サーバーレスアーキテクチャによる
企業エコノミクスの最適化

2017 年 10 月

© 2017, Amazon Web Services, Inc. or its affiliates. All rights reserved.

注意

本書は情報提供のみを目的としています。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 製品またはサービスの利用について、独自の評価に基づき判断する責任を負います。いずれの AWS 製品またはサービスも、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」提供されます。本書のいかなる内容も、AWS、その関係者、サプライヤ、またはライセンサーからの保証、表明、契約的責任、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 契約によって規定されています。また、本文書は、AWS とお客様との間の契約に属するものではなく、また、当該契約が本文書によって修正されることもありません。

目次

はじめに 1
サーバーレスアプリケーションの理解 2
　サーバーレスアプリケーションのユースケース 3
　サーバーレスは常に最適解か？ 6
クラウドベンダーのサーバーレスプラットフォームの評価 6
　AWS サーバーレスプラットフォーム 11
　AWS サーバーレスプラットフォームの機能 12
導入事例 15
　サーバーレス Web サイト、Web アプリ、 モバイルバックエンド 15
　IoT バックエンド 17
　データ処理 17
　ビッグデータ 19
　IT オートメーション 19
　その他のユースケース 20
まとめ 20
寄稿者 21
詳細情報 21
リファレンスアーキテクチャ 21
ドキュメントの改訂 22

要約

このホワイトペーパーの目的は、最高情報責任者 (CIO)、最高技術責任者 (CTO)、シニアアーキテクトの方に、サーバーレスアーキテクチャおよび市場投入時間、チームアジリティ、IT エコノミクスに対するその影響についての理解を深めていただくことです。設計レベルで、サーバーのアイドル状態の無駄をなくすことで、クラウドベースのソフトウェア設計を劇的にシンプルにするサーバーレスアプローチにより、IT 環境は急速に変化しています。

このホワイトペーパーでは、サーバーレスアプローチと AWS サーバーレスポートフォリオの基本を説明します。また、本書には多くの導入事例も含まれており、サーバーレスアプローチを導入した既存の企業がいかにして大きなアジリティと経済的利益を獲得しているかが説明されています。また本書では、さまざまな規模の組織がサーバーレスアーキテクチャをいかに活用して、従来よりも大幅に低いコストでリアクティブなイベントベースシステムを設計し、クラウドネイティブなマイクロサービスをスピーディに提供することができるかについても説明されています。

はじめに

パブリッククラウドでアプリケーションを実行することでメリットを享受している企業はすでに数多く存在します。その代表的なメリットには、サーバーが起動している時間だけの課金請求 (pay-as-you-go) によるコスト削減や IT リソースのオンデマンド利用によるアジリティの向上などがあります。さまざまなアプリケーションタイプや業界で行われた複数の研究により、既存のアプリケーションアーキテクチャをクラウドに移行することで、総所有コスト (TCO) が下がり、市場投入時間が改善することが証明されています。[1]

オンプレミスおよびプライベートクラウドソリューションと比べて、パブリッククラウドでは、必要なサーバーインスタンスの確保が大幅に容易になるため、そこで実行するアプリケーションの構築、デプロイ、管理がシンプルになります。しかし、今日の企業には、従来のサーバーインスタンスや VM ベースアーキテクチャ以外にもパブリッククラウドをさらに活用する選択肢が用意されています。クラウドを利用すれば自社のハードウェアを購入および維持する必要はなくなるものの、サーバーベースのアーキテクチャの場合はスケーラビリティと信頼性のための設計（たとえば冗長化など）を行う必要は依然としてあります。また、アプリケーションの長期運用や拡張の際にはサーバーインスタンスへのパッチ適用とデプロイの問題も発生します。さらに、ピーク時負荷を考慮してスケールアップしたり、できるだけコストを低くするためにスケールダウンしたりする必要もあります。もちろんこの間も、エンドユーザーの体験や内部システムの整合性に影響が出ないようにする必要があります。加えて、十分に活用されていないアイドル状態のサーバーは無駄なコストがかかります。アナリストは、十分に活用されていないキャパシティがあるサーバーは 85 パーセントにも上ると見積もっています。[2]

AWS Lambda などのサーバーレスコンピューティングサービスは、従来とは異なる手法でアプリケーション設計にアプローチすることでこうした問題に対処し、本質的なコストの削減と市場投入時間の短縮を実現します。
AWS Lambda では、アプリケーション開発においてサーバーインスタンスを取り扱うことにより生じる複雑さを取り払うと同時に、リクエストされた分だけの課金請求モデル (pay-per-request) **を採用しているため、アイドル状態のコンピューティングキャパシティに対して支払いが生じることはありません。**また、Lambda 関数によって組織はマイクロサービスアーキテクチャを自然に導入することができます。インフラストラクチャをなくし Lambda モデルを採用することで、以下の 2 つの経済的なメリットが生まれます。

- アイドル状態のサーバーがもたらす潜在的な問題はその不経済性とともに自然と消滅します。AWS Lambda などのサーバーレスコンピューティングサービスでは、意味のある作業が行われたときにのみ、ミリ秒レベルの精度で料金が発生するため、"コールド" な状態で課金されることは決してありません。
- インスタンス管理 (セキュリティパッチの適用、デプロイ、サーバーのモニタリングなど) が不要になります。つまり、サーバーインスタンスを常時稼働させておくために必要な関連ツール、作業手順、交代制での待機プロセスなどを維持する必要がなくなるということです。Lambda を使ってマイクロサービスを構築することで組織はアジリティを高めることができます。サーバー管理の負担がなくなることで、企業は貴重な IT リソースを本質的なこと、つまりビジネスに充てることができます。

インフラストラクチャコストの大幅削減、チームのアジリティの向上、本質的作業への注力、そして市場投入時間の短縮により、すでにサーバーレスアプローチを導入している企業は、ライバルに大きな差を付けはじめています。

サーバーレスアプリケーションの理解

サーバーレスアプローチの利点は上述の通りですが、実際に導入する際にはどんな検討事項があるのでしょうか？ サーバーレスと従来のサーバーベースのアプリケーションの違いは何でしょうか？

サーバーレスアプリケーションは、開発者が本質的な作業である実際のビジネスロジック記述に注力できるように設計されています。アプリケーションにとって基本的な定型コンポーネントの多く (Web サーバーなど) と差別化につながらない重労働のすべて (信頼性やスケーリングを処理するソフトウェア設計など) は、開発者の手から完全に離れます。残るのは、必要なとき (モバイルユーザーがメッセージを送信する、画像がクラウドにアップロードされる、レコードがストリームに到達するなど) にのみビジネスロジックがトリガーされる、クリーンなファンクショナルアプローチです。アプリケーション設計における非同期的なイベントベースアプローチは、サーバーレスアプリケーションではとても一般的です。 これは、必要な作業が発生したときにのみ実行する (そしてそのときのみ料金が発生する) コードのコンセプトと完全に一致するからです。

サーバーレスアプリケーションはパブリッククラウドの AWS Lambda などのサービスで実行します。これはイベントの受信やクライアントの呼び出しを処理し、それからインスタンスを作成してコードを実行します。このモデルは、

従来のサーバーベースのアプリケーション設計に比べて多くの利点をもたらします。具体的には、

- プロビジョニング、デプロイ、保守/更新、モニタリングなどのサーバー管理が不要。実際のハードウェアとサーバーソフトウェアはすべてクラウドプロバイダーによって制御されます。
- アプリケーションは実際の負荷に応じて自動スケーリング。これは、負荷のピーク時に備える受信用インスタンスと明示的なキャパシティ管理が必要な従来のアプリケーションとは本質的に異なります。
- スケーリングに加えて、可用性と耐障害性の機能も組み込み済み。こうした機能の恩恵を享受するためにコーディングや構成、管理を行う必要はありません。
- アイドル時のリソースに対する利用料からの解放。サーバーリソースを事前準備したりまたは余分に確保することがありません。請求はリクエストされた分だけの課金制 (pay-per-request) で、コードの実行にかかった時間で計算されます。

サーバーレスアプリケーションのユースケース

サーバーレスアプリケーションモデルは汎用的で、スタートアップの Web アプリから Fortune 100 社の株式取引分析プラットフォームまで、ほとんどのタイプのアプリケーションに適用することができます。以下にいくつかの例を挙げます。

- **Web アプリと Web サイト** – サーバーをなくすことで、トラフィックがないときにほとんどコストがかからないと同時に、予想を超えるほどのピーク時負荷でも処理できるほどにスケーリングできる Web アプリを作ることが可能になります。
- **モバイルバックエンド** – サーバーレスなモバイルバックエンドがあれば、モバイルネイティブクライアントの開発がメインの開発者にも分散システム設計の専門知識なしにセキュアで可用性が高く、そして完全にスケーリングされたバックエンドを簡単に作ることができます。
- **メディアおよびログ処理** – サーバーレスアプローチにより自然な並列処理が可能になるため、高負荷のコンピューティングワークロードの処理がよりシンプルになります。複雑なマルチスレッドシステムの構築やコンピューティング基盤の手動スケーリングは不要です。

- **IT オートメーション** – サーバーレス関数はアラームやモニター機能に取り付けることで必要に応じて機能をカスタマイズするために利用できます。Cron ジョブやその他の IT インフラストラクチャの要件は、特にジョブや要件の実行頻度がそれほど高くない、または変動的な性質がある場合、その使用のためにサーバーを所有および維持する必要性がなくなることでかなりシンプルになります。
- **IoT バックエンド** – サーバーレス環境にはネイティブライブラリを含むあらゆるコードを持ち込めるため、接続された IoT デバイスごとに異なる固有のアルゴリズムを実装したクラウドベースのシステムをシンプルに作成できます。
- **チャットボット (音声対応アシスタントを含む) およびその他の Webhook ベースシステム** – サーバーレスアプローチはチャットボットなどのあらゆる Webhook ベースシステムに完全に適合します。必要なとき (ユーザーがチャットボットに情報をリクエストしたときなど) にだけアクション (コードの実行など) を実行する機能により、こうしたアーキテクチャにとってわかりやすく、典型的に低コストなアプローチを実現できます。実際、Amazon Echo の Alexa スキルの大部分は AWS Lambda を使って実装されています。
- **クリックストリームおよびその他のニアリアルタイムのストリーミングデータ処理** – サーバーレスソリューションには、データフローに合わせてスケールアップおよびスケールダウンできる柔軟性があります。このため、アプリケーションごとにスケーラブルなコンピューティングシステムを構築する手間をかけることなく、スループット要件を満たすことができます。Amazon Kinesis などのテクノロジーとペアリングした場合、AWS Lambda はクリックストリーム分析、データ変更トリガー処理、株式取引情報などの高速処理を行うことができます。

上述の高度に導入されたユースケースに加えて、サーバーレスアプローチは以下の分野にも応用されています。

- ビッグデータ処理: MapReduce 型の処理、高速ビデオ変換、株式取引分析、ローン申し込みに対するモンテカルロシミュレーションのような大量のコンピューティング処理。このようなケースにおいて、多くの開発者は、特にイベントを通じてトリガーされる場合、サーバーレスアプローチで並列処理をさせた方がずっと簡単だということに気づき始めています[3]。こうした開発者の中でインフラストラクチャを管理する必要がないサーバーレステクニックの利用は広がり、さまざまなビッグデータ問題に応用されています。

- コンテンツ配信ネットワークを通じて提供されるような、Web アプリケーションおよびアセットの低レイテンシー要件におけるカスタム処理。サーバーレスなイベント処理をインターネットのエッジロケーションで実行させることで、開発者は低レイテンシーの恩恵を受けると同時に、検索やコンテンツのフェッチに起因する処理を簡単にカスタマイズできるようになります。これにより、Web クライアントがどこからアクセスしてきているか (位置) に基づいてレイテンシーを最適化するような、新領域のユースケースが可能になります。

- コネクテッドデバイス (商用、住宅用、携帯型などのモノのインターネット [IoT] デバイス) 内での AWS Lambda 関数などのサーバーレス機能による処理実装。Lambda 関数などのサーバーレスソリューションは、基盤である物理 (および仮想) ハードウェアを自然な形で抽象化して、その上での処理を実現します。これにより、データセンター内でもエッジデバイス上でも、特定のハードウェアアーキテクチャでも別のアーキテクチャでも、同一のプログラミングモデルで処理を実装し、その処理の実行場所を簡単に移行することが可能になります。

- オンプレミス向けのアプライアンス (AWS Snowball Edge など) の拡張機能としてのカスタムロジックおよびデータ処理。サーバーレスアプリケーションは実行環境の詳細からビジネスロジックを分離して設計できるため、アプライアンスを含むさまざまな環境で簡単に機能するようにできます。

通常、サーバーレスアプリケーションは**マイクロサービスアーキテクチャ**に基づいて構築されます。つまり、アプリケーションは個別のジョブを実行する独立したコンポーネントに分離される形になります。それぞれのコンポーネントは Lambda 関数を中核として API、メッセージキュー、データベース、およびその他のコンポーネントと組み合わされた構成で実現され、それぞれのコンポーネントは独立してデプロイ、テスト、スケーリングを行うことができます。実際、サーバーレスアプリケーションはその関数ベースモデルのためマイクロサービスに自然に調和します。モノリシックな設計とアーキテクチャを避けることで、組織はアジリティを向上させることができます。なぜなら、開発者が必要に応じてデータベース層などの個々のコンポーネントを徐々にデプロイしたり、置換またはアップグレードすることができるからです。

多くの場合、サーバーレスアプリケーションへと変更するためにもっとも必要な作業はアプリケーションのビジネスロジックをシンプルな形に分割することです。実際、AWS Lambda などのサービスは主要なプログラミング言語をサポートしており、カスタムライブラリの使用が可能です。長時間実行が求めら

れるタスクは、合理的な時間フレーム内で動作する関数の組み合わせによるワークフローとして表現できるでしょう。これにより、システムは必要に応じて個々のユニットを再起動したり並列処理することができるようになります。

サーバーレスは常に最適解か？

ほぼすべてのモダンアプリケーションは、サーバーレスプラットフォームで正常に実行するように変更することができ、大半の場合はより経済的かつスケーラブルになります。しかし、サーバーレスが最適な選択肢にならないケースも少なからず存在します。以下に具体例を挙げます。

- アプリケーションのあらゆる変更を避ける (既存のアプリケーションコードをそのまま利用したい) という明確な目標がある場合。
- コードを適切に実行するために基盤環境の低レベルでの制御が必要がある場合。たとえば、特定のオペレーティングシステムパッチの指定が求められたり、低レベルのネットワーキングオペレーションにアクセスする必要がある場合など。
- パブリッククラウドに移行していないオンプレミスアプリケーション。

クラウドベンダーのサーバーレスプラットフォームの評価

サーバーレスアプリケーションを設計するとき、企業や組織が考慮する必要があるものはアプリのコードを実行するサーバーレスコンピューティング機能だけではありません。完全なサーバーレスアプリケーションには、ストレージ、メッセージング、診断など、広範におよぶサービス、ツール、機能が必要です。クラウドベンダーが提供するサーバーレスポートフォリオが不完全または断片的な場合、一貫したレベルの抽象化でうまくコーディングできない状況が発生し、サーバーベースのアーキテクチャに戻す必要がでてくるため、サーバーレス開発者にとっては問題になります。

サーバーレスプラットフォームは、コンピューティングおよびストレージコンポーネントなどのサーバーレスアプリケーションの基礎となるサービス群に加えて、サーバーレスアプリケーションを作成、構築、デプロイ、および診断するために必要なツールで構成されます。本番環境でのサーバーレスアプリケーションの実行/運用を考えると、小規模のスタートアップの需要にも、世界中

に展開するグローバル企業の需要にも対処できる、信頼性と柔軟性に優れた信用できるプラットフォームである必要があります。このプラットフォームは、アプリケーションを構成する**すべて**の要素でスケールし、エンドツーエンドの信頼性を提供する必要があります。開発者がサーバーレスソリューションの作成と提供をうまく行うためには、従来型のアプリと同様に多面的なチャレンジが求められます。さまざまな業界の大企業のニーズを満たすために、サーバーレスプラットフォームは以下の機能を提供する必要があります。

図 1: サーバーレスプラットフォームの機能

- パフォーマンス、スケーラビリティ、信頼性に優れた**クラウドロジックレイヤー**。
- 応答性の高い**イベントとデータソースとの連携**および**サードパーティシステムに対する**シンプルな**接続性**。
- 開発者が簡単に始められ、また新しいパターンを既存のソリューションにすばやく安全に追加できる**インテグレーションライブラリ**。
- 開発者がさまざまな分野や、広範なサードパーティシステムおよびユースケースのソリューションを発見および適用することができる、活気に満ちた**開発者のエコシステム**。

- 目的に適合する**アプリケーションモデリングフレームワーク**。
- 状態およびワークフロー管理を提供する**オーケストレーション**。
- **グローバルなスケール**で、各種規制の認定保証プログラムなどにも広範にサポートが行き届いていること。
- **組み込みの信頼性とスケール可能なパフォーマンス**により、あらゆるレベルのスケールにおいてキャパシティのプロビジョニングが不要。
- ファーストパーティとサードパーティの両方のリソースとサービスに対する**組み込みのセキュリティ**と**柔軟なアクセス制御**。

サーバーレスプラットフォームの中心にあるのは、ビジネスロジックを表す関数の実行を担当する**クラウドロジックレイヤー**です。こうした関数は多くの場合、イベントに反応して実行されるため、**ファーストパーティ** (同じクラウドベンダー内の他サービス) **とサードパーティの両方のイベントソースとの**シンプルな**統合機能**が欠かせず、それによってソリューションとしての表現がシンプルになり、さまざまなワークロードに反応した自動スケーリングが可能になります。たとえば、オブジェクトがオブジェクトストアに作成されるたび、またはサーバーレス NoSQL データベースが更新されるたびに、サーバーレス関数の実行が必要かもしれません。サーバーレスアーキテクチャを利用することで、こうしたシステムを実装するために通常必要なスケーリングと管理コードのすべてを排除し、この運用負担をクラウドベンダーに移すことができます。

サーバーレスプラットフォームでの開発を成功させる要因の一つは、企業が簡単に始められることです。たとえば、よく利用されるユースケースに適した、事前設定テンプレートを簡単に見つけられる必要があるでしょう (組み合わせる他のサービスが自社のものかサードパーティ製かにかかわらず) 。こうした**インテグレーションライブラリ**は成功パターン (例えばレコードのストリーム処理や Webhook の実装など) の伝達に欠かせません。これは、開発者がサーバーベースからサーバーレスアーキテクチャに移行途中の期間は特に言えることです。これに密接に関係したものが、コアプラットフォームを囲む**広範かつ多様なエコシステム**です。活気に満ちた大規模エコシステムは、開発者がコミュニティからソリューションを簡単に発見および利用する助けになり、これにより新しいアイデアとアプローチを簡単に提供できるようになります。アプリケーションライフサイクル管理においてさまざまなツールの連鎖が使用されることを考えると、すべての言語、IDE、およびエンタープライズビルドテクノロジーがサーバーレスアプリケーションの構築およびデプロイにおいて既存のアプローチと連携できるようなランタイム、プラグイン、およびオープンソースソリューションがあることが不可欠であり、そのためにも、健全なエコ

システムは必要です。また、開発者の知見 (たとえば Express や Flask などのフレームワーク、人気のあるプログラミング言語など) といった既存の資産を活用することも、サーバーレスアプリケーションには重要です。広範なエコシステムはさまざまな分野で重要な促進材となり、開発者がサーバーレスアーキテクチャで既存のコードをより簡単に再利用することを可能にします。

オープン仕様の AWS Serverless Application Model (AWS SAM) などの**アプリケーションモデリングフレームワーク**を使えば、開発者はサーバーレスアプリケーションを構成するコンポーネントを表現することができるようになり、このアプリケーションを構築、デプロイ、およびモニタリングするために必要なツールやワークフローを有効にすることもできるようになります。サーバーレスプラットフォームの成功の鍵となる他のフレームワークに、**オーケストレーションと状態管理**があります。サーバーレスコンピューティングは基本的にステートレスな特性であるため、ときに長時間実行するワークフローを可能にするための補完的なメカニズムが求められます。オーケストレーションソリューションによって、開発者は、一つのサーバーレスアプリケーション内にある複数の関連するアプリケーションコンポーネントの実行を調整できる一方で、個々のアプリケーションは小さく生存期間の短い関数として取り扱うことが可能になります。また、オーケストレーションサービスはエラー処理をシンプルにし、サーバーレス関数自体が通常許可するよりも長い実行時間が求められるようなレガシーシステムおよびワークフローとの連携を可能にします。

世界中に展開するグローバル企業のような顧客をサポートするために、サーバーレスプラットフォームは**グローバルなスケール**を提供する必要があります。これには、世界中に配備されたデータセンターやエッジロケーションでのサポートが含まれます。エッジロケーションは、エンドユーザーの近くで低レンテンシーのサーバーレスコンピューティングをもたらすための鍵になります。サーバーレスアプリケーションのスケーラビリティと高可用性の提供を担当するのはアプリケーション開発者ではなくプラットフォームであるため、その本質的な**信頼性**は重要です。未処理イベントに対する組み込みの再試行やデッドレターキューなどの機能は、開発者がサーバーレスアプローチを使ってエンドツーエンドの信頼性を備えた堅牢なシステムを構築する上で役に立ちます。サーバーレスアプリケーションでは言語ランタイムと顧客のコードのインスタンス化がオンデマンドに行われることを考えると、パフォーマンス、特に低レンテンシー (オーバーヘッド) も等しく重要です。

そして、プラットフォームはさまざまな**セキュリティおよびアクセス制御機能**を備えている必要があります。これには、仮想プライベートネットワークのサポート、ロールベースおよびアクセスベースの権限、API ベースの認証および

アクセスコントロールメカニズム (サードパーティおよびレガシーシステムを含む) との堅牢な統合、環境変数設定などのアプリケーション要素の暗号化のサポートなどがあります。サーバーレスシステムは、その設計上、以下の理由で本質的に高レベルなセキュリティと制御機能を提供します。

- **セキュリティパッチの適用を含む最高のインフラ管理** – AWS Lambda などのシステムでは、リクエストを実行するサーバーに対して常時モニタリング、循環、およびセキュリティスキャンが行われます。主要なセキュリティ更新が利用可能になると数時間以内にパッチの適用が可能になります。これと比較すると、多くのエンタープライズにおけるインフラ管理では、パッチ適用と更新の SLA はずっと緩いものになってしまっています。
- **サーバーの有効期間の限定化** – AWS Lambda で顧客のコードを実行するすべてのマシンは 1 日に複数回循環し、攻撃への露出を制限すると同時に、オペレーティングシステムやセキュリティパッチが常に最新になるようにしています。
- **リクエストごとの認証、アクセス制御、監査** – AWS Lambda で実行されるすべてのコンピューティングリクエストは、そのソースにかかわらず、個々に認証され、指定のリソースへのアクセスが承認され、完全に監査されます。Amazon API Gateway 経由で AWS データセンターの外から到達したリクエストは、DoS 攻撃の防御など、インターネット向けの防御システムを追加で提供します。サーバーレスアーキテクチャに移行する企業は、AWS CloudTrail を使って、どのユーザーがどのシステムに何の権限でアクセスしているかを詳しく確認することができます。また、AWS Lambda を使って監査レコードをプログラム処理することができます。

AWS サーバーレスプラットフォーム

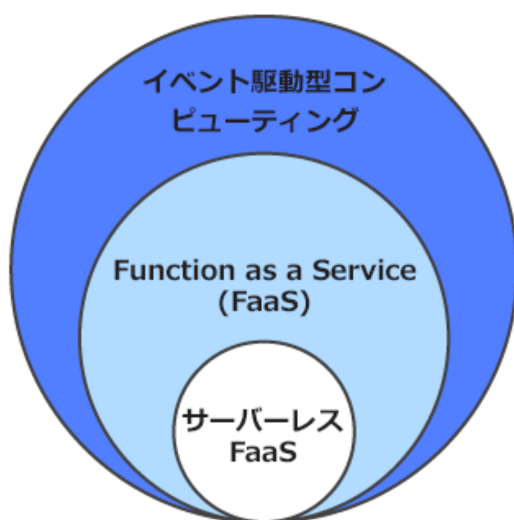
Lambda が 2014 年に導入されて以来、AWS は完全なサーバーレスプラットフォームを築いてきました。このプラットフォームには、他の AWS のサービスやサードパーティサービスとシームレスに統合することができるサーバーレスアプリケーションを組織が作ることを可能にする、完全マネージド型サービスの広範なコレクションが備わっています。AWS サーバーレスプラットフォームのコンポーネントのサブセットとその関係を 図に示します。

図 2: AWS サーバーレスプラットフォームのコンポーネント

AWS サーバーレスプラットフォームの機能

AWS には、前のセクションで完全なサーバーレスプラットフォームの要件として挙げられたすべてのコア機能が備わっています。クラウドロジックレイヤーは、関数を動かすコンピューティングサービスであり、大規模かつプロビジョニングフリーなサーバーレス環境を実現する AWS Lambda によって提供されます。AWS Lambda の派生機能である AWS Lambda@Edge は、エッジ最適化ルーティングを使って極めて低レイテンシーな Lambda 関数を実行するためのサポートを提供します。また AWS Greengrass は、AWS Snowball などのアプライアンスを含むコネクテッドデバイス上で Lambda 関数を実行することを可能にします。

Lambda 関数は、さまざまな AWS サービスおよびサードパーティ製品のイベントによって簡単にトリガーすることが可能で、これにより開発者は、従来のようにインフラストラクチャをセットアップおよび管理する手間をかけることなく、リアクティブなイベント駆動型システム (図 3) を構築することができます。複数のイベントが同時に発生した場合、Lambda は複数の関数を並列実行し、それぞれのトリガーに応答します。Lambda 関数は実際のワークロードのサイズに正確に合わせてスケールし、個々のリクエストまでスケールダウンします。その結果、アイドル状態のサーバーやコンテナが存在する余地は発生しません。インフラストラクチャに対する無駄な支出は、Lambda 関数を使うアーキテクチャでは**設計上**存在し得ないのです。



FaaS (**Function as a Service**) は、デプロイおよび実行の単位として関数に頼るイベント駆動型コンピューティングシステムを構築するためのアプローチの 1 つです。**サーバーレス FaaS** は FaaS の一種で、ここではプログラミングモデルに仮想マシンまたはコンテナが存在せず、また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ビルトインの信頼性を提供します。

図 3: イベント駆動型コンピューティング、FaaS、サーバーレスの関係。

Lambda が提供する AWS サーバーレスコンピューティング機能は、以下の AWS が提供するマネージドサービスともお互いにシームレスに統合します。

- Amazon API Gateway – すべての API プロキシおよび API 管理機能を含む Lambda 関数の HTTP エンドポイント。
- Amazon S3 – オブジェクトの作成、コピー、または削除時に Lambda 関数を自動的に起動するイベントトリガーとして使うことができます。
- Amazon DynamoDB – Lambda 関数を使ってデータベーステーブルのあらゆる変更を処理することができます。
- Amazon SNS – 処理のためにメッセージを Lambda 関数にルーティングすることができます。これにより、公開されたコンテンツに動的に応答することが可能になります。
- Amazon SQS – キューのメッセージを Lambda 関数で簡単に処理することができます。
- Amazon Kinesis Streams – ストリーミングデータにおける順番を維持したレコード処理として Lambda 関数を活用できます。これにより、ニアリアルタイムの判定処理エンジンを容易に構築できます。
- Amazon Kinesis Firehose –Firehose によって取り込まれたレコードに Lambda 関数処理を自動的に適用できます。これにより、変換、フィルタリング、判定機能をデータストリームに容易に追加できます。
- Amazon Athena – クエリの結果セットのオブジェクトごとに Lambda 関数を自動的にトリガーすることができます。
- AWS Step Functions – 複数の Lambda 関数をオーケストレーションして、ヒューマンセントリックプロセス (人が主体のプロセス) やシステム自動化プロセスのような長時間の実行が求められるワークフローを作成できるようにします。
- Amazon CloudWatch Events – Lambda 関数を使って、サードパーティイベントを含むイベントに自動的に応答することができます。
- Amazon Aurora – データベーストリガーを Lambda 関数として記述することができます。

Lambda は、Slack、Algorithmia、Twilio、Loggly、Splunk、SumoLogic、Box などのさまざまなサードパーティサービスとの容易な連携を可能にするインテグレーションライブラリ (設計図/ブループリント) を提供するため、開発者は数行のコードだけで分析、高度なアルゴリズム、コミュニケーションなどを含む応答性の高いアプリケーションを構築することができます。Express (NodeJS アプリケーション向け) や Flask (Python アプリケーション向け) などのさまざまな Web アプリケーションフレームワークが Lambda 関数との連携のために拡張されています。オープンソースの Web アプリケーションプロジェクトには、Serverless Framework、Sparta、Chalice などがあります。

通常、サーバーレスアプリケーションは複数の要素で構成されます。具体的には 1 つ以上の関数、Amazon DynamoDB などのサーバーレスデータベース、およびクライアントが呼び出しを行うためのインターフェース (API) かアプリをトリガーするイベントソースのいずれかで構成されます。これらの要素を整理するために、AWS ではオープン仕様の Serverless Application Model (SAM) を提供しています。SAM を使うことで、開発者は関数、API、イベントソース、データベーステーブル、およびサーバーレスアプリケーションのその他の部分を簡単に記述することができます。また、SAM の使用はソフトウェア開発ライフサイクルのさまざまな段階において開発者に有用なため、AWS はさまざまなツールやサービスで SAM のサポートを提供しています。たとえば、IDE (またはコマンドライン経由) によるローカルテストおよびデバッグとして SAM Local が提供され、AWS CloudFormation を使って SAM アプリのデプロイが可能で、AWS CodeBuild では SAM アプリのビルドがサポートされ、AWS CodePipeline では SAM アプリ向け GitHub ベース CI/CD のサポートが提供されています。こうした AWS サービスによるサポートに加えて、多くのオープンソースフレームワーク、CI/CD プロバイダーおよびパフォーマンス管理ベンダー (Serverless Framework、Claudia、CloudBees、Datadog など) が、SAM および Lambda 関数のサポートを提供しています。これらの例については、サーバーレスアプリケーション用開発者ツールを参照してください。[4]

Lambda 関数 (または SAM アプリの場合は、1 つではなく同時に動作する複数の Lambda 関数の場合もあります) が作成されると、開発者は、Amazon CloudWatch および CloudWatch Logs 向けに自動作成されるメトリクスとログを使ってこれを簡単にモニタリングすることができます。また AWS X-Ray も利用できます。これは開発者が個々の関数とそれが処理するイベントの動作や挙動をトレースすることを可能にする、サービスをまたがるリクエストの追跡およびパフォーマンス分析ソリューションです。

AWS サーバーレスプラットフォームサービスは世界的に展開されており、実質 AWS の世界中のすべてのリージョンで AWS Lambda と Amazon API Gateway がサポートされています。Lambda@Edge はすべてのエッジロケーションで利用できます。[5] Lambda は、お客様がアプリケーションの信頼性を改善する上で役に立つさまざまな機能を提供しています。これには、非同期イベント/順序付けイベントの処理における自動リトライ機能や、アプリケーションによって正常に処理されなかったイベントをキャプチャするデッドレターキューなどがあります。Amazon Virtual Private Cloud (Amazon VPC) との密接な統合や柔軟な認証およびアクセス制御機能により、組織は、最小権限の原則などのベストプラクティスに則したセキュアなアプリケーションを作ることができます。エンドユーザーのセキュリティと管理も同じく簡単です。Amazon Cognito では、Amazon API Gateway および AWS Lambda と簡単に組み合わせることができる認証と認可が提供されます。これにより、サーバーレスでのユーザー登録およびサインイン機能が有効になります。これには、Facebook などのソーシャルプロバイダーや企業ディレクトリとの統合機能が含まれます。

導入事例

企業によるサーバーレスアーキテクチャの応用先は、株式取引の検証から e コマース Web サイトの構築、自然言語処理までさまざまです。AWS Lambda および他の AWS サーバーレスポートフォリオではさまざまなアプリケーションを柔軟に作ることができます。ときには PCI や HIPAA コンプライアンスなどの各種規制に対する認定保証が求められる場合もあるでしょうが、それにも対応しています。以下のセクションでは、特に一般的なユースケースを紹介しますが、これがすべてではありません。顧客事例とユースケースのドキュメントについては、サーバーレスコンピューティングも参照してください。[6]

サーバーレス Web サイト、Web アプリ、モバイルバックエンド

サーバーレスアプローチは、負荷が動的に変化するアプリケーションに適しています。サーバーレスアプローチを使えば、エンドユーザートラフィックが存在しないときはコンピューティングコストが発生しない一方で、突発的なスケーリング機能を提供するため、トラフィックが急増する e コマースサイトのフラッシュセールやソーシャルメディアのメンションなどにも対応することができます。また、従来のインフラストラクチャアプローチと比べて、サーバーレス手法で設計された Web またはモバイルバックエンドは多くの場合、開発、提供、運営にかかるコストが大幅に削減されます。

AWS では、開発者がこのようなアプリケーションをスピーディに構築するために必要な以下のサービスが提供されます。

- Amazon S3 は、静的コンテンツのシンプルなホスティングソリューションを提供します。
- AWS Lambda は、Amazon API Gateway と連動して、関数による動的な API リクエストのサポートを提供します。
- Amazon DynamoDB は、セッションおよびユーザーごとの状態を保管するシンプルなストレージソリューションです。
- Amazon Cognito では、エンドユーザーの登録、認証、およびリソースに対するアクセス制御を簡単に処理することができます。
- AWS SAM を使うと、開発者はアプリケーションのさまざまな要素を記述することができます。
- AWS CodeStar では、クリック操作だけで CI/CD ツールチェーンをセットアップすることができます。

詳しくは、サーバーレス Web アプリケーションの構築パターンについて詳しく説明しているホワイトペーパー AWS Serverless Multi-Tier Architectures を参照してください。[7] 完全なリファレンスアーキテクチャについては、GitHub の Serverless Reference Architecture for creating a Web Application[8] および Serverless Reference Architecture for creating a Mobile Backend[9] を参照してください。

お客様事例 – Bustle.com

Bustle.com は、女性をターゲットにしたニュース、エンターテイメント、ライフスタイル、ファッションの Web サイトです。同社は AWS Lambda および Amazon API Gateway をベースにしたサーバーレスアーキテクチャに移行することで、コストを約 84 パーセント削減することができました。Bustle のエンジニアたちはアジリティが向上し、インフラストラクチャの管理やスケーリングの処理ではなく、新しい製品機能の構築に集中できるようになりました。Bustle のチームはより効率的になり、Bustle のスケールのサイトを構築および運営するために通常必要な人員を半分にすることができました。また、Bustle のサーバーレスバックエンドは同社の 2 つの iOS アプリをサポートしています (Bustle および Romper)。詳細については、Bustle Case Study を参照してください。[10]

IoT バックエンド

サーバーレスアーキテクチャが Web およびモバイルアプリにもたらす利点は、デバイスの台数にシームレスにスケーリングする IoT バックエンドおよびデバイスデータ処理システムの構築にも貢献します。リファレンスアーキテクチャの例については、GitHub の Serverless Reference Architecture for creating an IoT Backend を参照してください。[11]

お客様事例 – iRobot

掃除ロボット「ルンバ」などのロボットを製造する iRobot は、AWS Lambda を AWS IoT サービスと組み合わせて使い、同社の IoT プラットフォームのサーバーレスバックエンドを作っています。サーバーレスアーキテクチャを使うことで、iRobot のエンジニアリングチームはインフラストラクチャを管理したりコードを手動で記述して可用性およびスケーリングを処理する必要がなくなりました。これにより、イノベーションを加速化させ、顧客に集中することが可能になりました。詳しくは、AWS re:Invent 2016 のプレゼンテーション Serverless IoT Back Ends (IOT401)[12] を参照するか、こちらのビデオを視聴してください。[13]

データ処理

最大規模のサーバーレスアプリケーションは膨大な量のデータを取り扱うもので、その多くはリアルタイムプロセッシングです。典型的なサーバーレスデータ処理アーキテクチャは、Amazon Kinesis と AWS Lambda の組み合わせを使ってストリーミングデータを処理するか、Amazon S3 と AWS Lambda を組み合わせて、オブジェクトの作成または更新イベントに応答してコンピューティングをトリガーします。ワークロードにおいて、単純なトリガー処理ではなく複雑なオーケストレーションが必要な場合、AWS Step Functions で進行に応じて 1 つ以上の Lambda 関数を呼び出すステートフルまたは長時間実行型のワークフローを作成することができます。サーバーレスデータ処理アーキテクチャについて詳しくは以下の GitHub のページを参照してください。

- Serverless Reference Architecture for Real-time Stream Processing[14]
- Serverless Reference Architecture for Real-time File Processing[15]
- Image Recognition and Processing Backend reference architecture[16]

お客様事例 – FINRA

Financial Industry Regulatory Authority (FINRA) は、AWS Lambda を使ってサーバーレスデータ処理ソリューションを構築しました。これにより、毎日 370 億回におよぶ株式市場イベントに基づいて 5,000 億件のデータ検証を行うことができています。AWS re:Invent 2016 の The State of Serverless Computing (SVR311) と銘打ったプレゼンテーションで、[17] FINRA のシニアディレクターを務める Tim Griesbach 氏は次のように述べました。「Lambda がこのサーバーレスクラウドソリューションのための最高のソリューションになるということがわかりました。Lambda のおかげで、システムはより速く、より安く、そしてよりスケーラブルになったのです。このため最終的に、コストは 50 パーセント以上削減されました。私たちはそれを毎日どころか毎時間追跡することができるのです」

お客様事例 – Thomson Reuters

メディアや情報を手がける Thomson Reuters は、製品チームが製品利用データを簡単に可視化することができるサーバーレスビジネス分析ソリューションを構築しました。このソリューションでは、AWS Lambda、Amazon Kinesis Streams、Amazon Kinesis Firehose を組み合わせることで分析用のストリーミングイベントデータを収集および処理します。その結果のソリューションは Product Insight と呼ばれ、当初スケジュールの 2 か月前にローンチし、技術的な期待値を超えました。

Thomson Reuters の製品イノベーション部でシニアマネージャーを務める Anders Fritz 氏は次のように述べました。「私たちの当初の目標は 1 秒あたり 2,000 件のイベントを受け入れることでした。私たちのテストでは、AWS 上で構築した Product Insight は 1 秒あたり最大 4,000 件のイベントを処理できることが証明されています。そして 1 年以内にこの数は 1 秒あたり 10,000 件を超えると予想しています」 この数を 1 か月に換算すると、イベント数は 250 億件以上になります。これほどの高スループットでさえも、システムが開始以来データを喪失したことはありません。「AWS の堅牢なフェイルオーバーアーキテクチャと技術機能のおかげで、データの収集を開始して以来、私たちは 1 件のイベントさえ失ったことはありません」と Fritz 氏は言います。詳しくは、AWS 導入事例: Thomson Reuters[18] を参照するか、AWS re:Invent 2016 のプレゼンテーション Real-time Data Processing Using AWS Lambda (SVR301) を視聴してください。[19]

ビッグデータ

AWS Lambda は、大量の並列処理ワークロードに適しています。MapReduce を使ったリファレンスアーキテクチャの例については、Reference architecture for running serverless MapReduce jobs を参照してください。[20]

お客様事例 – Fannie Mae

住宅ローン融資会社の主要な資金調達源である Fannie Mae は、AWS Lambda を使って財務モデリングの “驚異的並列 (embarrassingly parallel)” ワークロードを実行しています。Fannie Mae はモンテカルロシミュレーション処理を使って抵当の将来のキャッシュフローを予想しており、これが抵当リスクの管理に役立っています。同社は、既存の HPC グリッドが同社の増大しているビジネスニーズをもう満たしていないことに気づきました。Fannie Mae は新しいプラットフォームを Lambda を使って構築し、そのシステムはテスト中に 15,000 件の同時関数実行にスケールアップすることに成功しました。新システムは 2,000 万件の抵当データに対する 1 つのシミュレーションの実行を 2 時間で完了しました。これは旧システムの 3 倍の速さです。サーバーレスアーキテクチャを使うことでアイドル状態のコンピューティングリソースに対して料金を支払うことがなくなったため、Fannie Mae は大規模なモンテカルロシミュレーションを高い費用対効果で実行することができるようになりました。また、複数の Lambda 関数を同時に実行することで、計算処理結果の取得までのスピードアップにもつながっています。さらに、サーバー管理とモニタリングの手間を省くと同時に、アプリケーションのスケーリングと信頼性を管理するために以前は必要だった複雑なコードの大部分をなくすことができるようになったため、Fannie Mae は市場投入時間を短縮することもできました。詳しくは、Fannie Mae の AWS Summit 2017 のプレゼンテーション SMC303: Real-time Data Processing Using AWS Lambda を参照してください。[21]

IT オートメーション

サーバーレスアプローチはサーバー管理のオーバーヘッドをなくすため、プロビジョニング、構成、管理、アラーム/モニタリング、スケジュールされた Cron ジョブなどの大半のインフラストラクチャタスクの作成と管理がずっと簡単になります。

お客様事例 – Autodesk

3D 設計およびエンジニアリングソフトウェアを手がける Autodesk は、AWS Lambda を使って同社のエンジニアリング組織全体の AWS アカウントの作成および管理プロセスを自動化しています。Autodesk は、98 パーセントのコスト削減を実現したと見積もっています (アカウントのプロビジョニングにかかる労働時間の節約分を織り込み済み)。同社は、以前のインフラストラクチャベースのプロセスでは 10 時間かかっていたアカウントのプロビジョニングをわずか 10 分に短縮することができるようになりました。サーバーレスソリューションによってオートメーションが増え、手動のタッチポイントが減ったことで、Autodesk は、アカウントのプロビジョニングの自動化はもちろん、その構成作業と社内標準の適用の強制、および監査の実行をも自動化することができています。詳しくは、Autodesk の AWS Summit 2017 のプレゼンテーション SMC301: The State of Serverless Computing を参照してください。[22]
Autodesk Tailor サービスについては GitHub を参照してください。

その他のユースケース

上述のユースケースは、Lambda やその他の AWS のサーバーレスサービスで可能になることのほんの一部にすぎません。その他のユースケースとしては、Amazon Lex および AWS Lambda を使って構築したチャットボットを通じた強力な自然言語の理解、Lambda@Edge と Amazon CloudFront を使った低レイテンシーのグローバルエッジコンピューティング、AWS Snowball 内の Lambda 関数による強力なオンプレミスファイル処理などがあります。 高範囲に利用可能なこのアプローチには、他にもまだまだ優れた機能があります。詳しくは、AWS Lambda を参照してください。[23]

まとめ

サーバーレスアプローチは、従来の IT 管理が抱える 2 つの問題を解決することを目的にしています。1 つは価値を提供することなく企業のバランスシートを悪化させるだけのアイドル状態のサーバー、そしてもう 1 つは差別化された顧客価値を生み出すビジネスへの取り組みの障害になるサーバーのインフラ管理およびサーバーソフトウェアの構築と運用コストです。AWS Lambda やその他の AWS のサーバーレスサービスでは、サーバー、コンテナ、ディスク、およびその他のインフラストラクチャレベルのリソースをプログラミングおよび請求モデルからなくすことで、この長年の問題を解決します。その結果、開

発者は、サービス提供までのスピードアップを実現し、意味のある作業にだけ料金が発生するクリーンなアプリケーションモデルで作業することができます。リアクティブなイベントベースシステムを設計し、クラウドネイティブなマイクロサービスを提供する最も簡単で最も速い手段が、サーバーレスアーキテクチャを利用することです。詳細や関連トピックに関しては、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。[24]

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。

- アマゾン ウェブ サービス、AWS サーバーレスアプリケーション ゼネラルマネージャー、Tim Wagner

詳細情報

詳細については、以下を参照してください。

- Serverless Reference Architectures with AWS Lambda、Amazon.com CTO Werner Vogels
- AWS re:Invent 2016: The State of Serverless Computing [プレゼンテーション]、AWS サーバーレスアプリケーション ゼネラルマネージャー、Tim Wagner
- The economics of serverless cloud computing、451 Research リサーチディレクター、Owen Rogers

リファレンスアーキテクチャ

- Web アプリケーション
- モバイルバックエンド
- IoT バックエンド
- ファイル処理
- ストリーム処理
- 画像認識処理
- MapReduce

ドキュメントの改訂

日付	説明
2017 年 10 月	初版発行

Notes

[1] https://www.forbes.com/sites/moorinsights/2016/04/11/tco-analysis-demonstrates-how-moving-to-the-cloud-can-save-your-company-money/#537e2bd07c4e
http://www.cloudstrategymag.com/articles/86033-understanding-tco-cloud-economics

[2] 2012 年にガートナーが見積もったデータセンターの使用率は 7～12% でした (http://www.nytimes.com/2012/09/23/technology/data-centers-waste-vast-amounts-of-energy-belying-industry-image.html)。2008 年の McKinsey の調査ではこの数値は 6% でした (https://www.sallan.org/pdf-docs/McKinsey_Data_Center_Efficiency.pdf)。Accenture が分析した一連の EC2 ベースアプリケーションの使用率は 7% 程度でした (http://ieeexplore.ieee.org/document/6118751/)。2014 年の NRDC および Anthesis の調査では、2013 年、30% 以上のサーバーが完全な "昏睡状態" (接続はされているが価値のあることは何もしていない状態) だったことがわかっています (http://anthesisgroup.com/wp-content/uploads/2015/06/Case-Study_DataSupports30PercentComatoseEstimate-FINAL_06032015.pdf)。

[3] Occupy the Cloud: Eric Jonas et al., **Distributed Computing for the 99%**, https://arxiv.org/abs/1702.04024.

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

[5] https://aws.amazon.com/lambda/edge/

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

[7] https://d0.awsstatic.com/whitepapers/AWS_Serverless_Multi-Tier_Architectures.pdf

[8] https://github.com/awslabs/lambda-refarch-webapp

[9] https://github.com/awslabs/lambda-refarch-mobilebackend

[10] https://aws.amazon.com/solutions/case-studies/bustle/

[11] https://github.com/awslabs/lambda-refarch-iotbackend

[12] https://www.slideshare.net/AmazonWebServices/aws-reinvent-2016-serverless-iot-back-ends-iot401

[13] https://www.youtube.com/watch?v=gKMaf5E-z7Q

[14] https://github.com/awslabs/lambda-refarch-streamprocessing

[15] https://github.com/awslabs/lambda-refarch-fileprocessing

[16] https://github.com/awslabs/lambda-refarch-imagerecognition

[17] https://www.youtube.com/watch?v=AcGv3qUrRC4&feature=youtu.be&t=1153

[18] https://aws.amazon.com/solutions/case-studies/thomson-reuters/

[19] https://www.youtube.com/watch?v=VFLKOy4GKXQ&feature=youtu.be&t=1449

[20] https://github.com/awslabs/lambda-refarch-mapreduce

[21] https://www.slideshare.net/AmazonWebServices/smc303-realtime-data-processing-using-aws-lambda/28

[22] https://www.slideshare.net/AmazonWebServices/smc301-the-state-of-serverless-computing-75290821/22

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

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