Amazon ECSとCloud Runの相互理解で広げるクラウドネイティブの景色

新井 雅也 (@msy78)

新井 雅也 @msy78

Synspective Inc. - Principal Cloud Architect

現在は衛星の開発・運用を手掛けるスタートアップ企業にて、
クラウドを中心とした技術力を活かしつつ、宇宙業界の発展に尽力している。

好きなサービス: Amazon ECS、AWS Fargate、GKE、Cloud Run
好きなエナジードリンク: Red Bull

AWS Container Hero
AWS Ambassador 2022-2023
Google Cloud Champion Innovator
Google Cloud Partner Top Engineer 2023/2025

本発表の目的

ECSもしくはCloud Runのいずれかの事前知識を活用することで、
他方のサービスと照らし合わせながら相互理解が進むことに気づいてもらう

⚠ 前提 ⚠

- 各クラウドサービスの優劣比較を目的とはしていません。
  あくまで、「他方の領域に学びを広げる観点」で捉えましょう。

・Amazon ECSのデータプレーンとして、AWS Fargateを前提に解説します。

・本発表で扱うアーキテクチャは一般的な例です。
  ビジネス要件によっては、他の構成が最適となる可能性があります。

・便宜上、解説はECS→Cloud Runの順番で行います。

Q: なぜAmazon ECS とCloud Run？

Q: なぜ  Amazon ECS と  Cloud Run？

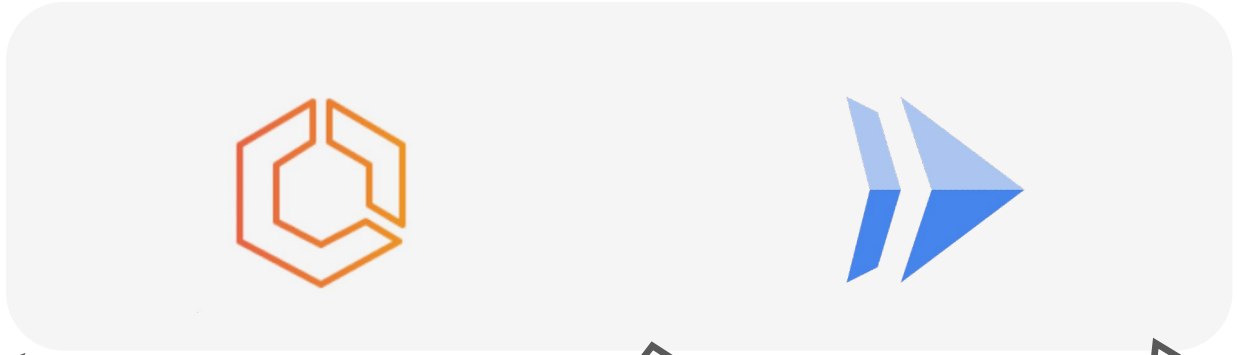
A: クラウド上でコンテナを活用した
システムを構築する際に最も受け入れら
れているサービスだから

- 世界中のECSタスク起動数は3億個/日
- AWSでコンテナ新規利用ユーザの65%がECSを選択
- 多数のECS/Fargateアーキテクチャ事例が公開

- 利用実績に関する定量的な数値は公開されていないものの、
  2019~2024年の5年間でPreviewを除く機能追加発表は151件
- Cloud Run functionsの統合など、主要サービスとしてリブランディング

アジェンダ

以下の3つの観点から相互理解を目指す



1. 基本的な特徴

2. アーキテクチャデザインの違い

3. 非機能デザインからの理解

アジェンダ

以下の3つの観点から相互理解を目指す

各サービスの基本的な特徴

1. コンポーネントとワークロード
2. VPC内外
3. アクセス制御
4. コンテナ実行環境

各サービスの基本的な特徴

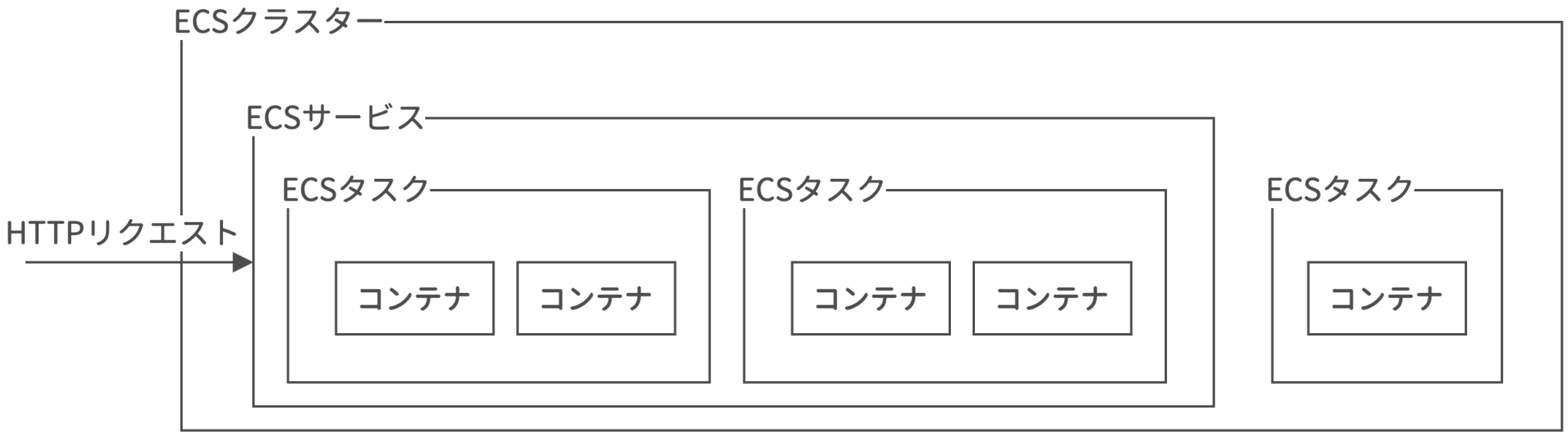
1. コンポーネントとワークロード
2. VPC内外
3. アクセス制御
4. コンテナ実行環境

ECSは主に5つのコンポーネントで構成

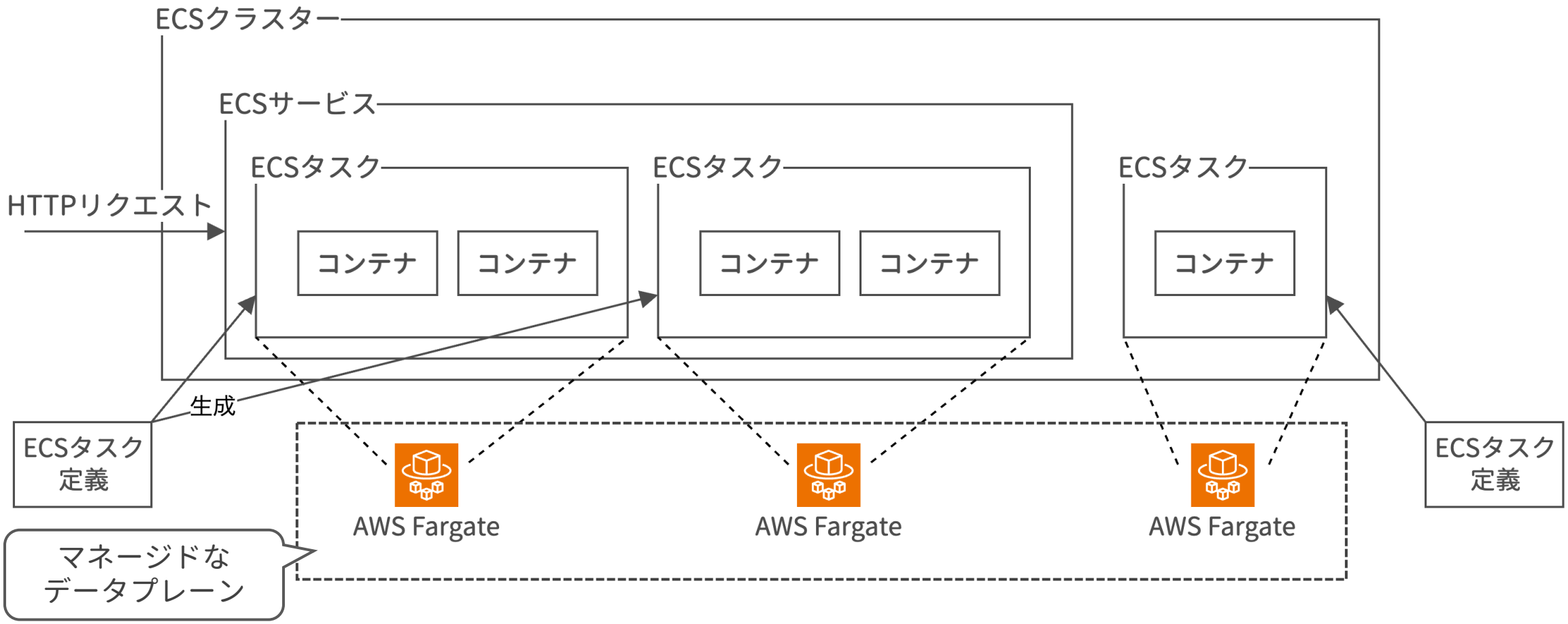
ECSは主に5つのコンポーネントで構成

コンテナ	・アプリケーションをパッケージ化した実行主体
ECSタスク	・1つ以上のコンテナを束ねるECSの最小実行単位
ECSタスク定義	・ECSタスクを生成するためのテンプレート ・リビジョンごとに管理
ECSサービス	・指定したECSタスクの起動数を維持
ECSクラスター	・複数のECSサービス・ECSタスクからなる 論理的なグループ

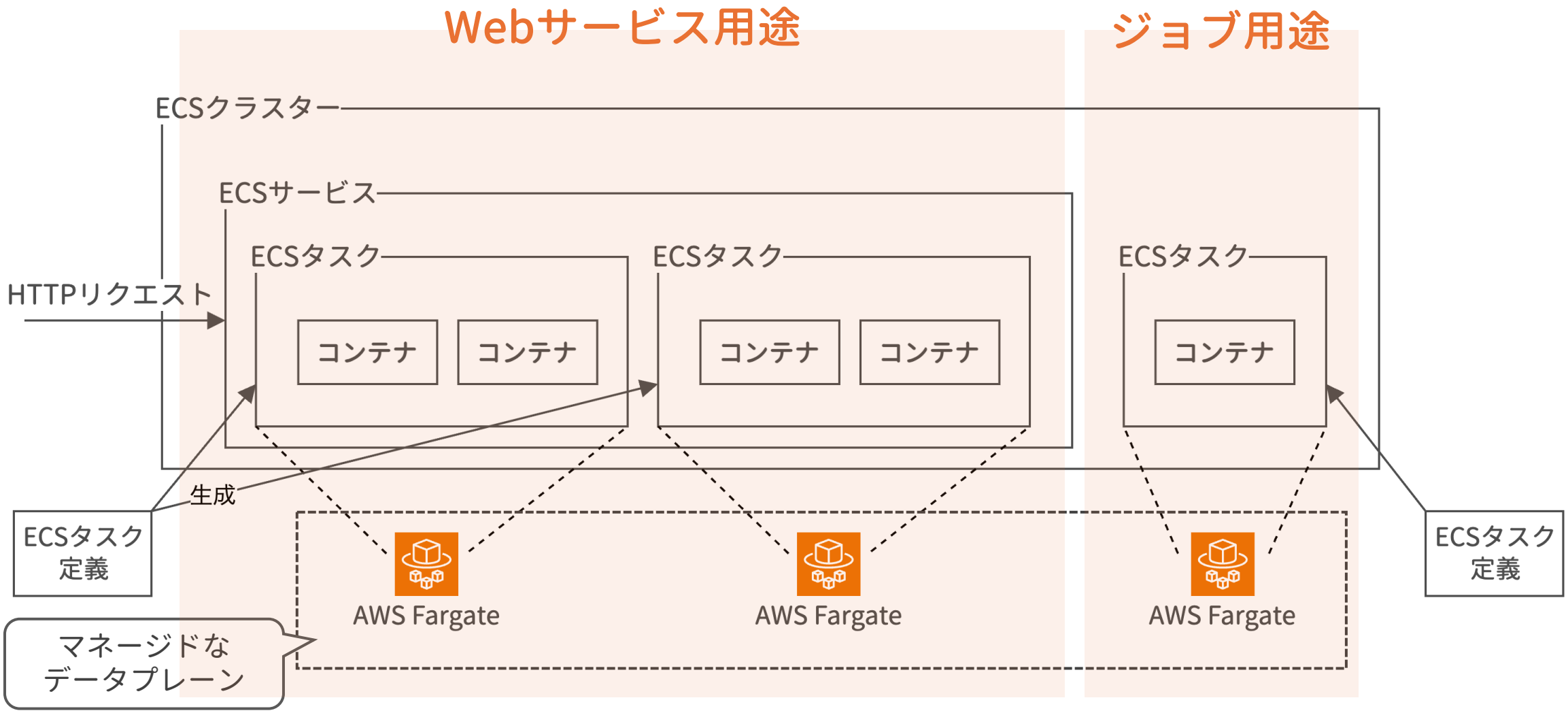
ECSの各コンポーネントとワークロードの関連性



ECSの各コンポーネントとワークロードの関連性



ECSの各コンポーネントとワークロードの関連性



基本的な各サービスの特徴 - コンポーネント

Cloud Run services は4つのコンポーネントで構成

コンテナ	・アプリケーションをパッケージ化した実行主体
インスタンス	・1つ以上のコンテナを束ねるCloud Runの最小実行単位 ・リビジョンに紐づく
リビジョン	・デプロイ毎に作成される設定の単位 ・リリース時、リビジョンごとにトラフィックを制御できる
サービス	・リビジョンを束ねてHTTPSエンドポイントを提供

Ref. https://cloud.google.com/run/docs/resource-model

基本的な各サービスの特徴 - コンポーネント

Cloud Run jobs は4つのコンポーネントで構成

Services と同じ

コンテナ

・アプリケーションをパッケージ化した実行主体

Services と同じ

インスタンス

・1つ以上のコンテナを束ねるCloud Runの最小実行単位
・リビジョンに紐づく

タスク

・並列実行の単位。インスタンスを1つ実行

ジョブ

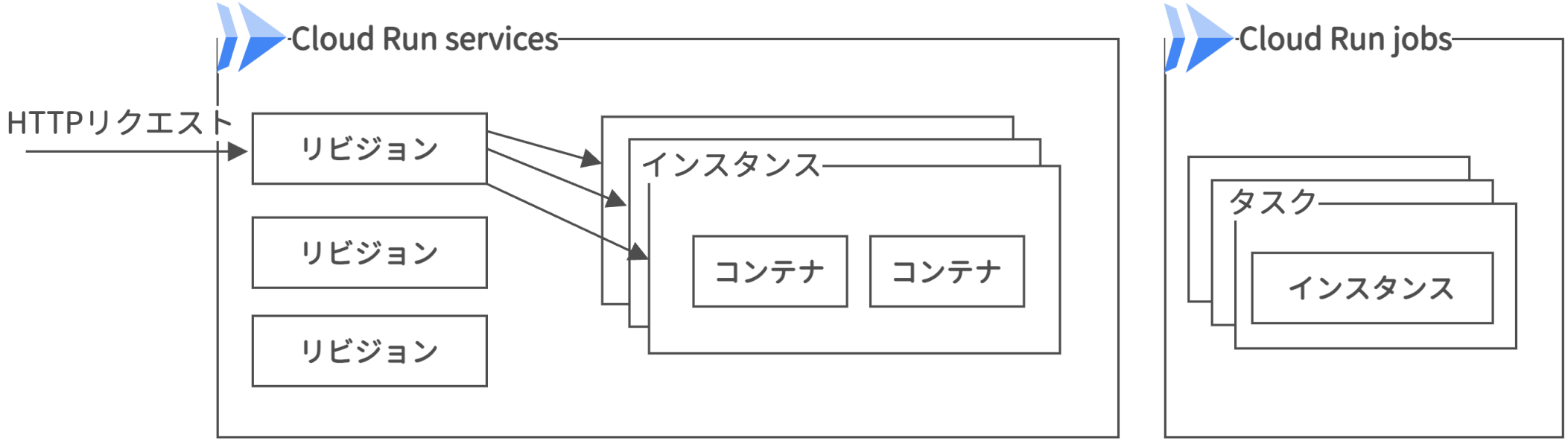
・複数のタスクを束ねて実行

Ref. https://cloud.google.com/run/docs/resource-model

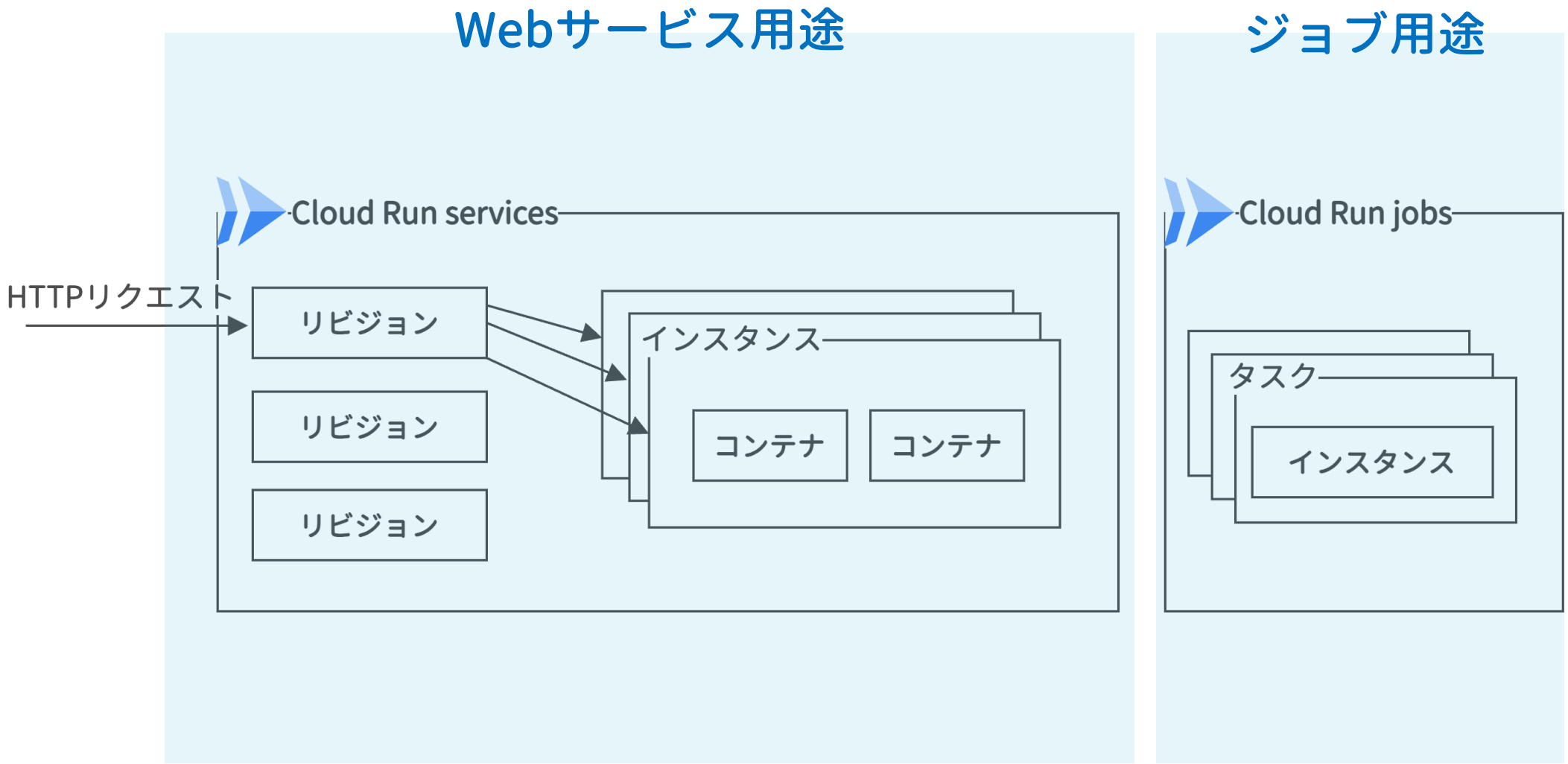
基本的な各サービスの特徴 - コンポーネントとワークロード

Cloud Runの各コンポーネントとワークロードの関連性

Amazon ECSと異なり、クラスターの概念がない



Cloud Runの各コンポーネントとワークロードの関連性



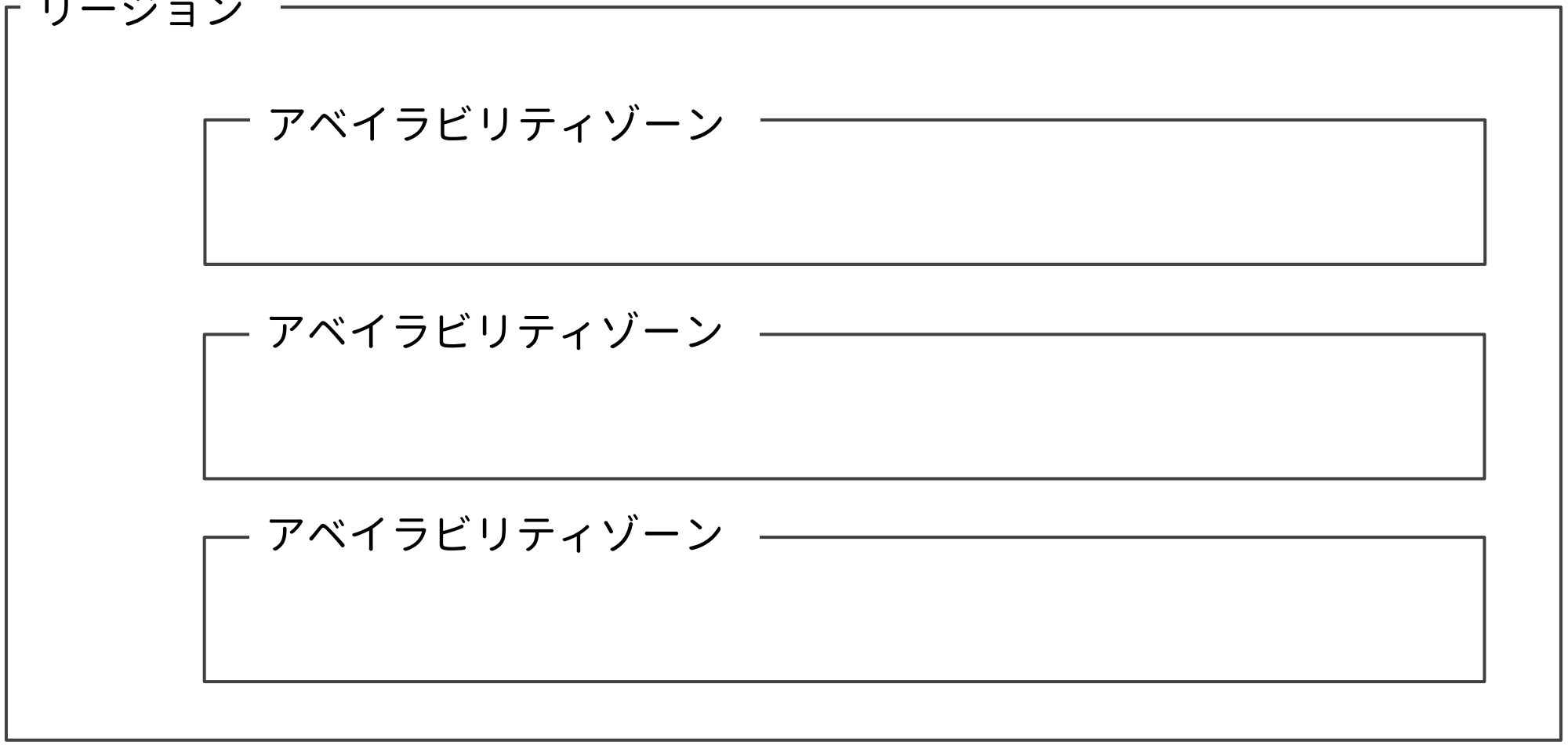
各サービスの基本的な特徴

1. コンポーネントとワークロード
2. VPC内外
3. アクセス制御
4. コンテナ実行環境

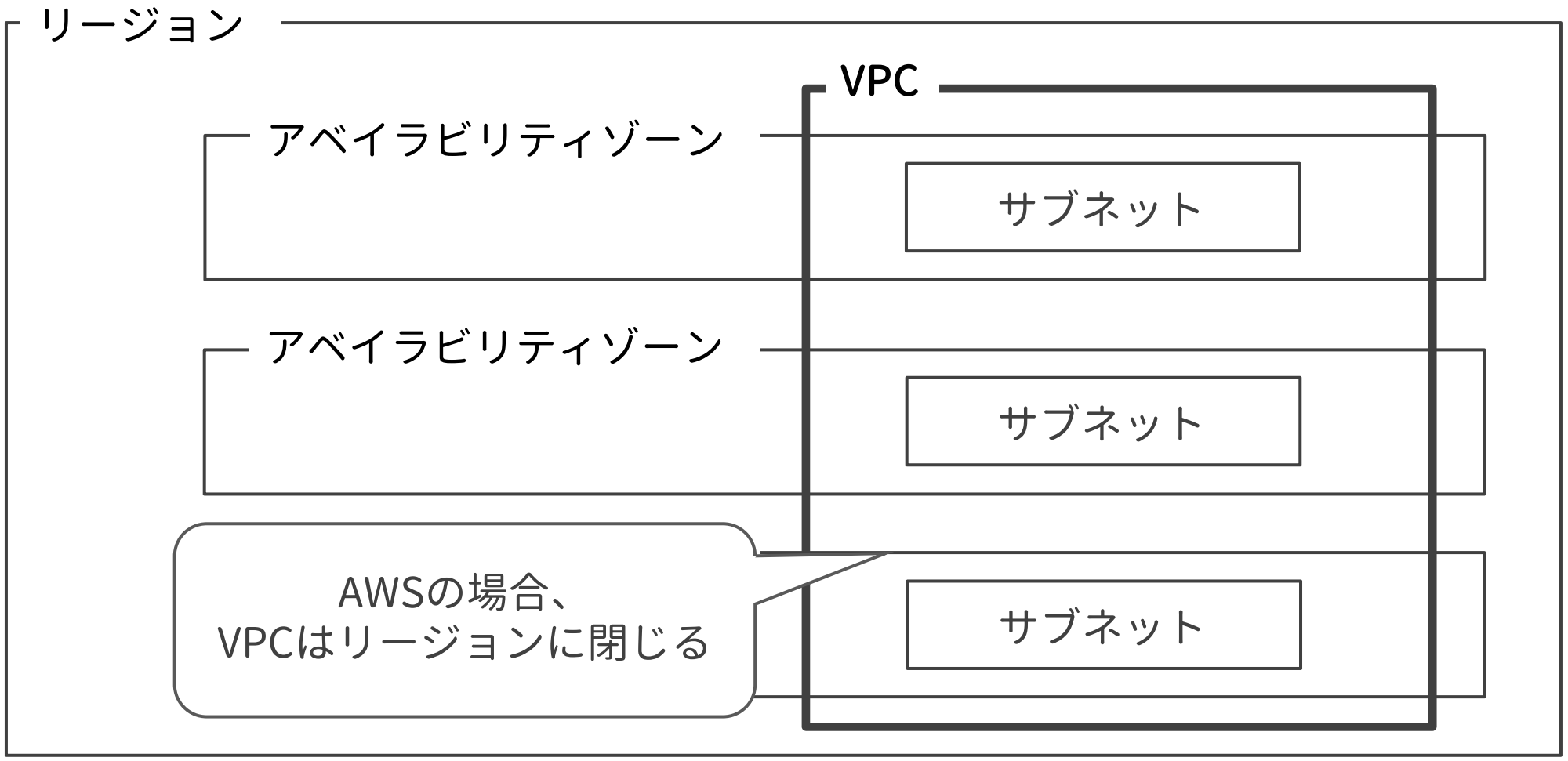
前提として、AWSとGoogle CloudではVPCのスコープが異なる

基本的な各サービスの特徴 - VPC内外

おさらい）AWSにおけるVPCのスコープ

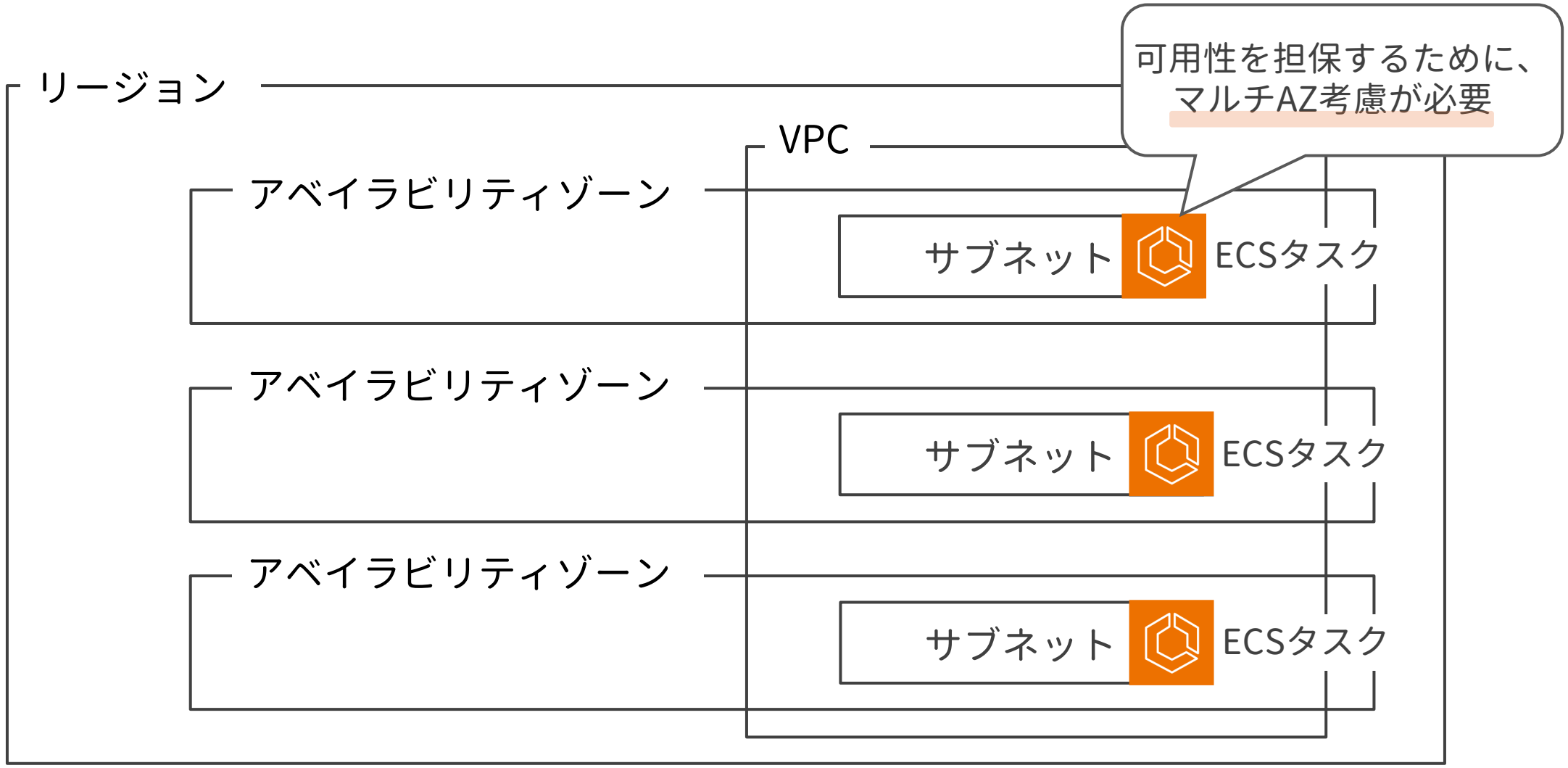


おさらい）AWSにおけるVPCのスコープ

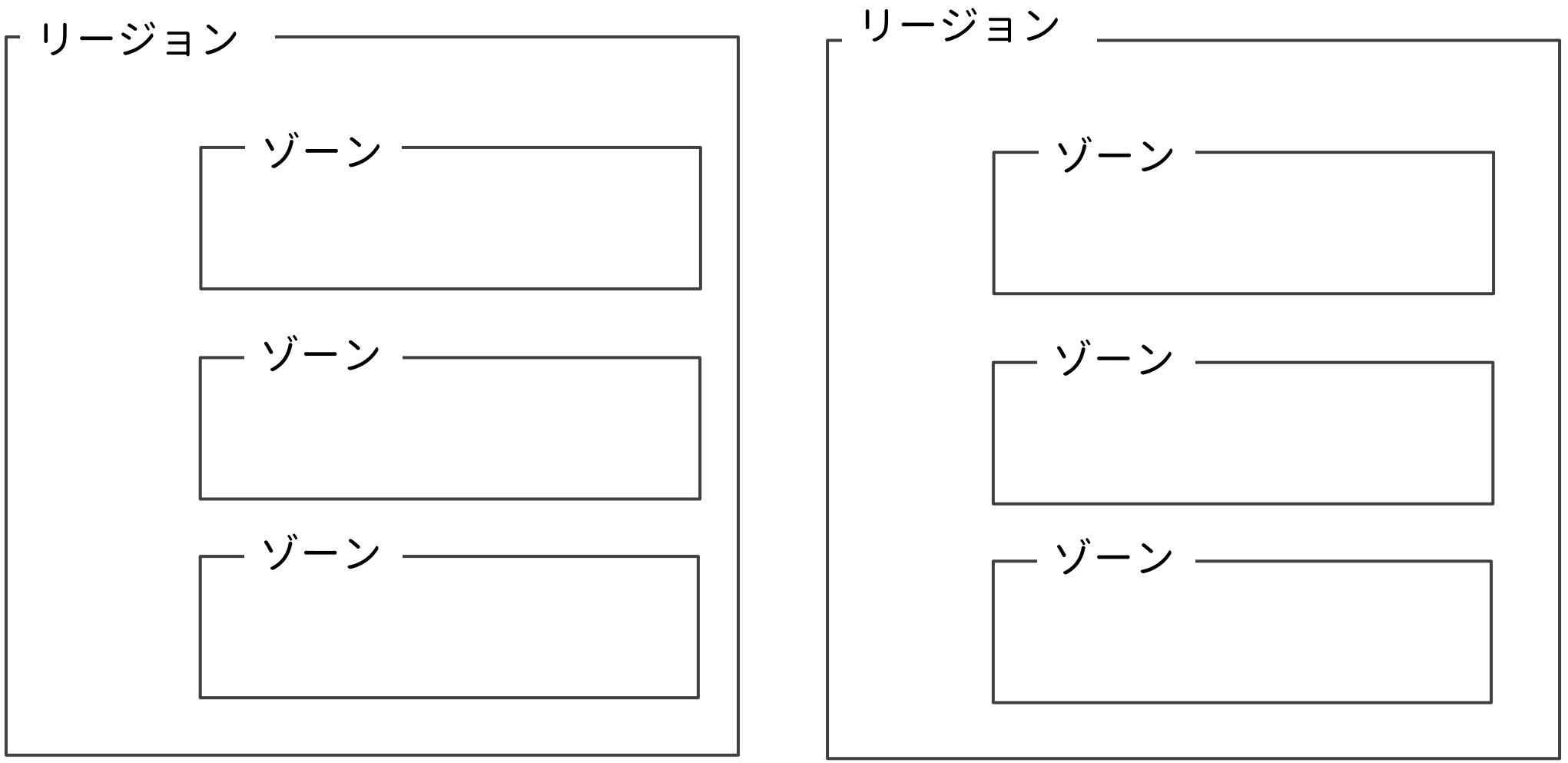


基本的な各サービスの特徴 - VPC内外

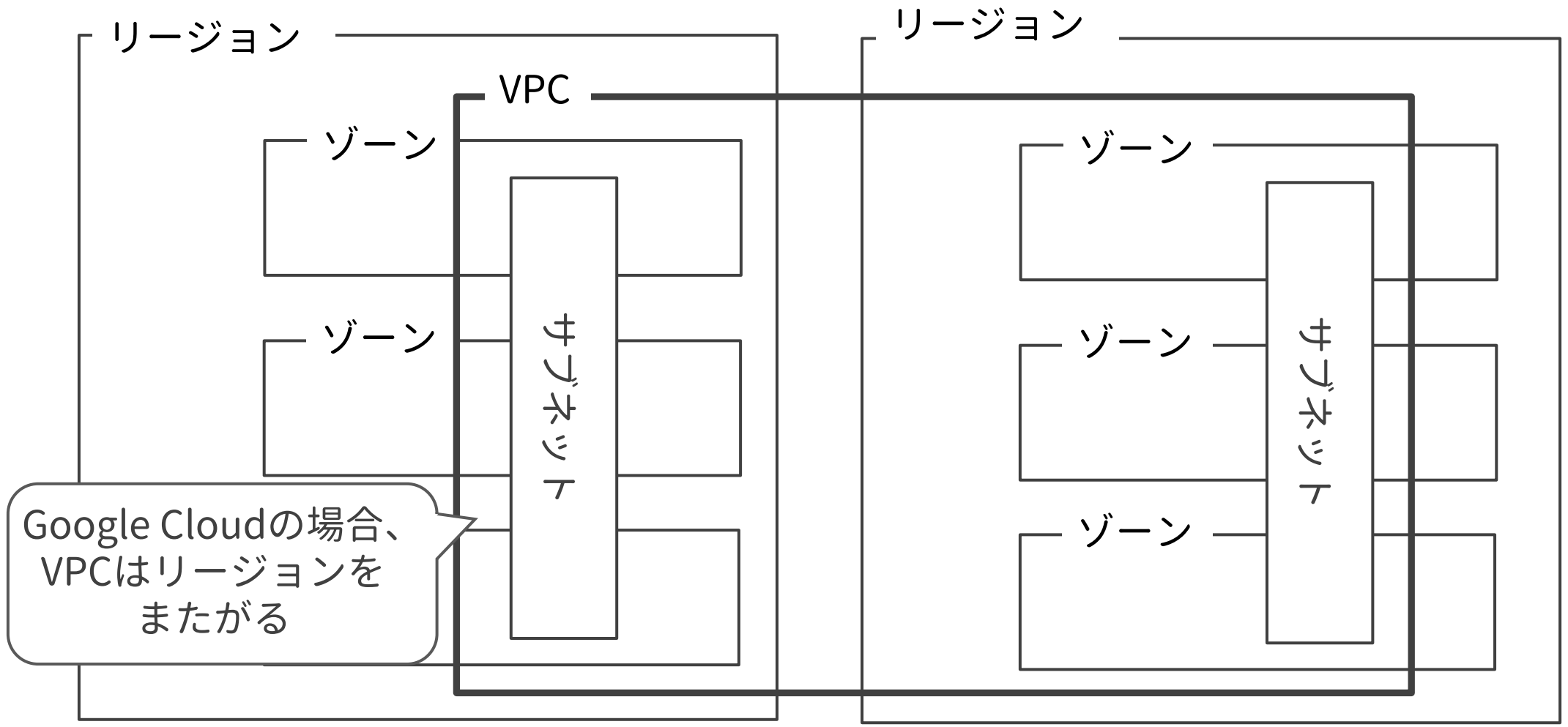
ECSタスクはVPC内サブネット上にデプロイされるリソース



おさらい）Google CloudにおけるVPCのスコープ

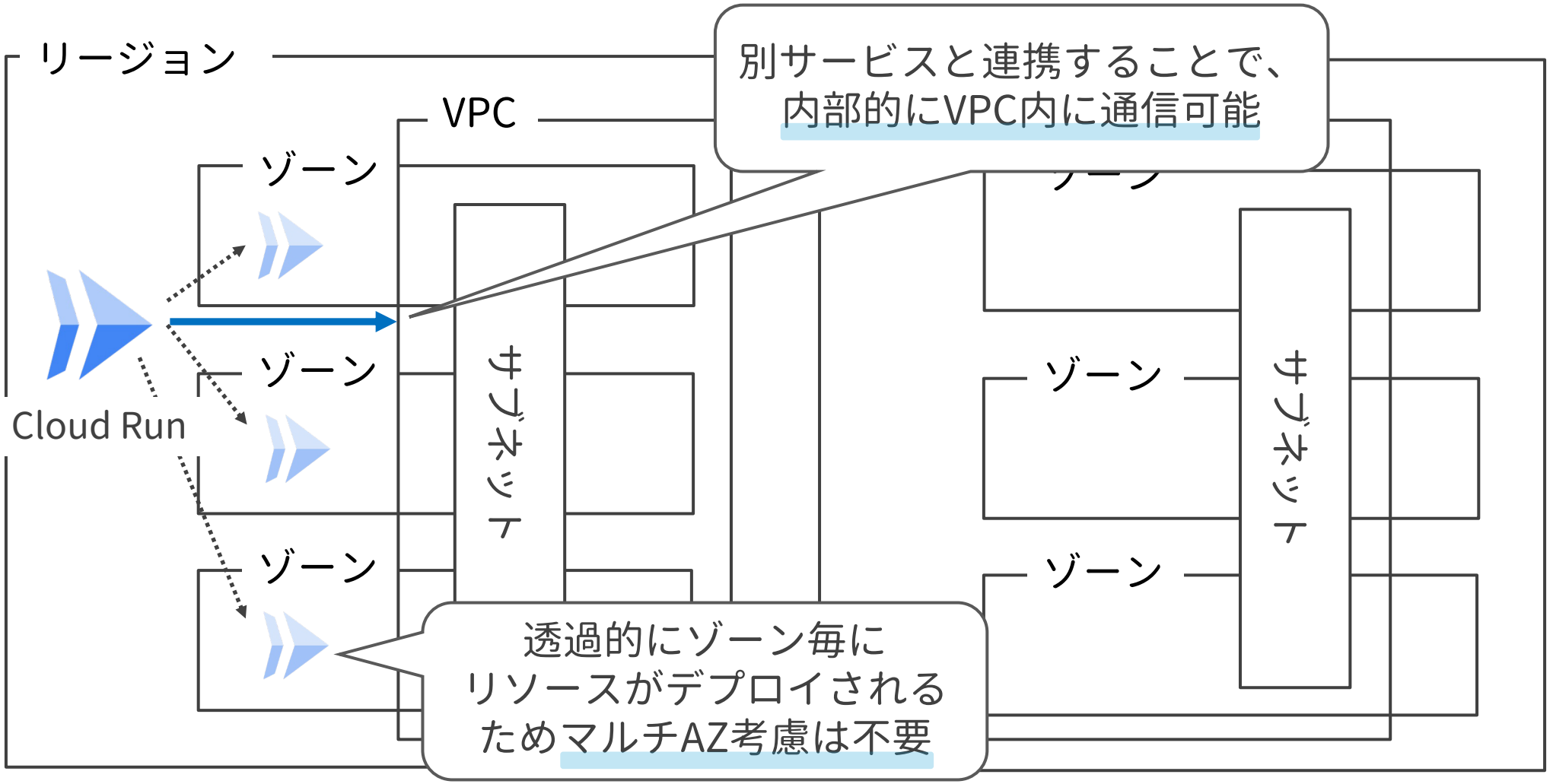


おさらい）Google CloudにおけるVPCのスコープ



基本的な各サービスの特徴 - VPC内外

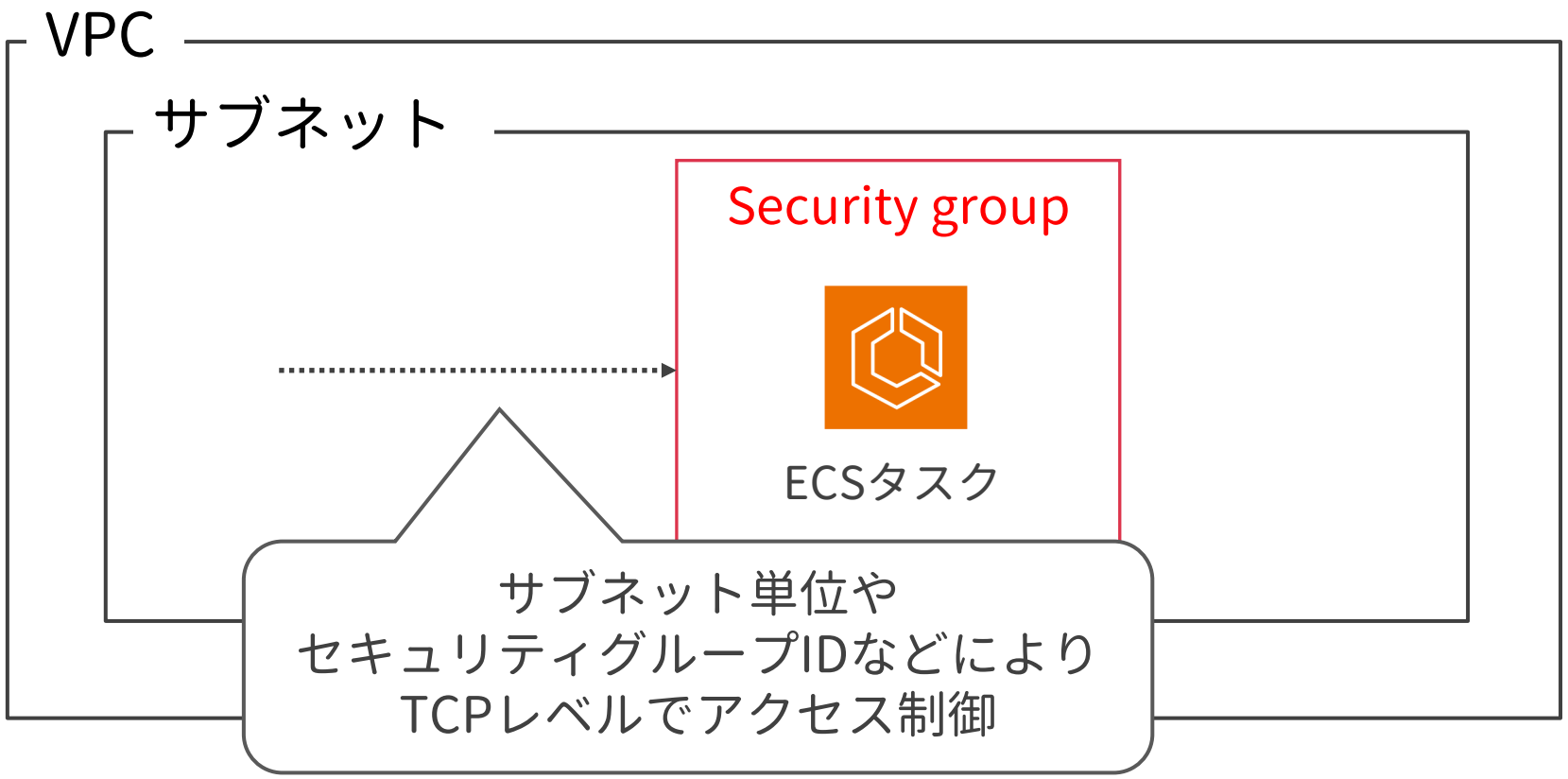
Cloud RunはVPC外にデプロイされるリソース



各サービスの基本的な特徴

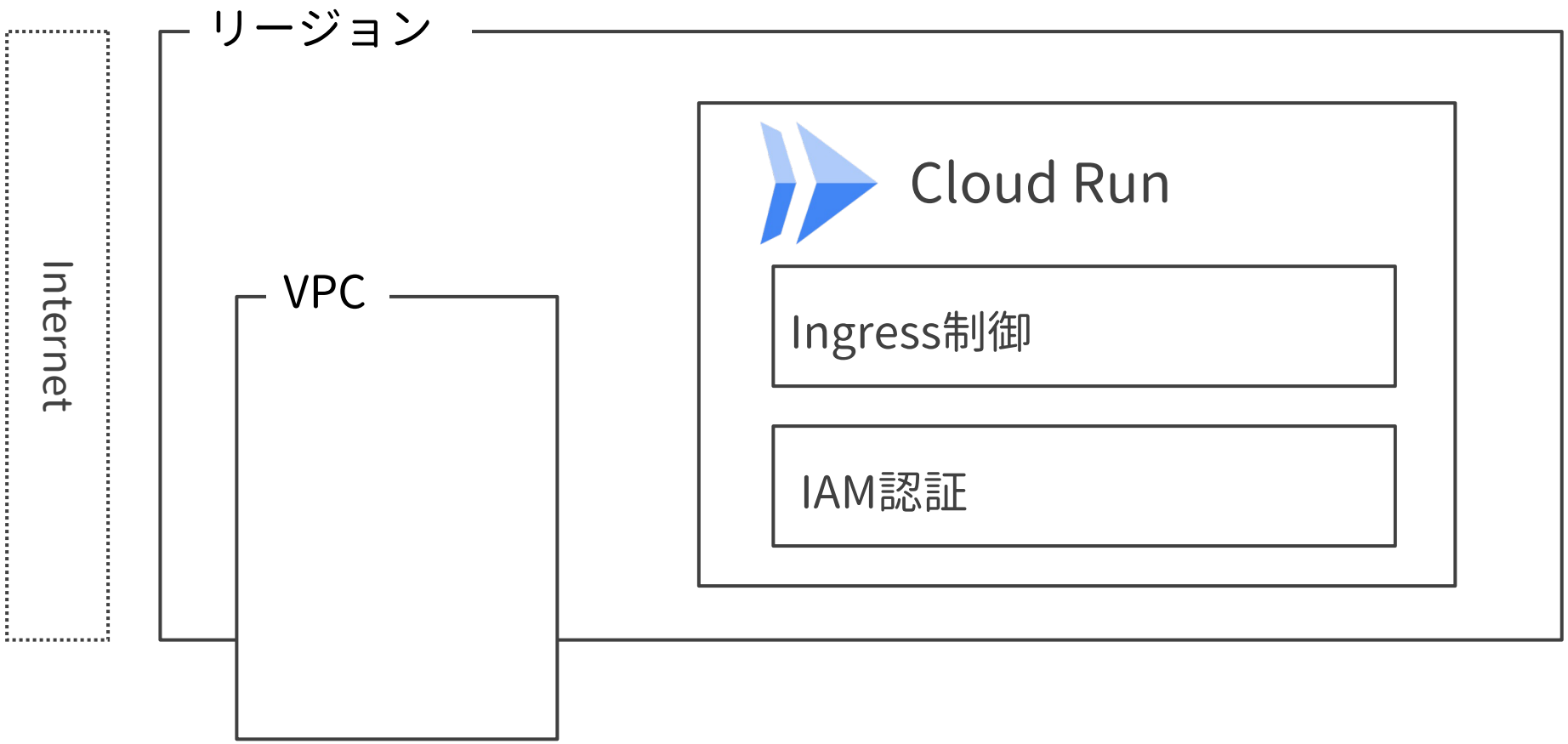
1. コンポーネントとワークロード
2. VPC内外
3. アクセス制御
4. コンテナ実行環境

ECSのHTTP(S)リクエストはセキュリティグループにて制御



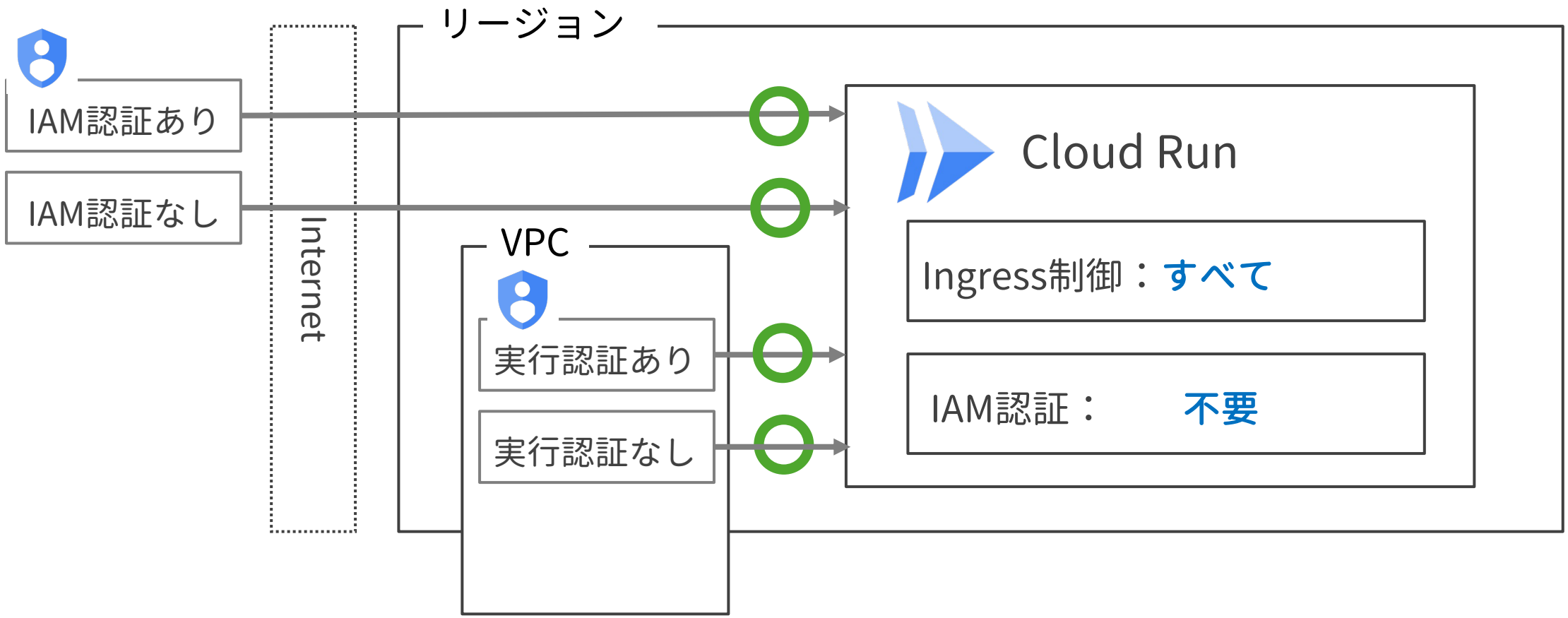
基本的な各サービスの特徴 - アクセス制御

Cloud RunのHTTPSリクエストはIngress制御+IAM認証で制御



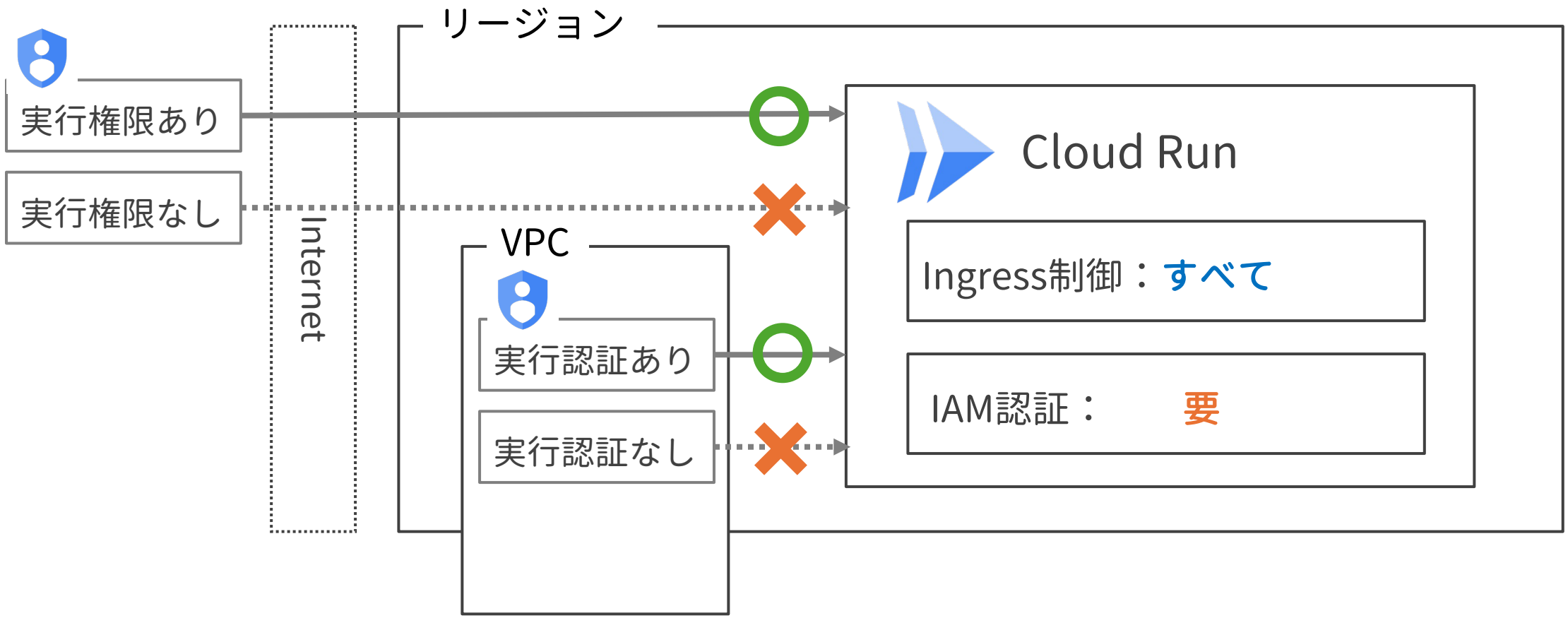
基本的な各サービスの特徴 - アクセス制御

Cloud RunのHTTPSリクエストはIngress制御+IAM認証で制御



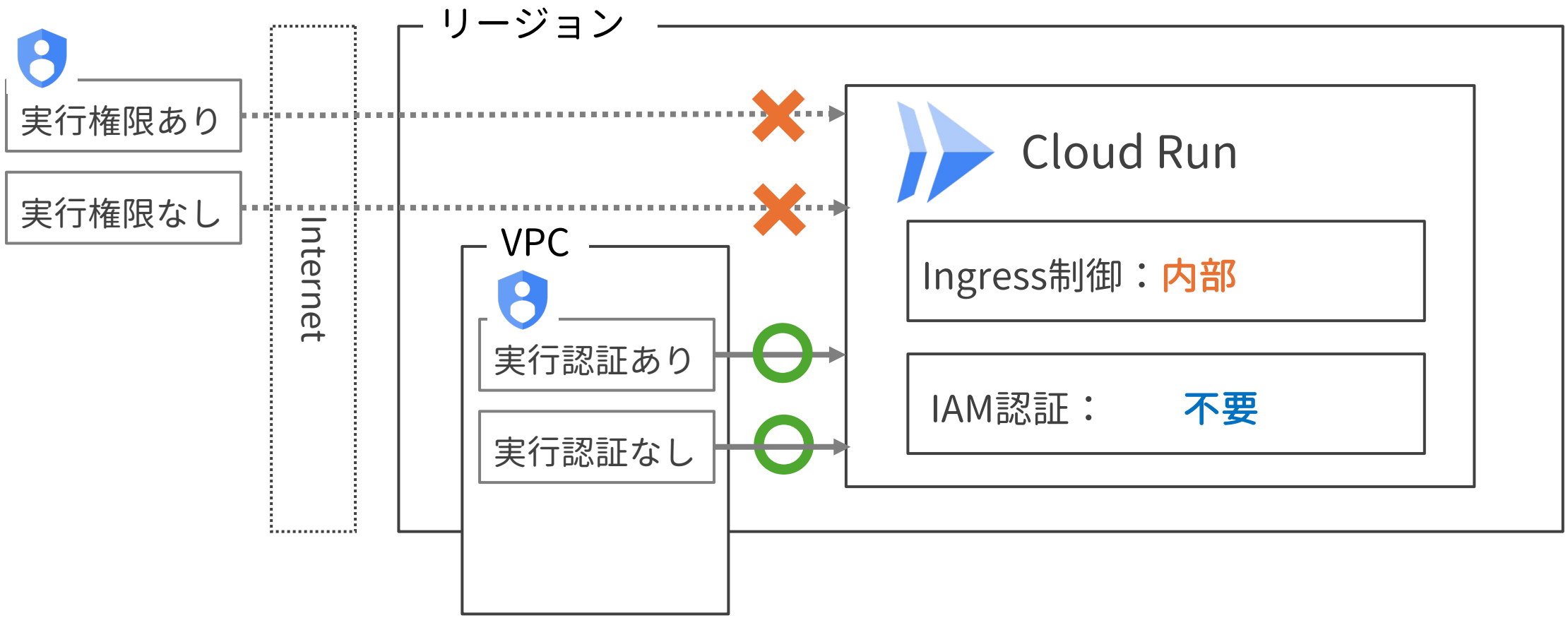
基本的な各サービスの特徴 - アクセス制御

Cloud RunのHTTPSリクエストはIngress制御+IAM認証で制御



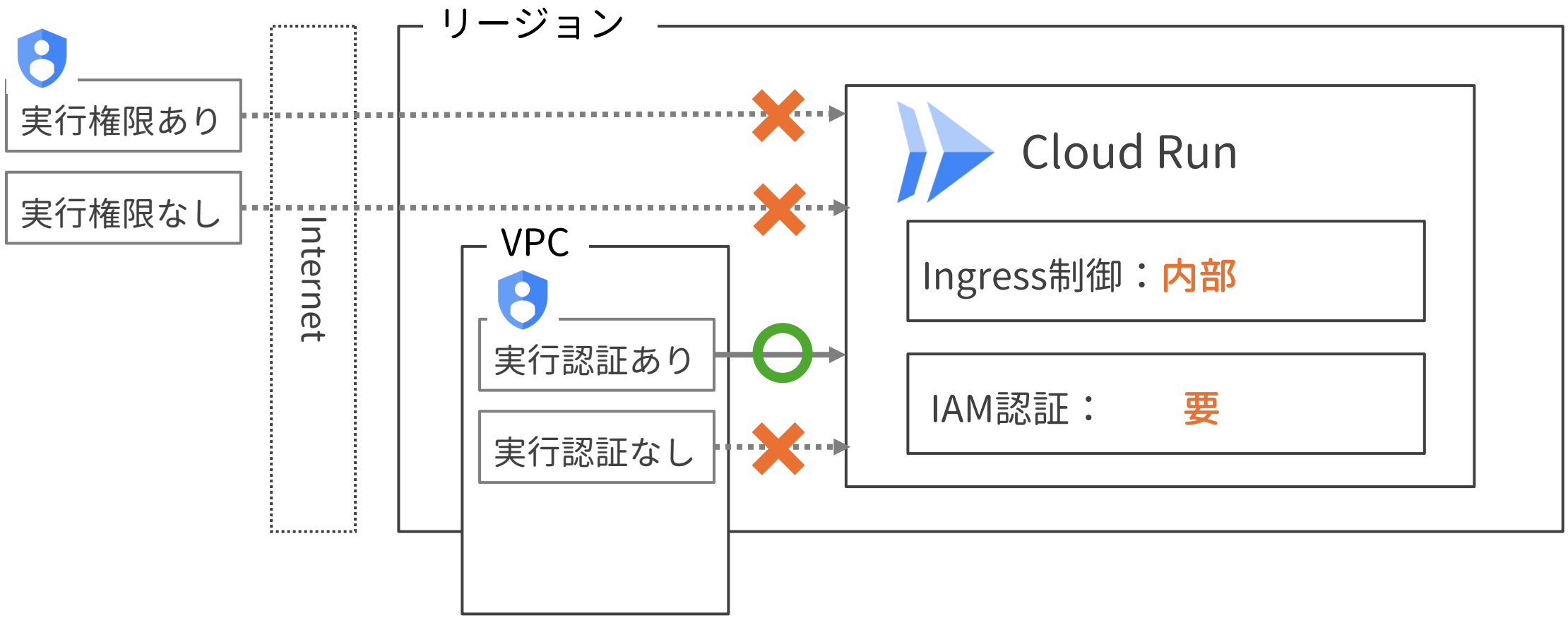
基本的な各サービスの特徴 - アクセス制御

Cloud RunのHTTPSリクエストはIngress制御+IAM認証で制御



基本的な各サービスの特徴 - アクセス制御

Cloud RunのHTTPSリクエストはIngress制御+IAM認証で制御



各サービスの基本的な特徴

1. コンポーネントとワークロード
2. VPC内外
3. アクセス制御
4. コンテナ実行環境

ECS/FargateはFirecrackerというmicroVMで起動

ECS/FargateはFirecrackerというmicroVMで起動

- KVMベースの仮想化
  ※同一マシン上で別VMとして分離

ECS/FargateはFirecrackerというmicroVMで起動

- KVMベースの仮想化
  ※同一マシン上で別VMとして分離
- GoogleのChromium OSで使われるcrosvmをフォークして再利用(※)
  ※seccomp-bpfやキャッシュによるデータ転送面で利点

(※)https://firecracker-microvm.github.io/

ECS/FargateはFirecrackerというmicroVMで起動

- KVMベースの仮想化
  ※同一マシン上で別VMとして分離
- GoogleのChromium OSで使われるcrosvmをフォークして再利用(※)
  ※seccomp-bpfやキャッシュによるデータ転送面で利点
- LambdaやApp Runner等でも利用

(※)https://firecracker-microvm.github.io/

ECS/FargateはFirecrackerというmicroVMで起動

Cloud Runは世代により実行環境が異なる

Cloud Runは世代により実行環境が異なる

- 第１世代はgVisorベース
  - コールドスタートが早い
  - 一部のLinuxシステムコールと非互換(※1)

(※1)https://gvisor.dev/docs/user_guide/compatibility/linux/amd64/

Cloud Runは世代により実行環境が異なる

- 第１世代はgVisorベース
  - コールドスタートが早い
  - 一部のLinuxシステムコールと非互換(※1)
- 第２世代はmicroVMベース(※2)
  - CPUやネットワークのパフォーマンスが改善
  - 第２世代はLinuxシステムコールと完全互換

(※1)https://gvisor.dev/docs/user_guide/compatibility/linux/amd64/

(※2)Cloud Run jobsは第２世代の実行環境のみ

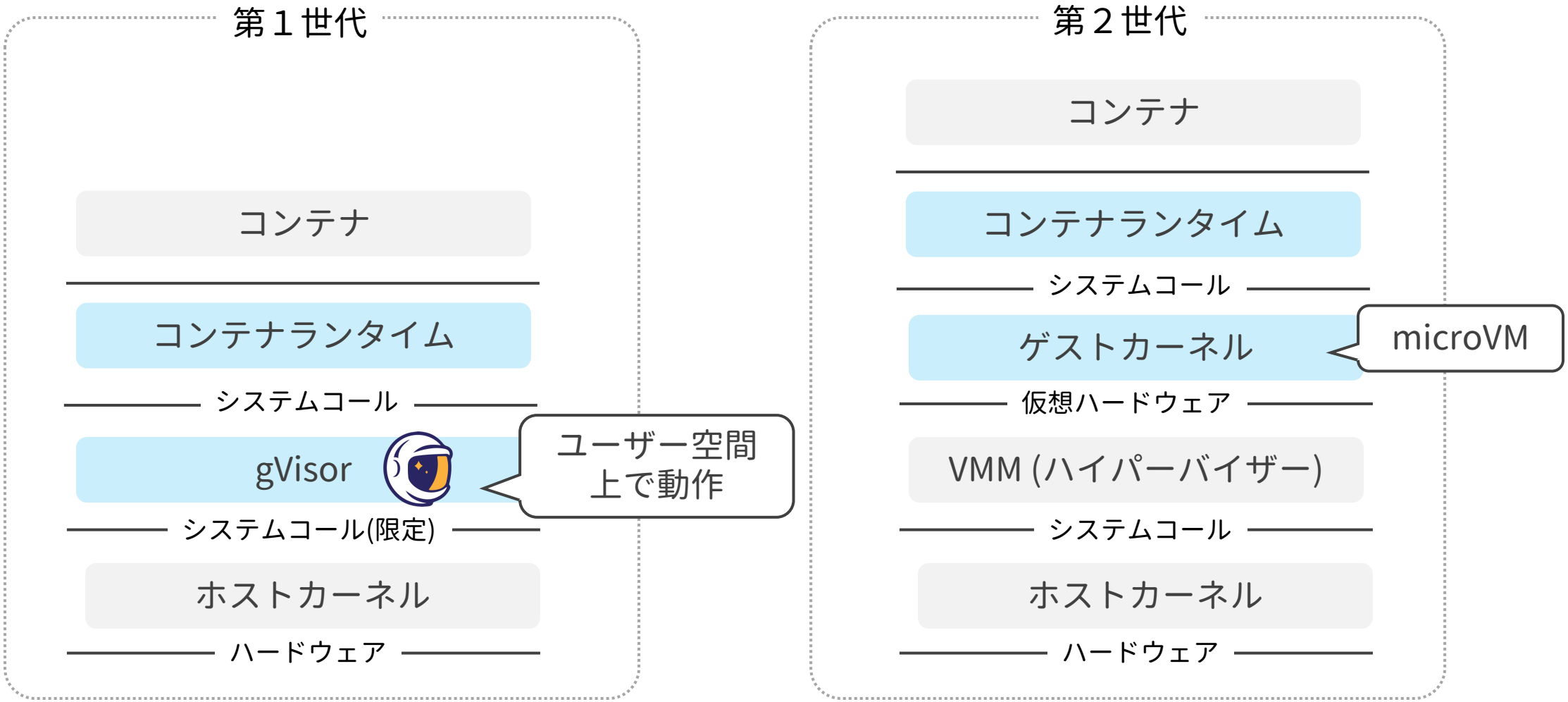
Cloud Runは世代により実行環境が異なる

- 第１世代はgVisorベース
  - コールドスタートが早い
  - 一部のLinuxシステムコールと非互換(※1)
- 第２世代はmicroVMベース(※2)
  - CPUやネットワークのパフォーマンスが改善
  - 第２世代はLinuxシステムコールと完全互換
- いずれの世代もBorg基盤上のKnative互換Appとして稼働
  ※Knativeそのものではなく、Knative互換のAPIを提供

(※1)https://gvisor.dev/docs/user_guide/compatibility/linux/amd64/

(※2)Cloud Run jobsは第２世代の実行環境のみ

世代毎のCloud Runコンテナ実行環境の違い



各サービスの基本的な特徴

ここまでのまとめ

1. コンポーネントとワークロード
2. VPC内外
3. アクセス制御
4. コンテナ実行環境

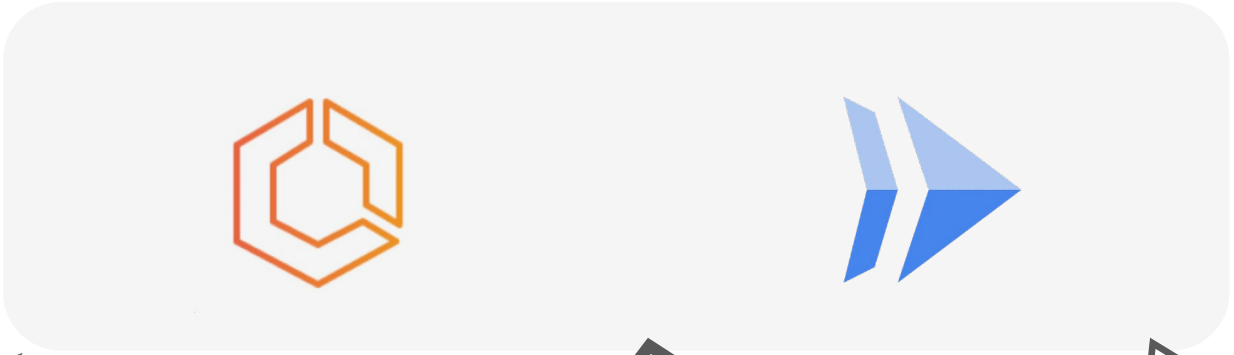
各サービスの基本的な特徴

ここまでのまとめ

1 コンポーネントとワークロード	コンテナ、タスク、サービス、クラスター	コンテナ、インスタンス、リビジョン、サービス
2 VPC内外	VPC内	VPC外
3 アクセス制御	セキュリティグループ	Ingress制御 + IAM認証
4 コンテナ実行環境	microVM (Firecracker)	gVisor / microVM

アジェンダ

以下の3つの観点から相互理解を目指す



1. 基本的な特徴

2. アーキテクチャデザインの違い

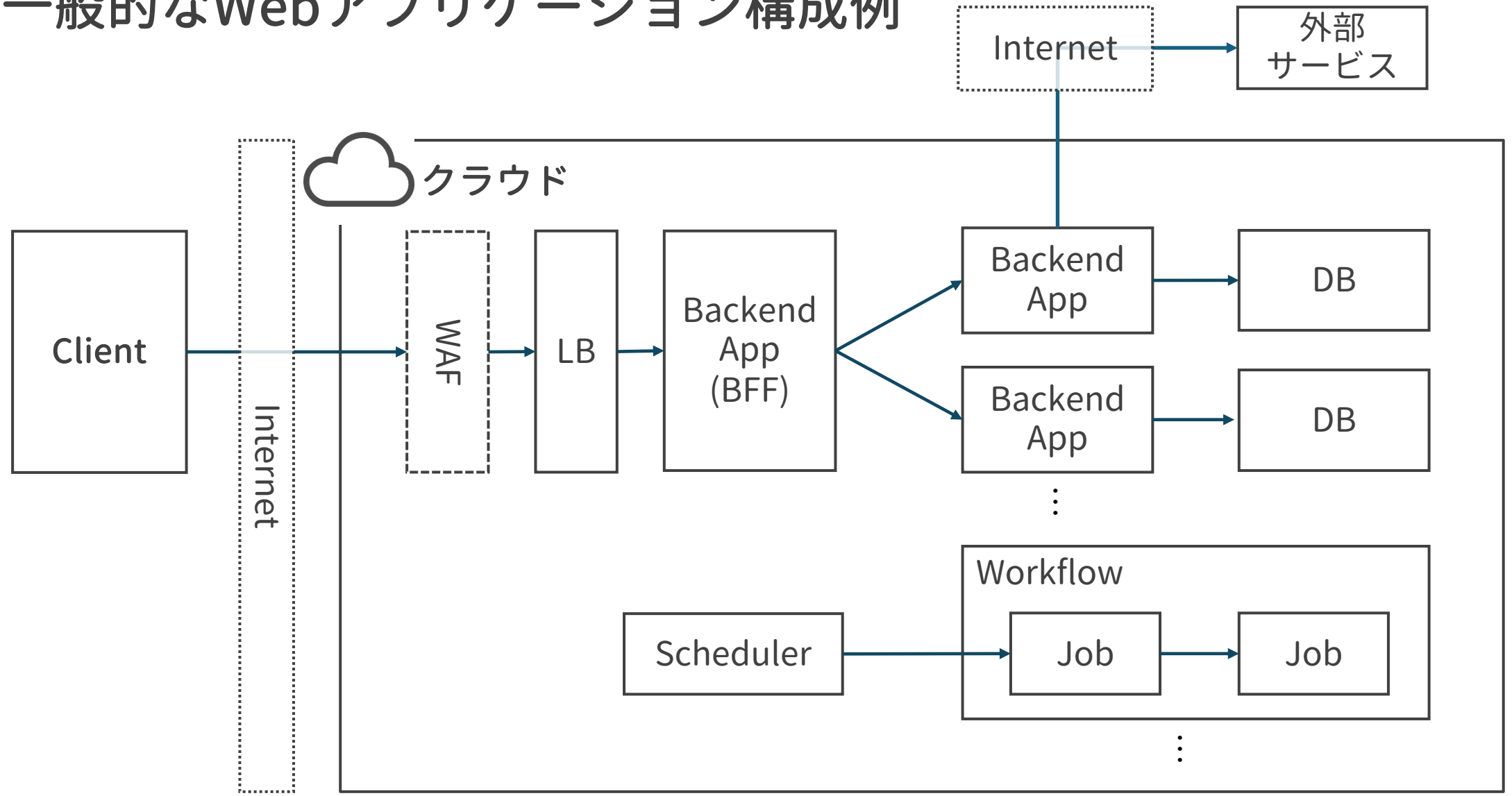
3. 非機能デザインからの理解

アーキテクチャデザインの違い

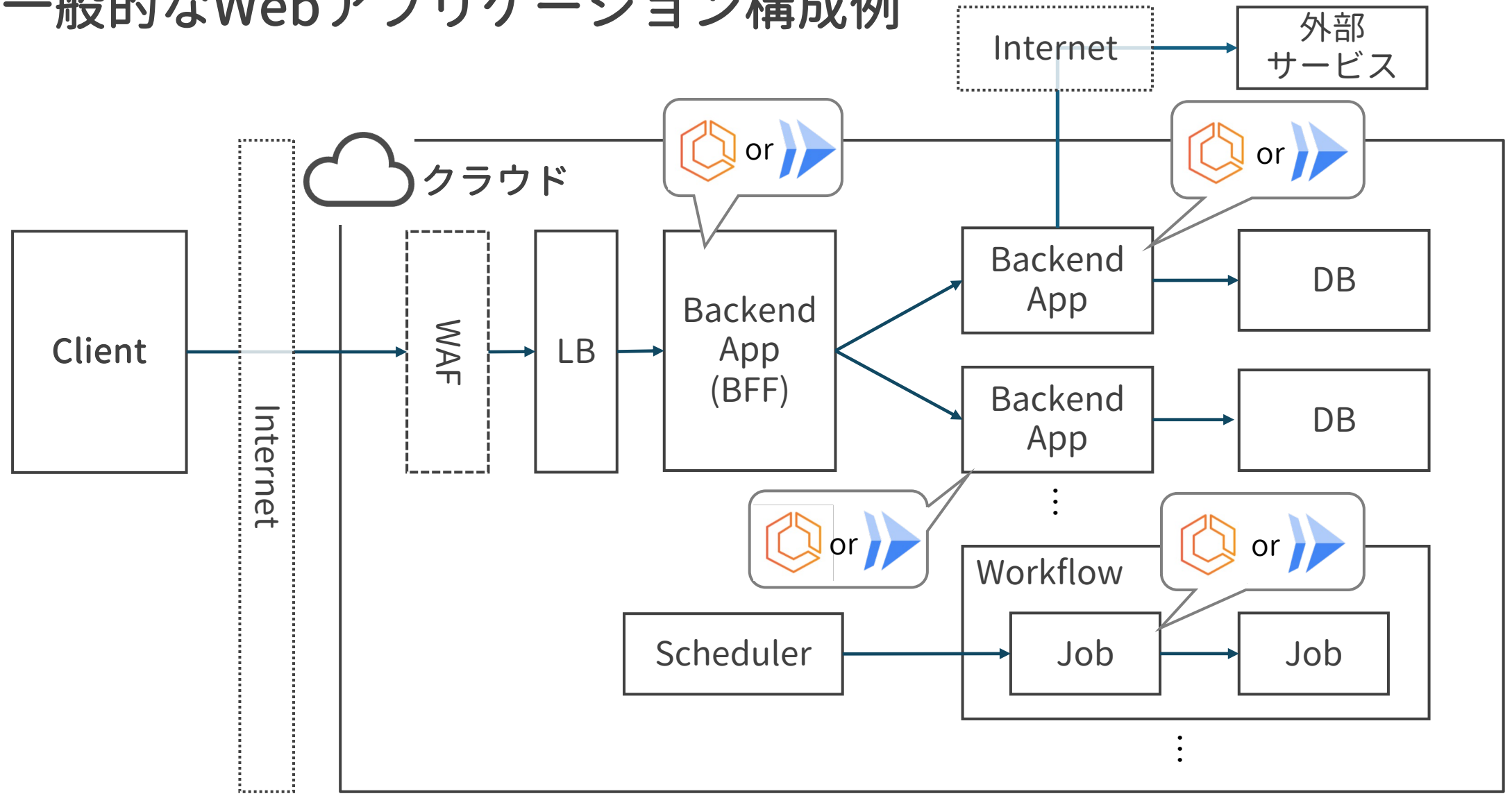
一般的なWebアプリケーション構成を例に
内容を追っていく

アーキテクチャデザインの違い

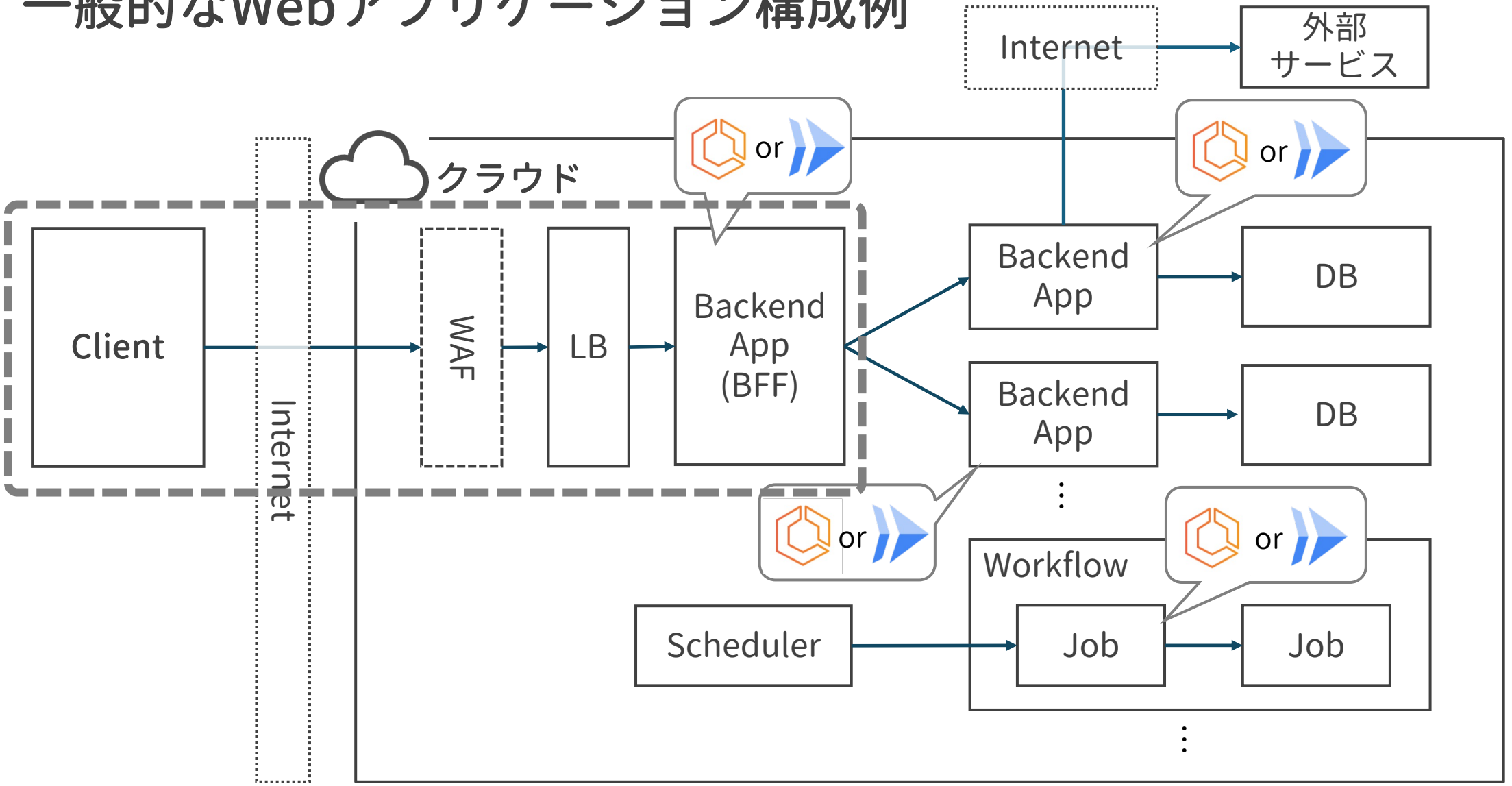
一般的なWebアプリケーション構成例



一般的なWebアプリケーション構成例

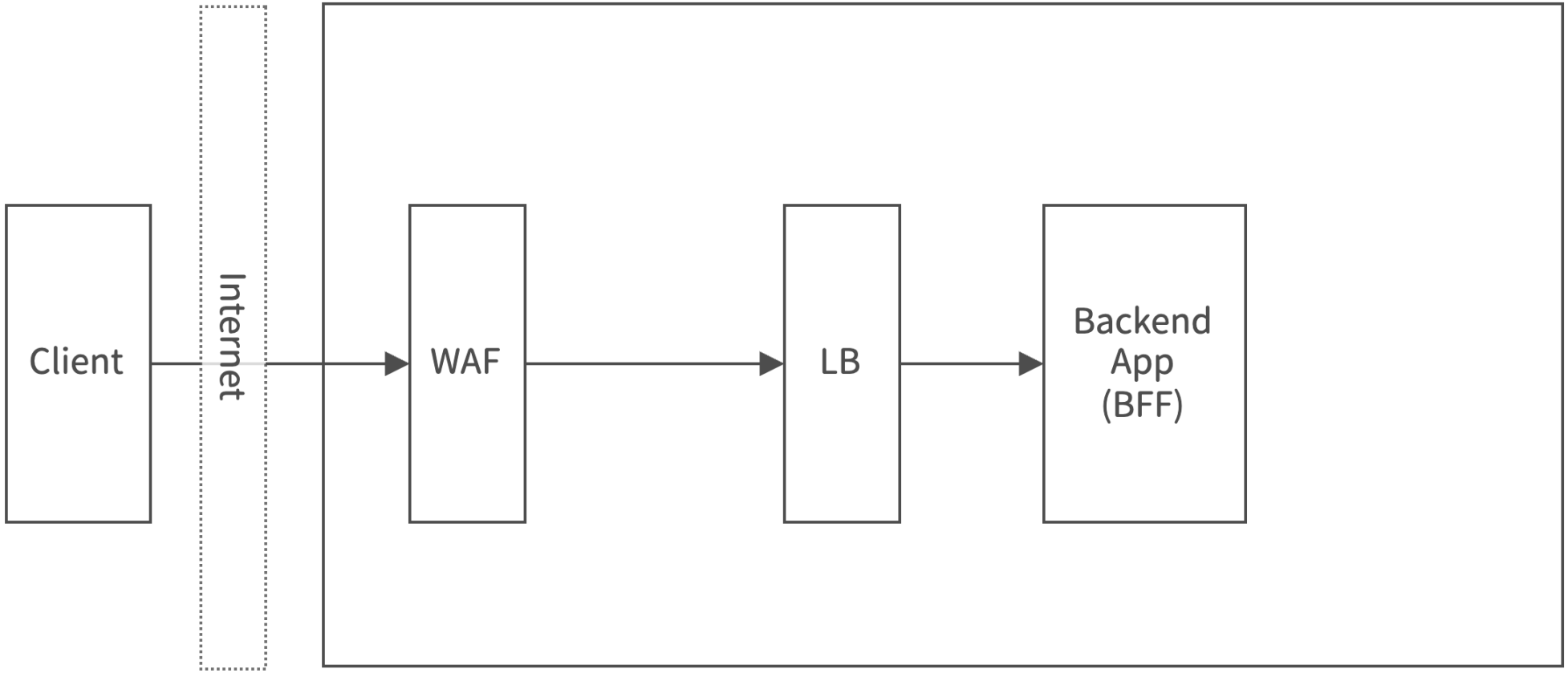


一般的なWebアプリケーション構成例



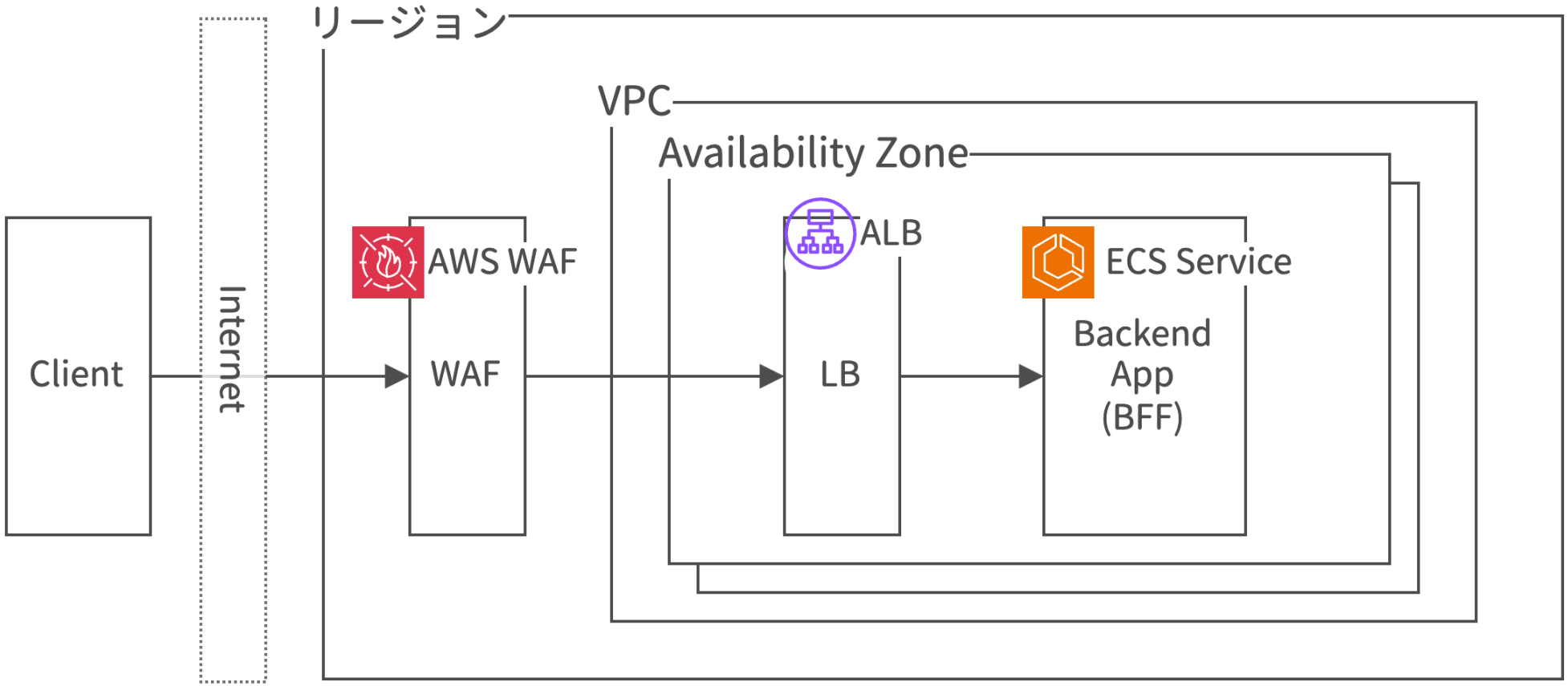
アーキテクチャデザインの違い – インターネットアクセス

セキュリティや可用性観点からWAFやロードバランサを配置する構成は共通



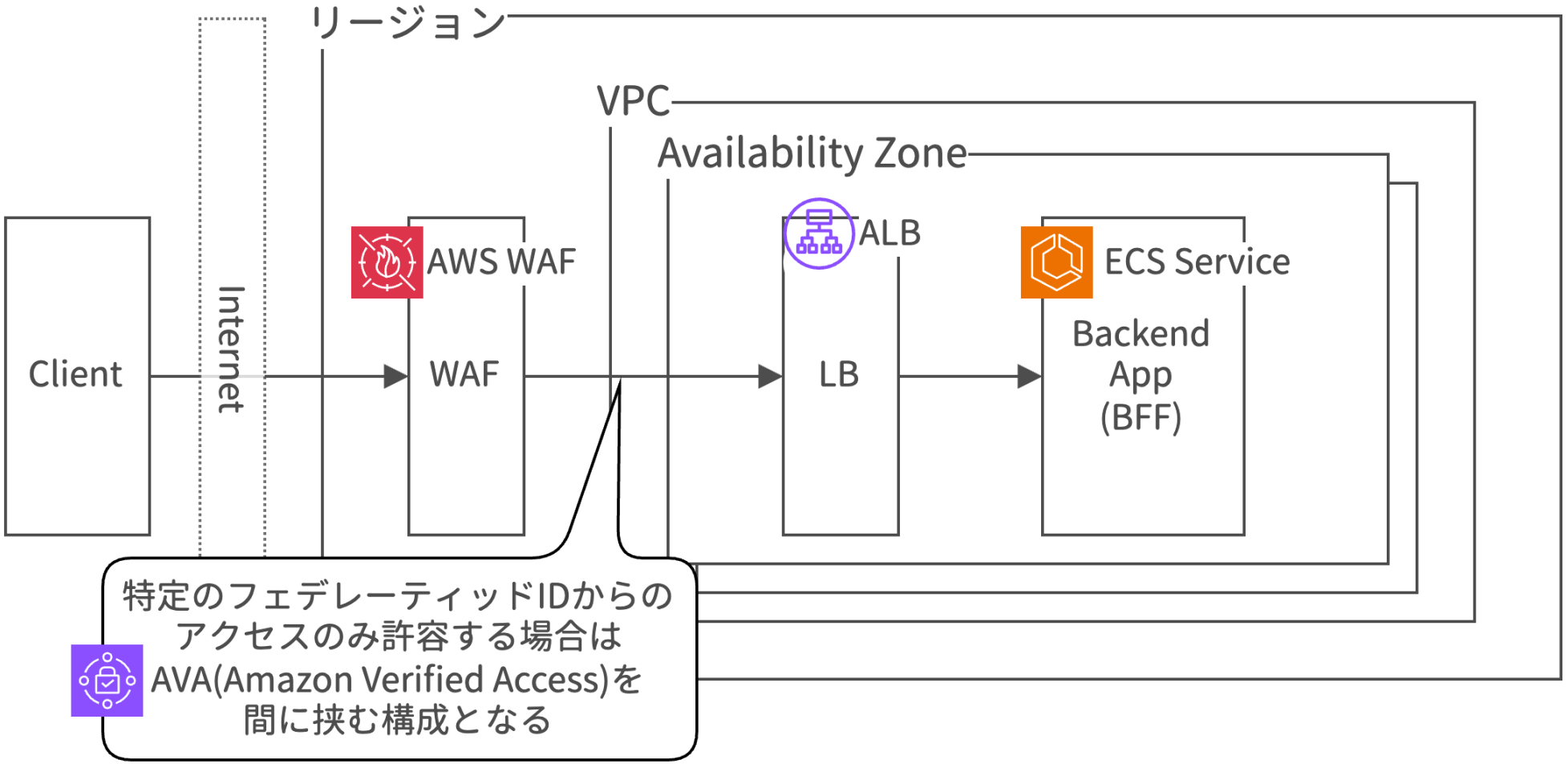
アーキテクチャデザインの違い – インターネットアクセス

セキュリティや可用性観点からWAFやロードバランサを配置する構成は共通



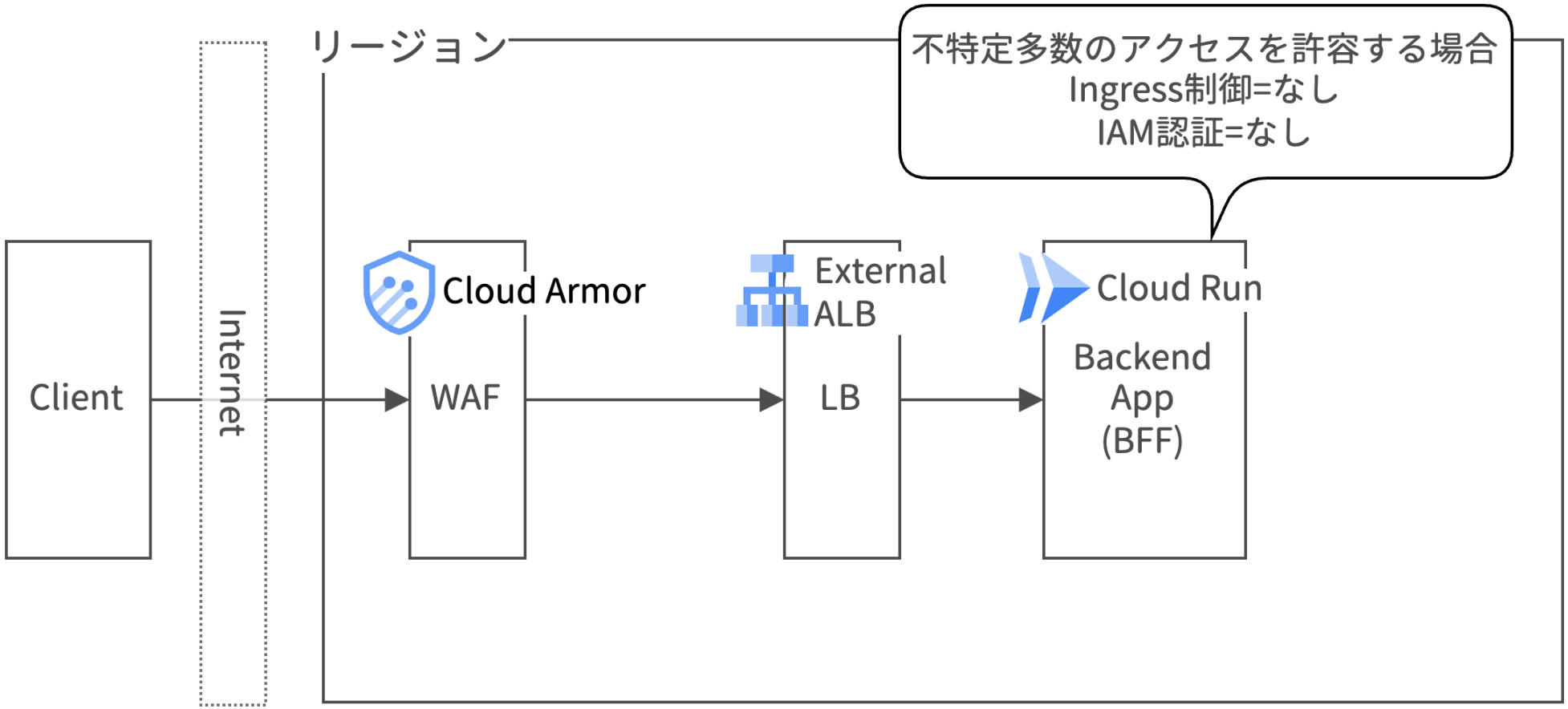
アーキテクチャデザインの違い – インターネットアクセス

セキュリティや可用性観点からWAFやロードバランサを配置する構成は共通



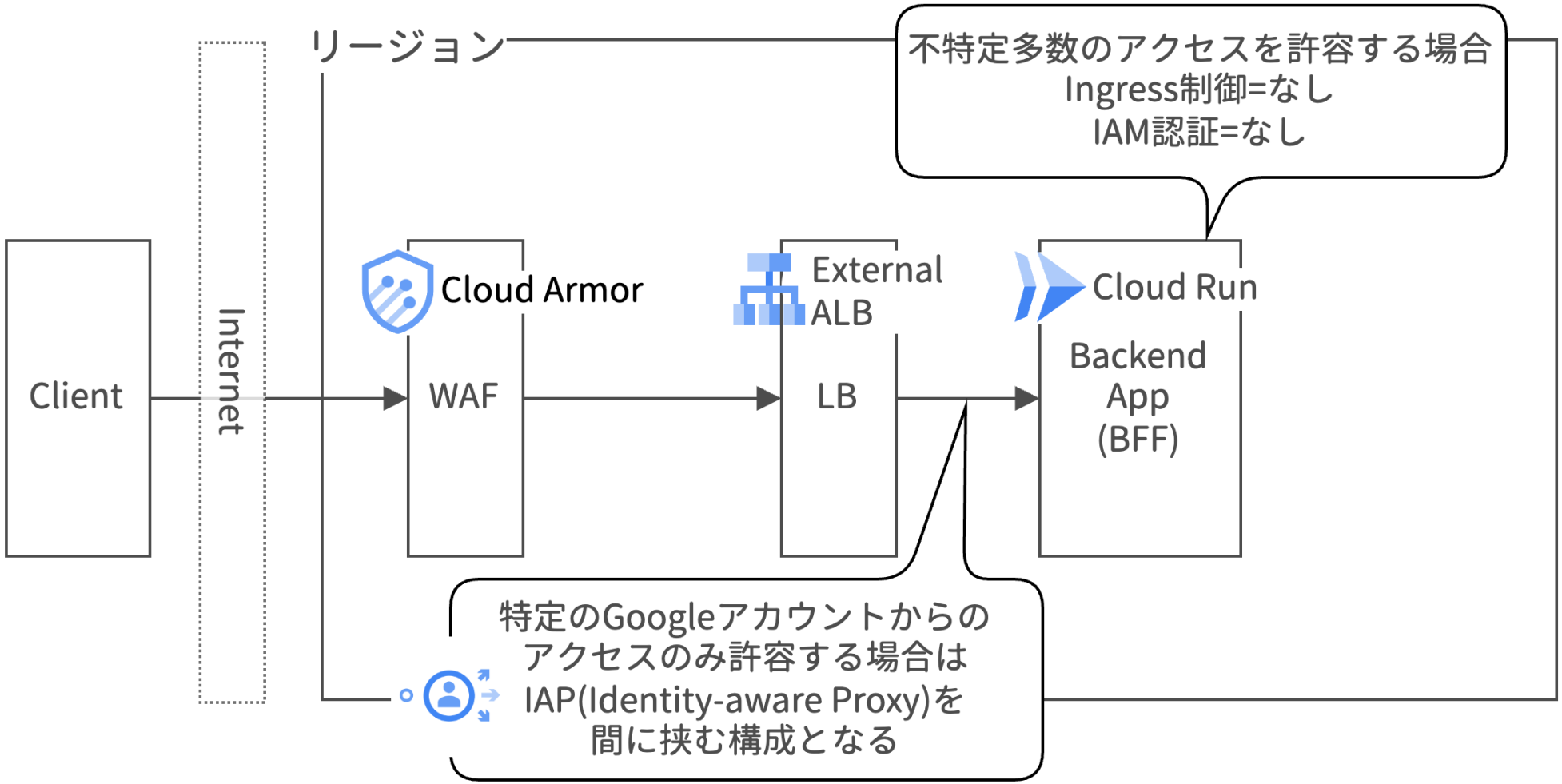
アーキテクチャデザインの違い – インターネットアクセス

セキュリティや可用性観点からWAFやロードバランサを配置する構成は共通



アーキテクチャデザインの違い – インターネットアクセス

セキュリティや可用性観点からWAFやロードバランサを配置する構成は共通

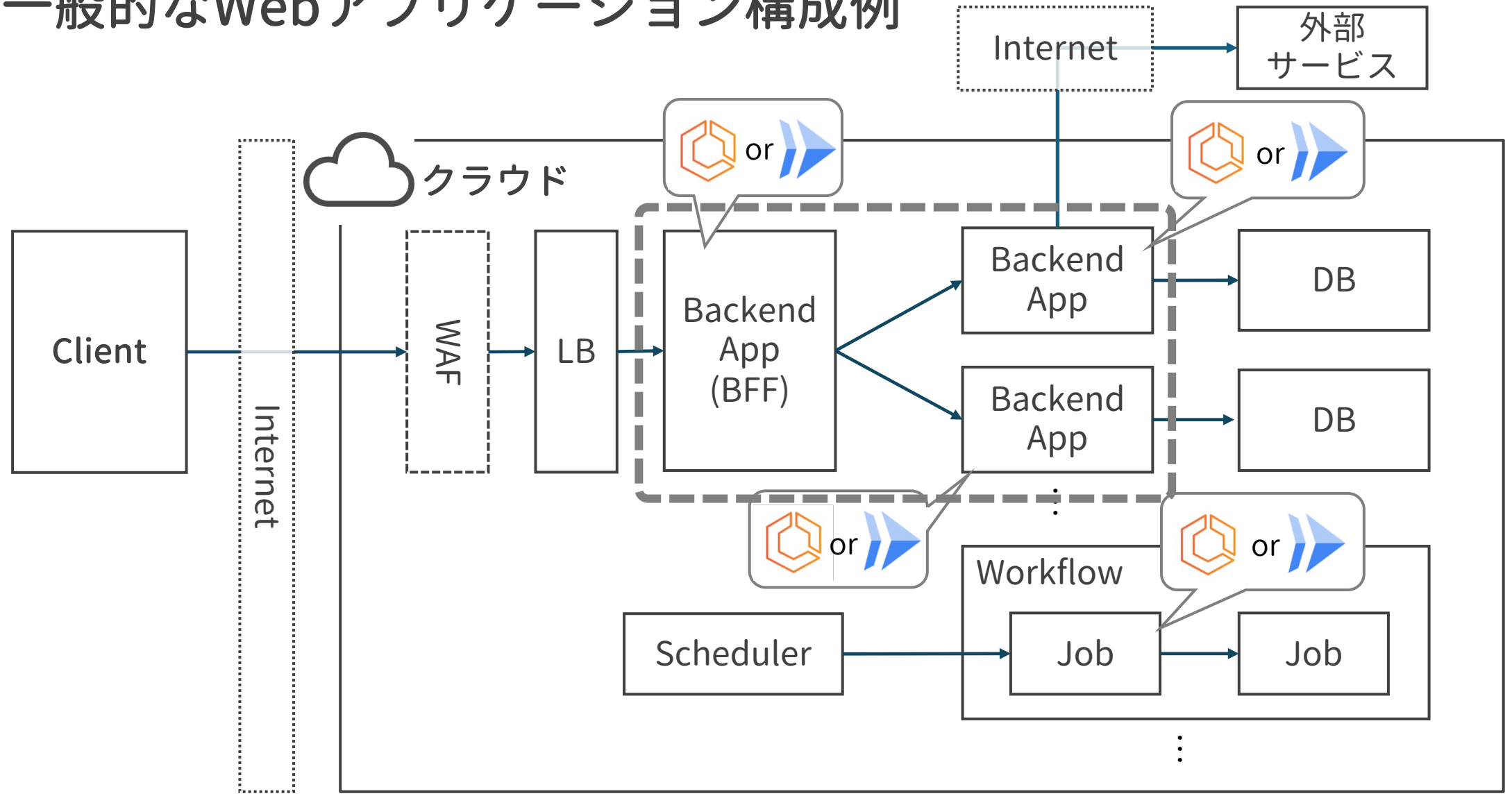


(※)2024年11月28日時点ではプレビューだが、Cloud Runに直接IAPの紐づけも可能

まとめ

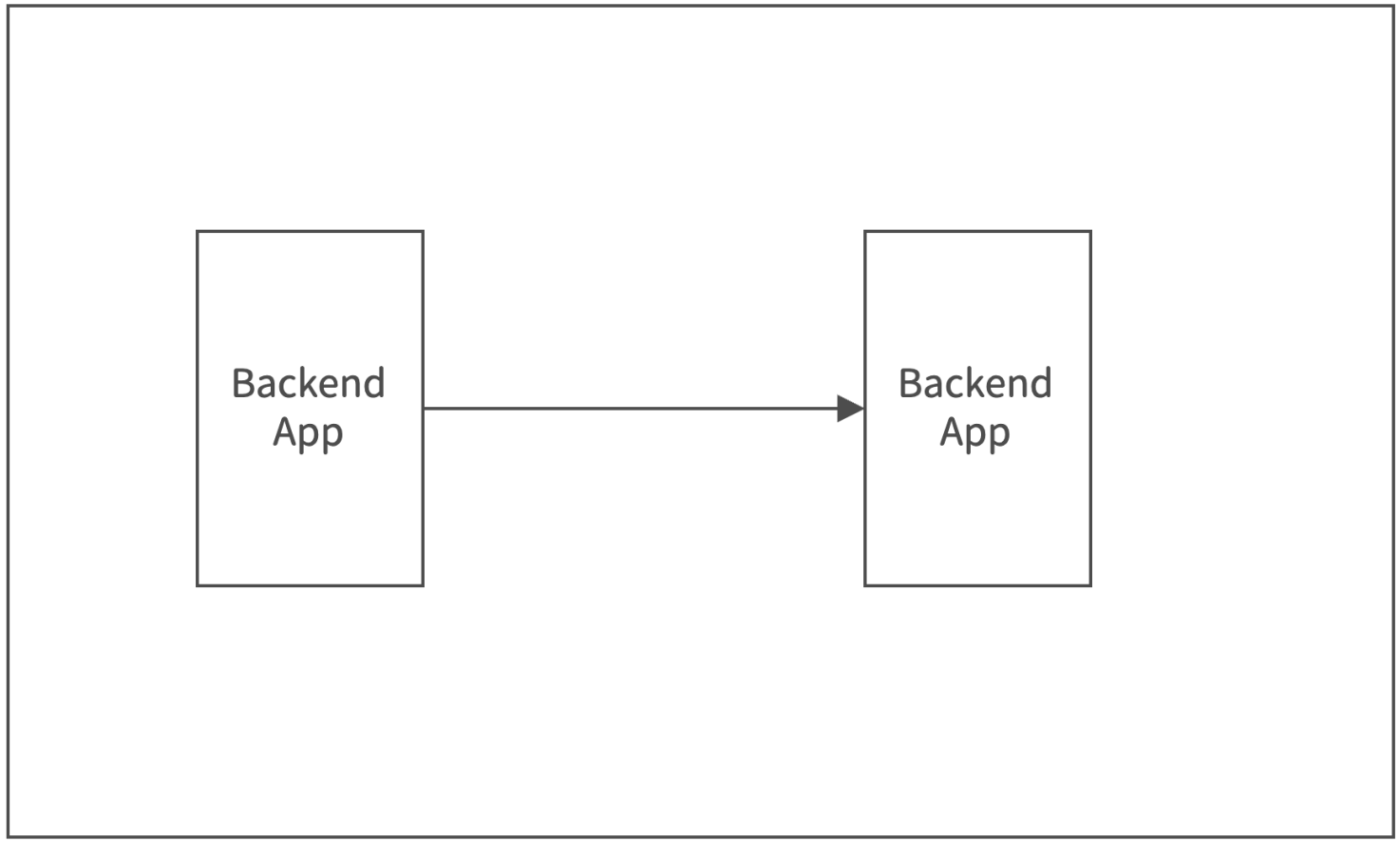
Amazon ECS	Cloud Run
• ロードバランサとしてALB、WAFとしてAWS WAFを配置	• ロードバランサとしてExternal ALB、WAFとしてCloud Armorを配置
• フェデレーティッドIDからのアクセスのみ許容する場合はAmazon Verified Accessと連携	• 特定Googleアカウントからのリクエストのみを許容する場合はIAP(Identity-aware Proxy)と連携

一般的なWebアプリケーション構成例

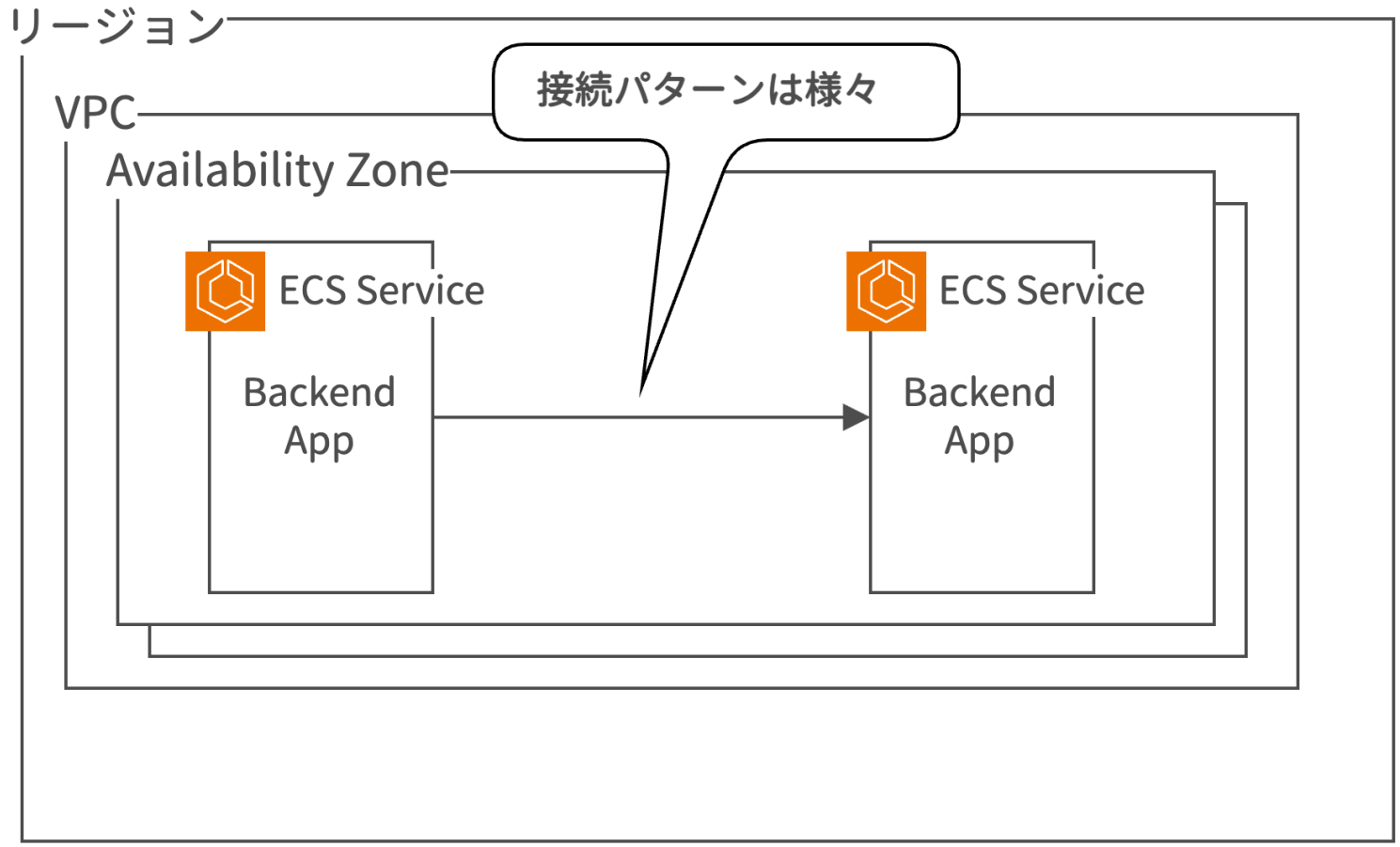


アーキテクチャデザインの違い – サービス間内部通信

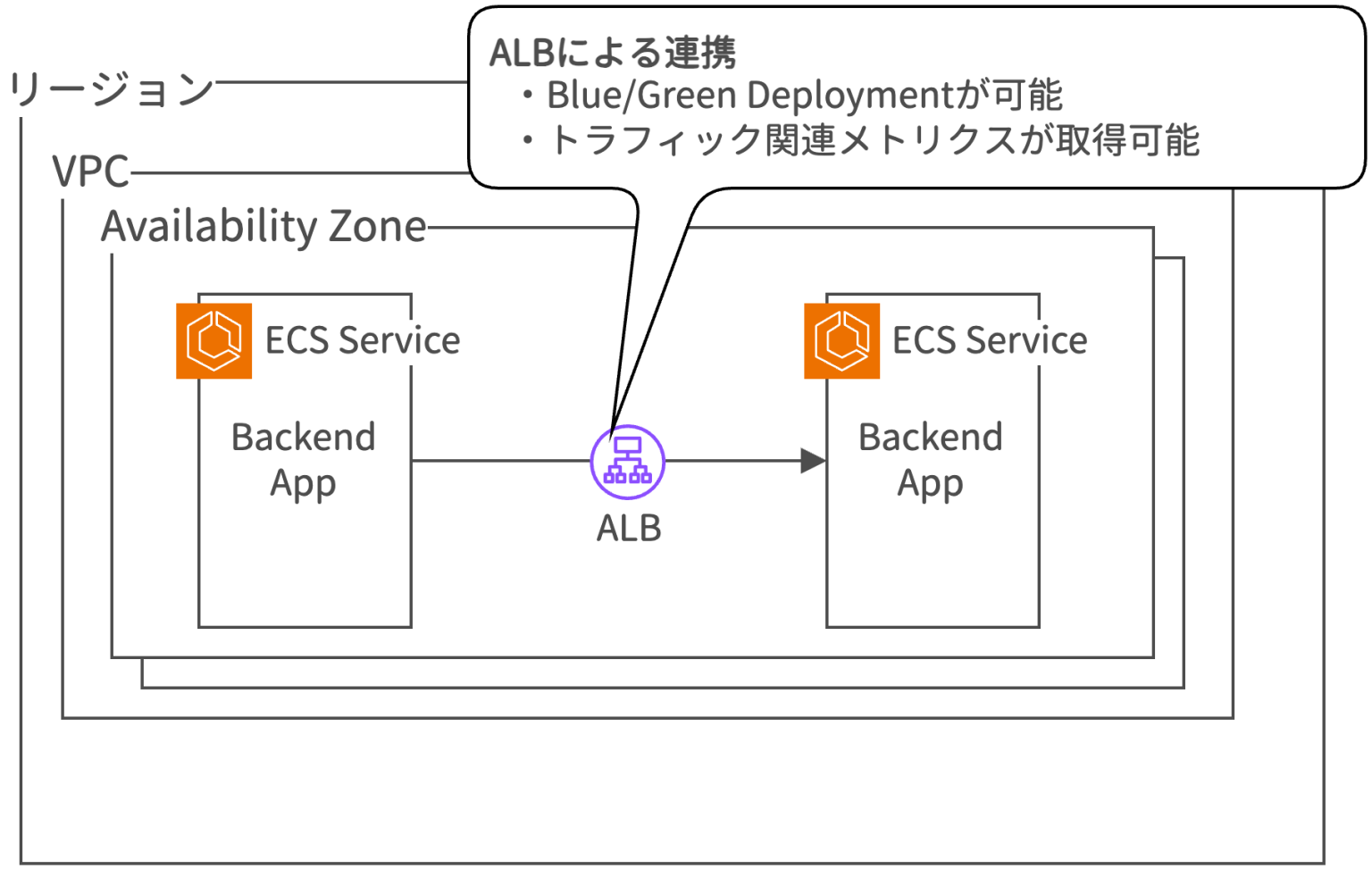
ECSとCloud Runで通信方法とアクセス制御方法が大きく異なる



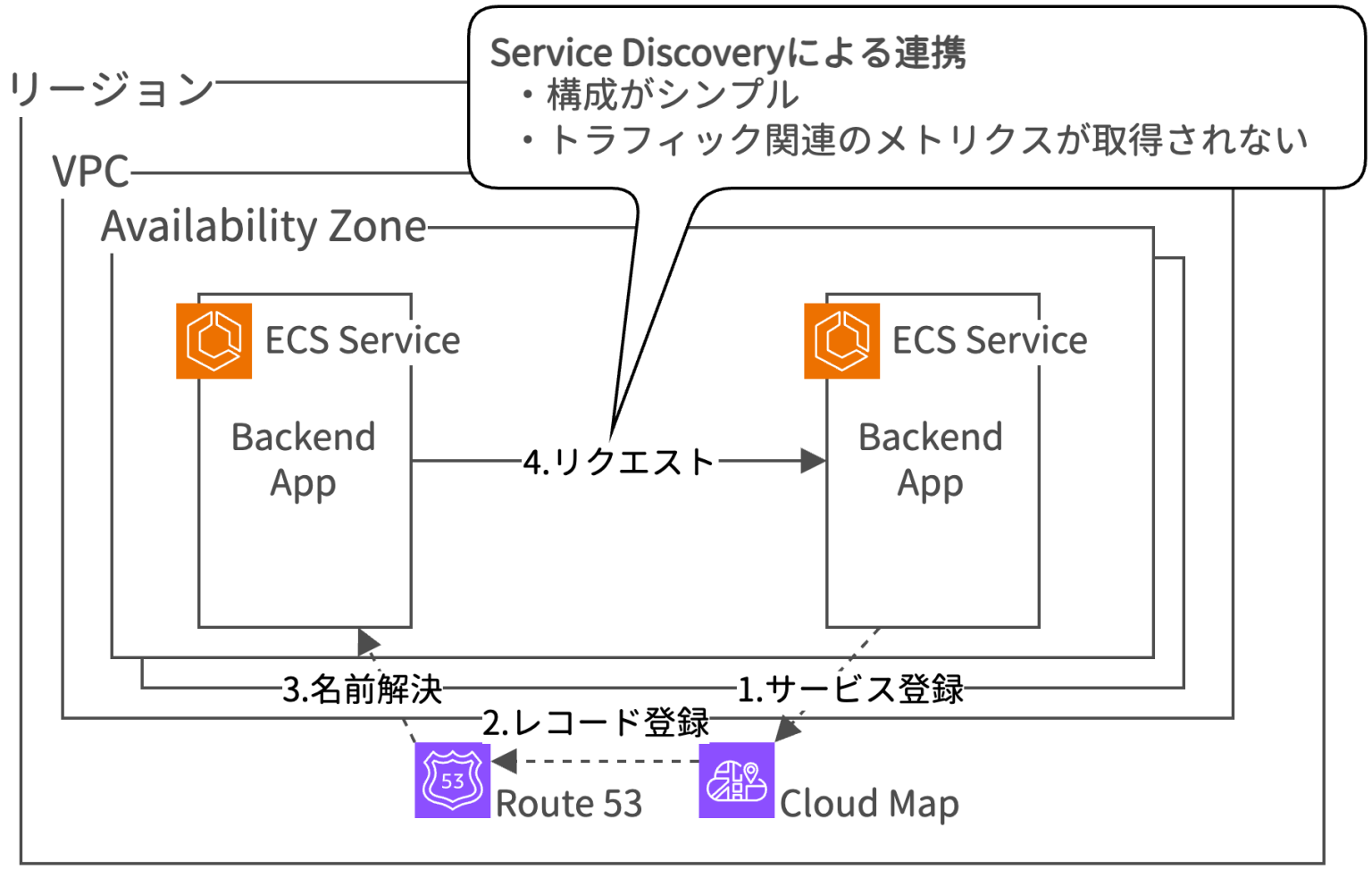
ECSとCloud Runで通信方法とアクセス制御方法が大きく異なる



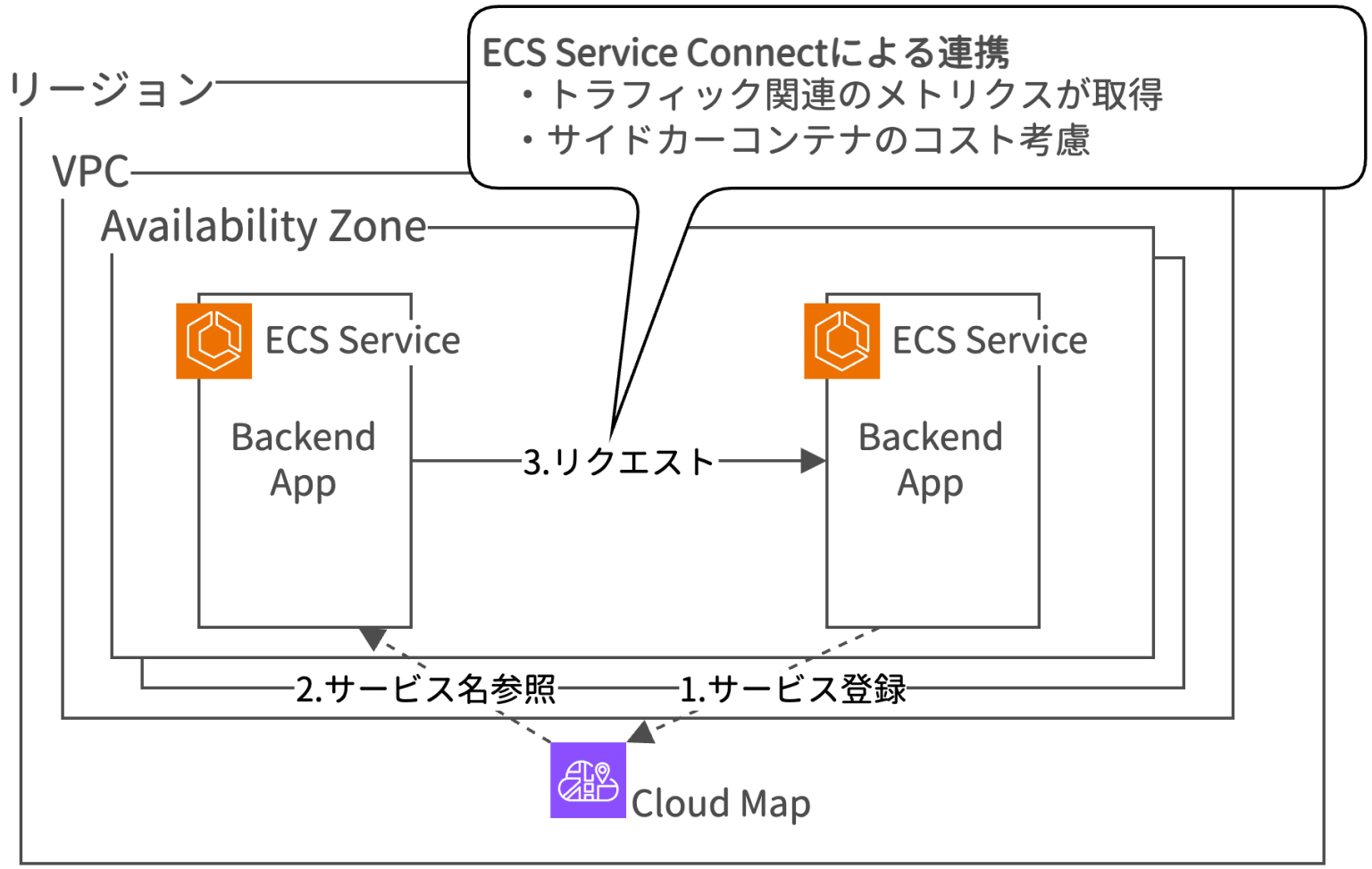
ECSとCloud Runで通信方法とアクセス制御方法が大きく異なる



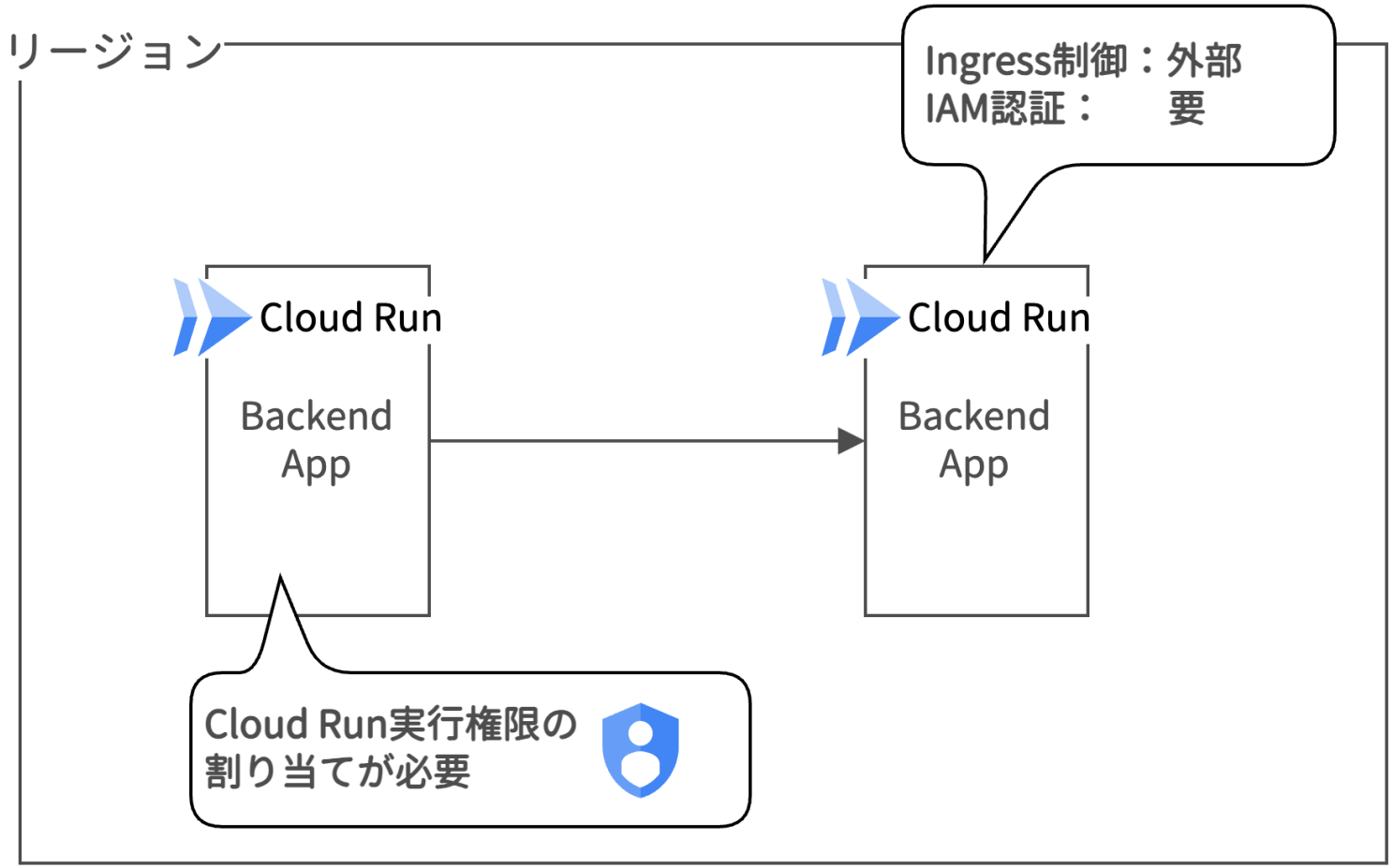
ECSとCloud Runで通信方法とアクセス制御方法が大きく異なる



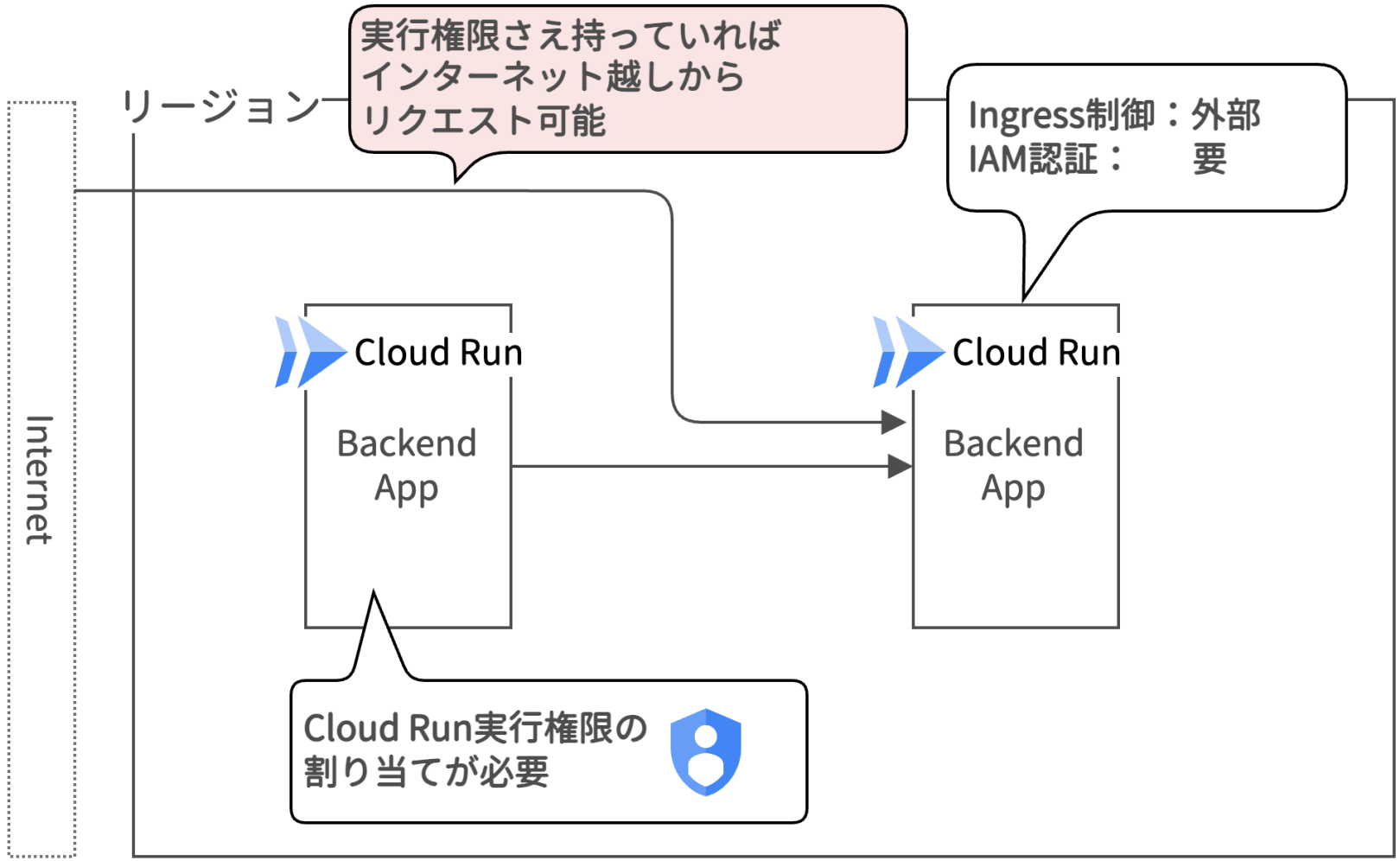
ECSとCloud Runで通信方法とアクセス制御方法が大きく異なる



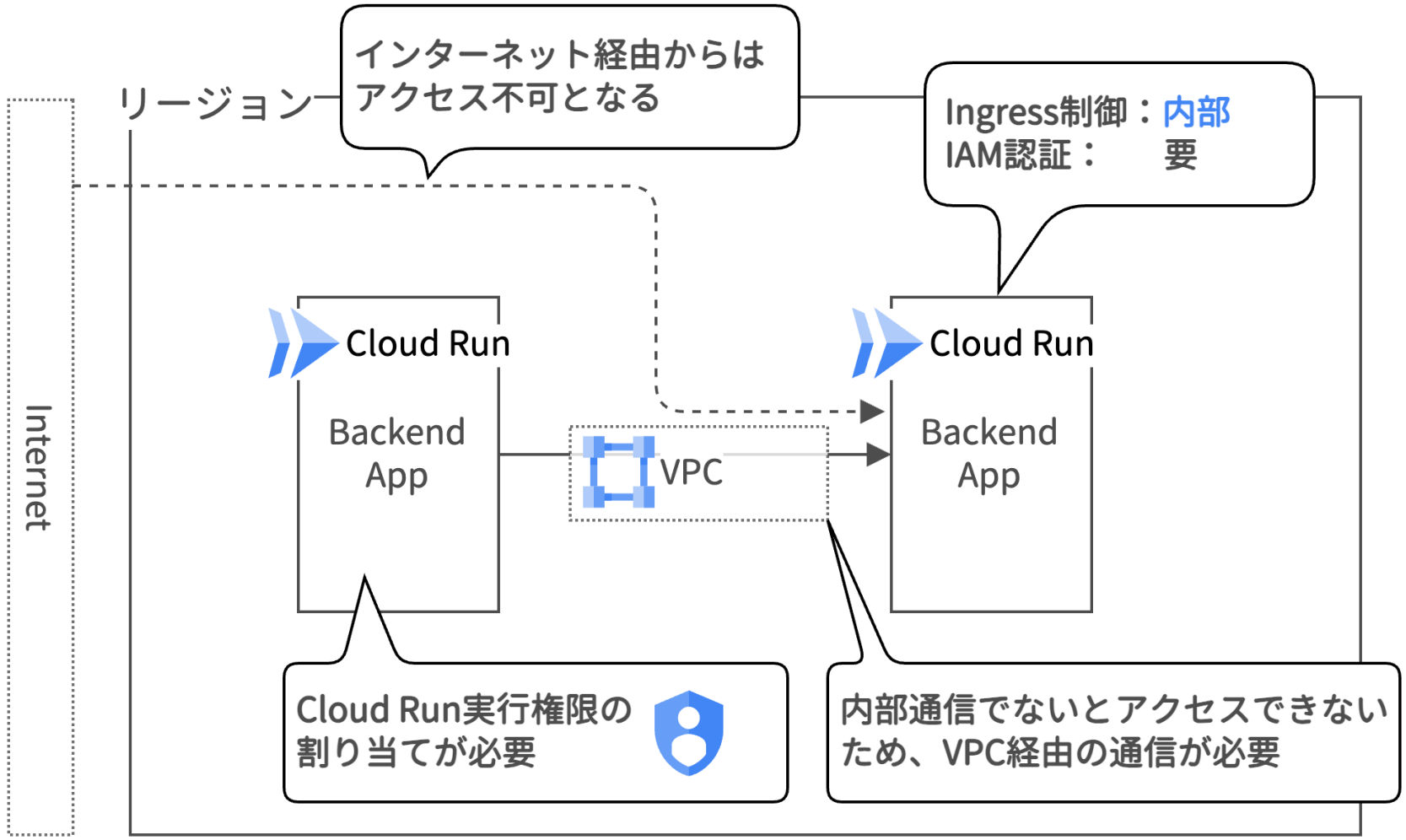
ECSとCloud Runで通信方法とアクセス制御方法が大きく異なる



ECSとCloud Runで通信方法とアクセス制御方法が大きく異なる



ECSとCloud Runで通信方法とアクセス制御方法が大きく異なる



まとめ

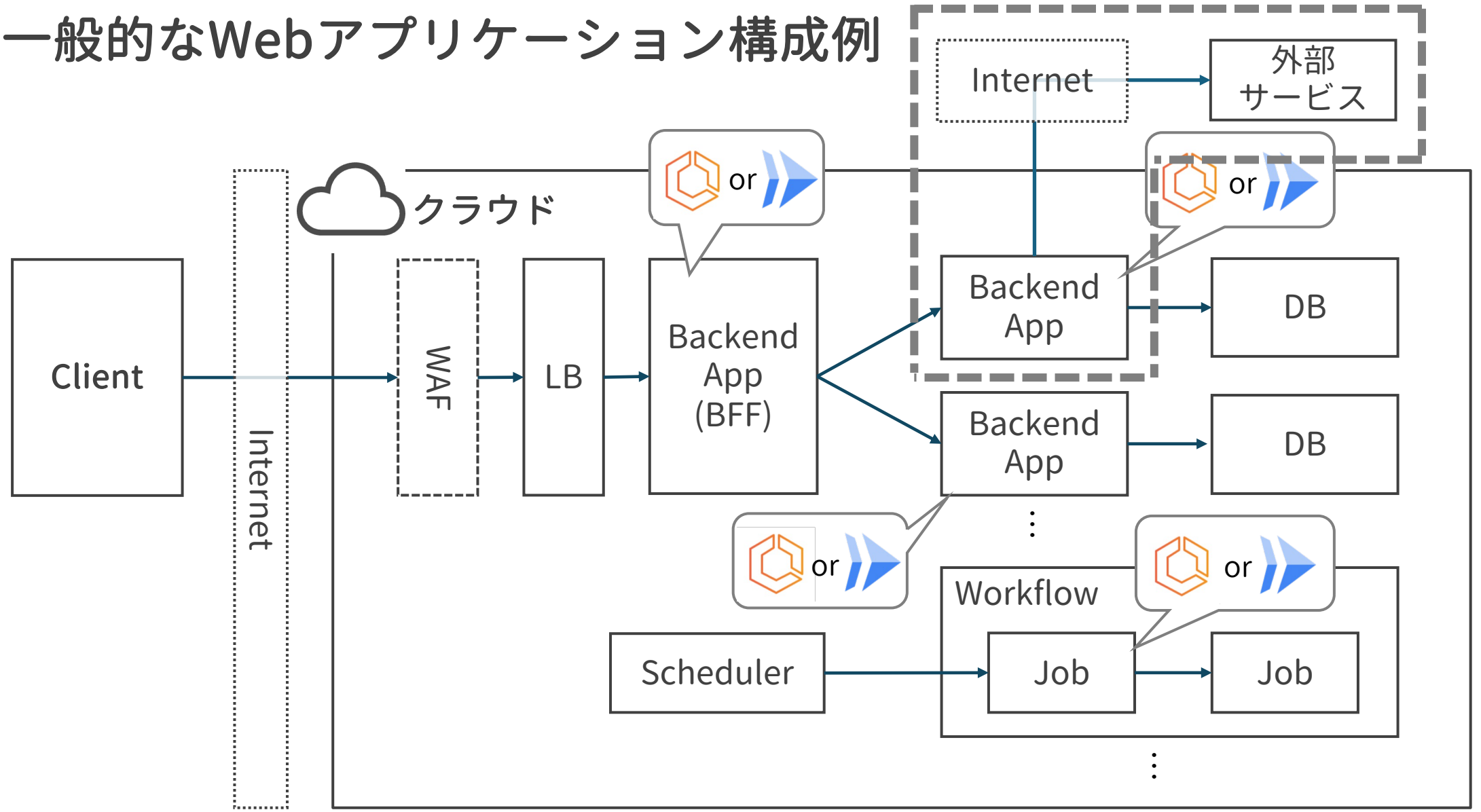
Amazon ECS

- 複数の接続パターンあり
  - ELB
  - ECS Service Discovery
  - ECS Service Connect
  - App Mesh(廃止予定)

Cloud Run

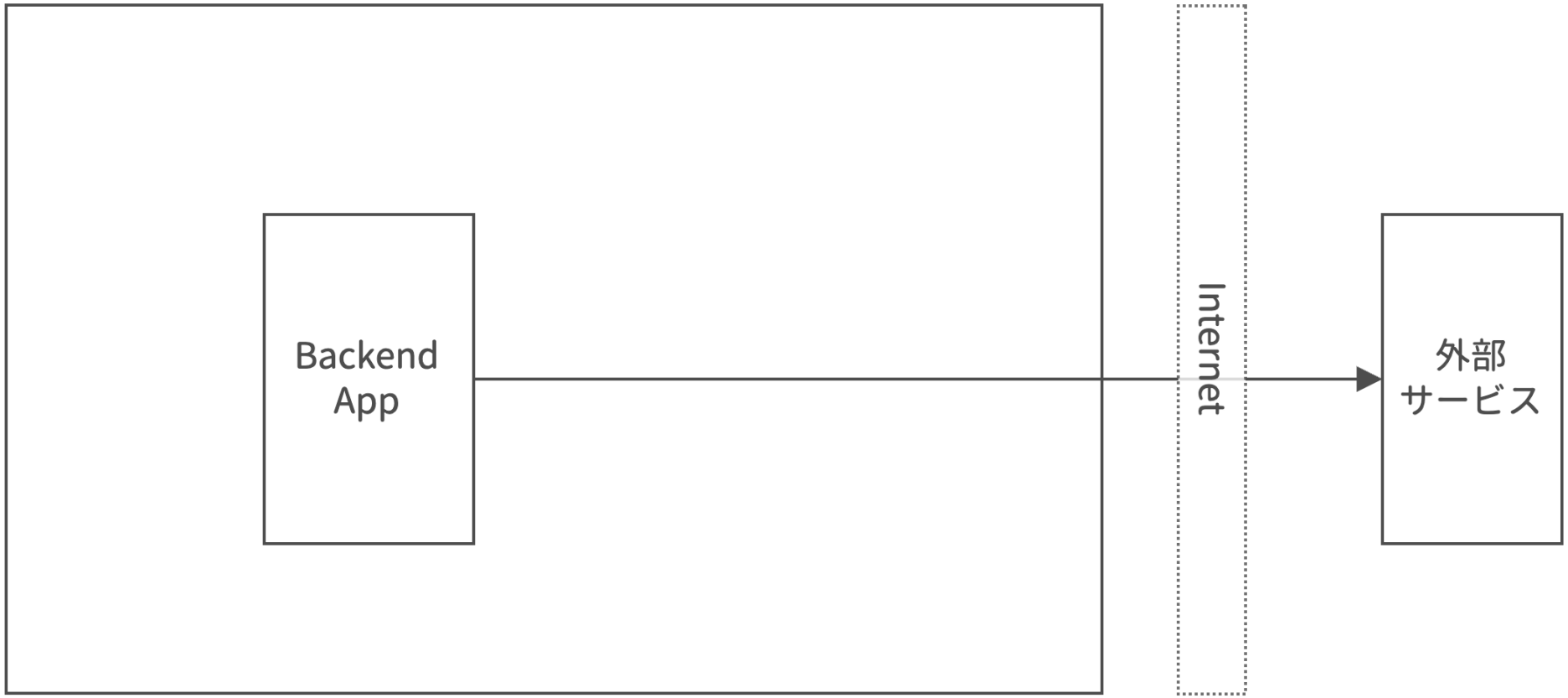
- Ingress制御+IAM認証の組み合わせによるアクセス許可
  - 内部通信からのみ許容する場合、VPCのバイパスが必要

一般的なWebアプリケーション構成例

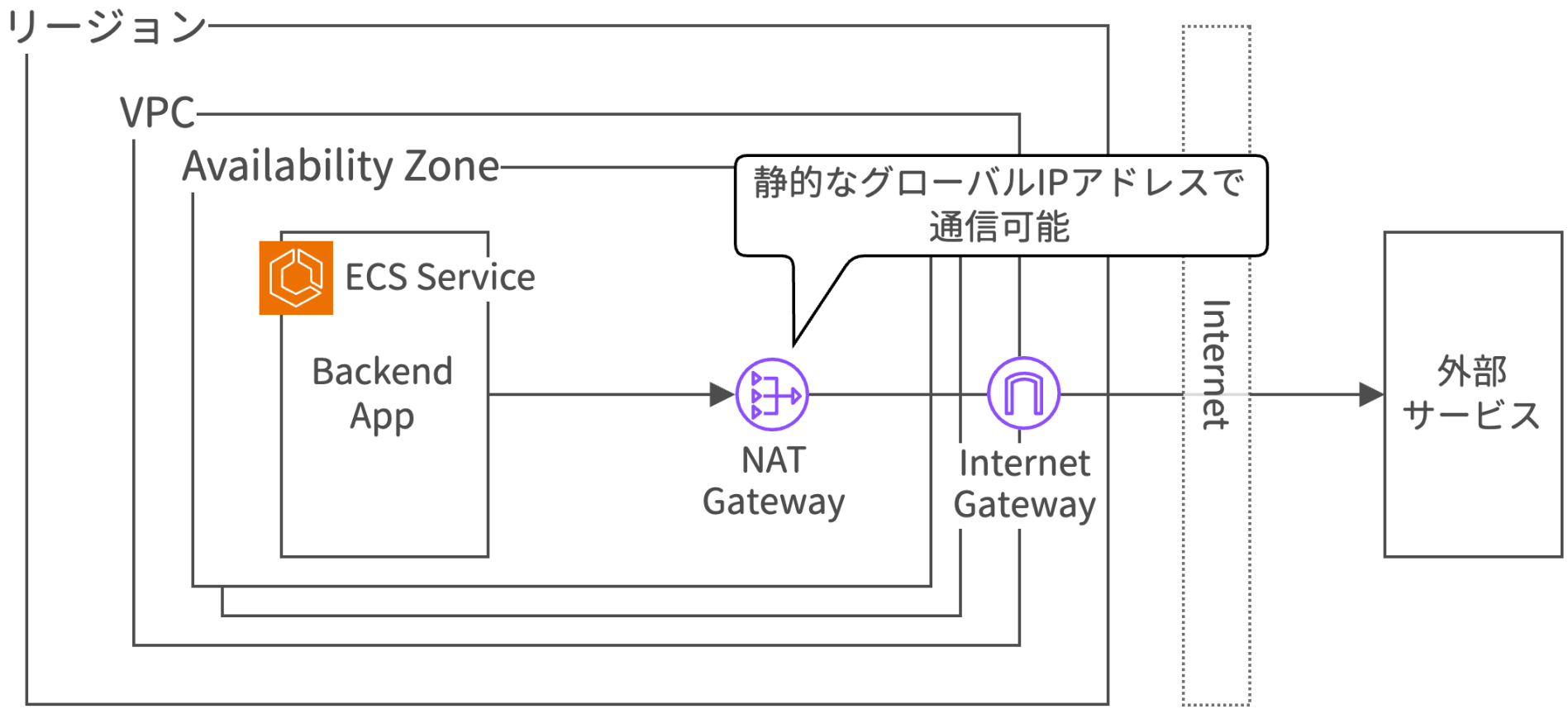


アーキテクチャデザインの違い – インターネットへの外部通信

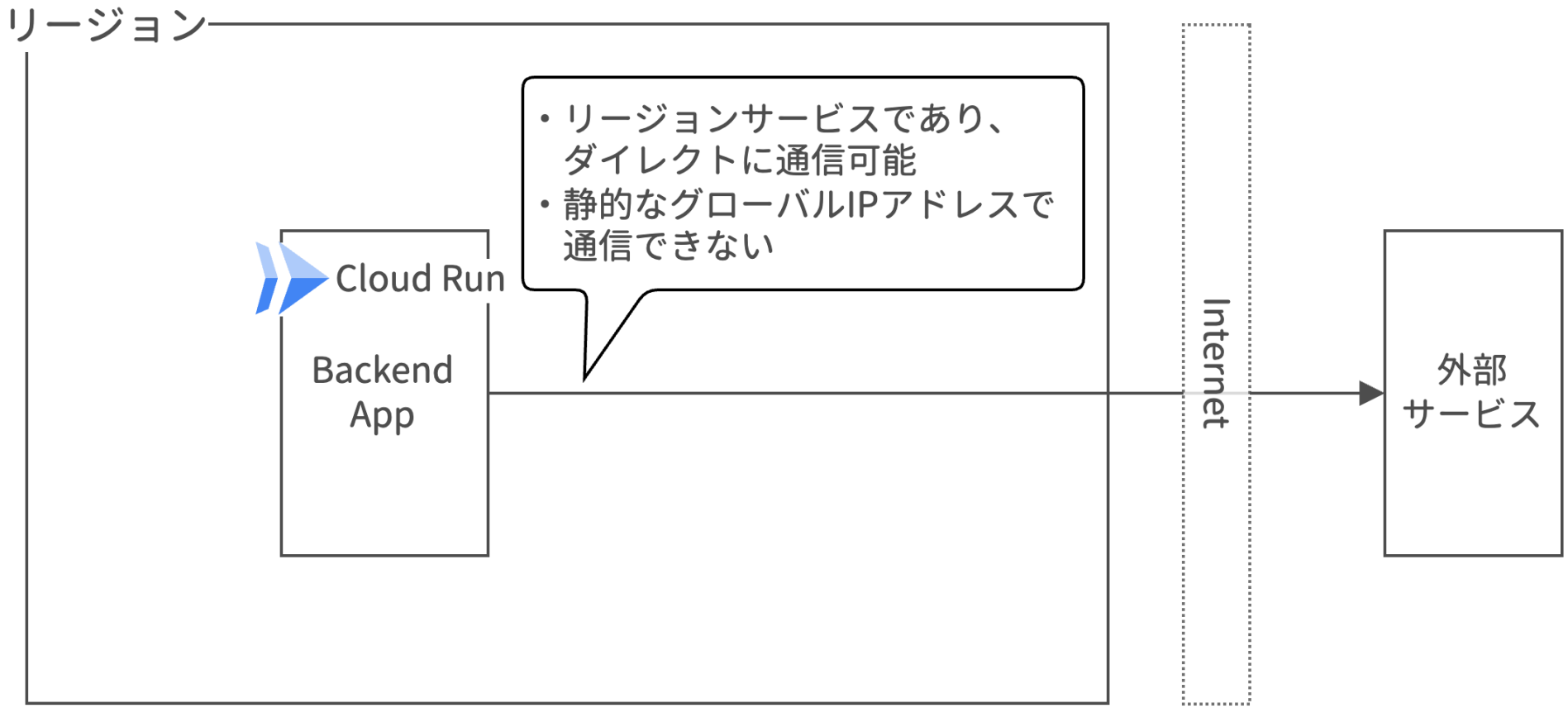
外部サービスへのアクセスはVPC要否により構成が異なる



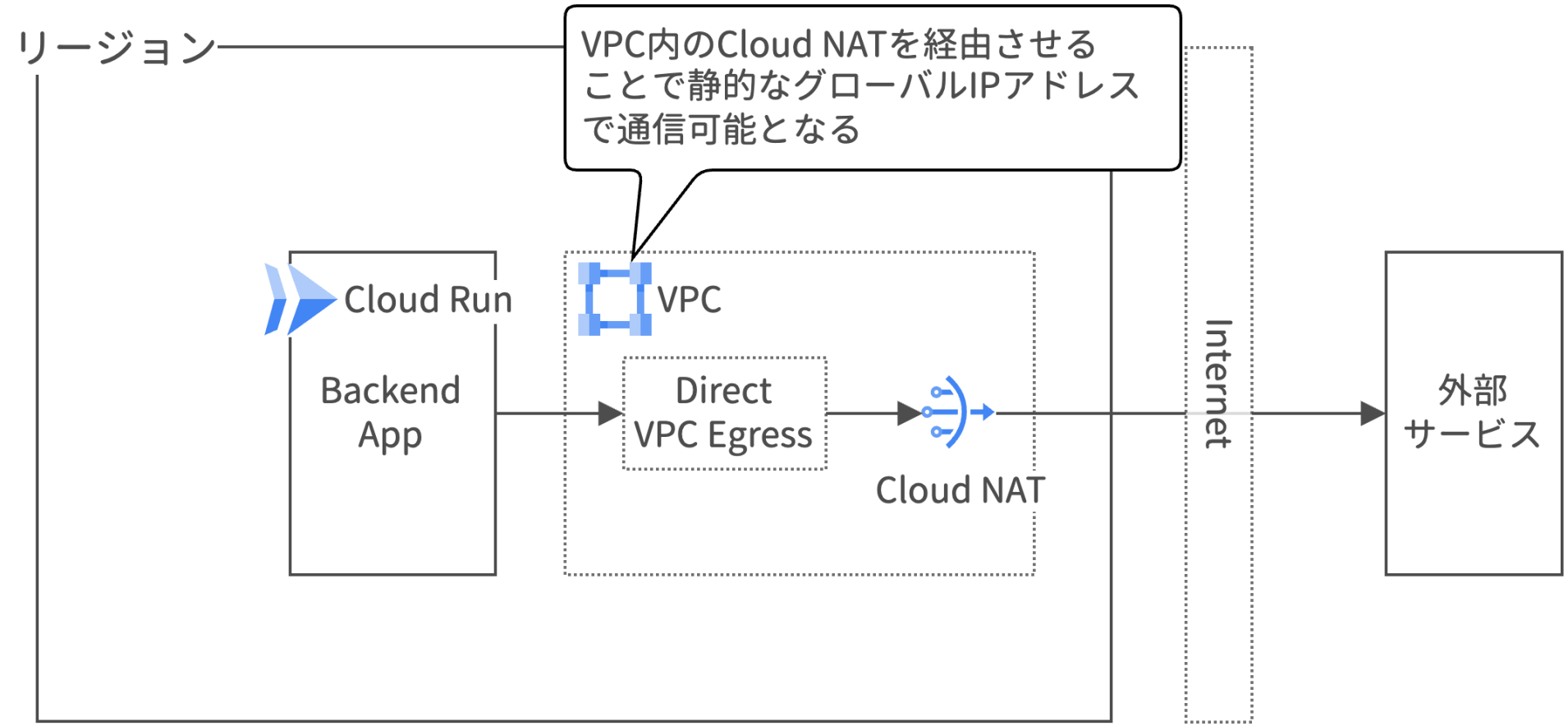
外部サービスへのアクセスはVPC要否により構成が異なる



外部サービスへのアクセスはVPC要否により構成が異なる



外部サービスへのアクセスはVPC要否により構成が異なる



まとめ

Amazon ECS

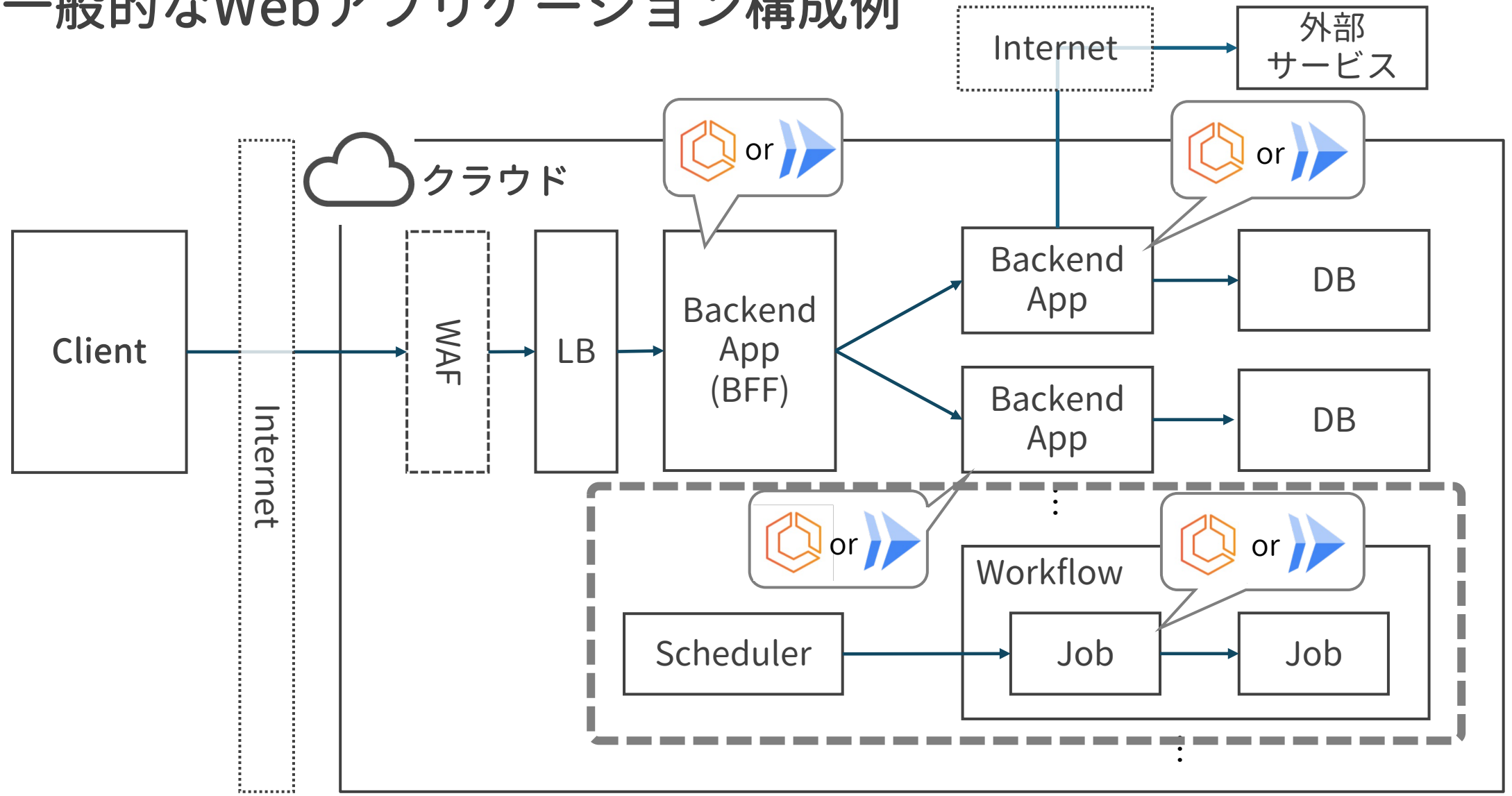
- VPC内から外部に通信する際はNAT Gatewayを経由

Cloud Run

- リージョンサービスなので、グローバルIPアドレスで直接外部サービスに通信可能
- 外部サービス側要件として、IPアドレス制限が必要な場合はVPC経由でCloud NATを通過

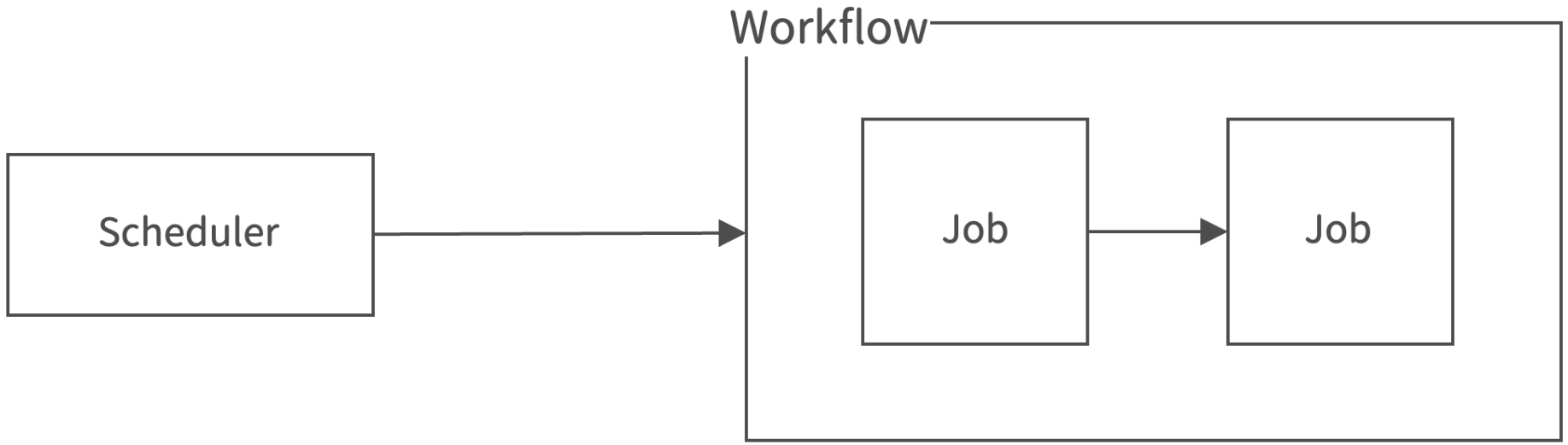
アーキテクチャデザインの違い – インターネットへの外部通信

一般的なWebアプリケーション構成例

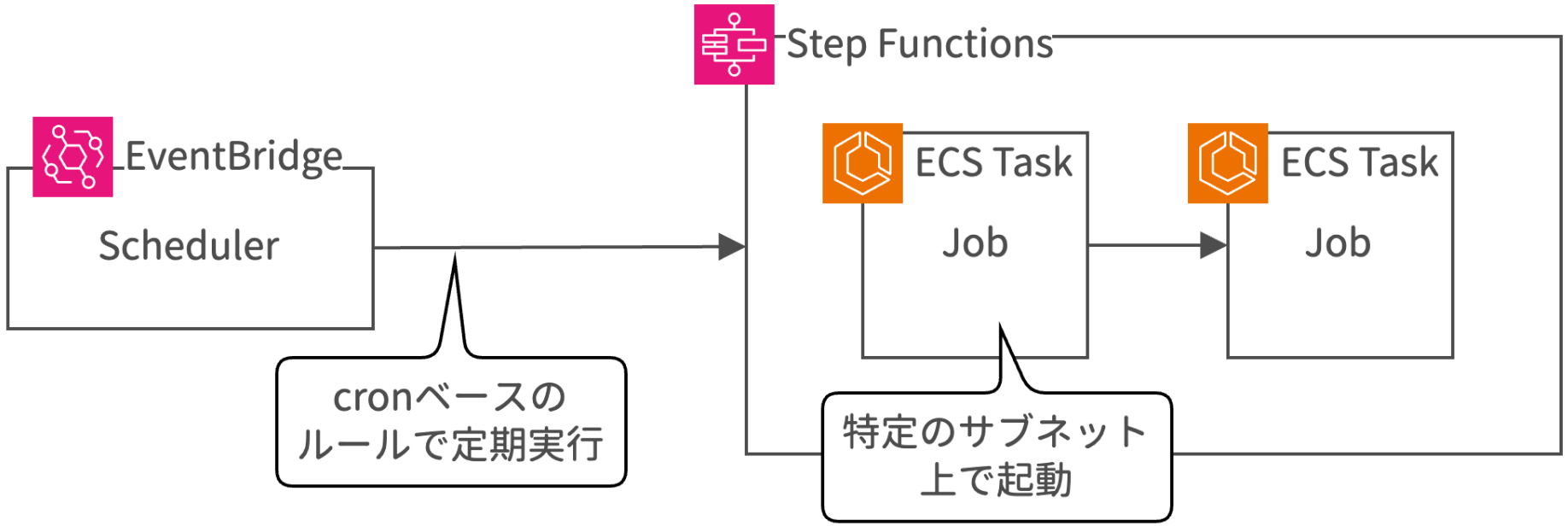


アーキテクチャデザインの違い – ジョブワークフロー

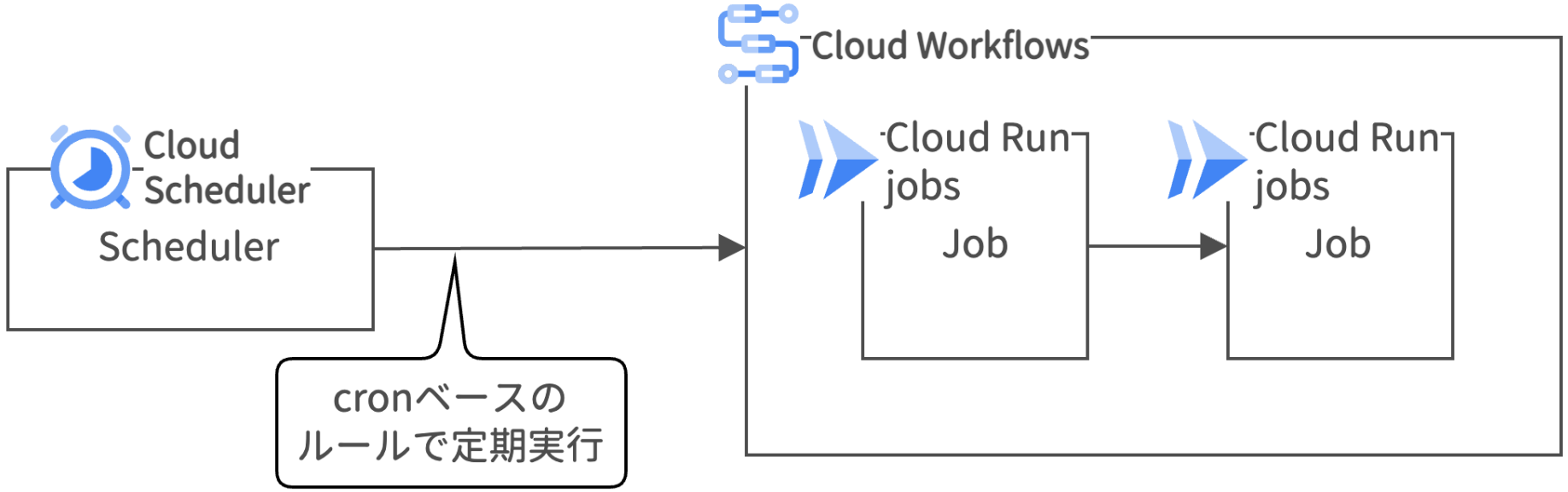
ジョブ実行は各クラウドのワークフローサービスを利用



ジョブ実行は各クラウドのワークフローサービスを利用



ジョブ実行は各クラウドのワークフローサービスを利用



まとめ

Amazon ECS

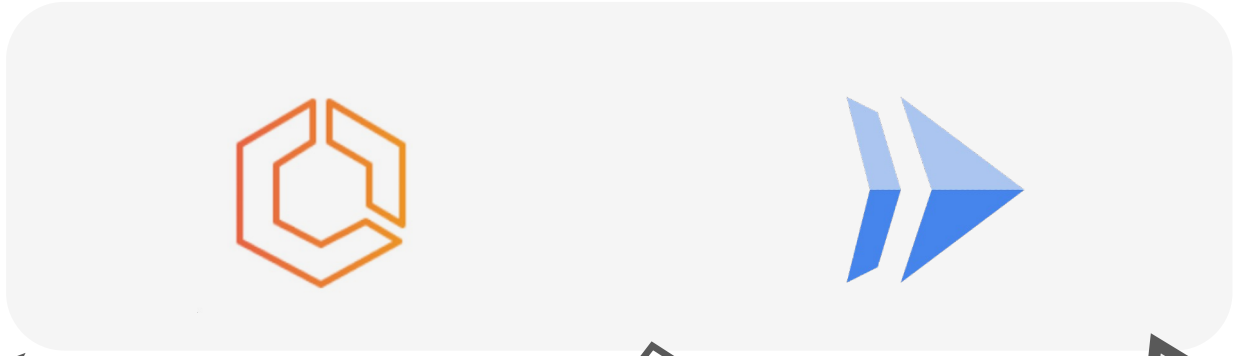
- Step Functionsにより、ECSタスクをコール
- 定期実行する場合はEventBridgeでStep Functionsを呼び出し

Cloud Run

- Cloud Workflowsにより、Cloud Run jobsをコール
- 定期実行する場合は、Cloud SchedulerでCloud Workflowsを呼び出し

アジェンダ

以下の3つの観点から相互理解を目指す



1. 基本的な特徴

2. アーキテクチャデザインの違い

3. 非機能デザインからの理解

非機能観点におけるアーキテクチャベストプラクティスが提供

非機能観点におけるアーキテクチャベストプラクティスが提供

Well-Architected Framework

AWS Well-Architected と 6 つの柱

フレームワークの概要

AWS Well-Architected Framework では、クラウド上でワークロードを設計および実行するための主要な概念、設計原則、アーキテクチャのベストプラクティスについて説明しています。いくつかの基本的な質問に答えると、アーキテクチャでクラウドのベストプラクティスがどの程度実践できているかを知り、改善のためのガイダンスを得ることができます。

HTML | ラボ

オペレーショナルエクセレンスの柱

オペレーショナルエクセレンスの柱では、システムの実行とモニタリング、およびプロセスと手順の継続的な改善に焦点を当てています。主なトピックには、変更の自動化、イベントへの対応、日常業務を管理するための標準化などが含まれます。

HTML | ラボ

セキュリティの柱

セキュリティの柱では、情報とシステムの保護に焦点を当てています。主なトピックには、データの機密性と完全性、ユーザー許可の管理、セキュリティイベントを検出するためのコントロールが含まれます。

HTML | ラボ

信頼性の柱

信頼性の柱は、期待通りの機能を実行するワークロードと、要求に応えられなかった場合に迅速に回復する方法に焦点を当てています。主なトピックには、分散システムの設計、復旧計画、および変化する要件への処理方法が含まれます。

HTML | ラボ

パフォーマンス効率の柱

パフォーマンス効率の柱は、IT およびコンピューティングリソースの構造化および合理化された割り当てに重点を置いています。主なトピックには、ワークロードの要件に応じて最適化されたリソースタイプやサイズの選択、パフォーマンスのモニタリング、ビジネスニーズの増大に応じて効率を維持することが含まれます。

HTML | ラボ

コスト最適化の柱

コスト最適化の柱は、不要なコストの回避に重点を置いています。主なトピックには、時間の経過による支出の把握と資金配分の管理、適切なリソースの種類と量の選択、および過剰な支出をせずにビジネスのニーズを満たすためのスケーリングが含まれます。

HTML | ラボ

持続可能性の柱

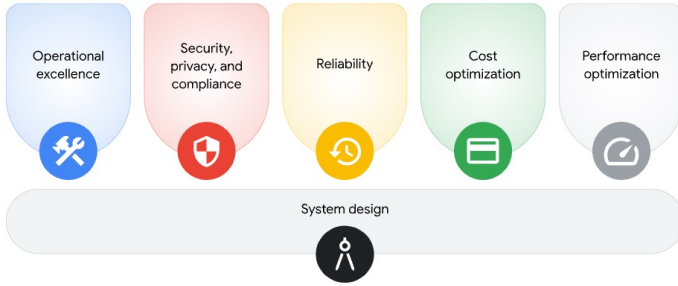
持続可能性の柱は、実行中のクラウドワークロードによる環境への影響を最小限に抑えることに重点を置いています。主なトピックには、持続可能性の責任共有モデル、影響についての把握、および必要なリソースを最小化してダウンストリームの影響を減らすための使用率の最大化が含まれます。

HTML | ラボ

Architecture Framework

アーキテクチャ フレームワークのカテゴリ

次の図に示すように、Google Cloud アーキテクチャ フレームワークは 5 つのカテゴリ（柱）に編成されています。



オペレーショナル エクセレンス

クラウド ワークロードを効率的にデプロイ、運用、モニタリング、管理します。

セキュリティ、プライバシー、コンプライアンス

クラウド内のデータとワークロードのセキュリティを最大化し、プライバシーを考慮した設計を行い、規制要件と標準に対応します。

信頼性

クラウドで復元性と高可用性を備えたワークロードを設計し、運用します。

費用の最適化

Google Cloud への投資のビジネス上の価値を最大化します。

パフォーマンスの最適化

パフォーマンスが最適化されるようにクラウド リソースを設計、調整します。

共通軸による設計観点をいくつかピックアップ

Well-Architected Framework

AWS Well-Architected と 6 つの柱

フレームワークの概要

AWS Well-Architected Framework では、クラウド上でワークロードを設計および実行するための主要な概念、設計原則、アーキテクチャのベストプラクティスについて説明しています。いくつかの基本的な質問に答えると、アーキテクチャでクラウドのベストプラクティスがどの程度実践できているかを知り、改善のためのガイダンスを得ることができます。

HTML | ラボ

オペレーショナルエクセレンスの柱

オペレーショナルエクセレンスの柱では、システムの実行とモニタリング、およびプロセスと手順の継続的な改善に焦点を当てています。主なトピックには、変更の自動化、イベントへの対応、日常業務を管理するための標準化などが含まれます。

HTML | ラボ

セキュリティの柱

セキュリティの柱では、情報とシステムの保護に焦点を当てています。主なトピックには、データの機密性と完全性、ユーザー許可の管理、セキュリティイベントを検出するためのコントロールが含まれます。

HTML | ラボ

信頼性の柱

信頼性の柱は、期待通りの機能を実行するワークロードと、要求に応えられなかった場合に迅速に回復する方法に焦点を当てています。主なトピックには、分散システムの設計、復旧計画、および変化する要件への処理方法が含まれます。

HTML | ラボ

パフォーマンス効率の柱

パフォーマンス効率の柱は、IT およびコンピューティングリソースの構造化および合理化された割り当てに重点を置いています。主なトピックには、ワークロードの要件に応じて最適化されたリソースタイプやサイズの選択、パフォーマンスのモニタリング、ビジネスニーズの増大に応じて効率を維持することが含まれます。

HTML | ラボ

コスト最適化の柱

コスト最適化の柱は、不要なコストの回避に重点を置いています。主なトピックには、時間の経過による支出の把握と資金配分の管理、適切なリソースの種類と量の選択、および過剰な支出をせずにビジネスのニーズを満たすためのスケーリングが含まれます。

HTML | ラボ

持続可能性の柱

持続可能性の柱は、実行中のクラウドワークロードによる環境への影響を最小限に抑えることに重点を置いています。主なトピックには、持続可能性の責任共有モデル、影響についての把握、および必要なリソースを最小化してダウンストリームの影響を減らすための使用率の最大化が含まれます。

HTML | ラボ

Architecture Framework

アーキテクチャ フレームワークのカテゴリ

次の図に示すように、Google Cloud アーキテクチャ フレームワークは 5 つのカテゴリ（柱）に編成されています。

オペレーショナル エクセレンス

クラウド ワークロードを効率的にデプロイ、運用、モニタリング、管理します。

セキュリティ、プライバシー、コンプライアンス

クラウド内のデータとワークロードのセキュリティを最大化し、プライバシーを考慮した設計を行い、規制要件と標準に対応します。

信頼性

クラウドで復元性と高可用性を備えたワークロードを設計し、運用します。

費用の最適化

Google Cloud への投資のビジネス上の価値を最大化します。

パフォーマンスの最適化

パフォーマンスが最適化されるようにクラウド リソースを設計、調整します。

共通軸による設計観点をいくつかピックアップ

運用デザイン	CI/CD設計、ログ運用
セキュリティ	クレデンシャル管理
パフォーマンス	スケールアウト設計
信頼性	シグナルハンドリング
コスト最適化	課金モデル

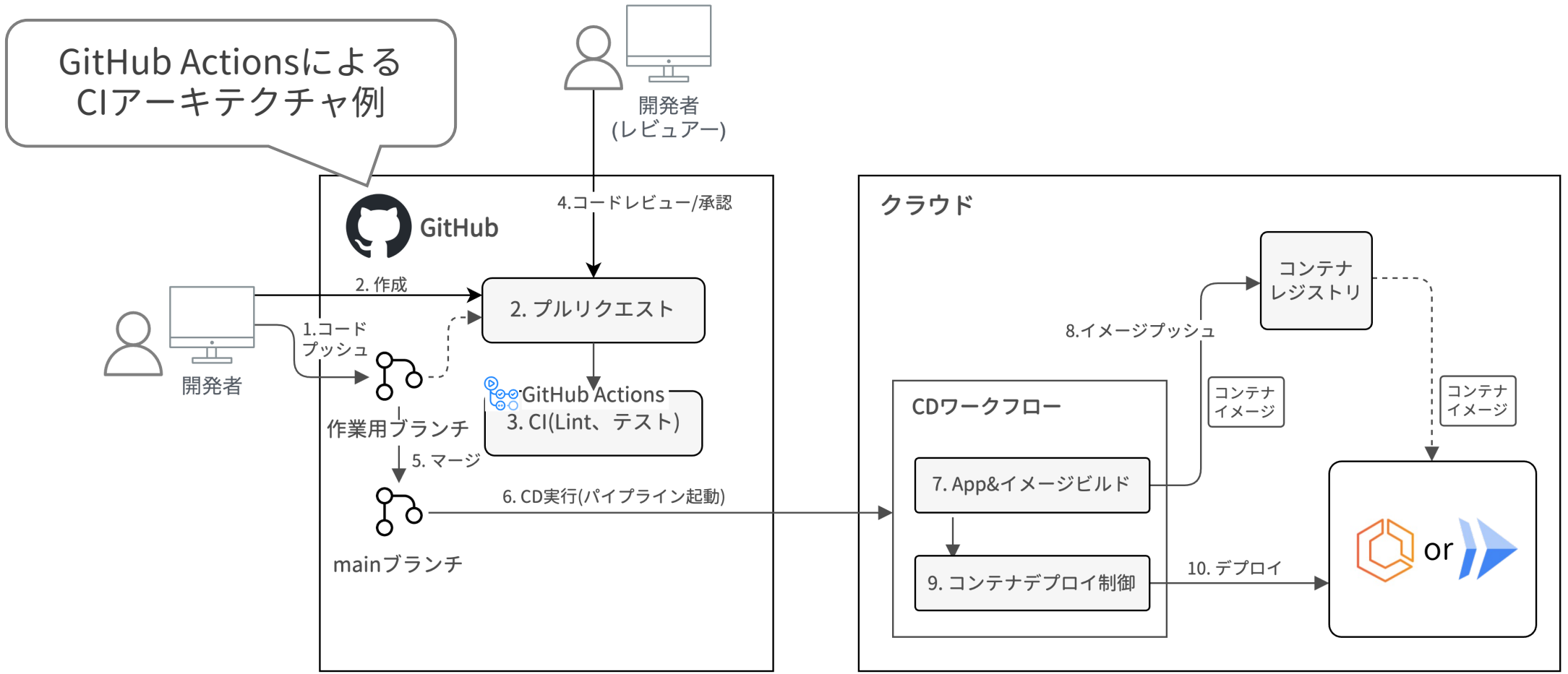
運用デザイン – CI/CD設計

非機能デザインからの理解 – CI/CD設計

大きな違いとしては、AWSは専用のワークフローサービスがある点や
Google Cloudでは デプロイサービスに承認プロセスがある

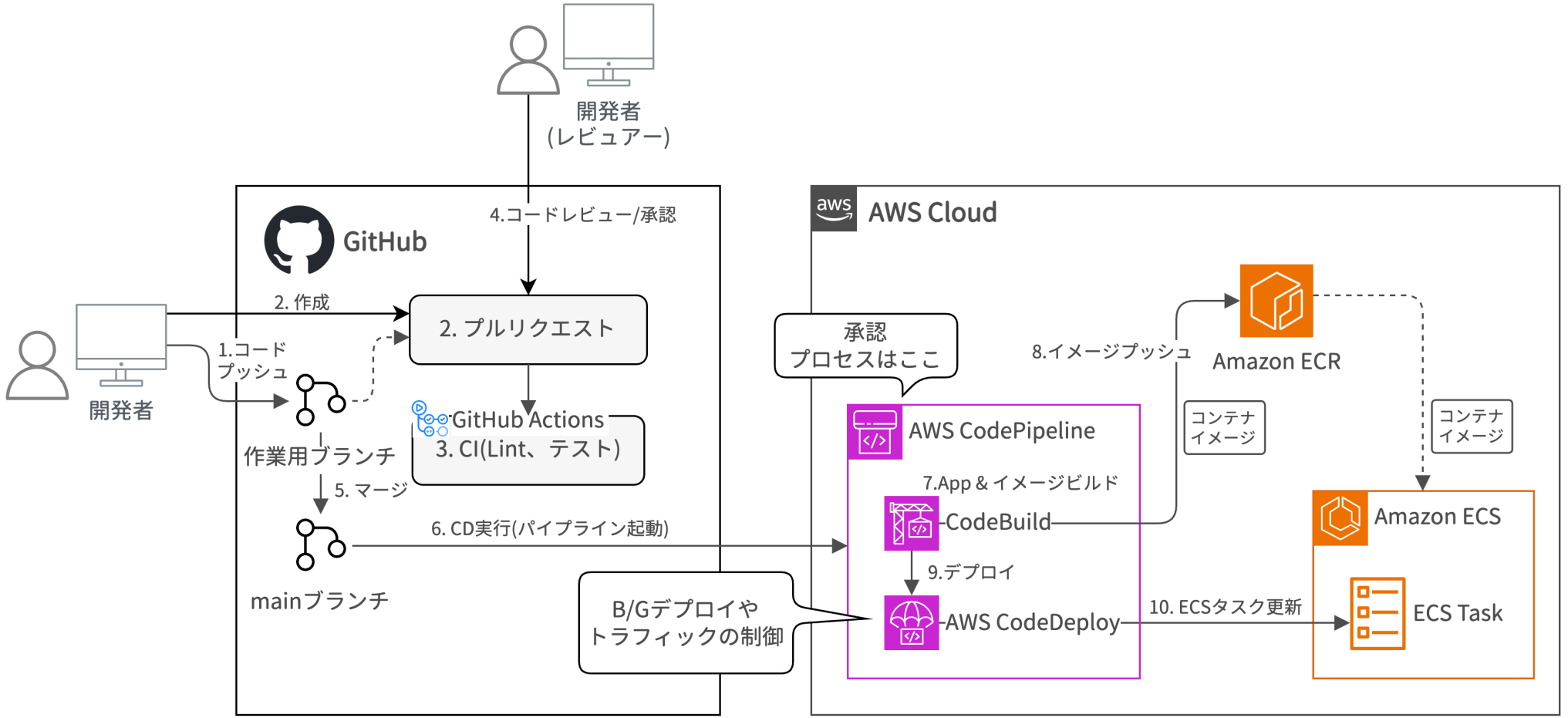
非機能デザインからの理解 – CI/CD設計

大きな違いとしては、AWSは専用のワークフローサービスがある点や Google Cloudでは デプロイサービスに承認プロセスがある



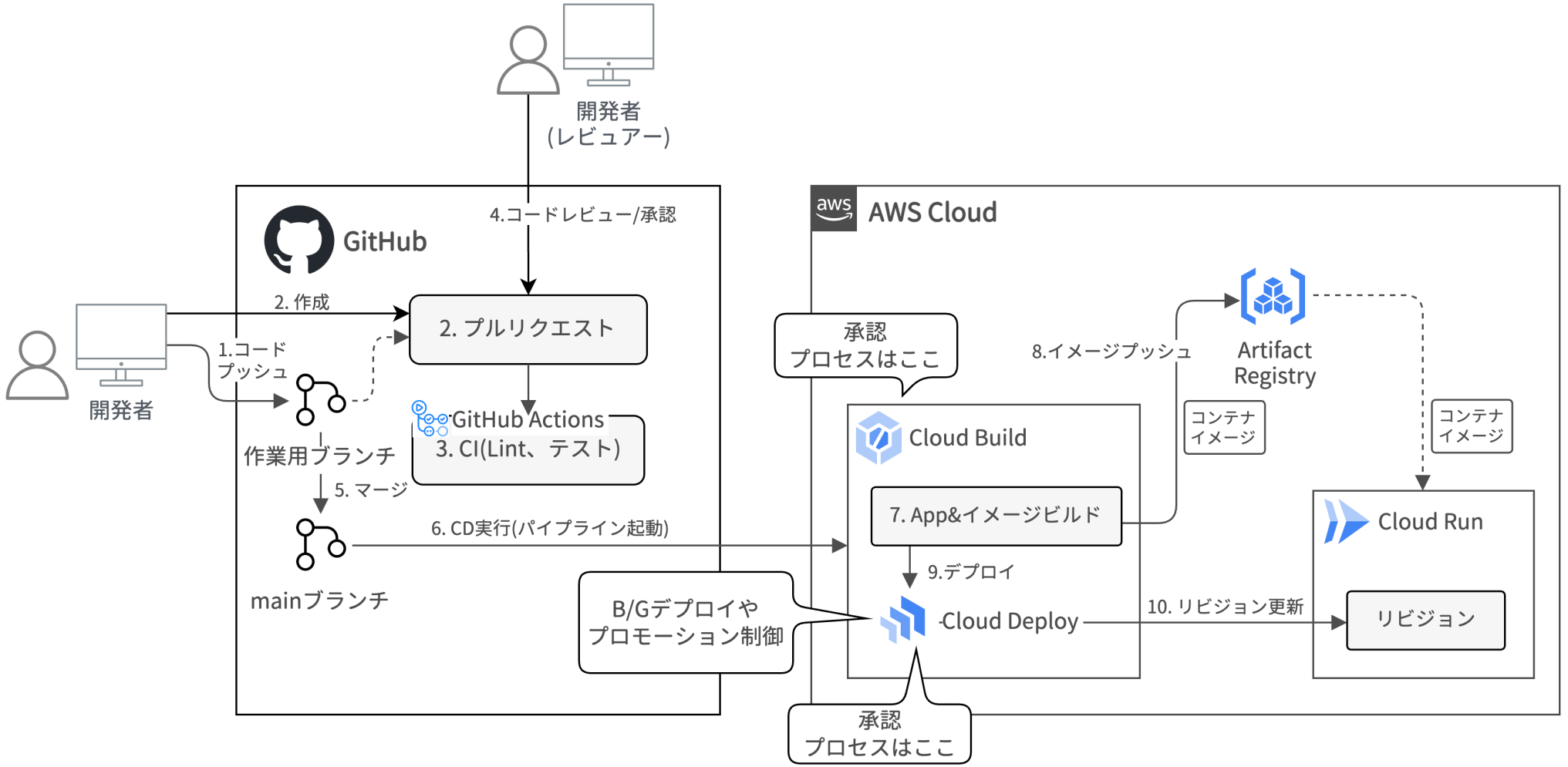
非機能デザインからの理解 – CI/CD設計

大きな違いとしては、AWSは専用のワークフローサービスがある点や Google Cloudでは デプロイサービスに承認プロセスがある



非機能デザインからの理解 – CI/CD設計

大きな違いとしては、AWSは専用のワークフローサービスがある点や Google Cloudでは デプロイサービスに承認プロセスがある



まとめ

Amazon ECS

- CI/CDワークフロー：CodePipeline or CodeBuild
- ビルド：CodeBuild
- デプロイ：CodeBuild or CodeDeploy
- 承認プロセス： CodePipeline

Cloud Run

- CI/CDワークフロー：Cloud Build
- ビルド：Cloud Build
- デプロイ：Cloud Build or Cloud Deploy
- 承認プロセス： CodeBuild or CodeDeploy

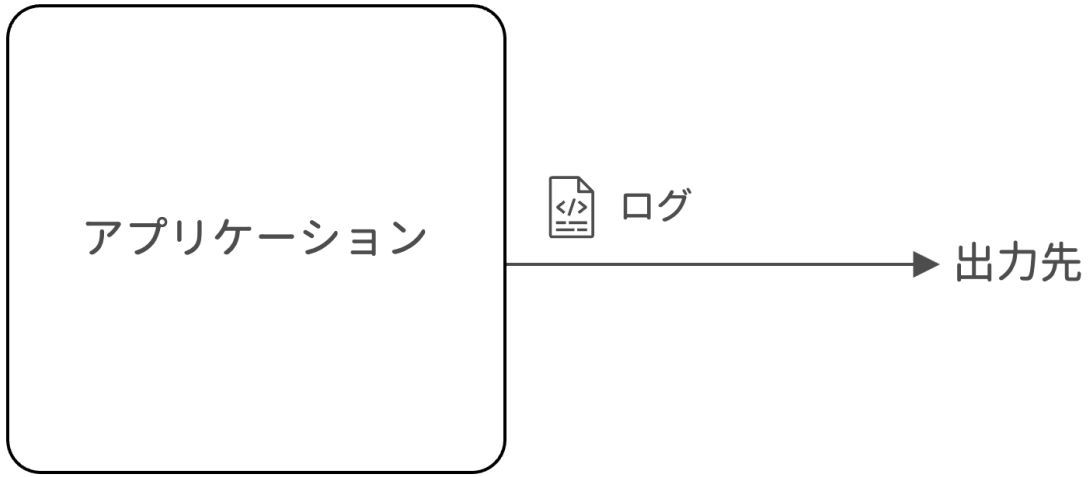
運用デザイン –ログ運用

非機能デザインからの理解 – ログ運用

ECSはFluentbitをサイドカー構成とすることで、
出力先を柔軟に変更できる一方、Cloud RunはCloud Logging出力がマスト

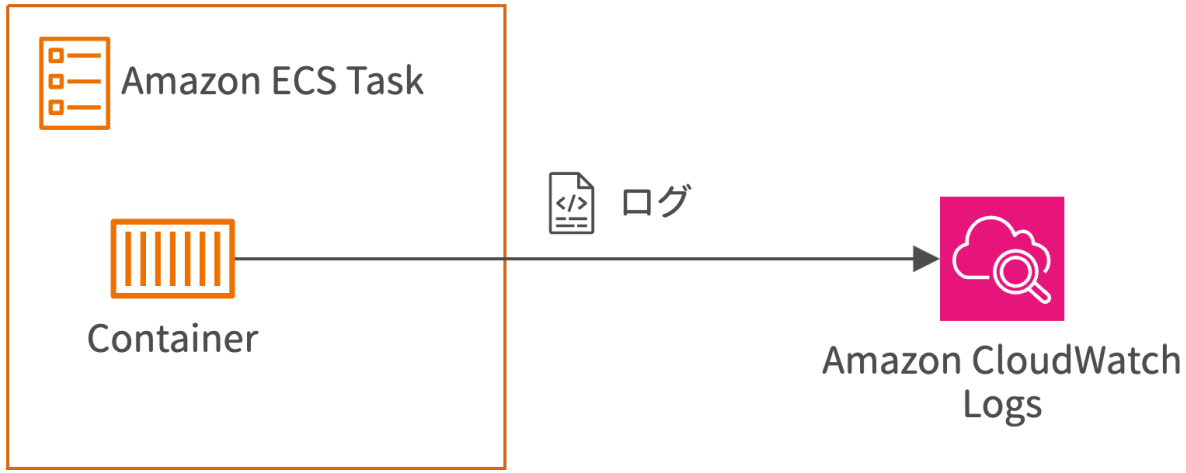
非機能デザインからの理解 – ログ運用

ECSはFluentbitをサイドカー構成とすることで、
出力先を柔軟に変更できる一方、Cloud RunはCloud Logging出力がマスト



非機能デザインからの理解 – ログ運用

ECSはFluentbitをサイドカー構成とすることで、
出力先を柔軟に変更できる一方、Cloud RunはCloud Logging出力がマスト



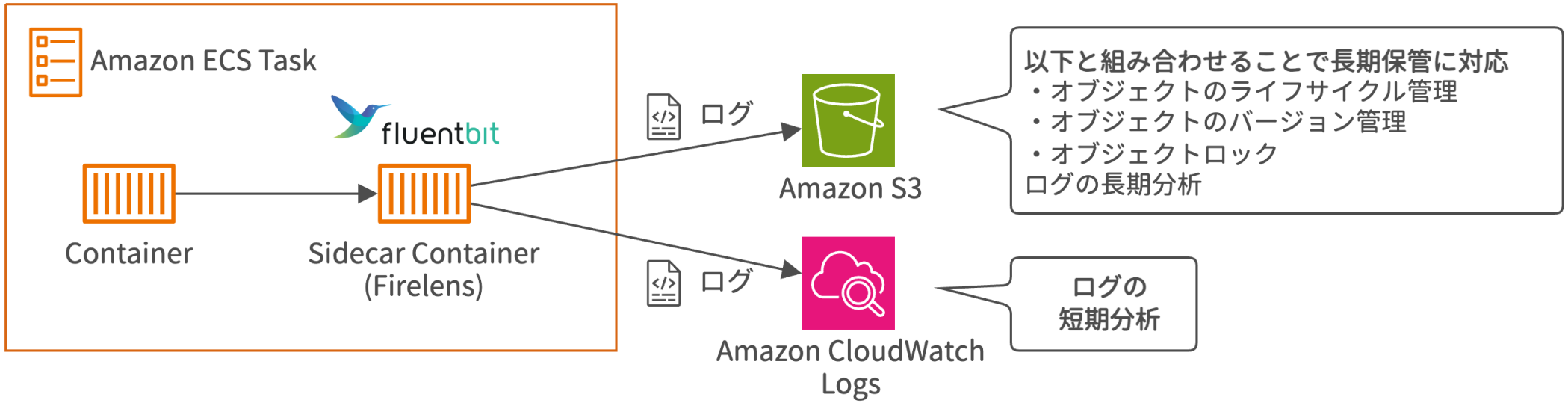
非機能デザインからの理解 – ログ運用

ECSはFluentbitをサイドカー構成とすることで、
出力先を柔軟に変更できる一方、Cloud RunはCloud Logging出力がマスト

ECSのログ長期保管と分析のために、
出力先を変えたい場合はどうする？

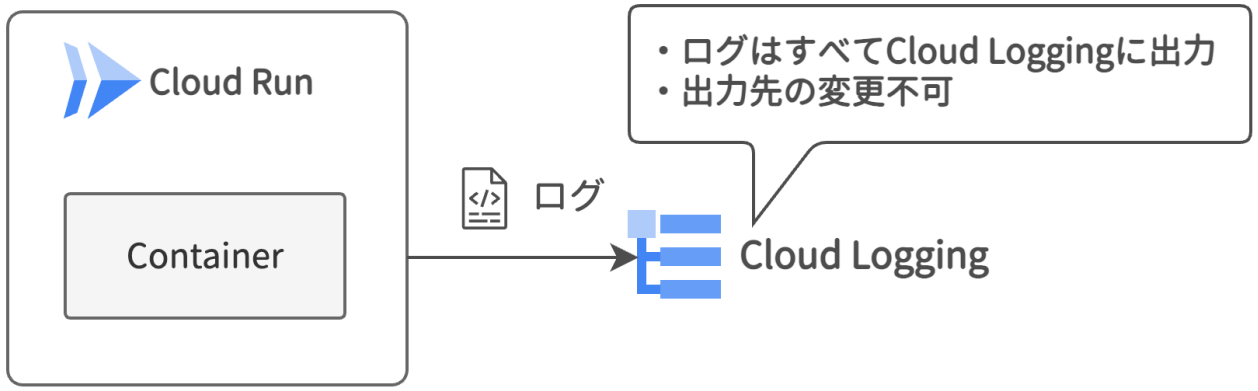
非機能デザインからの理解 – ログ運用

ECSはFluentbitをサイドカー構成とすることで、
出力先を柔軟に変更できる一方、Cloud RunはCloud Logging出力がマスト



非機能デザインからの理解 – ログ運用

ECSはFluentbitをサイドカー構成とすることで、出力先を柔軟に変更できる一方、Cloud RunはCloud Logging出力がマスト

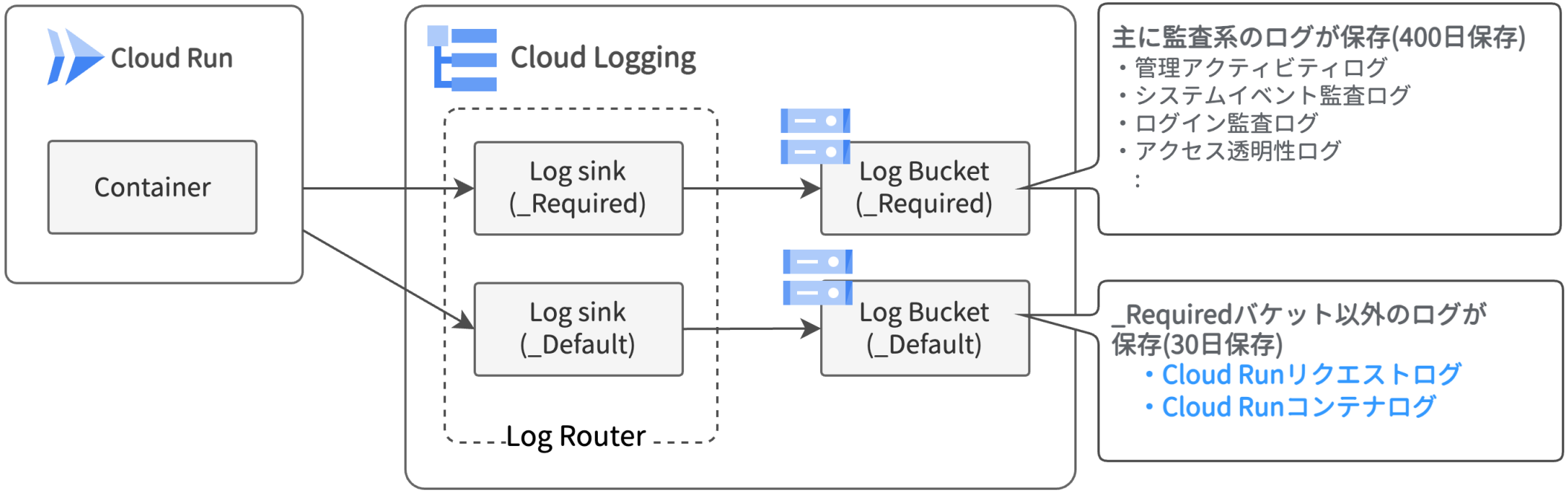


非機能デザインからの理解 – ログ運用

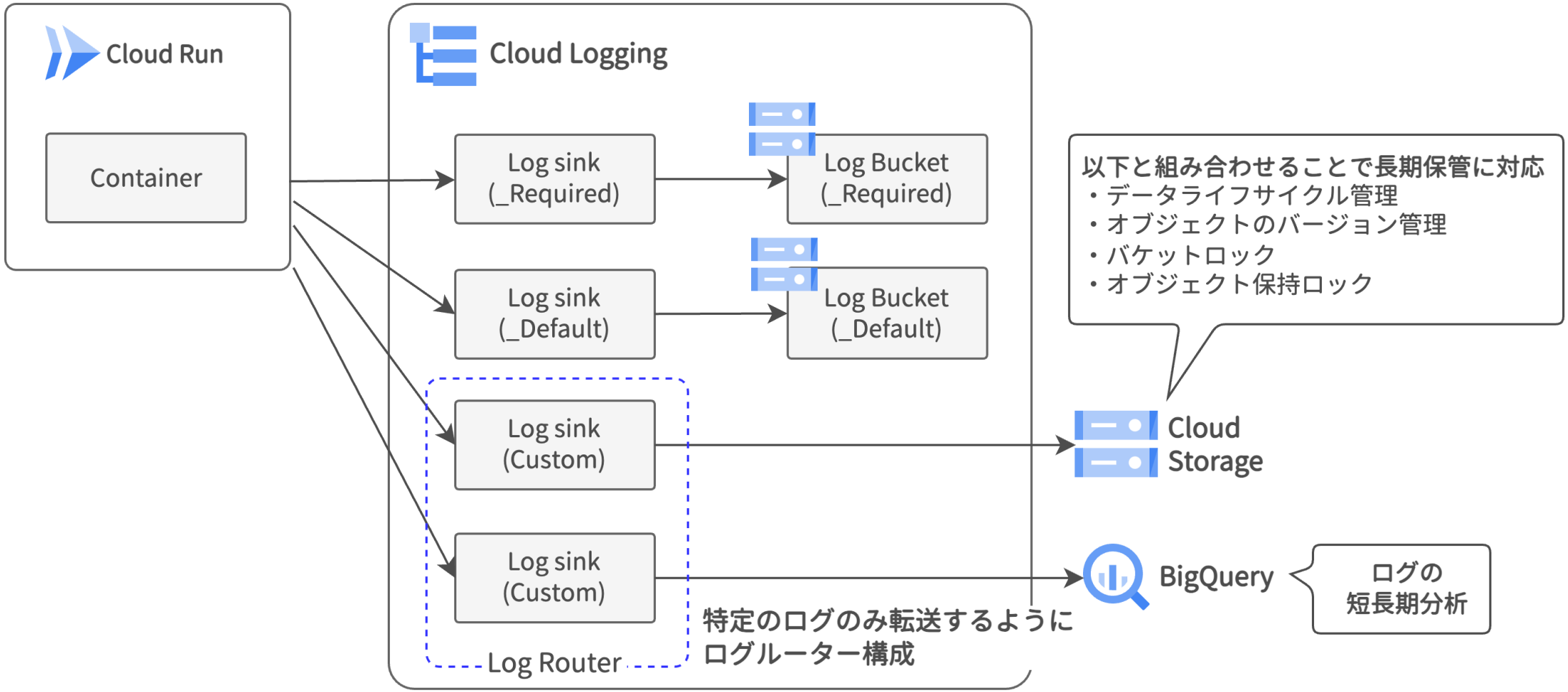
ECSはFluentbitをサイドカー構成とすることで、
出力先を柔軟に変更できる一方、Cloud RunはCloud Logging出力がマスト

Cloud Runのログ長期保管と分析のために、出力先を変えたい場合はどうする？

Cloud Runでログを振り分ける場合、Log sinkを作成して別途転送するように構成する



Cloud Runでログを振り分ける場合、Log sinkを作成して別途転送するように構成する



非機能デザインからの理解 – ログ運用

まとめ

Amazon ECS

- ログの基本出力はCloudWatch Logs
- ログ出力を振り分ける場合、fluentbitをサイドカー構成

Cloud Run

- ログの基本出力はCloud Logging
- ログ出力を振り分ける場合、Log Routerにより別のLog sinkを作成

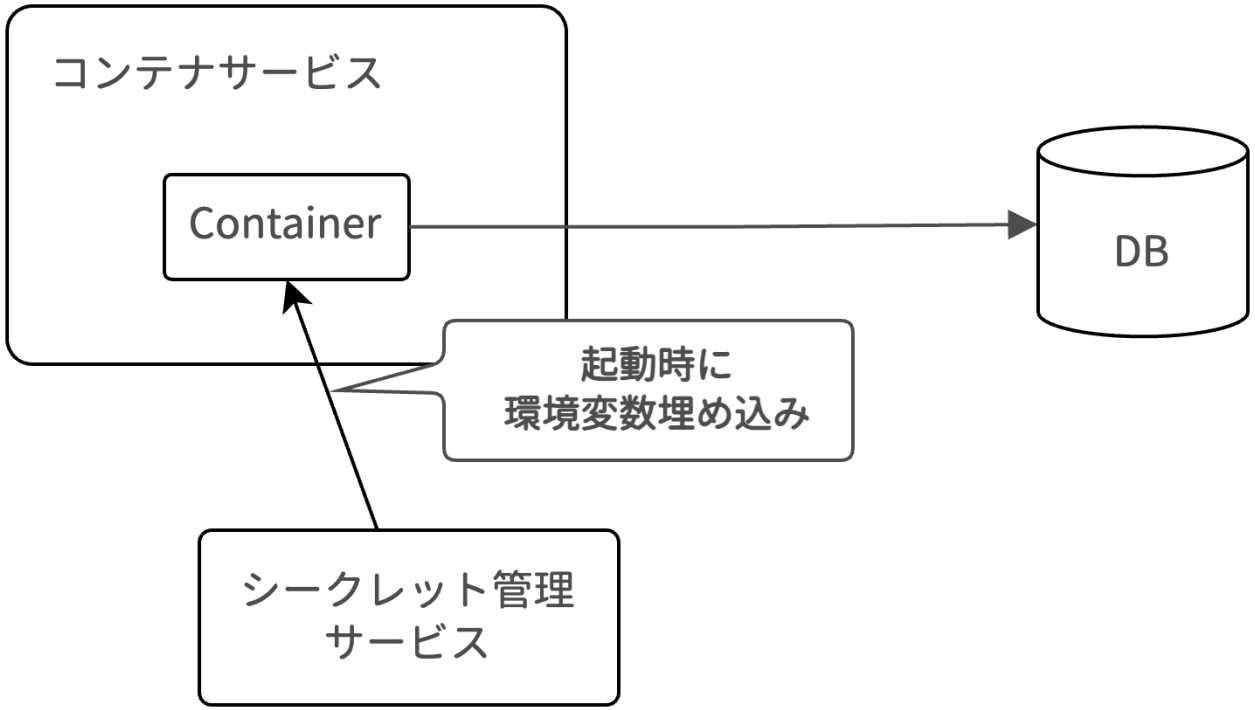
セキュリティ – クレデンシャル管理

非機能デザインからの理解 – セキュリティ/クレデンシャル管理

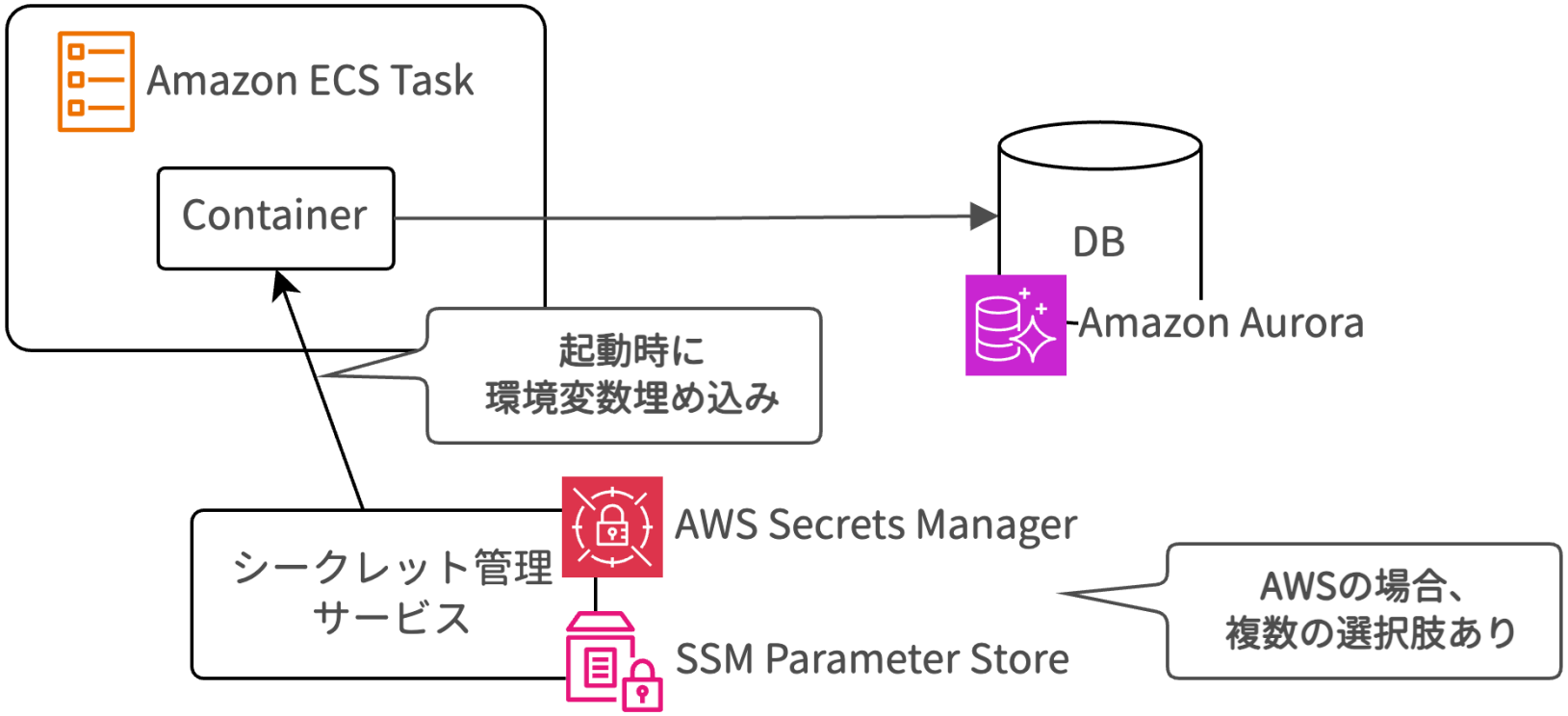
クレデンシャル管理サービスを活用し、
アプリケーション起動時に環境変数として埋め込むことで安全に利用

非機能デザインからの理解 – セキュリティ/クレデンシャル管理

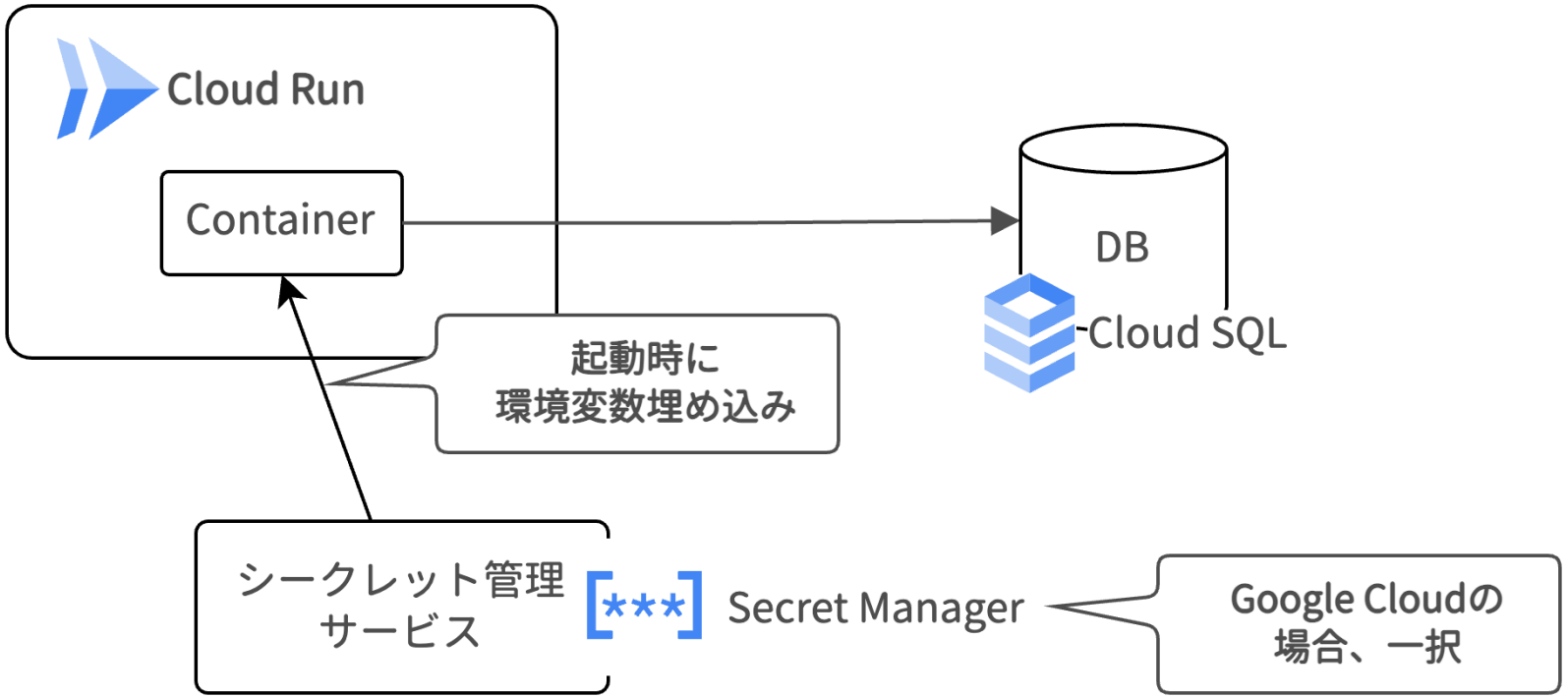
クレデンシャル管理サービスを活用し、
アプリケーション起動時に環境変数として埋め込むことで安全に利用



Amazon ECSではタスク定義にSecrets ManagerやSSM Parameter Storeの参照先を定義し、起動時に環境変数として埋め込み



Cloud RunではSecret Managerを参照先として定義し、起動時に環境変数として埋め込み



非機能デザインからの理解 – セキュリティ/クレデンシャル管理

まとめ

Amazon ECS

- ECSタスク定義にSecrets ManagerやSSM Parameter Store参照先を定義し、コンテナ起動時に環境変数として埋め込み

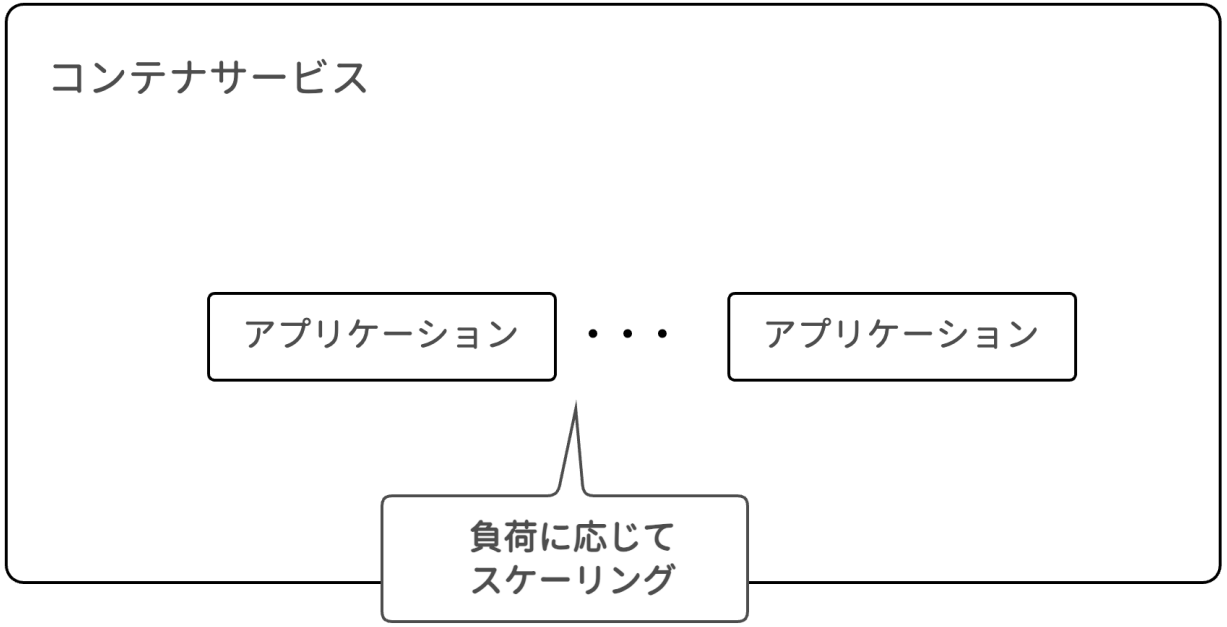
Cloud Run

- Cloud Run定義にSecret Managerの参照先を定義し、コンテナ起動時に環境変数として埋め込み

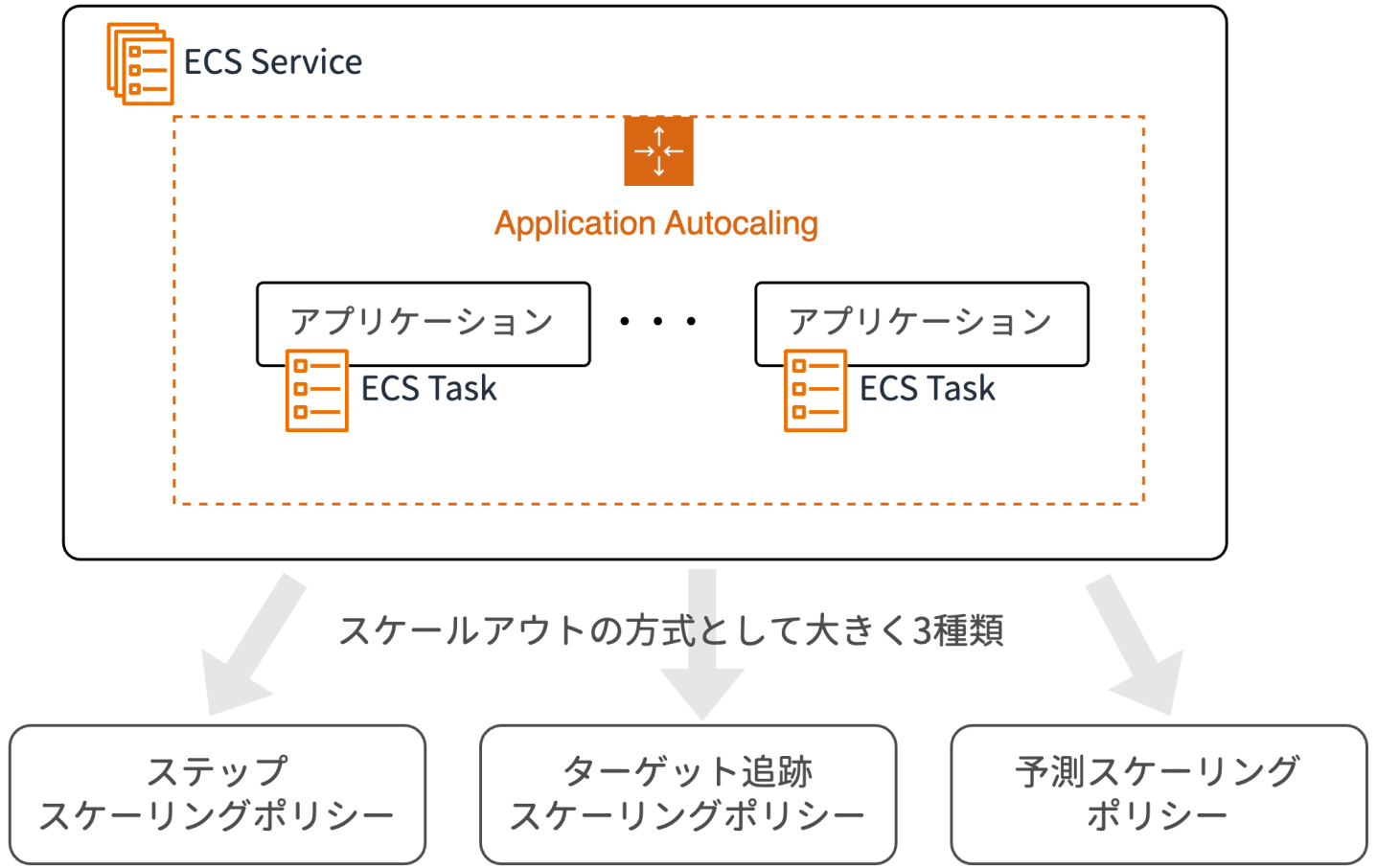
パフォーマンス – スケールアウト設計

非機能デザインからの理解 – パフォーマンス/スケールアウト設計

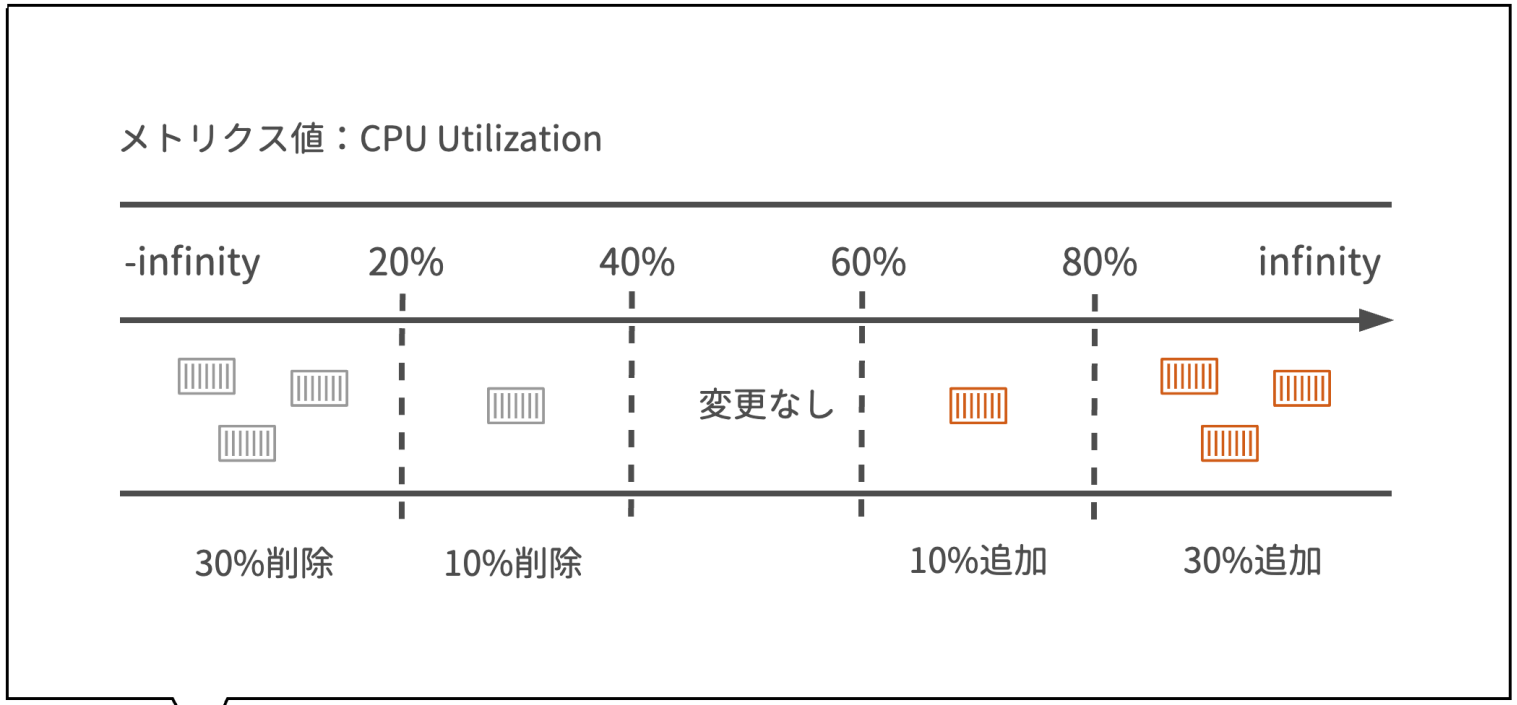
Amazon ECSとCloud Runではスケールアウト挙動のバリエーションが大きく異なる



Amazon ECSでは、3種類のスケーリングポリシーと スケールトリガ条件となるメトリクスを選択することで挙動を定義



Amazon ECSでは、3種類のスケーリングポリシーと スケールトリガ条件となるメトリクスを選択することで挙動を定義

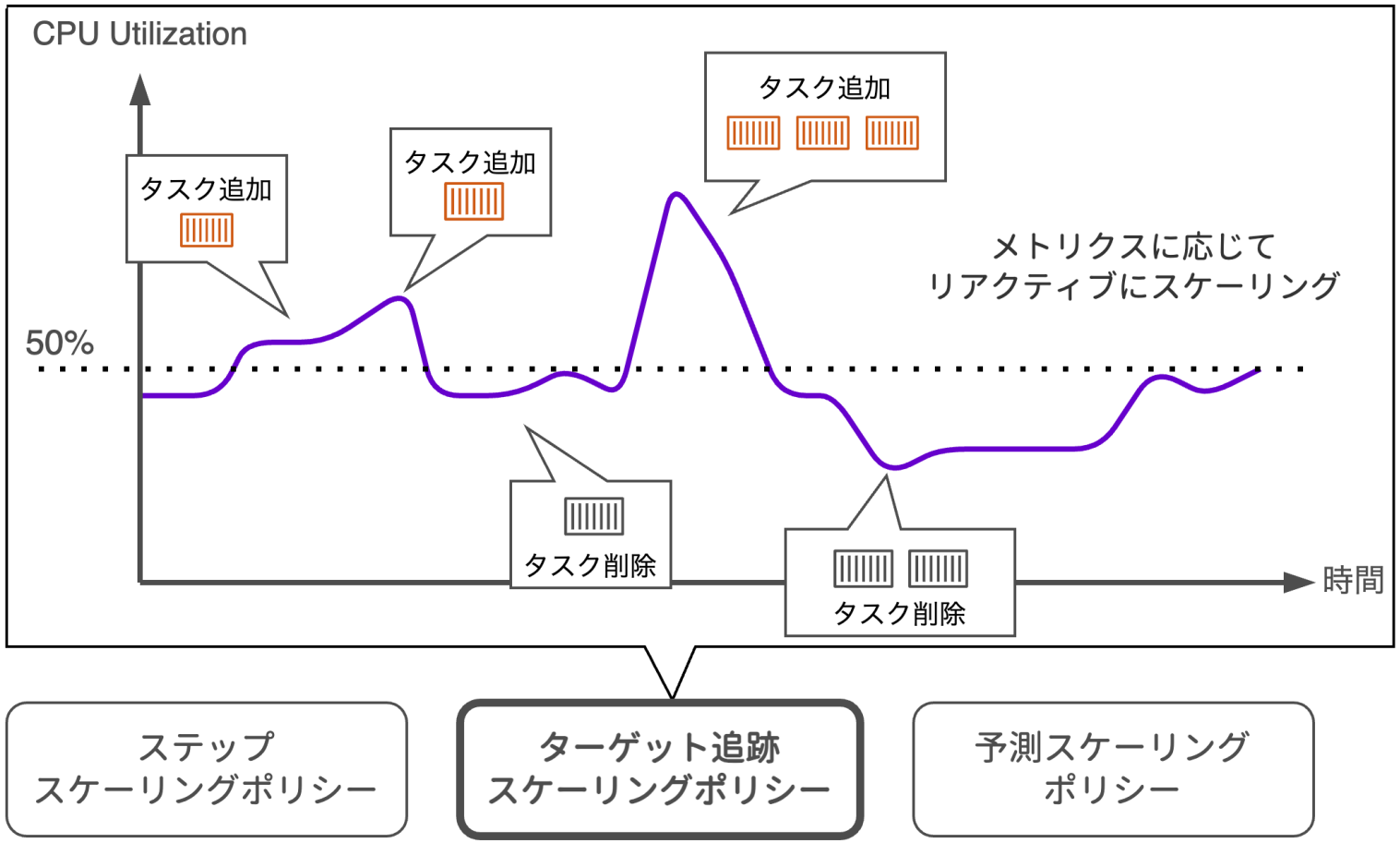


ステップ スケーリングポリシー

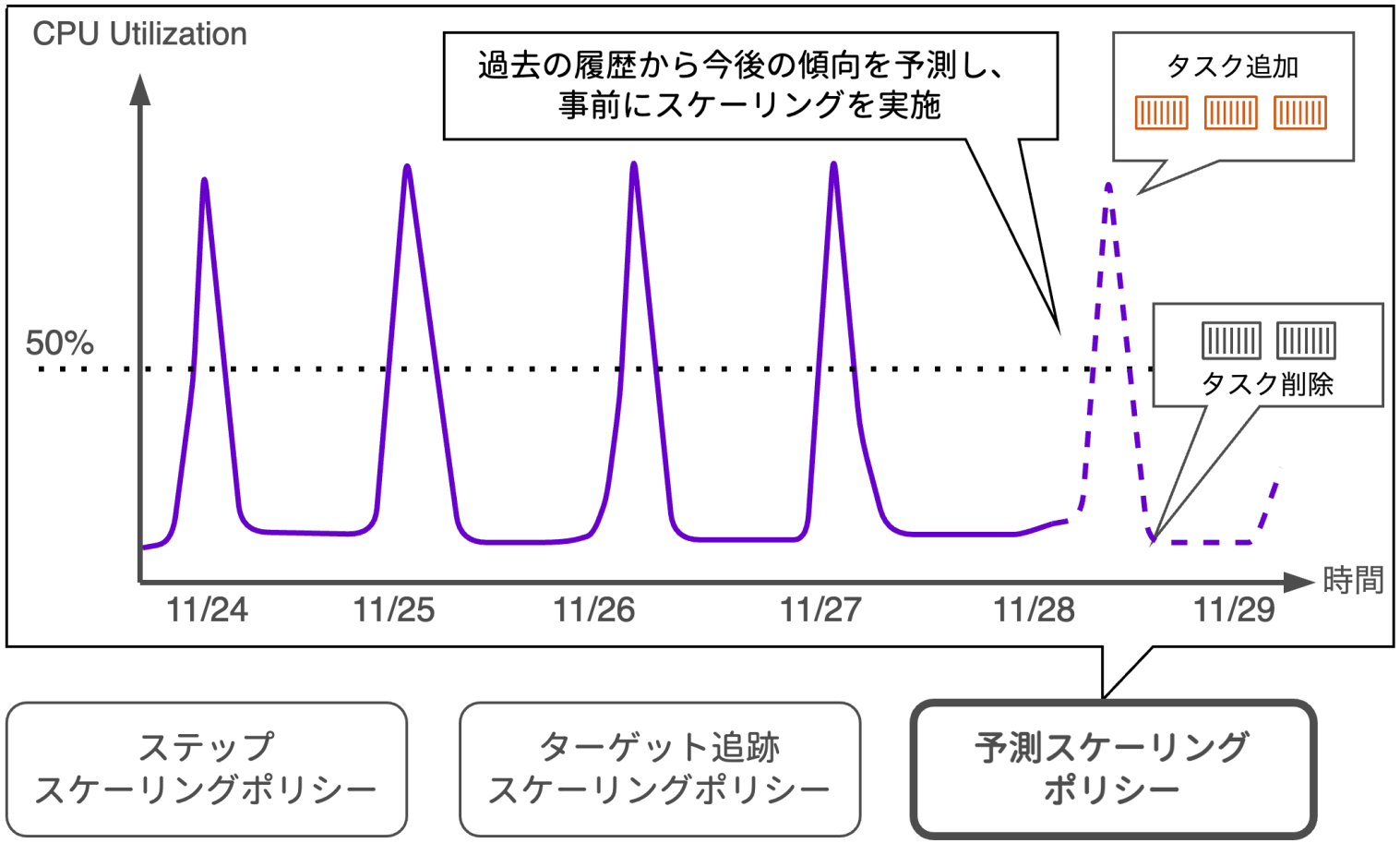
ターゲット追跡 スケーリングポリシー

予測スケーリング ポリシー

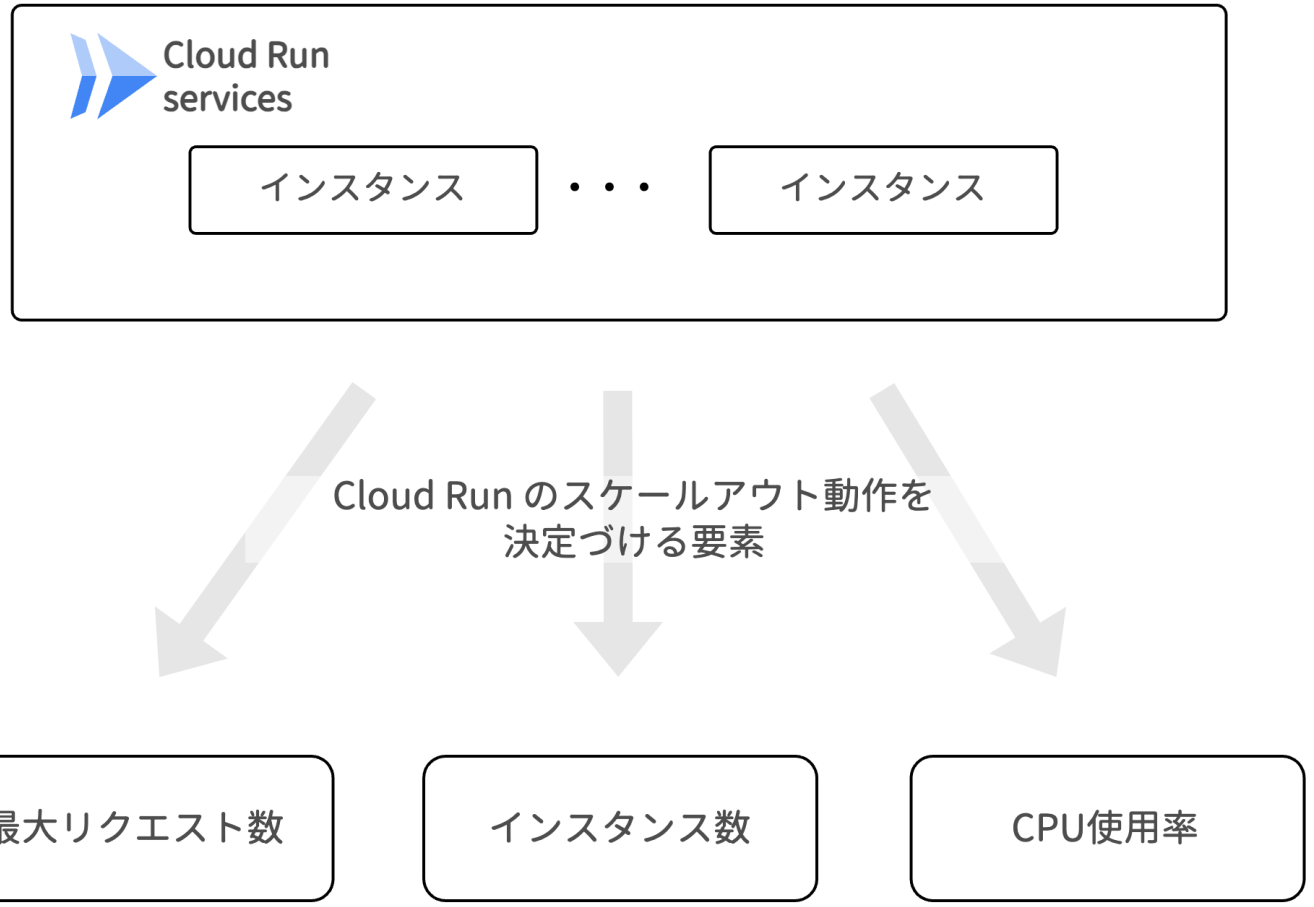
Amazon ECSでは、3種類のスケーリングポリシーと スケールトリガ条件となるメトリクスを選択することで挙動を定義



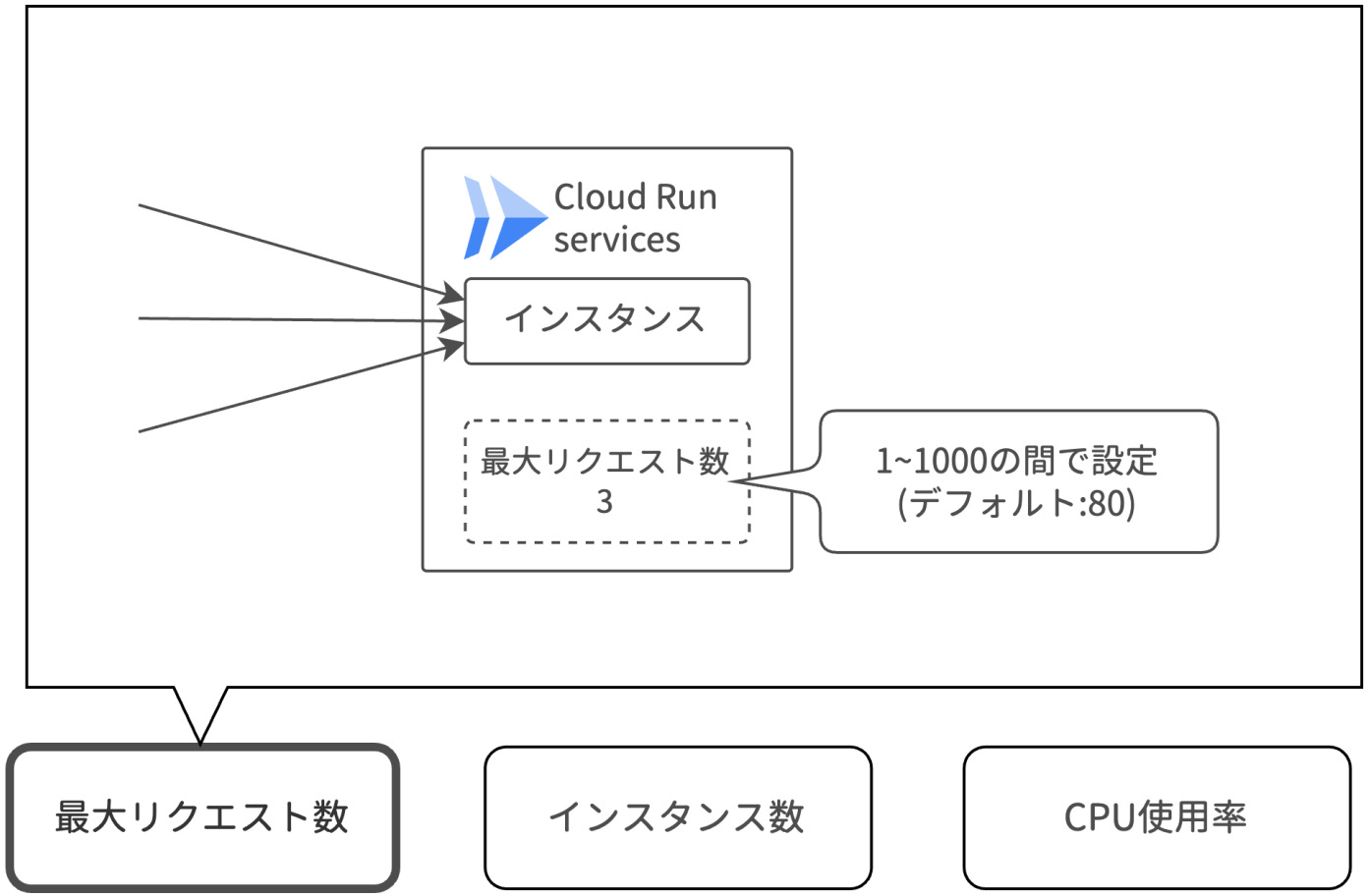
Amazon ECSでは、3種類のスケーリングポリシーと スケールトリガ条件となるメトリクスを選択することで挙動を定義



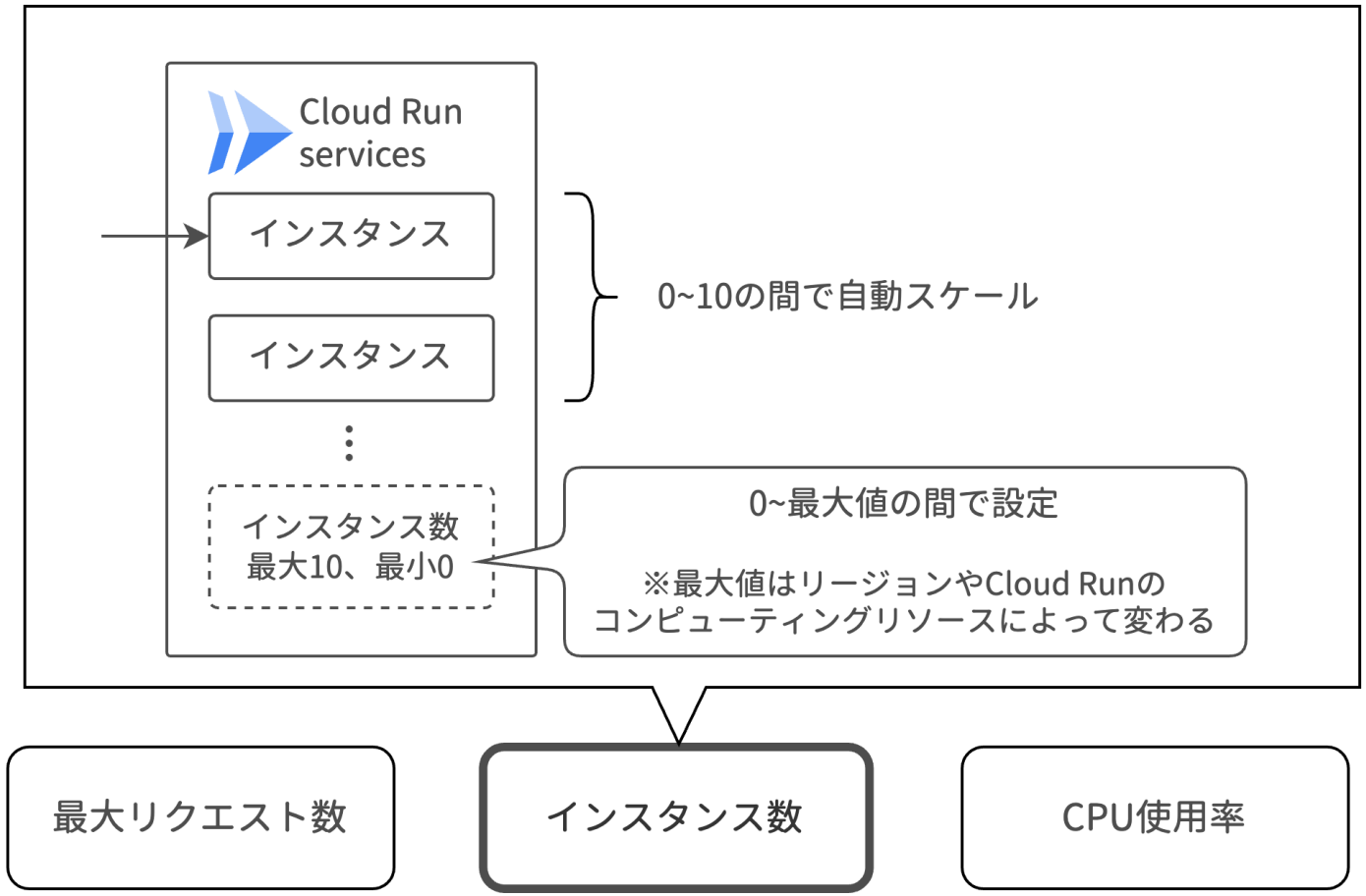
Cloud Runの場合、スケールアウトロジックは内部仕様として単一であり、最大リクエスト数・インスタンス数・CPU使用率により動作が決まる



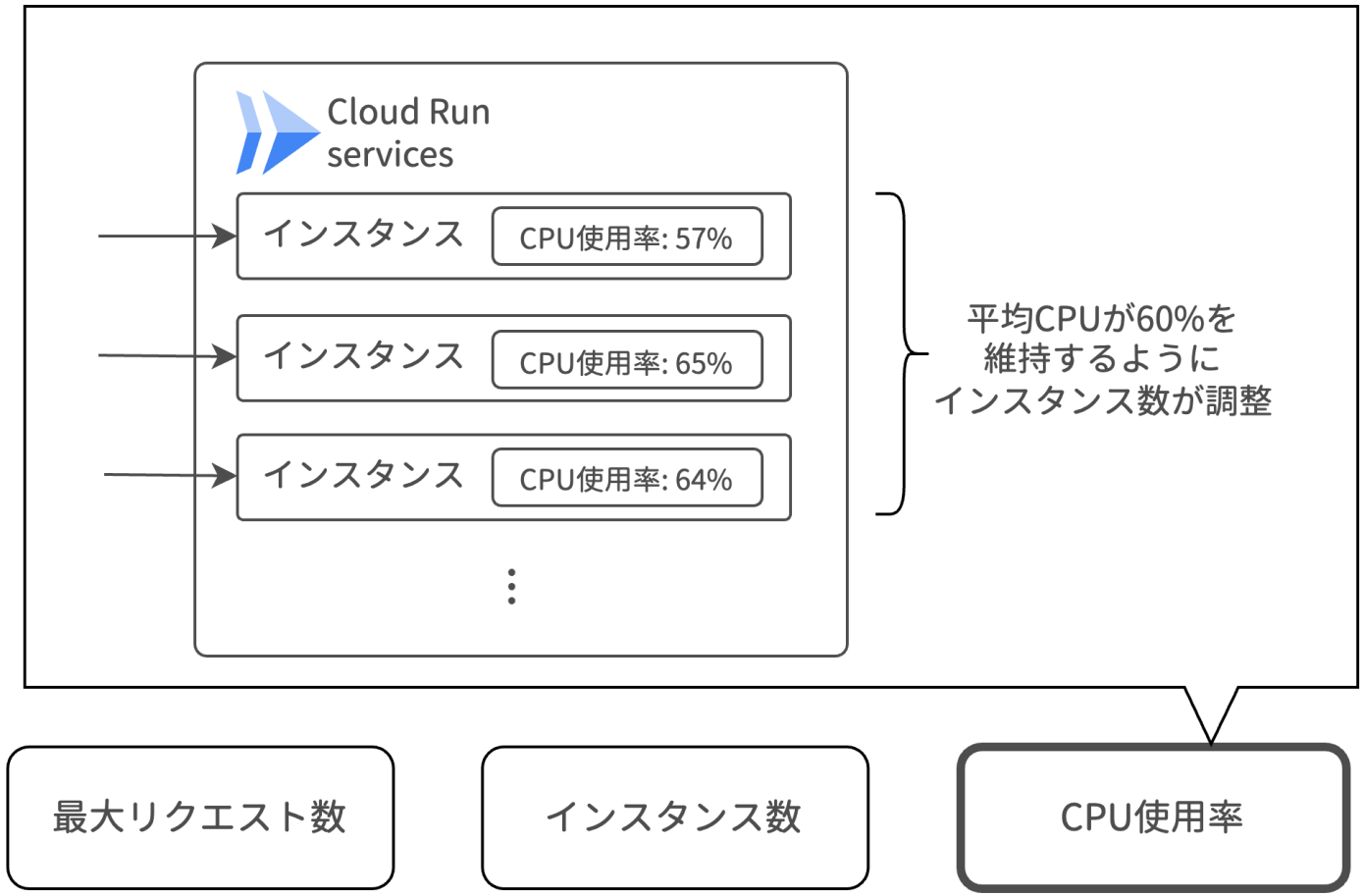
Cloud Runの場合、スケールアウトロジックは内部仕様として単一であり、最大リクエスト数・インスタンス数・CPU使用率により動作が決まる



Cloud Runの場合、スケールアウトロジックは内部仕様として単一であり、最大リクエスト数・インスタンス数・CPU使用率により動作が決まる



Cloud Runの場合、スケールアウトロジックは内部仕様として単一であり、最大リクエスト数・インスタンス数・CPU使用率により動作が決まる



まとめ

Amazon ECS

- ステップスケーリングポリシー or ターゲット追跡スケーリングポリシーに従ってスケールアウト

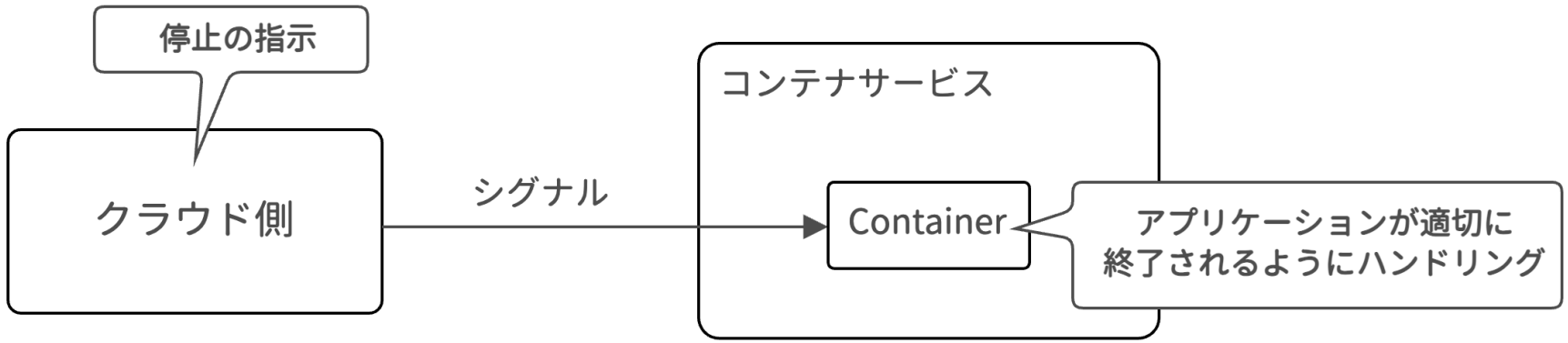
Cloud Run

- 最大リクエスト、インスタンス数、CPU使用率を基に、Google Cloudの内部仕様に従ってスケールアウト

信頼性 – シグナルハンドリング

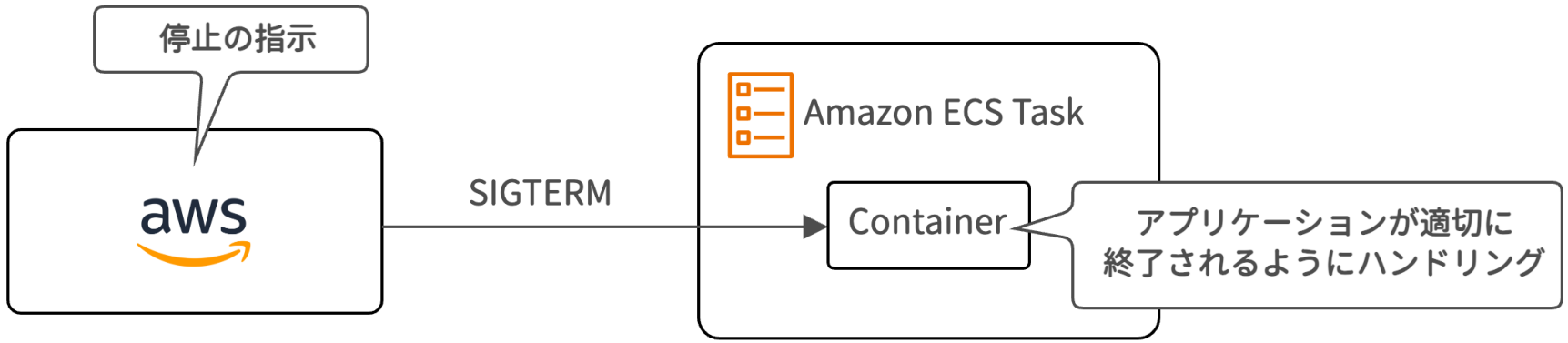
非機能デザインからの理解 – 信頼性/シグナルハンドリング

メンテナンスやスケールダウン時において
アプリケーションを安全に停止するためにはシグナルハンドリングが重要



非機能デザインからの理解 – 信頼性/シグナルハンドリング

Amazon ECSでは停止指示としてSIGTERMが発行される



参考）https://docs.aws.amazon.com/ja_jp/AmazonECS/latest/developerguide/task-lifecycle-explanation.html

非機能デザインからの理解 – 信頼性/シグナルハンドリング

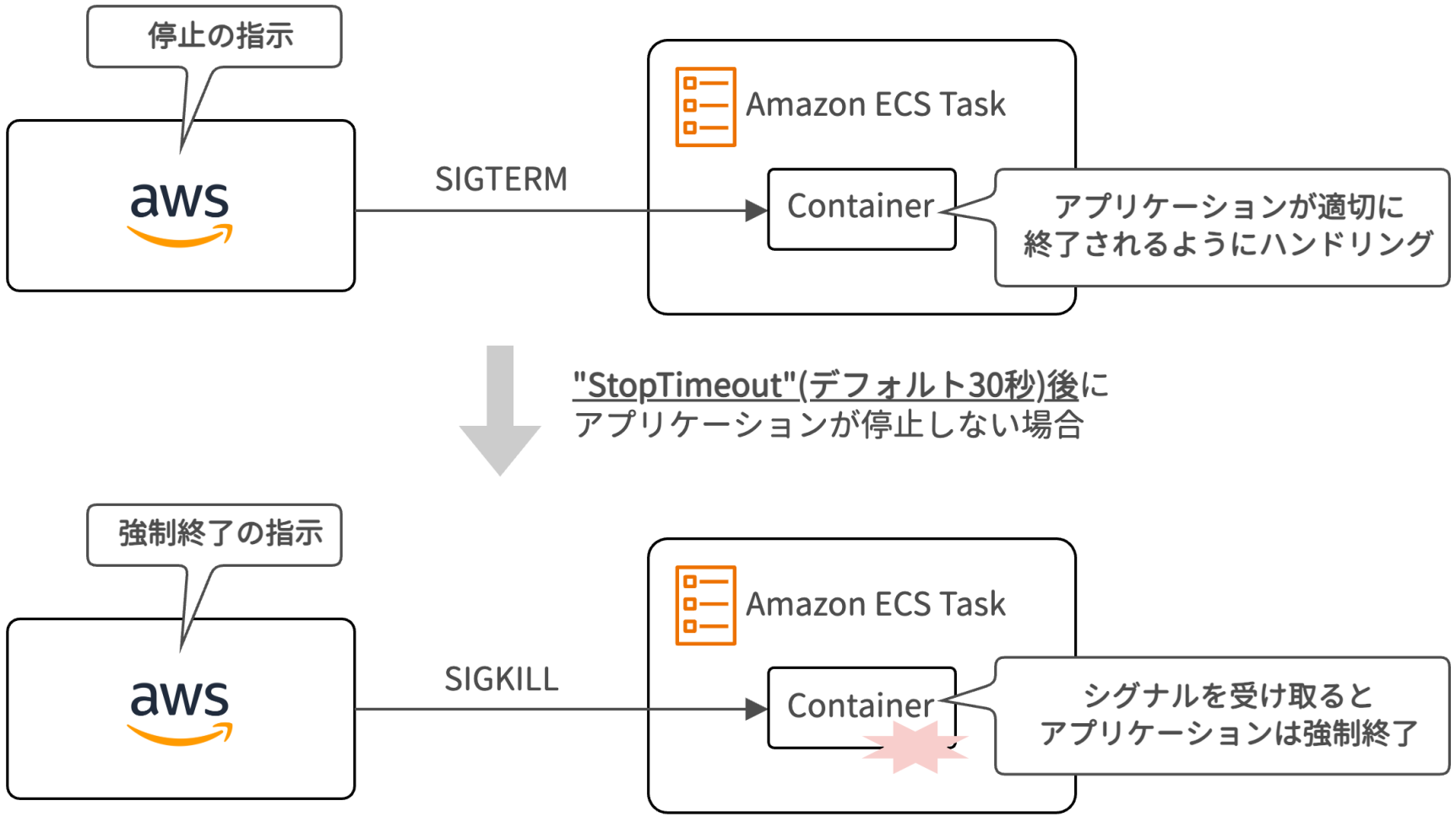
Amazon ECSでは停止指示としてSIGTERMが発行される

一定時間以内にアプリケーションが
終了しなかったらどうなる？

参考）https://docs.aws.amazon.com/ja_jp/AmazonECS/latest/developerguide/task-lifecycle-explanation.html

非機能デザインからの理解 – 信頼性/シグナルハンドリング

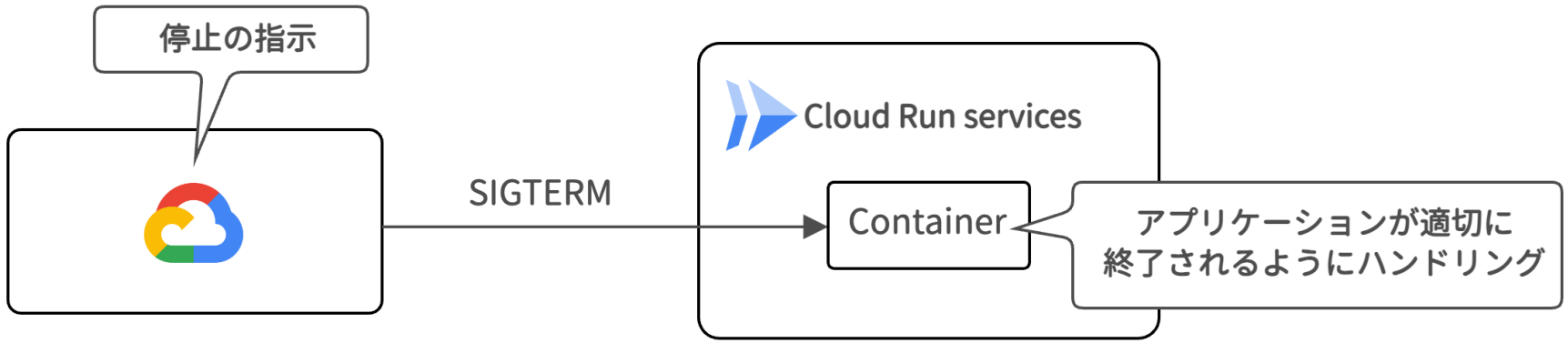
タイムアウト値以内にアプリケーションが停止しない場合、SIGKILLにより強制終了される



参考）https://docs.aws.amazon.com/ja_jp/AmazonECS/latest/developerguide/task-lifecycle-explanation.html

非機能デザインからの理解 – 信頼性/シグナルハンドリング

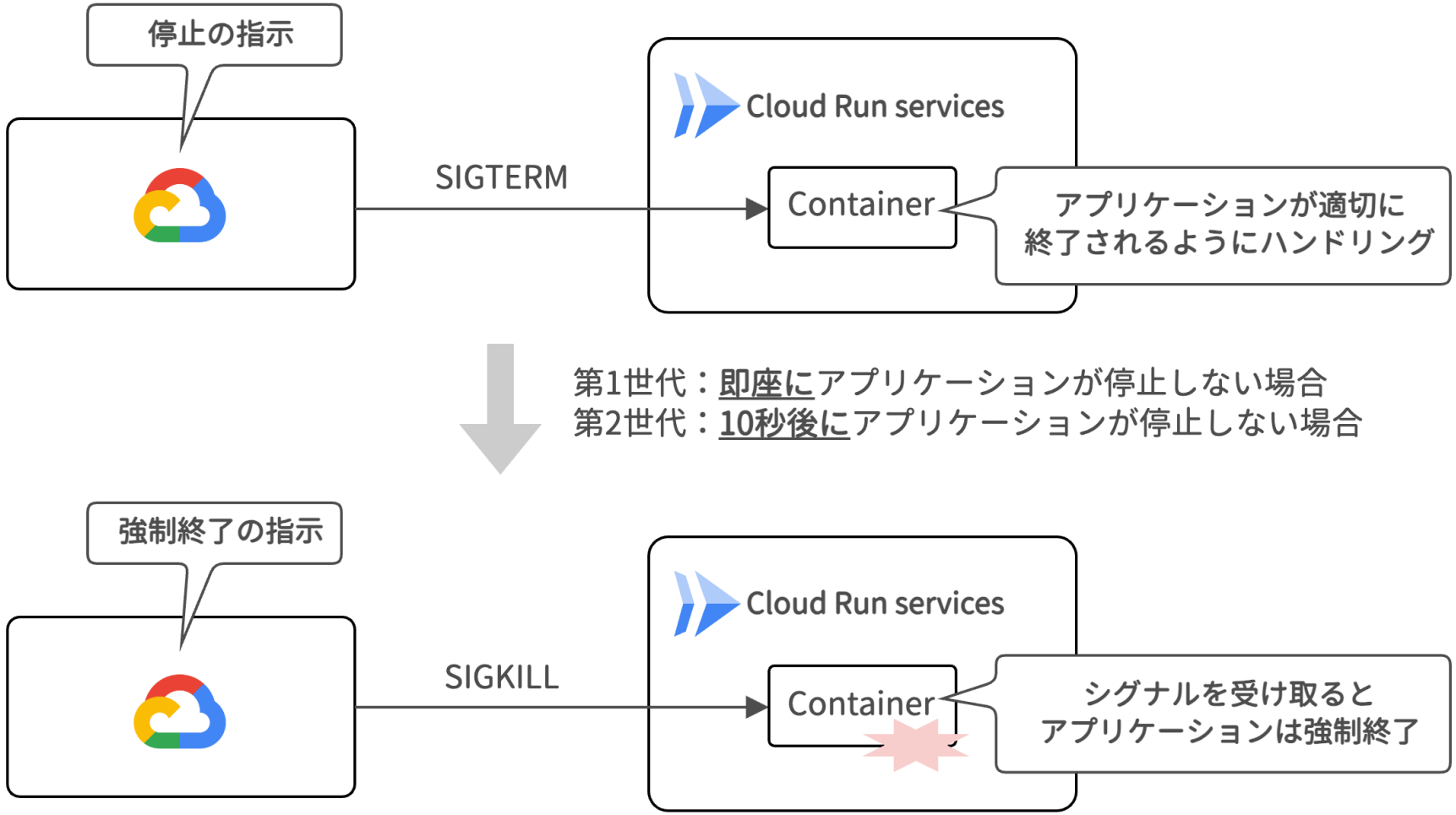
Cloud Runにおいても同様に、停止指示としてSIGTERMが発行される



参考）https://cloud.google.com/run/docs/samples/cloudrun-sigterm-handler?hl=ja

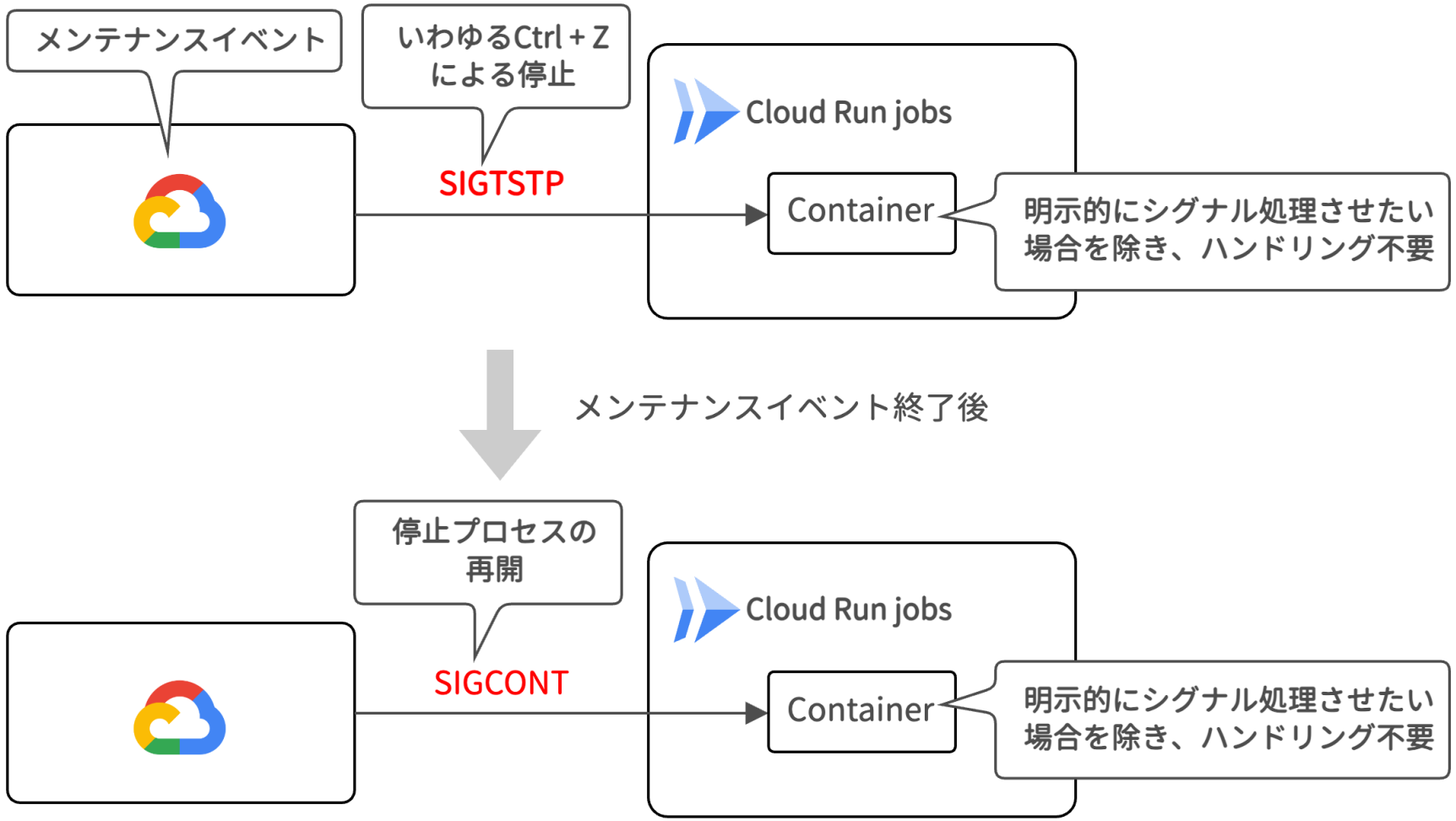
非機能デザインからの理解 – 信頼性/シグナルハンドリング

SIGKILLによる強制終了はECSと同じ。ただしシグナル発行までの猶予時間に関してはCloud Runの世代によって挙動が変わる



参考）https://cloud.google.com/run/docs/samples/cloudrun-sigterm-handler?hl=ja

参考）Cloud Run jobsに関しては、メンテナンスイベント時に一時停止用のシグナルが発行される



参考）https://cloud.google.com/run/docs/configuring/task-timeout?hl=ja#observing_and_handling_maintenance_events

まとめ

Amazon ECS

- 停止指示としてSIGTERMが発行
- 強制停止指示としてSIGKILLが発行

Cloud Run

- 停止指示としてSIGTERMが発行
- 強制停止指示としてSIGKILLが発行
- Cloud Run jobsに関してはメンテナンスイベント時にSIGTSTPとSIGCONTが発行

コスト最適化 – 課金モデル

非機能デザインからの理解 – コスト最適化/課金モデル

コスト観点比較の特徴はARM、Spot、リクエスト時CPU割り当て

非機能デザインからの理解 – コスト最適化/課金モデル

コスト観点比較の特徴はARM、Spot、リクエスト時CPU割り当て

独自のCPUプロセッサ
(Gravitonプロセッサ)

割り込みにより停止の可能性がある
Fargate起動タイプ

非機能デザインからの理解 – コスト最適化/課金モデル

コスト観点比較の特徴はARM、Spot、リクエスト時CPU割り当て

Amazon ECS

- ECS自体は無料。Fargateに割り当てるCPU/メモリ/ストレージで料金が発生
- ARMプロセッサ採用により、CPU/メモリのコストを約20%減
- Fargate Spot採用により、CPU/メモリのコストを約70%減

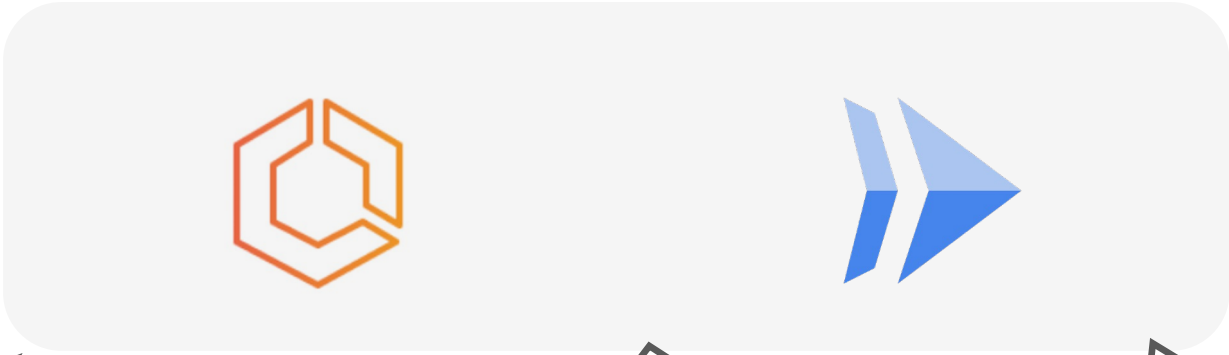
Cloud Run

- Cloud Runに割り当てるCPU/メモリで料金が発生
- リクエスト時CPU割り当てにすることでCPU/メモリの料金を削減可能(100msec切り上げ、リクエスト数課金あり)

本発表のまとめ

まとめ

以下の3つの観点から相互理解を目指す



1. 基本的な特徴

- コンポーネントとワークロード
- VPC内外
- アクセス制御
- コンテナ実行環境

2. アーキテクチャデザインの違い

- インターネットアクセス
- サービス間内部通信
- インターネットへの外部通信
- ジョブワークフロー

3. 非機能デザインからの理解

- 運用デザイン
- セキュリティ
- パフォーマンス
- 信頼性
- コスト最適化

ご清聴ありがとうございました。