サーバーレスアプリケーションレンズ

AWS Well-Architected フレームワーク

2018 年 11 月

注意

本文書は、情報提供の目的のみのために提供されるものです。このドキュメントは、発行日時点におけるAWSの最新の製品とプラクティスについて記載しており、予告なく変更される場合があります。このドキュメントに記載された情報の独自の評価を実施すること、およびAWSの製品またはサービスの使用については、お客様が責任を負っています。AWSのいずれの製品、いずれのサービスも「現状のまま」提供され、明示または黙示を問わず、いかなる種類の保証も行われません。本文書内のいかなるものも、AWS、その関係者、サプライヤ、またはライセンサーからの保証、表明、契約的なコミットメント、条件や確約を意味するものではありません。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 とお客様の間で行われるいかなる契約の一部でもなく、そのような契約の内容を変更するものでもありません。

コンテンツ

要約

このドキュメントでは、AWS Well-Architected フレームワークのサーバーレスアプリケーションレンズについて説明します。本書では一般的なサーバーレスアプリケーションのシナリオについて紹介し、利用者のワークロードがベストプラクティスに従って設計されるよう主要な要素について解説します。

はじめに

AWS Well-Architected フレームワークは、AWS[1] システムの構築中にお客様が行う決定の利点と欠点を理解するのに役立ちます。このフレームワークを使用することで、信頼性に優れた、セキュアで効率的な費用効果の高いシステムをクラウドで設計するためのアーキテクチャの観点から見たベストプラクティスを学ぶことができます。このフレームワークはベストプラクティスと改善のためのエリアを特定することを前提にアーキテクチャを一貫して測定する方法を提供します。Well-Architected なシステムによって、ビジネスが成功する可能性を大幅に高めることができると考えています。

この「レンズ」では、AWS クラウド上でサーバーレスアプリケーションのワークロードを設計、デプロイ、およびアーキテクティングすることに重点を置きます。ここでは煩雑になるのを避けるため、サーバーレスワークロードに固有の Well-Architected フレームワークの詳細についてのみ解説します。アーキテクチャを設計する場合は、このドキュメントに含まれていないベストプラクティスと質問を検討する必要があります。AWS Well-Architected フレームワークホワイトペーパー[2]を参照することを推奨します。

このドキュメントは、最高技術責任者 (CTO)、設計者、開発者、および運用チームのメンバーなど、技術分野の役割を担う人を対象としています。このドキュメントを読むことで、サーバーレスアプリケーションのアーキテクチャの設計時に使用すべき AWS のベストプラクティスと戦略を理解することができます。

定義

AWS Well-Architected フレームワークは、5 本の柱 (運用上の優秀性、セキュリティ、信頼性、パフォーマンス効率、コスト最適化) に基づいています。サーバーレスのワークロードの場合、AWS はサーバーレスアプリケーション向けの堅牢なアーキテクチャの設計を可能にする複数のコアコンポーネント (サーバーレスと非サーバーレス) を提供します。このセクションでは、本ドキュメントで使用されるサービスの概要について説明します。サーバーレスワークロードを構築する際には検討すべき 6 つのエリアがあります。

- コンピューティングレイヤー
- データレイヤー
- メッセージングとストリーミングレイヤー
- ユーザー管理とアイデンティティレイヤー

- システムモニタリングとデプロイ
- エッジレイヤー
- デプロイのアプローチ

コンピューティングレイヤー

ワークロードのコンピューティングレイヤーは外部システムからのリクエストを管理し、アクセスの制御と、リクエストが適切に承認されるようにします。このレイヤーには、ビジネスロジックがデプロイされ、実行されるランタイム環境が含まれています。

AWS Lambda は、関数レイヤーでマイクロサービスアーキテクチャ、デプロイ、および実行管理をサポートするマネージド型プラットフォームで、ステートレスなサーバーレスアプリケーションの実行を可能にします。

Amazon API Gateway では、ビジネスロジックを実行するために Lambda と統合された、トラフィック管理、認証とアクセス管理、モニタリング、API バージョニングを含む、フルマネージド型の REST API を実行できます。

AWS Step Functions は、連携、ステート、関数のチェーンなどのサーバーレスワークフローを調整するとともに、複数のステップに分割したり、Amazon Elastic Compute Cloud (Amazon EC2) インスタンスまたはオンプレミスで実行中のワーカーを呼び出したりすることで、Lambda 内でサポートされていない長期間に及ぶ実行を組み合わせます。

データレイヤー

ワークロードのデータレイヤーは、システム内から永続的ストレージを管理します。このレイヤーはビジネスロジックに欠かせないセキュアなメカニズムを提供します。データ変更に応じて、イベントをトリガーするメカニズムを提供します。

Amazon DynamoDB は永続的ストレージ向けにマネージド型 NoSQL データベースを提供することで、サーバーレスアプリケーションの構築を容易にします。**DynamoDB ストリーム**との組み合わせで Lambda 関数を呼び出すことにより、DynamoDB テーブル内の変更にほぼリアルタイムで対応できます。**DynamoDB Accelerator** (DAX) は DynamoDB に高可用性のインメモリキャッシュを追加します。これは数ミリ秒から数百マイクロ秒まで、最大で 10 倍のパフォーマンス改善をもたらします。

Amazon Simple Storage Service (Amazon S3) を使用することで、高可用なキーバリューストアを備えた、サーバーレスウェブアプリケーションやウェブサイトを構築できます。静的アセットは、Amazon S3から **Amazon CloudFront** などのコンテンツ配信ネットワーク (CDN) を介して提供できます。

Amazon Elasticsearch Service (Amazon ES) では、Elasticsearch を簡単にデプロイし、安全に保つとともにスケールして、ログ分析、全文検索、アプリケーションモニタリングなどを行うことができます。Amazon ES は検索エンジンと分析ツールの両方を提供するフルマネージドサービスです。

AWS AppSync は、エンタープライズクラスのセキュリティコントロールと、アプリケーション開発を容易にするリアルタイム機能とオフライン機能を備えたマネージド型 GraphQL サービスです。AWS AppSync は、DynamoDB、Amazon ES、Amazon S3 などのサービスに接続するためのアプリケーションおよびデバイス向けに、データ駆動型 API と、GraphQLによる一貫した記述方法を提供します。

メッセージングとストリーミングレイヤー

ワークロードのメッセージングレイヤーは、コンポーネント間の通信を管理します。ストリーミングレイヤーは、リアルタイム分析とストリーミングデータの処理を管理します。

Amazon Simple Notification Service (Amazon SNS) は、マイクロサービス、分散システム、サーバーレスアプリケーション向けに非同期イベント通知とモバイルプッシュ通知を使用して、pub/sub パターンのフルマネージドメッセージングサービスを提供します。

Amazon Kinesis では、リアルタイムのストリーミングデータの収集、処理、分析を簡単に行うことができます。Amazon Kinesis Data Analytics では、標準的な SQL を実行したり、SQL を使用してストリーミングアプリケーション全体を構築したりすることができます。

Amazon Kinesis Data Firehose はストリーミングデータの取得、変換、そして Kinesis Analytics、Amazon S3、Amazon Redshift、および Amazon ES へのロードなどを行います。これにより、既存のビジネスインテリジェンスツールを使って、ほぼリアルタイムの分析を実現できます。

ユーザー管理とアイデンティティレイヤー

ワークロードのユーザー管理とアイデンティティレイヤーは、ワークロードのインターフェイスから外部と内部のお客様向けに、アイデンティティ、認証、認可機能を提供します。

Amazon Cognito では、サーバーレスアプリケーションに対して、ユーザーサインアップ、サインイン、およびデータ同期を簡単に追加することができます。**Amazon Cognito** ユーザープールは、Facebook、Google、Amazon、Security Assertion Markup Language (SAML) などに、組み込みのサインイン画面とフェデレーションを提供します。**Amazon Cognito** フェデレーティッドアイデンティティは、サーバーレスアーキテクチャの一部となる、AWS リソースへの対象を絞ったアクセスを安全に提供します。

システムモニタリングとデプロイ

ワークロードのシステムモニタリングレイヤーは、メトリクスを通じて、システムの可視性を管理し、その運用と動作についてコンテキストを含めた経時的変化を認識できるようにします。デプロイレイヤーはワークロードの変更が、リリース管理プロセスを通じてどのようにプロモートされるかについて定義します。

Amazon CloudWatch では、お客様が利用するすべての AWS サービスのシステムメトリクスにアクセスしたり、アプリケーションレベルのログを統合したりできるほか、お客様固有のニーズに合わせたカスタムメトリクスとして、ビジネスの主要業績評価指標 (KPI) を作成できます。プラットフォームで自動化されたアクションをトリガーできるダッシュボードとアラートを提供します。

AWS X-Ray では、分散トレースとサービスマップを提供してサーバーレスアプリケーションの分析とデバッグを実行でき、リクエストをエンドツーエンドで視覚化して、パフォーマンスのボトルネックを簡単に特定することも可能です。

AWS サーバーレスアプリケーションモデル (AWS SAM) は、サーバーレスアプリケーションのパッケージ化、テスト、およびデプロイに使用される AWS CloudFormation の拡張機能です。SAM CLI を使用して、Lambda 関数をローカルで開発する際のデバッグサイクルを短縮することもできます。

デプロイのアプローチ

マイクロサービスアーキテクチャでのデプロイのベストプラクティスとして、変更は、コンシューマーのサービスコントラクトに影響しないように行います。API の所有者がサービスコントラクトに重要な変更を加える場合、コンシューマーの準備ができていないと、エラーが生じることがあります。

デプロイが安全であることを確認する最初のステップとして、API を使用しているコンシューマーを認識します。コンシューマーとその使用に関するメタデータを収集することで、変更の影響についてデータ主導の決定を行うことができます。API キーは、API コンシューマー/クライアントに関するメタデータを取得するための効果的な方法です。この方法は、API に重要な変更が加えられた場合の連絡方法として多く使用されます。

重要な変更に対して、リスク回避のアプローチを取るお客様によっては、既存のコンシューマーが影響を受けないようにするために、API のクローンを作成して、異なるサブドメイン (例: v2.my-service.com) にルーティングすることもできます。これにより、新しいサービスコントラクトで新しくデプロイを行うことができますが、トレードオフとしてデュアル API (およびその後のバックエンドインフラストラクチャ) を維持するためにさらなるオーバーヘッドが追加されます。

デプロイに対する異なるアプローチを以下の表に示します。

デプロイ	コンシューマーへの影響	ロールバック	イベントモデルの要因	デプロイの速度
All at once	一度にすべて	旧バージョンを再デプロイ	同時実行率が低いイベントモデル	即時
Blue/Green	一度にすべて（事前に一定レベルの本番環境(Green)テストを伴う）	以前の環境(Blue)にトラフィックを戻す	中規模の同時実行ワークロードでの非同期および同期イベントモデルに適している	数分から数時間の検証後すぐにお客様に
Canaries/Linear	1～10% の典型的な初期トラフィックの移行後、段階的に増加させるか、一度にすべて。	以前のデプロイにトラフィックの 100% を戻す	同時実行性が高いワークロードに適している	数分から数時間

All-at-once

名前が示すように、一度にすべてデプロイする場合は、既存の設定の上にすべての変更が一度に影響を及ぼします。このデプロイスタイルの利点は、リレーショナルデータベースなどのデータストアに対するバックエンドの変更では、変更サイクル中にトランザクションを調整するために必要な作業がはるかに少ないことです。このタイプのデプロイスタイルは手間がかからず、同時実行性の低いモデルでは少ない影響で作成できますが、ロールバックについてはリスクが増し、通常はダウ

ンタイムが発生します。このデプロイモデルの最適なユースケースは、ユーザーへの影響が最小限である開発環境です。

Blue-Green

もう 1 つのトラフィック移行パターンは、Blue-Green デプロイを有効にすることです。ダウンタイムがゼロに近いこのリリースでは、ロールバックが必要な場合でも、既存の実稼働環境 (blue) をアクティブに保ちながら、トラフィックを新しいライブ環境 (green) に移行できます。API Gateway では、時間の経過と共に特定の環境に移行するトラフィックの割合を定義できるため、このデプロイスタイルを効果的な手法になります。Blue-Green デプロイは、ダウンタイムを抑えるように設計されているため、多くのお客様が実稼働環境の変更にこのパターンを採用しています。

ステートレスと冪等のベストプラクティスに従うサーバーレスアーキテクチャは、基盤となるインフラストラクチャとのアフィニティがないため、このデプロイスタイルが適しています。必要に応じて、簡単に作業環境にロールバックできるように、これらのデプロイをより小さな増分の変更にバイアスを調整します。

ロールバックが必要かどうかを知るためには、適切な指標が必要です。ベストプラクティスとして、CloudWatch の高解像度メトリクスを使用することを推奨します。このメトリクスでは、1 秒間隔でモニタリングでき、下降傾向を素早く捉えることができます。CloudWatch アラームと組み合わせて使用できるため、ロールバックを早めることができます。CloudWatch メトリクスは、API Gateway、Step Functions、Lambda (カスタムメトリクスを含む)、および DynamoDB でキャプチャできます。

API Gateway Canary デプロイ

Canary デプロイは、制御された環境でソフトウェアの新しいリリースを活用し、迅速なデプロイサイクルを可能にする方法として、増え続けています。Canary デプロイでは、少数のユーザーへの影響を分析するために、新しい変更に対して、少数のリクエストを受け付けるようにデプロイします。AWSクラウドがファシリテートするため、新規デプロイの基盤インフラストラクチャについて、プロビジョニングやスケールを心配する必要がなくなり、このデプロイ方式を導入し易くなっています。

API Gateway で Canary デプロイすると、コンシューマー用に同じAPI Gateway HTTPエンドポイント を維持しながら、バックエンドエンドポイント (Lambda など) に変更をデプロイできます。

さらに、新しいデプロイにルーティングするトラフィックの割合 (%) を制御して、トラフィックのカットオーバーを制御できます。Canary デプロイの実際的なシナリオは、ウェブサイトの再設計です。

すべてのトラフィックを新しいデプロイに移行する前に、少数のエンドユーザーのクリック率をモニタリングできます。

Lambda バージョンコントロール

すべてのソフトウェアと同様に、バージョニングを管理することで、以前に機能していたコードをすばやく表示したり、新しいデプロイに失敗した場合に以前のバージョンに戻したりすることができます。Lambda では、個々の Lambda 関数に対して 複数のイミュータブルバージョンを発行することができます。以前のバージョンは変更できません。Lambda 関数の各バージョンには固有の Amazon リソースネーム (ARN) があり、新しいバージョンの変更は CloudTrail に記録されているため監査可能です。本番稼働環境のベストプラクティスとして、信頼性の高いアーキテクチャを最大限に活用するためにバージョニングを有効にすることを推奨します。

デプロイオペレーションを簡素化し、エラーのリスクを抑えるには、Lambda エイリアスを使用して、開発、ベータ、本稼働といった開発ワークフローにおいてさまざまな Lambda 関数を有効にできます。Lambda エイリアスの例として、API Gateway を Lambda と連携させる場合、Lambdaの本稼働エイリアスのARNを指定することができます。この手法が役立つ点は、新しいバージョンをライブ環境に昇格させるときに、安全なデプロイが可能になることです。これは、呼び出し元の設定内において Lambda エイリアスは静的なままであり、変更の発生を少なくできるためです。

エッジレイヤー

ワークロードのエッジレイヤーは、プレゼンテーションレイヤーおよび外部の顧客への接続を管理します。異なる地理的位置に居住する外部の顧客に、効率的な配信方法を提供します。

CloudFront を使用すると、低レイテンシーかつ速い転送速度で、ウェブアプリケーションのコンテンツとデータを安全に配信する CDN が提供されます。

一般的な設計の原則

Well-Architected フレームワークでは、サーバーレスアプリケーションに対してクラウドにおける適切な設計を可能にするため、次のような一般的な設計の原則を定めています。

- **高速、シンプル、単一**: 関数は短く簡潔で、1 つの目的のみに対応したものであり、その環境はリクエストのライフサイクルに沿っています。トランザクションはコスト効率が影響しやすいので、より速い実行が推奨されます。
- **スケーラビリティを高めるために、同時リクエストを考慮する**: サーバーレスアプリケーションは同時実行モデルを活用します。そのため、設計レベルでのトレードオフは同時実行に基づいて評価されます。
- **何も共有しない**: 関数ランタイム環境および基盤となるインフラストラクチャの存続期間は短いため、一時ストレージなどのローカルリソースは保証されません。ステートはステートマシンの実行ライフサイクル内で操作できます。高い耐久性が求められる場合は、永続的ストレージが推奨されます。
- **ハードウェアのアフィニティはないと想定**: 基盤となるインフラストラクチャは変更される可能性があります。たとえば最適化コードやCPUフラグのようにハードウェアの知り得ない事象への依存は、常に利用できるとは限りません。
- **ステートマシンを使用して、アプリケーションの複数の関数をオーケストレーションする**: コード内で Lambda の実行を連鎖させてアプリケーションのワークフローをオーケストレーションすると、モノリシックで密結合のアプリケーションになります。その代わりに、ステートマシンを使用してトランザクションと通信フローをオーケストレーションします。
- **イベントを使用してトランザクションをトリガーする**: 新しい Amazon S3 オブジェクトの格納や、データベースの更新などのイベントにより、ビジネス機能に応じたトランザクションの実行が可能になります。この非同期のイベント動作はコンシューマーが不可知であることが多く、リーンなサービス設計を実現するための、ジャストインタイムの処理を促進します。
- **故障や重複に備えて設計する**: 障害が発生し、特定のリクエスト/イベントが複数回起きる可能性があるため、リクエスト/イベントからトリガーされるオペレーションは冪等である必要があります。ダウンストリームには適切な再試行を含めることが推奨されます。

シナリオ

このセクションでは、多くのサーバーレスワークロードにおいて一般的な 5 つの主なシナリオと、それらによって AWS 上のサーバーレスワークロードの設計とアーキテクチャにどのような影響があるかについて説明します。これらの各シナリオについての前提事項、設計の一般的な要因、およびこれらのシナリオを実装する方法のリファレンスアーキテクチャを示します。

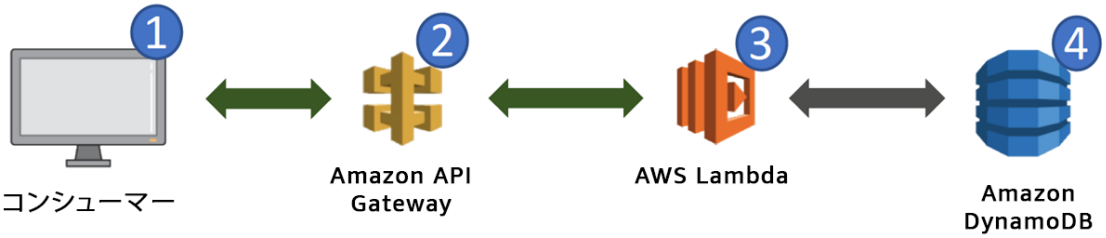
RESTful マイクロサービス

マイクロサービスを構築するときは、ビジネスコンテキストをコンシューマー向けの再利用可能なサービスとしてどのように配信できるかについて検討します。具体的な実装は個別のユースケースに合わせて調整されますが、実装がセキュアで回復性があり、顧客に最善のエクスペリエンスを提供するよう構築するために、マイクロサービスに対する、いくつかの共通した考え方があります。

AWS でマイクロサービスを構築することにより、サーバーレス機能を活用できますが、その他の AWS のサービスや機能、それに AWS のエコシステムや AWS パートナーネットワーク (APN) のパートナーツールも使用できるようになります。サーバーレステクノロジーには耐障害性が組み込まれており、ワークロードに対する信頼性の高いサービスを構築することができます。AWS のエコシステムにより、構築の効率化、タスクの自動化、依存関係のオーケストレーション、マイクロサービスのモニタリングと管理が可能になります。最後に、AWS サーバーレスツールは従量課金制で、ビジネスに合わせてサービスを拡大することで、導入段階やピーク時以外ではコストを抑えることができます。

特徴:

- 簡単にレプリケートでき、高度な耐障害性と可用性を備えた、セキュアで運用が容易なフレームワークを必要としています。
- 利用率やアクセスパターンをログに記録し、お客様をサポートするバックエンドを継続的に改善したいと考えています。
- プラットフォームに対して可能な限り多くマネージドサービスを活用することを求めており、セキュリティやスケーラビリティなど、共通のプラットフォームの管理に伴う負担軽減を期待しています。



リファレンスアーキテクチャ

図 1: RESTful マイクロサービスのリファレンスアーキテクチャ

1. **お客様**は、API (HTTP) コールを実行してマイクロサービスを活用します。理想的には、サービスレベルの一貫した期待値と変更管理を達成するために、API に緊密に結び付けられたサービスコントラクトがコンシューマーに必要です。

2. **Amazon API Gateway** は、RESTful HTTP リクエストをホストし、お客様にレスポンスします。このシナリオでは、API Gateway が組み込みの認証、スロットリング、セキュリティ、耐障害性、リクエスト/応答マッピング、パフォーマンスの最適化を提供します。

3. **AWS Lambda** には、受け取った API コールを処理し、DynamoDB を永続的ストレージとして活用するためのビジネスロジックが含まれています。

4. **Amazon DynamoDB** は永続的にマイクロサービスデータを保存し、要求に基づいてスケールします。多くの場合、マイクロサービスは単一のことを上手に処理するよう設計されているため、スキーマレス NoSQL データストアが通常は組み合わされます。

設定に関するメモ:

- API Gateway のログ記録を活用して、マイクロサービスコンシューマーのアクセス状況や振る舞いを可視化し、理解します。この情報は Amazon CloudWatch Logs で表示でき、Log Pivots を通じて迅速に表示したり、Amazon ES や Amazon S3 (Amazon Athena を使用) などその他の検索可能なエンジンに渡したりできます。配信される情報により、次のような主要な可視性が提供されます。
    - お客様の一般的なロケーションについてします。これは、バックエンドの近さによって地理的に変わる場合があります。
    - お客様がリクエストを入力する方法について理解すると、データベースのパーティション化に役立つ可能性があります。
    - 異常な動作のセマンティクスについて理解します。これはセキュリティフラグになり得ます。
    - エラー、レイテンシー、キャッシュヒット/ミスについて理解すると、設定を最適化できます。

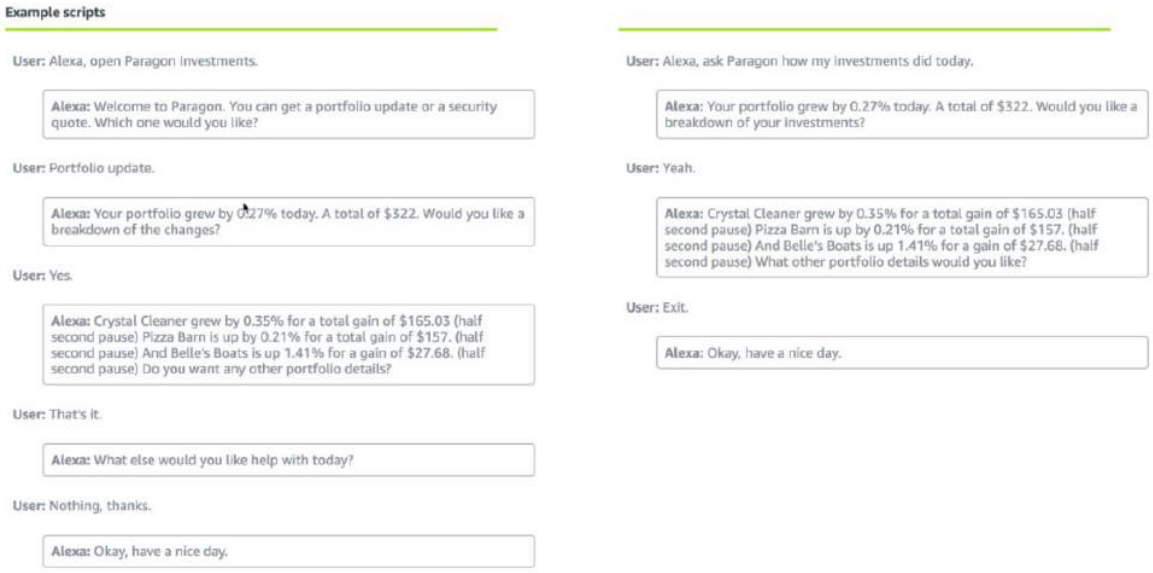
このモデルから、ニーズの成長に合わせてスケールするセキュアな環境を簡単にデプロイおよび維持するフレームワークが得られます。

Alexa スキル

Alexa Skills Kit (ASK) は、自然で魅力的な音声と視覚的なエクスペリエンスにより、開発者が Alexa の機能を拡張できるようにします。よくできたスキルは習慣を作り上げます。つまり、それはユニークな機能を提供するだけでなく、これまでにないスムーズで革新的な方法で価値を提供するため、ユーザーをリピーターにすることができます。

Alexa ユーザーからの最大の不満の原因は、スキルが期待どおりに機能せず、目的のことを達成するために複数のやり取りが必要になることです。最初に音声のインタラクションモデルを設計し、そこから逆算して作業を進めることが重要です。ユーザーによって、話しかける言葉が少なすぎる場合や多すぎる場合、または予期しない言葉を話しかける場合があるからです。音声の設計プロセスには、予期される発話や予期されない発話に対する概念化、スクリプト化、企画化が含まれます。

図 2: Alexa スキルの設計スクリプトの例



基本的なスクリプトを念頭に置き、スキルの構築を開始するときは次の手法を考慮します。

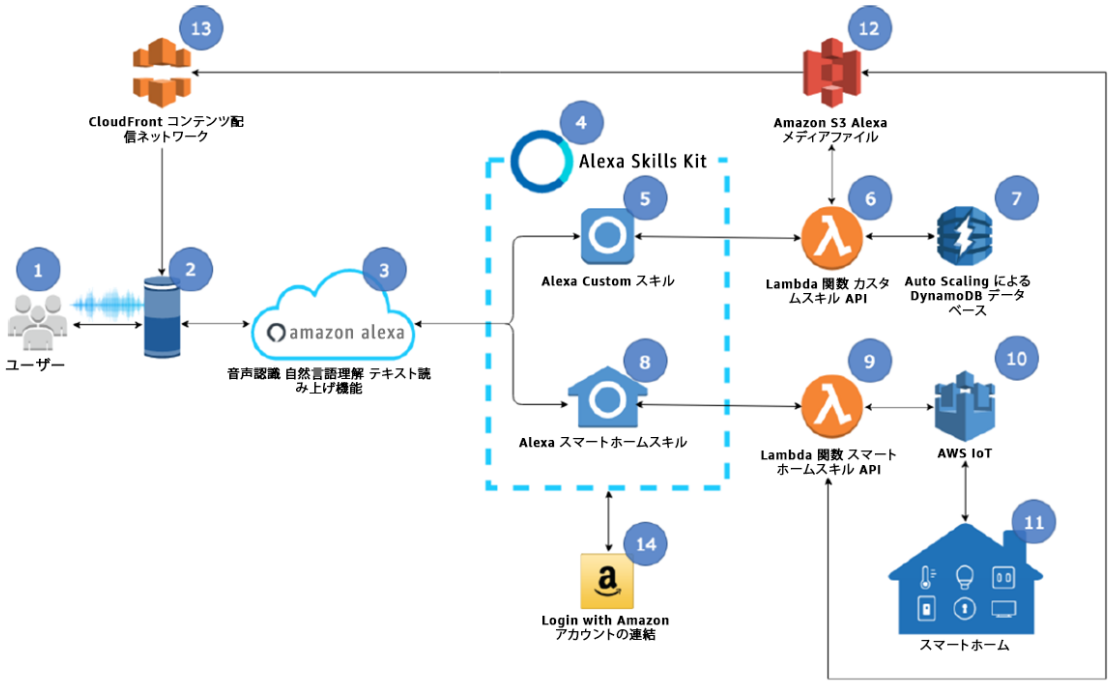
- **完了までの最短のルートを描く**
  - 通常、完了までの最短のルートは、ユーザーがすべての情報とスロットを一度にすべて提供した場合、該当する場合にアカウントが既にリンクされている場合、またはスキルの 1 回の呼び出しでその他の前提条件が満たされた場合です。
- **別の経路やディシジョンツリーを描く**

  - ユーザーの発話に、リクエストを完了するために必要なすべての情報が含まれていない場合は、別の経路やユーザーの意思決定を識別します。

- **システムロジックで行う必要があるバックグラウンドの意思決定を描く**
  - システムのバックグラウンドの意思決定 (新規ユーザーまたはリピートユーザーに対する意思決定など) を識別します。バックグラウンドのシステムチェックにより、ユーザーがたどるフローが変わる可能性があります。
- **スキルによってユーザーをどのように支援するか描く**
  - ユーザーがスキルで何ができるかについて、ヘルプに明確な手順を含めます。スキルの複雑さに基づいて、ヘルプトピックで 1 つのシンプルなレスポンスまたは多くのレスポンスを提供します。
- **存在する場合は、アカウントのリンクプロセスを描く**
  - アカウントのリンクに必要な情報を決定します。また、アカウントのリンクが完了していない場合にスキルが応答する方法を識別する必要もあります。

特徴:

- インスタンス、サーバー、またはその両方を管理することなく、完全なサーバーレスアーキテクチャを作成したいと考えています。
- コンテンツは可能な限り多くのスキルから分離したいと考えています。
- 広範な Alexa デバイス、リージョン、言語にわたる開発を最適化するため、API として公開された魅力的な音声エクスペリエンスを提供することを目指しています。
- ユーザーの需要に合わせてスケールアップまたはダウンし、予期しない使用パターンに対応する伸縮性が必要です。



リファレンスアーキテクチャ

図 3: Alexa スキルのリファレンスアーキテクチャ

1. **Alexa ユーザー**は、Alexa 対応デバイスに話しかけて Alexa スキルを使用します。音声がやり取りの主な方法です。

2. **Alexa 対応デバイス**はリッスンし、ウェイクワードを認識するなど、リクエストがあったときにアクティブ化されます。

3. **Alexa Service** は、Alexa スキルに代わって一般的な音声言語認識 (SLU) 処理を実行します。これには自動化された音声認識 (ASR)、自然言語理解 (NLU)、テキスト読み上げ (TTS) 変換が含まれます。

4. **Alexa Skills Kit** (ASK) は、セルフサービスの API、ツール、ドキュメンテーションとコードサンプルのセットであり、Alexa にスキルを素早く簡単に追加することができます。ASK は信頼された AWS Lambda トリガーで、シームレスな統合を可能にします。

5. **Alexa Custom Skill** はユーザーエクスペリエンスの制御を提供し、カスタムインタラクションモデルを構築できます。これは最も柔軟性の高いタイプのスキルですが、最も複雑です。

6. ASK SDK を使用する **Lambda** 関数により、シームレスにスキルをビルドし、不要な複雑さを回避できます。Alexa Service から送信されたさまざまなタイプのリクエストを処理し、音声レスポンスを構築できます。

7. **DynamoDB データベース**は、スキルの使用に合わせて伸縮自在にスケールする NoSQL データストアを提供できます。データベースは、ユーザーのステートとセッションを保持するためにスキルでよく使用されます。

8. **Alexa Smart Home Skill** により、照明、サーモスタット、スマート TV などのデバイスを Smart Home API を使用してコントロールできます。インタラクションモデルに対するコントロールを提供しないため、このカスタムスキルを構築するには、Smart Home スキルの方が簡単です。

9. デバイスの検出に応答し、Alexa Service からのリクエストを制御するため、**Lambda** 関数が使用されます。これにより、エンターテインメントデバイス、カメラ、照明、サーモスタット、ロックなど、さまざまなモノのコントロールが可能になります。

10. **AWS IoT** により、開発者はデバイスを AWS プラットフォームにセキュアに接続し、Alexa スキルとデバイス間のやり取りを制御できます。

11. Alexa 対応の **Smart Home** は、Alexa スキルのディレクティブを受信し、それに応答する多くの IoT 接続デバイスを持つことができます。

12. **Amazon S3** は画像、テキストコンテンツ、メディアなど、スキルの静的アセットを保存します。これらのコンテンツは CloudFront を使用して安全に配信できます。

13. **Amazon CloudFront** は、地理的に分散されたモバイルユーザーにコンテンツを配信し、Amazon S3 の静的アセット用のセキュリティメカニズムを含む高速なコンテンツ配信ネットワーク (CDN) です。

14. スキルを他のシステムに対して認証する必要があるときは、**アカウントのリンク**が必要です。このアクションは、Alexa ユーザーを他のシステムの特定のユーザーと関連付けます。

設定に関するメモ:

- スキルによって Alexa に送信された可能な限りの Alexa Smart Home メッセージについて、検証用のJSON スキーマを用いることで、Smart Home リクエストとレスポンスペイロードを検証します。
- Lambda 関数のタイムアウトが 8 秒以下に設定されていて、その期間内に関数がリクエストを処理できることを確認します (Alexa Service のタイムアウトは 8 秒です)。
- VPC 内で実行するように設定された Lambda 関数では、Elastic Network Interface (ENI) のセットアップにより起動時のペナルティが発生する場合があります。この追加の処理時間によって、最初のリクエストで Alexa サービスの 8 秒のタイムアウト制限を超える可能性があります。
- DynamoDB テーブルを作成する際は、ベストプラクティス[7] に従います。読み込み/書き込みキャパシティーとテーブルパーティショニングを計算し、応答時間が適切になるようにします。読み取り過多のスキルについては、Amazon DynamoDB Accelerator (DAX) が応答時間を大きく改善します。
- アカウントのリンクにより、外部システムに保存されたユーザー情報が提供されます。この情報を使用して、ユーザーに対してコンテキストに依存するパーソナライズされたエクスペリエンスを提供します。Alexa には、スムーズなエクスペリエンスの提供に役立つアカウントのリンクのガイドラインが用意されています。
- スキルのベータテストツールを使用して、開発中に初期フィードバックを収集し、スキルのバージョニングにより既に実稼働中のスキルへの影響を減らします。
- ASK コマンドラインインターフェイス (ASK CLI) を使用して、スキルの開発とデプロイを自動化します。

モバイルバックエンド

ユーザーは、迅速で一貫し、機能が豊富なユーザーエクスペリエンスをモバイルアプリケーションに期待します。それと同時に、モバイルユーザーのパターンは動的であり、予測できないピーク使用が発生します。また、多くの場合は世界各地で使用されます。

モバイルユーザーからの需要の成長により、アプリケーションはバックエンドインフラストラクチャのコントロールや柔軟性を犠牲にすることなく、シームレスに動作する豊富なモバイルサービスを必要としています。モバイルアプリケーション間では、次のような機能がデフォルトで期待されています。

- データベース変更のクエリ、ミューテーション、サブスクライブの機能
- オフライン時のデータ保持と、接続時の帯域幅の最適化
- アプリケーションでのデータの検索、フィルタ処理、検出
- ユーザー行動の分析
- 複数のチャネル (プッシュ通知、SMS、E メール) を通じた、ターゲットを絞ったメッセージング
- イメージやビデオなどの豊富なコンテンツ
- 複数のデバイスおよび複数のユーザー間のデータ同期
- データの表示と操作のためのきめ細かい認証コントロール

AWS でサーバーレスモバイルバックエンドを構築すると、これらの機能を提供すると同時に、スケーラビリティ、伸縮性、および可用性を効率的かつコスト効果の高い方法で自動的に管理できます。

特徴:

- クライアントからのアプリケーションデータの動作を制御し、API から取得するデータを明示的に選択したいと考えています。
- ビジネスロジックは可能な限り、モバイルアプリケーションから分離したいと考えています。
- API としてビジネス機能を提供し、複数のプラットフォーム間の開発を最適化することを目指しています。
- マネージドサービスを活用して、モバイルバックエンドインフラストラクチャを管理するための差別化につながらない重労働を減らすと同時に、高いレベルのスケーラビリティと可用性を提供することを目指しています。
- アイドルリソースへの支払いと対比して、実際のユーザーの需要に基づくモバイルバックエンドのコストを最適化したいと考えています。

リファレンスアーキテクチャ

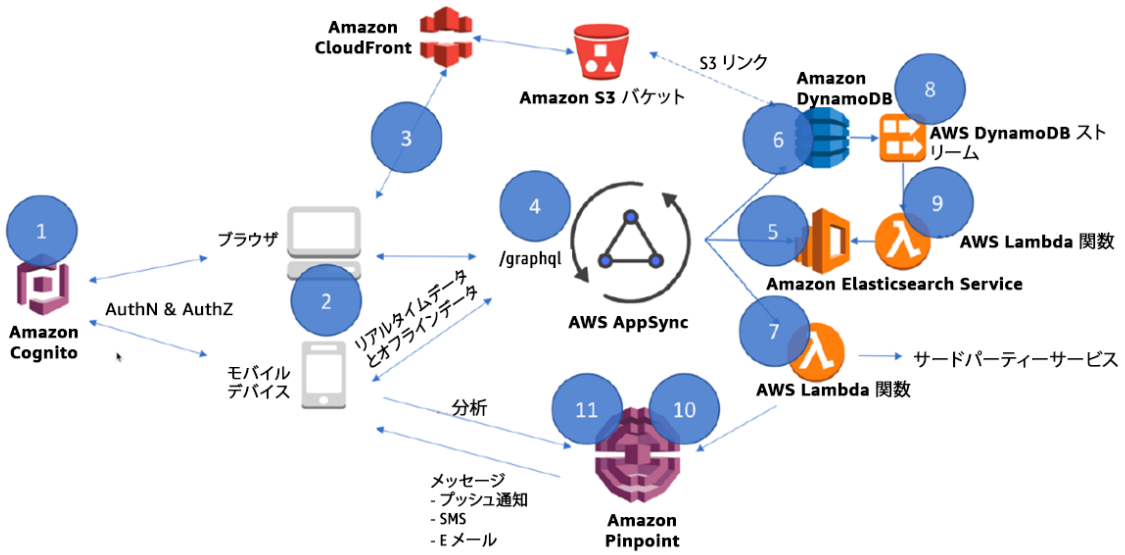


図 4: モバイルバックエンドのリファレンスアーキテクチャ

1. ユーザー管理のため、またモバイルアプリケーションの ID プロバイダーとして、**Amazon Cognito** が使用されます。さらに、モバイルユーザーは既存のソーシャル ID (Facebook、Twitter、Google+、Amazon など) を利用してサインインできます。

2. **モバイルユーザー**は、AWS AppSync および AWS サービス API (Amazon S3 や Amazon Cognito など) に対して GraphQL オペレーションを実行することで、モバイルアプリケーションバックエンドを操作できます。

3. **Amazon S3** は、プロファイルイメージなど特定のモバイルユーザーデータを含むモバイルアプリケーションの静的アセットを保存します。そのコンテンツは CloudFront を使用して安全に配信できます。

4. **AWS AppSync** は、GraphQL HTTP リクエストとレスポンスをモバイルユーザー側でホストします。このシナリオでは、デバイスが接続されている場合、AWS AppSync のデータはリアルタイムで配信されます。また、データはオフラインでも同様に利用できます。データソースは、**Amazon DynamoDB**、**Amazon Elasticsearch Service**、または **AWS Lambda** 関数が利用できます。

5. **Amazon Elasticsearch Service** は、モバイルアプリケーションおよび分析の主要な検索エンジンとして機能します。

6. **DynamoDB** はモバイルアプリケーションに永続的ストレージを提供します。たとえば、有効期限 (TTL) 機能を使用して非アクティブなモバイルユーザーの不要なデータの有効期限が切れるメカニズムを盛り込みます。

7. **Lambda** 関数は、他のサードパーティーサービスとのやり取り、またはカスタムフローで他の AWS サービスの呼び出しを処理します。これは、クライアントへの GraphQL レスポンスの一部になります。

8. **DynamoDB ストリーム**は項目レベルの変更をキャプチャし、Lambda 関数によって、さらに他のデータソースを更新できるようにします。

9. **Lambda** 関数は DynamoDB と Amazon ES の間のストリーミングデータを管理します。その結果、お客様は、データソースと GraphQL のタイプおよびオペレーションを論理的に結合できるようにします。

10. **Amazon Pinpoint** は、ユーザーセッションや、アプリケーションに関する洞察のカスタムメトリクスなど、クライアントの分析をキャプチャします。

11. **Amazon Pinpoint** は、収集された分析に基づき、すべてのユーザー/デバイスまたはターゲットとなるサブセットにメッセージを配信します。メッセージは、カスタマイズして、プッシュ通知、E メール、または SMS チャネルで送信できます。

設定に関するメモ:

- ジョブに対して、最適なリソースを使用していることを確認するために、異なるメモリやタイムアウト設定で Lambda 関数のパフォーマンステスト[3] を行います。
- DynamoDB テーブルを作成するときはベストプラクティス[4] に従い、AWS AppSync が GraphQL スキーマから自動的にプロビジョンすることを検討してください。この場合は、分散されたハッシュキーを使用してオペレーションのインデックスを作成します。適切な応答時間になるように、読み込み/書き込みキャパシティーとテーブルパーティショニングを計算します。
- AWS AppSync クライアント SDK を使用して、アプリケーションエクスペリエンスを最適化します。これは、ライトスルーキャッシュが実装され、クライアントからクラウドまでのレイテンシーが変化する、損失の多いネットワーク接続が考慮されるためです。
- Amazon ES ドメインを管理するときは、ベストプラクティス[5] に従います。また、Amazon ES は、ここでも適用されるシャードとアクセスパターンに関して、設計時には包括的なガイド[6] を提供します。

- 可能な場合は API レベルでキャッシュを活用することで、不要な Lambda 関数の呼び出しを削減します。
- リゾルバで設定された AWS AppSync の Fine-Grained-Access-Control を使用して、必要に応じて GraphQL リクエストをユーザー単位またはグループ単位のレベルにフィルタリングします。AWS AppSync を使用した AWS IAM または Cognito ユーザープールの認可に適用できます。
- アプリケーションを作成して AWS の複数のサービスと統合するには、AWS Amplify および Amplify CLI を使用します。Amplify ツールチェーンは、スタックのデプロイと管理も行います。

ビジネスロジックがほとんど必要とされない低レイテンシー要件の場合、Amazon Cognito フェデレーションアイデンティティは、ユーザーのプロフィール写真のアップロード時、ユーザーをスコープとして Amazon S3 からメタデータファイルを取得する際など、モバイルアプリケーションから直接 AWS サービスと通信できるように、スコープ認証情報を提供します。

ストリーム処理

リアルタイムストリーミングデータの取り込みおよび処理を行うには、アクティビティ追跡、トランザクション注文処理、クリックストリーム分析、データクレンジング、メトリクス生成、ログフィルタリング、インデックス作成、ソーシャルメディア分析、および IoT デバイスデータテレメトリと測定といった、さまざまなアプリケーションをサポートするためのスケーラビリティおよび低レイテンシーが必要です。これらのアプリケーションは、多くの場合、扱いづらく、1 秒あたり数千イベントを処理します。

AWS Lambda と Amazon Kinesis を使用することで、自動的にスケールするサーバーレスストリームプロセスを構築できます。サーバーをプロビジョニングまたは管理する必要はありません。AWS Lambda によって処理されるデータについては、DynamoDB に保存して、後で分析することができます。

特徴:

- インスタンス、サーバー、またはその両方を管理することなく、完全なサーバーレスアーキテクチャを作成して、ストリーミングデータを処理したいと考えています。
- Amazon Kinesis Producer Library (KPL) を使用して、データプロデューサーの観点からデータの取り込みを処理したいと考えています。

リファレンスアーキテクチャ

ここでは、ストリーム処理の一般的なシナリオをご紹介します。これは、ソーシャルメディアデータ分析のリファレンスアーキテクチャです。

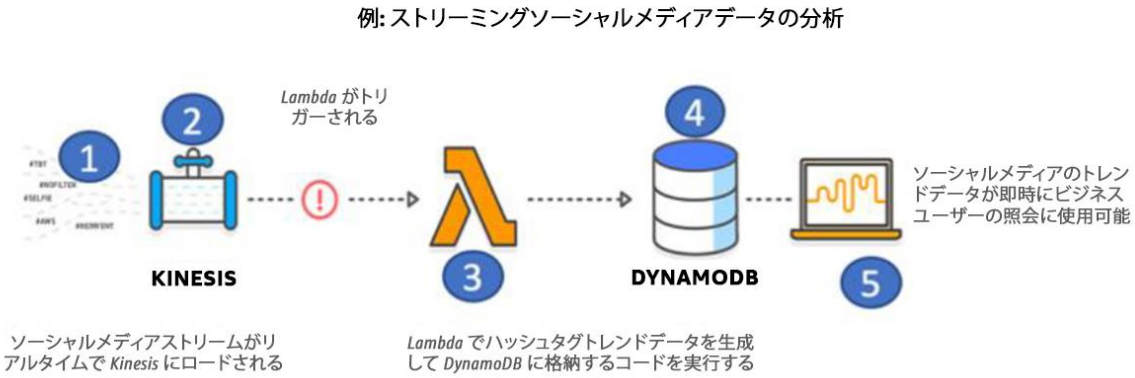


図 5: ストリーム処理のリファレンスアーキテクチャ

1. **データプロデューサー**は、Amazon Kinesis Producer Library (KPL) を使用して、Kinesis ストリームにソーシャルメディアのストリーミングデータを送信します。Kinesis API を活用した Amazon Kinesis Agent とカスタムデータプロデューサーも使用できます。

2. **Amazon Kinesis ストリーム**はデータプロデューサーにより生成されたリアルタイムのストリーミングデータを収集、処理、分析します。ストリームに取り込まれたデータはコンシューマーで処理することができます。この場合、Lambda になります。

3. **AWS Lambda** は単独のイベント/呼び出しとして取り込まれたデータの配列を受信するストリームのコンシューマーとして機能します。Lambda 関数でさらなる処理が実行されます。変換されたデータは永続的ストレージに保存されます。この場合、DynamoDB になります。

4. **Amazon DynamoDB** は AWS Lambda と統合できるトリガーなど、高速で柔軟な NoSQL データベースサービスを提供します。これにより、データはどこでも使用することができます。

5. **ビジネスユーザー**はソーシャルメディアのトレンドデータからインサイトを収集するために、DynamoDB の上でレポートインターフェイスを活用します。

設定に関するメモ:

- 高い取り込みレートに対応するためには、Kinesis ストリームの再シャーディング時に、ベストプラクティス[7] に従います。同時実行ストリーム処理はシャードの数によって決まります。そのため、スループット要件に応じて調整します。
- バッチ処理、ストリームの分析、その他の有用なパターンについては、ストリーミングデータソリューションのホワイトペーパー[8] を参照してください。
- KPL を使用していない場合には、Kinesis API は取り込み時に正常に処理されたレコードと、正常に処理されなかった両方のレコード[9] を返すため、PutRecords のようなアトミックでない操作の部分的な失敗を考慮してください。
- 重複したレコード[10] が発生する可能性があるため、コンシューマーとプロデューサーの両方について、アプリケーション内での再試行回数と冪等性の両方を活用する必要があります。
- 取り込まれたデータを継続的に Amazon S3、Amazon Redshift、または Amazon ES にロードする必要があるときには、Lambda で Kinesis Firehose を使用することを考慮してください。
- 標準 SQL を使用して、ストリーミングデータをクエリ実行し、その結果のみ Amazon S3、Amazon Redshift、Amazon ES、または Kinesis Streams にロードする必要がある場合は、Lambda で Kinesis Analytics を使用することを考慮してください。
- AWS Lambda のストリームベースの呼び出し[11] はバッチサイズ、シャードあたりの同時実行数、詳細に及ぶストリームプロセスのモニタリングなどに対応しているため、このベストプラクティスに従ってください。

ウェブアプリケーション

通常、ウェブアプリケーションには一貫性、安全性、信頼できるユーザーエクスペリエンスを確保するために、厳しい要件が課されています。高可用性、グローバルな可用性、数千または将来的な数百万のユーザーにスケールできる能力を確保するために、企業では予測される最大需要のウェブリクエストに対応するため、大幅な超過リソースの確保を余儀なくされてきました。こうした理由で、企業は膨大なサーバー環境や追加的なインフラストラクチャコンポーネントの管理を強いられることがよくありました。また、こうしたことが原因で高額の投資が必要になったり、リソースを確保するために、リードタイムがかかったりするのが難点でした。

AWS でサーバーレスコンピューティングを使用することにより、画一的なサーバー管理のための負担を強いられたり、プロビジョニングリソースに確信がないまま見込みでサービスを開始したり、無駄なリソースにお金を費やしたりすることなく、ウェブアプリケーションスタック全体をデプロイできます。また、セキュリティ、信頼性、またはパフォーマンスに妥協する必要はありません。

特徴:

- わずか数分で高水準の耐障害性と可用性とともに、グローバルに展開できるスケーラブルなウェブアプリケーションを探しています。
- 妥当な応答時間で一貫性のあるユーザーエクスペリエンスを求めています。
- 共通のプラットフォームの管理に関連する重大な負担を制限するために、自身のプラットフォーム用にマネージドサービスをできるだけ活用する術を模索しています。
- 無駄なリソースにお金を費やすのではなく、実際のユーザーの需要に基づいて、コストを最適化したいと考えています。
- セットアップと運営が容易で、後に最小限の影響で拡張が可能なフレームワークの作成を希望しています。

リファレンスアーキテクチャ

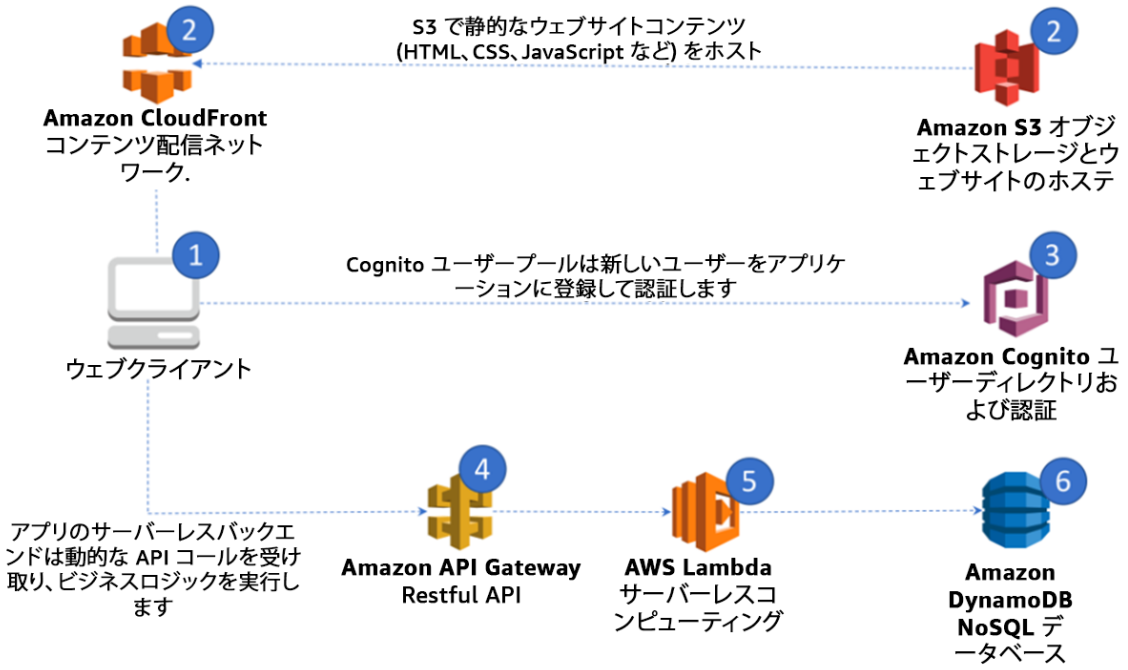


図 6: ウェブアプリケーションのリファレンスアーキテクチャ

1. このウェブアプリケーションの**コンシューマー**は地理的に集中することも、またはワールドワイドになることも想定されます。Amazon CloudFront を活用することで、キャッシングと最適なオリジンルーティングによりこれらのコンシューマーでより良いパフォーマンスのエクスペリエンスが得られるだけではなく、バックエンドへの無駄な呼び出しを制限します。

2. **Amazon S3** はウェブアプリケーションの静的アセットをホストし、CloudFront を通じて安全に供給されます。

3. **Amazon Cognito** ユーザープールは、ウェブアプリケーションにユーザー管理機能と ID プロバイダー機能を提供します。

4. Amazon S3 によって提供された静的コンテンツは、コンシューマーによってダウンロードされるため、多くのシナリオでは、動的コンテンツはアプリケーションに送信するか、アプリケーションで受信する必要があります。たとえば、ユーザーがフォームを介してデータを送信する場合、**Amazon API Gateway** はこれらの呼び出しを実行し、ウェブアプリケーションを通じて表示されるレスポンスを返す安全なエンドポイントとして機能します。

5. **AWS Lambda** 関数は、作成、読み取り、更新、削除 (CRUD) のオペレーションをウェブアプリケーション向けに DynamoDB 上で実行します。

6. **Amazon DynamoDB** は、ウェブアプリケーションのトラフィックに応じて伸縮自在にスケールするバックエンド NoSQL データストアを提供します。

設定に関するメモ:

- AWS にサーバーレスのウェブアプリケーションフロントエンドをデプロイするためのベストプラクティスに従ってください。詳細は、「運用上の優秀性の柱」を参照してください。静的ウェブコンテンツのホストに Amazon S3 を使用し、低レイテンシーかつ速い転送速度でコンテンツを安全に配信するために、CloudFront を活用します。
- シングルページアプリケーション(SPA)の場合、S3 オブジェクトのバージョニング、CloudFront キャッシュの失効、細かく調整されたコンテンツ TTL を活用して、デプロイ上の変更を反映させます。
- 認証と認可については、「セキュリティの柱」の推奨事項を参照してください。
- ウェブアプリケーションバックエンドに関する推奨事項については、RESTful マイクロサービスのシナリオを参照してください。
- パーソナライズされたサービスを提供するウェブアプリケーションの場合は、アクセス許可を持つ異なるユーザーセットの範囲を制限するために、API Gateway の使用量プラン[12] だけでなく、Amazon Cognito ユーザープールも活用できます。たとえば、プレミアムユーザーは API コールでより高いスループットを得られたり、追加の API や追加のストレージにアクセスしたりできるといったものです。
- アプリケーションでこのシナリオで説明されていない検索機能が使用されている場合は「モバイルバックエンドのシナリオ」を参照してください。

Well-Architected フレームワークの柱

このセクションでは、定義、ベストプラクティス、質問、考慮事項、サーバーレスアプリケーションのソリューションを構築する際に関係のある主要な AWS サービスなど、それぞれの柱について解説します。

ここでは煩雑になるのを避けるため、サーバーレスワークロードに固有の Well-Architected フレームワークの質問のみを選出しました。アーキテクチャの設計時には、このドキュメントに含まれていない質問についても考慮する必要があります。AWS Well-Architected フレームワークのホワイトペーパーを一読されることをお勧めします。

運用上の優秀性の柱

運用上の優秀性の柱には、ビジネス価値を提供して、継続的にサポートプロセスと手順を改善するためにシステムを実行し、モニタリングする能力が該当します。

定義

クラウド内での運用上の優秀性について 3 つの領域のベストプラクティスがあります。

- 準備
- 運用
- 進化

プロセス、ランブック、ゲームデイなどに関し、Well-Architected フレームワークで解説される内容に加えて、サーバーレスアプリケーションで運用上の優秀性を実践するために、目を向けるべき固有の分野があります。

ベストプラクティス

準備

このサブセクションに含まれるサーバーレスアプリケーションに固有のオペレーションプラクティスはありません。

運用

SERVOPS 1: サーバーレスアプリケーションでの異常をモニタリングし、対応する方法を教えてください。

非サーバーレスアプリケーションと同様に、異常はより大きな規模の分散システムで発生する可能性があります。サーバーレスアーキテクチャの性質上、分散トレースを備えていることが重要です。

サーバーレスアプリケーションを修正した場合にも、伝統的なワークロードのように、デプロイ管理、変更管理、リリース管理といった多くの原則を伴います。ただし、これらの原則は、既存のツール群に多少の変更を加えることで達成できるでしょう。

AWS X-Ray を使用したアクティブトレースは、分散トレース機能と、さらに迅速なトラブルシューティングを可能にするビジュアルサービスマップを提供するために有効にする必要があります。X-Ray はパフォーマンスの低下を検出し、レイテンシーディストリビューションなど、異常を迅速に把握するのに役立ちます。

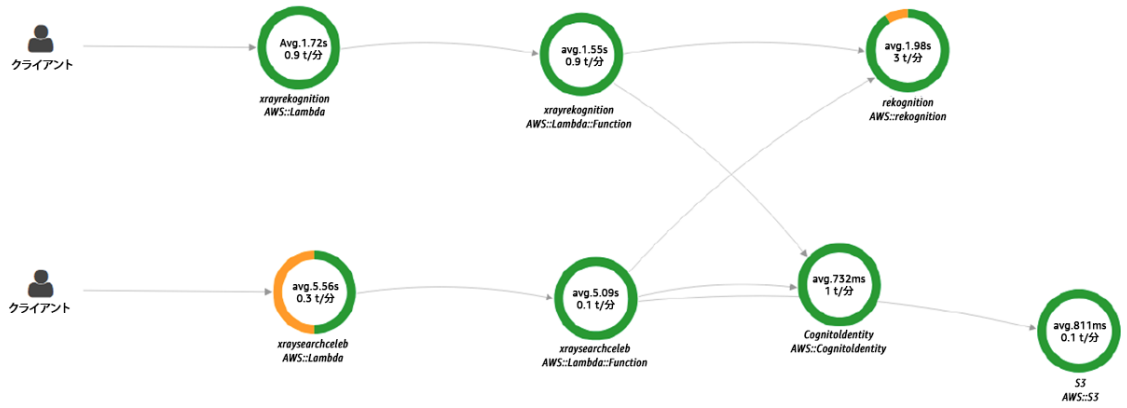


図 7: 2 つのサービスを視覚化した AWS X-Ray サービスマップ

サービスマップは、認知性と回復性のプラクティスを必要とする統合ポイントを理解するのにも役立ちます。統合呼び出しでは、障害がダウンストリームサービスに伝播するのを防ぐために、再試行、バックオフ、場合によってはサーキットブレーカーが必要です。もう 1 つの例はネットワークの異常です。デフォルトのサービス側のタイムアウトや再試行設定を信頼しすぎるべきではなく、特定のクライアント側のソケットの読み取り/書き込みタイムアウトが数分ではなく数秒である場合は、サービス側で早めに失敗するように調整してください。

X-Ray は、アプリケーション内の異常を識別する際の効率を向上させることができる 2 つの強力な機能 (サブセグメントと注釈) も提供します。

サブセグメントは、特定のアプリケーションのオペレーションやロジックが無効になる経緯を理解するのに役立ちます。たとえば、Lambda 関数ハンドラー全体に対して 1 つのサブセグメントを作

成し、追加関数 (sync または async) ごとに 個々に作成して、それらのオーバーヘッドをトップダウン方式で理解することができます。

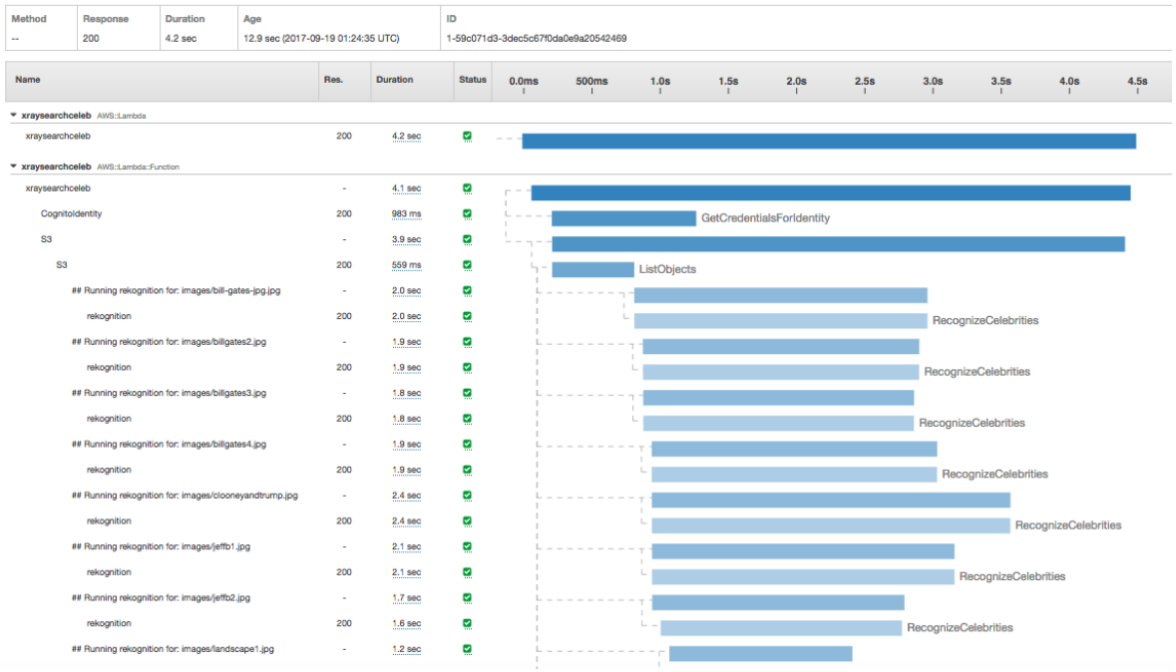


図 8: ## で始まるサブセグメントを持つ AWS X-Ray トレース

注釈は、AWS X-Ray によって自動的にインデックスが作成される文字列、数値、またはブール値を使用したキーと値のペアです。トレースは注釈によってグループ化され、アプリケーション固有の操作に関するパフォーマンス統計 (データベースへのクエリ実行に必要な時間、大勢の人を含む写真の処理にかかる時間、成功するまでに必要な再試行回数、ファイル/イベント/オペレーションあたりの処理時間など) を素早く識別するのに役立ちます。

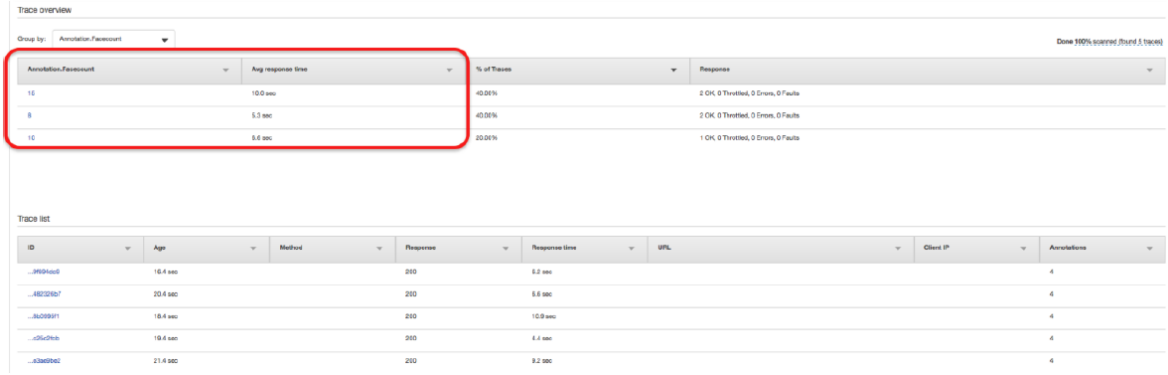


図 9: カスタムの注釈によってグループ化された AWS X-Ray トレース

アラームは個別と集約の両方のレベルで設定する必要があります。集約レベルの例としては、以下のメトリクスでのアラームの発行が挙げられますが、それらに限定されません。

- Lambda: Throttling、Errors、ConcurrentExecutions、UnreservedConcurrentExecutions
- Step Functions: ActivitiesTimedOut、ActivitiesFailed、ActivitiesHeartbeatTimedOut
- API Gateway: 5XXError、4XXError

個別のレベルの例では Lambda からの Duration メトリクスや、API 経由で呼び出された場合の API Gateway からの IntegrationLatency のアラームです。これはアプリケーションの異なるパーツが、異なるプロファイルを持つ可能性があるためです。通常よりはるかに長い時間、関数が実行されるようにする不適切なデプロイは、このインスタンスですぐにキャプチャできます。

API Gateway では、リソース/メソッド/ステージごとの詳細なメトリクス、および p99、p50 (中央値) などのパーセンタイルを作成して、集計された詳細なレイテンシーを表示することができます。同様に、ビジネスおよびアプリケーションの洞察を捉える CloudWatch カスタムメトリクスは、サーバーレスアーキテクチャにも適用されます。

動作を理解して適切な場所にカスタムメトリクスを追加できるように、使用する予定の AWS のすべてのサービスについて、すべての Amazon CloudWatch メトリクスとディメンションを理解して、計画を立てることが重要です。

以下は、ダッシュボードを作成しているか、メトリクスに関して新規および既存のアプリケーションの計画を作成しようとしているのかに関係なく、使用できるガイドラインです。

- **ビジネスメトリクス**
  - ビジネス KPI はビジネスの目的におけるアプリケーションのパフォーマンスを測定します。ビジネス全体に（収益的な観点でなくとも）、いつ甚大な影響が及ぼされているか知るためにとても重要となります。
  - 例: 注文の受理、デビッド/クレジットカードのオペレーション、航空券の購入など。
- **カスタマーエクスペリエンスのメトリクス**
  - カスタマーエクスペリエンスはその UI/UX の全体的な効果を示すだけではなく、アプリケーションの特定のセクションで変更や異常がカスタマーエクスペリエンスに影響を及ぼしているかどうかも示します。時間の経過における影響および顧客基盤を通してどのようにそれが拡大しているかを理解するときの異常値を防ぐ回数をパーセンタイル値で測定。
  - 例: 認知可能なレイテンシー、商品をカートに追加する/決済するまでの時間、ページのロード時間など。
- **システムメトリクス**
  - ベンダーとアプリケーションメトリクスは、前のセクションの根本的な要因を実証するために重要となります。また、これらはシステムが正常か異常か、あるいは顧客がすでに利用しているかなどを示します。
  - 例: HTTP のエラーあるいは成功のパーセンテージ、メモリの使用率、関数の継続時間/エラー/スロットリング、キューの長さ、ストリームレコードの長さ、統合レイテンシーなど。
- **運用メトリクス**
  - 運用メトリクスは、与えられたシステムの持続可能性および運用性を理解するために同じく重要であり、時間の経過によって安定性がどのように向上あるいは低下したかを正確に示すためには欠かせません。

- 例: チケットの回数 (成功/失敗の消化数)、当直担当者が呼び出された回数、可用性、CI/CD パイプラインの統計 (成功/失敗したデプロイ、フィードバック回数、サイクルタイムおよびリードタイムなど)。

SERVOPS 2: 変更の影響を最小限に抑えながら、サーバーレスのアプリケーションをどのように進化させていますか。

初めに、エイリアスとバージョンのみを使用して各ステージの API Gateway エンドポイント、Lambda 関数、Step Functions ステートマシン、およびリソースを分割することを推奨します。$LATEST から LIVE を判別するために Lambda バージョンとエイリアスを活用します。たとえば、CI/CD パイプライン Beta ステージは、次のリソースを Beta AWS アカウントで作成でき、また別々のアカウントのそれぞれのステージ (Gamma、Dev、Prod) に対しても同様に作成できます: OrderAPIBeta、OrderServiceBeta、OrderStateMachineWorkflowBeta、OrderBucketBeta、OrderTableBeta。

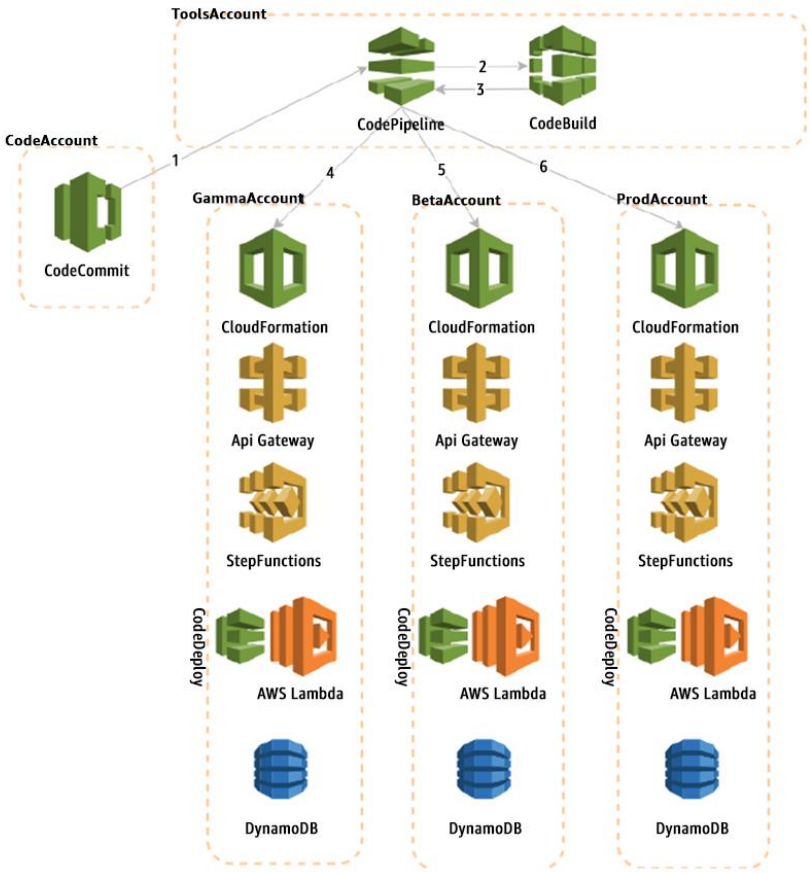


図 10: 複数アカウントの CI/CD パイプライン

次に、新しい変更は時間の経過ごとに徐々にエンドユーザーに移行することが推奨されるため、本番稼働システムでは All-at-once デプロイよりも Canary デプロイあるいは Linear デプロイのような、より安全なデプロイアプローチを使用するようにします。デプロイの検証、ロールバック、およびアプリケーションに必要なすべてのカスタマイズでより高度な制御を取得するために、CodeDeploy フック (BeforeAllowTraffic、AfterAllowTraffic) とアラーム機能を使用します。

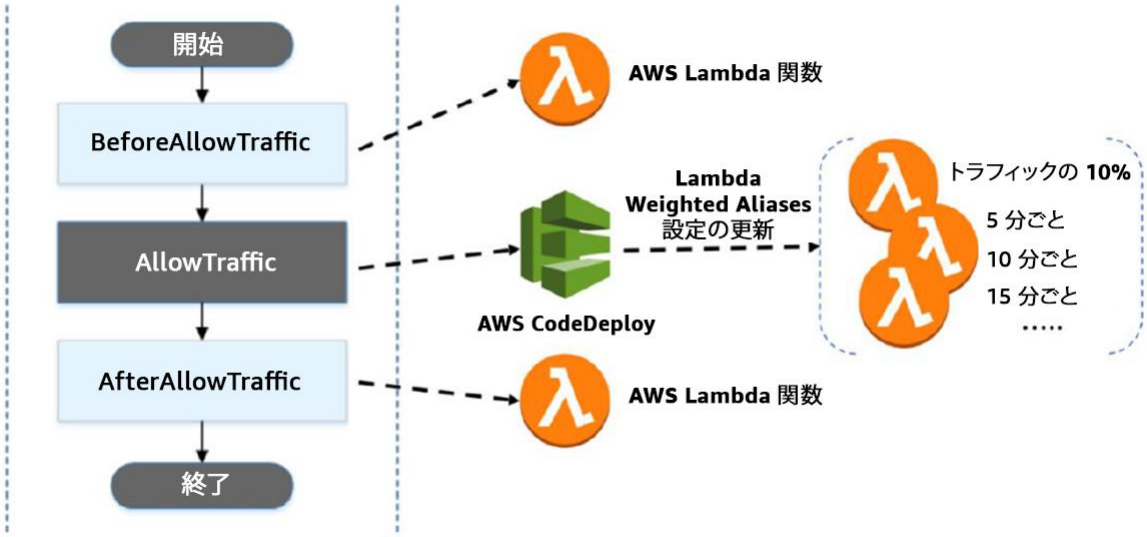


図 11: AWS CodeDeploy Lambda デプロイとフック

AWS SAM を使用して、サーバーレスアプリケーションのパッケージ化、デプロイおよびモデリングを行います。SAM CLI を使用して、Lambda 関数をローカルで開発する際のデバッグサイクルを短縮することもできます。また、AWS 上でサーバーレスソリューションのパッケージ、デプロイおよび管理に使用できるサードパーティー製のサーバーレスフレームワークも数多く出ています。

Lambda 環境変数は、設定からソースコードを分離するために便利であり、また、デプロイを効率化するために役立ちます。たとえば、Lambda 関数内で呼び出されるリソース名が変更された場合、環境変数値のみの修正が必要となり、コードを書き換える必要はありません。

複数のアプリケーションや機能間で共有しているサーバーレスアプリケーションの設定およびシークレットにおけるより細かな制御が必要な場合には、環境変数の AWS Systems Manager (SSM) パラメータストアを検討してください。パラメータストアでは追加のレイテンシーが生じる場合があるため、SSM パラメータストアあるいは環境変数の使用を決めたときは、ベンチマークテストを実行する必要があります。

SAM CLI はローカルのデバッグおよびイベントのシミュレートに便利ですが、本番稼働状況に寄せた環境で、システム動作がどれくらいよく反映できているか、AWS 環境内でもパフォーマンス、統合および回帰テストを実施することを推奨します。

API Gateway ステージ変数および Lambda のエイリアス/バージョンは、デプロイユニットとしてのモニタリングの可視性が低下するなど、運用性およびツールの複雑性をさらに増加させることがあるため、ステージを分離するために使用するべきではありません。

サービスメトリクスにおける高度な洞察を得るには、X-Ray を CloudWatch メトリクスと合わせて使用することを推奨します。X-Ray は、使用するサービス間での AWS Lambda の起動やスロットリングなどのデータポイントに対する可視性を提供し、異常を識別して対応するときに便利です。

進化

このサブセクションに含まれるサーバーレスアプリケーションに固有のオペレーションプラクティスはありません。

主要な AWS サービス

運用上の優秀性に関する主な AWS サービスには、AWS Systems Manager パラメータストア、SAM、CloudWatch、AWS CodePipeline、AWS X-Ray、Lambda および API Gateway などがあります。

リソース

運用上の優秀性のため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 & ブログ

- API Gateway ステージ変数[13]
- Lambda 環境変数[14]
- SAM CLI15
- X-Ray レイテンシー分散[16]
- X-Ray を使用した Lambda ベースのアプリケーションのトラブルシューティング[17]
- System Manager (SSM) パラメータストア[18]
- サーバーレスアプリケーションの継続的デプロイブログ投稿[19]

- SAM Farm: CI/CD 例[20]

ホワイトペーパー

- 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[21]

サードパーティー製ツール

- サーバーレス開発者用ツールページ (サードパーティーを含む) フレームワーク/ツール[22]
- Stelligent: 運用メトリクス用 CodePipeline ダッシュボード

セキュリティの柱

セキュリティの柱には、リスクの評価と軽減戦略を通して、ビジネス価値を提供しながら、情報、システム、アセットを保護する方法が含まれます。

定義

クラウド上のセキュリティには、次の 5 つのベストプラクティスの領域があります。

- ID およびアクセス管理
- 発見的統制
- インフラストラクチャ保護
- データ保護
- インシデントへの対応

サーバーレスはオペレーティングシステムパッチやバイナリ管理などのインフラストラクチャ管理タスクを排除することで、今日におけるセキュリティ上最大の懸念に対応します。非サーバーレスのアーキテクチャと比較して攻撃の発生率は減少しましたが、OWASP およびアプリケーションセキュリティのベストプラクティスは引き続き適用されます。

このセクションの課題は、攻撃者によるアクセス権の取得や、不正使用につながる誤った設定によるアクセス許可の悪用といった具体的な方法に対処することができるように作成されています。このセクションで説明する方法はお使いのクラウドプラットフォーム全体のセキュリティに大きく影響するため、慎重に検証するだけではなく、頻繁に見直す必要があります。

AWS Well-Architected フレームワークからのプラクティスが引き続き適用されるため、**インシデントへの対応**カテゴリについてはこのドキュメントで説明していません。

ベストプラクティス

ID およびアクセス管理

SERVSEC 1: サーバーレス API への認証・認可はどのようにしていますか？

API がよく攻撃者の対象とされるのは、API から重要なデータが取得できることと API を実行できるオペレーションがあることがその理由です。このような攻撃に対する防御としては、いくつかのセキュリティ上のベストプラクティスがあります。認証/認可の観点からは、現在のところ API Gateway 内で API コールを許可するための 3 つのメカニズムがあります。

- AWS_IAM 認可
- Amazon Cognito ユーザープール
- API Gateway Lambda オーソライザー
- リソースポリシー

まず、上記のメカニズムが実装されているか、およびその実装方法を理解します。現在 AWS 環境内にいるコンシューマー、または環境にアクセスするための AWS Identity and Access Management (IAM) 一時認証情報を取得できるコンシューマーの場合、AWS_IAM 認可を活用して、セキュリティが API を呼び出すように該当する IAM ロールに最小限の権限を持つアクセス許可を追加します。

次に示すのは、このコンテキストにおける AWS_IAM 認可を表す図です。

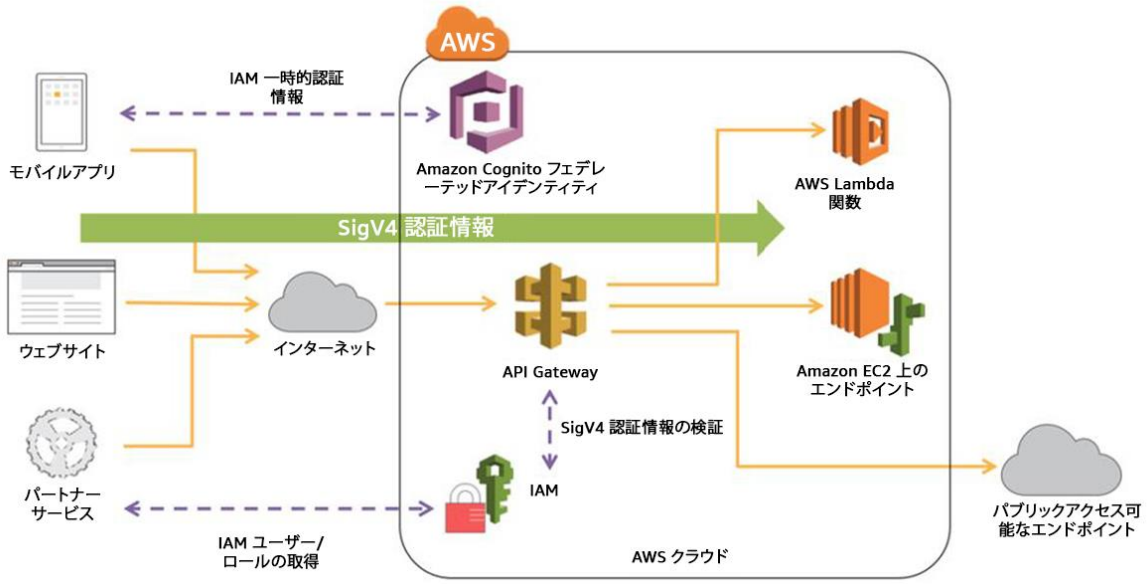


図 10: AWS_IAM 認可

既存の ID プロバイダー (IdP) を現在持っているコンシューマーの場合、API Gateway Lambda オーソライザーを活用して、IdP に対するユーザーを認可/検証するための Lambda 関数を呼び出します。このメカニズムは、既存の IdP の上に追加のロジックを実行する場合にもよく使用されます。Lambda オーソライザーは、ベアラートークンによって届けられた追加情報を送信したり、バックエンドサービスにコンテキスト値をリクエストしたりすることができます。たとえば、オーソライザーはユーザー ID、ユーザー名、およびスコープを含むマップを返すことができます。Lambda オーソライザーを使用することで、バックエンドはユーザーの業務データに認可トークンをマッピングしなくてもよくなるため、このような情報を認可機能にだけ開示するように制限できます。

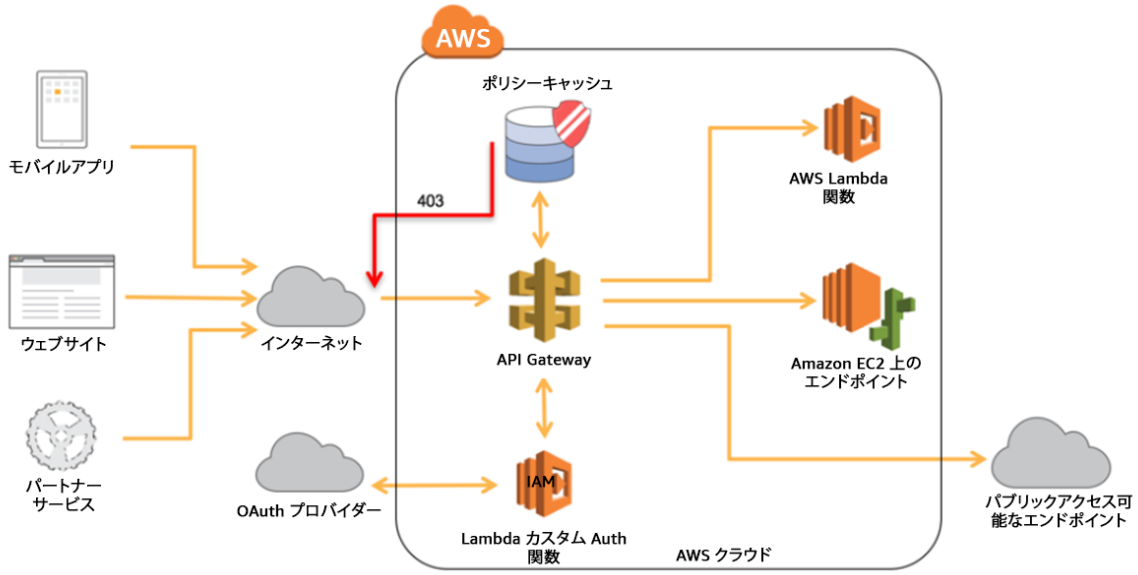


図 12: API Gateway Lambda オーソライザー

IdP を持っていないコンシューマーに対しては、Amazon Cognito ユーザープールを活用して、ビルトインのユーザー管理を提供するか、あるいは Facebook、Twitter、Google+ や Amazon などの外部の ID プロバイダーを使用して統合することができます。

これはモバイルバックエンドシナリオでよく見られ、既存のソーシャルアカウントを使用してユーザーの認可を行うことも、E メールアドレス/ユーザー名で登録やサインインもできます。また、このアプローチは OAuth スコープ を介した詳細な認可も提供しています。

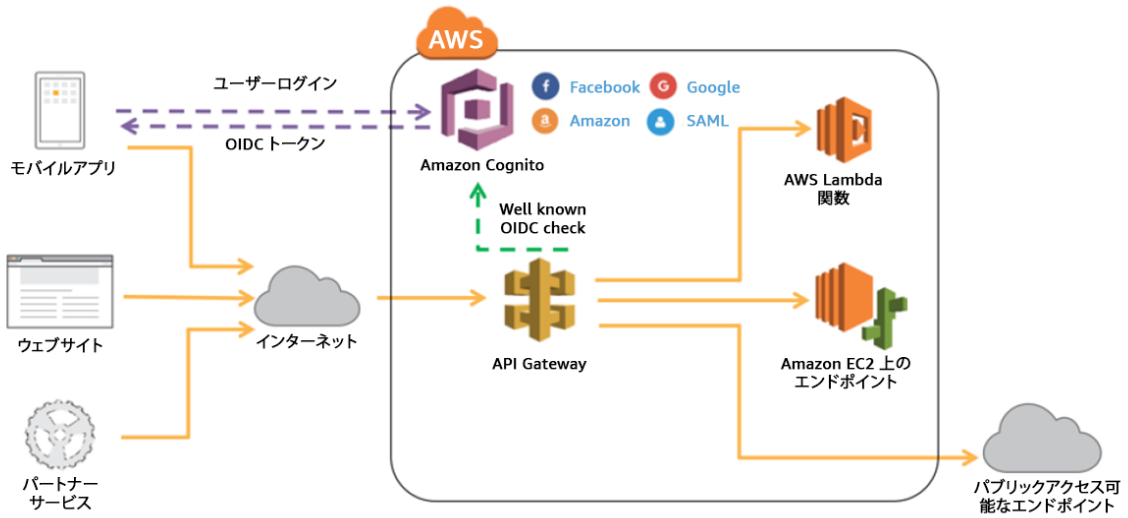


図 13: Amazon Cognito ユーザープール

API Gateway API キー機能はセキュリティメカニズムではないため、認可に使用するべきではありません。この機能は主にコンシューマーの API における使用率を追跡するために使われ、また、このセクションで前に説明したオーソライザーに加えて使用することができます。

Lambda オーソライザーを使用する場合、認証情報または一切の機密データをクエリ文字列パラメータあるいはヘッダーを介して渡さないことを強くお勧めします。そうしないとシステムが解放され、不正使用される恐れがあります。

Amazon API Gateway のリソースポリシーは、指定された AWS プリンシパルが API を呼び出せるかどうかを制御するために API にアタッチできる JSON ポリシードキュメントです。このメカニズムによって、以下のように API コールを制限できます。

- 指定された AWS アカウントのユーザー
- 指定されたソース IP アドレス範囲または CIDR ブロック
- 指定された仮想プライベートクラウド (VPC) または VPC エンドポイント (任意のアカウント)

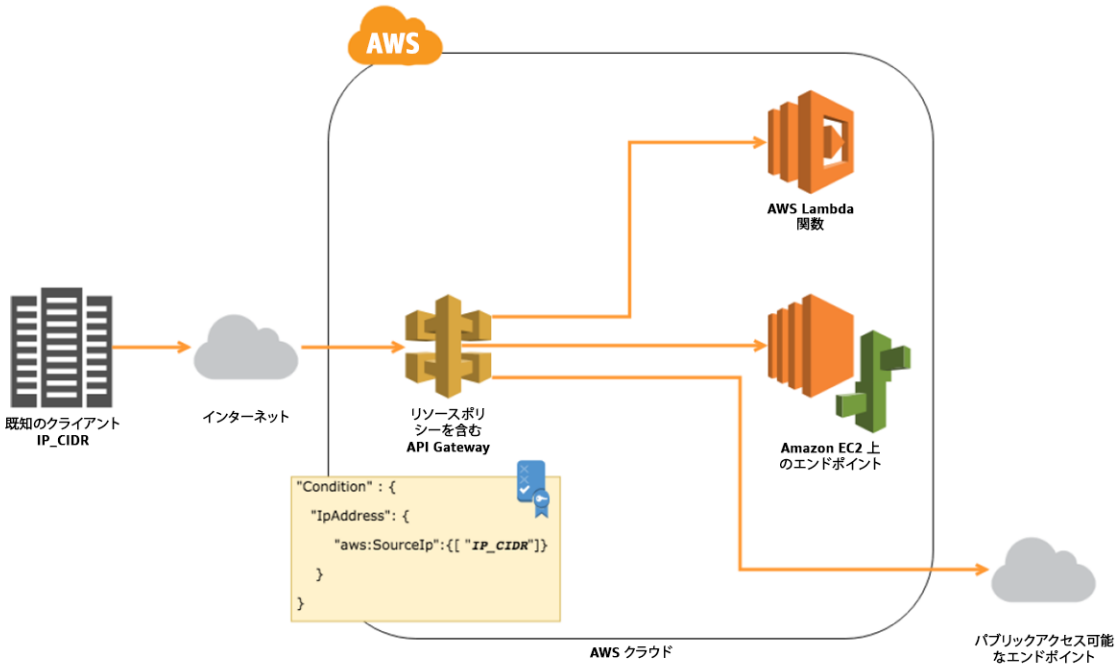


図 14: IP CIDR に基づいた Amazon API Gateway リソースポリシー

リソースポリシーを使用すると、特定の IP 範囲がある既知のクライアントまたは別の AWS アカウントから送信されるリクエストのみを許可するなどの一般的なシナリオを実行できます。

プライベート IP アドレスからのアクセスを制限するなら、代わりに API Gateway プライベートエンドポイント機能を使用します。プライベートエンドポイントを使用すると、API Gateway は VPC 内のサービスとリソースへのアクセスを制限することや、Direct Connect を介して所有のデータセンターに接続するサービスやリソースへのアクセスを制限することができます。

プライベートエンドポイントとリソースポリシーを組み合わせると、1 つの API を特定のプライベート IP 範囲内での特定のリソースの呼び出しに制限することができます。この組み合わせは、同じアカウントまたは別のアカウントにある内部のマイクロサービスによく使用されます。

大規模なデプロイおよび複数の AWS アカウントでは、組織において API Gateway オーソライザークロスアカウントを活用し、メンテナンスをしやすく、またセキュリティの実践を一元化できます。たとえば、API Gateway には別々のアカウントで Cognito ユーザープールを使用できる機能があります。また、Lambda オーソライザーは別々のアカウントで作成して管理でき、API Gateway で管理する複数の API 間で再利用することもできます。どちらのシナリオにおいても、API 間で認可プラクティスを標準化する目的の複数のマイクロサービスをデプロイすることが一般的です。

図 15: API Gateway クロスアカウントオーソライザー

SERVSEC 2: Lambda 関数の AWS サービスへのアクセスはどのように境界制御をしていますか？

Lambda 関数がアクセスできる内容に関しては、最低限のアクセス権限を付与し、指定のオペレーションを実行するために必要な権限のみに限定して許可することが推奨されます。必要以上のアクセス許可をロールにアタッチすることは、不正使用にシステムを解放するようなものです。

従って、限定されたアクティビティを実行するより小さな関数を使用することは、セキュリティ上でさらに Well-Architected なサーバーレスアプリケーションとなります。

IAM ロールに関しては、複数の Lambda 関数内で 1 つの IAM ロールを共有することは最低限のアクセス権限という規則に反する可能性があります。

発見的統制

SERVSEC 3: どのようにサーバーレスアプリケーションログを分析していますか？

セキュリティ/フォレンジックから規制および法的要件という幅広い理由から、ログ管理はWell-Architectedな設計の重要な一面となります。

同様に、使用されるプログラミング言語に関わらず、攻撃者は依存関係で見られる既知の脆弱性を悪用できるため、アプリケーションの依存関係まで脆弱性を追跡することは重要です。

CloudWatch Logs メトリクスフィルタを活用すると、正規表現 パターンマッチングでサーバーレスアプリケーションの標準出力をカスタムメトリクスに変換できます。また、アプリケーションカスタムメトリクスに基づいて CloudWatch アラームを作成し、アプリケーションがどのように動作しているかについての詳細な情報を素早く入手できます。

同様に、AWS CloudTrail ログは監査と API 呼び出し記録の両方で使用することを推奨します。

API Gateway アクセスログの有効化を検討して、必要なデータのみを選んで選択します。設計によっては、サーバーレスアプリケーションに機密データが含まれている場合があります。従って、サーバーレスアプリケーションを通過するすべての機密データを暗号化することが推奨されます。詳細については、データの保護セクションを参照してください。

Lambda 関数は 1 つのタスクを実行し、これをできるだけ早く完了するように設計されています。これにより、コード内で CloudWatch への API コールを実行すると、観察者効果が生じることがあり、過剰な出力記述から不必要なデータがログに取り込まれる場合があります。これは、信号対雑音比および CloudWatch Logs の取り込み料金の両方が増大する要因となります。

SERVSEC 4: サーバーレスアプリケーション内の依存関係の脆弱性をどのようにモニタリングしていますか？

アプリケーションの依存関係の脆弱性スキャンに関しては、数多くの商用ソリューションとオープンソースソリューションがあります (CI/CD パイプラインに統合できる OWASP 依存関係チェックなど)。AWS SDK を含むすべての依存関係をバージョン管理リポジトリに含めることが重要です。

インフラストラクチャ保護

サーバーレスアプリケーションには管理するインフラストラクチャはありませんが、仮想プライベートクラウド (VPC) またはオンプレミス上にあるアプリケーションにデプロイされた他のコンポーネントとのやりとりが必要になるシナリオがあるかもしれません。したがって、この前提に基づいてネットワークの境界が考慮されることが重要です。

SERVSEC 5: VPC アクセスにおいて、AWS Lambda 関数がアクセスできるようにするために、どのようにネットワークの境界を設定していますか？

Lambda 関数は、VPN 接続を介したAWS 外にあるリソースへのアクセスを含む、VPC 内のリソースにアクセスできるように設定できます。AWS Well-Architected フレームワークで説明されているセキュリティグループに関するベストプラクティスを参照してください。

非サーバーレスアプリケーションと同様に、セキュリティグループとネットワークアクセスコントロールリスト (NACL) は、ネットワーク境界の強制に基づくことを推奨します。コンプライアンス上の理由からアウトバウンドトラフィックフィルタリングを必要とするワークロードの場合は、非サーバーレスアーキテクチャに配備される場合と同様にプロキシを使用できます。

どのリソースにアクセスできるかなど、コードレベルの指示のみでネットワーク境界を強制する方法は、関心の分離の観点から推奨されません。

データ保護

SERVSEC 6: サーバーレスアプリケーション内でどのように機密データを保護していますか？

API Gateway は、クライアントおよびインテグレーションを含む、すべての通信においてTLS を使用しています。HTTP ペイロードは転送時に暗号化されますが、URL の一部に含まれるリクエストパスやクエリ文字列は暗号化されない場合があります。AWS Lambdaでは、暗号化されたHTTP ペイロードもまた、API Gateway からの受信時に暗号化されていない場合があります。この結果、もし標準出力に送信した場合、 CloudWatch Logs を通じて機密データが意図せず露出してしまうことがあります。

さらに、不正形式の入力や傍受された入力は、システムへのアクセスを取得するため、あるいは誤動作を引き起こすための攻撃ベクトルとして使用される場合もあります。

機密データは、AWS Well-Architected フレームワークで詳しく解説されているように、できる限り、全レイヤーで常時保護することを推奨します。上記のホワイトペーパーの推奨事項は、ここでも引き続き適用されます。

API Gateway に関して、機密データは、HTTP リクエストとして送信される前にクライアント側で暗号化するか、または、HTTP POST リクエストのペイロードとして送信するかのいずれかが推奨されます。これに加え、所定のリクエストを行う前に、機密データを含む可能性のあるすべてのヘッダーを暗号化することも推奨されます。

Lambda 関数あるいは API Gateway を伴うすべてのインテグレーションに関して、機密データは、全ての処理やデータ操作の前に暗号化する必要があります。これにより、選択された永続ストレージでのデータ露出や、あるいは CloudWatch Logs によって、ストリーム送信され永続化された標準出力を介してデータが露出した場合に漏洩を防止することができます。このドキュメントですでに説明したシナリオのとおり、Lambda 関数は DynamoDB 、 Amazon ES または Amazon S3のいずれかに、永続化時に暗号化データとして保管できます。

HTTP リクエストパス/クエリ文字列の一部であれ、Lambda 関数の標準出力であれ、暗号化されていない機密データを送信、ログ記録、保存することは避けるように強くお勧めします。

機密データが暗号化されないままで、 API Gateway のログ記録を有効にすることも同じく推奨されません。発見的統制のサブセクションで説明したように、このようなオペレーションについては、API Gateway のログ記録を有効化する前に、コンプライアンスチームに相談してください。

SERVSEC 7: 入力の検証はどのように行っていますか？

入力の検証では、最初のステップとして、設定された JSON スキーマリクエストモデル、および URI の必須パラメータ、クエリにリクエストが従っていることを確認したうえで、API Gateway のベーシックリクエスト検証のセットアップを行ってください。

アプリケーション固有の詳細な検証は、これがLambda 関数、ライブラリー、フレームワーク、あるいはサービスに分離されているかに関係なく個々に実装する必要があります。

主要な AWS サービス

セキュリティの主な AWS サービスは、Amazon Cognito、IAM、Lambda、CloudWatch Logs、AWS CloudTrail、AWS CodePipeline、Amazon S3、Amazon ES、DynamoDB、Amazon Virtual Private Cloud (Amazon VPC) です。

リソース

セキュリティに関する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 & ブログ

- Amazon S3 を使用した Lambda 関数用の IAM ロールの例[23]
- API Gateway リクエストの検証[24]
- API Gateway Lambda オーソライザー[25]
- Amazon Cognito フェデレーティッドアイデンティティを使用した API アクセスの保護。Amazon Cognito ユーザープールと Amazon API Gateway[26]
- AWS Lambda への VPC アクセスの設定[27]
- Squid プロキシを使用した VPC アウトバウンドトラフィックのフィルタリング[28]

ホワイトペーパー

- OWASP 安全コーディングのベストプラクティス[29]
- AWS セキュリティのベストプラクティス[30]

パートナーソリューション

- PureSec サーバーレスセキュリティ
- Twistlock サーバーレスセキュリティ[31]
- Protego サーバーレスセキュリティ
- Snyk - 商用脆弱性データベースと依存関係チェック[32]

サードパーティーツール

- OWASP 脆弱性依存関係チェック[33]

信頼性の柱

信頼性の柱には、インフラストラクチャやサービスの障害からの復旧、必要に応じた動的なコンピューティングリソースの獲得、設定ミスや一時的なネットワークの問題に対する障害軽減などのシステムの機能が含まれています。

定義

クラウド上での信頼性には、3 つのベストプラクティスの領域があります。

- 基盤
- 変更管理
- 障害管理

信頼性を確保するには、システムには十分に計画された基盤と適切なモニタリングが実施されている必要があり、需要や要件に応じた変更を行うメカニズムと不正なDoS攻撃を防御するメカニズムが備えている必要があります。システムは障害を発見するように設計されていることが推奨され、また自動的に自己修復できることが理想的です。

ベストプラクティス

基盤

SERVREL l: ピーク時のワークロードにおけるサーバーレスの制限について検討したことがありますか？

AWS では、サービスの不正使用からお客様を保護するために、デフォルトでサービスの制限を実施しています。適切にモニタリングされていない制限は、サービスの劣化やスロットリングにつな

がる場合があります。ほとんどの制限はソフト制限であり、制限を超えることが予想される場合には制限を緩和することができます。

AWS マネジメントコンソールと API 内では AWS Trusted Advisor を使用して、サービスが制限[34] の 80% を超えるかどうかを検出します。

また、ワークロードで制限超過が予想される場合には、前もって制限を引き上げておくこともできます。これらの制限を引き上げる場合は、適合スケールでの最大使用量とサービス制限との間に十分な差を確保するか、あるいは、DoS攻撃を吸収できるようにしてください。関連するアカウントとリージョン全体で上記の制限を検討することを推奨します。

サーバーレスアプリケーションがスパイクするかに関係なく、サービスとトランザクション間のコミュニケーションを設計する場合、非同期パターンに従うことで、より耐障害性に優れたサーバーレスアプリケーションとなります。

SERVREL 2: サーバーレスアプリケーションへのアクセスおよび、サーバーレスアプリケーション内のアクセスについて、 アクセスレートをどのように制限していますか？

スロットリング

マイクロサービスアーキテクチャでは、API のコンシューマーが別々のチームに属している場合や組織外にいる場合があります。これにより、不明なアクセスパターンによる脆弱性、およびコンシューマーの認証情報が漏洩するリスクが生じます。リクエスト数がロジック/バックエンドで処理できる量を超えると、サービス API が潜在的に影響を受ける場合があります。

また、データベースの行の更新や API の一部として S3 バケットに新しいオブジェクトが追加されるなど、新しいトランザクションをトリガーするイベントは、サーバーレスアプリケーションを通じて追加の実行をトリガーします。

スロットリングは、サービスコントラクトによって確立されたアクセスパターンを実施する API レベルで有効にする必要があります。リクエストアクセスパターン戦略を定義することは、コンシューマーがリソースレベルまたはグローバルレベルでサービスをどのように使用すべきかを確立する上で重要です。

API 内で適切な HTTP ステータスコードを返すことで (スロットリングの 429 など)、コンシューマーが適切なバックオフと再試行の実装により、スロットリングアクセスを調整することができます。

より詳細なスロットリングと使用量の計測においては、グローバルスロットリングの他に、使用プラン付きの API キーをコンシューマーに発行すると、API Gateway が想定外の動作に対してアクセスパターンを強制できるようになります。また、API キーは、個々のコンシューマーが疑わしいリクエストを実行した場合に、管理者によってアクセスを遮断するプロセスを簡易化します。

API キーを取得する一般的な方法は、開発者ポータルを通じて行われます。これによって、サービスプロバイダーとなるお客様にコンシューマーとリクエストに関連付けられた追加のメタデータが提供されます。アプリケーション、連絡先情報、およびビジネス領域/目的などをキャプチャして、DynamoDB などの堅牢なデータストアに保存することができます。これにより、コンシューマーの追加的な検証とアイデンティティによるログ追跡が可能になり、また、互換性を破る変更を伴うアップグレード/問題があった場合にコンシューマーに連絡できるようになります。

セキュリティの柱で説明したように、API キーはリクエストを認可するためのセキュリティメカニズムではないため、API Gateway で利用可能な認可オプションのいずれかと組み合わせて使用することを推奨します。

特定のワークロードは、Lambda のように高速でスケールしない場合があるため、サービス障害から保護するために同時実行数の管理が必要となることがあります。同時実行数の管理では、特定の Lambda 関数の同時呼び出し数の割り当てを制御することができ、個別の Lambda 関数レベルで設定できます。個別の関数で設定された同時実行数を超える Lambda の呼び出しは AWS Lambda サービスによってスロットリングされ、その結果はイベントソースによって異なります。同期呼び出しは HTTP 429 エラーを返し、非同期呼び出しはキューに挿入されて再試行となり、ストリームベースのイベントソースはレコードの有効期限が切れるまで再試行されます。

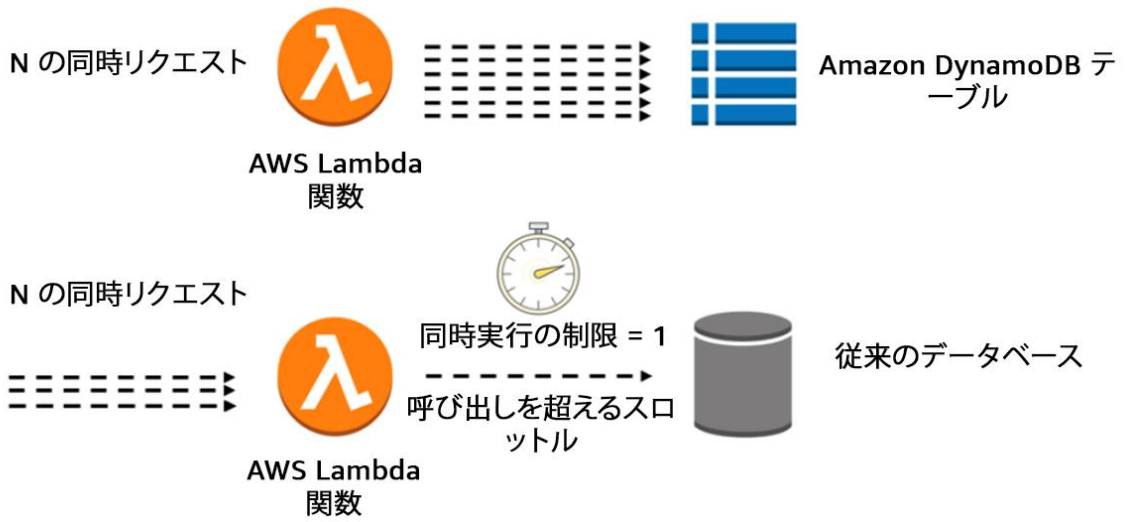


図 16: AWS Lambda 同時実行数制御

同時実行数を制限することは、次のシナリオで特に役立ちます。

- 機密性の高いバックエンドやスケーリング制限がある統合システム
- 同時実行数制限が適用されることのあるデータベース接続プール制約 (リレーショナルデータベースなど)
- クリティカルパスサービス: 同じアカウント内の制限において、優先度が高い Lambda 関数 (認可など) に対する優先度の低い関数 (バックオフィスなど)がある場合
- 異常なイベントにおいて Lambda 関数を無効化する (同時実行数 = 0) 機能
- 求められる同時実行を制限して、分散サービス妨害 (DDoS) 攻撃に対して防御する

SERVREL 3: サーバーレスアーキテクチャ内の非同期呼び出しとイベントにおける戦略はどうしていますか？

非同期呼び出しとイベント

非同期呼び出しは HTTP レスポンスにおけるレイテンシーを低減します。複数の同期呼び出しと長時間の待機サイクルは、タイムアウトや再試行ロジックの妨げとなるコードの「ロック」につながる場合があります。イベント駆動型アーキテクチャは、コードのストリーミング非同期実行を可能にし、それによりコンシューマーの待機サイクルを限定的にします。

サーバーレスアプリケーションで一般的に実装されるイベント駆動型アーキテクチャは、非同期です。ステートマシン、キュー、pub/sub、ウェブフック、イベント、および他の技法は一般的に、ビジネス機能を実行する複数のコンポーネントに適用されます。

ユーザーエクスペリエンスは非同期呼び出しで分離されます。全体的な実行が完了するまでエクスペリエンスのすべてをブロックする代わりに、フロントエンドシステムは初期リクエストの一環としてレファレンス/ジョブ ID を受け取り、ステータスをポーリングするために追加の API が使用されます。この分離により、フロントエンドが、レスポンスを部分的または完全に利用できるときには、リクエストをしている間や、アプリケーションの部分的な遅延読み込みをしている間に、イベントループ、並列実行、または平行実行などの技法を使用して、より効率的に動作させることができます。

また、フロントエンドは、再試行方法の調整やキャッシュの利用でより堅牢になるにしたがって、非同期呼び出しにおける主要素となります。フロントエンドは、異常や一時的な条件、ネットワーク

や劣化した環境などが原因で、SLA の許容範囲でレスポンスを受信できない場合には実行中のリクエストを停止することができます。

これに対して、同期呼び出しが必要な場合は、最低でも実行全体が API Gateway の最大タイムアウトを超過しないことを確実にし、リクエストのライフサイクルで起こりうる状態と例外の両方を制御するために、外部サービス (AWS Step Functions など) の使用が推奨されます。

SERVREL 4: サーバーレスアプリケーションにおけるテスト戦略はどのようにしていますか？

テスト

テストは一般的に単体テスト、統合テスト、および受け入れテストを通じて行われます。堅牢なテスト戦略を開発することで、様々な負荷と条件下でサーバーレスアプリケーションをエミュレートできます。

単体テストは非サーバーレスアプリケーションの場合と同じである必要があり、このため、変更の必要なくローカルで実行できます。

統合テストでは、自身でコントロールできないサービスをモック化すべきではありません。サービスは変化し、予期しない結果を引き起こすことを想定すべきだからです。結合テストは、実際のサービスを使用して実行するのが望ましいと言えます。これは、サーバーレスアプリケーションが本番のリクエストを処理するときに使用するのとまったく同じ環境を提供できるためです。

受け入れテストまたはエンドツーエンドのテストの主な目標は、エンドユーザーが利用できる外部インターフェイスでエンドユーザーのアクションをシミュレートすることであるため、一切の変更なしに実行する必要があります。したがって、ここで認識すべき固有の推奨事項はありません。

一般的に、Lambda とAWS Marketplace で利用可能なサードパーティー製のツールは、パフォーマンステスト実行コンテキストでテストハーネスとして使用できます。

ここでは、パフォーマンステスト時に注意すべきいくつかの考慮事項について説明します。

- Max Memory Usedなどのメトリクスは CloudWatch Logs で確認することができます。メトリクスによって、最適なメモリおよび適切なタイムアウト値を推定できるかもしれません。詳細については、パフォーマンスの柱セクションを参照してください。
- Lambda 関数が VPC 内で実行される場合は、サブネット内で利用できる IP アドレス空間に注意してください。詳細については、運用上の優秀性の柱セクションを参照してください。
- ハンドラーを分割して個別の関数にモジュール化されたコードを作成することで、より単体テストがし易くなります。
- Lambda 関数の静的コンストラクタ/初期化コード (つまり、グローバルスコープ、ハンドラー外部) 内で参照される外部の接続確立コード (リレーショナルデータベースへの接続プールなど) を用いることで、もしLambda の実行環境が再使用された場合には、外部接続のしきい値に達しないようになります。
- DynamoDB の読み込みおよび書き込みテーブルのスループットを適切に調整し、パフォーマンステストサイクルを通してスループットの変動に対応できるように Auto Scaling を設定します。
- パフォーマンステスト中に、ここにリストされていない他のサービス制限がサーバーレスアプリケーションにかかっている可能性があることを考慮します。

SERVREL 5: サーバーレスアプリケーションでの耐障害性をどのように組み込んでいますか？

変更管理

変更が失敗した場合に以前のバージョンに戻すことができる機能があると、サービスの可用性を向上できます。

最初に、モニタリングメトリクスを設定する必要があります。環境におけるワークロードの「通常」の状態が何かを決定し、履歴データをもとに「通常でない」状況を判別するため、CloudWatch に適切なメトリクスとしきい値パラメータを定義します。

また、デプロイをモニタリングし、自動化されたアクションを実装します。SAM Safe Deploymentsなどの機能では変更が環境に与える影響についての、より良い管理方法を提供します。CloudWatch アラームは、変更のいずれかに違反があった場合、以前のデプロイにロールバックするために使用することを推奨します。

障害管理

サーバーレスアプリケーションの一定数は、pub/sub および他のパターンを介し、イベント駆動型で、様々なコンポーネントに非同期呼び出しを行うことを求められます。非同期呼び出しが失敗した場合、それをキャプチャし、できる限り再試行することが推奨され、再試行しない場合、データが失われることがあります。さらに、その結果がカスタマーエクスペリエンスを低下させることがあります。

Lambda 関数の場合、突発的なワークロードの増大がバックエンドを高負荷にしないよう、Lambda クエリに再試行ロジックを構築します。また、関数コード内で Lambda のログライブラリーを活用して、CloudWatch Logs にエラーを記録します。これらの記録は、カスタムメトリクスとして取得できます。詳細については、運用上の優秀性の柱セクションを参照してください。

AWS SDK では、他の AWS のサービスと通信するときに、デフォルトでほとんどのケースにおいて十分なバックオフおよび再試行メカニズムを提供しています。ただし、このメカニズムも再確認することが推奨され、要求に適合するように調整の必要がある場合もあります。

AWS X-Ray およびサードパーティー製アプリケーションパフォーマンスモニタリング (APM) ソリューションは、スロットリングや、ディストリビューションレイテンシーが生じたときにどのように影響を受けるかを特定するために、分散トレースを実行可能にします。

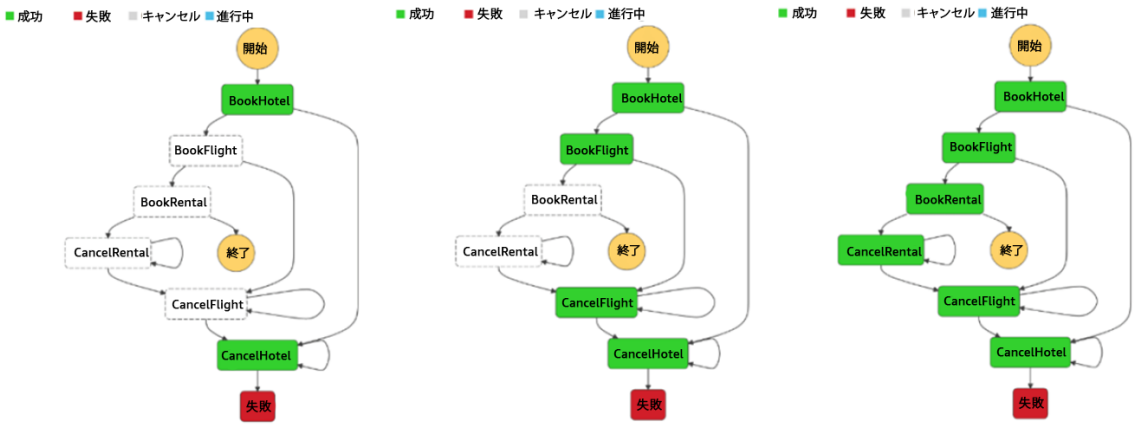
非同期呼び出しは失敗することが想定されるため、ベストプラクティスとしてデッドレターキュー(DLQ)を有効化し、専用のDLQリソース(Amazon SNS やAmazon Simple Queue Service (Amazon SQS)を利用)を個々のLambda関数に対し作成します。また、これらの障害が発生したイベントを対象となるサービスに再発行するために、分離したメカニズムとしてポーリングする仕組みを作ることも推奨されます。

サーバーレスアプリケーション内のカスタマイズされたtry/catch、バックオフ、再実行の量を最小限に抑えるために、可能な限り Step Functions を使用することが推奨されます。詳細については、コスト最適化の柱セクションをお読みください。

さらに、PutRecords(Kinesis) や BatchWriteItem(DynamoDB) のような非アトミックオペレーションは、部分的な障害が発生した場合でも成功を返すことがあります。従って、レスポンスについても、非アトミックオペレーションを使用する場合には検査し、さらにプログラム的に処理されることが推奨されます。

トランザクションベースで特定の保証や要件に依存する同期処理の場合、Saga パターン[35] で記述された失敗したトランザクションのローリングバックも、Step Functions のステートマシンを使用して達成できます。これは、アプリケーションのロジックを分離および単純化します。

図 17: Yan Cui による Step Functions の Saga パターン



主要な AWS サービス

信頼性に優れた主な AWS サービスは、AWS Marketplace、Trusted Advisor、CloudWatch Logs、CloudWatch、API Gateway、Lambda、X-ray、Step Functions、Amazon SQS、および Amazon SNS です。

リソース

セキュリティに関する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 & ブログ

- Lambda の制限[36]
- API Gateway の制限[37]
- Kinesis Streams の制限[38]
- DynamoDB の制限[39]
- Step Functions の制限[40]
- エラー処理パターン[41]
- Lambda を使用したサーバーレスのテスト[42]
- Lambda 関数ログのモニタリング[43]

- Lambda のバージョニング[44]
- API Gateway のステージ[45]
- AWS の API の再試行[46]
- Step Functions のエラー処理[47]
- X-Ray[48]
- LambdaDLQ[49]
- API Gateway および Lambda を使用したエラー処理パターン[50]
- Step Functions の Wait 状態[51]
- Saga パターン[52]
- Step Functions を使用した Saga パターンの適用[53]

ホワイトペーパー

- AWS におけるマイクロサービス[54]

パフォーマンス効率の柱

パフォーマンス効率の柱は、要求に合わせたコンピューティングリソースの効率的な使用と、需要の変化や技術の進化に合わせたその効率性の維持に重点を置いています。

定義

クラウドでのパフォーマンス効率は、次の 4 つの領域で構成されています。

- 選択
- レビュー
- モニタリング
- トレードオフ

高パフォーマンスのアーキテクチャを選択するために、データ駆動型アプローチを採用します。ハイレベルな設計からリソースタイプの選択と設定まで、アーキテクチャのあらゆる側面についてデータを収集します。定期的に選択内容を見直すことで、常に進化している AWS クラウドを最大限に活かすというメリットが得られます。モニタリングにより、期待するパフォーマンスからの逸脱を認識して、それに対するアクションをとることができます。最後に、圧縮またはキャッシュなどを使用す

るか、整合性要件を緩和することで、アーキテクチャのトレードオフを通じてパフォーマンスを向上させることができます。

選択

AWS Lambda のリソースモデルでは、お客様が関数に必要なメモリ量を指定すると、それに比例した CPU パワーとその他のリソース (例: ネットワーキング、ストレージ IOPS) が割り当てられます。たとえば、256 MB のメモリを選択すると、Lambda 関数に 128 MB のメモリの約 2 倍の CPU パワー、512 MB のメモリの約半分の CPU パワーが割り当てられます。

Kinesis リソースモデルでは、取り込みや消費レート (読み取り、書き込みデータサイズ) に基づき、必要なシャード数を選択します。DynamoDB リソースモデルでは、要件に基づき、1 秒あたりに必要になる読み取りや書き込み数を選択します。

サーバーレスアプリケーションのメモリとタイムアウト設定を決定する前に Lambda 関数でパフォーマンステストを実行してください。メモリとタイムアウトを微調整すると、パフォーマンス、コスト、運用手順に大きく影響します。

通信経路で使用されるダウンストリームサービスでの一時的な問題を考慮して、平均の実行値より数秒長く関数のタイムアウトを設定することを推奨します。これは、Step Functions のアクティビティおよびタスクを処理する際にも適用されます。

パフォーマンステストとデータの要件に加えて、Kinesis および DynamoDB のキャパシティーユニットは、ワークロードのプロファイルと密接に関連しています。

SERVPER 1: サーバーレスアプリケーション内で、最適なキャパシティーユニット (メモリ、シャード、1 秒あたりの読み取り/書き込み) をどのように選択していますか?

AWS Lambda でデフォルトのメモリ設定とタイムアウトを選択すると、パフォーマンス、コスト、運用手順で望ましくない影響が生じる可能性があります。

タイムアウトを平均実行時間より大幅に長く設定すると、コードの誤動作時、関数の実行時間が長くなるため、コストが高くなり、関数の呼び出し方法によっては同時実行の制限に達する可能性があります。

関数の単一の正常実行時間に等しいタイムアウトを設定すると、サーバーレスアプリケーションは、一時的なネットワークの問題やダウンストリームサービスでの何らかの異常が生じたときに、突然実行の停止がトリガーされる可能性があります。

ロードテストを実行しないだけでなく、さらに重要なことに、アップストリームサービスを考慮せずにタイムアウトを設定すると、最初にいずれかの部分がタイムアウトに達するといつでもエラーとなる可能性があります。

SERVPER 2: サーバーレスアプリケーションのパフォーマンスをどのように最適化していますか?

最適化

サーバーレスアーキテクチャが組織的に成長するにつれて、さまざまなワークロードプロファイルで一般的に使用されている特定のメカニズムがあります。パフォーマンステストであっても、SLA や要件を常に念頭に置いて、アプリケーションのパフォーマンスを向上させるために設計のトレードオフを考慮する必要があります。

API Gateway キャッシュは、該当するオペレーションのパフォーマンスを向上させるために有効にすることができます。同様に、DAX で読み取りレスポンスを大幅に向上し、グローバルおよびローカルのセカンダリインデックスで DynamoDB のフルスキャンオペレーションを回避することができます。これらの詳細やリソースは、モバイルバックエンドシナリオで説明されています。

API Gateway キャッシュに加えて、コンテンツのエンコードでは、API クライアントは、API リクエストへのレスポンスで送り返される前にペイロードを圧縮するようにリクエストできます。これにより、API Gateway から API クライアントに送信されるバイト数が削減され、データ転送にかかる時間が短縮されます。API 定義でコンテンツのエンコードを有効にするだけでなく、圧縮をトリガーする最小レスポンスサイズを設定することもできます。デフォルトでは、API でコンテンツエンコードのサポートは有効になっていません。

モバイルバックエンドのシナリオでも説明されているように、正確なサイズのサンプルワークフローを使用し、メモリ設定とタイムアウト値を変更して、Lambda 関数のパフォーマンスをテストすることを推奨します。

Lambda 関数コード内でグローバルスコープを活用することで、Lambda コンテナの再利用が可能になります。その場合、データベース接続と AWS サービスの初回接続および設定は、Lambda 関数が実行された環境が使用できる限り、1 回実行されます。

デプロイ

Lambda 関数は必ずしも VPC にデプロイする必要はありません。同様に、CPU とネットワークの帯域幅は、Lambda 関数用に設定されたメモリ設定に基づき、比例的に割り当てられます。

Elastic Network Interfaces (ENI) は事前に作成する必要があり、VPC に関数をデプロイすると Lambda 関数の起動時間が長くなるため、Lambda 関数への VPC アクセスは、必要な場合にのみ設定してください。

Lambda 関数で VPC やインターネットへのアクセスが必要な場合は、Lambda から、インターネットで一般公開されているリソースへのトラフィックを許可するために NAT ゲートウェイが必要になります。高可用性と高パフォーマンスを実現するために、NAT ゲートウェイは、複数のアベイラビリティーゾーンに配置することをお勧めします。

API Gateway については、REST API および API Gateway を使用したカスタムドメインを作成する際に、2 種類の API エンドポイント (エッジ最適化エンドポイント API とリージョンエンドポイント API) から選択できます。

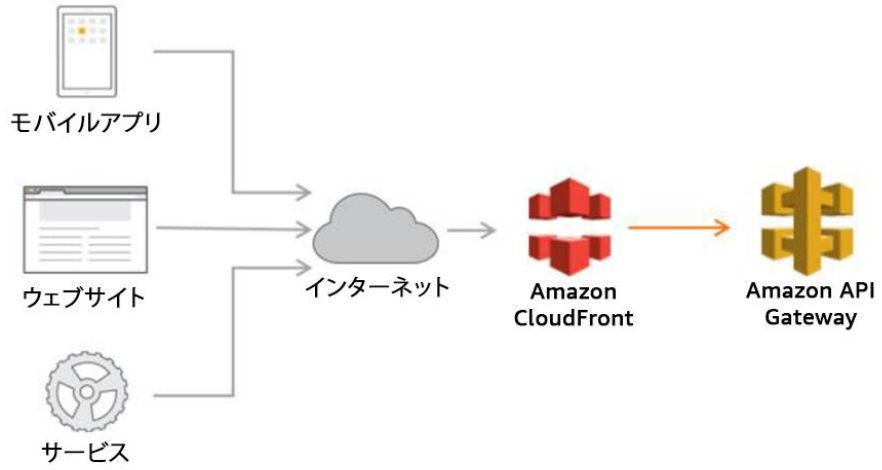
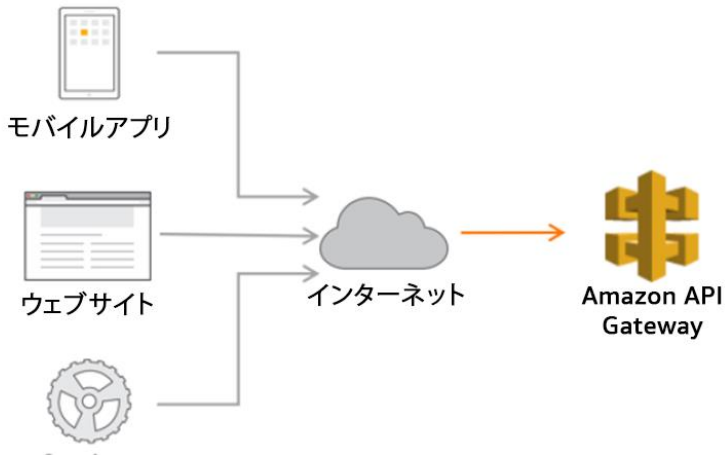


図 18: エッジ最適化 API Gateway のデプロイ

エッジ最適化 API は、API Gateway によって作成および管理される、CloudFront ディストリビューション経由でアクセスするエンドポイントです。API リクエストは最寄りの CloudFront 接続ポイント (POP) にルーティングされます。これにより、通常、地理的に分散したクライアントへの接続時間が短縮されます。APIを作成するときに API エンドポイントタイプを明示的に指定しない場合は、エッジ最適化 API となります。

図 19: リージョンエンドポイント API Gateway のデプロイ



リージョン API は、API Gateway を使用して API を作成する際のシステムデフォルトオプションです。リージョン API エンドポイントは、REST API をデプロイするのと同じ AWS リージョンからアクセスされます。これにより、API リクエストが REST API と同じリージョンから発信される場合に、リクエストのレイテンシーを小さくすることができます。さらに、独自の Amazon CloudFront ディストリビューションとリージョンの API エンドポイントの関連付けを選択することができます。リージョン内のリクエストを発信すると、リージョンエンドポイントは、CloudFront ディストリビューションへの不要なラウンドトリップをバイパスします。

エッジ最適化された API か、リージョン API エンドポイントかを判断するには、以下の表を参照してください。

	エッジ最適化 API	リージョン API エンドポイント
API は、リージョンを越えてアクセスされます。API Gateway 管理の CloudFront ディストリビューションを含みます。	X	
API は、同じリージョン内でアクセスされます。API がデプロイされているのと同じリージョンから API にアクセスした場合に最小リクエストレイテンシーとなります。		X
独自の CloudFront ディストリビューションを関連付ける機能		X

SERVPER 3: VPC にデプロイする必要のあるサーバーアプリケーションのコンポーネントはどのように決定しますか?

VPC で Lambda 関数をデプロイするかどうかを判断するには、以下のディシジョンツリーを参照してください。

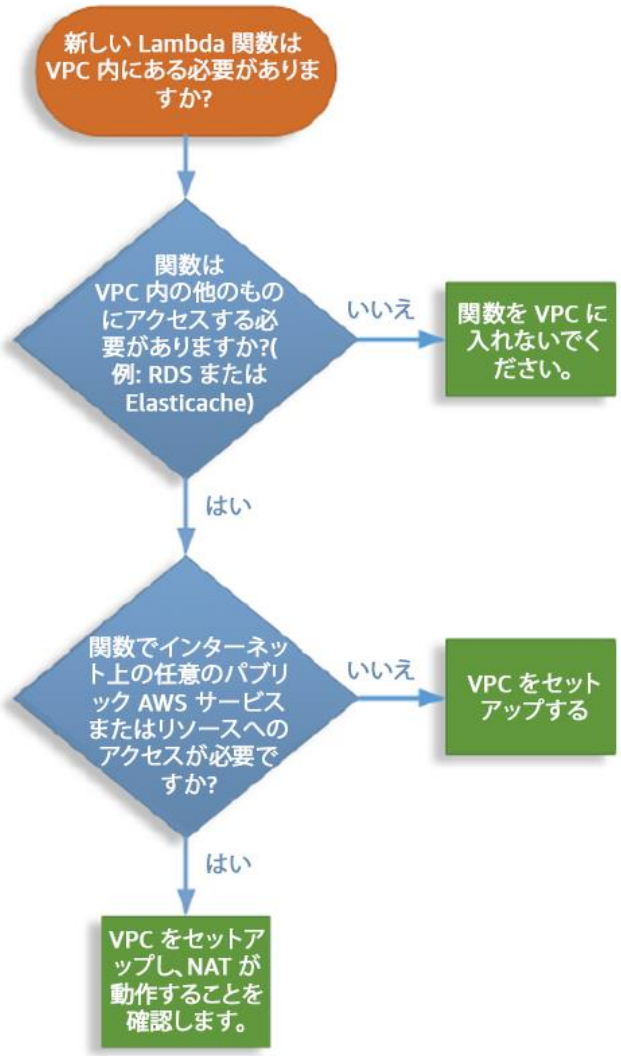


図 20: VPC に Lambda 関数をデプロイするためのディシジョンツリー

SERVPER 4: パフォーマンス向上のために Lambda コードをどのように最適化していますか?

コードの最適化

Lambda 関数は単一用途にすると、可能な限り高速で実行されます。ただし、非サーバーレスアプリケーションの既存のアプリケーション依存関係/フレームワークが、このコンテキストでは最良の選択でない可能性があるため、実行環境の利点を活かすために活用できる手法と、一般的な設計の原則があります。

AWS では、コンテナの再利用、ランタイムの必要に応じたデプロイパッケージサイズの最小化、依存関係の複雑さの最小化など、Lambda 関数 55 で作業するためのベストプラクティスに従います。この決定を行う際は、このトレードオフが重要です。他のチームと合意したアプリケーション SLA に影響を及ぼさないように、99 パーセンタイル値 (P99) を常に考慮する必要があります。

VPC 内の Lambda 関数では、VPC のパブリックホスト名の DNS 解決は回避することをお勧めします。この処理は解決に数秒かかり、リクエストに数秒の課金可能時間が追加される可能性があるためです。たとえば、Lambda 関数で VPC 内の Amazon RDS DB インスタンスにアクセスする場合は、no- publicly-accessible オプションを指定してインスタンスを起動します。

SERVPER 5: データベース接続の初期化はどのように行っていますか?

Lambda 関数が実行されると、AWS Lambda は別の Lambda 関数呼び出しに備えて、ランタイムコンテナを一定期間維持します。

「最適化」サブセクションに記載されているように、グローバルスコープを活用します。たとえば、Lambda 関数でデータベース接続を確立すると、その後の呼び出しには、接続を再確立するのではなく、元の接続が使用されます。関数が再度呼び出されたときにさらに最適化するために、Lambda 関数ハンドラコードの外部でデータベース接続と他のオブジェクト/変数を宣言します。接続を作成する前にすでに接続が存在するかどうかを確認するロジックをコードに追加してください。

SERVPER 6: 非同期トランザクションをどのように実装していますか?

同期トランザクションを使用して最新のインタラクティブユーザーインターフェイスを構築できますが、このアプローチは機能を追加していくと持続不可能になります。追加機能を実装するには、すべてのトランザクションを正常に実行するための複雑なワークフローが必要です。ワークフロー内のいずれかのチェーンが原因で問題やレイテンシーが生じ、非常に脆弱な状態になる可能性があります。

最新の UI フレームワーク (Angular.js、VueJS、React など)、非同期トランザクション、クラウドネイティブワークフローは、コンポーネントの分離と、プロセスやビジネスドメインへの集中を支援するだけでなく、お客様の需要を満たす持続可能なアプローチを提供します。

これらの非同期トランザクション (または「イベント駆動型アーキテクチャ」) は、クライアントに応答を待たせる代わりに、クラウド内のダウンストリームイベントをトリガーします。非同期ワークフローでは、データ取り込み、ETL オペレーション、注文/リクエストフルフィルメントなど、さまざまなユースケースを処理します。

これらのユースケースでは、データは到達時に処理され、変更時に取得されます。一般的な 2 つの非同期ワークフローのベストプラクティスについて説明します。

- サーバーレスデータ処理
- ステータス更新を伴うサーバーレスイベント送信

サーバーレスデータ処理

サーバーレスなデータ処理ワークフローでは、データはクライアントから Kinesis に取り込まれ (Kinesis エージェント、SDK、または API を使用)、Amazon S3 に到達します。これにより、オブジェクトが Amazon S3 に到達した後に自動的に実行される Lambda 関数が開始されます。データが Amazon S3 に安全に格納されると、通常 Lambda 関数は、データを変換または分割してさらに処理するために使用されます。場合によっては DynamoDB または別の S3 バケットなど、データが最終形式になる場所に格納されます。

データ型によって変換が異なる場合があるため、最適なパフォーマンスを得るために、変換を別々の Lambda 関数に細かく分割することを推奨します。このアプローチでは、データ変換を並列して実行し、スピードを上げると同時にコストを削減する柔軟性が得られます。

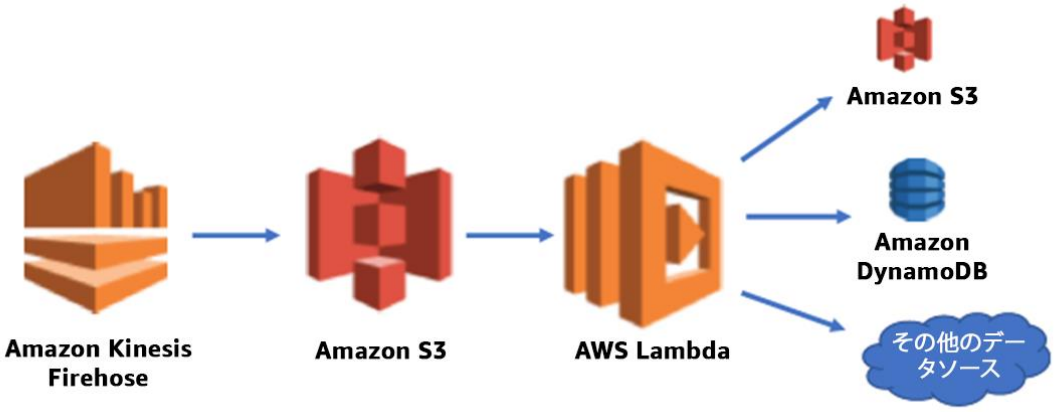


図 21: 非同期データ取り込み

Kinesis Data Firehose では、変換などの特定のシナリオにおいて後続の Lambda 関数が不要になるため、特定のユースケースに役立つ可能性のあるネイティブデータ変換を行うことで

きます (例: Apacheログ/システムログの CSV、JSON 変換、JSON から Parquet または ORC への変換)。このルールの例外として、Athena または Redshift Spectrum 向けに、ストレージ最適化やクエリ形式化をするために、データ分割する場合があります。

ステータス更新を伴うサーバーレスイベント送信

e コマースサイトがあり、ユーザーが注文を送信すると、在庫調整と出荷プロセス、または、応答に数分かかる大規模なクエリを送信するエンタープライズアプリケーションが開始されるとします。

これらのプロセスでは、終了までに時間を要するかもしれない複数のサービスコールを伴う、一般的なトランザクションを完了しなければなりません。また、これらのコールでは、再試行やエクスポネンシャルバックオフを追加することで障害から未然に保護しようと考えています。これによって、完了を待つユーザーにとってユーザーエクスペリエンスが最適ではなくなることがよくあります。

同じように長くて複雑なワークフローでは、新たな認可済みリクエストを受けて、このビジネスワークフローを開始する Step Functions の直前に 、API Gateway を使用できます。

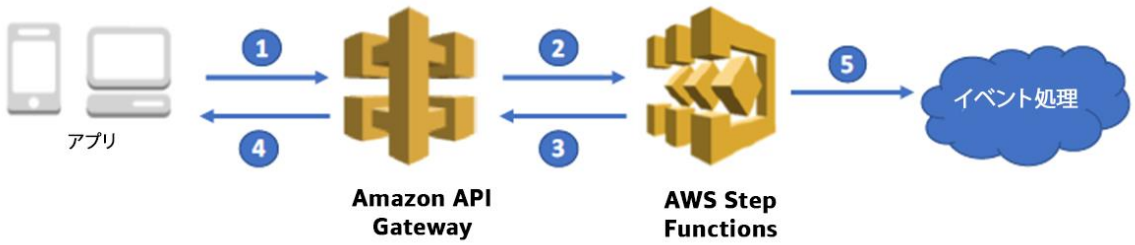


図 22: Step Functions ステートマシンによる非同期ワークフロー

Step Functions がステートマシンの実行を開始すると、即時にレスポンス (実行 ID) が API Gatewayに返され、続いて発信者 (モバイルアプリ、SDK、ウェブサービスなど) に返されます。このようなレスポンスを使用し、Step Functions ウェブサービスをクエリすることで、ワークフローの進行状況について、ユーザーにリアクティブにフィードバックを提供することができます。

一方、このようなワークフローを開始する前に何らかのデータ変換が必要な場合や、ワークフローの開始前に入ってくるリクエストを認識してイベントをキューに保存する必要がある場合も考えられます。

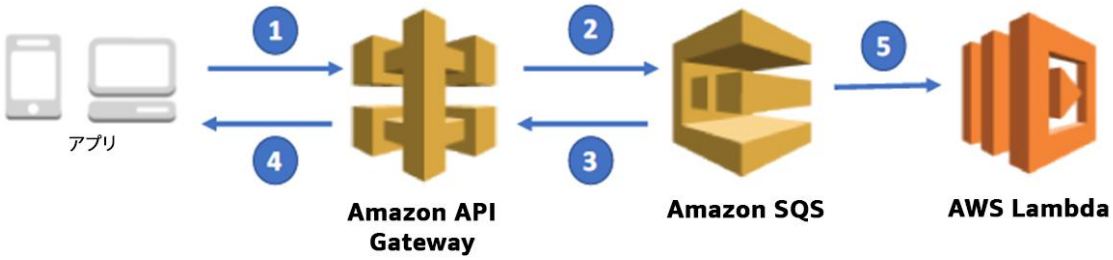


図 23: キューをスケーリングレイヤーとして使用した非同期ワークフロー

このシナリオでは、API Gateway はクライアントに、リクエスト ID を含むレスポンスを返します。クライアントは後でこのリクエスト ID を追跡できます。また、Lambda 関数を使用して、キューに挿入した後で個別のリクエスト ID を生成しクライアントに返すこともできます。

この例では、SQS によって管理されるこのようなキューには複数の目的があります。

1. リクエストレコードを永続的に保存することが重要です。これはクライアントが、リクエストが順番に処理されていることを確認しながら確実にワークフロー全体を続行できるようにするためです。

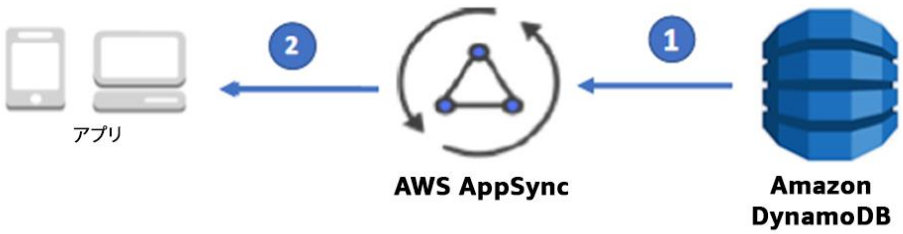
2. 一時的にバックエンドに負荷がかかるようなイベントのバースト時には、リソースが使用できるようになってからリクエストをポーリングして処理することができます。

注意: キューを使用しない最初の例と比較して、Step Functions はキューやステート追跡用のデータソースを必要とせずにデータを永続的に保存しています。

どちらの例でも、ベストプラクティスはクライアントがリクエストを送信した後、できる限り非同期ワークフローを追跡し、レスポンス結果待ちがブロックコードになるのを避けることです。ただし、それでもクライアントに結果を通知する必要がある場合、この目的のために検討できる一般的なソリューションはウェブソケット、ウェブフックおよびポーリングです。

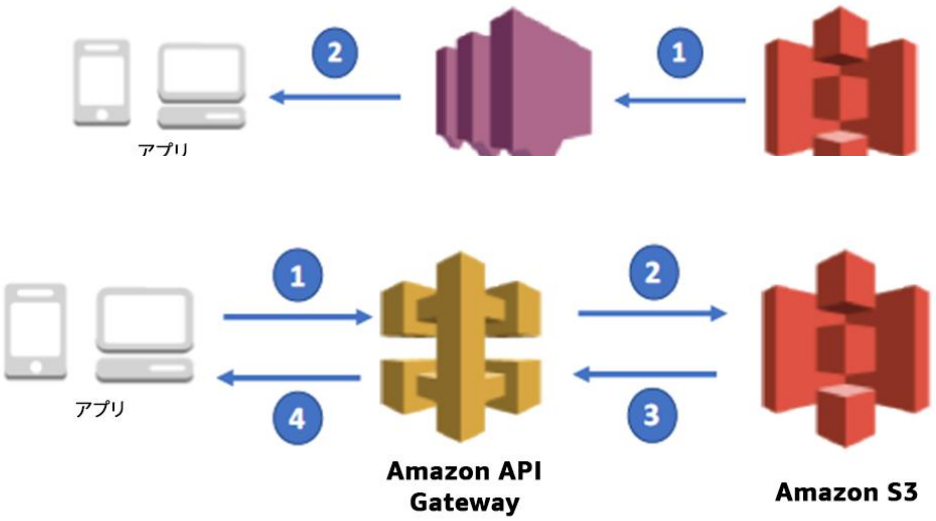
ウェブソケットについては、AWS AppSync がこの機能をすぐに使用できる形で提供しています。クライアントはウェブソケットを開いて GraphQL サブスクリプション (AppSync 経由でミューテーションコールをリッスンする) を使用できます。これはストリーミングデータや、複数のレスポンスが生じる可能性のあるデータにとって理想的です。AppSync を使用することで、ステータスが DynamoDB 内で変更されると、クライアントは自動的にサブスクライブしているため、発生のたびに更新を受信することができます。これはデータ駆動型ユーザーインターフェイスには最適なパターンです。

図 24: AppSync および GraphQL を使用したウェブソケット経由の非同期更新



ウェブフックの場合は、SNS トピックのサブスクリプションを使用して、イベント (例: データファイルが S3 に到着した) の際に通知する HTTP エンドポイントをクライアントがホストできます。このウェブフックパターンでは、イベントの際にメッセージがトピックに発行されると、SNS はそのイベントを POST 経由で別のHTTP(s) エンドポイントに配信します。クライアントがエンドポイントをホストできる、別のマイクロサービスとして構成できる場合は、このパターンが最適です。

図 25: SNS を使用したウェブフック経由の非同期通知



最後に、ポーリングを使用する場合、クライアントが定期的に API をポーリングしてステータスを取得するのは、コストの観点でもサービスをホストするリソースの観点でも、スケール時に負担となります。環境の制約のためにポーリングが唯一の選択肢である場合、ベストプラクティスは顧客と SLA を締結して「空のポーリング」数を制限することです。

図 26: 直近のトランザクションでの更新をポーリングするクライアント

たとえば、大きなデータウェアハウスのクエリがレスポンスに平均で 2 分かかる場合、もしクライアントがデータ利用できないならば、エクスポネンシャルバックオフを使用して、最長でも 2 分間隔で API をポーリングします。クライアントが意図した以上に頻繁なポーリングとならないようにする一般的なパターンは 2 つあります。スロットリングと、安全に再度ポーリングできる時刻を決めるタイムスタンプです。

スロットリングについては、API Gateway を利用すると、使用量プランに基づいて API キーを発行できます。使用量プランはコンシューマーから API へのアクセスを指定された間隔に制限します。

タイムスタンプについては、ポーリングされるシステムが、コンシューマーがもう一度ポーリングできるようになるタイムスタンプまたは期間を示す特別なフィールドを返します。このアプローチは、コンシューマーがこれを尊重して適切に使用するという善意のシナリオに基づいており、不正使用を考慮する場合にはスロットリングも採用してさらに完全に実装することもできます。

レビュー

サーバーレスアプリケーションに該当するパフォーマンスの効率性については、AWS Well-Architected フレームワークのホワイトペーパーでレビューエリアのベストプラクティスをご覧ください。

モニタリング

サーバーレスアプリケーションに該当するパフォーマンスの効率性については、AWS Well-Architected フレームワークのホワイトペーパーでモニタリングエリアのベストプラクティスをご覧ください。

トレードオフ

サーバーレスアプリケーションに該当するパフォーマンスの効率性については、AWS Well-Architected フレームワークのホワイトペーパーでトレードオフエリアのベストプラクティスをご覧ください。

主要な AWS サービス

パフォーマンス効率向けの主な AWS サービスは、DynamoDB Accelerator、API Gateway、Step Functions、NAT ゲートウェイ、Amazon VPC、および Lambda です。

リソース

セキュリティに関する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 & ブログ

- AWS Lambda のよくある質問[56]
- AWS Lambda 関数を使用する際のベストプラクティス[57]
- AWS Lambda: 仕組み[58]
- AWS Lambda のコンテナの再利用について[59]
- Amazon VPC 内のリソースにアクセスできるように Lambda 関数を構成する[60]
- API キャッシュを有効にして応答性を強化する[61]

- DynamoDB: グローバルセカンダリインデックス[62]
- Amazon DynamoDB Accelerator (DAX)[63]
- 開発者ガイド: Kinesis Streams[64]

コスト最適化の柱

コストの最適化の柱には、システムのライフサイクル全体に及ぶシステムの改良と改善の継続的なプロセスが含まれます。PoC (概念実証) の極めて早い段階の初期設計から本番環境の継続的な運営に至るまで、本書のプラクティスを実践することで、ビジネスの成果を得られコストを最小限に抑える、コストに配慮したシステムを構築および運用できるようになります。それにより、お客様のビジネスで投資収益率を最大化することができるようになります。

定義

クラウド上でのコストの最適化には、次の 4 つのベストプラクティスの領域があります。

- コスト効率に優れたリソース
- 需要と供給の一致
- 費用の把握
- 継続した最適化

他の柱と同様に、考慮すべきトレードオフがあります。たとえば、最適化する必要があるのは市場投入までのスピードですか、またはコストですか。 場合によっては、目先のコスト最適化に注力するのではなく、迅速な市場投入、新機能の提供、または単純に期日の遵守といった、スピードの最適化が最善であることがあります。設計上の決定は、検証されたデータに基づくのではなく、スピード重視で行われる場合があります。コストが最適化されたデプロイのためにベンチマークに多く時間を費やすよりも、「万が一」を想定して過補償した方が良いと思える場合があるためです。この結果、過剰にプロビジョニングされ、最適化されていないデプロイがよく発生します。以下のセクションでは、デプロイにおける初期コストの最適化および継続的なコストの最適化についての技術的および戦略的なガイダンスを示します。

一般的に、サーバーレスアーキテクチャでは、いくつかのサービス(AWS Lambda など) でアイドル中に料金が発生しないという事実から、コストを削減する傾向があります。ただし、特定のベストプラクティスに従いトレードオフを行うことで、このようなソリューションのコストをさらに削減できるようになります。

ベストプラクティス

SERVCOST1: Lambda に最適なメモリ割り当てを決定する際の戦略はどのようにしていますか？

コスト効率に優れたリソース

サーバーレスアーキテクチャは、正しいリソース割り当ての面において、より管理が容易です。アーキテクチャの設計時にサイズ設定がほとんど不要であるという事実と、AWS Lambda などのサービスの需要に基づいてスケールできる機能が、クラスター、インスタンスのサイズ設定、ストレージなど、決定しなければならないことの数を減らしてくれます。

ただし、パフォーマンス効率の柱と運用上の優秀性の柱のセクションで説明したように、最高のコスト/パフォーマンスが得られるようにするために、テストシナリオに基づいた最適なメモリ割り当てが必要になります。

また、運用上の優秀性の柱のセクションで説明したように、メモリとタイムアウトを細かく設定すると、パフォーマンスや運用上の手順だけでなく、コスト削減の可能性でも大きな影響が及びます。

Lambda 関数で適切にメモリを割り当てると、実行時間が減り、コストが削減されます。また、CPU 割り当ては、ユーザーが割り当てたメモリ量に直接関連しています。ユーザーが割り当てるメモリが大きいほど、CPU の割り当ても増え、パフォーマンスに影響が出ます。

コストを削減するには使用するメモリ量を最低限に抑えるのが完璧な戦略のように思われますが、メモリの量が減るということは、Lambda 関数の実行時間が長くなることを意味します。したがって、100 ミリ秒単位の増分請求ディメンションでは、よりコスト高になる可能性があります。

需要と供給の一致

AWS サーバーレスアーキテクチャは、需要に応じてスケールされるように設計されているため、プロビジョニングの過剰や不足を心配する必要はありません。

費用の把握

AWS Well-Architected フレームワークで解説されているように、クラウドによってもたらされる優れた柔軟性と俊敏性によって、イノベーションと速いペースの開発、およびデプロイが可能になります。これにより、オンプレミスのインフラストラクチャのプロビジョニングにおける手動作業と時間 (ハードウェア仕様の確認、価格見積もりの交渉、発注書の管理、配送のスケジューリング、リソースのデプロイ) が排除されます。

AWS Well-Architected フレームワークホワイトペーパーを読み、そこで解説されているトピックについて理解を深めるようお勧めします。ただし、そこで触れられている質問の一部は、サーバーレスアーキテクチャには一部適用されないものもあります。その一例としては、AWS Lambda などのリソースはアイドル状態のときには料金がかからないため、廃棄する必要はありません。

未使用の API、Kinesis シャード、DynamoDB テーブル、既存の課題と推奨事項など、その他のシナリオすべてが同様に適用され、それらにおけるサーバーレスアプリケーションに対する固有のプラクティスはありません。

サーバーレスアーキテクチャが成長するにつれ、Lambda 関数、API、ステージ、さらなるアセットの数も増えていきます。このようなアーキテクチャのほとんどで、コストとリソースの面で予算確保と予測を行う必要があります。

AWS Lambda と API Gateway はどちらも、それぞれ関数とステージへのタグ付けをサポートしています。この機能を使用して、AWS の請求書からコストを個別の関数や API に配分し、AWS Cost Explorer でプロジェクト単位のコストを詳細に表示できます。

簡易な実現方法は、あるプロジェクトに属するアセットで同じキーバリュータグを共有し、作成したタグに基づいてカスタムレポートを作成することです。

この機能は、コストの配分だけではなく、どのリソースがどのプロジェクトに属しているかを識別するのにも役立ちます。

継続した最適化

AWS による新しいサービスや機能のリリース時は、アーキテクチャに関する既存の決定事項を見直し、アーキテクチャが引き続き最もコスト効率に優れたものであることを確認するのがベストプラクティスです。お客様のインフラストラクチャがサーバーレスアーキテクチャで実行されるようになったら、Lambda 実行またはログのストレージなどに関してコストの最適化を図るため繰り返し見直すことを推奨します。

SERVCOST 2: Lambda 関数におけるコードのログ記録のための戦略はどのようにしていますか?

AWS Lambda は CloudWatch Logs を使用して実行時の出力を保存します。これを使用して、実行時の問題の識別やトラブルシューティング、またサーバーレスアプリケーションのモニタリングを行います。これらは、取り込みとストレージという 2 つのディメンションで、CloudWatch Logs サービスのコストに影響します。

AWS Lambda に関数をデプロイするときは、コード内の不要な print 文を削除することが重要です。print 文の出力が CloudWatch に取り込まれ、同一コードの取り込みあたりのコストが上昇するためです。必要なときにこれらの出力を維持する適切なアプローチは、logging[65] などツールやライブラリーを使用し、環境変数を通じて必要に応じた妥当なログレベルを設定するといったベストプラクティスに従うことです。こうすることで、明示的にアクティブにされていない限り、取り込みログは INFO になり、DEBUG は取り込まれません。

ログのストレージコストを削減するためには、次の機能を活用することを推奨します。

- AWS Lambda の Amazon CloudWatch Logs グループのログ保持期間を使用する。
- Amazon S3 や Amazon ES など、よりコスト効率の良いプラットフォームにログをエクスポートする。

これら 2 つのアプローチを使用すると、ログストレージの観点からコストを節約できます。また、必要なときには、これらのログをAmazon S3 上で別のツール (Amazon Athena など) を使用して直接調べたり、Amazon ES にアップロードしてトラブルシューティングをしたりすることができます。

SERVCOST 3: コードアーキテクチャは不要な Lambda 関数を実行していませんか?

アーキテクチャを設計する際、Lambda の不要な実行はソリューション全体のコストを増大させます。シンプルな原則に従うことでこれを回避できます。

Lambda 関数は、**データの転送**ではなく**データの変換**に使用する。

Lambda 関数が情報に対して何の変更も実行せずにスタックのあるレイヤーから別のレイヤーに渡すだけであれば、この Lambda 関数はたいてい、他のもっとコスト効率のいいソリューションに置き換えることができます。

API Gateway サービスプロキシや IoT と他の AWS サービス間の直接統合といった機能を使用すると、これらの Lambda 関数のコストの増大と、これらのリソースを管理するときの運用オーバーヘッドの両方を回避できます。

不必要な呼び出しについては、スループットのためのトレードオフとしてレイテンシーが受け入れられる場合、AWS Lambda に対して、 Kinesis とDynamoDB の両方を統合することで、複数リクエストを単一の呼び出しにまとめることができます。これにより、全体の同時実行数、呼び出し数、そしてコストを削減できます。

ほとんどのサーバーレスアーキテクチャでは、エンドユーザーのエントリポイントとして API Gateway を使用します。これは、実装にとらわれない RESTful API の性質によるもので、インフラストラクチャとエンドユーザー間の優れたサービスコントラクトとなります。詳細については、マイクロサービスのシナリオを参照してください。

ここで考慮すべきアプローチがいくつかあります。

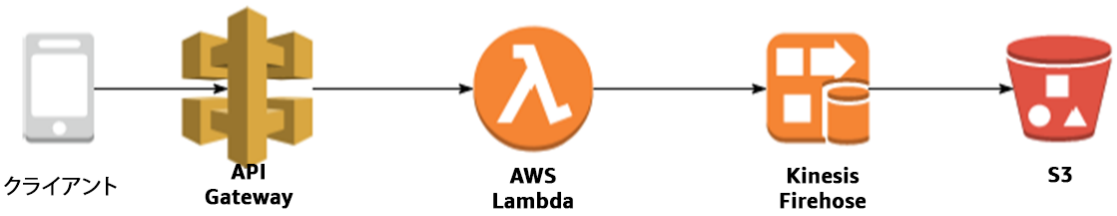


図 27: Kinesis Firehose を使用してデータを Amazon S3 に送信する

このシナリオでは、API Gateway が Lambda 関数を実行し、この関数によってデータが Kinesis Firehose に渡され、さらに Amazon S3 にも送信されます。よって、これらのサービスすべてにコストが発生します。

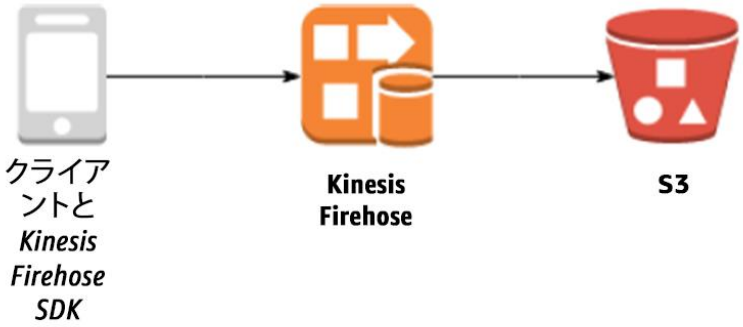
一方で、別のアプローチも考えられます。

図 28: AWS サービスプロキシを実装することで、Amazon S3 にデータを送信するためのコストを減らす

このアプローチでは、API Gateway に AWS サービスプロキシ機能を実装することで、Lambda の使用と不要な呼び出しのコストが取り除かれます。ただし、このアプローチは他のサービス、たとえば、Kinesis とのやり取りをする場合に、各呼び出し単位で取り込みの用のシャードを定義する必要が生じるなど、余分な複雑さをもたらす可能性があります。

また、Kinesis Firehose SDK を使用して、S3 バケットにクライアントから直接データをストリーミングすることも可能で、その場合、API Gateway と AWS Lambda に関連するコストを取り除き、アーキテクチャをまとめて単純化することができます。

Kinesis Firehose SDK を使用して直接ストリーミングすることにより、Amazon S3 へのデータ送信のコストを削減する



もちろん、この実装では RESTful API をアプリケーションで使用する恩恵を享受できませんが、コストに加えてストリーミングのレイテンシーを減らすことができます。個別のユースケースによりますが、これらのアプローチのいずれかが、ワークロードに適合する可能性があります。

VPC やオンプレミス内にあるなどのプライベートエンドポイントやリソースと通信を行う必要があるシナリオがあります。リクエストが、追加のヘッダーやパラメータ、またはペイロードなど、変換を必要としない場合は、Lambda 関数をプライベートコールのプロキシとして使用する代わりに、API Gateway のプライベート統合機能を使用できます。

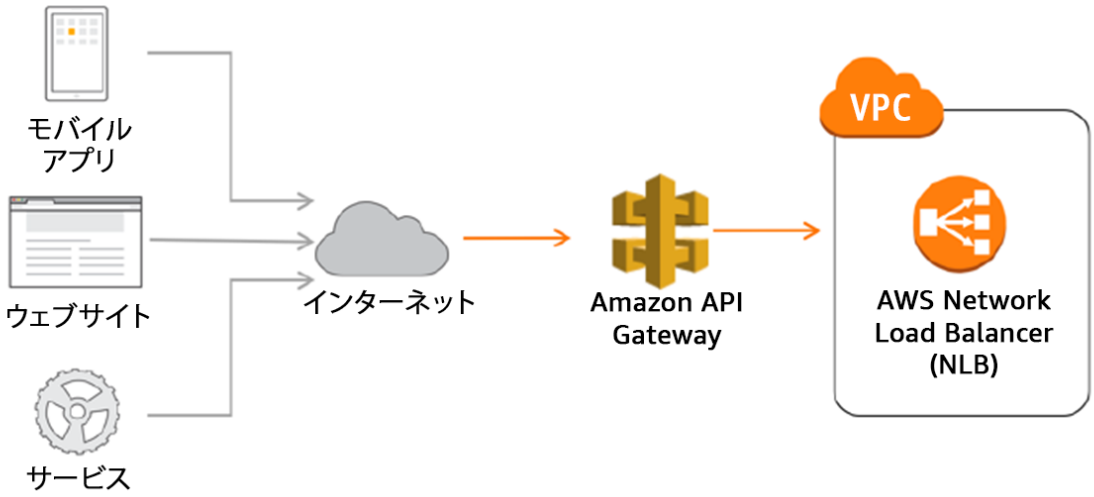


図 30: VPC 内 Lambda より効率よくプライベートリソースにアクセスするための Amazon API Gateway プライベート統合機能

このアプローチでは、API Gateway は入ってくるリクエストを VPC 内の内部ネットワークロードバランサーに送信し、そこから IP アドレスを使用して任意のバックエンド (同じ VPC 内またはオンプレミス) にトラフィックを転送できます。これはコストとパフォーマンスの両方に利点があります。基本的にプライベートバックエンドにリクエストを送信するためにホップを追加する必要がないためです。また、このようなリクエストをプロキシする Lambda 関数実装の必要がなく、API Gateway が提供する認可、スロットリング、キャッシングのしくみを利用できる利点もあります。

発行されたトピックメッセージに応答して AWS Lambda 関数をトリガーするには、Amazon SNS を使用できます。Amazon SNS はメッセージをすべてのトピック受信者に発行するため、SNS メッセージフィルタ処理を使用して受信者の関連性と AWS Lambda の使用量を最適化できます。これにより、イベントが処理される必要がある場合にのみ、受信者に通知が送信されるようにポリシーを適用できます。このアプローチにより、すべてのメッセージが不用意にすべての受信者にブロードキャスト送信されるのを回避できます。こうすることで、このフィルタリングの背後にあるロジックが Lambda 関数コード内ではなくなるため、インフラストラクチャのコストを削減できます。

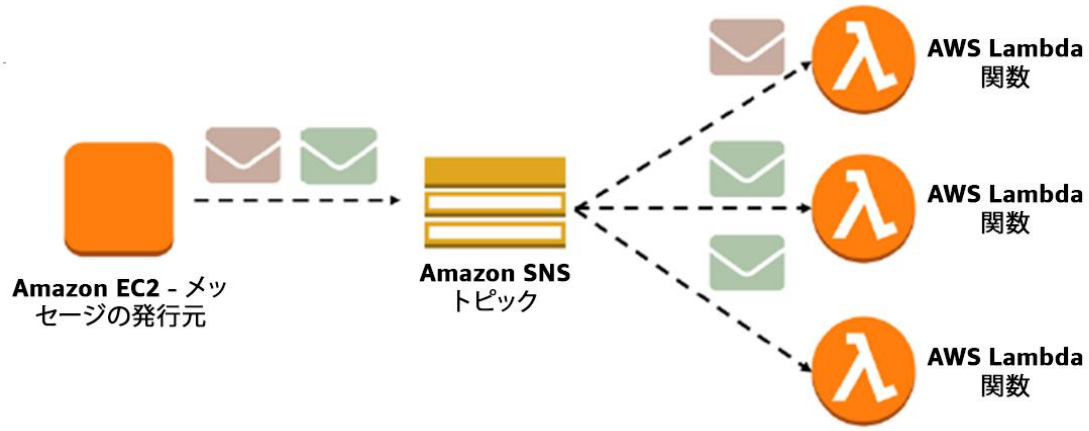


図 31: メッセージ属性のフィルタリングを使用した Amazon SNS

SERVCOST 4: 全体の実行時間を削減するために Lambda 関数をどのように最適化していますか?

サーバーレスアーキテクチャに実装するワークロードの中には、関数の長期的な実行を必要とするものがあります。これらのワークロードでは、一部の関数はリソースが使用できるようになるまで待機することもあります。この待機状態については、所定時間を待機させるよう Lambda コードとして実装できますが、代わりに Step Functions を使用して、待機処理を実装する方法を推奨します。

このパターンでは、リソースが使用できるようになるまで Lambda 関数が一定の時間待機することで料金が発生しアイドル中のリソースが無駄になる代わりに、Step Functions でこの待機処理を実装し、コストを減らすことができます。たとえば、次の図では、AWS Batch ジョブをポーリングし、30 秒ごとに状態を見てジョブが完了したかを確認します。Lambda 関数内にこの待機処理をコーディングする代わりに、次を実装します。poll (*GetJobStatus*) + wait (*Wait3oSeconds*) + decider (*CheckJobStatus*)。

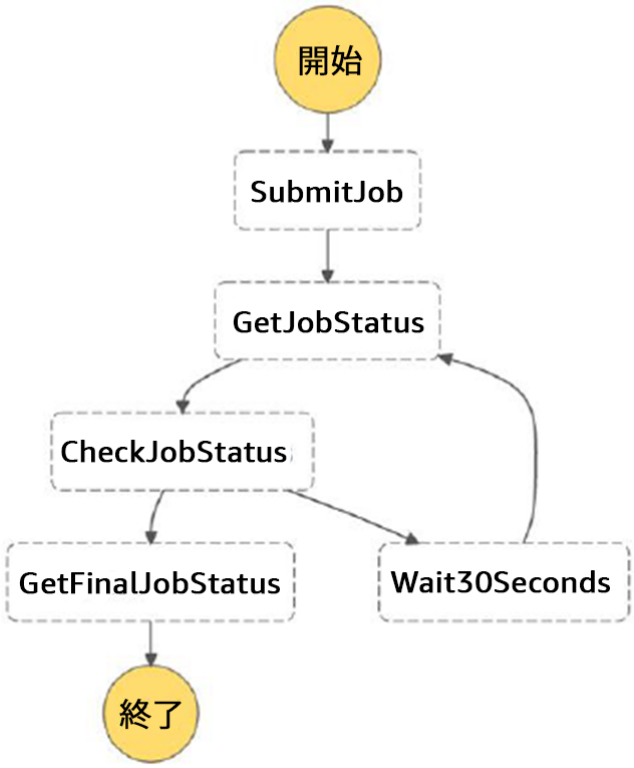


図 32: AWS Step Functions で Wait 状態 を実装する

Step Functions で Wait 状態 を実装しても、さらにコストがかかることはありません。これは、Step Functions の料金モデルが、状態間の遷移数に基づくものであり、単一状態内の処理時間に基づくものではないからです。

最後に、実行時間を減らすようにコード自体を最適化することで、Lambda 関数の実行あたりのコストを削減できます。グローバル変数を使用してデータストアや他のサービス/リソースへの接続を維持すると、パフォーマンスが上がり実行時間が短縮されるため、コストもまた削減されます。詳細については、パフォーマンスの柱のセクションを参照してください。

さらに、オブジェクトを Amazon S3 から取得する場合、他の機能やサービスを使用することで、Lambda 関数に必要なメモリ量と全体の実行時間の両方を削減できます。Athena SQL クエリを使用して、実行に必要な詳細情報を収集すると、オブジェクト取得時間や、変換を要するオブジェクトのサイズが削減されます。

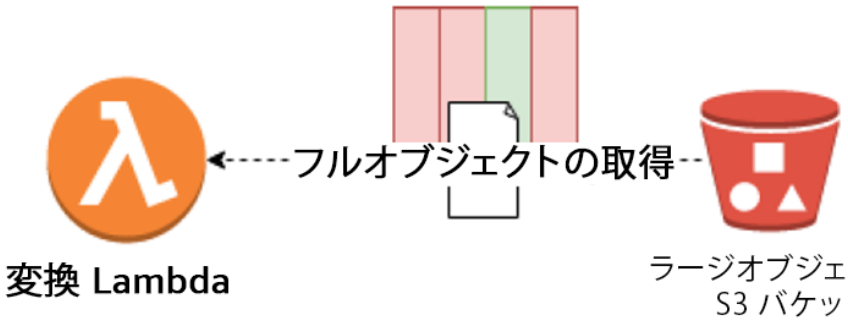


図 33: S3 オブジェクト全体を取得する Lambda 関数

上の図では、Amazon S3 から大きなオブジェクトを取得する場合は、Lambda でのメモリ消費を増やし、それに伴って (関数が必要なデータを変換、反復、または収集するため) 実行時間が増えますが、場合によっては実際に必要なのはこの情報の一部のみであることがわかります。このことは、3 列が赤 (不要なデータ)、1 列が緑 (必要なデータ) で表されています。

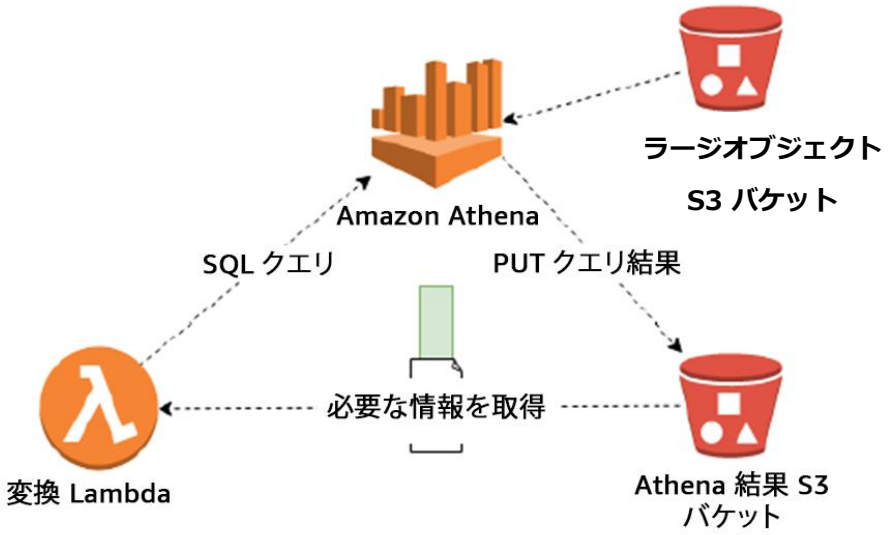


図 34: Athena を使用した Lambda でのオブジェクト取得

上の図では、Athena をクエリして特定のデータを取得することで、取得するオブジェクトのサイズを減らす様子を表しています。また、追加の利点として、Athena はクエリ結果を別の S3 バケットに保存するため、そのコンテンツを再利用できます。さらに、結果が Amazon S3 に到着するたびに、非同期に Lambda を呼び出すこともできます。

同様のアプローチは、S3 Select を使用しても可能です。S3 Select は、アプリケーションがシンプルな SQL 式を使用してオブジェクトからデータのサブセットのみを取得できるようにします。前述の Athena を使用した例のように、Amazon S3から取得するオブジェクトが小さいほど、Lambda 関数の実行時間とメモリ使用量が減ります。

図 35: S3 vs S3 Select を使用した Lambda perf stat

200 秒

```
# Download and process all keys
for key in src_keys:
  response =
s3_client.get_object(Bucket=src_bucket,
Key=key)
  contents = response['Body'].read()
  for line in contents.split('\n')[:-
1]:
      line_count +=1
      try:
          data = line.split(',')
           srcIp = data[0][:8]
             ....
```

95 秒

```
# Select IP Address and Keys
for key in src_keys:
  response =
s3_client.select_object_content
      (Bucket=src_bucket, Key=key,
expression =
        SELECT SUBSTR(obj._1, 1, 8),
obj._2 FROM s3object as obj)
  contents = response['Body'].read()
  for line in contents:
      line_count +=1
      try:
             ....
```

最後に、クライアントが API Gateway からデータを取得するためにかかる時間と、API レスポンスのコストの両方を削減するには、API Gateway の API レスポンスでペイロードの圧縮を有効にすることを推奨します。

リソース

セキュリティに関する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 & ブログ

- CloudWatch Logs の保持期間[66]
- CloudWatch Logs を Amazon S3 にエクスポートする[67]
- CloudWatch Logs を Amazon ES にストリーミングする[68]
- Step Functions のステートマシンで Wait 状態を定義する[69]
- Step Functions を利用した Coca-Cola Vending Pass ステートマシン[70]
- 高スループットのゲノム解析バッチワークフローを AWS に構築する[71]
- Amazon SNS のメッセージフィルタリングを使用して Pub/Sub メッセージングを簡素化する
- S3 Select と Glacier Select

- MapReduce 向け Lambda リファレンスアーキテクチャ

ホワイトペーパー

- サーバーレスアーキテクチャによるエンタープライズのエコノミクスの最適化[7]

まとめ

サーバーレスアプリケーションは開発者の差別化につながらない重労働を大幅に軽減しますが、その一方で適用すべき重要な原則は依然として存在します。

信頼性の面では、定期的に異常系のテストを実行することで、本番環境に達する前に、エラーを見つける可能性が大きくなります。パフォーマンスでは、顧客の期待から逆算していくことで、最適なエクスペリエンスを設計できるようになります。また、AWS にはパフォーマンスを最適化するためのツールが多数あります。コストの最適化では、トラフィックの需要に合わせてリソースをサイズ設定することによりサーバーレスアプリケーション内で不要な無駄を減らすことができます。オペレーションでは、イベントに呼応して、アーキテクチャの自動化を目指すことを推奨します。最後に、アプリケーションをセキュアにすれば、お客様組織の機密情報資産を保護し、あらゆるレイヤーのコンプライアンス要件を満たすようになるでしょう。

サーバーレスアプリケーションの状況は、今後もツールとプロセスが成長と成熟していくエコシステムの中で、進化していくことでしょう。そうした進化が顕著になったときは、本書を更新し、皆さんのサーバーレスアプリケーションが Well-Architected となるようお手伝いします。

作成者

本書の執筆に当たり、次の人物および組織が寄稿しました。

- Adam Westrich: AWS プリンシパルソリューションアーキテクト
- Mark Bunch: AWS エンタープライズソリューションアーキテクト
- Ignacio Garcia Alonso: AWS ソリューションアーキテクト
- Heitor Lessa: AWSシニアスペシャリストソリューションアーキテクト
- Philip Fitzsimons: AWSシニアマネージャー、AWS Well-Architected
- Dave Walker: AWSシニアスペシャリストソリューションアーキテクト
- Richard Threlkeld: AWSシニアプロダクトマネージャーモバイル

- Julian Hambleton-Jones: AWSシニアソリューションアーキテクト

参考資料

詳細については、以下を参照してください。

- AWS Well-Architected フレームワーク[73]

ドキュメントの改訂

日付	説明
2018 年 11 月	Alexa およびモバイル向けの新規シナリオ、および新機能とベストプラクティスの進化を反映させた全体の更新。
2017 年 11 月	初版。

注釈

[1]https://aws.amazon.com/well-architected

[2]http://d0.awsstatic.com/whitepapers/architecture/AWS_Well-Architected_Framework.pdf

[3]https://github.com/alexcasalboni/aws-lambda-power-tuning

[4]http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/BestPractices.html

[5]http://docs.aws.amazon.com/elasticsearch-service/latest/developerguide/es-managedomains.html

[6]https://www.elastic.co/guide/en/elasticsearch/guide/current/scale.html

[7]http://docs.aws.amazon.com/streams/latest/dev/kinesis-record-processor-scaling.html

[8]https://d0.awsstatic.com/whitepapers/whitepaper-streaming-data-solutions-on-aws-with-amazon-kinesis.pdf

[9]http://docs.aws.amazon.com/kinesis/latest/APIReference/API PutRecords.html

[10]http://docs.aws.amazon.com/ streams/latest/ dev/kinesis-record-processor-duplicates.html

[11]http://docs.aws.amazon.com/lambda/latest/dg/best-practices.html#stream-events

[12]http://docs.aws.amazon.com/apigateway/latest/developerguide/api- gateway-api-usage-plans.html

[13]http://docs.aws.amazon.com/apigateway/latest/developerguide/stage-variables.html

[14]http://docs.aws.amazon.com/lambda/latest/dg/env variables.html

[15]https://github.com/awslabs/serverless-application-model

[16]https://aws.amazon.com/blogs/aws/latency-distribution-graph-in-aws-x-ray/

[17]http://docs.aws.amazon.com/lambda/latest/dg/lambda-x-ray.html

[18]http://docs.aws.amazon.com/systems-manager/latest/userguide/systems-manager-paramstore.html

[19]https://aws.amazon.com/blogs/compute/ continuous-deployment-for-serverless-applications/

[20]https://github.com/awslabs/aws-serverless-samfarm

[21]https://do.awsstatic.com/whitepapers/DevOps/practicing-continuous-integration-continuous-delivery-on-AWS.pdf

[22]https://aws.amazon.com/serverless/developer-tools/

[23]http://docs.aws.amazon.com/lambda/latest/dg/with-s3-example-create-iam-role.html

[24]http://docs.aws.amazon.com/apigateway/latest/developerguide/api-gateway-method-request-validation.html

[25]http://docs.aws.amazon.com/apigateway/latest/developerguide/use-custom-authorizer.html

[26]https://aws.amazon.com/blogs/compute/secure-api-access-with-amazon-cognito-federated-identities-amazon-cognito-user-pools-and-amazon-api-gateway/

[27]http://docs.aws.amazon.com/lambda/latest/dg/vpc.html

[28]https://aws.amazon.com/pt/articles/using-squid-proxy-instances-for-web-service-access-in-amazon-vpc-another-example-with-aws-codedeploy-and-amazon-cloudwatch/

[29]https://www.owasp.org/images/o/o8/OWASP SCP Quick Reference Guide v2.pdf

[30]https://dQ.awsstatic.com/whitepapers/Security/AWS Security Best Practices.pdf

[31]https://www.twistlock.com/products/serverless-security/

[32]https://snyk.io/

[33]https://www.owasp.org/index.php/OWASP Dependency Check

[34]https://aws.amazon.com/answers/account-management/limit-monitor/

[35]http://theburningmonk.com/2017/07/applying-the-saga-pattern-with-aws-lambda-and-step-functions/

[36]http://docs.aws.amazon.com/lambda/latest/dg/limits.html

[37]http://docs.aws.amazon.com/apigateway/latest/ developerguide/limits.html#api-gateway-limits

[38]http://docs.aws.amazon.com/streams/latest/dev/service-sizes-and-limits.html

[39]http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Limits.html

[40]http://docs.aws.amazon.com/step-functions/latest/dg/limits.html

[41]https://aws.amazon.com/blogs/compute/error-handling-patterns-in-amazon-api-gateway-and-aws-lambda/

[42]https://aws.amazon.com/blogs/compute/serverless-testing-with-aws-lambda/

[43]http://docs.aws.amazon.com/lambda/latest/dg/monitoring-functions-logs.html

[44]http://docs.aws.amazon.com/lambda/latest/dg/versioning-aliases.html

[45]http://docs.aws.amazon.com/apigateway/latest/developerguide/stages.html

[46]http://docs.aws.amazon.com/general/latest/gr/api-retries.html

[47]http://docs.aws.amazon.com/step-functions/latest/dg/tutorial-handling-error-conditions.html#using-state-machine-error-conditions-step-4

[48]http://docs.aws.amazon.com/xray/latest/devguide/xray-services-lambda.html

[49]http://docs.aws.amazon.com/lambda/latest/dg/dlq.html

[50]https://aws.amazon.com/blogs/compute/ error-handling-patterns-in-amazon-api-gateway-and-aws-lambda/

[51]http://docs.aws.amazon.com/step-functions/latest/dg/amazon-states-language-wait-state.html

[52]http://microservices.io/patterns/data/saga.html

[53]http://theburningmonk.com/2017/07/applying-the-saga-pattern-with-aws-lambda-and-step-functions/

[54]https://d0.awsstatic.com/whitepapers/microservices-on-aws.pdf

[55]http://docs.aws.amazon.com/lambda/latest/dg/best-practices.html

[56]https://aws.amazon.com/lambda/faqs/

[57]http://docs.aws.amazon.com/lambda/latest/dg/best-practices.html

[58]http://docs.aws.amazon.com/lambda/latest/dg/lambda-introduction.html

[59]https://aws.amazon.com/blogs/compute/container-reuse-in-lambda/

[60]http://docs.aws.amazon.com/lambda/latest/dg/vpc.html

[61]http://docs.aws.amazon.com/apigateway/latest/developerguide/api-gateway-caching.html

[62]http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/GSI.html

[63] https://aws.amazon.com/ dynamodb/dax/

[64]http://docs.aws.amazon.com/ streams/latest/dev/amazon-kinesis-streams.html

[65]https://docs.python.org/2/library/logging.html

[66]http://docs.aws.amazon.com/AmazonCloudWatch/latest/logs/SettingLogRetention.html

[67]http://docs.aws.amazon.com/AmazonCloudWatch/latest/logs/S3ExportTasksConsole.html

[68]http://docs.aws.amazon.com/AmazonCloudWatch/latest/logs/CWL_ES_Stream.html

[69]http://docs.aws.amazon.com/step-functions/latest/dg/amazon-states-language-wait-state.html

[70]https://aws.amazon.com/blogs/aws/things-go-better-with-step-functions/

[71]https://aws.amazon.com/blogs/compute/building-high-throughput-genomics-batch-workflows-on-aws-workflow-layer-part-4-of-4/

[72]https://dQ.awsstatic.com/whitepapers/optimizing-enterprise-economics-serverless-architectures.pdf

[73]https://aws.amazon.com/well-architected