DEV DAY
TOKYO

2019.10.03-04

AWSサービスで実現する
バッチ実行環境のコンテナ/サーバレス化

永井 勝一郎
コネヒト株式会社

aws

自己紹介

永井 勝一郎

コネヒト株式会社　エンジニア

インフラ/ウェブオペレーション/機械学習(最近)

@shnagai

主な活動:

- JapanContainerDays v18.12, AWS Summit2018
- はじめてのPHPプロフェッショナル開発, 日経SYSTEMS(ママリが実践する　コンテナ移行の正攻法)
- コネヒトエンジニアブログ: https://tech.connehito.com/archive/author/nagais

mamari

About ママリ

ママリは、「ママの一歩を支える」ブランドです

ママリが約束すること

家族の舵取り役である ”ママ” を支えることを通して、家族がもっと幸せに過ごせる世の中をつくります。

「3つの切り口」でママを支えます

1. 自ら選ぶための「知識」を提供する

家族にとってより良い選択をするために必要な確かな情報に基づいた「知識」を提供します

2. 一歩を踏み出す「自信」を育む

知識とともに、一歩を踏み出す原動力となる「自信」を育むための場を提供します

3. 行動したママを受け入れる「社会」をつくる

ママの声を社会に発信し、ママの選択を受け入れる土壌のある社会をつくります

About ママリ

“悩み” と ”共感” を軸にママに寄り添い
アプリ・Web・SNSと多角的にサービスを展開しています

ママリ（アプリ・Web）

ママ同士で悩みを相談し合うQ&Aコミュニティを中心に
ユーザーを拡大しています

月間閲覧数 ※1

約 **1.5** 億回

月間利用者数 ※1

約 **650** 万人

ママ向けアプリ 第1位

ママ向けNo.１アプリに選出 ※2

1,084人のママが選ぶ「現在使っているアプリ」 にて、5項目（他のママにオススメしたい、認知度、
利用率、利便性、好感度）で１位を獲得しました

SNS（Instagram・ LINE・ Facebook）

「#ママリ」でユーザー同士が
どんどん繋がっています

累計ファン数 ※3

約 **91** 万人

記事

ママの生活に役立つ記事を
幅広いジャンルで配信しています

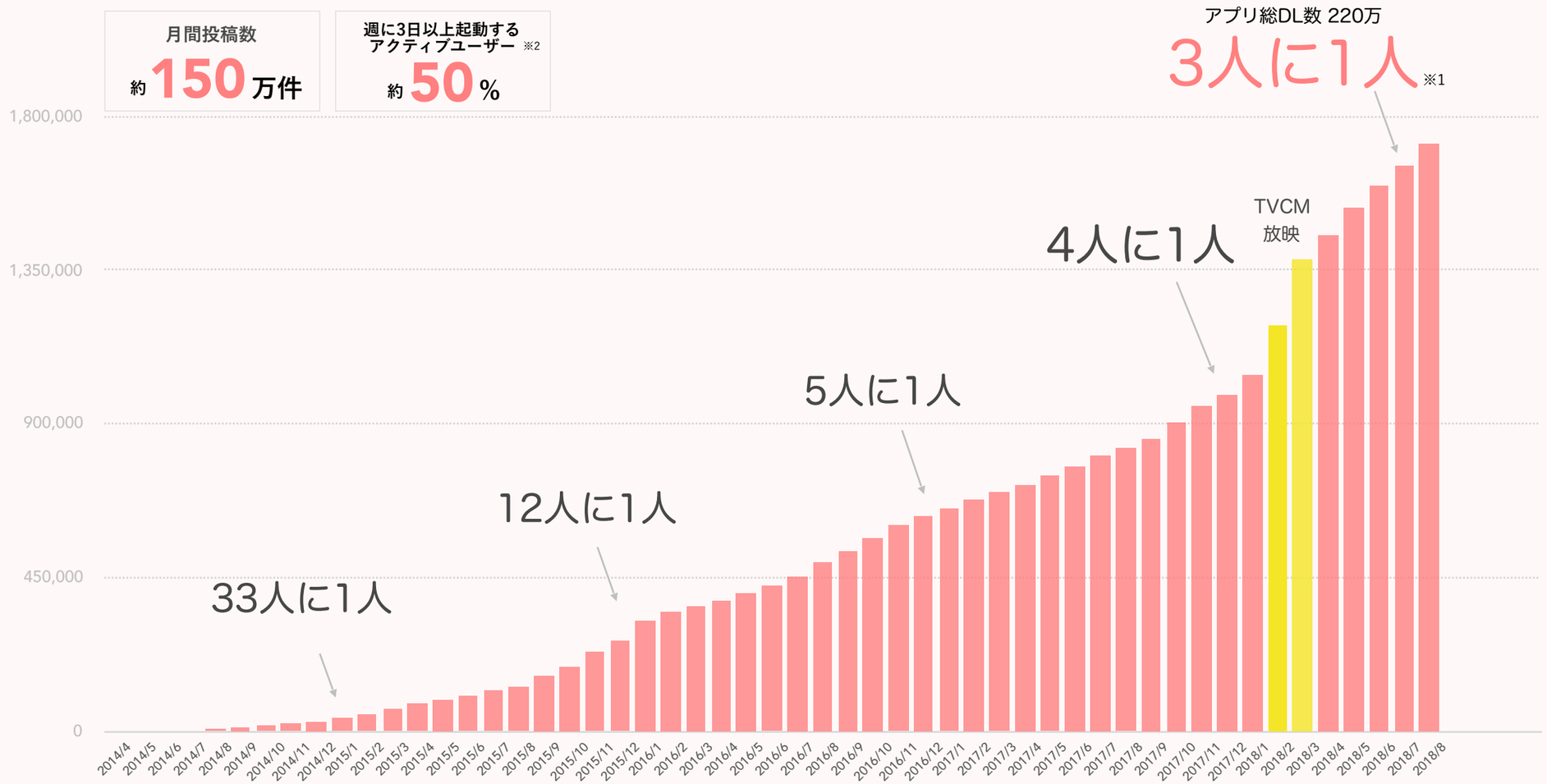
記事数

6,000 記事以上

※1 「閲覧数」「利用者数」はメディアとアプリの合計値（2019年4月-6月の平均値）
※2「ママ向けNo.1アプリ」は2019年3月インテージ調べ　調査対象：妊娠中～２歳０ヶ月の子供を持つ女性(n=1,084)を抽出
※3 Instagramのフォロワー数、Facebookのいいね数、LINEのともだち数の合計値(2019年6月時点)

日本最大級規模を誇るブランドへ

2018年に出産したママの「3人に1人」※1がママリを利用中
日本最大級規模を誇るブランドへと成長しています



※1 「ママリ」内の出産予定日を設定したユーザー数と、厚生労働省発表「人口動態統計」の出生数から算出
※2 週に1回以上起動するユーザー

mamari

アジェンダ

- **ホスト型バッチ実行環境の課題を振り返る**
- **バッチ処理のコンテナ化**

  【事例1】

  スケジュール系バッチ処理をコンテナ化してECS ScheduleTaskに移行
- **バッチ処理のコンテナ+サーバレス化**

  【事例2】

  機械学習の学習データ前処理フロー(Glue + Fargate)を

  Step Functionsで標準化

mamari

AWSでのバッチ処理パターンについて

一つでも気づきを得てもらえればうれしいです

DEV DAY

ホスト型バッチ実行環境の課題を振り返る

aws

ホスト型バッチサーバの例

1台のバッチサーバ

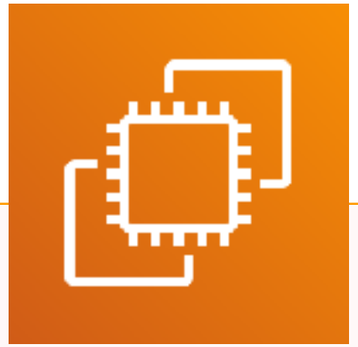
1台のEC2で運用

cronでスケジュール管理

様々なバッチ処理を集約

大きなコンピューティングリソース

2台構成にして冗長化

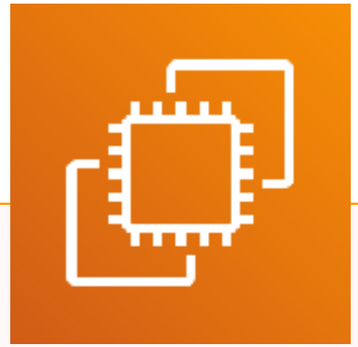


サービスAバッチ: cron(10 0 * * *)

サービスBバッチ: cron(20 0 * * *)

運用Cバッチ: cron(30 0 * * *)

EC2



サービスAバッチ: cron(10 0 * * *)

サービスBバッチ: cron(20 0 * * *)

運用Cバッチ: cron(30 0 * * *)

EC2

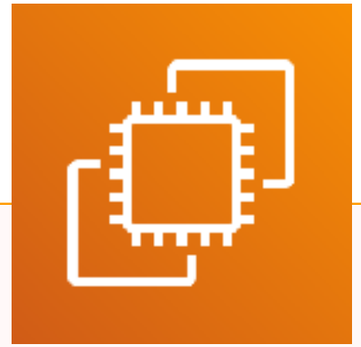
バッチ毎にActive/Standbyにする

サービス系バッチサーバ

サービスAバッチ: cron(10 0 * * *)

サービスBバッチ: cron(20 0 * * *)

運用Cバッチ: cron(30 0 * * *)



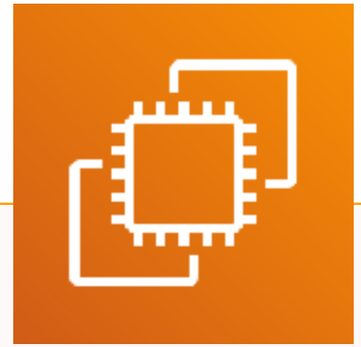
EC2

運用系バッチサーバ

サービスAバッチ: cron(10 0 * * *)

サービスBバッチ: cron(20 0 * * *)

運用Cバッチ: cron(30 0 * * *)

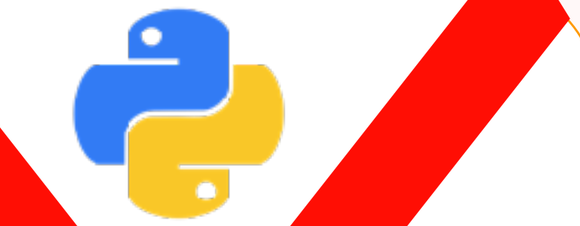


EC2

mamari

1台死んだら手動でフェイルオーバー

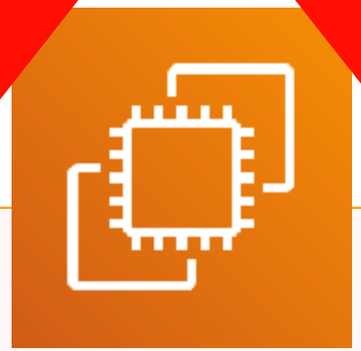
サービス系バッチサーバ



サービスAバッ cron(10 0 * * *)

サービスB n(20 0 * * *)

運用C チ: cron 0 * * *)



EC2

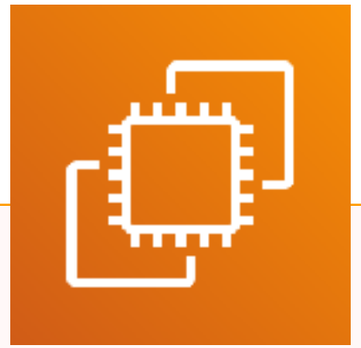
コメントアウト
外してActiveに

運用系バッチサーバ

サービスAバッチ: cron(10 0 * * *)

サービスBバッチ: cron(20 0 * * *)

運用Cバッチ: cron(30 0 * * *)



EC2

mamari

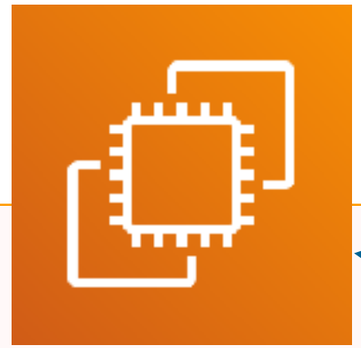
複数のサーバに全く同じ変更を行う必要がある

サービス系バッチサーバ

サービスAバッチ: cron(10 0 * * *)

サービスBバッチ: cron(20 0 * * *)

運用Cバッチ: cron(30 0 * * *)



EC2

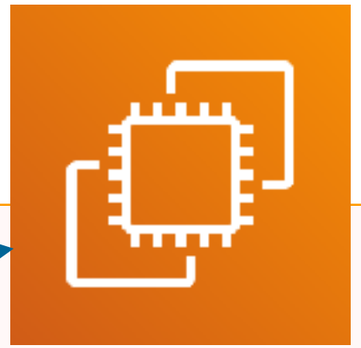
アプリケーションデプロイ
プラットフォームの変更

運用系バッチサーバ

サービスAバッチ: cron(10 0 * * *)

サービスBバッチ: cron(20 0 * * *)

運用Cバッチ: cron(30 0 * * *)



EC2

mamari

複数のサーバに全く同じ変更を行う必要がある

サービス系バッチサーバ

サービスAバッチ: cron(10 0 * * *)

サービスBバッチ: cron(20 0 * * *)

運用Cバッチ: cron(30 0 * * *)

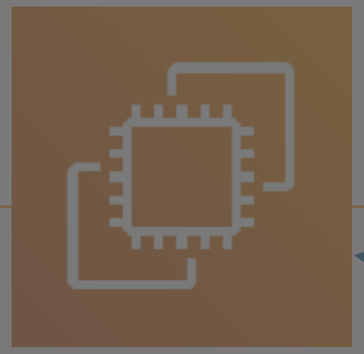
運用系バッチサーバ

サービスAバッチ: cron(10 0 * * *)

サービスBバッチ: cron(20 0 * * *)

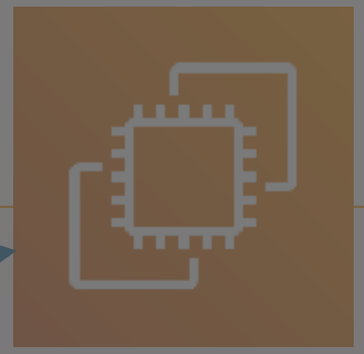
運用Cバッチ: cron(30 0 * * *)

バッチサーバにかかる運用負荷は大きい



EC2

アプリケーションデプロイ
プラットフォームの変更



EC2

mamari

ホスト型バッチ処理の課題

- プラットフォームに変更を加えるハードルが高い
  - バッチサーバで動く多岐にわたる処理が問題なく動くように考慮する必要がある
  - 複数サーバに全く同じ変更を加える必要がある
  - 常に何かしらの処理が動いているのでロールバックが難しい
- 運用負荷が高い
  - バッチサーバのインフラ設定変更
  - cron登録のオペレーションやステージングでのテスト実行
  - 障害時の切り分け(バッチ個別の問題?サーバリソース?ゾンビプロセスが、、、)

ホスト型バッチ処理の課題

- プラットフォームに変更を加えるハードルが高い
  - バッチサーバで動く多岐にわたる処理が問題なく動くように考慮する必要がある
  - 複数サーバに全く同じ変更を加える必要がある
  - 常に何かしらの処理が動いているのでロールバックが難しい
- 運用負荷が高い
  - バッチサーバのインフラ設定変更
  - cron登録のオペレーションやステージングでのテスト実行
  - 障害時の切り分け(バッチ個別の問題?サーバリソース?ゾンビプロセスが、、、)

バッチ処理をコンテナ化することで解決

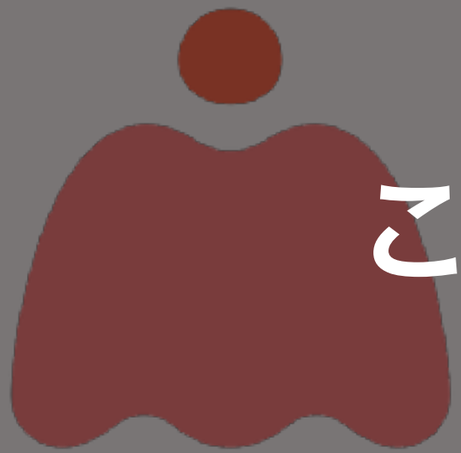
ホスト型バッチ処理の課題

- リソースを共有する観点から緻密なスケジュール管理が必要
  - 限られたマシンリソースを共有するので重い処理が重ならないよう考慮
  - 緻密なスケジュール管理(バッチ管理台帳)
  - ジョブスケジューラの導入→ジョブスケジューラ自体の運用が発生
  - スケジューリングの経緯等は一部に属人化することもおきがち
- 費用が高くなりがち
  - 一番マシンリソースを使う処理をベースにインスタンスを決定し常時起動
  - 冗長化や役割毎バッチサーバを作ると更に費用が高くなる
  - クラウドの場合は、必要な時に使った分だけの費用を払う形にしたい

ホスト型バッチ処理の課題

- リソースを共有する観点から緻密なスケジュール管理が必要
  - 限られたマシンリソースを共有するので重い処理が重ならないよう考慮
  - 緻密なスケジュール管理(バッチ管理台帳)
  - ジョブスケジューラの導入→ジョブスケジューラ自体の運用が発生
  - スケジューラの仕様差異等は一部に属人化することもあるがち
- 費用が高くなりがち
  - 一番マシンリソースを使う処理をベースにインスタンスを決定し常時起動
  - 冗長化や役割毎バッチサーバを作ると更に費用が高くなる
  - クラウドの場合は、必要な時に使った分だけの費用を払う形にしたい

バッチ処理をサーバレス化することで解決



これから2つの事例をベースに

バッチ実行環境をコンテナ・サーバレス化するメリットについてお話します

DEV DAY

【事例1】
スケジュール系バッチ処理をコンテナ化して ECS ScheduleTaskに移行

aws

この事例でお話すること

・サービス運用で使うスケジュールベースバッチをコンテナ化

・それにより解決した課題と構成について

mamari

扱うバッチ処理

- スケジュール系のバッチ処理
  - 事前に定義した時間に正確に動くことが期待される
  - 特定のサイクル(10分毎)で動くことが期待される
- バッチサーバのcronでスケジュール管理
  - バッチサーバにアプリケーションをデプロイ
  - crontabでスケジュール管理

コネヒトのバッチサーバに対する課題

- バッチを司るcronサーバの運用をやめたい
  - 全サービス共通のEC2
  - 動いているのは、時間通りに動くことが期待されるスケジュール系バッチ
- アプリケーションはコンテナ化済でその資産を活かしたい
  - ベースコードはwebと共通なのでコンテナ化済
  - AWSを基盤として使っているので、AWSでコンテナのバッチ処理を実現したい
  - 社内ではDockerでの開発が馴染んできていた

【コンテナ導入の正攻法】

https://speakerdeck.com/shoichiron/confrontation-of-container-transfer

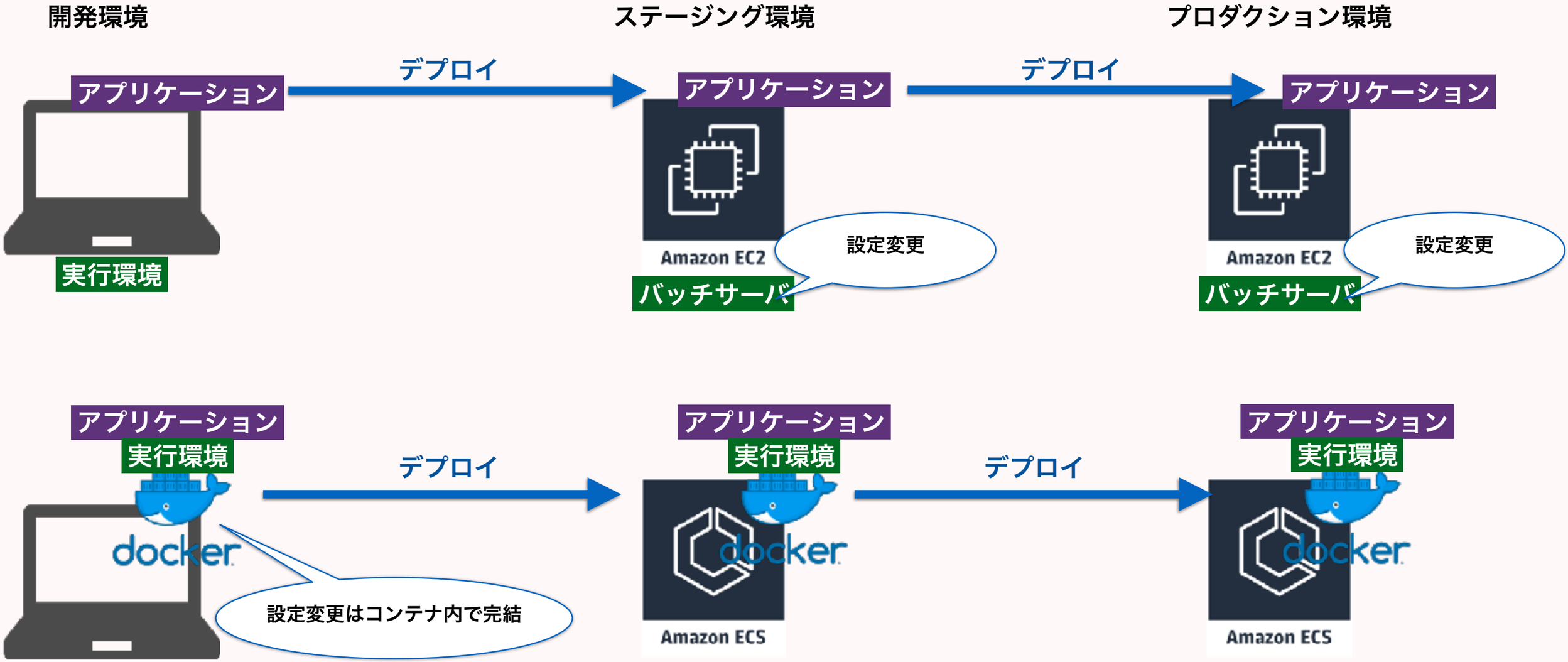
コネヒトのバッチサーバに対する課題

- 属人化しやすくレガシーな運用が残っている
  - 新規登録時は、crontabで手動スケジュール登録
  - リリース前のテスト実行はSSH経由でバッチサーバから動作確認
  - crontabに登録するのは一部のエンジニアのみ(一部しか全容を知らない状態)
- アプリケーションリリース以外は極力変更を加えないように
  - シングルポイントなので止まるとサービスに影響大
  - 出来るだけ変更は加えずに運用したいという意識が働く

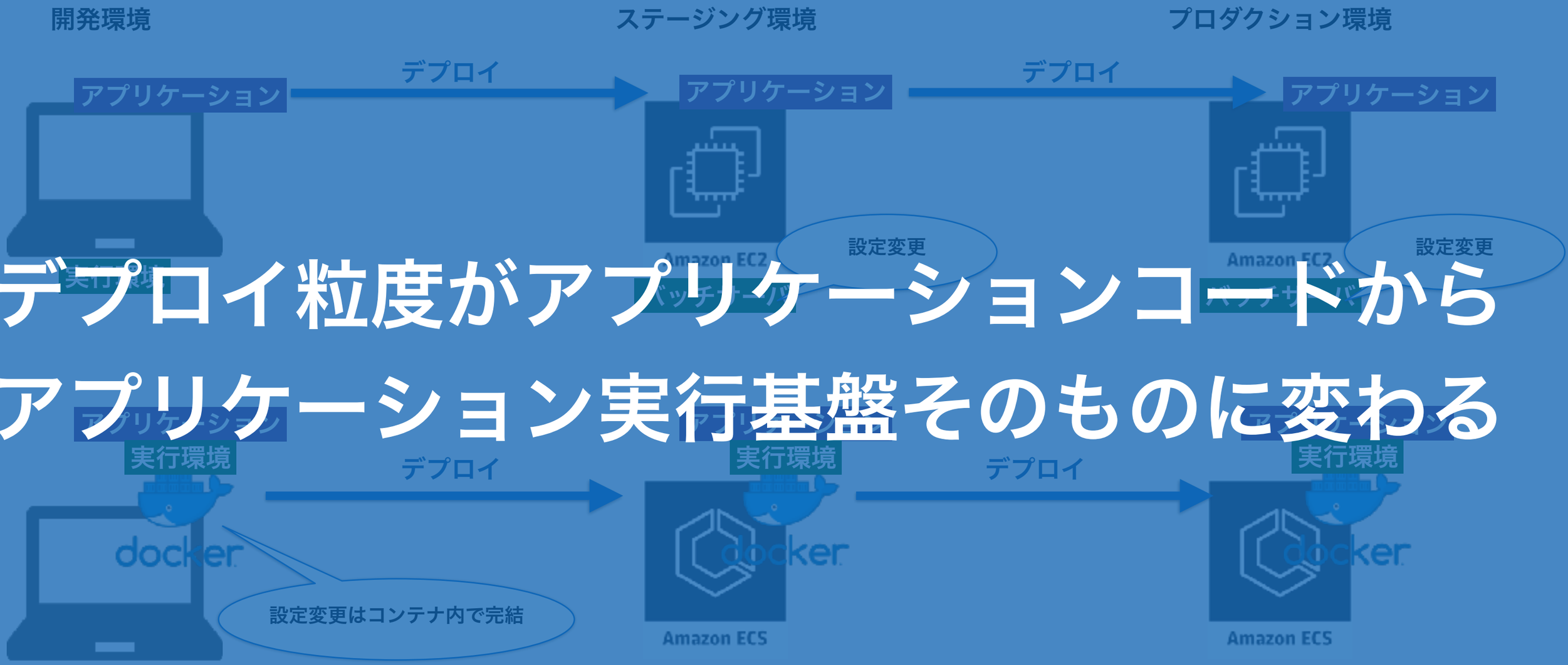
これらの課題を解決するために
コンテナ駆動なバッチ実行環境を構築

バッチ処理をコンテナ化することで
どのようなメリットが得られるのか？

①可搬性の高さを活かすことでプラットフォーム変更の敷居が下がる



①可搬性の高さを活かすことでプラットフォーム変更の敷居が下がる



デプロイ粒度がアプリケーションコードから
アプリケーション実行基盤そのものに変わる

②運用負荷が下がる

- 専用のバッチサーバは不要に
  - どのバッチをどのホストで動かすかはオーケストレーションツールにお任せ
  - バッチ処理=1コンテナプロセスの実行に集中出来る
  - crontab運用からの卒業
- 属人化を排除出来る
  - プラットフォームの定義はDockerfileでコード化
  - バッチ実行の定義もコード化することで通常の開発フローでバッチ運用が出来る

AWSでどのように
コンテナ基盤のバッチ実行環境を作るか?

技術選定

- コンテナon cron, Lambda, AWS Batch等を比較検討
  - 当時のアーキテクチャ選定時のpros&consはこちらに

【Amazon ECS ScheduleTaskで実現するスマートなDockerベースのバッチ実行環境】

https://tech.connehito.com/entry/2017/09/13/171914

- 全ての要件を満たしたのでECS ScheduleTaskの採用を決定
  - これからやるなら要件によってはAWS Batchでもいいのかも?(大規模データ等)
  - 当時はスケジュール起動に対応していなかった

- 導入当初は他社事例があまりないという不安はあった
  - SAに相談してアーキテクチャのレビューを受ける
  - ECSの運用実績はあり知見があったので導入を決定

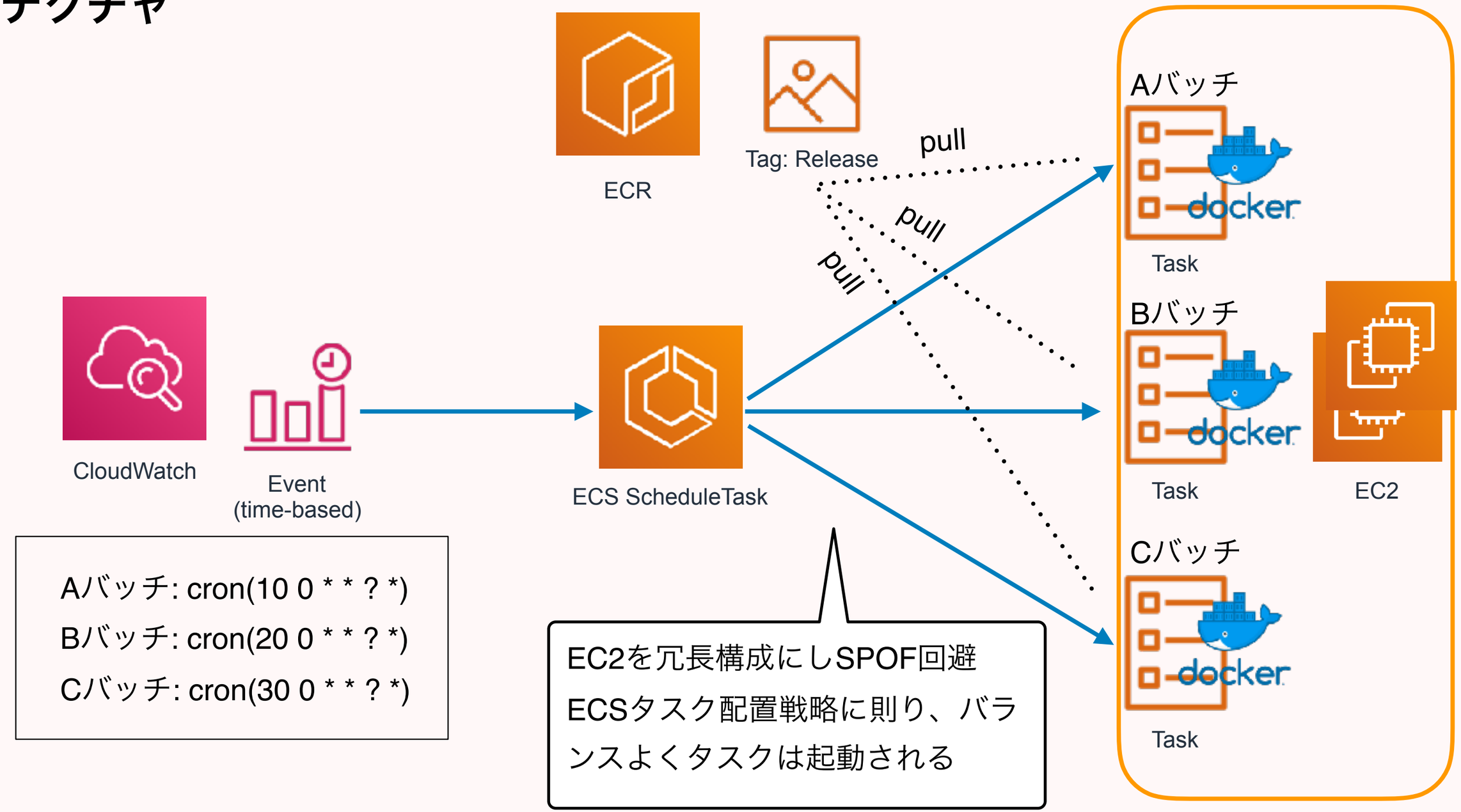
ECS ScheduleTaskとは??

- ECSはAWSが提供するコンテナ管理のマネージドサービス
  - コンテナオーケストレーションツール
- ECS ScheduleTaskはタスク(コンテナ)の実行をスケジュール管理してくれる機能
  - CloudWatchEventと連携

Event
(time-based)

mamari

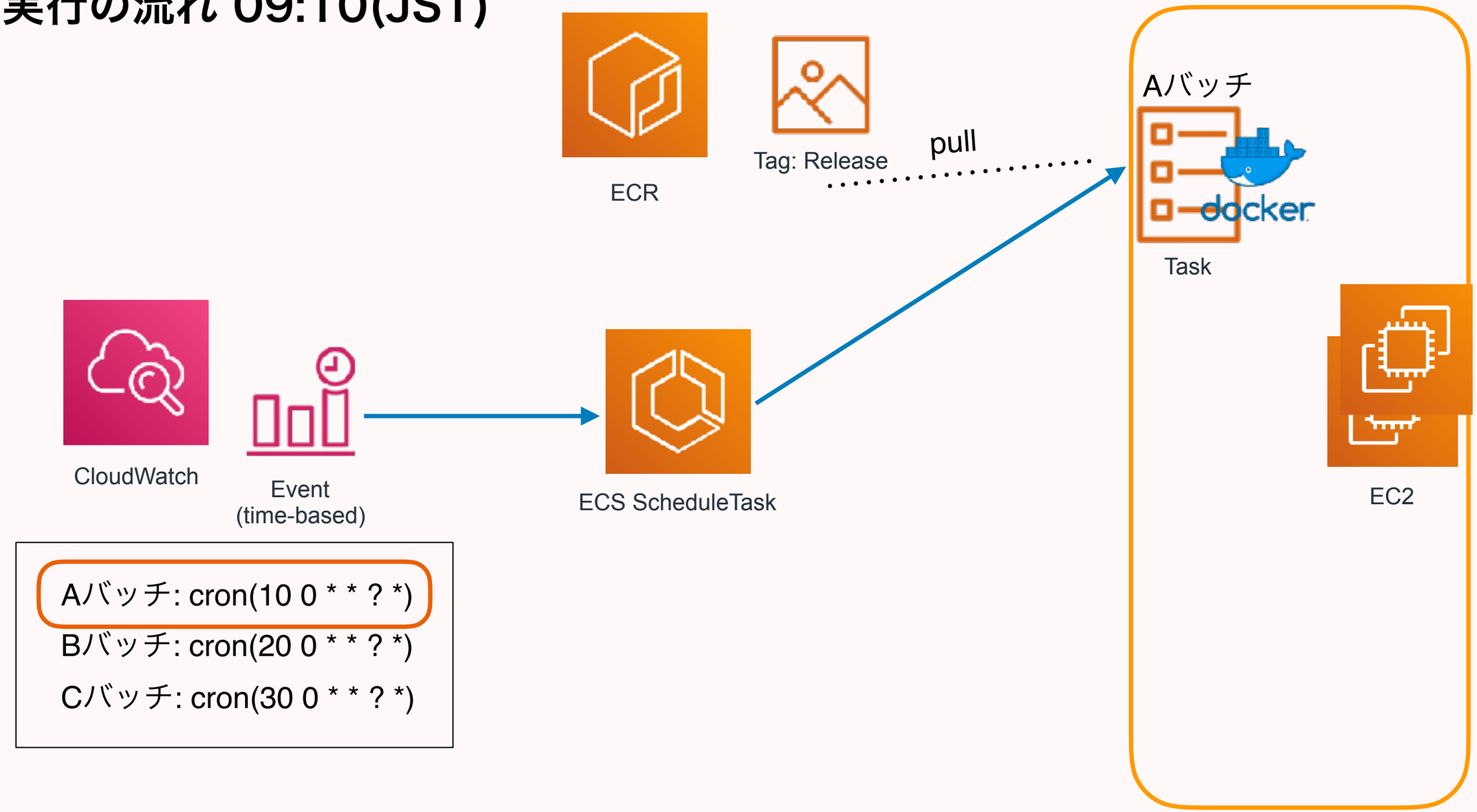
アーキテクチャ



mamari

ECS ScheduleTaskでの
バッチ実行の流れを見てみましょう

バッチ実行の流れ 09:10(JST)



mamari

バッチ実行の流れ 09:20(JST)

ECR

Tag: Release

pull

Aバッチ

Task

Bバッチ

Task

CloudWatch

Event
(time-based)

ECS ScheduleTask

EC2

Aバッチ: cron(10 0 * * ? *)

Bバッチ: cron(20 0 * * ? *)

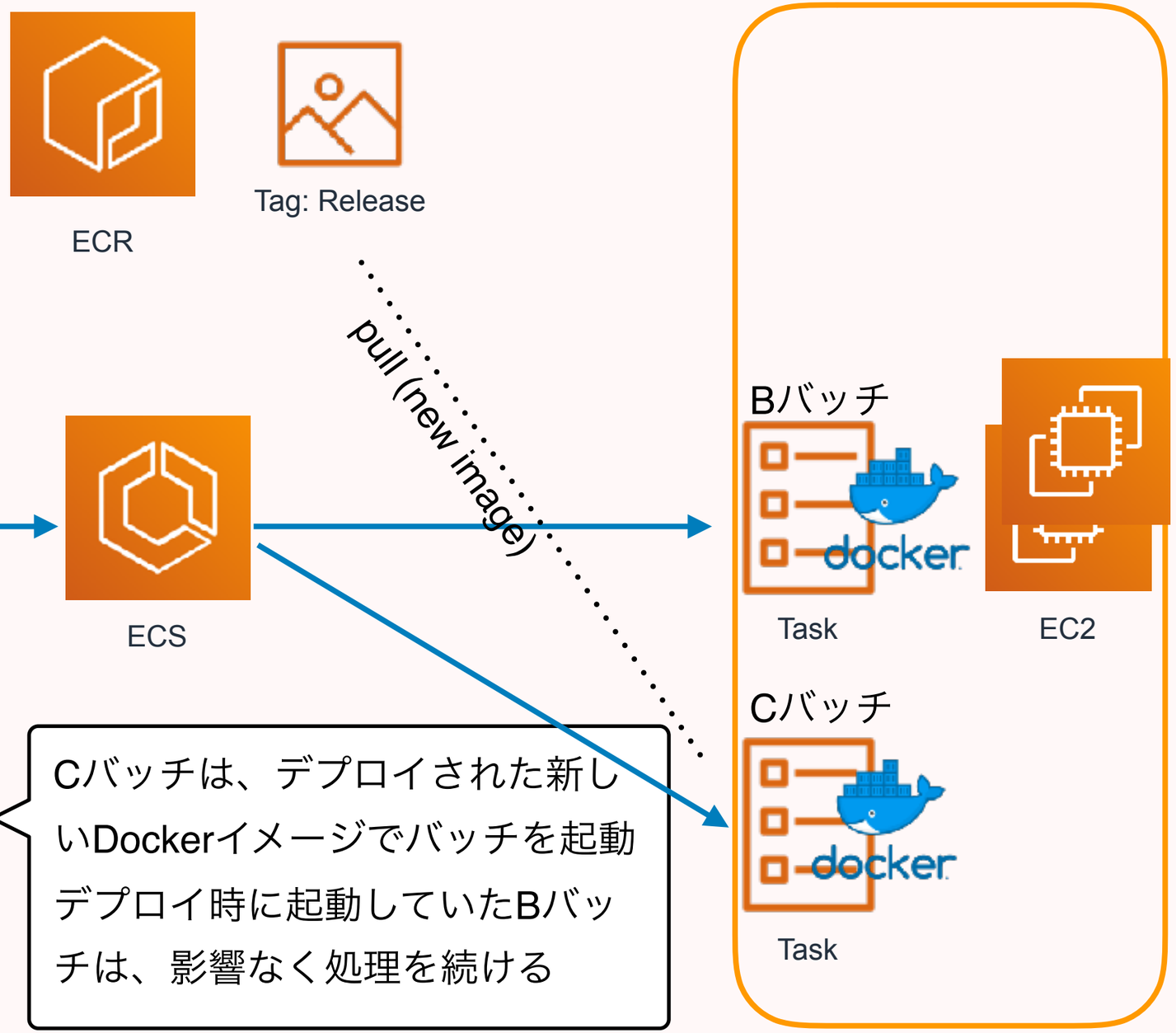
Cバッチ: cron(30 0 * * ? *)

Bバッチ起動時は、
Aバッチ起動時に出来たEC2上の
Dockerレイヤキャッシュが効くの
で高速起動

mamari

09:23(JST)に本番デプロイ発生

バッチ実行の流れ 09:30(JST)



mamari

ECS ScheduleTaskのメリット

- cron記法でスケジュール定義が出来るのでわかりやすい

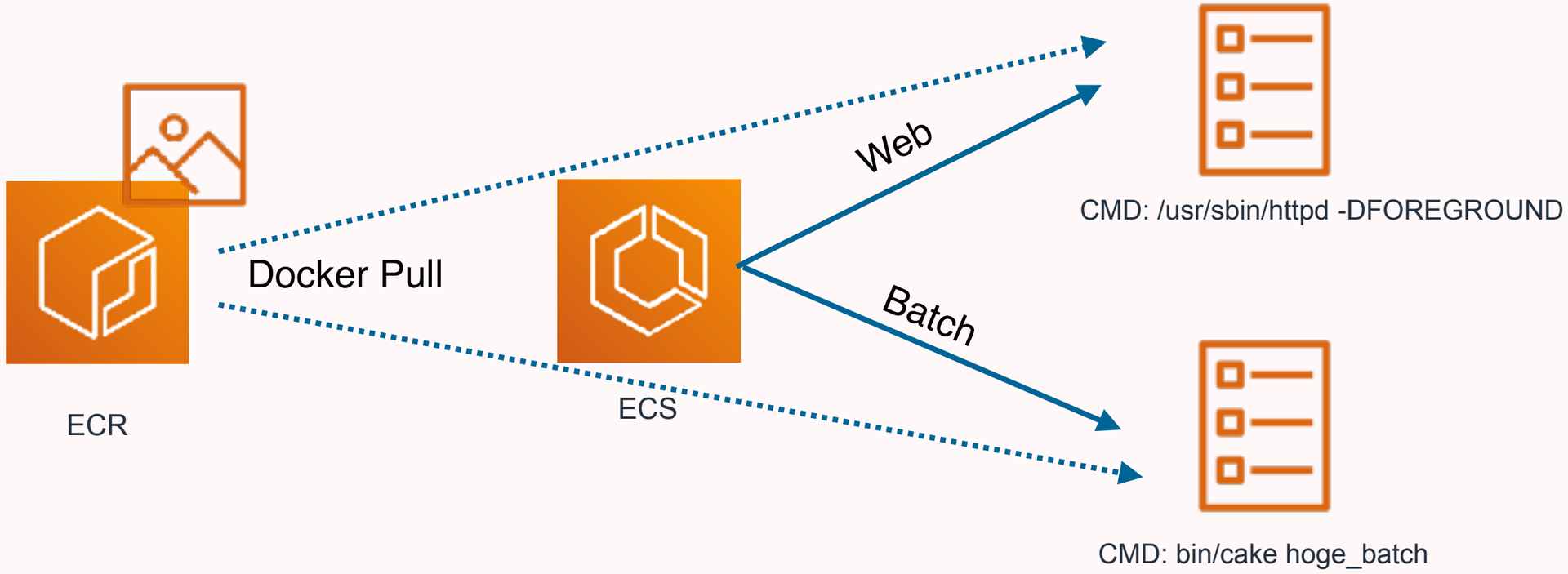
  CloudWatchEventsのcron式の特徴

  - 年という概念があり6フィールド
  - タイムゾーンはUTC
  - 毎日 15:00に起動する場合

```
cron(0 6 * * ? *)
```

ECS ScheduleTaskのメリット

- 既存のDockerイメージを資産として使える
  - webと同じコードベースでバッチを動かす場合はcmd句の書換えだけでOK



ECS ScheduleTaskのメリット

- EC2バックエンド構成だとコンテナの起動が早い
  - 起動にかかる時間は数秒なのでスケジュール起動の要件を満たせる
  - Docker実行ホストが固定されるのでDockerのレイヤキャッシュが効く
  - Fargateはホストを固定出来ないのでレイヤキャッシュが効かない&ENIの付け替えもあり起動に1分弱かかる(前は2分ほどかかってたので早くはなっている)

ECS ScheduleTaskのメリット

- デプロイ時に実行中バッチへの考慮が不要
  - 起動時にDockerイメージをpullするアーキテクチャ
  - 起動中のコンテナはpullした段階のアプリケーションを使うので影響を受けない
- Cloud Watch Logsで簡単にログが見れる
  - 標準出力をCloud Watch Logsへ出力
  - AWSコンソールからアプリケーションの出力を簡単に見れる
  - わざわざSSHをしなくてもOK

ECS ScheduleTaskのメリット

- GUIでバッチ一覧が見れる
  - コメントもつけれるのでバッチ管理台帳の代用になる

mamari

ECS ScheduleTaskは何がうれしいのか

- バッチサーバの運用が不要になる
  - コンテナとEC2の管理はECSが行ってくれる
  - EC2をmultiAZで運用することでSPOF解消→どこで起動するかもECSがよしなに
- Dockerレイヤキャッシュを使うことでスケジュール通りのバッチ実行が可能
- 起動タイプとしてFargateを使うことも出来る
  - サーバレスのメリットも享受可能

改善してほしいポイント

- タスクの実行履歴が時系列で見れない
  - 時間軸でどのタスクが動いていたのかデバッグするのが少し面倒。。

ECS ScheduleTaskでバッチを組むための要素

- クラスタ
  - タスクを動かすホスト(EC2)の集合体
  - Fargateの時は、ただの箱
- Dockerイメージ(ECR)
- タスク定義
  - Dockerコンテナの起動パラメータの定義
- クラスタ内のScheduleTask
  - CloudWatchEventsと裏で連携している

タスク定義をTerraformで管理

- クラスタ
  - タスクを動かすホスト(EC2)の集合体
  - Fargateの時は、ただの箱
- Dockerイメージ(ECR)
- タスク定義
  - Dockerコンテナの起動パラメータの定義
- クラスタ内のScheduleTask
  - CloudWatchEventsと裏で連携している

タスク定義をTerraformで管理

バッチサーバの属人化解消へ

Terraformで管理することで解決

- バッチサーバの運用が属人化
  - 新規登録時は、crontabで手動スケジュール登録
  - リリース前のテスト実行はSSH経由でバッチサーバから動作確認
  - crontabに登録するのは一部のエンジニアのみ(一部しか全容を知らない状態)
- アプリケーションリリース以外は極力変更を加えないように
  - シングルポイントなので止まるとサービスに影響大
  - 出来るだけ変更は加えずに運用したいという意識が働く

タスク定義をTerraformで管理

- 開発者がバッチの登録/変更/削除をPullRequest形式で行う
  - Terraformの定義を見ればどんなバッチがあるのか理解できる
  - crontabを誰が登録&チェックするのか問題を解決
- Terraformの実行も開発者が行う
  - tfstateファイルをS3で共通管理することで手元で実行できるように
  - Opsチームへのバッチ登録依頼も不要に

Terraformの定義

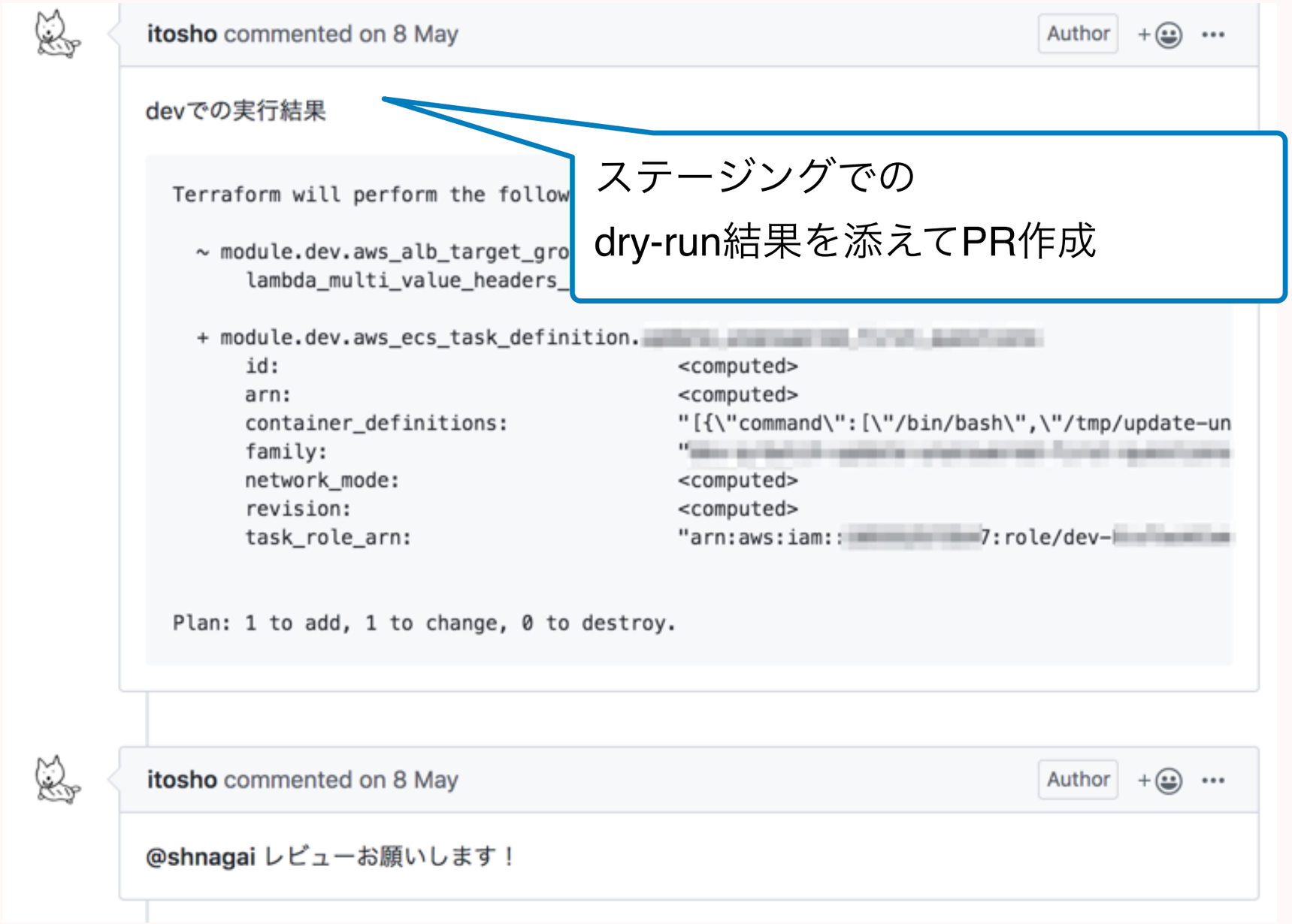
```
# 質問の状態更新バッチ / 5分ごと
# 2-57/5 * * * *
resource "aws_ecs_task_definition" "
  family                = "${var.environment}
  task_role_arn         = "${var.ecs_task_rol
  container_definitions = <<DEFINITION
[
  {
    "name": "${var.environment}-
    "image": "${var.ecr_url["
    "memory": 300,
    "command": ["/bin/bash", "/tmp/
    "workingDirectory": "/var/www/html",
    "environment": [
      {"name": "CAKEPHP_ENV", "value": "${var.ecs_task_environment["cakephp_env"]}"},
    ],
    "logConfiguration": {
      "logDriver": "awslogs",
      "options": {
        "awslogs-region": "ap-northeast-1",
        "awslogs-stream-prefix": "edit-question",
        "awslogs-group": "${var.awslogs_group_name["
```

Description書くことでひと目でわかる

値はvariables.tfで管理することで、
環境差分を吸収

mamari

PRベースでのバッチ登録



itosho commented on 8 May — Author

devでの実行結果

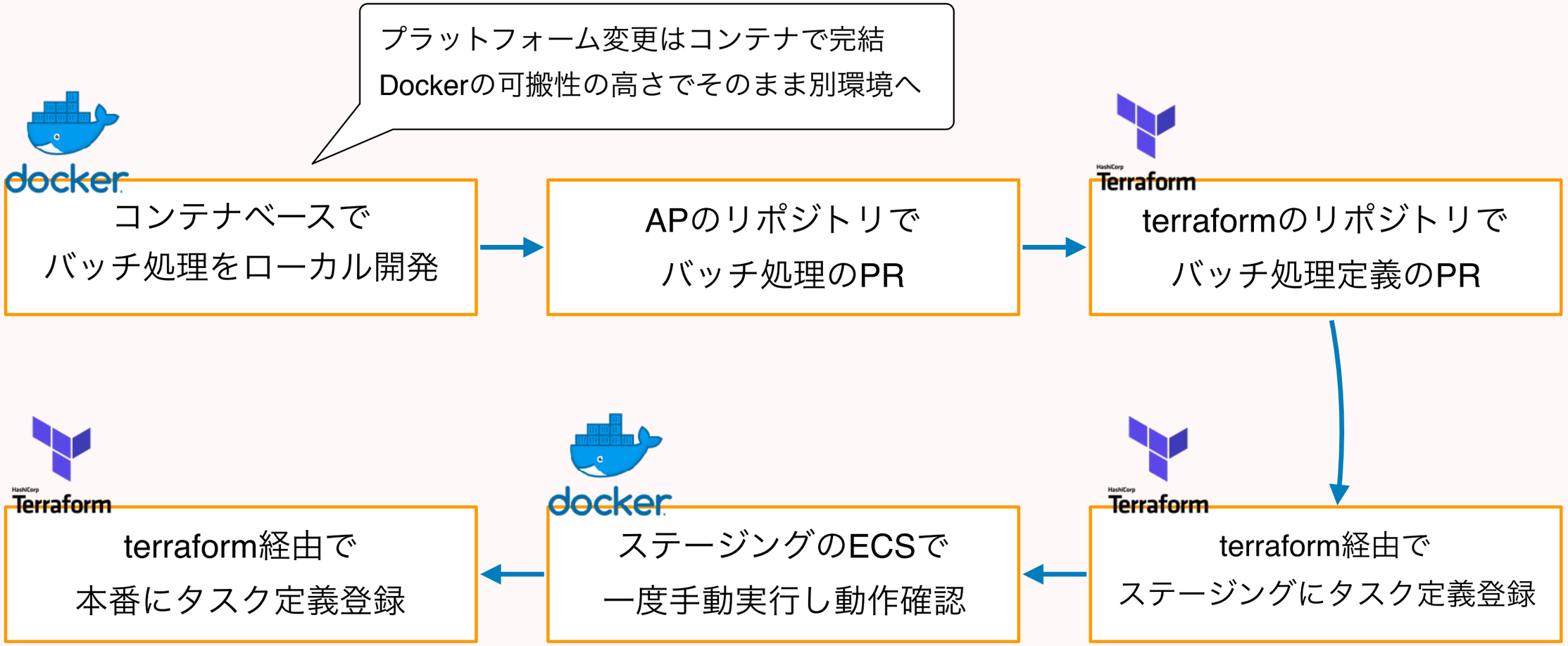
```
Terraform will perform the follow

  ~ module.dev.aws_alb_target_gro
      lambda_multi_value_headers_

  + module.dev.aws_ecs_task_definition.[redacted]
      id:                                   <computed>
      arn:                                  <computed>
      container_definitions:                "[{\"command\":[\"/bin/bash\",\"/tmp/update-un
      family:                               "[redacted]
      network_mode:                         <computed>
      revision:                             <computed>
      task_role_arn:                        "arn:aws:iam::[redacted]7:role/dev-[redacted]


Plan: 1 to add, 1 to change, 0 to destroy.
```

ステージングでの
dry-run結果を添えてPR作成

itosho commented on 8 May — Author

@shnagai レビューお願いします！

バッチの開発フロー


プラットフォーム変更はコンテナで完結
Dockerの可搬性の高さでそのまま別環境へ
コンテナベースで
バッチ処理をローカル開発
APのリポジトリで
バッチ処理のPR
terraformのリポジトリで
バッチ処理定義のPR
terraform経由で
ステージングにタスク定義登録
ステージングのECSで
一度手動実行し動作確認
terraform経由で
本番にタスク定義登録

バッチの開発フロー

プラットフォーム変更はコンテナで完結
Dockerの可搬性の高さでそのまま別環境へ

コンテナベースで
バッチ処理をローカル開発

→

APのリポジトリで
バッチ処理のPR

→

terraformのリポジトリで
バッチ処理定義のPR

開発/デプロイ/確認という開発フローを
すべて開発者の責務で行うことができる

terraform経由で
本番にタスク定義登録

→

ステージングのECSで
一度手動実行し動作確認

→

terraform経由で
ステージングにタスク定義登録

ECS Schedule Taskを使ったバッチ実行環境のまとめ

- コンテナ化しているアプリケーションにおいてcronサーバ運用なしにスケジュール実行が可能
- バッチサーバを意識せずに、バッチ処理=1プロセスの改修という粒度でプラットフォームも含めて変更が可能
- 定義をコード管理することで通常の開発サイクルでバッチ登録が可能になる

DEV DAY

【事例2】
機械学習の学習データ前処理フロー(Glue + Fargate)をStep Functionsで標準化

この事例でお話すること

- 機械学習の前処理をサーバレスで新規構築
- バッチ処理で求められることも多いワークフローもサーバレスに構築

mamari

月間数百万件の投稿から不適切なコンテンツを見つける

質問者

e.g. 簡単に稼げる方法教えますよ

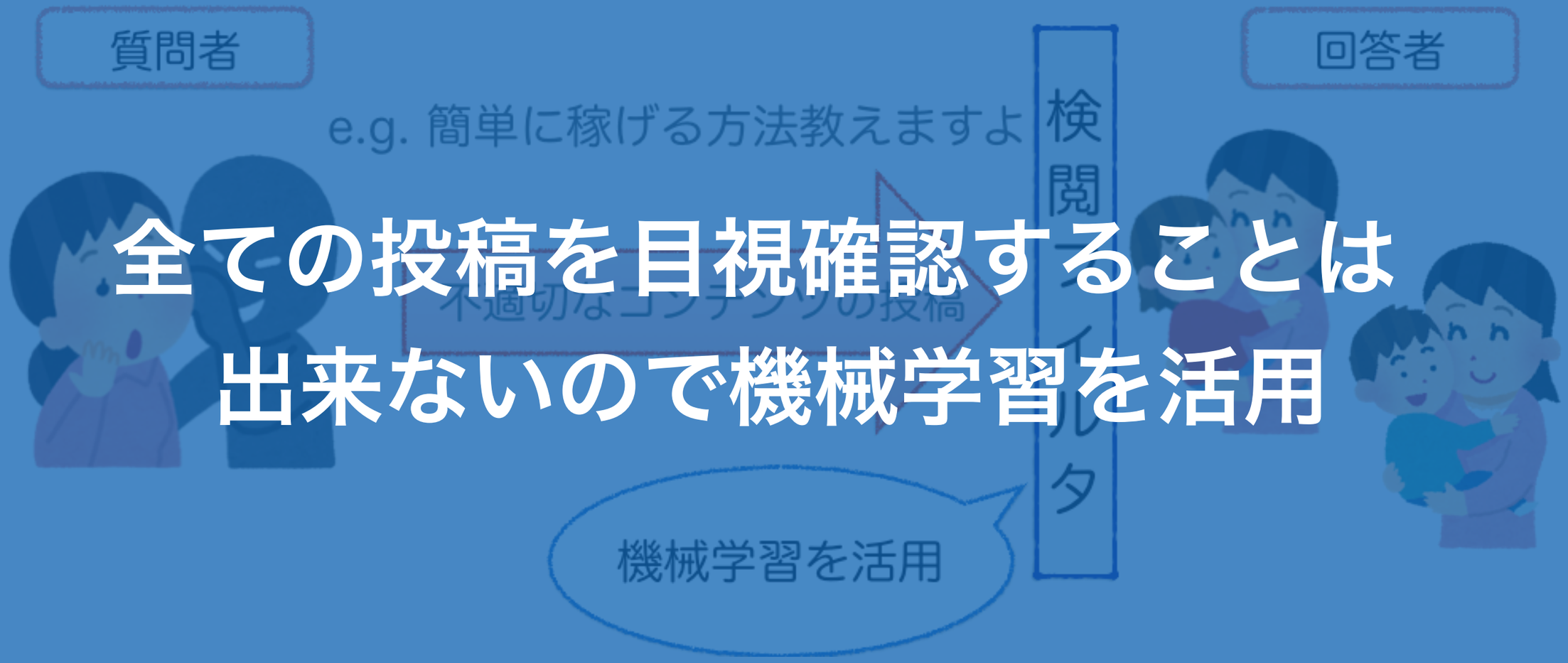
不適切なコンテンツの投稿

検閲フィルタ

機械学習を活用

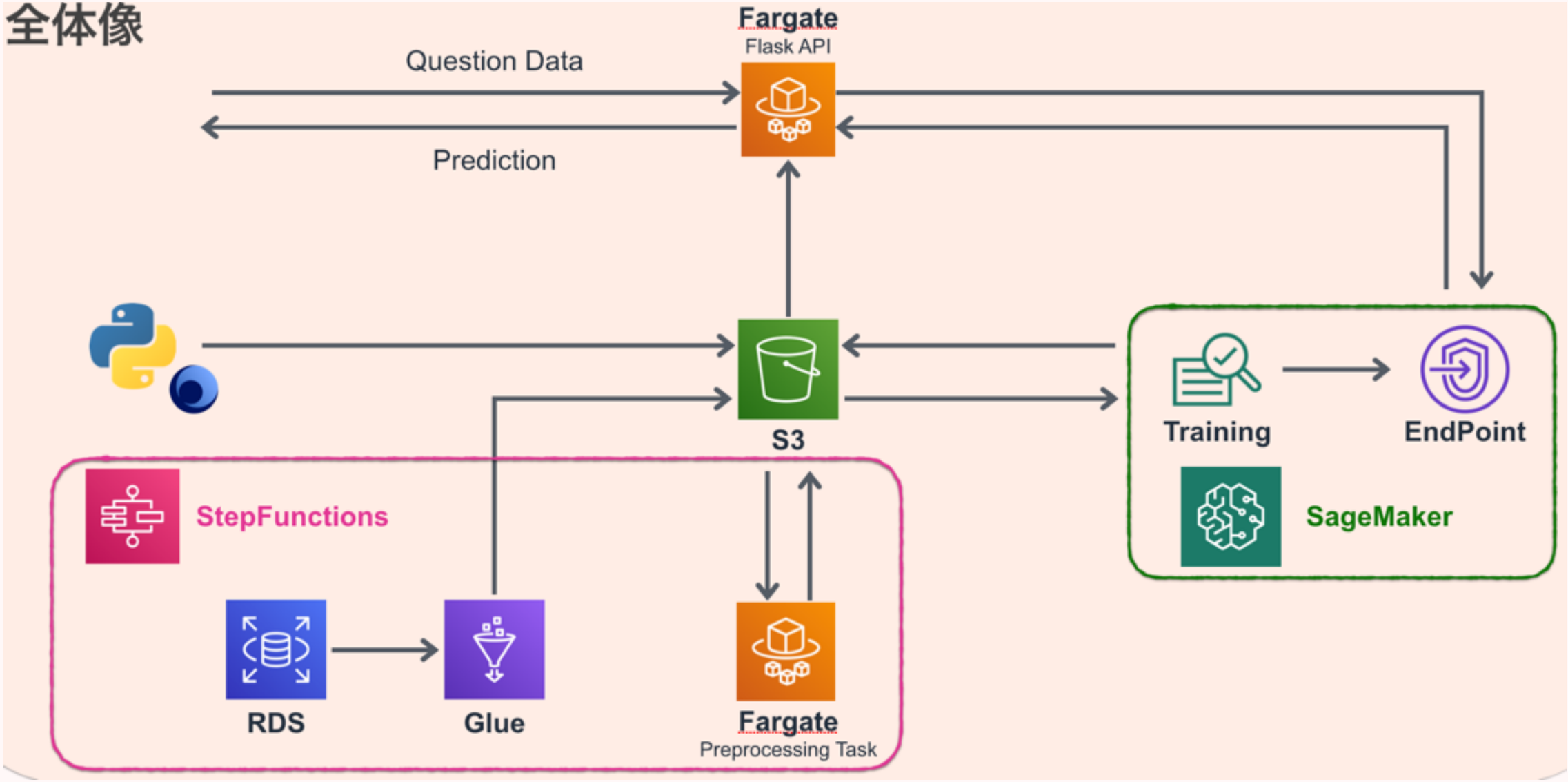
回答者

月間数百万件の投稿から不適切なコンテンツを見つける



全ての投稿を目視確認することは
出来ないので機械学習を活用

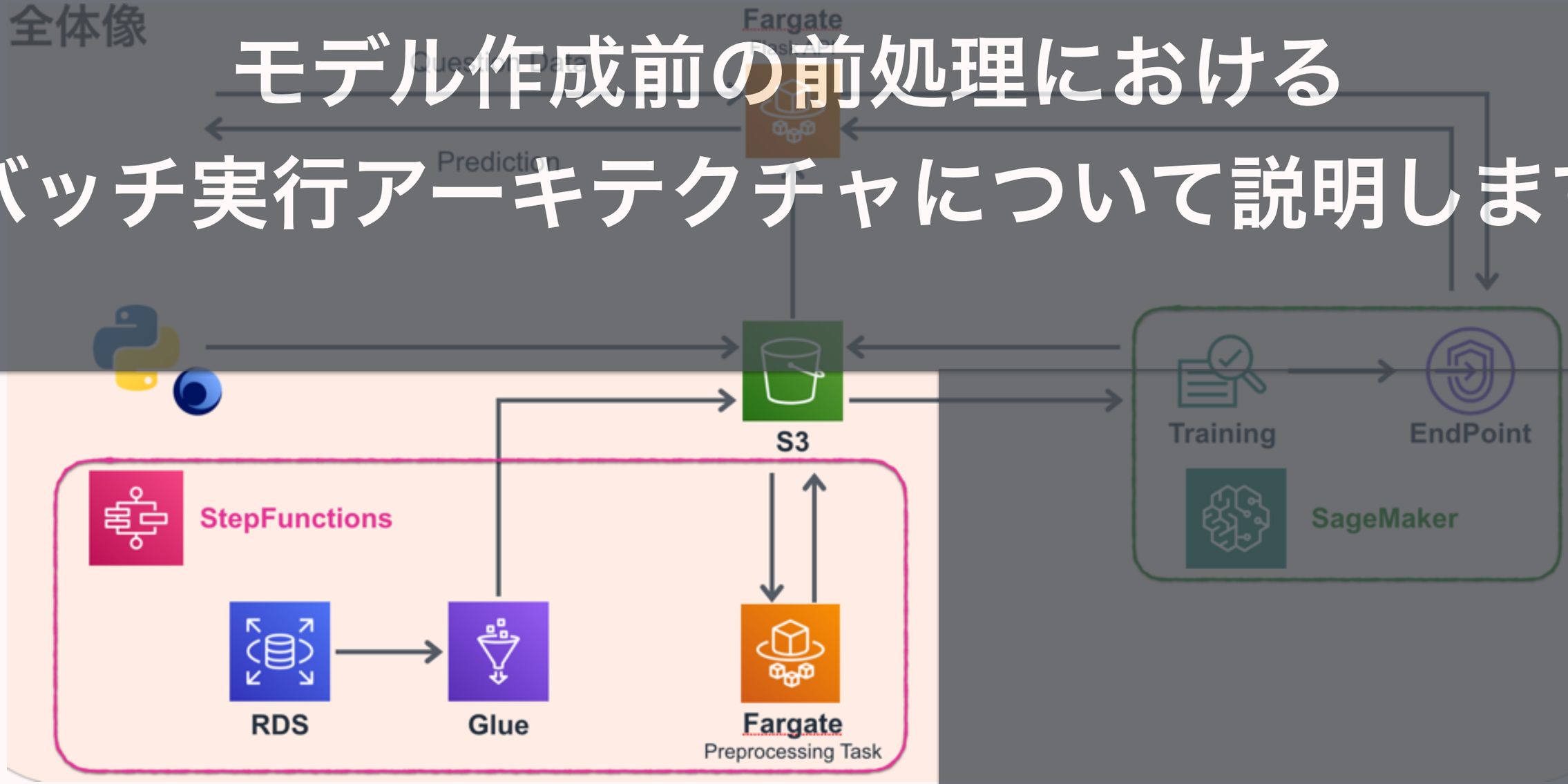
アーキテクチャ


全体像
Fargate
Flask API
Question Data
Prediction
S3
Training
EndPoint
SageMaker
StepFunctions
RDS
Glue
Fargate
Preprocessing Task

アーキテクチャ

全体像

モデル作成前の前処理における
バッチ実行アーキテクチャについて説明します



mamari

どのような処理を行っているか

- RDSに保存しているデータから学習データを抽出
  - AWS Glueを使ったデータ抽出処理

- 学習データに対しテキストの前処理とベクトル化を行う
  - コンテナ化したバッチ処理をFargateタスクで実行

AWS Glueとは

- データの抽出、変換、ロードを行うマネージドサービス
- 簡単な操作でETLジョブを構築可能
- **サーバレス**なので運用不要かつ低コスト運用が可能
  - ETL処理はマシンパワーが求められる処理が多いのでコスト効率は重要
  - DPU(Data Processing Unit)単位の課金

Glueを使ったデータ抽出処理

- RDSに保存されているデータを抽出・加工し、S3に保存

- 処理の内訳
  - データ件数：約700万件
  - Job実行時間：5分
  - 出力tsvデータ：約3GB

【AWS Glueを用いてETL環境を構築したお話（RDS for MySQL → S3）】
https://tech.connehito.com/entry/2019/04/18/164551

Fargateで行う前処理バッチ

- Glueで抽出した最新のデータを機械学習で使える形に変換
- Pythonを使った自然言語処理がメイン
  - 独自のランタイムを使いたいのでコンテナ化
  - SageMakerのビルトインコンテナでは使いたいライブラリが対応していなかった
- 開発フローはローカル開発+PRベース
  - masterマージ時にECRにDockerイメージをpush
- サーバレスなのでバッチ実行時のみ課金が発生
  - 機械学習の前処理は重めの処理が多いので料金面で恩恵を受けやすい

Fargateで大きめのファイルを扱う場合の注意点

- ストレージ容量に制限がある
  - 1 タスク内でコンテナのルートボリュームに10 GB、マウント用の共有ボリュームに4 GB利用可能
  - これを超えるファイルを扱う場合は保存先をS3等にする必要がある
  - EBSやEFSのマウントは現時点では出来ない

実行順序が重要な
2つのバッチ処理でワークフローを組みたい

mamari

どのようなパターンがあるか

キューイングパターン

- SQSを使ってジョブの実行順序を制御
  - LambdaまたはCloudWatchEvents経由で後続バッチを起動
  - キューイング間連の本来の処理とは関係ないコードを書かなければならない
- 処理毎にエラー通知やリトライ制御を入れる
  - Glueでのリトライ制御とエラー通知
  - バッチ処理のコードでリトライ制御とエラー通知
- 処理ステップが増えるほど全体像を把握するのが困難になる

スケジュール実行パターン

- 処理毎にスケジュール実行
  - Glueスケジュール実行(AM0:00)
  - Fargateスケジュール実行(AM1:00)
  - 最初の処理がこけて後続バッチが動いてもいい場合のみに使える
- 処理毎にエラー通知やリトライ制御を入れる
  - Glueでのリトライ制御とエラー通知
  - バッチ処理のコードでリトライ制御とエラー通知
- 処理ステップが増えるほど全体像を把握するのが困難になる

スケジュール実行パターン

- 処理毎にスケジュール実行
  - Glueスケジュール実行(AM0:00)
  - Fargateスケジュール実行(AM1:00)
  - 最初の処理がこけて後続バッチが動いてもいい場合のみに使える

もっと便利に出来る方法はないのか?

- 処理毎にエラー通知やリトライ制御を入れる
  - Glueでのリトライ制御とエラー通知
  - バッチ処理のコードでリトライ制御とエラー通知

- 処理ステップが増えるほど全体像を把握するのが困難になる

スケジュール実行パターン

- 処理毎にスケジュール実行
  - Glueスケジュール実行(AM0:00)
  - Fargateスケジュール実行(AM1:00)
  - 最初の処理がこけて後続バッチが動いてもいい場合のみに使える

Step Functionsを使うことで実現できた

- 処理毎にエラー通知やリトライ制御を入れる
  - Glueでのリトライ制御とエラー通知
  - バッチ処理のコードでリトライ制御とエラー通知

- 処理ステップが増えるほど全体像を把握するのが困難になる

Step Functionsとは

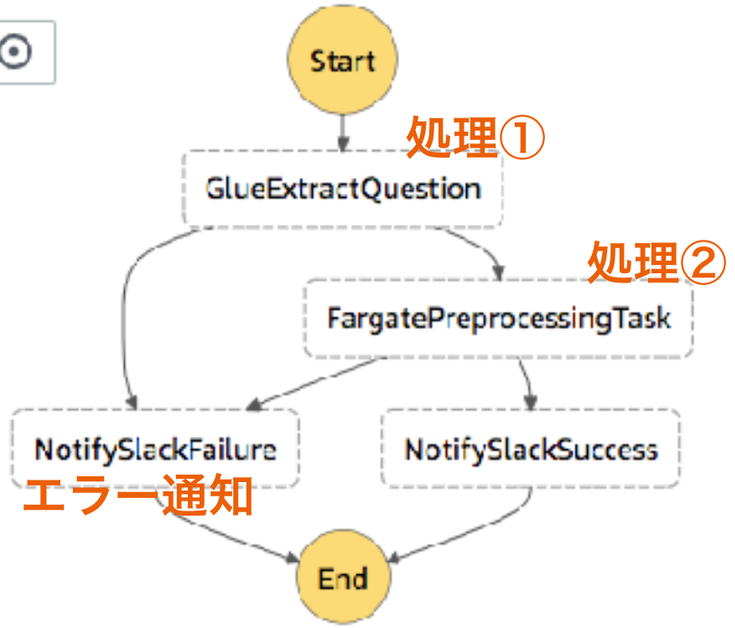
- AWSが提供するサーバーレスワークフローサービス
- 連携しているAWSリソースを簡単に制御出来る
  - 簡単に呼び出しかつジョブ終了待ちが出来る
  - 連携サービスは徐々に増えている
- JSONでワークフローを定義
  - リトライやエラーハンドリングが可能
  - 処理の前後関係をそのままビジュアライズしてくれる

StepFunctionsの定義

ステートマシンの定義

```
{
    "Comment": "PRD環境 [redacted]モデル自動更新処理[redacted]",
    "StartAt": "GlueExtractQuestion",
    "States": {
        "GlueExtractQuestion": {
            "Comment": "Glueを用いて[redacted]データの抽出",
            "Type": "Task",
            "Resource": "arn:aws:states:::glue:startJobRun.sync",
            "Parameters": {
                "JobName": "[redacted]"
            },
            "Catch": [{
                "ErrorEquals": ["States.ALL"],
                "Next": "NotifySlackFailure"
            }],
            "InputPath": "$",
            "ResultPath": "$.GlueExtractQuestionResult",
            "OutputPath": "$",
            "Next": "FargatePreprocessingTask"
        },
        "FargatePreprocessingTask": {
            "Comment": "FargateのTaskを用いて前処理",
            "Type": "Task",
            "Resource": "arn:aws:states:::ecs:runTask.sync",
```

エラーハンドリング



mamari

StepFunctionsの構成要素

- state
  - 実行する処理単位: Lambda, ECSタスク, Glue etc
  - Type句で処理内容を指定
    - Task(なんらかの処理実行)
    - Choice(入力に基づき処理分岐)
    - Wait(一定時間スリープ)
- state machine
  - steteの集合体(ワークフロー本体)
  - JSON ベースの Amazon ステートメント言語 で定義
  - この単位でビジュアライズされる

エラーハンドリング

- Catch句で定義する

- 事前定義されたエラーコード
  - States.ALL: すべてのエラー名に一致するワイルドカード
  - States.Timeout: タイムアウト関連
  - States.TaskFailed: タスクの実行失敗
  - States.Permissions: 定義したタスクの実行権限が足りない

リトライ処理

- Retry句で定義する

- 設定フィールド
  - ErrorEquals: 対象となるエラー名
  - IntervalSeconds: 再試行までのインターバル
  - MaxAttempts: 最大リトライ数
  - BackoffRate: 再試行時の増加する乗数 (例:3　1秒、4秒、7秒、、、)

AWSリソースとの連携 2019年9月時点

ジョブのハンドリングを
Step Functions側に任せられる

サポートされるサービス統合

サービス	リクエストレスポンス	ジョブの実行 (.sync)	コールバックまで待機 (.waitForTaskToken)
AWS Lambda	✓		✓
AWS Batch	✓	✓	✓
Amazon DynamoDB	✓		
Amazon ECS/Fargate	✓	✓	✓
Amazon Simple Notification Service	✓		✓
Amazon Simple Queue Service	✓		✓
AWS Glue	✓	✓	
Amazon SageMaker	✓	✓	
AWS Step Functions	✓	✓	✓

StepFunctionsの良いところ

- 処理の状態をハンドリングしてくれるので面倒な実装が不要
- AWSで完結するバッチ処理のワークフローが組みやすい
  - Glue→ECS→lambdaみたいなAWSサービスの連携
  - 複数のAWS Batchを使った集計処理
  - slack通知用の共通のlambdaを組み込む
- サーバレスなので運用不要でワークフローが組める

StepFunctionsの良いところ

- 処理の状態をハンドリングしてくれるので面倒な実装が不要
- AWSで完結するバッチ処理のワークフローが組みやすい
  - Glue + ECSでサービス間連携
  - 複数のAWS Batchを使った集計処理
- サーバレスなので運用不要でワークフローが組める

AWSでワークフローを組むケースではかなりおすすめ

サーバレスでバッチ処理をするメリット

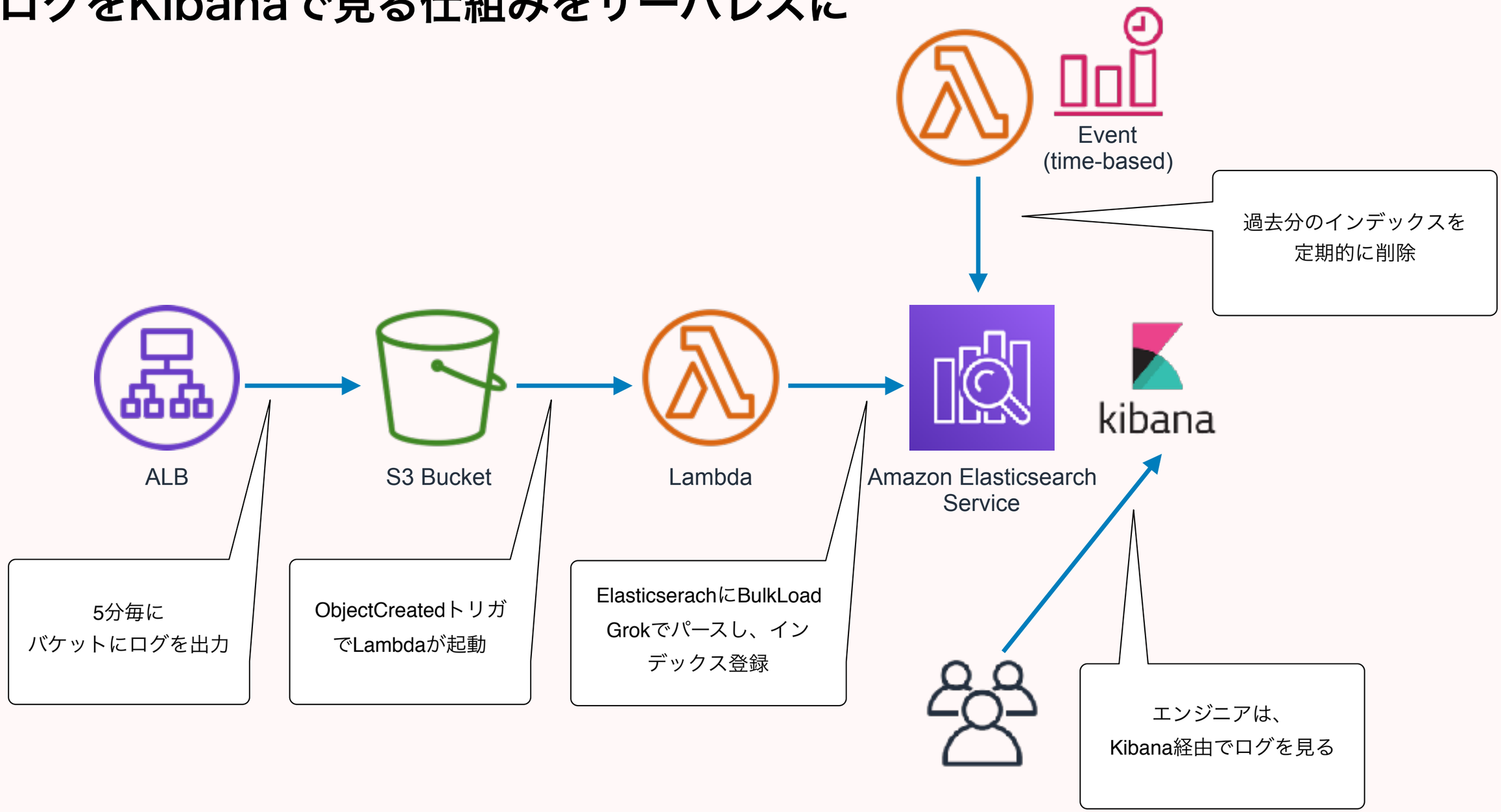
- リソースを専有出来るのでコンピューティングリソースを多く利用する処理に向いている
  - 大規模データの集計処理や機械学習の前処理
  - 使った分だけ課金されるのでコスト効率が高い
- 他の処理を考慮した実行計画が不要
  - コンピューティングリソース面の考慮
  - 手を入れるアプリケーションだけを考えれば良いので開発や試験がしやすい

サーバレスでバッチ処理をするメリット

- リソースを専有出来るのでコンピューティングリソースを多く利用する処理に向いている
  - 大規模データの集計処理や機械学習の前処理
  - 使った分だけ課金されるのでコスト効率が高い
- 他の処理を考慮した実行計画が不要
  - コンピューティングリソース面の考慮
  - 手を入れるアプリケーションだけを考えれば良いので開発や試験がしやすい

サーバを意識した運用が不要なので
実行する処理のことだけに専念出来る

ALBログをKibanaで見る仕組みをサーバレスに



mamari

これらの事例から考える
AWSでバッチ実行を考える時の方針

AWSでバッチ実行を考える時の方針

- **まずAWSのマネージドサービスで実現出来ないか考える**

  →もしかしたらバッチ処理でなくてもよい可能性も&必然的にサーバレス構成

- **Lambda(FaaS)で実現出来ないかを考える** 

  →ランタイムの管理が不要なのでやりたい処理のことだけに専念出来る

  →サーバレスかつAWSリソースと柔軟な連携が可能

- **独自のランタイムを使ってバッチ処理がしたい**  

  →アプリケーションのコンテナ化

  →**ECS Schedule Task** or **AWS Batch**を使う(k8sという選択肢も出てくる)

- **AWS上の処理でワークフローを組みたい** 

  →**StepFunctions**を使う

AWSでバッチ実行を考える時の方針

- まずAWSのマネージドサービスで実現出来ないか考える
  →もしかしたらバッチ処理でなくてもよい可能性も&必然的にサーバレス構成
- Lambda(FaaS)で実現出来ないかを考える
  →ランタイムの管理不要なのでやりたい処理のコードだけに専念可能
  →サーバレスかつAWSリソースと柔軟な連携が可能
- 独自のランタイムを使ってバッチ処理がしたい
  →アプリケーションのコンテナ化
  →**ECS Schedule Task** or **AWS Batch**を使う(k8sという選択肢も出てくる)
- AWS上の処理でワークフローを組みたい
  →**StepFunctions**を使う

運用=バッチ実行の責務を
できるだけAWSに寄せていくことで
本来価値のある開発に注力するのが大事

mamari

AWSでバッチ実行を考える時の方針

- まずAWSのマネージドサービスで実現出来ないか考える
  →もしかしたらバッチ処理でなくてもよい可能性も&必然的にサーバレス構成
- Lambda(FaaS)で実現出来ないかを考える
  →ランタイムの管理が不要なのでやりたい処理がシンプルに書き出来る
  →サーバレスかつAWSリソースと柔軟な連携が可能
- 独自のランタイムを使ってバッチ処理がしたい
  →アプリケーションのコンテナ化
  →**ECS Schedule Task** or **AWS Batch**を使う(k8sという選択肢も出てくる)
- AWS上の処理でワークフローを組みたい
  →**StepFunctions**を使う

Howとしてコンテナやサーバレスといった
アーキテクチャを理解し
それらを実サービスに落とし込む時に

AWSでバッチ実行を考える時の方針

- まずAWSのマネージドサービスで実現出来ないか考える
  →もしかしたらバッチ処理でなくてもよい可能性も&必然的にサーバレス構成
- Lambda(FaaS)で実現出来ないかを考える
  →ランタイムの管理が不要なのでやりたい処理のことだけに専念出来る
  →サーバレスかつAWSリソースと柔軟な連携が可能
- 独自のランタイムを使ってバッチ処理がしたい
  →オブジェクトコンテナ化
  →**ECS Schedule Task** or **AWS Batch**を使う(k8sという選択肢も出てくる)
- AWS上の処理でワークフローを組みたい
  →**StepFunctions**を使う

AWSだったり他のクラウドサービスを
いかにうまく使っていけるかが
これから更に求められてくるスキルだと思う

AWSでバッチ実行を考える時の方針

- まずAWSのマネージドサービスで実現出来ないか考える
  →もしかしたらバッチ処理でなくてもよい可能性も&必然的にサーバレス構成
- Lambda(FaaS)で実現出来ないかを考える
  →サーバレスかつAWSリソースと柔軟な連携が可能
- 独自のランタイムを使ってバッチ処理がしたい
  →アプリケーションのコンテナ化
  →ECS Schedule Task or AWS Batchを使う(k8sという選択肢も出てくる)
- AWS上の処理でワークフローを組みたい
  →StepFunctionsを使う

進化の早い各サービスをキャッチアップし
実際に手を動かして武器を増やすことが大事
クラウドネイティブな思想へ

さいごに宣伝

コネヒトではエンジニア積極採用中です！

興味がある方、もっと話を聞いてみたいと思う方がいましたら
声をかけていただくか、@shnagai までご連絡ください

mamari

DEV DAY

Thank you!

永井 勝一郎

コネヒト株式会社