AWS Builders Online Series

T5-3

アプリケーションリファクタリング FOR コンテナ

黄 光川

アマゾン ウェブ サービス ジャパン合同会社
技術統括本部 インターネットメディアソリューション本部 ソリューションアーキテクト

aws

本セッションについて

セッションの対象者

- 既存のアプリケーションをどのようにコンテナにすればいいのか、その方法と手順のイメージが沸いていないアプリケーション開発者の方
- ローカル環境で Docker コンテナを利用した経験はあるが、実際のシステムへの適用イメージが沸いていないアプリケーション開発者の方

セッションのゴール

- 既存のアプリケーションをコンテナにする為の方法と手順を理解できる様になる
- ローカル環境と AWS 環境でのコンテナの利用・開発について理解できる様になる
- プロダクション環境への適用に向けて小さな PoC が実施できる様になる

注：本セッションでのローカル環境とは、Laptop の様な開発環境一式を揃えた端末の事とする

自己紹介

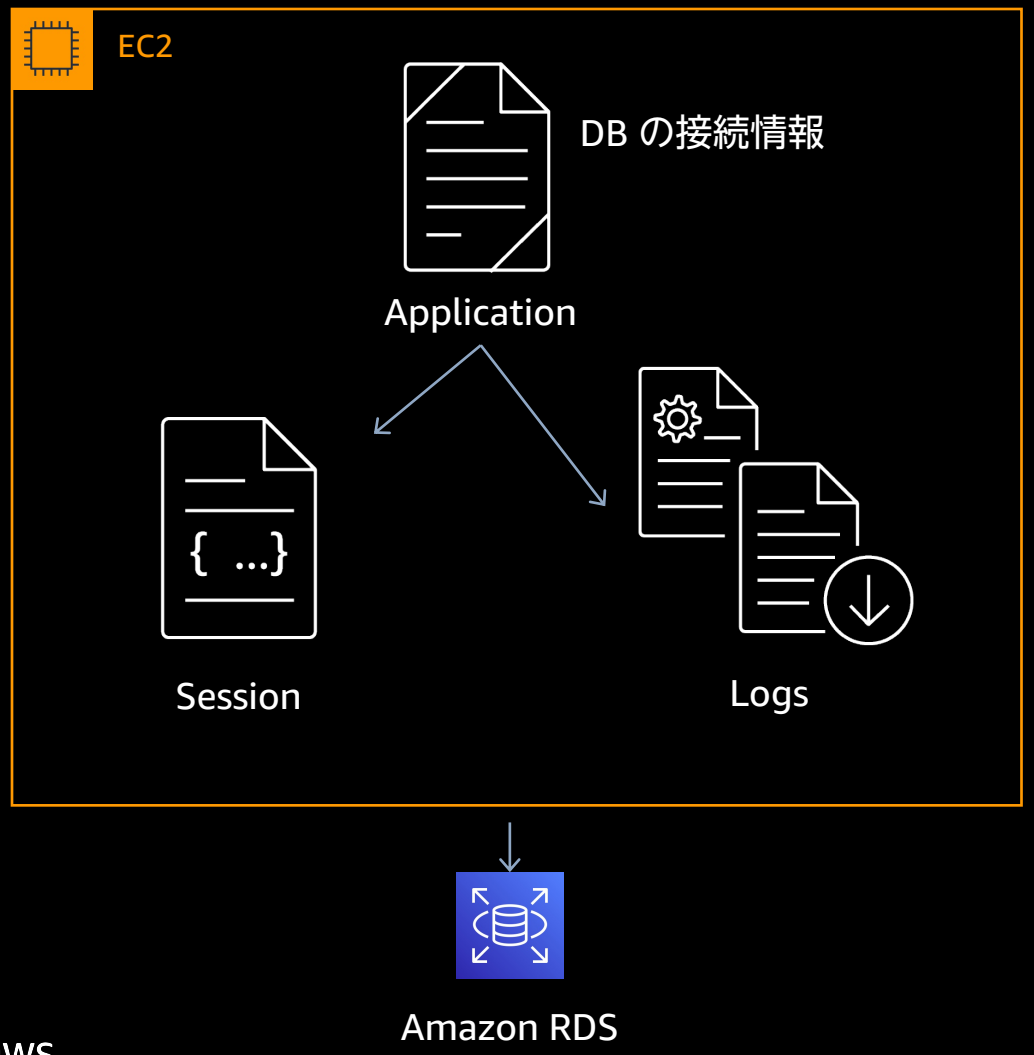
- 黄　光川 （コウ　コウセン）
- アマゾン ウェブ サービスジャパン 合同会社
  ソリューション アーキテクト
- 経歴：
  複数の業界でシステム開発、インフラエンジニア、SRE などを経験し 2020 年に AWS へ
- 趣味：
  シュノケーリング、サッカー、グルメ、映画鑑賞など
- 好きな AWS サービス：
  Amazon ECS, Amazon EKS, AWS Fargate
  などのコンテナ系のサービス

アジェンダ

- アプリケーションとステート (状態) とコンテナ
- アプリケーションのリファクタリング
- コンテナ化
- Amazon ECS へのデプロイ
- ローカル環境でのコンテナ開発
- さらなるチャレンジと学習
- まとめ

アプリケーションと
ステート (状態) とコンテナ

既存アプリケーション



1) セッション情報は、アプリケーションサーバー内に存在する
2) ログは、ローカルのファイルに出力し、定期的にバックアップする
3) DB の接続情報はアプリケーション内にハードコーディングしている

既存アプリケーションの課題

1. セッション情報は、アプリケーションサーバー内に存在する
   👉 サーバーが中断された場合、セッション情報を失う
   👉 サーバーを水平スケールできない、セッション管理ができない

2. ログは、ローカルのファイルに出力し、定期的にバックアップする
   👉 サーバーが停止した場合、ログのバックアップができない

3. DB への接続情報はアプリケーションのソースコード内にハードコーディングしている
   👉 セキュアではない。開発者は本番環境の機密情報を知る必要がない
   👉 接続情報に変更が発生すると、アプリケーションの修正と再デプロイが必要

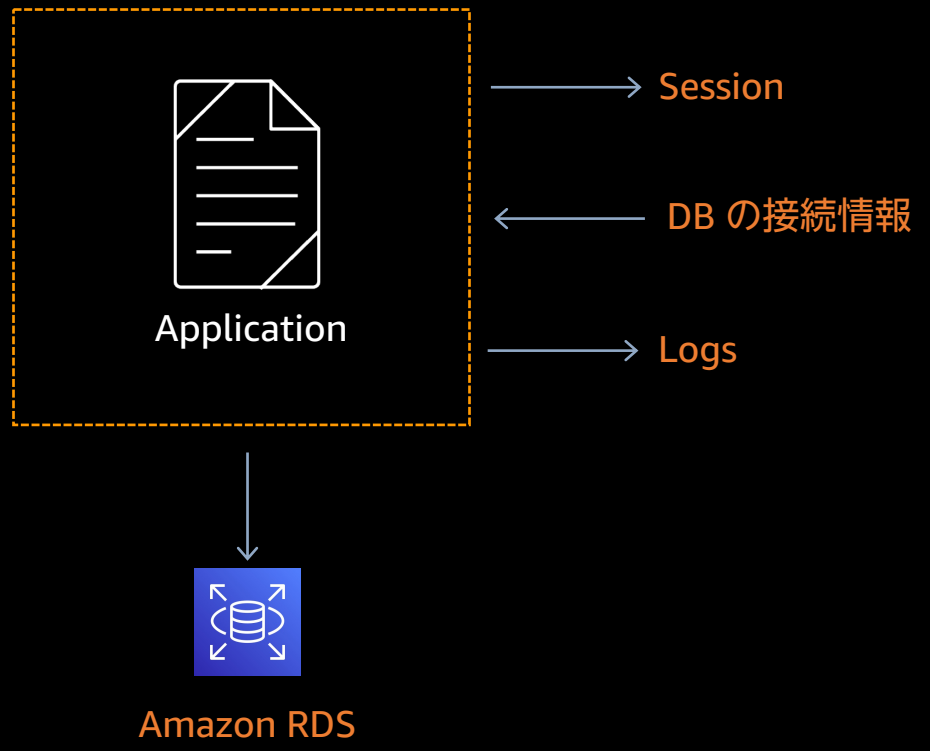
既存アプリケーションの課題

ステート（状態）を持っている

- アプリケーションをスケールさせる際の考慮事項が増える/できない
- サーバー障害によるデータの損失
- システム拡張が難しい

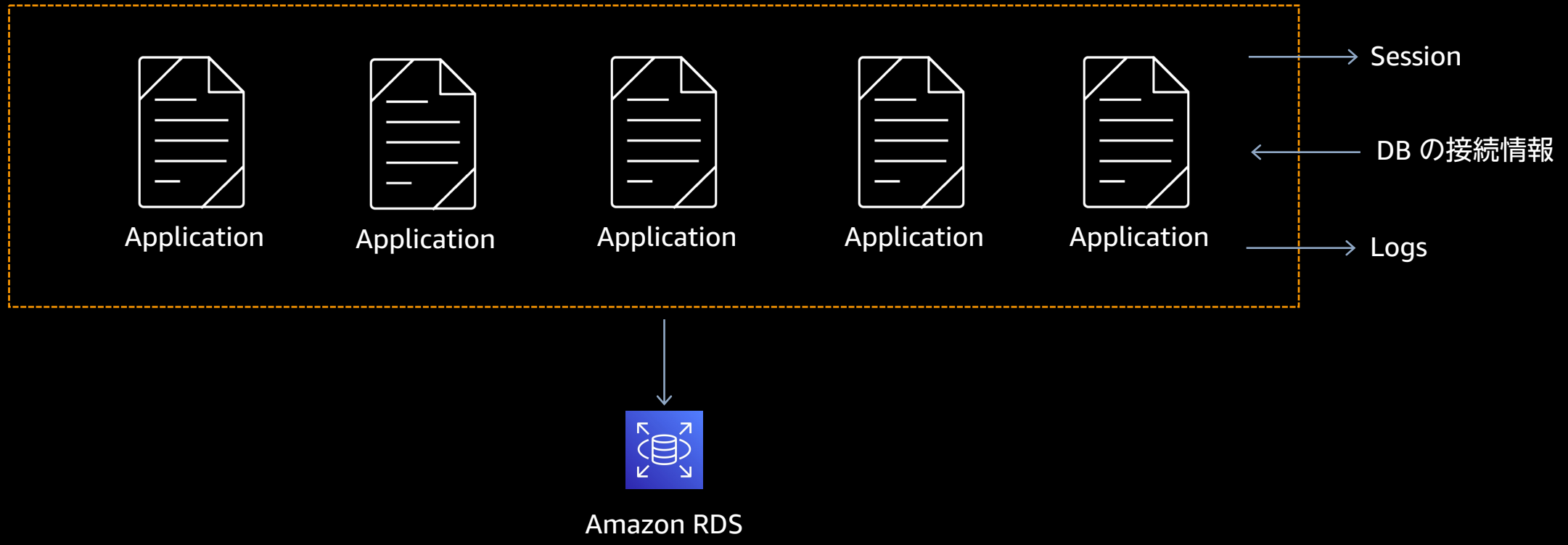
ステートレスなアプリケーション

ステート (状態) をアプリケーションから切り離す

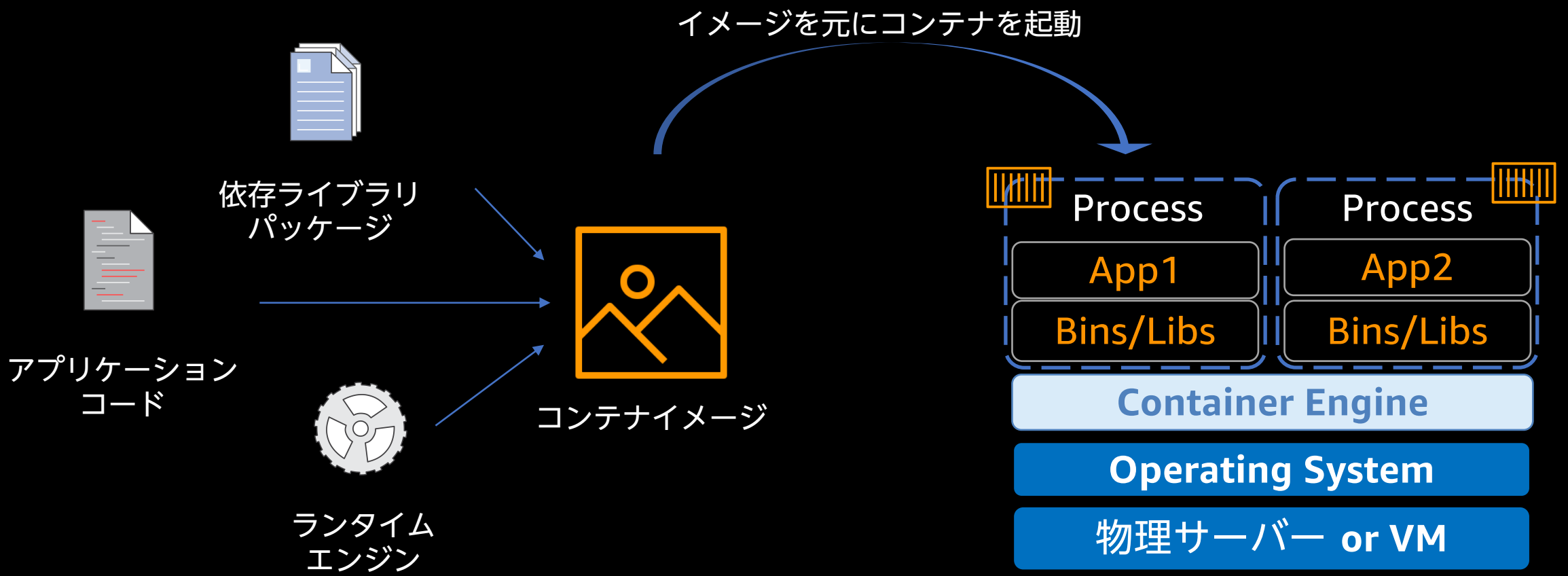


ステートレスなアプリケーション

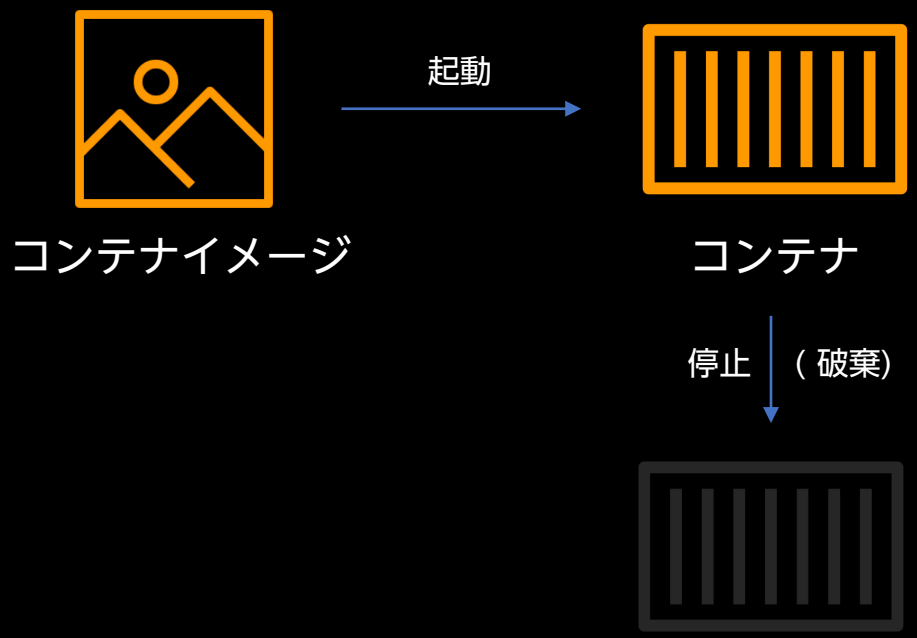
アプリケーションレイヤーのみでスケールできる
(耐障害性、拡張性の向上)



コンテナとは



コンテナとは



■ ステートレス

- ✓ コンテナは停止するとステート (状態) を失う
- ✓ ステートはコンテナの停止と共に消える
  - セッション、途中の処理結果などあらゆるコンテナ内のもの

■ イミュータブル (不変)

- ✓ コンテナは指定のイメージから起動する
  - いつ、どこで起動しても、同じ状態である
- ✓ 起動した後の変更は引き継がれない

コンテナとステートレス

- コンテナ と ステートレス は非常に相性が良い
- コンテナにすることで**軽量で持ち運びが簡単**になる
- コンテナは一つのイメージから起動するので、検証・本番など複数の**環境間の差異を吸収**できる
- ステートをアプリケーションから切り離すことで、**ビルドとデプロイが簡単・高速になる**

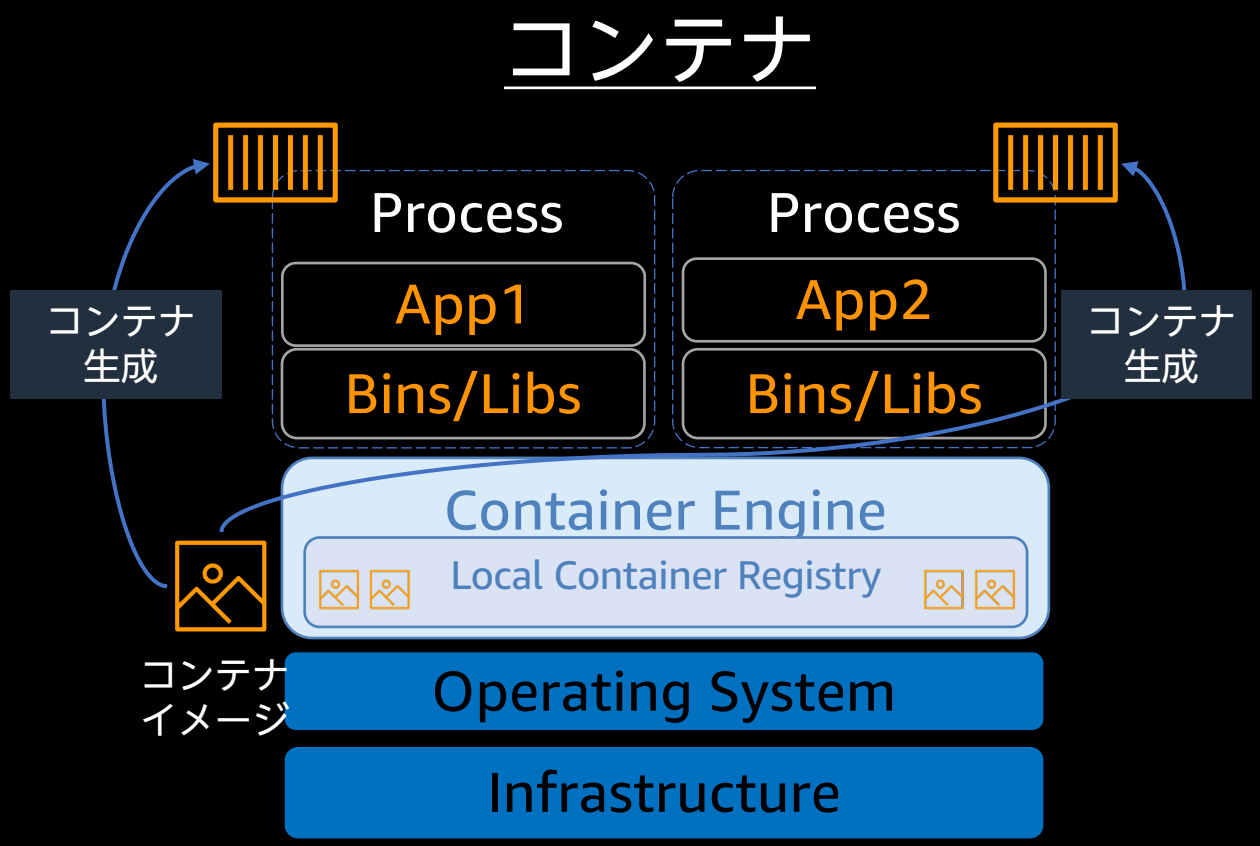
コンテナの特徴とメリット

リソース効率に優れ、オーバーヘッドが少ない
→ 粒度を細かく、利用率の向上が可能

■スピード
✓ 起動・停止が非常に高速

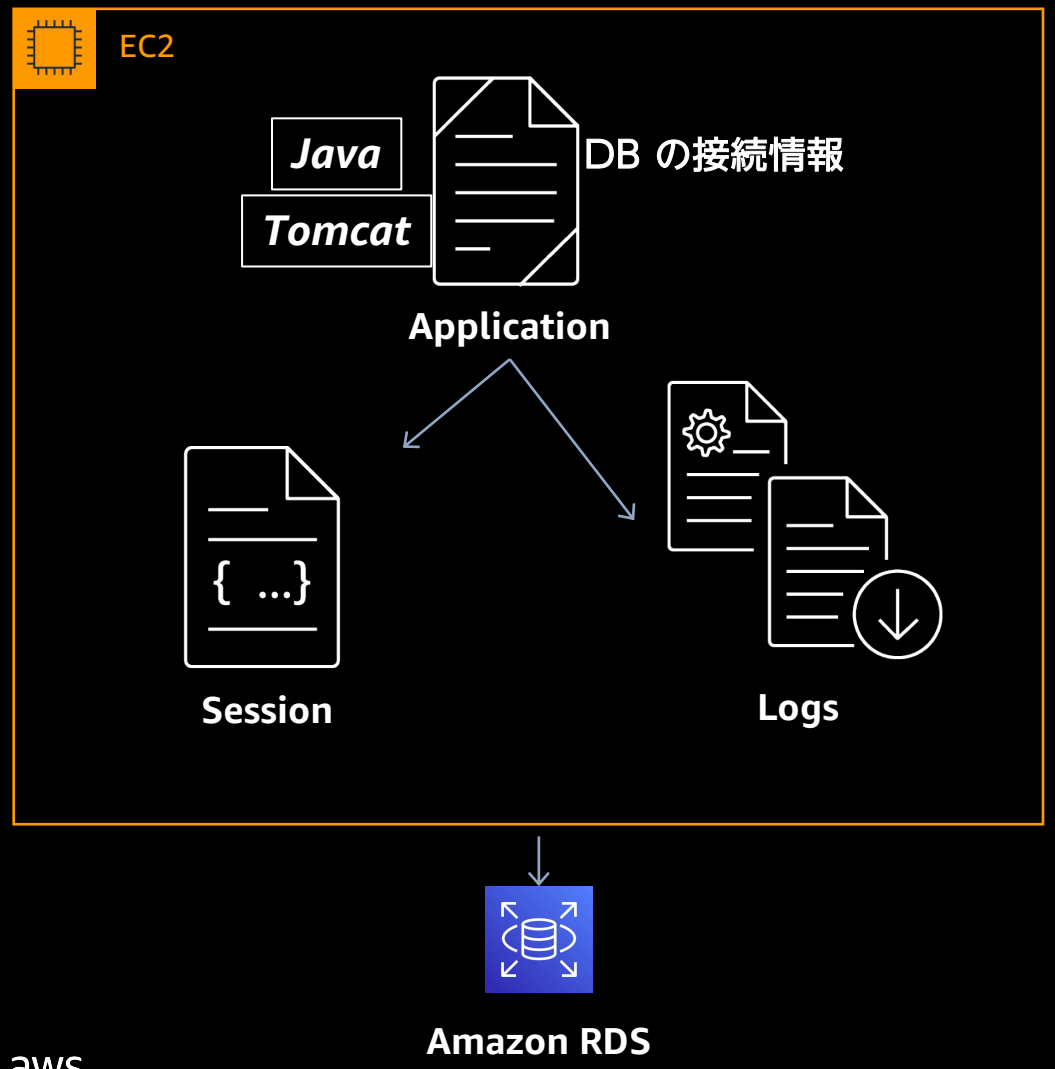
■柔軟性
✓ 1つのイメージから複数の
コンテナを起動可 (スケール性)

■可搬性
✓ コンテナイメージは「不変」
✓ 「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



アプリケーションのリファクタリング

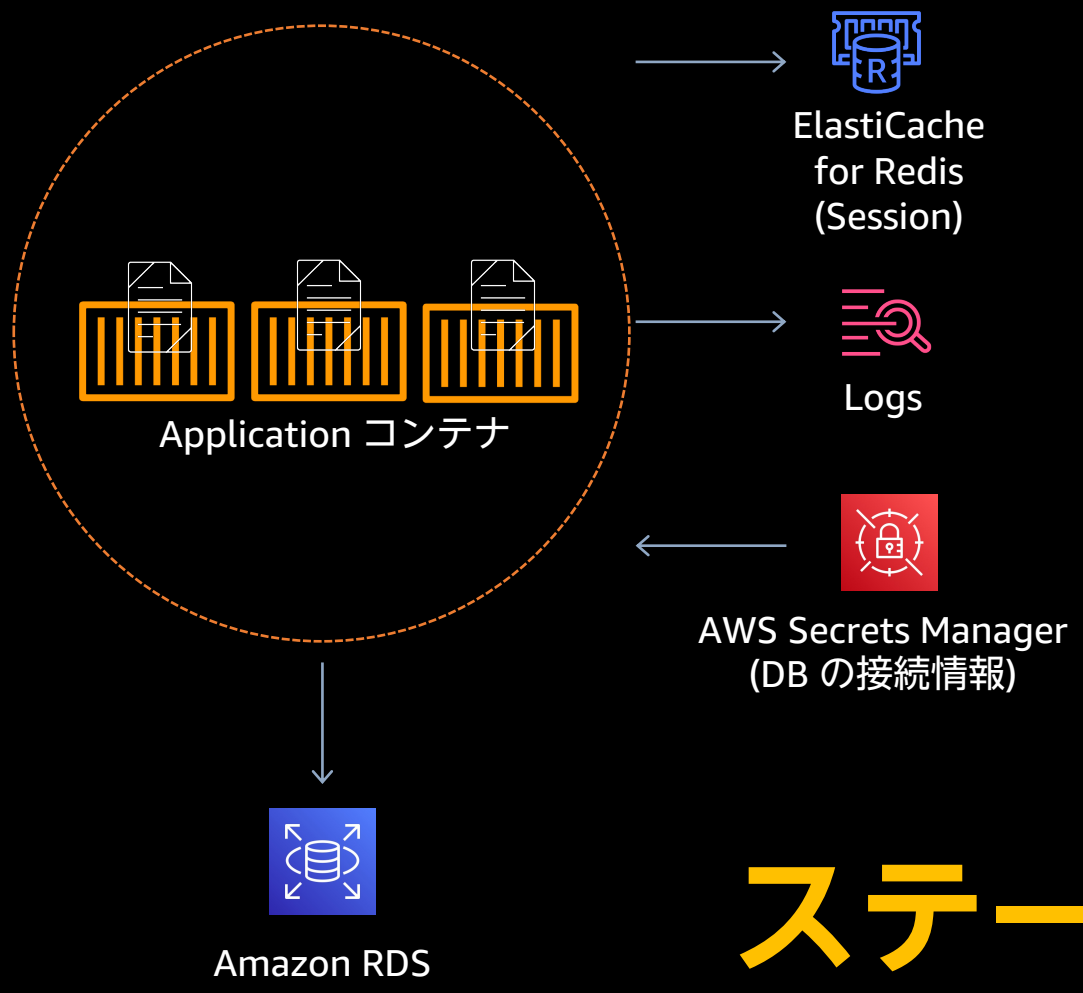
リファクタリング前のアプリケーション



- Java Servlet を利用した Web アプリケーションで、Tomcat, Java で開発されている

1) セッションはサーバーのメモリ上に保持している
2) ログは Log4j を利用してファイルに出力している
3) DB への接続情報は context.xml にハードコーディングしている

リファクタリング後のアプリケーション

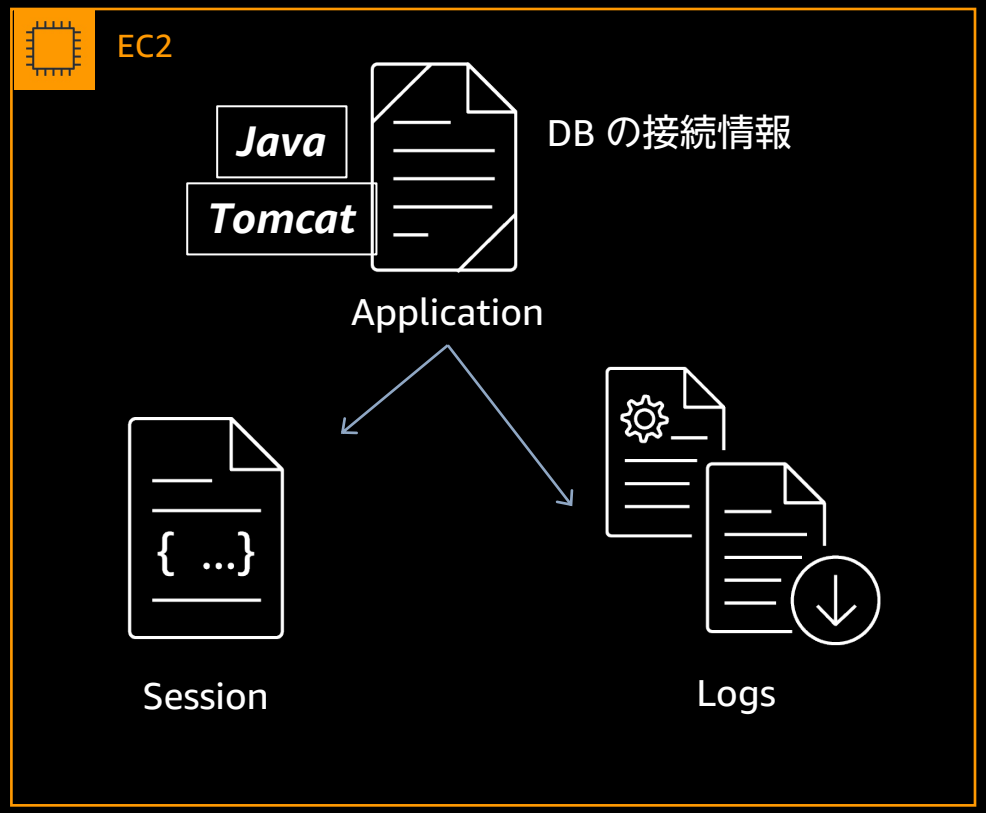


1) セッション情報は、Amazon ElasitCache for Redis に保持する
2) ログは、ストリームとして扱い、Amazon CloudWatch Logs に送る
3) DB の接続情報は AWS Secrets Manager を利用し、アプリケーションからは環境変数として参照する

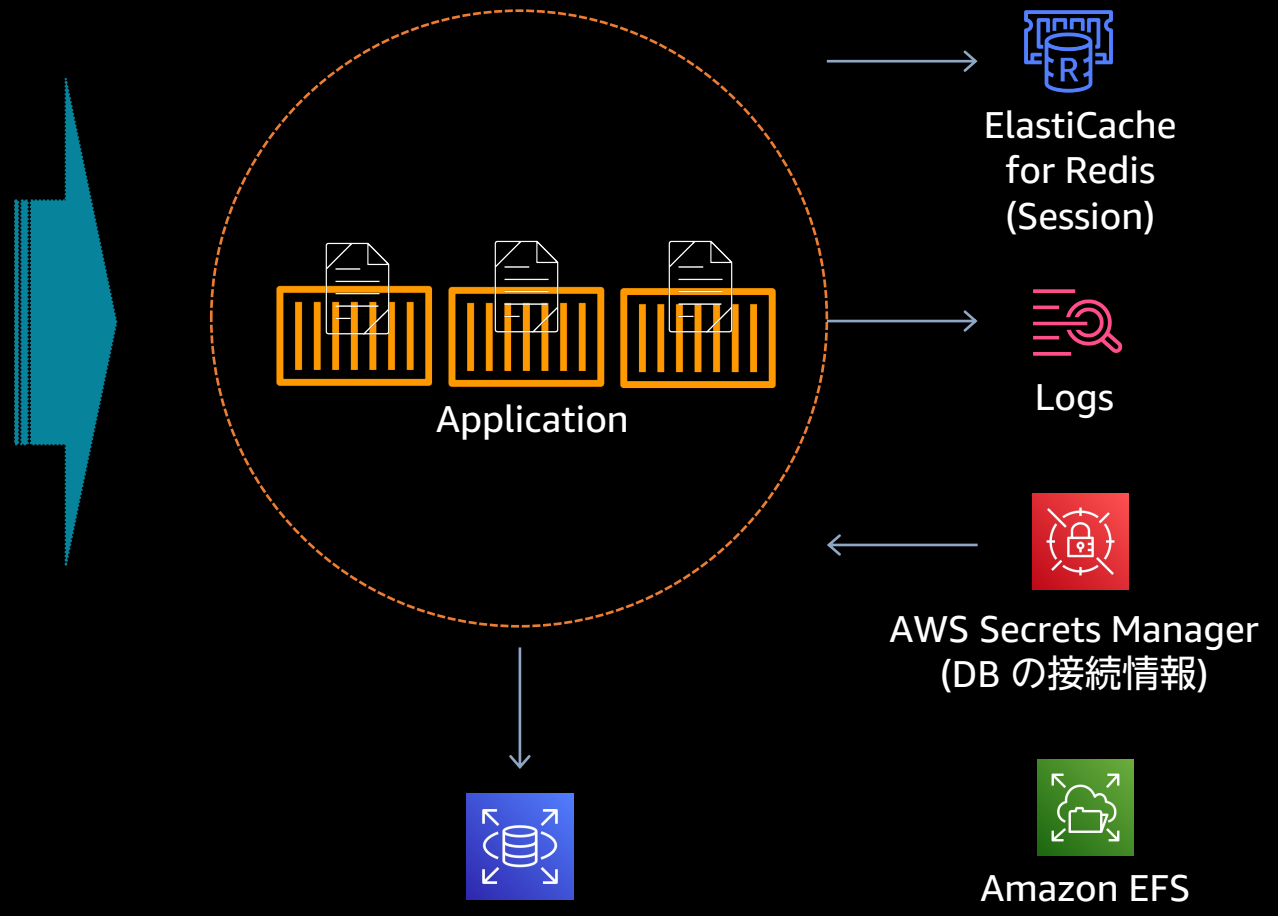
ステートレス

リファクタリング前後

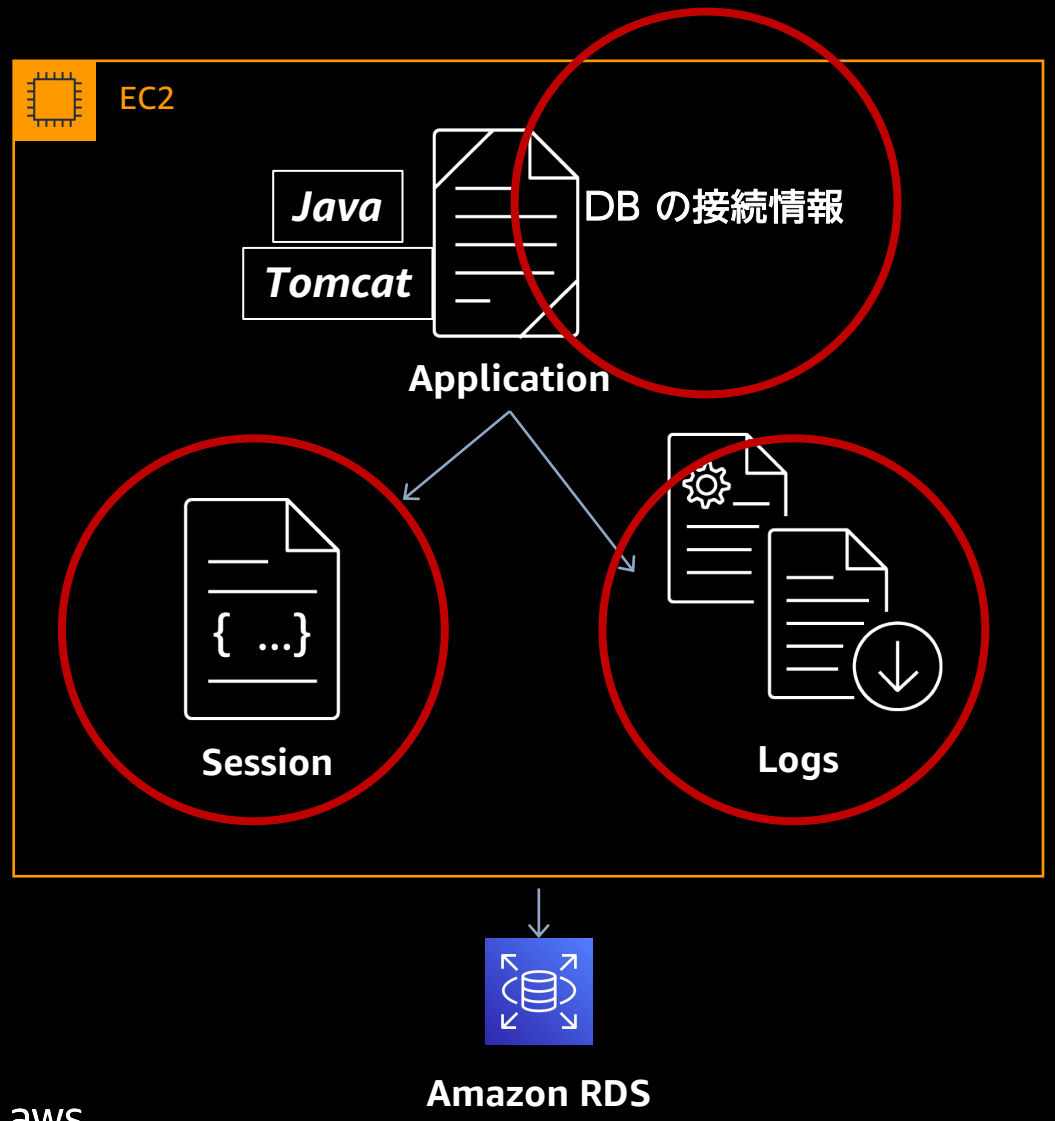
リファクタリング前のアプリケーション



リファクタリング後のアプリケーション



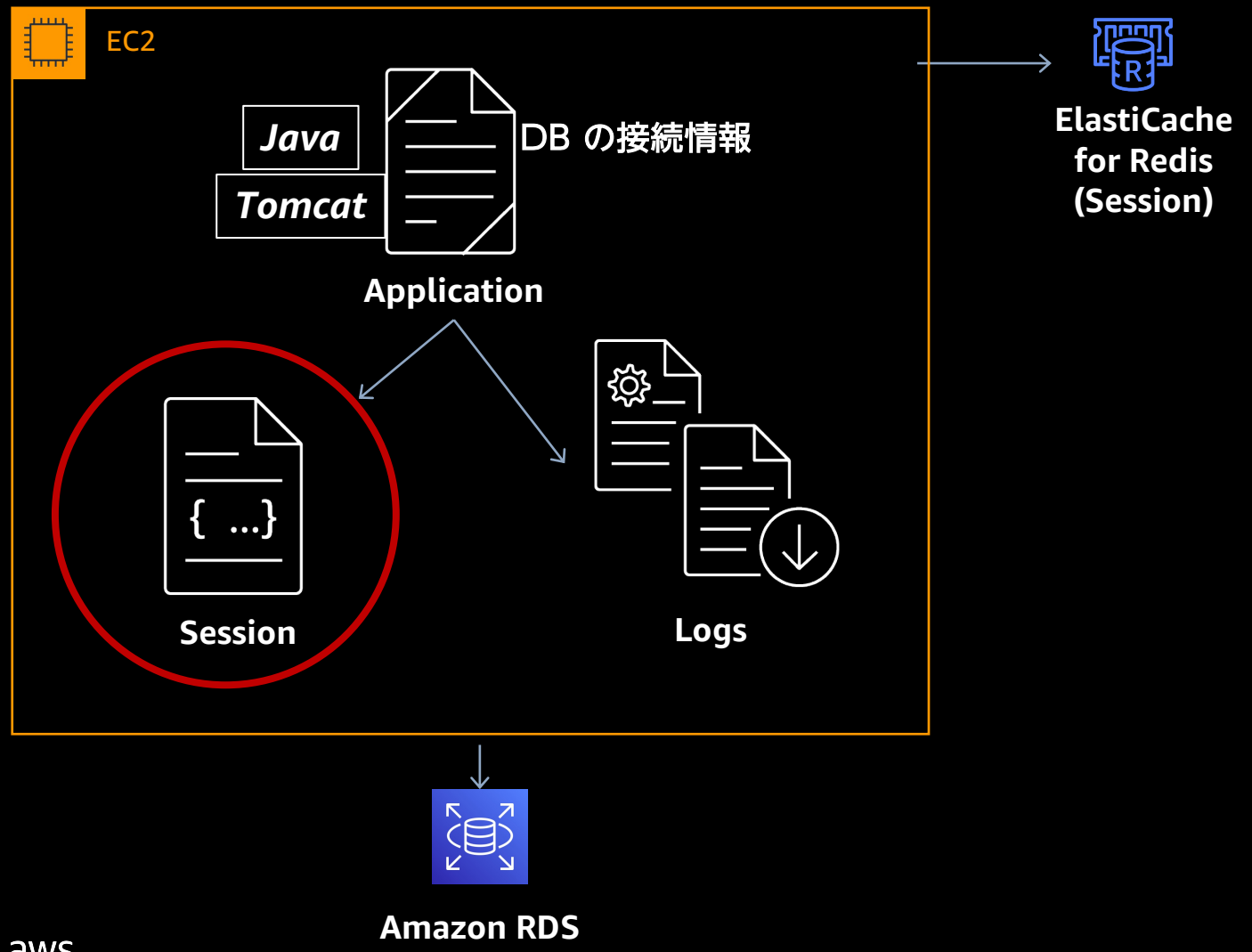
リファクタリングするコンポーネント



- Java Servlet を利用した Web アプリケーションで、Tomcat, Java で開発されている

1) セッションはサーバーのメモリ上に保持している
2) ログは Log4j を利用してファイルに出力している
3) DB への接続情報は context.xml にハードコーディングしている

リファクタリング – セッションの外出し



1) セッションはローカルのメモリ上に保持している

- マネージドサービスである Amazon ElastiCache for Redis に保存するように

セッションの外出し

例：Redisson を使う場合

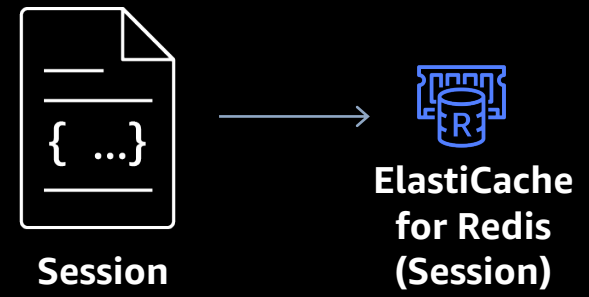
1. Tomcat に Redis 用のライブラリ (Redisson) を追加
2. Redisson の設定ファイルを追加

   例：$CATALINA_HOME/redisson.yaml

```
---
singleServerConfig:
  address: redis://${CACHE_HOST}:${CACHE_PORT}
```

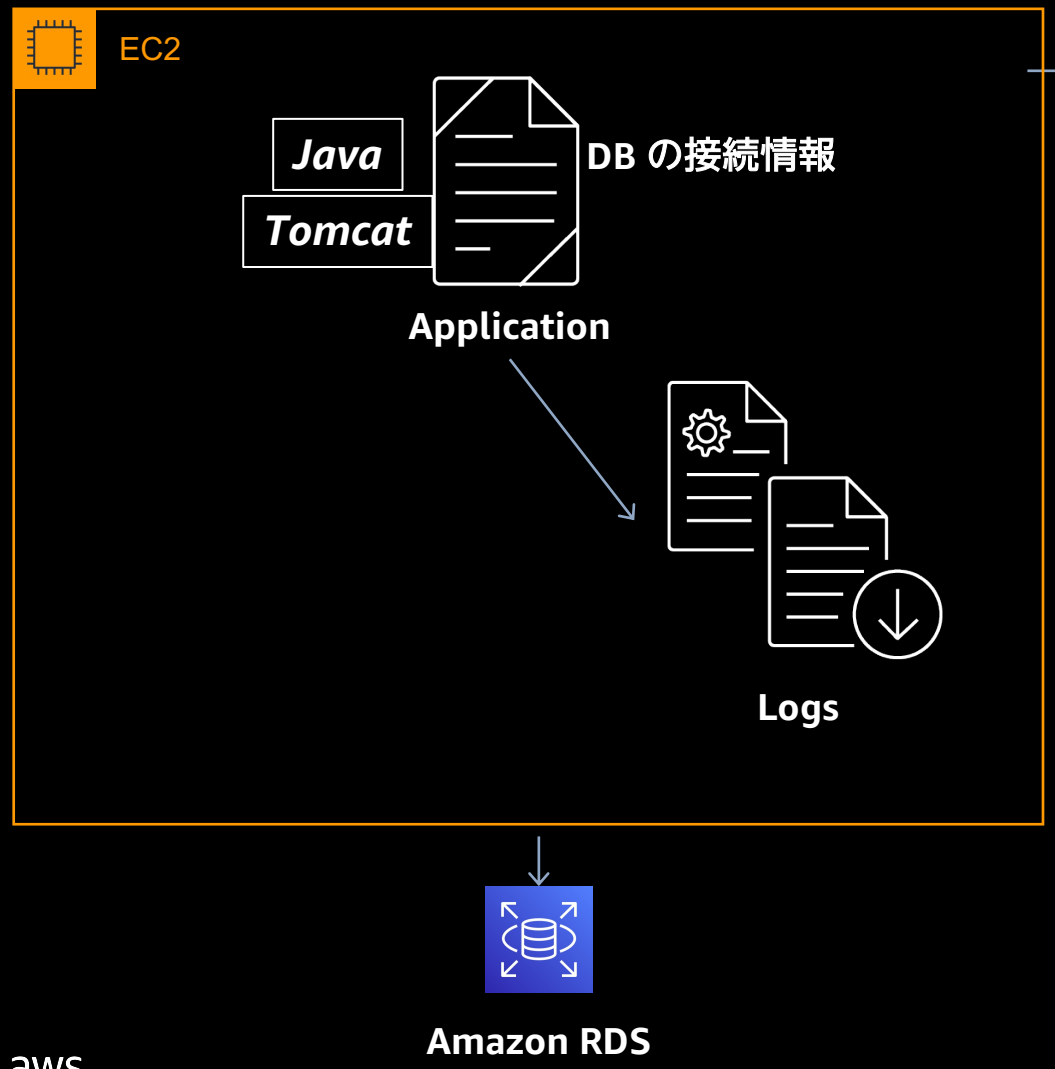
3. Tomcat と Redisson の設定
   - 例：context.xml

```
<Context>
 …
   <Manager className="org.redisson.tomcat.RedissonSessionManager"
     configPath="${CATALINA_HOME}/redisson.yaml"
     readMode="REDIS" updateMode="DEFAULT" />
 …
</Context>
```



https://github.com/redisson/redisson
https://github.com/redisson/redisson/tree/redisson-3.17.5/redisson-tomcat

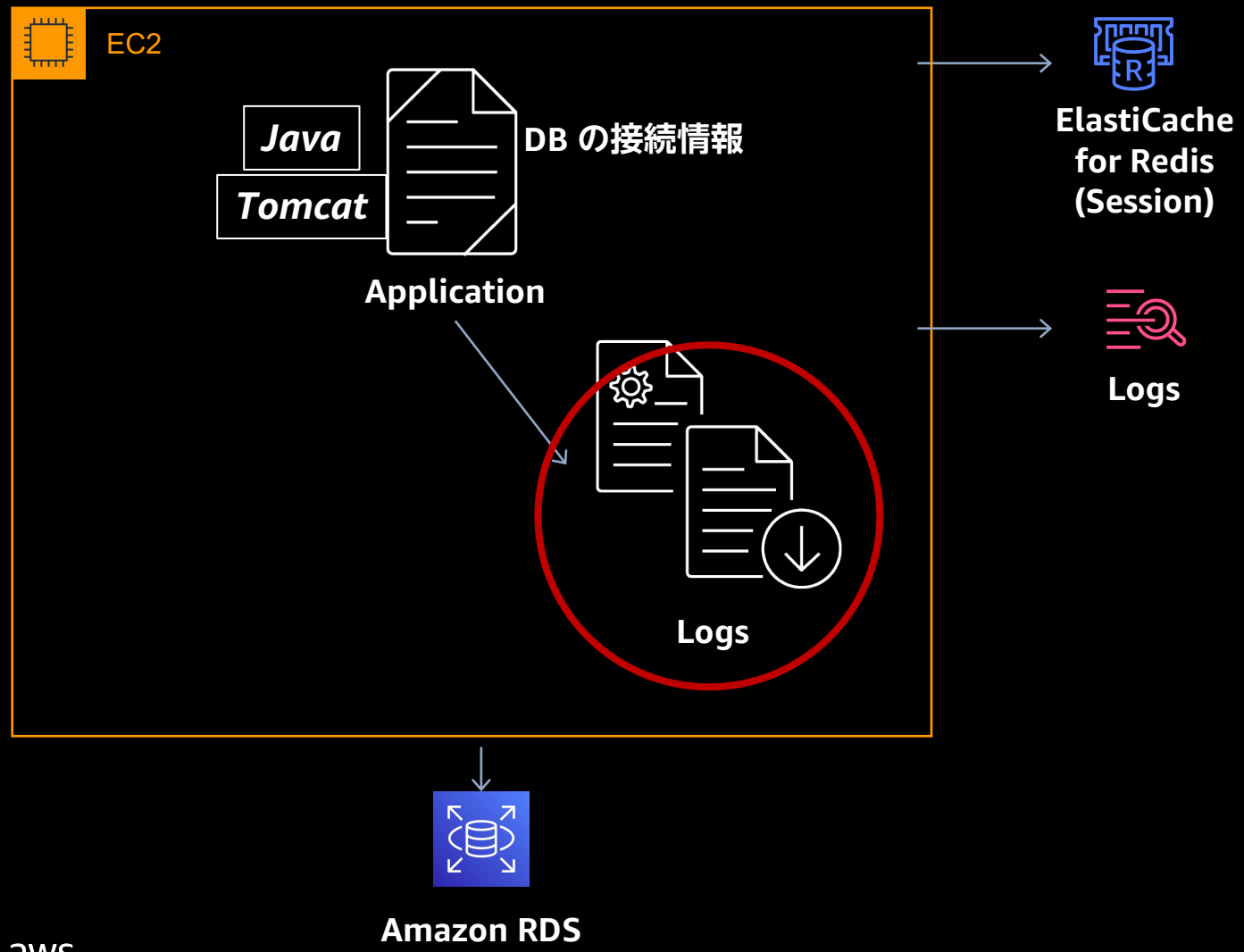
セッションの外出し



ElastiCache for Redis (Session)

- ✓ セッションはマネージドサービスである Amazon ElastiCache for Redis に保存するように

リファクタリング – ログの外出し

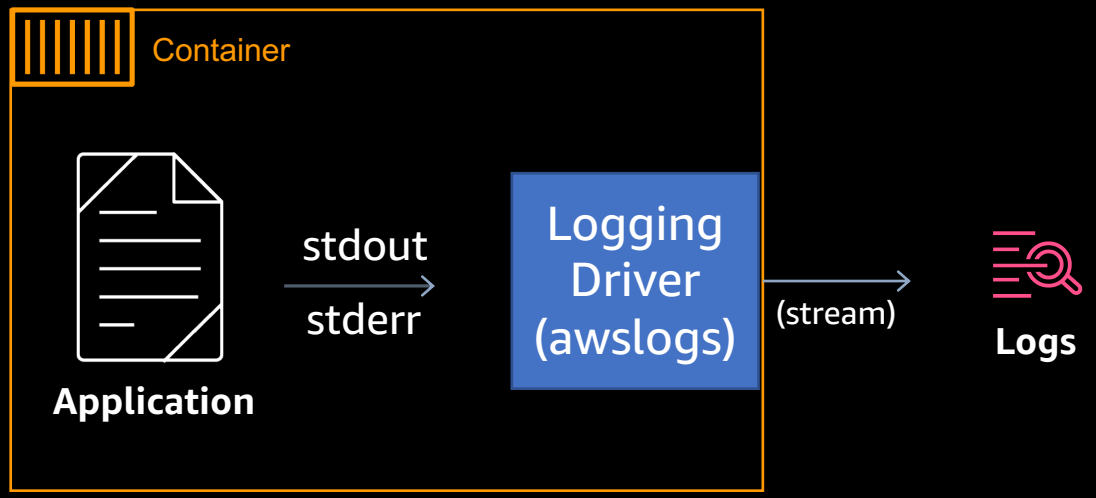


2) ログは Log4j を利用してファイルに出力している

- マネージドサービスである Amazon CloudWatch Logs に保存するように

ログの外出し

- ログは標準出力、標準エラー出力に (ログはストリームとして扱う)
- アプリケーションからの標準出力、標準エラー出力は、
  コンテナの Logging Driver によって収集され指定の場所に送られる



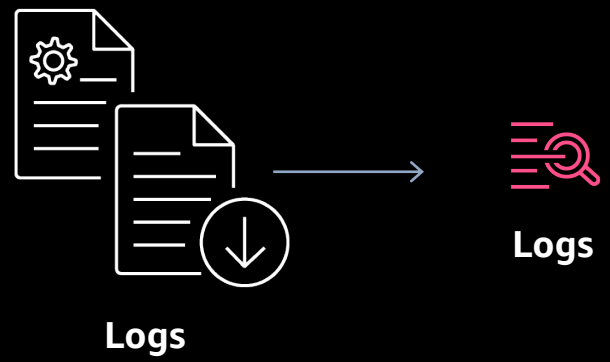
#	logDriver 名	説明
1	awslogs	CloudWatch Logs へ転送
2	FireLens	Fluentd、 CloudWatch Logs、 Kinesis へ転送、 etc
3	json-file	JSON 形式のログ出力
4	syslog	Syslog サーバーへログ転送
	・・・	その他複数

https://docs.docker.com/config/containers/logging/configure/

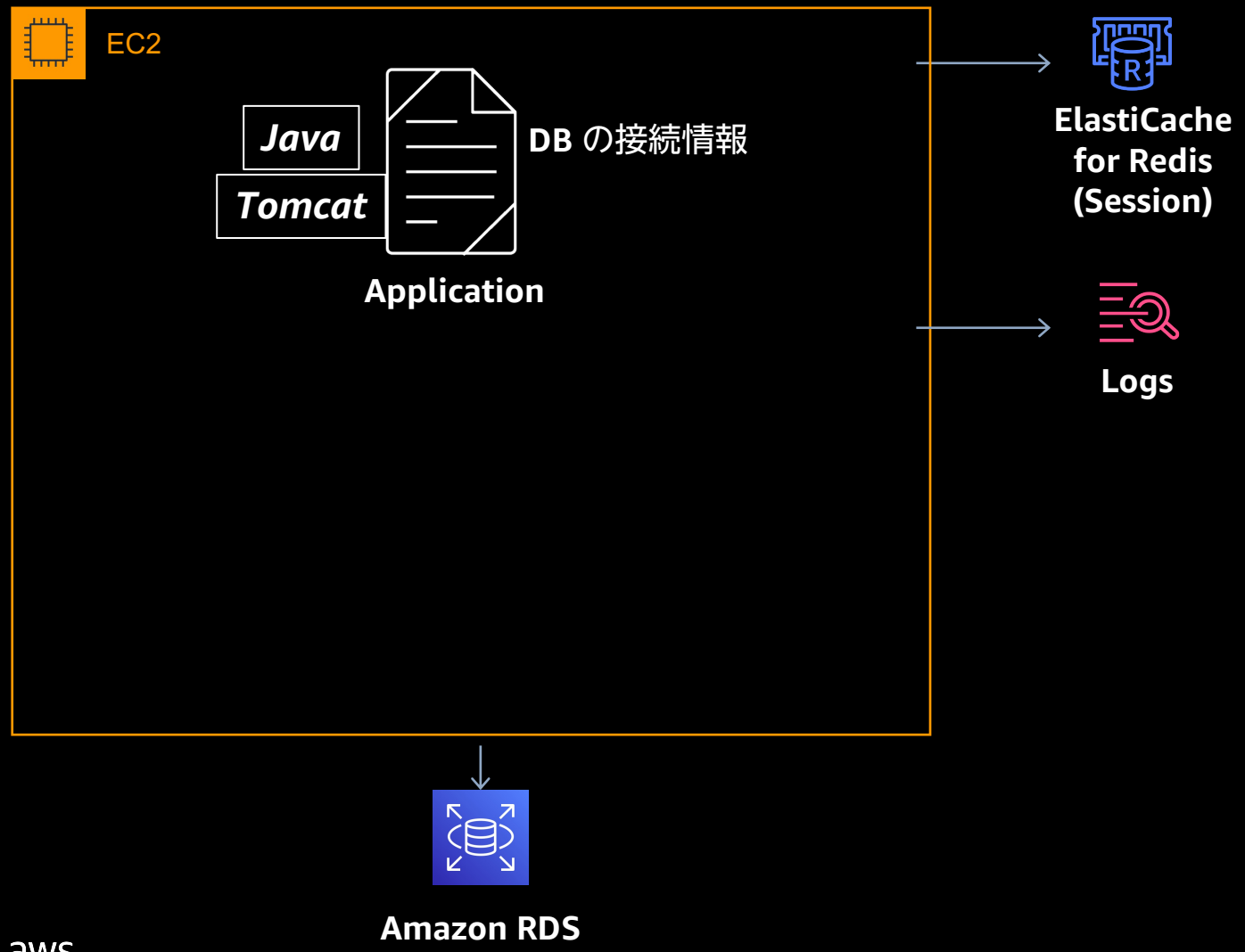
ログの外出し

例：Logger として Log4j 2 利用の場合 (log4j2.xml)

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="warn">
  <Appenders>
    <Console name="console" target="SYSTEM_OUT">
      <PatternLayout
        pattern="%d{yyyy-MM-dd HH:mm:ss} %-5p %c{1}:%L - %m%n" />
    </Console>
  </Appenders>
  <Loggers>
    <Root level="info" additivity="false">
      <AppenderRef ref="console" />
    </Root>
  </Loggers>
</Configuration>
```

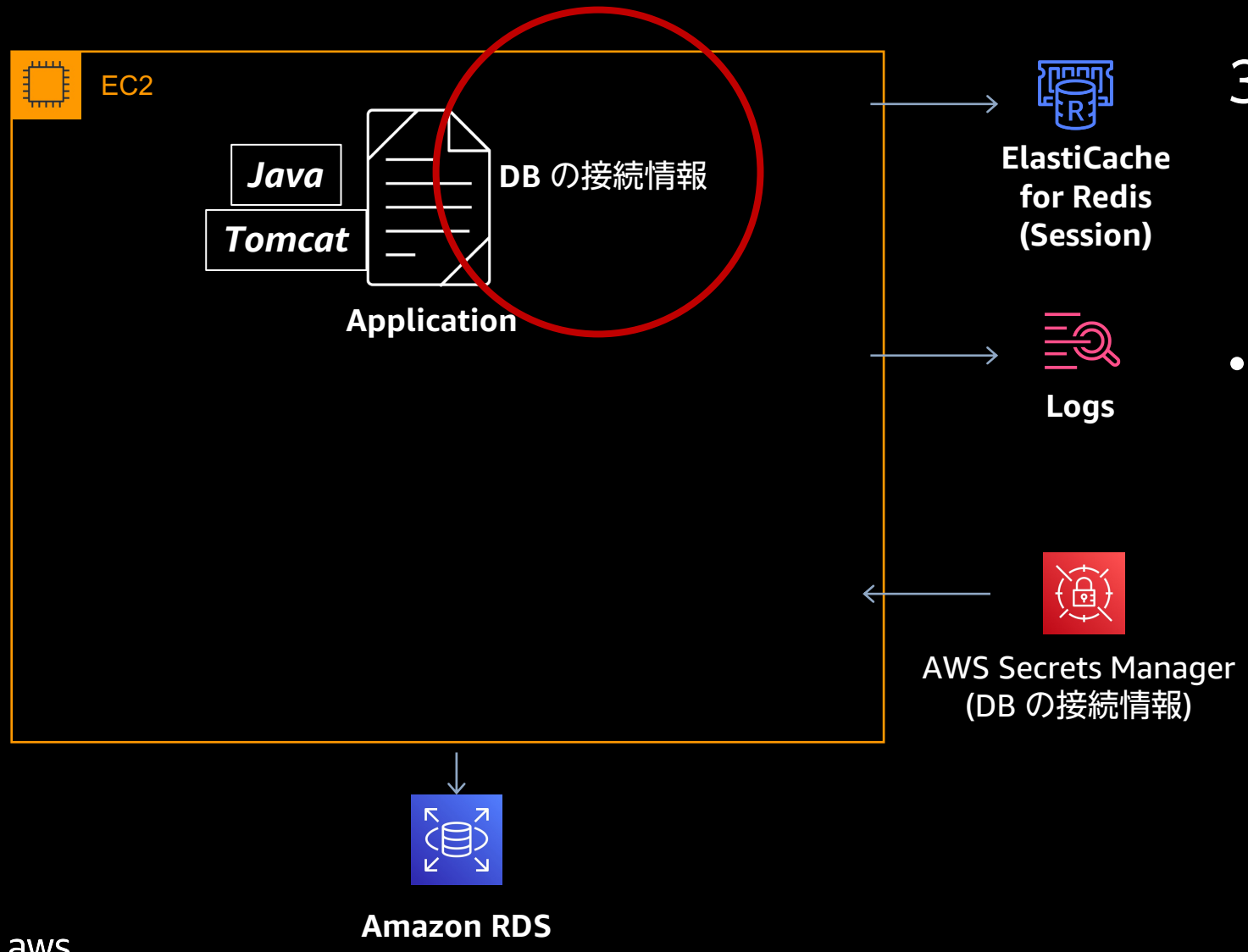


ログの外出し



- ✓ ログはストリームとして、stdout / stderr に出力する
- ✓ マネージドサービスである Amazon CloudWatch Logs に保存するように

リファクタリング – 環境変数からの参照



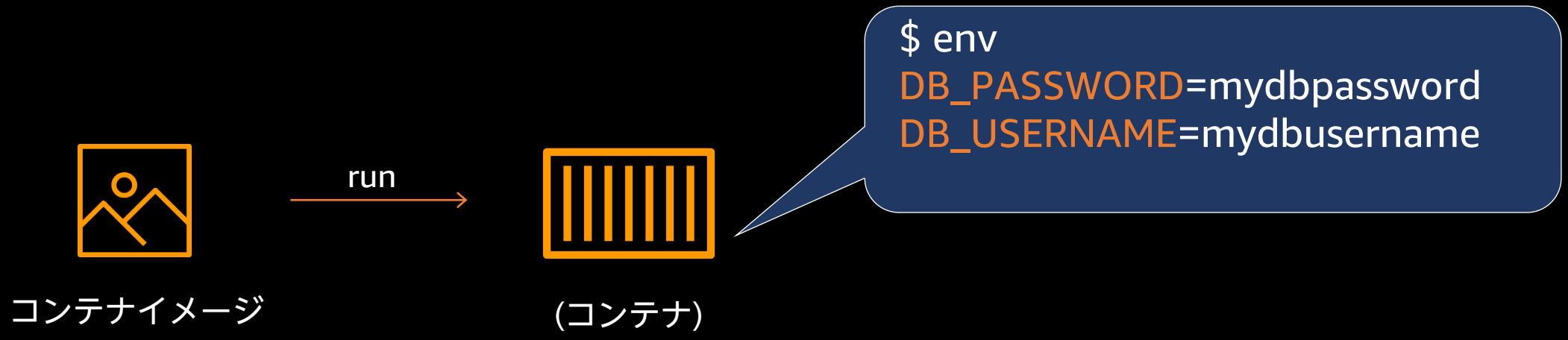
3) DB の接続情報は context.xml にハードコーディングしている

- AWS Secrets Manager に機密情報を保持し、アプリケーションからは環境変数として参照する

コンテナに環境変数を渡す

- Dockerfile に ENV 命令で記載する (Dockerfile については後術)
- コンテナを起動する際にオプションで環境変数を渡すことができる

例：
$ docker run -it --env DB_PASSWORD=mydbpassword --env DB_USERNAME=mydbusername app



機密情報の環境変数からの参照

- AWS Secrets Manager に DB の機密情報を作成
- アプリケーションは環境変数から機密情報を取得するように設定する
  - 例：context.xml

```
<Context>
 <Resource
  name="jdbc/MainDB"
  auth="Container"
  type="javax.sql.DataSource"
  maxTotal="100" maxIdle="30" maxWaitMillis="10000"
  username="${DB_USERNAME}"
  password="${DB_PASSWORD}"
  driverClassName="com.mysql.cj.jdbc.Driver"
  url="jdbc:mysql://${DB_HOST}:3306/mydatabase" />
</Context>
```

- コンテナ起動時に AWS Secrets Manager から機密情報を取得し、コンテナの環境変数にセットする

機密情報の環境変数からの参照

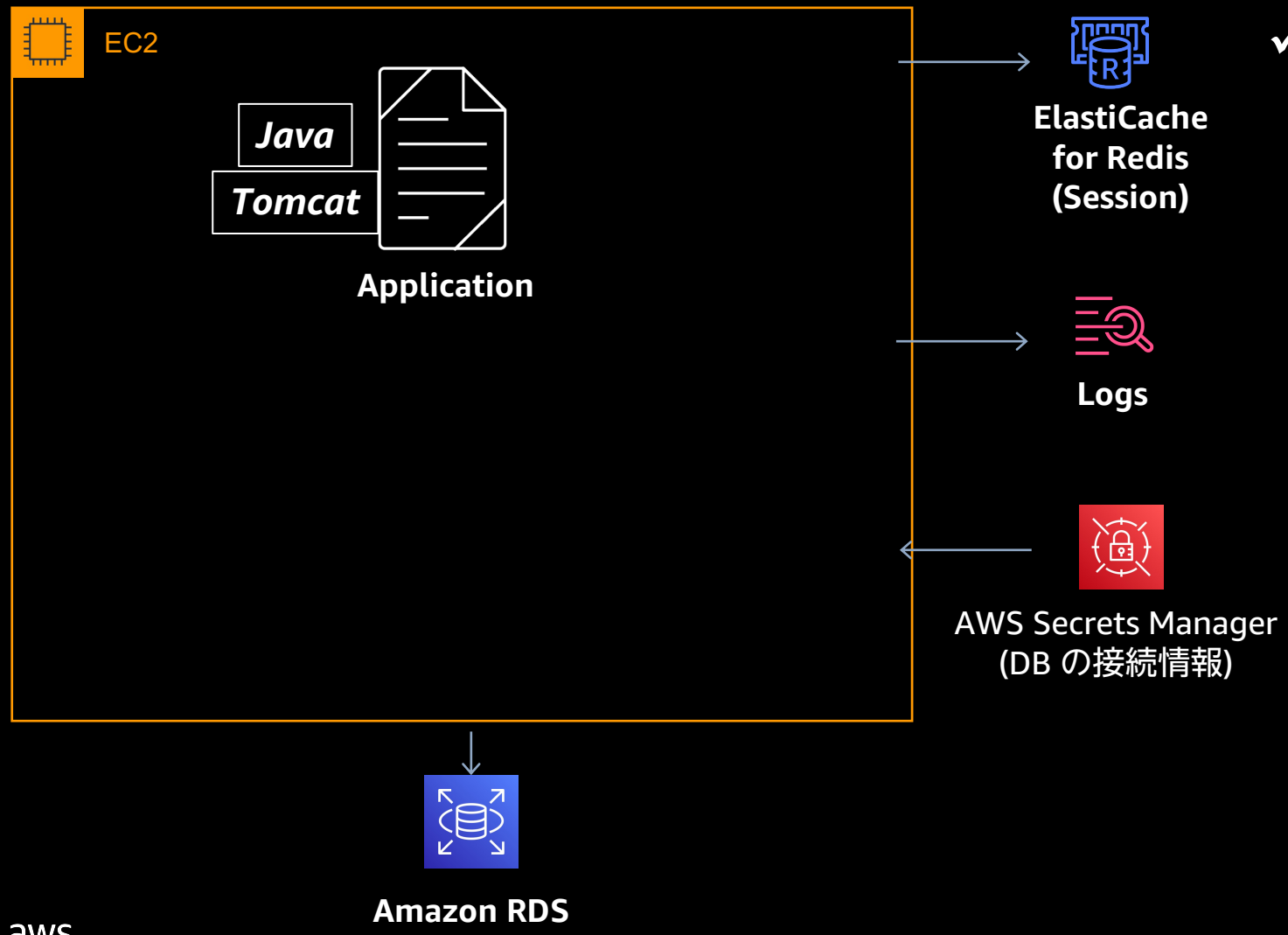
- Amazon ECS の場合 AWS Secrets Manager との連携がされている
  タスク定義の際に値のタイプに “ValueFrom” を指定し、
  作成した AWS Secrets Manager の ARN を指定する

AWS Secrets Manager

AWS Secrets Manager に設定した機密情報が環境変数としてコンテナ起動時に展開される

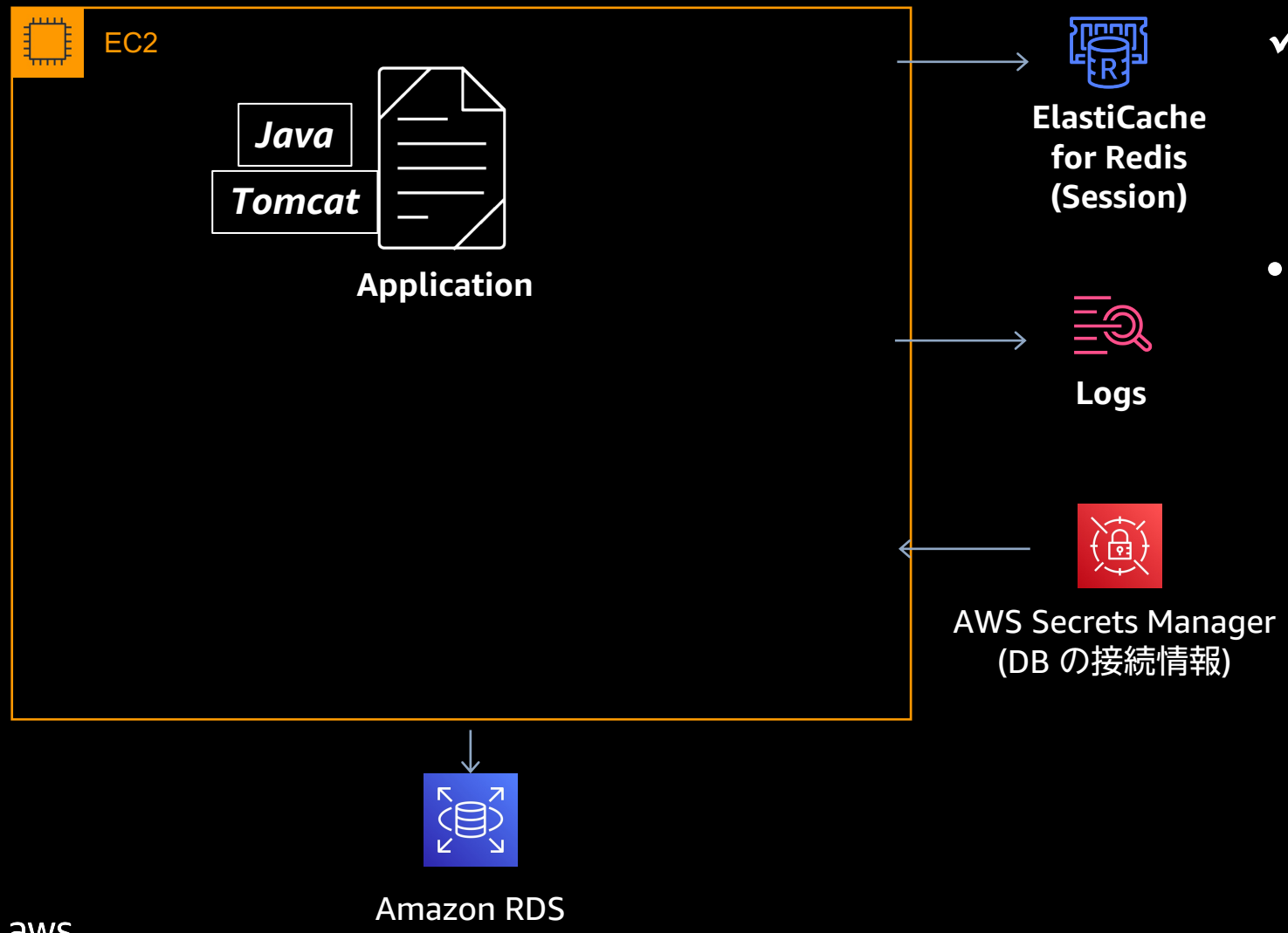
https://docs.aws.amazon.com/ja_jp/AmazonECS/latest/developerguide/specifying-sensitive-data-secrets.html

リファクタリング – 環境変数からの参照



- ✓ DB の接続情報 (機密情報)はアプリケーションソースコードの外部に保持し、必要な時に環境変数にセットして利用する

リファクタリング - 完了



- ✓ 残ったのはステートレスなアプリケーションのみ
- ステートはアプリケーションから切り離し、アプリケーションをコンテナにする為の準備完了！

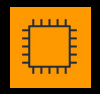
コンテナ化

コンテナの作成手順

- コンテナはコンテナイメージを元に run コマンドで起動する
- コンテナイメージは通常 Dockerfile を元に build コマンドで生成する
- Dockerfile はテキスト形式のファイルで、コンテナイメージを作成するための手順を記載する

アプリケーションのコンテナ化

- Dockerfile の基本は EC2 でのアプリケーションの作りと同じ



EC2 の user-data

```
#!/bin/bash

yum install -y java
wget tomcat
< 設定、ビルド、配置 >
```

Dockerfile

```
FROM public.ecr.aws/docker/library/amazoncorretto:latest

RUN wget tomcat
RUN < 設定、ビルド、配置 >

ENV JAVA_HOME=/usr/lib/jvm/default-jvm

ENTRYPOINT ["java","-jar","/app.jar"]
```

https://docs.docker.com/language/java/

Dockerfile のベストプラクティス

基本的な書き方と構文から高度なテクニックまで公開されている

Best practices for writing Dockerfiles

This document covers recommended best practices and methods for building efficient images.

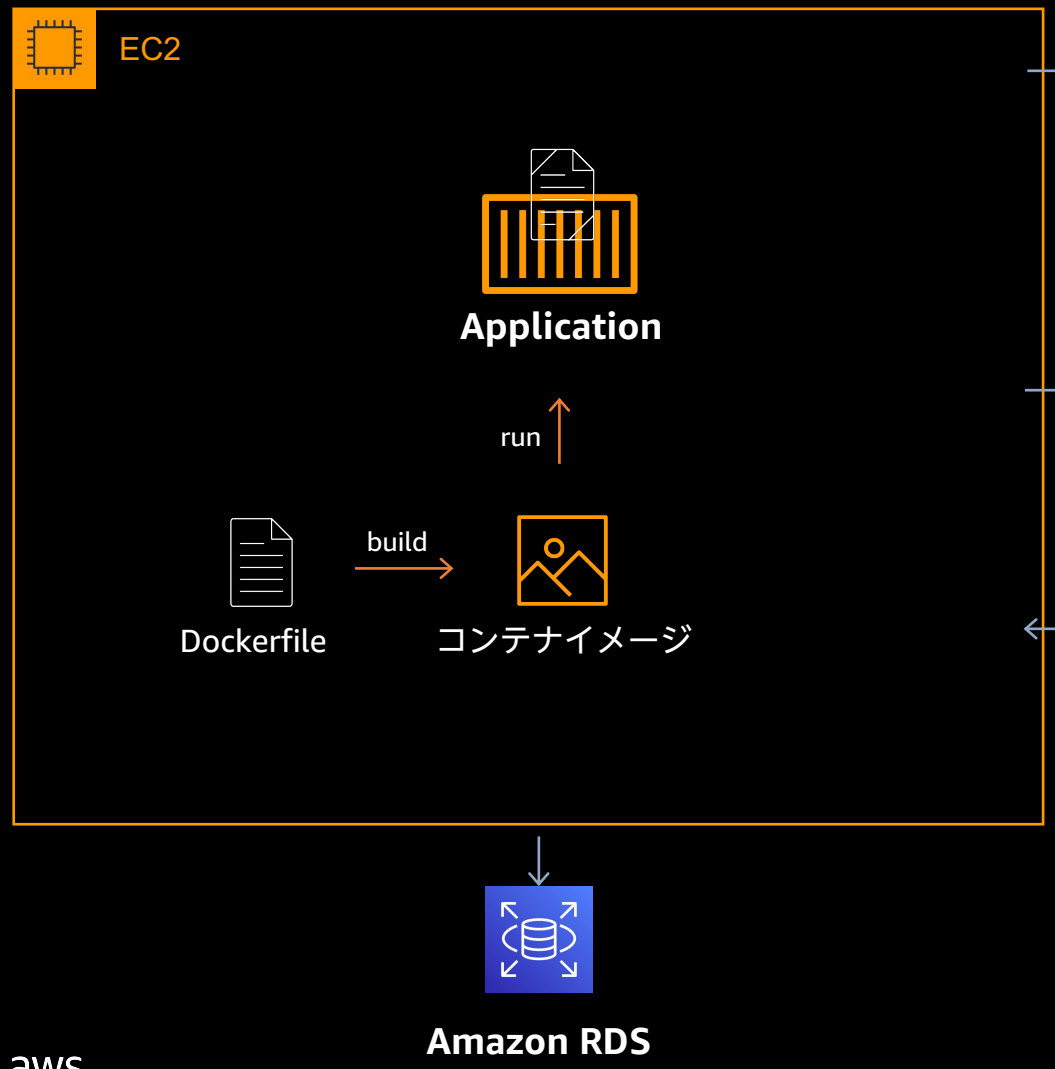
Docker builds images automatically by reading the instructions from a `Dockerfile` -- a text file that contains all commands, in order, needed to build a given image. A `Dockerfile` adheres to a specific format and set of instructions which you can find at Dockerfile reference.

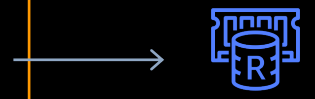
A Docker image consists of read-only layers each of which represents a Dockerfile instruction. The layers are stacked and each one is a delta of the changes from the previous layer. Consider this `Dockerfile` :

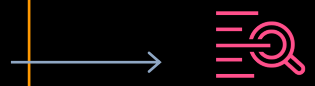
```
# syntax=docker/dockerfile:1
FROM ubuntu:18.04
COPY . /app
RUN make /app
CMD python /app/app.py
```

https://docs.docker.com/develop/develop-images/dockerfile_best-practices/

アプリケーションのコンテナ化



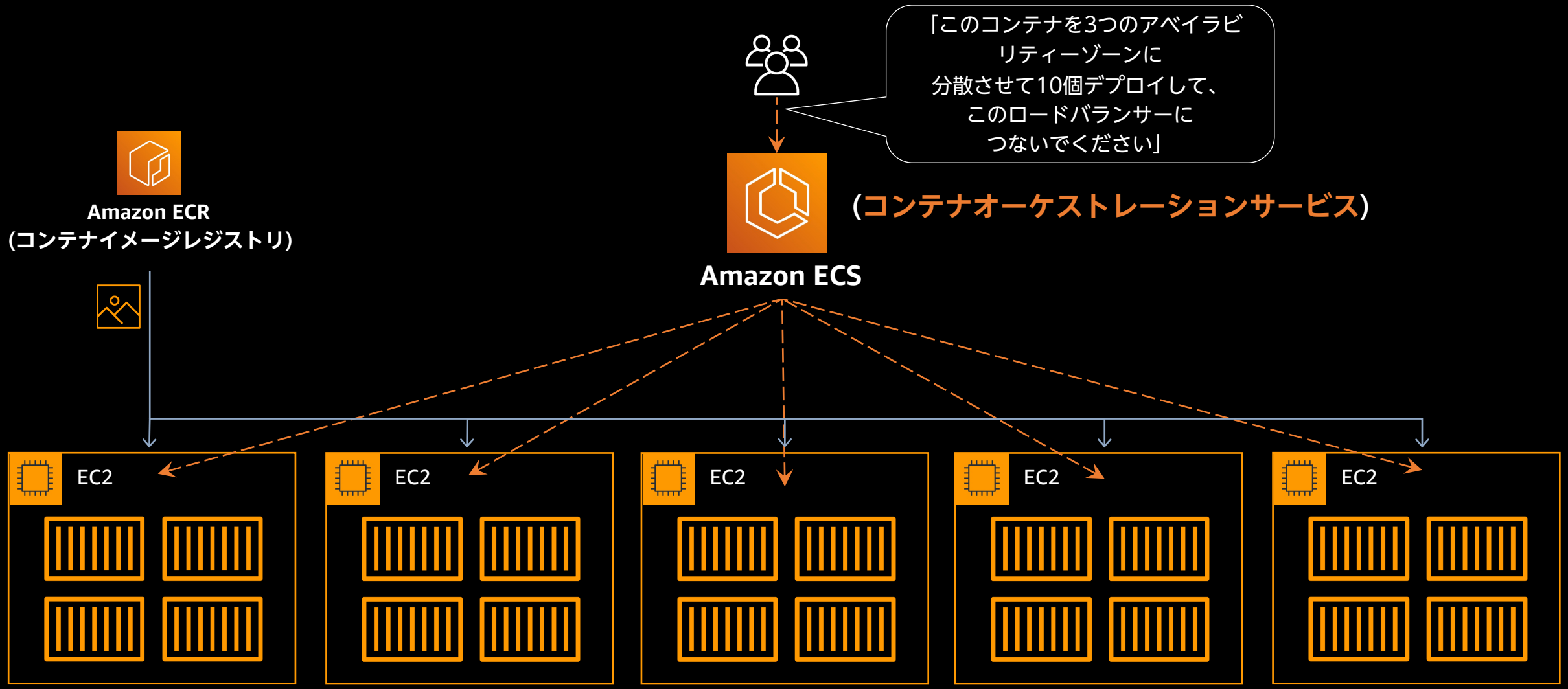




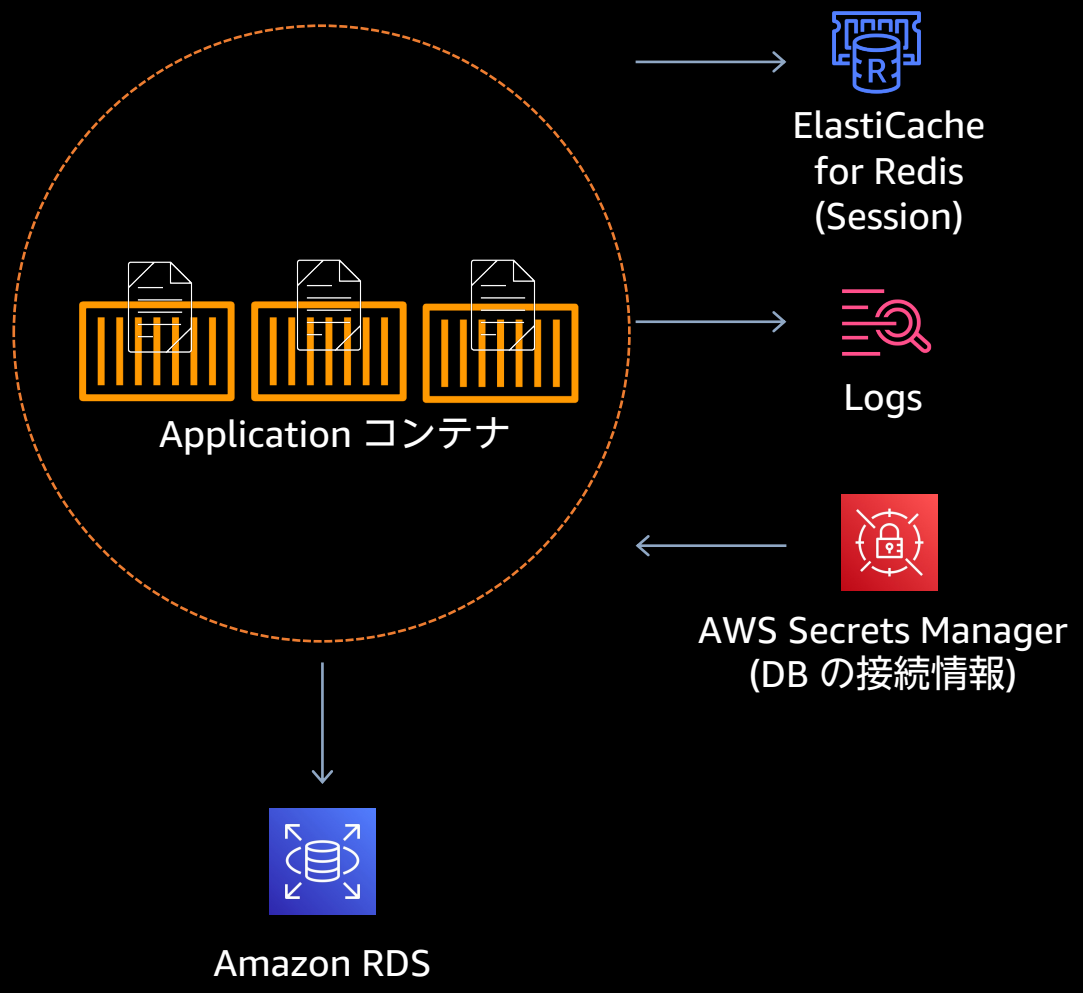
- ✓ アプリケーションのコンテナ化作業完了！

- しかし、、、
- どのようにコンテナを複数のサーバーで起動するのか？デプロイするのか？
- Amazon ECS にデプロイするには？

コンテナオーケストレーション



Amazon ECS にデプロイしたアプリケーション

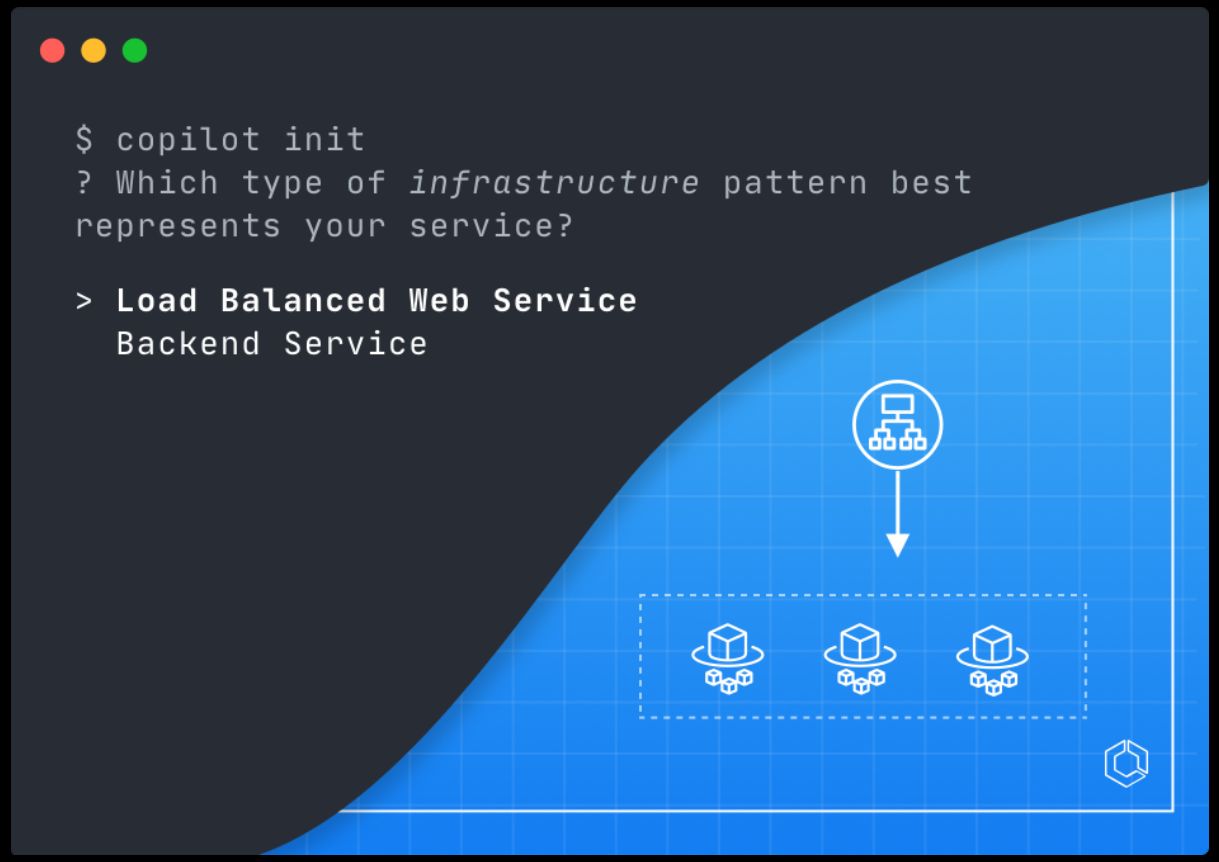


1) セッション情報は、Amazon ElasitCache for Redis に保持する
2) ログは、ストリームとして扱い、Amazon CloudWatch Logs に送る
3) DB の接続情報は AWS Secrets Manager を利用し、アプリケーションからは環境変数として参照する

Amazon ECS へのデプロイ

Amazon ECS へのデプロイ – AWS Copilot CLI

開発者が本番環境でコンテナ化されたアプリケーションを構築、リリース、操作するためのツール



- AWS が開発した OSS CLI ツール
- 開発者が Amazon ECS でコンテナ化されたアプリケーションを簡単に構築、リリース、操作できるように
- インフラストラクチャではなく、アプリケーション開発に焦点を当てている
  - ECS クラスターと CI/CD パイプラインをインタラクティブにプロビジョニングする
  - 宣言的サービス定義を使用

https://github.com/aws/copilot-cli

Amazon ECS へのデプロイ – AWS Copilot CLI

Manifest ファイル例

```
name: ‘myapp'
type: 'Load Balanced Web Service'

image:
  build: './frontend/Dockerfile'
  port: 8080

http:
  path: '/'
  healthcheck: '/_healthcheck'

cpu: 256
memory: 512

count: 20

variables:
  LOG_LEVEL: info

secrets:
 DB_USERNAME:
   secretsmanager: 'demo/test/mysql:username::'
  DB_PASSWORD:
   secretsmanager: 'demo/test/mysql:password::'
```

Dockerfile の場所
利用する port 番号

ALB の設定

タスクあたりの CPU, MEM
の利用設定

起動するタスクの数

環境変数の指定

AWS Secrets Manager
のシークレットの値を
環境変数に展開する設定

注：Amazon ECS におけるタスクは、1 つ以上のコンテナの集まり

- サービスの一般的な構成を含む yaml ファイル
- Copilot CLI は CloudFormation テンプレートを作成し、このファイルに基づいて AWS リソースをデプロイする
- ECS タスク定義、ECS サービス構成、および ALB 構成をこの 1 つのファイルで管理できる
- CI/CD パイプラインの作成、管理も可能

https://aws.github.io/copilot-cli/ja/docs/manifest/overview/

ローカル環境でのコンテナ開発

ローカル環境でのコンテナ開発

例：compose.yaml

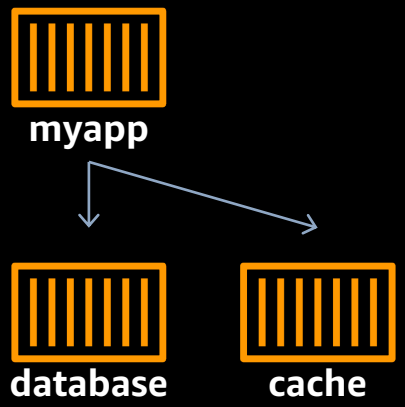
```yaml
services:
  # アプリケーションコンテナの設定
  myapp:
    build: .
    ports:
      - 8080:8080
    depends_on:
      - database
    environment:
      DATABASE_HOST: database
      DATABASE_USERNAME: username
      DATABASE_PASSWORD: databasepassword
      CACHE_HOST: cache

# データベースコンテナの設定
  database:
    image: “public.ecr.aws/docker/library/mysql:8.0.29”
    environment:
      MYSQL_ROOT_PASSWORD: adminpassword
      MYSQL_DATABASE: databasename
    volumes:
      - “./src/main/resources/data:/docker-entrypoint-initdb.d”

# Redisコンテナの設定
  cache:
    image: "public.ecr.aws/docker/library/redis:latest"
    ports:
      - 6379:6379
    volumes:
      - "./data/redis:/data"
```

- ローカルで開発する際には、Docker Desktop、Rancher Desktop、Finch などが利用可能

Finch

$ docker compose up –d

$ finch compose up -d



https://github.com/runfinch/finch

ローカル環境と AWS 環境での違い

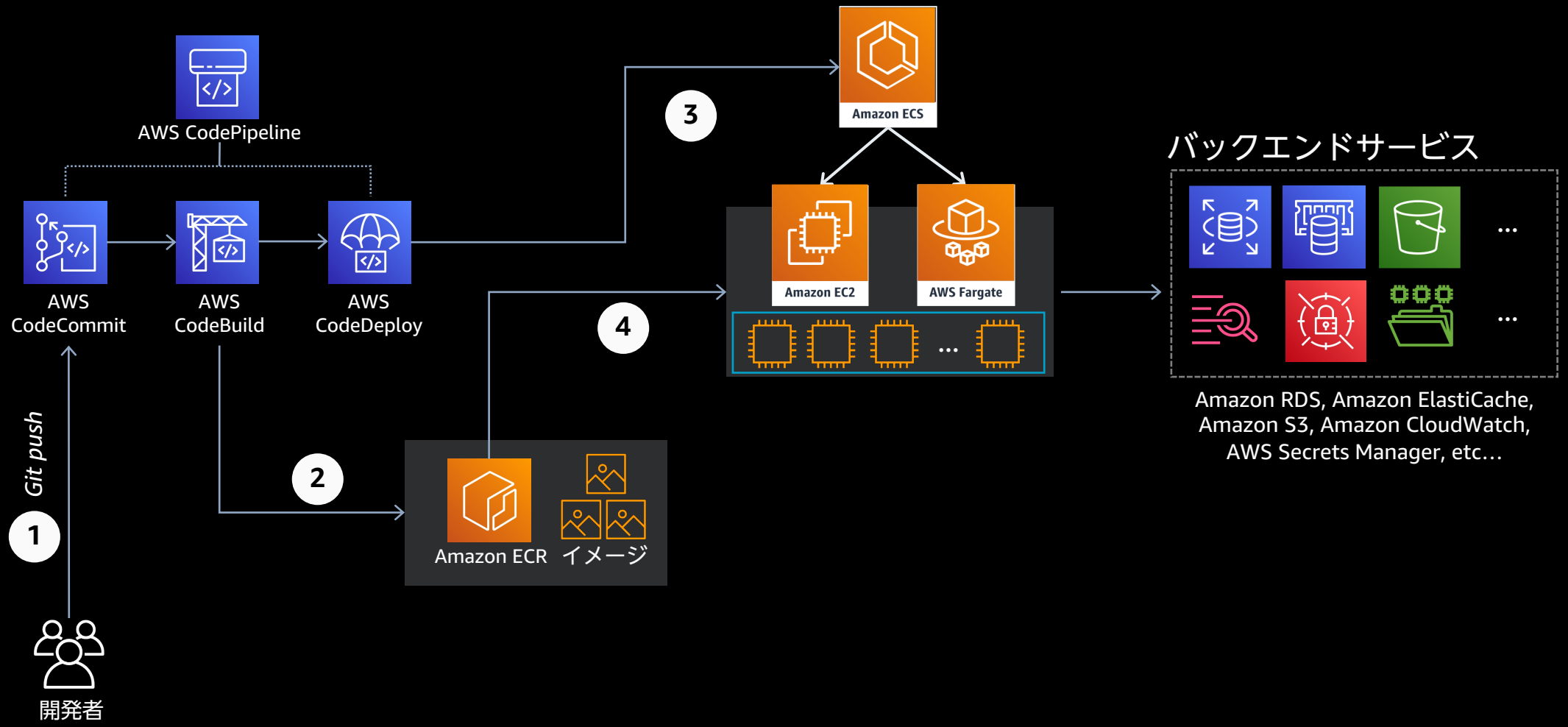
- ネットワーク：
  ローカル環境では、コンテナ間の通信は localhost 経由になるが、
  AWS 環境では VPC 内の IP 通信になる

- 環境変数：
  ローカル環境では、environment に記載するが、
  AWS 環境では SSM Parameter Store, AWS Secrets Manager を利用可能

- volumes：
  ローカル環境では、MySQL, Redis などを動かす為 volumes を使っているが、
  AWS 環境ではマネージドサービスの Amazon RDS, Amazon ElastiCache が使える
  （マネージドサービスのエンドポイントを環境変数で渡す）
  アプリケーションがストレージを必要とする場合、Amazon EBS、Amazon EFS を利用する

さらなるチャレンジと学習

コンテナ & CI/CD サービスの連携



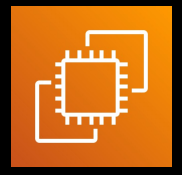
AWS の提供するコンテナ関連サービス


Amazon ECS


AWS App Runner


Amazon EKS


Amazon ECR


Amazon EC2


AWS Fargate


Amazon EKS Anywhere


Amazon EKS Distro

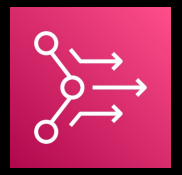

Red Hat OpenShift


Copilot CLI


Amazon ECS Anywhere


AWS Batch


AWS App Mesh


AWS Proton


Amazon CloudWatch Container Insights

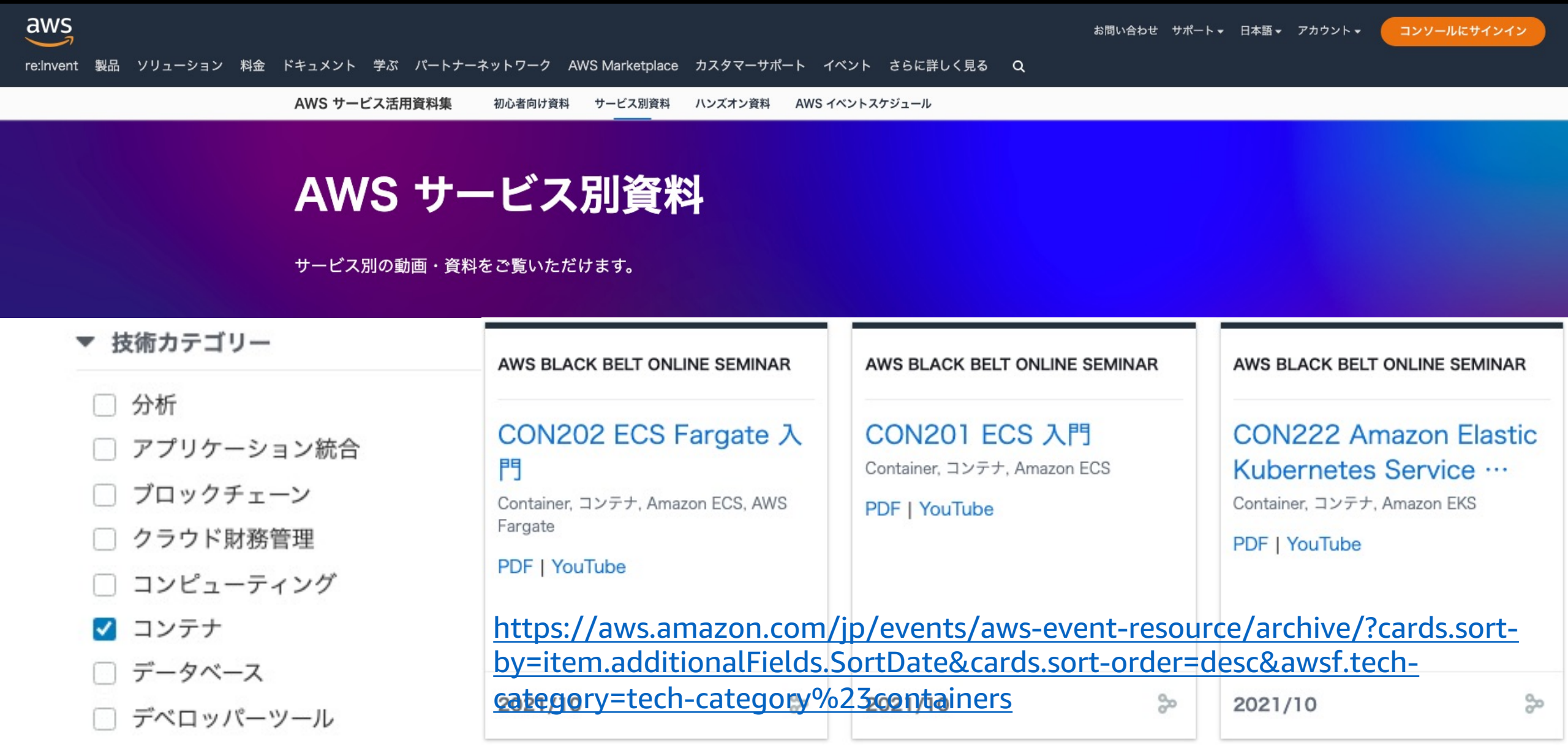
【ハンズオン】
コンテナ化のための
リファクタリング / リアーキテクティング
ワークショップ

https://catalog.us-east-1.prod.workshops.aws/workshops/a49e50ba-7473-4348-ba5d-6166385ad91d/ja-JP

【ハンズオン】
Copilot Primer Workshop

https://catalog.us-east-1.prod.workshops.aws/workshops/d03316be-3c29-49db-8dc3-eb196c1778c9/ja-JP

AWS サービス別資料 – コンテナ



https://aws.amazon.com/jp/events/aws-event-resource/archive/?cards.sort-by=item.additionalFields.SortDate&cards.sort-order=desc&awsf.tech-category=tech-category%23containers

まとめ

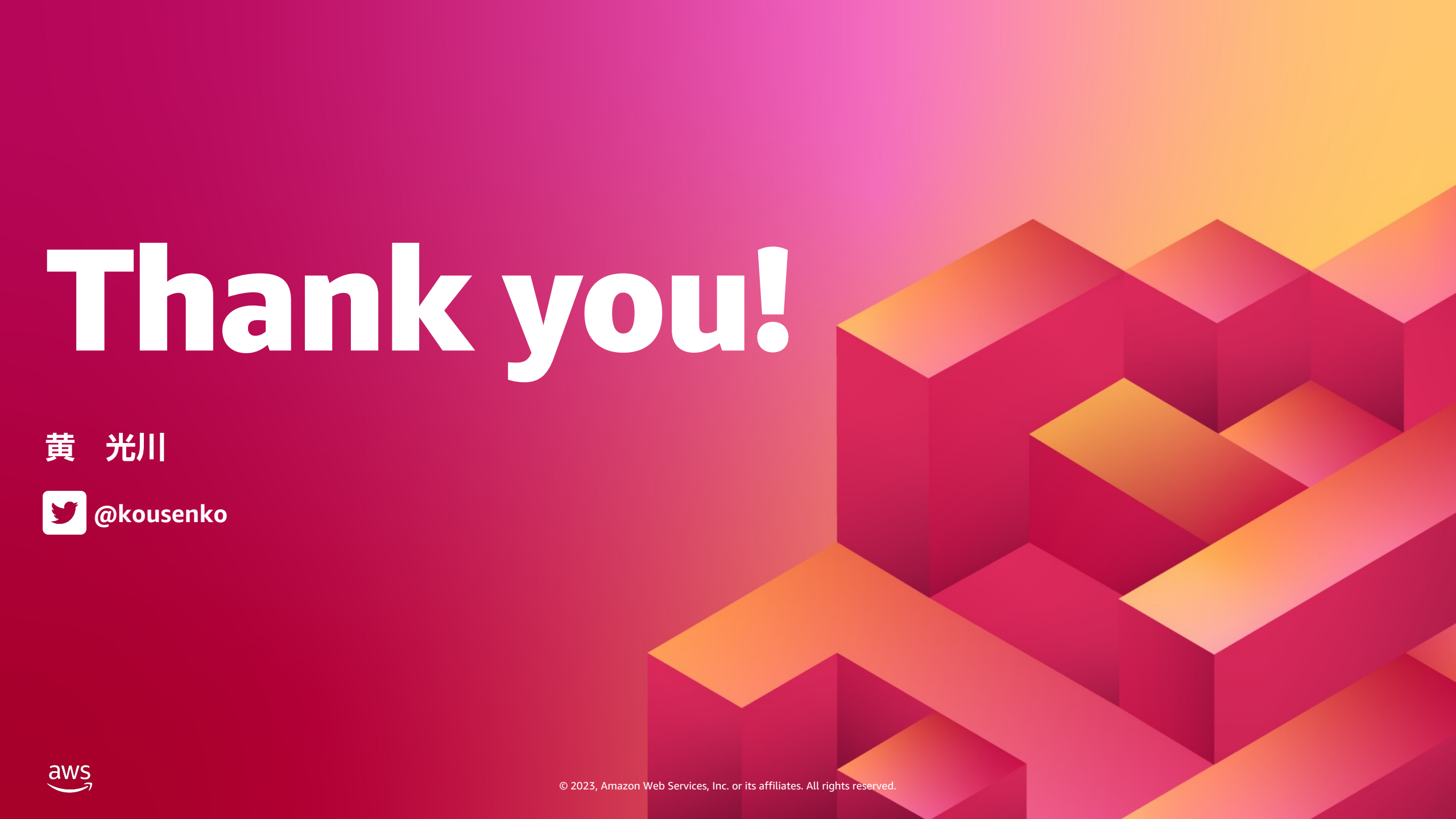
まとめ

- アプリケーションはステートレスにすることで、柔軟性が向上し耐障害性、拡張性が向上する
- コンテナはステートレスと相性が良い
  - ステートはコンテナの外に出す
  - ログはストリームとして標準出力・標準エラー出力に出し、ログドライバーで受取り処理する
  - 環境変数を利用してアプリケーションに変更を与える。機密情報はコンテナの外に保持する
- アプリケーションのコンテナ化は Dockerfile から
- コンテナのデプロイ
  - コンテナオーケストレーションサービスである Amazon ECS を利用する
  - Amazon ECS へのデプロイは CLI ツールの AWS Copilot CLI も使える
- ローカル環境でのコンテナ開発
- さらなるチャレンジと学習

Thank you!

黄　光川

@kousenko



AWS TRAINING & CERTIFICATION

AWS Skill Builder の 500+ の無料デジタルコースで学ぼう

30以上のAWSソリューションの中から、自分に最も関係のあるクラウドスキルとサービスにフォーカスし、自習用のデジタル学習プランとRamp-Upガイドで学ぶことができます。

- 自分のペースでAWSクラウド上を活用した未来を切り開く
- 学習プランでスキルや知識を向上
- AWS認定資格でクラウドの専門知識を証明する

自分に合ったスキルアップ方法をで学びましょう

EXPLORE.SKILLBUILDER.AWS »

aws

AWS Builders Online Series に
ご参加いただきありがとうございます

楽しんでいただけましたか? ぜひアンケートにご協力ください。
本日のイベントに関するご意見/ご感想や今後のイベントについての
ご希望や改善のご提案などがございましたら、ぜひお聞かせください。


aws-apj-marketing@amazon.com


twitter.com/awscloud_jp


facebook.com/600986860012140


https://www.youtube.com/user/AmazonWebServicesJP


https://www.linkedin.com/showcase/aws-careers/


twitch.tv/aws