令和6年度文部科学省委託「専門職業人材の最新技能アップデートのための専修学校リカレント教育推進」事業

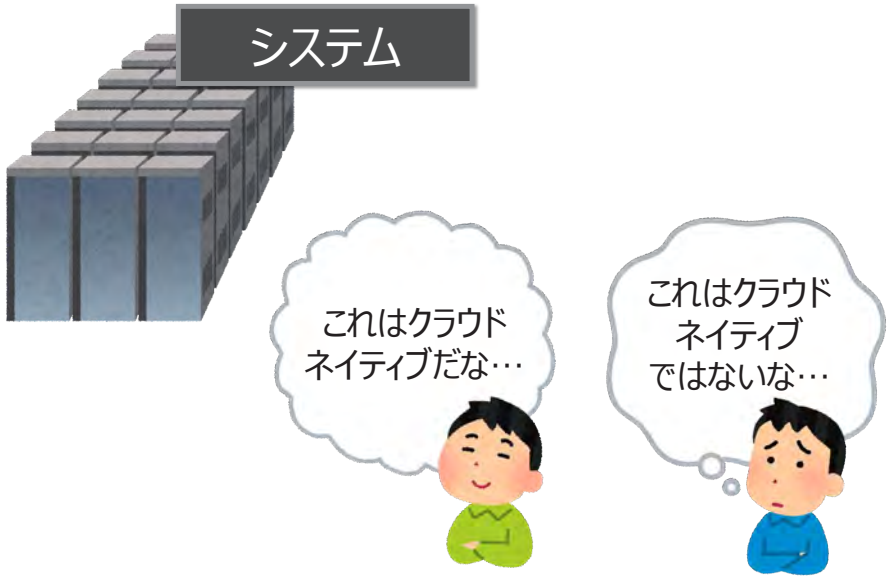
クラウドネイティブ概論教材資料

目次

第一章 クラウドネイティブの基本

クラウドネイティブとは？

- クラウドネイティブは、ITシステムを実装する際の**考え方**の一つです。
- ITシステムを実装する際に、「最初からクラウド上で実行することを前提に、設計・開発・構築を行う」という考え方です。
- 「考え方」なので
  - 実際の技術や製品、プログラミング言語などを指すものではない
  - 人によってその具体的な内容や判断基準が異なることもある



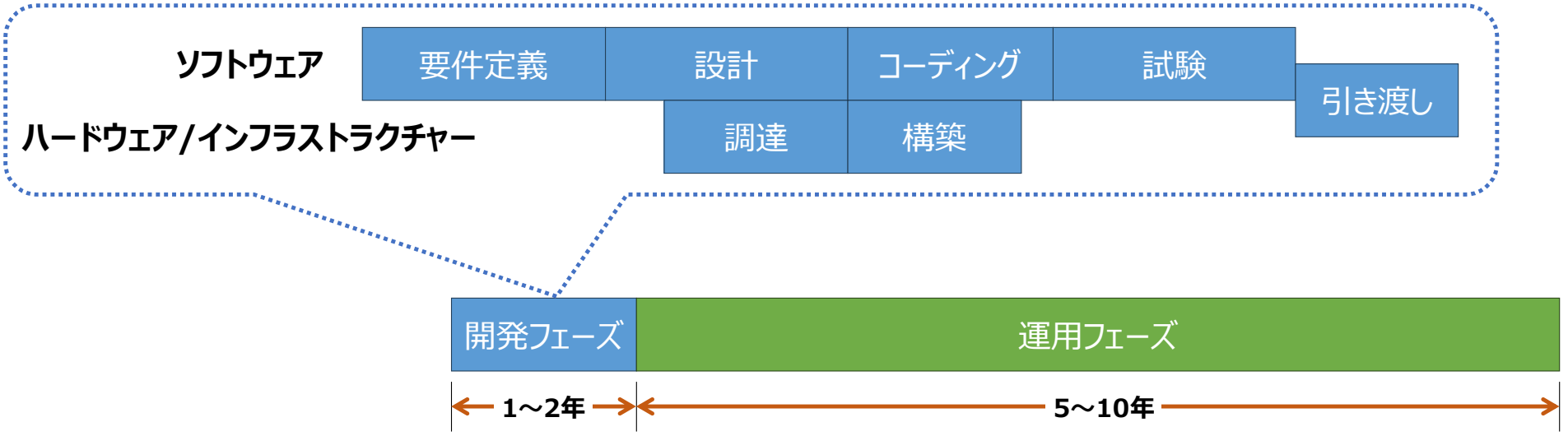
クラウドネイティブに含まれる内容

- クラウドネイティブという概念に含まれる内容は人によって異なるかもしれませんが、以下の表は、クラウドネイティブと考えられる主な技術を示しています。

	従来の考え方	クラウドネイティブ
アプリケーションの稼働場所	オンプレミス	クラウド
アプリケーションのアーキテクチャ	モノリシック	マイクロサービス
アプリケーションのモジュール間連携	内部API	サービスメッシュ
アプリケーションの開発モデル	ウォーターフォール	アジャイル
アプリケーションが稼働する基盤	物理サーバー、仮想マシン	仮想マシン、コンテナ、サーバーレス
基盤のオーケストレーション	なし	コンテナオーケストレーションツール（Kubernetesなど）
開発と運用の関係	開発と運用のサイロ化	DevOps、DevSecOps
テストとデプロイメント	計画書に従って手作業	CI/CD
インフラの構築と運用	手順書ベース	IaC、Immutable Infrastructure
オブザーバビリティ	ルールベースの監視	動的で探索的な監視

なぜクラウドネイティブ？

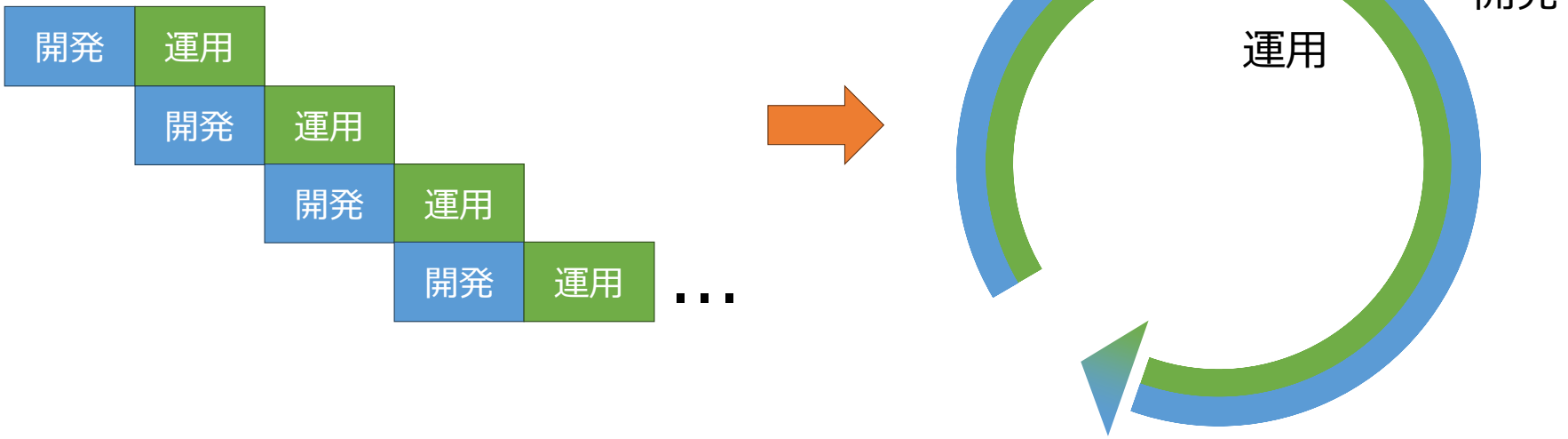
■ クラウドネイティブの必要性を理解するために、従来型のアプリケーション開発の考え方を見てみましょう。



従来型アプリケーションのライフサイクルの一例

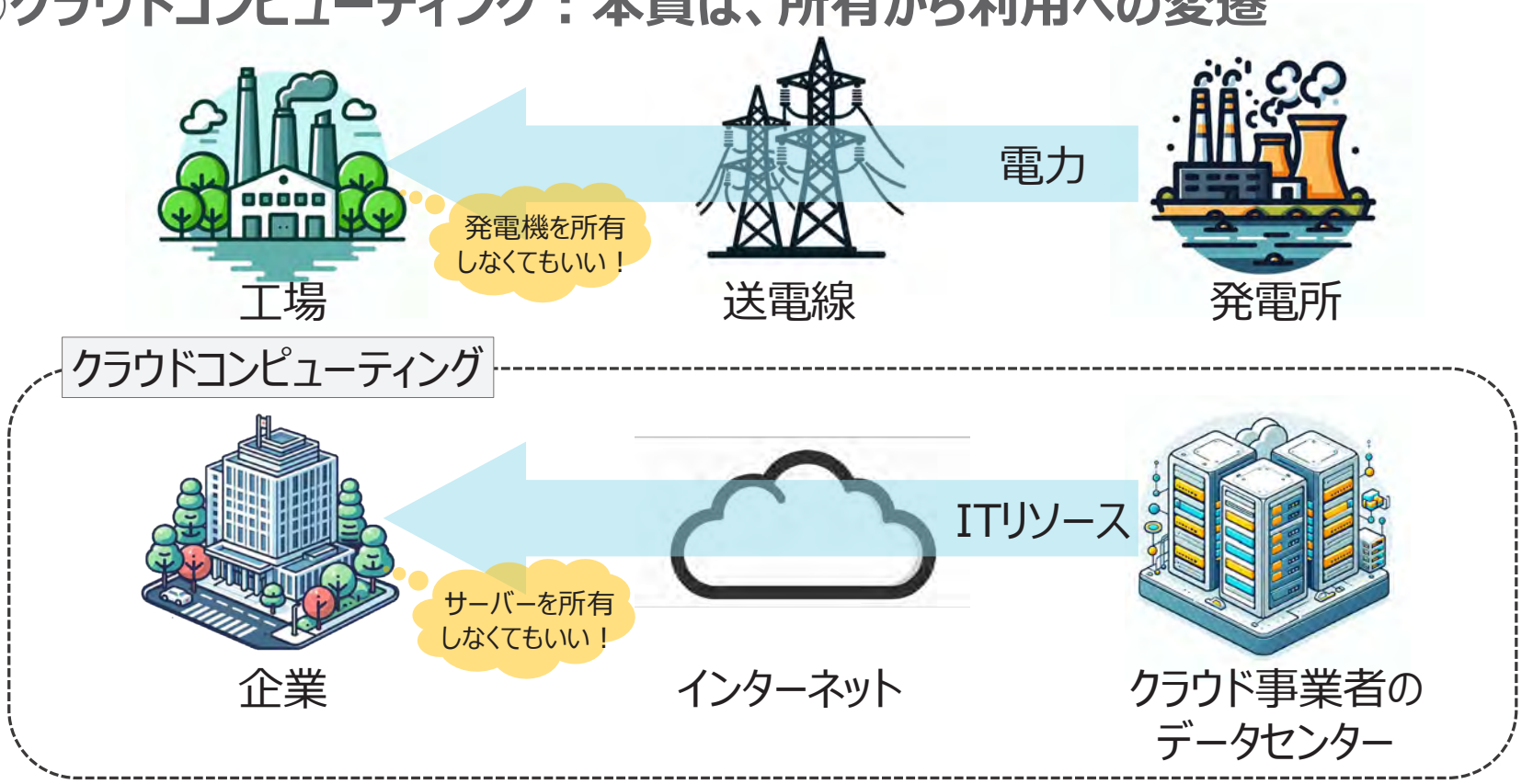
なぜクラウドネイティブ？

■ 理想的なアプリケーションのライフサイクル



クラウドネイティブの要素を見てみよう

■ ①クラウドコンピューティング：本質は、所有から利用への変遷



クラウドネイティブの要素を見てみよう

①クラウドコンピューティング：特徴

- 弾力性：
  迅速かつ柔軟にシステムの拡張・縮小を行える
- マルチテナント：
  複数のユーザが同じリソースをセキュアに共有できる
- セルフサービス：
  ユーザ自身でインターネットなどを経由して、オンデマンドでリソースの増減ができる
- 従量課金：
  利用した分だけ料金が発生する
- リモートでの利用：
  インターネットなどを通じてアクセスできる

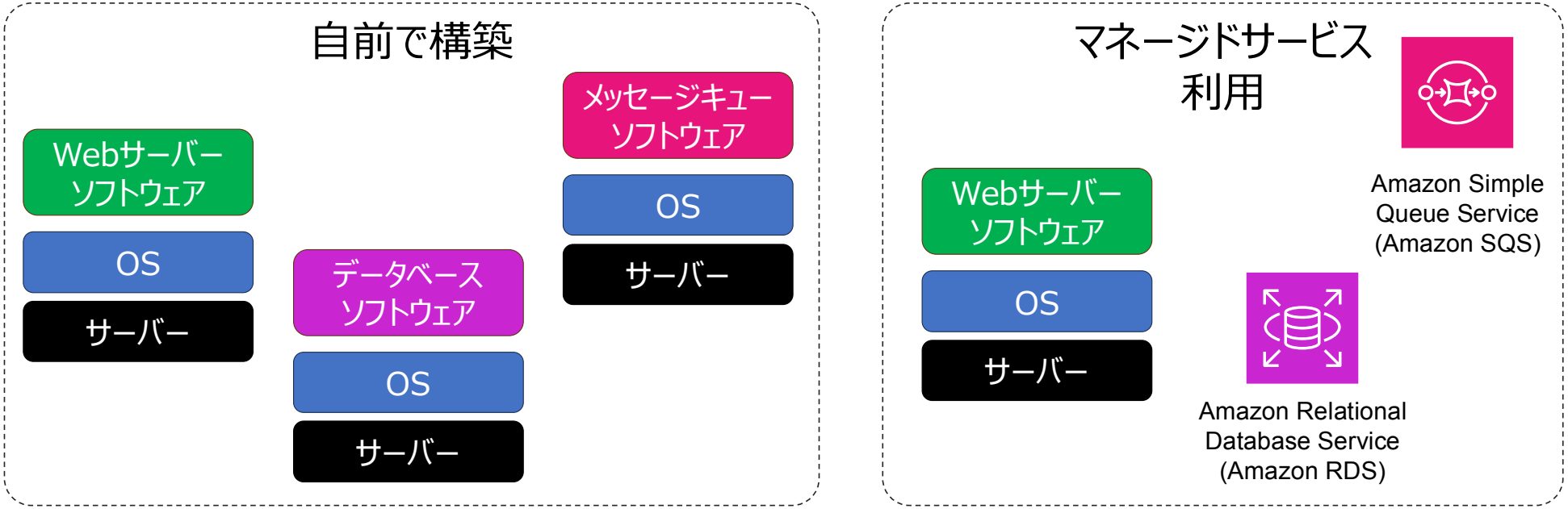
クラウドネイティブの要素を見てみよう

①クラウドコンピューティング：サービスモデル

クラウドネイティブの要素を見てみよう

①クラウドコンピューティング：マネージドサービス

- クラウドの大きなメリットの一つは、数多くのマネージドサービスを利用できる点です。



クラウドネイティブの要素を見てみよう

■ ②アーキテクチャ：モノリシックとマイクロサービス

モノリシック（monolithic）
アーキテクチャ

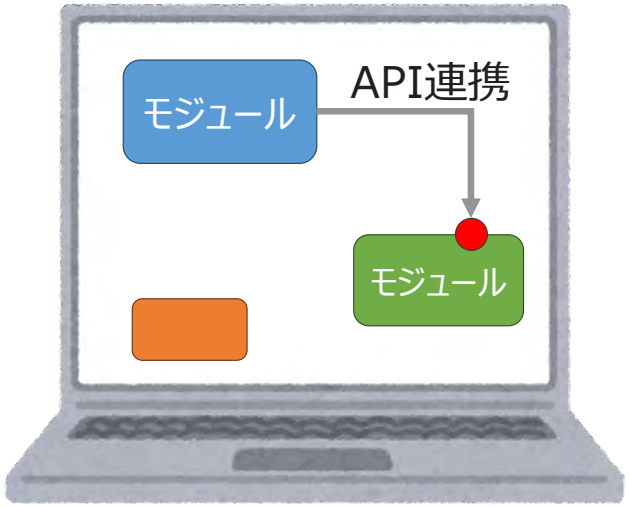
密結合

マイクロサービス
アーキテクチャ

疎結合

クラウドネイティブの要素を見てみよう

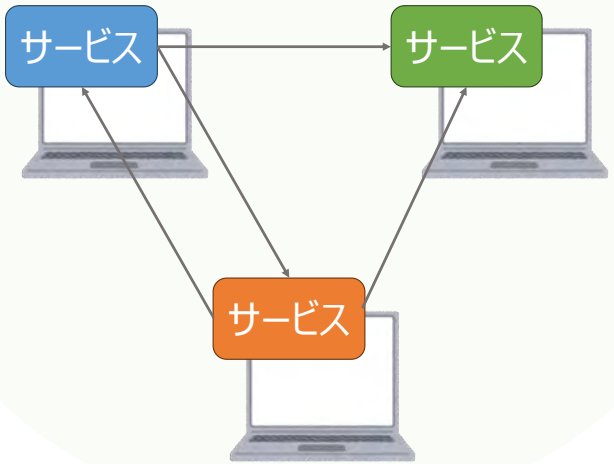
■ ③モジュール間連携



モノリシックの場合

サービスメッシュ

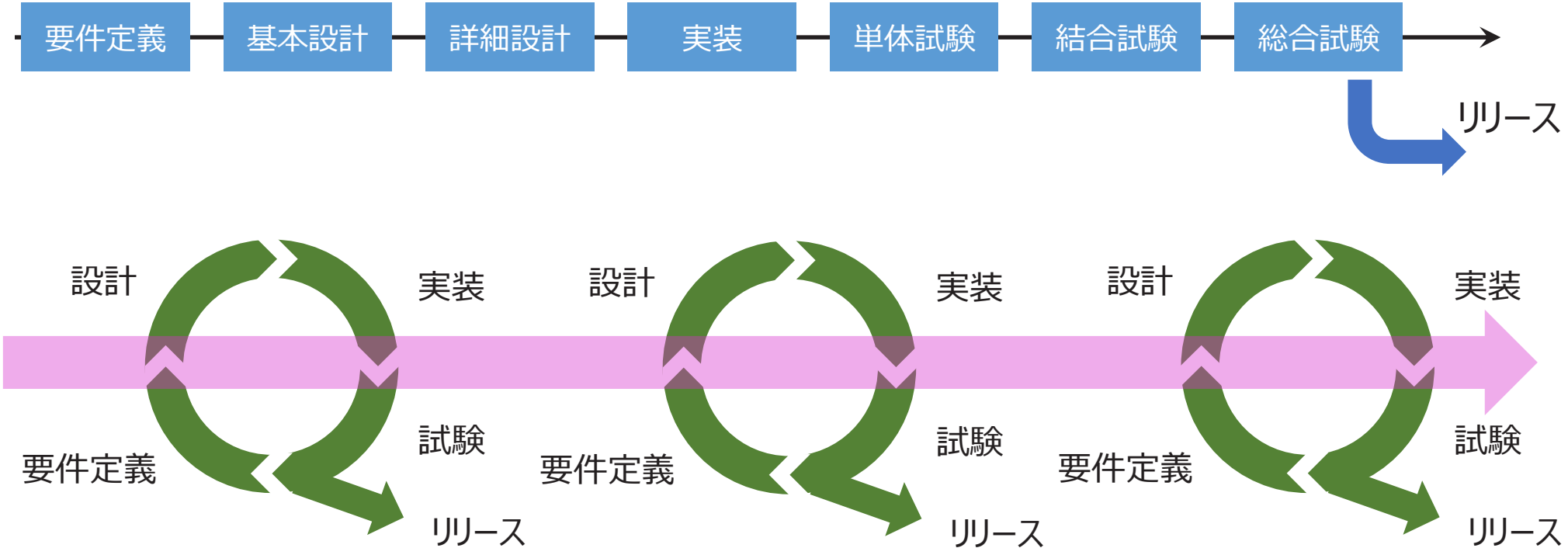
- 信頼性の高い通信経路の確立
- 通信の標準化
- 通信の監視と集計
- 通信の暗号化
- …



マイクロサービスの場合

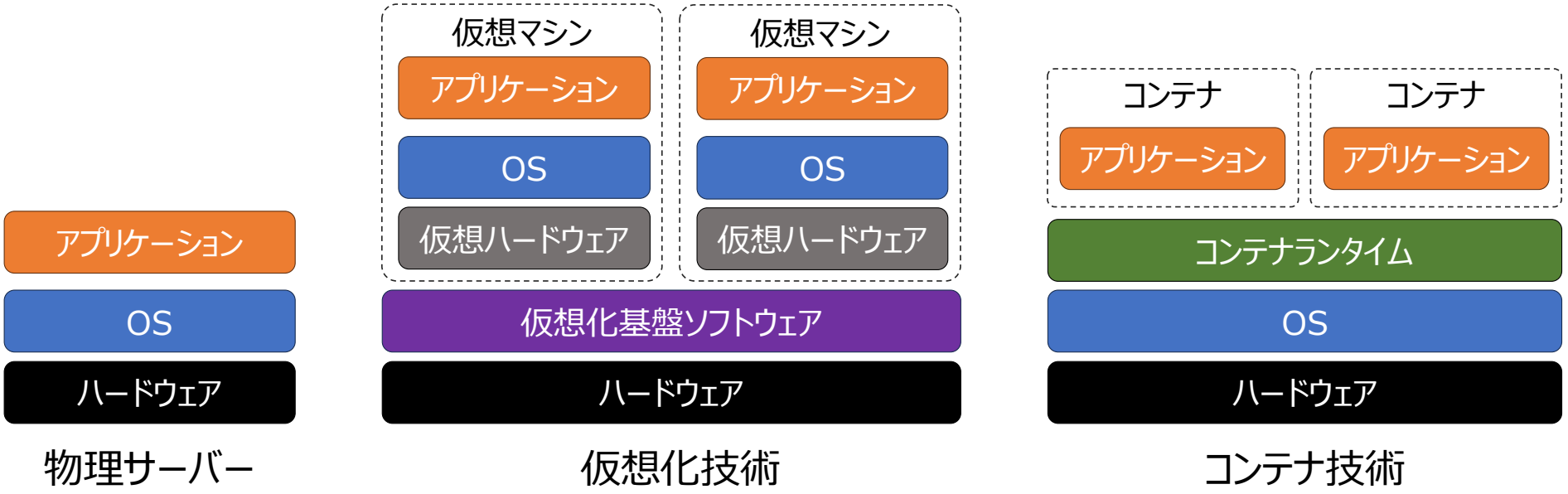
クラウドネイティブの要素を見てみよう

■ ④開発モデル：ウォーターフォールとアジャイル



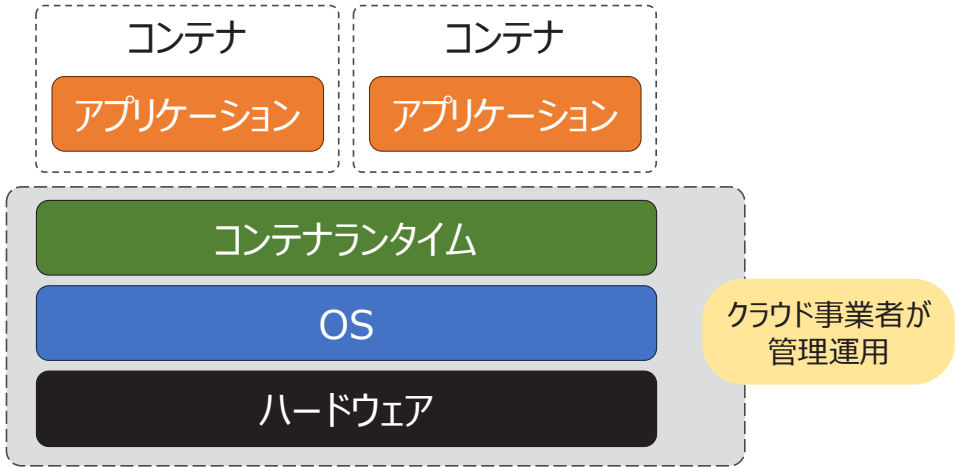
クラウドネイティブの要素を見てみよう

■ ⑤基盤（インフラストラクチャ）：仮想マシンとコンテナ

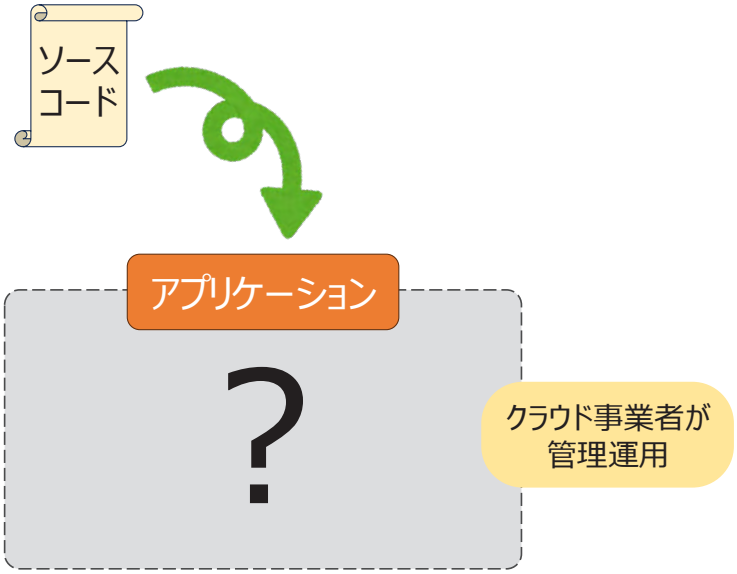


クラウドネイティブの要素を見てみよう

⑤基盤（インフラストラクチャ）：サーバーレス



コンテナにおける「サーバーレス」



FaaS

クラウドネイティブの要素を見てみよう

⑥オーケストレーション

Kubernetesの主な機能

- コンテナオーケストレーション（調整・管理）
- アプリケーションデプロイ
- ネットワーク管理
- ストレージ管理
- クラスタ管理
- リソース監視と制御
- 構成管理
- ジョブとタスク管理

クラウドネイティブの要素を見てみよう

■ ⑦開発と運用の関係：DevOps/DevSecOps

従来の
開発チームと運用チーム

DevOpsでの
開発チームと運用チーム

クラウドネイティブの要素を見てみよう

⑧テストとデプロイメント：CI/CD

- DevOpsが文化であれば、CI/CDはDevOps文化を可能にする技術の1つです。

CI (Continuous Integration)＝継続的インテグレーション

≒テストの自動化

CD (Continuous Delivery)＝継続的デリバリー

*CD (Continuous Deployment)＝継続的デプロイメント

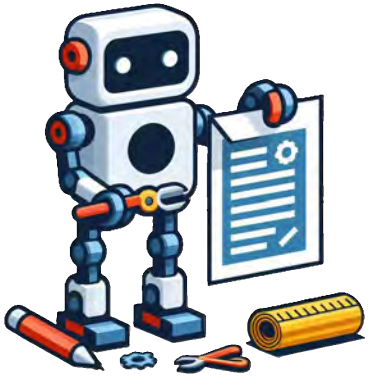
≒デプロイの自動化

クラウドネイティブの要素を見てみよう

⑨インフラの構築と運用：IaC

- IaCとはInfrastructure as Codeの略で、構築する内容を手順書ではなく、プログラムコードとして記述する方法です。

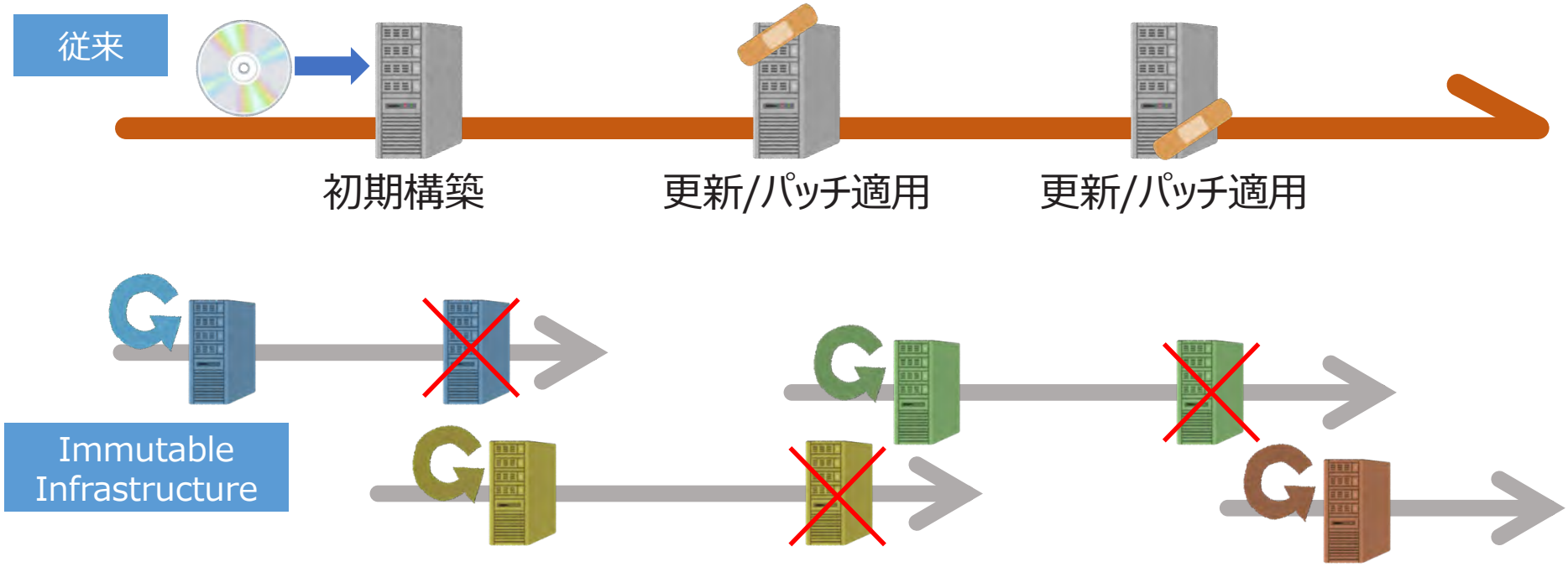
手順書ベースの構築



IaCによる構築の自動化
（イメージ）

クラウドネイティブの要素を見てみよう

■ ⑨インフラの構築と運用：Immutable Infrastructure



クラウドネイティブの要素を見てみよう

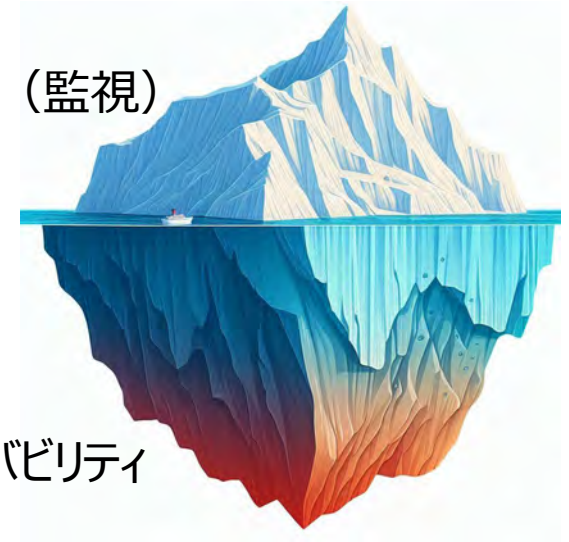
⑩オブザーバビリティ

モニタリング（監視）との違い

- モニタリング（監視）：
  when（いつ）、what（何が）発生した
- オブザーバビリティ：
  why（なぜ）、how（どのように）発生したか
  （動的で探索的な監視）

オブザーバビリティで利用する情報

メトリクス　ログ　トレース　イベント



CNCFによるクラウドネイティブの定義

「クラウドネイティブ」の定義

- クラウドネイティブの定義は人によって異なりますが、、2015年にLinux Foundationのプロジェクトとして創設されたCNCF（Cloud Native Computing Foundation）によるとクラウドネイティブとは次のように定義されています。

> クラウドネイティブ技術は、パブリッククラウド、プライベートクラウド、ハイブリッドクラウドなどの近代的でダイナミックな環境において、スケーラブルなアプリケーションを構築および実行するための能力を組織にもたらします。 このアプローチの代表例に、コンテナ、サービスメッシュ、マイクロサービス、イミュータブルインフラストラクチャ、および宣言型APIがあります。
>
> これらの手法により、回復性、管理力、および可観測性のある疎結合システムが実現します。 これらを堅牢な自動化と組み合わせることで、エンジニアはインパクトのある変更を最小限の労力で頻繁かつ予測どおりに行うことができます。
>
> Cloud Native Computing Foundationは、オープンソースでベンダー中立プロジェクトのエコシステムを育成・維持して、このパラダイムの採用を促進したいと考えています。 私たちは最先端のパターンを民主化し、これらのイノベーションを誰もが利用できるようにします。

出典：https://github.com/cncf/toc/blob/main/DEFINITION.md

まとめ：クラウドネイティブの基本

定義	「最初からクラウド上で実行することを前提に、設計・開発・構築を行う」という考え方（厳密な定義はない）
メリット	柔軟性：システムの変更や拡張が容易 スケーラビリティ：需要に応じてリソースを拡張または縮小可能 可用性：ダウンタイムなしで拡張または縮小可能 効率性：自動化された運用により負担を軽減
中核技術	クラウド、マイクロサービス、サービスメッシュ、コンテナ、イミュータブルインフラストラクチャ、CI/CD、DevOpsなど

■ **レガシーアプリケーションをそのままクラウドに移行しました。この場合、クラウドネイティブと呼べるのでしょうか？**

確認テスト1

Q1: クラウドネイティブについて、正しい記述を選びなさい。当てはまるものをすべて選択してください。

1. クラウドネイティブは、クラウド上のプログラミング言語のことを指す。
2. クラウドネイティブは、人によって定義が異なることがある。
3. クラウドネイティブは、クラウド時代に生まれた人を指す。
4. クラウドネイティブは、最初からクラウド上で実行することを前提に、設計・開発・構築を行うという考え方である。

Q2: 通常、クラウドネイティブの考え方に沿っているとされるものは、次のうちどれか？当てはまるものをすべて選択してください。

1. コンテナおよびコンテナオーケストレーションを利用する。
2. 自動化せず、運用手順書を作成して、手順書に基づいて運用する。
3. マイクロサービスアーキテクチャを採用してシステムを設計する。
4. 動的かつ探索的な監視を実装する。

確認テスト2

Q3: モノリシックなシステムにおいて、コンポーネント間はどのように結合されているか？次の中から最も適切なものを選択してください。

1. 密結合
2. 疎結合

Q4: 「コンテナのオーケストレーション」におけるオーケストレーションとは、何を指すか？次の中から最も適切なものを選択してください。

1. 自動作成
2. 監視
3. セキュリティ
4. 調整・管理

確認テスト3

Q5: DevOpsが解決しようとしているのは、次のうちどのチーム間の対立関係か？最も適切なものを選択してください。

1. 設計チームと開発チーム
2. 開発チームとテストチーム
3. 開発チームと運用チーム
4. プロジェクトチームと顧客チーム

Q6: IaC（Infrastructure as Code）とは何を指すか？次の中から最も適切なものを選択してください。

1. ソフトウェア開発自動化の手法
2. ソフトウェアテスト自動化の手法
3. インフラ構築の自動化の手法
4. インフラテストの自動化の手法
5. セキュリティ監査の自動化の手法

確認テスト4

Q7: 「Immutable Infrastructure（不変のインフラストラクチャ）」について、正しい記述を選びなさい。当てはまるものをすべて選択してください。

1. 古い考え方であり、いわゆる「塩漬け」である。
2. 一度構築したシステム環境を決して変更しないという考え方。
3. IaC（Infrastructure as Code）と密接に関係する。
4. 頻繁にパッチを適用し、システムを常に最新化することを重視する。

Q8: オブザーバビリティ（可観測性）において、利用される情報は次のうちどれか？当てはまるものをすべて選択してください。

1. ログ
2. トレース
3. プロセス
4. メトリクス

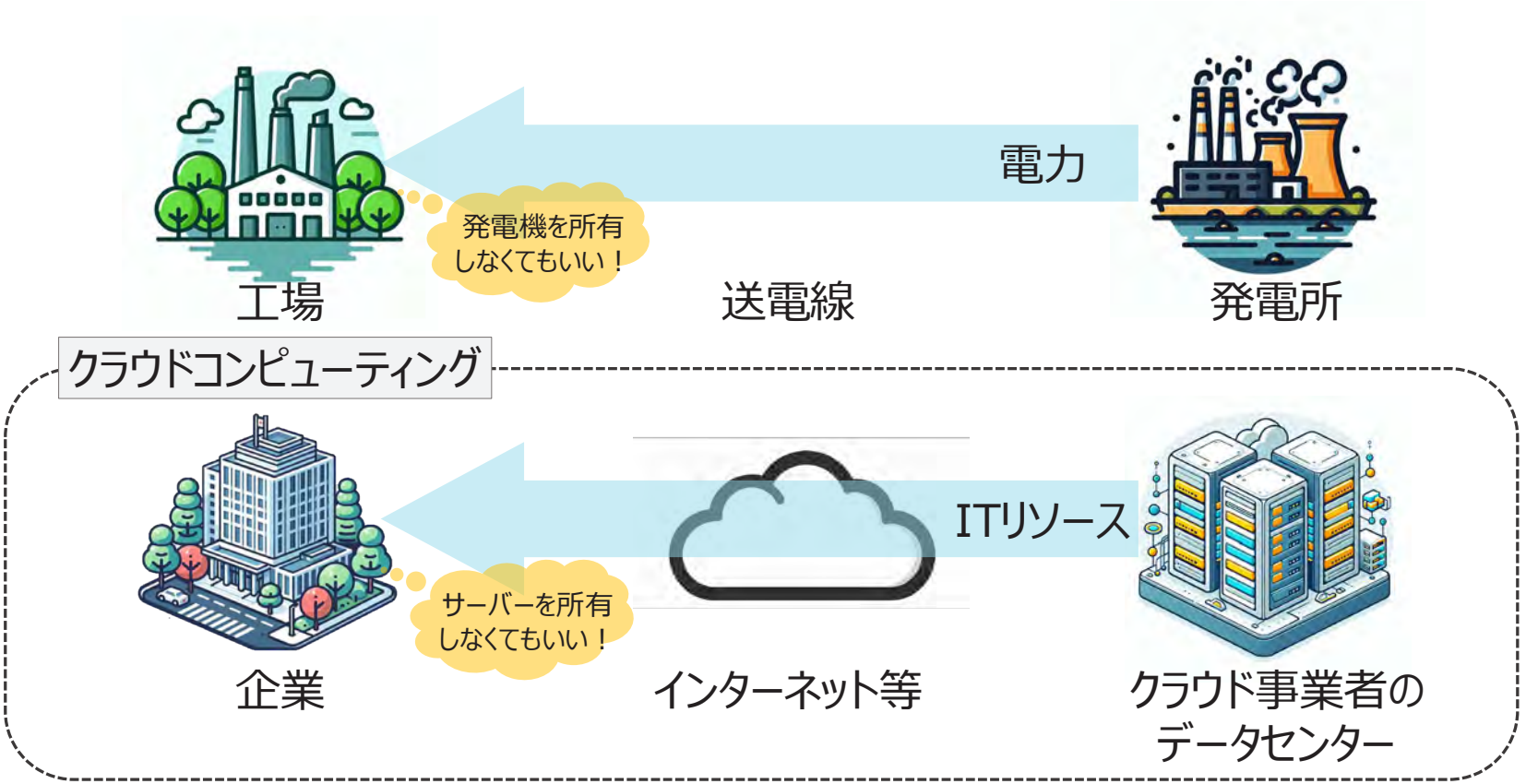
確認テスト5

Q9: CNCFのクラウドネイティブ成熟度モデルにおいて、
レベルとその内容の組み合わせが正しいものはど
れか？当てはまるものをすべて選択してください。

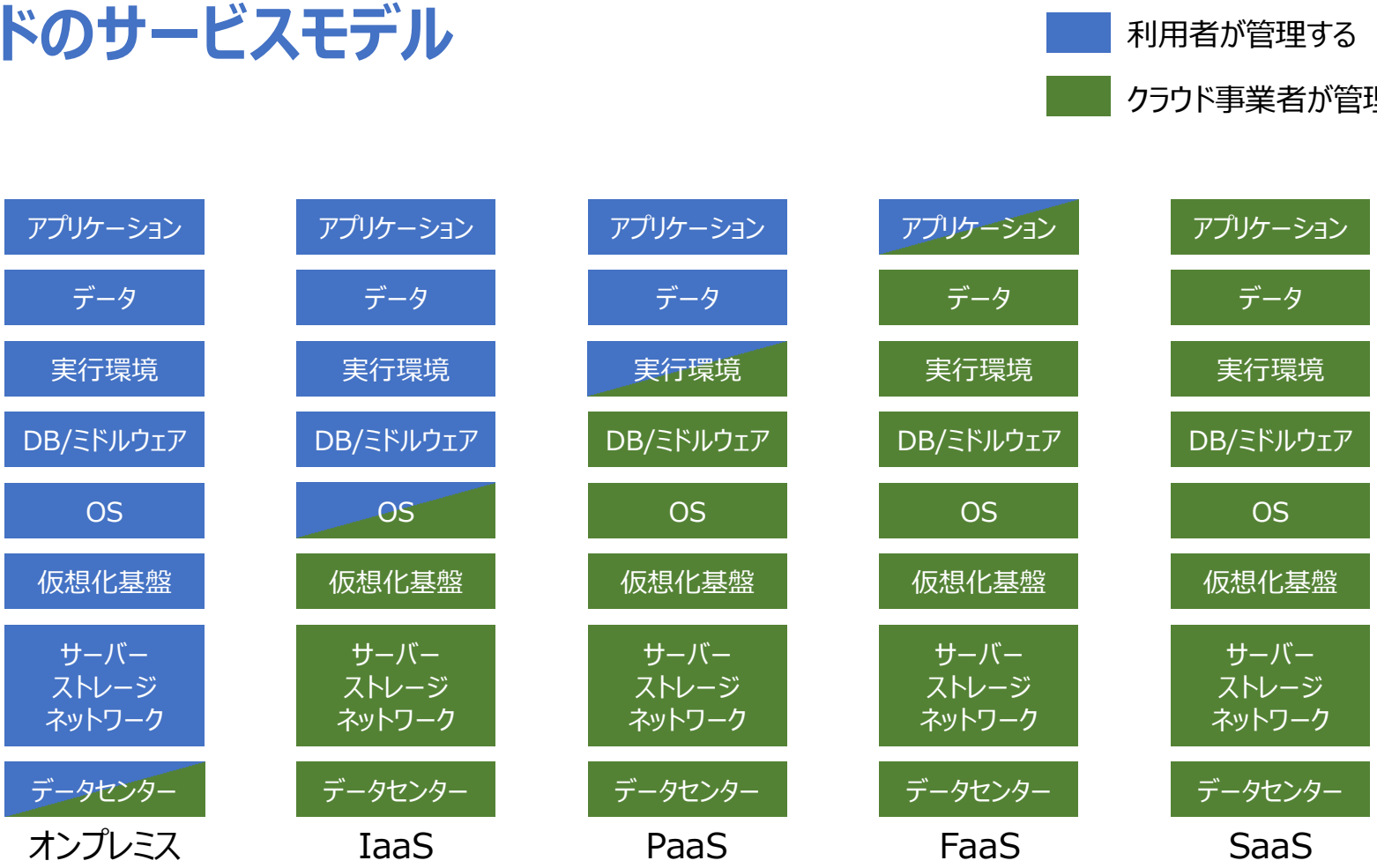
1. レベル1：検証
2. レベル2：運用
3. レベル3：構築
4. レベル4：改善
5. レベル5：終了

第二章 クラウド技術

クラウドコンピューティング



クラウドのサービスモデル



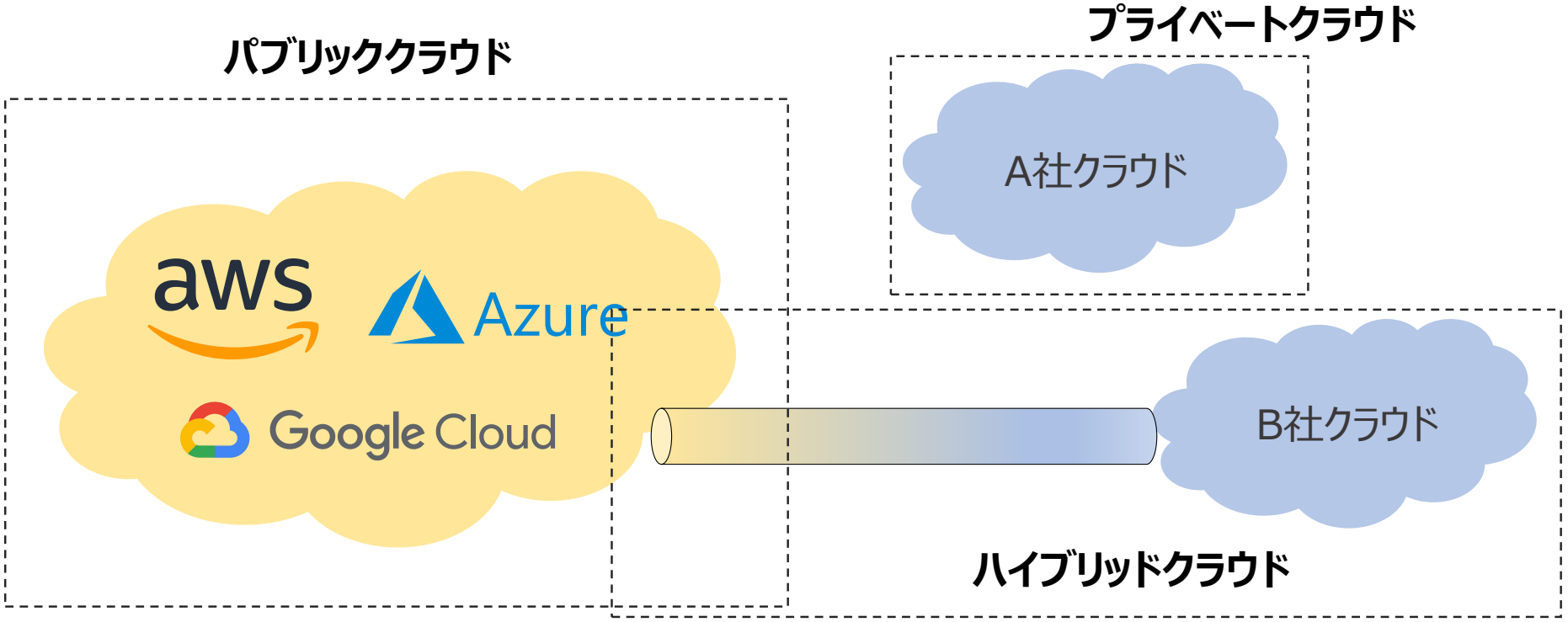
SaaS vs 他のクラウドサービスモデル

- 総務省が公表した「通信利用動向調査」（2023年版）によると、クラウドサービスを利用している企業は約8割に達しており、クラウドが非常に普及していることがうかがえます。
- しかし、その大半はOffice 365、Gmail、DropboxやOneDriveといった**SaaS**の利用に限られています。
- SaaSは、ほとんどIT技術を必要とせず簡単に導入できる一方で、営業やバックオフィスなどの定型業務に限られる場合が多いです。そのため本コースでは、主にSaaS以外の技術について解説していきます。

SaaS (Software as a Service)

クラウドのデプロイモデル

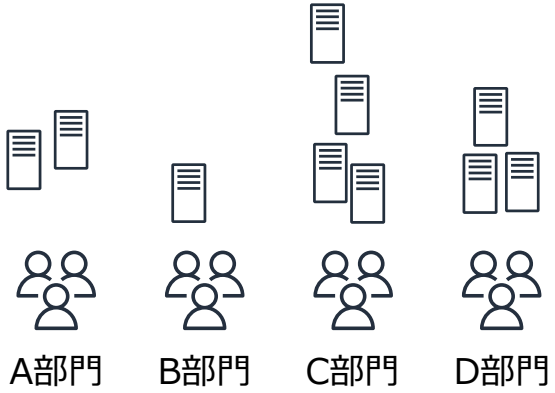
■「クラウド」は、パブリッククラウドだけではない



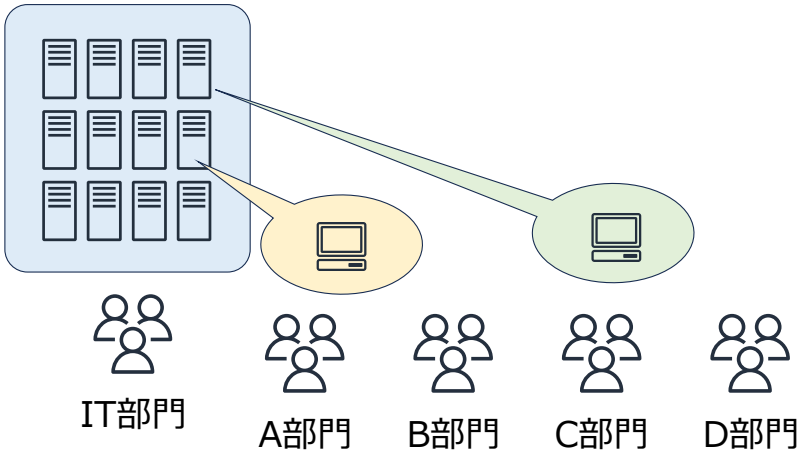
プライベートクラウドとオンプレミスの違い

どちらもインハウスのIT環境ですが、「所有」なのか「利用」なのかによって異なります

- オンプレミス：各部門がサーバーなどのIT資産を所有する
- プライベートクラウド：IT部門がサーバーなどのIT資産を所有し、リソースをプールする。各部門がオンデマンドで利用する。利用したリソースの量に応じてチャージされる



オンプレミス



プライベートクラウド

マルチクラウド

■ マルチクラウドとは、プライベートクラウドを含む複数のクラウド環境を要件や目的に合わせて使い分けることです。

- メリット：適材適所、最適化、リスク分散
- デメリット：互換性・移行性、複雑、管理コスト増

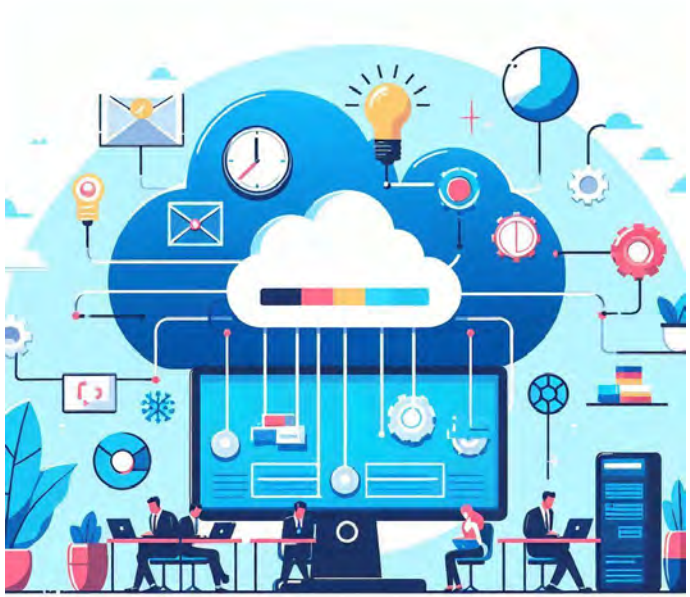
aws

Azure

自社
プライベートクラウド

パブリッククラウド

- クラウド技術の中で、もっともポピュラーなのはパブリッククラウドです。
- クラウドネイティブ開発を行うには、まずパブリッククラウドのコンポーネントやその使用方法を理解しておく必要があります。



パブリッククラウドの代表例

AWS (Amazon Web Services)	Amazonが提供するクラウドサービスで、グローバルおよび日本国内の市場でシェアNo.1を誇るパブリッククラウドです。200以上のサービスを展開しており、スタートアップから大企業まで、幅広いニーズに対応可能です。
Microsoft Azure	Microsoftが提供するクラウドサービスで、企業向けの強力な統合と幅広いサービスが特徴です。Microsoft製品（Windows Server、Active Directory、Office 365など）との高い親和性により、既存のIT環境をスムーズにクラウドへ移行できます。また、OpenAIとの戦略的パートナーシップにより、先進的なAI機能を提供しています。
GCP (Google Cloud Platform)	Googleが提供するクラウドサービスで、AI・データ解析分野における強みが際立っています。他のクラウドプロバイダーと比較して、Googleの検索技術やYouTubeの運用で培ったデータ処理能力を活かしたスケーラブルなインフラが特徴です。BigQueryなどのデータ解析サービスが有名です。
OCI (Oracle Cloud Infrastructure)	Oracleが提供するクラウドサービスで、特にエンタープライズ向けの高性能なデータベースサービスと低コストのストレージオプションが特徴です。他のクラウドプロバイダーと比較して、OCIはオンプレミス環境からの移行が容易で、既存のOracle Databaseをスムーズにクラウドへ移行できます。
Fjcloud (富士通クラウド)	富士通が提供する国産のクラウドサービスで、特に日本企業のニーズに合わせた高い信頼性とセキュリティを特徴としています。国内の堅牢なデータセンターで運用されており、データの国内保管が保証されるため、法規制やコンプライアンス要件を満たすことが可能です。
Alibaba Cloud	グローバル市場ではAWS、Azure、GCPに次いでマーケットシェア4位を誇る、中国発のクラウドサービスです。特に、中国市場へのビジネス展開を目指す企業にとっては、有力な選択肢となります。

マーケットシェアについては、statista社調査結果より引用（2024年11月1日現在）

AWS

- **2006年にサービス提供を開始したAWSは、EC大手であるAmazon社が開発したクラウドサービスです。**
  - 当初は、Amazon社内で培ったノウハウをサービスとして提供したもの
  - 2006年のリリース当時は3つのサービスのみだったが、現在では200種類以上のサービスを提供
  - 年間3,000回以上のバージョンアップや機能改修を行っている
  - 2024年11月現在245の国と地域でサービスを提供

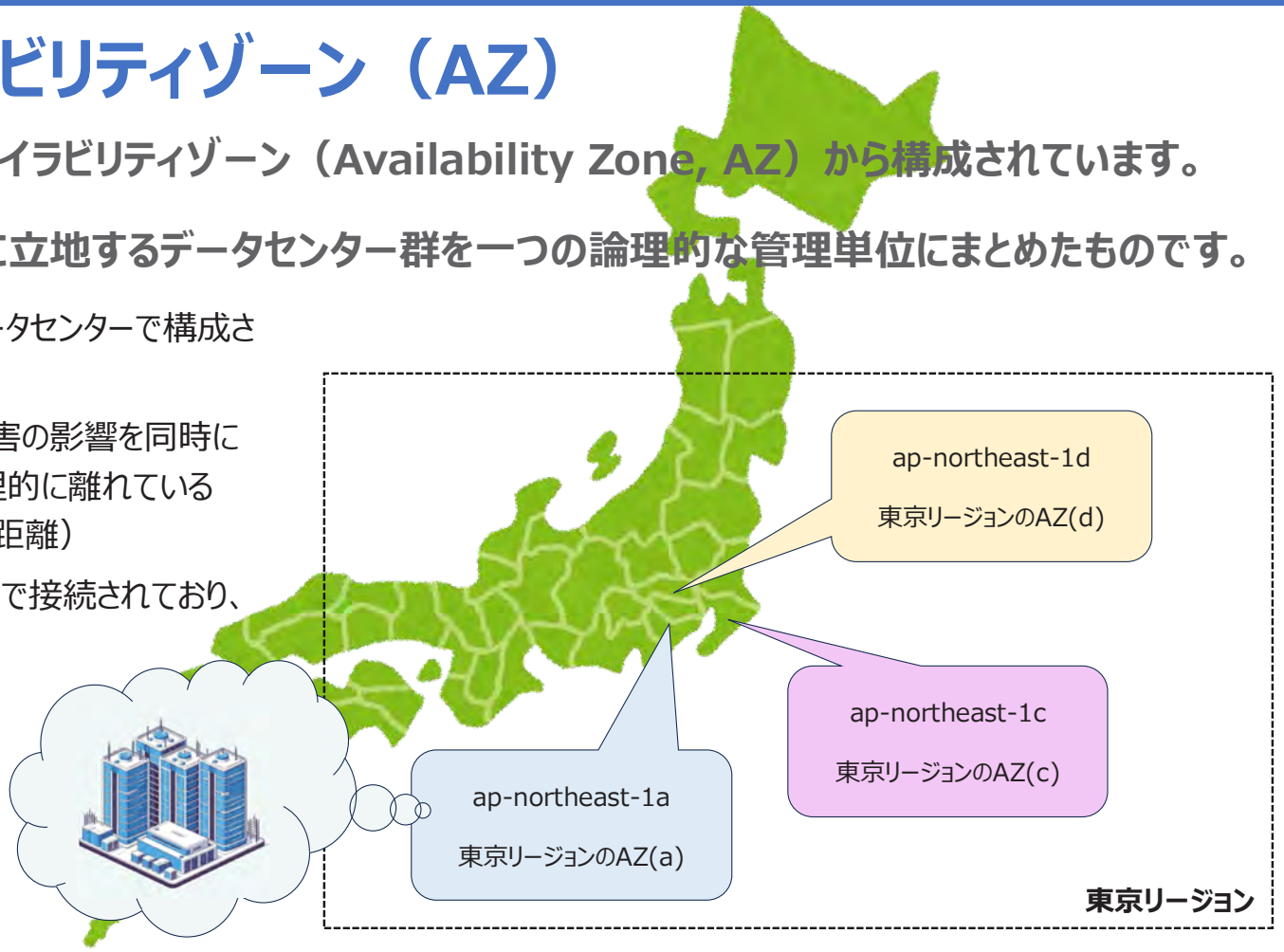
AWSのリージョン

- **AWSを利用する際に、どの「リージョン」を利用するかを最初に決めます。リージョンとは、地理的なエリアのことで、そのエリアにあるAWSの複数施設から構成されています。**
  - 例えば、「東京リージョン」や「大阪リージョン」、「シンガポールリージョン」、「バージニア北部リージョン」など
  - それぞれのリージョンに、コードが振られている。例えば東京リージョンのコードは、「ap-northeast-1」。大阪リージョンのコードは、「ap-northeast-3」
  - 各リージョンは完全に独立しており、データの遅延や法的要件、災害対策に対応するために設計されている

AWS社Webサイトより引用

AWSのアベイラビリティゾーン（AZ）

- リージョンは、複数のアベイラビリティゾーン（Availability Zone, AZ）から構成されています。
- 一つのAZは、ある地域に立地するデータセンター群を一つの論理的な管理単位にまとめたものです。
  - AZは、1つ以上の物理データセンターで構成されている
  - 火災や停電などの広域災害の影響を同時に受けないよう、AZ間は物理的に離れている（最大でも100㎞程度の距離）
  - AZ間は超高速ネットワークで接続されており、遅延は1桁ミリ秒以内



AWSの主なサービス（インフラ）

AWSの主なサービス（アプリケーション）

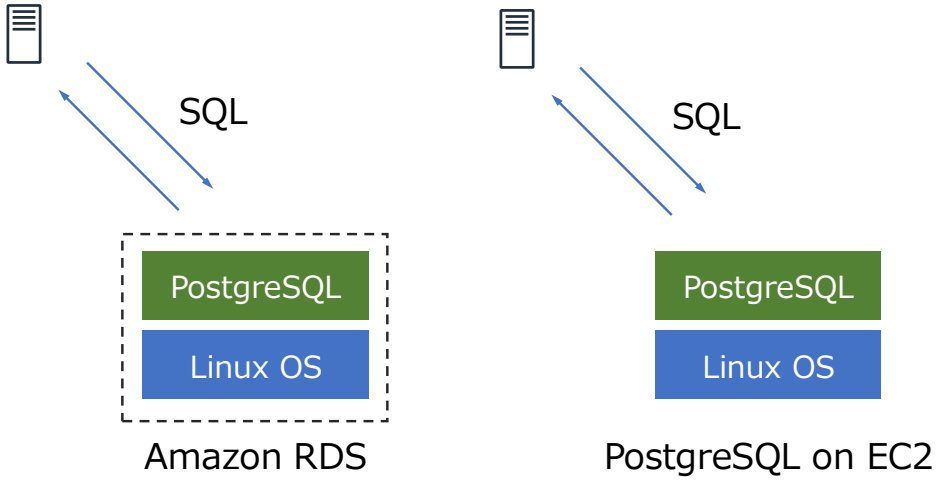
開発プロセス/開発支援	アプリケーション統合	フロントエンド開発	特定領域（IoT/AIなど）
AWS CodeCommit	Amazon Simple Notification Service (Amazon SNS)	AWS Amplify	AWS IoT Core
AWS CodeBuild	Amazon Simple Queue Service (Amazon SQS)	Amazon API Gateway	AWS IoT Greengrass
AWS CodeDeploy	Amazon EventBridge	AWS AppSync	Amazon SageMaker
AWS CodePipeline	AWS Step Functions		Amazon Bedrock
AWS X-Ray			

AWSのマネージドサービス

- AWSでは、データベースなどのソフトウェアが、あらかじめセットアップされ、運用管理が自動化された形でサービスとして提供されています。これらは「マネージドサービス」と呼ばれます。
- マネージドサービスを利用することで、導入がシンプルになるだけでなく、スケーリングやパッチ適用、障害復旧といった運用管理の多くをAWSに任せることができます。

導入手順の比較（例）

	Amazon RDS	PostgresSQL on EC2
1	RDSインスタンスを起動する	EC2インスタンスを起動する
2	-	OSをアップデートする
3	-	PostgreSQLをインストールする
4	-	PostgreSQLの初期設定を行う
5	ネットワーク接続を設定する	ネットワーク接続を設定する
6	DBとして利用する	DBとして利用する



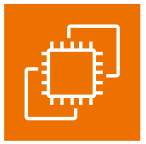
「マネージドサービス」の定義

- AWSの「マネージドサービス」には、実は3つの異なる意味があります。
- 前頁で紹介したのは、表の一番上にある「マネージドサービス」です。

「マネージドサービス（Managed Server）」	インフラストラクチャをSaaSのように利用できるサービスです。 例えば、Amazon RDSというサービスは、MySQLやPostgreSQLなどのデータベースを、セットアップ済みの状態で提供するサービスです。RDSを利用すれば、構築や設定が不要になり、インスタンスを起動するだけでSQLクライアントからSQL文を発行できるようになります。 また、Amazon EFSは構築済みのNFSファイルサーバーを提供するサービスで、こちらもマネージドサービスに該当します。
AWS Managed Service（AMS）	AWSが提供する、お客様に代わりAWSを運用するサービスです。
MSP（Managed Service Provider）	こちらも「マネージドサービス」という言葉が使われますが、原則として「MSP」と呼ばれます。MSPは、AWS社以外の第三者が提供するAWSの運用サービスを指します。 国内では、ソフトバンク社やアイレット社など、多くのMSPが存在します。

デモンストレーション：EC2インスタンスの起動

- AWSで最も利用頻度の高いサービスの一つに「EC2」があります。皆さんも一度は耳にしたことがあるのではないでしょうか？
- EC2では、サーバーに相当するものを「インスタンス」と呼びます。つまり、EC2インスタンスはクラウド上の仮想サーバーです。わずか数秒で、OSがあらかじめインストールされた状態のEC2インスタンスを起動することが可能です。
- それでは、実際のAWS環境でEC2インスタンスを起動するデモンストレーション動画を見てみましょう。どれほど簡単に仮想サーバーを立ち上げられるかがわかります。



	物理サーバ	EC2インスタンス
1	購入するサーバーのスペックを確定し、注文書発行	EC2インスタンスの起動画面で、必要なスペックやOSを選択し、「起動」ボタンをクリックする
2	（2～3か月のリードタイム）	（数秒待ち）
3	搬入、設置、結線作業（数日）	
4	電源投入。OSインストール、ネットワーク接続、パッチ適用（1～2日）	
5	利用可能になる	利用可能になる

第二章 クラウド技術


ap-southeast-2.console.aws.amazon.com
aws
検索
[オプション+S]
シドニー
education
コンソールのホーム 情報
デフォルトレイアウトにリセット
+ ウィジェットを追加
最近アクセスしたサービス 情報
最近アクセスしたサービスなし
頻繁に利用されているこれらの AWS サービスのいずれかについて詳しく見てみましょう。
EC2 S3 RDS Lambda
すべてのサービスを表示
アプリケーション (0) 情報
アプリケーションを作成
リージョン: Asia Pacific (Sydney)
ap-southeast-2 (現在のリージョン)
アプリケーションの検索
名前
説明
リー...
送信元ア..
アプリケーションはありません
アプリケーションを作成して開始しましょう。
アプリケーションを作成
myApplications に移動
AWS へようこそ
AWS の開始方法
AWS を最大限に活用するために基礎を学び、有益な情報を見つけましょう。
AWS Health 情報
未解決の問題
0
過去 7 日間
コストと使用状況 情報
CloudShell
フィードバック
© 2024, Amazon Web Services, Inc. またはその関連会社。
プライバシー
用語
Cookie の設定

確認テスト1

Q1: クラウドの本質として正しいものは次のうちどれか？最も適切なものを選択してください。

1. スタンドアローンからインターネットへ
2. 所有から利用へ
3. 内製化から外製化へ
4. Web2からWeb3へ

Q2: クラウドの特徴として当てはまらないものは、次のうちどれか？当てはまるものをすべて選択してください。

1. 弾力性
2. 従量課金
3. ネットワーク越しでの利用
4. セルフサービス
5. 高い初期投資が必要

確認テスト2

Q3: アプリケーションやデータを完全に利用者が管理し、ソースコードやコンパイルだけでなく、アプリケーションのスケーリングにも責任を持つのは、クラウドのどのサービスモデルか？当てはまるものをすべて選択してください。

1. IaaS
2. PaaS
3. FaaS
4. SaaS

Q4: プライベートクラウドとオンプレミスの関係について、正しい記述は次のうちどれか？最も適切なものを選択してください。

1. プライベートクラウドとオンプレミスは、同じものである。
2. 「弾力性」や「従量課金」などのクラウドの特徴を満たしていれば、プライベートクラウドとみなされる。
3. オンプレミスに専用のデバイスやクラウドソフトウェアを導入すればプライベートクラウドになる。
4. オンプレミスは、プライベートクラウドの形態の1つである。

確認テスト3

Q5: マルチクラウドとは何を指すか？最も適切なものを選択してください。

1. プライベートクラウドを含む複数のクラウド環境を、要件や目的に合わせて使い分けること。
2. パブリッククラウドとプライベートクラウドを組み合わせて連携させる形態。
3. 複数のプログラミング言語に対応し、要件に応じて柔軟に選択できるクラウドサービスのこと。
4. 並行処理を得意としたクラウドのこと。

Q6: パブリッククラウドについての記述の中で、正しいものはどれか？当てはまるものをすべて選択してください。

1. AWSは、EC大手であるAmazon社が開発したクラウドサービスである。
2. Azureは、Meta社（旧Facebook社）が開発したクラウドサービスである。
3. GCPは、Googleが提供するクラウドサービスで、1990年代よりサービス提供している。
4. AWSは現在、EC2やS3など200以上のサービスを提供している。

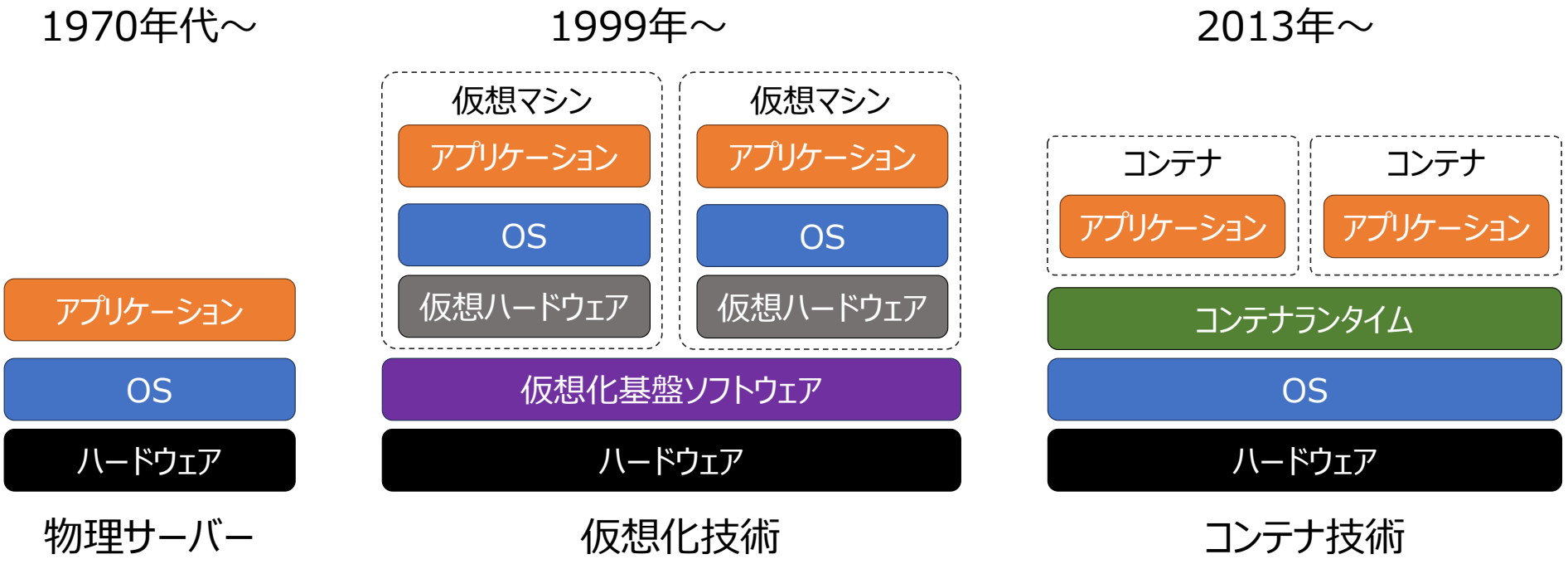
確認テスト4

Q7: AWSのマネージドサービスについての記述の中で、正しいものはどれか？（ここでいうマネージドサービスは、AMSやMSPではない）当てはまるものをすべて選択してください。

1. マネージドサービスは、すべてがAWSによって管理されるため、ユーザー側は何も考える必要がない。
2. マネージドサービスは、いわゆる運用代行サービスで、AWSに自社システムの運用を代行してもらうサービスである。
3. RDSやEFSはマネージドサービスにあたる。
4. AWSのマネージメントコンソールから操作できるサービスのこと。

第三章 コンテナ技術

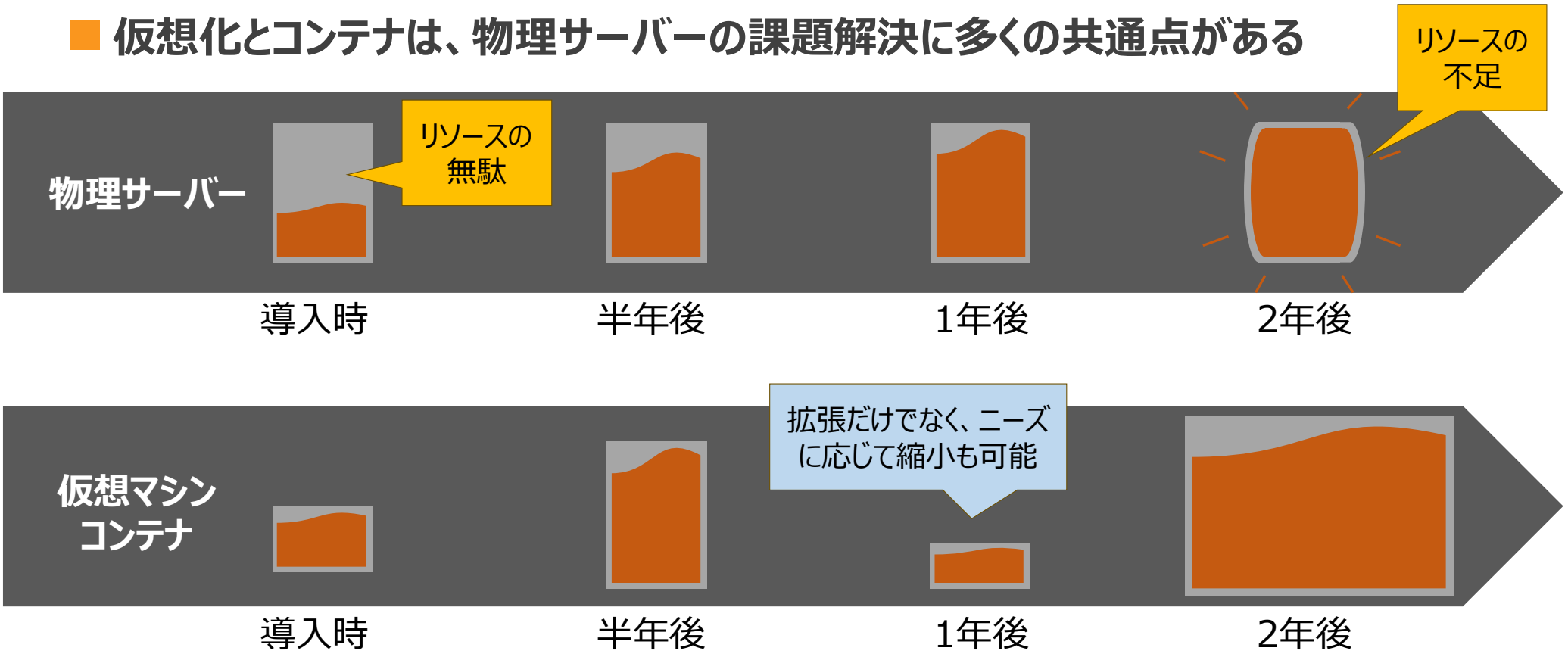
サーバーの進化形：コンテナ



＊ それぞれの技術が誕生した時期については諸説ありますが、
ここではコア技術そのものではなく、普及のきっかけとなった製品が登場した時期を基準としています。

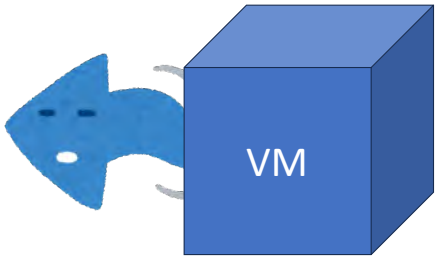
仮想化もコンテナとも、物理サーバーの課題を解決

■ 仮想化とコンテナは、物理サーバーの課題解決に多くの共通点がある



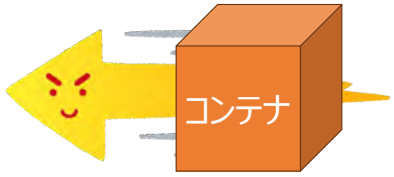
コンテナの優位性

■ 仮想マシンと比較して、コンテナの最大の特徴は「軽量」です。



仮想マシンの起動：数分～数十分

仮想マシンのイメージファイル：数GB



コンテナの起動：1秒以下～数秒

コンテナのイメージファイル：数十MB

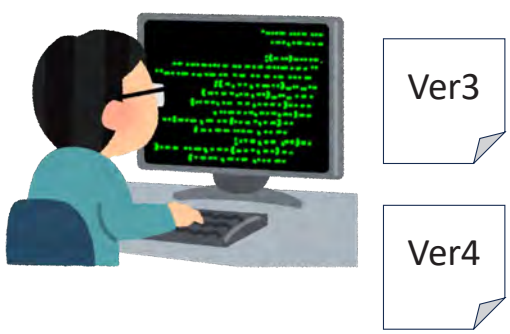
＊ どちらのも典型的な例です。仮想マシンまたはコンテナの大きさや内容によって大きく変わります。

コンテナの優位性がもたらす運用方法の変革

■「Immutable Infrastructure」（不変のインフラ）

- コンテナの場合、アプリケーションがバージョンアップの際に、動作環境を「アップグレードする」のではなく、動作環境を「作り直す」のです。
- これによって、Immutable Infrastructureを実現します。

物理サーバーと仮想マシンの場合



コンテナの場合

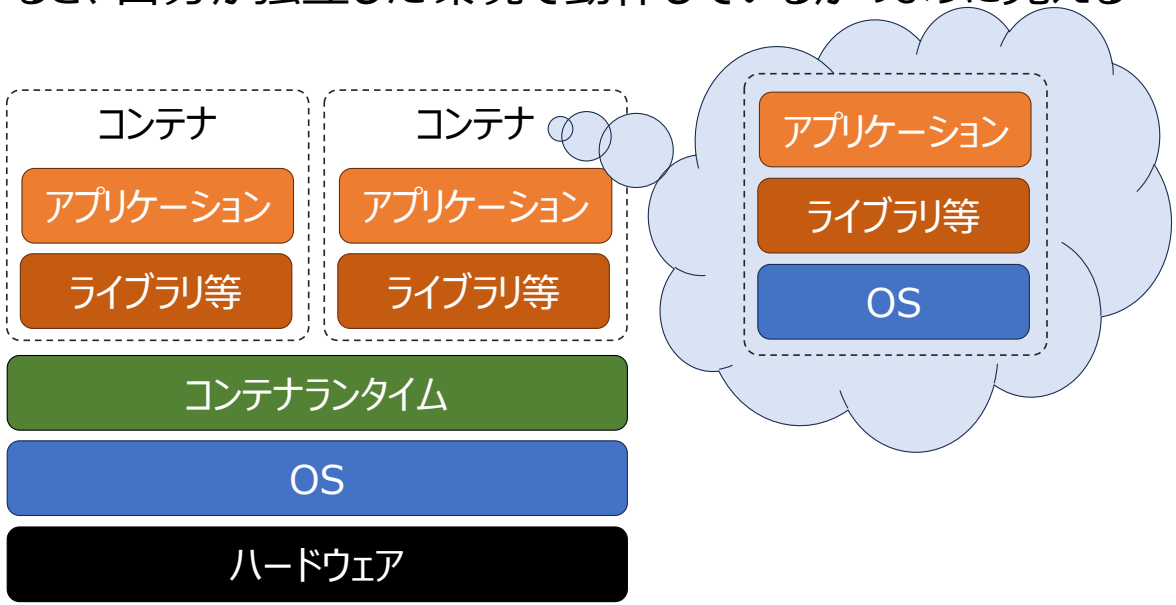
コンテナ技術の仕組み

コンテナランタイム（コンテナエンジン）：

- ファイルシステムやCPUメモリなど、あらゆるリソースを仮想的に分離
- コンテナ内のアプリケーションからすると、自分が独立した環境で動作しているかのように見える

仮想化との違い：OS選択の制限

- コンテナにOSが含まれないため、ベースのホストOSを使用
- そのため、Linux OS上でコンテナを構築する場合、コンテナ内ではLinuxのアプリケーションのみ実行可能



LinuxコンテナとWindowsコンテナ

- **Linuxにも、Windowsにも、コンテナ技術/製品が存在します。**
- **しかし以下の理由でWindowsコンテナの普及はまだ限定的です：**
  - 後発である
  - 互換性に制約がある
  - 運用の難しさ（自動化の難しさや
    コンテナイメージの大きさなど）
- **そのため、本講座では基本的に
  「Linuxコンテナ」のみ扱います。**

**Linuxコンテナの
よく使うイメージ**

alpine	5MB
ubuntu	30MB
redis	20MB
postgres	150MB

**Windowsコンテナの
よく使うイメージ**

フル	4.2GB
サーバーコア	3.7GB
ナノサーバー	251MB
iotcore	307MB

＊ コンテナイメージの詳細は、この後詳細に説明します。上記サイズ情報は2024年11月現在のものです。

Linuxコンテナの仕組み

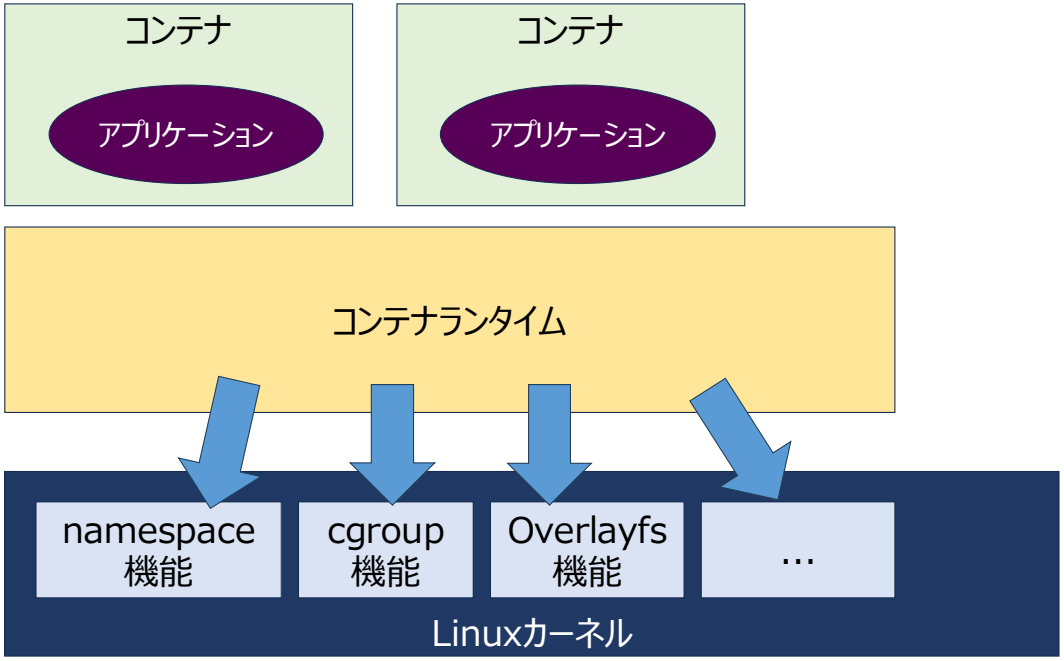
Linuxコンテナを支える主な技術

機能・技術	役割
namespace（名前空間）	プロセスなどの隔離機能です。 namespaceを利用することによって、1つのコンテナ内から、ほかのコンテナのプロセスにアクセスできなくなります。
cgroup（コントロールグループ）	CPUやメモリー、ネットワーク帯域などのリソースの隔離機能です。 cgroupにより、コンテナに対してCPUやメモリなどの動的に割り当てることが可能です。
Capability	root権限を細かく分け、必要な権限のみをプロセスやファイルに付与する機能です。
Overlayfs	レイヤー構造を持つファイルシステムです。 Overlayfsを使用することで、ベースとなるファイルシステムを読み取り専用にすることができます。その結果、同じホストマシン上で複数のコンテナが動作していても、互いのファイルアクセスが干渉しないようにすることが可能です。

Linuxコンテナの仕組み

コンテナランタイムの構造

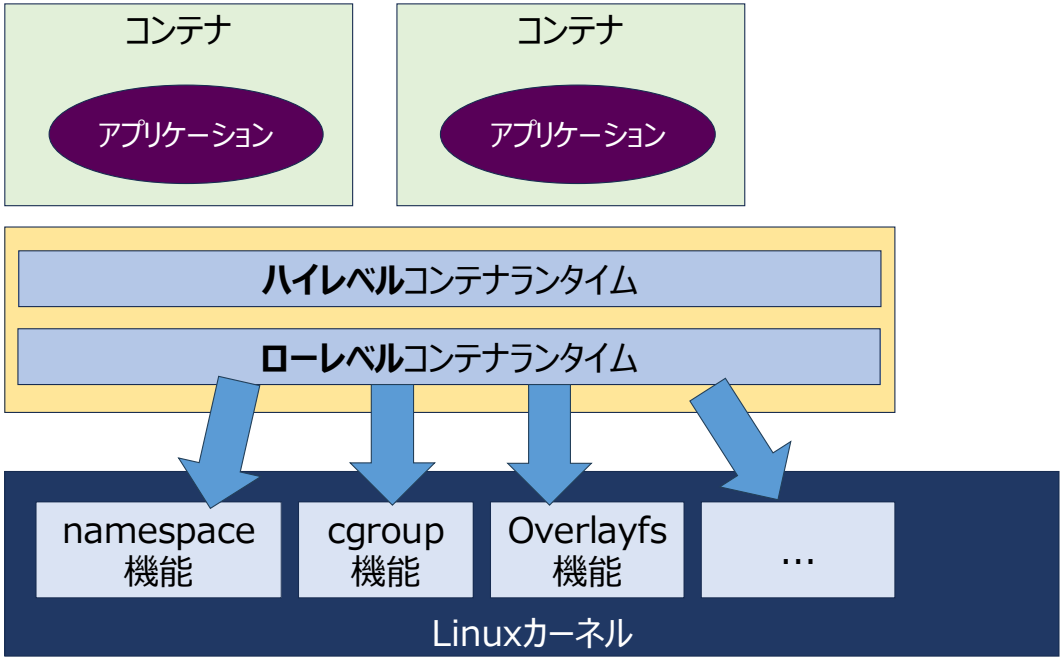
- namespaceなどの機能は、Linuxカーネルが提供する機能
- コンテナランタイムは、これらの機能を利用して、コンテナを実現



Linuxコンテナの仕組み

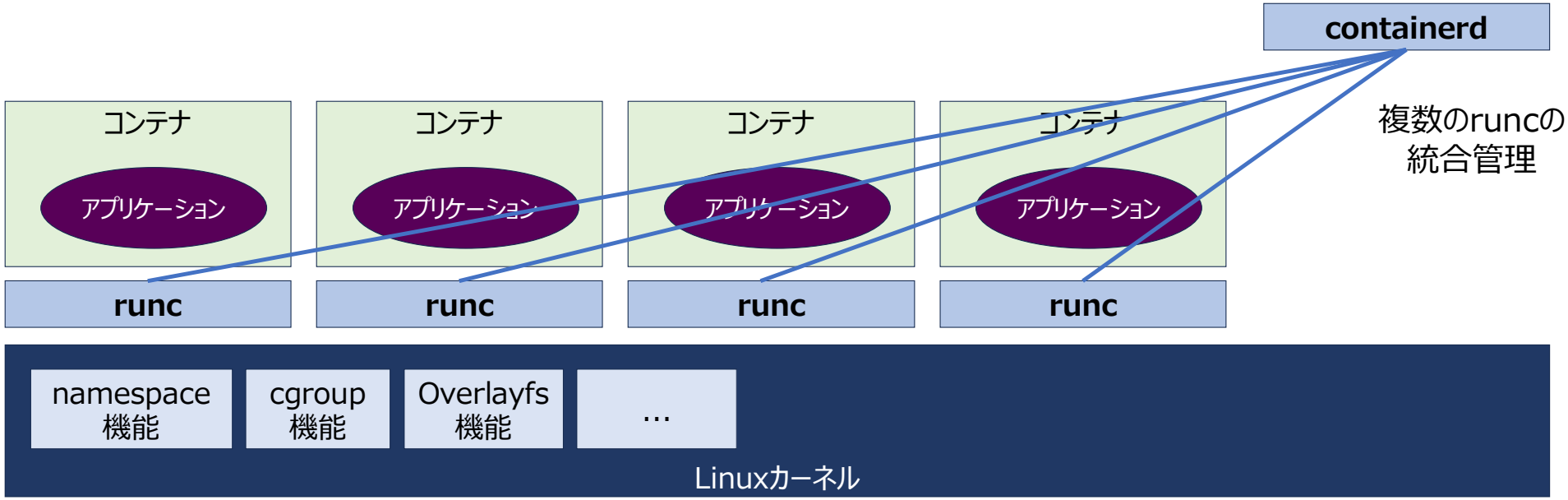
コンテナランタイムの構造

- namespaceなどの機能は、Linuxカーネルが提供する機能
- コンテナランタイムは、これらの機能を利用して、コンテナを実現
- コンテナランタイムは、通常ローレベルとハイレベルの2層に分かれます：
  ローレベルコンテナランタイム：Linuxカーネルの機能を用いてコンテナの作成、実行、削除などの基本的な操作を直接管理します。
  例：**runc**
  ハイレベルコンテナランタイム：コンテナイメージの管理や、複数のコンテナの統合管理などを実現します。
  例：**containerd**



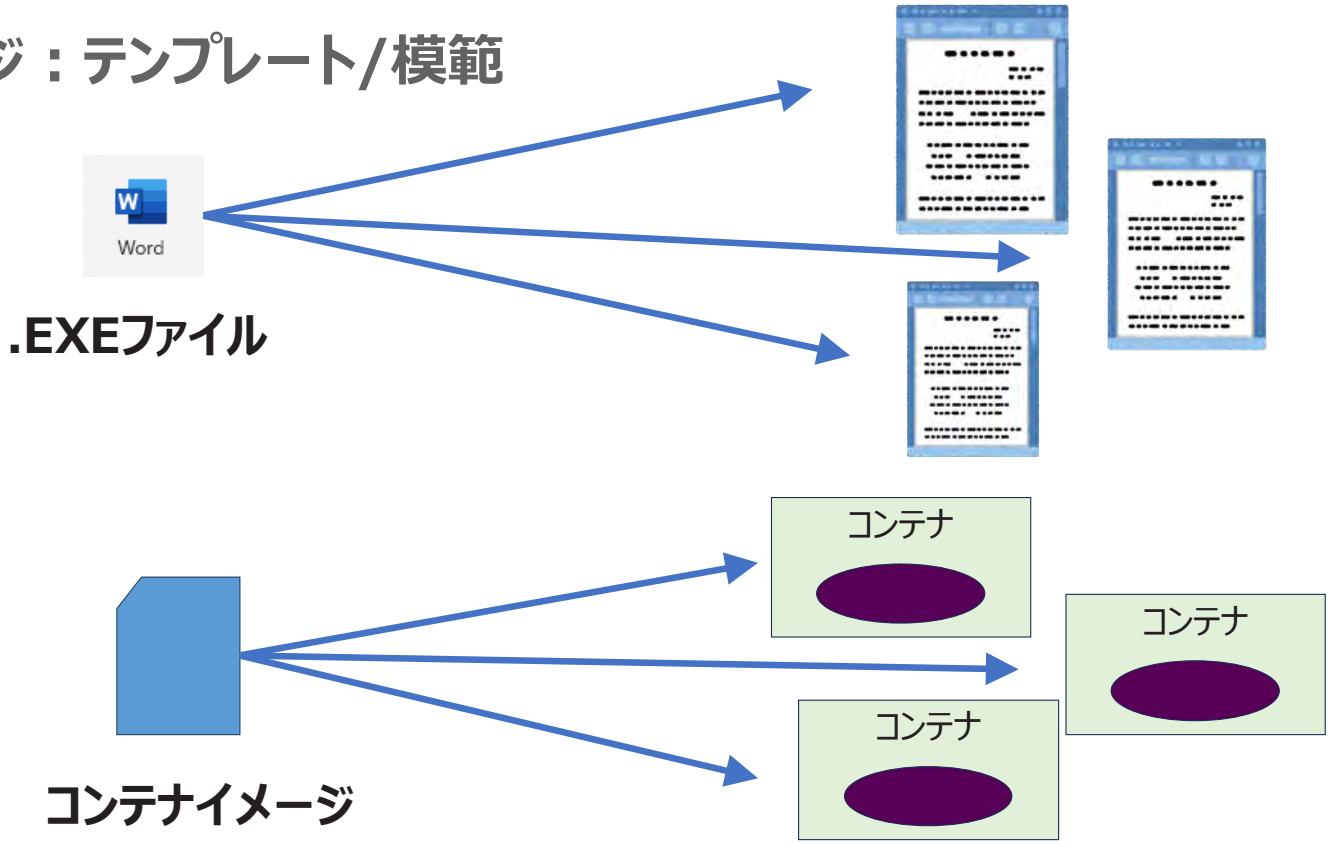
Linuxコンテナの仕組み

コンテナランタイムの構造



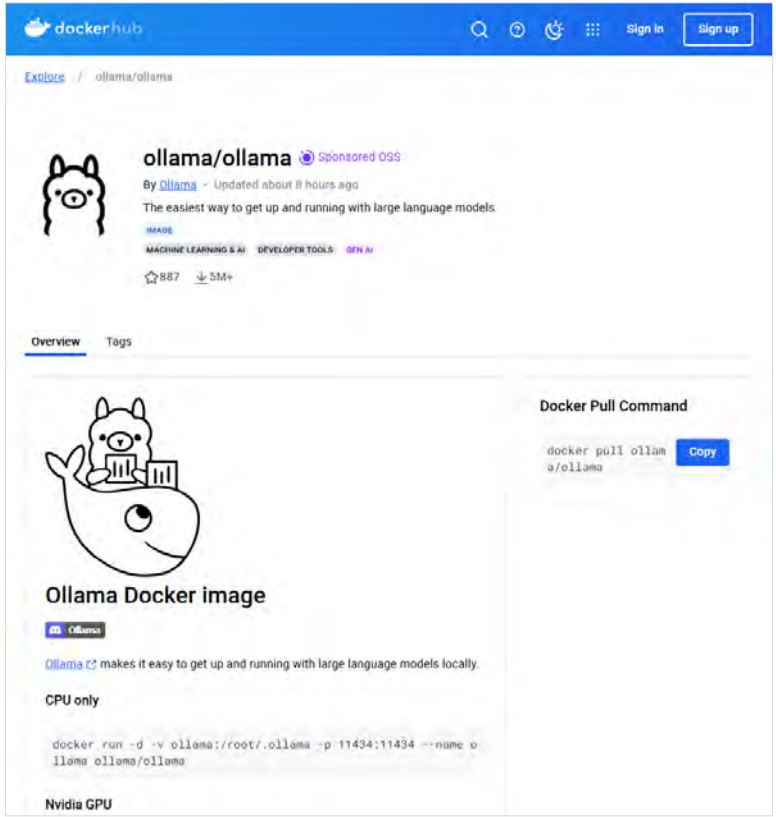
コンテナイメージ

■ コンテナイメージ：テンプレート/模範



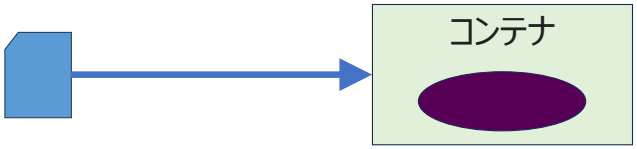
コンテナレジストリ

- コンテナイメージ及び関連情報を保管するためのサービスです。
- パブリック（一般公開）のもの
  - Docker Hubがもっとも有名
  - AWS/Azureなどのクラウドも一般公開のレジストリを提供
  - Githubは、GitHub Container Registry (GHCR)を提供
- プライベートのレジストリ（社内用など）
  - CNCF Distribution

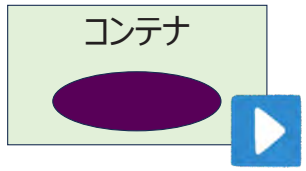


コンテナのライフサイクル例

コンテナイメージの入手
自分で作成/レジストリよりダウンロード



コンテナイメージからコンテナの作成



コンテナの起動

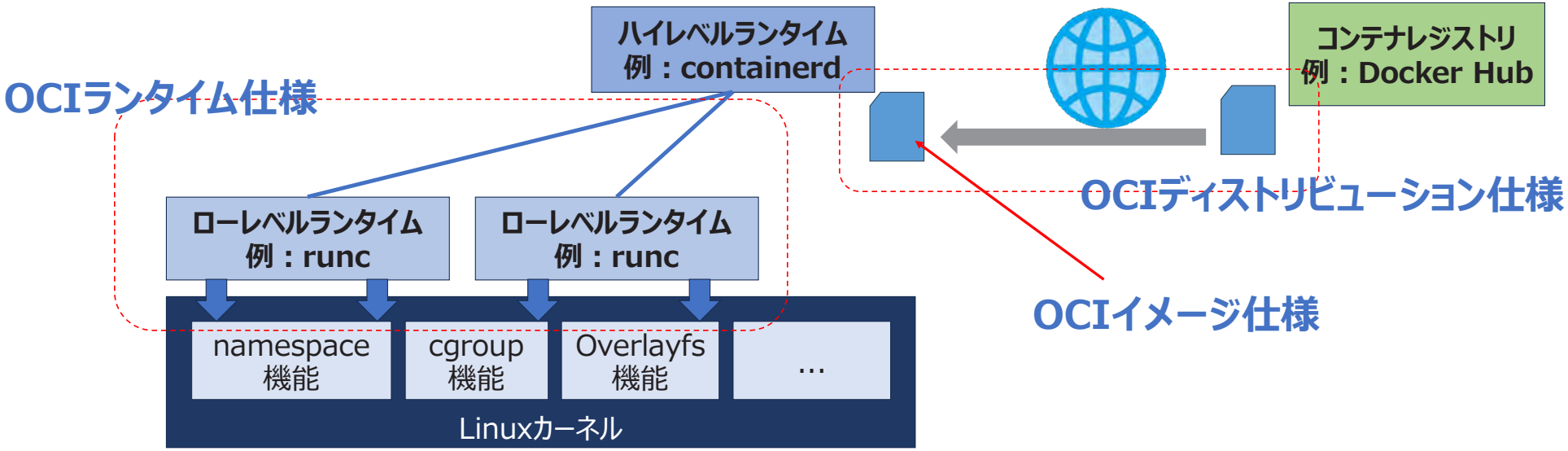
コンテナの停止
コンテナ内のアプリケーションが終了したら、
コンテナが停止状態になる
OR
手動でコンテナを停止する

コンテナの再開
停止したコンテナを再開できる

コンテナの削除
コンテナが不要になったとき、削除する
（コンテナ内のデータが完全に消える）

コンテナに関連する標準化

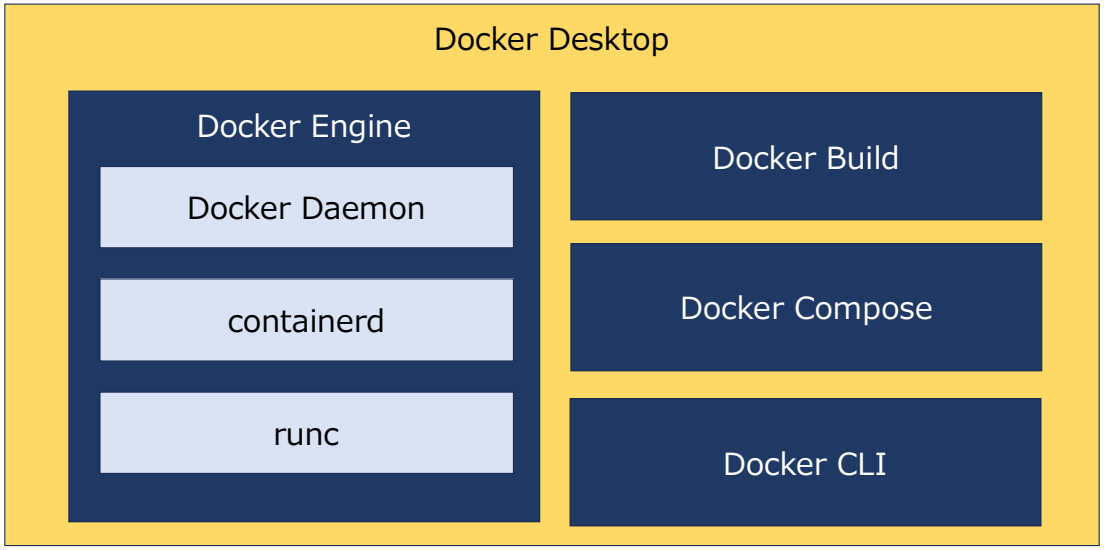
- OCI（Open Container Initiative）によって定められた仕様
  - OCIランタイム仕様（OCI Runtime Specification）
  - OCIイメージ仕様（OCI Image Specification）
  - OCIディストリビューション仕様（OCI Distribution Specification）



Dockerの基本

- Dockerは、製品群の名称です。
- WindowsではDocker Desktopが最もよく使われます。
  Linux向け製品である「Docker Engine」を、Windowsで動作するようにパッケージ化したものがDocker Desktopです。

内部では、containerdや
runcを使用しています

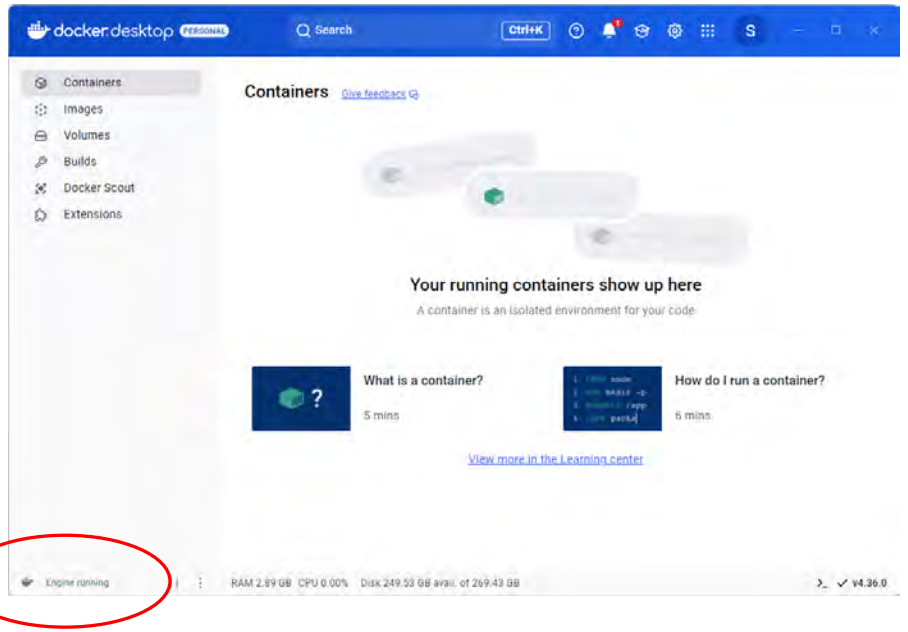


Docker Desktopを使ってみよう

Docker Desktopのインストール

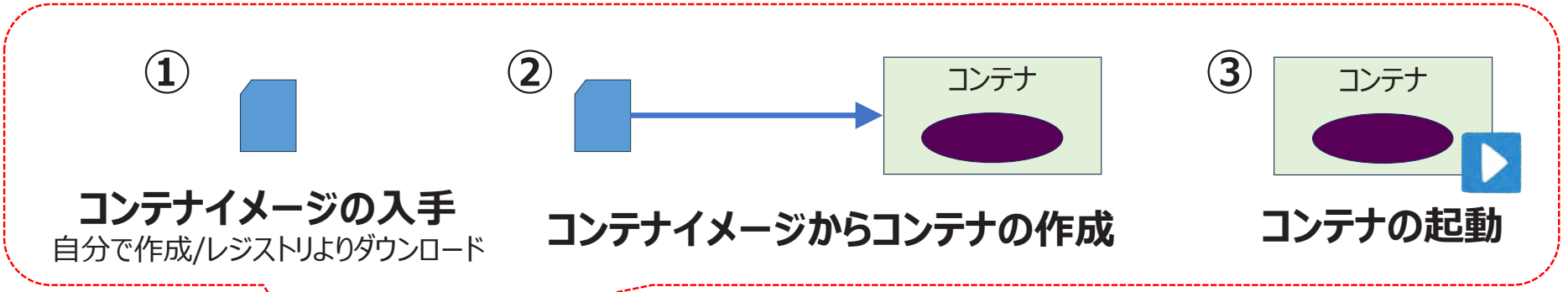
- Docker Desktopは、個人利用の場合は無料ですが、一定の条件を満たす組織での利用にはライセンス料が発生します。
- 公式ドキュメントに従ってインストールします：
  https://docs.docker.jp/docker-for-windows/install.html
- インストール後、Docker Desktopを起動します。右側のように、画面左下が「Engine running」になっていることを確認します。
- コマンドプロンプトで、「docker -v」でDocker CLIが正常に動作することを確認します。

```
C:\>docker -v
Docker version 27.3.1, build ce12230
```



Docker Desktopを使ってみよう

- Docker Desktopインストール後、試しに「docker run hello-world」コマンドでコンテナを実行してみます。
- 「docker run」の1つのコマンドで①～③のステップを実施してくれます。




```
docker run hello-world
```

- Docker Hubより、「hello-world」というイメージをダウンロードする
- 「hello-world」イメージから、コンテナを作成する
- 作成したコンテナを起動する

* 起動されたコンテナは、画面上に「Hello from Docker!...」のメッセージを出力して、そのまま終了します。

Docker Desktopを使ってみよう

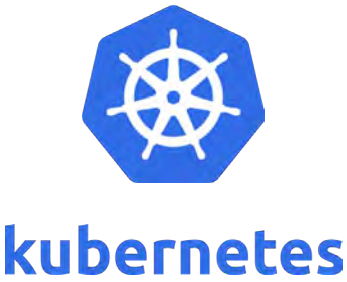
よく使うDockerコマンド

- 各コマンドの詳細は、ドキュメントを参照してください。

コマンド	説明
`docker run`	コンテナを作成して、起動する。イメージがローカルに存在しない場合、自動的にダウンロードする。
`docker exec`	実行中のコンテナに対して、コンテナ内でLinuxコマンドを実行する
`docker ps`	実行中のコンテナー一覧を表示する。-aオプションをつけることで、停止中のコンテナを含め、すべてのコンテナを表示する。
`docker stop`	実行中のコンテナを終了する
`docker kill`	実行中のコンテナを強制終了する
`docker rm`	コンテナを削除する
`docker build`	dockerfileに記述された内容に従い、コンテナイメージを作成する

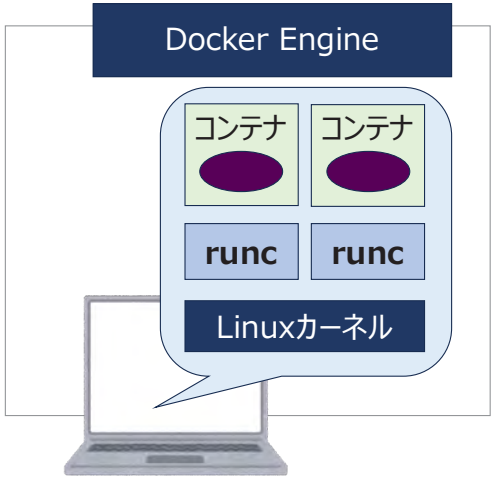
Kubernetesとは

- **Kubernetes（クバネティス、略記「K8s」）は、コンテナ及びコンテナ内のアプリケーション、コンテナの動作環境を取りまとめて管理し、運用、自動化するためのオープンソースのソフトウェアです。**
  - もともとは、Googleが開発した社内製品でしたが、2014年にオープンソース化しました。
  - "K"と"s"の間にある8つの文字を数えることから、K8sが略語として使われています。
  - Kubernetesプロジェクトは、現在CNCF（Cloud Native Computing Foundation）という組織によって管理されています。

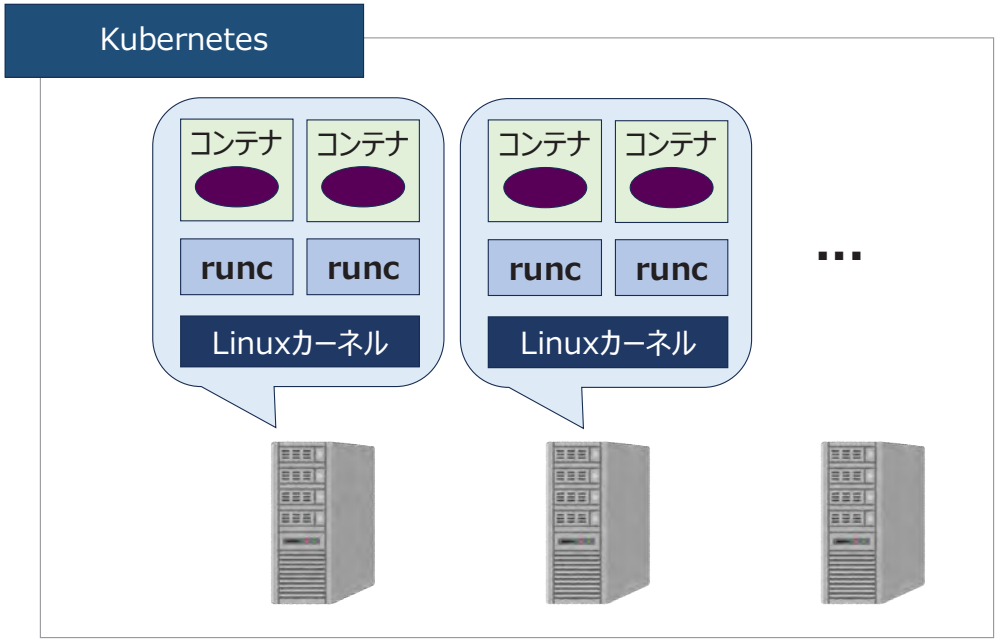


DockerとKubernetes

- どちらも、コンテナランタイム（containerd/runcなど）をベースとする製品
- Dockerは、最も一般的な使い方として**単一のマシン上**で**開発環境**として動作しますが、Kubernetesは**複数のマシン上**で**本番環境**として動作します。

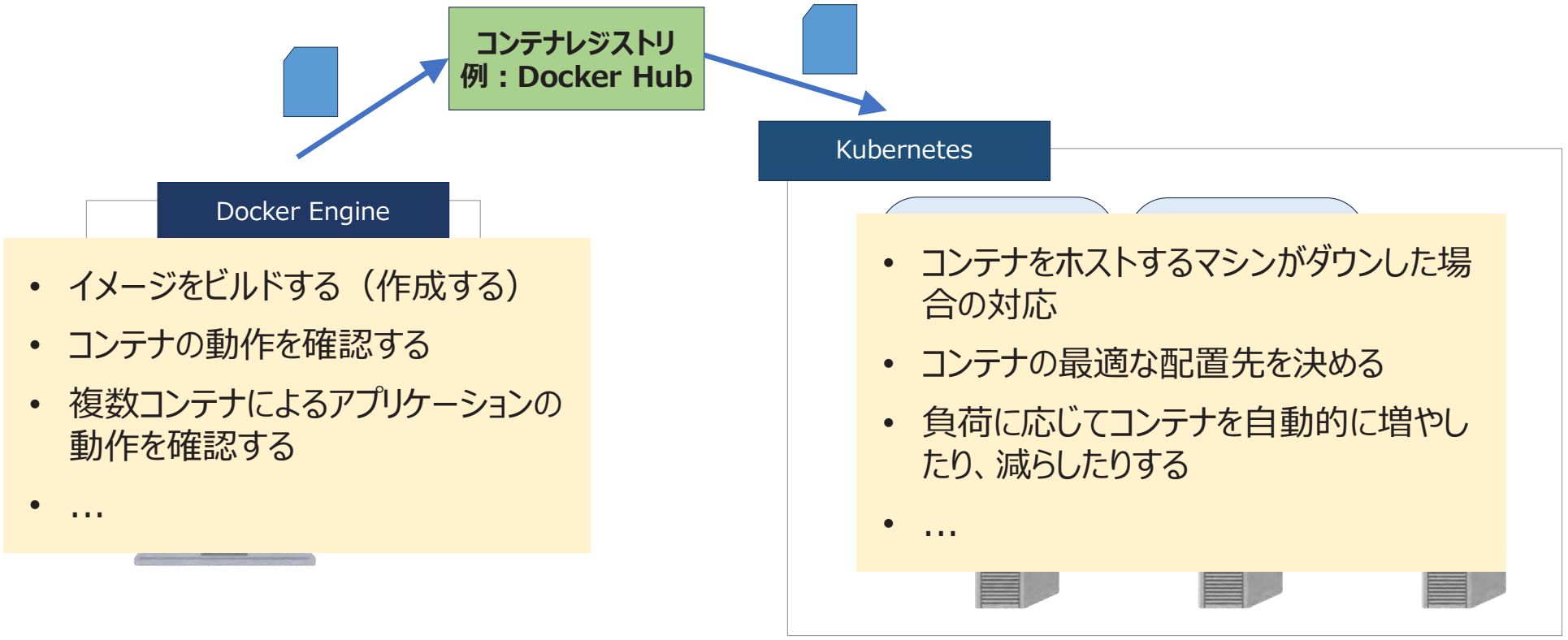


＊ あくまでも一般的な使い方です。Dockerを複数マシン上で本番環境として動作することも可能です。



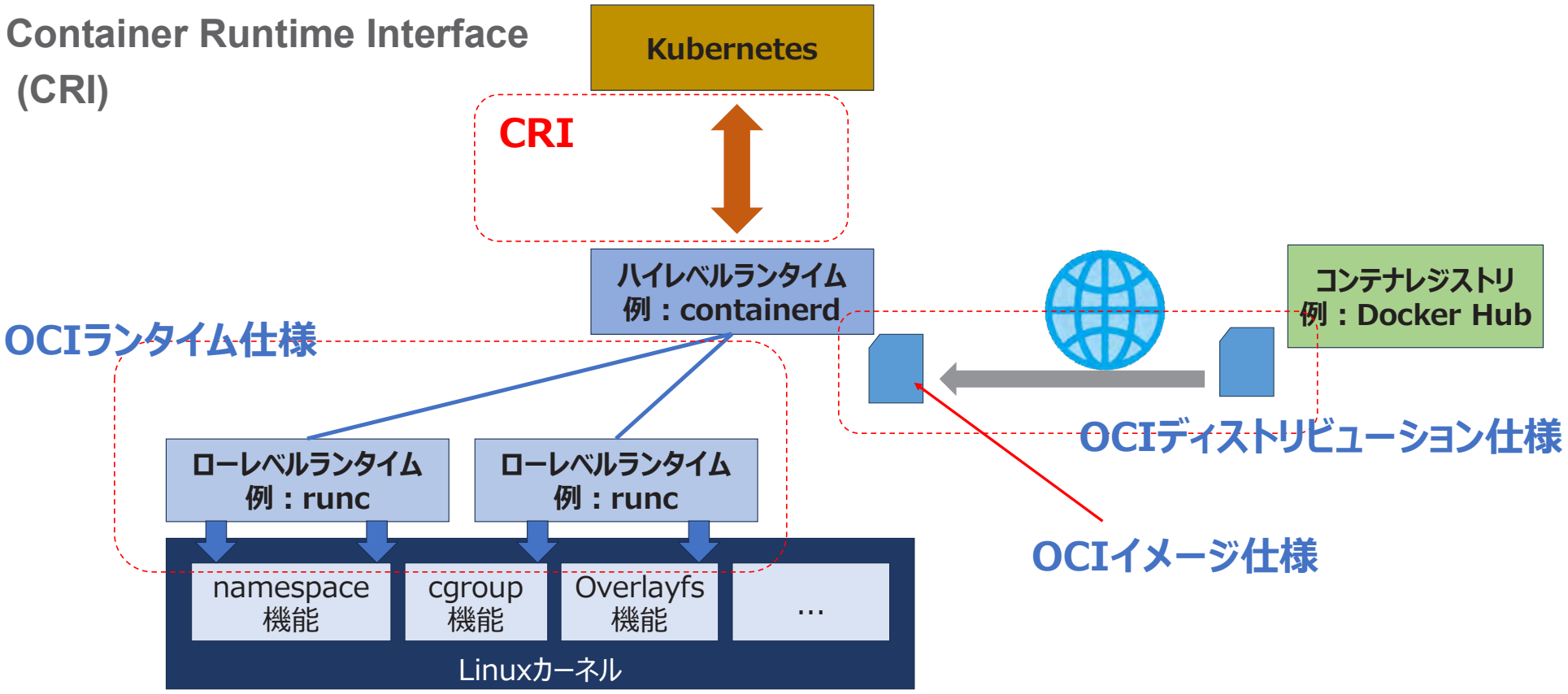
DockerとKubernetes

根本的な役割や性質が違います



Kubernetesとコンテナランタイムの間のインタフェース

- Container Runtime Interface (CRI)



確認テスト1

Q1: 一般的な話として、仮想マシンと比較した場合、コンテナの優位性として正しいものはどれか？当てはまるものをすべて選択してください。

1. 起動速度が速い。
2. OSの選択肢が多い。
3. イメージファイルが小さい。
4. 通信速度が速い。

Q2: 「コンテナはLinuxのみの技術であり、Windowsにはコンテナが存在しない」という記述は正しいか？最も適切なものを選択してください。

1. 正しい
2. 正しくない

確認テスト2

Q3: Linuxコンテナを支える技術として、以下のどれが含まれているか？当てはまるものをすべて選択してください。

1. namespace
2. cgroup
3. kvm
4. overlayfs

Q4: containerdとruncについて、正しい記述は次のうちどれか？当てはまるものをすべて選択してください。

1. runcはコンテナランタイムであるが、containerdはコンテナランタイムではない。
2. runcは、複数のcontainerdを束ねて管理する。
3. runcは、ローレベルのコンテナランタイムである。
4. containerdは、ハイレベルのコンテナランタイムである。
5. OCI仕様に準拠していれば、runcを他のランタイムに置き換えても問題なくcontainerdと組み合わせて動作することができる。

確認テスト3

Q5: Docker製品についての記述の中で、正しいものは次のうちどれか？当てはまるものをすべて選択してください。

1. Docker DesktopはWindows/Mac向け製品で、Linuxでは動作しない。
2. Dockerはcontainerdやruncと無関係で、独自の技術を利用している。
3. Dockerは、レジストリとしてDocker Hubのみ利用できる。
4. Dockerは完全無料で、どんな場合でも費用は発生しない。

Q6: 以下のDocker CLIコマンドとその説明の組み合わせについて、正しいものは次のうちどれか？当てはまるものをすべて選択してください。

1. `docker stop`：実行中のコンテナを終了する。
2. `docker run`：実行中のコンテナの一覧を表示する。
3. `docker rm`：コンテナをリネーム（名前変更）する。
4. `docker ps`：コンテナ中のプロセス一覧を表示する。
5. `docker build`：コンテナイメージを作成する。

確認テスト4

Q7: DockerとKubernetesの関係について、正しい記述は次のうちどれか？当てはまるものをすべて選択してください。

1. Dockerは単一マシンや小規模環境で利用されることが多く、Kubernetesは複数マシンや大規模環境で利用されることが多い。
2. Kubernetesは単一マシンや小規模環境で利用されることが多く、Dockerは複数マシンや大規模環境で利用されることが多い。
3. Dockerは開発環境で利用されることが多く、Kubernetesは本番環境で利用されることが多い。
4. Kubernetesは開発環境で利用されることが多く、Dockerは本番環境で利用されることが多い。

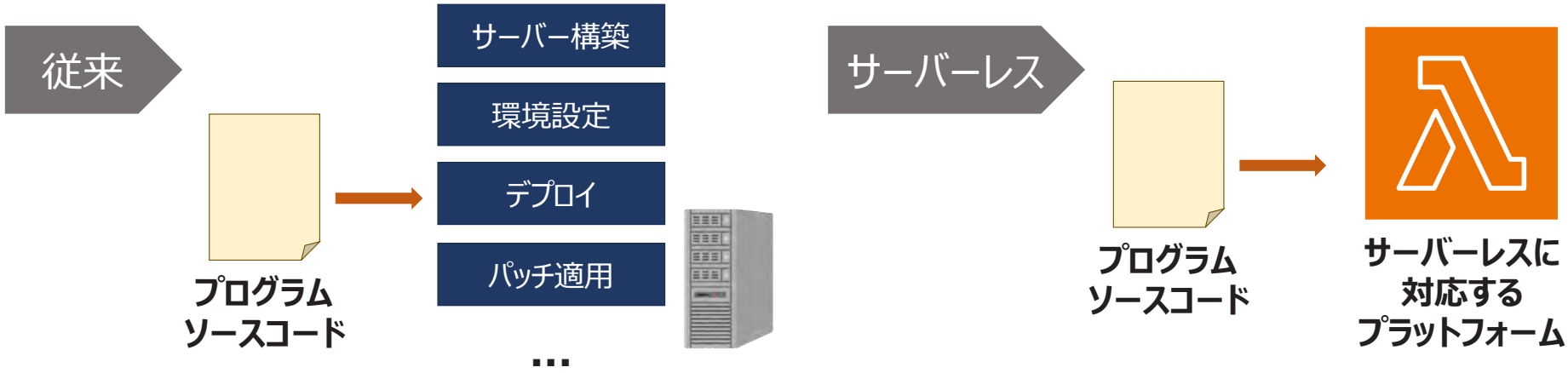
Q8: Kubernetesは、コンテナランタイムとどのようなインターフェースを使用してやり取りするか？最も適切なものを選択してください。

1. OCI
2. CGI
3. CRI
4. MSI

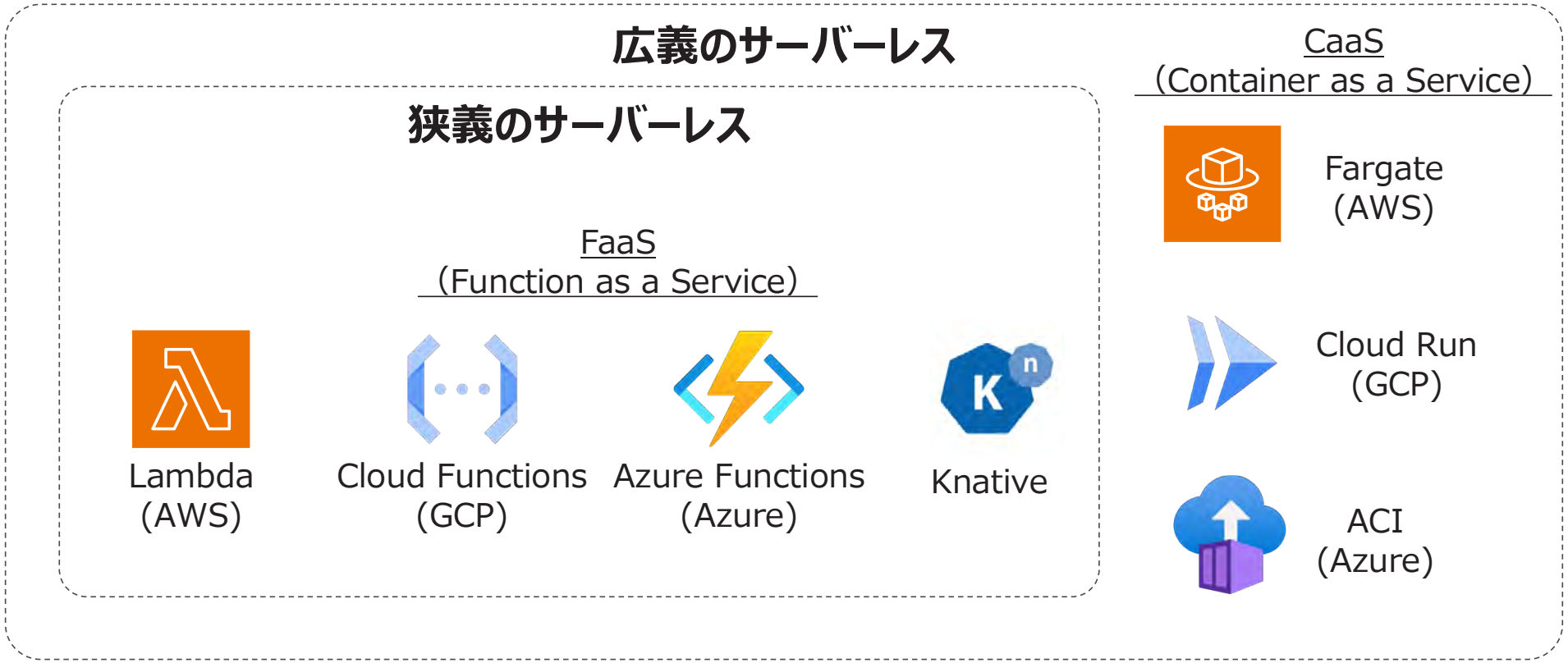
第四章 サーバーレスアーキテクチャ

サーバーレスとは

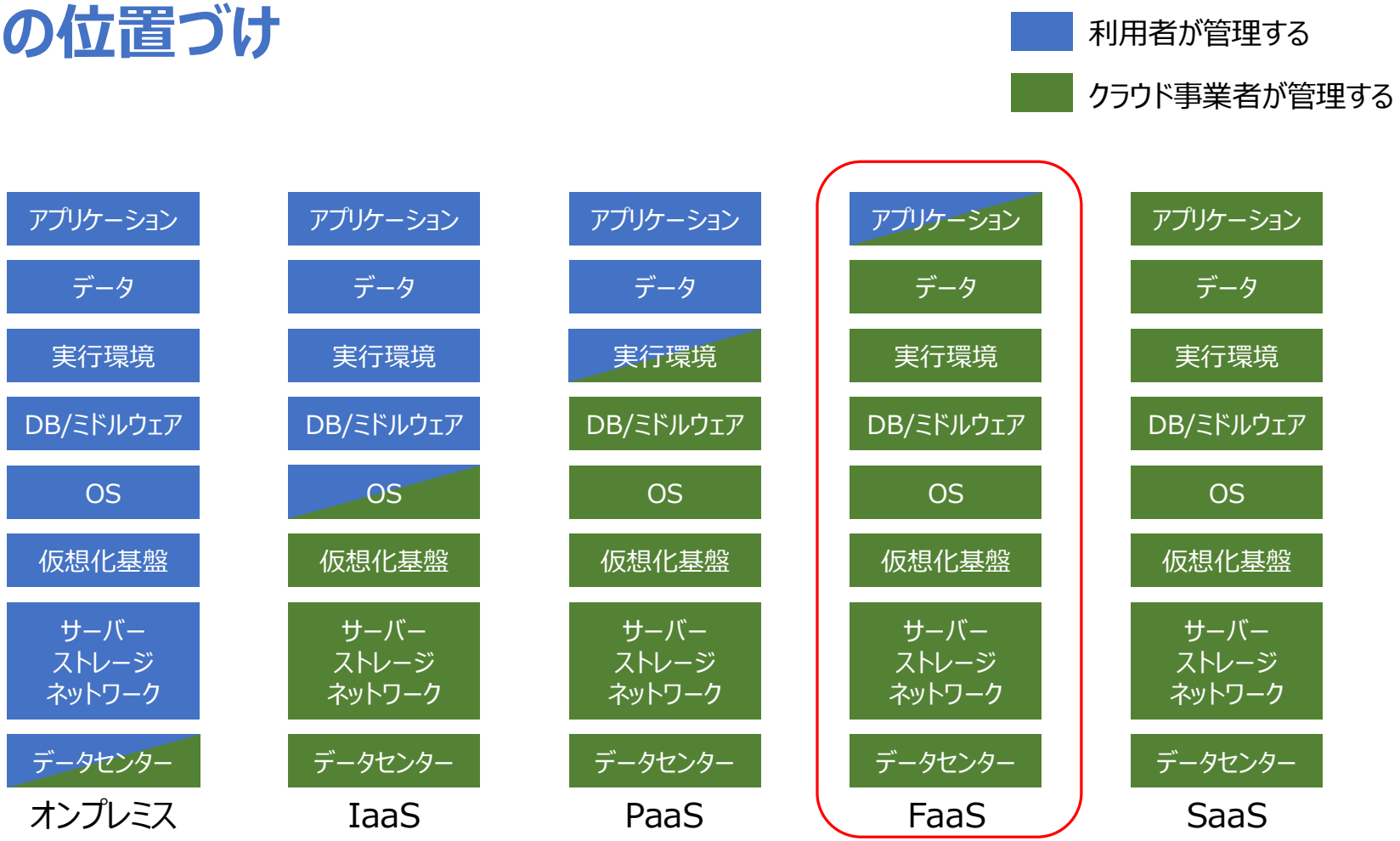
- Serverless（サーバーレス）とは、開発者がサーバーを管理することなくアプリケーションを構築および実行できるクラウドコンピューティングモデルです。
- 英語の「Serverless」を日本語に直訳すると「サーバーなし」になりますが、本当にサーバーが存在しないわけではなく、「サーバーを**構築する**ことも、**管理運用する**こともなく、意識することすら不要」という意味合いです。



狭義のサーバーレスと広義のサーバーレス

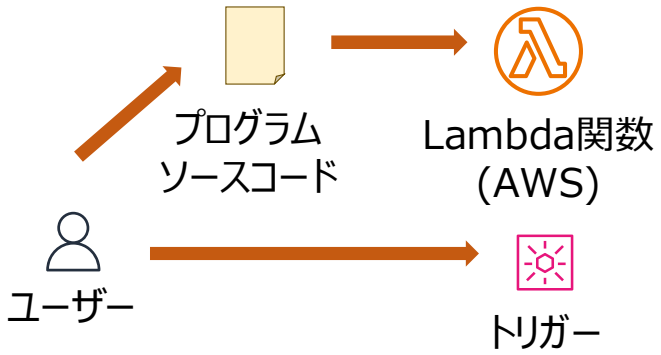


FaaSの位置づけ

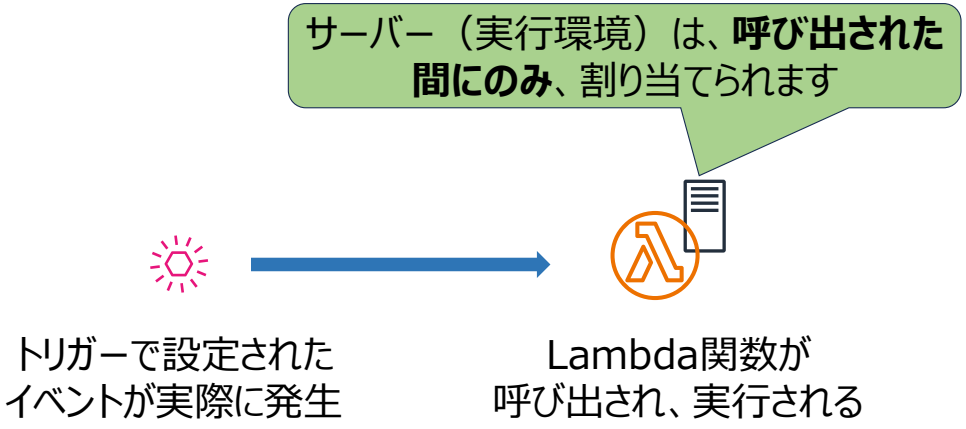

利用者が管理する
クラウド事業者が管理する
アプリケーション
データ
実行環境
DB/ミドルウェア
OS
仮想化基盤
サーバー
ストレージ
ネットワーク
データセンター
オンプレミス
IaaS
PaaS
FaaS
SaaS

実に簡単、FaaSの使い方

- **ここでは、AWS Lambdaを例として、FaaSの実際の使い方を説明します。**
  - 例えば「ユーザーからリクエストがあったときに、このプログラムを実行します」という要件があったとします。
  - 準備段階では、まずプログラムのソースコードをアップロードします。そして、ソースコード実行のきっかけとなる「トリガー」を設定します。
  - そうすると、トリガーで設定されたイベントが実際に発生した場合、Lambda関数（アップロードしたソースコード）が自動的に呼び出され、実行されます。



準備段階



実行段階

イベント駆動型アーキテクチャ

- 従来のアプリケーションでは、リクエスト駆動型のアーキテクチャを採用するのが主流でしたが、モダンなプリケーションでは**イベント駆動型アーキテクチャ**（Event-driven architecture, EDA）がよく見られるようになりました。

＊ すべてのリクエスト駆動型アプリケーションがイベント駆動型にリファクタリングできるわけではありません。
一般的には、リクエスト駆動型は同期通信に適しており、イベント駆動型は非同期通信に適しています。

イベント駆動で活躍するFaaS

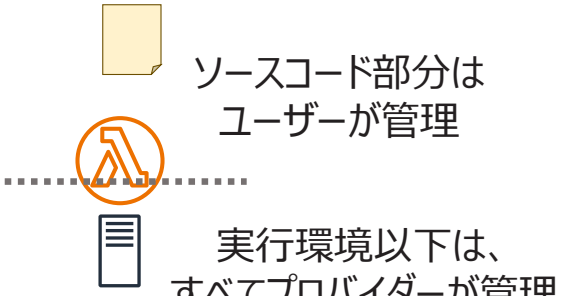
- **FaaSは、イベント駆動アーキテクチャと非常に相性が良いです。多くのイベントソースからトリガーを受け取れるため、イベント駆動アーキテクチャの中心的な役割を担います。**

もし＿＿＿＿＿＿＿＿＿＿＿＿ら、
Lambda関数を実行します

- ユーザーからWebリクエストが来た
- あるプログラムに処理エラーが発生した
- 事前に設定した時刻になった
- データベースに書き込みがあった
- 決済が完了した
- S3にファイルがアップロードされた
- EC2サーバーがダウンした
- メッセージキューに新しいメッセージが届いた
- 新しいIoTデバイスが接続された
- …

FaaSのメリット

■ イベント駆動アーキテクチャの中心的な役割を担うFaaSは、管理効率、コスト、可用性、拡張性などにも優れています。



サーバー管理不要

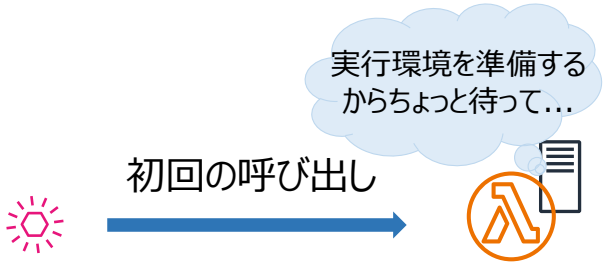
高可用性・高拡張性

従量課金

FaaSの注意点

コールドスタート（初回起動時の遅延）

- 初回起動時や、しばらく呼び出されていないときは、起動に時間がかかる現象のことです。

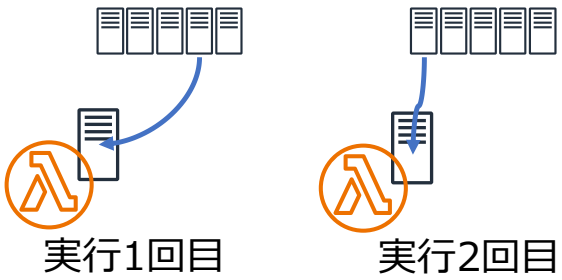


長時間の処理に不向き

- FaaSは、軽量なタスクやマイクロサービスのために設計されているため、例えば長時間の動画エンコードのようなタスクに適していません。

内部でステートを保持する必要がある場合は利用不可

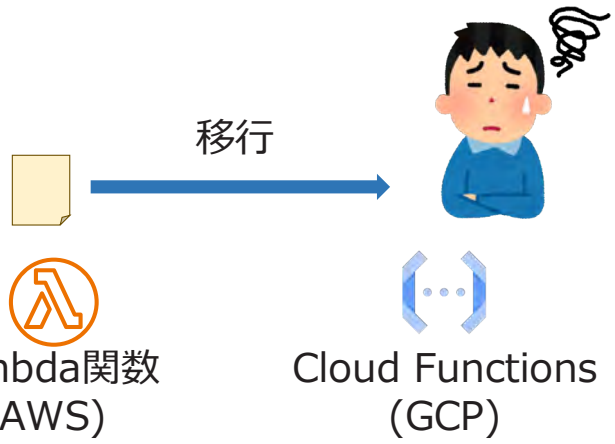
- データやセッション状態は、通常外部のデータベースなどで管理されますが、どうしてもサーバー内部で保持する必要がある場合は、FaaSを利用することはできません（ステートレス）。



FaaSの注意点（続き）

ベンダーロックインのリスク

- 各プロバイダーのFaaSには互換性がないので、移行が困難なケースもあります。
- 一方、コンテナは標準化されているため、例えばAWS上で動作しているコンテナをAzure上に移行するのは、簡単にできます。

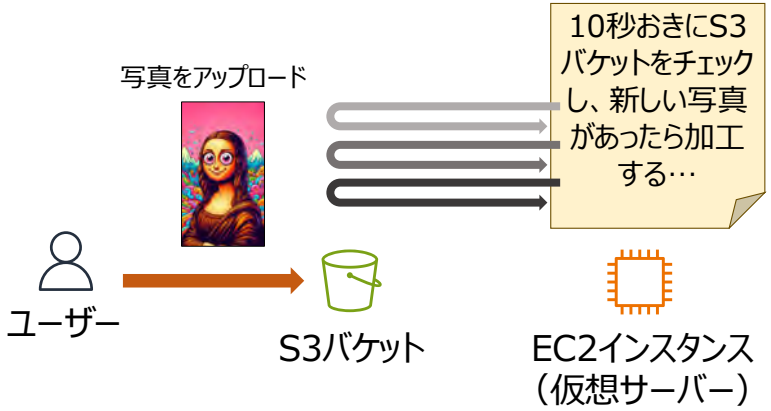


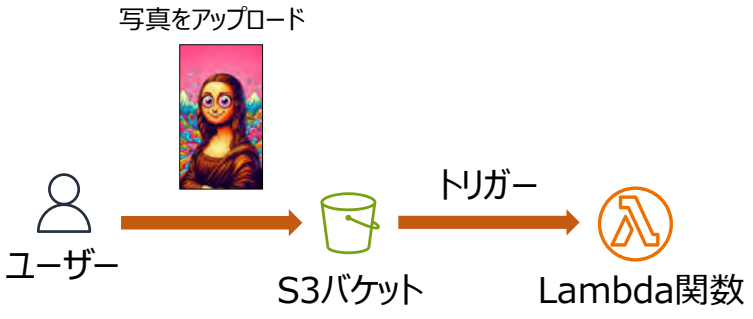
クラウド特有のサービスなのでローカルでの開発やデバッグが複雑になりがち

- FaaSはクラウド上で動作することを前提としています。そのため、ローカルで開発ツールなどが提供されているとは言え、開発環境やワークフローが複雑になりがちです。

FaaSの典型的なユースケース

- **例①：「ユーザーが写真をS3にアップロードする。写真がアップロードされると、リアルタイムでその写真を加工して、Webサイトに掲載したい」という要件**
  - FaaSを使用せず従来のサーバーで実現する場合、サーバーを常時稼働させ、数秒間隔でS3バケットに対してポーリングする必要があります。
  - 一方、FaaS（例：Lambda関数）を利用すると、事前にトリガーを設定しておけば、イベント駆動で写真がアップロードされる場合即座に関数が呼び出され、コードが実行されます。



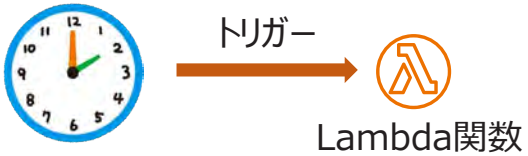


FaaSの典型的なユースケース

■ 例②：「毎日夜中2時に、不要な一時ファイルをまとめて削除したい」という要件

- FaaSを使用せず従来のサーバーで実現する場合、「cron」などのタスクスケジューラを使用するケースが多いです。その場合、タスクスケジューラを稼働させるサーバーは、常時に立ち上がっている必要があります。
- 一方、FaaS（例：Lambda関数）を利用すると、時刻をトリガーとして設定できるため、設定した時刻になると呼び出されます。

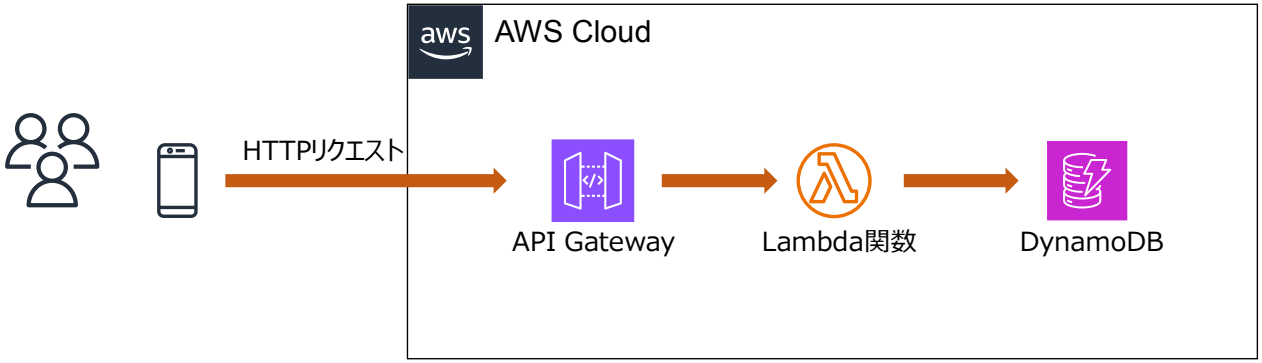
※ただし、例えば「毎晩すべてのファイルのバックアップを実施する」といったような長時間タスクには適していません。



FaaSの典型的なユースケース

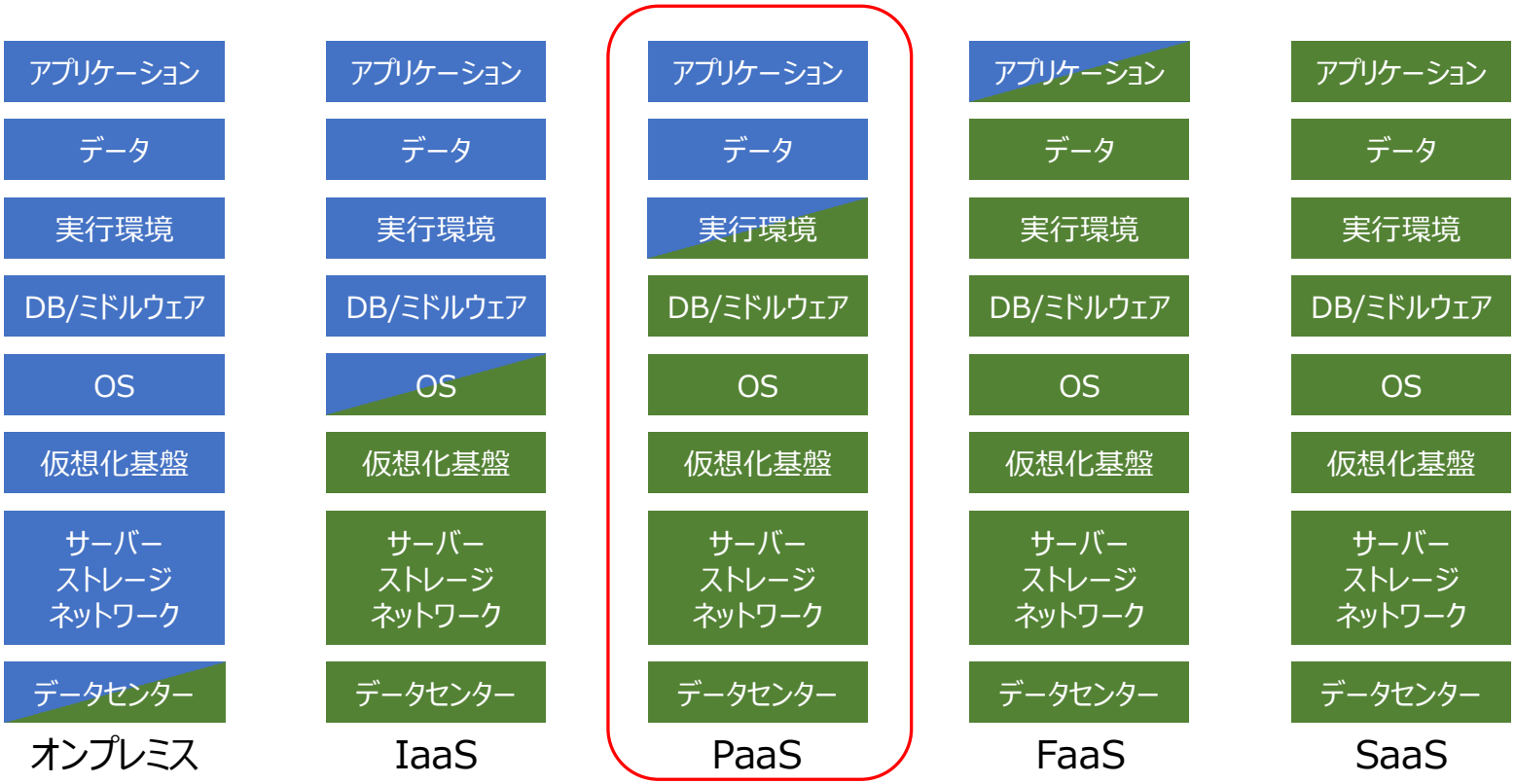
■ 例③：RESTful APIの構築

- FaaSは、RESTful API構築において高い柔軟性とコスト効率を提供するため、非常に適しています。
- AWSの場合、API Gatewayと組み合わせることで、HTTPリクエストをトリガーにLambda関数を実行し、DynamoDBとのデータ操作をシームレスに行えます。
- サーバー管理が不要で、さらにリクエストごとの課金モデルにより、アクセスが少ない場合でも無駄なコストを抑えられます。



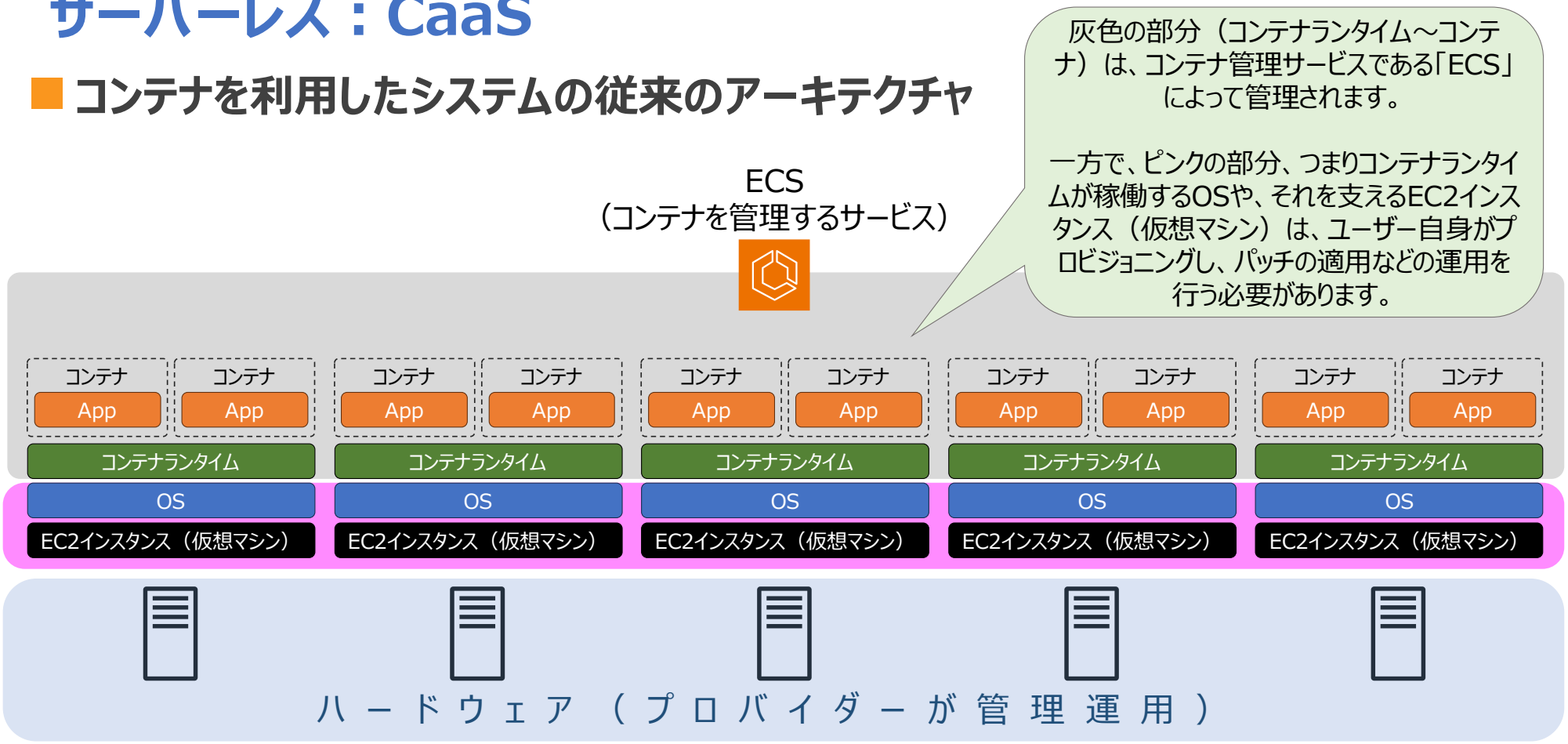
CaaSの位置づけ

- CaaS（Container as a Server）は、PaaSの一種です。



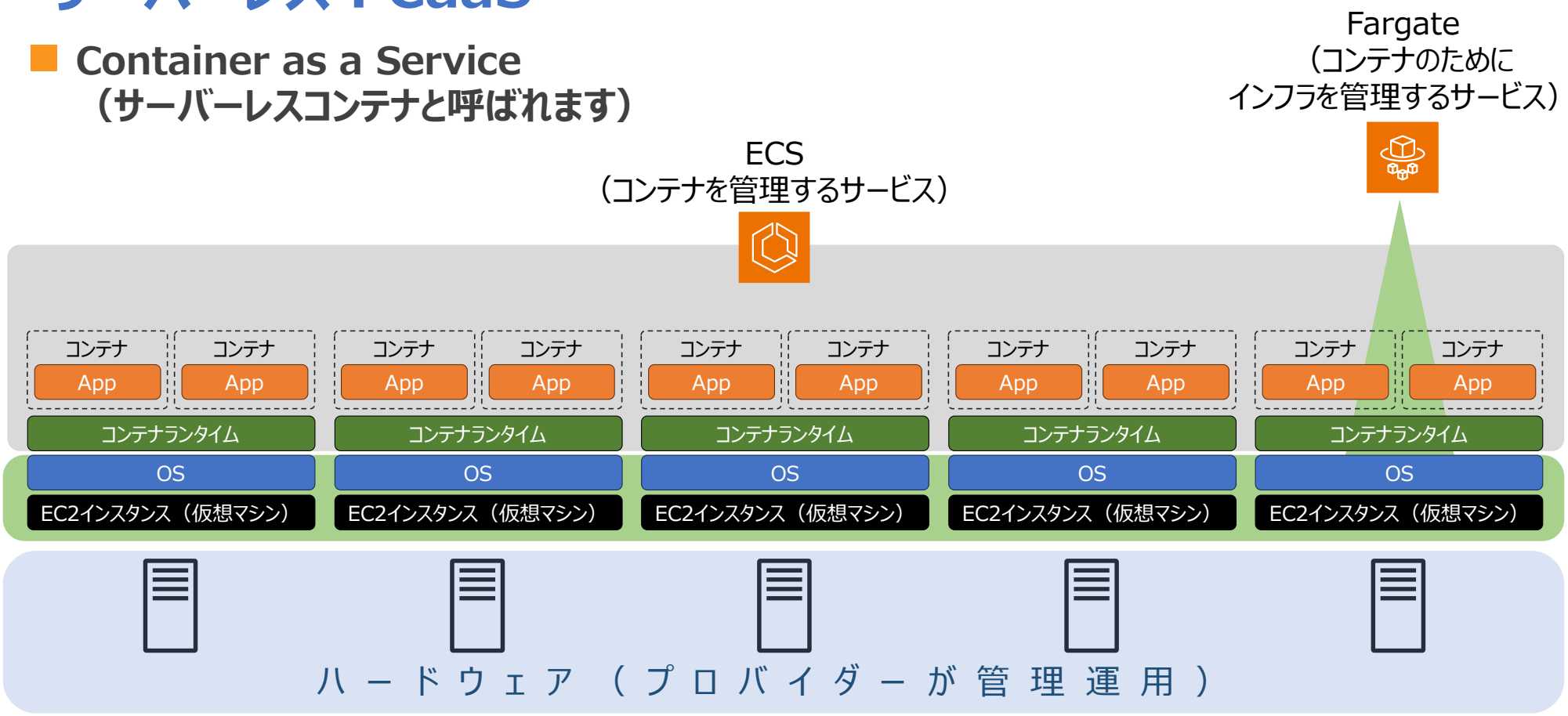
サーバーレス：CaaS

コンテナを利用したシステムの従来のアーキテクチャ



サーバーレス：CaaS

- **Container as a Service**
  （サーバーレスコンテナと呼ばれます）



コンピュート：適材適所

サービスモデル	IaaS	PaaS		FaaS
		従来型PaaS	CaaS	
代表的なAWSサービス	• EC2	• Elastic Beanstalk	• ECS on Fargate • EKS on Fargate	• Lambda
自由度	高 ←			→ 低
管理の手間	多 ←			→ 少
スケールの単位	インスタンス	インスタンス	アプリケーション	ファンクション（関数）
抽象化	ハードウェア	ハードウェア/OS	OS	ランタイム
使いドコロ	• OS/ネットワーク/ストレージのレベルで制御が必要 • 特殊なOSを利用する • OS以上を細かく制御する必要がある	• アプリケーションの構成やスケーリングを制御したい • OSレベルである程度制御したい	• アプリケーションの構成やスケーリングを制御したい	• 必要なときだけコードの実行を行いたい • アプリケーションの構成やスケーリングを任せたい • インフラの構成・管理を行いたくない

確認テスト1

Q1: サーバーレスとは何を指すか？最も適切なものを選択してください。

1. サーバーを不要とし、クライアントPCで実行する仕組み。
2. 「レス（less）」は「より少ない」という意味なので、サーバーをより少なく利用する節約法のことを指している。
3. サーバーを自動化する手法である。
4. 開発者がサーバーを管理することなくアプリケーションを構築および実行できること。

Q2: 次の中で、FaaS（Function as a Service）ではないものはどれか？当てはまるものをすべて選択してください。

1. AWS Lambda
2. Cloud Functions (Google)
3. Cloud Run (Google)
4. Azure Functions
5. Azure Container Instances (ACI)

確認テスト2

Q3: 次のうち、FaaS（Function as a Service）が向いているのはどれか？当てはまるものをすべて選択してください。

1. 1回の処理が2時間以上かかるタスク
2. RESTful APIのバックエンドとして、データベースから少量のデータを取得してそのままユーザーに渡すタスク
3. ユーザーが数KBのテキストファイルをアップロードしたら、それを圧縮して保存するタスク
4. 初回起動時から非常に速いレスポンスが求められるタスク

Q4: 「広義のサーバーレスには、FaaS（Function as a Service）だけでなくCaaS（Container as a Service）も含まれている」という記述は正しいか？最も適切なものを選択してください。

1. 正しい
2. 正しくない

確認テスト3

Q5: イベント駆動型アーキテクチャに関する記述のうち、正しいものをすべて選んでください。

1. イベント駆動型アーキテクチャでは、コンポーネント同士が直接やり取りを行う。
2. イベント駆動型アーキテクチャは、密結合によってリアルタイム性を向上させる。
3. イベント駆動型アーキテクチャでは、FaaSが多く活用されている。
4. イベントバスは、イベント駆動型アーキテクチャにおいて重要な役割を果たす。

Q6: CaaS（Container as a Service）についての記述のうち、正しいものはどれか？当てはまるものをすべて選択してください。

1. CaaSはサーバーレスコンテナとも呼ばれ、サーバーを不要とするコンテナである。
2. CaaSは通常FaaSよりも自由度が高い。
3. CaaSはコンテナとサーバーレスの併用により、サーバーレスのFaaSよりも管理の手間が少ない。
4. AWSのAWS Fargateは、CaaSの具体例である。

確認テスト4

Q7: オンプレミス環境で、cronによって毎朝10時に自動実行されるプログラムがある。このプログラムは、約30分程度かけて全社のセキュリティ状況をチェックする。AWSクラウドへ移行する際に、このプログラムをAWS Lambdaで実行することを検討している。次の記述のうち、正しいものをすべて選んでください。

1. FaaSは、「アップロードが終了した」などのイベントをトリガーに実行できるが、スケジュール実行には対応していない。そのため、Lambdaは利用できない。
2. このプログラムをそのままLambdaに移行するには、使用されているプログラム言語がLambdaでサポートされている必要がある。
3. 約30分の実行時間は長いため、Lambdaに適していない可能性がある。
4. LambdaではCronを利用できないため、このプログラムの移行には適していない可能性がある。

第五章 マイクロサービス

モノリシックとマイクロサービス

- クラウドネイティブなアプリケーションは、スケーラビリティや柔軟性を重視するため、マイクロサービスアーキテクチャを採用することが一般的です。

モノリシック（monolithic）
アーキテクチャ

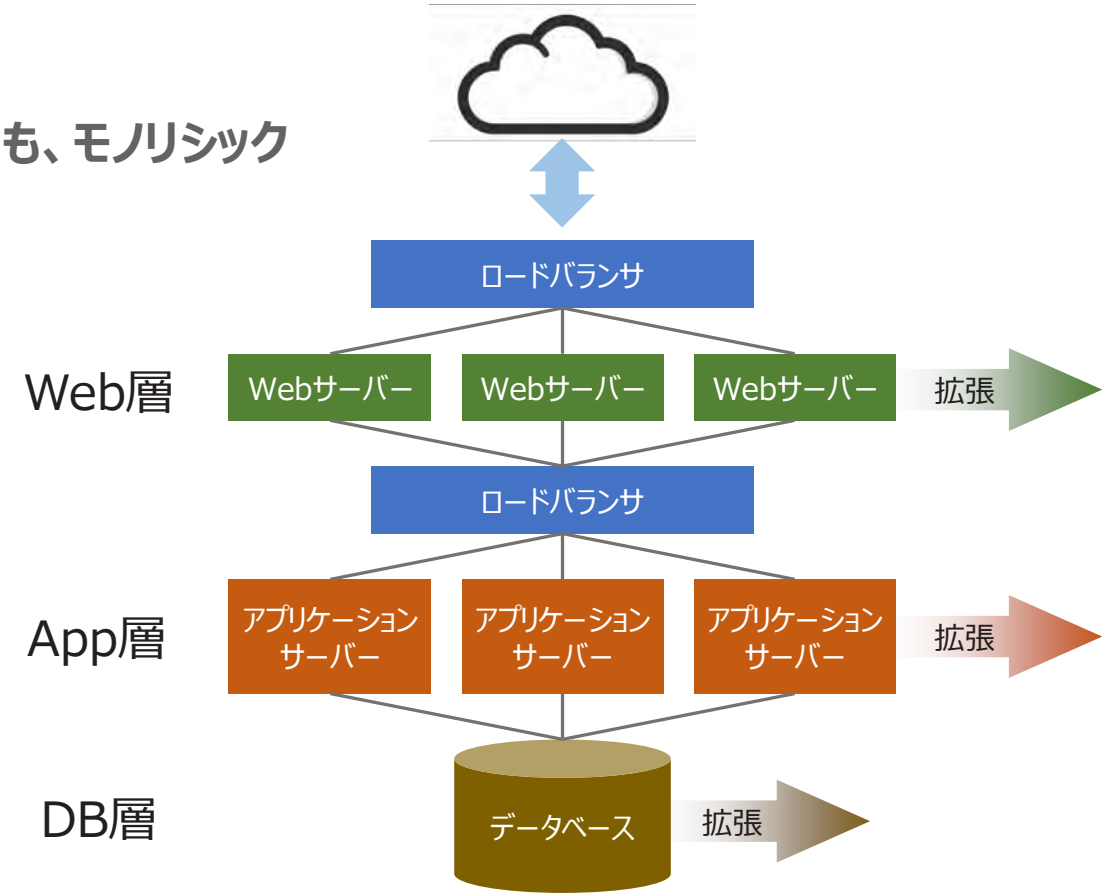
密結合

マイクロサービス
アーキテクチャ

疎結合

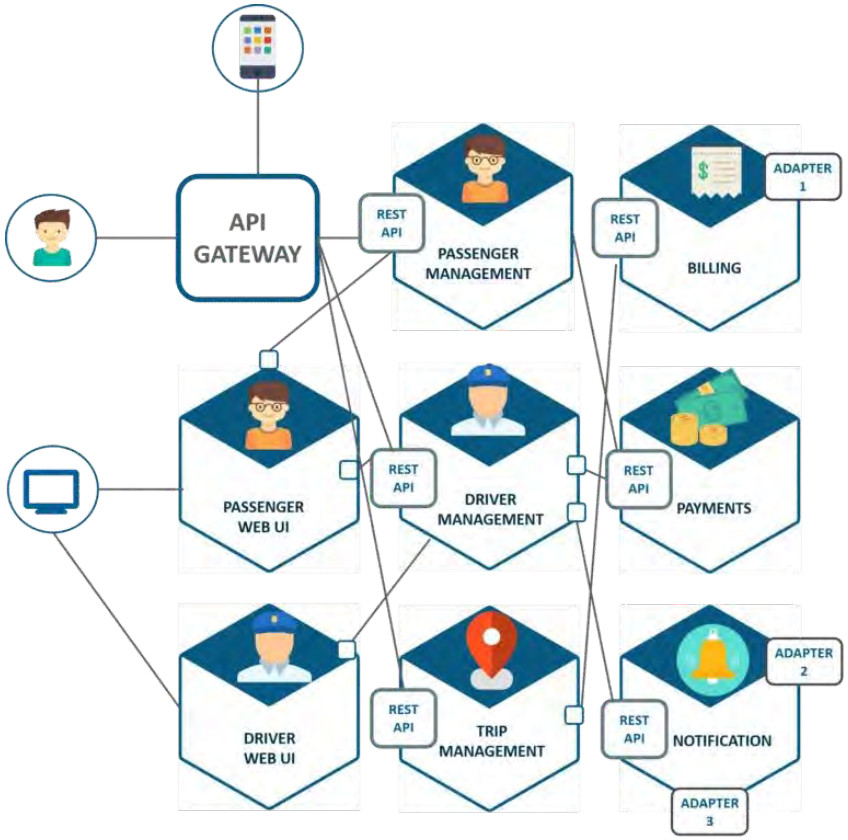
モノリシックアーキテクチャ

- モノリス、モノリシック：一枚岩
- 従来の3層構造のWebアプリケーションも、モノリシック



マイクロサービスアーキテクチャ

- **各サービスは独立して開発・デプロイが可能です。**
  - 疎結合：マイクロサービスでは、システム全体を複数の小さなサービスに分割
  - 異なるプログラミング言語を利用可能
  - それぞれ自分のスケジュールでリリース可能
  - 単一責任原則 (Single Responsibility Principle)：
    マイクロサービスは技術観点ではなく業務領域ごとに分割する
- **各サービスは独立して拡張/縮小可能です。**
  - 例えばアクセスが集中するサービスのみスケールアウト
  - コストの最適化やパフォーマンス向上実現
- **サービス間はリモートAPIやメッセージによる通信方法を採用します。**
  - REST APIやgRPC、メッセージキューなどを介して通信するため、
    疎結合が維持される
- **ほかのクラウドネイティブ技術と密に関係します。**
  - 小回りが利くコンテナやサーバーレスとの相性が非常に良い
  - 柔軟性の高い開発方法が求められるため、アジャイルとDevOpsの理念に合致
  - 内部通信が複雑になるため、従来の監視手法では対応できない。クラウドネイティブなオブザーバビリティが必要



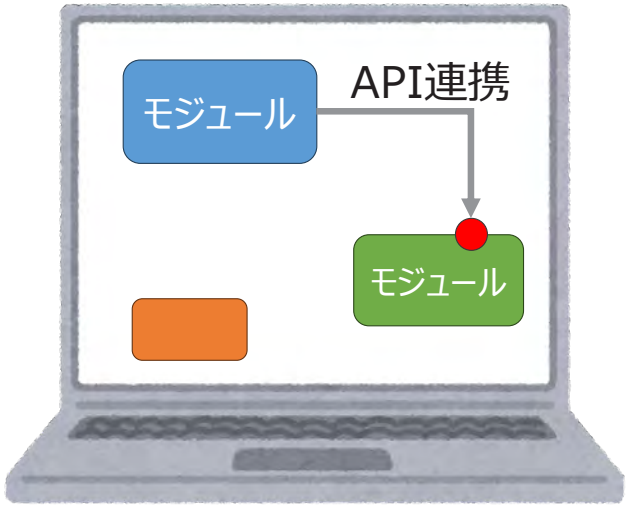
Sahiti Kappagantula (2018): Uber社のマイクロサービスアーキテクチャ

マイクロサービスの注意点

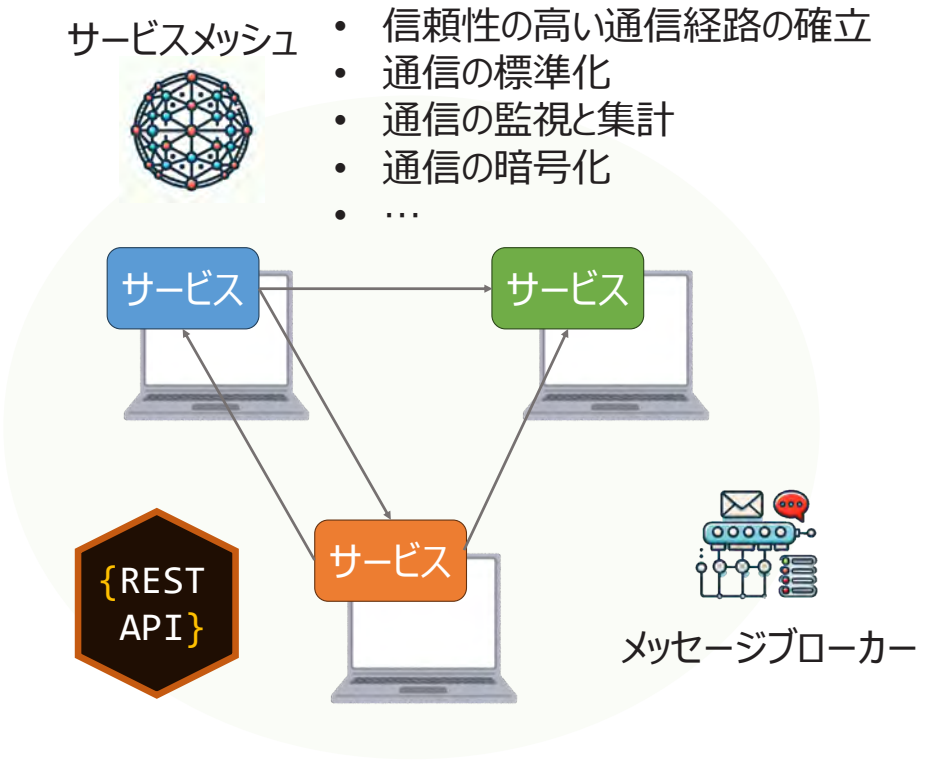
- **マイクロサービスは、クラウドネイティブのほかの要素に大きく依存するため、他の基盤が整っていない状態でマイクロサービスを導入すると、非常に高いリスクを伴います。**
  前提となるほかのクラウドネイティブ要素は：
  - アジャイルやDevOps文化の浸透
  - コンテナ基盤（Kubernetes）/サーバーレス技術の整備
  - CI/CDパイプラインやIaC環境の整備
  - オブザーバビリティ基盤の整備
- **コストに注意が必要です。**
  - KubernetesやCI/CD、オブザーバビリティ基盤など、前提条件が多いため初期コストが高い
- **サービス間通信がネットワーク経由となるため、ネットワーク障害の影響を受けやすいです。**
- **そのため、マイクロサービスは、クラウドネイティブという巨大パズルの最後のピースと言っても過言ではないかもしれません。**
  - 特にサービスメッシュを適用する、大規模なマイクロサービスの場合

マイクロサービス間通信

■ マイクロサービスは分散型システム



モノリシックの場合



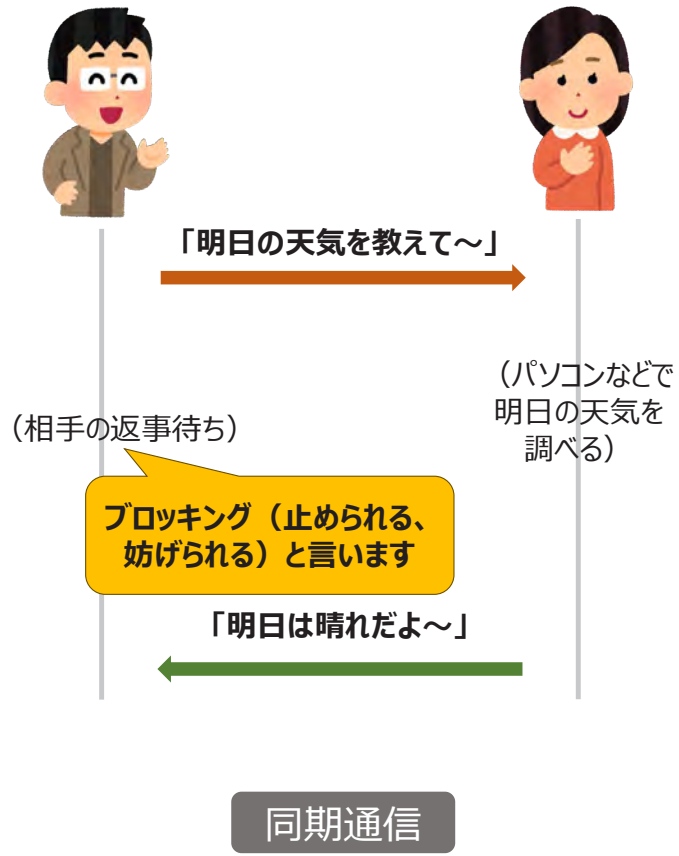
マイクロサービスの場合

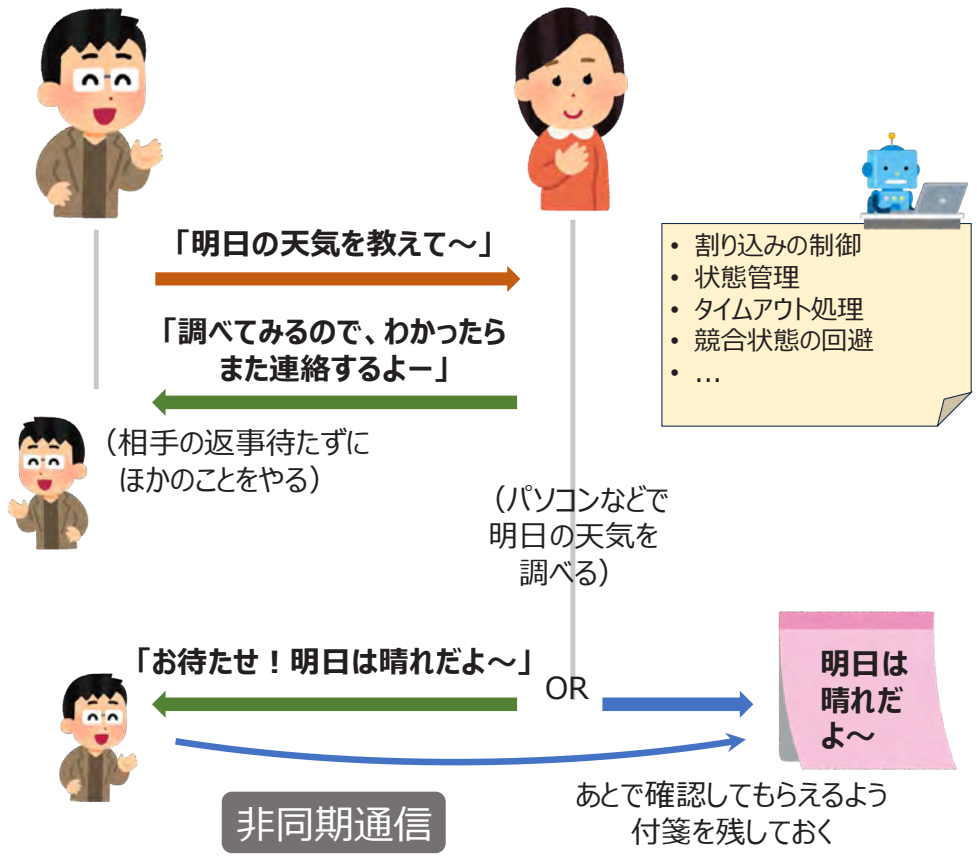
サービス間の通信方式

- ネットワーク経由のサービス間通信は、基本的に同期通信（ブロッキング通信）と非同期通信（ノンブロッキング通信）に分けられます。
- 通信パターンは、リクエスト/レスポンス型、イベント駆動型とデータ共有型があります。

* リクエスト/レスポンス型的な振る舞いを実現できるメッセージブローカーやメッセージキュー製品も存在します。
ここではあくまでも一般的な話としての分類になります。

同期通信と非同期通信


「明日の天気を教えて～」
（パソコンなどで
明日の天気を
調べる）
（相手の返事待ち）
ブロッキング（止められる、
妨げられる）と言います
「明日は晴れだよ～」
同期通信


「明日の天気を教えて～」
「調べてみるので、わかったら
また連絡するよー」
（相手の返事待たずに
ほかのことをやる）
（パソコンなどで
明日の天気を
調べる）
• 割り込みの制御
• 状態管理
• タイムアウト処理
• 競合状態の回避
• …
「お待たせ！明日は晴れだよ～」
OR
明日は
晴れだ
よ～
あとで確認してもらえるよう
付箋を残しておく
非同期通信

①リクエスト/レスポンス型

■ **マイクロサービスでは、同期通信が多く採用されています。**
同期通信は一般的にリクエストレスポンス型の方式で実装されます。

- 直感的で分かりやすい
- システムの設計と実装がシンプル
- 一対一の通信に適している
- ツールとエコシステムが充実（REST/gRPC関連ツールが多い）

＜参考情報：ほかのリクエスト/レスポンス型通信手段＞

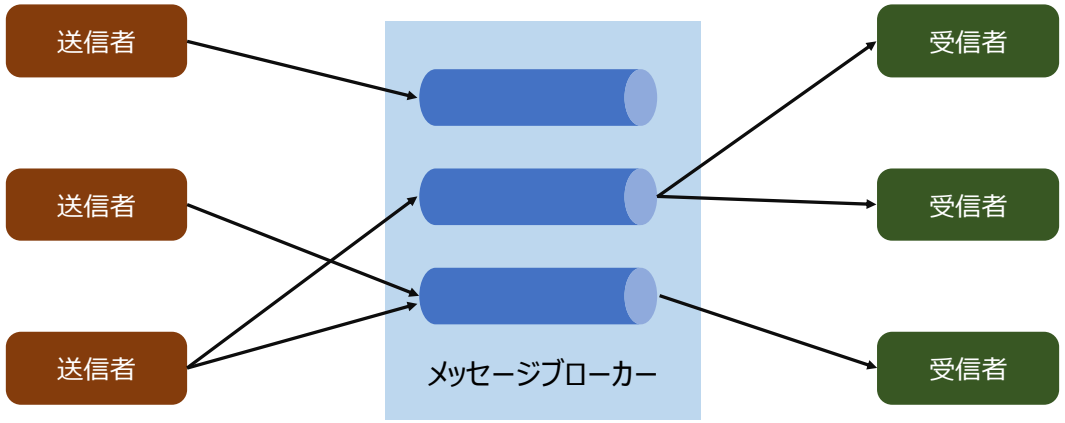
- **SOAP**：レガシー通信手段で複雑でオーバーヘッドが大きい。レガシーシステムではまだまだ現役
- **GraphQL**：リクエスト/レスポンス型だが、マイクロサービス間ではなく、フロントエンド-バックエンド間で利用
- **JSON-RPC**：シンプルで軽量。その反面、拡張の標準的な指針が乏しく、大規模システムでの採用が難しい

■ **リクエスト/レスポンス型でよく利用される通信手段は、REST（RESTful）とgRPCです。**

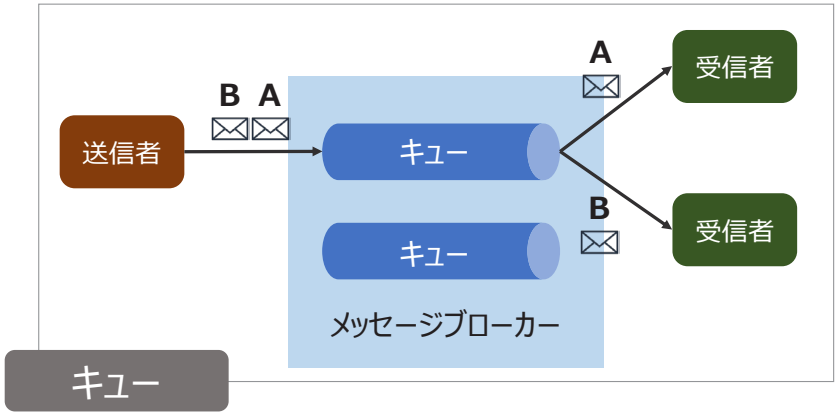
	REST	gRPC
ベースプロトコル	HTTP/1.1またはHTTP/2	HTTP/2
HTTPメソッドの利用	GET、POST、PUT、DELETEなどをそのまま利用	POSTで関数呼び出し
データ形式	テキスト（JSONやXMLなど）	バイナリ（Protocol Buffers）
エンドポイント	リソース指向	サービス指向
セキュリティ	HTTPSによって担保	HTTPS + mTLSを標準でサポート
通信モデル	主に同期通信で利用。 非同期通信でも利用可能だが作りこみが必要	標準で同期と非同期、ストリーミング通信をサポート

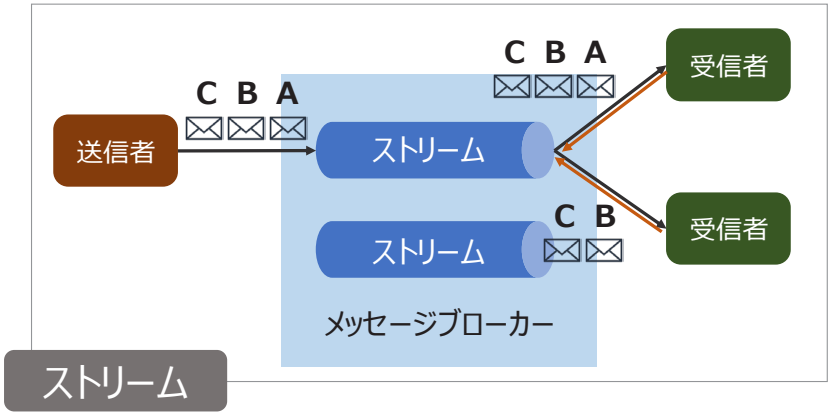
②イベント駆動型

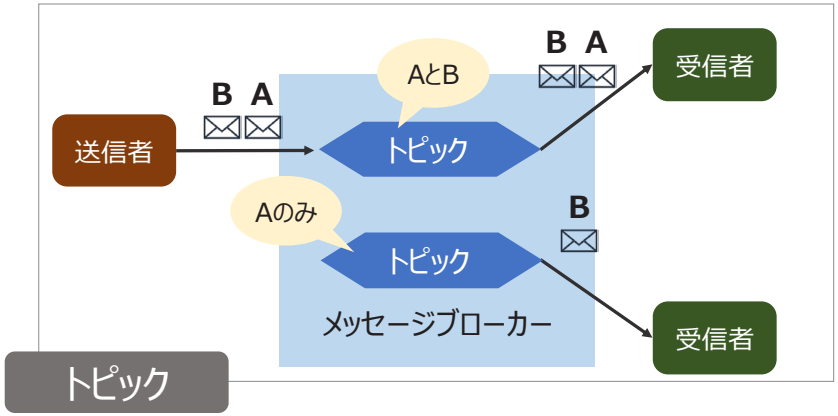
- イベント駆動型のアーキテクチャでは、送信者が直接受信者に接続しません。
「メッセージブローカー」（もしくはメッセージキュー、イベントルーター）と呼ばれる仲介役を介して通信します。
- 送信者は、Sender/Producer/Publisherなど、受信者は、Receiver/Consumer/Subscriberなど、ユースケースや製品、利用方法によって呼称が異なります。
- この仕組みを利用した通信は、「メッセージング」と言います。

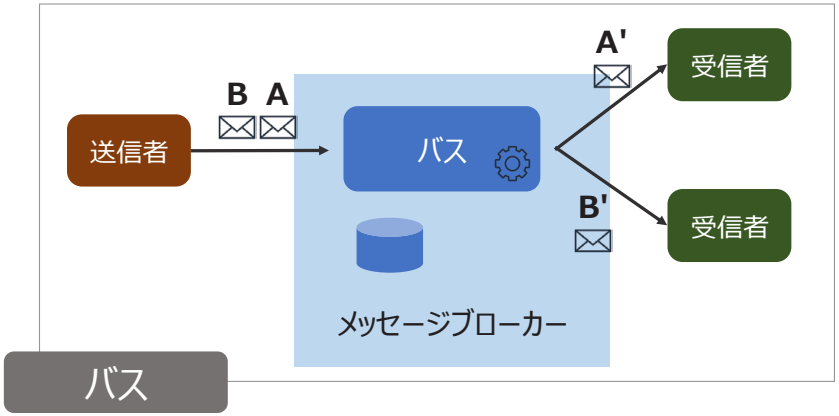


メッセージングの動作モデル

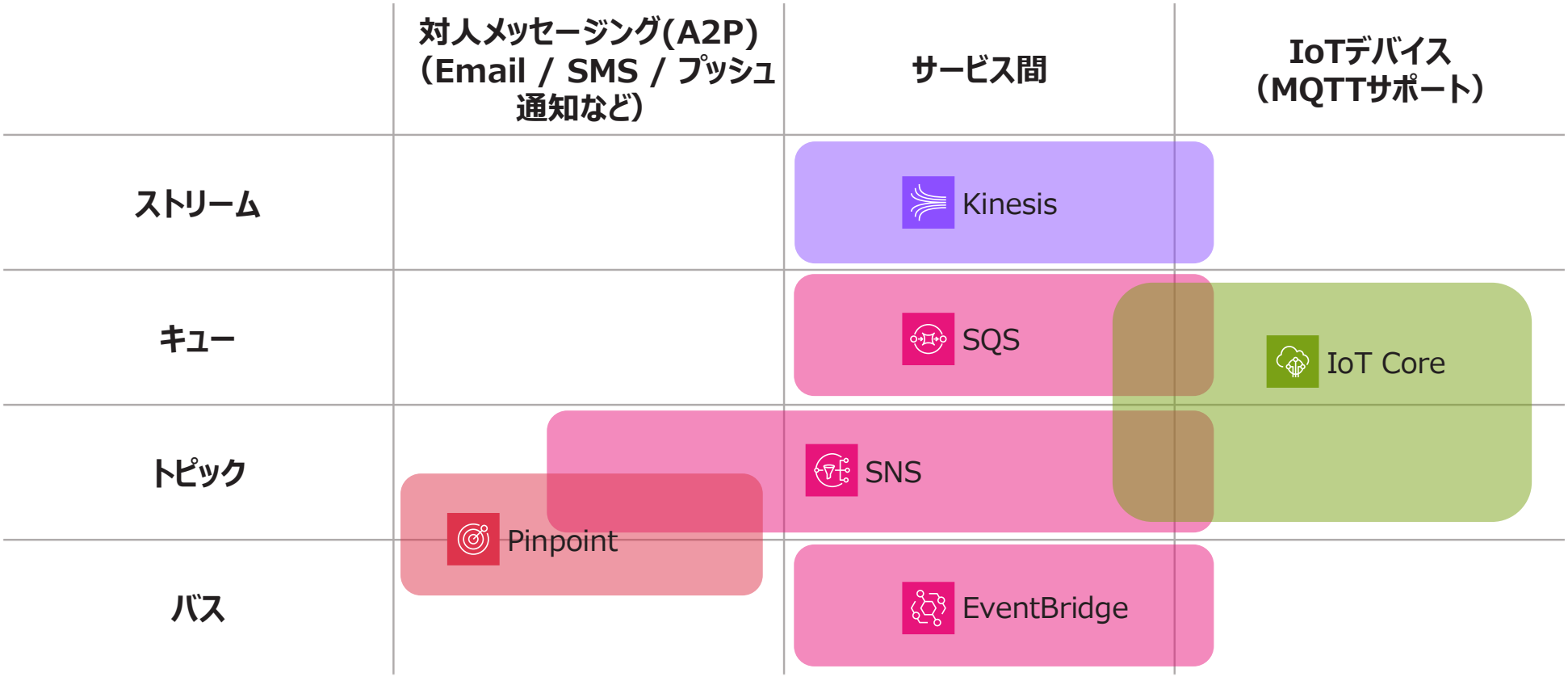








AWSが提供するメッセージングサービス（AWSネイティブ）



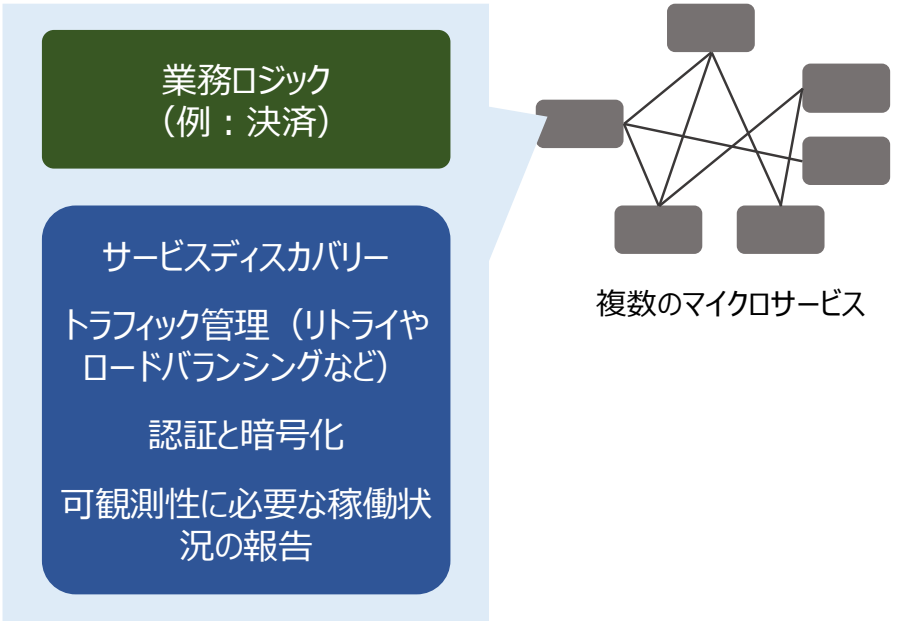
AWSが提供するメッセージングサービス（マネージドOSS）

③データ共有型

- **従来から利用されている方法です。ファイルサーバーやデータベース、FTPサーバーなどを利用して、送信者が通信したい内容を書き込んで、受信者が定期的に新規バージョンの存在を確認します。**
- **シンプルで成熟した技術を利用することで信頼性も非常に高いです。**
- **注意点・デメリット**
  - ポーリングが必要なのでリアルタイム性、効率性がやや低い
  - データ共有の仕組みそのもの（例えばファイルサーバー）の耐障害性、可用性を確保する必要がある
  - データベースやファイルサーバーは、本来サービス間通信のために設計されたものではないため、機能面や性能面で限界がある場合がある

サービスメッシュ：なぜ必要なのか？

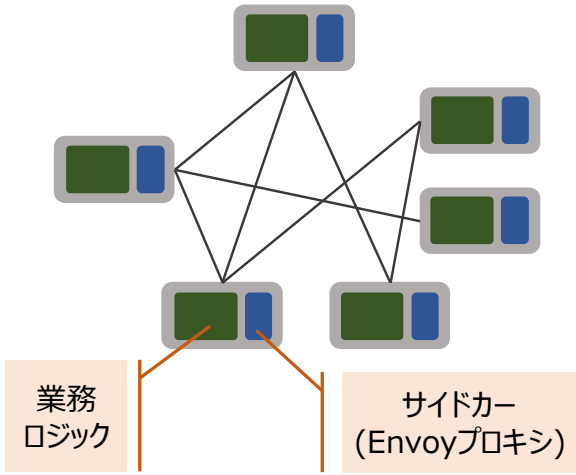
- **マイクロサービスを作成する際に、個々のマイクロサービスで業務ロジック（例えばECショップの場合、決済、在庫確認、出荷など）だけでなく、互いのマイクロサービスが通信するためのソースコードも作成しなければなりません。**
  - サービスディスカバリー（DNSの名前解決のような仕組み）
  - トラフィック管理（例：リトライ）
  - 認証と暗号化（例：mTLS）
  - 可観測性のための仕組み（例：Prometheus互換のメトリクスの提供）
- **マイクロサービスの数が増えると、この部分の管理運用が大変煩雑になります。**
- **そのため、サービスディスカバリー、トラフィック制御、認証、監視などを一元的に提供する仕組みが必要です。これが、「サービスメッシュ」です**



よく利用されるサービスメッシュ

Istio（イスティオ）

- Kubernetesで最も広く使われているサービスメッシュ
- 「サイドカー」構成：Envoy（エンヴォイ）と呼ばれる通信を肩代わりするプロキシをPod（マイクロサービスの稼働単位）に埋め込み、通信をEnvoyに経由させることで、トラフィック管理や認証などを実現

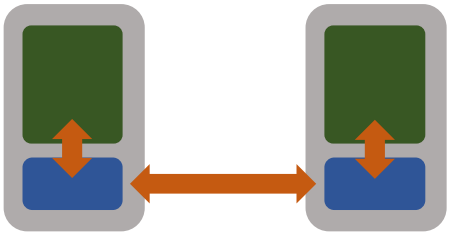


Cilium（シリウム）

- Ciliumという言葉は、細胞の表面にある微細な毛状の突起のこと（繊毛）を指す言葉です。コンテナを細胞に見立て、その表面にある繊毛から着想を得ており、ネットワークの接続性や観測を象徴しています。
- サイドカー不要で、LinuxカーネルのeBFP機能を利用します。

AWS App Mesh（サービス終了）

- AWSでは、同じくEnvoyプロキシを利用したサービスメッシュ「App Mesh」が存在していましたが、2024年9月にサービス終了が決まりました。既存ユーザーは2026年9月まで利用できます。
- 後続製品としては
  ECSクラスタの場合：ECS Service Connectの利用を推奨
  EKSクラスタの場合：
  ①IstioやCiliumなどのOSSサービスメッシュを利用
  ②サービスメッシュではないが、サービスメッシュに近いVPC Latticeを利用



API Gateway

マイクロサービスでの通信の2パターン

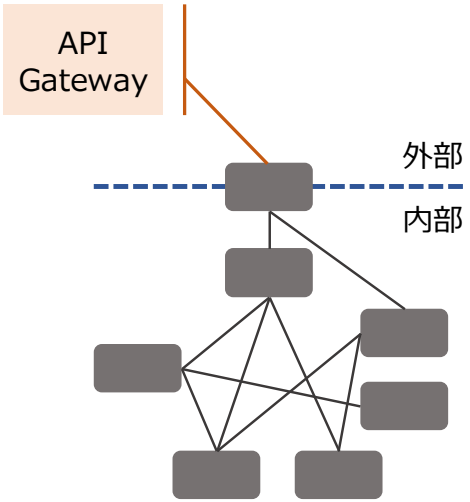
- East-West：水平方向の通信＝マイクロサービス間の通信
  →主に**サービスメッシュ**で対応
- North-South：垂直方向の通信＝外部（ユーザーなど）との通信
  →主に**API Gateway**で対応

API Gatewayの主な機能

- リクエストのルーティング：外部からのリクエストを適切なマイクロサービスに振り分ける
- フロントエンドと通信するためのプロトコル対応：REST, GraphQL, SOAP...
- 認証と認可、セキュリティ
- トラフィック管理

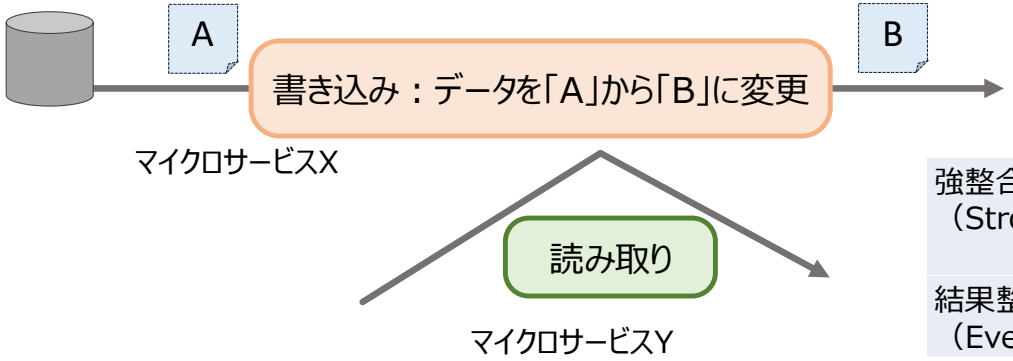
代表的な製品

- Kong (Kong API Gateway)
- Apigee（Google Cloudが提供する製品で、Google Cloud以外の環境でも利用可能）
- AWS上のAmazon API Gateway



マイクロサービスの最難関：結果整合性

- この章は、インフラ観点でマイクロサービスに必要な仕組みを紹介しました。
- しかし、アプリケーション観点やデータ観点で、マイクロサービスの設計開発に関してまだたくさんの考慮点が存在します。
- その中で、最も難しいとされるのは、データ同期・整合性です。
  - マイクロサービスでは、強整合性の保証が難しいため、**疎結合**、**非同期**、**結果整合性**が基本になる
- 各サービスに閉じたローカルトランザクションを非同期につないでワークフローを構成する手法「Sagaパターン」を利用して、結果整合を保証する実装が多いです。



強整合 （Strong Consistency）	マイクロサービスXの書き込みが完了するまでロックがかかり、書き込みが完了した後に「B」が読み取られる
結果整合 （Eventual Consistency）	ロックはかからず、書き込み開始前の「A」が読み取られる可能性がある

確認テスト1

Q1：「モノリシック」アーキテクチャについての記述のうち、正しいものはどれか？当てはまるものをすべて選択してください。

1. 「モノリシック」は「一枚岩のような」という意味で、一致団結してソフトウェアを開発する方法論である。
2. 「モノリシック」はスケーラビリティや柔軟性に優れる。
3. 「モノリシック」の場合、1か所でも不具合があると、全体が停止してしまうことが多い。
4. 「モノリシック」の場合、必要に応じて個別にスケールアップやスケールダウンが可能。

Q2：マイクロサービスを導入するには、前提となるのは次のうちどれか？当てはまるものをすべて選択してください。

1. アジャイルやDevOps文化の浸透
2. コンテナ技術やサーバーレス技術を利用した環境の整備
3. CI/CDパイプラインやIaC環境の整備
4. オブザーバビリティ基盤の整備

確認テスト2

Q3: 同期通信と非同期通信に関する記述のうち、正しいものをすべて選んでください。

1. 同期通信は「ブロッキング通信」とも呼ばれ、非同期通信は「ノンブロッキング通信」とも呼ばれる。
2. 同期通信ではリクエスト/レスポンス型が主流であり、RESTやgRPCがこれに含まれる。
3. データベースを介した通信があり、これは非同期通信に分類される。
4. イベント駆動型は、主に同期通信で利用される。
5. コンピューターの観点から見ると、非同期通信のほうが実装しやすい。

Q4: リクエスト/レスポンス型の通信についての記述のうち、正しいのはどれか？当てはまるものをすべて選択してください。

1. 密結合になってしまうため、極力利用しないほうが良い。
2. 利用する通信プロトコルは、RESTやgRPCなどがある。
3. RESTはHTTPでgRPCはHTTPSなので、gRPCのほうがよりセキュア。
4. RESTはテキストベースで、gRPCはバイナリベースのプロトコルなので、gRPCのほうが帯域の占用が少ない。

確認テスト3

Q5: AWSを使用してマイクロサービスアーキテクチャでシステムをメッセージングの動作モデルに含まれているのは、次のどれか？当てはまるものをすべて選択してください。構築している。サービス間の通信を実現するために利用できるAWSサービスとして、以下の中から当てはまるものをすべて選択してください。

1. キュー
2. バス
3. ストレージ
4. トピック

Q6: AWSを使用してマイクロサービスアーキテクチャでシステムを構築している。サービス間の通信を実現するために利用できるAWSサービスとして、以下の中から当てはまるものをすべて選択してください。

1. Amazon SQS
2. Amazon EventBridge
3. Amazon Pinpoint
4. Amazon MQ

確認テスト4

Q7: Istioについての記述のうち、正しいのはどれか？当てはまるものをすべて選択してください。

1. AWS上のサービスである。
2. Envoyプロキシなどのサイドカーを必要とする。
3. LinuxカーネルのeBPF機能を利用して通信を制御する。
4. 基本的にKubernetesで利用されている製品である。

Q8: 結果整合性の場合、データの読み取りに関する記述のうち、正しいのはどれか？最も適切なものを選択してください。

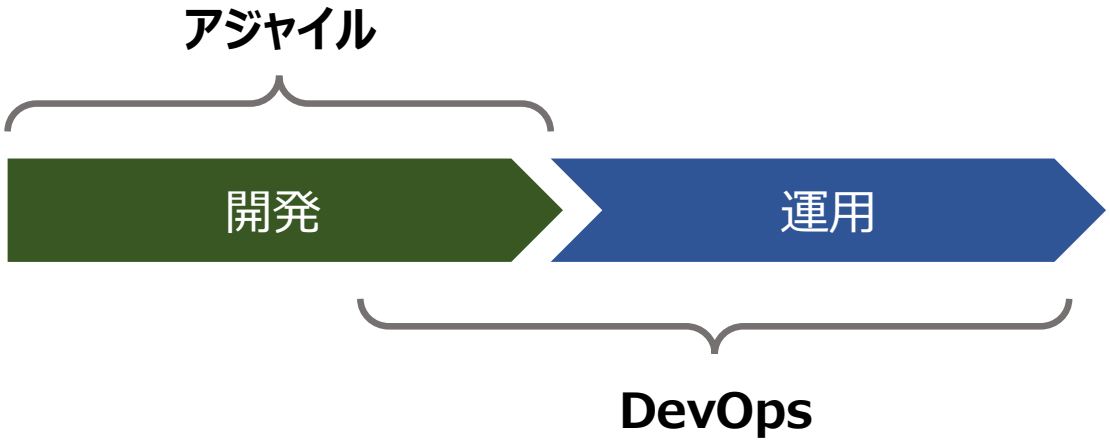
1. データを読み取る際に、もし他のサービスやコンポーネントによって書き込まれている最中であれば、それが完了するまでロックがかかり、最終的に変更後の値が取得される。
2. データを読み取る際に、もし他のサービスやコンポーネントによって書き込まれている最中であれば、ロックがかからず、変更前の値が取得される。
3. データを読み取る際に、もし他のサービスやコンポーネントによって書き込まれている最中であれば、ロックがかからず、変更前もしくは変更後の値が取得される。
4. データを読み取る際に、もし他のサービスやコンポーネントによって書き込まれている最中であれば、エラーが返される。

第六章 アジャイルとDevOps

第六章 アジャイルとDevOps

クラウドネイティブを支える「哲学」と「方法論」

- これまで、「クラウド」や「コンテナ」といった技術要素、そして「サーバーレス」や「マイクロサービス」といったアーキテクチャについて学んできました。
- この章では、技術的な側面ではなく、「哲学」や「文化」、「方法論」、「アプローチ」、あるいは「モデル」と呼ばれる、やや抽象的な概念に触れていきます。
- 具体的には、開発フェーズに焦点を当てた「アジャイル」と、運用フェーズや開発と運用の関係に焦点を当てた「DevOps」について学びます。



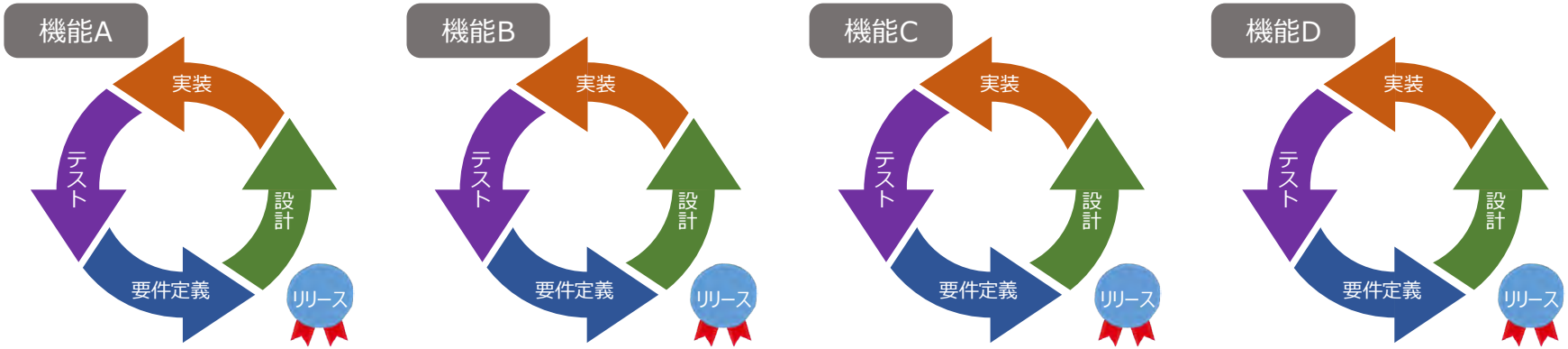
アジャイルとは

- **アジャイル、またはアジャイル開発は、迅速かつ柔軟にソフトウェア開発を進めるための手法および哲学です。まずは手法としての側面を見てみます。**
- **従来の「ウォーターフォールモデル」は、昨今のビジネスの不確実さ、変化の速さに対応できなくなっています。**
  - 「ウォーターフォールモデル」は、各工程が順序通りに進行する「段階的」かつ「線形的」なアプローチ
  - 最初の要件定義フェーズで、極力すべての不確実さをなくそうとし、開発途中の変更が発生しないという前提
  - 後続の設計実装フェーズで、変更があると手戻りが発生するため、原則変更を許容しない

ウォーターフォールモデル

アジャイルの「反復型のアプローチ」手法

- 従来のウォーターフォール型開発のように、要件定義、設計、開発、テスト、リリースといった工程を順番に進めるのではなく、短い開発サイクル（通常2～4週間程度）を繰り返すことで、継続的に価値を提供することを目指します。
- アジャイル開発では、ソフトウェアを短期間でリリースし、実際の使用状況に基づいてフィードバックを受け取りながら、次の開発サイクルに反映させます。この反復型のアプローチにより、顧客のニーズに迅速かつ的確に対応できる点が大きな特徴です。また、この方法は不確定要素の多いプロジェクトや要求の変化が激しいプロジェクトに特に効果的です。



アジャイルの開発手法

アジャイルの価値観

- **次はアジャイルの哲学、文化、マインドセットとしての側面です。**
- **アジャイルの価値観：アジャイルソフトウェア開発宣言（アジャイルマニフェスト）**
  - アジャイルの価値観は、2001年にアメリカやヨーロッパの17人の開発者が、従来のウォーターフォール型開発に代わる新しい開発アプローチを模索する中で生まれた「アジャイルソフトウェア開発宣言（アジャイルマニフェスト）」に由来します。
  - 宣言の中核となるのは、「**4つの価値**」と「**12の原則**」です。
  - 「4つの価値」は、左記（例えばプロセスやツール）のことがらを否定するわけではありません。左記のことがらに価値があることを認めながらも、右記のことがらにより価値を置く、という考え方です。
  - 詳細は、情報処理推進機構（IPA）が公開している「アジャイルソフトウェア開発宣言の読みとき方」という資料を参照してください：
    https://www.ipa.go.jp/jinzai/skill-standard/plus-it-ui/itssplus/ps6vr70000001i7c-att/000065601.pdf

4つの価値

プロセスやツールよりも「**個人と対話**」を

包括的なドキュメントよりも「**動くソフトウェア**」を

契約交渉よりも「**顧客との協調**」を

計画に従うことよりも「**変化への対応**」を

価値とする

“アジャイルソフトウェア開発宣言”
（アジャイルマニフェスト）より

スクラム（Scrum）開発

- **スクラム開発は最も広く使用されているアジャイル手法の1つで、フレームワークとしての構造が明確で取り入れやすいのが特徴です。**

- **スクラム開発の特徴**
  - アジャイルでの反復サイクル（イテレーション）は、スクラム開発では「**スプリント**（Sprint）」と呼ばれ、決められた期間内に動作する成果物を作成します。
  - 実装すべき要件や機能をリスト化して、**プロダクトバックログ**という一覧表に保管します。
  - スプリント開始時には「**スプリントプランニング（スプリント計画）**」を実施し、優先順位の高いプロダクトバックログアイテムをもとに、開発チームがスプリント内で達成可能な目標を設定します。
  - 毎日短時間（通常15分程度）のミーティング「**デイリースクラム**」を行い、進捗を共有します。
  - スプリント終了時には「**スプリントレトロスペクティブ（スプリント振り返り）**」を実施し、次のスプリントで改善すべき点を議論します。

アジャイルとスクラム開発の関係

1つのスプリント

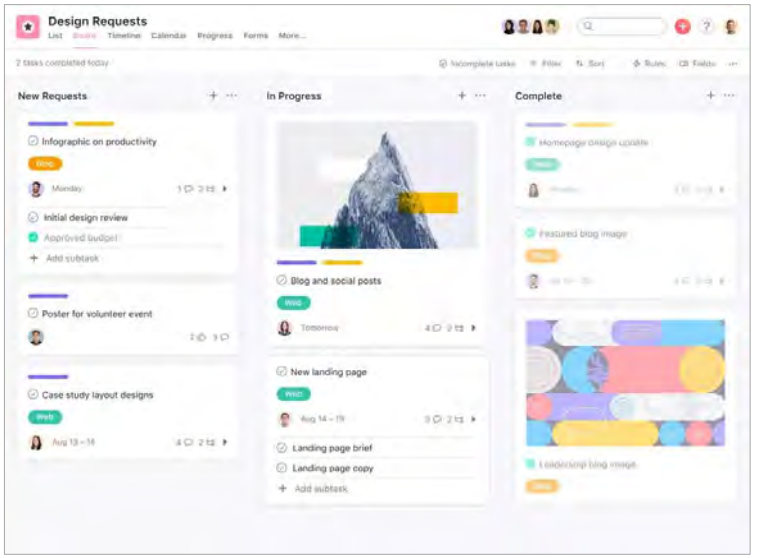
かんばん（カンバン方式）

かんばんについて

- かんばんも、アジャイル型から派生した手法です。
- もともとは、トヨタ自動車が考案した生産管理方式の一つで、「ジャストインタイム」と「リーン生産」という、生産計画・方式に基づいています。
- 「ビジュアル管理」を重視した手法で、特に進行中の作業量を最適化することを目的としています。
- IT業界では、ソフトウェア開発だけでなく、インシデント管理など運用フェーズでも広く活用されています。

スクラム開発との比較

- スクラムのようにルール化された会議はありません。
- スクラムより、かんばんのほうが柔軟性が高いです（スクラムでも、ウォーターフォールよりかなり柔軟ですが、スプリント途中の業務量変更などは難しいです）。
- かんばんでは、進行中の作業量（Work In Progress）を制限し、チームが過負荷にならないように管理します。

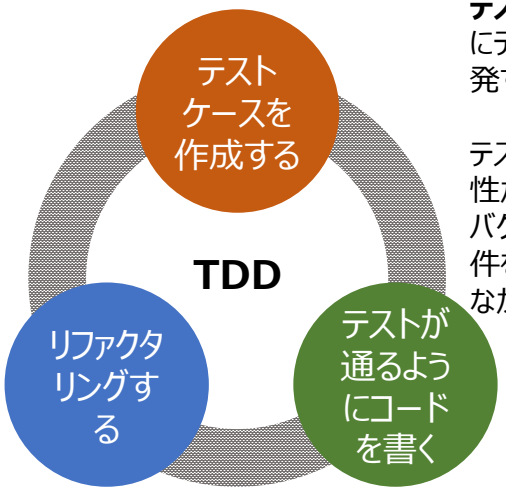


かんばんをサポートする製品の一例（Asana）

Asana公式サイトより引用：
https://asana.com/ja/uses/kanban-boards

アジャイル関連のプラクティス

- スクラム開発やかんばん以外にも、多種多様なアジャイル手法が存在します。これらの手法は排他的ではなく、むしろそれぞれの優れたプラクティスを状況やニーズに応じて柔軟に組み合わせて活用することが一般的です。
- ここではいくつかの例を紹介します。



テスト駆動開発（TDD）： コードを実装する前にテストを作成し、そのテストを通過するコードを開発する手法です。

テストが設計プロセスに組み込まれるため、保守性が高く信頼性のあるコードが得られます。また、バグの早期発見を促進します。テスト作成時に要件を具体的に考えることで、仕様の明確化にもつながります。

ペアプログラミング（通称：ペアプロ）：2人の開発者が1台のコンピュータを使い、コードの品質を高めます。また、少し似たようなコンセプトとして、**モブプログラミング（通称：モブプロ）**があります。3人以上のエンジニアが協力して1つのプログラムを作成する開発手法です。1台のコンピュータの前に集まり、役割を分担して作業を進めます。みんなで次々に意見を出し合い、ドライバー（操作する人）がその意見をコードに反映するスタイルです

DevOps

■「対立」から「統合」へ

従来の
開発チームと運用チーム

DevOpsでの
開発チームと運用チーム

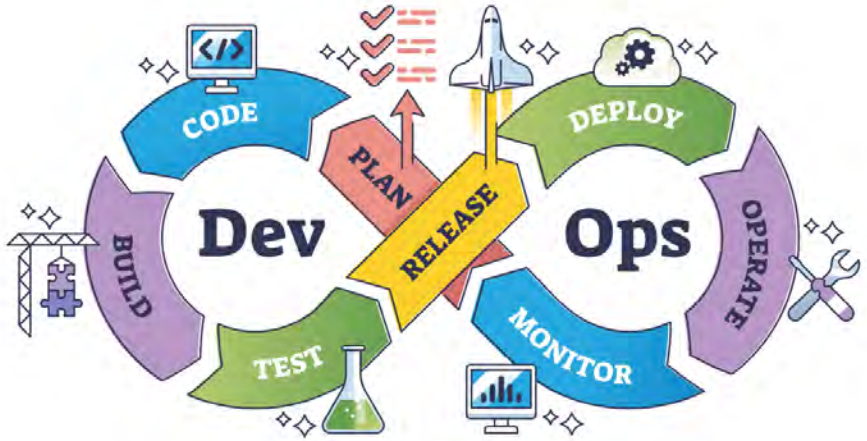
DevOpsとは

- **DevOps：アジャイルの原則をソフトウェアの展開および運用まで拡大したものです。**
- **DevOpsとは、ソフトウェアの開発（Development）と運用（Operations）を統合するアプローチおよび文化を指します。**
  - 技術的なツールだけでなく、プロセスと文化的な変革を含む包括的な概念です。

文化的な変革	DevOpsの成功には、部門間のサイロ化を解消し、コラボレーションを促進する文化的な変革が不可欠です。チームは共通の目標を持ち、責任を共有します。
プロセスの最適化	ソフトウェアの設計、開発、テスト、デプロイ、運用の各プロセスを統合し、効率化します。継続的インテグレーション（CI）や継続的デリバリー（CD）といった自動化の導入が代表例です。
技術的なツールの活用	自動化ツールやモニタリングツールを使用して、リリースや運用を効率化します。Git、Jenkins、Docker、Kubernetesなどがよく利用されるツールです。

DevOpsのシンボル

- DevOpsのシンボルとして「インフィニティループ」がよく使われます。
- DevOpsサイクル（計画 → 開発 → ビルド → リリース → 運用 → モニタリング → フィードバック → 再計画）を象徴的に表現しています。



DevOpsの原則

- **DevOpsの原則については、「3原則」「4原則」「5原則」「8原則」など、人によって異なる見解がありますが、基本的に共通しているのは以下の点です。**

原則	概要	それを支える技術、ツール例
自動化の推進	ソフトウェアライフサイクル（開発、リリース、監視など）のプロセスを自動化し、効率化を図る。	クラウド技術、コンテナ技術、CI/CDツール、インフラのコード化（IaC）ツール
コラボレーションとコミュニケーション	開発者が運用の責任を共有し、開発中や運用中の課題をチーム間で共有することで、部門の垣根を越えた連携を促進する。	Wikiや課題（issue）管理ツール、Slackのようなコミュニケーションツール、ChatOps
継続的な改善	プロセスやシステムを常に見直し、改善を続ける姿勢を持つ。	
顧客志向の考え方	最終的な顧客価値を重視し、ユーザーのニーズやフィードバックを迅速にプロダクトへ反映する。	
データ駆動型の意思決定	データに基づいて意思決定を行い、感覚や経験だけに頼らず、客観的な根拠を重視する。	モニタリングやログ解析ツール

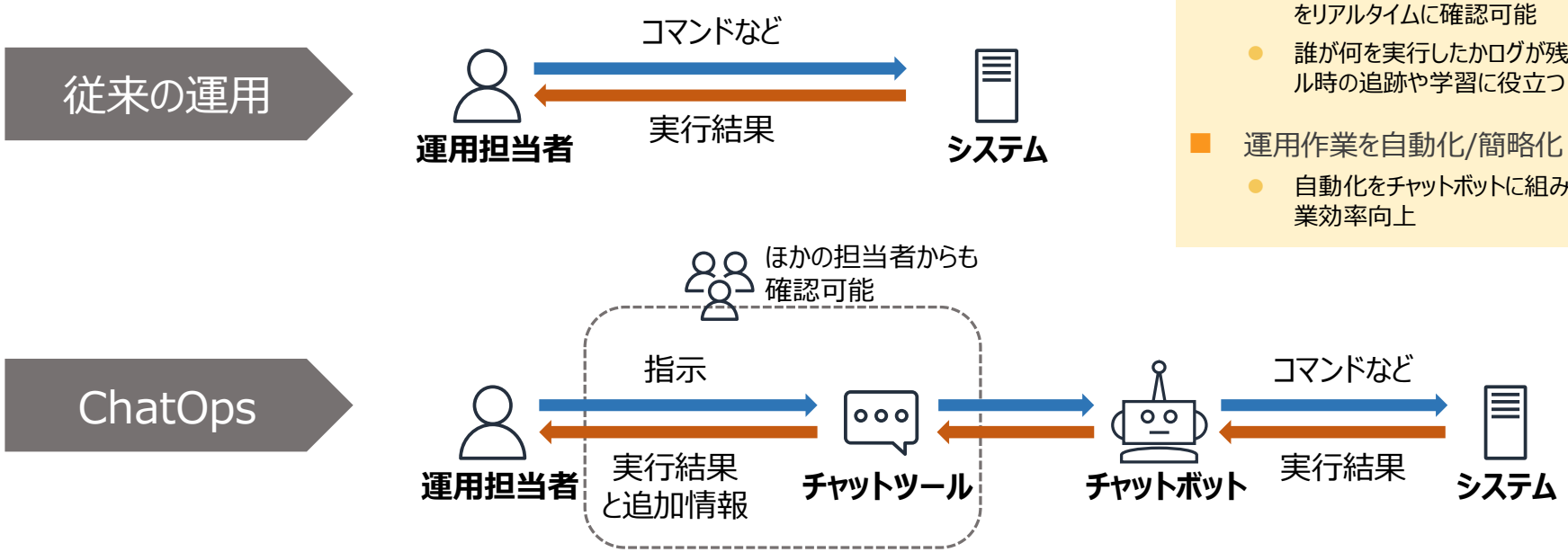
ほかに有名な5原則として、CALMSフレームワークがあります。CALMSとCulture（文化）、Automation（自動化）、Lean（無駄がない）、Measurement（測定）、Sharing（共有）の頭文字を組み合わせた造語です。

ChatOps

■ **ChatOps（チャットオプス）とは、チャットプラットフォームを活用して運用作業やチームのコラボレーションを効率化する手法です。コマンドや通知をチャット内で実行・受信します。**

■ 透明性の向上とコラボレーションの促進
- チーム全体が同じチャットルームで操作内容をリアルタイムに確認可能
- 誰が何を実行したかログが残るため、トラブル時の追跡や学習に役立つ

■ 運用作業を自動化/簡略化
- 自動化をチャットボットに組み込むことで作業効率向上



SREとDevOps

- **SRE（Site Reliability Engineering）は、Googleが提唱した運用モデルです。DevOpsを実現する具体的なフレームワークの一つと考えられています。**
  - C++言語風で記述すると（引用元：https://sre.google/workbook/how-sre-relates/）

```
class SRE implements interface DevOps
```

  - これは、直訳すると「SREはDevOpsというinterfaceを実装する」という意味になりますが、DevOpsの実現方法の一つがSREである、という意味を示唆する表現です。

- **DevOpsとSREは共通点が多い一方、SREは継続的な改善と測定可能な結果（特にSLOの使用を通して）を得ることに焦点を当てています。**

用語	説明	具体例
SLI	サービスレベルインディケータ。 サービスのパフォーマンスを定量的に測るための指標。	過去5分間のHTTPリクエストのエラー率が1%以下であれば、「可用」と定義する
SLO	サービスレベルオブジェクト。 SLIの目標値。SLOは運用チームやエンジニア間で合意されることが多く、サービスの目指すべき基準を設定する。	１年間の集計で、HTTPサーバーが「可用（SLIで定義）」である時間の割合が99.99%以上であること
SLA	サービスレベルアグリーメント。 サービス提供者と顧客間の合意事項。顧客に対して保証される目標値で、違反した場合にはペナルティ（例: 返金）が発生する可能性がある。	「可用」の目標を99.5%とする（*）。この目標を達成できなかった場合、当該サービス使用料金の40%を返金する

＊ SLAで設定する目標値は、必ずしもSLOと一致するわけではありません。SLOは、開発チームと運用チームが合意した内部目標値であり、サービスの健全性を維持するための基準として設定されます。一方で、SLAは、自社と顧客の間で合意した契約上の目標値であり、これが満たされない場合にはペナルティ（例: 返金）などの対応が発生することがあります。そのため、SLOは通常SLAよりも厳しい基準として設定されることが一般的です。

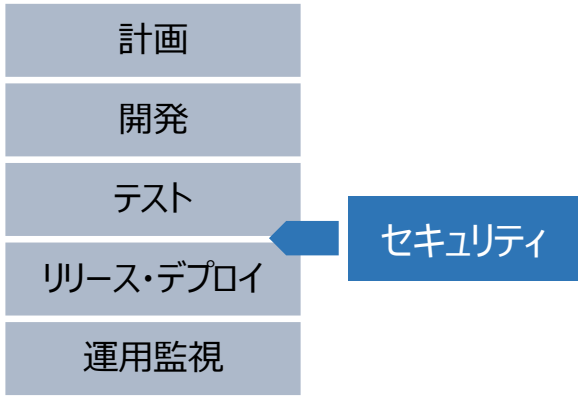
SREの考え方

SREのいくつかの考え方を紹介します。

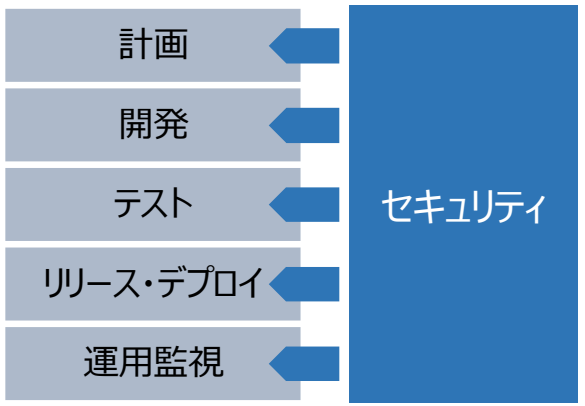
考え方	説明
トイルの削減	**トイル（Toil）**とは、自動化可能で価値を生み出さない反復的な作業（例：手動でのログ確認やスクリプト実行）を指します。SREでは、このトイルを可能な限り削減し、エンジニアリング活動に集中できるようにします。 例えば手動デプロイ作業をCI/CDで自動化することや、ログ収集・分析を自動化することなどです。
責任追及をしないポストモーテム	**ポストモーテム（Postmortem）**は、システム障害やインシデントが発生した後に、その出来事を振り返り、原因を特定し、再発防止策を立てるプロセスのことです。 障害が発生した場合、個人の責任を追及するのではなく、システムの改善に焦点を当てます。これにより、チーム内の心理的安全性を確保し、再発防止策を効果的に実施できます。
段階的なリリース	一度に大規模な変更をリリースするのではなく、小規模な段階的リリースを行い、リスクを軽減します。 例えば、カナリアリリース（Canary Release。少数のユーザーへの先行リリース）を利用することが一般的です。
SLOとエラーバジェットの活用	**エラーバジェット (Error Budget)** は、システムの可用性や信頼性に関する目標値（SLO）を達成する際に許容される失敗やダウンタイムの量を指します。 SLOを設定し、これを超えるエラーの許容範囲をエラーバジェットとして管理します。エラーバジェットを使い果たした場合、新機能開発より信頼性向上を優先します。
監視と可観測性	SREではSLOを重要視するため、SLO測定のために可観測性（オブザーバビリティ）の実現が不可欠です。SREの成功は、システムの可観測性を確保することによって大きく左右されます。そのため、可観測性を高めるツールやプロセスの導入は、SREにとっての優先事項となります。

DevSecOps

- DevSecOpsは、セキュリティをDevOpsのプロセス全体に組み込むアプローチです。
- 従来は、「開発の終盤でセキュリティをチェックする」という方法が一般的でしたが、DevSecOpsでは、開発プロセスの初期段階である左側から各タスクへセキュリティを組み込み、リスクの可視化と対処を早めることで”セキュリティのシフトレフト”を実現します。
- DevSecOpsを実現するアプローチには、「**セキュリティの自動化**」「**継続的なセキュリティ評価**」「**セキュリティの共有責任**」などがあります。



BEFORE DevSecOps



AFTER DevSecOps

確認テスト1

Q1: アジャイルでは「反復型のアプローチ」を提唱しています。その理由は次のどれか？最も適切なものを選択してください。

1. 同じことを反復することにより、不具合を極力減らすため。
2. 顧客のニーズや要求の変化に迅速に対応し、継続的に価値を提供するため。
3. カイゼンをモットーに、品質を向上させるため。
4. 複数の開発者の認識合わせのため。

Q2: ソフトウェアの開発フェーズだけでなく、運用フェーズもカバーしているのは次のうちどれですか？最も適切なものを選択してください。

1. アジャイルのみ
2. DevOpsのみ
3. アジャイルもDevOpsも、開発フェーズから運用フェーズまでカバーしている
4. アジャイルは開発フェーズのみ、DevOpsは運用フェーズのみカバーしているため、どちらもすべてカバーしていない

確認テスト2

Q3: アジャイルの価値観に含まれるのは以下のどれか？当てはまるものをすべて選択してください。

1. プロセスやツールよりも「個人と対話」を
2. 包括的なドキュメントよりも「動くソフトウェア」を
3. チームワークよりも「個々の能力」を
4. 計画に従うことよりも「変化への対応」を

Q4: スクラム開発における「スプリント」とは何か？最も適切なものを選択してください。

1. 開発のゴールを見える化したもの。
2. 開発における意思決定者のこと。
3. 開発チームの毎日の短時間のミーティングのこと。
4. 開発の反復サイクルのこと。

確認テスト3

Q5: 2人の開発者が1台のコンピューターを使用し、コードの品質を高めるために行う開発方法についての記述のうち、正しいものをすべて選んでください。

1. これは、「ペアプログラミング」という手法である
2. これは、予算が不足している兆しなので、予算を確保し、それぞれの開発者にコンピューターを与えるべき
3. これは、「モブプログラミング」という手法である
4. 一人がコードを書く（ドライバー）、もう一人がそれをレビューし、提案をする（ナビゲーター）という役割分担が一般的

Q6: ChatOpsについての記述のうち、正しいものをすべて選んでください。

1. チャット、つまり雑談しながら運用する、というカジュアルな運用スタイルである
2. 通常のチャットツール（SlackやTeamsなど）ではなく、専用のチャットが必要
3. チーム全体が同じチャットルームで操作内容をリアルタイムに確認できる
4. チャットボットに自動化を組み込むこともできる

確認テスト4

Q7: SREに関連する用語とその説明の組み合わせについて、正しいものは次のうちどれか？当てはまるものをすべて選択してください。

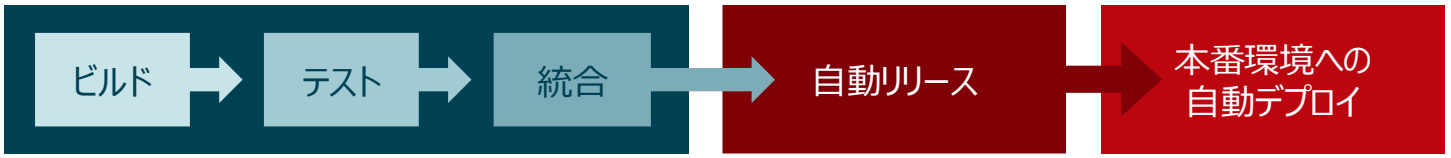
1. トイル：直訳すれば「労苦」であり、手作業、繰り返される、自動化が可能、戦術的、長期的な価値がない、サービスの成長に比例して増加する、といった特徴を持つ作業のこと。
2. ポストモーテム：直訳すれば「事後検証」であり、システム障害やインシデントが発生した後に、根本的な責任の所在を追求し、それぞれの個人やチームの責任の割合を明確にするためのプロセスのこと。
3. エラーバジェット：万が一障害（エラー）が発生した場合、どれくらいの損失が発生するかを定量的に見積もったものである。
4. SLO：サービスレベルの目標値で、システムの可用性を正確な数値目標として設定したもの。

第七章 CI/CD

第七章 CI/CD

CI/CDとは

- CI/CD (継続的インテグレーションおよび継続的デリバリー/デプロイメントの略) は、ソフトウェア開発ライフサイクルを最適化し、加速することを目的としています。
- 自動的かつ頻繁にインテグレーション、デリバリー（リリース）、デプロイメントを実施する手法です。
- CI/CDは、DevOpsの実現に欠かせない技術的な手段です。
  技術的な観点から見るとDevOps＝CI/CDと考える人も少なくありません。



CI (Continuous Integration)
継続的インテグレーション

CD (Continuous Delivery)
継続的デリバリー

CD (Continuous Deployment)
継続的デプロイメント

開発サイクルの後半戦

- 開発サイクルの前半（上流工程）では、要件定義や設計といった作業が中心となります。一方、後半（下流工程）では、主にテストやリリース作業が行われます。
- 従来のウォーターフォールモデルでは、単体試験、結合試験、総合試験、受け入れ試験といったフェーズを順番に進めていました。しかし、この手法では進捗に遅れが生じたり、バグの発見が後工程にずれ込むといった課題がありました。
- 現在ではアジャイルやCI/CDの導入により、これらの作業が継続的かつ効率的に進められるようになっています。

ユニットテスト
Unit Test
従来の単体試験。ソフトウェアの最小単位（関数やメソッド）の動作を確認するテスト。早期にバグを検出可能。ユニットテストでクリアしたコードのみ、インテグレーションの対象となる

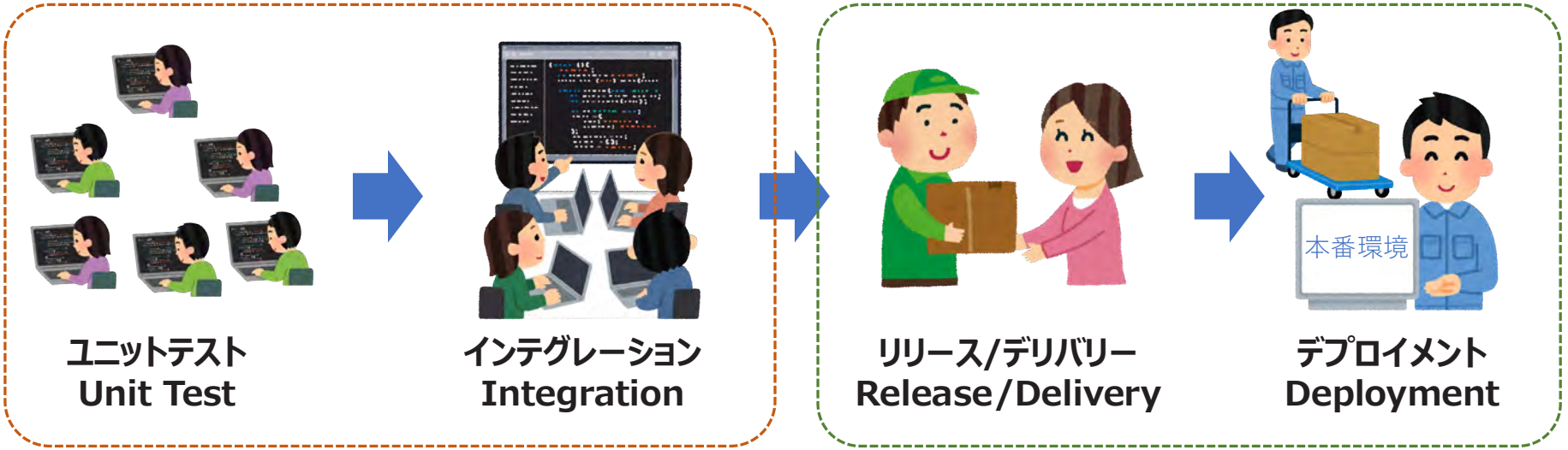
インテグレーション
Integration
従来の結合試験や総合試験のフェーズに該当する。複数のモジュールを統合し、データフローや連携機能を確認。依存関係や連携エラーを発見する

リリース/デリバリー
Release/Delivery
従来の総合試験 / 受け入れ試験のフェーズに該当する。完成したソフトウェアを運用環境へ準備するプロセス。品質を保証しユーザーに提供可能な状態にする

デプロイメント
Deployment
リリースされたソフトウェアを運用環境へ配置し、ユーザーが実際に利用できるようにする

CI/CDの対象

- CIは、ユニットテストからインテグレーションまでを対象とし、
  CDは、デリバリー、デプロイメントを対象とします。



CI (Continuous Integration)
継続的インテグレーション

CD (Continuous Delivery)
継続的デリバリー
CD (Continuous Deployment)
継続的デプロイメント

従来のテストからデプロイメントまで

テスト

- テストスクリプトや仕様書に基づいて、テスト担当者が実際にアプリケーションを操作し、期待される動作と照らし合わせて確認しました。
- バグを見つけるプロセスは経験と直感に頼る部分が多く、漏れが発生しやすいものでした。

コードの結合作業（インテグレーション）

- 各開発者が独自の環境でコードを作成し、定期的（多くの場合数週間ごと）に他の開発者のコードと統合しました。
- コードの統合後にのみ結合テストが行われるため、統合テストが開始される段階で大量のバグが発見されることがありました。

リリース（デリバリー）

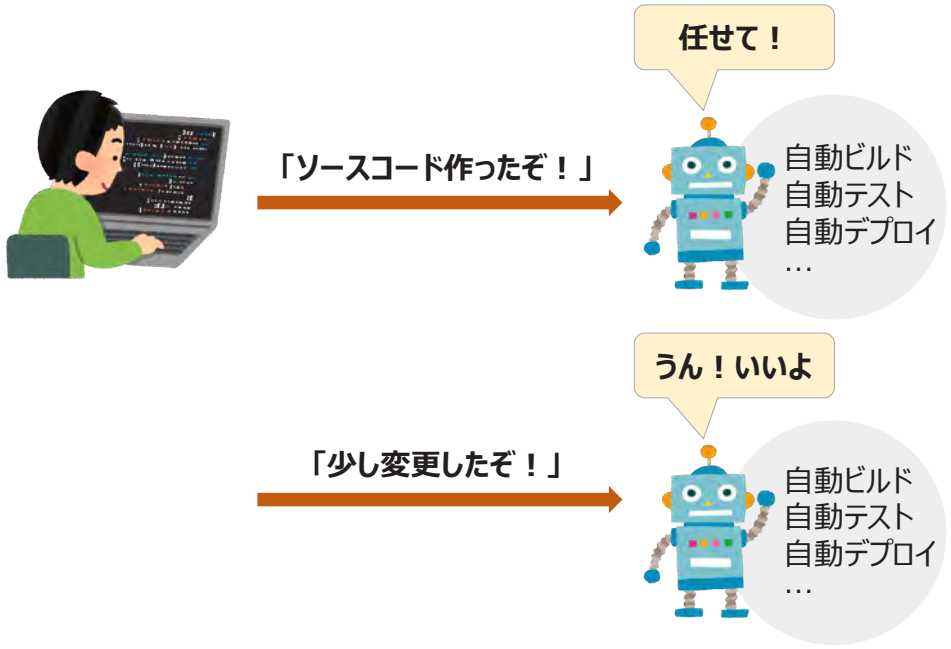
- 新しいバージョンをリリースするまでに数ヶ月から数年かかることが一般的でした。
- リリースノートの作成や、リリース準備のドキュメント作成も手作業で行われました。

デプロイメント

- サーバーへのコードのコピーや設定の変更、依存関係のインストールを手動で行いました。
- 「構築手順書」などに従い手作業で進めるケースが多く、シェルスクリプトやバッチスクリプトを利用することもありましたが、それも定型化されていない場合がほとんどでした。
- デプロイは一般的に深夜や業務時間外に行われ、システムがダウンタイムを伴うことが普通でした。

CI/CDでは

■ CI/CDでは、コード変更 → ビルド → テスト → デプロイのサイクルが自動的かつ継続的に実行されます。これにより、迅速なリリース、品質向上、チームの効率的な協力が可能になります。



■ テスト
- コードが変更されるたびに、テストが自動実行され、エラーを早期発見が可能になります。

■ コードの結合作業（インテグレーション）
- コードが変更されるたびに統合が行われるため、変更点が逐次反映されます。

■ リリース（デリバリー）
- 小さな変更を迅速にリリース可能となり、ユーザーのニーズに迅速に対応できます。

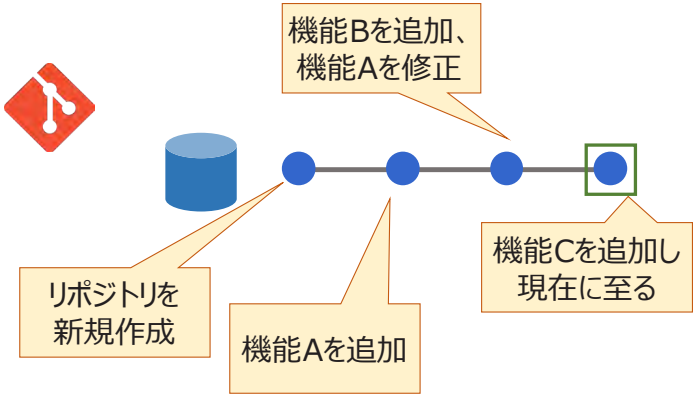
■ デプロイメント
- デプロイが自動化され、手作業のミスを防止できます。
- ダウンタイムの削減につながります。

第七章 CI/CD

Git

- CI/CD及び次の章で触れるIaCでは、「Git」と呼ばれるソースコードバージョン管理ツールが非常に重要な役割を果たしています。
- ソースコードは、万が一更新したら動作しなくなることに備えて、常に古いバージョンを保持する必要があります。また、複数の作業者が共同で編集することが一般的です。そのため、バージョン管理やデータの競合や重複を解決する必要があります。
- Gitは、このような「バージョン管理システム」です。すべての変更履歴を管理し、誰が、何を、いつ変更したかを記録します。

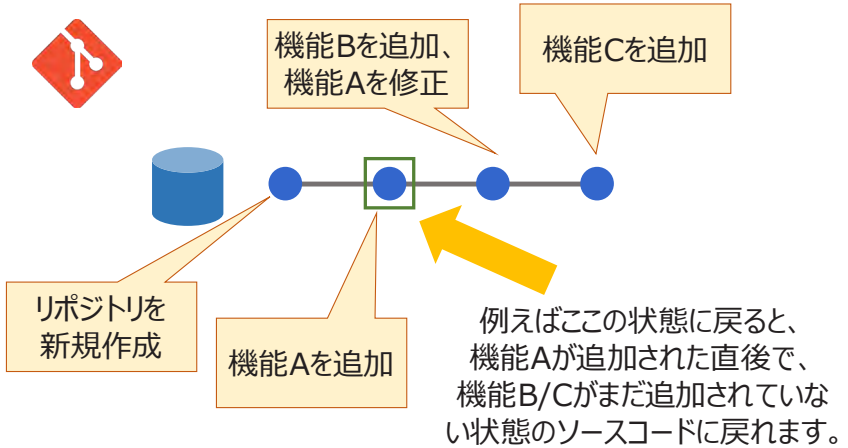
リポジトリとは、ソースコードなどのファイルを格納する場所を指します。一般的には「Gitが管理しているフォルダ」と捉えて問題ありません。

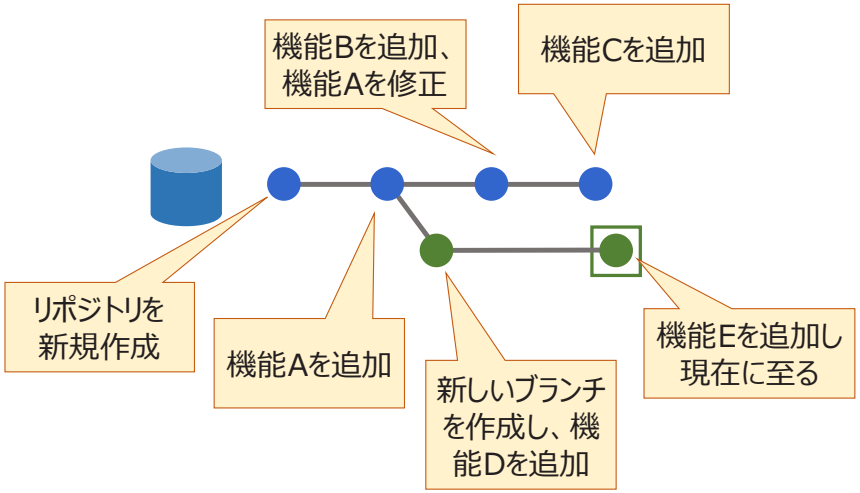


1つ1つの●は、**コミット（commit）**と言います。コミットは、Gitにおける「変更の記録」であり、ファイルの変更、新規作成、削除などを1つのまとまりとして記録します。この記録は履歴として残り、過去の状態に戻すことができます。

Git

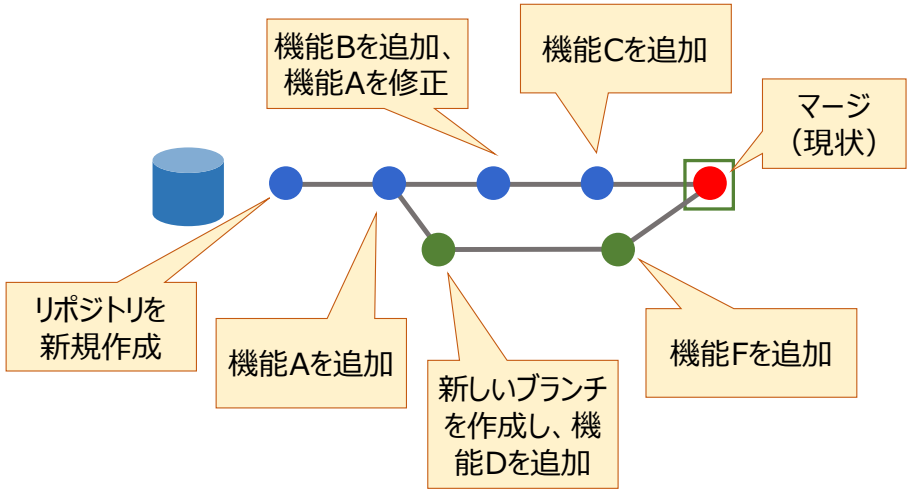
- Gitの変更履歴（コミット）は、後から参照可能で、過去の状態に戻したり、変更を取り消したりすることができます。
- Gitでは、プロジェクトのメインの作業とは別に「ブランチ（branch）」を作成して、独立した作業を行うことができます。
- 例えば、緑色のブランチの最新の状態では、機能A（修正されていない）、機能D、および機能Fを含むソースコードになっています。この状態には、機能Bと機能Cは存在しません。





Git

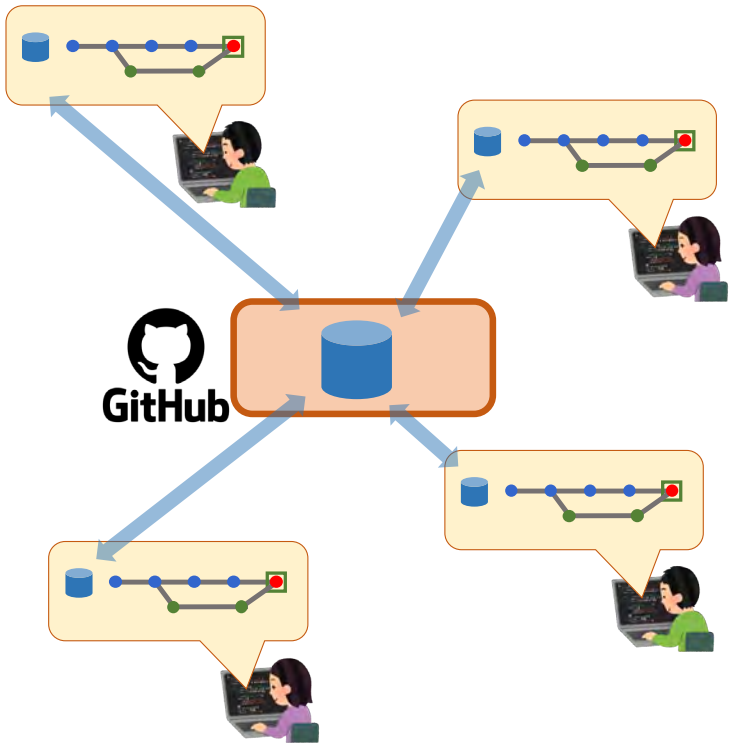
- ブランチをマージ（merge）することができます。ブランチをマージすると、それぞれのブランチに含まれる変更が統合されます。
- この例ではマージ後、機能A（修正を含む）、機能B、機能C、機能D、機能Eをすべて含むソースコードになります。



マージする際に**コンフリクト（競合）**が発生することがあります（例えば、2つのブランチで同じファイルを変更した場合）。コンフリクトは、手動で解決する、どちらか一方の変更を採用してもう一方を無視する、またはマージを中止することで対応できます。

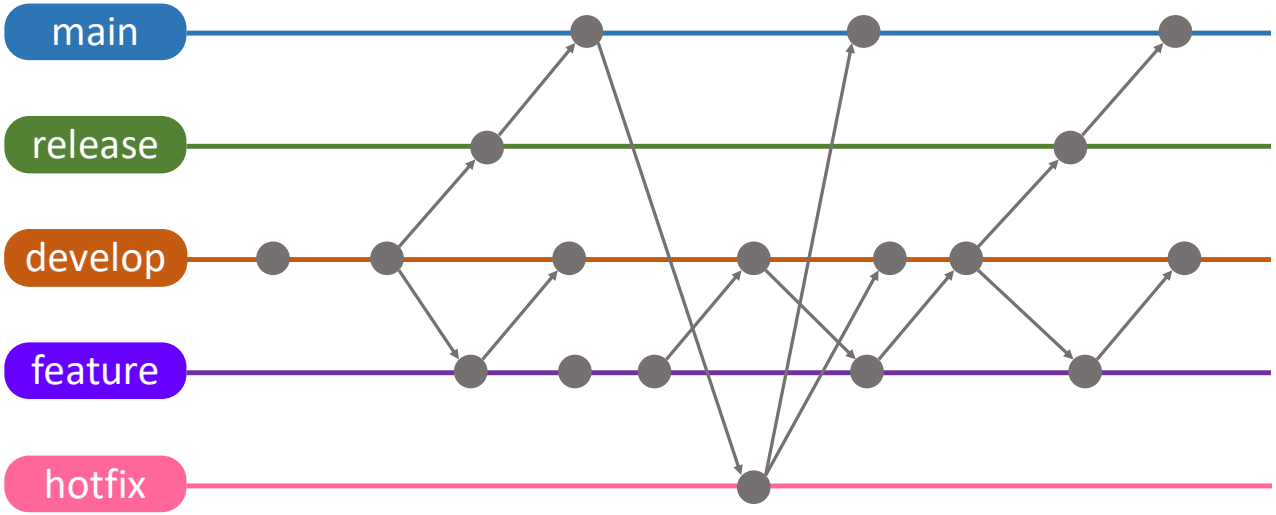
GitHub

- **GitHubは、Gitを利用したバージョン管理システムをクラウド上で提供するウェブサービスです。いわゆるGitリポジトリサービスです。**
  - コア機能は、「Gitリポジトリのサーバー」。ユーザーは、ローカルのリポジトリをGitHub上のリポジトリと同期できます。
  - コラボレーションのための機能。例えばプルリクエスト（Pull Requests）機能を使って、他の開発者に自分の変更を提案し、レビューしてもらうことができます。
  - 課題管理機能（イシュートラッキング）。バグや新機能の要求など、プロジェクトに関するタスクや問題（Issues）を管理できます。
  - コミュニティとフォーク機能。GitHubでは、他のユーザーのリポジトリを「フォーク」して、自分のコピーを作成できます。その後、変更を加えて、元のリポジトリにプルリクエストを送信することができます。この機能は、オープンソースコミュニティで非常に活発に利用されています。
- **GitHub以外にも、GitLab、BitbucketなどのGitリポジトリサービスがあります。**
- **クラウド上には、AWS CodeCommitやAzure Reposなどの Gitリポジトリサービスもあります。**



CI/CDのためのブランチ戦略

- Gitのブランチは非常に自由度が高い反面、無秩序に利用すると管理が複雑になります。そのため、ブランチの管理や利用に関するルールを策定する必要があります。これがブランチ戦略です。
- 有名なブランチ戦略の一つにgit-flowがあります。しかし、git-flowが誕生したのは2010年で、自動化やCI/CDがまだ普及していなかった時代です。そのため、現在ではやや重厚すぎると感じることもあります。一方で、金融など基幹業務システムの開発や自動テストが難しいシステムにおいては、今でも重宝されています。



本番環境にリリースするためのブランチ。常にリリース可能な状態を維持

リリースの準備のためのブランチ。ステージング環境にデプロイすることが多い

開発中の主軸となるブランチ。最新の開発状態が反映される

developブランチから分岐し、個別の新機能や改修作業を行うためのブランチ

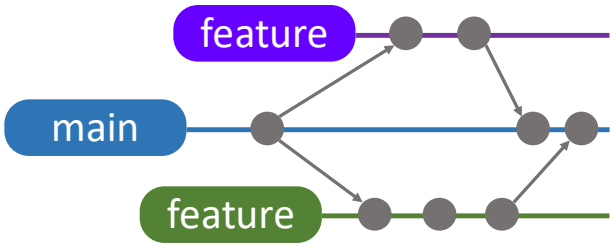
本番環境で発見された緊急のバグ修正を行うブランチ。修正作業完了後masterとdevelopにマージ

CI/CDのためのブランチ戦略

- モダンなWebアプリケーションなどの開発では、CI/CDによる完全な自動化を前提としたブランチ戦略を利用することが多くなっています。
- 特に、マイクロサービスアーキテクチャのように機能が小さく分割されている場合、git-flowのような重厚な戦略は非効率的です。CI/CDによって品質が担保されているのであれば、多数のブランチを使用する必要はなく、よりシンプルなブランチ戦略が適しています。

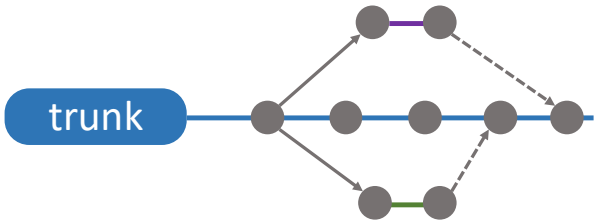
GitHub Flow

mainブランチを常にデプロイ可能な状態に保ち、機能ごとに短命のフィーチャーブランチを作成して作業します。作業が完了してmainにマージされると、通常は自動的に本番環境にデプロイされます。



トランクベース開発
Trunk-Based Development (TBD)

全開発者が単一のメインブランチ（トランク/trunk）に頻繁にコードを統合するブランチ戦略です。フィーチャーブランチは短命で、数時間～数日以内にトランクへマージします。未完成の機能はフィーチャーフラグを使用して制御し、本番環境へのデプロイを安全に行います。

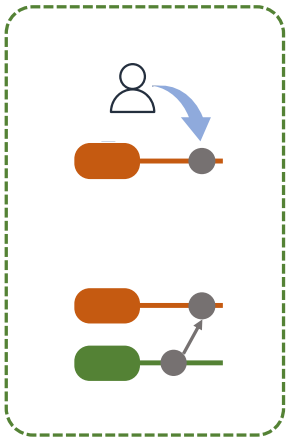


CIで実現できること

- **CI（Continuous Integration、継続的インテグレーション）では、具体的にどの作業が自動化されるかは使用するCIツールによって異なりますが、一般的には以下の作業が含まれます。**

一般的には、ユーザーがコミットを行った際や、ブランチがマージされる直前（プルリクエスト時）に自動的に実行されます。

その他、ファイルの変更や外部イベント（Webhook）をトリガーに実行すること、手動での実行やスケジュールに基づいた実行も可能です。



- コミットメッセージのチェック
- コーティング規約が守られているかをチェック
- フォーマットの確認と修正（インデントなど）
- 静的コード解析（Linting）

- 自動ビルド（ビルドが必要な言語のみ）
- 依存関係の管理、自動更新

- 静的セキュリティチェック（SAST）

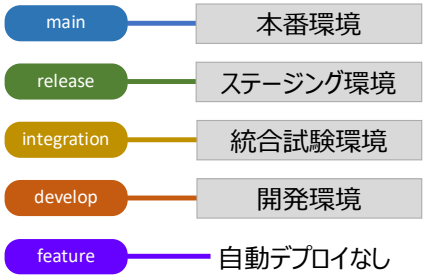
- ユニットテスト
- インテグレーションテスト
- 状況に応じた各種テストの実施（スモークテスト、回帰テスト、性能テストなど）

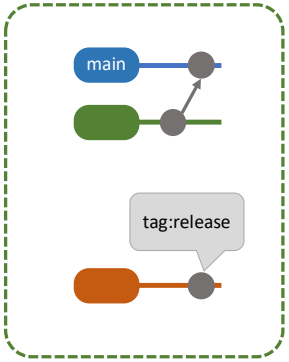
CDで実現できること

■ CD（Continuous Delivery、継続的デリバリーもしくはContinuous Deployment、継続的デプロイメント）では、具体的にどの作業が自動化されるかは使用するCIツールによって異なりますが、一般的には以下の作業が含まれます。

一般的には、mainブランチにマージされた際やリリースのタグが付与された際に自動的に実行されます。

また、それぞれのブランチにマージされた場合、自動的にそのブランチに対応している環境にデプロイすることもあります（git-flowなど）。





- リリース前の最終チェック

- バージョン管理
- タグ管理
- Feature Toggles/Feature Flags対応

- ソースコードからドキュメントの自動生成

- コンテナ化（コンテナイメージの作成）
- コンテナレジストリへの登録
- 各種成果物のアップロード

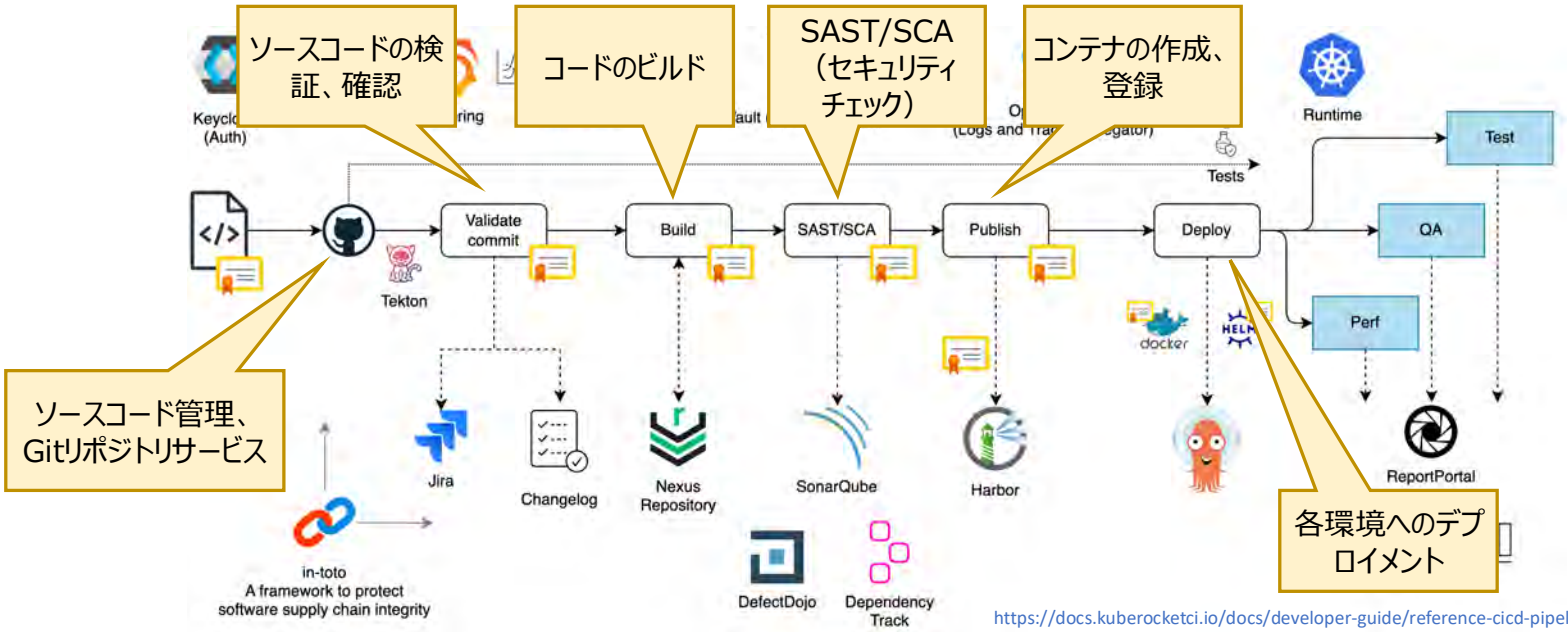
- 環境への自動デプロイ
- デプロイ戦略（カナリアリリース、ブルーグリーンデプロイメント…）
- デプロイ後の動作確認

Continuous Delivery、継続的デリバリー

Continuous Deployment、継続的デプロイメント

CI/CDパイプライン

- CI/CDは多くの作業を自動的に実行しますが、これらは一定の順序に従い、自動化されたワークフローとして実行されます。このワークフローをCI/CDパイプラインと呼びます。
- 下記は、Kubernetes環境のCI/CDツールのパイプラインの実際の例です。



https://docs.kuberocketci.io/docs/developer-guide/reference-cicd-pipeline

CI/CDツール（オンプレミス）

■ オンプレミスで多く利用されているCI/CDツールとして、Jenkinsがあります。

- オープンソースで無料：Jenkinsは完全にオープンソースであり、商用のライセンス費用は発生しません。また、コミュニティが活発で、無料で利用可能なリソースが豊富です。
- パイプライン定義：Jenkinsでは、ビルド、テスト、デプロイメントの流れを「**Jenkinsfile**」という設定ファイルで定義できます。これにより、パイプラインの再利用性が高まり、コード管理が容易になります。
- 分散ビルド：複数のサーバーでビルドを並行して実行することができ、大規模な開発環境にも対応可能です。
- 柔軟性とカスタマイズ性：Jenkinsは設定やスクリプトを自由にカスタマイズできるため、特定の開発プロジェクトに最適化したCI/CDプロセスを構築することができます。
- プラグインの豊富さ：Jenkinsは数百種類のプラグインが存在し、さまざまなビルドツールやソースコード管理ツール、デプロイメントツールなどと統合できます。これにより、個別のニーズに合わせて柔軟に拡張できます。

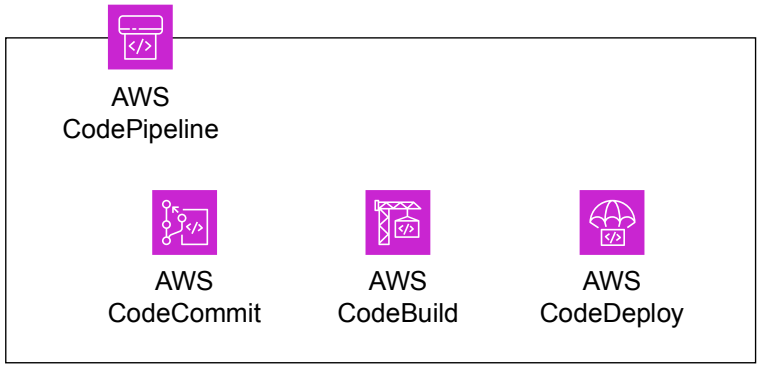
CI/CDツール（GitHub Actions）

GitHubリポジトリとシームレスに統合されたCI/CD：GitHub Actions

- クラウドベースのCI/CDツール：AWSやAzureなどのパブリッククラウドやオンプレミスから利用できます。
- シンプルな統合：GitHubリポジトリと直接統合されているため、他の外部ツールを使用せずにCI/CDの設定や管理ができます。GitHub ActionsはGitHubのネイティブな機能として利用でき、開発者がGitHubの環境内でスムーズに作業できます。
- 豊富なアクションライブラリ：GitHub Actionsには公式およびコミュニティ製の数千のアクション（定義済みのタスク）が存在し、ビルド、デプロイ、通知、テストの実行など、さまざまな処理を簡単に追加できます。これにより、外部ツールやスクリプトを少なくして、パイプラインの開発を迅速化できます。
- YAMLベースのワークフロー定義： GitHub Actionsでは、パイプラインをYAMLファイルで定義します。これにより、構成ファイルがバージョン管理され、チーム全体で共有・変更できます。
- マトリックスビルド：同じワークフローで複数の環境（異なるOS、異なるランタイム）で並列ビルドを実行できます。これにより、クロスプラットフォームのテストを効率よく行えます。

CI/CDツール（AWSクラウド）

- **AWSのCI/CDツールは、複数のツール/サービスから構成されています。**
  - CodeCommit：Gitリポジトリサービスです。
  - CodeBuild：CIツールです。
  - CodeDeploy：CDツールです。
  - CodePipeline：CodeCommit/CodeBuild/CodeDeployを組み合わせて、パイプラインを構築するツールです。CodeCommitの代わりに、GitHubなどを利用することも可能です。



確認テスト1

Q1: CI（継続的インテグレーション）の対象に含まれる工程は次のどれか？当てはまるものをすべて選択してください。

1. ユニットテスト
2. インテグレーション
3. リリース/デリバリー
4. デプロイメント

Q2: Gitのコミットとは次のうちのどれか？最も適切なものを選択してください。

1. ソフトウェアの最終発行
2. ソースコードの保存場所
3. ソースコードの変更の記録
4. 複数のソースコードの競合

確認テスト2

Q3: Gitにおいて、マージの際にコンフリクトが発生した場合、次のどの対処ができるか？当てはまるものをすべて選択してください。

1. マージを中止する
2. 両方の変更を自動的に取り入れる
3. 手動で解決する
4. どちらか一方の変更を採用し、もう一方を無視する

Q4: AWSが提供するAWSクラウド上のソースコード管理サービス、つまりGitリポジトリは、次のどれか？最も適切なものを選択してください。

1. AWS CodeDeploy
2. AWS CodeBuild
3. AWS CodePipeline
4. AWS CodeCommit

確認テスト3

Q5: Gitのブランチ戦略についての記述のうち、正しいのはどれか？正しいものをすべて選択してください。

1. どの担当者が、どのブランチを担当するかという組織的な戦略である。
2. CI/CDがそこまで自動化されていない基幹業務システムにおいて、git-flowを採用したのは適切である。
3. CI/CDがまったく整備されていないが、小型プロジェクトのためGithub Flowを採用したのは適切である。
4. マイクロサービスを開発している。CI/CDが整備されている。この場合TBD（Trunk-Based Development）を採用するのが適切である。

Q6: CIが実現することに含まれるのは、次のどれか？当てはまるものをすべて選択してください。

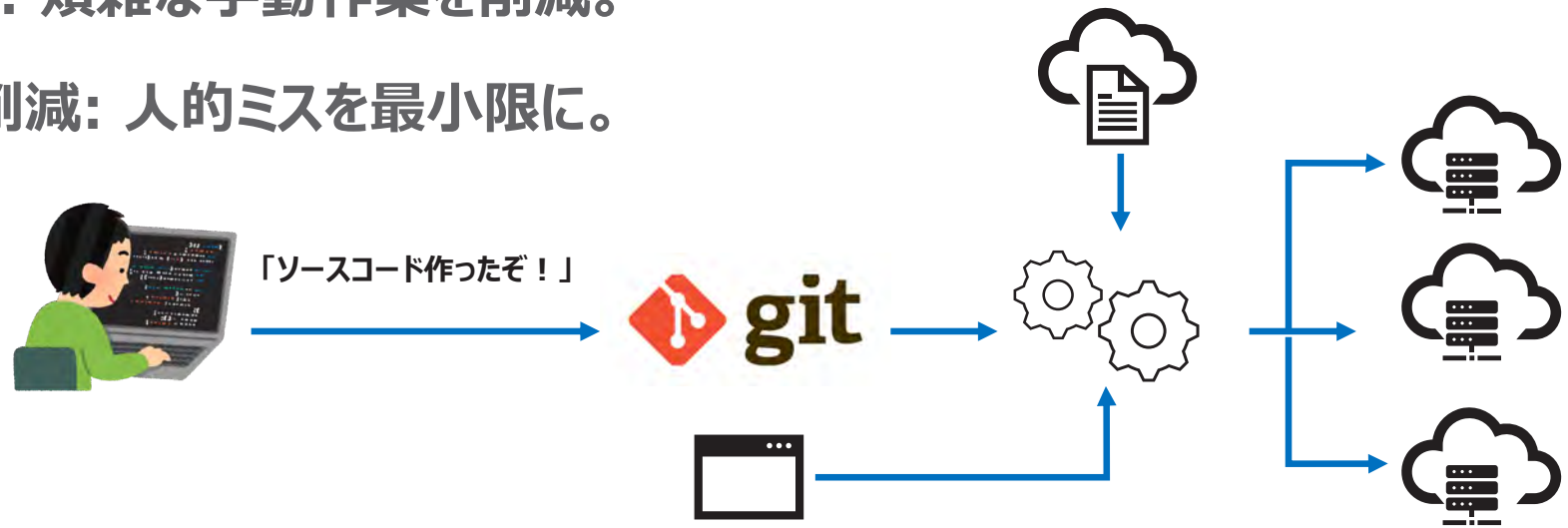
1. 静的セキュリティチェック（SAST）
2. リンティング（Linting）
3. カナリアリリース
4. コミットメッセージのチェック
5. コンテナレジストリへの登録

第八章 8.　IaC (Infrastructure as Code)

IaCとは？

Infrastructure as Code（IaC）とは、インフラ構成をコードで定義し、自動化する手法。

- 再現性: 同じ設定を何度でも再現可能。
- 自動化: 煩雑な手動作業を削減。
- エラー削減: 人的ミスを最小限に。



IaCの主要ツール

【宣言型（Declarative）】

まずは人間がこういうインフラであってほしいという状態を定義します。例えば、ロードバランサが1台あって、その下にアプリケーションサーバが10台あって、それらが参照する DB が1台ある、というような定義です。インフラ全体を制御する仕組みがあるという前提があり、その仕組みに上述のような定義を入力すると、人間が介入しなくてもシステムが自動的にその状態を維持します。

- CFn（AWS CloudFormation）：AWS専用のIaCツール。
- Terraform: マルチクラウド対応。

IaCの主要ツール

【手続型（Imperative）】

あるべき状態に向かって、手順を踏んで構築するようなインフラを手続型と呼んでいます。手続型はプリミティブな処理を積み重ねて、実現したい状態に持っていくインフラです。代表的なものとしては、人間が手順書に従って OS の設定をして、ミドルウェアのインストールを行い、設定を展開し、それを必要な台数用意する、というのも手続型だと思いますし、そのような構築に必要な処理が順に書かれているようなシェルスクリプトを実行する、といったやり方も手続型だと思います。

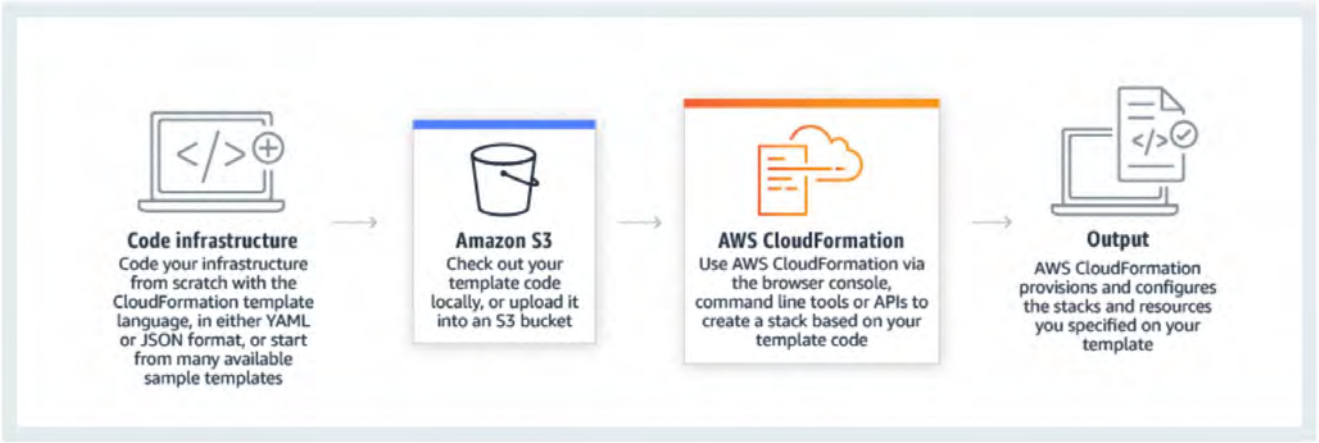
- Ansible: 構成管理ツール。

AWS CloudFormationの特徴

AWS専用で、AWSリソースの設定をYAML/JSONで記述。

【利点】

- **AWSサービスとの統合が容易。**
- **スタック管理により、一括操作が可能。**



Terraformの概要

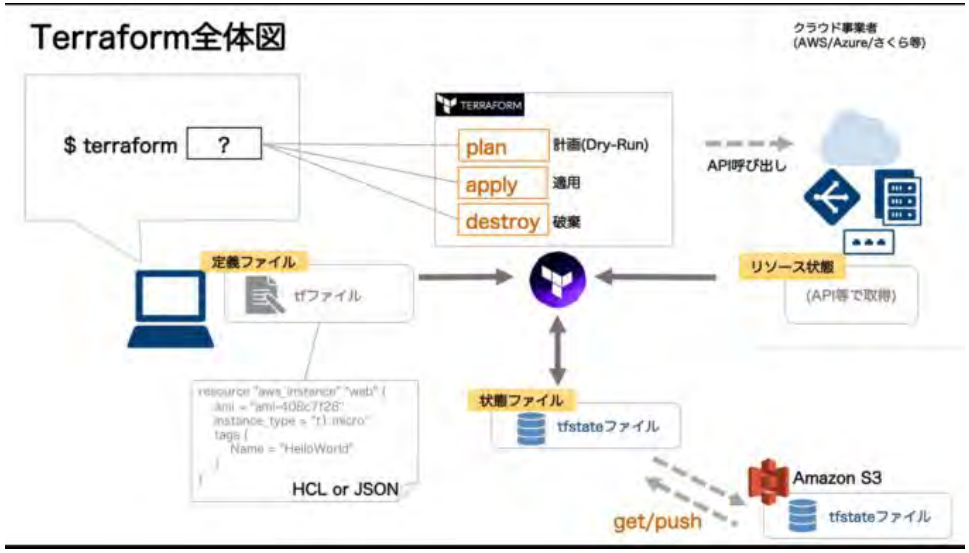
HashiCorpが提供するマルチクラウド対応のIaCツール。

【特徴】

- **宣言型で構成を記述。**
- **再利用可能なモジュール化。**

【利点】

- **クラウド環境を横断した管理が可能。**
- **コードでの変更追跡が容易。**

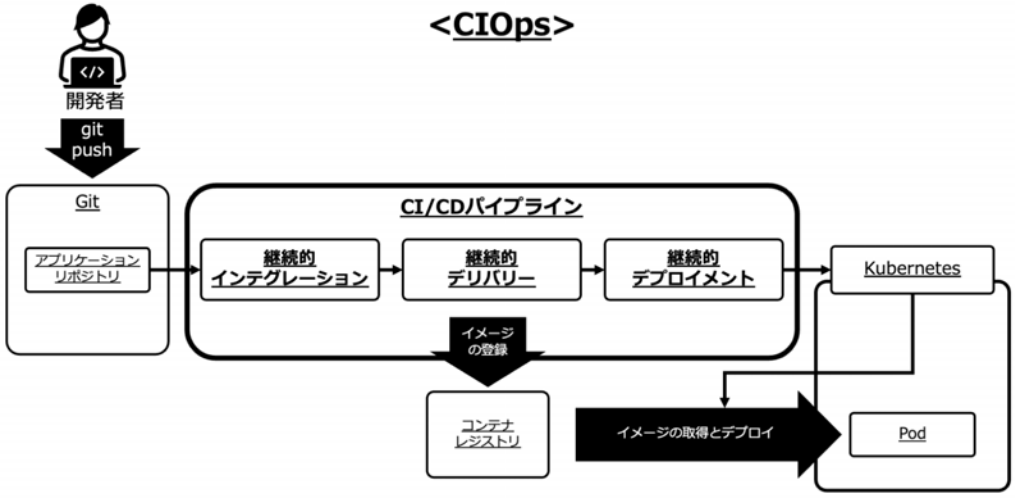


GitOpsの活用

GitOpsとは、GitをIaC管理の単一の真実の源とするアプローチ。

【利点】

- 変更の履歴管理が簡単。
- 自動デプロイをトリガーにより実現。



IaCの利点と課題

【利点】

- インフラの標準化と一貫性。
- 短時間で環境を再構築可能。
- 自動化によるコスト削減。

【課題】

- 初期設定の複雑さ。
- ツールや環境の選定が必要。
- 運用管理スキルが求められる。

まとめ

- IaCは、インフラ管理を効率化する重要な手法。
- CFnやTerraformなどのツールを活用し、自動化を推進。
- GitOpsとの統合で運用をさらに効率化可能。

確認テスト1

Q1: アイエーシーの主な利点として誤っているものはどれですか？

1. 同じ環境を何度でも再現できる
2. 自動化により人的ミスを削減できる
3. コストが必ず削減される
4. 環境の変更履歴を管理できる

Q2: 宣言型（デクララティブ）と手続型（インペラティブ）の違いは何ですか？

1. 宣言型は手順を細かく記述するが、手続型は最終結果のみを指定する
2. 宣言型は望ましい最終状態を記述し、手続型は手順を記述する
3. 宣言型のほうがエラーを起こしやすい
4. 手続型のほうが自動化しやすい

確認テスト2

Q3: エーダブリューエス クラウドフォーメーション（CFn）の特徴として正しいものはどれですか？

1. すべてのクラウドプロバイダーで利用可能
2. AWSサービスの設定をYAMLまたはJSONで記述
3. 物理サーバーのみ管理可能
4. すべてのクラウド環境を自動で最適化

Q4: テラフォームの特徴として正しくないものはどれですか？

1. マルチクラウド環境で利用できる
2. 宣言型の記述方式を採用している
3. 変更履歴を管理するtfstateファイルを使用する
4. AWS専用のツールである

確認テスト3

Q5: ギットオプスのメリットとして誤っているものはどれですか？

1. 変更履歴をGitで管理できる
2. インフラの自動デプロイを実現できる
3. 必ずセキュリティが向上する
4. 開発・本番環境を統一管理できる

Q6: アイエーシーの課題として正しくないものはどれですか？

1. 初期設定が複雑
2. すべてのシステムで適用できる
3. 運用管理のスキルが必要
4. ツール選定が重要
5. コンテナレジストリへの登録

確認テスト4

Q7: アイエーシーの導入によって直接得られるメリットはどれですか？

1. 環境構築の自動化
2. クラウドコストの削減
3. サーバーレスアーキテクチャの構築
4. 人的リソースの完全不要化

Q8: 次のうち、宣言型アイエーシーツールの例ではないものはどれですか？

1. テラフォーム
2. エーダブリューエス クラウドフォーメーション
3. シェルスクリプト
4. Ansible

確認テスト5

Q9: アイエーシーが一般的に適用される分野はどれですか？

1. ソフトウェア開発のみ
2. クラウドインフラ管理
3. ハードウェア製造
4. データ入力業務

Q10: 次のうち、アイエーシーの標準化に最も関係が深い考え方はどれですか？

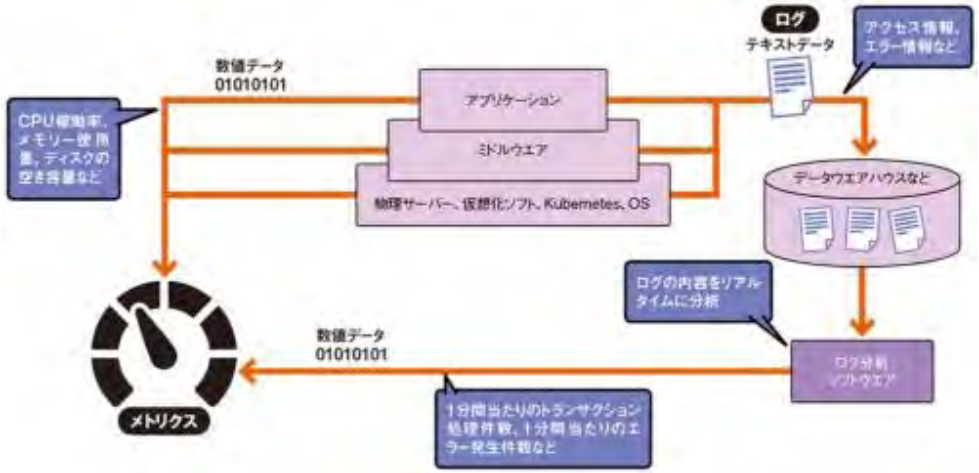
1. DevOps
2. アジャイル開発
3. ウォーターフォール開発
4. エンタープライズアーキテクチャ

第九章 9.可観測性（Observability）

第九章可観測性（Observability）

可観測性の基本概念

可観測性とは、システムやアプリケーションの出力、ログ、パフォーマンス指標を調べることにより、その状態を監視、測定、理解する能力を指します。先進的なソフトウェアシステムやクラウド・コンピューティングでは、アプリケーションやインフラストラクチャの信頼性、パフォーマンス、セキュリティを確保するために、可観測性がますます重要な役割を担うようになっています。



第九章可観測性（Observability）

モニタリングの目的

- **パフォーマンスの最適化**
  モニタリングを通じて、システムが最適な速度と効率で動作するように調整します。例えば、サーバーのCPU使用率やメモリ使用率を監視し、リソースの過剰消費や不足がないよう管理します。これにより、ユーザーがスムーズにサービスを利用できる環境を維持します。

- **問題の早期発見**
  システム障害やエラーが発生した際に、迅速に気付くことが重要です。モニタリングツールを使用することで、例えばレスポンスタイムが急激に遅くなったり、エラーレートが増加した場合にアラートを発生させ、運用担当者が即座に対応できるようにします。

- **ユーザーエクスペリエンスの向上**
  ユーザーが快適にアプリケーションやサービスを利用できるように、ロード時間や機能動作を監視します。例えば、ウェブページの読み込み時間を短縮したり、操作がスムーズに動作するよう改善点を見つけます。これにより、ユーザー満足度を向上させ、離脱率を低減します。

第九章可観測性（Observability）

モニタリングの対象

- インフラ
  サーバー、ネットワーク、ストレージなど、システムの基盤を構成する要素を監視します。具体的には、以下のような項目をチェックします。
  サーバー: CPU、メモリ、ディスク使用率、動作状態。
  ネットワーク: 帯域幅使用率、遅延、パケットロス。
  ストレージ: ディスクの空き容量、読み書き速度。

- アプリケーション
  アプリケーションの動作状況や性能を監視します。具体的には、以下のような項目が含まれます。
  レスポンス時間: APIやウェブページの応答速度。
  エラー率: アプリケーションの内部で発生するエラーの頻度。

- ユーザー体験:ユーザーがサービスをどのように感じるかを測定します。これには、以下のような要素が含まれます。
  ページロード時間: ウェブページが完全に表示されるまでの時間。
  機能動作: ユーザーがボタンをクリックした際のレスポンスやエラーの有無。

代表的なモニタリングツール

モニタリングツールを活用することで、システムの状態をリアルタイムで監視し、問題の早期発見やパフォーマンスの最適化を実現できます。手動での監視は非効率であり、適切なツールを導入することで自動化と可視化が可能になります。

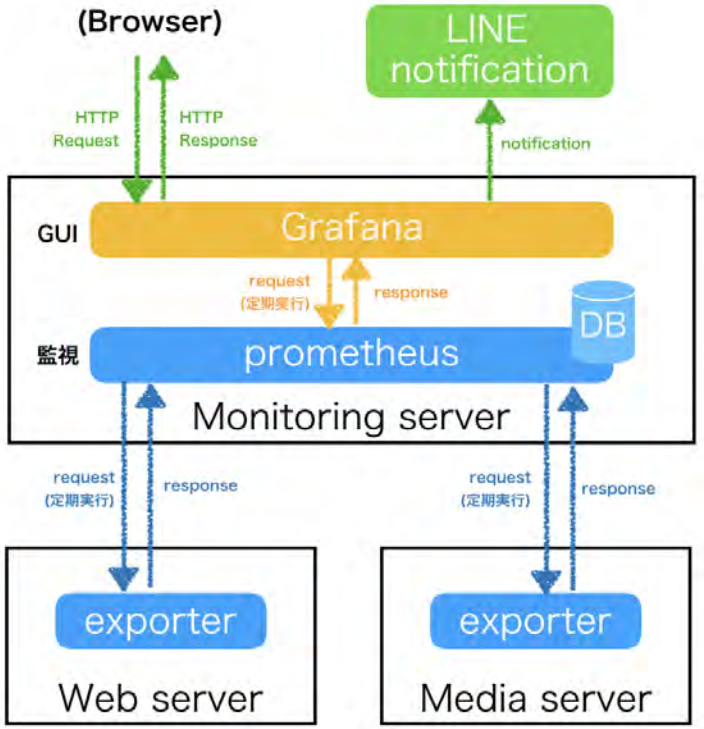
	Prometheus（プロメテウス）	Grafana（グラファナ）
役割	モニタリング（データ収集）	可視化（ダッシュボード作成）
データ形式	時系列データベース	任意のデータソース対応
アラート機能	あり（AlertManager）	あり（しきい値設定可）
主な用途	メトリクス収集・監視	グラフ表示・ダッシュボード作成
利用例	CPU使用率の監視	監視データの視覚化

PrometheusとGrafanaの連携

Prometheusはデータ収集に特化、Grafanaはデータの可視化に特化→ 組み合わせることで、データを収集しながら分かりやすいダッシュボードで監視できます。

■ 連携の仕組み

1. Prometheusが監視対象（サーバーやアプリ）からメトリクスを収集
2. Prometheusがデータを時系列データベースに保
3. 存GrafanaがPrometheusのデータを取得し、ダッシュボードに可視化
4. 必要に応じてGrafanaからアラートを発生



第九章可観測性（Observability）

ログ管理の重要性

システム運用において、ログは「問題の診断」と「セキュリティ監視」に不可欠なデータです。
適切なログ管理を行うことで、以下のメリットが得られます。

1. 問題の根本原因分析（Root Cause Analysis）

- システム障害が発生した際、ログを解析することで原因を特定しやすくなります。

2. リアルタイム監視とアラート設定

- 異常な動作（例: 500エラーの急増）が検知された際、即座にアラートを発生させ対応可能。

3. セキュリティ監視とコンプライアンス対応

- 不正アクセスやデータ改ざんなどの兆候をログから分析し、サイバー攻撃を検出。

4. システムのパフォーマンス改善

- リクエスト処理時間やデータベースクエリの実行時間を分析し、ボトルネックを特定。

第九章可観測性（Observability）

ログ管理のワークフロー

1. ログ収集
   - サーバー、アプリ、ネットワークからログを収集（Fluentd, Filebeat）
2. ログ集約
   - 一元管理のためにログを中央リポジトリへ送信
3. ログ処理・変換
   - データを統一フォーマットに変換し、不要な情報を除去（Logstash, Kafka）
4. 保存・インデックス化
   - Elasticsearchでログを整理し、高速検索を可能にする
5. 検索・可視化
   - KibanaやGrafanaでダッシュボードを作成し、ログを分析
6. アラート・監視
   - 異常を検知した場合、PrometheusやAlert Managerが通知を発生

ログ管理ツールの比較

	役割	主な機能	代表的な用途
Fluentd	ログ収集	軽量・多種フォーマット	分散環境でのログ収集
Filebeat	ログ収集	軽量・Elastic Stack向	サーバーログの転送
Logstash	データ統合	ログのフィルタリング・変換	大規模データの前処理
Elasticsearch	ログ検索	高速検索・分析	システム全体のログ管理
Kibana	ログ可視化	ダッシュボード・リアルタイム監視	監視ダッシュボード作成
Grafana	統合可視化	メトリクス＋ログの分析	インフラ監視と組み合わせ

まとめ

可観測性（Observability）は、モダンなシステム運用において不可欠な要素です。特に、ログ・メトリクス・トレースを統合的に活用し、リアルタイムで異常を検知・対応する仕組みを構築することが求められます。

- 可観測性の3要素: ログ・メトリクス・トレース
- 主要ツール: Prometheus（メトリクス収集）+ Grafana（可視化）+ ELK（ログ管理）
- モニタリングの目的: 障害対応の迅速化、システムの最適化、UX向上
- 実践的な活用: アラート設定とダッシュボードによるリアルタイム監視

確認テスト1

Q1: 可観測性の目的として適切でないものはどれですか？

1. システムの状態を把握する
2. 異常の早期発見
3. 手動で全ての運用を管理する
4. パフォーマンスの最適化

Q2: 可観測性の主要な要素として含まれないものはどれですか？

1. 測定データ
2. 記録データ
3. 機械学習モデル
4. 追跡情報

確認テスト2

Q3: 記録データを利用する主な目的はどれですか？

1. 障害の原因分析
2. リアルタイム監視
3. セキュリティ監視
4. 全て正しい

Q4: 監視の対象として正しくないものはどれですか？

1. ネットワークの帯域幅
2. ユーザーのブラウザ履歴
3. サーバーのCPU使用率
4. システムのレスポンス時間

確認テスト3

Q5: 可視化ツールの目的として正しくないものはどれですか？

1. リアルタイムの状態把握
2. 異常のトレンド分析
3. データの完全削除
4. システムの最適化

Q6: プロメテウスの主な役割は何ですか？

1. データ収集
2. ダッシュボード作成
3. ログ管理
4. システムの自動修正
5. コンテナレジストリへの登録

確認テスト4

Q7: グラファナの主な役割は何ですか？

1. データの可視化
2. データの可視化
3. アプリケーションの開発
4. データの収集

Q8: 記録データの管理手順に含まれないものはどれですか？

1. データの収集
2. データの変換
3. ログの破棄
4. データの可視化

確認テスト5

Q9: 異常検知のために使用される手法はどれですか？

1. ルールベースの監視
2. 機械学習によるパターン認識
3. しきい値を設定したアラート
4. 全て正しい

Q10: 可観測性の向上によって得られる効果として適切でないものはどれですか？

1. 障害対応の迅速化
2. システムの最適化
3. 手作業の増加
4. ユーザー体験の向上

第十章
クラウドネイティブにおけるセキュリティ

クラウドネイティブセキュリティの概要

- クラウド環境でのセキュリティの考え方の変化
- 従来型の境界防御 vs. ゼロトラストモデル
- シフトレフト（Shift Left）とDevSecOpsの重要性
- 主要なセキュリティ対策
  （IAM, 暗号化, ネットワークセキュリティ, 監視）

クラウド環境でのセキュリティの変化

クラウド環境におけるセキュリティは、従来のオンプレミス環境とは大きく異なります。従来のセキュリティは、ファイアウォールやVPNを使った境界防御型が主流でした。しかし、クラウド環境では、ネットワークの境界が曖昧になり、「ネットワーク内にいるから安全」という考え方は通用しません。

クラウド環境では「ゼロトラストセキュリティ」へのシフトが必要

ゼロトラストモデルとは？

ゼロトラストモデルでは、「すべての通信は信頼しない」という前提でセキュリティを強化します。つまり、社内・社外・クラウド・オンプレミスの区別なく、全てのアクセスを検証する必要があります。

- アクセス管理: IAM（Identity and Access Management）を活用し、ユーザーごとのアクセス制御を徹底
- データ保護: 保存データと通信データの暗号化を義務化（AES-256, TLS 1.2/1.3）
- ネットワークセキュリティ: マイクロセグメンテーションとゼロトラストネットワークを導入
- リアルタイム監視: ログ監視とアラート（SIEM, SOAR, EDR）で攻撃を検知

シフトレフト（Shift Left）とは？

従来の開発プロセスでは、開発後の最終段階でセキュリティチェックを行うのが一般的でした。しかし、この方法では、後から発見された脆弱性の修正コストが高くなるという問題があります。そこで、開発初期段階からセキュリティ対策を導入する「シフトレフト（Shift Left）」の考え方が登場しました。

シフトレフトとは、開発の左側（初期段階）でセキュリティ対策を行うこと

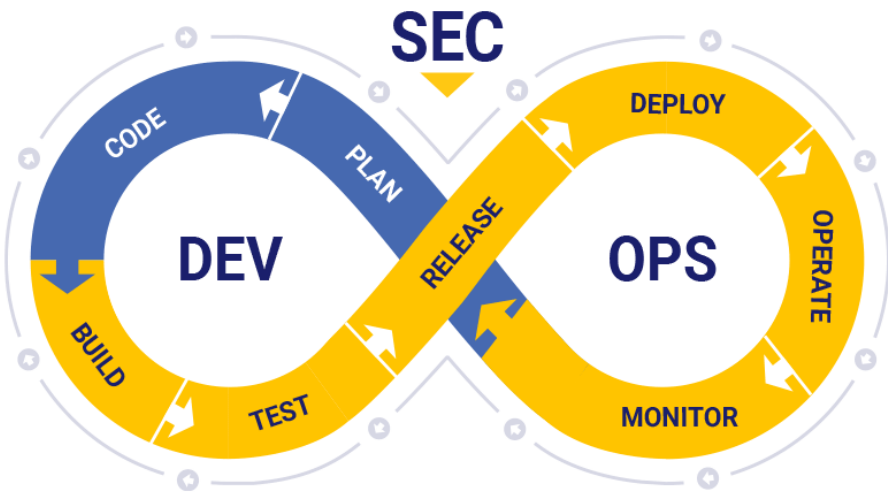
DevSecOpsとは？

シフトレフトを実現するために、セキュリティを開発プロセスに統合したのが DevSecOps です。DevSecOpsとは、Development（開発）+ Security（セキュリティ）+ Operations（運用） の統合概念で、開発と運用のすべての段階でセキュリティを考慮することを意味します。

- セキュリティテストの自動化（CI/CDパイプラインに統合）
- SAST（静的解析）とDAST（動的解析）の活用
- Infrastructure as Code（IaC）を用いたセキュリティ管理
- コンテナ・Kubernetes環境のセキュリティ対策（OPA, Falco）

シフトレフト（Shift Left）とDevSecOpsの重要性

クラウド環境では、開発スピードが非常に速く、セキュリティは「後付け」では対応しきれません。そこで、セキュリティ対策を開発の初期段階から組み込む必要があります。これが、シフトレフトセキュリティ（Shift Left Security）の考え方です。また、セキュリティを自動化し、開発プロセスに統合するのが DevSecOps（Development + Security + Operations） です。



クラウドセキュリティの主要対策

カテゴリ	対策例
IAM（アクセス管理）	RBAC（ロールベースアクセス制御）、MFA（多要素認証）
暗号化とデータ保護	AES-256, TLS/SSL, KMS（鍵管理システム）
ネットワークセキュリティ	WAF（Web Application Firewall）, マイクロセグメンテーション
セキュリティ監視	SIEM（ログ分析）、SOAR（自動対応）、脅威インテリジェンス

CI/CDパイプラインとセキュリティ

DevSecOpsの重要な要素として、CI/CDパイプラインの中でセキュリティを組み込むことが挙げられます。

ステージ	セキュリティ対策
コードコミット	SAST（静的コード解析）を実施ビルド 依存関係の脆弱性スキャン（SCA）
ビルド	依存関係の脆弱性スキャン（SCA）
テスト	DAST（動的アプリケーションテスト）
デプロイ前	IaCのセキュリティチェック（Open Policy Agent, Terraform Scan）
運用	監視とログ分析（SIEM, SOAR）

IAMとは？

IAM（Identity and Access Management）は、クラウド環境における「アクセス管理」と「認証」の仕組みです。システムへの不正アクセスや誤操作を防ぐために、ユーザーごとのアクセス制御を適切に設定することが重要です。

【IAMの主な使用目的】

- ユーザーのアクセスを適切に制限（不正アクセスを防ぐ）
- サービス間の通信を制御し、セキュリティリスクを低減
- 監査ログを取得し、コンプライアンスを遵守

IAMの主要な機能

IAMの主な機能として、以下の3つが挙げられます

1. RBAC（ロールベースアクセス制御）

- ユーザーごとにアクセス権限を個別に設定するのではなく、役割（ロール）単位で管理する方式。
- メリット: 大規模なシステムでも管理が容易

2. PoLP（最小権限の原則）

- ユーザーやサービスには「最低限必要な権限のみを付与」するのが基本ルール。
- 誤操作や内部不正を防ぐために重要な考え方。

3. MFA（多要素認証）

- 単一のパスワード認証だけでなく、追加の認証要素（ワンタイムパスワードなど）を求める仕組み。
- クラウド管理者アカウントでは必須のセキュリティ対策。

クラウド環境でのIAMの活用

主要なクラウドプロバイダは、それぞれIAMの仕組みを提供しています。

クラウドサービス	IAMソリューション	主な特徴
AWS	AWS IAM	IAMポリシーによる詳細な権限制御、MFA対応
Azure	Azure AD (Active Directory)	組織向けのアクセス管理、SAML認証
Google Cloud	Google Cloud IAM	RBACとポリシーベースのアクセス管理

クラウド環境では、IAMを適切に設定し、管理者の過剰権限を防ぐことが重要！

IAMのセキュリティベストプラクティス

IAMを適切に設定しないと、セキュリティリスクが高まります。以下のベストプラクティスを活用し、IAMの安全性を確保しましょう。

- ✓ すべての管理者アカウントにMFAを適用する
- ✓ 最小権限の原則（PoLP）を徹底する（不必要な権限を付与しない）
- ✓ IAMの監査ログを定期的に確認し、不審なアクセスを検出する
- ✓ IAMのアクセスキーは極力使用せず、ロール（Role）を活用する

暗号化とは？

クラウド環境では、データのセキュリティを確保するために暗号化が不可欠です。暗号化を適用することで、万が一データが盗まれても、第三者が解読できないように保護できます。

【暗号化の主な目的】

- データの機密性（Confidentiality）を確保（外部に漏れても読めない）
- データの完全性（Integrity）を維持（改ざんされていないことを保証）
- コンプライアンス要件（GDPR, HIPAA, ISO 27001）に準拠

暗号化の種類

クラウド環境では、主に2種類の暗号化が使用されます。

1. データ転送中の暗号化

- TLS（Transport Layer Security）/SSL（Secure Sockets Layer）を使用し、通信経路上のデータを暗号化して保護します。
- 例: HTTPS（TLS 1.2/1.3）が適用されたWebサイトの通信

2. データ保存時の暗号化（At-Rest Encryption）

- AES-256（Advanced Encryption Standard 256-bit） が標準的に使用される
- クラウド環境では、ストレージ（S3, Blob, Cloud Storage）に保存されるデータを自動的に暗号化可能
- 顧客データや機密情報を格納するすべてのデータベースは暗号化を適用すべき

データ転送中・保存時の両方で暗号化を適用し、データを安全に保つ！

クラウドの暗号化ツール

クラウドプロバイダーは、暗号化を管理する専用ツールを提供しています。

クラウドサービス	暗号化ツール	主な機能
AWS	AWS KMS （Key Management Service）	暗号鍵の管理、暗号化APIの提供
Azure	Azure Key Vault	鍵の管理、証明書管理、HSM（Hardware Security Module）
Google Cloud	Google Cloud KMS	データ暗号化、鍵管理、自動ローテーション

これらのツールを活用し、データの暗号化と鍵管理を適切に行うことが重要！

鍵管理の重要性

データの暗号化と同様に、暗号鍵の管理も非常に重要です。
適切に管理されないと、暗号化していてもデータが漏洩するリスクがあります。

- ✓ キーのローテーションを定期的に実施する（長期間使用しない）
- ✓ 鍵はセキュアなHSM（Hardware Security Module）に保存する
- ✓ アクセス制御（IAM）を適用し、不要なユーザーが暗号鍵にアクセスできないようにする

ネットワークセキュリティの重要性

クラウド環境では、物理的なネットワーク境界が存在しないため、従来のファイアウォールを使った「境界防御型セキュリティ」では不十分です。

クラウド環境では、ネットワーク内部・外部のすべての通信を保護する仕組みが必要！

- 未認証のユーザーからのアクセスを防ぐ（不正アクセス防止）
- クラウドリソース間のトラフィックを制御する（内部セキュリティ強化）
- DDoS攻撃やSQLインジェクションなどの脅威を防御する

ゼロトラストネットワークとは？

ゼロトラストネットワーク（Zero Trust Network, ZTN）とは、「ネットワーク内であっても、すべての通信を検証し、信頼しない」という考え方です。

【ゼロトラストの主な特徴】

- 最小権限の原則（PoLP）を徹底し、アクセスを制限
- ネットワークセグメンテーションを行い、内部ネットワークの侵害を防ぐ
- マルチファクター認証（MFA）を適用し、なりすましを防止
- クラウドファイアウォールとWAF（Web Application Firewall）を活用

ネットワークセキュリティの主要対策

セキュリティ対策	概要	主なツール・サービス
WAF（Web Application Firewall）	SQLインジェクション、XSS攻撃からWebアプリを保護	AWS WAF, Azure WAF, Cloudflare WAF
ネットワークセグメンテーション	VPC, サブネットを分離し、重要データへのアクセスを制限	AWS VPC, Azure VNet, Google Cloud VPC
VPN・SD-WAN	安全なリモートアクセスとデータセンター間の通信保護	AWS VPN, Azure VPN Gateway, Google Cloud VPN
DDoS対策	大量のトラフィック攻撃からシステムを保護	AWS Shield, Azure DDoS Protection, Google Cloud Armor

クラウド環境では、これらの対策を組み合わせて、強固なネットワークセキュリティを実現！

WAF（Web Application Firewall）とは？

WAF（Web Application Firewall） は、Webアプリケーションを狙った攻撃（SQLインジェクション、XSSなど）を防ぐためのファイアウォールです。

【主な特徴】

- ユーザーのリクエストをリアルタイムで分析し、不正な通信をブロック
- カスタムルールを設定し、企業ごとのセキュリティポリシーを適用可能
- クラウド型WAF（AWS WAF, Azure WAF, Cloudflare WAF）を活用することで、運用の負担を軽減

Webアプリケーションを保護するために、WAFは必須のセキュリティ対策！

ネットワークセグメンテーション

クラウド環境では、VPC（Virtual Private Cloud）を活用して、リソースごとにネットワークを分割し、不正アクセスを防ぐことが重要です。

【主な特徴】

- パブリックサブネット（外部公開用）とプライベートサブネット（内部用）を分離
- データベースや機密データはプライベートサブネットに配置
- ネットワークACLやセキュリティグループを活用し、アクセスを制限

ネットワークを適切に分割し、重要データへの直接アクセスを防ぐ！

VPN・SD-WANによるセキュアな接続

リモートワークの増加に伴い、クラウド環境に安全にアクセスするためのVPNやSD-WANの活用が重要になっています。

1. VPN（仮想プライベートネットワーク）

- クラウド環境へ安全にリモートアクセスするための手段
- 企業ネットワークとクラウドをセキュアに接続
- AWS VPN, Azure VPN Gateway, Google Cloud VPN などが利用可能

2. SD-WAN（Software-Defined Wide Area Network）

- クラウドと拠点を柔軟かつ効率的に接続する技術
- 高速で安全なネットワーク通信を実現
- Google Cloud Network Connectivity Center などが代表例

セキュリティ監視の重要性

クラウド環境では、常にセキュリティリスクが存在するため、リアルタイムでの監視が不可欠です。特に、攻撃者がシステムに侵入した場合、異常なアクティビティを素早く検知し、対処できるかが重要となります。

- ✓ 不審なログインやアクセスをリアルタイムで検知
- ✓ DDoS攻撃や不正アクセスの兆候を早期発見
- ✓ セキュリティインシデントが発生した際の迅速な対応

クラウド環境では、手動監視だけではなく、

SIEMやSOARを活用した自動監視が求められる！

ログ監視と異常検知

ログデータの分析を行い、異常を検知する仕組みが必要です。クラウド環境では、各リソースのログを一元的に収集し、リアルタイムで分析・検知を行います。

1. SIEM（Security Information and Event Management）

- 複数のシステムから収集したログを統合的に管理し、異常を検知
- 機械学習を活用した脅威検出が可能

2. リアルタイム監視のポイント

- 異常なログインやアクセスを検知（例: 管理者アカウントの海外ログイン）
- 急激なトラフィック増加を検知（例: DDoS攻撃）
- マルウェア感染の兆候を発見（例: システムプロセスの異常動作）

SIEMを活用し、すべてのログを一元管理し、異常をリアルタイムで検知する！

セキュリティインシデント対応

セキュリティインシデント（不正アクセス、データ漏洩、攻撃など）が発生した場合、迅速に対応し、被害を最小限に抑えることが求められます。

1. SOC（Security Operations Center）

- セキュリティ専門チームが24時間体制で監視
- SIEMの分析結果をもとに、異常が発生した場合に対応

2. SOAR（Security Orchestration, Automation, and Response）

- インシデント対応を自動化し、迅速な対処を実現
- 脅威を検知した際に、事前に定めた対応プロセスを自動実行

クラウド環境でのセキュリティ監視ツール

クラウドプロバイダーは、暗号化を管理する専用ツールを提供しています。

セキュリティ監視のカテゴリ	代表的なツール	主な機能
SIEM（ログ分析・異常検知）	AWS GuardDuty, Microsoft Sentinel, Google Chronicle	ログの一元管理、異常検知
SOC（セキュリティ監視センター）	AWS Security Hub, Azure Security Center	リアルタイム監視、セキュリティダッシュボード
SOAR（自動対応）	IBM Resilient, Palo Alto Cortex XSOAR	インシデント対応の自動化

これらのツールを活用し、セキュリティ監視を強化することが重要！

クラウド環境におけるセキュリティ対策の必要性

クラウド環境では、従来のセキュリティ対策に加えて、クラウド特有の脅威（DDoS攻撃、設定ミス、不正アクセス）への対応が求められます。

- ✓ クラウドプロバイダが提供する専用セキュリティツールを活用し、セキュリティ強化を図る
- ✓ アクセス管理、ネットワーク防御、ログ監視、脅威検出などの機能を統合的に活用

クラウドのセキュリティ対策は、プロバイダが提供するツールを適切に設定し、活用することが重要！

まとめ

クラウド環境におけるセキュリティ対策は単発ではなく、継続的な運用が必要です。

適切なツールを活用し、自動化と監視の強化を進めることで、安全なクラウド環境を維持しましょう。

確認テスト1

Q1: クラウド環境において従来のセキュリティ対策では不十分な理由は何ですか？

1. ゼロトラストが適用されていないから
2. クラウドは物理的な管理ができないから
3. クラウド環境にはセキュリティリスクが存在しないから
4. オンプレミスよりも安全だから

Q2: シフトレフトの目的は何ですか？

1. セキュリティを開発初期段階から組み込む
2. 開発の最終段階でセキュリティを強化する
3. 開発者の作業量を減らす
4. セキュリティを手動で実施する

確認テスト2

Q3: デブセックオプスの特徴として正しいものはどれですか？

1. セキュリティを運用フェーズだけで適用する
2. セキュリティを開発プロセス全体に統合する
3. 開発の最後にのみセキュリティチェックを行う
4. セキュリティ対策を外部業者に委託する

Q4: ゼロトラストモデルの考え方として適切なものはどれですか？

1. 一度認証されたユーザーは信頼する
2. 社内ネットワークは安全とみなす
3. すべてのアクセスを検証する
4. パスワードがあれば認証は不要

確認テスト3

Q5: クラウド環境での暗号化に関する正しい説明はどれですか？

1. 暗号化はデータ保存時のみ適用される
2. データ転送中の暗号化は不要
3. データの保存と転送の両方で暗号化が必要
4. 暗号化を使用するとシステムのパフォーマンスが必ず低下する

Q6: エスアイイーエム（SIEM）の役割は何ですか？

1. データの圧縮
2. セキュリティイベントの監視と分析
3. クラウドの料金計算
4. ファイアウォールの設定管理
5. コンテナレジストリへの登録

確認テスト4

Q7: ネットワークセグメンテーションの目的は何ですか？

1. ネットワークのトラフィックを減らす
2. 外部からの攻撃を防ぐためにネットワークを分割する
3. インターネットの速度を向上させる
4. 全ユーザーのアクセスを一括で管理する

Q8: クラウド環境でのインシデント対応に関する適切な方法はどれですか？

1. 手動でログを解析し、対応する
2. インシデント対応を自動化し、迅速な対応を行う
3. 発生したインシデントは記録しない
4. 対応はオンプレミス環境の時のみ行う

確認テスト5

Q9: クラウドセキュリティにおけるIAM（アイエーエム）の重要性は何ですか？

1. すべてのユーザーに管理者権限を付与する
2. アクセス制御を行い、不要な権限を制限する
3. パスワードなしで認証を行う
4. 特定の端末のみアクセスを許可する

Q10: クラウドセキュリティにおけるDDoS攻撃の防御策として適切なものはどれですか？

1. 攻撃が発生したらすぐにサーバーを停止する
2. DDoS防御ツールを活用し、異常なトラフィックを遮断する
3. 攻撃の発生を監視しない
4. サーバーを増やして負荷を分散する

確認テスト回答

□第一章 クラウドネイティブの基本

Q1　2|4

Q2　1|3|4

Q3　1

Q4　4

Q5　3

Q6　3

Q7　2|3

Q8　1|2|4

Q9　2|4

第二章 クラウド技術

Q1　2

Q2　5

Q3　1|2

Q4　2

Q5　1

Q6　1|4

Q7　3

第三章 コンテナ技術

Q1　1|3

Q2　2

Q3　1|2|4

Q4　3|4|5

Q5　1

Q6　1|5

Q7　1|3

Q8　3

第四章 サーバーレスアーキテクチャ

Q1　4

Q2　3|5

Q3　2|3

Q4　1

Q5　3|4

Q6　2|4

Q7　2|3

第五章 マイクロサービス

Q1　3

Q2　1|2|3|4

Q3　1|2|3

Q4　2|4

Q5　1|2|4

Q6　1|2|4

Q7　2|4

Q8　3

第六章 アジャイルと DevOps

Q1　2

Q2　2

Q3　1|2|4

Q4　4

Q5　1|4

Q6　3|4

Q7　1|4

第七章 CI/CD

Q1　1|2

Q2　3

Q3　1|3|4

Q4　4

Q5　2|4

Q6　1|2|4

第八章 8. IaC (Infrastructure as Code)

Q1　3

Q2　2

Q3　2

Q4　4

Q5　3

Q6　2

Q7　1

Q8　3

Q9　2

Q10　1

第九章 9.可観測性（Observability）

Q1　3

Q2　3

Q3　4

Q4　2

Q5　3

Q6　1

Q7　1

Q8　3

Q9　4

Q10　3

第十章 クラウドネイティブにおけるセキュリティ

Q1　1

Q2　1

Q3　2

Q4　3

Q5　3

Q6　2

Q7　2

Q8　2

Q9　2

Q10　2

ー 演習課題ー

演習課題①

課題 1

IaaS（Infrastructure as a Service）、PaaS（Platform as a Service）、SaaS（Software as a Service）の違いを説明し、それぞれの代表的なクラウドサービスを1つずつ挙げてください。

演習課題②

課題２

AWS LambdaまたはGoogle Cloud Functionsを使って、HTTPリクエストを受け取る簡単な関数を作成し、API Gateway（またはCloud Endpoints）経由で実行できるようにしてください。

演習課題③

課題３

PrometheusとGrafanaを使って、コンテナ化されたアプリケーションのメトリクス（CPU使用率やメモリ使用量など）を可視化する方法を説明してください。

演習課題④

課題４

クラウドネイティブ環境で考慮すべきセキュリティ対策（IAMポリシー、ゼロトラスト、コンテナセキュリティなど）を3つ挙げ、それぞれの適用例を説明してください。

令和6年度文部科学省委託
「専門職業人材の最新技能アップデートのための専修学校リカレント教育推進」事業
情報技術者の技能アップデートのためのリカレント教育推進事業

クラウドネイティブ概論教材資料

令和7年2月

一般社団法人全国専門学校情報教育協会
〒164-0003 東京都中野区東中野1-57-8 辻沢ビル3F
電話：03-5332-5081 FAX：03-5332-5083