

サーバーレス ストリーミングアーキテクチャ とベストプラクティス

2018年6月



通知

本書は情報提供のみを目的としています。本書は、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。お客様は本文書の情報および AWS 製品またはサービス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。これらの製品またはサービスは、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」提供されます。本書は、AWS、その関連者、サプライヤ、またはライセンサーからの保証、表明、契約的なコミットメント、条件や確約を意味するものではありません。お客様に対する AWS の責任は、AWS 契約によって規定されています。本書は、AWS とお客様との間の契約に属するものではなく、また、当該契約が本書によって修正されることもありません。

目次

はじめに	6
サーバーレスコンピューティングの概要および使用理由	6
ストリーミングデータとは	6
このドキュメントの対象者	7
ストリーム処理のアプリケーションシナリオ	7
サーバーレスストリーム処理	8
想定される 3 パターン	9
サーバーベースとサーバーレスアーキテクチャのコストに関する考慮事項	10
ユースケース例	12
センサーデータ収集	13
ベストプラクティス	14
費用の予測	14
Ingest Transform Load (ITL) のストリーミング	15
ベストプラクティス	16
費用の予測	17
リアルタイム分析	18
ベストプラクティス	21
費用の予測	22
お客様の導入事例	23
まとめ	24
共同編集者	25
付録 A - 詳細なコスト予測	26
コストに関する共通の前提事項	26
付録 A.1 - センサーデータ収集	26
付録 A.2 - Ingest Transform Load (ITL) のストリーミング	29
付録 A.3 - リアルタイム分析	31

付録 B - パターンのデプロイおよびテスト	33
共通タスク	33
付録 B.1 - センサーデータ収集	34
付録 B.2 - Ingest Transform Load (ITL) のストリーミング	38
付録 B.3 - リアルタイム分析	42

要約

サーバーレスコンピューティングでは、アプリケーションとサービスを構築、実行する際に、サーバーについて考慮する必要がありません。つまり、インフラストラクチャの管理やプロビジョニングではなく、ビジネスロジックの書き込みに集中できます。AWS Lambda は AWS のサーバーレスコンピューティングサービスです。これを使い、イベントによって実行がトリガーされる個別の単位 (関数) にコードを書き込むことができます。Lambda は、Amazon S3 バケットの変更、Amazon DynamoDB におけるテーブル内の更新、カスタムアプリケーションからの HTTP リクエストなどのイベントに反応して自動的にコードを実行し、スケールします。AWS Lambda はリクエストされた分だけの課金請求制 (pay-per-request) でもあるため、コードが実行される場合にのみ支払いが発生します。サーバーレスアプローチを使用するとアプリケーションを低コストかつ迅速に構築できるようになり、継続的な管理も削減されます。AWS Lambda とサーバーレスアーキテクチャは、イベント駆動型でスパイクが発生しやすい、コンピューティング要件が常に変動するストリーム処理ワークロードに適しています。ストリーム処理アーキテクチャは、ボリュームの大きいイベントを処理したり、ほぼリアルタイムで情報を生成したりするためにデプロイされる数が増加しています。

このホワイトペーパーでは、サーバーレスアプローチを使った 3 つのストリーム処理パターンについて説明します。各パターンについて、実際のユースケースに適用する方法、実装時のベストプラクティスと考慮事項、費用の予測を説明します。各パターンには、AWS アカウントでパターンをデプロイしやすくするテンプレートも含まれています。

はじめに

サーバーレスコンピューティングの概要および使用理由

サーバーレスコンピューティングでは、アプリケーションとサービスを構築、実行する際に、サーバーについて考慮する必要がありません。サーバーレスアプリケーションではサーバーのプロビジョン、スケール、管理が不要です。サーバーレスアプリケーションはほぼすべてのタイプのアプリケーションまたはバックエンドサービス用に構築でき、高可用性を実現しながら、アプリケーションの実行とスケールに必要なことをすべて行います。

サーバーレスアプリケーションを構築することで、クラウドでもオンプレミスでも、開発者はサーバーやランタイムの管理や操作に煩わされることなく、主力製品に集中することができます。このような経費の削減によって、開発者は、拡張性と信頼性の高い優れた製品の開発に費やす時間とエネルギーを取り戻すことができます。

サーバーレスアプリケーションには主に 3 つの利点があります。

- サーバーの管理が不要
- 柔軟性のあるスケーリング
- 高可用性の自動化

このホワイトペーパーでは、AWS のサーバーレスコンピューティングサービスである AWS Lambda で構築したサーバーレスストリーム処理アプリケーションに的を絞って説明します。AWS Lambda を使用すると、サーバーのプロビジョニングや管理を行わずにコードを実行できます。使用したコンピューティング時間に対してのみ利用料金が発生します。コードを実行していない時間については課金されません。

Lambda では、実質どのようなタイプのアプリケーションやバックエンドサービスでもサーバーを管理する必要なしでコードを実行できます。コードをアップロードするだけで、コードの実行、スケールおよび高可用性を実現するためのすべてを Lambda が管理してくれます。コードは、他の AWS サービスから自動的にトリガーされるように設定することも、ウェブまたはモバイルのアプリケーションから直接呼び出されるように設定することもできます。

ストリーミングデータとは

ストリーミングデータは、数千ものデータソースによって継続的に生成されるデータです。通常、小さなサイズ (キロバイト単位) で同時にデータレコードが送信されます。ストリーミングデータには、モバイルアプリケーションやウェブアプリケーションで生成されるログファイル、e コマースでの購入内容、ゲーム内でのプレイヤーのアクティビティ、ソーシャルネットワーク、証券取引所の立会

場、または地理空間サービスからの情報、およびデータセンター内の接続されたデバイスや計器からのテレメトリなど、広範なデータがあります。

ストリーミングデータは、リアルタイムまたはニアリアルに処理することができ、これまでにない速度で変動する条件や顧客アクティビティにすばやく対応し、実行可能な情報を提供します。これは、データが格納された後で処理または分析されるために、場合によっては古くなったデータから派生した情報が提供される従来のデータベースモデルとは対照的です。

このドキュメントの対象者

このドキュメントの対象者は、ストリーム処理のサーバーレスパターン、ベストプラクティス、考慮事項について理解を深める目的を持ったアーキテクトおよびエンジニアです。ストリーム処理の実務知識があることを前提としています。ストリーム処理の概要については、[ホワイトペーパー: Streaming Data Solutions on AWS with Amazon Kinesis](#) を参照してください。

ストリーム処理のアプリケーションシナリオ

ストリーミングデータ処理には、新しい動的なデータが継続的に生成されるほとんどのシナリオで利点があります。これは、データ処理はほとんどのビッグデータのユースケースに適用され、表 1 に示す幅広い業界で利用されます。このホワイトペーパーでは、モノのインターネット (IoT) 業界のケースを代表的な例として取り上げながら、ストリーム処理アーキテクチャを実際の課題に適用する例を示します。

シナリオ/ 業界	Ingest - Transform - Load の処理対象	連続的なメトリクス 生成	データ分析の 反応領域
IoT	センサー、デバイス テレメトリデータ収集	操作メトリクスおよび ダッシュボード	デバイスオペレーショ ナルインテリジェンス およびアラート
デジタル広告 マーケティング	発行者、入札者の データ集約	到達範囲、収益、コンバ ージョンなどの広告のメ トリクス	広告のユーザーエンゲ ージメント、最適化さ れた入札/購入手段
ゲーム	オンラインデータ集 約、例: 上位 10 名の プレイヤー	大規模マルチプレイオン ラインゲーム (MMOG) の ライブダッシュボード	リーダーボードジェネ レーション、プレイヤー スキルマッチ
一般消費者向け オンライン サービス	クリック ストリーミング分析	表示回数やページビュー などのメトリクス	推薦エンジン、プロア クティブケア

表 1 各分野のストリーミングデータシナリオ



ストリームプロセッシングまたはリアルタイムの分析ワークロードには、いくつかの特徴があります。

- 重要な変更を十分に信頼性高く処理できなければなりません。例えば、検索インデックスのようなデータベースの変更ログをレプリカストアに複製する際にデータの損失なく順番に提供しているようなレベルです。
- 大容量のログまたはイベントデータストリームを処理できるように、十分なスループット容量をサポートしていなければなりません。
- 長期間データをバッファリングまたは保持し、定期的にロードと処理のみを実行できるバッチシステムとの統合をサポートできなければなりません。
- リアルタイムアプリケーションに十分な低レイテンシーを持つデータを提供する必要があります。
- 企業の全負荷を保持できるセントラルシステムとして動作し、すべてが同じセントラルシステムに接続された異種のチームによって構築された何百ものアプリケーションとともに動作できなければなりません。
- ストリーム処理システムとの密接な統合をサポートする必要があります。

サーバーレスストリーム処理

従来、ストリーム処理アーキテクチャでは、Apache Kafka のようなフレームワークを使用してデータを取り込んで格納し、Apache Spark や Storm のような技術を使用してほぼリアルタイムでデータを処理しています。こうしたソフトウェアコンポーネントは、Apache ZooKeeper などのクラスターを管理するためのインフラストラクチャとともにサーバーのクラスターに展開されます。今日、パブリッククラウドを利用している企業は、独自にハードウェアを購入して保守する必要がなくなりました。しかし、サーバーベースのアーキテクチャでは、未だにスケーラビリティと信頼性を設計し、アプリケーションの発展に合わせて、こうした基盤となるサーバーインスタンスにパッチを適用して展開しなければならないという課題を認識する必要があります。さらには、ピーク負荷を考慮してサーバーをスケールし、費用を抑えられそうな時にはできるだけスケールダウンしようとしなければなりません。同時にエンドユーザーのエクスペリエンスと内部システムの完全性を保護する必要があります。

AWS Lambda のようなサーバーレスコンピューティングサービスでは、アプリケーション設計を行うさまざまなアプローチを企業に提供することで、こうした課題を解消することができます。本質的なコスト削減と市場に出るまでの時間の短縮を実現するアプローチで、テクノロジースタックのあらゆるレベルでサーバーを扱う複雑さを排除することができます。インフラの廃止とリクエストされた分だけの課金請求制 (pay-per-request) への移行は、二重の経済的な利点をもたらします。

- 利用されていないサーバーや有効活用されていないストレージなどの問題が解消され、費用が抑えられます。有益な仕事が行われているときのみ料金が発生し、ミリ秒単位の課金で行われるので、AWS Lambda のようなサーバーレスコンピューティングシステムが期待に背くことはありません。
- 24 時間 365 日のサーバー稼働時間をサポートするために必要な関連ツール、プロセス、緊急時のための呼び出し対応ローテーションを維持するという課題が解消されるとともに、セキュリティパッチの適用、展開を含むインスタスの管理から解放され、サーバーインスタスの監視をする必要はなくなります。サーバー管理の負担がなくなれば、企業は IT リソースを重要なビジネスに使用することができます。

インフラストラクチャコストの大幅な削減、機敏性と集中力が改善したチーム、市場に出るまでの時間の短縮により、すでにサーバーレスのアプローチを採用している企業は、競合他社に勝る大きな利点を得ています。

想定される 3 パターン

このホワイトペーパーでは、サーバーレスアプローチを使ったストリーム処理の 3 つのパターンについて説明します。

- **簡単な変換によるデータ収集** - このパターンでは、IoT センサーデバイスが測定をインジェストサービス（データ取り込み）に送信しています。データが取り込まれると、単純な変換を実行して、データを下流処理に適した形に変換することができます。例：医療センサーデバイスは匿名化しなくてはならない患者のデータストリームを生成し、保護された健康情報 (PHI) および個人識別情報 (PII) を隠すことで、HIPAA コンプライアンスを厳守しています。
- **Ingest Transform Load (ITL) のストリーミング** - このパターンは、以前のパターンを拡張し、比較的小さく静的なデータセットからフィールドレベルのデータ補完（エンリッチメント）を行います。例：データベースから検索された位置情報やデバイスの詳細などの医療機器のセンサーデータにデータフィールドを追加します。この例もログデータのエンリッチ化と変換に使用される一般的なパターンです。
- **リアルタイム分析** - このパターンでは、以前のパターンをもとにタイムウィンドウ集約と異常検出の処理を追加します。例：ユーザーアクティビティの追跡、ログ分析、不正検出、推奨エンジン、メンテナンス警告をほぼリアルタイムで実行するなどです。

このセクションでは、各パターンのユースケース例を紹介します。私たちは実装の選択肢について検討し、費用の予測を提供します。このホワイトペーパーで説明されている各サンプルパターンは Github でも利用できるのので ([付録 B を参照](#))、こうしたサンプルパターンを AWS アカウントにすばやく簡単に配置することができます。

サーバーベースとサーバーレスアーキテクチャのコストに関する考慮事項

サーバーレスソリューションの費用をサーバーベースのアプローチと比較する場合、サーバーインフラストラクチャの費用に加えて間接費となるいくつかの要素を考慮する必要があります。

間接費には、追加のパッチ、監視、管理に追加のリソースが必要となるサーバーベースのアプリケーションを保守する義務が含まれます。

コストに関する多数の検討事項は、表 2 に記載されています。

考慮すべき費用	サーバーベースアーキテクチャ	サーバーレスアーキテクチャ
パッチの適用	この環境内のすべてのサーバーでは定期的にパッチを適用する必要があります。これには、オペレーティングシステム (OS) と、ワークロードが機能するために必要な一連のアプリケーションが含まれます。	サーバーレスのアプローチで管理するサーバーがないため、これらのパッチ処理が実行されることはほとんどありません。AWS Lambda を使用すれば、関数コードの更新を行うだけで済みます。
セキュリティスタック	多くの場合、サーバーベースアーキテクチャには、マルウェア対策、ログ監視、ホストベースのファイアウォール、IDS (侵入検知) の製品など、構成と管理を要するセキュリティスタックが考慮すべき事項として含まれています。	ファイアウォールと IDS に該当する機能はAWS サービスにて大部分が処理され、CloudTrail などのサービス固有のセキュリティログは、エージェントやログ収集メカニズムの設定や構成を必要とせずに監査目的で提供されます。
モニタリング	サーバーベースの監視では、低レベルのメトリックは表示される場合があるものの、多くの場合、監視、相関をして、高いサービスレベルのメトリックに変換をする必要が出てきます。たとえば、ストリーム処理パイプラインでは、このパイプラインのパフォーマンスを理解するために、CPU 使用率、ネットワーク使用率、ディスク IO、ディスク容量使用率などの個々のサーバーメトリックをすべて監視し相関させる必要があります。	サーバーレスのアプローチでは、各 AWS サービスが、パイプラインのパフォーマンスを理解するために直接使用できる CloudWatch メトリックを提供します。例: Kinesis Firehose は、IncomingBytes、IncomingRecords、S3.DataFreshness の CloudWatch メトリックを公開しており、オペレーターはストリーミングアプリケーションのパフォーマンスをより直接的に理解することができます。
補助インフラストラクチャ	多くの場合、サーバーベースのクラスターでは、クラスター管理ソフトウェアなどのインフラストラクチャをサポートする必要があります。集中管理されたログ収集も管理する必要があります。	AWS は、AWS サービスを提供するクラスターを管理し、顧客のこうした負担を解消します。さらに、AWS Lambda のようなサービスでは、ログ記録を CloudWatch ログに配信することで、集中的なログ収集、処理、分析を行うことができます。
ソフトウェアライセンス	お客様は、オペレーティングシステム、ストリーミングプラットフォーム、アプリケーションサーバー、セキュリティ、管理、監視用パッケージなどの、ソフトウェアのライセンスおよび商用サポートの費用を考慮する必要があります。	AWS サービスの料金にはソフトウェアライセンスが含まれており、こうしたサービスのセキュリティ、管理、監視については追加パッケージは必要ありません。

表 2 サーバーレスとサーバーベースのアーキテクチャを比較する際のコスト面で考慮すべき事項

ユースケース例

このホワイトペーパーでは、病院で治療を受けている患者に接続されている医療用センサーデバイスのユースケースを題材として話を進めていきます。まず、規模に応じて確実にセンサーデータを取得する必要があります。次に、患者の保護された健康情報 (PHI) を匿名化して、特定できない方法で処理します。処理の一環として、新たなフィールドでデータを充実させたりデータを変換したりしなければならない場合があります。最後に、センサーデータをリアルタイムで分析し、異常の検出やトレンドパターンの発現などの知見を得ます。以下のセクションでは、このユースケース例を 3 つのパターンの実現例とともに詳しく説明します。

センサーデータ収集

ウェアラブルデバイスの使用は急速に拡大している IoT ユースケースです。ウェアラブルデバイスを使用すると、患者の健康状態をリアルタイムでモニタリングすることができます。ウェアラブルデバイスを使用するためには、まず規模に応じて確実にセンサーデータを取得する必要があります。その後、データを匿名化し、患者の個人健康情報 (PHI) を削除しなければなりません。そうすることで匿名化されたデータは下流の他のシステムで処理できるようになります。

これらの要件を満たすソリューションの例を図 1 に示します。

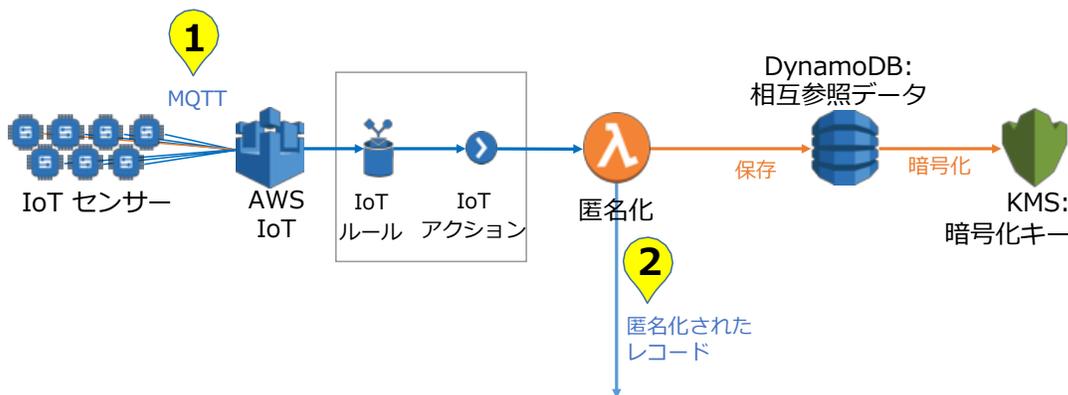


図 1. 医療機器のユースケース例の概要 - センサーまたはデバイスのデータ収集

図 1 のポイント 1 では、1 つ以上の医療機器 (「IoT センサー」) が病院内の患者に接続されています。このデバイスでは、センサーデータを病院の IoT ゲートウェイに送信し、MQTT プロトコルを使用して AWS の IoT ゲートウェイサービスに安全に転送して処理します。この時点でのサンプルレコードは次のとおりです。

```
{
  "timestamp": "2018-01-27T05:11:50",
  "device_id": "device8401",
  "patient_id": "patient2605",
  "name": "Eugenia Gottlieb",
  "dob": "08/27/1977",
  "temperature": 100.3,
  "pulse": 108.6,
  "oxygen_percent": 48.4,
  "systolic": 110.2,
  "diastolic": 75.6
}
```

次に、データを匿名化して、特定されない方法で処理しなければなりません。AWS IoT は、特定の患者セットの測定値を選択する IoT ルールと、これらの選択された測定値を匿名化する Lambda 関数に配信する IoT アクションで構成されています。Lambda は 3 つのタスクを実行します。最初にレコードから PHI 属性と PII 属性 (患者名 "name" と患者の誕生日 "dob") を削除します。次に、後で相互参照できるように、患者名と患者の誕生日の属性を暗号化し、患者 ID と一緒に DynamoDB テーブルに格納します。最後に、匿名化されたレコードとして Kinesis Data Firehose の配信ストリーム (図 1 のポイント 2) に送信します。この時点のサンプルレコードを以下に示します - 生年月日 ("dob") と「名前」のフィールドは削除されています。

```
{
  "timestamp": "2018-01-27T05:11:50",
  "device_id": "device8401",
  "patient_id": "patient2605",
  "temperature": 100.3,
  "pulse": 108.6,
  "oxygen_percent": 48.4,
  "systolic": 110.2,
  "diastolic": 75.6,
}
```

ベストプラクティス

このパターンをデプロイする際は、以下のベストプラクティスを考慮してください。

- Lambda ハンドラーのエントリーポイントをコアロジックから分離します。これにより、多くの単体テスト可能な関数を作ることができます。
- Lambda 関数のパフォーマンスのために、コンテナが再利用される状況を利用するようにします。そして、コードが取得する必要のある構成情報や依存関係がある場合は、最初の実行後にローカルに格納され、次回以降は参照するだけですむように設計します。また、すべての呼び出しの変数/オブジェクトの再初期化を制限するようにします。静的な初期化/コンストラクタ、グローバル/静的変数、シングルトンを代わりに使用します。
- データを S3 に配信する場合、目的のオブジェクトサイズに合わせて Kinesis Data Firehose のバッファサイズとバッファ間隔を調整します。小さいオブジェクトの場合、オブジェクトの PUT アクションと GET アクションのコストは高くなります。
- 圧縮形式を使用してストレージとデータ転送コストをさらに削減します。Kinesis Data Firehose は GZIP、Snappy、Zip のデータ圧縮をサポートしています。

費用の予測

AWS IoT ゲートウェイへのセンサーデータの採集、Lambda 関数による匿名化処理、DynamoDB テーブルへの相互参照データの格納に掛かる AWS サービスの月額費用は、小規模シナリオで 117.19 USD、中規模シナリオで 1,132.01 USD、大規模シナリオで 4,977.99 USD です。

サービスごとの料金の内訳については、「[付録 A.1 - センサーデータ収集](#)」を参照してください。

Ingest Transform Load (ITL) のストリーミング

センサーデータは取り込まれた後、比較的小さく静的なデータセットからフィールドレベルの置換処理やデータ補完 (エンリッチメント) にするような、シンプルな変換によるエンリッチ化や変更を行う必要があります。

ユースケース例では、センサーの測定をデバイスモデルと製造元の情報に関連付ける場合を紹介しています。図 3 にこの要件を満たすソリューションを示します。前述のパターンから、匿名化されたレコードは、Kinesis Data Firehose の配信ストリーム (図 2 のポイント 2) に集約されています。

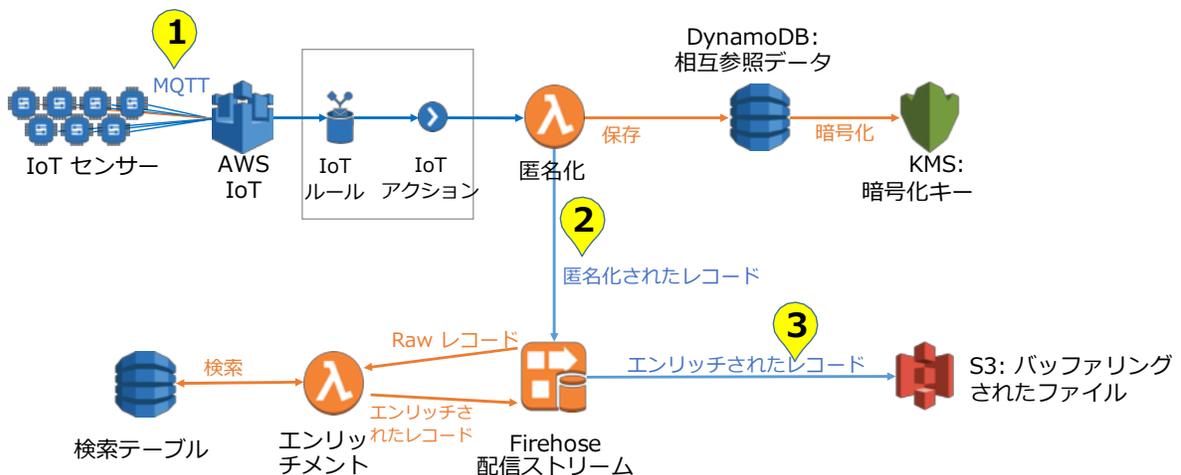


図 2. 医療機器のユースケース例の概要 - Ingest Transform Load (ITL) のストリーミング

このソリューションでは、配信ストリームによってレコードが受信されるたびに、Kinesis Data Firehose によって呼び出される Lambda 関数を導入しています。Lambda 関数は各デバイスに関する情報を DynamoDB テーブルから検索し、フィールドとして測定レコードに追加します。次に Firehose は変更済みレコードを設定済みの送信先に送信します (図 2 のポイント 3)。ソースレコードのコピーがバックアップとして、また今後の分析用に S3 に保存されます。以下に、この時点でのサンプルレコードを示します (エンリッチされたフィールドが強調表示されています)。

```
{
  "timestamp": "2018-01-27T05:11:50",
  "device_id": "device8401",
  "patient_id": "patient2605",
  "temperature": 100.3,
  "pulse": 108.6,
  "oxygen_percent": 48.4,
  "systolic": 110.2,
  "diastolic": 75.6,
  "manufacturer": "Manufacturer 09",
  "model": "Model 02"
}
```

このパターンで AWS Lambda 関数を変換に使用すれば、これまでのようなインフラストラクチャの設定と管理の手間を省くことができます。Lambda は同時変換呼び出しにตอบสนองして関数の複数のコピーを並列で実行し、ワークロードのサイズに合わせて個別のリクエストに正確にスケールします。この結果、アイドル状態のインフラストラクチャと無駄なインフラストラクチャのコストの問題が解消されます。

データが Firehose に取り込まれると Lambda 関数が呼び出され、以下のような簡単な変換を実行させることができます。

- 数値のタイプスタンプ情報を日、月、または年に基づいてデータをクエリできる可読形式の文字列に変換します。たとえば、タイプスタンプ「1508039751778」をタイムスタンプ文字列「2017-10-15T03:55:51.778000」に変換します。
- デバイス ID を使用してテーブル (DynamoDB に格納されている) をクエリすることによりデータレコードをエンリッチし、対応するデバイスの製造元とデバイスモデルを取得します。関数は DynamoDB を頻繁にクエリしなくても済むようにデバイスの詳細をメモリにキャッシュすることで、読み込みキャッシュユニット (RCU) 数を削減できます。この設計は [AWS Lambda でコンテナを再利用する状況での](#)利点を上手く使って、コンテナの再利用時に効率的にデータをキャッシュしている例です。

ベストプラクティス

このパターンをデプロイする際は、以下のベストプラクティスを考慮してください。

- データを S3 に配信する場合、目的のオブジェクトサイズに合わせて Kinesis Data Firehose のバッファサイズとバッファ間隔を調整します。小さいオブジェクトの場合、オブジェクトアクションのコスト (PUT と GET) は高くなります。

- 圧縮形式を使用してストレージとデータ転送コストを削減します。Kinesis Data Firehose は GZIP、Snappy、Zip のデータ圧縮をサポートしています。
- Redshift へのデータ共有については、[Amazon Redshift のデータロードのベストプラクティス](#)を考慮します。
- AWS Lambda 関数を使用して Firehose の配信ストリームでデータを変換する場合は、配信ストリームのソースレコードバックアップを有効にすることを検討してください。この機能は変換されたレコードを送信先に配信する間に、未変換レコードを S3 へすべてバックアップします。これにより S3 のストレージサイズは増加しますが、このバックアップデータは Lambda の変換処理でエラーが発生した場合に役立ちます。
- Firehose はバッファサイズまたは 3MB のいずれか少ない方までレコードをバッファリングし、バッファリングした各バッチで変換のための Lambda 関数を呼び出すようにします。このバッファサイズによって Lambda 関数の呼び出し回数と、呼び出しごとに送信される作業量が決まります。バッファサイズが小さいと Lambda 関数の呼び出し回数が多くなり、呼び出しコストが高くなります。バッファサイズが大きいと呼び出し回数は少なくなりますが、変換の複雑度に応じて呼び出しごとの作業量が増え、関数の最大呼び出し所要時間が 5 分を超える可能性もあります。
- 変換中のデータ参照 (ルックアップ) はレコードの取り込み速度とともに発生します。Amazon DynamoDB Accelerator (DAX) を使用して結果をキャッシュし、ルックアップのレイテンシーを削減してルックアップスループットを上げることを検討してください。

費用の予測

Kinesis Data Firehose へのストリーミングデータの取り込み、Lambda 関数での変換、ソースレコードと変換されたレコード両方の S3 への配信から発生する AWS サービスの月額料金は、スモールシナリオの場合 18.11 USD、ミディアムシナリオの場合 138.16 USD、ラージシナリオの場合 672.06 USD ほどです。

サービスごとの料金の内訳については、「[付録 A.2 - Ingest Transform Load \(ITL\) のストリーミング](#)」を参照してください。

リアルタイム分析

ストリーミングデータが取り込まれてエンリッチされると、リアルタイムで分析され知見が得られるようになります。

ユースケース例では、匿名化されてエンリッチされたレコードをリアルタイムに分析し、病院内のデバイスから異常を検出して、該当するデバイスの製造元に通知します。デバイスの状態を評価することにより、製造元は障害が発生しそうな状況を示すパターンを特定し始めることができます。また、ほぼリアルタイムに情報をモニタリングすることで、デバイスのプロバイダは不具合が発生する前に懸念事項にすばやく反応できます。異常が検出された場合、そのデバイスは直ちにに取り除かれ、検査に送られます。このアプローチには、デバイスのダウンタイムの削減、デバイスのモニタリング強化、人件費の削減、メンテナンスのスケジューリングの効率化といった利点があります。また、デバイスの製造元はパフォーマンスを基準にしたさまざまなメンテナンス契約を病院に提供できるようになります。

図 3 にこの要件を満たすソリューションを示します。

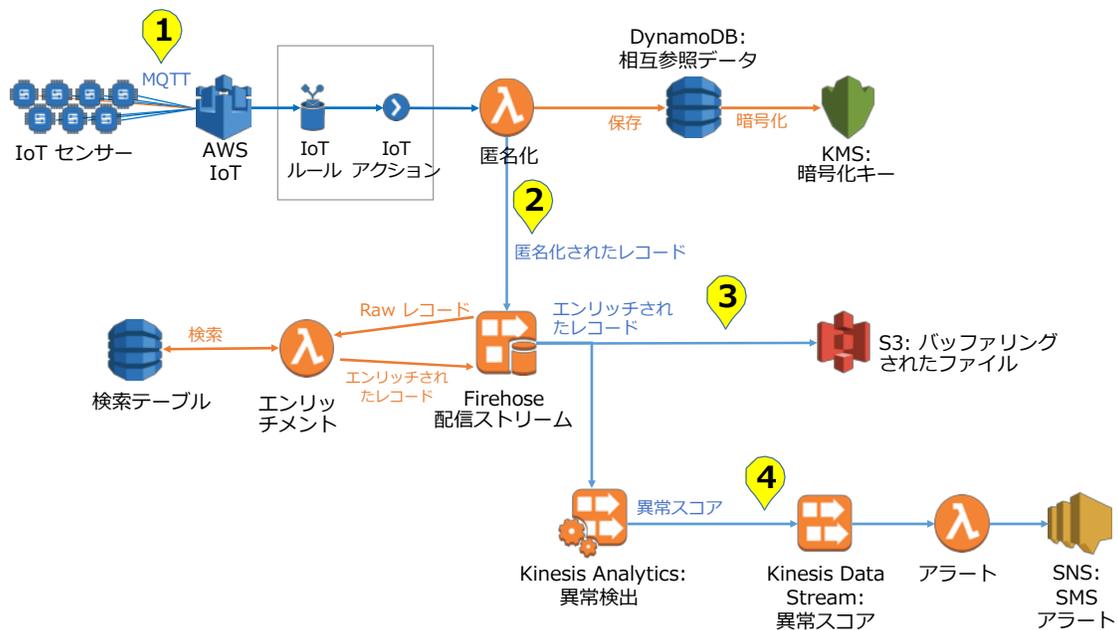


図 3. 医療機器のユースケースの概要 - リアルタイム分析

前述のパターンからエンリッチされたレコード (図 3 のポイント 3) のコピーを Kinesis Data Analytics アプリケーションに配信することで、製造元のすべてのデバイスに対する測定値から異常を検出できるようになります。異常スコア (図 3 のポイント 4) は Kinesis Data Stream に送信され、Lambda 関数で処理されます。以下に、異常スコアが追加されたサンプルレコードを示します。

```
{
  "timestamp": "2018-01-27T05:11:50",
  "device_id": "device8401",
  "patient_id": "patient2605",
  "temperature": 100.3,
  "pulse": 108.6,
  "oxygen_percent": 48.4,
  "systolic": 110.2,
  "diastolic": 75.6,
  "manufacturer": "Manufacturer 09",
  "model": "Model 02",
  "anomaly_score": 0.9845
}
```

検出された異常の範囲またはしきい値に基づいて、Lambda 関数は製造元に通知を送信します。通知には、異常の原因となったモデル番号、デバイス ID、一連の測定値が含まれます。

Kinesis Analytics のアプリケーションコードは、事前に構築されている異常検出関数 **RANDOM_CUT_FOREST** で構成されます。この関数は異常検出で最も重要です。関数はメッセージから数値データを処理します。この場合は "temperature"、"pulse"、"oxygen_percent"、"systolic"、"diastolic" によって異常スコアが決定されます。

関数 **RANDOM_CUT_FOREST** の詳細については、Amazon Kinesis Analytics のドキュメント (<https://docs.aws.amazon.com/kinesisanalytics/latest/sqlref/sqlrf-random-cut-forest.html>) を参照してください。

以下に、異常検出例を示します。この図には、3 つのクラスターとランダムに入り込んだいくつかの異常が表示されています。赤い四角は **RANDOM_CUT_FOREST** 関数に従って最も高い異常スコアを受信したレコードを示します。青のダイヤモンド形は残りのレコードを表します。最も高いスコアが付いたレコードはクラスターの外に配置される傾向があることに注目してください。

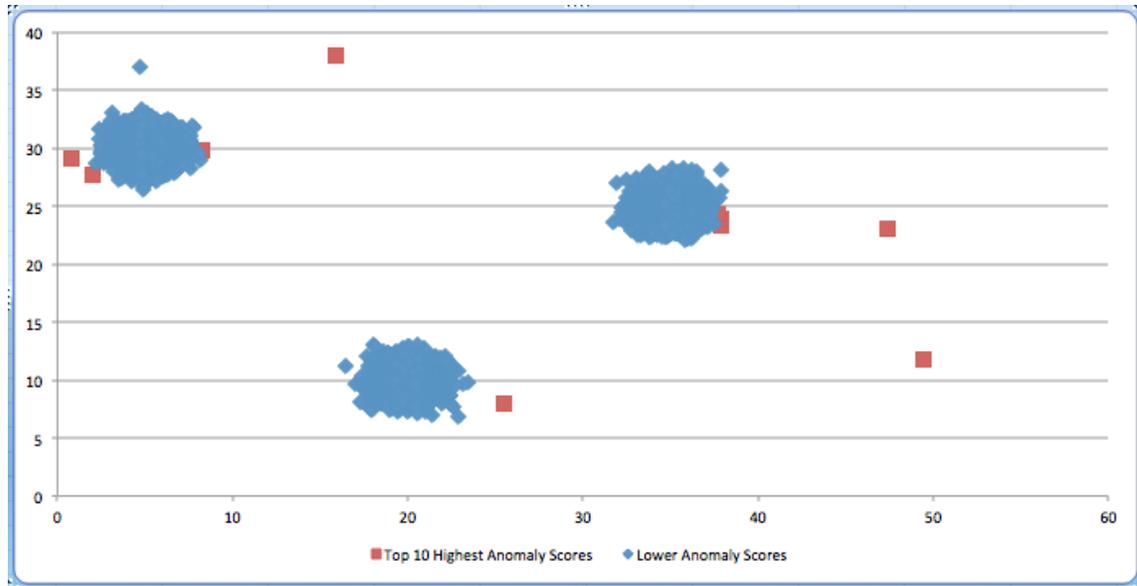


図 4.異常検出例

以下に Kinesis Analytics アプリケーションコードを示します。最初のコードブロックは RANDOM_CUT_FOREST 関数によって生成された異常スコアの出力を格納するためのものです。コードブロックは受信センサーデータストリーム ("STREAM_PUMP") を使用して、異常検出関数 RANDOM_CUT_FOREST を呼び出します。

```
-- Creates a temporary stream and defines a schema
CREATE OR REPLACE STREAM "TEMP_STREAM" (
  "device_id" VARCHAR(16),
  "manufacturer" VARCHAR(16),
  "model" VARCHAR(16),
  "temperature" integer,
  "pulse" integer,
  "oxygen_percent" integer,
  "systolic" integer,
  "diastolic" integer,
  "ANOMALY_SCORE" DOUBLE);

-- Compute an anomaly score for each record in the source stream
-- using Random Cut Forest
CREATE OR REPLACE PUMP "STREAM_PUMP" AS INSERT INTO
"TEMP_STREAM"
```

```
SELECT STREAM
"device_id", "manufacturer", "model", "temperature", "pulse",
"oxygen_percent", "systolic", "diastolic", "ANOMALY_SCORE" FROM
TABLE (RANDOM_CUT_FOREST (
    CURSOR (SELECT STREAM "device_id", "manufacturer", "model",
"temperature", "pulse", "oxygen_percent", "systolic",
"diastolic"
        FROM "SOURCE_SQL_STREAM_001")
    )
);
```

このユースケースの後続処理である Lambda 関数では、異常スコアのある分析データレコードに対して以下の簡単なタスクを実行します。

- この Lambda 関数では **ANOMALY_THRESHOLD_SCORE** と **SNS_TOPIC_ARN** という 2 つの環境変数を使用しています。環境変数 **ANOMALY_THRESHOLD_SCORE** は異常判定のしきい値として使うものですので、対象データを使用して初期テストを実行し、設定する適切な値を決定した後、設定/調整するとよいでしょう。**SNS_TOPIC_ARN** は Lambda 関数から異常レコードが配信される SNS トピックを表します。
- Lambda 関数は分析データレコードのバッチに対して反復処理を行って異常スコアを確認し、しきい値を超える異常スコアのレコードを見つけます。
- 次に Lambda 関数はしきい値レコードを環境変数で定義された SNS トピックに発行します。「付録B.3」の「パッケージとデプロイ」セクションで説明するデプロイメントスクリプトで、SNS トピックの受信登録に使用する E メール用に変数 **NotificationEmailAddress** を設定します。

センサーデータも S3 に格納されるため、さまざまなドメインで作業するデータサイエンティストがあらゆるタイプの分析でそのデータを今後利用できるようになります。ストリームセンサーデータは Kinesis Firehose 配信ストリームに渡され、S3 への PUT 操作を行う前にバッファリングされて圧縮されます。

ベストプラクティス

このパターンをデプロイする際は、以下のベストプラクティスを考慮してください。

- Amazon CloudWatch アラームを設定する。Amazon Kinesis Data Analytics が提供する CloudWatch メトリクスを使用: 入力バイト、入力レコード (アプリケーションに入力するバイト数とレコード数)、出力バイト、出力レコード、MillisBehindLatest (ストリーミングソースからアプリケーションの読み込みの遅れを追跡)

- 入力スキーマを定義する。自動的に推測されたスキーマを適切にテストします。スキーマの検出プロセスではストリーミングソースのサンプルレコードのみを使用してスキーマを推測しています。ストリーミングソースに多数のレコードタイプがある場合、1 つ以上のレコードタイプのサンプリングを検出 API が見逃す可能性があり、その場合、スキーマにストリーミングソースのデータが正確に反映されません。
- 出力に接続する。すべてのアプリケーションに 2 つ以上の出力を設定することをお勧めします。最初の送信先は SQL クエリの結果を挿入するために使用します。2 番目の送信先は、エラーストリーム全体を挿入して、Amazon Kinesis Firehose 配信ストリーム経由で S3 バケットに送信するために使用します。
- アプリケーションコードのオーサリング:
 - 開発中は、SQL ステートメントのウィンドウサイズを小さく保ち、結果がすばやく表示されるようにします。アプリケーションを本番稼働環境にデプロイする際に、適切なウィンドウサイズに設定できます。
 - 1 つの複雑な SQL ステートメントではなく、複数のステートメントに分割し、仲介となるアプリケーション内ストリームに各ステップで結果を保存することを検討できます。この操作によりデバッグを短縮できる場合があります。
 - タンプリングウィンドウを使用する場合は、処理時間用と論理時間用 (取り込み時間またはイベント時間) に 2 つのウィンドウを使用することをお勧めします。詳細については、[Timestamps and the ROWTIME Column](#) を参照してください。

費用の予測

Kinesis Analytics で異常検出を行い、Lambda 関数を使用して SNS トピックに異常スコアを報告し、今後の分析用に異常スコアデータを S3 バケットに格納する AWS サービスの月額料金は、スモールシナリオの場合 **705.81 USD** です。ミディアムシナリオの場合 **817.09 USD** でラージシナリオの場合 **1312.05 USD** です。

サービスごとの料金の内訳については、「[付録 A.3 -リアルタイム分析](#)」を参照してください。

お客様の導入事例

さまざまな事業分野にわたる各事業規模のお客様がデータ処理と分析にサーバーレスアプローチを使用しています。いくつかの事例を以下に示します。サーバーレスの導入事例やお客様の声は、[AWS Lambda のリソース](#)ページをご覧ください。



THOMSON REUTERS

[Thomson Reuters](#) は、世界中の企業や専門家向けの代表的な情報源であり、世界で最も信頼されている報道機関も含まれています。2016 年に Thomson Reuters は、自社サービスから生成される分析データをキャプチャ、分析、可視化して、製品チームによるユーザーエクスペリエンスの継続的改善に役立つ洞察を提供するソリューションを構築することにしました。このソリューションは Product Insights と呼ばれ、[AWS Lambda](#)、[Amazon Kinesis Streams](#)、[Amazon Kinesis Data Firehose](#) を使用してストリーミングデータパイプラインへのデータの取り込みと配信を行います。データはその後、リアルタイムのデータ分析用に固定ストレージまたは Elasticsearch クラスタにパイプラインで送られます。Thomson Reuters は現在、1 か月あたり 250 億ものイベントを処理できます。

[導入事例を読む >>](#)



[iRobot](#) は世界トップクラスの一般消費者向けロボット専門メーカーであり、家庭の中でも外でもより多くのことができるように人々を支援するロボットを設計、製造しています。2002 年には掃除機ロボット「ルンバ」をリリースし、ロボット家電というカテゴリを創出しました。現在、iRobot は「コネクテッド」ルンバ掃除機が 60 か国以上で稼働しており、2017 年末までの予測総売上高は 200 万以上になると報告しています。このような規模をグローバルレベルで処理するために、iRobot はミッションクリティカルなデブプラットフォーム向けに完全にサーバーレスのアーキテクチャを導入しました。このソリューションの中心にあるのが [AWS Lambda](#)、[AWS IoT プラットフォーム](#)、[Amazon Kinesis](#) です。サーバーレスにしたことで、iRobot はクラウドプラットフォームのコストを低く保つことができ、このソリューションを 10 人に満たない人数で管理しています。

[導入事例を読む >>](#)



Nextdoor

[Nextdoor](#)は、近隣住民同士で使う無料のプライベートソーシャルネットワークです。Nextdoor のシステムチームは、1 日あたり 25 億の syslog とトラッキングイベントを扱うデータ取得パイプラインの管理を担当しています。データボリュームが増すにつれ、チームはデータ取得パイプラインの安定性を維持することで手一杯になり、製品開発などの主要な担当業務が妨げられるようになりました。

Nextdoor は、データパイプラインを動かすために大規模インフラストラクチャの稼働を続ける代わりに、サーバーレス ETL を [AWS Lambda](#) で構築することに決めました。Nextdoor のサーバーレスソリューションについて、また Nextdoor のオープンソースプロジェクト [Bender](#) で同社規模のサーバーレス ETL を活用する方法についての詳細は、AWS re:Invent 2017 での Nextdoor によるトークをご覧ください。

[Nextdoor のトークを視聴する »](#)

まとめ

サーバーレスコンピューティングを利用すれば、あらゆるレベルの技術スタックにおけるサーバーインフラストラクチャの構築と管理にまつわる、差別化されていない厄介な作業を行う必要がなくなります。また、リクエストの分だけ支払う課金モデルを導入して、使用していないコンピューティング性能に対するコストを削減できます。データのストリーム処理を行うことで、従来のバッチ処理からリアルタイム分析へとアプリケーションを進化させることができ、ビジネスのパフォーマンスに関する情報をより深いところから抽出できるようになります。このホワイトペーパーでは、これら 2 つの強力な概念を組み合わせることによって、開発者は複雑なデータ処理アプリケーションをより迅速に開発し、企業は有益な作業のみに支払うというクリーンなアプリケーションモデルを使用できることを確認しました。サーバーレスコンピューティングの詳細については、AWS の [サーバーレスコンピューティングとアプリケーション](#) のページを参照してください。また、より多くのリソース、お客様によるトーク、チュートリアルを [Serverless Data Processing](#) のページをご覧ください。

追加リソース

サーバーレスデータ処理に関するさらなるリソース (チュートリアル、ドキュメント、顧客事例、トークなど) については、AWS の [Serverless Data Processing のページ](#) を参照してください。サーバーレスおよび AWS Lambda に関するさらなるリソースについては、[AWS Lambda のリソースのページ](#) を参照してください。

関連するホワイトペーパーで、サーバーレスコンピューティングとデータ処理についてお読みいただけます。

- [Amazon Kinesis を使用した AWS でのストリーミングデータソリューション](#)
- [サーバーレス: 企業エコノミクスの在り方を一変させる](#)
- [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](#)

共同編集者

このドキュメントは、下記の個人および組織の共同編集によるものです。

- Amazon Web Services、グローバルライフサイエンス担当シニアソリューションアーキテクト、Akhtar Hossain
- Amazon Web Services、ソリューションアーキテクト、Maitreya Ranganath
- Amazon Web Services、製品マーケティングマネージャー、Linda Lian
- Amazon Web Services、製品マネージャー、David Nasi

付録 A - 詳細なコスト予測

この付録では、本文で概要を述べたコスト予測の詳細について説明します。

コストに関する共通の前提事項

次に挙げる 3 つのトラフィックシナリオの各パターンを実装するのに必要なリソースの 1 か月あたりコストを予測します。

- スモール - ピークレートは 50 レコード/秒、1 レコードあたり平均 1 KB
- ミディアム - ピークレートは 1000 レコード/秒、1 レコードあたり平均 1 KB
- ラージ - ピークレートは 5000 レコード/秒、1 レコードあたり平均 1 KB

前提として、各シナリオのピークレートでレコードが取得されるピーク時間が 1 日あたり 4 時間あり、残りの 20 時間ではピーク時の 20% までデータ取得レートが下がるものとします。これは、毎月取得されるデータボリュームを予測するための単純な可変レートモデルです。

付録 A.1 - センサーデータ収集

センサーデータ収集の場合の 1 か月あたり予測コストの詳細を下の表 3 に示します。このサービスの設定は次のとおりです。

- 前提として、AWS IoT ゲートウェイサービスの 1 日あたりの接続性は、スモールのユースケースでは 25%/日、ミディアムでは 50%/日、ラージでは 70%/日であるとしています。
- Kinesis Firehose のバッファサイズは 100 MB です。
- Kinesis Firehose のバッファ間隔は 5 分 (300 秒) です。

	スモール	ミディアム	ラージ
ピークレート (メッセージ/秒)	100	1000	5000
レコードサイズ (KB)	1	1	1
1 日あたりレコード (数)	2880000	28800000	144000000
1 か月あたりレコード (数)	86400000	864000000	4320000000
1 か月あたりボリューム (KB)	86400000	864000000	4320000000
1 か月あたりボリューム (GB)	82.39746094	823.9746094	4119.873047
1 か月あたりボリューム (TB)	0.08046627	0.804662704	4.023313522

(表は次頁へ続く)

AWS IoT コスト			
デバイス数	1	1	1
接続性 - 時間の割合 (%) / 日	25	50	75
メッセージング (メッセージ数 / 日)	2880000	28800000	144000000
ルールエンジン (ルール数)	1	1	1
デバイスシャドウ	0	0	0
デバイスレジストリ	0	0	0
合計コスト - AWS IoT Core 見積りツールに基づく	112.00 USD	1,123.00 USD	4,952.00 USD
Amazon Kinesis Firehose 配信ストリーム			
5 KB に切り上げたレコードサイズ	5	5	5
Firehose の ボリューム (KB) / 月	432000000	4320000000	21600000000
Firehose の ボリューム (GB) / 月	411.9873047	4119.873047	20599.36523
Firehose のコスト / 月	11.94763184	119.4763184	597.3815918
Amazon Dynamo DB			
RCU	1	1	1
WCU	10	10	10
サイズ (MB)	1	1	1
RCU コスト	0.0936	0.0936	0.0936
WCU コスト	4.68	4.68	4.68
サイズコスト	0	0	0
DynamoDB のコスト / 月	4.7736	4.7736	4.7736
AWS Key Management Service (KMS) コスト			
1 か月あたりレコード数	86400000	864000000	4320000000
暗号化リクエスト数 - 20,000 件無料	86380000	863980000	4319980000
暗号化コスト	259.14	2591.94	12959.94
KMS のコスト / 月	259.14	2591.94	12959.94

AWS Lambda			
呼び出し	59,715	597,149	2,985,745
所要時間 (ミリ秒)	16,496,242	164,692,419	824,812,095
メモリ (MB)	1536	1536	1536
メモリ所要時間 (GB/秒)	24,744.35	247,443.63	1,237,218.14
Lambda のコスト / 月	0.42	4.24	21.22
1 か月あたり予測合計コスト	388.28 USD	3,843.43 USD	18,535.32 USD

表 3. センサーデータ収集 - 予測コストの詳細

付録 A.2 – Ingest Transform Load (ITL) のストリーミング

Ingest Transform Load (ITL) のストリーミングの場合の 1 か月あたり予測コストの詳細を下の表 4 に示します。このサービスの設定は次のとおりです。

- Kinesis Firehose のバッファサイズは 100 MB です。
- Kinesis Firehose のバッファ間隔は 5 分 (300 秒) です。
- バッファされたレコードは GZIP で圧縮され S3 に保存されます。圧縮率は 1/4 であるものとします。

	スモール	ミディアム	ラージ
ピークレート (レコード/秒)	100	1000	5000
レコードサイズ (KB)	1	1	1
Amazon Kinesis Firehose			
ボリューム (GB) / 月 (注 1)	411.987	4,119.87	20,599.37
Kinesis のコスト / 月	11.95 USD	119.48 USD	597.38 USD
Amazon S3			
ソースレコードストレージ (GB)	21.02	210.22	1,051.08
変換済みレコードストレージ (GB)	21.02	210.22	1,051.08
PUT API コール (注 2)	17280	17280	84375
S3 のコスト / 月	2.47 USD	23.87 USD	119.35 USD
AWS Lambda			
呼び出し	59,715	597,149	2,985,745
所要時間 (ミリ秒)	16,496,242	164,962,419	824,812,095
関数メモリ (MB)	1536	1536	1536
メモリ所要時間 (GB - 秒)	24,744.36	247,443.63	1,237,218.14
Lambda のコスト / 月 (注 3)	0.42 USD	4.24 USD	21.22 USD
Amazon DynamoDB			
RCU (注 4)	50	50	50
DynamoDB のコスト / 月	4.68 USD	4.68 USD	4.68 USD
1 か月あたり合計コスト	18.11 USD	138.16 USD	672.06 USD

表 4. Ingest Transform Load (ITL) ストリーミング - 推定コストの詳細

注意:

1. Kinesis Firehose ではレコードサイズを直近の 5 KB に切り上げます。上記 3 つのシナリオではそれぞれ、1 KB のレコードを 5 KB に切り上げて 1 か月あたりのボリュームを算出しています。
2. S3 の PUT API コールの推定にあたっては、Firehose の配信ストリームで作成される S3 オブジェクト 1 つにつき 1 件の PUT コールがあるものと想定しています。レコードレートが低い場合、S3 オブジェクトの数は Firehose のバッファ所要時間 (5 分) によって決まります。レコードレートが高い場合、S3 オブジェクトの数は Firehose のバッファサイズ (100 MB) によって決まります。
3. AWS Lambda の無料利用枠には、1 か月あたり 100 万回の無料リクエストと、1 か月あたり 400,000 GB - 秒のコンピューティング時間が含まれます。上記で推定した 1 か月あたりのコストは、無料利用枠適用前のものです。
4. 上記で推定した DynamoDB の読み込みキャパシティーユニット (RCU) は、検索をメモリ内にキャッシュし、コンテナの再利用を活用した結果です。これは、このテーブルで必要とされる RCU 数が少なくなっていることを意味します。

付録 A.3 - リアルタイム分析

リアルタイム分析パターンの 1 か月あたり予測コストの詳細を下の表5 に示します。

	スモール	ミディアム	ラージ
ピークレート (メッセージ/秒)	100	1000	5000
レコードサイズ (KB)	1	1	1
1 日あたりレコード (数)	2880000	28800000	144000000
1 か月あたりレコード (数)	86400000	864000000	4320000000
1 か月あたりボリューム (KB)	86400000	864000000	4320000000
1 か月あたりボリューム (GB)	82.39746094	823.9746094	4119.873047
1 か月あたりボリューム (TB)	0.08046627	0.804662704	4.023313522
Amazon Kinesis Analytics			
1 日あたりピーク時間 (時間)	4	4	4
1 日あたり平均時間 (時間)	20	20	20
Kinesis Processing Unit (KPU)/ 時間 - ピーク	2	2	2
Kinesis Processing Unit (KPU)/ 時間 - 平均	1	1	1
Kinesis Analytics のコスト / 月	692.40 USD	692.40 USD	692.40 USD
Amazon Kinesis Firehose 配信ストリーム			
5 KB に切り上げたレコードサイズ	5	5	5
Firehose のボリューム (KB) / 月	432000000	4320000000	21600000000
Firehose のボリューム (GB) / 月	411.9873047	4119.873047	20599.36523
Kinesis Firehose のコスト / 月	11.94763184	119.4763184	597.3815918

(表は次頁へ続く)

	スモール	ミディアム	ラージ
Amazon S3			
サイズのみに基づく S3 PUT / 月	843.75	843.75	843.75
時間のみに基づく S3 PUT / 月	8640	8640	8640
期待される S3 PUT (サイズと時間の最大値)	8640	8640	8640
合計 PUT (ソースのバック アップ + 分析データ)	17280	17280	17280
圧縮済み分析データ (GB)	21.02150444	21.02	21.02
圧縮済みソースデータ (GB)	21.02	21.02	21.02
ソースレコードバックアップ	0.48346	0.48346	0.48346
PUT	0.0864	0.0864	0.0864
分析データレコード	0.483494602	0.48346	0.48346
S3 のコスト / 月	1.053354602	1.05332	1.05332
AWS Lambda			
呼び出し	59,715	597,149	2,985,745
所要時間 (ミリ秒)	16,496,242	164,692,419	824,812,095
メモリ (MB)	1536	1536	1536
メモリ所要時間 (GB/秒)	24,744.35	247,443.63	1,237,218.14
Lambda のコスト / 月	0.42	4.24	21.22
1 か月あたり予測合計コスト	705.82 USD	817.17 USD	1,312.05 USD

表 5.リアルタイム分析 - 推定コストの詳細

付録 B - パターンのデプロイおよびテスト

共通タスク

3 パターンの実装の詳細については、後続のセクションで説明します。各パターンのデプロイ、実行、テストは他のパターンと関係なく行えます。各パターンのデプロイのために、どの AWS リージョンにもデプロイできる AWS サーバーレスアプリケーションモデル (AWS SAM) テンプレートへのリンクを提供します。[AWS SAM](#) は、AWS CloudFormation の機能を強化して、サーバーレスアプリケーションで必要とされる Amazon API Gateway の API、AWS Lambda 関数、Amazon DynamoDB テーブルを定義するための簡単な構文を提供するものです。

次の GitHub 公開リポジトリから、3 つのパターンに対するソリューションをダウンロードできます。

<https://github.com/aws-samples/aws-serverless-stream-ingest-transform-load>
<https://github.com/aws-samples/aws-serverless-real-time-analytics>
<https://github.com/aws-labs/aws-serverless-sensor-data-collection>

アーティファクトの S3 バケットを作成または特定する

AWS サーバーレスアプリケーションモデル (SAM) を使用するには、コードとテンプレートのアーティファクトのアップロード先となる S3 バケットが必要です。適切なバケットが AWS アカウント内に既にある場合は、S3 バケット名を書き留めるだけで、このステップをスキップできます。新しいバケットを作成する場合は、次のステップに従います。

1. S3 コンソールにログインします。
2. **[バケットを作成する]** を選択し、バケット名を入力します。バケット名は必ず一意であるようにします。<ランダムな文字列>-stream-artifacts のような名前が考えられます。パターンをデプロイする AWS リージョンを選択します。
3. 後続のページで **[次へ]** を選択してデフォルトの設定内容を許可します。最後のページで、**[バケットを作成]** を選択してバケットを作成します。下の 3 パターンをデプロイするのにバケット名が必要になるため、書き留めておきます。

Kinesis Data Generator 用に Amazon Cognito ユーザーを作成する

病院のデバイスのシミュレーションを行って Streaming Ingest Transform Load (ITL) パターンとリアルタイム分析パターンをテストするには、Amazon Kinesis Data Generator (KDG) ツールを使用します。KDG ツールについての詳細は、こちらの[ブログ投稿](#)をご参照ください。



Amazon Kinesis Data Generator には[こちら](#)からアクセスしていただけます。KDG へのログインに使用する Cognito ユーザー名およびパスワードを作成するには、[Help] メニューをクリックして指示に従います。

付録 B.1 - センサーデータ収集

このセクションでは、ユースケースを AWS アカウントにデプロイしその実行とテストを行う方法を説明します。

SAM テンプレートを確認する

任意のエディタで「SAM-For-SesorDataCollection.yaml」というファイルを開き、サーバーレスアプリケーションモデル (SAM) テンプレートを確認します。Notepad++ を使うと JSON ファイルをうまく表示できます。

このテンプレートによって、AWS アカウント内に次のリソースが作成されます。

- 匿名化レコードの保存に使用する S3 バケット。
- ZIP ファイル内に圧縮され S3 バケットに保存された匿名化レコードをバッファおよび収集するのに使用する、Firehose 配信ストリームと関連する IAM ロール。
- 保護された健康情報 (PHI) および個人識別情報 (PII) のデータを削除することで受信メッセージの非特定化を行う AWS Lambda 関数。またこの関数により、保護された健康情報 (PHI) および個人識別情報 (PII) のデータは、クロスリファレンス用の PatientID と共に DynamoDB に保存されます。保護された健康情報 (PHI) および個人識別情報 (PII) のデータは、AWS の KMS キーを使用して暗号化されます。
- ユースケースに対する病院デバイスのシミュレーションを実行する AWS Lambda 関数。Lambda 関数を使用することで、センサーシミュレーションデータが生成され、IoT MQTT のトピックに発行されます。
- 暗号化済みクロスリファレンスデータの患者 ID、タイムスタンプ、患者の名前、患者の生年月日を保存する DynamoDB テーブル。

パッケージおよびデプロイ

次の手順を実行して、センサーデータ収集のモデルのパッケージおよびデプロイを行います。

1. [こちら](#)の GitHub フォルダからファイルをダウンロードし、お使いのローカルマシンにコピーします。ローカルマシンに次のファイルがあることを確認します。
 - 1.1 DeIdentification.zip
 - 1.2 PublishIoTData.zip
 - 1.3 SAM-For-SesorDataCollection.yaml
 - 1.4 deployer-sensordatacollection.sh

2. ソリューションのデプロイ先の AWS リージョンで、[**S3 Deployment Bucket**] を作成します。S3 バケット名を書き留めておきます。S3 バケット名は後で必要となります。
3. 次の Lambda コードの zip ファイルを、ローカルマシンから手順 2 で作成した [**S3 Deployment Bucket**] にアップロードします。
 - 3.1 DeIdentification.zip
 - 3.2 PublishIoTData.zip
4. AWS マネジメントコンソールで、CloudFormation テンプレートを実行するために使用する EC2 Linux インスタンスを起動します。ソリューションのデプロイ先の AWS リージョンで、Amazon Linux AMI 2018.03.0 (HVM),SSD Volume Type (t2.macro) ec2 というタイプの EC2 インスタンスを起動します。インスタンスに SSH アクセスができることを確認してください。EC2 インスタンスを起動し SSH アクセスをする方法の詳細については、<https://aws.amazon.com/ec2/getting-started/> を参照してください。
5. お使いのローカルマシンで、テキストエディタを使用して `deployer-sensordatacollection.sh` ファイルを開き、**PLACE_HOLDER** として示された 3 つの変数、すなわち、**S3ProcessedDataOutputBucket** (処理済み出力データが保存される S3 バケット名)、**LambdaCodeUriBucket** (手順 2 で作成し Lambda コードファイルをアップロードした S3 バケット名)、およびソリューションのデプロイ先の AWS リージョンに対する環境変数である **REGION** を更新します。`deployer-sensordatacollection.sh` ファイルを保存します。
6. 起動した EC2 インスタンスの状態が実行中になったら、SSH を使用して EC2 Linux にログインします。`/home/ec2-user/` の下に `samdeploy` という名のフォルダを作成します。`/home/ec2-user/samdeploy` のフォルダに次のファイルをアップロードします。
 - 5.1 SAM-For-SesorDataCollection.yaml
 - 5.2 `deployer-sensordatacollection.sh`
7. EC2 インスタンスでディレクトリを `/home/ec2-user/samdeploy` に変更します。次に、パッケージとデプロイと呼ばれる 2 つの CloudFormation CLI コマンドを実行します。どちらのステップも 1 つのスクリプトファイル、**deployer-sensordatacollection.sh** にあります。このスクリプトファイルを確認してください。コマンドプロンプトで次のコマンドを実行すれば、SAM テンプレートのパッケージとデプロイが実行できます。

```
$ sh ./deployer-sensordatacollection.sh
```



スタックの詳細を表示する

CloudFormation コンソールにログインすることで、スタック作成の進行状況を表示することができます。必ず、スタックをデプロイした AWS リージョンを選択してください。スタックのリストで **[SensorDataCollectionStack]** という名のスタックを探し、そのイベントタブを選択してページを更新し、リソースの作成の進捗を表示します。

スタックの作成には 3~5 分を要します。スタックの状態は、すべてのリソースが正常に作成されると、**CREATE_COMPLETE** に変わります。

パイプラインをテストする

[SensorDataCollectionStack] には、PublishToIoT という名の **IoT デバイスシミュレーター Lambda 関数**があります。この Lambda 関数は AWS CloudWatch のイベントルールでトリガーされます。このイベントルールで Lambda 関数が 5 分間隔で呼び出されます。Lambda 関数は、前述のパターンにマッチするセンサーデバイスのメッセージをシミュレートして生成し、これを MQTT のトピックに発行します。

この関数は、JSON 文字列を **SimulatorConfig** という入力として受け取り、呼び出しごとに生成するメッセージ数を設定します。今の例では Lambda 関数の呼び出しごとに 10 のメッセージが設定されます。Lambda 関数への入力パラメータは、JSON 文字列 `{"NumberOfMsgs": "10"}` に設定されます。

スタックが正常にデプロイされるとすぐにソリューションが開始します。次の事項が確認できます。

1. CloudWatch の Event/Rule は 5 分ごとにデバイスシミュレーターに Lambda 関数を呼び出します。Lambda 関数はデフォルトで、呼び出しごとに 10 のセンサーデータを生成し、これを IoT トピックの「LifeSupportDevice/Sensor」に発行するよう設定されています。
2. 処理された (PHI と PII を含まない) データは、**[S3 Processed Data Bucket]** に表示されます。
3. DynamoDB コンソールでは、**[PatientReferenceTable]** という表で、PatientID、PatientName、PatientDOB からなるクロスリファレンスデータが表示されます。

パターンのテストを停止するには、CloudWatch コンソールに移動して、SensorDataCollectionStack-IoTDeviceSimulatorFunct-XXXXXXX という名の Events/Rule を無効化します。

補足:

このホワイトペーパーでは触れませんでしたでしたが、AWS ソリューショングループのチームは堅牢な IoT デバイスシミュレーターを用意しています。これにより、デバイスの統合と IoT のバックエンドサービスをテストすることが容易になります。このソリューションはウェブベースのグラフィカルユーザーインターフェイス (GUI) コンソールを備えており、物理デバイスの設定や管理、または時間を要するスクリプトの開発を行うことなく、何百もの仮想接続デバイスを作成しシミュレートできます。詳細については、<https://aws.amazon.com/answers/iot/iot-device-simulator/> を参照してください。

ただし、単純なパターンでは、このホワイトペーパーで紹介したように、CloudWatch Event/Rule で呼び出される IoT デバイスシミュレーター Lambda関数を使用することもあります。デフォルトでは、5 分ごとにトリガーされることになっています。

リソースをクリーンアップする

このパターンをテストした後は、作成したリソースに対して料金が発生しないようにこれらのリソースを削除してクリーンアップできます。

1. パターンのデプロイ先の AWS リージョンにある CloudWatch コンソールの [Events/Rules] で、SensorDataCollectionStack-IoTDeviceSimmulatorFunc-XXXXXXX ルールを無効にします。
2. S3 コンソールで、出力に [**S3 Processed Data Bucket**] を選択して、[空にする] を選択します。
3. CloudFormation コンソールで [**SensorDataCollectionStack**] スタックを選択し、[**スタックの削除**] を選択します。
4. 最後に、CloudFormation テンプレートを実行してソリューションをデプロイするために作成した EC2 Linux インスタンスを EC2 コンソールで終了します。

付録 B.2 – Ingest Transform Load (ITL) のストリーミング

このセクションでは、パターンを AWS アカウントにデプロイして変換関数をテストし、パイプラインのパフォーマンスをモニタリングする方法を説明します。

SAM テンプレートを確認する

「streaming_ingest_transform_load.template」ファイルで サーバーレスアプリケーションモデル (SAM) テンプレートを確認します。

このテンプレートによって、AWS アカウント内に次のリソースが作成されます。

- 変換済みレコードと Kinesis Firehose からのソースレコードを保存するために使用する S3 バケット。
- レコードを取り込むために使用する Firehose 配信ストリームおよびそれに関連付けられた IAM ロール。
- 上述の変換とエンリッチメントを実行する AWS Lambda 関数。
- デバイスの詳細が保存され、変換関数が検索を行う DynamoDB テーブル。
- サンプルデバイスの詳細レコードを DynamoDB テーブルに挿入する AWS Lambda 関数。この関数はカスタムの CloudFormation リソースとして、スタックを作成するときに一度だけ呼び出され、テーブルを配置します。
- 処理中のパイプラインのモニタリングを容易にする CloudWatch ダッシュボード。

パッケージおよびデプロイ

このステップでは、CloudFormation パッケージコマンドを使用して、ローカルのアーティファクトを前のステップで選択または作成したアーティファクト S3 バケットにアップロードします。また、ローカルのアーティファクトへの参照がパッケージコマンドによるアーティファクトのアップロード先の S3 の場所に置き換えられた後、このコマンドによって SAM テンプレートが返されます。

この後、CloudFormation デプロイコマンドを使用して、スタックとそれに関連付けられるリソースを作成します。

上記の両ステップとも、GitHub リポジトリにある 1 つのスクリプトデプロイヤーに含まれています。このスクリプトを実行する前に、アーティファクトの S3 バケット名とリージョンをそのスクリプトで設定する必要があります。任意のテキストエディ

タでスクリプトを編集し、PLACE_HOLDER の S3 バケットとリージョンの名前を前のセクションのものから置き換えます。ファイルを保存します。

次のコマンドを実行すれば、SAM テンプレートのパッケージとデプロイが実行できます。

```
$ sh ./deployer.sh
```

スタックの詳細を表示する

CloudFormation コンソールにログインすることで、スタック作成の進行状況を表示することができます。必ず、スタックをデプロイした AWS リージョンを選択してください。スタックのリストで **[StreamingITL]** という名のスタックを探し、そのイベントタブを選択してページを更新し、リソースの作成の進行状況を表示します。

スタックの作成には 3~5 分を要します。スタックの状態は、すべてのリソースが正常に作成されると、**CREATE_COMPLETE** に変わります。

パイプラインをテストする

パイプラインをテストするには、次の手順を実行します。

1. CloudFormation コンソールにログインし、既述の「[Kinesis Data Generator 用に Amazon Cognito ユーザーを作成する](#)」で作成した Kinesis Data Generator Cognito ユーザーをスタックから検索します。
2. **[出力]** タブを選択し、**[KinesisDataGeneratorUrl]** キーの値をクリックします。
3. Cognito User CloudFormation スタックを作成したときに使用した**ユーザー名とパスワード**でログインします。
4. Kinesis Data Generator で、サーバーレスアプリケーションリソースを作成したリージョンを選択し、ドロップダウンから **[IngestStream]** 配信ストリームを選択します。
5. 1 秒あたりのレコード数を 100 に設定して最初のトラフィックシナリオをテストします。
6. テストデータを生成するには、レコードのテンプレートを次のように設定します。

```
{
  "timestamp" : "{{date.now("x")}}",
  "device_id" : "device{{helpers.replaceSymbolWithNumber("####")}}",
  "patient_id" : "patient{{helpers.replaceSymbolWithNumber("####")}}",
  "temperature" : "{{random.number({"min":96,"max":104})}}",
  "pulse" : "{{random.number({"min":60,"max":120})}}",
  "oxygen_percent" : "{{random.number(100)}}",
  "systolic" : "{{random.number({"min":40,"max":120})}}",
  "diastolic" : "{{random.number({"min":40,"max":120})}}",
  "text" : "{{lorem.sentence(140)}}"
}
```

テストレコードにこのシナリオに必要な 1 KBの容量があることを確認するため、テンプレートの **[text]** フィールドを使用しています。

7. **[Send Data]** を選択して、設定したレートで生成データをKinesis Firehose Stream に送信します。

パイプラインをモニタリングする

S3 でパイプラインのパフォーマンスをモニタリングし、結果のオブジェクトを確認するには、次の手順を実行します。

1. CloudWatch コンソールに切り替え、左側のメニューでダッシュボードを選択します。
2. **[StreamingITL]** というダッシュボードを選択します。

3. ダッシュボードで、Lambda、Kinesis Firehose、DynamoDB のメトリクスを表示します。期間を選択して目的の期間の表示を拡大します。

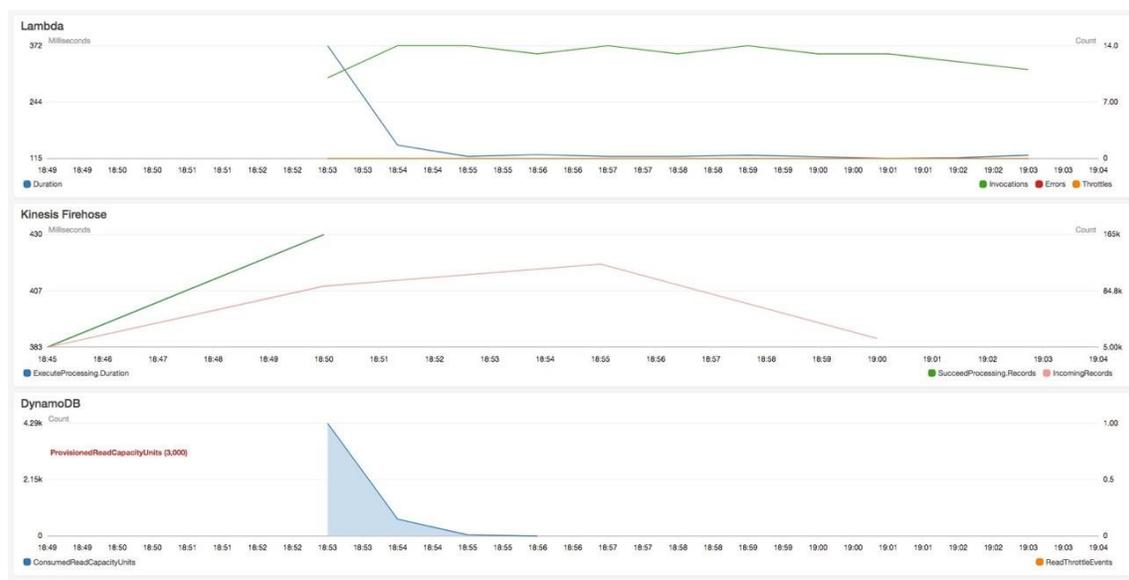


図 7.Streaming ITL 用の CloudWatch ダッシュボード

4. 5～8分後、変換されたレコードが **transformed/** というプレフィックスの出力 S3 バケットに表示されます。
5. S3 からサンプルオブジェクトをダウンロードしてその内容を検証します。なお、メモリ容量とデータ転送量の削減のため、オブジェクトは GZIP 形式に圧縮されて保存されています。
6. 変換されたレコードには、タイムスタンプ文字列、デバイスモデル、および製造者が、人の読める形で含まれていることを確認します。これらは、内容が増強されたフィールドで、DynamoDB テーブルからの検索に使用します。
7. 未変換のソースレコードのコピーも **source_records/** というプレフィックスの同じバケットに配信されていることを確認します。

パイプラインが最初のトラフィックシナリオで正常に機能していることが確認できたら、メッセージのレートを毎秒 1,000 リクエストに増加し、その後毎秒 5,000 リクエストに増加します。

リソースをクリーンアップする

このパターンをテストした後は、作成したリソースに対して料金が発生しないようにこれらのリソースを削除してクリーンアップできます。



1. Kinesis Data Generator からのデータ送信を停止します。
2. S3 コンソールで、[output S3 bucket] を選択して、[空にする] を選択します。
3. CloudFormation コンソールで [StreamingITL] スタックを選択し、[スタックを削除] を選択します。

付録 B.3 - リアルタイム分析

このセクションでは、ユースケースを AWS アカウントにデプロイしその実行とテストを行う方法を説明します。

SAM テンプレートを確認する

任意のテキストエディタで「SAM-For-RealTimeAnalytics.yaml」というファイルを開き、サーバーレスアプリケーションモデル (SAM) テンプレートを確認します。Notepad++ を使うと JSON ファイルをうまく表示できます。

このテンプレートによって、AWS アカウント内に次のリソースが作成されます。

- 異常なスコアを含む Real-Time Analytics のレコードを保存するために使用する S3 バケット (**S3ProcessedDataOutputBucket**)。
- Kinesis Analytic サービスへの入カストリームとして使用する Firehose Delivery Stream とそれに関連付けられた IAM ロール。
- **DeviceDataAnalytics** という名の Kinesis Analytics Application。これには 1 つの入カストリーム (Firehose Delivery Stream)、アプリケーションコード (SQL Statements)、
- Lambda 関数としての配信先接続 (Kinesis Analytics Application Output)、Kinesis Firehose 配信ストリームとしての第二の配信先接続 (Kinesis Analytics Output) が含まれます。
- **publishtomanufacturer** という名の SNS トピックとその SNS トピックのメールサブスクリプション。デプロイメントスクリプトの **deployer-realtimetypeanalytics.sh** でメール設定ができます。メール設定のための変数は、デプロイメントスクリプトで **NotificationEmailAddress** と呼ばれています。
- レコードを受信し SNS トピックに発行する Analytics Stream から受信したデータレコードセット要求する AWS Lambda 関数。このトピックでは、異常なスコアが (この場合は Lambda 関数の環境変数内で) 定義済みのしきい値より高くなっています。
- Kinesis Analytics Application が作成された直後に、Kinesis Analytics Application の **DeviceDataAnalytics** を開始するために使用される、

KinesisAnalyticsHelper という名の第二の AWS Lambda 関数。

- Analytics Destination Stream のレコードを統合し、そのレコードをバッファリングして圧縮し、その zip 済みファイルを S3 バケット (**S3ProcessedDataOutputBucket**) に保存する Kinesis Firehose Delivery Stream。

パッケージおよびデプロイ

次の手順を実行して、Real-Time Analytics のモデルのパッケージおよびデプロイを行います。

1. [こちらの](#) GitHub フォルダからファイルをダウンロードし、お使いのローカルマシンにコピーします。ローカルマシンに次のファイルがあることを確認します。

- 1.1 KinesisAnalyticsOutputToSNS.zip
- 1.2 SAM-For-RealTimeAnalytics.yaml
- 1.3 deployer-realttimeanalytics.sh

2. ソリューションのデプロイ先の AWS リージョンで、[**S3 Deployment Bucket**] を作成します。S3 バケット名を書き留めておきます。S3 バケット名は後で必要となります。

3. 次の Lambda コードの zip ファイルを、ローカルマシンから手順 2 で作成した [**S3 Deployment Bucket**] にアップロードします。

- 3.1 KinesisAnalyticsOutputToSNS.zip

4. AWS マネジメントコンソールで、CloudFormation テンプレートを実行するために使用する EC2 Linux インスタンスを起動します。ソリューションのデプロイ先の AWS リージョンで、Amazon Linux AMI 2018.03.0 (HVM), SSD Volume Type (t2.micro) ec2 というタイプの EC2 インスタンスを起動します。インスタンスに SSH アクセスができることを確認してください。EC2 インスタンスを起動し、SSH アクセスをする方法の詳細については、<https://aws.amazon.com/ec2/getting-started/> を参照してください。

5. ローカルマシンで任意のテキストエディタを使用して deployer-realttimeanalytics.sh ファイルを開き、**PLACE HOLDER** と記された 5 つの変数を更新します。その変数は、**S3ProcessedDataOutputBucket** (Processed Output Data が保存される S3 バケット名)、**NotificationEmailAddress** (異常なスコアがしきい値を超えた場合に通知を受信するために指定したメールアドレス)、**AnomalyThresholdScore** (Lambda 関数が通知を送信するレコードを指定するために使用するしきい値)、**LambdaCodeUriBucket** (手順 2 で作成した S3 バケット名と Lambda コードファイル)、ソリューションのデプロイ先の AWS リージョンへの変数である **REGION** です。deployer-sensordatacollection.sh file を保存します。

6. 起動した EC2 インスタンスの状態が実行中になったら、SSH を使用して EC2 Linux にログインします。/home/ec2-user/ の下に samdeploy という名の



フォルダを作成します。/home/ec2-user/samdeploy のフォルダに次のファイルをアップロードします。

6.1 SAM-For-RealTimeAnalytics.yaml

6.2 deployer-realtimetypeanalytics.sh

7. EC2 インスタンスでディレクトリを /home/ec2-user/samdeploy に変更します。次に、パッケージとデプロイと呼ばれる 2 つの CloudFormation CLI コマンドを実行します。どちらのステップも 1 つのスクリプトファイル、**deployer-realtimetypeanalytics.sh** にあります。このスクリプトファイルを確認してください。コマンドプロンプトで次のコマンドを実行すれば、SAM テンプレートのパッケージとデプロイが実行できます。

```
$ sudo yum install dos2unix
$ dos2unix deployer-realtimetypeanalytics.sh
$ sh ./ deployer-realtimetypeanalytics.sh
```

8. パターンのデプロイメントの一部として、メール (手順 5 で指定したメールアドレス) サブスクリプションが SNS トピックに設定されます。メールの受信ボックスで、**サブスクリプションの確認**を依頼するメールがあることを確認してください。メールを開き、サブスクリプションの検証を確認します。その後は、デバイスデータレコードが指定したしきい値を超えたことを通知するメールを受信するようになります。

スタックの詳細を表示する

CloudFormation コンソールにログインすることで、スタック作成の進行状況を表示することができます。必ず、スタックをデプロイした AWS リージョンを選択してください。スタックのリストで [DeviceDataRealTimeAnalyticsStack] という名のスタックを探し、そのイベントタブを選択してページを更新し、リソースの作成の進行状況を表示します。

スタックの作成には 3~5 分を要します。スタックの状態は、すべてのリソースが正常に作成されると、**CREATE_COMPLETE** に変わります。

パイプラインをテストする

このパターンをテストするには、ツールに Kinesis Data Generator (KDG) を使用して、テストデータを生成し発行できます。ホワイトペーパーの「[Kinesis Data Generator 用に Amazon Cognito ユーザーを作成する](#)」のセクションを参照してください。設定時に作成したユーザー名とパスワードを使用して KDG ツールにログインします。次の情報を入力します。

1. リージョン: 選択するリージョンは、**DeviceDataRealTimeAnalyticsStack のインストール先**です。



2. ストリーム/配信ストリーム: 選択する配信ストリームは、**DeviceDataInputDeliveryStream** です。
3. 1 秒あたりのレコード数: 病院のデバイスデータをシミュレートするために生成/発行するレコードのレートを入力します。
4. レコードテンプレート: KDG はレコードテンプレートを使用してレコードフィールドごとにランダムなデータを生成します。次の JSON テンプレートを使用して Kinesis 配信ストリーム **DeviceDataInputDeliveryStream** に発行されるレコードを生成します。

```
{
  "timestamp" : "{{date.now("x")}}",
  "device_id" : "device{{helpers.replaceSymbolWithNumber("####")}}",
  "patient_id" : "patient{{helpers.replaceSymbolWithNumber("#####")}}",
  "temperature" : "{{random.number({"min":96,"max":104})}}",
  "pulse" : "{{random.number({"min":60,"max":120})}}",
  "oxygen_percent" : "{{random.number(100)}}",
  "systolic" : "{{random.number({"min":40,"max":120})}}",
  "diastolic" : "{{random.number({"min":40,"max":120})}}",
  "manufacturer" : "Manufacturer{{helpers.replaceSymbolWithNumber("#")}}",
  "model" : "Model {{helpers.replaceSymbolWithNumber("#")}}"
```

Amazon Kinesis Data Generator

Region: ap-south-1

Stream/delivery stream: No destinations found in this region

Records per second: 100

Record template: RealTimeAnalytics

```
{
  "timestamp" : "{{date.now("x")}}",
  "device_id" : "device{{helpers.replaceSymbolWithNumber("####")}}",
  "patient_id" : "patient{{helpers.replaceSymbolWithNumber("#####")}}",
  "temperature" : "{{random.number({"min":96,"max":104})}}",
  "pulse" : "{{random.number({"min":60,"max":120})}}",
  "oxygen_percent" : "{{random.number(100)}}",
  "systolic" : "{{random.number({"min":40,"max":120})}}",
  "diastolic" : "{{random.number({"min":40,"max":120})}}",
  "manufacturer" : "Manufacturer {{helpers.replaceSymbolWithNumber("#")}}",
  "model" : "Model {{helpers.replaceSymbolWithNumber("#")}}"
```

Send data Test template



RealTime Analytics アプリケーションを実行するには、KDG ツールの下部にある **Send Data** ボタンをクリックします。KDG がデバイスデータレコードを Kinesis Delivery Stream に出し始めると、レコードは Kinesis Analytics Application に流入します。アプリケーションコードは、ストリーミングデータを分析し、アルゴリズムを適用し各行に異常なスコアを生成します。Kinesis Analytics コンソールでこのデータストリーミングを表示できます。次の図は、データストリームの例です。

The screenshot shows the Kinesis Analytics console interface. At the top, there are tabs for 'Source data', 'Real-time analytics', and 'Destination'. The 'Real-time analytics' tab is active, showing 'In-application streams' with a 'Pause results' button and a note: 'New results are added every 2-10 seconds. The results below are sampled.' Below this, there is a search bar for 'Filter by column name' and a table of data. The table has columns: id, manufacturer, model, temperature, pulse, oxygen_percent, systolic, diastolic, and ANOMALY_SCORE. The data rows show various manufacturer and model combinations with their respective vitals and an anomaly score.

id	manufacturer	model	temperature	pulse	oxygen_percent	systolic	diastolic	ANOMALY_SCORE
78	Manufacturer 0	Model 66	101	88	80	100	94	0.766767675038771
33	Manufacturer 4	Model 26	97	104	43	82	88	0.6821320021826388
13	Manufacturer 6	Model 05	100	90	27	96	51	0.7744066963755888
31	Manufacturer 7	Model 27	102	71	62	42	66	0.8689001985092837
04	Manufacturer 8	Model 18	97	97	38	65	75	0.7083369698811833
85	Manufacturer 7	Model 63	102	65	56	63	113	0.8633016189792241
44	Manufacturer 0	Model 42	104	75	72	85	65	0.7340417674859959
76	Manufacturer 2	Model 30	100	113	78	49	74	0.76959000273048
22	Manufacturer 5	Model 81	100	95	88	47	63	0.8192073984234418
90	Manufacturer 5	Model 27	96	90	76	92	79	0.7468301516909267
33	Manufacturer 8	Model 50	102	90	100	117	50	1.061288391971577

Kinesis Analytics Application は、2 つの送信先接続で設定されています。第一の送信先コネクタ (または出力) は Lambda 関数です。Lambda 関数は、Application DESTINATION_SQL_STREAM_001 が配信するレコードのバッチに反復適用され、異常なスコアフィールドにレコードを要求します。異常なスコアが、Lambda 関数の環境変数 **ANOMALY_THRESHOLD_SCORE** で定義されたしきい値を超える場合、Lambda 関数は、そのレコードを **publishtomanufacturer** という名の Simple Notification Service (SNS) トピックに発行します。

第二の送信先接続は、**DeviceDataOutputDeliveryStream** という Kinesis Firehose Delivery Stream に設定されています。配信システムはレコードをバッファリングし、バッファリングされたレコードを zip ファイルに圧縮してから、**S3ProcessedDataOutputBucket** という名の S3 バケットに保存します。

次の事項が確認できます。

1. お使いの (デプロイメントスクリプトで指定した) メールを受信ボックスの最初のメールで、指定のしきい値を超えた異常なスコアがあるデバイスデータレコードを受信します。
2. AWS Kinesis Data Analytics コンソールで **DeviceDataAnalytics** というアプリケーションを選択し、下部にあるアプリケーションの詳細ボタンをク

リックします。これにより、**DeviceDataAnalytics** アプリケーションの詳細ページに移動します。[Real-Time Analytics] のページの中央にある「Go to the SQL Results」ボタンをクリックします。[Real-Time Analytics] ページでタブを使用して、Source Data、Real-Time Analytics Data、Destination Data を確認します。

3. 異常なスコアのあるレコードは [S3 Processed Data Bucket] に保存されています。この異常スコアを含むレコードを確認します。

パターンのテストを停止するには、KDG ツールを実行しているブラウザに移動し、「Stop Sending Data to Kinesis」ボタンをクリックします。

リソースをクリーンアップする

このパターンをテストした後は、作成したリソースに対して料金が発生しないようにこれらのリソースを削除してクリーンアップできます。

1. KDG ツールを起動したブラウザに戻り、[停止] ボタンをクリックします。ツールは追加データを、入力 Kinesis ストリームに送信するのを停止します。
2. S3 コンソールで、出力に [S3 Processed Data Bucket] を選択して、[空にする] を選択します。
3. Kinesis コンソールで Kinesis データアプリケーション [DeviceDataAnalytics] を選択します。
4. CloudFormation コンソールで [SensorDataCollectionStack] スタックを選択し、[スタックの削除] をクリックします。
5. 最後に、CloudFormation テンプレートを実行してソリューションをデプロイするために作成した EC2 Linux インスタンスを EC2 コンソールで終了します。