[AWS Black Belt Online Seminar]

AWSにおけるマイクロサービスアーキテクチャ設計パターンと実装

サービスカットシリーズ

シニアソリューションアーキテクト
福井 厚
2020/03/25

AWS 公式 Webinar
https://amzn.to/JPWebinar

過去資料
https://amzn.to/JPArchive

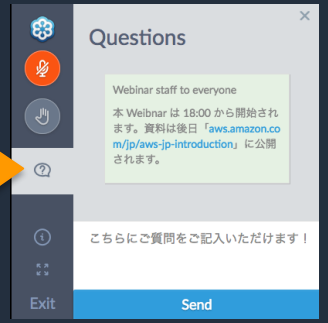
AWS Black Belt Online Seminar とは

「サービス別」「ソリューション別」「業種別」のそれぞれのテーマに分かれて、アマゾン ウェブ サービス ジャパン株式会社が主催するオンラインセミナーシリーズです。

質問を投げることができます！

- 書き込んだ質問は、主催者にしか見えません
- 今後のロードマップに関するご質問は
  お答えできませんのでご了承下さい

① 吹き出しをクリック
② 質問を入力
③ Sendをクリック



Twitter ハッシュタグは以下をご利用ください
#awsblackbelt

内容についての注意点

- 本資料では2020年3月25日時点のサービス内容および価格についてご説明しています。最新の情報はAWS公式ウェブサイト(http://aws.amazon.com)にてご確認ください。

- 資料作成には十分注意しておりますが、資料内の価格とAWS公式ウェブサイト記載の価格に相違があった場合、AWS公式ウェブサイトの価格を優先とさせていただきます。

- 価格は税抜表記となっています。日本居住者のお客様には別途消費税をご請求させていただきます。

- AWS does not offer binding price quotes. AWS pricing is publicly available and is subject to change in accordance with the AWS Customer Agreement available at http://aws.amazon.com/agreement/. Any pricing information included in this document is provided only as an estimate of usage charges for AWS services based on certain information that you have provided. Monthly charges will be based on your actual use of AWS services, and may vary from the estimates provided.

自己紹介

- **名前**
  - 福井 厚（ふくい あつし）fatsushi@
- **所属**
  - アマゾン ウェブ サービス ジャパン株式会社
  - 技術統括本部レディネスソリューション本部
  - シニアソリューションアーキテクト
サーバーレス スペシャリスト
- **関心領域**
  - ソフトウェア アーキテクチャ、オブジェクト指向設計、アジャイル開発
- **好きなAWSサービス**
  - サーバーレステクノロジー全般、 AWS Code シリーズ、AWS Amplify

本日の内容は、AWS モダンアプリケーション開発ホワイトペーパーの一部を解説するものです

- 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～
  https://d1.awsstatic.com/whitepapers/ja_JP/modern-application-development-on-aws.pdf?did=wp_card&trk=wp_card
- モダンアプリケーションにおける代表的な設計パターン
  （上記ホワイトペーパーの要約記事）
  https://aws.amazon.com/jp/builders-flash/202002/modern-app-wp/

本日のアジェンダ

- なぜマイクロサービスアーキテクチャが必要か
- マイクロサービスアーキテクチャ設計パターン
- まとめ

なぜマイクロサービス
アーキテクチャが必要なのか

急速なイノベーションはもはや必須

利益を伸ばす

今ある人材を活用

新たな機会を探索

新しいアイデアを育てる

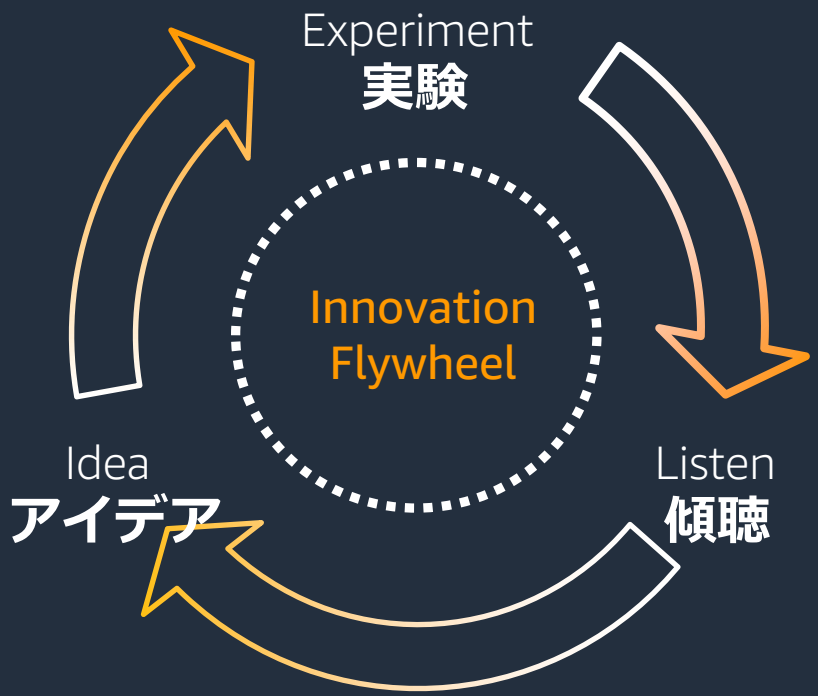
急速なイノベーションがビジネスを進化させる

新たなマーケット

新たな顧客価値

新たなデジタル製品とサービス

実験がイノベーションを加速する

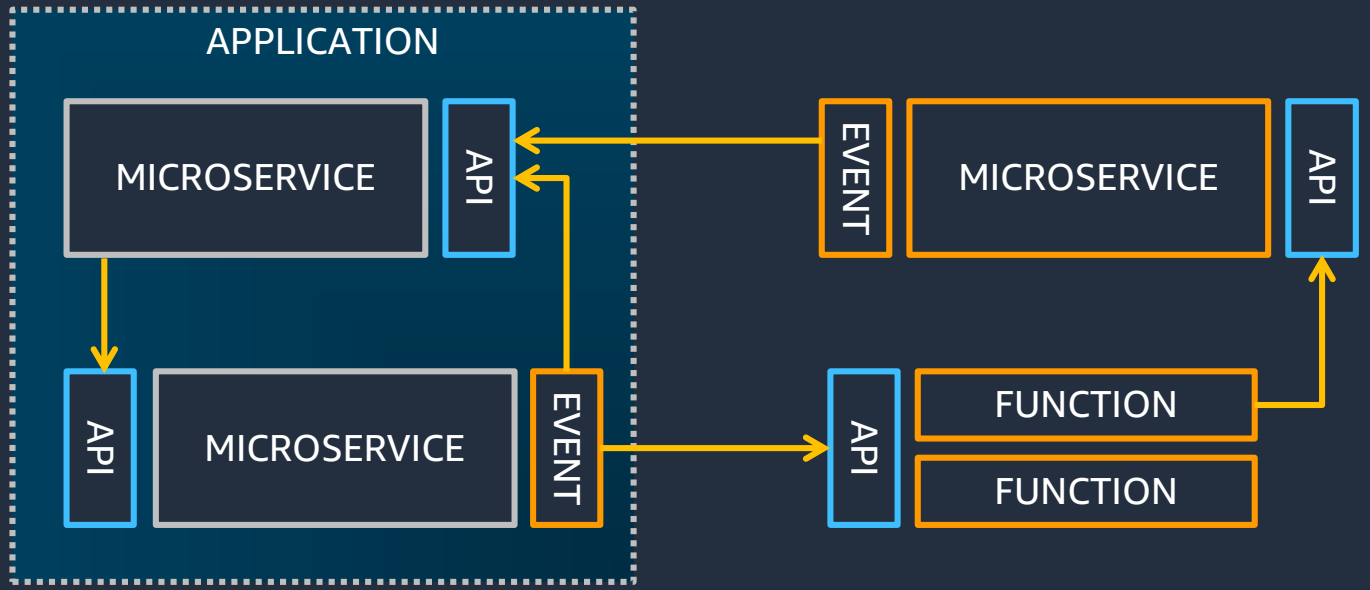


変更の影響が小さくなると、
リリースの速度が向上可能に

Monolith
すべてを実行

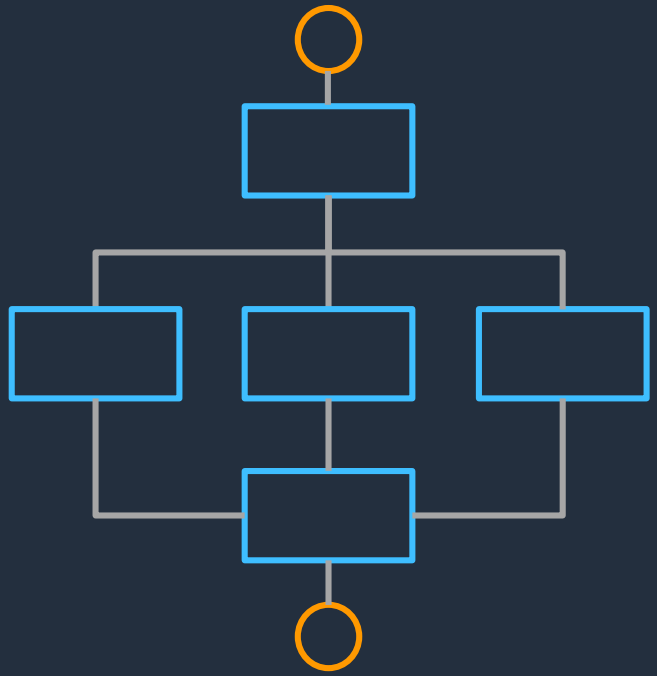
Microservices
ひとつのことを実行

「API」と「疎結合なコミュニケーション」が自動化を可能にして信頼性を向上



ワークフローで複数のサービスを連携することで、敏捷性、生産性、および柔軟性が向上

データと処理の状態をトラッキング



冗長なコードを削除

マイクロサービスアーキテクチャ設計パターン

ソフトウェア設計（デザイン）パターンとは

- ソフトウェア開発における**デザインパターン**（型紙（かたがみ）または**設計パターン**、英: design pattern）とは、過去のソフトウェア設計者が発見し編み出した設計ノウハウを蓄積し、名前をつけ、再利用しやすいように特定の規約に従ってカタログ化したものである。（出典: フリー百科事典『ウィキペディア（Wikipedia）』）

- Name、Context、Problem、Forces、Solutionなどのパターンテンプレートを利用して記述される

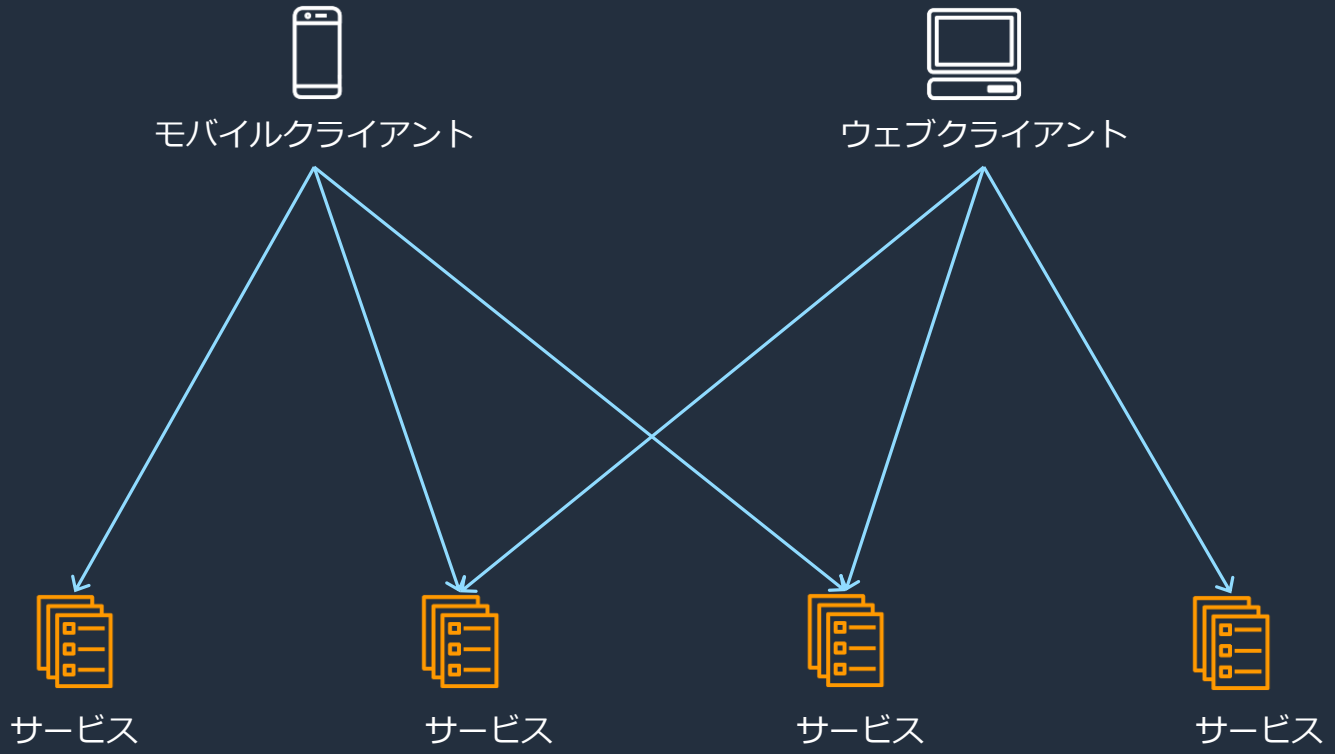
- GoF（Gang of Four）の「オブジェクト指向における再利用のためのデザインパターン」が有名

APIゲートウェイパターン

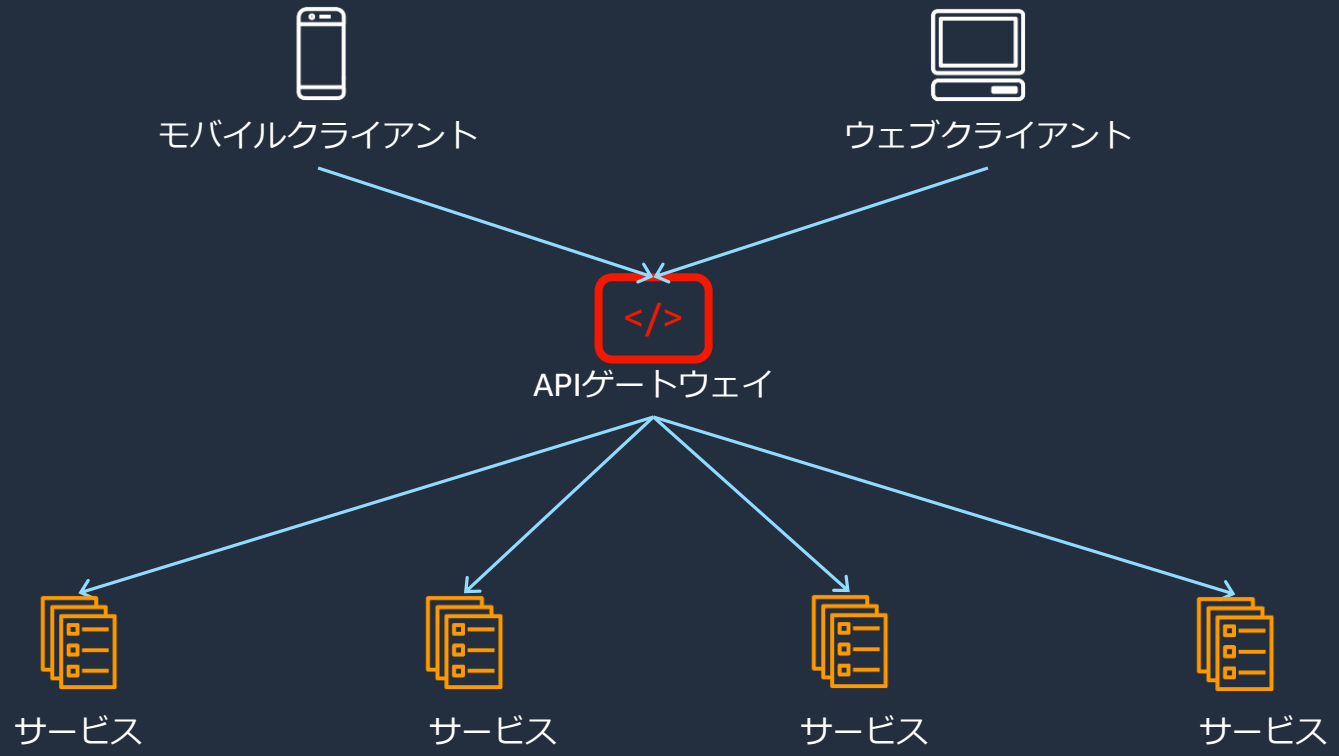
API ゲートウェイパターンとは

- Backends for Frontends（BFF）パターンとも言われる
- 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可能
- 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の呼び出しを統一された API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

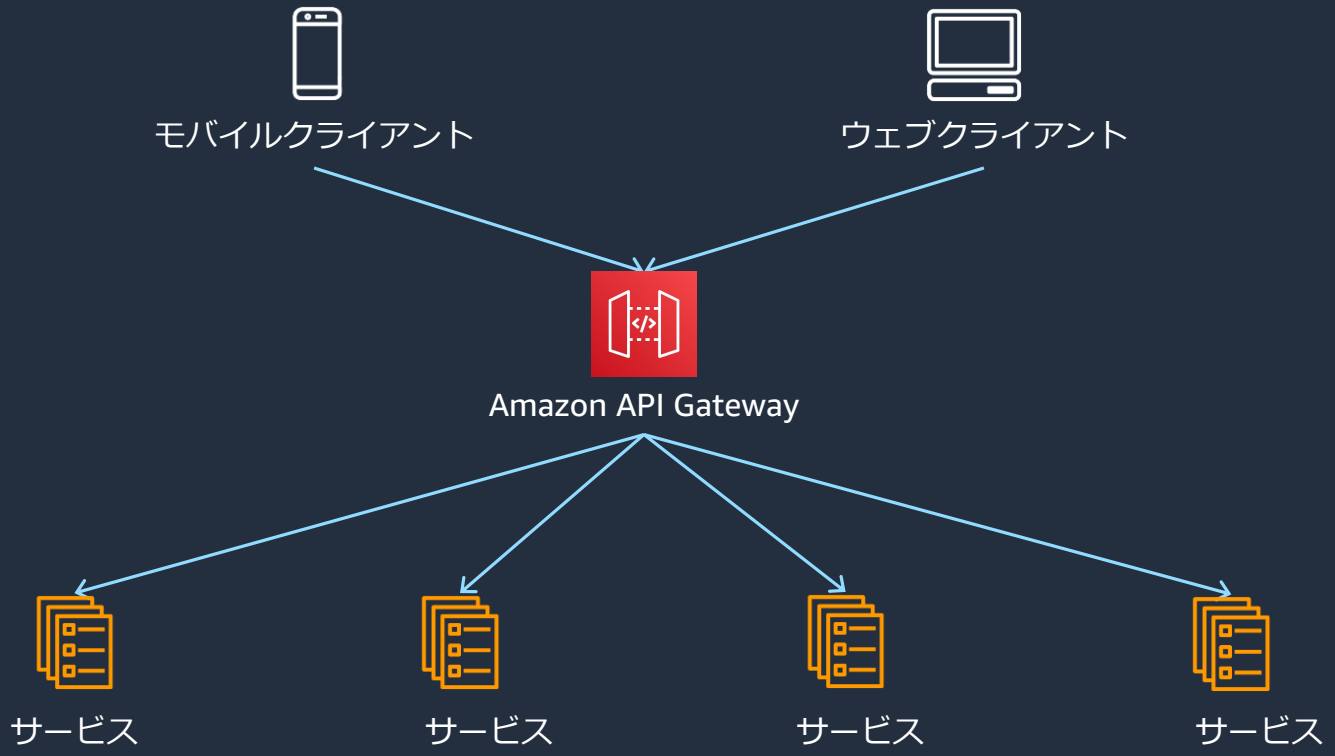
API ゲートウェイパターン



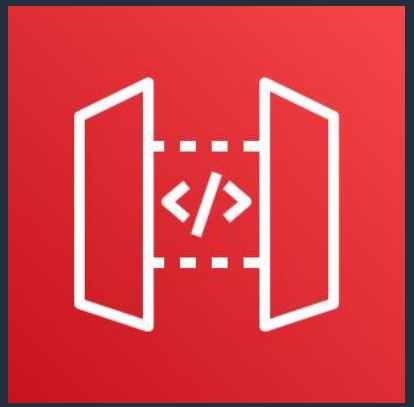
API ゲートウェイパターン



Amazon API Gateway を利用



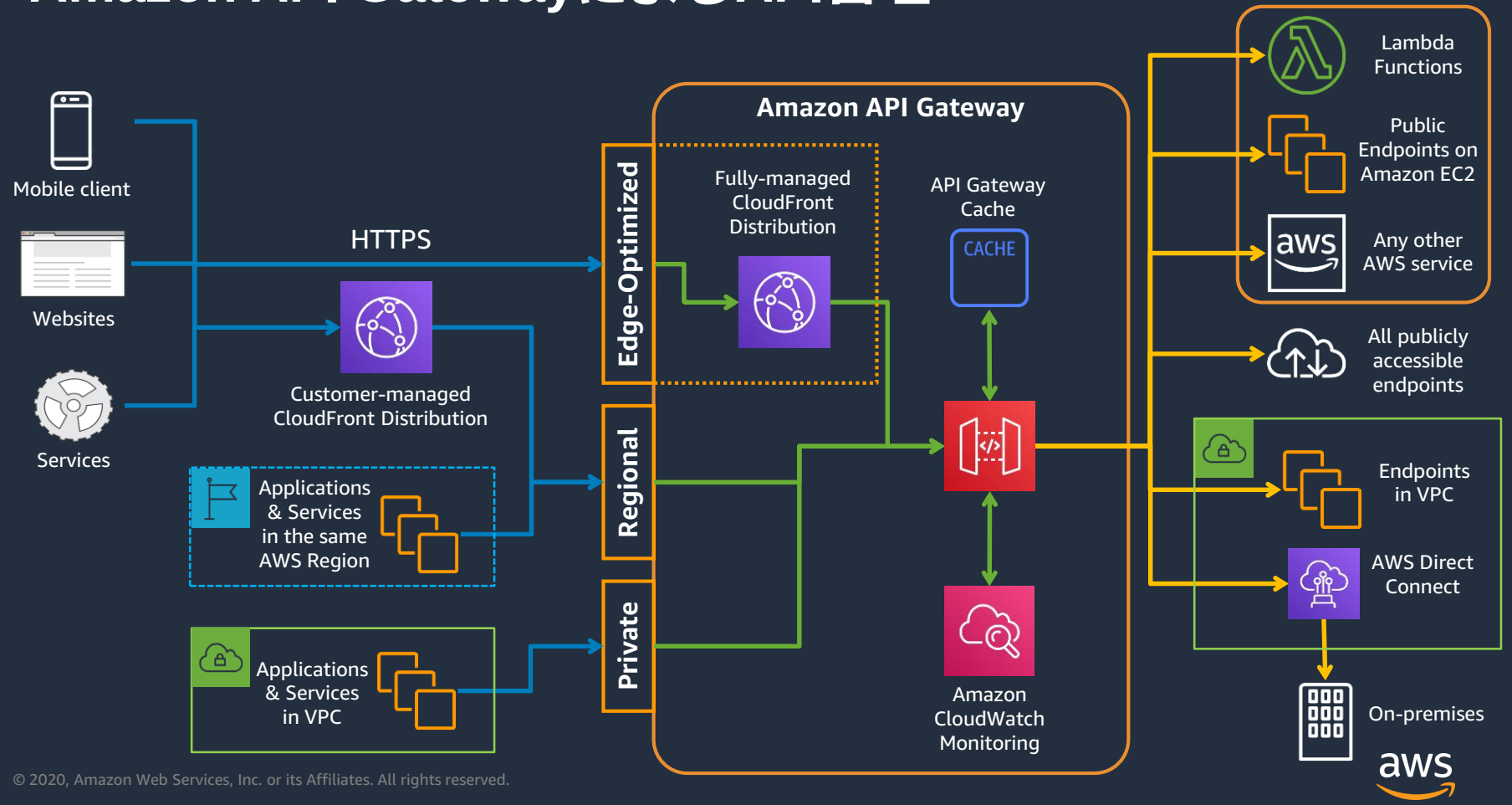
Amazon API Gateway とは



Amazon API Gatewayはあらゆるスケールのセキュアな APIを容易に作成、発行、保守、モニタ可能なフルマネージドなサービス：

- APIの複数バージョンとステージをホスト
- APIキーの作成と開発者への配布
- バックエンド保護のためのリクエストのスロットルとモニタ
- APIアクセスの認証にSIG V4、AWS Lambdaオーサライザー、Amazon Cognitoユーザープールなどの認証を利用可能
- リクエスト/レスポンス データ変換とAPIモック
- CloudFrontを通じたレイテンシの削減とDDoS保護
- APIレスポンスのためのオプションのマネージドキャッシュ
- APIのためのJava、JavaScript、Java for Android、Objective-C、Swift for iOS、RubyのSDKを生成
- Swagger/OpenAPI サポート

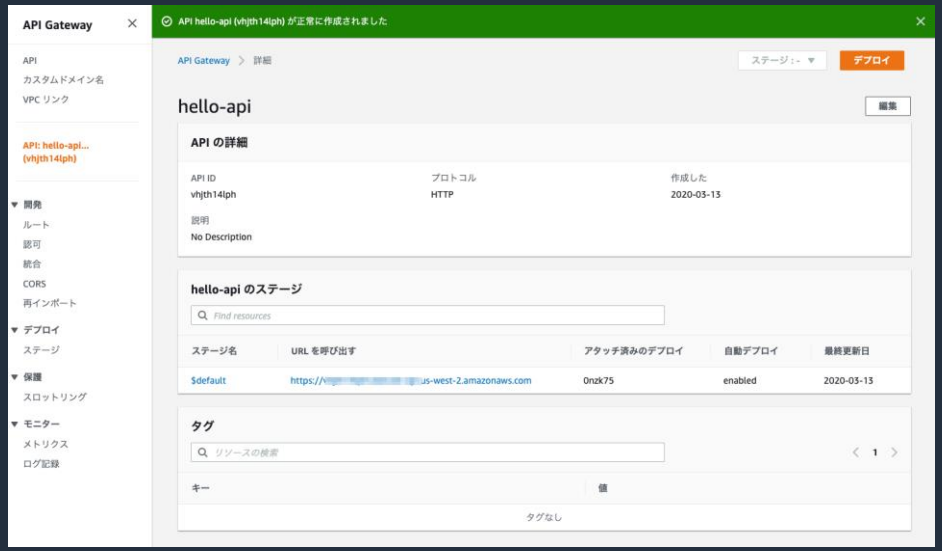
Amazon API GatewayによるAPI管理

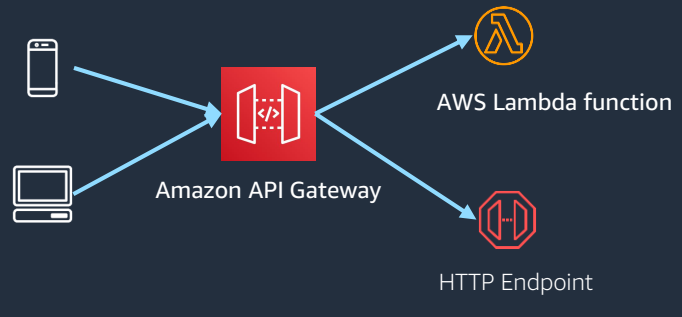


HTTP APIs for Amazon API Gateway が一般利用可能に！

New!

- 低コスト（最大71%）
- 低レイテンシ（最大60%低減）
- Lambda, HTTP プロキシ統合
- プライベート統合
  - ALB/NLB/Cloud Map
- カスタムドメインの相互互換性
- リクエストスロットリング
- JWT オーソライザ
- ステージへのオートデプロイ
- CORS 設定の簡素化
- Lambdaペイロード Version 2.0
- ルートスロットリング
- プロトコルは HTTPS





aws

Amazon API Gatewayについて詳しくは

- [AWS Black Belt Online Seminar] Amazon API Gateway もご参照ください。
  https://aws.amazon.com/jp/blogs/news/webinar-bb-amazon-api-gateway-2019/

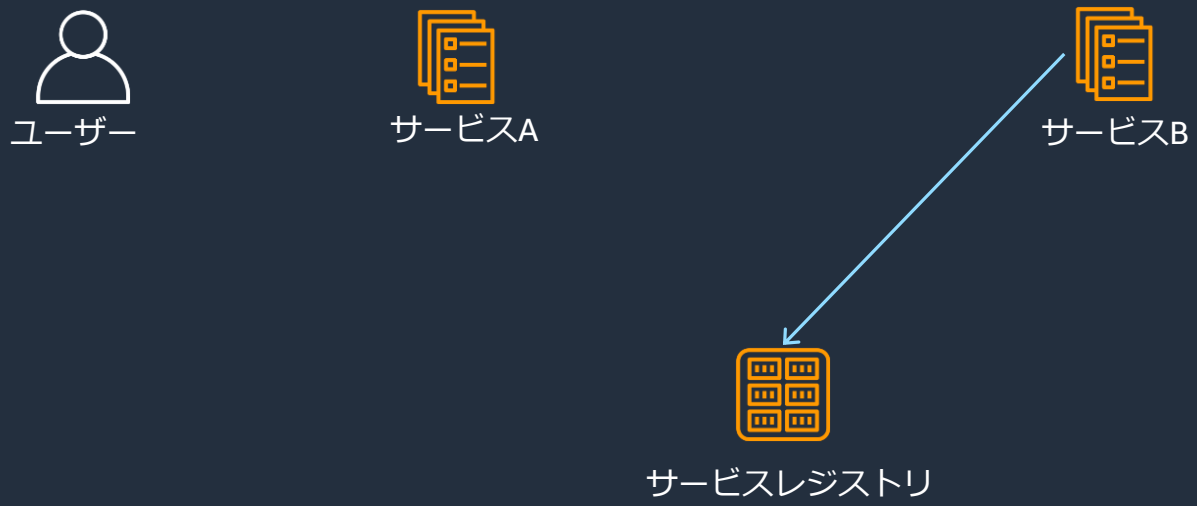
サービス ディスカバリ/サービス レジストリパターン

サービス ディスカバリ/サービス レジストリパターンとは

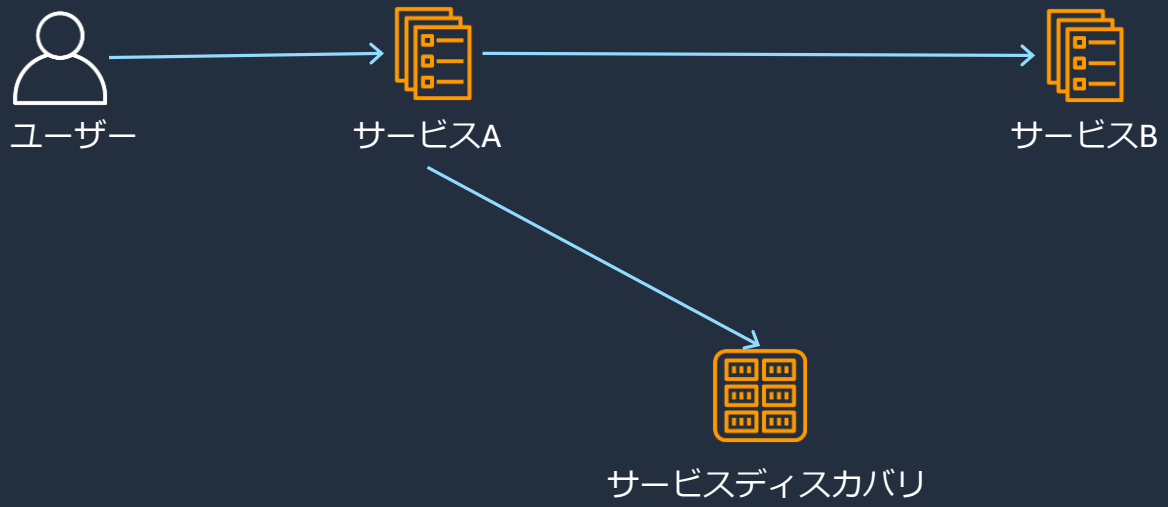
- サービスが他のサービスを利用する際の課題を解決
- **サービスレジストリ**は、個々のコンテナやインスタンスの起動時に、呼び出されるサービスがそのサービス自体の情報を保存
- **サービスディスカバリ**は、動的に変更する関連サービスのアドレスを取得

aws

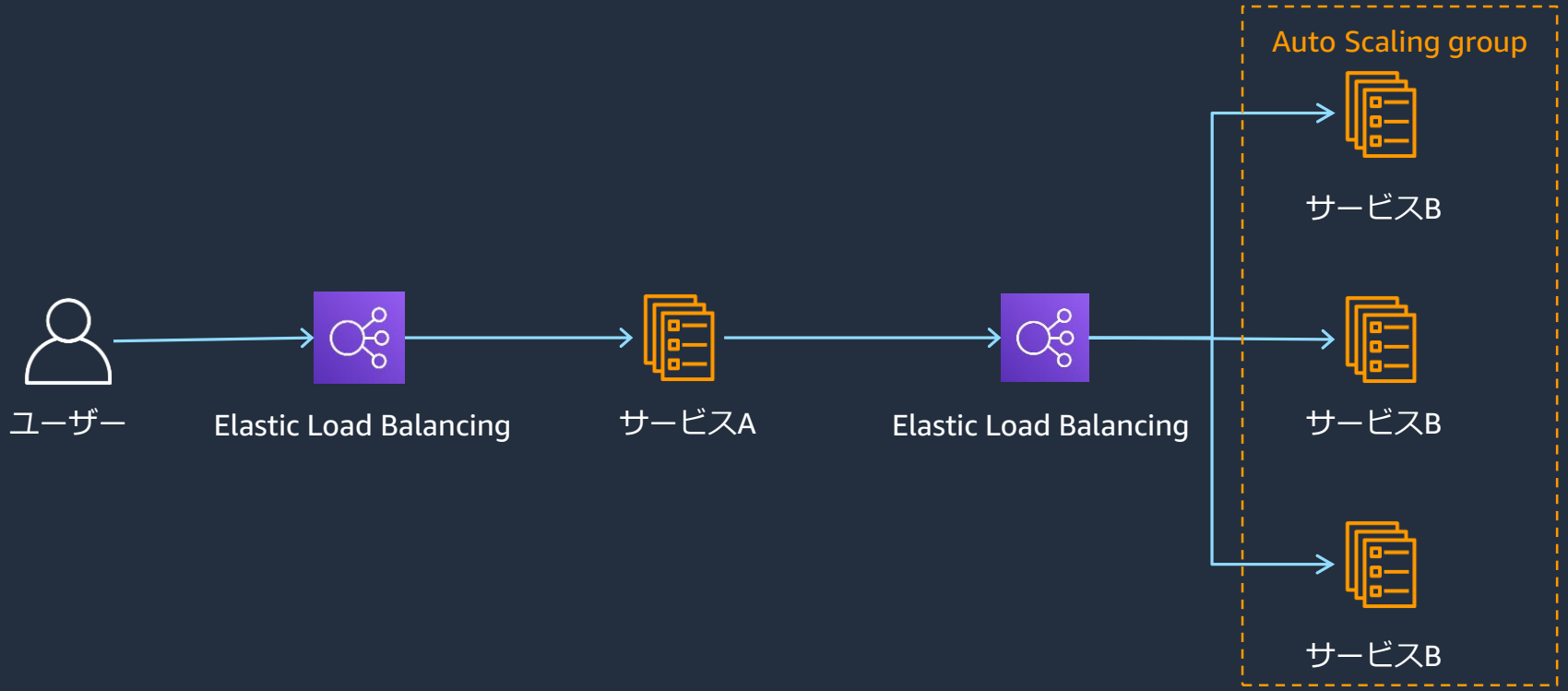
サービス レジストリパターン



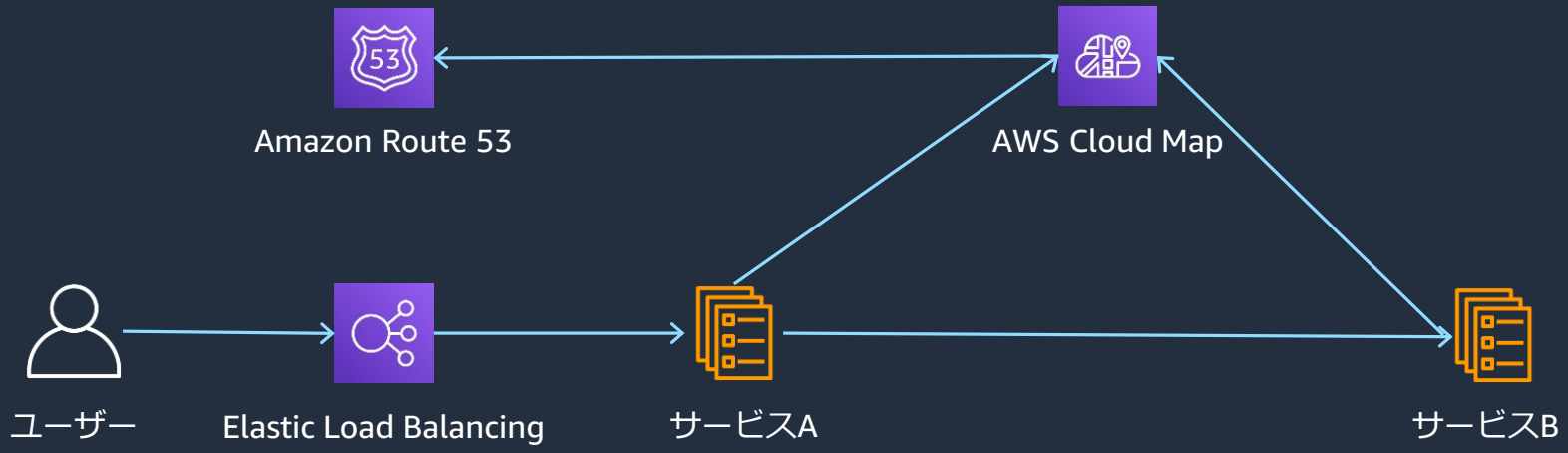
サービス ディスカバリパターン



実装例１：Elastic Load Balancing を利用



実装例２： AWS Cloud Map の利用



AWS Cloud Mapの 機能紹介

すべてのクラウドリソースのためのレジストリ

高速でセキュアな名前解決

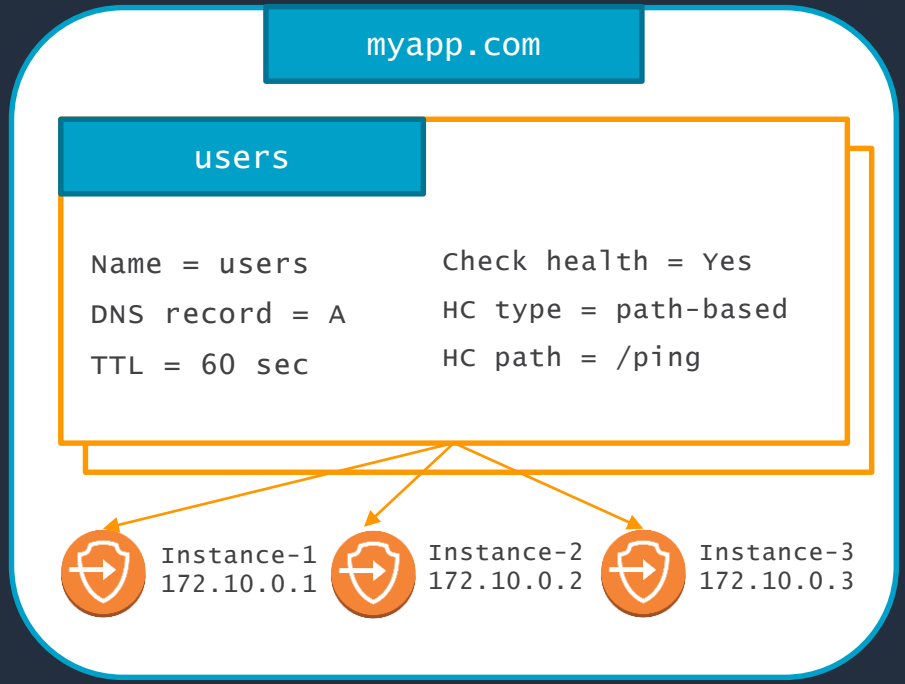
属性ベースのディスカバリ

Amazon Route 53のヘルスチェックによる一部エラーのハンドリング

AWSクラウドやOSSとの統合

AWS Cloud Map サービスレジストリ

- 名前空間
- サービス
- サービスインスタンス (リソース)



API + パブリックDNS ディスカバリモードのリソース登録

```
1. aws servicediscovery create-public-dns-namespace --name
cloudmapdemo.com

2. aws servicediscovery create-service --name frontend
   --dns-config “NamespaceId=%namespace_id%, DnsRecords=[{Type=A,
TTL=60}]”
3. aws servicediscovery register-instance --service-id
%service_id% --instance-id %id%
   --attributes
   AWS_INSTANCE_IPV4=52.89.144.60,
   stage=beta,
   version=1.0,
   ready=yes
```


aws

API ディスカバリのクラウドリソース登録

```
1. aws servicediscovery create-http-namespace --name shared

2. aws servicediscovery create-service --name logs --namespace-id
%namespace_id%

3. aws servicediscovery register-instance --service-id
%service_id% --instance-id %id%
   --attributes
   ARN=arn:aws:s3:::cloudmapdemoservicelogsbeta1,
   stage=beta,
   shard=s_1,
   read_only=no,
   path=/mylogs
```


aws

APIコールによるセキュアな名前解決

```
aws servicediscovery discover-instances --namespace-name shared --service-
name logs
-->
{    "Instances": [
        {
            "InstanceId": "i1",
            "NamespaceName": "shared",
            "ServiceName": "logs",
            "HealthStatus": "UNKNOWN",
            "Attributes": {
                "read_only": "no",
                "path": "/mylogs",
                "shard": "s_1",
                "ARN": "arn:aws:s3:::cloudmapdemoservicelogsbeta1",
                "stage": "beta”
            }
        }
    ]
}
```

Python SDKを利用したサービスインスタンスの登録例

```
def register_service():
    client = boto3.client('servicediscovery')

    response = client.register_instance(
        ServiceId = SERVICE_ID,
        InstanceId = INSTANCE_ID,
        Attributes = {
            'version': '1.0',
            'stage': 'prod',
            'AWS_INSTANCE_IPV4': '34.209.47.250',
            'AWS_INSTANCE_PORT': '80'
        }
    )
    return response
```

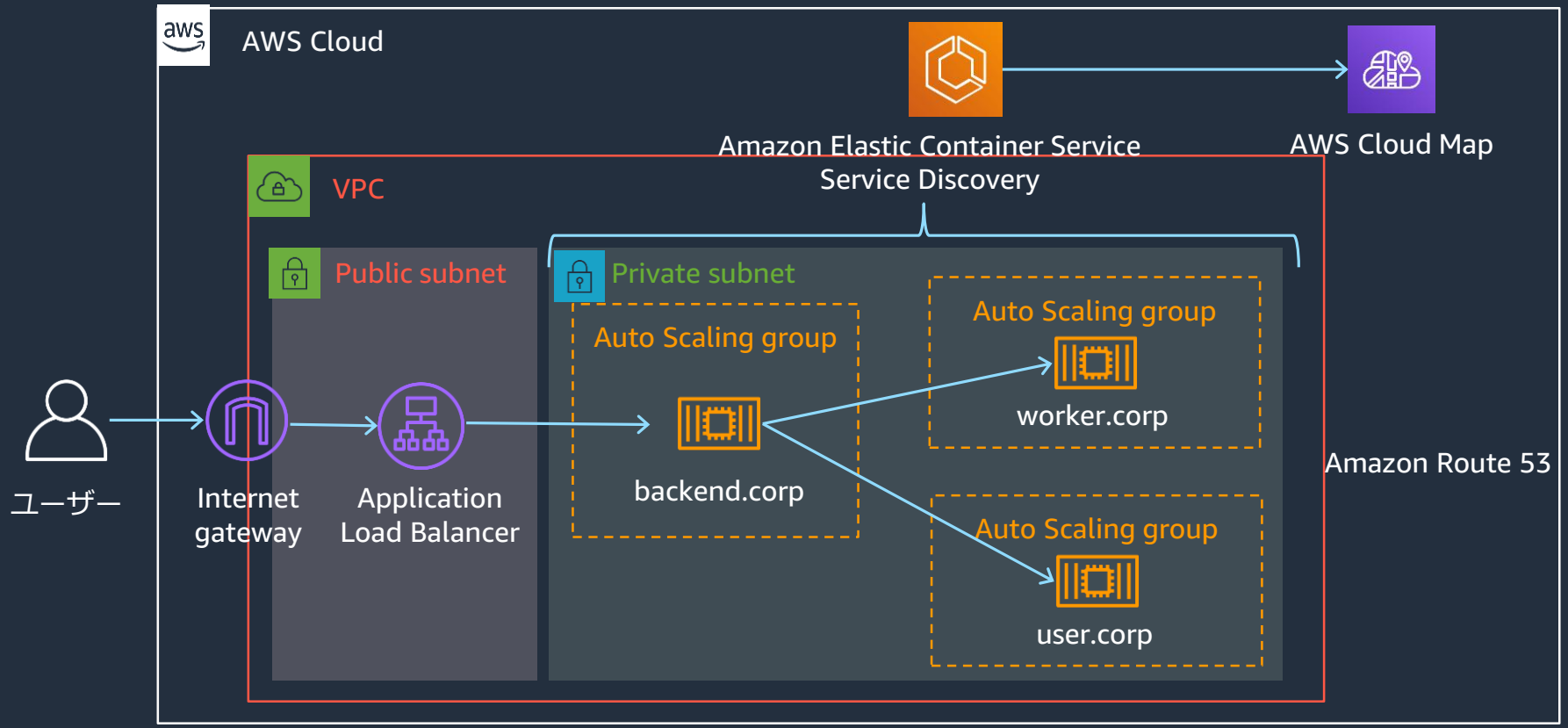

aws

Python SDKを利用したサービスインスタンスの取得例

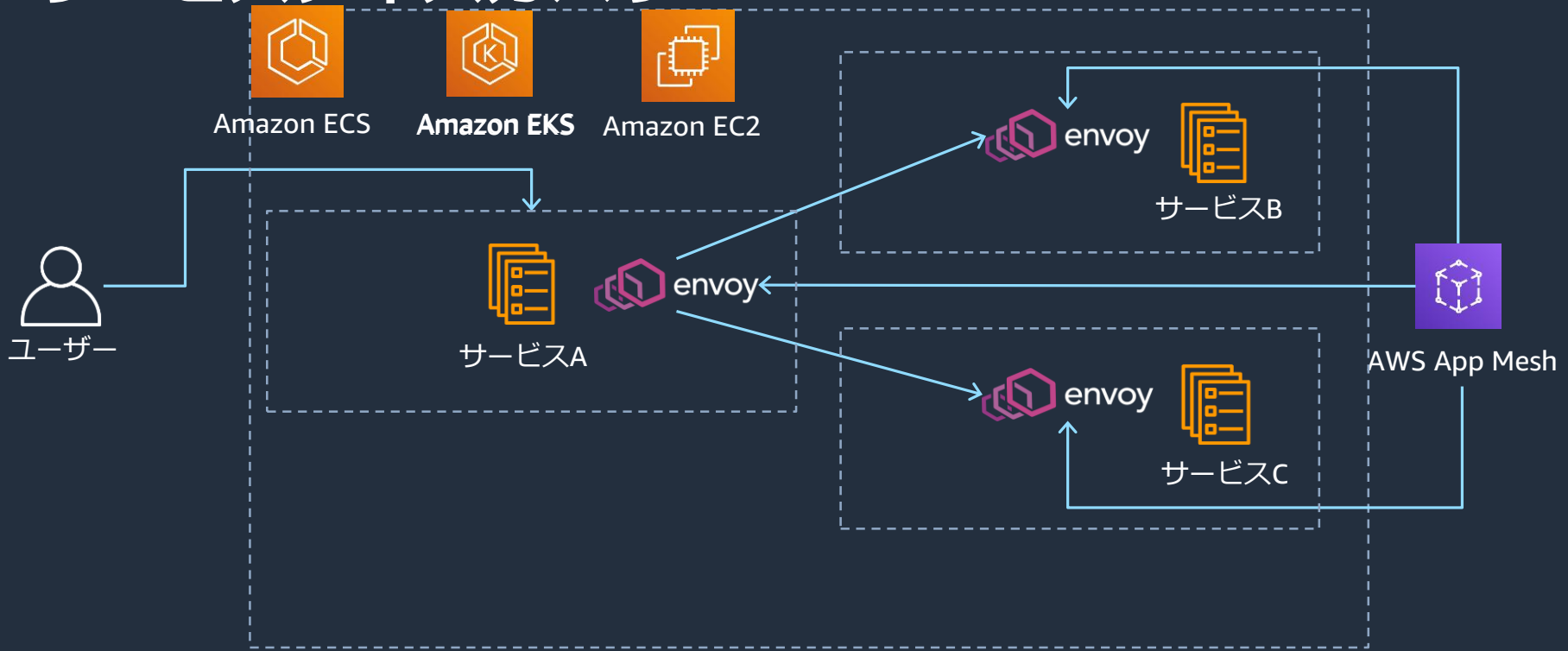
```
def discover_services():
    client = boto3.client('servicediscovery')

    response = client.discover_instances(
        NamespaceName = NAMESPACE_NAME,
        ServiceName = SERVICE_NAME,
        QueryParameters = {
            "version": "1.0",
            "stage": "prod"
        }
    )
    return response
```


aws

実装例３： Amazon ECSのサービスディスカバリを利用



実装例４： AWS App Meshのサービスメッシュのサービスディスカバリ



AWS App Mesh

アプリケーションレベルのネットワーク

- ログ・メトリクス・トレース情報の容易な出力
- トラフィック・ルーティングポリシー

クラスタやサービスにまたがるメッシュの構築

- Amazon ECS
- AWS Fargate
- Amazon EKS
- Kubernetes on EC2
- Amazon EC2

マネージドコントロールプレーン

- 容易なオペレーション
- 高いスケーラビリティ

透過的な実装のメリット

- 可観測性、分析性、ルーティングの機能をサービス間で一貫した形で適用可能

- 2つのサービスの相互接続間で、ロギング、トレーシング、メトリックスの収集をアプリケーションコードへの影響を最小限になるように実装

- これらの機能を提供するSDKを利用してコンパイルする ”インプロセスパターン”を利用と比較して
    - すべてのメンバーがSDKを学習する必要がない
    - Polyglot 開発を実現している場合、多数の言語用のSDKを用意しなくて良い
    - SDKの各言語用のバージョン間の差異に苦しむことがない
    - SDKの組み込みはアプリとオペレーションの密結合を生む

aws

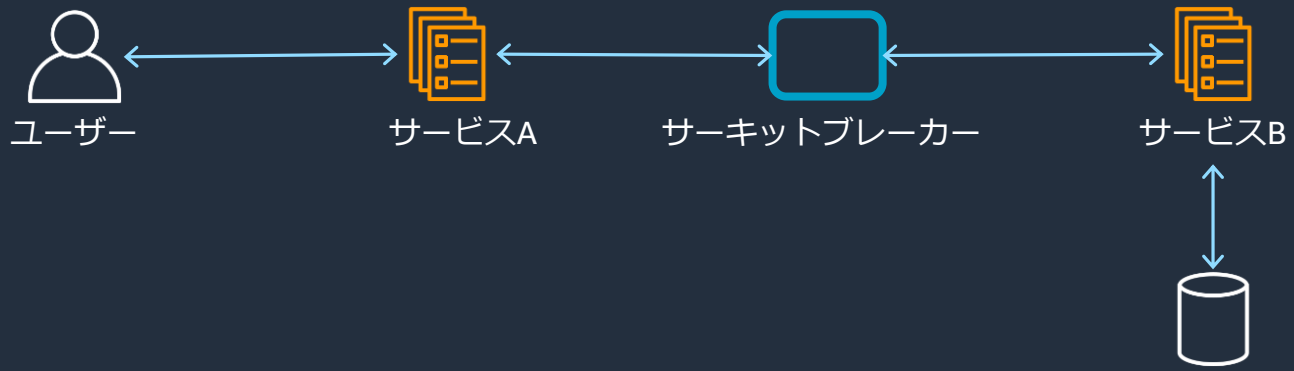
サーキット ブレーカーパターン

サーキットブレーカーとは

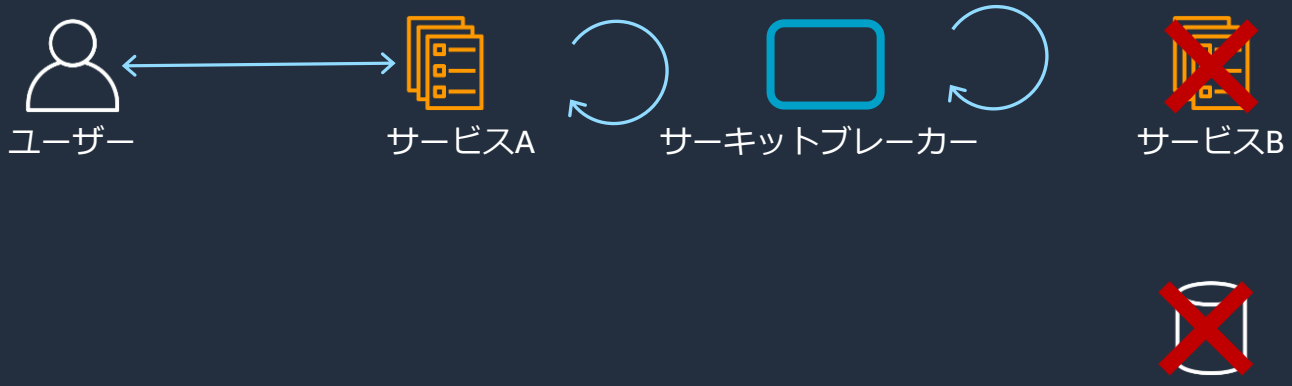
- マイクロサービス間の呼び出しを規制し、グレースフルな機能低下を実現
- サービスが相手サービスの呼び出しに通常より長い時間がかかった場合やエラーが返された場合、サーキットブレーカーがインシデント数をカウント
- カウントが設定された制限を超えると**オープン状態**に移行、オープン状態になると、サーキットブレーカーは呼び出し元に即座にエラーを返却
- 一定時間が経過、またはヘルスチェックによって相手サービスが復活すると、サーキットブレーカーは**クローズ状態**に戻り相手サービスへの呼び出しも平常時の状態に戻る
-

aws

サーキットブレーカーパターン

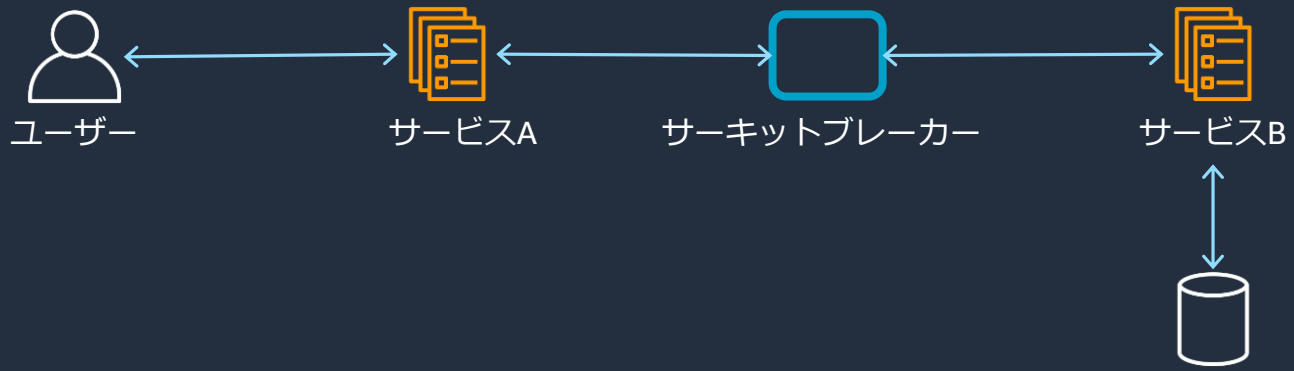


サーキットブレーカーパターン

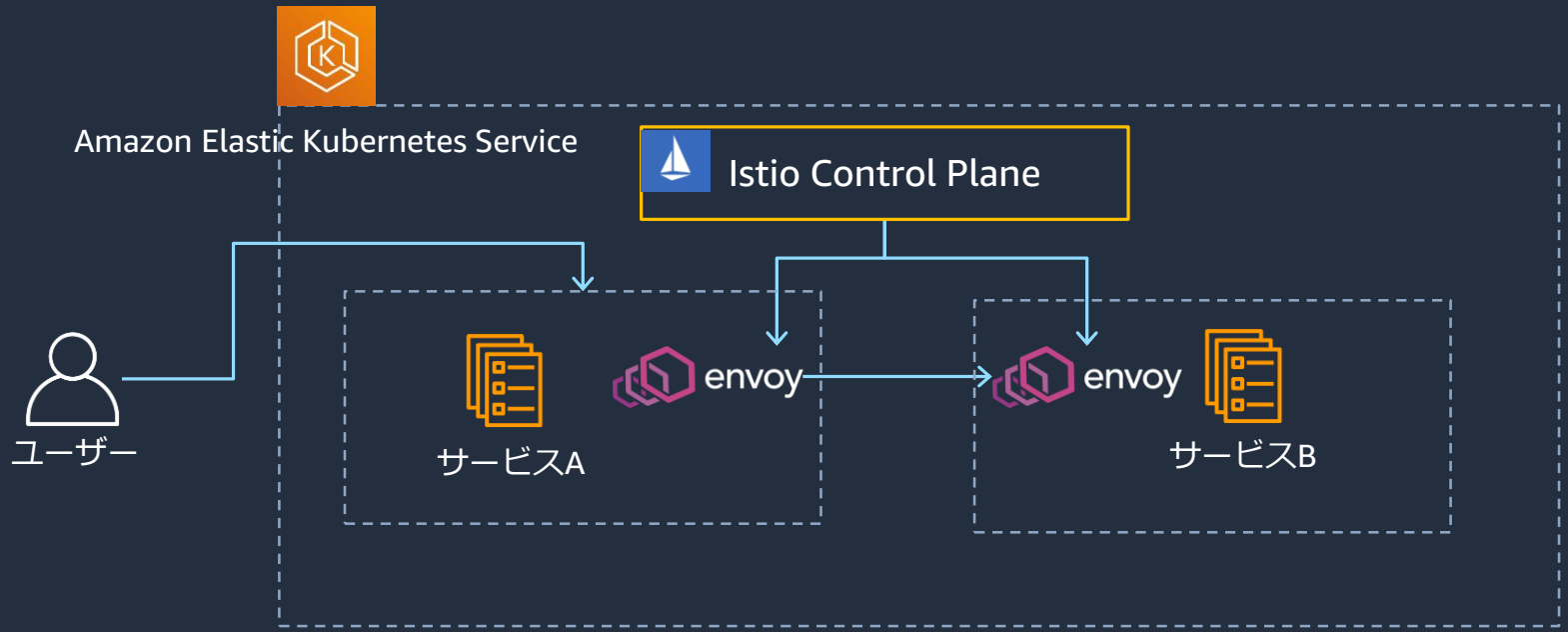


サーキットブレーカーパターン

サーキットブレーカーパターン



IstioとEnvoy ProxyをAmazon EKSで利用



Istio DestinationRuleによるサーキットブレーカー設定

```yaml
apiVersion:
networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: httpbin
spec:
  host: httpbin
  trafficPolicy:
    connectionPool:
      tcp:
        maxConnections: 1
      http:
        http1MaxPendingRequests: 1
        maxRequestsPerConnection: 1
```

```yaml
outlierDetection:
      consecutiveErrors: 2
      interval: 1s
      baseEjectionTime: 30s
      maxEjectionPercent: 100
```

consecutiveErrors: 継続したエラーと判断する閾値
Interval: 解析を行う間隔
baseEjectionTime: ホストがイジェクトされる最小時間。この時間とイジェクト回数を掛けた時間イジェクトされる
maxEjectionPercent: 異常値検知によってイジェクトされるアップストリームのホストの最大閾値%

aws

コマンドクエリ責務分離（CQRS）パターン

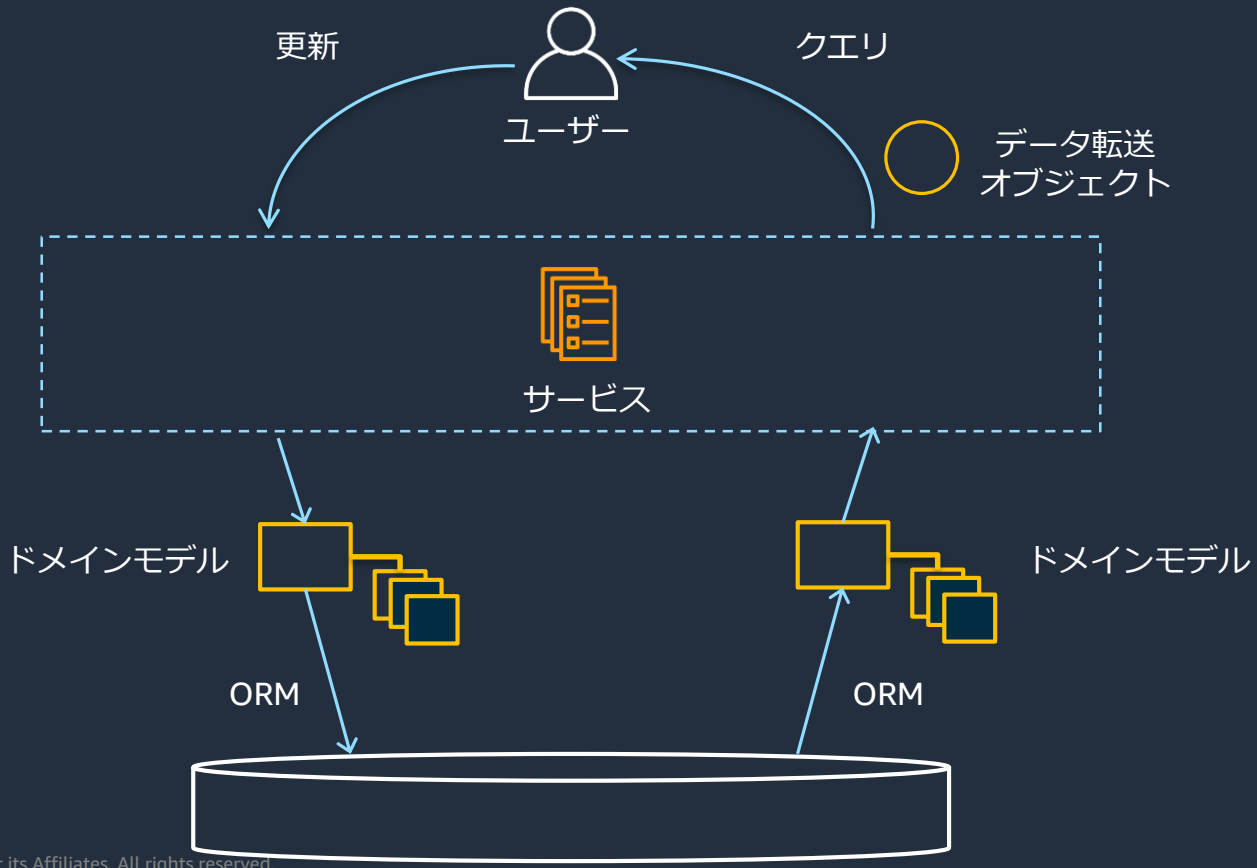
コマンドクエリ責務分離パターン

- コマンドクエリ責務分離（Command-Query Responsibility Segregation：CQRS）パターンはデータを更新するコマンドと参照するクエリを分離することで、ユースケースに応じて個別にスケールすることを可能とするパターン

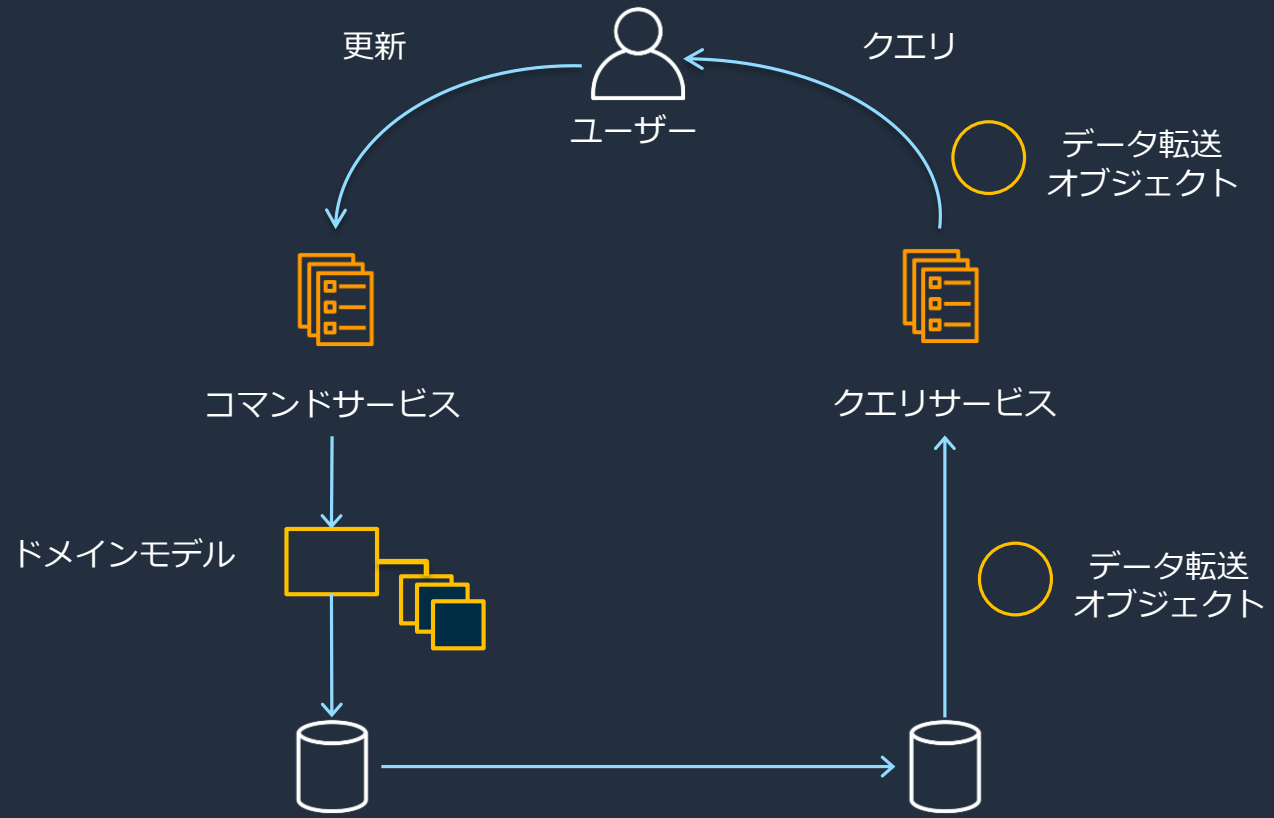
- データソースを分離することで異なるデータ構造を取ることも可能となり、クエリ側はクエリで返すデータ転送オブジェクト（Data Transfer Object : DTO）に合わせた形のスキーマとすることでオブジェクト関係マッピング（Object Relational Mapping : ORM）のオーバーヘッドを軽減することも可能になる

- コマンド側とクエリ側が結果整合性を許容する必要がある

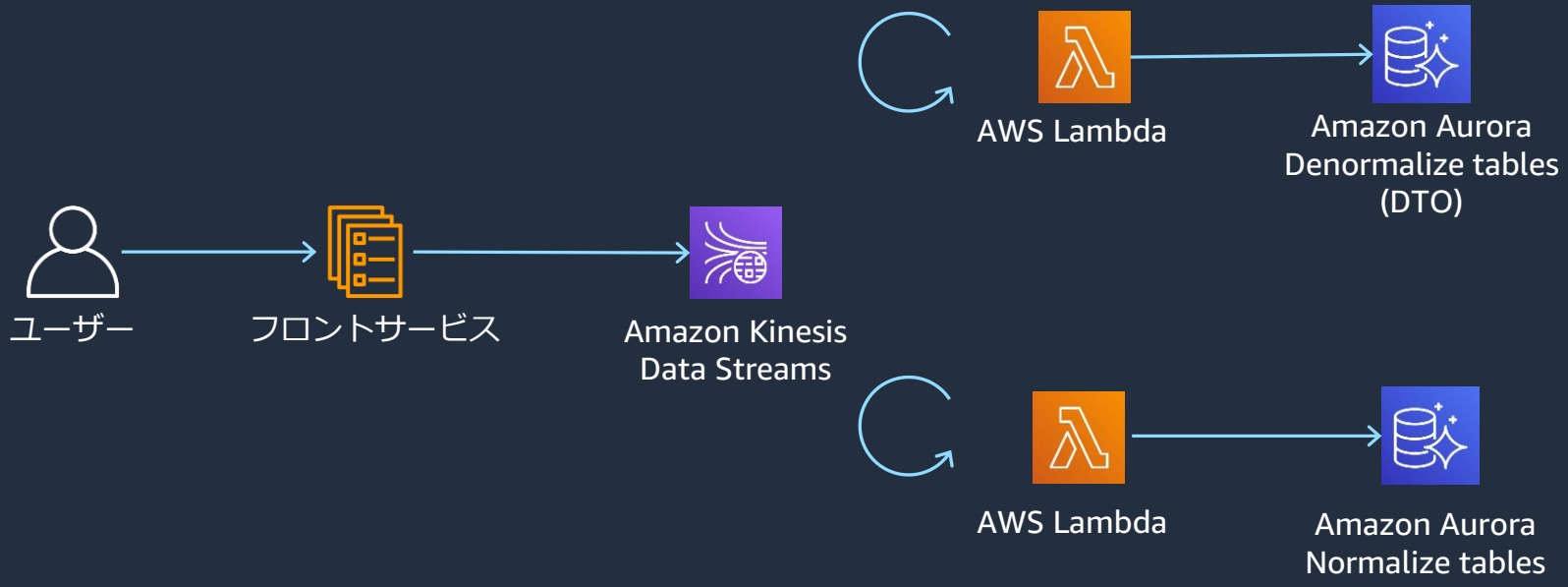
コマンドクエリ責務分離（CQRS）パターン



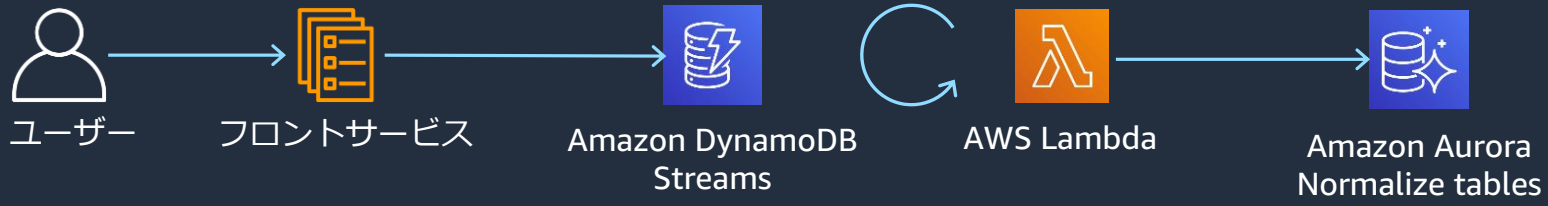
コマンドクエリ責務分離（CQRS）パターン



実装例１： Amazon Kinesis Data StreamsとAWS Lambdaを利用



実装例２： Amazon DynamoDB StreamsとAWS Lambdaを利用



Amazon DynamoDB Streamsの有効化

```
aws dynamodb create-table ¥
    --table-name StreamsDemo ¥
    --attribute-definitions ¥
 AttributeName=Username,AttributeType=S ¥
 AttributeName=Timestamp,AttributeType=S ¥
    --key-schema AttributeName=Username,KeyType=HASH ¥
 AttributeName=Timestamp,KeyType=RANGE ¥
    --provisioned-throughput ¥
ReadCapacityUnits=5,WriteCapacityUnits=5 ¥
    --stream-specification ¥
StreamEnabled=true,StreamViewType=NEW_AND_OLD_IMAGES
```


aws

DynamoDB Streams Event パラメータ


```
{
    "Records": [{
        "eventID": "82e65165cd273bb5c8ac9c5b16604bfc",
        "eventName": "INSERT",
        "eventVersion": "1.1",
        "eventSource": "aws:dynamodb",
        "awsRegion": "us-west-2",
        "dynamodb": {
            "ApproximateCreationDateTime": 1584252775.0,
            "Keys": {
                "Username": {
                    "S": "fatsushi"
                },
                "Timestamp": {
                    "S": "2020-03-15 06:12:19.177560"
                }
            },
            "NewImage": {
                "Username": {
                    "S": "fatsushi"
                },
                "Timestamp": {
                    "S": "2020-03-15 06:12:19.177560"
                }
            },
            "SequenceNumber": "100000000048382961814",
            "SizeBytes": 102,
            "StreamViewType": "NEW_AND_OLD_IMAGES"
        },
        "eventSourceARN": "arn:aws:dynamodb:us-west-2:786032344772:table/StreamsDemo/stream/2020-03-15T04:45:48.745"
    }]
}
```

AWS Lambda イベントハンドラ

```
import json

print('Loading function')
def lambda_handler(event, context):
    for record in event['Records']:
        print(record['eventID’])
        print(record['eventName’])
        print(json.dumps(record['dynamodb']))
        # ここで更新データを元に処理を実施

    return 'Successfully processed {} records.’
.format(len(event['Records']))
```


aws

イベント ソーシングパターン

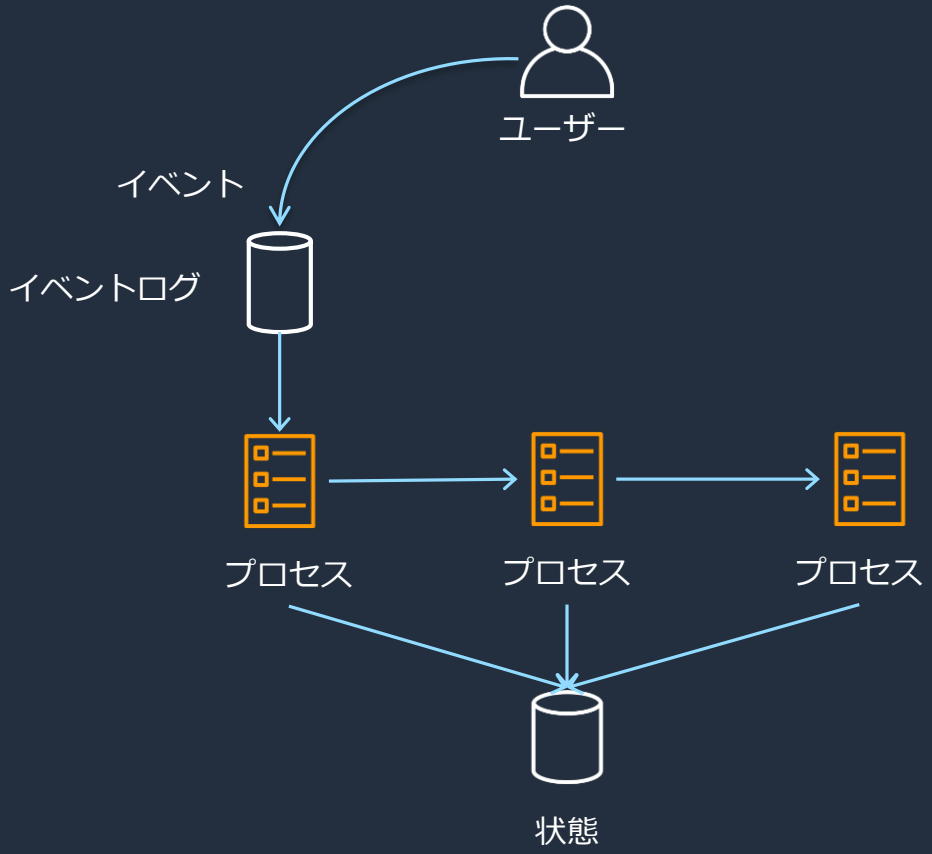
イベントソーシングパターンとは

- 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、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

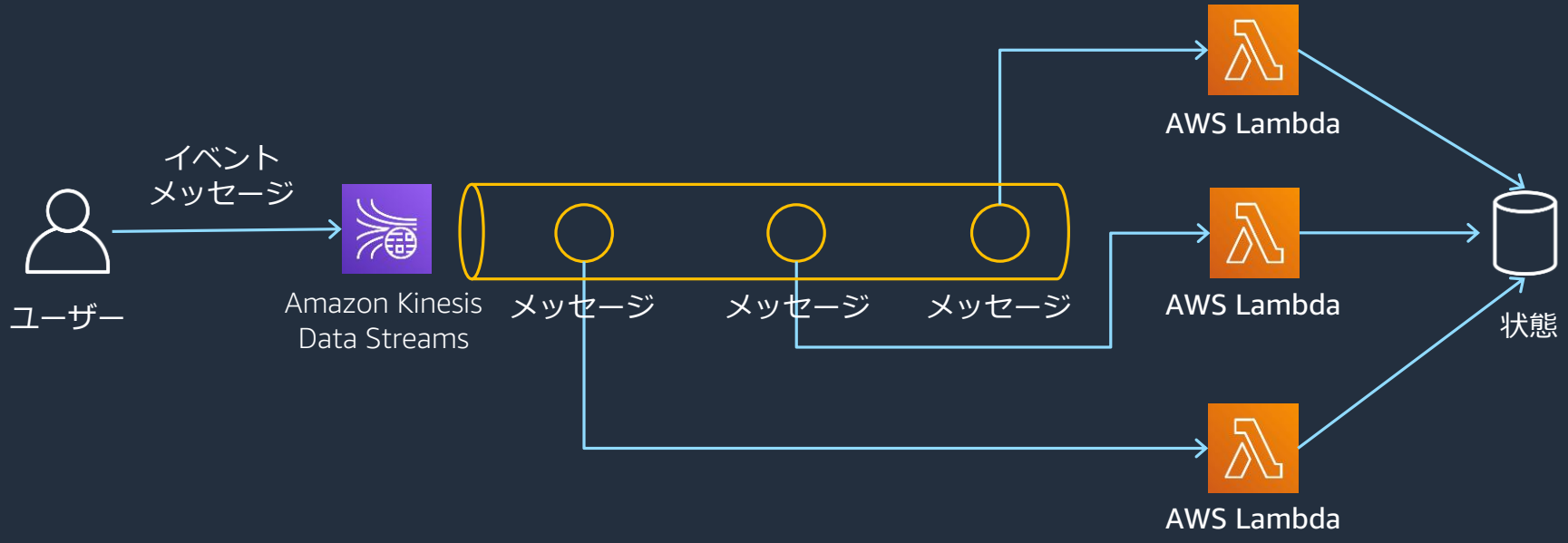
- イベントレコードは個別に保存されるため、すべての更新は**アトミック** (分割不可かつ削減不可)

- 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 可能

イベント ソーシングパターン



Amazon Kinesis Data StreamsとAWS Lambdaによる実装例



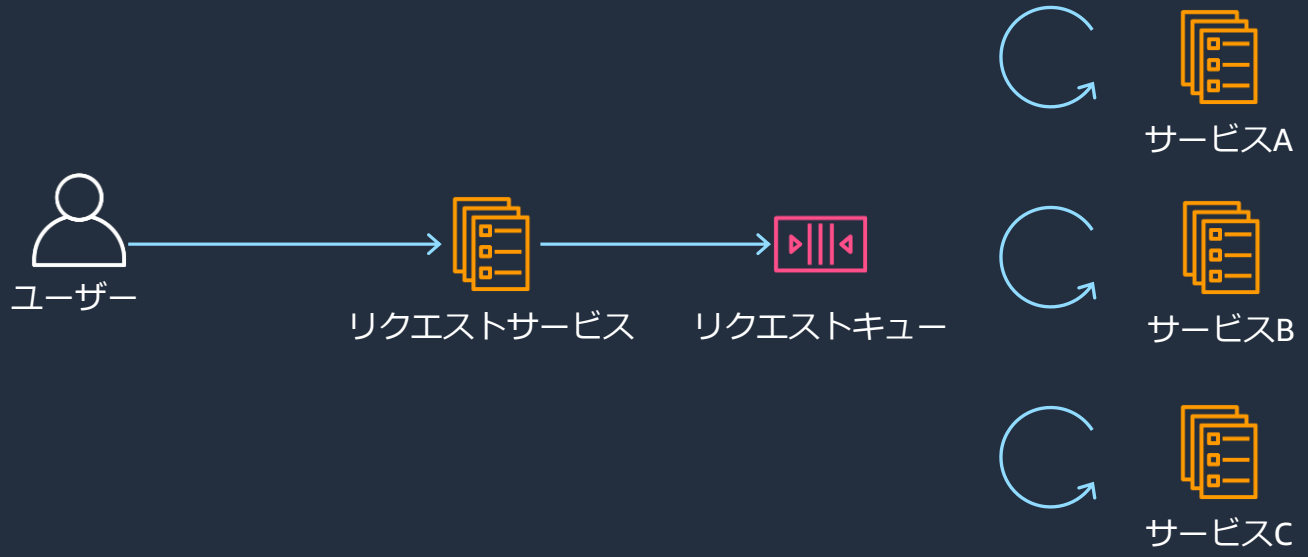
コレオグラフィーパターン

コレオグラフィーパターンとは

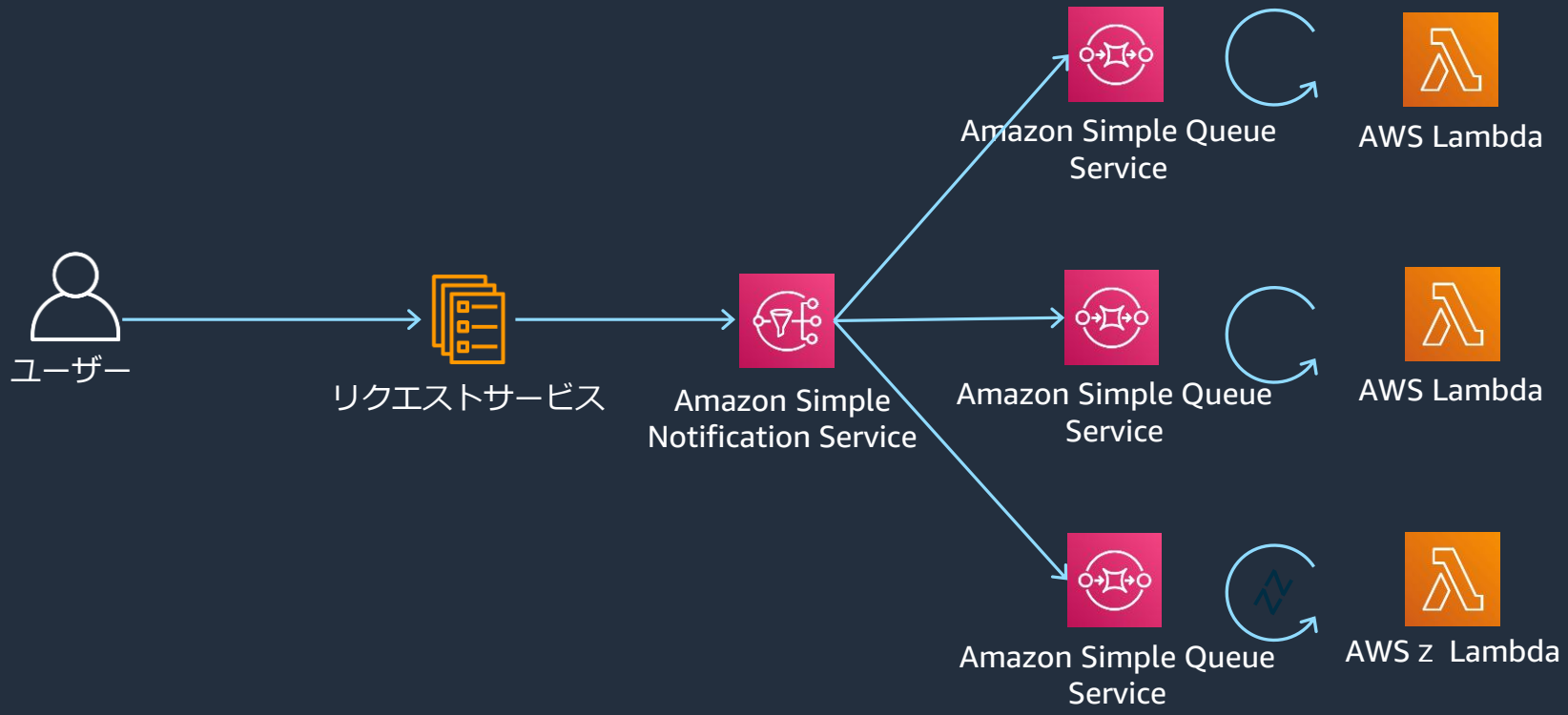
- 必要なすべての情報を含んだ最初のイベントを 1 つのメッセージに保存して、最初のトランザクションを完了
- 他のサービスがそのメッセージを非同期的に取得し、それぞれのタスクを完了させる
- サービスが疎結合になり、直接互いに影響を与えない
- メッセージの保存と取得が非同期の関係になり、スケーラビリティと信頼性が向上

aws

コレオグラフィーパターン

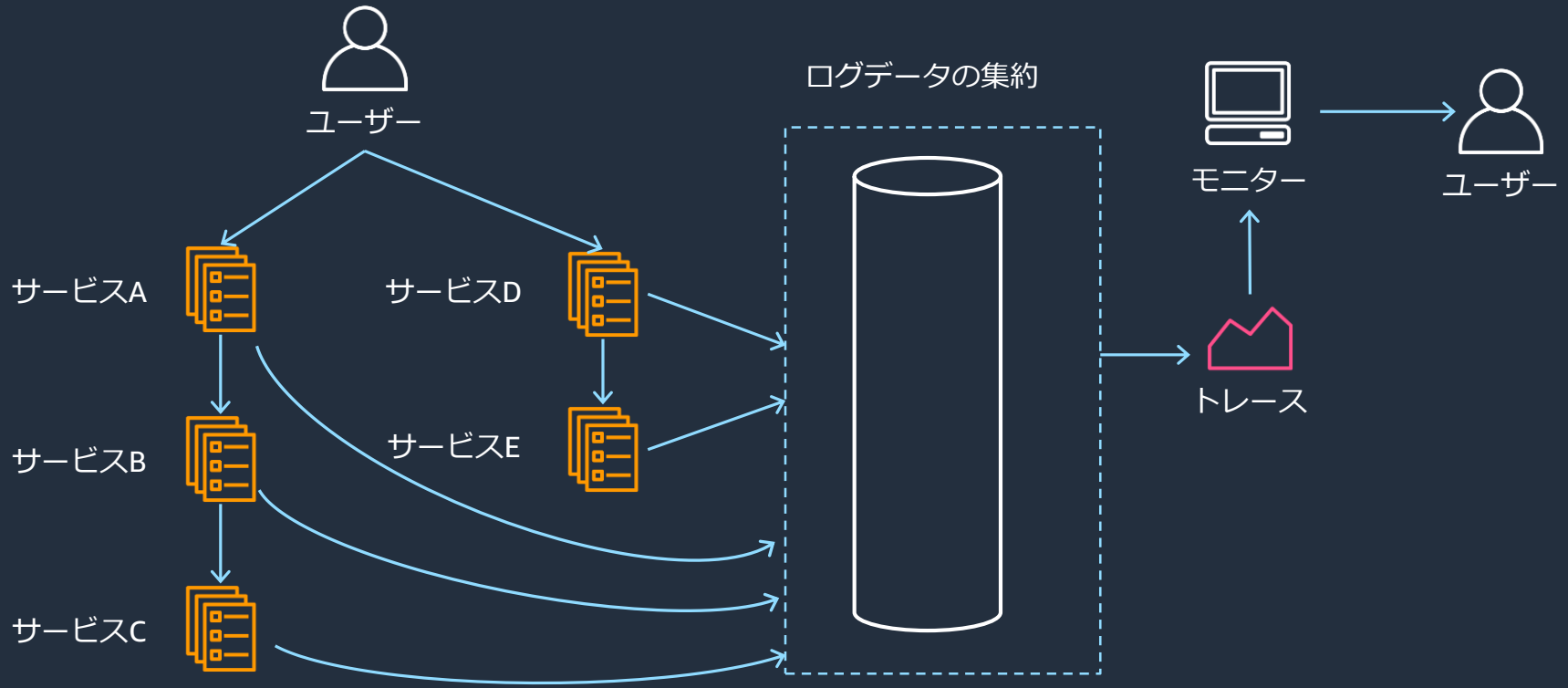


Amazon SNS、Amazon SQS、AWS Lambdaの利用

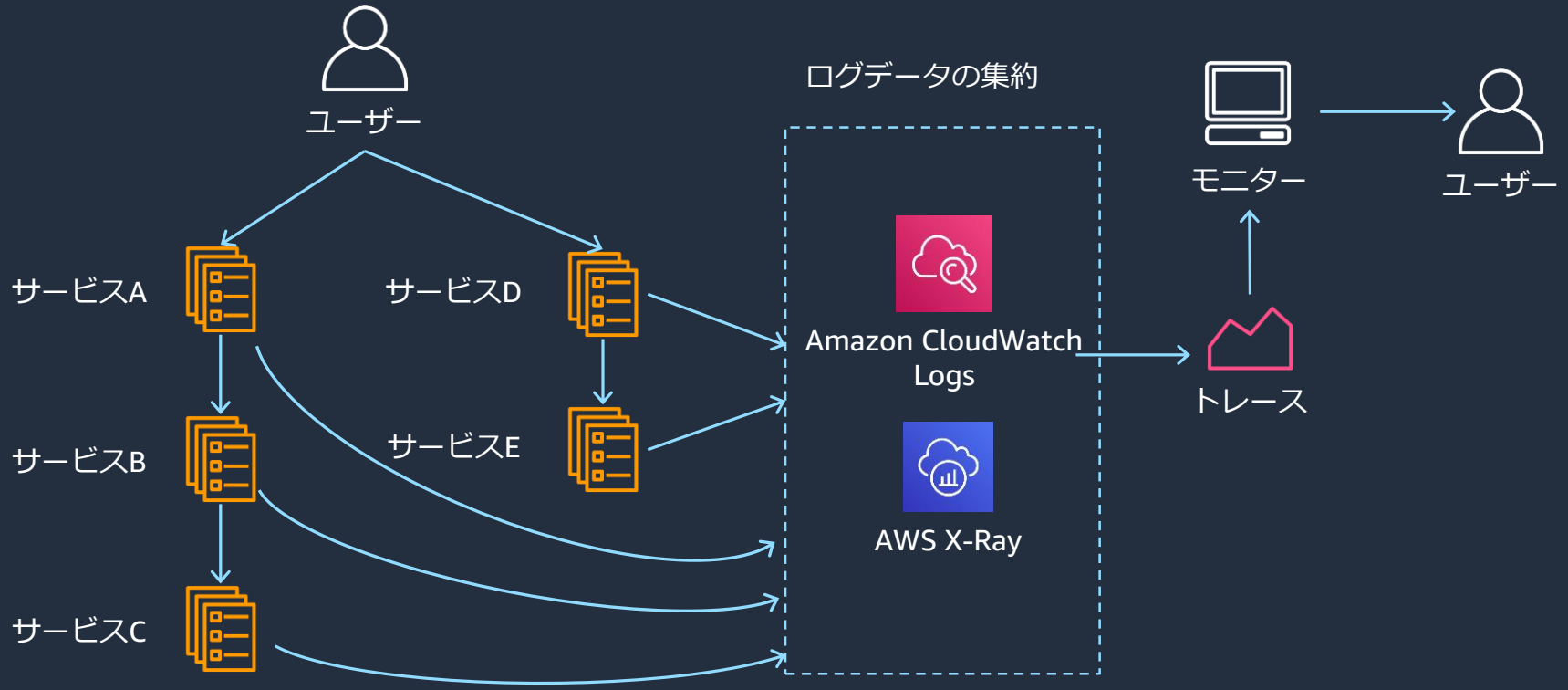


集約ログパターン

集約ログパターン



集約ログパターン



まとめ

- デザインパターンは先人の知恵を共有する仕組み
- パターンの適用によって問題をうまく解決できるか検討する
    - 但し、乱用に注意
        - No Silver Bullet
          銀の弾丸はない
        - If all you have is a hammer, Everything looks like a nail
          ハンマーを持つと全てが釘に見える

- AWSは多数のアプリケーションブロックを提供
    - アプリケーションブロックをうまく活用することで、素早く構築

- フルマネージドなサービスを利用することで運用負荷も軽減

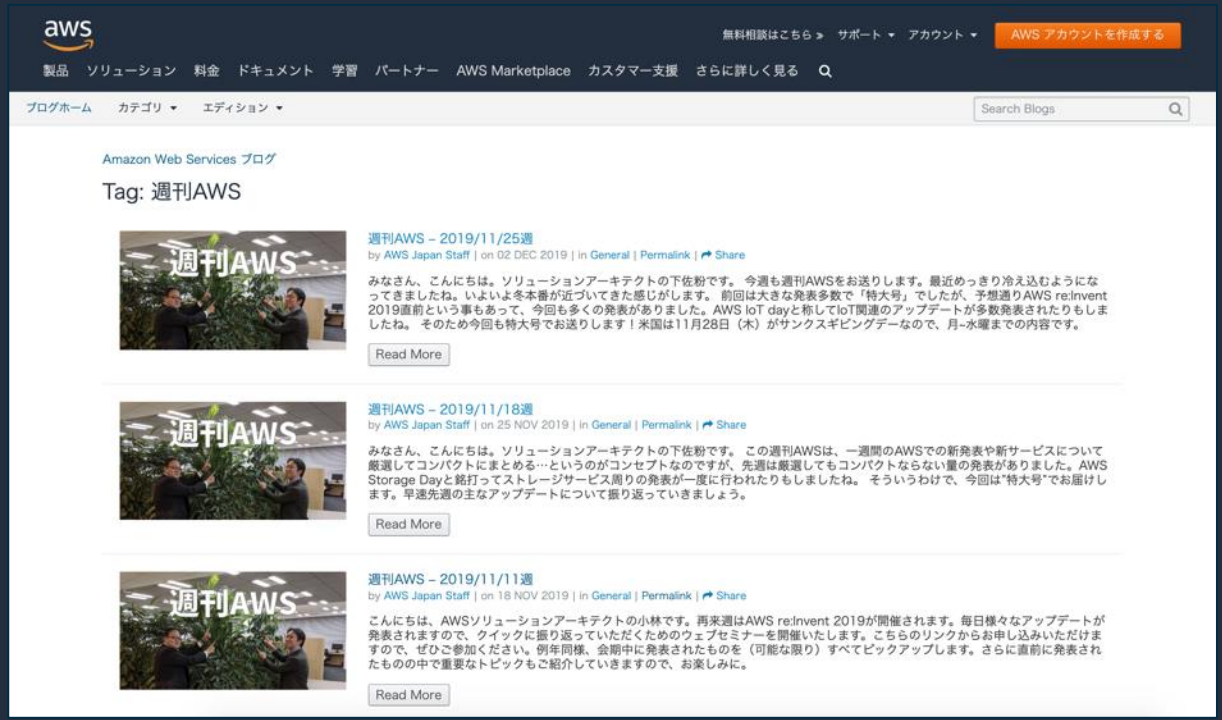
Q&A

お答えできなかったご質問については
AWS Japan Blog 「https://aws.amazon.com/jp/blogs/news/」にて
後日掲載します。

AWS の日本語資料の場所「AWS 資料」で検索

https://amzn.to/JPArchive

日々のアップデートチェックは週刊AWSで



週刊AWS で[検索]

AWS Well-Architected 個別技術相談会

毎週”W-A個別技術相談会”を実施中

- AWSのソリューションアーキテクト(SA)に対策などを相談することも可能
- 申込みはイベント告知サイトから

(https://aws.amazon.com/jp/about-aws/events/)

AWS イベント で[検索]

ご視聴ありがとうございました

AWS 公式 Webinar
https://amzn.to/JPWebinar

過去資料
https://amzn.to/JPArchive