



マイクロサービスブートキャンプ

DAY1 サーバーレス：座学

三井住友信託銀行様

アマゾンウェブサービスジャパン合同会社
プロフェッショナルサービス本部

DAY2 のタイムテーブル

	DAY1 (1/26)	DAY2 (1/27)
10:00~12:00	オープニング 座学 サーバーレス概要／AWSのサーバーレスサービス	
12:00~13:00		昼休憩
13:00~18:00	ハンズオン① API Gateway + Lambda によるAPI構築 ハンズオン② イベント駆動によるLambda関数起動	ハンズオン① ALB + Fargate によるAPIの構築 ハンズオン② X-Ray による可観測性の向上
18:00~19:00	終わったチームから解散	終わったチームから解散

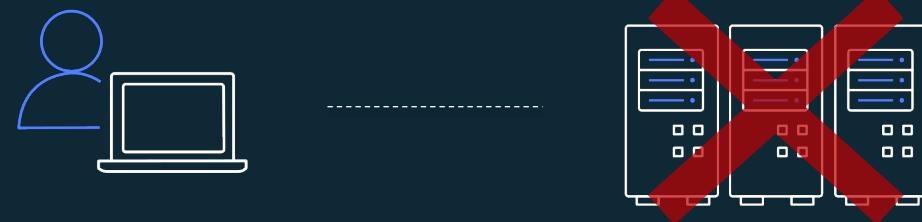
座学：アジェンダ

1. サーバーレス概要
 1. サーバーレスとは
 2. サーバーレスの基礎
 3. サーバーレスのメリット・デメリット
 4. サーバーレスの設計ポイント
2. AWSのサーバーレスサービス
 1. AWS のサーバーレスサービス
 2. AWS でのサーバーレスサービスの種類や特徴
 3. サーバーレスアーキテクチャパターンというか事例というか

サーバーレスの基礎

サーバーレスにまつわる誤解

サーバーレスにはサーバーが存在しない？



サーバーレスとは何か

サーバーレスとは、
サーバーの存在を意識しないアーキテクチャのこと



「サーバーがない」ということではない

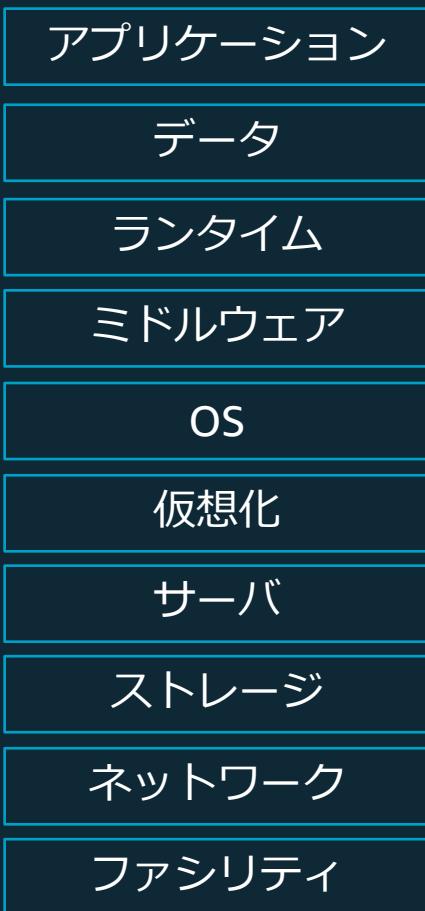
サーバーレスの特徴

サーバーについて検討することなく
スケーラブルで高い可用性を持つアプリケーション
を簡単に構築できる



- ✓ サーバーを管理することなく、コードの実行やデータ管理などの処理を行うことができる
- ✓ オートスケーリング、組み込まれた高可用性、使用量に応じた課金モデルなどの特徴があり、俊敏性の向上とコスト最適化を実現する
- ✓ インフラストラクチャ管理タスクが不要になり、コアなビジネスロジックの作成に集中できる

サーバーレスが注目される背景



自社固有 = ビジネスのコアであり、時間をかけたい

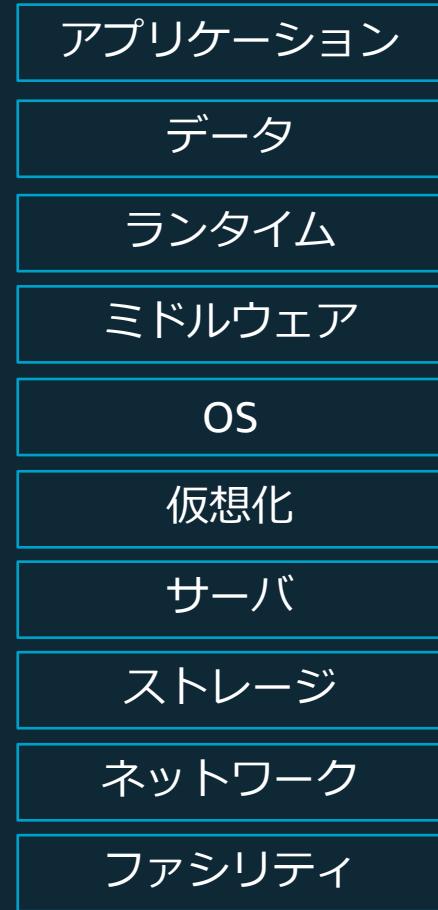
業界類似 ⇌ 差がつきにくいので、時間をかけたくない

ここを代わりに担当してくれる
クラウドとサーバーレス

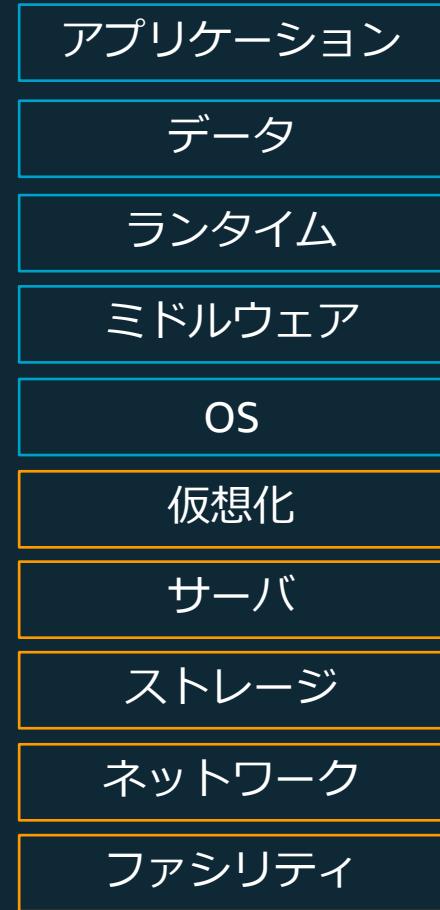
管理が必要な責任範囲の違い

自社担当

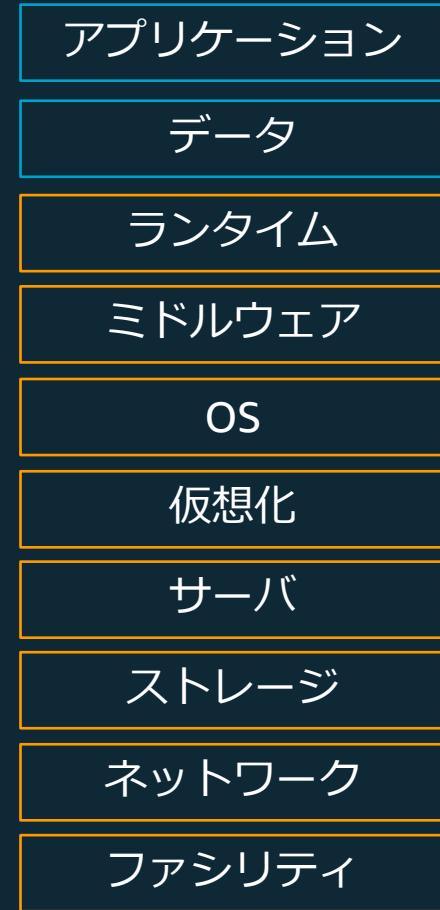
クラウドベンダ担当



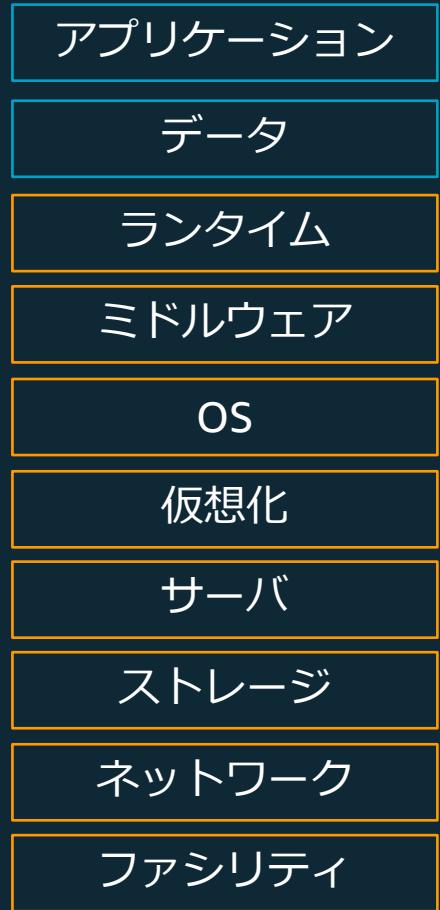
オンプレミス



クラウド (IaaS)



クラウド (PaaS)



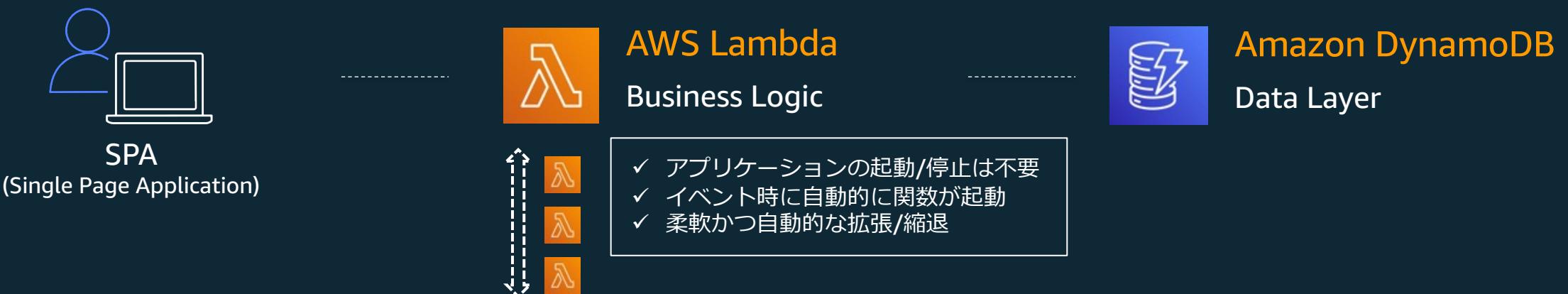
クラウド (サーバーレス)

PaaSとFaaS（サーバーレス）の違い

✓ PaaSシステム



✓ サーバーレスシステム



サーバーレスによってリソースを集中



自社固有 = ビジネスのコアであり、時間をかけたい

自社のリソースを最大限投下して
ビジネスの拡大・差別化を図る

業界類似 ≈ 差がつきにくいので、時間をかけたくない

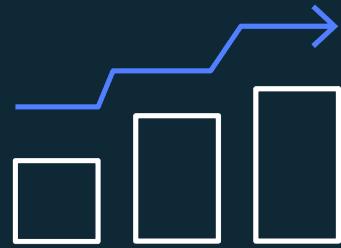
クラウドベンダーに任せて
開發生産性や運用効率を向上

サーバーレスのメリット・デメリット

サーバーレスのメリット



サーバー管理不要



自動でスケール



従量課金



高可用性

参考:今から始めるサーバーレス
<https://aws.amazon.com/jp/serverless/patterns/start-serverless/>

サーバー管理が不要

- サーバー管理に伴う運用作業を省力化できる（以下例）



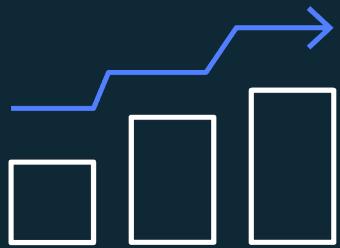
- | | |
|--|---|
| <p><ハードウェア導入></p> <ul style="list-style-type: none">• サーバーの選定・購入• サーバーの搬入 | <p><運用保守></p> <ul style="list-style-type: none">• OSアップデート• セキュリティパッチ対応• インフラ障害対応 |
| <p><サーバー設計/構築></p> <ul style="list-style-type: none">• 各種方式設計• サーバー/NW構築 | |

- 必要になったタイミングですぐに利用を開始できる

期待効果

- ✓ 運用負担を削減し、コア事業に注力
- ✓ 開発・ビジネス拡大スピードの向上

自動でスケール



- 自社サービス/製品がテレビで放送されたり、インターネットで話題になった場合など、急激にトラフィックが増加する場合がある
- サーバを利用する場合にはマシン増設が必要なため、予測できない急激なトラフィックには対応し辛い
- サーバレスは自動的かつ柔軟な拡張/縮退が可能
※ 事前に設定が必要な場合もあります

期待効果

- ✓ エンドユーザーの利便性向上
- ✓ ビジネス機会損失の防止
- ✓ スケーラビリティ設計の省力化

従量課金



- ・ オンプレミスの場合、サーバー購入の初期コストに加え、継続的なサーバ運用費用が必要となる
- ・ クラウドでもサーバーがある場合、使われていなくても立ち上がっている時間だけ課金される
- ・ サーバレスの場合、使ったタイミングだけ課金される。使っていないときには課金されず、初期コストも必要ない

期待効果

- ✓ サーバー導入初期費用の削減
- ✓ アプリケーション運用費用の最適化
- ✓ 投資対効果の向上

高可用性



- ・ オンプレミスやクラウド（サーバーあり）の場合、高い可用性を保つためにデータのレプリケーションや冗長設計を自身で実施する必要がある
- ・ サーバーレスの場合、レプリケーションや冗長設計が組み込まれているため、利用者が設計開発することがなく高い可用性を実現

期待効果

- ✓ エンドユーザーの利便性向上
- ✓ ビジネス機会損失の防止
- ✓ 高可用性設計の省力化

サーバーレスによるお客様のビジネス効果

5x

Agility

機能開発・デプロイの時間が
加速化し、競争力に貢献

409%

ROI (投資対効果)

5年間で計算した投資対効果

1-3ヶ月

Productivity

スケール、冗長化などの考慮不要で
短期実装が可能

33%

Productivity

アプリケーション開発の生産性が
向上

“ゼロ”

Low Maintenance

1年運用で保守作業がほぼゼロ
ほぼ人手を割かずに運用

9:1

Low Maintenance

“開発：保守運用”的作業比率が
1:9から大きく改善

参考 : サーバーレスのビジネス効果とは

<https://aws.amazon.com/jp/serverless/patterns/serverless-benefit/>

サーバーレスのデメリット

自由度とのトレードオフ＝制約を受け入れる



サーバーレスの前提/制約

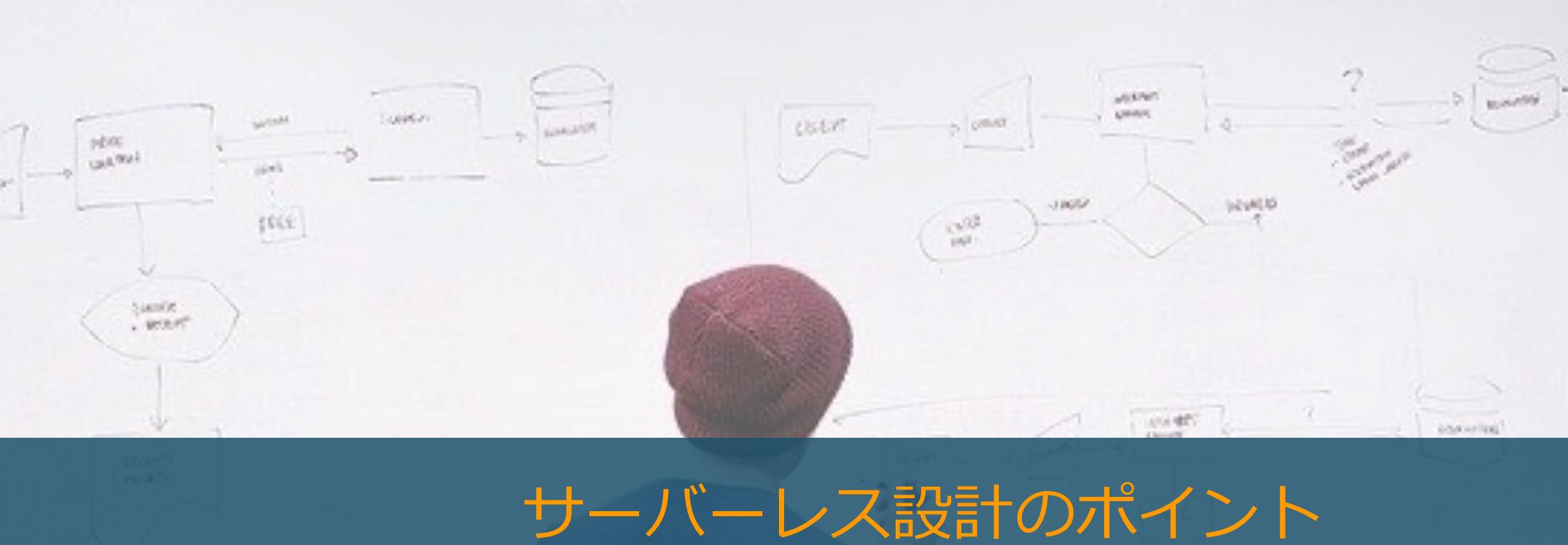


考慮事項

- 既存のアプリケーションやパッケージ製品をリホスト (Lift & Shift) で移行するのが困難
- サービス固有の制約が存在する (例えば、Lambdaの場合、最大処理時間15分 等のHard Limit)

許容できない制約やマッチしない要件がある場合、
サーバーレスが最適な選択肢にならないことがある

- (例)
- 処理時間を数時間要するバッチ処理
 - レガシーなオンプレミス・システムの移行
 - 複雑な業務ロジックや高速な計算処理を実行するアプリケーション



サーバーレス設計のポイント

サーバーレス設計のポイント



ビルディングブロック

サービスを組み合わせる



イベント駆動

イベントでサービス間を連携する



ステートレス

状態を持たない

ビルディングブロック

従来のシステム



サーバーレスシステム



SPA
(Single Page Application)



AWS Lambda



Amazon
DynamoDB

実現したい機能を担うサービスを組み合わせてシステムを構築する
(ブロックの組み合わせ)

ビルディングブロック

デプロイまでの自動化を実現したい（CI/CD）



AWS Cloud9



AWS CodeBuild



AWS CodeCommit



AWS CodeDeploy

認証機能を設けて
セキュリティを強化したい 特定ユーザのみアクセスさせたい



SPA
(Single Page Application)



AWS WAF



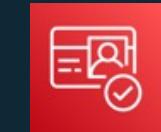
Amazon API Gateway



AWS Lambda



Amazon DynamoDB



Amazon Cognito



Amazon CloudWatch



Amazon SNS

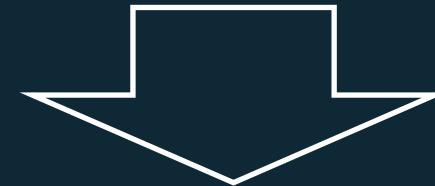


Amazon Connect

イベント駆動



ビルディングブロックで組み合わせたサービス同士を
どう連携させるのか？



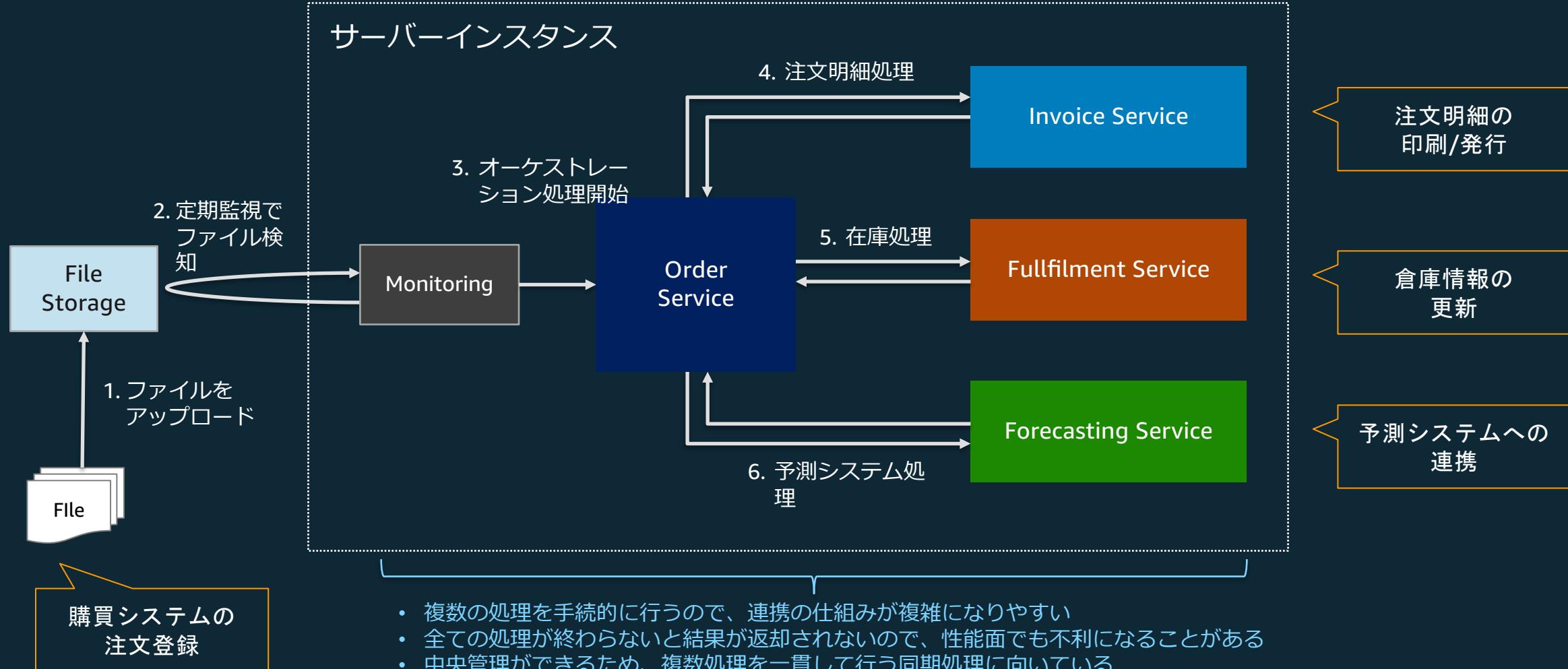
イベント駆動
イベント（システム状態の変化）をサービス間で共有する

イベント駆動

イベント駆動とはどのようなものか？
具体例を基にシステム構成を考える

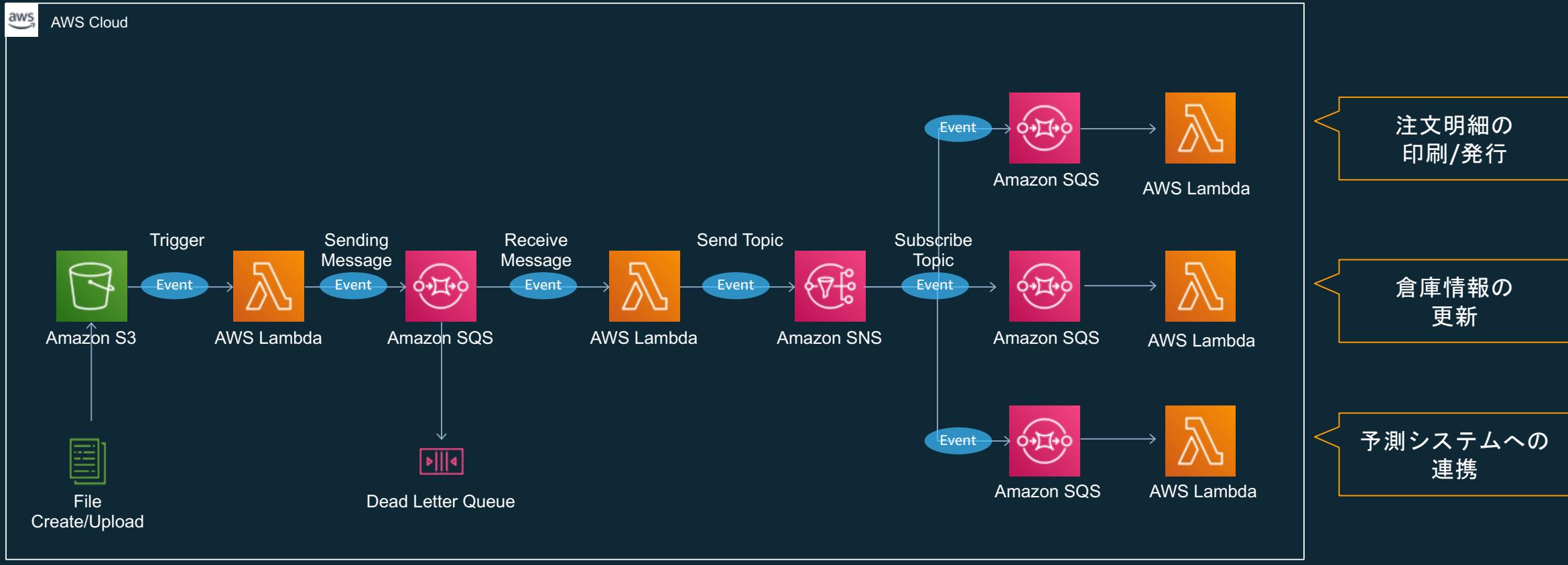
イベント駆動（ケーススタディ） 購買システムの注文登録処理

非イベント駆動の場合：中央管理による複雑な同期処理に向いており、サービス結合度は強い



イベント駆動（ケーススタディ） 購買システムの注文登録処理

イベント駆動の場合：イベントによって非同期でシンプルに連携、サービスが疎結合になる



購買システムの
注文登録

- ・ イベント伝搬で処理を非同期でつないでいいけるので、連携の仕組みがシンプルになり性能面でも有利
- ・ 各サービスの結合度が下がり、それぞれのサービスの開発や運用が効率的になる
- ・ サービス間で同期を取ったり、複数処理を一貫して行うような仕様には不向き

イベント駆動 特性・メリット

イベント駆動にすると何が良いのか？ → 跛結合 / 性能 / 拡張性

- **疎結合化による独立性**

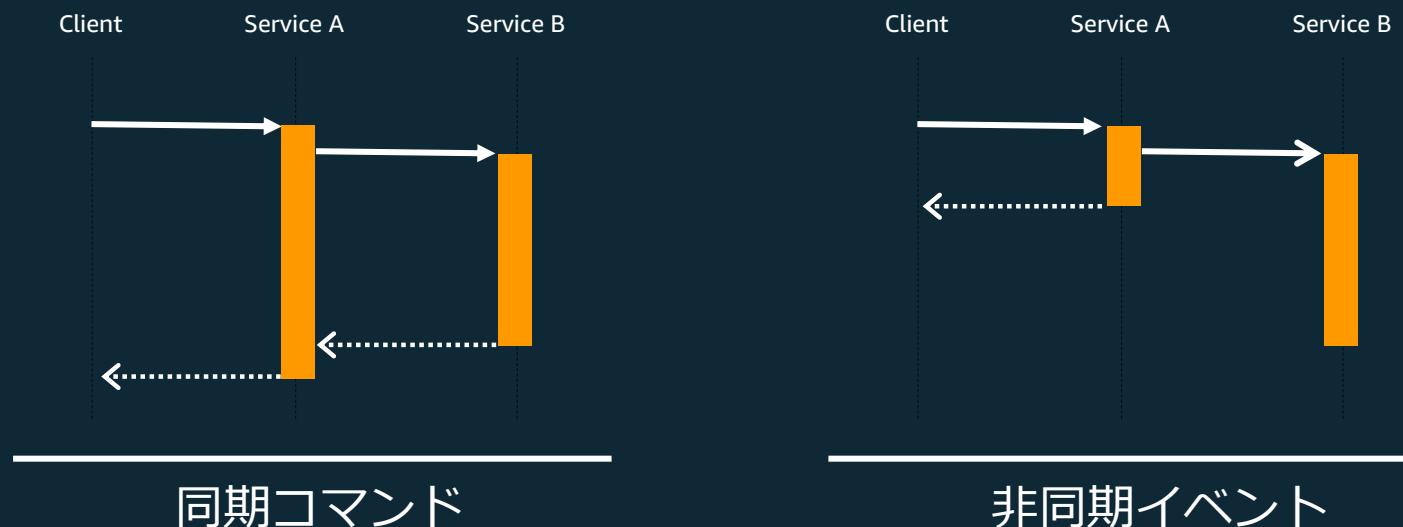
前後のサービス処理内容を気にせず、処理結果としてのイベント（通知）のみを
関心事にサービスやビジネスロジックを独立させることができる。
その結果、他のサービス依存せずサービス個別の開発やスケールが実現する。



イベント駆動 特性・メリット

イベント駆動にすると何が良いのか？ → 跛結合 / 性能 / 拡張性

- 性能：応答速度の向上
処理を非同期化することで、処理の完了を待つことなく素早く応答を返すことができる。



イベント駆動 特性・メリット

イベント駆動にすると何が良いのか？ → 跛結合 / 性能 / 拡張性

- 性能：サービス可用性の向上
キューを利用して、メッセージ（リクエスト）バッファリングの仕組みを作ることができる。
その結果、大量リクエスト発生時でもエラーを返さずに安全な流量制御ができる。

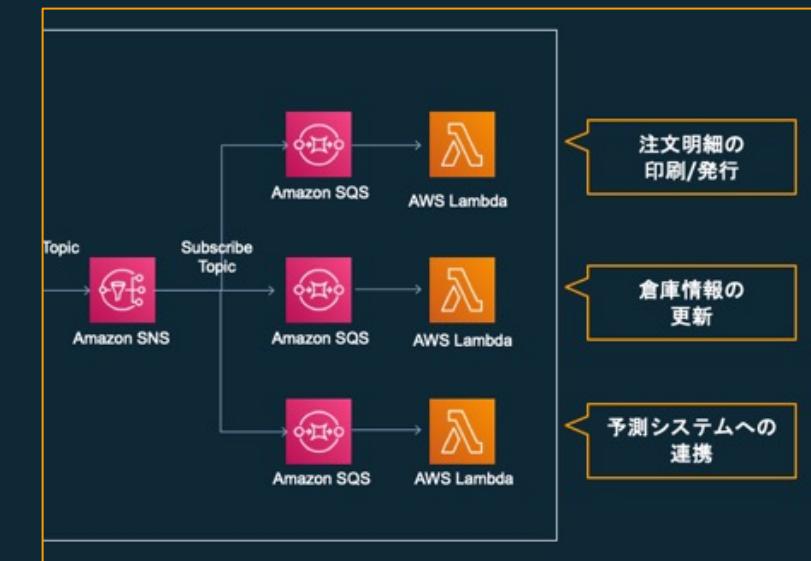
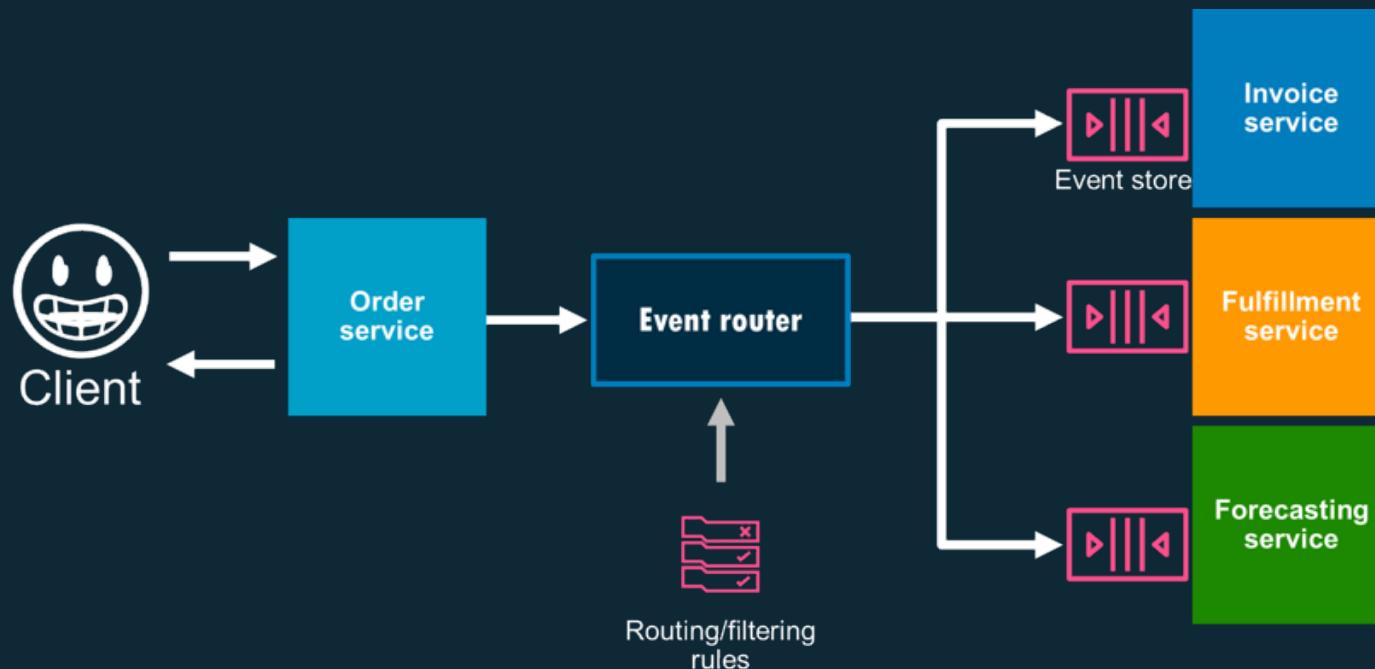


サービスが処理するまでメッセージ
をバッファリング

イベント駆動 特性・メリット

イベント駆動にすると何が良いのか？ → 跛結合 / 性能 / 拡張性

- 拡張性
連携先が増減した場合に、ハブとなるサービスの修正で拡張/縮小が可能



ステートレス

ステートレスとは？

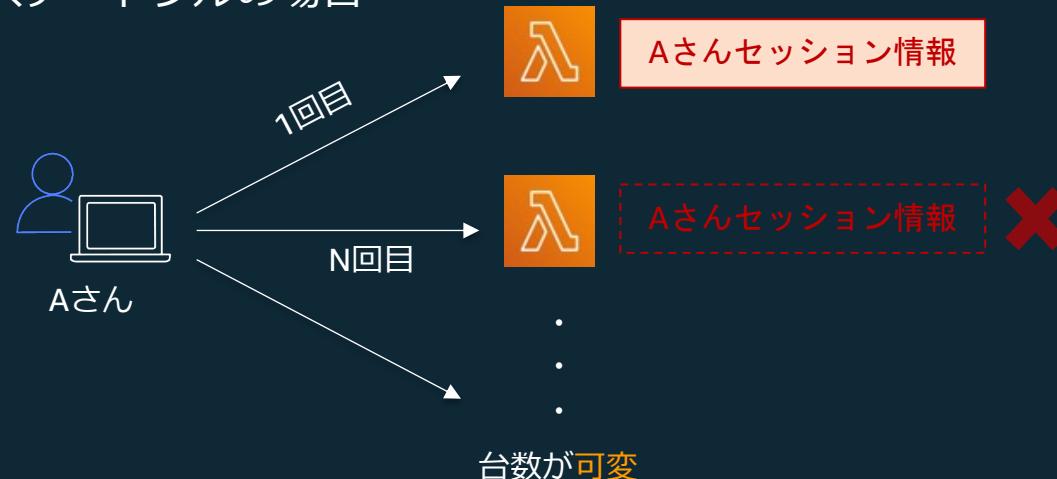
- アプリケーションが状態を保持しない性質のこと。「状態」の例として、セッション情報やログイン認証後にDBから取得した情報などがある。
- 「処理をするサービス」と「データを持つサービス」で責務を分離する。このとき、アプリケーションは「処理をするサービス」側になる
- 処理に「状態」が必要な場合には、「データを持つサービス」から取得/格納する。(データキャッシュやデータベースなど、データを保持するためのサービス)
- ステートレスにすることで、サーバーレスの自動スケールを生かしながら高い可用性を維持できる

ステートレス

なぜステートレスである必要があるのか

- サーバーレスのメリットは「自動でスケール」 = 負荷に応じて増えたり減ったりする
- 例えばリクエストに応じて処理する場合、1回目のリクエストと2回目のリクエストが同じ場所に割り振られるとは限らない（状態を再利用できない）
※同じ場所に割り振られる仕組みにもできるが、負荷の偏りが発生するため適切な処理分散の観点で問題がある。

ステートフルの場合



ステートレスの場合



まとめ：サーバーレスの基礎知識

サーバーレスとは

✓ サーバーレスとは、
サーバーの存在を意識
しないアーキテクチャ

✓ サーバー管理を任せて自社固
有の領域に集中できる

✓ PaaS / FaaSの違い

- FaaSはイベント時に稼働
- PaaSはスケールの条件が
必要
- FaaSは自動でスケール

✓ サーバーレスを使って様々な
ユースケースに対応可能

メリット・デメリット

✓ サーバーレスのメリット

- サーバー管理不要
- 自動でスケール
- 従量課金モデル
- 高可用性



✓ サーバーレスのデメリット

- 基盤の自由度が低い
- サービス固有の制約

• 全てをサーバーレスで実装
しようとするのではなく、
要件と制約を考慮して選択す
る

設計のポイント

✓ ビルディングブロック
サービスを複数組み合わせてシ
ステムを構成する考え方

✓ イベント駆動
システムをイベントで非同期連
携する考え方。疎結合 / 性能 /
拡張性 の特性を得られる

✓ ステートレス
スケーラビリティを維持するた
め、状態を持たない設計にする



ビルディングブロック
サービスを組み合わせる



イベント駆動
イベントでサービス間を連携する



ステートレス
状態を持たない

A blurred photograph showing the silhouettes of several people's hands raised in the air, likely during a question-and-answer session at a conference or event. The hands are against a bright, overexposed sky.

Q&A

A photograph of a modern, multi-story office building with a glass facade. The Amazon logo, featuring the word "amazon" in lowercase with a yellow smiley arrow underneath, is prominently displayed on the side of the building. The sky is clear and blue.

AWSでのサーバーレスサービスの種類や特徴

主なAWSのサーバーレスサービス

COMPUTE



AWS
Lambda



AWS
Fargate

DATA STORES



Amazon
S3



Amazon Aurora
Serverless



Amazon
DynamoDB

INTEGRATION



Amazon
EventBridge



Amazon
API Gateway



Amazon
SQS



Amazon
SNS



AWS
Step Functions



AWS
AppSync

Compute



AWS Lambda

- コードを圧縮してアップロードするだけで実行可能
- 200以上のAWSサービスやSaaSをトリガーにしたイベント駆動



AWS Fargate

- コンテナワークロードの実行

Data Stores



Amazon S3

- 高い可用性と耐久性によるストレージ
- パフォーマンスやコストなど特性に応じたストレージクラスを選択



Amazon Aurora
Serverless

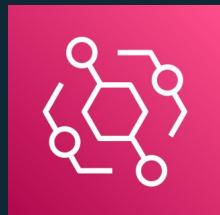
- データベースリソースの管理が不要



Amazon DynamoDB

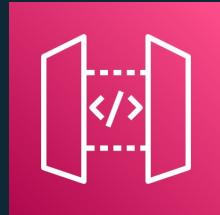
- NoSQL
- SQL互換であるPartiQLを使用して検索、挿入、更新、削除が可能

Integration



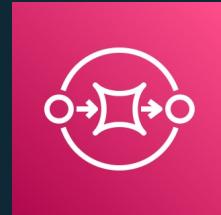
Amazon EventBridge

- AWS上のサービス同士を接続
- SaaSとAWSのサービスを接続



Amazon API Gateway

- APIを管理
- トラフィックの管理、承認、
アクセス制御、監視を実施



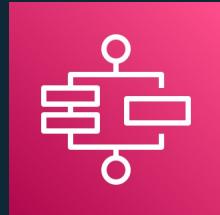
Amazon SQS

- メッセージキューイングサービス
- マネコン、AWS CLI、SDKなど
からキューの送受信が可能



Amazon SNS

- メッセージングサービス
- Kinesis Data Firehose、SQS
、Lambda、HTTP、メー
ル、モバイルプッシュ通
知、SMSなどに配信



AWS Step Functions

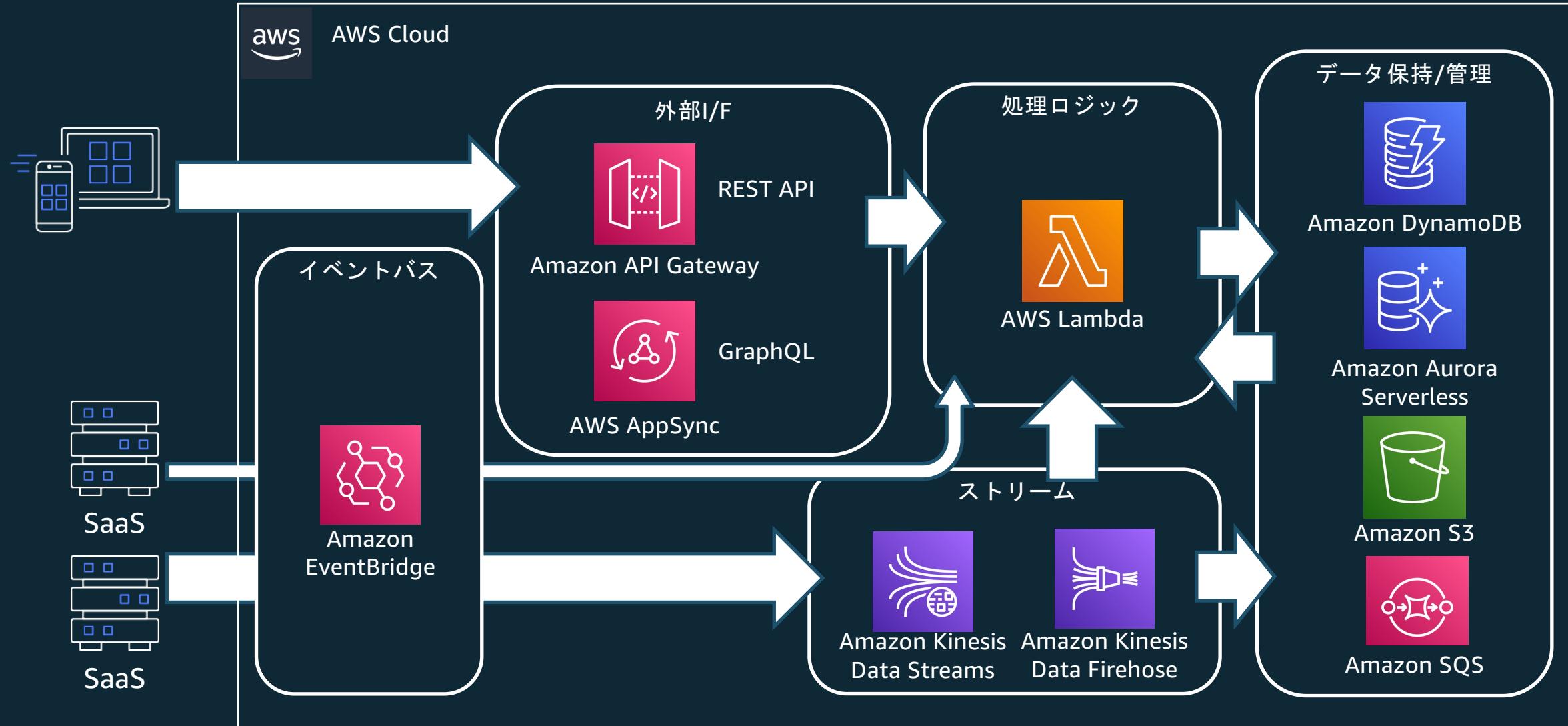
- 視覚的なワークフロー
- ワークフロースタジオによ
るビジュアルツールで構築
可能



AWS AppSync

- GraphQLによるAPI
- キャッシュ機能による
低レイテンシー

サーバーレスアーキテクチャパターン



A photograph of two scuba divers swimming in clear blue water. One diver is in the foreground, facing away from the camera towards the horizon. Another diver is slightly behind and to the right. Sunlight filters down from the surface, creating bright highlights on the divers and the sandy ocean floor. The background shows distant coral reefs.

Dive Deep into the Services





AWS Lambda

© 2023, Amazon Web Services, Inc. or its Affiliates. All rights reserved.

AWS Lambda とは



サーバーの管理をすることなくコードを実行するサーバーレス・コンピューティングサービス

- ✓ インフラストラクチャのプロビジョニングや管理をすることなくコードを実行
- ✓ AWS Lambda で他のサービス間をつなぎ、イベント駆動型アプリケーションを作成
- ✓ 数十イベント/日 から数十万イベント/秒 まであらゆる規模のリクエストに自動的に対応
- ✓ 使用するコンピューティング時間に対してのみミリ秒単位で支払うことで、コストを削減



AWS Lambdaの基礎

サポートするランタイム

言語	バージョン	備考
Node.js	12.x、14.x、16.x、18.x	Node.js12.xは2023年3月31日まで
Python	3.7、3.8、3.9	
Ruby	2.7	
Java	8、11	JDK: amazon-corretto
Go	1.x	
.NET Core	5、6、3.1	.NET Core 3.1は2023年3月31日まで

※ バージョンは2023/1 時点

※ 上記以外に、カスタムランタイムで任意のランタイムを使用可能

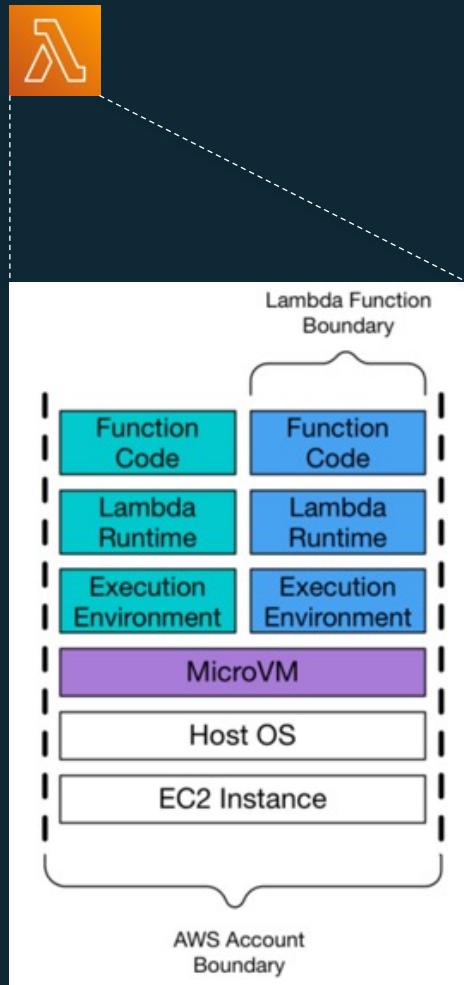
AWS Lambdaの制約

参考 : Lambda クオータ

https://docs.aws.amazon.com/ja_jp/lambda/latest/dg/gettingstarted-limits.html

リソース	デフォルトの制限	上限緩和
同時実行数	1,000(東京リージョン)	可
関数とレイヤーストレージ	75GB	
関数のメモリ割り当て	128MB から 10,240MB (1 MB単位)	不可
関数タイムアウト	900秒(15分)	
関数の環境変数(キーとバリューの合計)	4KB(AWS Lambda予約の変更不可な環境変数以外)	
関数リソースベースポリシー	20KB	
関数レイヤー	5 layers	
呼び出しペイロード	6MB(同期)/256KB(非同期)	
デプロイパッケージサイズ	50MB (zip圧縮済み、直接アップロード) 250MB (解凍後) 3MB (コンソールエディタ)	
コンテナイメージのコードパッケージサイズ	10GB	
テストイベント(コンソールエディタ)	10	
/tmpディレクトリのストレージ	512 MB から 10,240 MB (1 MB単位)	
ファイルディスクリプタ	1,024	
実行プロセス/スレッド	1,024	

AWS Lambdaの動き方



- EC2インスタンス（つまりサーバ）の上で稼働している
- その上で、Lambdaに必要な実行環境がある（ユーザは意識しなくて良い部分）
- その上に、ユーザが開発したコードが稼働する領域がある

<https://www.amazon.science/publications/firecracker-lightweight-virtualization-for-serverless-applications>

AWS Lambdaのライフサイクル



- コンテナ生成
- S3からのZIPダウンロード
- ZIPファイルの展開

AWS Lambdaのライフサイクル



- 各ランタイムの初期化処理
- グローバルスコープ処理

AWS Lambdaのライフサイクル



- ハンドラーで指定した
関数/メソッドの実行

AWS Lambdaのライフサイクル



AWS Lambdaのコールドスタート

1から6全て実行するのがコールドスタート
(コールドスタートはDurationには含まれない)

Lambdaのライフサイクル

1. コンテナ生成
2. パッケージロード
3. パッケージ展開
4. ランタイム起動・初期化
5. 関数/メソッドの実行
6. コンテナの破棄

基本的に
同じ処理を実行



再利用できれば
省略可能
(ウォームスタート)

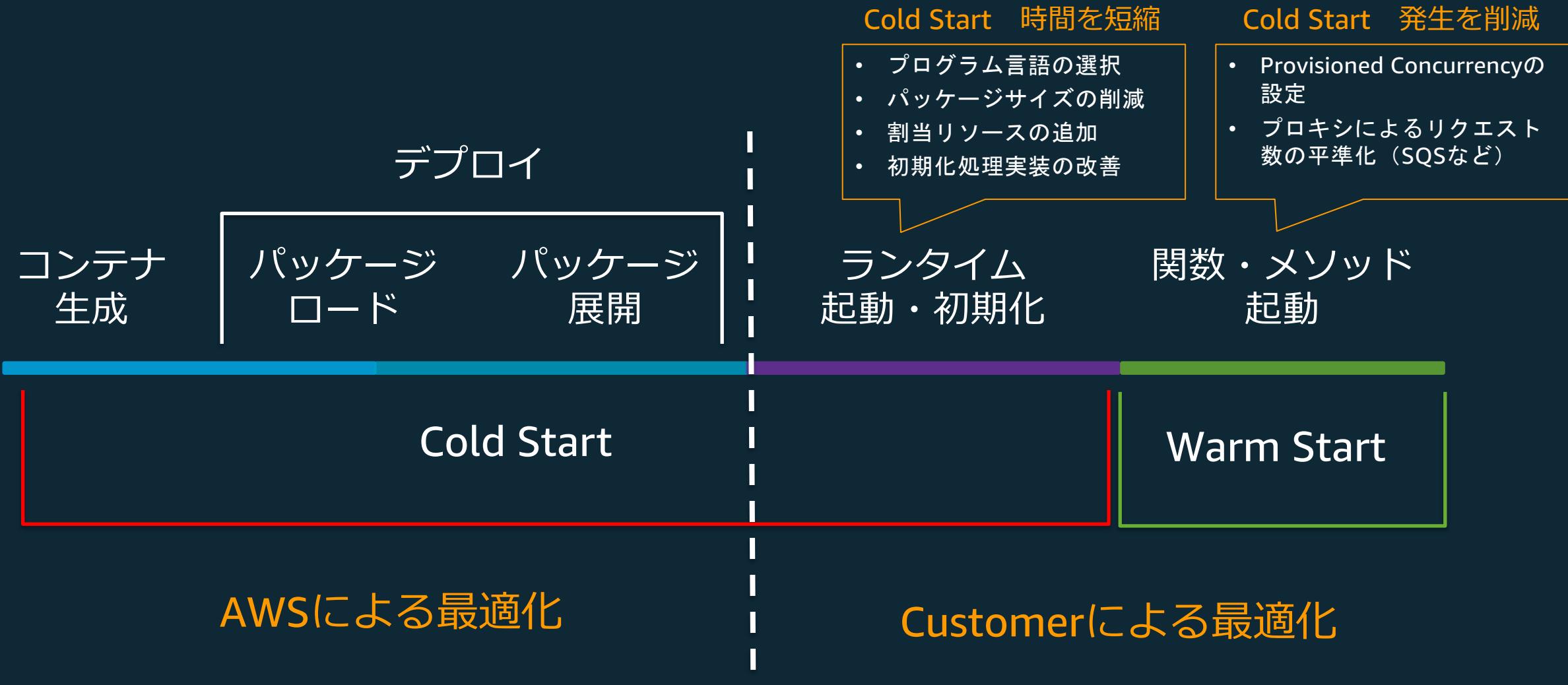
コールドスタートが起こる条件

- コンテナが1つもない（初回起動）
- 同時実行数の増加
- Lambdaのコード、設定変更
- コールドスタートを0にすることは難しい

コールドスタートが起こらないケース

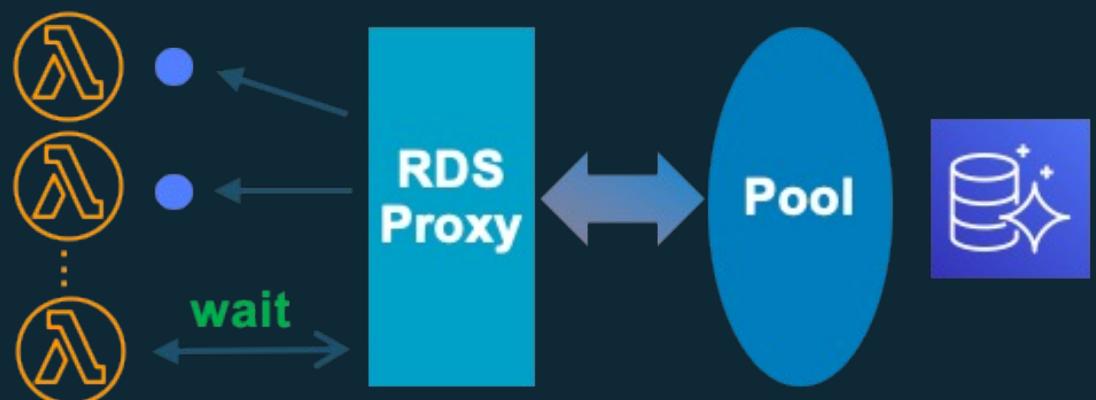
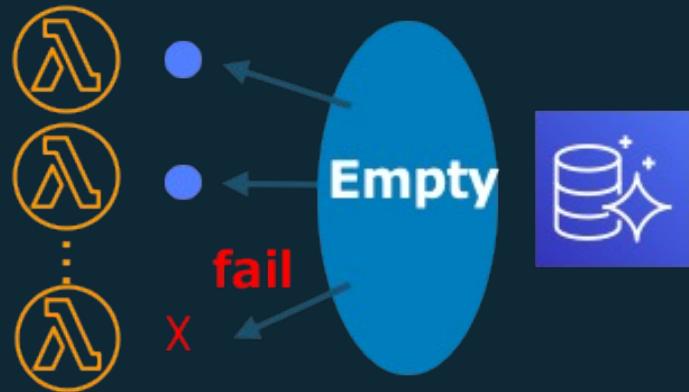
- 安定的なリクエスト数
- Provisioned Concurrencyの設定

AWS Lambdaのライフサイクル



Amazon RDS Proxy

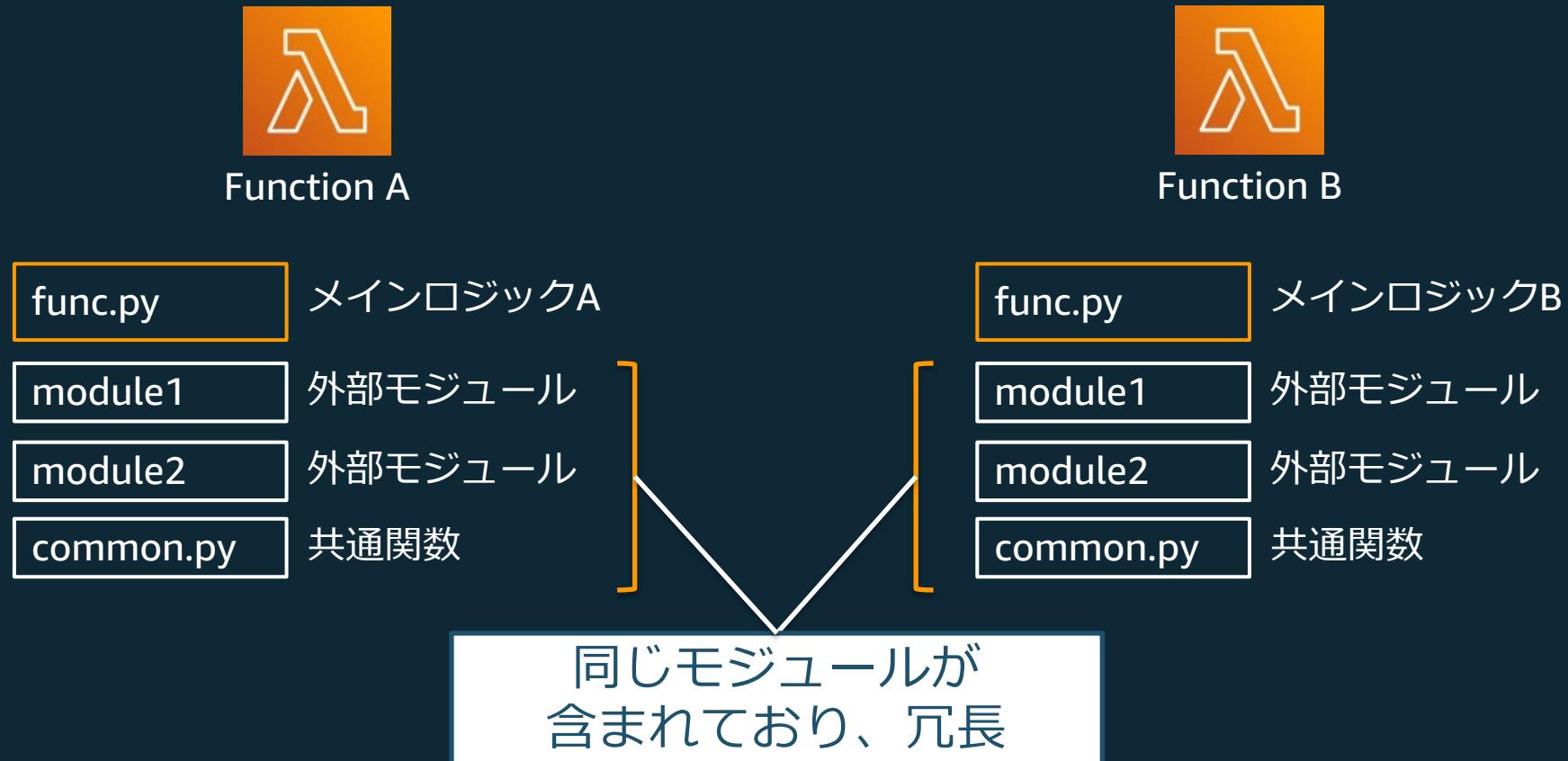
アンチパターンであったLambda + RDBMSの要因の1つであったコネクション数の問題に対応
従来のコネクション RDS Proxyを利用したDBのコネクション



- 接続プーリング
 - 接続の開閉に伴うデータベースの負荷 (TLS/SSL のハンドシェイク、認証、ネゴシエーション機能などのCPU負荷など) を削減
- 接続の多重化
 - 接続の再利用により、データベース接続に必要なコンピューティングリソース (主にメモリ) を削減
 - `max_connections` エラーの発生頻度の抑制

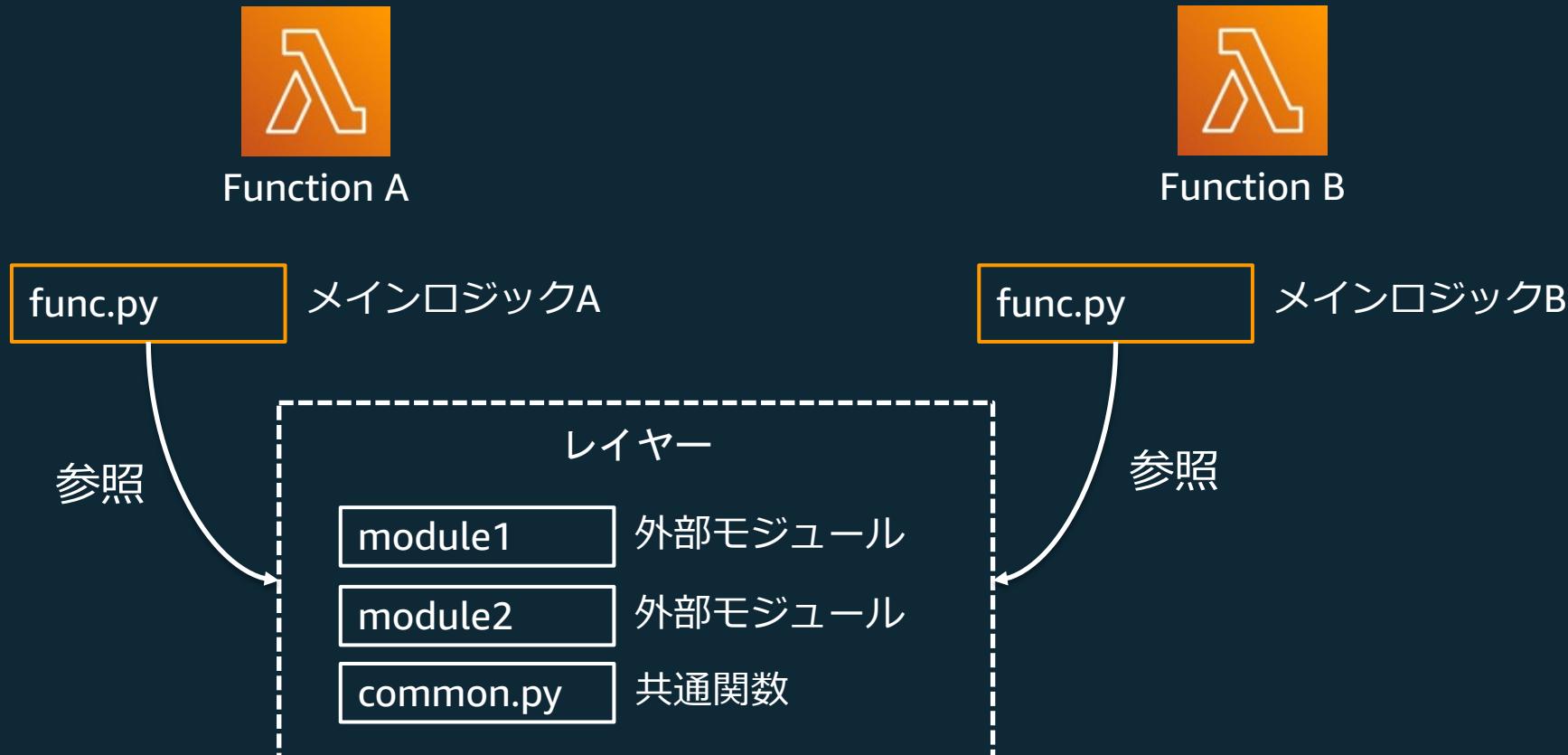
AWS Lambdaとレイヤー

各Lambda関数に必要なライブラリを含めてしまう例



AWS Lambdaとレイヤー

各Lambda関数に必要なライブラリを含めてしまう例



AWS Lambda バージョニングとエイリアス

バージョニングとエイリアスを用いてカナリアリリースのような制御が可能

バージョニング

- ある時点の関数をバージョンとして管理
- 各バージョンには一意のARN
- 一度発行すると編集不可
- 環境変数やロールなども含まれる

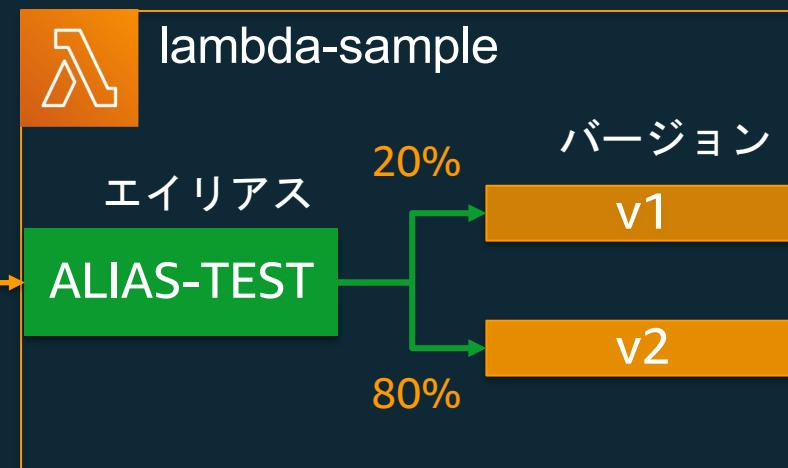
エイリアス

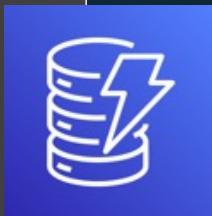
- 特定バージョンのポインタのようなもの
- いつでも付け替え可能

(例)
バージョン・エイリアスによるリク
エスト制御



Kinesis Data Stream





Amazon DynamoDB

Amazon DynamoDB とは

規模に応じたパフォーマンスを実現する高速で柔軟な NoSQL データベースサービス



- ✓ 1桁ミリ秒単位の安定したパフォーマンス
- ✓ ほぼ無制限のスループットとストレージ、自動マルチリージョンレプリケーション、自動バックアップと復元
- ✓ 最大可用性 99.999% のSLA により保証された信頼性でデータを保護
- ✓ AWS のサービスとの統合によりデータ利活用が可能、インサイトの抽出やトラフィックの傾向をモニタリング

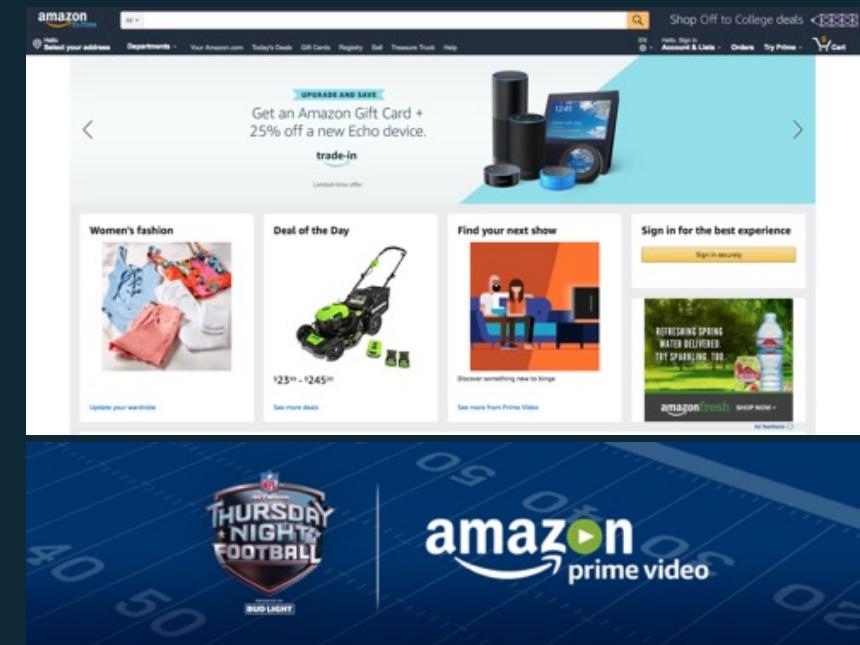
Amazon DynamoDBの基礎

Amazon.comで何かを購入するたびに内部では
多くのシステムが稼働し、日々何億ものアクティブ
なワークフローを処理している

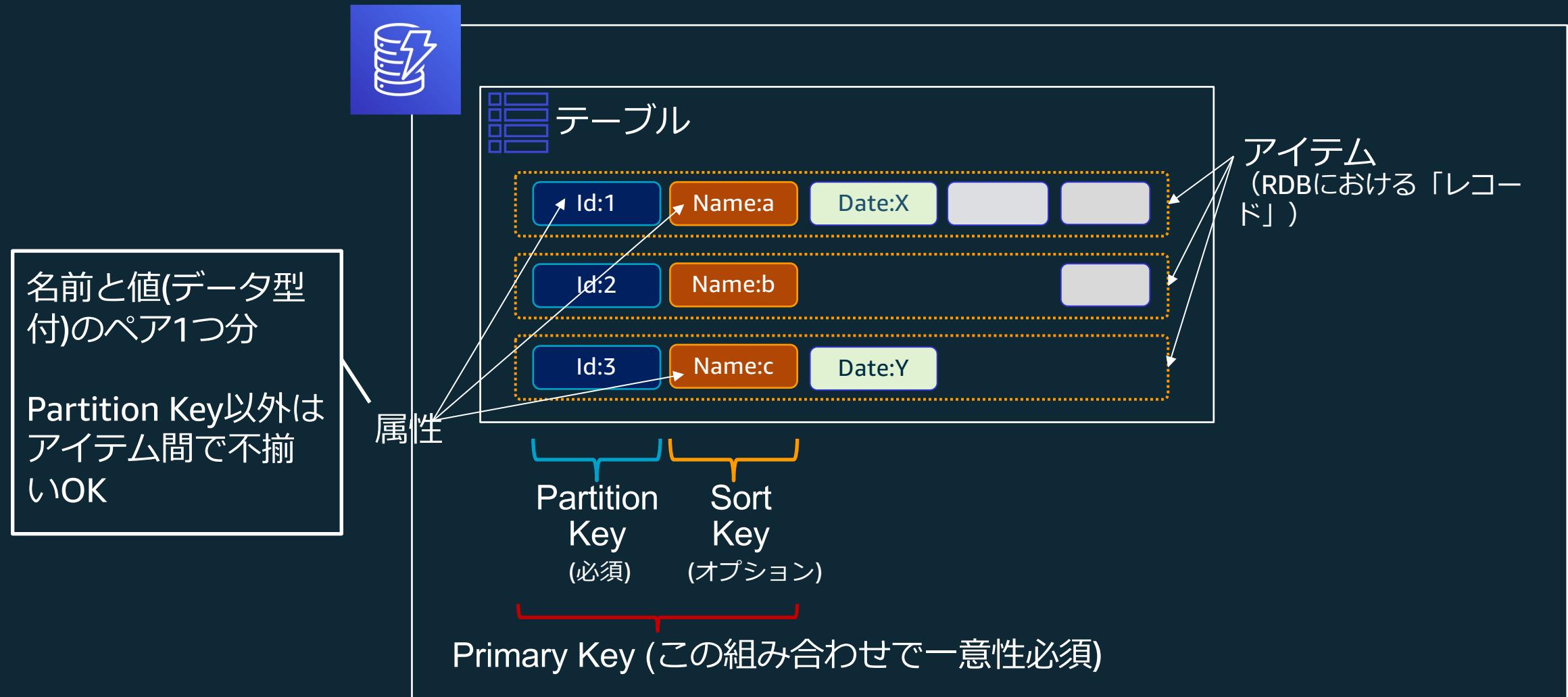


OracleからDynamoDBへ

- ワークフローの処理時間が1秒程度から100 msまで改善
- スケーリングとメンテナンスのオペレーションコストが1/10に減少
- 300ものOracle DBホストを削減



Amazon DynamoDBの主要構造



Amazon DynamoDBの基礎

制約

項目	制限	備考
アカウントあたりの テーブル数	2,500	リージョンごと 引き上げ可能
テーブル名、 セカンダリインデックス名	3 文字以上、255 文字以下 英数字 _(アンダースコア)、 - (ハイフン) . (ピリオド)	
アイテムの合計サイズ	400KB	

※ 記載内容は2023/1時点

参考 : DynamoDB サービスクオータ

https://docs.aws.amazon.com/ja_jp/general/latest/gr/ddb.htm

https://docs.aws.amazon.com/ja_jp/amazondynamodb/latest/developerguide/ServiceQuotas.html

AWSでのDBサービス



Relational

Key-value

Document

In-memory

Graph

Time-series

Ledger

特徴

参照整合性、
ACIDトランザクション、
Schema-On-Write

高スループット、低レイテンシーの読み取り、書込み、無限のスケーラル

ドキュメントを保存し、任意の属性にクエリーですばやくアクセス

マイクロ秒のレイテンシーでキーによるクエリ

すばやく簡単にデータ間の関係を作成しナビゲート

データを時間順に収集、格納、処理

完全で不変で検証可能なアプリケーションデータに対するすべての変更履歴

ユースケース

リフト&シフト、
ERP、CRM、金融

リアルタイム入札、ショッピングカート、ソーシャル、製品タログ、顧客の好み

コンテンツ管理、パーソナライゼーション、モバイル

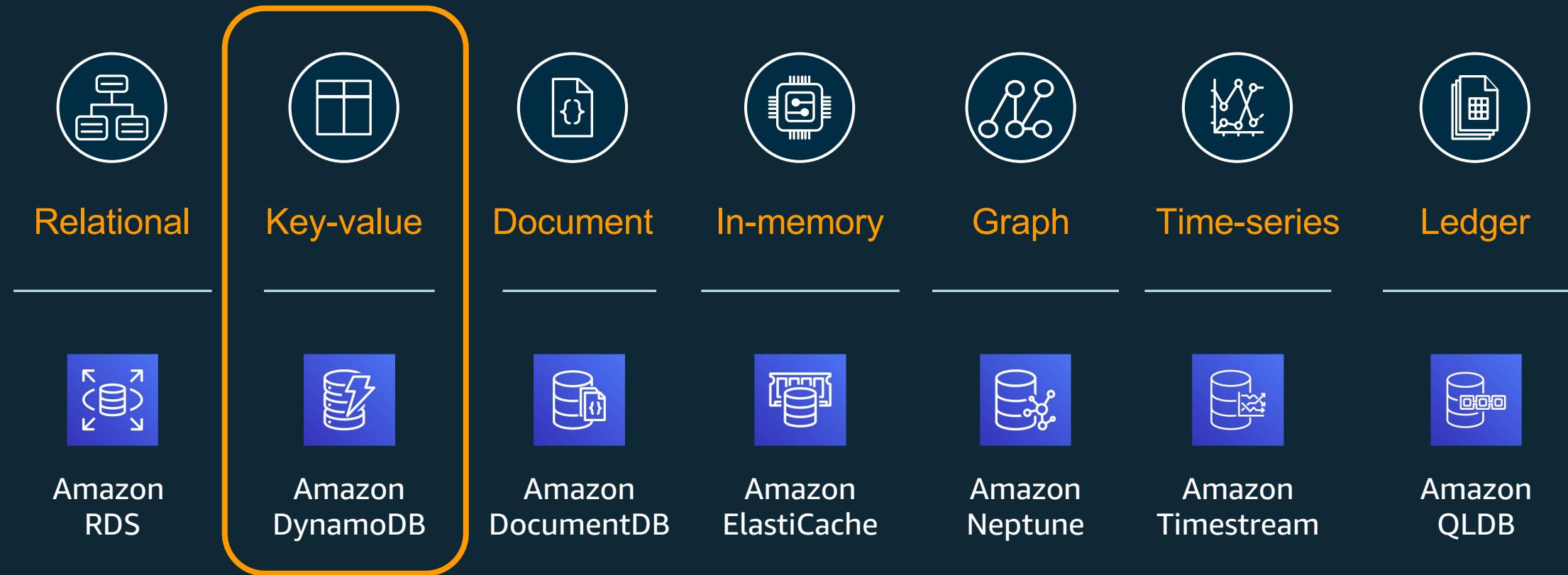
リーダーボード、リアルタイム分析、キャッシング

不正検出、ソーシャルネットワーキング、レコメンドエンジン

IoTアプリケーション、イベントトラッキング

SoR(System of Record)、サプライチェーン、ヘルスケア、届出、財務

AWSでのDBサービス



RelationalとKey-valueの使い分け

	Relational(RDS)	Key-value (DynamoDB)
処理速度	低速 (Key-valueと比較して)	高速
データ構造	正規化/リレーションナル	非正規化/階層構造
データ操作方法	SQL (複雑な操作に強い)	データベースに依存 (複雑な操作は苦手)
トランザクション	あり	限定的
データの一貫性	強い一貫性(ACID)	データベースに依存
スケーラビリティ	限定的	高い



Amazon API Gateway

Amazon API Gateway とは



規模に関わらず簡単にAPIを作成、公開、保守、モニタリング、保護できる API管理サービス



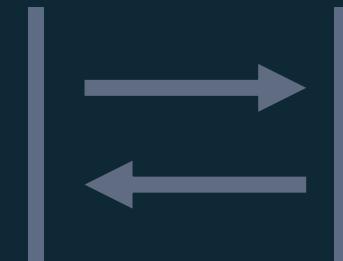
- ✓ スピーディーに開発、テスト、リリースにより、効率的なAPI開発を行うことが可能
- ✓ ダッシュボードから視覚的にパフォーマンスマトリクスとエラー情報のモニタリングが可能
- ✓ IAM や Cognito と組み合わせて、APIへのアクセス制御が簡単に実現できる
- ✓ APIの使用分のみが課金されるため、使用していない間の料金を抑えてコストの最適化を図ることができる

「API」とは何か

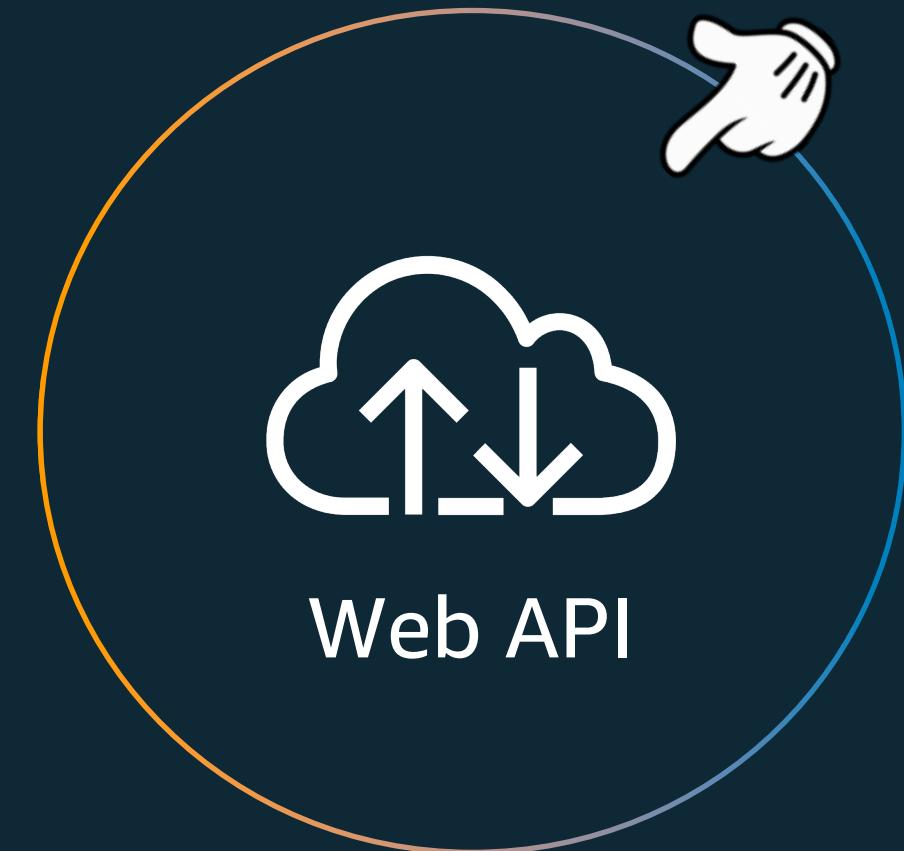
“Application Programming Interface”：
プログラムやソフトウェア同士がやり取りするための取り決め・仕様



言語処理系API
[SDK/ライブラリ等]



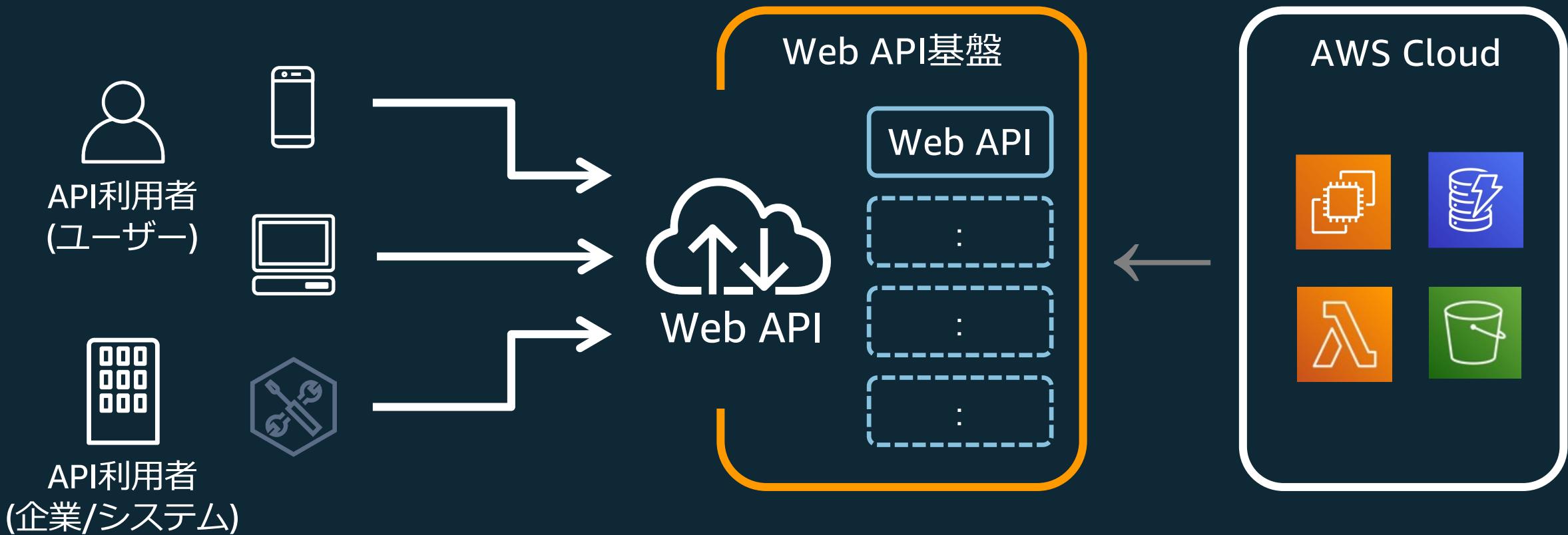
ネットワーク
呼び出しAPI



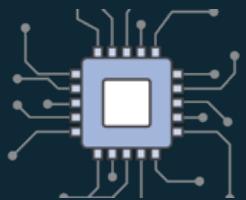
Web API

「Web API」によるシステムの構築

パブリックなWeb API、企業内のプライベートなWeb API基盤などで
Web API連携を活用したエコシステムが発展してきた



Web API提供時の共通課題



1. インフラの管理
(可用性とスケーラビリティ)



2. APIの管理
(設定やデプロイの制御)

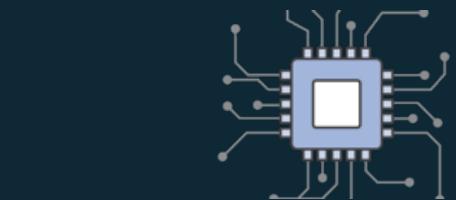
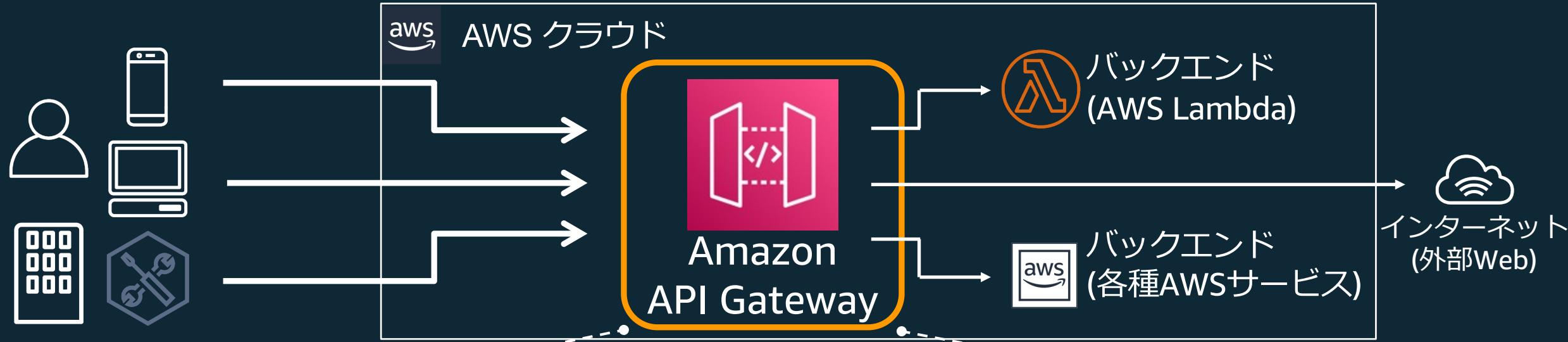


3. 認証と認可
(アクセスの制御)



4. 流量制御と保護
(スロットリング)

Amazon API Gateway の位置付け



1. インフラの管理
(可用性とスケーラビリティ)



2. APIの管理
(設定やデプロイの制御)

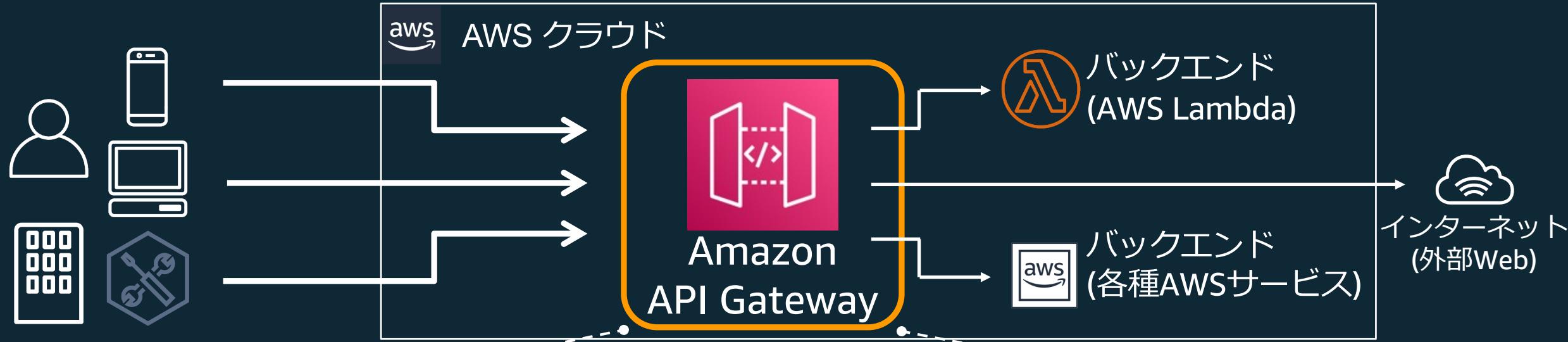


3. 認証と認可
(アクセスの制御)



4. 流量制御と保護
(スロットリング)

Amazon API Gateway の位置付け



Amazon API Gateway は
フルマネージド型のサービス

自動でスケール

仮想サーバー(OS)管理不要

使用量に応じた課金

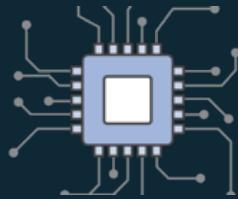
1. インフラの管理
(可用性とスケーラビリティ)

2. APIの管理
(設定やプロトコルの制御)

3. 認証と認可
(アクセスの制御)

4. 流量制御と保護
(スロットリング)

Amazon API Gateway の特徴



1. インフラの管理 (可用性とスケーラビリティ)



- マネージドサービスとしてインフラをAWSが管理
- サーバーレスアーキテクチャ実現に適した基盤



2. APIの管理 (設定やデプロイの制御)



- ステージや統合リクエスト/レスポンスなどの設定
- カナリアリリースも設定で実現



3. 認証と認可 (アクセスの制御)



- オーソライザーや AWS署名v4を使った認証と認可
- リソースポリシーでの細かなアクセス制御



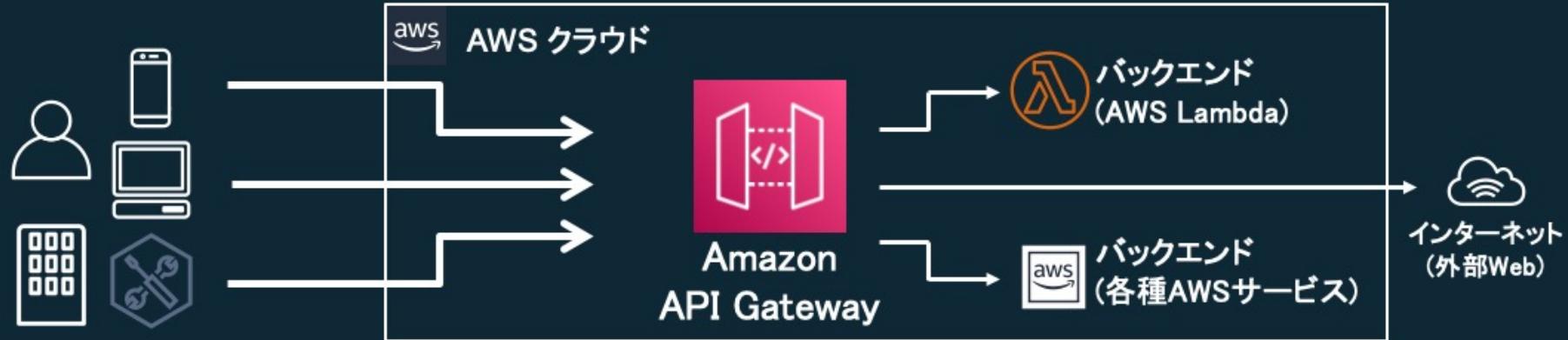
4. 流量制御と保護 (スロットリング)



- APIキーと使用量プランによるスロットリング
- WAF*連携によるAPIの保護

* WAF: Web Application Firewall

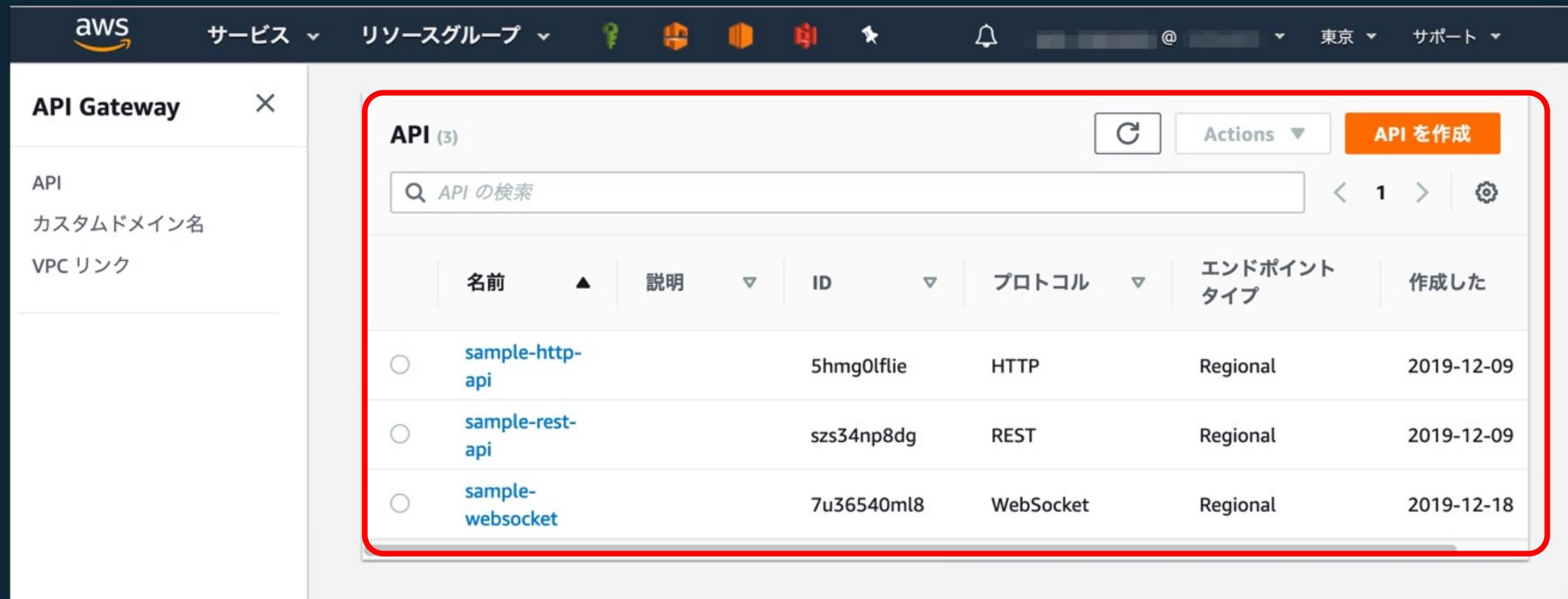
Amazon API Gateway のユースケース



1. インターネットからアクセス可能なパブリックなWeb API基盤を提供する
2. 自社企業・企業グループ内のプライベートなWeb API基盤を提供する
3. AWSサービスを独自にWeb API化する手段として利用する
4. サーバーレスアーキテクチャの実現手段として利用する

Amazon API Gateway によるAPI管理イメージ

Amazon API Gateway は、マネジメントコンソールや CLIなどによって API の作成や設定が可能

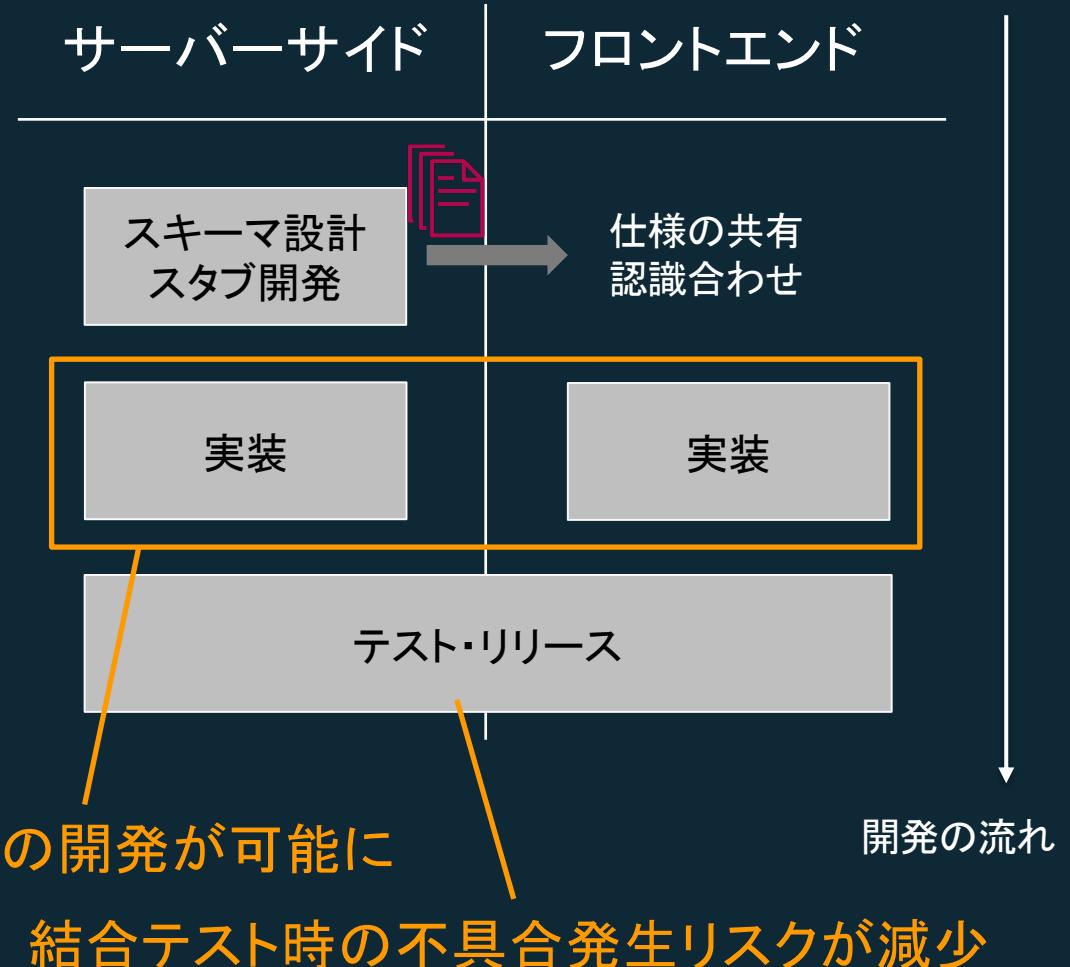


The screenshot shows the AWS Management Console interface for the API Gateway service. A red box highlights the central content area where three sample APIs are listed in a table.

名前	説明	ID	プロトコル	エンドポイントタイプ	作成した
sample-http-api		5hmg0lfie	HTTP	Regional	2019-12-09
sample-rest-api		szs34np8dg	REST	Regional	2019-12-09
sample-websocket		7u36540ml8	WebSocket	Regional	2019-12-18

外部APIの開発アプローチ

- 外部APIに関わらず、APIを開発する際は、スキーマ設計から始める**スキーマファースト開発**がおすすめです
- スキーマ定義を最初に作成してフロントエンド側と共有することで、開発やテスト作業の効率化が可能になります



外部APIの開発アプローチ

- APIのスキーマ定義においては、「OpenAPI」が用いられることが多い
 - OpenAPIとはRESTful APIを記述するためのフォーマット
 - YAMLまたはJSONの形式で作成可能
(コメントが書き込めるYAML形式がおすすめです)
 - API Gatewayでも対応しておりOpenAPIで作成したAPI仕様書をインポート・エクスポートすることが可能

A blurred photograph showing the silhouettes of several people's hands raised in the air, likely during a question-and-answer session at a conference or event.

Q&A



Thank you!

後日アンケートを送付します。運営改善のため回答にご協力お願いします。