でのマイクロフロントエンドの理解と実装 AWS

AWS 規範ガイダンス

AWS 規範ガイダンス: でのマイクロフロントエンドの理解と実装 AWS

Table of Contents

でのマイクロフロントエンドの理解と実装 AWS

Amazon Web Services (寄稿者)

2024 年 7 月 (ドキュメント履歴)

組織が俊敏性とスケーラビリティを求めるにつれて、従来のモノリシックアーキテクチャがボトルネックになり、迅速な開発とデプロイが妨げられることがよくあります。マイクロフロントエンドは、複雑なユーザーインターフェイスを、自律的に開発、テスト、デプロイできるより小さな独立したコンポーネントに分割することで、これを軽減します。このアプローチにより、開発チームの効率が向上し、バックエンドとフロントエンド間のコラボレーションが容易になり、分散システムのend-to-endの連携が促進されます。

この規範的なガイダンスは、さまざまなプロフェッショナルドメインの IT リーダー、製品所有者、アーキテクトがマイクロフロントエンドアーキテクチャを理解し、アマゾン ウェブ サービス () でマイクロフロントエンドアプリケーションを構築できるように作成されていますAWS。

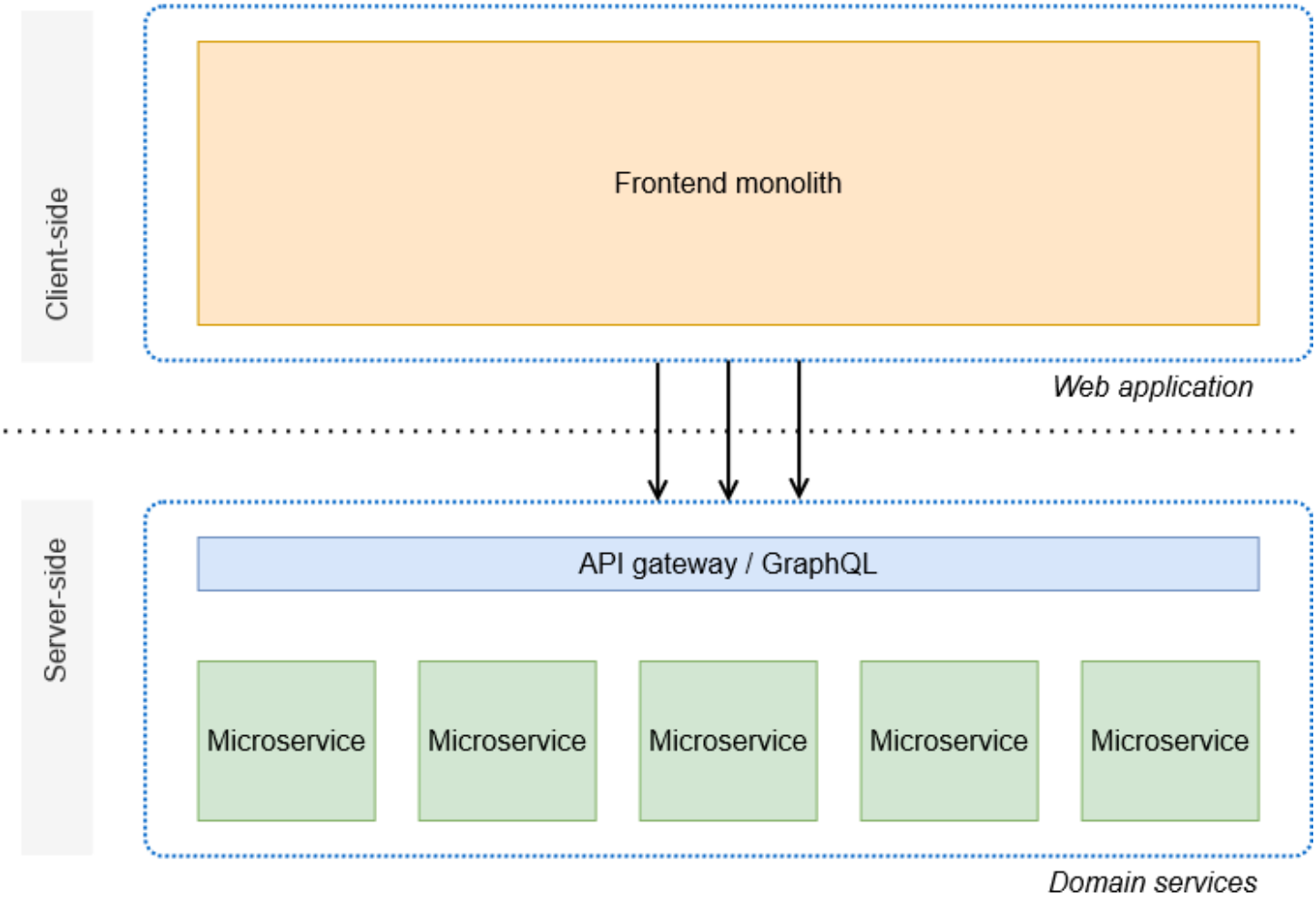
概要:

マイクロフロントエンドは、アプリケーションフロントエンドを独立して開発およびデプロイされたアーティファクトに分解するために構築されたアーキテクチャです。大きなフロントエンドを自律的なソフトウェアアーティファクトに分割すると、ビジネスロジックをカプセル化し、依存関係を減らすことができます。これにより、製品増分をより迅速かつ頻繁に配信できます。

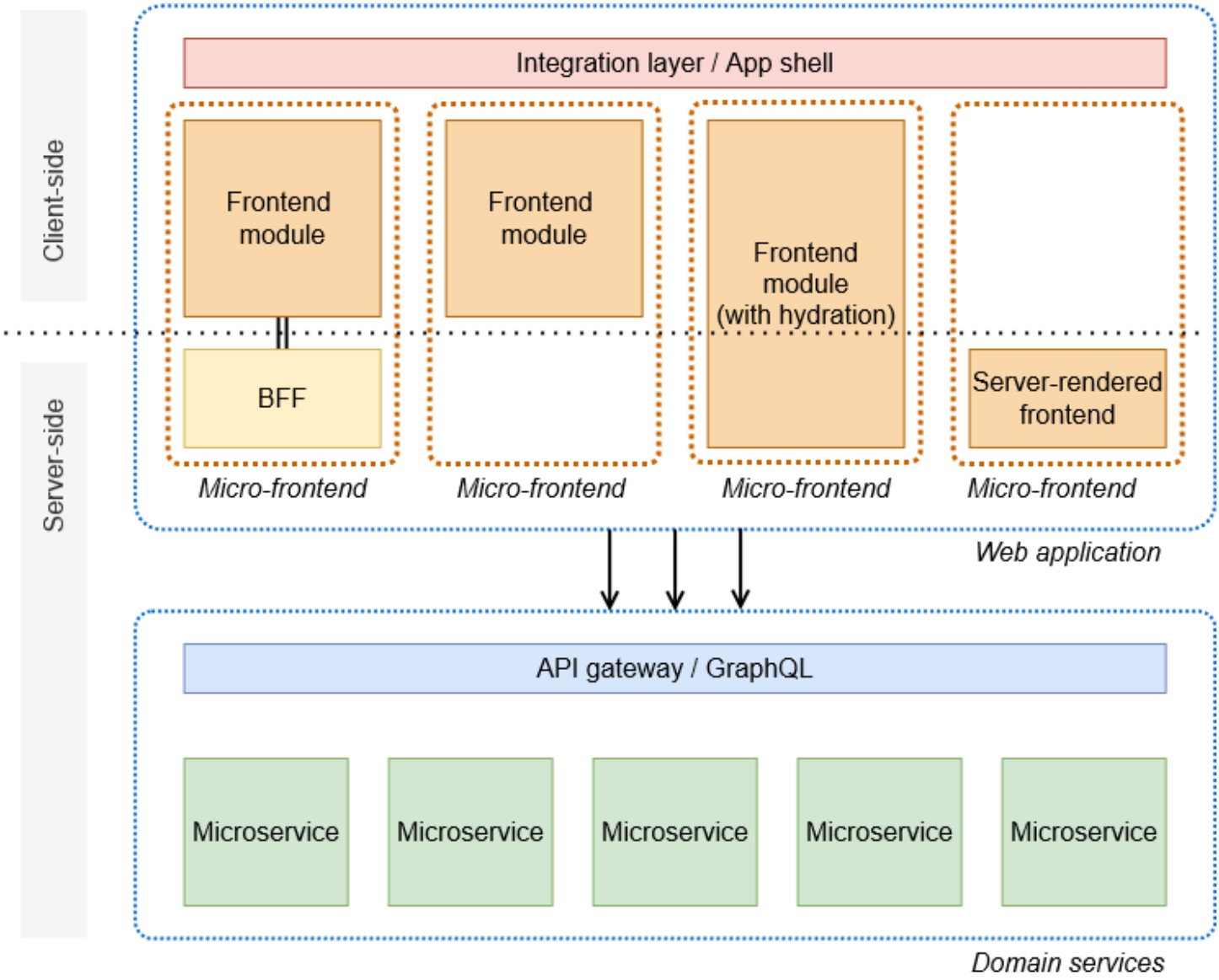
マイクロフロントエンドはマイクロサービスに似ています。実際、マイクロフロントエンドという用語はマイクロサービスという用語から派生しており、マイクロサービスの概念をフロントエンドとして伝えることを目的としています。マイクロサービスアーキテクチャは通常、バックエンドの分散システムとモノリシックフロントエンドを組み合わせますが、マイクロフロントエンドは自己完結型の分散フロントエンドサービスです。これらのサービスは、次の 2 つの方法で設定できます。

- フロントエンドのみ、 がマイクロサービスアーキテクチャを実行する共有 API レイヤーと統合
- フルスタック、つまり各マイクロフロントエンドには独自のバックエンド実装があります。

次の図は、API ゲートウェイを使用してバックエンドマイクロサービスに接続するフロントエンドモノリスを備えた従来のマイクロサービスアーキテクチャを示しています。



次の図は、マイクロサービスの実装が異なるマイクロフロントエンドアーキテクチャを示しています。



前の図に示すように、クライアント側のレンダリングまたはサーバー側のレンダリングアーキテクチャでマイクロフロントエンドを使用できます。

- クライアント側でレンダリングされたマイクロフロントエンドは、一元化APIs Gateway によって公開された API を直接使用できます。
- チームは、境界コンテキスト内にbackend-for-frontend (BFF) を作成して、APIs に対するフロントエンドの混雑を軽減できます。
- サーバー側では、マイクロフロントエンドは、ハイドレーションと呼ばれる手法を使用してクライアント側で拡張されたサーバー側のアプローチで表現できます。ブラウザによってページがレンダリングされると、関連する JavaScript がハイドレートされ、ボタンのクリックなどの UI 要素とのやり取りが可能になります。

- マイクロフロントエンドはバックエンドでレンダリングでき、ハイパーリンクを使用してウェブサイトの新しい部分にルーティングできます。

マイクロフロントエンドは、以下を実行する組織に最適です。

- 同じプロジェクトに取り組む複数のチームでスケールします。
- 意思決定の分散を受け入れ、デベロッパーが特定されたシステム境界内でイノベーションできるようにします。

このアプローチにより、チームの認知負荷が大幅に軽減されます。これは、チームがシステムの特定の部分を担当するためです。残りの部分を中断することなくシステムの一部に変更を加えることができるため、ビジネスの俊敏性が向上します。

マイクロフロントエンドは、個別のアーキテクチャアプローチです。マイクロフロントエンドを構築するにはさまざまな方法がありますが、それらにはすべて共通の特性があります。

- マイクロフロントエンドアーキテクチャは、複数の独立した要素で構成されています。この構造は、バックエンドのマイクロサービスで発生するモジュール化に似ています。
- マイクロフロントエンドは、以下で構成される境界コンテキスト内のフロントエンド実装に完全に責任を負います。
  - [ユーザーインターフェイス]
  - データ
  - 状態またはセッション
  - ビジネスロジック
  - フロー

境界コンテキストは、入出力を仲介する慎重に設計された境界を持つ内部整合性のあるシステムです。マイクロフロントエンドは、ビジネスロジックやデータをできるだけ他のマイクロフロントエンドと共有しないでください。共有が必要な場所では、カスタムイベントやリアクティブストリームなどの明確に定義されたインターフェイスを介して行われます。ただし、設計システムやログ記録ライブラリなど、クロスカットに関する懸念がある場合は、意図的に共有することをお勧めします。

推奨されるパターンは、クロスファンクショナルチームを使用してマイクロフロントエンドを構築することです。つまり、各マイクロフロントエンドは、バックエンドからフロントエンドまで作業する同じチームによって開発されます。コーディングから本番環境でのシステムの運用化まで、チームの所有権は重要です。

このガイダンスは、特定のアプローチを推奨するものではありません。代わりに、さまざまなパターン、ベストプラクティス、トレードオフ、アーキテクチャと組織の考慮事項について説明します。

基本的な概念

マイクロフロントエンドアーキテクチャは、3 つの以前のアーキテクチャ概念から大きく影響を受けています。

- ドメイン駆動型設計は、複雑なアプリケーションを一貫したドメインに構造化するためのメンタルモデルです。
- 分散システムは、独自に開発され、独自の専用インフラストラクチャで実行される疎結合サブシステムとしてアプリケーションを構築するためのアプローチです。
- クラウドコンピューティングは、IT インフラストラクチャを pay-as-you-go モデルでサービスとして実行するためのアプローチです。

ドメイン駆動型設計

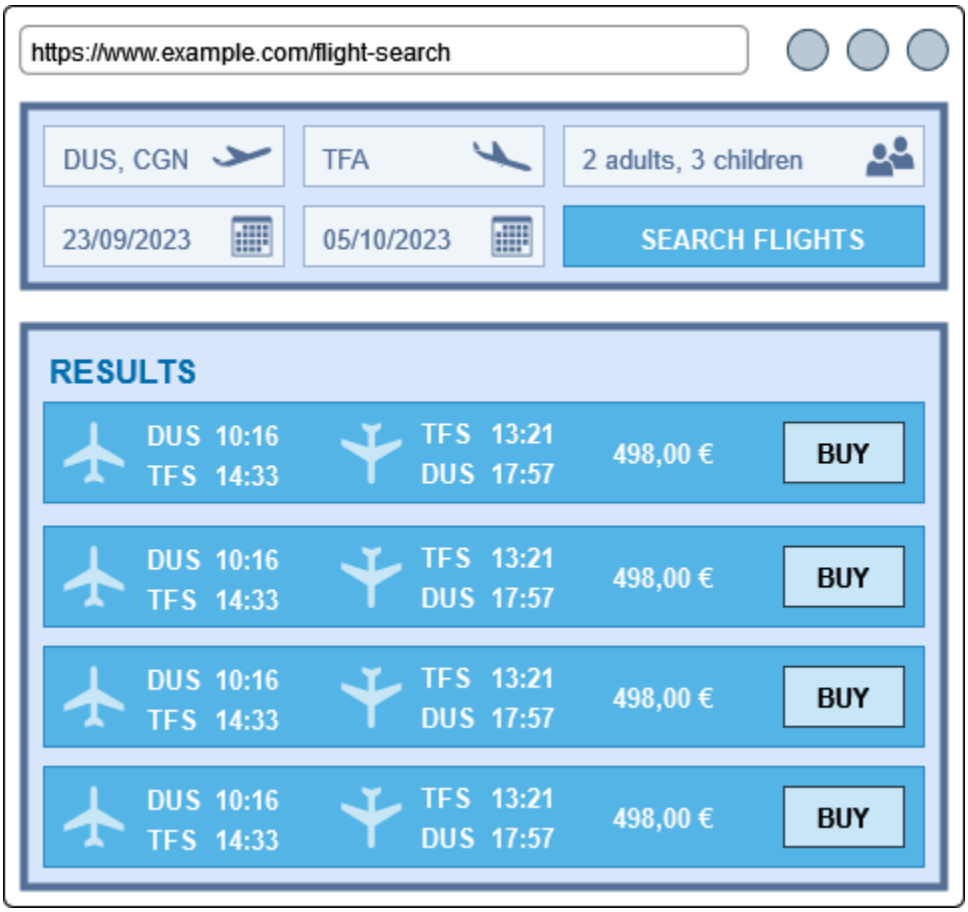
ドメイン駆動型設計 (DDD) は、Eric Evans によって開発されたパラダイムです。2003 年の本「Domain-Driven Design: Tackling Complexity in the Heart of Software」では、Evans はソフトウェア開発は技術的な問題ではなくビジネス上の懸念によって推進されるべきだと想定しています。Evans は、IT プロジェクトが最初に、技術専門家とドメイン専門家が共通の理解を見つけるのに役立つユビキタスな言語を開発することを提案します。その言語に基づいて、ビジネス上の現実について相互に理解されたモデルを策定できます。

そのアプローチは明らかですが、多くのソフトウェアプロジェクトはビジネスと IT の切断に悩まされています。これらの切断は、多くの場合、大きな誤解を引き起こし、予算の超過、品質の低下、またはプロジェクトの失敗につながります。

Evans では、他の複数の重要な用語が導入されています。そのうちの 1 つが境界コンテキスト です。境界付きコンテキストは、1 つのビジネス上の懸念に対するソリューションまたは実装を含む大規模な IT アプリケーションの自己完結型セグメントです。大規模なアプリケーションは、統合パターンを介して疎結合された複数の境界コンテキストで構成されます。これらの境界コンテキストは、ユビキタス言語の独自の方言を持つこともできます。例えば、アプリケーションの支払いコンテキストのユーザーは、配送の概念が支払い中に無関係になるため、配送コンテキストのユーザーとは異なる側面を持つ場合があります。

Evans は、境界コンテキストの規模を定義しません。サイズはソフトウェアプロジェクトによって決定され、時間の経過とともに進化する可能性があります。コンテキストの境界を示す優れた指標は、エンティティ (ドメインオブジェクト) とビジネスロジックの間の結合度です。

マイクロフロントエンドのコンテキストでは、フライト予約ページなどの複雑なウェブページの例でドメイン駆動型設計を説明できます。



このページでは、主な構成要素は検索フォーム、フィルターパネル、結果リストです。境界を特定するには、独立した機能コンテキストを特定する必要があります。さらに、再利用性、パフォーマンス、セキュリティなどの非機能的な側面も考慮してください。「一緒に属するモノ」という最も重要な指標は、コミュニケーションパターンです。アーキテクチャの一部の要素が頻繁に通信し、複雑な情報を交換する必要がある場合、同じ境界コンテキストを共有する可能性があります。

ボタンなどの個々の UI 要素は機能的に独立していないため、境界コンテキストにはなりません。また、ページ全体は、より小さな独立したコンテキストに分割できるため、境界コンテキストには適していません。妥当な方法は、検索フォームを 1 つの境界コンテキストとして扱い、結果リストを 2 番目の境界コンテキストとして扱うことです。これら 2 つの境界コンテキストをそれぞれ個別のマイクロフロントエンドとして実装できるようになりました。

分散システム

メンテナンスを容易にし、進化する機能をサポートするために、単純な IT ソリューションの大部分はモジュール式です。この場合、モジュール式とは、IT システムが識別可能な構成要素で構成され、インターフェイスを介して分離され、懸念を分離することを意味します。

分散システムは、モジュール型であることに加えて、独自の独立したシステムである必要があります。単にモジュラーシステムの場合、各モジュールは理想的にはカプセル化され、インターフェイスを介してその関数を公開しますが、個別にデプロイしたり、単独で機能させたりすることはできません。また、モジュールは通常、同じシステムの一部である他のモジュールと同じライフサイクルに従います。一方、分散システムの構成要素には、それぞれ独自のライフサイクルがあります。ドメイン駆動型設計パラダイムを適用すると、各構成要素は 1 つのビジネスドメインまたはサブドメインに対処し、独自の境界コンテキストに属します。

構築中に分散システムが相互作用する場合、一般的なアプローチは、問題をすばやく特定するためのメカニズムを開発することです。例えば、型付き言語を採用して、ユニットテストに多額の投資を行う場合があります。複数のチームがモジュールの開発とメンテナンスに共同作業できます。多くの場合、npm、Apache Maven、 NuGetpip などのツールでシステムを消費するためのライブラリとして配布されます。

実行時には、対話型の分散システムは通常、個々のチームによって所有されます。依存関係を消費すると、エラー処理、パフォーマンスバランシング、セキュリティのために運用が複雑になります。統合テストとオブザーバビリティの成果は、リスクを軽減するために不可欠です。

今日の分散システムの最も一般的な例はマイクロサービスです。マイクロサービスアーキテクチャでは、バックエンドサービスは (UI や認証などの技術的な懸念によって駆動されるのではなく) ドメイン駆動型であり、自律型チームによって所有されます。マイクロフロントエンドは同じ原則を共有し、ソリューションの範囲をフロントエンドに拡張します。

クラウドコンピューティング

クラウドコンピューティングは、独自のデータセンターを構築し、オンプレミスで運用するためのハードウェアを購入するのではなく、 pay-as-you-go モデルを使用して IT インフラストラクチャをサービスとして購入する方法です。クラウドコンピューティングにはいくつかの利点があります。

- 組織は、大規模で長期的な財務上のコミットメントを事前に行うことなく、新しいテクノロジーを試すことができるため、ビジネスの俊敏性が大幅に向上します。
- などのクラウドプロバイダーを使用することで AWS、組織はメンテナンスが少なく、高度に統合されたサービス (API ゲートウェイ、データベース、コンテナオーケストレーション、クラウド機

能など) の幅広いポートフォリオにアクセスできます。これらのサービスにアクセスすると、組織は競合と区別される作業に集中できるようになります。

- 組織がソリューションをグローバルに展開する準備ができたら、ソリューションを世界中のクラウドインフラストラクチャにデプロイできます。

クラウドコンピューティングは、高度に管理されたインフラストラクチャを提供することで、マイクロフロントエンドをサポートします。これにより、部門横断的なチームが end-to-end 所有権を簡単に取得できます。チームは運用に関する強力な知識を持っている必要がありますが、インフラストラクチャのプロビジョニング、オペレーティングシステムの更新、ネットワークなどの手動タスクは面倒な作業になります。

マイクロフロントエンドは制限されたコンテキストで動作するため、チームはそれらを実行するのに最適なサービスを選択できます。例えば、チームはコンピューティング用にクラウド関数とコンテナから選択でき、さまざまな種類の SQL データベースと NoSQL データベース、またはインメモリキャッシュを選択できます。チームは、サーバーレスインフラストラクチャ用に事前設定された構成要素が付属している AWS Amplifyなどの高度に統合されたツールキットでマイクロフロントエンドを構築することもできます。

マイクロフロントエンドと代替アーキテクチャの比較

すべてのアーキテクチャ戦略と同様に、マイクロフロントエンドを採用する決定は、組織の原則に基づく評価基準に基づいている必要があります。マイクロフロントエンドには利点と欠点があります。組織がマイクロフロントエンドを使用することを決定した場合、分散システムの課題に対処するための戦略を設定する必要があります。

アプリケーションアーキテクチャを選択する場合、マイクロフロントエンドの最も一般的な代替手段は、モノリス、n 層アプリケーション、マイクロサービスとシングルページアプリケーション (SPA) フロントエンドの組み合わせです。これらはすべて有効なアプローチであり、それぞれに利点と欠点があります。

モノリス

頻繁な変更を必要としない小さなアプリケーションは、モノリスとして非常に迅速に配信できます。大幅な成長が予想される状況でも、モノリスは自然な最初のステップです。その後、モノリスをリタイアするか、より柔軟な構造にリファクタリングできます。モノリスから始めることで、組織は市場に進出し、顧客のフィードバックを得て、製品を迅速に改善できます。

ただし、モノリシックアプリケーションは、慎重に保守されていない場合や、コードベースのサイズが時間の経過とともに大きくなると、低下する傾向があります。複数のチームが同じコードベースに大きく貢献する場合、すべてがメンテナンスと運用に寄与することはほとんどありません。これにより責任のバランスが崩れ、速度に影響し、非効率になります。同時に、モノリスのモジュールが誤って結合されると、コードベースが進化するにつれて意図しない副作用が発生します。これらの副作用により、誤動作や機能停止が発生する可能性があります。

N 層アプリケーション

比較的静的な進化ペースを持つより複雑なアプリケーションは、フロントエンドとバックエンドの間に REST または GraphQL レイヤーを持つ 3 層アーキテクチャ (表現、アプリケーション、データ) として構築できます。これははるかに柔軟であり、異なる階層のチームはある程度独立して開発できます。n 層アプリケーションの欠点は、機能のデプロイがはるかに難しいことです。フロントエンドとバックエンドは API 契約によって分離されるため、重大な変更を一緒にデプロイするか、API をバージョニングする必要があります。

次の一般的なシナリオを検討してください。新機能をリリースする際にデータスキーマの変更が必要な場合、製品所有者がフロントエンドチームと一連の機能について合意するまでに数日かかることが

あります。次に、フロントエンドチームはバックエンドチームに、ユーザー側で機能の開発とリリースを依頼します。バックエンドチームは、データ所有者と協力してデータベーススキーマの更新をリリースします。次に、バックエンドチームは API の新しいバージョンをリリースし、フロントエンドチームが変更を開発してリリースできるようにします。このシナリオでは、各チームに変更の開発、テスト、リリースに関する独自のバックログ、優先順位、メカニズムがあるため、本番環境へのすべての変更の伝播に数週間または数か月かかる場合があります。

マイクロサービス

マイクロサービスアーキテクチャでは、バックエンドは小さなサービスに分解され、それぞれが制限付きコンテキスト内の特定のビジネス上の懸念に対処します。各マイクロサービスは、明確に定義されたインターフェイス契約を公開することで、他のサービスとの密接な分離も行われます。

境界コンテキストとインターフェイスコントラクトは、適切に作成されたモノリスと n 層アーキテクチャにも存在する必要があります。ただし、マイクロサービスアーキテクチャでは、通信はネットワーク、通常は HTTP プロトコルを介して行われ、サービスには専用のランタイムインフラストラクチャがあります。これにより、各バックエンドサービスの開発、配信、運用が個別にサポートされます。

要件に対するアプローチの選択

モノリスと n 層アーキテクチャは、複数のドメインの懸念を 1 つの技術アーティファクトにバンドルします。これにより、依存関係や内部データフローなどの側面の管理が容易になりますが、新しい機能の提供はより困難になります。一貫したコードベースを維持するために、チームは多くの場合、リファクタリングとデカップリングに時間を費やします。これは、処理する必要のあるコードベースが大きいためです。

一部のチームによって開発されたアプリケーションは、マイクロフロントエンドへの移行に伴う追加の複雑さを必要としない場合があります。これは、チームが高い結合とリリース変更への長いリードタイムのペナルティを支払っていない場合に特に当てはまります。

つまり、より複雑で分散型のアーキテクチャは、複雑で動きの速いアプリケーションに適しています。小規模から中規模のアプリケーションでは、特にアプリケーションが短期間で劇的に進化しない場合、分散アーキテクチャがモノリシックアーキテクチャよりも必ずしも優れているとは限りません。

マイクロフロントエンドにおけるアーキテクチャ上の意思決定

マイクロフロントエンドアーキテクチャパターンをアプリケーションに適用するチームは、アーキテクチャに関するいくつかの決定を早期に行う必要があります。

- マイクロフロントエンドの識別と境界の定義
- マイクロフロントエンドを使用したページとビューの作成
- マイクロフロントエンド間のルーティング、状態管理、通信
- クロスカットの懸念に対する依存関係の管理

以下のセクションでは、これらのトピックについて詳しく説明します。

アーキテクチャを決定するときは、正しいメトリクスを持ち、アプリケーションの特性とトレードオフの使用パターンを理解することが重要です。たとえば、e コマースサイトには、動画編集ツールやオブザーバビリティダッシュボードとは異なる特性と使用パターンがあります。

トラフィックが多く、セッション深度が短い公開アプリケーションは、Time to Interactive (TTI) や First Contentful Paint (FCP) などの初期ページロードメトリクスに合わせて最適化できます。対照的に、ユーザーが 1 日の開始時にログインし、1 日を通して とやり取りし続けるアプリケーションは、アプリケーション内エクスペリエンスに最適化される可能性があります。アプリケーションチームは、最初のページロードではなく、各ナビゲーションの後に最初の入力遅延 (FID) メトリクスを最適化する場合があります。

パブリックウェブサイトは、さまざまなブラウザ環境に対応している必要があります。クライアント環境に既知の制約があるエンタープライズアプリケーションは、その制約に従ってマイクロフロントエンド構成を最適化できます。

アーキテクチャの決定には、1 つの適切な選択肢はありません。トレードオフ、ビジネスの運営状況、使用パターン、メトリクスを理解して、個々のアプリケーションに適した決定を導き出します。

マイクロフロントエンド境界の特定

チームの自律性を向上させるために、アプリケーションが提供するビジネス機能は、相互に最小限の依存関係で複数のマイクロフロントエンドに分解できます。

前述の DDD 方法論に従って、チームはアプリケーションドメインをビジネスサブドメインと境界コンテキストに分割できます。その後、自律チームは境界コンテキストの機能を所有し、それらのコンテキストをマイクロフロントエンドとして配信できます。

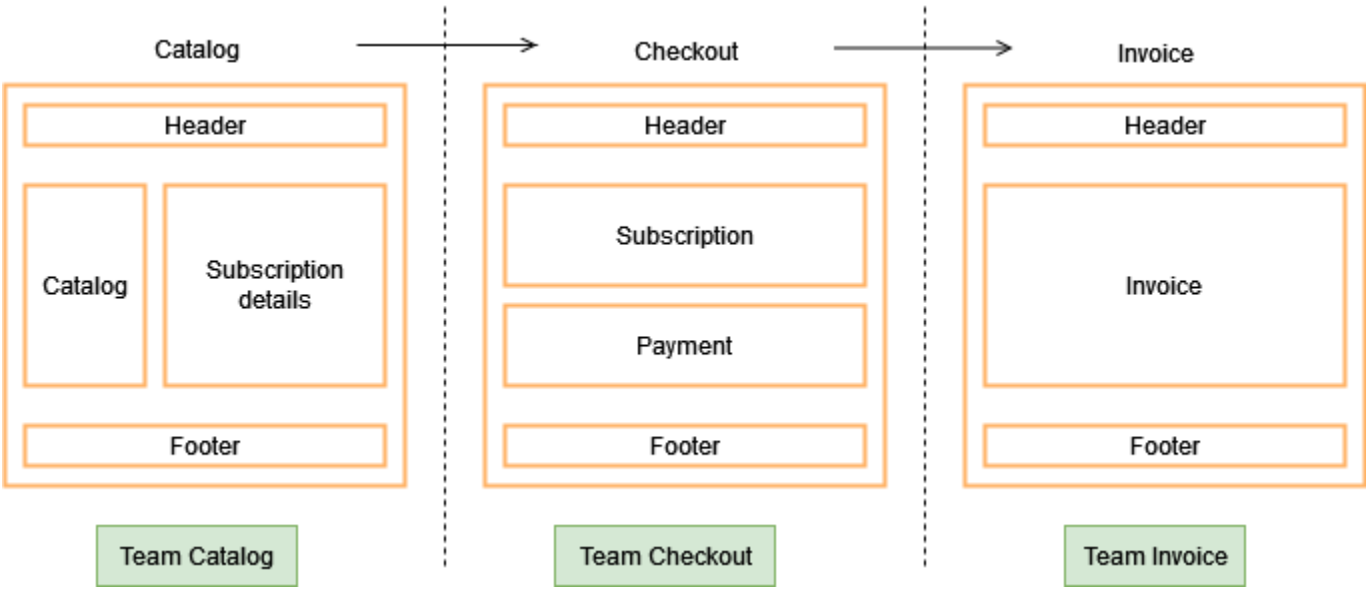
明確に定義された境界コンテキストは、機能的な重複とコンテキスト間のランタイム通信の必要性を最小限に抑える必要があります。必要な通信は、イベント駆動型メソッドで実装できます。これは、マイクロサービス開発のイベント駆動型アーキテクチャとは異なります。

適切に設計されたアプリケーションは、新しいチームによる将来の拡張機能の提供もサポートし、お客様に一貫したエクスペリエンスを提供する必要があります。

モノリシックアプリケーションをマイクロフロントエンドにスライスする方法

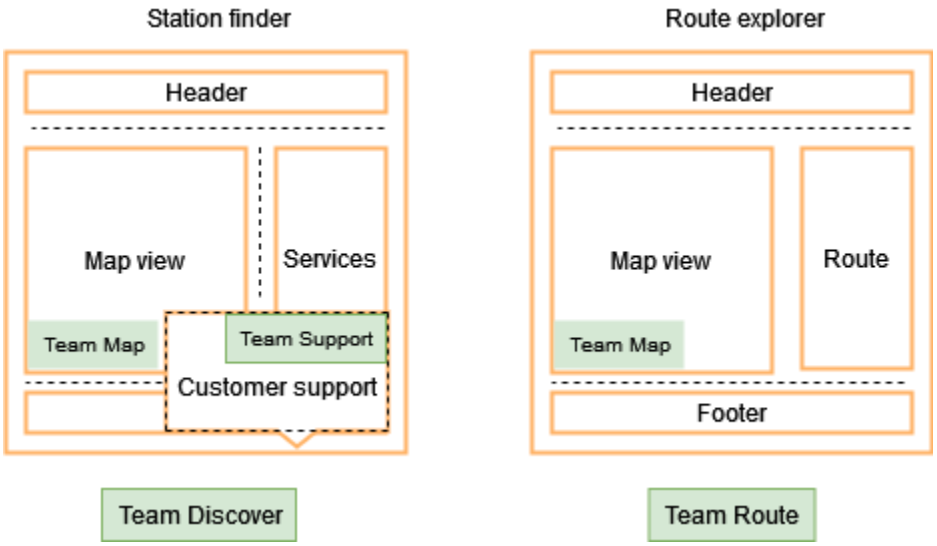
概要セクションには、ウェブページで独立した機能コンテキストを識別する例が含まれています。ユーザーインターフェイスの機能を分割するためのいくつかのパターンが表示されます。

たとえば、ビジネスドメインがユーザージャーニーのステージを形成する場合、フロントエンドの垂直分割を適用できます。この場合、ユーザージャーニーのビューのコレクションはマイクロフロントエンドとして配信されます。次の図は、カタログ、チェックアウト、請求書の各ステップが別々のマイクロフロントエンドとして別々のチームによって配信される垂直分割を示しています。



一部のアプリケーションでは、垂直分割だけでは十分ではない場合があります。たとえば、一部の機能を多くのビューで提供する必要がある場合があります。これらのアプリケーションでは、混合分割

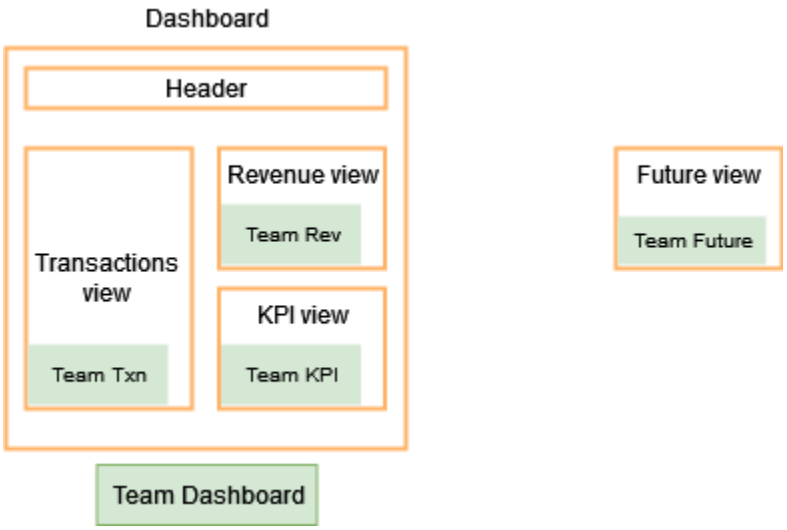
を適用できます。次の図は、ステーションファインダーとルートエクスプローラーのマイクロフロントエンドの両方がマップビュー機能を使用する混合分割ソリューションを示しています。



ポータルタイプまたはダッシュボードタイプのアプリケーションでは、通常、フロントエンド機能が 1 つのビューにまとめられます。これらのタイプのアプリケーションでは、各ウィジェットをマイクロフロントエンドとして配信でき、ホスティングアプリケーションはマイクロフロントエンドが実装する必要がある制約とインターフェイスを定義します。

このアプローチは、ビューポートのサイズ設定、認証プロバイダー、構成設定、メタデータなどの懸念をマイクロフロントエンドが処理するメカニズムを提供します。これらのタイプのアプリケーションは、拡張性のために最適化されます。新機能は、ダッシュボード機能をスケールするために新しいチームによって開発できます。

次の図は、チームダッシュボードの一部である 3 つのチームによって開発されたダッシュボードアプリケーションを示しています。



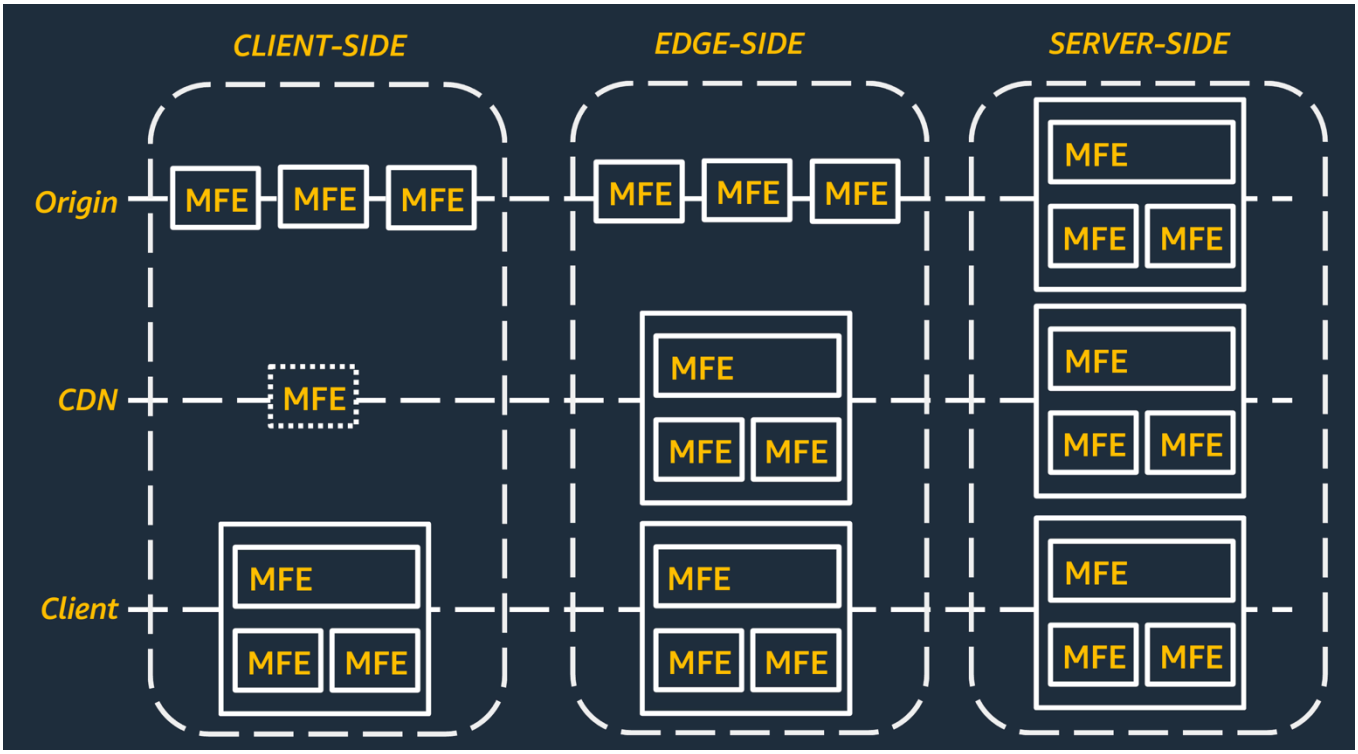
この図では、将来のビューは、チームダッシュボードとダッシュボード機能をスケールするために新しいチームが開発した新機能を表しています。

ポータルアプリケーションとダッシュボードアプリケーションは通常、UI で混合分割を使用して機能を構成します。マイクロフロントエンドは、位置やサイズの制約など、明確に定義された設定で設定できます。

マイクロフロントエンドを使用したページとビューの作成

クライアント側のコンポジション、エッジ側のコンポジション、サーバー側のコンポジションを使用して、アプリケーションのビューを作成できます。構成パターンには、必要なチームスキル、耐障害性、パフォーマンス、キャッシュ動作の点で異なる特性があります。

次の図は、マイクロフロントエンドアーキテクチャのクライアント側、エッジ側、サーバー側のレイヤーで構成がどのように行われるかを示しています。



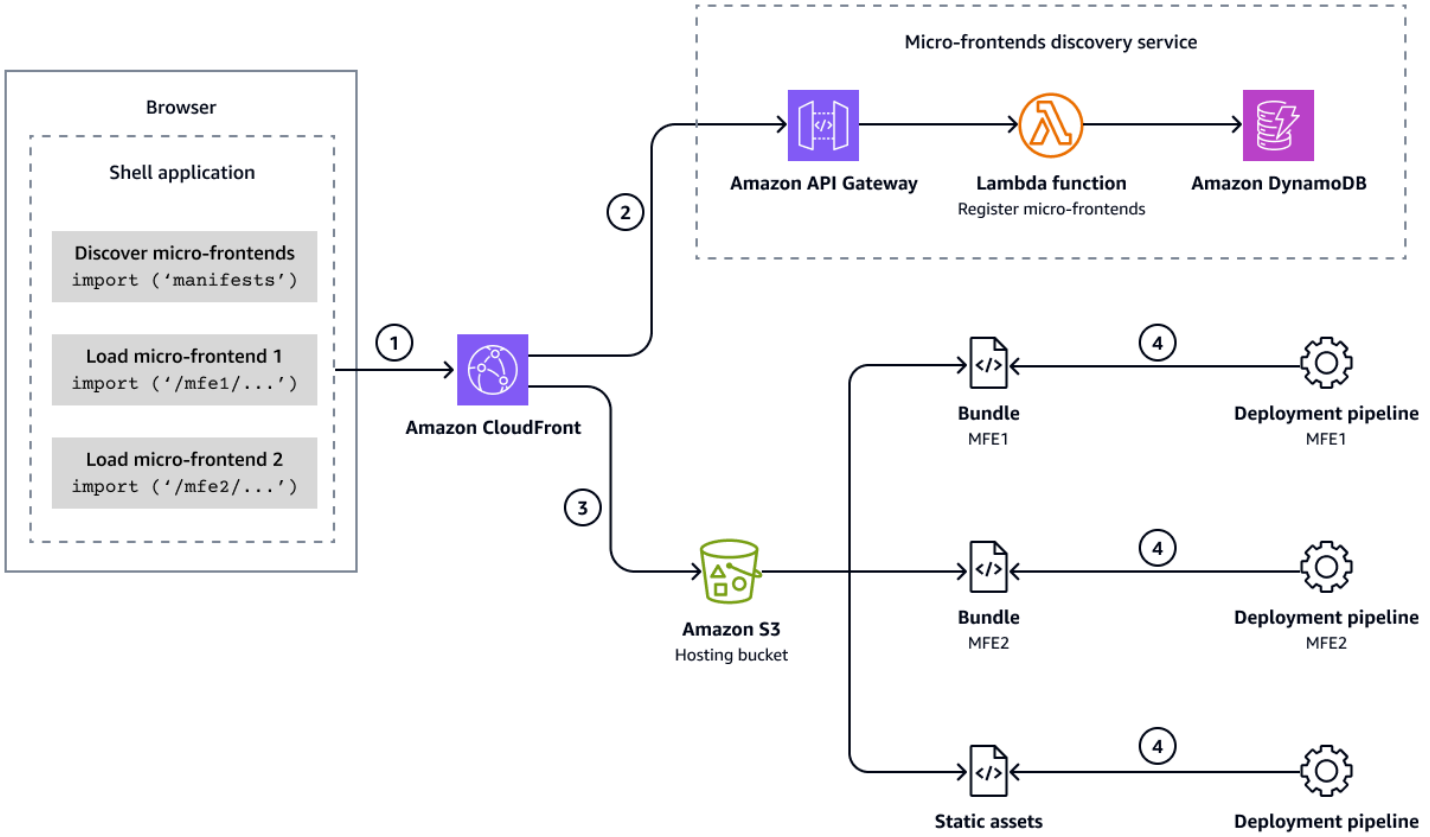
クライアント側、エッジ側、サーバー側のレイヤーについては、以下のセクションで説明します。

クライアントサイドコンポジション

クライアント (ブラウザまたはモバイルウェブビュー) でドキュメントオブジェクトモデル (DOM) フラグメントとしてマイクロフロントエンドを動的にロードして追加します。JavaScript ファイルや CSS ファイルなどのマイクロフロントエンドアーティファクトは、コンテンツ配信ネットワーク (CDNs) からロードしてレイテンシーを短縮できます。クライアント側の構成には、以下が必要です。

- シェルアプリケーションまたはマイクロフロントエンドフレームワークを所有および維持し、ブラウザで実行時にマイクロフロントエンドコンポーネントを検出、ロード、レンダリングできるようにするチーム
- HTML、CSS、JavaScript などのフロントエンドテクノロジーの高度なスキルレベルと、ブラウザ環境の深い理解
- ページにロードされる JavaScript の量の最適化と、グローバル名前空間の競合を回避するための統制

次の図は、サーバーレスクライアント側の構成の AWS アーキテクチャの例を示しています。



クライアント側のコンポジションは、シェルアプリケーションを介してブラウザ環境で行われます。この図は、次の詳細を示しています。

1. シェルアプリケーションがロードされると、Amazon CloudFront に最初のリクエストを行い、マニフェストエンドポイントを介してロードされるマイクロフロントエンドを検出します。
2. マニフェストには、各マイクロフロントエンドに関する情報 (名前、URL、バージョン、フォールバック動作など) が含まれます。マニフェストは、マイクロフロントエンド検出サービスによって提供されます。この図では、この検出サービスは Amazon API Gateway、 AWS Lambda 関数、および Amazon DynamoDB で表されます。シェルアプリケーションはマニフェスト情報を使用して、個々のマイクロフロントエンドに特定のレイアウト内のページの作成をリクエストします。
3. 各マイクロフロントエンドバンドルは、静的ファイル (JavaScript、CSS、HTML など) で構成されます。ファイルは Amazon Simple Storage Service (Amazon S3) バケットでホストされ、CloudFront を介して提供されます。
4. チームは、所有するデプロイパイプラインを使用して、新しいバージョンのマイクロフロントエンドをデプロイし、マニフェスト情報を更新できます。

エッジサイドコンポジション

オリジンサーバーの前の一部の CDNs やプロキシでサポートされるエッジサイドインクルード (GIS) やサーバーサイドインクルード (SSI) などのトランスクルージョン手法を使用して、クライアントにワイヤ経由で送信する前にページを作成します。VoIP には以下が必要です。

- DDoS 機能を備えた CDN、またはサーバー側のマイクロフロントエンドの前にプロキシをデプロイします。HAProxy、Varnish、NGINX などのプロキシ実装は SSI をサポートしています。
- VoIP および SSI 実装の使用と制限の理解。

通常、新しいアプリケーションを開始するチームは、コンポジションパターンにエッジサイドコンポジションを選択しません。ただし、このパターンは、トランスクルージョンに依存するレガシーアプリケーションのパスを提供する可能性があります。

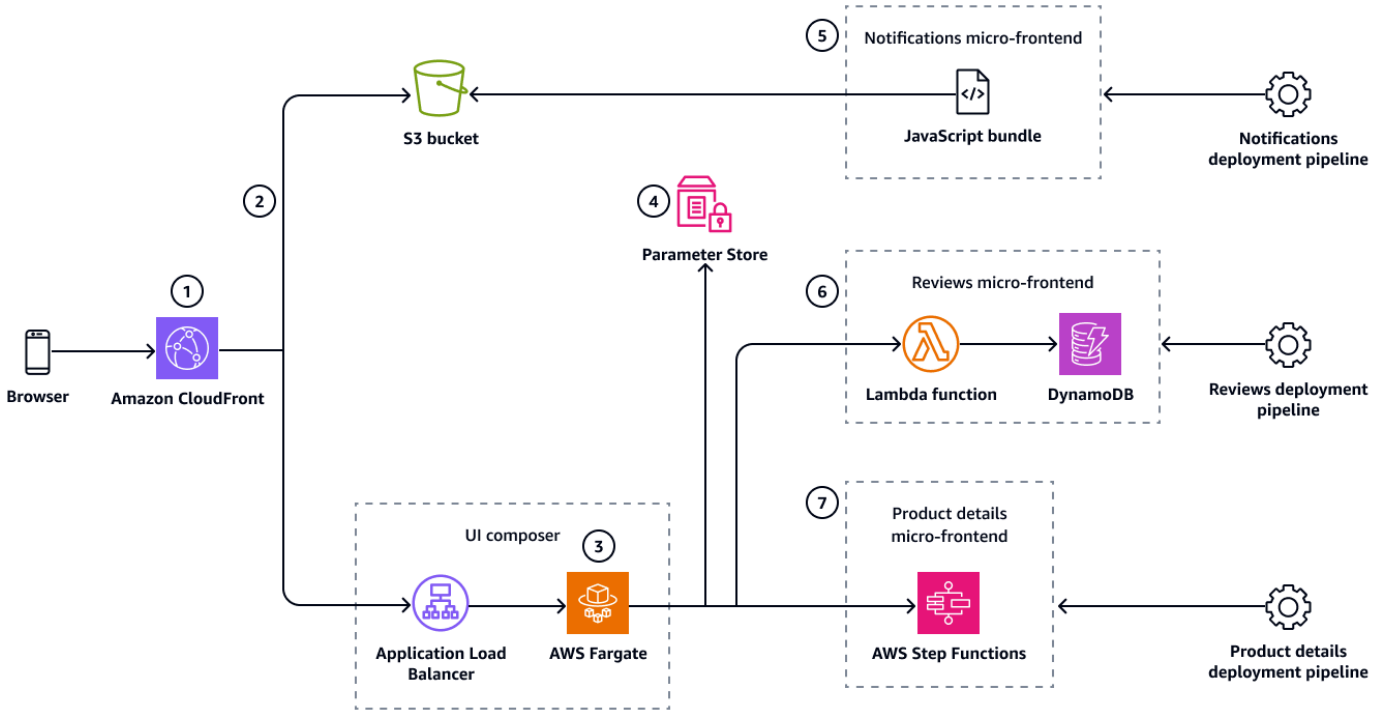
サーバーサイドコンポジション

オリジンサーバーを使用して、エッジにキャッシュされる前にページを作成します。これは、PHP、Jakarta Server Pages (JSP)、テンプレートライブラリなどの従来のテクノロジーを使用して、マイクロフロントエンドのフラグメントを含めることでページを構成することができます。サーバーで実行されている Next.js などの JavaScript フレームワークを使用して、サーバー側のレンダリング (SSR) でサーバー上のページを作成することもできます。

ページがサーバーにレンダリングされたら、CDNsにキャッシュしてレイテンシーを短縮できます。新しいバージョンのマイクロフロントエンドをデプロイする場合、ページを再レンダリングし、キャッシュを更新して最新バージョンをお客様に配信する必要があります。

サーバー側の構成では、デプロイ、サーバー側のマイクロフロントエンドの検出、キャッシュ管理のパターンを確立するために、サーバー環境を深く理解する必要があります。

次の図は、サーバー側の構成を示しています。



この図には、次のコンポーネントとプロセスが含まれています。

1. Amazon CloudFront は、アプリケーションに一意のエントリポイントを提供します。ディストリビューションには 2 つのオリジンがあります。1 つ目は静的ファイル用、2 つ目は UI コンポーザー用です。
2. 静的ファイルは Amazon S3 バケットでホストされます。これらは、HTML テンプレートのブラウザと UI コンポーザーによって消費されます。
3. UI コンポーザーは、 のコンテナクラスターで実行されますAWS Fargate。コンテナ化されたソリューションでは、必要に応じてストリーミング機能とマルチスレッドレンダリングを使用できます。
4. の一機能である Parameter Store は AWS Systems Manager、基本的なマイクロフロントエンド検出システムとして使用されます。この機能は、消費するマイクロフロントエンドエンドポイントを取得するために UI コンポーザーが使用するキーバリューストアを提供します。
5. 通知マイクロフロントエンドは、最適化された JavaScript バンドルを S3 バケットに保存します。これは、ユーザーのインタラクションに反応する必要があるため、クライアントでレンダリングされます。
6. レビューのマイクロフロントエンドは Lambda 関数で構成され、ユーザーレビューは DynamoDB に保存されます。レビューのマイクロフロントエンドはサーバー側で完全にレンダリングされ、HTML フラグメントを出力します。

7. 製品詳細マイクロフロントエンドは、 を使用するローコードのマイクロフロントエンドですAWS Step Functions。Express ワークフローは同期的に呼び出すことができ、HTML フラグメントとキャッシュレイヤーをレンダリングするためのロジックが含まれています。

サーバー側の構成の詳細については、ブログ記事「サーバー側のレンダリングマイクロフロントエンド - アーキテクチャ」を参照してください。

マイクロフロントエンド間のルーティングと通信

ルーティングオプションは、構成アプローチによって異なります。フロントエンドコンポーネント間の結合を減らすことで、通信を最適化できます。

ルーティング

垂直分割でクライアント側のコンポジションを使用するアプリケーションは、サーバー側のルーティング (複数ページのアプリケーション) またはクライアント側のルーティング (単一ページのアプリケーション) を使用できます。UI コンポジションに混合分割を使用する場合、ページ上のマイクロフロントエンドのより深いルーティング階層をサポートするには、クライアント側のルーティングが必要です。

エッジ側のコンポジションとサーバー側のコンポジションを使用するアプリケーションは、サーバー側のルーティング、または Amazon CloudFront での Lambda@Edge などのエッジコンピューティングによるルーティングにより適しています。

マイクロフロントエンド間の通信

マイクロフロントエンドアーキテクチャでは、フロントエンドコンポーネント間の結合を減らすことをお勧めします。結合を減らす方法の 1 つは、同期関数呼び出しから非同期メッセージングに移行することです。

ブラウザランタイムとユーザーインタラクションは、本質的に非同期です。イベントは、メッセージを通じてプロデューサーとコンシューマーの間で交換できます。イベントは、マイクロフロントエンド間で通信するための明確に定義されたインターフェイスを提供します。

DDD プラクティスに従ってマイクロフロントエンドの境界コンテキストを特定する場合、次のステップは境界を越えて通信する必要があるイベントを特定することです。

イベントのメッセージングメカニズムは、ネイティブ DOM イベント (`CustomEvents`)、JavaScript イベントエミッタ、またはプラットフォームチームが提供するリアクティブストリームライブラリで

す。マイクロフロントエンドはイベントを発行し、境界コンテキストに関連するイベントにサブスクライブします。この方法では、パブリッシャーとサブスクライバーが相互に認識する必要はありません。契約はイベント定義です。これの視覚的表現については、イベントアーキテクチャ図の境界コンテキストの「イベントとの通信」セクションを参照してください。 https://eda-visuals.boyney.io/visuals/bounded-context-with-event-architectures

クロスカットの懸念に対する依存関係の管理

マイクロフロントエンドなどの分散アーキテクチャを成功させるには、意識的な依存関係管理が不可欠です。依存関係管理は、マイクロフロントエンド開発の最も困難な部分の 1 つです。

マイクロフロントエンドアーキテクチャでは、依存関係管理の 2 つの重要な側面として、大きなコードアーティファクトをクライアントに転送することによるパフォーマンス上のペナルティと、コンピューティングリソースのオーバーヘッドがあります。理想的には、組織は分散フロントエンドアーキテクチャの依存関係を維持する方法を義務付ける必要があります。

依存関係のメンテナンスを必須にするための 3 つの実行可能な戦略は、インポートマップやモジュールフェデレーションなどのウェブ標準を使用して、何も共有しないことです。他のアプローチは、分散アーキテクチャの基本原則に違反するため、アンチパターンです。

可能な場合は何も共有しない

共有なしのアプローチでは、独立したソフトウェアアーティファクト間の依存関係をまったく共有しないこと、または少なくとも統合時やランタイム時に共有しないことを前提としています。つまり、2 つのマイクロフロントエンドが同じライブラリに依存している場合、それぞれがビルド時にライブラリ内でベークし、個別に出荷する必要があります。また、各マイクロフロントエンドは、ライブラリがグローバル名前空間と共有リソースをポーリングしないことを検証する必要があります。

これは冗長性につながりますが、最大の俊敏性を備えた意識的なトレードオフです。ランタイムの依存関係を共有しない場合、チームはソリューションの範囲内でソフトウェアを進化させ、インターフェイス契約を破らない限り、どのような方法でもソフトウェアを進化させる最大限の柔軟性が得られます。

マイクロフロントエンドが共有なしの原則に従うプラットフォームでは、マイクロフロントエンドをできるだけ軽量に保つことが重要です。これには、パフォーマンスのためにマイクロフロントエンドを最適化することに熟練し、開発者エクスペリエンスのためにユーザーエクスペリエンスを犠牲にしない開発者が必要です。

コードを共有する場合

コードの共有を決定するときは、ライブラリまたはランタイムモジュールとして共有できます。たとえば、フロントエンドコアチームは CDNs を介してマイクロフロントエンド消費用のライブラリを提供します。ビジネスバリューチームは、実行時にライブラリをロードすることも、パッケージリポジトリを使用してライブラリを公開することもできます。マイクロフロントエンドチームは、ハイブリッドフレームワークを使用するモバイルアプリケーションと同様に、ビルド時にパッケージ化されたライブラリの特定のバージョンに対して開発できます。

3 番目のオプションは、プライベートパッケージレジストリを使用して、共通ライブラリのビルドタイム統合をサポートすることです。これにより、ライブラリ契約の変更が実行時にエラーを開始するリスクが軽減されます。ただし、このより保守的なアプローチでは、すべてのマイクロフロントエンドを新しいライブラリバージョンと同期させるためのガバナンスを強化する必要があります。

ページのロード時間を短縮するために、マイクロフロントエンドは Amazon CloudFront などの CDN からキャッシュされたチャンクからロードするライブラリの依存関係を外部化できます。

ランタイムの依存関係を管理するために、マイクロフロントエンドはインポートマップ (または などのライブラリ`System.js`) を使用して、実行時に各モジュールがどこからロードされるかを指定できます。webpack Module Federation は、リモートモジュールのホストバージョンをポイントし、独立したマイクロフロントエンド間で一般的な依存関係を解決するもう 1 つのアプローチです。

もう 1 つの方法は、検出エンドポイントへの初期リクエストを使用してインポートマップの動的ロードを容易にすることです。

共有状態

マイクロフロントエンドの結合を減らすには、モノリシックアーキテクチャと同様に、同じビュー内のすべてのマイクロフロントエンドからアクセス可能なグローバル状態管理を避けることが重要です。例えば、グローバル Redux ストアをすべてのマイクロフロントエンドからアクセスできるようにすると、結合が増加します。

共有状態を排除するパターンは、マイクロフロントエンド内にカプセル化し、前述のように非同期メッセージと通信することです。

絶対に必要な場合は、グローバル状態に明確に定義されたインターフェイスを導入し、予期しない動作を避けるために読み取り専用共有をオプトインします。

- 垂直分割が存在する場合は、URL コンポーネントとブラウザストレージを使用して、ホスト環境から情報にアクセスできます。

- 分割が混在している場合は、DOM 標準カスタムイベントまたはイベントエミッタや双方向ストリームなどの JavaScript ライブラリを使用して、マイクロフロントエンドに情報を渡すこともできます。

マイクロフロントエンド間で複数の情報を共有する必要がある場合は、マイクロフロントエンドの境界を再検討することをお勧めします。共有の必要性は、ビジネスの進化や初期設計が不十分であることが原因である可能性があります。

また、各マイクロフロントエンドがセッション識別子を使用して必要なデータを取得するサーバー側のセッションを使用することもできます。結合を減らすには、共有状態を排除し、マイクロフロントエンド固有のセッションデータを分離することが重要です。

フレームワークとツール

Angular や Next.js などのフロントエンドフレームワークは不足していませんが、ほとんどはマイクロフロントエンドを念頭に置いて作成されていません。したがって、マイクロフロントエンドアーキテクチャの課題に対処するメカニズムが欠落することがあります。

フレームワークに関する一般的な考慮事項

このガイドは、個々のフレームワークを推奨したり比較したりすることを目的としていません。複数のマイクロフロントエンドが同じウェブアプリケーションページで実行されることが多いため、ロードとランタイムのパフォーマンスが主な懸念事項です。オーバーヘッドをできるだけ少なくするフレームワークを選択することが重要です。

フレームワークは、レンダリングレイヤーに基づいて分割されます。

- クライアント側のレンダリング (CSR)
- サーバーサイドレンダリング (SSR)

フロントエンドアーキテクチャには、静的サイト生成 (SSG) などの他の機能が含まれます。ただし、SSG は 1 回のみ実行されます。マイクロフロントエンドは主に実行時に構成されるため、CSR と SSR が主なオプションです。

クライアント側のレンダリング

CSR には、次の 2 つの一般的なオプションがあります。

- 単一 SPA フレームワーク
- モジュールフェデレーション

シングル SPA は、マイクロフロントエンドを作成するための軽量な選択肢です。同じページに複数のマイクロフロントエンドを作成し、依存関係の競合を回避するなど、マイクロフロントエンドアーキテクチャの最も一般的な課題を解決します。

モジュールフェデレーションは Webpack 5 が提供するプラグインとして開始され、さまざまなアーティファクトにわたる依存関係管理など、マイクロフロントエンドアーキテクチャにおける課題の大部分を解決します。モジュールフェデレーション 2.0 は、Rspack、webpack、esbuild とネイティブに連携し、現在は と連携します JavaScript。

フレームワークをまったく使用しないことを検討してください。caniuse.com によると、最新のブラウザは市場シェアが 98% で、カスタム要素などの機能をネイティブに提供しており、マイクロフロントエンドアプリケーションに適しています。必要に応じて、イベント伝達、国際化、またはその他の特定の懸念のために、カスタム要素を軽量ライブラリと組み合わせてください。

サーバー側のレンダリング

SSR 側では、2 つの主なオプションがより複雑です。

- Next.js などの既存のフレームワークを受け入れ、モジュールフェデレーションを使用するマイクロフロントエンドの原則を適用します。
- HTML over-the-wire を使用して、マイクロフロントエンドを表す HTML フラグメントを交換し、実行時にテンプレート内でこれらのフラグメントを構成します。このアプローチの例は、Podium です。

API 統合 – フロントエンドのバックエンド

フロントエンドのバックエンド (BFF) パターンは通常、マイクロサービス環境で使用されます。マイクロフロントエンドのコンテキストでは、BFF はマイクロフロントエンドに属するサーバー側のサービスです。すべてのマイクロフロントエンドに BFF が必要なわけではありません。ただし、BFF を使用している場合は、同じ境界コンテキスト内で実行する必要があり、他の境界コンテキスト間で共有することはできません。

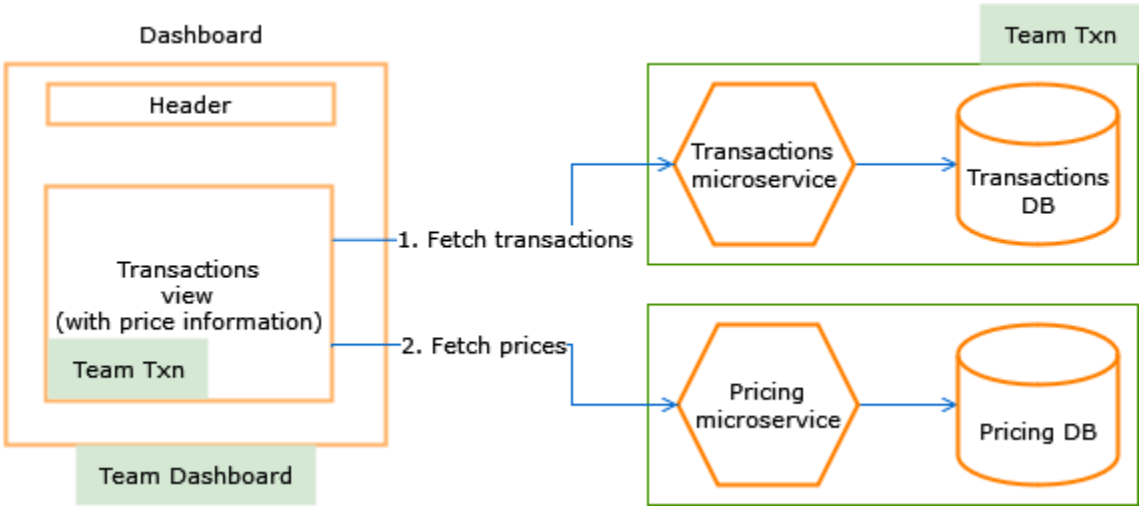
従来のサービスとは異なり、BFF はドメインモデルに従っていません。代わりに、マイクロフロントエンドがクライアントに到達する前にデータを前処理するための API レイヤーです。これが役立つ領域は次のとおりです。

- プライベート APIsに対する認可
- さまざまなソースからのデータの集約
- ネットワーク負荷を軽減し、クライアントによるデータの消費を容易にするためのデータの変換

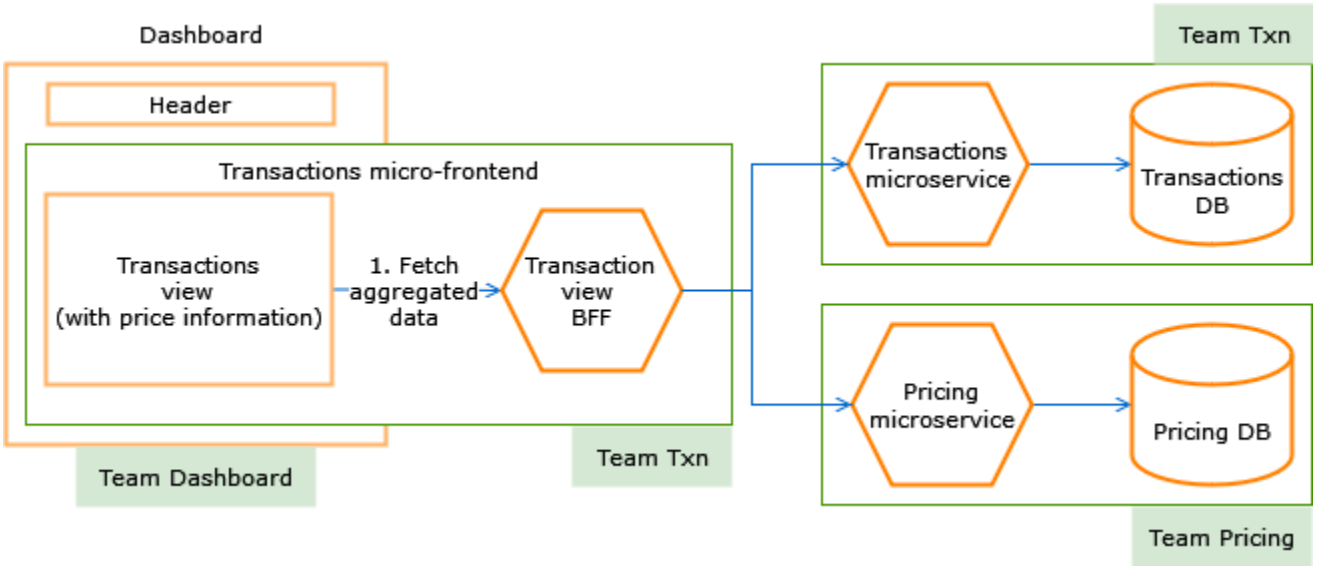
そのため、BFF はドメインサービス層ではなくマイクロフロントエンドによって所有されます。BFFsは、以下を使用してデプロイできます。

- AWS AppSync GraphQL APIs
- AWS Lambda 関数のセット
- Amazon ECS、Amazon EKS、または AWS AppRunner で実行されているコンテナとして

次の図は、BFF パターンがない場合、マイクロフロントエンドが個々のマイクロサービス API エンドポイントに接続してデータを取得および集計する必要があることを示しています。



代わりに、次の図の BFF パターンを使用すると、マイクロフロントエンドは独自のバックエンドと通信し、集約データを取得できます。



チームは、モバイル、ウェブ、または特定のビューなど、さまざまなチャネルの BFFs を開発できます。また、混雑を軽減することでバックエンドインタラクションを最適化する要件があります。

スタイルと CSS

カスケードスタイルシート (CSS) は、テキストとオブジェクトのフォーマットをハードコーディングするのではなく、ドキュメントの表示を一元的に決定するための言語です。言語のカスケード機能は、継承を使用してスタイル間の優先順位を制御するように設計されています。マイクロフロントエンドで作業し、依存関係を管理する戦略を作成する場合、言語のカスケード機能は難しい場合があります。

たとえば、2 つのマイクロフロントエンドが同じページに共存し、それぞれが HTML body 要素の独自のスタイルを定義します。それぞれが独自の CSS ファイルを取得し、`style`タグを使用して DOM にアタッチする場合、共通の HTML 要素、クラス名、または要素 IDs。これらの問題に対処するには、スタイルを管理するために選択する依存関係戦略に応じて、さまざまな戦略があります。

現在、パフォーマンス、一貫性、開発者エクスペリエンスのバランスを取る最も一般的なアプローチは、設計システムの開発と維持です。

システムを設計する – 何かを共有するアプローチ

このアプローチでは、システムを使用して、必要に応じてスタイルを共有しながら、時折分散をサポートし、一貫性、パフォーマンス、デベロッパーエクスペリエンスのバランスを取ります。設計システムは、明確な標準に基づいて再利用可能なコンポーネントのコレクションです。設計システム開発は通常、多くのチームからのインプットと貢献を持つ 1 つのチームによって推進されます。実際には、設計システムは、JavaScript ライブラリとしてエクスポートできる低レベルの要素を共有する方法です。マイクロフロントエンド開発者は、事前に作成された利用可能なリソースを作成し、新しいインターフェイスを作成するための出発点として、ライブラリを依存関係として使用してシンプルなインターフェイスを構築できます。

フォームを必要とするマイクロフロントエンドの例を考えてみましょう。一般的なデベロッパーエクスペリエンスは、設計システムで使用可能な事前に作成されたコンポーネントを使用して、テキストボックス、ボタン、ドロップダウンリスト、その他の UI 要素を構成することです。開発者は、実際のコンポーネントの外観についてのみ、スタイルを作成する必要はありません。構築およびリリースするシステムは、Webpack モジュールフェデレーションまたは同様のアプローチを使用して設計システムを外部依存関係として宣言できるため、フォームのロジックは設計システムを含めずにパッケージ化されます。

その後、複数のマイクロフロントエンドが同じ操作を行って、共有された懸念に対処できます。チームが複数のマイクロフロントエンド間で共有できる新しいコンポーネントを開発すると、それらのコンポーネントは成熟後に設計システムに追加されます。

設計システムアプローチの主な利点は、高いレベルの一貫性です。マイクロフロントエンドはスタイルを記述し、設計システムから上書きすることがありますが、その必要性はほとんどありません。主な低レベル要素は頻繁に変更されることはなく、デフォルトで拡張可能な基本的な機能を提供します。もう 1 つの利点はパフォーマンスです。構築とリリースに適した戦略では、アプリケーションシェルによって計測される最小限の共有バンドルを作成できます。複数のマイクロフロントエンド固有のバンドルがオンデマンドで非同期的にロードされ、ネットワーク帯域幅のフットプリントを最小限に抑えると、さらに改善できます。最後に、デベロッパーエクスペリエンスは理想的です。これは、ホイールを再構築することなく (ボタンがページに追加されるたびに JavaScript や CSS を記述するなど) 、豊富なインターフェイスの構築に集中できるためです。

欠点は、あらゆる種類の設計システムが依存関係であるため、維持し、場合によっては更新する必要があることです。複数のマイクロフロントエンドで共有依存関係の新しいバージョンが必要な場合は、次のいずれかを使用できます。

- 競合することなく、その共有依存関係の複数のバージョンを取得できるオーケストレーションメカニズム
- すべての依存関係を移動して新しいバージョンを使用するための共有戦略

たとえば、すべてのマイクロフロントエンドが設計システムのバージョン 3.0 に依存しており、共有方式で使用する 3.1 という新しいバージョンがある場合、すべてのマイクロフロントエンドに機能フラグを実装して、最小限のリスクで移行できます。詳細については、「機能フラグ」セクションを参照してください。もう 1 つの潜在的な欠点は、設計システムは通常、スタイルよりも多くの問題に対処することです。また、JavaScript のプラクティスとツールも含まれています。これらの側面では、議論とコラボレーションを通じて合意に達する必要があります。

設計システムの実装は、長期的な投資に適しています。これは一般的なアプローチであり、複雑なフロントエンドアーキテクチャに取り組むすべての人が検討する必要があります。通常、フロントエンドエンジニア、製品チーム、設計チームが連携し、相互にやり取りするメカニズムを定義する必要があります。目的の状態になるまでの時間をスケジュールすることが重要です。また、長期的に信頼性が高く、適切に維持され、パフォーマンスの高いものを構築できるように、リーダーシップからのスポンサーシップを持つことも重要です。

完全にカプセル化された CSS – 何も共有しないアプローチ

各マイクロフロントエンドは、規則とツールを使用して CSS のカスケード機能を克服します。たとえば、各要素のスタイルが常に要素の ID ではなくクラス名に関連付けられ、クラス名は常に一意であることを確認できます。これにより、すべてが個々のマイクロフロントエンドに限定され、不要な

競合のリスクが最小限に抑えられます。アプリケーションシェルは通常、マイクロフロントエンドのスタイルを DOM にロードした後でロードしますが、一部のツールは JavaScript を使用してスタイルをバンドルします。

何も共有しないことの主な利点は、マイクロフロントエンド間で競合が発生するリスクを減らすことです。もう 1 つの利点は、開発者の経験です。各マイクロフロントエンドは、他のマイクロフロントエンドと何も共有しません。単独でのリリースとテストは、より簡単で迅速です。

共有なしアプローチの主な欠点は、一貫性の欠如の可能性です。整合性を評価するシステムがない。共有内容を複製することが目標であっても、リリースとコラボレーションの速度のバランスをとると困難になります。一般的な緩和策は、整合性を測定するツールを作成することです。たとえば、ヘッドレスブラウザを使用してページにレンダリングされた複数のマイクロフロントエンドの自動スクリーンショットを取得するシステムを作成できます。その後、リリース前にスクリーンショットを手動で確認できます。ただし、これには規律とガバナンスが必要です。詳細については、「Balancing autonomy with alignment」セクションを参照してください。

ユースケースによっては、もう 1 つの潜在的な欠点はパフォーマンスです。すべてのマイクロフロントエンドで大量のスタイルが使用されている場合、お客様は多数の重複したコードをダウンロードする必要があります。これはユーザーエクスペリエンスに悪影響を及ぼします。

この共有なしのアプローチは、少数のチームのみを含むマイクロフロントエンドアーキテクチャ、または低い一貫性を許容できるマイクロフロントエンドでのみ考慮する必要があります。また、組織が設計システムに取り組んでいる間は、自然な最初のステップになることもあります。

共有グローバル CSS – 共有オールアプローチ

このアプローチでは、スタイルに関連するすべてのコードは中央リポジトリに保存され、寄稿者は CSS ファイルを操作するか、Sass などのプリプロセッサを使用してすべてのマイクロフロントエンドの CSS を記述します。変更が行われると、ビルドシステムは CDN でホストでき、アプリケーションシェルによって各マイクロフロントエンドに含めることができる単一の CSS バンドルを作成します。マイクロフロントエンド開発者は、ローカルにホストされたアプリケーションシェルを介してコードを実行することで、アプリケーションを設計および構築できます。

マイクロフロントエンド間の競合リスクを軽減する明らかな利点とは別に、このアプローチの利点は一貫性とパフォーマンスです。ただし、マークアップやロジックからスタイルを切り離すと、デベロッパーはスタイルの使用方法、進化方法、廃止方法を理解するのが難しくなります。例えば、既存のクラスとそのプロパティを編集した結果について学ぶよりも、新しいクラス名を導入する方が早い場合があります。新しいクラス名を作成する欠点は、バンドルサイズの増加です。これはパフォーマンスに影響し、ユーザーエクスペリエンスに不整合が生じる可能性があります。

共有グローバル CSS はmonolith-to-micro-frontendsの出発点になる可能性がありますが、複数のチームが連携するマイクロフロントエンドアーキテクチャにとって有益であることはほとんどありません。設計システムの開発中は、できるだけ早く設計システムに投資し、共有なしのアプローチを実装することをお勧めします。

組織と作業方法

すべてのアーキテクチャ戦略と同様に、マイクロフロントエンドは、組織が実装することを選択するテクノロジーをはるかに超えた意味を持ちます。マイクロフロントエンドアプリケーションを構築する決定は、ビジネス、製品、組織、運用、さらには文化 (チームの権限付与や意思決定の分散など) と一致する必要があります。見返りとして、このタイプのマイクロフロントエンドアーキテクチャは、真の俊敏性、製品主導の開発をサポートします。これは、独立しているチーム間の通信オーバーヘッドを大幅に削減するためです。

アジャイル開発

アジャイルソフトウェア開発の概念は、ここ数年で非常に普遍的になり、事実上すべての組織がアジャイルに取り組むと主張しています。アジャイルの決定的な定義はこの戦略の範囲外ですが、マイクロフロントエンド開発に関連する主要な要素を確認する価値があります。

アジャイルパラダイムの基盤は Agile Manifesto (2001) です。このマニフェストは、4 つの主要な原則 (プロセスやツールに対する個人とインタラクションなど) と 12 の原則を前提としています。Scrum や Scaled Agile Framework (SAFe) などのプロセスフレームワークは、Agile Manifesto を中心に出現し、日常的なプラクティスへの道を見つけました。ただし、その背後にある哲学は、ほとんど誤解されているか無視されています。

マイクロフロントエンドアーキテクチャでは、以下のアジャイル原則を採用することが重要です。

- 「作業ソフトウェアを数週間から数か月の間で頻繁に配信し、より短いタイムスケールを優先します。」

  この原則は、増分で作業し、ソフトウェアを可能な限り定期的に本番環境に配信することの重要性を強調しています。技術的な観点からは、継続的インテグレーションと継続的デリバリー (CI/CD) を指します。CI/CD では、構築、テスト、デプロイのためのツールとプロセスは、各ソフトウェアプロジェクトの不可欠な部分です。プリンシパルは、ランタイムインフラストラクチャと運用責任がチームによって所有されている必要があることも意味します。この所有権は、独立したサブシステムがインフラストラクチャと運用の要件を大幅に異なる可能性がある分散システムでは特に重要です。
- 「意欲のある個人を中心にプロジェクトを構築します。必要な環境とサポートを提供し、ジョブを完了するために信頼します。」

  「最高のアーキテクチャ、要件、設計は、自己組織化チームから生まれます。」

これらの原則はいずれも、所有権、独立性、end-to-endの責任の利点を強調しています。マイクロフロントエンドアーキテクチャは、チームがマイクロフロントエンドを真に所有している場合 (およびマイクロフロントエンドを所有している場合のみ) に成功します。構想から設計、実装、デリバリー、運用までEnd-to-endの責任により、チームは実際に所有権を行使できます。この独立性は、チームが戦略的方向性を自律させるために、技術的にも組織的にも必要です。ウォーターフォール開発モデルを使用する一元化された組織でマイクロフロントエンドプラットフォームを使用することはお勧めしません。

チームの構成とサイズ

ソフトウェアチームが所有権を行使するには、組織が課す境界内で、チームが提供する方法と内容を含め、自らを管理する必要があります。

有効にするには、チームはソフトウェアを個別に配信でき、配信する最善の方法を決定する権限を持っている必要があります。これらの項目の計画に関与せずに、外部製品マネージャーまたは外部デザイナーから UI 設計から機能要件を受け取るチームは、自律的と見なすことはできません。機能は、既存の契約や機能に違反する可能性があります。このような違反には、さらなる議論と交渉が必要であり、配信が遅延し、チーム間で不要な競合が発生するリスクがあります。

同時に、チームが大きすぎないようにする必要があります。大規模なチームにはより多くのリソースがあり、個々の欠席に対応できますが、コミュニケーションの複雑さは新しいメンバーごとに指数関数的に増加します。普遍的に有効な最大チームサイズを記述することはできません。プロジェクトに必要な人数は、チームの成熟度、技術的複雑さ、イノベーションのペース、インフラストラクチャなどの要因によって異なります。例えば、Amazon は 2 つのピザのルールに従います。2 つのピザを食べるには大きすぎるチームは、小さなチームに分割する必要があります。これは難しい場合があります。分割は自然境界に沿って行われ、各チームに作業に対する自律性と所有権を与える必要があります。

DevOps 文化

DevOps とは、開発ライフサイクルのステップを組織的および技術的な観点から緊密に統合するソフトウェアエンジニアリングプラクティスを指します。一般的な考え方とは対照的に、DevOps は文化と考え方に関するものであり、役割とツールに関するものはほとんどありません。

従来、ソフトウェア組織には、設計、実装、テスト、デプロイ、運用などのスペシャリストのチームがありました。チームがジョブを完了するたびに、プロジェクトを次のチームに引き渡します。ただ

し、専門チームのサイロ化によるソフトウェアの配信は、引き渡し中に摩擦を引き起こします。同時に、スペシャリストが狭い焦点で作業を強いられると、近隣のドメインに関する知識が不足し、製品の体系的なビューがありません。これらの障害により、ソフトウェア製品の一貫性が低下する可能性があります。

たとえば、ソフトウェアアーキテクトが別のチームの誰かによって実装されるソリューションを設計する場合、実装の固有の側面 (依存関係の不一致など) を見落としている可能性があります。その後、開発者はショートカット (猿のパッチなど) を使用するか、アーキテクトと開発チームの間で形式化されたback-and-forthが開始されます。これらのプロセスを管理するオーバーヘッドのため、開発はアジャイルではなくなりました (柔軟、適応的、段階的、非公式) 。

DevOps という用語は主に文化に関連していますが、実際に DevOps を可能にするテクノロジーとプロセスを意味します。DevOps は CI/CD と密接に関連しています。開発者は、ソフトウェアの増分の実装を完了すると、Git などのバージョン管理システムにコミットします。従来、ビルドシステムはソフトウェアを構築して統合し、多かれ少なかれ統合され一元化されたプロセスでテストしてリリースしていました。CI/CD では、ソフトウェアの構築、統合、テスト、リリースが本質的に自動化されています。理想的には、プロセスは、特定のプロジェクトに合わせて特別に調整された設定ファイルを通じてソフトウェアプロジェクト自体の一部です。

できるだけ多くのステップが自動化されます。たとえば、ほぼすべてのタイプのテストを自動化できるため、手動テストのプラクティスを減らす必要があります。そのようにプロジェクトを設定すると、ソフトウェア製品の更新を 1 日に数回、高い信頼性で配信できます。DevOps をサポートするもう 1 つのテクノロジーは、Infrastructure as Code (IaC) です。

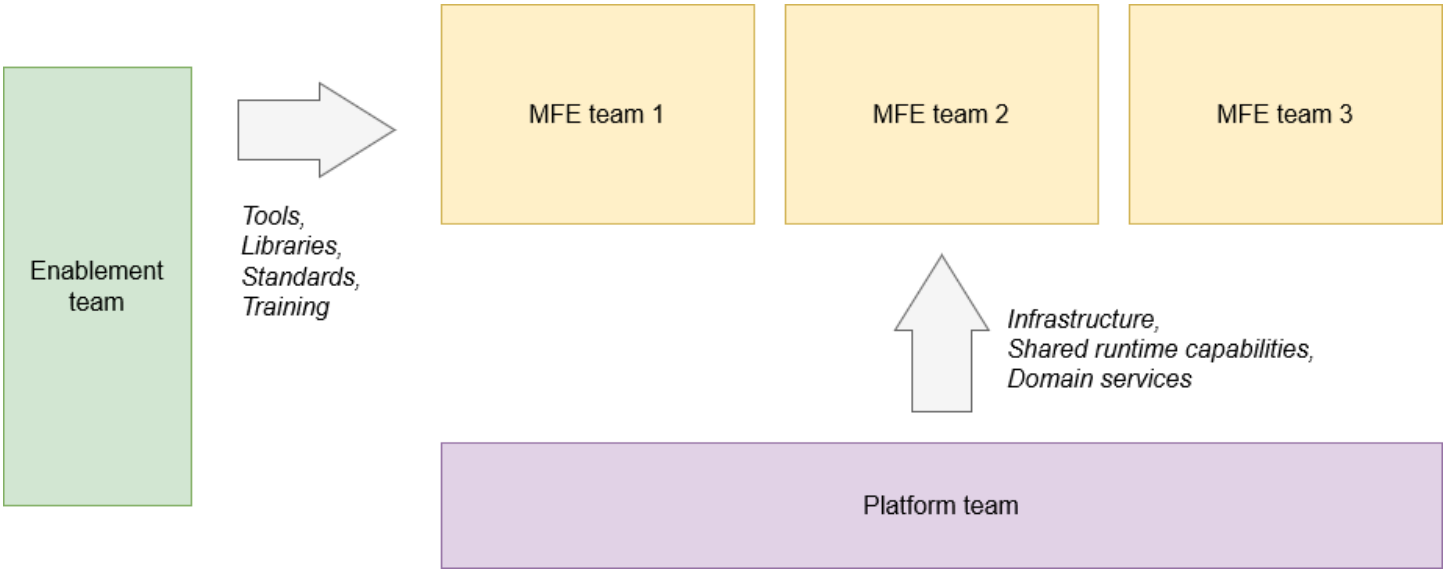
従来、IT インフラストラクチャのセットアップと保守には、ハードウェア (データセンター内のケーブルとサーバーのセットアップ) と運用ソフトウェアのインストールと保守を手動で行う必要があります。これは必要でしたが、多くの欠点がありました。セットアップに時間がかかり、エラーが発生しやすくなります。ハードウェアのプロビジョニングが過剰であったり、プロビジョニングが不足していたりすることがよくあり、過剰な支出やパフォーマンスの低下につながります。IaC を使用すると、クラウドサービスを自動的にデプロイおよび更新できる設定ファイルを使用して、IT システムのインフラストラクチャ要件を 記述できます。

これらはすべてマイクロフロントエンドとどのような関係がありますか? DevOps、CI/CD、IaC は、マイクロフロントエンドアーキテクチャの補完に最適です。マイクロフロントエンドの利点は、高速でスムーズな配信プロセスに依存しています。DevOps 文化は、チームがend-to-endの責任を負うソフトウェアプロジェクトを所有する環境でのみ実現できます。

複数のチームにわたるマイクロフロントエンド開発のオーケストレーション

複数の部門横断的なチームでマイクロフロントエンド開発をスケーリングする場合、2 つの問題が発生します。まず、チームはパラダイムの独自の解釈を開発し、フレームワークとライブラリの選択を行い、独自のツールとヘルパーライブラリを作成します。次に、完全自律型チームが、低レベルのインフラストラクチャ管理などの一般的な機能を担当する必要があります。したがって、マルチチームのマイクロフロントエンド組織に 2 つのチーム、つまり有効化チームとプラットフォームチームを導入することは理にかなっています。これらの概念は、分散システムを持つ最新の IT 組織で広く採用されており、チームトポロジーで十分に文書化されています。

次の図は、3 つのマイクロフロントエンドチームにツール、ライブラリ、標準、テストを提供する有効化チームを示しています。プラットフォームチームは、同じ 3 つのマイクロフロントエンドチームにインフラストラクチャ、共有ランタイム機能、ドメインサービスを提供します。



プラットフォームチームは、マイクロフロントエンドチームを差別化されていない重労働から解放することで、マイクロフロントエンドチームをサポートします。このサポートには、コンテナランタイム、CI/CD パイプライン、コラボレーションツール、モニタリングなどのインフラストラクチャサービスが含まれます。ただし、プラットフォームチームを設定すると、開発がオペレーションからデタッチされる組織につながるべきではありません。逆に、プラットフォームチームはエンジニアリング製品を提供し、マイクロフロントエンドチームはプラットフォーム上のサービスの所有権とランタイム責任を負います。

有効化チームは、ガバナンスに重点を置き、マイクロフロントエンドチーム全体の一貫性を確保することでサポートを提供します。(プラットフォームチームはこれに関与しないでください) 。有効

化チームは UI ライブラリなどの共有リソースを維持し、フレームワークの選択、パフォーマンス予算、相互運用性規則などの標準を作成します。同時に、ガバナンスで定義されている標準とツールの適用に関するトレーニングを新しいチームまたはチームメンバーに提供します。

デプロイ

マイクロフロントエンドチームの自律性の北の星は、他のマイクロフロントエンドチームから独立した本番稼働へのパスを持つ自動パイプラインを持つことです。共有なしの原則に従うチームは、独立したパイプラインを実装できます。ライブラリを共有したり、プラットフォームチームに依存するチームは、デプロイパイプラインの依存関係を管理する方法を決定する必要があります。

通常、各パイプラインは以下を実行します。

- フロントエンドアセットを構築します
- アセットをホスティングにデプロイして消費します
- 新しいバージョンを顧客に配信できるように、レジストリとキャッシュが更新されていることを確認します

実際のパイプラインステップは、テクノロジースタックとページ構成アプローチによって異なります。

クライアント側の構成では、アプリケーションバンドルをホスティングバケットにアップロードし、CDN でのキャッシュを通じて消費に解放することを意味します。サービスワーカーでブラウザキャッシュを使用するアプリケーションでは、サービスワーカーキャッシュを更新する方法も実装する必要があります。

サーバー側の構成の場合、これは通常、サーバーコンポーネントの新しいバージョンをデプロイし、新しいバージョンを検出できるようにマイクロフロントエンドレジストリを更新することを意味します。Blue/Green または Canary デプロイパターンを使用して、新しいバージョンを徐々にロールアウトできます。

ガバナンス

複数のペルソナは通常、マイクロフロントエンドで動作し、それぞれが共通のビジネス目標に対して異なる制約の下で動作します。人々間のコミュニケーションとコラボレーションが成功の鍵ですが、過度に複雑なプロセスを過度に伝達して実装すると、開発サイクルが遅くなります。これにより、士気が低下し、品質バーが低下します。

複数のチームを使用してマイクロフロントエンドを実装する最も成功した企業は、自律性と整合性のバランスをとるメカニズムを作成します。これにより、意思決定者はローカルでアクションを実行し、必要な場合にのみ階層的にエスカレーションできます。メカニズムには以下が含まれます。

- API 契約
- イベントを使用した相互対話
- 自律性とアラインメントのバランスをとる
- 機能フラグ
- サービス検出

API 契約

各マイクロフロントエンドは、意見、ロジック、複雑さをカプセル化できるシステムです。クロスカットの懸念には通常、次のようなものがあります。

- システムを設計する - ライブラリとして配布UIs を開発するためのツール
- コンポジション – マイクロフロントエンドがアプリケーションシェルとやり取りしてレンダリングし、そのコンテキストを継承する方法
- ロジック処理 – 永続状態を処理する APIsとのやり取り
- 他のマイクロフロントエンドとのインタラクティブ性 - イベントの発行と消費、マイクロフロントエンドから別のマイクロフロントエンドへの移動などのシナリオ

消費とトラブルシューティングを高速化するには、マイクロフロントエンドの依存関係など、これらのインターフェイスを宣言して文書化する方法の標準化に投資するのが一般的です。人間によってキュレートされた Wiki は良いスタートです。よりスケーラブルなアプローチは、この情報を構造化メタデータとしてコードに保存することです。その後、自動化を使用して履歴の変更を追跡し、全文検索を提供することで、消費のために一元化できます。

マイクロフロントエンドに多数のチームが関与する場合は、チーム間で調整する戦略が必要です。API 契約を統一された方法で共有することは、通信のオーバーヘッドを減らし、デベロッパーエクスペリエンスを向上させるため、必須となります。

OpenAPI は、APIs インターフェイスとコントラクトを統一された方法で定義することをサポートする HTTP API の仕様言語です。Amazon APIs Gateway で OpenAPI を使用して REST API を実装できます。 OpenAPI Amazon API Gateway コンテナや仮想マシンでホストできるさまざまなオープンソースフレームワークを使用することもできます。大きな利点は、OpenAPI が一貫した形式でドキュメントを自動的に生成できるため、複数のチームが最小限の初期投資で知識を共有できることです。

複数のチームがマイクロフロントエンドに取り組む場合、多くの場合、グループを形成します。これらのグループでは、人々は全体像について考え、貢献しながら、互いに出会い、学び合うことができます。これらのイニシアチブは通常、所有権の境界を定義して文書化し、クロスカットの懸念について話し合い、一般的な問題を解決するための取り組みの重複を早期に特定します。

イベントを使用したクロスインタラクション

シナリオによっては、状態の変化やユーザーアクションに対応するために、複数のマイクロフロントエンドが相互にやり取りする必要がある場合があります。たとえば、ページ上の複数のマイクロフロントエンドに折りたたみ可能なメニューを含めることができます。ユーザーがボタンを選択すると、メニューが表示されます。メニューは、別のマイクロフロントエンド内でレンダリングされる別のメニューなど、ユーザーが他の場所をクリックすると非表示になります。

技術的には、Redux などの共有状態ライブラリは複数のマイクロフロントエンドで使用でき、シェルで調整できます。ただし、アプリケーション間の結合が大きくなり、コードのテストが難しくなり、レンダリング中のパフォーマンスが低下する可能性があります。

一般的で効果的なアプローチの 1 つは、ライブラリとして配布され、アプリケーションシェルによってオーケストレーションされ、複数のマイクロフロントエンドで使用されるイベントバスを開発することです。このようにして、各マイクロフロントエンドは特定のイベントを非同期的に発行してリッスンし、その動作を独自の内部状態のみに基づいて行います。その後、複数のチームが、イベントを記述し、ユーザーエクスペリエンスデザイナーが合意した動作を文書化する共有 Wiki ページを維持できます。

イベントバスの例の実装では、ドロップダウンコンポーネントは共有バスを使用して、ペイロードが `drop-down-open-menu`の というイベントを発行します`{"id": "homepage-aboutus-button"}`。コンポーネントはイベントにリスナーを追加して`drop-down-open-menu`、イベント

が新しい ID に対して発生した場合、ドロップダウンコンポーネントがレンダリングされ、折りたたみ可能なセクションが非表示になります。このようにして、マイクロフロントエンドはパフォーマンスの向上とカプセル化の向上により、変更に非同期的に対応できるため、複数のチームが動作を設計およびテストしやすくなります。

シンプルさと保守性を向上させるために、最新のブラウザによってネイティブに実装された標準 APIs を使用することをお勧めします。MDN イベントリファレンスは、クライアント側でレンダリングされたアプリケーションでのイベントの使用に関する情報を提供します。

自律性とアラインメントのバランスをとる

マイクロフロントエンドアーキテクチャは、チームの自律性に強く偏っています。ただし、問題を解決するための柔軟性と多様なアプローチをサポートできる領域と、調整を達成するために標準化が必要な領域を区別することが重要です。シニアリーダーやアーキテクトは、これらの分野を早期に特定し、マイクロフロントエンドのセキュリティ、パフォーマンス、運用上の優秀性、信頼性のバランスを取るために投資を優先する必要があります。このバランスを見つけるには、マイクロフロントエンドの作成、テスト、リリース、ログ記録、モニタリング、アラートが含まれます。

マイクロフロントエンドの作成

理想的には、エンドユーザーのパフォーマンスの観点からメリットを最大化するために、すべてのチームが緊密に連携しています。実際には、これは難しい場合があり、より多くの労力が必要になる場合があります。複数のチームがオープンで透明性のある議論を通じて貢献できる、いくつかの書面によるガイドラインから始めることをお勧めします。その後、チームは Cookiecutter ソフトウェアパターンを徐々に採用できます。これにより、プロジェクトをスキャフォールドするための統一された方法を提供するツールの作成がサポートされます。

このアプローチを使用すると、意見や制約を活かすことができます。欠点は、これらのツールの作成とメンテナンスに多大な投資が必要であり、デベロッパーの生産性に影響を与えずにブロッカーに迅速に対処できるようにするためです。

マイクロフロントエンドのEnd-to-endテスト

ユニットテストは所有者に任せることができます。独自のシェルで実行されているマイクロフロントエンドをクロステストするための戦略を早期に実装することをお勧めします。この戦略には、本番リリースの前後にアプリケーションをテストする機能が含まれています。重要な機能を手動でテストするには、技術担当者と非技術担当者向けのプロセスとドキュメントを作成することをお勧めします。

変更によって機能的または非機能的カスタマーエクスペリエンスが損なわれないようにすることが重要です。理想的な戦略は、主要な機能とセキュリティやパフォーマンスなどのアーキテクチャ特性の両方について、自動テストに徐々に投資することです。

マイクロフロントエンドのリリース

各チームには、コードをデプロイし、意見を出し合い、独自のインフラストラクチャを構築するための独自の方法があります。このようなシステムを維持するための複雑さのコストは通常、抑止力です。代わりに、 に早期投資して、共有ツールで適用できる共有戦略を実装することをお勧めします。

選択した CI/CD プラットフォームを使用してテンプレートを開発します。その後、チームは事前承認されたテンプレートと共有インフラストラクチャを使用して、本番稼働用に変更をリリースできます。これらのシステムでは、テストと統合の初期期間後に大幅な更新が必要になることはほとんどないため、この開発作業に早期に投資を開始できます。

ログ記録とモニタリング

各チームは、運用または分析の目的で追跡するさまざまなビジネスメトリクスとシステムメトリクスを持つことができます。Cookiecutter ソフトウェアパターンは、ここでも適用できます。イベントの配信は抽象化でき、複数のマイクロフロントエンドが使用できるライブラリとして使用できます。柔軟性のバランスを取り、自律性を提供するには、カスタムメトリクスをログに記録し、カスタムダッシュボードまたはレポートを作成するためのツールを開発します。このレポートは、製品所有者との緊密なコラボレーションを促進し、エンドユーザーのフィードバックループを減らします。

配信を標準化することで、複数のチームが協力してメトリクスを追跡できます。たとえば、e コマースウェブサイトでは、「製品の詳細」マイクロフロントエンドから「カート」マイクロフロントエンド、「購入」マイクロフロントエンドまでのユーザージャーニーを追跡して、エンゲージメント、解約、問題を測定することができます。各マイクロフロントエンドが 1 つのライブラリを使用してイベントをログに記録する場合、このデータ全体を消費し、包括的に調査して、洞察力のある傾向を特定できます。

[アラート]

ログ記録とモニタリングと同様に、アラートはある程度の柔軟性を備えた標準化の利点があります。チームによって、機能アラートと非機能アラートの反応が異なる場合があります。ただし、すべてのチームが、共有プラットフォームで収集および分析されたメトリクスに基づいてアラートを開始する統合された方法を持っている場合、ビジネスはチーム間の問題を特定できます。この機能は、インシデント管理イベント中に便利です。たとえば、アラートは以下によって開始できます。

- 特定のブラウザバージョンでの JavaScript クライアント側の例外の数の増加
- 特定のしきい値でレンダリングが大幅に低下する時間
- 特定の API を使用する場合の 5xx ステータスコードの数の増加

システムの成熟度に応じて、次の表に示すように、インフラストラクチャのさまざまな部分で労力のバランスをとることができます。

導入	研究と開発	昇順	成熟度
マイクロフロントエンドを作成します。	学習内容を実験、文書化、共有します。	新しいマイクロフロントエンドをスキャフォールドするためのツールに投資します。導入を助長します。	足場のツールを統合します。導入をプッシュします。
マイクロフロントエンドをエンドツーエンドでテストします。	関連するすべてのマイクロフロントエンドを手動でテストするメカニズムを実装します。	セキュリティとパフォーマンスの自動テストのためのツールに投資します。機能フラグとサービス検出を調査します。	サービス検出、本番環境でのテスト、end-to-endテストのためのツールを統合します。
マイクロフロントエンドを解放します。	共有 CI/CD インフラストラクチャと自動複数環境リリースに投資します。導入を助長します。	CI/CD インフラストラクチャのツールを統合する 手動ロールバックメカニズムを実装します。導入をプッシュします。	システムおよびビジネスメトリクスとアラートに基づいて自動ロールバックを開始するメカニズムを作成します。
マイクロフロントエ	システムイベントとビジネスイベントの一貫し	モニタリングとアラートのためのツールを統合します。チーム間のダッシュ	ログ記録スキーマを標準化します。コストに合わせて最適化します。複

導入	研究と開発	昇順	成熟度
ンドのパフォーマンスを確認します。	たログ記録のために、共有モニタリングインフラストラクチャとライブラリに投資します。	ボードを実装して、全般的なヘルスをモニタリングし、インシデント管理を改善します。	雑なビジネスメトリクスに基づいてアラートを実装します。

機能フラグ

機能フラグをマイクロフロントエンドに実装して、複数の環境で機能のテストとリリースを容易に調整できます。機能フラグ手法は、ブールベースのストアで決定を一元化し、それに基づいて動作を駆動することで構成されます。多くの場合、特定の時点まで非表示にできる変更をサイレントに伝達し、ブロックされる新機能の新しいリリースをロック解除して、チームの速度を低下させるために使用されます。

特定の日付に起動されるマイクロフロントエンド機能に取り組むチームの例を考えてみましょう。この機能は準備完了ですが、個別にリリースされる別のマイクロフロントエンドの変更とともにリリースする必要があります。両方のマイクロフロントエンドのリリースをブロックすると、アンチパターンと見なされ、デプロイ時にリスクが高まります。

代わりに、チームはレンダリング時間中に (おそらく共有特徴量フラグ API への HTTP 呼び出しを通じて) 両方が消費するデータベースにブール機能フラグを作成できます。チームは、ブール値が に設定されているテスト環境で変更をリリースTrueして、本稼働環境に起動する前にプロジェクト間の機能要件と非機能要件を検証することもできます。

機能フラグの使用のもう 1 つの例は、 QueryStringパラメータを使用して特定の値を設定するか、特定のテスト文字列を Cookie に保存することで、フラグの値を上書きするメカニズムを実装することです。製品所有者は、起動日まで他の機能やバグ修正のリリースをブロックすることなく、機能を反復処理できます。指定された日付に、データベースのフラグ値を変更すると、チーム間で調整されたリリースを必要とせずに、変更がすぐに本番環境に表示されます。機能がリリースされると、開発チームはコードをクリーンアップして古い動作を削除します。

その他のユースケースには、コンテキストベースの機能フラグシステムのリリースが含まれます。例えば、1 つのウェブサイトで複数の言語で顧客にサービスを提供している場合、特定の国の訪問者のみが機能を利用できる可能性があります。機能フラグシステムは、国のコンテキストを送信するコン

シューマーに依存することができ (HTTP Accept-Language ヘッダーを使用するなど)、そのコンテキストに応じて動作が異なる場合があります。

機能フラグはデベロッパーと製品所有者間のコラボレーションを容易にする強力なツールですが、コードベースの大幅な低下を避けるために、人々の努力に依存しています。複数の機能でフラグをアクティブにしておくと、問題のトラブルシューティング時の複雑さが増し、JavaScript バンドルのサイズが増加し、最終的に技術的負債が蓄積される可能性があります。一般的な緩和アクティビティは次のとおりです。

- フラグの背後にある各機能をユニットテストしてバグの可能性を減らすことで、テストを実行する自動 CI/CD パイプラインに長いフィードバックループを導入できます。
- コード変更中のバンドルサイズの増加を測定するツールの作成。コードレビュー中に軽減できます。

AWS は、Amazon CloudFront 関数または Lambda@Edge を使用してエッジでの A/B テストを最適化するためのさまざまなソリューションを提供します。これらのアプローチは、仮定をアサートするために使用しているソリューションまたは既存の SaaS 製品の統合の複雑さを軽減するのに役立ちます。詳細については、「A/B テスト」を参照してください。

サービス検出

フロントエンド検出パターンは、マイクロフロントエンドを開発、テスト、配信する際の開発エクスペリエンスを向上させます。このパターンでは、マイクロフロントエンドのエントリポイントを記述する共有可能な設定を使用します。共有可能な設定には、Canary リリースを使用して各環境で安全なデプロイに使用される追加のメタデータも含まれています。

最新のフロントエンド開発では、開発中のモジュール性をサポートするために、さまざまなツールとライブラリを使用する必要があります。従来、このプロセスは、初期ロード (アプリがブラウザで開いたとき) や使用状況 (顧客がボタンの選択や情報の挿入などのアクションを実行するとき) など、ランタイム中にネットワーク呼び出しを最小限に抑えることを目的として、CDN でホストできる個々のファイルにコードをバンドルすることで構成されていました。

バンドルの分割

マイクロフロントエンドアーキテクチャは、大量の機能を個別にバンドルすることで生成された非常に大きなバンドルによって発生するパフォーマンスの問題を解決します。たとえば、非常に大規模な e コマースウェブサイトを 6 MB の JavaScript ファイルにバンドルできます。圧縮にもかかわらず、

そのファイルのサイズは、アプリをロードし、エッジ最適化 CDN からファイルをダウンロードするときに、ユーザーのエクスペリエンスに悪影響を及ぼす可能性があります。

アプリケーションをホームページ、製品の詳細、カートのマイクロフロントエンドに分割する場合、バンドルメカニズムを使用して 3 つの個別の 2 MB バンドルを作成できます。この変更により、ユーザーがホームページを使用する場合、最初のロードのパフォーマンスが 300% 向上する可能性があります。製品またはカートのマイクロフロントエンドバンドルは、ユーザーがアイテムの製品ページにアクセスして購入を決定した場合にのみ非同期的にロードされます。

このアプローチに基づいて多くのフレームワークとライブラリが利用可能であり、顧客とデベロッパーの両方に利点があります。コード内の依存関係の分離につながる可能性のあるビジネスの境界を特定するには、さまざまなビジネス機能を複数のチームにマッピングできます。分散所有権は、独立性と俊敏性をもたらします。

ビルドパッケージを分割する場合、設定を使用してマイクロフロントエンドをマッピングし、最初のロードとロード後のナビゲーションのオーケストレーションを駆動できます。その後、ビルド時ではなくランタイム時に設定を使用できます。たとえば、クライアント側のフロントエンドコードまたはサーバー側のバックエンドコードは、API への初期ネットワーク呼び出しを実行して、マイクロフロントエンドのリストを動的に取得できます。また、構成と統合に必要なメタデータも取得します。フェイルオーバー戦略とキャッシュを設定して、信頼性とパフォーマンスを実現できます。マイクロフロントエンドをマッピングすると、シェルアプリケーションによってオーケストレーションされた以前にデプロイされたマイクロフロントエンドによって、マイクロフロントエンドの個別のデプロイを検出できるようになります。

Canary リリース

Canary リリースは、マイクロサービスをデプロイするための確立された一般的なパターンです。Canary は、リリースのターゲットユーザーを複数のグループにバケット化し、即時の置き換え (ブルー/グリーンデプロイとも呼ばれます) ではなく、徐々に変更をリリースします。Canary リリース戦略の例は、ターゲットユーザーの 10% に新しい変更をロールアウトし、1 分ごとに 10% を追加し、合計時間が 10 分で 100% に達することです。

Canary リリースの目的は、変更に関するフィードバックを早期に取得し、システムをモニタリングして問題の影響を減らすことです。自動化が実施されると、デプロイを停止したり、ロールバックを開始したりできる内部システムによってビジネスメトリクスまたはシステムメトリクスをモニタリングできます。

たとえば、変更により、リリースの最初の数分間に収益が失われたり、パフォーマンスが低下したりするバグが発生する可能性があります。自動モニタリングはアラームを開始できます。サービス検出

パターンを使用すると、このアラームはデプロイを停止してすぐにロールバックでき、100% ではなく 20% のユーザーのみに影響します。ビジネスは、問題の範囲を縮小することでメリットを得られます。

ストレージとして DynamoDB を使用して REST 管理 API を実装するアーキテクチャの例については、GitHub の「AWS でのフロントエンドサービス検出」ソリューションを参照してください。テンプレートを使用して AWS CloudFormation 、アーキテクチャを独自の CI/CD パイプラインに統合します。このソリューションには、ソリューションをフロントエンドアプリケーションと統合するための REST コンシューマー API が含まれています。

プラットフォームチームが必要ですか？

一部の企業には、マイクロフロントエンドに取り組むために他のチームによって採用されるコード、インフラストラクチャ、プロセスを所有および維持する責任を持つチームがあります。一般的な責任には以下が含まれます。

- マイクロフロントエンドを含むリポジトリで使用できる CI/CD パイプラインを作成して維持します。コード変更を構築してテストし、複数の環境でリリースします。
- 問題に対応するために、共有ダッシュボード、アラートメカニズム、システムなどのオブザーバビリティ関連のツールを作成して維持します。
- イベント処理、共有サービスの使用、およびサードパーティーの依存関係のための共有ライブラリを作成して維持します。
- システムのパフォーマンス、セキュリティ、信頼性など、機能以外の品質を継続的にモニタリングするツールを作成および維持します。
- 設計システムを作成して維持します。
- マイクロフロントエンドシステムのアプリケーションシェルを作成、保守、サポートします。

プロジェクトの規模に応じて、次のいずれかのアプローチを使用して、これらの責任を管理できます。

- 共有ツールを扱う唯一の責任を持つ専用のプラットフォームチームを作成します。
- 複数のチームのメンバーで構成されるグループを作成します。グループメンバーは、マイクロフロントエンドでの作業と、共有ツールでの作業の時間を分割します。これはトラチームとも呼ばれます。

タイガーチームアプローチは顧客中心の効果的な方法ですが、プロジェクトが推進力と責任を獲得すれば、タイガーチームはプラットフォームチームに進化することがよくあります。プラットフォームチームとタイガーチームの両方で、マイクロフロントエンドに取り組む最も成功した企業がこれらのチームを形成し、複数のバックグラウンドとスキルを持つ複数の人が貢献できるようにします。チームメンバーには、バックエンドエンジニア、フロントエンドエンジニア、ユーザーエクスペリエンス (UX) デザイナー、テクニカルプロダクトマネージャーが含まれる場合があります。この多様性により、人々はシンプルさを念頭に置いて、健全な議論や設計に継続的に取り組むことができます。

次のステップ

このガイドでは、アーキテクチャと組織のパターン、主要な意思決定のトレードオフ、マイクロフロントエンドに関連するガバナンスの懸念について説明しました。表は、このドキュメントで説明されているプラクティスのトレードオフを以下のディメンションの観点からまとめたものです。

- 自律性 – 各マイクロフロントエンドチームが実装とリリースをエンドユーザーに個別に進化させる能力。
- 一貫性 - 各マイクロフロントエンドが期待どおりに動作するアプリケーションの全体的なエクスペリエンス。一貫性が高いということは、マイクロフロントエンドがアプリケーションの他の部分と一貫性があり、アプリケーション全体のユーザーエクスペリエンスに悪影響を及ぼさないことを意味します。
- 複雑さ – マイクロフロントエンド、アプリケーション全体、ガバナンスコントロールの実装とテストに必要なインフラストラクチャ、コード、労力の量。

練習	自律性	整合性	複雑さ
モノリシックアプリケーションではなくマイクロフロントエンドで構築する	高	中程度	高い

コード共有のプラクティス	自律性	整合性	複雑さ
何も共有しない	高い	低	低
カット間の懸念を共有する	中程度	高	中程度
ビジネスロジックを共有する	低	高	中程度
ビルド時にライブラリを通じて共有する	中程度	高い	低
実行時に共有する	高い	高い	高い

マイクロフロントエンド検出プラクティス	自律性	整合性	複雑さ
アプリケーション構築中に を設定する	低	高い	低
サーバー側の検出	高い	高	中程度
クライアント側 (ランタイム) 検出	高い	高	中程度

コンポジションプラクティスを表示する	自律性	整合性	複雑さ
サーバーサ	高	中程度	高い

コンポジションプラクティスを表示する	自律性	整合性	複雑さ
イドコンポジション			
エッジサイドコンポジション	中程度	中程度	高い
クライアントサイドコンポジション	高	中	中程度

このガイダンスで導入された概念の詳細については、「リソース」セクションを参照してください。

リソース

- コンテキスト内のマイクロフロントエンド
- ドメイン駆動型設計
- EDA ビジュアル
- フロントエンド検出
- でのフロントエンドサービス検出 AWS
- アジャイルマニフェスト
- MDN イベントリファレンス
- OpenAPI

寄稿者

以下の個人がこのガイドに貢献しました。

- プリンシパルソリューションアーキテクト、Matteo MCU AWS
- Alexander Guensche、シニアソリューションアーキテクト、 AWS
- Harun Hasdal、シニアソリューションアーキテクト、 AWS
- Luca Mezzalira、プリンシパル Go to Market Specialist Solutions Architect Serverless UK、 AWS

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
初版発行	—	2024 年 7 月 12 日

AWS 規範ガイダンスの用語集

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

数字

7 Rs

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

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

A

ABAC

「属性ベースのアクセス制御」をご覧ください。

抽象化されたサービス

「マネージドユーザー」をご覧ください。

ACID

「原子性、一貫性、分離性、耐久性 (ACID)」をご覧ください。

アクティブ/アクティブ移行

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

アクティブ/パッシブ移行

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

集計関数

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

AI

「人工知能」をご覧ください。

AIOps

「AI オペレーション」をご覧ください。

匿名化

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

アンチパターン

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

アプリケーション制御

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

アプリケーションポートフォリオ

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

人工知能 (AI)

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

AI オペレーション (AIOps)

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

非対称暗号化

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

原子性、一貫性、分離性、耐久性 (ACID)

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

属性ベースのアクセス制御 (ABAC)

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

信頼できるデータソース

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

アベイラビリティーゾーン (AZ)

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

AWS クラウド導入フレームワーク (AWS CAF)

組織がクラウドへの移行を成功させるための効率的で効果的な計画を立て AWS るための、 のガイドラインとベストプラクティスのフレームワークです。 AWS CAF は、ビジネス、人材、ガバナンス、プラットフォーム、セキュリティ、運用という 6 つの重点分野にガイダンスを整理しています。ビジネス、人材、ガバナンスの観点では、ビジネススキルとプロセスに重点を置き、プラットフォーム、セキュリティ、オペレーションの視点は技術的なスキルとプロセスに焦点を当てています。例えば、人材の観点では、人事 (HR)、人材派遣機能、および人材管理を扱うステークホルダーを対象としています。この観点から、 AWS CAF は、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

「ビジネス継続性計画 (BCP)」をご覧ください。

動作グラフ

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

ビッグエンディアンシステム

最上位バイトを最初に格納するシステム。「エンディアン性」もご覧ください。

二項分類

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

ブルームフィルター

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

ブルー/グリーンデプロイ

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

ボット

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

ボットネット

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

ブランチ

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

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

ブレークグラスアクセス

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

ブラウンフィールド戦略

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

バッファキャッシュ

アクセス頻度が最も高いデータが保存されるメモリ領域。

ビジネス能力

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

ビジネス継続性計画 (BCP)

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

C

CAF

「AWS クラウド導入フレームワーク」を参照してください

カナリアデプロイ

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

CCoE

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

CDC

「変更データキャプチャ」を参照してください。

変更データキャプチャ (CDC)

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

カオスエンジニアリング

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

CI/CD

「継続的インテグレーションと継続的デリバリー」を参照してください。

分類

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

クライアント側の暗号化

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

Cloud Center of Excellence (CCoE)

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

クラウドコンピューティング

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

クラウド運用モデル

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

導入のクラウドステージ

組織が、 AWS クラウドへの移行時に通常実行する 4 つの段階。

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

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

CMDB

「構成管理データベース (CMDB)」を参照してください。

コードリポジトリ

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

コールドキャッシュ

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

コールドデータ

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

コンピュータビジョン (CV)

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

設定ドリフト

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

構成管理データベース (CMDB)

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

コンフォーマンスパック

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

継続的インテグレーションと継続的デリバリー (CI/CD)

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

CV

「コンピュータビジョン」を参照してください。

D

保管中のデータ

ストレージ内にあるデータなど、常に自社のネットワーク内にあるデータ。

データ分類

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

データドリフト

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

転送中のデータ

ネットワーク内 (ネットワークリソース間など) を活発に移動するデータ。

データメッシュ

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

データ最小化

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

データ境界

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

データの前処理

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

データ出所

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

データ件名

データを収集、処理している個人。

データウェアハウス

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

データベース定義言語 (DDL)

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

データベース操作言語 (DML)

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

DDL

「データベース定義言語」を参照してください。

ディープアンサンブル

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

深層学習

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

多層防御

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

委任管理者

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

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

トラブルシューティング

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

開発環境

「環境」を参照してください。

検出管理

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

開発バリューストリームマッピング (DVSM)

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

デジタルツイン

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

ディメンションテーブル

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

ディザスタ

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

ディザスタリカバリ (DR)

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

DML

「データベース操作言語」を参照してください。

ドメイン駆動型設計

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

DR

「ディザスタリカバリ」を参照してください。

ドリフト検出

ベースライン設定からの偏差を追跡します。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するランディングゾーンの変更を検出したりできます。

DVSM

「開発バリューストリームマッピング」を参照してください。

E

EDA

「探索的データ分析」を参照してください。

EDI

「電子データ交換」を参照してください。

エッジコンピューティング

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

電子データ交換 (EDI)

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

暗号化

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

暗号化キー

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

エンディアン

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

エンドポイント

「サービスエンドポイント」を参照してください。

エンドポイントサービス

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

エンタープライズリソースプランニング (ERP)

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

エンベロープ暗号化

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

環境

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

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

エピック

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

ERP

「エンタープライズリソース計画」を参照してください。

探索的データ分析 (EDA)

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

F

ファクトテーブル

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

フェイルファスト

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

障害分離境界

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

機能ブランチ

「ブランチ」を参照してください。

特徴量

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

特徴量重要度

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

機能変換

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

数ショットプロンプト

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

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

FGAC

「きめ細かなアクセス制御」を参照してください。

きめ細かなアクセス制御 (FGAC)

複数の条件を使用してアクセス要求を許可または拒否すること。

フラッシュカット移行

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

FM

「基盤モデル」を参照してください。

基盤モデル (FM)

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

G

生成 AI

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

ジオブロッキング

「地理的制限」を参照してください。

地理的制限 (ジオブロッキング)

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

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

Gitflow ワークフロー

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

ゴールデンイメージ

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

グリーンフィールド戦略

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

ガードレール

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

H

HA

「高可用性」を参照してください。

異種混在データベースの移行

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

高可用性 (HA)

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

ヒストリアンのモダナイゼーション

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

ホールドアウトデータ

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

同種データベースの移行

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

ホットデータ

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

ホットフィックス

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

ハイパーケア期間

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

I

IaC

「Infrastructure as Code」を参照してください。

ID ベースのポリシー

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

アイドル状態のアプリケーション

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

IIoT

「インダストリアル IoT」を参照してください。

イミュータブルインフラストラクチャ

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

インバウンド (受信) VPC

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

増分移行

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

インダストリー 4.0

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

インフラストラクチャ

アプリケーションの環境に含まれるすべてのリソースとアセット。

Infrastructure as Code (IaC)

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

インダストリアル IoT (IIoT)

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

インスペクション VPC

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

IoT

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

解釈可能性

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

IoT

「IoT」を参照してください。

IT 情報ライブラリ (ITIL)

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

IT サービス管理 (ITSM)

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

ITIL

「IT 情報ライブラリ」を参照してください。

ITSM

「IT サービス管理」を参照してください。

L

ラベルベースアクセス制御 (LBAC)

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

ランディングゾーン

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

大規模言語モデル (LLM)

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

大規模な移行

300 台以上のサーバの移行。

LBAC

「ラベルベースアクセス制御」を参照してください。

最小特権

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

リフトアンドシフト

「7 Rs」を参照してください。

リトルエンディアンシステム

最下位バイトを最初に格納するシステム。「エンディアン性」もご覧ください。

LLM

「大規模言語モデル」を参照してください。

下位環境

「環境」を参照してください。

M

機械学習 (ML)

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

メインブランチ

「ブランチ」を参照してください。

マルウェア

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

マネージドサービス

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

製造実行システム (MES)

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

MAP

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

メカニズム

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

メンバーアカウント

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

MES

「製造実行システム」を参照してください。

Message Queuing Telemetry Transport (MQTT)

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

マイクロサービス

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

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

マイクロサービスアーキテクチャ

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

Migration Acceleration Program (MAP)

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

大規模な移行

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

移行ファクトリー

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

移行メタデータ

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

移行パターン

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

Migration Portfolio Assessment (MPA)

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

移行準備状況評価 (MRA)

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

移行戦略

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

ML

「機械学習」を参照してください。

モダナイゼーション

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

モダナイゼーション準備状況評価

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

モノリシックアプリケーション (モノリス)

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

MPA

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

MQTT

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

多クラス分類

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

ミュータブルなインフラストラクチャ

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

O

OAC

「オリジンアクセス制御」を参照してください。

OAI

「オリジンアクセスアイデンティティ」を参照してください。

OCM

「組織変更管理」を参照してください。

オフライン移行

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

OI

「オペレーション統合」を参照してください。

Ola

「オペレーショナルレベルアグリーメント」を参照してください。

オンライン移行

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

OPC-UA

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

Open Process Communications - Unified Architecture (OPC-UA)

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

オペレーショナルレベルアグリーメント (OLA)

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

運用準備状況レビュー (ORR)

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

運用テクノロジー (OT)

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

オペレーション統合 (OI)

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

組織の証跡

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

組織変更管理 (OCM)

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

オリジンアクセス制御 (OAC)

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

オリジンアクセスアイデンティティ (OAI)

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

ORR

「運用準備状況レビュー」を参照してください。

OT

「運用テクノロジー」を参照してください。

アウトバウンド (送信) VPC

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

P

アクセス許可の境界

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

個人を特定できる情報 (PII)

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

PII

「個人を特定できる情報」を参照してください。

プレイブック

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

PLC

「プログラマブルロジックコントローラー」を参照してください。

PLM

「製品ライフサイクル管理」を参照してください。

ポリシー

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

多言語の永続性

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

ポートフォリオ評価

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

述語

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

述語プッシュダウン

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

予防的コントロール

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

プリンシパル

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

プライバシーバイデザイン

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

プライベートホストゾーン

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

プロアクティブコントロール

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

製品ライフサイクル管理 (PLM)

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

本番環境

「環境」を参照してください。

プログラマブルロジックコントローラー (PLC)

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

プロンプトチェイニング

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

仮名化

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

発行/サブスクライブ (pub/sub)

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

Q

クエリプラン

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

クエリプランのリグレッション

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

R

RACI マトリックス

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

RAG

「検索拡張生成」を参照してください。

ランサムウェア

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

RASCI マトリックス

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

RCAC

「行と列のアクセス制御」を参照してください。

リードレプリカ

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

リアーキテクト

「7 Rs」を参照してください。

目標復旧時点 (RPO)

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

目標復旧時間 (RTO)

サービスの中断から復旧までの最大許容遅延時間。

リファクタリング

「7 Rs」を参照してください。

リージョン

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

リグレッション

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

リホスト

「7 Rs」を参照してください。

リリース

デプロイプロセスで、変更を本番環境に昇格させること。

再配置

「7 Rs」を参照してください。

リプラットフォーム

「7 Rs」を参照してください。

再購入

「7 Rs」を参照してください。

回復性

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

リソースベースのポリシー

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

実行責任者、説明責任者、協業先、報告先 (RACI) に基づくマトリックス

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

レスポンシブコントロール

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

保持

「7 Rs」を参照してください。

廃止

「7 Rs」を参照してください。

検索拡張生成 (RAG)

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

ローテーション

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

行と列のアクセス制御 (RCAC)

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

RPO

「目標復旧時点」を参照してください。

RTO

「目標復旧時間」を参照してください。

ランブック

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

S

SAML 2.0

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

SCADA

「監視制御とデータ取得」を参照してください。

SCP

「サービスコントロールポリシー」を参照してください。

シークレット

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

セキュリティバイデザイン

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

セキュリティコントロール

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

セキュリティ強化

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

Security Information and Event Management (SIEM) システム

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

セキュリティレスポンスの自動化

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

サーバー側の暗号化

送信先で、それ AWS のサービス を受け取る によるデータの暗号化。

サービスコントロールポリシー (SCP)

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

サービスエンドポイント

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

サービスレベルアグリーメント (SLA)

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

サービスレベルインジケータ (SLI)

エラー率、可用性、スループットといった、サービスパフォーマンス面の指標。

サービスレベル目標 (SLO)

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

責任共有モデル

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

SIEM

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

単一障害点 (SPOF)

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

SLA

「サービスレベルアグリーメント」を参照してください。

SLI

「サービスレベルインジケータ」を参照してください。

SLO

「サービスレベルの目標」を参照してください。

スプリットアンドシードモデル

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

SPOF

「単一障害点」を参照してください。

スタースキーマ

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

strangler fig パターン

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

サブネット

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

監視制御とデータ取得 (SCADA)

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

対称暗号化

データの暗号化と復号に同じキーを使用する暗号化のアルゴリズム。

合成テスト

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

システムプロンプト

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

T

タグ

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

ターゲット変数

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

タスクリスト

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

テスト環境

「環境」を参照してください。

トレーニング

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

トランジットゲートウェイ

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

トランクベースのワークフロー

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

信頼されたアクセス

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

チューニング

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

ツーピザチーム

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

U

不確実性

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

未分化なタスク

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

上位環境

「環境」を参照してください。

V

バキューミング

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

バージョンコントロール

リポジトリ内のソースコードへの変更など、変更を追跡するプロセスとツール。

VPC ピアリング

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

脆弱性

システムのセキュリティを脅かすソフトウェアまたはハードウェアの欠陥。

W

ウォームキャッシュ

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

ウォームデータ

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

ウィンドウ関数

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

ワークロード

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

ワークストリーム

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

WORM

「Write-Once-Read-Many」を参照してください。

WQF

「AWS ワークロード資格フレームワーク」を参照してください

Write-Once-Read-Many (WORM)

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

Z

ゼロデイエクスプロイト

ゼロデイ脆弱性を悪用した攻撃 (一般的にマルウェアによる)。

ゼロデイ脆弱性

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

ゼロショットプロンプト

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

ゾンビアプリケーション

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

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