AWS ホワイトペーパー

AWS でのマイクロサービスの実装

AWS でのマイクロサービスの実装: AWS ホワイトペーパー

Table of Contents

要約と紹介 i
要約 1
はじめに 1
AWS でのマイクロサービスアーキテクチャ 3
ユーザーインターフェイス 4
マイクロサービス 4
マイクロサービスの実装 4
プライベートリンク 6
データストア 6
運用の複雑さの低減 8
API の実装 8
サーバーレスマイクロサービス 9
災害対策 11
高可用性 12
Lambda ベースのアプリケーションのデプロイ 12
分散システムのコンポーネント 14
サービス検出 14
DNS ベースのサービス検出 14
サードパーティー製ソフトウェア 15
サービスメッシュ 15
分散データ管理 16
設定管理 19
非同期通信と軽量メッセージング 19
REST ベースの通信 19
非同期メッセージングとイベントパッシング 20
オーケストレーションとステート管理 22
分散モニタリング 24
モニタリング 24
ログの一元管理 25
分散トレース 26
AWS でのログ分析のオプション 28
チャティネス (対話頻度) 31
監査 32
まとめ 36

リソース 37
ドキュメントの改訂履歴と寄稿者 38
ドキュメント履歴 38
寄稿者 39
注意 40

AWS でのマイクロサービスの実装

発行日: 2021 年 11 月 9 日 (ドキュメントの改訂履歴と寄稿者)

要約

マイクロサービスは、ソフトウェア開発のアーキテクチャ的、組織的アプローチです。アジャイル手法に従って複数のチームが相互に独立して作業することで、デプロイサイクルを高速化し、イノベーションとオーナーシップを促進します。また、ソフトウェアアプリケーションの保全性とスケーラビリティを向上させ、ソフトウェアとサービスを配信する組織をスケールします。マイクロサービスアプローチでは、複数の小さなサービスでソフトウェアを構成します。これらのサービスは、個別にデプロイできる、明確に定義されたアプリケーションプログラムインターフェイス (API) を介して相互に通信します。これらのサービスは、小規模な独立したチームが所有します。このアジャイル手法が、組織を適切にスケールする鍵となります。

AWS のお客様がマイクロサービスを構築する際の一般的なパターンとして、API 駆動型、イベント駆動型、データストリーミングの 3 つが確認されています。このホワイトペーパーでは、3 つすべてのアプローチを紹介し、マイクロサービスの一般的な特徴を概括します。また、マイクロサービスを構築する際の主な課題について検討し、これらの課題を克服するために製品チームがアマゾン ウェブ サービス (AWS) をどのように活用できるかについて説明します。

このホワイトペーパーで説明するトピックは、データストア、非同期通信、サービス検出など、相互に複雑に関連しているため、本書内のガイダンスに加えてアプリケーションの特定の要件やユースケースを検討したうえで、アーキテクチャを選択することお勧めします。

はじめに

マイクロサービスアーキテクチャは、ソフトウェアエンジニアリングに対するまったく新しいアプローチというわけではなく、むしろ以下のような成功実績のあるさまざまなコンセプトを組み合わせたものです。

- アジャイルソフトウェア開発
- サービス指向アーキテクチャ
- API ファースト設計
- 継続的インテグレーション/継続的デリバリー (CI/CD)

多くの場合、マイクロサービスには Twelve-Factor App の設計パターンが使用されています。

このホワイトペーパーでは、まず、スケーラビリティと耐障害性に優れたマイクロサービスアーキテクチャのさまざまな側面 (ユーザーインターフェイス、マイクロサービスの実装、データストア) を紹介し、コンテナ技術を活用して AWS でアーキテクチャを構築する方法について説明します。次に、一般的なサーバーレスマイクロサービスアーキテクチャを実装して運用の複雑さを低減させるための AWS のサービスを推奨します。

サーバーレスは、次の基本的概念に基づく運用モデルとして定義されます。

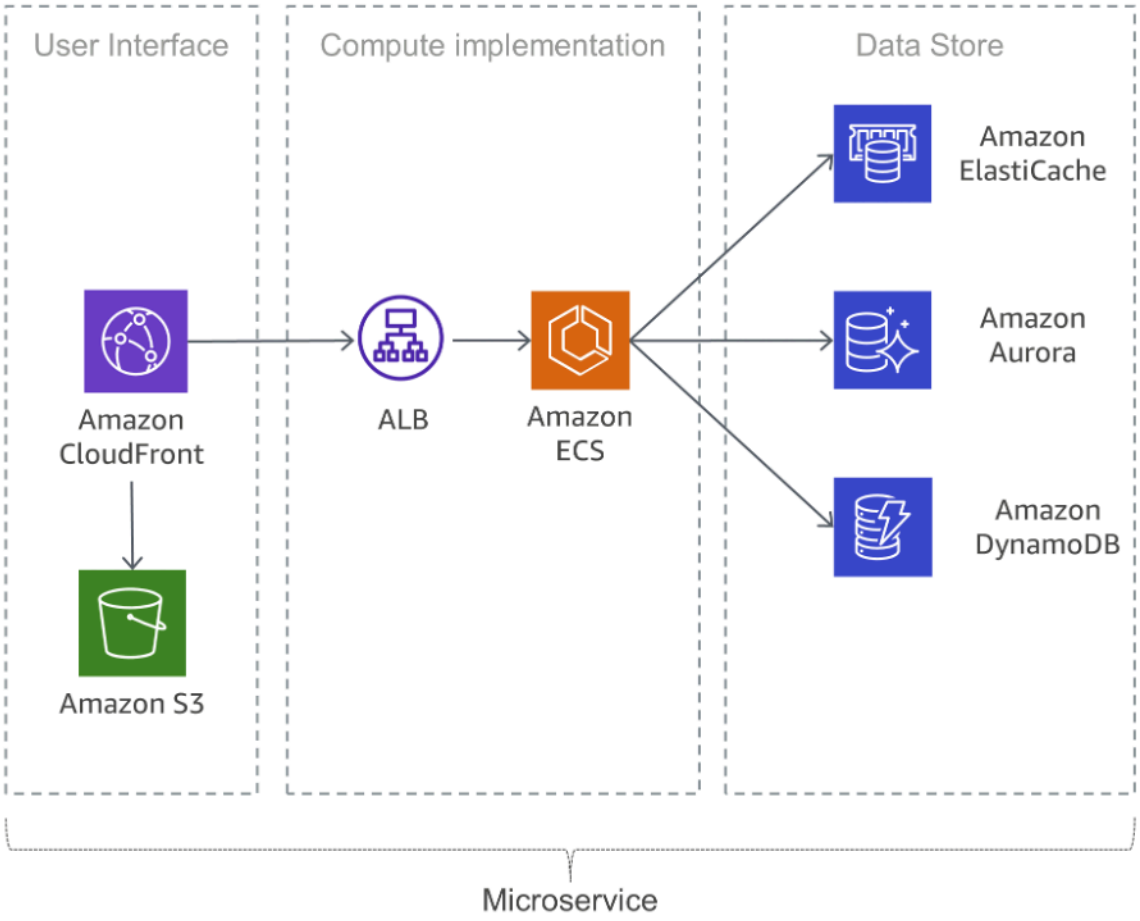
- インフラストラクチャのプロビジョニングや管理が不要
- 消費単位によるオートスケーリング
- 「価値に対する支払い」請求モデル
- 可用性と耐障害性が組み込み済み

最後に、このホワイトペーパーではシステム全体について、分散モニタリング、分散監査、データ整合性、非同期通信など、マイクロサービスアーキテクチャのクロスサービスの側面について説明します。

このホワイトペーパーでは、AWS クラウド内で実行されるワークロードのみを対象としています。ハイブリッドシナリオや移行戦略は対象外です。移行の詳細については、「コンテナ移行の方法論」ホワイトペーパーを参照してください。

AWS でのマイクロサービスアーキテクチャ

一般的なモノリシックアプリケーションは、ユーザーインターフェイス (UI) レイヤー、ビジネスレイヤー、永続レイヤーなど、さまざまなレイヤーを使用して構築されています。マイクロサービスアーキテクチャの主旨は、複数の機能をまとまりのあるバーティカルに分割することです。技術的レイヤーで分割するのではなく、特定のドメインを実装することで分割します。次の図は、AWS での一般的なマイクロサービスアプリケーションのリファレンスアーキテクチャを示しています。



AWS での一般的なマイクロサービスアプリケーション

トピック

- ユーザーインターフェイス
- マイクロサービス
- データストア

ユーザーインターフェイス

モダンウェブアプリケーションでは、多くの場合、JavaScript フレームワークを使用して、Representational State Transfer (REST) または RESTful API と通信するシングルページアプリケーションを実装します。静的ウェブコンテンツは、Amazon Simple Storage Service (S3) と Amazon CloudFront を使用して提供できます。

マイクロサービスのクライアントは、最寄りのエッジロケーションからサービスの提供を受け、キャッシュまたは最適な形で送信元に接続するプロキシサーバーから応答を得るため、レイテンシーを大幅に短縮できます。ただし、互いに近くで実行しているマイクロサービスにはコンテンツ配信ネットワークの恩恵がありません。場合によっては、このアプローチで余計なレイテンシーが実際に増える可能性もあります。チャティネス (chattiness) を軽減し、レイテンシーを最小化するには、他のキャッシュメカニズムを実装することをお勧めします。詳細については、「「the section called “チャティネス (対話頻度)”」トピックを参照してください。

マイクロサービス

API はマイクロサービスの玄関口です。つまり、API は一連のプログラムインターフェイス (通常は RESTful ウェブサービス API) の背後にあるアプリケーションロジックのエントリポイントとして機能します。この API はクライアントからの呼び出しを受信して処理し、トラフィック管理、リクエストのフィルタリング、ルーティング、キャッシュ、認証、認可などの機能を実装する場合があります。

マイクロサービスの実装

AWS は、マイクロサービスの開発をサポートするための統合された構成ブロックを備えています。2 つの一般的なアプローチは、AWS Lambda を使用すること、および Docker コンテナとともに AWS Fargate を使用することです。

AWS Lambda を使用すると、コードをアップロードするだけで、高可用性を維持しながら実装を実行およびスケールして実際の需要曲線を満たすために必要なすべてのことを Lambda が引き受けます。インフラストラクチャの管理は不要です。Lambda は数種類のプログラミング言語をサポートしており、AWS の他のサービスからトリガーすることも、任意のウェブアプリケーションやモバイルアプリケーションから直接呼び出すこともできます。AWS Lambda を利用する最大のメリットの 1 つが、迅速に対応できるという点です。AWS がセキュリティとスケーリングを管理するので、ユーザーはビジネスロジックに集中できます。Lambda の自律的なアプローチによって、スケーラブルなプラットフォームが実現します。

デプロイの運用作業を低減する一般的な方法は、コンテナベースのデプロイです。Docker などのコンテナ技術は、ポータビリティ、生産性、効率性などの利点を理由に、ここ数年人気が高まっています。コンテナの習得には集中して取り組む必要があります。また、Docker イメージのセキュリティ修正とモニタリングについて考える必要があります。Amazon Elastic Container Service (Amazon ECS) と Amazon Elastic Kubernetes Service (Amazon EKS) では、独自のクラスター管理インフラストラクチャをインストール、運用、スケールする必要がありません。API コールを使用して、Docker 対応のアプリケーションを起動および停止したり、クラスターの全体的な状態をクエリしたりできます。また、セキュリティグループ、ロードバランシング、Amazon Elastic Block Store (Amazon EBS) ボリューム、AWS Identity and Access Management (IAM) ロールなど、多くの使い慣れた機能にもアクセスできます。

AWS Fargate は、コンテナ向けサーバーレスコンピューティングエンジンであり、Amazon ECS および Amazon EKS の両方と連携します。Fargate では、コンテナアプリケーションに対して十分なコンピューティングリソースのプロビジョニングを心配する必要はなくなります。Fargate では、何万ものコンテナを起動して、簡単にスケーリングし、最もミッションクリティカルなアプリケーションを実行できます。

Amazon ECS は、タスクを配置および終了する方法をカスタマイズするためのコンテナ配置戦略と制約事項をサポートしています。タスク配置の制約事項とは、タスクを配置する際に考慮するルールを指します。コンテナインスタンスに属性 (基本的にキーと値のペア) を関連付けた後で、制約事項を使用して、これらの属性に基づいてタスクを配置できます。例えば、制約事項を使用して、インスタンスタイプやインスタンス機能 (GPU を使用するインスタンスなど) に基づいて特定のマイクロサービスを配置できます。

Amazon EKS は、オープンソース Kubernetes ソフトウェアの最新バージョンを実行します。これにより、ユーザーは Kubernetes コミュニティのすべての既存のプラグインとツールを使用できます。Amazon EKS で実行されるアプリケーションは、オンプレミスのデータセンターで実行されるものであれ、パブリッククラウドで実行されるものであれ、標準的な Kubernetes 環境で実行されるアプリケーションと完全な互換性があります。Amazon EKS では IAM と Kubernetes を統合しているため、IAM エンティティを Kubernetes のネイティブ認証システムに登録できます。Kubernetes コントロールプレーンで認証するための認証情報を手動で設定する必要はありません。IAM 統合により、IAM を使用してコントロールプレーン自体で直接認証し、Kubernetes コントロールプレーンのパブリックエンドポイントにきめ細かくアクセスできるようになります。

Amazon ECS と Amazon EKS で使用する Docker イメージは、Amazon Elastic Container Registry (Amazon ECR) に保存できます。Amazon ECR では、コンテナレジストリを実行するためにインフラストラクチャを運用およびスケールする必要はありません。

継続的インテグレーションと継続的デリバリー (CI/CD) はベストプラクティスであり、システムの安定性とセキュリティを維持しながら、ソフトウェアを迅速に変更できるようにする DevOps イニシアチブの重要な部分です。ただし、このホワイトペーパーでは対象外です。詳細については、「AWS での継続的インテグレーションと継続的デリバリーの実践」ホワイトペーパーを参照してください。

プライベートリンク

AWS PrivateLink は、可用性とスケーラビリティに優れたテクノロジーであり、仮想プライベートクラウド (VPC) を、サポートされている AWS のサービス、他の AWS アカウントでホストされているサービス (VPC エンドポイントサービス)、およびサポートされている AWS Marketplace パートナーサービスにプライベートに接続できるようにします。サービスと通信するために、インターネットゲートウェイ、ネットワークアドレス変換デバイス、パブリック IP アドレス、AWS Direct Connect の接続、または VPN 接続を必要としません。VPC と サービスとの間のトラフィックは、Amazon ネットワークを離れません。

プライベートリンクは、マイクロサービスアーキテクチャの分離とセキュリティを強化する優れた方法です。例えば、マイクロサービスを完全に独立した VPC にデプロイして、ロードバランサーを前に配置し、AWS PrivateLink エンドポイントを介して他のマイクロサービスに公開できます。このセットアップでは、AWS PrivateLink を使用することで、マイクロサービスとの間で送受信されるネットワークトラフィックがパブリックインターネットを通過することはありません。このような分離のユースケースの 1 つとして、PCI、HIPPA、EU/US Privacy Shield などの機密データを扱うサービスに対する規制コンプライアンスがあります。さらに、AWS PrivateLink を使用すると、ファイアウォールルール、パス定義、またはルートテーブルを必要とせずに、複数の異なるアカウントや Amazon VPC をまたいでマイクロサービスを接続できるため、ネットワーク管理が簡素化されます。PrivateLink を利用することで、Software as a Service (SaaS) プロバイダーや ISV は、運用を完全に分離してアクセスをセキュリティで保護した、マイクロサービスベースのソリューションも提供できます。

データストア

データストアは、マイクロサービスに必要なデータを保持するために使用されます。セッションデータ用のストアとしては、Memcached や Redis などのインメモリキャッシュが通常使用されます。AWS は、両方のテクノロジーを Amazon ElastiCache マネージドサービスの一部として提供しています。

アプリケーションサーバーとデータベースの間にキャッシュを配置することは、データベースでの読み取り負荷を軽減するためによく使用されるメカニズムです。負荷の軽減によって、リソースでより

多くの書き込みをサポートできるようになります。キャッシュによってレイテンシーも改善できます。

リレーショナルデータベースは、構造化データやビジネスオブジェクトを保存する手段として今でもよく使用されています。AWS は、6 つのデータベースエンジン (Microsoft SQL Server、Oracle、MySQL、MariaDB、PostgreSQL、Amazon Aurora) をマネージドサービスとして、Amazon Relational Database Service (Amazon RDS) 経由で提供しています。

ただし、リレーショナルデータベースは無制限にスケールできるように設計されていないため、大量のクエリに対応する手法を適用するのは困難で、時間がかかる可能性があります。

NoSQL データベースは、リレーショナルデータベースの一貫性よりも、スケーラビリティ、パフォーマンス、可用性を重視するように設計されています。NoSQL データベースは通常は、厳格なスキーマを強制しないことが、重要な要素の 1 つです。データは水平方向にスケーリング可能な複数のパーティションに分散され、パーティションキーを使用して取得されます。

各マイクロサービスは、1 つの分野に特化するよう設計されているため、通常そのデータモデルは単純で、NoSQL の持続的特質に適している場合があります。NoSQL データベースにはリレーショナルデータベースとは異なるアクセスパターンがあることに注意してください。例えば、テーブルを結合することはできません。結合の必要がある場合は、アプリケーションにロジックを実装する必要があります。Amazon DynamoDB を使用して、任意の量のデータの保存や取得、任意の量のリクエストトラフィックの処理が可能なデータベーステーブルを作成できます。DynamoDB は 1 桁ミリ秒のパフォーマンスが可能ですが、特定のユースケースでは、マイクロ秒単位の応答時間が必要です。Amazon DynamoDB Accelerator (DAX) は、データにアクセスするためのキャッシュ機能を備えています。

DynamoDB は、実際のトラフィックに応じてスループット容量を動的に調節するオートスケーリング機能も提供します。ただし、アプリケーションのアクティビティが短時間に急増するため、キャパシティープランニングが困難になるか、または不可能となる場合があります。そういった状況に備え、DynamoDB ではオンデマンドオプションを用意しており、シンプルなリクエスト単位での料金支払いとなっています。DynamoDB オンデマンドは、キャパシティープランニングなしでただちに 1 秒に数千のリクエストを処理する能力があります。

運用の複雑さの低減

このホワイトペーパーで前述したアーキテクチャでは既にマネージドサービスを使用していますが、Amazon Elastic Compute Cloud (Amazon EC2) インスタンスは依然として管理する必要があります。完全なサーバーレスアーキテクチャを使用することで、マイクロサービスの実行、維持、モニタリングに必要な運用作業をさらに低減できます。

トピック

- API の実装
- サーバーレスマイクロサービス
- 災害対策
- 高可用性
- Lambda ベースのアプリケーションのデプロイ

API の実装

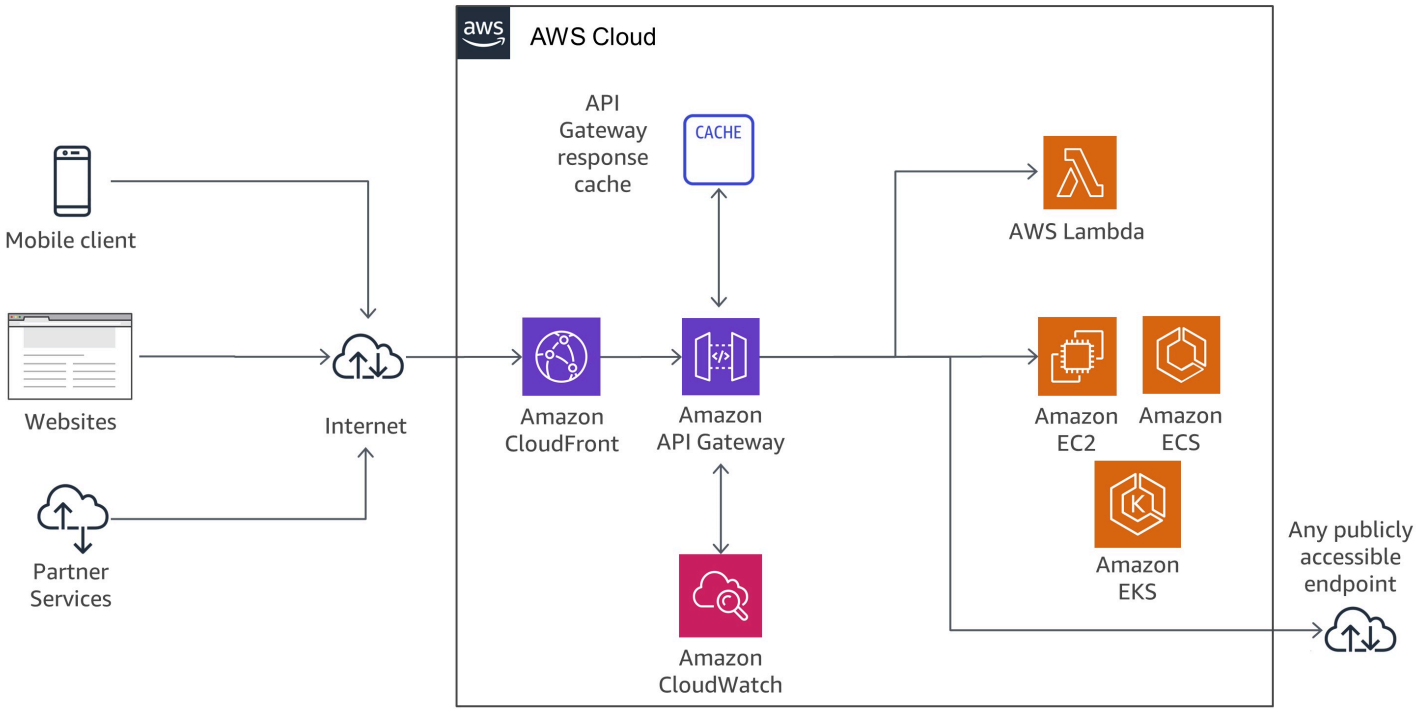
API のアーキテクチャの設計、デプロイ、モニタリング、継続的な改善、メンテナンスは、時間のかかるタスクとなりがちです。すべてのクライアントに対する下位互換性を確保するために、複数の異なるバージョンの API を実行することが必要になる場合もあります。開発サイクルの異なるステージ (開発、テスト、本番) により、運用作業はさらに何倍も増加します。

認可はすべての API に重要な機能ですが、通常は構築が複雑で、反復作業が伴います。API を公開し、それが正常に稼働し始めると、その API を利用するサードパーティーのデベロッパーのエコシステムを管理、モニタリング、収益化することが次の課題となります。

他の重要な機能や課題としては、バックエンドサービスを保護するためのリクエストのスロットリング、API レスポンスのキャッシュ、リクエストとレスポンスの変換、Swagger などのツールを使用した API 定義やドキュメントの生成が挙げられます。

Amazon API Gateway は、これらの課題に対処し、RESTful API の作成とメンテナンスに伴う運用上の複雑さを低減します。API Gateway を使用すると、AWS API または AWS マネジメントコンソールを通じて Swagger 定義をインポートすることで、API をプログラムによって作成できます。API Gateway は、Amazon EC2、Amazon ECS、AWS Lambda、または任意のオンプレミス環境で実行されるあらゆるウェブアプリケーションの玄関口として機能します。要するに、API Gateway を使用することで、サーバーを管理することなく API を実行できるということです。

次の図は、API Gateway が API コールをどのように処理し、他のコンポーネントとどのようにやり取りするかを示しています。レイテンシーを最小限に抑え、最適なユーザーエクスペリエンスを実現するために、モバイルデバイス、ウェブサイト、その他のバックエンドサービスからのリクエストは、最寄りの CloudFront POP (Point of Presence) にルーティングされます。



API Gateway の呼び出しフロー

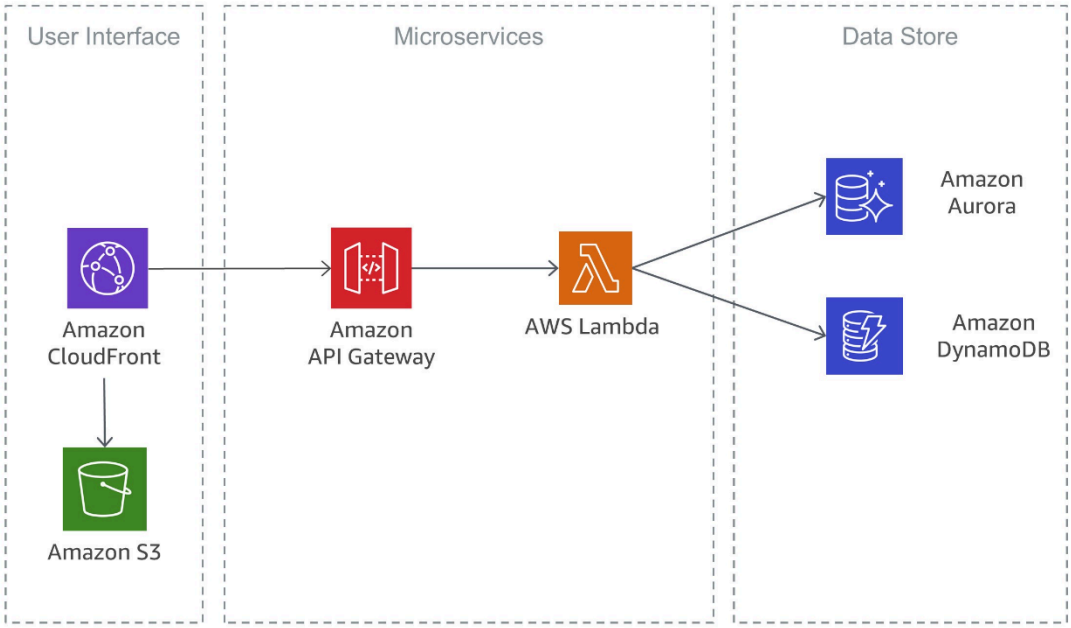
サーバーレスマイクロサービス

「サーバーが無いに越したことはない」

サーバーを一掃することは、運用面での複雑さを排除する究極の手段です。

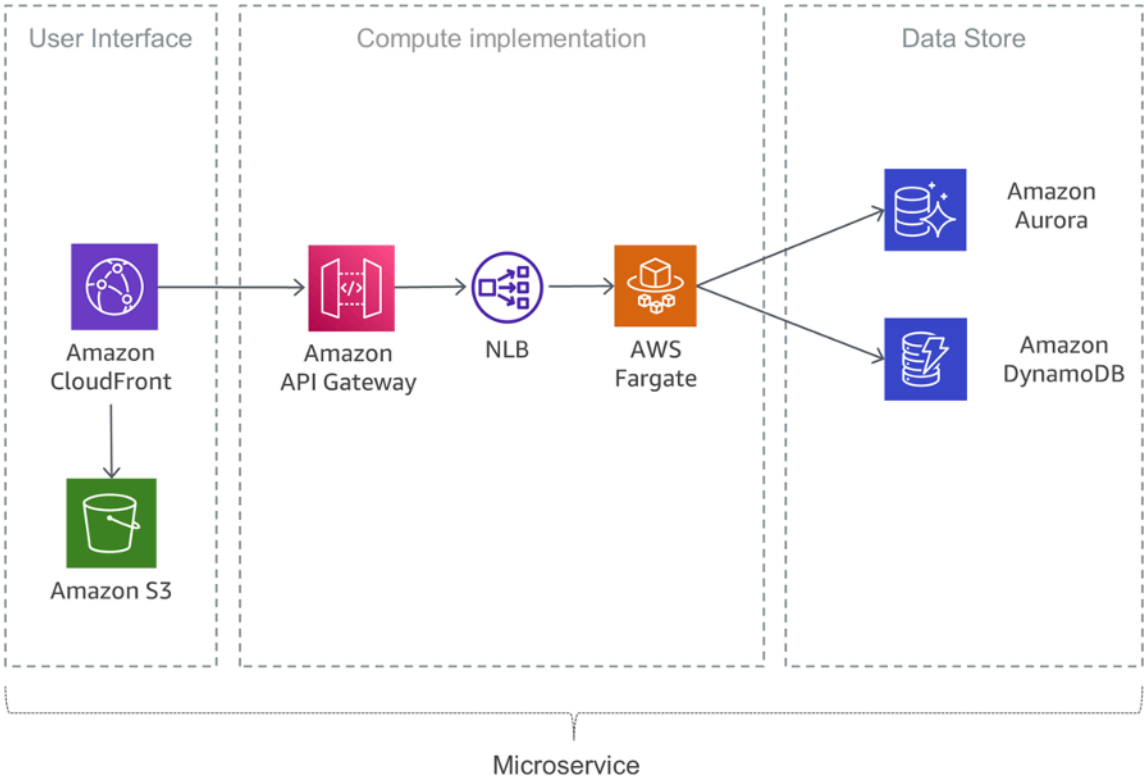
Lambda は、API Gateway と緊密に統合されています。API Gateway から Lambda に同期呼び出しができると、完全なサーバーレスアプリケーションを作成できるようになります。これについては、Amazon API Gateway デベロッパーガイドに詳しい説明があります。

次の図に示す AWS Lambda を使用したサーバーレスマイクロサービスのアーキテクチャでは、サービス全体がマネージドサービスで構築されています。このアーキテクチャでは、スケーリングと高可用性を考慮してアーキテクチャを設計するという負担がなくなり、マイクロサービスの基盤となるインフラストラクチャを実行およびモニタリングする運用作業が不要になります。



AWS Lambda を使用するサーバーレスマイクロサービス

次の図も、サーバーレスサービスに基づく同様の実装を示しています。このアーキテクチャでは、Docker コンテナを AWS Fargate とともに使用するため、基盤となるインフラストラクチャについて気にする必要はありません。Amazon DynamoDB に加え、Amazon Aurora Serverless を使用しています。これは、Amazon Aurora (MySQL 互換エディション) のオンデマンドの Auto Scaling 設定であり、アプリケーションのニーズに合わせてデータベースの起動、シャットダウン、容量のスケールアップ/ダウンが自動的に行われます。



Fargate を使用するサーバーレスマイクロサービス

災害対策

このホワイトペーパーの冒頭で述べたように、一般的なマイクロサービスアプリケーションは Twelve-Factor Application パターンを使用して実装されます。「プロセス」セクションには、「Twelve-Factor のプロセスはステートレスかつシェアードナッシングである。永続化する必要のあるすべてのデータは、ステートフルなバックエンドサービス (典型的にはデータベース) に格納しなければならない」と記載されています。

一般的なマイクロサービスアーキテクチャの場合、これは、災害対策の主な焦点がアプリケーションの状態を維持するダウンストリームサービスにあるべきだということを意味します。例えば、ファイルシステム、データベース、キューなどが該当します。災害対策戦略を策定する場合、組織では通常、目標復旧時間と目標復旧時点について計画を立てます。

目標復旧時間は、サービスの中断からサービスの復元までの最大許容遅延です。この目標は、サービスが利用できない時間枠としてどの程度を許容するかを決定するものであり、組織が定義します。

目標復旧時点は、最後のデータ復旧時点からの最大許容時間です。この目標は、最後の復旧時点からサービスの中断までの間にどの程度のデータ損失を許容するかを決定するものであり、組織が定義します。

詳細については、「AWS でのワークロードの災害対策: クラウド内での復旧」ホワイトペーパーを参照してください。

高可用性

このセクションでは、さまざまなコンピューティングオプションでの高可用性について詳しく見ていきます。

Amazon EKS は、高可用性を確保するために、複数のアベイラビリティーゾーンで Kubernetes コントロールおよびデータプレーンインスタンスを実行します。Amazon EKS は、異常なコントロールプレーンインスタンスを自動的に検出して置換し、これらのインスタンスのバージョンアップグレードやパッチ適用を自動的に行います。このコントロールプレーンは、リージョン内の 3 つのアベイラビリティーゾーンにまたがって実行される、少なくとも 2 つの API サーバーノードと 3 つの etcd ノードで構成されます。Amazon EKS では、AWS リージョン のアーキテクチャを使用して高可用性を維持します。

Amazon ECR は、高可用性と高パフォーマンスのアーキテクチャでイメージをホストし、複数のアベイラビリティーゾーンにわたってコンテナアプリケーションのイメージを確実にデプロイできるようにします。Amazon ECR は Amazon EKS、Amazon ECS、AWS Lambda と連携し、開発から本番稼動までのワークフローを簡素化します。

Amazon ECS は、AWS リージョン 内の複数のアベイラビリティーゾーンにわたって高可用性を維持しながらコンテナの実行を簡素化するリージョナルサービスです。Amazon ECS には、リソースのニーズ (CPU、RAM など) と可用性の要件に基づいて、クラスター全体にコンテナを配置する複数のスケジューリング戦略が含まれています。

AWS Lambda は、複数のアベイラビリティーゾーンで関数を実行することで、1 つのゾーンでサービスの中断が発生した場合にも、イベントを継続して処理できるようにします。お客様のアカウントで仮想プライベートクラウド (VPC) に接続するように関数を設定する場合は、複数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で、高可用性を確保します。

Lambda ベースのアプリケーションのデプロイ

AWS CloudFormation を使用して、サーバーレスアプリケーションを定義、デプロイ、設定できます。

AWS Serverless Application Model (AWS SAM) は、サーバーレスアプリケーションを定義する便利な方法です。AWS SAMは CloudFormation によってネイティブにサポートされ、サーバーレスリ

ソースを表現するための簡略化された構文を定義します。アプリケーションをデプロイするには、アプリケーションの一部として必要なリソースおよび関連する許可ポリシーを CloudFormation テンプレートで指定し、デプロイアーティファクトをパッケージ化して、そのテンプレートをデプロイします。SAM Local は、AWS SAM に基づく AWS Command Line Interface (AWS CLI) ツールであり、サーバーレスアプリケーションをローカルに開発、テスト、分析したうえで Lambda ランタイムにアップロードするための環境を提供します。AWS SAM Local を使用して、AWS ランタイム環境をシミュレートするローカルテスト環境を作成できます。

分散システムのコンポーネント

AWS によって個々のマイクロサービスに関連する課題を解決する方法を見てきました。ここからは、サービス検出、データ整合性、非同期通信、分散モニタリングおよび監査など、サービス間の課題に注目します。

トピック

- サービス検出
- 分散データ管理
- 設定管理
- 非同期通信と軽量メッセージング
- 分散モニタリング

サービス検出

マイクロサービスアーキテクチャの主な課題の 1 つとして、サービス相互間での検出とやり取りを可能にすることが挙げられます。分散型というマイクロサービスアーキテクチャの特性により、サービス間の通信が困難になるだけでなく、システムのヘルスチェックや、新しいアプリケーションが使用可能になったときの通知など、他の課題も生じます。アプリケーションで使用可能な設定データなどのメタストア情報を保存する方法と場所についても、決定する必要があります。このセクションでは、マイクロサービスベースのアーキテクチャについて、AWS でサービス検出を実施するための手法をいくつか紹介します。

DNS ベースのサービス検出

Amazon ECS には、コンテナ化されたサービス間で相互を簡単に検出して接続するための統合サービス検出が追加されました。

以前は、サービス間で相互を検出して接続できることを確認するために、Amazon Route 53、AWS Lambda、ECS Event Stream に基づく独自のサービス検出システムを設定して実行するか、すべてのサービスをロードバランサーに接続する必要がありました。

Amazon ECS は、Route 53 の Auto Naming API を利用し、サービス名のレジストリを作成して管理します。名前は自動的に DNS レコードセットにマッピングされるため、コード内で名前を使ってサービスを参照したり、DNS クエリを書いて実行時にサービスのエンドポイントを名前解決させ

たりできます。サービスのタスク定義でヘルスチェックの状況を指定できます。Amazon ECS は、サービスを探す時に正常なサービスエンドポイントだけを返します。

さらに、Kubernetes のマネージドサービスに対して統合サービス検出も使用できます。この統合を実現するため、AWS は Kubernetes インキュベータープロジェクトの 1 つである External DNS プロジェクトに貢献しました。

また、AWS Cloud Map の機能を使用する方法もあります。AWS Cloud Map は、Auto Naming API の機能を拡張するために、インターネットプロトコル (IP)、Uniform Resource Locator (URL)、Amazon リソースネーム (ARN) などのリソースのサービスレジストリを提供しています。さらに、変更を高速に伝播し、検出した一連のリソースを属性で絞り込む能力を備えた API ベースのサービス検出メカニズムも提供しています。既存の Route 53 の Auto Naming リソースは、自動的に AWS Cloud Map にアップグレードされます。

サードパーティー製ソフトウェア

サービス検出を実装するための別のアプローチは、HashiCorp Consul、etcd、Netflix Eureka などのサードパーティー製ソフトウェアを使用することです。これら 3 つの例はすべて、信頼性に優れた分散 key-value ストアです。HashiCorp Consul には AWS クイックスタートが用意されており、柔軟かつスケーラブルな AWS クラウド環境をセットアップし、HashiCorp Consul を任意の設定で自動的に起動できます。

サービスメッシュ

高度なマイクロサービスアーキテクチャの場合、実際のアプリケーションは数百、数千ものサービスで構成されていることがあります。多くの場合、アプリケーションの最も複雑な部分は、サービス自体ではなく、サービス間の通信です。サービスメッシュは、サービス間の通信を処理する追加のレイヤーであり、マイクロサービスアーキテクチャをモニタリングおよび制御する役割を担っています。これにより、このレイヤーでサービス検出などのタスクが完全に処理されます。

サービスメッシュは通常、データプレーンとコントロールプレーンに分離されています。インテリジェントプロキシのセットで構成されるデータプレーンは、マイクロサービス間のすべてのネットワーク通信を傍受する特別なサイドカープロキシとしてアプリケーションコードでデプロイされます。一方、コントロールプレーンはプロキシとのコミュニケーションを行います。

サービスメッシュは透過的であるため、アプリケーションの開発者はこの追加レイヤーを意識することもなく、既存のアプリケーションコードに変更を加えたりする必要もありません。AWS App Mesh は、アプリケーションレベルのネットワークを提供するサービスメッシュであり、複数のタ

イプのコンピューティングインフラストラクチャにわたってサービス間での相互通信を可能にします。App Mesh は、サービス間の通信方法を標準化して、完全な可視性を提供し、アプリケーションの高可用性を確保します。

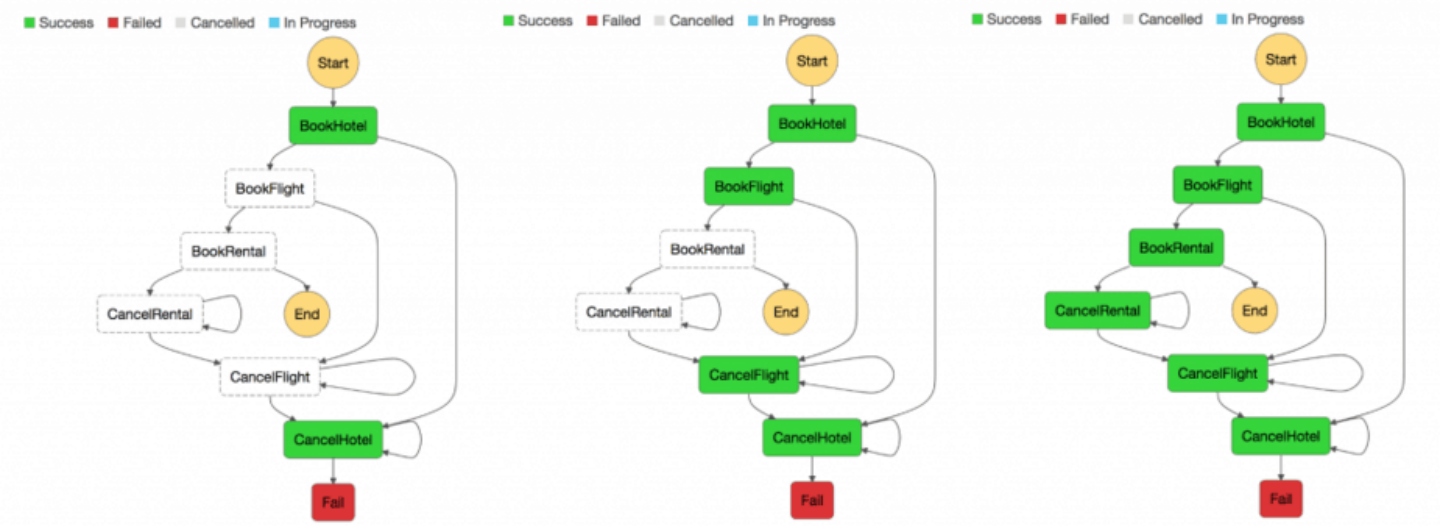
AWS App Mesh は、AWS Fargate、Amazon ECS、Amazon EKS、および AWS でのセルフマネージド Kubernetes で実行されている既存または新規のマイクロサービスで使用できます。App Mesh は、クラスター、オーケストレーションシステム、または VPC にわたって、コード変更なしで単一のアプリケーションとして実行されているマイクロサービスの通信をモニタリングおよび制御できます。

分散データ管理

モノリシックなアプリケーションは、通常、大規模なリレーショナルデータベースによってサポートされ、すべてのアプリケーションコンポーネントに共通のデータモデルが 1 つ定義されます。中央データベースのようなマイクロサービスアプローチでは、分散型の独立したコンポーネントを構築するという目標を達成できません。マイクロサービスの各コンポーネントに、データ永続レイヤーが必要です。

ただし、分散データ管理には新しい課題もあります。CAP 定理の結果として、分散マイクロサービスアーキテクチャでは、本質的にパフォーマンスを優先するため、整合性が犠牲になります。したがって、結果整合性を組み込む必要があります。

分散システムでは、ビジネストランザクションが複数のマイクロサービスに及ぶことがあります。複数のマイクロサービスで 1 つの ACID トランザクションを使用することはできないため、部分的な実行で終わる場合があります。このような場合、処理済みのトランザクションをやり直すためにコントロールロジックが必要になります。この目的のために、分散 Saga パターンが一般的に使用されます。ビジネストランザクションが失敗した場合、それまでのトランザクションでなされた変更を元に戻すために、Saga が一連の補正トランザクションをオーケストレートします。次の図に示すとおり、AWS Step Functions を使用すると、Saga 実行コーディネーターを簡単に実装できます。



Saga 実行コーディネーター

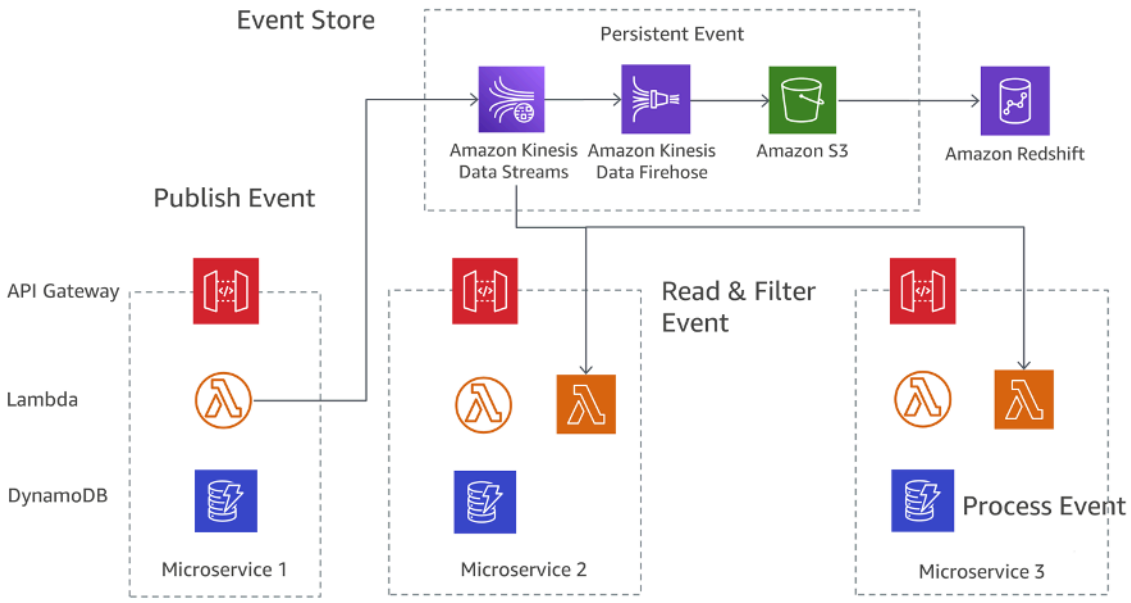
コアデータ管理ツールおよび手順でキュレートした重要なリファレンスデータの一元化されたストアを構築すると、マイクロサービスで重要なデータを同期し、さらに状態をロールバックできる場合があります。AWS Lambda を Amazon CloudWatch Events のスケジュールと組み合わせて使用すれば、クリーンアップおよび重複排除のシンプルなメカニズムを構築できます。

状態の変化が複数のマイクロサービスに影響を与えることがよくあります。このような場合、イベントソーシングが役立つパターンであることがわかっています。イベントソーシングの背景にある主な考え方は、アプリケーションの変更をすべてイベントレコードとして表現し、保持することです。アプリケーションの状態を保持する代わりに、データがイベントのストリームとして保存されます。データベーストランザクションのログ記録とバージョン管理のシステムは、よく知られているイベントソーシングの 2 つの例です。イベントソーシングにはいくつかの利点があります。任意の時点で状態の判断や再構築を行うことができます。また、永続的な監査証跡が自動的に生成され、デバッグも容易になります。

マイクロサービスアーキテクチャのコンテキストでは、イベントソーシングを通じてパブリッシュ/サブスクライブパターンを使用することで、アプリケーションのさまざまなパーツのデカップリングを行うことができます。また、同一のイベントデータをさまざまなデータモデルに提供してマイクロサービスを分離できます。イベントソーシングは、コマンド、クエリ、責任、分離 (CQRS) パターンと組み合わせて使用されることが多く、書き込みのワークロードから読み取りをデカップリングして、両方のパフォーマンス、スケーラビリティ、セキュリティを最適化します。従来のデータ管理システムでは、コマンドとクエリは同じデータリポジトリに対して実行されます。

次の図は、AWS でイベントソーシングパターンをどのように実装できるかを示しています。Amazon Kinesis Data Streams は、中央イベントストアという主要なコンポーネントとして

機能し、アプリケーションの変更をイベントとしてキャプチャし、Amazon S3 に保存します。図は、Amazon API Gateway、AWS Lambda、Amazon DynamoDB で構成される 3 つの異なるマイクロサービスを示しています。矢印は、イベントの流れを示しています。マイクロサービス 1 は、イベントの状態の変化を検出すると、Kinesis Data Streams にメッセージを書き込むことで、イベントを発行します。すべてのマイクロサービスは、AWS Lambda で独自の Kinesis Data Streams アプリケーションを実行しています。これは、メッセージのコピーを読み取ってマイクロサービスとの関連性に基づいてフィルタリングし、必要に応じて次の処理のためにメッセージを転送します。関数からエラーが返されると、Lambda は、処理が成功するか、データの有効期限が切れるまでバッチを再試行します。シャードの停止を避けるには、より小さいバッチサイズで再試行するか、再試行数を制限するか、古すぎるレコードを破棄するように、イベントソースマッピングを設定できます。破棄したイベントを保持するには、失敗したバッチに関する詳細を Amazon Simple Queue Service (Amazon SQS) キューまたは Amazon Simple Notification Service (Amazon SNS) トピックに送信するように、イベントソースマッピングを設定できます。



AWS でのイベントソーシングパターン

Amazon S3 は、すべてのマイクロサービスにわたってすべてのイベントを永続的に保存し、アプリケーションの状態をデバッグまたは復旧したり、アプリケーションの変更を監査したりするときに、信頼性の高い単一の情報源となります。レコードは Kinesis Data Streams アプリケーションに複数回配信される場合があります。この理由として、プロデューサーの再試行とコンシューマーの再試行の 2 つがあります。アプリケーションは、各レコードの複数回処理を予測して適切に処理する必要があります。

設定管理

数十のさまざまなサービスで構成される一般的なマイクロサービスアーキテクチャの場合、各サービスは、データをサービスに公開する複数の下流サービスやインフラストラクチャコンポーネントにアクセスする必要があります。例としては、メッセージキュー、データベース、その他のマイクロサービスがあります。重要な課題の 1 つは、下流サービスやインフラストラクチャへの接続に関する情報を一貫した方法で提供するように各サービスを設定することです。また、設定にはサービスが動作している環境に関する情報も含める必要があります。アプリケーションを再起動しなくても新しい設定データを使用できるようにします。

Twelve-Factor App パターンの 3 番目の原則は、次のトピックをカバーしています:「Twelve-Factor App は設定を環境変数に格納する (多くの場合、env vars または env と略記される)」 Amazon ECS では、docker run の `--env` オプションにマッピングされる環境コンテナの定義パラメータを使用して、環境変数をコンテナに渡すことができます。環境変数を一括してコンテナに渡すには、`environmentFiles` コンテナの定義パラメータを使用して、環境変数を含む 1 つ以上のファイルをリストします。ファイルは Amazon S3 でホストされている必要があります。AWS Lambda の場合、ランタイムは、環境変数をコードで使用できるようにし、関数と呼び出しリクエストに関する情報を含む追加の環境変数を設定します。Amazon EKS の場合は、対応するポッドの設定マニフェストの env フィールドで環境変数を定義できます。env 変数を使用する別の方法としては、ConfigMap を使用します。

非同期通信と軽量メッセージング

従来のモノリシックなアプリケーションの通信は非常に明快です。メソッド呼び出しまたは内部のイベント分散メカニズムを使用すれば、アプリケーションのパーツ間で相互に通信できます。同じアプリケーションをデカップリングされたマイクロサービスを使用して実装する場合、アプリケーションのパーツ間の通信は、ネットワーク通信を使用して実装する必要があります。

REST ベースの通信

HTTP/S プロトコルは、マイクロサービス間の同期通信を実装するための最も一般的な方法です。ほとんどの場合、RESTful API では HTTP が転送レイヤーとして使用されます。REST アーキテクチャスタイルでは、ステートレスな通信、統一されたインターフェイス、標準メソッドが使用されます。

API Gateway の場合は、アプリケーションの玄関口として機能する API を作成し、バックエンドサービスからデータ、ビジネスロジック、機能にアクセスできます。API デベロッパーは、AWS や他のウェブサービスにアクセスする API を作成し、さらに AWS クラウド内に保存されているデータ

にアクセスする API を作成できます。API Gateway サービスを使って定義された API オブジェクトはリソースとメソッドのグループです。

リソースは API のドメイン内にある型付きオブジェクトで、他のリソースにデータモデルまたはリレーションシップを関連付けている場合があります。各リソースは、1 つまたは複数のメソッド、つまり、GET、POST、PUT といった標準的な HTTP 動詞に応答するように設定できます。REST API は、さまざまなステージへのデプロイ、バージョニング、新しいバージョンへの複製が可能です。

API Gateway は、トラフィック管理、認可とアクセスコントロール、モニタリング、API のバージョン管理など、数十万件までの同時 API コールの受信と処理に伴うすべてのタスクに対応します。

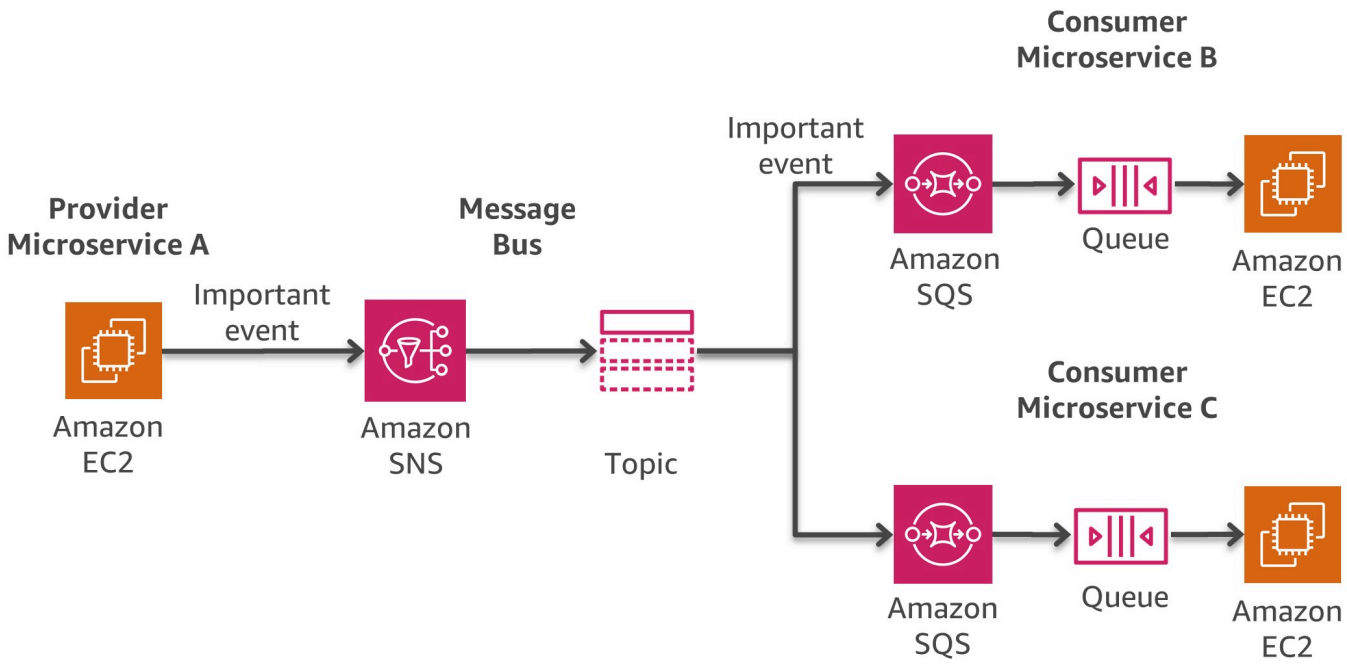
非同期メッセージングとイベントパッシング

マイクロサービス間の通信を実装するための別のパターンとして、メッセージパッシングがあります。サービス間の通信には、キューを使用してメッセージを交換します。この通信方法の主な利点の 1 つは、サービス検出を行う必要がなく、サービスは疎結合されることです。

同期システムは密結合されるため、同期ダウンストリームの依存関係に問題があると、アップストリームの呼び出し元にただちに影響が及びます。アップストリームの呼び出し元による再試行はすぐにファンアウトし、問題を増幅する場合があります。

AWS は、プロトコルなどの特定の要件に応じ、このパターンの実装に役立つさまざまなサービスを提供しています。1 つの実装方法として、Amazon Simple Queue Service (Amazon SQS) キューや Amazon Simple Notification Service (Amazon SNS) の組み合わせを使用できます。

両サービスは密接に連携しています。Amazon SNS を使用することで、プッシュメカニズムを通じてアプリケーションから複数のサブスクライバーにメッセージを送信できます。Amazon SNS と Amazon SQS を組み合わせて使用することで、1 つのメッセージを複数のコンシューマーに配信できます。次の図は、Amazon SNS と Amazon SQS の統合を示しています。



AWS でのメッセージバスパターン

Amazon SQS キューを SNS トピックにサブスクライブすると、そのトピックにメッセージを発行できるようになり、サブスクライブした SQS キューに Amazon SNS からメッセージが送信されます。メッセージには、トピックに発行した件名とメッセージ、および JSON 形式のメタデータ情報が含まれています。

内部アプリケーション、サードパーティーの SaaS アプリケーション、AWS のサービスにまたがるイベントソースを使用してイベント駆動型アーキテクチャを大規模に構築する別のオプションとして、Amazon EventBridge があります。フルマネージドのイベントバスサービスである EventBridge は、さまざまなソースからイベントを受信して、ルーティングルールに基づいてターゲットを識別し、AWS Lambda、Amazon SNS、Amazon Kinesis Streams などのターゲットにほぼリアルタイムでデータを配信します。受信イベントは、配信に先立って入力トランスフォーマーでカスタマイズすることもできます。

イベント駆動型アプリケーションの開発を大幅に高速化するために、EventBridge スキーマレジストリはスキーマを収集して整理します。これには、AWS のサービスで生成されたすべてのイベントのスキーマも含まれます。また、カスタムスキーマを定義したり、推論スキーマオプションを使用してスキーマを自動的に検出したりすることもできます。ただし、これらすべての機能の潜在的なトレードオフは、EventBridge 配信のレイテンシー値が比較的高くなることです。また、EventBridge のデ

フォルトのスループットとクォータは、ユースケースに応じ、サポートリクエストを通じて増やす必要が生じる場合があります。

Amazon MQ に基づく別の実装戦略があります。この戦略は、既存のソフトウェアがオープンスタンダードの API と、メッセージングのプロトコルとして JMS、NMS、AMQP、STOMP、MQTT、WebSocket などを使用している場合に利用できます。Amazon SQS はカスタム API を公開します。つまり、既存のアプリケーションをオンプレミス環境などから AWS に移行する場合は、コードの変更が必要になります。Amazon MQ では、多くの場合、この作業が不要になります。

Amazon MQ は、人気の高いオープンソースメッセージブローカーである ActiveMQ の管理とメンテナンスを行います。基盤となるインフラストラクチャは、自動的にプロビジョニングされて高可用性とメッセージの耐久性を確保し、アプリケーションの信頼性をサポートします。

オーケストレーションとステート管理

マイクロサービスの分散型の特性により、複数のマイクロサービスが関与するワークフローをオーケストレートするのは困難を伴います。デベロッパーは、オーケストレーションコードをサービスに直接追加したくなるかもしれません。これは避けてください。密結合が生じて個々のサービスを迅速に置き換えることが困難になるためです。

AWS Step Functions を使用すると、それぞれが別個の機能を実行する独立したコンポーネントでアプリケーションを構築できます。Step Functions が提供するステートマシンは、エラー処理、シリアル化、並列化など、サービスオーケストレーションの複雑さを隠します。これにより、サービス内に調整用のコードを追加することなく、アプリケーションのスケールや変更を簡単に行うことができます。

Step Functions を使用すれば、確実にコンポーネントを調整し、アプリケーションの機能を配置できます。また、グラフィカルコンソールを使用して、アプリケーションのコンポーネントを一連のステップとして配置して可視化できます。これにより、分散サービスを簡単に構築して実行できるようになります。

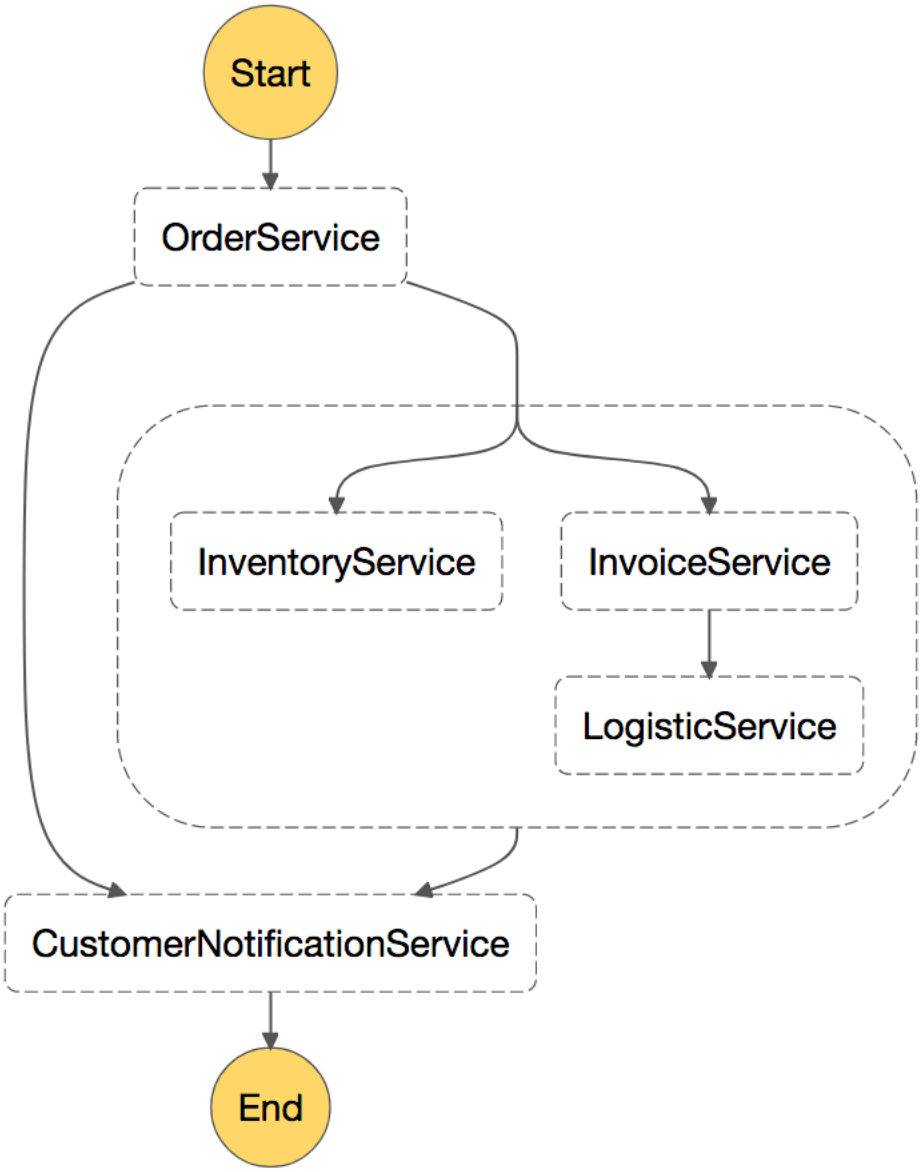
Step Functions は、自動的に各ステップを開始して追跡し、エラーが発生した場合は再試行するため、アプリケーションは意図したとおりに整然と実行されます。また、各ステップの状態が記録されるため、問題が発生した場合は診断とデバッグをすぐに実行できます。コードを記述することなくステップを変更、追加できるため、アプリケーションの進化とイノベーションが高速化します。

Step Functions は AWS サーバーレスプラットフォームの一部であり、Lambda 関数や、コンピューティングリソースに基づくアプリケーション (Amazon EC2、Amazon EKS、Amazon ECS など)、

追加のサービス (Amazon SageMaker や AWS Glue など) のオーケストレーションをサポートします。Step Functions は、オペレーションや基盤となるインフラストラクチャを管理し、どのような規模でもアプリケーションを確実に使用できるようにします。

Step Functions は、ワークフローを構築するために Amazon ステートメント言語を使用します。ワークフローには、シーケンシャルステップ、パラレルステップ、分岐ステップを含めることができます。

次の図は、シーケンシャルステップとパラレルステップを組み合わせたマイクロサービスアーキテクチャのワークフロー例を示しています。このようなワークフローの呼び出しには、Step Functions API または API Gateway を使用できます。



Step Functions によるマイクロサービスワークフローの呼び出しの例

分散モニタリング

マイクロサービスアーキテクチャは、多数の分散型のパーツで構成されており、モニタリングが必要です。Amazon CloudWatch を使用すると、メトリクスの収集と追跡、ログファイルの一元管理とモニタリング、アラームの設定を行い、AWS 環境内の変更に対して自動的に対応できます。Amazon CloudWatch は、AWS リソース (Amazon EC2 インスタンス、Amazon DynamoDB テーブル、Amazon RDS DB インスタンスなど)、アプリケーションやサービスが生成するカスタムメトリクス、アプリケーションが生成するあらゆるログファイルをモニタリングできます。

モニタリング

CloudWatch を使用して、リソース使用率、アプリケーションパフォーマンス、オペレーション状態についてシステム全体の可視性を得ることができます。ほんの数分で使用を開始できる CloudWatch は、信頼性、拡張性、および柔軟性あるモニタリングソリューションを提供します。独自のモニタリングシステムやインフラストラクチャをセットアップ、管理、スケールする必要がなくなります。またさらなる恩恵として、マイクロサービスアーキテクチャでは、CloudWatch を使用したカスタムメトリクスのモニタリング機能により、開発者がサービスごとに収集するメトリクスを決められます。さらに、カスタムメトリクスに基づいて 動的スケーリングを実装できます。

Amazon CloudWatch に加えて CloudWath Container Insights も使用することで、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のメトリクスとログを収集、集計、要約できます。CloudWatch Container Insights は、CPU、メモリ、ディスク、ネットワークなどの多くのリソースのメトリクスを自動的に収集し、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約します。CloudWatch Container Insights を使用すると、CloudWatch Container Insights ダッシュボードのメトリクスにアクセスできます。また、コンテナの再起動障害などの診断情報を活用して、問題をすばやく切り分けて解決できます。Container Insights が収集するメトリクスに CloudWatch アラームを設定することもできます。

Container Insights は、Amazon EC2 上の Amazon ECS、Amazon EKS、Kubernetes の各プラットフォームで利用できます。Amazon ECS サポートには、Fargate のサポートが含まれています。

別の人気の高いオプション (特に Amazon EKS で) は、Prometheus を使用することです。Prometheus はオープンソースのモニタリングおよびアラートツールキットです。多くの場合、収集したメトリクスを可視化するために Grafana と組み合わせて使用されます。Kubernetes の多くのコンポーネントは、`/metrics` にメトリクスを保存します。Prometheus は、これらのメトリクスを定期的に取得できます。

Amazon Managed Service for Prometheus (AMP) は、Prometheus と互換性があるモニタリングサービスであり、コンテナ化されたアプリケーションの大規模なモニタリングを可能にします。AMP を使用すると、オープンソースの Prometheus クエリ言語 (PromQL) を使用して、コンテナ化されたワークロードのパフォーマンスをモニタリングできます。運用メトリクスの取り込み、保存、クエリを管理するための基盤のインフラストラクチャを管理する必要はありません。Prometheus のメトリクスを Amazon EKS 環境や Amazon ECS 環境から収集するには、AWS Distro for OpenTelemetry または Prometheus サーバーを収集エージェントとして使用できます。

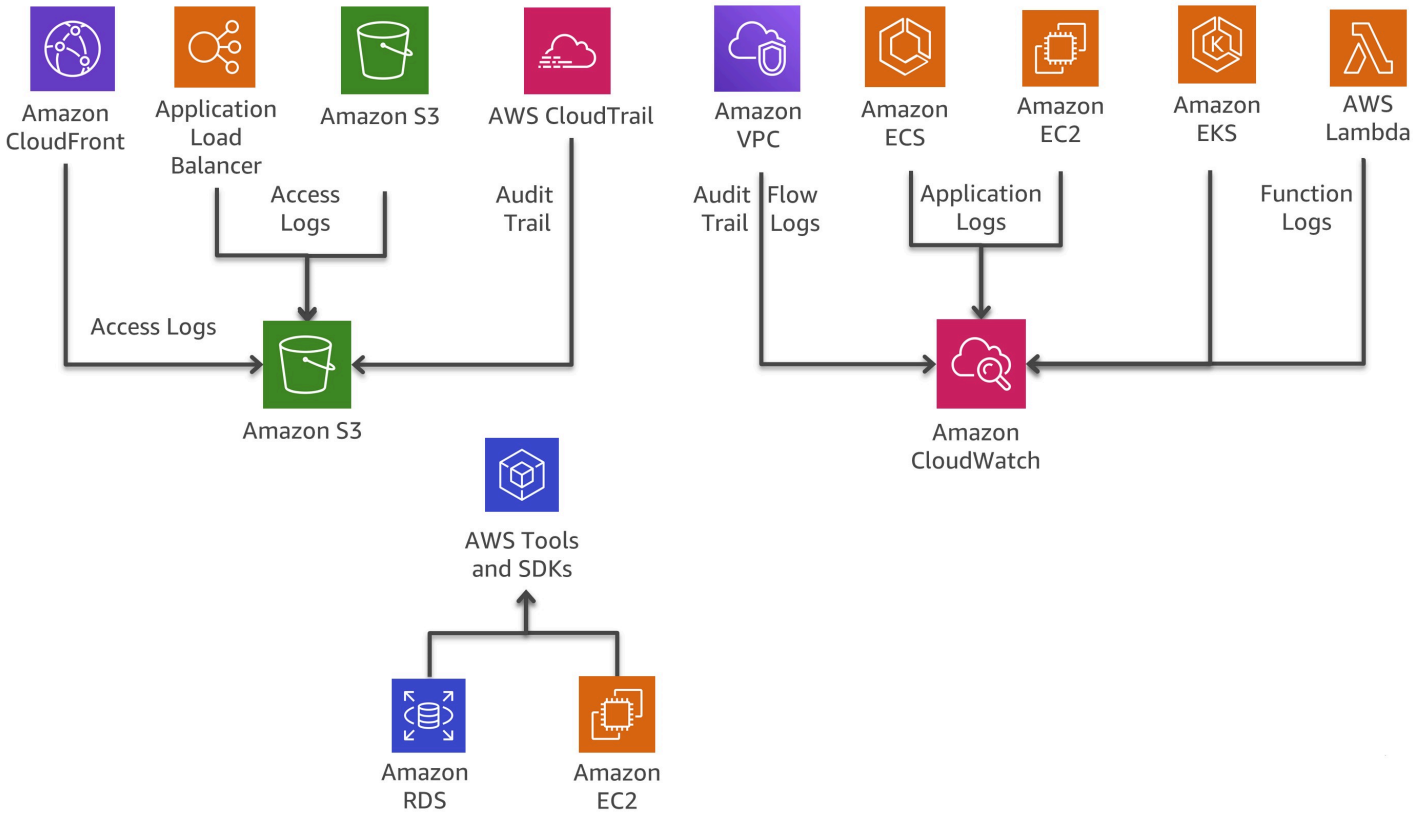
AMP は、多くの場合、Amazon Managed Service for Grafana (AMG) と組み合わせて使用されます。AMG を使用すると、メトリクスの保存場所に関係なく、メトリクスのクエリ、可視化、アラート、理解が簡単になります。AMG を使用することで、サーバーのプロビジョニング、ソフトウェアの設定や更新、または本番環境での Grafana の保護とスケーリングに伴う面倒な作業を回避して、メトリクス、ログ、トレースを分析できます。

ログの一元管理

トラブルシューティングと問題の検出には、一貫したログ記録が不可欠です。マイクロサービスを使用すると、チームはこれまでよりも多くのリリースを市場投入することができます。また、エンジニアリングチームは本番環境で新機能を実験できます。アプリケーションを徐々に改善するには、お客様への影響を理解することが不可欠です。

AWS のサービスのほとんどは、デフォルトでログファイルを一元管理します。AWS でのログファイルの主な送信先は、Amazon S3 および Amazon CloudWatch Logs です。Amazon EC2 インスタンスで実行しているアプリケーションの場合は、デーモンを使用してログファイルを CloudWatch Logs に送信できます。Lambda 関数はネイティブにログ出力を CloudWatch Logs に送信します。また、Amazon ECS には awslogs ログドライバーのサポートが含まれているため、コンテナログを CloudWatch Logs で一元管理できます。Amazon EKS の場合は、Fluent Bit または Fluentd を使用してクラスター内の個々のインスタンスから一元化されたログ記録先の CloudWatch Logs にログを転送できます。ここで、ログは Amazon OpenSearch Service と Kibana で結合され、より高レベルのレポートが作成されます。Fluent Bit は、フットプリントが小さく、パフォーマンス上の利点があるため、FluentD の代わりに使用することをお勧めします。

次の図は、一部のサービスのログ記録機能を示しています。チームは、Amazon OpenSearch Service や Kibana などのツールを使用して、これらのログを検索および分析できます。Amazon Athena を使用すると、Amazon S3 に一元化されたログファイルに対して 1 回限りのクエリを実行できます。



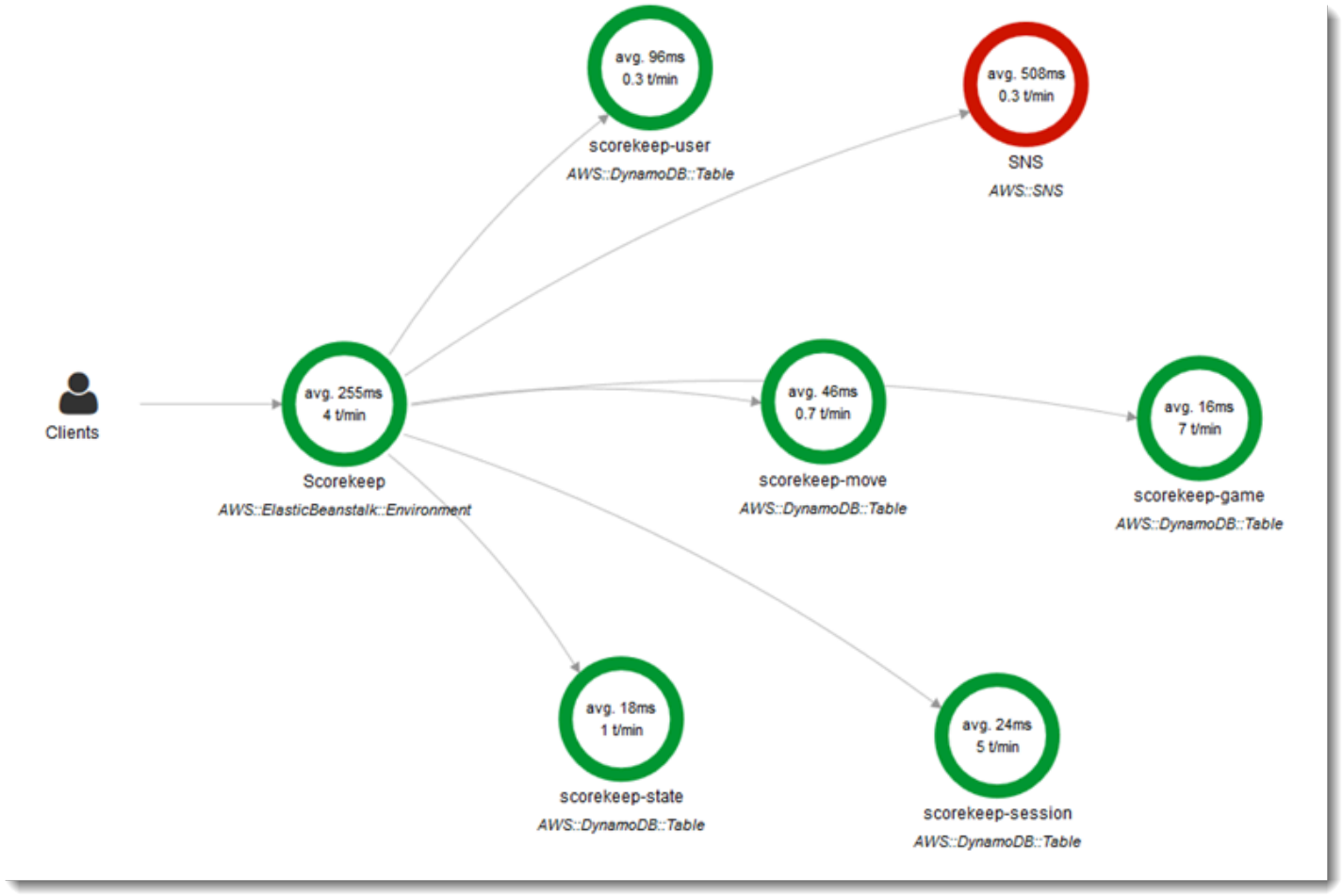
AWS のサービスのログ記録機能

分散トレース

多くの場合、一連のマイクロサービスは連携してリクエストを処理します。コールチェーン内のサービスの一つにエラーが発生する、数十のマイクロサービスで構成される複雑なシステムを想像してください。すべてのマイクロサービスでログが適切に記録され、集中システムに統合されていても、関連するすべてのログメッセージを見つけるのは非常に困難です。

AWS X-Ray の主旨は相関 ID を使用することにあります。相関 ID は、特定のイベントチェーンに関連するすべてのリクエストやメッセージにアタッチされる一意の識別子です。トレース ID は、リクエストが最初の X-Ray 統合サービス (Application Load Balancer や API Gateway など) に到達すると、`X-Amzn-Trace-Id` という特定のトレースヘッダーの HTTP リクエストに追加され、レスポンスに含められます。X-Ray SDK を使用すると、あらゆるマイクロサービスがこのヘッダーの読み取りだけでなく、追加や更新もできます。

X-Ray は、Amazon EC2、Amazon ECS、Lambda、AWS Elastic Beanstalk と連携します。X-Ray は、これらのサービスにデプロイされている Java、Node.js、.NET で記述したアプリケーションで使用できます。



AWS X-Ray サービスマップ

Epsagon は、すべての AWS のサービス、サードパーティーの API (HTTP 呼び出し経由)、他の一般的なサービス (Redis、Kafka、Elastic など) のトレースを含むフルマネージド SaaS です。Epsagon サービスには、モニタリング機能、代表的なサービスへのアラート、コードからのすべての呼び出しのペイロードの可視性が含まれます。

AWS Distro for OpenTelemetry は、OpenTelemetry プロジェクトのセキュアで本番環境に対応した AWS サポートのディストリビューションです。AWS Distro for Open Telemetry は、Cloud Native Computing Foundation の一部であり、アプリケーションモニタリング用の分散トレースやメトリクスを収集するためのオープンソース API、ライブラリ、エージェントを提供します。AWS Distro for OpenTelemetry を使用すると、アプリケーションを 1 回だけインストルメント化するだけで、相関するメトリクスやトレースを AWS やパートナーの複数のモニタリングソリューションに送信できます。コードを変更せずにトレースを収集するには、自動インストルメンテーションエージェントを使用します。また、AWS Distro for OpenTelemetry は、AWS リソースやマネージドサービスからメタデータも収集し、アプリケーションのパフォーマンスデータと基盤のインフラストラクチャデータ

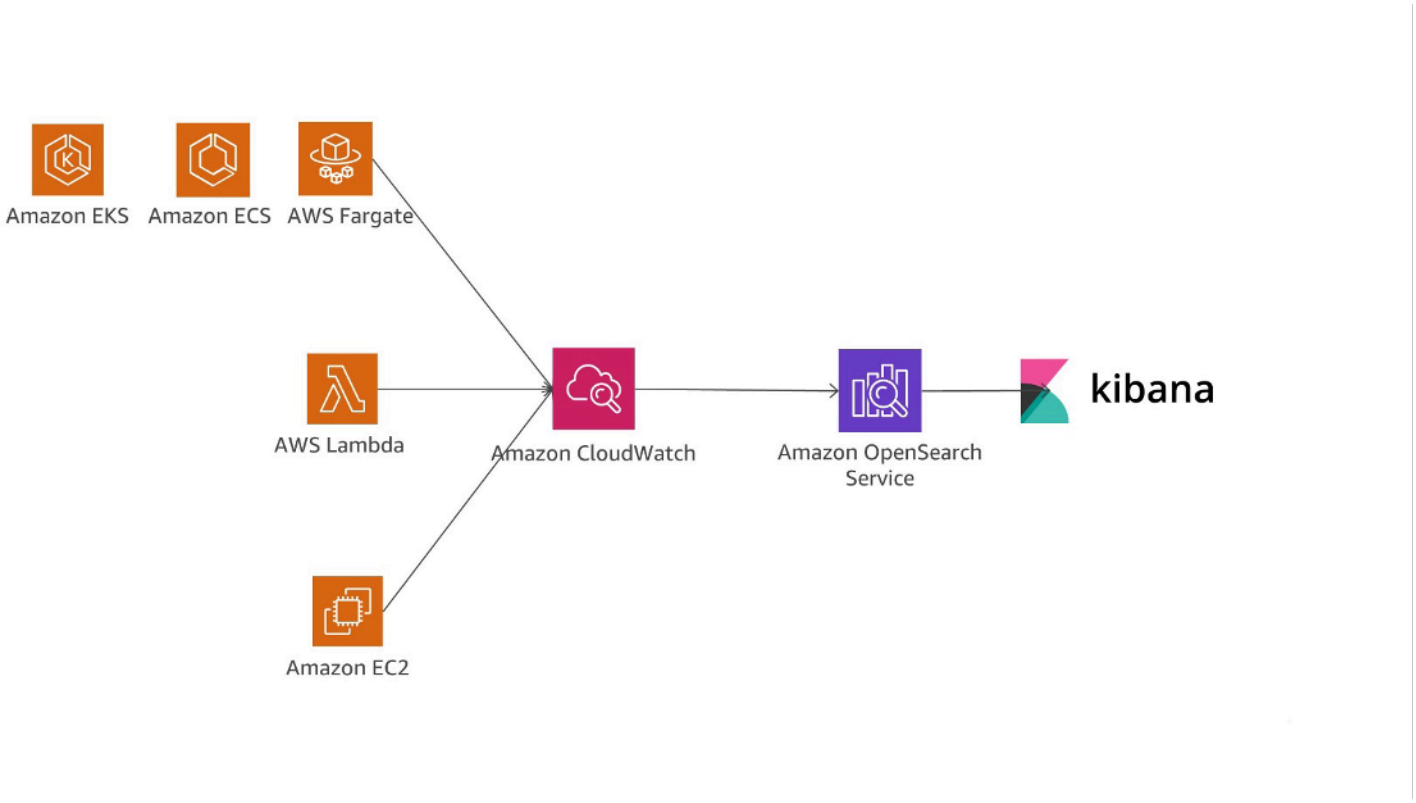
を相関させて、問題解決までの平均時間を短縮します。AWS Distro for OpenTelemetry を使用すると、Amazon EC2、Amazon EC2、Amazon EC2 での Amazon EKS、Fargate、AWS Lambda、およびオンプレミスで実行されているアプリケーションをインストルメント化できます。

AWS でのログ分析のオプション

ログデータの検索、分析、可視化は、分散システムを理解するうえで重要な要素です。Amazon CloudWatch Logs Insights を使用すると、ログを即座に検索、分析、可視化できます。これにより、運用上の問題のトラブルシューティングが可能になります。ログファイルを分析する別のオプションは、Amazon OpenSearch Service を Kibana と組み合わせて使用することです。

Amazon OpenSearch Service では、全文検索、構造化検索、分析、およびこれら 3 つを組み合わせて実行できます。Kibana は、Amazon OpenSearch Service とシームレスに統合するオープンソースのデータ可視化プラグインです。

次の図は、Amazon OpenSearch Service と Kibana を使用したログ分析を示しています。CloudWatch Logs は、CloudWatch Logs のサブスクリプションを通じて、Amazon OpenSearch Service にログエントリをほぼリアルタイムでストリーミングするように設定できます。Kibana はデータを可視化し、Amazon OpenSearch Service のデータストアに便利な検索インターフェイスを公開します。このソリューションを ElastAlert などのソフトウェアと組み合わせて使用することで、データに異常、スパイク、その他の該当パターンが検出された場合に、SNS 通知や E メールを送信したり、JIRA チケットを作成したりするためのアラートシステムを実装できます。



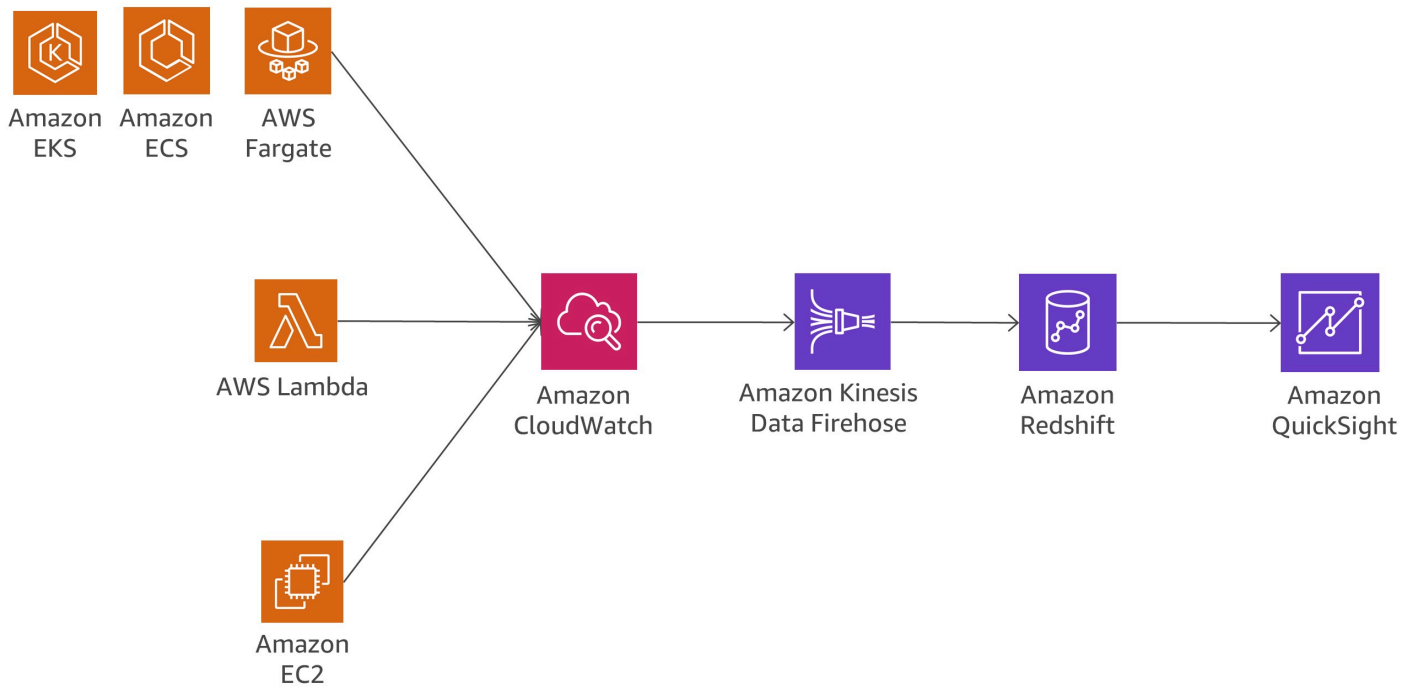
Amazon OpenSearch Service と Kibana によるログ分析

ログファイルを分析するための別のオプションとして、Amazon Redshift と Amazon QuickSight を組み合わせて使用することもできます。

QuickSight は、Amazon Redshift、Amazon RDS、Amazon Aurora、Amazon EMR、DynamoDB、Amazon S3、Amazon Kinesis などの AWS データサービスに簡単に接続できます。

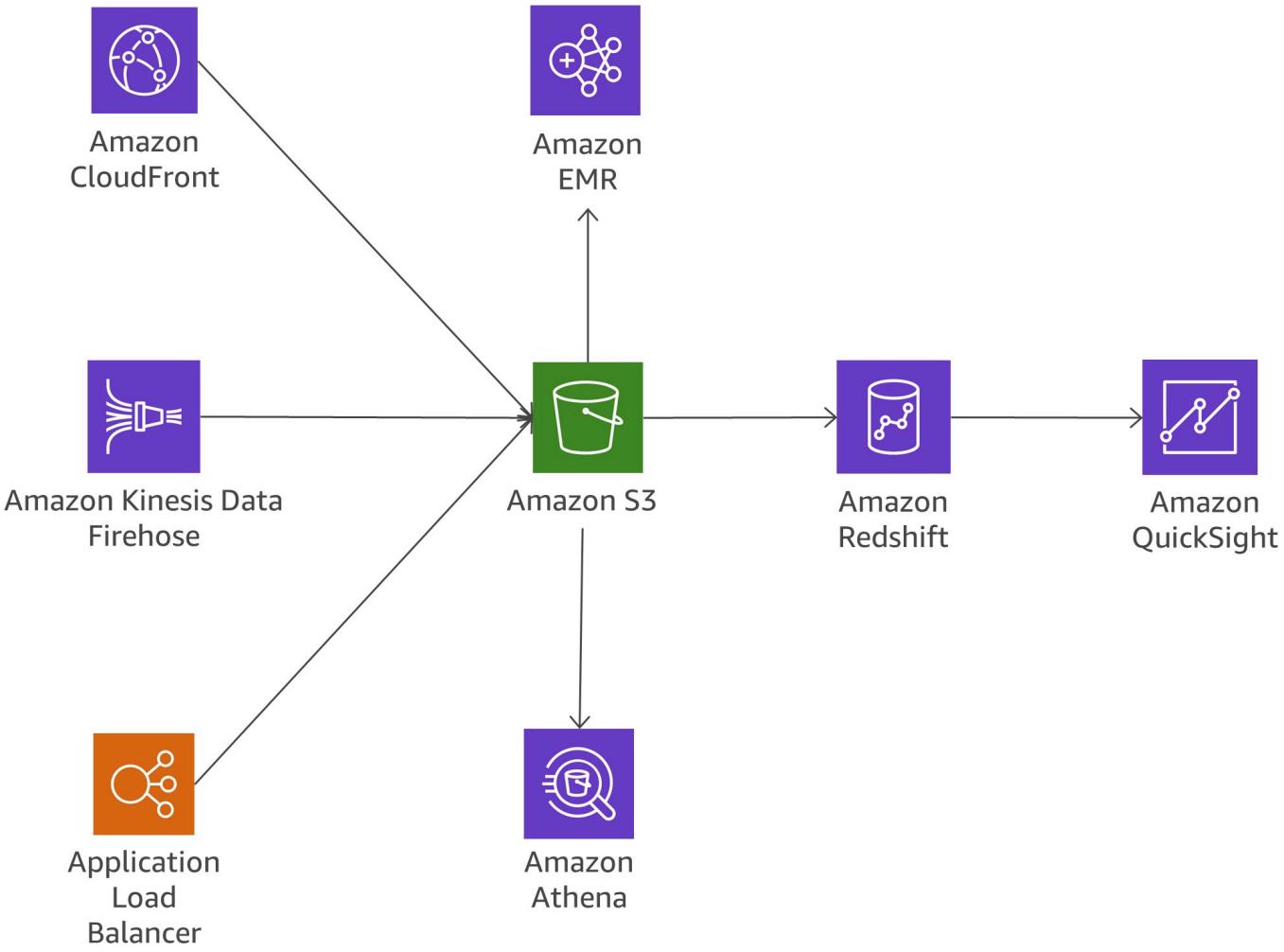
CloudWatch Logs は、ログデータの一元化されたストアとして機能しますが、データを保存するだけでなく、Amazon Kinesis Data Firehose にログエントリをストリーミングすることもできます。

次の図は、CloudWatch Logs と Kinesis Data Firehose を使用して、ログエントリをさまざまなソースから Amazon Redshift にストリーミングするシナリオを示しています。Amazon QuickSight は、Amazon Redshift に保存されたデータを使用して分析、レポート、可視化を行います。



Amazon Redshift および Amazon QuickSight によるログ分析

次の図は、Amazon S3 でのログ分析のシナリオを示しています。ログを Amazon S3 バケットに保存すると、ログデータを Amazon Redshift や Amazon EMR などのさまざまな AWS データサービスにロードして、ログストリームに保存されているデータを分析して異常を検出できます。



Amazon S3 でのログ分析

チャティネス (対話頻度)

モノリシックなアプリケーションを小さなマイクロサービスに分割すると、マイクロサービス同士でやりとりする必要が生じるため、通信オーバーヘッドが増加します。多くの実装で、通信プロトコルとして REST over HTTP を使用しています。このプロトコルは軽量であるためですが、メッセージの量が多いと問題が発生する可能性があります。場合によっては、大量のメッセージを送受信するサービスを統合することを検討できます。チャティネス (対話頻度) を削減するためだけにますます多くのサービスを統合するような状況になった場合は、問題ドメインとドメインモデルを見直す必要があります。

プロトコル

このホワイトペーパーの前半にある「the section called “非同期通信と軽量メッセージング”」セクションで、さまざまなプロトコルの可能性について説明しています。マイクロサービスでは、HTTP などの単純なプロトコルを使用することが一般的です。サービスによって交換されるメッセージは、JSON や YAML といった人が読み取り可能な形式、または Avro やプロトコルバッファなどの効率的なバイナリ形式など、さまざまな方法でエンコードできます。

キャッシュ

キャッシュは、マイクロサービスアーキテクチャのレイテンシーとチャティネス (対話頻度) の低減に最適です。実際のユースケースとボトルネックに応じて、いくつかのキャッシュレイヤーを使用できます。AWS で実行されている多くのマイクロサービスアプリケーションは、ElastiCache を使用して結果をローカルにキャッシュすることで、他のマイクロサービスへの呼び出し回数を削減しています。API Gateway では、組み込みのキャッシュレイヤーによってバックエンドサーバーの負荷を軽減します。さらに、キャッシュはデータ永続レイヤーからの負荷軽減にも役立ちます。すべてのキャッシュメカニズムに伴う課題は、良好なキャッシュヒット率と、データの適時性/整合性の適正なバランスを見つけることにあります。

監査

分散サービスが数百にも及ぶ可能性があるマイクロサービスアーキテクチャでは、もう 1 つの対処すべき課題として、各サービスでユーザーアクションの可視性を確保し、組織レベルですべてのサービスの全体像をよく把握できるようにする必要があります。セキュリティポリシーの適用を円滑にするには、リソースへのアクセスと、システム変更につながるアクティビティの両方を監査することが重要です。

変更は、個別のサービスレベルと、より広いシステムで実行されるサービス全体で追跡する必要があります。変更は通常マイクロサービスアーキテクチャで頻繁に発生し、それにより監査の変更がさらに重要になります。このセクションでは、マイクロサービスアーキテクチャの監査に役立つ AWS の主なサービスと機能に注目します。

監査証跡

AWS CloudTrail は、マイクロサービスでの変化を追跡するために役立つツールです。このツールでは、AWS クラウド 内で実行されたすべての API コールをログに記録し、これらのログをリアルタイムで CloudWatch Logs に送信したり、数分以内に Amazon S3 に送信したりできます。

すべてのユーザーや自動システムのアクションが検索可能になり、これらを分析して予期しない動作や企業ポリシー違反を検出したり、デバッグしたりできます。記録される情報には、タイムスタ

ンプ、ユーザー情報、アカウント情報、呼び出されたサービス、リクエストされたサービスアクション、呼び出し元の IP アドレス、リクエストのパラメータとレスポンス要素が含まれます。

CloudTrail では、同じアカウントに対して複数の証跡を定義することが許可されます。これにより、セキュリティ管理者、ソフトウェアデベロッパー、IT 監査担当者など、さまざまなステークホルダーが独自の証跡を作成および管理できます。マイクロサービスチームごとに AWS アカウントが異なる場合は、証跡を単一の S3 バケットに集約することができます。

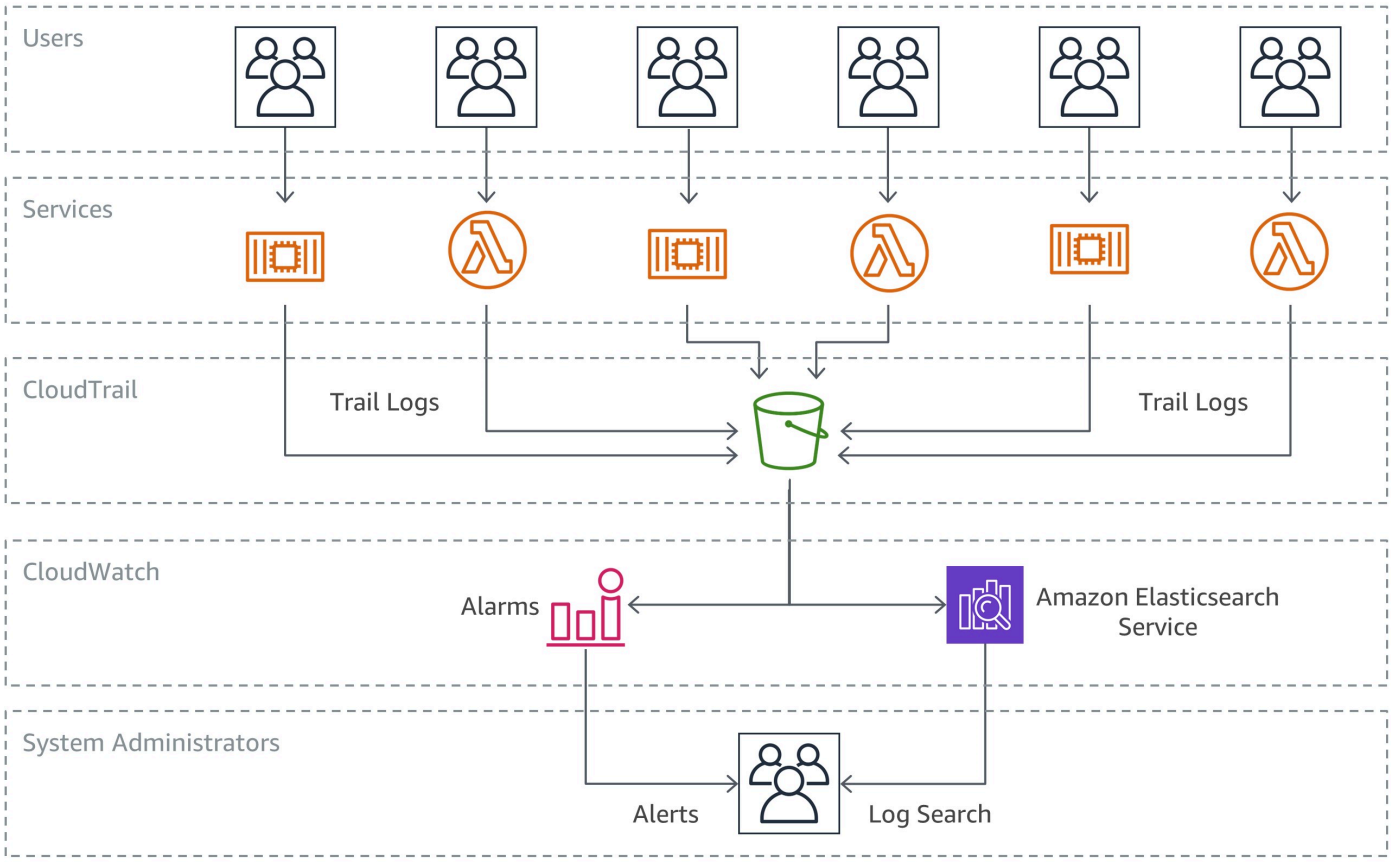
監査証跡を CloudWatch に保存する利点は、証跡データがリアルタイムでキャプチャされること、情報を Amazon OpenSearch Service に再ルーティングすることで検索と可視化が容易になることです。Amazon S3 と CloudWatch Logs の両方に口グを記録するように CloudTrail を設定できます。

イベントとリアルタイムアクション

システムアーキテクチャの特定の変更には迅速な対応が必要であり、状況を修復するための措置を講じるか、変更を承認するための特定のガバナンス手順を開始する必要があります。Amazon CloudWatch Events と CloudTrail の統合により、すべての AWS のサービスにわたってすべての API コールの変化に対するイベントを生成できるようになります。カスタムイベントの定義や、固定スケジュールに基づくイベントの生成も可能になります。

イベントが発生し、これが定義済みのルールに一致すると、組織内の定義済みのグループは即座に通知を受け取り、必要なアクションを実行できます。必要なアクションを自動化できる場合、ルールは自動的に内蔵ワークフローをトリガーするか、Lambda 関数を呼び出して問題を解決できます。

次の図に示す環境では、CloudTrail と CloudWatch Events が連携してマイクロサービスアーキテクチャ内の監査と修復の要件に対処しています。すべてのマイクロサービスは CloudTrail によって追跡され、監査証跡が Amazon S3 バケットに保存されます。CloudWatch Events が発生すると、運用上の変更が認識されます。CloudWatch Events は、オペレーションの変更に応答し、必要に応じて、応答メッセージを環境に送り、機能をアクティブ化し、変更を行い、状態情報を収集することによって、是正措置を実行します。CloudWatch Events は、CloudTrail 上に配置され、アーキテクチャに特定の変更が行われると、アラートをトリガーします。



監査と修復

リソースインベントリと変更管理

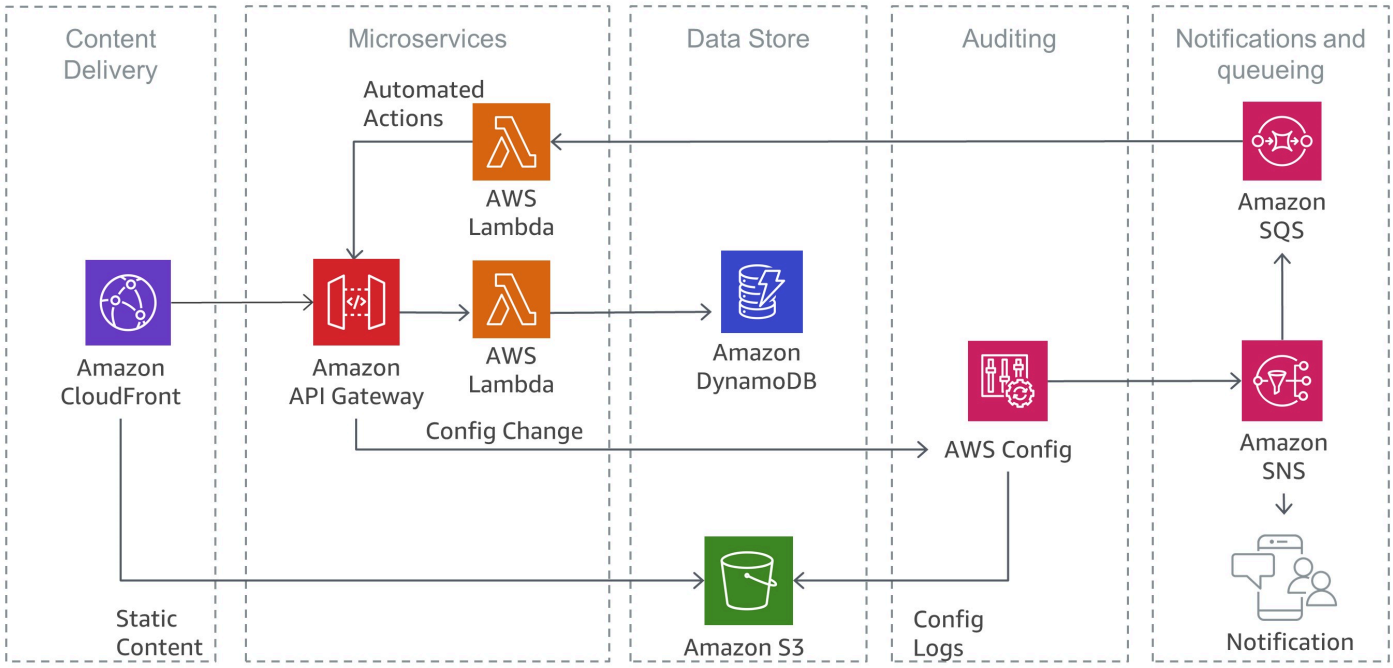
アジャイル開発環境で急速に変化するインフラストラクチャ設定に対する制御を維持するには、アーキテクチャの監査と制御に対するさらに自動化されたマネージドアプローチが不可欠です。

CloudTrail と CloudWatch Events はマイクロサービス全体にわたってインフラストラクチャの変更を追跡して対応するための重要な構成ブロックです。一方、AWS Config のルールを使用すると、企業がセキュリティポリシーを定義し、これらのポリシーに対する違反を自動的に検出、追跡、警告できます。

次の例では、マイクロサービスアーキテクチャ内で非準拠の変更を検出および通知し、自動的に対応する方法を示しています。開発チームのメンバーが、マイクロサービスがエンドポイントに HTTPS リクエストだけを許可するのでなくインバウンド HTTP トラフィックを受け入れるよう許可するため、API ゲートウェイで変更を行いました。

この状況は以前に組織からセキュリティコンプライアンス問題として認識されているため、AWS Config ルールではモニタリング対象となっています。AWS Config ルールは、この変更をセキュリ

ティ違反として認識し、これに対応する 2 つのアクションとして、Amazon S3 バケットで検出された変更のログの作成 (監査用) と、SNS 通知の作成を行います。このシナリオでは、Amazon SNS を 2 つの目的で使用します。指定されたグループに E メールを送信してセキュリティ違反を通知することと、SQS キューにメッセージを追加することです。このメッセージを取得し、API Gateway の設定を変更することで、準拠状態が回復します。



AWS Config を使用したセキュリティ違反の検出

まとめ

マイクロサービスアーキテクチャは、従来のモノリシックアーキテクチャの限界を克服することを目的とした分散型の設計手法です。マイクロサービスはサイクルタイムを改善しながら、アプリケーションや組織の拡張を支援します。ただし、アーキテクチャの複雑さや、運用上の負担が増えるという課題があります。

AWS には、製品チームがマイクロサービスアーキテクチャを構築し、アーキテクチャ上および運用上の複雑さを最小限に抑えるために利用できる多くのマネージドサービスのポートフォリオがあります。このホワイトペーパーでは、AWS の関連サービスと、AWS のサービスでサービス検出やイベントソーシングなどの代表的なパターンをネイティブに実装する方法について説明します。

リソース

- AWS アーキテクチャセンター
- AWS ホワイトペーパー
- AWS 月間アーキテクチャ
- AWS アーキテクチャブログ
- This is My Architecture の動画
- AWS Answers
- AWS ドキュメント

ドキュメントの改訂履歴と寄稿者

ドキュメント履歴

このホワイトペーパーの更新に関する通知を受け取るには、RSS フィードをサブスクライブしてください。

update-history-change	update-history-description	update-history-date
ホワイトペーパーの更新	Amazon EventBridge、AWS OpenTelemetry、AMP、AMG、Container Insights の統合、マイナーなテキスト変更	2021 年 11 月 9 日
マイナーな更新	ページレイアウトの調整	2021 年 4 月 30 日
マイナーな更新	マイナーなテキスト変更。	2019 年 8 月 1 日
ホワイトペーパーの更新	Amazon EKS、AWS Fargate、Amazon MQ、AWS PrivateLink、AWS App Mesh、AWS Cloud Map の統合	2019 年 6 月 1 日
ホワイトペーパーの更新	AWS Step Functions、AWS X-Ray、ECS イベントストリームの統合。	2017 年 9 月 1 日
初版発行	AWS でのマイクロサービスの実装を発行。	2016 年 12 月 1 日

> **Note**
>
> RSS の更新を購読するには、使用しているブラウザで RSS プラグインを有効にする必要があります。

寄稿者

本ドキュメントは、次の人物および組織が寄稿しました。

- Sascha Möllering、AWS、ソリューションアーキテクチャ
- Christian Müller、AWS、ソリューションアーキテクチャ
- Matthias Jung、AWS、ソリューションアーキテクチャ
- Peter Dalbhanjan、AWS、ソリューションアーキテクチャ
- Peter Chapman、AWS、ソリューションアーキテクチャ
- Christoph Kassen、AWS、ソリューションアーキテクチャ
- Umair Ishaq、AWS、ソリューションアーキテクチャ
- Rajiv Kumar、AWS、ソリューションアーキテクチャ

注意

お客様は、この文書に記載されている情報を独自に評価する責任を負うものとします。本書は、(a) 情報提供のみを目的とし、(b) AWS の現行製品と慣行について説明しており、これらは予告なしに変更されることがあり、(c) AWS およびその関連会社、サプライヤーまたはライセンサーからの契約上の義務や保証をもたらすものではありません。AWS の製品やサービスは、明示または暗示を問わず、一切の保証、表明、条件なしに「現状のまま」提供されます。お客様に対する AWS の責任は、AWS 契約により規定されます。本書は、AWS とお客様の間で締結されるいかなる契約の一部でもなく、その内容を修正するものでもありません。