

## 目次

まえがき	3
AWS認定について	5
AWS認定デベロッパー・アソシエイトについて	7
本書の活用方法	10
学習の進め方・参考教材	13
<b>第1章 AWSの概要・基礎・前提知識</b>	<b>19</b>
<b>1-1 AWSの概要・基礎・前提知識</b>	<b>20</b>
<b>1-2 AWS デベロッパー・アソシエイト試験で出題頻度が高いサービス</b>	<b>24</b>
<b>1-3 AWS開発の基本となるサービス</b>	<b>27</b>
<b>第2章 開発関連サービス</b>	<b>43</b>
<b>2-1 Amazon API Gateway</b>	<b>44</b>
<b>2-2 AWS Lambda</b>	<b>59</b>
<b>2-3 Amazon DynamoDB</b>	<b>71</b>
<b>2-4 AWS Step Functions</b>	<b>85</b>
<b>2-5 Amazon Route 53</b>	<b>94</b>
<b>2-6 ELB (Elastic Load Balancer)</b>	<b>103</b>
<b>2-7 Amazon ECS</b>	<b>115</b>
<b>2-8 Amazon RDS</b>	<b>126</b>
<b>2-9 Amazon S3</b>	<b>134</b>
<b>2-10 Amazon Kinesis</b>	<b>153</b>
<b>2-11 他の開発関連サービス</b>	<b>165</b>
演習問題	168
<b>第3章 セキュリティ関連サービス</b>	<b>205</b>
<b>3-1 AWS IAM (Identity and Access Management)</b>	<b>206</b>
<b>3-2 AWS STS (Security Token Service)</b>	<b>227</b>
<b>3-3 AWS KMS (Key Management Service)</b>	<b>234</b>
<b>3-4 Amazon Cognito</b>	<b>241</b>
<b>3-5 他のセキュリティ関連サービス</b>	<b>249</b>
演習問題	258

## 第4章 展開関連サービス ..... 279

<b>4-1 AWS CodeCommit</b>	<b>280</b>
<b>4-2 Amazon ECR</b>	<b>286</b>
<b>4-3 AWS CodeBuild</b>	<b>288</b>
<b>4-4 AWS CodePipeline</b>	<b>293</b>
<b>4-5 AWS CodeDeploy</b>	<b>299</b>
<b>4-6 AWS CloudFormation</b>	<b>306</b>
<b>4-7 AWS Serverless Application Model</b>	<b>317</b>
<b>4-8 AWS Elastic Beanstalk</b>	<b>323</b>
<b>4-9 他の展開関連サービス</b>	<b>328</b>
演習問題	330

## 第5章 トラブルシューティングと最適化に関するサービス ..... 359

<b>5-1 Amazon CloudWatch</b>	<b>360</b>
<b>5-2 AWS CloudTrail</b>	<b>377</b>
<b>5-3 AWS X-Ray</b>	<b>383</b>
<b>5-4 Amazon ElastiCache</b>	<b>393</b>
<b>5-5 Amazon SQS</b>	<b>400</b>
<b>5-6 Amazon SNS</b>	<b>411</b>
<b>5-7 Amazon CloudFront</b>	<b>416</b>
<b>5-8 Auto Scaling</b>	<b>429</b>
演習問題	438

## 第6章 総仕上げ問題 ..... 463

<b>6-1 問題</b>	<b>464</b>
<b>6-2 解答と解説</b>	<b>495</b>

索引 ..... 512

# 第1章

## AWSの概要・基礎・前提知識

1-1. AWSの概要・基礎・前提知識

1-2. AWS Developer Associate試験で出題頻度が高いサービス

1-3. AWS開発の基本となるサービス

## 1-1 AWSの概要・基礎・前提知識

本節では、試験の学習を進める前に知っておくべきAWSの概要・基礎・前提知識について説明します。

### 1 AWSが提供するクラウドコンピューティングサービスのリージョン

AWSは、パブリッククラウド市場で世界No.1のシェアを持つ、世界最大級のクラウドコンピューティングサービスです。まず、AWSが提供するクラウドコンピューティングサービスについて見ていきましょう。2024年1月現在、32の地域（リージョン）にクラウドサービスを展開しています。

#### 【AWSリージョン<sup>\*1</sup>】



まず前提として注意したいのは、利用者は居住する地域がどこであろうと、インターネットにつながってさえいれば、AWSのどのリージョンでも利用できるということです。リージョンはあくまで、クラウドサービスを構築する環境・場所を選ぶための基本的な単位でしかありません。

例えば、日本でWebサイトを構築する場合、AWSユーザは一般的にサービスを提供するリージョンとして「東京」または「大阪」を選択します。特別な理由がなければ、米国や欧州といった別のリージョンを選ぶことはありません。サービスを最終的に利用するユーザ（エンドユーザ）が最も高速にアクセスできる（低レイテンシ：遅延が少ない）リージョンを指定するのが普通です。

<sup>\*1</sup> <https://aws.amazon.com/jp/about-aws/global-infrastructure/>

リージョンは各々を識別するリージョンコードを持っていて、AWSを利用する際は常にリージョンを意識する必要があります。また、高度な可用性が求められるシステムでは、災害対策として複数のリージョンでのシステムを構成する場合があります。



マルチリージョン構成の注意点として、リージョンを分けた構成には、少々やっかいな面もあります。リージョンごとにシステムやデータは独立して存在することになるからです。

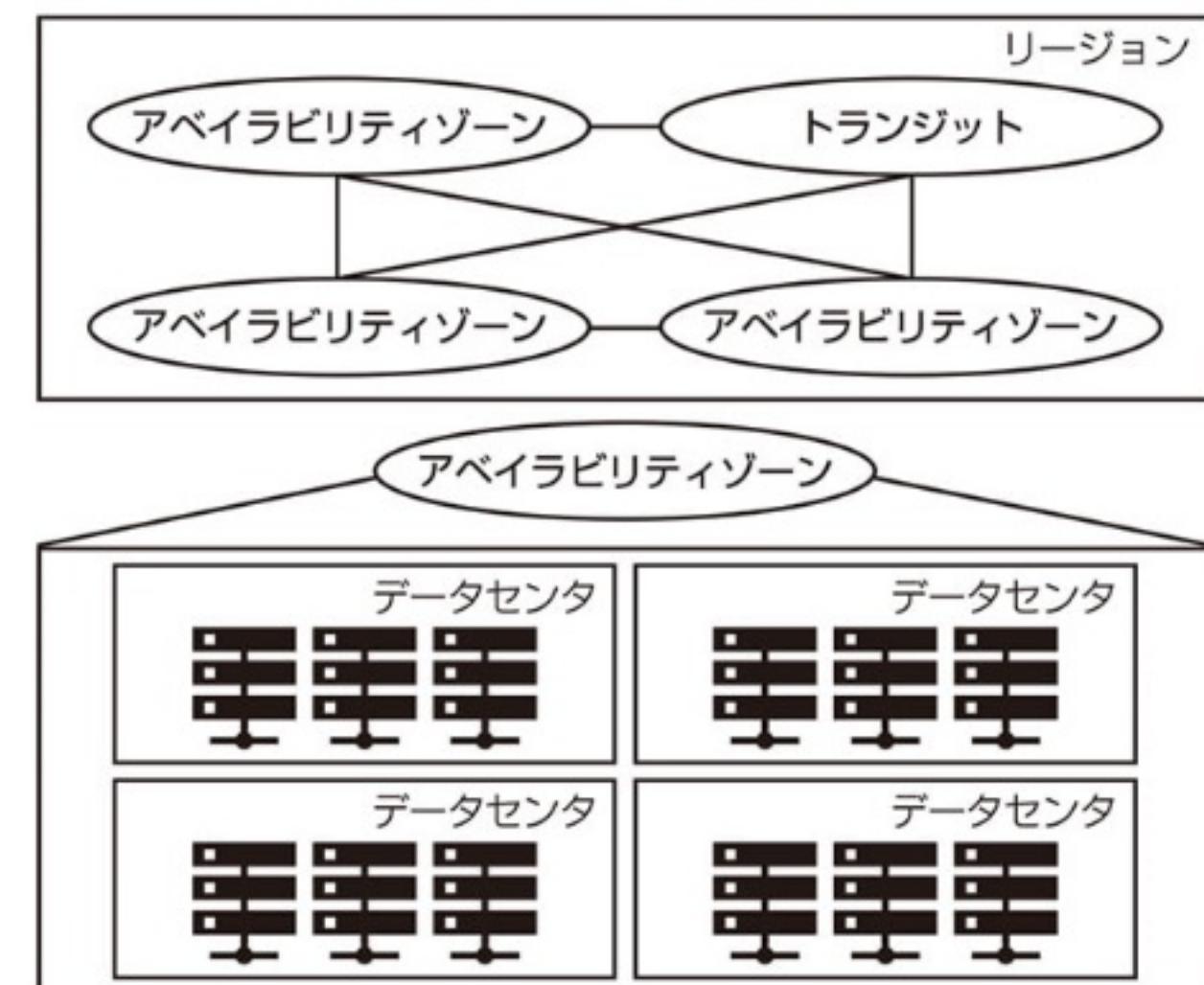
あるエンドユーザが、同じサービスを日本と米国で利用していたとしましょう。リージョンを分けた構成にしていると、日本でできていたことが、渡米するとできなくなるといったケースも起こり得るので注意が必要です。Amazon.comとAmazon.co.jpが顕著な例ですが、Amazonの場合はアクセスするURLのドメインを明確に分離しているので、それほど大きな問題は発生しません。

複数のリージョンを利用する際は、この点を留意しておく必要があります。AWSの一部のマネジドサービス（S3やDynamoDBなど）で、各リージョンでデータを同期するクロスリージョンレプリケーション機能をサポートしているので、そのような用途を想定した機能があることを知っておきましょう。

### 2 AWSが展開するアベイラビリティゾーン

続いて、リージョンの中身を詳しく見ていきます。AWSの各リージョンは、複数のアベイラビリティゾーンおよびトランジットで構成されています。

#### 【アベイラビリティゾーンの構成】



アベイラビリティゾーン自体も多数のサーバからなる、1つ以上のデータセンタから構成されており、アベイラビリティゾーン間は複数の冗長化されたネットワークで接続されています。データセンタの物理的なロケーションは、被災を考慮して隔離しつつも、相互の通信遅延が2ms（ミリ秒）未満となるよう、100km以内で立地しています。

アベイラビリティゾーンからほかのAWSリージョンや企業のオンプレミス環境、インターネットへ接続を行う場合は、トランジットと呼ばれるデータセンタを経由して行われます。データセンタ内部では、多数のサーバやネットワーク機器がさまざまな対策を施された上で稼働しています。

防犯上の観点からデータセンタの位置は非公開となっています。また、公式サイトではAWSが実施しているさまざまな被災、障害、セキュリティ対策が公開されています<sup>※2</sup>。

### 3 AWSが展開するエッジロケーションとリージョンエッジキャッシュ

AWSでは、上述のリージョンに加え、エンドユーザからのアクセス速度向上のため、42カ国84都市にある216箇所にエッジロケーションおよびリージョンエッジキャッシュと呼ばれるデータセンタを構築しています。

日本では東京リージョンおよび大阪リージョンに複数配置されています。これらはCloudFrontなどのコンテンツ配信ネットワーク（CDN：Content Delivery Network）サービスに利用されており、ユーザの物理的な場所に応じて、最も早く接続可能なデータセンタにアクセスするよう構成されています。

#### 【エッジロケーション<sup>※3</sup>】



<sup>※2</sup> <https://aws.amazon.com/jp/compliance/data-center/data-centers/>

<sup>※3</sup> <https://aws.amazon.com/jp/cloudfront/features/>

### 4 AWSの利用方法

AWSのクラウド環境やサービスを利用するには、まずAWSのサイト上でアカウントを作成します。アカウント作成には連絡先や支払い情報、本人確認が必要になりますが、作成自体は無料でできます。また、作成から1年間、限定された条件のもとでEC2やS3といった60以上のサービスを無料で使用できます。

AWSのサービスは、以下の3つの方法で利用します。

方法	概要
AWSマネジメントコンソール	AWSの各種サービスを実行・利用するWebベースのGUI。AWSのサイトからログインできます。幅広いユーザーに一般的によく利用される方法で、視覚的にわかりやすいため、サービス運用状況の確認などでも利用されます。
AWS CLI (Command Line Interface)	自身のパソコン端末にあるターミナルなどのコマンドラインを実行し、各サービスを実行・利用する方法です。AWS上に展開するアプリケーションを開発する場合など、高速に、繰り返しサービスを起動・実行・停止する用途に適しています。
AWS SDK (Software Development Kit)	主にアプリケーションから利用されることを想定した、AWSが提供するSDKと呼ばれるライブラリを使ったアクセス方法です。2023年12月時点では、C++、Go、Java、JavaScript、Kotlin、.NET、Node.js、PHP、Python、Ruby、Rust、Swift、SAP ABAPといった13種類のプログラミング言語にそれぞれSDKが提供され、開発したアプリケーションの処理の中でSDKのライブラリをコールしてAWSサービスへアクセスできるようになっています。

AWS CLIやSDKを利用する際は、上記のアカウントに関する設定を済ませた後、マネジメントコンソール上からアプリケーション開発者用ユーザーを作成し、認証情報（アクセスキーとシークレットキー）を払い出します。CLIやSDKはこの認証情報があれば、コマンドやライブラリでキーを指定して実行することでAWSのサービスのAPIを実行できるようになります。

## 1-2

## AWS デベロッパー - アソシエイト試験で出題頻度が高いサービス

本節では、AWS デベロッパー - アソシエイト認定試験で取り上げられることが多いサービスについて解説します。試験では、ユースケースにもとづき、適切なサービスはどれかを問う問題と、サービスの適切な記述や使い方を問う問題があります。まずは各サービスの概要や特徴を押さえ、ワークフローや要件に応じた適切なサービスを選択できるようになります。

### 1 試験の出題頻度が高いサービス一覧と概要

試験の出題頻度が高く、詳細を押さえておきたいサービスは以下のとおりです。

#### 【サービス一覧】

カテゴリ	サービス	説明
コンピューティング	EC2	任意のOSで仮想的にサーバ環境を構築・利用できるサービスです。
	ECR	Dockerコンテナイメージを管理するマネージドレジストリサービスです。
	ECS	マネージドコンテナオーケストレーションサービスです。
	Elastic Beanstalk	アプリケーションを自動でデプロイするサービスです。
	Lambda	定義したアプリケーションコードをサーバなしで実行するサービスです。
ネットワーキング	API Gateway	APIの作成、公開を行うサービスです。
	CloudFront	静的コンテンツキャッシングなどを実現するCDN (Content Delivery Network : コンテンツ配信ネットワーク) サービスです。
	ELB	ALB (Application Load Balancer)、CLB (Classic Load Balancer)、NLB (Network Load Balancer) の3種類をオプションで選択できるマネージドロードバランシングサービスです。
	Route 53	ドメインの登録、DNSルーティング、ヘルスチェックなどを行うDNSサービスです。
	VPC	AWS内にプライベートな仮想ネットワークやサブネットを構築するサービスです。
ストレージ	S3	ログ保存、バックアップ、静的ホスティング等多様な用途のオンラインストレージサービスです。

カテゴリ	サービス	説明
データベース	DynamoDB	スケーラブルな特性を持つ、NoSQLスキーマレスデータベースです。
	ElastiCache	オープンソースソフトウェアのMemcachedおよびRedisを利用した、キャッシング向けデータベースです。
	RDS	オープンソースおよびプロプライエタリのマネージドリレーショナルデータベースです。
アプリケーション統合	SNS	スケーラブルなプッシュ配信型メッセージ送信サービスです。
	Step Functions	AWS Lambdaで定義した関数をワークフロー式に実行するためのサービスです。
	SQS	スケーラブルな完全マネージド型メッセージキューサービスです。
分析	Kinesis	ストリーミングデータをリアルタイムで収集、処理、分析するマネージドサービスです。データストアにストリームデータをロードするKinesis Data Firehose、データストリームのキャッシング、処理、保存をするKinesis Data Streams、SQL や Java でデータストリームを分析するKinesis Data Analytics、ストリーミング動画のキャッシング、処理、保存をするKinesis Video Streamsがあります。
開発者ツール	CodeBuild	クラウド環境で継続的インテグレーションを実現するためのビルドサービスです。
	CodeCommit	マネージドGitリポジトリサービスです。
	CodeDeploy	EC2インスタンスやLambda、ECSなどへのアプリケーションデプロイを自動化するサービスです。
	CodePipeline	コードのビルド、テスト、デプロイまでのワークフローをパイプラインとして定義・実行し、継続的デリバリーを実現するサービスです。
	X-Ray	分散されたアプリケーションのデータを収集し、可視化・分析するサービスです。
マネジメントとガバナンス	Auto Scaling	負荷に応じて自動的にリソースのスケールアウト・スケールインを行うサービスです。
	CLI	コマンドラインからAWS APIを実行するツールです。
	Cloud Formation	テンプレートをもとに、AWSリソース基盤を自動構築するIaC (Infrastructure as Code) ツールです。
	CloudTrail	マネジメントコンソールでの操作とAWS APIコールを記録するサービスです。
	CloudWatch	AWSリソースのさまざまなメトリクスやログを収集・可視化するサービスです。
	SAM	サーバレスアプリケーションを簡単に実装するためのCLIおよびオープンソースフレームワークです。CloudFormationを拡張したSAMテンプレートを作成することで、API GatewayやLambda、DynamoDBなどの環境を簡易構築できます。

カテゴリ	サービス	説明
セキュリティ	IAM	AWSアカウント内でユーザ、グループ、ロールなどを作成し、各サービスやリソースに対するアクセス管理を行うサービスです。
	KMS	データを暗号化するキーを作成・管理するマネージドサービスです。
	Cognito	ユーザディレクトリおよび認証・認可サービスです。
	STS	AWSのサービスへのアクセスに使用できる一時的な限定権限認証情報を取得するサービスです。

以降の章では、前提知識として知っておきたい基本的なものから順に上記のサービスを解説します。知識が足りていない分野・サービスに集中して学習を進めるのもよい方法です。

## 1-3 AWS開発の基本となるサービス

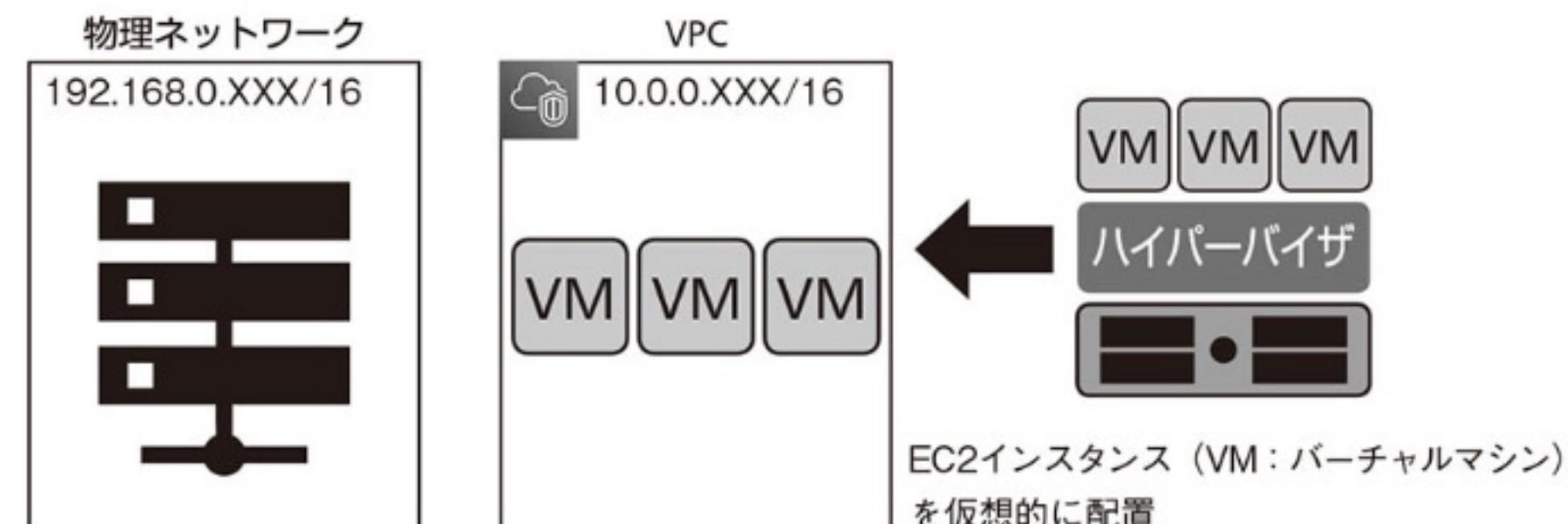
本節では、AWSを使って開発する上で基本となるサービスについて説明します。AWSの基礎知識と用語を押さえ、試験問題の意味を確実に理解できるようにしておきましょう。

### 1 VPC・サブネット

VPCは、仮想プライベートネットワークを構築できるサービスです。ここでの「仮想」とは、実際のサーバ機器が手元にない状態でネットワークを構築することを指します。

具体的には、仮想イメージのサーバ「EC2」やデータベース「RDS」などのAWSリソースに、構築した仮想プライベートネットワーク上のIPアドレスを割り当て、仮想的にそのアドレスにあるように配置する……といったような使い方をします。この仕組みにより、ネットワークおよびサーバを含むシステム環境を、オンデマンドかつ迅速に構築できます。

#### 【VPCと物理ネットワークの違い】



VPCは、AWSのリージョン単位に作成でき、XXX.XXX.XXX.XXX/16～28であるCIDR表記に沿ってネットワークを構築することができます。構築したVPCでは、セグメントに相当する複数のサブネットを定義して、用途に応じて柔軟にネットワークを構成でき、アクセス制御が行えます。

そのほか、VPCやサブネットは以下のようないくつかの特徴があります。

- ・VPCは複数のアベイラビリティゾーンをまたいで構築できますが、サブネットはアベイラビリティゾーンごとに作成する必要があります。

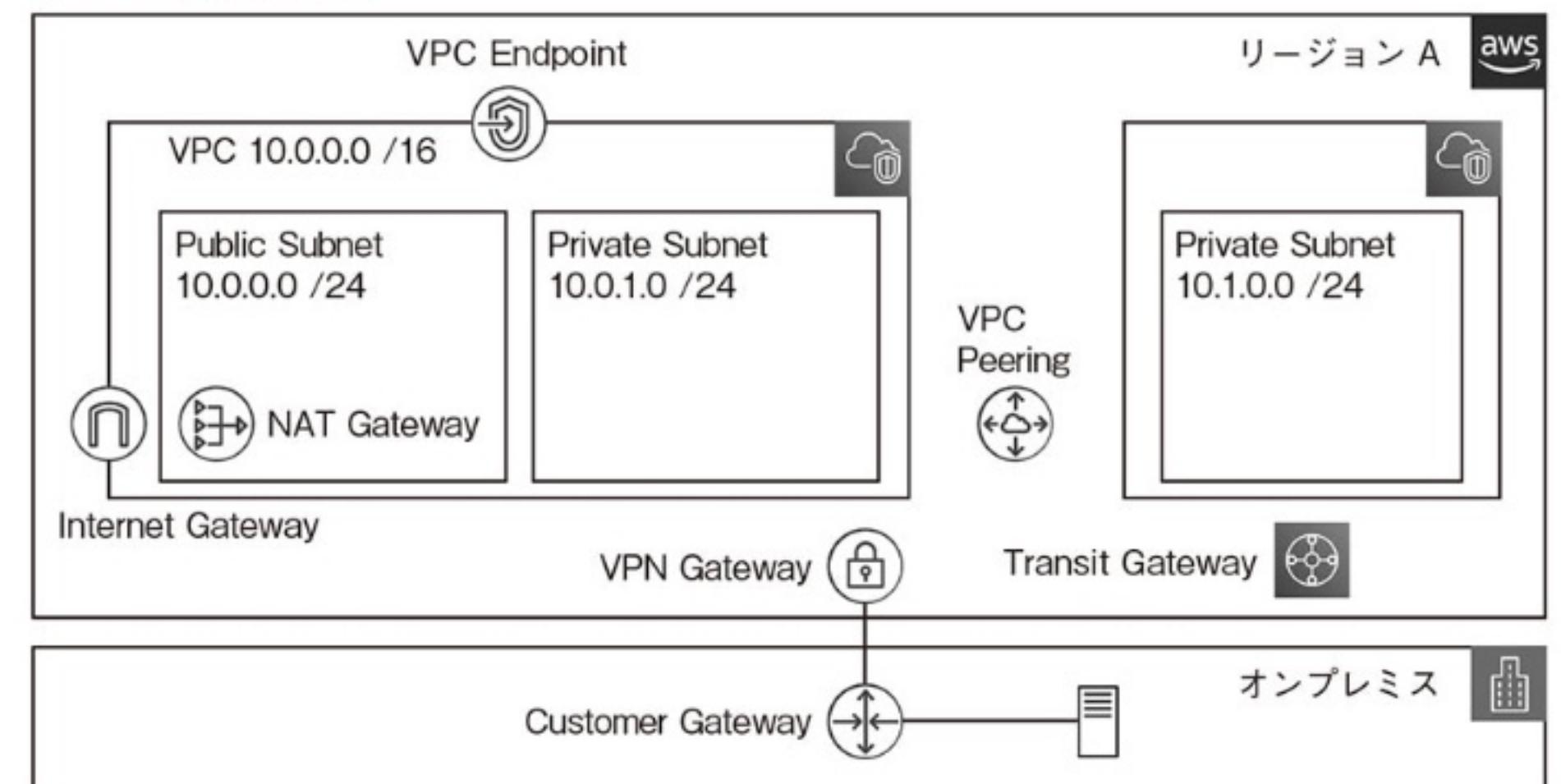
- VPC内のアドレスはCIDRが/16から/28の間で使用できます。ただし、各サブネットの最初の4アドレス、最後の1アドレスは予約されているため使用できません。
  - XXX.XXX.XXX.0 (ネットワークアドレス用)
  - XXX.XXX.XXX.1 (VPCルーター)
  - XXX.XXX.XXX.2 (Amazon Provided DNS : Route 53 Resolver)
  - XXX.XXX.XXX.3 (予備)
  - XXX.XXX.XXX.255 (プロードキャストはサポートしていないが予約)
- CIDRは一度作成すると変更できませんでしたが、2017年10月より、VPCに割り当てるCIDRを拡張できるようになりました。
- サブネットは外部との接続を想定したパブリックサブネットと、リソースやデータを外部アクセスから保護する用途を想定したプライベートサブネットがあります。パブリックサブネットではデフォルトでトラフィックを外部に送信できますが、プライベートサブネットはできません。
- 複数のアカウントでVPCおよびその中のリソースを共有できるVPC Sharing<sup>※4</sup> をオプションとして選択することもできます。

## 2 VPCの外部通信

VPCでは、複数のVPC間で通信することはもちろん、オンプレミスとの接続サービスであるDirect ConnectやVPNを通じて、オンプレミス環境にあるサーバとも通信できます。VPC内のリソースが外部通信する際は、下記に挙げるようなネットワークゲートウェイをVPCに設置します。

- インターネットゲートウェイ (Internet Gateway)
- NATゲートウェイ (NAT Gateway)
- 仮想プライベートゲートウェイ (VPN Gateway)
- カスタマーゲートウェイ (Customer Gateway)
- VPCピアリング接続 (VPC Peering)
- VPCエンドポイント (VPC Endpoint)
- トランジットゲートウェイ (Transit Gateway)

### 【VPCの外部接続】



#### ●インターネットゲートウェイ (Internet Gateway)

インターネットゲートウェイは、文字どおりVPCとインターネットの通信用途で設置されるネットワークゲートウェイです。ただし、実際に通信を行うにはそれに加えて、通信元のリソースがパブリックIPアドレスを持っている必要があります。AWSで付与できるパブリックアドレスは動的に割り当てられるものと、固定的に割り当てられるものがあり、後者を「Elastic IPアドレス」と呼びます。

インターネットゲートウェイは定義としては1つの機器のように扱われますが、内部的に冗長化されており、通信量が増加した際には自動的にスケーリングされます。

#### ●NATゲートウェイ (NAT Gateway)

NATはNetwork Address Translation (ネットワークアドレス変換) の略であり、プライベートサブネット内でインターネットへの通信を行いたい場合にパブリックサブネットへ設置する中継用ゲートウェイです。NATゲートウェイを介した通信は单方向通信となります。プライベートサブネットからインターネットへ通信することはできますが、インターネットからプライベートサブネット内へ接続することはできません。

NATゲートウェイは2015年ごろに登場したNATのマネージドサービスであり、冗長化構成がサポートされます。それまでは自身で同じ用途でEC2を利用してNATインスタンスを立てる必要がありました。現在も同じくNATインスタンスを使用する方法も可能ですが、障害発生を考慮した構成など構築の手間が煩雑になります。外部接続は許可するが特定のサービス通信しか許容しないといったような特別な制約がない限り、NATゲートウェイを使用します。

※4 [https://docs.aws.amazon.com/ja\\_jp/vpc/latest/userguide/vpc-sharing.html](https://docs.aws.amazon.com/ja_jp/vpc/latest/userguide/vpc-sharing.html)

## ●仮想プライベートゲートウェイ (VPN Gateway)

仮想プライベートゲートウェイは、VPCとオンプレミス環境を接続するためにAWS内に作成するネットワークゲートウェイです。インターネットゲートウェイと同様、冗長化構成をとります。VPCとオンプレミスを接続するには大きく2つの方法があり、専用線を用いるDirect Connectと、インターネット回線をベースとしたVPN接続があります。

VPN接続では、IPアドレスの動的ルーティングと静的ルーティングを双方ともサポートしますが、どちらの場合も次で説明するカスタマーゲートウェイが必要になります。

## ●カスタマーゲートウェイ (Customer Gateway)

AWSとのVPN接続において、オンプレミス側に配置するゲートウェイです。IPアドレスを動的に割り当てるルーティングではBGP (Border Gateway Protocol)と呼ばれるピア接続を利用して、オンプレミスと接続します。

あらかじめ決められたIPアドレスを使用する静的ルーティングを利用する場合は、オンプレミス環境では、通信が行えるネットワーク物理機器と、接続を許可する固定のパブリックIPアドレスが必要になります。実際に配置されるネットワーク物理機器はカスタマーゲートウェイデバイスと呼びます。オンプレミス側へ機器を設置した後に、仮想プライベートゲートウェイと接続するためのカスタマーゲートウェイ設定をAWSコンソール上から行う必要があります。

## ●VPC ピアリング接続 (VPC Peering)

VPCピアリング接続は、独立した2つのVPCを接続し、プライベートアドレスを使って相互に通信します。こちらもインターネットゲートウェイ、仮想プライベートゲートウェイと同様、冗長化されたかたちで構築されます。この接続では自分が所有しているVPCだけではなく、同じリージョン内にある別アカウントのVPCとも接続が可能です。ただし、接続できるのはピアリングしたVPCの範囲内に限られ、接続先VPCが別途接続しているネットワークとは接続できません。



VPCは1リージョン内に構築されます。リージョンをまたいで別のVPCと通信したい場合は、一部のリージョン同士でVPC間の相互接続が可能になる「インターリージョンVPC」を使います。

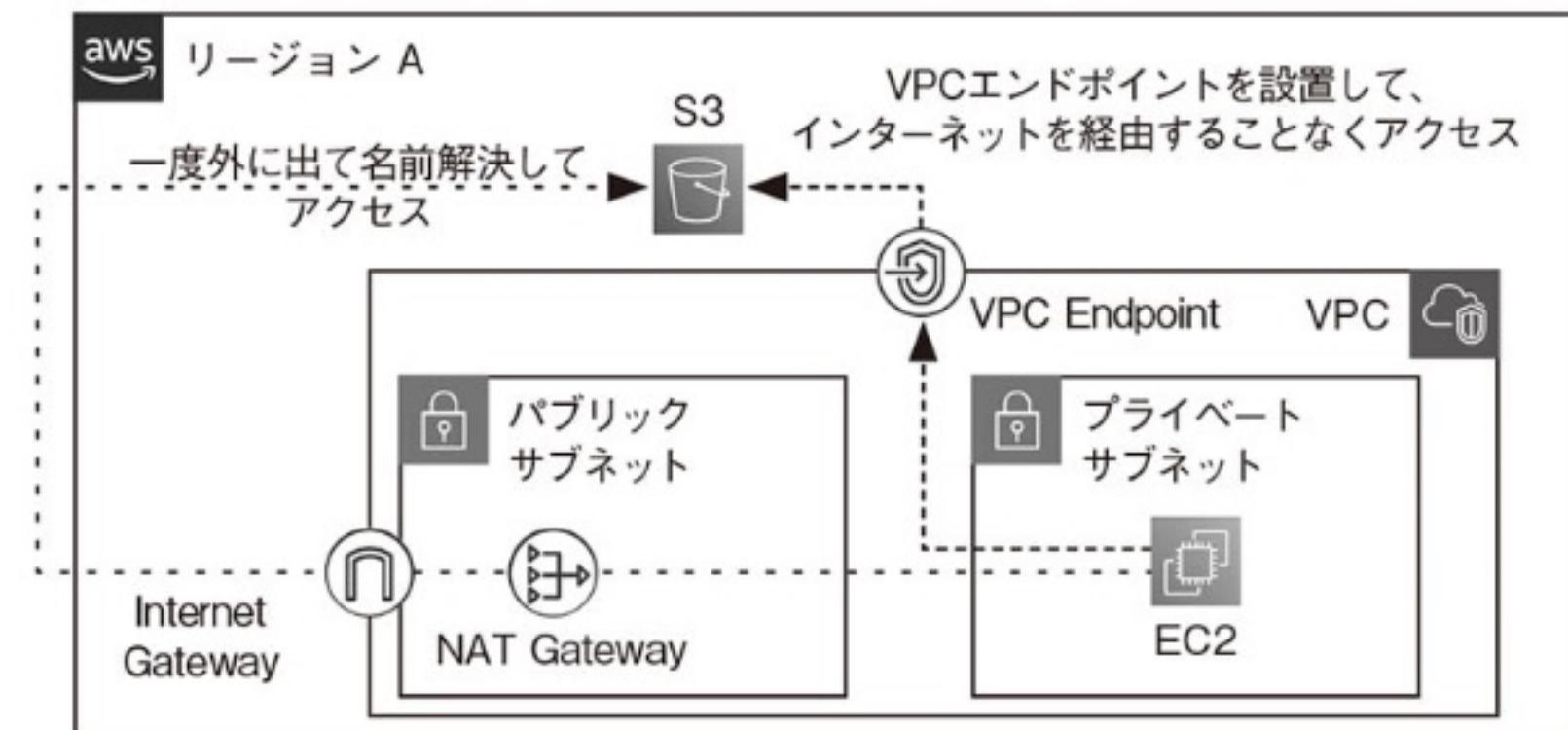
## ●VPC エンドポイント (VPC Endpoint)

VPCエンドポイントはVPCの中からAWSのマネージドサービスにアクセスするためのサービスです。AWSのマネージドサービスには、EC2やRDSのようにVPC・サブネット内に構築されるリソースと、S3やDynamoDB、SQSのようにリージョン単位で提供されるリソース（ここではリージョンサービスと呼びます）が

あります。

リージョンサービスは当然VPCの外にあるAWSネットワーク内に構築されているため、アクセスするにはVPCから外部へ出るかたちになります。S3やDynamoDBなどはデフォルトで、インターネット経由のアクセスが可能なので、インターネットゲートウェイやNATゲートウェイの設定が実行されていれば通信できます。ただし、インターネットを経由することなくアクセスしたい場合は、VPCエンドポイントを設置する必要があります。

### 【VPCエンドポイントを使ったAWSリソースへのアクセス】



VPCエンドポイントはゲートウェイ型とインターフェース型の2種類があり、設定やアクセス制御の方法が異なります。

### 【VPCエンドポイント】

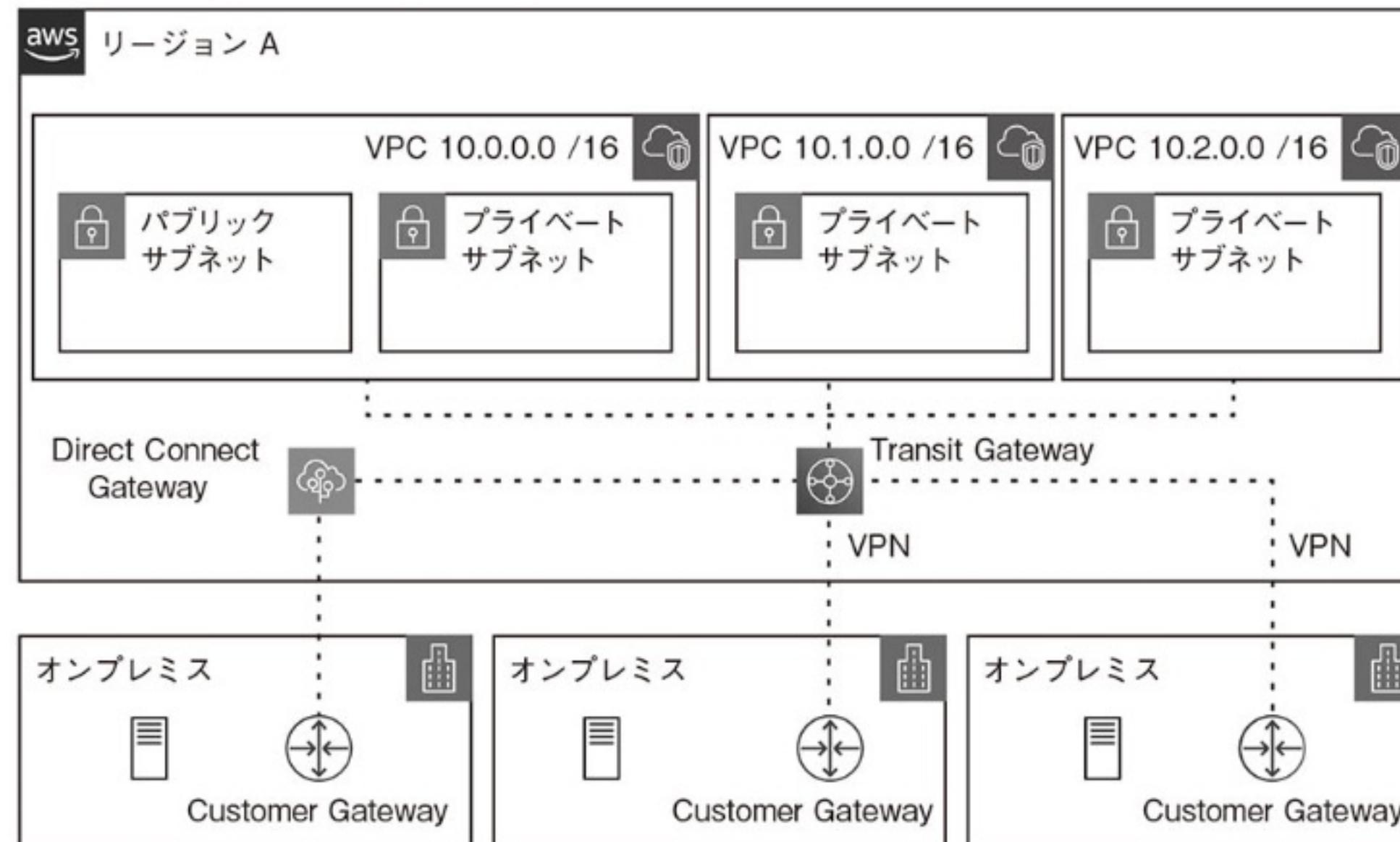
ゲートウェイ型	インターフェース型
ゲートウェイ型は、インターネットゲートウェイやNATゲートウェイと同様、「ルートテーブル」（次項参照）への設定が必要なエンドポイントです。2023年12月時点では、リージョンサービスであるS3とDynamoDBがこのタイプで提供されています。	AWS PrivateLink <sup>※5</sup> を利用したインターフェース型のエンドポイントです。ゲートウェイ型であるS3やDynamoDBを除く、多くのサービスがこのタイプとなります。VPCエンドポイントを使用したいリソースがあるサブネットに直接アタッチし、許可する接続をセキュリティグループにて制御します。

※5 <https://aws.amazon.com/jp/privatelink/>

## ●トランジットゲートウェイ (Transit Gateway)

トランジットゲートウェイは、これまで説明してきたさまざまなVPC間の接続を統合するアグリゲーション型のネットワーク管理サービスです。下記の図のように、VPCピアリング接続が多数発生する場合や、オンプレミスとのDirect Connect接続やVPN接続が他拠点で発生する場合、接続設定を一元的に管理できます。

### 【トランジットゲートウェイ】



トランジットゲートウェイはVPCで設定するものとは別のルーティングテーブルを持っており、VPCピアリング接続において各VPC間の通信を制御します。多数のVPCでピアリング接続が複数発生する場合は、通常、各VPC間の接続ごとに設定を行う必要があります。ここでトランジットゲートウェイを利用すれば、一元的な設定により効率的な管理が可能です。

また、オンプレミスとの接続では、前述の仮想プライベートゲートウェイで通常どおり個別に接続していく場合よりも、AWSのデフォルトの上限設定を超えて多くの接続先を構築できます。なお、トランジットゲートウェイを利用した接続では、仮想プライベートゲートウェイを構築する必要はありません。

もう1つトランジットゲートウェイのメリットとして、VPCピアリングとオンプレミス、どちらの接続でもクロスアカウントアクセスが利用できる点が挙げられます（クロスアカウントアクセスは、あるアカウントのAWSリソースに別アカウントからのアクセスを許可する設定のこと）。マルチアカウントや多数の拠点・VPCを接続するユースケースで効果が高いサービスです。

## 3 VPCのアクセス制御

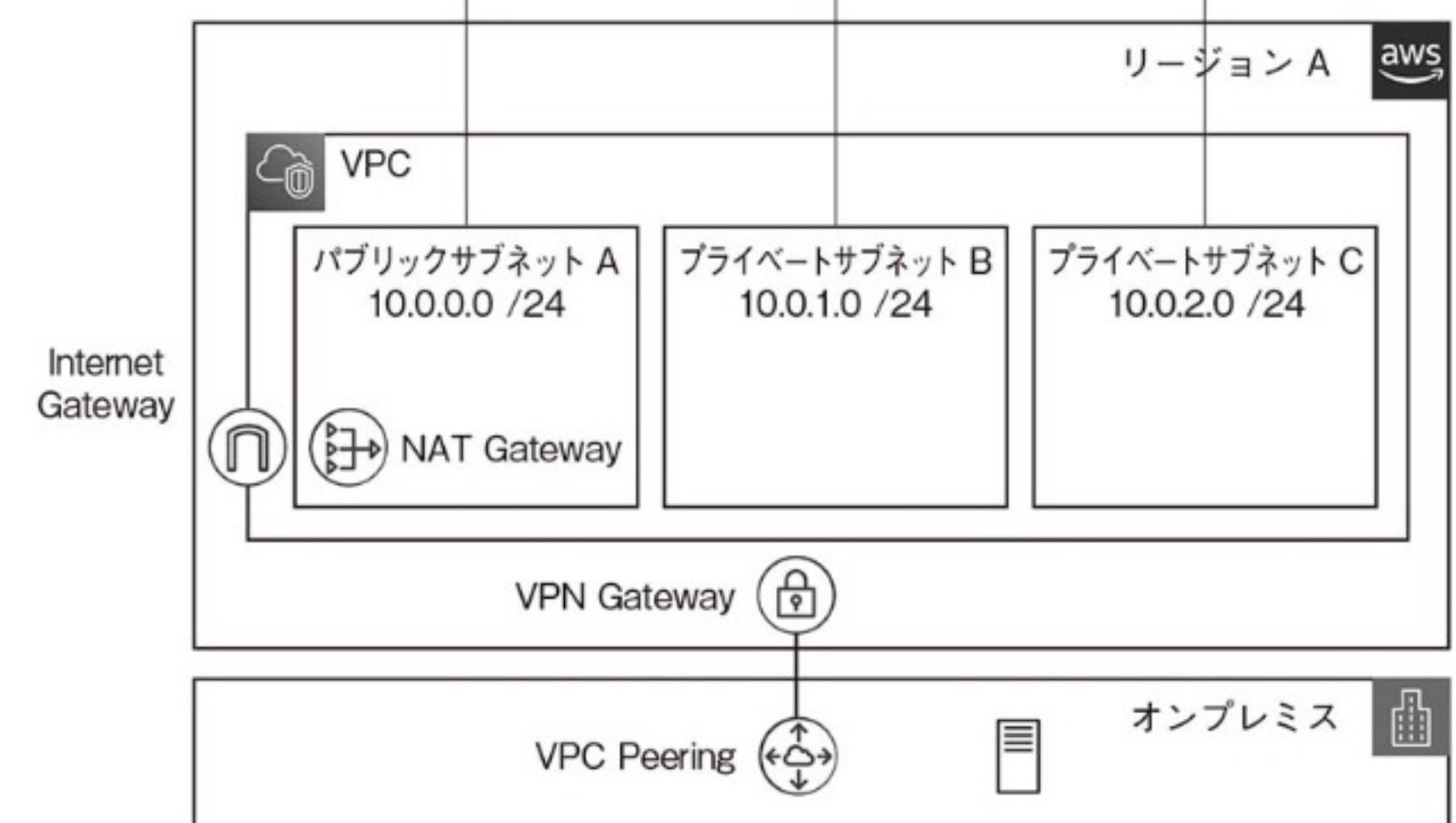
VPCにおける通信のルート設定・アクセス制御方法には、以下のようなものがあります。

### ●ルートテーブル

VPC内の通信では、指定されたCIDR表記のアドレスで通信をルーティングするルールセットを定義する必要があります。これを「ルートテーブル」といい、下記の図のようなかたちで表します。

#### 【ルートテーブルの作成・設定例】

カスタムルートテーブル		メインルートテーブル		カスタムルートテーブル	
10.0.0.0/16	Local	10.0.0.0/16	Local	10.0.0.0/16	Local
0.0.0.0/0	igw-yyyyyyyy	0.0.0.0/0	igw-wwwwwww	0.0.0.0/0	Vgw-zzzzzzzzzz



VPCを作成すると、同時にメインルートテーブルが自動で作成されます。ユーザーは通常、メインルートテーブルを編集するのではなく、カスタムルートテーブルを追加で作成することで、サブネット間のアクセス・通信制御を行います。

VPC内の各サブネットはなにかしらのルートテーブルに関連付けられる必要があります、サブネットが特定のルートテーブルに明示的に関連付けられていない場合、サブネットはメインルートテーブルに暗黙的に関連付けられます。

1つのサブネットに同時に複数のルートテーブルを関連付けることはできませんが、複数のサブネットを1つのルートテーブルに関連付けることは可能です。

## ●セキュリティグループ

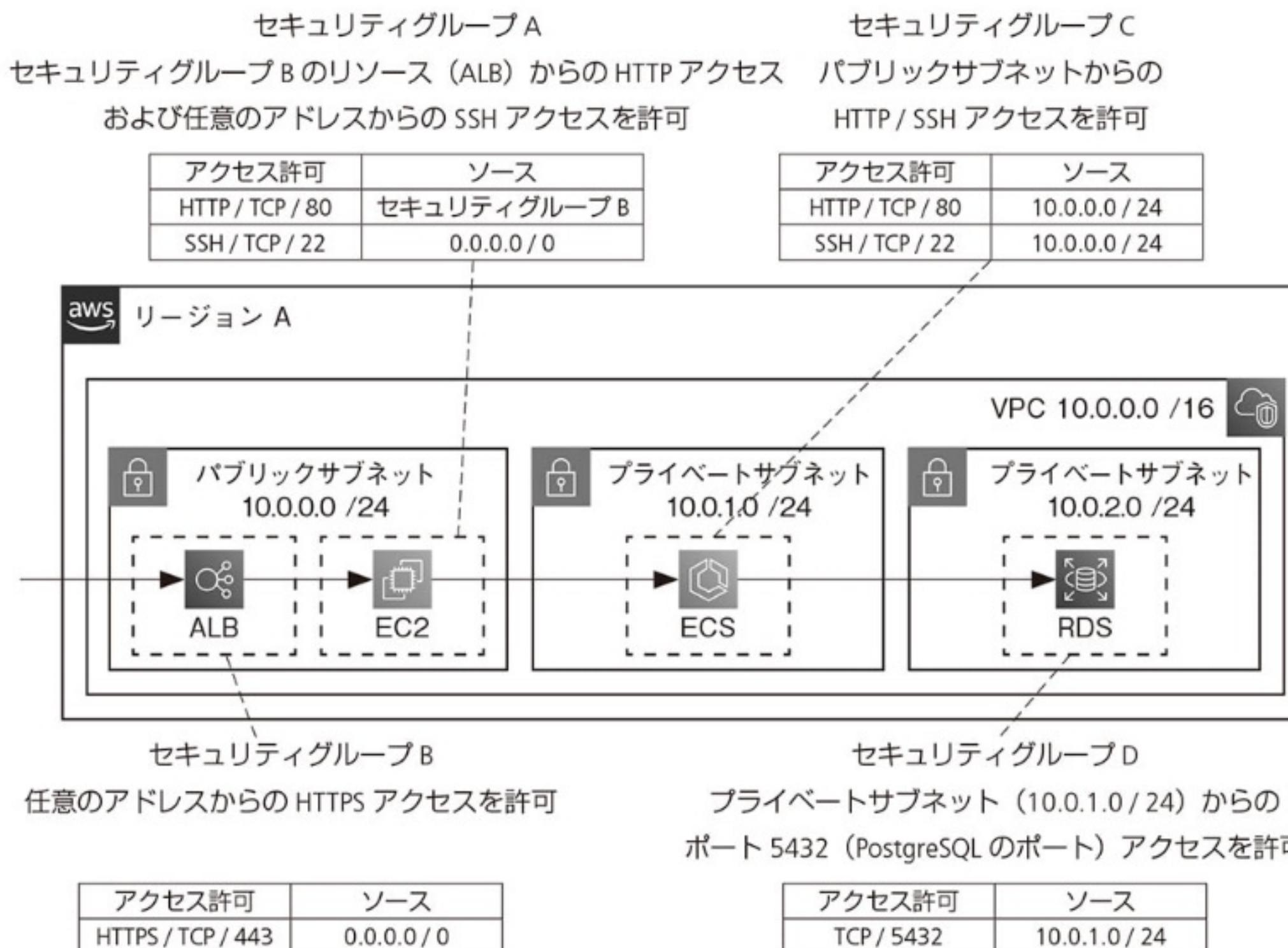
セキュリティグループは、Linuxにおけるファイアウォール機能「IPTables」とほぼ同じ役割を担います。

アクセス許可するソースやプロトコル、ポートのルール定義を作成し、EC2やRDSなどのAWSリソースに関連付けることで、セキュリティグループはファイアウォールとして動作します。受信と送信の両方をリソースレベルで制御でき、リソースをグループ化し、共通のセキュリティグループを関連付けることもできます。

一般的なファイアウォール同様、ステートフルに動作し、発生した通信に対する応答通信は自動的に許可されます。以下の例では、ALBやEC2/ECS、RDSで構成されるWebサーバ/APサーバ/DB 3層アプリケーション構成のセキュリティグループ設定例を示しています。

- ALB : 任意のアドレスからのHTTPSアクセスを許可
- EC2 (Webサーバ) : ALBのセキュリティグループからのHTTPアクセスおよび、任意のアドレスからの運用作業のためのSSHアクセスを許可
- ECS(APサーバ) : パブリックサブネットからのHTTPアクセスおよび、パブリックサブネットからの運用作業のためのSSHアクセスを許可
- RDS (DB[PostgreSQL]) : ECSが属するプライベートサブネットからのTCP 5432ポートアクセスを許可

### 【セキュリティグループの作成・設定例】

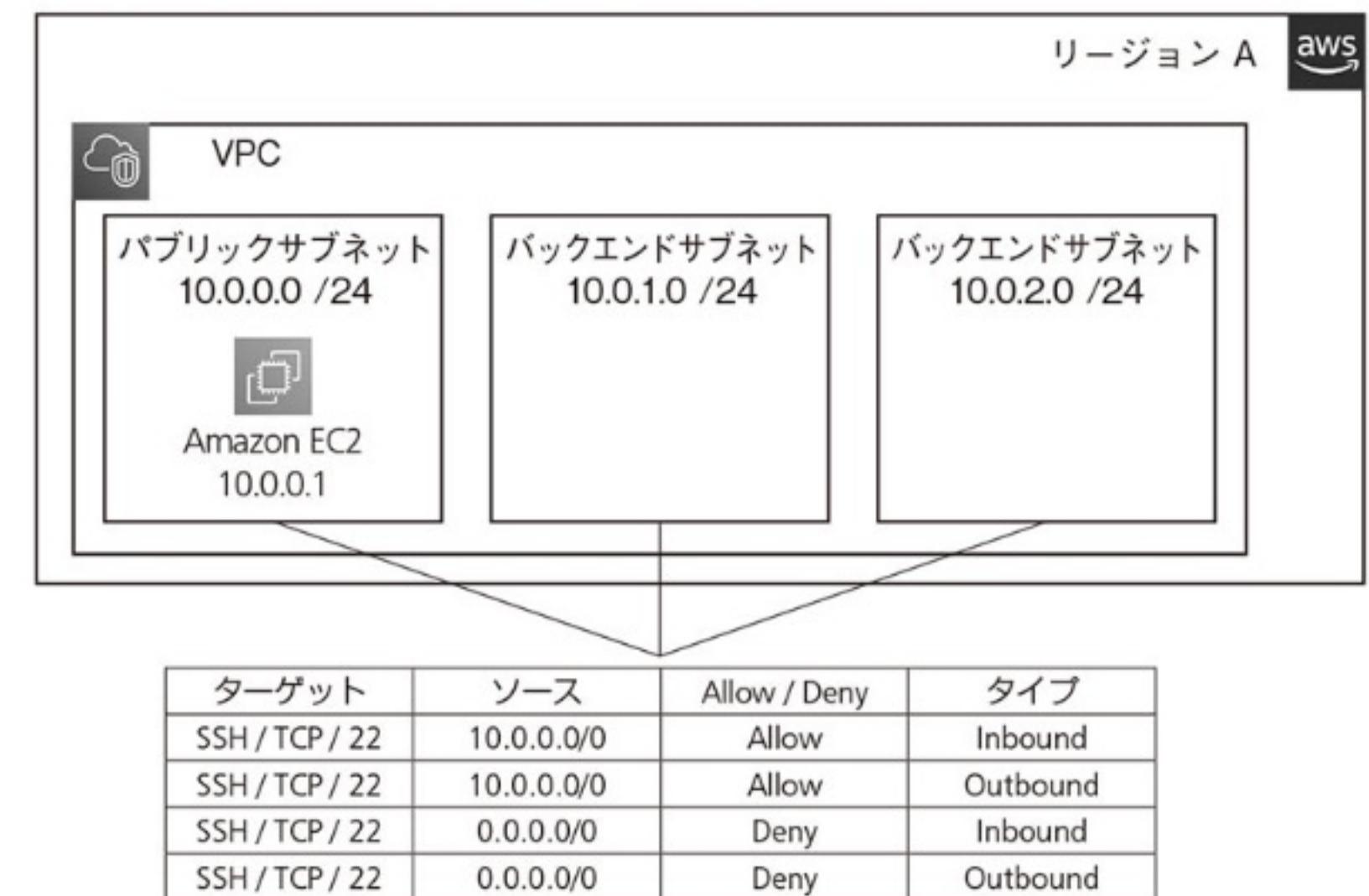


## ●ネットワークACL

ネットワークACLはサブネットに関連付けて、受信と送信、両方の通信をサブネットレベルで制御する機能です。

関連付けられたサブネットすべてに適用されるので、セキュリティグループよりも広範囲な制御・設定が可能ですが、セキュリティグループとは異なりステートレスに動作するため、応答通信にも許可の設定が必要です。

### 【ネットワークACLの作成・設定例】



参考

発生した通信に対する応答通信を許可する場合は、戻り宛先のIPアドレスとポートが必要（ポートは特定できないので事実上任意設定）となることに注意が必要です。

参考

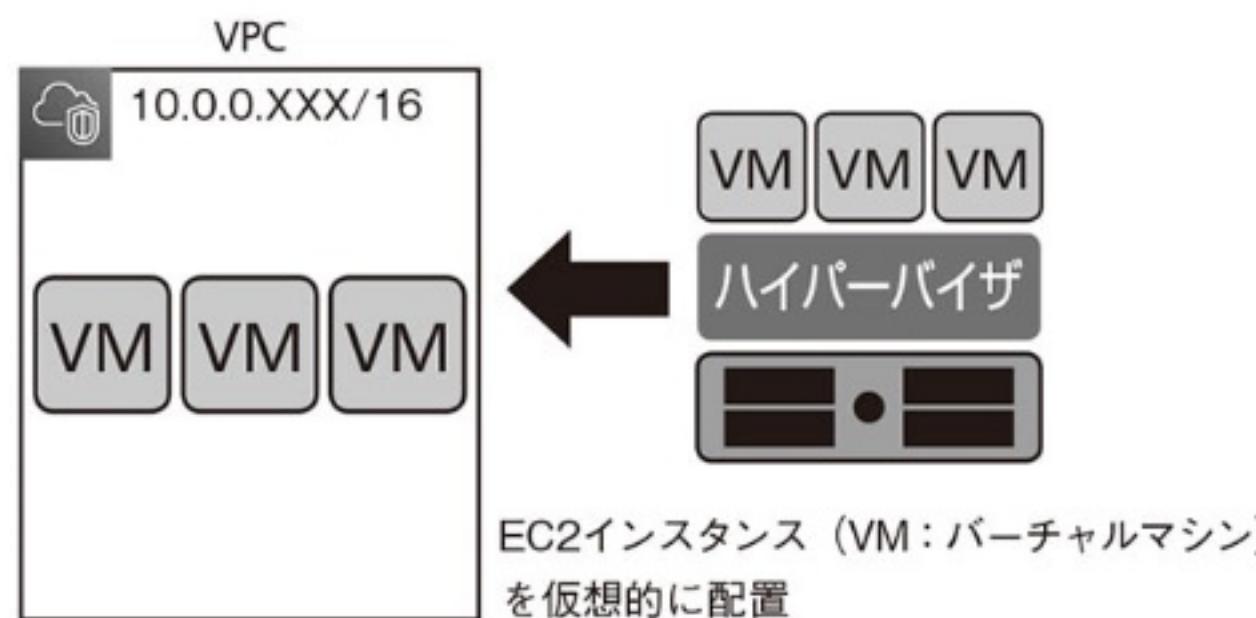
セキュリティグループは許可設定しかできないので、特定のクライアントのアクセスをブロックしたいときはネットワークACLを使います。

## 4 Amazon EC2(Elastic Compute Cloud)

### ●EC2 の概要と使用方法

EC2は、仮想的にサーバ環境を提供するサービスです。

#### 【仮想サーバの配置】



配置する仮想サーバを「インスタンス」と呼び、インスタンスのCPUやメモリ、ストレージといったリソースキャパシティを定義したものを「インスタンスタイプ」と呼びます。

また、EC2上で仮想サーバとして動かすゲストOSをAMI (Amazon Machine Image) と呼びます。AMIにはAWSが提供するAmazon Linuxをはじめ、Windows Server、オープンソースの各種Linuxディストリビューション (CentOS、Debian、Ubuntu、Fedora) などが用意されています。

AWSユーザは、インスタンスタイプと実行するAMIを指定して仮想サーバを構築します。インスタンスの起動はマネジメントコンソールおよび、AWS CLIを用いて実行できます。マネジメントコンソールから仮想サーバを起動する場合、以下のようなフローで、ウィザード形式で設定を行います。

#### 【インスタンスの起動フロー】

- ①名前とタグの設定
- ②AMIの選択
- ③インスタンスタイプの選択
- ④キーペア設定
- ⑤ネットワーク設定
- ⑥ストレージ詳細設定
- ⑦インスタンス詳細設定

#### 【インスタンスの起動フローの詳細】

フロー	説明
①名前とタグの設定	インスタンスの名前と付与するタグを設定します。タグを付与することで、指定したタグが付与されたインスタンスをコンソールのフィルタ機能からピックアップできるようになります。
②AMIの選択	使用するAMIを選択します。自分が登録したAMIのほか、サードパーティやAWS MarketPlaceで提供されているものを選択できます。
③インスタンスタイプの選択	EC2のインスタンスタイプを選択します。インスタンスタイプの一覧から選択可能です。
④キーペア設定	インスタンスに接続するために必要な公開鍵を設定します。なお、Systems Manager Session Managerを使用することにより、キーペアを設定しなくても、より安全にインスタンスに接続できます。
⑤ネットワーク設定	インスタンスを配置するVPCやサブネット、セキュリティグループを指定します。セキュリティグループの詳細は前節VPCのセキュリティグループを参照してください。
⑥ストレージ詳細設定	インスタンスにアタッチするストレージを設定します。なお、ストレージの詳細やオプションについては次項Amazon EBSを参照してください。
⑦インスタンス詳細設定	インスタンスを配置するVPCやサブネット、ハードウェア専有オプションやインスタンスに割り当てるIAMロール、ユーザデータなどを設定します。いくつかのオプションについては後述します。

インスタンス起動後は、キーペアとして設定した公開鍵の対となる秘密鍵を使って、SSHコマンドなどでインスタンスへアクセスします。Windows Serverの場合はリモートデスクトップなどで接続し、サーバ内で必要な追加の環境構築作業を行うかたちになります。

なお、インスタンスを起動する際に、さまざまなソフトウェアをインストールしたり、繰り返し行う共通的な設定作業・項目を実行したりする場合があります。EC2では、以下のようにインスタンス起動時にカスタム処理を実行する機能を提供しています。

#### 【インスタンス起動時のオプション】

ユーザデータ	起動テンプレート (Launch Template)
EC2インスタンス起動時にスクリプト実行を行う機能です <sup>※6</sup> 。シェルスクリプトおよびcloud-initディレクティブ <sup>※7</sup> を用いた2つの方法があります。	前述のインスタンス起動フローの一連の設定をテンプレート化して実行する機能です <sup>※8</sup> 。

#### ●プレイスメントグループ

プレイスメントグループは、EC2インスタンス実行時に、AWSのアベイラビリティゾーンの物理的な配置戦略を選択するオプションです。配置戦略には以下の3種類が用意されています。非機能要件に応じて、適宜必要なプレイスメントグループを作成し、EC2インスタンスを関連付けるとよいでしょう。

#### 【プレイスメントグループの戦略】

配置戦略	説明
Cluster	この戦略のプレイスメントグループに属すると、単一のアベイラビリティゾーンに閉じて、できるだけ近い位置でインスタンスを配置します。インスタンス間で低レイテンシかつ広帯域な通信が必要になる場合に向いているオプションです。
Spread	この戦略のプレイスメントグループに属すると、EC2インスタンスを別々のハードウェアに分散して配置し、障害時に複数のインスタンスが同時にダウンする確率を軽減します。アベイラビリティゾーンをまたいで展開することも可能なため、高い可用性が求められる場合に向いているオプションです。
Partition	この戦略のプレイスメントグループに属すると、インスタンスは同一のハードウェアを共有しない論理的なパーティションに分割して配置されます。分割するパーティション数はユーザが定義できます。同一パーティション内で低レイテンシを確保しつつ、ハードウェア障害による影響を抑えたい場合に有効なオプションです。

※6 [https://docs.aws.amazon.com/ja\\_jp/AWSEC2/latest/UserGuide/user-data.html](https://docs.aws.amazon.com/ja_jp/AWSEC2/latest/UserGuide/user-data.html)

※7 [https://docs.aws.amazon.com/ja\\_jp/AWSEC2/latest/UserGuide/amazon-linux-ami-basics.html#amazon-linux-cloud-init](https://docs.aws.amazon.com/ja_jp/AWSEC2/latest/UserGuide/amazon-linux-ami-basics.html#amazon-linux-cloud-init)

※8 [https://docs.aws.amazon.com/ja\\_jp/AWSEC2/latest/UserGuide/ec2-launch-templates.html](https://docs.aws.amazon.com/ja_jp/AWSEC2/latest/UserGuide/ec2-launch-templates.html)

#### ●EC2インスタンスからのAWSリソースへのアクセス

EC2から、S3やDynamoDBなどほかのAWSサービスへアクセスする場合（APIを実行する場合）、IAMを使った権限の付与が必要になります。

EC2インスタンス上から実行される処理にも、必要に応じて権限を割り当てる必要があります。EC2では主に以下の方法で権限を割り当てますが、最近ではIAMロールを用いた方法が推奨されています<sup>※9</sup>。

#### 【EC2インスタンスからAWSリソースへアクセスする際の権限の設定方法】

方法	EC2上の設定方法	説明
IAMユーザ認証情報を設定	EC2のホームディレクトリ配下の.awsディレクトリに認証情報（アクセスキー、シークレットキー）が設定されたCredentialファイルを配置する。	必要な権限（ポリシー）を設定したIAMユーザを作成し、認証情報をEC2に設定します。ただし、アプリケーションが実際に稼働することになるEC2環境で認証情報を作成して設定することは、認証情報漏洩の観点から推奨されていません。
	EC2の環境変数としてAWS_ACCESS_KEY_IDおよびAWS_SECRET_ACCESS_KEYに認証情報を設定する。	
IAMロールをEC2インスタンスプロファイルに設定	EC2インスタンス起動フロー：インスタンスの詳細設定で、IAMロールをEC2に割り当てる。	必要な権限（ポリシー）を設定したIAMロールを作成し、EC2起動時に設定します。

なお、EC2上のアプリケーションからAWSリソースへのアクセスは、通常SDKを通じて行います。SDKでは、AWSサービスにアクセスする際は主に以下の順序で権限情報を取得します（言語や実行環境ごとに差はあります）。

1. 環境変数
2. ホームディレクトリ配下の認証情報
3. EC2インスタンスプロファイル（EC2で実行される場合）

したがって、EC2ではない環境でアプリケーションを開発する際は、開発用のユーザを作成して、開発端末のホームディレクトリに開発用ユーザの認証情報を保存しておき、実際にEC2へ配置する際はEC2インスタンスプロファイルから権限情報を参照するやり方が推奨されます。

※9 [https://docs.aws.amazon.com/ja\\_jp/IAM/latest/UserGuide/id\\_roles\\_use\\_switch-role-ec2-instance-profiles.html](https://docs.aws.amazon.com/ja_jp/IAM/latest/UserGuide/id_roles_use_switch-role-ec2-instance-profiles.html)

## 5 Amazon EBS

Amazon Elastic Block Storage (EBS) は、EC2向けにブロックストレージを提供するサービスです。EBSで作成されたブロックストレージはEBSボリュームと呼ばれ、EC2にアタッチし、マウントすることで利用できます。

EBSボリュームにはいくつかの種類があり、利用料金やパフォーマンスが異なります。使用頻度の高い順で列挙すると以下のとおりです。

### 【EBSボリュームの種類】

ボリューム	概要	1GBあたりの利用料金	ユースケース
汎用 SSD (gp2)	標準の設定で最も汎用的なボリューム。2020年12月により高スループットなgp3が発表されている。	高額	ブートボリューム、開発テスト環境など
プロビジョンド IOPS SSD (io2)	必要なIOPS（1秒あたりに処理できる読み込み/書き込みのアクセス数）を指定し作成するボリューム。2020年8月により耐久性の高いio2が発表されている。	最も高額	高スループットを要求するデータベースやアプリケーション
スループット最適化 HDD (st1)	高スループットを実現しつつ、利用料金を抑えたボリューム。	低額	データウェアハウス、ビッグデータ分析
Cold HDD (sc1)	データのアーカイブなど利用頻度の低い大量データの格納に向くボリューム。	最も低額	ログデータ、アーカイブ

EBSでは、EBSボリュームのバックアップをスナップショットとして取得できます。スナップショットはS3に格納されます。EBSボリュームのバックアップは、増分バックアップで保存されます。

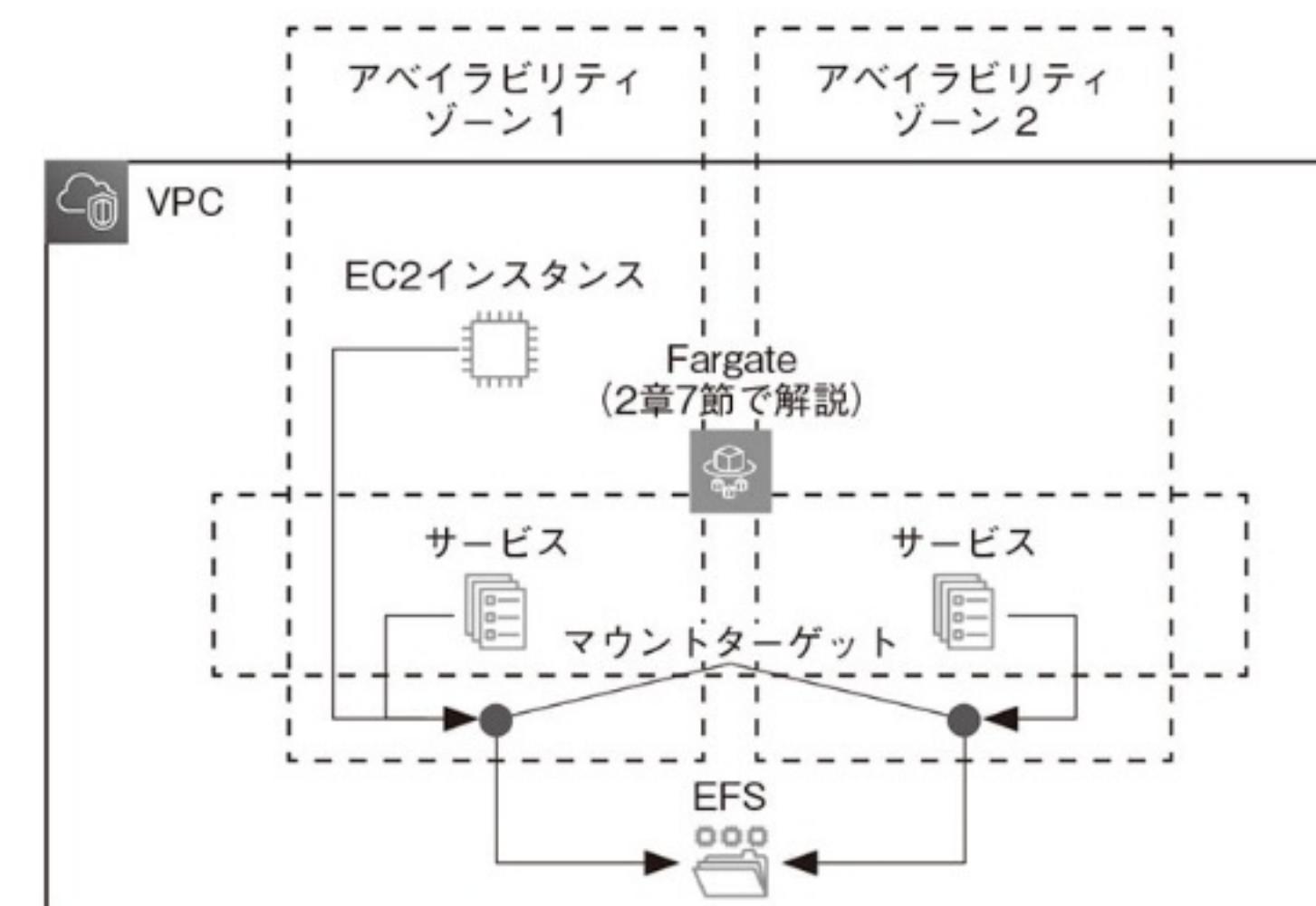
スナップショットは、リージョン内に保存されます。取得したスナップショットを使って、EBSボリュームを作成できます。このとき、アベイラビリティゾーンを指定し直せるため、アベイラビリティゾーンが異なるEC2にもアタッチできます。また、スナップショットは異なるAWSアカウントやリージョンにも複製できます。

## 6 Amazon EFS

Amazon Elastic File System (EFS) は、AWS上でファイルストレージを提供するサービスです。EFSが提供するストレージは、EC2や2章7節で解説するECS、Fargateなど各種サービスから利用可能です。

EFSはVPC内に作成されます。EC2インスタンスやコンテナは各アベイラビリティゾーン（AZ）に作成されるマウントターゲット（EFSに接続するためのネットワークインターフェース）のIPアドレスを使って、EFSに接続します。作成されたファイルストレージはAWSによって管理されるため、サーバやディスクの管理は不要になります。

### 【EFSへの接続】



EFSは通常のファイルストレージとは異なり、以下のような特徴を有しています。

### 【EFSの特徴】

項目	特徴
可用性	EFSストレージに格納されたデータは複数のAZに複製されます。これによりいずれかのAZで障害が発生した場合にも継続してデータアクセスを行えます。
柔軟性	一般的なファイルストレージでは、保存するデータ量の上限が購入したストレージ容量に依存していました。EFSでは、ストレージに格納されたファイルに合わせて自動的にファイルシステムが柔軟に拡張・収縮します。
拡張性	一般的なファイルストレージでは、保存するデータ量が多くなるとリソースを増強する必要がありました。EFSはストレージに格納されたデータ量に応じて自動的にリソース拡張を行えます。

EFSも、S3やEBSのようにストレージクラスを選択できます。利用形態に応じて適切なストレージクラスを選択することで、利用料金を抑えられます。

#### 【ストレージクラス】

ストレージクラス	特徴
標準ストレージクラス	デフォルトで設定されているストレージクラス
標準 - 低頻度アクセスストレージクラス	アクセス頻度の低いデータの格納に向くストレージクラス。データの保存量に応じた料金は標準ストレージクラスより低いものの、ファイルの読み書きのデータ量に応じた料金が追加で発生
1ゾーンストレージクラス	単一のAZに保存されるため可用性が低くなるが、容量あたりの利用料金が標準ストレージクラスよりも低いストレージ。EFSの可用性低下を受容できるシステムで、アクセス頻度の低いデータの格納に向く
1ゾーン - 低頻度アクセスストレージクラス	前述の1ゾーンストレージクラスと低頻度アクセスストレージクラスの特徴を併せ持ったストレージ。アクセス頻度が低く、可用性低下を受容できるシステムに向け、EFSストレージの中で最も料金が低額となる

## 第2章

# 開発関連サービス

2-1. Amazon API Gateway

2-2. AWS Lambda

2-3. Amazon DynamoDB

2-4. AWS Step Functions

2-5. Amazon Route 53

2-6. ELB (Elastic Load Balancer)

2-7. Amazon ECS

2-8. Amazon RDS

2-9. Amazon S3

2-10. Amazon Kinesis

2-11. その他の開発関連サービス

## 2-1 Amazon API Gateway

API Gatewayは、オンラインサービスへのリクエストを受け付ける機能を提供するサービスです。リクエストのタイプに応じて複数のAPIをサポートしています。

本節で解説するサービス「Amazon API Gateway」の説明に入る前に、「API」と「Gateway」とはそもそもどんな役割を果たす機能なのかおさらいしましょう。「API (Application Programming Interface)」とは、サービスの提供者が、自身が構築したアプリケーションサービスを外部に公開して、利用者に使ってもらうために提供するインターフェースのことです。APIにはいくつかの種類がありますが、Amazon API Gatewayは「Web API」を対象とします。

Gatewayは元来、通信プロトコルが異なるネットワーク環境間で、通信方式やデータフォーマットを変換する役割を持つ機器やソフトウェアのことを指します。AWSが提供するAPI Gatewayはこれらの仕組みを簡単に構築できます。また、高い可用性を持ち、HTTP/HTTPSリクエストを受け取ってさまざまなAWS内部のリソースへ中継する機能を持っています。

### 1 API Gatewayが提供するAPIの種類

API Gatewayでは以下の特徴を持つ3種類のAPIを作成できます。

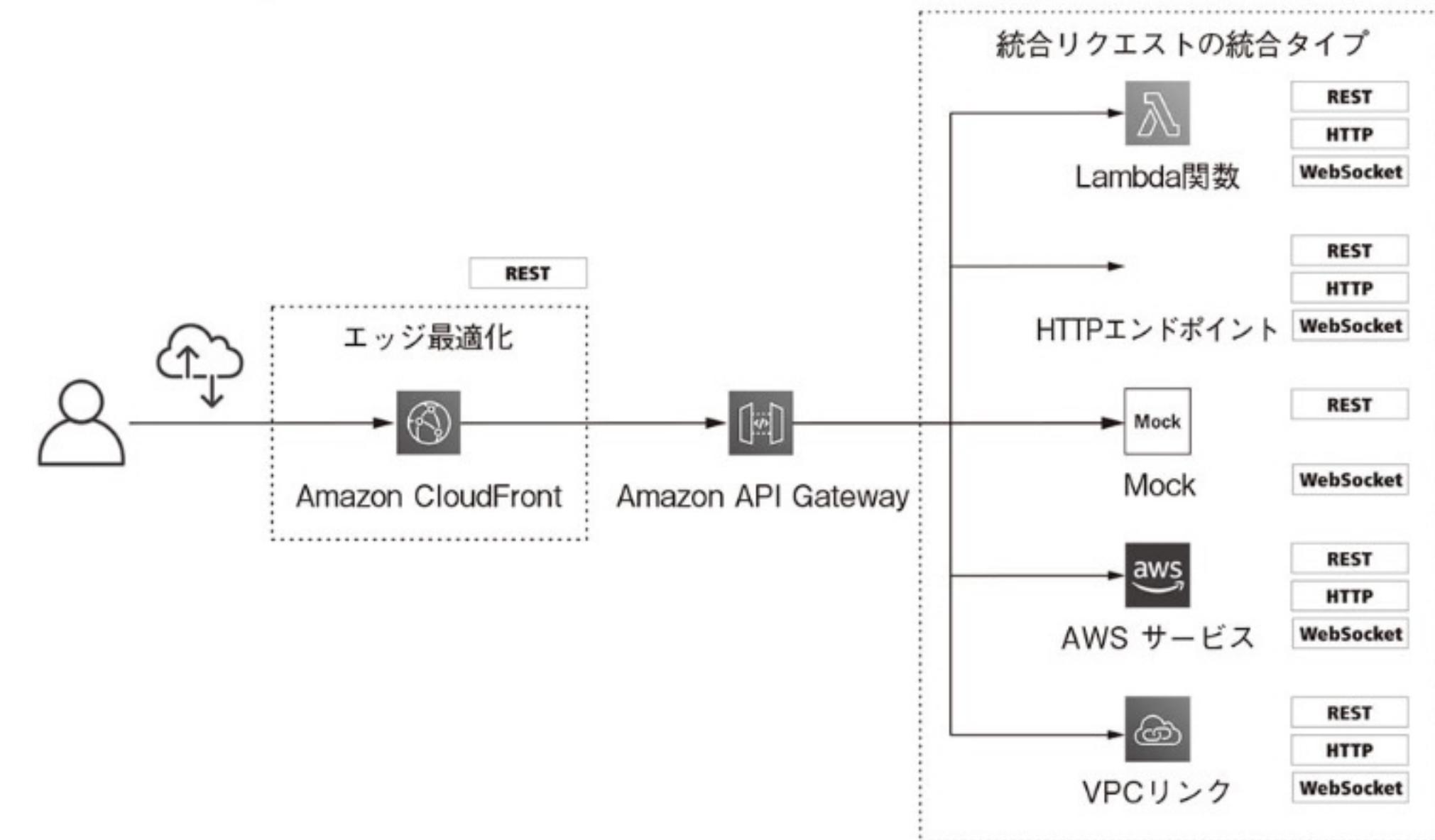
【APIの種類と特徴のサマリー】

API の種類	APIの特徴
REST API	<ul style="list-style-type: none"> <li>AWS Lambda、HTTPエンドポイント、Mock、AWSサービス、VPCリンクにリクエストを転送できる。</li> <li>REST APIをVPC内に構成する「プライベートAPI」も作成可能。</li> <li>OpenAPI (Swagger) v2.0/3.0に準拠した定義ファイルからのAPI作成や既存APIのクローンによる作成が可能。また、JSON/YAML形式のエクスポートにも対応する。</li> <li>ステートレス通信を行う。</li> </ul>
WebSocket API	<ul style="list-style-type: none"> <li>AWS Lambda、HTTPエンドポイント、Mock、AWSサービス、VPCリンクにリクエストを転送できる。</li> <li>WebSocketプロトコルで、チャットやダッシュボードといった双方向通信を扱う際に利用する。</li> <li>ステートフル通信を行う。</li> </ul>

API の種類	APIの特徴
HTTP API	<ul style="list-style-type: none"> <li>AWS Lambda、HTTPエンドポイント、AWSサービス、VPCリンクにリクエストを転送できる。</li> <li>REST APIよりも軽量な実装となっており、性能および費用対効果が高い。</li> <li>OIDCとOAuth 2.0 の認証をサポートし、CORSデプロイと自動デプロイのサポートが組み込まれている。</li> <li>ステートレス通信を行う。</li> </ul>

各APIは統合タイプと呼ばれるさまざまなAWSリソースへリクエストを中継しますが、APIによって選択できるリソースは異なります。例えば、REST APIはエッジ最適化が利用できる唯一のAPIであり、HTTP APIはMockに対応していない点に注意が必要です。各APIが構成可能なエンドポイントや統合タイプを示した全体像は次の図のとおりです。

【API Gatewayの全体像】



### 2 REST API

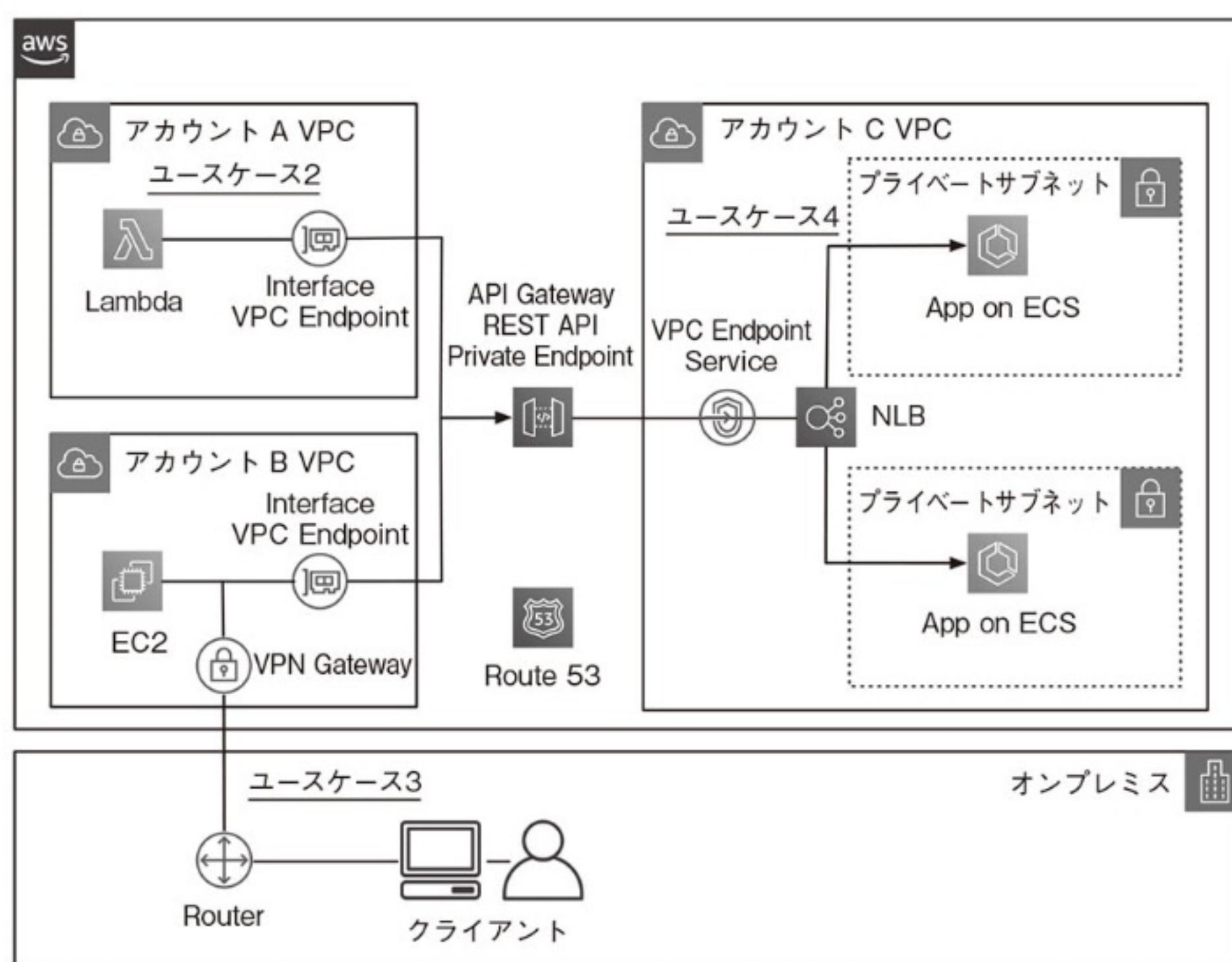
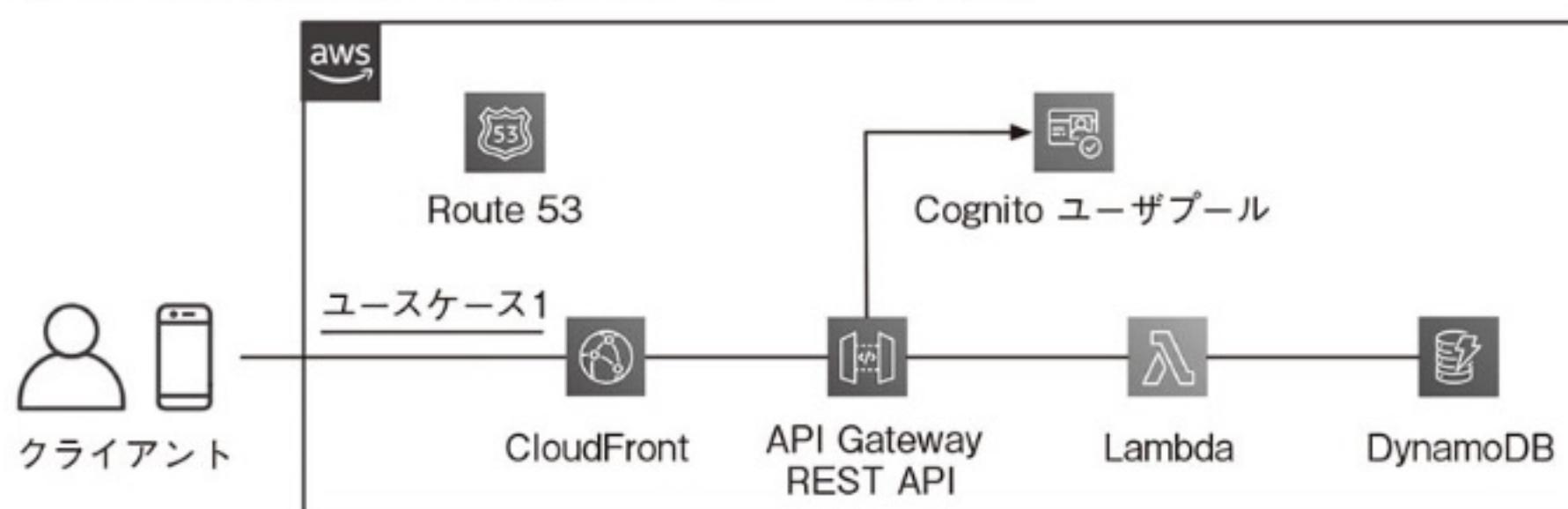
REST APIは、RESTアーキテクチャスタイルにもとづくAPIです。クライアントからリクエストを受け付けるエンドポイントには、以下の3つのエンドポイントタイプを選択できます。

### 【REST APIのエンドポイントタイプ】

APIのタイプ	説明
エッジ最適化APIエンドポイント	Amazon CloudFrontのエッジロケーションを使用して、クライアントに最寄りの接続ポイント(POP)にルーティングする。
リージョンAPIエンドポイント	指定したリージョンにAPIエンドポイントをデプロイし、同一リージョン内のクライアントにサービスを提供する。
プライベートAPIエンドポイント	パブリックなインターネットから分離して、アクセス権限を持ったVPCエンドポイントからのアクセスに限定する。

REST APIを使って構築される代表的なユースケースは以下のようなものがあります。

### 【REST APIを使った代表的なユースケースの図】



### ●【ユースケース1】エッジ最適化されたAPIを利用したサーバレスアプリケーション

モバイルなどのクライアントのバックエンド処理をサーバレスアーキテクチャで構成するユースケースです。バックエンドにあるDynamoDBへアクセスする処理をLambdaを使って構築し、REST APIでリクエストを中継します。API Gatewayはユーザ認証処理にCognitoを利用できます。また、エッジ最適化エンドポイントタイプを利用することで、コンテンツをキャッシュしながら、クライアントに最も近いCloudFrontにルーティングすることができます。必要に応じて、Route 53を用いて、APIのドメインをカスタマイズします。

### ●【ユースケース2】インターフェースVPCエンドポイント（PrivateLink）を経由した、クロスアカウントプライベートAPIへのアクセス

VPC内リソースから、別のアカウントが持つプライベートAPIへアクセスするユースケースです。VPCにアタッチしているLambdaやEC2で実行されている処理の中で、別のアカウントが提供するWebサービスAPIへアクセスしたい場合のリクエストを中継します。エンドポイントタイプとしてプライベートAPIエンドポイントを利用することで、パブリックアクセスを許可せず、インターフェースVPCエンドポイント（PrivateLink）からのみトラフィックを有効化します。API Gatewayリソースポリシーを利用して、特定のアカウントやIPアドレスのトラフィックからのアクセス制御を行うことも可能です。

### ●【ユースケース3】オンプレミスクライアントからのプライベートAPIへのアクセス

オンプレミスで実行されるクライアントから、AWS内で構築されているプライベートAPIへアクセスするユースケースです。まずはオンプレミスのクライアントからVPN GWやDirect Connectを介してVPCに接続。そこから、ユースケース2と同様、インターフェースVPCエンドポイントを経由してプライベートAPIエンドポイントへアクセスします。なお、オンプレミスからプライベートAPIへアクセスするには、Route 53エイリアスレコードを使用するか、オンプレミス内のDNSサーバにRoute 53 Resolverのフォワードを設定します。

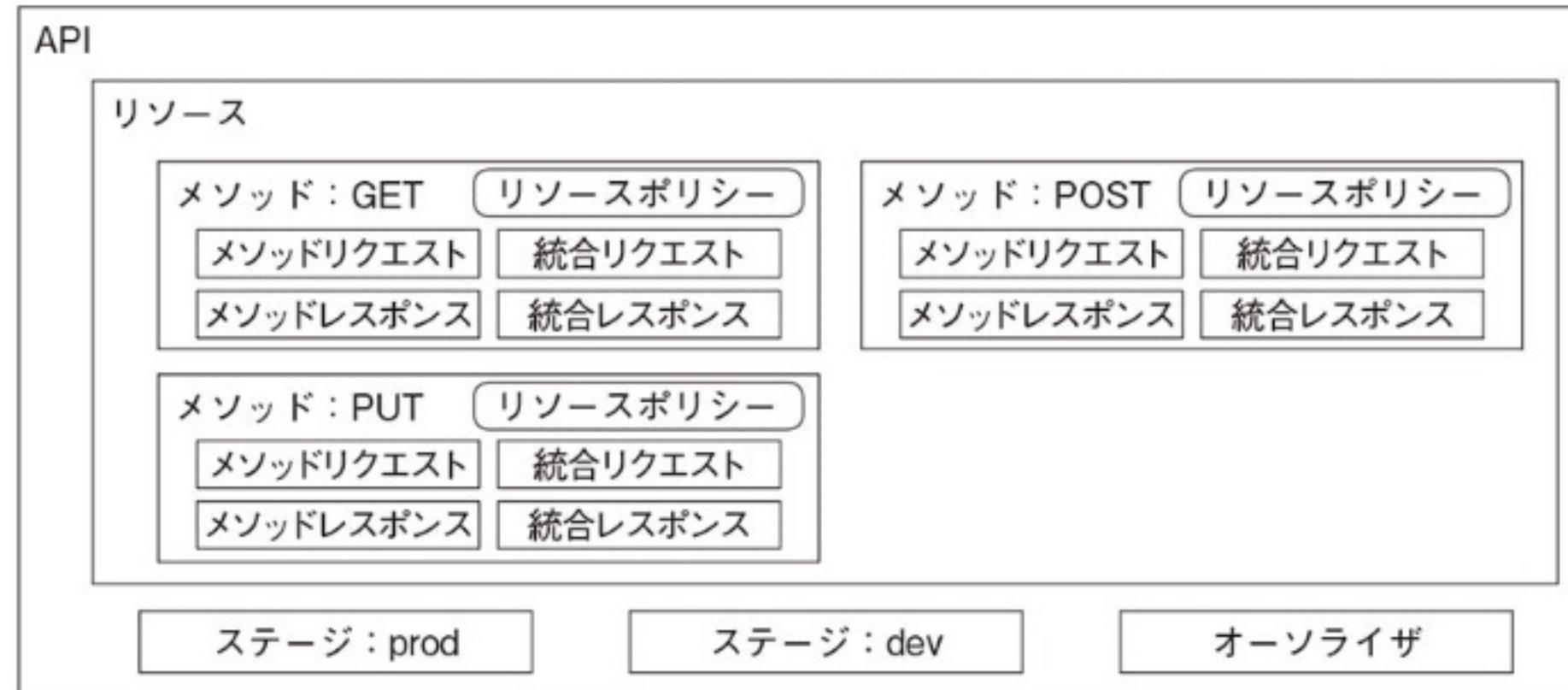
### ●【ユースケース4】API Gatewayを介したコンテナアプリケーションへのアクセス

バックエンド処理をコンテナアプリケーションで構成するユースケースです。統合タイプとしてVPCリンクというネットワークインターフェースを用いて、NLBへのリクエストを中継します。NLBはプライベートサブネットに構築されたECSクラスタへトラフィックをルーティングします。なお、REST APIでは、NLBの代わりにALBを使用することはできません。リクエストパスのルーティングなど、L7レイヤーでリクエストを振り分けるALBを使用するには後述するHTTP APIを使用するか、NLBの背後にALBを配置するかたちで構成します。

REST APIでは、「リソース」と「メソッド」および「ステージ」などを定義して、エンドポイントとなるAPIを構成します。各定義項目の詳細は次のとおりです。

#### 【REST APIの構成要素】

【エンドポイントURL】[https://\[api-id\].execute-api.\[region\].amazonaws.com/\[stageName\]/\[resource\]](https://[api-id].execute-api.[region].amazonaws.com/[stageName]/[resource])



#### 【REST APIの定義項目】

定義内容	説明
リソース	APIが返却するデータを表すモデル。「/」を最上位としたツリー構造で、主に名詞の英語表記を定義する。「/users」、「/users/{id}」、「/users/{id}/address」など、パス変数を{}で指定することも可。
メソッド	リソースが受け付けるHTTPメソッドを定義する。「GET」、「PUT」、「POST」、「HEAD」、「DELETE」、「OPTIONS」、「PATCH」または「ANY」が利用可能。
メソッドリクエスト	メソッドが受け付けるリクエストのクエリパラメータ、認証、必須となるヘッダやAPIキーなどを定義する。
統合リクエスト	バックエンドへ転送するリクエストに関する設定を定義する。 <ul style="list-style-type: none"> <li>「Lambda関数」、「HTTP」、「Mock」、「AWSサービス」、「VPCリンク」などの統合タイプ</li> <li>マッピングテンプレートなどリクエスト変換ルール</li> </ul>
統合レスポンス	バックエンドから返却されたレスポンスに関する設定を定義する。 <ul style="list-style-type: none"> <li>ステータスコードやヘッダのマッピング</li> <li>マッピングテンプレートなどレスポンス変換ルール</li> </ul>
メソッドレスポンス	HTTPステータスコードやコンテンツタイプなど、API Gatewayから返却されるレスポンスの設定を定義する。
モデル	リクエストやレスポンスで共通的に扱われるデータスキーマを定義する。
ステージ	APIがデプロイされる論理的な環境で、エンドポイントURLにステージ名が反映される。慣例的に「prod」、「staging」、「dev」などを必要な環境に応じて設定する。
オーソライザ	API Gatewayで認証・認可に使用される設定を定義する。後述する「IAMアクセス権限」、「Lambdaオーソライザ」、「Cognitoオーソライザ」が選択できる。

## 3 WebSocket API

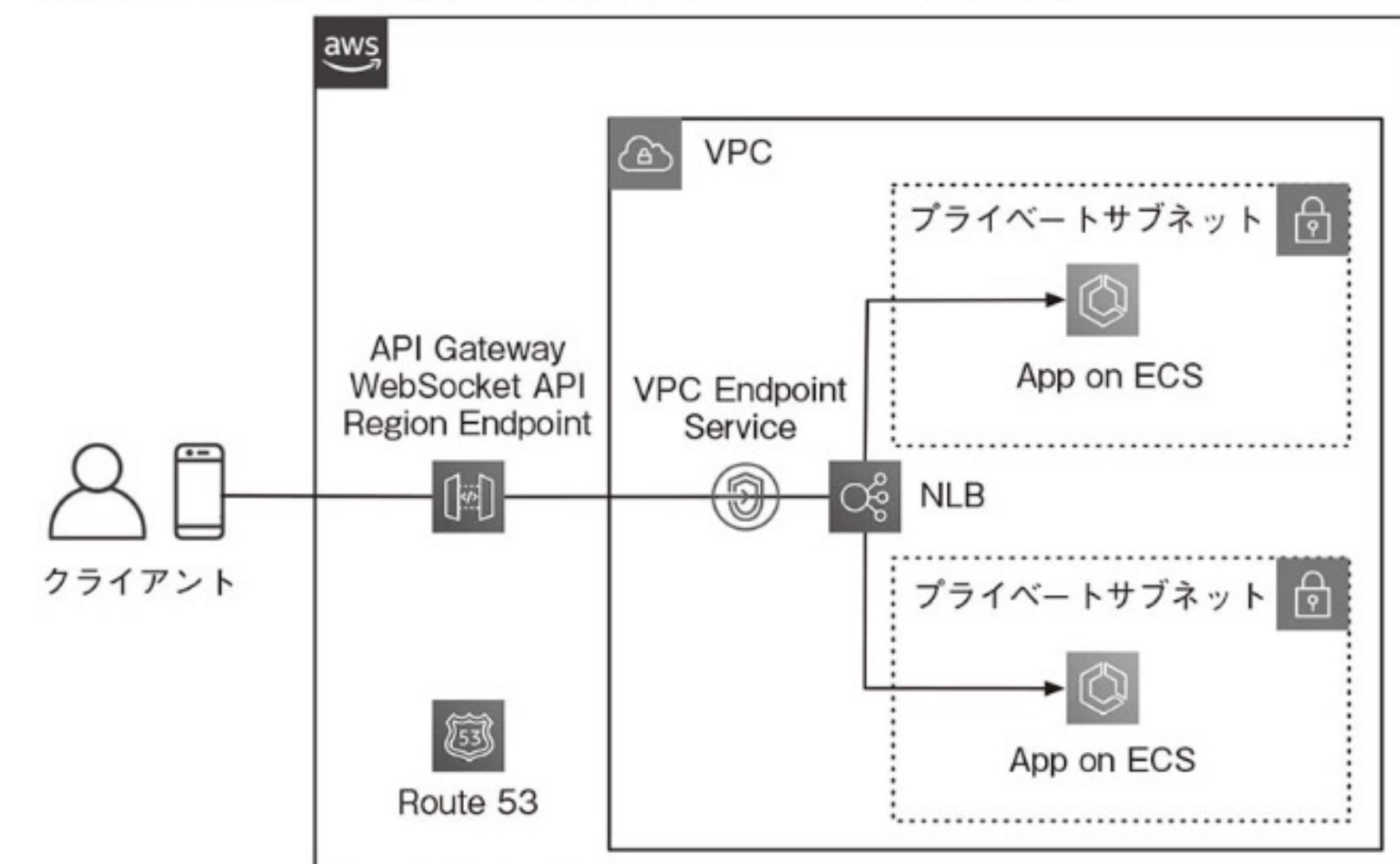
WebSocket APIは、WebSocketプロトコルを使用して双方向のステートレス通信を行うAPIを提供します。WebSocket APIでは、エンドポイントタイプとして、リージョンAPIエンドポイントが利用可能です。

#### 【WebSocket APIのエンドポイントタイプ】

APIのタイプ	説明
リージョンAPIエンドポイント	指定したリージョンにデプロイし、同一リージョン内のクライアントにサービスを提供する。

ディスパッチ先となる統合リクエストタイプはREST APIと同様、「AWS Lambda」、「HTTPエンドポイント」、「Mock」、「AWSサービス」、「VPCリンク」が選択できます。ただし、双方向通信という特性上、VPCリンクを通じたEC2やECSコンテナアプリケーションでの利用が主に想定されます。

#### 【WebSocket APIを使った代表的なユースケースの図】



#### ●【ユースケース】API Gatewayを介したコンテナアプリケーションへのアクセス

バックエンド処理をコンテナアプリケーションで構成するユースケースです。統合タイプとしてVPCリンクというネットワークインターフェースを用いて、NLBへのリクエストを中継します。NLBはプライベートサブネットに構築されたECSクラスタへトラフィックをルーティングします。なお、REST APIと同様、

NLBの代わりにALBを使用することはできません。リクエストパスのルーティングなど、L7レイヤーでリクエストを振り分けるALBを使用するには後述するHTTP APIを使用するか、NLBの背後にALBを配置するかたちで構成します。

WebSocket APIでは、以下のような項目定義が必要です。

#### 【WebSocket APIの設定項目】

設定項目	説明
ルート選択式	API Gatewayが受信したメッセージに対して行う評価の定義。例えば、JSON形式のデータに"action"というキーが存在し、その値によって処理の振り分けを行う場合は <code>\$request.body.action</code> というルート選択式を定義する。
ルート	ルート選択式によって評価された後の処理の振り分け先を定義する。「ルートキー」とも呼ばれる。下記の3つの事前定義されたルートがあり、「カスタムキー」として独自に定義することも可能。 <ul style="list-style-type: none"> <li><code>\$connect</code>: WebSocket接続時のルート</li> <li><code>\$disconnect</code>: WebSocket切断時のルート</li> <li><code>\$default</code>: どれにもあてはまらなかった場合のルート</li> </ul>
統合	リクエストをバックエンドに転送する。バックエンドには、AWS Lambda、HTTPエンドポイント、Mock、AWSサービス、VPCリンクが選択可能である。
ステージ	APIをデプロイする論理的な環境。WebSocket APIは双方向通信であるため、クライアントがバックエンドサービスを呼び出すためのWebSocket URLと、バックエンドサービスがクライアントにメッセージを送信するためのCallback URLの2つが発行される。 エンドポイントURLの例 <ul style="list-style-type: none"> <li>WebSocket URL: <code>wss://[api-id].execute-api.[region].amazonaws.com/{stageName}/</code></li> <li>Callback URL: <code>https://[api-id].execute-api.[region].amazonaws.com/{stageName}/@connections/[connection-id]</code></li> </ul> 上記のURLにおける <code>{api-id}</code> はAPIの固有ID、 <code>{region}</code> はリージョンを示すID、 <code>{stageName}</code> はステージ名、 <code>{connection-id}</code> はクライアントへのコーリバッブを行うために使用する接続IDを表す。

## 4 HTTP API

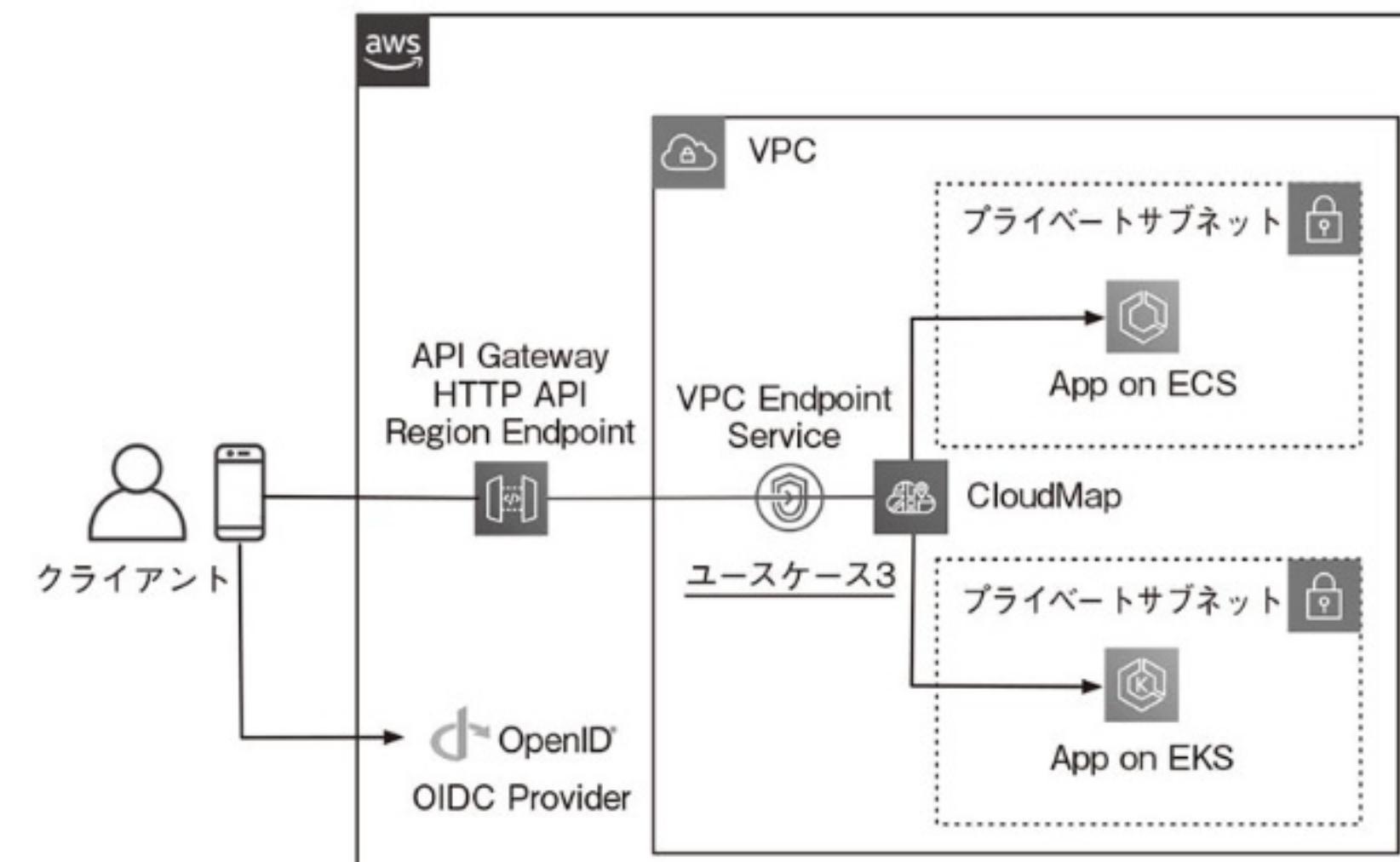
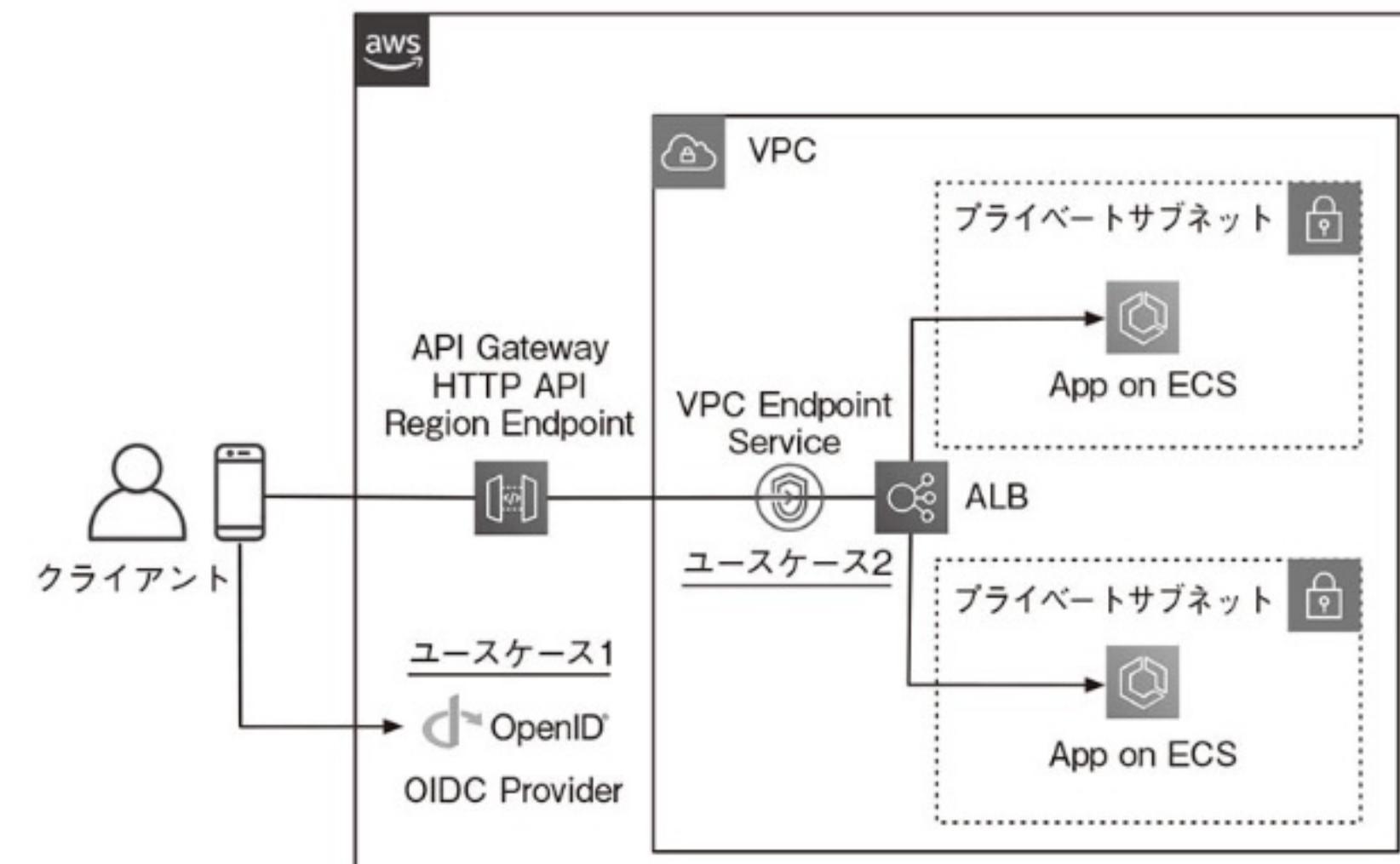
HTTP APIは、REST APIと同様、HTTPリクエストの中継を行います。エンドポイントタイプとして、リージョンAPIエンドポイントが利用可能です。

#### 【HTTP APIのエンドポイントタイプ】

APIのタイプ	説明
リージョンAPIエンドポイント	指定したリージョンにデプロイし、同一リージョン内のクライアントにサービスを提供する。

ディスパッチ先となる統合タイプは「AWS Lambda」、「HTTPエンドポイント」、「AWSサービス」、「VPCリンク」を選択できます。「REST API」と「HTTP API」の違いがわかりづらいですが、ともにRESTfulなAPIを提供するサービスです（節末に「REST API」と「HTTP API」がサポートする機能の違いをまとめています）。機能面の大きな違いとしては、HTTP APIは、OpenID Connect/OAuth 2.0 認証のJWTオーソライザをサポートし、VPCリンクの接続先にNLBのほかALB、AWS CloudMapといったAWSリソースを配置することも可能です。また、HTTP APIはREST APIと比較すると軽量な実装となっており、低レイテンシかつ低コストでRESTful APIを作成できます。

#### 【HTTP APIを使った代表的なユースケースの図】



### ● [ユースケース1]API GatewayのJWTオーソライザ

OpenID Connect (OIDC) プロバイダで認証を行い、発行されたJWTトークンを用いて、API Gatewayの認証を行うユースケースです。ユーザの管理や認証処理をサードパーティプロバイダへ委ね、API Gatewayでアクセス制御します（詳細は後述）。なお、OIDCプロバイダとしてCognitoを選択することも可能です。

### ● [ユースケース2]ダイレクトにALBを経由したコンテナサービスへのアクセス

バックエンド処理をコンテナサービスで構成するユースケースです。統合タイプとしてVPCリンクを用いて、プライベートサブネットに構築されたECSクラスタへトラフィックをルーティングする際に、L7レイヤーでリクエストを振り分けるALBへのリクエストを中継します。マイクロサービスアーキテクチャなどリクエストベースで異なるコンテナヘルーティングしたい場合に構成します。

### ● [ユースケース3]CloudMapを経由したコンテナサービスへのアクセス

バックエンド処理をコンテナサービスで構成するユースケースです。統合タイプとしてVPCリンクを用いて、プライベートサブネットに構築されたECSやEKSへトラフィックをルーティングする際に、サービスディスクバリを行うCloudMapへのリクエストを中継します。複数のコンテナオーケストレーションからなるハイブリッドアーキテクチャを作る場合に構成します。

HTTP APIでは、以下のような項目定義が必要です。

#### 【HTTP APIの設定項目】

設定項目	説明
ルート	HTTPメソッドとリソースパスの2つから構成される。7つのHTTPメソッド (GET、POST、PUT、HEAD、DELETE、OPTIONS、PATCH) とANYが利用可能。
統合	リクエストをバックエンドに転送する。バックエンドには、AWS Lambda、HTTPエンドポイント、AWSサービス、VPCリンクが選択可能である。
ステージ	APIをデプロイする論理的な環境。HTTP APIは\$defaultというステージが用意されており、これを利用するとステージ名を含まないエンドポイントURLが発行される。 エンドポイントURLの例： <a href="https://{{api-id}}.execute-api.{{region}}.amazonaws.com/">https://{{api-id}}.execute-api.{{region}}.amazonaws.com/</a> なお、上記のURLで、{{api-id}} はAPIの固有ID、{{region}} はリージョンを示すIDである。

## 5 API Gatewayの重要な機能と特徴

API Gatewayを利用する上で押さえておきたい重要な機能を解説します。

### ●認証・認可

Amazon API Gatewayで認証や認可の仕組みを実装する場合は、下記の方法から選択することができます。

### ●IAMアクセス権限

REST    WebSocket    HTTP

IAMアクセス権限では、AWS署名バージョン4を利用した認証と認可を行います。AWS署名バージョン4は、IAMユーザーのアクセスキー（アクセスキーIDとシークレットアクセスキー）をもとに作成したハッシュ値です。クライアントは、このハッシュ値をHTTPリクエストヘッダにセットして、API Gatewayにリクエストを送信します。API Gatewayでハッシュ値の検証を行い、問題がなければAPIの呼び出しが許可されます。

### ●Lambdaオーソライザ

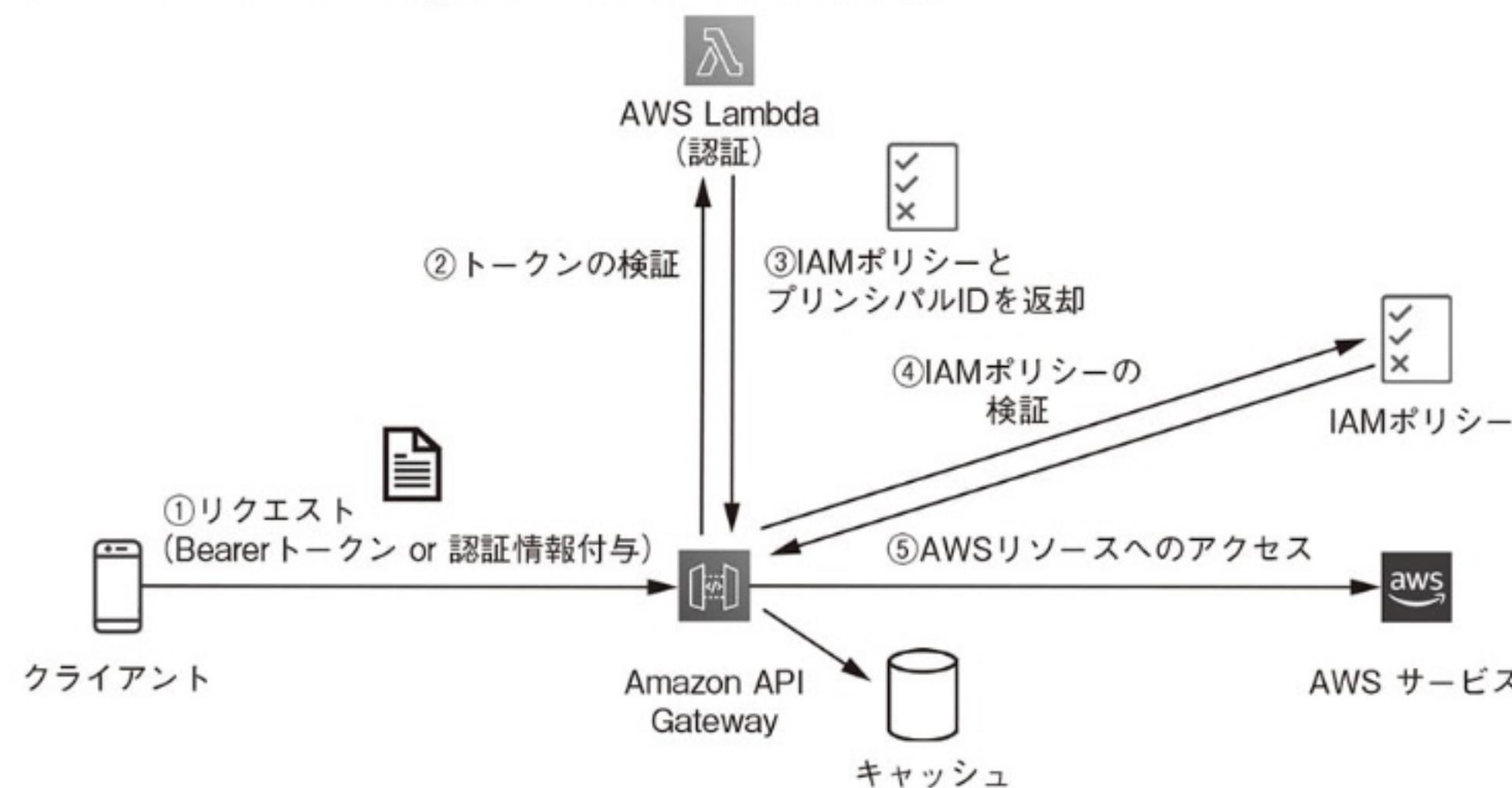
REST    WebSocket    HTTP

Lambdaオーソライザでは、Lambda関数を利用した認証と認可を行います。

クライアントは、API Gatewayに対してBearerトークンもしくはHTTPリクエストヘッダのパラメータに認証情報を付与したリクエストを送付します。Lambda関数がトークンの検証を行い、認証に成功するとLambda関数からIAMポリシーとプリンシパルIDを含むオブジェクトを返却します。API GatewayがIAMポリシーの評価を行い、問題がなければAPIの呼び出しが許可されます。オーソライザの設定でキャッシュが有効となっている場合は、ポリシーがキャッシュされ、次回のアクセスの際にLambdaオーソライザの呼び出しが行われません。

なお、Bearerトークンはトークンを所有していることで認証と認可がされる仕組みです。Bearerトークンはリクエストの送信元を確認しません。何らかの方法で第三者にBearerトークンが渡ってしまった場合も認証と認可ができてしまうので、Bearerトークンの取り扱いには十分注意してください。

### 【Amazon API GatewayのLambdaオーソライザ】



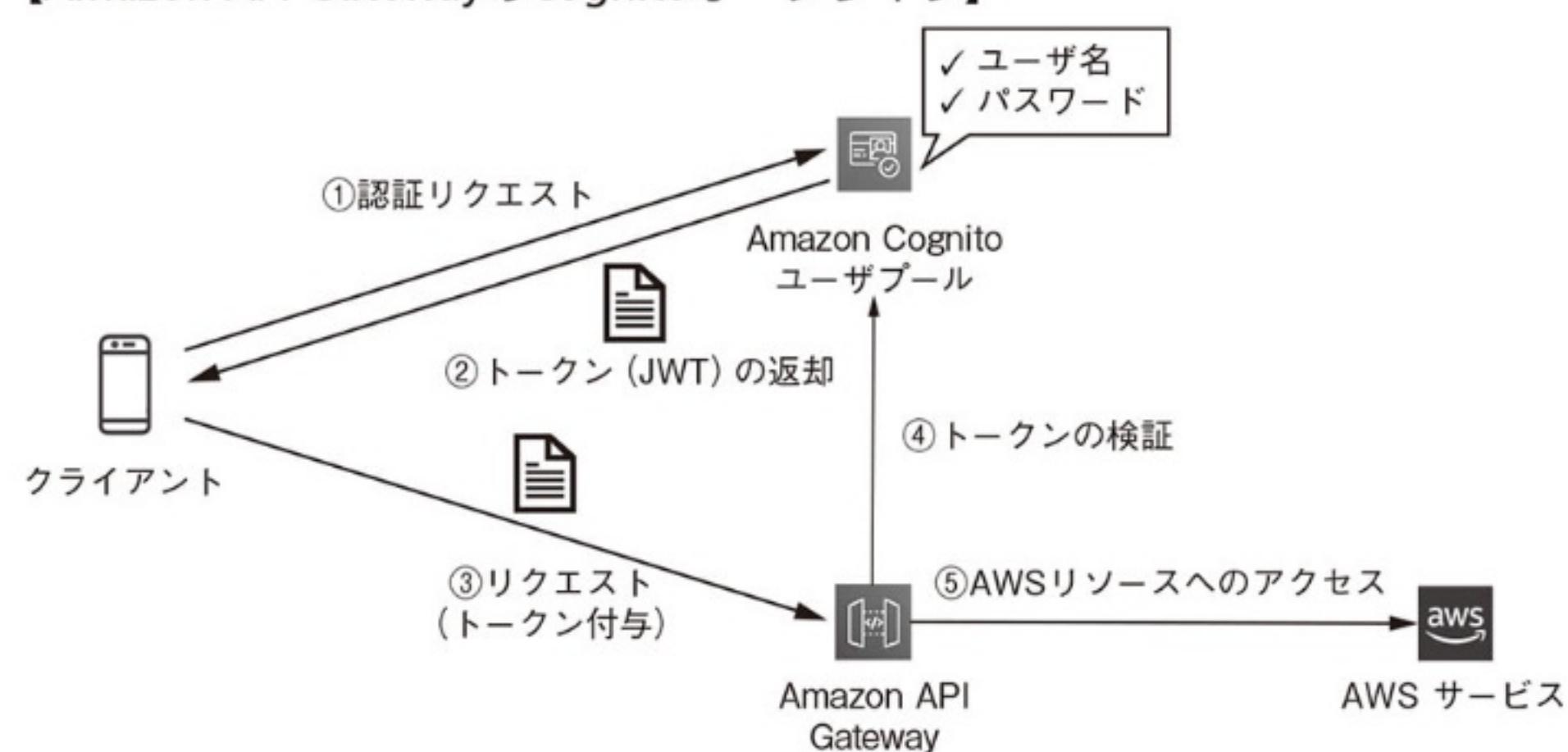
#### ●Cognitoオーソライザ

**REST** **WebSocket**

Cognitoオーソライザでは、Amazon Cognitoのユーザプールを利用した認証を行います。

クライアントは、Amazon Cognitoのユーザプールで認証を行ってトークン (JWT) を取得し、HTTPリクエストヘッダにトークンをセットして、API Gatewayにリクエストを送信します。API Gatewayは、Amazon Cognitoを参照してトークンを検証し、問題がなければAPIの呼び出しが許可されます。

### 【Amazon API GatewayのCognitoオーソライザ】



#### ●JWTオーソライザ

**HTTP**

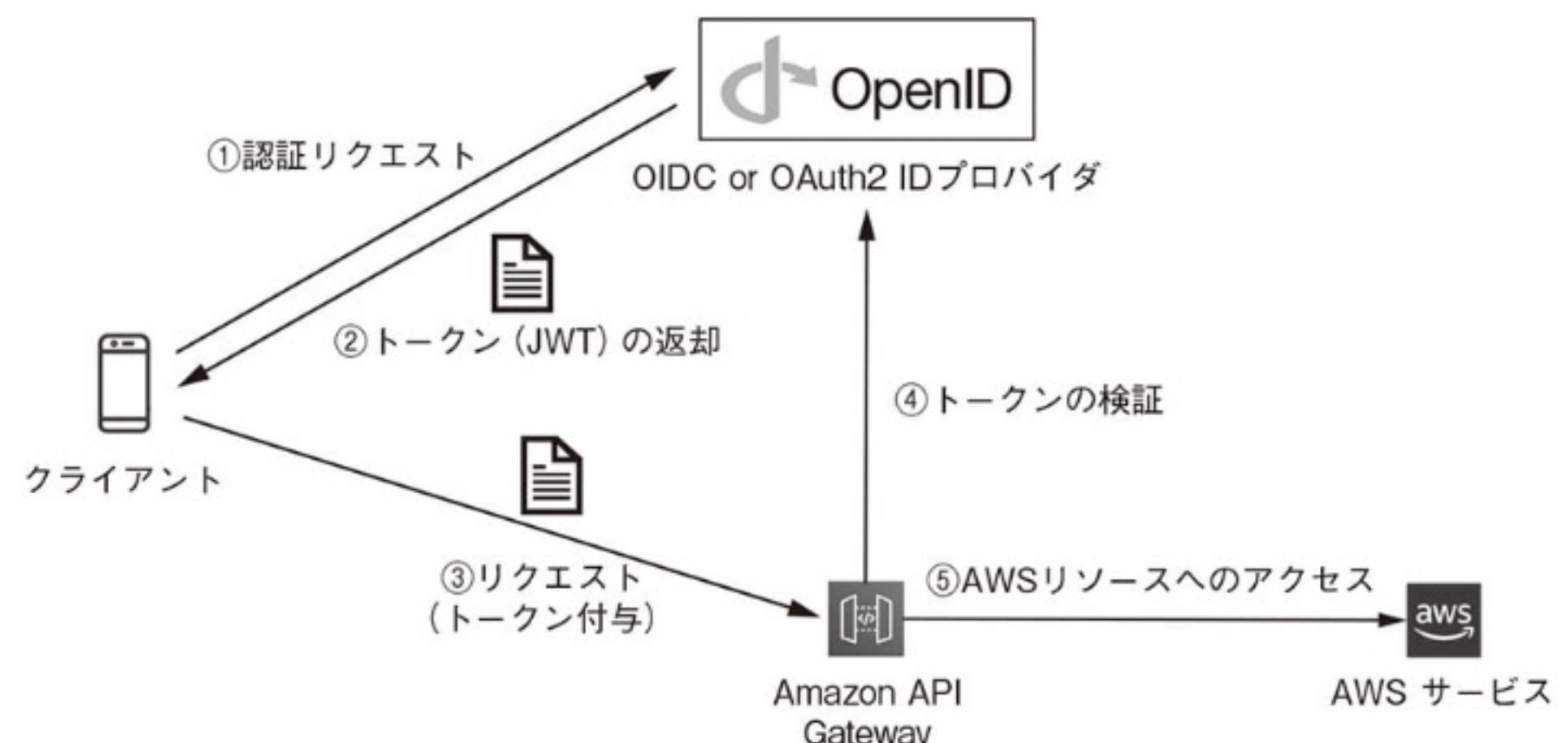
JWTオーソライザでは、OpenID ConnectまたはOAuth 2.0を利用した認証を行います。従来はLambdaオーソライザの開発が必要でしたが、JWTオーソライザの

登場により簡単に認証を実装できるようになりました。本書の執筆時点（2023年12月時点）では、HTTP APIでのみ利用することができます。

クライアントは、OpenID ConnectまたはOAuth 2.0で認証を行ってトークン (JWT) を取得し、HTTPリクエストヘッダに付与して、API Gatewayにリクエストを送信します。API Gatewayは、OpenID Connect (OIDC) またはOAuth 2.0のIDプロバイダにアクセスしてトークンを検証し、問題がなければAPIの呼び出しが許可されます。

なお、プロバイダをCognitoに設定することももちろん可能です。その場合は、Cognitoオーソライザの構成と同じとなります。

### 【Amazon API GatewayのJWTオーソライザ】



#### ●相互TLS認証

**REST** **HTTP**

通常のTLS通信では、クライアントがサーバに対してサーバ証明書を要求して検証を行うことで、接続先のサーバの正しさを確認します。すなわち、片方のみの認証です。

Amazon API Gatewayは、相互TLS (mTLS) 認証をサポートします。相互TLS認証では、サーバがクライアントにクライアント証明書を要求して検証を行い、接続元のクライアントの正しさも確認するため、クライアントとサーバの双方の認証がサポートされます。クライアントおよびサーバの双方の正しさを確認できるため、よりセキュアにAPI Gatewayを利用することができます。

相互TLS認証により、オープンバンキングのセキュリティ要件の準拠やAWS IoTにおけるクライアント認証に対応することができます。

#### ●スロットリング

Amazon API Gatewayには、「スロットリング」と呼ばれるトークンバケットアルゴリズムにもとづく流量制御の仕組みがあります。トークンバケットアルゴリズムでは、1リクエストを処理するたびにバケット内に保持されている「トークン」

を1つ消費します。トークンの補充速度を「定常レート」、バケット内に格納するトークンの最大サイズを「バースト」と呼びます。サーバ側とクライアント側の両方に下記の制限が設定されており、利用者の要件に従って設定することもできます。

- ・サーバ側のスロットリング：すべてのクライアントに適用し、過多なリクエストからバックエンドを守るために設定する。
- ・クライアントあたりのスロットリング：クライアントごとに「使用量プラン」を設定して制限を行う。

それぞれ下記の制限が設けられています。スロットリングの適用順序はクライアントに近いものから順番に適用されます。

#### 【Amazon API Gatewayのスロットリング】

スロットリングの種類	スロットリングのレベル	説明	制限の適用順序
サーバ側のスロットリング	AWSアカウントレベル	デフォルトで、定常レートおよびバーストに下記が設定されている。 ・定常レート：10,000リクエスト/秒 ・バースト：5,000リクエスト	4
	ステージまたはメソッドレベル	特定のステージもしくはAPIの個別のメソッドに対して制限値をオーバーライド（上書き）できる。設定可能な上限値は、AWSアカウントレベルの制限値となる。	3
クライアントあたりのスロットリング	メソッドレベル	クライアントの使用量プランにもとづくメソッドレベルでのスロットリング。	2
	ステージレベル	クライアントの使用量プランにもとづくステージレベルでのスロットリング。	1

これらの制限を超えると、Amazon API GatewayはクライアントにHTTPステータスコード「429 Too Many Requests」を返します。

クライアントあたりのスロットリングに登場する「使用量プラン」とは、クライアントのAPI Gatewayの利用に対してスロットリングや制限を設け、制約をかけるものです。クライアントごとに「APIキー」を作成し、使用量プランと紐付けを行います。



「APIキー」という名称から「認証」を連想しがちです。しかし、クライアントと使用量プランの紐付けに利用するためのものです。APIキーを認証目的で利用することはバッドプラクティスとされています。認証・認可を行いたい場合は、LambdaオーソライザなどのAmazon API Gatewayでサポートされているオーソライザの仕組みを利用して実装します。



SLA (Service Level Agreement) を満たすための使用プランやスロットの上限の設定方法について、試験で問われる可能性があるため、基本事項を押さえておきましょう。

#### ●APIのキャッシング

API Gatewayではステージごとにキャッシングを定義し、バックエンドのトラフィックを削減できます。キャッシングするコンテンツの容量や有効期限（TTL: TimeToLive）も設定可能です。アプリケーションのリリース時などで、キャッシングしたコンテンツを更新したいときは、一時的にキャッシングを無効化します。これは、キャッシングをフラッシュしたり、TTLを一時的に0に設定することで対応します。個別のリクエストでキャッシングを無効化したい場合は、HTTPヘッダに「Cache-Control : max-age = 0」を設定したリクエストを送信することで最新のコンテンツを取得できます。

#### ●カナリアリリース

API Gatewayでは、任意のステージに対して「Canary」という特別なステージを定義できます。API Gatewayへのリクエストを、指定した比率でCanaryステージへ分配することができます。Canaryステージに新たに定義し直したAPIをデプロイすることで、既存と新しいAPIを並列的に共存でき、新しいAPIの試行テストが完了したタイミングで、Canaryを昇格させて置き換える運用が可能になります。

#### ●WAF連携

API GatewayにAWS WAF (Web Application Firewall) のACL (Access Control List) を指定することで、SQLインジェクションやクロスサイトスクリプティングなどのセキュリティ攻撃を防ぐことができます。なお、ACLのベースルールについては AWS マネージドルールグループリストを参照してください<sup>※1</sup>。

#### ●監視連携

API Gatewayは、CloudWatchやX-Rayとシームレスに連携できます。「API呼び出し回数」や「レイテンシ」、「統合レイテンシ」、「4XX系エラー数」や「5XX系エラー数」といった情報が標準メトリクスとしてCloudWatchに収集されるほか、X-Rayトレースを有効化するだけで、リクエストの処理パフォーマンス分析やデバッグが可能になります。

※1 [https://docs.aws.amazon.com/ja\\_jp/waf/latest/developerguide/aws-managed-rule-groups-list.html](https://docs.aws.amazon.com/ja_jp/waf/latest/developerguide/aws-managed-rule-groups-list.html)



「REST API」と「HTTP API」は重複する機能が多いため、サポートする機能の違いをまとめます。どちらも頻繁に機能追加がされているので、最新情報は開発者ガイド<sup>\*2</sup>を参照してください。

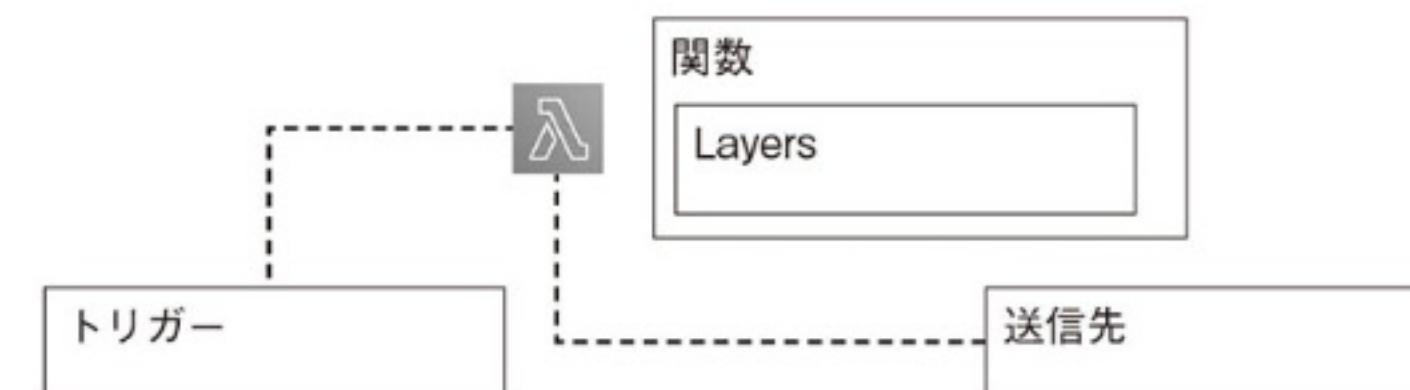
#### 【HTTP APIとREST APIの提供機能の違い】

カテゴリ	項目	HTTP API	REST API
APIのエンドポイントタイプ	エッジ最適化		○
	リージョン	○	○
	プライベート		○
セキュリティ	相互TLS認証	○	○
	クライアント証明書		○
	AWS WAF		○
オーソライザ	IAMアクセス権限	○	○
	リソースポリシー		○
	Cognitoオーソライザ	○	○
	Lambdaオーソライザ	○	○
	JWTオーソライザ	○	
APIの管理	カスタムドメイン名	○	○
	APIキー		○
	クライアントごとのレート制限		○
	クライアントごとの使用量調整		○
開発	CORSの設定	○	○
	テスト呼び出し		○
	キャッシュ		○
	ユーザ制御のデプロイ	○	○
	自動デプロイ	○	
	カスタムゲートウェイレスポンス		○
	カナリアリリースのデプロイ		○
	リクエスト/レスポンスの検証		○
	リクエストパラメータ交換	○	○
	リクエスト本文変換		○
監視	Amazon CloudWatch メトリクス	○	○
	Amazon CloudWatch Logsへのアクセスログ	○	○
	Amazon Kinesis Data Firehoseへのアクセスログ		○
	実行ログ		○
	AWS X-Ray		○
統合リクエストの統合タイプ	HTTPエンドポイント	○	○
	AWSサービス	○	○
	AWS Lambda	○	○
	Network Load Balancer (プライベート統合)	○	○
	Application Load Balancer (プライベート統合)	○	
	CloudMap (プライベート統合)	○	
	Mock		○

## 2-2 AWS Lambda

Lambdaはサーバレスの中核をなすサービスであり、アプリケーションサーバなどの実行ランタイムが完全マネージドとなるコンピューティングサービスです。

Lambdaを利用するには、処理を実装したアプリケーションに加え、下記の図の要素をコンソールから設定します。



### 1 Lambda関数の構成

#### ●ランタイムとアプリケーション

アプリケーションを実行させるために必要なライブラリやパッケージのことを「ランタイム」といいます。各開発言語に対応したランタイムが必要であり、本書の執筆時点では下記のランタイムがLambdaの標準で用意されています。

ただし、一部のランタイムは執筆時点（2023年12月）で非推奨となっています。最新の情報<sup>\*3</sup>も併せて参照してください。

#### 【AWS Lambdaが標準サポートするランタイム】

ランタイム (開発言語)	サポートバージョン
Node.js	18、16、14
Python	3.10、3.9、3.8、3.7
Java	17、11、8
.NET	7、6
Go	1.x
Ruby	3.2、2.7

Lambdaで実行するアプリケーションのソースコードとソースコードの開発言語に対応したランタイムを合わせて「関数」といいます。

\*2 [https://docs.aws.amazon.com/ja\\_jp/apigateway/latest/developerguide/http-api-vs-rest.html](https://docs.aws.amazon.com/ja_jp/apigateway/latest/developerguide/http-api-vs-rest.html)

\*3 [https://docs.aws.amazon.com/ja\\_jp/lambda/latest/dg/lambdaruntimes.html](https://docs.aws.amazon.com/ja_jp/lambda/latest/dg/lambdaruntimes.html)

以下のPythonでHelloWorldを実装したアプリケーションを見てみましょう。関数となるアプリケーションでは、デフォルトで各プログラミング言語で指定された名称のハンドラメソッド（Pythonの場合はLambda\_handler関数）に、実行したい処理を実装します。

#### 【Pythonで実装したLambda関数のHelloWorld】

```
def lambda_handler(event, context):
    return {
        'isBase64Encoded': False,
        'statusCode': 200,
        'headers': {},
        'body': '{"message": "Hello! AWS Lambda!"}'
    }
```

コンソール上で直接ソースコードを記述することができますが、一般的にはソースコードをプログラミング言語の仕様に応じてパッケージ化し、アップロードします（本項の「デプロイパッケージ」にて後述）。アップロードに合わせて、コンソール上で以下のような関数実行のための設定を行います。

#### 【Lambda関数の設定項目】

設定項目	説明
環境変数	コード実行時に読み込む環境変数をKey-Value形式で設定する。
タグ	Lambda関数に付与するタグをKey-Value形式で設定する。
基本設定	基本設定項目として、下記の項目を設定する。 <ul style="list-style-type: none"> <li>説明：Lambda関数に関する説明を記載する。</li> <li>ランタイム：Lambda関数の実行環境を設定する。</li> <li>ハンドラ：Lambda関数の呼び出し時にランタイムで実行するメソッド（エントリーポイント）を指定する。</li> <li>メモリ（MB）：Lambda関数の実行時に確保するメモリを設定する。128MB～3,008MBの範囲で64MB単位で割り当てが可能。割り当てるメモリサイズに応じてCPUの割り当ても増える。</li> <li>タイムアウト：Lambda関数の実行が許可される時間を定義する。1秒単位で最大900秒（15分）まで設定可能。</li> <li>実行ロール：Lambda関数の実行に必要なアクセス権限が定義されたIAMロールを設定する。最低でもログの出力用にCloudWatch Logsへのアクセス権限が必要。</li> </ul>
モニタリングツール	Lambda関数の実行におけるモニタリングツールを設定する。 <ul style="list-style-type: none"> <li>ログとメトリクス：Amazon CloudWatchによるログとメトリクスの取得を有効化する。デフォルトで有効化されている。</li> <li>アクティビトレース：AWS X-Rayによるデータ収集を有効化する。</li> </ul>

設定項目	説明
VPC	Lambda関数をVPC内に配置する際に設定する。RDSなどのVPCリソースへのアクセスが必要な場合に設定する（本項の「VPCアクセス」にて詳述）。
ファイルシステム	Lambda関数がAmazon EFSによるファイルシステムのマウントが必要な場合に設定する。
同時実行数	Lambda関数の同時実行数を設定する。東京リージョンにおける最大値は1,000に設定されており、これを超えるとスロットリミングエラーが発生する。上限数を超えて実行したい場合は、AWSサポートに上限緩和申請を行う必要がある。
非同期呼び出し	Lambda関数の非同期呼び出し時のリトライ操作を設定する（詳細は次項の「同期呼び出しと非同期呼び出し」でも解説）。非同期呼び出しの起動契機となるイベントを、非同期イベントキューに保持する最大時間を設定する。デフォルトで最大値である6時間が設定されている。 再試行：再試行回数を設定する。デフォルトで2回実行される。 デッドレターキューサービス：Lambda関数が処理に失敗したイベントの送信先となるSQSキューもしくはSNSトピックを設定する。
データベースプロキシ	Lambda関数からRDSにアクセスする際に利用するAmazon RDS Proxyを設定する。プロキシ識別子とアクセス対象のRDSインスタンス、データベースのユーザ名とパスワードを保存するSecrets Manager、およびプロキシがシークレットにアクセスするためのIAMロールを設定する。なお、IAM認証でRDS Proxyへ接続する場合、実行するLambdaの権限にも「rds-db:connect」パーミッションが必要になる。



メモリなど、Lambda関数を実行する際にパフォーマンスを向上させるために必要なオプションを押さえておくようにしましょう。

#### ●VPCアクセス

通常、Lambda関数はAWSが管理する環境で実行されますが、RDSやElastiCacheといったVPC内で実行されているリソースにアクセスできるよう設定することもできます。具体的には、Lambda関数の設定でVPCサブネットとLambda向けのセキュリティグループを設定します。この設定により、Lambda関数が実行される際、ENI（Elastic Network Interface）が作成され、このネットワークインターフェース経由でVPCへ接続されます。なお、VPCへのアクセス設定を追加する場合は、Lambda関数の実行ロールに「AWSLambdaVPCAccessExecutionRole」ポリシーをアタッチしておく必要があります。

また、VPCへのアクセス設定を追加することにより、Lambda関数からインターネットへのアクセス経路が失われます。そのため、処理の中でインターネット接続が必要な場合は、Lambda関数に設定したVPCサブネットのルートテーブルにNAT Gatewayへのルーティングを追加しておく必要があります。

## ●コールドスタート・ウォームスタート・Provisioned Concurrency

Lambda関数の実行環境の実体は、Amazon LinuxもしくはAmazon Linux 2をベースとするDockerコンテナです。Lambda関数は、下記に示すライフサイクルにより管理されています。

1. コンテナの作成
2. デプロイパッケージのロード
3. デプロイパッケージの展開
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

Lambda関数に対するリクエストが継続的に発生する場合、コンテナは再利用されるため、1~4に示すLambda関数の準備処理が省略されます。これを「ウォームスタート」と呼びます。

しかし、不要と判断されるとコンテナは破棄され、次回Lambda関数を実行する場合に1から準備をやり直します。これを「コールドスタート」と呼びます。ウォームスタートと比較すると、1~4に示す処理が余計にかかるため、場合によっては性能上の問題が発生することがあります。

性能要件が厳しい箇所でLambda関数を利用する場合は、「Provisioned Concurrency」が効果的です。Lambda関数を事前にプロビジョニングして、Lambda関数のコールドスタートに対処することができます。



上記に加えて、re:Invent 2022でコールドスタートのレイテンシを大幅に改善した「AWS Lambda SnapStart」が発表されました<sup>※4</sup>。Corretto (Java11) ランタイムをはじめ、スナップショット機能を用いて大幅に起動時間を短縮する機能がサポートされています。

## ●カスタムランタイム

Lambdaでは「カスタムランタイム」と呼ばれる機能が用意されています。カスタムランタイムを利用すると、標準のランタイムでサポートされていないバージョンやその他の開発言語のランタイムを作成して実行することができます。例えば、C++で実装されたアプリケーションの実行や「Python 3.8.7」といった特定のバージョンで実装されたアプリケーションの実行が可能となります。

※4 [https://docs.aws.amazon.com/ja\\_jp/lambda/latest/dg/snapstart.html](https://docs.aws.amazon.com/ja_jp/lambda/latest/dg/snapstart.html)

## ●Layers

「Layers (Lambda Layers)」は、複数のLambda関数で共通利用するライブラリ、カスタムランタイム、依存関係をZIPファイルで切り出して共有する機能です。それぞれのLambda関数からは必要なコンポーネントを参照すればよく、コーディング量が減るため、Lambda関数のサイズを小さくすることができます。なお、Layersは、1つのLambda関数から最大で5つまで利用可能です。



ライブラリの共通利用をはじめとしたLambda Layersの利用シーンを押さえておきましょう。

## ●Extensions

「Extensions (Lambda Extensions)」は、Layersを利用したAWS Lambdaの拡張機能です。任意のモニタリング、オブザーバビリティ、セキュリティ、ガバナンス用ツールを利用できるようにAWS Lambdaの機能を拡張することができます。

例えば、EC2インスタンスでモニタリングツールを利用する場合は、EC2インスタンスにクライアントツールをインストールして、クライアントツールが情報の収集と送信を行うケースが多いと思います。しかし、AWS Lambdaの場合は実行環境へのログインが許可されていないため、クライアントツールをインストールすることができません。ツールの開発元が提供するライブラリなどを利用して、関数コードに情報を収集・送信する仕組みを実装する必要がありました。

Extensionsを利用すると、モニタリングツールのクライアント機能をLayersとして実装して、モニタリングツール用に情報を収集・送信できるようになります。

Layersの仕様で1つのLambda関数に最大5個のLayerを使用できますが、Extensionは最大10個まで設定することができます。複数のExtensionを1つのLayerに含めることもできます。

## ●デプロイパッケージ

Lambda関数をビルドして作成したパッケージをAWS Lambdaにデプロイすることを「デプロイパッケージ」といいます。パッケージの形式として、下記の2つがサポートされています。

### 【デプロイパッケージでサポートされる形式と特徴】

サポート形式	特徴
ZIP	<ul style="list-style-type: none"> <li>・従来からサポートされていた方法</li> <li>・サイズの上限が250MB</li> </ul>
Dockerコンテナイメージ	<ul style="list-style-type: none"> <li>・AWS re:Invent 2020にて新たにサポートされた方法</li> <li>・サイズの上限が10GB</li> <li>・ZIP形式よりも柔軟性が高い</li> <li>・ECSやFargateなど、ほかのコンテナベースのアプリケーションとデプロイ方法を統一できる</li> <li>・機械学習のように依存関係が多く、パッケージサイズが大きくなりがちなワークロード（処理）にも対応可能</li> </ul>

AWSが管理するDockerコンテナイメージ「AWSベースイメージ」がDocker HubやAmazon ECR Publicにて提供されています。ベースイメージとは、実際に作るコンテナの土台（ベース）となるDockerイメージのことです。

Docker Hub / Amazon ECR Public用のDockerfileもGitHubにて公開されている<sup>※5</sup>ので、これを流用して、独自のDockerコンテナイメージを作成することも可能です。

Dockerコンテナイメージによるデプロイパッケージは、ZIP形式と比較すると、アプリケーション実行環境のカスタム設定を含めてビルドできるので、高い柔軟性を備えています。

AWSベースイメージは、Amazon Linux 2もしくはAmazon Linuxがベースイメージとなっています。AWS環境に最適化されているため、特に理由がなければこちらのイメージを使うとよいでしょう。Dockerコンテナイメージによるデプロイパッケージでは、AWSベースイメージのほかAlpine、CentOS、Debian、UbuntuなどのOSのベースイメージも利用できます。カスタムランタイムの構成も可能です。

Dockerコンテナイメージ内の関数コードからAWS LambdaのサービスにアクセスするにはRuntime APIが必要となります。AWSからRuntime APIをパッケージ化した「Runtime Interface Client (RIC)」が提供されています。AWSベースイメージを利用しない場合は、RICのインストールが必要となる点に注意してください。

### ●バージョニングとエイリアス

Lambda関数のコンソールから、現時点の設定内容をバージョン番号を割り振って管理することができます。これをバージョニングと呼びます。作成したバージョンは内容の変更不可（イミュータブル）となります。バージョン番号は1から順にインクリメントされていきますが、\$LATESTが最新のバージョンを示しています。

また、これらのバージョンに対し、任意の名前をエイリアスとして付与できます。エイリアスは呼び出し側から参照する名前として利用でき、例えば「Dev」や

「Production」といったエイリアスをバージョンに付与することで、呼び出し元がバージョンを意識せずに常に固定の名前でLambda関数を呼び出すことができます。

なお、エイリアスの設定では、加重エイリアスという、2つのバージョンでトラフィックの割合を分ける機能があります。カナリアリリースのように新しいバージョンを一部のリクエストだけに割り当て、問題の有無を確認し段階的にリリースしたい場合に利用するとよいでしょう。



バージョニングとエイリアスを使ってLambda関数でカナリアリリースを実現する場合の設定方法を押さえておきましょう。



LambdaからRDSへの接続をサポートするRDS Proxyが2019年に発表されています。従来、Lambdaのように1リクエストにつき、1つのコンテナを占有して使用するサーバレスコンピューティングは、RDSをはじめとしたRDBのコネクションプーリングによる接続とは相性がよくないといわれていました。これは、リクエストが発生するたびにコネクションの生成・接続を行い、リソースの消費やオーバーヘッドが大きいためです。従来、RDBへ接続するアプリケーションは、数が大きく変動することではなく、また多数のリクエストを1つのアプリケーションで同時に処理できるよう構成されています。これと同じようななかたちでRDSへの接続を実現するのがRDS Proxyであり、プロキシという名前のとおり、複数のLambda関数からのアクセスを中継し、RDSコネクションプールを管理して接続します。プロキシを通してRDSへ接続しますが、従来どおりRDSの認証情報を使った接続のほか、IAMを使った認証の接続がサポートされています。

## 2 Lambda関数の呼び出し設定

Lambda関数を呼び出すリソースまたは設定を「トリガー」といいます。Lambda関数はコンソール画面、AWS CLI、AWS SDKから直接呼び出すことも、ほかのAWSサービスから呼び出すこともできます。

### ●イベントとイベントソース

Lambda関数の呼び出し元から連携されるJSON形式のドキュメントデータを「イベント」といいます。Lambdaランタイムはイベントをオブジェクトに変換してハンドラにパラメータとして渡します。

※5 <https://github.com/aws/aws-lambda-base-images>

次のJSONドキュメントは、デプロイパッケージをAmazon S3にアップロードした際にLambda関数に連携されるイベントの例です<sup>※6</sup>。

```
{
  "Records": [
    {
      "eventVersion": "2.1",
      "eventSource": "aws:s3",
      "awsRegion": "us-east-2",
      "eventTime": "2019-09-03T19:37:27.192Z",
      "eventName": "ObjectCreated:Put",
      "userIdentity": {
        "principalId": "AWS:AIDAINPONIXQXHT3IKHL2"
      },
      "requestParameters": {
        "sourceIPAddress": "205.255.255.255"
      },
      "responseElements": {
        "x-amz-request-id": "D82B88E5F771F645",
        "x-amz-id-2": "vlR7PnpV2Ce81l0PRw6jlUpck7Jo5ZsQjryTjKlc5aLWGV
HPZLj5NeC6qMa0emYBDXOo6QBU0Wo="
      },
      "s3": {
        "s3SchemaVersion": "1.0",
        "configurationId": "828aa6fc-f7b5-4305-8584-487c791949c1",
        "bucket": {
          "name": "lambda-artifacts-deafc19498e3f2df",
          "ownerIdentity": {
            "principalId": "A3I5XTEXAMA13E"
          },
          "arn": "arn:aws:s3:::lambda-artifacts-deafc19498e3f2df"
        },
        "object": {
          "key": "b21b84d653bb07b05b1e6b33684dc11b",
          "size": 1305107,
          "eTag": "b21b84d653bb07b05b1e6b33684dc11b",
          "sequencer": "0C0F6F405D6ED209E1"
        }
      }
    }
  ]
}
```

```
}
```

イベントの発生元となるAWSサービスおよびユーザアプリケーションをイベントソースと呼びます。イベントソースは、イベントの連携方法によって次の2種類に分けられます。

### 1. ポーリングベースのイベントソース

Lambda関数がポーリングを実行することによりイベントが連携されるパターンです。2章3節で解説するDynamoDB Streamsや2章10節で解説するKinesis Data Streams、Amazon Managed Streaming for Apache Kafkaのようなストリームベースのものと、5章4節で解説するSQS、Amazon MQのようなキューベースのタイプがあります。ポーリングベースのサービスでは、Lambda関数でイベントソースの情報を設定する必要があります（イベントソースマッピング）。

### 2. イベントドリブンのイベントソース

イベントソース自体からLambda関数を呼び出すパターンです。イベントソース自体で呼び出すLambda関数の情報を持っています。

## ●同期呼び出しと非同期呼び出し

Lambda関数の呼び出し方にも「同期呼び出し」と「非同期呼び出し」の2種類があります。利用者が独自に実装したアプリケーションはどちらの方法もとれます。以下はAWS CLIでコマンドラインからLambda関数の同期呼び出しと非同期呼び出しをそれぞれ実行する例です。同期呼び出しではinvocation-typeオプションを指定しない、もしくはRequestResponseを指定し、非同期呼び出しではEventを指定して実行します。

### 【同期呼び出し例】

```
aws lambda invoke \
--function-name my-function \
--payload '{ "name": "Bob" }' \
response.json
```

### 【同期呼び出し結果】

```
{
  "ExecutedVersion": "$LATEST",
  "StatusCode": 200
}
```

※6 [https://docs.aws.amazon.com/ja\\_jp/lambda/latest/dg/with-s3.html](https://docs.aws.amazon.com/ja_jp/lambda/latest/dg/with-s3.html) より引用

## 【非同期呼び出し例】

```
aws lambda invoke \
--function-name my-function \
--invocation-type Event \
--payload '{ "name": "Bob" }' \
response.json
```

## 【非同期呼び出し結果】

```
{
  "StatusCode": 202
}
```

非同期呼び出しでは、リクエストが正常に受け付けられたかどうかのステータスコードが返却されるのみです。

AWSサービスから呼び出す場合はサービスごとに呼び出しが下記のように決まっています。

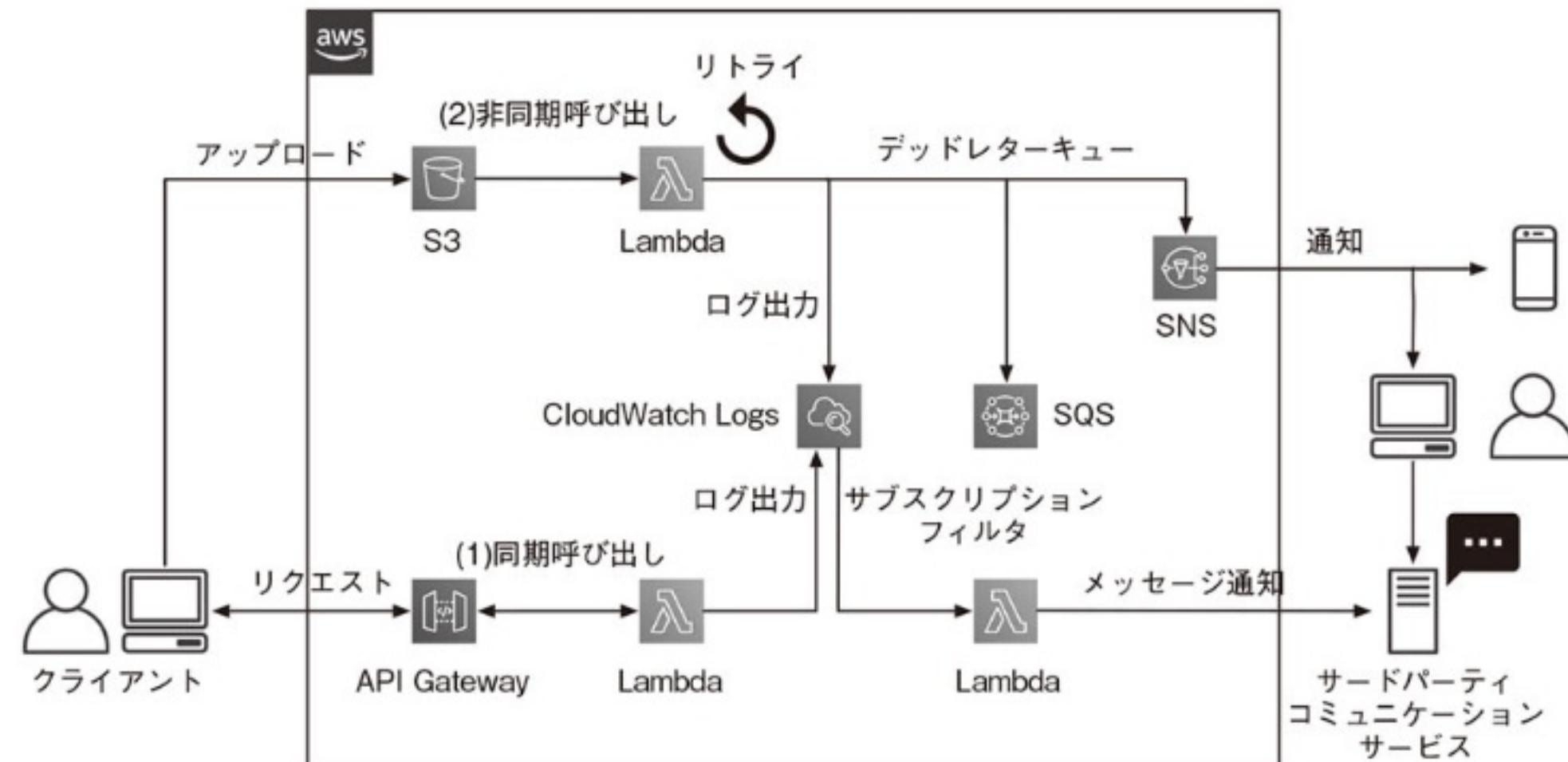
## 【Lambda関数の呼び出し方】

呼び出し方	説明	呼び出し元となるAWSサービスの例
同期呼び出し	Lambda関数からのレスポンスの返却を待つ。	<ul style="list-style-type: none"> <li>• Amazon API Gateway</li> <li>• Amazon DynamoDB</li> <li>• Amazon SQS</li> <li>• Amazon Cognito</li> <li>• Amazon Kinesis Data Streams</li> <li>• Amazon Kinesis Data Firehose</li> </ul>
非同期呼び出し	Lambda関数のレスポンスの返却を待たず、リクエストの受付結果のみを受領する。	<ul style="list-style-type: none"> <li>• Amazon S3</li> <li>• Amazon SNS</li> <li>• Amazon CloudWatch Events</li> <li>• Amazon EventBridge</li> </ul>

## ●エラーハンドリングと通知、リトライ処理

前述のLambdaへの呼び出し方法によって、Lambdaにおけるエラーハンドリングのやり方や、リトライ処理の挙動が変わってきます。以下の図は、Lambdaを同期・非同期で呼び出した場合のエラーハンドリングや通知、ログ出力、リトライに関するユースケースを示したものです。

## 【Lambdaにおけるエラーハンドリングと通知、リトライ処理】



## 1. 同期呼び出し

同期呼び出しの代表例は、API GatewayからのLambda関数の呼び出します。関数実行時にエラーが発生した場合、レスポンスにエラーの内容を示すステータスコードおよび、レスポンスヘッダ「X-Amz-Function-Error」にFunctionErrorが設定された状態で、呼び出し元にレスポンスが返却されます。特定のエラーが発生した際に、管理者などへ何らかの通知を行いたい場合は、CloudWatch Logsのサブスクリプションフィルタ（5章1節「Amazon CloudWatch」にて詳述）などを利用するとよいでしょう。なお、SlackやMattermostといったサードパーティコミュニケーションツールを利用する場合は、Lambda関数でサービスエンドポイントへ連携するかたちになります。

## 2. 非同期呼び出し

非同期呼び出しの代表例は、S3へファイルアップロードした際にイベント駆動でLambda関数を実行するケースです。この場合、Lambdaの処理中にエラーが発生するとデフォルトで2回までリトライされます。非同期呼び出しでは、同期呼び出しと異なり、処理が異常終了しても呼び出し元へ通知されることはありません。代わりにSNSやSQSに対し、デッドレターキューを設定することができます。クライアントや管理者への通知に活用しましょう。サードパーティコミュニケーションツールを利用する場合は、CloudWatch Logsのサブスクリプションフィルタで特定のエラーをピックアップし、Lambda関数でサービスエンドポイントへ連携します。



デッドレターキューがどのような場合やユースケースで利用できるか押さえておくようにしましょう。

なお、上記の図には登場していませんが、DynamoDB StreamsやKinesis Data Streamsのストリームベースのイベントソースで Lambda関数の処理実行中にエラーが発生した場合、イベント元のデータの有効期限が切れるまでリトライ処理が継続されます。SQSをはじめとしたキューベースのタイプのイベントソースは、可視性タイムアウト（5章4節「Amazon SQS」にて詳述）が過ぎたのち、リトライされます。



2018年からALB（アプリケーションロードバランサー）のターゲットにAWS Lambdaを選択できるようになりました<sup>※7</sup>。Lambdaといえば、API Gatewayとセットで構築するイメージがありますが、パスルーティングやコンテンツルーティングを使用して、特定のURLのパスだけをLambdaで処理するように構成することもできます。

### ●送信先

呼び出し方法が非同期呼び出し、もしくはポーリングベースのストリーム呼び出しの場合、同期呼び出しと異なりレスポンスを返却しません。代わりに、呼び出しレコード（JSON形式のリクエストとレスポンスに関する詳細が含まれるレコード）をLambda関数を含む別のサービスに送信することができます。この設定は任意であり、必須ではありません。2023年12月時点では、SQS、SNS、Amazon EventBridgeおよびLambda関数がサポートされています。

送信先の設定では、「On failure（失敗時）」、「On success（成功時）」の2つの条件で、実行後に呼び出すサービスを指定できます。非同期呼び出しの結果を呼び出し元へ通知する必要がある場合に設定するとよいでしょう。

### ●スケジュール起動

イベントソースでEventBridge（CloudWatch Events）を選択し、ルールタイプとして「スケジュール式」を選択することにより、Lambda関数を起動することができます。この機能を「スケジュール起動」といいます。時間指定はcron式またはrate式で記述できます。

## 2-3 Amazon DynamoDB

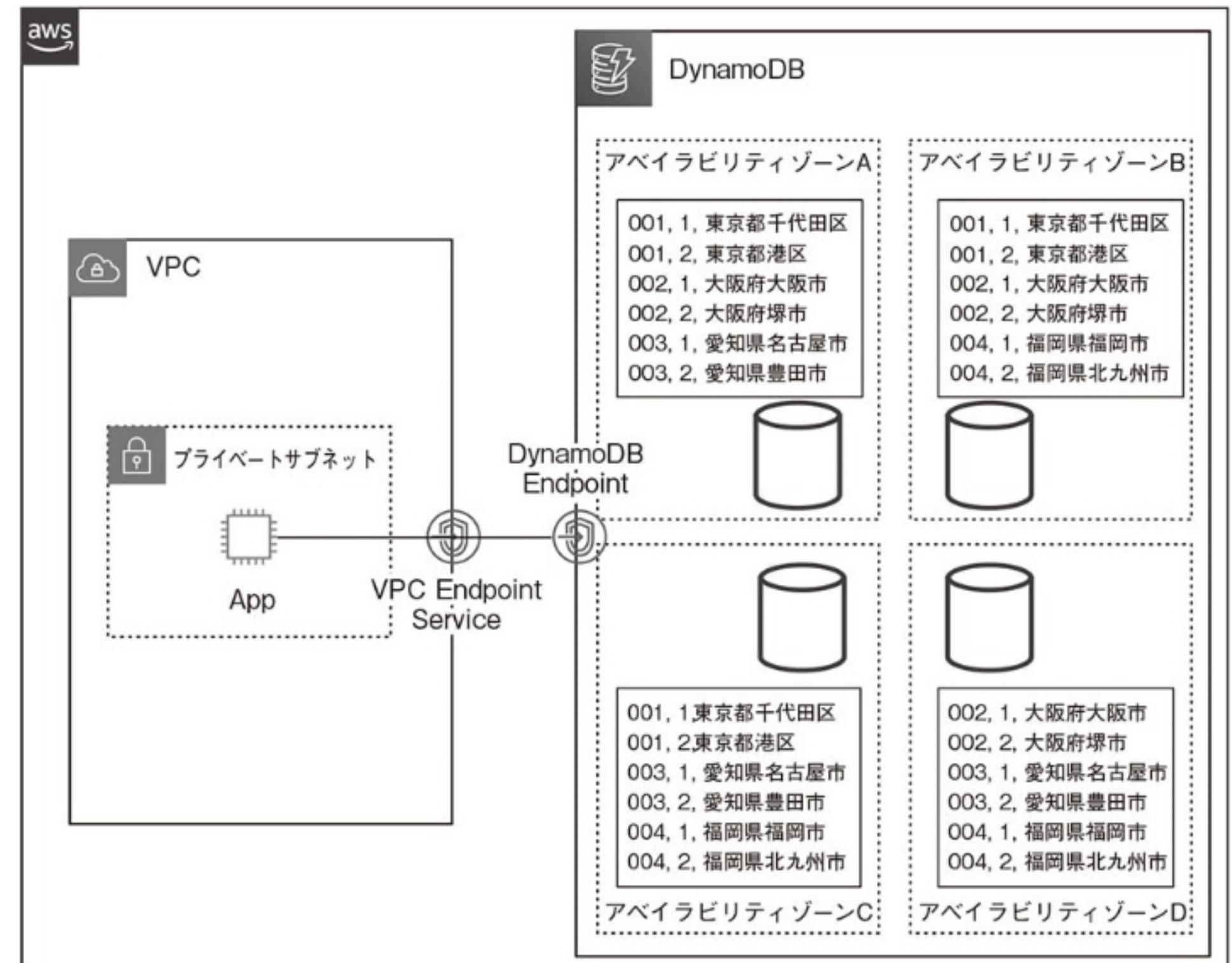
Amazon DynamoDBは、優れたスケーラビリティ・拡張性を持つNoSQLデータベースサービスです。DynamoDBはリージョンごとに構築されているデータベースであり、ユーザーはテーブルのキーなど最低限の設定を行うことで簡単に利用できます。試験対策に限らず、実際の開発で利用する機会も多いので、その機能や特性をよく理解しておきましょう。

### 1 DynamoDBのアーキテクチャ、データ配置と結果整合性

2

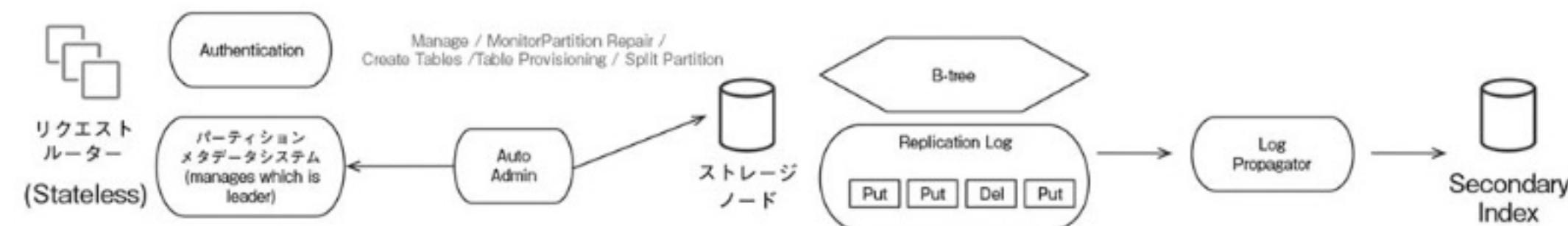
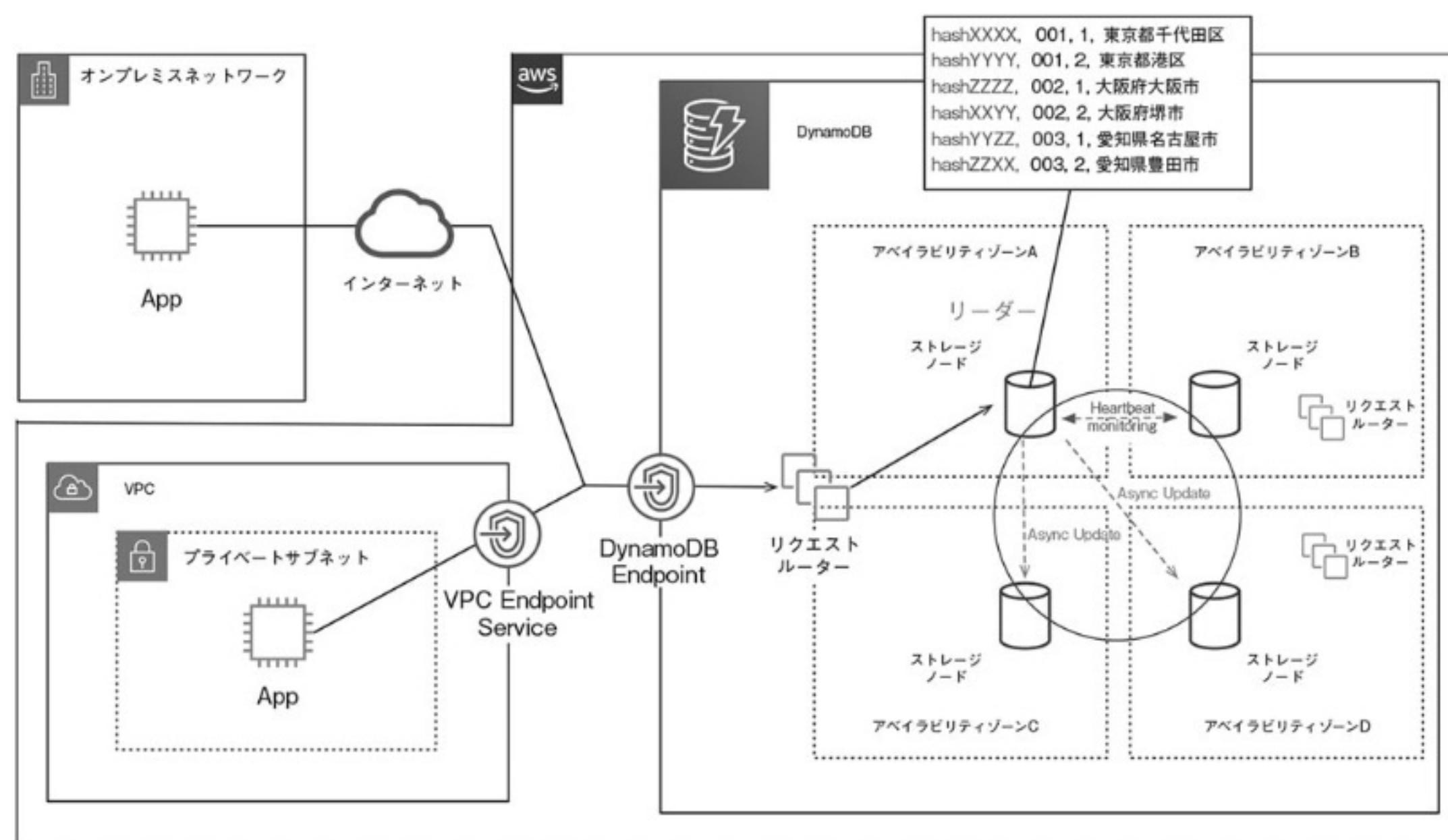
Amazon DynamoDBは、AWSフルマネージドなNoSQLデータベースを提供するサービスです。以下のイメージのようにリージョン単位でデータベースが構築され、3箇所の異なるアベイラビリティゾーンにデータがレプリケーションされます。

【DynamoDBのデータ配置イメージ】



※7 <https://aws.amazon.com/jp/blogs/news/lambda-functions-as-targets-for-application-load-balancers/>

DynamoDBでは、結果整合性などのデータベース特性上ユニークな特徴を試験で問われることがあります。実際に運用する場合にも役立ちますので、具体的にどのようなアーキテクチャ・ストレージ構成でデータを保持しているか概要を押さえておきましょう。DynamoDBでは、エンドポイントを通じて、データが保存されている各アベイラビリティゾーンにある正常なノードにアクセスします。これらのノードはストレージノードと呼ばますが、このストレージノードには、各テーブルごとにプライマリとなるデータを保存するリーダーが存在します。DynamoDBではパーティションメタデータと呼ばれるシステムでこのリーダーがどのストレージノードに配置されているかを管理しています。リーダーは一定の間隔で他のストレージノードが正常稼働しているか、Heartbeatを通じて監視します。また、パーティションメタデータはAutoAdminと呼ばれるプロセスによって管理されています。AutoAdminはストレージノードを管理し、テーブルの作成やデータ修復、再配置などのデータメンテナンス作業も行います。



ストレージノードのデータ自体は内部的にハッシュキーを持ち、B-Tree型で保存されています。データが更新されるとレプリケーションログに更新内容が保存され、このログをもとにプロパゲーターが後述するセカンダリインデックスなどのデータを更新する仕組みになっています。

インターネットの外やVPCにデプロイされたアプリケーションなどから届いたDynamoDBにあるデータへのアクセス要求は、内部的にリクエストルーターと呼ばれる

ステートレスなコンポーネントを通じて処理されます。各アベイラビリティゾーンで冗長化されている、このリクエストルーターではリクエストに適切なIAM認証情報が付与されているかを検証し、パーティションメタデータシステムを参照して要求されたデータへアクセスします。この際、アクセスするリクエスト種別に応じて挙動は異なります。Put/Delete系のリクエストではリーダーのデータを更新したのち、他のストレージノードに配置されている同一のデータを非同期更新します。参照のためのGet系のリクエストは、リーダーのみならず、いずれかのストレージノードにあるデータへアクセスします。そのため極めて短時間の間だけですが、データ更新後にストレージノード間で整合性がとれないケースが発生し得ます。ただし、DynamoDBでは、この振る舞いに関して、以下のようにオプションで選択することができます。

#### 【DynamoDBの結果整合性オプション】

読み込み/ 書き込み	結果整合性の オプション	動作
読み込み	結果整合性のある 読み込み	通常のオプション。リクエストルーターが、リーダーもしくはそれ以外のストレージノードのデータを返す。リクエストルーターがリーダー以外を選択した場合、低確率・短時間の間、不整合なデータ参照が発生する可能性がある
	強い読み込み整合性	リーダーを参照するオプション。必ず最新の値を返す
	トランザクション 読み込み	直列化可能分離レベル (SERIALIZABLE) でデータを読み取るオプション
書き込み	結果整合性のある 書き込み	通常のオプション。リーダーの更新が完了したのち正常応答
	トランザクション 書き込み	すべてのノードにおける、指定されたすべてのデータの書き込みが成功した場合正常応答

なお、DynamoDBのエンドポイントは、HTTPSでパブリックなネットワークを通じてアクセスできるようデフォルトで設定されています（VPC内からのアクセスでも、一度インターネットに出ることになります）。前ページの図のようにプライベートネットワーク内でアクセスを完結したい場合は、VPCエンドポイントを作成して、VPCにアタッチしておく必要があります。

## 2 DynamoDBのキーと属性・インデックス

Amazon DynamoDBではPaxosと呼ばれるアルゴリズムにより、各ノードと配置されるデータを決定しています。71ページの図では都道府県と市／区は1:Nの関係であり、1列目を都道府県を区別する「親キー」、2列目を市や区を区別する「子キー」としています。DynamoDBには、これらのキーに対して、以下のような名称で対応付けられるキーがあります。

### 【DynamoDBのキー要素】

親キー	子キー
パーティションキー (Partition Key)	ソートキー (Sort Key)

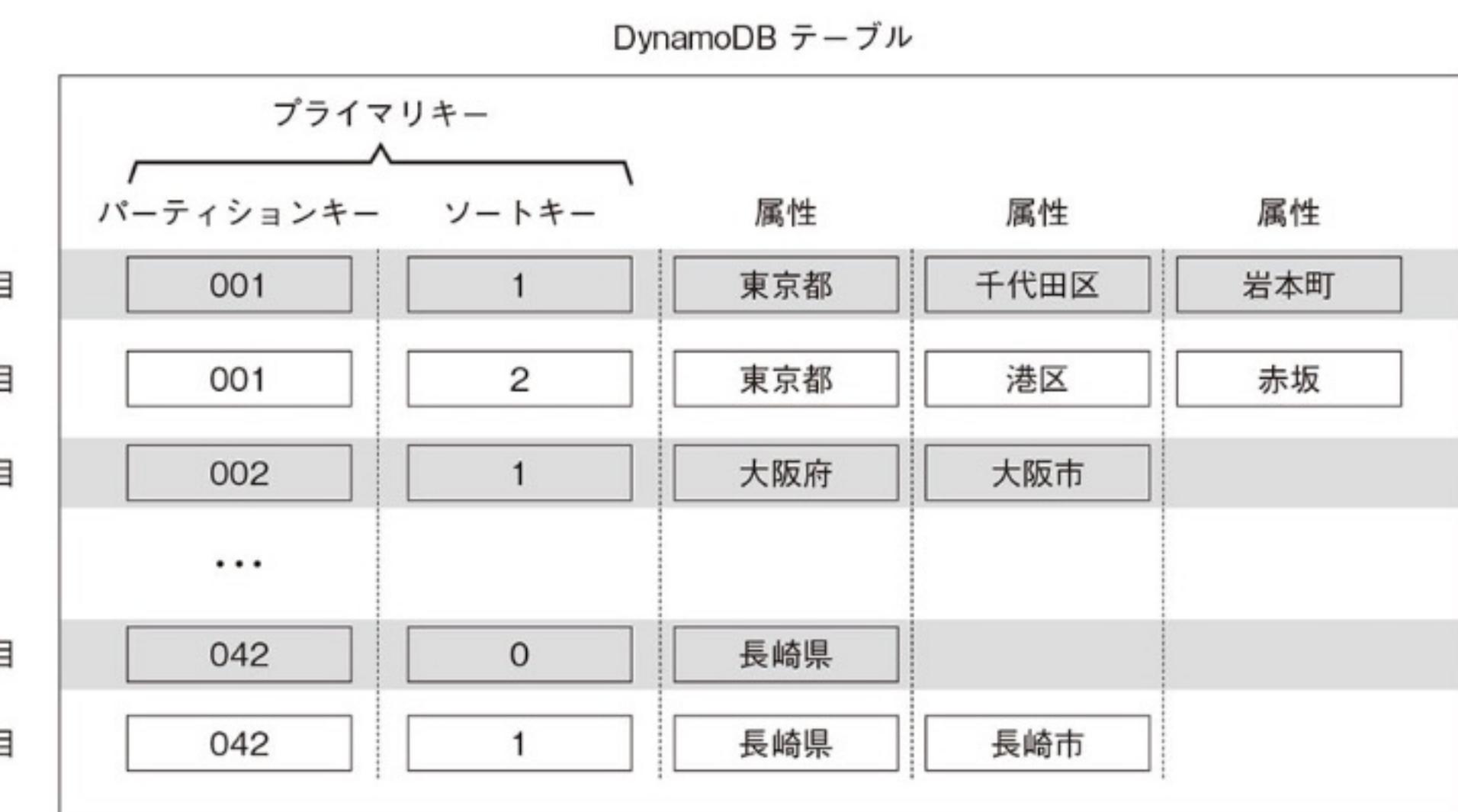
なお、パーティションキーは「ハッシュキー (Hash Key)」、ソートキーは「レンジキー (Range Key)」という名称で以前は呼ばれていました。また現在、パーティションキーとソートキーを合わせて「プライマリキー (Primary Key)」と呼ばれています。

この2つのキーに対して、以下のルールを押さえておきましょう。

- ・親キーで配置されるパーティションが決定する。
- ・パーティション内のデータ順序を決定する子キーを任意に設定できる。
- ・親キーと子キーの組み合わせでデータを一意に特定できるよう構成する。
- ・子キーを作成しない場合は親キーでデータを一意に特定できるようにする必要がある。
- ・キーおよび項目にはインデックスを設定できる。

DynamoDBは、キー以外の項目・属性の定義が必要ないスキーマレスのテーブル構成をとります。属性は項目ごとに異なっても問題ありません。パーティションキーでデータを配置するノードが決定し、ソートキーでノード内でのデータ順序が決定します。そのため、1つの注文と複数の注文商品明細のような1対多のデータ構造でもそのまま保存できます。キーの検索には関係演算子 ( $<$ 、 $>$ 、 $<=$ 、 $>=$ ) や等値系演算子 ( $==$ 、 $!=$ ) が利用可能です。なお、DynamoDBのデータサイズ上限は1項目あたり400KBのため、バイナリデータなどサイズの大きくなりがちなデータは保存できないので注意が必要です。

### 【DynamoDBのテーブル内データのイメージ】



データを一意に識別するためのプライマリキーは、パーティションキー単独あるいは、パーティションキーとソートキーとの組み合わせで構成されます。プライマリキーに該当する属性以外は事前に定義しておく必要はありません。基本的にプライマリキーを用いて検索しますが、任意の属性をパーティションキーとするグローバルセカンダリインデックス (GlobalSecondaryIndex : GSI) と、パーティションキーと別の属性とを組み合わせて作成するローカルセカンダリインデックス (LocalSecondaryIndex : LSI) が利用できます。上記の図の例では、GSIとして「東京都」などの都道府県別の検索を、LSIとしてパーティションキー「001」と「岩本町」などの町単位検索を実現できます。

ただし、セカンダリインデックスを使用するには、次項のキャパシティユニットやインデックスを保存するためのストレージが別途必要になるので注意が必要です。ローカルセカンダリインデックスはテーブルとインデックスの合計サイズが各パーティションキーごとに10GBまで制限されます。



### RDBと比較した場合のDynamoDBの機能的差分

これまでの説明にあるとおり、DynamoDBはノードごとにデータが分散するアーキテクチャとなっています。そのため、以下のような機能的な差分が生じます。

#### 【RDBと比較した場合のDynamoDBの制約】

機能的差分	理由
テーブル間の結合ができない	データが分散して配置されているので、データ同士を結合して射影するといった操作はできない。アプリケーションで適宜アグリゲーション（集約処理）する必要がある。
外部キーがない	データベースの特性上、スキーマが存在しないため、キーはパーティションキーとソートキーに限定される。
条件指定は基本的にプライマリキー以外は使用できない	データが分散して配置されているので、プライマリキー以外で検索をかけることができない。それ以外の項目で検索が必要な場合は、インデックスを作成する、もしくはソートキーを指定することができる（ただし、ソートキーのみの検索は性能上問題が出る可能性がある）。
副問い合わせができる	データが分散して配置されるので、検索結果のデータを条件とすることはできない。
GROUP BYなどの集約関数が存在しない	データが分散して配置されるので、集約に必要なデータが検索時に足りない。
OR、NOTなどの論理演算子はなくANDのみ	データベースの性質上、サポートしないが、OR条件はフィルタなどの機能により代替可能。

RDBで実現できる機能を使いたい場合は、RDBを導入すべきです。コネクションプールなど多数のアプリケーションからのデータアクセスがボトルネックとなる場合や、需要の予測が難しく後々拡張したいケース、グローバルな複数のデータセンタにまたがってデータを共通化 / レプリケートしたいケース、トランザクションが一定時間に集中するECサイトの注文処理やIoTセンサーデータといったリードレプリカでは対応が難しい書き込みが多いケースなど、その特性を生かしたユースケースに対して、DynamoDBの導入を検討すべきです。

## 3 キャパシティユニット

DynamoDBの課金体系には2つのオプションがあります。リクエスト数に応じたオンデマンドキャパシティモードと、テーブル単位で読み書きのパフォーマンスをスループットとして定義するプロビジョンドモードです。このときの読み込みのスループットを「読み込みキャパシティユニット（Read Capacity Units : RCUs）」と呼び、書き込みスループットを「書き込みキャパシティユニット（Write Capacity Units : WCUs）」と呼びます。RCUsとWCUsはそれぞれ以下のように定義されています。

#### 【キャパシティユニットの定義】

RCUs	WCUs
・1秒あたりの読み込み項目数 × 項目のサイズ（4KBまでを1ブロックとして計算）	・1秒あたりの書き込み項目数 × 項目のサイズ（1KBまでを1ブロックとして計算）
・結果整合性がある読み込みの場合はスループットが2倍になる	・トランザクション書き込みの場合はスループットが半分になる（パフォーマンスの負荷が倍になる）
・トランザクション読み込みの場合はスループットが半分になる（パフォーマンスの負荷が倍になる）	

例えば、3KBのデータを読み込むリクエストが毎秒1,000回あるのであれば、1,000RCUsです。仮に6KBのデータだとすると、ブロック単位の4KBを超えているので、2ブロック（8KB）が必要です。すなわち、2倍のRCUsとして計算し、2,000RCUsとなります。結果整合性のある読み込みの場合、スループットが2倍になるのでRCUsは半分で済み、3KBだと500RCUs、6KBだと1,000RCUsです。逆にトランザクション読み込みの場合は、スループットが半分となり、必要なRCUsは3KBだと2,000RCUs、6KBだと4,000RCUsと計算されます。書き込みも計算方法は同様で、3KBのデータを書き込むリクエストが毎秒10回あるとすると、30WCUsです。トランザクション書き込みはスループットが半分になるので、倍の60WCUsが必要になります。

## 4 その他の特徴

そのほか、DynamoDBには以下のようないくつかの特徴があります。各特徴について以降、詳細を説明していきます。

## 【DynamoDBの特徴】

特徴	説明
フィルタを使った読み込み	クエリやスキャンした結果にフィルタ式を設定し、結果を絞り込む機能。
条件付き書き込み	データの有無や項目値に応じた条件を設定し、該当したとき更新を行う機能。
TTL (TimeToLive)	データ属性にTTLを設定し、有効期限を過ぎると自動的にテーブルからデータを削除する機能。
DAX (DynamoDB Accelerator)	マルチアベイラビリティゾーン構成で自動フェイルオーバー機能を持つインメモリキャッシュ。
DynamoDB Streams	DynamoDBで行われたデータの追加・変更・削除履歴を記録する機能。更新前のデータを残す、更新前後のデータ双方を残すなどオプションで選択できる。
DynamoDB Trigger	DynamoDBへのデータ更新をきっかけにLambda関数を実行する機能。別のテーブルの更新や監査ログの保存、プッシュ通知などの用途で利用可能。
グローバルテーブル	リージョンをまたいでDynamoDBを構築する機能。リージョンを越えて1つのテーブルが構築されるわけではなく、DynamoDBのレプリカが別のリージョンに構成されるイメージ。更新は双方で同期される。

## ● フィルタを使った読み込み・条件付き書き込み

DynamoDBに対して読み込み・書き込み・削除といった操作を行いたいときは、下記のAPIを使用します。

## 【DynamoDBの主なAPI】

API	説明
GetItem	テーブルから単一の項目を取り出します。目的の項目のプライマリキーを指定する必要があります。項目全体またはその属性のサブセットのみを取り出すことができます。
BatchGetItem	1つ以上のテーブルから最大100個の項目を取り出します。GetItemを複数回呼び出すよりも効率的にデータ取得を行うことができます。
Query	特定のパーティションキーを持つ、すべての項目を取り出します。パーティションキーの値を指定し、項目全体またはその属性のサブセットの一部を取り出すことができます。また、ソートキーの値に条件を適用し、同じパーティションキーがあるデータのサブセットだけを取り出すオプションも利用できます。このオペレーションは、テーブルにパーティションキーとソートキーの両方がある場合にのみ使用可能です。また、インデックスにパーティションキーとソートキーの両方がある場合、インデックスでも使用できます。

API	説明
Scan	指定されたテーブルまたはインデックスのすべての項目を取り出します。項目全体またはその属性のサブセットのみを取り出すことができます。
PutItem	項目を作成します。プライマリキーは一部の属性ではなく、すべての属性を指定する必要があります。例えば、テーブルに複合プライマリキーがある場合、パーティションキーおよびソートキーの値を提供する必要があります。同じキーを持つ項目がテーブルにすでに存在する場合は、新しい項目に置き換えられます。
UpdateItem	項目の1つ以上の属性を変更します。変更する項目のプライマリキーを指定する必要があります。新しい属性を追加したり、既存の属性を変更または削除したりできます。ユーザ定義の条件を満たす場合にのみ更新が成功するように、条件付きの更新を実行できます。オプションで、アトミックカウンタを実装できます。このカウンタは、他の書き込みリクエストを妨害することなく、数値属性をインクリメントまたはデクリメントします。
DeleteItem	テーブルから単一の項目を削除します。削除する項目のプライマリキーを指定する必要があります。
BatchWriteItem	1つ以上のテーブルから最大25個の項目を更新・削除します。UpdateItemやDeleteItemを複数回呼び出すよりも効率的にデータ削除を行うことができます。
TransactGetItems	最大100個のGetアクションをまとめてグループ化して実行するオペレーションで、分離レベルは「直列可能分離レベル(SERIALIZABLE)」です。実行中にTransactWriteItemsと競合した場合、TransactionCanceledExceptionエラーで失敗します。
TransactWriteItems	最大100個の書き込みアクションをまとめてグループ化して、All or Nothingに実行するオペレーションです。実行中に別のTransactWriteItemsと競合した場合、TransactionCanceledExceptionエラーで失敗します。このオペレーションは、含まれるアクションがすべて完了するか、そうでない場合は変更がまったく行われないという点でBatchWriteItemとは異なります。

QueryやScanなど、データを取得するAPIに対して、フィルタを使った絞り込みが可能ですが、QueryでのフィルタはAWS SDKでも実装できますが、ここでは簡単に実行できるAWS CLIを用いた例で解説します。「key-condition-expression」を使って対象データを絞り込み、以下のようにフィルタとなる条件の表現を指定します。

```
aws dynamodb query ¥
--table-name Thread ¥
--key-condition-expression "ForumName = :fn and Subject = :sub" ¥
--filter-expression "#v >= :num" ¥
--expression-attribute-names '{"#v": "Views"}' ¥
--expression-attribute-values { ":fn": {"S": "Amazon DynamoDB"}, ":sub": {"S": "DynamoDB Thread 1"}, ":num": {"N": "3"} }
```

上記の例では、Threadテーブルに対し、ForumName（パーティションキー）、Subject（ソートキー）で特定の属性値を持ち、一定数以上のViewsを持つ場合、データが返却されます<sup>※8</sup>。なお、ViewsはDynamoDBでは予約語<sup>※9</sup>であるため、プレースホルダ「#v」を使用しています。また、Scanでのフィルタは、以下の形式で実行されます。

```
aws dynamodb scan ¥
--table-name Thread ¥
--filter-expression "LastPostedBy = :name" ¥
--expression-attribute-values '{":name": {"S": "User A"}}'
```

上記の例では、Threadテーブルでスキャンしたデータに対し、「LastPostedBy」で指定された値を持つデータがフィルタされます<sup>※10</sup>。

DynamoDBでデータを取得する際には、以下の点で留意が必要です。

- ・一般に、Scanはテーブル全体またはセカンダリインデックスがスキャンされてしまうため、データが大きくなるとキャパシティユニットが消費され、パフォーマンスが低下します。
- ・1回のScanオペレーションで、最大1MBのデータを取得できます（page-sizeパラメータでサイズ指定することも可能ですが）。サイズを超えるデータが存在する場合は、取得結果にLastEvaluatedKeyパラメータが含まれ、この値を含めて、Scan APIを再実行することで、残りのデータが取得できます。
- ・QueryおよびScanオペレーションは、結果で返される項目数を制限することができます。フィルタ式を評価する前に、APIが返す項目の最大数をLimitパラメータに設定します。
- ・DynamoDBは結果整合性のある読み込みをデフォルトで行います。強い読み込み整合性が必要な場合は、ConsistentReadパラメータをTrueに設定します。

**※8** [https://docs.aws.amazon.com/ja\\_jp/amazondynamodb/latest/developerguide/Query.html#Query.FilterExpression](https://docs.aws.amazon.com/ja_jp/amazondynamodb/latest/developerguide/Query.html#Query.FilterExpression)

**※9** [https://docs.aws.amazon.com/ja\\_jp/amazondynamodb/latest/developerguide/ReservedWords.html](https://docs.aws.amazon.com/ja_jp/amazondynamodb/latest/developerguide/ReservedWords.html)

**※10** [https://docs.aws.amazon.com/ja\\_jp/amazondynamodb/latest/developerguide/Scan.html#Scan.FilterExpression](https://docs.aws.amazon.com/ja_jp/amazondynamodb/latest/developerguide/Scan.html#Scan.FilterExpression)

また、DynamoDBでは、以下のように条件付き書き込みをサポートします。以下は、Idが1である製品カタログ（ProductCatalog）の、Price属性が10の場合、8に更新する例です。

```
aws dynamodb update-item ¥
--table-name ProductCatalog ¥
--key '{"Id":{"N":"1"}}' ¥
--update-expression "SET Price = :newval" ¥
--condition-expression "Price = :currval" ¥
--expression-attribute-values { ":newval": {"N": "8"}, ":currval": {"N": "10"} }
```

条件付き書き込みは、指定する条件が更新前後で変化するよう実装した場合に、幂等性（べきとうせい。何回実行しても同じ結果が得られること）が保たれます。

例えば、項目のPriceが20である場合のみ、3増加させるUpdateItemリクエストを実行する条件付き書き込みを考えてみましょう。リクエストを送信した後、その結果を得る前にネットワークエラーが発生して、リクエストが成功したかどうか不明だったとします。この条件付き書き込みは幂等性が担保されたオペレーションであるため、同じUpdateItemリクエストを再試行できます。なぜなら、DynamoDBは、現在のPriceが20である場合のみ項目を更新するからです。更新が成功して23になっていたとすると、再送したリクエストが重複処理されることはありません。



データを操作する各APIの内容やオプション・制約を押さえておくようにしましょう。

### ●TimeToLive (TTL)

Amazon DynamoDBは項目に対して、有効期限（TimeToLive: TTL）をサポートしています。TTLを使用すると、指定されたタイムスタンプの日時に、DynamoDBが書き込みスループットを消費することなく、テーブルから項目を削除します。コンソール上からでも実行できますが、以下のように、テーブルに対してTTLを有効化する属性を指定します。

```
aws dynamodb update-time-to-live --table-name TTLExample --time-to-live-specification "Enabled=true,AttributeName=ttl"
```

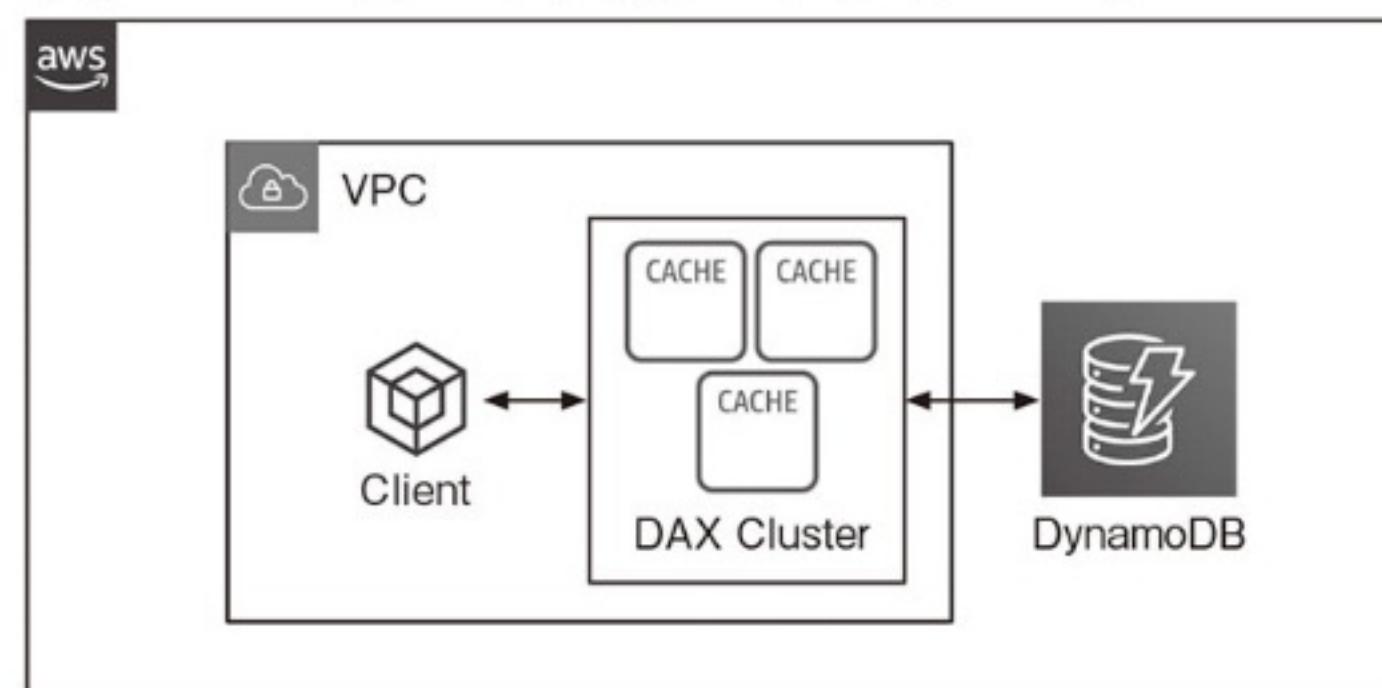
この属性（ttl）に対して、有効な日付データを追加しておくと、そのタイミングで削除されます。以下はBashシェルスクリプトで、5日後に項目削除するデータを追加する例です。

```
EXP='date -d '+5 days' +%s'
aws dynamodb put-item --table-name "TTLExample" --item '{"id": {"N": "1"}, "ttl": {"N": "'$EXP'"}}'
```

## ●DAX (DynamoDB Accelerator)

DAXはVPC内でキャッシングクラスタを構築し、DynamoDBのデータへ高速にアクセスできるキャッシングサービスです。リアルタイム入札やソーシャルゲーム、トレーディングなど可能な限り迅速な読み込み応答時間が必要とするユースケースに適しています。

【DynamoDBのテーブル内データのイメージ】



アプリケーションはDAXクライアントから、クラスタを通じてDynamoDBに透過的にアクセスします。DAXでは読み込みAPIとして、GetItem、BatchGetItem、Query、Scan等がサポートされています。結果整合性のある読み込みの場合、DAXクラスタにキャッシングデータがあればそれを返却し（キャッシングヒット）、なければ（キャッシングミス）DynamoDBへアクセスしてデータを取得したのち、キャッシングデータとして書き込んでクライアントへ返却します。

書き込みAPIは、BatchWriteItem、UpdateItem、DeleteItem、PutItem等がサポートされており、いずれもライトスルーで処理（DynamoDBへ書き込まれたのち、キャッシングデータを更新）して正常終了します。クラスタはユーザが任意の台数で構成できますが、高可用性を確保するために、マルチアベイラビリティゾーンで3箇所以上のノードで構成したほうがよいでしょう。DynamoDB同様、クラスタにもエンドポイントが割り当てられるため、アプリケーションはエンドポイントを通じて常に正常なノードにアクセスできます。

## ●DynamoDB Streams / DynamoDB Trigger

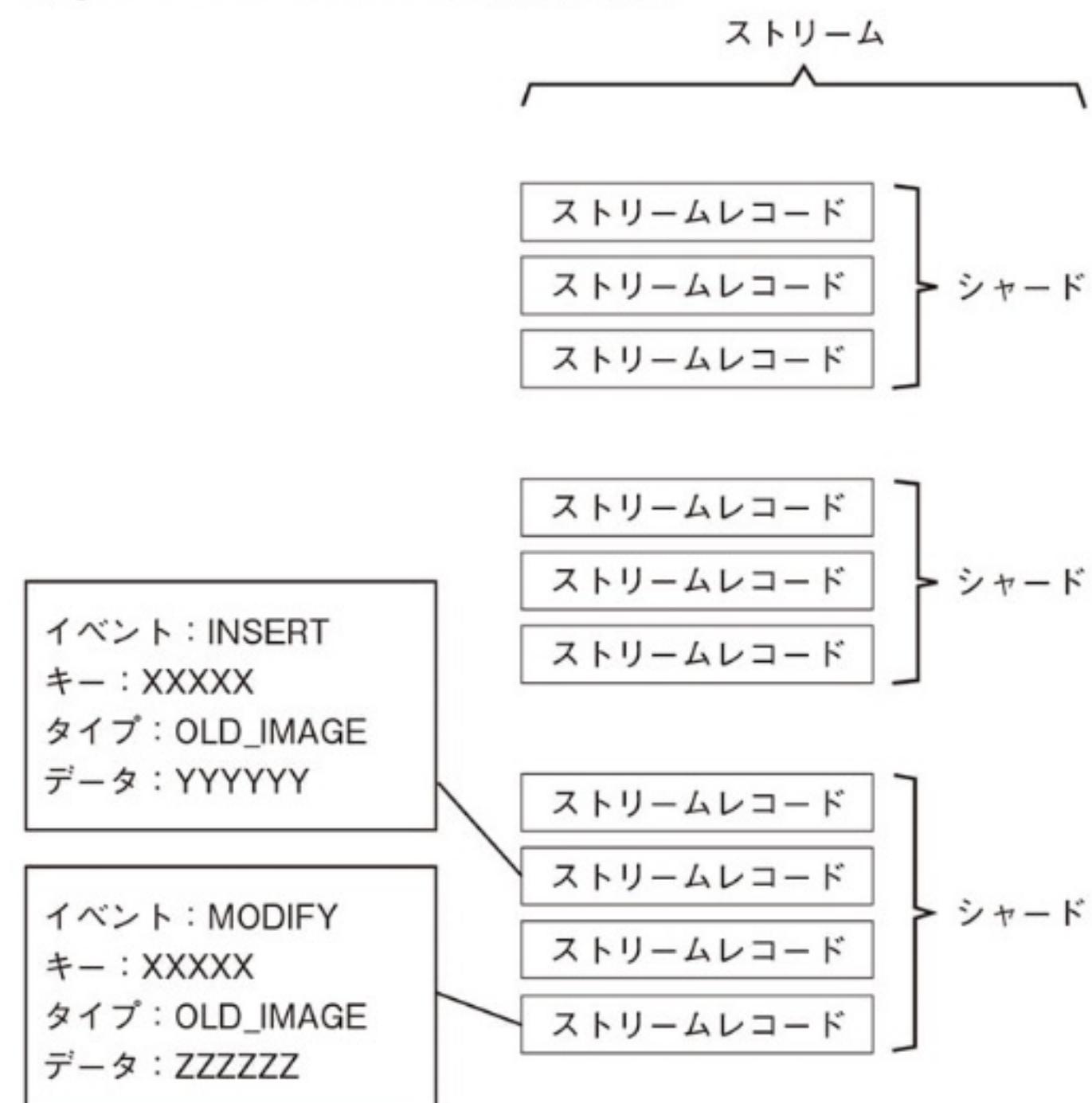
DynamoDB Streamsはテーブル内で項目の変更に関するデータを記録しておく機能です。データが作成・更新・削除されたときに、項目のプライマリキー属性に加えて変更前後のデータがストリームレコードとして書き込まれます。書き込みは以下の4つのオプションから選択可能です。

【DynamoDB Streamsの書き込みオプション】

オプション	説明
KEYS_ONLY	変更された項目のキー属性のみを保存する
NEW_IMAGE	変更後の項目データ全体を保存する
OLD_IMAGE	変更前の項目データ全体を保存する
NEW_AND_OLD_IMAGE	変更前後の項目データ全体を保存する

このデータは暗号化がサポートされており、最大24時間保存されます。ユーザはDynamoDB Streams用のエンドポイントを使ってデータにアクセスできます。データは変更内容を表すストリームレコードとして、ある程度の時間範囲や異なるパーティションキーの単位でグループ（シャード）化されています。

【DynamoDB Streamsの構成要素】



DynamoDB Streamsでは、以下のAPIがサポートされています。

### 【DynamoDB Streams API】

API	説明
ListStreams	アカウントおよびエンドポイントのストリームの一覧を返す
DescribeStream	特定のストリームに属するシャードの一覧など詳細な情報を返す
GetShardIterator	シャード内のストリームレコードの場所を指示するイテレータを返す
GetRecord	指定したシャードイテレータが持つストリームレコードを返す

DynamoDB StreamsはAWS LambdaおよびAmazon Kinesisと統合されており、LambdaもしくはKCL (Kinesis Client Library) がStream APIを使ってポーリングを行い、変更を検知後、ストリームレコードデータを取得して処理を実行できます (DynamoDB Trigger)。

### ●グローバルテーブル

通常、DynamoDBはリージョンごとにデータベースが分かれていますが、複数のリージョンにまたがる共通のデータベース「グローバルテーブル」を構築することもできます。この機能を利用することで、複数のリージョンで展開されるアプリケーションでも共通のデータへアクセスできます。また、異なるAWSリージョンでビジネス展開する場合に、データをバッチ処理転送することなく、リアルタイムでの集約・解析を実現することが可能です。ただし、本節の冒頭で解説したような、複数のリージョンで多数のノードでデータベースが構築されているというより、別のリージョンで複数のレプリカテーブルで構成される形式のため、アプリケーションがほぼ同時に異なるリージョンで同じ項目を更新する場合、競合が発生する可能性があることに注意が必要です。この場合、後勝ちとなり最後の書き込みが反映されます。

## 2-4 AWS Step Functions

AWS Step Functions (Step Functions) は、分散アプリケーションとマイクロサービスを「ステートマシン」と呼ばれるサーバレスなワークフローを使って、制御 (オーケストレーション)・可視化するサービスです。

Step Functionsは、LambdaをはじめとするほかのAWSサービスを呼び出すことができ、処理の順序を定義や並列実行、条件分岐させる、失敗時のリトライや例外・エラー処理など、さまざまな制御を行えます。「アプリケーションフロー処理」や「機械学習/ETLデータパイプライン」のユースケースパターンで利用されます。

### 【サーバレスでよく使われるサービスとユースケースパターン】

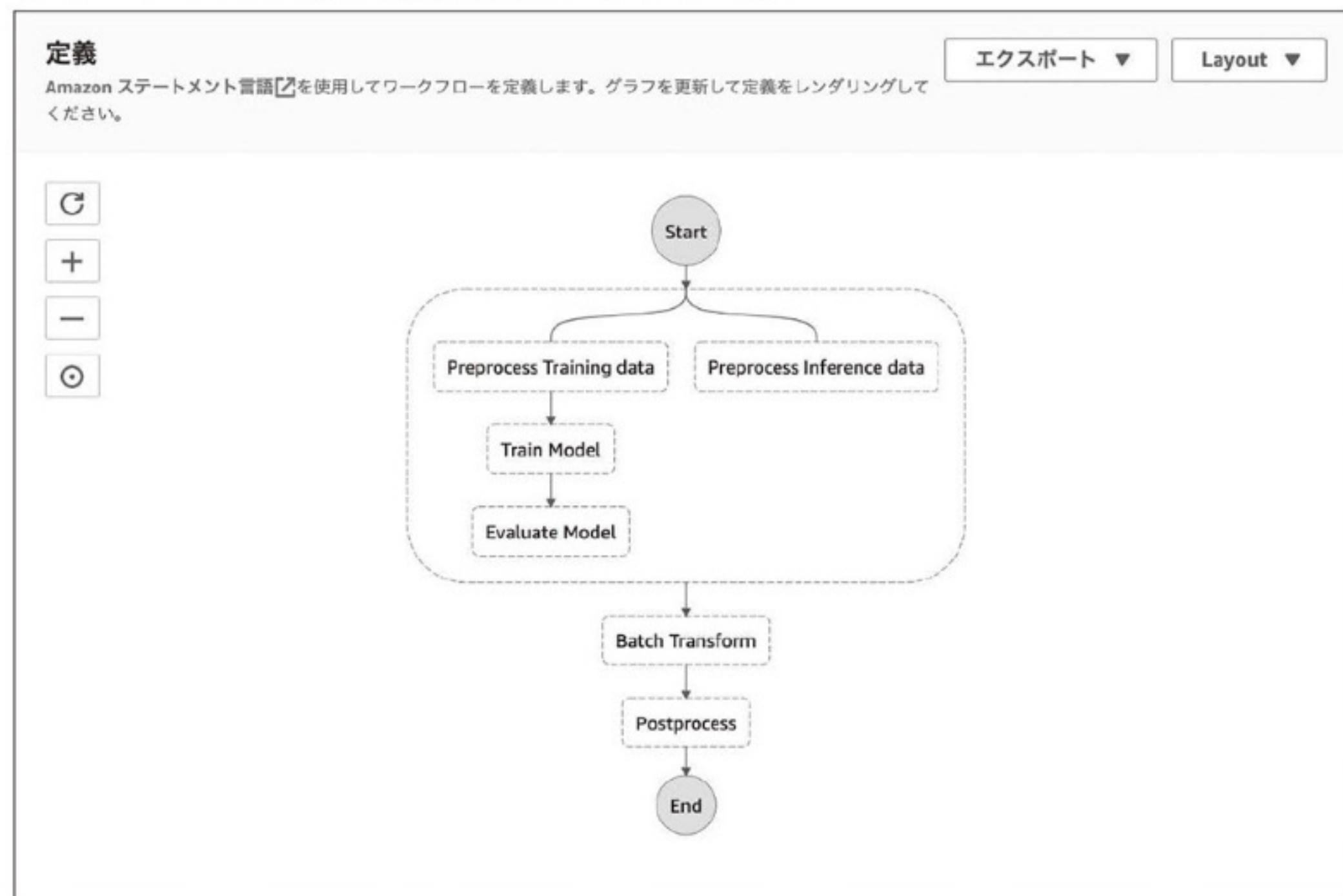
ユースケース名	概要	主な利用サービス
アプリケーションフロー処理	• Step Functionsを利用して、Lambda関数などサーバレスサービスのコンポーネントの実行順序、並列処理、条件分岐などの状態遷移をワークフローとして定義して制御する。	• Step Functions • Lambda • Fargate • DynamoDB • SNS • SQS
機械学習/ETLデータパイプライン	• 「アプリケーションフロー処理」と本質的には同一である。 • 機械学習ワークフローやデータ分析におけるETL処理を対象とし、Step Functionsによりワークフローを定義して実行を制御する。	• Step Functions • Lambda • ECS/EKS/Fargate • Batch • Glue • SageMaker

## 1 ステートマシン

ステートマシンは「ステート（状態）」という要素で構成されます。ASL (Amazon States Language) と呼ばれるJSON形式の独自言語を使って、ステートの状態遷移をワークフローとして定義します。

下記に例として、簡単な機械学習ワークフローを実行するステートマシンを示します。

### 【ステートマシンの例（機械学習ワークフロー）】



上図において、小さな破線で囲まれたオブジェクトがステートです。「Preprocess Training data」から始まる3つの「Task」と、「Preprocess Inference data」の1つのTaskがParallelで実行（並列実行）されるように定義しています。Parallelで実行する処理が完了した後に、「Batch Transform」と「Postprocess」の2つのタスクを直列で実行します。

### ●ステートマシンの種類

ステートマシンには、「標準ワークフロー」と「Expressワークフロー」の2種類があります。

標準ワークフローは、最大実行時間が1年と長いため、さまざまな種類のワークフローを実装できます。例えば、ユースケースパターンとして紹介した「アプリケーションフロー処理」や「機械学習/ETLデータパイプライン」の実装などに利用できます。

Expressワークフローは、最大実行時間が5分と短い反面、1秒あたりに起動できるステートマシン数（実行開始レート）が高く、1秒あたりの状態遷移できる数（状態遷移レート）がほぼ無制限に設定されているワークフローです。Expressワークフローは、さらに同期および非同期Expressワークフローの2種類が用意されています。IoTデータの取り込みやストリーミングデータの処理や変換などの短時間に大量のデータを処理するようなワークフローに向いています。

「標準ワークフロー」と「Expressワークフロー」の主な違いを下記にまとめます。

### 【「標準ワークフロー」と「Expressワークフロー」の主な違い】

項目	標準ワークフロー	Expressワークフロー
最大実行時間	1年	5分
実行開始レート	毎秒2,000以上	毎秒100,000以上
状態遷移レート	1アカウントあたり4,000以上	ほぼ無制限
料金	状態遷移ごとの従量課金	実行回数、実行時間、消費されたメモリに対する従量課金
実行履歴	•Step Functions API •AWS マネジメントコンソール •CloudWatch Logs	•CloudWatch Logs
実行セマンティクス	1回だけ実行	最低1回実行
サービス統合	すべてのサービス統合とパターンをサポート	すべてのサービス統合をサポート。ジョブ実行 (.sync) パターンまたはコールバック (.waitForTaskToken) パターンはサポートしない
Activity のサポート	○	×



#### 標準ワークフローと Express ワークフロー

- 上表のすべての項目を覚える必要はありません。標準ワークフローと Express ワークフローの選定時における判断材料としてください。
- ステートマシンで扱うワークロードが短時間に大量のデータを処理するような場合は Express ワークフローを、それ以外の場合は標準ワークフローを選択することを目安にするとよいでしょう。

本書の執筆時点（2023年12月）では、Step Functionsのステートマシンは、下記の AWSサービスと統合されています。

•Lambda	•Batch	•DynamoDB	•ECS/Fargate
•SNS	•SQS	•Glue	•SageMaker
•EMR	•EMR on EKS	•CodeBuild	•Athena
•EKS <sup>*11</sup>	•API Gateway	•Glue DataBrew	•EventBridge <sup>*12</sup>
•Step Functions	•SDK <sup>*13</sup>		

統合（呼び出し）のパターンは、「リクエストレスポンス」「ジョブの実行 (.sync)」「コールバックまで待機 (.waitForTaskToken)」の3種類があります。ワークフローの種類に

\*11 <https://aws.amazon.com/jp/blogs/news/introducing-aws-step-functions-integration-with-amazon-eks-jp/>

\*12 <https://aws.amazon.com/jp/blogs/news/introducing-the-amazon-eventbridge-service-integration-for-aws-step-functions/>

\*13 <https://aws.amazon.com/jp/blogs/news/now-aws-step-functions-supports-200-aws-services-to-enable-easier-workflow-automation/>

よってサポートされる統合パターンに違いがありますので、詳細は開発者ガイドを確認してください<sup>※14</sup>。

なお、上記に挙げたAWSサービスは、それぞれのワークフローから直接呼び出し可能なサービスです。上記にないAWSサービスでも、LambdaやECS/Fargateを使って独自に実装することで、ワークフローから呼び出せるようになります。

## 2 ASLによるステートマシンとステートの定義

ステートマシンはASLと呼ばれるJSON形式の独自言語で定義します。下記にASLの例を示します。

```
{
  "Comment": "An example of the Amazon States Language for
notification on an AWS Fargate task completion",
  "StartAt": "Run Fargate Task",
  "TimeoutSeconds": 3600,
  "States": {
    "Run Fargate Task": {
      "Type": "Task",
      "Resource": "arn:aws:states:::ecs:runTask.sync",
      "Parameters": {
        "LaunchType": "FARGATE",
        "Cluster": "arn:aws:ecs:ap-northeast-1:123456789012:cluster/
FargateTaskNotification-ECSCluster-VHLR20IF9IMP",
        "TaskDefinition": "arn:aws:ecs:ap-northeast-1:123456789012:task-
definition/FargateTaskNotification-ECSTaskDefinition-13Y0JT8Z2LY5Q:1",
        "NetworkConfiguration": {
          "AwsVpcConfiguration": {
            "Subnets": [
              "subnet-07e1ad3abcfce6758",
              "subnet-04782e7f34ae3efdb"
            ],
            "AssignPublicIp": "ENABLED"
          }
        }
      },
      "Next": "Notify Success",
      "Catch": [
        {

```

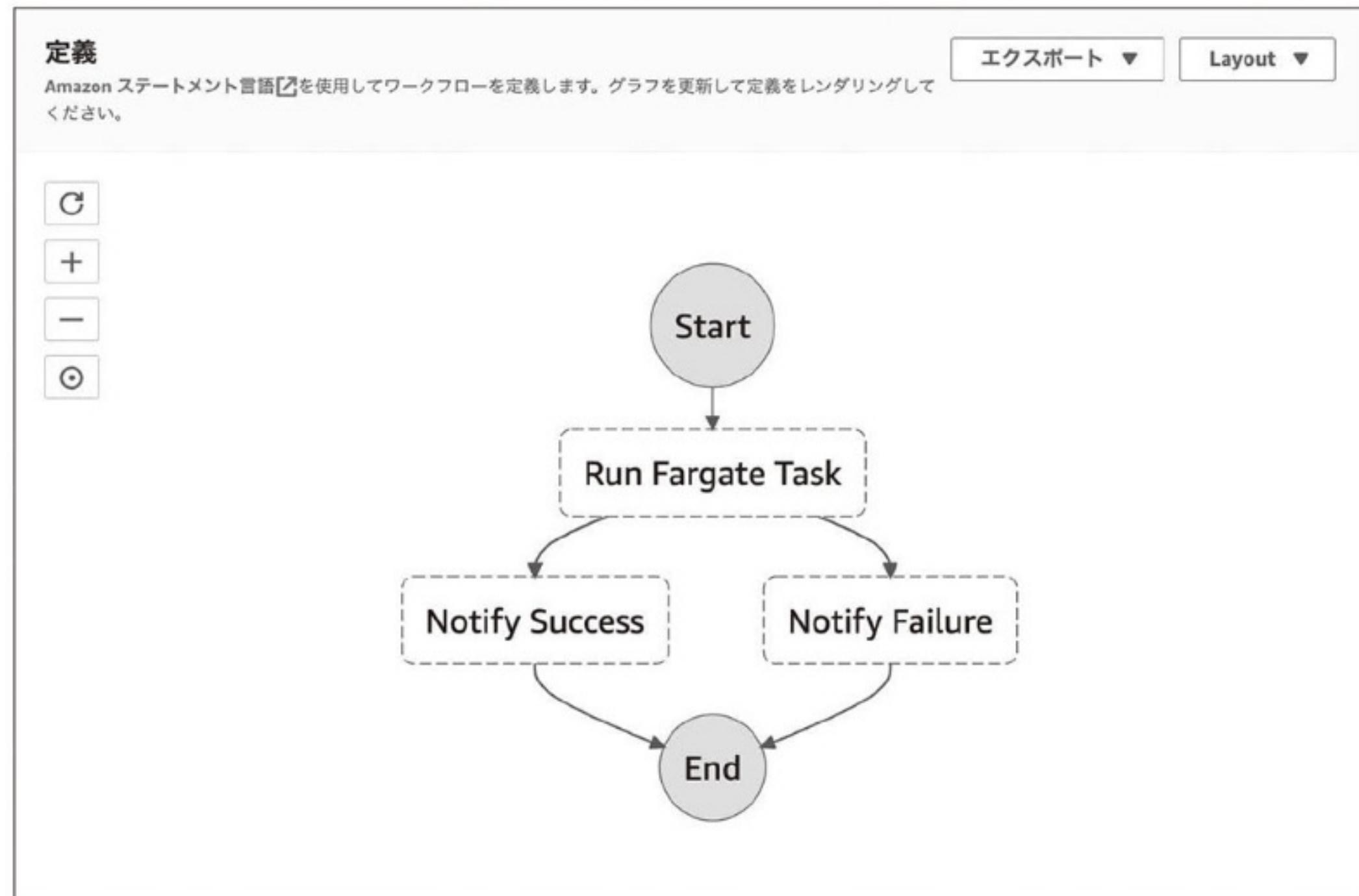
```
          "ErrorEquals": [ "States.ALL" ],
          "Next": "Notify Failure"
        }
      ],
      "Notify Success": {
        "Type": "Task",
        "Resource": "arn:aws:states:::sns:publish",
        "Parameters": {
          "Message": "AWS Fargate Task started by Step Functions
succeeded",
          "TopicArn": "arn:aws:sns:ap-northeast-1:123456789012:FargateTa
skNotification-SNSTopic-1XYW5YD5V0M7C"
        },
        "End": true
      },
      "Notify Failure": {
        "Type": "Task",
        "Resource": "arn:aws:states:::sns:publish",
        "Parameters": {
          "Message": "AWS Fargate Task started by Step Functions
failed",
          "TopicArn": "arn:aws:sns:ap-northeast-1:123456789012:FargateTa
skNotification-SNSTopic-1XYW5YD5V0M7C"
        },
        "End": true
      }
    }
  }
}
```

このASLから、ECSタスクをFargate上で実行し、処理の成否をSNSで通知するステートマシンが作成されます<sup>※15</sup>。

※14 [https://docs.aws.amazon.com/ja\\_jp/step-functions/latest/dg/concepts-service-integrations.html](https://docs.aws.amazon.com/ja_jp/step-functions/latest/dg/concepts-service-integrations.html)

※15 AWS Step Functionsの開発者ガイド ([https://docs.aws.amazon.com/ja\\_jp/AmazonECS/latest/userguide/task\\_definitions.html](https://docs.aws.amazon.com/ja_jp/AmazonECS/latest/userguide/task_definitions.html)) より引用。

### 【ステートマシンの例 (ECSタスクの実行結果の成否をSNSで通知するステートマシン)】



### ●ステートの定義

ステートは、下記に示す8種類のタイプが用意されています。

ステートタイプ	役割
Task	•ステートマシンによって実行する作業を定義する。 •Lambda関数、Activity、AWSサービスを指定できる。
Choice	ステートマシンに条件分岐を設定する。
Fail	ステートマシンの実行を失敗で停止させる。
Succeed	ステートマシンの実行を成功で正常停止させる。
Pass	何の処理もせずに入力を出力に渡す。
Wait	指定された時間だけ待機して、Nextフィールドに指定したステートに遷移する。
Parallel	ステートを並列に実行する。
Map	•入力配列の要素ごとに反復処理や並列処理を実行する。 •Parallelは同じ入力を使って並列に処理を実行するが、Mapの入力は配列となるので、配列の要素に応じて入力を変えて実行できる。

ステートタイプにはそれぞれ設定できるフィールドが存在します。ここでは記載を割愛するので、詳細は開発者ガイドを参照してください<sup>※16</sup>。

### ●ステートマシンの定義

ステートマシン全体の構造は、下記のフィールドを使って定義します。なお、フィールドには必須とオプションの2種類があります。

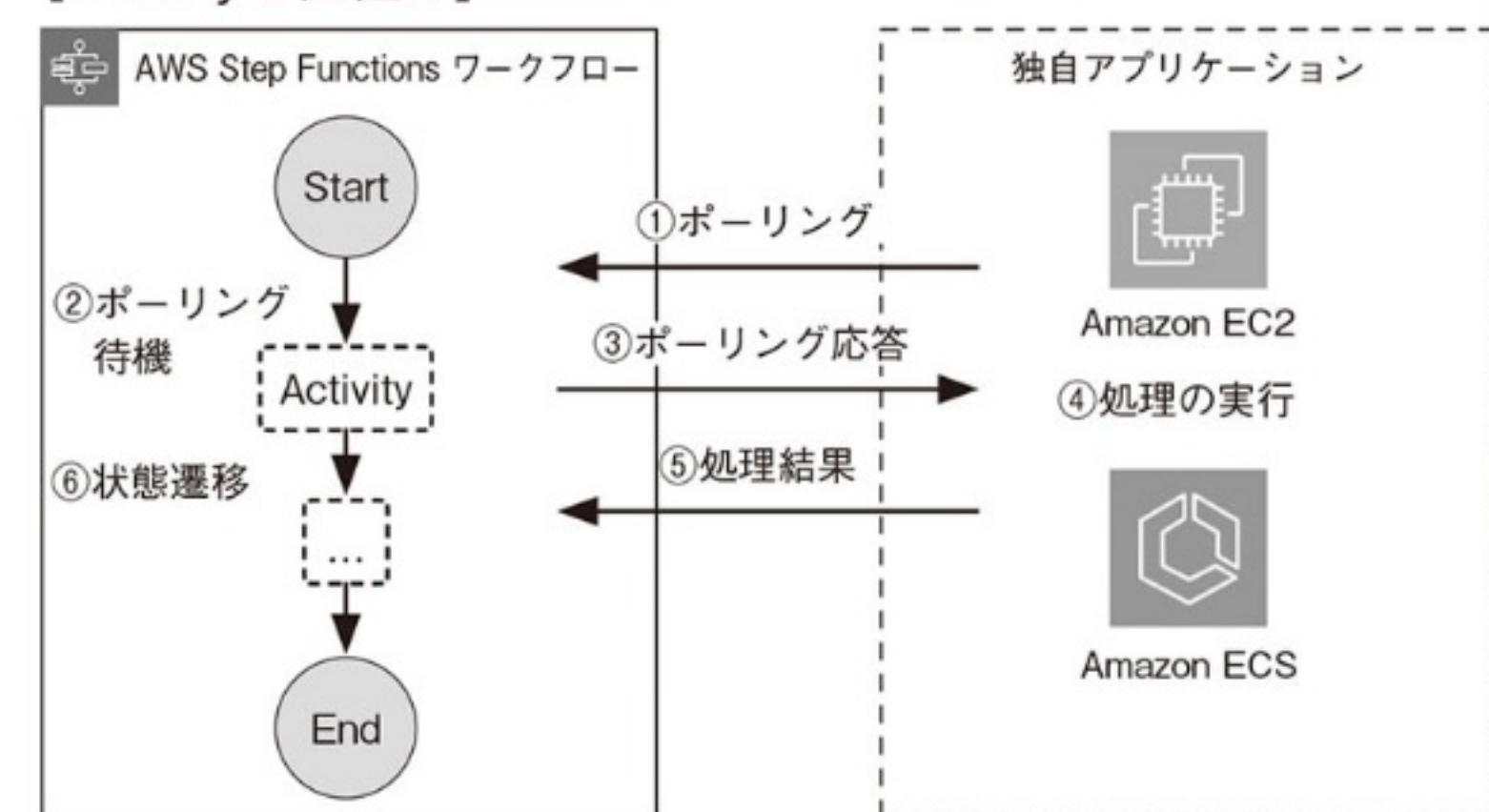
#### 【ステートマシン全体の構造を定義するフィールド】

フィールド名	必須/オプション	役割
Comment	オプション	•ステートマシンの説明を記載する。
StartAt	必須	•最初に実行するステートを指定する。
TimeoutSeconds	オプション	•ステートマシンを実行できる最大時間を秒単位で指定する。 •設定値を超えるとステートマシンの実行がタイムアウトし、States.Timeoutエラーが発生する。
Version	オプション	•ステートマシンで使用するASLのバージョンを設定する。 •デフォルトでは「1.0」が設定される。
States	必須	•ステートマシンで実行するステートを設定する。 •複数のステートが設定できる。

### Activity

Taskに指定ができる「Activity」とは、利用者がEC2やECSなどに独自に実装したアプリケーションと、Step Functionsのステートを関連付けて実行できる機能です。ステートマシンのステートがActivityの状態になると、独自アプリケーションで稼働する「ワーカー」からのポーリングを待機します。ポーリングに対する応答がステートマシンから返されると、アプリケーション側で処理を実行します。処理結果をステートマシンに渡して、後続のステートに遷移してステートマシンの処理を継続します。

#### 【Activityの仕組み】



\*16 [https://docs.aws.amazon.com/ja\\_jp/step-functions/latest/dg/concepts-states.html](https://docs.aws.amazon.com/ja_jp/step-functions/latest/dg/concepts-states.html)

### 3 データの入出力

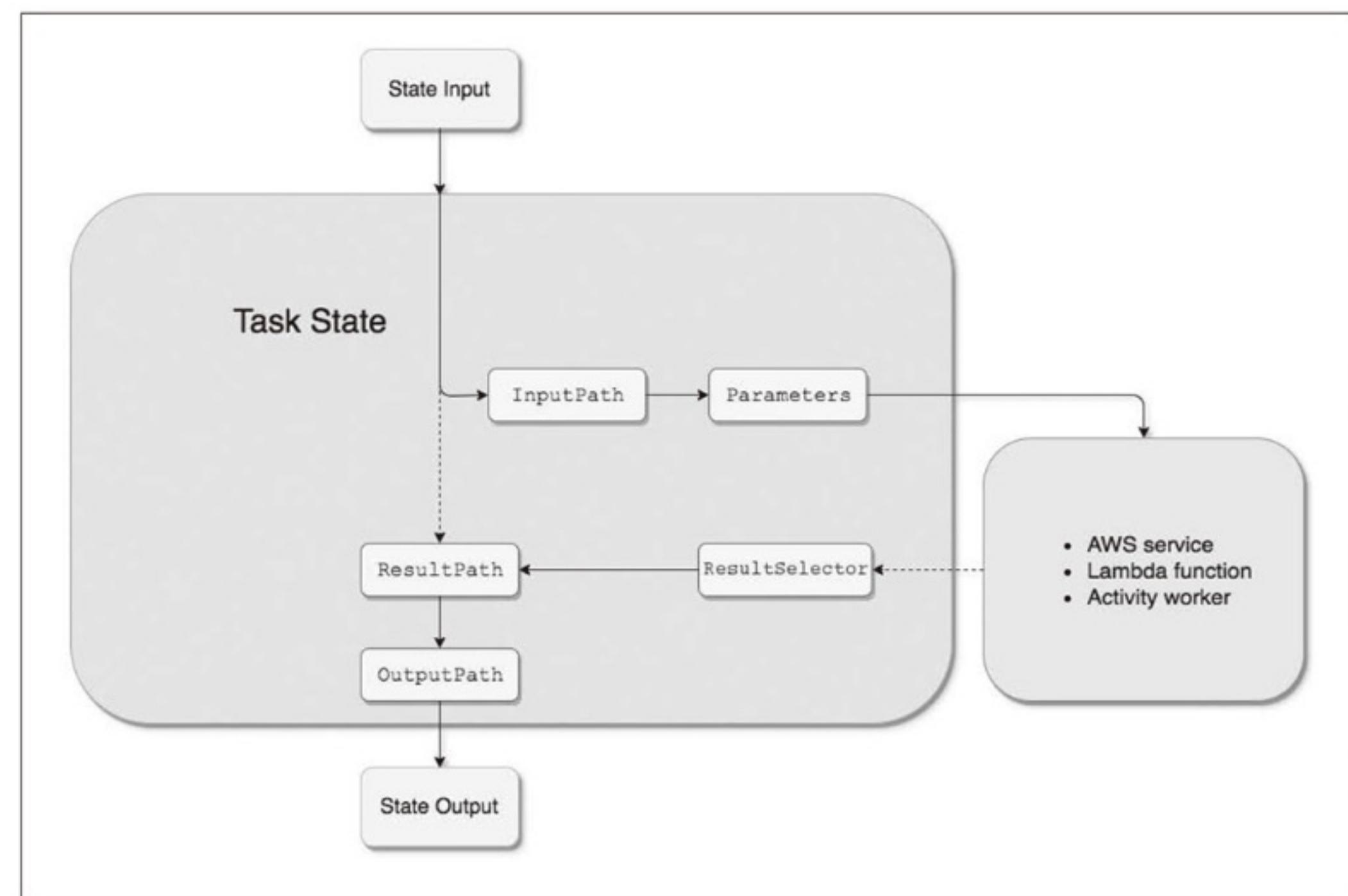
ステートのデータの受け渡しはJSON形式のデータで行われます。ASLで下記のフィールドを指定すれば、データの入出力をJSONPath構文で定義することができます<sup>\*17</sup>。

#### 【ステートマシン全体の構造を定義するフィールド】

フィールド名	役割
InputPath	<ul style="list-style-type: none"> <li>Inputとして受け取ったデータのうち、ステートの処理で使うものを選択する。</li> <li>InputPathを指定しない場合は、受け取ったデータをすべて渡す。</li> </ul>
Parameters	<ul style="list-style-type: none"> <li>InputPathで指定されたデータを受け取ってKey-Value形式のコレクションを作成して、ステートの処理のパラメータとして渡す。</li> <li>Parametersを指定しない場合は、受け取ったデータをすべて渡す。</li> </ul>
ResultSelector	<ul style="list-style-type: none"> <li>「ステートでの処理結果」を受け取ってResultPathに渡すデータを選択し、Key-Value形式のコレクションを作成してResultPathに渡す。</li> <li>ResultSelectorは下記のステートタイプで利用可能である。           <ul style="list-style-type: none"> <li>Task</li> <li>Parallel</li> <li>Map</li> </ul> </li> </ul>
ResultPath	<ul style="list-style-type: none"> <li>「ステートに渡されたInputデータ」と「ステートでの処理結果」を受け取り、OutputPathに渡すデータを指定する。</li> <li>ResultPathを指定しない場合は、「ステートでの処理結果」を渡す。</li> <li>ResultPathは下記のステートタイプで利用できる。           <ul style="list-style-type: none"> <li>Task</li> <li>Pass</li> <li>Parallel</li> </ul> </li> </ul>
OutputPath	<ul style="list-style-type: none"> <li>「ステートの処理結果」として次のステートに渡すデータを指定する。</li> <li>ResultPathを指定しない場合は、ResultPath以前のデータを出力として渡す。</li> </ul>

ステートにおけるデータの入出力を可視化すると下図のようになります。

#### 【ステートのデータ入出力<sup>\*18</sup>】



**Step Functions Distribution Map が re:Invent 2022 で発表**  
 Step Functionsでは、従来実行可能な並列処理は40でしたが、最大10,000まで対応可能なDistribution Mapが利用可能になりました。この新しい機能により、サーバレスアプリケーション内の大規模な並列ワークロードを調整するためのStep Functionsを記述できます。例えば、Amazon Simple Storage Service (Amazon S3) に保存されているログ、画像、.csv ファイルなど、数百万を超えるオブジェクトに対して、イテレーションで処理実行できるようになります<sup>\*19</sup>。

\*17 JSONPath構文については次のドキュメントを参照してください。<https://github.com/json-path/JsonPath>

\*18 [https://docs.aws.amazon.com/ja\\_jp/step-functions/latest/dg/concepts-input-output-filtering.html](https://docs.aws.amazon.com/ja_jp/step-functions/latest/dg/concepts-input-output-filtering.html) より引用

\*19 <https://aws.amazon.com/jp/blogs/news/step-functions-distributed-map-a-serverless-solution-for-large-scale-parallel-data-processing/>

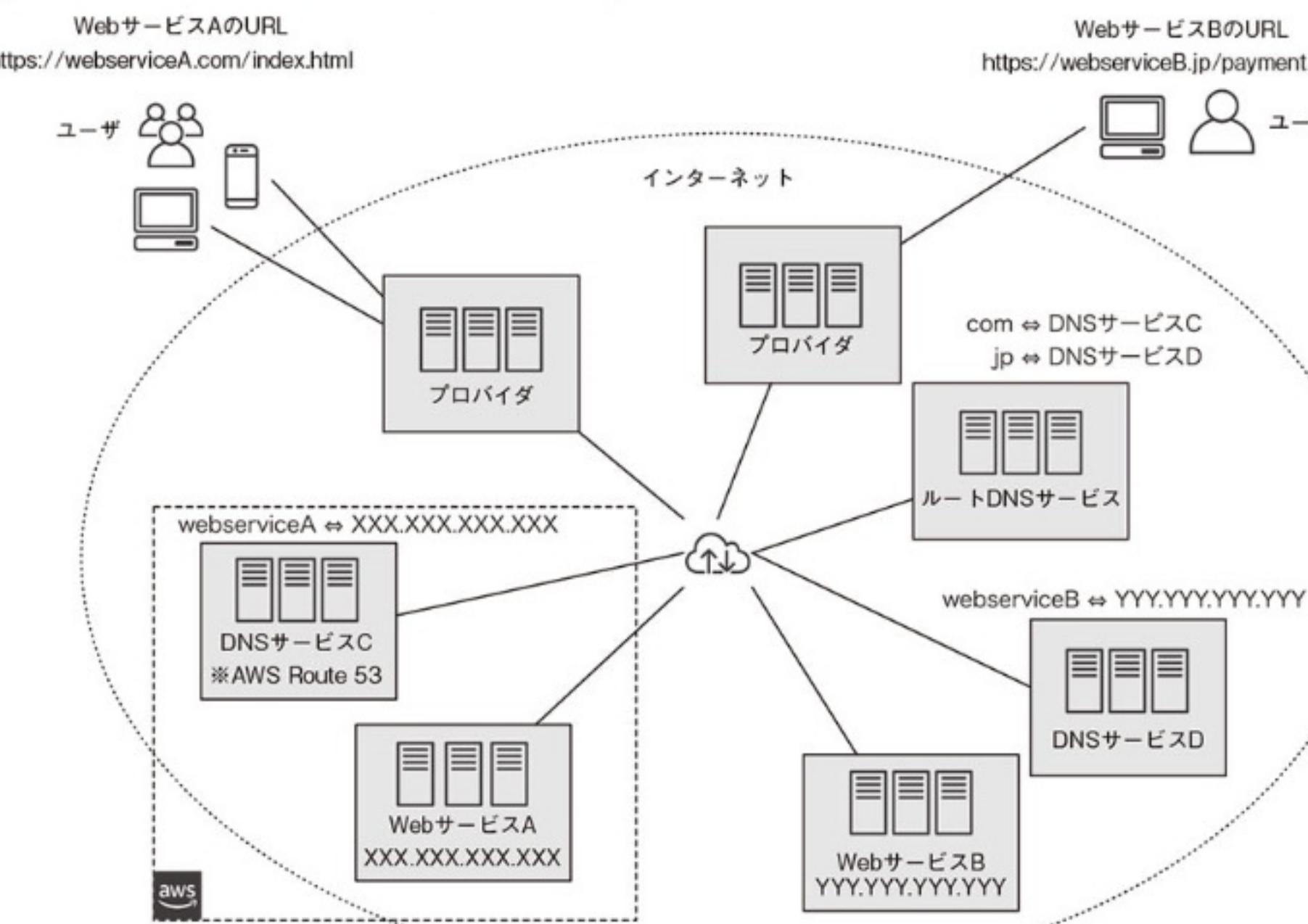
## 2-5 Amazon Route 53

Route 53はAWSから提供されているDNS (Domain Name Service) サービスです。DNSは、URLに代表される名前とIPアドレスの解決（対応付け、マッピング）を行うサービスで、さまざまなベンダーから提供されています。AWSではマネージドサービスとして独自の機能を追加して提供しています。

### 1 DNSの役割

Route 53の解説に入る前に、DNSの機能やサービスについておさらいしておきます。例えば、私たちが手元のパソコンでインターネットを通じて、あるWebサービスを利用したい場合、サービスを提供するサーバのIPアドレスを知る必要があります。IPアドレスは有限なリソースであり、相当数がサーバへ動的に割り当てられます。また、同じ役割を持つサーバが、通常冗長化されて複数存在するため、直接IPアドレスを指定してサービスにアクセスするのは現実的ではありません。そこで、サービスを一意に識別するためのURLを使って対象のサービスへ、プロバイダ（インターネットサービスプロバイダ）を通じてアクセスします。

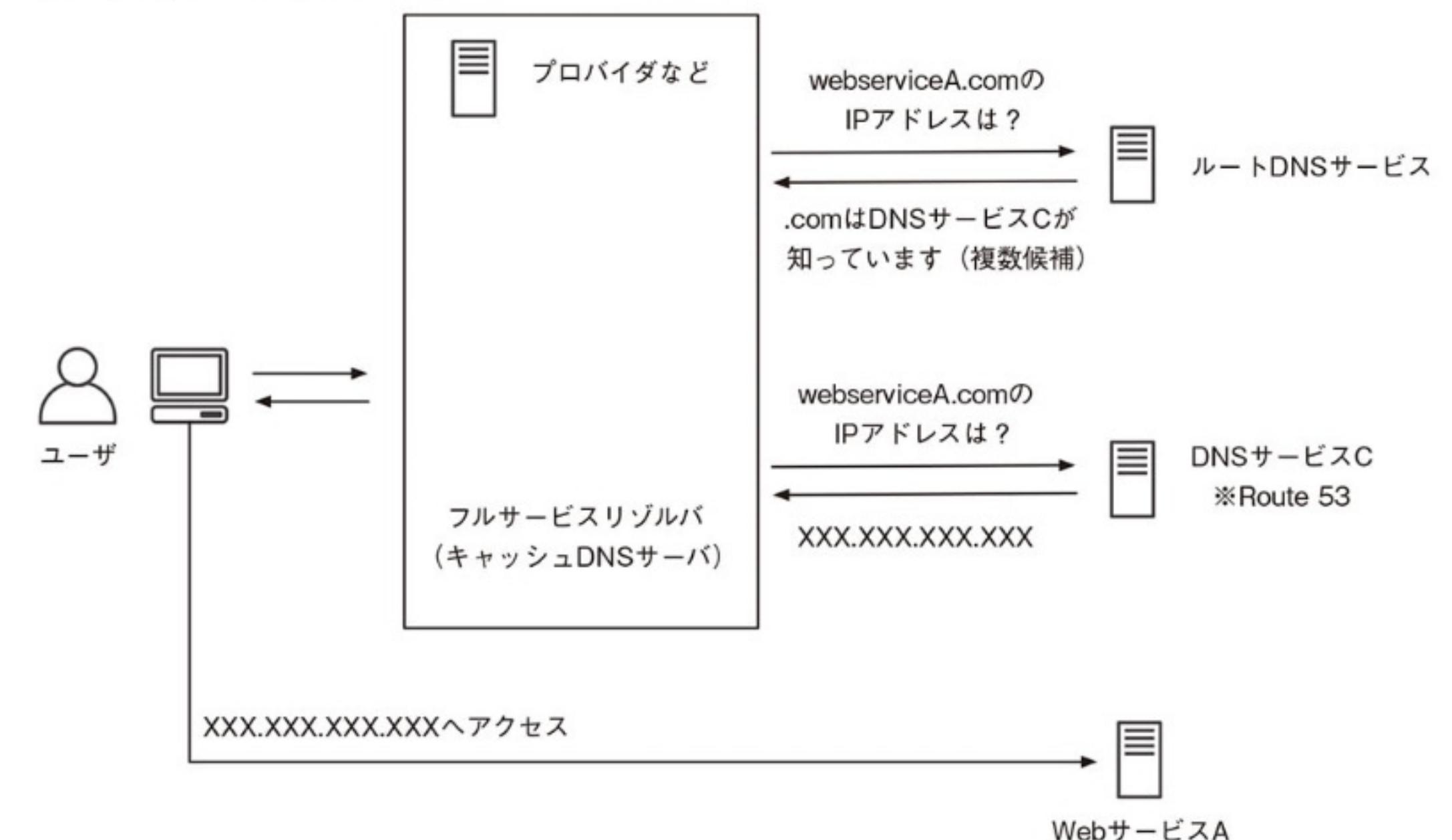
#### 【DNSの役割とRoute 53の位置付け】



URLは「Uniform Resource Locator」の略称で、「プロトコル://ドメイン名/ディレクトリパス名/ファイル名」という形式で構成されます。ここで、Webサービスを一意に識別する情報がドメイン名であり、DNSはこのドメイン名と実際にサービスを提供するサーバのIPアドレスを変換する役割を担うサービスです。DNSは階層構造をとり、ドット(.)で区切られたドメイン名を逆引きしていく構造になっています。まずルートDNSサービスにアクセスし、ドットで区切られたドメイン名に関する情報を持つDNSサービスに再帰的にアクセスしていく、最終的に目的となるドメインのIPアドレスを持つDNSサービスにアクセスします。Route 53は、主にAWSで提供されるWebサービスに関するIPアドレスやDNSサーバ情報を持つDNSサービス（図中でいうとDNSサービスC）と考えて差し支えはありません。Route 53自体は汎用的なDNSサービスなので、特にAWSで提供されるサービスに限るわけではありませんが、AWSで提供されるサービスに対してよく用いられています。

前図を処理フローの断面で見てみましょう。Route 53が果たす役割がより明確になるはずです。

#### 【Route 53の処理フロー】



ユーザがURLを指定してインターネットにアクセスしようとすると、通常フルサービスリゾルバというDNSクライアントがルートサーバに問い合わせます。フルサービスリゾルバは利用しているプロバイダのDNSキャッシュサーバを指定したり、ルーターがその機能を代替するなど、クライアント端末や環境によって提供する機能が変わりますが、設定によって任意に構成することができます。フルサービスリゾルバはURLが示すIPアドレスを得られるまで再帰的に問い合わせを続けます。Route 53はドメイン名が指示するIPアドレスを提供する、もしくはIPアドレスを提供するサーバの情報を提供するDNSサービスです。

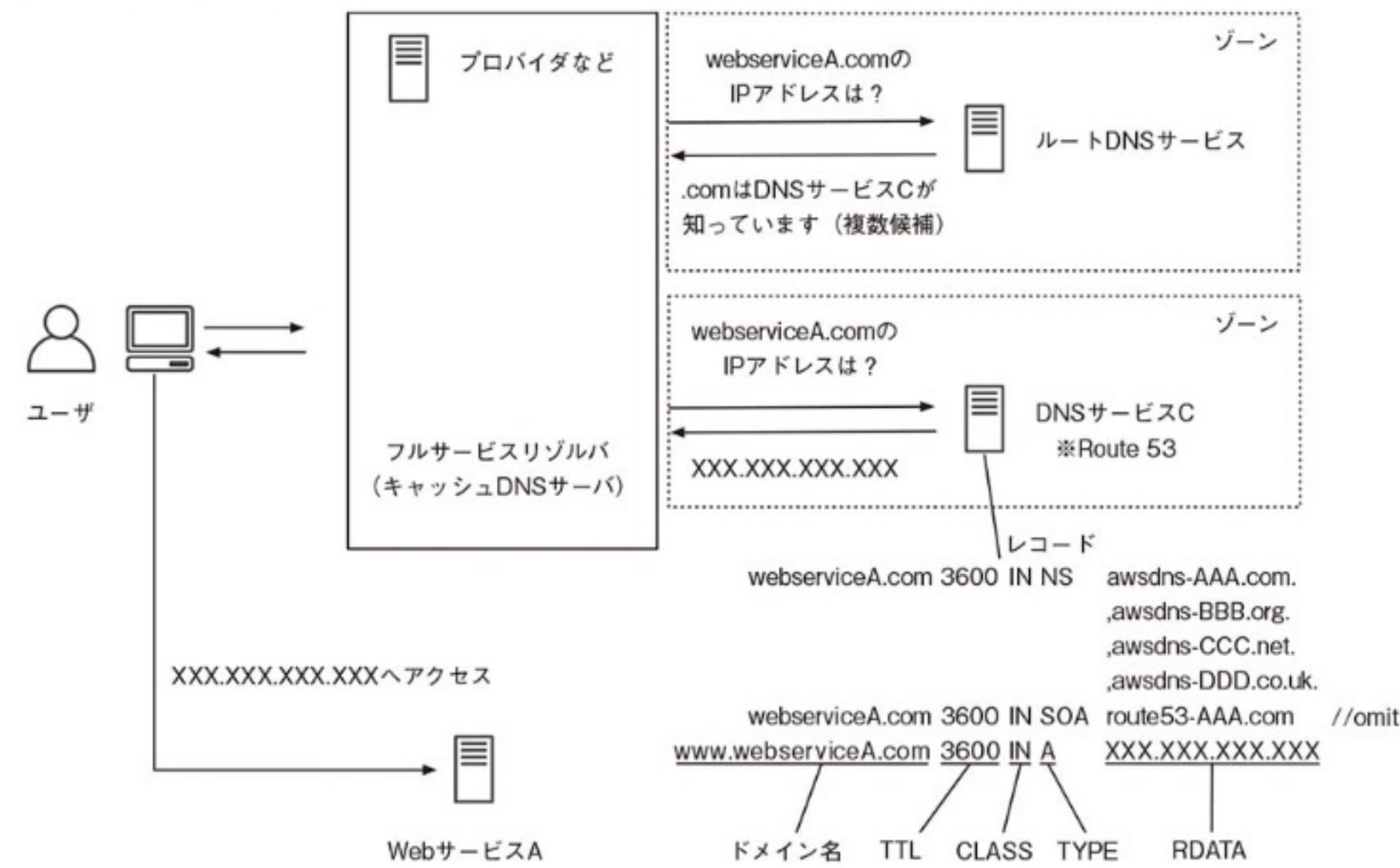
## 2 ホストゾーン(Hosted Zone)

身近な例がわかりやすいので、インターネットを例に説明しましたが、Route 53はVPC内の閉塞的なネットワークでもDNSサーバとしての役割を提供します。

### ●ゾーンとレコード

DNSでは下位のドメインサーバに委ねる過程で、所掌範囲をゾーンという領域に分割して管理します。上記の図でゾーンを明示すると以下のような点線の範囲に区切られます。

#### 【ゾーンのスコープ】



各ゾーンにおける、DNSサーバでは以下のようない形式のレコードで、自身の属性や権限移譲先のネームサーバ、IPアドレスを保持しています。

### ●ドメイン名

ゾーンを管理するネームサーバのドメイン名を指し示します。

### ●TTL

レコードが有効な期限です。

### ●CLASS

ネットワークプロトコルを示します。ほとんどの場合、インターネットであるINが使用されます。

### ●TYPE

レコードのタイプを指定します。一般的には、主に以下のようない種類がありますが、各ゾーンには、ゾーンの起点となるSOAレコードと、親ゾーンに含まれるものと同じNSレコード、およびルーティング対象となるAWSサービスを示したAレコードもしくはAAAAレコードが含まれます。

#### 【使用頻度の高い代表的なレコードタイプ】

レコードタイプ	概要
SOA (StartOfAuthority)	ゾーンを管理する主体であること (DNSキャッシュなどで参照される大本のデータのこと) を示すレコード。
NS (NameServer)	ゾーンを管理するネームサーバのドメイン名を指し示す。通常、自身とその親となるDNSサーバにレコードタイプNSのデータが登録されている。親ゾーンからはNSレコードの値を通じて、子ゾーンである自身に再帰問い合わせされてくる。
A	ドメインに対応するIPv4アドレスもしくはDNS名を指定する場合に利用する。MXレコード、CNAMEレコード、NSレコードを設定する場合、事前にAレコードとして、各レコードのVALUEに入力するホスト (FQDN) を設定しておく。Aレコードには、Route 53の独自の拡張機能レコードとして、ELBやCloudFront、Amazon S3バケットといったAWSリソースにトラフィックをルーティングするエイリアスレコードを設定できる <sup>*20</sup> 。
AAAA	ドメインに対応するIPv6アドレスもしくはDNS名を指定する場合に利用する。
CNAME	複数のドメインの名前解決を置き換える場合などに利用する。
MX	対象ドメイン宛のメールの配達先 (メールサーバ) のホスト名を定義するレコード。

### ●RDATA

レコードに対して実際に指定する値です。

Route 53で上記のようなかたちのDNSサービスを定義することをホストゾーン(Hosted Zone)と呼びます。ホストゾーンには、パブリックなネットワークで実行されるパブリックホストゾーンと、閉塞されたネットワークで動作するプライベートホストゾーンに分類されます。上記の例では、webserviceA.comをパブリックホストゾーンとして設定し、www.webserviceA.comドメイン向けのリクエストを受け取った際に、XXX.XXX.XXX.XXXへルーティングするように設定しています。

\*20 [https://docs.aws.amazon.com/ja\\_jp/Route53/latest/DeveloperGuide/resource-record-sets-choosing-alias-non-alias.html](https://docs.aws.amazon.com/ja_jp/Route53/latest/DeveloperGuide/resource-record-sets-choosing-alias-non-alias.html)

## ● トラフィックルーティング

Route 53では、各A/AAAAレコードに設定された値にもとづき、以下のルールに則ってトラフィックのルーティングを行います。

### 1. シンプルルーティング

ドメインに紐付いたアドレスにシンプルにルーティングします。複数ルーティング先がある場合はラウンドロビンで振り分けられます。

### 2. 位置情報ルーティング

クライアントのIPアドレスから位置情報を特定し、ルーティングを行います。国や地域ごとにルーティング先を設定し、ユーザの位置情報に応じてルーティングされます。このルーティングを使用するには後述するトラフィックフローを設定する必要があります。

### 3. フェイルオーバールーティング

ヘルスチェックと組み合わせてシステムがダウンしたときに自動的にフェイルオーバーを行い、待機システムへと切り替えます。

### 4. 地理的近接性ルーティング

ユーザとサービスを提供するサーバの物理的な距離にもとづいてルーティングを行います。サービスのエンドポイントをリージョンやカスタム座標で指定し、ユーザとの物理的な距離に応じて最も近接するエンドポイントルーティングされます。

### 5. レイテンシベースドルーティング

クライアントが最もレイテンシが低くなるようにルーティングを行います。一般的に最も近いロケーションとなりますが、一定期間ネットワーク速度などの測定値を考慮してルーティングされます。

### 6. 加重ルーティング

ユーザによって指定された割合にもとづき、複数のAWSリソースへルーティングされます。

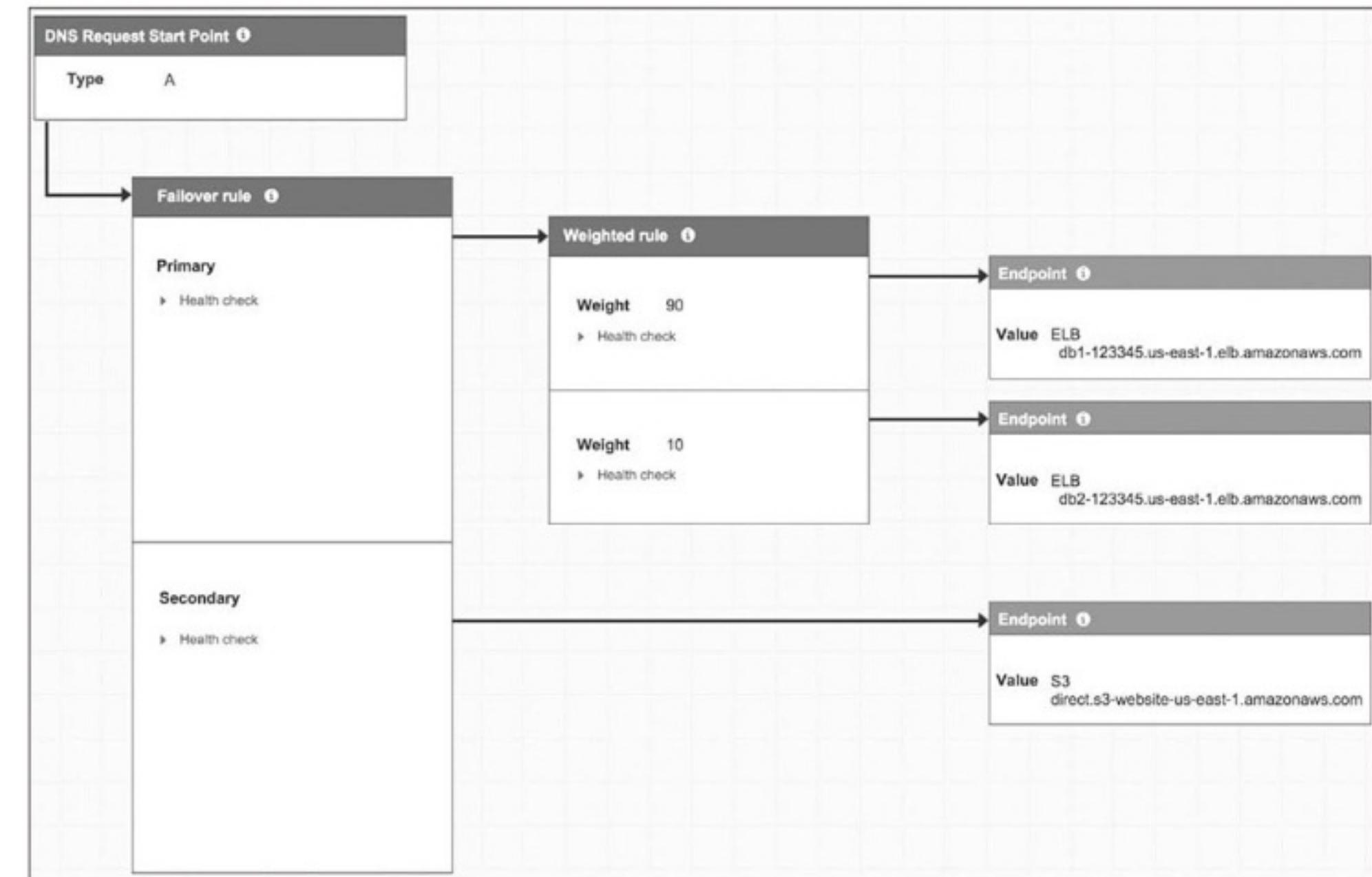
### 7. 複数回答ルーティング

1つのドメインに対し、最大8つのランダムに選択された正常なレコードの結果でDNSクエリに応答します。

## ● トラフィックフロー

トラフィックフローは上述したさまざまなトラフィックルーティングを簡単に組み合わせて作成できる機能です。以下のように、GUIコンソール上から直感的なビジュアルエディタを使ってフローを作成し、トラフィックポリシーとして保存します。

### 【トラフィックフロー】



トラフィックポリシーは、ポリシーレコード（ホストゾーンとDNS名を持つレコード）を作成することによりRoute 53に適用されます<sup>\*21</sup>。

## 3 Route 53 Application Recovery Controller

Application Recovery Controllerは、2021年にリリースされた、複数のリージョン間でシステム構成するリソースの設定・状態をチェックしたり、リージョン間でフェイルオーバー時に安全にルーティングするためのディザスタークリア機能です<sup>\*22</sup>。この機能は次の2つの仕組みから実現されています。

\*21 [https://docs.aws.amazon.com/ja\\_jp/Route53/latest/DeveloperGuide/traffic-policy-records.html#traffic-policy-records-creating-values](https://docs.aws.amazon.com/ja_jp/Route53/latest/DeveloperGuide/traffic-policy-records.html#traffic-policy-records-creating-values)

\*22 <https://aws.amazon.com/jp/about-aws/whats-new/2021/07/amazon-route-53-announces-route-53-application-recovery-controller/>

- Readiness check

複数のリージョンにまたがり、冗長化されたリソース間の差分をチェックする機能です。チェック対象とするリソースをCell/Recovery Group/Resource Setなどのコンポーネントとして定義し、リソース間の差分や状態のチェックを定義します。また、フェイルオーバーされる対象の定義したリソースの準備状況を検証します。

- Routing Control

Routing Controlでは、ヘルスチェックと連動して、あるリージョンで障害が発生した場合、トラフィックを安全にフェイルオーバー先のリソースに切り替えます。Cluster/Control Panel/Safety Ruleといったコンポーネントを定義し、ルーティングの設定を行います。



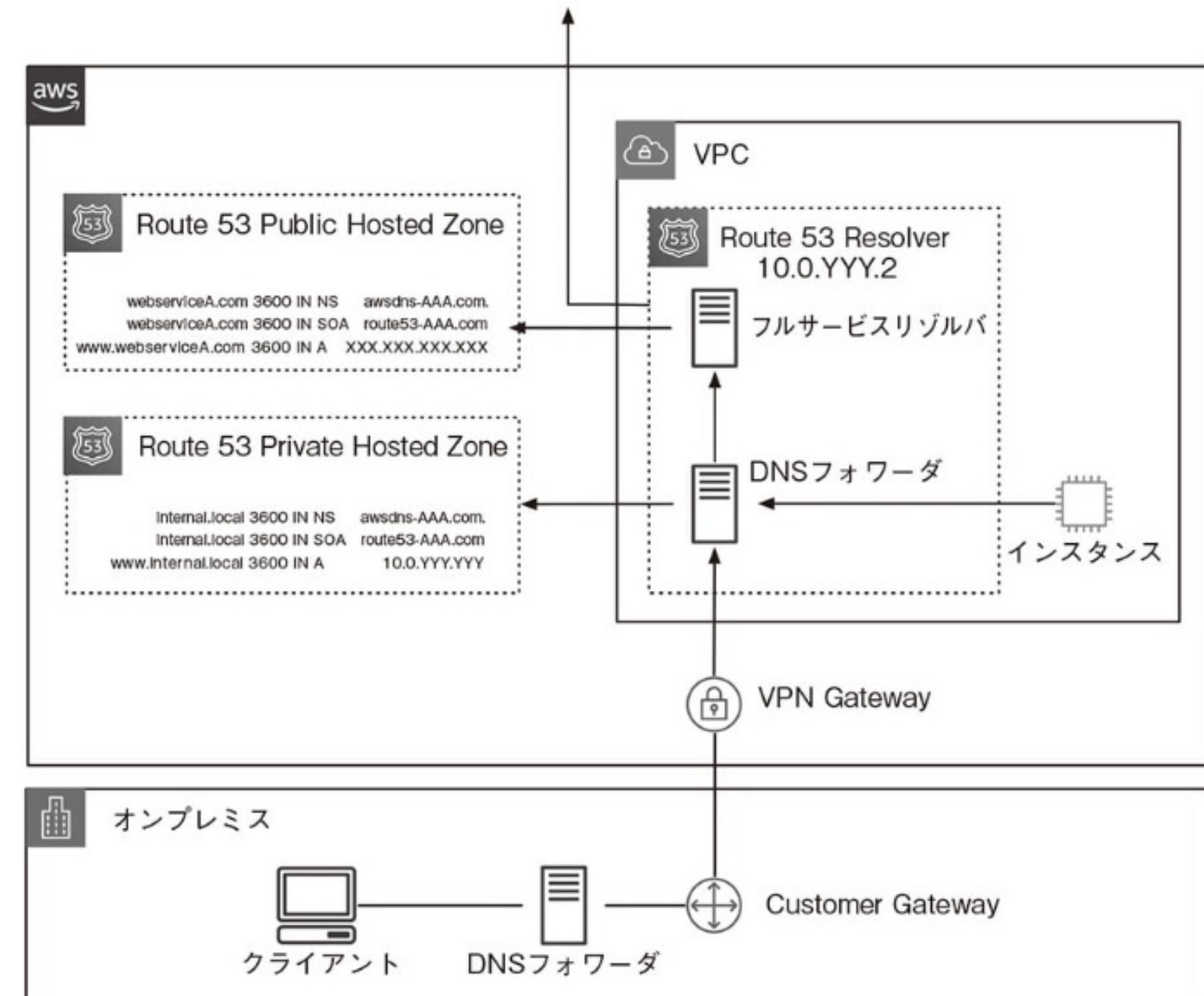
#### Application Recovery Controller Zonal Shift の発表<sup>※23</sup>

re:Invent 2022では、このApplication Recovery Controllerの新しい機能として、特定のアベイラビリティゾーンに障害が発生した際、障害が発生した部分のトラフィックを切り離すZonal Shiftが発表されました。

## 4 Route 53 Resolver

Route 53 ResolverはVPCを構築したときに同時に作成されるDNSサーバです。前項で解説した、パブリック・プライベートホストゾーンのDNSサービスとは別のサービスです。従来は「Amazon Provisioned DNS」と呼ばれていましたが、2018年のアップデートでDNSサーバへのフォワード機能が付与されて、Route 53 Resolverという名前に改称されました。以下の図は、Route 53 ResolverのVPCにおける役割を図に示したものです。

### 【Route 53 Resolverの役割】



Route 53 Resolverは内部的にはフルサービスリゾルバとDNSフォワーダーの機能を持っています。フルサービスリゾルバは前項でも簡単に触れたとおり、ネームサーバに対しドメインに対応するIPアドレスを再帰的に問い合わせする役割を担います。ここでは、Route 53 Resolverのもう1つの機能であるDNSフォワーダについて説明します。

#### ●DNS フォワーダ

クライアントの多くは、名前解決のリクエストをフルサービスリゾルバに対して行う際、ドメインの種類に応じて問い合わせ先のDNSサーバを振り分けるような機能は持っていません。企業内ネットワークのような大規模な環境だとフルサービスリゾルバにさまざまなドメインの名前解決リクエストが集中して届いてしまうため、アクセス過多によるパフォーマンス低下などの問題が発生します。そこで名前解決リクエストを振り分ける機能を持つDNSフォワーダを設置します。

フォワーダには、特定のドメインの名前解決リクエストが届いたときに、フルサービスリゾルバやネームサーバ、あるいは別のDNSサーバなど中継先となるDNSサーバに振り分けるルールを設定します。これにより、DNSフォワーダが名前解決リクエストを適切なDNSサーバに振り分けてくれるため、フルサービスリゾルバの負荷を軽減することができます。

※23 <https://aws.amazon.com/jp/about-aws/whats-new/2022/11/amazon-route-53-application-recovery-controller-zonal-shift/>

### ●Route 53 Resolverの役割

Route 53 Resolverは、EC2インスタンスをはじめとした、主にVPC内のクライアントから要求される名前解決リクエストを処理します。内部的にはフルサービスリゾルバとDNSフォワーダで構成されており、AWS内部ネットワークの名前解決のリクエストはRoute 53プライベートホストゾーンにフォワードし、インターネット外の名前解決はフルサービスリゾルバが外部DNSサーバやRoute 53パブリックホストゾーンに再帰的に問い合わせします。

また、オンプレミス環境を含めた名前解決も可能です（Route 53 Resolver for Hybrid Clouds）。具体的には以下のようなユースケースで相互接続のためのDNS設定を行えます。

1. オンプレミスネットワークからVPCリソースへの名前解決
2. オンプレミスネットワークからインターネット外への名前解決
3. VPCリソースからオンプレミスネットワーク向けの名前解決

オンプレミスと接続するには、Route 53 Resolverで、インバウンド・アウトバウンドエンドポイントを定義します。また、ドメイン名とDNSフォワーダ、フルサービスリゾルバ・ホストゾーンとの名前解決マッピングルールを定義します。オンプレミスネットワークからアクセスする際は、オンプレミス内のDNSサーバにRoute 53 ResolverへのDNSフォワーダを設ける必要があります。

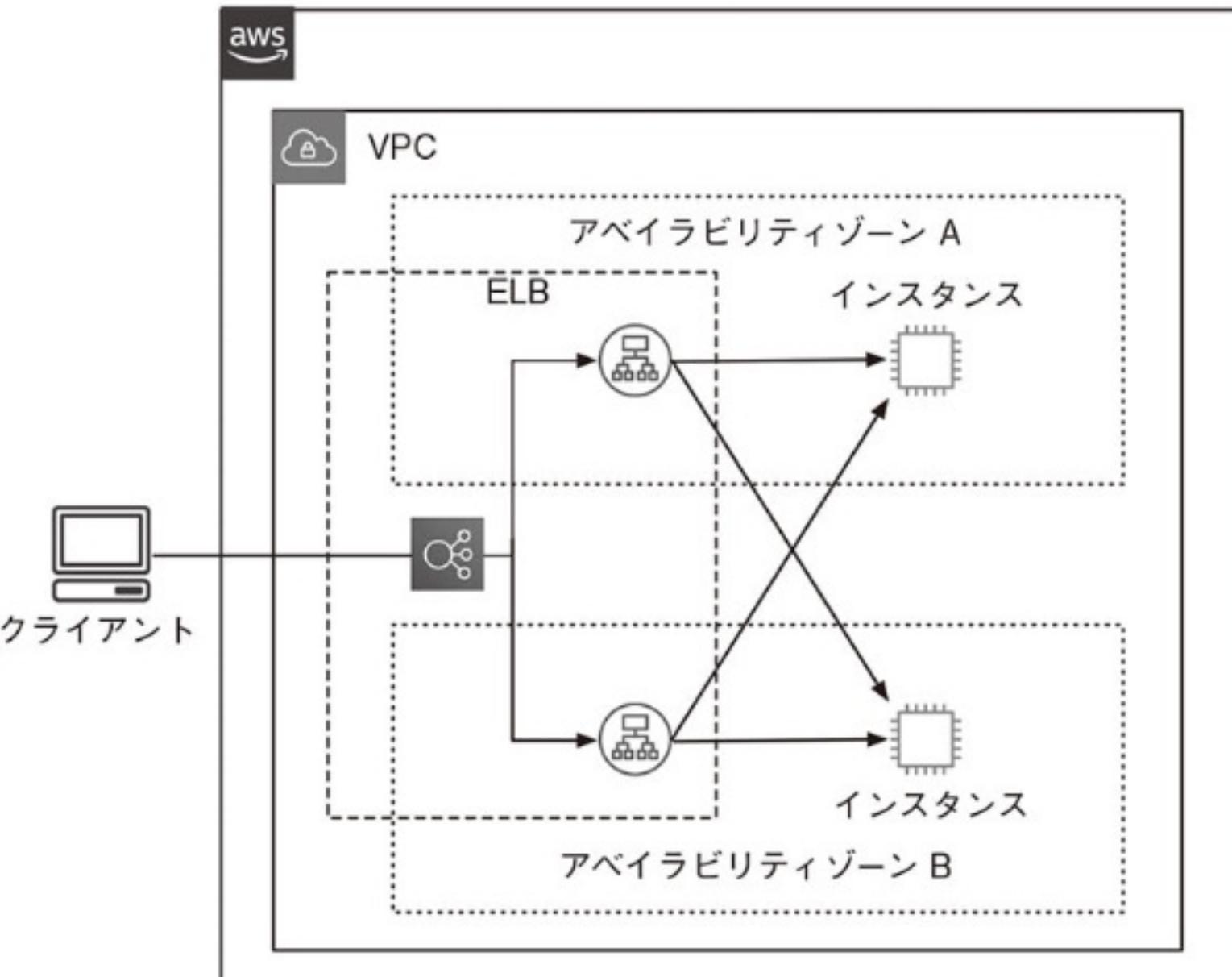
## 2-6 ELB(Elastic Load Balancer)

ELBは、AWSが提供する完全マネージドな仮想ロードバランシングサービスです。ELBという名称は、アプリケーションロードバランサー（Application Load Balancer: ALB）、クラシックロードバランサー（Classic Load Balancer: CLB）、ネットワークロードバランサー（Network Load Balancer: NLB）、ゲートウェイロードバランサー（Gateway Load Balancer: GLB）の総称として扱われています（2023年12月時点）。

### 1 ELBで共通する特徴

ロードバランシングとは、リクエストの負荷を複数のサーバに分散する技術のことです。ELBは複数のWebサーバやアプリケーションサーバでWebサービスを構成する場合に使用される、AWSの仮想ロードバランシングサービスです。

【ELBのロードバランシングイメージ】



ELBは、EC2やRDSのようにVPC内に仮想的に配置されます。定義上は1つのリソースのように扱われますが、内部的には冗長化構成されており、トラフィックが増加すると台数を増やしてスケールアウトしたり、スペックを上げてスケールアップしたりすることで対応します。逆に負荷が小さくなるとスケールダウンといったかたちで自動的に