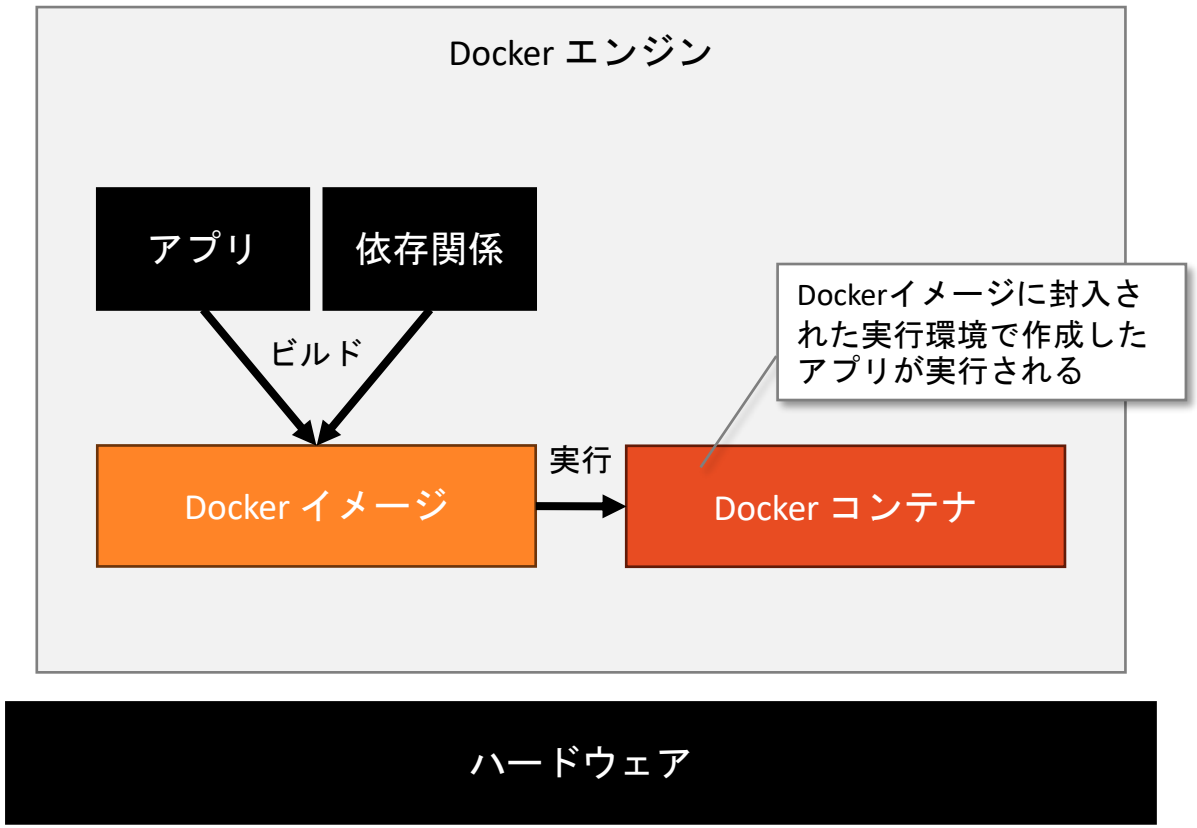
コンテナ型仮想化

仮想環境はハードウェア上で独立した複数の環境を実行する技術で、ホスト型、ハイパーバイザー型、コンテナ型の3種類が存在します

	ホスト型	ハイパーバイザー型	コンテナ型
概要	物理的なホスト OS 上で動作	ハードウェア上で直接実行	単一のOSカーネルで複数のコンテナ実行
利点	● インストールの簡単さ ● ホスト OS のデバイスドライバ利用 ● 異なる OS の同時実行	● 高いパフォーマンスと安定性 ● ハードウェアと直接通信するためのオーバーヘッドが少ない ● 完全なゲスト OS の隔離 ● ハードウェアの効率的な利用	● 起動の速さ ● リソースのオーバーヘッドの少なさ ● 移植性
欠点	● パフォーマンスのオーバーヘッド ● セキュリティ問題の影響	● セットアップと管理が複雑 ● ハードウェアのリソースに直接依存 ● Type 2 の場合 ホスト OS のオーバーヘッドが存在	● 完全な隔離の不足 ● 特定の OS カーネルへの依存
ソフトウェアの例	● VMware Workstation ● Oracle VirtualBox	● VMware vSphere ● Microsoft Hyper-V	● Docker ● LXC
備考	なし	Type 1とType 2の違い: Type 1: ● 高いパフォーマンスと安定性 Type 2 ● ホスト OS 上で動作 ● 設定の簡単さ	なし

Dockerはコンテナ型仮想化ソリューションで、事実上の標準として利用されています



■ Docker の定義と背景

- アプリケーションと依存関係をコンテナとしてパッケージ化
- どの環境でも一貫した動作を保証
- アプリケーションの依存関係をインフラから分離

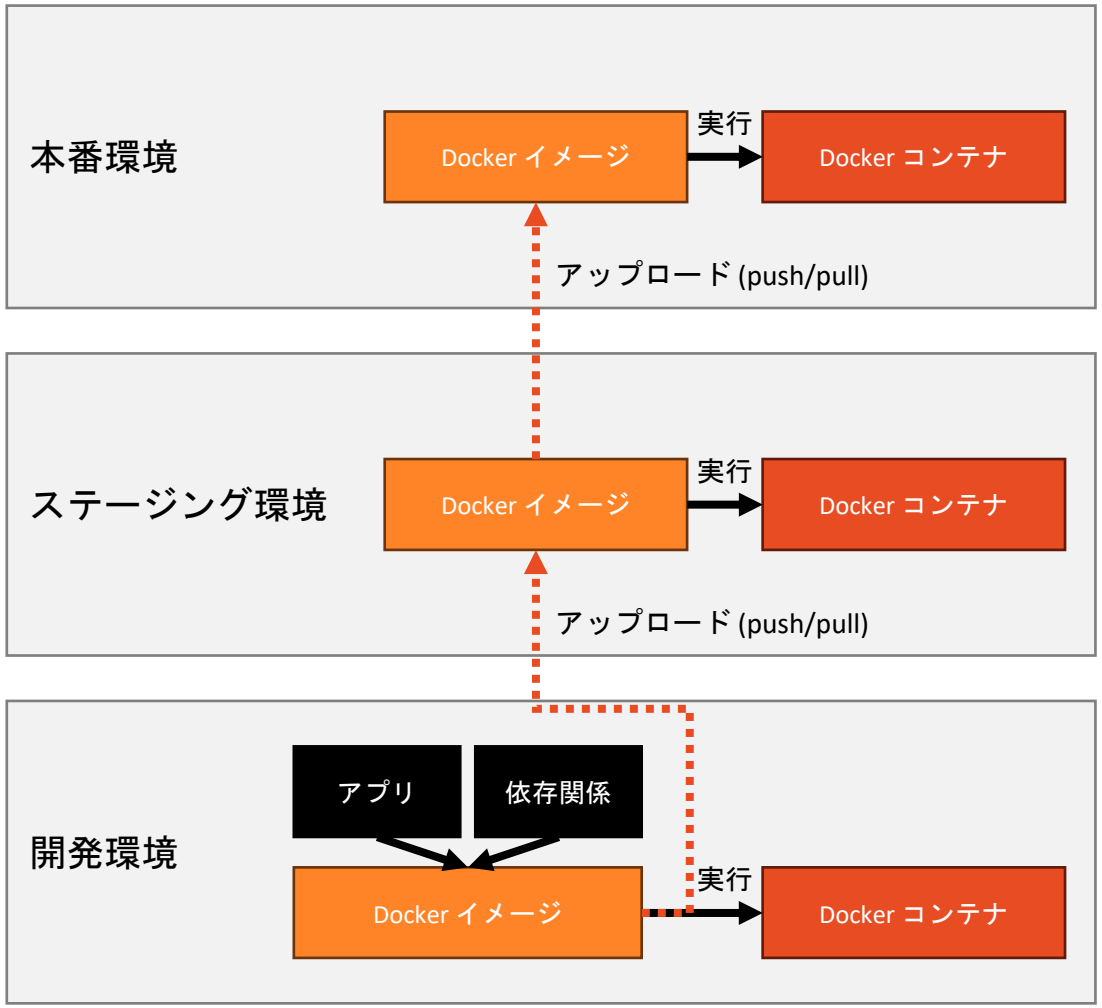
■ Docker のアーキテクチャ

- Docker エンジン
  - コア技術
  - Linux の cgroups、namespacesを利用
- Docker イメージ
  - アプリケーションと依存関係のスナップショット
- Docker コンテナ
  - イメージから生成
  - アプリケーションの実行環境

■ Docker の歴史と普及の背景

- 2013 年: Docker Engineとして公開
- Microsoft との協力で Windows Server に導入
- 現在: データセンター、クラウドプロバイダーでの採用

Dockerは、アプリケーションの開発からデプロイメントまでの一貫性と効率性を向上させるために広く利用されています



※ 実際には、ステージング環境・本番環境のDockerイメージはクラウド上でビルドすることがあります

■ アプリケーションの開発とテスト

- 現代の開発の複雑さ
- Docker によるワークフローの簡素化と加速

■ 本番環境でのデプロイ

- コンテナの概念を導入することによる環境の隔離
- 2013年のDocker導入以降の業界標準化

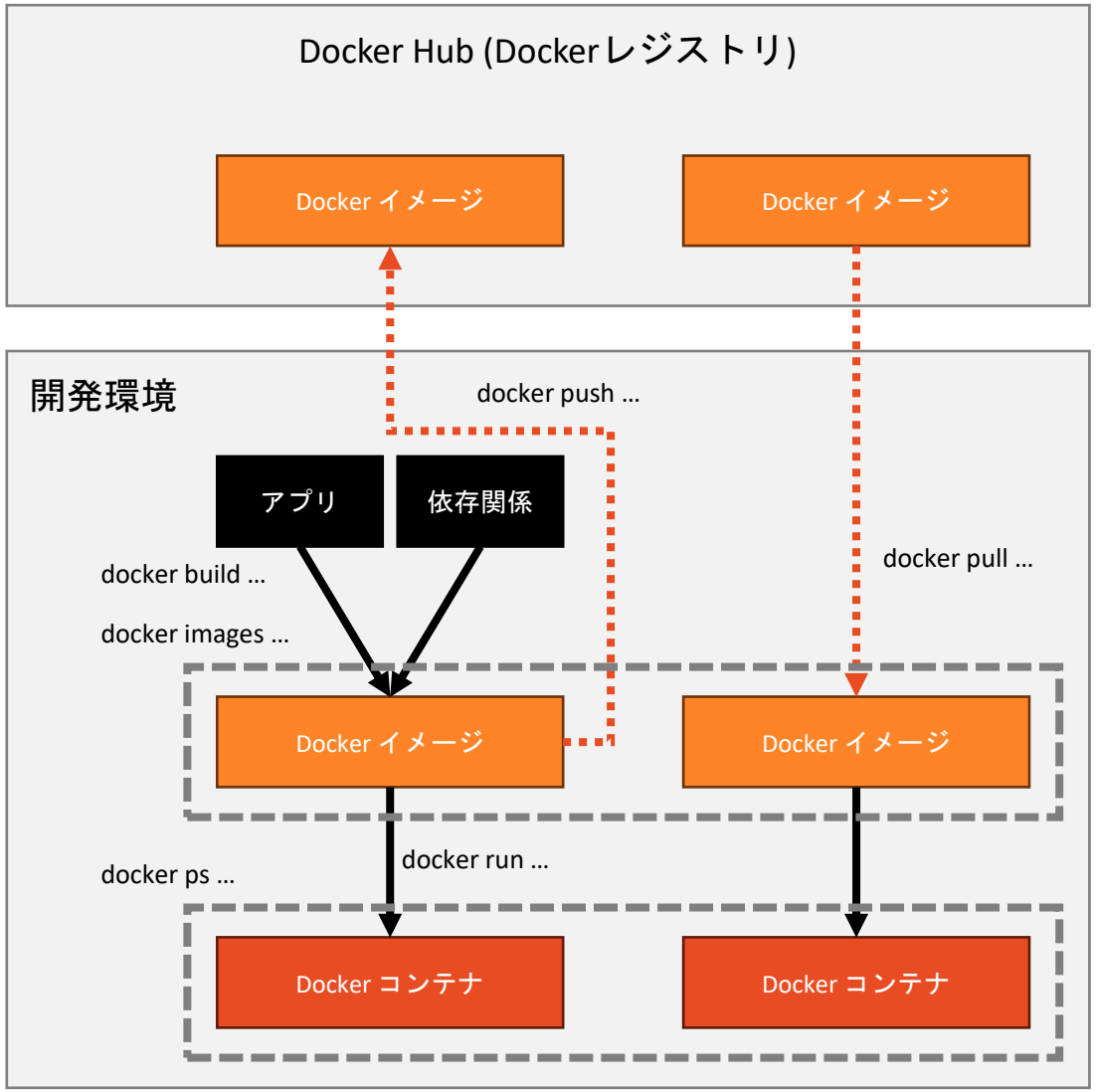
■ マイクロサービスアーキテクチャの実装

- 各マイクロサービスの独立したデプロイ
- システムの柔軟性と耐障害性の向上

■ 環境の統一と再現性の確保

- アプリケーションと依存関係のコンテナ化
- 開発、テスト、本番環境の動作の一貫性

Dockerの基本コマンドは、コンテナの作成、運用、管理を効率的に行うための不可欠です



■ 主要なDockerコマンド

- docker build: Dockerfile からイメージの作成
  - ソースコードやアプリケーションの依存関係を含むイメージ作成
- docker run: コンテナの起動
  - 指定されたイメージから新しいコンテナを作成・実行
  - オプションや引数でカスタマイズ可能
- docker ps: 実行中のコンテナの一覧表示
  - -a オプションで停止中のコンテナも表示
- docker images: ローカルのイメージ一覧表示
  - ID、リポジトリ名、タグ、作成日時などの情報表示
- docker pull: Docker Hub や他のレジストリからイメージ取得
  - ローカルに存在しないイメージのダウンロード
- docker push: ローカルのイメージをレジストリにアップロード
  - 自作イメージの公開・共有
- docker start
  - 停止しているコンテナの起動
  - 既存の停止コンテナを再起動
- docker stop
  - 実行中のコンテナの停止
  - docker startで再起動可能
- docker rm
  - コンテナの削除
  - 停止しているコンテナをシステムから完全削除
  - 一度削除したコンテナは復元不可

Dockerfileは、コンテナイメージの作成手順を定義するための指示書であり、その命令とベストプラクティスを理解します

■ 主要な命令

- FROM: ベースイメージの指定
  - 例: FROM ubuntu:20.04
- RUN: シェルコマンドの実行
  - 例: RUN apt-get update && apt-get install -y curl
- CMD: デフォルトのコマンド指定
  - 例: CMD ["echo", "Hello, World!"]
- ENTRYPOINT: デフォルトのアプリケーション指定
  - 例: ENTRYPOINT ["echo"]
- COPY: ホストからコンテナへのファイルコピー
  - 例: COPY ./app /app
- ADD: ファイルの追加 (tar解凍やURLからのダウンロードも可)
  - 例: ADD https://example.com/app.tar.gz /app
- WORKDIR: 作業ディレクトリの設定
  - 例: WORKDIR /app
- ENV: 環境変数の設定
  - 例: ENV MY_ENV_VARIABLE=value

■ Dockerfile とは

- Docker コンテナイメージを作成するためのスクリプトファイル
- ベースイメージの選択、アプリケーションのコードの追加、依存関係のインストール、環境変数の設定などが記述される

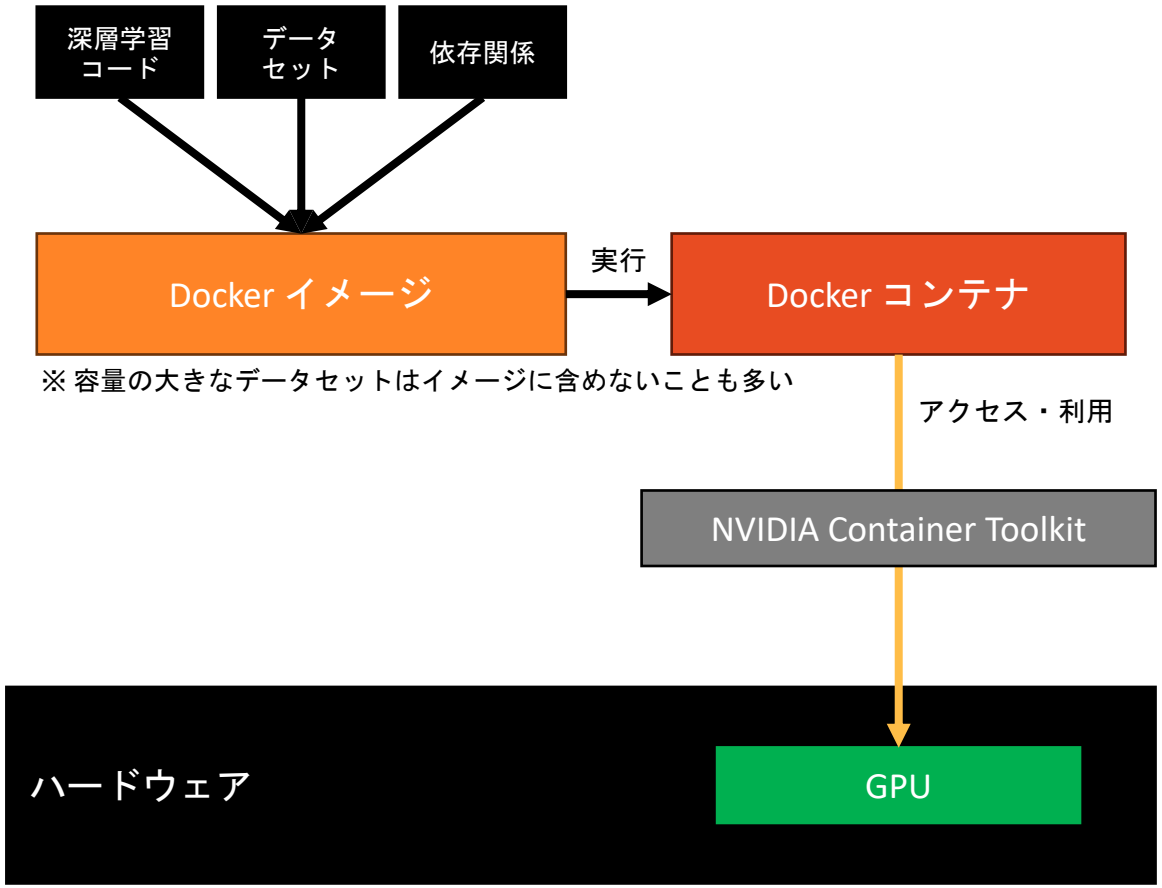
■ 基本構文

- 命令は大文字で始まり、引数が続く
- 例: INSTRUCTION arguments

■ ベストプラクティス

- .dockerignore ファイルを使用して不要なファイルを除外
- キャッシュを効率的に使用するための配置
- 不要なパッケージや一時ファイルの削除
- 複数の RUN 命令を連鎖させる
- 公式イメージの使用を推奨

Dockerを使用することで、GPUの計算リソースを効率的に活用したアプリケーションの開発とデプロイが簡単に行えます



■ 背景

- GPUはディープラーニングや高性能計算に広く利用
- DockerがGPUサポートを強化

■ Docker での GPU サポート

- NVIDIA Container Toolkit 提供
- コンテナランタイムライブラリやユーティリティを含む
- Docker、LXC、Podman などのサポート

■ GPU アプリケーションのコンテナ化

- GPUを活用したアプリケーションをDockerで実行可能
- 効率的なGPUリソースの利用

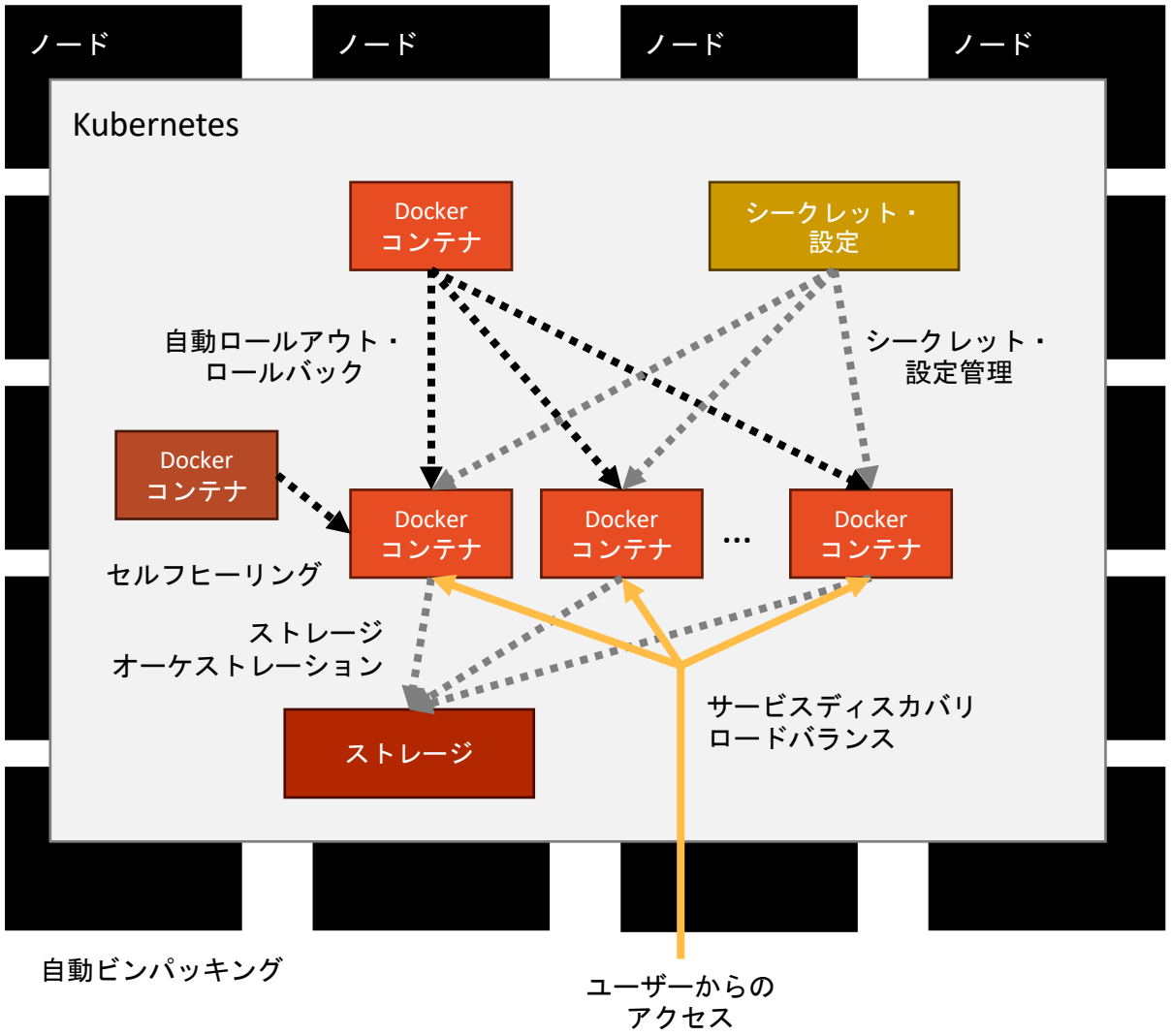
■ NVIDIA Container Toolkit の特徴

- GPU アクセラレーション: コンテナ内でのNVIDIA GPU直接利用
- エコシステムのサポート: DockerやPodman等のサポート
- 自動設定: NVIDIA GPU利用のための自動設定

■ 深層学習モデルの開発とDocker

- 一貫した環境設定の簡易化
- モデルの開発やテストの一貫性

コンテナオーケストレーションは、大規模なアプリケーションなどで複数のコンテナ管理を実現するソリューションです



■ コンテナオーケストレーション

- 定義
  - ➢ コンテナのデプロイ、スケーリング、運用の自動化プロセス
- 必要性
  - ➢ 大規模アプリケーション・マイクロサービスの効果的な管理
  - ➢ コンテナダウン時の自動復旧
- 代表的ツール
  - ➢ Kubernetes（オーケストレーションニーズ対応）

■ 主な機能:

- サービスディスカバリーとロードバランシング
- ストレージオーケストレーション
- 自動ロールアウト・ロールバック
- 自動ビンパッキング
- セルフヒーリング
- シークレット・設定管理

■ 代表的オーケストレーションツール:

- Kubernetes: オープンソース、大規模運用向け
- docker-compose: Docker 公式、開発環境・小規模デプロイ向け