AWS-48

クラウドにリフトしたアプリケーションをコンテナ化するためのアプローチ

竹本 将気
技術統括本部 ソリューションアーキテクト
アマゾン ウェブ サービス ジャパン合同会社

自己紹介

竹本　将気(タケモト マサキ)

所属

- ソリューションアーキテクト
- いわゆる "Web 系" のお客様を担当

好きな AWS サービス

- AWS Support
- コンテナ系サービス

Amazon ECS

AWS Fargate

バックグラウンド

- SIer
- Web系企業でインフラエンジニア
- スタートアップでインフラエンジニア

本セッションについて

想定聴講者

- コンテナというテクノロジーやAWSのコンテナサービスについて理解がある方
- EC2で運用しているアプリケーションをコンテナ化したいが、実のところ何をすればいいのか？といった疑問をお持ちの方

アジェンダ

- なぜコンテナなのか？
- コンテナ化のアプローチ
- まとめ

なぜコンテナなのか？

企業は迅速なイノベーションを求めている

迅速なイノベーションが求められている

- 変化への迅速な対応
- 市場投入までの時間を短縮する

しかし今日のアプリケーションは大きく異なっている

- モジュラーアーキテクチャパターン
- サーバーレス運用モデル
- アジャイル開発プロセス

そして、モダンなアプリケーションを構築するのは難しい

- 数百万人規模のユーザーへの拡張性
- グローバルな可用性
- ミリ秒単位の応答
- ペタバイト級のデータ処理

アプリケーション基盤の再検討が必要

カスタマーエクスペリエンスの向上

ペタバイトクラスのデータをハンドル

ビジネスの俊敏性を高める

ROIの向上とTCOの削減

コンテナによるモダナイゼーション

なぜコンテナなのか？

- 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可能(可搬性の高さ)

- 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
  - ✓ オペレーションの効率化
  - ✓ スケーラビリティ
  - ✓ 高速な開発サイクル

どのようにコンテナ化していけば良いでしょうか？

コンテナ化、マネージドサービス活用を進めていきたいがノウハウが少ない...

既存のEC2アプリケーションからどのようなアプローチでコンテナ化を進めれば良いか分からない...

コンテナ化のアプローチ

コンテナ化するアプローチの例

1. コンテナ化の目的を整理する
2. 移行方法を検討する
3. コンテナを設計する
4. コンテナイメージを作成する
5. AWS のコンテナサービスを検討する

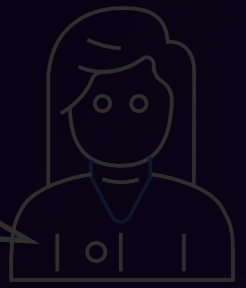
1. コンテナ化の目的を整理する

どのようにコンテナ化していけば良いでしょうか？

コンテナ化、マネージドサービス活用を進めていきたいがノウハウが少ない…

そもそも何を達成するために
コンテナ化したいのでしょうか？

既存のEC2アプリケーションからどのようなアプローチでコンテナ化を進めれば良いか分からない…



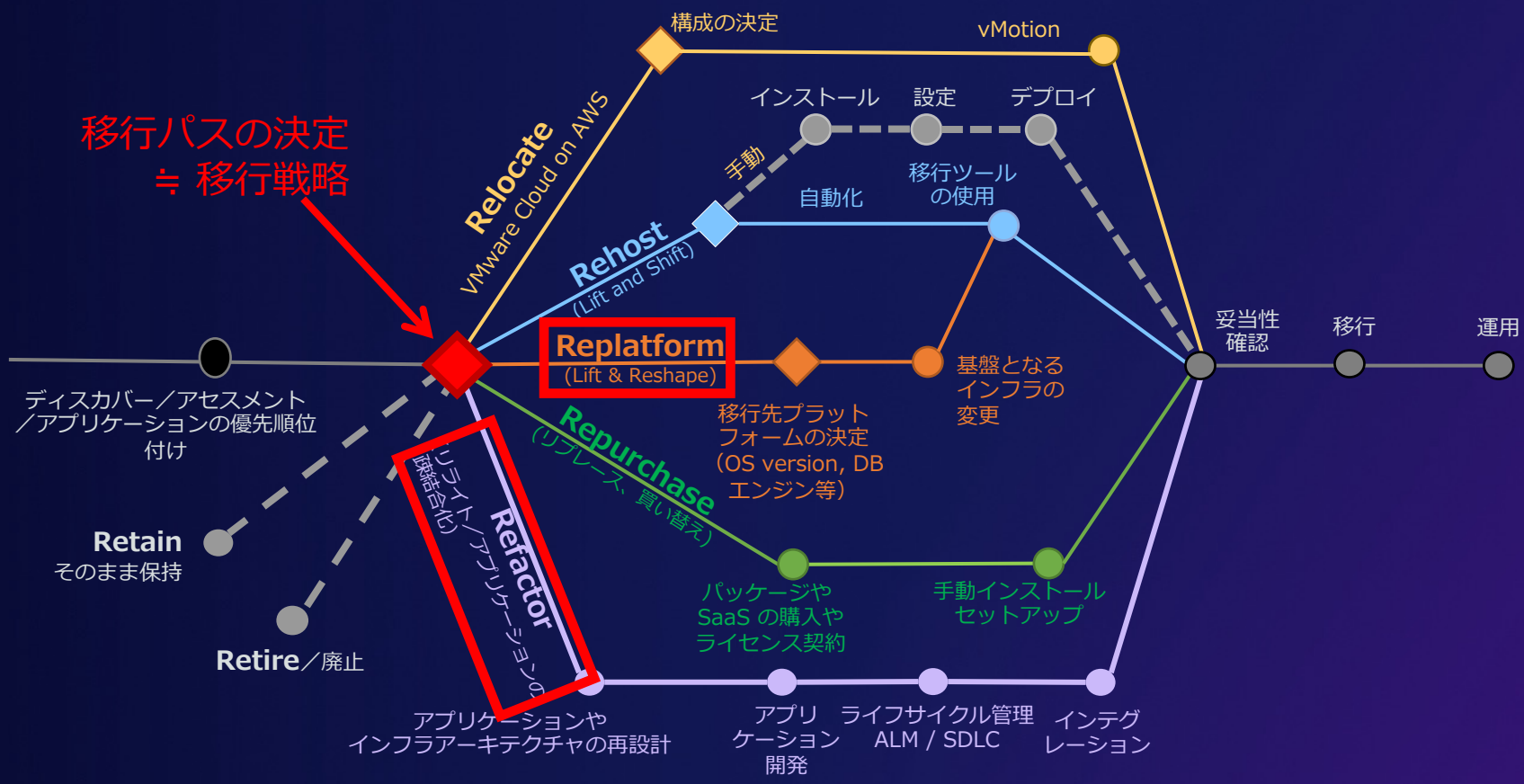
コンテナ化することで何を実現したいのか整理

- オペレーションの効率化
  - Amazon EC2 で行っている運用業務を AWS Fargate 等のマネージドサービスを活用することで運用コストを XX% 下げたい
- スケーラビリティ
  - スケーリングに必要なオペレーションを自動化したい
  - スケーリングに必要な物理時間を XX% 減らしたい
- 高速な開発サイクル
  - 現状 1日かかっている新機能のリリースを 例えば2時間に短縮したい
  - CI/CD パイプラインの整備し、手作業によるデプロイをなくしたい
  - 継続的な開発・運用が見込める体制を整えたい
  - 自動化文化を根付かせたい

2. 移行方法を検討する

マイグレーションパターン（7R）

AWS の考えるマイグレーション戦略には 7 つの R がある
このうちコンテナ化に関連するのは **Replatform** と **Refactor**



6 Strategies for Migrating Applications to the Cloud
https://aws.amazon.com/jp/blogs/enterprise-strategy/6-strategies-for-migrating-applications-to-the-cloud/
Six Common Strategies for Migration: "The 6 R's"
https://d1.awsstatic.com/Migration/migrating-to-aws-ebook.pdf

マイグレーションパターン（7R）

ラベル	移行 パターン名	移行パターンの説明	例
	**Retain （保持）**	• お客様はサーバーとアプリケーションをオンプレミス環境に残す • 移行の対象ではない	• 解決できない物理的な依存性 • Mainframe/AS400 • X86ではないUnixアプリケーション
	**Retire （リタイア）**	• オンプレミス環境でサーバーやアプリケーションを廃止する • 移行の対象ではない	• 既存の廃止プログラムの範囲に含まれるもの • DR目的でクラスター化されたホスト • 代替HAホスト
	**Relocate （リロケート）**	• アプリケーション変更不要の移行 • VMware 仮想環境のクラウド拡張/移行	• VMware Cloud on AWS
	**Rehost （ホスト変更）**	• サーバーをそのままオンプレ環境からクラウドに移行する • クラウドで稼働させるためには最小限の変更が必要になる場合がある	• AWS Server Migration Service / AWS Application Migration Service を使用したオンプレミスの仮想サーバーの EC2 インスタンスへの移行
	**Repurchase （買い替え）**	• アプリケーションを SaaS や新しいパッケージアプリケーションに置き換える	• CRM to Salesforce.com
	**Replatform （プラットフォーム変更）**	• **アプリケーションをコンテナ化してクラウドへ移行する** • **アプリケーションの変更が必要になる場合がある** • **データベースをマネージドサービスへ移行する** • **OS やデータベースを変更する**	• **アプリケーションのコンテナ化して ECS / EKS へ移行** • **Amazon RDS / Amazon Aurora への移行** • **AWS App2Container, AWS Elastic Beanstalk**
	**Refactor （リファクタリング）**	• **アプリケーションアーキテクチャーを再構築して最適化する** • **モノリスからマイクロサービスへ移行する**	• **マイクロサービス** • **サーバーレス** • **Amazon API Gateway / AWSLambda / Amazon DynamoDB**

Replatform によるコンテナ化

- コンテナ化によるメリットが享受できる場合
  - オペレーションの効率化
  - スケーラビリティ(Auto Scaling)
  - 高速なデプロイサイクルを回す(Try & Errorの回数を増やす)
- 単体(モノリス)なコンテナでシンプルなアプリケーションを構築したい場合
  - AWS App2Container 等の活用
- コンテナ化による恩恵が移行コストを上回ると予測できる場合
  - コンテナ化に多くのステップがある

コンテナ化に必要な多くのステップ

- アプリケーションを理解する
  - サードパーティーのライブラリと依存関係、ネットワークへの依存
- コンテナの作成
  - Dockerfile の作成、コンテナのビルドとアップロード
- CI/CD プロセスの整備
  - AWS CodePipeline 等を使用した CI／CD パイプラインの設計、実装
- コンテナをデプロイするインフラストラクチャーの整備
  - Amazon ECS, Amazon EKS, Red Hat OpenShift Service on AWS など
  - Amazon VPC、セキュリティグループ、オートスケーリング、ロードバランサーなど

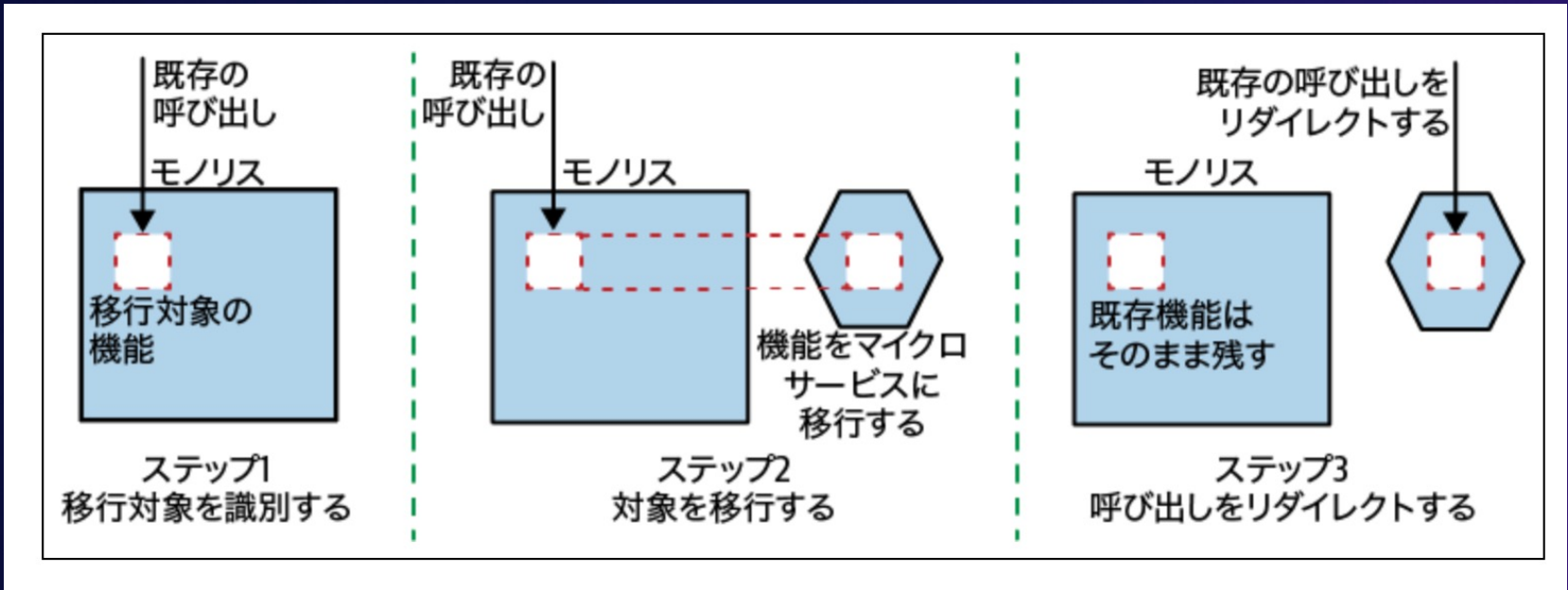
必要な 移行コスト と コンテナ化のメリット を比較する

Refactoring によるコンテナ化

- 現状のモノリスアプリケーションで課題がある場合
  - 変更による影響範囲が広い
  - 非効率なスケーリング
  - テスト・ビルドに要する時間が長い...等
- モノリシックかつ巨大なレガシーアプリケーションで、Replatform によるコンテナ化が難しい場合
  - リファクタリングしてからじゃないとコンテナ化できない場合
- 移行戦略としてストラングラーパターンなどがある
  - アプリケーションから機能を切り出し、徐々にコンテナ化する

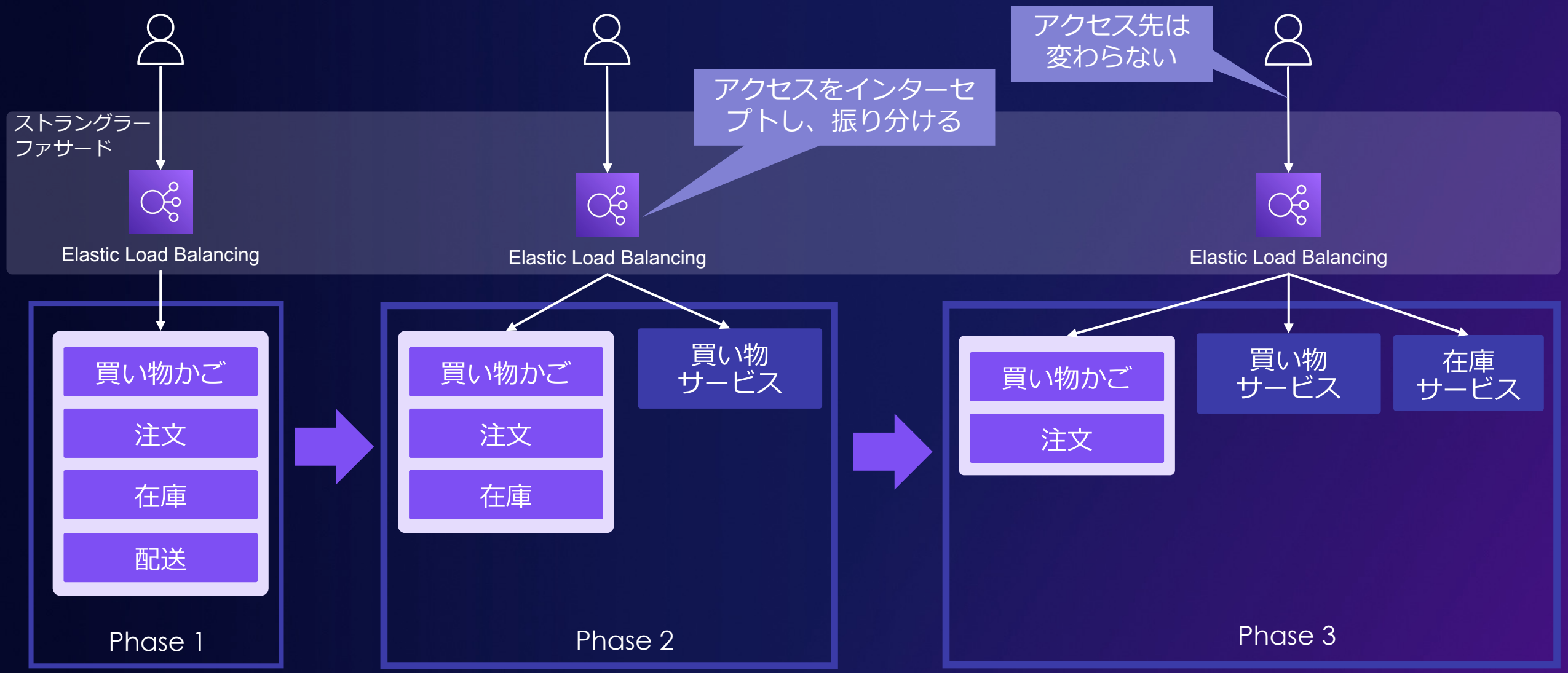
ストラングラーパターン

- モノリスから段階的にマイクロサービスへ分割していく手法
- 最も一般的に使われているマイクロサービスへの移行テクニック
- 2004 年に Martin Fowler が旅行先の熱帯雨林でつる植物（Strangler Fig）がじわじわと成長して自身の寄生した木を徐々に絞めていき、本体を乗っ取るさまからヒントを得て命名
  - https://bliki-ja.github.io/StranglerApplication/



書籍「モノリスからマイクロサービスへ」（Sam Newman 著、島田 浩二 訳）より抜粋
ISBN978-4-87311-931-1

ストラングラーパターンの例



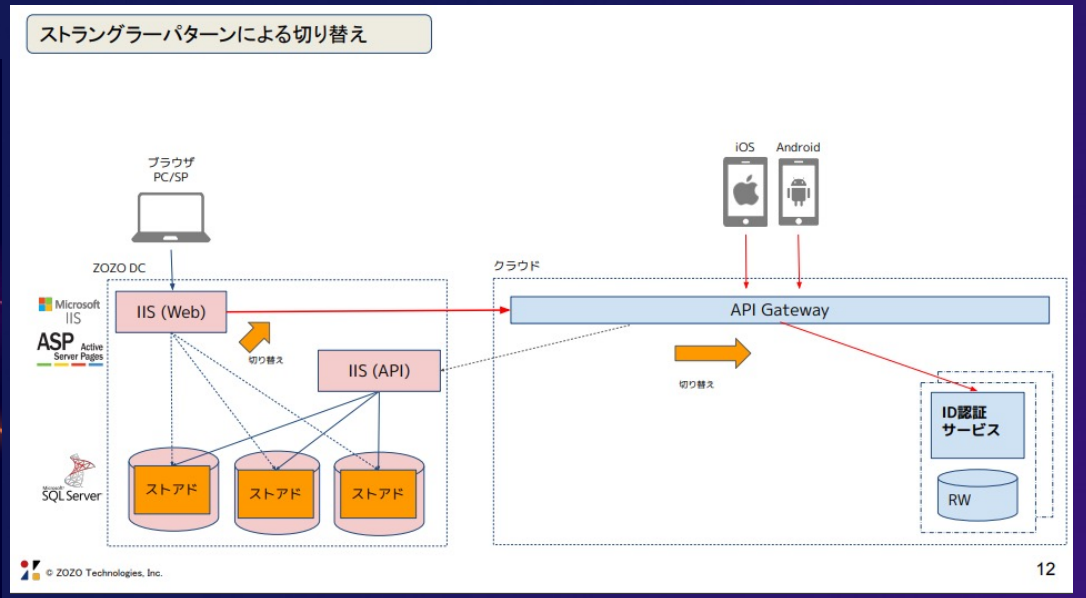
ストラングラーパターンの事例

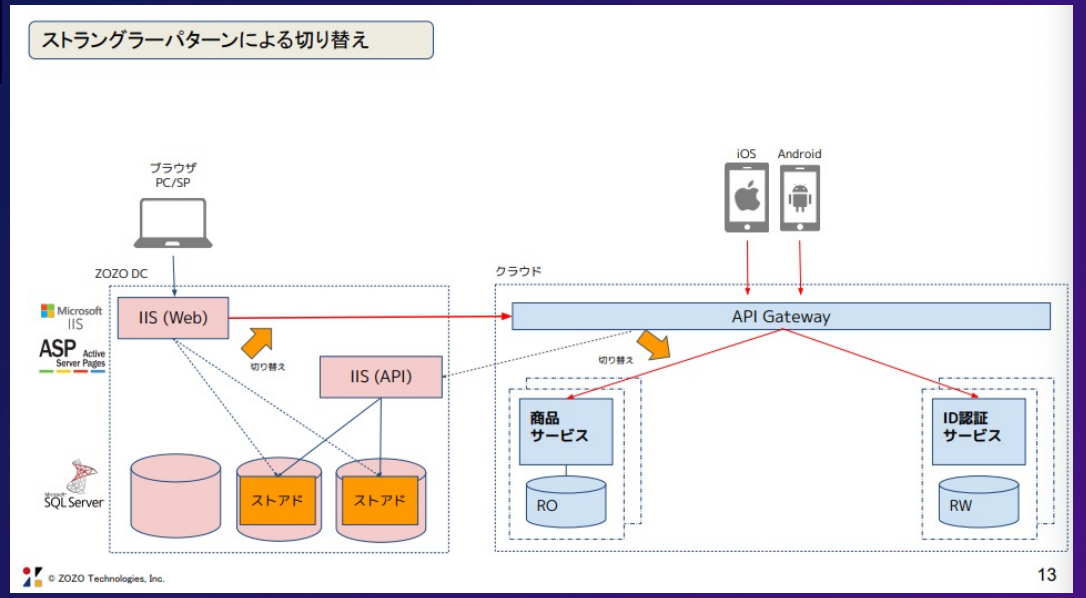
CUS-01

ZOZOTOWNにおける
Amazon EKSを中心に据えた
マイクロサービスアーキテクチャへの変遷

瀬尾 直利
株式会社ZOZOテクノロジーズ
SRE部 リーダー

https://d1.awsstatic.com/events/jp/2021/summit-online/CUS-01_AWS_Summit_Online_2021_ZOZO_Technologies.pdf





3. コンテナを設計する

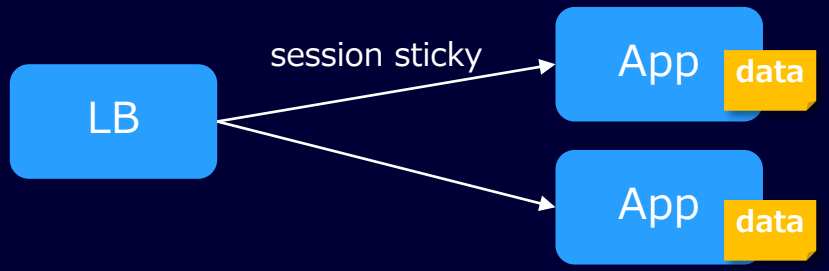
コンテナの可搬性を高めるために大事なこと

- エフェメラルコンテナを作成する
- ホストマシンへ依存しない
- コンテナに設定情報を含まない
- コンテナごとに 1 つのプロセスを実行
- シグナルハンドリングによる安全な停止
- ログは標準出力・標準エラー出力へ

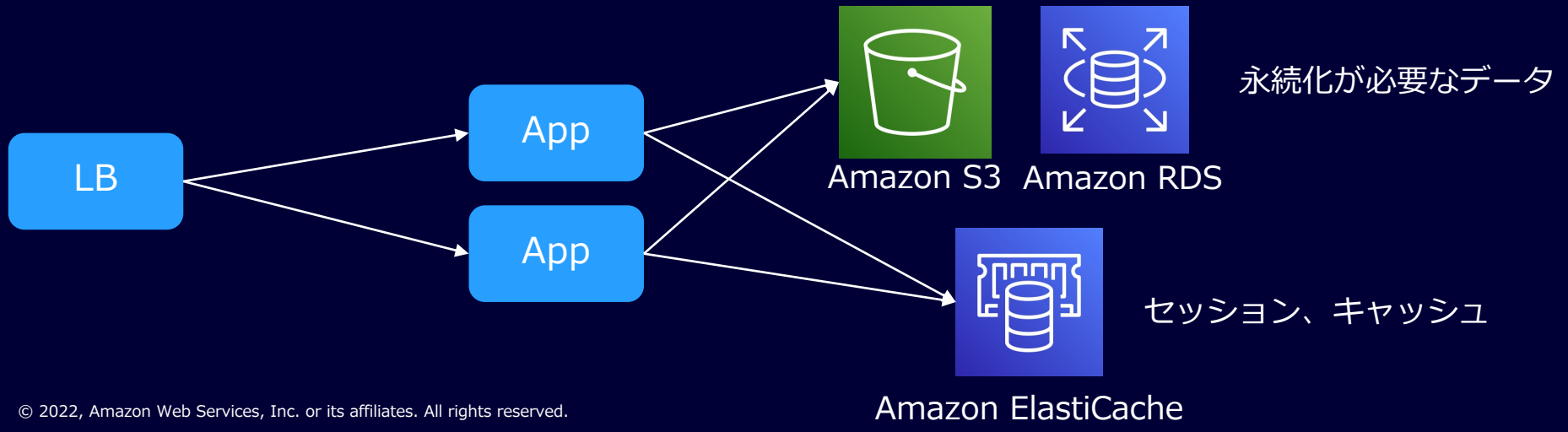
エフェメラルコンテナを作成する

アプリケーションを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
- スティッキーセッションを利用する



- 永続化が必要なデータを Amazon S3 や Amazon RDS に保管する
- セッション情報やキャッシュは Amazon DynamoDB や Amazon ElastiCashe に保管する



シグナルハンドリングによる安全な停止

- デプロイやスケーリングなど、コンテナは頻繁に再作成される
- SIGTERM 受信時 (コンテナ終了時) の graceful shutdown や checkpoint の書き出しなど、アプリケーション側でシグナルハンドリングを行う

```python
import signal, time, os

def shutdown(signum, frame):
    print('Caught SIGTERM, shutting down')
    # Finish any outstanding requests, then...
    exit(0)

if __name__ == '__main__':
    # Register handler
    signal.signal(signal.SIGTERM, shutdown)
    # Main logic goes here
```

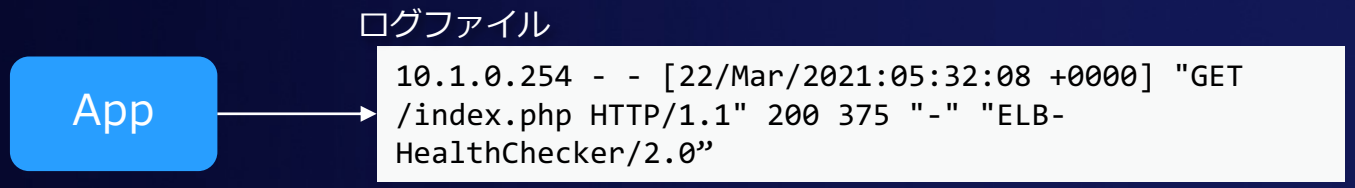
```javascript
process.on('SIGTERM', () => {
  console.log('The service is about to shut down!');

  // Finish any outstanding requests, then...
  process.exit(0);
});
```

ログは標準出力・標準エラー出力へ

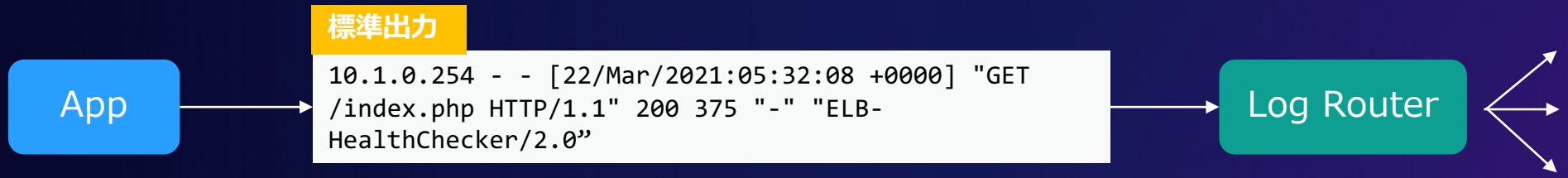
ログをイベントストリームとして扱う

- 🤦‍♂️ 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**

- 🤦 ログは標準出力/標準エラーに出力
- 🤦 アプリケーション側でログ管理は行わない



ログルーター(Amazon CloudWatch Agent, Fluent Bit など）が様々な保存先にログを転送

ログを標準出力・標準エラーに出力する例

Nginx のログを標準出力/標準エラーに出力

🤦 Nginx のログをファイルに出力

```
# nginx.conf
http {
    ...
    access_log /var/log/nginx/access.log;
    error_log /var/log/nginx/example-error.log info;
    ...
```

🤷 Nginx のログを標準出力するよう設定

```
# nginx.conf
http {
    ...
    access_log /dev/stdout;
    error_log /dev/stderr info;
    ...
```

タスク定義でログを Amazon CloudWatch logs に転送

```
# Task Definition
...
   "logConfiguration": {
      "logDriver": "awslogs",
      "options": {
         "awslogs-group": "awslogs-nginx",
         "awslogs-region": "ap-northeast-1",
         "awslogs-stream-prefix": "awslogs-example"
      }
   },
...
```

まとめ：コンテナの可搬性を高めるために大事なこと

- エフェメラルコンテナを作成する
  - 永続化が必要なデータは、ステートフルなバックエンドサービス(Amazon RDS 等)に格納する
- ホストマシンへ依存しない
  - サイドカーコンテナ等を活用し、特定ホストに依存しない構成にする
- コンテナに設定情報を含まない
  - AWS Systems Manager Parameter Store を活用し、コンテナの外部から設定情報を注入可能にしておく
- コンテナごとに 1 つのプロセスを実行
  - 一つのコンテナに複数の役割を持たせないようにコンテナを設計する
- シグナルハンドリングによる安全な停止
  - アプリケーション側で SIGTERM を適切に処理できるようにシグナル処理を行う
- ログは標準出力・標準エラー出力へ
  - コンテナランタイムのログ収集の仕組みを利用する、サイドカーコンテナとして Flunet Bit を利用する

4. コンテナイメージを作成する

コンテナイメージ作成方法

1. Dockerfile を使用する
2. AWS App2Container を使用する
3. Cloud Native Buildpacks(CNB) を使用する

コンテナイメージ作成方法

1. Dockerfile を使用する
2. AWS App2Container を使用する
3. Cloud Native Buildpacks(CNB) を使用する

Dockerfile を使用する

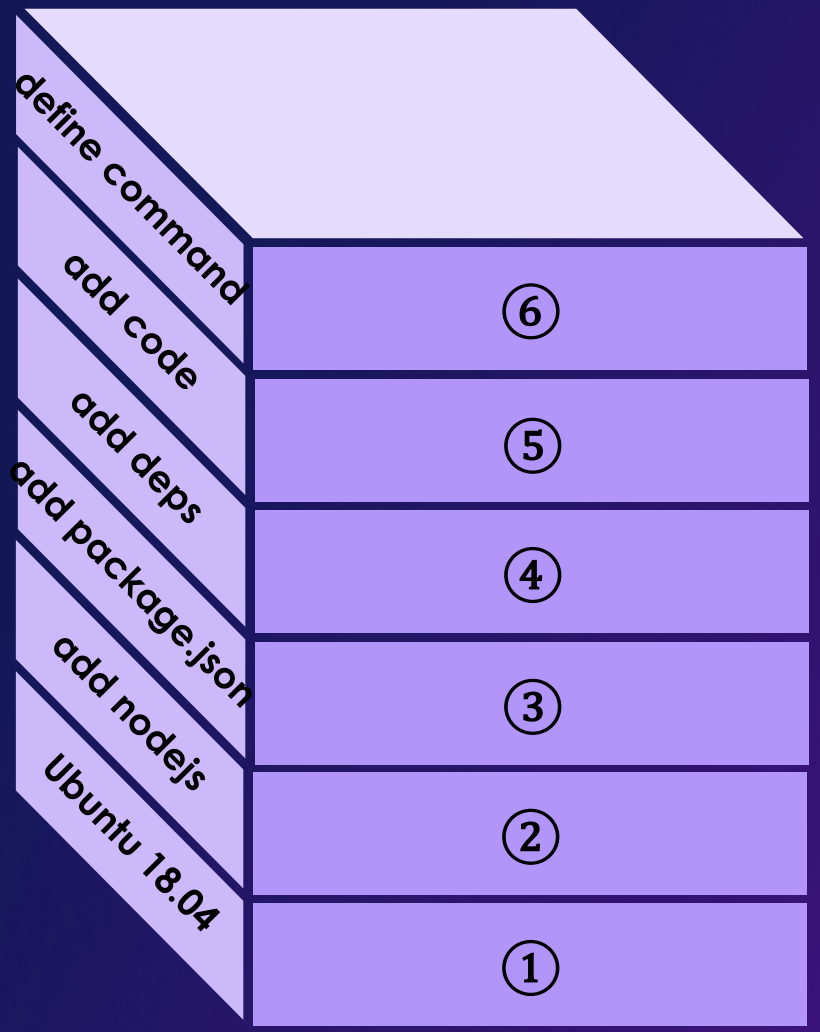
Dockerfile の例

```
FROM ubuntu:18.04                                  --- ①

RUN apt-get update && ¥                            --- ②
    apt-get install -y nodejs

COPY ./package.json .                              --- ③
COPY ./node_modules ./node_modules                 --- ④
COPY ./index.js .                                  --- ⑤

CMD ["/usr/bin/node", "index.js"]                  --- ⑥
```



Linterを使用する

hadolint

ベストプラクティス沿ったDockerイメージを構築するのに役立つ Dockerfile の linter

```
# Macにおける hadolint のインストール
$ brew install hadolint

# hadolintの実行
$ hadolint ${Dockerfileのpath}
Dockerfile:3 DL3008 warning: Pin versions in apt get install. Instead of `apt-get install <package>` use `apt-get install <package>=<version>`
Dockerfile:3 DL3009 info: Delete the apt-get lists after installing something
Dockerfile:3 DL3015 info: Avoid additional packages by specifying `--no-install-recommends`
Dockerfile:6 DL3045 warning: `COPY` to a relative destination without `WORKDIR` set.
Dockerfile:7 DL3045 warning: `COPY` to a relative destination without `WORKDIR` set.
Dockerfile:8 DL3045 warning: `COPY` to a relative destination without `WORKDIR` set.
```

解析ルール一覧及び詳細は、以下のページに記載
https://github.com/hadolint/hadolint#rules.

Linter を使用する

hadolint の例

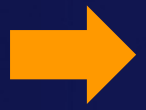
Dockerfile (before)

```
FROM ubuntu:18.04

RUN apt-get update && ¥
    apt-get install -y nodejs

COPY ./package.json .
COPY ./node_modules ./node_modules
COPY ./index.js .

CMD ["/usr/bin/node", "index.js"]
```



Dockerfile (after)

```
FROM ubuntu:18.04

RUN apt-get update && ¥
  apt-get install -y --no-install-recommends nodejs=17.0.0 && ¥
  apt-get clean && ¥
  rm -rf /var/lib/apt/lists/*

WORKDIR /
COPY ./package.json .
COPY ./node_modules ./node_modules
COPY ./index.js .

CMD ["/usr/bin/node", "index.js"]
```

コンテナイメージ作成方法

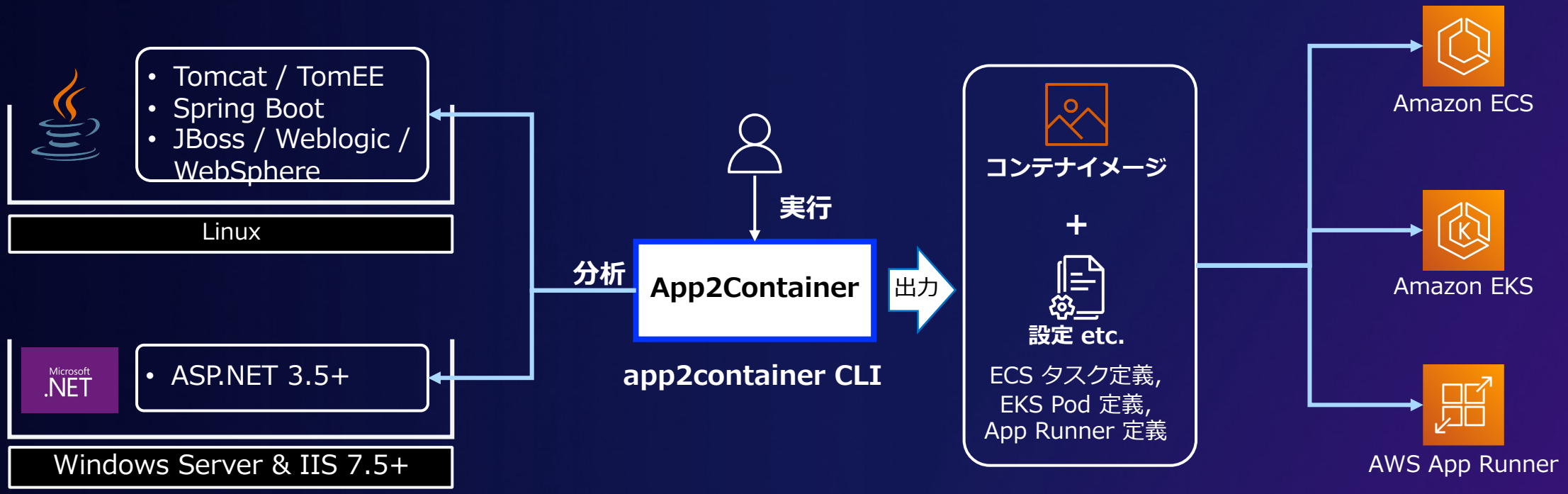
1. Dockerfile を使用する
2. AWS App2Container を使用する
3. Cloud Native Buildpacks(CNB) を使用する

レガシーアプリケーションのコンテナ化の課題

- 手動またはカスタムスクリプトベースのビルドとデプロイ作業
- レガシーアプリケーションに関する知識の不足
- コンテナ/クラウド技術に関する専門知識の欠如
- 日々の業務に追われている
  - コンテナ化の優先順位が上がらない(ビジネス優先)
  - いつかはやりたいと思っているが、機能開発や運用業務に追われている

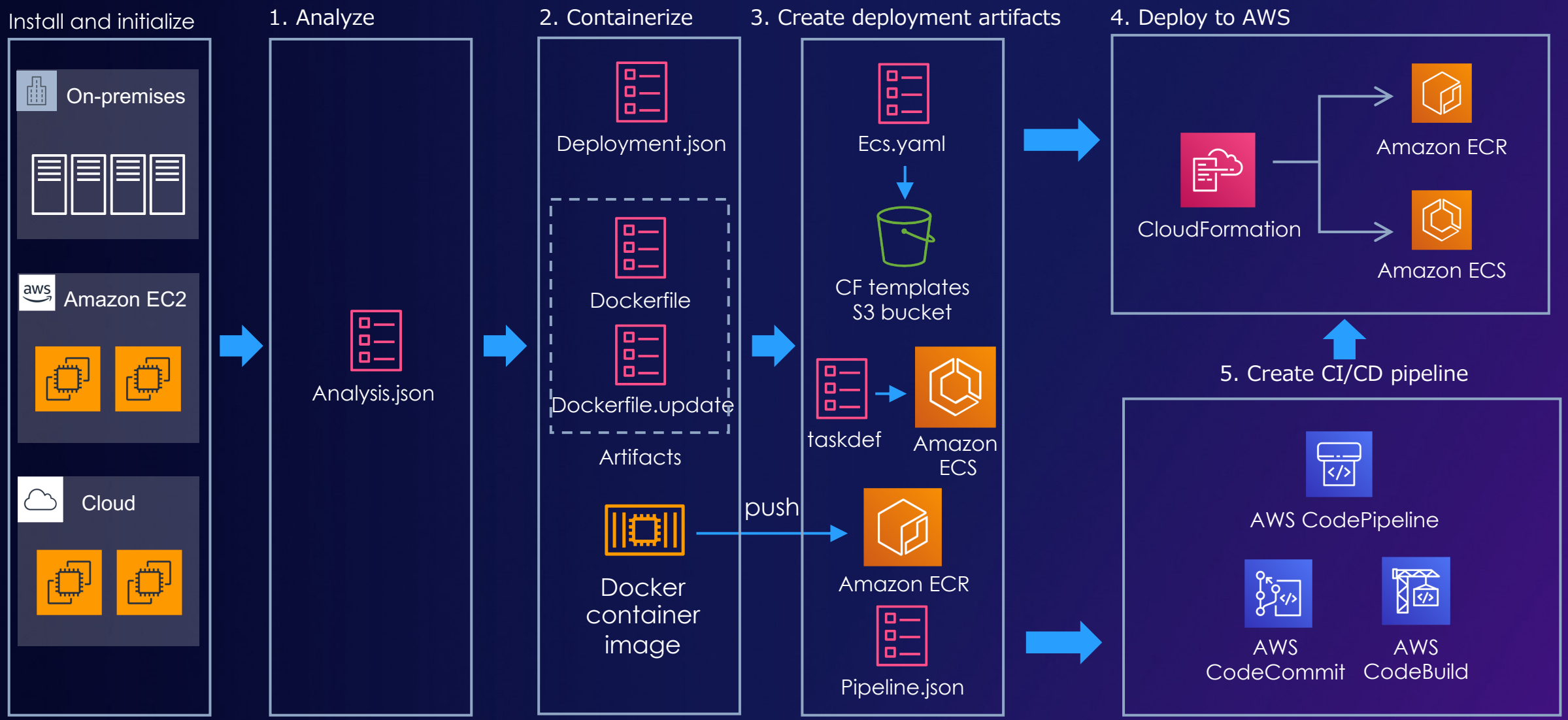
AWS App2Container (A2C) とは

Java および ASP.NET アプリケーションのコンテナ化を実行する CLI ツール (無料で利用可能)

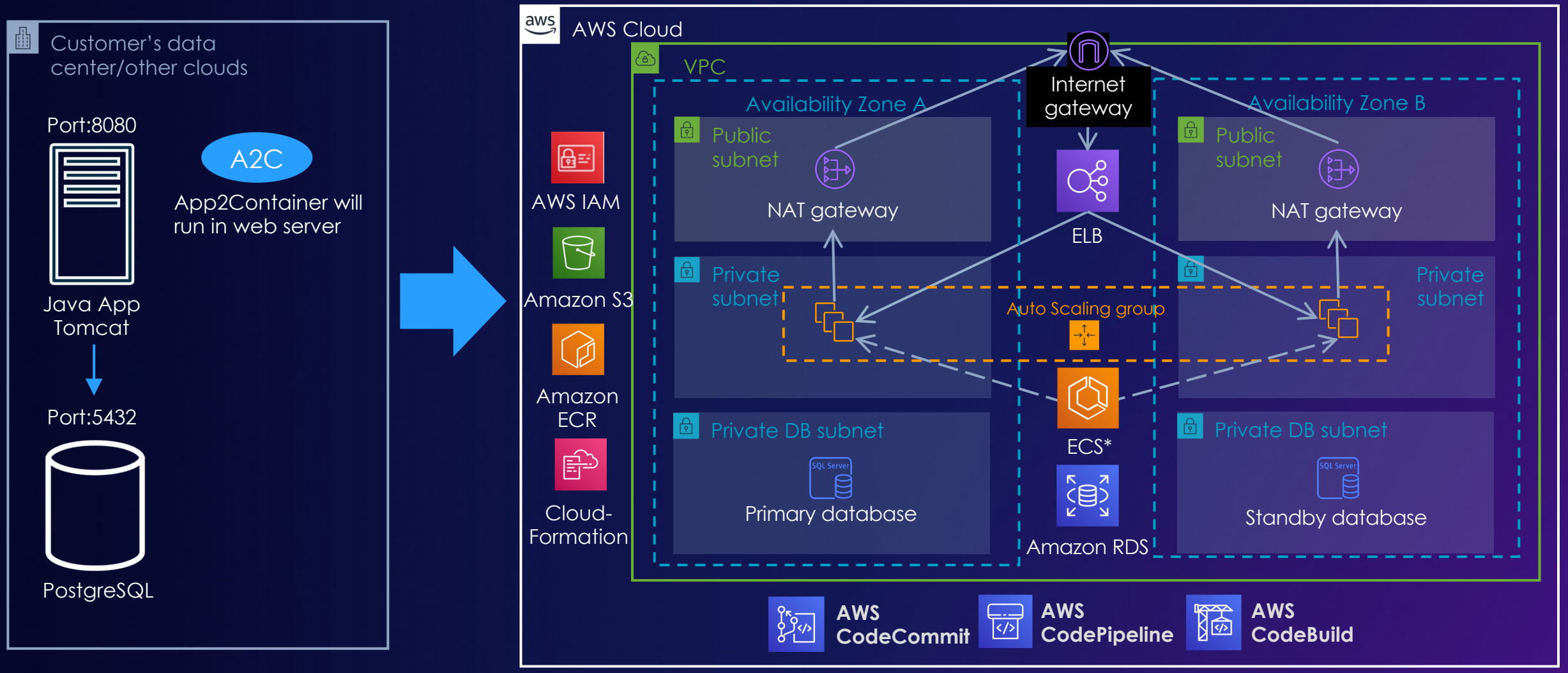


https://docs.aws.amazon.com/app2container/latest/UserGuide/what-is-a2c.html

Replatforming with App2Container



Modernize with AWS App2Container Workshop



*Deployment target will be similar for Amazon ECS, Amazon EKS, or AWS App Runner

https://catalog.us-east-1.prod.workshops.aws/workshops/2c1e5f50-0ebe-4c02-a957-8a71ba1e8c89/en-US/

コンテナイメージ作成方法

1. Dockerfile を使用する
2. AWS App2Container を使用する
3. Cloud Native Buildpacks (CNB) を使用する

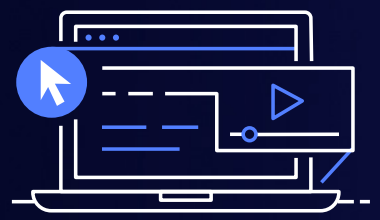
Cloud Native Buildpacks(CNB)によるビルド

- Dockerfile の代替ツールの 1 つ
- **Dockerfile を使わずに** アプリケーションのソースコードを OCI 準拠のコンテナイメージに変換する仕組みを提供
- ソースコードに応じたベストプラクティスに沿ったビルド
- 非 root ユーザーで実行され、最小限のパッケージがインストールされセキュアにコンテナイメージを作成
- Java、.NET Core、Ruby、Node.js、Go、Python など、さまざまなプラットフォームをサポートするオープンソースの Builder がある

5. AWS のコンテナサービスを検討する

AWSのコンテナサービスでアプリケーションのModernize を容易に

AWSのコンテナサービスは、オンプレミスまたはクラウドにかかわらず、基盤となるコンテナ・インフラストラクチャの管理を簡単にします。



DIVE DEEP

Container services and migration

- Amazon ECS workshop
- Amazon EKS workshop

REGISTRY

- Amazon ECR

ORCHESTRATION

- Amazon ECS
- Amazon EKS
- Red Hat OpenShift Service on AWS
- AWS App Runner

COMPUTE

- AWS Fargate
- Amazon EC2
- AWS App Runner

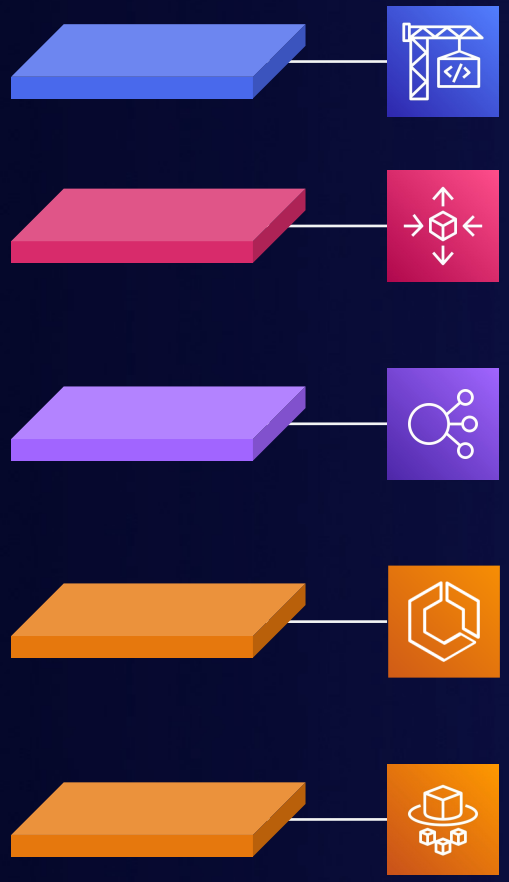
コンテナ化に必要な多くのステップ

- アプリケーションを理解する
  - サードパーティーのライブラリと依存関係、ネットワークへの依存
- コンテナの作成
  - Dockerfile の作成、コンテナのビルドとアップロード
- CI/CD パイプラインの整備
  - AWS CodePipeline 等を使用した CI／CD パイプラインの設計、実装
- コンテナをデプロイするインフラストラクチャーの整備
  - Amazon ECS, Amazon EKS, Red Hat OpenShift Service on AWS など
  - Amazon VPC、セキュリティグループ、オートスケーリング、ロードバランサーなど

考えること、やることがいっぱい…orz

必要な改修コスト と コンテナ化のメリット を比較する

クラウドインフラストラクチャには多様なサービスが存在



フロントエンドアプリケーション

バックエンドアプリケーション

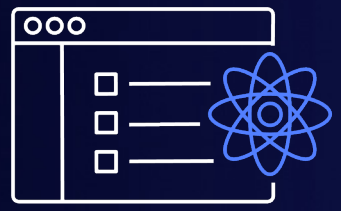
クラウドインフラストラクチャには多様なサービスが存在

AWS App Runner

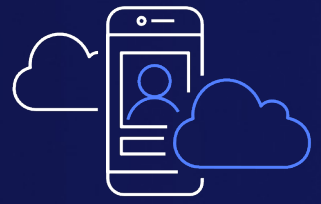
フロントエンドアプリケーション

バックエンドアプリケーション

ユースケース



モノリスアプリケーション



モバイルバックエンド

マイクロサービス / API

AWS App Runner のサービス仕様と他の選択肢

現時点(2022年5月現在)では以下のようなサービス仕様があります。

- App Runner では最大 2 vCPU / 4GB メモリ
- App Runner は HTTP2, WebSocket をサポートしていない
- App Runner で非同期処理が実行できない

またアプリケーションを要件にfitしない場合もあるかと思います。

- Application Load Balancer や CI/CD をカスタマイズしたい
- サイドカーとして特定エージェントを動かしたい
- プライベートなエンドポイントとして使用したい …等

AWS App Runner のサービス仕様と他の選択肢

現時点(2022年5月現在)では以下のようなサービス仕様があります。

- App Runner では最大 2 vCPU / 4GB メモリ
- App Runner は HTTP2, WebSocket をサポートしていない
- App Runner で非同期処理が実行できない

また、アプリケーションを要件にfitしない場合は…と思います。

- Application Load Balancer や CI/CD をカスタマイズしたい
- サイドカーとして特定エージェントを動かしたい
- プライベートなエンドポイントとして使用したい …等

AWS App Runner が要件にフィットしなければ…

ORCHESTRATION

Amazon Elastic Container Service (Amazon ECS)

Amazon Elastic Kubernetes Service (Amazon EKS)

Red Hat OpenShift Service on AWS (ROSA)

まとめ

- なぜコンテナなのか
  - コンテナはその可搬性の高さから、効率的なオペレーション、スケーラビリティ、高速な開発サイクルといった効果が期待できる

まとめ

コンテナ化のアプローチ

- コンテナ化の目的を整理する
  - なぜコンテナ化するのか、何を実現したのか？を考える
- 移行方法を検討する
  - Replatform or Refactoring
- コンテナを設計する
  - 可搬性を高めるための設計ためのポイント
- コンテナイメージを作成する
  - Dockerfile or App2Container or Cloud Native buildpacks
- AWSのコンテナサービスを検討する
  - コンテナのマネージドサービスである AWS App Runner

Thank you!

竹本　将気

技術統括本部 ソリューション
アーキテクト
Solutions Architect

関連セッション

AWS-05 リホストから始めるAWSへのサーバー移行

AWS-08 最も効果的なエンタープライズシステム・モダナイゼーションの進め方とは

Appendix

参考情報(App2Container)

- AWS ブログ
  - AWS App2Container – Java および ASP.NET アプリケーション用の新しいコンテナ化ツール
    - https://aws.amazon.com/jp/blogs/news/aws-app2container-a-new-containerizing-tool-for-java-and-asp-net-applications/
  - AWS App2Container を利用して Java および .NET アプリケーションをリモートでモダナイズする
    - https://aws.amazon.com/jp/blogs/news/modernize-java-and-net-applications-remotely-using-aws-app2container/
  - Accelerate modernization of your application using App2Container
    - https://aws.amazon.com/jp/blogs/containers/accelerate-modernization-of-your-application-using-app2container-containerizing-it-and-deploying-to-ecs-fargate/
- ワークショップ
  - https://app2container.workshop.aws/
- 動画
  - Exploring AWS App2Container - A tool for modernizing and containerizing your legacy apps
    - https://www.youtube.com/watch?v=8mB5wPnPaMw
  - AWS re:Invent 2020: Quickly containerize .NET & Java applications with AWS App2Container
    - https://www.youtube.com/watch?v=69S3AyBWIjo
  - AWS re:Invent 2020: Best practices for containerizing legacy applications
    - https://www.youtube.com/watch?v=8iVQ9Kugzl0
  - Application Modernization/Migration with AWS
    - https://www.youtube.com/watch?v=BhTyndp3cRA

参考情報(App Runner)

- App Runner RoadMap
  - https://github.com/aws/apprunner-roadmap/projects/1
- App Runner 機能要望
  - https://github.com/aws/apprunner-roadmap/issues
- [AWS Black Belt Online Seminar] CON243 App Runner 入門
  - https://www.youtube.com/watch?v=1-A5XlwM7Xg
- App Runner workshop
  - https://www.apprunnerworkshop.com/
- **AWS App Runner のご紹介**
  - https://aws.amazon.com/jp/blogs/news/introducing-aws-app-runner/

参考情報(CNB)

- CNBご紹介ブログ
  - **Cloud Native Buildpacks による AWS CodeBuild と AWS CodePipeline を使ったコンテナイメージの作成**

- **Workshop(英語のみ)**
  - MIGRATING WORKLOADS TO ECS