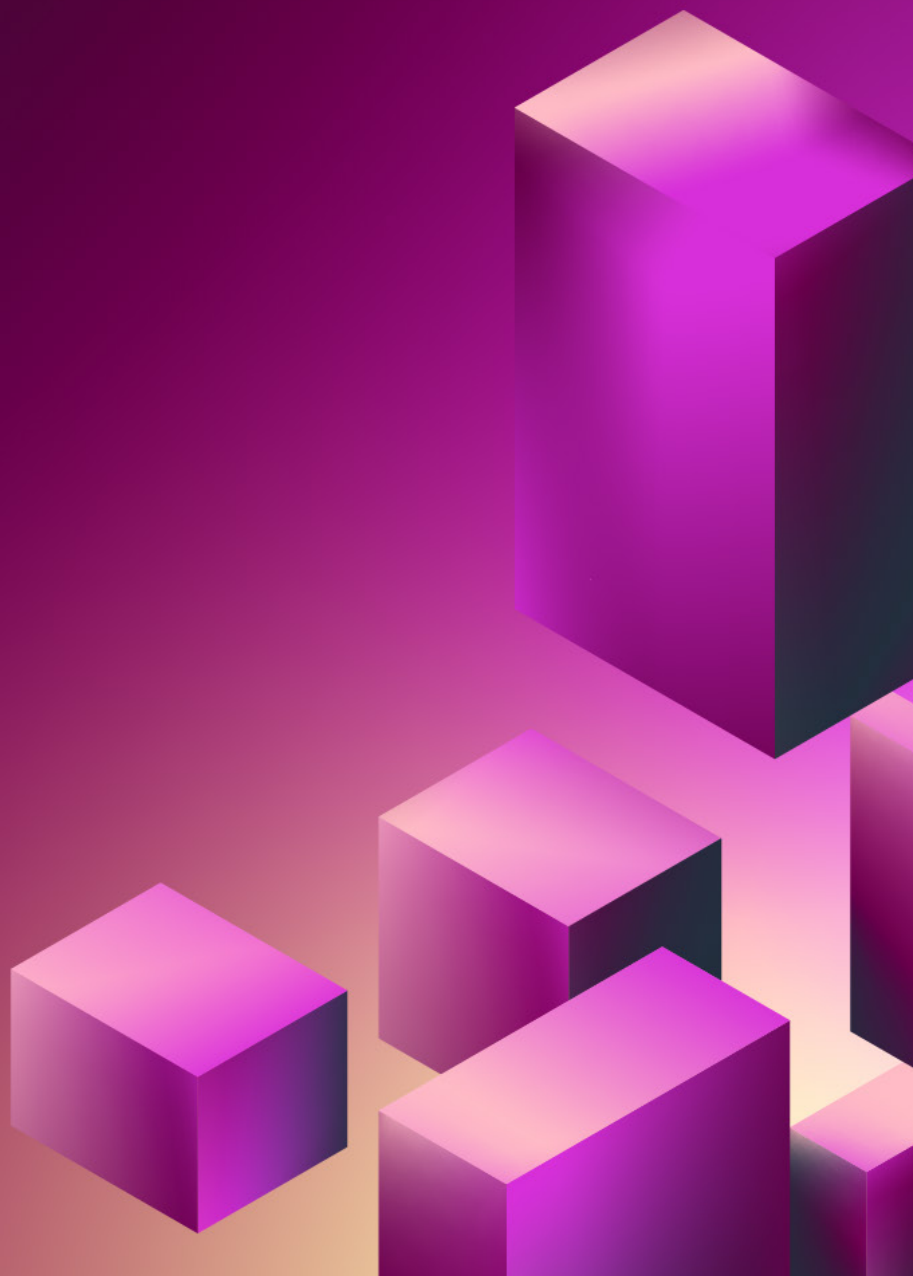
AWS でイベント駆動型アーキテクチャを構築する



要約

イベントは至るところで発生します。新しいアカウントが作成された、ショッピングカートに商品が入った、財務書類が提出された、医療データセットがアップロードされたなど、どれもイベントです。イベント駆動型アーキテクチャにおいて、イベントはアプリケーションの中心にあり、統合システムと開発チーム間のコミュニケーションを推進します。

このガイドでは、イベント駆動型アーキテクチャの主要な概念とパターンを紹介し、それらを実装するために一般的に使用される Amazon Web Services (AWS) のソリューションとサービスを確認します。さらに、イベントスキーマの設計から冪等性の処理に至るまで、イベント駆動型アーキテクチャの構築に関するベストプラクティスを紹介します。

セクション 1

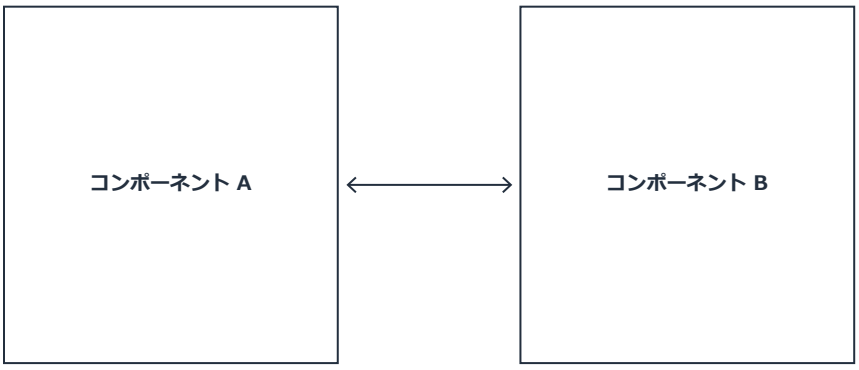
イベント駆動型アーキテクチャの概要と主要な概念

密結合と疎結合

結合とは、アプリケーションの各コンポーネントの相互依存の尺度です。システム間の結合にはさまざまな形態があります。

- データ形式の依存関係 (バイナリ、XML、JSON)
- 時間的な依存関係 (コンポーネントを呼び出す順序)
- 技術的な依存関係 (Java、C++、Python)

密結合システムは、アプリケーションのコンポーネントの数が少ない場合や、単独のチームや開発者がアプリケーション全体を所有する場合には効果的です。コンポーネント間の結合が密になるにつれて、特定のコンポーネントの変更や運用に関する問題が他のコンポーネントに波及する可能性が高まります。

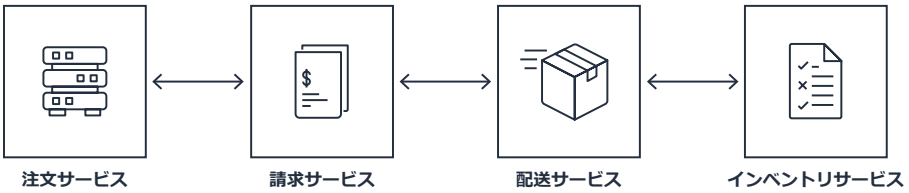


多数のチームが関与する複雑なシステムの場合、密結合はいくつかの難点があります。

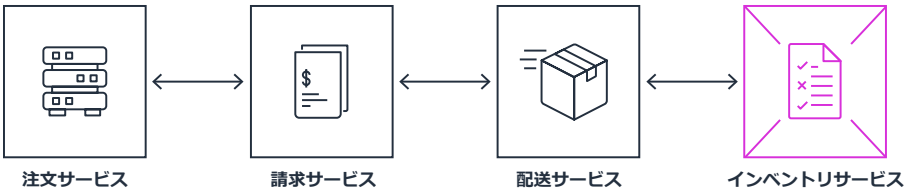
コンポーネントが緊密に相互依存していると、他のコンポーネントに影響を与えることなく単一のコンポーネントにのみ独立した変更を行うことは、困難でリスクがあります。その結果、開発プロセスが遅れたり、機能の速度が低下したりする可能性があります。

密結合のコンポーネントは、アプリケーションのスケーラビリティと可用性にも影響を与える可能性があります。2 つのコンポーネントが互いの同期応答に依存していると、一方の障害がもう一方の障害を引き起こします。これらの障害によりアプリケーション全体の耐障害性が低下します。

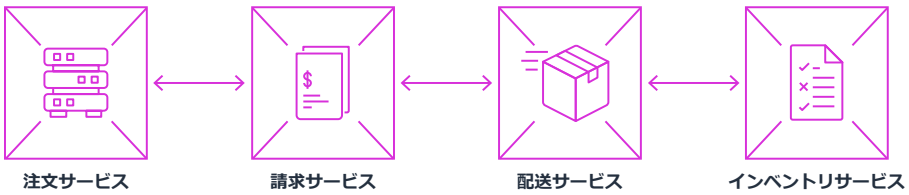
例えば、e コマースアプリケーションには複数のサービス (注文、請求、配送、インベントリ) があり、これらのサービスに対して一連の同期呼び出しを行います。



これらのサービスのいずれかで障害が発生すると ...



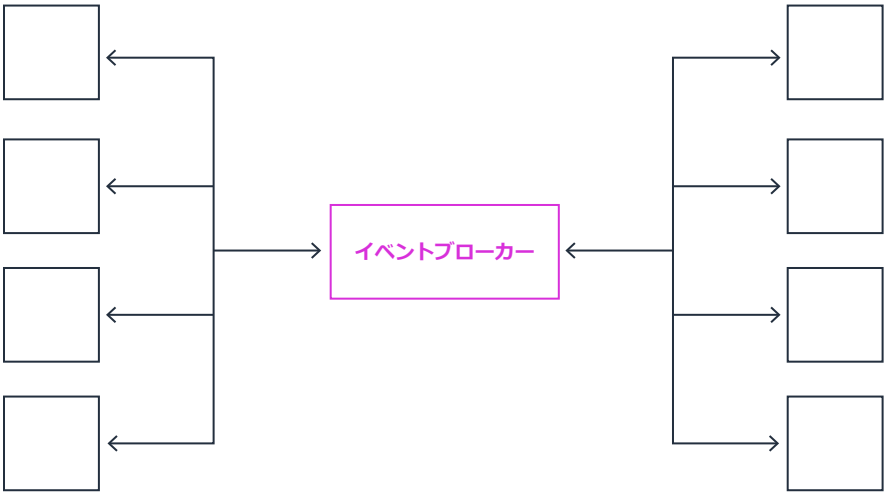
... 他のすべてのサービスに影響が及んでしまうのです。



結合の削減とは、コンポーネント間の相互依存性と各コンポーネントの相互認識の必要性を削減することです。

イベント駆動型アーキテクチャは、イベントを使った非同期通信によって疎結合を実現します。疎結合が実現されるのは、コンポーネントが相手からの応答を必要としない場合です。その代わり、最初のコンポーネントがイベントを送信し、次のコンポーネントで遅延や障害が発生した場合でも、影響を受けずに続行できます。

イベントを使って通信する場合、コンポーネントが認識する必要があるのは、独立したイベントだけです。送信元のコンポーネントやその他のコンポーネントの動作 (エラー処理、再試行ロジックなど) を認識する必要はありません。イベントの形式が同じである限り、ある 1 つのコンポーネントに行う変更がその他のコンポーネントに影響することはありません。これにより、アプリケーションに変更を加える際のリスクが低下します。非同期イベントがコンポーネントをお互いに抽象化すると、複雑なアプリケーションのレジリエンスとアクセシビリティが向上します。



イベント駆動型アーキテクチャの主な利点

非同期システムには独特の特性と考慮事項があるため、イベント駆動型アーキテクチャを構築する際には考え方の切り替えが求められます。それでも、このアーキテクチャスタイルは複雑なアプリケーションに次のような大きなメリットをもたらします。イベント駆動型アーキテクチャにより、次のことが可能になります。

- **新しい機能を独立して構築しデプロイする**
  個々のサービスを担当する開発チーム間の依存関係が削減されます。中央のイベントブローカーを介してイベントを発行および消費できるため、開発者は独立して機能を構築およびリリースできます。他のチームにコードを変更してもらわないと、統合を円滑に進められないといった懸念もありません。単一のサービスに対する変更が他のサービスに影響を与えるリスクが低下します。
- **イベントを使用して、既存のアプリケーションを変更せずに新しい機能を構築する**
  イベント駆動型アーキテクチャは、コンポーネントがイベントを送出するため、拡張が簡単です。新しいサービスは、既存のアプリケーションや開発チームに影響を与えることなく、既に発行されているイベントをサブスクライブできます。そのため、企業は中断のリスクを抑えながら、新しい機能や製品をより速いペースで構築できます。
- **耐障害性とスケーラビリティを強化する**
  非同期イベントを利用することで、上流のシステムは下流のシステムに送信するイベントの量をバッファリングできます。アプリケーションはピークに対応できるようスケールできるようになり、アプリケーションのどの部分にも負担がかかりません。イベント駆動型アーキテクチャでは、プロデューサーは下流のコンシューマーのアクティビティを認識しないため、障害が発生しても気付きません。

イベント駆動型アーキテクチャの主要な概念

イベントは、状態が変化したことを示すシグナルです。例えば、ショッピングカートに商品が入った、クレジットカードの申請が提出されたなどです。イベントは過去に発生したもの(「OrderCreated」(注文が作成された)、「ApplicationSubmitted」(申請が提出された) など) であり、不変つまり変更できません。コンポーネント間で変更を同期する必要がなくなるため、分散システムにおいて便利です。

イベントは観察されるものであり、指示されるものではありません。イベントを送出するコンポーネントでは、特定の宛先が指定されることも、そのイベントを利用する下流のコンポーネントが認識されることもありません。

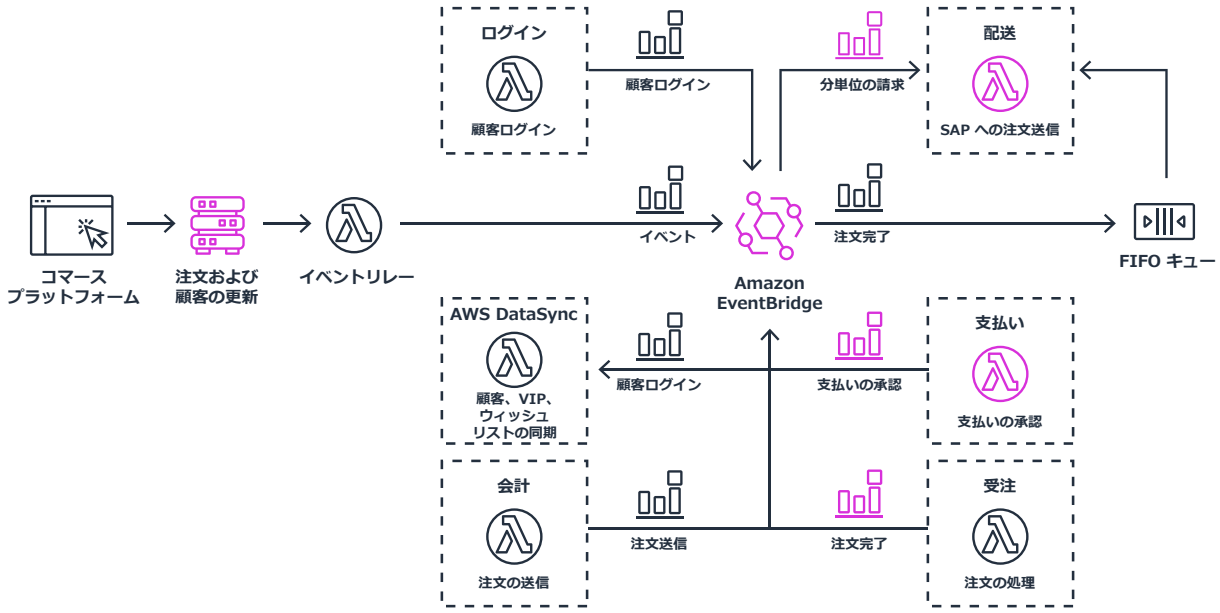
イベント駆動型アーキテクチャの主要なコンポーネントを以下に示します。

- **イベントプロデューサー**はイベントを発行します。例えば、フロントエンドウェブサイト、マイクロサービス、モノのインターネット (IoT) デバイス、AWS のサービス、SaaS (Software-as-a-Service) アプリケーションなどがこれに当たります。
- **イベントコンシューマー**はイベントでアクティブ化される下流のコンポーネントです。同じイベントに複数のコンシューマーが存在する場合もあります。イベントの利用には、ワークフローの開始、分析の実行、データベースの更新などがあります。
- **イベントブローカー**はプロデューサーとコンシューマーを仲介し、共有イベントを発行および利用して、両者の緩衝役を務めます。例えば、イベントをターゲットにプッシュするイベントルーターや、コンシューマーがイベントをプルするイベントストアなどがこれに当たります。

ユースケースの例

マイクロサービス通信

このユースケースは、予測できないトラフィックを処理するためにスケールアップする必要がある小売店やメディア、エンターテインメントのウェブサイトでよく見られます。お客様が e コマースウェブサイトにアクセスし、注文するとします。注文イベントはイベントルーターに送信され、すべてのダウンストリームマイクロサービスで注文イベントを取得して処理することが可能です。例えば、注文の送信、支払いの承認、配送業者への注文の詳細の送信などです。各マイクロサービスのスケーリングや障害は独立して発生するため、単一障害点はありません。**LEGO** はブラックフライデーのピークトラフィックに対応するためにこのパターンで e コマースのウェブサイトを拡張しました。

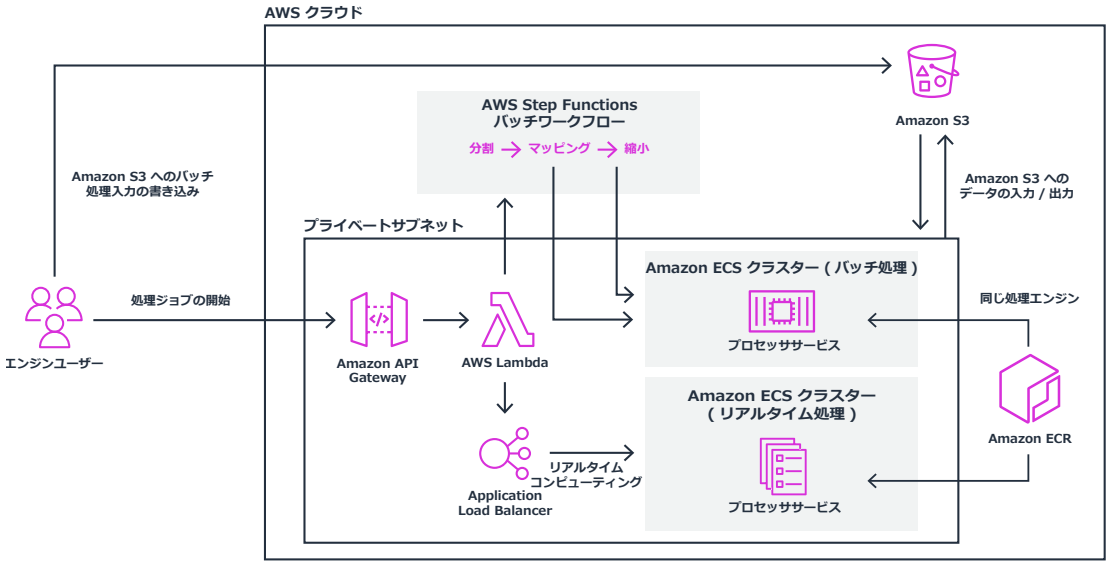


IT オートメーション

AWS のサービスを使用しているインフラストラクチャでは、**Amazon Elastic Compute Cloud** (Amazon EC2) インスタンスの状態変更イベント、**Amazon CloudWatch** のログイベント、**AWS CloudTrail** のセキュリティイベントなど、さまざまなイベントが既に作成されています。これらのイベントを使用して、設定の検証、ログ内のタグの読み取り、ユーザー行動の監査、セキュリティインシデントの修正についてインフラストラクチャを自動化できます。

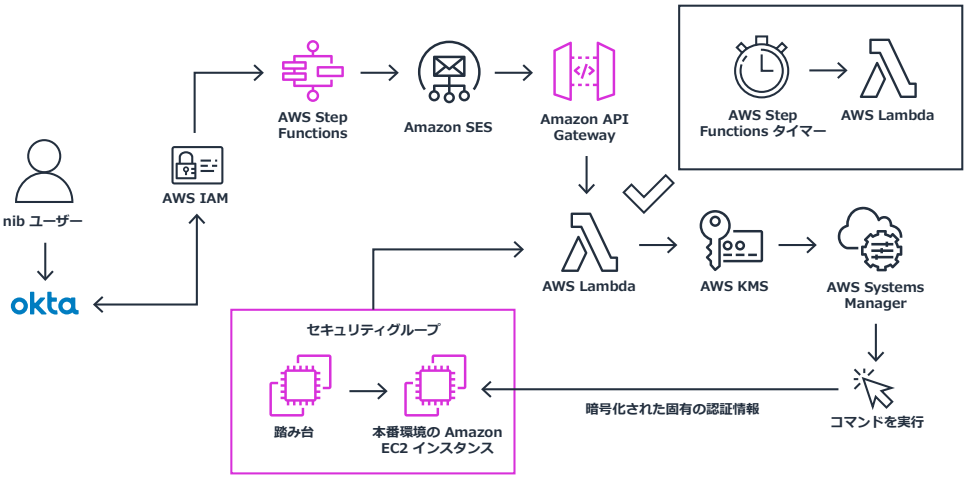
財務分析、ゲノミクス研究、メディアトランスコーディングなど、コンピューティングを多用するワークロードを実行しているお客様は、コンピューティングリソースをトリガーして、高度な並列処理のためにスケールアップし、ジョブが完了したらそれらのリソースをスケールダウンできます。**Société Générale** は、信用リスク分析のリソースを自動的にスケールアップまたはスケールダウンしています。

SOCIETE GENERALE



規制の厳しい業界 (医療や金融など) のお客様は、イベント駆動型アーキテクチャを使用して、インシデントに対応するセキュリティ態勢をスピンアップしたり、セキュリティポリシーがアラートを送信したときに修復アクションを実行できます。**nib Group (nib)** は、リソースへの安全なアクセスのために、期限付きの監査可能なセキュリティ体制を構築します。

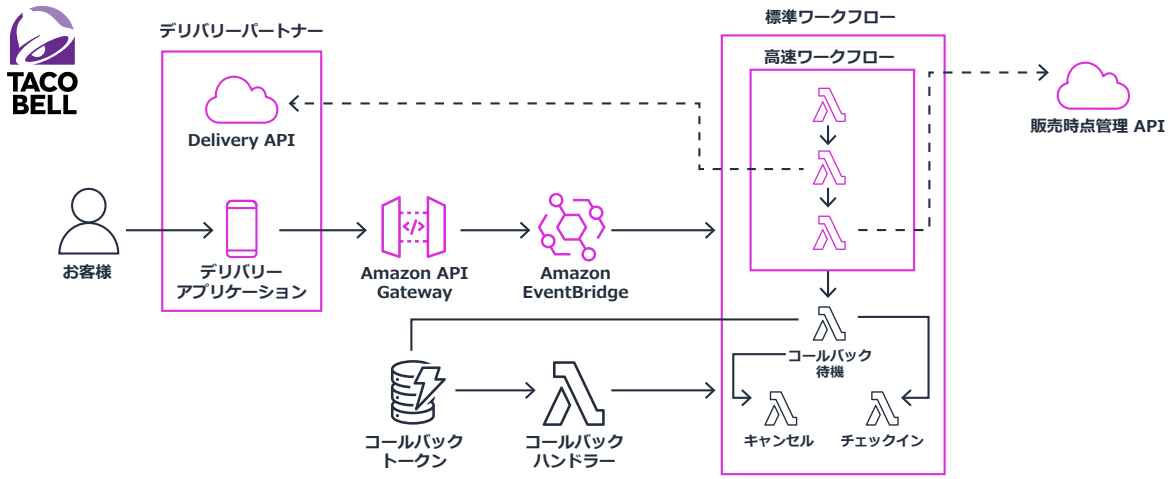
nib



アプリケーションの統合

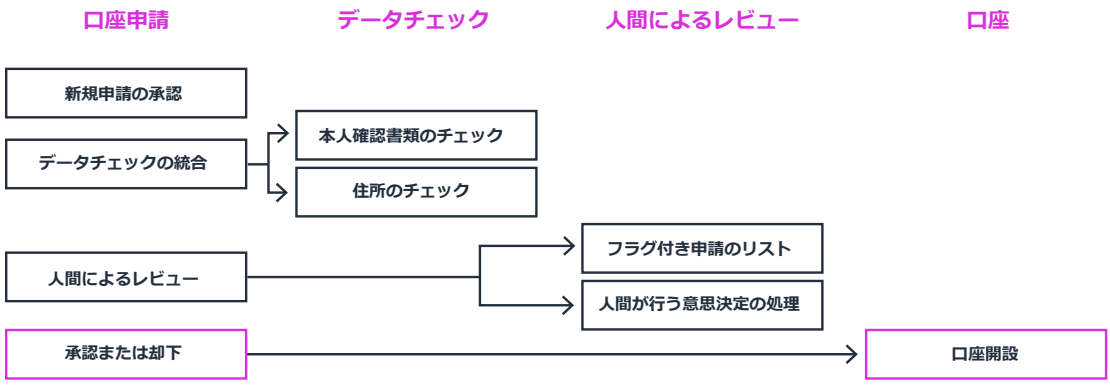
イベントを使用して、他のアプリケーションを統合することができます。オンプレミスのアプリケーションからクラウドにイベントを送信し、それらを使用して新しいアプリケーションを構築できます。SaaS アプリケーションを統合すると、それらのイベントを使用してカスタムワークフローを作成できます。顧客関係管理、決済処理、カスタマーサポート、アプリケーションのモニタリングとアラートなどのサービスの SaaS アプリケーションを使用できます。

BetterCloud によると、企業は 2021 年に平均 110 個の SaaS アプリケーションを使用していました。回答者の半数以上が、SaaS 環境の最大の課題はユーザーアクティビティとデータの可視性が欠如していることであると述べています。お客様はサイロ化されたデータを解放するために、SaaS アプリケーションイベントを取り込んだり、SaaS アプリケーションにイベントを送信するイベント駆動型アーキテクチャを構築します。**Taco Bell** は、デリバリーパートナーのアプリケーションの注文を取り込んで、店舗の POS アプリケーションに直接送信する注文ミドルウェアソリューションを構築しました。



ビジネスワークフローの自動化

このユースケースは、金融サービストランザクションまたはビジネスプロセスの自動化でよく見られます。多くのビジネスワークフローでは同じステップを繰り返す必要があります。それらのステップはイベント駆動型モデルで自動化して実行できます。例えば、顧客が銀行で新規の口座開設を申請する場合、銀行はいくつかのデータチェック (身分証明、書類、住所など) を実行する必要があり、口座によっては人間による承認段階が必要になります。これらのすべてのステップは、ワークフローサービスを介してオーケストレートでき、新規口座の申請が送信されるたびにワークフローが自動的にトリガーされます。



セクション 2

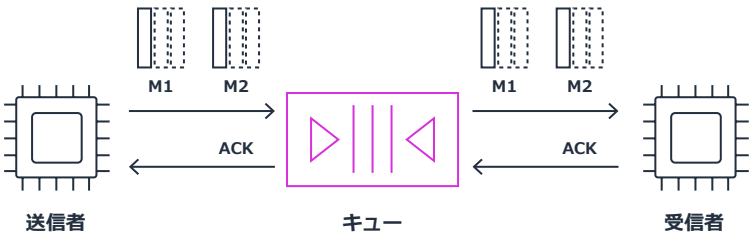
イベント駆動型アーキテクチャの一般的なパターン

このセクションでは、イベント駆動型アーキテクチャでよく見られるビルディングブロックとパターンについて説明します。これらのパターンは、プロデューサーとコンシューマー間の疎結合な非同期通信を可能にする一方で、さまざまなアプリケーションの要件を満たすために必要性が高まる独自性を生み出しています。

ポイントツーポイントメッセージング

ポイントツーポイントメッセージングとは、プロデューサーが送信するメッセージが原則単一のコンシューマーに宛てられるパターンです。多くの場合、メッセージキューがイベントブローカーとして使用されます。キューは、送信者と受信者の間の非同期通信を実現するメッセージングチャネルです。コンシューマーが使用不可の場合や、所定の時間内に処理するメッセージ数を制御する必要がある場合に、メッセージのバッファとして機能します。メッセージは、コンシューマーによって処理されてキューから削除されるまで保持されます。

マイクロサービスアプリケーションでは、マイクロサービス間の非同期なポイントツーポイントメッセージングのことを「**ダムパイプ**」と呼びます。



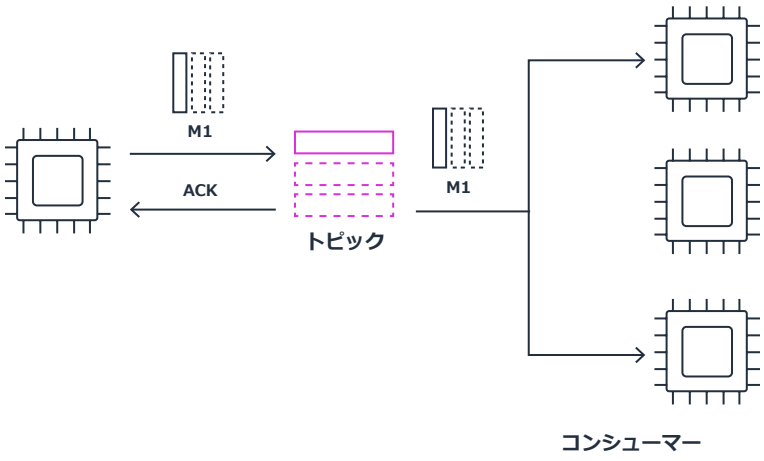
イベント駆動型アーキテクチャのメッセージキューには、**Amazon Simple Queue Service** (Amazon SQS)、**Amazon MQ** などのサービスがよく使用されます。**AWS Lambda の非同期呼び出し**を使用することもできます。

同期呼び出しでは、呼び出し元は Lambda 関数の実行が完了して、関数が値を返すのを待ちます。非同期呼び出しでは、呼び出し元が内部キューにイベントを入れると、Lambda 関数がイベントを処理します。このモデルを使用すると、仲介するキューやルーターを管理する必要がなくなります。呼び出し元は、Lambda 関数にイベントを送信したら、そのまま先に進むことができます。関数の結果は**送信先**に送信でき、成功の場合と失敗の場合で設定を変更できます。呼び出し元と関数の間の内部キューは、メッセージが永続的に保存されることを保証します。

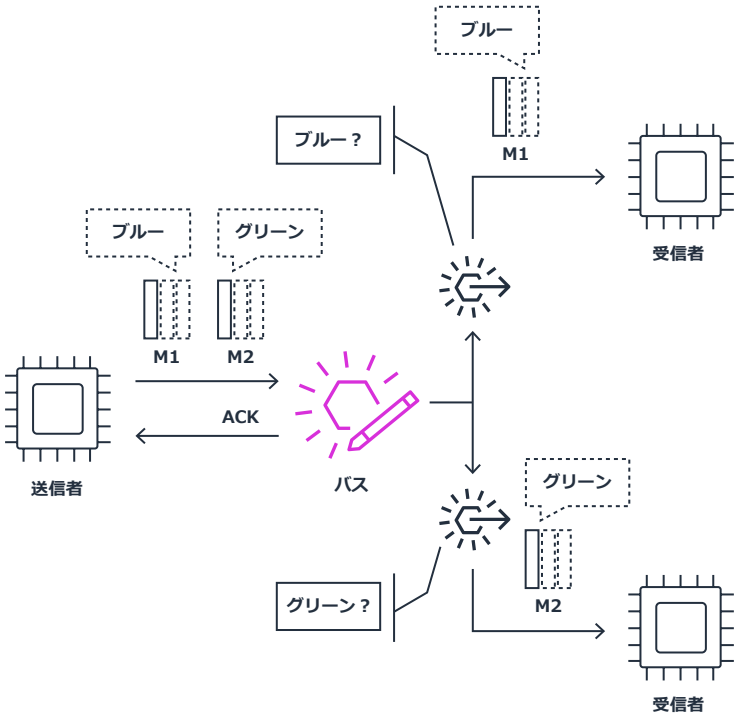
パブリッシュ/サブスクライブメッセージング

パブリッシュ/サブスクライブメッセージングは、プロデューサーが同じメッセージを 1 つまたは複数のコンシューマーに送信する方法です。ポイントツーポイントメッセージングでは、メッセージが単一のコンシューマーのみに送信されますが、パブリッシュ/サブスクライブメッセージングでは、メッセージをブロードキャストして各コンシューマーにコピーを送信できます。これらのモデルのイベントブローカーは多くの場合、イベントルーターです。イベントルーターでは、キューとは違って、通常はイベントの永続性が確保されません。

イベントルーターの一例となるのが**トピック**です。トピックとはハブアンドスポーク統合を実現するメッセージの宛先です。このモデルでは、プロデューサーがメッセージをハブに発行し、コンシューマーが選択したトピックをサブスクライブします。



イベントバスもイベントルーターの一種です。イベントバスは、複雑なルーティングロジックを組むことができます。トピックでは、送信されたメッセージがすべてサブスクライバーにプッシュされますが、イベントバスでは、送られてくるメッセージをフィルタリングして、イベントの属性に基づいて異なるコンシューマーにプッシュできます。



トピックを作成するには **Amazon Simple Notification Service** (Amazon SNS) を、イベントバスを作成するには **Amazon EventBridge** を使用できます。EventBridge では、**アーカイブ**機能によってイベントの永続化がサポートされます。トピックとルーティングは **Amazon MQ** でもサポートされています。

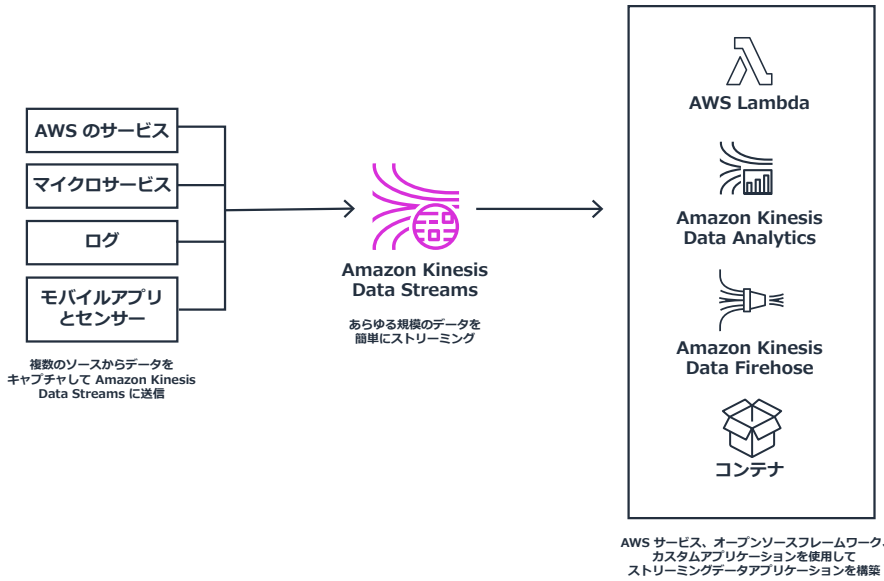
イベントのストリーミング

イベントやデータの連続的なフローであるストリームの使用も、プロデューサーとコンシューマーを抽象化する手法のひとつです。ストリームでは、イベントルーターとは対照的に (キューと比べて)、通常はコンシューマーが新しいイベントをポーリングする必要があります。コンシューマーは、独自のフィルタリングロジックを維持して、ストリーム内の位置を追跡しながら、使用するイベントを決定します。

イベントストリームは、イベントの連続的なフローです。イベントが個別に処理される場合もあれば、一定の期間のイベントがまとめて処理される場合もあります。お客様の位置の変化がイベントとしてストリーミングされるライドシェアアプリケーションは、イベントストリーミングの一例です。各「LocationUpdated」(位置が更新された) イベントは、地図上に表示されるお客様の位置を更新するための有意なデータポイントです。時間の経過に伴うロケーションイベントを分析して、運転速度などの分析情報を得ることもできます。

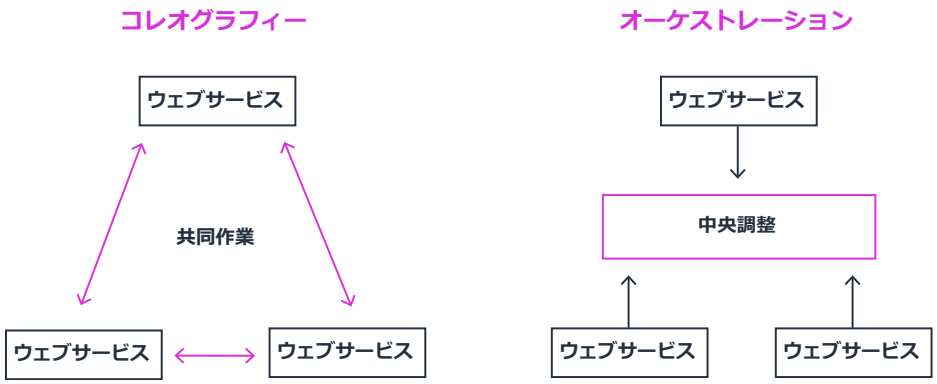
データストリームは、データを常に時間の経過で解釈する点で、イベントストリームとは異なります。このモデルでは、個々のデータポイントまたは記録が単独で使用されることはありません。データストリーミングアプリケーションは、オプションのエンリッチメントの後にデータを保持したり、データを経時的に処理してリアルタイムの分析を引き出したりするためによく使用されます。例えば、IoT デバイスのセンサーデータのストリーミングなどがこれに当たります。個々のセンサーの読み取り記録は、コンテキストがなければ価値がないかもしれませんが、それらを一定の期間にわたって収集すると、いろいろなことがわかるようになります。

イベントストリーミングとデータストリーミングのユースケースには、**Amazon Kinesis Data Streams** と **Amazon Managed Streaming for Apache Kafka** (Amazon MSK) を使用できます。



コレオグラフィーとオーケストレーション

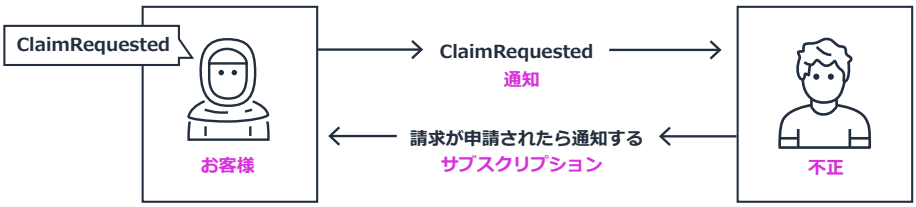
コレオグラフィーとオーケストレーションは、分散サービスの相互通信を実現するための 2 つの異なるモデルです。コレオグラフィーでは、厳密に制御されない通信が実現されます。オーケストレーションでは通信がより厳密に制御されます。中央サービスが、サービスを呼び出す相互作用と順序を調整します。サービス間のイベントフローが一元的に調整されることはありません。多くのアプリケーションでは、コレオグラフィーとオーケストレーションの両方が、それぞれ異なるユースケースで使用されます。



コレオグラフィー

境界付けられたコンテキスト間の通信では、多くの場合、コレオグラフィーが最も効果的に使用されます。コレオグラフィーでは、イベントがいつどのように処理されるかをプロデューサーが想定することはありません。プロデューサーの責任は、イベントをイベント取り込みサービスに送信することと、スキーマに準拠することだけです。これにより、境界付けられた 2 つのコンテキスト間の依存関係が削減されます。

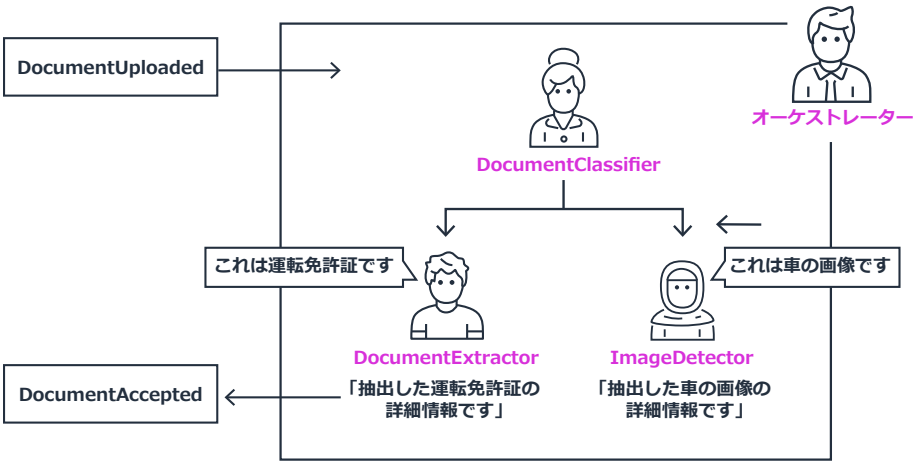
下図は、保険金請求処理アプリケーションの 2 つの異なるドメイン (境界付けられたコンテキスト) である「お客様ドメイン」と「不正ドメイン」を示しています。お客様ドメインは、保険金請求が提出されるたびに「ClaimRequested」(請求がリクエストされた) イベントを送出します。不正ドメインは、保険金請求が提出された時点でドメインロジックを適用するために、お客様ドメインが送出した「ClaimRequested」イベントをサブスクライブする必要があります。「ClaimRequested」イベントが送出されると、不正ドメインに通知が届きます。イベントのコレオグラフィーの全過程において、お客様 (プロデューサー) ドメインも不正 (コンシューマー) ドメインも、相手の内部ビジネスロジックについて把握している必要がありません。こうしたコレオグラフィーのアプローチなら、疎結合が機能します。



オーケストレーション

境界付けられたコンテキスト内で、サービス統合の順序の制御、状態の維持、エラーと再試行の処理が必要になることもよくあります。これらのユースケースにはオーケストレーションの方が適しています。

下図は、保険金請求処理アプリケーションの文書処理ドメイン (境界付けられたコンテキスト) を示しています。「DocumentUploaded」(書類がアップロードされた) イベントをドメインが受信します。文書処理ドメインにはオーケストレーターが存在し、アップロードされた文書の種類を調べます。オーケストレーターは、アップロードされた画像が運転免許証か自動車かに基づいてワークフローパスを決定します。文書分類担当者 (DocumentClassifier) が、画像が運転免許証であると判断します。その後、免許証から情報をすべて抽出するように文書抽出担当者 (DocumentExtractor) に指示します。抽出されたデータがデータベースで更新され、その後、画像に関連する詳細情報をすべて添えて文書処理ドメインが「DocumentAccepted」イベントを送出します。

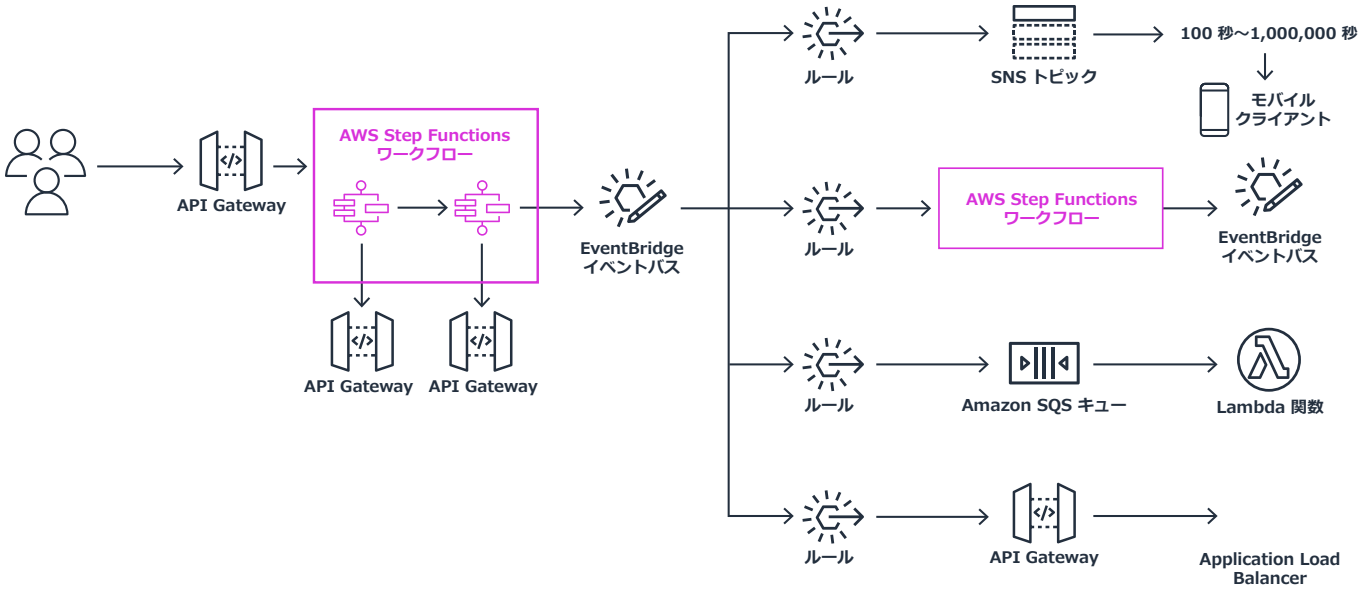


EventBridge などのイベントバスはコレオグラフィーに使用できます。**AWS Step Functions** や **Amazon Managed Workflows for Apache Airflow** (Amazon MWAA) などのワークフローオーケストレーションサービスは、オーケストレーションの構築に役立ちます。コレオグラフィーとオーケストレーションを併用する例としては、イベントを送信して Step Functions ワークフローをトリガーした後、さまざまなステップで**イベントを送出**する場合などが考えられます。

コレオグラフィーとオーケストレーションの共存

同一アプリケーション内のコレオグラフィーとオーケストレーションを取り上げた上記の例から、両者が相互に排他的な関係ではないことがわかります。多くのアプリケーションでは、コレオグラフィーとオーケストレーションの両方が、それぞれ異なるユースケースで使用されます。

下の例では、左側のプロデューサーが EventBridge イベントバスを介してイベントを送出し、右側の複数のコンシューマーがそれらのイベントを利用しています。プロデューサーとコンシューマー間ではコレオグラフィーでイベントが処理され、プロデューサーは、境界付けられたコンテキストの中で Step Functions を使用して **Amazon API Gateway** への 2 つの API 呼び出しをオーケストレートしています。また、右側のコンシューマーのうち 1 つが、境界付けられたコンテキスト内で Step Functions を使用してオーケストレートしているのもわかります。



コレオグラフィーとオーケストレーションを共存させることで、ドメイン駆動設計のさまざまなニーズに柔軟に対応できます。

イベントソースの接続

外部イベントソースを持つアプリケーションは数多くあります。例えば、給与の計算、記録の保存、チケットの発行などを行うビジネスアプリケーションのような SaaS アプリケーションがこれに当たります。オンプレミスで実行されている既存のアプリケーションやデータベースからイベントを取り込むこともできます。イベント駆動型アーキテクチャでは、これらのすべてのソースのイベントを使用できます。

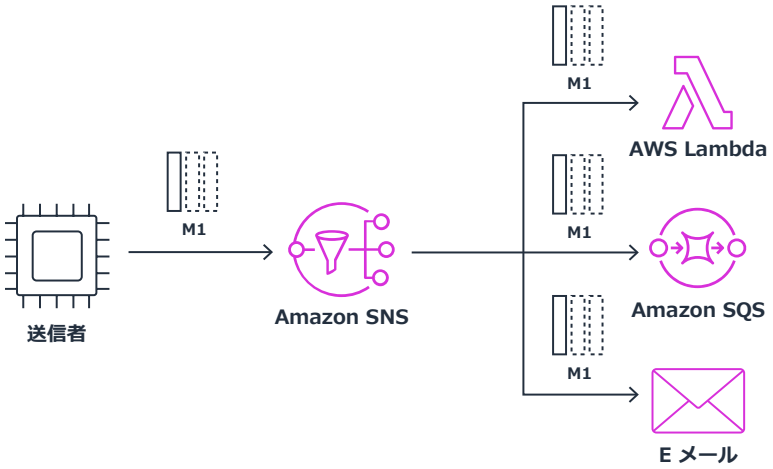
アプリケーションから送出されたビジネスイベントを伝播するには、コネクタ、つまりメッセージブローカーを使用するのが一般的です。これらのコネクタは、SaaS アプリケーションやオンプレミスのソースとの橋渡しとなり、イベントをストリームまたはルーターに送信して、コンシューマーが処理できるようにします。**EventBridge のパートナーのイベントソース**を使用すると、統合された SaaS アプリケーションから AWS アプリケーションにイベントを送信できます。

こちらの **AWS Architecture の記事**では、**Amazon MQ** または **Amazon MSK** のどちらかを使用してメインフレームコネクタを構築する例を紹介しています。

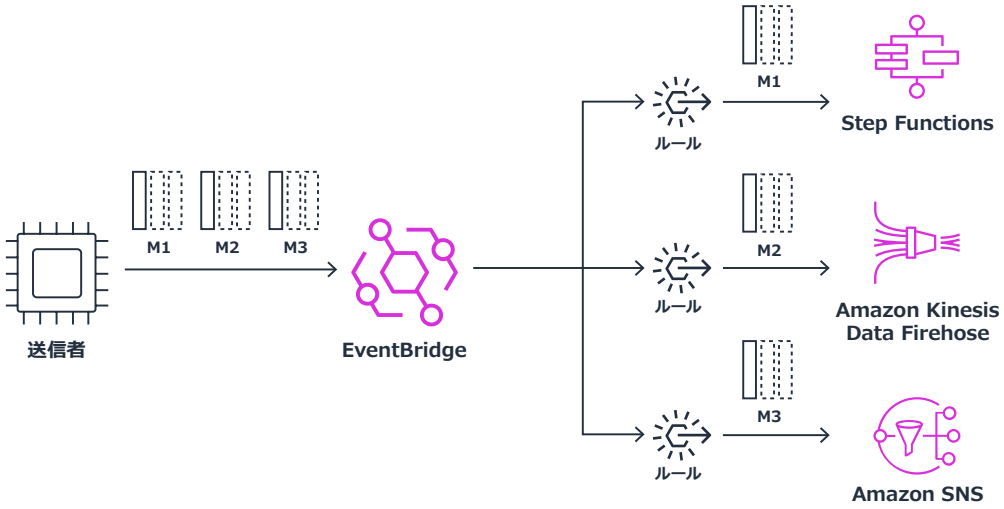
パターンの組み合わせ

イベント駆動型アーキテクチャでは、1 つのパターンで要件が満たされる場合もありますが、多くの場合、以下のパターンの組み合わせが使用されます。

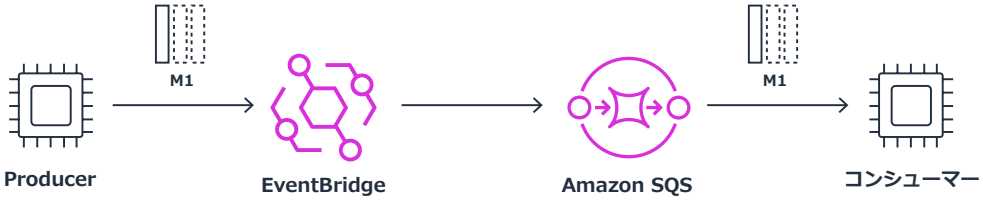
- ファンアウトして、単一のトピックの複数のサブスクライバーに同じメッセージを送信する



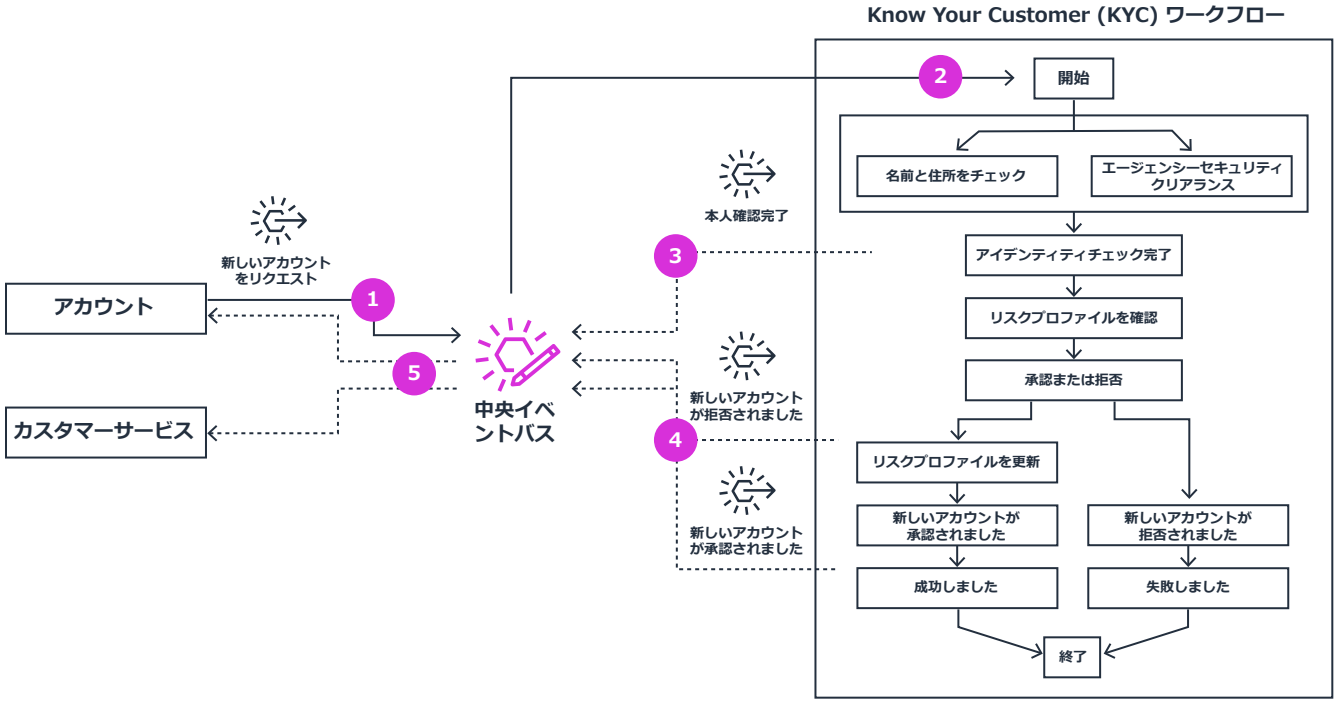
- イベントのフィルタリングとルーティングにより、特定のイベントを異なるターゲットに送信する



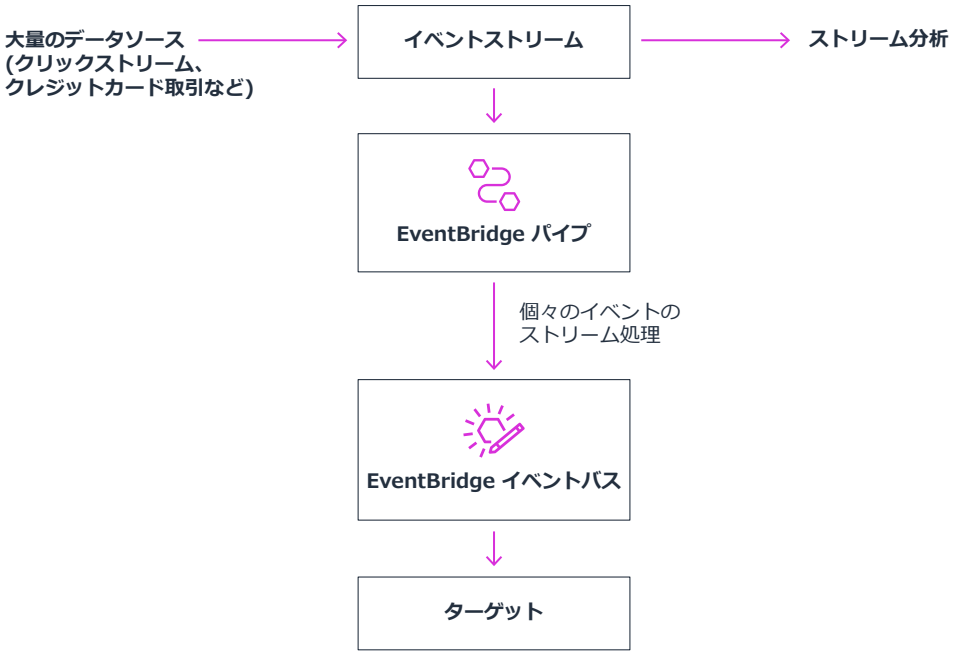
- キューを使用して下流のコンシューマーへのイベントやメッセージの量をバッファリングする



- ワークフローをオーケストレートして、ワークフロー内の各ステップでイベントを送出する



- イベントストリーミングプラットフォームと EventBridge を組み合わせて、統合コードを記述せずに重要なビジネスイベントをサブスクライブする



セクション 3

イベント駆動型アーキテクチャに関する考慮事項

イベント駆動型アーキテクチャは、大規模なアプリケーションの構築や運用に役立つ反面、新たな課題や複雑さをもたらす可能性もあります。このセクションでは、イベント駆動型アーキテクチャを設計する際に留意すべき考慮事項について説明します。

結果整合性

イベント駆動型アーキテクチャでは、イベントが優先的に扱われ、データの管理は分散されます。結果として、アプリケーションは結果整合性があるのが一般的です。つまり、アプリケーションでデータの完全な同期は確保されませんが、最終的な整合性は確保されることを表します。

結果整合性は、トランザクションの処理、重複の処理、システム全体の正確な状態の特定などの際に、事態が複雑になる可能性があります。多くのアプリケーションでは、他のコンポーネントに比べて結果整合性データの処理に適したコンポーネントがあります。

可変レイテンシー

イベント駆動型のアプリケーションは、単一デバイス上の同じメモリのスペースを使用してすべての処理を行うモノリシックなアプリケーションとは違って、ネットワーク経由で通信します。その結果、可変レイテンシーが生じます。イベント駆動型のアプリケーションでも、設計によってレイテンシーを最小化することは可能ですが、モノリシックなアプリケーションは、スケーラビリティやアクセシビリティと引き換えに、ほぼ常に低レイテンシーに最適化されます。

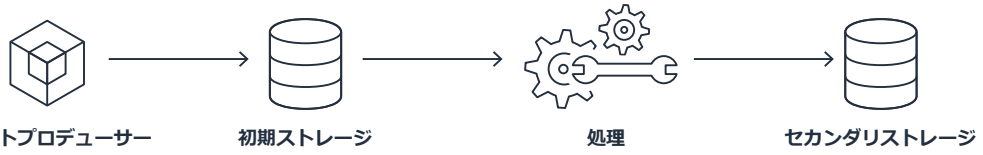
一貫した低レイテンシーのパフォーマンスが要求されるワークロードは、イベント駆動型アーキテクチャに適していません。例えば、銀行における高頻度取引アプリケーション、倉庫におけるミリ秒未満のロボティクスオートメーションなどがこれに当たります。

テストとデバッグ

自動テストは、イベント駆動型アーキテクチャの要となるコンポーネントです。そのおかげで、高品質のシステムを効率的かつ正確に開発することができます。このセクションでは、イベント駆動型アーキテクチャと非同期システムの自動テストを設計する上でのガイダンスを紹介します。

一般的な非同期パターン

非同期システムは通常、イベントパブリッシャーとイベントコンシューマーで構成されます。パブリッシャーがコンシューマーにメッセージを送信し、コンシューマーはそのメッセージを後での処理のために即座に保存します。その後、下流のシステムが、保存されたデータを操作する場合があります。処理されたデータは、別のサービスに出力として送信されたり、別の保管場所に配置されます。下図は、一般的な非同期パターンを示しています。



論理的な境界の確立

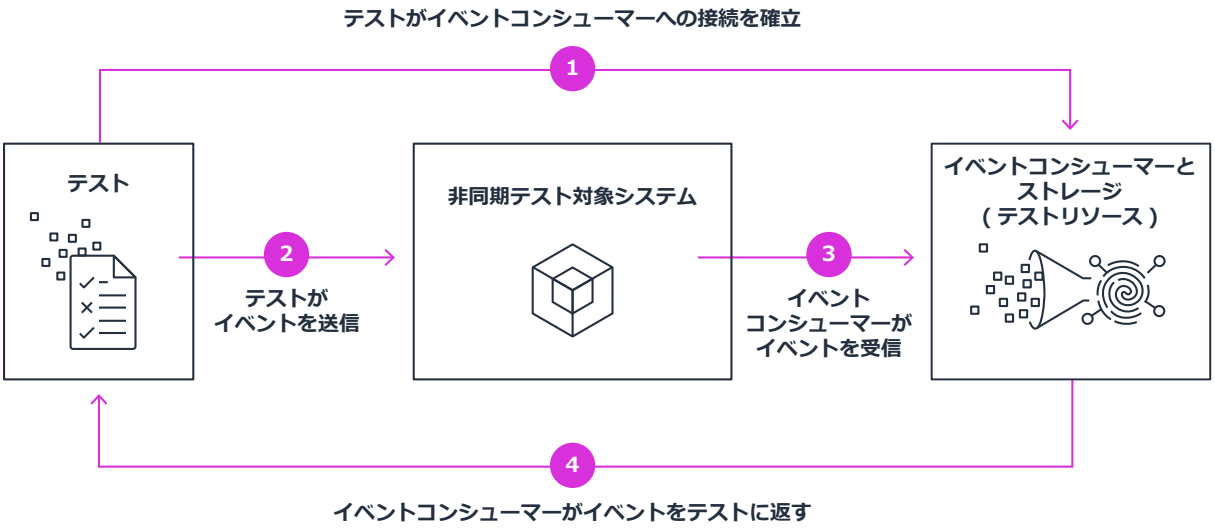
上の図のような非同期パターンが単独で存在することはほとんどありません。通常、本番環境のシステムは、多数のサブシステムが相互に接続されて成り立っています。妥当なテスト戦略を立てる上で有効なのは、複雑なシステムを論理的なサブシステムのセットに分割することです。サブシステムは、例えば、連携して単一のタスクを実行するサービスのグループです。サブシステムの入力と出力を明確に理解しておくことが必要です。複雑なアーキテクチャを比較的小さいサブシステムに分割することで、対象を絞って切り分けたテストを簡単に作成できます。

テストハーネスの作成

非同期システムのテストでは、テストハーネスを作成しておくと役に立ちます。ハーネスには、サブシステムへの入力を生成し、システムからの出力を受け取るリソースを含めておきます。テストでは、ハーネスを使ってシステムを動かし、想定どおりに動作しているかどうかを判断します。これらのハーネスは、テスト用途にのみ使用されるリソースです。本番環境の機能では使用しません。テストハーネスは通常、運用前の環境にのみデプロイされます。しかし、本番環境で要件を検証したい場合もあるでしょう。テストデータの処理に耐えられるようにシステムを設計できれば、ハーネスを使ったテストを本番環境にデプロイすることができます。

テストイベントプロデューサーとテストイベントコンシューマーの設定

テストハーネスは通常、テストイベントプロデューサーとテストイベントコンシューマーで構成されます。プロデューサーはテスト対象システム (SUT) に入力を提供し、コンシューマーは出力を受け取るように構成されます。自動テストはプロデューサーにイベントを送信し、コンシューマーと通信して出力を調べます。出力が想定どおりであれば、テストは合格です。



サービスレベルアグリーメントの定義

アーキテクチャが非同期ではあっても、システムが障害状態にあると判断されるまでのイベント処理にかかる最長時間について、合理的な期待値を設定することは依然として有効です。こうした期待値は、サービスレベルアグリーメント (SLA) として明示的に定義できます。テストの設計時に、SLA を満たすタイムアウトを設定できます。システムがタイムアウト期間内に結果を返さない場合は、SLA 違反と見なすことができます。その場合は不合格になるように、テストを設計する必要があります。

スキーマとコントラクトのテストの作成

イベント駆動型アーキテクチャでは、プロデューサーとコンシューマーがインフラストラクチャ層では分離していても、アプリケーション層ではイベントコントラクトによって結合されている場合があります。コンシューマーは、イベントが特定のスキーマに準拠することを前提としたビジネスロジックを記述する場合があります。スキーマが時間の経過に伴い変化すると、コンシューマーのコードが失敗する可能性があります。イベントスキーマを検証する自動テストを作成しておけば、システムの品質向上や破壊的変更の防止につながります。

すべてのステージにおけるテスト

イベント駆動型アーキテクチャではテストが不可欠です。イベント駆動型アーキテクチャでは、イベントの送信が引き起こす一連の反応を正確に模倣するのは難しい場合があります。すべてのステージでテストを実施すれば、さまざまなユースケースシナリオを特定し、バグを見つけやすくなります。医療システムや金融システムなど、規制の厳しい業界でシステムを運用している組織では、運用前の包括的なテスト環境に投資することが重要です。

テスト用ツールの作成

ツールに投資することで、障害による影響を最小限に抑えることができます。カナリアデプロイを採用して、コードの変更を一気に全体に適用するのではなく、定義した単位で徐々に環境に導入できます。機能フラグを使用すれば、コードの導入とバックアウトをすばやく実施できます。可観測性のツールに投資することで、環境内部のエラー率を測定できます。継続的インテグレーション/継続的デリバリー (CI/CD) パイプラインで確立されたロールバック手順により、障害の原因となるコードを削除できます。コード変更を小さい単位で徐々にデプロイすることで、デプロイ時のリスクが軽減され、俊敏性が向上します。

テストサンプルを見る

コードサンプル

この GitHub リポジトリには、自動テストのサンプルが多数含まれています。**このプロジェクト**では、テストとイベントリスナーを含む Python で記述されたシンプルな非同期システムを作成します。プロジェクトをビルドしてテストを実行するには、**README** ファイルの指示に従ってください。

統合テストの例

サーバーレステストサンプル (GitHub)

テストを実行したら、**/tests/** ディレクトリのコードを調べて、テストがどのように記述されているかを確認してください。

スキーマとコントラクトのテストの例

serverless-test-samples プロジェクトのルートに移動します。次に記載されているコマンドを実行して、TypeScript で記述された**スキーマとコントラクトのテスト用のサンプルプロジェクト**を見つけてください。**README** ファイルの指示に従ってテストを実行します。

サーバーレステストサンプル (GitHub)

テストを実行したら、**/tests/** ディレクトリのコードを調べて、テストがどのように記述されているかを確認してください。

組織文化

イベント駆動型アーキテクチャの採用は、単なる技術上の決定ではありません。そのメリットを最大限に引き出すには、考え方の切り替えと開発カルチャーの転換が必要です。また、ビジネスドメインを軸としたチームの編成、DevOps カルチャーを取り入れた**分散型ガバナンスモデル**の構築、進化型設計の原則の実践などの変更も必要でしょう。これらの変化にはチームのスキルアップが必要になることもありますが、イベント駆動型アーキテクチャが実現されれば、アプリケーションに俊敏性、スケーラビリティ、信頼性などのメリットがもたらされます。

開発者には、担当するマイクロサービスのコンテキストでテクノロジーとアーキテクチャを自由に選択できる高度な自律性が必要です。ガードレールと可観測性を主軸に据えると、イベント駆動型アーキテクチャに必要な組織変革を成功裏に進めることができます。ガードレールと可観測性を実現すれば、標準を補強しつつ自律性のカルチャーを醸成できます。

ガードレール
ガードレールとは、あらかじめ定義された方針、コントロール、標準のことです。組織全体にベストプラクティスを適用するうえで有用です。ガードレールを使用すれば、偶発的な構成ミスを防ぎ、セキュリティ方針を適用し、本番環境でのインシデントのリスクも軽減されます。ガードレールにより安全な開発と運用を目指した明確なガイドラインが設定され、質の高いカルチャーを確立できます。

可観測性
可観測性とは、本番環境におけるシステムやアプリケーションの動作を監視、分析、測定する能力のことです。可観測性を実現することで、組織はパフォーマンス、信頼性、セキュリティ、コンプライアンスの面でシステムの理解を深めることができます。最小限の監視で、顧客に影響が及ぶ前に問題を特定し、顧客に影響を与えることなくシステムを継続的に改善することができます。可観測性は、透明性や説明責任を向上させ、継続的改善を促進して、信頼のカルチャーを育みます。

安全な開発と運用を目指した明確なガイドラインを設け、透明性と継続的改善を促進することで、自律性、セキュリティ、信頼性の高いカルチャーを醸成できます。ソフトウェア開発と IT 運用のライフサイクルの中でこれらを実践できれば、チームは自信を持ってシステムやアプリケーションの所有者を務められるでしょう。

セクション 4

ベストプラクティス

このセクションでは、イベント駆動型アーキテクチャにおけるアーキテクチャの選択と一般的な課題への対処に関するベストプラクティスを紹介します。

イベントの設計

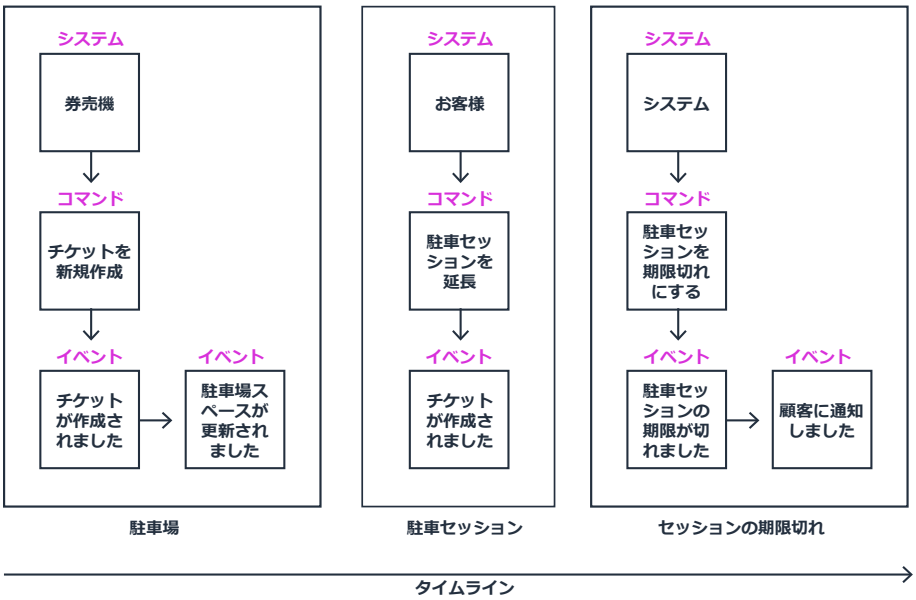
既に説明したとおり、イベントとは、状態の変化を示すシグナルです。イベントは過去に発生したことを表し (例えば、「OrderCreated」(注文が作成された) など)、変更できません。

イベント駆動型アプリケーションの構築にあたっては、イベントを特定し、イベントの設計に時間をかけます。プロデューサーやコンシューマーが多く、時間の経過とともに組織全体に拡張可能なイベント駆動型アーキテクチャを開発および実装する場合、この点はとりわけ重要です。

イベントを設計する前に、システム内のイベント (アーキテクチャにとって重要であるだけでなく、ビジネスにとっても重要なイベント) を特定することが重要です。イベントを発生させて、既存または将来の下流のコンシューマーがそのイベントに反応し、ビジネスロジックを処理できるようにします。

イベントストーミングによるイベントの特定

イベントストーミングは、協働的にシステムの動作を視覚的に把握し、イベント駆動型アーキテクチャにおけるイベントを特定していく手法です。システムのさまざまなドメインから関係者を集め、システムのイベントとアクションを協働的に視覚化し、話し合うワークショップを促進します。主な目標は、システムについての理解を共有し、重要なビジネスイベントを洗い出すことです。



イベントストーミングの過程で、アクター、コマンド、アグリゲート、イベントを明らかにすることができます。そのおかげで、自分やチームがシステムの動作を理解でき、実装前にイベントを特定することもできます。イベントストーミングは、新規プロジェクトと既存プロジェクトの両方で活用できます。関係者間で共通の理解が深まれば、イベント駆動型アーキテクチャの構築にプラスに働きます。

イベントが特定できたら、それらに対し命名規則を実装し、定義することが重要です。

イベントの命名規則

イベントは過去に起きた不変の事実であるため、イベントの命名規則は正確かつ明確でなければなりません。コンシューマーがイベントを発見して使用するうえで、不明瞭な点やあいまいな点がある場合は取り除き、イベントの意味とコンテキストを伝えなくてはなりません。

例えば、「UserLoggedIn」(ユーザーがログインした) と「LoggedIn」(ログインした) を比較すれば、この点は明らかです。「UserLoggedIn」イベントは意味やコンテキストが明確で、ユーザーがログインしたことを伝えています。一方の「LoggedIn」は、イベントの意味とコンテキストが不明瞭です。イベントの命名規則が明確であれば、下流のコンシューマーがイベントの意図を理解してサブスクライブしやすくなり、関連があるかどうかを深く掘り下げて判断する必要がなくなります。

イベントの命名規則を確立したら、検討対象となり得る、さまざまなイベントパターンを理解することが重要です。多くのイベント駆動型アーキテクチャには、**通知イベントと ECST (イベント実施状態転送) イベント**があります。

通知イベント

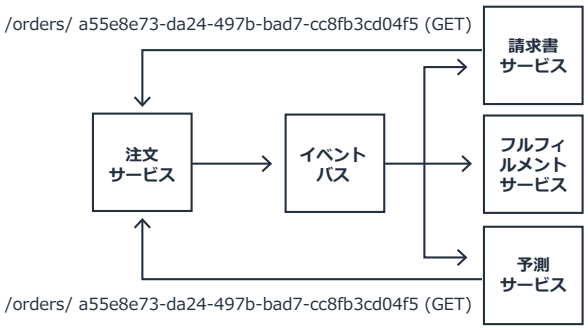
通知イベントは、下流のシステムに出来事の発生を知らせる役割を果たします。例えば、ユーザーが発注した際に、「OrderCreated」(注文が作成された) イベントを作成して、新しい発注について下流のコンシューマーに通知できます。通知イベントは小さなペイロードを含み、必要な情報だけを下流のコンシューマーに伝えます。このシンプルさのおかげで、プロデューサーとコンシューマーとのコントラクトが管理および保守しやすく簡単なものになります。

```
{
  "version" : "1" ,
  "id" : "07fffe3b-4b25-48a7-b4a8-
432bcc3bfb2c" ,
  "detail-type" : "OrderCreated" ,
  "source" : "myapp.orders" ,
  "account" : "123456789" ,
  "time" : "2022-06-01T00:00:00Z" ,
  "region" : "us-west-1" ,
  "detail" : {
    "data" : {
      "orderId" : "3c947443-fd5f-4bfa-
      8a12-2aa348e793ae" ,
      "userId" : "09586e5c-9983-
      4111-8395-2ad5cfd3733b"
    }
  }
}
```

通知イベントの例

単純なペイロードを含む EventBridge 通知イベントの例

下流のコンシューマーが通知イベントを処理する際に、追加の情報が必要になる場合があります。例えば、「OrderCreated」イベントが下流に送信され、「ordered」(発注された) という情報だけを伝えた場合、下流ドメインのコンシューマーが、イベントの処理にあたって注文の詳細情報を取得しなければならないケースが考えられます。その場合、必要な情報を取得するために望ましくないバックプレッシャーがプロデューサーや別の API にかかる可能性があります。このトレードオフは認識しておく必要があります。



AWS のサービスでは、ターゲットへの配信前にメッセージやイベントをフィルタリングできます。例えば、**EventBridge** では、下流のコンシューマーに届く前にイベントをフィルタリングするルールを作成できます。通知イベントを扱う場合、イベントに含まれている情報そのものが限られているため、実行できるフィルタリングも限定的になる場合があります。これが懸念される場合は、ECST (イベント実施状態転送) イベントの使用を検討してください。

ECST イベント

通知イベントはスパースで、簡潔なコントラクトですが、その対局にあるのが ECST イベントです。ECST イベントは、より多くの情報を添えて発行されるため、下流のコンシューマーが補足情報を取得する手間を省くことができます。必要に応じて、下流のコンシューマーもこの情報のローカルキャッシュコピーを保持できます。イベントに含まれるデータが増えるので、AWS サービスのフィルタリング機能を利用して、下流のコンシューマーに届く前に情報をフィルタリングできます。例えば、下流のコンシューマー向けに EventBridge でフィルターを使用したルールを作成できます。

```
{
  "version" : "1" ,
  ...
  "detail" : {
    "metadata" : {
      "domain" : "ORDERS"
    },
    "data" : {
      "order" : {
      "id" : "3c947443-fd5f-4bfa-8a12-
      2aa348e793ae" ,
        "amount" : 50,
        "deliveryAddress" : {
          "postCode" : "PE1111"
        }
      },
      "user" : {
      "id" : "09586e5c-9983-4111-8395-
      2ad5cfd3733b" ,
        "firstName" : "Dave" ,
        "lastname" : "Boyne" ,
        "email" : "dboyne@dboyne.com"
      }
    }
  }
```

ECST イベントの例

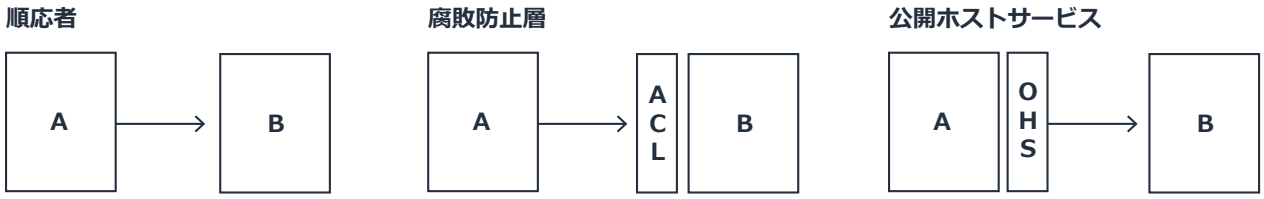
ペイロードが大きいイベントを発行する場合、いくつかのことを考慮する必要があります。まず考慮すべき点は、実装の詳細がドメイン/サービスからイベントに漏れるのを防ぐことです。イベントにどの情報を含めるべきか、何を下流のコンシューマーに公開すべきかを慎重に検討する必要があります。サービス間には境界が存在することに留意し、イベントスキーマを通じてプロデューサーとコンシューマーの間のコントラクトを作成します。このコントラクトはメンテナンスと管理が必要です。イベントに含める情報が多いほど、それらのイベントの変更やバージョンをより慎重に管理する必要があります。

選択すべきイベント設計のパターンは、ユースケースによって異なります。多くのアプリケーションには両方のパターンが組み込まれています。通知イベントや ECST イベントを利用する際は、コンシューマーの利用パターン (境界付けられたコンテキストのマッピング) を考慮する必要があります。そうすることで、プロデューサーとコンシューマーとのコントラクトで破壊的変更が生じた場合のリスクを管理および軽減できます。

境界付けられたコンテキストのマッピング：イベントの利用に役立つパターン

境界付けられたコンテキストのマッピングは、ドメイン駆動設計 (DDD) で使用される手法で、システム内の異なる境界付けられたコンテキスト間の関係を定義し、文書化します。境界付けられたコンテキストとは、大きなシステムの中の自己完結したドメインです。境界が明確で、独自の言語、モデル、ビジネスロジックを備えています。イベント駆動型アーキテクチャを構築する場合、メッセージやイベントを使用して、こうしたシステムの境界間で通信するのが一般的です。

発行されたイベントを利用する際、コンシューマーはその情報の利用方法をいくつかの選択肢から選択できます。イベントそのものに従うか (順応者)、イベントを変換するラッパーを記述するか (腐敗防止層 [ACL])、共有の公表された言語に同意して変換をプロデューサーにプッシュバックするか (公開ホストサービス [OHS]) を選択できます。



イベント駆動型アーキテクチャにおける境界付けられたコンテキストのマッピングの例

順応者 (Conformist) パターン
順応者パターンは、イベント駆動型アーキテクチャで使用されるパターンで、発行されたイベントを変換や変更を加えずにそのまま処理します。下流のコンシューマーがプロデューサーのスキーマに確実に適合し、プロデューサーとコンシューマーの間で明確なコントラクトを提供します。

順応者パターンを使用する場合は、発行されたドメイン情報のどの程度がドメイン/境界に漏れているかに注意してください。プロデューサーとコンシューマー間のコントラクトの変更による影響を抑えたい場合は、腐敗防止層 (ACL) の実装を検討することをお勧めします。

ACL
イベント駆動型アーキテクチャでは、ACL は、あるドメインのデータが破損したり、別のドメインで悪用されたりしないよう防ぐためのパターンです。ACL は、言語、モデル、またはビジネスロジックが異なる 2 つの境界付けられたコンテキスト間の仲介役となり、2 つのコンテキスト間の変換層として機能します。プロデューサーからのイベントを利用する場合、ACL を実装すると、コントラクトの変更による影響をコントロールでき、異なるドメインモデル間のマッピングに役立ちます。

OHS
イベント駆動型アーキテクチャでは、OHS パターンは、共有の公表された言語を定義することで、境界付けられた複数のコンテキストが相互に通信できるようにする設計パターンです。OHS パターンでは、通信の仲介役として使用できる共有の公表された言語をドメイン間で確立します。共有言語は、合意された一連のインターフェイス、コントラクト、またはスキーマによって定義され、変換をプロデューサーにプッシュバックするために使用できます。

境界付けられたコンテキストのマッピングパターンを実装すると、プロデューサーによるイベントの利用方法を制御でき、ドメインを処理または分離できたり、コントラクトの変更を中止することもできる可能性があります。検討すべきマッピングオプションは、ユースケースによって異なります。

イベントファーストの思考
イベントの特定と設計のプロセスは、継続的に行っていくものです。変化するビジネス要件にイベントが適合しているか確認するため、このプロセスは定期的に繰り返した方がいいでしょう。イベント駆動型アーキテクチャを実装する際には、イベント設計を重要な要素として扱うことが不可欠です。通常、イベントは必要に応じて設計および実装されますが、多くの場合後付けになります。しかし、考え方を**イベントファースト思考**のアプローチに転換し、明確な意図を持ち、正しいイベントパターンと利用パターンを採用すれば、エンジニアはイベント駆動型アーキテクチャを構築および実装しやすくなります。

冪等性

冪等性とは、最初に実行したときの結果を変更せずに何度でも適用できるというオペレーションの特性です。冪等性がある操作は複数回実行しても安全で、データの重複や不整合などの副作用を生むことはありません。

イベント駆動型アーキテクチャでは再試行メカニズムが使用されるため、冪等性が重要な概念になります。例えば、イベントを使用して Lambda 関数を非同期で呼び出す場合、最初の呼び出しで関数が失敗すると、Lambda の組み込みの**再試行ロジック**によって関数が再度呼び出されます。注文、決済など、1 回だけ処理する必要があるトランザクションを管理する際には、これが重要な考慮事項になります。

すべてのサービスを冪等に構築することを選べます。すべての操作を副作用なしに複数回実行できるようになるため、重複イベントに対処するのに便利です。しかし、アプリケーションが複雑になる可能性があります。別の方法として、各イベントに一意の ID を含めて**冪等キー**として使用できます。

イベントが処理されたら、その結果で永続データストアを更新します。同じ冪等キーを持つ新しいイベントが届いた場合は、最初のリクエストに結果を戻します。

```
{
   “source” : “com.orders” ,
   “detail-type” : “OrderCreated” ,
   “detail” : {
      “metadata” : {
       “idempotency-key” : “c9894c60-
       0558-4533-a9b0-8bb579303428”
      },
      “data” : {
       “orderId” : “e92570b8-3fb3-4a4b-
       b24b-d697919fe56c”
      }
   }
}
```

冪等性イベントの例

順序

イベント駆動型アーキテクチャでは、イベントが特定の順序で配信される必要があるか (順序付けられたイベント)、順序が問題にならないか (順序付けられていないイベント) が重要な考慮事項です。順序は通常、特定の範囲内で保証されます。

順序付けられたイベント
構築に使用するサービスとして、順序を保証する Kinesis Data Streams や Amazon SQS FIFO (First-In-First-Out) などを選択できます。**Kinesis Data Streams** では、シャード内のメッセージで順序が保持されます。**Amazon SQS FIFO** では、メッセージグループ ID 内で順序が保持されます。ただし、順序の保証には、順序付けられていないイベントに比べて、コストが増加する、オペレーションのボトルネックになる可能性があるなどのデメリットもあります。

EventBridge、**Amazon SQS** の標準キューなどの他のサービスでは、ベストエフォートの順序付けが提供されます。この場合、メッセージが正しい順序で受信されないことをアプリケーションで想定しておく必要があります。多くのアプリケーションでは、これは問題になりません。ただし、アプリケーションに順序が必要な場合は、問題を簡単に軽減できます。

順序付けられていないイベント
順序が入れ替わったイベントに対処できるように構築すると、アプリケーションのレジリエンスが向上します。その場合、イベントで順序が入れ替わって送信されてもアプリケーションが失敗することはありません。例えば、アプリケーションで顧客の注文を処理する場合、「OrderUpdated」(注文が更新された) イベントを「OrderCreated」(注文が作成された) イベントの前に受信することはできないようにします。その状況が起きても、「OrderCreated」イベントを受信するために待機している間に「OrderUpdated」イベントが受信されたら、部分的なトランザクションレコードを作成してデータベースで対処できます。不完全なトランザクションの詳細をまとめた運用レポートを作成して、問題の確認と解明を行うことができます。イベントが正しい順序であると想定して、そうでない場合は失敗するのではなく、順序が入れ替わったイベントでも対処できるように構築して、アプリケーションの耐障害性とスケーラビリティを向上させることができます。

セクション 5

まとめ

イベント駆動型アーキテクチャを採用すると、企業全体で俊敏性を高め、拡張性と信頼性に優れたアプリケーションを構築できます。このアプローチは、新たな課題をもたらす可能性がありますが、複数のチームが独立して作業できるようになるため、複雑なアプリケーションを構築する効果的な方法です。

AWS は、イベント駆動型アーキテクチャの構築に役立つ幅広いサーバーレスサービスを提供しています。AWS では、200 を超える AWS サービス、45 以上の SaaS アプリケーション、カスタムアプリケーションのイベントを統合でき、拡張性に優れたイベント駆動型アプリケーションを簡単かつ高速に構築できます。

このガイドでは、イベント駆動型アーキテクチャの概念やベストプラクティスと、チームが AWS でイベント駆動型アーキテクチャを構築するうえで役立つ関連サービスを紹介しました。

セクション 6

リソース

AWS Skill Builder: Architecting Serverless Solutions コース日本語吹き替え版 ›

イベント駆動型アーキテクチャに関するチュートリアルとブログ (英語) ›

AWS ワークショップ : Build a serverless and event-driven Serverlesspresso coffee shop (英語) ›

AWS ワークショップ : Building event-driven architectures on AWS (英語) ›