



マイクロサービスブートキャンプ DAY2 コンテナ：座学

三井住友信託銀行様

アマゾンウェブサービスジャパン合同会社
プロフェッショナルサービス本部

DAY1/DAY2 のタイムテーブル

	DAY1 (1/26)	DAY2 (1/27)
10:00~12:00	オープニング 座学 サーバーレス概要／AWSのサーバーレスサービス	
12:00~13:00		昼休憩
13:00~18:00	ハンズオン① API Gateway + Lambda によるAPI構築／ DynamoDBへのアクセス	ハンズオン① 簡単なコンテナイメージの作成／ スキーマ駆動でのアプリケーション開発
18:00~19:00	ハンズオン② IaC を利用したAPIの構築／ イベント駆動によるLambda関数起動	ハンズオン② Fargate での Web アプリケーションの構築／ X-Ray によるトレーシングの有効化
	終わったチームから解散	終わったチームから解散

アジェンダ

座学

1. コンテナ概要
 1. コンテナの基礎
 2. コンテナのメリット・デメリット
 3. コンテナを支える技術
 4. コンテナに適したアプリケーション
2. AWSでのコンテナの利用
 1. コンピューティングサービスの選定
 2. AWSのコンテナ関連サービス

コンテナ概要

コンテナの基礎

背景にある課題



環境は変化し続ける

ビジネスシーンにおいて、市場の変化への迅速な対応が要求される

そのために、安全かつ迅速にプロダクトや新機能を市場へ投入したい

変化に対応するために



アプリケーションへの
フォーカス



要件に応じた
インフラストラクチャの管理

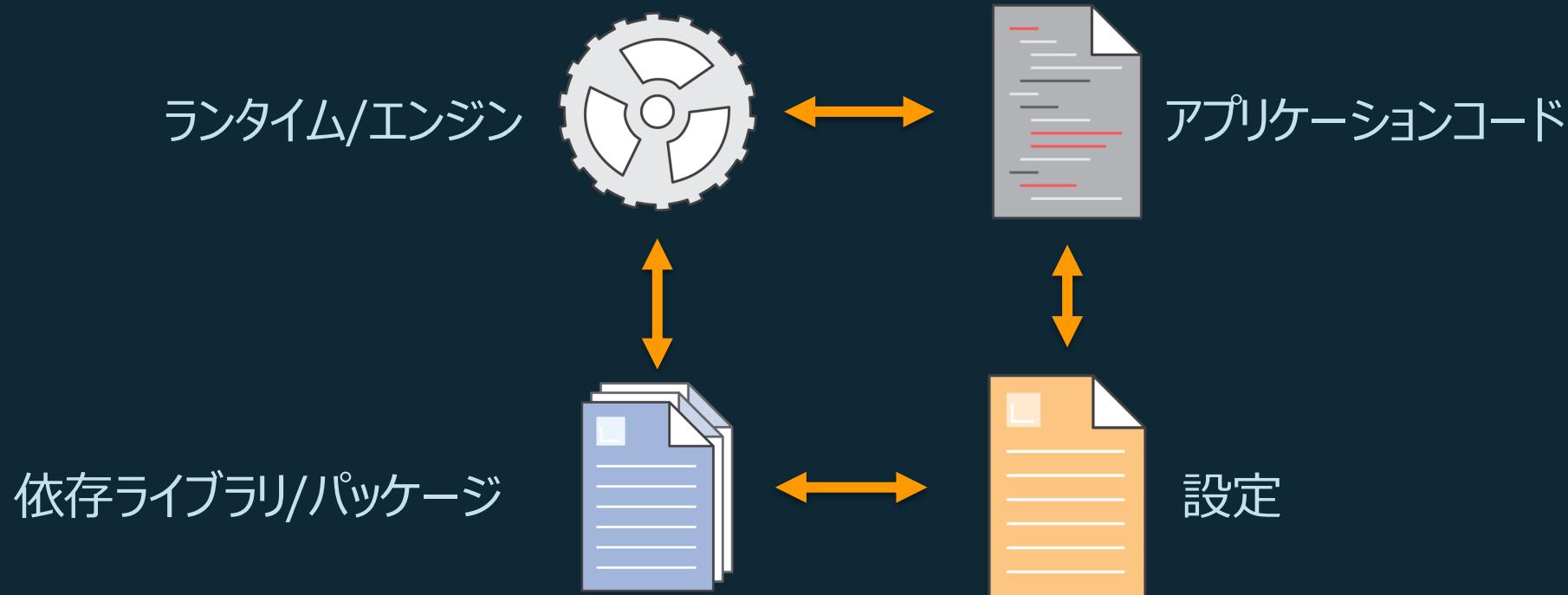


迅速なスケーリング

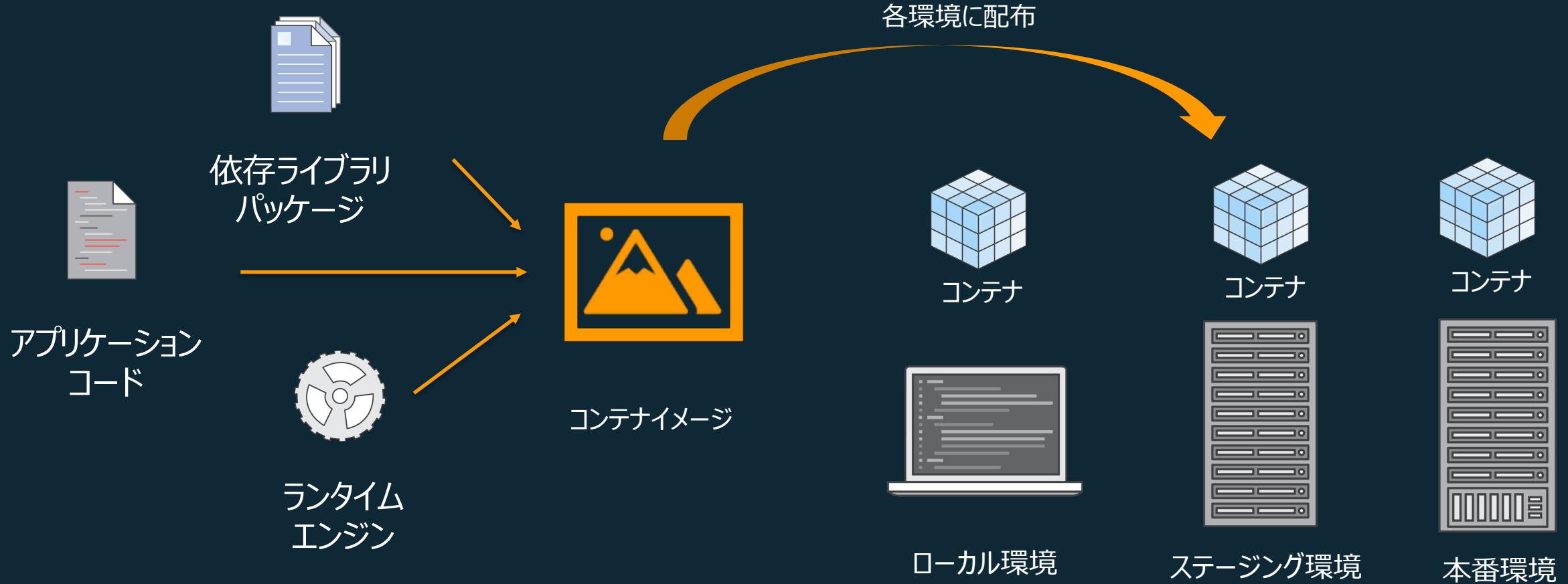


セキュリティの確保

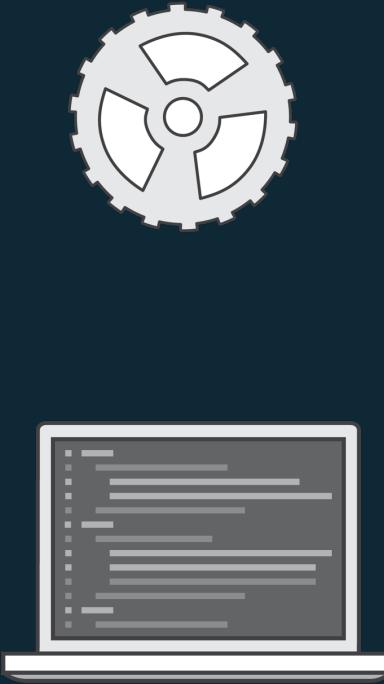
アプリケーションを構成するコンポーネント



コンテナとは？



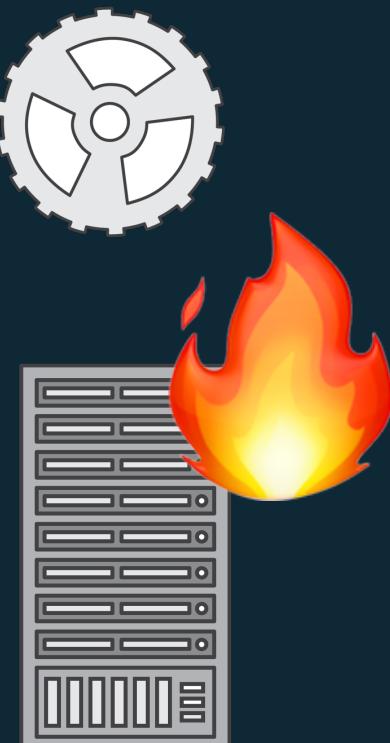
ローカルでは動いたけど、本番で動かない？



ローカルラップトップ
v6.0.0

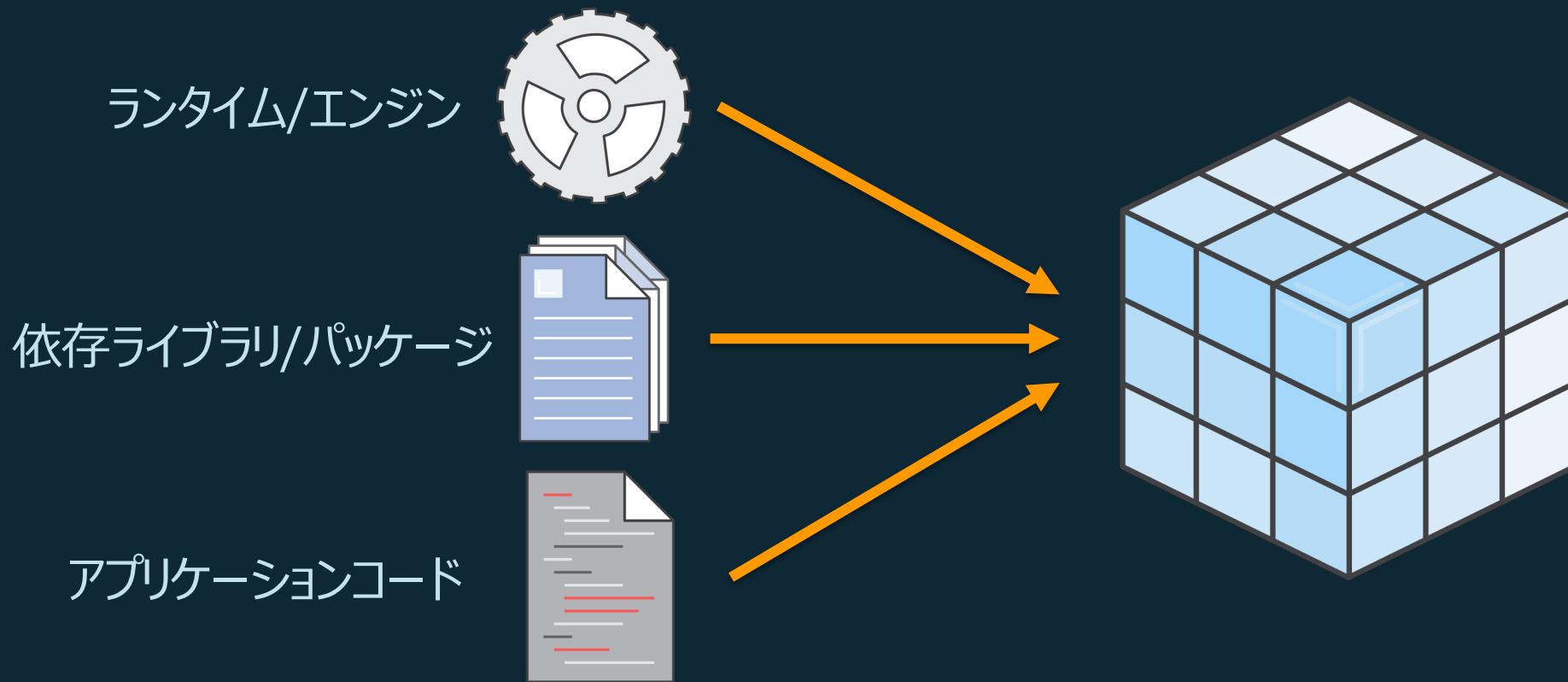


ステージング / QA
v7.0.0



本番
v4.0.0

「コンテナ」という解決策



Docker

Docker 社が開発

Apache 2.0 ライセンス

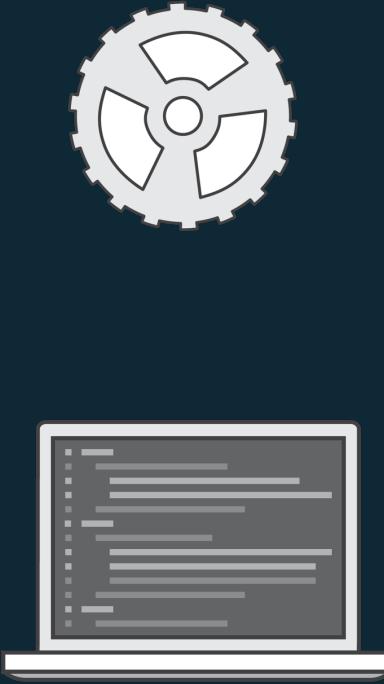
2013 年 3 月 OSS リリース

常駐型コンテナ実行エンジン

コンテナのライフサイクル管理・デプロイツール



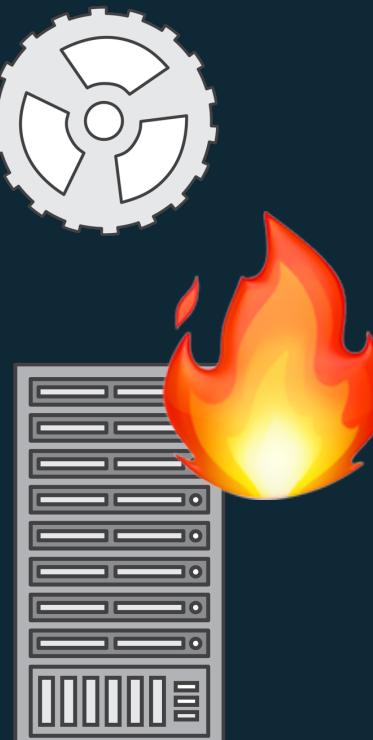
ローカルでは動いたけど、本番で動かない？（再掲）



ローカルラップトップ
v6.0.0

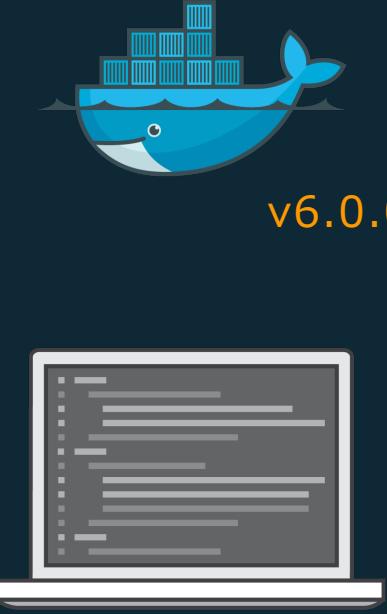


ステージング / QA
v7.0.0



本番
v4.0.0

全ての環境で同じコンテナを動かす



ローカルラップトップ



ステージング / QA



本番

なぜコンテナか

- Docker コンテナの技術的特性
 - ✓ アプリケーションの依存物全てを一つにパッケージング
 - ✓ 軽量で起動が高速
 - ✓ コンテナを実行するための統一的なコマンド群
- 技術的特性を活かすことで次のような効果が期待できる
 - ✓ アプリケーション実行環境の再現容易性
 - ✓ アプリケーションの可搬性
 - ✓ 高速な開発とリリースサイクルの実現

コンテナを選択する理由

リスクの低減



様々な環境において、
均一なセキュリティを維持

運用の効率性



「差別化に繋がらない重労働」を排除

スピード



一貫性のある環境で開発を加速

俊敏性



自動化により、テストや反復作業の
スピードと容易さを向上

コンテナのユースケース



アプリケーション



共通サービス
プラットフォーム



エンタープライズ app のマイ
グレーディング



機械学習 (ML)

Web アプリケーション

CI/CD

.NET Classic Windows app

自動運転車

モバイルアプリケーション

マネジメント、セキュリティガバナンス

レコメンデーションエンジン

IoT

ロギング、モニタリング

不正検出

データ処理

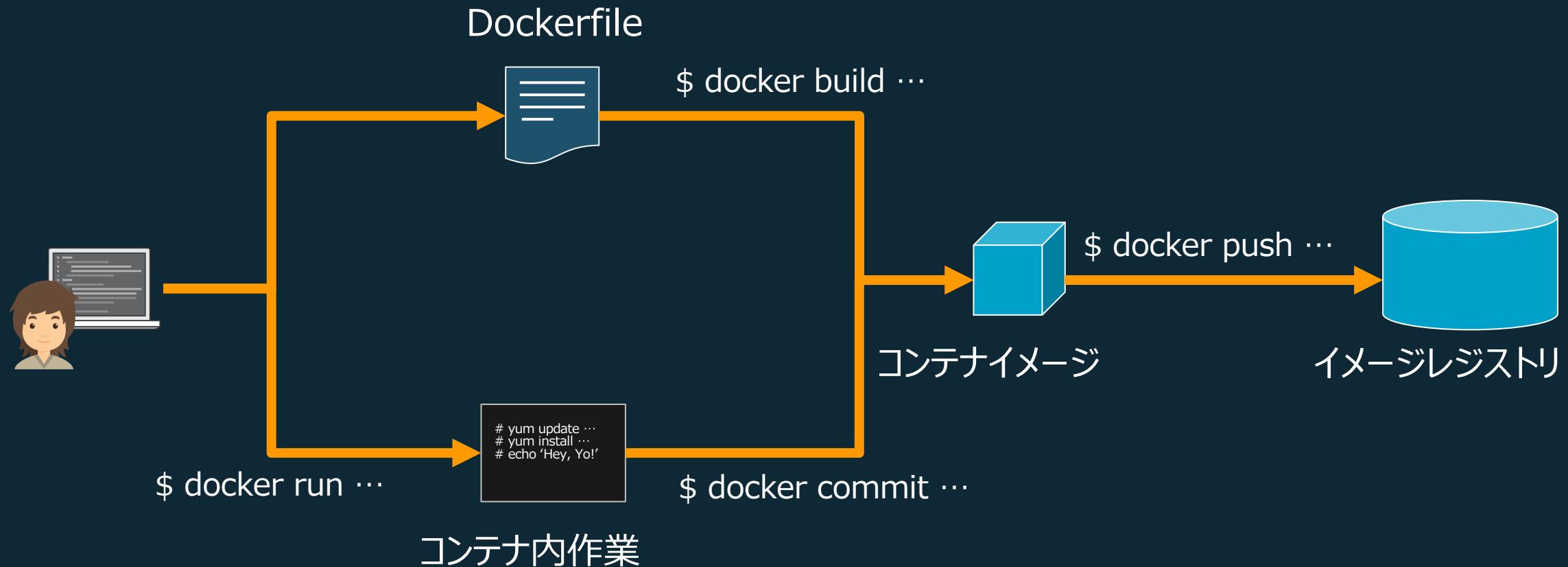
Linux app

3rd party app

Chatbots

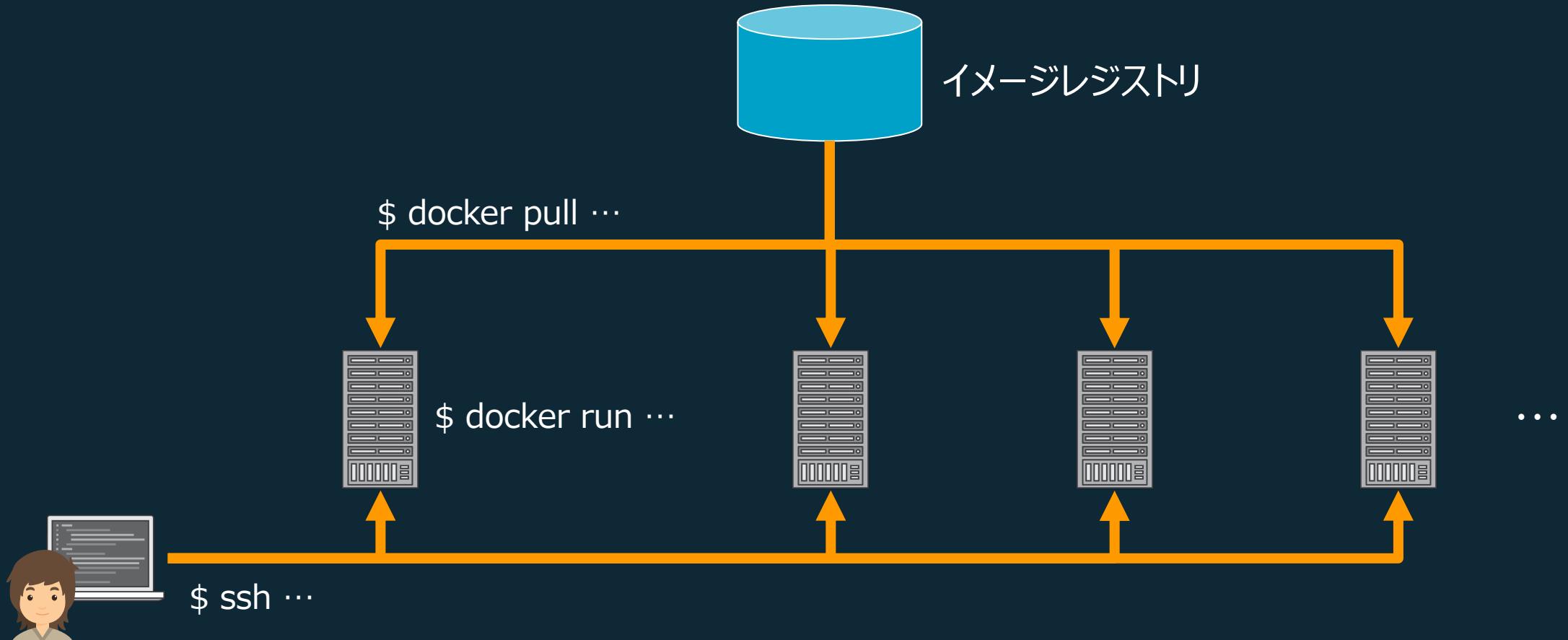
Docker を利用した基本的ワークフロー

- コンテナイメージ作成 -



Docker を利用した基本的ワークフロー

- コンテナ実行 -



コンテナのメリット・デメリット

コンテナのメリット

■ 軽量

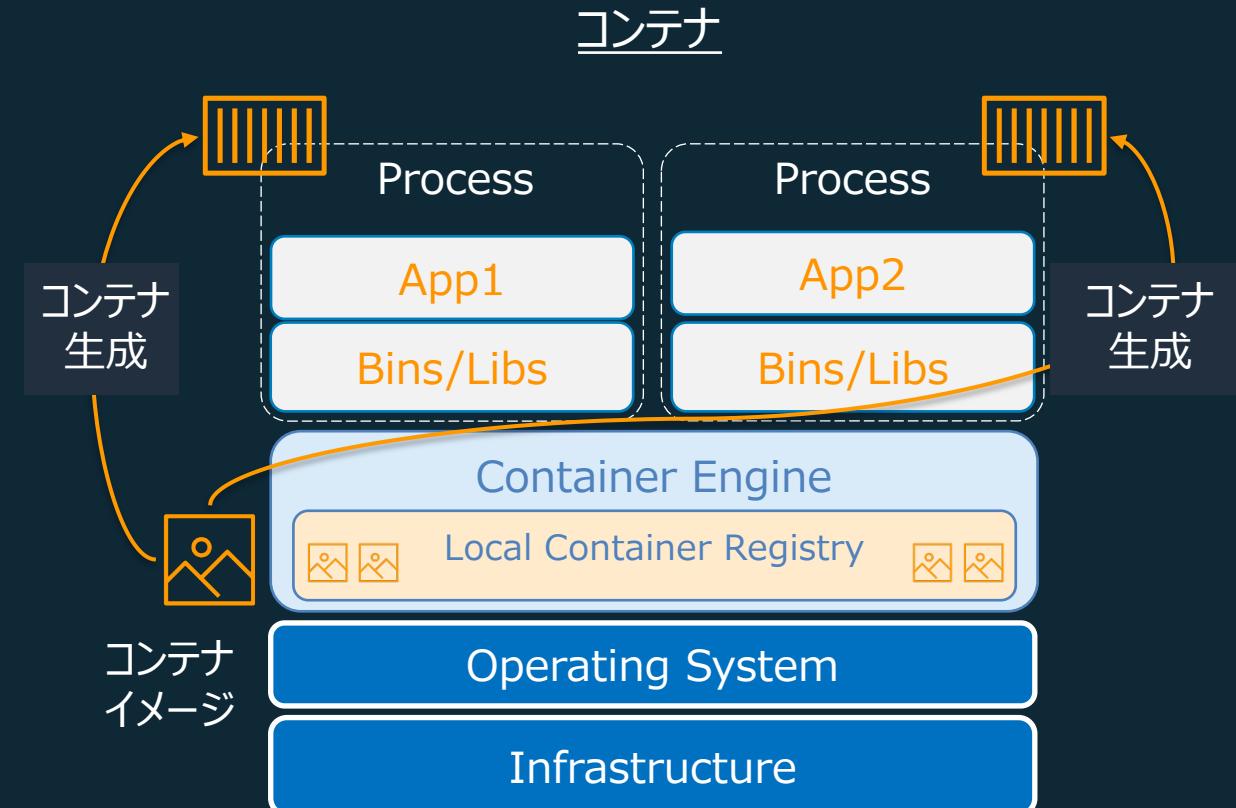
- ✓ 起動・停止が非常に高速
- ✓ スケールが容易

■ 高いリソース利用効率

- ✓ Kernel を共有するためオーバーヘッドが小さい

■ 可搬性

- ✓ コンテナイメージは「不变」
- ✓ 「アプリケーションのビルドとデプロイ」への組み込みが容易



コンテナのデメリット

コンテナやコンテナオーケストレーションのレイヤーが増えることにより、
それらについて**学習するコストや運用のコストが必要**

- Dockerfile の作成
- ECS や Kubernetes の学習
- Kubernetes などのバージョンアップ
- 周辺のエコシステムツール群のキャッチアップ
- など

仮想マシンとは異なり、アプリケーションを修正しなければならないこともある
コンテナはカーネルを共有するため、セキュリティ要件を再検討する必要がある

どのようなシステムをコンテナ化するとよいのか

コンテナに向いたシステム

- コンテナ化のメリットを活用したいアプリケーション
 - 頻繁なデプロイを行うアプリケーション
 - 高速なスケールを必要とするアプリケーション
 - 例) コンシューマー向け Web アプリケーション
- ステートレス化が可能なアプリケーション

コンテナにあまり向かないシステム

- コンテナ化のメリットがあまり関係のないアプリケーション
 - 頻繁にデプロイを行わないアプリケーション
- ステートフルなアプリケーション
 - データベースなど
 - 高速なファイルシステム I/O が必要
- パッケージアプリケーションなど制約がある場合

※ 更新頻度の少ないレガシーアプリケーションにおいてもメリットはある

- 運用効率化（オートヒーリング、オートスケーリング など）
- ハードウェア・OS・アプリケーションのレイヤー分離による保守コスト最適化
- 余剰リソースの集約によるランニングコスト最適化
- コンテナ内で利用するモジュールやライブラリを明示的に記述することによる構成管理 など

コンテナ利用に対する各役割の目的（ねらい）

「Ops と Dev の境界分離」&
「厳密な同一性を維持した形で、成果物がテストから本番環境まで稼働する」



アプリ開発(Dev)



インフラ運用(Ops)



- ・開発物(コード)の同一性を担保
- ・アプリから見て OS や M/W 層までを制御下に置くことが可能
- ・CI/CD をより容易に実現
- ・ステートレスなアプリケーションテクチャ採用への後押し

コンテナ
[代表的特性]

- ・スケール性が仮想マシン (VM) と比較して優位
- ・一度ビルドされたコンテナイメージは不变 (Immutable)
- ・コンテナが削除されるとコンテナ内データも削除

- ・運用の範囲・工数を「コンテナ基盤」に集中できる
- ・リソース効率向上が期待できる
- ・インフラのコード化 (Infrastructure as Code) 実現による自動化範囲の拡大

それぞれ異なる目的(=恩恵)あり

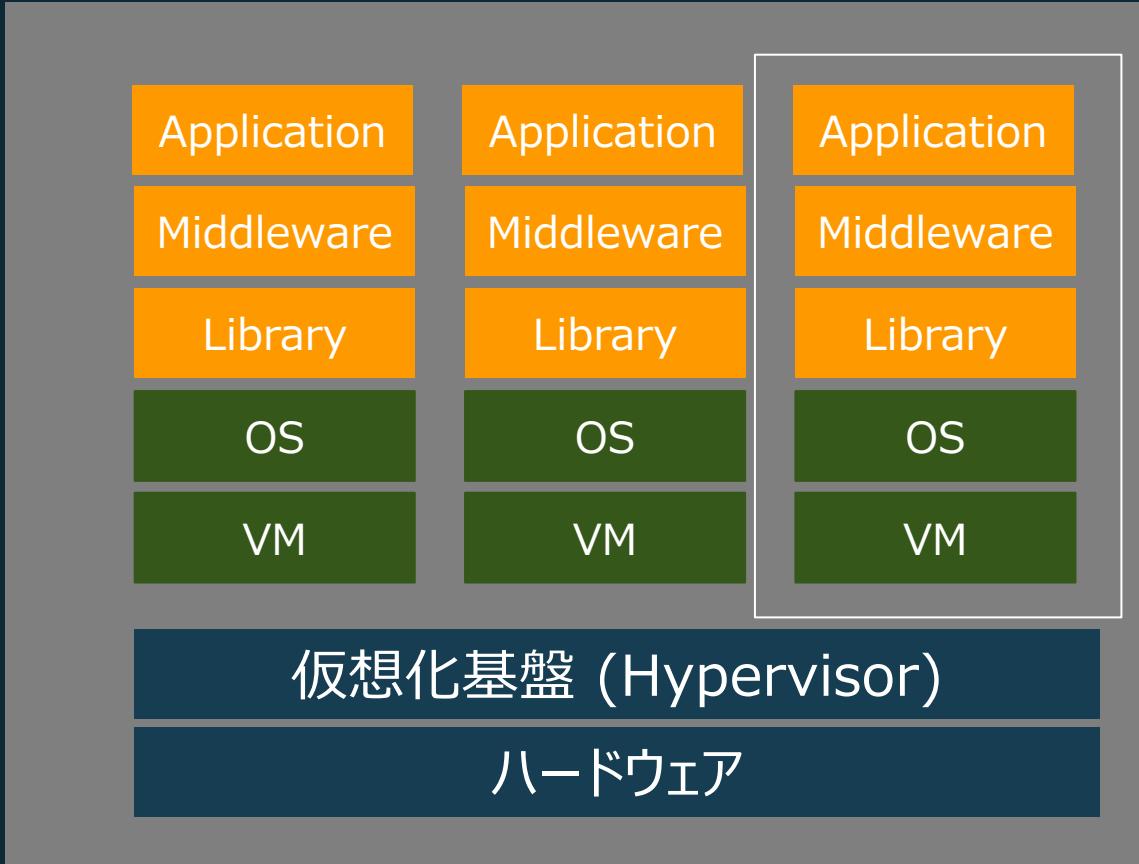
A photograph of a person's hand raised in the air, palm facing forward, against a blurred background of colorful stage lights. The background is a solid teal color.

Q&A

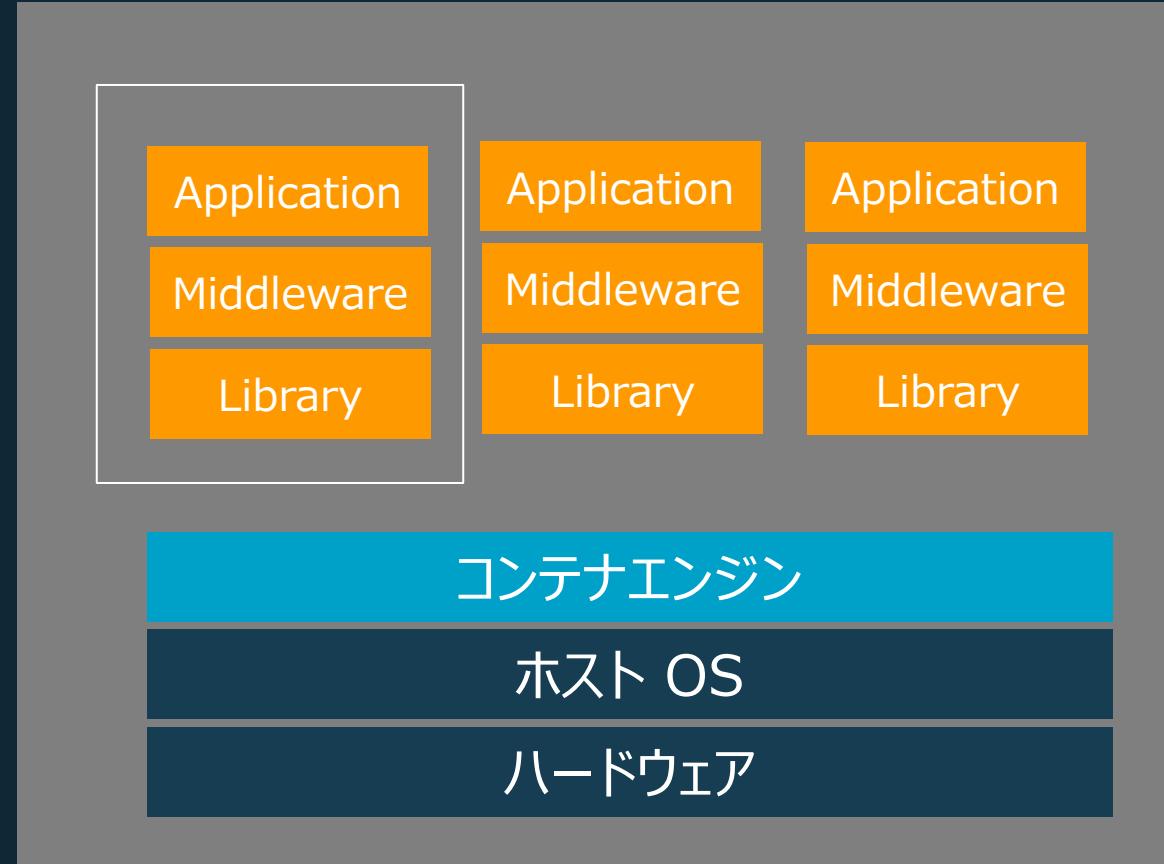
コンテナを支える技術

仮想マシンとコンテナ

仮想マシン

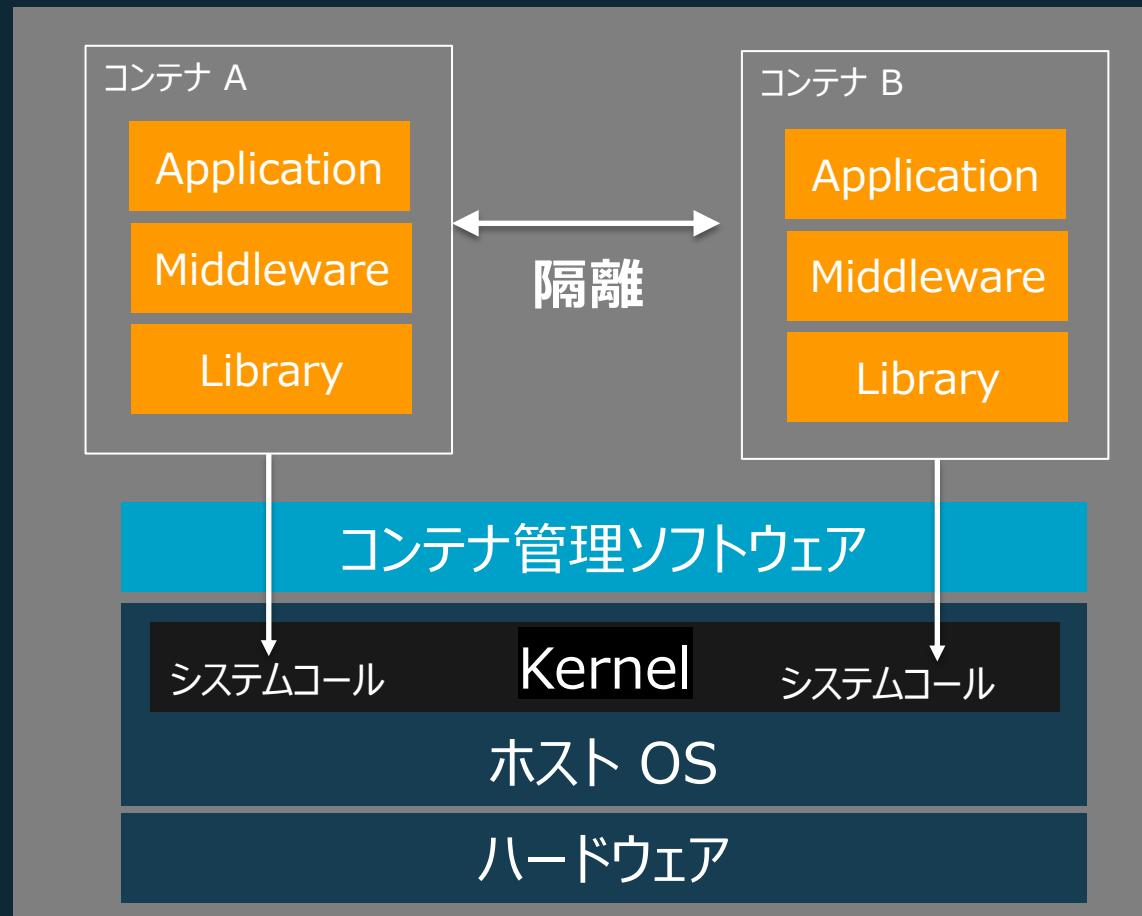


コンテナ



コンテナ

コンテナ



- コンテナアプリケーションはホスト OS のカーネルを直接利用
- コンテナは隔離されたプロセス

ホスト OS から見た Docker コンテナのプロセス

- ホスト OS でプロセス確認 (ps aux)
 - 実行中のコンテナアプリケーションがプロセスとして表示される

```
root  3310  0.2  2.2 1498496 44944 ?      Ssl 11:44 0:00 /usr/bin/containerd
root  6216  0.0  0.3 708828 6952 ?      Sl 11:46 0:00 \_ containerd-shim -namespace moby -workdir /var/lib/containerd/io.containerd.runtime.v1.linux/moby/f3d7b09d27b51541e3620440d3
root  6252  0.2  0.2 10636 5762 ?      Ss 11:46 0:00 | \_ nginx: master process nginx -g daemon off;
101   6315  0.0  0.1 11032 2640 ?      Ss 11:46 0:00 | \_ nginx: worker process
root  6436  0.0  0.2 708444 5660 ?      Sl 11:47 0:00 \_ containerd-shim -namespace moby -workdir /var/lib/containerd/io.containerd.runtime.v1.linux/moby/d390655be02e3956b4e5174843
root  6472  5.0  0.2 5940 4360 ?      Ss 11:47 0:00 \_ httpd -DFOREGROUND
bin   6515  0.0  0.1 1210608 3528 ?      Sl 11:47 0:00 \_ httpd -DFOREGROUND
bin   6516  0.0  0.1 1210608 3528 ?      Sl 11:47 0:00 \_ httpd -DFOREGROUND
bin   6518  0.0  0.1 1210608 3528 ?      Sl 11:47 0:00 \_ httpd -DFOREGROUND
```

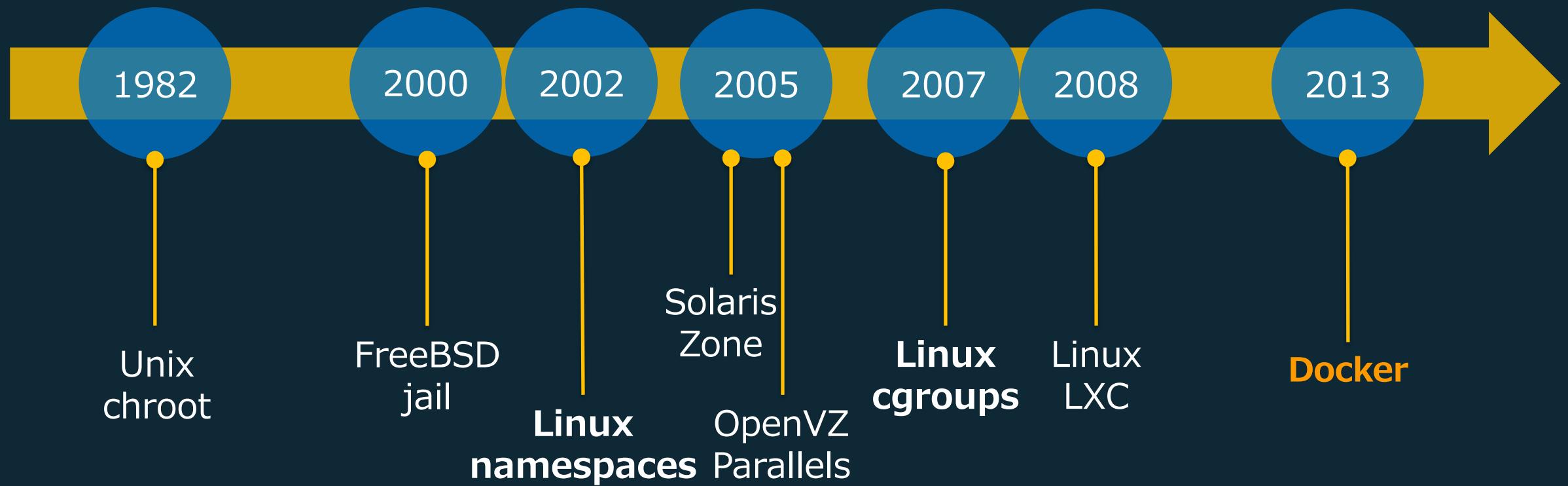
Dockerランタイム
Dockerランタイム配下のプロセスとして
nginxとApache httpdが動作中

- 実行中コンテナにアタッチしてプロセス確認
 - コンテナ内のプロセスのみ表示される

```
root@d390655be02e:/usr/local/apache2# ps aux
USER      PID %CPU %MEM    VSZ   RSS TTY      STAT START  TIME COMMAND
root        1  0.0  0.2    5940   4360 ?      Ss 11:47 0:00 httpd -DFOREGROUND
daemon     7  0.0  0.1 1210608  3528 ?      Sl 11:47 0:00 httpd -DFOREGROUND
daemon     8  0.0  0.1 1210608  3528 ?      Sl 11:47 0:00 httpd -DFOREGROUND
daemon    10  0.0  0.1 1210608  3528 ?      Sl 11:47 0:00 httpd -DFOREGROUND
root       91  0.1  0.1    3972   3128 pts/0    Ss 11:53 0:00 bash
root      419  0.0  0.1    7640   2740 pts/0    R+ 11:55 0:00 ps aux
```

コンテナ技術の歴史

コンテナ = “リソース隔離を実現する技術”



コンテナを支える基本技術

- Namespaces
 - ホスト OS のシステムリソースを名前空間で隔離
 - プロセス ID (PID 名前空間)、ホスト名 (UTS 名前空間) 、 UID/GID (ユーザー名前空間)、IP アドレス (ネットワーク名前空間)、マウント (マウント名前空間) etc.
- cgroups
 - ホスト OS の物理リソースをプロセス毎に隔離
 - CPU, Memory, ディスク IO etc.
- いざれも Linux Kernel の機能

なぜコンテナが注目されているのか？

コンテナ自体は別に新しい技術ではないが、**Docker** が開発プロセスを整理したことで、主に DevOps の文脈で**コンテナが再発見**され、昨今のブームを作っている

肝となるのは CLI ツールと Dockerfile とイメージレジストリ

- CLI ツール：非常に直感的な操作
 - Dockerfile：シンプルな Provisioning スクリプト
 - イメージレジストリ：スムーズにイメージを共有
- 開発者が簡単にコンテナイメージを作成・共有できる



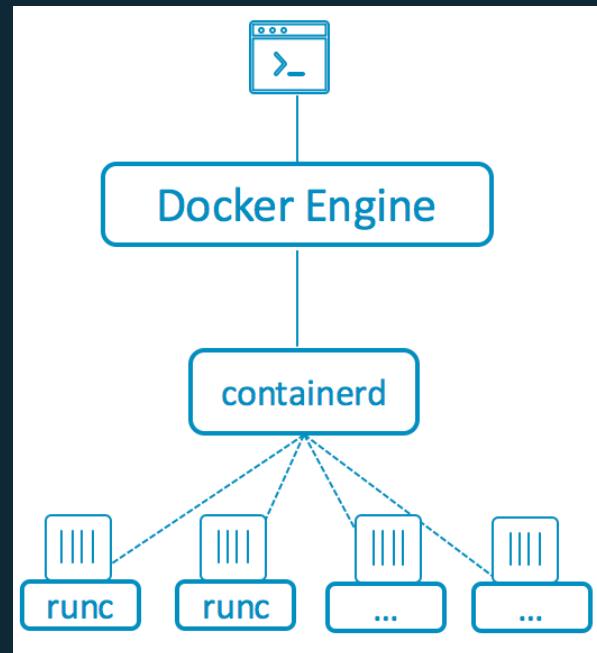
コンテナの互換性

- Open Container Initiative (OCI)
 - Docker の人気の高まりを受けて、複数存在したコンテナ仮想化実装のオープン業界標準策定を目的として2015年に設立された団体
- OCI が策定した標準仕様
 - OCI Image Specification イメージの標準仕様
 - OCI Runtime Specification 実行ランタイムの標準仕様
 - OCI Distribution Specification レジストリの標準仕様

https://ja.wikipedia.org/wiki/Open_Container_Initiative

Docker の構成 1.11 ~

- コンテナランタイム = Docker という状況だったが、その技術仕様は標準化され、Docker 自身もオープンな実装となっている



containerd

高レベルランタイム
コンテナイメージの Pull, コンテナの実行を管理
Docker 社が開発、2017 年に Cloud Native Computing Foundation へ寄贈

runC

低レベルランタイム
イメージからコンテナの生成、実行を行う
2017 年に Docker から分離し OCI が管理

コンテナのワークフロー



Dockerfile のサンプル

```
FROM ruby:2.5
RUN apt-get update -qq && apt-get install -y nodejs postgresql-client
WORKDIR /myapp
COPY Gemfile /myapp/Gemfile
COPY Gemfile.lock /myapp/Gemfile.lock
RUN bundle install
COPY . /myapp

# Add a script to be executed every time the container starts.
COPY entrypoint.sh /usr/bin/
RUN chmod +x /usr/bin/entrypoint.sh
ENTRYPOINT ["entrypoint.sh"]
EXPOSE 3000

# Start the main process.
CMD ["rails", "server", "-b", "0.0.0.0"]
```

Quickstart: Compose and Rails
<https://docs.docker.com/compose/rails/>

クイズ

- コンテナはホスト OS のカーネルを利用します
- Linux にはさまざまなディストリビューションがあり、ディストリビューションやバージョンによって Linux カーネルのバージョンは異なります
- それなのになぜコンテナには可搬性がある（どこでも動く）のでしょうか？

クイズ

- Linux ではない macOS や Windows で Docker でコンテナが動くのはなぜでしょうか？

コンテナの理解を深めるポイントまとめ

- コンテナは隔離されたプロセス
 - Linux Kernel の機能 (namespaces, cgroups) で隔離を実現している
 - 仮想マシンとは異なり、ただのプロセスなので非常に軽量
- Linux カーネルの後方互換性が保たれているため、コンテナはどこでも実行可能

A photograph of a person's hand raised in the air, palm facing forward, against a blurred background of colorful stage lights. The background is a solid teal color.

Q&A



コンテナに適したアプリケーション

コンテナに対応したアプリケーションの設計

The Twelve-Factor App

- <https://12factor.net/ja/>
- 2011 年に Heroku のエンジニアが提唱
- アプリケーションを疎結合にするための指針・方法論
- クラウド環境と相性が良い
 - デプロイ、起動、停止が容易
 - スケールアウト、スケールインが容易
- コンテナアプリケーションにも有効

Twelve-Factor App

I. コードベース

バージョン管理されている1つのコードベースと複数のデプロイ

II. 依存関係

依存関係を明示的に宣言し分離する

III. 設定

設定を環境変数に格納する

IV. バックエンドサービス

バックエンドサービスをアタッチされたリソースとして扱う

V. ビルド、リリース、実行

ビルド、リリース、実行の3つのステージを厳密に分離する

VI. プロセス

アプリケーションを1つもしくは複数のステートレスなプロセスとして実行する

VII. ポートバインディング

ポートバインディングを通してサービスを公開する

VIII. 並行性

プロセスモデルによってスケールアウトする

IX. 廃棄容易性

高速な起動とグレースフルシャットダウンで堅牢性を最大化する

X. 開発/本番一致

開発、ステージング、本番環境をできるだけ一致させた状態を保つ

XI. ログ

ログをイベントストリームとして扱う

XII. 管理プロセス

管理タスクを1回限りのプロセスとして実行する

最初にやるべきポイント

- III. 設定 (設定を環境変数に格納する)
 - 設定はイメージに含めず、環境変数などで実行時に外から注入する
- VI. プロセス (アプリを1つもしくは複数のステートレスなプロセスとして実行する)
 - 永続化する必要があるデータはデータベースなどのバックエンドサービスに保管する
 - ステイッキー・セッションは利用せず、セッションはバックエンドサービスに保管する
 - ステートフルアプリケーションでのコンテナ利用は避ける
- XI. ログ (ログをイベントストリームとして扱う)
 - ログはファイルに書かず、ストリームとして外部に送信する
 - アプリケーションとしては stdout/stderr に出し、CloudWatch や EFK スタックなどによって収集する

III. 設定

設定を環境変数に格納する

- 💡 設定値をソースコードや設定ファイルとしてイメージに含める

```
myapp.conf
// ** MySQL settings **
define('DB_NAME','mydb');
define('DB_USER','mydbuser');
define('DB_PASSWORD','mydbpassword');
define('DB_HOST','192.168.0.100');
```

- 💡 設定値を環境変数から取得する

- 💡 設定値の保管に SSM パラメータストアや Secrets Manager を利用する

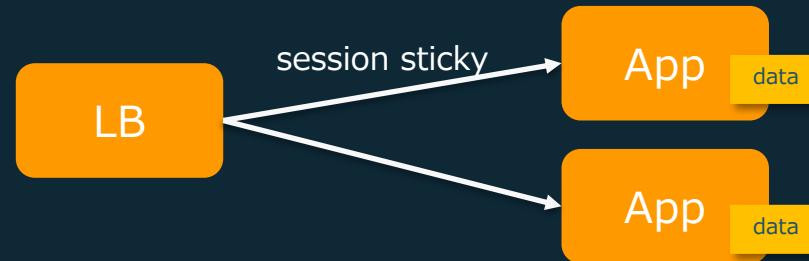
```
myapp.conf
// ** MySQL settings **
define('DB_NAME','ENV_DB_NAME');
define('DB_USER','ENV_DB_USERNAME');
define('DB_PASSWORD','ENV_DB_PASSWORD');
define('DB_HOST','ENV_DB_HOSTNAME');
```

```
# コンテナの起動時に環境変数から設定ファイルを書き換える
sed -i "s/ENV_DB_NAME/$DB_NAME/" myapp.conf
sed -i "s/ENV_DB_USERNAME/$DB_USERNAME/" myapp.conf
sed -i "s/ENV_DB_PASSWORD/$DB_PASSWORD/" myapp.conf
sed -i "s/ENV_DB_HOSTNAME/$DB_HOSTNAME/" myapp.conf
# start myapp
```

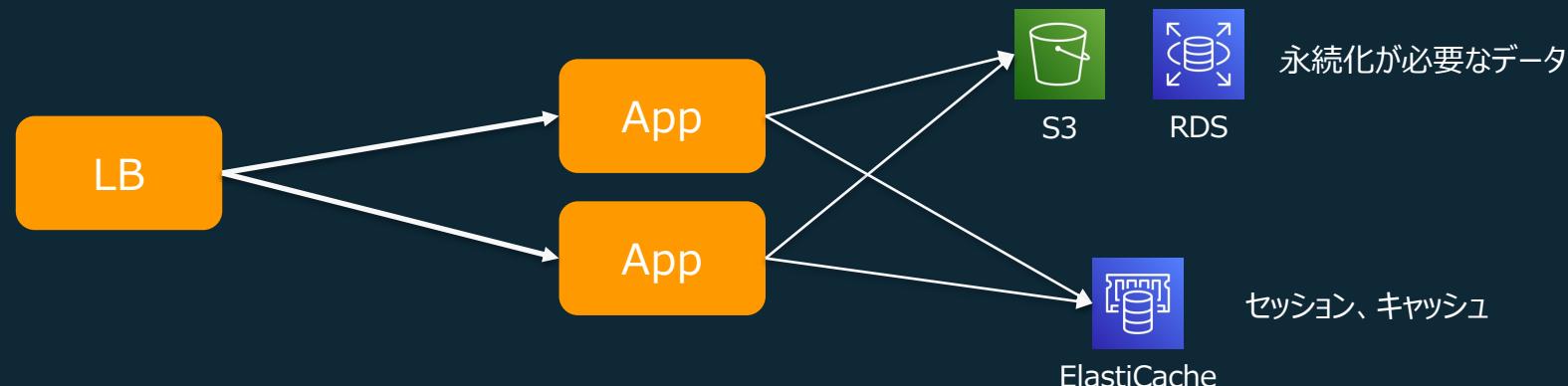
VI. プロセス

アプリケーションを1つもしくは複数のステートレスなプロセスとして実行する

- データをローカルにファイルとして保存する
- ステイッキーセッションを利用する



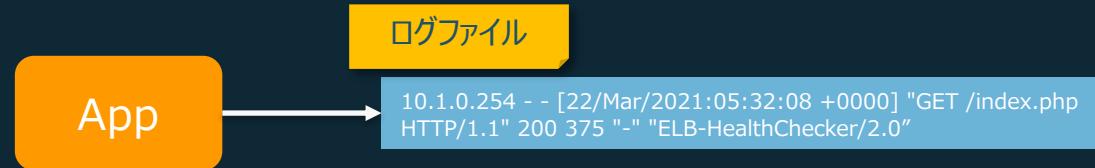
- 永続化が必要なデータを S3 やデータベースに保管する
- セッション情報は DynamoDB や ElastiCache に保管する



XI. ログ

ログをイベントストリームとして扱う

- 👉 ログファイルに書き込んだり管理しようとする
- 👉 アプリケーションログの送り先やストレージについて設定する



- コンテナが破棄されるとログも消失してしまう
- アプリ側でログの永続化や転送先まで設定すると設定が複雑に

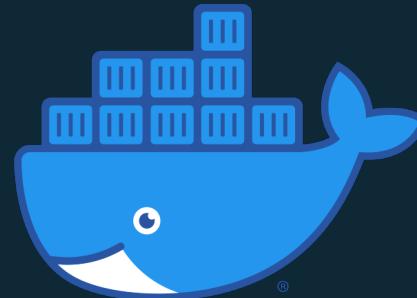
- 👉 ログストリームを標準出力にバッファリングせずに書き出す
- 👉 ログストリームの保存のための目的地はアプリケーションから設定せずに、実行環境によって管理する



改めて…コンテナとは

商船三井 サービスサイト ブログ
20世紀最大の人類の発明は、鉄の箱？！
～世界の物流を変えたイノベーション～

<https://www.mol-service.com/ja/blog/greatest-inventions>



Docker のロゴマーク

<https://www.docker.com/company/newsroom/media-resources>

コンテナは従来のアプリケーション開発・テスト・配布フローを置換
コンテナイメージがアプリケーションの流通対象となる可能性を持つ



コンテナ物語 世界を変えたのは「箱」の発明だった 増補改訂版 単行本 – 2019/10/24

マルク・レビンソン (著), 村井 章子 (翻訳)

★★★★★ 571個の評価

すべての形式と版を表示

Kindle版 (電子書籍)
¥2,772
獲得ポイント: 28pt

今すぐお読みいただけます: 無料アプリ

単行本
¥3,080
獲得ポイント: 92pt prime

¥2,485 より 8 中古品
¥3,080 より 10 新品
¥5,484 より 2 コレクター商品

読書の秋 本のまとめ買いキャンペーン
エントリーはこちら 11/1まで

A photograph of a person's hand raised in the air, palm facing forward, against a blurred background of colorful stage lights. The background is a dark teal color.

Q&A

AWSでのコンテナの利用

コンピューティングサービスの選択

amazon



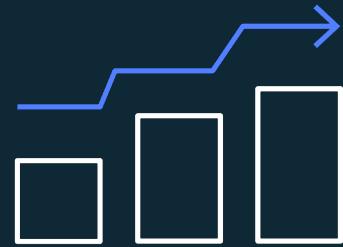
マイクロサービスの観点からみたサーバーレスとコンテナの使い道

サーバーレス	コンテナ
<ul style="list-style-type: none">✓ Web画面から呼び出されるAPI✓ モバイルアプリケーションから呼び出されるAPI✓ イベントトリガーによる処理の実行 (例 : S3へのファイルアップロード)	<ul style="list-style-type: none">✓ Web画面から呼び出されるAPI✓ モバイルアプリケーションから呼び出されるAPI✓ 処理に時間がかかるても問題ないワークフロー

サーバーレスのメリット



サーバ管理が不要



自動でスケール



従量課金



高可用性

<https://aws.amazon.com/jp/serverless/patterns/start-serverless/>

サーバーレスのデメリット

自由度とのトレードオフ = 制約を受け入れる



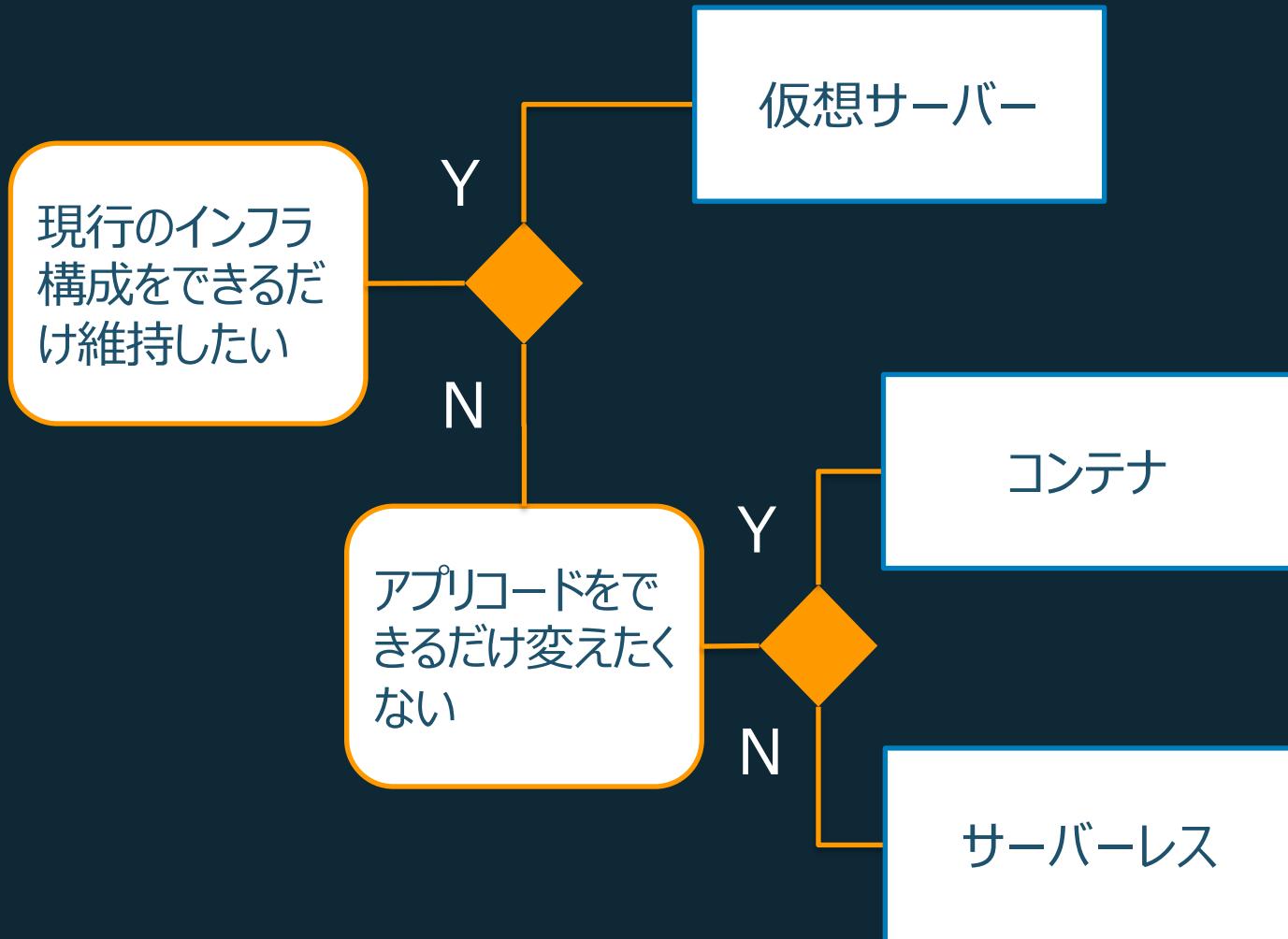
制約



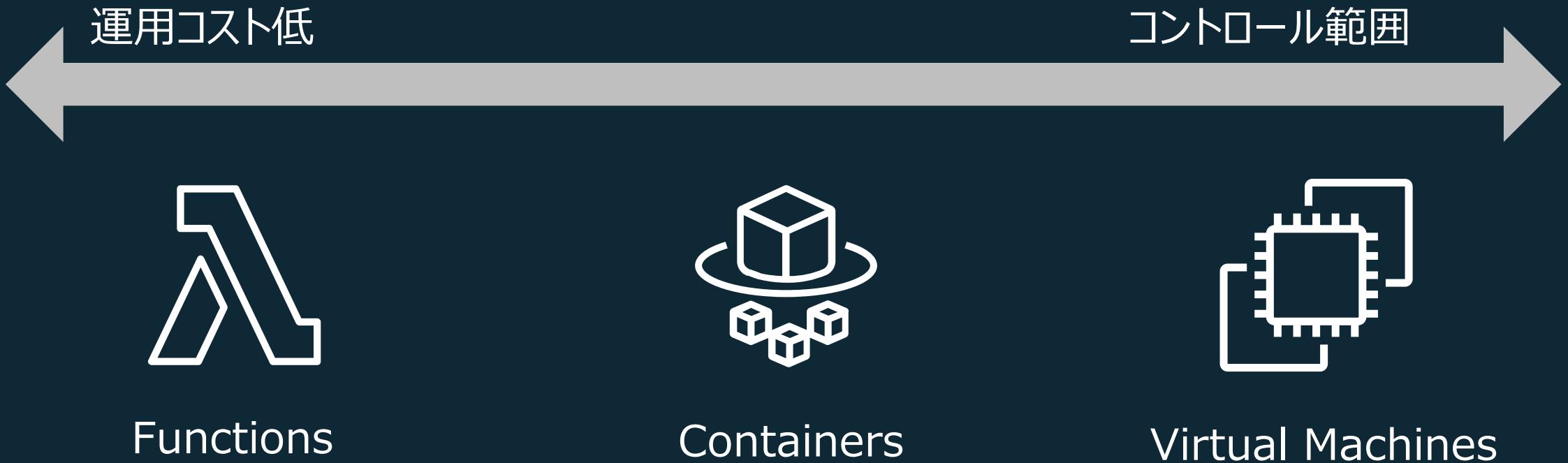
- 既存のアプリケーションをリホスト（Lift & Shift）で移行するのが困難
- サービスロックインしやすい
- サービス固有の制約（例えば、Lambdaの場合、最大処理時間15分まで、等のHard Limit）

多くはメリットが優位であるが
享受できない制約がある場合、サーバーレスは不向き

コンピューティング環境の選択



コンピューティングの多様な選択肢



適切なコンピューティングサービスの選択が重要

AWS が提供するコンピュートの多様な選択肢



Lambda
Serverless functions



Fargate
Serverless containers



Amazon ECS /
Amazon EKS
Container management
as a service



Amazon EC2
Infrastructure as a
service

AWS がやること

データソース統合
物理ハードウェア、ソフトウェア、ネットワーク、
ファシリティ
プロビジョニング

お客様がやること

アプリケーションコード

コンテナのオーケストレーション、プロビジョニング、クラスタのスケーリング
物理ハードウェア、ホスト OS / カーネル、
ネットワーク、ファシリティ

アプリケーションコード
データソース統合
セキュリティ設定と更新

ネットワーク設定
管理タスク

コンテナのオーケストレーション、コントロールプレーン
物理ハードウェア、ソフトウェア、ネットワーク、ファシリティ

アプリケーションコード
データソース統合
クラスター
セキュリティ設定と更新

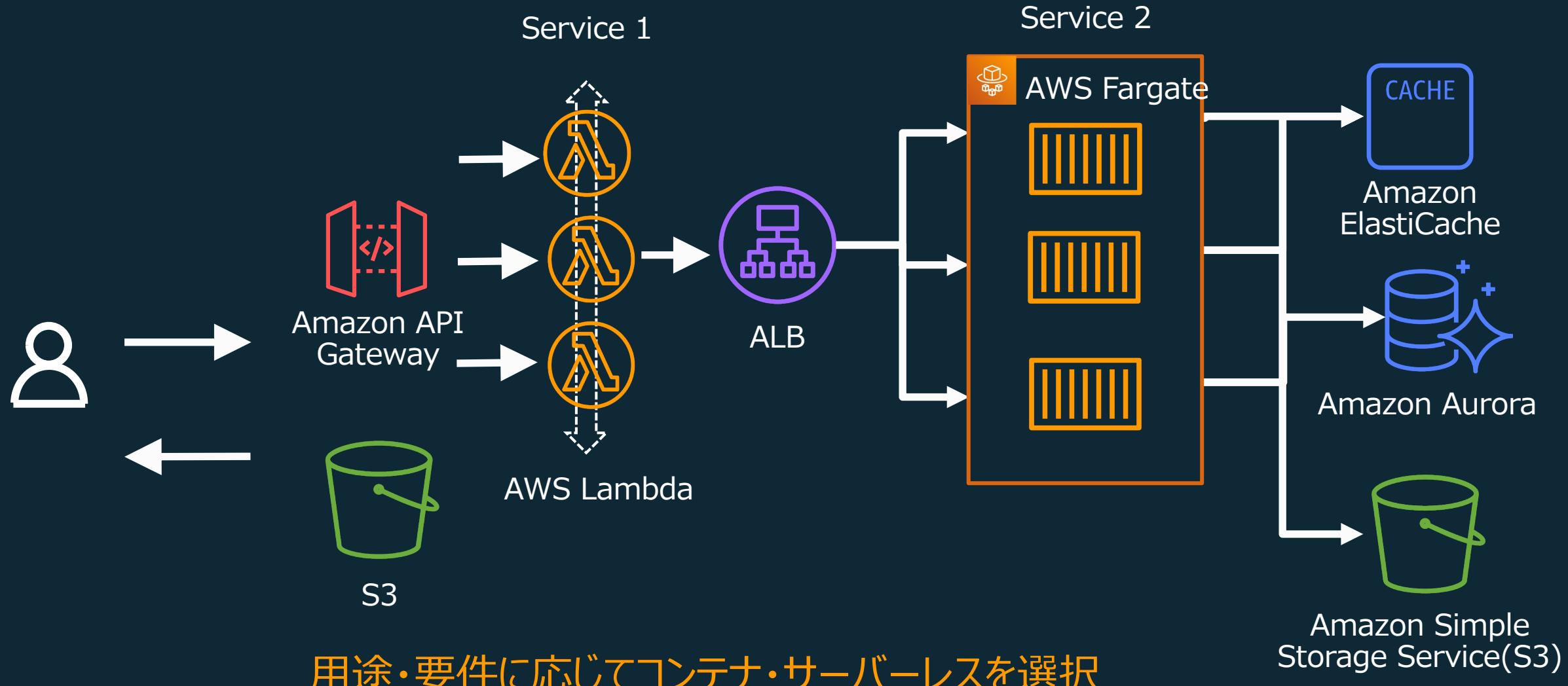
ネットワーク設定
ファイアウォール
管理タスク

物理ハードウェア、ソフトウェア、ネットワーク、ファシリティ

アプリケーションコード
データソース統合
スケーリング
セキュリティ設定と更新
ネットワーク設定

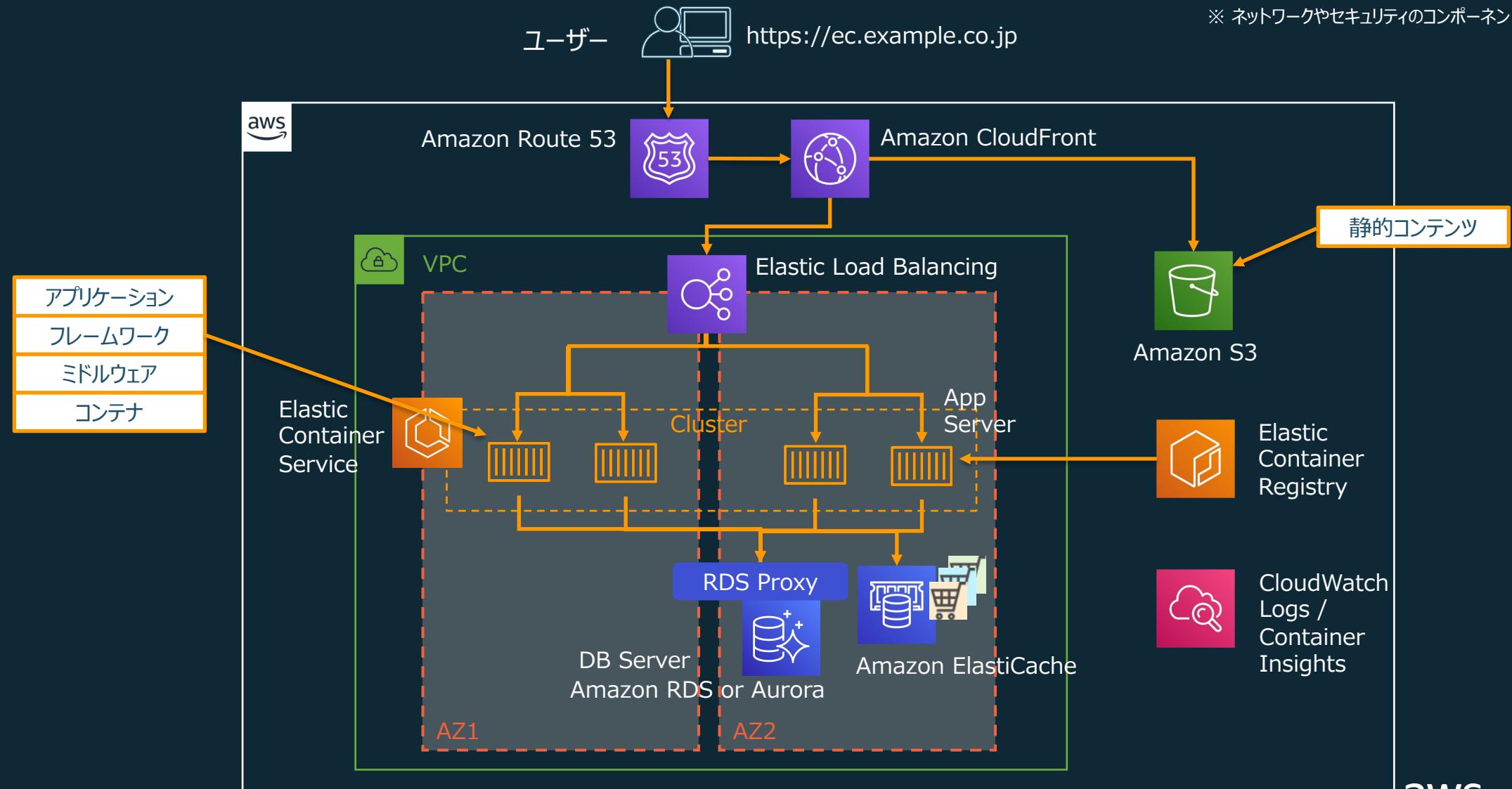
管理タスク
プロビジョニング
スケーリング管理
サーバーのパッチング

AWS のコンテナ・サーバーレスサービス活用例



AWS 上での基本的な EC サイト構成例

※ ネットワークやセキュリティのコンポーネントは省略しています



A photograph of a person's hand raised in the air, palm facing forward, against a blurred background of colorful stage lights. The background is a solid teal color.

Q&A

AWSのコンテナ関連サービス

amazon

AWS のコンテナ関連サービス

オーケストレーション

コンテナのデプロイ、スケジューリング、スケーリング、クラスター管理



Amazon ECS



Amazon EKS

コンテナ実行基盤

コンテナの実行



Amazon EC2



AWS Fargate

イメージレジストリ

コンテナイメージの格納



Amazon ECR

その他の関連サービス

サービスメッシュ、サービスディスカバリ、モニタリング、より抽象化されたコンテナ実行基盤など



AWS App Mesh



AWS Cloud Map



Amazon CloudWatch



AWS App Runner



AWS Batch

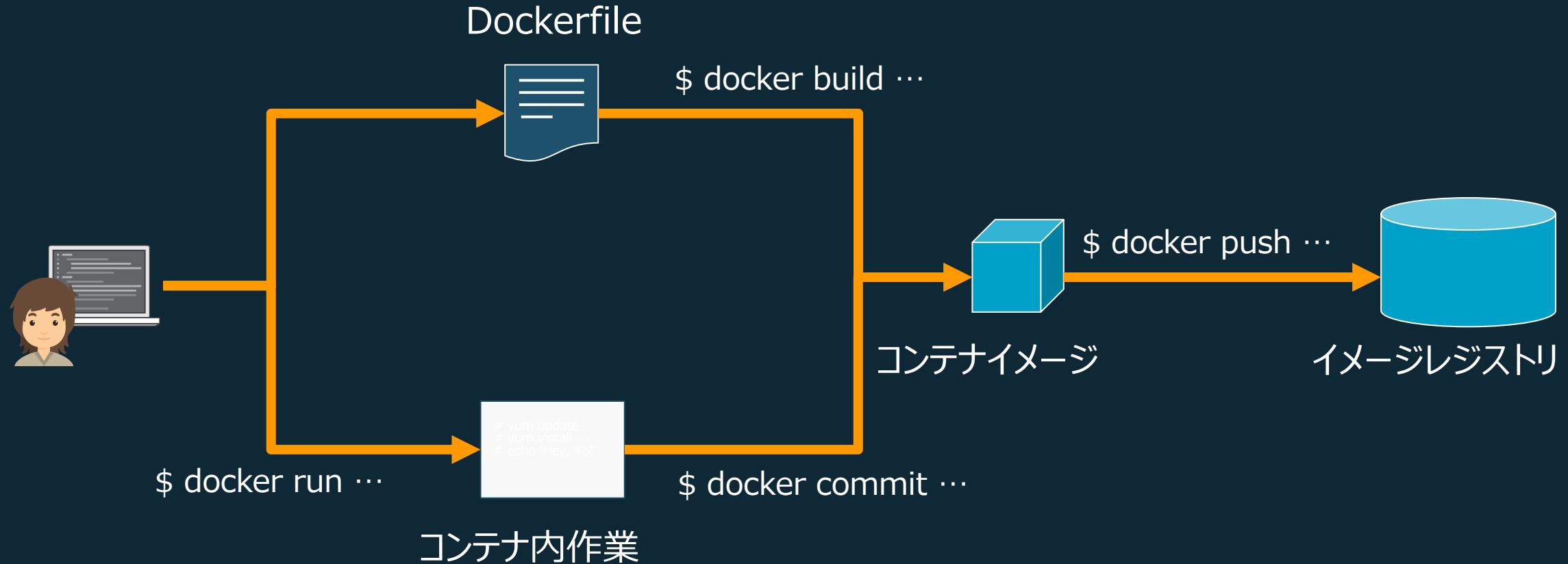


コンテナオーケストレーションとは



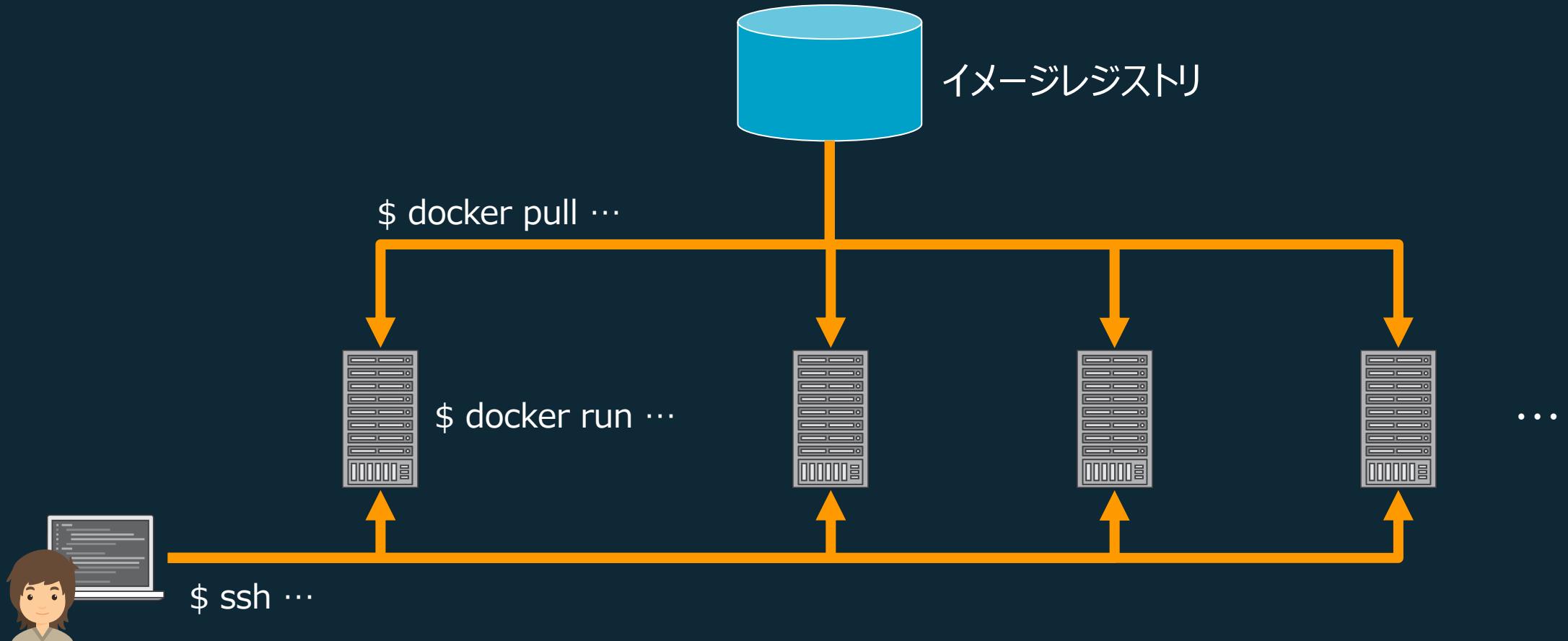
Docker を利用した基本的ワークフロー（再掲）

- コンテナイメージ作成 -

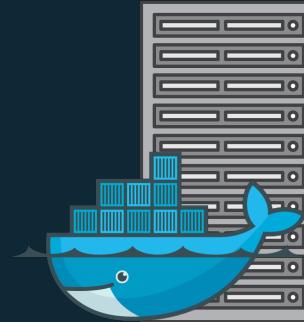


Docker を利用した基本的ワークフロー（再掲）

- コンテナ実行 -

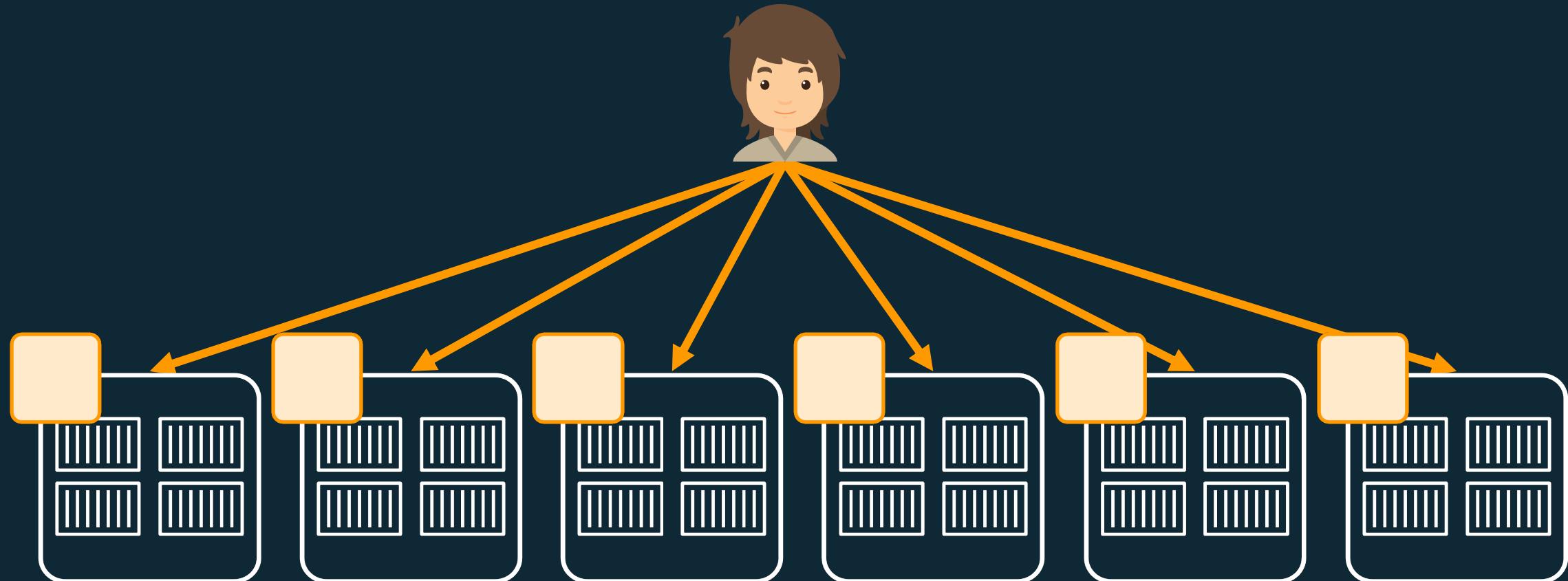


思っていたよりも手作業？



- Docker の責務は单一サーバー上のコンテナのライフサイクル管理
- 複数サーバーやコンテナを束ねた概念に対するオペレーションはスコープ外

手作業でのコンテナイメージダウンロードと実行は
非効率かつミスオペレーションを招く



「この EC2 インスタンスの
クラスタでコンテナを
実行したいです」



「このコンテナを 3 つの AZ に
分散させて 10 個デプロイして、
このロードバランサーに
つないでください」

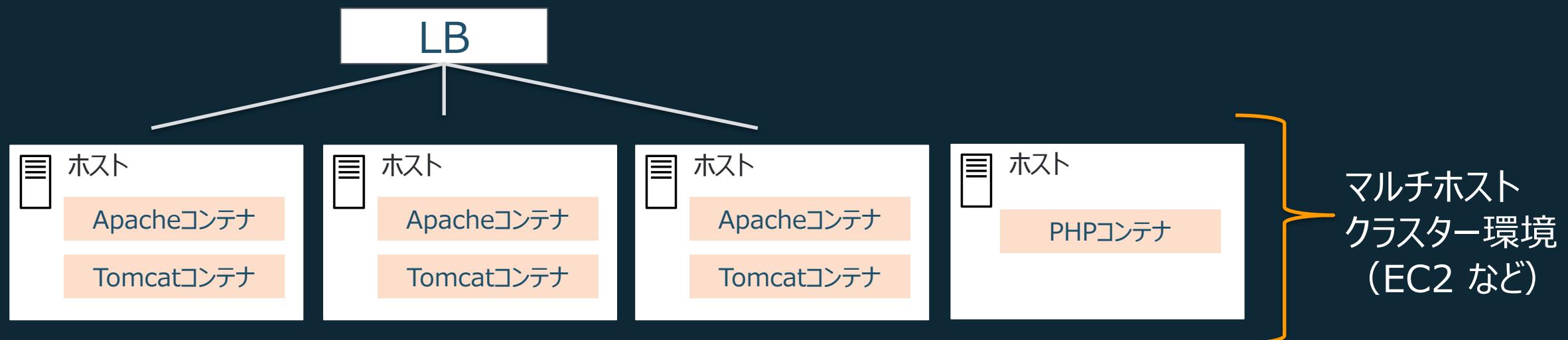


Amazon
ECS

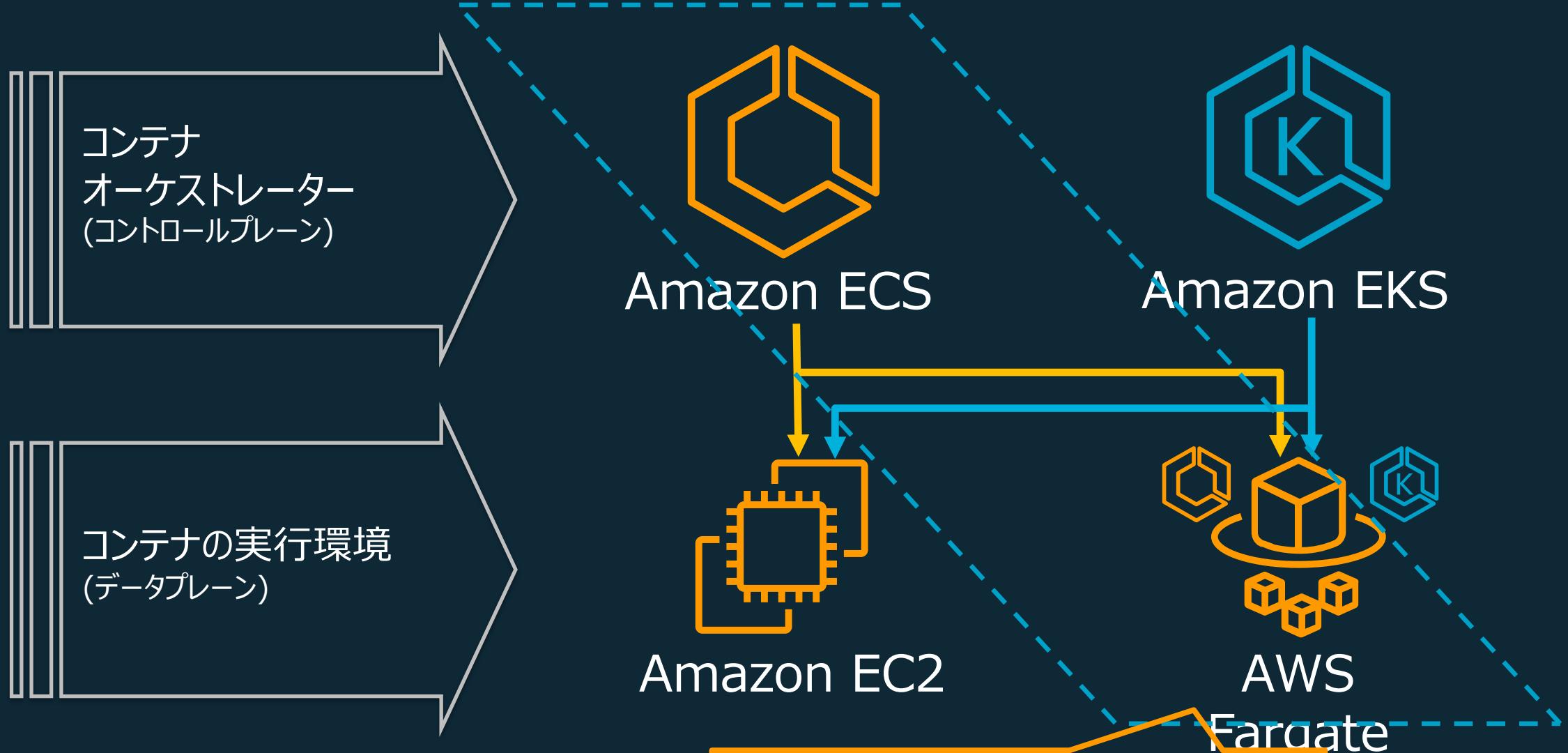


“Docker”だけでは不足していたもの

- ホストが 1 台だけであれば運用の手間はかからない
- コンテナの有用性が認められ、開発だけでなく本番環境でも利用ニーズが高まる
- 複数のホスト、コンテナを統合的に管理する仕組みが求められた



AWS 上でのコンテナ実行における選択肢



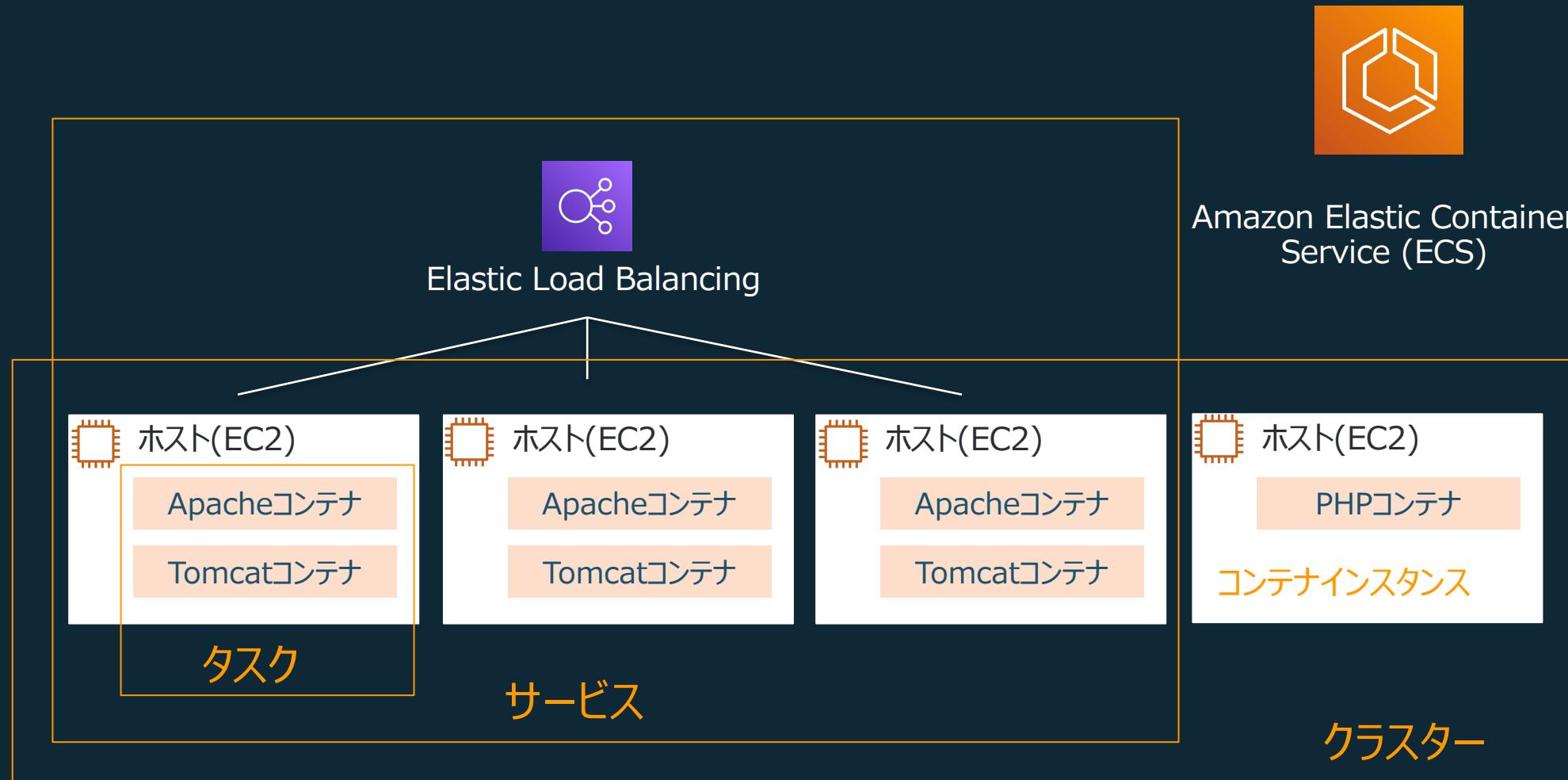
Amazon Elastic Container Service (ECS)

- Re:Invent 2014 にて発表、2015 年 4 月 GA
- フルマネージド型コンテナオーケストレーションサービス
- AWS サービスとの統合
 - Elastic Load Balancing (ELB)
 - Amazon CloudWatch
 - AWS Code シリーズ etc.
- 料金
 - EC2 起動タイプ: AWS リソース (EC2、EBS など) に対してのみ料金が発生
 - Fargate 起動タイプ: 使用した vCPU とメモリリソースの実行時間で計算



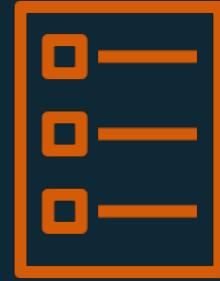
Amazon Elastic
Container Service

ECS 概要



ECS の構成要素

- タスク
 - ワークロード実行の最小単位
 - 1 つ以上のコンテナで構成される
 - タスク定義にもとづいて起動される
 - タスク定義では以下などのパラメータを指定
 - タスク内の各コンテナで使用する Docker イメージ
 - タスク内の各コンテナで使用する CPU とメモリの量
 - ネットワークモード
 - タスクに割り当てる IAM ロール
 - コンテナの環境変数
 - ログ設定 etc.



Task

ECS の構成要素

- サービス
 - 指定した数のタスクを実行し、維持する
 - タスクが何らかの理由で停止した場合、新しいタスクを起動し、サービスで必要なタスク数を維持
 - ロードランサーの背後でサービスを実行し、タスク間でトラフィックを分散させる



Service

ECS の構成要素

- クラスター
 - タスクを実行する実行環境の境界
 - タスクやサービスを起動する際にはクラスターを指定する
 - EC2 でタスクを実行する場合、実行環境となる EC2 インスタンス (ECS コンテナインスタンス) のグループがクラスター
 - Fargate の場合は単なる論理的な境界

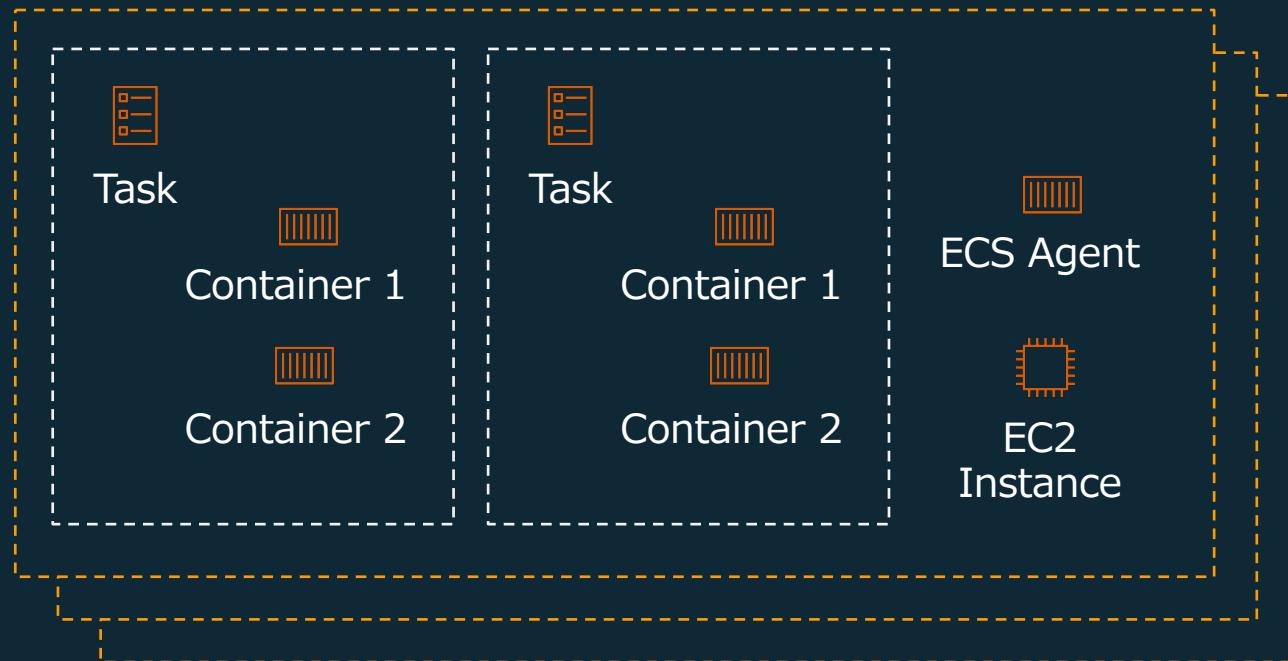
※ ECS コンテナインスタンス

ECS クラスターに登録されている EC2 インスタンス

ECS コンテナエージェントを実行してコントロールプレーン (ECS) と通信し、指示を受けてタスクを起動する

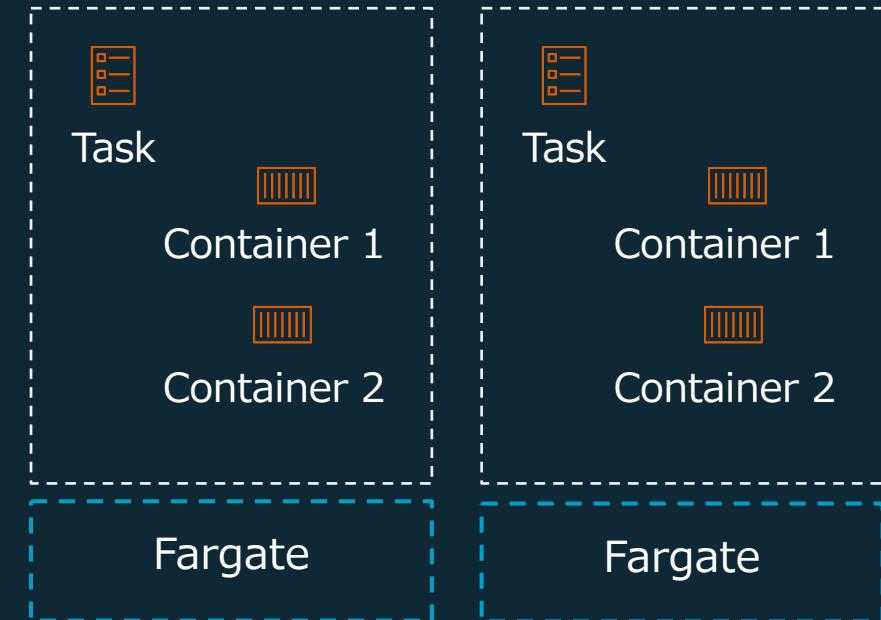
Amazon ECS 起動タイプの選択

EC2 起動タイプ



コンテインインスタンス (EC2) が必要
ホスト OS のバッチ適用など、管理はお客様責任範囲

Fargate 起動タイプ



インスタンス (EC2) は不要
インフラストラクチャ管理は AWS

AWS Fargate



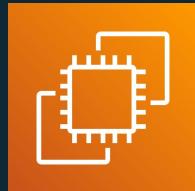
AWS Fargate

- コンテナ向けサーバーレスコンピューティングエンジン
- コンテナを実行するサーバーのプロビジョンと管理が不要
- コンテナが実行されるインフラストラクチャは必要なパッチが適用され常に最新の状態に保たれる
- タスクごとにリソースを指定し、使用時間分だけ料金が発生
- EC2 の様に直接インフラストラクチャにアクセス (SSH ログインなど) することはできない
- Fargate で実行されるタスクには独立した Kernel が割り当てられる

Amazon ECS におけるコンテナ実行環境

ECS on EC2

(コンテナを仮想サーバー上で動作)



アプリケーションコンテナ

ホストのスケーリング

コンテナエージェント設定

ホスト OS / ライブラリ設定

お客様が管理するレイヤー

ECS on Fargate

(コンテナをサーバーレスで動作)



アプリケーションコンテナ

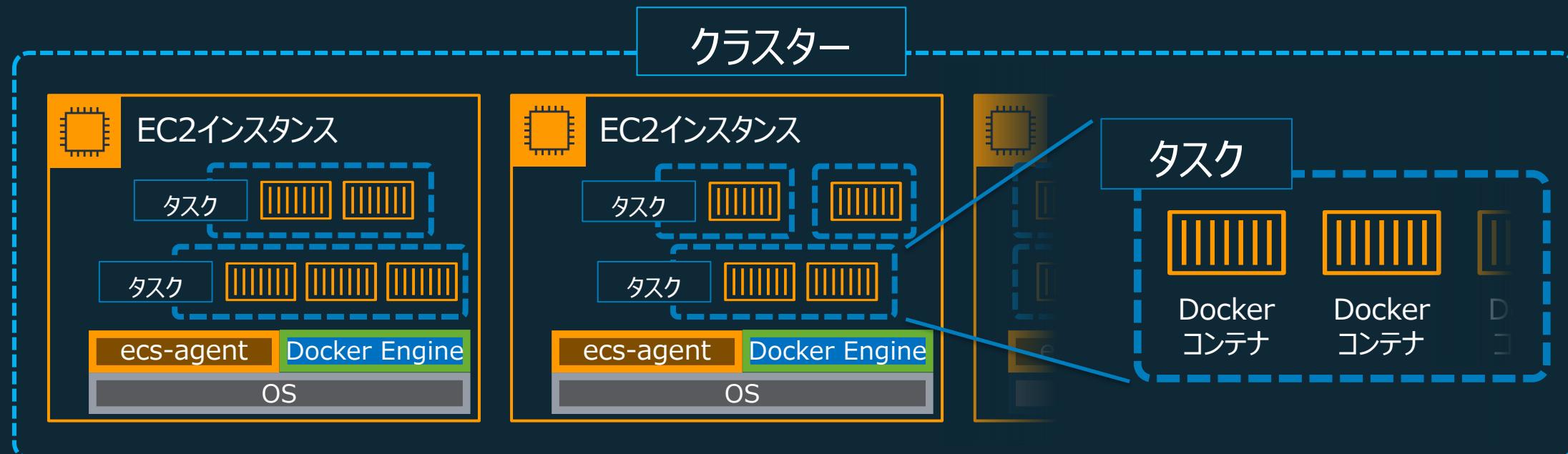
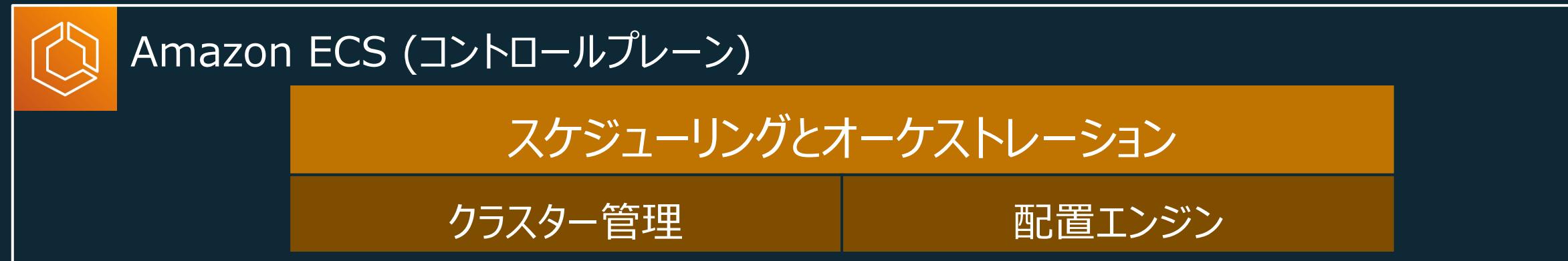
ホストのスケーリング

コンテナエージェント設定

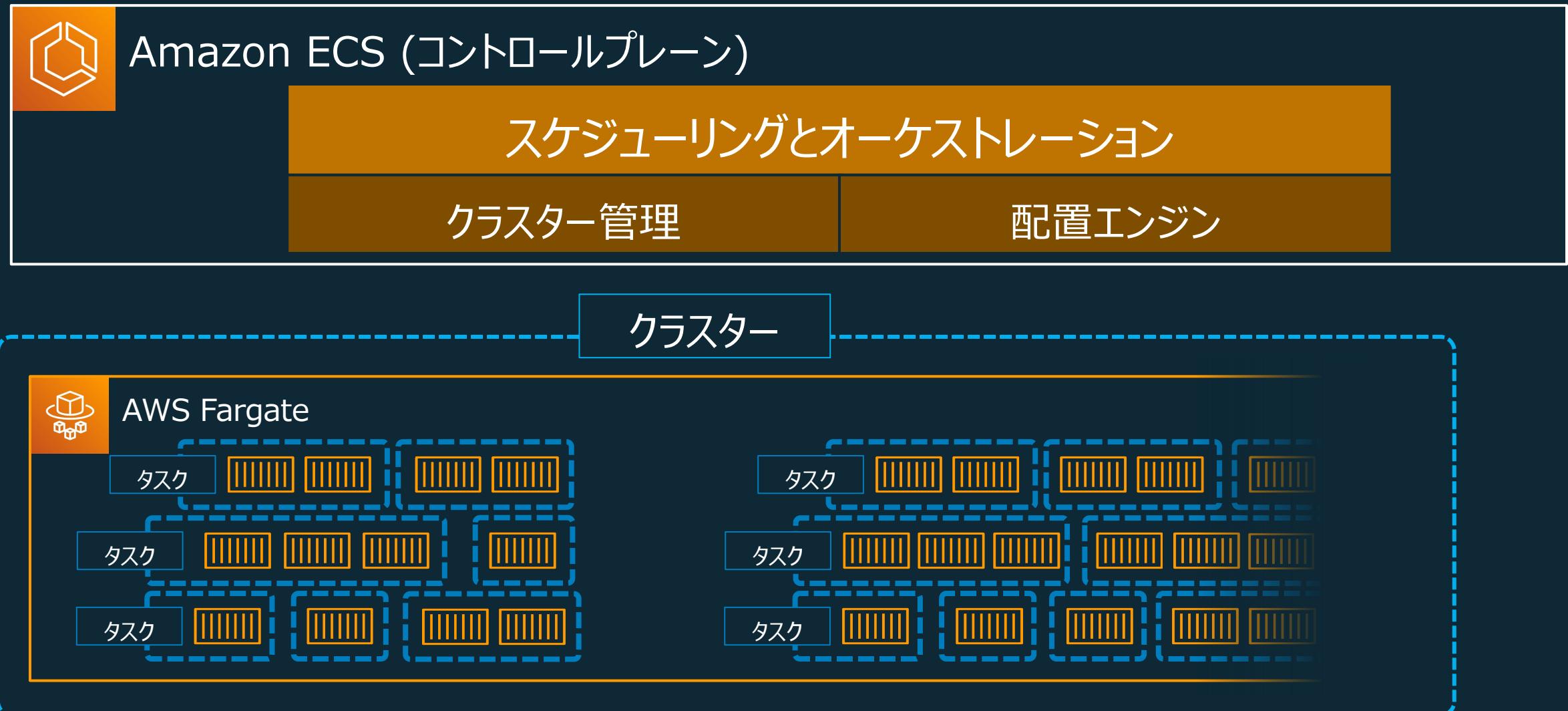
ホスト OS / ライブラリ設定

AWSが提供するレイヤー

Amazon ECS の動作イメージ (on EC2)



Amazon ECS の動作イメージ (on Fargate)



Amazon Elastic Container Registry (ECR)



Amazon Elastic
Container Registry

- 完全マネージド型のコンテナレジストリ
- コンテナイメージを簡単に保存、管理、共有、デプロイ
- 高い高可用性と耐久性により、自前のコンテナリポジトリの運用や、基盤となるインフラストラクチャのスケーリングの検討は不要に
- 組織内でプライベートに共有することも、世界に向けてパブリックに共有することも可能
- リポジトリに保存するデータと転送するデータ量に対してのみ課金

AWS のコンテナ関連サービス（再掲）

オーケストレーション

コンテナのデプロイ、スケジューリング、スケーリング、クラスター管理



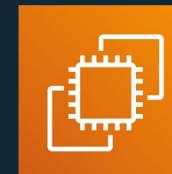
Amazon ECS



Amazon EKS

コンテナ実行基盤

コンテナの実行



Amazon EC2



AWS Fargate

イメージレジストリ

コンテナイメージの格納



Amazon ECR

その他の関連サービス

サービスメッシュ、サービスディスカバリ、モニタリング、より抽象化されたコンテナ実行基盤など



AWS App Mesh



AWS Cloud Map



Amazon CloudWatch



AWS App Runner



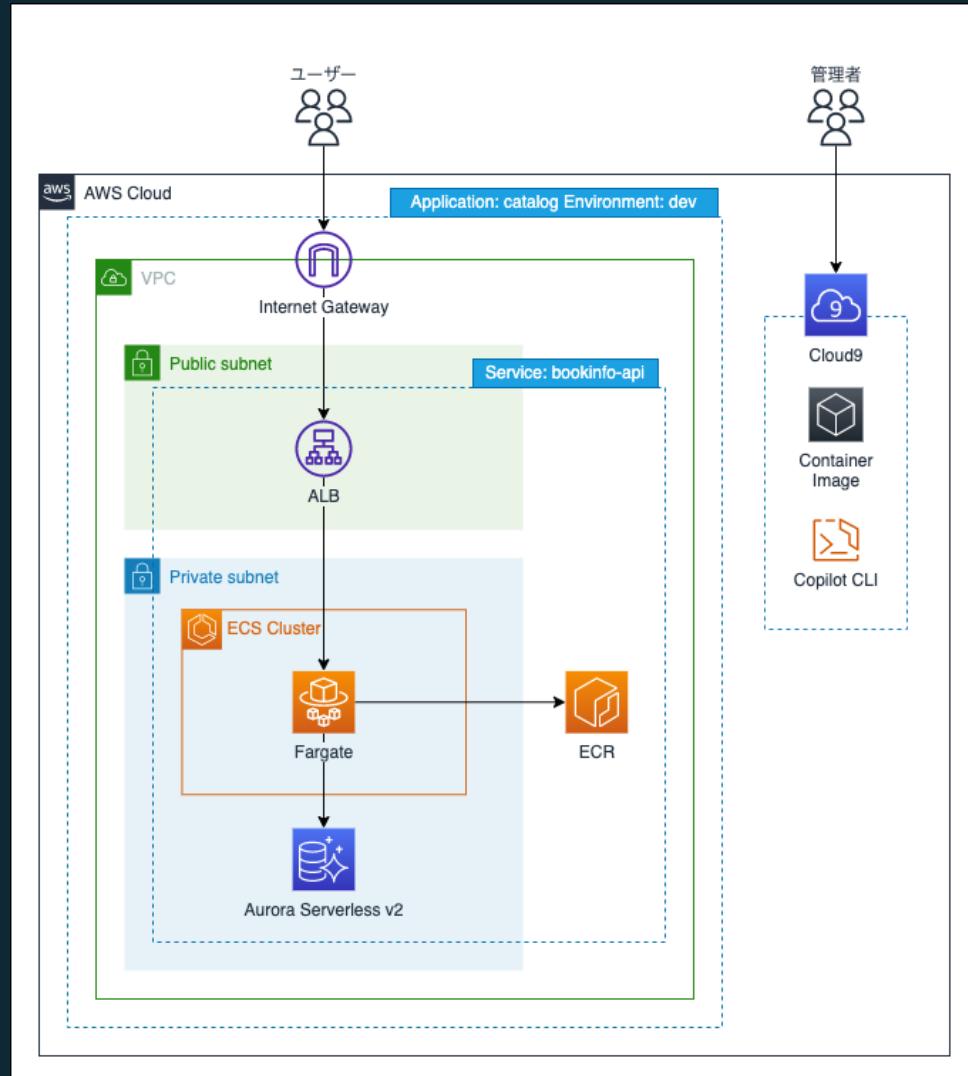
AWS Batch

A photograph of a person's hand raised in the air, palm facing forward, against a blurred background of colorful stage lights. The background is a solid teal color.

Q&A

ハンズオンで利用するAWSサービス

ハンズオンで構築するAWS構成



AWSサービス

- AWS Fargate / Amazon ECS
- Amazon ECR
- Amazon VPC
- ALB (Application Load Balancer)
- Amazon Aurora Serverless v2
- AWS Cloud9
- AWS Copilot CLI

Amazon VPC

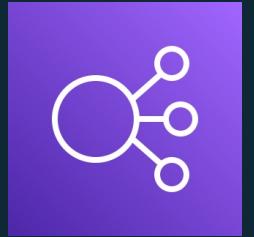
Amazon Virtual Private Cloud の略で、AWS 上に構築できる仮想ネットワーク環境です

インターネットに（から）接続できる Public Subnet、インターネットと分離されている Private Subnet を使い分けることで、従来の DMZ と Firewall の内側のような構成が簡単に構築できます



Elastic Load Balancing (ELB)

AWSクラウド上のロードバランシングサービス



ELBで実現できるシステム

- **スケーラブル**：複数のEC2インスタンス/ECSコンテナ..etc（ターゲット）に負荷分散
- **高い可用性**：複数のアベイラビリティゾーンにある複数のターゲットの中から正常なターゲットにのみ振り分け

ELB自体の特徴

- **スケーラブル**：ELB自体も負荷に応じてキャパシティを自動増減
- **安価な従量課金**：従量課金で利用可能
- **運用管理が楽**：マネージドサービスなので管理が不要
- **豊富な連携機能**：Auto Scaling, Route 53, Cloud Formation... などと連携

Elastic Load Balancing (ELB) の種類



Application Load Balancer
(ALB)



Network Load Balancer
(NLB)



Classic Load Balancer
(CLB)

HTTP, HTTPS, HTTP/2

TCP, UDP, TLS

HTTP, HTTPS, TCP

VPC

VPC

EC2-Classic, VPC

L7 のコンテンツベース
のロードバランサー

L4機能を提供するロード
バランサー

EC2-Classic ネットワー
ク用ロードバランサー

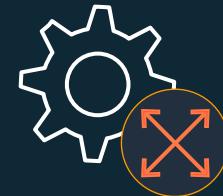
Amazon Aurora

クラウド向けに構築されたMySQLとPostgreSQL互換のリレーショナル・データベース
商用データベースの性能と可用性を1/10のコストで実現



パフォーマンスとスケーラビリティ

標準的なMySQL, PostgreSQLに比べ
パフォーマンス効率が良い
15個のリードレプリカ



可用性と耐久性

フォールトトレラント自己回復
ストレージ
3つのAZで6つのデータコピー
グローバルデータベースと
クロスリージョンレプリケーション



高いセキュリティ

ネットワークの分離
保存時/転送時の暗号化



フルマネージド

RDSによる管理：
ハードウェアのプロビジョニング、
ソフトウェアのパッチ適用、
セットアップ、設定、
バックアップは不要

The fastest growing service in the history of AWS

Amazon Aurora Serverless v2



オンデマンドで自動的にスケール

アプリケーションのニーズに応じて自動的に容量を拡張

秒単位のシンプルな**従量課金**

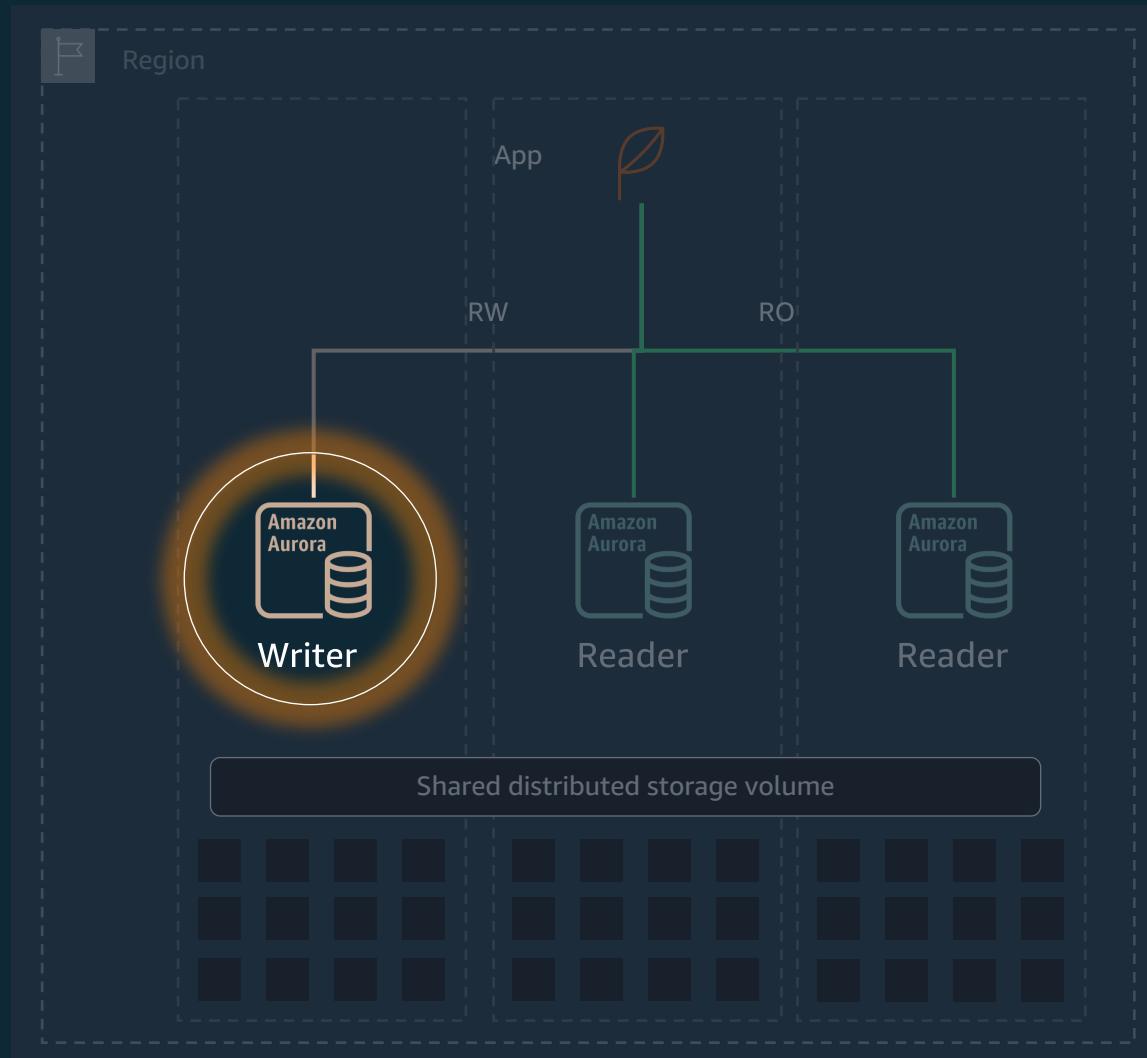
v2は柔軟に拡張し、要求の厳しいアプリケーションをサポート

データベースの容量管理の心配からの解放

Amazon Aurora Serverless v2 スケーリング (1)

インスタンスのスケールに関して

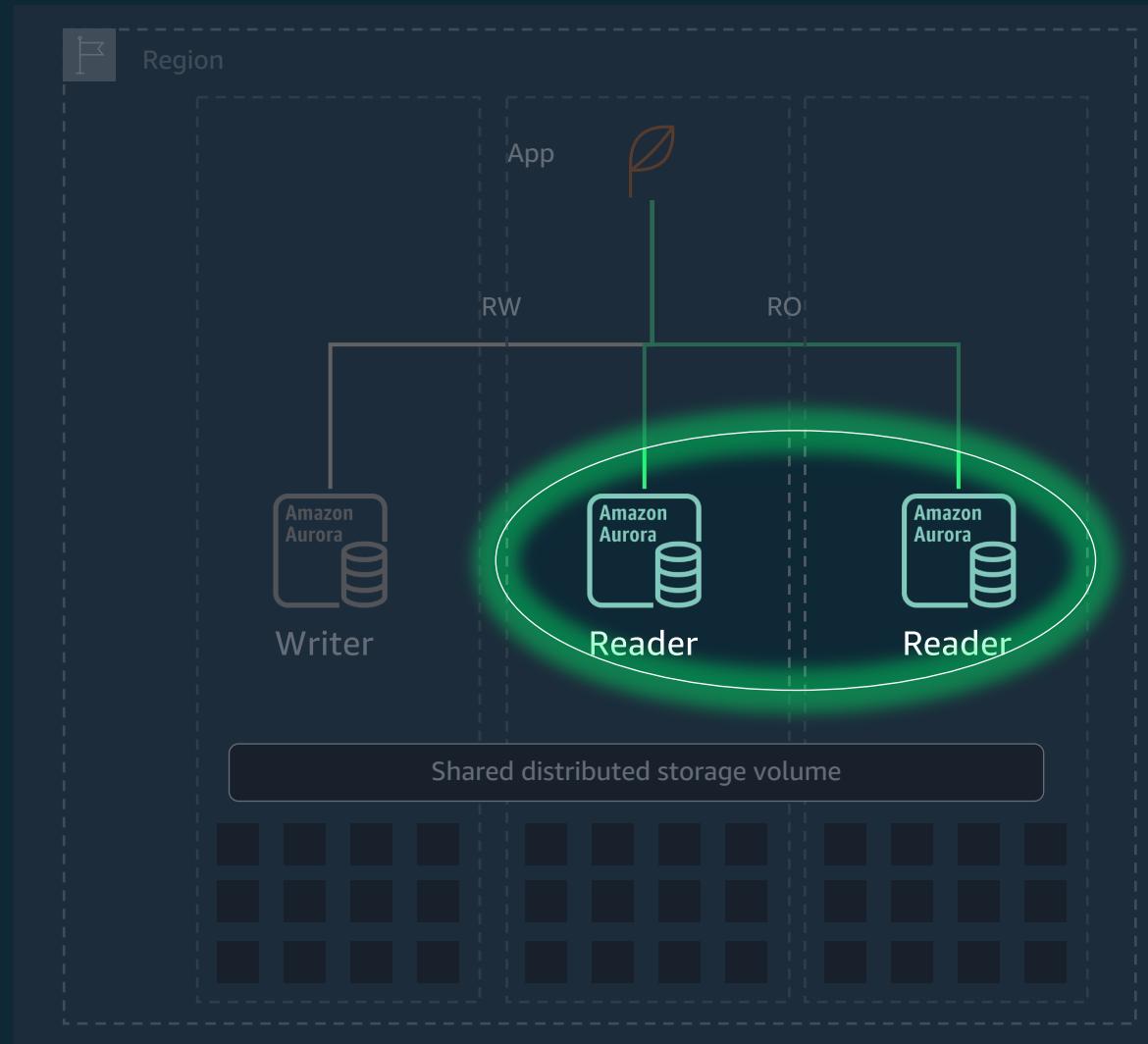
- CPU/Memory/Networkの使用量などからシームレスにスケールしセッションの状態、バッファープールの内容も維持される
- v2ではv1と異なりScaling Pointを取ることなしに必要な時に即座にスケール可能
- v2ではv1と異なり0.5 ACUの粒度で最大128 ACUまでスケールアップ可能(v1は $1 > 2 > 4 > 8$ のように2倍(2のべき乗)のキャパシティで増加し最大256 ACU)
- スケールアップと異なりスケールダウンは、ある程度を時間をかける(ただしv1と比べて最大15倍高速(<1分))



Amazon Aurora Serverless v2 スケーリング (2)

Readerによるスケールについて

- ReaderはProvisionedと同様に最大15インスタンスまで設定可能
- 各Readerは最大128 ACUまで自動でスケールアップ可能
- * Provisioned Clusterで提供されているReaderのAuto Scaling機能(負荷状況に応じたReaderの自動追加)はAurora Serverless v2ではサポートされていません



AWS Cloud9



ブラウザのみでコードを記述、実行、デバッグできるクラウドベースの統合開発環境(IDE)



コードエディタ、デバッガー、ターミナルが含まれている

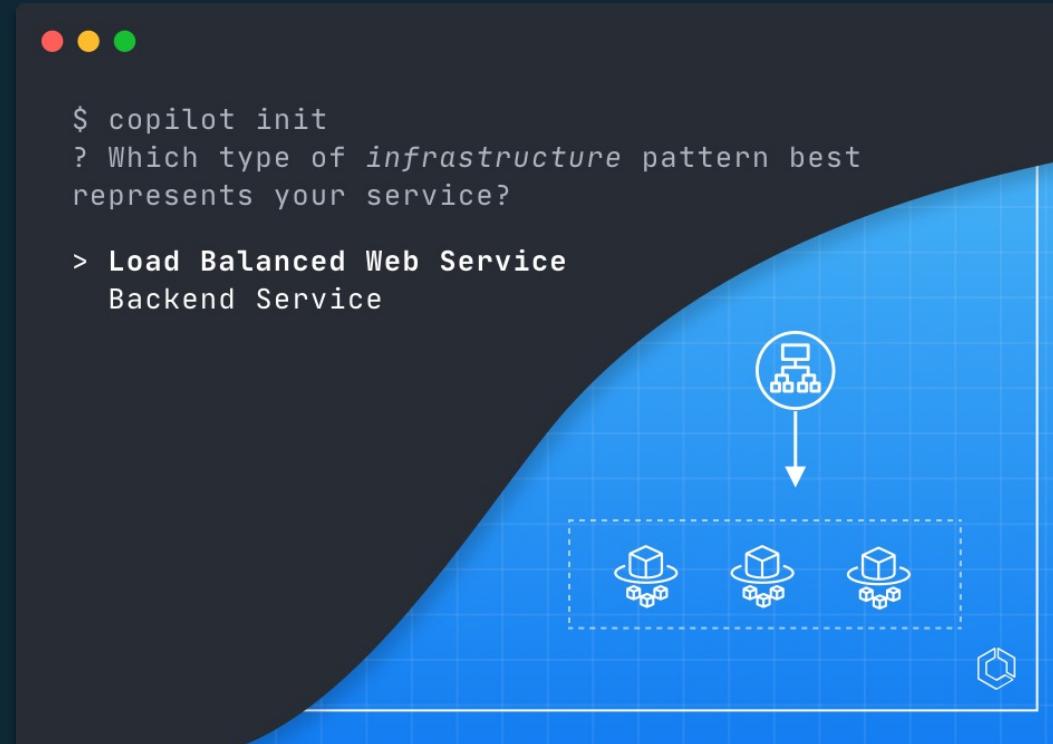


JavaScript、Python、PHPなどの一般的なプログラム言語に不可欠なツールがあらかじめパッケージ化されているため、新しいプロジェクトを開始するためにファイルをインストールしたり、開発マシンを設定したりする必要がない

AWS Copilot CLI



プロダクションレディなコンテナアプリケーションのビルド・リリース・運用のための開発向けツール



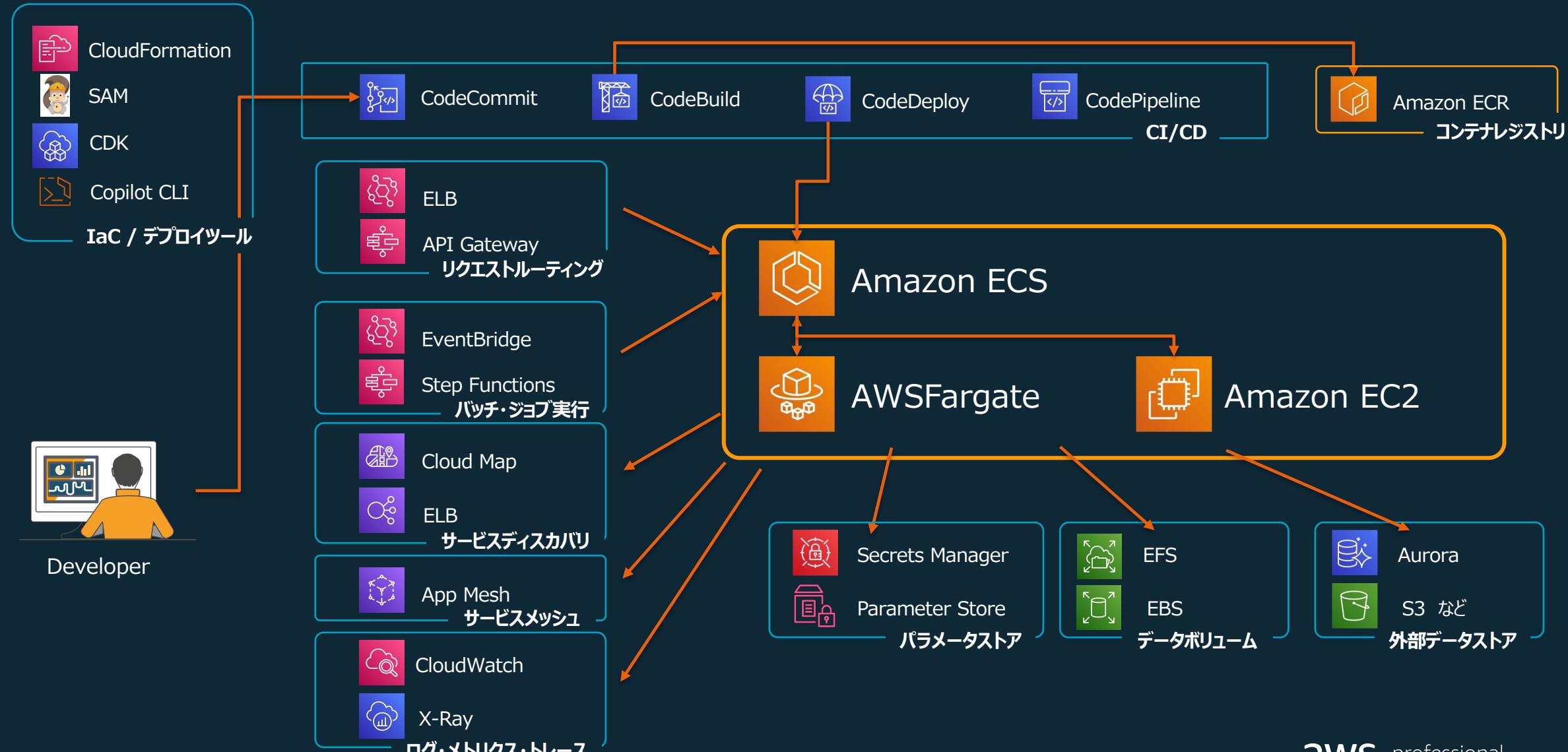
- AWSが開発したOSSのCLIツール
 - <https://github.com/aws/copilot-cli>
 - 最新バージョンは v1.25.0
- 開発者がAmazon ECS上で簡単にビルド・リリース・運用をできるようにするツール
- インフラストラクチャーよりもアプリ開発にフォーカス
 - ECSクラスターやCI/CDパイプラインを対話式にプロビジョン稼働
- 宣言的フォーマットでのサービスの定義

Appendix

ECS利用時のAWSサービス構成



ビルディングブロックの要素 – ECS関連



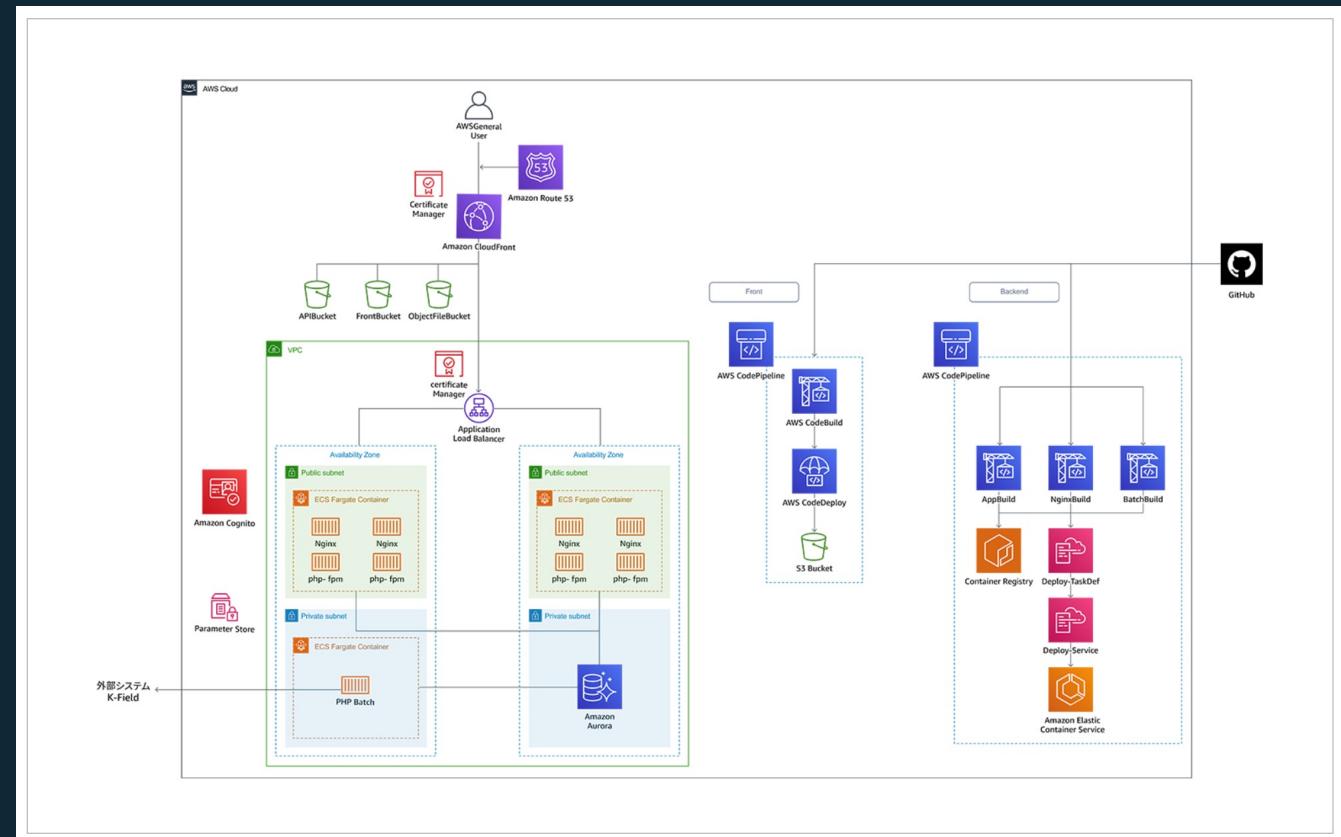
ローリングアップデート

他社事例

鹿島建設株式会社様

- 建設現場の全体像把握に向けて 3 次元の現場管理システムを検討
- スケーラブルなアーキテクチャを採用し 6 ヶ月の短期間で『3D K-Field』をリリース
- 遠隔監視により現場管理が効率化
- 労働時間の短縮と働き方改革に貢献

「AWSを採用した理由は、当社で最もノウハウがあり、公開情報も豊富なことです。加えて可用性の高さ、メンテナンスのしやすさ、機能やサービスのアップデートの早さも選定の決め手となりました。アプリケーションは、コンテナを使ったマイクロサービスアーキテクチャで構成し、コンテナの稼働基盤と運用基盤は管理が容易な AWS Fargate と Amazon Elastic Container Service(Amazon ECS)で構築しています」

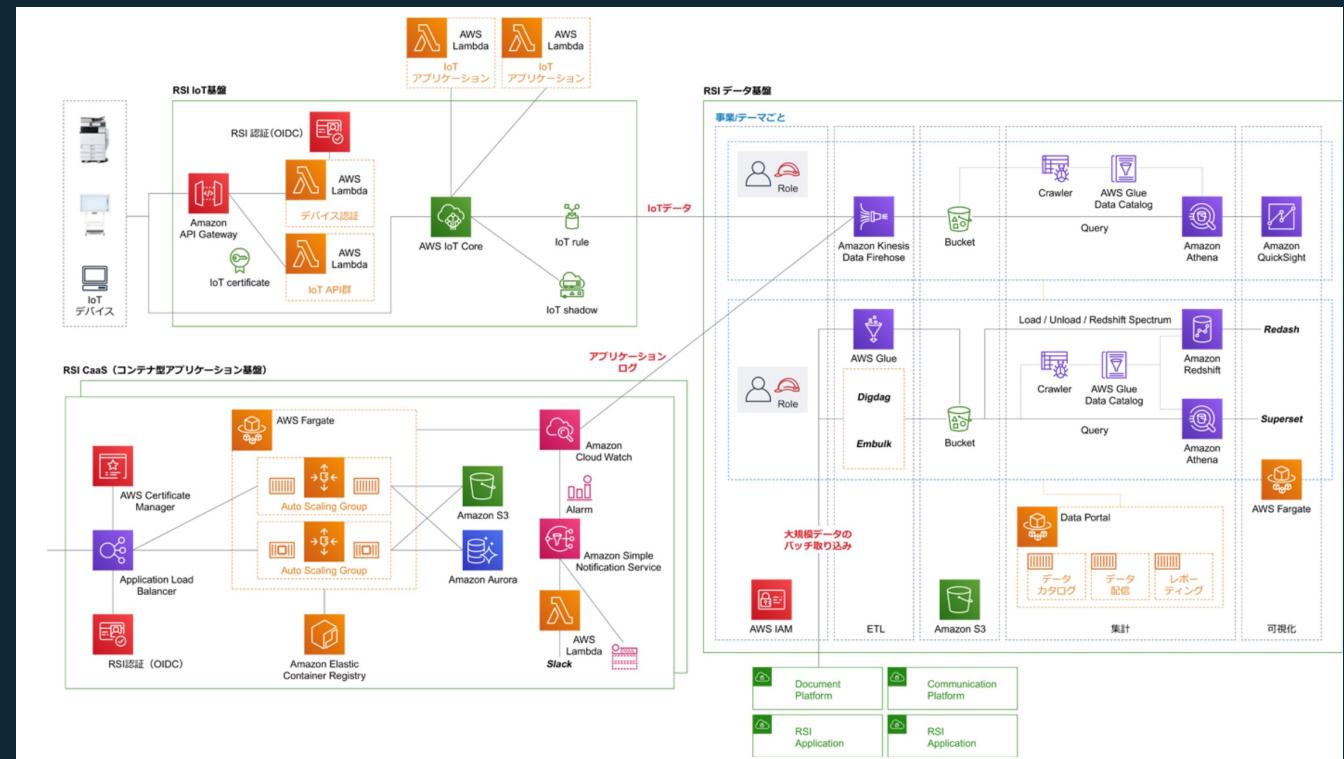


https://aws.amazon.com/jp/solutions/case-studies/kajima-asiaquest/?did=cr_card&trk=cr_card

株式会社リコー様

- 複合機を中心とするビジネスから人の働き方変革を支援するビジネスに転換
- 事業部のシステムとの親和性と豊富なマネージドサービスを評価
- データ収集の基盤を構築してコンテナとIoTの基盤を接続

コンテナ基盤では、AWS Fargateを中心に事業部の開発者や利用者が簡単にアプリケーションをプロイしてビジネスを始められる環境を提供し、データ活用基盤に接続することでデータを自動収集できる仕組みを整備しました。プラットフォーム事業本部 RSI 開発センター PF 開発室 インフラグループの神田博之氏は「独自に構築すると半年かかるところが、AWS をベースとしたコンテナ基盤を活用することでクラウド環境を即日払い出すことが可能になり、すぐに開発に着手することができます」と語ります。

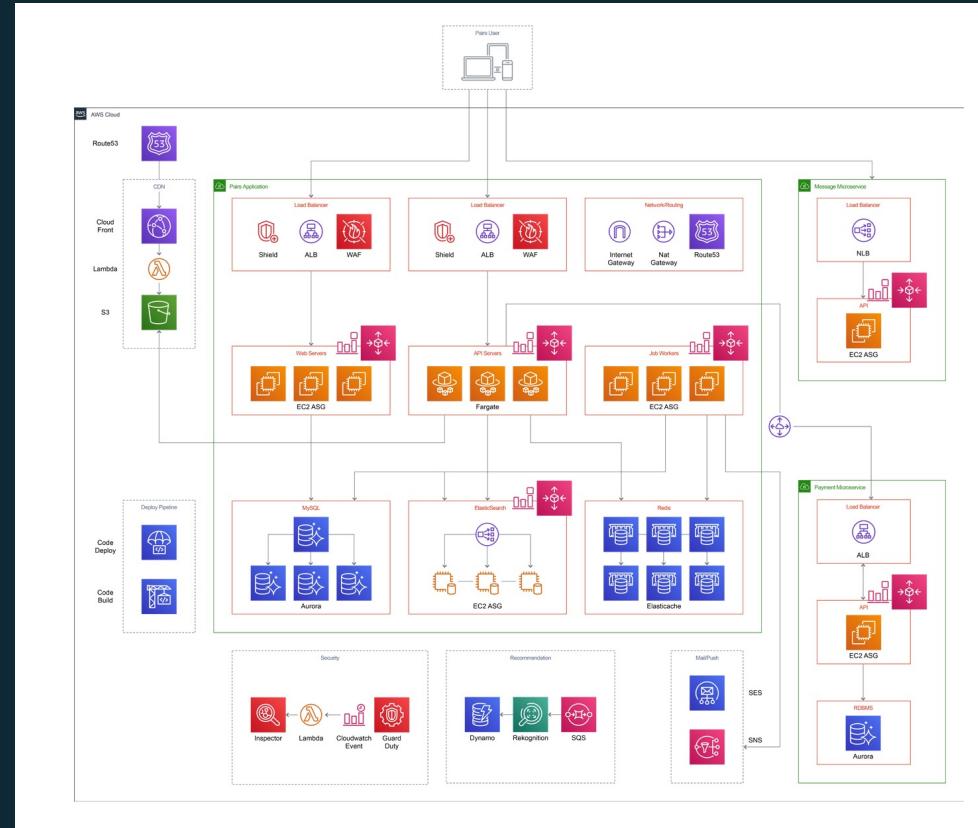


https://aws.amazon.com/jp/solutions/case-studies/ricoh/?did=cr_card&trk=cr_card

株式会社エウレカ様

- ・ 日本、台湾、韓国でトップクラスのマッチングサービスをアプリで提供
 - ・ サービスのコアとなる最も重要なセキュリティを AWS の機能で実現
 - ・ コンテナ化による開発効率化とレコメンドの精度向上で満足度をアップ

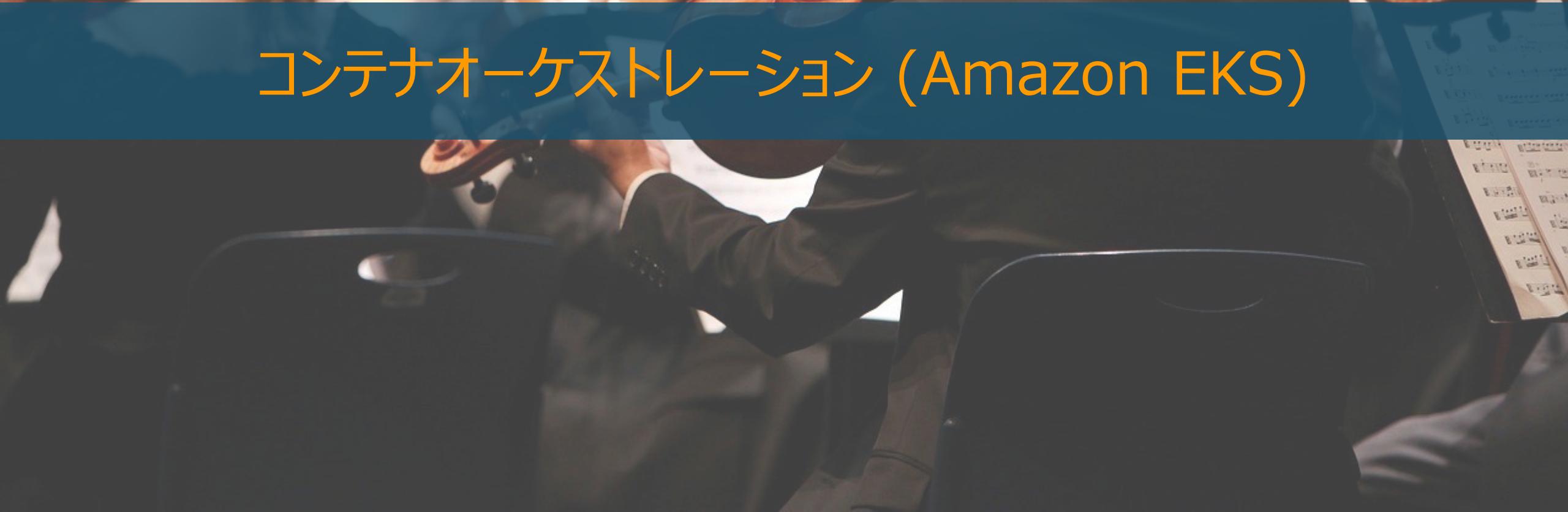
「ユーザー数が増え、広告や露出が増えていくにつれてトラフィックの増減が予測しづらくなる一方、新たな機能の追加や機能のリッチ化も求められます。その中でも開発速度を落としたくないため、変更を迅速に適用できるコンテナ技術を使おうと考え、Amazon ECS と AWS Fargate を活用しています。コンテナ化によって機能別に小さく作ることが可能となり、ビルドやデプロイを迅速に行うことができるようになりました。」



https://aws.amazon.com/jp/solutions/case-studies/eureka/?did=cr_card&trk=cr_card



コンテナオーケストレーション (Amazon EKS)



Kubernetes (K8s) とは

- ・**宣言的な構成管理と自動化**を促進し、コンテナ化されたワークロードやサービスを管理するための、**ポータブルで拡張性**のあるオープンソースプラットホーム
- ・**膨大で、急速に成長しているエコシステム**を備えており、それらのサービス、サポート、ツールは幅広い形で利用可能
- ・Google が 2014 年に Kubernetes プロジェクトをオープンソース化
- ・Google が大規模な本番ワークロードを動かしてきた 10 年半の経験と、コミュニティから得られた最善のアイデア、知見に基づく



<https://kubernetes.io/ja/docs/concepts/overview/what-is-kubernetes/>

Amazon EKS

運用難易度の高い Kubernetes マスターをマネージドで提供

Tenet 1

EKS はエンタープライズ企業が本番のワークロードを実行するためのプラットフォームであること

Tenet 3

EKS ユーザが他の AWS サービスを使う時、シームレスな連携を実現し不要な作業を取り除くこと

Tenet 2

EKS はネイティブで最新のKubernetes の体験を提供すること

Tenet 4

EKS チームは積極的に Kubernetes プロジェクトに貢献していくこと



ECS or EKS



Amazon ECS

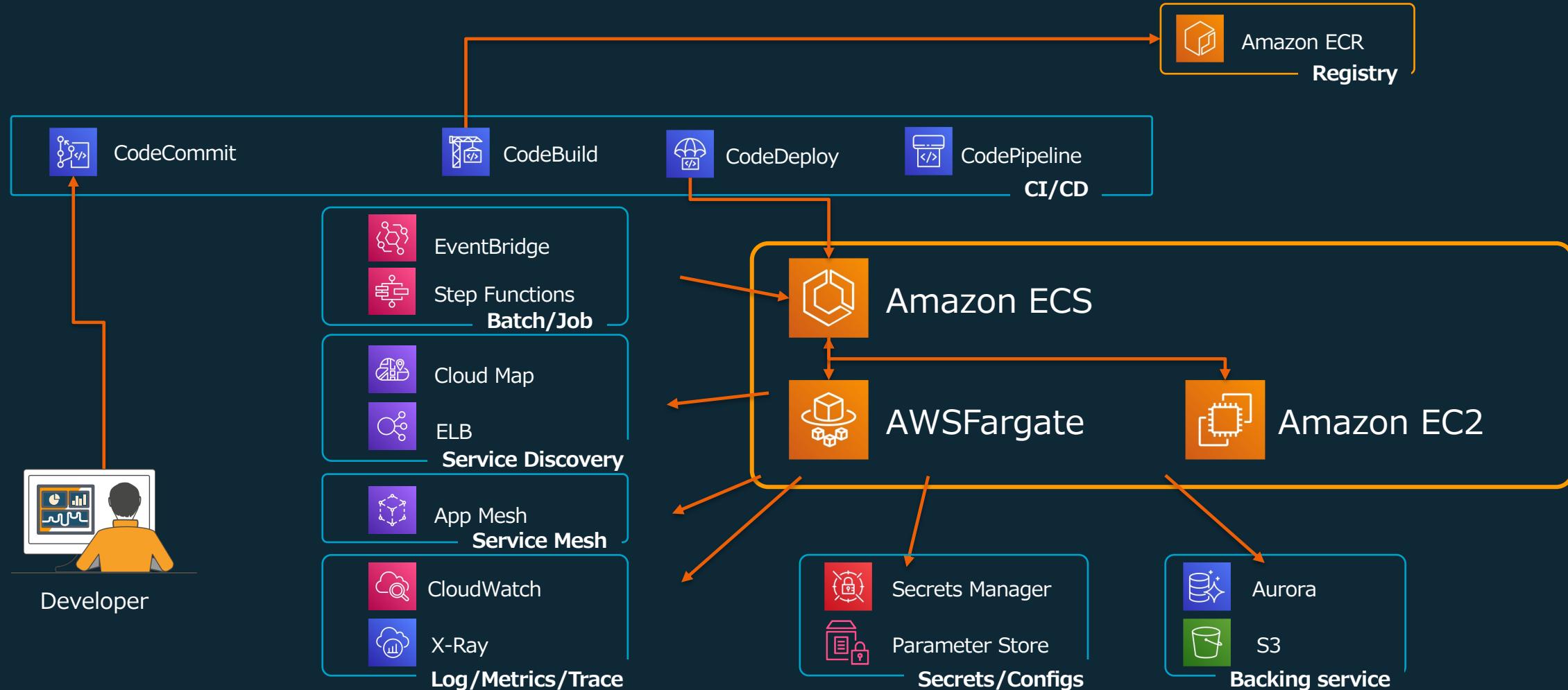
- Amazon ECS はコンテナオーケストレーションを行うサービス
- AWS 上でシステムを構築する際に、コンテナで動かす範囲を管理・運用するものであり、AWS というプラットフォーム上で利用される**ビルディングブロック**の 1 つ
- **高い後方互換性**による運用の容易性
- 多様なワークフローをサポートする「タスク」と「サービス」というシンプルなリソース表現



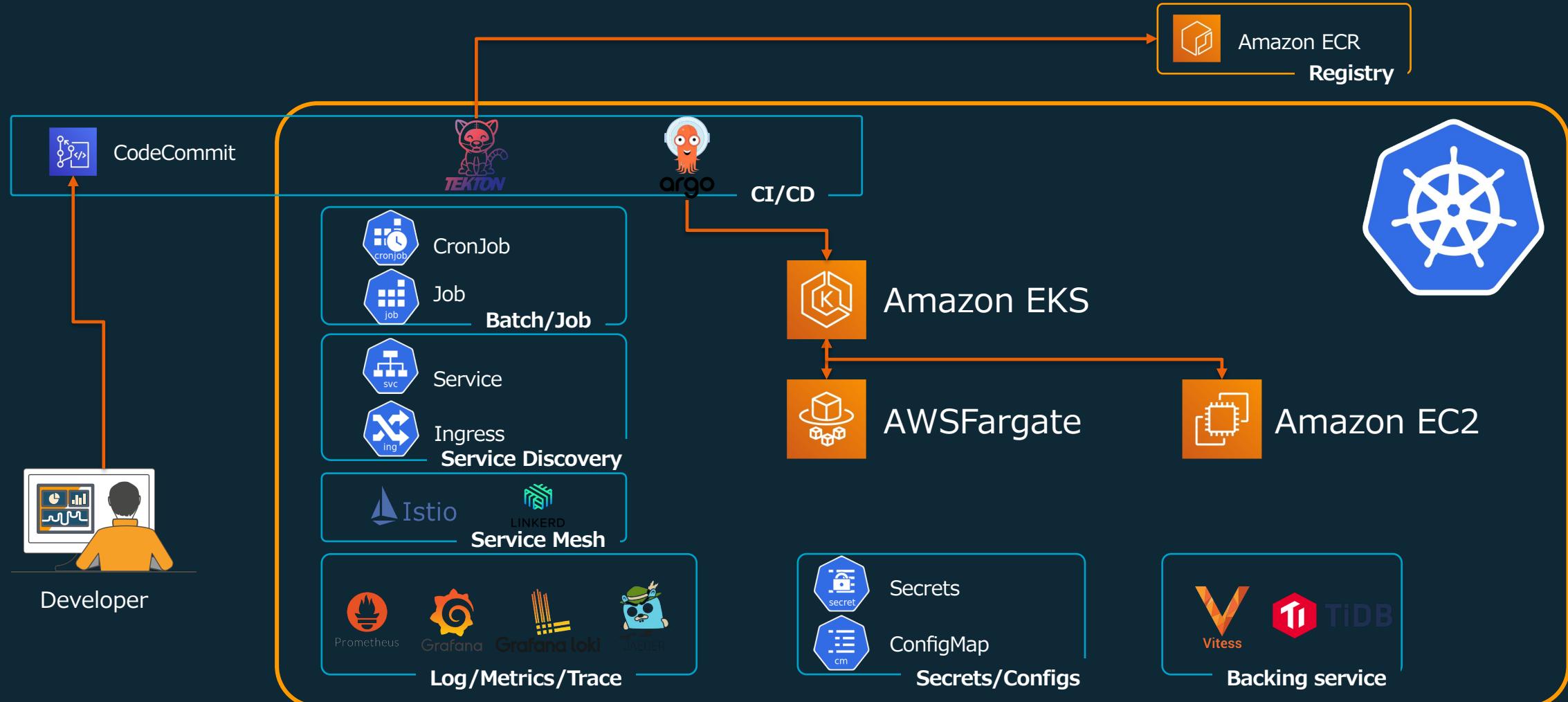
Amazon EKS

- Amazon EKS はコンテナオーケストレーターである Kubernetes をプロビジョニングし、マネージドで提供するサービス
- K8s はオンプレミスでも動作するプロダクトであり、システムを構築するために必要な「すべて」を K8s 上で実現することを目指している) **プラットフォーム**
- K8s はオープンソースで活発な開発がされている
- **K8s エコシステムの OSS ツール**をそのまま動かせる
- Pod, Deployment, Service, CronJob などのリソースに代表される高い表現力と、カスタムリソースによる高い拡張性

ECS: ビルディングブロックの 1 つ



Kubernetes: プラットフォーム





Thank you!

後日アンケートを送付します。運営改善のため回答にご協力お願いします。