AWS
モダンアプリケーション開発

AWS におけるクラウドネイティブ
モダンアプリケーション開発と設計パターン

2019 年 10 月

注意

お客様は、この文書に記載されている情報を独自に評価する責任を負うものとします。本書は、(a) 情報提供のみを目的としており、(b) AWS の現行製品とプラクティスを表しますが、予告なしに変更されることがあり、(c) AWS およびその関連会社、サプライヤーまたはライセンサーからの契約義務や確約を意味するものではありません。AWS の製品やサービスは、明示または暗示を問わず、いかなる保証、表明、条件を伴うことなく「現状のまま」提供されます。お客様に対する AWS の責任は、AWS 契約により規定されます。本書は、AWS とお客様の間で行われるいかなる契約の一部でもなく、そのような契約の内容を変更するものでもありません。

© 2019 Amazon Web Services, Inc. or its affiliates. All rights reserved.

目次

はじめに....6
イノベーションのフライホイールを加速させる....6
モダンアプリケーション開発 7
モダンアプリケーションの機能 8
モダンアプリケーション開発のベストプラクティス 9
モダンアプリケーションの設計パターン 17
AWS のサービスを使用した マイクロサービスアーキテクチャの実装 17
AWS での継続的インテグレーションと 継続的デリバリー 37
AWS での CI/CD サービス....37
さまざまなアプリケーションタイプに合わせた CI/CD パターン 41
まとめ 46
寄稿者 46
その他の資料 47
AWS のサービス....47
ホワイトペーパー....48
動画....49
改訂履歴....49

要約

コンテナとサーバーレステクノロジーを使用したモダンアプリケーション開発は、イノベーションを加速させるための優れた手法です。このホワイトペーパーでは、AWS クラウドにおけるモダンアプリケーション構築に活用できる、重要なベストプラクティスと設計パターンに関する情報をご提供します。

はじめに

現代の企業は、ますますグローバル化を続けています。その製品のデジタル化も進んでいます。そうしたデジタル製品 (クラウドインフラストラクチャ、モバイルアプリケーション、ビッグデータパイプライン、ソーシャルメディアなど) は、アプリケーションの開発手法に影響を与えています。このため、企業には前例がないペースでの変化が求められています。そのようなスピードを達成するため、ビジネスリーダーは自社のカルチャー、プロセス、テクノロジーをデジタル時代の新たな現実に適応させていく必要があります。

現代の企業は自社の人材を最大限に活用し、新たな機会を模索し、新しいアイデアを育成して成長を促進させる必要があり、そのためには迅速なイノベーションが非常に重要となります。迅速なイノベーションの中核を担うのが、デジタルテクノロジーです。

イノベーションのフライホイールを加速させる

ほとんどすべての業界が、ビジネス面で前例がないペースでの変化を経験しています。このため、ビジネスのペースを加速させるために、迅速なイノベーションが不可欠です。小規模な無名の競合相手がイノベーションに注力した結果、そうしたライバルに数か月で大きな差をつけられてしまう、ということがあり得ます。このため、ただのイノベーションではなく、迅速なイノベーションが必要です。

Amazon は、実験によってイノベーションが加速することを学んできました。イノベーションを加速させるには、実験し、ユーザーのフィードバックに耳を傾けてから、もう一度実験するのです。失敗を恐れることなく、毎回の実験で学んだことを将来の取り組みに活かしていきます。これを、**イノベーションのフライホイール**と呼ぶことにします。このフライホイールをすばやく回していくためには、製品をリリースし、フィードバックを収集し、新機能を追加して再度リリースするシステムが必要です。こうしたプロセスは、モダンアプリケーションが持つ特徴によって実現します。フライホイールを回し、迅速なイノベーションにより他社との競争で優位に立つことができるのです。

モダンアプリケーション開発

最も成功している企業の多くは、競合他社より優位に立てているのは自社のテクノロジーのおかげだと考えています。企業がビジネスを成長させ、成功を収めるには、新しい製品をすばやく発明していく必要があります。これを実現できるようなイノベーションのカルチャーを促進するため、成功を収めている企業はアプリケーションの設計、構築、管理の手法を継続的に更新しています。これを、**モダンアプリケーション開発**と呼ぶことにします。

モダンアプリケーション開発を活用する企業は、より迅速にイノベーションを実現し、競争力を獲得できます。イノベーションを重視する企業は、インフラストラクチャの管理やプロビジョニングのような画一的で面倒な作業から、もっと価値のある活動へとリソースを移行させせることで、実験回数を増やし、より短い時間でアイデアを市場に投入することができます。

モダンアプリケーション開発のプラクティスは、企業がイノベーションに見合ったスピードと俊敏性を実現するのに役立ちます。一部のお客様は、オンプレミスの仮想マシン (VM) をクラウドに移し、ホストするために (これは Lift & Shift とも呼ばれる)、Amazon Elastic Compute Cloud (Amazon EC2[1]) を使用されています。クラウド向けに最適化されたコンテナベースのモデルに、アプリケーションのプラットフォームを変更するお客様もいらっしゃいます。また、モノリシックアプリケーションをリファクタリングし、マイクロサービスベースのアーキテクチャに移行する企業もあります。大部分の企業では、クラウドネイティブのアプリケーションの構築によって、管理オーバーヘッドに費やす時間が減り、コアビジネスにさらに集中できることが明らかになっています。

モダンアプリケーションの機能

モダンアプリケーションは、次のような特徴を備えていることが必要です。

- **セキュリティ** – どのようなアプリケーションでも、セキュリテイは非常に重要です。アプリケーションの特定の部分のみでなく、すべての階層とライフサイクルの各段階に、セキュリティ対策を実装することが必要です。
- **弾力性** – モダンアプリケーションは弾力性を備えています。例えば、アプリケーションでは外部データソースの呼び出し時に障害が発生したとしても、そこで応答が停止することなく、呼び出しを再試行するか、例外処理を実行することが必要です。それと同時に、動作を継続させせつつ、機能をグレースフルに低下させていきます。このパターンは、マイクロサービスアーキテクチャ[2]、および他のサービスとのやり取りにも適用されます。
- **伸縮性** – モダンアプリケーションでは、リクエストの速度やその他のメトリクスに応じて、柔軟にスケールアウト/スケールインすることで、ビジネスチャンスを逃すことなくコストを最適化することができます。スケールアウトとスケールインのプロセスの自動化や、自動スケーリング機能を提供するマネージドサービスの使用により、日常的な管理業務の負担を軽減し、機能停止による深刻な中断を防止できます。
- **モジュール化** – モダンアプリケーションはモジュール化されており、高い凝集度と低い結合度が実現されています。大規模なシステムは単一のモノリスにしてしまうのではなく、ドメインの境界に従って、それぞれが特定の役割を果たす個別のコンポーネントへと分解します。このように分離することで、可用性とスケーラビリティが高まります。さらに、さまざまなコンポーネントを個別にデプロイできるようになるため、リリースも頻繁に実行できます。

- **自動化** – モダンアプリケーションでは、インテグレーションとデプロイを自動化し、高品質のリリースを頻繁に実施することが必要です。手動プロセスはエラーが発生しやすいだけではありません。特定の個人に依存してしまう可能性もあります。デプロイを行えるのは管理者 1 人だけ、といった状況が起こり得ます。アジャイルな開発と頻繁なリリースをサポートするため、モダンアプリケーションのデプロイは継続的インテグレーションと継続的デリバリー (CI/CD) のパイプラインによって実施することが必要です。CI/CD モデルの場合、コードがバージョン管理にプッシュされ、クリーンな CI 環境でテストが実行されます。そしてすべてのテストに合格すると、自動的にデプロイが実行されます。
- **相互運用性** – モダンアプリケーションでは、それぞれのサービスが別のサービスとやり取りし、リクエストされたリソースを提供し、要求されたタスクを実行します。また、他のサービスに影響を与えることなく、さまざまなサービスに個別に機能を追加し、頻繁なリリースを継続的に実施できることが求められます。つまり、各サービスの詳細な実装はプライベートに保ち、必要な機能は堅牢なパブリック API を通じて公開する必要があります。また、それらのパブリック API は、独立してリリースできるよう、安定した、下位互換性を持つものであることが求められます。

モダンアプリケーションの実装には、さまざまな手法を活用できます。このホワイトペーパーでは、コンテナとサーバーレステクノロジーを使用してアプリケーションをクラウドにデプロイする方法についての情報をご提供します。

モダンアプリケーション開発のベストプラクティス

AWS では、お客様や AWS の自社開発チームとの対話を通して、革新的なアイデアを市場に迅速に投入している組織には、モダンアプリケーション開発に関して共通するベストプラクティスがいくつかあることを発見しました。

セキュリティとコンプライアンス

AWS クラウドでのシステム構築では、いつでも最初にセキュリティとコンプライアンスについて考慮することをお勧めします。アプリケーションのライフサイクル全体を保護することで、イノベーションの速度を犠牲にすることなく、セキュリティの脅威に対処できます。次に例を示します。

- **認証** – 悪意あるアクセスを防止するようアクセス許可を設定し、システムへのアクセスを制御します。AWS 管理者は、AWS Identity and Access Management (IAM) の認証情報を使用して、または Microsoft Active Directory や SAML ID プロバイダーとの統合を通じて、AWS コンソールにサインインできます。AWS に構築されたアプリケーションであれば、Amazon Cognito を活用して、エンドユーザーを認証し、リソースへのアクセスを許可できます。
- **認可** – 柔軟なポリシーを使用してロールベースのアクセスコントロールを実装することで、管理を過度に複雑化することなく、リソースの使用を制限できます。IAM によって、すべての AWS リソースに対するきめ細かな認可ポリシーを使用できます。
- **監査とガバナンス** – ワークロードの動作を評価し、コンプライアンス要件と組織の標準に準拠した状態を確保します。AWS CloudTrail によって、AWS API を使用したやり取りを監査できます。また、Amazon CloudWatch でログを集約することで、アプリケーションの監査を実施できます。AWS Config では、AWS リソースの設定を確実に組織の標準に適合させることが可能です。
- **検証** – アプリケーションの機能の全側面をテストし、意図したとおりに機能することを確認します。継続的インテグレーションと継続的デリバリー (CI/CD) を使用して、可能な限り検証を自動化します。

モダンアプリケーションのテストは、徹底的かつ頻繁に実施する必要がありますが、開発速度を低下させることは許されません。同様に、開発者のアクセス許可を制限する必要がありますが、必要なアクセス許可を取り消すことはできません。アプリケーション

のライフサイクル全体にセキュリティを組み込み、セキュリティに関するプロセスと標準を自動化し、継続的に再評価することが求められます。

マイクロサービスアーキテクチャ

モノリシックアプリケーションでは、サイズの肥大化に伴って機能の変更や追加が困難になります。1 つの変更によってコードベースのどの部分が影響を受けるのかを追跡することも難しくなります。その結果、わずかな変更に対しても時間がかかる回帰テストが必要になり、新機能の開発が遅れてしまいます。マイクロサービスアーキテクチャと疎結合コンポーネントによって構築されたアプリケーションでは、新機能やバグ修正の多くを単一サービスのレベルで実装できるため、より迅速なリリースが可能になります。

現在モノリシックなレガシーアプリケーションを使用している組織であれば、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再設計することで、俊敏性と柔軟性を高めることができます。各サービスは個別にデプロイされますが、すべてのサービスが連携して動作することで、モノリシックシステムと同じ機能が実現されます。マイクロサービスの構築、変更、リリースは簡単に行えるため、実験とイノベーションにかかる時間を短縮できます。また、マイクロサービスを構築する各チームは、自分たちの設計、開発、デプロイ、運用について明確な所有権を獲得できます。

このような疎結合を実現するには、システム内のマイクロサービスが相互に通信する必要があります。1 つのデータストアが複数のサービスで共有されると、密結合、隠れた依存関係、タイミングの問題、スケーリングと可用性に関する問題が発生してしまいます。それよりも、独立したサービスの間で、公開 API や非同期メッセージキューを使用して通信する方が優れています。プロセスをさまざまな部分に分割し、キューのメッセージで接続するようにすれば、明確なトランザクション境界が作成されます。これにより、各サービスがさらに独立して動作できるようになります。

メッセージングシステムではスケーラビリティ、弾力性、可用性、整合性、さらに分散型のトランザクションを実現できます。これは、次のような特性を備えているためです。

- メッセージ配信システムに信頼性と弾力性がある
- ノンブロッキングな一方向の動作が可能
- サービスが疎結合されている
- システム内の異なる論理コンポーネントに重点が置かれ、各自が独立して動作可能

このような要素を活用したアーキテクチャでは、堅牢な API と非同期通信チャネルを簡単に公開できます。そのため、各サービスの運用と自動化を個別に実行できます。また信頼性も向上します。

1 つのプロセスを実行するために多数のマイクロサービスが接続されている場合、エンドツーエンドの単一タスクの状態をモニタリングする方法が必要になります。また、すべての必要なステップが正しい順序とタイミングで実行されていることを確認する必要もあります。ステートマシンを使用すれば、タスクの状態をモニタリングし、正しい順序で実行されていることを確認できます。

また、サービス間で全体的なワークフローを管理し、さまざまなタイムアウト、キャンセル、長時間実行されるタスクのハートビート、詳細なモニタリングと監査を設定するための方法も必要です。こうした種類の用途のツールでサービスを管理することによって、スピード、生産性、柔軟性を向上させることができます。マイクロサービスが適切なタイミングと順序で実行されるようにするため、モダンアプリケーションではオーケストレーションツールとメッセージングツールを使用します。オーケストレーションツールを使用すれば、繰り返し可能な方法で堅牢なサービスを簡単に構築できます。AWS Step Functions は、サービス間で任意のワークフローを調整できる、フルマネージド型のツールです。メッセージングツールを使用すると、サービス間の直接的な依存関係を排除し、信頼性とスケーラビリティを向上させることができます。特定のワークロードに応じて、Amazon Simple Queue Service (Amazon SQS)、Amazon CloudWatch Events、Amazon Kinesis といったさまざまなツールをご使用ください。オーケストレーションツールとメッセージングツールを併用することで、開発者

はワークフロー実行、ステート管理、サービス間通信に時間を費やす必要がなくなります。重要なビジネスロジックに貴重な時間を注ぐことができます。

サーバーレステクノロジーの使用

組織のアプリケーションを実行するためにサーバーやオペレーティングシステム (OS) の運用とメンテナンスを行う場合は、システム管理者が単純な反復タスク (OS へのセキュリティパッチ適用など) のために時間を費やさざるを得ません。また、リクエストのボリュームに応じて柔軟にスケールアップすることは難しいため、システム管理者は可用性と耐久性の要件を慎重に検討しながら、ピーク時のボリュームに備えてサーバーをあらかじめプロビジョニングしておく必要があります。こうして事前にプロビジョニングした余分のインフラストラクチャについても料金が発生します。実際に使った分のみを従量制で支払うわけにはいきません。

AWS Auto Scaling や AWS Systems Manager といったサービスの利用により、このような従来の VM ベースのインフラストラクチャにおける負担を軽減することが可能です。ただし、サーバーレステクノロジーでシステムを構築してしまえば、サーバーのプロビジョニングと管理さえ不要になります。システム管理者が OS へのパッチ適用に時間を費やす必要や、時折発生する使用のピークに備えて普段使わないリソースを維持しておく必要もありません。サーバーレスアプリケーションでは、需要に的確に対応できるよう、それぞれのコンポーネントがスケールします。信頼性と耐障害性についても、アプリケーションの大部分にデフォルトで組み込まれているため、システムのこれらの側面に必要な設計と運用の時間を大幅に削減できます。最初からサーバーレステクノロジーを使用してモダンアプリケーションを構築すれば、アプリケーションの構築、デプロイ、実行のライフサイクル全体を安全に保つことができます。運用上の複雑さが解消されれば、開発者は顧客を喜ばせる製品の構築に時間と労力を集中することができます。

AWS では、AWS Lambda[3] や AWS Fargate[4] といったサーバーレスコンピューティングサービスをご提供しています。オブジェクトストレージには Amazon Simple Storage Service (Amazon S3)[5] をご使用いただけます。サーバーレスデータベースには現在 2 種類の選択肢をご用意しています。高速で柔軟な NoSQL データベースである Amazon DynamoDB[6] と、Amazon Aurora でオンデマンド自動スケーリングを実現するよう設

定された Amazon Aurora Serverless[7] です。エンドツーエンドのサーバーレスアプリケーションを構築する場合、コンピューティング、データベース、ストレージ以外のサービスも必要になることがあります。API 管理、メッセージング、オーケストレーションから、トラブルシューティングやモニタリングに至るまで、ワークロード全体でその他の AWS のサーバーレス製品[8] をご利用いただけます。

CI/CD を使用したデプロイの自動化

顧客に対してできるだけ早く、可能な限り大きな価値を提供するため、企業は迅速なイノベーションに努めています。これを実現するため、モダンアプリケーションでは継続的インテグレーションと継続的デリバリー (CI/CD) を使用して、リリースプロセス全体、つまりコードのビルド、テストの実行、アーティファクトのステージング環境へのデプロイ、本番環境への最終デプロイを自動化します。CI/CD では、既知の脆弱性のスキャンや、静的分析の実行など、セキュリティコントロールの一部を自動化することも可能です。CI/CD パイプライン全体は、任意の数の品質ゲートとコントロールで構成されます。新しいコードは、本番環境への反映前にそのすべてに合格することが必要です。

ビルド、テスト、デプロイのプロセス全体が自動化されると、再現性のみでなく、速度も向上します。このプロセスはより頻繁に、おそらくは 1 日に何度も実行できるようになります。このため、1 回のデプロイに含まれる変更の数が減り、リスクも低くなります。CI/CD によって、本番デプロイは、関係者総出で対応する高リスクイベントから、日常的な業務へと変化します。さらに、コードのコミットからデプロイまでの時間が手動プロセスに比べて大きく短縮されるため、優先度の高いセキュリティ修正や設定変更にも特別なホットパッチが不要で、標準のパイプラインでリリースできます。

AWS では、オープンソースの選択肢やサードパーティーが Marketplace で提供するサービスに加えて、AWS CodeBuild、AWS CodePipeline、AWS CodeDeploy といったフルマネージド型の CI/CD サービスをご利用いただけます。

コードとしてのインフラストラクチャの管理

CI/CD を最大限に活用するには、アプリケーション全体と Infrastructure as Code (IaC) のモデルを作成する必要があります。インフラストラクチャはコードとしてモデル化すると、標準のアプリケーション開発ライフサイクルに組み込み、インフラストラクチャの変更を CI/CD パイプラインで実行できるようになります。これにより、設定エラーの減少やプロビジョニングの高速化といったメリットを得ることができます。AWS でご利用いただける IaC ツールをいくつかご紹介します。1 つ目は AWS CloudFormation[9] です。このサービスでは、必要なクラウドインフラストラクチャをシンプルなテンプレートファイル内で指定し、自動でプロビジョニングできます。2 つ目は AWS Serverless Application Model (SAM)[10] です。このツールは AWS CloudFormation をベースとしており、サーバーレスアプリケーション構築用のツールと便利な機能が追加されています。3 つ目は AWS Cloud Development Kit (CDK)[11] です。任意の言語を使用してコードの中でクラウドインフラストラクチャを設計し、CloudFormation でプロビジョニングするためのフレームワークが提供されます。

モニタリングとログ記録

モダンアプリケーション開発者には、モニタリングツールとログ記録ツールを使用してアプリケーションの実行時の動作をモニタリングし、そのデータを使用してカスタマーエクスペリエンスの維持や改善を図ることが求められます。最新のデジタル製品では、アプリケーションログ、モバイルデバイスからのデータ、ウェブのクリックストリーム、IoT センサーのデータ、その他の使用状況データなど、数多くの種類のデータをモニタリングする場合があります。モダンアプリケーション開発者は、製品の拡張と強化を続ける中で、それらすべてのデータを活用することが必要です。

AWS では、すべてのアプリケーションコンポーネントのモニタリング、ログ記録、アラームを Amazon CloudWatch で設定できます。ログ記録の詳細については、「**ログの集約**」をご参照ください。

モダンアプリケーションのチェックリスト

アプリケーションの最新化レベルの検証に、次の情報をご使用ください。

- セキュリティとコンプライアンスが、アプリケーションのライフサイクル全体に組み込まれている
- アプリケーションはマイクロサービスの集合として構造化されている
- 可能な限りサーバーレステクノロジーが使用されている
- 高品質の機能を迅速に提供するために CI/CD が使用されている
- インフラストラクチャはコードとして開発され、デプロイされている
- アプリケーションの動作について深く理解するためにモニタリングツールが使用されている

モダンアプリケーションの設計パターン

モダンアプリケーション開発のベストプラクティスの 1 つは、アプリケーションの設計と実装にパターンを使用することです。アプリケーションのビルディングブロックとして AWS のサービスをご使用いただくと、実装の労力が大幅に減り、信頼性と可用性が向上します。開発者は、アプリケーションに価値をもたらすビジネスロジックに注力することができます。

AWS のサービスを使用したマイクロサービスアーキテクチャの実装

一般的なパターンとベストプラクティスに従ってマイクロサービスを実装するのに、AWS のサービスをご使用いただけます。

API ゲートウェイ

API ゲートウェイパターンは、バックエンドサービスへの呼び出しが多数ある場合や、クライアントインターフェイスまたはデバイスタイプによって提供コンテンツが違う場合に使用できます。API ゲートウェイによって、さまざまなバックエンドサービスを統一された API の後ろで統合し、それぞれのデバイスに必要なコンテンツを提供できます。

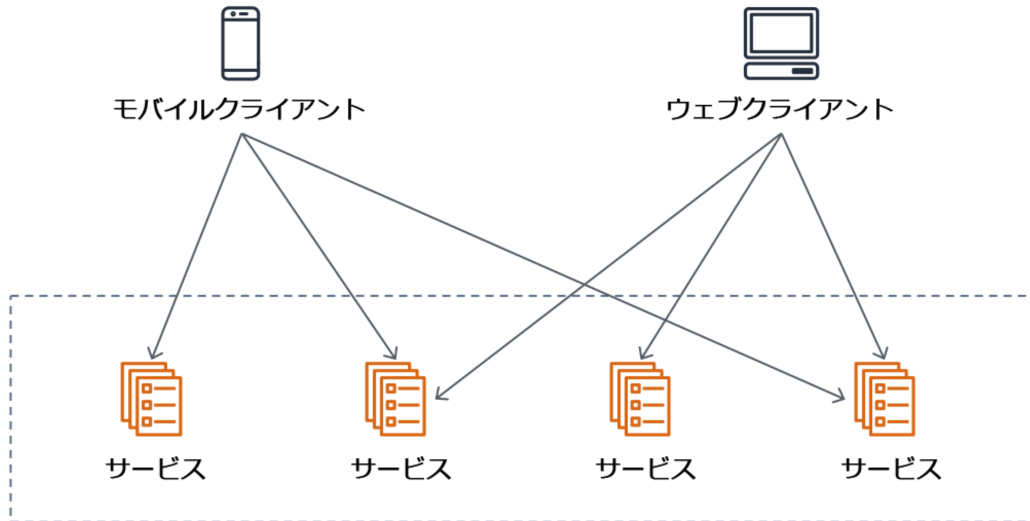

図 1 – サービスとモバイルデバイスやコンピュータブラウザの間での通信例 (API ゲートウェイを不使用)

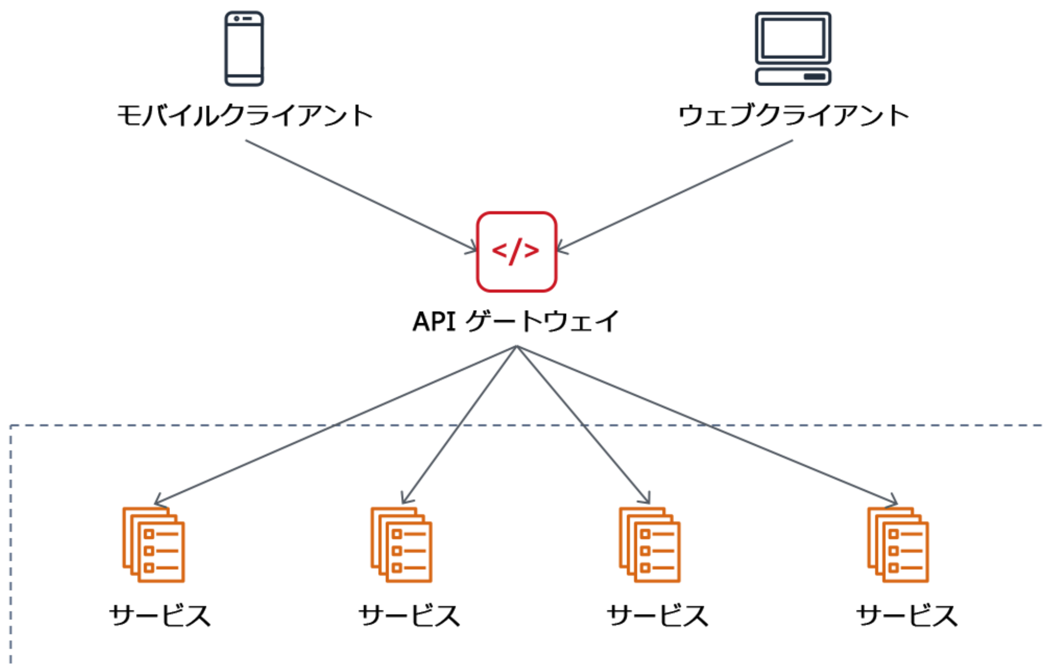

図 2 – サービスとモバイルデバイスやコンピュータブラウザの間での通信例 (API ゲートウェイを使用)

AWS クラウドで API ゲートウェイパターンを使用する場合、バックエンドのエンドポイントとの統合は Amazon API Gateway[12] で行えます。また、Amazon API Gateway によって、任意の規模で REST API や WebSocket API の作成、発行、保守、モニタリング、保護を行えます。

Amazon API Gateway は本番グレードの API に必要な他の多くの機能を備えています。そうした機能には、スロットリング、キャッシュ、ログ記録、API トークン、Amazon Cognito に統合された認証や認可、カスタムオーソライザー、他の AWS のサービス向けリクエストのプロキシなどがあります。Amazon API Gateway によるプロキシリクエストの送信先として重要な AWS のサービスの 1 つが、AWS Lambda です。Lambda は自由なウェブサービスを作成するための基盤になります。サーバーインフラストラクチャの管理は不要です。

Amazon API Gateway は AWS によって管理されるため、ユーザーによる運用や保守は不要です。Amazon API Gateway を使用することで安全性、信頼性、可用性が向上するため、開発者はアプリケーションの重要な機能により多くの時間を費やすことが可能です。

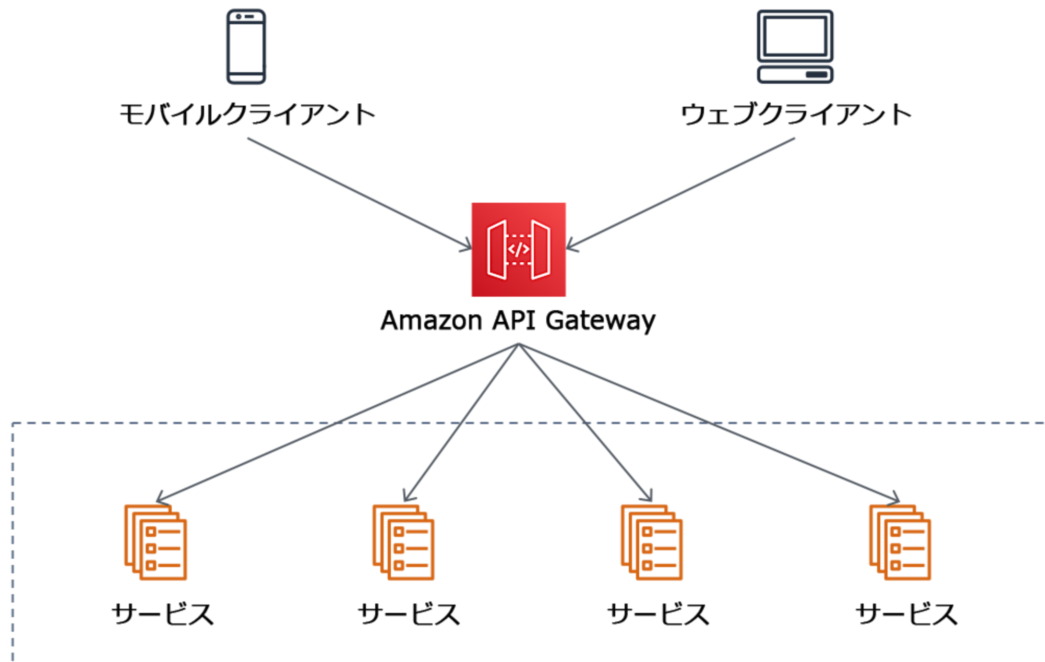

図 3 – サービスとモバイルデバイスやコンピュータブラウザの間での通信例 (Amazon API Gateway を使用)

サービスディスカバリーとサービスレジストリ

システムに複数のマイクロサービスが含まれている場合、各サービスが依存先のサービスの場所を見つけられることが必要です。マイクロサービスにはスケーラビリティや伸縮性が求められます。コンポーネントに障害が発生した場合は、新しいインスタンスやコンテナをオンラインにして、常に可用性を確保する必要があります。このため、マイクロサービス内のインスタンスやコンテナの IP アドレスは、絶えず変更される可能性があります。サービスのインスタンスそれぞれについても、可用性を継続的にモニタリングする必要があります。ロードバランサーを使用すると、可用性の高い安定したエンドポイントを実現できます。これは通常、パブリック側のウェブエンドポイントに最適です。ただし、ロードバランサー用のコンピューティングリソースを追加する必要があり、レイテンシーも発生します。クライアントや、マイクロサービス間の呼び出しを管理できるのであれば、**サービスディスカバリー**パターンを使用する方が効率的な場合があります。このパターンは、クライアント側での負荷分散と考えることもできます。

サービスディスカバリーパターンでは、検出される側のサービスについての情報が登録されていることが必要です。**サービスレジストリ**は、個々のコンテナやインスタンスの起動時に、呼び出されるサービスがそのサービス自体の情報を一元的に保存できる場所です。

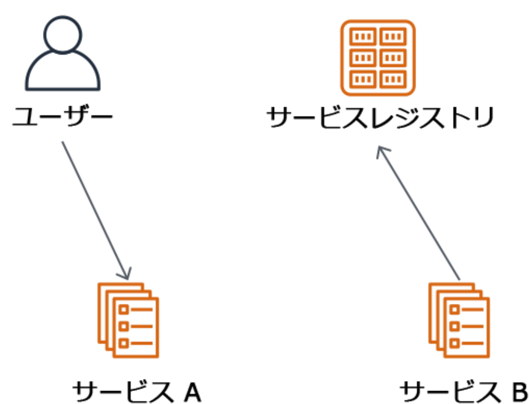

図 4 – サービスレジストリパターンの例

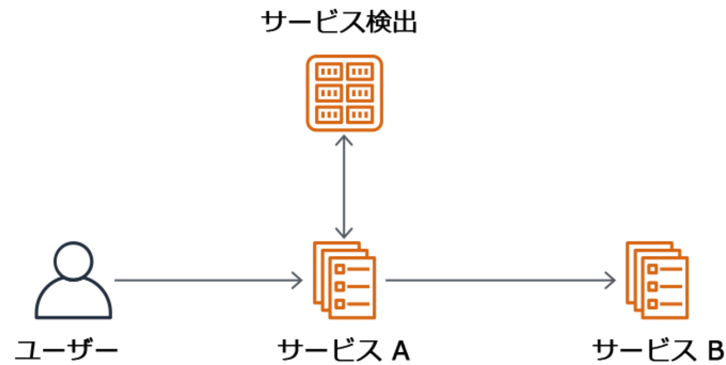

図 5 – サービスディスカバリーパターンの例

AWS クラウドでは、サービスレジストリとサービスディスカバリーパターンの実装に AWS Cloud Map をご使用いただけます。AWS Cloud Map はフルマネージドサービスです。このサービスを使用すると、クライアントが DNS を使用してサービスインスタンスの IP アドレスとポートの組み合わせを検索し、HTTP ベースのサービス検出 API 経由で URL や Amazon リソースネーム (ARN) などの抽象エンドポイントを動的に取得することが可能になります。

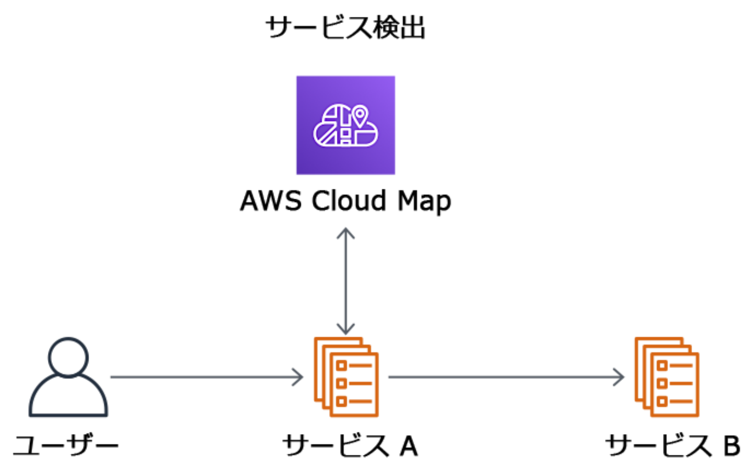

図 6 – AWS Cloud Map を使用したサービスレジストリとサービスディスカバリーパターンの例

サーキットブレーカー

サーキットブレーカーパターンは、アプリケーション内のマイクロサービス間の呼び出しを規制します。アプリケーション内のマイクロサービスは、ユーザーからのリクエストに応答して相互に呼び出しを行います。サービス A がサービス B に呼び出しを送信したものの、サービス B からの応答コールで遅延やエラーが発生した場合、サービス A はユーザーにエラーを返します。サービス A がエラーを返す代わりに呼び出しを再試行すれば、ユーザーエクスペリエンスが向上する可能性があります。しかし、再試行によって負荷の増大と長時間の遅延が発生し、サービス A は結局ユーザーにエラーを返して終了する可能性もあります。こうしたプロセスではなく、サービス A にサービス B がダウンしていることを認識させ、可能であればグレースフルデグラデーションを実行することが必要です。

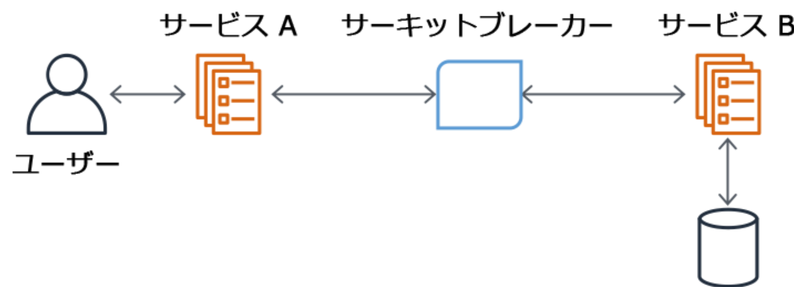

図 7 – マイクロサービス間での応答コールを使用したサーキットブレーカーパターンの例

サーキットブレーカーパターンでは、他のサービスの呼び出しに通常より長い時間がかかった場合やエラーが返された場合、サーキットブレーカーがインシデント数をカウントします。カウントが設定された制限を超えると**オープン**状態に移行します。オープン状態になると、サーキットブレーカーは呼び出し元に即座にエラーを返します。ダウンストリームのサービスは呼び出されません。一定時間が経過すると、サーキットブレーカーは**クローズ**状態に戻ります。ダウンストリームのサービスへの呼び出しも平常時の状態に戻ります。

図 8 – エラーが即座にユーザーに返されたサーキットブレーカーパターンの例

サーキットブレーカーの実装は、以前はサービスコード内のライブラリやフレームワークを使用して行うことがベストプラクティスでした。現在では、**サイドカー**を使用した、コンテナ化されたマイクロサービスで処理されることが多くなっています。サイドカーは、主要なサービスを公開するメインコンテナとともに起動される、独立したヘルパーコンテナです。Envoy Proxy[13] はサイドカーのよく知られた例です。Envoy Proxy は単独でもデプロイできますが、多くの場合、サービスメッシュの一部としてデプロイされます。このタイプのデプロイでは、Envoy Proxy が**データプレーン**となり、AWS App Mesh や Istio などのツールが**コントロールプレーン**になります。

コマンドクエリ責任分離

コマンドクエリ責任分離 (CQRS) とは、システムのデータミューテーションや**コマンド**部分を**クエリ**部分から分離することです。更新とクエリは、通常、単一のデータストアを使用して実行されます。2 つのワークロードのスループット、レイテンシー、整合性の要件が異なっている場合、CQRS を使用してそれらのワークロードを分離できます。コマンド関数とクエリ関数は分離すると、個別にスケールできるようになります。例えば、水平スケーラブルなリードレプリカにクエリを送信できます。コマンド関数とクエリ関数をさらに明確に分離する場合、更新用とクエリ用に別々のデータモデルとデータストアを使用できます。書き込みはオブジェクトリレーショナルマッピング (ORM) によって、リレーショナルデータベース内の正規化されたモデルに対して実行します。クエリは API (データ転送オブジェクトつまり **DTO** など) で要求されているのと同じ形式により、データが保存されている非正規化データベースに対して実行します。これにより、処理のオーバーヘッドを減らします。

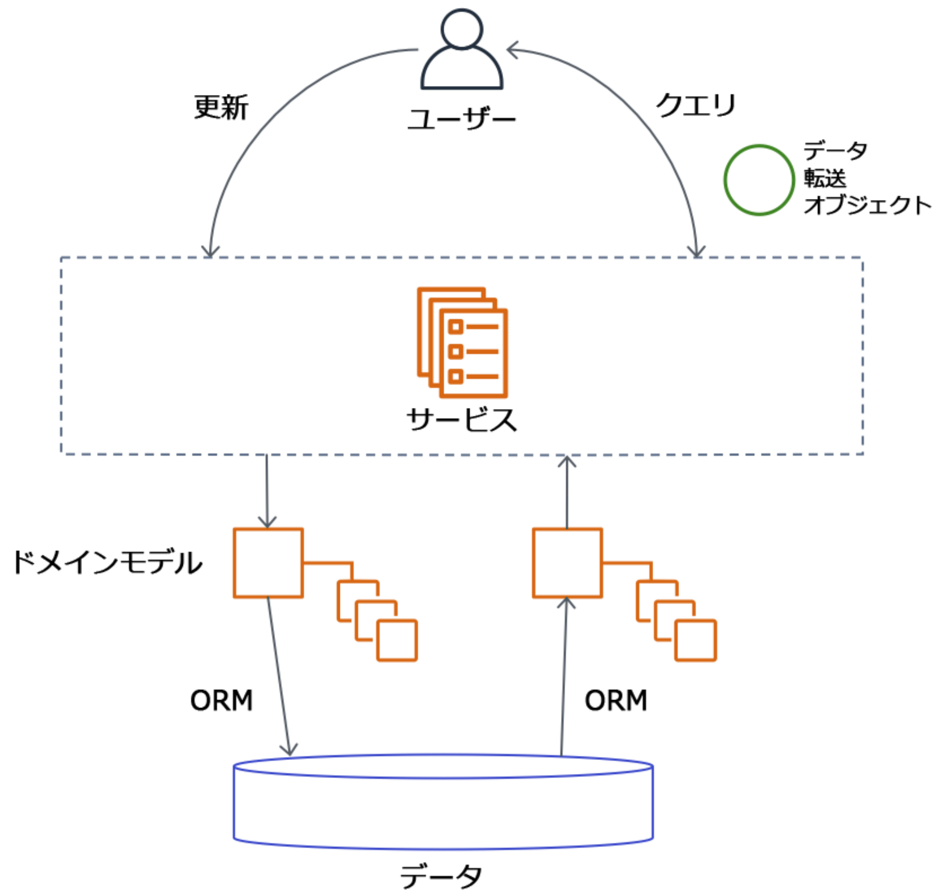

図 9 – 単一のデータストアと ORM を使用して更新とクエリを実行するアーキテクチャの例

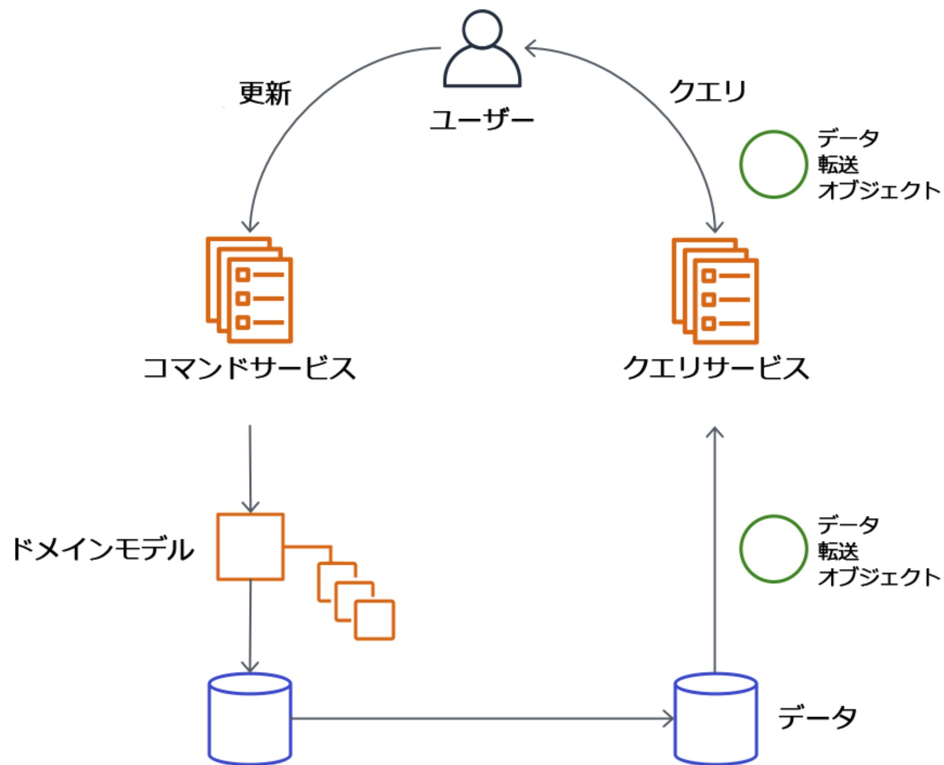

図 10 – コマンドとクエリを分離させたワークロードと
2 つのデータストアを使用した CQRS アーキテクチャの例

この例のアーキテクチャは、リレーショナルデータベースによる整合性のある書き込みと非常にレイテンシーの低い読み込みを実現するために最適化されています。ただし、非常に高い書き込みスループットと、柔軟なクエリ機能を実現するための最適化が必要な場合もあります。そのような状況では Amazon DynamoDB のような NoSQL データストアを使用し、明確に定義された特定のアクセスパターンを持つワークロードによって、データ追加時の書き込みスケーラビリティを高めることができます。そして、Amazon Aurora のようなリレーショナルデータベースを使用して、複雑な 1 回限りのクエリ実行機能を実現できます。このオプションでは、Amazon DynamoDB ストリームを使用してデータを AWS Lambda 関数に送信し、その関数によって Amazon Aurora のデータを適宜更新して、データを最新の状態に維持できます。

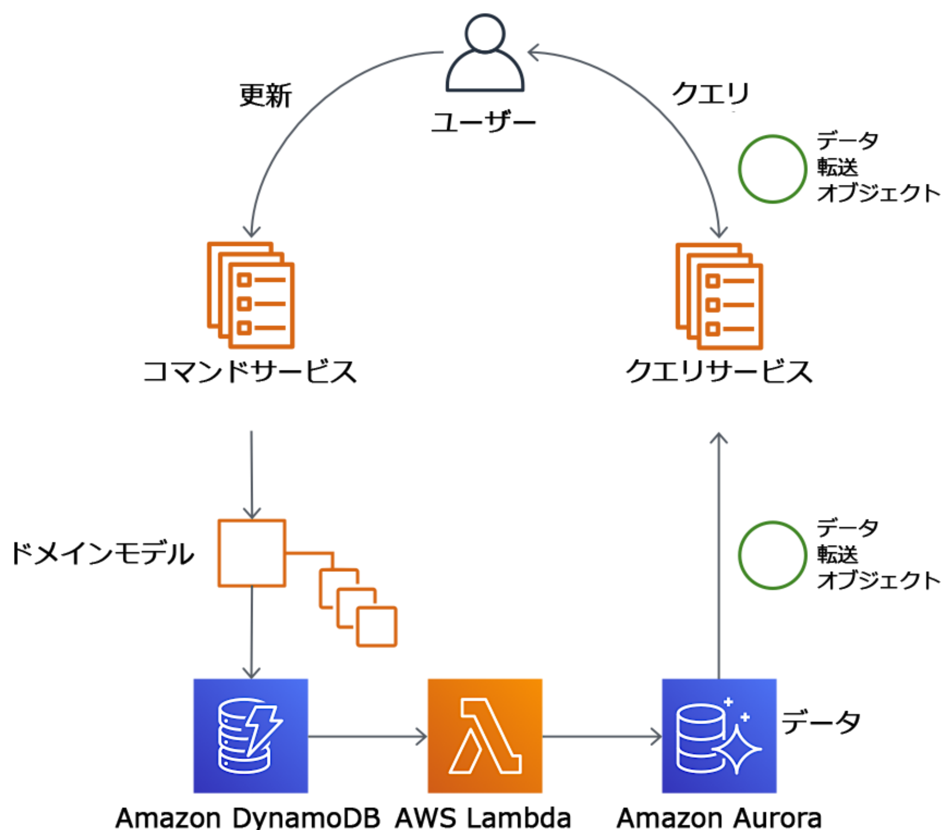

図 11 – DynamoDB、Lambda、Aurora を使用した AWS での CQRS アーキテクチャの例

CQRS アーキテクチャのコマンド部分を**イベントソーシング**パターン（続くセクションを参照）と組み合わせることもできます。これらのパターンを組み合わせると、更新イベントを再生することで、アプリケーションの最新の状態を使ってサービスクエリデータモデルを再構築できます。重要な注意点として、CQRS パターンの場合、通常、クエリされたデータストアと書き込み先データストアの間の整合性は結果整合性になります。

イベントソーシング

イベントソーシングパターンでは、データストアを直接更新する代わりに、注文の発注、クレジット照会、処理中または出荷中の注文など、ビジネスロジックに重要なイベントを耐久性のあるイベントログに追加します。各イベントレコードは個別に保存されるため、すべての更新は**アトミック**（分割不可かつ削減不可）になります。

このパターンの重要な特徴は、保存されたイベントを再度処理するだけで、アプリケーションのいかなる時点の状態でも再構築できることです。データはデータストアで直接更新されるのではなく、一連のイベントとして保存されます。このため、さまざまなサービスはイベントストアからイベントを再生することで、それぞれのデータストアの適切な状態を計算できます。イベントソーシングは、特に、コマンドとクエリのデータストアのスキーマが同じかどうかにかかわらずイベントのデータを再現できるため、前述の CQRS パターンでうまく機能します。

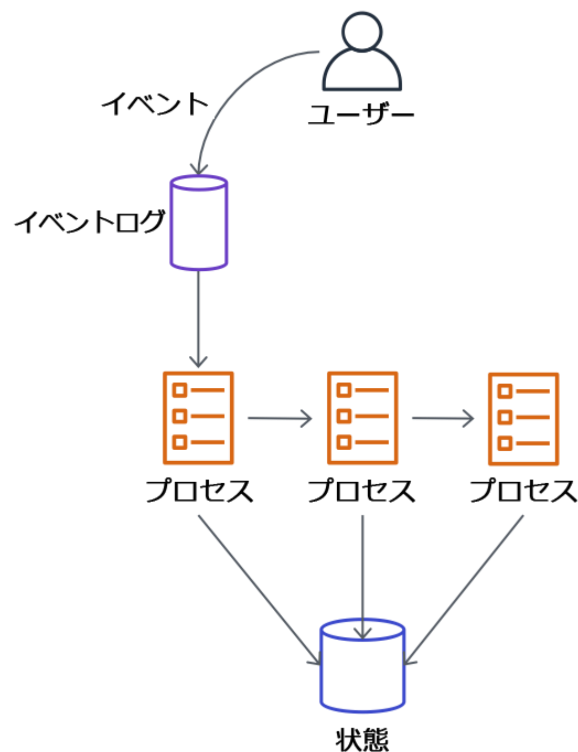

図 12 – イベントソーシングパターンの例

イベントソーシングパターンではイベントメッセージの保存と再生が実行されるため、メッセージの保存と取得のためのメカニズムが必要です。このパターンを AWS クラウドで採用する場合、ユースケースに応じて Amazon Kinesis Data Streams[14]、Amazon Simple Queue Service (SQS)[15]、Amazon MQ[16]、Amazon Managed Streaming for Kafka (Amazon MSK)[17]が利用可能です 。イベントソーシングパターンでは、システムを変更する各イベントが最初にメッセージキューに保存され、その後そのイベントに基づいてアプリケーションの状態に対する更新が実行されます。例

えば、あるイベントが Amazon Kinesis ストリームのレコードとして書き込まれると、AWS Lambda で構築されたサービスがそのレコードを取得し、独自のデータストアで更新を実行する、といったことが可能です。

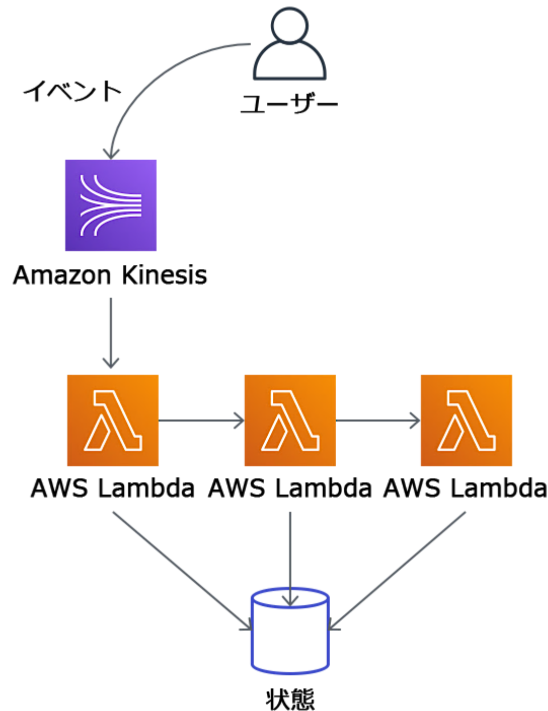

図 13 – Amazon Kinesis と AWS Lambda を使用したイベントソーシングパターンの例

複数のイベントから成る 1 つのソースを複数のターゲットに拡張すると便利な場合があります。Amazon Kinesis Data Streams ではこれを直接実現できます。複数のコンシューマーがストリームからデータを取得可能になります。また、Amazon Simple Notification Service (Amazon SNS) を使用して複数の Lambda 関数に拡張することもできます。それらの関数すべてが同じトピックをリッスンするようにして、イベントデータを Kinesis から他のステートフルコンポーネントに転送できます。このような構成の場合、Amazon SNS と特定の Lambda 関数の間に Amazon Simple Queue Service (Amazon SQS) キューを追加して、Lambda 関数を実行させるトリガーを指定することもできます。

コレオグラフィー

新しい顧客がウェブサイトでアカウントを作成するとしましょう。この際、顧客はプロファイル情報の保存、招待 E メールの受信、サイトで使える初期ポイントの取得といったことを行う必要があるかもしれません。これらのアクティビティはすべて、異なるサービスによって実装されます。

複数のマイクロサービス間でこうしたタスクを実行するための実装方法として、**オーケストレーション**パターンと**コレオグラフィー**パターンの 2 つがあります。**オーケストレーション**パターンでは、指揮者とオーケストラの関係のように、中央のサービスが他のサービスにコマンドを発行し、プロセス全体を確実に完了させます。コレオグラフィーパターンでは、ダンサーがダンスの振り付けを学習した後に独立して動くのと同じように、それぞれのサービスは特定のイベントに対応して個別に実行できます。

コレオグラフィーパターンを使用すると、必要なすべての情報を含んだ最初のイベントを 1 つのメッセージに保存して、最初のトランザクションを完了できます。その後、その他のサービスがそのメッセージを非同期的に取得し、それぞれのタスクを完了させます。このアーキテクチャでは、サービスが疎結合になり、直接互いに影響を与えることがありません。メッセージの保存と取得が非同期の関係になることには、スケーラビリティと信頼性の点でも利点があります。

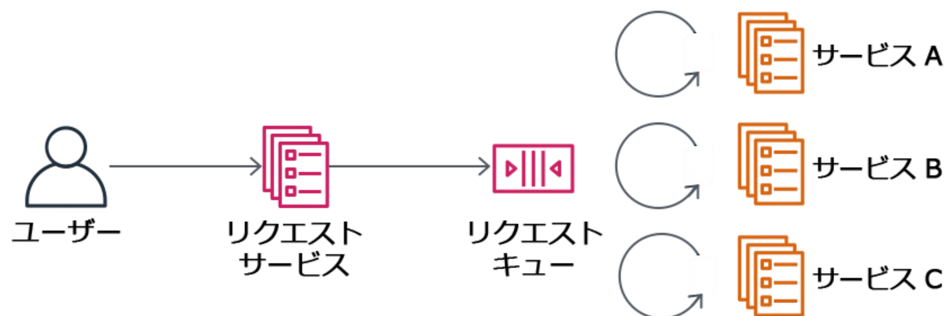

図 14 – コレオグラフィーパターンの例

AWS クラウドでコレオグラフィーパターンを実装するには、Amazon Kinesis と AWS Lambda をご使用いただくか、要件に応じて Amazon Simple Notification Service (Amazon SNS)、Amazon Simple Queue Service (Amazon SQS)、AWS Lambda を組み合わせてご利用ください。

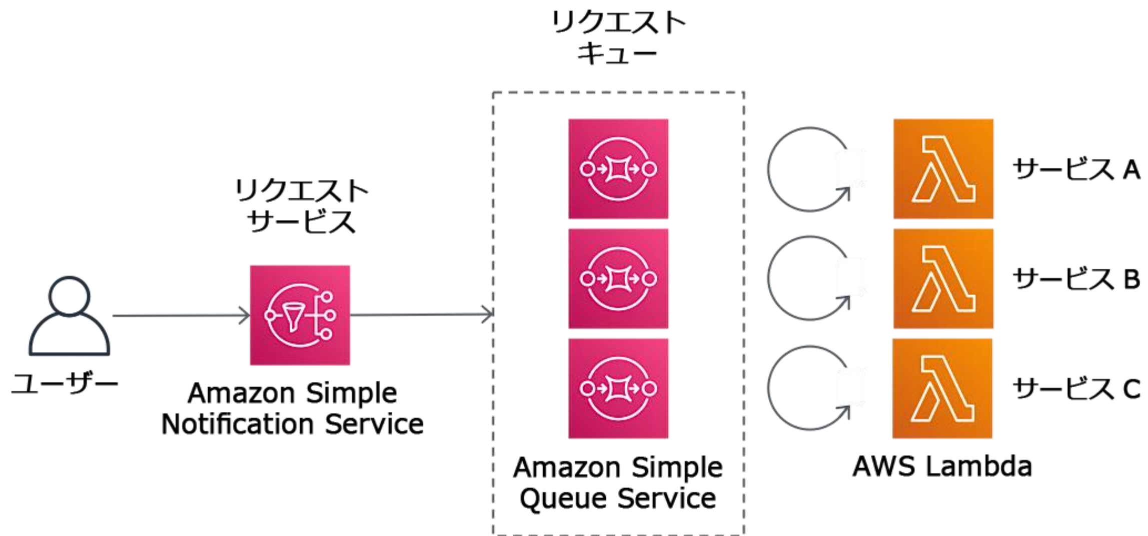

図 15 – Amazon SNS、Amazon SQS、AWS Lambda を使用したコレオグラフィーパターンの例

ログの集約

システムの複雑さが増すにつれて、ログの重要性が大きくなります。1 つの課題となるのは、ログがさまざまなサービスに分散している場合、システム全体をまとめて把握することが難しくなることです。ログを 1 つの場所で統一的に管理し、検出できるようにしておくことが欠かせません。また、メトリクスの収集も重要です。マイクロサービスアーキテクチャでは、1 件のリクエストを処理するためにさまざまなサービスを呼び出すことが必要な場合があります。そのため、モノリスと比較してパフォーマンス低下やエラーの原因を見つけることが難しくなる可能性もあります。このような理由により、ログと実行時のメトリクスを一元的に集約することが非常に重要です。

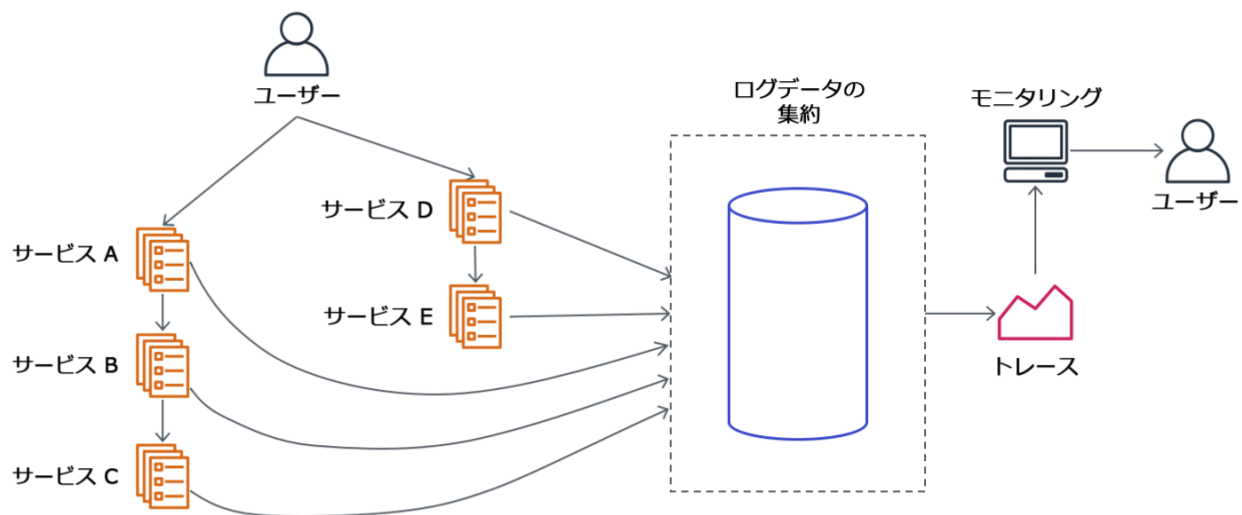

図 16 – ログ集計を使用したアーキテクチャの例

AWS クラウドでログを集約する場合、Amazon CloudWatch Logs[18] をご使用いただけます。AWS Lambda を使用してマイクロサービスを実装する場合、stdout に書き込むものがすべて CloudWatch Logs に送信されます。Amazon Elastic Container Service (Amazon ECS) と AWS Fargate では、stdout に書き込まれた内容を awslogs ログドライバーを使用して Amazon CloudWatch Logs に送信することも可能です。Amazon Elastic Kubernetes Service (Amazon EKS) を使用している場合、Fluentd[19] または Fluent Bit[20] によるサイドカーパターンを使用して、Amazon CloudWatch Logs にログを送信できます。Amazon ECS と AWS Fargate、または Amazon EKS で実行

されるコンテナ化されたアプリケーションについてのログとメトリクスは、CloudWatch Container Insights[21] を使用して CloudWatch に送信することも可能です。

サービス間の呼び出しの実行時間やエラーの追跡には、AWS X-Ray[22] をご使用いただけます。AWS X-Ray を使用することで、アプリケーションとその基盤サービスがどのように動作しているかを理解できるため、パフォーマンスの問題やエラーの根本的な原因を識別してトラブルシューティングできます。X-Ray では、アプリケーション内で転送されるリクエストがエンドツーエンドで表示され、アプリケーションの基盤となるコンポーネントのマップも表示されます。

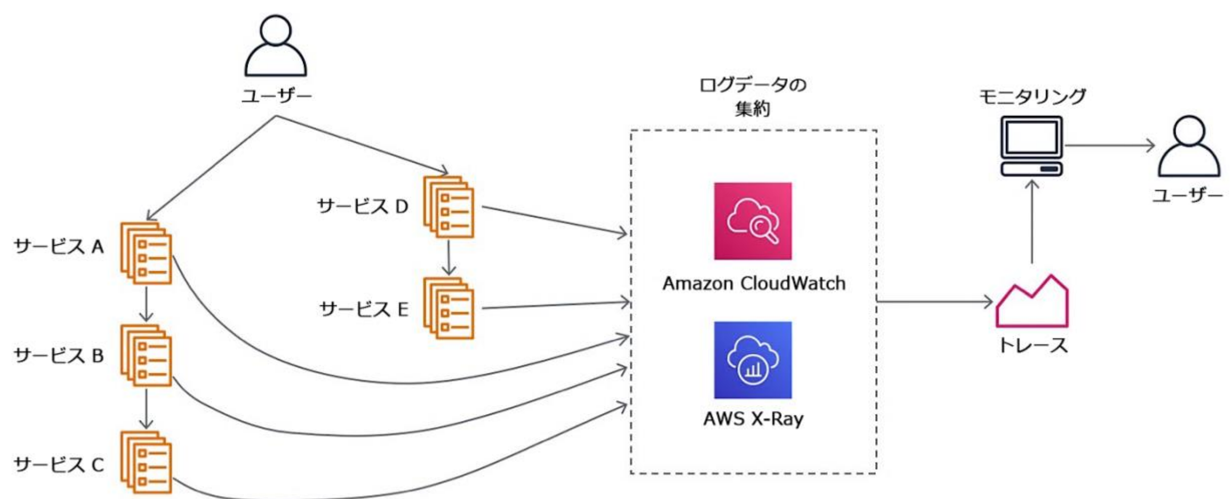

図 17 – Amazon CloudWatch と AWS X-Ray によるログ集計を使用したアーキテクチャの例

多言語パーシステンス

マイクロサービスアーキテクチャでは、各サービスがパブリック API を公開していることや、それぞれのサービスの実装に関する詳細が他のサービスから隠されていることが必要です。また、サービスの構築チームは API の約束事を守っている限り、サービスの内部を自由に変更して構いません。変更対象のコードに依存する他のサービスについて心配する必要はありません。さらに、各チームは自分たちが所有するサービスに対して必要に応じてデプロイを行うことや、サービスの実装に使用するプログラミング言語やデータベースを自由に選ぶことができます。**多言語パーシステンス**があれば、データのアクセスパターンとサービスのその他の要件に基づいて、適切なデータストレージテクノロジーを選択できます。

すべてのサービス構築チームが同じデータストレージテクノロジーを使用することが必要な場合、データストアが各チームの状況に適していなければ、実装の問題やパフォーマンスの低下が発生する可能性があります。各チームがチーム固有の要件に最も適したデータストアを選択できるのであれば、サービスの実装は簡単になり、パフォーマンスとスケーラビリティが向上します。

AWS では多言語パーシステンスを実現するためのデータストレージサービスをご提供しています。それらのサービスを次の表にまとめました。

表 1 – 多言語パーシステンスに対応した AWS データストレージサービス

データストア	機能
Amazon DynamoDB	あらゆる規模で 10 ミリ秒未満のパフォーマンスを実現する、キーバリューデータベースおよびドキュメントデータベースです。マルチリージョンとマルチマスターに対応し、耐久性に優れたフルマネージドデータベースです。インターネットスケールのアプリケーションに対応した、組み込みのセキュリティ、バックアップと復元の機能、インメモリキャッシュを備えています。DynamoDB では、1 日あたり 10 兆件を超えるリクエストを処理でき、ピーク時で 1 秒あたり 2,000 万件を超えるリクエストに対応できます。
Amazon Aurora と Amazon Relational Database Service (RDS)	Amazon Aurora は、MySQL および PostgreSQL と互換性のあるリレーショナルデータベースで、クラウドに合わせて構築されています。このデータベースは、オープンソースデータベースのシンプルさとコスト効率を備え、従来のエンタープライズ用データベースのパフォーマンスと可用性を併せ持っています。 Amazon Relational Database Service (Amazon RDS)[23] を使用すると、クラウド内でのリレーショナルデータベースの設定、運用、スケーリングを簡単に実行できます。Amazon RDS はコスト効率に優れ、サイズ変更が可能なキャパシティーを提供しながら、ハードウェアのプロビジョニング、データベースのセットアップ、パッチ適用、バックアップといった時間のかかる管理タスクを自動化します。これによりユーザーは、アプリケーションに集中できるようになるため、アプリケーションで必要とされる高速なパフォーマンス、高可用性、セキュリティ、互換性を実現できます。
Amazon ElastiCache	Amazon ElastiCache では、フルマネージド型の Redis と Memcached が提供されています。広く使われているオープンソース互換インメモリデータストアのデプロイ、実行、スケールをシームレスに実行できます。データ集約型のアプリケーションを構築することや、高スループットかつ低レイテンシーのインメモリデータストアからのデータ取得によって既存アプリケーションのパフォーマンスを向上させることが可能です。

データストア	機能
Amazon EBS	Amazon Elastic Block Store (EBS) は、Amazon Elastic Compute Cloud (EC2) と組み合わせて使用できるように設計された、使いやすく、高性能なブロックストレージサービスです。スループット負荷が高いワークロードでも、トランザクション負荷が高いワークロードでも、あらゆる規模で利用できます。 Amazon EBS ボリュームのデータは、1 つのコンポーネントで障害が発生した場合のデータ損失を防ぐため、アベイラビリティーゾーン内の複数のサーバーにレプリケートされます。Amazon EBS ボリュームは、ワークロードの実行に必要とされる安定した低レイテンシーのパフォーマンスを実現します。Amazon EBS では使用量の拡張と縮小を分単位で行うことができます。さらに、プロビジョニングしたサイズに合わせて、低料金でご利用いただけます。
Amazon EFS	Amazon Elastic File System (Amazon EFS) は、AWS クラウドサービスとオンプレミスリソースでの Linux ベースのワークロード向けに、シンプルでスケーラブル、かつ伸縮自在なファイルシステムを提供します。アプリケーションを中断することなく、オンデマンドでペタバイト規模までスケールできるよう構築されており、ファイルの追加や削除に合わせて自動的に拡大/縮小します。このため、アプリケーションでは必要なときに必要なサイズのストレージを利用できます。 Amazon EFS は数千の Amazon EC2 インスタンスからの大量の同時アクセスを処理できるよう設計されているため、アプリケーションにおいて高レベルの集約スループットと IOPS を一貫した低レイテンシーで実現できます。
Amazon S3	Amazon Simple Storage Service (Amazon S3) は、業界をリードするスケーラビリティ、データ可用性、セキュリティ、パフォーマンスを備えたオブジェクトストレージサービスです。つまり、規模や業界を問わず、このサービスをご使用いただくと、ウェブサイト、モバイルアプリケーション、バックアップと復元、アーカイブ、エンタープライズアプリケーション、IoT デバイス、ビッグデータ分析といったさまざまなユースケースでどのような量のデータでも保存し、保護することができます。

AWS での継続的インテグレーションと継続的デリバリー

継続的インテグレーション (CI) と継続的デリバリー (CD) はモダンアプリケーション開発の価値を実感するうえで不可欠であるため、前述のベストプラクティスを AWS で実装する方法について慎重にご検討ください。AWS では、「**CI/CD を使用したデプロイの自動化**」セクションでご説明しているようにモダンアプリケーションをすばやく提供するのに役立つさまざまなサービスをご利用いただけます。これらのサービスはフルマネージド型であるため、開発チームの皆さまには、CI サーバーの管理や保護といった画一的で面倒な作業ではなく、デプロイの自動化や新しい機能の迅速な提供に注力していただけます。

AWS での CI/CD サービス

AWS クラウドでの CI/CD デプロイには AWS の次のサービスをご使用いただけます。

AWS Cloud9

AWS Cloud9[24] は、ブラウザのみでコードを記述、実行、デバッグできるクラウドベースの統合開発環境 (IDE) です。このサービスには、コードエディタ、デバッガー、ターミナルが含まれています。AWS Cloud9 では、JavaScript、Python、PHP などの一般的なプログラミング言語向けの基本的なツールが用意されているため、新しいプロジェクトを始めるのにファイルをインストールしたり、開発マシンを設定したりする必要がありません。

AWS Cloud9 IDE はクラウドベースであるため、オフィス、自宅、またはインターネットに接続されたマシンのある任意の場所からプロジェクトで作業できます。また、AWS Cloud9 では、シームレスな操作でサーバーレスアプリケーションを開発できます。これにより、リソースの定義、デバッグ、サーバーレスアプリケーションのローカル実行とリモート実行の切り替えを簡単に行えます。AWS Cloud9 を使用すると、開発環境をチームで簡単に共有できるため、ペアプログラミングを行ったり、お互いの入力をリアルタイムで追跡したりすることができます。

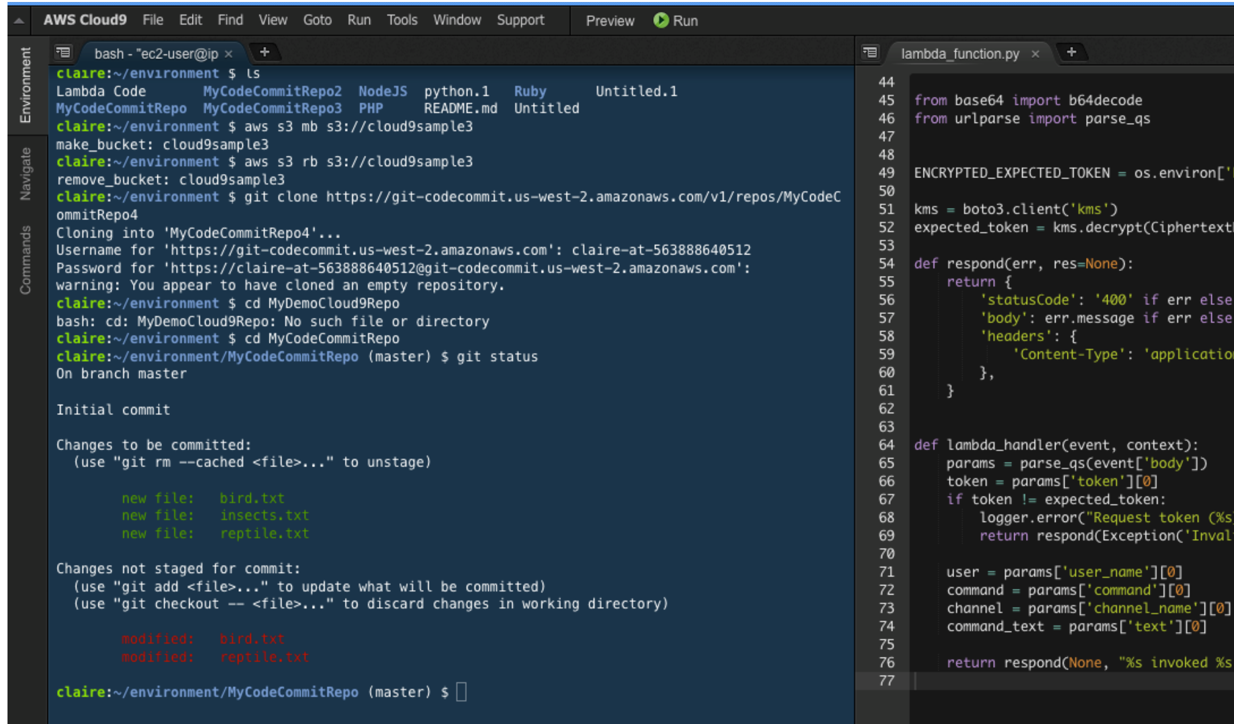

図 18 – AWS Cloud9 のコードの例

AWS CodeStar

AWS CodeStar[25] を使用すると、AWS クラウドでアプリケーションをすばやく開発、構築、デプロイできます。AWS CodeStar の統合されたユーザーインターフェイスによって、ソフトウェア開発アクティビティを 1 つの場所で簡単に管理できます。AWS CodeStar では、継続的デリバリーツールチェーン全体を数分でセットアップできるため、コードのリリースをすばやく開始することができます。また、AWS CodeStar によりチーム全体で安全な連携を簡単に行うことができます。アクセスの管理や、プロジェクトの所有者、担当者、閲覧者の追加を簡単に実行できます。各 AWS CodeStar プロジェクトには、作業項目のバックログからチームによる最近のコードデプロイまで、ソフトウェア開発プロセス全体の進行状況を簡単に追跡できるプロジェクト管理ダッシュボードが用意されています。

AWS CodePipeline

AWS CodePipeline[26] は、フルマネージド型の継続的デリバリーサービスです。リリースパイプラインを自動化し、アプリケーションとインフラストラクチャの更新を高速に、かつ信頼性の高い方法で実行できます。CodePipeline では、コードの変更があるたびに、ユーザーが定義したリリースモデルに基づいてリリースプロセスのビルド、テスト、デプロイの各フェーズを自動的に実行します。これにより、機能と更新をすばやく、信頼性の高い方法で配信できます。

AWS CodeCommit

AWS CodeCommit[27] は Git ベースの安全なリポジトリをホストするフルマネージド型のソース管理サービスです。[28] 安全で非常にスケーラブルなエコシステムで、チームと簡単にコードの共同作業を行うことができます。CodeCommit を使用することにより、独自のソース管理システムを運用したり、そのインフラストラクチャをスケールしたりする必要がなくなります。CodeCommit は、ソースコードからバイナリまで、あらゆる要素を安全に保存するために使用できます。また既存の Git ツールとシームレスに連動します。

AWS CodeBuild

AWS CodeBuild[29] は、フルマネージド型の継続的インテグレーションサービスです。ソースコードのコンパイル、テストの実行、すぐにデプロイできるソフトウェアパッケージの生成を行います。CodeBuild により、ビルドサーバーのプロビジョニング、管理、スケーリングが不要になります。CodeBuild では、スケーリングが継続的に行われ、複数のビルドが同時に実行されます。ビルドの実行までキューで待つ必要はありません。パッケージ済みのビルド環境で、すぐに開始できます。お使いのビルドツールを使用するために、カスタムビルド環境を作成することもできます。

AWS CodeDeploy

AWS CodeDeploy[30] は、フルマネージド型のデプロイサービスです。Amazon Elastic Compute Cloud (Amazon EC2)、AWS Fargate、AWS Lambda といったさまざまなコンピューティングサービスとオンプレミスサーバーへのソフトウェアのデプロイを自動化します。AWS CodeDeploy を使用すると、新しい機能を迅速にリリースして、アプリケーションのデプロイ中のダウンタイムを回避できます。AWS CodeDeploy では、アプリケーションの複雑な更新も実行できます。さらに、エラーの原因となりやすい手動オペレーションを排除し、ソフトウェアのデプロイを自動化できます。このサービスは、デプロイのニーズに合わせてスケールします。

AWS Amplify コンソール

AWS Amplify コンソール[31] では、Git ベースのワークフローを利用して、フルスタックのサーバーレスウェブアプリケーションをデプロイしてホストできます。フルスタックのサーバーレスアプリケーションは、GraphQL[32] や REST API などのクラウドリソースで構築されたバックエンド、ファイルとデータストレージ、さらに React[33]、Angular[34]、Vue[35]、Gatsby[36] といったシングルページアプリケーションフレームワークで構築されたフロントエンドで構成されます。フルスタックのサーバーレスウェブアプリケーションの機能は、多くの場合、ブラウザで実行されるフロントエンドコードとクラウドで実行されるバックエンドのビジネスロジックに分散されています。このため、アプリケーションのデプロイは複雑になり、時間もかかります。リリースサイクルを慎重に調整して、フロントエンドとバックエンドの互換性を確保し、新しい機能によって本番環境の顧客に中断が発生しないようにする必要があるためです。Amplify コンソールでは、フルスタックのサーバーレスアプリケーションをデプロイするためのシンプルなワークフローを提供することで、アプリケーションリリースサイクルを加速させます。ユーザーがアプリケーションのコードリポジトリを Amplify コンソールに接続すると、フロントエンドとバックエンドへの変更は、コードコミットごとに単一のワークフローでデプロイされます。

さまざまなアプリケーションタイプに合わせた CI/CD パターン

CI/CD パターンは、AWS クラウドにデプロイするモダンアプリケーションの主要な各タイプに使用できます。AWS ネイティブの開発ツールをご使用いただくと、CI/CD をすばやく実装できます。複雑な CI 環境のセットアップと管理に伴う面倒な作業に悩まされることはありません。AWS クラウドで CI/CD パターンを使用する方法ついて、次の例をご紹介します。

シングルページアプリケーションのデプロイ

シングルページアプリケーション (SPA) は、ブラウザにダウンロードされる静的コンテンツ (HTML、CSS、JavaScript、メディア) で構成されます。ブラウザからバックエンド API への呼び出しが実行されます。AWS Amplify コンソールを使用すると、SPA をすばやく構築およびリリースできます。AWS Amplify コンソールでは、GitHub[37] や AWS CodeCommit などのリポジトリに新しいコードがプッシュされたことを自動的に検出します。次に、静的フロントエンドコンテンツを Amazon Simple Storage Service (Amazon S3) にデプロイし、コンテンツ配信ネットワークである Amazon CloudFront[38] を経由してコンテンツをユーザーに配信します。また、GraphQL と REST API、認証、分析、Amplify CLI で作成したストレージで構成されるサーバーレスバックエンドに変更をデプロイすることもできます。

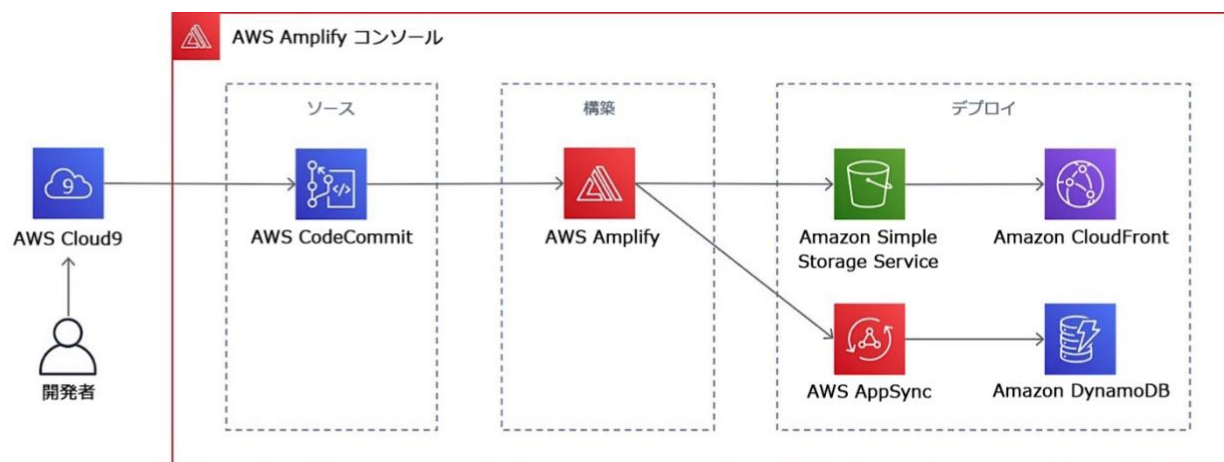

図 19 – シングルページアプリケーションのデプロイのアーキテクチャ例

コンテナへのデプロイ

AWS CodePipeline を使用すると、最小限の構成で Amazon Elastic Container Service (Amazon ECS) コンテナオーケストレーションサービスへ継続的にデプロイできます。ソースステージでは、AWS CodePipeline によってソースコードリポジトリの変更が自動的に検出されます。構築ステージでは、AWS CodeBuild を使用して Docker イメージが構築され、Amazon Elastic Container Registry (ECR)[39] などの Docker リポジトリにそのイメージがプッシュされます。最後に、AWS CodePipleline によって Amazon ECS にデプロイされます。

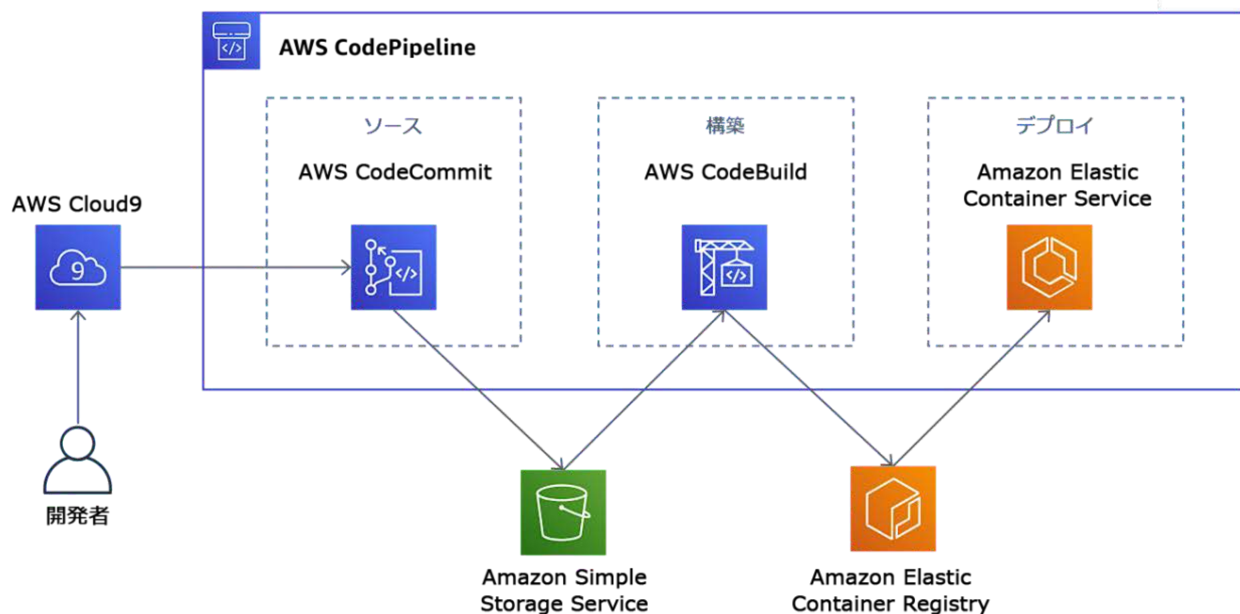

図 20 – コンテナへのデプロイのアーキテクチャ例

コンテナへのデプロイ (ブルー/グリーンデプロイ)

Amazon ECS と AWS CodeDeploy は、コンテナへのブルー/グリーンデプロイもサポートしています。AWS CodeDeploy は、Amazon Elastic Load Balancing[40] の種類の 1 つである Application Load Balancer (ALB) を使用して、2 つの並列ターゲットグループ間でトラフィックをスムーズに切り替えることで、ブルー/グリーンデプロイを自動化します。

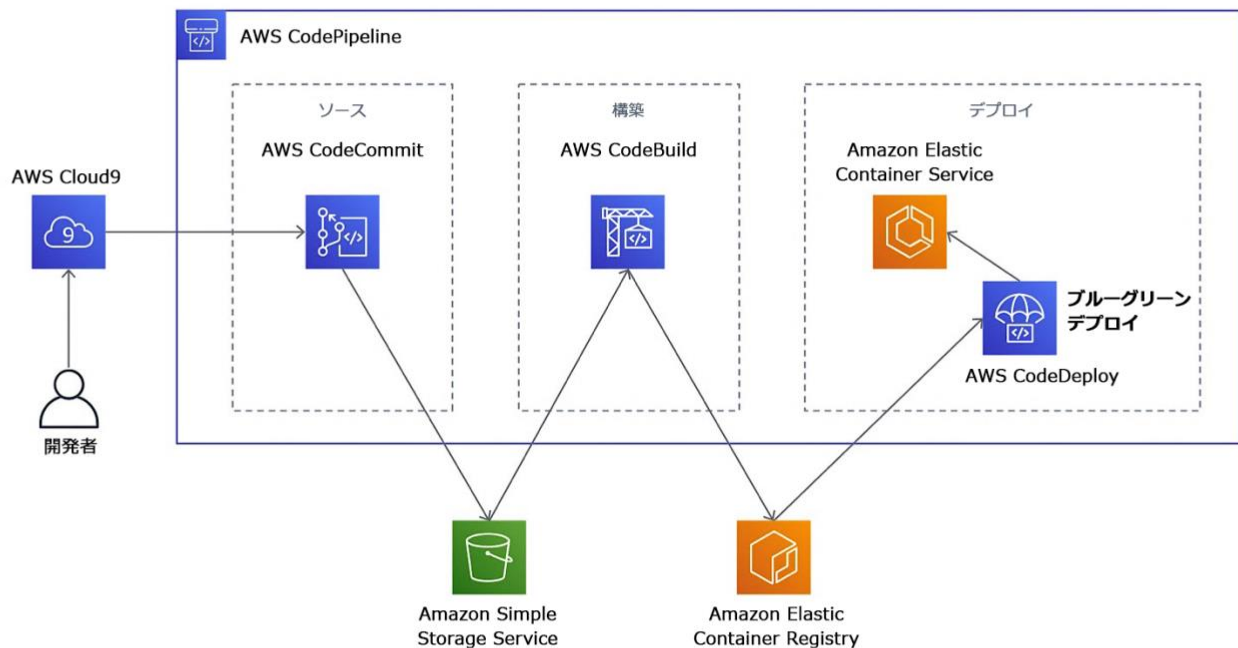

図 21 – コンテナへのブルー/グリーンデプロイのアーキテクチャ例

AWS Lambda へのカナリアデプロイ

AWS CodeDeploy は、AWS Lambda へのカナリアデプロイもサポートしています。AWS CodeDeploy では Lambda のトラフィック移行機能を使用して、関数の新しいバージョンの段階的なロールアウトを自動化します。これにより、2 つのバージョン間でトラフィックを段階的に移行できるため、リスクを軽減したり、Lambda の新しいデプロイによる影響を抑えたりすることができます。

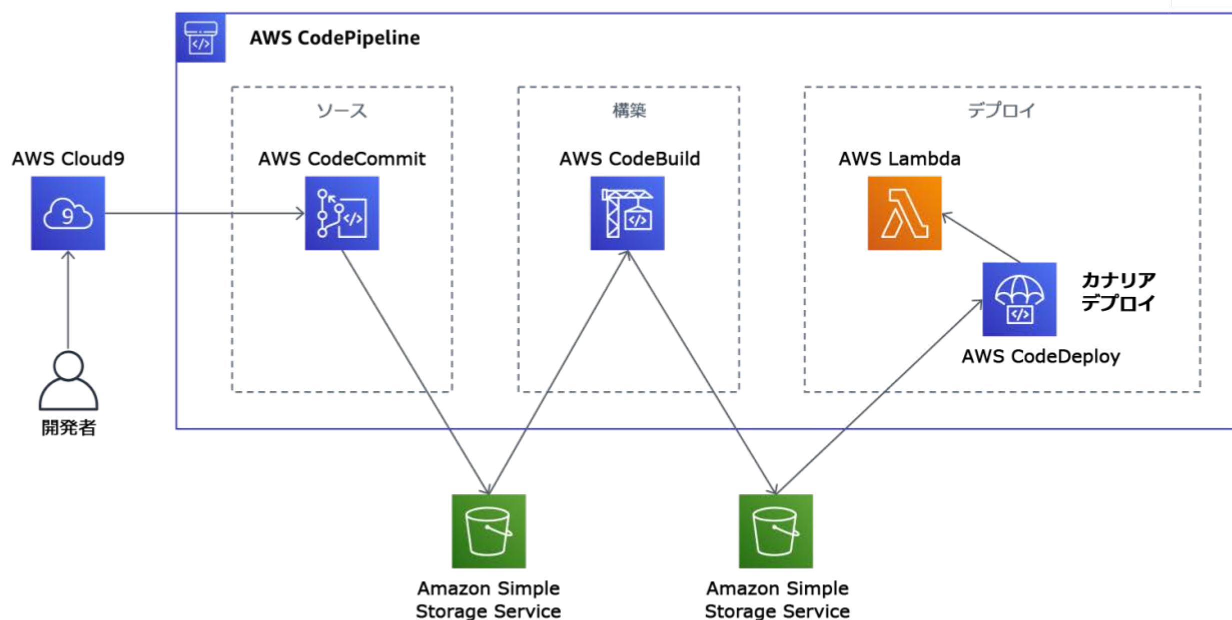

図 22 – AWS クラウドでのカナリアデプロイのアーキテクチャ例

AWS CodeDeploy を使用して AWS Lambda のデプロイを実行するときは、事前定義されたデプロイ設定オプション (以下の表を参照) のいずれかを使用するか、カスタム設定を自分で作成することができます。また、これらのオプションはすべて、サーバーレスアプリケーションモデル (SAM) に基づくアプリケーションをデプロイするときにも使用できます。

表 2 – AWS Lambda と AWS CodeDeploy を使用したカナリアデプロイ用の事前定義されたデプロイ設定オプション

デプロイ設定	説明
CodeDeployDefault.LambdaCanary10Percent5Minutes	最初の段階でトラフィックの 10% を移行します。残りの 90% は 15 分後にデプロイされます。
CodeDeployDefault.LambdaCanary10Percent10Minutes	最初の段階でトラフィックの 10% を移行します。残りの 90% は 10 分後にデプロイされます。
CodeDeployDefault.LambdaCanary10Percent15Minutes	最初の段階でトラフィックの 10% を移行します。残りの 90% は 15 分後にデプロイされます。
CodeDeployDefault.LambdaCanary10Percent30Minutes	最初の段階でトラフィックの 10% を移行します。残りの 90% は 30 分後にデプロイされます。
CodeDeployDefault.LambdaLinear10PercentEvery1Minute	すべてのトラフィックを移行するまで、毎分トラフィックの 10% を移行します。
CodeDeployDefault.LambdaLinear10PercentEvery2Minutes	すべてのトラフィックを移行するまで、2 分おきにトラフィックの 10% を移行します。
CodeDeployDefault.LambdaLinear10PercentEvery3Minutes	すべてのトラフィックを移行するまで、3 分おきにトラフィックの 10% を移行します。
CodeDeployDefault.LambdaLinear10PercentEvery10Minutes	すべてのトラフィックを移行するまで、10 分おきにトラフィックの 10% を移行します。
CodeDeployDefault.LambdaAllAtOnce	更新された Lambda 関数に一度にすべてのトラフィックを移行します。

まとめ

現代の企業には、競合他社に打ち勝つために世界規模でビジネスを展開し、デジタルイニシアチブに投資することが求められています。ユーザーによるデジタル製品の利用方法の変化に応じて、カスタマーエクスペリエンスを向上させ、ますます多様化するユーザーを満足させる必要があります。ユーザーの高い期待に応えるには、企業は失敗を恐れず、常にユーザーからのフィードバック内容を実際に試して製品に取り入れていく必要があります。

モダンアプリケーション開発とは、迅速な更新とリリースを行うための考え方と手法のことです。このような最新の手法を採用した開発チームは、可能な限りマネージドサービスを使用して反復作業を自動化することで、画一的で面倒な作業から解放されます。その結果、お客様を喜ばせる製品の構築に多くの時間を費やすことが可能になるのです。

モダンアプリケーション開発のベストプラクティスを巧みに導入することで、実験とイノベーションを迅速に進めることができるようになります。さらに、そうしたベストプラクティスの導入に AWS のネイティブサービスを利用されると、お客様の組織で俊敏性を実現していただくことができます。

寄稿者

この文書の寄稿者は次のとおりです。

- 福井 厚、ソリューションアーキテクト、アマゾン ウェブ サービス
- Kevin Bell、ソリューションアーキテクト、アマゾン ウェブ サービス

その他の資料

詳細については、次のリソースをご参照ください。

AWS のサービス

- Amazon API Gateway
  https://docs.aws.amazon.com/apigateway/latest/developerguide/welcome.html
- AWS Cloud Map
  https://docs.aws.amazon.com/cloud-map/latest/dg/what-is-cloud-map.html
- AWS Lambda
  https://docs.aws.amazon.com/lambda/latest/dg/welcome.html
- Amazon Kinesis Data Streams
  https://docs.aws.amazon.com/streams/latest/dev/introduction.html
- Amazon Simple Queue Service
  https://docs.aws.amazon.com/AWSSimpleQueueService/latest/SQSDeveloperGuide/welcome.html
- Amazon Simple Notification Service
  https://docs.aws.amazon.com/sns/latest/dg/welcome.html
- Amazon Elastic Container Service
  https://docs.aws.amazon.com/AmazonECS/latest/userguide/Welcome.html
- Amazon EKS
  https://docs.aws.amazon.com/eks/latest/userguide/what-is-eks.html
- AWS CodePipeline
  https://docs.aws.amazon.com/codepipeline/latest/userguide/welcome.html
- AWS CodeCommit
  https://docs.aws.amazon.com/codecommit/latest/userguide/welcome.html

- AWS CodeBuild
  https://docs.aws.amazon.com/codebuild/latest/userguide/welcome.html
- AWS CodeDeploy
  https://docs.aws.amazon.com/codedeploy/latest/userguide/welcome.html
- AWS CodeDeploy から Amazon ECS へのブルー/グリーンデプロイ
  https://docs.aws.amazon.com/AmazonECS/latest/userguide/deployment-type-bluegreen.html
- AWS CodeStar
  https://docs.aws.amazon.com/codestar/latest/userguide/welcome.html
- AWS Cloud9
  https://docs.aws.amazon.com/cloud9/latest/user-guide/welcome.html
- メッセージングサービス
  https://aws.amazon.com/messaging/
- サーバーレスサービス
  https://aws.amazon.com/serverless/

ホワイトペーパー

- AWS のマイクロサービス
  https://d1.awsstatic.com/whitepapers/microservices-on-aws.pdf
- AWS での DevOps 入門
  https://d1.awsstatic.com/whitepapers/AWS_DevOps.pdf
- コードとしてのインフラストラクチャ
  https://d1.awsstatic.com/whitepapers/infrastructure-as-code.pdf
- AWS での継続的インテグレーションと継続的デリバリーの実践
  https://d1.awsstatic.com/whitepapers/DevOps/practicing-continuous-integration-continuous-delivery-on-AWS.pdf

動画

Choosing the Right Messaging Service for Your Distributed App (API305)
https://www.youtube.com/watch?v=4-JmX6MIDDI

改訂履歴

日付	説明
2019 年 10 月	初版発行

ノート

1 Amazon EC2 – https://aws.amazon.com/ec2/

2 マイクロサービス – https://martinfowler.com/articles/microservices.html

3 AWS Lambda – https://aws.amazon.com/lambda/

4 AWS Fargate – https://aws.amazon.com/fargate/

5 Amazon S3 – https://aws.amazon.com/s3/

6 Amazon DynamoDB – https://aws.amazon.com/dynamodb/

7 Amazon Aurora Serverless – https://aws.amazon.com/rds/aurora/serverless/

8 サーバーレス – https://aws.amazon.com/serverless/

9 AWS CloudFormation – https://aws.amazon.com/cloudformation/

10 AWS Serverless Application Model – https://aws.amazon.com/serverless/sam/

11 AWS CDK – https://docs.aws.amazon.com/cdk/latest/guide/what-is.html

12 Amazon API Gateway – https://aws.amazon.com/api-gateway/

13 Envoy Proxy – https://www.envoyproxy.io/

14 Amazon Kinesis – https://aws.amazon.com/kinesis/

15 Amazon Simple Queue Service – https://aws.amazon.com/sqs/

16 Amazon MQ – https://aws.amazon.com/amazon-mq/

17 Amazon MSK – https://aws.amazon.com/msk/

18 Amazon CloudWatch – https://aws.amazon.com/cloudwatch/

19 Fluentd – https://www.fluentd.org/

20 Fluent Bit – https://fluentbit.io/

21 Container Insights の使用 – https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/ContainerInsights.html

22 AWS X-Ray – https://aws.amazon.com/xray/

23 Amazon RDS – https://aws.amazon.com/rds/

24 Amazon RDS – https://aws.amazon.com/rds/

25 AWS CodeStar – https://aws.amazon.com/codestar/

26 AWS CodePipeline – https://aws.amazon.com/codepipeline/

27 AWS CodeCommit https://aws.amazon.com/codecommit/

28 Git – https://git-scm.com/

29 AWS CodeBuild – https://aws.amazon.com/codebuild/

30 AWS CodeDeploy – https://aws.amazon.com/codedeploy/

31 AWS Amplify コンソール – https://aws.amazon.com/amplify/console/

32 GraphQL – https://graphql.org/

33 React – https://reactjs.org/

34 Angular – https://angular.io/

35 Vue – https://vuejs.org/index.html

36 Gatsby – https://www.gatsbyjs.org/

37 GitHub – https://github.com/

38 Amazon CloudFront – https://aws.amazon.com/cloudfront/

39 Amazon Elastic Container Registry – https://aws.amazon.com/ecr/

40 Amazon Elastic Load Balancing – https://aws.amazon.com/elasticloadbalancing/