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

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

Table of Contents

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

公開日: 2019 年 12 月 (改訂履歴 (p. 61))

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。このドキュメントでは、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説明し、ワークロードがベストプラクティスに従って設計されていることを確認するための重要な要素を示します。

はじめに

AWS Well-Architected フレームワークの目的は、AWS でシステムを構築する際の選択肢の#所と短所をお客様が理解できるように支援することです。効率が良く、費用対効果が#く、安全で信頼のおけるクラウド対応システムを設計して運#するために、アーキテクチャに関するベストプラクティスをこのフレームワークに従って学ぶことができます。フレームワークにより、アーキテクチャをベストプラクティスに照らし合わせて一貫的に測定し、改善すべき点を特定する手段を提供します。当社では、Well-Architected システムを持つことが、ビジネスで成功を収める可能性を大幅に向上させると考えています。

この「レンズ」では、AWS クラウド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、および構築する方法に焦点を当てます。ここでは簡潔に、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のみを取り上げてみました。実際にアーキテクチャを設計する際は、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や疑問点も考慮してください。AWS Well-Architected フレームワークホワイトペーパーを読むことをお勧めします。

本書は、最高技術責任者 (CTO)、設計者、開発者、運用チームメンバーなどの技術担当者のみなさまを対象に書かれています。本書を一通り読んでいただくことで、サーバーレスアプリケーションのアーキテクチャを設計するための 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、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、コスト最適化という 5 本の柱を基本としています。サーバーレスワークロードの場合、AWS には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) があり、サーバーレスアプリケーションの堅牢なアーキテクチャを設計できます。このセクションでは、このドキュメント全体で使用されるサービスの概要を示します。サーバーレスのワークロードを構築する際に考慮すべき領域は 7 つあります。

トピック

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、アクセスをコントロールし、リクエストが適切に許可されるようにします。これには、ビジネスロジックがデプロイおよび実行されるランタイム環境が含まれます。

AWS Lambda では、関数レイヤーでのマイクロサービスアーキテクチャ、デプロイ、実行の管理をサポートするマネージド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を実行できます。

Amazon API Gateway では、完全管理型の REST API を実行でき、Lambda と統合してトラフィック管理、認証とアクセス制御、モニタリング、API バージョニングなどのビジネスロジックを実行できます。

AWS Step Functions は、複数のステップに分割するか、Amazon Elastic Compute Cloud (Amazon EC2) インスタンスまたはオンプレミスで実行されているワーカーを呼び出すことで、調整、状態、および関数の連鎖、さらには Lambda 実行制限内でサポートされていない長時間実行の組み合わせを含むサーバーレスワークフローを調整します。

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。ビジネスロジックが必要とする状態を保存するための安全なメカニズムを提供します。データの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、永続的ストレージのためのマネージド型 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支援します。DynamoDB Streams と組み合わせると、Lambda 関数を呼び出すことによって、DynamoDB テーブルの変更にほぼリアルタイムで応答できます。DynamoDB Accelerator (DAX) では、DynamoDB 用の可用性の高いインメモリキャッシュが追加され、パフォーマンスはミリ秒からマイクロ秒まで最大 10 倍向上します。

Amazon Simple Storage Service (Amazon S3) では、可用性の高いキー値ストアを提供することで、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。このストアから、Amazon CloudFront などのコンテンツ配信ネットワーク (CDN) 経由で静的アセットを提供できます。

Amazon Elasticsearch Service (Amazon ES) を使用すると、Elasticsearch を簡単にデプロイ、保護、運用、スケールして、ログ分析、全文検索、アプリケーション監視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完全マネージド型サービスです。

AWS AppSync は、リアルタイムおよびオフライン機能を備えたマネージド型の GraphQL サービスで、アプリケーションの開発を簡単にするエンタープライズグレードのセキュリティコントロールを備えています。AWS AppSync は、アプリケーションおよびデバイスを DynamoDB、Amazon ES、Amazon S3 などのサービスに接続するための、データ駆動型の API と一貫したプログラミング言語を提供します。

メッセージングおよび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、ストリーミングデータのリアルタイム分析と処理を管理します。

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の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 パターン用の完全マネージド型メッセージングサービスを提供します。

Amazon Kinesis を使用すると、リアルタイムのストリーミングデータを簡単に収集、処理、分析できます。Amazon Kinesis Data Analytics では、標準 SQL の実行や、SQL を使用したストリーミングアプリケーション全体の構築が可能です。

Amazon Kinesis Data Firehose では、ストリーミングデータをキャプチャして変換し、Kinesis Data Analytics、Amazon S3、Amazon Redshift、Amazon ES にロードします。これにより、既存のビジネスインテリジェンスツールでほぼリアルタイムの分析が可能になります。

ユーザー管理と ID レイヤー

ワークロードのユーザー管理と ID レイヤーは、ワークロードのインターフェイスの外部顧客と内部顧客の両方に ID、認証、承認を提供します。

Amazon Cognito では、サーバーレスアプリケーションに対してサインアップ、サインイン、データ同期を簡単に追加できます。Amazon Cognito ユーザープールは内蔵サインイン画面と、Facebook、Google、Amazon、Security Assertion Markup Language (SAML) との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 ID を使用すると、サーバーレスアーキテクチャの一部である AWS リソースへのスコープ付きのアクセスを安全に提供できます。

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顧客への接続を管理します。異なる地理的場所に居住する外部顧客への効率的な配信方法です。

Amazon CloudFront は、低レイテンシーで高速な転送速度で、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 を提供します。

システムの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、時間の経過に伴う動作と動作のコンテキスト認識を作成します。デプロイレイヤーは、リリース管理プロセスを通じてワークロードの変更を昇格させる方法を定義します。

Amazon CloudWatch を使用すると、使用するすべての AWS のサービスのシステムメトリクスにアクセスし、システムレベルとアプリケーションレベルのログを統合し、ビジネスキーパフォーマンス指標 (KPI) を特定のニーズに合わせてカスタムメトリクスとして作成できます。プラットフォームで自動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が提供され、リクエストをエンドツーエンドで視覚化することで、パフォーマンスのボトルネックを簡単に識別できるため、サーバーレスアプリケーションを分析およびデバッグできます。

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、デプロイに使用される AWS CloudFormation の拡張機能です。AWS SAM CLI では、Lambda 関数をローカルで開発するときにデバッグサイクルを高速化することもできます。

デプロイアプローチ

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は、変更によってコンシューマーのサービス契約違反とならないようにすることです。API 所有者がサービス契約違反となるような変更を行い、コンシューマーがその準備を整えていない場合、エラーが発生する可能性があります。

どのコンシューマーが API を使用しているかを認識することは、デプロイの安全性を確保するための最初のステップです。コンシューマーとその使用状況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に基づく決定を下すことができます。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キャプチャする効果的な方法であり、API に重大な変更が行われた場合に問い合わせの形としてよく使用されます。

違反に至る変更によるリスクを回避したいお客様の中には、API のクローンを作成し、顧客を別のサブドメイン (例 :v2.my-service.com) にルーティングして、既存のコンシューマーに影響を与えないようにすることを選択することがあります。このアプローチでは、新しいサービス契約で新しいデプロイが可能ですが、デュアルの API (および後続のバックエンドインフラストラクチャ) を維持するオーバーヘッドには追加のオーバーヘッドが必要であるというトレードオフがあります。

この表は、デプロイに対するさまざまなアプローチを示しています。

デプロイ	コンシューマーの影響	ロールバック	イベントモデルの要因	デプロイ速度
すべてを一度に実行	すべてを一度に実行	古いバージョンを再デプロイ	低同時実行率の任意のイベントモデル	即時
ブルー/グリーン	前もってあるレベルの本番環境テストですべてを一度に実行	トラフィックを以前の環境に元に戻す	中程度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の向上	数分 ~ 数時間の検証を行い、その後すぐに顧客にも対応する
Canary/線形	1 ~ 10% の一般的な初期トラフィックシフト、その後段階的に増加、または一度にすべて増加	トラフィックの 100% を以前のデプロイに戻す	高い同時実行ワークロードに適した	数分 ~ 数時間

トピック

- すべてを一度にデプロイ (p. 6)

すべてを一度にデプロイ

すべてを一度にデプロイするには、既存の設定に加えて変更を加える必要があります。このデプロイスタイルならば、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、変更サイクル中にトランザクションを調整するための労力を大幅に小さくできます。この対p@宇のデプロイスタイルは、労力が少なくて済み、低同時実行モデルへの影響はほとんどありませんが、ロールバックに関してはリスクが大きくなり、たいていの場合ダウンタイムが発生してしまいます。このデプロイモデルを使用するシナリオの例は、ユーザーへの影響が最小限である開発環境です。

ブルー/グリーンデプロイメント

別のトラフィック移行パターンは、ブルー/グリーンデプロイを有効にすることです。このほぼダウンタイムがゼロのリリースにより、ロールバックが必要な場合に備えて、古い本番環境 (blue) をウォーム状態に維持しながら、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、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、このデプロイのスタイルは効果的な手法となります。Blue/Green デプロイはダウンタイムを減らすように設計されているため、多くのお客様が本稼働環境の変更にこのパターンを採用しています。

ステートレスとべき等性のベストプラクティスに従った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へのアフィニティがないため、このデプロイスタイルを使用可能です。必要に応じて、作業環境に簡単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変更にバイアスする必要があります。

ロールバックが必要かどうかを知るには、適切なインジケータが必要です。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お勧めします。このメトリクスは、1 秒間隔でモニタリングし、下向きの傾向をすばやくキャプチャできます。CloudWatch アラームとともに使用すると、加速されたロールバックを有効にできます。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。

Canary デプロイ

Canary デプロイは、コントロールされた環境でソフトウェアの新しいリリースを活用し、高速デプロイサイクルを可能にする、増え続ける方法です。Canary デプロイでは、少数のユーザーに対する影響を分析するために、新しい変更に対する少数のリクエストをデプロイします。新しいデプロイの基盤となるインフラストラクチャのプロビジョニングとスケーリングについて心配する必要がなくなったため、AWS クラウドはこの導入を促進しています。

API Gateway の Canary デプロイ では、コンシューマーに対して同じ 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されるトラフィックの割合や、コントロールされるトラフィックのカットオーバーをコントロールすることもできます。Canary デプロイの実用的なシナリオは、新しいウェブサイトとなる場合があります。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スルー率をモニタリングできます。

Lambda バージョン管理

すべてのソフトウェアと同様に、バージョニングを維持することで、以前に機能していたコードをすばやく把握できるだけでなく、新しいデプロイが失敗した場合に以前のバージョンに戻すこともできます。Lambda では、個々の Lambda 関数に対して 1 つ以上の変更不可能なバージョンを発行できます。これにより、以前のバージョンは変更できなくなります。Lambda 関数の各バージョンには一意の 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 に記録されるとおりに監査できます。本番稼働環境でのベストプラクティスとして、お客様はバージョニングを有効にして、信頼性の高いアーキテクチャを最大限に活用する必要があります。

デプロイ操作を簡素化し、エラーのリスクを減らすために、Lambda エイリアスは、開発、ベータ、本稼働など、開発ワークフローでさまざまなバリエーションの Lambda 関数を有効にします。この例は、Lambda との API Gateway 統合が本番稼働エイリアスの ARN を指す場合です。本番稼働用エイリアスは Lambda バージョンを指します。この手法の価値は、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を可能にすることです。これは、発信者側の設定内の Lambda エイリアスは静的なままであるため、行う変更が少なくなるためです。

一般的な設計原則

Well-Architected フレームワークは、クラウド上における適切な設計を可能にする一般的な設計の原則を提供します。

- スピーディ、シンプル、単数: 関数は簡潔で短く、単一の目的であり、その環境はリクエストのライフサイクルに応じて存続できます。トランザクションはコスト意識が効率的であるため、より高速な実行が推奨されます。
- 合計リクエストではなく同時リクエストを検討する: サーバーレスアプリケーションは同時実行モデルを利用し、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。
- 共有なし: 関数のランタイム環境と基盤となるインフラストラクチャは存続期間が短いため、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、非常に高い耐久性の要件には永続的ストレージが優先されます。
- ハードウェアアフィニティがないと仮定する: 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、CPU フラグとしてハードウェアに依存しないコードや依存関係を活用すると、一貫して利用できない場合があります。
- 関数ではなくステートマシンでアプリケーションをオーケストレーションする: コード内で Lambda 実行を連鎖し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されたアプリケーションになります。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 オブジェクトやデータベースへの更新を書き込むなどのイベントでは、ビジネス機能に応じてトランザクションを実行できます。この非同期イベント動作は、多くの場合、コンシューマーに依存せず、無駄のないサービス設計を確保するためにジャストインタイム処理を推進します。
- 失敗と重複を考慮した設計: リクエスト/イベントからトリガーされる運用は、失敗が発生する可能性があり、特定のリクエスト/イベントを複数回配信できるので、べキ等元である必要があります。ダウンストリーム呼び出しの適切な再試行を含めます。

シナリオ

このセクションでは、多くのサーバーレスアプリケーションで共通する 5 つの主要なシナリオと、それらが AWS のサーバーレスアプリケーションワークロードの設計とアーキテクチャに与える影響について説明します。これらのシナリオごとの前提条件、設計の一般的なドライバー、およびこれらのシナリオの実装方法のリファレンスアーキテクチャについて説明します。

トピック

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、コンシューマーのために再利用可能なサービスとしてビジネスコンテキストをどのように配信できるかを検討しています。具体的な実装は個々のユースケースに合わせて調整されますが、マイクロサービス全体で共通のテーマがいくつかあります。これにより、実装が安全で耐障害性に優れ、顧客に最適なエクスペリエンスを提供するように構築されます。

AWS でサーバーレスマイクロサービスを構築すると、サーバーレス機能自体を利用できるだけでなく、他の AWS のサービスや機能、AWS や AWS パートナーネットワーク (APN) ツールのエコシステムも利用できます。サーバーレステクノロジーは、耐障害性を備えたインフラストラクチャ上に構築されており、ミッションクリティカルなワークロードに対して信頼性の高いサービスを構築できます。ツールのエコシステムにより、ビルドの合理化、タスクの自動化、依存関係の調整、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 のサーバーレスツールは従量課金制であるため、ビジネスに合わせてサービスを拡張し、エントリフェーズやピーク時以外の時間帯にコストを抑えることができます。

特性:

- レプリケートが簡単で、高いレベルの耐障害性と可用性を備える、安全で操作が簡単なフレームワークが必要です。
- 使用状況とアクセスパターンをログに記録して、お客様の使用状況をサポートするためにバックエンドを継続的に改善したいと考えています。
- お客様はプラットフォームでできる限りマネージドサービスを活用したいと考えています。これにより、セキュリティやスケーラビリティなど、一般的なプラットフォームの管理に伴う手間のかかる作業が軽減されます。

リファレンスアーキテクチャ

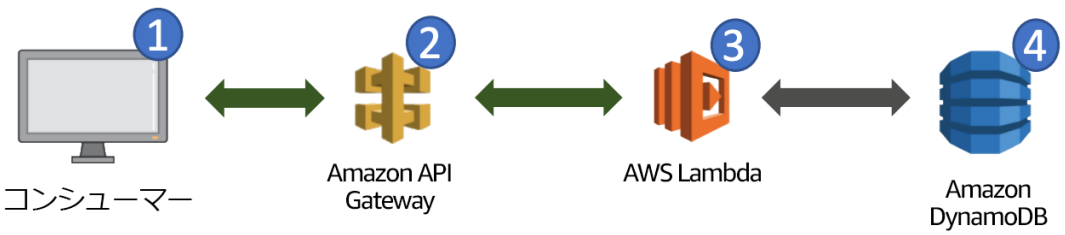

図 1: RESTful マイクロサービスのリファレンスアーキテクチャ

1. 顧客は HTTP API 呼び出しを行うことで、マイクロサービスを活用します。理想的には、コンシューマーは API に対して緊密に拘束されたサービス契約を締結し、サービスレベルと変更管理に関して一貫した期待を達成する必要があります。
2. Amazon API Gateway は、RESTful HTTP リクエストとお客様への応答をホストします。このシナリオでは、API Gateway は組み込みの承認、スロットリング、セキュリティ、耐障害性、リクエスト/レスポンスのマッピング、およびパフォーマンスの最適化を提供します。
3. AWS Lambda には、受信 API 呼び出しを処理し、永続的ストレージとして DynamoDB を活用するビジネスロジックが含まれています。
4. Amazon DynamoDB は、マイクロサービスデータを永続的に保存し、需要に応じてスケールします。マイクロサービスは多くの場合、1 つのことをうまく実行するように設計されているため、スキーマレスの NoSQL データストアは定期的に組み込まれています。

設定に関する注意事項:

- API Gateway のログ記録を活用して、マイクロサービスのコンシューマーアクセス動作の可視性を理解します。この情報は Amazon CloudWatch Logs に表示され、Log Pivots を通じてすばやく表示したり、CloudWatch Logs Insights で分析したり、Amazon ES や Amazon S3 などの検索可能なエンジン (Amazon Athena を使用) に入力したりできます。提供される情報により、次のような主要な可視性が得られます。
  - バックエンドの近接性に基づいて地理的に変化する可能性がある一般的な顧客の所在地を理解する
  - 顧客の入力リクエストがデータベースのパーティション化方法に与える影響を理解する
  - セキュリティフラグとなる異常な動作のセマンティクスについて
  - エラー、レイテンシー、キャッシュヒット/ミスを理解して設定を最適化する

このモデルは、デプロイと保守が簡単なフレームワークと、ニーズの拡大に応じてスケールできる安全な環境を提供します。

Alexa スキル

Alexa スキルキットを使用すると、デベロッパーは自然で魅力的な音声やビジュアルエクスペリエンスを構築することで、Alexa の機能を拡張できます。成功するスキルは習熟形成であり、ユーザーは独特なものを提供し、新しく、斬新で、摩擦のない方法で価値をもたらすからです。

ユーザーからのフラストレーションの最大の原因は、スキルが期待どおりに行動せず、必要なものを達成する前に複数のやり取りが必要になる場合です。音声インタラクションモデルを設計し、それから逆に作業することから始めることが不可欠です。なぜなら一部のユーザーは話すことが少なすぎる、多すぎる、または予期できないことがあるからです。音声設計プロセスには、予期される発話や予期しない発話の作成、スクリプティング、計画が含まれます。

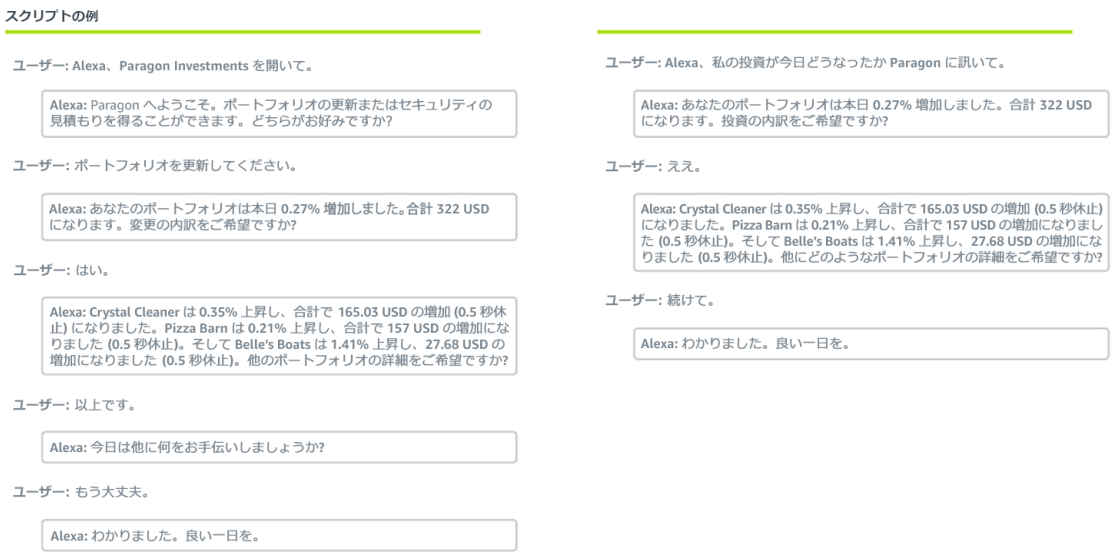

図 2: Alexa スキルの設計スクリプトの例

基本的なスクリプトを念頭に置いて、スキルの構築を開始する前に、次の手法を使用できます。

- 完了までの最短ルートの概要を示す
  - 通常、完了までの最短ルートは、ユーザーがすべての情報とスロットを一度に提供し、関連する場合はアカウントが既にリンクされており、スキルの 1 回の呼び出しで他の前提条件を満たしている場合です。

- 代替パスと決定ツリーの概要を示す
  - 多くの場合、ユーザーの発言には、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ません。フローで、代替パスとユーザーの決定を特定します。

- システムロジックが行う必要がある裏側決定の概要を示す
  - たとえば、新規ユーザーやリピーターユーザーなど、バックグラウンドでのシステム決定を特定します。バックグラウンドシステムチェックによって、ユーザーがフォローするフローが変わる場合があります。

- スキルがユーザーにどのように役立つかの概要を示す
  - ユーザーがスキルを使用して実行できる操作について、ヘルプに明確な指示を含めます。スキルの複雑さに基づいて、ヘルプは 1 つの単純な応答または多数の応答が存在する場合があります。

- アカウントリンクプロセスの概要を示す (存在する場合)
  - 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ときにスキルがどのように応答するかを識別する必要があります。

特性:

- インスタンスやサーバーを管理せずに、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。
- コンテンツをスキルからできるだけ疎結合化します。
- API として公開されている魅力的な音声エクスペリエンスを提供して、さまざまな Alexa デバイス、リージョン、言語にわたる開発を最適化しようとしています。

- ユーザーの需要に合わせてスケールアップおよびスケールダウンし、予期しない使用パターンを処理する伸縮性が求められています。

リファレンスアーキテクチャ

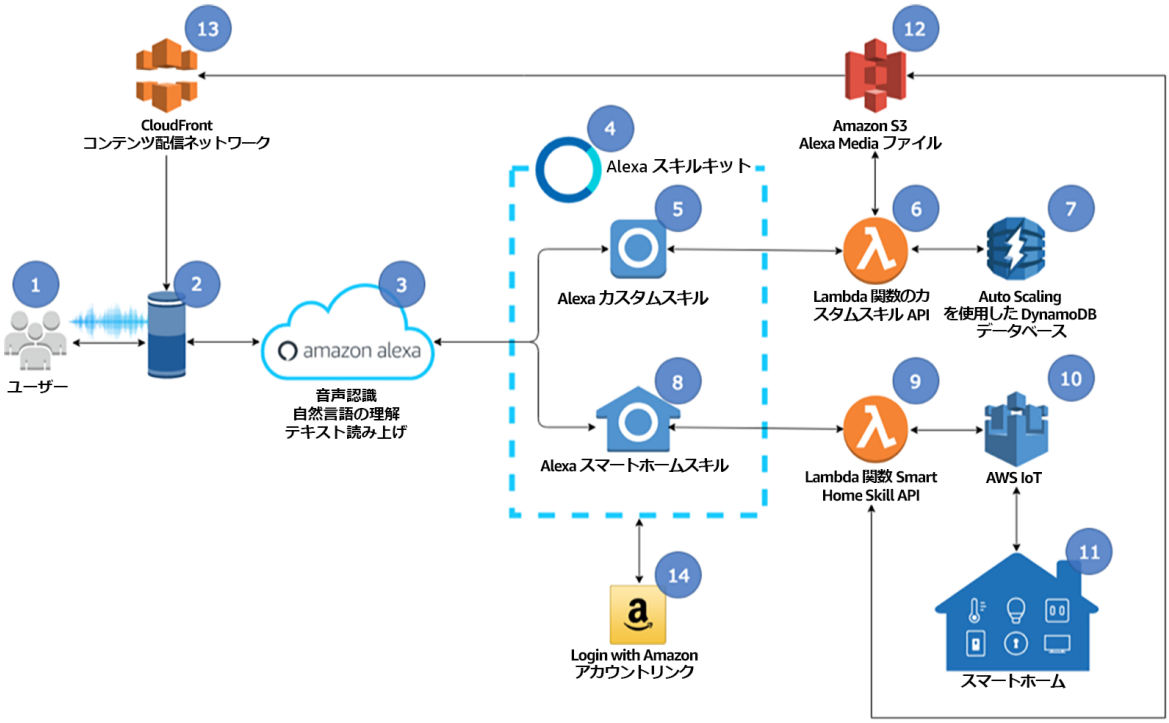

図 3: Alexa スキルのリファレンスアーキテクチャ

1. Alexa ユーザーは、Alexa 対応のデバイスと音声を対話の主要な方法として使用して、Alexa のスキルとやり取りします。
2. Alexa 対応のデバイスは、ウェイクワードをリッスンし、認識されるとすぐにアクティブ化します。サポートされているウェイクワードは、Alexa、Computer、Echo です。
3. Alexa サービスは、自動音声認識 (ASR)、自然言語理解 (NLU)、テキスト読み上げ (TTS) 変換など、Alexa スキルに代わって一般的な音声言語理解 (SLU) 処理を実行します。
4. Alexa スキルキット (ASK) は、Alexa にすばやく簡単にスキルを追加できるようにするセルフサービス API、ツール、ドキュメント、およびコード例のコレクションです。ASK は信頼できる AWS Lambda トリガーで、シームレスな統合を可能にします。
5. Alexa カスタムスキルでは、ユーザーエクスペリエンスをコントロールできるため、カスタムインタラクションモデルを構築できます。最も柔軟なタイプのスキルですが、最も複雑です。
6. Alexa スキルキットを使用した Lambda 関数です。不要な複雑さを回避するスキルをシームレスに構築できます。これを使用して、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、数居の使用量に応じて伸縮自在にスケールできる NoSQL データストアを提供できます。これは、ユーザーの状態とセッションを保持するためのスキルによって一般的に使用されます。
8. Alexa スマートホームスキルでは、スマートホーム API を使用して、照明、サーモスタット、スマートテレビなどのデバイスをコントロールできます。スマートホームスキルは、インタラクションモデルをコントロールできないため、カスタムスキルを構築するほうが簡単です。
9. Lambda 関数は、Alexa サービスからのデバイスの検出とコントロールリクエストに応答するために使用されます。開発者は、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、さまざまなデバイスを制御するために使用しています。

10 AWS Internet of Things (IoT) を使用すると、デベロッパーはデバイスを AWS に安全に接続し、Alexa スキルとデバイス間のやり取りをコントロールできます。

11 Alexa 対応のスマートホームに接続できる、Alexa スキルからのディレクティブについて受信および応答に対応する IoT 接続デバイスの数は無限です。

12 Amazon S3 では、画像、コンテンツ、メディアなどのスキルの静的アセットが保存されます。そのコンテンツは CloudFront を使用して安全に提供されます。

13 Amazon CloudFront コンテンツ配信ネットワーク (CDN) は、地理的に分散したモバイルユーザーにコンテンツを高速に配信する CDN を提供し、Amazon S3 の静的アセットに対するセキュリティメカニズムを備えています。

14. アカウントリンクは、スキルが別のシステムで認証する必要がある場合に必要です。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に関連付けます。

設定に関する注意事項:

- スキルによって Alexa に送信される可能性のあるすべての Alexa スマートホームメッセージについて JSON スキーマに対して検証することで、スマートホームのリクエストとレスポンスのペイロードを検証します。
- Lambda 関数のタイムアウトが 8 秒未満で、その期間内にリクエストを処理できることを確認します。(Alexa Service タイムアウトは 8 秒です。)
- ベストプラクティス[7]に従って、DynamoDB テーブルを作成します。必要な読み取り / 書き込みキャパシティーが分からない場合は、オンデマンドテーブルを使用します。それ以外の場合は、Auto Scaling を有効にしてプロビジョニングされたキャパシティーを選択します。準備が整っているスキルの場合、DynamoDB Accelerator (DAX) によって応答時間が大幅に短縮されます。
- アカウントリンクは、外部システムに保存される可能性のあるユーザー情報を提供できます。この情報を使用して、ユーザーにコンテキストと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を提供するためのアカウントリンクに関するガイドラインがあります。
- スキルベータテストツールを使用すると、スキル開発に関する早期フィードバックを収集したり、スキルのバージョニングを行ったり、すでにライブになっているスキルへの影響を軽減したりできます。
- 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

ユーザーから、モバイルアプリケーションに高速で一貫性がある、機能豊富なユーザーエクスペリエンスを求める声がますます上がっています。同時に、モバイルユーザーパターンは動的で、ピーク使用率は予測不能です。国際的に移動して利用する例も多くあります。

モバイルユーザーからの需要の高まりは、バックエンドインフラストラクチャのコントロールと柔軟性を犠牲にすることなく、シームレスに連携する豊富なモバイルサービスをアプリケーションに求められていることを意味します。以下の機能はモバイルアプリケーションでは必須です。

- データベースの変更をクエリ、ミューテーション、サブスクライブする機能
- オフラインでのデータ永続性と接続時の帯域幅の最適化
- アプリケーション内のデータの検索、フィルタリング、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) を介したターゲットを絞ったメッセージング
- 画像や動画などの豊富なコンテンツ
- 複数のデバイスと複数のユーザー間でのデータの同期
- データの表示と操作のためのきめ細かな認証コントロール

AWS でサーバーレスモバイルバックエンドを構築すると、スケーラビリティ、伸縮性、可用性を効率的かつコスト効率の高い方法で自動的に管理しながら、これらの機能を提供することができます。

特性:

- クライアントからアプリケーションデータの動作をコントロールし、API から必要なデータを明示的に選択する
- ビジネスロジックをモバイルアプリケーションからできるだけ疎結合化したい。
- 複数のプラットフォームにわたる開発を最適化するための API としてビジネス機能を提供することを考えている。
- マネージドサービスを活用して、高いレベルのスケーラビリティと可用性を提供しながら、モバイルバックエンドインフラストラクチャを維持する手間を省くことを目指している。
- 実際のユーザー需要とアイドルリソースに対する支払いに基づいて、モバイルバックエンドのコストを最適化したい。

リファレンスアーキテクチャ

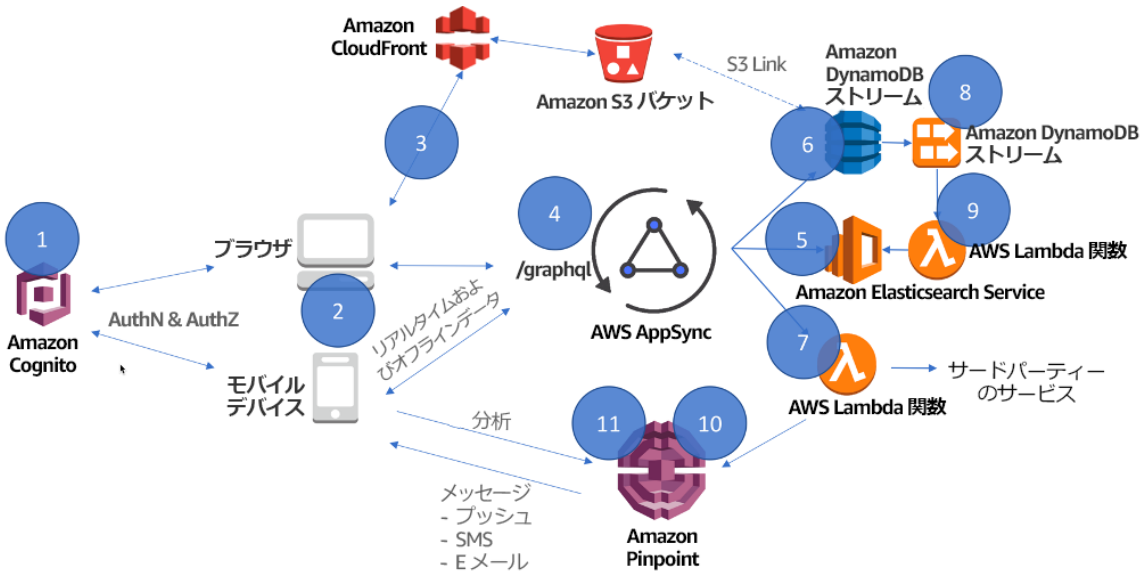

図 4: モバイルバックエンドのリファレンスアーキテクチャ

1. Amazon Cognito は、ユーザー管理に使用され、モバイルアプリケーションの ID プロバイダーとして使用されます。さらに、モバイルユーザーは Facebook、Twitter、Google+、Amazon などの既存のソーシャル ID を活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 のサービス API (Amazon S3 や Amazon Cognito など) に対して GraphQL 運用を実行することで、モバイルアプリケーションのバックエンドを操作します。
3. Amazon S3 では、プロファイルイメージなどの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が保存されます。そのコンテンツは CloudFront を介して安全に提供されます。
4. AWS AppSync は、モバイルユーザーに対する GraphQL HTTP リクエストとレスポンスをホストします。このシナリオでは、デバイスの接続時に AWS AppSync からのデータがリアルタイムであり、データもオフラインで利用できます。このシナリオのデータソースは、Amazon DynamoDB、Amazon Elasticsearch Service、または AWS Lambda 関数です
5. Amazon Elasticsearch Service は、モバイルアプリケーションと分析の主要な検索エンジンとして機能します。

6. DynamoDB は、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からの不要なデータを期限切れにするメカニズムなど、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。
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用の他の AWS のサービスの呼び出しを処理します。カスタムフローは、クライアントに対する GraphQL レスポンスの一部です。
8. DynamoDB ストリームは項目レベルの変更をキャプチャし、Lambda 関数で追加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 の間でストリーミングデータを管理するため、お客様はデータソースの論理的な GraphQL タイプと運用を組み合わせることができます。
10. Amazon Pinpoint は、アプリケーションインサイトのユーザーセッションやカスタムメトリクスなど、クライアントから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いて、すべてのユーザー/デバイス、またはターゲットを絞ったサブセットにメッセージを配信します。メッセージは、プッシュ通知、E メール、または SMS チャネルを使用してカスタマイズおよび送信できます。

設定に関する注意事項:

- さまざまなメモリとタイムアウト設定で Lambda 関数のパフォーマンステストを行い、ジョブに最適なリソースを使用していることを確認します。
- DynamoDB テーブルを作成するときのベストプラクティスに従い、AWS AppSync で GraphQL スキーマから自動的にプロビジョニングすることを検討してください。GraphQL スキーマでは、適切に分散されたハッシュキーを使用し、運用にインデックスを作成します。合理的な応答時間を確保するため、読み取り / 書き込みキャパシティーとテーブルのパーティション分割を計算してください。
- AWS AppSync のサーバー側データキャッシュ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。それ以降の API に対するクエリリクエストはすべてキャッシュから返されるため、TTL の有効期限が切れるまでデータソースに直接アクセスすることはできません。
- Amazon ES ドメインを管理するときは、ベストプラクティスに従います。さらに、Amazon ES では、ここにも適用されるシャーディングとアクセスパターンの設計に関する広範なガイドを提供しています。
- リゾルバーで設定された AWS AppSync のきめ細かなアクセスコントロールを使用して、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レベルにフィルタリングします。これは、AWS Identity and Access Management (IAM) または AWS AppSync による Amazon Cognito user pools 認証に適用できます。
- AWS Amplify と Amplify CLI を使用して、アプリケーションを複数の AWS のサービスで構成し、統合します。Amplify コンソールは、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ぼ必要とされない低レイテンシーの要件の場合、Amazon Cognito フェデレーティッド ID は、範囲を指定した認証情報を提供することで、モバイルアプリケーションが AWS のサービスと直接やり取りできます。たとえば、ユーザーのプロファイル写真をアップロードするとき、スコープ範囲を指定した Amazon S3 からメタデータファイルを取得するときなどです。

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と処理には、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、メトリクスの生成、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、IoT デバイスデータテレメトリおよび計測など、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と低レイテンシーが必要です。こういったアプリケーションでは、1 秒あたり数千のイベントを処理しなければならず、スパイクが多く発生します。

AWS Lambda と Amazon Kinesis を使用するならば、サーバーのプロビジョニングや管理を行うことなく、自動的にスケールできるサーバーレスストリームプロセスを構築できます。AWS Lambda で処理されたデータは DynamoDB に保存されるので、後から分析が可能です。

特性:

- ストリーミングデータを処理するためのインスタンスやサーバー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。
- Amazon Kinesis プロデューサーライブラリ (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。

リファレンスアーキテクチャ

ここでは、ソーシャルメディアデータを分析するためのリファレンスアーキテクチャである一般的なストリーム処理のシナリオを示します。

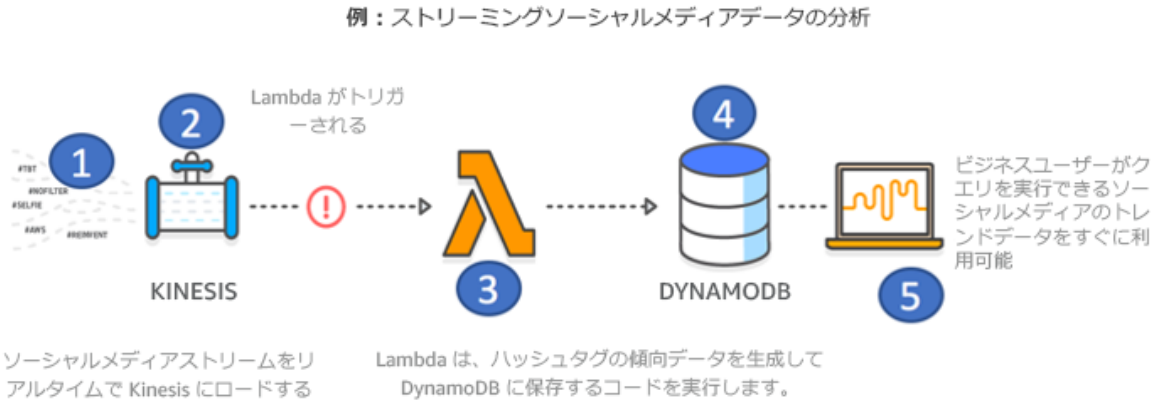

図 5: ストリーム処理のリファレンスアーキテクチャ

1. データプロデューサーは Amazon Kinesis プロデューサーライブラリ (KPL) を使用して、ソーシャルメディアストリーミングデータを Kinesis ストリームに送信します。Amazon Kinesis Agent と Kinesis API を活用する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って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、コンシューマーによって処理できます。コンシューマーは Lambda です
3. AWS Lambda は、取り込まれたデータの配列を 1 つのイベント/呼び出しとして受け取るストリームのコンシューマーとして機能します。さらなる処理は Lambda 関数によって実行されます。変換されたデータは、永続的ストレージ (この場合は DynamoDB) に保存されます。
4. Amazon DynamoDB は、AWS Lambda と統合できるトリガーを含む、高速で柔軟な NoSQL データベースサービスを提供し、他の場所でそのようなデータを利用できるようにします。
5. ビジネスユーザーは、DynamoDB のレポートインターフェイスを活用して、ソーシャルメディアのトレンドデータから洞察を収集する

設定に関する注意事項:

- Kinesis ストリームを再シャーディングするときのベストプラクティスに従って、より高い取り込みレートに対応してください。ストリーム処理の同時実行は、シャードの数と並列化係数によって決まります。したがって、スループット要件に応じて調整してください。
- 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ホワイトペーパーを検討してください。
- KPL を使用しない場合は、PutRecords などの非アトミック運用の部分的な失敗を考慮してください。これは、Kinesis API が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 レコードの両方を返すためです。
- 重複したレコードが発生 する可能性があり、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再試行とべき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場合は、Lambda よりも Kinesis Data Firehose を使用することを検討してください。
- 標準 SQL を使用してストリーミングデータのクエリを実行し、その結果のみを Amazon S3、Amazon Redshift、Amazon ES、または Kinesis にロードできる場合は、Lambda での Kinesis Data Analytics の使用を検討してください。
- AWS Lambda ストリームベースの呼び出しのベストプラクティスに従います。これは、バッチサイズ、シャードあたりの同時実行、ストリーム処理のモニタリングに対する影響を詳細にカバーするためです。
- Lambda の最大再試行回数、最大レコード期間、関数エラーに対するバッチの二分化、および失敗時の送信先エラーコントロールを使用して、より弾力性のあるストリーム処理アプリケーションを構築します。

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、信頼性の高いユーザーエクスペリエンスを確保するための要求度の高い要件があります。これまでは多くの場合、高可用性、グローバルな可用性、数千人または潜在的に数百万人のユーザーにスケールする能力を確保するには、予想される最大の要求度のウェブリクエストを処理するために、かなりの余分なキャパシティーを確保する必要がありました。そのため、多くの場合、サーバー群や追加のインフラストラクチャコンポーネントを管理する必要がありました。その結果、かなりの資本支出とキャパシティープロビジョニングのためのリードタイムが長くなりました。

AWS でサーバーレスコンピューティングを使用すると、サーバーの管理やキャパシティーのプロビジョニングの推測、アイドル状態のリソースに対する支払いといった手間のかかる作業を行うことなく、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。さらに、セキュリティ、信頼性、またはパフォーマンスを犠牲にする必要はありません。

特性:

- 高いレベルの弾力性と可用性を備えた、数分で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が欲しい。
- 適切な応答時間で一貫したユーザーエクスペリエンスを実現したい。
- 一般的なプラットフォームの管理に関連する手間のかかる作業をプラットフォームで制限するために、できる限りマネージド型サービスを活用したい。
- 実際のユーザー需要とアイドル状態のリソースに対する支払いに基づいて、コストを最適化したい。
- セットアップと運用が簡単で、後でも限定的な影響だけで拡張できるフレームワークを作りたい。

リファレンスアーキテクチャ

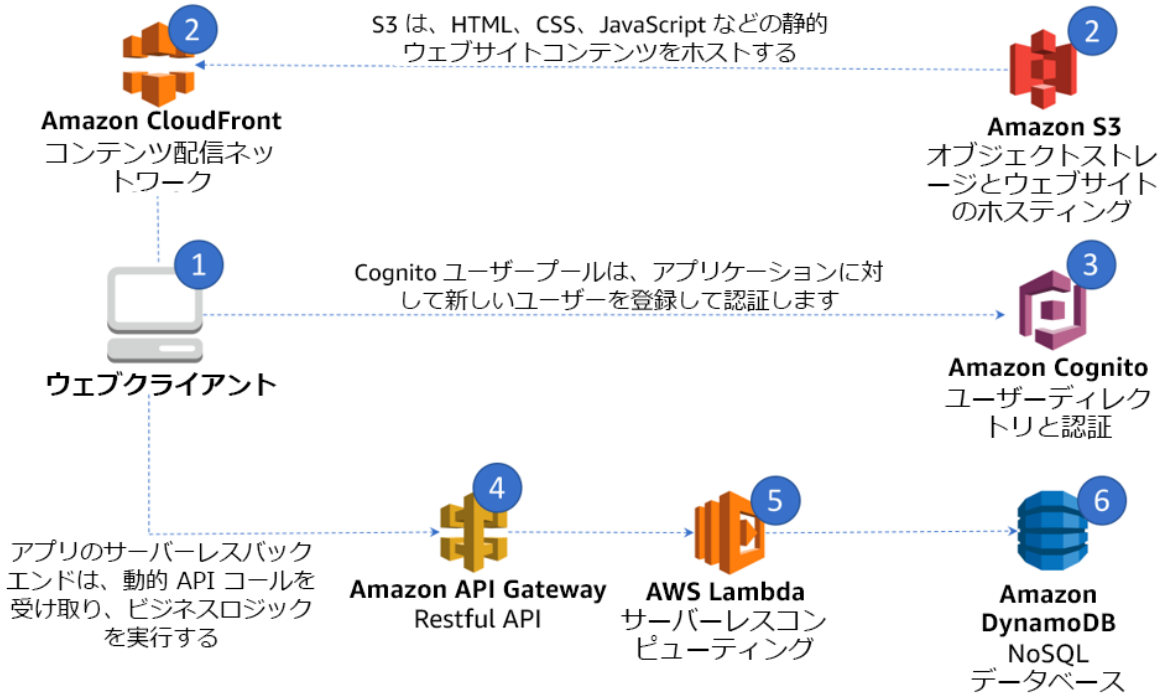

図 6: ウェブアプリケーションのリファレンスアーキテクチャ

1. このウェブアプリケーションのコンシューマー は、地理的に集中しているか、世界中に分散している可能性があります。Amazon CloudFront を活用すると、キャッシュ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のパフォーマンスエクスペリエンスが向上するだけでなく、バックエンドへの冗長呼び出しも制限されます。
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 を介して安全に提供されます。
3. Amazon Cognito ユーザープールは、ウェブアプリケーションのユーザー管理機能と ID プロバイダ機能を提供します。
4. 多くのシナリオでは、Amazon S3 からの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、動的コンテンツはアプリケーションで送受信される必要があります。例えば、ユーザーがフォーム経由でデータを送信すると、Amazon API Gateway はこれらの呼び出しを行い、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ための安全なエンドポイントとして機能します。
5. AWS Lambda 関数は、DynamoDB 上でウェブアプリケーションの作成、読み込み、更新、削除 (CRUD) 運用を提供します。
6. Amazon DynamoDB では、バックエンドの NoSQL データストアを提供して、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できます。

設定に関する注意事項:

- AWS にサーバーレスウェブアプリケーションの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います。詳細については、運用上の優秀性の柱をご覧ください。
- 単一ページのウェブアプリケーションの場合、AWS Amplify コンソールを使用して、アトミックデプロイ、キャッシュの有効期限、カスタムドメイン、ユーザーインターフェイス (UI) テストを管理します。
- 認証と認可に関する推奨事項については、セキュリティの柱を参照してください。
- ウェブアプリケーションの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 (p. 9) 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、API Gateway の使用量プランと Amazon Cognito user poolsを活用して、さまざまなユーザーがアクセスできる範囲を絞り込めま

す。たとえば、プレミアムユーザーは、API 呼び出し、追加の API へのアクセス、追加のストレージなどのスループットが高くなります。

- このシナリオで扱われていない検索機能をアプリケーションで使用する場合は、モバイルバックエンドシナリオを (p. 13) 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、各柱について説明し、サーバーレスアプリケーションのソリューションを設計する際に関連する定義、ベストプラクティス、質問、考慮事項、主要な AWS のサービスについて説明します。

簡潔にするために、サーバーレスワークロードに特有の Well-Architected フレームワークからのみ質問を選択しました。この文書に含まれていない質問は、アーキテクチャの設計時に考慮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を読むことをお勧めします。

トピック

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、サポートプロセスと手順を継続的に改善していくために、システムを実行してモニタリングする能力が含まれています。

定義

クラウド内での運用の卓越性について 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、ゲームの日に関する Well-Architected フレームワークの対象となるものに加えて、サーバーレスアプリケーション内で運用上の優秀性を促進するために調べる必要がある特定の領域があります。

ベストプラクティス

トピック

準備

このサブセクションに属するサーバーレスアプリケーションに固有の運用方法はありません。

運用

OPS 1: サーバーレスアプリケーションの状態をどのように理解しますか?

メトリクスとアラート

使用するすべての AWS のサービスの Amazon CloudWatch メトリクスとディメンションを理解しておくことが重要です。そうすれば、その行動を評価し、適切なカスタムメトリクスを追加するための計画を立てることができます。

Amazon CloudWatch は、自動化されたクロスサービスおよびサービスごとのダッシュボードを提供し、使用する AWS のサービスの主要メトリクスを理解するのに役立ちます。カスタムメトリクスの場合は、Amazon CloudWatch Embedded Metric Format を使用して、サーバーレスアプリケーションのパフォーマンスに影響を与えることなく、CloudWatch によって非同期的に処理されるメトリクスのバッチを記録します。

次のガイドラインは、ダッシュボードを作成する場合でも、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策定する場合でも使用できます。

- ビジネスメトリクス
  - ビジネス目標に対してアプリケーションのパフォーマンスを測定するビジネス KPI で、ビジネス全体に重大な影響を及ぼす場合、収益に関係するかどうかを知るために重要です。
  - 例: 発注された注文、デビット/クレジットカード運用、購入されたフライトなど。

- カスタマーエクスペリエンスメトリクス
  - カスタマーエクスペリエンスデータは、UI/UX の全体的な有効性だけでなく、アプリケーションの特定のセクションにおける変更や異常がカスタマーエクスペリエンスに影響を与えているかどうかも示します。多くの場合、これらはパーセンタイルで測定され、時間の経過に伴う影響や顧客ベース全体にどのように広がっているかを理解しようとするときに異常値を防ぐことができます。
  - 例: 知覚されるレイテンシー、カートへの商品の追加やチェックアウトにかかる時間、ページの読み込み時間など。

- システムメトリクス
  - ベンダーとアプリケーションのメトリクスは、前のセクションの根本原因を支えるために重要です。また、システムが正常であるか、リスクがあるか、すでに顧客であるかもわかります。
  - 例: HTTP エラー/成功、メモリ使用率、関数の継続時間/エラー/スロットリング、キュー長、ストリームレコードの長さ、統合レイテンシーなどの割合。

- 運用メトリクス
  - 運用メトリクスは、特定のシステムの持続可能性とメンテナンスを理解するためにも同様に重要であり、時間の経過とともに安定性がどの程度向上または低下したかを特定するために不可欠です。

- 例: チケット数 (成功と失敗の解決など)、オンコールのユーザーがページ分割された回数、可用性、CI/CD パイプラインの統計 (成功/失敗したデプロイ、フィードバック時間、サイクル、リードタイムなど)

CloudWatch アラームは、個々のレベルと集計レベルの両方で設定する必要があります。個々のレベルの例として、アプリケーションの異なる部分には異なるプロファイルがあるため、API を介して呼び出されたときに、Lambda からの Duration メトリクスまたは API Gateway からの IntegrationLatency でアラームが発生します。この場合、関数を通常よりかなり長く実行させる不適切なデプロイをすばやく特定できます。

集約レベルの例にはアラームが含まれますが、次のメトリクスに限定されません。

- AWS Lambda: Duration、Errors、Throttling、ConcurrentExecutions です。ストリームベースの呼び出しの場合、IteratorAge でアラートを出します。非同期呼び出しの場合、DeadLetterErrors でアラートを出します。
- Amazon API Gateway: IntegrationLatency、Latency、5XXError
- Application Load Balancer: HTTPCode_ELB_5XX_Count、RejectedConnectionCount、HTTPCode_Target_5XX_Count、UnHealthyHostCount、L
- AWS AppSync: 5XX とレイテンシー
- Amazon SQS: ApproximateAgeOfOldestMessage
- Amazon Kinesis Data Streams: ReadProvisionedThroughputExceeded、WriteProvisionedThroughputExceeded、GetRecords.IteratorAgeMilliseconds、(Kinesis プロデューサーライブラリを使用している場合)、および GetRecords.Success
- Amazon SNS: NumberOfNotificationsFailed、NumberOfNotificationsFilteredOut-InvalidAttributes
- Amazon SES: 拒否、バウンス、苦情、レンダリングの失敗
- AWS Step Functions: ExecutionThrottled、ExecutionsFailed、ExecutionsTimedOut
- Amazon EventBridge: FailedInvocations、ThrottledRules
- Amazon S3: 5xxErrors、TotalRequestLatency
- Amazon DynamoDB: ReadThrottleEvents、WriteThrottleEvents、SystemErrors、ThrottledRequests、UserErrors

一元化された構造化されたログ記録

アプリケーションのログ記録を標準化して、トランザクション、相関識別子、コンポーネント間のリクエスト識別子、ビジネス上の成果に関する運用情報を出力させます。この情報を使用して、ワークロードの状態に関する任意の質問に回答します。

JSON を出力として使用する構造化ログの例を次に示します。

```
{ "timestamp":"2019-11-26 18:17:33,774", "level":"INFO",
 "location":"cancel.cancel_booking:45", "service":"booking", "lambda_function_name":"test",
 "lambda_function_memory_size":"128", "lambda_function_arn":"arn:aws:lambda:eu-west-1:
 12345678910:function:test", "lambda_request_id":"52fdfc07-2182-154f-163f-5f0f9a621d72",
 "cold_start": "true", "message":{ "operation":"update_item",
 "details":":{ "Attributes":{ "status":"CANCELLED" }, "ResponseMetadata":
{ "RequestId":"G7S3SCFDEMEINPG6AOC6CL5IDNVV4KQNSO5AEMVJF66Q9ASUAAJG", "HTTPStatusCode":200,
 "HTTPHeaders":{ "server":"Server", "date":"Thu, 26 Nov 2019 18:17:33 GMT", "content-
type":"application/x-amz-json-1.0", "content-length":"43", "connection":"keep-alive",
 "x-amzn-requestid":"G7S3SCFDEMEINPG6AOC6CL5IDNVV4KQNSO5AEMVJF66Q9ASUAAJG", "x-amz-
crc32":"1848747586" }, "RetryAttempts":0 } } } }
```

集中ロギングは、サーバーレスアプリケーションログの検索と分析に役立ちます。構造化ロギングを使用すると、アプリケーションの状態に関する任意の質問に答えるクエリを簡単に取得できます。システムが

大きくなり、より多くのログが取り込まれる場合は、適切なログレベルとサンプリングメカニズムを使用して、DEBUG モードでログの一部を記録することを検討してください。

分散トレース

非サーバーレスアプリケーションと同様に、分散システムではより大きな規模で異常が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、分散トレースを行うことが基本です。

サーバーレスアプリケーションに変更を加えるには、従来のワークロードで使用されているデプロイ、変更、リリース管理の原則の多くが必要です。ただし、これらの原則を達成するために既存のツールを使用する方法には微妙な変更があります。

AWS X-Ray を使用したアクティブトレースを有効にして、分散トレース機能を提供するとともに、より高速なトラブルシューティングのためのビジュアルサービスマップを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特定し、レイテンシーの分散などの異常をすばやく把握するのに役立ちます。

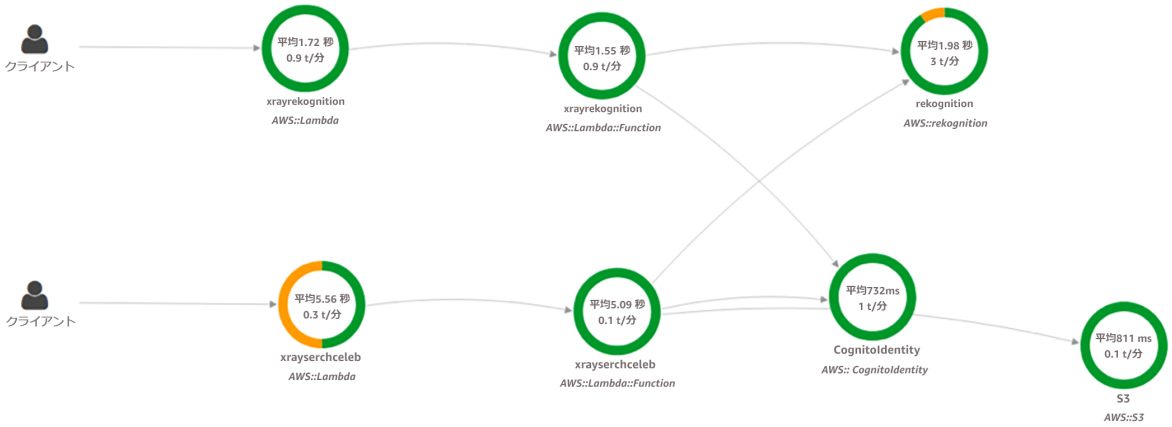

図 7: 2 つのサービスを視覚化する AWS X-Ray サービスマップ

サービスマップは、注意と弾力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、および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。デフォルトのタイムアウトと再試行設定に依存するべきではありません。代わりに、ソケットの読み取り / 書き込みタイムアウトが発生した場合に失敗するように調整します。これは、特定のクライアントでデフォルトが分ではなく秒になる場合があるためです。

X-Ray には、アプリケーション内の異常の特定効率を向上させることができる 2 つの強力な機能 (注釈とサブセグメント) も用意されています。

サブセグメントは、アプリケーションロジックがどのように構築され、どのような外部依存関係と通信する必要があるかを理解するのに便利です。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、またはブール値を持つキーと値のペアです。

これらを組み合わせると、データベースのクエリにかかる時間や、多くの人員がいる写真の処理にかかる時間など、特定の運用やビジネストランザクションのパフォーマンス統計をすばやく特定できます。

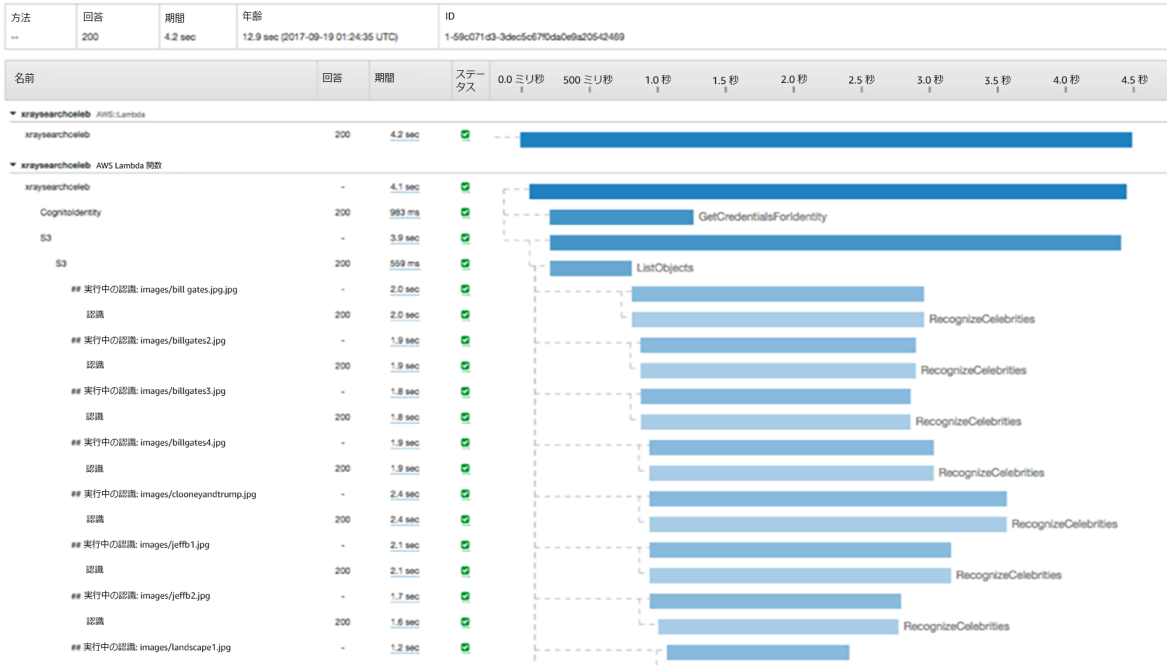

図 8: ## で始まるサブセグメントを持つ AWS X-Ray トレース

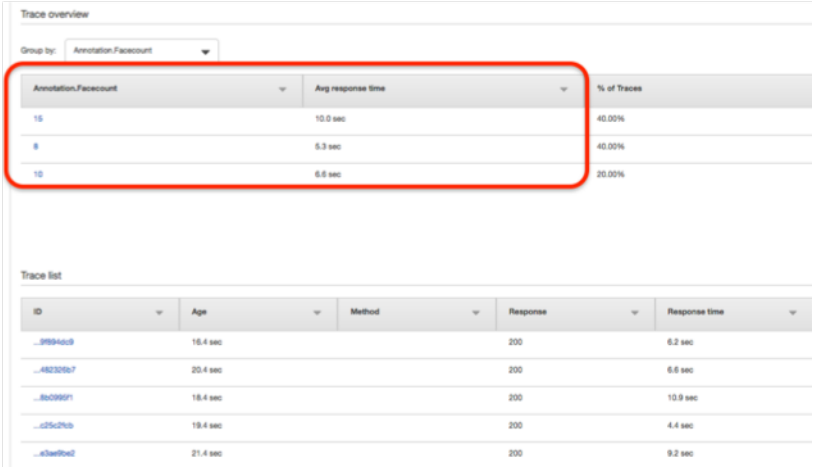

図 9: カスタム注釈でグループ化された AWS X-Ray トレース

OPS 2: アプリケーションライフサイクル管理にはどのようにアプローチしますか？

プロトタイプ

コードとしてのインフラストラクチャを使用して、プロトタイプを作成する新しい機能の一時的な環境を作成し、完了時にそれらを解体します。チームのサイズや組織内のオートメーションレベルに応じて、チームごと、またはデベロッパーごとに、専用アカウントを使用できます。

一時的な環境では、マネージドサービスを使用する際の忠実度が高くなり、コントロールレベルが向上し、ワークロードが意図したとおりに統合および動作することの信頼性が得られます。

設定管理では、ログレベルやデータベース接続文字列など、頻繁ではない変更に環境変数を使用します。機能の切り替えなどの動的設定には AWS System Manager パラメータストアを使用し、AWS Secrets Manager を使用して機密データを保存します。

テスト

テストは通常、単体テスト、統合テスト、受諾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、さまざま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、サーバーレス以外のアプリケーションと変わらないため、変更を加えることなくローカルで実行できます。

統合テストでは、コントロールできないサービスを模擬しないでください。サービスが変更され、予期しない結果が生じる可能性があるためです。これらのテストは、実稼働環境でリクエストを処理するときにサーバーレスアプリケーションが使用するのと同じ環境を提供できるため、実際のサービスを使用する場合により適切に実行されます。

承諾テストまたはエンドツーエンドテストは、変更を加えることなく実行する必要があります。主な目的は、利用可能な外部インターフェイスを通じて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す。したがって、ここで注意すべき固有の推奨事項はありません。

一般的に、AWS Marketplace で使用できる Lambda およびサードパーティー製のツールは、パフォーマンステストの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中に注意すべき考慮事項をいくつか示します。

- 呼び出された最大メモリ使用量や初期化期間などのメトリクスは、CloudWatch Logs で使用できます。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。
- Lambda 関数が Amazon Virtual Private Cloud (VPC) 内で実行されている場合は、サブネット内で利用可能な IP アドレス空間に注意してください。
- モジュール化されたコードをハンドラの外部で個別の関数として作成すると、より多くのユニットテスト可能な関数が可能になります。
- Lambda 関数の静的コンストラクタ/初期化コード (ハンドラ外のグローバルスコープ) で参照される外部化された接続コード (リレーショナルデータベースへの接続プールなど) を確立すると、Lambda 実行環境が再利用された場合に外部接続のしきい値に達しないことが保証されます。
- パフォーマンステストがアカウントの現在の制限を超えない限り、DynamoDB オンデマンドテーブルを使用してください。
- パフォーマンステストでサーバーレスアプリケーション内で使用できるその他のサービスの制限を考慮してください。

デプロイ

コードとしてのインフラストラクチャとバージョン管理を使用し、変更とリリースの追跡を有効にします。開発ステージと本稼働ステージを別々の環境に分離します。これにより、手動プロセスに起因するエラーが軽減され、コントロールレベルが向上し、ワークロードが意図したとおりの動作を行う確実性を増すことができます。

サーバーレスフレームワークを使用して、AWS SAM や Serverless Framework などのサーバーレスアプリケーションのモデル化、プロトタイプ、構築、パッケージ化、デプロイを行います。コードとしてのインフラストラクチャとフレームワークを使用すると、サーバーレスアプリケーションとその依存関係をパラメータ化して、隔離されたステージと AWS アカウント間のデプロイを容易にすることができます。

たとえば、CI/CD パイプラインのベータステージでは、ベータ AWS アカウントで次のリソースを作成できます。また、異なるアカウント (Gamma、Dev、Prod) に必要になる各ステージ OrderAPIBeta、OrderServiceBeta、OrderStateMachineBeta、OrderBucketBeta、OrderTableBeta にも同様です。

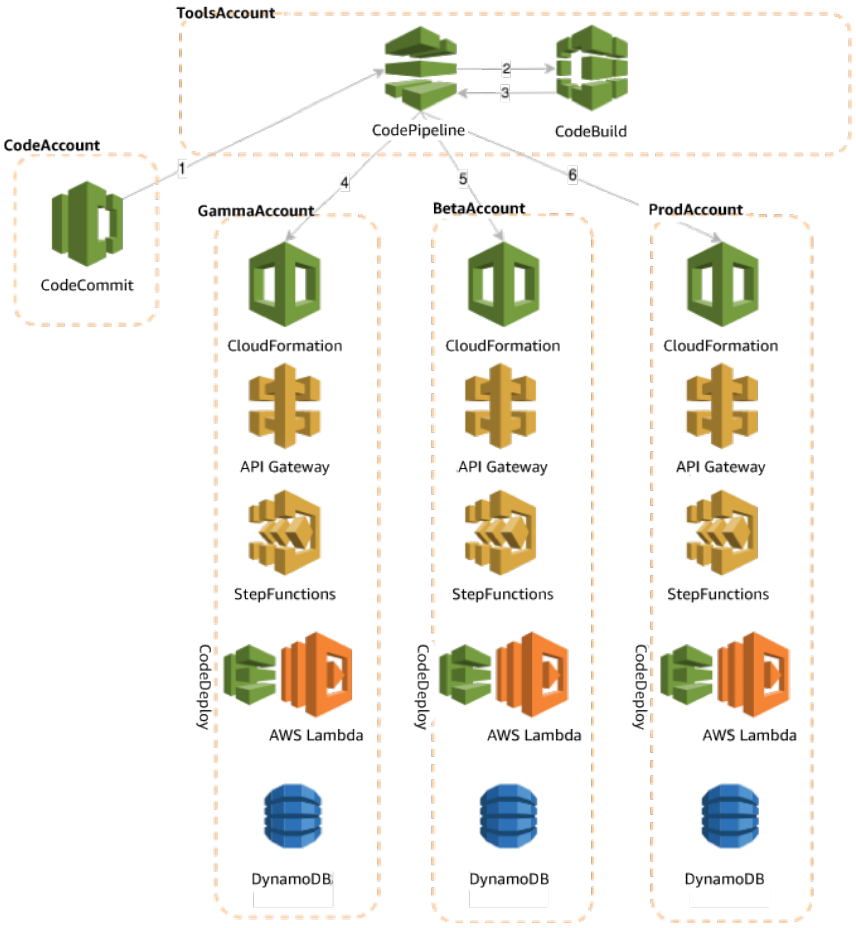

図 10: 複数のアカウントの CI/CD パイプライン

本稼働環境にデプロイする場合は、一度のシステムよりも安全なデプロイを優先します。これは、Canary または線形デプロイでは、新しい変更が徐々にエンドユーザーに移行するためです。CodeDeploy フック (BeforeAllowTraffic、AfterAllowTraffic) とアラームを使用して、デプロイの検証、ロールバック、アプリケーションに必要なカスタマイズをより詳細にコントロールできます。

また、合成トラフィック、カスタムメトリクス、アラートの使用をロールアウトデプロイの一部として組み合わせることもできます。これにより、カスタマーエクスペリエンスに影響を与えている可能性のある新しい変更によるエラーを積極的に検出できます。

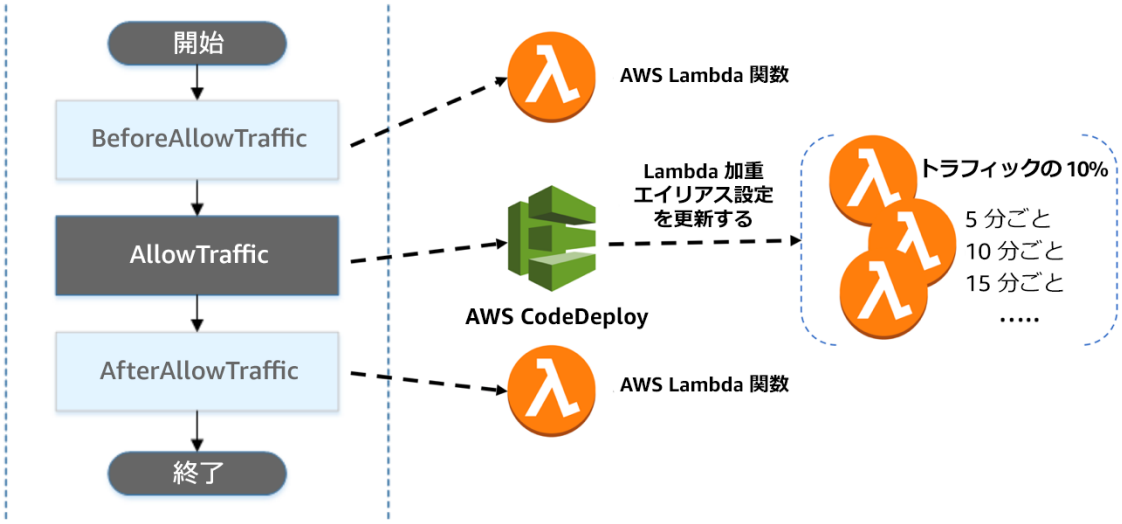

図 11: AWS CodeDeploy Lambda のデプロイとフック

進化

このサブセクションに属するサーバーレスアプリケーションに固有の運用方法はありません。

主要な AWS のサービス

優れた運用を実現する主要な AWS のサービスには、AWS Systems Manager Parameter Store、AWS SAM、CloudWatch、AWS CodePipeline、AWS X-Ray、Lambda、API Gateway があります。

リソース

運用上の優秀性に関する当社のベストプラクティスの詳細については、以下のリソースを参照してください。

ドキュメントとブログ

- API Gateway のステージ変数
- Lambda環境変数
- AWS SAM CLI
- X-Ray レイテンシーディストリビューション
- 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
- System Manager (SSM) パラメータストア
- サーバーレスアプリケーションの継続的デプロイに関するブログ記事
- SamFarm: CI/CD の例
- CI/CD を使用したサーバーレスアプリケーションの例
- アラートとダッシュボードを自動化するサーバーレスアプリケーションの例
- Python 用 CloudWatch 埋め込みメトリクス形式ライブラリ
- Node.js 用 CloudWatch 埋め込みメトリクス形式ライブラリ
- トレース、構造化ログ記録、カスタムメトリクスを実装するサンプルライブラリ

- 全般的な AWS の制限
- Stackery: マルチアカウントのベストプラクティス

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

サードパーティー製ツール

- サードパーティー製フレームワーク/ツールを含むサーバーレス開発者用ツール
- Stelligent: 運用メトリクスの CodePipeline ダッシュボード

セキュリティの柱

セキュリティの柱とは、リスクの評価とその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、資産を保護する能力のことを指します。

定義

クラウド内でのセキュリティには、5 つのベストプラクティス領域があります。

- アイデンティティとアクセスの管理
- 発見的統制
- インフラストラクチャの保護
- データ保護
- インシデントへの対応

サーバーレスの場合、オペレーティングシステムのパッチ適用、バイナリの更新などのインフラストラクチャ管理タスクが不要になるため、今日の最大のセキュリティ上の懸念事項に対処することができます。サーバーレスではないアーキテクチャに比べて攻撃領域は縮小されますが、Open Web Application Security Project (OWASP) と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。

このセクションでは、攻撃者が誤って設定されたアクセス許可にアクセスしようとしたり、悪用したりする可能性のある特定の手法に対処するために役立つ質問を記載しました。このセクションで説明する方法は、クラウドプラットフォーム全体のセキュリティに強く影響するため、慎重に検証し、頻繁に確認する必要があります。

インシデント対応カテゴリについては、AWS Well-Architected フレームワークの手法が利用可能です。そのため、この文書では省略します。

ベストプラクティス

トピック

アイデンティティとアクセスの管理

SEC 1: サーバーレス API へのアクセスをどのようにコントロールしますか？

API は、実行できる運用と取得できる貴重なデータのために、攻撃者の標的となることがよくあります。これらの攻撃を防御するためのさまざまなセキュリティ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、API Gateway 内で API 呼び出しを承認するための 4 つのメカニズムがあります。

- AWS_IAM 認証
- Amazon Cognito user pools
- API Gateway Lambda オーソライザー
- リソースポリシー

主に、これらのメカニズムが実装されているかどうか、および実装方法を理解する必要があります。現在 AWS 環境内にいるコンシューマーや、環境にアクセスするために AWS Identity and Access Management (IAM) の一時的な認証情報を取得する手段を持っているコンシューマーに対しては、AWS_IAM 認証でそれぞれの IAM ロールに API を安全に呼び出すための最小権限のアクセス権限を追加します。

次の図は、このコンテキストでの AWS_IAM 認証の使用を示しています。

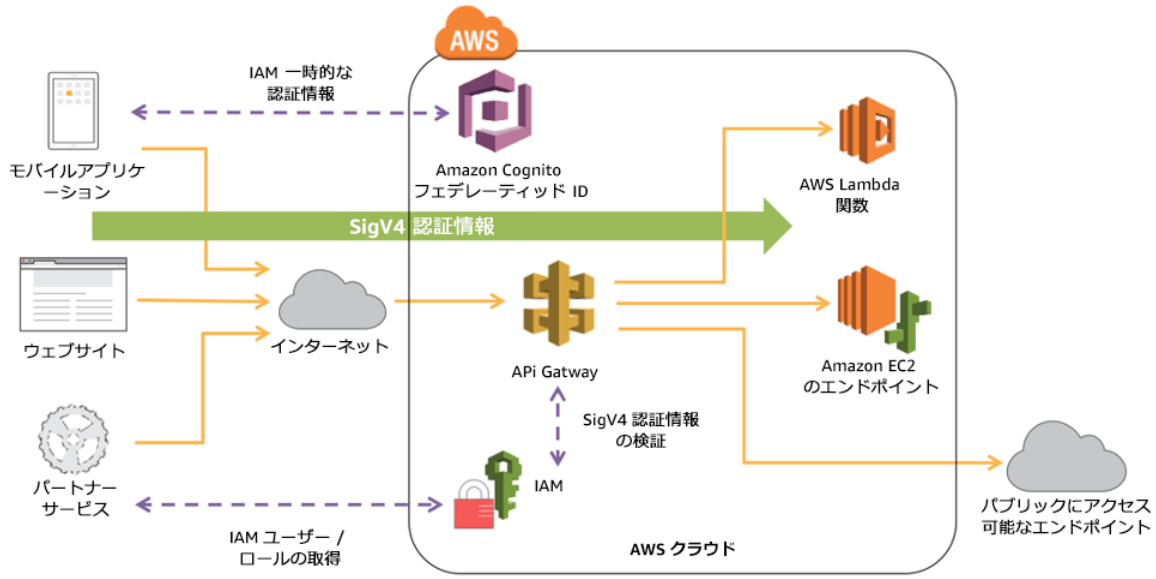

図 10: AWS_IAM 認証

既存の ID プロバイダー (IdP) がある場合は、API Gateway Lambda オーソライザーを使用して Lambda 関数を呼び出し、IdP に対して特定のユーザーを認証/検証できます。ID メタデータに基づくカスタム検証ロジックに Lambda オーソライザーを使用できます。

Lambda オーソライザーは、ベアラートークンまたはリクエストコンテキスト値から派生した追加情報をバックエンドサービスに送信できます。たとえば、オーソライザーは、ユーザー ID、ユーザー名、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、バックエンドは認証トークンをユーザー中心のデータにマッピングする必要がなく、そのような情報の公開を認証関数のみに制限できます。

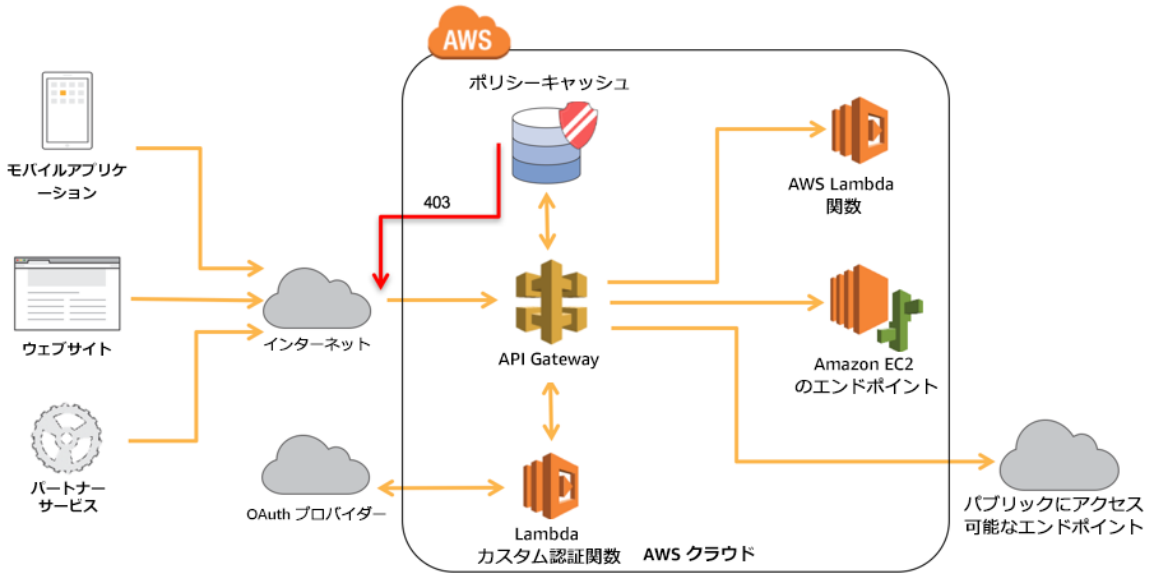

図 12: API Gateway Lambda オーソライザー

IdP がない場合は、Amazon Cognito user poolsを活用して、組み込みのユーザー管理を提供するか、Facebook、Twitter、Google+、Amazon などの外部 ID プロバイダーと統合できます。

これは、モバイルバックエンドのシナリオでよく見られます。ユーザーは、ソーシャルメディアプラットフォームの既存のアカウントを使用して認証しながら、E メールアドレス/ユーザー名で登録/サインインできます。このアプローチでは、OAuth Scopes による詳細な認証も提供されます。

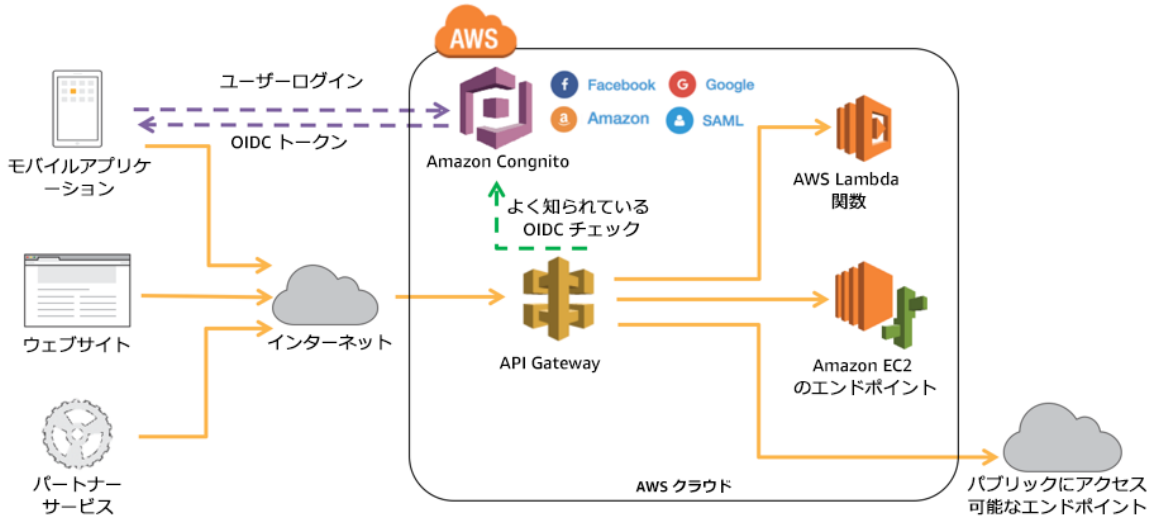

図 13: Amazon Cognito user pools

API API Gateway キーはセキュリティメカニズムではなく、パブリック API でない限り、認証に使用しないでください。主に API 全体のコンシューマーの使用状況を追跡するために使用し、このセクションで前述したオーソライザーに加えて使用できます。

Lambda オーソライザーを使用する場合は、クエリ文字列パラメータやヘッダーを介して認証情報や機密データを渡さないよう強くお勧めします。そうしないと、システムを開いて悪用される可能性があります。

Amazon API Gateway リソースポリシーは、指定された AWS プリンシパルが API を呼び出すことができるかどうかをコントロールするために、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り、API 呼び出しを制限できます。

- 指定された AWS アカウントのユーザー、または任意の AWS IAM Identity
- 指定された送信元 IP アドレス範囲または CIDR ブロック
- 指定された Virtual Private Cloud (VPC) または VPC エンドポイント (任意のアカウント)

リソースポリシーを使用すると、特定の IP 範囲を持つ既知のクライアントまたは別の AWS アカウントからのリクエストのみを許可するなど、一般的なシナリオを制限できます。プライベート IP アドレスからのリクエストを制限したい場合は、代わりに API Gateway プライベートエンドポイントを使用することをお勧めします。

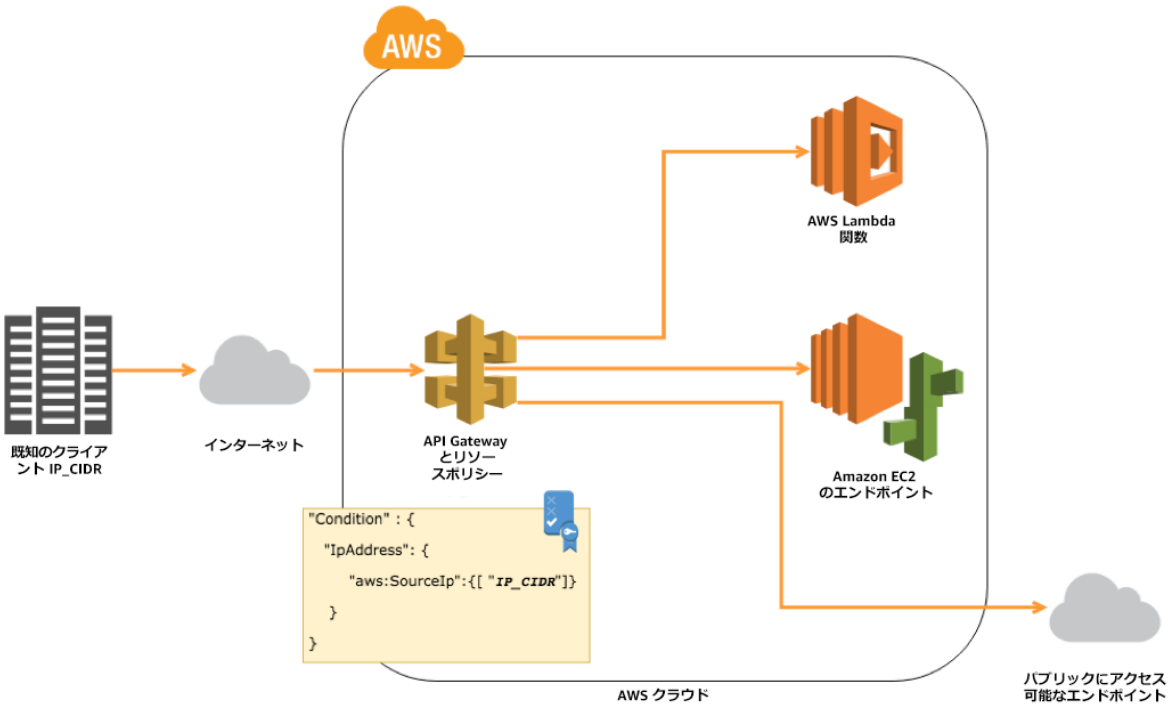

図 14: IP CIDR に基づく Amazon API Gateway リソースポリシー

プライベートエンドポイントを使用すると、API Gateway は VPC 内のサービスやリソース、または Direct Connect 経由でお客様のデータセンターに接続されるサービスやリソースへのアクセスを制限します。

プライベートエンドポイントとリソースポリシーの両方を組み合わせることで、API は特定のプライベート IP 範囲内の特定のリソース呼び出しに制限できます。この組み合わせは、主に、同じアカウントまたは別のアカウントにある可能性がある内部マイクロサービスで使用されます。

大規模なデプロイや複数の AWS アカウントの場合、組織は API Gateway でクロスアカウントの Lambda オーソライザーを活用して、メンテナンスを減らし、セキュリティプラクティスを一元化できます。たとえば、API Gateway には、別のアカウントで Amazon Cognito user pools ユーザープールを使用できる機能があります。Lambda オーソライザーは、別のアカウントで作成および管理し、API Gateway によって管理される複数の API にわたって再利用することもできます。どちらのシナリオも、API 全体で認証プラクティスを標準化する必要がある複数のマイクロサービスを使用したデプロイに共通しています。

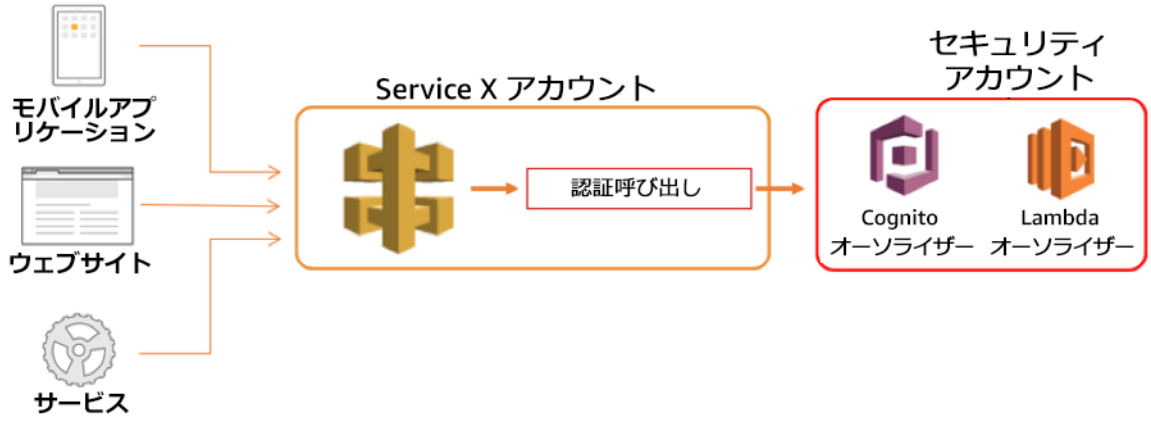

図 15: API Gateway のクロスアカウントオーソライザー

SEC 2: サーバーレスアプリケーションのセキュリティ境界をどのように管理していますか？

Lambda 関数では、最小権限のアクセスに従い、特定の運用の実行に必要なアクセスのみを許可することをお勧めします。必要以上のアクセス許可を持つロールをアタッチすると、システムが悪用される可能性があります。

セキュリティコンテキストでは、範囲を限定したアクティビティを実行する小さな機能があると、Well-Architected のサーバーレスアプリケーションにつながります。IAM ロールについては、複数の Lambda 関数内で IAM ロールを共有すると、最小権限のアクセスに違反する可能性があります。

発見的統制

ログ管理は、セキュリティやフォレンジックから規制要件や法的要件まで十分に対応できる、Well-Architected 設計に重要な部分です。

アプリケーションの依存関係の脆弱性を追跡することも同様に重要です。攻撃者は、使用するプログラミング言語に関係なく、依存関係にある既知の脆弱性を悪用できるためです。

アプリケーション依存関係の脆弱性スキャンには、CI/CD パイプライン内に統合できる OWASP 依存関係チェックなど、商用およびオープンソースのソリューションがいくつかあります。バージョン管理ソフトウェアリポジトリの一部として、AWS SDK を含むすべての依存関係を含めることが重要です。

インフラストラクチャの保護

サーバーレスアプリケーションが Virtual Private Cloud (VPC) にデプロイされた他のコンポーネントやオンプレミスにあるアプリケーションとやり取りする必要があるシナリオでは、ネットワークの境界を考慮することが重要です。

Lambda 関数は、VPC 内のリソースにアクセスす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ように、すべてのレイヤーでトラフィックをコントロールします。コンプライアンス上の理由でアウトバウンドトラフィックのフィルタリングが必要なワークロードの場合でも、プロキシは非サーバーレスアーキテクチャに適用されるのと同じ方法で使用できます。

ネットワーキングの境界をアプリケーションコードレベルでのみ適用し、アクセスできるリソースに関する指示を与えることは、懸念が分離されているため、推奨されません。

サービス間の通信では、静的キーよりも AWS IAM を使用した一時的な認証情報などの動的認証を優先します。API Gateway と AWS AppSync はどちらも IAM 認証をサポートしており、AWS のサービスとの通信を保護するのが理想的です。

データ保護

サーバーレスアプリケーションの設計によっては、ログに機密データが含まれている可能性があるため、API Gateway アクセスログを有効にして、必要なもののみを選択することを検討してください。このため、サーバーレスアプリケーションを通過する機密データは、暗号化することをお勧めします。

API Gateway と AWS AppSync では、すべての通信、クライアント、統合で TLS を採用しています。HTTP ペイロードは転送中に暗号化されますが、URL の一部であるリクエストパスとクエリ文字列は暗号化されない場合があります。したがって、機密データが標準出力に送信されると、CloudWatch Logs を介して誤って公開される可能性があります。

さらに、不正な形式または傍受された入力を攻撃ベクトルとして使用して、システムへのアクセスを取得したり、誤動作を引き起こす可能性があります。機密データは、「AWS Well-Architected フレームワーク」で説明されているように、可能なすべてのレイヤーで常に保護する必要があります。このホワイトペーパーの推奨事項は、ここでも適用されます。

API Gateway に関しては、機密データは HTTP リクエストの一部として暗号化する前にクライアント側で暗号化するか、HTTP POST リクエストの一部としてペイロードとして送信する必要があります。これには、特定のリクエストを行う前に機密データが含まれている可能性があるヘッダーの暗号化も含まれます。

Lambda 関数または API Gateway を設定できる統合については、処理やデータ操作の前に機密データを暗号化する必要があります。これにより、永続的ストレージにデータが公開された場合や、CloudWatch Logs によってストリーミングおよび保持される標準出力によってデータが漏洩するのを防ぐことができます。

のドキュメントで前に説明したシナリオでは、Lambda 関数は暗号化されたデータを DynamoDB、Amazon ES、または Amazon S3 に保持し、保管時の暗号化も保持します。HTTP リクエストパス/クエリ文字列の一部として、または Lambda 関数の標準出力として、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強くお勧めします。

機密データが暗号化されていない場合の API Gateway での口グ記録を有効にすることもお勧めしません。次のサブセクション「発見的統制 (p. 32)」で説明したように、このような場合は API Gateway の口グ記録を有効にする前に、コンプライアンスチームに相談してください。

SEC 3: ワークロードにどのようにアプリケーションセキュリティを実装しますか?

AWS Well-Architected フレームワークで説明されているように、AWS セキュリティ速報および業界の脅威インテリジェンスによって作成されたセキュリティ意識ドキュメントを確認します。アプリケーションのセキュリティに関する OWASP ガイドラインは引き続き適用されます。

インバウンドイベントを検証してサニタイズし、サーバーレス以外のアプリケーションの場合と同様にセキュリティコードレビューを実行します。API Gateway では、基本リクエスト検証を最初のステップとして設定し、リクエストが設定された JSON スキーマリクエストモデルと、URI、クエリ文字列、またはヘッダーの必須パラメータに準拠していることを確認します。アプリケーション固有の深層検証は、個別の Lambda 関数、ライブラリ、フレームワーク、サービスとして実装する必要があります。

データベースのパスワードや API キーなどのシークレットを、ローテーション、セキュアで監査されたアクセスを許可する Secrets Manager に保存します。Secrets Manager では、監査を含むシークレットに対してきめ細かなポリシーを許可します。

主要な AWS のサービス

セキュリティのための主要な AWS のサービスは、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

当社のセキュリティのベストプラクティスの詳細については、以下のリソースを参照してください。

ドキュメントとブログ

- Amazon S3 での Lambda 関数の IAM ロールの例
- API Gateway リクエストの検証
- API Gateway Lambda オーソライザー
- Amazon Cognito フェデレーティッド ID、Amazon Cognito user pools、Amazon API Gateway による API アクセスの保護
- AWS Lambda の VPC アクセスの設定
- 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
- Lambda での AWS Secrets Manager の使用
- AWS Secrets Manager を使用したシークレットの監査
- OWASP 入力検証チートシート
- AWS サーバーレスセキュリティワークショップ

ホワイトペーパー

- OWASP Secure Coding Best Practices
- AWS セキュリティのベストプラクティス

パートナーのソリューション

- 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
- Protego サーバーレスセキュリティ
- Snyk - 商用脆弱性 DB と依存関係のチェック
- Lambda と API Gateway で Hashicorp Vault を使用する

サードパーティー製ツール

- OWASP 脆弱性の依存関係のチェック

信頼性の柱

信頼性の柱には、インフラストラクチャまたは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などによる障害の軽減などのシステムの能力が含まれます。

定義

クラウド内で信頼性のあるベストプラクティスには 3 つの領域があります：

- 基盤
- 変更管理
- 障害管理

信頼性を達成するには、システムは、需要や要件の変化を処理するためのメカニズムを備えた、十分に計画された基盤とモニタリングが必要です。また、サービス妨害攻撃を防御するメカニズムも必要になります。障害を検出し、理想的には、自動的に修復できるシステムを設計することが必要です。

ベストプラクティス

トピック

基盤

REL 1: インバウンドリクエスト率はどのように規制されていますか?

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 コンシューマーは別々のチームに所属する場合も、組織外にいる場合さえもあります。これにより、不明なアクセスパターンや、コンシューマーの認証情報が侵害されるリスクが原因で脆弱性が発生します。リクエストの数が処理ロジック / バックエンドで処理できる数を超えると、サービス API に影響する可能性があります。

さらに、データベース行の更新や、API の一部として S3 バケットに追加された新しいオブジェクトなど、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全体での追加実行をトリガーします。

サービス契約によって確立されたアクセスパターンを適用するには、API レベルでスロットリングを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、リソースレベルかグローバルレベルかにかかわらず、コンシューマーがサービスを使用する方法を確立するうえで不可欠です。

API 内で適切な HTTP ステータスコード (スロットリング用の 429 など) を返すと、コンシューマーはバックオフを実装し、それに応じて再試行することで、スロットルされたアクセスを計画するのに役立ちます。

より詳細なスロットリングと計測の使用については、グローバルスロットリングに加えて使用量プランを使用してコンシューマーに API キーを発行することで、API Gateway は予期しない動作でクォータとアクセスパターンを強制できます。また、API キーにより、個々のコンシューマーが不審なリクエストを行う場合に、管理者がアクセスを切断するプロセスが簡素化されます。

API キーをキャプチャする一般的な方法は、デベロッパーポータルを使用することです。これにより、サービスプロバイダーとして、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、ビジネス領域/目的をキャプチャし、DynamoDB などの耐久性のあるデータストアにこのデータを保存できます。これにより、コンシューマーの追加の検証が提供され、ID を使用したログ記録の追跡可能性が提供されるため、重大な変更のアップグレード/問題についてコンシューマーに連絡できます。

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証するセキュリティメカニズムではないため、API Gateway 内で使用可能ないずれかの認証オプションでのみ使用してください。

Lambda ほど高速にスケールできない可能性があるため、サービスの障害から特定のワークロードを保護するために、同時実行のコントロールが必要な場合があります。同時実行のコントロールを使用すると、個々の Lambda 関数レベルで設定される特定の Lambda 関数の同時呼び出し数の割り当てを制御できます。

個々の関数の同時実行セットを超える Lambda 呼び出しは、AWS Lambda サービスによって調整され、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 エラーを返します。非同期呼び出しはキューに入れられ、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まで再試行されます。

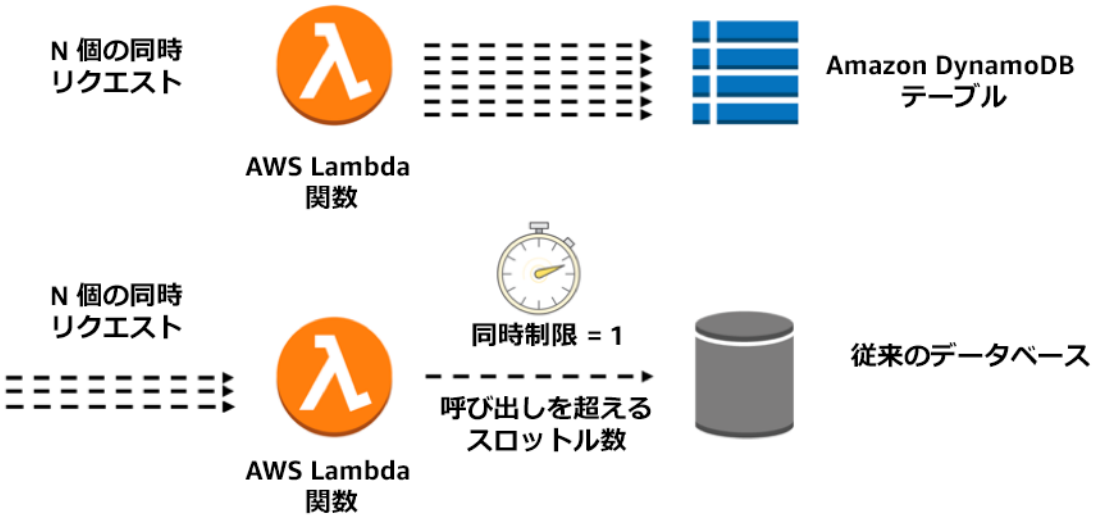

図 16: AWS Lambda の同時実行コントロール

同時実行の制御は、次のシナリオで特に便利です。

- 拡張制限がある可能性がある機密性の高いバックエンドまたは統合システム
- リレーショナルデータベースなどのデータベース接続プールの制限。同時制限が課せられる可能性があります。
- クリティカルパスサービス: 同じアカウントの制限に対する、承認などの高優先度の Lambda 関数、低優先度の関数 (バックオフィスなど)
- 異常が発生したときに Lambda 関数 (同時実行数 =0) を無効にする機能。
- 分散サービス妨害 (DDoS) 攻撃から保護するために、必要な実行の同時実行数を制限する

Lambda 関数の同時実行コントロールは、同時実行セットを超えてスケールする能力を制限し、アカウントの予約された同時実行プールから引き出します。非同期処理の場合、Kinesis Data Streams を使用して、Lambda 関数の同時実行制御ではなく、1 つのシャードで同時実行を効果的にコントロールします。これにより、シャードの数や並列化係数を増やす柔軟性が得られ、Lambda 関数の同時実行数を増やすことができます。

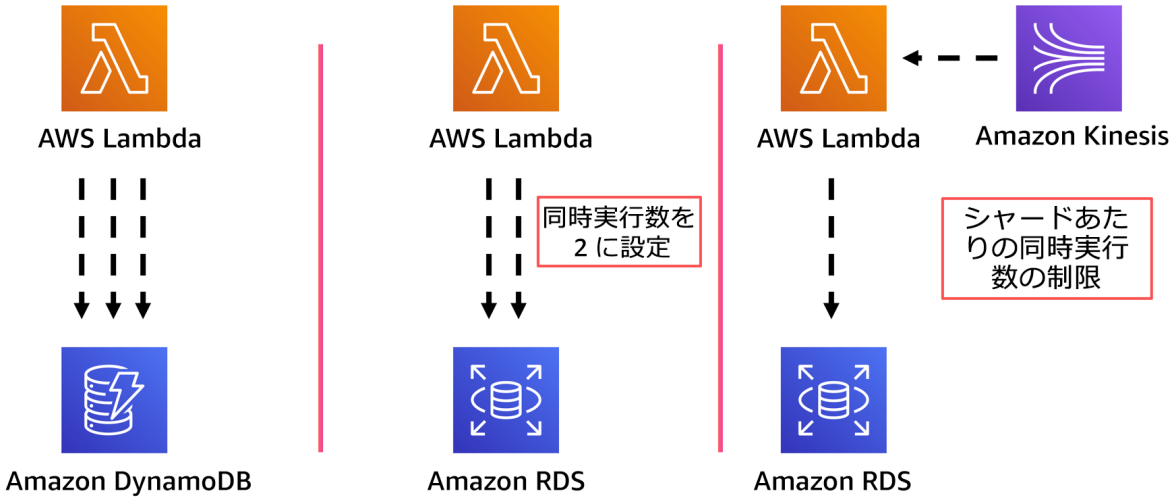

図 17: 同期および非同期リクエストの同時実行のコントロール

REL 2: サーバーレスアプリケーションに弾力性をどのように構築していますか?

非同期呼び出しとイベント

非同期呼び出しにより、HTTP レスポンスのレイテンシーが短縮されます。複数の同期呼び出しと実行時間の長い待機サイクルにより、タイムアウトが発生し、再試行ロジックが防止される「ロックされた」コードが発生する場合があります。

イベント駆動型アーキテクチャにより、コードの非同期実行を合理化できるため、コンシューマーの待機サイクルを制限できます。これらのアーキテクチャは、通常、ビジネス機能を実行する複数のコンポーネントにわたって、キュー、ストリーム、pub/sub、Webhooks、ステートマシン、およびイベントルールマネージャを使用して非同期的に実装されます。

ユーザーエクスペリエンスは非同期呼び出しで疎結合化されます。フロントエンドシステムは、全体的な実行が完了するまでエクスペリエンス全体をブロックするのではなく、最初のリクエストの一部として参照 / ジョブ ID を受け取り、リアルタイムの変更をサブスクライブするか、レガシーシステムでは、追加の API を使用してステータスをポーリングします。この分離により、イベントループ、並列、または同時実行の技術を使用して、フロントエンドがより効率的になり、応答が部分的にまたは完全に利用可能になったときにアプリケーションの一部が遅延してロードされます。

フロントエンドは、カスタム再試行とキャッシュにより堅牢になるため、非同期呼び出しの重要な要素になります。異常、一時的な状態、ネットワーキング、または環境の低下が原因で、許容される SLA 内に応答がない場合、処理中のリクエストを停止できます。

または、同期呼び出しが必要な場合、合計実行時間が API Gateway または AWS AppSync の最大タイムアウトを超えないようにするために、最低限推奨されます。外部サービス (AWS Step Functions など) を使用して、複数のサービス間のビジネストランザクションを調整し、状態をコントロールし、リクエストのライフサイクルに沿って発生するエラー処理を処理します。

変更管理

これは AWS Well-Architected フレームワークで説明されています。サーバーレスに関する具体的な情報は、運用上の優秀性の柱にあります。

障害管理

サーバーレスアプリケーションの特定の部分は、pub/sub などのパターンなど、イベント駆動型の方法でさまざまなコンポーネントへの非同期呼び出しによって決定されます。非同期呼び出しが失敗した場合は、可能な限りキャプチャして再試行する必要があります。そうしないと、データ損失が発生し、カスタマーエクスペリエンスが低下する可能性があります。

Lambda 関数の場合、Lambda クエリに再試行ロジックを構築して、スパイクが発生するワークロードがバックエンドに過負荷にならないようにします。運用上の優秀性の柱で説明されている構造化ロギングを使用して、再試行をログに記録します。これには、カスタムメトリクスとしてキャプチャできるエラーに関するコンテキスト情報が含まれます。Lambda 送信先を使用して、エラー、スタックトレース、再試行に関するコンテキスト情報を SNS トピックや SQS キューなどの専用のデッドレターキュー (DLQ) に送信します。また、これらの失敗したイベントを意図したサービスに戻すために、別のメカニズムでポーリングする計画を開発したいと考えています。

AWS SDK は、ほとんどのケースで十分な他の AWS のサービスとやり取りするときに、デフォルトでバックオフと再試行メカニズムを提供します。ただし、ニーズに合わせて、特に HTTP キープアライブ、接続、ソケットのタイムアウトを確認して調整します。

可能な限り、Step Functions を使用して、サーバーレスアプリケーション内のカスタム試行/キャッチ、バックオフ、再試行ロジックの量を最小限に抑えます。詳細については、コスト最適化の柱のセクションを参照してください。Step Functions 統合を使用して、失敗した状態の実行とその状態を DLQ に保存します。

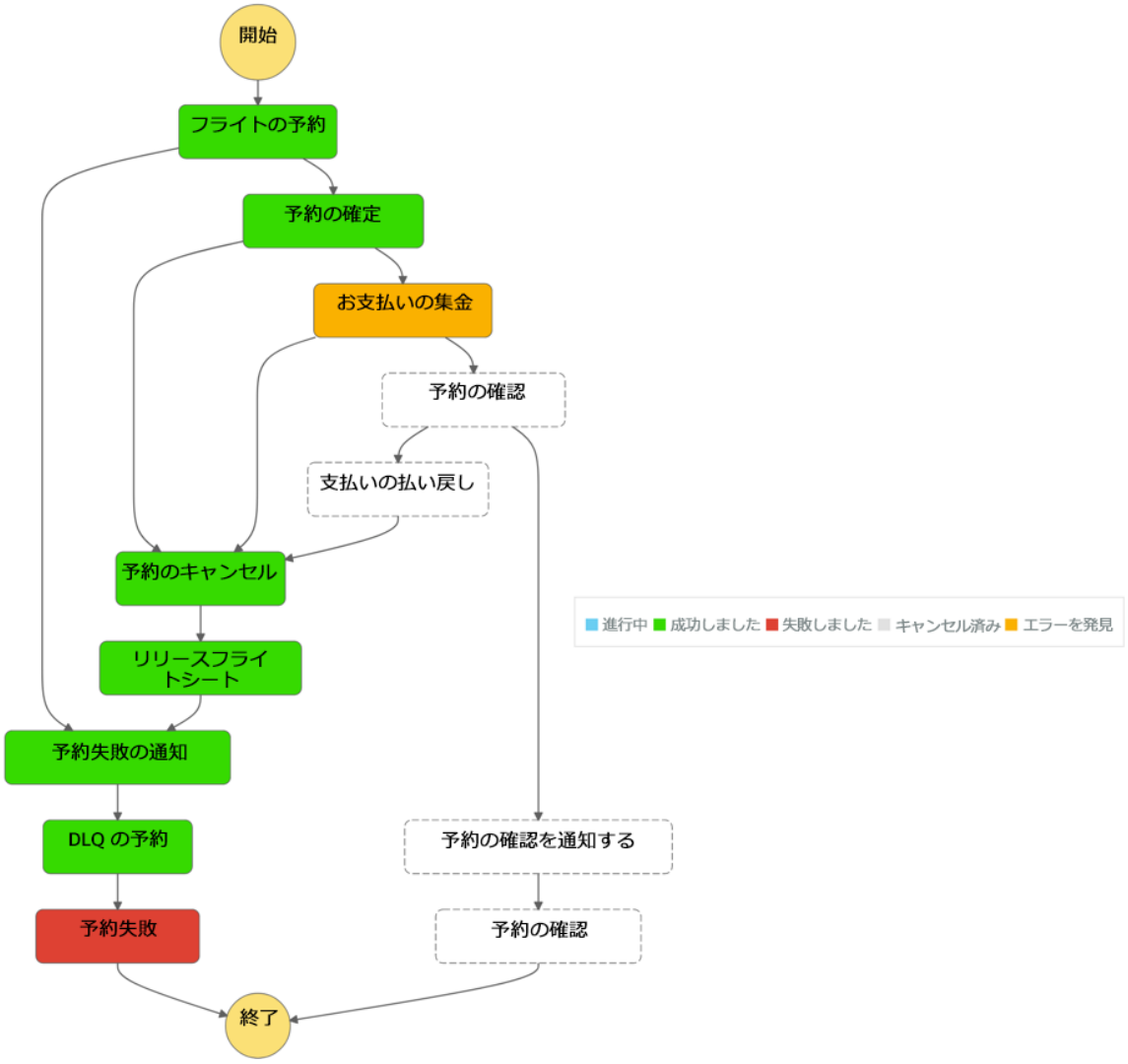

図 18: Step Functions ステートマシンと DLQ ステップ

PutRecords (Kinesis) や BatchWriteItem (DynamoDB) などの非アトミック運用では部分的な障害が発生することがあります。これは、少なくとも 1 つのレコードが正常に取り込まれた場合に成功を返すためです。このような運用を使用するときは、常にレスポンスを検査し、プログラムによって部分的な失敗に対処します。

Kinesis または DynamoDB ストリームから消費する場合は、最大レコード期間、最大再試行回数、失敗時の DLQ、関数エラー時の Bisect バッチなどの Lambda エラー処理コントロールを使用して、アプリケーションにさらなる弾力性を構築します。

トランザクションベースで、特定の保証と要件に依存する同期パートの場合、Saga パターンで説明されているように失敗したトランザクションのロールバックは、アプリケーションのロジックを疎結合化して簡素化する Step Functions ステートマシンを使用することで実現できます。

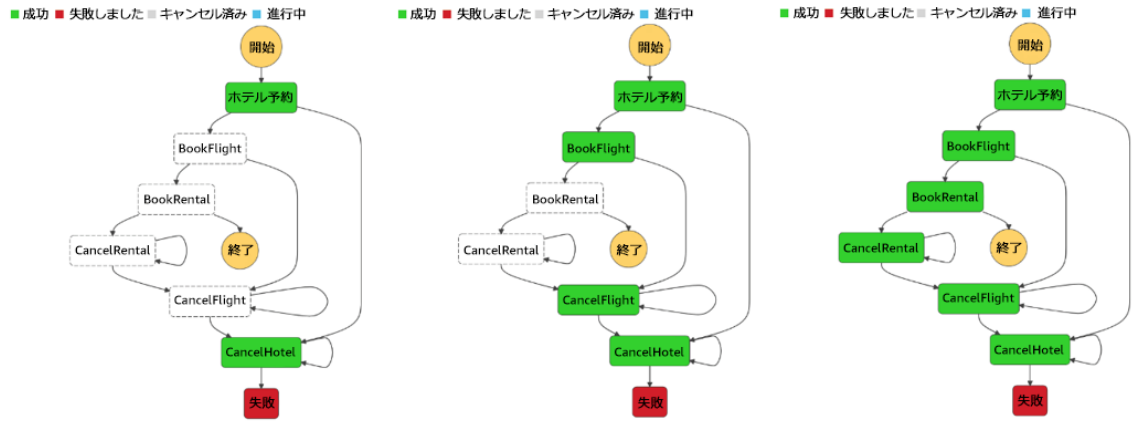

図 19: Yan Cui による Step Functions の Saga パターン

制限

Well-Architected フレームワークで説明されている内容に加えて、バーストおよびスパイクが発生するユースケースの制限を確認することを検討してください。たとえば、API Gateway と Lambda では、一定のリクエスト率とバーストリクエスト率に異なる制限があります。可能な場合はスケーリングレイヤーと非同期パターンを使用し、負荷テストを実行して、現在のアカウント制限が実際の顧客の需要を維持できるようにします。

主要な AWS のサービス

信頼性の重要な AWS のサービスは、AWS Marketplace、Trusted Advisor、CloudWatch Logs、CloudWatch、API Gateway、Lambda、X-Ray、Step Functions、Amazon SQS、Amazon SNS です。

リソース

信頼性のベストプラクティスの詳細については、以下のリソースを参照してください。

ドキュメントとブログ

- Lambda の制限
- API Gateway の制限
- Kinesis Streams の制限
- DynamoDB の制限
- Step Functions の制限
- エラー処理パターン
- Lambda を使用したサーバーレスのテスト
- Lambda 関数ログのモニタリング
- バージョン管理 Lambda
- API Gateway のステージ
- AWS での API の再試行
- Step Functions のエラー処理
- X-Ray

- Lambda DLQ
- API Gateway および Lambda でのエラー処理パターン
- Step Functions の待機状態
- Saga パターン
- Step Functions による Saga パターンの適用
- サーバーレスアプリケーションのリポジトリアプリ – DLQ Redriver
- AWS SDK での再試行とタイムアウトの問題のトラブルシューティング
- ストリーム処理のための Lambda 弾力性コントロール
- Lambda 送信先
- サーバーレスアプリケーションのリポジトリアプリ – イベントの再生
- サーバーレスアプリケーションのリポジトリアプリ – イベントのストレージとバックアップ

ホワイトペーパー

- AWS におけるマイクロサービス

パフォーマンス効率の柱

パフォーマンス効率の柱は、コンピューティングリソースを効率的に使って要件を満たし、需要が変化し技術が発展するのに合わせて効率性を維持する能力に焦点を当てます。

定義

クラウドでのパフォーマンス効率には以下の 4 つの領域があります。

- 選択
- レビュー
- モニタリング
- トレードオフ

高性能なアーキテクチャの選択には、データ主導のアプローチを取ります。ハイレベルな設計から、リソースタイプの選択と設定に至るまで、アーキテクチャのあらゆる側面に関するデータを収集してください。選択した内容を定期的にレビューすることで、絶えず進化し続けている AWS クラウドを活用できているかを確認できます。

モニタリングを実施すれば、予想したパフォーマンスからのずれを把握し、その対策を講じることができます。最後に、圧縮またはキャッシュの使#、または整合性要件の緩和など、アーキテクチャのトレードオフによってパフォーマンスを向上できます。

PER 1: 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?

選択

安定したバーストレートを使用して、サーバーレスアプリケーションでパフォーマンステストを実行します。結果を使用して、キャパシティーユニットを調整し、変更後のロードテストを試して、最適な設定を選択できるようにしてください。

- Lambda: CPU、ネットワーク、およびストレージの IOPS が比例的に割り当てられるため、異なるメモリ設定をテストします。
- API Gateway: 地理的に分散した顧客にはエッジエンドポイントを使用します。リージョンのお客様や、同じリージョン内で他の AWS のサービスを使用する場合は、リージョンを使用します。
- DynamoDB: 予測不可能なアプリケーショントラフィックにはオンデマンドを使用し、整合性のあるトラフィックにはプロビジョニングモードを使用します。
- Kinesis: 複数のコンシューマーシナリオで、コンシューマーごとの専用の入力/出力チャネルに拡張ファンアウトを使用します。Lambda を使用した低ボリュームのトランザクションには、拡張バッチウィンドウを使用します。

必要な場合にのみ Lambda 関数への VPC アクセスを設定します。VPC 対応 Lambda 関数がインターネットにアクセスする必要がある場合は、NAT ゲートウェイをセットアップします。Well-Architected フレームワークで説明されているように、高可用性とパフォーマンスを実現するために、複数のアベイラビリティーゾーンにまたがる NAT ゲートウェイを設定します。

API Gateway エッジ最適化 API は、完全マネージド型の CloudFront ディストリビューションを提供し、地理的に分散したコンシューマーのアクセスを最適化します。API リクエストは、最も近い CloudFront 接続ポイント (POP) にルーティングされるため、通常は接続時間が短縮されます。

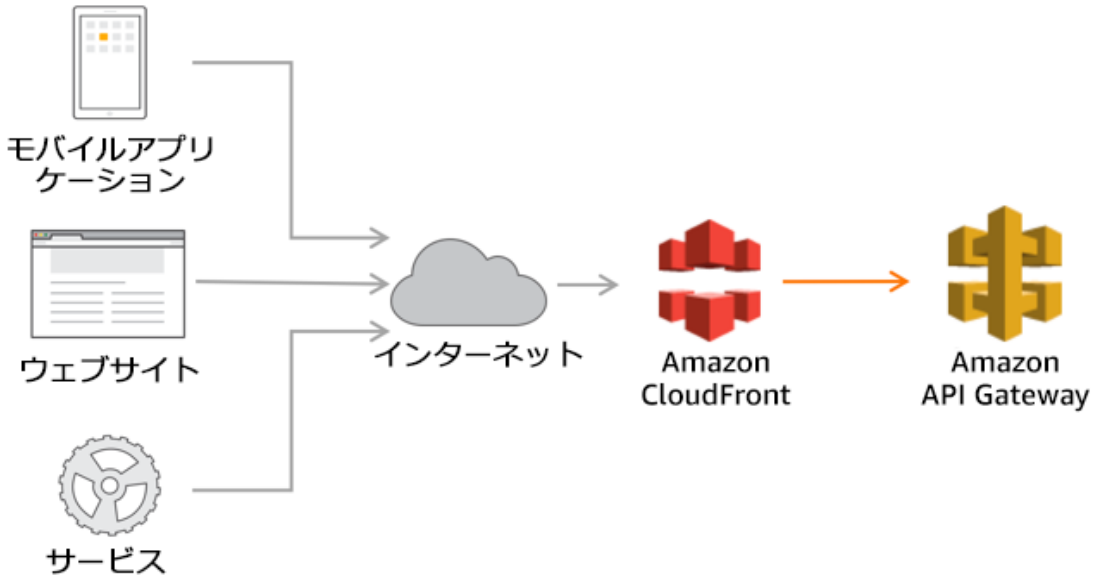

図 20: エッジ最適化 API Gateway のデプロイ

API Gateway 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を提供せず、デフォルトで HTTP2 を有効にします。これにより、リクエストが同じリージョンから送信される場合の全体的なレイテンシーを削減できます。リージョン別エンドポイントでは、独自の Amazon CloudFront ディストリビューションまたは既存の CDN を関連付けることもできます。

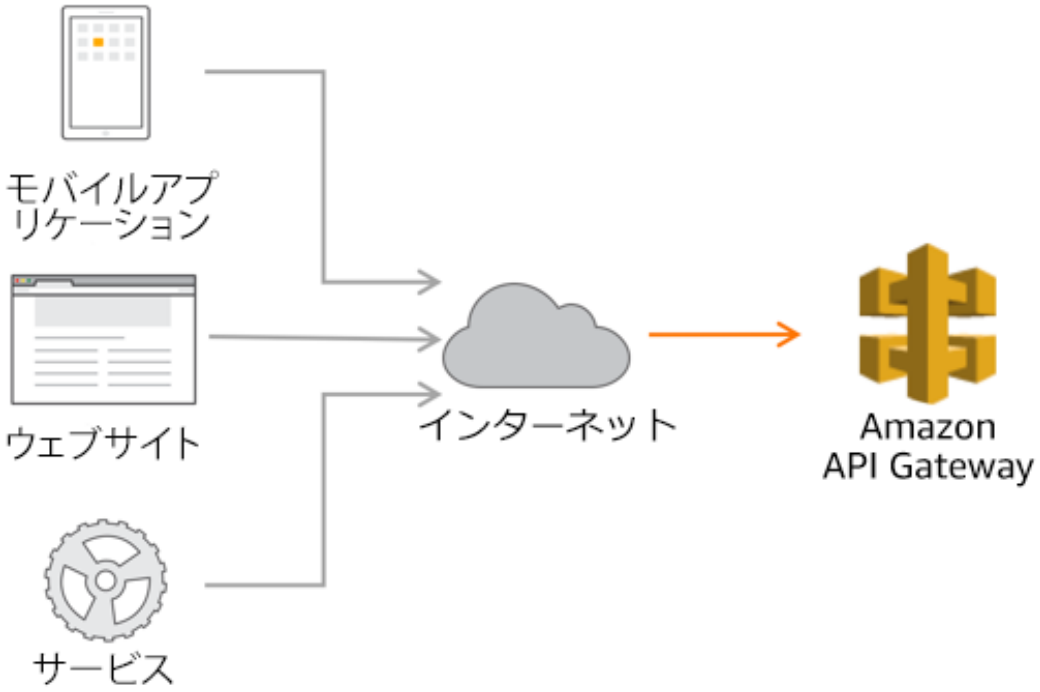

図 21: リージョン別エンドポイント API Gateway のデプロイ

このテーブルは、デプロイおよびエッジ最適化 API エンドポイントまたはリージョン API エンドポイントのどちらを行うかを決定するのに役立ちます。

	エッジ最適化 API	リージョン別 API エンドポイント
API は、リージョン間でアクセスされます。API Gateway で管理された CloudFront ディストリビューションが含まれます。	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のリクエストレイテンシーは最小です。		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

この決定ツリーは、Lambda 関数を VPC にデプロイするタイミングを決定するのに役立ちます。

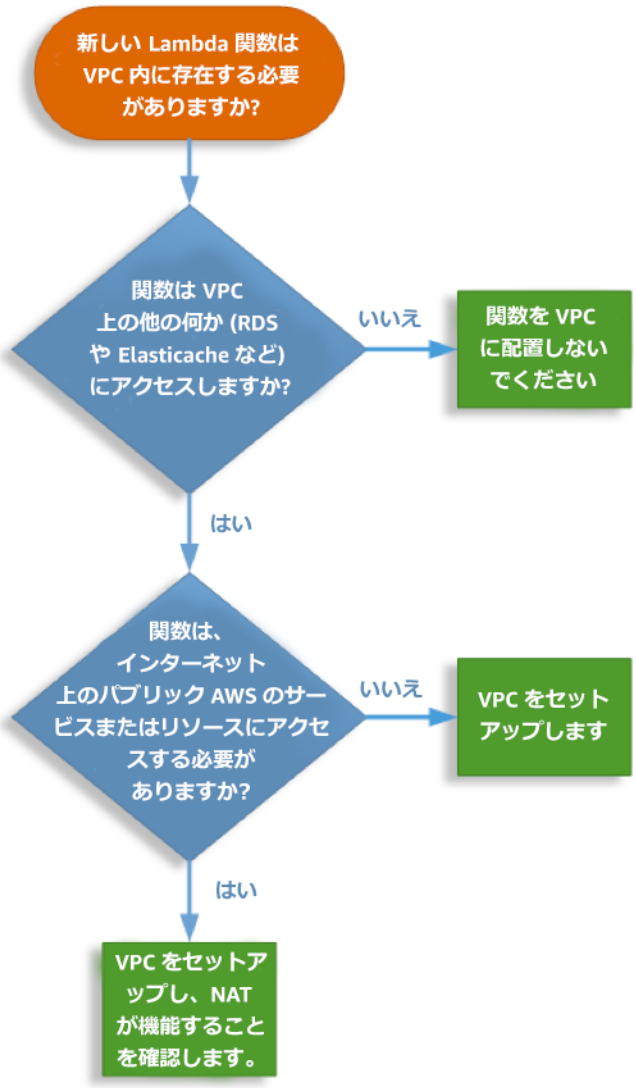

図 22: VPC に Lambda 関数をデプロイするための決定ツリー

最適化

サーバーレスアーキテクチャが有機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る特定のメカニズムがあります。パフォーマンステストにもかかわらず、アプリケーションのパフォーマンスを向上させるには、設計上のトレードオフを考慮し、SLA と要件を常に念頭に置いてください。

API Gateway と AWS AppSync キャッシュを有効にして、該当する運用のパフォーマンスを向上させることができます。DAX は、DynamoDB の完全テーブルスキャン運用を防ぐために、読み込みレスポンスを大幅に向上させ、グローバルセカンダリインデックスとローカルセカンダリインデックスを改善できます。これらの詳細とリソースは、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 のコンテンツエンコーディングにより、API クライアントは API リクエストへのレスポンスで返送される前にペイロードを圧縮するよう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減り、データの転送にかかる時間が短縮されます。API 定義でコンテンツエンコーディングを有効にし、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 ではコンテンツエンコーディングのサポートが有効になっていません。

関数のタイムアウトを平均実行数より数秒高く設定し、通信パスで使用されるダウンストリームサービスの一時的な問題に対処します。これは、Step Functions のアクティビティ、タスク、SQS メッセージの可視性を使用する場合にも適用されます。

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、運用手順に望ましくない影響が生じる場合があります。

タイムアウトを平均実行よりかなり高く設定すると、コードの誤動作時に関数が長期間実行される可能性があり、その結果、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数の制限に達する可能性があります。

関数の正常な実行に等しいタイムアウトを設定すると、一時的なネットワーキングの問題やダウンストリームサービスで異常が発生した場合に、サーバーレスアプリケーションが実行を突然停止する可能性があります。

負荷テストを実行せずにタイムアウトを設定すると、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、いずれかのパートが最初にタイムアウトに達するたびにエラーが発生する可能性があります。

コンテナの再利用、ランタイムに必要なデプロイパッケージサイズの最小化、高速起動用に最適化されないフレームワークなど、依存関係の複雑さの最小化といった Lambda 関数の操作に関するベストプラクティスに従います。レイテンシー 99^{の} パーセンタイル (P99) は常に考慮する必要があります。他のチームと合意されたアプリケーション SLA に影響を与えないためです。

VPC の Lambda 関数の場合、VPC 内の基盤となるリソースのパブリックホスト名の DNS 解決は避けてください。たとえば、Lambda 関数が VPC 内の Amazon RDS DB インスタンスにアクセスする場合、no-publicly-accessible オプションを使用してインスタンスを起動します。

Lambda 関数が実行された後、AWS Lambda は別の Lambda 関数の呼び出しを予期して、任意の時間実行コンテキストを維持します。これにより、データベース接続や初期化ロジックの確立など、1 回限りのコストの高い運用にグローバルスコープを使用できます。後続の呼び出しでは、それがまだ有効であるかどうかを確認し、既存の接続を再利用できます。

非同期トランザクション

顧客はよりモダンでインタラクティブなユーザーインターフェイスを期待しているため、同期トランザクションを使用して複雑なワークフローを維持することはできなくなります。必要なサービスのやり取りが増えるほど、呼び出しを連鎖させることになり、サービスの安定性や応答時間に関するリスクが増える可能性があります。

Angular.js、VueJS、React、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などの最新の UI フレームワークは、顧客の需要を満たす持続可能なアプローチを提供します。また、コンポーネントを疎結合化し、代わりにプロセスドメインとビジネスドメインに集中するのに役立ちます。

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として記述されることも多い) は、クライアントがレスポンスのロックと待機 (I/O ブロック) に制約するのではなく、クラウド内でダウンストリームの後続の振り付けイベントを開始します。非同期ワークフローは、データの取り込み、ETL 運用、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。

このようなユースケースでは、データは到着時に処理され、変更されると取得されます。統合と非同期処理の最適化パターンを学習できる 2 つの一般的な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。

サーバーレスのデータ処理

サーバーレス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 に届きます。

新しいオブジェクトは、自動的に実行される Lambda 関数から開始されます。この関数は一般的に、さらなる処理のためにデータを変換またはパーティション化するために使用され、DynamoDB などの他の送信先、またはデータが最終的な形式である別の S3 バケットに保存される場合があります。

データ型ごとに異なる変換があることがあるため、最適なパフォーマンスを得るために変換を異なる Lambda 関数に細かく分割することをお勧めします。このアプローチにより、データ変換を並行して実行し、スピードとコストを実現する柔軟性が得られます。

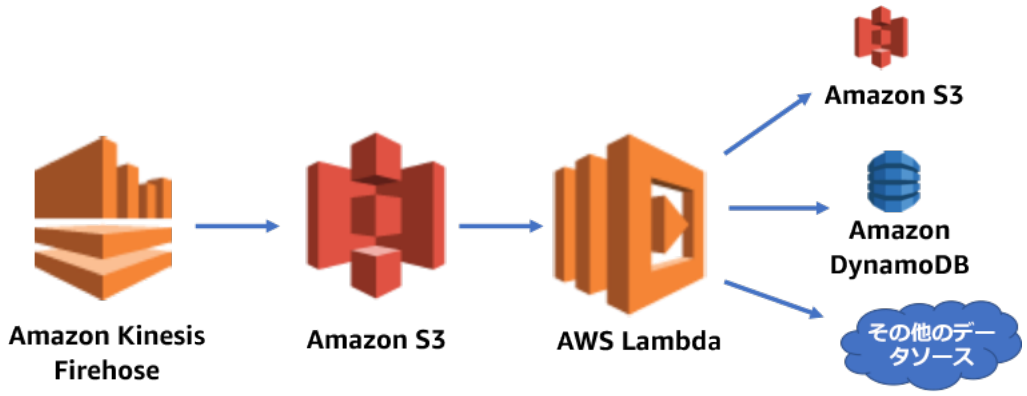

図 23: 非同期データの取り込み

Kinesis Data Firehose では、Lambda の代わりに使用できるネイティブデータ変換を利用できます。この変換では、Apache ログ/システムログのレコードを CSV または JSON に、あるいは JSON を Parquet または ORC に変換するための追加のロジックは必要ありません。

ステータス更新によるサーバーレスイベント送信

e コマースサイトがあり、顧客が在庫の控除と出荷プロセスを開始する注文、または応答に数分かかることがあ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を送信するとします。

この共通のトランザクションを完了するために必要なプロセスでは、複数のサービス呼び出しが必要であり、完了までに数分かかる場合があります。これらの呼び出し内では、再試行とエクスポネンシャルバックオフを追加することで、潜在的な失敗から保護する必要があります。ただし、トランザクションの完了を待機しているユーザーエクスペリエンスが最適でない可能性があります。

このような長くて複雑なワークフロー の場合は、API Gateway または AWS AppSync を Step Functions と統合できます。これにより、新しい承認されたリクエストによってこのビジネスワークフローが開始されます。Step Functions は、発信者 (モバイルアプリ、SDK、ウェブサービスなど) に実行 ID で即座に応答します。

レガシーシステムの場合、実行 ID を使用して、別の REST API 経由で Step Functions のビジネスワークフローステータスをポーリングできます。WebSocket では、REST と GraphQL のいずれを使用している場合でも、ワークフローの各ステップで更新を提供することで、ビジネスワークフローのステータスをリアルタイムで受け取ることができます。

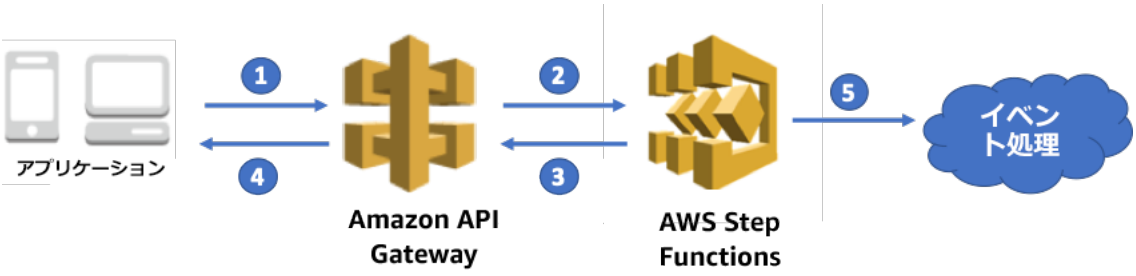

図 24: Step Functions ステートマシンを使用した非同期ワークフロー

もう 1 つの一般的なシナリオは、スケーリングレイヤーとして API Gateway を SQS または Kinesis と直接統合することです。Lambda 関数は、発信者から追加のビジネス情報またはカスタムリクエスト ID 形式が予期される場合にのみ必要です。

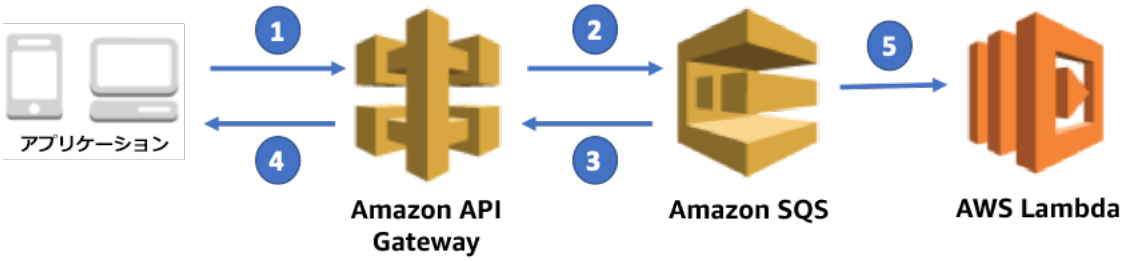

図 25: スケーリングレイヤーとしてキューを使用する非同期ワークフロー

この 2 番目の例では、SQS は複数の目的を果たしています。

1. リクエストレコードを永続的に保存することは重要です。クライアントが確実にワークフロー全体を通して進み、最終的にリクエストが処理されることが分かるためです。
2. 一時的にバックエンドを圧倒する可能性のあるイベントのバースト時に、リソースが使用可能になったときにリクエストを処理するためにポーリングできます。

キューのない最初の例と比較すると、Step Functions はキューやステート追跡データソースを必要とせずに、永続的にデータを保存します。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に非同期ワークフローを追求し、完了に数分かかることがある場合は、その結果のレスポンスをブロックコードとして回避することです。

WebSockets では、AWS AppSync は GraphQL サブスクリプションを介してこの機能をすぐに利用できます。サブスクリプションを使用すると、承認されたクライアントは、関心のあるデータミューテーションをリッスンできます。これは、ストリーミング中のデータや、複数のレスポンスを返すデータに最適です。

AWS AppSync では、DynamoDB でステータスの更新が変化すると、クライアントは更新が発生したときに自動的にサブスクライブして受信できます。データがユーザーインターフェイスを駆動するタイミングに最適なパターンです。

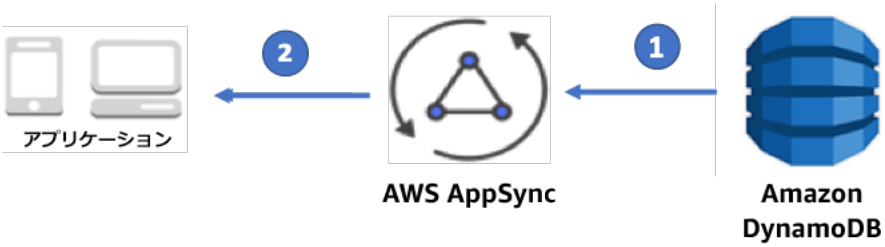

図 26: AWS AppSync および GraphQL を使用した WebSockets による非同期更新

ウェブフックは、SNS トピックの HTTP サブスクリプションで実装できます。コンシューマーは、イベント発生時に SNS が POST メソッドを介してコールバックする HTTP エンドポイント (Amazon S3 に到着するデータファイルなど) をホストできます。このパターンは、エンドポイントをホストできる別のマイクロサービスなど、クライアントを設定できる場合に最適です。または、Step Functions は、特定のタスクのレスポンスを受け取るまでステートマシンがブロックするコールバックをサポートします。

図 27: SNS を使用した Webhook 経由の非同期通知

最後に、複数のクライアントが常に API をポーリングしてステータスを確認するため、ポーリングはコスト面とリソース面の両方からコストがかかることもあります。環境の制約のためにポーリングが唯一のオプションである場合は、「空のポーリング」の数を制限するために、クライアントとの SLA を確立することがベストプラクティスです。

図 28: 最近行われたトランザクションの更新をクライアントがポーリングする

たとえば、大規模なデータウェアハウスのクエリで応答に平均 2 分かかる場合、データが利用できない場合、クライアントは 2 分後に API をポーリングし、エクスポネンシャルバックオフを使用します。クライアントが予想よりも頻繁にポーリングしないようにするには、2 つの一般的なパターンがあります。スロットリングとタイムスタンプ (再度ポーリングしても安全である場合のタイムスタンプ) です。

タイムスタンプの場合、ポーリング中のシステムは、コンシューマーが再度ポーリングしても安全である場合に、タイムスタンプまたは期間を含む追加のフィールドを返すことができます。このアプローチは、コンシューマーがこれを尊重して賢明に使用し、不正使用が発生した場合にスロットリングを採用してより完全な実装を行うことができるというオプティミスティックシナリオに従います。

レビュー

サーバーレスアプリケーションに適用されるパフォーマンス効率について、レビュー領域のベストプラクティスについては、AWS Well-Architected フレームワークホワイトペーパーを参照してください。

モニタリング

サーバーレスアプリケーションに適用されるパフォーマンス効率について、モニタリング領域のベストプラクティスについては、AWS Well-Architected フレームワークホワイトペーパーを参照してください。

トレードオフ

サーバーレスアプリケーションに適用されるパフォーマンス効率のトレードオフ領域のベストプラクティスについては、AWS Well-Architected フレームワークホワイトペーパーを参照してください。

主要な AWS のサービス

パフォーマンスを効率化するための主要な AWS のサービスは、DynamoDB Accelerator、API Gateway、Step Functions、NAT ゲートウェイ、Amazon VPC、Lambda です。

リソース

当社のパフォーマンス効率性のベストプラクティスの詳細については、以下のリソースを参照してください。

ドキュメントとブログ

- AWS Lambda に関するよくある質問
- AWS Lambda Functions を使用するためのベストプラクティス
- AWS Lambda: 仕組み
- AWS Lambda でのコンテナの再利用について
- Amazon VPC 内のリソースにアクセスできるように Lambda 関数を設定する
- API キャッシュを有効にして応答性を向上させる
- DynamoDB: グローバルセカンダリインデックス
- Amazon DynamoDB Accelerator (DAX)
- デベロッパーガイド: Kinesis Streams
- Java SDK: パフォーマンス向上の設定
- Node.js SDK: HTTP キープアライブを有効にする
- Node.js SDK: インポートの向上
- Amazon SQS キューと AWS Lambda を使用した高スループット
- 拡張ファンアウトによるストリーム処理パフォーマンスの向上
- Lambda パワーチューニング
- Amazon DynamoDB のオンデマンドモードとプロビジョニングモードを使用するタイミング
- Amazon CloudWatch Logs Insights を使用したログデータの分析
- 複数のデータソースと AWS AppSync の統合
- Step Functions サービスの統合
- キャッシュパターン
- サーバーレスアプリケーションのキャッシング
- Amazon Athena と AWS Glue のベストプラクティス

コスト最適化の柱

コスト最適化の柱には、システムのライフサイクル全体を通じた微調整と改善の継続的なプロセスが含まれます。最初の PoC (実証支援) の初期設計から、製品ワークロードの継続運用まで、コスト意識の高いシステムを構築・運用する際にドキュメントのプラクティスを採用すれば、ビジネス上の成果を上げながら、コストの最小化と投資利益率の最大化の両方を達成できるようになります。

定義

4 つの分野のベストプラクティスなど、クラウド内でのコストの最適化:

- 費用対効果の高いリソース
- 需要と供給の一致
- 費用認識
- 長期的な最適化

他の柱と同様、検討する必要があるトレードオフも存在します。例えば、最適化したいのは、市場投入までの速度とコストのどちらですか? 市場投入のスピードを向上する、新しい機能を導入する、締め切りに間に合わせるといったケースでは、前払いコストの投資を最適化するよりも、スピードを重視して最適化することが最善です。

設計上の決定は、実際のデータに導かれずに、急いで行われる場合があります。なぜなら、コスト最適化のベンチマークを行う時間をかけず、過剰な投資を「念のため」行う傾向があるからです。

その結果、過剰なプロビジョニングと、あまり最適化されていないデプロイが生じることになります。以下のセクションでは、デプロイにおける初期段階でのコスト最適化と継続的なコスト最適化について、そのテクニックと戦略的ガイダンスを紹介します。

一般的に、サーバーレスアーキテクチャはコストを削減する傾向があります。これは、AWS Lambda などの一部のサービスでは、アイドル状態の間は何も費用がかからないためです。ただし、特定のベストプラクティスに従ってトレードオフを行うと、これらのソリューションのコストをさらに削減できます。

ベストプラクティス

コスト 1: コストをどのように最適化しますか?

費用対効果の高いリソース

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観点から管理しやすくなります。従量制の料金モデルと需要に基づくスケーリングにより、サーバーレスはキャパシティープランニングの労力を効果的に削減します。

運用上の優秀性とパフォーマンスの柱で取り上げられているように、サーバーレスアプリケーションを最適化することは、それがもたらす価値とそのコストに直接影響します。

Lambda はメモリに基づいて CPU、ネットワーク、およびストレージの IOPS を比例的に割り当てるため、実行速度が速くなるほど、課金増分ディメンションが 100 ミリ秒であるため、関数が生成する価値が高くなります。

需要と供給の一致

AWS サーバーレスアーキテクチャは、需要に基づいてスケールするように設計されているため、適用可能なプラクティスはありません。

費用認識

AWS Well-Architected フレームワークで対象としているとおり、クラウドによる柔軟性と俊敏性の向上は、イノベーションを促進し、開発とデプロイのペースを高めます。クラウドによってオンプレミスインフラストラクチャのプロビジョニングに関連した手動プロセスや時間を省くことができます。これにはハードウェア仕様の決定、価格交渉、注文管理、発送のスケジュール設定、リソースのデプロイなどが含まれます。

サーバーレスアーキテクチャが増えるにつれて、Lambda 関数、API、ステージ、その他のアセットの数が倍増します。これらのアーキテクチャのほとんどは、コストとリソース管理の観点から予算と予測する必要があります。タグ付けはここで役立ちます。AWS 請求書から個々の関数と API にコストを割り当て、AWS Cost Explorer でプロジェクトごとのコストを詳細に確認できます。

適切な実装は、プロジェクトに属するアセットの同じキーと値のタグをプログラムで共有し、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、コストを割り当てるだけでなく、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役立ちます。

長期的な最適化

サーバーレスアプリケーションに適用されるコスト最適化については、時間の経過に伴う最適化領域のベストプラクティスについては、AWS Well-Architected フレームワークホワイトペーパーを参照してください。

ログ記録の取り込みとストレージ

AWS Lambda は CloudWatch Logs を使用して実行の出力を保存し、実行に関する問題を特定してトラブルシューティングし、サーバーレスアプリケーションをモニタリングします。これらは、CloudWatch Logs サービスのコストに、取り込みとストレージという 2 つの側面で影響を与えます。

適切なログ記録レベルを設定し、不要なログ記録情報を削除して、ログの取り込みを最適化します。環境変数を使用してアプリケーションのログレベルをコントロールし、DEBUG モードでのサンプルログ記録を制御し、必要に応じて追加の洞察を得られるようにします。

新規および既存の CloudWatch Logs グループのログ保持期間を設定します。ログのアーカイブには、ニーズに最適なコスト効率の高いストレージクラスをエクスポートして設定します。

直接統合

Lambda 関数が他の AWS のサービスとの統合中にカスタムロジックを実行していない場合、不要である可能性があります。

API Gateway、AWS AppSync、Step Functions、EventBridge、および Lambda 送信先は、多数のサービスと直接統合でき、より多くの価値と運用オーバーヘッドを削減できます。

ほとんどのパブリックサーバーレスアプリケーションは、RESTful マイクロサービス (p. 9)で説明されているように、提供された契約を非依存に実装する API を提供します。

直接統合が適しているシナリオの例は、REST API を通じてクリックストリームデータを取り込むことです。

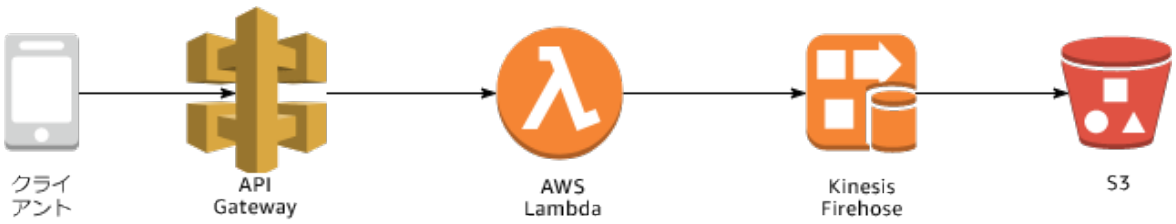

図 29: Kinesis Data Firehose を使用した Amazon S3 へのデータの送信

このシナリオでは、API Gateway は受信レコードを Kinesis Data Firehose に取り込み、その後 S3 バケットに保存する前にレコードをバッチ処理する Lambda 関数を実行します。この例では追加のロジックは必要ないため、API Gateway サービスプロキシを使用して Kinesis Data Firehose と直接統合できます。

図 30: AWS のサービスプロキシの実装による Amazon S3 へのデータ送信コストの削減

このアプローチにより、API Gateway 内に AWS のサービスプロキシを実装することで、Lambda と不要な呼び出しの使用コストを削減できます。トレードオフとして、取り込みレートを満たすために複数のシャードが必要な場合、これは多少複雑になる可能性があります。

レイテンシーが重要な場合は、抽象化、契約、および API 機能を犠牲にして、正しい認証情報を持つことで Kinesis Data Firehose に直接データをストリーミングできます。

図 31: Kinesis Data 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 へのデータ送信コストを削減する

VPC またはオンプレミスの内部リソースに接続する必要があり、カスタムロジックが必要ないシナリオでは、API Gateway プライベート統合を使用します。

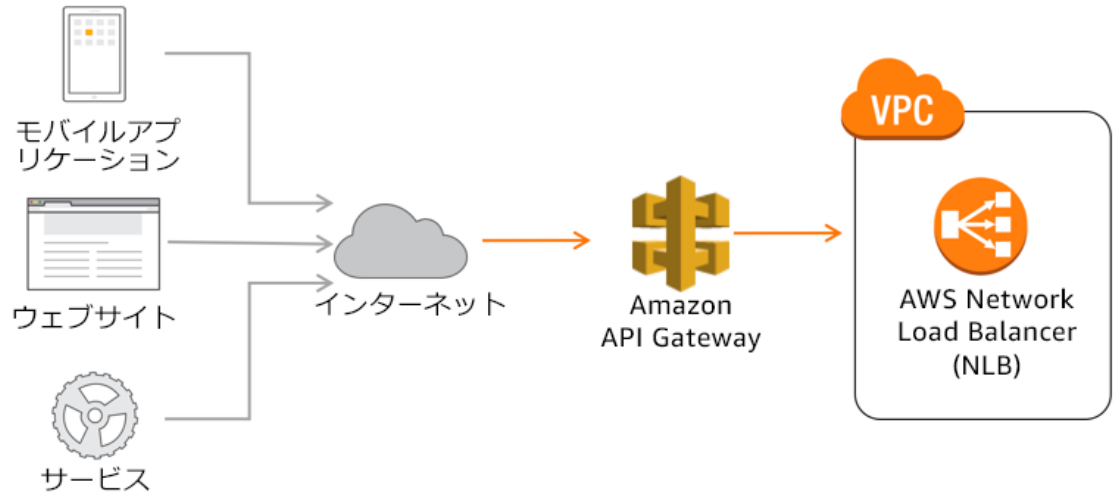

図 32: プライベートリソースにアクセスするための VPC 内の Lambda を介した Amazon API Gateway のプライベート統合

この方法では、API Gateway は受信リクエストを VPC で所有する内部 Network Load Balancer に送信します。これにより、同じ VPC 内または IP アドレス経由でオンプレミスで任意のバックエンドにトラフィックを転送できます。

このアプローチには、承認、スロットリング、キャッシュメカニズムの利点が追加された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、コストとパフォーマンスの両方の利点があります。

もう 1 つのシナリオは、Amazon SNS がすべての受信者にメッセージをブロードキャストするファンアウトパターンです。このアプローチでは、不要な Lambda 呼び出しをフィルタリングして回避するための追加のアプリケーションロジックが必要です。

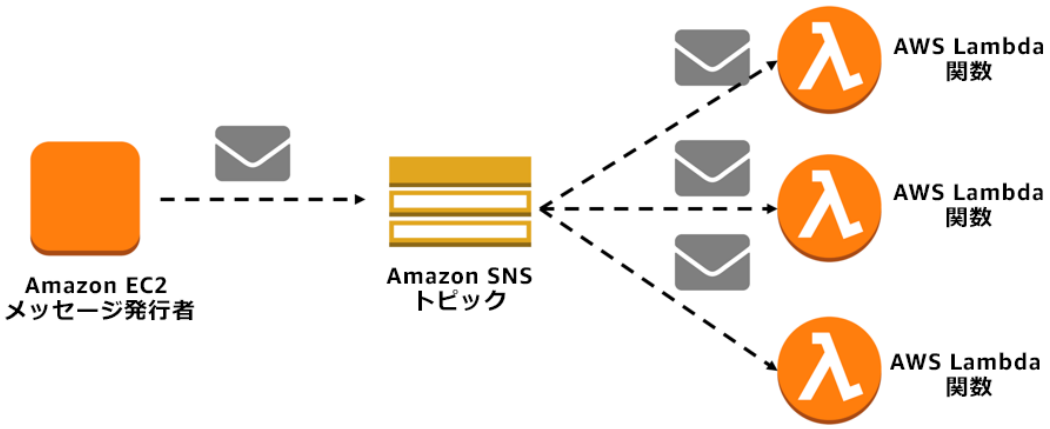

図 33: メッセージ属性のフィルタリングなしの Amazon SNS

SNS は、メッセージ属性に基づいてイベントをフィルタリングし、より効率的に適切な受信者にメッセージを配信できます。

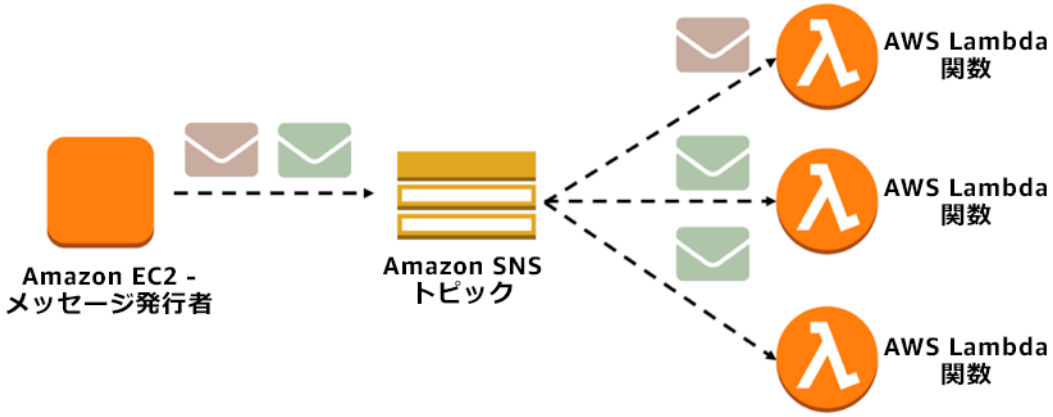

図 34: メッセージ属性フィルタリングを使用した Amazon SNS

もう 1 つの例は、タスクの完了を待ってから次のステップに進む必要がある場合がある処理タスクです。この待機状態は Lambda コード内に実装できますが、イベントを使用して非同期処理に変換するか、Step Functions を使用して待機状態を実装する方がはるかに効率的です。

たとえば、次のイメージでは、AWS Batch ジョブをポーリングし、30 秒ごとにその状態を確認して、完了したかどうかを確認します。Lambda 関数内でこの待機をコーディングする代わりに、ポーリング (GetJobStatus) + wait (Wait30Seconds) + decider (CheckJobStatus) を実装します。

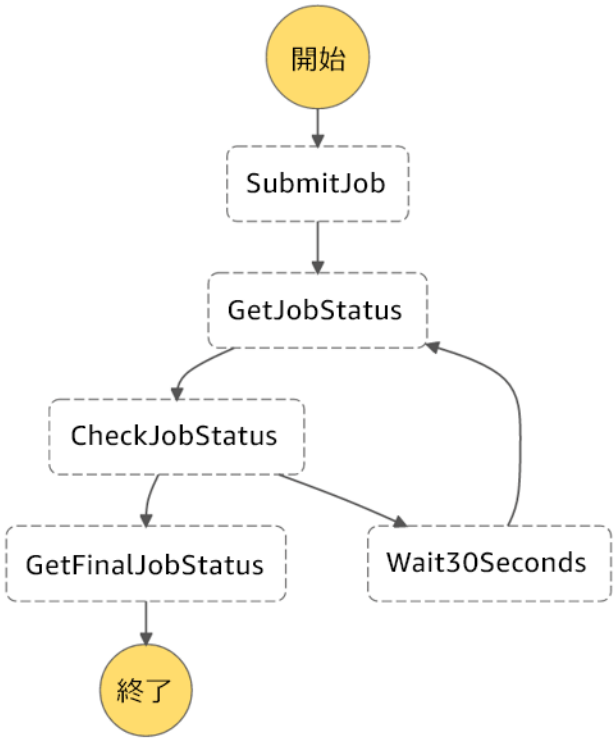

図 35: AWS Step Functions を使用した待機状態の実装

Step Functions で待機状態を実装しても、それ以上のコストは発生しません。Step Functions の料金モデルは、状態内で費やされた時間ではなく状態間の遷移に基づいているためです。

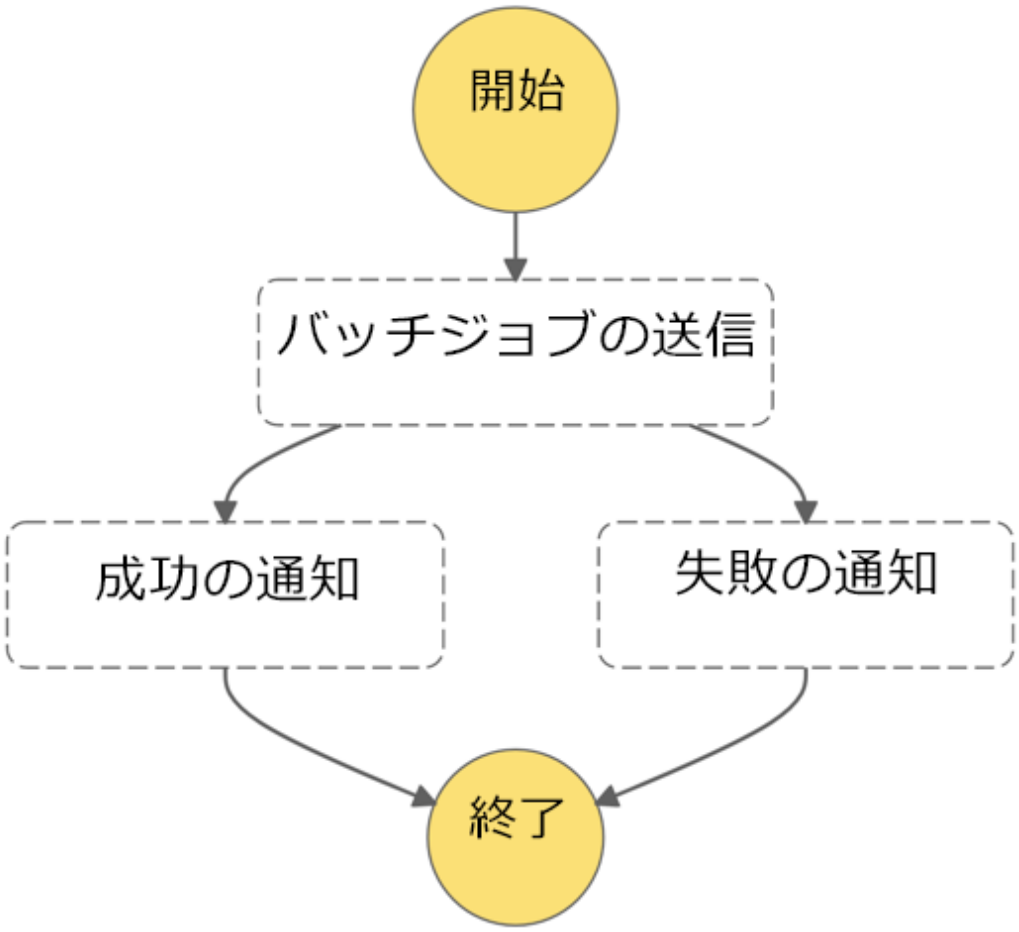

図 36: Step Functions サービス統合同期待機

待機する必要がある統合に応じて、Step Functions では同期的に待機してから次のタスクに移行できるため、追加の移行を節約できます。

コスト最適化

パフォーマンスの柱で説明されているように、サーバーレスアプリケーションを最適化することで、実行ごとに生成される価値を効果的に向上させることができます。

データストアやその他のサービスやリソースへの接続を維持するためにグローバル変数を使用すると、パフォーマンスが向上し、実行時間が短縮され、コストも削減されます。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。

マネージドサービス機能を使用して実行あたりの値を向上させることができる例としては、Amazon S3 からのオブジェクトの取得とフィルタリングがあります。Amazon S3 から大きなオブジェクトをフェッチするには、Lambda 関数に大量のメモリが必要です。

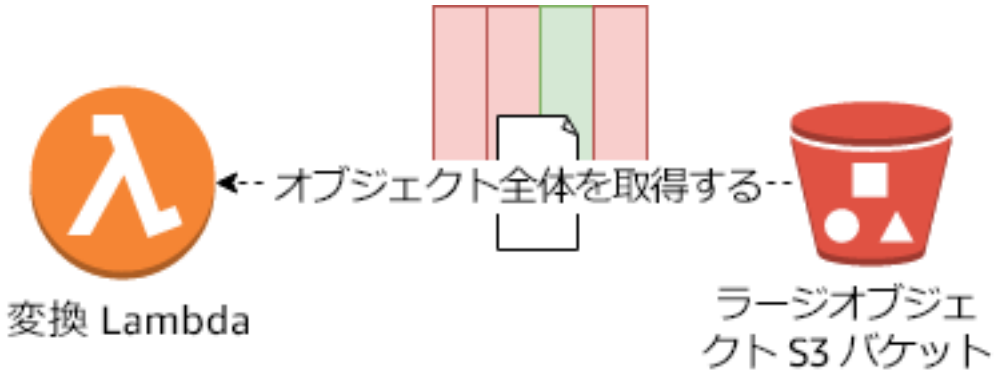

図 37: 完全な S3 オブジェクトを取得する Lambda 関数

前の図では、Amazon S3 から大きなオブジェクトを取得するときに、Lambda のメモリ消費量が増え、実行時間が長くなり (関数で必要なデータを変換、反復処理、または収集可能にするため)、場合によっては、この情報の一部のみが必要になることがわかります。

これは、赤色の 3 つの列 (データ不要) と緑色の 1 つの列 (データ必須) で表されます。Athena SQL クエリを使用して実行に必要な詳細な情報を収集すると、変換を実行する取得時間とオブジェクトサイズが短縮されます。

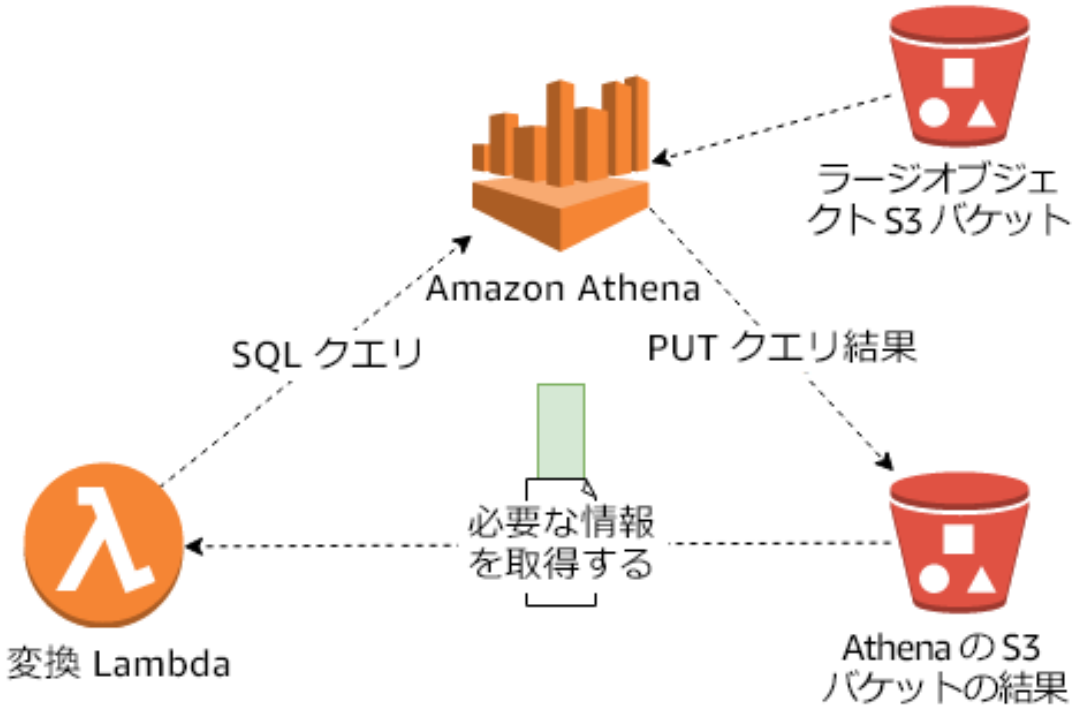

図 38: Lambda と Athena オブジェクトの取得

次の図では、Athena にクエリを実行して特定のデータを取得することで、取得されるオブジェクトのサイズが縮小され、追加の利点として、Athena はクエリ結果を S3 バケットに保存し、結果が Amazon S3 に非同期で着地すると Lambda 呼び出しを呼び出すため、そのコンテンツを再利用できます。

S3 Select で使用する場合も同様です。S3 Select はアプリケーションがシンプルな SQL 式によりオブジェクトからデータのサブセットのみを取得することができるようにします。Athena での前の例と同様に、Amazon S3 から小さなオブジェクトを取得すると、Lambda 関数によって使用される実行時間とメモリが削減されます。

200 秒	95 秒間

# すべてのキーをダウンロードして処理します 	
 	
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]
 	
….
	# IP アドレスとキーを選択します
	
	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:
	
	….

図 39: Amazon S3 と S3 Select を使用した Lambda パフォーマンス統計

リソース

コスト最適化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。

ドキュメントとブログ

- CloudWatch Logs 保持
- Amazon S3 への CloudWatch Logs のエクスポート
- CloudWatch Logs から Amazon ES へのストリーミング
- Step Functions ステートマシンでの待機状態の定義
- Step Functions を搭載した Coca-Cola Vending Pass ステートマシン
- AWS での高スループットのゲノミクスバッチワークフローの構築
- Amazon SNS メッセージフィルタリングによる Pub/Sub メッセージングの簡素化
- S3 Select と Glacier Select
- MapReduce の Lambda リファレンスアーキテクチャ
- サーバーレスアプリケーションのリポジトリアプリ – CloudWatch Logs グループの保持期間を自動設定
- サーバーレスアーキテクトごとに知っておくべき 10 のリソース

ホワイトペーパー

- サーバーレスアーキテクチャによる企業経済の最適化

まとめ

サーバーレスアプリケーションは、差別化につながらない重労働を開発者に任せますが、それでも適用すべき重要な原則があります。

信頼性のために、障害経路を定期的にテストすることで、本稼働環境に到達する前にエラーをキャッチする可能性が高くなります。パフォーマンスのため、お客様の期待から後方から始めると、最適なエクスペリエンスを実現するように設計できます。パフォーマンスの最適化に役立つ AWS ツールも多数あります。

コスト最適化のために、トラフィック需要に応じてリソースのサイズを設定することでサーバーレスアプリケーション内の不要な無駄を削減し、アプリケーションを最適化することで価値を向上させることができます。運用の場合、アーキテクチャはイベントへの応答の自動化に努める必要があります。

最後に、安全なアプリケーションが組織の機密情報資産を保護し、すべてのレイヤーでコンプライアンス要件を満たします。

サーバーレスアプリケーションの環境は、ツールやプロセスのエコシステムが成長し、成熟していくと共に進化し続けています。これが発生すると、サーバーレスアプリケーションのアーキテクチャが Well-Architected になるように、このペーパーを継続的に更新していきます。

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。

- Adam Westrich: シニアソリューションアーキテクト、アマゾン ウェブ サービス
- Mark Bunch: エンタープライズソリューションアーキテクト、アマゾン ウェブ サービス
- Ignacio Garcia Alonso: ソリューションアーキテクト、アマゾン ウェブ サービス
- Heitor Lessa: プリンシパルサーバーレスリード Well-Architected、アマゾン ウェブ サービス
- Philip Fitzsimons、シニアマネージャー (Well-Architected)、アマゾン ウェブ サービス
- Dave Walker: プリンシパルスペシャリストソリューションアーキテクト、アマゾン ウェブ サービス
- Richard Threlkeld: シニアモバイル担当プロダクトマネージャー、アマゾン ウェブ サービス
- Julian Hambleton-Jones: シニアソリューションズアーキテクト、アマゾン ウェブ サービス

その他の資料

追加の情報については、以下を参照してください。

- AWS Well-Architected フレームワーク

改訂履歴

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
マイナーな更新 (p. 61)	HTML に数値がない状態やマイナーな編集上の変更を修正しました。	July 15, 2020
ホワイトペーパーの更新 (p. 61)	新機能とベストプラクティスの進化について、全体を更新します。	December 19, 2019
ホワイトペーパーの更新 (p. 61)	Alexa と Mobile の新しいシナリオ、およびベストプラクティスの進化を反映するために全体を更新します。	November 1, 2018
初版発行 (p. 61)	サーバーレスアプリケーションレンズが初めて公開されました。	November 1, 2017

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ており、(b) AWS の現行製品と慣行ついて説明していますが、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。