリティーゾーンを持つ VPC のダイアグラム」は適切な alt-text です。
4. 次の例を alt-text とイメージファイル名で更新します。目的の場所に を挿入します。

Example

image::<image-filename>[<image-alt-text>]

例えば、

Example

image::eks-data-plane-connectivity.jpeg[ネットワーク図]

Vale でスタイルを確認する

1. Vale CLI をインストールします。
2. `vale sync` を実行する
3. Visual Studio Marketplace から Vale Extension をインストールします。
4. VS Code を再起動し、AsciiDoc ファイルを開きます。
5. VS Code は問題のあるテキストを強調します。VS Code ドキュメントでエラーと警告を使用する方法について説明します。

ローカルプレビューを構築する

1. Linux または MacOS `brew` で を使用して`asciidoctor`ツールをインストールする
   - AsciiDoctor ドキュメントで asciidoctor CLI をインストールする方法について説明します。AsciiDoctor
   - brew パッケージマネージャーをインストールする方法について説明します。
2. ターミナルを開き、 に移動します。 `latest/bpg/`
3. `asciidoctor book.adoc` を実行する
   - 構文の警告とエラーを確認する
4. `book.html` 出力ファイルを開きます。
   - MacOS では、 `open book.html`を実行してデフォルトのブラウザでプレビューを開くことができます。

AsciiDoc チートシート

基本的なフォーマット

```
*bold text*
_italic text_
```

```
`monospace text`
```

ヘッダー

```
= Document Title (Header 1)
== Header 2
=== Header 3
==== Header 4
===== Header 5
====== Header 6
```

Lists

順序付けられていないリスト:

```
- Item 1
- Item 2
-- Subitem 2.1
-- Subitem 2.2
- Item 3
```

順序付けられたリスト:

```
. First item
. Second item
.. Subitem 2.1
.. Subitem 2.2
. Third item
```

Links

```
External link:  https://example.com[Link text]
Internal link: <<page-id>>
Internal link: xref:page-id[Link text]
```

イメージ

```
image::image-file.jpg[Alt text]
```

コードブロック

```
[source,python]
----
def hello_world():
    print("Hello, World!")
----
```

テーブル

基本的なテーブルの構築方法はこちらをご覧ください。

```
[cols="1,1"]
|===
|Cell in column 1, row 1
|Cell in column 2, row 1

|Cell in column 1, row 2
|Cell in column 2, row 2

|Cell in column 1, row 3
|Cell in column 2, row 3
|===
```

警告

```
NOTE: This is a note admonition.

WARNING: This is a warning admonition.

TIP: This is a tip admonition.

IMPORTANT: This is an important admonition.

CAUTION: This is a caution admonition.
```

プレビュー:

> **Note**
>
> これはメモの警告です。

含まれるもの

```
include::filename.adoc[]
```

翻訳は機械翻訳により提供されています。提供された翻訳内容と英語版の間で齟齬、不一致または矛盾がある場合、英語版が優先します。