AWS Lambda を使用した
サーバーレスアーキテクチャ

概要とベストプラクティス

2017 年 11 月

注意

本書は、情報提供の目的のみのために提供されるものです。本書の発行時点における AWS の現行製品と慣行を表したものであり、それらは予告なく変更されることがあります。お客様は本書の情報および AWS 製品の使用について独自に評価する責任を負うものとします。これらの情報は、明示または黙示を問わずいかなる保証も伴うことなく、「現状のまま」提供されるものです。本書のいかなる内容も、AWS、その関係者、サプライヤー、またはライセンサーからの保証、表明、契約的責任、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。

目次

要約

2014 年の AWS re:Invent で登場して以来、AWS Lambda は最も急速に成長している AWS のサービスの 1 つとなっています。AWS Lambda の登場により、新しいアプリケーションアーキテクチャのパラダイムとして、**サーバーレス**というものが生まれました。AWS は現在、サーバーを管理する必要なく、完全なアプリケーションスタックの構築を可能にする、さまざまなサービスを提供しています。たとえば、ウェブやモバイルのバックエンド、リアルタイムのデータ処理、チャットボットや仮想アシスタント、IoT のバックエンドなどのユースケースを、すべて完全なサーバーレスにすることが可能です。サーバーレスアプリケーションのロジックレイヤーでは、AWS Lambda を使用してビジネスロジックを実行できます。開発者や組織は AWS Lambda を使用することで、従来のサーバーベースの環境にアプリケーションをデプロイするよりも開発速度が上がり、また実験を早めることが可能になっていると認識しつつあります。

このホワイトペーパーは、AWS Lambda の概要と機能、また AWS でサーバーレスアプリケーションを構築するための推奨事項やベストプラクティスを数多く提供することを目的としています。

はじめに - サーバーレスとは

サーバーレスとは、多くの場合、サーバーレスアプリケーションを指します。サーバーレスアプリケーションは、サーバーのプロビジョニングや管理を必要としないアプリケーションです。オペレーティングシステム (OS) のアクセスコントロール、OS のパッチ適用、プロビジョニング、サイジングの適正化、スケーリング、可用性などについて責任を負う代わりに、コアとなる製品やビジネスロジックに集中することができます。サーバーレスプラットフォームにアプリケーションを構築することで、プラットフォームがお客様に代わってこれらを管理します。

サービスやプラットフォームは、以下の機能を提供する場合、サーバーレスであると言えます。

- **サーバー管理不要** – サーバーのプロビジョニングや維持が不要です。インストール、維持、管理を要するソフトウェアやランタイムがありません。
- **柔軟なスケーリング** – アプリケーションを自動的に、あるいは、個々のサーバーという単位ではなく、消費単位 (スループット、メモリなど) を切り替えて容量を調整することで、スケーリングできます。
- **高可用性** – サーバーレスアプリケーションには高可用性と耐障害性が組み込まれています。これらの機能は、アプリケーションを実行するサービスによってデフォルトで提供されるため、お客様がアーキテクチャを設計する必要はありません。
- **無駄なキャパシティなし** – 使用していないキャパシティにお金を支払う必要はありません。コンピューティングやストレージなどのために、事前に、あるいは余分にキャパシティをプロビジョニングしておく必要はありません。コードが実行されていない時の料金は発生しません。

AWS クラウドは、サーバーレスアプリケーションのコンポーネントとなるさまざまなサービスを数多く提供しています。たとえば、次のような機能があります。

- コンピューティング – AWS Lambda[1]
- API – Amazon API Gateway[2]
- ストレージ – Amazon Simple Storage Service (Amazon S3)[3]
- データベース – Amazon DynamoDB[4]
- プロセス間メッセージング – Amazon Simple Notification Service (Amazon SNS)[5] および Amazon Simple Queue Service (Amazon SQS)[6]
- オーケストレーション – AWS Step Functions[7] および Amazon CloudWatch Events[8]
- 分析 – Amazon Kinesis[9]

このホワイトペーパーでは、AWS Lambda、コードが実行されるサーバーレスアプリケーションのコンピューティングレイヤー、Lambda でサーバーレスアプリケーションを構築して維持する場合のベストプラクティスを可能にする AWS 開発者用ツールおよびサービスに焦点を当てます。

AWS Lambda — 基本

Lambda は、大規模でプロビジョニング不要な、**関数**に基づいたサーバーレスコンピューティング製品です。アプリケーションにクラウドロジックレイヤーを提供します。Lambda 関数は、AWS や対応するサードパーティー製サービスで発生するさまざまなイベントによりトリガーすることができます。リアクティブで、イベント駆動型のシステム構築を可能にします。対応すべきイベントが同時に複数ある場合、Lambda は単純に関数の多数のコピーを並行で実行します。Lambda 関数は、ワークロードのサイズにあわせて正確にスケールしたり、個々のリクエストまでスケールダウンします。したがって、サーバーやコンテナがアイドル状態となる可能性が極めて低くなります。Lambda 関数を使用するアーキテクチャは、無駄な容量を削減するよう設計されています。

Lambda は、サーバーレスな「Function-as-a-Service (FaaS)」の一種だと説明できます。FaaS はイベント駆動型のコンピューティングシステムを構築する手法の 1 つです。デプロイメントと実行の単位として関数に依存します。サーバーレス FaaS は、プログラミングモデルに仮想マシンやコンテナが存在しないタイプの FaaS です。またこのタイプの FaaS では、ベンダーがプロビジョニング不要のスケーラビリティと組み込みの信頼性を提供します。
図 1 は、イベント駆動型コンピューティング、FaaS、サーバーレス FaaS の関係を示しています。

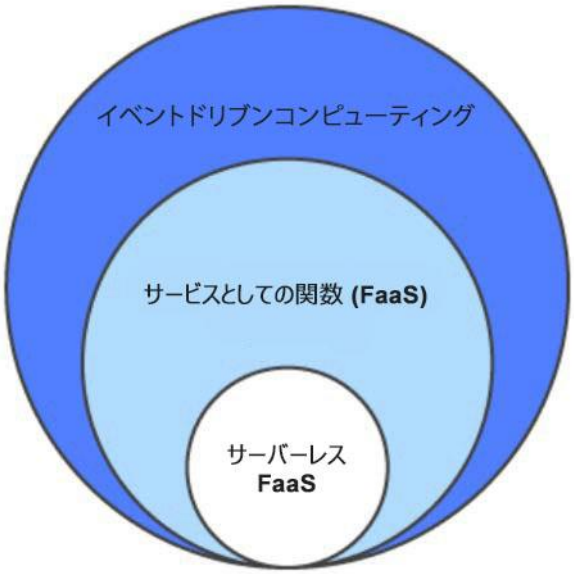

図 1:イベント駆動型コンピューティング、FaaS、サーバーレス FaaS の関係

Lambda を使用すると、事実上あらゆるタイプのアプリケーションやバックエンドサービス用のコードを実行できます。Lambda は優れた可用性でコードを実行し、スケールします。

作成した各 Lambda 関数には、実行する**コード**、コードの実行について定める**設定**、また必要な場合には、イベントの発生時にこれを検知して関数を呼び出す 1 つまたは複数の**イベントソース**が含まれます。これらの要素については、次のセクションで詳細に取り上げます。

イベントソースの例としては API Gateway があり、API Gateway で作成した API メソッドが HTTPS リクエストを受信するといつでも、Lambda 関数を呼び出すことができます。また別の例として Amazon SNS があります。Amazon SNS は、SNS トピックに新しいメッセージが投稿された時点で、いつでも Lambda 関数を呼び出すことができます。多くのイベントソースのオプションがLambda関数をトリガーできます。完全なリストについては、こちらのドキュメントを参照してください。[10] また、Lambda は RESTful サービス API も提供していますが、これには直接 Lambda 関数を呼び出す機能が含まれています。[11] この API を使用すると、別途イベントソースを設定せずに、コードを直接実行できます。
イベントソースと Lambda 関数を統合するためにコードを記述する必要はありません。また、イベントを検知してそれを関数に配信するインフラストラクチャの管理も、配信されるイベントの数に合わせた Lambda 関数のスケーリング管理も不要です。アプリケーションのロジックに集中し、ロジックを実行させるイベントソースを設定することができます。

Lambda 関数は、簡略化すると、図 2 に示すようなアーキテクチャ内で実行されます。

図 2:Lambda 関数を実行するアーキテクチャの簡略図

関数にイベントソースを設定すると、イベント発生時にコードが呼び出されます。コードでは、任意のビジネスロジックの実行、外部のウェブサービスへのアクセス、他の AWS サービスとの統合など、アプリケーションが必要とするあらゆることを行うことができます。Lambda を使用する場合、選択した言語で、お客様が使い慣れているのと同じ機能とソフトウェア設計原則が

すべて適用されます。さらに、Lambda関数とイベントソースの統合を通じてサーバーレスアプリケーションで強制される特有の疎結合化のため、Lambda関数を使ってマイクロサービスを構築するのは自然なことです。

サーバーレスの原理と Lambda の基本を理解すれば、コードを書き始める準備はできたと考えられます。すぐに Lambda を使い始めるには、以下のリソースが役立ちます。

- Hello World チュートリアル：http://docs.aws.amazon.com/lambda/latest/dg/get-started-create- function.html[12]
- Serverless workshops and walkthroughs for building sample applications: https://github.com/awslabs/aws-serverless-workshops[13]

AWS Lambda — さらに詳しく

このホワイトペーパーの残りのセクションは、Lambda のコンポーネントと機能を理解するのに役立ちます。続いては、Lambda を使用したサーバーレスアプリケーションの構築と所有に関するさまざまな側面でのベストプラクティスを示します。

関数のコード、イベントソース、関数の設定など、「はじめに」で説明した Lambda の主要コンポーネントのそれぞれについて、さらに深く掘り下げていきましょう。

Lambda 関数のコード

本質的に、Lambda はコードを実行するために使用します。このコードは、Lambda がサポートするいずれの言語 (本ホワイトペーパーの発行時点では Java、Node.js、Python、C#) でお客様が書いたものでも、また、お客様が記述したコードと一緒にアップロードしたコードやパッケージも有効です。関数コードのパッケージの一部として、ランタイム環境で実行可能な任意のライブラリ、アーティファクト、あるいはコンパイル済みのネイティブバイナリを自由に持ち

込んでかまいません。また必要に応じて、AWS Lambda のランタイム環境がサポートする言語からコードを呼び出すのであれば、別のプログラミング言語 (PHP、Go、SmallTalk、Ruby など) で書いたコードを実行することも可能です (こちらのチュートリアルを参照してください)。[14]

Lambda ランタイム環境は、Amazon Linux AMI をベースとしているため (現在の環境についての詳細はこちらを参照)、Lambda 内で実行する予定のコンポーネントはそれと一致する環境内でコンパイルし、テストする必要があります。[15] Lambda 内で実行する前にこの種のテストを行えるよう、AWS ではローカルで Lambda 関数のテストを可能にする AWS SAM CLI というツールセットを提供しています。[16] これらのツールについては、このホワイトペーパーの「サーバーレス開発のベストプラクティス」セクションで説明します。

関数コードのパッケージ

関数コードの**パッケージ**には、コードを実行する場合にローカルで必要になるアセットがすべて含まれています。パッケージには最低でも、関数が呼び出された時に Lambda サービスが実行すべきコード関数が入ります。ただし、コードの実行時に参照する他のアセット、たとえば、コードがインポートする追加のファイルとクラスとライブラリ、実行したいバイナリ、呼び出しにあたってコードが参照すべき設定ファイルなどが含まれる場合があります。関数コードパッケージの最大サイズは、本ホワイトペーパーの発行時点で圧縮した状態で 50 MB かつ展開した状態で250 MB です(AWS Lambda の制限に関する完全なリストは、こちらのドキュメント[17]を参照してください)。

Lambda 関数を作成すると (AWS マネジメントコンソールを介するか CreateFunction API を使用)、パッケージをアップロードした先の S3 バケットとオブジェクトキーを参照できます。[18] あるいは、関数の作成時に直接コードパッケージをアップロードすることもできます。Lambda はその後、このサービスが管理する S3 バケットにコードパッケージを保存します。同一の方法は、更新されたコードを既存の Lambda 関数に発行する場合 (UpdateFunctionCode API 経由)[19] にも利用できます。

イベントが発生すると、コードパッケージは S3 バケットからダウンロードされ、Lambda ランタイム環境にインストールされて、必要に応じて呼び出されます。これは、関数をトリガーするイベント数が必要とする規模で、Lambda が管理する環境の中で、オンデマンドで行われます。

ハンドラー

Lambda 関数が呼び出されると、コードの実行は**ハンドラー**と呼ばれるものから開始されます。ハンドラーとは、お客様が作成してパッケージに含めた特定のコードメソッド (Java、C#) または関数 (Node.js、Python) です。ハンドラーは、Lambda 関数の作成時に指定します。パッケージ内で関数ハンドラーを定義、参照する仕組みについては、Lambda がサポー

トする言語ごとに独自の要件があります。

各サポート対象言語の使用を開始するには、次のリンクが役立ちます。

言語	ハンドラー定義の例

Java[20]	MyOutput output handlerName(MyEvent event, Context context) {
	...
	}

Node.js[21]	exports.handlerName = function(event, context, callback) {
	...
	// callback parameter is optional
	}

Python[22]	def handler_name(event, context): ... return some_value

C#[23]	myOutput HandlerName(MyEvent event, ILambdaContext context) {
	...
	}

ハンドラーが Lambda 関数内で正常に呼び出されると、ランタイム環境はお客様の記述したコードに属します。Lambda 関数では、ハンドラーで始まるコードを記述しておくことにより、お客様が適切だと考えるあらゆるロジックを自由に実行できます。つまり、ハンドラーは、アップロードしておいたファイルとクラス内の他のメソッドと関数を呼び出すことができます。コードでは、アップロードしておいたサードパーティー製ライブラリをインポートしたり、アップロードしておいたネイティブバイナリをインストールして実行したりすることができます (Amazon Linux で実行可

能なものに限ります)。他の AWS のサービスとやり取りしたり、依存するウェブサービスへの API リクエストを行うことなども可能です。

イベントオブジェクト

Lambda 関数がサポート対象言語の 1 つで呼び出される場合に、ハンドラー関数に提供されるパラメータの 1 つが **イベントオブジェクト**です。イベントは、それを作成したイベントソースにより、構造やコンテンツが異なります。イベントパラメータのコンテンツには、Lambda 関数がロジックを実行するのに必要なすべてのデータとメタデータが含まれています。たとえば、API Gateway で作成したイベントには、API クライアントが作成した HTTPS リクエストに関連する詳細 (パス、クエリ文字列、リクエストボディなど) が含まれ、また新しいオブジェクトの作成時に Amazon S3 が作成したイベントには、バケットと新しいオブジェクトに関する詳細も含まれます。

context オブジェクト

Lambda 関数には、**context オブジェクト**も提供されます。context オブジェクトにより、関数コードは Lambda の実行環境とやり取りできるようになります。context オブジェクトのコンテンツと構造は、Lambda 関数が使用する言語ランタイムごとに異なりますが、最低でも次のものが含まれます。

- **AWS RequestId** – Lambda 関数の特定の呼び出しを追跡するために使用されます (エラー報告や AWS サポートに連絡する際に重要です)。
- **残時間** – 関数のタイムアウトが発生するまでの残りのミリ秒時間です (Lambda 関数は本ホワイトペーパーの発行時点で最大 300 秒実行可能ですが、設定によりタイムアウトを短縮できます)。
- **ロギング**– 各言語ランタイムが、Amazon CloudWatch Logs にログステートメントをストリームする機能を提供しています。context オブジェクトには、ログステートメントの送り先となる CloudWatch Logs ストリームに関する情報が含まれます。各言語のランタイムでロギングがどのように扱われるかについての詳細は、以下を参照してください。

- Java[24]
- Node.js[25]
- Python[26]
- C#[27]

AWS Lambda のコードの記述 ― ステートレスと再利用

Lambda のコードを記述する場合には、次の中央テナントを理解することが重要です。**コードは、**ステートがあると想定してはいけません。なぜなら、Lambda が新しい関数のコンテナがいつ作成され、最初に呼び出されるかを完全に管理するためです。コンテナが最初に呼び出される理由は、数多く考えられます。たとえば、Lambda 関数をトリガーするイベントの同時実行が関数用に以前作成したコンテナの数を超えるほど増加している場合や、イベントが数分ぶりに Lambda 関数をトリガーする場合などです。実際の需要に応じて Lambda が関数コンテナをスケールアップまたはスケールダウンした場合は、コードもそれに従って動作できなければなりません。Lambda はすでに実行中の特定の呼び出しプロセスを中断することはありませんが、コードが変動のレベルをを考慮する必要はありません。

つまり、コードでは、ある呼び出しから次の呼び出しまでステートが保存されるだろうと仮定することはできません。ただし、関数コンテナは作成されて呼び出されるたびに、少なくとも終了する前の数分間はアクティブかつ後続の呼び出しに使用可能なままになります。後続の呼び出しが、すでにアクティブで少なくとも先に一度は呼び出されたコンテナで発生する場合を、呼び出しが**ウォームコンテナ**で実行されていると言います。Lambda 関数の呼び出しで、初めて関数コードパッケージを作成して呼び出す必要がある場合を、呼び出しは**コールドスタート**を行っていると言います。

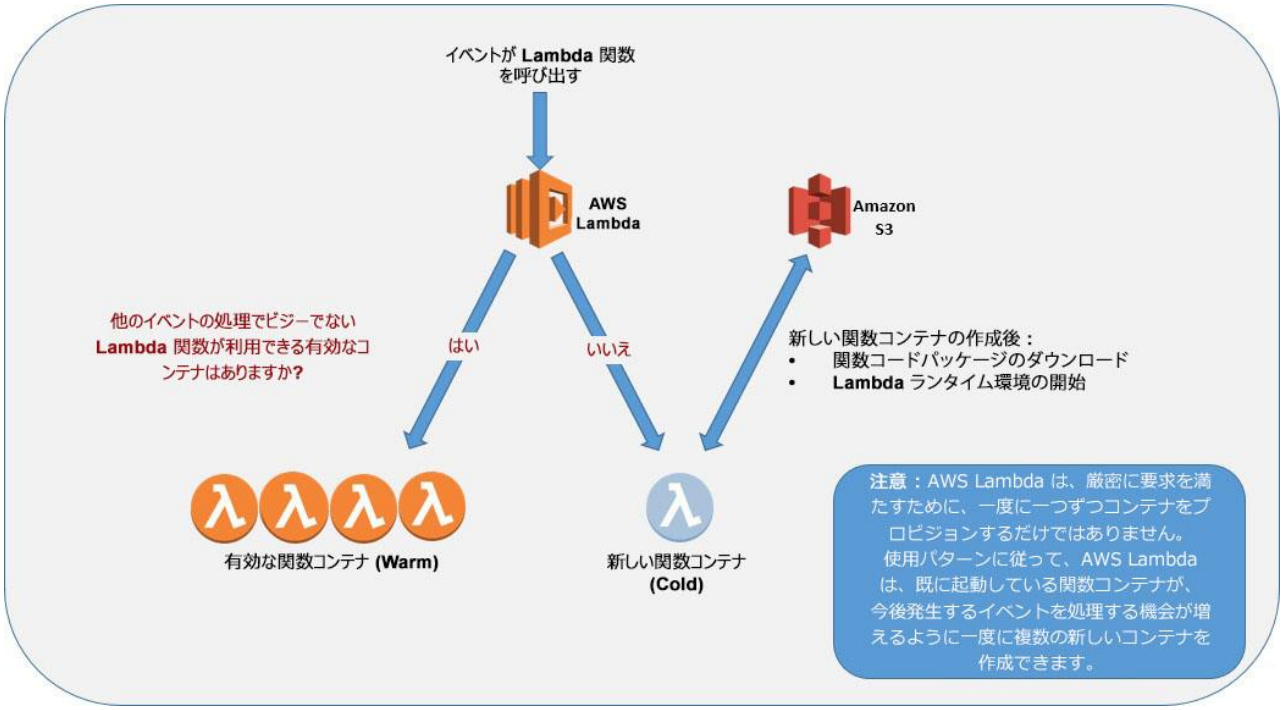

図 3:ウォーム関数コンテナとコールド関数コンテナの呼び出し

コードが実行するロジックによっては、コードがウォームコンテナを活用する仕組みを理解することで、Lambda でのコード実行速度を上げられる可能性があります。これが応答を速め、コストを下げる結果につながります。さらに詳細な情報について、また、ウォームコンテナを活用して Lambda 関数のパフォーマンスを向上させる方法の例については、このホワイトペーパーで後述する「ベストプラクティス」のセクションを参照してください。

全体として、Lambda がサポートする各言語には、それぞれにソースコードのパッケージ化のモデルとそれを最適化できる可能性があります。それぞれのサポート対象言語の使用を開始するには、こちらのページをご覧ください。[28]

Lambda 関数のイベントソース

Lambda 関数のコードに何が含まれるかを理解したところで、コードを呼び出すトリガーであ

る、**イベントソース**について見ていきましょう。Lambda は、関数を直接呼び出せる Invoke API を提供していますが、これを使用するのはテストや運用目的の場合に限られるでしょう。[2] [9] 代わりに、Lambda 関数を AWS のサービスで発生するイベントソースに関連付けることで、必要に応じて関数を呼び出すことができます。イベントソースと Lambda 関数を統合するためのソフトウェアを記述し、スケールし、維持する必要はありません。

呼び出しパターン

Lambda 関数の呼び出しには、2 つのモデルがあります。

- **プッシュモデル** – Lambda 関数は、別の AWS サービスで特定のイベントが発生するたび (たとえば新しいオブジェクトが S3 バケットに追加された場合) に呼び出されます。
- **プルモデル** – Lambdaはデータソースをポーリングして、新しいレコードが届くと関数を呼び出し、単一の関数呼び出しで新しいレコードを一緒にバッチ処理します (たとえば Amazon Kinesis や Amazon DynamoDB ストリームの新しいレコード)。

また、Lambda 関数は同期して実行することも、非同期で実行することも可能です。これは、Lambda 関数の呼び出し時に提供されるパラメータ、**InvocationType** で選択します。このパラメータには、選択可能な値が 3 つあります。

- **RequestResponse** – 同期的に実行します。
- **Event** – 非同期で実行します。
- **DryRun** – 発信者の呼び出しが許可されているかをテストしますが、関数は実行しません。

各イベントソースは、関数がどのように呼び出されるかを規定します。イベントソースは、前述の通り、自身のイベントパラメータ作成も担当します。

次の表では、より一般的なイベントソースのいくつかについて、これを Lambda 関数と統合する方法を詳細に示します。サポート対象となるイベントソースの完全なリストは、こちらで確認できます。[30]

プッシュモデルのイベントソース

Amazon S3

呼び出しモデル	プッシュ
呼び出しタイプ	Event
説明	S3 のイベント通知 (ObjectCreated や ObjectRemoved など) が発行された時に Lambda 関数を呼び出すよう設定できます。
ユースケースの例	ユーザーがアプリケーション経由で S3 バケットにアップロードしたイメージに対して、イメージ変更 (サムネイル、異なる解像度、ウォーターマークなど) を作成します。 S3 バケットにアップロードされた未加工データを処理し、ビッグデータパイプラインの一部として、変換されたデータを別の S3 バケットに移動します。

Amazon API Gateway

呼び出しモデル	プッシュ
呼び出しタイプ	Event または RequestResponse
説明	API Gateway を使用して作成した API メソッドで、Lambda 関数をそのサービスのバックエンドとして利用できます。API メソッドの統合タイプとして Lambda を選択すると、Lambda 関数が同期的に呼び出されます (La

	mbda 関数のレスポンスが API のレスポンスとして機能します)。この統合タイプを利用する場合、API Gateway は Lambda 関数へのシンプルなプロキシとしても機能します。API Gateway は、それ自体では処理も変換も実行しませんが、リクエストのコンテンツをすべて Lambda に伝えます。 イベントとして API から関数を非同期で呼び出し、ただちに空のレスポンスを返したい場合には、API Gateway を AWS サービスプロキシとして利用できます。リクエストのヘッダーで呼び出しタイプを event と規定することで、Lambda の呼び出し API と統合することが可能です。これは、API クライアントがリクエストから返される情報を必要とせず、応答時間を可能な限り最短にしたい場合に最適なオプションです(このオプションは、ウェブサイトやアプリケーションでのユーザー操作を、分析目的でサービスのバックエンドに転送する場合に最適です)。
ユースケースの例	ウェブサービスのバックエンド (ウェブアプリケーション、モバイルアプリ、マイクロサービスアーキテクチャなど)。 レガシーサービスの統合 (レガシー SOAP バックエンドを新しい最新の REST API に移行する Lambda 関数)。 その他、アプリケーションのコンポーネント間で HTTPS が適切な統合メカニズムである場合のユースケース。

Amazon SNS

呼び出しモデル	プッシュ
呼び出しタイプ	Event
説明	SNS トピックに発行されるメッセージを Lambda 関数のイベントとして配信できます。

ユースケースの例	CloudWatch アラームへの自動応答。 SNS トピックにそのまま発行可能な、他のサービス (AWS またはその他) からのイベントの処理。

AWS CloudFormation

呼び出しモデル	プッシュ
呼び出しタイプ	RequestResponse
説明	AWS CloudFormation スタックをデプロイする一環として、カスタムコマンドを実行し、作成中のスタックにデータを返すためのカスタムリソースとして Lambda 関数を指定できます。
ユースケースの例	AWS CloudFormation の機能を拡張し、AWS CloudFormation では本来まだサポート外である AWS のサービス機能を含めるようにします。 スタックの作成、更新、削除処理の主要な段階でカスタム検証やレポート作成を実行します。

Amazon CloudWatch Events

呼び出しモデル	プッシュ
呼び出しタイプ	Event
説明	AWS のサービスの多くは、CloudWatch Events にリソースの状態変更を発行します。その後で、自動応答のため、これらのイベントにフィルタリングを行い、Lambda 関数にルーティングすることができます。

ユースケースの例	イベント駆動型オペレーションのオートメーション (たとえば、新しい EC2 インスタンスが起動するたびにアクションを実行し、AWS Trusted Advisor が新たにステータスの変更を報告すると適切なメーリングリストに通知するなど)。 事前に cron で完了したタスクの置換 (予定されているイベントを CloudWatch Events がサポート)。

Amazon Alexa

呼び出しモデル	プッシュ
呼び出しタイプ	RequestResponse
説明	Amazon Alexa Skills 用にサービスのバックエンドとして機能する Lambda 関数を記述できます。Alexa ユーザーがお客様のスキルとやり取りを行うと、Alexa の自然言語理解と自然言語処理の機能がそのやり取りを Lambda 関数に配信します。
ユースケースの例	お客様独自の Alexa スキル。

プルモデルのイベントソース

Amazon DynamoDB

呼び出しモデル	プル
呼び出しタイプ	RequestResponse
説明	Lambda は DynamoDB ストリームを 1 秒あたり複数回ポーリングし、前回のバッチ後にストリームに発行された更新バッチで Lambda 関数を呼び出します。それぞれの呼び出しのバッチサイズを設定できます。

ユースケースの例	DynamoDB テーブルに変更が発生するとトリガーされるべき、アプリケーション中心のワークフロー (たとえば新規ユーザー登録、注文の発生、友達リクエストの承認など)。 DynamoDB テーブルの別のリージョンへのレプリケーション (災害復旧のため) や、別のサービスへのレプリケーション (バックアップや分析のために S3 バケットにログとして発送)。

Amazon Kinesis Streams

呼び出しモデル	プル
呼び出しタイプ	RequestResponse
説明	Lambda は各ストリームのシャードごとに 1 秒あたり 1 回、Kinesis ストリームをポーリングして、シャードの次のレコードで Lambda 関数を呼び出します。関数に一度に配送されるレコードの数や、同時に実行する Lambda 関数コンテナの数に対して、バッチサイズを定義できます (ストリームのシャードの数 = 同時実行する関数コンテナの数)。
ユースケースの例	ビッグデータパイプラインのリアルタイムデータ処理。 ログステートメントのストリーミングや他のアプリケーションイベントに対するリアルタイムのアラート通知/モニタリング。

Lambda 関数の設定

Lambda 関数のコードを記述してパッケージ化した後、どのイベントソースで関数をトリガーするかを選択する他にも、Lambda でコードが実行される仕組みを定めるためのさまざまな設定オプションがあります。

関数のメモリ

実行する Lambda 関数に割り当てるリソースを定義するために、関数のリソースであるメモリ、つまり RAM を増減する単一のダイヤルが提供されています。Lambda 関数には 128 MB から最大 1.5 GB までの RAM を割り当てることができます。このダイヤルでは、関数が実行中に使用可能なメモリの最大容量を指示するだけではなく、その同じダイヤルで、関数が使用できる CPU とネットワークリソースにも影響を与えることができます。

適切なメモリの割り当てを選択することは、Lambda 関数の価格とパフォーマンスを最適化する時には非常に重要な手順です。パフォーマンスの最適化に関するさらに具体的な情報は、このホワイトペーパーで後述のベストプラクティスを参照してください。

バージョンとエイリアス

以前デプロイされていたコードを参照したり、Lambda 関数を前の状態に戻したりする必要が生じる場合もあります。Lambda では AWS Lambda 関数に**バージョン**付けをすることができます。あらゆる Lambda 関数にはもれなく、デフォルトバージョンである$LATEST が組み込まれています。この $LATEST バージョンにより、Lambda 関数にアップロードされている最新のコードを扱うことができます。現在 $LATEST が参照しているコードのスナップショットを取得し、PublishVersion API を介して、番号付きのバージョンを作成できます。[31] また、UpdateFunctionCode API を介して関数コードを更新する場合には、オプションの Boolean パラメータ、publish があります。[32] リクエストに publish: true と設定すると、Lambda は最後に発行されたバージョンから値を増加させて、新しい Lambda 関数のバージョンを作成します。

Lambda 関数の各バージョンは、いつでも個別に呼び出すことができます。各バージョンには独自の Amazon リソースネーム (ARN) があり、次のように参照されます。

```
arn:aws:lambda:[region]:[account]:function:[fn_name]:[version]
```

呼び出し API を呼び出す場合、または、Lambda 関数のイベントソースを作成する場合、Lambda 関数の特定のバージョンを指定して実行することもできます。[33] バージョン番号を示さない場合、またはバージョン番号を含まない ARN を使用する場合には、デフォルトで $LATEST が呼び出されます。

Lambda 関数コンテナは関数の特定のバージョンに固有であると知っておくことが重要です。つまり、たとえば関数のバージョン 5 用にすでにいくつかの関数コンテナがデプロイ済みであり、Lambda ランタイム環境で使用可能である場合、同じ関数のバージョン 6 は既存のバージョン 5 コンテナでは実行できません。関数のバージョンごとに異なるコンテナのセットがインストールされて、管理されることになります。

バージョン番号による Lambda 関数の呼び出しは、テストや運用のアクティビティで便利です。ただし、実際のアプリケーショントラフィックにおいて、特定のバージョン番号で Lambda 関数をトリガーすることは推奨しません。そうしてしまうと、コードを更新しようとするたび、新しい関数バージョンを指定するために Lambda 関数を呼び出すトリガーとクライアントのすべてを更新する必要が生じます。代わりに、ここでは Lambda **エイリアス**を使用すべきです。関数エイリアスを使用すると、特定の Lambda 関数バージョンを呼び出してイベントソースを示すことができます。

ただし、エイリアスがどのバージョンを参照するかは、いつでも更新することが可能です。たとえば、live エイリアスでバージョン番号 5 を呼び出すイベントソースやクライアントでは、liveというエイリアスがバージョン番号 6 を示すよう更新されると、関数のバージョン番号 6 に移行できます。各エイリアスは、関数のバージョン番号を参照する場合と同様に、ARN 内で参照できます。

```
arn:aws:lambda:[region]:[account]:function:[fn_name]:[alias]
```

注意 :エイリアスは、特定のバージョン番号を示す単なるポインターです。つまり、同一の Lambda 関数バージョンを示す異なるエイリアスが同時に複数ある場合、各エイリアスへのリクエストは同一のインストール済み関数コンテナのセットで実行されます。各エイリアスに対するリクエストを個別に処理したい場合には、誤って複数のエイリアスを同じ関数バージョンで指定しないよう、この点を理解しておくことが重要です。

推奨する Lambda エイリアスとその使い方の例を以下に示します。

- **live/prod/active** – 本稼働でトリガーされる、またはクライアントが一体化された Lambda 関数バージョンを表すことができます。
- **blue/green** – エイリアスの使用を介した Blue/Green デプロイパターンを可能にします。
- **debug** – アプリケーションをデバッグするためにテストスタックを作成していた場合、さらに綿密な分析をする必要があれば、このようなエイリアスと統合できます。

関数エイリアスの使用に関する戦略を確実に文書化することで、精巧でサーバーレスなデプロイメントと運用についてのプラクティスを手に入れることができます。

IAM ロール

AWS Identity and Access Management (IAM) が提供する機能では、AWS のサービスと API とのやり取りに関するアクセス許可を定義する IAM ポリシーを作成できます。[34] ポリシーは **IAM ロール**と関連付けることが可能です。特定のロール用に生成されたアクセスキー ID とシークレットアクセスキーは、そのロールにアタッチされたポリシーにおいて定義されたアクションの実行を許可されます。IAM のベストプラクティスに関する詳細は、こちらのドキュメントを参照してください。[35]

Lambda の context では、IAM ロール (**実行ロール**と呼ぶ) を各 Lambda 関数に割り

当てます。ロールにアタッチされた IAM ポリシーは、関数コードがどの AWS のサービスの API とのやり取りを認められるのかを定義します。これには 2 つの利点があります。

- ソースコードは、AWS API とやり取りするために、AWS の認証情報の管理やローテーションを必要としません。単純に AWS SDK とデフォルトの認証情報プロバイダーを使用することで、Lambda 関数では自動的に、関数に割り当てられた実行ロールと関連付けられている一次的な認証情報が使用されます。
- ソースコードは自身のセキュリティ体制から分離されます。関数がアクセスできないサービスと統合するように、発者が Lambda 関数コードを変更しようとする場合、関数に割り当てられた IAM ロールによって、この統合は失敗します(ソースコード内に AWS の認証情報が存在していないことを確認するためには、実行ロールから切り離された IAM の認証情報を使用していない限り、静的コード解析ツールを使用する必要があります)。

各 Lambda 関数には、特定の最小権限の IAM ロールを個別に割り当てることが重要です。この戦略により、各 Lambda 関数は他の Lambda 関数の認可範囲を広げることなく、個別に進化することが保証されます。

Lambda 関数のアクセス許可

アクセス許可と呼ばれる概念を介して、プッシュモデルのどのイベントソースで Lambda 関数を呼び出すことができるかを定義できます。アクセス許可を使用すると、関数の呼び出しが許可された AWS Resource Names (ARN) をリストする**関数ポリシー**を定義できます。

プルモデルのイベントソース (たとえば Kinesis ストリームや DynamoDB ストリーム) では、Lambda 関数に割り当てられた IAM 実行ロールによって適切なアクションが許可されていることを確認する必要があります。必要なアクセス許可の管理を自身で行いたくない場合、AWS では、プルベースの各イベントソースに関連付けられたマネージド型 IAM ロールのセット

を使用できます。ただし、最小権限の IAM ポリシーを保証するため、目的のイベントソースのみにアクセスを許可する、リソースを特定したポリシーで、お客様自身の IAM ロールを作成する必要があります。

ネットワークの設定

Lambda 関数は、AWS Lambda サービス API の一部である**呼び出し** API の使用を介して実行されます。つまり、管理が必要な、関数に対する直接のインバウンドでのネットワークアクセスはありません。ただし、関数コードは外部の依存関係 (内部でまたはパブリックにホストされたウェブサービス、AWS サービス、データベースなど) と統合する必要があります。Lambda 関数では、アウトバウンドネットワーク接続に、大きく分けて 2 つのオプションがあります。

- **デフォルト** – Lambda 関数は、Lambda が管理する仮想プライベートクラウド (VPC) の内部から接続します。インターネット接続は可能ですが、個別の VPC で実行される、プライベートにデプロイされたリソースには接続できません。
- **VPC** – Lambda 関数は、個別のアカウントで選択する VPC とサブネット内にプロビジョニングされた **Elastic Network Interface (ENI)** を介して接続します。これらの ENI にはセキュリティグループを割り当てることが可能で、EC2 インスタンスが同一のサブネットに配置されている場合とまったく同じように、その ENI が配置されているサブネットのルートテーブルに基づいて、トラフィックがルーティングされることになります。

Lambda 関数がプライベートにデプロイされたリソースとの接続を必要としない場合には、デフォルトのネットワークオプションを選択することをお勧めします。VPC オプションを選択する場合、次のことを管理する必要があります。

- 高可用性を目的として、複数のアベイラビリティーゾーンが確実に使用されている適切なサブネットを選択すること。

- 容量を管理するため、各サブネットに適切な数の IP アドレスを割り当てること。

- Lambda 関数が必要な接続とセキュリティを備えることを可能にする、VPC ネットワーク設計を実装すること。
- Lambda 関数の呼び出しパターンで、新しい ENI がジャストインタイムで作成される必要がある場合には、Lambdaのコールドスタート時間の増加(現在、ENI の作成には、かなり時間がかかる場合があります)。

ただし、ユースケースでプライベート接続を要する場合には、Lambda で VPC オプションを使用してください。Lambda 関数を個別の VPC でデプロイする場合の綿密なガイダンスについては、こちらのドキュメントを参照してください。[36]

環境変数

ソフトウェア開発ライフサイクル (SDLC) のベストプラクティスでは、開発者はコードと設定を分離するよう求められます。Lambda で環境変数を使用すると、これが実現できます。Lambda 関数の環境変数では、コードに変更を加えることなく、関数コードとライブラリに動的にデータを渡すことが可能です。環境変数は、お客様が関数設定の一部として作成し変更する、キーと値のペアです。デフォルトでは、これらの変数は保管時に暗号化されます。機密情報を Lambda 関数の環境変数として保存する場合には、関数の作成前に AWS Key Management Service (AWS KMS) を使用して値を暗号化し、環境変数として暗号化テキストを保管することをお勧めします。その後 Lambda 関数に、関数の実行時にメモリ内でその変数を復号させます。

使用する環境変数をどのように決めるべきか、以下に例を示します。

- ログ設定 (FATAL、ERROR、INFO、DEBUG など)
- 依存関係やデータベース接続の文字列と認証情報

- 関数のフラグとトグル

Lambda 関数の各バージョンは、環境変数に独自の値を持つことができます。ただし、番号を付された Lambda 関数バージョンで値が確立した後は、その値を変更することはできません。Lambda 関数の環境変数を変更するには、$LATEST バージョンで変更を加え、その新しい環境変数の値を含む新しいバージョンを発行します。これにより、関数の以前のバージョンにどの環境変数の値が関連付けられているのかを、常に記録しておくことがきます。多くの場合、ロールバック手順の間、またはアプリケーションの過去の状態をトリアージする時にこの記録が重要になります。

デッドレターキュー

サーバーレスの世界であっても、やはり例外は発生する可能性があります(たとえば、Lambda のイベントでは正常に解析できない新しい関数コードをアップロードしてしまったり、あるいは、関数が呼び出されるのを妨げる AWS 内の操作イベントが存在したりするかもしれません)。非同期のイベントソース (**InvocationType** のイベント) では、AWS は関数の呼び出しを担当するクライアントソフトウェアを所有します。AWS には、呼び出しの発生時において、それが正常に行われたか否かを同期的に通知する機能はありません。これらのモデルで関数を呼び出そうとする場合に例外が発生すると、あと 2 回、呼び出しが試されます (再試行の間隔はバックオフです)。関数に設定されていた場合には、3 回目の試行の後で、そのイベントを破棄するかまたは**デッドレターキュー**に置くか、いずれかの処理が行われます。

デッドレターキューは、失敗した呼び出しイベントすべての送信先としてお客様が指定しておいた、SNS トピックか SQS キューのどちらかです。失敗イベントが発生する場合、デッドレターキューを使用することで、イベント中に処理が失敗したメッセージだけを保持できます。関数を再度呼び出せるようになった後、デッドレターキューにある失敗したイベントを再処理の対象にできます。デッドレターキューに置かれた関数の呼び出し試行について、再処理または再試行するメカニズムは、お客様自身で決めることになります。デッドレターキューに関する詳細は、こちらのチュートリアルを参照してください。[37] アプリケーションにとって、最終的に Lambda 関数の呼び

出しのすべてを完了させることが重要である場合には、それにより実行が遅延するとしても、デッドレターキューを使用する必要があります。

タイムアウト

タイムアウトが返されるまでの、単一の関数の実行を完了するために認められる時間の最大値を指定できます。Lambda 関数のタイムアウトの最大値は、本ホワイトペーパー発行時点では 300 秒であり、Lambda 関数の単一の呼び出しが 300 秒より長く実行されることはありません。Lambda 関数のタイムアウトに、常に最大値を設定する必要はありません。アプリケーションが速やかに失敗すべき場合も多くあります。Lambda 関数は 100 ミリ秒単位の実行時間に基づいて料金が発生するため、関数について時間のかかるタイムアウトを避けることで、関数がタイムアウトを待っている時間分の請求を防止できます (外部の依存関係が利用できなかったり、誤って無限ループをプログラムしてしまったり、他の類似の事態が生じたりする場合があります)。

また、Lambda 関数の実行が完了するか、タイムアウトが発生してレスポンスが返されると、すべての実行が止まります。これには、Lambda 関数が実行中に大量に引き起こしたバックグラウンド処理、サブプロセス、非同期処理などが含まれている場合があります。そのため、重要なアクティビティをバックグラウンドや非同期での処理に依存すべきではありません。タイムアウトや関数からのレスポンスが返される前に、コードはそれらのアクティビティを確実に完了させる必要があります。

サーバーレスのベストプラクティス

Lambda ベースのサーバーレスアプリケーションに含まれるコンポーネントを確認したので、次に推奨されるベストプラクティスについて説明します。SDLC とサーバーベースのアーキテクチャには、サーバーレスアーキテクチャにも当てはまるベストプラクティスが数多くあります。単一障害点を排除する、デプロイ前に変更をテストする、機密情報を暗号化するなどです。

ただし、サーバーレスアーキテクチャにとってのベストプラクティスは、運用モデルの違いによって異なるタスクになる可能性があります。オペレーティングシステムや、インフラストラクチャ内の下位

レベルのコンポーネントにアクセスすることはないため、気にかける必要はありません。このため、独自のアプリケーションコードやアーキテクチャ、自身が従う開発プロセス、アプリケーションが活用することでベストプラクティスに従うことが可能になる AWS のサービスの機能だけに集中することになります。

まず、AWS Well-Architected フレームワークに沿って、サーバーレスアーキテクチャを設計するための一連のベストプラクティスを確認します。その後で、サーバーレスアプリケーションを構築する時に開発プロセスに役立つベストプラクティスと推奨事項をいくつか説明します。

サーバーレスアーキテクチャのベストプラクティス

AWS Well-Architected フレームワークには、ベストプラクティスとワークロードを比較し、安定した効率的なシステムを作成するためのガイダンスを得るのに役立つ戦略が含まれているため、お客様は機能的な要件に集中することができます。[38] このフレームワークは、セキュリティ、信頼性、パフォーマンス効率、コストの最適化、運用上の卓越性という 5 つの柱をベースにしています。このフレームワークのガイドラインの多くが、サーバーレスアプリケーションに適合します。ただし、サーバーレスアーキテクチャに特有の実装手順や実装パターンがあります。次のセクションでは、Well-Architected の柱それぞれについて、サーバーレス特有の一連の推奨事項を説明します。

セキュリティのベストプラクティス

アプリケーションのセキュリティの設計と実装は常に第 1 の優先事項とする必要があり、サーバーレスアーキテクチャにおいてもこの点は変わりません。サーバーレスアプリケーションの保護については、サーバーがホストするアプリケーションと比較して明らかに大きく異なる点があります。サーバーレスアプリケーションに、保護すべきサーバーはありません。ただし、アプリケーションのセキュリティについては考える必要があります。依然として、サーバーレスセキュリティのための責任共有モデルもあります。

Lambda とサーバーレスアーキテクチャを使用すると、アンチウィルス/マルウェアソフトウェア、ファイル統合モニタリング、侵入検知/防止システム、ファイアウォールなどによりアプリケーションのセキュリティを実装するよりも確実に、セキュリティのベストプラクティスを実現できます。これを可能にするのが、安全なアプリケーションコードの記述、ソースコードの変更における厳しいアクセスコントロール、Lambda 関数が統合する各サービスについて次に挙げる AWS のセキュリティのベストプラクティスです。

多くのサーバーレスのユースケースに適合するサーバーレスセキュリティのベストプラクティスのショートリストを以下に示します。ただし、独自のセキュリティ要件とコンプライアンス要件はよく理解しておく必要があり、ここで述べる以上の要件を含む可能性もあります。

- **関数ごとに 1 つの IAM ロール**

  AWS アカウントの各 Lambda 関数はすべて、IAM ロールと 1 対 1 の関係でなければなりません。複数の関数が完全に同じポリシーで開始するとしても、関数が将来的に最小権限のポリシーを保証できるようにするため、常に IAM ロールは分離します。

  たとえば、AWS KMS キーへのアクセスを要する関数の IAM ロールを複数の Lambda 関数で共有したとすると、その後、共有した関数のすべてが同一の暗号化キーへのアクセス権を持ってしまいます。

- **一次的な AWS 認証情報**

  Lambda 関数のコードや設定に、永続的な AWS 認証情報を保持してはいけません(永続的な認証情報がコードベースに 1 つも存在しないことを保証するには、静的コード解析ツールの使用が最適です)。ほとんどの場合、他の AWS のサービスと統合する必要があるのは、IAM 実行ロールのみです。認証情報を提供することなく、AWS SDK を介して簡単にコード内に AWS のサービスのクライアントを作成し

ます。SDK は、ロールに対して生成された一次的な認証情報の取得とローテーションを自動的に管理します。以下に Java を使用する例を示します。

```
AmazonDynamoDB client = AmazonDynamoDBClientBuilder.defaultClient(); Table
myTable = new Table(client, "MyTable");
```

AWS SDK for Java が DynamoDB テーブルとやり取りして DynamoDB API へのリクエストに自動的に署名するオブジェクトを作成する場合、必要となるのはこのコードスニペットのみで、ここでは関数に割り当てられた一次的な IAM 認証情報を利用しています。[39]

ただし、お客様の関数が必要とするアクセスには、その実行ロールでは不十分という場合もあります。これは、お客様の Lambda 関数が一部のクロスアカウント統合を行う際にあり得る事態です。また、Amazon Cognito[40] のアイデンティティロールと DynamoDB の詳細なアクセスコントロール[41] 設定の組み合わせによる、ユーザー固有のアクセスコントロールポリシーを持っている場合も同様です。クロスアカウントのユースケースでは、AWS Security Token Service 内で AssumeRole API へのアクセス権を与えて統合されるべき実行ロールに、一時的なアクセス認証情報を取得する権利を付与する必要があります。[42]

ユーザー固有のアクセスコントロールポリシーの場合、関数に当該ユーザーのアイデンティティが提供されたうえで、Amazon Cognito API GetCredentialsForIdentity と統合される必要があります。[43] このケースでは、Lambda 関数の呼び出しに関連付けられた各ユーザーについて正しい認証情報を利用できるよう、コードによる認証情報の適切な管理を確実に行うことが不可欠です。ユーザーごとの認証情報は、アプリケーションが暗号化し、ユーザーのセッションデータの一部として DynamoDB やAmazon ElastiCache のような場所に保管するのが一般的です。そうすることで、リピーターに後続リクエストを送って認証情報を再生成するよりも、レイテンシーを低減しながらスケーラビリティを広げることができます。[44]

- **シークレットの永続化**

  Lambda 関数で使用する必要がある永続的なシークレット (たとえば、データベース認証情報、依存関係にあるサービスへのアクセスキー、暗号化キーなど) を所有する場合があります。アプリケーションにおいてシークレットのライフサイクルを管理するための、いくつかのオプションをお勧めします。

  - Lambda Environment Variables with Encryption Helpers[45]

    利点 – 関数のランタイム環境に直接提供されるため、レイテンシーおよびシークレットの取得に要するコードが最小限になります。

    欠点 – 環境変数が関数バージョンに結合されます。環境変数を更新する時に、新しい関数バージョンを求められます (堅牢になりますが、同時に安定版の履歴も残ります)。

  - Amazon EC2 Systems Manager パラメータストア[46]

    利点 – Lambda 関数から完全に分離され、シークレットと関数とが相互に関連する仕組みについて最大限の柔軟性が提供されます。

    欠点 – パラメータストアに対するリクエストに、パラメータ/シークレットの取得が求められます。大幅ではないものの、環境変数および追加のサービスの依存関係におけるレイテンシーが増加します。また、わずかにコードを書き加える必要があります。

- **シークレットの使用**

  シークレットが存在するのは常にメモリ内に限られる必要があり、ディスクに記録されたり書き込まれたりしてはなりません。シークレットのローテーションを管理するコードは、アプリケーションの実行中にシークレットを呼び出す必要のあるイベントの中に記述します。

- **API 認証**

API Gateway を Lambda 関数のイベントソースとして使用する点が、API クライアントの認証と許可に関する所有権がお客様にある、他の AWS のサービスにおけるイベントソースのオプションとは異なります。API Gateway は、ネイティブな AWS 署名バージョン 4 認証情報、[47] generated クライアント SDK の生成、[48] また カスタムオーソライザー[49] などを提供することで、多くの手間のかかる作業を行います。ただし、API のセキュリティ体制が設定した基準を確実に満たすようにする責任は、引き続きお客様にあります。API セキュリティのベストプラクティスに関する詳細は、こちらのドキュメントを参照してください。[50]

- **VPC セキュリティ**

  Lambda 関数が VPC 内にデプロイされたリソースへのアクセスを必要とする場合、ネットワークセキュリティのベストプラクティスを使用する必要があります。これには最小権限のセキュリティグループ、Lambda 関数固有のサブネット、ネットワーク ACL、お客様の Lambda 関数からのトラフィックのみが目的の送信先に達するようにするルートテーブルを使用します。

これらのプラクティスとポリシーは、Lambda 関数と依存関係との接続方法に影響を与えることに留意します。Lambda 関数の呼び出しは、引き続きイベントソースと呼び出し API (どちらも VPC 設定の影響を受けない) を介して発生します。

- **アクセスコントロールのデプロイメント**

  UpdateFunctionCode API の呼び出しは、コードのデプロイメントに類似しています。UpdateAlias API を介して新しく発行されたバージョンにエイリアスを移すことは、コードのリリースに類似しています。関数コード/エイリアスを作動させる Lambda API へのアクセスには、最新の注意を払う必要があります。したがって、人的エラーの可能性を取り除くため、こういった何らかの関数用の API に対しては (少なくとも本稼働用関数では)、直接的なユーザーアクセスを排除しなければなりません。Lambda 関数のコード変更は、オートメーションによって行う必要があります。これを念頭に置

くと、Lambda へのデプロイメントに対するエントリポイントは、継続的インテグレーション/継続的デリバリー (CI/CD) のパイプラインが開始される場所になります。これはリポジトリ内のリリースブランチであるか、AWS CodePipeline のパイプラインをトリガーする新しいコードパッケージがアップロードされた S3 バケットであるか、または独自の組織や処理に固有の別の場所であるかもしれません。[51] どこであっても、そこでは新たにチーム構造と役割に合わせた厳しいアクセスコントロールのメカニズムを強化する必要が生じます。

信頼性のベストプラクティス

サーバーレスアプリケーションは、ミッションクリティカルなユースケースをサポートするように構築することができます。ミッションクリティカルなアプリケーションを使用するのと同じように、Amazon.com の CTO、Werner Vogels が提唱する「すべてが常に失敗する」マインドセットに基づいて設計することが重要です。サーバーレスアプリケーションにとって、このマインドセットの意味するところは、コードに論理的なバグが入り込んだり、アプリケーションの依存関係が動かなかったりすることかもしれません。また、防止に努め、サーバーレスアプリケーションにも引き続き適用できる既存のベストプラクティスを使用すべき、その他の似たようなアプリケーションレベルの問題かもしれません。インフラストラクチャレベルのサービスのイベントでは、サーバーレスアプリケーションのイベントからは離れますが、高可用性と耐障害性を達成するため、アプリケーションをどのように設計してきたか理解する必要があります。

高可用性

高可用性は本稼働アプリケーションにとって重要です。Lambda 関数の可用性の状態は、その中での実行が可能なアベイラビリティーゾーンの数に依存します。関数がデフォルトのネットワーク環境を使用する場合、関数は AWS リージョン内のアベイラビリティーゾーンのすべてで自動的に実行可能です。デフォルトのネットワーク環境で関数に高可用性を設定するためには、他には何も必要ありません。関数がお客様所有の VPC 内にデプロイされている場合、サブネット (および関連するアベイラビリティーゾーン) は、アベイラビリティーゾーンが停止した場合に関数が使用可能かどうかを定義します。したがって、VPC の設計には複数のアベイラビリ

ティーゾーンのサブネットが含まれることが重要です。アベイラビリティーゾーンの停止が発生した場合、同時実行の関数が必要とする数をサポートするため、残りのサブネットが適切な IP アドレスを保持し続けることが重要です。関数が必要とする IP アドレスの数を計算する仕組みについては、こちらのドキュメントを参照してください。[52]

耐障害性

アプリケーションに必要な可用性を得るために、複数の AWS リージョンを活用する必要がある場合は、設計の前に考慮しなければなりません。Lambda 関数コードのパッケージを複数の AWS リージョンにレプリケートすることは、複雑な操作ではありません。複雑になり得る点は、ほとんどのマルチリージョンのアプリケーション設計と同様、アプリケーションスタックのすべての層にわたるフェイルオーバーの決定を調整することです。つまり、Lambda 関数だけではなく、イベントソース (およびイベントソースのさらに上流の依存関係) ならびに永続的なレイヤーについて、別の AWS リージョンへの移動を理解して調整する必要があるということです。結局、マルチリージョンのアーキテクチャは非常にアプリケーション固有であると言えます。実行可能なマルチリージョン設計を行うために最も大切なことは、設計に優先して耐障害性を考慮することです。

復旧

サーバーレスアプリケーションは関数が実行できない場合にどのように動作すべきかを考えます。API Gateway がイベントソースとして使用されるユースケースでは、復旧はエラーメッセージを適切に扱ったり、もしパフォーマンスの低下があれば、関数が再び正常に実行されるまでの間、実行可能なユーザーエクスペリエンスを提供したりするのと同じように簡単です。

非同期でのユースケースでは、停止期間中の関数の呼び出しが失敗とならないように引き続き保証することが非常に重要となる可能性があります。受信したイベントのすべてが関数の回復後に確実に処理されるようにするには、デッドレターキューを活用し、キューに置かれたイベントを復旧後に処理する方法を実装する必要があります。

パフォーマンス効率のベストプラクティス

パフォーマンスのベストプラクティスを深く掘り下げる前に、ユースケースが非同期で達成可能な場合には、関数のパフォーマンスを (コスト最適化以外は) 気にかける必要はないことに留意してください。イベントの **InvocationType** またはプルベースの呼び出しモデルを使用するイベントソースの 1 つを活用できます。それらのメソッドだけで、Lambda が個別にイベントを処理し続ける間に、アプリケーションのロジックをさらに進めることができるかもしれません。最適化するものが Lambda 関数の実行時間であれば、Lambda 関数の実行の間隔に大きく影響を与えるものが 3 つあります。最適化が簡単なものから順に、関数設定に割り当てるリソース、選択する言語ランタイム、記述するコードです。

最適なメモリサイズの選択

関数で使用可能なコンピューティングリソースの総量、つまり関数に割り当てられた RAM の合計容量を増減させるため、Lambda は単一のダイヤルを提供します。割り当てられた RAM の容量もまた、関数が受け取る CPU 時間とネットワーク帯域幅の総量に影響します。単純に、関数を十分な速度で実行できる最小量のリソースを選択するのでは、不適切です。Lambda は 100 ミリ秒単位で料金が発生するため、この戦略ではアプリケーションのレイテンシーが増加する可能性があるだけでなく、増加したレイテンシーがリソースのコスト削減を上回る場合には、結局コストが高額になる可能性もあります。

アプリケーションにとって最適な価格とパフォーマンスのレベルを決定するには、Lambda 関数を使用可能な各リソースレベルでテストするようお勧めします。リソースレベルが高まるに連れ、関数のパフォーマンスを対数的に向上させる必要があることがわかります。実行するロジックは、関数の実行時間に対する下限を定義します。関数で使用可能な追加の RAM/CPU/帯域幅がもはや大幅なパフォーマンスの向上をもたらさないという、リソースのしきい値もあります。ただし、料金は Lambda 内のリソースレベルが増加するのに比例して上昇します。関数にとっての最適な設定を選択するには、テストで対数関数の曲がりを見つける必要があります。

以下のグラフは、例示の関数に対して理想的なメモリを割り当てることで、いかにコストの適正化とレイテンシーの低減の両方を実現できるのかを示しています。ここでは、低めのメモリ選択を超える 512 MB の使用により増加する 100 ms あたりのコンピューティングコストに比べ、リソースの割り当てを増やしたことにより関数で削減されたレイテンシーの合計が勝っています。しかし、512 MB より上を見ると、今回の特定の関数のロジックではパフォーマンスの向上が小さくなりますから、100 ms あたりの追加コストが合計コストを上げることになります。上記から、合計コストを最小化する最適な選択は 512 MB であるとわかります。

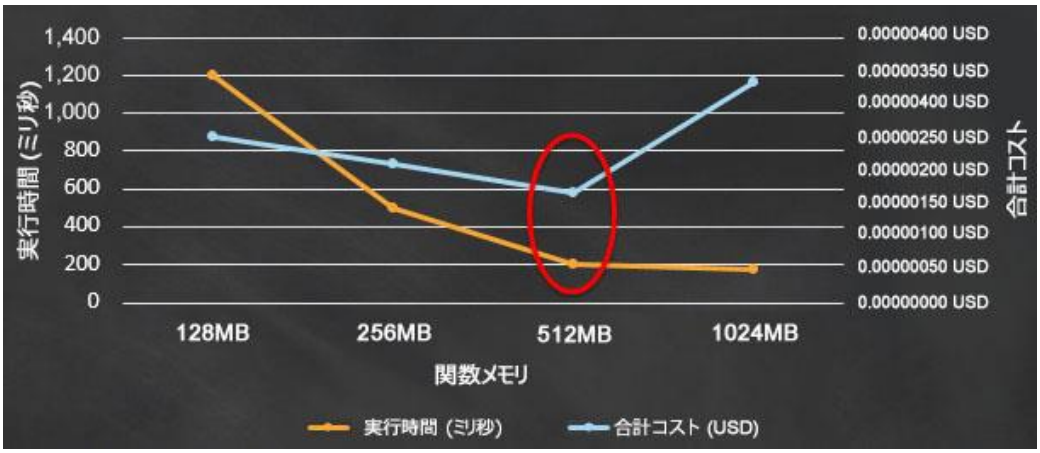

図 4:最適な Lambda 関数のメモリサイズの選択

関数に対するメモリ使用は呼び出しごとに決定され、CloudWatch Logs で閲覧できます。[53] 以下に示すように、それぞれの呼び出しで、REPORT:エントリが作成されます。

```
REPORT RequestId:3604209a-e9a3-11e6-939a-754dd98c7be3 Duration:
12.34 ms Billed Duration:100 ms Memory Size:128 MB Max Memory Used:18 MB
```

Max Memory Used: フィールドを分析することにより、関数がより多くのメモリを必要としているか、または関数のメモリサイズを過剰にプロビジョニングしていないか、調べることができます。

言語ランタイムのパフォーマンス

言語ランタイムのパフォーマンスの選択は、明らかに使用感と、サポートされている各ランタイムのスキルに依存します。しかし、アプリケーションのパフォーマンスについて熱心に考慮している場合、Lambda で期待すべきものは、他のランタイム環境にある場合と同様に、各言語のパフォーマンス特性です。コンパイラ言語 (Java と .NET) はコンテナの最初の呼び出し時に最大の初期起動コストを生じさせますが、後続の呼び出しではベストパフォーマンスを示します。インタプリタ言語 (Node.js と Python) ではコンパイラ言語と比べて最初の呼び出しにかかる時間が非常に短くなりますが、最大パフォーマンスレベルはコンパイラ言語に及びません。

アプリケーションのユースケースがレイテンシーに敏感で、かつ最初の呼び出しコストが頻繁に発生することの影響を受けやすい (トラフィックに非常にスパイクが多い、または使用頻度が非常に高い) 場合は、いずれかのインタプリタ言語をお勧めします。

アプリケーションのトラフィックパターンに大きなピークや谷の部分がない、またはアプリケーションに Lambda 関数の応答時間が障害となるユーザーエクスペリエンスがない場合は、すでになじみのある言語を選択することをお勧めします。

コードの最適化

Lambda 関数におけるパフォーマンスの大部分は、Lambda 関数を実行するのに必要なロジックとその依存関係により決まります。可能な最適化はアプリケーションによって異なるため、そのすべてを取り上げることはしません。ただし、Lambda のコードを最適化するための一般的なベストプラクティスがいくつかあります。これらはコンテナの再利用を活用すること (概要で前述のとおり)、また、コールドスタートにおける初期コストを最小化することと関連します。

ウォームコンテナが呼び出された時に関数コードのパフォーマンスを高める方法について、以下に例を示します。

- 最初の実行の後、コードがローカルで取得する外部化した設定または依存関係を、保存して参照します。
- すべての呼び出しで変数やオブジェクトの再初期化を制限します (グローバル変数や静的変数、シングルトンなどを使用します)。
- キープアライブを有効にして、前の呼び出しで確立された接続 (HTTP、データベースなど) を再利用します。

最後に、Lambda 関数でコールドスタートにかかる総時間を制限するため、以下を行う必要があります。

1. プライベート IP を介して VPC 内でリソースに接続する必要がない限り、常にデフォルトのネットワーク環境を使用します。これは、Lambda 関数の VPC 設定に関連して (VPC 内での ENI の作成に関連して)、コールドスタートに追加のシナリオがあるためです。
2. コンパイラ言語ではなくインタプリタ言語を選択します。
3. 関数コードのパッケージを削って、ランタイムに必須のものだけにします。それによって、コードパッケージの呼び出し前に Amazon S3 からのダウンロードにかかる総時間数が減少します。

アプリケーションのパフォーマンスの理解

アプリケーションのアーキテクチャには 1 つまたは複数の Lambda 関数が含まれる場合がありますが、アーキテクチャのさまざまなコンポーネントを可視化するには、AWS X-Ray を使用することをお勧めします。[54] X-Ray を使用すると、以下の図に示すように、各コンポーネントのレイテンシーやその他のメトリクスが単独で示され、コンポーネントの各部を経由するアプリケーションのリクエストについて完全なライフサイクルを追跡できます。

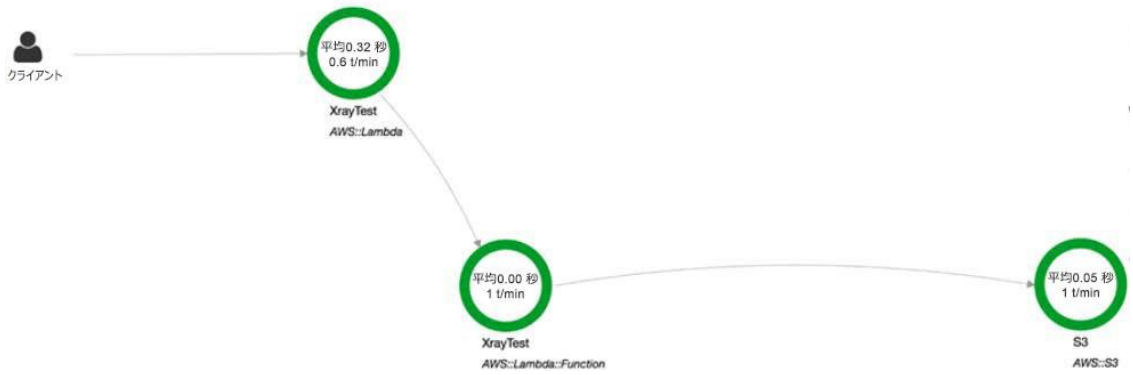

図 5:AWS X-Ray で視覚化したサービスマップ

X-Ray の詳細については、こちらのドキュメントを参照してください。[55]

運用上の卓越性のベストプラクティス

サーバーレスアプリケーションを作成すると、従来のアプリケーションにはつきものである多くの運用上の負担が取り除かれます。これは、運用上の優秀性に対して関心を減らすべきだという意味ではありません。責任の範囲を減らして運用上注意する事項を絞り込み、うまくいけば運用上の優秀性についてレベルを高めることが可能だということです。

ログ記録

Lambda の各言語ランタイムは、関数が記録されたステートメントを CloudWatch Logs に配信するメカニズムを提供します。ログを適切に使用することは当然であり、Lambda やサーバーレスアーキテクチャにとって新しいことではありません。今日ベストプラクティスとは考えられていないとしても、多くの運用チームが、アプリケーションがデプロイされたサーバーの上に生成されたログを見ることに頼っています。しかし、Lambda ではログを見ることは不可能です。なぜならサーバーがないからです。今実際に実行中の Lambda 関数のコードを「ステップスルー」する機能もありません (ただし、デプロイする前にAWS SAM Local を使用してステップスルーすることは可能です)。[56] デプロイ済みの関数では、関数の動作についての調査を通知するために、お客様が作成するログに大きく依存します。したがって、作成するログが、問題発生時にその問題の追跡/トリアージに役立つように、ログの作成に過大なコンピューティング時間を必要としない、冗長性の適切なバランスを見つけることが特に重要です。

ランタイム中に作成するログステートメントを決定できるよう、関数が参照可能な LogLevel 変数の作成には、Lambda 環境変数を使用するようお勧めします。適切なログレベルを使用することで、追加のコンピューティングコストとストレージコストを運用上のトリアージ中に限って選択的に発生させることができるようになります。

メトリクスとモニタリング

Lambda では、他の AWS のサービスと同じように、多くの CloudWatch メトリクスが用意されています。これには、関数が受信した呼び出し数や関数の実行の間隔などに関連するメ

トリクスが含まれます。CloudWatch を介して提供されるメトリクスのすべてで、各Lambda 関数にアラームのしきい値 (上下) を作成することが、ベストプラクティスになります。関数が呼び出される方法、または実行に要する時間に大きな変化があれば、それがアーキテクチャにおける問題の最初の兆候であるかもしれません。

アプリケーションが集める必要のある追加のメトリクス (たとえば、アプリケーションのエラーコード、依存関係特有のレイテンシーなど) については、CloudWatch または指定のモニタリングソリューションに保管されたそれらのカスタムメトリクスを入手する方法として 2 つのオプションがあります。

- カスタムメトリクスを作成し、Lambda 関数が実行時に必要とする API と直接統合します。依存関係は最少であり、メトリクスを可能な限り速く記録します。ただし、Lambda の実行時間および他のサービスの依存関係と統合するためのリソースを費やす必要があります。この方法をとる場合は、メトリクスを取得するためのコードが確実にモジュール化されており、かつ、特定の Lambda 関数と密結合されるのではなく、複数の Lambda 関数で再利用可能であるようにします。

- Lambda で提供されたログ記録のメカニズムを使用し、Lambda 関数のコードにメトリクスを取得して記録します。その後、メトリクスを抽出して CloudWatch で使用可能にするため、関数ストリームに CloudWatch Logs のメトリクスフィルタを作成します。あるいは、フィルタ後のログステートメントを別のメトリクスソリューションにプッシュするため、CloudWatch Logs のストリームに別の Lambda 関数をサブスクリプションフィルタとして作成することもできます。このパスでは、さらに複雑さが増し、メトリクスを取得するための以前のソリューションほどリアルタイムに近くはありません。ただし、関数はログ記録を介することで、外部サービスへのリクエストを作成する場合より迅速にメトリクスを作成できます。

デプロイメント

Lambda にデプロイメントを行うことは、新しい関数コードのパッケージをアップロードしたり、新しいバージョンを発行したり、エイリアスを更新したりするのと同じ程度に簡単です。ただし、この手順は Lambda でのデプロイメントプロセスの一部である必要があります。各デプロイメントプロセスは、アプリケーション固有です。お客様のユーザーやアプリケーションの操作を否定的に中断させるのを避けるようにデプロイメントプロセスを設計するには、各 Lambda 関数やそのイベントソースと依存関係との関係を理解する必要があります。考慮すべき事項を以下に示します。

- **Parallel バージョンの呼び出し** – Lambda 関数の新しいバージョンを指定するエイリアスの更新は、サービス側で非同期的に行われます。以前のソースコードのパッケージを含む既存の関数コンテナが、エイリアスが更新された新しい関数のバージョンと一緒に引き続き呼び出される期間が少しあります。このプロセスでは、アプリケーションが期待されるとおりに動作し続けることが重要です。このアーティファクトでは、デプロイメントの後に廃棄されているスタックの依存関係 (たとえば、データベーステーブル、メッセージキューなど) は、新しい関数のバージョンを対象とするすべての呼び出しを観察し終えるまでは廃棄されない可能性があります。
- **デプロイのスケジュール** – トラフィックのピーク時に Lambda 関数をデプロイすると、想定以上にコールドスタートの回数が増える結果になる可能性があります。Lambda 環境でプロビジョニングされている新しい、またはコールド状態の関数コンテナから受ける直接的な影響を最小化するには、関数は必ずトラフィックの少ない期間にデプロイする必要があります。
- **ロールバック** – Lambda は Lambda 関数のバージョンに関する詳細 (たとえば、作成時刻、インクリメント番号など) を提供します。ただし、アプリケーションのライフサイクルがそれらのバージョンをどのように使用してきたかについては論理的には追跡しません。Lambda 関数のコードをロールバックする必要がある場合、この詳細情報は以前デプロイされていた関数バージョンにロールバックするプロセスにおいて重要な事項です。

負荷テスト

最適なタイムアウト値を決定するため、Lambda 関数の負荷テストを実行します。依存関係のサービスに伴う問題は関数の同時実行を想定以上に増加させる可能性がありますが、この問題を適切に解決できるよう、関数の実行にかかる時間を分析することが重要です。これは、Lambda 関数が Lambda のスケーリングを扱う可能性がないリソースに対してネットワークの呼び出しを行う場合、特に重要な事項です。

トリアージとデバッグ

ログを記録して調査を可能にすることと、X-Ray を使用してアプリケーションをプロファイルすることは、どちらも運用上のトリアージを行うのに役立ちます。さらに、統合テスト、パフォーマンステスト、デバッグなど、運用上のアクティビティを表す Lambda 関数のエイリアスを作成することを考えます。チームが運用上の目的に役立つテストスイートやセグメント化されたアプリケーションのスタックを構築するのが一般的です。これらの運用上のアーティファクトは、エイリアス経由で Lambda 関数と統合するためにも構築する必要があります。ただし、エイリアスは必ずしも完全に分離した Lambda 関数のコンテナを求めるものではないことに留意してください。そのため、関数のバージョン番号 N を指す PerfTest のようなエイリアスは、バージョン N を指しているすべての他のエイリアスと同一の関数コンテナを使用します。分離したコンテナが必要な場合に確実に呼び出されるようにするには、適切なバージョニングとエイリアスの更新プロセスを定義する必要があります。

コスト最適化のベストプラクティス

Lambda の料金請求は関数の実行時間と割り当てられたリソースに基づいているため、コストを最適化するには、この 2 つのディメンションの最適化を中心に行います。

規模の適正化

「パフォーマンス効率」で扱ったように、リソースサイズを最小にすることで合計コストが最小になると考えるのは適切ではありません。関数のリソースサイズが小さ過ぎると、多くのリソースが使

用可能でより速く関数を完了できる場合と比べて実行時間が長くなり、コストが増加する可能性があります。

詳細は、「最適なメモリサイズの選択」セクションを参照してください。

分散した非同期のアーキテクチャ

一連のブロッキング/同期的 API のリクエストとレスポンスによって、すべてのユースケースを実行する必要はありません。アプリケーションを非同期で設計できる場合、分離されたアーキテクチャの各コンポーネントは、密結合されたコンポーネントよりも作業にかかるコンピューティング時間が少なくなります。密結合されたコンポーネントでは、同期的なリクエストに対するレスポンスを待って CPU サイクルを費やします。Lambda のイベントソースの多くは分散システムに適合しており、モジュール式の分離された関数を、よりコスト効率に優れた方法で統合するために使うことができます。

バッチサイズ

Lambda イベントソースには、各関数の呼び出しで配信されるレコード数のバッチサイズを定義できるものがあります (たとえば Kinesis や DynamoDB)。関数がどれほど迅速にタスクを完了できるかに合わせて各イベントソースのポーリング頻度が調整されるように、各バッチサイズの最適なレコード数を見つけるテストを行う必要があります。

イベントソースの選択

Lambda と統合可能なイベントソースの幅広さは、要件に応じてさまざまなソリューションのオプションが利用可能であることを意味します。ユースケースと要件 (リクエストの規模、データ量、求められるレイテンシーなど) 次第で、Lambda 関数を取り巻くコンポーネントとしてどの AWS のサービスを選択するかに基づいたアーキテクチャの合計コストには、些細とは言えない差があるかもしれません。

サーバーレス開発のベストプラクティス

Lambda でアプリケーションを作成すると、今までに経験したことのない開発ペースが実現します。実用的で堅牢なサーバーレスアプリケーションのために記述を要するコードの量は、サーバーベースのモデルで記述を要するコード量の何割かで済む可能性があります。しかし、サーバーレスのアーキテクチャが実現する新しいアプリケーションの配信モデルには、開発プロセスが意思決定を下さなければならない新たなディメンションと構成要素が存在します。Lambda 関数を念頭にコードベースを体系化すること、開発者のラップトップからサーバーレスの本稼働環境にコード変更を移行すること、Lambda のランタイム環境や AWS 外のイベントソースをシミュレートできなくてもテストによりコードの品質を保証することなどです。サーバーレスアプリケーションを所有することに関するこれらの側面への対処に役立てるため、以下に開発中心のベストプラクティスを示します。

コードとしてのインフラストラクチャ – AWS サーバーレスアプリケーションモデル (AWS SAM)

インフラストラクチャをコードとして表すと、インフラストラクチャの作成と変更の管理における監査性、オートマタビリティ、再現性の面で多くの利点が生じます。サーバーレスアプリケーションを構築する際にはインフラストラクチャを管理する必要がないとしても、アーキテクチャの中で役割を担うコンポーネントは多くあります。IAM ロール、Lambda 関数とその設定、そのイベントソース、およびその他の依存関係です。これらすべてを AWS CloudFormation 内で表すには、本来、大量の JSON や YAML が必要でした。どのサーバーレスアプリケーションをとっても、その多くがほぼ同一のものです。

AWS サーバーレスアプリケーションモデル (AWS SAM) では、サーバーレスのアプリケーションの構築がより簡単に行えるようになり、コードとしてのインフラストラクチャの恩恵を受けることができます。AWS SAM は、AWS CloudFormation のオープン仕様の抽象化レイヤーです。[57] わずか数行の JSON または YAML で完全なサーバーレスアプリケーションスタックを

定義し、Lambda 関数のコードをインフラストラクチャの定義と一緒にパッケージ化し、さらに AWS にデプロイすることを可能にする、一連のコマンドラインユーティリティを提供します。サーバーレスアプリケーションの環境を定義して変更するには、AWS SAM を AWS CloudFormation と組み合わせて使用することをお勧めします。

ただし、インフラストラクチャレベルまたは環境レベルで発生する変更と既存の Lambda 関数内で発生するアプリケーションコードの変更との間には、差異があります。AWS CloudFormation と AWS SAM は、Lambda 関数のコードの変更用にデプロイメントパイプラインを構築するためだけに必要なツールではありません。Lambda 関数のコード変更の管理に関する推奨事項の詳細は、このホワイトペーパーの「CI/CD」セクションを参照してください。

ローカルでのテスト – AWS SAM Local

AWS へのデプロイ前にサーバーレス関数とアプリケーションをローカルでテストするため、AWS SAM Local は AWS SAM と連動して、AWS SAM に追加で増やすことができるコマンドラインツールを提供します。[58] AWS SAM Local は Docker を使用して、一般的なイベントソース (たとえば、Amazon S3、DynamoDB など) で開発したLambda 関数のテストを迅速に行えるようにします。API Gateway に作成される前に、SAM テンプレートに定義する API をローカルでテストすることができます。また、作成した AWS SAM テンプレートを検証することもできます。まだ開発者用ワークステーションにある Lambda 関数に対してこれらの機能を実行できるようにすることで、ログをローカルで閲覧したり、デバッガーでコードをステップスルーしたり、AWS に新しいコードのパッケージをデプロイする必要なしで迅速に変更を繰り返したりすることができます。

コーディングとコード管理のベストプラクティス

Lambda 関数のコードを開発する場合、多くの Lambda 関数を管理することが複雑なタスクにならないようにするため、コードの記述と整理の両方をどのように行うべきかについて、特有の推奨事項があります。

コーディングのベストプラクティス

構築に使用する Lambda のランタイム言語により、引き続き、すでにその言語で確立されているベストプラクティスに従います。コードが呼び出される方法を取り巻く環境は Lambda で変わってきましたが、言語のランタイム環境はあらゆる他のものと同じです。コーディング標準とベストプラクティスは引き続き適合します。選択する言語におけるこういった一般的なベストプラクティスとは別に、以下に Lambda のコード記述に特有の推奨事項を示します。

ハンドラーの外でのビジネスロジック

Lambda 関数は、コードパッケージ内で定義する ハンドラー関数から実行が開始されます。ハンドラー関数では、Lambda が提供するパラメータを受け取り、そのパラメータを他の関数に引き渡して、アプリケーションに対する文脈に当てはまる新しい変数やオブジェクトに解析し、その後でハンドラー関数とファイルの外にあるビジネスロジックに至る必要があります。これにより、Lambda ランタイム環境から可能な限り分離されたコードパッケージの作成が可能になります。これには、作成したオブジェクトと関数のコンテキスト内でコードをテストし、Lambda 外の他の環境で記述したビジネスロジックを再利用することができるようになるという、大きな利点があります。

以下の例 (Java で記述) では、アプリケーションの核となるビジネスロジックが Lambda と密に結びつけられている、好ましくない手法を示します。この例では、ビジネスロジックはハンドラーメソッドで作成され、Lambda イベントソースのオブジェクトに直接的に依存しています。

```
//Poor example of best practices, business logic is tightly coupled to Lambda environment
public class LambdaFunctionHandler {

    public String handleRequest(S3Event event, Context context) {
        context.getLogger().log("Starting execution.");

        AmazonDynamoDB client = AmazonDynamoDBClientBuilder.defaultClient();
        Table myTable = new Table(client, "MyTable");

        /**
         * The simple business logic below (marking items processed in DDB)
         * is directly dependent on the structure of the Lambda events
         * that S3 sends.  Performing unit tests of core business logic
         * will require me to mock an S3Event object and the Lambda service
         * created Context object in order to invoke this method.
         */
        for (S3EventNotificationRecord e : event.getRecords()) {
            context.getLogger().log("Processing records.");
            PrimaryKey key = new PrimaryKey("S3ObjectKey", e.getS3().getObject().getKey());
            Item item = new Item().withPrimaryKey(key).withBoolean("Processed", true);
            myTable.putItem(item);
        }

        context.getLogger().log("Completing execution");
        return "{\"status\": \"succes\"}";

    }

}
```

ウォームコンテナ — Caching/Keep-Alive/Reuse

先に説明したとおり、コードはウォーム状態の関数コンテナを活用するように記述する必要があります。これは、変数と変数コンテナについて、可能な場合には後続の呼び出しで再利用できるように変数の範囲を定めるという意味です。これは特に、設定のブートストラップ、外部の依存関係との接続保持、または 1 回の呼び出しから次回まで存続可能な大きなオブジェクトの 1 回限りの初期化などに対して影響力があります。

依存関係の制御

Lambda の実行環境には、Node.js と Python のランタイム用の AWS SDK など、多くのライブラリが含まれます(完全なリストは、Lambda 実行環境と利用できるライブラリを参照してください[59])。最新の一連の機能とセキュリティの更新を保証するため、Lambda は定期的にこれらのライブラリを更新します。これらの更新は Lambda 関数の動作にわずかな変化をもたらす可能性があります。関数が使用する依存関係を完全に管理するには、依存関係のすべてをデプロイパッケージとともにパッケージ化することをお勧めします。

依存関係の削減

Lambda 関数コードのパッケージは最大で圧縮時 50 MB、ランタイム環境に展開時は 250 MB まで認められています。関数コードに大きな依存関係のアーティファクトを含めている場合、含まれる依存関係をランタイムに必須のものだけに削減することが必要かもしれません。それにより、Lambda 関数のコードがダウンロードされ、ランタイム環境にインストールされる速度が、コールドスタートでも向上することになります。

速やかな失敗

あらゆる外部の依存関係には、合理的に短いタイムアウトを設定します。Lambda 関数全般のタイムアウトも、合理的に短く設定します。依存関係の応答を待つ間、関数が何もできずに空回りすることがないようにします。Lambda は関数実行の所要時間に応じて料金が発生するため、関数の依存関係が応答しない場合に必要以上に高額な課金が生じるのは望ましくありません。

例外の処理

Lambda のユースケースに応じて、異なる例外のスローと処理を決定することができます。API Gateway の API を Lambda 関数の前に置いている場合、例外がコンテンツに応じて適切な HTTP のステータスコードと発生したエラーに対するメッセージに変換された API Gateway に対し、例外のスローを戻すよう定めることができます。非同期のデータ処理システムを構築している場合、コードベース内の例外は再処理のためのデッドレターキューに動かす呼び出しと同じ扱いとする一方で、その他のエラーは記録するだけでデッドレターキューに置かないことができると定めることができます。どのような動作を失敗とするかを評価し、その動作が実現するように、コード内で正しい例外タイプが作成されスローされるようにする必要があります。例外の処理についてさらに学習するには、各言語ランタイム環境で例外が定義される仕組みに関して、以下に示す詳細を確認してください。

- Java[60]
- Node.js[61]
- Python[62]
- C#[63]

コード管理のベストプラクティス

ここまでで、Lambda 関数用に記述したコードがベストプラクティスに従うようにできましたが、では、そのコードの管理はどのように行うべきでしょうか。Lambda が実現した開発スピードにより、典型的なプロセスでは考えられないペースでコード変更を完了できます。サーバーレスアーキテクチャが必要とするコードの量が削減されるということは、Lambda 関数のコードがアプリケーションスタック機能の大部分を表しているということです。したがって Lambda 関数のコードについて優れたソースコード管理を行うと、セキュアで効率的でスムーズな変更管理プロセスの実現に役立ちます。

コードリポジトリ組織

Lambda 関数のソースコードは、選択するソースコード管理ソリューションの中で、詳細に設定されているよう整理することをお勧めします。これは一般的に、Lambda 関数とコードリポジトリまたはリポジトリプロジェクトとが 1 対 1 の関係をもつという意味です(レキシコンはソースコード管理ツールごとに異なります)。ただし、同一の論理関数のさまざまなライフサイクルの段階に対し別々の Lambda 関数を作成する (つまり 2 つの Lambda 関数があり、一方を MyLambdaFunction-DEV、もう一方を MyLambdaFunction-PROD と呼ぶ) という戦略に従っている場合、これらの分離した Lambda 関数がコードベースを共有することは理にかなっています (分離されたリリースブランチからデプロイしている場合があります)。

このようにコードを整理する主な目的は、特定の Lambda 関数のコードパッケージに寄与するコードのすべてが、個別にバージョニングされてコミットするように、また自身の依存関係とその依存関係のバージョンを確実に定義するようにすることです。各 Lambda 関数は、デプロイされた時の状態とちょうど同じように、他の Lambda 関数から見て、ソースコードから完全に分離されている必要があります。モノリシックで密結合されたコードベースと一緒に残すためだけに、アプリケーションのアーキテクチャを Lambda で近代化してモジュラー化し、分離するプロセスを経るのは望ましくありません。

リリースブランチ

リポジトリまたはプロジェクトの分岐戦略を作成し、Lambda 関数のデプロイメントをリリースブランチで増加するコミットと関連付けられるようにすることをお勧めします。リポジトリ内のソースコード変更やデプロイされた変更と実際の Lambda 関数とを、確信を持って関連付ける方法がない場合、運用上の調査は常に、コードベースのどのバージョンが現在デプロイされているものか識別しようとすることから開始されます。CI/CD のパイプラインを構築し、Lambda コードパッケージの作成とデプロイの時間と、その Lambda 関数のリリースブランチで発生したコードの変更との関連付けを可能にする必要があります (この点についての他の推奨事項は後述します)。

テスト

サーバーレスアーキテクチャにおける品質を保証するためには、コードの徹底的なテストを開発するのに時間を費やすことが最良の方法です。ただし、サーバーレスアーキテクチャでは、適切な単体テストの手法が、これまで行っていたよりも多くなるかもしれません。コードにその依存関係もテストさせるテストを記述するために、単体テストツールとフレームワークを使用する開発者が多くいます。これは単体テストと統合テストを結合する単一のテストですが、どちらもそれほどうまく働きません。

アップストリームからの入力とダウンストリームからの出力のすべてをモックして、すべての単体テストケースの範囲を単一の論理関数内の単一のコードパスに制限することが重要です。それにより、テストケースを自身のコードのみに限定することができます。単体テストを記述する場合、依存関係についてはコードが備える API や ライブラリなどとの契約に基づいて適切に動作すると想定でき、また想定する必要があります。

統合テストがコードとその依存関係との統合を実際の環境の模擬環境においてテストすることは、同じように重要です。開発者のラップトップまたはビルドサーバーがダウンストリームの依存関係と統合可能かテストしても、コードが実際の環境で正常に統合するかどうかを完全にテストすることはできません。このことは、Lambda 環境で特に当てはまります。Lambda環境では、コードはイベントソースから配信される予定のイベントの所有権を持たず、Lambda ランタイム環境は Lambda の外に作成することができません。

単体テスト

先に述べてきたことに留意しつつ、主にハンドラー関数の外のビジネスロジックに焦点を当てて、Lambda 関数のコードに徹底的な単体テストを行うことをお勧めします。関数のイベントソース用のサンプル/モックオブジェクトを解析する機能に単体テストを行うことも必要です。ただし、ロジックとテストの大部分は、コードベース内でお客様が完全なコントロールを維持する、モックされたオブジェクトや関数と一緒に存在する必要があります。単体テストを行う必要があ

るハンドラー関数の中に重要なものがあると思われる場合、これはハンドラー関数のロジックをさらにカプセル化し外面化することが必要だという兆候かもしれません。さらに、記述した単体テストを補完するため、ローカルで関数コードにエンドツーエンドのテストを提供できる AWS SAM Local を使用してローカルテストのオートメーションを作成する必要があります (単体テストの置換にはならない点に留意してください)。

統合テスト

統合テストでは、CI/CD パイプラインがトリガーして結果を検査できるサンプルイベントを介して、コードパッケージがデプロイされて呼び出される、ライフサイクルの短い Lambda 関数のバージョンを作成するようお勧めします(実装はアプリケーションとアーキテクチャに依存します)。

継続的デリバリー

CI/CD パイプラインを介したサーバーレスのデプロイメントはすべて、プログラム的に管理することをお勧めします。これは、Lambda では新しい機能を開発してコードの変更をプッシュできるスピードが速いため、より頻繁にデプロイできるようになるためです。手動デプロイでは、より頻繁なデプロイの必要性と合わさって、しばしば結果として手動プロセスがボトルネックになり、エラーにつながりやすくなります。

AWS CodeCommit、AWS CodePipeline、AWS CodeBuild、AWS SAM、AWS CodeStar は、全体論的で自動化されたサーバーレスの CI/CD パイプラインとネイティブに結びつけることができる一連の機能を提供します (パイプライン自体にも管理が必要なインフラストラクチャはありません)。

これらの各サービスが、明確に定義された継続的デリバリー戦略の中で役割を果たす仕組みを示します。

AWS CodeCommit – サーバーレスのソースコードをホストすること、推奨事項 (きめ細

かなアクセスコントロールを含む) に合わせたブランチ戦略を作成すること、新しいコミットがリリースブランチで発生した場合に新しいパイプラインの実行をトリガーするために AWS CodePipeline と統合することを実現する、ホストされたプライベート Git リポジトリを提供します。

AWS CodePipeline – パイプラインのステップを定義します。基本的に、AWS CodePipeline のパイプラインは、ソースコードの変更が届く場合に開始します。その後、構築フェーズを実行し、新しいビルドに対してテストを行い、実際の環境にビルドをデプロイしてリリースします。AWS CodePipeline は、これらの各段階で他のAWS のサービスと一緒に、ネイティブな統合オプションを提供します。

AWS CodeBuild – パイプラインのビルド状態で使用可能です。コードの構築、単体テストの実行、新しい Lambda コードパッケージの作成に使用します。その後、AWS SAM と統合し、Amazon S3 にコードパッケージをプッシュし、また、AWS CloudFormation 経由で Lambda に新しいパッケージをプッシュします。

AWS CodeBuild を介して Lambda 関数に新しいバージョンが発行された後、デプロイメント中心の Lambda 関数を作成することにより、AWS CodePipeline のパイプラインで後続の手順を自動化できます。統合テストの実行、関数エイリアスの更新、即時のロールバックが必要かどうかの判断、アプリケーションのデプロイ中に発生する必要があるその他のアプリケーション中心のステップ (キャッシュのフラッシュ、通知メッセージなど) に対するロジックがあります。これらのデプロイメント中心の Lambda 関数はそれぞれ、呼び出しアクションを使用して AWS CodePipeline のパイプラインの中で一連のステップにより呼び出すことが可能です。AWS CodePipeline 内での Lambda の使用に関する詳細は、こちらのドキュメントを参照してください。[64]

最後に、各アプリケーションと組織には、ソースコードをリポジトリから本稼働に移動することに関する独自の要件があります。このプロセスにオートメーションを取り入れるほど、Lambda を

使用して俊敏性を高めていくことができます。

AWS CodeStar – サーバーレスのアプリケーション (および他のタイプのアプリケーション) を作成するための統合ユーザーインターフェースで、最初からそのベストプラクティスに従うことを可能にします。AWS CodeStar で新しいプロジェクトを作成する場合、完全に実装されて統合された継続的デリバリーのツールチェーンが自動的に開始されます (前述の AWS CodeCommit、AWS CodePipeline、AWS CodeBuild サービスを使用)。チームメンバーの管理、問題の追跡、開発、デプロイメント、運用など、プロジェクトにかかる SDLC の全要素を管理できる場所もあります。AWS CodeStar についての詳細は、こちらを参照してください。[65]

サーバーレスアーキテクチャの例

サーバーレスアーキテクチャや AWS アカウントでそれらを再構築する手順には、多くの例があります。それらの例については、GitHubを参照してください。[66]

まとめ

AWS でサーバーレスアプリケーションを構築すると、サーバーがもたらす責任や制約から解放されます。サーバーレスのロジックレイヤーとして AWS Lambda を使用すると、迅速な構築が可能になり、独自のアプリケーションを差別化するものを開発する作業に集中できます。Lambda と合わせて、AWS は付加的なサーバーレス機能を提供しているため、堅牢かつ高性能で、イベント駆動型で信頼性があり、セキュアで、費用効果の高いアプリケーションを構築できます。このホワイトペーパーで説明した機能や推奨項目を理解すると、独自のサーバーレスアプリケーション構築で確実に成功するのに役立ちます。関連トピックの詳細については、サーバーレスコンピューティングとアプリケーションを参照してください。[67]

寄稿者

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。

- Andrew Baird (AWS シニアソリューションアーキテクト)
- George Huang (AWS シニアプロダクトマーケティングマネージャー)
- Chris Munns (AWS シニアデベロッパーアドボケイト)
- Orr Weinstein (AWS シニアプロダクトマネージャー)

参考リンク

[1] https://aws.amazon.com/lambda/

[2] https://aws.amazon.com/api-gateway/

[3] https://aws.amazon.com/s3/

[4] https://aws.amazon.com/dynamodb/

[5] https://aws.amazon.com/sns/

[6] https://aws.amazon.com/sqs/

[7] https://aws.amazon.com/step-functions/

[8] https://docs.aws.amazon.com/AmazonCloudWatch/latest/events/WhatIsCloud WatchEvents.html

[9] https://aws.amazon.com/kinesis/

[10] http://docs.aws.amazon.com/lambda/latest/dg/invoking-lambda- function.html

[11] http://docs.aws.amazon.com/lambda/latest/dg/API_Invoke.html

[12] http://docs.aws.amazon.com/lambda/latest/dg/get-started-create- function.html

[13] https://github.com/awslabs/aws-serverless-workshops

[14] https://aws.amazon.com/blogs/compute/scripting-languages-for-aws-lambda-running-php-ruby-and-go/

[15] http://docs.aws.amazon.com/lambda/latest/dg/current-supported- versions.html

[16] https://github.com/awslabs/aws-sam-local

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

[18] http://docs.aws.amazon.com/lambda/latest/dg/API_CreateFunction.html

[19] http://docs.aws.amazon.com/lambda/latest/dg/API_UpdateFunctionCode.html

[20] http://docs.aws.amazon.com/lambda/latest/dg/java-programming- model.html

[21] http://docs.aws.amazon.com/lambda/latest/dg/programming-model.html

[22] http://docs.aws.amazon.com/lambda/latest/dg/python-programming-model.html

[23] http://docs.aws.amazon.com/lambda/latest/dg/dotnet-programming-model.html

[24] http://docs.aws.amazon.com/lambda/latest/dg/java-logging.html

[25] http://docs.aws.amazon.com/lambda/latest/dg/nodejs-prog-model- logging.html

[26] http://docs.aws.amazon.com/lambda/latest/dg/python-logging.html

[27] http://docs.aws.amazon.com/lambda/latest/dg/dotnet-logging.html

[28] http://docs.aws.amazon.com/lambda/latest/dg/programming-model-v2.html

[29] http://docs.aws.amazon.com/lambda/latest/dg/API_Invoke.html

[30] http://docs.aws.amazon.com/lambda/latest/dg/invoking-lambda- func

tion.html

[31] http://docs.aws.amazon.com/lambda/latest/dg/API_PublishVersion.html

[32] http://docs.aws.amazon.com/lambda/latest/dg/API_UpdateFunctionCode.html

[33] http://docs.aws.amazon.com/lambda/latest/dg/API_Invoke.html

[34] http://docs.aws.amazon.com/IAM/latest/UserGuide/access_policies.html

[35] http://docs.aws.amazon.com/IAM/latest/UserGuide/best-practices.html

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

[37] https://aws.amazon.com/blogs/compute/robust-serverless-application - design-with-aws-lambda-dlq/

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

[39] https://aws.amazon.com/sdk-for-java/

[40] https://aws.amazon.com/cognito/

[41] http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/speci fying-conditions.html

[42] http://docs.aws.amazon.com/cognitoidentity/latest/APIReference/API_GetC redentialsForIdentity.html

[44] https://aws.amazon.com/elasticache/

[45] http://docs.aws.amazon.com/lambda/latest/dg/env_variables.html#env_

encrypt

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

[47] http://docs.aws.amazon.com/general/latest/gr/signature-version-4.html

[48] http://docs.aws.amazon.com/apigateway/latest/developerguide/how-to- generate-sdk.html

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

[50] http://docs.aws.amazon.com/apigateway/latest/developerguide/apigateway- control-access-to-api.html

[51] https://aws.amazon.com/codepipeline/

[52] http://docs.aws.amazon.com/lambda/latest/dg/vpc.html#vpc-setup-guidelines

[53] http://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/WhatIs CloudWatchLogs.html

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

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

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

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

[58] https://github.com/awslabs/aws-sam-local

[59] http://docs.aws.amazon.com/lambda/latest/dg/current-supported- versions.html

[60] http://docs.aws.amazon.com/lambda/latest/dg/java-exceptions.html

[61] http://docs.aws.amazon.com/lambda/latest/dg/nodejs-prog-mode- exceptions.html

[62] http://docs.aws.amazon.com/lambda/latest/dg/python-exceptions.html

[63] http://docs.aws.amazon.com/lambda/latest/dg/dotnet-exceptions.html

[64] http://docs.aws.amazon.com/codepipeline/latest/userguide/actions-invoke- lambda-function.html

[65] https://aws.amazon.com/codestar/

[66] https://github.com/awslabs/aws-serverless-workshops

[67] https://aws.amazon.com/serverless/