



クラウド設計パターン、アーキテクチャ、および実装

AWS 規範ガイド



AWS 規範ガイド: クラウド設計パターン、アーキテクチャ、および実装

Copyright © 2024 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

Amazon の商標とトレードドレスは、Amazon 以外の製品またはサービスとの関連において、顧客に混乱を招いたり、Amazon の名誉または信用を毀損するような方法で使用することはできません。Amazon が所有していない他のすべての商標は、それぞれの所有者の所有物であり、Amazon と提携、接続、または後援されている場合とされていない場合があります。

Table of Contents

序章	1
ターゲットを絞ったビジネス成果	2
腐敗防止層パターン	3
Intent	3
モチベーション	3
適用性	3
問題と考慮事項	4
実装	5
高レベルのアーキテクチャ	5
を使用した実装AWSサービス	6
サンプルコード	7
GitHub倉庫	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
メリット	11
デメリット	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 ヘッダルーティング	17
メリット	18
デメリット	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 の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」	37
関連情報	41
動画	41
パブリッシュ – サブスクライブパターン	42
Intent	42
導入する理由	42
適用対象	42
問題点と考慮事項	43
実装	44
高レベルのアーキテクチャ	44
AWS のサービスを使用した実装	45
ワークショップ	47
ブログの参考情報	47
関連情報	47
バックオフパターンで再試行	48

Intent	48
モチベーション	48
適用性	48
問題と考慮事項	48
実装	49
高レベルのアーキテクチャ	49
を使用した実装AWSサービス	50
サンプルコード	51
GitHub倉庫	51
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
散布図パターン	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
目的	78
モチベーション	78
適用性	79
問題と考慮事項	79
実装	80
高レベルのアーキテクチャ	81
AWSサービスを使った実装	85
ワークショップ	90
ブログリファレンス	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 の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	109
C	111
D	114
E	118

F	120
G	121
H	122
I	123
L	125
M	126
O	130
P	133
Q	135
R	136
S	138
T	142
U	143
V	144
W	144
Z	145
.....	cxlvii

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](#))

このガイドでは、AWS のサービスを使用して、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、リグレー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。オープンソース GitHub のサンプルとワークショップのリンクは、利用可能な場合は提供されています。

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](#)
- [散布図](#)
- [Strangler fig](#)
- [トランザクション送信トレイ](#)

ターゲットを絞ったビジネス成果

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層として機能します。上流チームが確立した通信契約を利用する前に、上流の境界付きコンテキスト (モノリス) のモデルを下流の境界付きコンテキスト (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、下流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や、上流モデルが変更不可能なレガシーシステムである場合に当てはまる可能性があります。また、発信者の通話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通話者が変更されるのを防ぐことで、変革のリスクと業務の中断を軽減します。

モチベーション

移行プロセス中に、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、新しく移行され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。モノリス内の機能がこれらのマイクロサービスを呼び出す必要がある場合は、呼び出し元のサービスを変更することなく、呼び出しを移行された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、モノリスは、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することで、マイクロサービスを透過的に呼び出すことができます。

適用性

次のような場合にこのパターンを使用すること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は、マイクロサービスに移行された関数と通信する必要があり、移行されたサービスドメインモデルとセマンティクスは元の機能とは異なります。
- 2つのシステムはセマンティクスが異なり、データを交換する必要がありますが、一方のシステムを他方のシステムと互換性があるように変更することは現実的ではありません。
- 迅速でシンプルなアプローチを使用して、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たいと考えています。
- アプリケーションは外部システムと通信しています。

問題と考慮事項

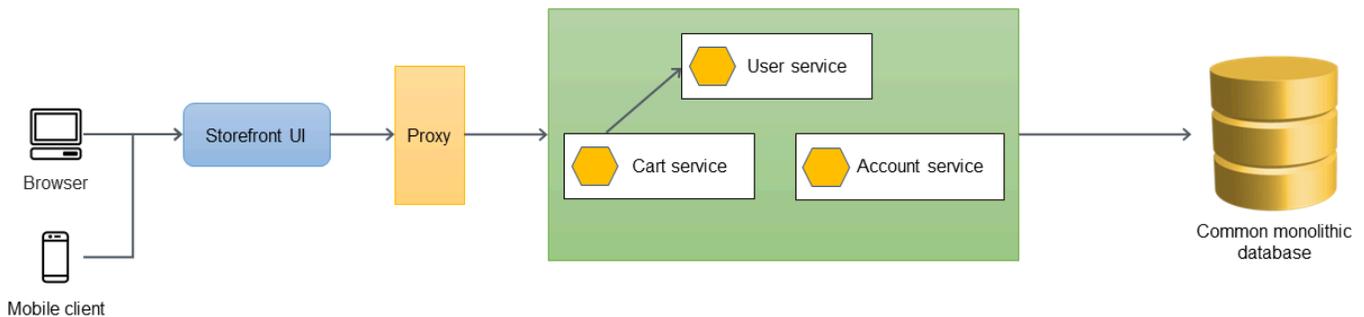
- チームの依存関係:システム内のさまざまなサービスがさまざまなチームによって所有されている場合、移行されたサービスの新しいドメインモデルのセマンティクスにより、呼び出し側のシステムが変更される可能性があります。ただし、チームには他の優先事項がある可能性があるため、これらの変更を調整して行うことができない場合があります。ACL は呼び出し先を切り離し、新しいサービスのセマンティクスに合わせて呼び出しを変換します。これにより、呼び出し元が現在のシステムを変更する必要がなくなります。
- 運用上のオーバーヘッド:ACL パターンの運用と保守には追加の労力が必要です。この作業には、ACLを監視および警告ツール、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) プロセスと統合することが含まれます。
- 単一障害点:ACL に障害があると、ターゲットサービスにアクセスできなくなり、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、リトライ機能とサーキットブレーカーを組み込む必要があります。を参照してください[バックオフして再試行](#)そして[サーキットブレーカー](#)これらのオプションについてさらに理解を深めるためのパターン。適切なアラートとロギングを設定すると、平均解決時間 (MTTR) が改善されます。
- 技術的負債:移行またはモダナイゼーション戦略の一環として、ACL が一時的な解決策になるのか、暫定的な解決策になるのか、それとも長期的な解決策になるのかを検討してください。暫定的な解決策であれば、ACL を技術的負債として記録し、依存関係にあるすべての発信者を移行した後、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へのリクエストの変換により、レイテンシーが発生する可能性があります。ACL を実稼働環境に展開する前に、応答時間に敏感なアプリケーションのパフォーマンスストレランスを定義してテストすることをお勧めします。
- スケーリングのボトルネック:サービスがピーク負荷までスケーリングできる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リングする場合は、それに応じてスケーリングするように 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して、呼び出しを複数のサービスまたは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、および耐障害性を考慮してください。

実装

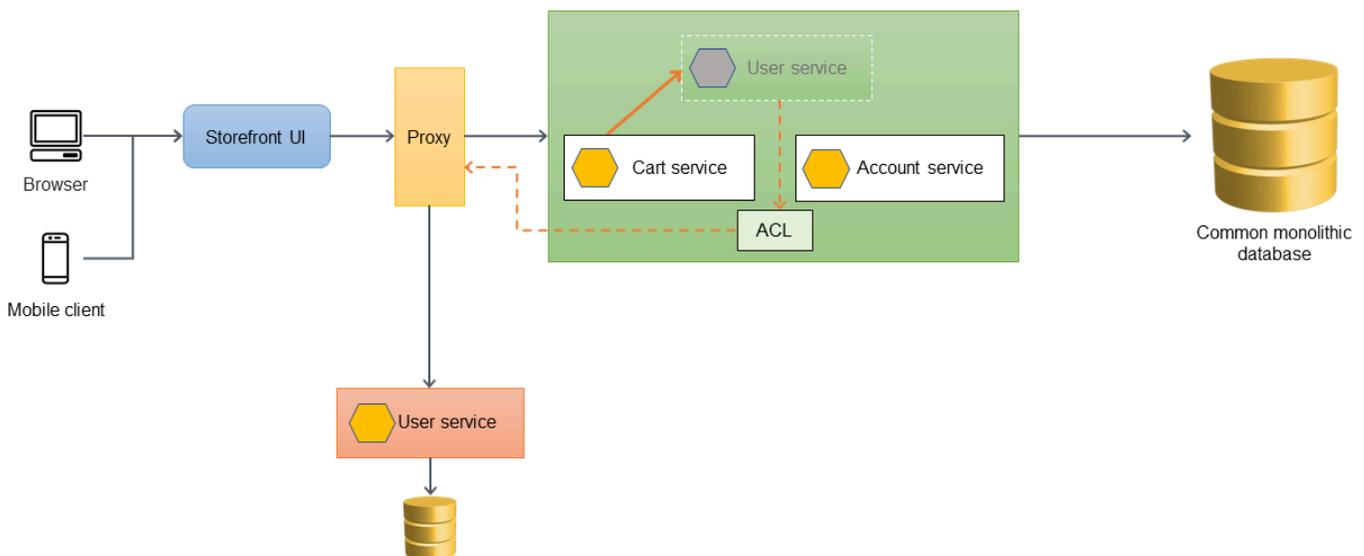
ACL は、移行するサービスに固有のクラスとして、または独立したサービスとしてモノリシックアプリケーション内に実装できます。すべての依存サービスがマイクロサービスアーキテクチャに移行されたら、ACL を廃止する必要があります。

高レベルのアーキテクチャ

次のアーキテクチャ例では、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の 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、アプリケーションはモノリシックなリレーショナルデータベースを使用します。

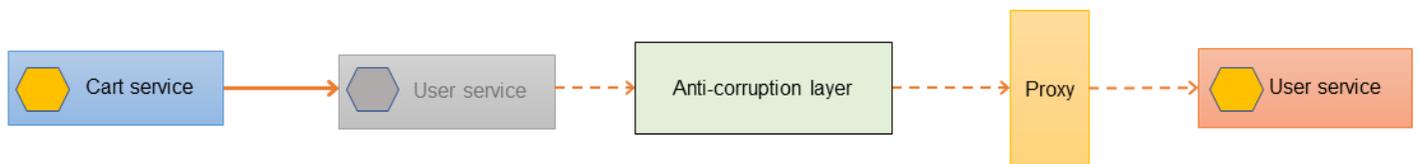


次のアーキテクチャでは、ユーザーサービスは新しいマイクロサービスに移行されました。カートサービスはユーザーサービスを呼び出すが、実装はモノリス内では使用できなくなった。また、新しく移行したサービスのインターフェースが、モノリシックアプリケーション内にあったときの以前のインターフェースと一致しない可能性もあります。



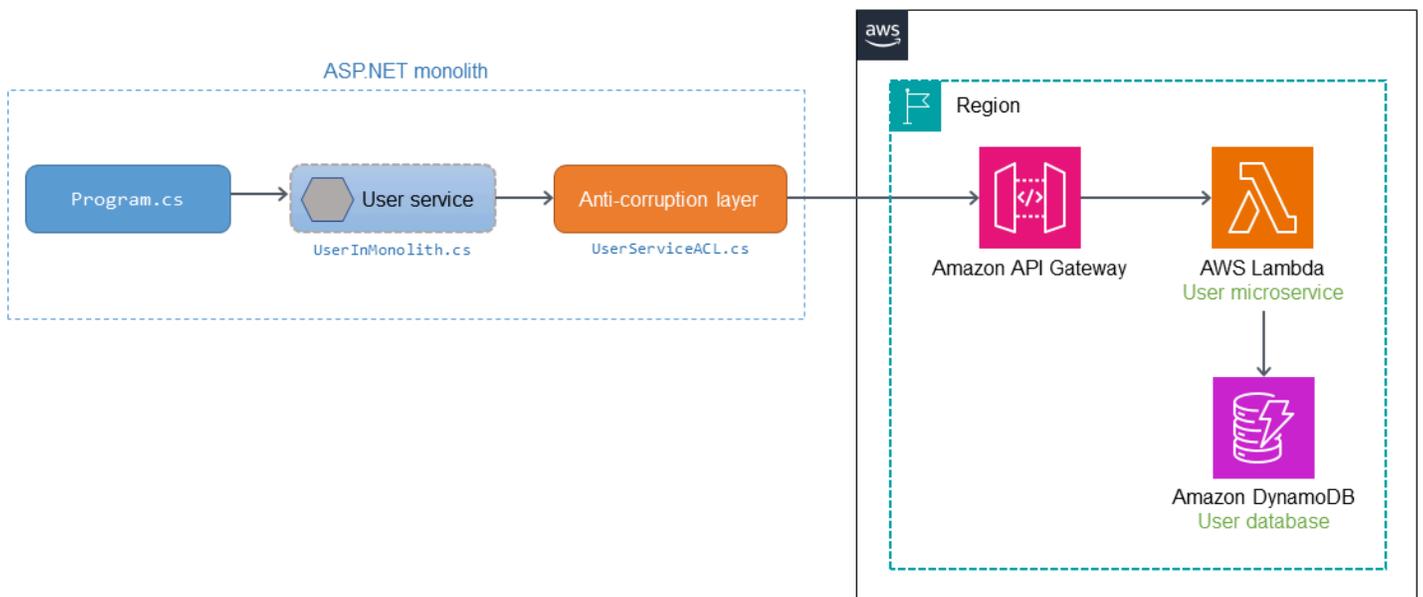
カートサービスが新しく移行したユーザーサービスを直接呼び出す必要がある場合は、カートサービスを変更し、モノリシックアプリケーションを徹底的にテストする必要があります。これにより、変革のリスクが高まり、ビジネスが中断する可能性があります。目標は、モノリシックアプリケーションの既存の機能への変更を最小限に抑えることです。

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で ACL を導入することをお勧めします。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またはファサードとして機能します。ACL はモノリシックアプリケーション内でクラスとして実装できます (例えば、UserServiceFacade または UserServiceAdapter) これは移行されたサービスに固有のもので、すべての依存サービスがマイクロサービスアーキテクチャに移行されたら、腐敗防止レイヤーを廃止する必要があります。



を使用した実装AWSサービス

次の図は、この ACL の例を次のように実装する方法を示しています。AWS サービス。



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され、としてデプロイされます [AWS Lambda](#) AWS での関数。Lambda 関数の呼び出しは次の方法でルーティングされます [アマゾン API ゲートウェイ](#)。ACL はモノリスにデプロイされ、呼び出しがユーザーのマイクロサービスのセマンティクスに適応するように変換されます。

いつProgram.csユーザーサービスを呼び出します (UserInMonolith.cs) モノリス内では、コールはACLにルーティングされます (UserServiceACL.cs)。ACL は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、API Gateway エンドポイントを介してマイクロサービスを呼び出します。発信者 (Program.cs) は、ユーザーサービスと ACL で行われる変換とルーティングを認識しません。呼び出し側はコードの変更に気付かないため、業務の中断が少なくなり、変革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実装を示していますUserServiceACL.cs。リクエストを受信すると、元のユーザーサービスが 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されたサービスのインターフェイスと一致するように変換し、サービスを呼び出し、呼び出し元に応答を返します。

```
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 倉庫

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、[を参照してください](https://github.com/aws-samples/anti-corruption-layer-pattern)。GitHub リポジトリ <https://github.com/aws-samples/anti-corruption-layer-pattern>。

関連情報

- [ストラングラーイチャジク柄](#)

- [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 (スクワッドやトライブなど) 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。

方法	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

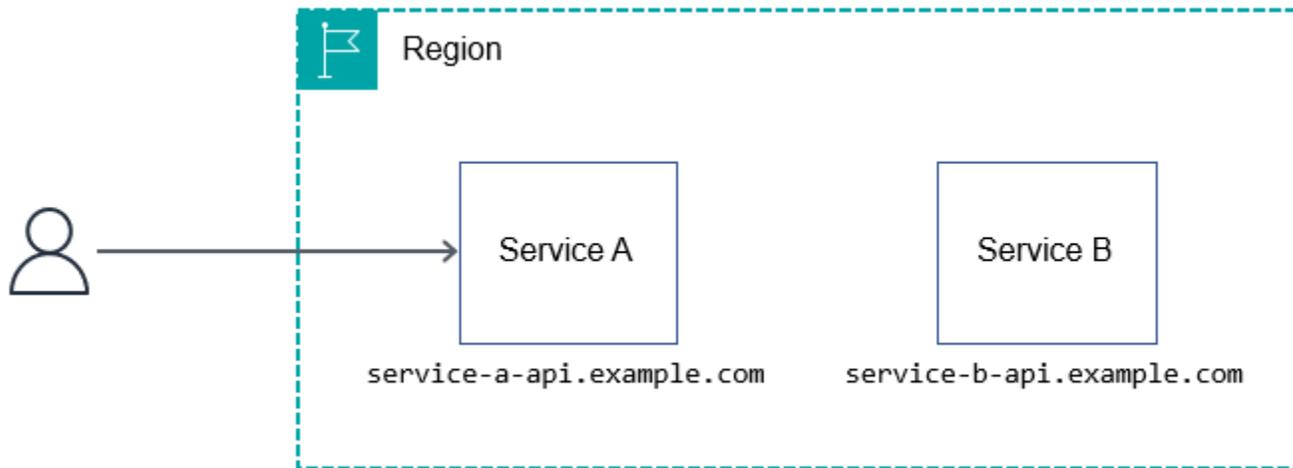
このセクションでは、これら 3 つのルーティング方式の一般的な使用例とそれぞれのトレードオフについて概説し、どの方法が要件と組織構造に最も適しているかを判断できるようにします。

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。



メリット

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。関連する任意のサービスを使用して AWS、この方法に従うアーキテクチャを構築できます。[Amazon API Gateway](#)、[Application Load Balancer](#)[AWS AppSync](#)、Amazon [Elastic Compute Cloud \(Amazon EC2\)](#)、またはその他の HTTP 準拠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、`region.service-a.api.example.com` や など、特定の AWS リージョン または バージョンのデプロイを分離、テスト、オーケストレーションすることが容易になります `dev.region.service-a.api.example.com`。

デメリット

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。

では AWS、API を共有し、正しいサービスに効果的にルーティングする方法が複数あります。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、Amazon の 3 つのアプローチについて説明します CloudFront。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GRESS) を作成することもできます。(このガイドでは Kubernetes のインGRESS 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;

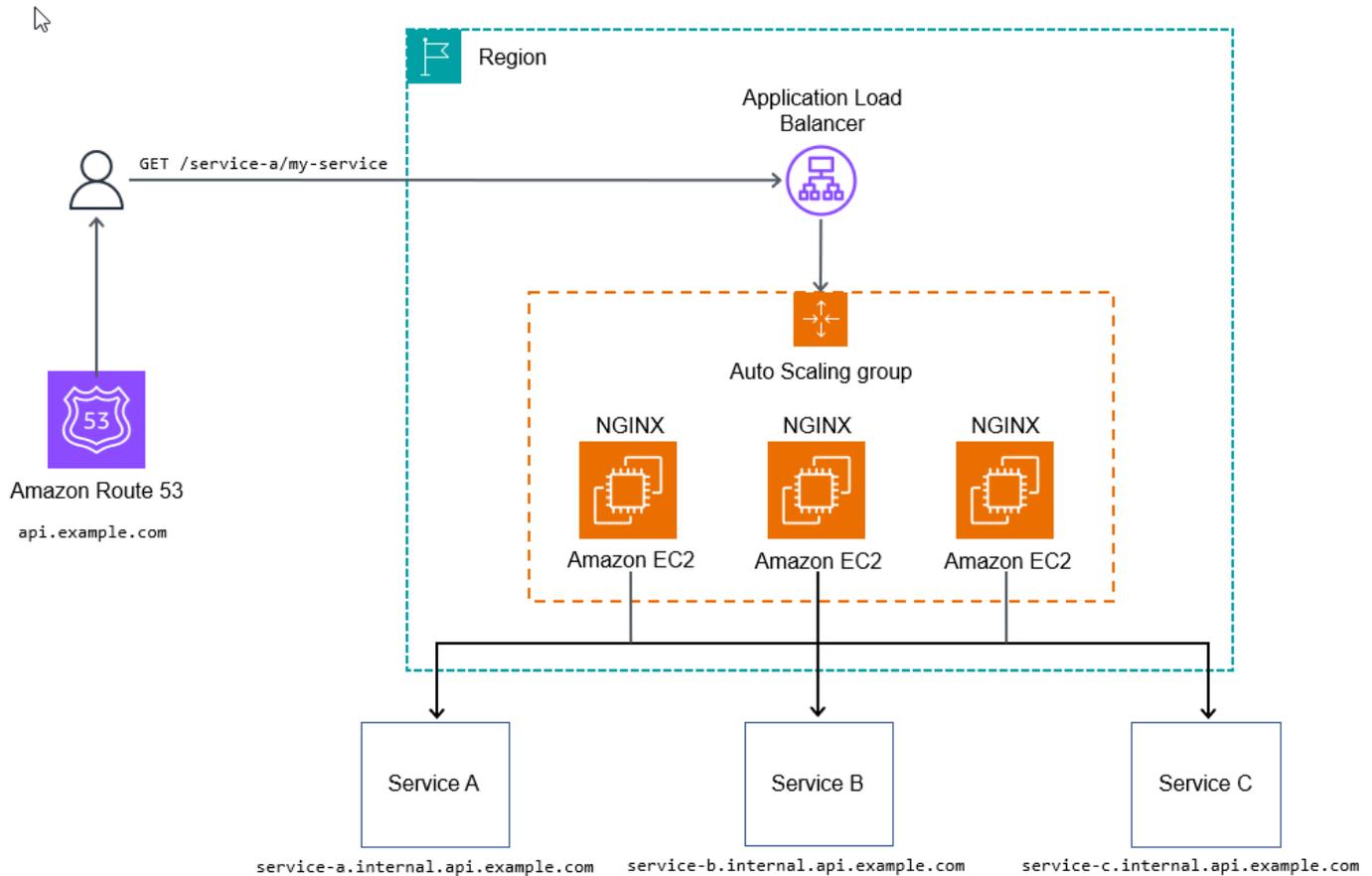
    location (^/[w-]+)/(.*) {
        proxy_pass $scheme://$1.internal.api.example.com/$2;
    }
}
```

```

}
}

```

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GRESS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。

メリット

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

ドを最小限に抑えながら、独自の APIs をデプロイおよび管理することもできます。[AWS X-Ray](#) やなどのトレース用の AWS マネージドサービスは [AWS WAF](#)、ここでも引き続き適用されます。

デメリット

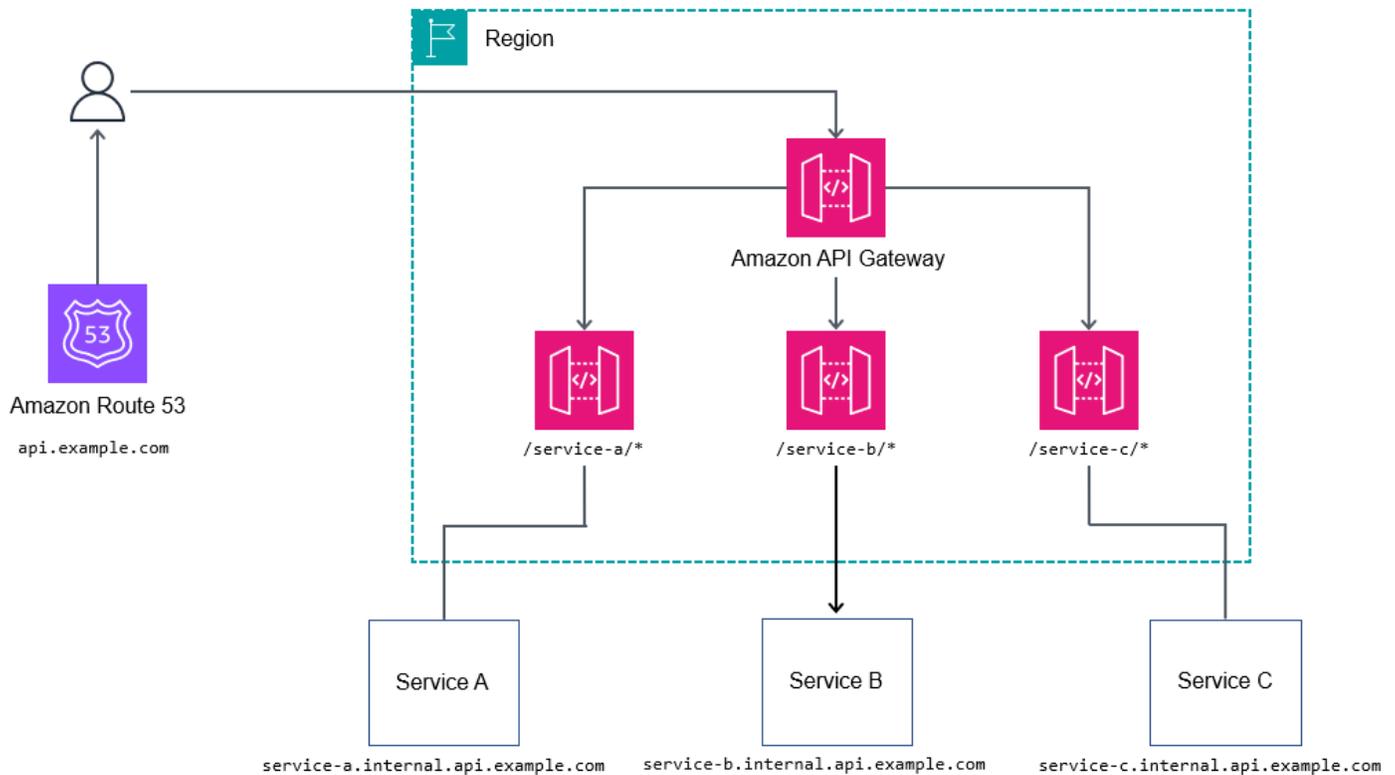
このアプローチの主な欠点は、必要なインフラストラクチャコンポーネントの広範なテストと管理が必要になることです。また、サイト信頼性エンジニアリング (SRE) チームが配置されていれば、これは問題にならない可能性があります。

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트리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。



メリット

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。

- [AWS Identity and Access Management \(IAM\)、Amazon Cognito、またはオーソライザーによる認証 N/Z Amazon Cognito AWS Lambda](#)
- [AWS X-Ray トレース用](#)
- [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)

デメリット

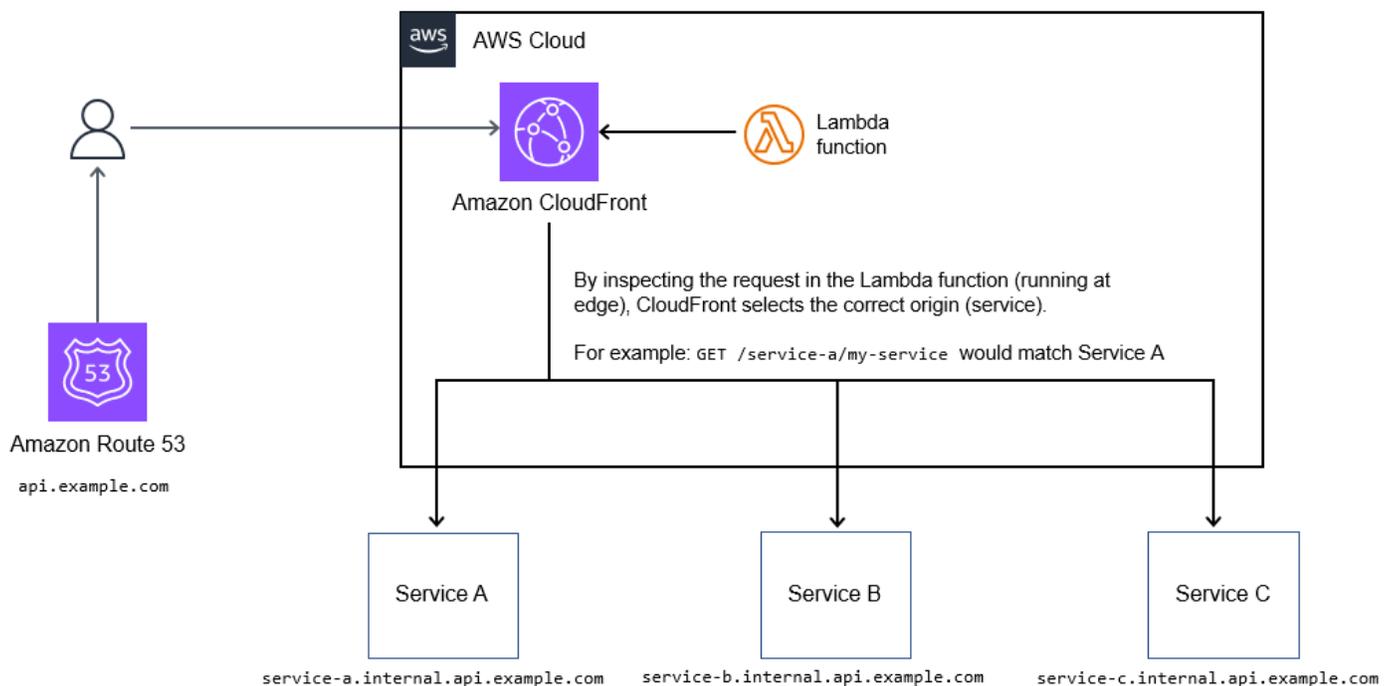
データ量が多いと、ユーザーによってはコストが問題になる場合があります。

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。



メリット

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、[フィールドレベルの暗号化](#)と、基本レート制限と基本 ACLs AWS WAF のためのとの統合 CloudFront をサポートしています。

デメリット

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。

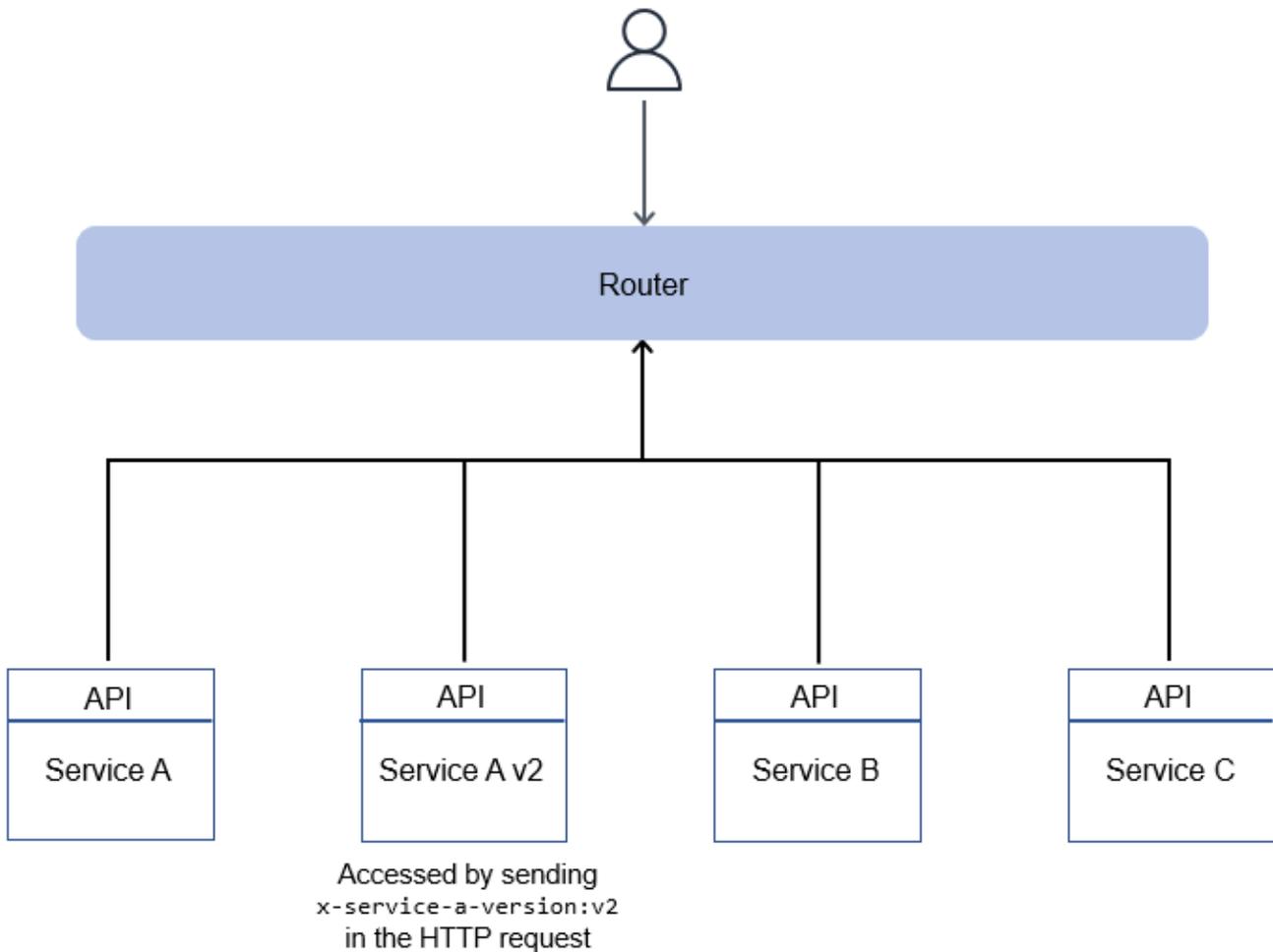
現在、Lambda@Edge 関数の更新には数分かかります。CloudFront また、すべての存在ポイントへの変更の伝達が完了するまでに最大 30 分かかります。そのため、更新が完了するまで、それ以降の更新は最終的にはブロックされます。

HTTP ヘッダルーティングパターン

ヘッダベースのルーティングでは、HTTP リクエストの HTTP ヘッダを指定することで、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 `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。



メリット

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。

デメリット

ホスト名ルーティング方式と同様に、HTTP ヘッダルーティングでは、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は、呼び出しが以前に繰り返しタイムアウトまたは失敗の原因になったときに、呼び出し元サービスが別のサービス (呼び出し元) への呼び出しを再試行するのを防ぐことができます。このパターンは、呼び出し元サービスが再び機能するタイミングを検出するために使用されます。

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する場合、1つ以上のサービスが使用できなくなったり、レイテンシーが長くなったりする可能性があります。複雑なアプリケーションがマイクロサービスを使用する場合、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可能性があります。マイクロサービスはリモートプロシージャ呼び出しを介して通信し、ネットワーク接続に一時的なエラーが発生して障害が発生する可能性があります。(一時的なエラーは、[バックオフパターンを使用した再試行](#)を使用して処理できます)。同期実行中にタイムアウトや障害がカスケードすると、ユーザーエクスペリエンスが低下する可能性があります。

ただし、状況によっては、障害の解決に時間がかかることがあります。例えば、発信者サービスがダウンしている場合や、データベースの競合によってタイムアウトが発生した場合などです。このような場合、呼び出し元のサービスが呼び出しを繰り返し再試行すると、これらの再試行によってネットワークの競合が発生し、データベーススレッドプールが消費される可能性があります。さらに、複数のユーザーがアプリケーションを繰り返し再試行すると、問題が悪化し、アプリケーション全体のパフォーマンスが低下する可能性があります。

サーキットブレーカーパターンは、マイケル・ナイガードによる本「リリース・イ」(2018年ナイガード)で人気がありました。この設計パターンにより、以前にタイムアウトや障害が繰り返し発生したサービス呼び出しを呼び出し元サービスが再試行するのを防ぐことができます。また、発信者サービスが再び機能するタイミングを検出することもできます。

サーキットブレーカーオブジェクトは、回路に異常がある場合に現在のものを自動的に中断する電気サーキットブレーカーのように機能します。障害発生時に、回路ブレーカーが現在のフローをシャットダウンまたは作動させます。同様に、サーキットブレーカーオブジェクトは発信者と受信者サービスの間で動作し、発信者が利用できない場合は作動します。

[分散コンピューティングのフォールカシー](#)は、Sun Microsystems で TAK Deutsch や他のによって行われた一連のアサーションです。分散アプリケーションを初めて使用するプログラマーは、常に

誤った仮定を行うと言います。ネットワークの信頼性、ゼロレイテンシーの想定、帯域幅の制限により、ネットワークエラーのエラー処理を最小限に抑えてソフトウェアアプリケーションが記述されず。

ネットワークの停止中、アプリケーションは応答を無期限に待機し、アプリケーションリソースを継続的に消費する可能性があります。ネットワークが使用可能になったときにオペレーションを再試行しないと、アプリケーションのパフォーマンスが低下する可能性があります。ネットワークの問題により、データベースまたは外部サービスへの API コールがタイムアウトした場合、サーキットブレーカーなしで呼び出しを繰り返すと、コストとパフォーマンスに影響する可能性があります。

適用対象

このパターンは次の場合に使用します。

- 発信者サービスは、失敗する可能性が最も高い呼び出しを行います。
- 発信者サービスが示すレイテンシーが高い場合 (例えば、データベース接続が遅い場合)、発信者サービスにタイムアウトが発生します。
- 発信者サービスは同期呼び出しを行います。発信者サービスは利用できないか、レイテンシーが長くなります。

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ため、マイクロサービスに依存しない API 主導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ことをお勧めします。
- 発信者による回路閉鎖：発信者がパフォーマンスの問題または障害から回復すると、回路のステータスを に更新できませんCLOSED。これはサーキットブレーカーパターンの拡張であり、目標復旧時間 (RTO) で必要な場合に実装できます。
- マルチスレッド呼び出し：有効期限タイムアウト値は、サービスの可用性を確認するために呼び出しが再度ルーティングされるまでに回路が作動する期間として定義されます。複数のスレッドで被着信者サービスが呼び出されると、最初に失敗した呼び出しによって有効期限のタイムアウト値が定義されます。実装では、後続の呼び出しによって有効期限が無限に移動しないようにする必要があります。
- 回路を強制的に開いたり閉じたりする：システム管理者は、回路を開いたり閉じたりする必要があります。これは、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実行できます。

- オブザーバビリティ：アプリケーションでは、サーキットブレーカーが開いているときに失敗する呼び出しを識別するためのログ記録を設定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、発信者が注文サービスで、発信者が支払いサービスです。

障害がない場合、次の図に示すように、注文サービスはすべての呼び出しをサーキットブレーカーによって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、障害を追跡できます。



Circuit breaker with payment service failure

タイムアウトが指定されたしきい値を超えると、アプリケーションは回路を開きます。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、即時の失敗が返されます。



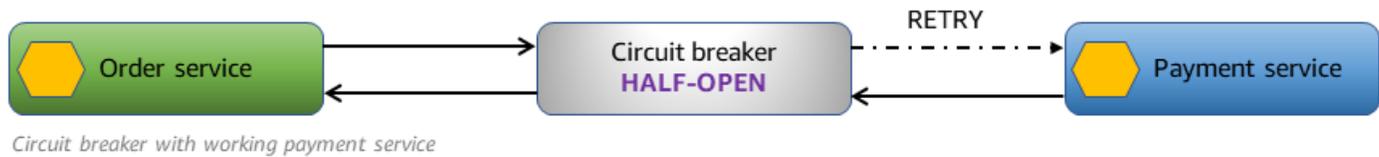
Circuit breaker stops routing to payment service

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ようとします。



Circuit breaker periodically retries payment service

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



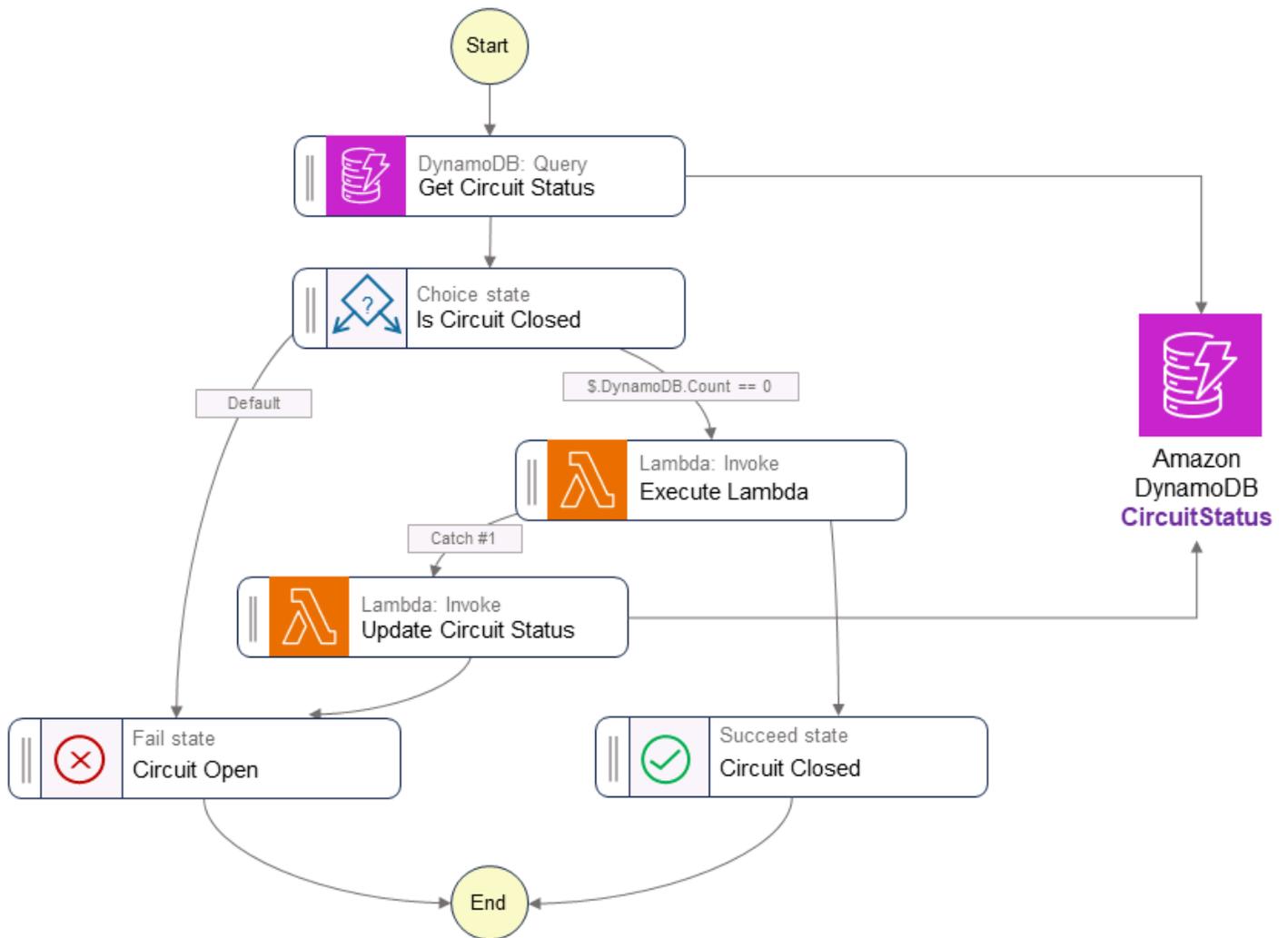
AWS のサービスを使用した実装

サンプルソリューションでは、の Express ワークフロー [AWS Step Functions](#) を使用して、サーキットブレーカーパターンを実装します。Step Functions ステートマシンでは、パターン実装に必要な再試行機能と決定ベースの制御フローを設定できます。

このソリューションでは、[Amazon DynamoDB](#) テーブルをデータストアとして使用して、回路の状態を追跡します。これは、パフォーマンスを向上させるために [Amazon ElastiCache for Redis](#) などのインメモリデータストアに置き換えることができます。

サービスが別のサービスを呼び出す場合、呼び出し元サービスの名前でワークフローを開始します。ワークフローは DynamoDB テーブルからサーキットブレーカーのステータスを取得します。DynamoDB CircuitStatus テーブルには、現在パフォーマンスが低下しているサービスが保存されます。に発信者の期限切れでないレコード CircuitStatus が含まれている場合、回路は開いています。Step Functions ワークフローは即時障害を返し、FAIL 状態で終了します。

CircuitStatus テーブルに発信者のレコードが含まれていない場合、または期限切れのレコードが含まれている場合、サービスは動作します。ステートマシン定義の ExecuteLambda ステップは、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。呼び出しが成功すると、Step Functions ワークフローは SUCCESS 状態で終了します。



サービスコールが失敗するか、タイムアウトが発生した場合、アプリケーションは定義された回数だけエクスポネンシャルバックオフで再試行します。再試行後にサービス呼び出しが失敗すると、ワークフローはを使用してサービスのCircuitStatusテーブルにレコードを挿入しExpiryTimeStamp、ワークフローは FAIL状態で終了します。同じサービスへの後続の呼び出しは、サーキットブレーカーが開いている限り、即時の障害を返します。ステートマシン定義のGet Circuit Statusステップは、ExpiryTimeStamp値に基づいてサービスの可用性をチェックします。期限切れの項目は、DynamoDBの有効期限(TTL)機能を使用して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、GetCircuitStatusLambda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

- [ストラングラー・フィグ パターン](#)
- [バックオフパターンで再試行](#)
- [AWS App Mesh サーキットブレーカー機能](#)

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、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。

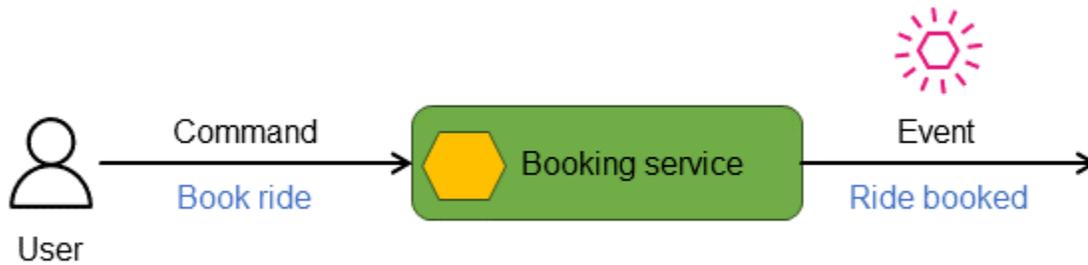
実装

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。



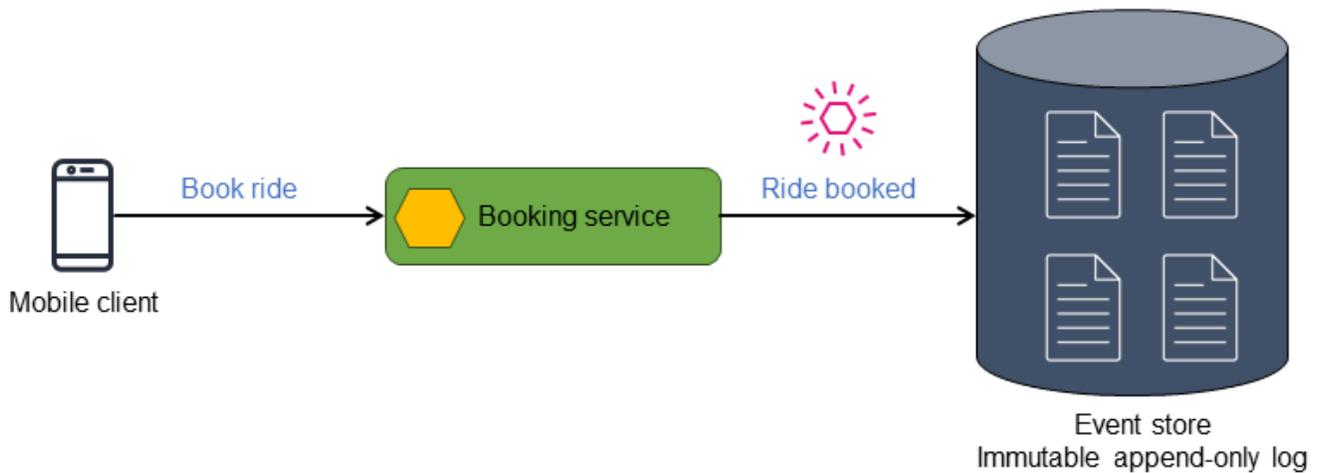
イベントストア

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。

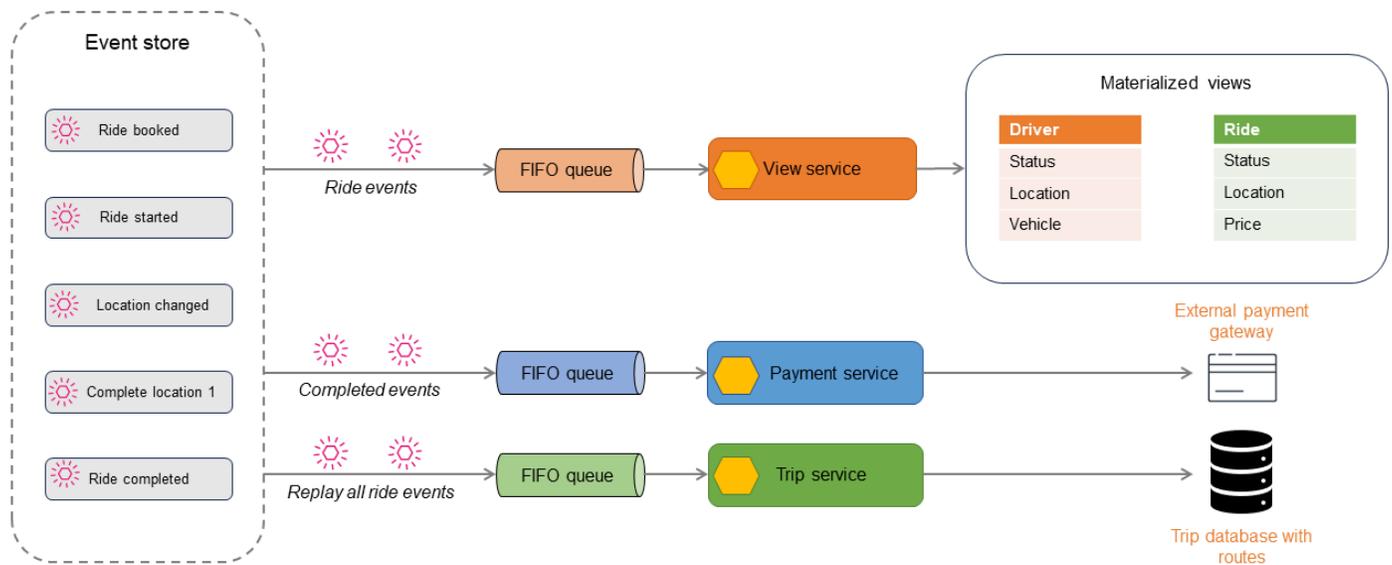
- 関連イベントを集計する。関連するイベントオブジェクトが結合されて、クエリ用の現在の状態が生成されます。この方法は、イベントが結合されて読み取り専用のデータストアに書き込まれるという点で、多くの場合 CQRS パターンと組み合わせて使用されます。
- マテリアライズドビューを使用する。マテリアライズドビューパターンで実装されたイベントソーシングを使用して、イベントデータを計算または要約すると、関連データの現在の状態を取得できます。
- イベントを再生する。イベントオブジェクトを再生すると、現在の状態を生成するアクションを実行できます。

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。



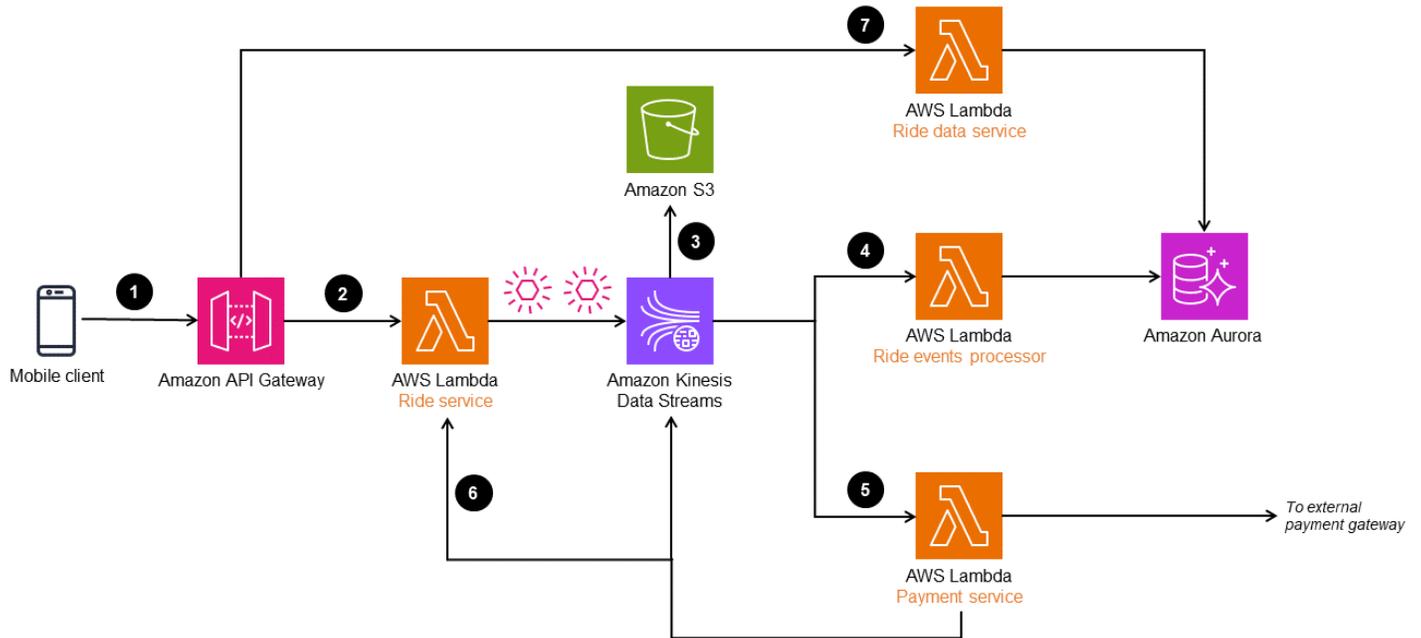
乗車のイベントで、CQRS またはマテリアライズドビューパターンを使用して読み取り専用のデータストアを生成できます。読み込まれたストアにクエリを実行すると、乗車、ドライバー、または予約の現在の状態を取得できます。Location changed や Ride completed などの一部のイベントは、支払い処理のために別のコンシューマーに公開されます。乗車が完了すると、すべての乗車イベントが再生され、監査または報告の目的で乗車の履歴が作成されます。

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

ポートとアダプターのパターンとも呼ばれる六角形アーキテクチャパターンは、2005年に Alistair Cockburn によって提案されました。データストアやユーザーインターフェイス (UIs) に依存せずに、アプリケーションコンポーネントを個別にテストできる疎結合アーキテクチャを作成することを目的としています。このパターンは、データストアと UIs のテクノロジーロックインを防ぐのに役立ちます。これにより、ビジネスロジックへの影響が限定的またはまったくなく、時間の経過とともにテクノロジースタックを簡単に変更できます。この疎結合アーキテクチャでは、アプリケーションはポートと呼ばれるインターフェイスを介して外部コンポーネントと通信し、アダプターを使用してこれらのコンポーネントとの技術的な交換を変換します。

導入する理由

六角形アーキテクチャパターンは、データベースや外部 APIs を分離するために使用されます。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 関数用に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、ストアドプロシージャとしてデータベースレイヤーにビジネスロジックを埋め込み、ユーザーインターフェイスにビジネスロジックを埋め込むことが一般的です。この手法は、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とともに、データベース移行とユーザーエクスペリエンス (UX) のモダナイゼーションの取り組みにボトルネックを引き起こすアーキテクチャを密接に結び付けます。六角形アーキテクチャパターンを使用すると、テクノロジー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ます。この戦略により、データベース、UX、サービスコンポーネントなどのアプリケーションコンポーネントを簡単に交換できます。

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ります。
- 複数のタイプのクライアントが同じドメインロジックを使用できます。
- UI とデータベースコンポーネントには、アプリケーションロジックに影響を与えない定期的なテクノロジー更新が必要です。

- アプリケーションには複数の入力プロバイダーと出力コンシューマーが必要であり、アプリケーションロジックをカスタマイズすると、コードが複雑になり、拡張性が損なわれます。

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、ドメイン駆動型設計 (DDD) で特にうまく機能します。各アプリケーションコンポーネントは DDD のサブドメインを表し、六角形アーキテクチャを使用して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**：設計上、六角形アーキテクチャは入力と出力に抽象化を使用します。したがって、本質的に疎結合であるため、単体テストの作成と単体テストが容易になります。
- **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複雑さは、慎重に処理すると、俊敏性、テストカバレッジ、テクノロジーの適応性などの大きな利点をもたらす可能性があります。そうしないと、解決が複雑になる可能性があります。
- **メンテナンスオーバーヘッド**：アーキテクチャをプラグブルにする追加のアダプターコードは、アプリケーションコンポーネントが複数の入力ソースと出力先への書き込みを必要とする場合、または入力と出力のデータストアが時間の経過とともに変更される場合にのみ正当化されます。そうしないと、アダプターはメンテナンスする別の追加レイヤーになり、メンテナンス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別のレイヤーが追加され、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は、インフラストラクチャコードと、アプリケーションを UIs、外部 APIs、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のアプリケーションとビジネスロジックの分離をサポートします。ポートやアダプターを使用して、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) に簡単に接続できます。

ポートは、アプリケーションコンポーネントへのテクノロジーに依存しないエントリポイントです。これらのカスタムインターフェイスは、誰が、または何がインターフェイスを実装するかに関係なく、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できるようにするインターフェイスを決定します。これは、USB ポートが USB アダプターを使用している限り、さまざまなタイプのデバイスがコンピュータと通信できるようにする方法に似ています。

アダプターは、特定のテクノロジーを使用してポートを介してアプリケーションとやり取りします。アダプターはこれらのポートに接続し、 からデータを受信したり、ポートにデータを提供したり

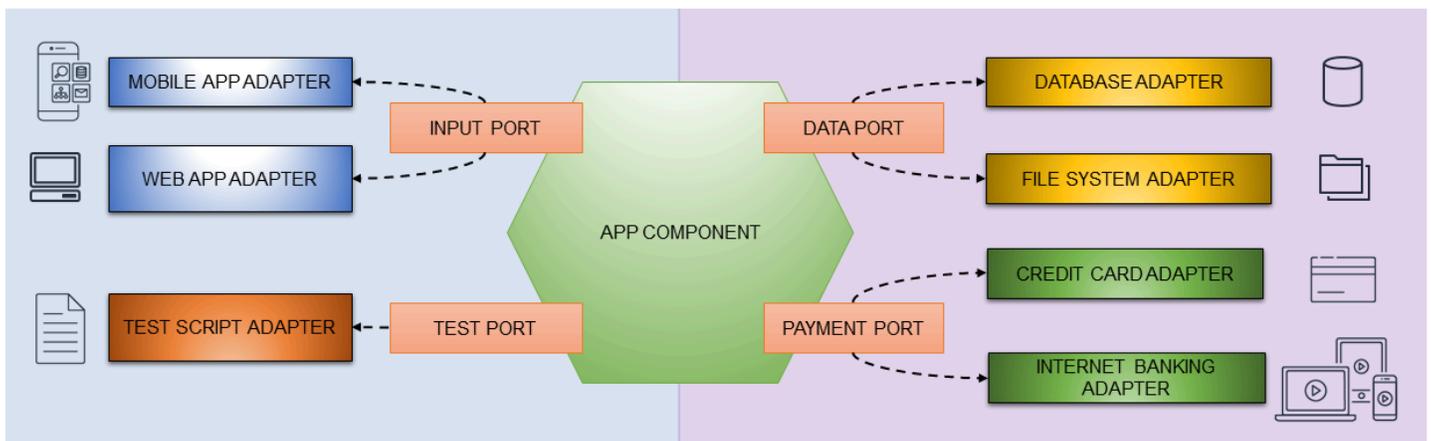
して、データを変換してさらに処理します。例えば、REST アダプターを使用すると、アクターは REST API を介してアプリケーションコンポーネントと通信できます。ポートには、ポートやアプリケーションコンポーネントにリスクを与えることなく、複数のアダプターを含めることができます。前の例を拡張するために、同じポートに GraphQL アダプターを追加すると、アクターが REST API、ポート、またはアプリケーションに影響を与えずに GraphQL API を介してアプリケーションとやり取りするための追加の手段が提供されます。

ポートはアプリケーションに接続し、アダプターは外部への接続として機能します。ポートを使用して、疎結合のアプリケーションコンポーネントを作成し、アダプターを変更して依存コンポーネントを交換できます。これにより、アプリケーションコンポーネントはコンテキストに応じた認識を必要とせずに、外部入出力とやり取りできます。コンポーネントはどのレベルでも交換できるため、自動テストが容易になります。テストを実行するために環境全体をプロビジョニングするのではなく、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ます。アプリケーションロジックは外部要因に依存しないため、テストが簡素化され、依存関係を模倣することが容易になります。

例えば、疎結合アーキテクチャでは、アプリケーションコンポーネントはデータストアの詳細を知らなくてもデータの読み書きを行える必要があります。アプリケーションコンポーネントの責任は、インターフェイス (ポート) にデータを提供することです。アダプターは、データストアへの書き込みロジックを定義します。データストアは、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です。

高レベルのアーキテクチャ

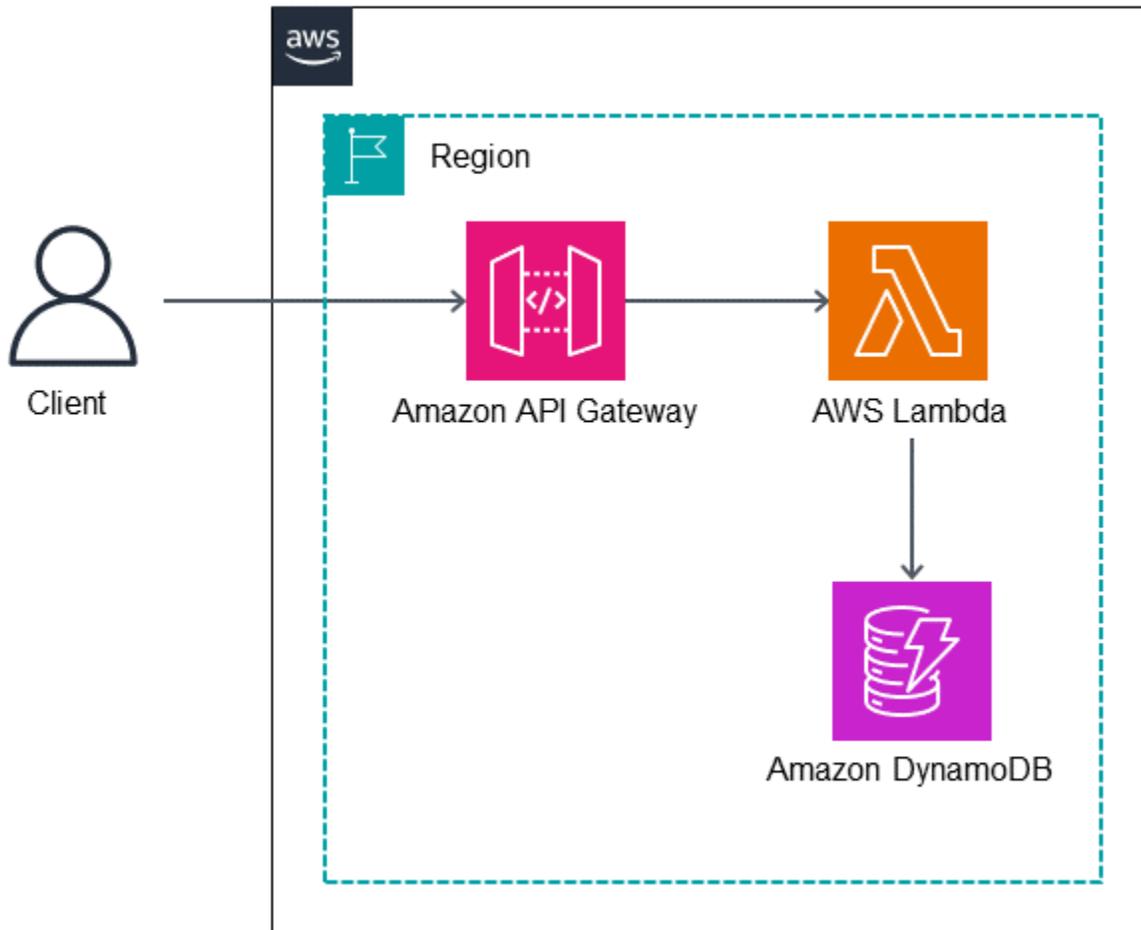
アプリケーションまたはアプリケーションコンポーネントには、コアビジネスロジックが含まれています。次の図に示すように、ポートからコマンドまたはクエリを受信し、ポートを介して外部アクターにリクエストを送信します。外部アクターはアダプターを介して実装されます。



を使用した実装 AWS のサービス

AWS Lambda 関数には、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いることが多く、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して、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。この分離により、データベースコードに依存すること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 関数が六角形アーキテクチャパターンを実装します。Lambda 関数は、Amazon API Gateway REST API によって開始されます。この関数はビジネスロジックを実装し、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は、Lambda を使用してドメインモデルを実装し、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、関数のユニットテストを実装する方法を示しています。

ドメインモデル

ドメインモデルクラスには、外部コンポーネントや依存関係に関する知識はなく、ビジネスロジックを実装するだけです。次の例では、クラスRecipientは、予約日の重複をチェックするドメインモデルクラスです。

```
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 クラスは受信者クラスに接続し、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

    def make_reservation(self, recipient_id:str, slot_id:str) -> Status:
        status = None
```

```
recipient = self.__recipient_output_port.get_recipient_by_id(recipient_id)
slot = self.__slot_output_port.get_slot_by_id(slot_id)
.....

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

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` クラスのインスタンスのファクトリです。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

モッククラスを挿入することで、ドメインモデルクラスのビジネスロジックをテストできます。次の例では、ドメインモデル `Recipient` クラスのユニットテストを示します。

```
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

- [「六角形アーキテクチャ」](#)、Alistair Cockburn による記事
- [を使用した進化的アーキテクチャの開発 AWS Lambda](#) (日本語ブログAWS 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の実装における六角形アーキテクチャの使用について説明します。

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ン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。

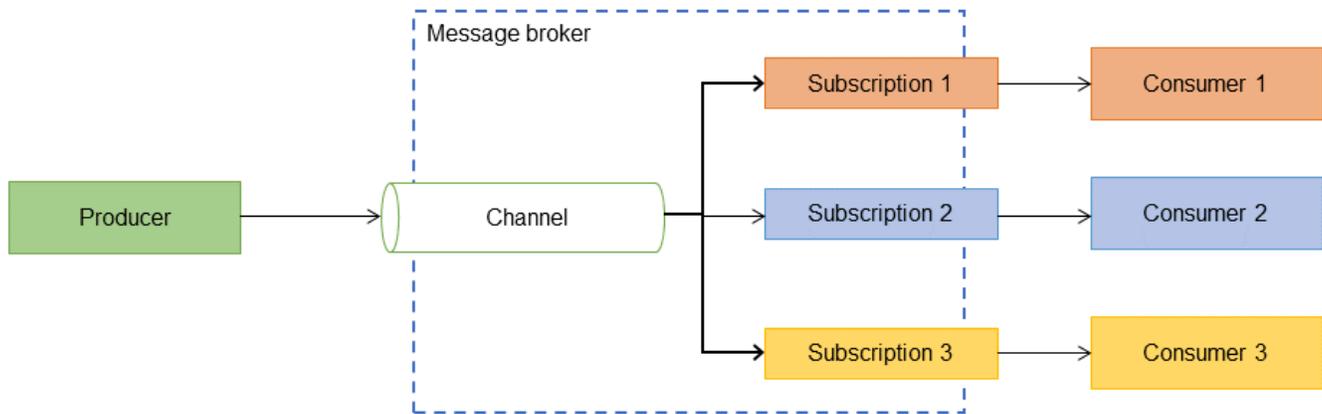
- **メッセージ再生:** メッセージの再生機能は、メッセージングインフラストラクチャにより異なります。また、ユースケースに応じ、カスタム実装を提供することもできます。
- **デッドレターキュー:** 郵便システムでは、配達不能な郵便物を処理するための施設である、デッドレターオフィスがあります。[pub/sub のメッセージング](#)においては、サブスクライブ済みのエンドポイントに配信できないメッセージのためのキューとして、デッドレターキューが存在します。

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、メッセージング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ンネルを、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。



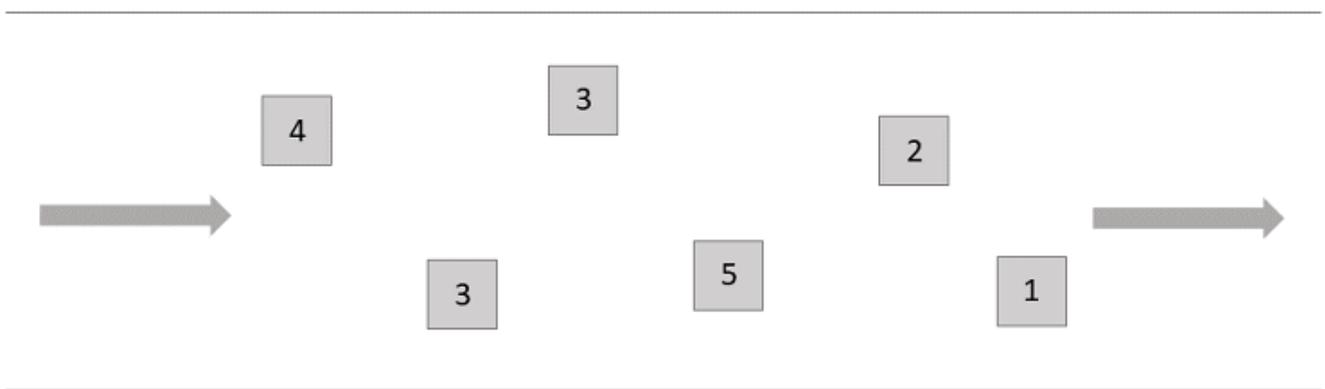
AWS のサービスを使用した実装

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。

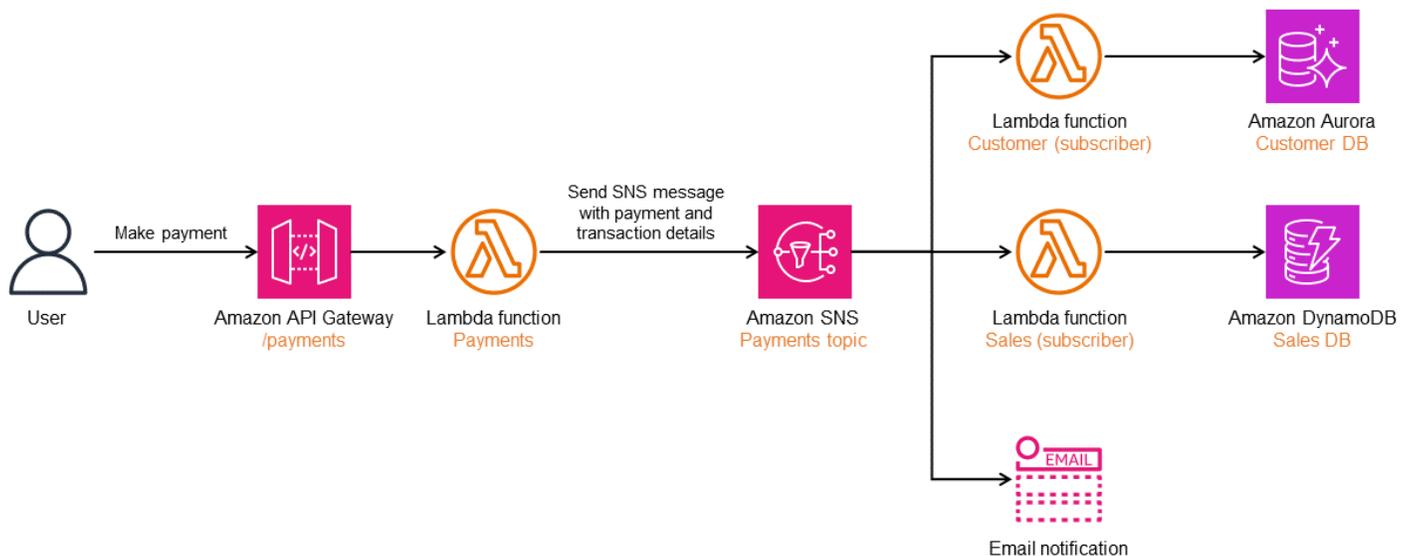
- 標準トピックでは、1 秒あたりのメッセージ数を上限なくサポートし、また順序付けと重複排除はベストエフォートになります。



- 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。



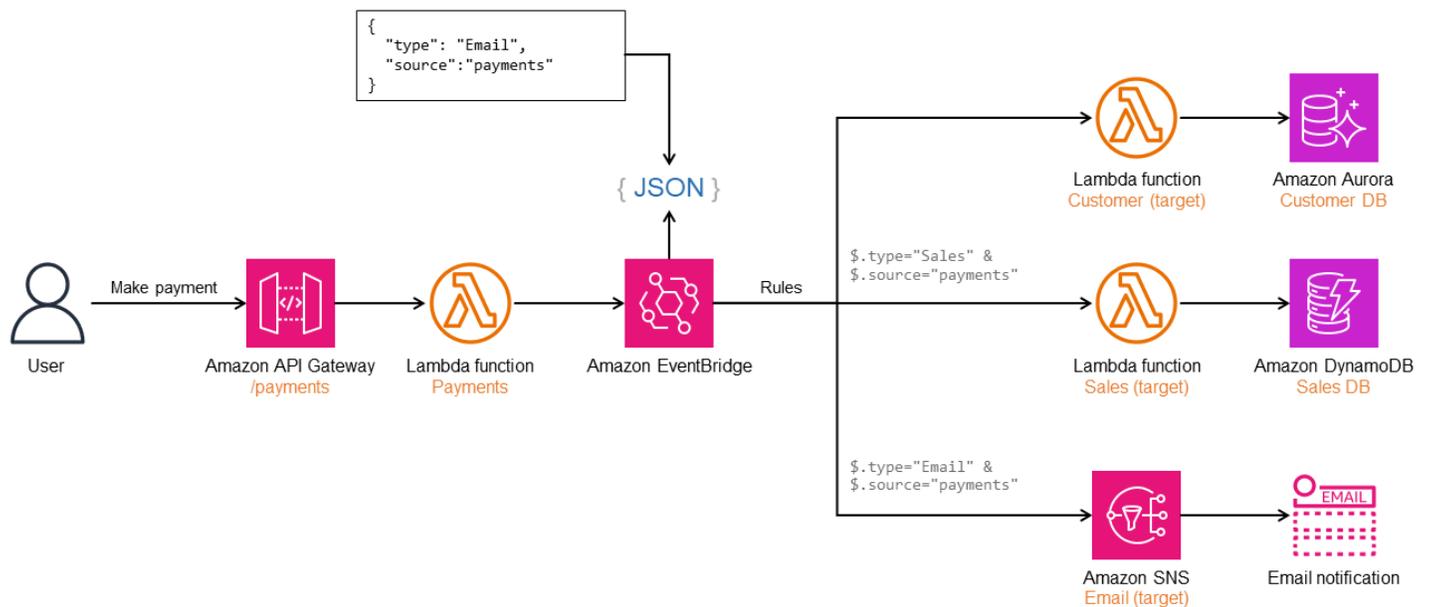
次の図は、Amazon SNS を使用したパブリッシュ – サブスクライブパターンの実装方法を示しています。ユーザーが支払いを行うと、SNS メッセージが Payments Lambda 関数により Payments SNS トピックに送信されます。この SNS トピックには 3 人のサブスクライバーがいます。各サブスクライバーがメッセージのコピーを受信して、その処理を行います。



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

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パターンでは、一時的なエラーが原因で失敗した操作を透過的に再試行することで、アプリケーションの安定性が向上します。

モチベーション

分散型アーキテクチャでは、一時的なエラーは、サービスの調整、ネットワーク接続の一時的な喪失、または一時的なサービスの利用不能が原因である可能性があります。これらの一時的なエラーが原因で失敗した操作を自動的に再試行することで、ユーザーエクスペリエンスとアプリケーションの回復力が向上します。ただし、頻繁に再試行すると、ネットワーク帯域幅が過負荷になり、競合が発生する可能性があります。指数バックオフとは、指定した回数だけ待機時間を増やして操作を再試行する手法です。

適用性

次の場合は、バックオフによる再試行パターンを使用してください。

- サービスでは、過負荷を防ぐためにリクエストを頻繁に調整しているため、429 リクエストが多すぎます呼び出しプロセスの例外。
- ネットワークは分散型アーキテクチャでは目に見えない存在であり、一時的なネットワークの問題によって障害が発生します。
- 呼び出されているサービスが一時的に利用できなくなり、障害が発生しています。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、頻繁に再試行するとサービスが低下する可能性があります。

問題と考慮事項

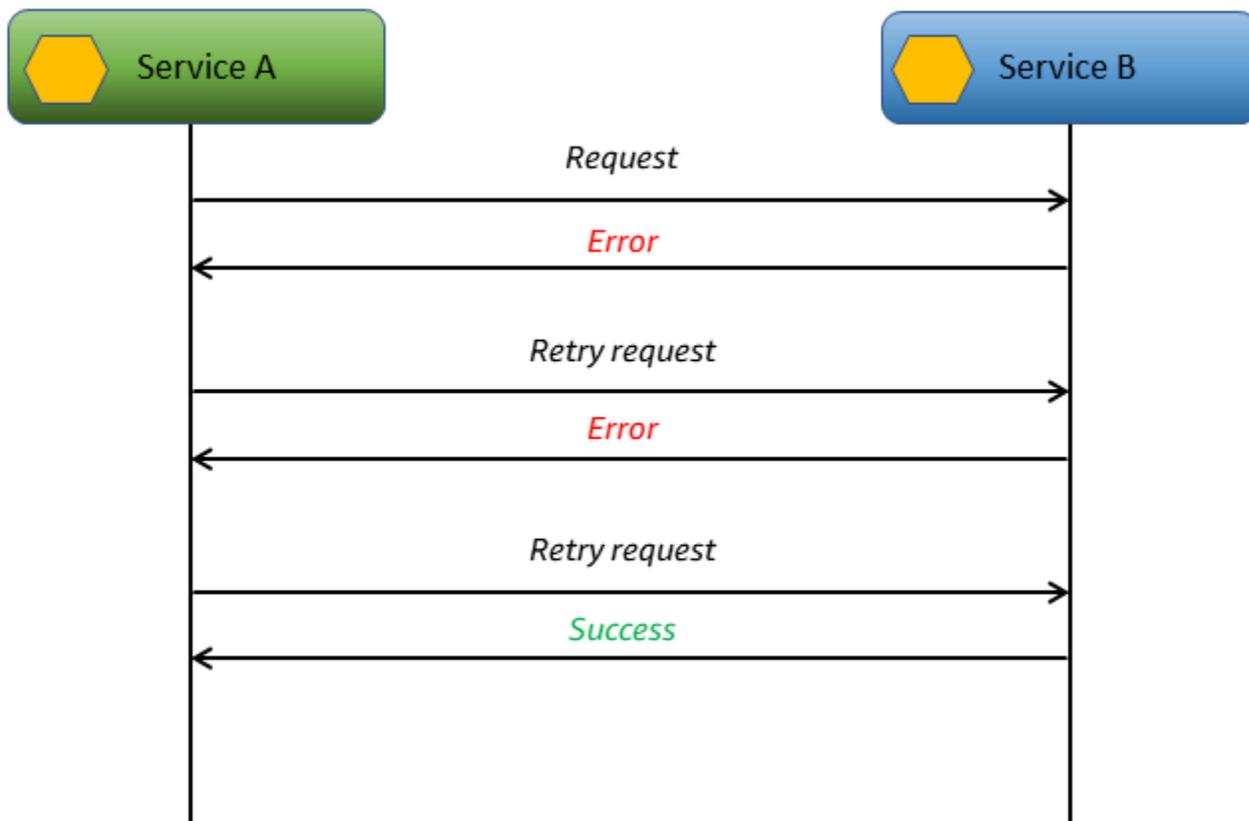
- イデムポテンス: メソッドへの複数の呼び出しがシステム状態に対する 1 回の呼び出しと同じ効果をもたらす場合、その操作は無能と見なされます。バックオフ付きの再試行パターンを使用する場合、操作は無駄になるはずですが、そうしないと、部分的な更新によってシステム状態が壊れる可能性があります。

- ネットワーク帯域幅: ネットワーク帯域幅を占有する再試行回数が多すぎると、サービスのパフォーマンスが低下し、応答時間が遅くなる可能性があります。
- フェイルファストシナリオ: 非一時的なエラーの場合、障害の原因を特定できれば、サーキットブレーカーパターンを使用して迅速に障害が発生した方が効率的です。
- バックオフ率: 指数関数的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び、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、正常に応答が返されるまで、サービス A がサービス B への呼び出しを再試行する方法を示しています。サービス B が数回試行しても成功応答を返さない場合、サービス A は再試行を中止して、呼び出し元に失敗を返すことができます。

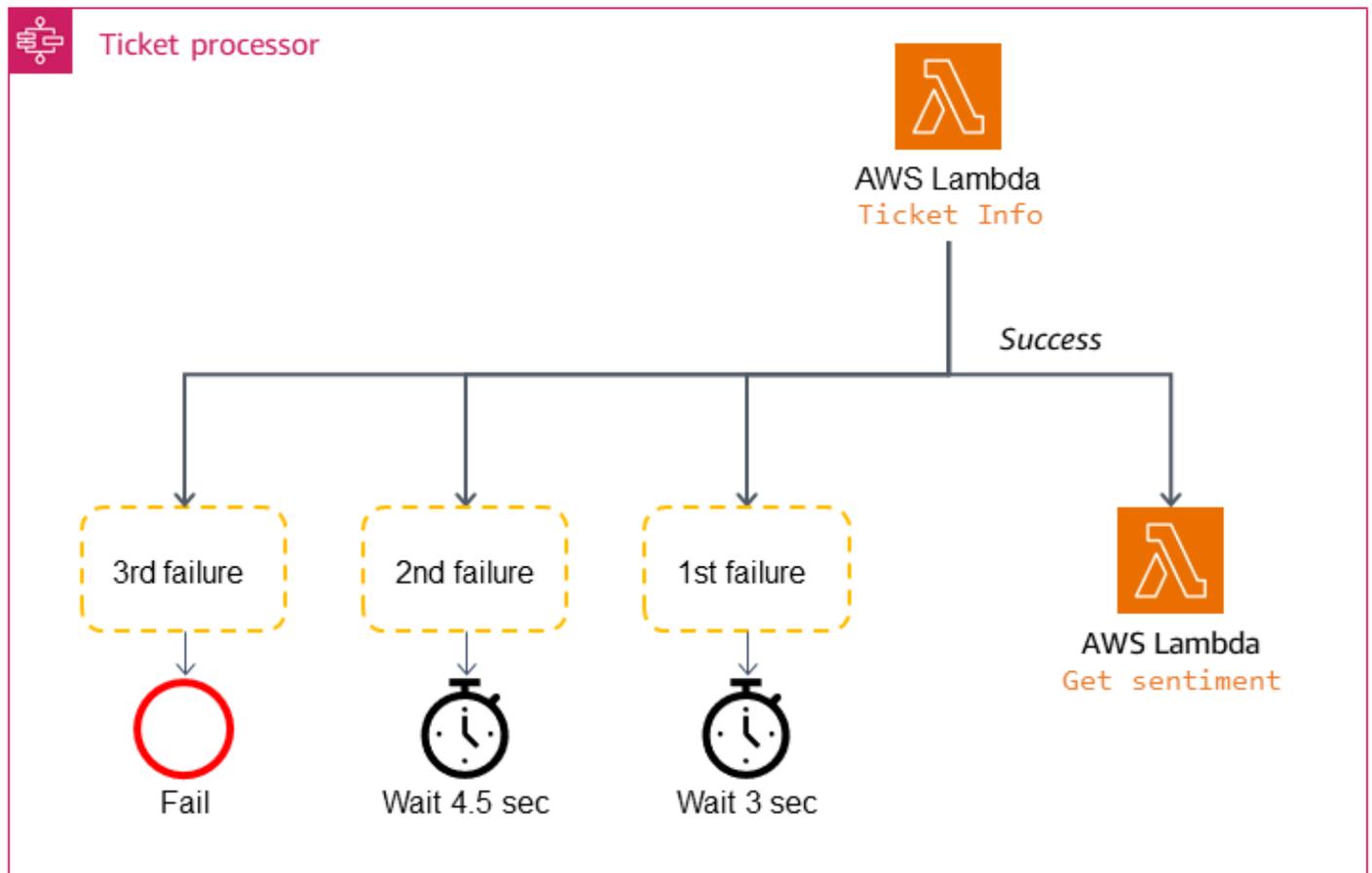


を使用した実装AWSサービス

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。不満を抱いている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ることによって迅速に処理されます。ザ・ Ticket infoLambda 関数はチケットの詳細を抽出し、Get sentimentラムダ関数。ザ・ Get sentimentLambda 関数は、説明をに渡して顧客の感情をチェックします。[アマゾン・コンプライエント](#)(表示されていません)。

への電話の場合Get sentimentLambda 関数が失敗すると、ワークフローは操作を 3 回再試行します。AWS Step Functionsバックオフ値を設定できるようにすることで、指数関数的バックオフが可能になります。

この例では、1.5 秒の増加倍率で最大 3 回の再試行が設定されています。最初の再試行が 3 秒後に行われた場合、2 回目の再試行は 3×1.5 秒 = 4.5 秒後に、3 回目の再試行は 4.5×1.5 秒 = 6.75 秒後に行われます。3 回目の再試行が失敗した場合、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必要ありません。次の方法で設定として提供されます。AWS Step Functions。



サンプルコード

次のコードは、バックオフパターンによる再試行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 occurred, so stop calling the API
            default:
                retry = false;
                break;
        }
        retries++;
    } while (retry && retries < MAX_RETRIES);
}
```

GitHub 倉庫

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、[を参照してください](https://github.com/aws-samples/retry-with-backoff)。GitHub リポジトリ <https://github.com/aws-samples/retry-with-backoff>。

関連情報

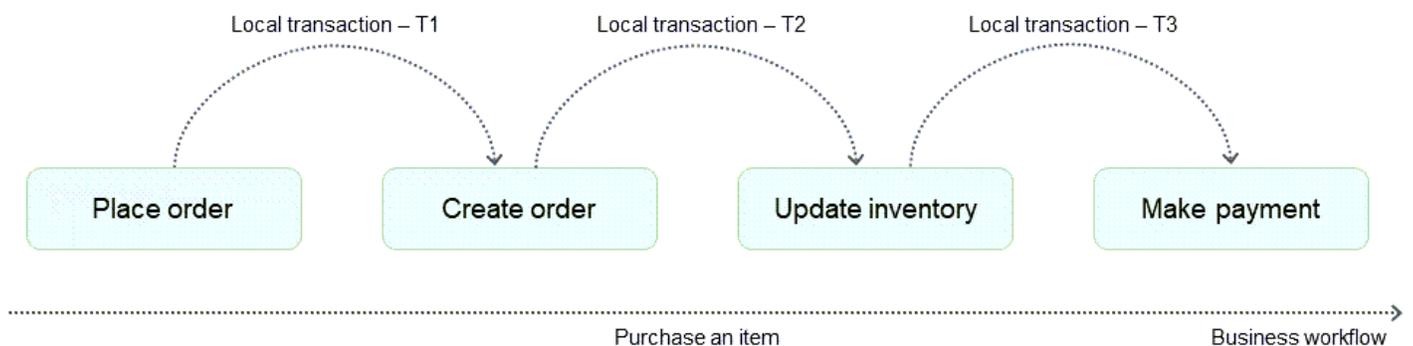
- [タイムアウト、リトライ、ジッターを伴うバックオフ\(アマゾンビルダーズライブラリー\)](#)

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。



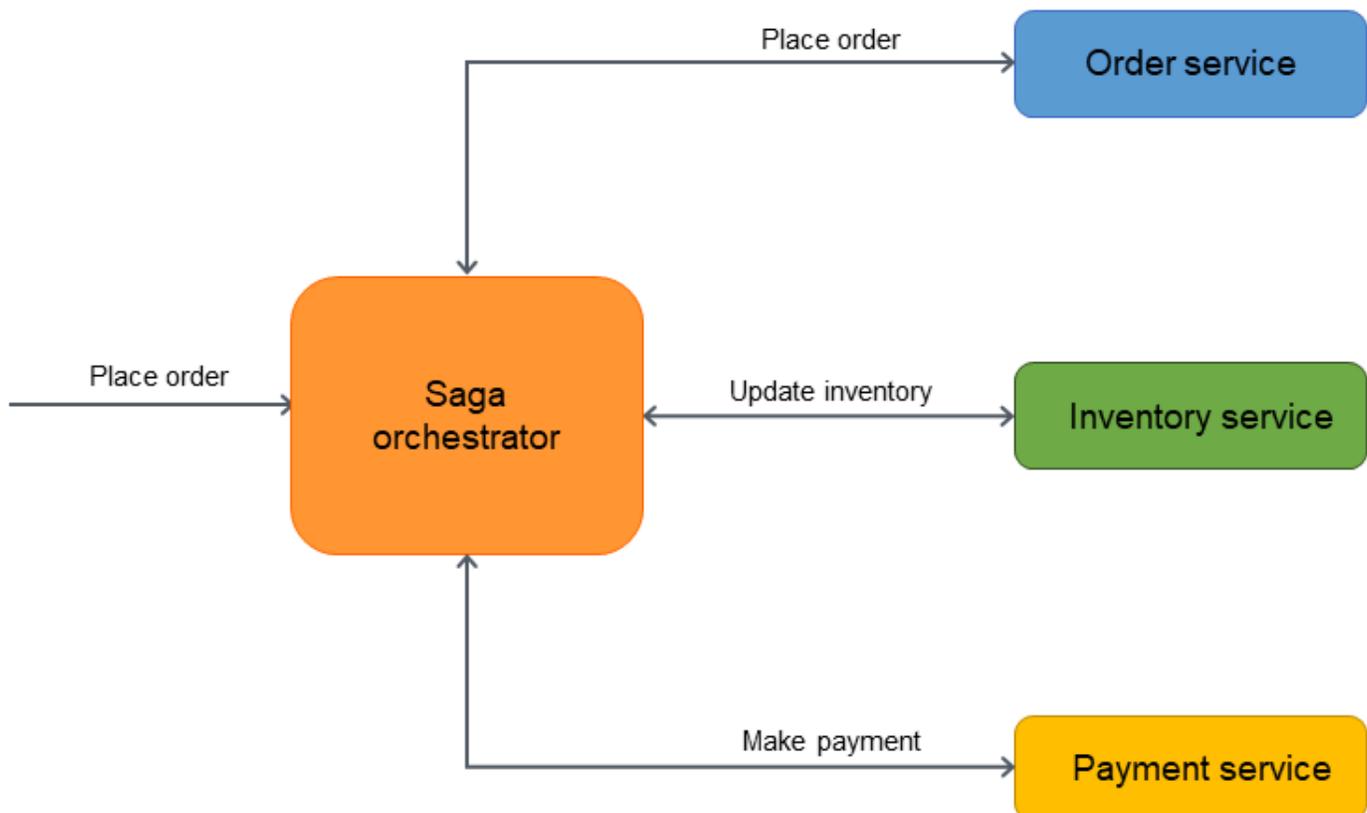
詳細なレビューについては、このガイドの「[Saga コレオグラフィ](#)」セクションを参照してください。

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

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オプションではありません。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ベースの 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。

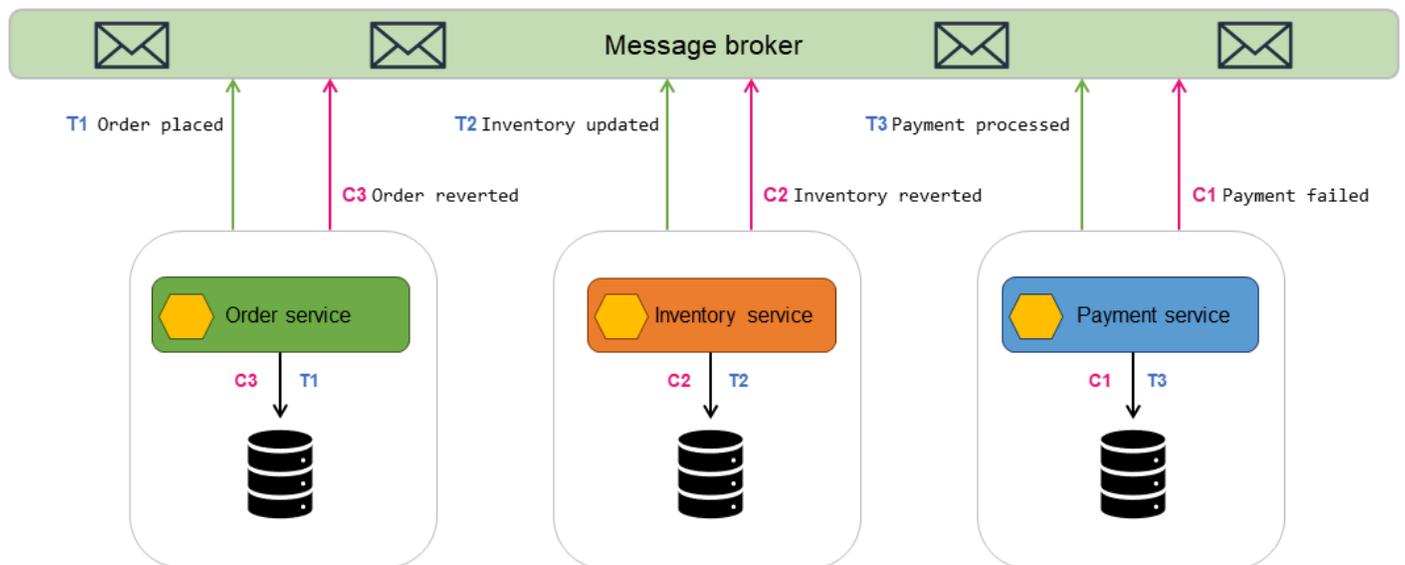
ります。このようなシナリオに対処するために、セマンティックロックを使用することをお勧めします。

- **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較して、End-to-end のモニタリングとレポート作成は困難です。
- **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。

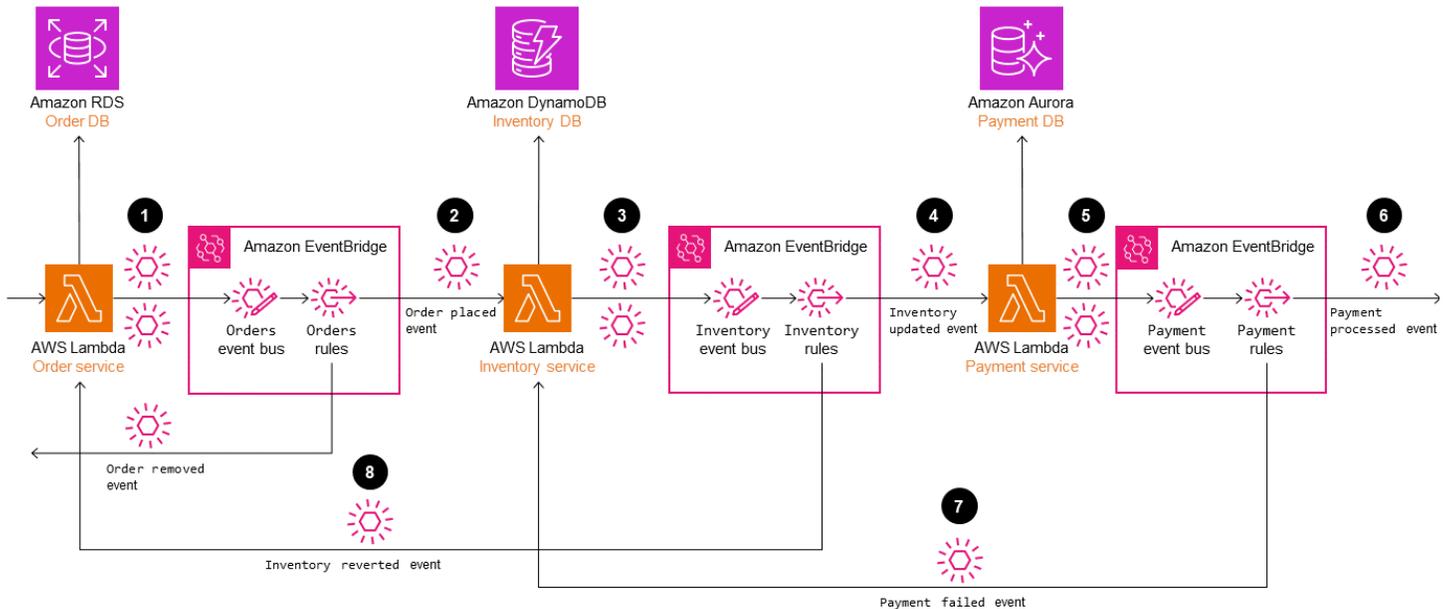
- 在庫サービスはローカルトランザクション T2 を実行します。このトランザクションはデータベースをアトミックに更新し、Inventory updated メッセージをメッセージブローカーに発行します。
- 支払いサービスは在庫サービスからのメッセージをサブスクライブし、在庫が更新されたというメッセージを受信します。
- 支払いサービスはローカルトランザクション T3 を実行します。このトランザクションはデータベースに支払いの詳細を付け加えてアトミックに更新し、Payment processed メッセージをメッセージブローカーに発行します。
- 支払いが失敗した場合、支払いサービスは補償トランザクション 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

Amazon を使用して Saga コレオグラフィパターンを実装できます EventBridge。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。
- イベント EventBridge バス、Orders イベントバス、Inventory イベントバスの 3 つのカスタム Payment バスがあります。
- 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。

または、順序処理に問題がある場合、EventBridge ルールはデータの整合性と整合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オプションではありません。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ベースの 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。

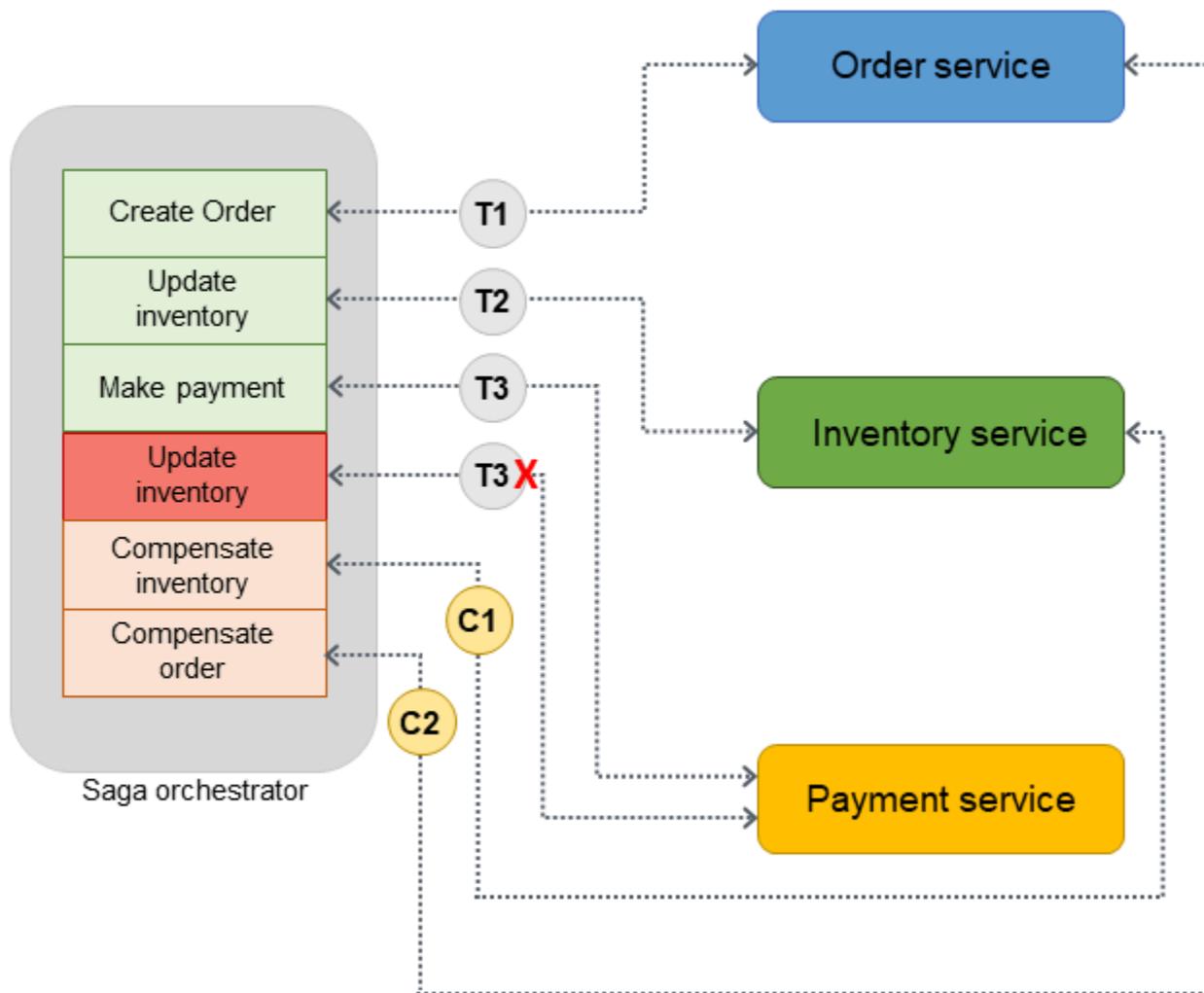
問題点と考慮事項

- 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重要です。
- 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 オークストレーターは、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 オーク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、オーク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 オークストレーションを実装できます。

AWS の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。

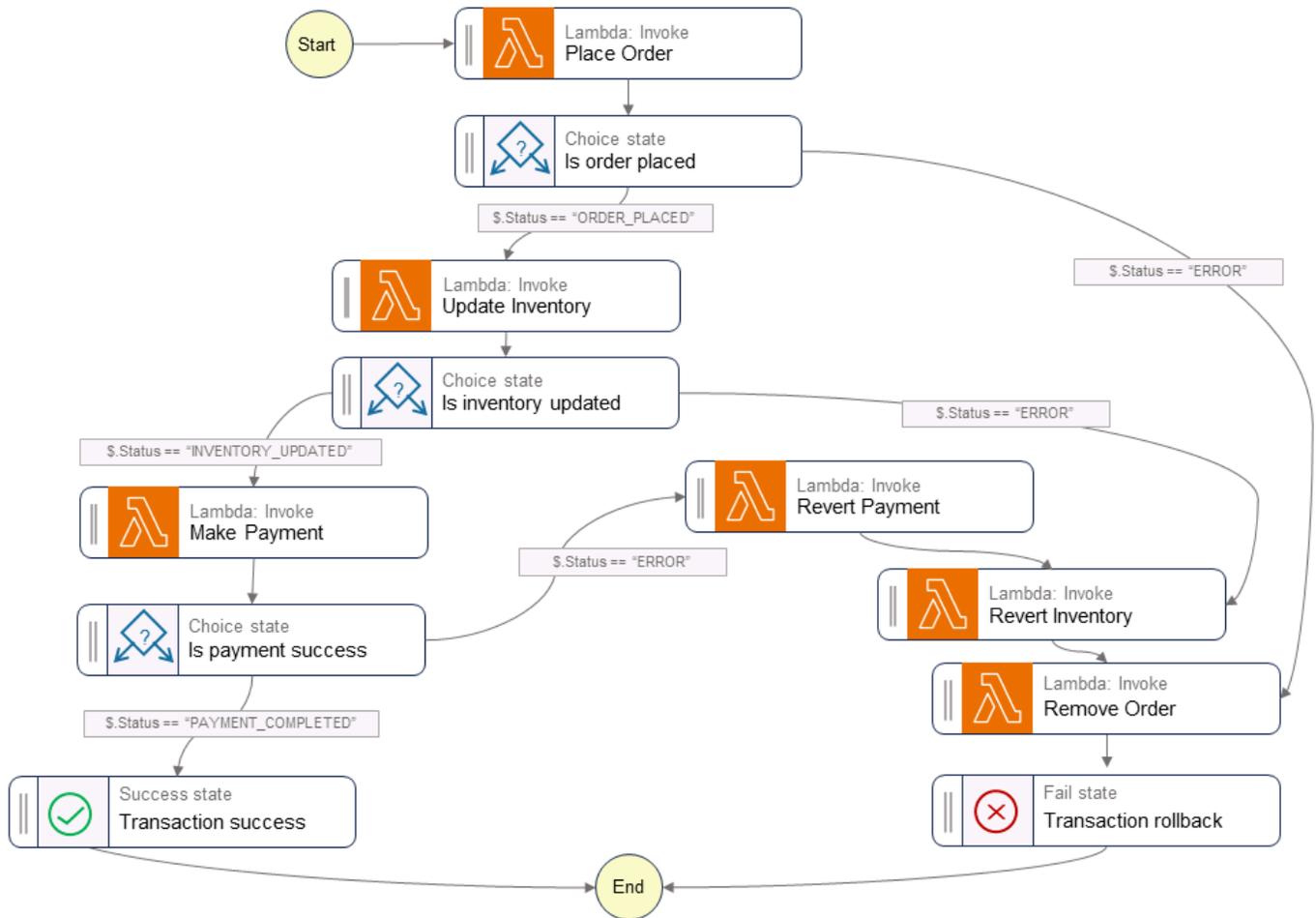
Step Functions ワークフロー

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。完全なコードを表示するには、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> の 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

次の動画では、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ます AWS Step Functions。

散布図パターン

Intent

散布図パターンは、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、アグリゲータと呼ばれるコンポーネントを使用してレスポンスを1つのメッセージに集約することを含むメッセージルーティングパターンです。このパターンは、並列化の達成、処理レイテンシーの短縮、非同期通信の処理に役立ちます。同期アプローチを使用して散布図パターンを実装するのは簡単ですが、より強力なアプローチでは、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では、順次処理に時間がかかるリクエストを、並列処理される複数のリクエストに分割できます。API コールを使用して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できます。散布図パターンは、複数のソースからの入力が必要な場合に役立ちます。Scatter-gather は、情報に基づいた意思決定やリクエストに最適なレスポンスの選択に役立つように結果を集約します。

散布図パターンは、名前が示すように、2つのフェーズで構成されます。

- 散布フェーズは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。このフェーズでは、アプリケーションはリクエストをネットワーク全体に分散し、即時の応答を待たずに実行し続けます。
- 収集フェーズでは、アプリケーションは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または統合レスポンスに結合します。すべてのレスポンスが収集されると、1つのレスポンスに集約することも、さらに処理するために最適なレスポンスを選択することもできます。

適用対象

次の場合は、散布図パターンを使用します。

- 正確なレスポンスを作成するために、さまざまな APIs を統合する計画を立てます。このパターンは、さまざまなソースからの情報をまとまりのある全体に統合します。例えば、予約システムは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りを取得できます。

- トランザクションを完了するには、同じリクエストを複数の受信者に同時に送信する必要があります。例えば、このパターンを使用して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。
- 複数の受信者にリクエストを分散することで負荷分散を実現できる、信頼性が高くスケーラブルなシステムを実装したいと考えています。1人の受信者が失敗したり、高い負荷がかかったりしても、他の受信者は引き続きリクエストを処理できます。
- 複数のデータソースを含む複雑なクエリを実装するときに、パフォーマンスを最適化する必要があります。クエリに関連するデータベースに分散し、結果の一部を収集して、包括的な回答にまとめることができます。
- シャーディングとレプリケーションのために、データリクエストが複数のデータ処理エンドポイントにルーティングされるマップ削減処理のタイプを実装しています。部分的な結果がフィルタリングされ、結合されて適切なレスポンスが作成されます。
- キーバリュースペースの書き込みが多いワークロードのパーティションスペースに書き込みオペレーションを分散したい。アグリゲータは、各シャード内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します。

問題点と考慮事項

- 耐障害性: このパターンは、並行して動作する複数の受信者に依存するため、障害を適切に処理することが不可欠です。受信者の障害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できます。
- スケールアウト制限: 処理ノードの合計数が増えると、関連するネットワークオーバーヘッドも増加します。ネットワーク経由の通信を伴うすべてのリクエストは、レイテンシーを増加させ、並列化の利点に悪影響を及ぼす可能性があります。
- 応答時間のボトルネック: 最終処理が完了する前にすべての受信者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、受信者の応答時間が最も遅くなります。
- 部分的なレスポンス: リクエストが複数の受信者に分散されると、一部の受信者がタイムアウトすることがあります。このような場合、実装はレスポンスが不完全であることをクライアントに伝える必要があります。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こともできます。
- データ整合性: 複数の受信者間でデータを処理する場合は、データの同期と競合解決の手法を慎重に検討して、最終的な集計結果が正確で一貫性があることを確認する必要があります。

実装

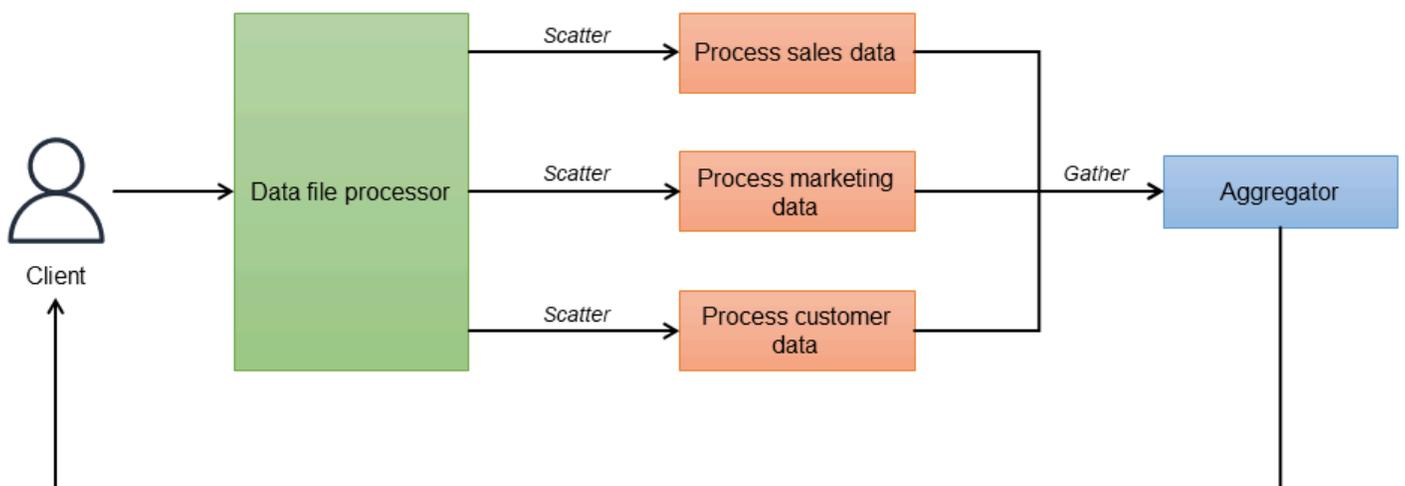
高レベルのアーキテクチャ

散布図パターンでは、ルートコントローラーを使用して、リクエストを処理する受信者にリクエストを分散します。散布フェーズでは、このパターンで2つのメカニズムを使用して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、結果を取得するために呼び出す必要がある受信者の既知のリストがあります。受信者は、一意の関数を持つ異なるプロセスでも、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もかまいません。いずれかの処理ノードがタイムアウトしたり、応答の遅延が表示されたりすると、コントローラーは処理を別のノードに再分散できます。
- Scatter by auction: アプリケーションは、[パブリッシュ/サブスクライブパターン](#) を使用して、関心のある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。

ディストリビューションによる散布

分散による散布方法では、ルートコントローラーは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、その計算で独立して並列に動作し、レスポンスの一部を生成します。受信者がタスクを完了すると、アグリゲータ (収集フェーズ) に応答を送信します。アグリゲータは部分的なレスポンスを結合し、最終的な結果をクライアントに返します。次の図は、このワークフローを示しています。

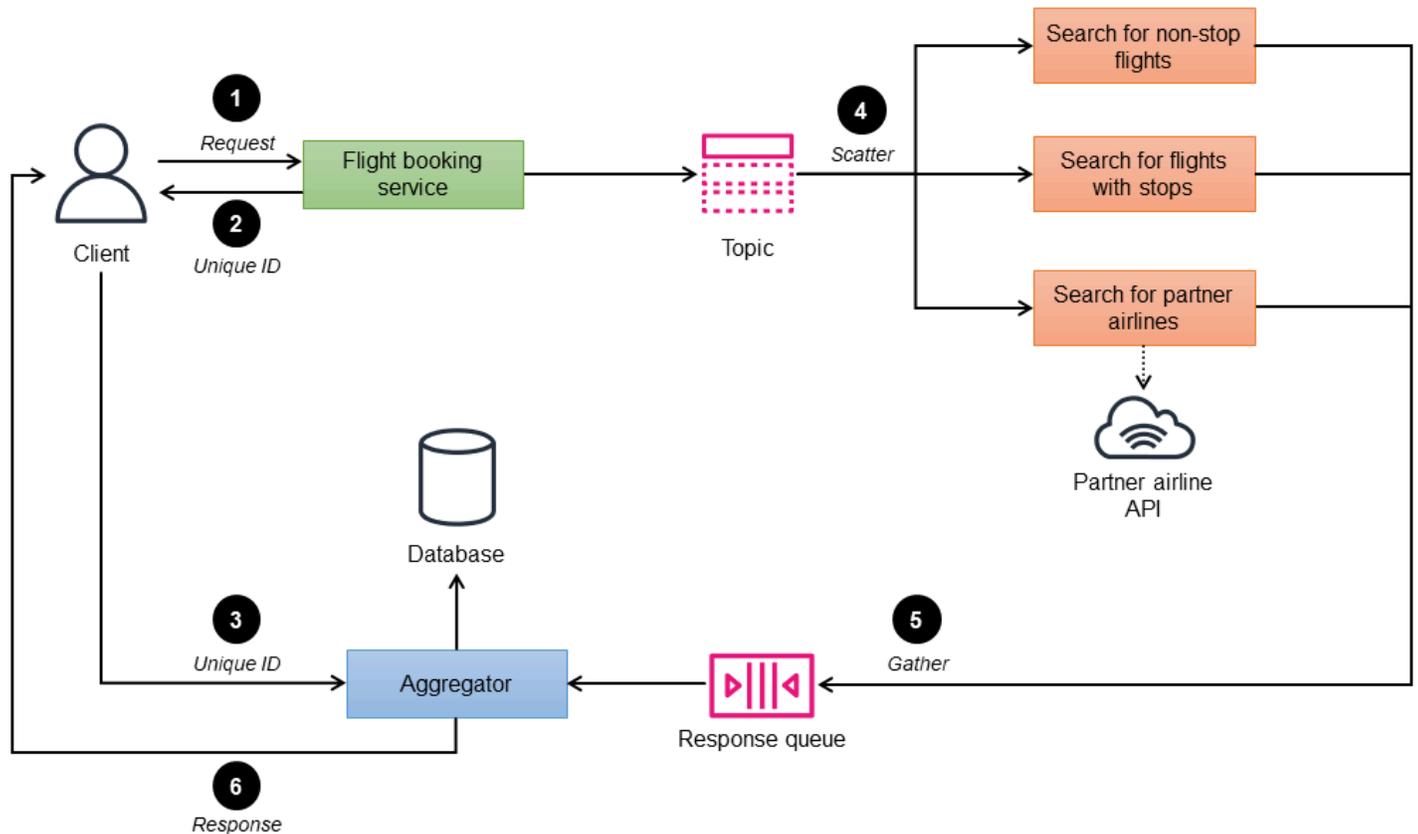


コントローラー (データファイルプロセッサ) は呼び出しのセット全体をオーケストレーションし、呼び出すすべての予約エンドポイントを認識します。タイムアウトパラメータを設定して、時間がかかりすぎるレスポンスを無視できます。リクエストが送信されると、アグリゲータは各エンドポイントからのレスポンスを待ちます。レジリエンスを実装するために、各マイクロサービスを複数のインスタンスでデプロイして負荷分散を行うことができます。アグリゲータは結果を取得し、それらを1つの応答メッセージに結合して、さらに処理する前に重複データを削除します。タイムアウトしたレスポンスは無視されます。コントローラーは、別のアグリゲータサービスを使用する代わりに、アグリゲータとして機能することもできます。

入札による散布

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されている場合は、アトクシオン方式で散布図を使用できます。この方法では、受信者はトピックをサブスクライブし、コントローラーは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ないため、収集プロセスではアグリゲータ (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、1つのレスポンスメッセージに分割します。アグリゲータは一意の ID を使用してリクエストのグループを識別します。

例えば、次の図では、入札による散布法を使用して、航空会社のウェブサイトにはフライト予約サービスを実装しています。このウェブサイトでは、ユーザーは航空会社のキャリアとそのパートナーのキャリアからフライトを検索して表示でき、検索のステータスをリアルタイムで表示する必要があります。フライト予約サービスは、3つの検索マイクロサービスで構成されています。ノンストップ便、ストップ付き便、およびパートナー航空会社です。パートナー航空会社の検索では、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。

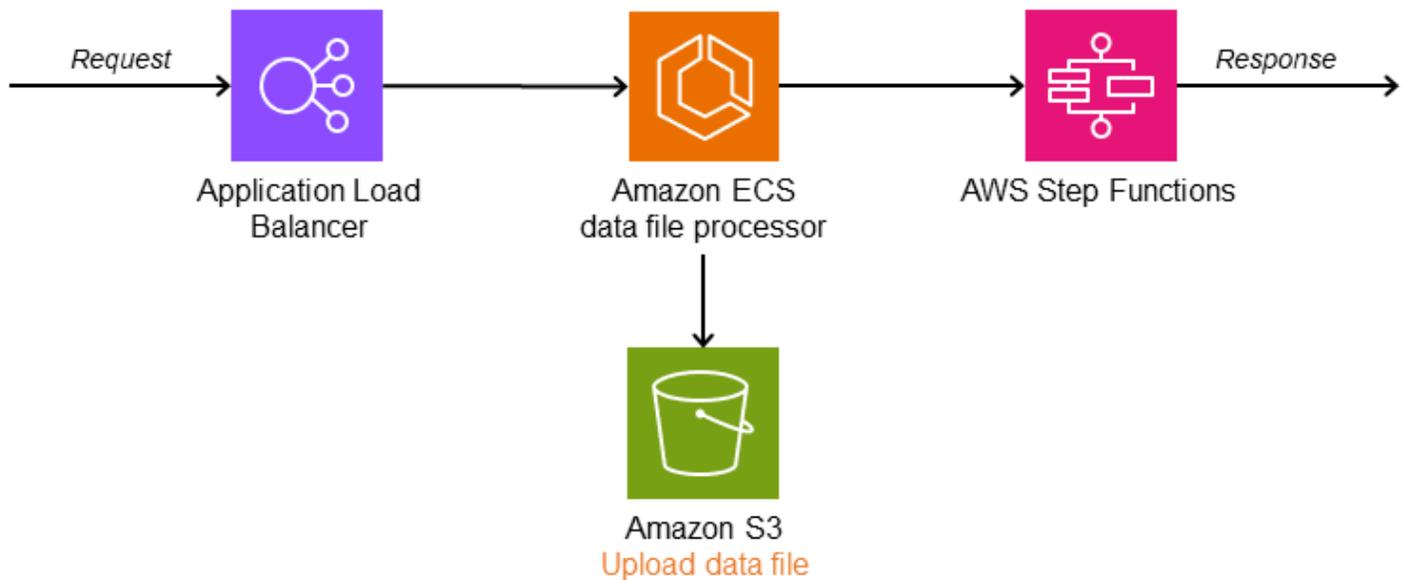


1. フライト予約サービス (コントローラー) は、検索条件をクライアントからの入力として受け取り、リクエストを処理してトピックに発行します。
2. コントローラーは一意の ID を使用してリクエストの各グループを識別します。
3. クライアントは、ステップ 6 の一意の ID をアグリゲータに送信します。
4. 予約トピックにサブスクライブしている予約検索マイクロサービスはリクエストを受け取ります。
5. マイクロサービスはリクエストを処理し、指定された検索条件の座席の可用性をレスポンスキューに返します。
6. アグリゲータは、一時データベースに保存されているすべてのレスポンスメッセージを照合し、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、クライアントに送り返します。

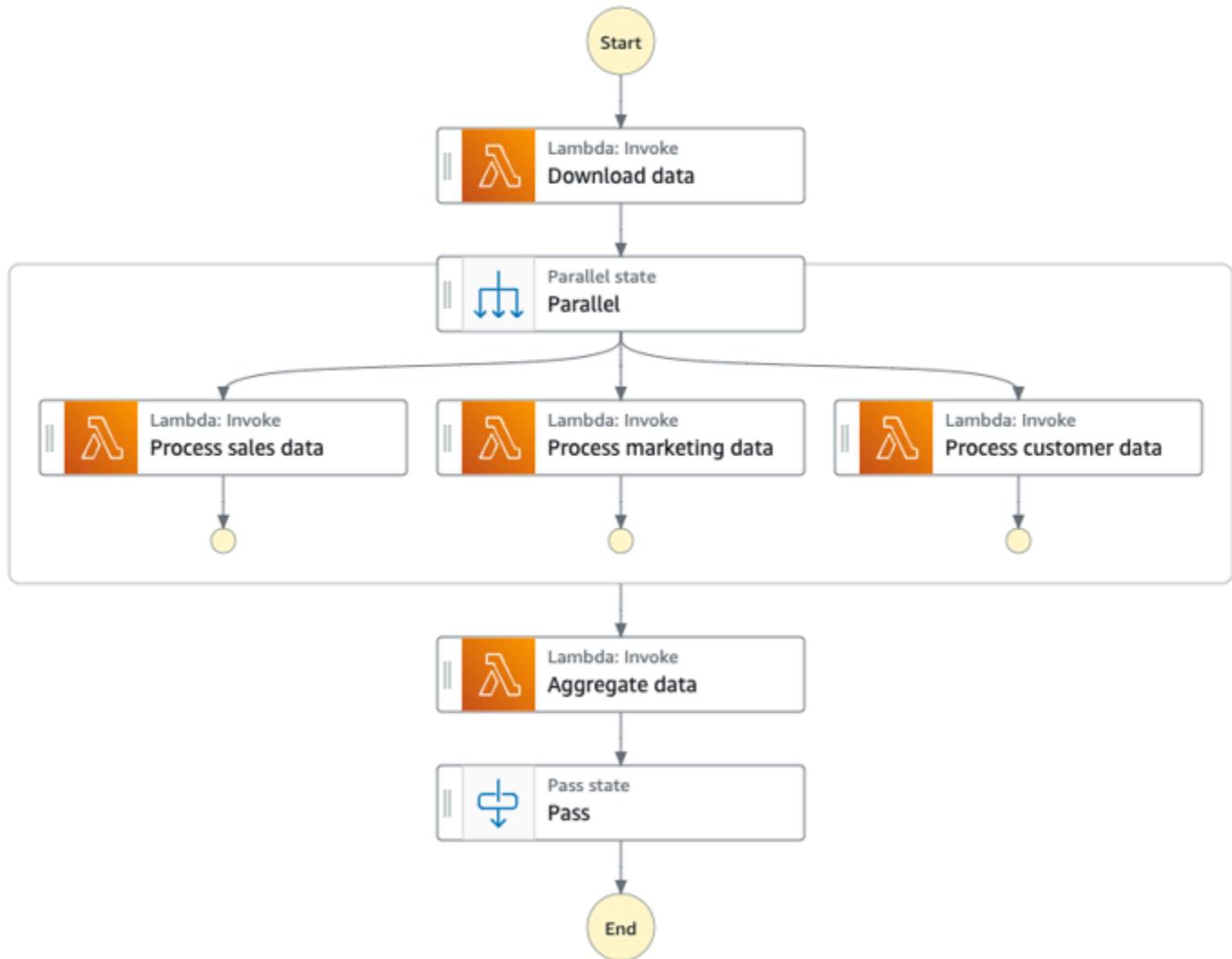
を使用した実装 AWS のサービス

ディストリビューションによる散布

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 ワークフローはデータをダウンロードし、並列ファイル処理を開始します。Parallel 状態は、すべてのタスクがレスポンスを返すのを待ちます。AWS Lambda 関数はデータを集約し、Amazon S3 に保存します。

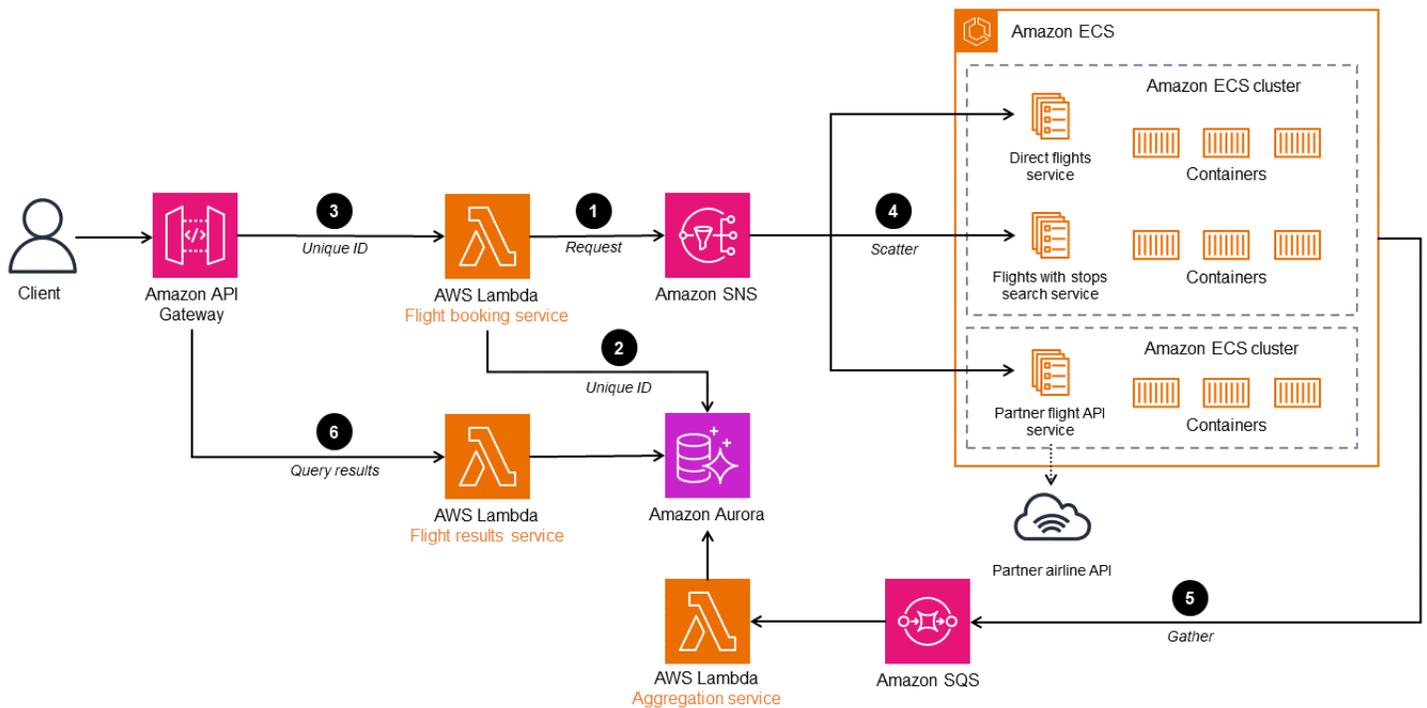


次の図は、Parallel状態の Step Functions ワークフローを示しています。



入札による散布

次の図は、の分散の AWS アーキテクチャを入札方法別に示しています。ルートコントローラーフ ライト予約サービスは、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ンネルは、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) で実装されます。Amazon SNS は、分離されたマイクロ サービスアプリケーション間、またはユーザーへの直接通信間のメッセージをサポートします。受 信者マイクロサービスを Amazon Elastic Kubernetes Service (Amazon EKS) または Amazon Elastic Container Service (Amazon ECS) にデプロイして、管理とスケーラビリティを向上させることが できます。フライト結果サービスは結果をクライアントに返します。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) は、クライアントからの入力として検索条件を受け取り、リクエストを処理して SNS トピックに発行します。
2. コントローラーは一意的 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. クライアントは、ステップ 6 で一意的 ID をクライアントに送信します。
4. 予約トピックにサブスクライブしている予約検索マイクロサービスは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可用性を Amazon Simple Queue Service (Amazon SQS) のレスポンスキューに返します。アグリゲータは、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は、フライトを一意的 ID でグループ化し、単一の統合レスポンスを作成して、クライアントに送り返します。

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、SNS トピックをサブスクライブし、SQS キューに発行するマイクロサービスを追加します。

要約すると、散布図パターンにより、分散システムは効率的な並列化を実現し、レイテンシーを短縮し、非同期通信をシームレスに処理できます。

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

- デカップリングマイクロサービスワークショップの[散布図ラボ](#)

ブログの参考情報

- [マイクロサービスのアプリケーション統合パターン](#)

関連情報

- [Publish-Subscribe](#) パターン

ストラングラーフィグパターン

目的

strangler figパターンは、モノリシックな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するのに役立ち、トランスフォーメーションリスクとビジネス中断を軽減します。

モチベーション

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実現するように開発されています。コードは緊密に結合されています。そのため、アプリケーションの変更では、リグレッションの問題を避けるために徹底的な再テストが必要になります。変更を単独でテストすることはできず、サイクルタイムに影響します。アプリケーションがより多くの機能で強化されるにつれて、複雑さが増すと、メンテナンスに費やされる時間が増え、市場投入までの時間が長くなり、その結果、製品のイノベーションが遅くな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、チームの認知的負荷が増大し、チームの所有権の境界が不明確になる可能性があります。負荷に基づいて個々の機能をスケールアップすることはできません。ピーク時の負荷に対応するようにアプリケーション全体をスケールアップする必要があります。システムが古くなると、テクノロジーが時代遅れになり、サポートコストが高くな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、開発当時は利用可能で、配布するようには設計されていなかったベストプラクティスに従っています。

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移行すると、より小さなコンポーネントに分割できます。これらのコンポーネントは個別に拡張でき、個別にリリースでき、個々のチームが所有できます。その結果、変更がローカライズされ、テストやリリースが迅速に行えるため、変更の速度が速くなります。コンポーネントは緩く結合されており、個別に展開できるため、変更の影響範囲は小さくなります。

コードを書き直したりリファクタリングしたりして、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のは大変な作業であり、大きなリスクを伴います。モノリスを一度の操作で移行するビッグバン移行は、変革リスクと事業中断を招きます。アプリケーションがリファクタリングされている間は、新しい機能を追加するのは非常に難しく、不可能ですらあります。

この問題を解決する1つの方法は、マーティン・ファウラーによって導入されたストラングラー・フィグ・パターンを使用することです。このパターンでは、徐々に特徴を抽出し、既存のシステムを中心に新しいアプリケーションを作成することで、マイクロサービスに移行します。モノリスの機

能は徐々にマイクロサービスに置き換えられ、アプリケーションユーザーは新しく移行された機能を徐々に使用できるようになります。すべての機能を新しいシステムに移行すれば、モノリシックなアプリケーションを安全に廃止できます。

適用性

ストラングラー・フィグ・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徐々にマイクロサービスアーキテクチャに移行したい。
- モノリスはその規模と複雑さゆえに、ビッグバン移行アプローチにはリスクが伴います。
- 企業は新しい機能の追加を望んでおり、変革が完了するのが待ちきれません。
- 変革中のエンドユーザーへの影響を最小限に抑える必要があります。

問題と考慮事項

- 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、monolith アプリケーションのコードベースにアクセスする必要があります。機能がモノリスから移行されると、コードを少し変更し、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する必要があります。コードベースにアクセスできなければ、呼び出しをインターセプトすることはできません。コードベースアクセスは、受信リクエストをリダイレクトするためにも重要です。移行された機能の呼び出しをプロキシレイヤーが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、コードのリファクタリングが必要になる場合があります。
- 不明瞭なドメイン:システムの早期の分解は、特にドメインが明確でない場合はコストがかかり、サービスの境界を間違える可能性があります。ドメイン駆動設計 (DDD) はドメインを理解するためのメカニズムであり、イベントストーミングはドメイン境界を決定する手法です。
- 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ための重要なツールとして使用できます。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区分を探してください。多くのサービス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、簡単に切り離すことができます。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、またはほとんどないクラスは、マイクロサービスの有力な候補です。モノリスを分解する前にコードをリファクタリングして、緊密な結合を防ぐことができます。また、コンプライアンス要件、リリース間隔、チームの地理的位置、スケーリングニーズ、ユースケース主導型のテクノロジーニーズ、チームの認知的負荷について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された機能を呼び出す必要がある場合は、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腐敗防止

層 (ACL) を実装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て変更を防ぐため、ACL は呼び出しを新しいインターフェー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ます。これについては、このガイドの前半の ACL パターンの「[実装](#)」セクションで詳しく説明しています。

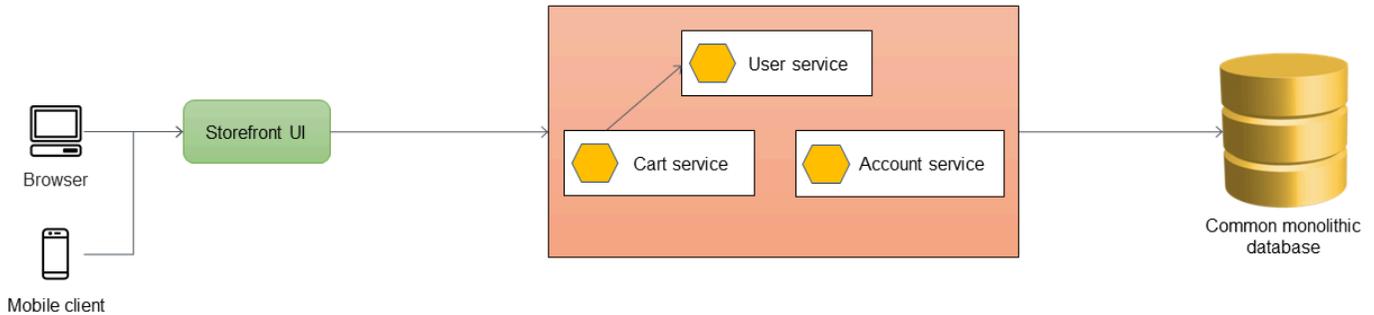
- **プロキシレイヤーの障害:**移行中、プロキシレイヤーはモノリシックアプリケーションに送られるリクエストをインターセプトし、レガシーシステムまたは新しいシステムにルーティングします。ただし、このプロキシレイヤーは単一障害点になったり、パフォーマンスのボトルネックになったりする可能性があります。
- **アプリケーションの複雑さ:**ストラングラー・フィグ・パターンは、大きなモノリスに最大のメリットをもたらします。完全なリファクタリングの複雑さが低い小規模なアプリケーションでは、アプリケーションを移行するよりもマイクロサービスアーキテクチャで書き直す方が効率的かもしれません。
- **サービスインタラクション:**マイクロサービスは同期または非同期で通信できます。同期通信が必要な場合は、タイムアウトによって接続やスレッドプールの消費が生じ、その結果、アプリケーションのパフォーマンスに問題が生じる可能性があるかどうかを検討してください。このような場合は、[サーキットブレーカーパターン](#)を使用して、長期間失敗する可能性が高い操作をただちにエラーを返します。非同期通信は、イベントとメッセージキューを使用することで実現できます。
- **データ集約:**マイクロサービスアーキテクチャでは、データはデータベース全体に分散されます。データ集約が必要な場合は、フロントエンドで使用するか、[AWS AppSync](#)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。
- **データの一貫性:**マイクロサービスは独自のデータストアを所有しており、モノリシックなアプリケーションでもこのデータを使用できる可能性があります。共有を有効にするには、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できます。ただし、これによって2つのデータストア間でデータの冗長性が生じ、最終的に整合性が保たれ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できるようになるまでは、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

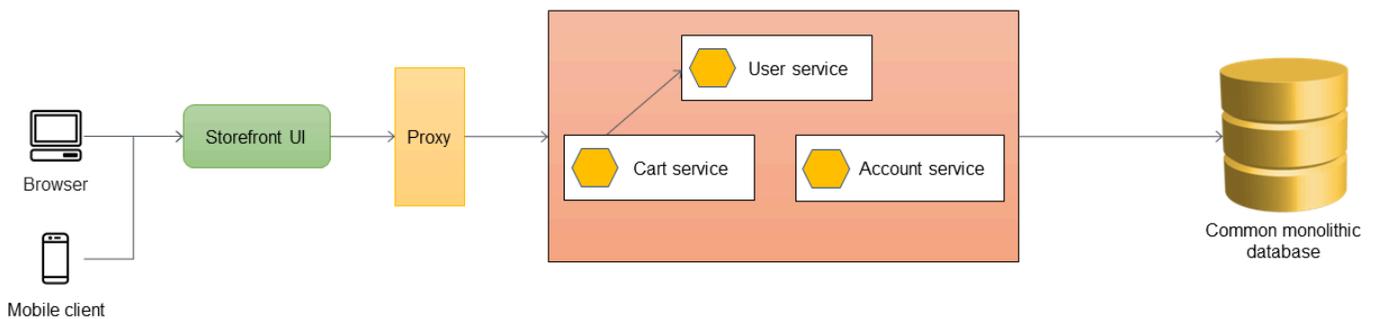
strangler figパターンでは、特定の機能を新しいサービスまたはアプリケーションに置き換えます。コンポーネントは1つずつです。プロキシ層は、モノリシックアプリケーションに送られる要求をインターセプトし、レガシーシステムまたは新しいシステムのいずれかにルーティングします。プロキシレイヤーはユーザーを正しいアプリケーションにルーティングするので、モノリスが機能し続けることを保証しながら、新しいシステムに機能を追加できます。新しいシステムでは、最終的に古いシステムのすべての機能が置き換えられ、使用を停止することができます。

高レベルのアーキテクチャ

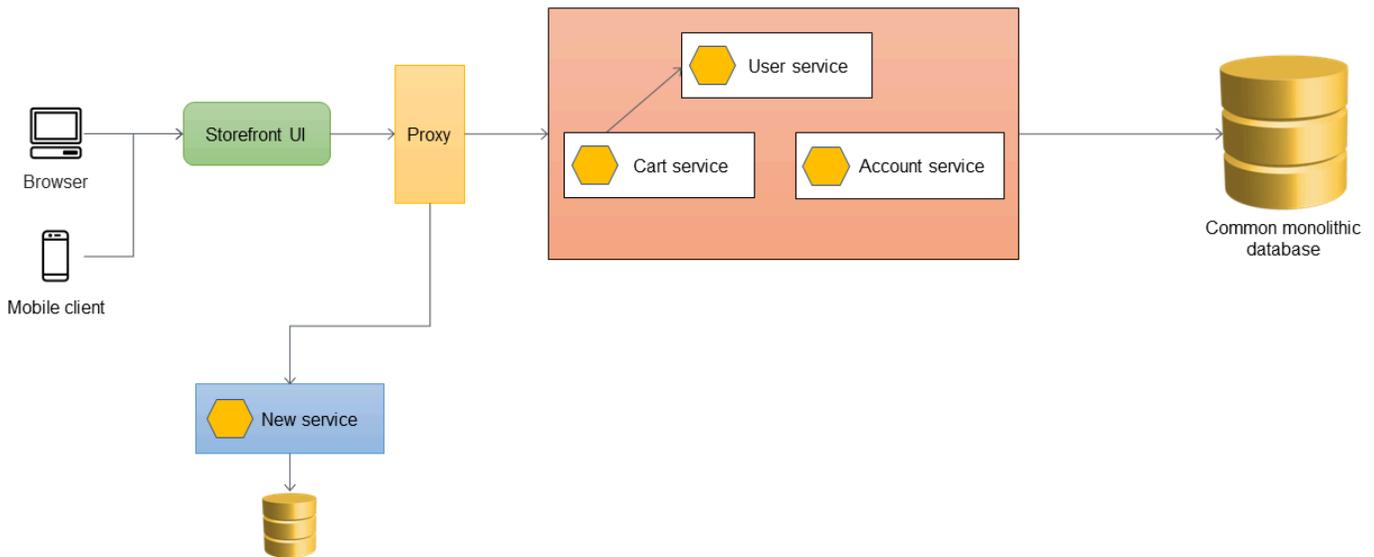
以下の図では、モノリシックアプリケーションには、ユーザーサービス、カートサービス、アカウントサービスの3つのサービスがあります。カートサービスはユーザーサービスによって異なり、アプリケーションはモノリシックリレーショナルデータベースを使用します。



最初のステップは、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することです。最初に、プロキシはすべてのトラフィックをモノリシックアプリケーションにルーティングします。

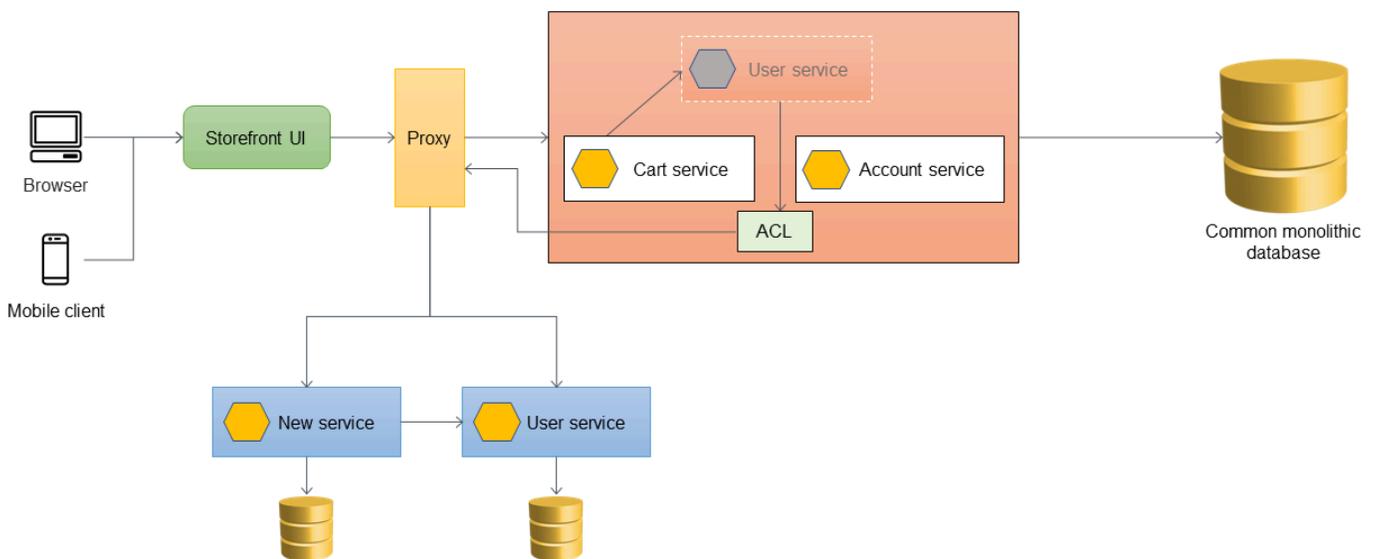


アプリケーションに新しい機能を追加したいときは、既存のモノリスに機能を追加するのではなく、新しいマイクロサービスとして実装します。ただし、アプリケーションの安定性を確保するために、引き続きモノリスのバグを修正します。以下の図では、プロキシ層は API URL に基づいて呼び出しをモノリスまたは新しいマイクロサービスにルーティングします。



腐敗防止レイヤーの追加

以下のアーキテクチャでは、ユーザーサービスはマイクロサービスに移行されています。カートサービスはユーザーサービスを呼び出しますが、その実装はモノリス内では利用できなくなりました。また、新しく移行されたサービスのインターフェースは、モノリシックアプリケーション内の以前のインターフェースと一致しない場合があります。これらの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された機能呼び出す必要がある場合、ACL は呼び出しを新しいインターフェースに変換し、適切なマイクロサービスにルーティングします。

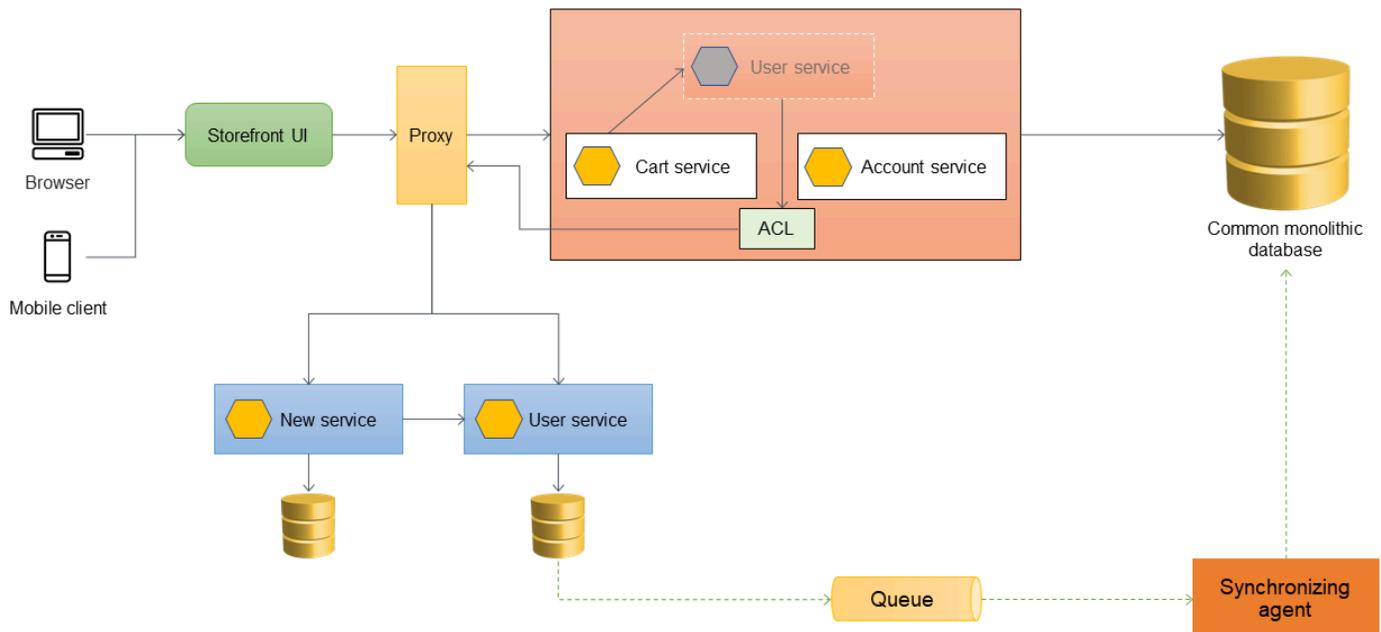


ACL は、移行されたサービスに固有のクラスとしてモノリシックアプリケーション内に実装できます (例:、)。UserServiceFacade UserServiceAdapter 依存するサービスをすべてマイクロサービスアーキテクチャに移行したら、ACL を廃止する必要があります。

ACL を使用しても、カートサービスは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その呼び出しを ACL を介してマイクロサービスにリダイレクトします。カートサービスは依然として、マイクロサービスの移行を意識せずにユーザーサービスを呼び出す必要があります。このような緩やかな結合は、リグレッションや事業の中断を減らすために必要です。

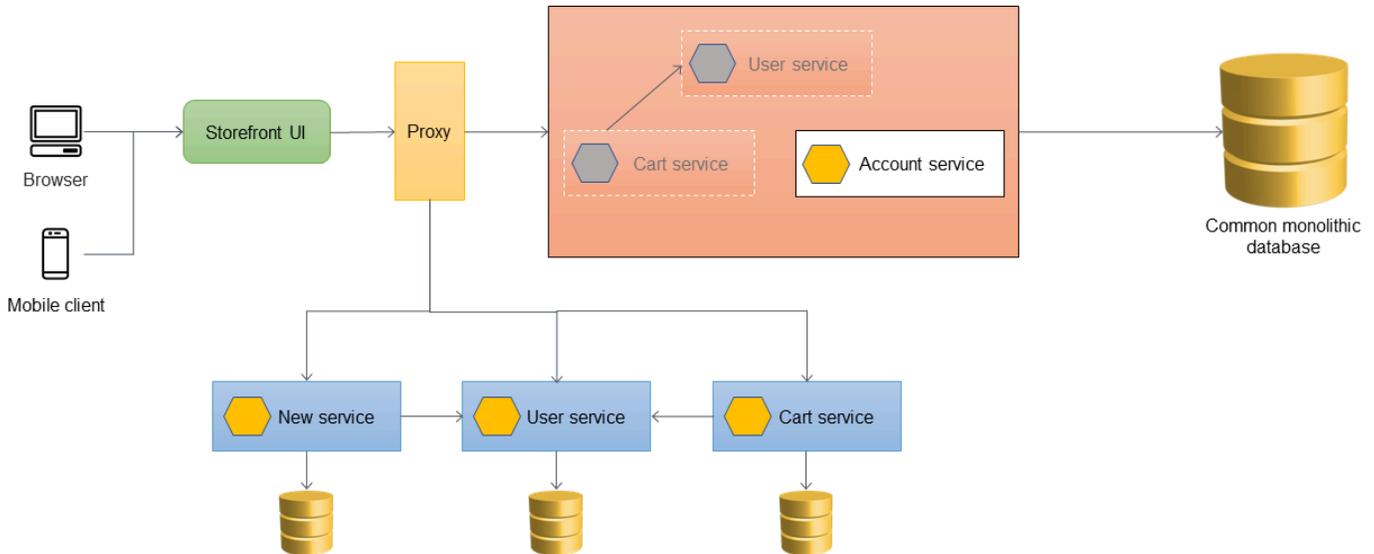
データ同期の処理

ベストプラクティスとして、マイクロサービスはデータを所有する必要があります。ユーザーサービスは独自のデータストアにデータを保存します。レポートなどの依存関係処理したり、まだマイクロサービスに直接アクセスする準備ができていないダウンストリームのアプリケーションをサポートしたりするために、モノリシックデータベースとデータを同期する必要がある場合があります。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されていない他の機能やコンポーネントのデータが必要になる場合もあります。そのため、新しいマイクロサービスとモノリスの間でデータを同期する必要があります。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できます。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キューにイベントを送信します。同期エージェントはキューを監視し、モノリシックデータベースを継続的に更新します。モノリシックデータベース内のデータは、最終的に同期されるデータと一致します。

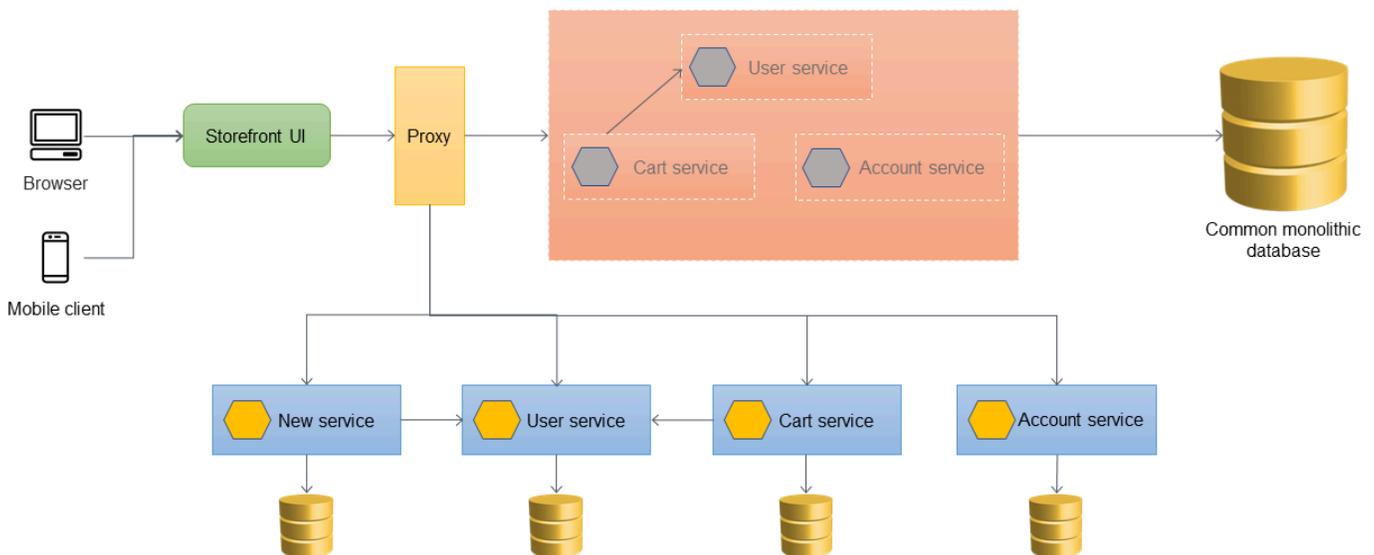


追加サービスの移行

カートサービスがモノリシックアプリケーションから移行されると、新しいサービスを直接呼び出すようにコードが修正され、ACL はそれらの呼び出しをルーティングしなくなります。このアーキテクチャを以下に図で示します。

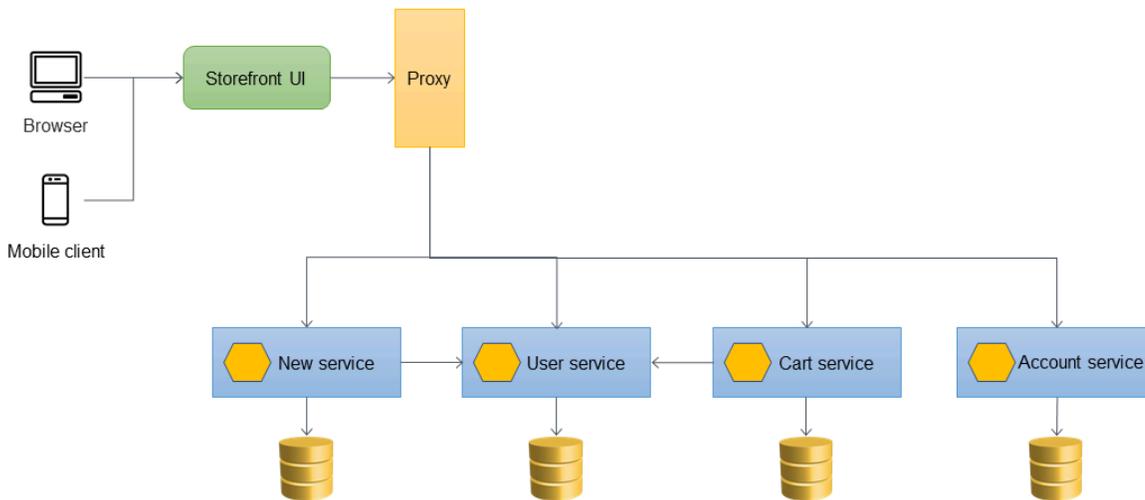


以下の図は、すべてのサービスがモノリスから移行され、モノリスのスケルトンのみが残るといふ最終状態を示しています。履歴データは、個々のサービスが所有するデータストアに移行できます。ACL は削除可能で、この段階でモノリスを廃止する準備ができています。



次の図は、モノリシックアプリケーションが廃止された後の最終的なアーキテクチャを示しています。アプリケーションの要件に応じて、リソースベースの URL (など `http://`

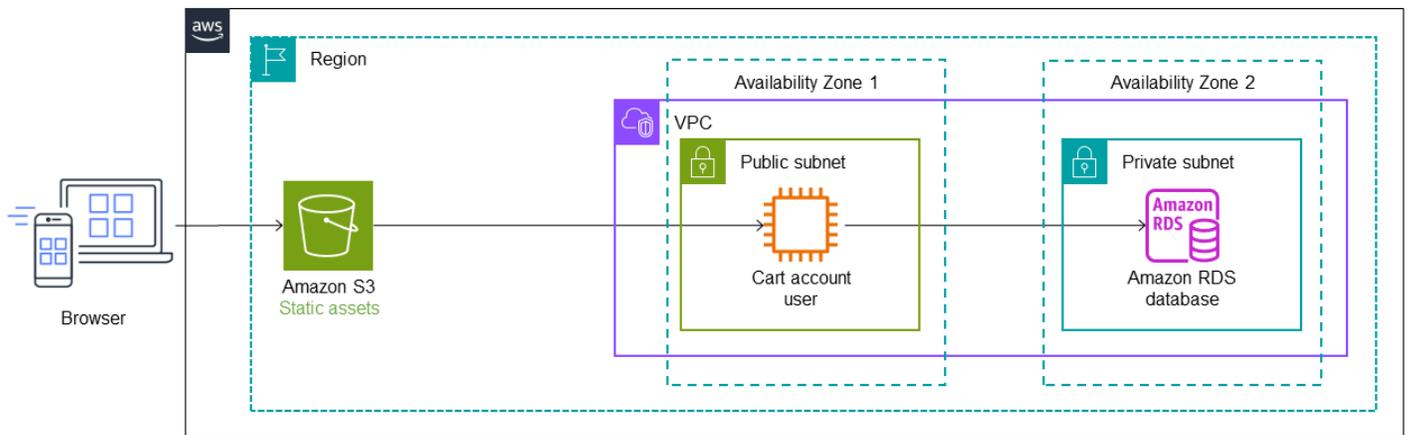
`www.storefront.com/user`) または独自のドメイン (など) を使用して個々のマイクロサービスをホストできます。 <http://user.storefront.com> ホスト名とパスを使用して HTTP API を上流の消費者に公開する主な方法の詳細については、「API ルーティングパターン」セクションを参照してください。



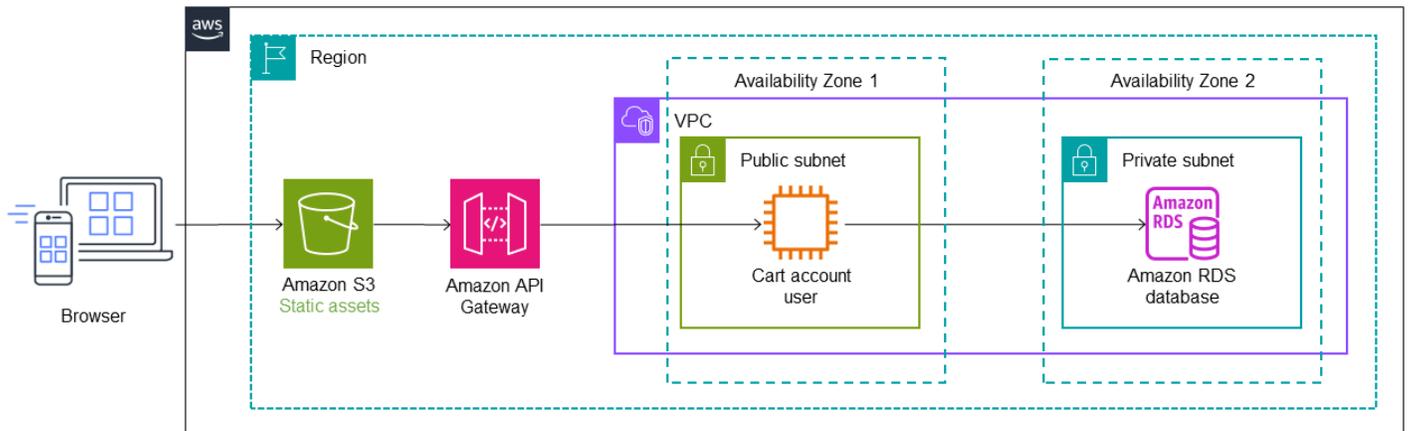
AWSサービスを使った実装

API Gateway をアプリケーションプロキシとして使用

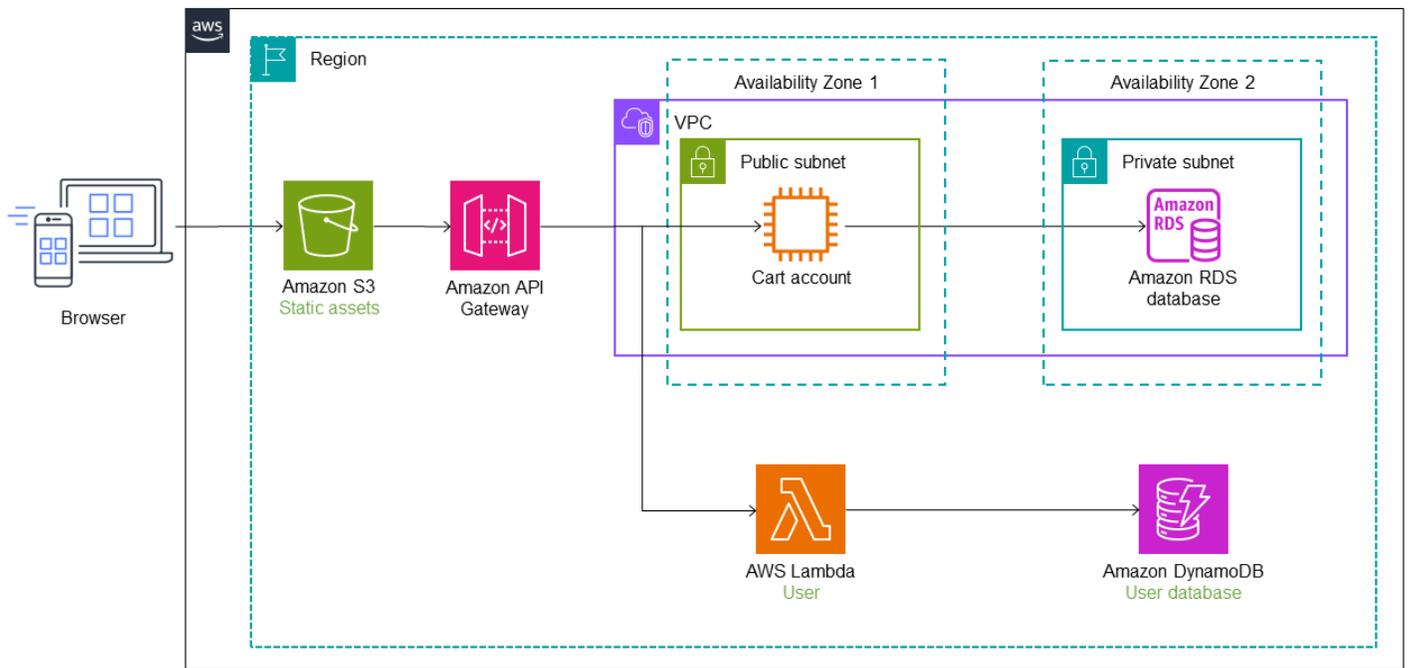
次の図は、モノリシックアプリケーションの初期状態を示しています。 lift-and-shift ストラテジーを使用してに移行されたと仮定してAWS、 [Amazon Elastic Compute Cloud \(Amazon EC2\) インスタンス上で実行され、Amazon Relational Database Service \(Amazon RDS\) データベースを使用している](#) とします。わかりやすくするために、このアーキテクチャでは 1 つのプライベートサブネットと 1 つのパブリックサブネットを持つ 1 つの仮想プライベートクラウド (VPC) を使用します。マイクロサービスは最初と同じサブネット内にデプロイされると仮定します。AWS アカウント(実稼働環境でのベストプラクティスは、デプロイの独立性を確保するためにマルチアカウントアーキテクチャを使用することです)。EC2 インスタンスはパブリックサブネットの 1 つのアベイラビリティーゾーンにあり、RDS インスタンスはプライベートサブネットの 1 つのアベイラビリティーゾーンに存在します。 [Amazon Simple Storage Service \(Amazon S3\)](#) には、ウェブサイトの JavaScript、CSS、React ファイルなどの静的アセットが保存されます。



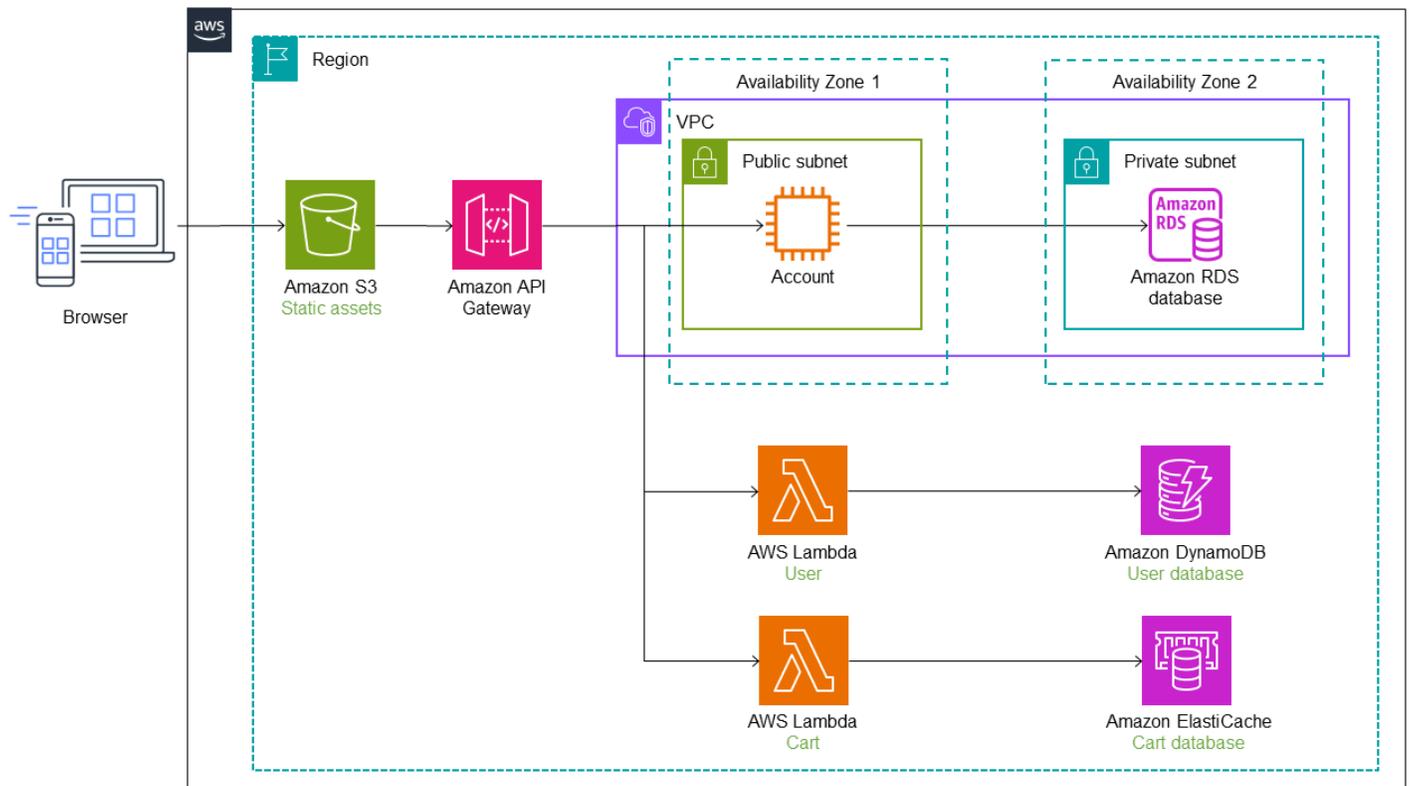
次のアーキテクチャでは、モノリシックアプリケーションの前に [Amazon API Gateway AWS Migration Hub Refactor Spaces](#) をデプロイします。Refactor Spaces はアカウント内にリファクタリングインフラストラクチャを作成し、API Gateway は呼び出しをモノリスにルーティングするためのプロキシレイヤーとして機能します。最初は、すべての呼び出しがプロキシ層を経由してモノリシックアプリケーションにルーティングされます。前に説明したように、プロキシ層は単一障害点になる可能性があります。ただし、API Gateway はサーバーレスのマルチ AZ サービスであるため、プロキシとして使用するとリスクが軽減されます。



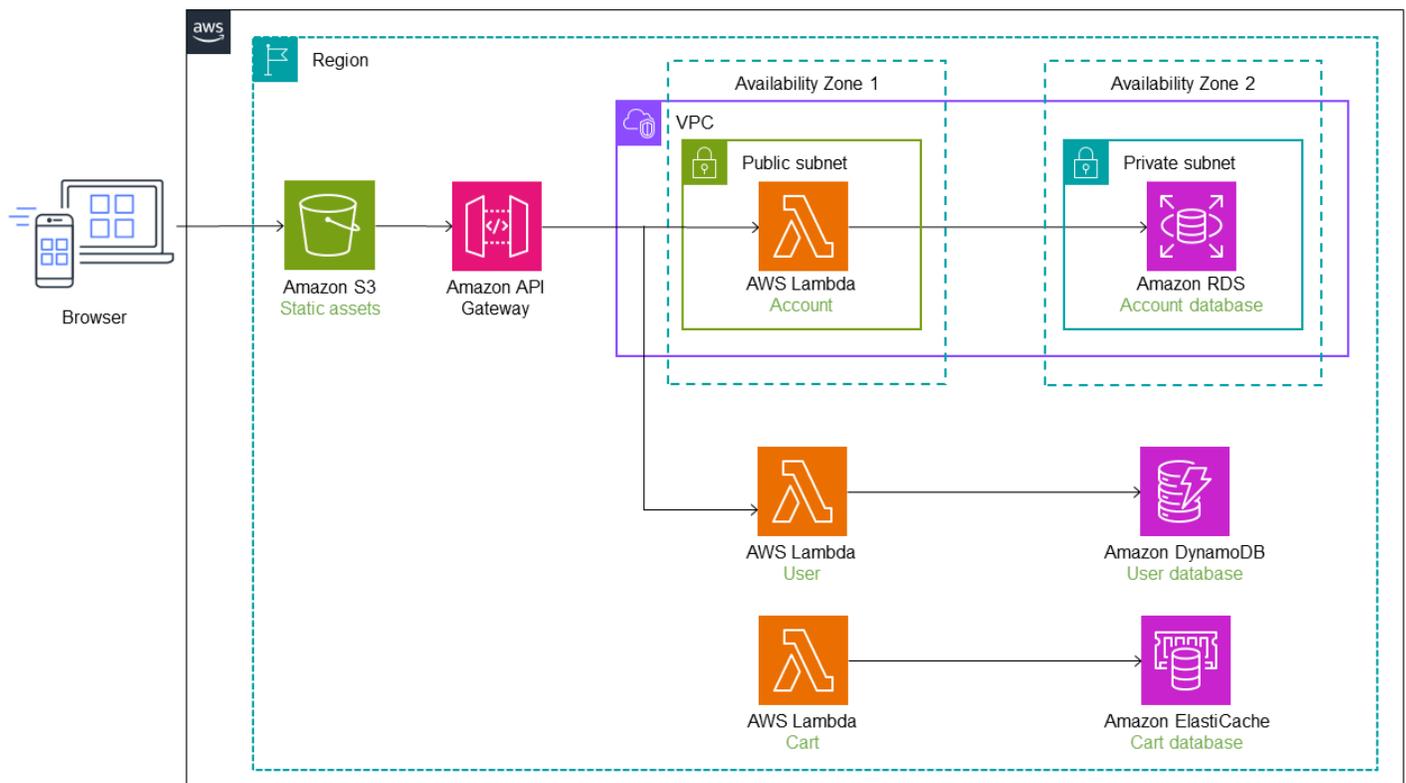
ユーザーサービスは Lambda 関数に移行され、[Amazon DynamoDB](#) データベースはそのデータを保存します。Lambda サービスのエンドポイントとデフォルトルートがリファクタリングスペースに追加され、呼び出しを Lambda 関数にルーティングするように API Gateway が自動的に設定されます。[実装の詳細については、「反復型アプリケーションモダナイゼーションワークショップ」のモジュール 2 を参照してください。](#)



以下の図では、カートサービスもモノリスから Lambda 関数に移行されています。ルートとサービスエンドポイントがリファクタリングスペースに追加され、トラフィックは自動的に Cart Lambda 関数に切り替わります。[Lambda 関数のデータストアは Amazon によって管理されています。](#)
[ElastiCache](#) モノリシックアプリケーションは Amazon RDS データベースとともに EC2 インスタンスに残ります。



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 関数に移行されます。元の Amazon RDS データベースを引き続き使用します。現在、新しいアーキテクチャには、別々のデータベースを持つ 3 つのマイクロサービスがあります。各サービスは異なる種類のデータベースを使用します。マイクロサービスの特定のニーズを満たすために専用のデータベースを使用するというこの概念は、ポリグロットパーシスタンスと呼ばれます。Lambda 関数は、ユースケースに応じてさまざまな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、Refactor Spaces はトラフィックのカットオーバーと Lambda へのルーティングを自動化します。これにより、ビルダーはルーティングインフラストラクチャの設計、デプロイ、設定に必要な時間を節約できます。



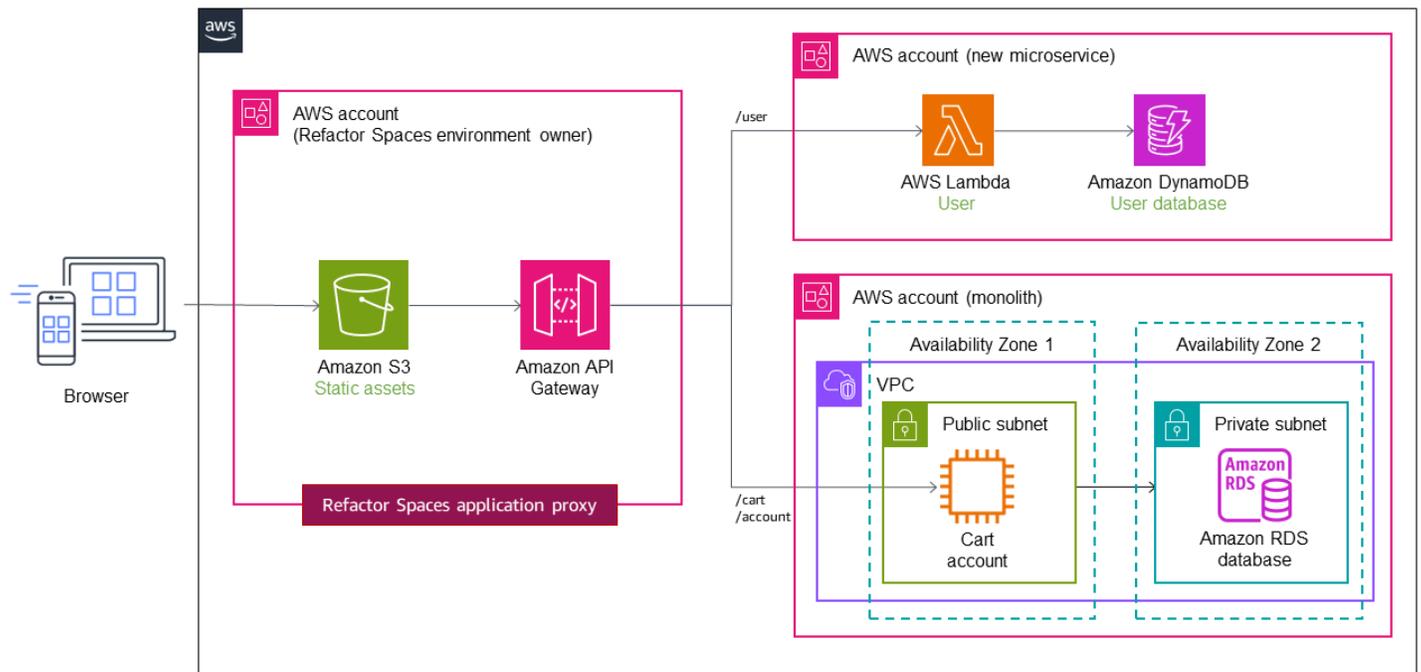
複数のアカウントを使用する

以前の実装では、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を持つ単一の VPC を使用し、AWS アカウント簡単にするためにマイクロサービスを同じ内部にデプロイしました。ただし、実際のシナリオではそうなることはほとんどありません。デプロイの独立性を保つため、マイクロサービスが複数にデプロイされることがよくあります。AWS アカウントマルチアカウント構造では、モノリスから新しいサービスへのトラフィックのルーティングを異なるアカウントで設定する必要があります。

[Refactor Spaces](#) は、モノリシックなアプリケーションから離れて API AWS 呼び出しをルーティングするためのインフラストラクチャーの作成と設定に役立ちます。Refactor Spaces は、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](#)、およびリソースベース [AWS Identity and Access Management \(IAM\)](#) ポリシーを調整します。1 AWS アカウント 1 つのアカウントまたは複数のアカウントにまたがる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社内で調整され、AWS アカウント導入後にカスタマイズや設定が可能です。

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が 2 つの異なるアカウントにデプロイされていると仮定します。Refactor Spaces を使用する場合は、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 は [API Gateway と Lambda の統合と Lambda リソースポ](#)

リシーの作成を自動化するため、モノリス以外のサービスを安全にリファクタリングすることに集中できます。



リファクタリングスペースの使用に関するビデオチュートリアルについては、「[を使用してアプリを段階的にリファクタリングする](#)」を参照してください。AWS Migration Hub Refactor Spaces

ワークショップ

- [反復型アプリモダナイゼーションワークショップ](#)

ブログリファレンス

- [AWS Migration Hub Refactor Spaces](#)
- [ディープ・ダイブ・オン・マン AWS Migration Hub Refactor Spaces](#)
- [デプロイパイプライン、リファレンスアーキテクチャ、リファレンス実装](#)

関連情報

- [API ルーティングパターン](#)
- [リファクタリングスペース-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

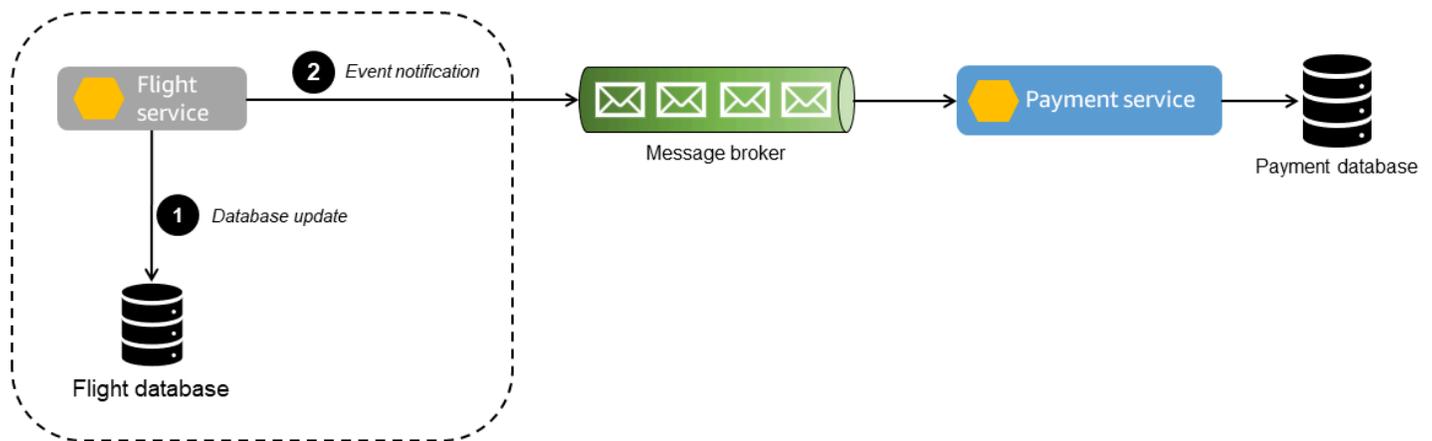
- 重複メッセージ: イベント処理サービスは重複したメッセージやイベントを送信する可能性があるため、処理されたメッセージを追跡して使用側サービスに冪等性があるようにすることをお勧めします。

- 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、データストアの point-in-time 復旧にイベントストアを使用できるイベント調達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。



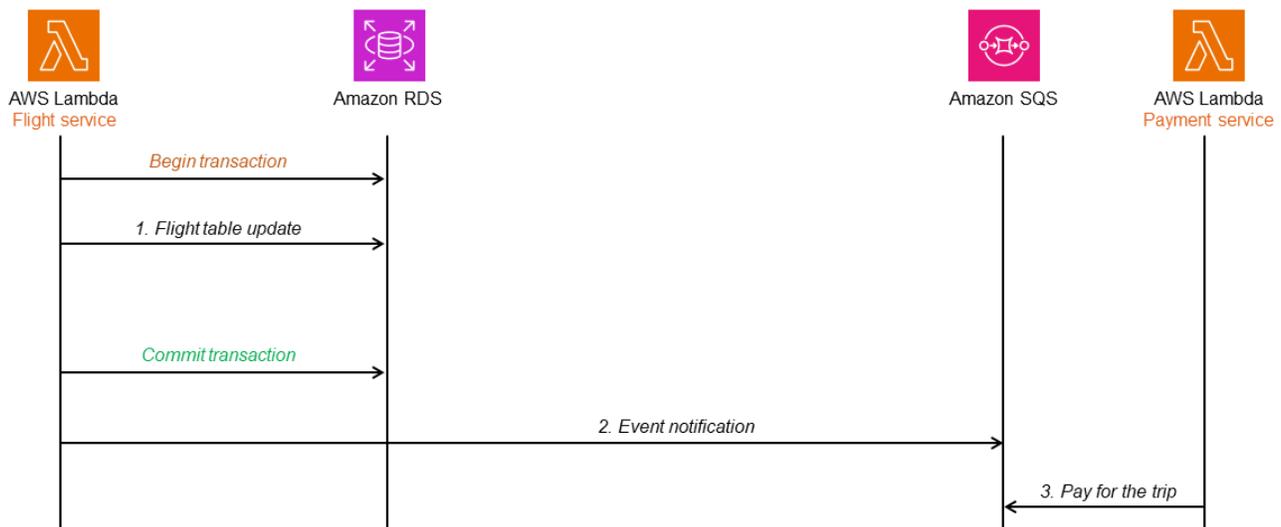
1. フライトサービスはデータベースに書き込み、支払いサービスにイベント通知を送信します。
2. メッセージブローカーはメッセージとイベントを支払いサービスに伝えます。メッセージブローカーに障害が発生すると、支払いサービスは更新を受信できなくなります。

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。

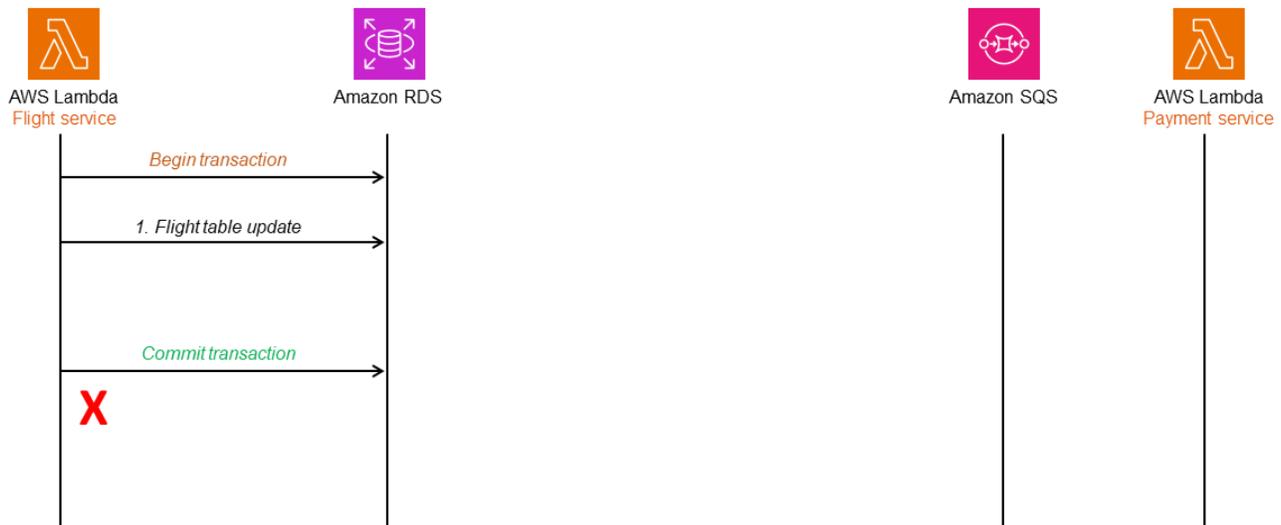
AWS のサービスを使用した実装

シーケンス図のパターンを示すために、次の AWS 図に示すように、次のサービスを使用します。

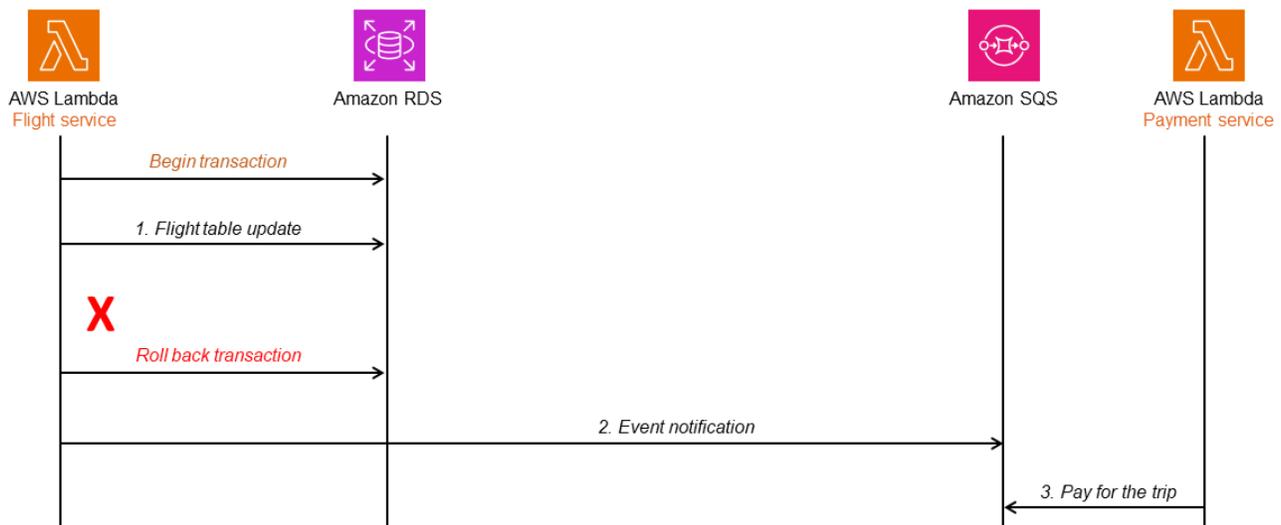
- マイクロサービスは [AWS Lambda](#) を使用して実装されます。
- プライマリデータベースは、[Amazon Relational Database Service \(Amazon RDS\)](#) によって管理されます。
- [Amazon Simple Queue Service \(Amazon SQS\)](#) は、イベント通知を受信するメッセージブローカーとして機能します。



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。



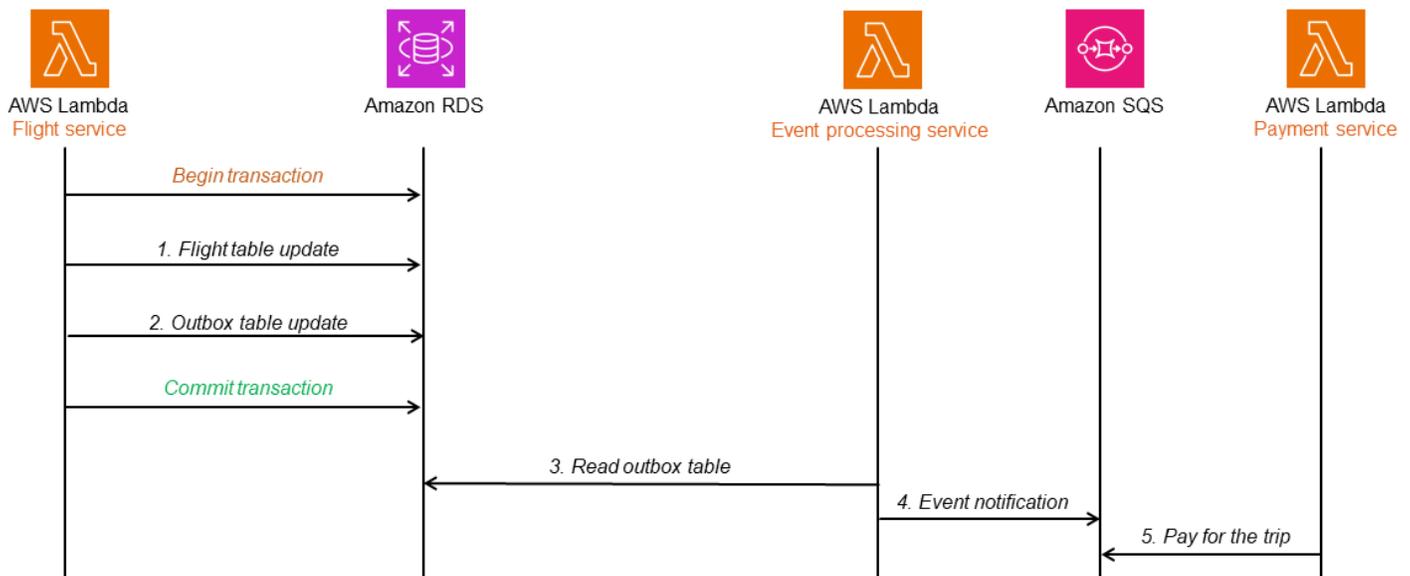
この問題に対処するには、アウトボックステーブルまたは変更データキャプチャ (CDC) を使用します。以下のセクションでは、これら 2 つのオプションと、AWS のサービスを使用してそれらを実装する方法について説明します。

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

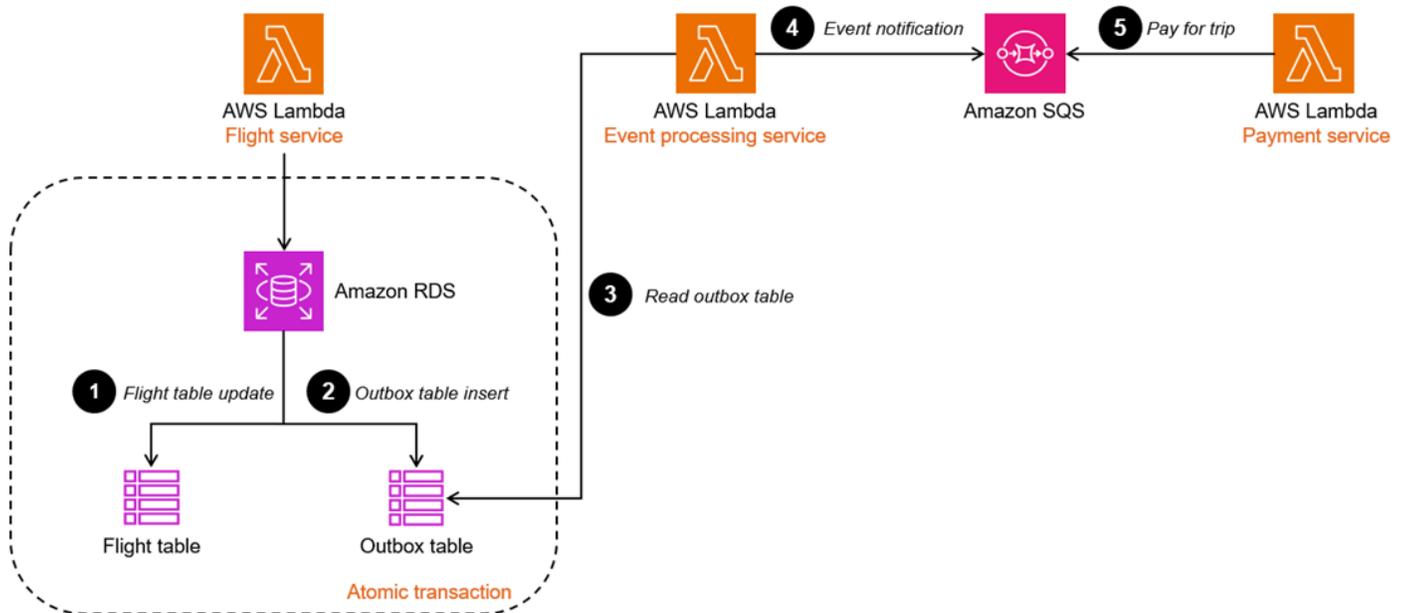
アウトボックステーブルには、フライトサービスからのすべてのイベントがタイムスタンプおよびシーケンス番号とともに保存されます。

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。メッセージをメッセージ順序付きで 1 回だけ配信する必要がある場合は、[Amazon SQS 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。



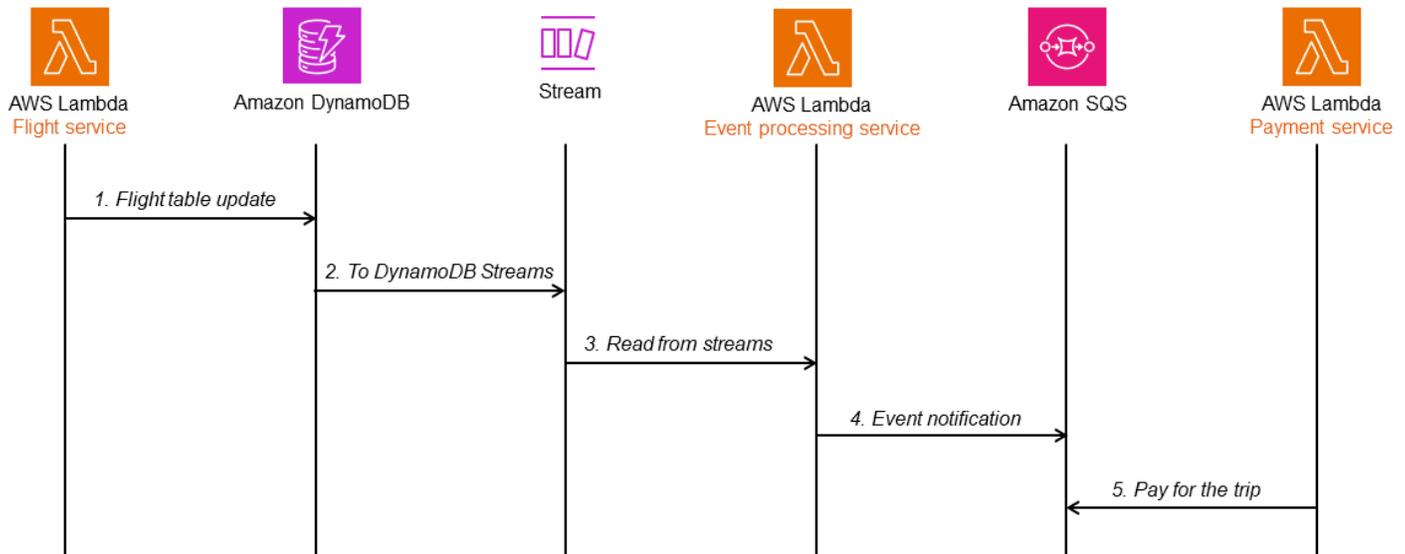
以下の図では、トランザクションアウトボックスアーキテクチャは Amazon RDS データベースを使用して実装されています。イベント処理サービスがアウトボックステーブルを読み取ると、コミットされた (成功した) トランザクションに含まれる行のみを認識し、イベントのメッセージを SQS キューに格納します。SQS キューは支払いサービスによって読み取られ、さらに処理されます。この設計では、タイムスタンプとシーケンス番号を使用して二重書き込みオペレーションの問題が解決され、メッセージとイベントの順序が保持されます。



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。

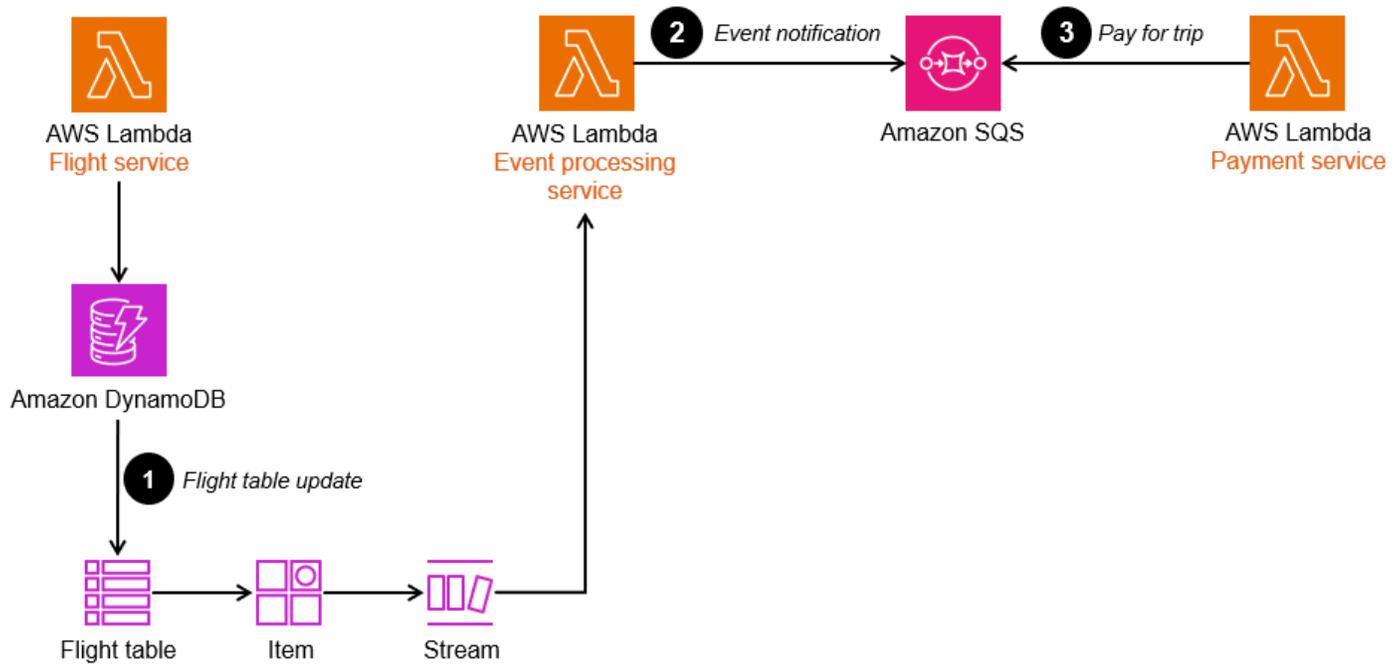
[Amazon DynamoDB](#) は CDC 更新をサポートするキー値 NoSQL データベースです。次のシーケンス図では、DynamoDB はアイテムレベルの変更を Amazon DynamoDB Streams に発行しています。イベント処理サービスはストリームから読み取り、さらに処理するためにイベント通知を支払いサービスに発行します。



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。完全なコードを表示するには、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.findAllByOrderByOrderIdAsc(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。完全なコードを表示するには、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) コードスニペットは、DynamoDB のフライトテーブルと Amazon Kinesis データストリーム (`cdcStream`) を作成し、すべての更新をストリームに送信するようにフライトテーブルを設定します。

```
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は、Kinesis ストリーム内の更新を取得し、これらのイベントを SQS キューに転送してさらに処理する Spound cloud stream 関数を定義します。

```
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> の 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](#)
- [アマゾン・ビルダース・ライブラリー](#)

Tools

- [AWS Well-Architected Tool](#)
- [AWS アプリケーション 2 コンテナ](#)
- [AWS Microservice Extractor for .NET](#)

方法論

- [トゥエルブ・ファクター・アプリ](#)(アダム・ウィギンズによるePub)
- マイケル・T・ナイガード [リリースしろ！: 本番環境に対応したソフトウェアの設計と展開2](#). 第2版 ノースカロライナ州ローリー: プラグマティックブックシェルフ、2018年。
- [ポリグロットパーシスタンス](#)(マーティン・ファウラーによるブログ投稿)
- [StranglerFigApplication](#)(マーティン・ファウラーによるブログ投稿)

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 散布図 の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	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トランザクションアウトボックスパターンに追加しました。	2024年2月23日
新しいコード例	<ul style="list-style-type: none">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

更新

[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、2023 年 7 月 28 日
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。

AWS 規範的ガイドの用語集

以下は、AWS 規範的ガイドが提供する戦略、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: オンプレミスの Oracle データベースを AWS クラウドの Oracle 用 Amazon Relational Database Service (Amazon RDS) に移行します。
- 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: AWS クラウドの EC2 インスタンスでオンプレミスの Oracle データベースを Oracle に移行します。
- 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。この移行シナリオは、オンプレミス環境と間の仮想マシン (VM) の互換性とワークロードの移植性 AWS をサポートする VMware Cloud on に固有のもので AWS。AWS の VMware Cloud にインフラを移行する際、オンプレミスのデータセンターから VMware Cloud Foundation のテクノロジーを使用することができます。例: Oracle データベースをホストするハイパーバイザーを VMware Cloud on に再配置します 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、およびそれらを行移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。

- 使用停止 — 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コントロール」](#)を参照してください。

抽象化されたサービス

[「マネージドサービス」](#)を参照してください。

ACID

[「原子性、一貫性、分離性、耐久性」](#)を参照してください。

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](#) よりも柔軟性がありますが、より多くの作業が必要です。

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の方法。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

行のグループを操作し、グループの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」](#)を参照してください。

AI Ops

[「人工知能オペレーション」](#)を参照してください。

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。

アプリケーションコントロール

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](#)の需要要素であり、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?](#)」を参照してください。

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。

アベイラビリティゾーン

他のアベイラビリティゾーンの障害から 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)

組織がクラウドに正常に移行 AWS するための効率的で効果的な計画を立てるのに役立つ、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。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編成します。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、組織がクラウド導入を成功させるための準備に役立つ、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

[「事業継続計画」](#)を参照してください。

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[Data in a behavior graph](#)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。[エンディアンネス](#)も参照してください。

二項分類

バイナリ結果 (2 つの可能なクラスのうちの一つ) 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

2 つの異なる同一の環境を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は 1 つの環境 (青) で実行し、新しいアプリケーションバージョンは他の環境 (緑) で実行します。この戦略は、影響を最小限に抑えながら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インタラクション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

トの中には、個人や組織に混乱を与えたり、損害を与えたりすることを意図しているものがあります。

ボットネット

[マルウェア](#)に感染し、[ボット](#)のヘルダーまたはボットオペレーターと呼ばれる、単一関係者の管理下にあるボットのネットワーク。ボットは、ボットとその影響をスケールするための最もよく知られているメカニズムです。

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、「[ブランチについて](#) (GitHub ドキュメント)」を参照してください。

ブレイクグラスアクセス

例外的な状況や承認されたプロセスを通じて、ユーザーが通常アクセス許可を持たない AWS アカウントにすばやくアクセスできるようになります。詳細については、Well-Architected [ガイド](#)の「[ブレイクグラス手順の実装](#)」インジケータ AWS を参照してください。

ブラウフィールド戦略

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、ホワイトペーパー [AWSでのコンテナ化されたマイクロサービスの実行](#) 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。](#)

Canary デプロイ

エンドユーザーへのバージョンの低速かつ増分的なリリース。確信できたら、新しいバージョンをデプロイし、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

[「データキャプチャの変更」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します。[AWS Fault Injection Service \(AWS FIS \)](#) を使用して、AWS ワークロードに負荷をかけてレスポンス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に、ローカルでデータを暗号化します。

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 の投稿](#)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](#)テクノロジー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 クラウドに移行する際に通常実行する 4 つのフェーズ：

- 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、クラウド AWS 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。移行戦略とどのように関連しているかについては、AWS [「移行準備ガイド」](#)を参照してください。

CMDB

[「設定管理データベース」](#)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 または含まれます AWS CodeCommit。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ます。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などのビジュアル形式から情報を分析および抽出する [AI](#) の分野。例えば、はオンプレミスのカメラネットワークに CV を追加するデバイス AWS Panorama を提供し、Amazon SageMaker は CV の画像処理アルゴリズムを提供します。

設定ドリフト

ワークロードの場合、設定は想定した状態から変化します。これにより、ワークロードが非標準になる可能性があり、通常は段階的かつ意図的ではありません。

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 リージョンに、または組織全体に 1 つのエンティティとしてデプロイできます。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

一元化された管理とガバナンスにより、分散型の分散型データ所有権を提供す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していることを確認できます。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通常、大量の履歴データが含まれており、クエリや分析によく使用されます。

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る。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。

ディープラーニング

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。

defense-in-depth

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。この戦略をに採用するときは AWS、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て、リソースの安全性を確保します。例えば、defense-in-depth 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で利用できるサービス](#)を参照してください。

デプロイメント

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、Implementing security controls on AWSの[Detective controls](#)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ユファクチャリング・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](#) では、ファクトテーブル内の量的データに関するデータ属性を含む小さなテーブル。ディメンションテーブル属性は通常、テキストフィールドまたはテキストのように動作する離散数値です。これらの属性は、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一般的に使用されます。

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)

[災害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に使用する戦略とプロセス](#)。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の追跡。例えば、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 ガバナンス要件への準拠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

[「開発値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。

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、応答時間を短縮できます。

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プロセス。

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 (アカウンティング、[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利用できる、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](#) の中央テーブル。事業運営に関する定量的データを保存します。通常、ファクトテーブルには、メジャーを含む列とディメンションテーブルへの外部キーを含む列の 2 種類の列が含まれます。

フェイルファスト

頻繁で段階的なテストを使用して開発ライフサイクルを短縮する哲学。これはアジャイルアプローチの重要な部分です。

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向上させるアベイラビリティゾーン 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です。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。

FGAC

[「きめ細かなアクセスコントロール」](#) 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

段階的なアプローチを使用するのではなく、[変更データキャプチャ](#)による継続的なデータレプリケーションを使用して、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。

G

Geo Blocking

[「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)

Amazon では CloudFront、特定の国のユーザーがコンテンツディストリビューションにアクセスできないようにするオプションです。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 ワークフローはレガシーと見なされ、[トランクベースのワークフロー](#)はモダンで推奨されるアプローチです。

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、Amazon AWS Security Hub、GuardDuty、Amazon Inspector AWS Trusted Advisor、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。](#)

ハイアベイラビリティ (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。

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。緊急性のため、通常、修正は一般的な DevOps リリースワークフローの外で行われます。

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

[「産業モノのインターネット」](#) 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

既存のインフラストラクチャを更新、パッチ適用、または変更するのではなく、本番ワークロード用の新しいインフラストラクチャをデプロイするモデル。イミュータブルなインフラストラクチャは、[本質的にミュータブルなインフラストラクチャ](#) よりも一貫性、信頼性、予測性が高くなります。詳細については、AWS Well-Architected フレームワークの[「変更不可能なインフラストラクチャを使用したデプロイ」](#) 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、アプリケーション外からの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

セッ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

接続、リアルタイムデータ、自動化、分析、AI/ML の進歩を通じて、のビジネスプロセスのモダナイゼーションを指すために 2016 年に [Klaus Schwab](#) によって導入された用語。

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。

産業分野における 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、「[Building an industrial Internet of Things \(IIoT\) digital transformation strategy](#)」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、「[AWS を使用した機械学習モデルの解釈](#)」を参照してください。

IoT

「[モノのインターネット](#)」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。

大規模な移行

300 台以上のサーバの移行。

LBAC

[「ラベルベースのアクセスコントロール」](#) 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

[「7R」](#) 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。[エンディアンネス](#) も参照してください。

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またはプライバシーを侵害するように設計されているソフトウェア。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、不正アクセスにつながる

可能性があります。マルウェアの例としては、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 のサービスが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、ユーザーが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。Amazon Simple Storage Service (Amazon S3) と Amazon DynamoDB は、マネージドサービスの例です。これら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、制御するためのソフトウェアシステム。このソフトウェアシステムは、加工品を現場の完成製品に変換します。

MAP

[「移行促進プログラム」](#) を参照してください。

メカニズム

ツールを作成し、ツールの導入を推進し、調整のために結果を検査する完全なプロセス。メカニズムは、動作中にそれ自体を強化して改善するサイクルです。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

内の組織の一部である管理アカウント AWS アカウントを除くすべての 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。

メッセージキューイングテレメトリトランスポート (MQTT)

リソースに制約のある IoT デバイス用の、[パブリッシュ/サブスクライブ](#) パターンに基づく軽量の machine-to-machine (M2M) 通信プロトコル。

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)

組織がクラウドへの移行のための強固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、オペレーション、ビジネスアナリストと所有者、移行エンジニア、デベロッパー、スプリントに取り組む 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には、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: Application Migration Service を使用して Amazon EC2 AWS への移行をリホストします。

Migration Portfolio Assessment (MPA)

AWS クラウドに移行するためのビジネスケースを検証するための情報を提供するオンラインツール。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されるアプローチ。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、「」の「[アプリケーションをモダナイズするための戦略 AWS クラウド](#)」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せん。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

[「移行ポートフォリオ評価」](#)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。

変更可能なインフラストラクチャ

本番ワークロードの既存のインフラストラクチャを更新および変更するモデル。Well-Architected AWS Framework 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

[「オリジンアクセスコントロール」](#)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

「[運用レベルの契約](#)」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。

オープンプロセス通信 - 統合アーキテクチャ (OPC-UA)

産業オートメーション用の machine-to-machine (M2M) 通信プロトコル。OPC-UA は、データの暗号化、認証、認可スキームを備えた相互運用性標準を提供します。

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)

インシデントや潜在的な障害の理解、評価、防止、または範囲の縮小に役立つ質問および関連するベストプラクティスのチェックリスト。詳細については、AWS Well-Architected フレームワークの「[運用準備状況レビュー \(ORR\)](#)」を参照してください。

運用テクノロジー (OT)

産業運用、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するハードウェアおよびソフトウェアシステム。製造では、OT と情報技術 (IT) システムの統合が、[Industry 4.0](#) トランスフォーメーションの主要な焦点です。

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

の組織 AWS アカウント 内のすべての のすべてのイベントをログ AWS CloudTrail 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、ドキュメントの「[組織の証跡の作成](#)」を参照してください。CloudTrail

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から、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。

オリジンアクセスコントロール (OAC)

では CloudFront、Amazon Simple Storage Service (Amazon S3) コンテンツを保護するためのアクセスを制限するための拡張オプションです。OAC は、すべての のすべての S3 バケット AWS リージョン、AWS KMS (SSE-KMS) によるサーバー側の暗号化、および 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)

では CloudFront、Amazon S3 コンテンツを保護するためのアクセスを制限するオプションです。OAI を使用する場合は、Amazon S3 が認証できるプリンシパル CloudFront を作成します。認証されたプリンシパルは、特定の CloudFront ディストリビューションを介してのみ S3 バケット内のコンテンツにアクセスできます。[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コントロールが可能です。

ORR

[「運用準備状況レビュー」](#)を参照してください。

OT

[「運用技術」](#)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る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

[「プログラム可能な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

アクセス許可の定義 ([アイデンティティベースのポリシー](#) を参照)、アクセス条件の指定 ([リソースベースのポリシー](#) を参照)、または の組織内のすべてのアカウントに対する最大アクセス許可の定義 AWS Organizations ([サービスコントロールポリシー](#) を参照) が可能なオブジェクト。

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。詳細については、[マイクロサービスでのデータ永続性の有効化](#)を参照してください。

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、「[移行準備状況ガイド](#)」を参照してください。

述語

true または を返すクエリ条件。false 通常は WHERE 句にあります。

述語のプッシュダウン

転送前にクエリ内のデータをフィルタリングするデータベースクエリ最適化手法。これにより、リレーショナルデータベースから取得して処理する必要があるデータの量が減少し、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、Implementing security controls on AWSの[Preventative controls](#)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 ロール AWS アカウント、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](#)内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

エンジニアリングプロセス全体を通してプライバシーを考慮に入れた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

非準拠のリソースのデプロイを防止するように設計された[セキュリティコントロール](#)。これらのコントロールは、プロビジョニング前にリソースをスキャンします。リソースがコントロールに準拠していない場合、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「[での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)

設計、開発、発売から成長と成熟まで、製品のデータとプロセスのライフサイクル全体にわたる管理。

本番環境

[「環境」](#)を参照してください。

プログラミング可能ロジックコントローラー (NAL)

製造では、マシンをモニタリングし、承認プロセスを自動化する、信頼性が高く、適応性の高いコンピュータです。

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。

パブリッシュ/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。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](#)では、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できるチャンネルにイベントメッセージを発行できます。システムは、公開サービスを変更せずに新しいマイクロサービスを追加できます。

Q

クエリプラン

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る手順などの一連のステップ。

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

[責任、説明責任、相談、情報 \(RACI\)](#) 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

[責任、説明責任、相談、情報 \(RACI\)](#) を参照してください。

RCAC

[「行と列のアクセスコントロール」](#) を参照してください。

リードレプリカ

読み取り専用で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。

再構築

[「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、他のとは分離され、独立しています。詳細については、[AWS リージョン「を使用できるアカウントを指定する」](#)を参照してください。

回帰

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

[「7R」](#)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

[「7R」](#)を参照してください。

プラットフォーム変更

[「7R」](#)を参照してください。

再購入

[「7R」](#)を参照してください。

回復性

中断に耐えたり、中断から回復したりするアプリケーションの機能。で障害耐性を計画する場合、[高可用性](#)と[ディザスタリカバリ](#)が一般的な考慮事項です AWS クラウド。詳細については、[AWS クラウド「レジリエンス」](#)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。サポートを含めると、そのマトリックスは RASCI マトリックスと呼ばれ、サポートを除外すると RACI マトリックスと呼ばれます。

レスポンスコントロール

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、Implementing security controls on AWSの[Responsive controls](#)を参照してください。

保持

[「7R」を参照してください。](#)

廃止

[「7R」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](#)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 (IdPs) が使用するオープンスタンダード。この機能により、フェデレーテ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 [AWS](#)

Management Console したり、組織内のすべてのユーザーを IAM で作成しなくても 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

[「監視コントロールとデータ収集」](#)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

では AWS Secrets Manager、暗号化された形式で保存される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。シークレット値は、バイナリ、1つの文字列、または複数の文字列にすることができます。詳細については、[Secrets Manager](#) ドキュメントの「シークレット」を参照してください。

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、[予防的](#)、[検出的](#)、[???応答的](#)、[プロアクティブ](#)の4つの主なタイプがあります。

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に自動的に応答または修正するように設計された、事前定義されたプログラムされたアクション。これらのオートメーションは、セキュリティのベストプラクティスの実装に役立つ[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としては、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、認証情報のローテーションなどがあります。

サーバー側の暗号化

送信先にあるデータの、それを受け取る AWS のサービス による暗号化。

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

のエンドポイントの 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、スループットなど、サービスのパフォーマンス側面の測定。

サービスレベルの目標 (SLO)

サービス [レベルのインジケータ](#) によって測定される、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、お客様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

[「セキュリティ情報とイベント管理システム」](#)を参照してください。

単一障害点 (SPOF)

システムを中断させる可能性のあるアプリケーションの単一の重要なコンポーネントの障害。

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。

split-and-seed モデル

モダナイゼーションプロジェクトのスケール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、[「」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 AWS クラウド」](#)を参照してください。

SPOF

[単一障害点](#)を参照してください。

star スキーマ

トランザクションデータまたは測定データを保存するために1つの大きなファクトテーブルを使用し、データ属性を保存するために1つ以上の小さなディメンションテーブルを使用するデータベースの組織構造。この構造は、[データウェアハウス](#)またはビジネスインテリジェンスの目的で使用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ゾーンに存在する必要があります。

監視コントロールとデータ収集 (SCADA)

製造では、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生産オペレーション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

ユーザーインタラクションをシミュレートして潜在的な問題を検出したり、パフォーマンスをモニタリングしたりする方法でシステムをテストします。[Amazon CloudWatch Synthetics](#) を使用してこれらのテストを作成できます。

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が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 と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するサービスへの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と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「[AWS Organizations を他の AWS のサービスで使用する 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 つのピザを食べることができる小さな 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

現在のレコードに関連する行のグループに対して計算を実行する SQL 関数。ウィンドウ関数は、移動平均の計算や、現在の行の相対位置に基づく行の値へのアクセスなどの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

[「書き込み 1 回」](#)を参照し、[多くの](#)を読み取ります。

WQF

[「AWS ワークロード認定フレームワーク」](#)を参照してください。

Write Once, Read Many (WORM)

データを 1 回書き込み、データの削除や変更を防ぐストレージモデル。承認されたユーザーは、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 [イミュータブルな](#) 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](#) を利用する攻撃、通常はマルウェア。

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。

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。