クラウド設計パターン、アーキテクチャ、および実装

AWS 規範ガイダンス

AWS 規範ガイダンス: クラウド設計パターン、アーキテクチャ、および実装

Table of Contents

序章 1
目標とするビジネス成果 2
腐敗防止層パターン 3
Intent 3
導入する理由 3
適用対象 3
問題点と考慮事項 4
実装 5
高レベルのアーキテクチャ 5
AWS のサービスを使用した実装 6
サンプルコード 7
GitHub リポジトリ 8
関連情報 8
API ルーティングパターン 10
ホスト名ルーティング 10
一般的なユースケース 10
長所 11
短所 11
パスルーティング 12
一般的なユースケース 12
HTTP サービスのリバースプロキシ 12
API Gateway 14
CloudFront 16
HTTP ヘッダールーティング 17
長所 18
短所 18
サーキットブレーカーパターン 19
Intent 19
導入する理由 19
適用対象 20
問題点と考慮事項 20
実装 21
高レベルのアーキテクチャ 21
AWS サービスを使用した実装 22

「サンプルコード」 23
GitHub リポジトリ 24
ブログの参考情報 25
関連情報 25
イベントソーシングパターン 26
Intent 26
導入する理由 26
適用対象 26
問題点と考慮事項 27
実装 28
高レベルのアーキテクチャ 28
AWS のサービスを使用した実装 31
ブログの参考情報 33
六角形アーキテクチャパターン 34
Intent 34
導入する理由 34
適用対象 34
問題点と考慮事項 35
実装 35
高レベルのアーキテクチャ 36
を使用した実装 AWS のサービス 37
「サンプルコード」 38
関連情報 42
動画 42
パブリッシュ – サブスクライブパターン 43
Intent 43
導入する理由 43
適用対象 43
問題点と考慮事項 44
実装 45
高レベルのアーキテクチャ 45
AWS のサービスを使用した実装 46
ワークショップ 48
ブログの参考情報 48
関連情報 48
バックオフパターンで再試行 49

Intent 49
導入する理由 49
適用対象 49
問題点と考慮事項 49
実装 50
高レベルのアーキテクチャ 50
AWS のサービスを使用した実装 50
サンプルコード 51
GitHub リポジトリ 52
関連情報 52
Saga パターン 53
Saga コレオグラフィ 54
Saga オーケストレーション 54
Saga コレオグラフィ 55
Intent 55
導入する理由 56
適用対象 56
問題点と考慮事項 57
実装 58
関連情報 61
Saga オーケストレーション 61
Intent 61
導入する理由 61
適用対象 62
問題点と考慮事項 62
実装 63
ブログの参考情報 67
関連情報 68
動画 68
散布-収集パターン 69
Intent 69
導入する理由 69
適用対象 69
問題点と考慮事項 70
実装 71
高レベルのアーキテクチャ 71

を使用した実装 AWS のサービス 74
ワークショップ 77
ブログの参考情報 77
関連情報 77
ストラングラーフィグパターン 78
Intent 78
導入する理由 78
適用対象 79
問題点と考慮事項 79
実装 80
高レベルのアーキテクチャ 81
AWS サービスを使用した実装 86
ワークショップ 90
ブログの参考情報 90
関連情報 90
トランザクションアウトボックスパターン 91
Intent 91
導入する理由 91
適用対象 91
問題点と考慮事項 91
実装 92
高レベルのアーキテクチャ 92
AWS サービスを使用した実装 93
「サンプルコード」 98
アウトボックステーブルの使用 98
変更データキャプチャ (CDC) の使用 99
GitHub リポジトリ 101
リソース 102
ドキュメント履歴 103
用語集 105
..... 105
A 106
B 108
C 110
D 113
E 117

F 120
G 121
H 122
I 124
L 126
M 127
O 131
P 134
Q 137
R 137
S 140
T 144
U 145
V 146
W 146
Z 147
..... cxlviii

クラウド設計パターン、アーキテクチャ、および実装

Amazon Web Services (AWS)、Anitha Deenadayalan

2024 年 5 月 (ドキュメント履歴)

本ガイドでは、一般的に使用されるモダナイゼーション設計パターンを AWS のサービスを使って実装するためのガイダンスを提供します。スケーラビリティの実現、リリース速度の向上、変更による影響範囲の縮小、リグレッションの軽減を目的として、マイクロサービスアーキテクチャを使用して設計される最新のアプリケーションが増えています。これにより、開発者の生産性が向上し、俊敏性が向上し、イノベーションが促進され、ビジネスニーズへの注力を強化することができます。マイクロサービスアーキテクチャは、サービスとデータベースに最適なテクノロジーの使用もサポートし、多言語コードや多言語永続性を促進します。

従来、モノリシックアプリケーションは単一プロセスで実行され、1 つのデータストアを使用し、垂直方向にスケーリングするサーバー上で実行されていました。それに比べ、最新のマイクロサービスアプリケーションはきめ細かく、独立したフォールトドメインを持ち、ネットワーク全体でサービスとして実行され、ユースケースによっては複数のデータストアを使用できます。サービスは水平方向にスケーリングするため、1 つのトランザクションが複数のデータベースにまたがる場合もあります。開発チームは、マイクロサービスアーキテクチャを使用してアプリケーションを開発する場合、ネットワーク通信、多言語永続性、水平スケーリング、結果整合性、およびデータストア全体のトランザクション処理に重点を置く必要があります。したがって、モダナイゼーションパターンは、最新のアプリケーション開発でよく発生する問題を解決する上で不可欠であり、ソフトウェア配信の迅速化にも役立っています。

本ガイドでは、綿密に設計されたベストプラクティスに基づいて設計パターンに適したクラウドアーキテクチャを選択したいと考えているクラウドアーキテクト、テクニカルリード、アプリケーションオーナー、ビジネスオーナー、および開発者向けのテクニカルリファレンスを提供します。本ガイドで説明する各パターンは、マイクロサービスアーキテクチャにおける 1 つまたは複数の既知のシナリオに対応しています。このガイドでは、各パターンに関連する問題と考慮事項について説明し、高レベルのアーキテクチャ実装を示すほか、パターンに合わせた AWS 実装について説明します。利用可能な場合は、オープンソースの GitHub サンプルとワークショップのリンクが提供されます。

このガイドでは以下のパターンがカバーされています。

- 腐敗防止層
- API ルーティングパターン:
  - ホスト名ルーティング

  - パスルーティング
  - HTTP ヘッダールーティング
- 回路ブレーカー
- イベントソーシング
- 六角形アーキテクチャ
- Publish-subscribe
- バックオフでの再試行
- Saga パターン:
  - Saga コレオグラフィ
  - Saga オーケストレーション
- 散布-収集
- Strangler fig
- トランザクション送信トレイ

目標とするビジネス成果

本ガイドで説明されているパターンを使用してアプリケーションをモダナイズすると、次のことが可能になります。

- コストとパフォーマンスが最適化された、信頼性が高く、セキュアで、運用効率の高いアーキテクチャを設計および実装できます。
- これらのパターンを必要とするユースケースのサイクルタイムを短縮できるため、代わりに組織固有の課題に集中できるようになります。
- AWS のサービスを使用してパターン実装を標準化することにより、開発を加速できます。
- 技術的負債を引き継ぐことなく、開発者が最新のアプリケーションを構築できるように支援できます。

腐敗防止層パターン

Intent

腐敗防止層 (ACL) パターンは、ドメインモデルのセマンティクスをあるシステムから別のシステムに変換する仲介レイヤーとして機能します。これにより、確立されている通信規約がアップストリームチームによって消費される前に、アップストリームの境界コンテキスト (モノリス) のモデルをダウンストリームの境界コンテキスト (マイクロサービス) に適したモデルに変換します。このパターンは、ダウンストリームの境界コンテキストにコアサブドメインが含まれている場合、またはアップストリームモデルが変更不可能なレガシーシステムである場合に適用できます。また、呼び出しをターゲットシステムに透過的にリダイレクトする必要がある場合に呼び出し元が変更されないようにし、変革上のリスクやビジネスの中断を軽減します。

導入する理由

移行プロセスにあるモノリシックアプリケーションをマイクロサービスに移行すると、新しく移行したサービスのドメインモデルセマンティクスが変更される可能性があります。これらのマイクロサービスを呼び出すためにモノリス内の機能が必要な場合は、呼び出し元のサービスを変更せずに、呼び出しを移行済みサービスにルーティングする必要があります。ACL パターンを使用すると、モノリス側では、このパターンが呼び出しを新しいセマンティクスに変換するアダプターまたはファサードレイヤーとして機能し、マイクロサービスを透過的に呼び出すことができます。

適用対象

次の場合は、このパターンの使用を検討してください。

- 既存のモノリシックアプリケーションがマイクロサービスに移行済みの関数と通信する必要があり、移行済みサービスドメインモデルとセマンティクスの機能が元の機能とは異なっている。
- 2 つのシステムのセマンティクスが異なっている状態で、データ交換が必要になっているが、双方のシステムで互換性を確保するのは実用的でない。
- 迅速かつシンプルなアプローチを取ることで、影響を最小限に抑えながら、あるシステムを別のシステムに適応させなければならない。
- 対象のアプリケーションが外部システムと通信している。

問題点と考慮事項

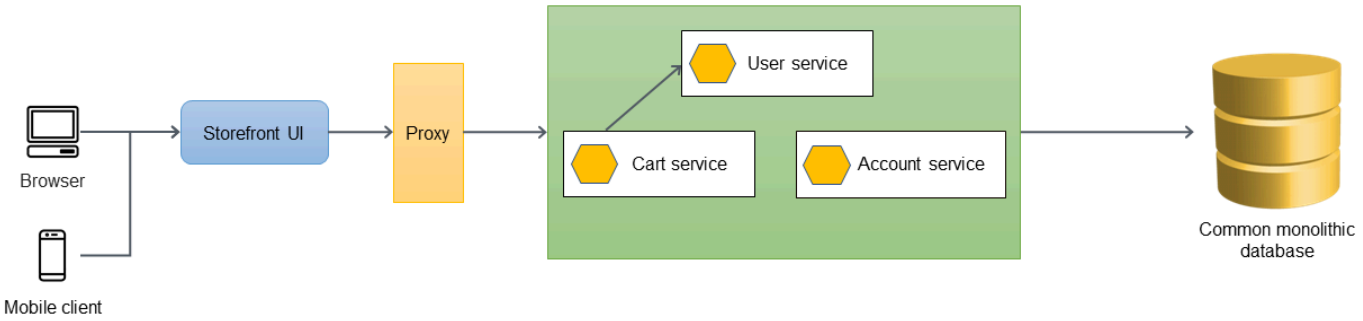
- チームの依存関係: システム内の各種サービスを異なるチームが所有していると、移行済みサービスの新しいドメインモデルセマンティクスが理由で、呼び出し元システムに変更が必要になるかもしれません。しかし、他に優先すべき事項があり、調整しながらはそうした変更を行えない場合があります。ACL を使用すると、呼び出し先を切り離し、呼び出しが新しいサービスのセマンティクスと一致するように変換されるため、呼び出し側で現在のシステムを変更しなくても済みます。
- 運用上のオーバーヘッド: ACL パターンの使用には、運用および保守にさらに労力が必要となります。例えば、ACL を、モニタリングおよびアラートツール、リリースプロセス、継続的インテグレーションおよび継続的デリバリー (CI/CD) プロセスなどと統合する作業も発生するからです。
- 単一障害点: ACL で障害が発生すると、ターゲットサービスへの到達が不可能になり、アプリケーションの問題が発生する可能性があります。この問題を軽減するには、再試行機能とサーキットブレーカーを構築する必要があります。こうしたオプションの詳細については、バックオフでの再試行とサーキットブレーカーの各パターンを参照してください。適切なアラートと口グ記録を設定すると、平均解決時間 (MTTR) が向上します。
- 技術的負債: 移行またはモダナイズ戦略の一環として、ACL が一時的または暫定的なソリューションとなるか、長期的なソリューションとなるかを検討してください。暫定的なものである場合は、ACL を技術的負債として記録し、それに依存する呼び出し元の移行が完了した後に、その ACL を廃止する必要があります。
- レイテンシー: レイヤーを追加すると、あるインターフェイスから別のインターフェイスにリクエストが変換されることで、レイテンシーが発生する可能性があります。本番環境に ACL をデプロイする場合は、その前に、応答時間の影響を受けやすいアプリケーションでパフォーマンス耐性を定義し、テストすると良いでしょう。
- スケーリング上のボトルネック: ピーク負荷までサービスがスケール可能な高負荷アプリケーションでは、ACL がボトルネックになり、スケーリング上の問題が発生する可能性があります。ターゲットサービスがオンデマンドでスケールする場合は、それに応じてスケールが可能となるように ACL を設計する必要があります。
- サービス固有の実装または共有実装: ACL を共有オブジェクトとして設計すると、呼び出しを複数のサービスやサービス固有のクラスに変換およびリダイレクトできます。ACL の実装タイプを決定する際には、レイテンシー、スケーリング、障害耐性を考慮してください。

実装

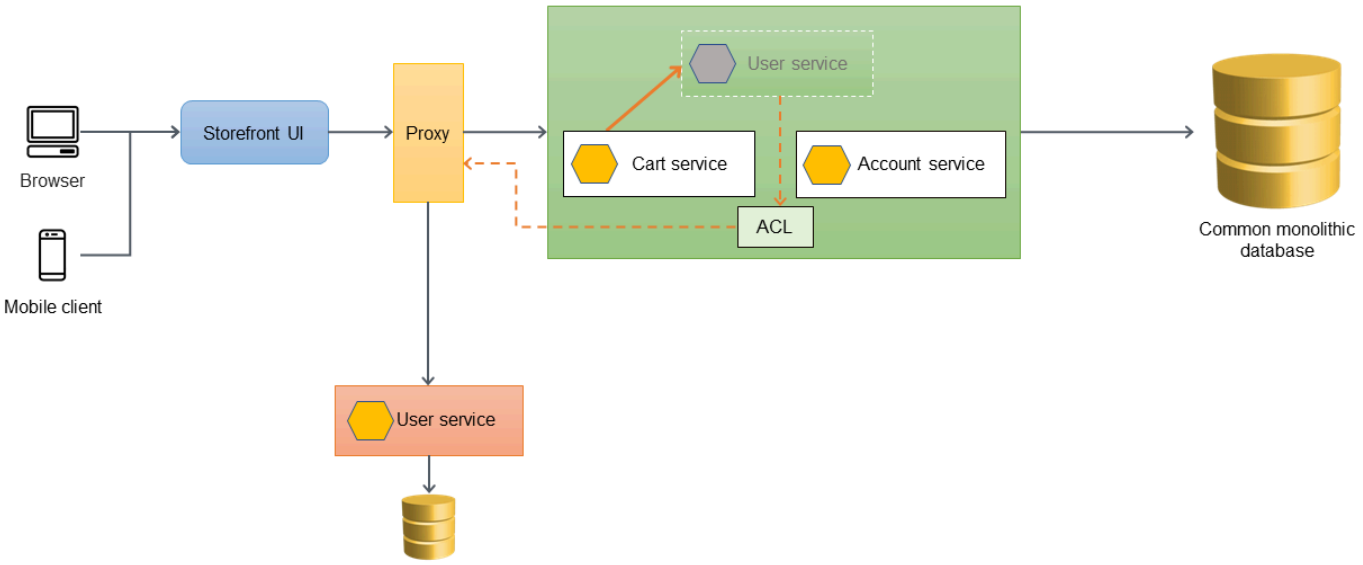
ACL は、移行対象サービスに固有のクラスとして、または独立したサービスとして、モノリシックアプリケーション内に実装できます。また、ACL は、それに依存するサービスをマイクロサービスアーキテクチャにすべて移行した後に廃止する必要があります。

高レベルのアーキテクチャ

次のアーキテクチャ例に示すモノリシックアプリケーションには、ユーザーサービス、カートサービス、アカウントサービスという 3 つのサービスがあります。カートサービスはユーザーサービスに依存しており、アプリケーションではモノリシックリレーショナルデータベースが使用されています。

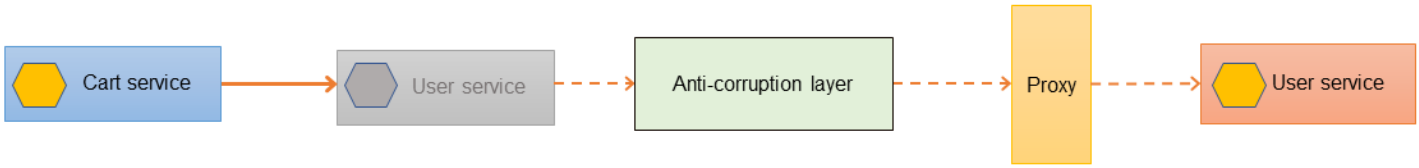


次のアーキテクチャでは、ユーザーサービスが新しいマイクロサービスに移行済みです。カートサービスはユーザーサービスを呼び出しますが、対象サービスはモノリス内で利用できなくなっています。 また、新しく移行したサービスのインターフェイスが、モノリシックアプリケーション内にあった旧サービスのインターフェイスと一致しない可能性もあります。



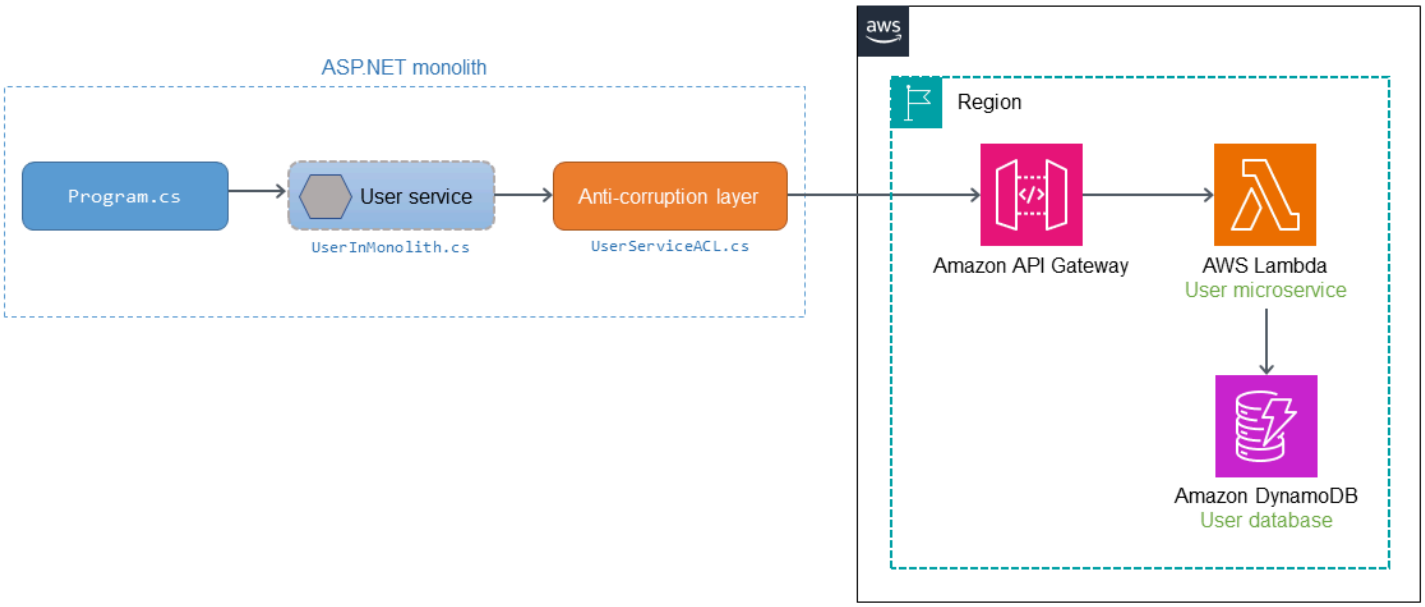
カートサービスが新しく移行したユーザーサービスを直接呼び出す必要がある場合は、カートサービスを変更し、モノリシックアプリケーションを徹底的にテストしなければなりません。これにより、変革上のリスクやビジネスの中断が増える可能性があります。そのため、モノリシックアプリケーションが既に持つ機能への変更を最小限に抑えることを目指すべきです。

この場合、古いユーザーサービスと新しく移行したユーザーサービスの間に ACL を導入すると良いでしょう。ACL は、呼び出しを新しいインターフェイスに変換するアダプターまたはファサードとして機能し、移行済みサービスに固有のクラス (`UserServiceFacade` や `UserServiceAdapter` など) としてモノリシックアプリケーション内に実装できます。また、腐敗防止層は、それに依存するサービスをマイクロサービスアーキテクチャにすべて移行した後に廃止する必要があります。



AWS のサービスを使用した実装

次の図は、AWS サービスを使用してこの ACL の例を実装する方法を示しています。



ユーザーマイクロサービスは、ASP.NET モノリシックアプリケーションから移行し、AWS 上に AWS Lambda 関数としてデプロイします。Lambda 関数への呼び出しは Amazon API Gateway 経由でルーティングします。モノリス内に ACL をデプロイすることで、呼び出しを変換し、ユーザーマイクロサービスのセマンティクスに適合させます。

Program.cs がモノリス内のユーザーサービス (UserInMonolith.cs) を呼び出すと、呼び出しは、ACL (UserServiceACL.cs) にルーティングされます。次に ACL が、呼び出しを新しいセマンティクスとインターフェイスに変換し、さらに、API ゲートウェイエンドポイント経由でマイクロサービスを呼び出します。呼び出し元 (Program.cs) は、ユーザーサービスおよび ACL で実行される変換とルーティングを認識していません。呼び出し元では、コード変更が認識されないため、ビジネスの中断や変革上のリスクが軽減されます。

サンプルコード

次のコードスニペットは、元のサービスへの変更と、UserServiceACL.cs の実装を示しています。リクエストを受信すると、元のユーザーサービスは、ACL を呼び出します。ACL は、ソースオブジェクトを新しく移行したサービスのインターフェイスに合わせて変換し、その後サービスを呼び出して、呼び出し元にレスポンスを返します。

```
public class UserInMonolith: IUserInMonolith
{
    private readonly IACL _userServiceACL;
    public UserInMonolith(IACL userServiceACL) => (_userServiceACL) = (userServiceACL);
    public async Task<HttpStatusCode> UpdateAddress(UserDetails userDetails)
    {
        //Wrap the original object in the derived class
        var destUserDetails = new UserDetailsWrapped("user", userDetails);
        //Logic for updating address has been moved to a microservice
        return await _userServiceACL.CallMicroservice(destUserDetails);
    }
}

public class UserServiceACL: IACL
{
    static HttpClient _client = new HttpClient();
    private static string _apiGatewayDev = string.Empty;

    public UserServiceACL()
    {
        IConfiguration config = new
 ConfigurationBuilder().AddJsonFile(AppContext.BaseDirectory + "../../../
config.json").Build();
        _apiGatewayDev = config["APIGatewayURL:Dev"];
        _client.DefaultRequestHeaders.Accept.Add(new
 MediaTypeWithQualityHeaderValue("application/json"));
    }
    public async Task<HttpStatusCode> CallMicroservice(ISourceObject details)
```

```
    {
        _apiGatewayDev +=  "/" + details.ServiceName;
        Console.WriteLine(_apiGatewayDev);

        var userDetails = details as UserDetails;
        var userMicroserviceModel = new UserMicroserviceModel();
        userMicroserviceModel.UserId = userDetails.UserId;
        userMicroserviceModel.Address = userDetails.AddressLine1 + ", " +
 userDetails.AddressLine2;
        userMicroserviceModel.City = userDetails.City;
        userMicroserviceModel.State = userDetails.State;
        userMicroserviceModel.Country = userDetails.Country;

        if (Int32.TryParse(userDetails.ZipCode, out int zipCode))
        {
            userMicroserviceModel.ZipCode = zipCode;
            Console.WriteLine("Updated zip code");
        }
        else
        {
            Console.WriteLine("String could not be parsed.");
            return HttpStatusCode.BadRequest;
        }

        var jsonString =
 JsonSerializer.Serialize<UserMicroserviceModel>(userMicroserviceModel);
        var payload = JsonSerializer.Serialize(userMicroserviceModel);
        var content = new StringContent(payload, Encoding.UTF8, "application/json");

        var response = await _client.PostAsync(_apiGatewayDev, content);
        return response.StatusCode;
    }
}
```

GitHub リポジトリ

このパターンのサンプルアーキテクチャの完全な実装については、https://github.com/aws-samples/anti-corruption-layer-pattern にある GitHub リポジトリを参照してください。

関連情報

- Strangler fig パターン

- サーキットブレーカーパターン
- バックオフパターンで再試行

API ルーティングパターン

アジャイル開発環境では、自律的なチーム (スクワッズやトライブなど) が多数のマイクロサービスを含む 1 つ以上のサービスを所有します。チームはこれらのサービスを API として公開し、コンシューマーがサービスやアクションのグループとやり取りできるようにします。

ホスト名とパスを使用して HTTP API をアップストリームのコンシューマーに公開するには、主に 3 つの方法があります。

メソッド	説明	例
ホスト名ルーティング	各サービスをホスト名として公開します。	`billing.api.example.com`
パスルーティング	各サービスをパスとして公開します。	`api.example.com/billing`
ヘッダーベースのルーティング	各サービスを HTTP ヘッダーとして公開します。	`x-example-action: something`

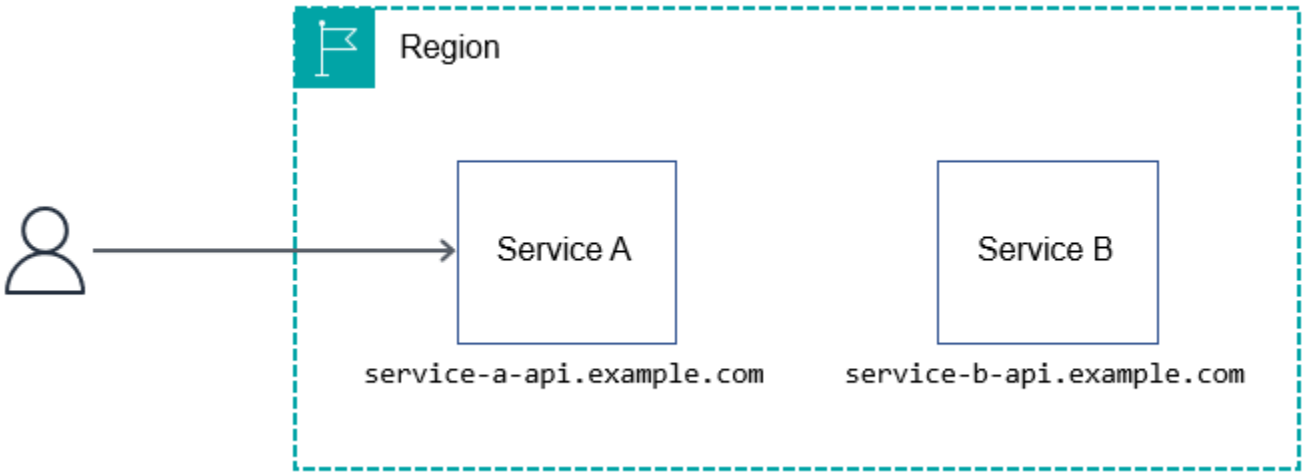
このセクションでは、これら 3 つのルーティング方式の一般的な使用例とそれぞれのトレードオフについて概説し、どの方法が要件と組織構造に最も適しているかを判断できるようにします。

ホスト名ルーティングパターン

ホスト名によるルーティングは、各 API に独自のホスト名 (`service-a.api.example.com` や `service-a.example.com` など) を与えることで API サービスを分離するメカニズムです。

一般的なユースケース

ホスト名を使ったルーティングでは、サービスチーム間で何も共有されないため、リリース時の摩擦が軽減されます。チームは DNS エントリから本番環境でのサービス運用に至るまで、すべてを管理する責任があります。



長所

ホスト名ルーティングは、HTTP API ルーティングのための最も簡単でスケーラブルな方法です。関連する AWS のサービスを使用して、この方法に従ったアーキテクチャを構築できます。Amazon API Gateway、AWS AppSync、Application Load Balancer、Amazon Elastic Compute Cloud (Amazon EC2)、またはその他の HTTP 準拠のサービスを使用してアーキテクチャを作成できます。

チームはホスト名ルーティングを使用してサブドメインを完全に所有できます。また、特定の AWS リージョン またはバージョン (`region.service-a.api.example.com` または `dev.region.service-a.api.example.com` など) では、デプロイの分離、テスト、および調整も容易になります。

短所

ホスト名ルーティングを使用する場合、コンシューマーは公開する各 API とやり取りするために異なるホスト名を覚えておく必要があります。クライアント SDK を提供することで、この問題を軽減できます。ただし、クライアント SDK には独自の課題があります。例えば、ローリングアップデート、複数言語、バージョン管理、セキュリティ問題やバグ修正による重大な変更の伝達、ドキュメントなどがサポートされている必要があります。

ホスト名ルーティングを使用する場合は、新しいサービスを作成するたびにサブドメインまたはドメインを登録する必要があります。

パスルーティングパターン

パスによるルーティングは、複数またはすべての API を同じホスト名でグループ化し、リクエスト URI を使用してサービスを分離する (`api.example.com/service-a` または `api.example.com/service-b` など) メカニズムです。

一般的なユースケース

ほとんどのチームはシンプルなアーキテクチャを必要とするため、この方式を選択します。開発者は HTTP API とやり取りするための URL (`api.example.com` など) を 1 つだけ覚えておく必要があります。API ドキュメントは、異なるポータルや PDF に分割されるのではなく、まとめて保管されることが多いため、理解しやすい場合が多いです。

パスベースのルーティングは HTTP API の共有に適したシンプルなメカニズムと考えられています。ただし、設定、承認、統合、マルチホップによるレイテンシーの増加などの運用上のオーバーヘッドが発生します。設定ミスによってすべてのサービスが中断されないように、成熟した変更管理プロセスも必要です。

AWS では、API を共有して適切なサービスに効果的にルーティングする方法を複数用意しています。以下のセクションでは、HTTP サービスリバースプロキシ、API Gateway、および Amazon CloudFront の 3 つのアプローチについて説明します。API サービスの統合に推奨されているこれらのアプローチはいずれも、AWS で実行されているダウンストリームサービスに依存していません。これらのサービスは、HTTP 互換である限り、どこででも問題なく、またどのテクノロジーでも実行できます。

HTTP サービスのリバースプロキシ

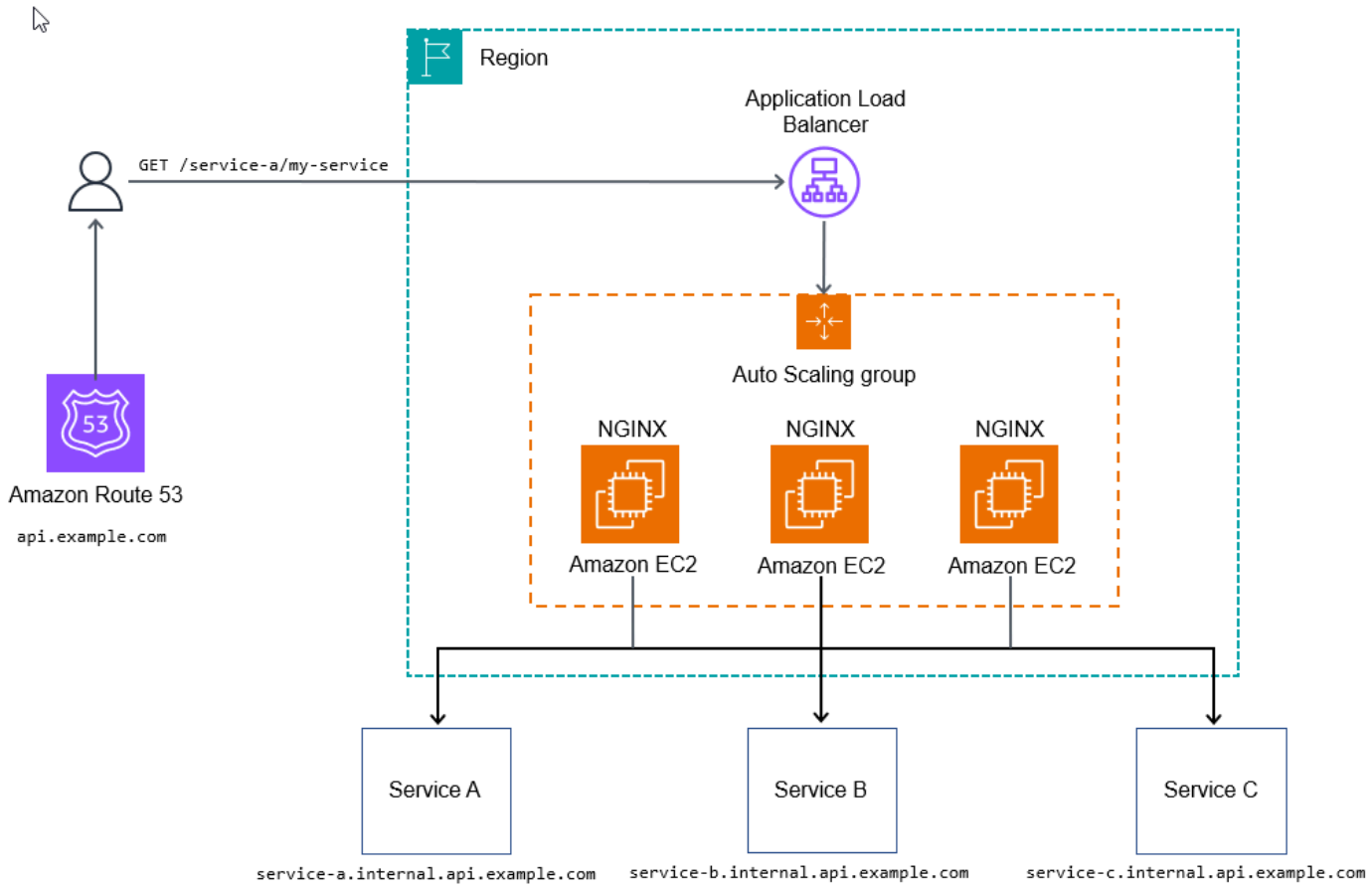
NGINX などの HTTP サーバーを使用して動的ルーティング設定を作成できます。Kubernetes アーキテクチャでは、パスをサービスと一致させるインバウンドトラフィック (イングレス) を作成することもできます。(このガイドでは Kubernetes のイングレスについて説明していません。詳細については、「Kubernetes のドキュメント」を参照してください)

以下の NGINX の設定では、`api.example.com/my-service/` の HTTP リクエストが `my-service.internal.api.example.com` に動的にマッピングされます。

```
server {
    listen  80;
```

```
    location (^/[\w-]+)/(.*) {
        proxy_pass $scheme://$1.internal.api.example.com/$2;
    }
}
```

次の図は、HTTP サービスのリバースプロキシ方式を示しています。



リクエストの処理を開始するために追加の設定を使用せず、ダウンストリーム API でメトリクスとログを収集できる一部のユースケースでは、このアプローチで十分な場合があります。

本番運用の準備を整えるには、スタックのあらゆるレベルにオブザーバビリティを追加したり、設定を追加したり、API のイングレスポイントをカスタマイズするためにスクリプトを追加したりして、レート制限や使用トークンなどのより高度な機能に対応できるようにする必要があります。

長所

HTTP サービスのリバースプロキシ方式の最終的な目的は、API を 1 つのドメインに統合して、どの API コンシューマーにとっても一貫性があるように見せるためのスケーラブルで管理しやすいア

プローチを作成することです。このアプローチにより、サービスチームは、デプロイ後のオーバーヘッドを最小限に抑えながら、独自の API をデプロイおよび管理することもできるようになります。AWS X-Ray や AWS WAF などのトレーシングのための AWS マネージドサービスは、ここでも引き続き適用できます。

短所

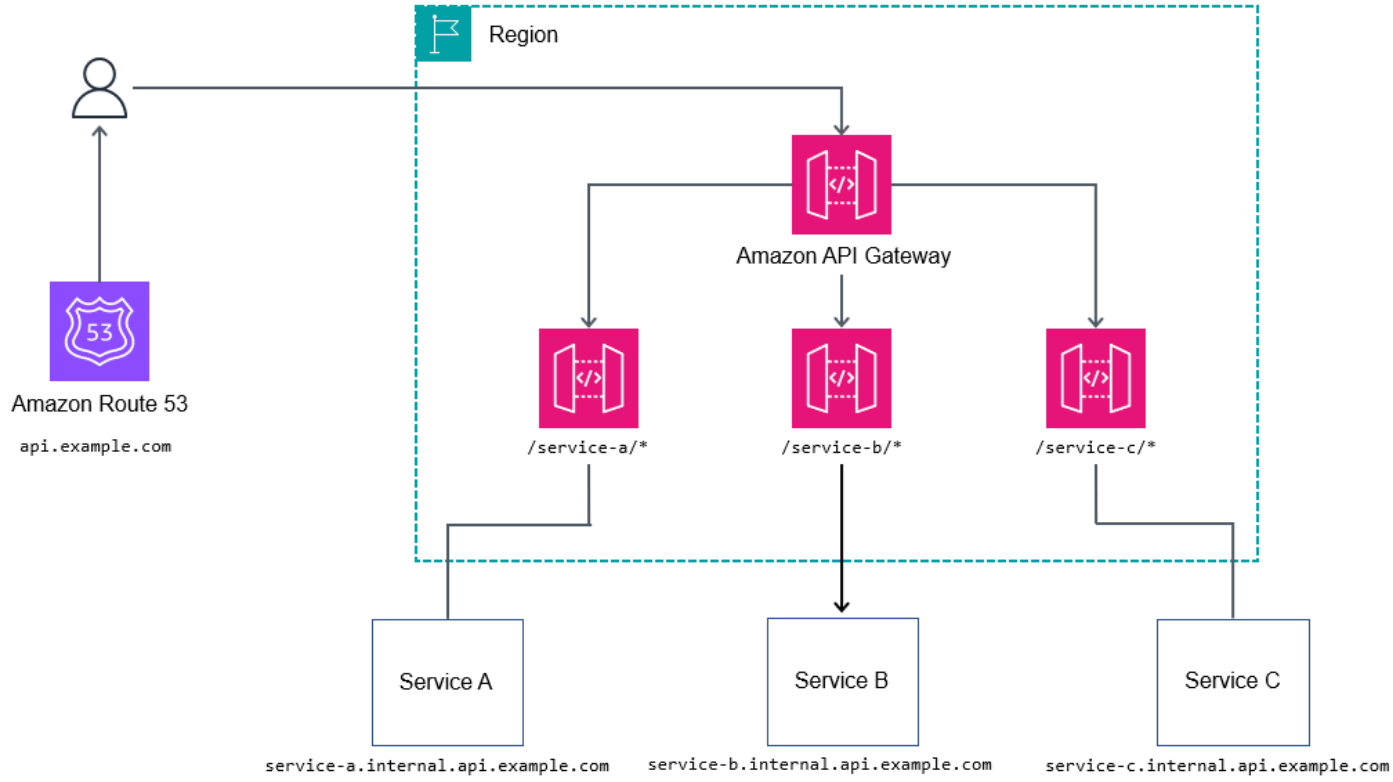
このアプローチの主な欠点は、必要なインフラストラクチャコンポーネントの広範なテストと管理が必要になることですが、サイト信頼性エンジニアリング (SRE) チームが配置されていれば、これは問題にならない可能性があります。

この方式にはコストの転換点があります。データ量が小 ~ 中程度の場合、このガイドで説明する他の方式よりもコストがかかります。データ量が多いと、費用対効果が非常に高くなります (1 秒あたり約 10 万トランザクション以上)。

API Gateway

Amazon API Gateway サービス (REST API および HTTP API) は、HTTP サービスのリバースプロキシ方式と同様の方法でトラフィックをルーティングできます。API ゲートウェイを HTTP プロキシモードで使用すると、簡単に多くのサービスを最上位サブドメイン `api.example.com` へのエントリポイントにラップし、ネストされたサービス (例: `billing.internal.api.example.com`) にリクエストをプロキシできます。

ルートまたはコア API ゲートウェイのすべてのサービスのすべてのパスをマッピングするような、きめ細かな作業は不要な場合があります。代わりに、リクエストを請求サービスに転送するワイルドカードパス (`/billing/*` など) を選択してください。ルートまたはコア API ゲートウェイのすべてのパスをマッピングしないことにより、API を変更するたびにルート API ゲートウェイを更新する必要がなくなるため、API の柔軟性が高まります。



長所

リクエスト属性の変更など、より複雑なワークフローを制御するために、REST API は Apache Velocity Template Language (VTL) を公開して、リクエストとレスポンスを変更できるようにします。REST API には他にも次のようなメリットがあります。

- Auth N/Z with AWS Identity and Access Management (IAM)、Amazon Cognito、または AWS Lambda オーソライザー
- トレーシングのための AWS X-Ray
- との統合AWS WAF
- 基本レート制限
- コンシューマーをさまざまな階層にバケット化するための使用トークン (「API Gateway ドキュメント」の「API リクエストを調整してスループットを向上させる」を参照してください)

短所

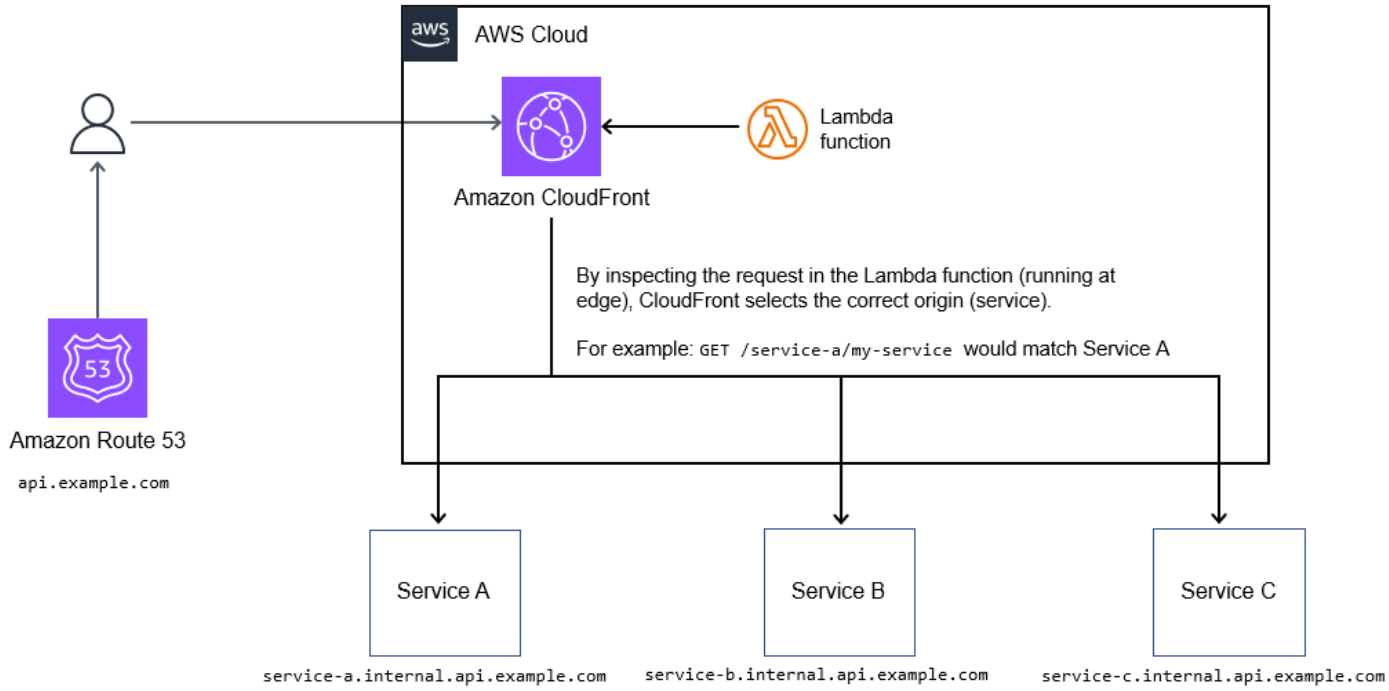
データ量が多いと、ユーザーによってはコストが問題になる場合があります。

CloudFront

Amazon CloudFront の動的オリジン選択機能を使用して、リクエストを転送するオリジン (サービス) を条件付きで選択できます。この機能を使用すると、1 つのホスト名 (`api.example.com` など) で多数のサービスをルーティングできます。

一般的なユースケース

ルーティングロジックは Lambda@Edge 関数内のコードとして存在するため、A/B テスト、canary リリース、機能のフラグ付け、パスの書き換えなど、高度にカスタマイズ可能なルーティングメカニズムをサポートできます。これについては、以下の図で示されています。



長所

API レスポンスをキャッシュする必要がある場合、この方式は単一のエンドポイントの背後に存在するサービスのコレクションを統合するのに適した方法です。これは、API のコレクションを統合するための費用対効果の高い方式です。

また、CloudFront は、フィールドレベルの暗号化に加えて、基本的なレート制限と基本的な ACL のための AWS WAF との統合もサポートしています。

短所

この方式でサポートされる統合可能なオリジン (サービス) は最大で 250 個です。ほとんどのデプロイではこの制限で十分ですが、サービスのポートフォリオが拡大するにつれて、多数の API で問題が発生する可能性があります。

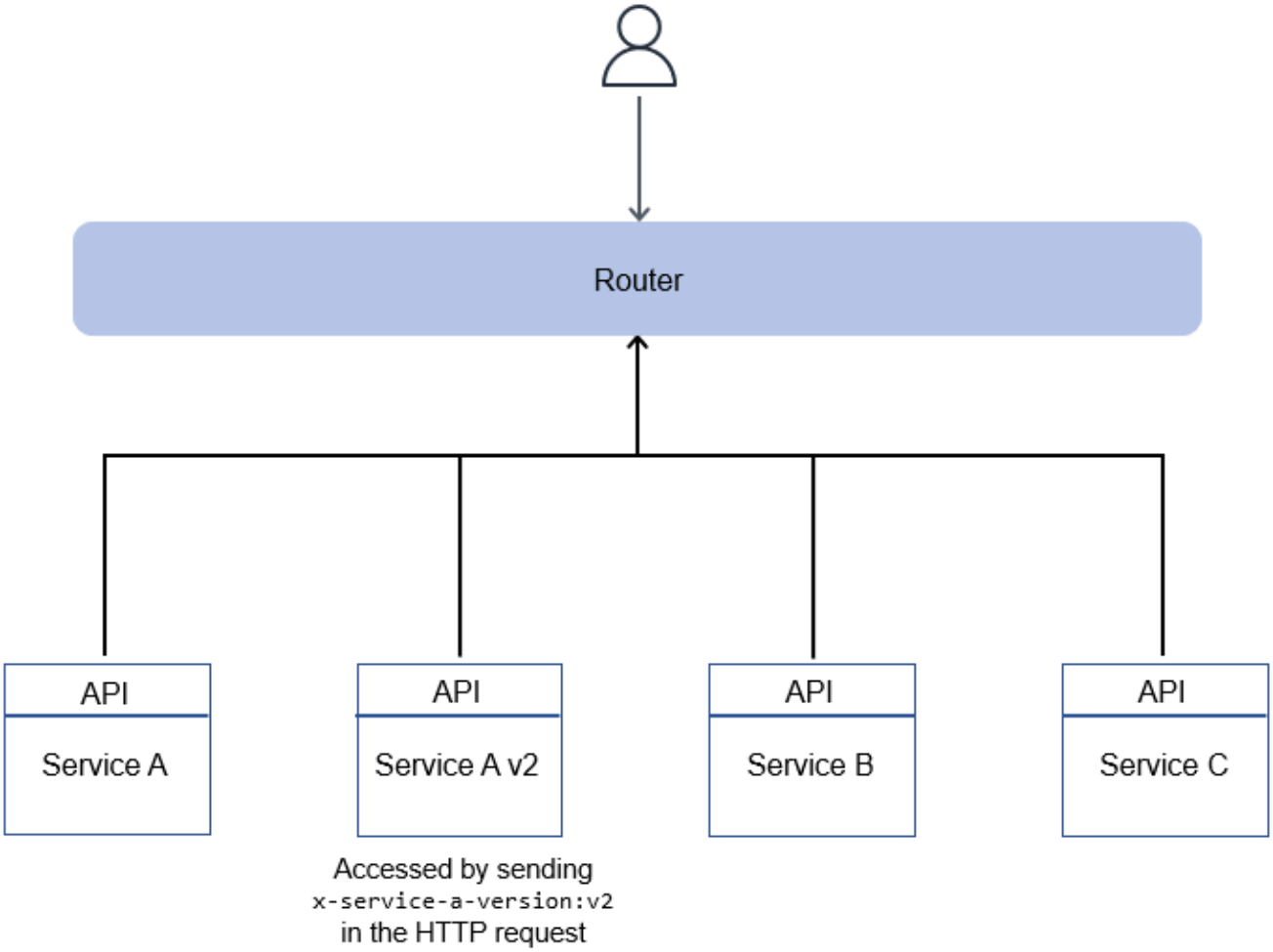
現在、Lambda@Edge 関数の更新には数分かかります。また、CloudFront から、すべての PoP (ポイントオブプレゼンス) への変更の伝達が完了するまでに、最大 30 分かかります。そのため、更新が完了するまで、それ以降の更新は最終的にはブロックされます。

HTTP ヘッダールーティングパターン

ヘッダーベースのルーティングでは、HTTP リクエストの HTTP ヘッダーを指定することで、リクエストごとに正しいサービスをターゲットにすることができます。例えば、ヘッダー `x-service-a-action: get-thing` を送信すると、`Service A` からの `get thing` が可能になります。リクエストのパスは、作業しようとしているリソースのガイダンスを提供するため依然として重要です。

HTTP ヘッダールーティングをアクションに使用するだけでなく、バージョンルーティングのメカニズムとしても使用できるため、機能フラグや A/B テストなどのニーズに対応できます。実際には、ヘッダールーティングを他のルーティング方式のいずれかと併用して、堅牢な API を作成する可能性が高くなります。

HTTP ヘッダールーティングのアーキテクチャでは通常、次の図に示すように、正しいサービスにルーティングして応答を返すマイクロサービスの前に薄いルーティング層が存在します。このルーティング層は、すべてのサービスを対象とすることも、バージョンベースのルーティングなどの操作を可能にする一部のサービスだけを対象とすることもできます。



長所

設定変更の労力は最小限で済み、簡単に自動化できます。この方式は柔軟性も高く、サービスに求める特定の操作だけを公開するクリエイティブな方法をサポートしています。

短所

ホスト名ルーティング方式と同様に、HTTP ヘッダールーティングでは、ユーザーがクライアントを完全に制御でき、カスタム HTTP ヘッダーを操作できることを前提としています。プロキシ、コンテンツ配信ネットワーク (CDN)、およびロードバランサーによって、ヘッダーサイズが制限される場合があります。これが問題になる可能性は低いですが、追加するヘッダーと Cookie の数によっては問題になる可能性があります。

サーキットブレーカーパターン

Intent

サーキットブレーカーパターンを使用すると、呼び出しのタイムアウトまたは失敗が繰り返し発生した場合に、呼び出し元が別のサービス (呼び出し先) への呼び出しを再試行できないようにすることができます。このパターンは、呼び出し先サービスが再び稼働状態になったことを検出するためにも使用されます。

導入する理由

複数のマイクロサービスが連携してリクエストを処理していると、1 つまたは複数のサービスが使用できなくなったり、レイテンシーが大きくなったりする可能性があります。また、複雑なアプリケーションでマイクロサービスを使用している状態で 1 つのマイクロサービスが停止すると、アプリケーションの障害が発生する場合もあるでしょう。マイクロサービスがリモートプロシージャ呼び出しを介して通信しており、ネットワーク接続で一時的なエラーが発生した場合にも、障害が発生する可能性があります。(こうした一時的なエラーは、バックオフパターンで再試行することで処理できます)。さらに、同期実行中にタイムアウトや障害がカスケードされると、ユーザーエクスペリエンスの低下を招きかねません。

しかし、一時的ではなく、解決に時間がかかる障害も発生する可能性があります。例えば、呼び出し先サービスがダウンしていたり、データベースの競合によってタイムアウトが発生したりする場合などです。そのような状況で、呼び出し元サービスが呼び出しを繰り返すと、それらの再試行によってネットワーク競合が発生し、データベーススレッドプールが消費されてしまうでしょう。さらに、複数のユーザーがこのアプリケーションの利用を再試行すると、問題が悪化し、アプリケーション全体のパフォーマンス低下を招きかねません。

サーキットブレーカーパターンは、Michael Nygard 氏の著書「Release It」(2018 年出版) によって、広く知られるようになりました。この設計パターンを使用すると、呼び出しにタイムアウトや失敗が繰り返し生じた呼び出し元による再試行を防ぐことができ、呼び出し先サービスが再び利用可能になったことを検出することも可能です。

サーキットブレーカーオブジェクトは、回路に異常が生じた際に自動的に電流を遮断する電気回路ブレーカーのように動作します。電気回路ブレーカーは、障害発生時に電流を遮断 (トリップ) するものですが、同様に、サーキットブレーカーオブジェクトを呼び出し元サービスと呼び出し先サービスの間に配置すると、呼び出し先サービスが利用できない場合にトリップを実行できます。

Sun Microsystems の Peter Deutsch 氏らは、分散コンピューティングには誤解が見られると主張しており、分散アプリケーションの経験がないプログラマーは、常に誤った前提を立てると述べています。ネットワークの信頼性、ゼロレイテンシーの期待、帯域幅の制限などをポジティブな要因と思い込み、ソフトウェアアプリケーションの開発で、最低限のネットワークエラー処理しか行わないのです。

ネットワーク障害の発生中、アプリケーションは無期限に応答を待ち、自身のリソースを消費し続ける可能性があり、ネットワークが使用可能になったタイミングで操作を再試行しないと、そのパフォーマンスが低下しかねません。また、ネットワークの問題によって、データベースや外部サービスへの API 呼び出しがタイムアウトし、サーキットブレーカーがないために呼び出しが繰り返された場合、コストとパフォーマンスに影響が出ることもあります。

適用対象

このパターンは、次の場合に使用します。

- 呼び出し元サービス側で、失敗の可能性が最も高い呼び出しを行っている。
- 呼び出し先サービスのレイテンシーが大きく (データベース接続が遅いなど) 、呼び出し元サービス側でタイムアウトが発生している。
- 呼び出し元サービス側で同期呼び出しを行っているが、呼び出し先のサービスが利用できないか、レイテンシーが大きい。

問題点と考慮事項

- サービスに依存しない実装: コードの肥大化を防ぐには、マイクロサービスに依存しない API 駆動型の方法でサーキットブレーカーオブジェクトを実装すると良いでしょう。
- 呼び出し先によるサーキットクローズ: 呼び出し先がパフォーマンスの問題や障害から回復したタイミングで、サーキットのステータスを CLOSED に更新できます。これはサーキットブレーカーパターンを拡張したもので、そうした更新が目標復旧時間 (RTO) の要件である場合は実装すると良いでしょう。
- マルチスレッド呼び出し: 有効期限タイムアウト値には、サービスの可用性確認のために呼び出しを再度ルーティングするまでサーキットをトリップ状態にしておく期間を定義しますが、複数のスレッドで呼び出し先サービスを呼び出す場合は、最初に失敗した呼び出しによって有効期限タイムアウト値が定義されます。これを実装する場合は、後続の呼び出しによって有効期限タイムアウトがその後無限に延長されないようにする必要があります。

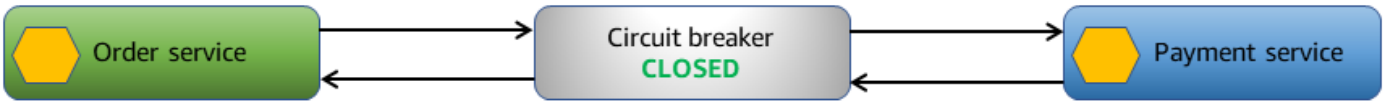
- サーキットの強制開閉: システム管理者がデータベーステーブルの有効期限タイムアウト値を 更新することでサーキットを開閉できるようにしておく必要があります。
- オブザーバビリティ: アプリケーションには ログ記録を設定し、サーキットブレーカーが開いているときに失敗した呼び出しを特定できるようにする必要があります。

実装

高レベルのアーキテクチャ

次の例では、呼び出し元が注文サービスを、呼び出し先が支払いサービスを表しています。

次の図のように、失敗が発生しない場合、サーキットブレーカーは、注文サービスからのすべての呼び出しを支払いサービスにルーティングします。



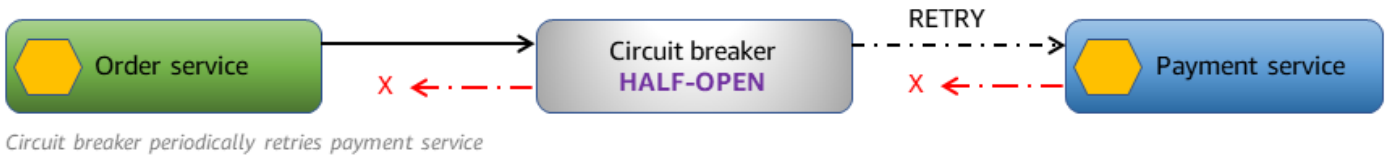
支払いサービスがタイムアウトすると、サーキットブレーカーは、タイムアウトを検出し、失敗を追跡します。

Circuit breaker with payment service failure

タイムアウトが指定したしきい値を超えると、サーキットが開きます。回路が開いている場合、サーキットブレーカーオブジェクトは、呼び出しを支払いサービスにルーティングしません。注文サービスが支払いサービスを呼び出すと、すぐにエラーが返ります。

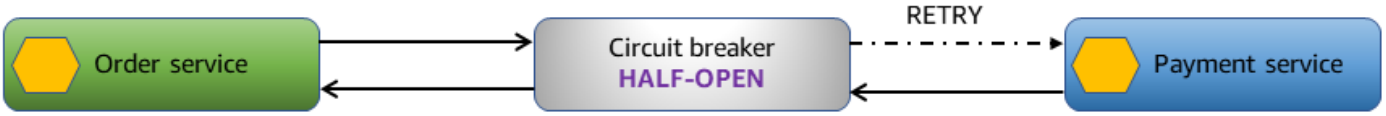
Circuit breaker stops routing to payment service

サーキットブレーカーオブジェクトは、支払いサービスへの呼び出しが成功したかどうかを定期的に確認します。



Circuit breaker periodically retries payment service

支払いサービスの呼び出しが成功すると、回路は閉じ、それ以降のすべての呼び出しが支払いサービスに再度ルーティングされます。



Circuit breaker with working payment service

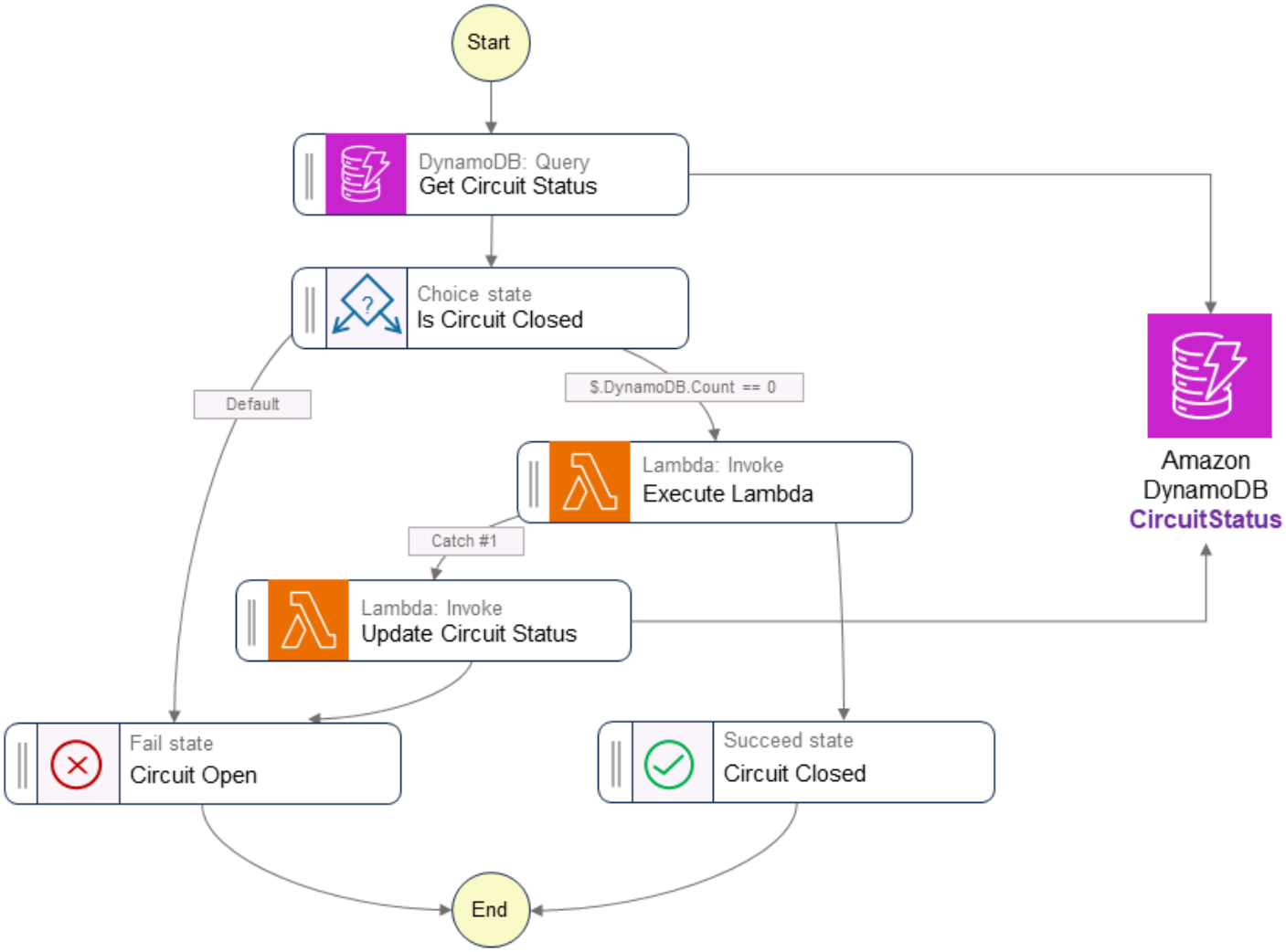
AWS サービスを使用した実装

サンプルソリューションでは、AWS Step Functions ワークフローを使用してサーキットブレーカーパターンを実装しています。Step Functions のステートマシンを使用すると、このパターン実装に必要な再試行機能と意思決定ベースの制御フローを設定できます。

また、このソリューションでは、Amazon DynamoDB テーブルをデータストアに使用してサーキットステータスを追跡しています。パフォーマンスを向上させるには、これを、Amazon ElastiCache (Redis OSS) などのインメモリデータストアに置き換えると良いでしょう。

サービスが別のサービスを呼び出すと、呼び出し先サービスの名前でワークフローが開始されます。ワークフローでは、DynamoDB `CircuitStatus` テーブル (その時点でパフォーマンスが低下しているサービスの格納先) からサーキットブレーカーのステータスを取得します。`CircuitStatus` に、有効期限が切れていない呼び出し先レコードが含まれている場合、サーキットは開いています。Step Functions ワークフローからは、即座に失敗のステータスが返り、ワークフローは `FAIL` ステートで終了します。

`CircuitStatus` テーブルに呼び出し先レコードが含まれていない、または期限切れレコードが含まれている場合は、サービスは稼働状態にあります。ステートマシン定義内の `ExecuteLambda` ステップが、パラメータ値を介して送信される Lambda 関数を呼び出します。呼び出しが成功した場合、Step Functions ワークフローは `SUCCESS` ステートで終了します。



サービスの呼び出しが失敗するか、タイムアウトが発生すると、定義した回数だけ エクスポネンシャルバックオフによる再試行が発生します。再試行後もサービスの呼び出しに失敗する場合、対象サービスの `CircuitStatus` テーブルに `ExpiryTimeStamp` を持つレコードが挿入され、ワークフローは `FAIL` ステートで終了します。同じサービスがその後も呼び出されると、サーキットブレーカーが開いている限り、即座に失敗が返ります。また、ステートマシン定義内の `Get Circuit Status` ステップによって、`ExpiryTimeStamp` 値を基にサービスの可用性が確認され、DynamoDB の有効期限 (TTL) 機能により、期限切れ項目が `CircuitStatus` テーブルから削除されます。

「サンプルコード」

次のコードでは、`GetCircuitStatus` Lambda 関数を使用してサーキットブレーカーのステータスを確認しています。

```
var serviceDetails = _dbContext.QueryAsync<CircuitBreaker>(serviceName,
 QueryOperator.GreaterThan,
                new List<object>
                    {currentTimeStamp}).GetRemainingAsync();

if (serviceDetails.Result.Count > 0)
{
    functionData.CircuitStatus = serviceDetails.Result[0].CircuitStatus;
}
else
{
    functionData.CircuitStatus = "";
}
```

次のコードは、Step Functions ワークフローの Amazon States Language ステートメントを示しています。

```
"Is Circuit Closed": {
    "Type": "Choice",
    "Choices": [
    {
        "Variable": "$.CircuitStatus",
        "StringEquals": "OPEN",
        "Next": "Circuit Open"
    },
    {
        "Variable": "$.CircuitStatus",
        "StringEquals": "",
        "Next": "Execute Lambda"
    }
    ]
},
"Circuit Open": {
    "Type": "Fail"
}
```

GitHub リポジトリ

このパターンのサンプルアーキテクチャの完全な実装については、https://github.com/aws-samples/circuit-breaker-netcore-blog の GitHub リポジトリを参照してください。

ブログの参考情報

- AWS Step Functions および Amazon DynamoDB でのサーキットブレーカーパターンの使用

関連情報

- Strangler fig パターン
- バックオフパターンで再試行

イベントソーシングパターン

Intent

イベント駆動型アーキテクチャでは、イベントソーシングパターンが、状態変化を引き起こすイベントをデータストアに保存します。これにより、状態変化の詳細な履歴をキャプチャして保持できるようになり、監査性、トレーサビリティ、および過去の状態を分析する機能が向上します。

導入する理由

複数のマイクロサービスが連携してリクエストを処理することができますが、その際イベントを通じて通信を行います。これらのイベントが状態 (データ) の変化を引き起こす可能性があります。イベントオブジェクトを発生順に格納すると、データエンティティの現在の状態に関する貴重な情報と、その状態にどのように到達したかについての追加情報が得られます。

適用対象

イベントソーシングパターンは次の場合に使用します。

- 追跡には、アプリケーションで発生したイベントのイミュータブルな履歴が必要です。
- 信頼できる唯一の情報源 (SSOT) からの多言語データ射影が必要です。
- アプリケーションの状態をポイントインタイムで再構築する必要があります。
- アプリケーションの状態を長期間保存する必要はありませんが、必要に応じて再構築することもできます。
- ワークロードの読み込みボリュームと書き込みボリュームは異なります。例えば、リアルタイム処理を必要としない、書き込み集約型のワークロードがあるとします。
- アプリケーションのパフォーマンスやその他のメトリクスを分析するには、変更データキャプチャ (CDC) が必要です。
- 報告やコンプライアンスの目的のためには、システム内で発生するすべてのイベントについて監査データが必要です。
- 再生プロセス中にイベントを変更 (挿入、更新、または削除) して What-If シナリオを導き出し、考えられる終了状態を判断する必要があります。

問題点と考慮事項

- オプティミスティックコンカレンシーコントロール: このパターンでは、システムの状態変化を引き起こすすべてのイベントが保存されます。複数のユーザーまたはサービスが同じデータを同時に更新しようとすると、イベントの衝突が発生する可能性があります。このような衝突は、競合するイベントが同時に作成されて適用された場合に発生し、最終的なデータの状態は現実と一致しないものになってしまいます。この問題を解決するために、イベントの衝突を検出して解決するストラテジーを実装します。例えば、バージョニングを含めたり、イベントにタイムスタンプを追加して更新の順序を追跡したりすることで、オプティミスティックコンカレンシーコントロールのスキームを実装できます。
- 複雑さ: イベントソーシングを実装するには、従来の CRUD 操作からイベント駆動型思考へと考え方を転換する必要があります。システムを元の状態に戻す場合に使用される再生プロセスは、データの冪等性を確保するために複雑になる可能性があります。イベントストレージ、バックアップ、およびスナップショットも、さらに複雑さを増す可能性があります。
- 結果整合性: コマンドクエリ責任分離 (CQRS) パターンまたはマテリアライズドビューを使用してデータを更新する際にレイテンシーが発生するため、イベントから取得されたデータ射影は結果的に整合します。コンシューマーがイベントストアからのデータを処理し、パブリッシャーが新しいデータを送信すると、データ射影またはアプリケーションオブジェクトが現在の状態を表していない場合があります。
- クエリ実行: イベントログからの最新データまたは集計データの取得は、特に複雑なクエリやレポート作成タスクの場合、従来のデータベースに比べて複雑で時間がかかる可能性があります。この問題を軽減するために、イベントソーシングは多くの場合 CQRS パターンで実装されます。
- イベントストアのサイズとコスト: 特にイベントスループットが高いシステムや保持期間が長いシステムでは、イベントが継続的に永続化されるため、イベントストアのサイズが急激に増加する可能性があります。したがって、イベントストアが大きくなり過ぎないように、イベントデータを費用対効果の高いストレージに定期的にアーカイブする必要があります。
- イベントストアのスケーラビリティ: イベントストアでは、大量の書き込み操作と読み込み操作の両方が効率的に処理される必要があります。イベントストアのスケーリングは難しい場合があるため、シャードとパーティションを提供するデータストアを用意することが重要です。
- 効率性と最適化: 書き込み操作と読み込み操作の両方が効率的に処理されるイベントストアを選択または設計します。イベントストアは、アプリケーションの期待されるイベントボリュームとクエリパターンに合わせて最適化する必要があります。インデックス作成とクエリのメカニズムを実装すると、アプリケーションの状態を再構築する際のイベントの取得を高速化できます。クエリ最適化機能を備えた、専用のイベントストアデータベースまたはライブラリの使用を検討することもできます。

- スナップショット: 時間ベースのアクティベーション機能で定期的にイベントログをバックアップする必要があります。前回正常に実行されたデータのバックアップに関するイベントを再生すると、アプリケーションの状態に対してポイントインタイムの復旧が行われます。目標復旧時点 (RPO) は、最後のデータ復旧時点からの最大許容時間です。RPO は、最後の復旧時点からサービスが中断されるまでの間の許容可能なデータ損失量を決定します。データストアとイベントストアの日次スナップショットの頻度は、アプリケーションの RPO に基づいて決定する必要があります。
- 時間依存性: イベントは発生した順序で保存されます。したがって、ネットワークの信頼性は、このパターンを実装する際に考慮すべき重要な要素です。レイテンシーの問題により、不正確なシステム状態が発生する可能性があります。イベントをイベントストアに渡すには、先入れ先出し (FIFO) キューを使用して 1 回限りの配信を行います。
- イベント再生パフォーマンス: 現在のアプリケーションの状態を再構築する場合、相当数のイベントの再生に時間がかかる可能性があります。特にアーカイブされたデータからイベントを再生する場合は、パフォーマンスを向上させるための最適化作業が必要です。
- 外部システムの更新: イベントソーシングパターンを使用するアプリケーションは、外部システムのデータストアを更新し、その更新をイベントオブジェクトとしてキャプチャする場合があります。外部システムの更新を想定していない場合、イベントの再生中にこの動作が問題になることがあります。このような場合は、機能フラグを使用して外部システムの更新を制御できます。
- 外部システムのクエリ: 外部システムコールが呼び出しの日付と時刻の影響を受ける場合は、受信したデータを内部データストアに保存して再生中に使用できます。
- イベントのバージョン管理: アプリケーションの進化に伴い、イベントの構造 (スキーマ) が変わる可能性があります。イベントのバージョン管理戦略を実装して、下位互換性と上位互換性を確保することが必要です。これには、イベントペイロードにバージョンフィールドを含めること、および再生中にさまざまなイベントバージョンを適切に処理することが伴います。

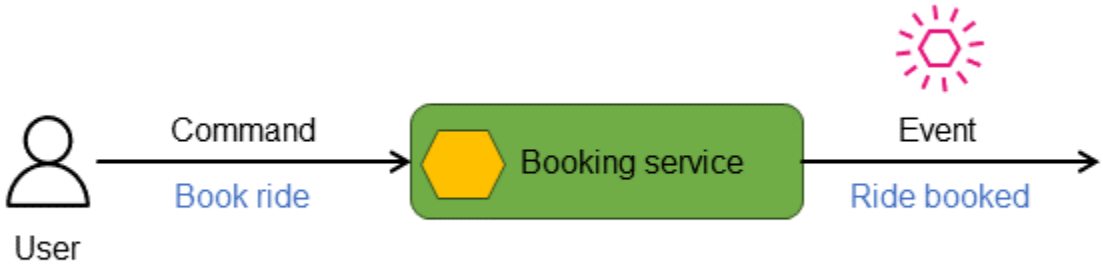
実装

高レベルのアーキテクチャ

コマンドとイベント

分散型のイベント駆動型マイクロサービスアプリケーションでは、コマンドはサービスに送信される命令やリクエストを表し、通常はサービスの状態の変更を開始することを目的としています。サービスはこれらのコマンドを処理し、コマンドの有効性と現在の状態への適用性を評価します。コマンドが正常に実行されると、サービスは実行されたアクションと関連する状態情報を示すイベントを発行

することによって応答します。例えば、以下の図では、予約サービスは「Ride booked」(乗車予約済み) イベントを発行することによって「Book ride」(乗車予約) コマンドに応答しています。



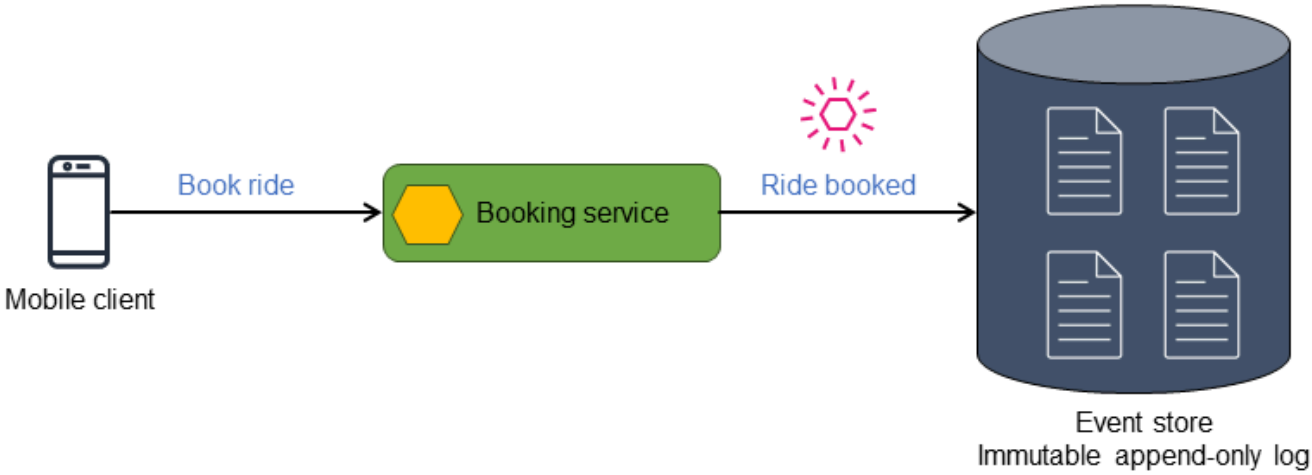
イベントストア

イベントは、イベントストアと呼ばれる、変更不能で追加専用の、時系列順に並べられたリポジトリまたはデータストアにログ記録されます。状態の変化はそれぞれ、個別のイベントオブジェクトとして扱われます。初期状態、現在の状態、および任意のポイントインタイムビューがわかっているエンティティオブジェクトまたはデータストアは、イベントを発生順に再生することで再構築できます。

イベントストアは、すべてのアクションと状態変化の履歴レコードとして機能し、価値ある「信頼できる唯一の情報源」としての役割を果たします。イベントストアを使用してイベントを再生プロセッサに渡すことで、システムの最終的かつ最新の状態を取得できます。再生プロセッサは、これらのイベントを適用して最新のシステム状態の正確な表現を生成する機能を備えています。また、イベントストアを使用して、状態のポイントインタイム視点を生成することもできます。これは、再生プロセッサを介してイベントを再生することにより実現できます。イベントソーシングパターンでは、最新のイベントオブジェクトが現在の状態を完全に表現しているとは限らない場合があります。現在の状態は、次の 3 つの方法のいずれかで取得できます。

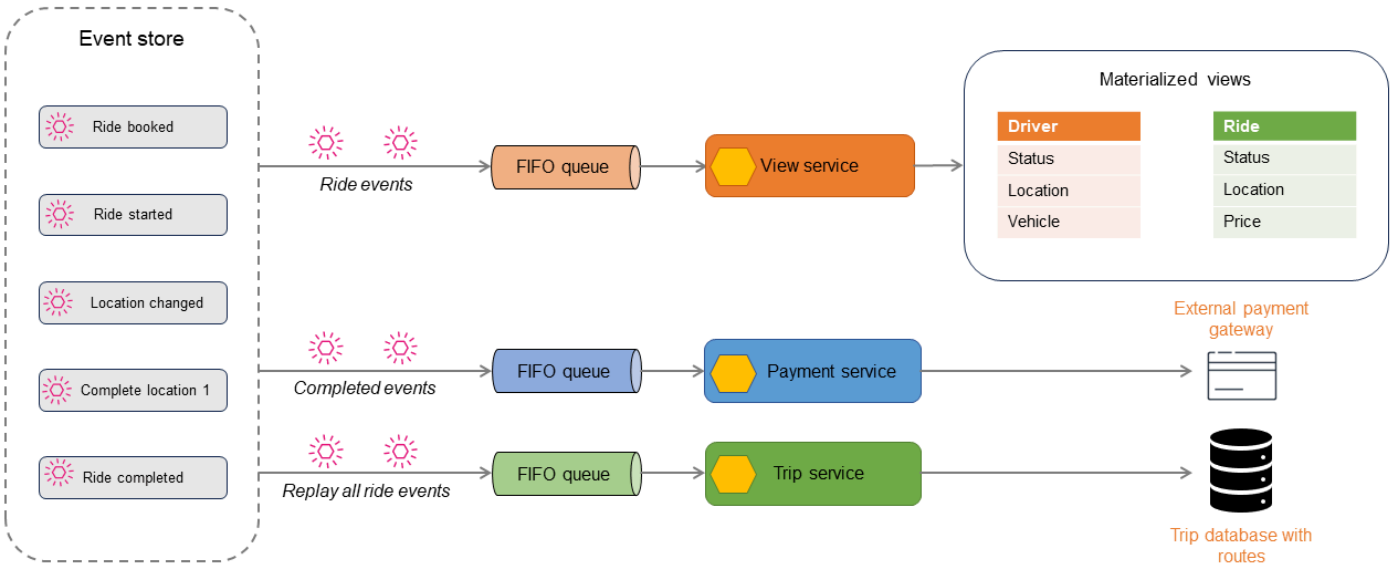
- 関連イベントを集計する。関連するイベントオブジェクトが結合されて、クエリ用の現在の状態が生成されます。この方法は、イベントが結合されて読み取り専用のデータストアに書き込まれるという点で、多くの場合 CQRS パターンと組み合わせて使用されます。
- マテリアライズドビューを使用する。マテリアライズドビューパターンで実装されたイベントソーシングを使用して、イベントデータを計算または要約すると、関連データの現在の状態を取得できます。
- イベントを再生する。イベントオブジェクトを再生すると、現在の状態を生成するアクションを実行できます。

次の図は、イベントストアに保存されている Ride booked イベントを示しています。



イベントストアが保存したイベントを公開すると、そのイベントはフィルタリングされて適切なプロセッサにルーティングされ、後続のアクションで使用できるようになります。例えば、イベントをビュープロセッサにルーティングすると、ビュープロセッサは状態を要約してマテリアライズドビューを表示します。イベントは、ターゲットデータストアのデータ形式に変換されます。このアーキテクチャを拡張すると、さまざまなタイプのデータストアを派生させることができ、データの多言語永続性につながります。

次の図は、乗車予約アプリケーションのイベントを説明しています。アプリケーション内で発生するすべてのイベントは、イベントストアに保存されます。保存されたイベントはフィルタリングされ、異なるコンシューマーにルーティングされます。



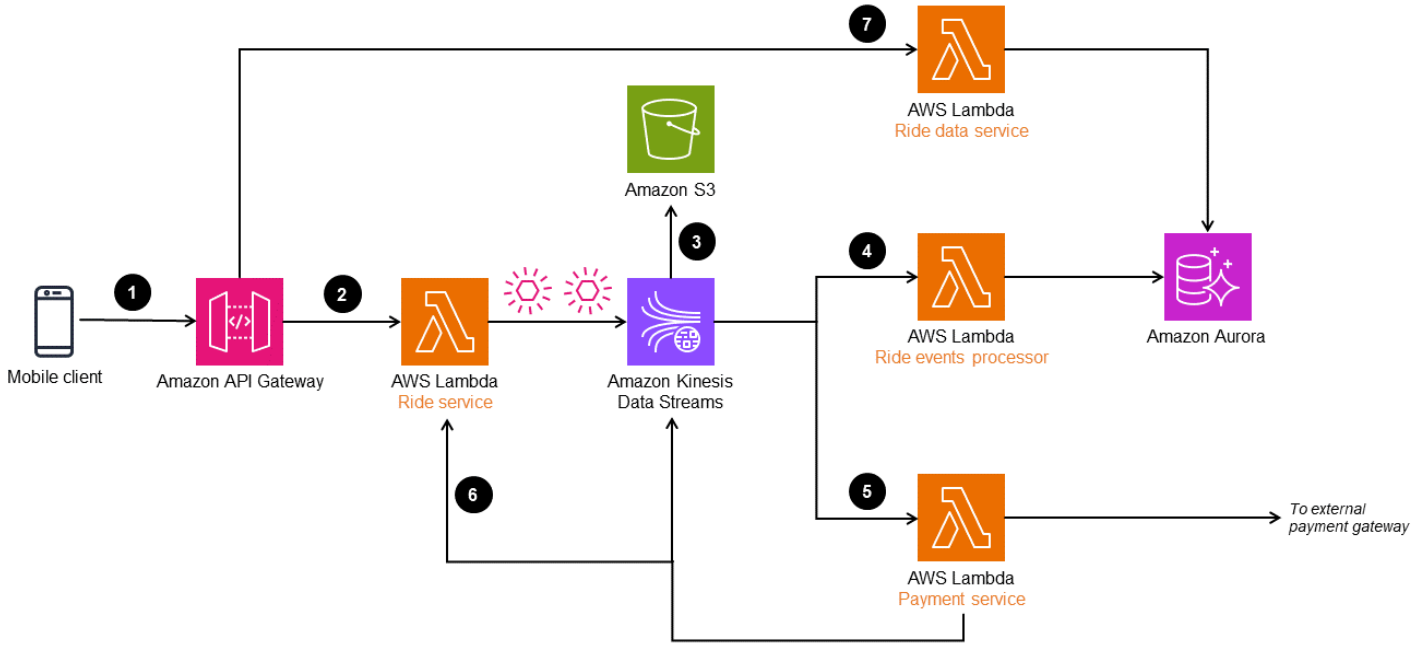
乗車のイベントで、CQRS またはマテリアライズドビューパターンを使用して読み取り専用のデータストアを生成できます。読み込まれたストアにクエリを実行すると、乗車、ドライバー、または予約の現在の状態を取得できます。`Location changed` や `Ride completed` などの一部のイベントは、支払い処理のために別のコンシューマーに公開されます。乗車が完了すると、すべての乗車イベントが再生され、監査または報告の目的で乗車の履歴が作成されます。

イベントソーシングパターンは、ポイントインタイムの復元が必要なアプリケーションでよく使用されますが、信頼できる唯一の情報源を使ってデータをさまざまな形式で射影する必要がある場合にも頻繁に使用されます。どちらの操作でも、イベントを実行して必要な終了状態を取得するための再生プロセスが必要です。また、再生プロセッサには既知の開始点が必要な場合もあります。効率的なプロセスという点では、アプリケーションの起動からではないことが理想です。システム状態のスナップショットを定期的に作成し、より少ない数のイベントを適用して最新の状態を取得することをお勧めします。

AWS のサービスを使用した実装

以下のアーキテクチャでは、Amazon Kinesis Data Streams がイベントストアとして使用されています。このサービスは、アプリケーションの変更をイベントとしてキャプチャして管理し、高スループットでリアルタイムのデータストリーミングソリューションを提供します。AWS でイベントソーシングパターンを実装する場合、アプリケーションのニーズに応じて Amazon EventBridge や Amazon Managed Streaming for Apache Kafka (Amazon MSK) などのサービスを使用することもできます。

耐久性を高め、監査を有効にする場合は、Amazon Simple Storage Service (Amazon S3) の Kinesis Data Streams によってキャプチャされたイベントをアーカイブします。このデュアルストレージアプローチは、将来の分析やコンプライアンスの目的に備えて、過去のイベントデータを安全に保持するのに役立ちます。



ワークフローの手順は以下のとおりです。

1. 乗車予約リクエストは、モバイルクライアントを介して Amazon API Gateway エンドポイントに送信されます。
2. 乗車マイクロサービス (`Ride service` Lambda 関数) はリクエストを受け取り、オブジェクトを変換して Kinesis Data Streams に公開します。
3. Kinesis Data Streams のイベントデータは、コンプライアンスおよび監査履歴の目的で Amazon S3 に保存されます。
4. イベントは `Ride event processor` Lambda 関数によって変換および処理され、Amazon Aurora データベースに保存されて、乗車データのマテリアライズドビューが提供されます。
5. 完了した乗車イベントはフィルタリングされ、支払い処理のために外部の支払いゲートウェイに送信されます。支払いが完了すると、別のイベントが Kinesis Data Streams に送信され、乗車データベースが更新されます。
6. 乗車が完了すると、乗車イベントが `Ride service` Lambda 関数に再生され、ルートと乗車履歴が作成されます。

7. 乗車情報は、Aurora データベースから読み込まれる `Ride data service` を通じて読み込むことができます。

また、API Gateway は、`Ride service` Lambda 関数を使用しなくても、イベントオブジェクトを Kinesis Data Streams に直接送信することができます。ただし、配車サービスのような複雑なシステムでは、イベントオブジェクトをデータストリームに取り込む前に処理して強化する必要がある場合があります。このため、アーキテクチャには、Kinesis Data Streams に送信する前にイベントを処理する `Ride service` が存在します。

ブログの参考情報

- AWS Lambda 向けの新機能 – イベントソースとしての SQS FIFO

六角形アーキテクチャパターン

Intent

六角形アーキテクチャパターンは、ポートおよびアダプターパターンとも呼ばれ、2005 年に Alistair Cockburn 博士によって提唱された手法であり、データストアやユーザーインターフェイス (UI) に依存せず、アプリケーションコンポーネントを独立してテストできる疎結合アーキテクチャの構築を目的にしています。このパターンを使用すると、データストアおよび UI の技術的ロックインを防ぎやすくなるため、長期的な観点から技術スタックを容易に変更できるうえ、ビジネスロジックへの影響もほとんど、あるいはまったく生じません。この疎結合アーキテクチャのアプリケーションでは、ポートというインターフェイスを介して外部コンポーネントと通信すると共に、アダプターによってこうしたコンポーネント間でやり取りされる技術情報が翻訳されます。

導入する理由

六角形アーキテクチャパターンを使用すると、ビジネスロジック (ドメインロジック) を、関連インフラストラクチャコード (データベースや外部 API にアクセスするコード) から分離できます。このパターンは、外部サービスとの統合を必要とする AWS Lambda 関数用の疎結合のビジネスロジックとインフラストラクチャコードを作成するのに役立ちます。従来のアーキテクチャでは、一般的に、ビジネスロジックを、ストアドプロシージャとしてデータベース層に埋め込み、ユーザーインターフェイスにも埋め込んでいました。また、ビジネスロジック内で UI 固有のコンストラクトを使用することで、密結合のアーキテクチャが形成され、これによって、データベース移行やユーザーエクスペリエンス (UX) のモダナイズに取り組む際にボトルネックが生じていました。六角形アーキテクチャパターンにより、技術別ではなく、目的別にシステムやアプリケーションを設計できるうえ、データベース、UX、サービスといったアプリケーションコンポーネントを簡単に交換することも可能です。

適用対象

六角形アーキテクチャパターンは、次の場合に使用します。

- アプリケーションアーキテクチャを切り離して、完全にテストできるコンポーネントを作成する必要がある。
- 複数のクライアントタイプに同じドメインロジックを使用できる。
- UI およびデータベースコンポーネントに、アプリケーションロジックに影響を与えることなく定期的な技術更新を行う必要がある。

- アプリケーションに複数の入力プロバイダーと出力コンシューマーが必要なため、アプリケーションロジックをカスタマイズするとコードが複雑化し、拡張が難しくなる。

問題点と考慮事項

- ドメイン駆動型設計: 六角形アーキテクチャは、とりわけ、ドメイン駆動型設計 (DDD) の場合に効果を発揮します。各アプリケーションコンポーネントを DDD のサブドメインとして表し、六角形アーキテクチャを適用すると、アプリケーションコンポーネント間の疎結合を実現できます。
- テスト可能性: 六角形アーキテクチャでは、設計上、入力と出力に抽象化を使用するため、この手法特有の疎結合によって、単体テストの作成や独立したテストの実行が容易になります。
- 複雑さ: ビジネスロジックをインフラストラクチャコードから分離するのは複雑なプロセスですが、これに慎重に対処できれば、俊敏性、テストカバレッジ、技術的適応性といった大きな利点を得られる可能性があります。そのように対処できない場合は、問題解決が複雑になりかねません。
- メンテナンス上のオーバーヘッド: アーキテクチャをプラグイン可能にするアダプターコードの追加が、理にかなっているのは、アプリケーションコンポーネントに複数の入力元と、複数の出力書き込み先が必要な場合か、入力データストアと出力データストアを時間の経過と共に変更する必要がある場合のみです。それ以外の場合は、このアダプターによってメンテナンス対象のレイヤーが増えることになり、メンテナンス上のオーバーヘッドが発生します。
- レイテンシーの問題: ポートとアダプターを使用すると、レイヤーをさらに追加することになるため、レイテンシーが発生する可能性があります。

実装

六角形アーキテクチャを適用すると、アプリケーションとビジネスロジックを、インフラストラクチャコードから分離すると同時に、アプリケーションを UI、外部 API、データベース、メッセージブローカーと統合するコードからも分離できます。また、ポートとアダプターを介して、ビジネスロジックコンポーネントを、アプリケーションアーキテクチャの他のコンポーネント (データベースなど) に簡単に接続することも可能です。

ポートは、アプリケーションコンポーネントに接続する技術に依存しないエントリポイントとして機能します。こうしたカスタムインターフェイスによって、外部アクターがアプリケーションコンポーネントと通信可能となるインターフェイスが決定します。そのインターフェイスを実装するユーザーやシステムを問わずそのように動作します。こうした動作は、USB ポートで USB アダプターを使用している限り、各種デバイスがコンピュータと通信できるようになるのと似ています。

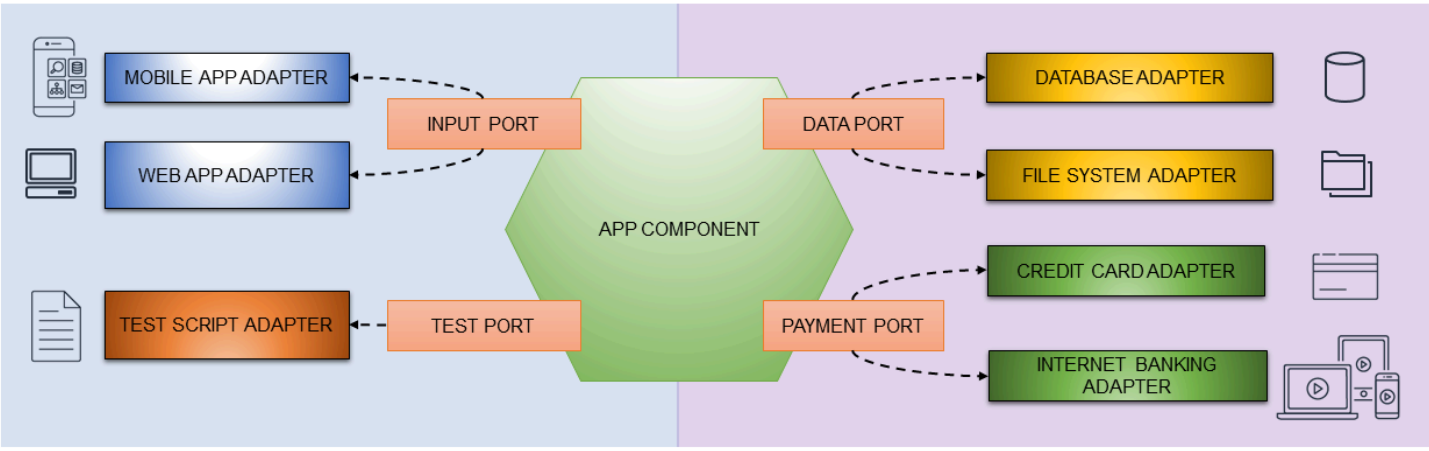
アダプターは、特定の技術を使用して、ポート経由でアプリケーションとやり取りしており、こうしたポートに接続して、ポートからのデータ受信、ポートへのデータ提供、処理に必要なデータ変換を行っています。例えば、REST アダプターを使用すると、アクターが REST API を介してアプリケーションコンポーネントと通信できるようになります。ポートには、ポートやアプリケーションコンポーネントへのリスクなしに、複数のアダプターを追加できます。前の例を拡張するために、同じポートに GraphQL アダプターを追加すると、GraphQL API を介してアプリケーションとやり取りするという別の手段をアクターに提供できます。その場合でも、REST API、ポート、アプリケーションへの影響はありません。

ポートはアプリケーションに接続する機能を、アダプターは外部に接続する機能を備えています。ポートを使用すると、疎結合のアプリケーションコンポーネントを作成でき、アダプターの変更によって、依存コンポーネントを交換することが可能です。これにより、アプリケーションコンポーネントと外部の入出力要素が、コンテキストの認識なしに、やり取りを行えるようになります。また、どのレベルでもコンポーネントを交換可能なため、自動テストが容易になります。インフラストラクチャコードに依存することなく、コンポーネントを個別にテストでき、テスト実行のために環境全体をプロビジョニングする必要はありません。アプリケーションロジックに外部要因との依存関係がないため、テストの簡素化や、依存関係の模倣も簡単に行えるようになります。

例えば、疎結合アーキテクチャのアプリケーションコンポーネントでは、データストアの詳細を認識せずに、読み取りと書き込みを行えなければなりません。インターフェイス (ポート) にデータを提供することが、アプリケーションコンポーネントの役割だからです。アダプターでは、データストアへの書き込みロジックを定義し、データストアには、アプリケーションのニーズに応じて、データベース、ファイルシステム、または Amazon S3 などのオブジェクトストレージシステムを使用できます。

高レベルのアーキテクチャ

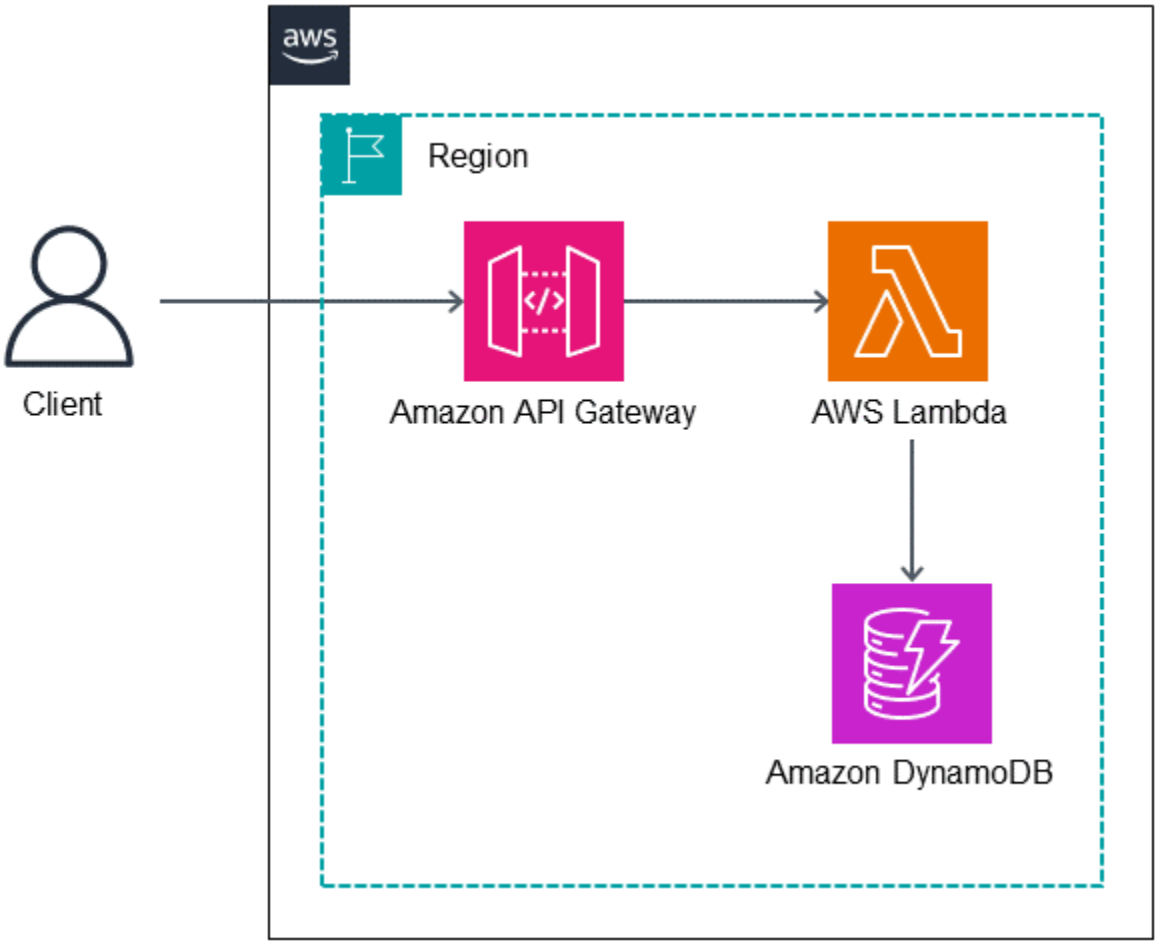
アプリケーションまたはアプリケーションコンポーネントは、コアビジネスロジックを備えており、ポートからコマンドまたはクエリを受信し、ポートを介して外部アクターにリクエストを送信します。外部アクターはアダプターを介して実装します。これを次の図に示します。



を使用した実装 AWS のサービス

AWS Lambda 関数には、多くの場合、ビジネスロジックとデータベース統合コードの両方が含まれており、目的を達成するために緊密に結合されています。六角形アーキテクチャパターンを使用すると、ビジネスロジックをインフラストラクチャコードから分離できます。こうした分離により、データベースコードへの依存なくビジネスロジックのユニットテストが可能になり、開発プロセスの俊敏性が向上します。

次のアーキテクチャでは、Lambda 関数に、六角形アーキテクチャパターンを実装しており、Lambda 関数は、Amazon API Gateway REST API によって開始します。この関数にビジネスロジックを実装することで、DynamoDB テーブルにデータを書き込みます。



「サンプルコード」

このセクションのサンプルコードでは、Lambda を使用してドメインモデルを実装し、それをインフラストラクチャコード (DynamoDB にアクセスするためのコードなど) から分離した後に、関数のユニットテストを実装しています。

ドメインモデル

ドメインモデルクラスは、外部コンポーネントや依存関係に関する情報は持たず、ビジネスロジックのみを実装します。次の例の `Recipient` クラスは、ドメインモデルクラスであり、これによって予約日の重複をチェックします。

```
class Recipient:
    def __init__(self, recipient_id:str, email:str, first_name:str, last_name:str,
 age:int):
        self.__recipient_id = recipient_id
        self.__email = email
```

```
        self.__first_name = first_name
        self.__last_name = last_name
        self.__age = age
        self.__slots = []

    @property
    def recipient_id(self):
        return self.__recipient_id
    #.....

    def are_slots_same_date(self, slot:Slot) -> bool:
        for selfslot in self.__slots:
            if selfslot.reservation_date == slot.reservation_date:
                return True
        return False

    def is_slot_counts_equal_or_over_two(self) -> bool:
    #.....
```

入力ポート

RecipientInputPort クラスは recipient クラスに接続し、ドメインロジックを実行します。

```
class RecipientInputPort(IRecipientInputPort):
    def __init__(self, recipient_output_port: IRecipientOutputPort, slot_output_port:
 ISlotOutputPort):
        self.__recipient_output_port = recipient_output_port
        self.__slot_output_port = slot_output_port

    '''
    make reservation: adapting domain model business logic
    '''
    def make_reservation(self, recipient_id:str, slot_id:str) -> Status:
        status = None

        # -------------------------------------------------
        # get an instance from output port
        # -------------------------------------------------
        recipient = self.__recipient_output_port.get_recipient_by_id(recipient_id)
        slot = self.__slot_output_port.get_slot_by_id(slot_id)

        if recipient == None or slot == None:
            return Status(400, "Request instance is not found. Something wrong!")
```

```
        print(f"recipient: {recipient.first_name}, slot date: {slot.reservation_date}")

        # ------------------------------------------------
        # execute domain logic
        # ------------------------------------------------
        ret = recipient.add_reserve_slot(slot)

        # ------------------------------------------------
        # persistent an instance throgh output port
        # ------------------------------------------------
        if ret == True:
            ret = self.__recipient_output_port.add_reservation(recipient)

        if ret == True:
            status = Status(200, "The recipient's reservation is added.")
        else:
            status = Status(200, "The recipient's reservation is NOT added!")
        return status
```

DynamoDB アダプタークラス

DDBRecipientAdapter クラスは、DynamoDB テーブルへのアクセスを実装します。

```
class DDBRecipientAdapter(IRecipientAdapter):
    def __init__(self):
        ddb = boto3.resource('dynamodb')
        self.__table = ddb.Table(table_name)

    def load(self, recipient_id:str) -> Recipient:
        try:
            response = self.__table.get_item(
                Key={'pk': pk_prefix + recipient_id})
        ...

    def save(self, recipient:Recipient) -> bool:
        try:
            item = {
                "pk": pk_prefix + recipient.recipient_id,
                "email": recipient.email,
                "first_name": recipient.first_name,
                "last_name": recipient.last_name,
                "age": recipient.age,
```

```
                "slots": []
            }
          # ...
```

Lambda 関数 get_recipient_input_port は、RecipientInputPort クラスインスタンスのファクトリであり、関連するアダプターインスタンスを使用して、出力ポートクラスのインスタンスを構築します。

```
def get_recipient_input_port():
    return RecipientInputPort(
        RecipientOutputPort(DDBRecipientAdapter()),
        SlotOutputPort(DDBSlotAdapter()))

def lambda_handler(event, context):

    body = json.loads(event['body'])
    recipient_id = body['recipient_id']
    slot_id = body['slot_id']

    # get an input port instance
    recipient_input_port = get_recipient_input_port()
    status = recipient_input_port.make_reservation(recipient_id, slot_id)

    return {
        "statusCode": status.status_code,
        "body": json.dumps({
            "message": status.message
        }),
    }
```

ユニットテスト

ドメインモデルクラスのビジネスロジックをテストするには、モッククラスを挿入します。次の例は、ドメインモデル Recipent クラスのユニットテストを示しています。

```
def test_add_slot_one(fixture_recipient, fixture_slot):
    slot = fixture_slot
    target = fixture_recipient
    target.add_reserve_slot(slot)
    assert slot != None
    assert target != None
    assert 1 == len(target.slots)
```

```
    assert slot.slot_id == target.slots[0].slot_id
    assert slot.reservation_date == target.slots[0].reservation_date
    assert slot.location == target.slots[0].location
    assert False == target.slots[0].is_vacant

def test_add_slot_two(fixture_recipient, fixture_slot, fixture_slot_2):
    #.....

def test_cannot_append_slot_more_than_two(fixture_recipient, fixture_slot,
 fixture_slot_2, fixture_slot_3):
    #.....

def test_cannot_append_same_date_slot(fixture_recipient, fixture_slot):
    #.....
```

GitHub リポジトリ

このパターンのサンプルアーキテクチャの完全な実装については、https://github.com/aws-samples/aws-lambda-domain-model-sample の GitHub リポジトリを参照してください。

関連情報

- Hexagonal architecture (Alistair Cockburn 氏による記事)
- を使用した進化アーキテクチャの開発 AWS Lambda (日本語AWS ブログ記事)

動画

次の動画 (日本語) では、Lambda 関数を使用したドメインモデル実装における六角形アーキテクチャの使用について解説しています。

パブリッシュ – サブスクライブパターン

Intent

パブリッシュ – サブスクライブパターンは、メッセージ送信者 (パブリッシャー) を対象となる受信者 (サブスクライバー) から切り離すメッセージングパターンのことで、pub-sub パターンとも呼ばれています。このパターンは、メッセージブローカーまたはルーター (メッセージインフラストラクチャ) と呼ばれる中間層を介してメッセージまたはイベントを発行することにより、非同期通信を実装します。パブリッシュ – サブスクライブパターンでは、メッセージ配信の責任をメッセージインフラストラクチャに移管することで、送信者は中核的なメッセージ処理に集中できるため、送信者のスケーラビリティと応答性が向上します。

導入する理由

分散アーキテクチャでは、システム内でイベントが発生した際、システムコンポーネントから他のコンポーネントに情報を提供する必要性が生じることがよくあります。パブリッシュ – サブスクライブパターンでは、考慮すべき事項を分離できるため、アプリケーションは中核的な機能に集中できます。メッセージルーティングや信頼性の高い配信などの通信は、メッセージインフラストラクチャにより処理されます。パブリッシュ – サブスクライブパターンは、非同期メッセージングでパブリッシャーとサブスクライバーを切り離すことを可能にします。またパブリッシャーは、サブスクライバーに関する情報を持たなくても、メッセージを送信することができます。

適用対象

パブリッシュ – サブスクライブパターンは次の場合に使用します。

- 1 つのメッセージのワークフローが異なるので、並列処理が必要なとき。
- 複数のサブスクライバーにメッセージをブロードキャストしており、受信者からのリアルタイムの応答が必要ないとき。
- 最終的にデータや状態の一貫性が保たれるのであれば、システムやアプリケーションはそれを許容できるとき。
- 異なる言語、プロトコル、またはプラットフォームを使用する可能性のある他のアプリケーションまたはサービスと、アプリケーションまたはコンポーネントが通信を行う必要があるとき。

問題点と考慮事項

- サブスクライバーとの接続性: パブリッシャーは、サブスクライバーがリスナーとして機能しているかどうかを把握しません。つまり、リスナーは機能していない可能性もあります。発行されたメッセージは一時的なものであり、サブスクライバーからの応答がない場合には、ドロップされることがあります。
- メッセージの配信保証: 通常、Amazon Simple Notification Service (Amazon SNS) などの特定のサービスで、あるサブスクライバーサブセットに対し配信できるのは 1 回限りですが、パブリッシュ – サブスクライブパターンでは、すべてのサブスクライバータイプへのメッセージの配信が保証される訳ではありません。
- 有効期限 (TTL): メッセージには寿命があり、その期間内に処理されないと有効期限切れになります。TTL 期間を超えてもメッセージが確実に処理されるようにするには、発行されたメッセージをキューに追加して永続化することを検討してください。
- メッセージの関連性: プロデューサーは、メッセージデータの中に関連性についての時間スパンを設定でき、メッセージがこの日付を過ぎた場合はそれを破棄できます。メッセージの処理方法を決定する場合には、設計段階でコンシューマーがこの情報を検証するようにすることを検討してください。
- 最終的な一貫性: メッセージが発行されてからサブスクライバーがそのメッセージを消費するまでには遅延が存在します。これにより、強い一貫性が求められる場合でも、サブスクライバーデータストアの一貫性が実現されるのが、最終段階になる可能性があります。プロデューサーとコンシューマーがほぼリアルタイムのやり取りを必要とする場合も、最終的な一貫性の実現が問題になることがあります。
- 単方向の通信: パブリッシュ – サブスクライブパターンの通信は単方向と見なされます。返信サブスクリプションチャネルによる双方向メッセージングを必要とするアプリケーションで、応答の同期が必要な場合は、要求 – 応答パターンの使用を検討する必要があります。
- メッセージの順序: メッセージの順序は保証されません。コンシューマーが順序付けられたメッセージを必要とする場合は、Amazon SNS FIFO トピックを使用して、順序を保証することをお勧めします。
- メッセージの重複: メッセージ伝送インフラストラクチャによっては、重複したメッセージがコンシューマーに配信される場合があります。コンシューマーは、重複したメッセージの処理に関し冪等性を持つように設計されている必要があります。または、Amazon SNS FIFO トピックを使用すれば、配信を確実に1 回のみ行うようにもできます。
- メッセージのフィルタリング: コンシューマーが関心を寄せるのは、プロデューサーが発行したメッセージのサブセットのみである、ということが良くあります。トピックやコンテンツフィル

ターを提供することで、サブスクライバーが受信したメッセージにフィルターをかけたり、絞り込んだりできるメカニズムを実現できます。

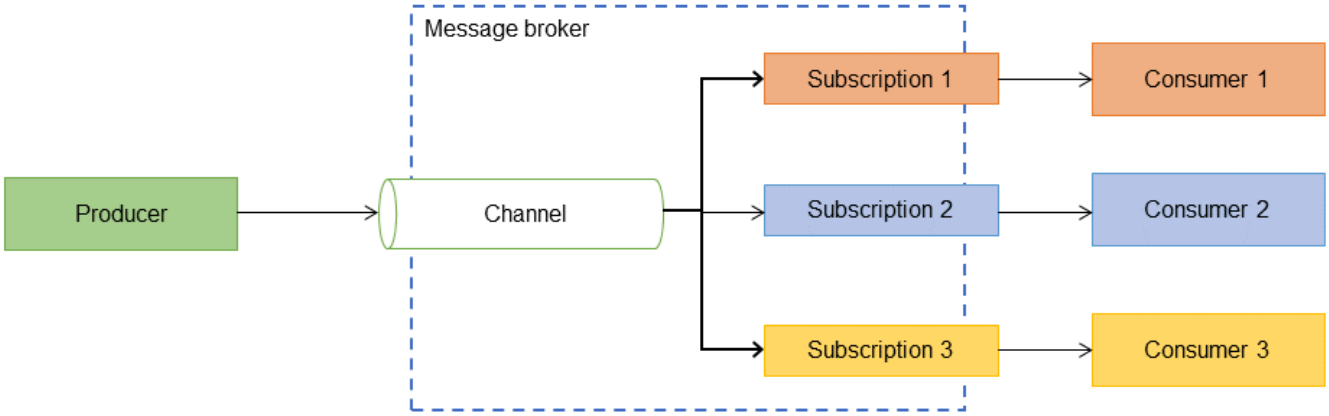
- メッセージ再生: メッセージの再生機能は、メッセージングインフラストラクチャにより異なってきます。また、ユースケースに応じ、カスタム実装を提供することもできます。
- デッドレターキュー: 郵便システムでは、配達不能な郵便物を処理するための施設である、デッドレターオフィスがあります。pub/sub のメッセージングにおいては、サブスクライブ済みのエンドポイントに配信できないメッセージのためのキューとして、デッドレターキューが存在します。

実装

高レベルのアーキテクチャ

パブリッシュ – サブスクライブパターンでは、メッセージブローカーまたはルーターと呼ばれる非同期メッセージングサブシステムが、サブスクリプションを追跡します。プロデューサーがイベントを発行すると、メッセージングインフラストラクチャは各コンシューマーにメッセージを送信します。サブスクライバーに送信されたメッセージは、メッセージインフラストラクチャから削除されるので再生はできなくなり、対象のイベントは新しいサブスクライバーには表示されません。メッセージブローカーまたはルーターは、以下の方法でイベントプロデューサーをメッセージコンシューマーから切り離します。

- メッセージにパッケージ化されたイベントを発行するための入力チャネルを、定義されたメッセージ形式を使用してプロデューサーに提供する。
- サブスクリプションごとに個別の出力チャンネルを作成する。サブスクリプションとは、特定の入力チャンネルに関連するイベントメッセージを受信するための、コンシューマーの接続です。
- イベントが発行されたときに、すべてのコンシューマーに対し、メッセージを入力チャンネルから出力チャンネルにコピーします。



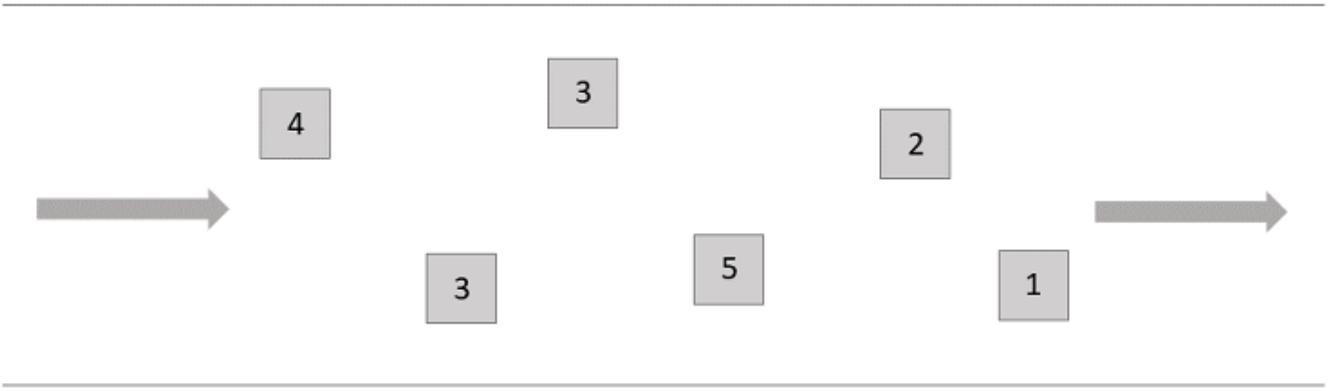
AWS のサービスを使用した実装

Amazon SNS

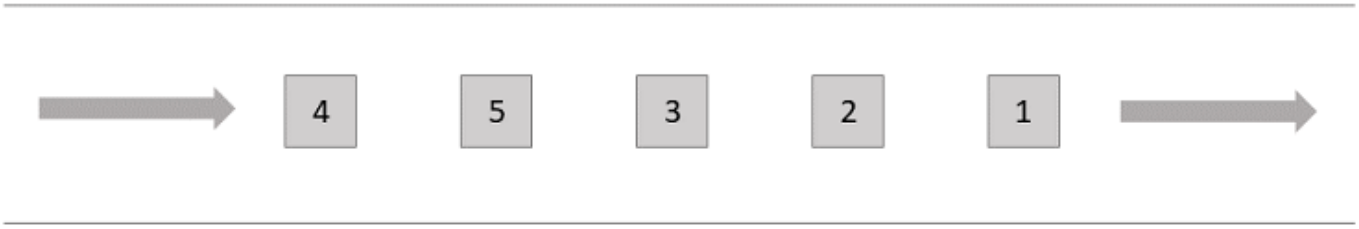
Amazon SNS は、分散したアプリケーションを分離するために、アプリケーションからアプリケーションへ (A2A) のメッセージ伝送を提供する、完全マネージド型のパブリッシャー – サブスクライバーサービスです。またこのサービスでは、SMS、E メール、その他のプッシュ通知を送信するために、アプリケーションから個人へ (A2P) のメッセージ伝送も行えます。

Amazon SNS では、標準と、先入れ先出し (FIFO) の、2 種類のトピックを提供しています。

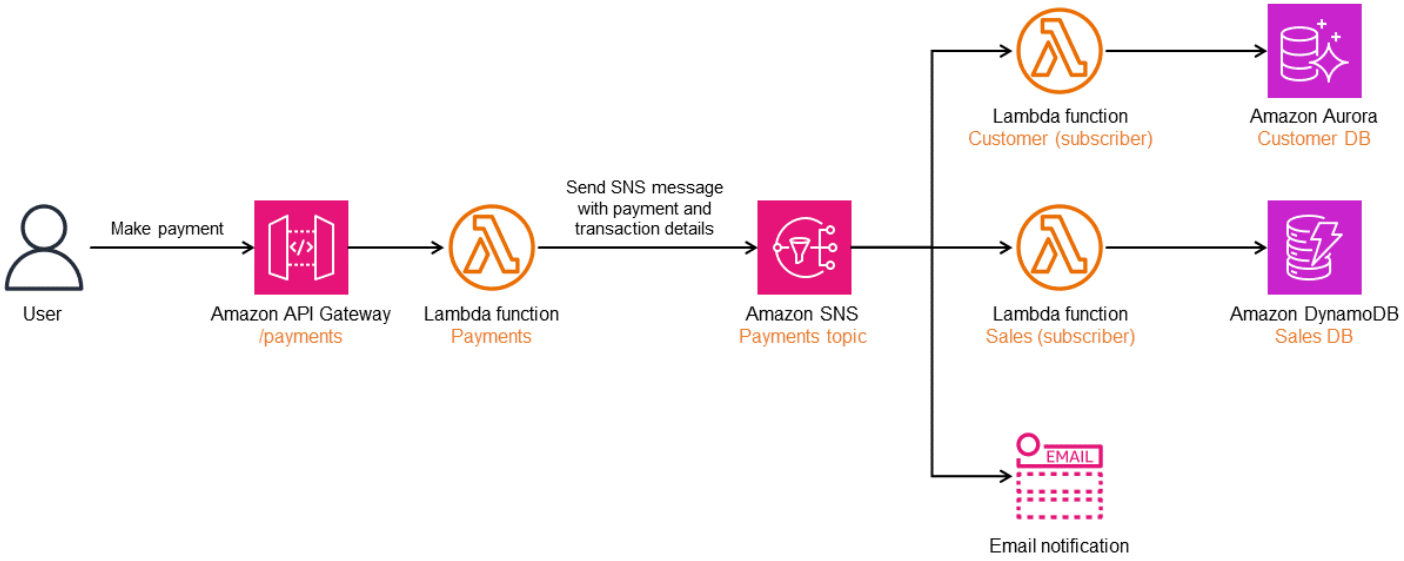
- 標準トピックでは、1 秒あたりのメッセージ数を上限なくサポートし、また順序付けと重複排除はベストエフォートになります。



- FIFO トピックでは、厳密な順序付けと重複排除が行われます。また、FIFO トピックごとに 1 秒あたり 300 件のメッセージ、または1 秒あたり 10 MB (のどちらか早く発生した方) がサポートされます。



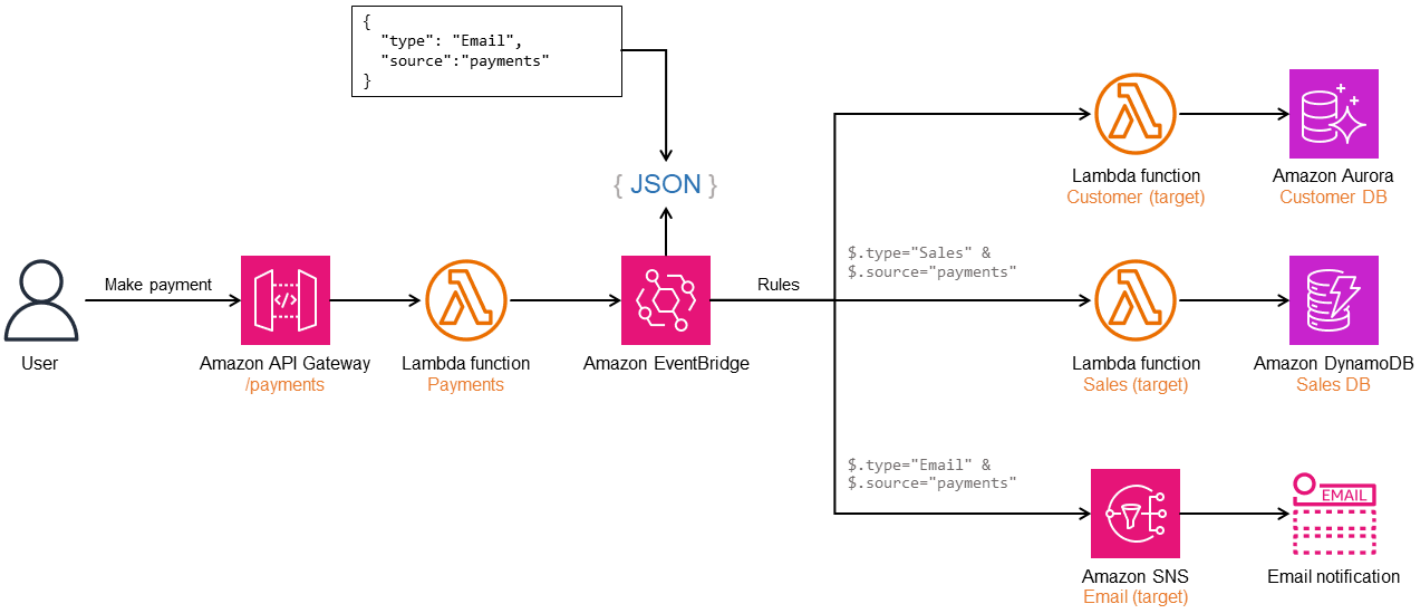
次の図は、Amazon SNS を使用したパブリッシュ – サブスクライブパターンの実装方法を示しています。ユーザーが支払いを行うと、SNS メッセージが Payments Lambda 関数によりPayments SNS トピックに送信されます。この SNS トピックには 3 人のサブスクライバーがいます。各サブスクライバーがメッセージのコピーを受信して、その処理を行います。



Amazon EventBridge ()

Amazon EventBridge は、異なるプロトコルにおよぶ複数のプロデューサーからのメッセージを、サブスクライブしているコンシューマー、または直接的およびファンアウトのサブスクリプションに対して送る、より複雑なルーティングが必要な場合に使用します。また EventBridge では、コンテンツに基づいたルーティング、フィルタリング、シーケンシング、および分割や集約もサポートされています。以下の図では、パブリッシュ – サブスクライブパターンの (イベントルールを使用してサブスクライバーが定義される) バージョンを、EventBridge を使用して構築しています。ユーザーが支払いを行うと、Payments Lambda 関数は、デフォルトの (異なるターゲットを指す 3 つのルールを含

むカスタムスキーマに基づく) イベントバスを使用して、EventBridge にメッセージを送信します。各マイクロサービスはメッセージを処理し、必要なアクションを実行します。



ワークショップ

- AWS でのイベント駆動型アーキテクチャの構築
- Amazon Simple Queue Service (Amazon SQS) と Amazon Simple Notification Service (Amazon SNS) を使用したファンアウトイベントの送信

ブログの参考情報

- サーバーレスアプリケーション向けメッセージングサービスの選択
- Amazon SNS、Amazon SQS、AWS Lambda 向けの耐久性の高いサーバーレスアプリケーションの DLQ を使用した設計
- Amazon SNS のメッセージフィルタリングによる pub/sub メッセージングの簡素化

関連情報

- pub/sub メッセージングの特徴

バックオフパターンで再試行

Intent

バックオフによる再試行のパターンでは、一時的なエラーによって失敗したオペレーションを透過的に再試行して、アプリケーションの安定性を向上させます。

導入する理由

分散アーキテクチャでは、サービススロットリング、ネットワーク接続の一時的な喪失、一時的なサービス利用不可などによって、一時的なエラーが発生する可能性があります。こうした一時的なエラーによって失敗したオペレーションを自動的に再試行すると、ユーザーエクスペリエンスやアプリケーションのレジリエンスが向上します。しかし、頻繁な再試行は、ネットワーク帯域幅への過負荷や、競合を招きかねません。エクスポネンシャルバックオフという手法を使用すると、指定した回数だけ再試行の待機時間を延長し、オペレーションを再試行できます。

適用対象

バックオフによる再試行のパターンは次の場合に使用します。

- サービス側で頻繁にリクエストをスロットリングしてオーバーロードを防いでいるため、呼び出しプロセスで「429 Too many requests」の例外が発生している。
- 分散アーキテクチャでは、ネットワークの動作が視野から外れやすいため、一時的なネットワークの問題によって障害が発生している。
- 呼び出しているサービスが一時的に使用できず、障害が発生している。このパターンを使用してバックオフタイムアウトを導入しない限り、頻繁な再試行がサービス低下を招く可能性がある。

問題点と考慮事項

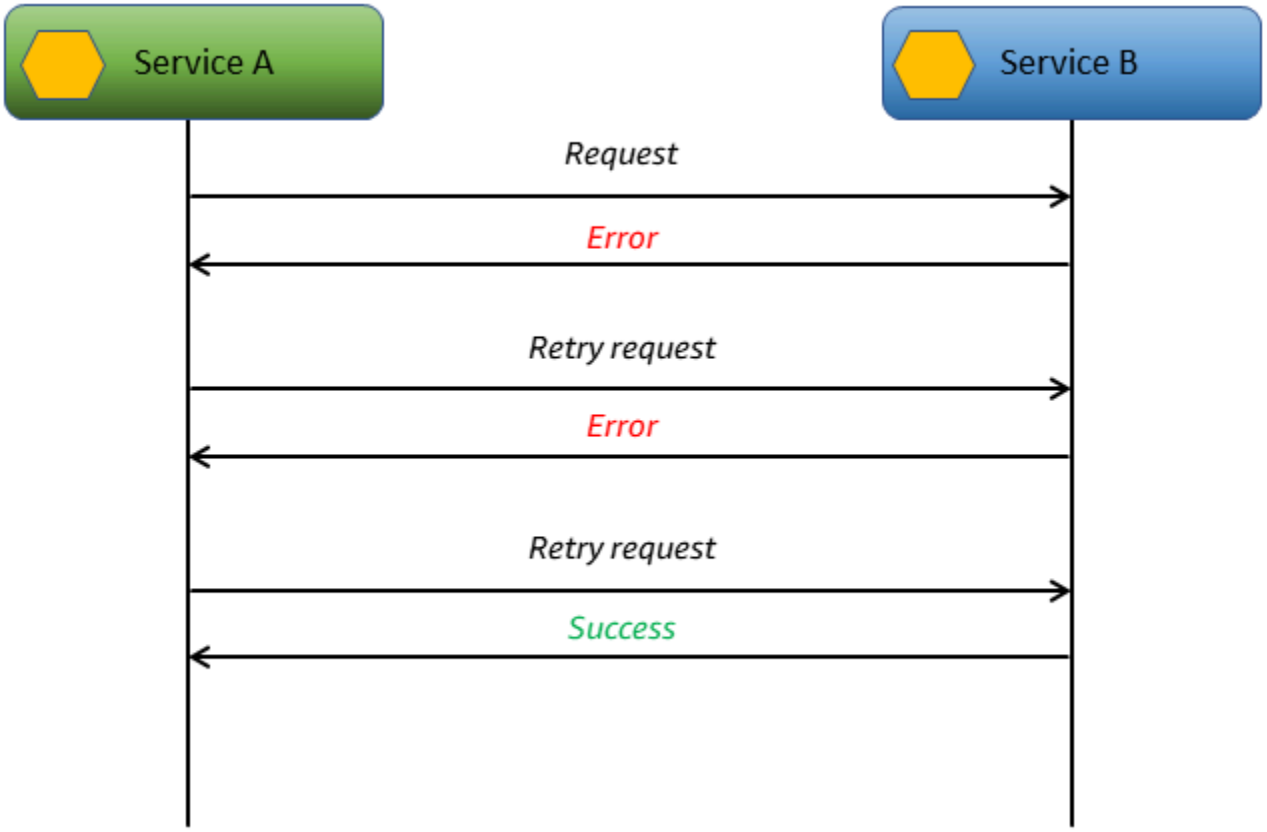
- べき等性: あるメソッドを複数回呼び出したときの効果と、あるシステム状態を 1 回呼び出したときの効果が同じ場合、そのオペレーションにはべき等性があると見なされます。バックオフによる再試行のパターンを使用する場合、そのオペレーションは、べき等でなければなりません。そうでない場合は、部分的な更新によってシステム状態が損なわれる可能性があります。
- ネットワーク帯域幅: 再試行回数が多すぎ、ネットワーク帯域幅が占有されると、応答が遅延するようになり、サービス品質が低下する可能性があります。

- フェイルファストのシナリオ: 一時的でないエラーの原因を特定できる場合は、サーキットブレーカーパターンを使用してフェイルファストを実践した方が効率的です。
- バックオフレート: エクスポネンシャルバックオフを導入すると、サービスのタイムアウトに影響が及ぶため、エンドユーザーの待ち時間が長くなる可能性があります。

実装

高レベルのアーキテクチャ

次の図は、サービス A が、正常なレスポンスが返るまでサービス B への呼び出しを再試行する方法を示しています。数回試行しても、サービス B から成功のレスポンスが返らない場合、サービス A は再試行を停止し、呼び出し元に失敗のステータスを返すことができます。



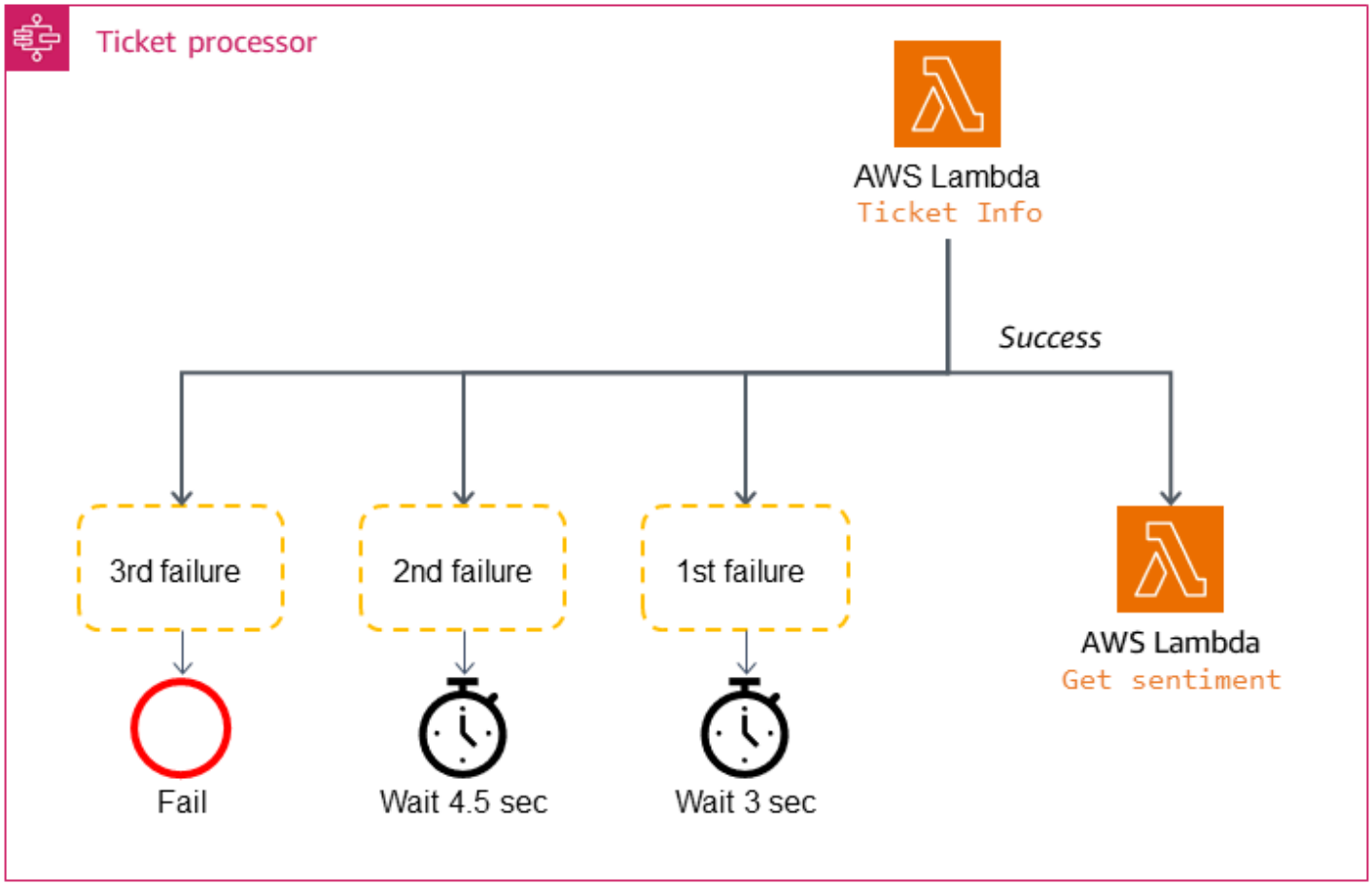
AWS のサービスを使用した実装

次の図は、カスタマーサポートプラットフォームでのチケット処理ワークフローを示しています。不満を持つ顧客からのチケットは、チケットの優先度を自動的に上げ、その対応を迅速化します。`Ticket info` Lambda 関数はチケットの詳細を抽出し、`Get sentiment` Lambda 関数を呼び

出します。Get sentiment Lambda 関数は、Amazon Comprehend (図示なし) にその説明を渡して顧客の感情をチェックします。

Get sentiment Lambda 関数の呼び出しが失敗した場合、オペレーションが 3 回再試行されます。エクスポネンシャルバックオフバックオフを有効にするには、AWS Step Functions でバックオフ値を設定します。

この例では、最大 3 回の再試行を 1.5 秒の増加乗数で設定しています。最初の再試行を 3 秒後に行う場合、2 回目の再試行は 3 x 1.5 秒 = 4.5 秒後に、3 回目の再試行は 4.5 x 1.5 秒 = 6.75 秒後に行われます。3 回目の再試行が失敗すると、ワークフローは失敗します。バックオフロジックにはカスタムコードは不要で、AWS Step Functions の設定項目を利用します。



サンプルコード

次のコードは、バックオフによる再試行のパターンの実装を示しています。

```
public async Task DoRetriesWithBackOff()
  {
```

```
    int retries = 0;
    bool retry;
    do
    {
      //Sample object for sending parameters
      var parameterObj = new InputParameter { SimulateTimeout = "false" };
      var content = new StringContent(JsonConvert.SerializeObject(parameterObj),
                              System.Text.Encoding.UTF8, "application/json");
      var waitInMilliseconds = Convert.ToInt32((Math.Pow(2, retries) - 1) * 100);
      System.Threading.Thread.Sleep(waitInMilliseconds);
      var response =  await _client.PostAsync(_baseURL, content);
      switch (response.StatusCode)
      {
        //Success
        case HttpStatusCode.OK:
          retry = false;
          Console.WriteLine(response.Content.ReadAsStringAsync().Result);
          break;
        //Throttling, timeouts
        case HttpStatusCode.TooManyRequests:
        case HttpStatusCode.GatewayTimeout:
          retry = true;
          break;
        //Some other error occured, so stop calling the API
        default:
          retry = false;
          break;
      }
      retries++;
    } while (retry && retries < MAX_RETRIES);
  }
```

GitHub リポジトリ

このパターンのサンプルアーキテクチャの完全な実装については、https://github.com/aws-samples/retry-with-backoff にある GitHub リポジトリを参照してください。

関連情報

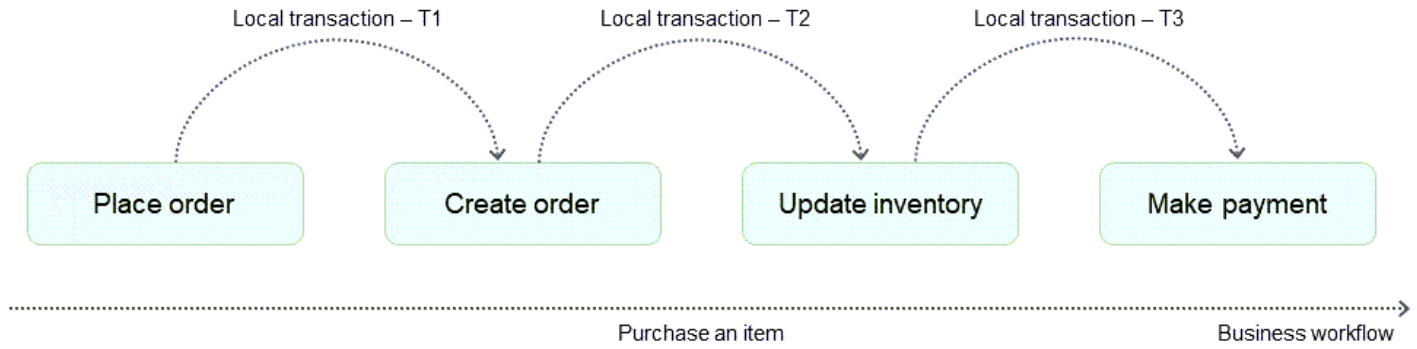
- ジッターを伴うタイムアウト、再試行、およびバックオフ (Amazon Builders' Library)

Saga パターン

Saga は一連のローカルトランザクションで構成されます。Saga 内の各ローカルトランザクションはデータベースを更新し、次のローカルトランザクションをトリガーします。トランザクションが失敗した場合、Saga は補償トランザクションを実行して、以前のトランザクションによって行われたデータベースの変更を元に戻します。

この一連のローカルトランザクションは、継続と補償の原則を採用することでビジネスワークフローを実現するのに役立ちます。ワークフローのフォワードリカバリは継続原則によって決定され、バックワードリカバリは補償原則によって決定されます。トランザクションのどの段階でも更新が失敗した場合、Saga は継続 (トランザクションを再試行する) か補償 (前のデータ状態に戻す) かのいずれかのイベントを発行します。これにより、データの完全性が維持され、データストア全体の整合性が保たれます。

例えば、ユーザーがオンライン小売業者から本を購入する場合、そのプロセスは、ビジネスワークフローを反映して、注文の作成、在庫の更新、支払い、出荷などの一連のトランザクションで構成されます。このワークフローを完了するために、この分散型アーキテクチャでは一連のローカルトランザクションを発行して、注文データベースでの注文の作成、在庫データベースの更新、支払いデータベースの更新を行います。処理が成功する場合は、次の図に示すように、これらのトランザクションが順番に呼び出され、ビジネスワークフローが完了します。一方、これらのローカルトランザクションのいずれかが失敗した場合、システムは次の適切なステップ、つまりフォワードリカバリとバックワードリカバリのどちらかを決定できるようになっています。



次の 2 つのシナリオは、次のステップがフォワードリカバリかバックワードリカバリかを判断するのに役立ちます。

- プラットフォームレベルの障害 (基盤となるインフラストラクチャに何らかの問題が発生し、トランザクションが失敗する場合)。この場合、saga パターンはローカルトランザクションを再試行してビジネスプロセスを続行することでフォワードリカバリを実行できます。

- アプリケーションレベルの障害 (無効な支払いが原因で支払いサービスが停止する場合)。この場合、saga パターンは、補償トランザクションを発行して在庫データベースと注文データベースを更新し、以前の状態に戻すことでバックワードリカバリを実行できます。

saga パターンは、ビジネスワークフローを処理し、フォワードリカバリによって望ましい最終状態に到達するようにします。障害が発生した場合は、データ整合性の問題が発生しないように、バックワードリカバリを使用してローカルトランザクションを元に戻します。

saga パターンには、コレオグラフィとオーケストレーションという 2 つのバリエーションがあります。

Saga コレオグラフィ

Saga コレオグラフィパターンは、マイクロサービスが発行するイベントによって異なります。Saga を構成するサービス (マイクロサービス) はイベントをサブスクライブし、イベントトリガーに基づくアクションを実行します。例えば、次の図の注文サービスは `OrderPlaced` イベントを発生させます。在庫サービスはそのイベントをサブスクライブし、`OrderPlaced` イベントが発生すると在庫を更新します。同様に、構成しているサービスは発生したイベントのコンテキストに基づいて動作します。

Saga コレオグラフィパターンは、Saga を構成しているサービスが少なく、単一障害点のないシンプルな実装が必要な場合に適しています。構成しているサービスが増えると、このパターンを使用して構成しているサービス間の依存関係を追跡するのが難しくなります。

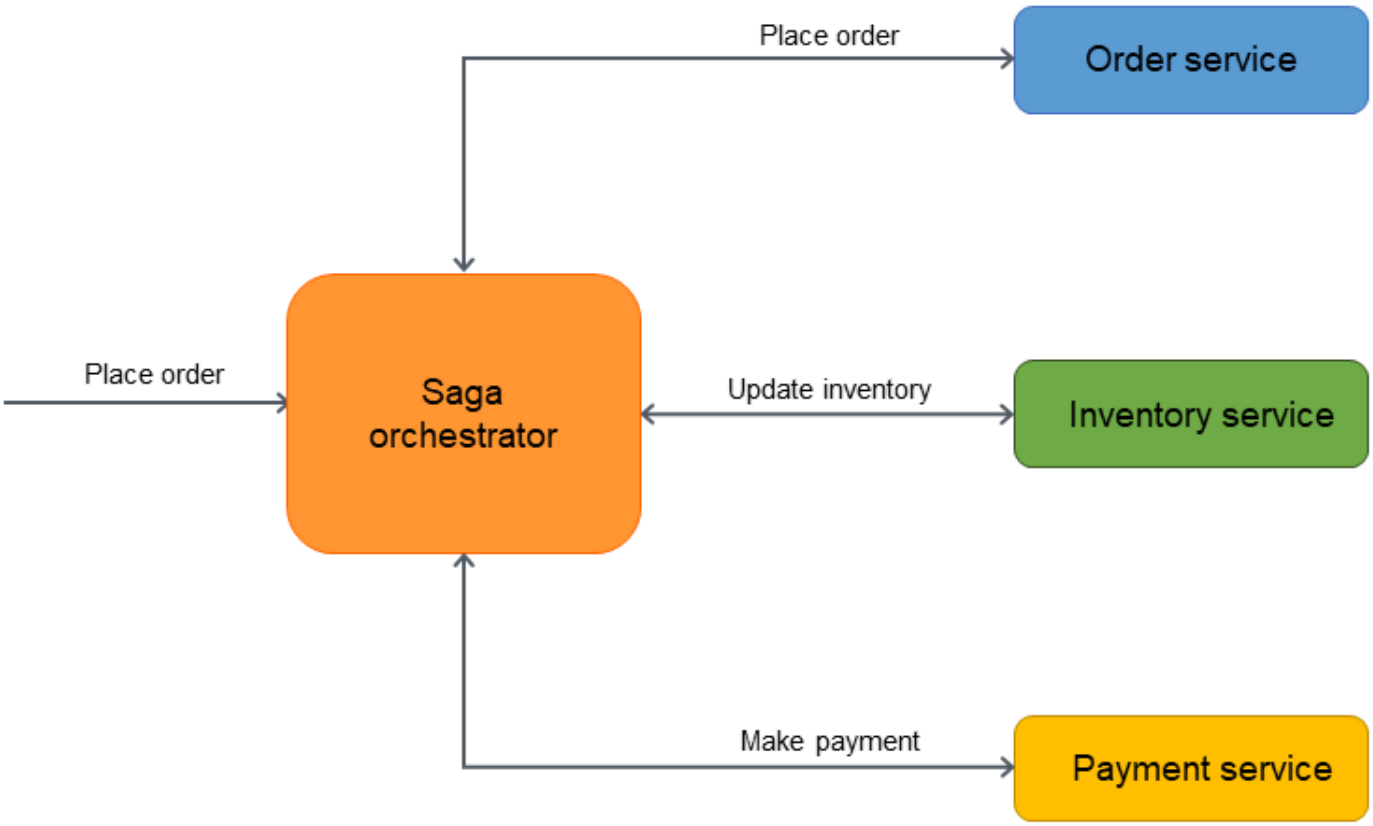
詳細なレビューについては、このガイドの「Saga コレオグラフィ」セクションを参照してください。

Saga オーケストレーション

Saga オーケストレーションパターンには、オーケストレーターと呼ばれるセントラルコーディネーターが存在します。Saga オーケストレーターは、トランザクションのライフサイクル全体を管理し、調整します。トランザクションを完了するために実行すべき一連のステップを認識しています。ステップを実行するには、構成しているマイクロサービスにメッセージを送信してオペレーションを実行します。構成しているマイクロサービスはオペレーションを完了し、オーケストレーターにメッ

セージを返信します。受信したメッセージに基づいて、オーケストレーターはトランザクションで次に実行するマイクロサービスを決定します。

Saga オーケストレーションパターンは、構成しているサービスが多く、Saga を構成するサービス間で疎結合が必要な場合に適しています。オーケストレーターは、ロジックの複雑さをカプセル化するために、構成しているサービスを疎結合にしています。ただし、オーケストレーターがワークフロー全体を制御するため、単一障害点になる可能性があります。



詳細なレビューについては、このガイドの「Saga オーケストレーション」セクションを参照してください。

Saga コレオグラフィパターン

Intent

Saga コレオグラフィパターンは、イベントサブスクリプションを使用することで、複数のサービスにまたがる分散トランザクションのデータ完全性を維持するのに役立ちます。分散トランザクションでは、トランザクションが完了する前に複数のサービスを呼び出すことができます。異なるデータス

トアにサービスがデータを保存する場合、これらのデータストア間でデータ整合性を維持するのが難しい場合があります。

導入する理由

トランザクションとは、複数のステップを含む可能性のある単一の作業単位です。すべてのステップが完全に実行されるか、まったく実行されないため、データストアは整合した状態を保持します。アトミック性、整合性、分離性、耐久性 (ACID) という用語によって、トランザクションの特性が定義されます。リレーショナルデータベースは ACID トランザクションを提供してデータ整合性を維持します。

トランザクションの整合性を維持するため、リレーショナルデータベースでは 2 フェーズコミット (2PC) 方式を使用します。これは準備フェーズとコミットフェーズで構成されます。

- 準備フェーズでは、調整プロセスはトランザクションを構成するプロセス (構成プロセス) に、トランザクションをコミットするかロールバックするかを約束するように要求します。
- コミットフェーズでは、調整プロセスが構成プロセスにトランザクションのコミットを要求します。準備フェーズで構成プロセスがコミットに同意できない場合、トランザクションはロールバックされます。

Database-per-service 設計パターンに従う分散システムでは、2 フェーズコミットは選択肢になりません。これは、各トランザクションがさまざまなデータベースに分散され、リレーショナルデータストアの 2 フェーズコミットと同様にプロセスを調整できる単一のコントローラーが存在しないためです。この場合の解決策の 1 つは、Saga コレオグラフィパターンを使用することです。

適用対象

Saga コレオグラフィパターンは次のような場合に使用します。

- 複数のデータストアにまたがる分散トランザクションのデータ完全性と整合性がシステムに求められる場合。
- データストア (NoSQL データベースなど) が 2PC を用意して ACID トランザクションに対応していない場合で、1 つのトランザクション内で複数のテーブルを更新する必要があり、アプリケーションの境界内で 2PC を実装する作業が複雑な場合。
- 構成プロセスのトランザクションを管理するセントラル制御プロセスが、単一障害点になる可能性がある場合。
- Saga の構成プロセスが独立したサービスであり、疎結合にする必要がある場合。

- ビジネスドメインのコンテキスト境界にまたがったコミュニケーションがある場合。

問題点と考慮事項

- 複雑さ: マイクロサービスの数が増えるにつれて、マイクロサービス間のやり取りの数が増えるため、Saga コレオグラフィを管理するのが難しくなる可能性があります。さらに、補償トランザクションや再試行によってアプリケーションコードが複雑になり、メンテナンスのオーバーヘッドが発生する可能性があります。コレオグラフィは、Saga を構成しているサービスが少なく、単一障害点のないシンプルな実装が必要な場合に適しています。構成しているサービスが増えると、このパターンを使用して構成しているサービス間の依存関係を追跡するのが難しくなります。
- 回復力のある実装: Saga コレオグラフィでは、Saga オーケストレーションと比べて、タイムアウト、再試行、その他の回復パターンをグローバルに実装するのが難しくなります。コレオグラフィはオーケストレーターレベルではなく、個々のコンポーネントに実装する必要があります。
- 循環的な依存関係: 構成しているサービスはお互いに発行したメッセージを使用します。その結果、依存関係が循環的に発生して、コードが複雑になり、メンテナンスのオーバーヘッドが発生し、デッドロックの可能性も発生します。
- 二重書き込みの問題: マイクロサービスはアトミックにデータベースを更新し、イベントを発行する必要があります。いずれかのオペレーションが失敗すると、不整合な状態になる可能性があります。これを解決する 1 つの方法は、トランザクションアウトボックスパターンを使用することです。
- イベントの保存: Saga を構成するサービスは、発行されたイベントに基づいてアクションを実行します。監査、デバッグ、再生のために、イベントを発生順に保存することが重要です。データ整合性を回復するためにシステム状態を再生する必要がある場合に備えて、イベントソーシングパターンを使用してイベントをイベントストアに保存できます。イベントストアにはシステム内のすべての変更が反映されるため、監査やトラブルシューティングにも使用できます。
- 結果整合性: ローカルトランザクションを順次処理することで結果整合性が保たれます。これは、強い整合性を必要とするシステムでは課題となることがあります。この問題を解決するには、整合性モデルに対するビジネスチームの期待を設定するか、ユースケースを再評価して、強固な整合性を備えたデータベースに切り替えます。
- 冪等性: Saga を構成するサービスには、予期しないクラッシュやオーケストレーターの障害によって一時的な障害が発生した場合に繰り返し実行できるように冪等性が必要です。
- トランザクションの分離: saga パターンには、ACID トランザクションの 4 つの特性の 1 つであるトランザクション分離機能がありません。トランザクションの分離度によって、トランザクションが処理するデータに他の同時トランザクションがどの程度影響する可能性があるかが決まります。トランザクションの同時オーケストレーションによって、最新でないデータが発生する可能性があ

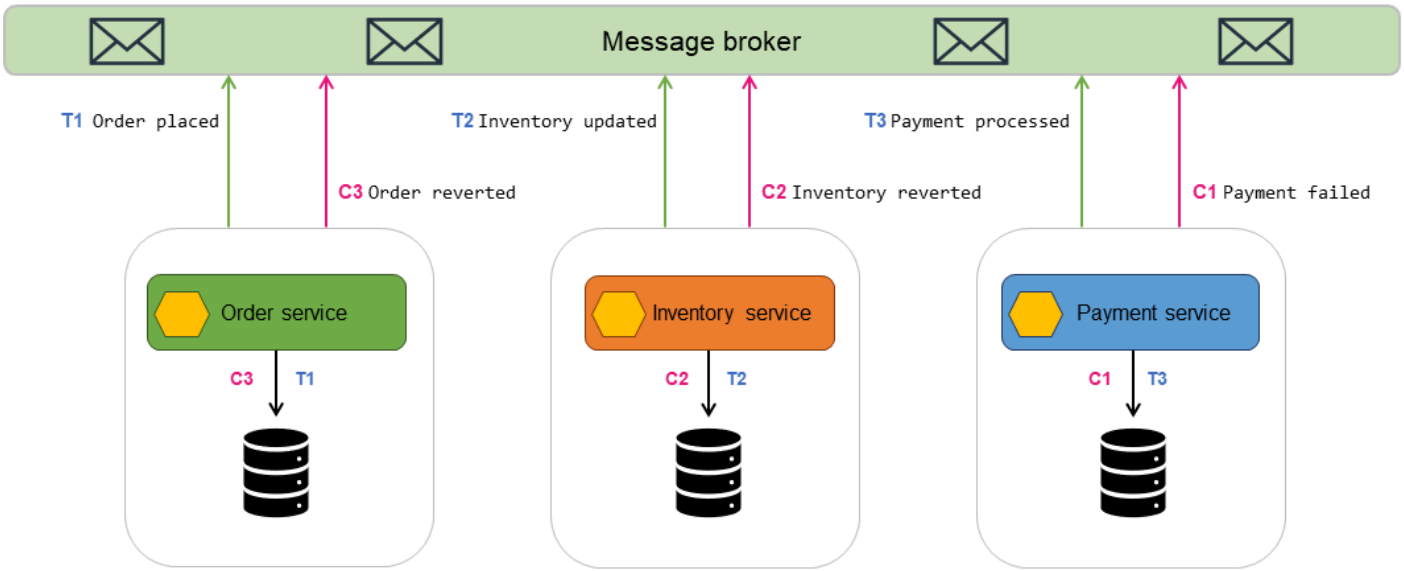
ります。このようなシナリオに対処するために、セマンティックロックを使用することをお勧めします。

- オブザーバビリティ: オブザーバビリティとは、実装とオーケストレーションのプロセスにおける問題をトラブルシューティングするための、詳細なログ記録とトレースを指します。これは、Saga を構成するサービスが増え、デバッグが複雑になる場合に重要になります。Saga コレオグラフィでは、Saga オーケストレーションと比べて、エンドツーエンドのモニタリングとレポーティングを実現するのが困難です。
- レイテンシーの問題: Saga が複数のステップで構成されている場合、補償トランザクションによって全体の応答時間にレイテンシーが加わることがあります。トランザクションが同期呼び出しを行うと、レイテンシーがさらに大きくなる可能性があります。

実装

高レベルのアーキテクチャ

以下のアーキテクチャ図では、Saga コレオグラフィは、注文サービス、在庫サービス、支払いサービスの 3 つで構成されています。トランザクションを完了するには、T1、T2、T3 の 3 ステップが必要です。3 つの補償トランザクション (C1、C2、C3) によってデータが初期状態に戻ります。



- 注文サービスはローカルトランザクション T1 を実行します。このトランザクションはデータベースをアトミックに更新し、`Order placed` メッセージをメッセージブローカーに発行します。
- 在庫サービスは注文サービスのメッセージをサブスクライブし、注文が作成されたというメッセージを受信します。

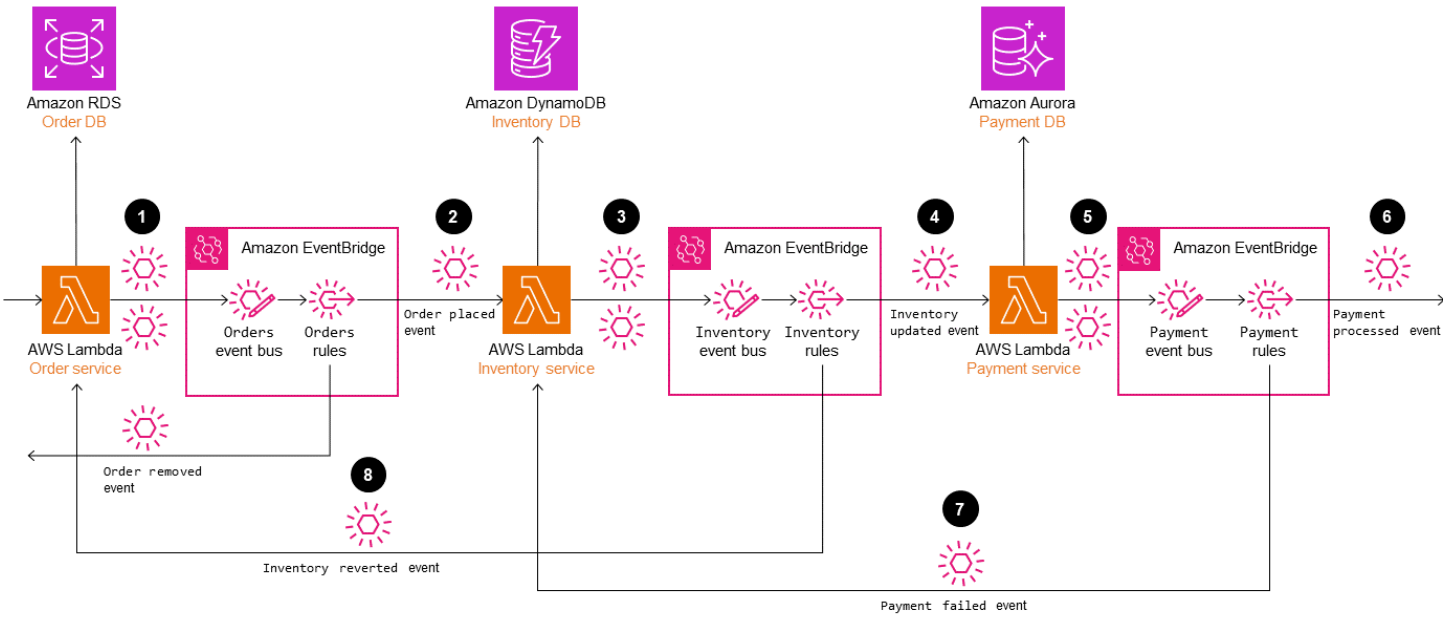
- 在庫サービスはローカルトランザクション T2 を実行します。このトランザクションはデータベースをアトミックに更新し、`Inventory updated` メッセージをメッセージブローカーに発行します。
- 支払いサービスは在庫サービスからのメッセージをサブスクライブし、在庫が更新されたというメッセージを受信します。
- 支払いサービスはローカルトランザクション T3 を実行します。このトランザクションはデータベースに支払いの詳細を付け加えてアトミックに更新し、`Payment processed` メッセージをメッセージブローカーに発行します。
- 支払いが失敗した場合、支払いサービスは補償トランザクション C1 を実行します。これにより、データベース内の支払いがアトミックに元に戻され、`Payment failed` メッセージがメッセージブローカーに発行されます。
- 補償トランザクション C2 と C3 は、データの整合性を復元するために実行されます。

AWS のサービスを使用した実装

Saga コレオグラフィパターンは Amazon EventBridge を使用して実装できます。EventBridge はイベントを使用してアプリケーションコンポーネントを接続します。イベントバスまたはパイプを介してイベントを処理します。イベントバスは、イベントを受信して、ゼロ個以上の送信先 (ターゲット) に配信するルーターです。イベントバスに関連付けられたルールによって、受信したイベントが評価され、処理のためにターゲットに送信されます。

以下のアーキテクチャは次のような構成です。

- マイクロサービス (注文サービス、在庫サービス、支払いサービス) は Lambda 関数として実装されます。
- カスタム EventBridge バスには、`Orders` イベントバス、`Inventory` イベントバス、`Payment` イベントバスの 3 つがあります。
- `Orders` ルール、`Inventory` ルール、`Payment` ルールは、対応するイベントバスに送信されるイベントに一致し、Lambda 関数を呼び出します。



成功したシナリオでは、注文が入ると以下の処理が実行されます。

1. 注文サービスはリクエストを処理し、イベントを Orders イベントバスに送信します。
2. Orders ルールがイベントに一致し、在庫サービスが開始されます。
3. 在庫サービスは在庫を更新し、イベントを Inventory イベントバスに送信します。
4. Inventory ルールがイベントに一致し、支払いサービスが開始されます。
5. 支払いサービスは支払いを処理し、イベントを Payment イベントバスに送信します。
6. Payment ルールがイベントに一致し、Payment processed イベント通知をリスナーに送信します。

   一方、注文処理に問題がある場合は、EventBridge ルールは、データ整合性と完全性を維持するために、データ更新を元に戻すための補償トランザクションを開始します。
7. 支払いが失敗した場合、Payment ルールはイベントを処理し、在庫サービスを開始します。在庫サービスは補償トランザクションを実行して在庫を元に戻します。
8. 在庫が元に戻されると、在庫サービスは Inventory reverted イベントを Inventory イベントバスに送信します。このイベントは Inventory ルールによって処理されます。注文サービスを開始し、補償トランザクションを実行して注文を削除します。

関連情報

- Saga オーケストレーションパターン
- トランザクションアウトボックスパターン
- バックオフパターンで再試行

Saga オーケストレーションパターン

Intent

Saga オーケストレーションパターンでは、セントラルコーディネーター (オーケストレーター) を使用して、複数のサービスにまたがる分散トランザクションのデータ完全性を維持します。分散トランザクションでは、トランザクションが完了する前に複数のサービスを呼び出すことができます。異なるデータストアにサービスがデータを保存する場合、これらのデータストア間でデータ整合性を維持するのが難しい場合があります。

導入する理由

トランザクションとは、複数のステップを含む可能性のある単一の作業単位です。すべてのステップが完全に実行されるか、まったく実行されないため、データストアは整合した状態を保持します。アトミック性、整合性、分離性、耐久性 (ACID) という用語によって、トランザクションの特性が定義されます。リレーショナルデータベースは ACID トランザクションを提供してデータ整合性を維持します。

トランザクションの整合性を維持するため、リレーショナルデータベースでは 2 フェーズコミット (2PC) 方式を使用します。これは準備フェーズとコミットフェーズで構成されます。

- 準備フェーズでは、調整プロセスはトランザクションを構成するプロセス (構成プロセス) に、トランザクションをコミットするかロールバックするかを約束するように要求します。
- コミットフェーズでは、調整プロセスが構成プロセスにトランザクションのコミットを要求します。準備フェーズで構成プロセスがコミットに同意できない場合、トランザクションはロールバックされます。

Database-per-service 設計パターンに従う分散システムでは、2 フェーズコミットは選択肢になりません。これは、各トランザクションがさまざまなデータベースに分散され、リレーショナルデータストアの 2 フェーズコミットと同様にプロセスを調整できる単一のコントローラーが存在しないためです。この場合の解決策の 1 つは、Saga オーケストレーションパターンを使用することです。

適用対象

Saga オーケストレーションパターンは次の場合に使用します。

- 複数のデータストアにまたがる分散トランザクションのデータ完全性と整合性がシステムに求められる場合。
- データストアに ACID トランザクションを実行するための 2PC が用意されていないため、アプリケーションの境界内で 2PC を実装する作業が複雑な場合。
- NoSQL データベース (ACID トランザクションが用意されていない) があり、1 つのトランザクションで複数のテーブルを更新する必要がある場合。

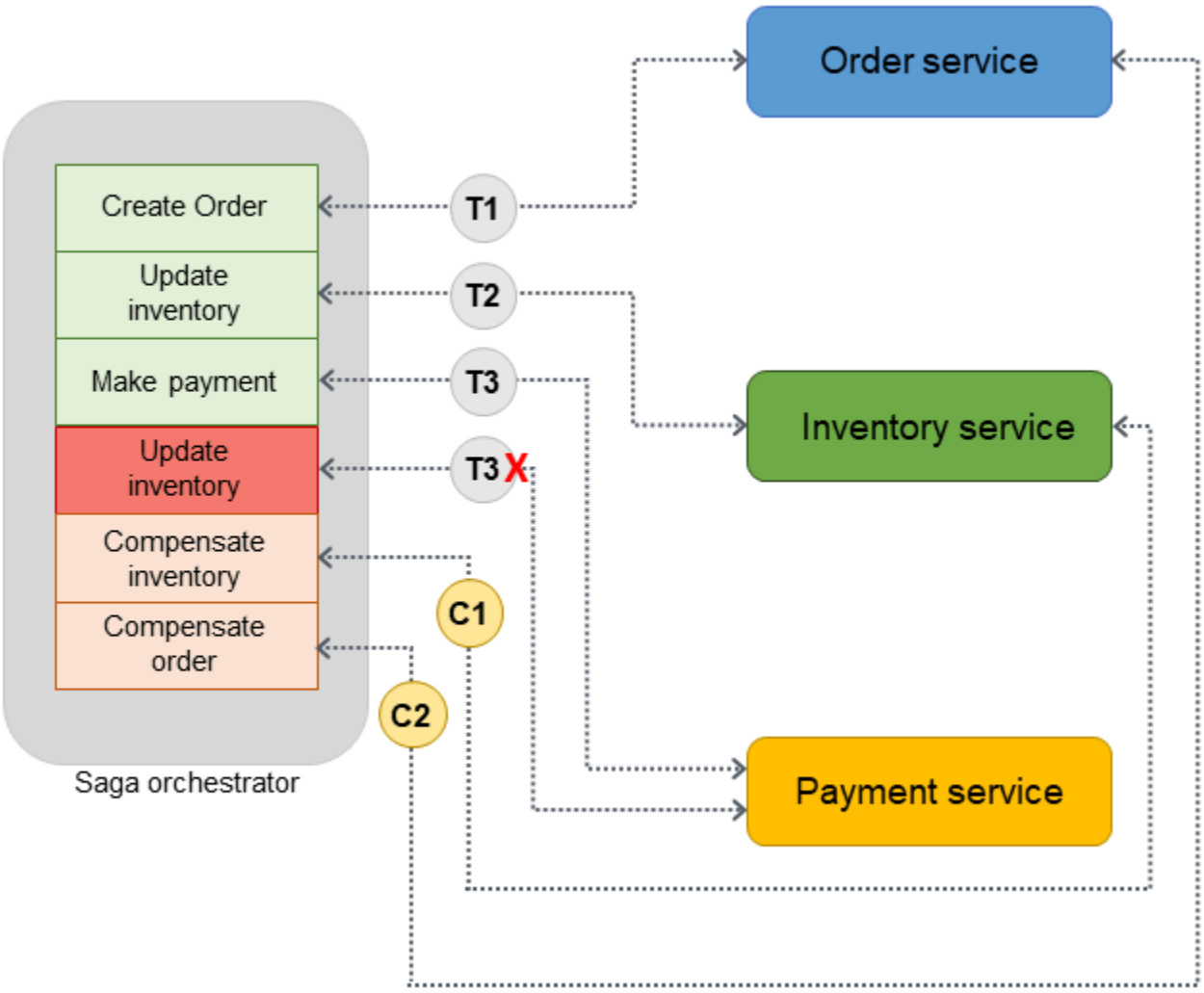
問題点と考慮事項

- 複雑さ: 補償トランザクションや再試行によってアプリケーションコードが複雑になり、メンテナンスのオーバーヘッドが発生する可能性があります。
- 結果整合性: ローカルトランザクションを順次処理することで結果整合性が保たれます。これは、強い整合性を必要とするシステムでは課題となることがあります。この問題を解決するには、整合性モデルに対するビジネスチームの期待を設定するか、強固な整合性を備えたデータストアに切り替えます。
- 冪等性: Saga を構成するサービスには、予期しないクラッシュやオーケストレーターの障害によって一時的な障害が発生した場合に繰り返し実行できるように冪等性が必要です。
- トランザクションの分離: Saga にはトランザクションの分離機能がありません。トランザクションの同時オーケストレーションによって、最新でないデータが発生する可能性があります。このようなシナリオに対処するために、セマンティックロックを使用することをお勧めします。
- オブザーバビリティ: オブザーバビリティとは、実行とオーケストレーションのプロセスにおける問題をトラブルシューティングするための、詳細なログ記録とトレースを指します。これは、Saga を構成するサービスが増え、デバッグが複雑になる場合に重要になります。
- レイテンシーの問題: Saga が複数のステップで構成されている場合、補償トランザクションによって全体の応答時間にレイテンシーが加わることがあります。このような場合は、同期呼び出しを避けてください。
- 単一障害点: オーケストレーターはトランザクション全体を調整するので、単一障害点になる可能性があります。この問題から、Saga コレオグラフィパターンが望ましい場合もあります。

実装

高レベルのアーキテクチャ

以下のアーキテクチャ図では、Saga オーケストレーターは、注文サービス、在庫サービス、支払いサービスの 3 つで構成されています。トランザクションを完了するには、T1、T2、T3 の 3 ステップが必要です。Saga オーケストレーターは手順を認識し、必要な順序で実行します。ステップ T3 に失敗 (支払い失敗) すると、オーケストレーターは補償トランザクション C1 および C2 を実行してデータを初期状態に戻します。



トランザクションが複数のデータベースに分散されている場合、AWS Step Functions を使用して Saga オーケストレーションを実装できます。

AWS サービスを使用した実装

サンプルソリューションでは、Step Functions の標準ワークフローを使用して Saga オーケストレーションパターンを実装しています。

顧客が API を呼び出すと、Lambda 関数が呼び出され、Lambda 関数で前処理が行われます。この関数は Step Functions ワークフローを開始し、分散トランザクションの処理を開始します。前処理が不要な場合は、Lambda 関数を使用せずに API Gateway から直接 Step Functions ワークフローを開始できます。

Step Functions を使用すると、Saga オーケストレーションパターンの実装に内在する単一障害点の問題が軽減されます。Step Functions には耐障害性が組み込まれており、各 AWS リージョンの複数のアベイラビリティーゾーンにわたってサービス容量を維持して、個々のマシンやデータセンターの障害からアプリケーションを保護します。これにより、サービス自体とサービスが運用するアプリケーションワークフローの両方の高可用性が確保されます。

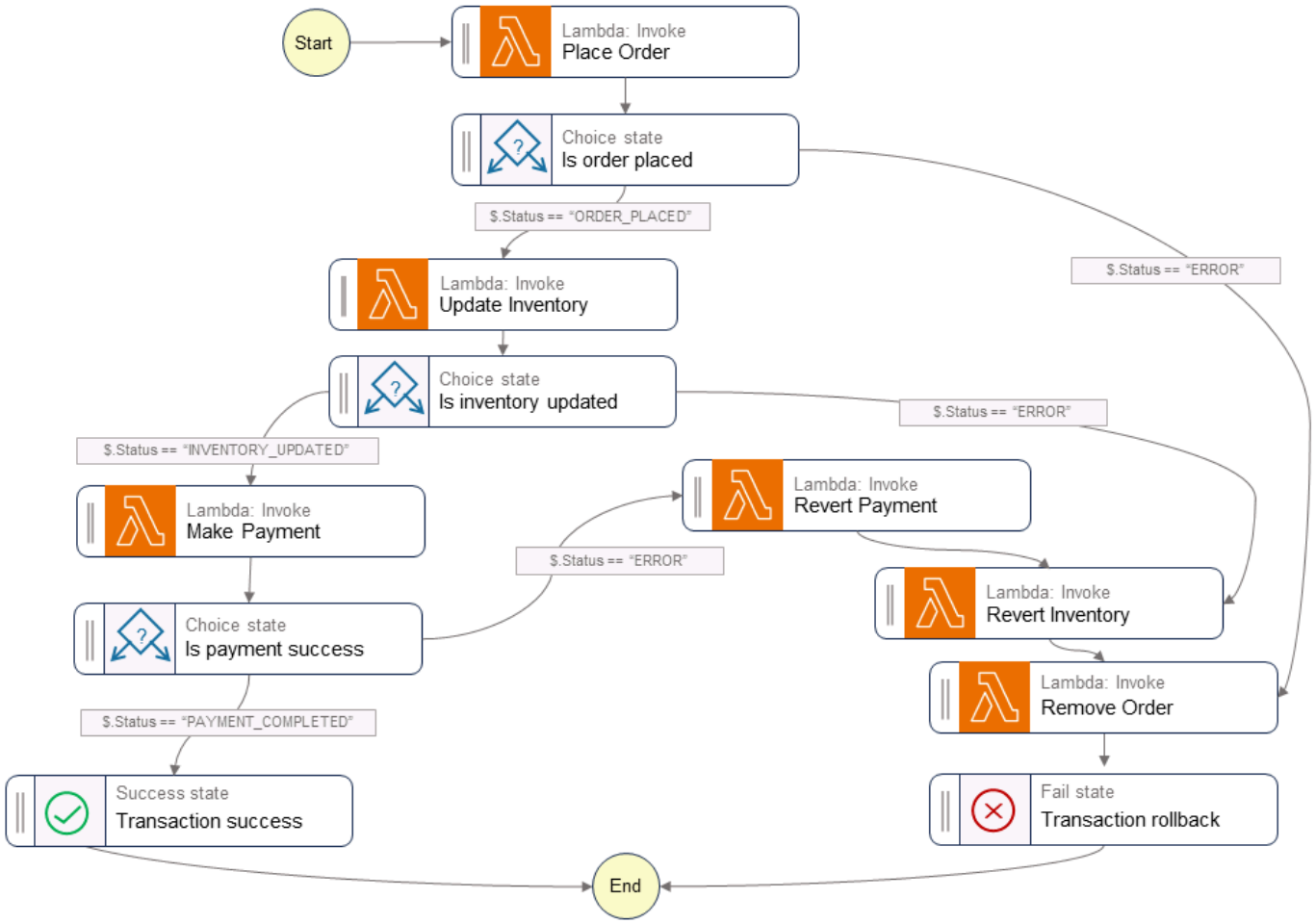
Step Functions ワークフロー

Step Functions ステートマシンでは、パターン実装の意思決定ベースの制御フロー要件を設定できます。Step Functions ワークフローは、注文、在庫更新、支払い処理のための個々のサービスを呼び出してトランザクションを完了し、さらに処理を進めるためにイベント通知を送信します。Step Functions ワークフローは、トランザクションを調整するオーケストレーターとして機能します。ワークフローにエラーがある場合、オーケストレーターは補償トランザクションを実行して、サービス間でデータの完全性が維持されるようにします。

次の図は、Step Functions ワークフロー内で実行される手順を示します。`Place Order`、`Update Inventory`、`Make Payment` の各ステップは成功した場合の径路を示しています。注文が行われ、在庫が更新され、支払いが処理されてから、呼び出し元に `Success` 状態が返されます。

`Revert Payment`、`Revert Inventory`、`Remove Order` の各 Lambda 関数は、ワークフローのいずれかのステップが失敗したときにオーケストレーターが実行する補償トランザクションを示しています。ワークフローが `Update Inventory` ステップで失敗した場合、オーケストレーターは `Revert Inventory` および `Remove Order` のステップを呼び出してから、呼び出し元に `Fail` 状

態を返します。これらの補償トランザクションにより、データの完全性を確実に維持できます。在庫は元のレベルに戻り、注文は元に戻されます。



「サンプルコード」

以下のサンプルコードは、Step Functions を使用して Saga オーケストレーターを作成する方法を示しています。コード全体を確認するには、この例の GitHub リポジトリを参照してください。

タスク定義

```
var successState = new Succeed(this,"SuccessState");
var failState = new Fail(this, "Fail");

var placeOrderTask = new LambdaInvoke(this, "Place Order", new LambdaInvokeProps
{
    LambdaFunction = placeOrderLambda,
    Comment = "Place Order",
    RetryOnServiceExceptions = false,
```

```
    PayloadResponseOnly = true
});

var updateInventoryTask = new LambdaInvoke(this,"Update Inventory", new
 LambdaInvokeProps
{
    LambdaFunction = updateInventoryLambda,
    Comment = "Update inventory",
    RetryOnServiceExceptions = false,
    PayloadResponseOnly = true
});

var makePaymentTask = new LambdaInvoke(this,"Make Payment", new LambdaInvokeProps
{
    LambdaFunction = makePaymentLambda,
    Comment = "Make Payment",
    RetryOnServiceExceptions = false,
    PayloadResponseOnly = true
});

var removeOrderTask = new LambdaInvoke(this, "Remove Order", new LambdaInvokeProps
{
    LambdaFunction = removeOrderLambda,
    Comment = "Remove Order",
    RetryOnServiceExceptions = false,
    PayloadResponseOnly = true
}).Next(failState);

var revertInventoryTask = new LambdaInvoke(this,"Revert Inventory", new
 LambdaInvokeProps
{
    LambdaFunction = revertInventoryLambda,
    Comment = "Revert inventory",
    RetryOnServiceExceptions = false,
    PayloadResponseOnly = true
}).Next(removeOrderTask);

var revertPaymentTask = new LambdaInvoke(this,"Revert Payment", new LambdaInvokeProps
{
    LambdaFunction = revertPaymentLambda,
    Comment = "Revert Payment",
    RetryOnServiceExceptions = false,
    PayloadResponseOnly = true
}).Next(revertInventoryTask);
```

```
var waitState = new Wait(this, "Wait state", new WaitProps
{
    Time = WaitTime.Duration(Duration.Seconds(30))
}).Next(revertInventoryTask);
```

ステップ関数とステートマシンの定義

```
var stepDefinition = placeOrderTask
                .Next(new Choice(this, "Is order placed")
                    .When(Condition.StringEquals("$.Status", "ORDER_PLACED"),
 updateInventoryTask
                        .Next(new Choice(this, "Is inventory updated")
                            .When(Condition.StringEquals("$.Status",
 "INVENTORY_UPDATED"),
                                makePaymentTask.Next(new Choice(this, "Is payment
 success")
                                    .When(Condition.StringEquals("$.Status",
 "PAYMENT_COMPLETED"), successState)
                                    .When(Condition.StringEquals("$.Status", "ERROR"),
 revertPaymentTask)))
                            .When(Condition.StringEquals("$.Status", "ERROR"),
 waitState)))
                    .When(Condition.StringEquals("$.Status", "ERROR"), failState));

var stateMachine = new StateMachine(this, "DistributedTransactionOrchestrator", new
 StateMachineProps {
    StateMachineName = "DistributedTransactionOrchestrator",
    StateMachineType = StateMachineType.STANDARD,
    Role = iamStepFunctionRole,
    TracingEnabled = true,
    Definition = stepDefinition
});
```

GitHub リポジトリ

このパターンのサンプルアーキテクチャの完全な実装については、https://github.com/aws-samples/saga-orchestration-netcore-blog にある GitHub リポジトリを参照してください。

ブログの参考情報

- Saga オーケストレーションパターンを使用したサーバーレス分散アプリケーションの構築

関連情報

- Saga コレオグラフィパターン
- トランザクションアウトボックスパターン

動画

次の動画は、 AWS Step Functionsを使用して Saga オーケストレーションパターンを実装する方法について説明しています。

散布-収集パターン

Intent

散布-収集パターンは、メッセージルーティングパターンであり、これによって、類似または関連するリクエストを複数の受信者にブロードキャストし、アグリゲータというコンポーネントを使用して受信者からのレスポンスを 1 つのメッセージに集約します。このパターンは、並列化の実現、処理レイテンシーの削減、非同期通信の処理に有用です。同期アプローチによる、散布-収集パターンの実装は簡単ですが、それ以上に効果的なアプローチを取るには、メッセージングサービスの有無にかかわらず、非同期通信でのメッセージルーティングとして実装する必要があります。

導入する理由

アプリケーション処理において、シーケンシャル処理に時間がかかりそうなリクエストを、複数のリクエストに分割し並列処理することができ、API コールで複数の外部システムにリクエストを送信し、レスポンスを取得することも可能です。散布-収集パターンが便利な例として、複数ソースから入力が必要な場合が挙げられます。このパターンで結果を集約すると、情報に基づく意思決定や、リクエストに最適なレスポンスの選択を行いやすくなります。

散布-収集パターンは、その名前が示すように、2 つのフェーズで構成されます。

- 散布フェーズ: リクエストメッセージを処理し、複数の受信者に並行して送信します。このフェーズの間、リクエストをネットワーク全体に散布し、即時のレスポンスを待たずに実行し続けます。
- 収集フェーズ: 受信者からレスポンスを収集し、それらをフィルタリングするか、統合レスポンスとして結合します。すべてのレスポンスを収集した後は、1 つのレスポンスに集約することも、最適なレスポンスを選択して、詳細に処理することもできます。

適用対象

散布-収集は、次の場合に使用します。

- 正確なレスポンスを作成するために、さまざまな API からデータを集約し、統合する計画を立てている。このパターンでは、各種ソースからの情報全体をまとまった形で統合できます。例えば、予約システムで、複数の受信者にリクエストを行い、複数の外部パートナーから見積もりを取得できます。

- トランザクションを完了するために、同じリクエストを複数の受信者に同時送信する必要がある。例えば、このパターンでインベントリデータを並行してクエリし、製品の可用性を確認できます。
- 信頼性とスケーラビリティに優れたシステムを実装して、複数の受信者にリクエストを配布することで、負荷分散を実現する必要がある。ある受信者が処理に失敗したり、負荷が高かったりする場合でも、他の受信者がリクエストを処理できます。
- 複数のデータソースが含まれる複雑なクエリを実装する場合に、パフォーマンスを最適化する必要がある。クエリを関連データベースに散布して、一部の結果を収集し、包括的な回答にまとめることができます。
- マップ削減処理の一種を実装しており、この処理では、複数のデータ処理エンドポイントへのルーティングによって、シャーディングとレプリケーションを行う。結果の一部をフィルタリングして結合し、適切なレスポンスを作成できます。
- key-value データベースにおいて書き込みが多いワークロードのパーティションキースペースに、書き込みオペレーションを分散する必要がある。アグリゲータによって、各シャードのデータをクエリして結果を読み取り、それらを 1 つのレスポンスに統合できます。

問題点と考慮事項

- 耐障害性: このパターンでは、並行して動作する複数の受信者に依存するため、障害を円滑に処理することが不可欠です。受信者側の失敗がシステム全体に与える影響を軽減するために、冗長性、レプリケーション、障害検出などの戦略を実装すると良いでしょう。
- スケールアウト上の制約: 処理ノードの総数が増えると、関連するネットワークオーバーヘッドも増大します。ネットワークを介したどのようなリクエストによっても、レイテンシーが増大し、並列化の利点が損なわれる可能性があります。
- レスポンス時間上のボトルネック: 最終処理の完了前にすべての受信者からのレスポンスを処理する必要があるオペレーションの場合、システム全体のパフォーマンスは、最も遅い受信者のレスポンス時間の制約を受けます。
- 部分的なレスポンス: 複数の受信者にリクエストを散布している場合、一部の受信者でタイムアウトが発生することがあります。こうした場合は、レスポンスが不完全であることがクライアントに伝わる仕組みを実装しなければなりません。UI フロントエンドを使用してレスポンス集約の詳細を表示するのも良いでしょう。
- データ整合性: 複数の受信者を対象にデータを処理する場合は、データ同期および競合解決の手法を慎重に検討して、最終的に、正確で一貫性がある集計結果を得られるようにする必要があります。

実装

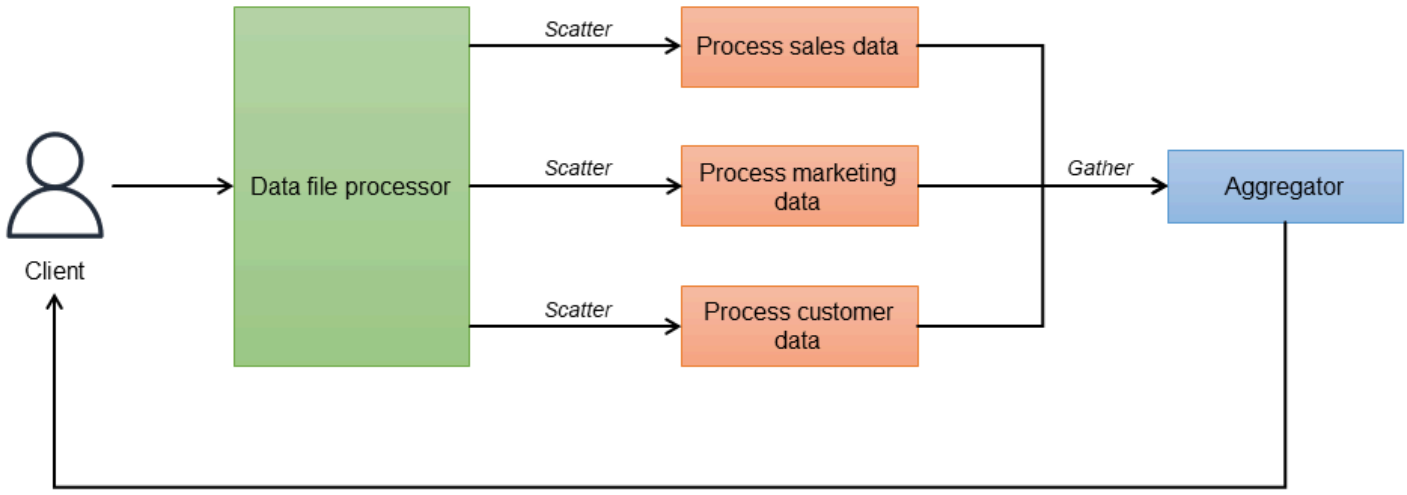
高レベルのアーキテクチャ

散布-収集パターンでは、ルートコントローラーを使用して、処理対象の受信者にリクエストを散布します。散布フェーズでは、このパターンによって 2 つの仕組みを使用し、受信者にメッセージを送信できます。

- 分散による散布: アプリケーションには、結果取得のために呼び出す必要がある既知の受信者リストがあります。受信者は、一意の関数を持つ異なるプロセスや、処理負荷を分散するためにスケールアウトされた単一のプロセスである可能性があります。いずれかの処理ノードがタイムアウトしたり、レスポンスが遅延したりすると、コントローラーによって、処理を別のノードに再分散することができます。
- オークションによる散布: パブリッシュ – サブスクライブパターンを使用して、対象の受信者にメッセージをブロードキャストします。???この場合、受信者はいつでもメッセージをサブスクライブしたり、サブスクリプションから脱退したりできます。

分散による散布

「分散による散布」方式では、ルートコントローラーによって受信リクエストを独立したタスクに分割し、使用可能な受信者に割り当てます (散布フェーズ)。各受信者 (プロセス、コンテナ、または Lambda 関数) は、独立して並列に計算を行い、レスポンスの一部を生成し、タスクを完了すると、アグリゲータに応答を送信します (収集フェーズ)。アグリゲータは、レスポンスの一部を結合し、最終結果をクライアントに返します。このワークフローを次の図に示します。

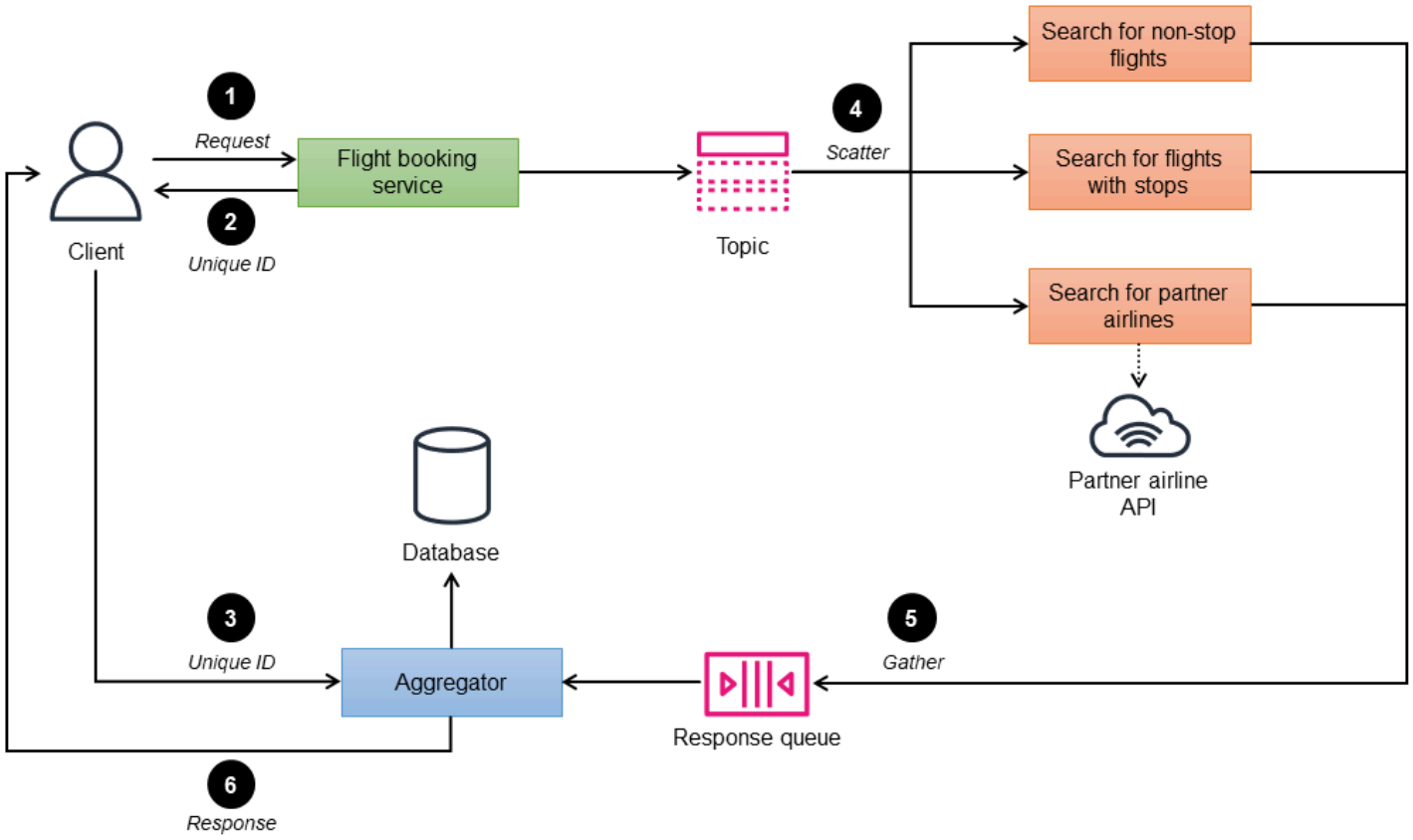


コントローラー (データファイルプロセッサ) は、一連の呼び出し全体をオーケストレーションするもので、呼び出す予約エンドポイントをすべて認識しています。コントローラーにタイムアウトパラメータを設定すると、時間がかかりすぎるレスポンスを無視できます。リクエストが送信されると、アグリゲータは各エンドポイントからのレスポンスを待ちます。レジリエンスを実装する場合は、各マイクロサービスを複数のインスタンスで構成してデプロイし、負荷分散を行うと良いでしょう。アグリゲータは、結果を取得してそれらを 1 つのレスポンスメッセージに結合します。さらに重複データを削除して、その後の処理に備えます。タイムアウトしたレスポンスは、無視されます。別のアグリゲータサービスを使用せず、コントローラーをアグリゲータとして機能させることもできます。

オークションによる散布

コントローラーが受信者を認識していない場合、または受信者が疎結合の場合は、「オークションによる散布」方式を使用できます。この方式では、受信者がトピックをサブスクライブし、コントローラーがトピックにリクエストを発行します。受信者は、結果をレスポンスキューに発行します。ルートコントローラーは受信者を認識していないため、収集プロセスでアグリゲータ (別のメッセージングパターン) を使用してレスポンスを収集し、単一のレスポンスメッセージとして抽出します。アグリゲータは、一意の ID を使用してリクエストグループを識別します。

次の図に示す例では、「オークションによる散布」方式を使用して、航空会社のウェブサイトにフライト予約サービスを実装しています。ウェブサイトでは、ユーザーが航空会社独自のキャリアと、そのパートナーキャリアからフライトを検索して表示でき、検索のステータスもリアルタイムで表示される必要があります。フライト予約サービスは、直行便、経由便、パートナー航空会社という 3 つの検索マイクロサービスで構成されます。パートナー航空会社を検索する際には、パートナーの API エンドポイントを呼び出してレスポンスを取得します。

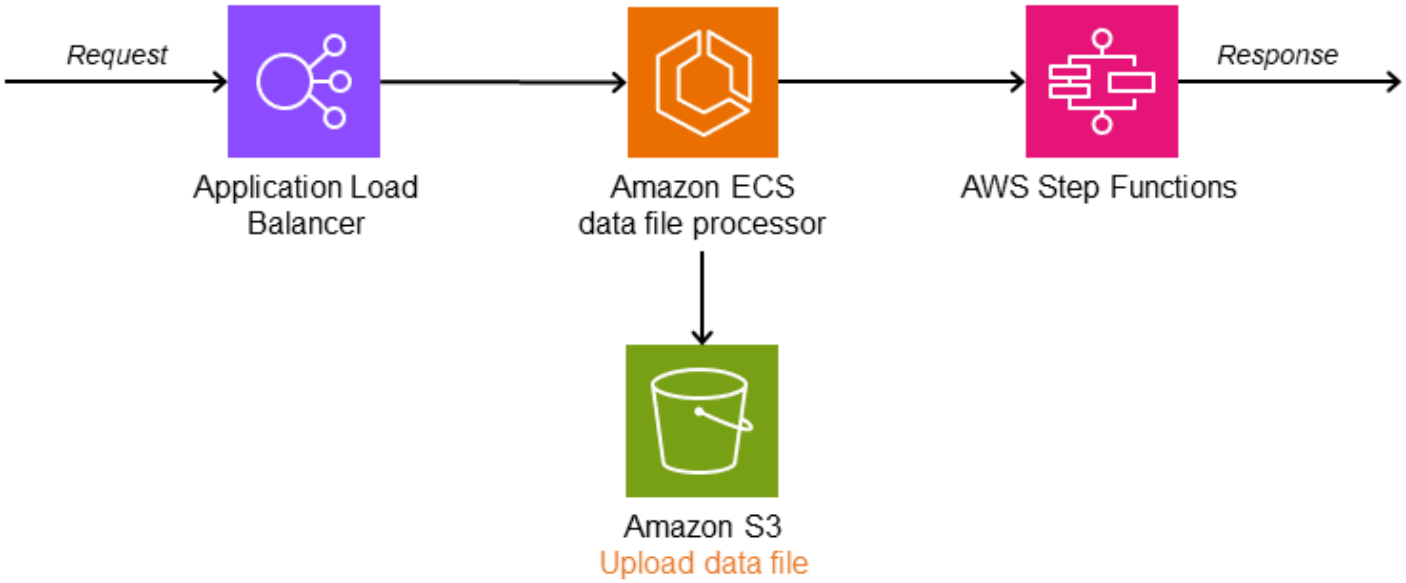


1. フライト予約サービス (コントローラー) が、検索条件をクライアントからの入力として受け取り、リクエストを処理してトピックに発行します。
2. コントローラーは、一意の ID を使用して、各リクエストグループを識別します。
3. クライアントが、ステップ 6 の一意の ID をアグリゲータに送信します。
4. 予約検索マイクロサービス (予約トピックをサブスクライブ済み) が、リクエストを受け取ります。
5. マイクロサービスは、リクエストを処理し、指定された検索条件の座席が利用可能かどうかレスポンスキューに返します。
6. アグリゲータが、すべてのレスポンスメッセージを照合し、一時データベースに保存します。さらに、フライトを一意の ID でグループ化して、単一の統合レスポンスを作成し、クライアントに送信します。

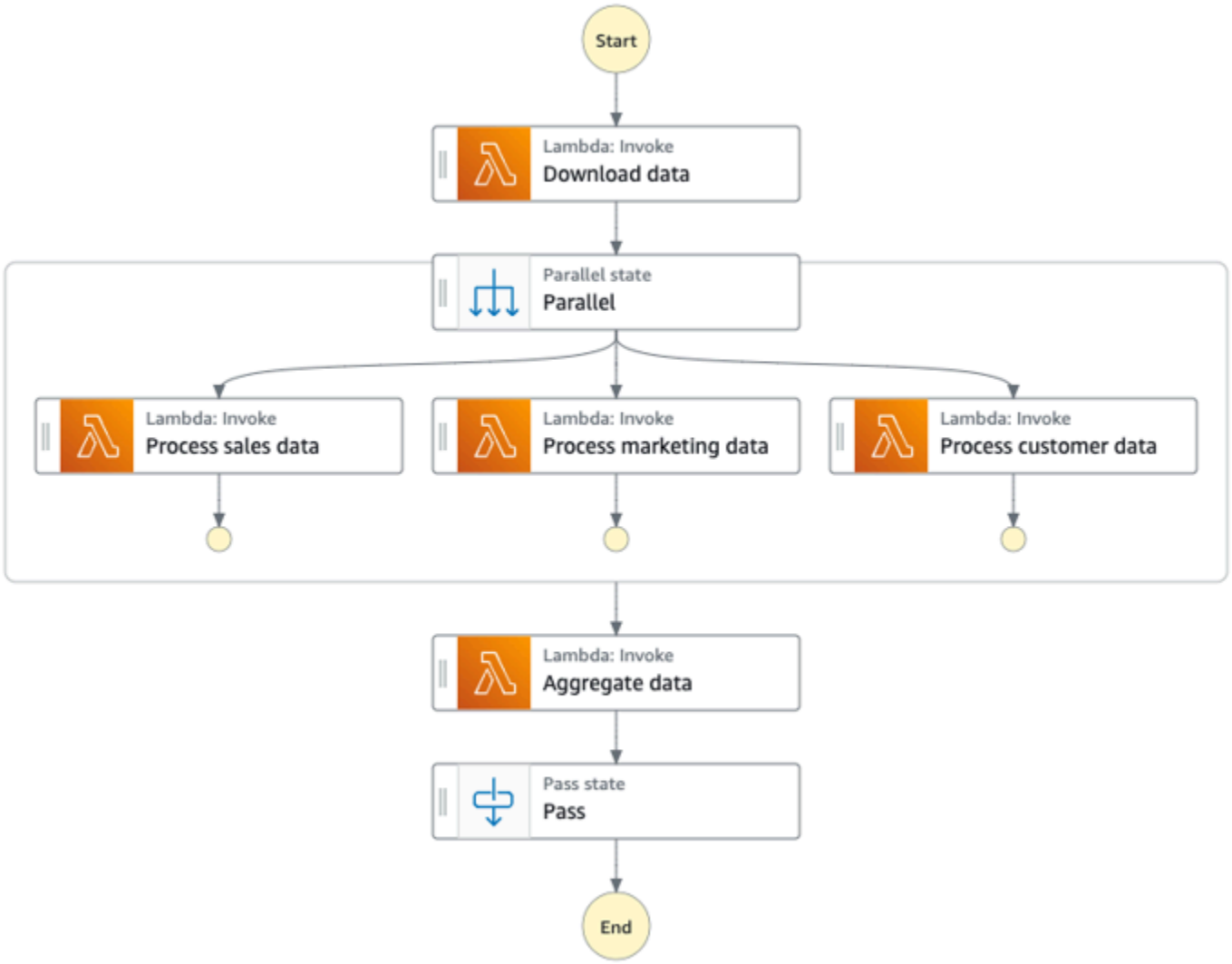
を使用した実装 AWS のサービス

分散による散布

次のアーキテクチャでは、ルートコントローラーは、受信リクエストデータを個々の Amazon Simple Storage Service (Amazon S3) バケットに分割し、ワークフローを開始するデータファイルプロセッサ (Amazon ECS) です。 AWS Step Functions このワークフローにより、データをダウンロードして、並列ファイル処理を開始します。`Parallel` ステートにより、すべてのタスクからレスポンスが返るのを待ち、 AWS Lambda 関数はデータを集約し、Amazon S3 に保存します。

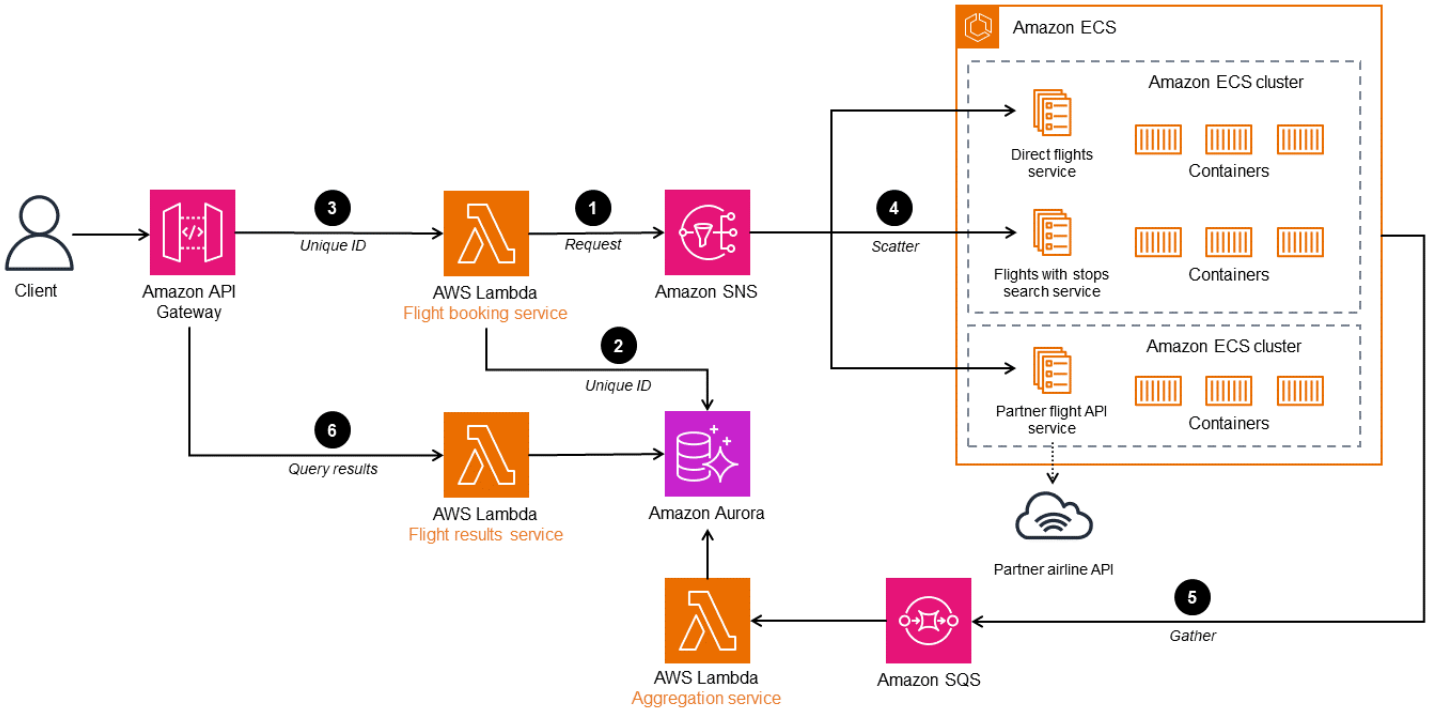


次の図は、`Parallel` ステートの Step Functions ワークフローを示しています。



オークションによる散布

次の図は、入札方法別の散布図の AWS アーキテクチャを示しています。ルートコントローラーであるフライト予約サービスによって、フライト検索リクエストを複数のマイクロサービスに分散します。パブリッシュ – サブスクライブチャネルは、通信用のマネージドメッセージングサービスである Amazon Simple Notification Service (Amazon SNS) を使用して実装します。Amazon SNS では、分離されたマイクロサービスアプリケーション間のメッセージ、またはユーザーへの直接通信がサポートされています。受信者のマイクロサービスを、Amazon Elastic Kubernetes Service (Amazon EKS) または Amazon Elastic Container Service (Amazon ECS) にデプロイすると、管理とスケーラビリティが向上します。フライト結果サービスにより、結果がクライアントに返ります。または Amazon ECS AWS Lambda や Amazon EKS などの他のコンテナオーケストレーションサービスに実装できます。



1. フライト予約サービス (コントローラー) が、検索条件をクライアントからの入力として受け取り、リクエストを処理して SNS トピックに発行します。
2. このコントローラーは、一意の ID を Amazon Aurora データベースに発行して、リクエストを識別します。
3. クライアントが、ステップ 6 の一意の ID をクライアントに送信します。
4. 予約検索マイクロサービス (予約トピックをサブスクライブ済み) が、リクエストを受け取ります。
5. マイクロサービスは、リクエストを処理し、指定された検索条件の座席が利用可能かどうかを Amazon Simple Queue Service (Amazon SQS) のレスポンスキューに返します。アグリゲータが、すべてのレスポンスメッセージを照合し、一時データベースに保存します。
6. フライト結果サービスが、フライトを一意の ID でグループ化して、単一の統合レスポンスを作成し、クライアントに送信します。

このアーキテクチャに別の航空会社検索を追加する場合は、その SNS トピックをサブスクライブし、SQS キューへの発行を行うマイクロサービスを追加します。

要約すると、散布-収集パターンを実装した分散システムでは、効率的な並列化の実現、レイテンシーの削減、非同期通信のシームレスな処理が可能になります。

GitHub リポジトリ

このパターンのサンプルアーキテクチャの完全な実装については、https://github.com/aws-samples/asynchronous-messaging-workshop/tree/master/code/lab-3 の GitHub リポジトリを参照してください。

ワークショップ

- 散布-収集のラボ (Decoupled Microservices ワークショップ)

ブログの参考情報

- Application integration patterns for microservices

関連情報

- パブリッシュ – サブスクライブパターン

ストラングラーフィグパターン

Intent

strangler fig パターンは、モノリシックアプリケーションをマイクロサービスアーキテクチャに段階的に移行して、変革上のリスクやビジネスの中断を軽減するのに有用です。

導入する理由

モノリシックアプリケーションは、ほとんどの機能を単一のプロセスまたはコンテナ内で提供できるように設計されています。コードは、密結合の状態にあります。そのため、アプリケーションの変更時には、徹底的な再テストを行い、リグレッションの問題を回避しなければなりません。変更は個別にテストできず、サイクル時間にその影響が及びます。アプリケーションが多くの機能で強化されているため、かなりの複雑さによって、メンテナンスにかかる時間や市場投入までの時間が長くなり、結果的に、製品のイノベーションが遅れる可能性があります。

アプリケーションの規模が大きくなると、チームの認知負荷が増大すると同時に、チームの所有権の境界が不明瞭になる可能性があります。負荷に応じて個々の機能をスケーリングすることはできず、ピーク負荷に対応するには、アプリケーション全体をスケーリングしなければなりません。また、システムが古くなると、技術が時代遅れになり、サポートコストが増加する可能性があります。モノリシックなレガシーアプリケーションは、開発時に利用可能だったベストプラクティスに従っており、分散を目的とするものではありません。

モノリシックアプリケーションをマイクロサービスアーキテクチャに移行すると、コンポーネントを小規模化して分割できます。これらのコンポーネントは、スケールやリリースを独立して行え、個々のチームが所有することも可能です。これにより、変更を短期間で行えます。変更をローカライズすると共に、テストとリリースを迅速化できるからです。コンポーネントが疎結合されており、個別にデプロイ可能なため、変更の影響範囲も狭められます。

コードの書き換えやリファクタリングによってモノリスを完全にマイクロサービスアプリケーションに置き換える作業には、非常に労力がかかり、大きなリスクも伴います。ビッグバン方式の移行、つまり、一度の作業によるモノリスの移行は、変革上のリスクやビジネスの中断につながります。また、アプリケーションのリファクタリング中に、新機能を追加することは、非常に困難か、不可能です。

この問題を解決する方法の 1 つは、Martin Fowler 氏が提唱した strangler fig パターンの使用です。このパターンでは、機能を徐々に抽出し、既存システムを中心に新しいアプリケーションを構築することで、マイクロサービスに移行します。モノリスの機能を段階的にマイクロサービスに置き換え

て、アプリケーションユーザーが移行済み機能を徐々に利用できるようにします。すべての機能を新しいシステムに移行したら、モノリシックアプリケーションは安全に廃止できます。

適用対象

strangler fig パターンは、次の場合に使用します。

- モノリシックアプリケーションをマイクロサービスアーキテクチャに徐々に移行する必要がある。
- モノリスの規模と複雑さを考えると、ビッグバン方式の移行アプローチにはリスクがある。
- ビジネス上、新機能の追加が必要なため、変革の完了を待つことはできない。
- 変革中には、エンドユーザーへの影響を最小化する必要がある。

問題点と考慮事項

- コードベースへのアクセス: strangler fig パターンを実装するには、モノリスアプリケーションのコードベースにアクセスできる必要があります。機能をモノリスから移行した後は、軽微なコード変更を行い、モノリス内に破損防止レイヤーを実装して、呼び出しを新しいマイクロサービスにルーティングしなければなりません。コードベースにアクセスできないと、呼び出しをインターセプトできません。コードベースへのアクセスは、受信リクエストのリダイレクトにも重要となります。なぜなら、プロキシレイヤーが移行済み機能の呼び出しをインターセプトしてマイクロサービスにルーティングできるように、一部のコードリファクタリングが必要になる場合があるからです。
- 不明瞭なドメイン: ステムの早期分割にはコストがかかる可能性があり、特に、ドメインが明確でない場合にそれが顕著です。また、サービス境界が誤解されかねません。ドメイン駆動型設計 (DDD) はドメインを把握する仕組みを、イベントストーミングはドメインの境界を決定する手法を意味します。
- マイクロサービスの識別: DDD は、マイクロサービスを識別する主要なツールとして使用でき、マイクロサービスを識別するには、サービスクラス間の自然な分割を特定します。サービスの多くは、独自のデータアクセスオブジェクトを所有しているため、分割が容易であり、関連するビジネスロジックを持つサービスや、依存関係がまったくないか少ないクラスは、マイクロサービスに適しています。モノリスを分解する前にコードをリファクタリングすると、密結合を防ぐことができます。また、コンプライアンス要件、リリース頻度、チームの地理的位置、スケーリングのニーズ、ユースケース主導の技術ニーズ、チームの認知負荷なども考慮する必要があります。
- 腐敗防止層: 移行プロセス中に、モノリス内の機能がマイクロサービスとして移行済みの機能を呼び出す必要がある場合は、腐敗防止層 (ACL) を実装して、各呼び出しを適切なマイクロサービス

にルーティングする必要があります。モノリス内の既存の呼び出し元を切り離し、変更を防ぐために、ACL を、呼び出しを新しいインターフェイスに変換するアダプターまたはファサードとして機能させます。この点については、このガイドの前半にある実装セクションで詳しく説明しています。

- プロキシレイヤーでの障害: 移行中、プロキシレイヤーでは、モノリシックアプリケーションに送信されるリクエストをインターセプトし、レガシーシステムまたは新しいシステムにルーティングします。しかし、こうしたプロキシレイヤーは、単一障害点やパフォーマンス上のボトルネックになる可能性があります。
- アプリケーションの複雑さ: strangler fig パターンの利点は、モノリスが大規模な場合に最大化します。完全なリファクタリングがあまり複雑でない小規模アプリケーションでは、移行するよりも、マイクロサービスアーキテクチャ内でアプリケーションを書き換える方が効率的かもしれません。
- サービスインタラクション: マイクロサービスでは、同期的にも非同期的にも通信を行えます。同期通信が要件の場合は、タイムアウトによって接続またはスレッドプールが消費され、アプリケーションパフォーマンスの問題が発生する可能性がないかどうかを検討してください。発生する場合は、サーキットブレーカーパターンを使用して、長時間に及びかねないオペレーションの失敗がすぐに返るようにします。非同期通信を実現するには、イベントとメッセージングキューを使用します。
- データ集約: マイクロサービスアーキテクチャのデータは、複数のデータベースに分散されます。データ集約が要件の場合は、フロントエンドで AWS AppSync を使用するか、バックエンドでコマンドクエリ責任分離 (CQRS) パターンを使用できます。
- データ整合性: マイクロサービスにはデータストアがあり、モノリシックアプリケーションもこのデータを使用する場合があります。データを共有するには、キューとエージェントを使用して、新しいマイクロサービスのデータストアをモノリシックアプリケーションのデータベースと同期すると良いでしょう。ただし、2 つのデータストア間でデータの冗長性と結果整合性が生じる可能性があるため、データレイクなどの長期的なソリューションを確立するまでは、戦術的なソリューションとして扱うことをお勧めします。

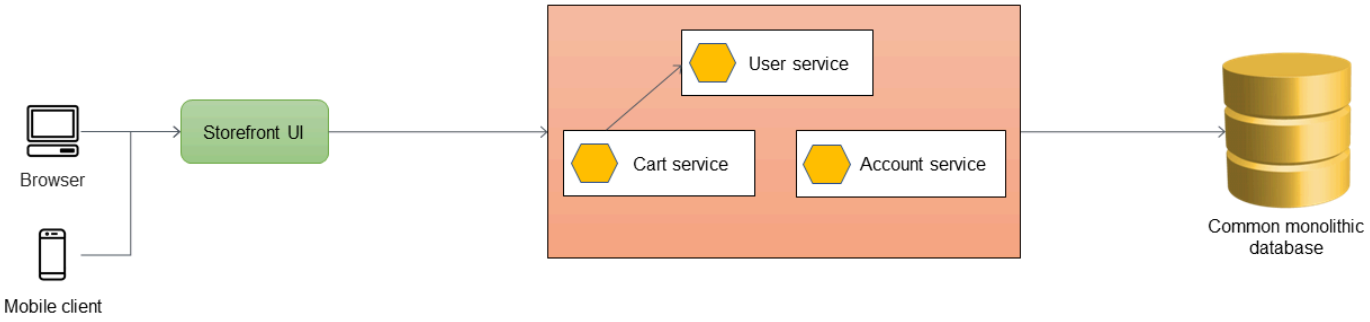
実装

strangler fig パターンは、特定の機能を、1 つのコンポーネントにした新しいサービスまたはアプリケーションに置き換える方式で、置き換えは、コンポーネントごとに行います。プロキシレイヤーでは、モノリシックアプリケーションに送信されるリクエストをインターセプトし、レガシーシステムまたは新しいシステムにルーティングします。このレイヤーにより、ユーザーが適切なアプリケーションにルーティングされるため、モノリスの機能を継続しながら、新しいシステムに機能を追加で

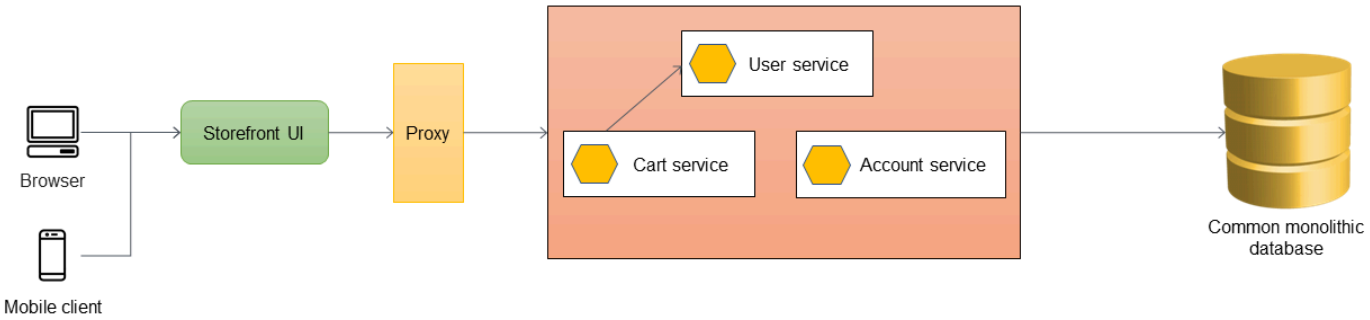
きます。古いシステムの機能が新しいシステムの機能にすべて置き換わったら、旧機能を廃止できます。

高レベルのアーキテクチャ

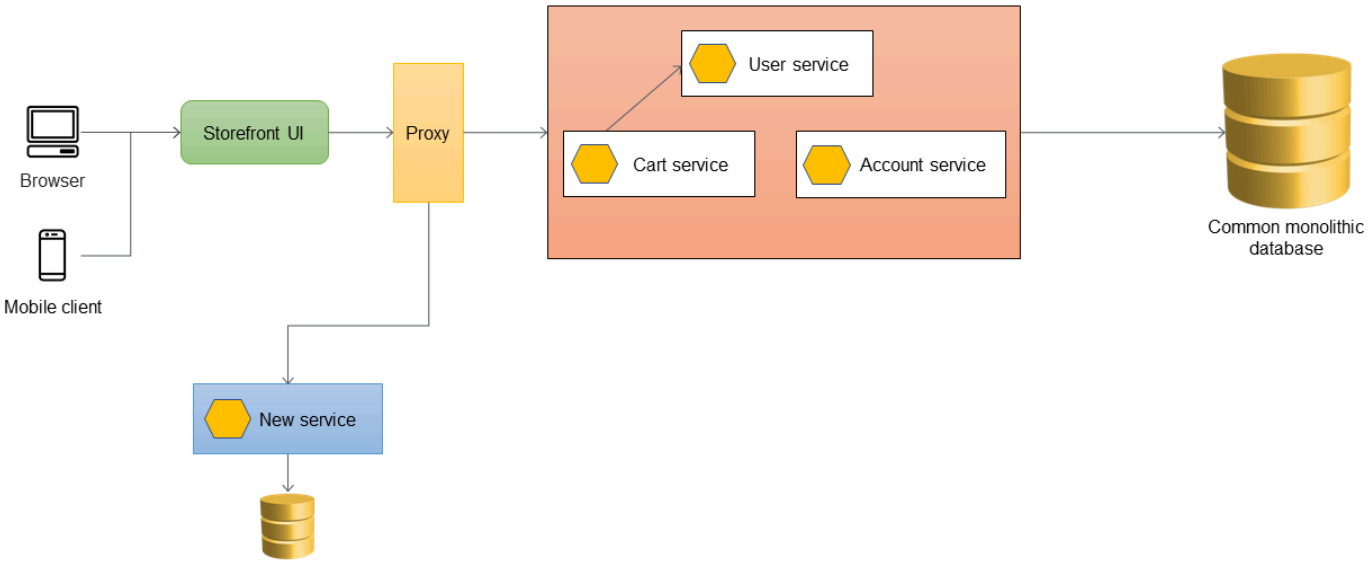
次の図に示すモノリシックアプリケーションには、ユーザーサービス、カートサービス、アカウントサービスという 3 つのサービスがあります。カートサービスはユーザーサービスに依存しており、アプリケーションではモノリシックリレーショナルデータベースが使用されています。



最初のステップでは、ストアフロント UI とモノリシックアプリケーションの間にプロキシレイヤーを追加します。開始時、プロキシでは、すべてのトラフィックがモノリシックアプリケーションにルーティングされます。

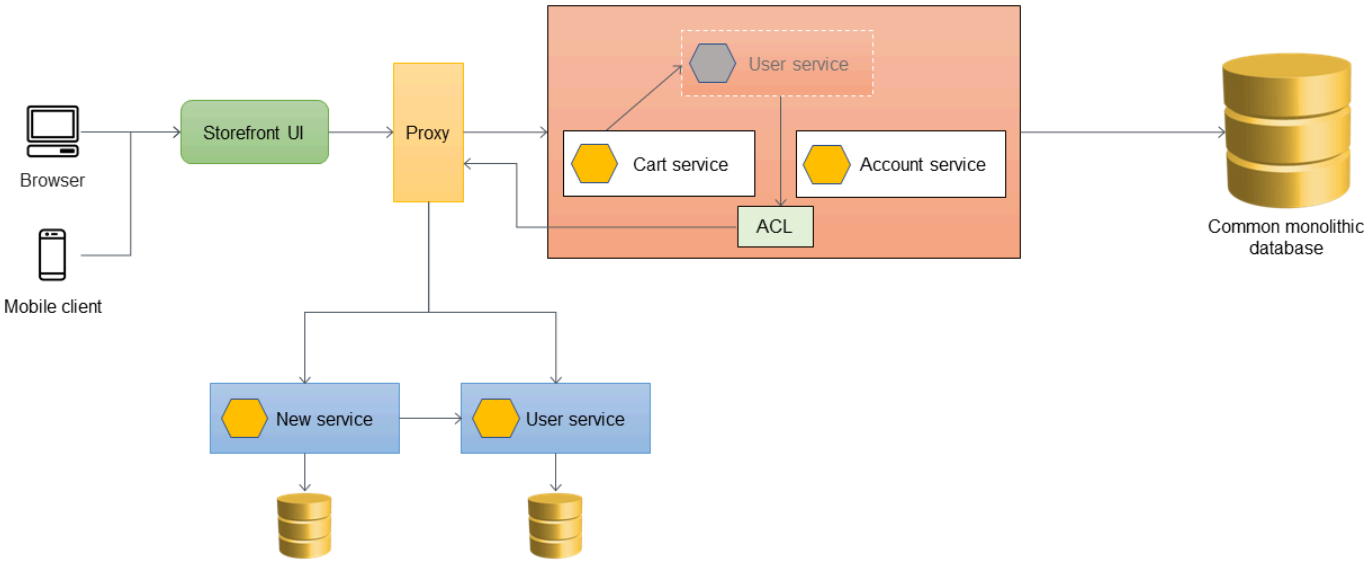


アプリケーションに新しい機能を追加する場合は、既存のモノリスには追加せず、その機能を新しいマイクロサービスとして実装します。ただし、モノリスのバグは引き続き修正し、アプリケーションの安定性を確保します。次の図に示すプロキシレイヤーでは、API URL に基づいて、呼び出しをモノリスまたは新しいマイクロサービスにルーティングしています。



腐敗防止層の追加

次のアーキテクチャでは、ユーザーサービスがマイクロサービスに移行済みです。カートサービスはユーザーサービスを呼び出しますが、対象サービスはモノリス内で利用できなくなっています。また、新しく移行したサービスのインターフェイスが、モノリシックアプリケーション内にある以前のインターフェイスと一致しない可能性もあります。こうした変更に対処するには、ACL を実装します。移行プロセス中に、モノリス内の機能がマイクロサービスとして移行済みの機能を呼び出す必要がある場合、ACL によって、呼び出しを新しいインターフェイスに変換し、適切なマイクロサービスにルーティングします。

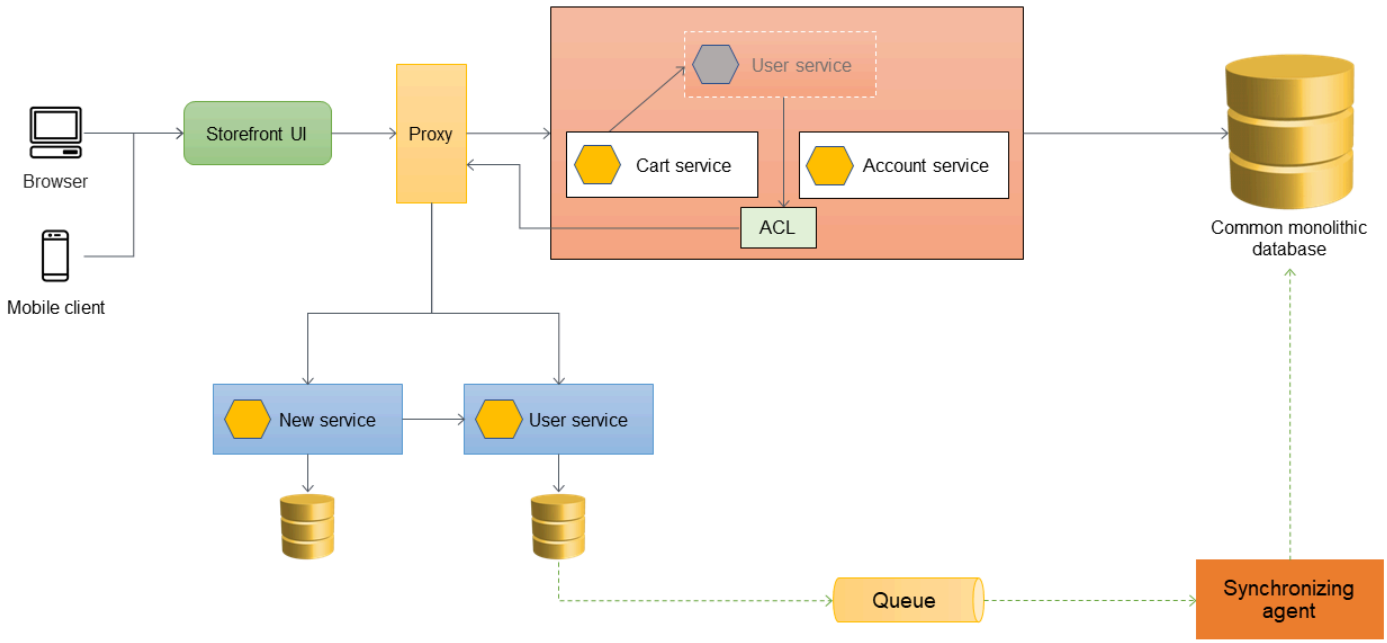


ACL は、モノリシックアプリケーション内に、移行済みサービスに固有のクラス (`UserServiceFacade` や `UserServiceAdapter` など) として実装できます。また、ACL は、それに依存するサービスをマイクロサービスアーキテクチャにすべて移行した後に廃止する必要があります。

ACL を使用する場合、カートサービスは引き続きモノリス内のユーザーサービスを呼び出し、ユーザーサービスは ACL を介して呼び出しをマイクロサービスにリダイレクトします。カートサービスは、マイクロサービスへの移行を認識せずにユーザーサービスを呼び出す必要があります。リグレッションやビジネスの中断を軽減するには、こうした疎結合が必要です。

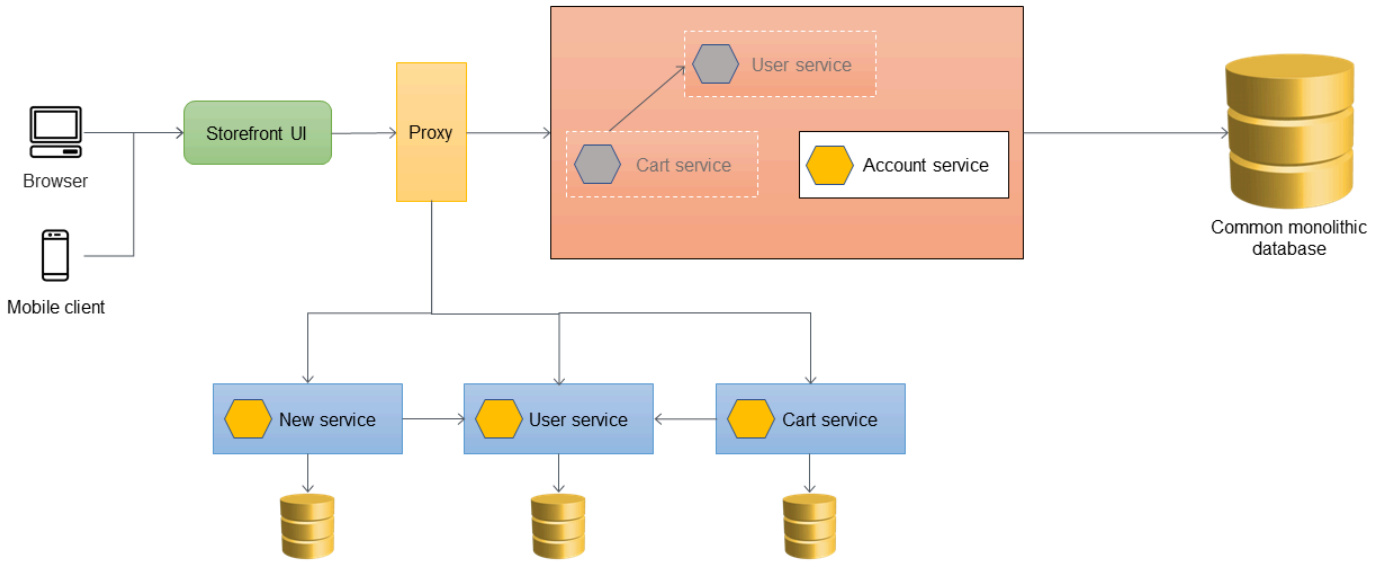
データ同期の処理

ベストプラクティスを実装するには、マイクロサービス内に自身のデータを所有する必要があるため、ユーザーサービスでは、そのデータを独自のデータストアに保存します。場合によっては、データをモノリシックデータベースと同期する必要があり、これによって、レポートなどの依存関係を処理すると共に、マイクロサービスに直接アクセスする準備が整っていないダウンストリームアプリケーションに対応します。モノリシックアプリケーションでは、まだマイクロサービスに移行済みでない他の関数やコンポーネントのデータが必要になる場合もあるため、新しいマイクロサービスとモノリスのデータを同期しなければなりません。データを同期するには、次の図に示すように、ユーザーマイクロサービスとモノリシックデータベースの間に同期エージェントを導入すると良いでしょう。これにより、ユーザーマイクロサービスは、データベースが更新されるたびにイベントをキューに送信し、同期エージェントは、キューをリッスンし、モノリシックデータベースを継続的に更新します。そうすることで、モノリシックデータベース内のデータと同期対象データとの間に、結果整合性が確保されます。



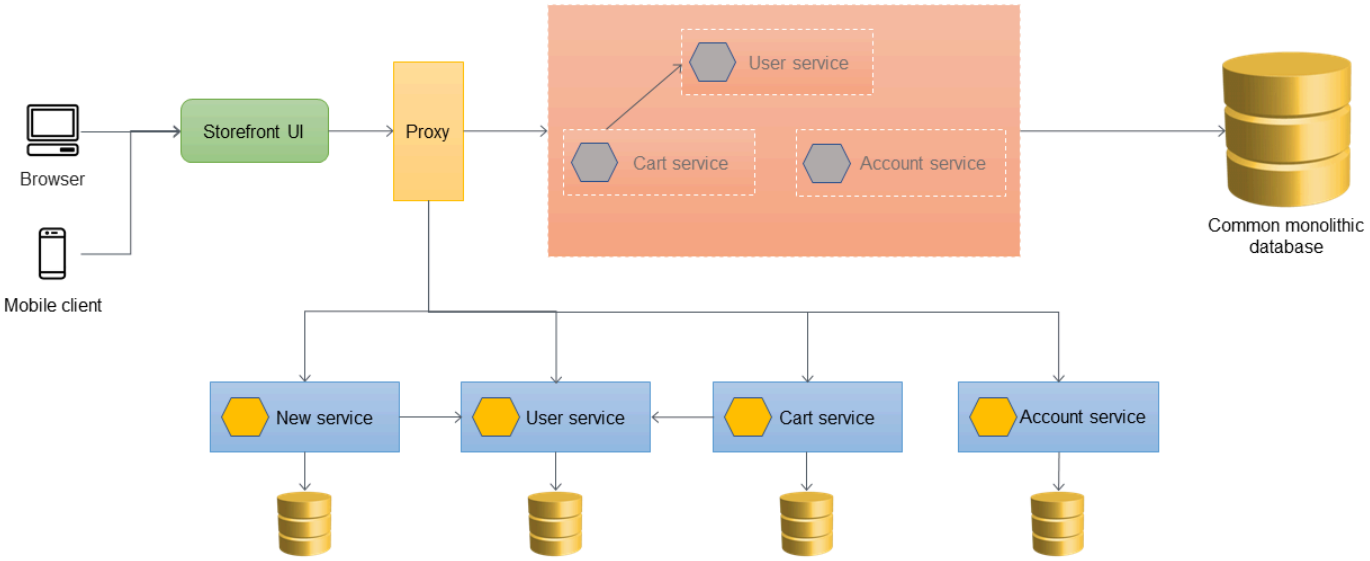
別のサービスの移行

カートサービスをモノリシックアプリケーションから移行する際、そのコードを修正し、新しいサービスを直接呼び出すようにします。これにより、ACL ではそれらの呼び出しがルーティングされなくなります。このアーキテクチャを以下に図で示します。

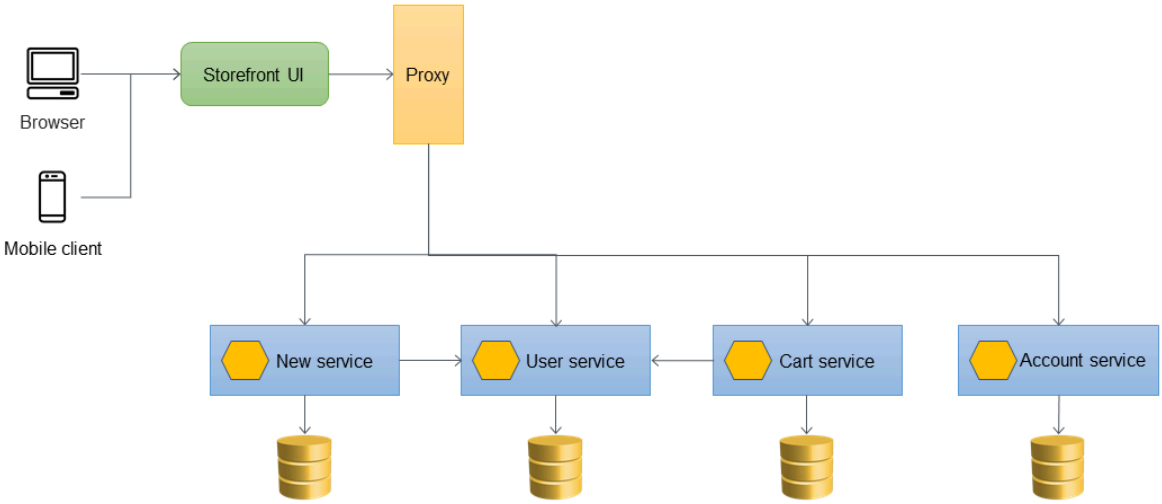


次の図は、strangler fig パターンによる移行の最終状態を示しており、すべてのサービスがモノリスから移行済みで、モノリスのスケルトンのみが残っています。履歴データは、個々のサービスで所有

されているデータストアに移行できます。ACL は削除可能な状態であり、この段階で、モノリスを廃止する準備が整います。



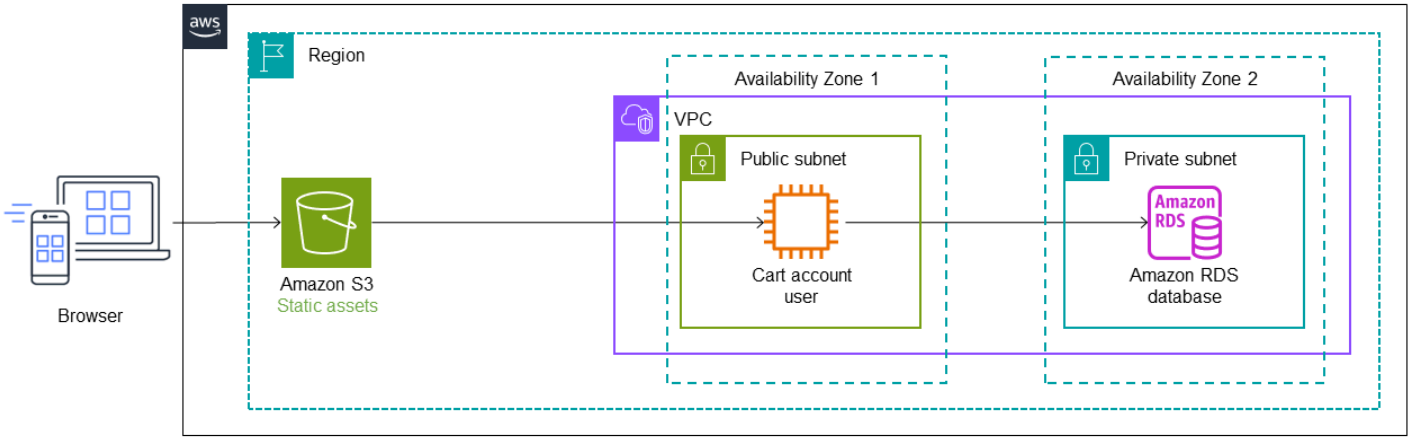
次の図は、モノリシックアプリケーション廃止後の最終的なアーキテクチャを示しています。個々のマイクロサービスは、リソースベースの URL (`http://www.storefront.com/user` など) またはアプリケーションの要件に基づいて、独自のドメイン (`http://user.storefront.com` など) 経由でホストできます。ホスト名とパスを使用して HTTP API をアップストリームコンシューマーに公開する主な方法の詳細については、「API ルーティングパターン」セクションを参照してください。



AWS サービスを使用した実装

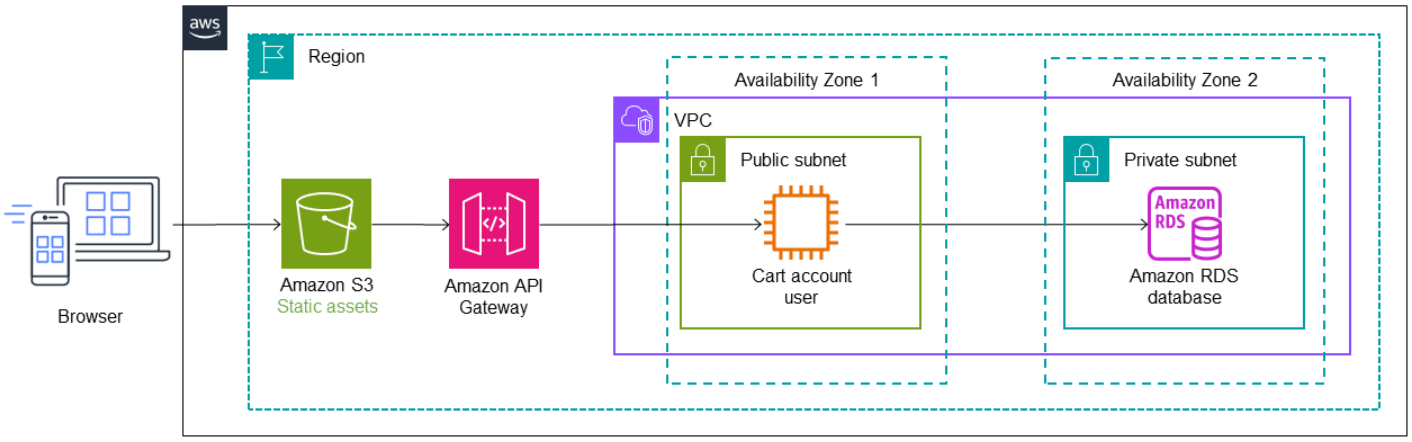
アプリケーションプロキシとしての API Gateway の使用

次の図は、モノリシックアプリケーションの初期状態を示しています。lift-and-shift戦略 AWS を使用して に移行されたと仮定して、Amazon Elastic Compute Cloud (Amazon EC2) インスタンスで実行され、Amazon Relational Database Service (Amazon RDS) データベースを使用しているとします。単純化するために、このアーキテクチャでは、1 つのプライベートサブネットと 1 つのパブリックサブネットがある 1 つの仮想プライベートクラウド (VPC) を使用しているとします。また、マイクロサービスは最初に同じ AWS アカウント内にデプロイすると仮定します。(本番環境のベストプラクティスを実装するには、マルチアカウントアーキテクチャを使用してデプロイの独立性を確保します)。EC2 インスタンスは、パブリックサブネット内にある単一のアベイラビリティーゾーンに存在し、RDS インスタンスは、プライベートサブネット内にある単一のアベイラビリティーゾーンに存在しています。Amazon Simple Storage Service (Amazon S3) には、ウェブサイトの JavaScript、CSS、React ファイルといった、静的アセットを保存します。

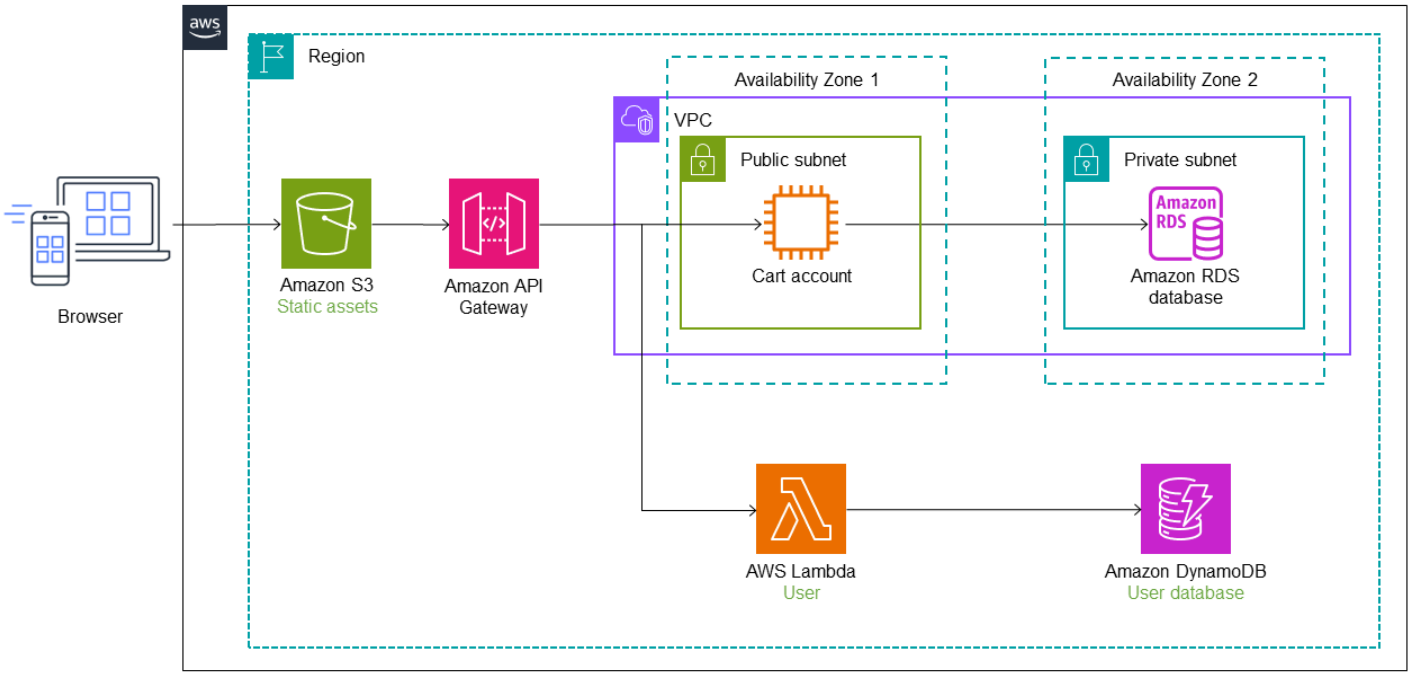


次のアーキテクチャでは、AWS Migration Hub Refactor Spaces によって、モノリシックアプリケーションの前に Amazon API Gateway をデプロイしています。Refactor Spaces によってアカウント内にリファクタリングインフラストラクチャを構築し、API Gateway を、モノリスへの呼び出しをルーティングするプロキシレイヤーとして機能させます。初期状態では、すべての呼び出しがプロキシレイヤーを介してモノリシックアプリケーションにルーティングされます。前述したとおり、プロキシレイヤーは単一障害点になる可能性がありますが、API Gateway はサーバーレスのマルチ AZ サービスであるため、これをプロキシとして使用すると、リスクが軽減されます。

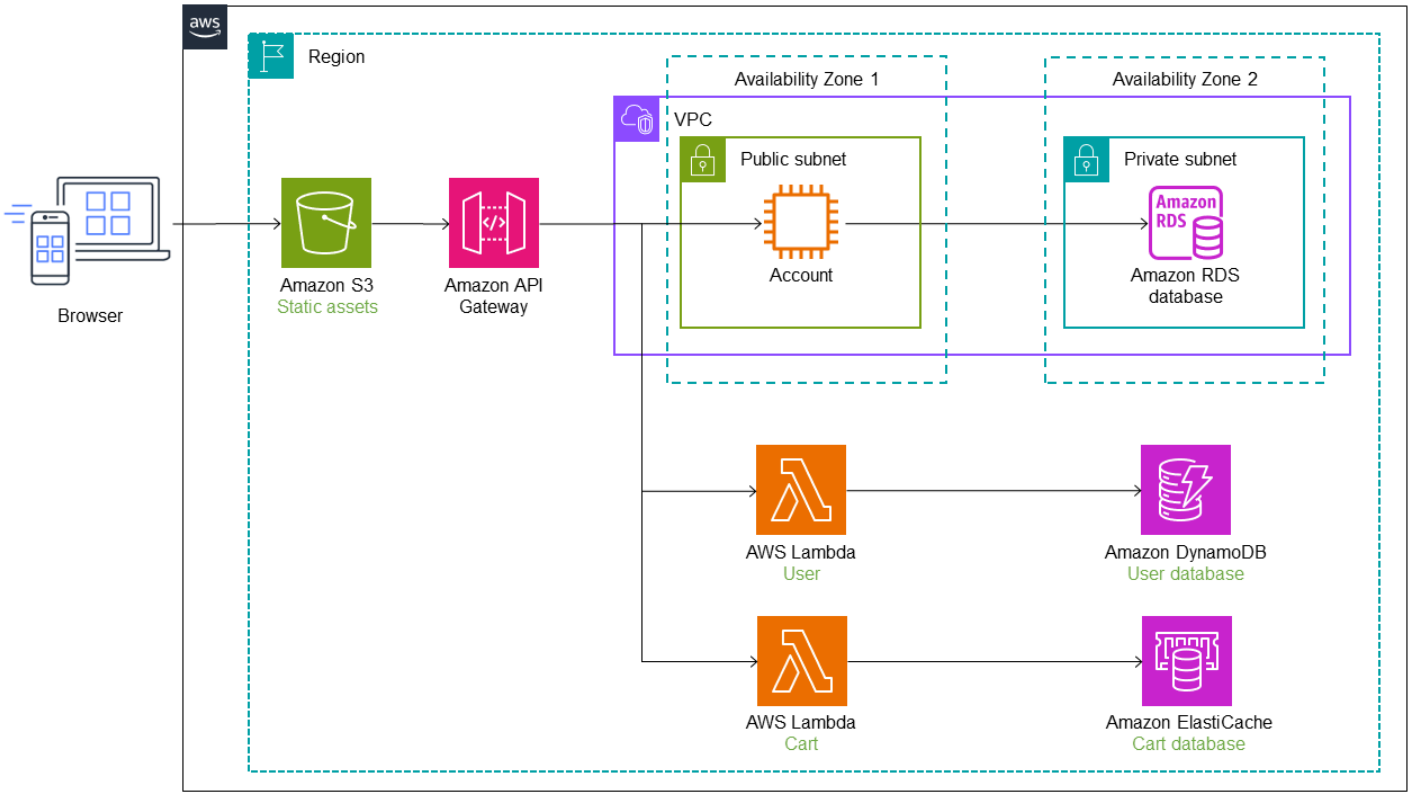
> **Note**
>
> AWS Migration Hub Refactor Spaces は、2025 年 11 月 7 日現在、新規のお客様に公開されていません。に似た機能については AWS Migration Hub Refactor Spaces、「」を参照してくださいAWS Transform。



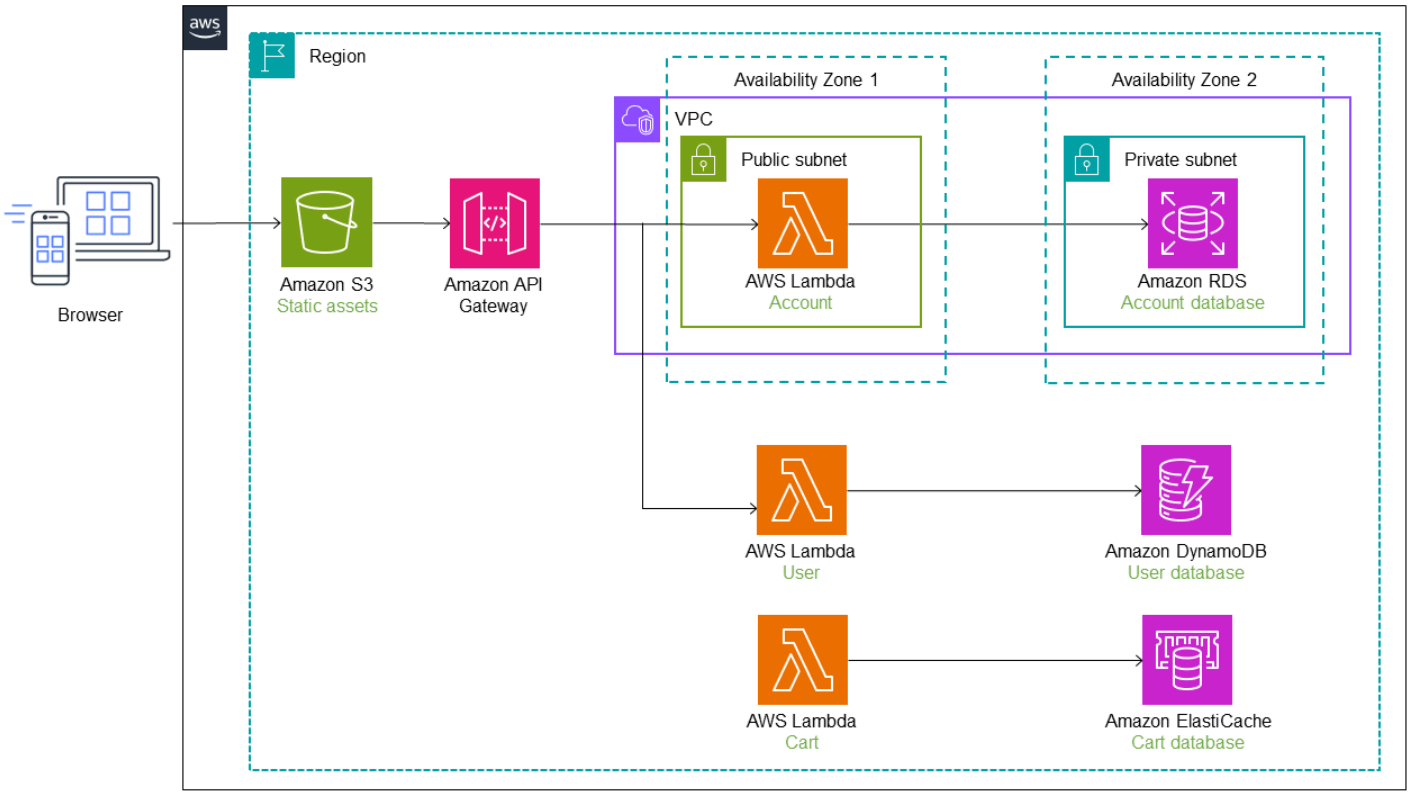
ユーザーサービスを Lambda 関数に移行して、Amazon DynamoDB データベースにそのデータを保存します。また、Lambda サービスエンドポイントとデフォルトルートを Refactor Spaces に追加し、API Gateway を自動設定して、呼び出しが Lambda 関数にルーティングされるようにします。



次の図では、カートサービスも、モノリスから Lambda 関数に移行済みであり、Refactor Spaces にルートとサービスエンドポイントを追加することで、トラフィックを自動的に `Cart` Lambda 関数にカットオーバーしています。Lambda 関数のデータストアは、Amazon ElastiCache によって管理され、モノリシックアプリケーションは、Amazon RDS データベースと共に EC2 インスタンスに残っています。



次の図では、最後のサービス (アカウント) が、モノリスから Lambda 関数に移行済みで、このサービスは、元の Amazon RDS データベースを引き続き使用しています。また、新しいアーキテクチャには、データベースを個別に持つ 3 つのマイクロサービスが追加済みで、各サービスは、異なるタイプのデータベースを使用しています。目的別データベースを使用してマイクロサービスの特定のニーズを満たすというこうした概念は、ポリグロット永続性と呼ばれます。Lambda 関数は、ユースケースによって決定される各種プログラミング言語で実装することもできます。リファクタリング中は、Refactor Spaces によって、Lambda に向かうトラフィックのカットオーバーおよびルーティングを自動化します。これにより、ルーティングインフラストラクチャの設計、デプロイ、設定に必要なビルダーの時間を節約できます。



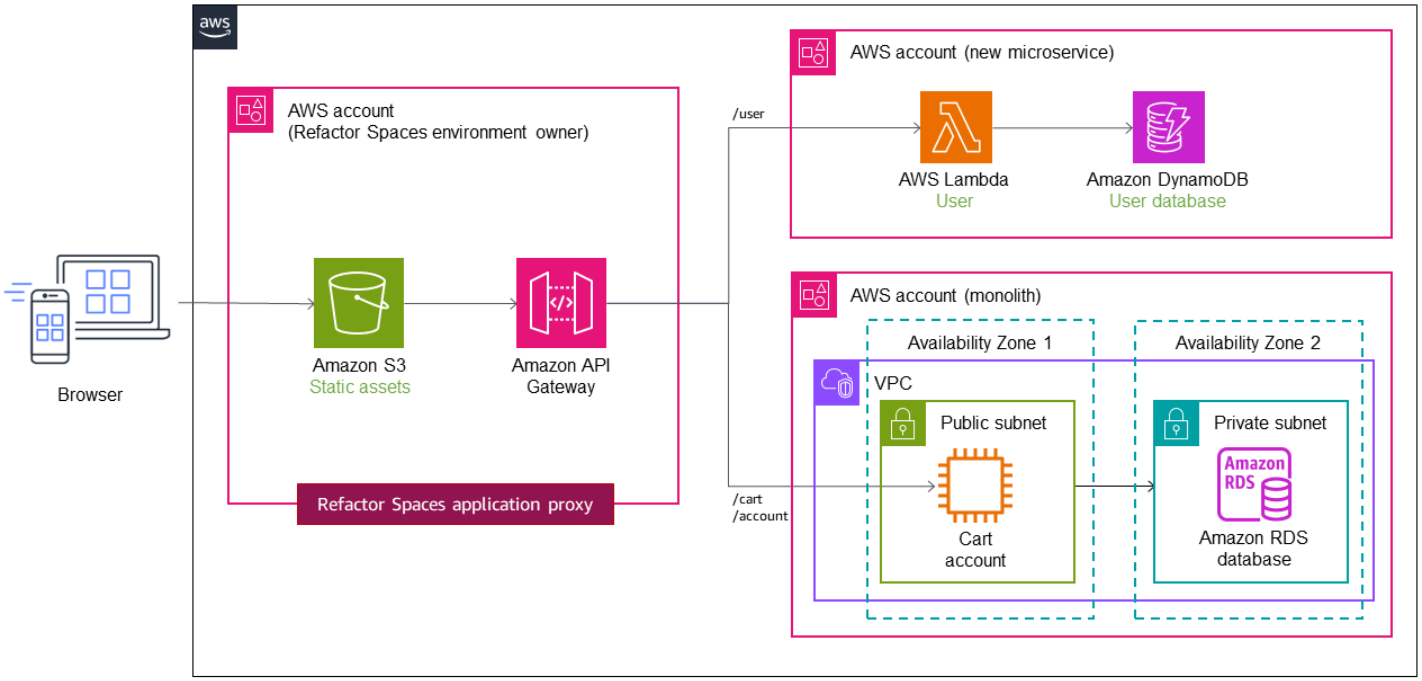
複数アカウントの使用

前の実装では、モノリシックアプリケーションに、プライベートサブネットとパブリックサブネットがある単一の VPC を使用し、単純化のために、同じ AWS アカウント 内にマイクロサービスをデプロイしました。ただし、マイクロサービスがデプロイの独立性 AWS アカウント のために複数の にデプロイされることがよくある実際のシナリオでは、このようなケースはほとんどありません。また、マルチアカウント構造では、設定によって、モノリスから異なるアカウントの新しいサービスに向かうトラフィックをルーティングしなければなりません。

リファクタリングスペースは、モノリシックアプリケーションから API コールをルーティングするための AWS インフラストラクチャの作成と設定に役立ちます。これを使用すると、アプリケーションリソースの一部として、 AWS アカウント内の API Gateway、Network Load Balancer、リソースベース AWS Identity and Access Management (IAM) ポリシーをオーケストレーションできます。単一のアカウント AWS アカウント または複数のアカウント間で新しいサービスを外部 HTTP エンドポイントに透過的に追加できます。これらのリソースはすべて 内でオーケストレーション AWS アカウント され、デプロイ後にカスタマイズして設定できます。

次の図に示すように、ユーザーサービスとカートサービスを 2 つの異なるアカウントにデプロイするとします。その場合、Refactor Spaces を使用すると、サービスエンドポイントとルートを設定するだけで済みます。Refactor Spaces により、API Gateway と Lambda の統合に加え Lambda リ

ソースポリシーの作成が自動化されるため、モノリスからサービスを安全にリファクタリングすることに注力できます。



Refactor Spaces の使用に関するビデオチュートリアルについては、「Refactor Apps Incrementally with AWS Migration Hub Refactor Spaces」を参照してください。

ワークショップ

- イテレーティブアプリモダナイゼーションワークショップ

ブログの参考情報

- AWS Migration Hub Refactor Spaces
- Deep Dive on an AWS Migration Hub Refactor Spaces
- Deployment Pipelines Reference Architecture and Reference Implementations

関連情報

- API ルーティングパターン
- Refactor Spaces のドキュメント

トランザクションアウトボックスパターン

Intent

トランザクションアウトボックスパターンは、単一オペレーションにデータベース書き込みオペレーションとメッセージまたはイベント通知の両方が含まれる場合に分散システムで発生する二重書き込みオペレーションの問題を解決します。二重書き込みオペレーションは、アプリケーションが 2 つの異なるシステムに書き込みを行う場合に発生します。例えば、マイクロサービスがデータをデータベースに保持し、メッセージを送信して他のシステムに通知する必要がある場合などです。これらのオペレーションのいずれかが失敗すると、データが不整合になる可能性があります。

導入する理由

データベースの更新後にマイクロサービスがイベント通知を送信する場合、データ整合性と信頼性を確保するために、これら 2 つのオペレーションはアトミックに実行する必要があります。

- データベースの更新が成功してもイベント通知が失敗した場合、ダウンストリームのサービスはその変更を認識せず、システムは整合性のない状態になる可能性があります。
- データベースの更新が失敗してもイベント通知が送信されると、データが破損し、システムの信頼性に影響する可能性があります。

適用対象

トランザクションアウトボックスパターンは以下の場合に使用します。

- データベースの更新によってイベント通知が開始されるように、イベント駆動型アプリケーションを構築している場合。
- 2 つのサービスが関与するオペレーションを確実にアトミックにしたい場合。
- イベントソーシングパターンを実装したい場合。

問題点と考慮事項

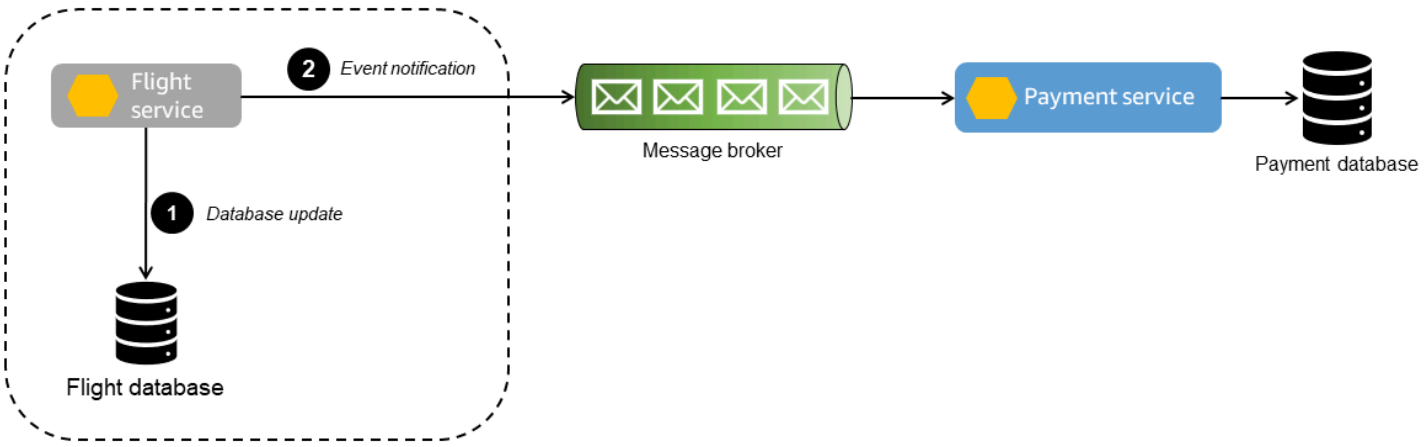
- 重複メッセージ: イベント処理サービスは重複したメッセージやイベントを送信する可能性があるため、処理されたメッセージを追跡して使用側サービスに冪等性があるようにすることをお勧めします。

- 通知の順序: サービスがデータベースを更新するのと同じ順序でメッセージまたはイベントを送信します。これは、イベントストアを使用してデータストアをポイントインタイムで復元できるイベントソーシングパターンにとって重要です。順序が正しくないと、データの品質が低下する可能性があります。通知の順序が保持されないと、結果整合性とデータベースのロールバックが問題をさらに悪化させる可能性があります。
- トランザクションロールバック: トランザクションがロールバックされた場合は、イベント通知を送信しないでください。
- サービスレベルのトランザクション処理: トランザクションがデータストアの更新を必要とするサービスにまたがる場合は、Saga オーケストレーションパターンを使用してデータストア全体のデータの完全性を維持します。

実装

高レベルのアーキテクチャ

次のシーケンス図は、二重書き込みオペレーション中に発生するイベントの順序を示しています。



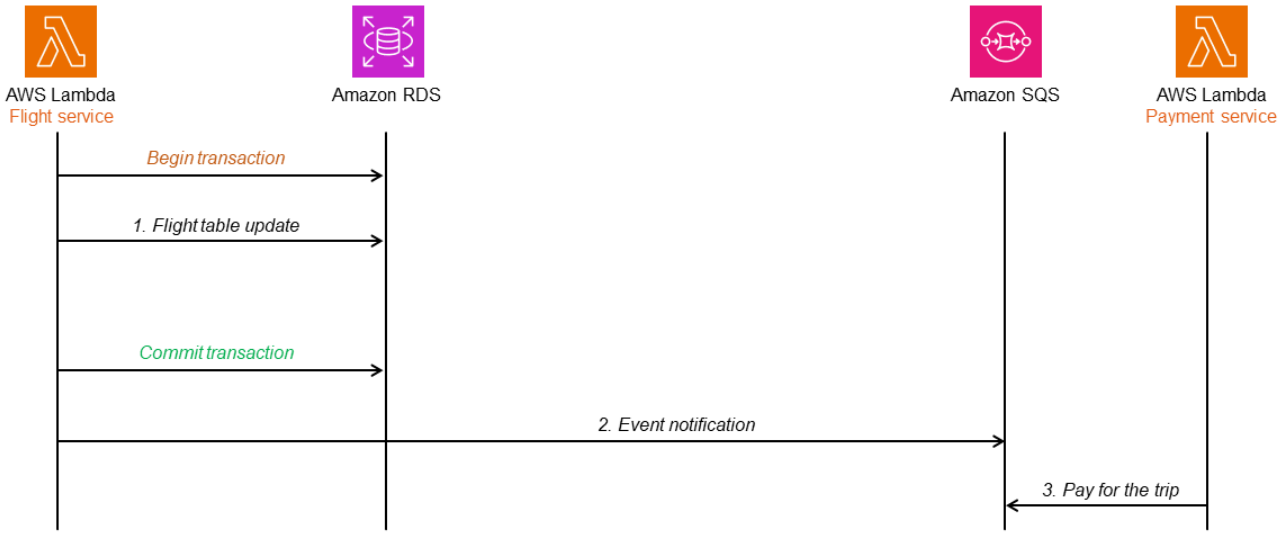
1. フライトサービスはデータベースに書き込み、支払いサービスにイベント通知を送信します。
2. メッセージブローカーはメッセージとイベントを支払いサービスに伝えます。メッセージブローカーに障害が発生すると、支払いサービスは更新を受信できなくなります。

フライトデータベースの更新に失敗しても通知が送信された場合、支払いサービスはイベント通知に基づいて支払いを処理します。これは、ダウンストリームのデータ不整合の原因になります。

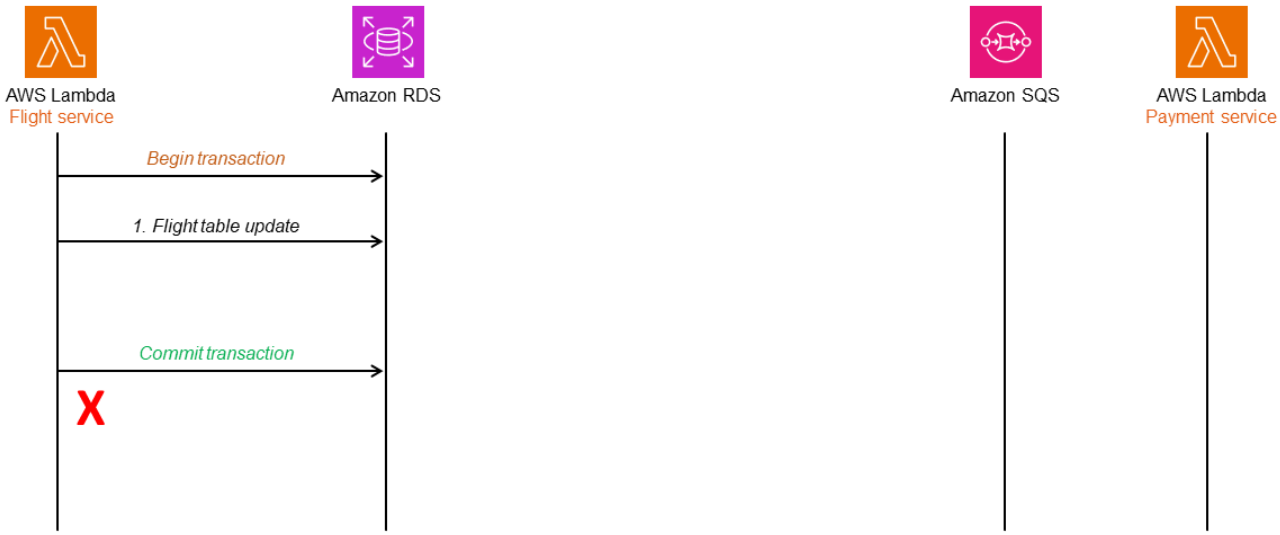
AWS サービスを使用した実装

このシーケンス図のパターンを実証するために、以下の図に示すように、次の AWS のサービスを使用します。

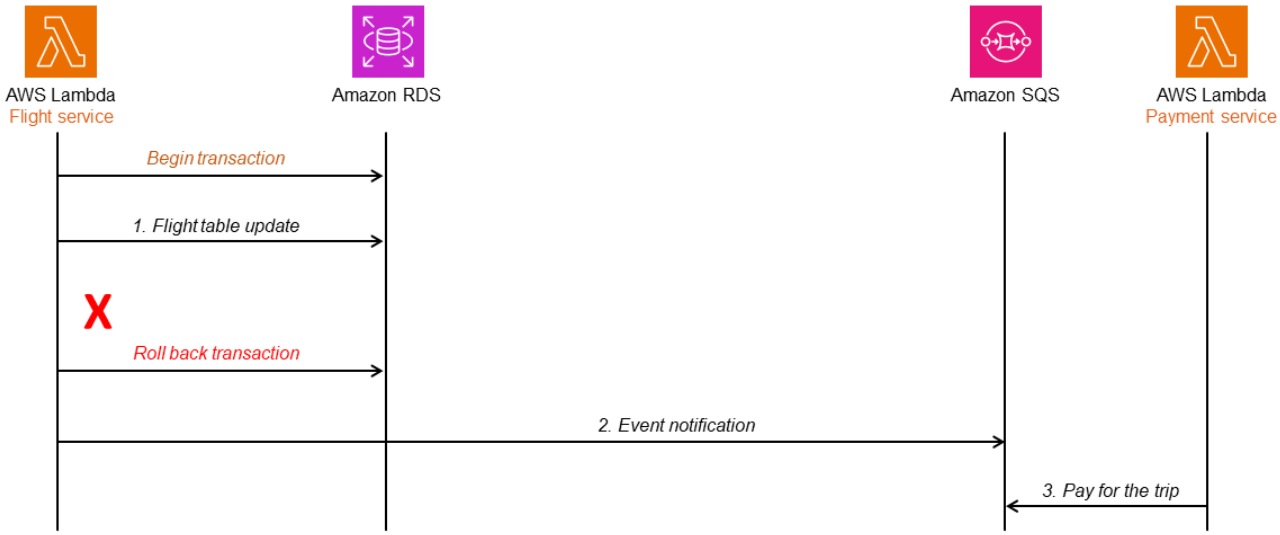
- マイクロサービスは AWS Lambda を使用して実装されます。
- プライマリデータベースは、Amazon Relational Database Service (Amazon RDS) によって管理されます。
- Amazon Simple Queue Service (Amazon SQS) は、イベント通知を受信するメッセージブローカーとして機能します。



トランザクションをコミットした後にフライトサービスに障害が発生すると、イベント通知が送信されない可能性があります。



ただし、トランザクションが失敗してロールバックされたにもかかわらずイベント通知が送信され、支払いサービスが支払いを処理する場合があります。



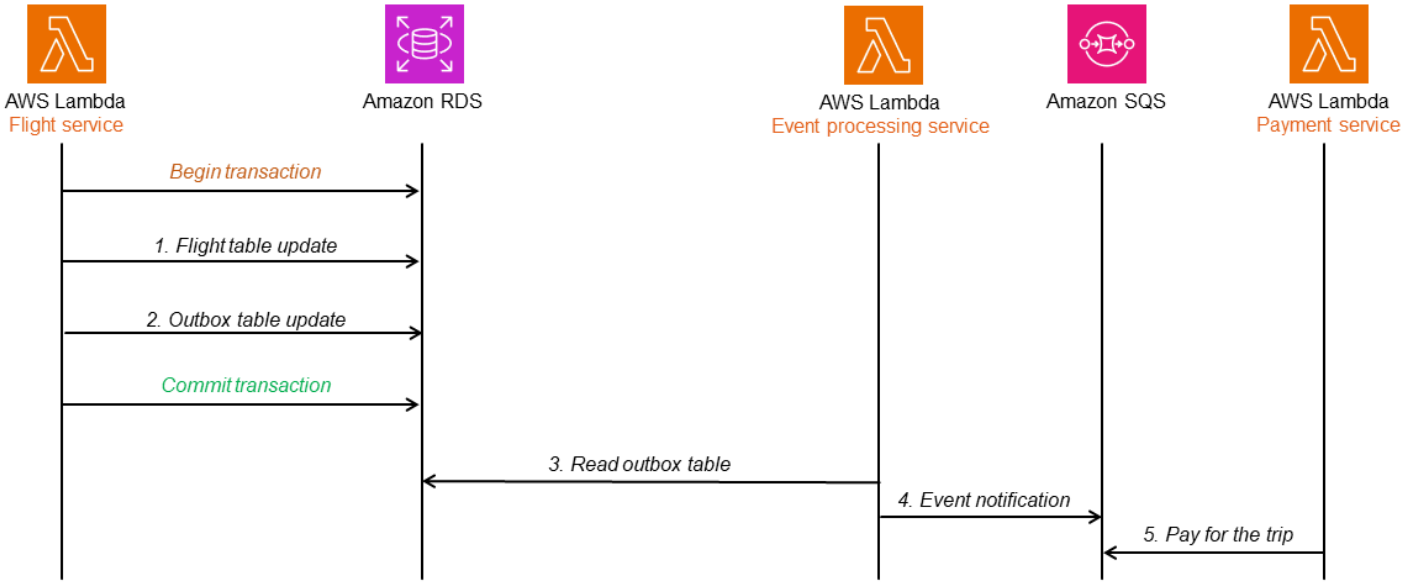
この問題に対処するには、アウトボックステーブルまたは変更データキャプチャ (CDC) を使用します。以下のセクションでは、これら 2 つのオプションと、AWS のサービスを使用してそれらを実装する方法について説明します。

アウトボックステーブルをリレーショナルデータベースで使用する

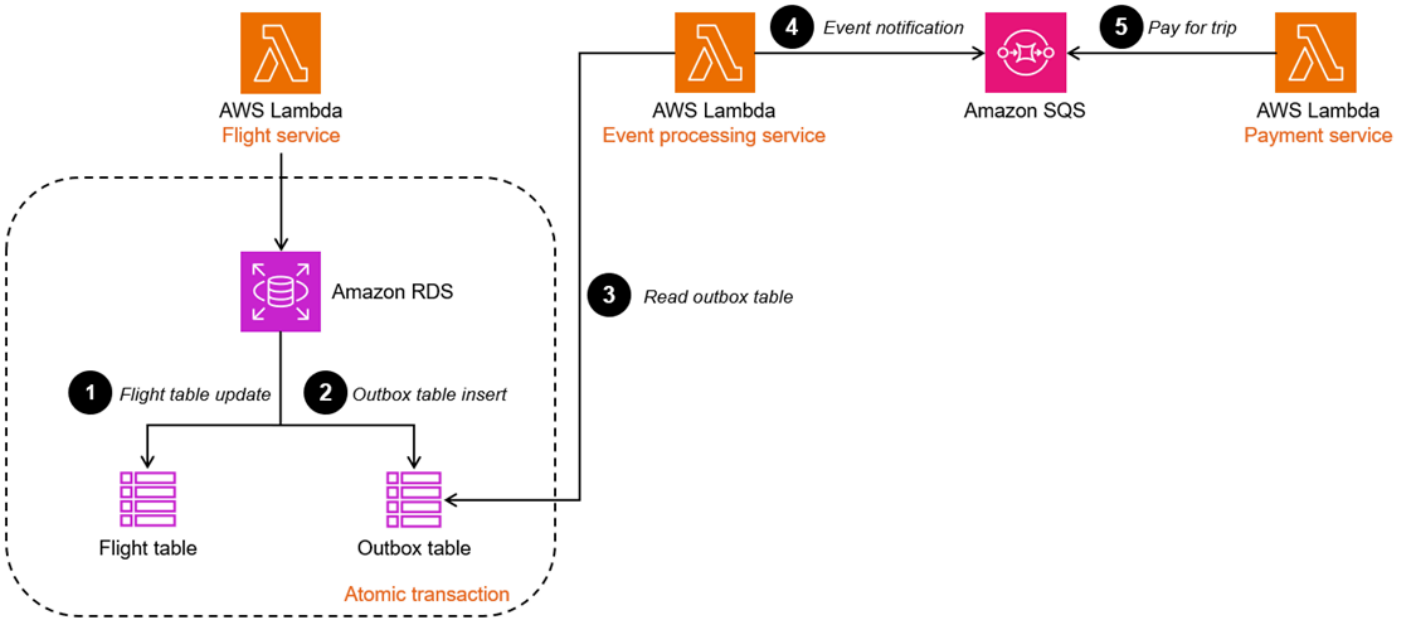
アウトボックステーブルには、フライトサービスからのすべてのイベントがタイムスタンプおよびシーケンス番号とともに保存されます。

フライトテーブルが更新されると、同じトランザクションでアウトボックステーブルも更新されます。別のサービス (イベント処理サービスなど) がアウトボックステーブルから読み取り、Amazon SQS にイベントを送信します。Amazon SQS は、さらに処理するためにイベントに関するメッセージを支払いサービスに送信します。Amazon SQS スタンダードキューは、メッセージが少なくとも 1 回配信され、メッセージが失われないことを保証します。ただし、Amazon SQS スタンダードキューを使用する場合、同じメッセージまたはイベントが複数回配信される可能性があるため、イベント通知サービスの冪等性 (つまり、同じメッセージを複数回処理しても悪影響が無いこと) を確認する必要があります。メッセージの順序付けでメッセージを 1 回だけ処理する必要がある場合は、Amazon SQS の先入れ先出し (FIFO) キューを使用できます。

フライトテーブルの更新またはアウトボックステーブルの更新が失敗した場合、トランザクション全体がロールバックされるため、ダウンストリームのデータに不整合が生じることはありません。



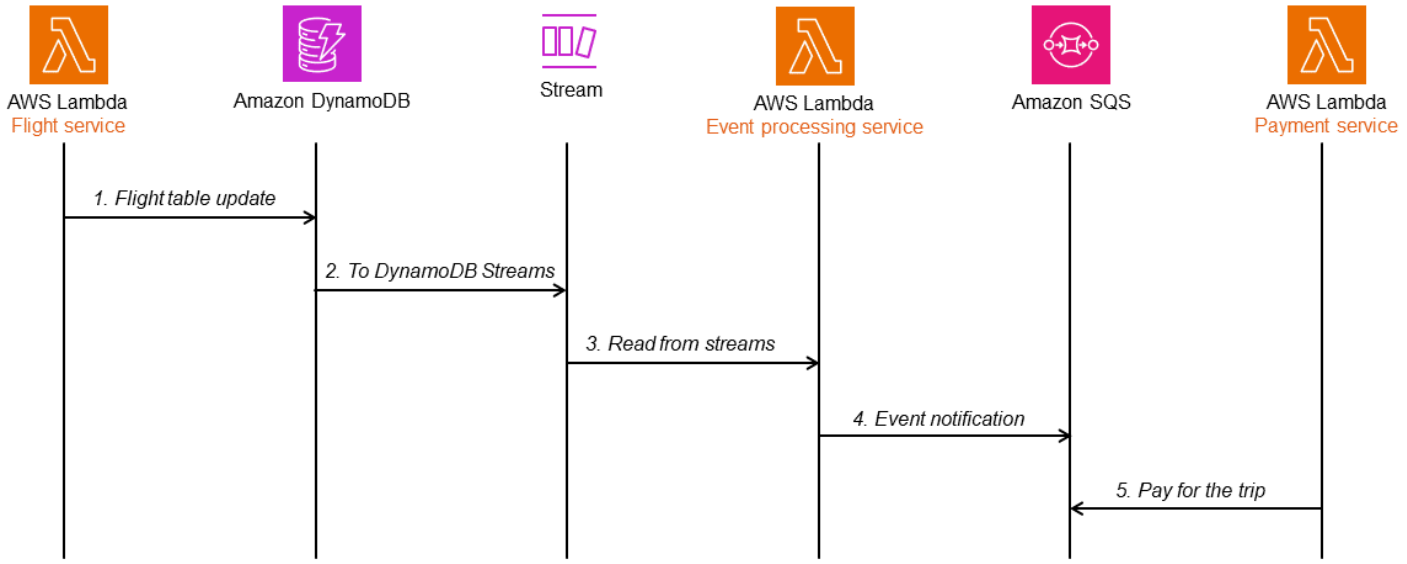
以下の図では、トランザクションアウトボックスアーキテクチャは Amazon RDS データベースを使用して実装されています。イベント処理サービスがアウトボックステーブルを読み取ると、コミットされた (成功した) トランザクションに含まれる行のみを認識し、イベントのメッセージを SQS キューに格納します。SQS キューは支払いサービスによって読み取られ、さらに処理されます。この設計では、タイムスタンプとシーケンス番号を使用して二重書き込みオペレーションの問題が解決され、メッセージとイベントの順序が保持されます。



変更データキャプチャ (CDC) の使用

一部のデータベースでは、変更されたデータをキャプチャするためのアイテムレベルの変更の発行がサポートされています。変更された項目を特定し、それに応じてイベント通知を送信できます。これにより、更新を追跡するテーブルをもう 1 つ作成する手間が省けます。フライトサービスによって開始されたイベントは、同じアイテムの別の属性に保存されます。

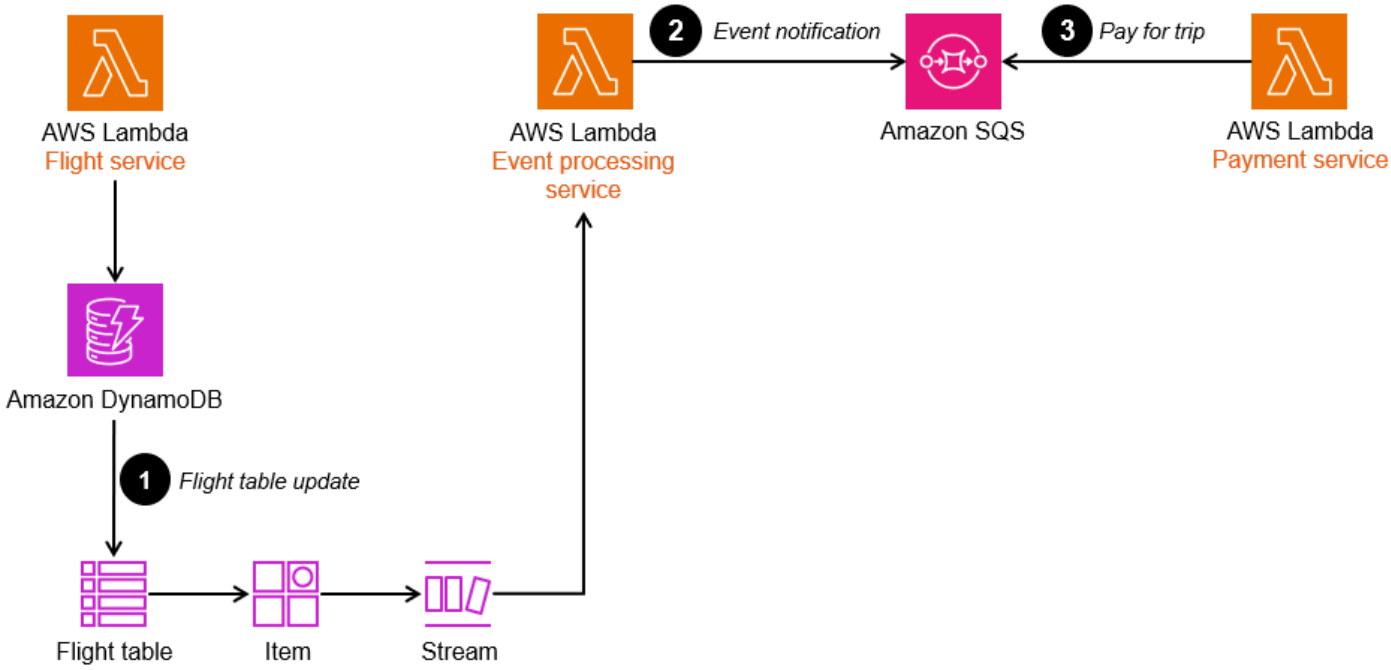
Amazon DynamoDB は CDC 更新をサポートするキー値 NoSQL データベースです。次のシーケンス図では、DynamoDB はアイテムレベルの変更を Amazon DynamoDB Streams に発行しています。イベント処理サービスはストリームから読み取り、さらに処理するためにイベント通知を支払いサービスに発行します。



DynamoDB Streams は、時系列シーケンスを使用して DynamoDB テーブル内のアイテムレベルの変更に関連する情報の流れをキャプチャします。

DynamoDB テーブルでストリームを有効にすることで、トランザクションアウトボックスパターンを実装できます。イベント処理サービスの Lambda 関数は、これらのストリームに関連付けられています。

- フライトテーブルが更新されると、変更されたデータが DynamoDB Streams によってキャプチャされ、イベント処理サービスがストリームをポーリングして新しいレコードを探します。
- 新しいストリームレコードが使用可能になると、Lambda 関数はイベントのメッセージを同期的に SQS キューに配置し、さらに処理します。DynamoDB アイテムに属性を追加して、必要に応じてタイムスタンプとシーケンス番号をキャプチャし、実装の堅牢性を高めることができます。



「サンプルコード」

アウトボックステーブルの使用

このセクションのサンプルコードは、アウトボックステーブルを使用してトランザクションアウトボックスパターンを実装する方法を示しています。コード全体を確認するには、この例の GitHub リポジトリを参照してください。

次のコードスニペットは、データベース内の Flight エンティティと Flight イベントを 1 回のトランザクションでそれぞれのテーブルに保存します。

```
@PostMapping("/flights")
    @Transactional
    public Flight createFlight(@Valid @RequestBody Flight flight) {
        Flight savedFlight = flightRepository.save(flight);
        JsonNode flightPayload = objectMapper.convertValue(flight, JsonNode.class);
        FlightOutbox outboxEvent = new FlightOutbox(flight.getId().toString(),
 FlightOutbox.EventType.FLIGHT_BOOKED,
                flightPayload);
        outboxRepository.save(outboxEvent);
        return savedFlight;
    }
```

別のサービスが定期的にアウトボックステーブルをスキャンして新しいイベントがないか確認し、Amazon SQS に送信し、Amazon SQS が正常に応答した場合はテーブルから削除します。ポーリングレートは `application.properties` ファイルで設定できます。

```
@Scheduled(fixedDelayString = "${sqs.polling_ms}")
    public void forwardEventsToSQS() {
        List<FlightOutbox> entities =
 outboxRepository.findAllByOrderByIdAsc(Pageable.ofSize(batchSize)).toList();
        if (!entities.isEmpty()) {
            GetQueueUrlRequest getQueueRequest = GetQueueUrlRequest.builder()
                    .queueName(sqsQueueName)
                    .build();
            String queueUrl = this.sqsClient.getQueueUrl(getQueueRequest).queueUrl();
            List<SendMessageBatchRequestEntry> messageEntries = new ArrayList<>();
            entities.forEach(entity ->
 messageEntries.add(SendMessageBatchRequestEntry.builder()
                    .id(entity.getId().toString())
                    .messageGroupId(entity.getAggregateId())
                    .messageDeduplicationId(entity.getId().toString())
                    .messageBody(entity.getPayload().toString())
                    .build())
            );
            SendMessageBatchRequest sendMessageBatchRequest =
 SendMessageBatchRequest.builder()
                    .queueUrl(queueUrl)
                    .entries(messageEntries)
                    .build();
            sqsClient.sendMessageBatch(sendMessageBatchRequest);
            outboxRepository.deleteAllInBatch(entities);
        }
    }
```

変更データキャプチャ (CDC) の使用

このセクションのサンプルコードは、DynamoDB の変更データキャプチャ (CDC) 機能を使用してトランザクションアウトボックスパターンを実装する方法を示しています。コード全体を確認するには、この例の GitHub リポジトリを参照してください。

次の AWS Cloud Development Kit (AWS CDK) コードスニペットでは、DynamoDB フライトテーブルと Amazon Kinesis データストリーム (`cdcStream`) を作成し、すべての更新がストリームに送信されるようにフライトテーブルを設定しています。

```
Const cdcStream = new kinesis.Stream(this, 'flightsCDCStream', {
    streamName: 'flightsCDCStream'
})

const flightTable = new dynamodb.Table(this, 'flight', {
    tableName: 'flight',
    kinesisStream: cdcStream,
    partitionKey: {
        name: 'id',
        type: dynamodb.AttributeType.STRING,
    }

});
```

次のコードスニペットと設定では、Spring Cloud Stream 関数を定義します。これによって、Kinesis ストリームの更新を取得し、これらのイベントをさらに処理するために SQS キューに転送しています。

```
applications.properties
spring.cloud.stream.bindings.sendToSQS-in-0.destination=${kinesisstreamname}
spring.cloud.stream.bindings.sendToSQS-in-0.content-type=application/ddb

QueueService.java
@Bean
public Consumer<Flight> sendToSQS() {
    return this::forwardEventsToSQS;
}

public void forwardEventsToSQS(Flight flight) {
    GetQueueUrlRequest getQueueRequest = GetQueueUrlRequest.builder()
            .queueName(sqsQueueName)
            .build();
    String queueUrl = this.sqsClient.getQueueUrl(getQueueRequest).queueUrl();
    try {
        SendMessageRequest send_msg_request = SendMessageRequest.builder()
                .queueUrl(queueUrl)
                .messageBody(objectMapper.writeValueAsString(flight))
                .messageGroupId("1")
                .messageDeduplicationId(flight.getId().toString())
                .build();
        sqsClient.sendMessage(send_msg_request);
    } catch (IOException | AmazonServiceException e) {
```

```
            logger.error("Error sending message to SQS", e);
        }
    }
```

GitHub リポジトリ

このパターンのサンプルアーキテクチャの完全な実装については、https://github.com/aws-samples/transactional-outbox-pattern にある GitHub リポジトリを参照してください。

リソース

リファレンス

- AWS アーキテクチャセンター
- AWS デベロッパーセンター
- Amazon Builders' Library

ツール

- AWS Well-Architected Tool
- AWS App2Container
- AWS Microservice Extractor for .NET

方法論

- The Twelve-Factor App (Adam Wiggins 氏による ePub)
- Michael T. Nygard 氏著「Release It!: Design and Deploy Production-Ready Software」第 2 版、ノースカロライナ州ローリー: Pragmatic Bookshelf、2018 年。
- Polyglot Persistence (Martin Fowler 氏によるブログ記事)
- StranglerFigApplication (Martin Fowler 氏によるブログ記事)

ドキュメント履歴

以下の表は、本ガイドの重要な変更点について説明したものです。今後の更新に関する通知を受け取る場合は、RSS フィード をサブスクライブできます。

変更	説明	日付
新しいパターン	六角形アーキテクチャと散布-収集という 2 つの新しいパターンを追加しました。	2024 年 5 月 7 日
新しいコード例	トランザクションアウトボックスパターンに、変更データキャプチャ (CDC) ユースケースのサンプルコードを追加しました。	2024 年 2 月 23 日
新しいコード例	• トランザクションアウトボックスパターンのサンプルコードを更新しました。 • オーケストレーションパターンとコレオグラフィーパターンに関するセクションを削除しました。これらのパターンは、saga コレオグラフィと saga オーケストレーションに置き換わっています。	2023 年 11 月 16 日
新しいパターン	saga コレオグラフィ、publish-subscribe、イベントソーシングの 3 つの新しいパターンを追加しました。	2023 年 11 月 14 日

を更新する	strangler fig パターンの実装セクションを更新しました。	2023 年 10 月 2 日
初版発行	この最初のリリースには、腐敗防止層 (ACL)、API ルーティング、回路ブレーカー、オーケストレーションとコレオグラフィ、バックオフでの再試行、saga オーケストレーション、strangler fig、およびトランザクション送信トレイの 8 つの設計パターンが含まれています。	2023 年 7 月 28 日

AWS 規範ガイダンスの用語集

以下は、 AWS 規範ガイダンスによって提供される戦略、ガイド、パターンで一般的に使用される用語です。エントリを提案するには、用語集の最後のフィードバックの提供リンクを使用します。

数字

7 Rs

アプリケーションをクラウドに移行するための 7 つの一般的な移行戦略。これらの戦略は、ガートナーが 2011 年に特定した 5 Rs に基づいて構築され、以下で構成されています。

- リファクタリング/アーキテクチャの再設計 — クラウドネイティブ特徴を最大限に活用して、俊敏性、パフォーマンス、スケーラビリティを向上させ、アプリケーションを移動させ、アーキテクチャを変更します。これには、通常、オペレーティングシステムとデータベースの移植が含まれます。例: オンプレミスの Oracle データベースを Amazon Aurora PostgreSQL 互換エディションに移行する。
- リプラットフォーム (リフトアンドリシェイプ) — アプリケーションをクラウドに移行し、クラウド機能を活用するための最適化レベルを導入します。例: お客様のオンプレミスの Oracle データベースを AWS クラウドの Oracle 用の Amazon Relational Database Service (Amazon RDS) に移行する。
- 再購入 (ドロップアンドショップ) — 通常、従来のライセンスから SaaS モデルに移行して、別の製品に切り替えます。例: 顧客関係管理 (CRM) システムを Salesforce.com に移行する。
- リホスト (リフトアンドシフト) — クラウド機能を活用するための変更を加えずに、アプリケーションをクラウドに移行します。例: お客様のオンプレミスの Oracle データベースを AWS クラウドの EC2 インスタンス上の Oracle に移行する。
- 再配置 (ハイパーバイザーレベルのリフトアンドシフト) – 新しいハードウェアを購入したり、アプリケーションを書き換えたり、既存の運用を変更したりすることなく、インフラストラクチャをクラウドに移行できます。オンプレミスプラットフォームから同じプラットフォームのクラウドサービスにサーバーを移行します。例： Microsoft Hyper-Vアプリケーションを に移行します AWS。
- 保持 (再アクセス) — アプリケーションをお客様のソース環境で保持します。これには、主要なリファクタリングを必要とするアプリケーションや、お客様がその作業を後日まで延期したいアプリケーション、およびそれらを移行するためのビジネス上の正当性がないため、お客様が保持するレガシーアプリケーションなどがあります。
- 廃止 — お客様のソース環境で不要になったアプリケーションを停止または削除します。

A

ABAC

「属性ベースのアクセス制御」をご覧ください。

抽象化されたサービス

「マネージドユーザー」をご覧ください。

ACID

「原子性、一貫性、分離性、耐久性 (ACID)」をご覧ください。

アクティブ/アクティブ移行

(双方向レプリケーションツールまたは二重書き込み操作を使用して) ソースデータベースとターゲットデータベースを同期させ、移行中に両方のデータベースが接続アプリケーションからのトランザクションを処理するデータベース移行方法。この方法では、1 回限りのカットオーバーの必要がなく、管理された小規模なバッチで移行できます。アクティブ/パッシブ移行よりも柔軟な方法ですが、さらに多くの作業が必要となります。

アクティブ/パッシブ移行

ソースデータベースとターゲットデータベースを同期させながら、データがターゲットデータベースにレプリケートされている間、接続しているアプリケーションからのトランザクションをソースデータベースのみで処理するデータベース移行方法。移行中、ターゲットデータベースはトランザクションを受け付けません。

集計関数

複数行に処理を行い、グループ全体を対象に単一の戻り値を計算する SQL 関数。集計関数の例としては、SUM や MAX などがあります。

AI

「人工知能」をご覧ください。

AIOps

「AI オペレーション」をご覧ください。

匿名化

データセット内の個人情報を完全に削除するプロセス。匿名化は個人のプライバシー保護に役立ちます。匿名化されたデータは、もはや個人データとは見なされません。

アンチパターン

繰り返し起こる問題に対して頻繁に用いられる解決策で、その解決策が逆効果であったり、効果がなかったり、代替案よりも効果が低かったりするもの。

アプリケーション制御

マルウェアからシステムを保護するために、承認されたアプリケーションのみを使用できるようにするセキュリティアプローチ。

アプリケーションポートフォリオ

アプリケーションの構築と維持にかかるコスト、およびそのビジネス価値を含む、組織が使用する各アプリケーションに関する詳細情報の集まり。この情報は、ポートフォリオの検出と分析プロセス の重要な要素であり、移行、モダナイズ、最適化するアプリケーションを特定し、優先順位を付けるのに役立ちます。

人工知能 (AI)

コンピューティングテクノロジーを使用し、学習、問題の解決、パターンの認識など、通常は人間に関連づけられる認知機能の実行に特化したコンピュータサイエンスの分野。詳細については、「人工知能 (AI) とは何ですか?」をご覧ください。

AI オペレーション (AIOps)

機械学習技術を使用して運用上の問題を解決し、運用上のインシデントと人の介入を減らし、サービス品質を向上させるプロセス。 AWS 移行戦略での AIOps の使用方法については、オペレーション統合ガイド を参照してください。

非対称暗号化

暗号化用のパブリックキーと復号用のプライベートキーから成る 1 組のキーを使用した、暗号化のアルゴリズム。パブリックキーは復号には使用されないため共有しても問題ありませんが、プライベートキーの利用は厳しく制限する必要があります。

原子性、一貫性、分離性、耐久性 (ACID)

エラー、停電、その他の問題が発生した場合でも、データベースのデータ有効性と運用上の信頼性を保証する一連のソフトウェアプロパティ。

属性ベースのアクセス制御 (ABAC)

部署、役職、チーム名など、ユーザーの属性に基づいてアクセス許可をきめ細かく設定する方法。詳細については、 AWS Identity and Access Management (IAM) ドキュメントの「 の ABAC AWS」を参照してください。

信頼できるデータソース

最も信頼性のある情報源とされるデータのプライマリーバージョンを保存する場所。匿名化、編集、仮名化など、データを処理または変更する目的で、信頼できるデータソースから他の場所にデータをコピーすることができます。

アベイラビリティーゾーン (AZ)

他のアベイラビリティーゾーンの障害から AWS リージョン 隔離され、同じリージョン内の他のアベイラビリティーゾーンへの低コストで低レイテンシーのネットワーク接続を提供する 内の別の場所。

AWS クラウド導入フレームワーク (AWS CAF)

組織がクラウドへの移行を成功させるための効率的で効果的な計画を立て AWS るための、 のガイドラインとベストプラクティスのフレームワークです。 AWS CAF は、ビジネス、人材、ガバナンス、プラットフォーム、セキュリティ、運用という 6 つの重点分野にガイダンスを整理しています。ビジネス、人材、ガバナンスの観点では、ビジネススキルとプロセスに重点を置き、プラットフォーム、セキュリティ、オペレーションの視点は技術的なスキルとプロセスに焦点を当てています。例えば、人材の観点では、人事 (HR)、人材派遣機能、および人材管理を扱うステークホルダーを対象としています。この観点から、 AWS CAF は人材開発、トレーニング、コミュニケーションに関するガイダンスを提供し、組織がクラウド導入を成功させるための準備を支援します。詳細については、AWS CAF ウェブサイトと AWS CAF のホワイトペーパー を参照してください。

AWS ワークロード認定フレームワーク (AWS WQF)

データベース移行ワークロードを評価し、移行戦略を推奨し、作業見積もりを提供するツール。AWS WQF は AWS Schema Conversion Tool (AWS SCT) に含まれています。データベーススキーマとコードオブジェクト、アプリケーションコード、依存関係、およびパフォーマンス特性を分析し、評価レポートを提供します。

B

不正なボット

個人や組織に混乱や損害を与えることを目的としたボット。

BCP

「ビジネス継続性計画 (BCP)」をご覧ください。

動作グラフ

リソースの動作とインタラクションを経時的に示した、一元的なインタラクティブビュー。Amazon Detective の動作グラフを使用すると、失敗したログオンの試行、不審な API 呼び出し、その他同様のアクションを調べることができます。詳細については、Detective ドキュメントの「動作グラフのデータ」を参照してください。

ビッグエンディアンシステム

最上位バイトを最初に格納するシステム。「エンディアン性」もご覧ください。

二項分類

バイナリ結果 (2 つの可能なクラスのうちの 1 つ) を予測するプロセス。例えば、お客様の機械学習モデルで「この E メールはスパムですか、それともスパムではありませんか」などの問題を予測する必要があるかもしれません。または「この製品は書籍ですか、車ですか」などの問題を予測する必要があるかもしれません。

ブルームフィルター

要素がセットのメンバーであるかどうかをテストするために使用される、確率的でメモリ効率の高いデータ構造。

ブルー/グリーンデプロイ

それぞれが独立しているが、同一の環境を 2 つ作成するデプロイ戦略。現在のアプリケーションバージョンを 1 つの環境 (ブルー) で実行し、新しいアプリケーションバージョンを別の環境 (グリーン) で実行します。この戦略は、最小限の影響で迅速にロールバックするのに役立ちます。

ボット

インターネット経由で自動タスクを実行し、人間のアクティビティややり取りをシミュレートするソフトウェアアプリケーション。インターネット上の情報のインデックスを作成するウェブクローラーなど、一部のボットは有用または有益です。悪質なボットと呼ばれる他のボットの中には、個人や組織を混乱させたり、損害を与えたりすることを意図したものもあります。

ボットネット

マルウェアに感染しており、ボットハーダーまたはボットオペレーターと呼ばれる単一の当事者によって制御されているボットのネットワーク。ボットネットは、ボットとその影響力を拡大する仕組みとして、非常によく知られています。

ブランチ

コードリポジトリに含まれる領域。リポジトリに最初に作成するブランチは、メインブランチといいます。既存のブランチから新しいブランチを作成し、その新しいブランチで機能を開発した

り、バグを修正したりできます。機能を構築するために作成するブランチは、通常、機能ブランチと呼ばれます。機能をリリースする準備ができたら、機能ブランチをメインブランチに統合します。詳細については、「ブランチの概要」(GitHub ドキュメント) を参照してください。

ブレークグラスアクセス

例外的な状況では、承認されたプロセスを通じて、ユーザーが AWS アカウント 通常アクセス許可を持たない にすばやくアクセスできるようにします。詳細については、 AWS Well-Architected ガイダンスの「ブレークグラス手順の実装」インジケータを参照してください。

ブラウンフィールド戦略

環境の既存インフラストラクチャ。システムアーキテクチャにブラウンフィールド戦略を導入する場合、現在のシステムとインフラストラクチャの制約に基づいてアーキテクチャを設計します。既存のインフラストラクチャを拡張している場合は、ブラウンフィールド戦略とグリーンフィールド戦略を融合させることもできます。

バッファキャッシュ

アクセス頻度が最も高いデータが保存されるメモリ領域。

ビジネス能力

価値を生み出すためにビジネスが行うこと (営業、カスタマーサービス、マーケティングなど)。マイクロサービスのアーキテクチャと開発の決定は、ビジネス能力によって推進できます。詳細については、AWSでのコンテナ化されたマイクロサービスの実行ホワイトペーパーの「ビジネス機能を中心に組織化」セクションを参照してください。

ビジネス継続性計画 (BCP)

大規模移行など、中断を伴うイベントが運用に与える潜在的な影響に対処し、ビジネスを迅速に再開できるようにする計画。

C

CAF

「AWS クラウド導入フレームワーク」を参照してください

カナリアデプロイ

エンドユーザーへのバージョンリリースを、時間をかけて段階的に行うこと。確信が持てたら新規バージョンをデプロイして、現在のバージョン全体を置き換えます。

CCoE

「Cloud Center of Excellence」を参照してください。

CDC

「変更データキャプチャ」を参照してください。

変更データキャプチャ (CDC)

データソース (データベーステーブルなど) の変更を追跡し、その変更に関するメタデータを記録するプロセス。CDC は、ターゲットシステムでの変更を監査またはレプリケートして同期を維持するなど、さまざまな目的に使用できます。

カオスエンジニアリング

障害や破壊的なイベントを意図的に導入して、システムの耐障害性をテストすること。AWS Fault Injection Service (AWS FIS) を使用して、 AWS ワークロードにストレスを与え、その応答を評価する実験を実行できます。

CI/CD

「継続的インテグレーションと継続的デリバリー」を参照してください。

分類

予測を生成するのに役立つ分類プロセス。分類問題の機械学習モデルは、離散値を予測します。離散値は、常に互いに区別されます。例えば、モデルがイメージ内に車があるかどうかを評価する必要がある場合があります。

クライアント側の暗号化

ターゲットがデータ AWS のサービス を受信する前のローカルでのデータの暗号化。

Cloud Center of Excellence (CCoE)

クラウドのベストプラクティスの作成、リソースの移動、移行のタイムラインの確立、大規模変革を通じて組織をリードするなど、組織全体のクラウド導入の取り組みを推進する学際的なチーム。詳細については、 AWS クラウド エンタープライズ戦略ブログの CCoE 投稿を参照してください。

クラウドコンピューティング

リモートデータストレージと IoT デバイス管理に通常使用されるクラウドテクノロジー。クラウドコンピューティングは、一般的に、エッジコンピューティングに接続されています。

クラウド運用モデル

IT 組織において、1 つ以上のクラウド環境を構築、成熟、最適化するために使用される運用モデル。詳細については、「クラウド運用モデルの構築」を参照してください。

導入のクラウドステージ

組織が、 AWS クラウドへの移行時に通常実行する 4 つの段階。

- プロジェクト — 概念実証と学習を目的として、クラウド関連のプロジェクトをいくつか実行する
- 基礎固め — お客様のクラウドの導入を拡大するための基礎的な投資 (ランディングゾーンの作成、CCoE の定義、運用モデルの確立など)
- 移行 — 個々のアプリケーションの移行
- 再発明 — 製品とサービスの最適化、クラウドでのイノベーション

これらのステージは、 AWS クラウド エンタープライズ戦略ブログのブログ記事「クラウドファーストへのジャーニー」と「導入のステージ」で Stephen Orban によって定義されました。AWS 移行戦略との関連性については、「移行準備ガイド」を参照してください。

CMDB

「構成管理データベース (CMDB)」を参照してください。

コードリポジトリ

ソースコードやその他の資産 (ドキュメント、サンプル、スクリプトなど) が保存され、バージョン管理プロセスを通じて更新される場所。一般的なクラウドリポジトリには、GitHub や Bitbucket Cloud があります。コードの各バージョンはブランチと呼ばれます。マイクロサービスの構造では、各リポジトリは 1 つの機能専用です。1 つの CI/CD パイプラインで複数のリポジトリを使用できます。

コールドキャッシュ

空である、または、かなり空きがある、もしくは、古いデータや無関係なデータが含まれているバッファキャッシュ。データベースインスタンスはメインメモリまたはディスクから読み取る必要があり、バッファキャッシュから読み取るよりも時間がかかるため、パフォーマンスに影響します。

コールドデータ

めったにアクセスされず、通常は過去のデータです。この種類のデータをクエリする場合、通常は低速なクエリでも問題ありません。このデータを低パフォーマンスで安価なストレージ階層またはクラスに移動すると、コストを削減することができます。

コンピュータビジョン (CV)

機械学習を使用してデジタルイメージやビデオといった、ビジュアル形式の情報を分析および抽出する AI の分野。例えば、Amazon SageMaker AI では、CV 用の画像処理アルゴリズムを利用できます。

設定ドリフト

ワークロードにおいて、設定が想定した状態から変化すること。これによって、ワークロードが非準拠になる可能性がありますが、この状態は、徐々に生じ、意図的なものではありません。

構成管理データベース (CMDB)

データベースとその IT 環境 (ハードウェアとソフトウェアの両方のコンポーネントとその設定を含む) に関する情報を保存、管理するリポジトリ。通常、CMDB のデータは、移行のポートフォリオの検出と分析の段階で使用します。

コンフォーマンスパック

コンプライアンスチェックとセキュリティチェックをカスタマイズするためにアセンブルできる AWS Config ルールと修復アクションのコレクション。YAML テンプレートを使用して、コンフォーマンスパックを AWS アカウント および リージョンの単一のエンティティとしてデプロイすることも、組織全体にデプロイすることもできます。詳細については、 AWS Config ドキュメントの「コンフォーマンスパック」を参照してください。

継続的インテグレーションと継続的デリバリー (CI/CD)

ソフトウェアリリースプロセスのソース、ビルド、テスト、ステージング、本番の各ステージを自動化するプロセス。CI/CD は一般的にパイプラインと呼ばれます。プロセスの自動化、生産性の向上、コード品質の向上、配信の加速化を可能にします。詳細については、「継続的デリバリーの利点」を参照してください。CD は継続的デプロイ (Continuous Deployment) の略語でもあります。詳細については「継続的デリバリーと継続的なデプロイ」を参照してください。

CV

「コンピュータビジョン」を参照してください。

D

保管中のデータ

ストレージ内にあるデータなど、常に自社のネットワーク内にあるデータ。

データ分類

ネットワーク内のデータを重要度と機密性に基づいて識別、分類するプロセス。データに適した保護および保持のコントロールを判断する際に役立つため、あらゆるサイバーセキュリティのリスク管理戦略において重要な要素です。データ分類は、 AWS Well-Architected フレームワークのセキュリティの柱のコンポーネントです。詳細については、「データ分類」を参照してください。

データドリフト

実稼働データと ML モデルのトレーニングに使用されたデータとの間に有意な差異が生じたり、入力データが時間の経過と共に有意に変化したりすることです。データドリフトは、ML モデル予測の全体的な品質、精度、公平性を低下させる可能性があります。

転送中のデータ

ネットワーク内 (ネットワークリソース間など) を活発に移動するデータ。

データメッシュ

非一元的で分散型のデータ所有権を持つとともに、一元的な管理およびガバナンスを行えるアーキテクチャフレームワーク。

データ最小化

厳密に必要なデータのみを収集し、処理するという原則。でデータ最小化を実践 AWS クラウドすることで、プライバシーリスク、コスト、分析のカーボンフットプリントを削減できます。

データ境界

AWS 環境内の一連の予防ガードレール。信頼できる ID のみが、期待されるネットワークから信頼できるリソースにアクセスできるようにします。詳細については、「 でのデータ境界の構築 AWS」を参照してください。

データの前処理

raw データをお客様の機械学習モデルで簡単に解析できる形式に変換すること。データの前処理とは、特定の列または行を削除して、欠落している、矛盾している、または重複する値に対処することを意味します。

データ出所

データの生成、送信、保存の方法など、データのライフサイクル全体を通じてデータの出所と履歴を追跡するプロセス。

データ件名

データを収集、処理している個人。

データウェアハウス

分析などのビジネスインテリジェンスをサポートするデータ管理システム。データウェアハウスには、一般的に、大量の履歴データが含まれており、多くの場合、それらはクエリや分析に使用されます。

データベース定義言語 (DDL)

データベース内のテーブルやオブジェクトの構造を作成または変更するためのステートメントまたはコマンド。

データベース操作言語 (DML)

データベース内の情報を変更 (挿入、更新、削除) するためのステートメントまたはコマンド。

DDL

「データベース定義言語」を参照してください。

ディープアンサンブル

予測のために複数の深層学習モデルを組み合わせます。ディープアンサンブルを使用して、より正確な予測を取得したり、予測の不確実性を推定したりできます。

深層学習

人工ニューラルネットワークの複数層を使用して、入力データと対象のターゲット変数の間のマッピングを識別する機械学習サブフィールド。

多層防御

一連のセキュリティメカニズムとコントロールをコンピュータネットワーク全体に層状に重ねて、ネットワークとその内部にあるデータの機密性、整合性、可用性を保護する情報セキュリティの手法。この戦略を採用するときは AWS、リソースの保護に役立つように、 AWS Organizations 構造の異なるレイヤーに複数のコントロールを追加します。たとえば、多層防御アプローチでは、多要素認証、ネットワークセグメンテーション、暗号化を組み合わせることができます。

委任管理者

では AWS Organizations、互換性のあるサービスが AWS メンバーアカウントを登録して組織のアカウントを管理し、そのサービスのアクセス許可を管理できます。このアカウントを、そのサービスの委任管理者と呼びます。詳細、および互換性のあるサービスの一覧は、 AWS

Organizations ドキュメントの「AWS Organizationsで使用できるサービス」を参照してください。

トラブルシューティング

アプリケーション、新機能、コードの修正をターゲットの環境で利用できるようにするプロセス。デプロイでは、コードベースに変更を施した後、アプリケーションの環境でそのコードベースを構築して実行します。

開発環境

「環境」を参照してください。

検出管理

イベントが発生したときに、検出、ログ記録、警告を行うように設計されたセキュリティコントロール。これらのコントロールは副次的な防衛手段であり、実行中の予防的コントロールをすり抜けたセキュリティイベントをユーザーに警告します。詳細については、「 AWSでのセキュリティコントロールの実装」の「検出的コントロール」を参照してください。

開発バリューストリームマッピング (DVSM)

ソフトウェア開発ライフサイクルのスピードと品質に悪影響を及ぼす制約を特定し、優先順位を付けるために使用されるプロセス。DVSM は、もともとリーンマニュファクチャリング・プラクティスのために設計されたバリューストリームマッピング・プロセスを拡張したものです。ソフトウェア開発プロセスを通じて価値を創造し、動かすために必要なステップとチームに焦点を当てています。

デジタルツイン

建物、工場、産業機器、生産ラインなど、現実世界のシステムを仮想的に表現したものです。デジタルツインは、予知保全、リモートモニタリング、生産最適化をサポートします。

ディメンションテーブル

スタースキーマにおいて、ファクトテーブルの定量データに関するデータ属性が含まれる小さいテーブル。ディメンションテーブルの属性は、通常、テキストフィールド、またはテキストのように扱える個別の数値で示されます。これらの属性は、一般的に、クエリの制約、フィルタリング、結果セットのラベル付けに使用されます。

ディザスタ

ワークロードまたはシステムが、導入されている主要な場所でのビジネス目標の達成を妨げるイベント。これらのイベントは、自然災害、技術的障害、または意図しない設定ミスやマルウェア攻撃などの人間の行動の結果である場合があります。

ディザスタリカバリ (DR)

ディザスタによるダウンタイムとデータ損失を最小限に抑えるための戦略とプロセス。詳細については、 AWS Well-Architected フレームワークの「 でのワークロードのディザスタリカバリ AWS: クラウドでのリカバリ」を参照してください。

DML

「データベース操作言語」を参照してください。

ドメイン駆動型設計

各コンポーネントが提供している変化を続けるドメイン、またはコアビジネス目標にコンポーネントを接続して、複雑なソフトウェアシステムを開発するアプローチ。この概念は、エリック・エヴァンスの著書、Domain-Driven Design: Tackling Complexity in the Heart of Software (ドメイン駆動設計:ソフトウェアの中心における複雑さへの取り組み) で紹介されています (ボストン: Addison-Wesley Professional、2003)。strangler fig パターンでドメイン駆動型設計を使用する方法の詳細については、「コンテナと Amazon API Gateway を使用して、従来の Microsoft ASP.NET (ASMX) ウェブサービスを段階的にモダナイズ」を参照してください。

DR

「ディザスタリカバリ」を参照してください。

ドリフト検出

ベースライン設定からの偏差を追跡します。たとえば、 AWS CloudFormation を使用してシステムリソースのドリフトを検出したり、 を使用して AWS Control Tower 、ガバナンス要件のコンプライアンスに影響を与える可能性のあるランディングゾーンの変更を検出したりできます。

DVSM

「開発バリューストリームマッピング」を参照してください。

E

EDA

「探索的データ分析」を参照してください。

EDI

「電子データ交換」を参照してください。

エッジコンピューティング

IoT ネットワークのエッジにあるスマートデバイスの計算能力を高めるテクノロジー。クラウドコンピューティングと比較すると、エッジコンピューティングは通信レイテンシーを短縮し、応答時間を改善できます。

電子データ交換 (EDI)

組織間で行う、ビジネスドキュメントの自動交換。詳細については、「電子データ交換とは」を参照してください。

暗号化

人間が読み取り可能なプレーンテキストデータを暗号文に変換するコンピューティング処理。

暗号化キー

暗号化アルゴリズムが生成した、ランダム化されたビットからなる暗号文字列。キーの長さは決まっておらず、各キーは予測できないように、一意になるように設計されています。

エンディアン

コンピュータメモリにバイトが格納される順序。ビッグエンディアンシステムでは、最上位バイトが最初に格納されます。リトルエンディアンシステムでは、最下位バイトが最初に格納されます。

エンドポイント

「サービスエンドポイント」を参照してください。

エンドポイントサービス

仮想プライベートクラウド (VPC) 内でホストして、他のユーザーと共有できるサービス。を使用してエンドポイントサービスを作成し AWS PrivateLink 、他の AWS アカウント または AWS Identity and Access Management (IAM) プリンシパルにアクセス許可を付与できます。これらのアカウントまたはプリンシパルは、インターフェイス VPC エンドポイントを作成することで、エンドポイントサービスにプライベートに接続できます。詳細については、Amazon Virtual Private Cloud (Amazon VPC) ドキュメントの「エンドポイントサービスを作成する」を参照してください。

エンタープライズリソースプランニング (ERP)

エンタープライズの主要なビジネスプロセス (会計、MES、プロジェクト管理など) を自動化および管理するシステム。

エンベロープ暗号化

暗号化キーを、別の暗号化キーを使用して暗号化するプロセス。詳細については、 AWS Key Management Service (AWS KMS) ドキュメントの「エンベロープ暗号化」を参照してください。

環境

実行中のアプリケーションのインスタンス。クラウドコンピューティングにおける一般的な環境の種類は以下のとおりです。

- 開発環境 — アプリケーションのメンテナンスを担当するコアチームのみが使用できる、実行中のアプリケーションのインスタンス。開発環境は、上位の環境に昇格させる変更をテストするときに使用します。このタイプの環境は、テスト環境と呼ばれることもあります。
- 下位環境 — 初期ビルドやテストに使用される環境など、アプリケーションのすべての開発環境。
- 本番環境 — エンドユーザーがアクセスできる、実行中のアプリケーションのインスタンス。CI/CD パイプラインでは、本番環境が最後のデプロイ環境になります。
- 上位環境 — コア開発チーム以外のユーザーがアクセスできるすべての環境。これには、本番環境、本番前環境、ユーザー承認テスト環境などが含まれます。

エピック

アジャイル方法論で、お客様の作業の整理と優先順位付けに役立つ機能カテゴリ。エピックでは、要件と実装タスクの概要についてハイレベルな説明を提供します。例えば、 AWS CAF セキュリティエピックには、ID とアクセスの管理、検出コントロール、インフラストラクチャセキュリティ、データ保護、インシデント対応が含まれます。 AWS 移行戦略のエピックの詳細については、プログラム実装ガイドを参照してください。

ERP

「エンタープライズリソース計画」を参照してください。

探索的データ分析 (EDA)

データセットを分析してその主な特性を理解するプロセス。お客様は、データを収集または集計してから、パターンの検出、異常の検出、および前提条件のチェックのための初期調査を実行します。EDA は、統計の概要を計算し、データの可視化を作成することによって実行されます。

F

ファクトテーブル

スタースキーマの中央にあるテーブル。ビジネスオペレーションに関する定量的データが保存されます。一般的に、ファクトテーブルは、2 種類の列で構成されます。1 つは測定値が含まれる列、もう 1 つはディメンションテーブルへの外部キーが含まれる列です。

フェイルファスト

開発ライフサイクルを短縮するために、頻繁かつ段階的にテストを行う哲学であり、アジャイルアプローチでは、この考え方がきわめて重要です。

障害分離境界

では AWS クラウド、障害の影響を制限し、ワークロードの耐障害性を高めるのに役立つアベイラビリティーゾーン AWS リージョン、コントロールプレーン、データプレーンなどの境界。詳細については、「AWS 障害分離境界」を参照してください。

機能ブランチ

「ブランチ」を参照してください。

特徴量

お客様が予測に使用する入力データ。例えば、製造コンテキストでは、特徴量は製造ラインから定期的にキャプチャされるイメージの可能性もあります。

特徴量重要度

モデルの予測に対する特徴量の重要性。これは通常、Shapley Additive Deskonations (SHAP) や積分勾配など、さまざまな手法で計算できる数値スコアで表されます。詳細については、「 を使用した機械学習モデルの解釈可能性 AWS」を参照してください。

機能変換

追加のソースによるデータのエンリッチ化、値のスケーリング、単一のデータフィールドからの複数の情報セットの抽出など、機械学習プロセスのデータを最適化すること。これにより、機械学習モデルはデータの恩恵を受けることができます。例えば、「2021-05-27 00:15:37」の日付を「2021 年」、「5 月」、「木」、「15」に分解すると、学習アルゴリズムがさまざまなデータコンポーネントに関連する微妙に異なるパターンを学習するのに役立ちます。

数ショットプロンプト

LLM に、タスクと望ましい出力を示す例を少数提示した後に、類似のタスクを実行させること。この手法は、プロンプトに記述された例 (ショット) からモデルが学習する「インコンテキスト学

習」の一種です。数ショットプロンプトは、特定のフォーマット、推論、専門知識が必要なタスクに効果的です。「ゼロショットプロンプト」も参照してください。

FGAC

「きめ細かなアクセス制御」を参照してください。

きめ細かなアクセス制御 (FGAC)

複数の条件を使用してアクセス要求を許可または拒否すること。

フラッシュカット移行

変更データのキャプチャによる継続的なデータ複製を利用して、段階的なアプローチではなく、可能な限り短時間でデータを移行するデータベース移行方法。目的はダウンタイムを最小限に抑えることです。

FM

「基盤モデル」を参照してください。

基盤モデル (FM)

大規模な深層学習ニューラルネットワークであり、一般化およびラベル付けされていないデータからなる大規模データセットでトレーニングされています。FM により、言語理解、テキストおよび画像生成、自然言語での会話といった、一般的な各種タスクを実行できます。詳細については、「基盤モデルとは何ですか?」を参照してください。

G

生成 AI

AI モデルのサブセット。大量のデータでトレーニングされており、シンプルなテキストプロンプトを使用して、画像、動画、テキスト、オーディオなどの新しいコンテンツやアーティファクトを作成できます。詳細については、「生成 AI とは何ですか?」を参照してください。

ジオブロッキング

「地理的制限」を参照してください。

地理的制限 (ジオブロッキング)

特定の国のユーザーがコンテンツ配信にアクセスできないようにするための、Amazon CloudFront のオプション。アクセスを許可する国と禁止する国は、許可リストまたは禁止リスト

を使って指定します。詳細については、CloudFront ドキュメントの「コンテンツの地理的ディストリビューションの制限」を参照してください。

Gitflow ワークフロー

下位環境と上位環境が、ソースコードリポジトリでそれぞれ異なるブランチを使用する方法。Gitflow ワークフローは古いと見なされている方法であり、トランクベースのワークフローは推奨されている新しい方法です。

ゴールデンイメージ

システムまたはソフトウェアのスナップショットであり、システムまたはソフトウェアの新規インスタンスをデプロイするテンプレートとして使用されます。製造の例で言えば、ゴールデンイメージを使用すると、複数のデバイスにソフトウェアをプロビジョニングして、デバイス製造オペレーションの速度、スケーラビリティ、生産性を向上させることができます。

グリーンフィールド戦略

新しい環境に既存のインフラストラクチャが存在しないこと。システムアーキテクチャにグリーンフィールド戦略を導入する場合、既存のインフラストラクチャ (別名ブラウンフィールド) との互換性の制約を受けることなく、あらゆる新しいテクノロジーを選択できます。既存のインフラストラクチャを拡張している場合は、ブラウンフィールド戦略とグリーンフィールド戦略を融合させることもできます。

ガードレール

組織単位 (OU) 全般のリソース、ポリシー、コンプライアンスを管理するのに役立つ概略的なルール。予防ガードレールは、コンプライアンス基準に一致するようにポリシーを実施します。これらは、サービスコントロールポリシーと IAM アクセス許可の境界を使用して実装されます。検出ガードレールは、ポリシー違反やコンプライアンス上の問題を検出し、修復のためのアラートを発信します。これらは AWS Config、、 AWS Security Hub CSPM、Amazon GuardDuty、、 AWS Trusted Advisor Amazon Inspector、およびカスタム AWS Lambda チェックを使用して実装されます。

H

HA

「高可用性」を参照してください。

異種混在データベースの移行

別のデータベースエンジンを使用するターゲットデータベースへお客様の出典データベースの移行 (例えば、Oracle から Amazon Aurora)。異種間移行は通常、アーキテクチャの再設計作業の一部であり、スキーマの変換は複雑なタスクになる可能性があります。AWS は、スキーマの変換に役立つ AWS SCTを提供します。

高可用性 (HA)

課題や災害が発生した場合に、介入なしにワークロードを継続的に運用できること。HA システムは、自動的にフェイルオーバーし、一貫して高品質のパフォーマンスを提供し、パフォーマンスへの影響を最小限に抑えながらさまざまな負荷や障害を処理するように設計されています。

ヒストリアンのモダナイゼーション

製造業のニーズによりよく応えるために、オペレーションテクノロジー (OT) システムをモダナイズし、アップグレードするためのアプローチ。ヒストリアンは、工場内のさまざまなソースからデータを収集して保存するために使用されるデータベースの一種です。

ホールドアウトデータ

機械学習モデルのトレーニング用データセットから保留される、ラベル付き履歴データの一部。ホールドアウトデータを使用すると、モデル予測をホールドアウトデータと比較して、モデルのパフォーマンスを評価できます。

同種データベースの移行

お客様の出典データベースを、同じデータベースエンジンを共有するターゲットデータベース (Microsoft SQL Server から Amazon RDS for SQL Server など) に移行する。同種間移行は、通常、リホストまたはリプラットフォーム化の作業の一部です。ネイティブデータベースユーティリティを使用して、スキーマを移行できます。

ホットデータ

リアルタイムデータや最近の翻訳データなど、頻繁にアクセスされるデータ。通常、このデータには高速なクエリ応答を提供する高性能なストレージ階層またはクラスが必要です。

ホットフィックス

本番環境の重大な問題を修正するために緊急で配布されるプログラム。緊急性が高いため、通常の DevOps のリリースワークフローからは外れた形で実施されます。

ハイパーケア期間

カットオーバー直後、移行したアプリケーションを移行チームがクラウドで管理、監視して問題に対処する期間。通常、この期間は 1～4 日です。ハイパーケア期間が終了すると、アプリケーションに対する責任は一般的に移行チームからクラウドオペレーションチームに移ります。

I

IaC

「Infrastructure as Code」を参照してください。

ID ベースのポリシー

AWS クラウド 環境内のアクセス許可を定義する 1 つ以上の IAM プリンシパルにアタッチされたポリシー。

アイドル状態のアプリケーション

90 日間の平均的な CPU およびメモリ使用率が 5～20% のアプリケーション。移行プロジェクトでは、これらのアプリケーションを廃止するか、オンプレミスに保持するのが一般的です。

IIoT

「インダストリアル IoT」を参照してください。

イミュータブルインフラストラクチャ

既存インフラストラクチャの更新、パッチ適用、変更などを行わずに、本番環境ワークロードに使用する新規インフラストラクチャをデプロイするモデル。本質的に、イミュータブルインフラストラクチャは、ミュータブルインフラストラクチャよりも一貫性、信頼性、予測性に優れています。詳細については、 AWS Well-Architected フレームワークにある「イミュータブルインフラストラクチャを使用してデプロイする」のベストプラクティスを参照してください。

インバウンド (受信) VPC

AWS マルチアカウントアーキテクチャでは、アプリケーションの外部からネットワーク接続を受け入れ、検査し、ルーティングする VPC。AWS Security Reference Architecture では、アプリケーションとより広範なインターネット間の双方向のインターフェイスを保護するために、インバウンド、アウトバウンド、インスペクションの各 VPC を使用してネットワークアカウントを設定することを推奨しています。

増分移行

アプリケーションを 1 回ですべてカットオーバーするのではなく、小さい要素に分けて移行するカットオーバー戦略。例えば、最初は少数のマイクロサービスまたはユーザーのみを新しいシステムに移行する場合があります。すべてが正常に機能することを確認できたら、残りのマイクロサービスやユーザーを段階的に移行し、レガシーシステムを廃止できるようにします。この戦略により、大規模な移行に伴うリスクが軽減されます。

インダストリー 4.0

2016 年に Klaus Schwab 氏が提唱した用語で、接続、リアルタイムデータ、オートメーション、分析、AI/ML の進歩による、ビジネスプロセスのモダナイズを意味します。

インフラストラクチャ

アプリケーションの環境に含まれるすべてのリソースとアセット。

Infrastructure as Code (IaC)

アプリケーションのインフラストラクチャを一連の設定ファイルを使用してプロビジョニングし、管理するプロセス。IaC は、新しい環境を再現可能で信頼性が高く、一貫性のあるものにするため、インフラストラクチャを一元的に管理し、リソースを標準化し、スケールを迅速に行えるように設計されています。

インダストリアル IoT (IIoT)

製造、エネルギー、自動車、ヘルスケア、ライフサイエンス、農業などの産業部門におけるインターネットに接続されたセンサーやデバイスの使用。詳細については、「インダストリアル IoT (IIoT) デジタルトランスフォーメーション戦略の構築」を参照してください。

インスペクション VPC

AWS マルチアカウントアーキテクチャでは、VPC (同一または異なる 内 AWS リージョン)、インターネット、オンプレミスネットワーク間のネットワークトラフィックの検査を管理する一元化された VPCs。AWS Security Reference Architecture では、アプリケーションとより広範なインターネット間の双方向のインターフェイスを保護するために、インバウンド、アウトバウンド、インスペクションの各 VPC を使用してネットワークアカウントを設定することを推奨しています。

IoT

インターネットまたはローカル通信ネットワークを介して他のデバイスやシステムと通信する、センサーまたはプロセッサが組み込まれた接続済み物理オブジェクトのネットワーク。詳細については、「IoT とは」を参照してください。

解釈可能性

機械学習モデルの特性で、モデルの予測がその入力にどのように依存するかを人間が理解できる度合いを表します。詳細については、「 を使用した機械学習モデルの解釈可能性 AWS」を参照してください。

IoT

「IoT」を参照してください。

IT 情報ライブラリ (ITIL)

IT サービスを提供し、これらのサービスをビジネス要件に合わせるための一連のベストプラクティス。ITIL は ITSM の基盤を提供します。

IT サービス管理 (ITSM)

組織の IT サービスの設計、実装、管理、およびサポートに関連する活動。クラウドオペレーションと ITSM ツールの統合については、オペレーション統合ガイドを参照してください。

ITIL

「IT 情報ライブラリ」を参照してください。

ITSM

「IT サービス管理」を参照してください。

L

ラベルベースアクセス制御 (LBAC)

強制アクセス制御 (MAC) の実装で、ユーザーとデータ自体にそれぞれセキュリティラベル値が明示的に割り当てられます。ユーザーセキュリティラベルとデータセキュリティラベルが交差する部分によって、ユーザーに表示される行と列が決まります。

ランディングゾーン

ランディングゾーンは、スケーラブルで安全な、適切に設計されたマルチアカウント AWS 環境です。これは、組織がセキュリティおよびインフラストラクチャ環境に自信を持ってワークロードとアプリケーションを迅速に起動してデプロイできる出発点です。ランディングゾーンの詳細については、「安全でスケーラブルなマルチアカウント AWS 環境のセットアップ」を参照してください。

大規模言語モデル (LLM)

大量のデータで事前トレーニングされた深層学習 AI モデル。LLM では、質問への回答、ドキュメントの要約、他言語へのテキスト翻訳、文を完成させるなど、さまざまなタスクを実行できます。詳細については、「大規模言語モデル (LLM) とは何ですか?」を参照してください。

大規模な移行

300 台以上のサーバの移行。

LBAC

「ラベルベースアクセス制御」を参照してください。

最小特権

タスクの実行には必要最低限の権限を付与するという、セキュリティのベストプラクティス。詳細については、IAM ドキュメントの「最小特権アクセス許可を適用する」を参照してください。

リフトアンドシフト

「7 Rs」を参照してください。

リトルエンディアンシステム

最下位バイトを最初に格納するシステム。「エンディアン性」もご覧ください。

LLM

「大規模言語モデル」を参照してください。

下位環境

「環境」を参照してください。

M

機械学習 (ML)

パターン認識と学習にアルゴリズムと手法を使用する人工知能の一種。ML は、モノのインターネット (IoT) データなどの記録されたデータを分析して学習し、パターンに基づく統計モデルを生成します。詳細については、「機械学習」を参照してください。

メインブランチ

「ブランチ」を参照してください。

マルウェア

コンピュータのセキュリティやプライバシーを侵害するように設計されたソフトウェア。マルウェアは、コンピュータシステムの中断、機密情報の漏洩、不正アクセスを招く可能性があります。マルウェアの例には、ウイルス、ワーム、ランサムウェア、トロイの木馬、スパイウェア、キーロガーなどがあります。

マネージドサービス

AWS のサービス はインフラストラクチャレイヤー、オペレーティングシステム、プラットフォーム AWS を運用し、エンドポイントにアクセスしてデータを保存および取得します。マネージドサービスの例として、Amazon Simple Storage Service (Amazon S3) と Amazon DynamoDB が挙げられます。このサービスは、抽象化されたサービスとも呼ばれます。

製造実行システム (MES)

生産プロセスを追跡、モニタリング、文書化、制御するソフトウェアシステムであり、工場では、これによって、原材料から製品を完成させます。

MAP

「Migration Acceleration Program」を参照してください。

メカニズム

ツールを作成してその導入を推進し、導入結果を調べて調整を行うための包括的なプロセス。メカニズムとは、運用中にそれ自体を強化し改善するサイクルを意味します。詳細については、AWS 「 Well-Architected フレームワーク」の「メカニズムの構築」を参照してください。

メンバーアカウント

組織の一部である管理アカウント AWS アカウント 以外のすべて AWS Organizations。 アカウントが 組織のメンバーになることができるのは、一度に 1 つのみです。

MES

「製造実行システム」を参照してください。

Message Queuing Telemetry Transport (MQTT)

発行/サブスクライブのパターンに基づく、軽量のマシンツーマシン (M2M) 通信プロトコルであり、リソースに限りのある IoT デバイスに使用されます。

マイクロサービス

明確に定義された API を介して通信し、通常は小規模な自己完結型のチームが所有する、小規模で独立したサービスです。例えば、保険システムには、販売やマーケティングなどのビジネス

機能、または購買、請求、分析などのサブドメインにマッピングするマイクロサービスが含まれる場合があります。マイクロサービスの利点には、俊敏性、柔軟なスケーリング、容易なデプロイ、再利用可能なコード、回復力などがあります。詳細については、AWS「サーバーレスサービスを使用したマイクロサービスの統合」を参照してください。

マイクロサービスアーキテクチャ

各アプリケーションプロセスをマイクロサービスとして実行する独立したコンポーネントを使用してアプリケーションを構築するアプローチ。これらのマイクロサービスは、軽量 API を使用して、明確に定義されたインターフェイスを介して通信します。このアーキテクチャの各マイクロサービスは、アプリケーションの特定の機能に対する需要を満たすように更新、デプロイ、およびスケーリングできます。詳細については、「 でのマイクロサービスの実装 AWS」を参照してください。

Migration Acceleration Program (MAP)

組織がクラウドに移行するための強力な運用基盤を構築し、移行の初期コストを相殺するのに役立つコンサルティングサポート、トレーニング、サービスを提供する AWS プログラム。MAP には、組織的な方法でレガシー移行を実行するための移行方法論と、一般的な移行シナリオを自動化および高速化する一連のツールが含まれています。

大規模な移行

アプリケーションポートフォリオの大部分を次々にクラウドに移行し、各ウェーブでより多くのアプリケーションを高速に移動させるプロセス。この段階では、以前の段階から学んだベストプラクティスと教訓を使用して、移行ファクトリー チーム、ツール、プロセスのうち、オートメーションとアジャイルデリバリーによってワークロードの移行を合理化します。これは、AWS 移行戦略 の第 3 段階です。

移行ファクトリー

自動化された俊敏性のあるアプローチにより、ワークロードの移行を合理化する部門横断的なチーム。移行ファクトリーチームには、通常、運用、ビジネスアナリストおよび所有者、移行エンジニア、デベロッパー、およびスプリントで作業する DevOps プロフェッショナルが含まれます。エンタープライズアプリケーションポートフォリオの 20～50% は、ファクトリーのアプローチによって最適化できる反復パターンで構成されています。詳細については、このコンテンツセットの移行ファクトリーに関する解説とCloud Migration Factory ガイドを参照してください。

移行メタデータ

移行を完了するために必要なアプリケーションおよびサーバーに関する情報。移行パターンごとに、異なる一連の移行メタデータが必要です。移行メタデータの例としては、ターゲットサブネット、セキュリティグループ、 AWS アカウントなどがあります。

移行パターン

移行戦略、移行先、および使用する移行アプリケーションまたはサービスを詳述する、反復可能な移行タスク。例: AWS Application Migration Service を使用して Amazon EC2 への移行をリホストします。

Migration Portfolio Assessment (MPA)

オンラインツール。これによって、 AWS クラウドに移行するビジネスケースの検証に必要な情報を得られます。MPA は、詳細なポートフォリオ評価 (サーバーの適切なサイジング、価格設定、TCO 比較、移行コスト分析) および移行プラン (アプリケーションデータの分析とデータ収集、アプリケーションのグループ化、移行の優先順位付け、およびウェーブプランニング) を提供します。MPA ツール (ログインが必要) は、すべての AWS コンサルタントと APN パートナーコンサルタントが無料で利用できます。

移行準備状況評価 (MRA)

AWS CAF を使用して、組織のクラウド準備状況に関するインサイトを取得し、長所と短所を特定し、特定されたギャップを埋めるためのアクションプランを構築するプロセス。詳細については、移行準備状況ガイドを参照してください。MRA は、AWS 移行戦略の第一段階です。

移行戦略

ワークロードを AWS クラウドに移行するために使用するアプローチ。詳細については、この用語集の 7 Rs エントリと、「組織を動員して大規模な移行を加速する」を参照してください。

ML

「機械学習」を参照してください。

モダナイゼーション

古い (レガシーまたはモノリシック) アプリケーションとそのインフラストラクチャをクラウド内の俊敏で弾力性のある高可用性システムに変換して、コストを削減し、効率を高め、イノベーションを活用します。詳細については、「AWS クラウドでのアプリケーションのモダナイズ戦略」を参照してください。

モダナイゼーション準備状況評価

組織のアプリケーションのモダナイゼーションの準備状況を判断し、利点、リスク、依存関係を特定し、組織がこれらのアプリケーションの将来の状態をどの程度適切にサポートできるかを決定するのに役立つ評価。評価の結果として、ターゲットアーキテクチャのブループリント、モダナイゼーションプロセスの開発段階とマイルストーンを詳述したロードマップ、特定されたギャップに対処するためのアクションプランが得られます。詳細については、「AWS クラウドでのアプリケーションのモダナイゼーションの準備状況を評価する」を参照してください。

モノリシックアプリケーション (モノリス)

緊密に結合されたプロセスを持つ単一のサービスとして実行されるアプリケーション。モノリシックアプリケーションにはいくつかの欠点があります。1 つのアプリケーション機能エクスペリエンスの需要が急増する場合は、アーキテクチャ全体をスケーリングする必要があります。モノリシックアプリケーションの特徴を追加または改善することは、コードベースが大きくなると複雑になります。これらの問題に対処するには、マイクロサービスアーキテクチャを使用できます。詳細については、「モノリスをマイクロサービスに分解する」を参照してください。

MPA

「Migration Portfolio Assessment」を参照してください。

MQTT

「Message Queuing Telemetry Transport」を参照してください。

多クラス分類

複数のクラスの予測を生成するプロセス (2 つ以上の結果の 1 つを予測します)。例えば、機械学習モデルが、「この製品は書籍、自動車、電話のいずれですか?」 または、「このお客様にとって最も関心のある商品のカテゴリはどれですか?」と聞くかもしれません。

ミュータブルなインフラストラクチャ

本番ワークロードに使用する既存のインフラストラクチャを更新および変更するためのモデル。Well-Architected AWS フレームワークでは、一貫性、信頼性、予測可能性を向上させるために、イミュータブルインフラストラクチャの使用をベストプラクティスとして推奨しています。

O

OAC

「オリジンアクセス制御」を参照してください。

OAI

「オリジンアクセスアイデンティティ」を参照してください。

OCM

「組織変更管理」を参照してください。

オフライン移行

移行プロセス中にソースワークロードを停止させる移行方法。この方法はダウンタイムが長くなるため、通常は重要ではない小規模なワークロードに使用されます。

OI

「オペレーション統合」を参照してください。

Ola

「オペレーショナルレベルアグリーメント」を参照してください。

オンライン移行

ソースワークロードをオフラインにせずにターゲットシステムにコピーする移行方法。ワークロードに接続されているアプリケーションは、移行中も動作し続けることができます。この方法はダウンタイムがゼロから最小限で済むため、通常は重要な本番稼働環境のワークロードに使用されます。

OPC-UA

「Open Process Communications - Unified Architecture」を参照してください。

Open Process Communications - Unified Architecture (OPC-UA)

産業オートメーション用のマシンツーマシン (M2M) 通信プロトコル。OPC-UA により、相互運用の際に、データ暗号化、認証、認可の各スキームを標準化できます。

オペレーショナルレベルアグリーメント (OLA)

サービスレベルアグリーメント (SLA) をサポートするために、どの機能的 IT グループが互いに提供することを約束するかを明確にする契約。

運用準備状況レビュー (ORR)

質問と関連するベストプラクティスのチェックリスト。インシデントや起こり得る障害を理解、評価、防止したり、その範囲を縮小したりする際に役立ちます。詳細については、 AWS Well-Architected フレームワークの「Operational Readiness Reviews (ORR)」を参照してください。

運用テクノロジー (OT)

産業オペレーション、機器、インフラストラクチャを制御するために物理環境と連携させるハードウェアおよびソフトウェアシステム。製造分野では、Industry 4.0 への変革を進める上で、OT と情報技術 (IT) システムの統合に焦点が当てられています。

オペレーション統合 (OI)

クラウドでオペレーションをモダナイズするプロセスには、準備計画、オートメーション、統合が含まれます。詳細については、オペレーション統合ガイドを参照してください。

組織の証跡

組織 AWS アカウント 内のすべての のすべてのイベント AWS CloudTrail を ログに記録する によって作成された証跡 AWS Organizations。証跡は、組織に含まれている各 AWS アカウント に作成され、各アカウントのアクティビティを追跡します。詳細については、CloudTrail ドキュメントの「組織の証跡の作成」を参照してください。

組織変更管理 (OCM)

人材、文化、リーダーシップの観点から、主要な破壊的なビジネス変革を管理するためのフレームワーク。OCM は、変化の導入を加速し、移行問題に対処し、文化や組織の変化を推進することで、組織が新しいシステムと戦略の準備と移行するのを支援します。 AWS 移行戦略では、クラウド導入プロジェクトに必要な変化のスピードにより、このフレームワークは人材アクセラレーションと呼ばれます。詳細については、OCM ガイドを参照してください。

オリジンアクセス制御 (OAC)

Amazon Simple Storage Service (Amazon S3) コンテンツを保護するための、CloudFront のアクセス制限の強化オプション。OAC は AWS リージョン、すべての S3 バケット、 AWS KMS (SSE-KMS) によるサーバー側の暗号化、S3 バケットへの動的 PUT および DELETEリクエストをサポートします。

オリジンアクセスアイデンティティ (OAI)

CloudFront の、Amazon S3 コンテンツを保護するためのアクセス制限オプション。OAI を使用すると、CloudFront が、Amazon S3 に認証可能なプリンシパルを作成します。認証されたプリンシパルは、S3 バケット内のコンテンツに、特定の CloudFront ディストリビューションを介してのみアクセスできます。OAC も併せて参照してください。OAC では、より詳細な、強化されたアクセス制御が可能です。

ORR

「運用準備状況レビュー」を参照してください。

OT

「運用テクノロジー」を参照してください。

アウトバウンド (送信) VPC

AWS マルチアカウントアーキテクチャでは、アプリケーション内から開始されたネットワーク接続を処理する VPC。AWS Security Reference Architecture では、アプリケーションとより広範なインターネット間の双方向のインターフェイスを保護するために、インバウンド、アウトバウンド、インスペクションの各 VPC を使用してネットワークアカウントを設定することを推奨しています。

P

アクセス許可の境界

ユーザーまたはロールが使用できるアクセス許可の上限を設定する、IAM プリンシパルにアタッチされる IAM 管理ポリシー。詳細については、IAM ドキュメントのアクセス許可の境界を参照してください。

個人を特定できる情報 (PII)

直接閲覧した場合、または他の関連データと組み合わせた場合に、個人の身元を合理的に推測するために使用できる情報。PII の例には、氏名、住所、連絡先情報などがあります。

PII

「個人を特定できる情報」を参照してください。

プレイブック

クラウドでのコアオペレーション機能の提供など、移行に関連する作業を取り込む、事前定義された一連のステップ。プレイブックは、スクリプト、自動ランブック、またはお客様のモダナイズされた環境を運用するために必要なプロセスや手順の要約などの形式をとることができます。

PLC

「プログラマブルロジックコントローラー」を参照してください。

PLM

「製品ライフサイクル管理」を参照してください。

ポリシー

次の操作を可能にするオブジェクト: アクセス許可を定義する (ID ベースのポリシーを参照)。アクセス条件を指定する (リソースベースのポリシーを参照)。 AWS Organizations の組織における全アカウントにアクセス許可の上限を定義する (サービスコントロールポリシーを参照)。

多言語の永続性

データアクセスパターンやその他の要件に基づいて、マイクロサービスのデータストレージテクノロジーを個別に選択します。マイクロサービスが同じデータストレージテクノロジーを使用している場合、実装上の問題が発生したり、パフォーマンスが低下する可能性があります。マイクロサービスは、要件に最も適合したデータストアを使用すると、より簡単に実装でき、パフォーマンスとスケーラビリティが向上します。

ポートフォリオ評価

移行を計画するために、アプリケーションポートフォリオの検出、分析、優先順位付けを行うプロセス。詳細については、「移行の準備状況の評価」を参照してください。

述語

`true` または `false` を返すためのクエリ条件。一般的に、`WHERE` 句に記述されます。

述語プッシュダウン

データベースクエリを最適化する手法。これによって、転送前にクエリ内のデータをフィルタリングします。この手法を取ると、リレーショナルデータベースから取得し処理する必要のあるデータの量が減少するため、クエリのパフォーマンスが向上します。

予防的コントロール

イベントの発生を防ぐように設計されたセキュリティコントロール。このコントロールは、ネットワークへの不正アクセスや好ましくない変更を防ぐ最前線の防御です。詳細については、「AWSでのセキュリティコントロールの実装」の「予防的コントロール」を参照してください。

プリンシパル

アクションを実行し AWS 、リソースにアクセスできる のエンティティ。このエンティティは通常、、IAM AWS アカウントロール、または ユーザーのルートユーザーです。詳細については、IAM ドキュメントの「ロールに関する用語と概念」にあるプリンシパルを参照してください。

プライバシーバイデザイン

開発プロセス全体を通してプライバシーが考慮されているシステムエンジニアリングのアプローチ。

プライベートホストゾーン

1 つ以上の VPC 内のドメインとそのサブドメインへの DNS クエリに対し、Amazon Route 53 がどのように応答するかに関する情報を保持するコンテナ。詳細については、Route 53 ドキュメントの「プライベートホストゾーンの使用」を参照してください。

プロアクティブコントロール

非準拠リソースのデプロイ防止を目的としたセキュリティコントロール。このコントロールにより、プロビジョニング前にリソースをスキャンします。コントロールに準拠していないリソースは、プロビジョニングされません。詳細については、 AWS Control Tower ドキュメントの「コントロールリファレンスガイド」および「 セキュリティコントロールの実装」の「プロアクティブコントロール」を参照してください。 AWS

製品ライフサイクル管理 (PLM)

製品の設計、開発、発売から、成長、成熟、衰退、廃棄に至る、製品のライフサイクル全体を通してデータとプロセスを管理すること。

本番環境

「環境」を参照してください。

プログラマブルロジックコントローラー (PLC)

製造分野で使用される、信頼性と適応性に優れたコンピュータであり、これによって、マシンをモニタリングするとともに、製造プロセスを自動化します。

プロンプトチェイニング

1 つの LLM プロンプトによる出力を次のプロンプトの入力に使用して、より良いレスポンスを生成します。この手法を使用すると、複雑なタスクをサブタスクに分割したり、事前レスポンスを繰り返し改良または拡張したりできます。これによって、モデルのレスポンスの精度と関連性が向上し、粒度の高いパーソナライズされた結果を得られます。

仮名化

データセット内の個人識別子をプレースホルダー値に置き換えるプロセス。仮名化は個人のプライバシー保護に役立ちます。仮名化されたデータは、依然として個人データとみなされます。

発行/サブスクライブ (pub/sub)

マイクロサービス間の非同期通信を可能にするパターン。これにより、スケーラビリティと応答性を向上させます。例えば、マイクロサービスベースの MES の場合、マイクロサービスは、他のマイクロサービスがサブスクライブ可能なチャネルにイベントメッセージを発行できます。このシステムでは、発行サービスの変更なしに、新規マイクロサービスを追加できます。

Q

クエリプラン

手順などの一連のステップであり、SQL リレーショナルデータベースシステムのデータにアクセスするために使用されます。

クエリプランのリグレッション

データベースサービスのオプティマイザーが、データベース環境に特定の変更が加えられる前に選択されたプランよりも最適性の低いプランを選択すること。これは、統計、制限事項、環境設定、クエリパラメータのバインディングの変更、およびデータベースエンジンの更新などが原因である可能性があります。

R

RACI マトリックス

「実行責任者、説明責任者、協業先、報告先 (RACI)」を参照してください。

RAG

「検索拡張生成」を参照してください。

ランサムウェア

決済が完了するまでコンピュータシステムまたはデータへのアクセスをブロックするように設計された、悪意のあるソフトウェア。

RASCI マトリックス

「実行責任者、説明責任者、協業先、報告先 (RACI)」を参照してください。

RCAC

「行と列のアクセス制御」を参照してください。

リードレプリカ

読み取り専用に使用されるデータベースのコピー。クエリをリードレプリカにルーティングして、プライマリデータベースへの負荷を軽減できます。

リアーキテクト

「7 Rs」を参照してください。

目標復旧時点 (RPO)

最後のデータリカバリポイントからの最大許容時間です。これにより、最後の回復時点からサービスが中断されるまでの間に許容できるデータ損失の程度が決まります。

目標復旧時間 (RTO)

サービスの中断から復旧までの最大許容遅延時間。

リファクタリング

「7 Rs」を参照してください。

リージョン

地理的エリア内の AWS リソースのコレクション。各 AWS リージョン は、耐障害性、安定性、耐障害性を提供するために、他の から分離され、独立しています。詳細については、「アカウントが使用できる AWS リージョン を指定する」を参照してください。

リグレッション

数値を予測する機械学習手法。例えば、「この家はどれくらいの値段で売れるでしょうか?」という問題を解決するために、機械学習モデルは、線形回帰モデルを使用して、この家に関する既知の事実 (平方フィートなど) に基づいて家の販売価格を予測できます。

リホスト

「7 Rs」を参照してください。

リリース

デプロイプロセスで、変更を本番環境に昇格させること。

再配置

「7 Rs」を参照してください。

リプラットフォーム

「7 Rs」を参照してください。

再購入

「7 Rs」を参照してください。

回復性

中断に抵抗または中断から回復するアプリケーションの機能。 AWS クラウドでの回復力を計画する際には、一般的に、高可用性とディザスタリカバリが考慮されます。詳細については、「AWS クラウド の耐障害性」を参照してください。

リソースベースのポリシー

Amazon S3 バケット、エンドポイント、暗号化キーなどのリソースにアタッチされたポリシー。このタイプのポリシーは、アクセスが許可されているプリンシパル、サポートされているアクション、その他の満たすべき条件を指定します。

実行責任者、説明責任者、協業先、報告先 (RACI) に基づくマトリックス

移行活動とクラウド運用に関わるすべての関係者の役割と責任を定義したマトリックス。マトリックスの名前は、マトリックスで定義されている責任の種類、すなわち責任 (R) 、説明責任 (A) 、協議 (C) 、情報提供 (I) に由来します。サポート (S) タイプはオプションです。サポートが含まれる場合は RASCI マトリックスと呼ばれ、含まれない場合は RACI マトリックスと呼ばれます。

レスポンシブコントロール

有害事象やセキュリティベースラインからの逸脱について、修復を促すように設計されたセキュリティコントロール。詳細については、「 AWSでのセキュリティコントロールの実装」の「レスポンシブコントロール」を参照してください。

保持

「7 Rs」を参照してください。

廃止

「7 Rs」を参照してください。

検索拡張生成 (RAG)

生成 AI の技術。これにより、LLM では、レスポンスの生成前に、トレーニングデータソースの外部にある信頼できるデータソースが参照されます。例えば、RAG モデルによって、組織のナレッジベースまたはカスタムデータのセマンティック検索を実行できる場合があります。細については、「RAG (検索拡張生成) とは何ですか?」を参照してください。

ローテーション

定期的にシークレット情報を更新して、攻撃者が認証情報にアクセスするのをより困難にするプロセス。

行と列のアクセス制御 (RCAC)

アクセスルールが定義された、基本的で柔軟な SQL 表現の使用。RCAC は行権限と列マスクで構成されています。

RPO

「目標復旧時点」を参照してください。

RTO

「目標復旧時間」を参照してください。

ランブック

特定のタスクを実行するために必要な手動または自動化された一連の手順。これらは通常、エラー率の高い反復操作や手順を合理化するために構築されています。

S

SAML 2.0

多くの ID プロバイダー (IdP) が使用しているオープンスタンダード。この機能を使用すると、フェデレーティッドシングルサインオン (SSO) が有効になるため、ユーザーは組織内のすべてのユーザーを IAM で作成しなくても、 AWS マネジメントコンソール にログインしたり AWS 、API オペレーションを呼び出すことができます。SAML 2.0 ベースのフェデレーションの詳細については、IAM ドキュメントの「SAML 2.0 ベースのフェデレーションについて」を参照してください。

SCADA

「監視制御とデータ取得」を参照してください。

SCP

「サービスコントロールポリシー」を参照してください。

シークレット

暗号化された形式で保存する AWS Secrets Managerパスワードやユーザー認証情報などの機密情報または制限付き情報。シークレット値とそのメタデータで構成されます。シークレット値には、バイナリ、1 つの文字列、複数の文字列を指定できます。詳細については、Secrets Manager ドキュメントの「Secrets Manager シークレットの概要」を参照してください。

セキュリティバイデザイン

開発プロセス全体を通してセキュリティが考慮されているシステムエンジニアリングのアプローチ。

セキュリティコントロール

脅威アクターによるセキュリティ脆弱性の悪用を防止、検出、軽減するための、技術上または管理上のガードレール。セキュリティコントロールには、主に 4 つの種類があります。4 つとは、予防、検出、レスポンシブ、プロアクティブです。

セキュリティ強化

アタックサーフェスを狭めて攻撃への耐性を高めるプロセス。このプロセスには、不要になったリソースの削除、最小特権を付与するセキュリティのベストプラクティスの実装、設定ファイル内の不要な機能の無効化、といったアクションが含まれています。

Security Information and Event Management (SIEM) システム

セキュリティ情報管理 (SIM) とセキュリティイベント管理 (SEM) のシステムを組み合わせたツールとサービス。SIEM システムは、サーバー、ネットワーク、デバイス、その他ソースからデータを収集、モニタリング、分析して、脅威やセキュリティ違反を検出し、アラートを発信します。

セキュリティレスポンスの自動化

セキュリティイベントへの自動レスポンスまたは自動修復を目的として、事前定義およびプログラムされたアクション。これらの自動化は、セキュリティのベストプラクティスを実装するのに役立つ検出的または応答的な AWS セキュリティコントロールとして機能します。自動レスポンスアクションの例には、VPC セキュリティグループの変更、Amazon EC2 インスタンスへのパッチ適用、認証情報の更新などがあります。

サーバー側の暗号化

送信先で、それ AWS のサービス を受け取る によるデータの暗号化。

サービスコントロールポリシー (SCP)

AWS Organizationsの組織内の、すべてのアカウントのアクセス許可を一元的に管理するポリシー。SCP は、管理者がユーザーまたはロールに委任するアクションに、ガードレールを定義したり、アクションの制限を設定したりします。SCP は、許可リストまたは拒否リストとして、許可または禁止するサービスやアクションを指定する際に使用できます。詳細については、 AWS Organizations ドキュメントの「サービスコントロールポリシー」を参照してください。

サービスエンドポイント

のエントリポイントの URL AWS のサービス。ターゲットサービスにプログラムで接続するには、エンドポイントを使用します。詳細については、「AWS 全般のリファレンス」の「AWS のサービス エンドポイント」を参照してください。

サービスレベルアグリーメント (SLA)

サービスのアップタイムやパフォーマンスなど、IT チームがお客様に提供すると約束したものを明示した合意書。

サービスレベルインジケータ (SLI)

エラー率、可用性、スループットといった、サービスパフォーマンス面の指標。

サービスレベル目標 (SLO)

サービスレベルインジケータによって測定され、サービスの状態を表すターゲットメトリクス。

責任共有モデル

クラウドのセキュリティとコンプライアンス AWS について と共有する責任を説明するモデル。AWS はクラウドのセキュリティを担当しますが、 はクラウドのセキュリティを担当します。詳細については、「責任共有モデル」を参照してください。

SIEM

「Security Information and Event Management システム」を参照してください。

単一障害点 (SPOF)

特定のアプリケーションを構成する単一の重要なコンポーネントで発生し、システム稼働に支障をきたす可能性のある障害。

SLA

「サービスレベルアグリーメント」を参照してください。

SLI

「サービスレベルインジケータ」を参照してください。

SLO

「サービスレベルの目標」を参照してください。

スプリットアンドシードモデル

モダナイゼーションプロジェクトのスケーリングと加速のためのパターン。新機能と製品リリースが定義されると、コアチームは解放されて新しい製品チームを作成します。これにより、お客様の組織の能力とサービスの拡張、デベロッパーの生産性の向上、迅速なイノベーションのサポートに役立ちます。詳細については、「AWS クラウドでのアプリケーションをモダナイズするための段階的アプローチ」を参照してください。

SPOF

「単一障害点」を参照してください。

スタースキーマ

データベースの編成構造を意味し、1 つの大きいファクトテーブルにトランザクションデータまたは測定データが保存され、1 つ以上の小さいディメンションテーブルにデータ属性が保存されます。この構造は、データウェアハウスやビジネスインテリジェンスを用途とするように設計されています。

strangler fig パターン

レガシーシステムが廃止されるまで、システム機能を段階的に書き換えて置き換えることにより、モノリシックシステムをモダナイズするアプローチ。このパターンは、宿主の樹木から根を成長させ、最終的にその宿主を包み込み、宿主に取って代わるイチジクのつるを例えています。そのパターンは、モノリシックシステムを書き換えるときのリスクを管理する方法として Martin Fowler により提唱されました。このパターンの適用方法の例については、「コンテナと Amazon API Gateway を使用して、従来の Microsoft ASP.NET (ASMX) ウェブサービスを段階的にモダナイズ」を参照してください。

サブネット

VPC 内の IP アドレスの範囲。サブネットは、1 つのアベイラビリティーゾーンに存在する必要があります。

監視制御とデータ取得 (SCADA)

製造分野において、ハードウェアとソフトウェアを使用して物理アセットと本番運用をモニタリングするシステム。

対称暗号化

データの暗号化と復号に同じキーを使用する暗号化のアルゴリズム。

合成テスト

ユーザーとのやり取りをシミュレートして、起こり得る問題を検出したり、パフォーマンスをモニタリングしたりすることで、システムをテストします。Amazon CloudWatch Synthetics を使用すると、こうしたテストを作成できます。

システムプロンプト

コンテキスト、指示、ガイドラインなどを提示して、LLM に動作を指示する手法。システムプロンプトは、コンテキストを設定して、ユーザーとやり取りするルールを確立するのに有用です。

T

タグ

AWS リソースを整理するためのメタデータとして機能するキーと値のペア。タグは、 リソースの管理、識別、整理、検索、フィルタリングに役立ちます。詳細については、「AWS リソースのタグ付け」を参照してください。

ターゲット変数

監督された機械学習でお客様が予測しようとしている値。これは、結果変数のことも指します。例えば、製造設定では、ターゲット変数が製品の欠陥である可能性があります。

タスクリスト

ランブックの進行状況を追跡するために使用されるツール。タスクリストには、ランブックの概要と完了する必要のある一般的なタスクのリストが含まれています。各一般的なタスクには、推定所要時間、所有者、進捗状況が含まれています。

テスト環境

「環境」を参照してください。

トレーニング

お客様の機械学習モデルに学習するデータを提供すること。トレーニングデータには正しい答えが含まれている必要があります。学習アルゴリズムは入力データ属性をターゲット (お客様が予測したい答え) にマッピングするトレーニングデータのパターンを検出します。これらのパターンをキャプチャする機械学習モデルを出力します。そして、お客様が機械学習モデルを使用して、ターゲットがわからない新しいデータでターゲットを予測できます。

トランジットゲートウェイ

VPC とオンプレミスネットワークを相互接続するために使用できる、ネットワークの中継ハブ。詳細については、 AWS Transit Gateway ドキュメントの「トランジットゲートウェイとは」を参照してください。

トランクベースのワークフロー

デベロッパーが機能ブランチで機能をローカルにビルドしてテストし、その変更をメインブランチにマージするアプローチ。メインブランチはその後、開発環境、本番前環境、本番環境に合わせて順次構築されます。

信頼されたアクセス

ユーザーに代わって AWS Organizations およびそのアカウントで組織内でタスクを実行するために指定したサービスにアクセス許可を付与します。信頼されたサービスは、サービスにリンクされたロールを必要なときに各アカウントに作成し、ユーザーに代わって管理タスクを実行します。詳細については、 ドキュメントの「Using AWS Organizations with other AWS services AWS Organizations 」を参照してください。

チューニング

機械学習モデルの精度を向上させるために、お客様のトレーニングプロセスの側面を変更する。例えば、お客様が機械学習モデルをトレーニングするには、ラベル付けセットを生成し、ラベルを追加します。これらのステップを、異なる設定で複数回繰り返して、モデルを最適化します。

ツーピザチーム

2 枚のピザを分け合えることができるくらい小さな DevOps チーム。ツーピザチームの規模では、ソフトウェア開発におけるコラボレーションに最適な機会が確保されます。

U

不確実性

予測機械学習モデルの信頼性を損なう可能性がある、不正確、不完全、または未知の情報を指す概念。不確実性には、次の 2 つのタイプがあります。認識論的不確実性は、限られた、不完全なデータによって引き起こされ、弁論的不確実性 は、データに固有のノイズとランダム性によって引き起こされます。詳細については、深層学習システムにおける不確実性の定量化ガイドを参照してください。

未分化なタスク

ヘビーリフティングとも呼ばれ、アプリケーションの作成と運用には必要だが、エンドユーザーに直接的な価値をもたらさなかったり、競争上の優位性をもたらしたりしない作業です。未分化なタスクの例としては、調達、メンテナンス、キャパシティプランニングなどがあります。

上位環境

「環境」を参照してください。

V

バキューミング

ストレージを再利用してパフォーマンスを向上させるために、増分更新後にクリーンアップを行うデータベースのメンテナンス操作。

バージョンコントロール

リポジトリ内のソースコードへの変更など、変更を追跡するプロセスとツール。

VPC ピアリング

プライベート IP アドレスを使用してトラフィックをルーティングできる、2 つの VPC 間の接続。詳細については、Amazon VPC ドキュメントの「VPC ピア機能とは」を参照してください。

脆弱性

システムのセキュリティを脅かすソフトウェアまたはハードウェアの欠陥。

W

ウォームキャッシュ

頻繁にアクセスされる最新の関連データを含むバッファキャッシュ。データベースインスタンスはバッファキャッシュから、メインメモリまたはディスクからよりも短い時間で読み取りを行うことができます。

ウォームデータ

アクセス頻度の低いデータ。この種類のデータをクエリする場合、通常は適度に遅いクエリでも問題ありません。

ウィンドウ関数

現在のレコードに何らかの形で関連している行のグループに計算を実行する SQL 関数。ウィンドウ関数は、移動平均を計算したり、現在の行の相対位置に基づいて他の行の値にアクセスするといったタスクの処理に役立ちます。

ワークロード

ビジネス価値をもたらすリソースとコード (顧客向けアプリケーションやバックエンドプロセスなど) の総称。

ワークストリーム

特定のタスクセットを担当する移行プロジェクト内の機能グループ。各ワークストリームは独立していますが、プロジェクト内の他のワークストリームをサポートしています。たとえば、ポートフォリオワークストリームは、アプリケーションの優先順位付け、ウェーブ計画、および移行メタデータの収集を担当します。ポートフォリオワークストリームは、これらの設備を移行ワークストリームで実現し、サーバーとアプリケーションを移行します。

WORM

「Write-Once-Read-Many」を参照してください。

WQF

「AWS ワークロード資格フレームワーク」を参照してください

Write-Once-Read-Many (WORM)

データを 1 回のみ書き込むことで、データの削除や変更を防ぐストレージモデル。承認済みユーザーは、必要な回数だけデータを読み取ることができますが、変更することはできません。このデータストレージインフラストラクチャは、イミュータブルと見なされます。

Z

ゼロデイエクスプロイト

ゼロデイ脆弱性を悪用した攻撃 (一般的にマルウェアによる)。

ゼロデイ脆弱性

実稼働システムにおける未解決の欠陥または脆弱性。脅威アクターは、このような脆弱性を利用してシステムを攻撃する可能性があります。開発者は、よく攻撃の結果で脆弱性に気付きます。

ゼロショットプロンプト

LLM にタスク実行の手順は提示するが、実行のガイドとして役立つ例 (ショット) は提示しない方法。LLM は、事前トレーニング済みの知識を使用してタスクを処理する必要があります。ゼロショットプロンプトの有効性は、タスクの複雑さとプロンプトの品質によって異なります。「数ショットプロンプト」も参照してください。

ゾンビアプリケーション

平均 CPU およびメモリ使用率が 5% 未満のアプリケーション。移行プロジェクトでは、これらのアプリケーションを廃止するのが一般的です。

翻訳は機械翻訳により提供されています。提供された翻訳内容と英語版の間で齟齬、不一致または矛盾がある場合、英語版が優先します。