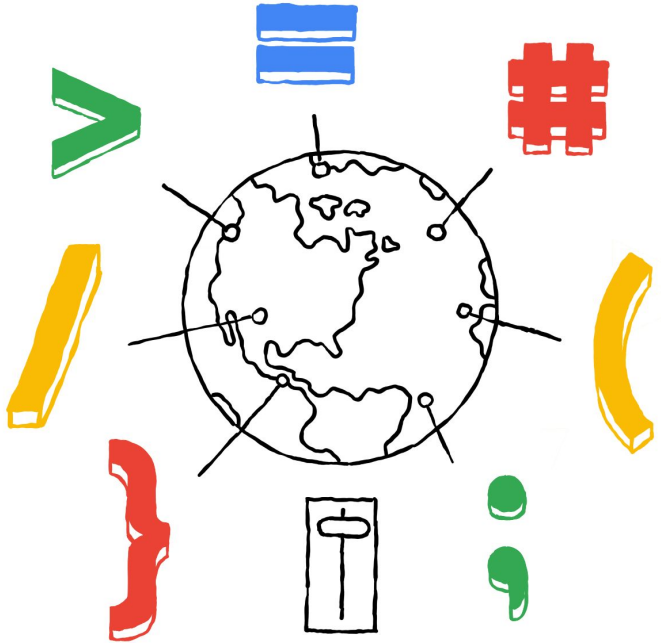


徹底解説！
Google Cloud
モダン アーキテクチャ パターン

Google Cloud
ソリューションズ アーキテクト
長谷部 光治

長谷部 光治

Google Cloud
ソリューションズ アーキテクト

Technology area:
Hybrid cloud, Application Development, Infrastructure

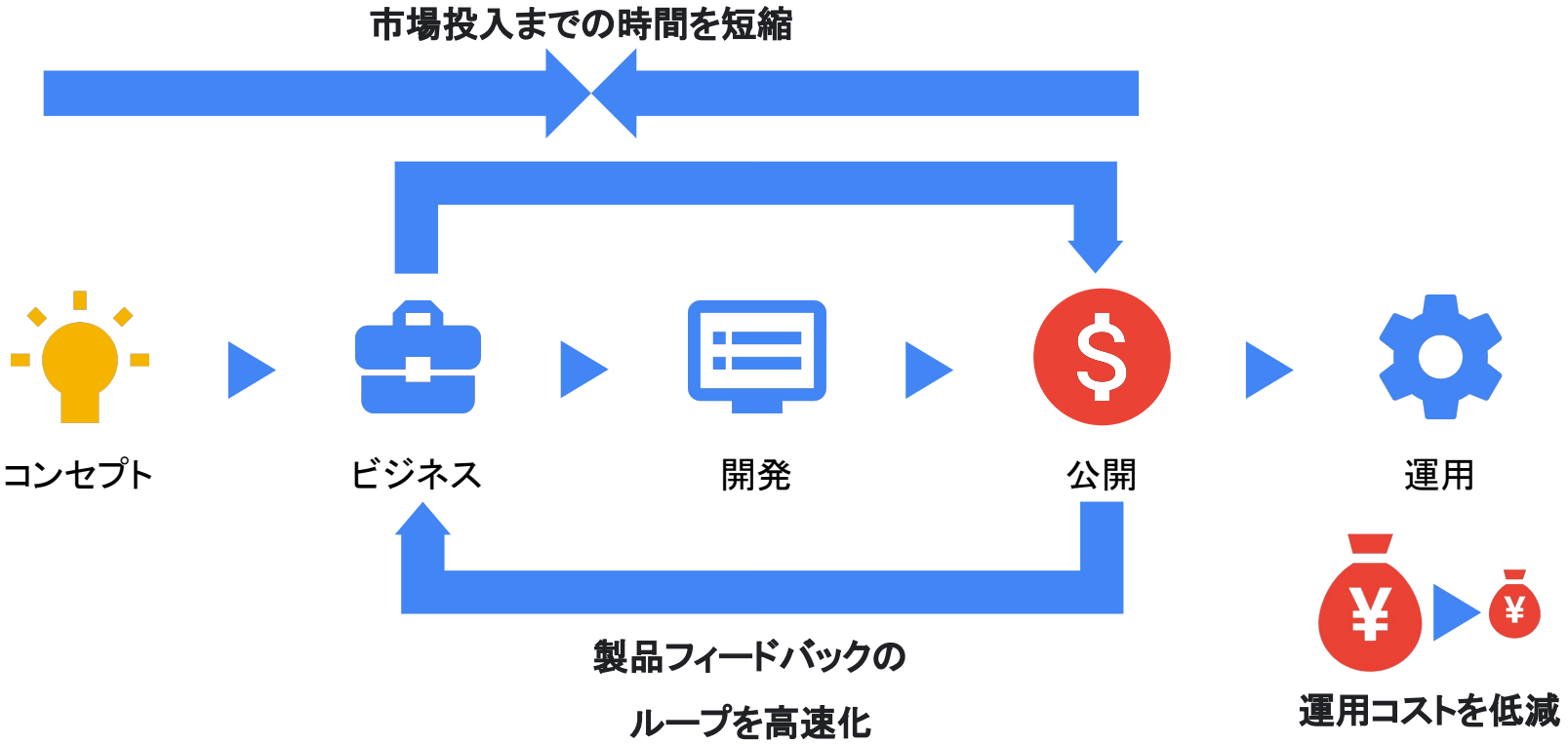
Agenda

1. モダンアーキテクチャ
2. コンテナ化、マイクロサービス
3. サービスメッシュ、グローバル展開
4. 宣言型による定義、単一の仕組みで管理
5. 本番導入に向けて
6. まとめ

モダン
アーキテクチャ

Google Cloud

なぜ ”モダン” なアーキテクチャにする？

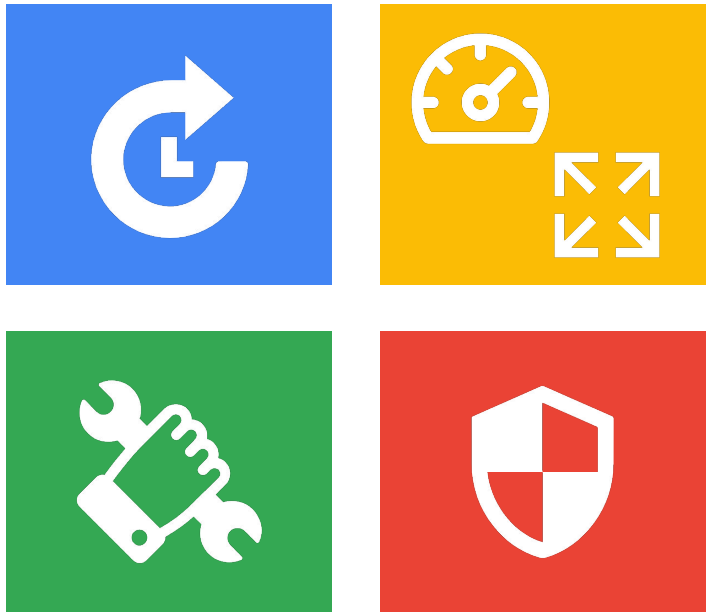


DevOps の考え方が参考になります。詳細については、本Webinar 過去セッション「Google 流、理想的な CI / CD を導入するための考え方とその実践」をご覧ください

“モダン” なアーキテクチャとは？

それぞれの項目を高いレベルで満たした
アプリケーションを構築するためのアーキテクチャ

- 可用性
- 性能・拡張性
- 運用保守性
- セキュリティ



Google が考えるアプリケーションの未来

アプリケーションの未来、それを動かすインフラは
コンテナ化された**マイクロサービス**で作られる。

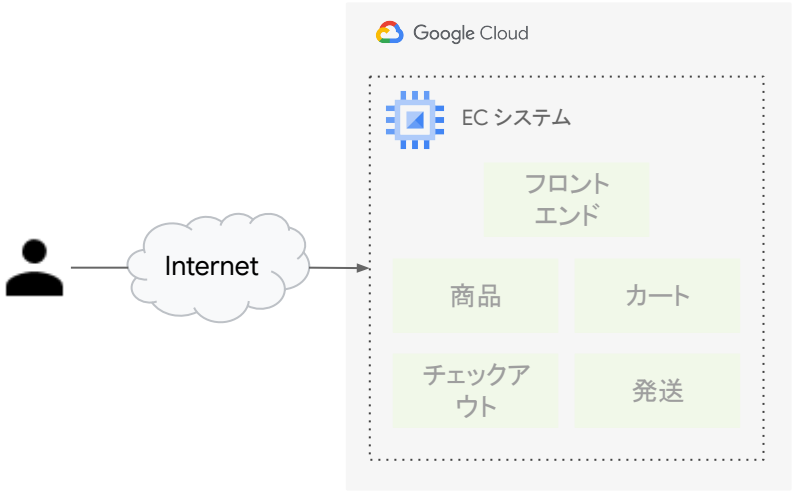
そして、それらは**宣言型**で定義されており
単一の仕組みを通じて管理される。

また、**サービスメッシュ**を利用し
あらゆるロケーションにサービスを提供する。

サンプルアプリケーション

- 小規模 EC サイト
- 仮想マシンで動いている
- 主にワークロード部分に注目

これをモダンなアーキテクチャにしていく



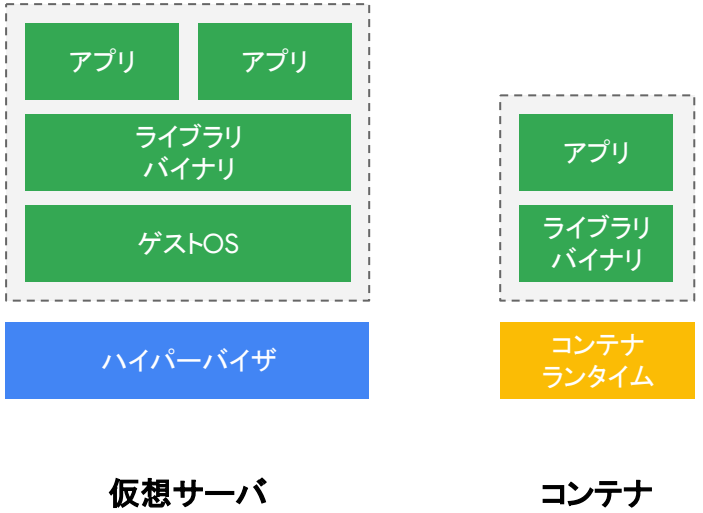
コンテナ化、
マイクロサービス

02

Google Cloud

コンテナとは？

- アプリケーションと、その実行に
  必要なものをまとめたもの（イメージ）
- 通常、1コンテナあたり1アプリケーション
- 形式が標準化されている



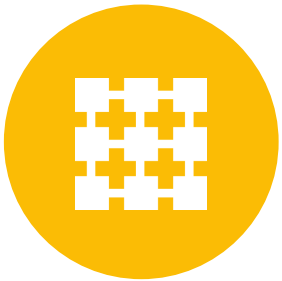
コンテナを使うメリット

高速

VM より高速に
数 ms で起動

ポータビリティ

コンテナ実行環境間で
容易な移行



効率性

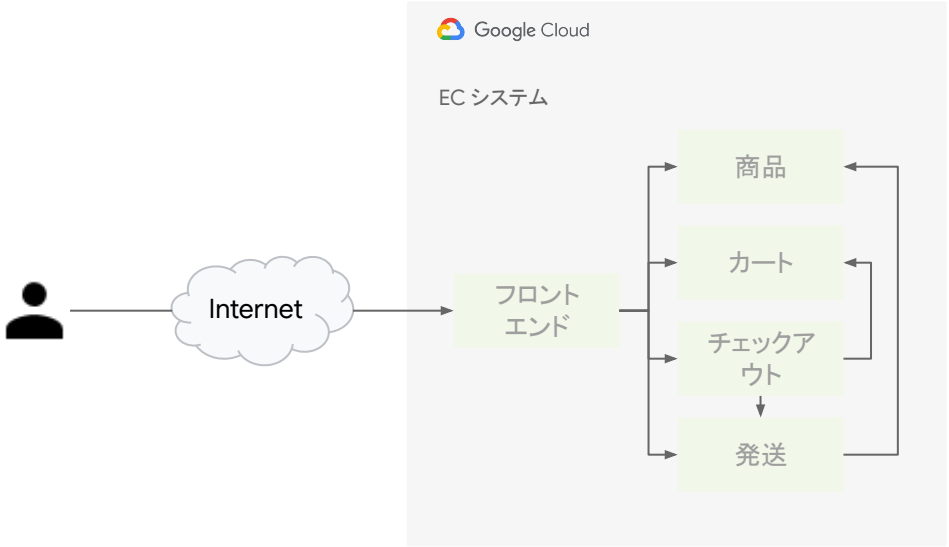
低オーバーヘッドで
リソースの有効利用

マイクロサービス

アプリケーションを複数のサービスを組みあわせ
作り上げる構成パターン（あくまで手段）

個々のサービスは

- 運用、テストしやすい
- 疎結合
- ビジネス要件に基づいて分けられる



本例では 5 つのサービスで
EC システムを構成している

複雑なアプリケーションを素早く、頻繁に信頼性のある形でリリースするため

コンテナ オーケストレーション
- Kubernetes -

コンテナにおける複数のチャレンジ

- コンテナ管理
- ネットワーク
- スケーリング
- デプロイ、更新
- コンテナ間のディスカバリ
- ヘルスチェック
- 永続データの扱い
- ロギング、監視
- デバッグ
- etc...

Kubernetes

特徴

- オープンソース
- Google の社内コンテナ管理ツールである Borg にインスパイア
- 宣言型 API を通じて設定した“あるべき状態” に自動的に収束
- 高い拡張性
- デファクト スタンダード

Google Cloud

詳細については、本Webinar 過去セッション
「Google のベストプラクティスから生まれたKubernetes のオープンソースエコシステム」をご覧ください

Google Cloud Platform におけるロケーションの概念
- リージョン、ゾーン -

ゾーン

- GCP のリソースを配置する地域

リージョン

- 複数のゾーン*1 で構成された地理エリア

リソースの配置例

仮想マシンを、東京リージョンの a ゾーンに配置

単体ゾーン、リージョン全体が利用不能に
陥ることを想定してアプリケーションを配置する

*1 通常 3 ゾーン

*2 例外として 4 つのゾーンを持つリージョン

Google Cloud

コンテナ化、マイクロサービス
- 推奨プラクティス on GCP -

- 権限、コスト、リソース上限値の分離
- あらゆるレイヤーで可用性、拡張性を担保
- クラスタの有効利用、一貫した管理
- ネットワーク構成をシンプルに

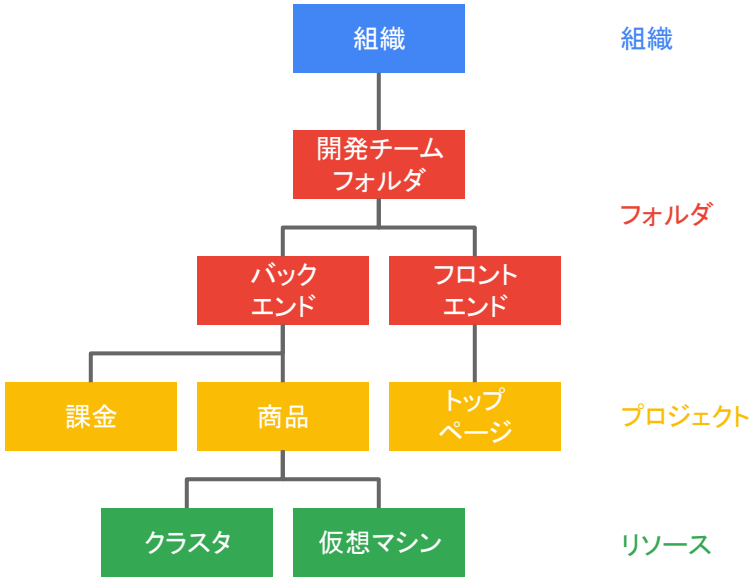
権限、コスト、リソース上限値の分離
- プロジェクトの適切な利用 -

Google Cloud プロジェクトを適切に使い、下記を分離する

- 利用可能な API（機能）
- 課金、コスト情報
- リソースに対する権限
- リソース上限値

特にリソース上限値に達してしまった場合、プロジェクトを分けるしか対応方法がない事があるため慎重に検討する

また、これらのリソースを階層を使って管理する



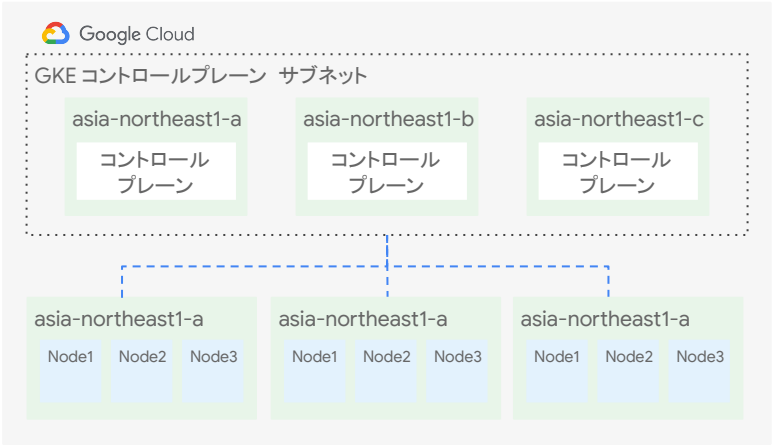
詳細については、「エンタープライズ企業のベストプラクティス」をご覧ください

Google Cloud

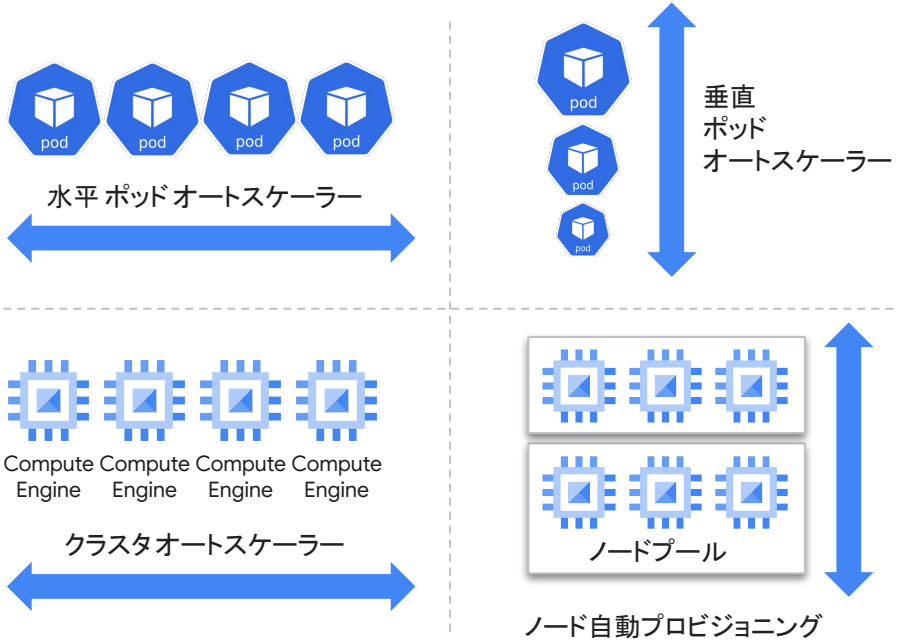
あらゆるレイヤーで可用性、拡張性を担保
- マルチゾーン、マルチリージョン -

可用性

- リージョンクラスタによるマルチゾーン展開
- マルチリージョンでの可用性を担保するため、複数リージョンへ展開



拡張性



Google Cloud

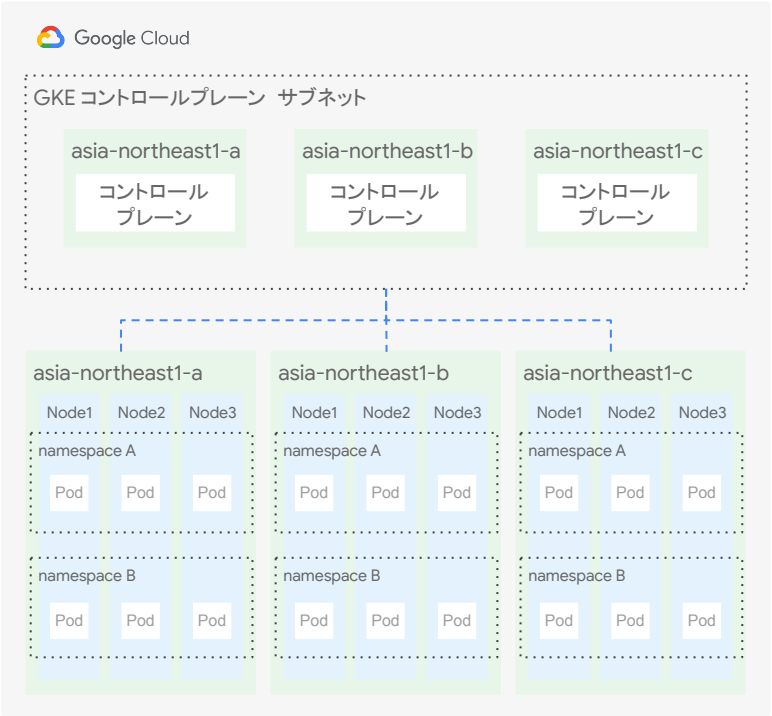
クラスタの有効利用、一貫した管理
- マルチテナント -

マルチテナントとは、クラスタを複数のテナントで
共有する設計パターン

目的

- クラスタリソースの利用率向上によるコスト削減
- 一貫した管理ポリシーをテナントに適用

Kubernetes のクラスタでは namespace を
使ってテナントを分離する



詳細については、「エンタープライズ マルチテナンシーのベストプラクティス」をご覧ください

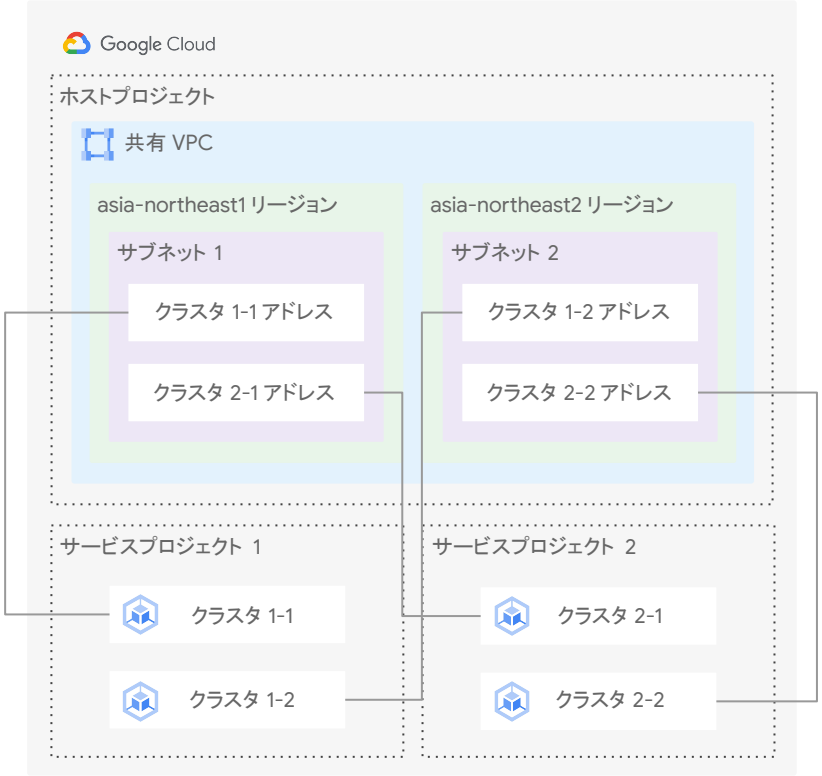
ネットワーク構成をシンプルに
- 共有 VPC -

共有 VPC を使い、フラットなネットワークをベースとする

- 構築、運用の簡単さ
- ネットワークリソースの集中管理
- ネットワーク管理者、プロジェクト管理者の権限分離

注意

- 組織リソースが必要

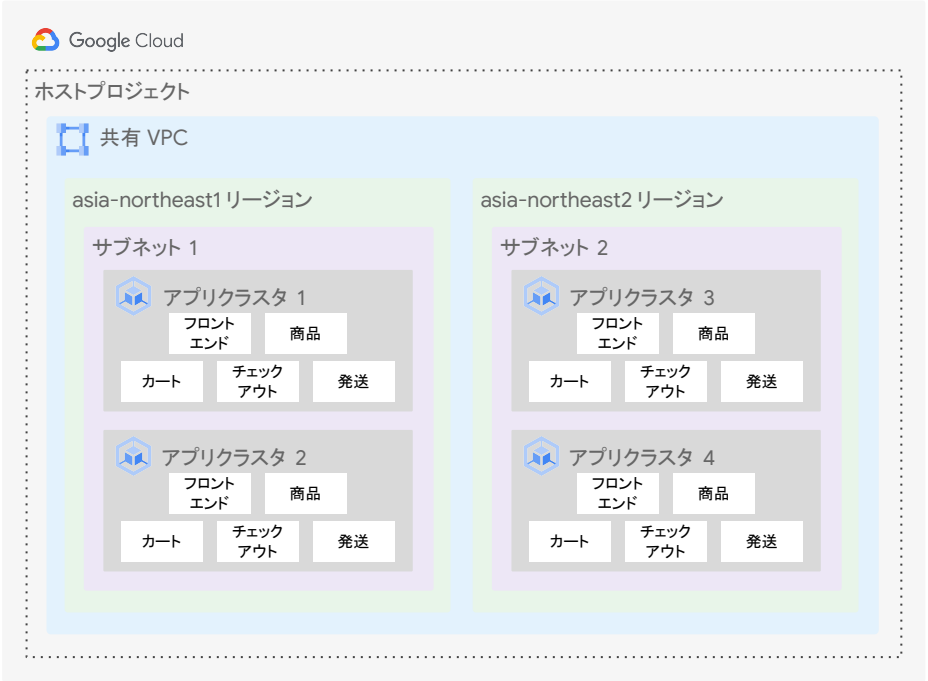


詳細については、「VPC 設計のためのベストプラクティスとリファレンスアーキテクチャ」をご覧ください

Google Cloud

プラクティス適用後 GCP アーキテクチャ

- 全てのクラスタに対しマイクロサービスごとに namespace を作成
- 共有 VPC を利用
- 東京リージョン、大阪リージョンに環境を準備
- アプリクラスタ 1 - 2 、3 - 4 用に別のプロジェクトを作成
- 1 リージョンあたりゾーンを分けて、2 クラスタを用意するため、ゾーンクラスタで構築



※ サービスプロジェクトは省いています

サービスメッシュ、グローバル展開

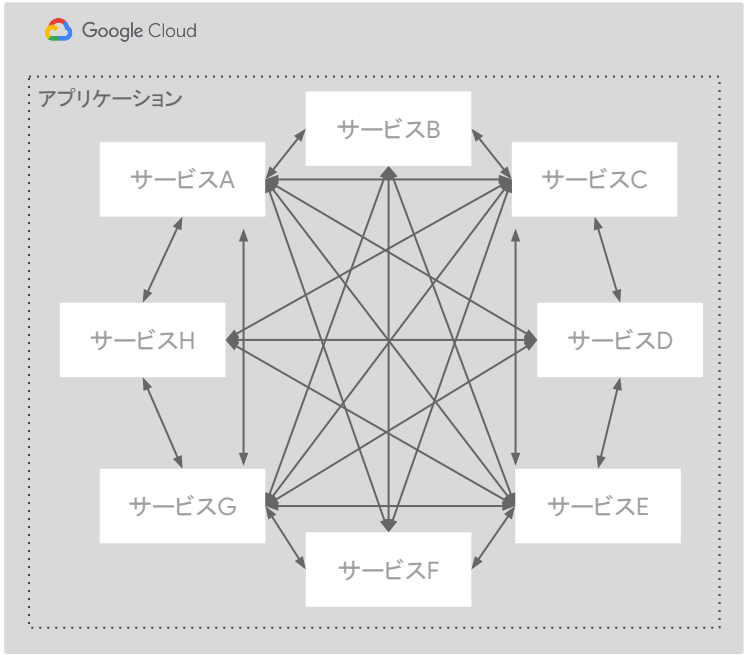
Google Cloud

サービスメッシュとは？

アプリケーションを構成するマイクロサービス間の
ネットワークとそれらの連携の問題を解決する方法

多数のマイクロサービスで構成されたアプリケーションの課題

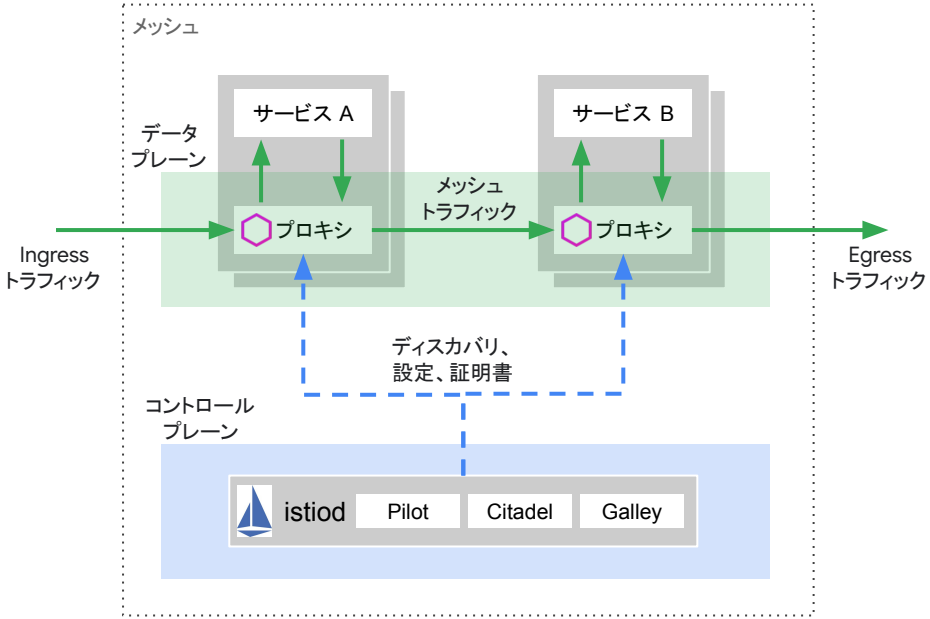
- サービス間連携が複雑化し、管理が困難
  - セキュリティ
  - モニタリング
- ネットワーク関連機能が各アプリケーションに散在



マイクロサービス間連携の問題を解決
- Istio, Anthos Service Mesh -

特徴

- オープンソース
- プロキシ（Envoy）を各アプリケーションに追加する
  - アプリケーションへの変更は最小限
- コントロールプレーン、データプレーンの 2 層構造
- ネットワーク関連機能をアプリケーションから分離
  - トラフィック管理
  - セキュリティ
  - 可観測性
- Anthos Service Mesh (ASM)
  - Istio のフルマネージド版



Istio アーキテクチャ

Google Cloud

ASM の詳細については、本Webinar 過去セッション
「Google が提供するフルマネージドのサービスメッシュ「Anthos Service Mesh (ASM)」とは」をご覧ください

グローバル展開

グローバル展開を見据えた
アーキテクチャ構成を採用する

- ネットワーク
- クラスタ
- ロードバランサ

例えば、日本の利用者を対象とする場合、
第 1 ステップとして東京リージョン
大阪リージョンの 2 リージョン構成から検討を進める

参考
リージョン間レイテンシ、スループットダッシュボード

サービスメッシュ、グローバル展開
- 推奨プラクティス on GCP -

- コントロールプレーンの可用性、拡張性を確保
- サービスの配置をシンプルに
- 利用者へのレスポンスを第一に

コントロールプレーンの可用性、拡張性を確保
- コントロールプレーンを各リージョンに配置 -

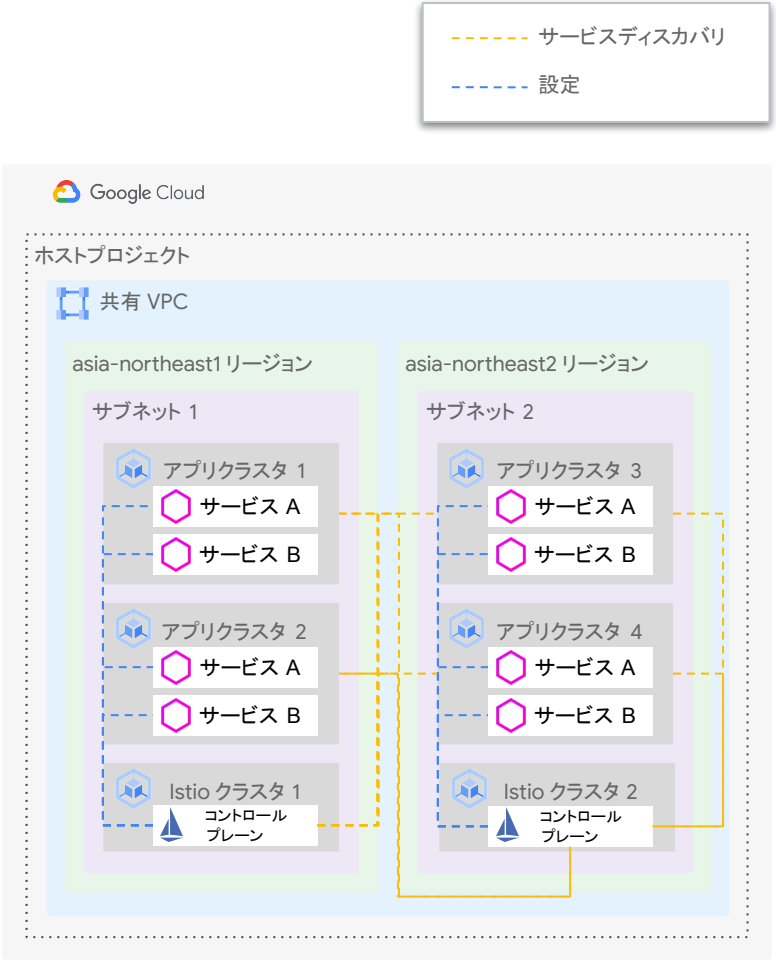
Istio アーキテクチャ設計には下記の検討が必要になる

- シングル or マルチクラスタ
- シングル or マルチネットワーク
- シングル or マルチコントロールプレーン
- シングル or マルチメッシュ

目的

- 可用性（フェイルオーバーも含む）、拡張性の向上
- 設定反映範囲の分離
- 構成のシンプルさ

詳細については、「Istio: Deployment Models」をご覧ください



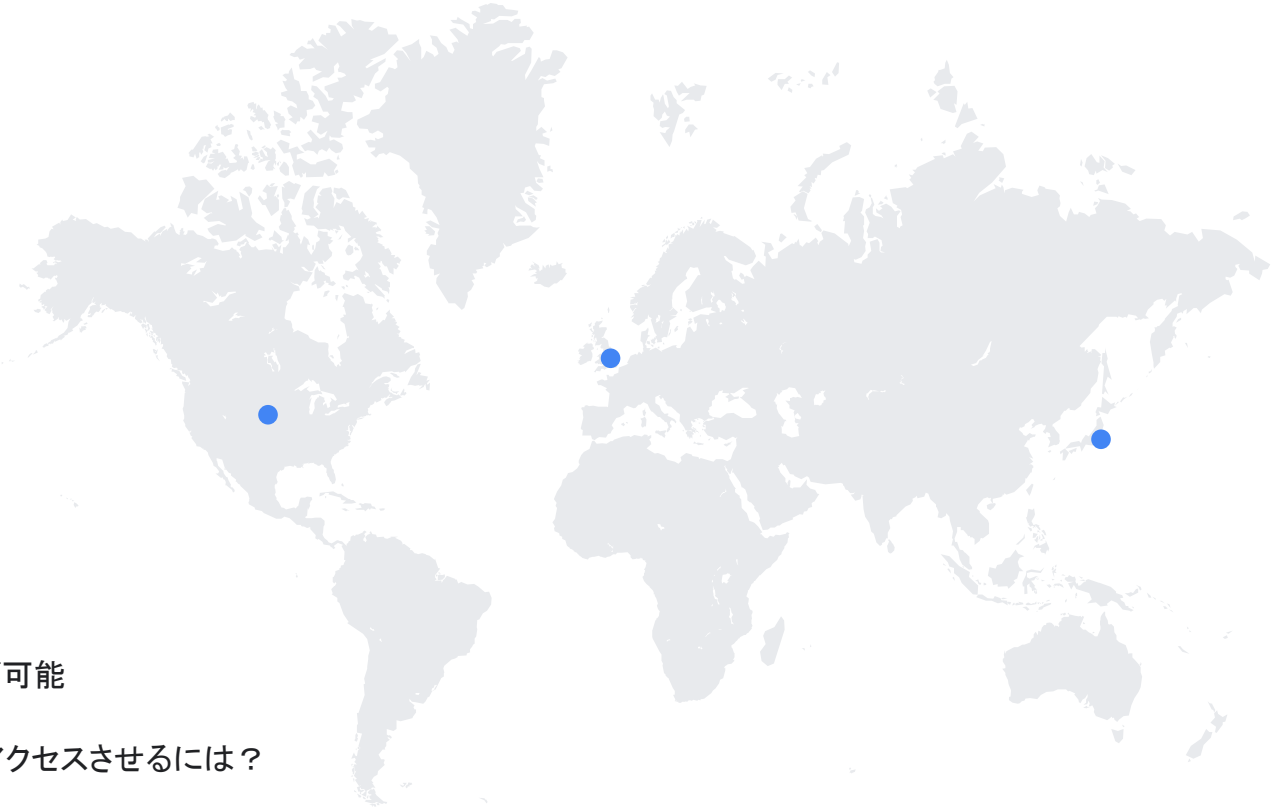
※ サービスプロジェクトは省いています

サービスの配置をシンプルに
- 各サービスを各ロケーションに同じように配置 -

ゾーン、リージョンごとの構成を揃え
シンプルな構成を目指す

- 管理のしやすさ
- 可用性の担保
- 各ロケーションでスケーリング

利用者へのレスポンスを第一に



利用者に近いところに
アプリケーションを配置する

例、アメリカ、ヨーロッパ
　日本市場がターゲットの場合

- us-central1: アイオワ
- europe-west2: ロンドン
- asia-northeast1: 東京

1VPC でグローバルに分散した
複数リージョンを一元管理することが可能

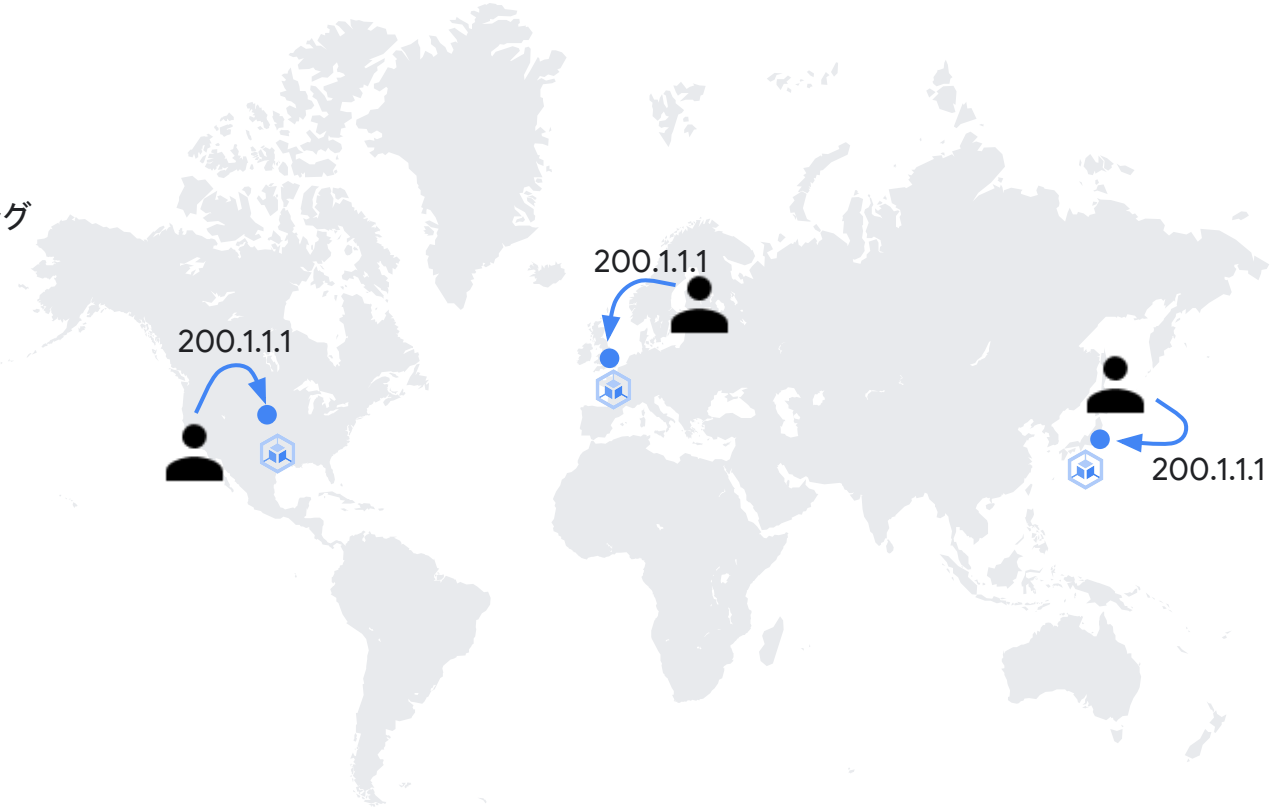
では、利用者に最適のリージョンにアクセスさせるには？

Google Cloud

利用者へのレスポンスを第一に
- Ingress for Anthos -

特徴

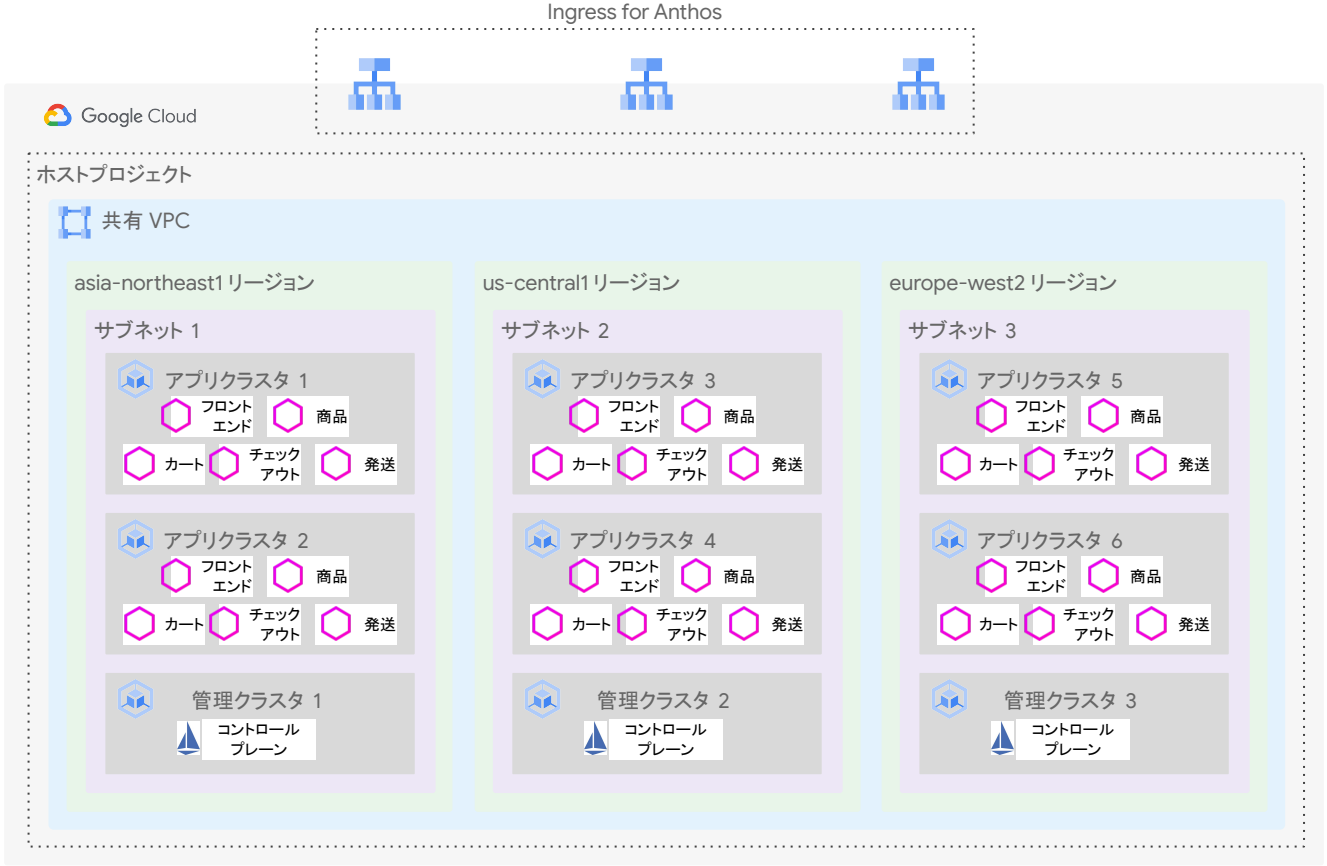
- 利用者に近いところにルーティング
- 単一かつ静的な仮想 IP
- マルチクラスタサポート
- ヘルスチェック、
  フェイルオーバーのサポート



Google Cloud

プラクティス適用後 GCP アーキテクチャ

- 全サービスに Istio を適用
- Ingress for Anthos で
  インターネットからの
  アクセスを受け付け
- Istio のコントロール
  プレーンは同リージョンのクラ
  スタを管理
- アプリクラスタとは別に
  クラスタ管理者用の
  プロジェクト
  クラスタ 1 - 3 を作成



※ サービスプロジェクトは省いています

Google Cloud

プラクティス適用後 GCP アーキテクチャ
- データベースはどうする？ -

Cloud Spanner

- スケーラブルで可用性の高い
  ミッションクリティカル RDBMS
- 要件に応じた選択肢
  - ゾーン間可用性（リージョン配置）
  - リージョン間可用性（マルチ リージョン）
    - 3 大陸構成：Iowa, Belgium, Taiwan
    - 1 大陸構成：Tokyo, Osaka など
    - etc...

Google Cloud

宣言型による定義
単一の仕組みで管理

Google Cloud

宣言型による定義

宣言型とは？

- あるべき状態を記述
  - そこに至る手続きはシステム（ツール、ライブラリなど）が担当
- 定義と状態を分離
  - あるべき状態のみを定義するため、現在の状態は関係ない
- 宣言型で定義を行うソフトウェアの例
  - Kubernetes, Istio, Terraform
  - こちらの方式が主流になりつつある
- 反対語は手続き型、命令型

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: nginx
  name: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - image: nginx
        name: nginx
```

宣言型定義の例

（Kubernetes の Deployment）

nginx のコンテナイメージを元に

レプリカの数は 3 で Deployment を

作成する

デメリット

手続き部分が隠蔽されているため、

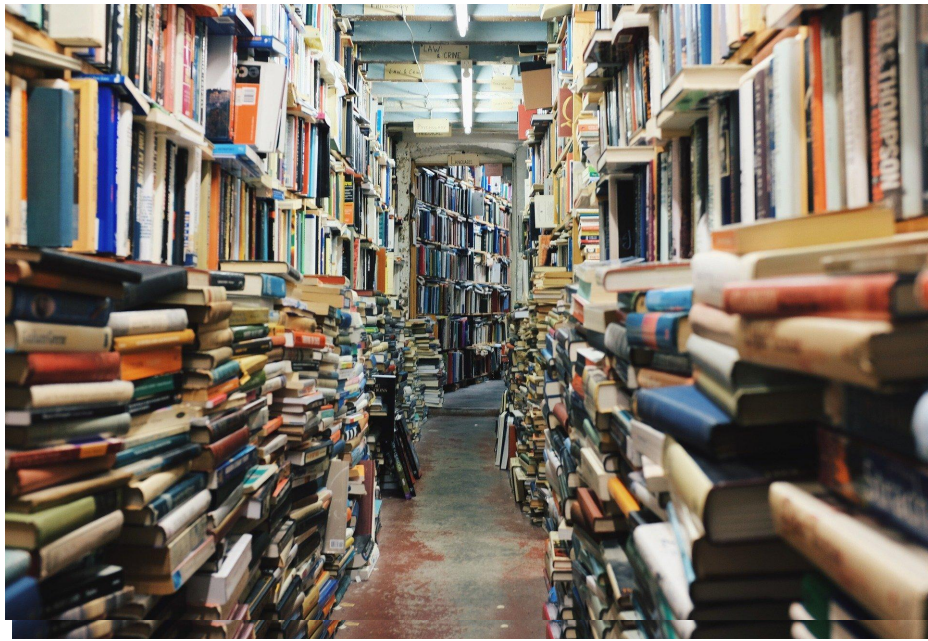
- デバッグ、問題の修正が困難
- 細かい制御が困難な場合がある

単一の仕組みで管理

コンテナ、マイクロサービス、更にアプリケーションが
グローバル展開されている世界では、管理対象が膨大になり
それらを一元化して管理する仕組みが必須となる

例、

- 設定情報
- システムログ、アプリケーションログ
- メトリック、アラート、トレース
- セキュリティ関連情報、監査情報
- UI



宣言型による定義、単一の仕組みで管理
- 推奨プラクティス on GCP -

- 全てをコードで管理
- コードを開発プロセスと同様に管理

全てをコードで管理
- Infrastructure as Code (IaC) -

コード化できるものは、できる限りコード化して管理する

設定ファイルから、運用のためのダッシュボードまで

メリット

- 手作業によるミスの低減
- デプロイ時間の短縮
- 暗黙知から形式知への変換

宣言型定義がベースのツールが増えてきたため

コード化が容易になった

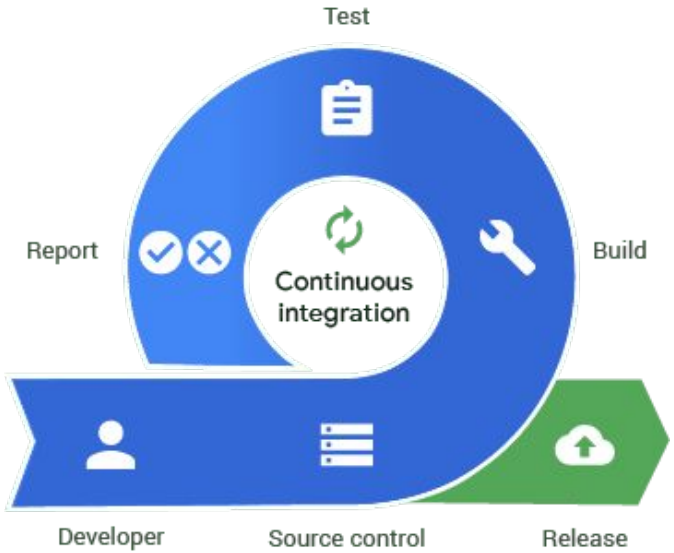
そのため、意識せずともそのようになっていることも

IaC をアプリケーションコードと同様に管理
- GitOps -

Git リポジトリに IaC のコードを格納し、アプリケーションの
ソースコードと同様のプロセスでバージョン管理する

例、

1. ローカルで作成、動作確認
2. Git へ commit
3. 自動テスト
4. Pull Request 作成
5. レビュー
6. LGTM をもらった後に merge
7. 本番に反映



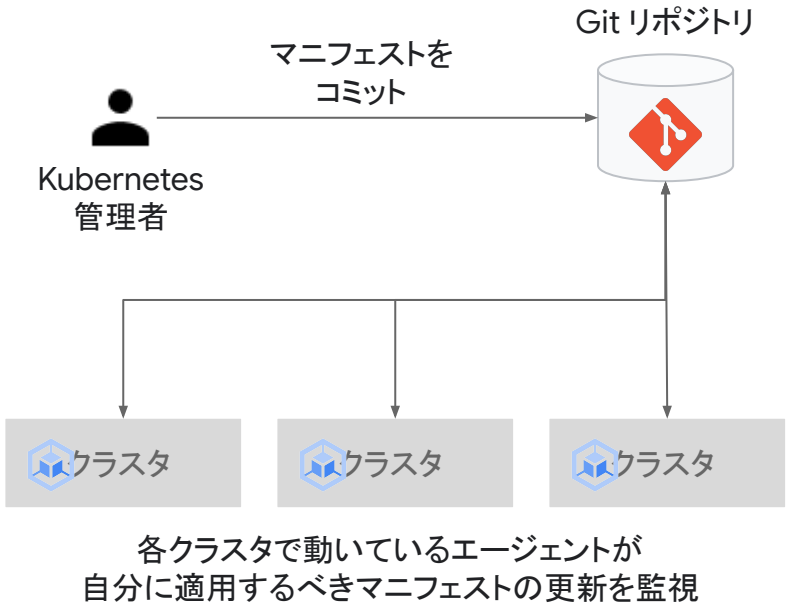
Kubernetes のマニフェストを GitOps で管理
- Anthos Config Management (ACM) -

グローバルに展開されたクラスタ群のマニフェスト

（Namespace, Quota, RoleBinding など）を一元管理、適用

特徴

- Git リポジトリに構成の定義を格納
- 構成定義の検証
- 構成定義の自動適用

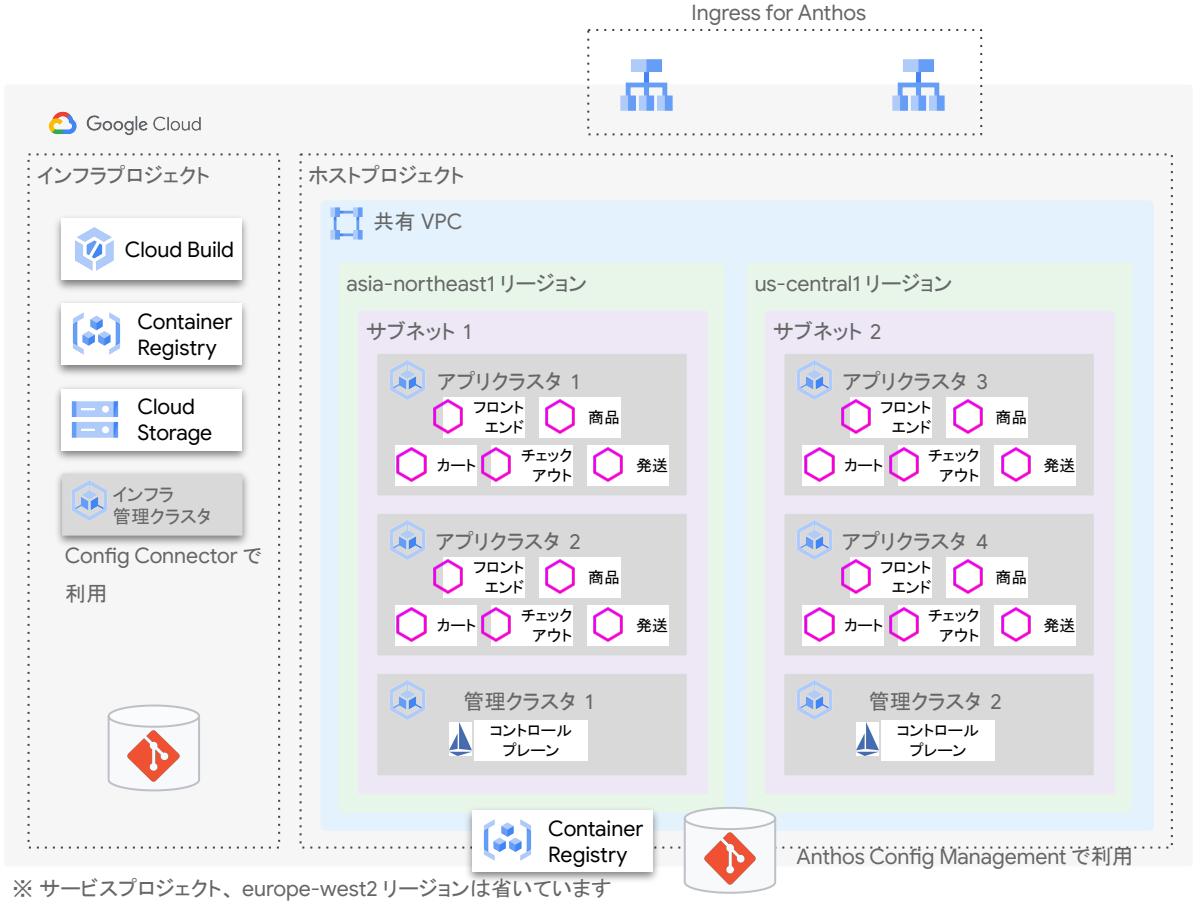


Google Cloud

プラクティス適用後 GCP アーキテクチャ

- クラスタ管理者プロジェクトに Git リポジトリを作成し、ACM を通じて全クラスタを管理する
- 基礎となる GCP リソース（VPC、プロジェクトなど）を Config Connector* または、Terraform を使って管理する インフラプロジェクトを作成

* Config Connector とは Kubernetes から GCP リソースを管理するツールで、ACM とも連携可能



※ サービスプロジェクト、europe-west2 リージョンは省いています

Google Cloud

本番導入に向けて

Google Cloud

“モダン” なアーキテクチャを本番導入する際の課題

新たなタスク、複雑性が追加される

“モダン” なアーキテクチャ採用の目的

- 市場投入までの時間を短縮
- 製品フィードバックのループを高速化
- 運用コストの削減

“モダン” なテクノロジーの採用により

- 管理対象の増加
- 更新への追従対応増加
- 問題発生時の対応増加

※ モダンなテクノロジーは進化が早い

では、どうするべきか？

マネージド サービスの活用

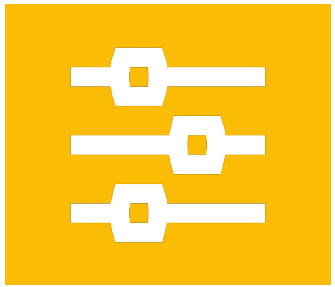
マネージド サービスを最大限活用し
”モダン” なテクノロジーの良いとこ取り

Google によるテスト

事前にテスト済みの
バージョンを利用できる

Google による運用

Google の SRE チームが
管理を行う



コントロールプレーンの
自動スケーリング

利用状況によりコントロール
プレーンを自動スケーリング

マネージド サービスを推し進めた Anthos

コンテナ オーケストレーション

Kubernetes → Anthos GKE

サービスメッシュ

Istio → Anthos Service Mesh

サーバーレス

Knative → Cloud Run for Anthos

まとめ

Google Cloud

まとめ

“モダン” なアーキテクチャの採用の目的

- 市場投入までの時間を短縮
- 製品フィードバックのループを高速化
- 運用コストの削減

“モダン” なアーキテクチャとは？

- 可用性
- 性能・拡張性
- 運用保守性
- セキュリティ

Google が考えるアプリケーションの未来

- コンテナ化、マイクロサービス
- 宣言型による定義、一元管理
- サービスメッシュを利用し、
  あらゆるロケーションへ展開

まとめ

GCP で ”モダン” なアーキテクチャを
実現するためには様々なプラクティスが存在

“モダン” なテクノロジーが入ってきたことにより

- 管理対象の増加
- 更新への追従対応増加
- 問題発生時の対応増加

マネージド サービスの活用により新たな作業、運用の負荷を抑え
“モダン” なテクノロジーの**メリットを享受**

Google Cloud

Thank you