

ある SRE チームの挑戦 運用6年目の大規模ゲームを AWS 移設後に 安定運用するための技術と今後の展望

石原 裕之
TeamManager
Sumzap Inc./SRE Team

吉岡 賢
TechLead
Sumzap Inc./SRE Team

アジェンダ

自己紹介/会社概要

AWS への移設の背景

旧環境での課題と AWS での解決方法

今後の展望

アジェンダ

自己紹介/会社概要

AWS への移設の背景

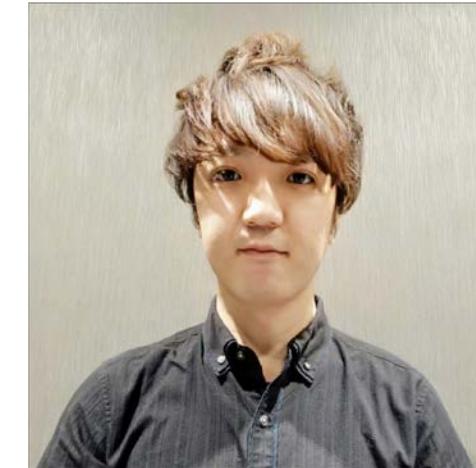
旧環境での課題と AWS での解決方法

今後の展望

自己紹介



株式会社サムザップ
SRE チーム
TeamManager
石原 裕之



株式会社サムザップ
SRE チーム
TechLead
吉岡 賢

業務内容

2014年12月 サイバーエージェント 入社
サムザップに出向し、SRE チーム責任者としてチームマネジメントと全サービスのシステム運用、改善業務に従事。
好きな AWS サービスは Amazon Aurora

業務内容

2016年4月 サイバーエージェント 入社
サムザップに出向し、AWS 移設の設計・運用・開発を担当。
現在も開発と運用両方のシステム改善に注力している。
好きな AWS サービスは Route 53

サムザップの SRE チーム

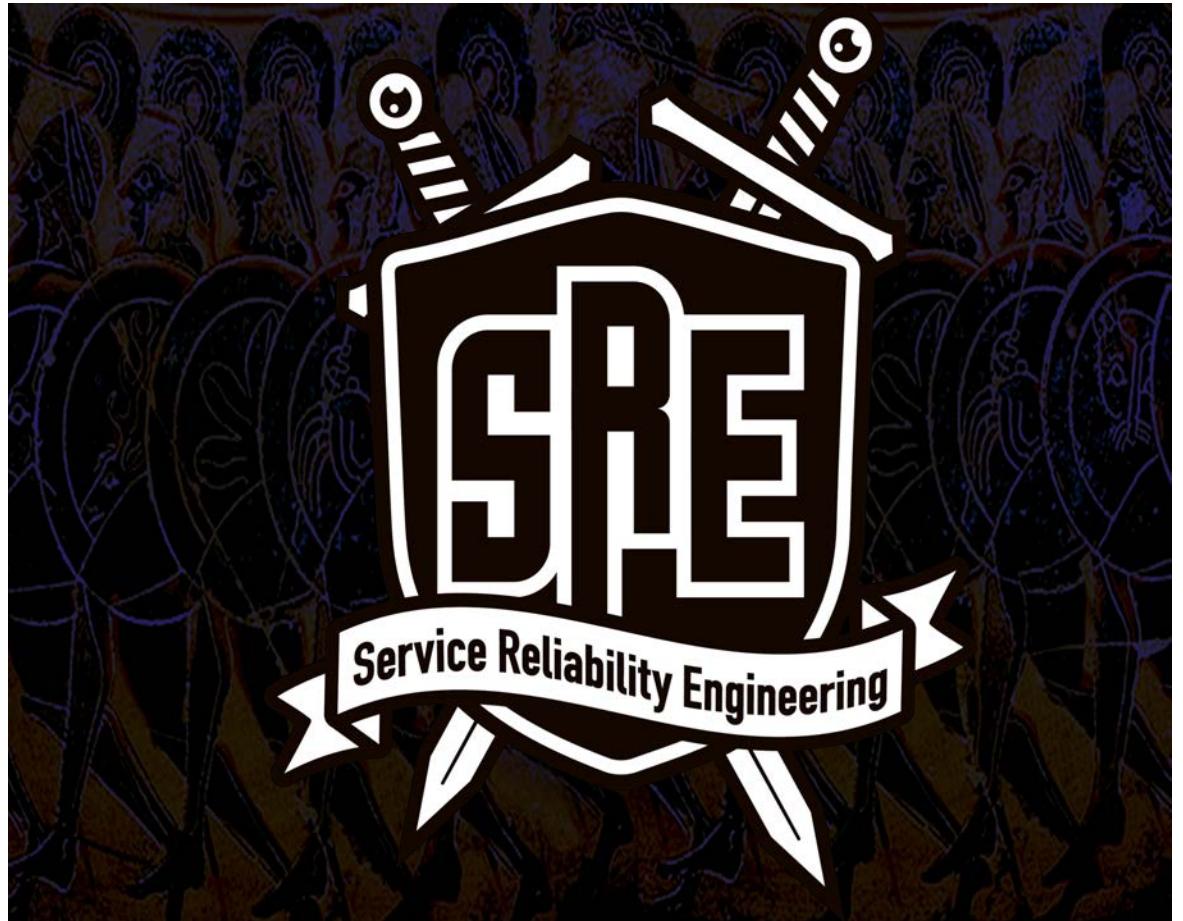
2018年4月発足、現在6名

サムザップのゲームをユーザーのみなさまにいつも快適にプレイしてもらえるよう、日々サービスの信頼性を高める活動をしている

<業務内容>

インフラの設計、構築、日々のオペレーションの自動化、効率化、コード化、オンコール対応、モニタリング環境の構築、パフォーマンスチューニング、開発フロー、デプロイ、モニタリング環境の整備…etc

ゲームの機能開発は行わないが、安定性の向上やパフォーマンス改善等につながる開発は行う



会社概要

株式会社サムザップ

2009年5月 CyberAgent 子会社として設立

事業内容

スマートフォン向けゲームアプリの企画・運営・配信



© 2019, Amazon Web Services, Inc. or its affiliates. All rights reserved.



提供サービス



リンクスリングス
2019年5月配信開始
4人対4人の陣取りアクションゲーム



戦国炎舞 - KIZNA -
2013年4月配信開始
戦国時代を舞台にしたカードバトルゲーム
1日3回20人対20人のリアルタイムバトル
「合戦」が行われる

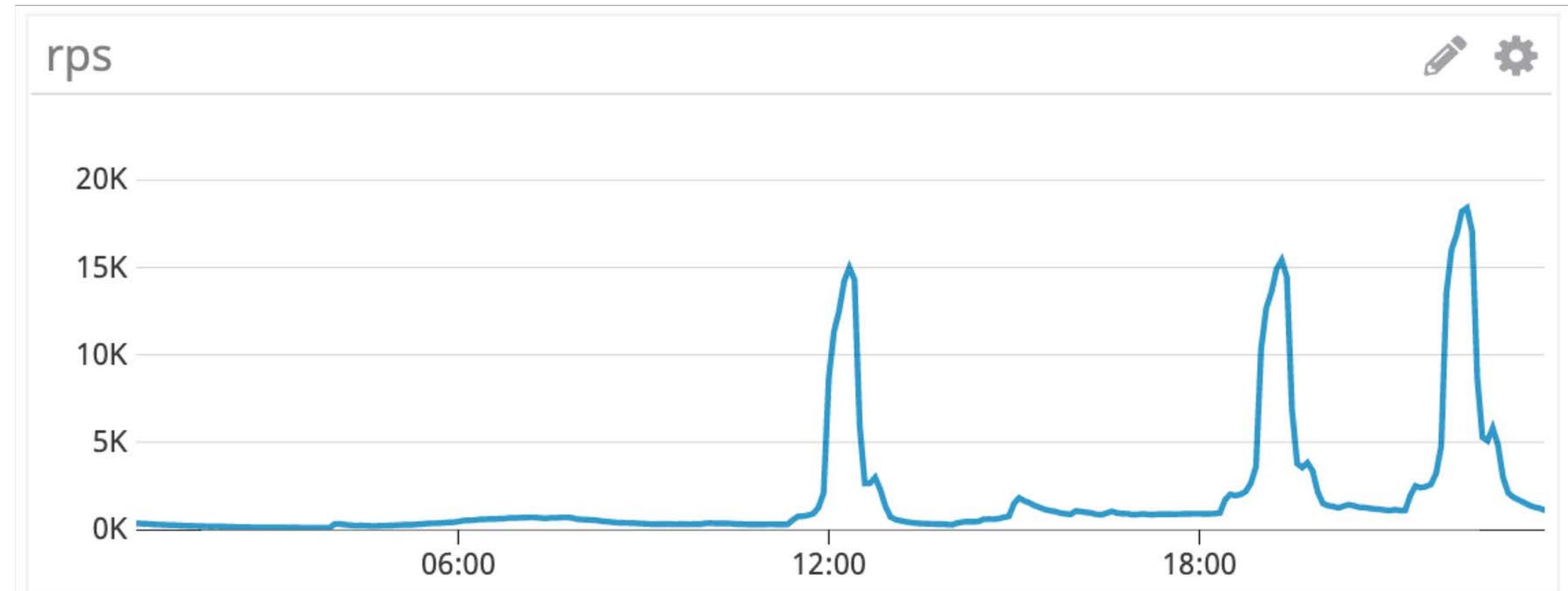
戦国炎舞 -KIZNA-

- 560万ダウンロード突破
- 2019年2月
プライベートクラウドから
AWS へ移設
- 200+ instances (peak)
- 23,000 req/sec (peak)

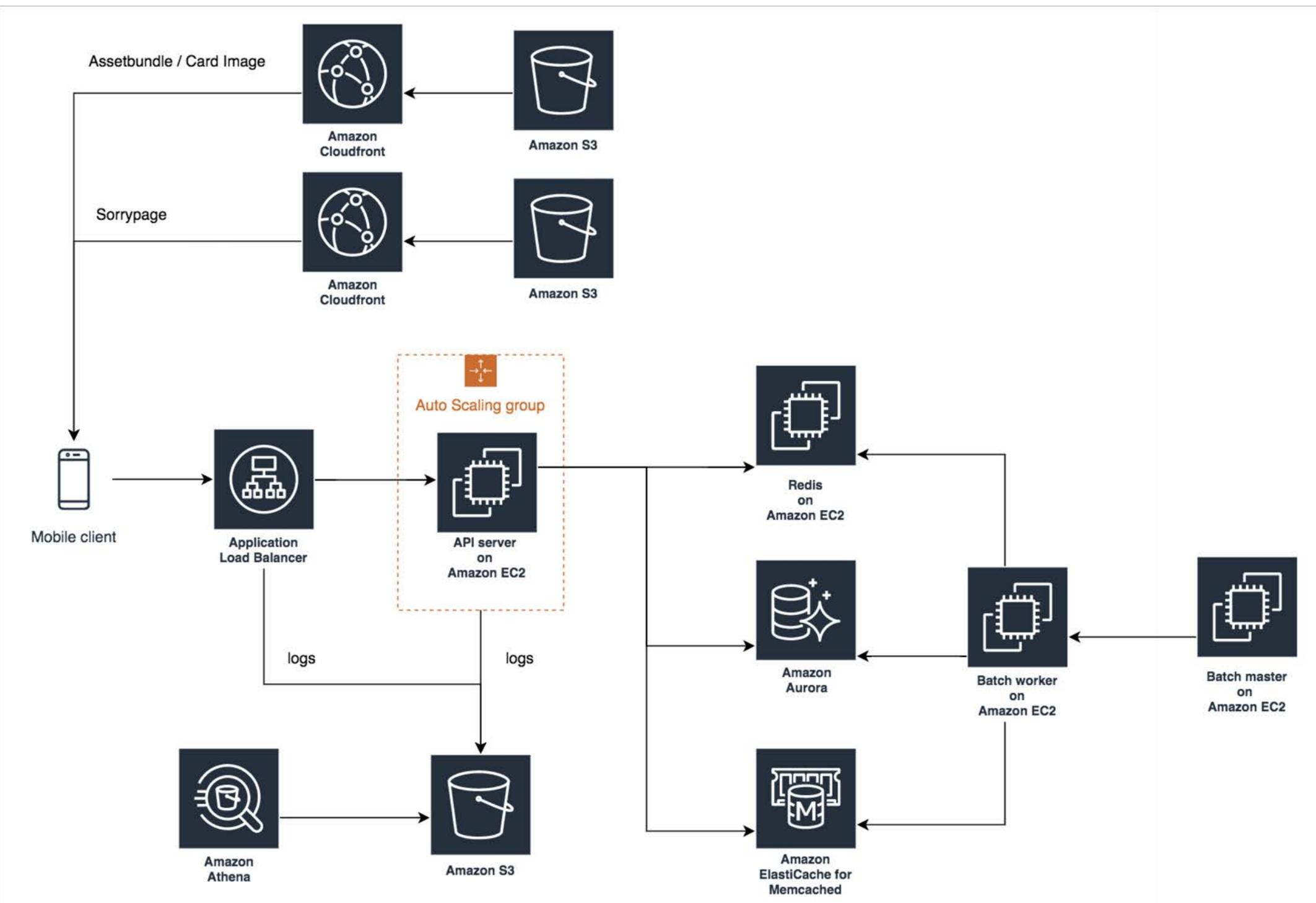


ユーザーアクセスの特徴

1日3回の合戦にアクセスが集中



構成図



アジェンダ

自己紹介/会社概要

AWS への移設の背景

旧環境での課題と AWS での解決方法

今後の展望

プライベートクラウドの提供終了

サムザップで利用していた CyberAgent のプライベートクラウドの一部ゾーンが、ネットワーク機器の EOL 等の理由により2019年12月での提供終了が決定

2019年末までに別環境へ移設する必要があった

なぜ AWS なのか？

最大の理由は高性能なマネージド DB サービス Aurora の存在
旧環境で苦労した DB サーバーのトラブルを減らしたかった

アジェンダ

自己紹介/会社概要

AWS への移設の背景

旧環境での課題と AWS での解決方法

今後の展望

旧環境での課題

- 合戦前後の柔軟なサーバー増減
- 肥大化したデータとログ
- 自動化しづらい構成と環境

旧環境での課題

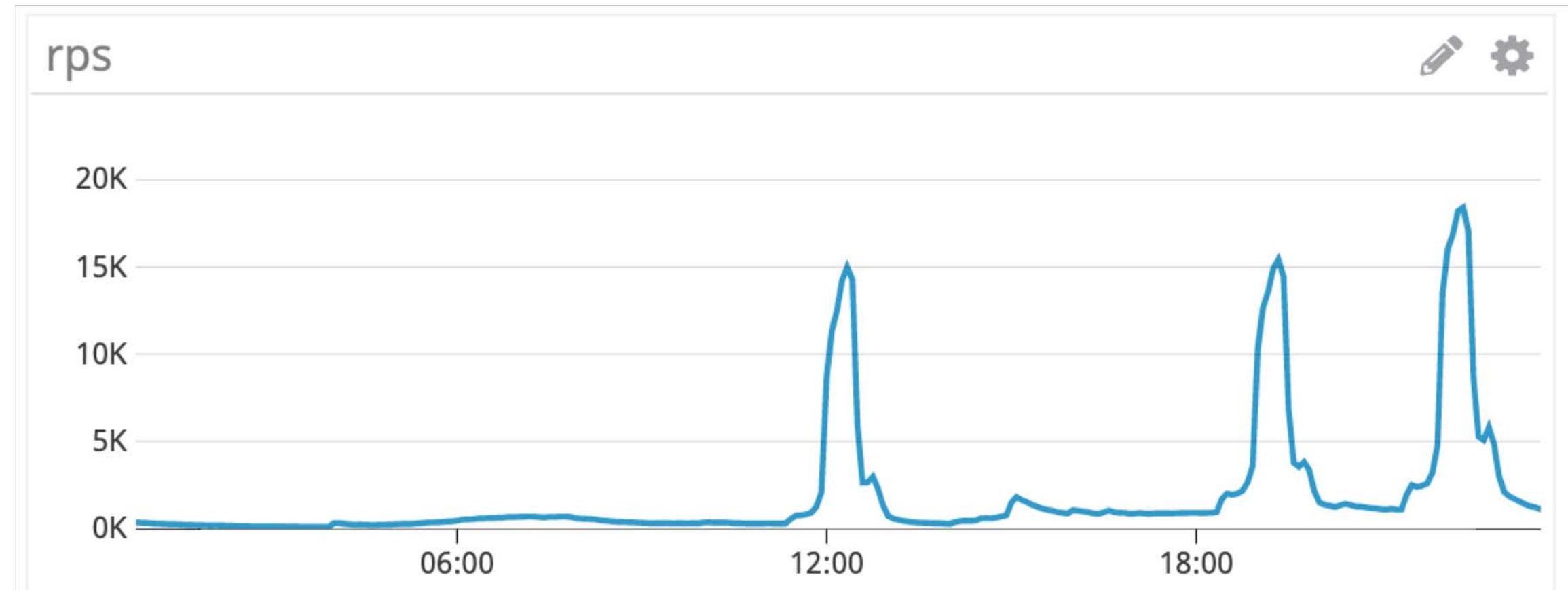
➤ 合戦前後の柔軟なサーバー増減

➤ 肥大化したデータとログ

➤ 自動化しづらい構成と環境

合戦前後の柔軟なサーバー増減

コスト最適化のためにアクセスが集中する合戦の前後で Web サーバを増減させたい

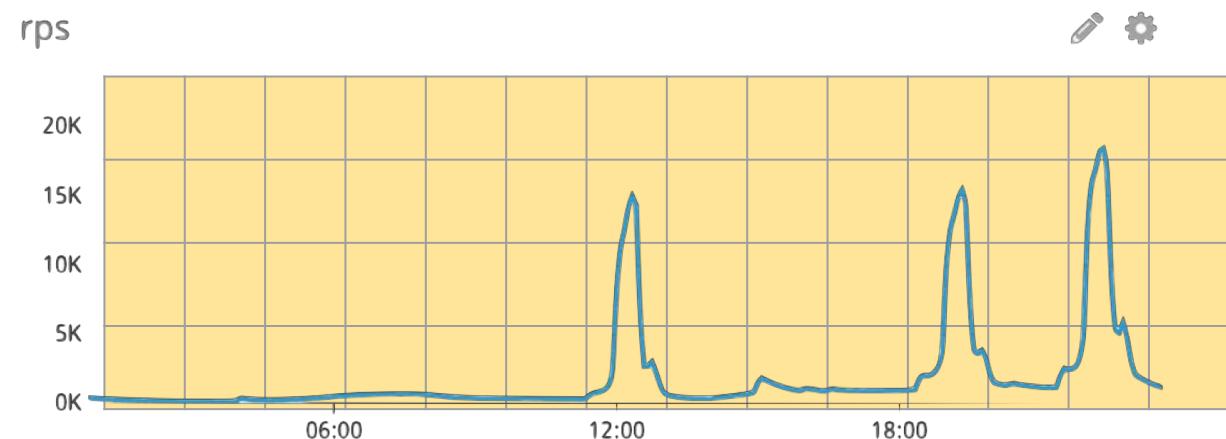
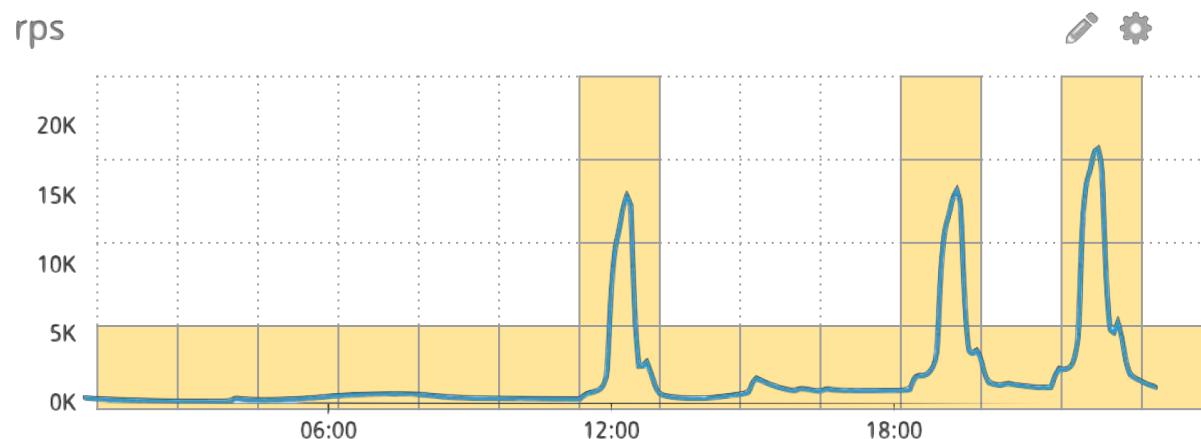


旧環境では実現できなかった

- マシンイメージでのインスタンス作成に非対応
- 構築開始からサービス投入までに1時間かかる
- API エラーでインスタンス作成失敗が多発

理想：合戦直前に Web サーバを追加、
合戦後に廃棄(コストが1/3になる)

現実：ピーク時間のリクエスト処理に必要な
Web サーバ台数を常に起動



移設後は AWS の機能を使って実現

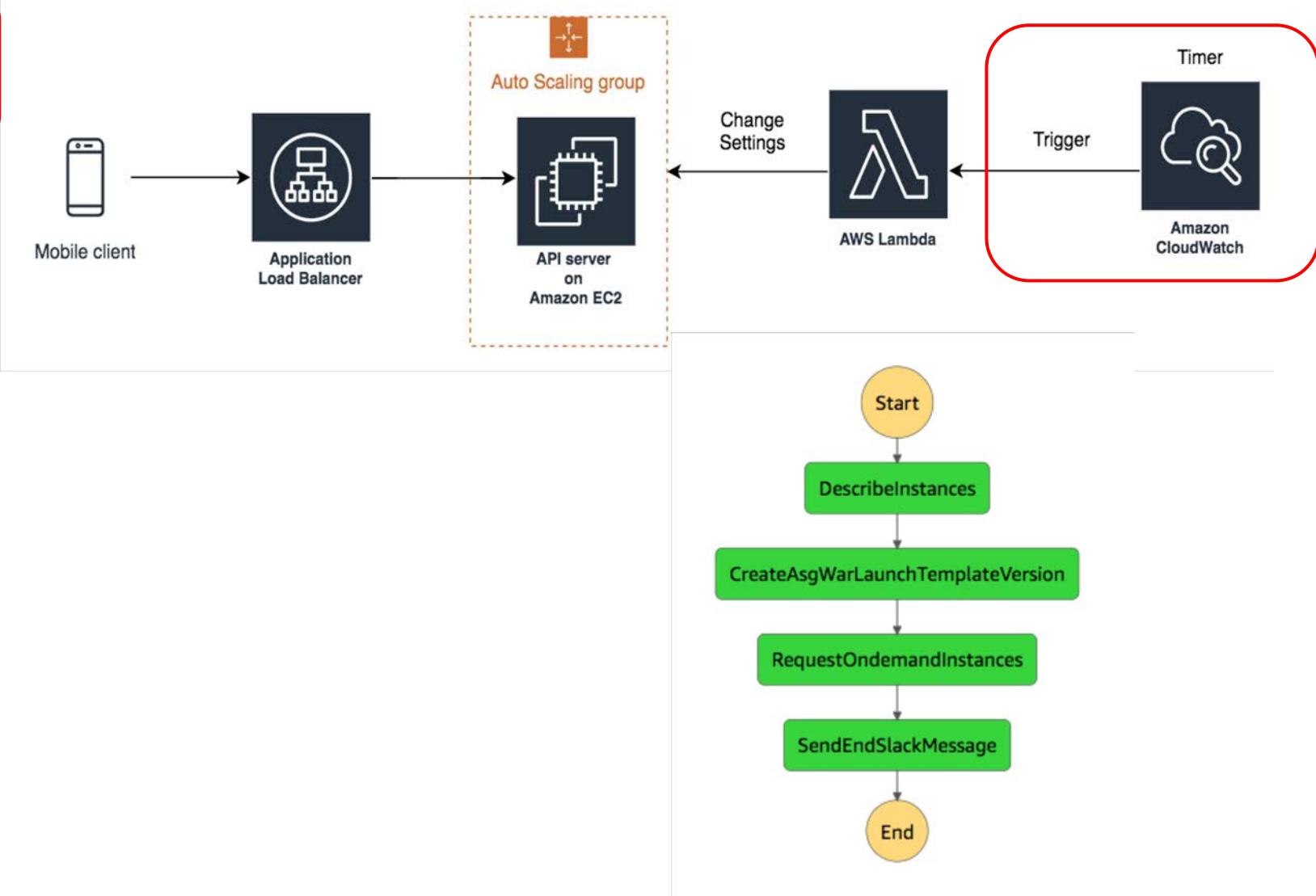
1) CloudWatch Eventsで時限トリガー

2) Step Functions を呼び出すLambda
が実行される

2-1) 現在のインスタンス数を確認
2-2) 合戦用の Auto Scaling グループに増設/停止をスケジューリング
2-3) slack へ増設の開始を通知

3) 増設されたインスタンスが
自動でサービスインする

4) アプリケーションからみて、
何も変化がないまま負荷分散が可能



移設後は AWS の機能を使って実現

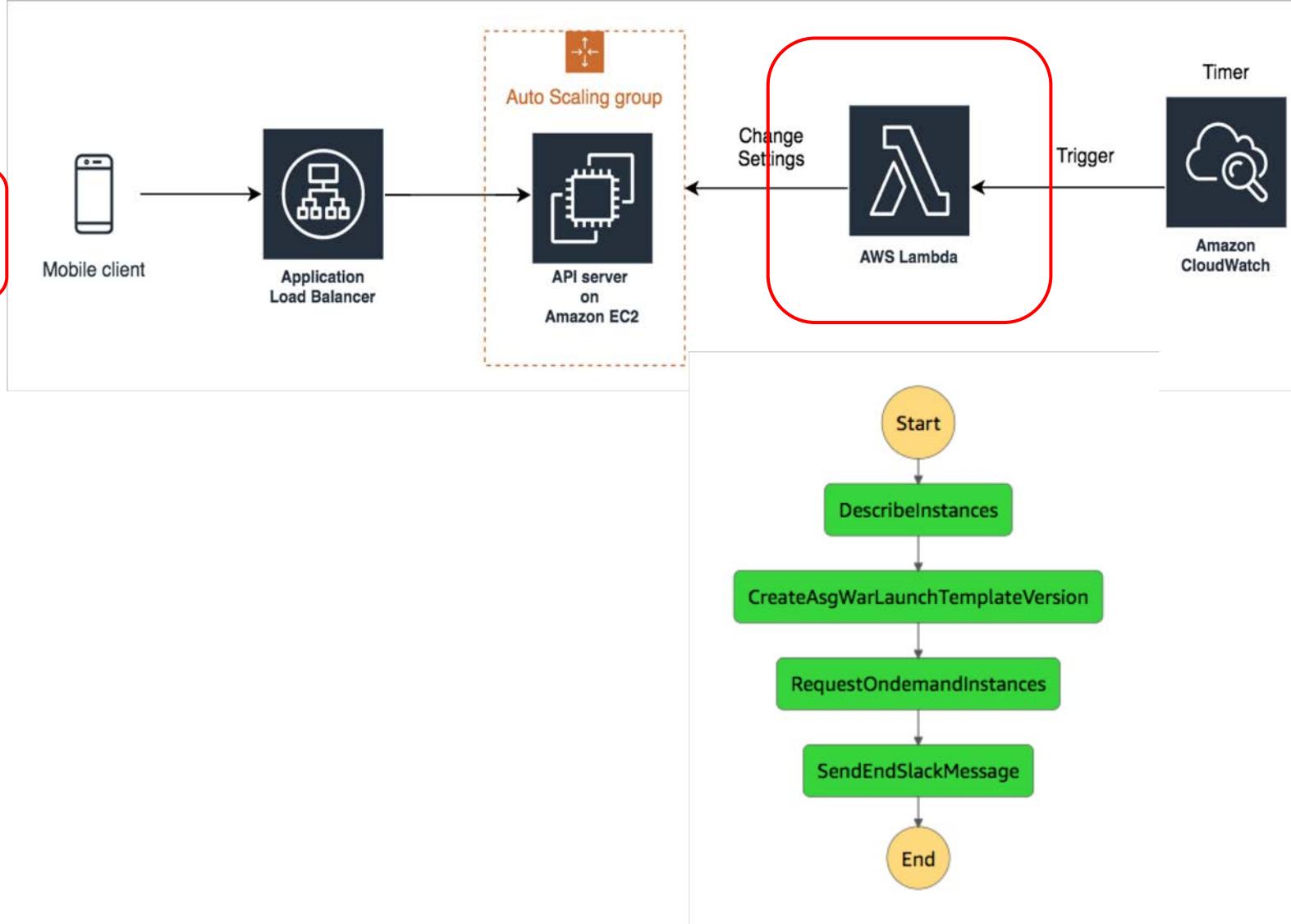
1) CloudWatch Eventsで時限トリガー

2) Step Functions を呼び出すLambda
が実行される

- 2-1) 現在のインスタンス数を確認
- 2-2) 合戦用の Auto Scaling グループに増設/停止をスケジューリング
- 2-3) slack へ増設の開始を通知

3) 増設されたインスタンスが
自動でサービスインする

4) アプリケーションからみて、
何も変化がないまま負荷分散が可能



移設後は AWS の機能を使って実現

1) CloudWatch Eventsで時限トリガー

2) Step Functions を呼び出すLambda
が実行される

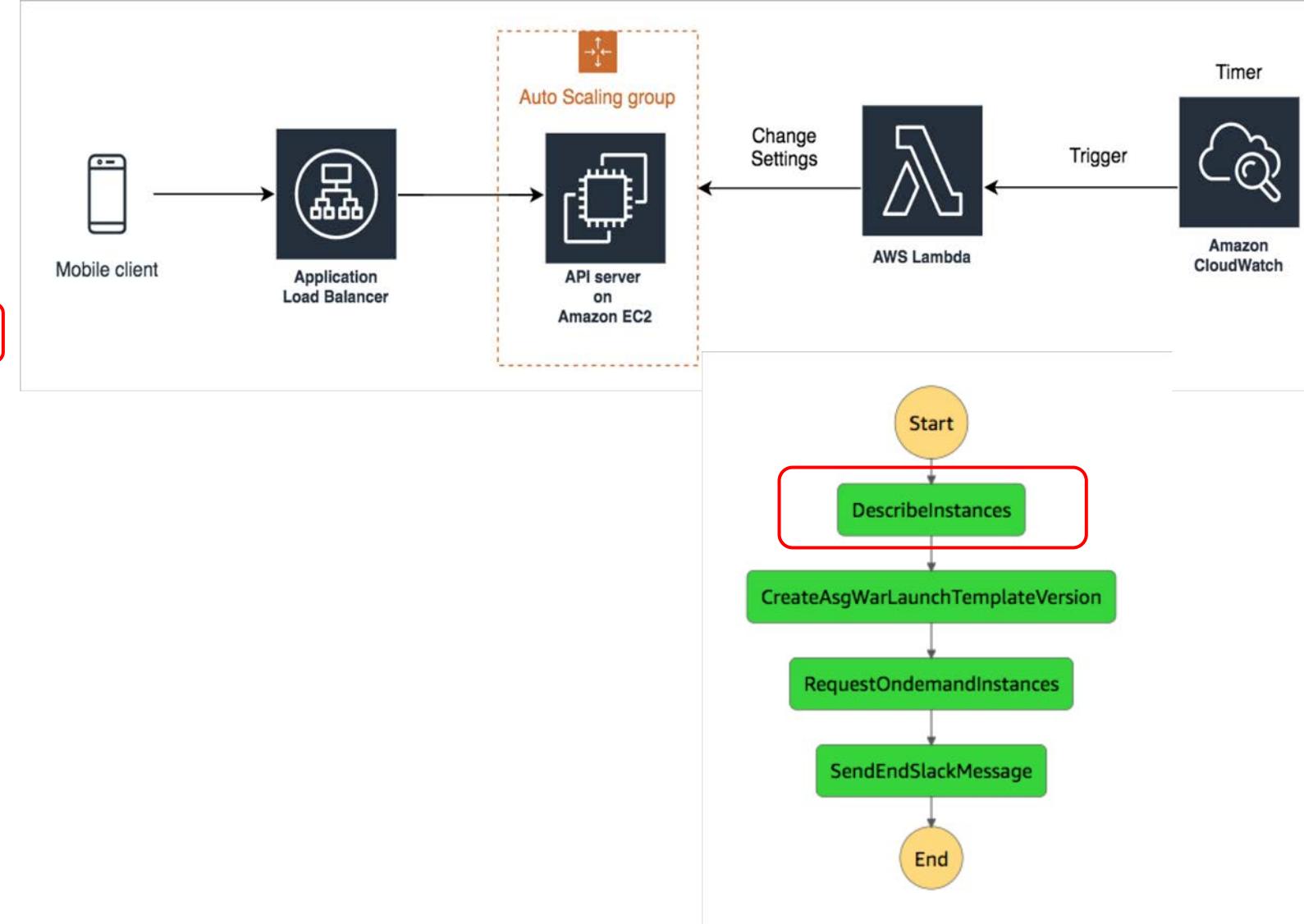
2-1) 現在のインスタンス数を確認

2-2) 合戦用の Auto Scaling グループに増設/停止をスケジューリング

2-3) slack へ増設の開始を通知

3) 増設されたインスタンスが
自動でサービスインする

4) アプリケーションからみて、
何も変化がないまま負荷分散が可能



移設後は AWS の機能を使って実現

1) CloudWatch Eventsで時限トリガー

2) Step Functions を呼び出すLambda
が実行される

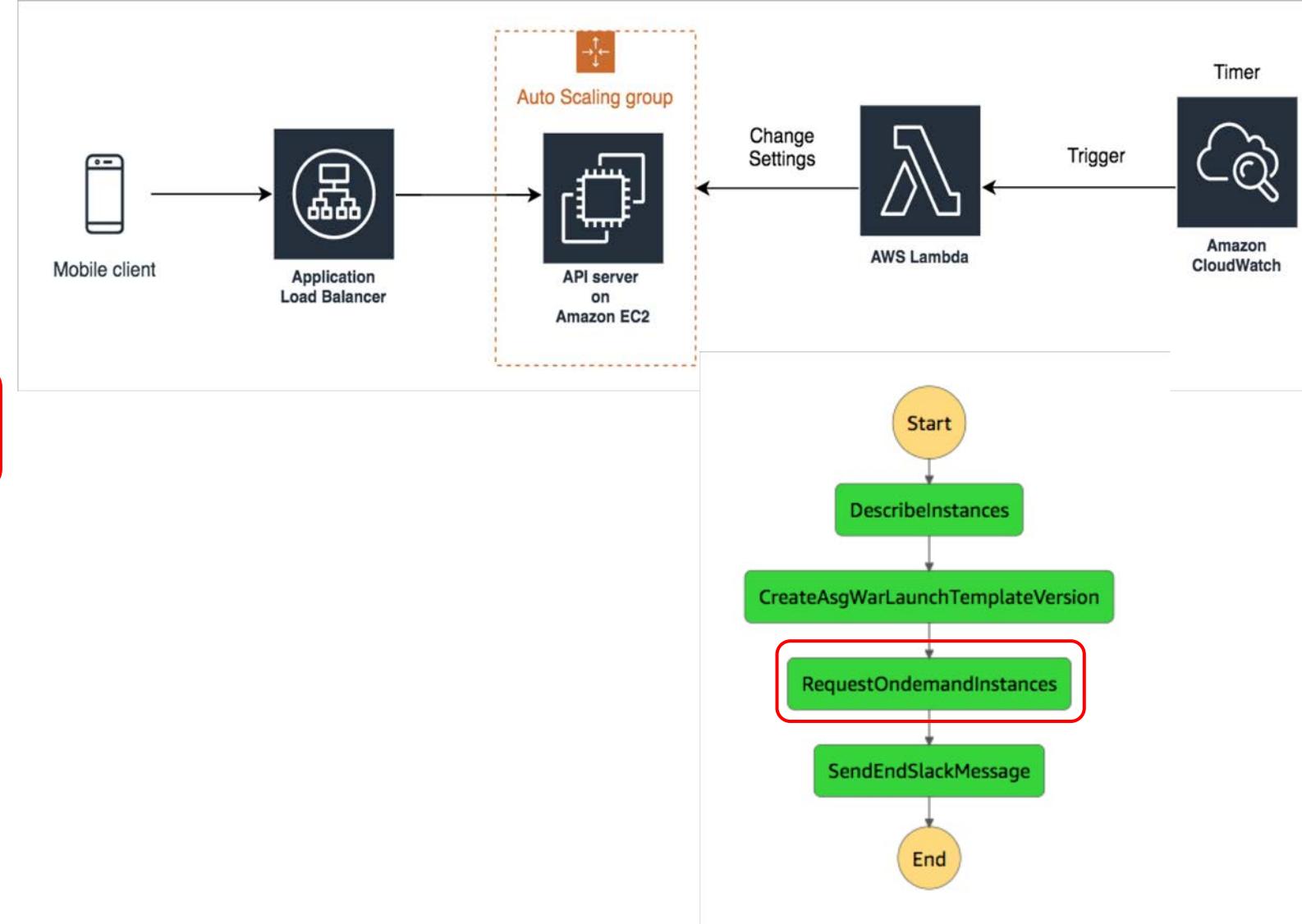
2-1) 現在のインスタンス数を確認

2-2) 合戦用の Auto Scaling グループに増設/停止をスケジューリング

2-3) slack へ増設の開始を通知

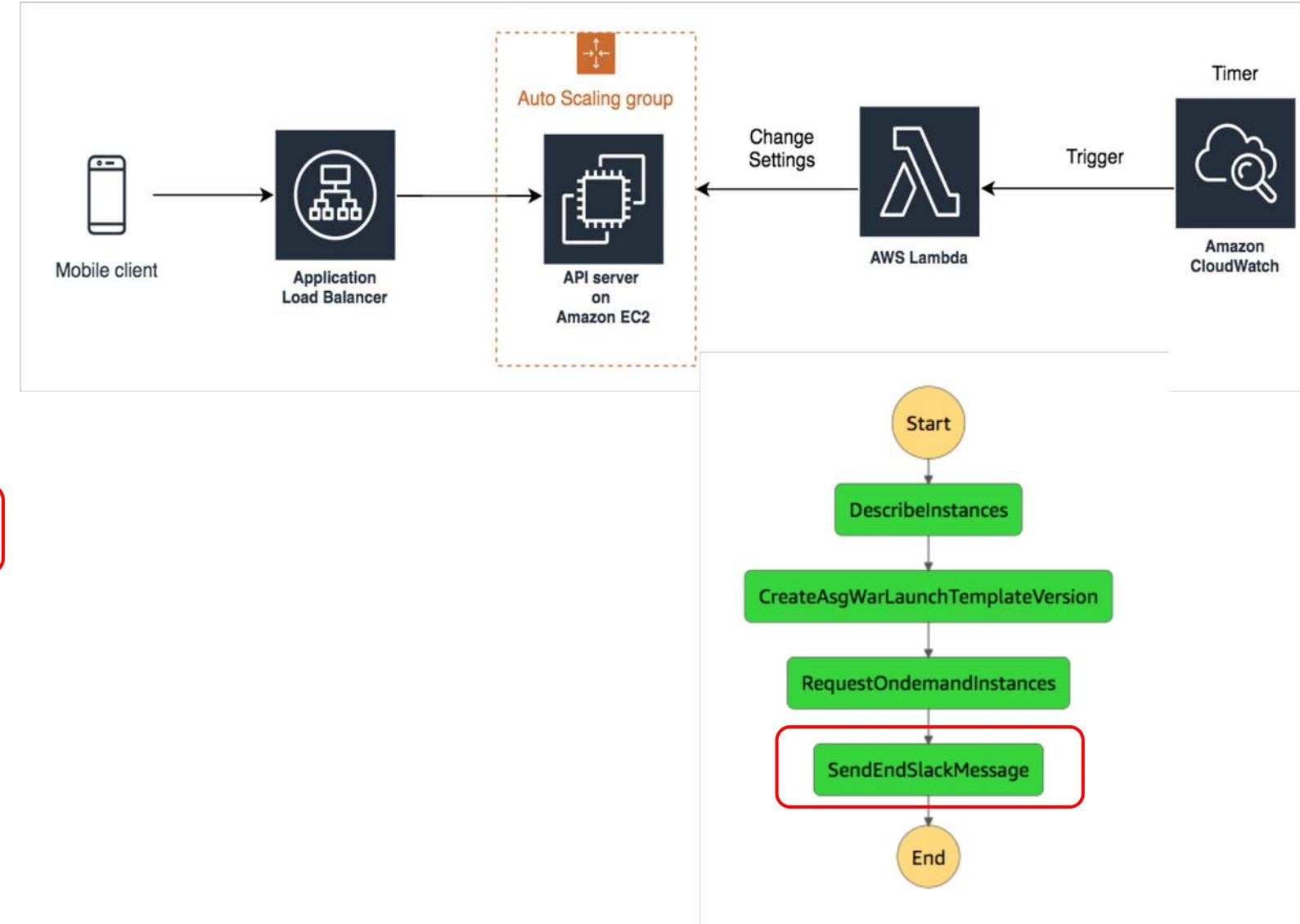
3) 増設されたインスタンスが
自動でサービスインする

4) アプリケーションからみて、
何も変化がないまま負荷分散が可能



移設後は AWS の機能を使って実現

- 1) CloudWatch Eventsで時限トリガー
- 2) Step Functions を呼び出すLambda
が実行される
 - 2-1) 現在のインスタンス数を確認
 - 2-2) 合戦用の Auto Scaling グループに増設/停止をスケジューリング
 - 2-3) slack へ増設の開始を通知
- 3) 増設されたインスタンスが
自動でサービスインする
- 4) アプリケーションからみて、
何も変化がないまま負荷分散が可能



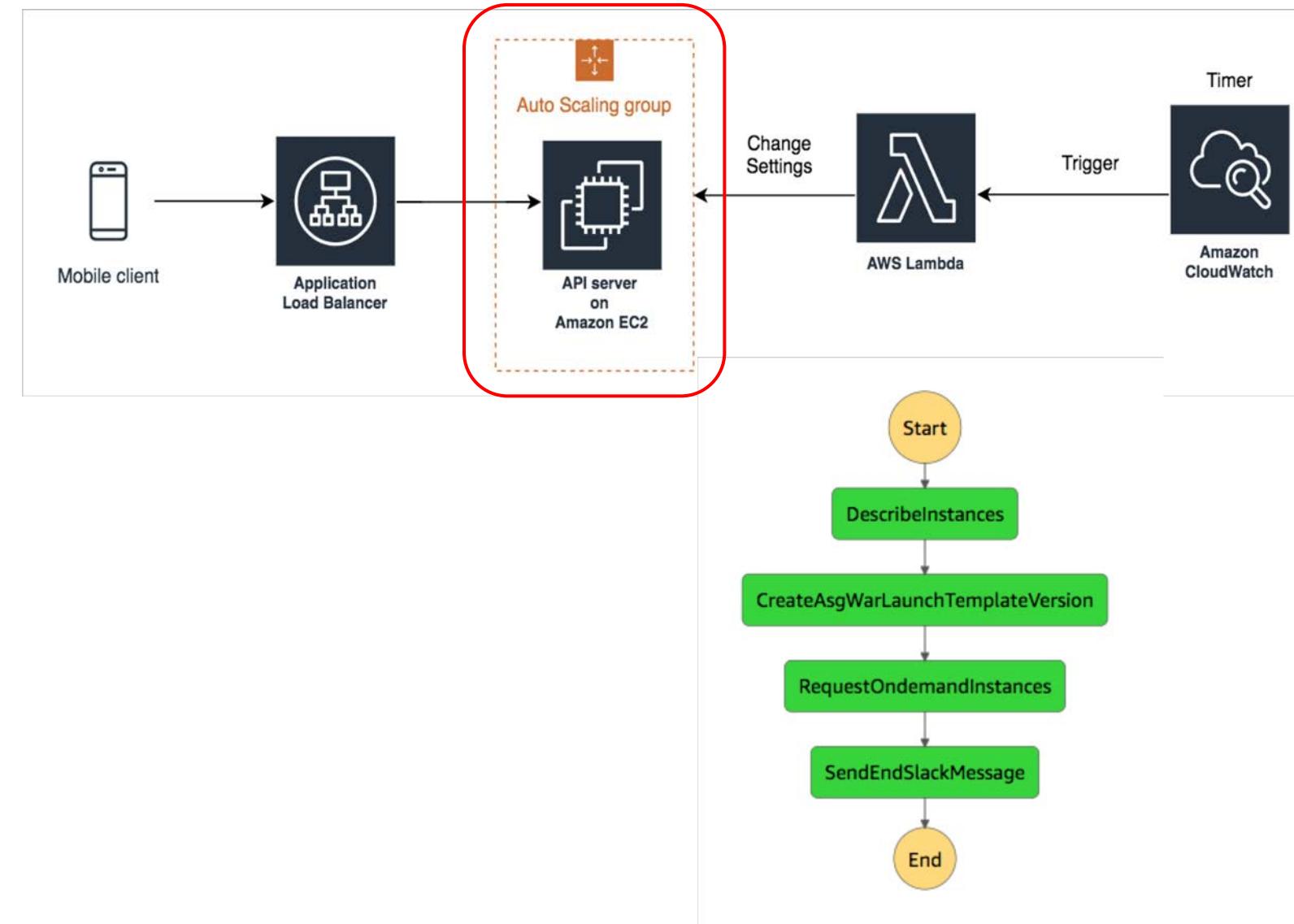
移設後は AWS の機能を使って実現

- 1) CloudWatch Eventsで時限トリガー
- 2) Step Functions を呼び出すLambda
が実行される

- 2-1) 現在のインスタンス数を確認
- 2-2) 合戦用の Auto Scaling グループに増設/停止をスケジューリング
- 2-3) slack へ増設の開始を通知

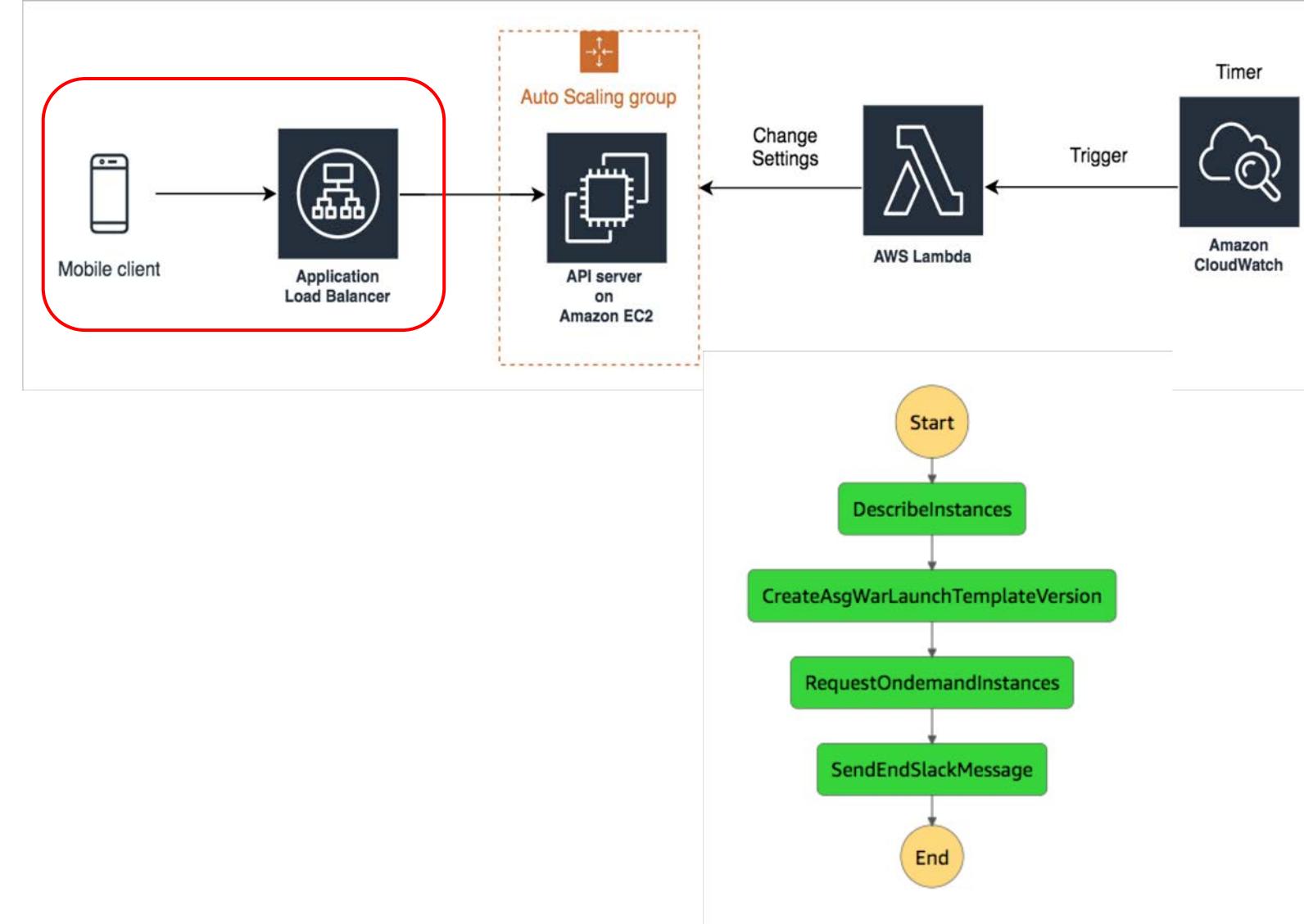
- 3) 増設されたインスタンスが
自動でサービスインする

- 4) アプリケーションからみて、
何も変化がないまま負荷分散が可能

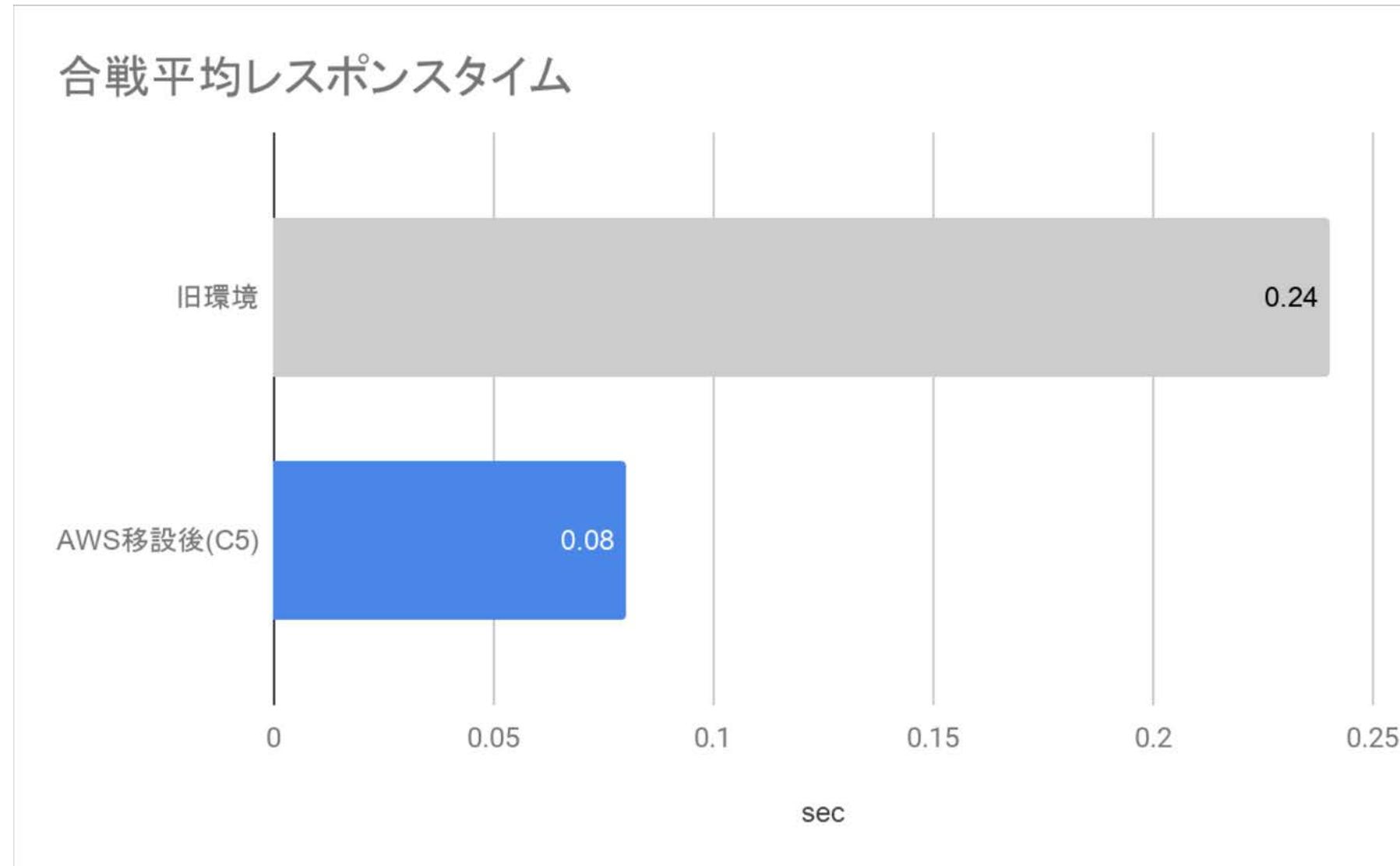


移設後は AWS の機能を使って実現

- 1) CloudWatch Eventsで時限トリガー
- 2) Step Functions を呼び出すLambda
が実行される
 - 2-1) 現在のインスタンス数を確認
 - 2-2) 合戦用の Auto Scaling グループに増設/停止をスケジューリング
 - 2-3) slack へ増設の開始を通知
- 3) 増設されたインスタンスが
自動でサービスインする
- 4) アプリケーションからみて、
何も変化がないまま負荷分散が可能



マシン性能が上がり、応答速度アップ



Spot Instance

今後は、さらなるコスト最適化に向けてスポットインスタンスを導入を進めています。

現在、本番環境の一部 Web サーバーで試験中で今月末から来月にかけて本格導入予定

旧環境での課題

- 合戦前後の柔軟なサーバー増減
- 肥大化したデータとログ
- 自動化しづらい構成と環境

肥大化したデータ、ログ

- › DB サーバーの運用、管理に疲弊
 - Percona Server for MySQL 5.5 全78台
 - ディスク容量不足によるサーバー増設
 - サーバートラブル時の対応
- › 手間のかかるアプリケーションログ調査
 - gzip圧縮した状態で36GB/day
 - プライベートクラウドのオブジェクトストレージに格納
 - 調査の度にログファイルのダウンロードが必要
 - 調査に數十分、範囲によっては数時間かかることも

Auroraに移行することで解決

› DB サーバーの運用、管理に疲弊

- Percona Server for MySQL 5.5 全78台 → 最大5倍の性能。複数DBを集約できそう
- ディスク容量不足によるサーバー増設 → ディスクは64TBまで自動拡張
- サーバートラブル時の対応 → 障害時は自動でfailover、再構築が行われる

› 手間のかかるアプリケーションログ調査

- gzip圧縮した状態で36GB/day
- プライベートクラウドのオブジェクトストレージに格納
- 調査の度にログファイルのダウンロードが必要
- 調査に數十分、範囲によっては数時間かかることも

Amazon Aurora MySQLへの移行と集約

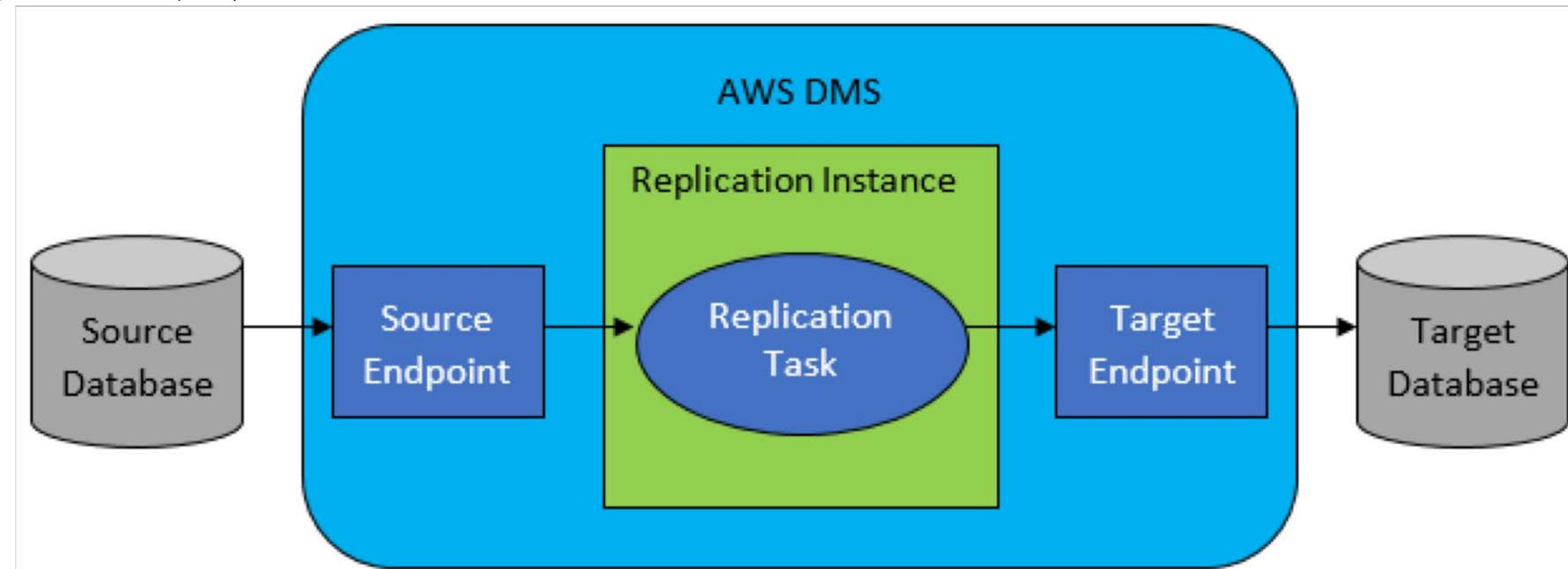
本番同等リクエスト数での負荷試験の結果、以下のスペック、
インスタンス数に決定

- db.r4.8xlarge × 8クラスタ(16インスタンス)
- db.r4.16xlarge × 2クラスタ(4インスタンス)

Auroraへの移行と集約方法は DMS を選択

DMS (AWS Database Migration Service)

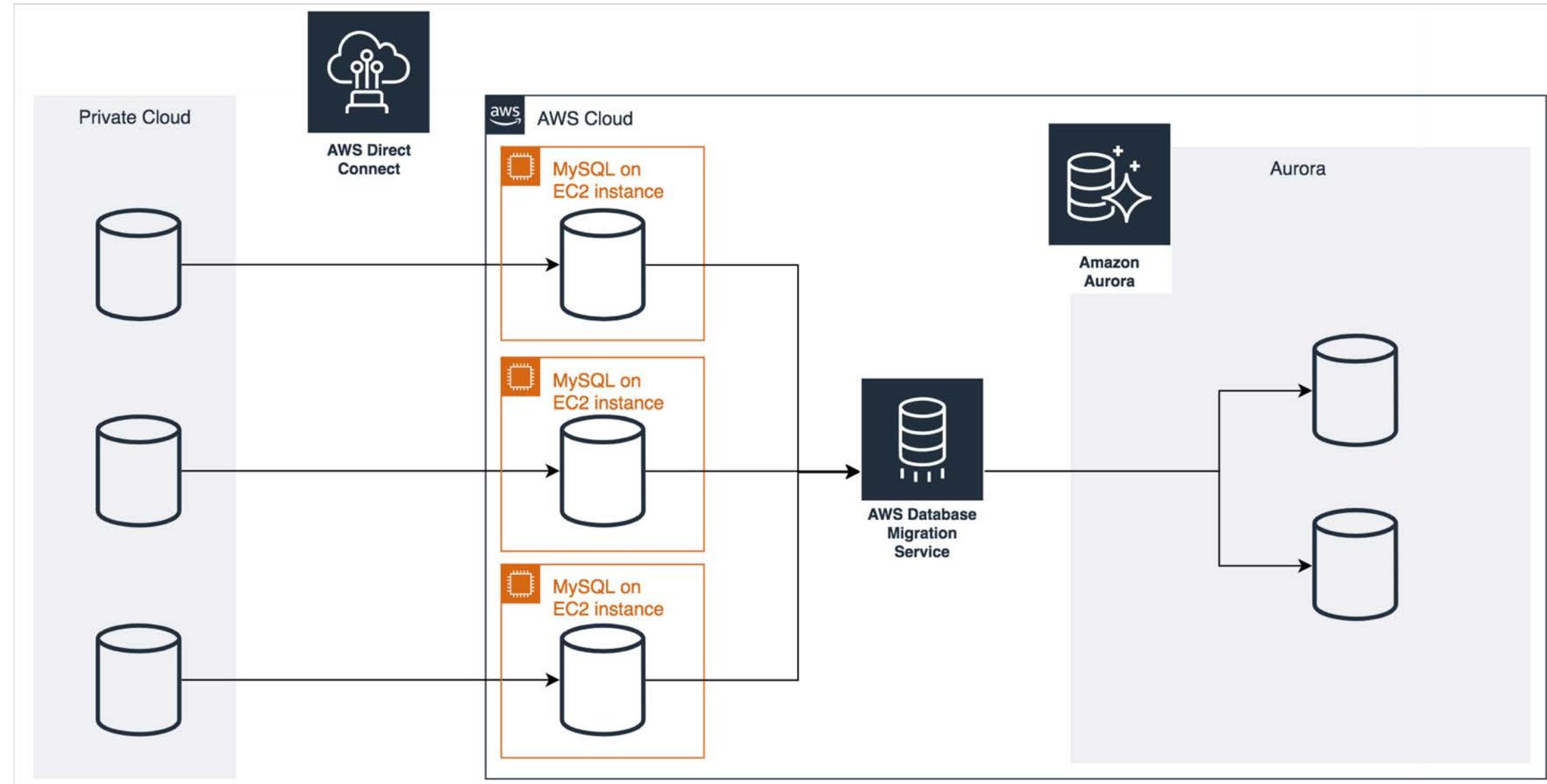
既存のデータベースを最小限のダウンタイムでマイグレーションするサービス



DMS 選択の理由

Aurora への移行と集約を同時に実現できそうな方法が DMS 以外に見つからなかった

DMS を使ったデータ移行の構成



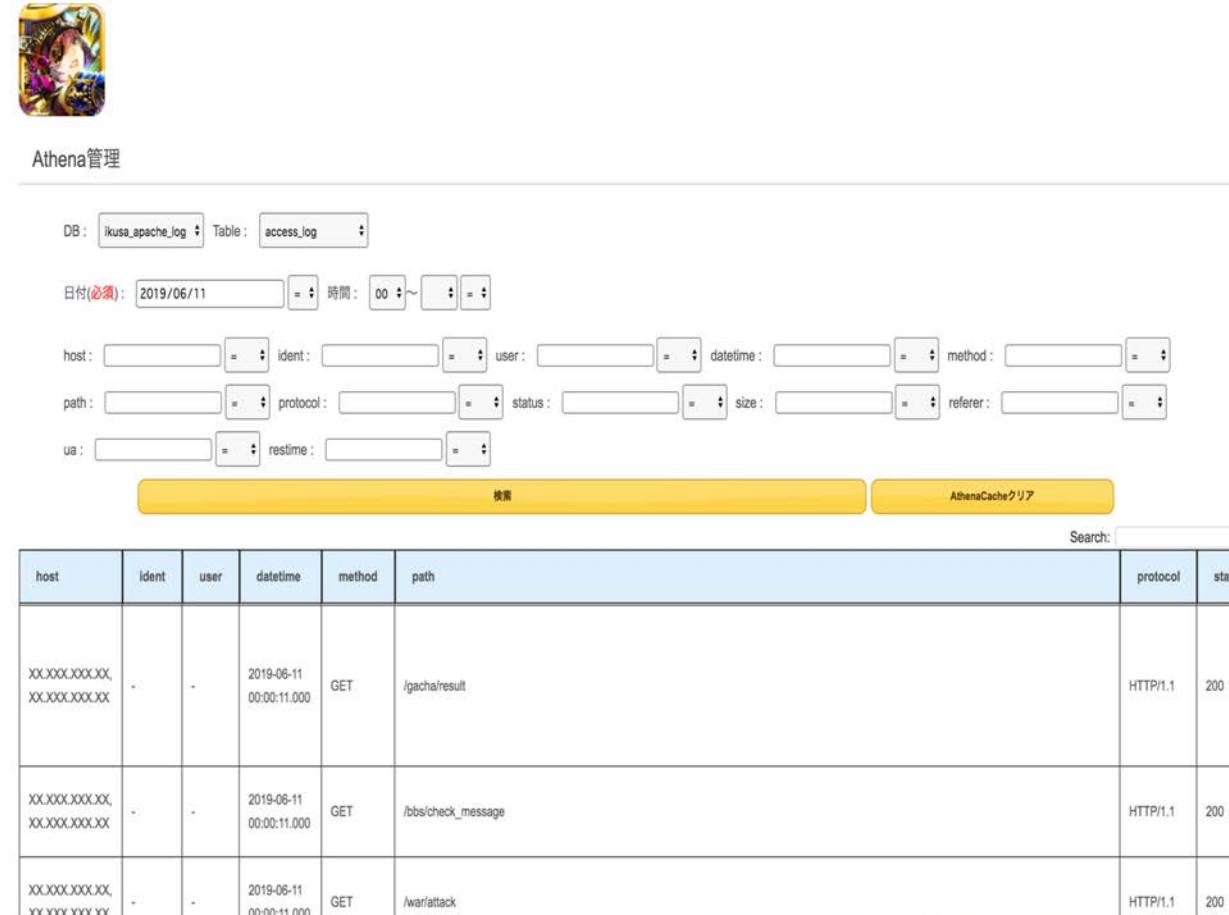
DMS 検証中に苦労したポイント

- 動作検証チャンスは1日1回
 - DB のデータ量が多く、データ転送が業務時間中に終わらない
 - 退社前に DMS タスクをセット。翌朝実行結果を確認
 - エラーが起きていたら調査して再チャレンジ。以降繰り返し
- レプリケーションインスタンス台数とスペックの調整
 - CloudWatch を見ながらスペック調整
 - dms.c4.4xlarge から最終的に dms.r4.8xlarge × 20台に落ち着く
- DMS タスク分割数の調整
 - 6億レコードあるテーブルは1つの DMS タスクだと1日経っても終わらない
 - 1億件毎にタスクを分割

肥大化したデータ、ログ

- DBサーバーの運用、管理に疲弊
 - Percona Server for MySQL 5.5 全78台(1セット3台構成を26セット)
 - ディスク容量不足によるサーバー増設
 - サーバートラブル時の対応
- 手間のかかるアプリケーションログ調査
 - gzip 圧縮した状態で36 GB/day
 - プライベートクラウドのオブジェクトストレージに格納
 - 調査の度にログファイルのダウンロードが必要
 - 調査に數十分、範囲によっては数時間かかることも

移設後はログを S3 に保存して Athena で調査



The screenshot shows the Athena Management interface. At the top, there's a logo and the text "Athena管理". Below that, a search bar has "DB: ikusa_apache_log" and "Table: access_log" selected. The search criteria include "日付(必須): 2019/06/11" and various filters for host, path, ua, etc. A yellow "検索" button is highlighted. Below the search area is a table with columns: host, ident, user, datetime, method, path, protocol, and stat. The table contains three rows of log data.

host	ident	user	datetime	method	path	protocol	stat
XX.XXX.XXX.XX, XX.XXX.XXX.XX	-	-	2019-06-11 00:00:11.000	GET	/gacha/result	HTTP/1.1	200
XX.XXX.XXX.XX, XX.XXX.XXX.XX	-	-	2019-06-11 00:00:11.000	GET	/bbs/check_message	HTTP/1.1	200
XX.XXX.XXX.XX, XX.XXX.XXX.XX	-	-	2019-06-11 00:00:11.000	GET	/war/attack	HTTP/1.1	200

- ・ログのダウンロードが不要に
- ・調査にかかる時間は数秒～数十秒
- ・SQL を使って簡単に調査
- ・SQL を書かなくても定型の調査は
社内管理ツール上からできるように
した

旧環境での課題

- 合戦前後の柔軟なサーバー増減
- 肥大化したデータとログ
- 自動化しづらい構成と環境

なぜ自動化したいのか

NoOps = No “Uncomfortable” Ops

システム運用の嬉しくないことをなくそう

NoOps Japan発起人 岡大勝さん

「トイル」をなくし、安定稼働させるための改善に時間をつかう

自動化しづらい構成と環境



外部サービスとの連携が
しづらいクローズドな環境



効率の悪いデプロイと
コストの高いデプロイ対象管理



職人技によって支えられている
マーケティング用分析基盤

どう解決したか

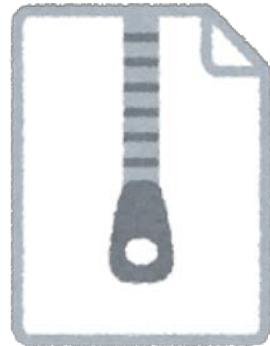
- デプロイ 自動化
- 接続先管理 自動化
- 分析基盤 効率化

どう解決したか

- デプロイ自動化
- 接続先管理自動化
- 分析基盤 効率化

デプロイ自動化

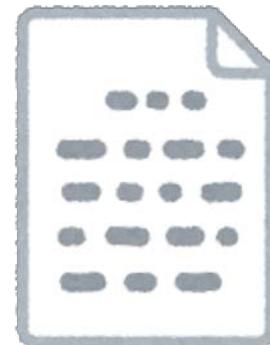
デプロイしたいファイルの種類



アセットバンドル



画像



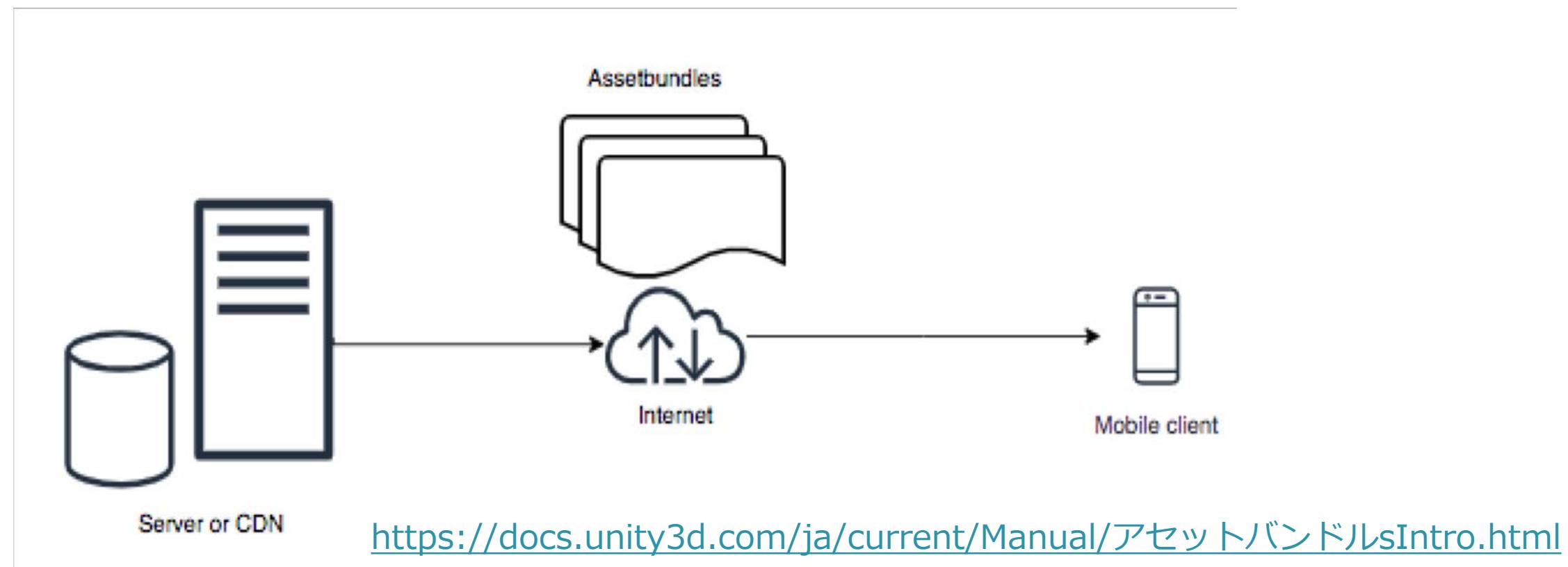
アプリケーション
ソースコード

デプロイ自動化

1. アセットバンドルデプロイ自動化
2. 画像デプロイ自動化
3. アプリケーションソースデプロイ効率化

アセットバンドル とは...

アセットバンドル はランタイムに読み込むプラットフォーム特有のアセット (モデル、テクスチャ、フレハブ、オーディオクリップ、シーン全体) を含むアーカイブファイルです。



アセットバンドルデプロイ自動化

どのアプリケーションにどのアセットバンドルが対応しているのか管理が必要

<移設前>

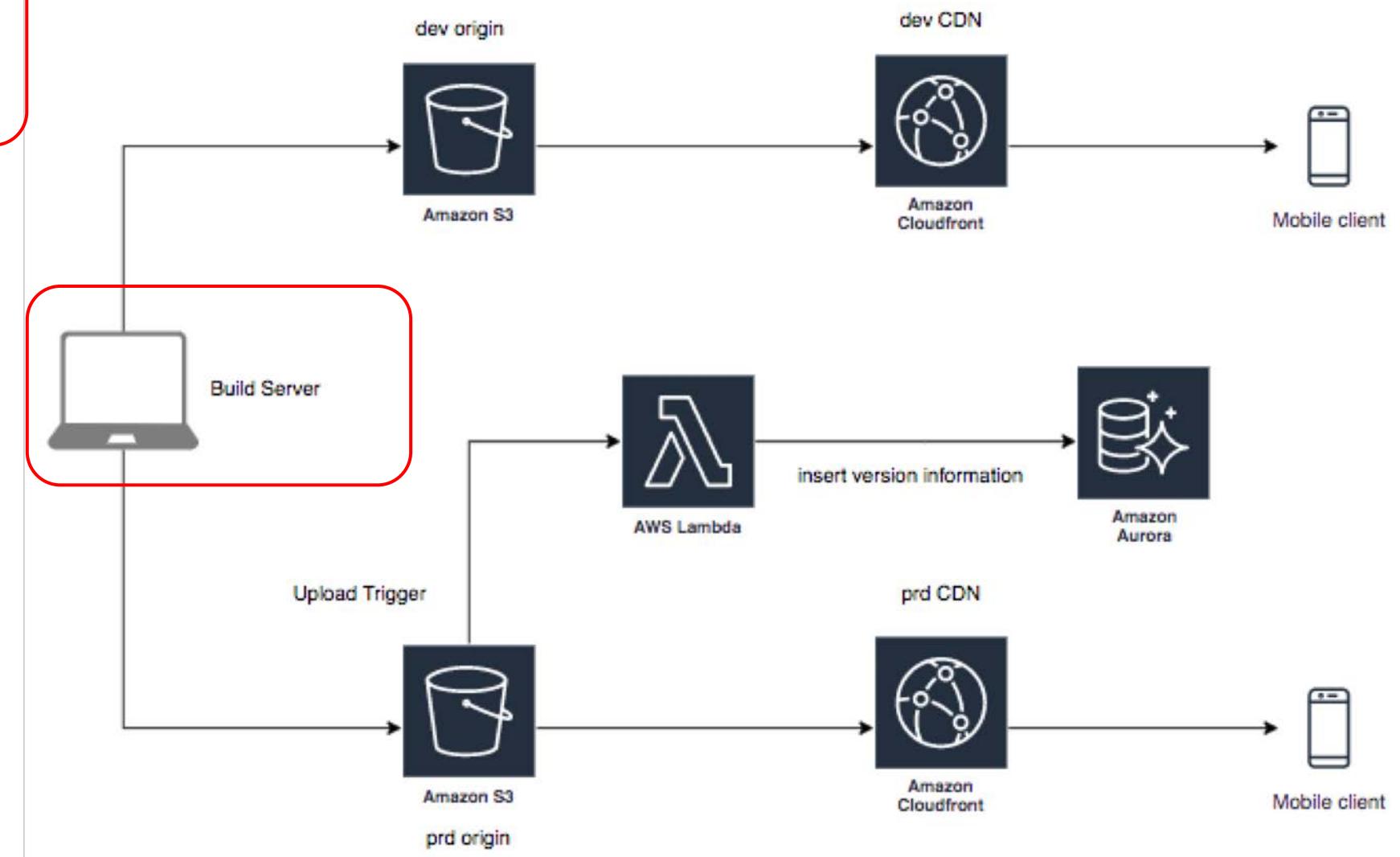
アプリケーションバージョンと
アセットバンドルバージョンの
対応情報を
アプリケーションソースで管理

<移設後>

アセットバンドルデプロイ時に
自動でインサートされる

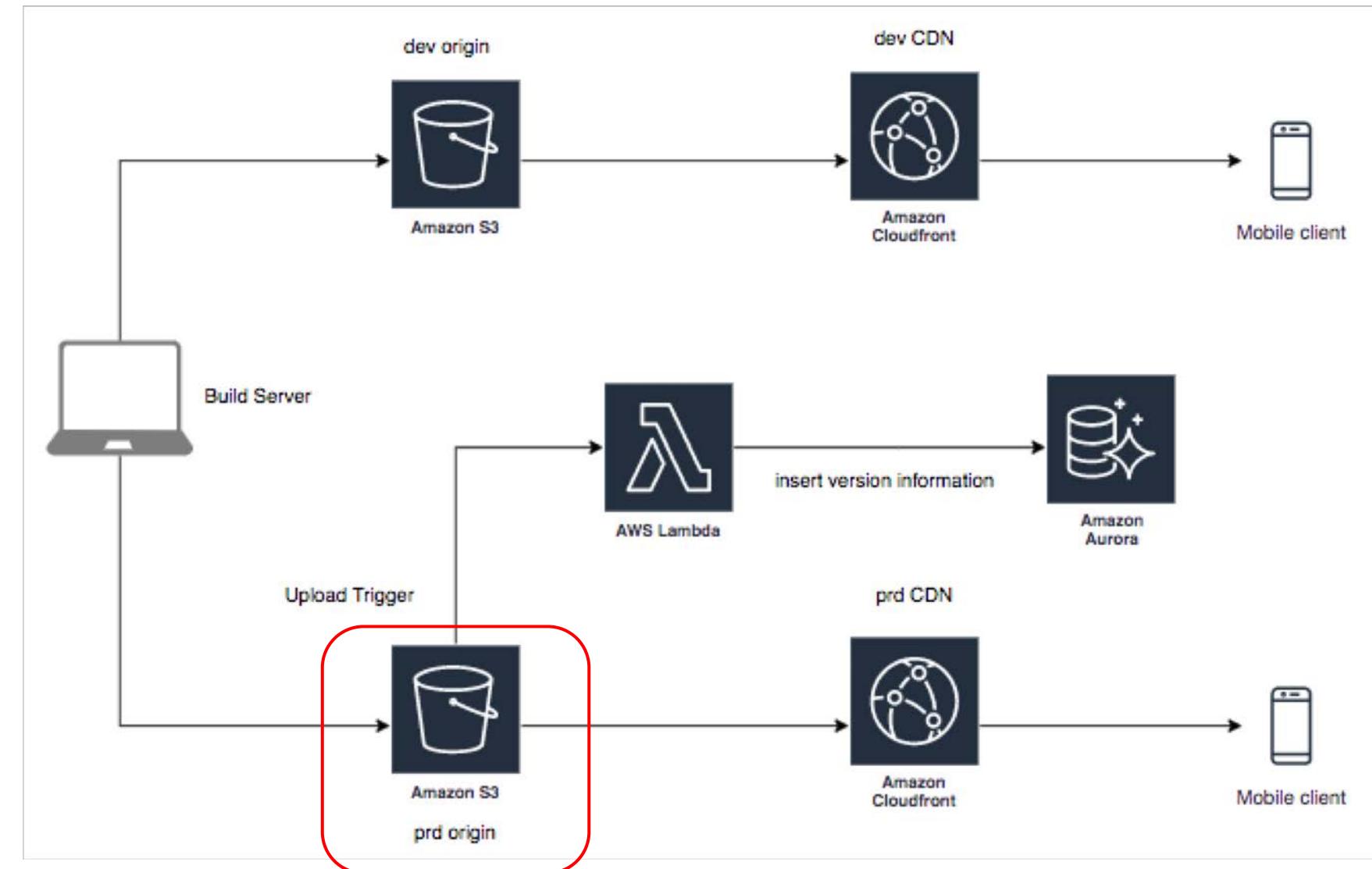
移設後のアセットバンドルデプロイ

- ① Build Server (jenkins) のビルドシーケンスの管理
- ② 本番用ビルドの最終ステップで S3 へアップロードする
- ③ S3 へのアップロードをトリガーにしてアプリケーションとアセットバンドルの対応バージョン情報を DB にインサート
- ④ アプリケーションが対応のバージョンを起動時に確認して、CDN 経由でダウンロードして使用



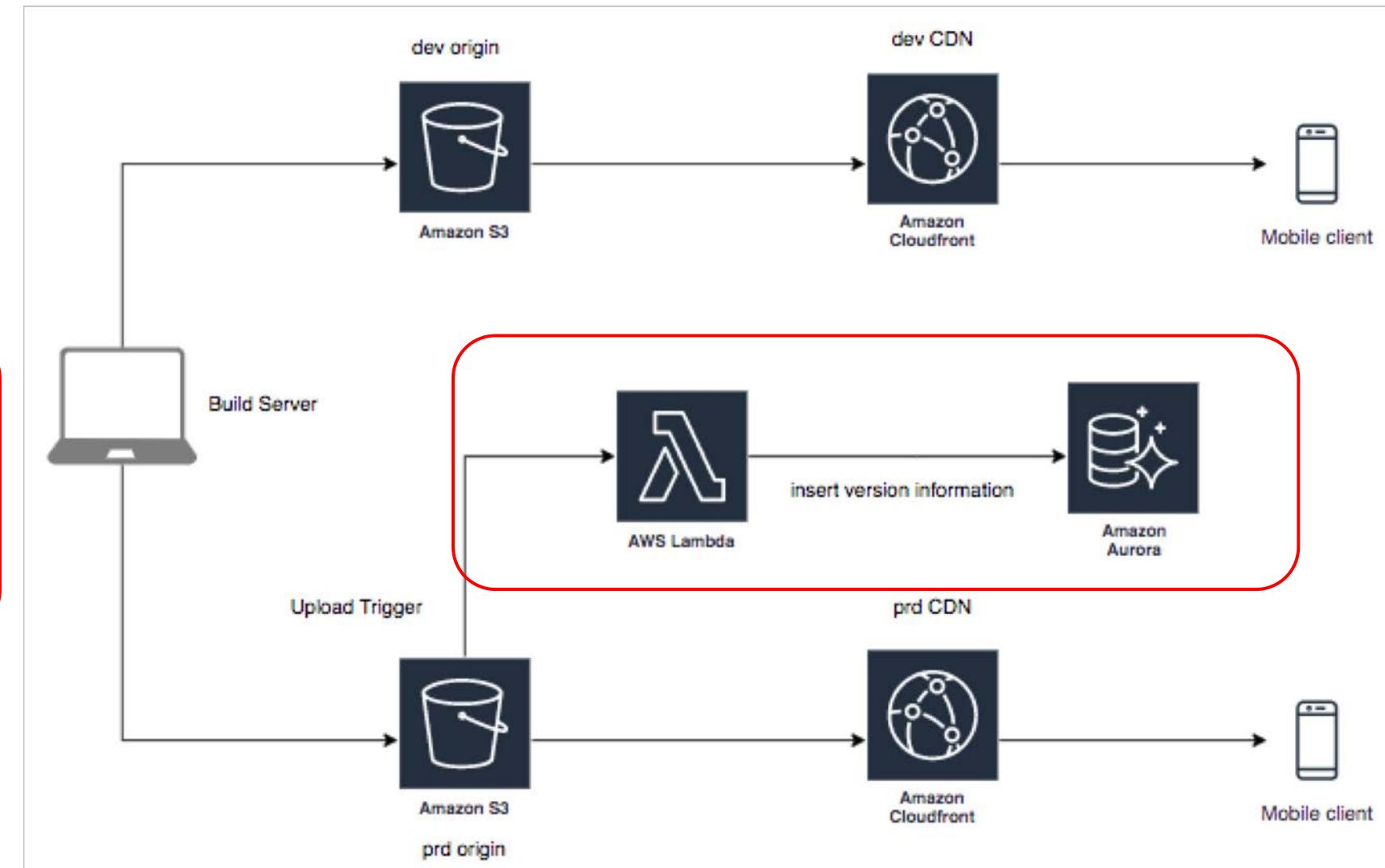
移設後のアセットバンドルデプロイ

- ① Build Server (jenkins) のビルドシーケンスの管理
- ② 本番用ビルドの最終ステップで S3 へアップロードする
- ③ S3 へのアップロードをトリガーにしてアプリケーションとアセットバンドルの対応バージョン情報を DB にインサート
- ④ アプリケーションが対応のバージョンを起動時に確認して、CDN 経由でダウンロードして使用



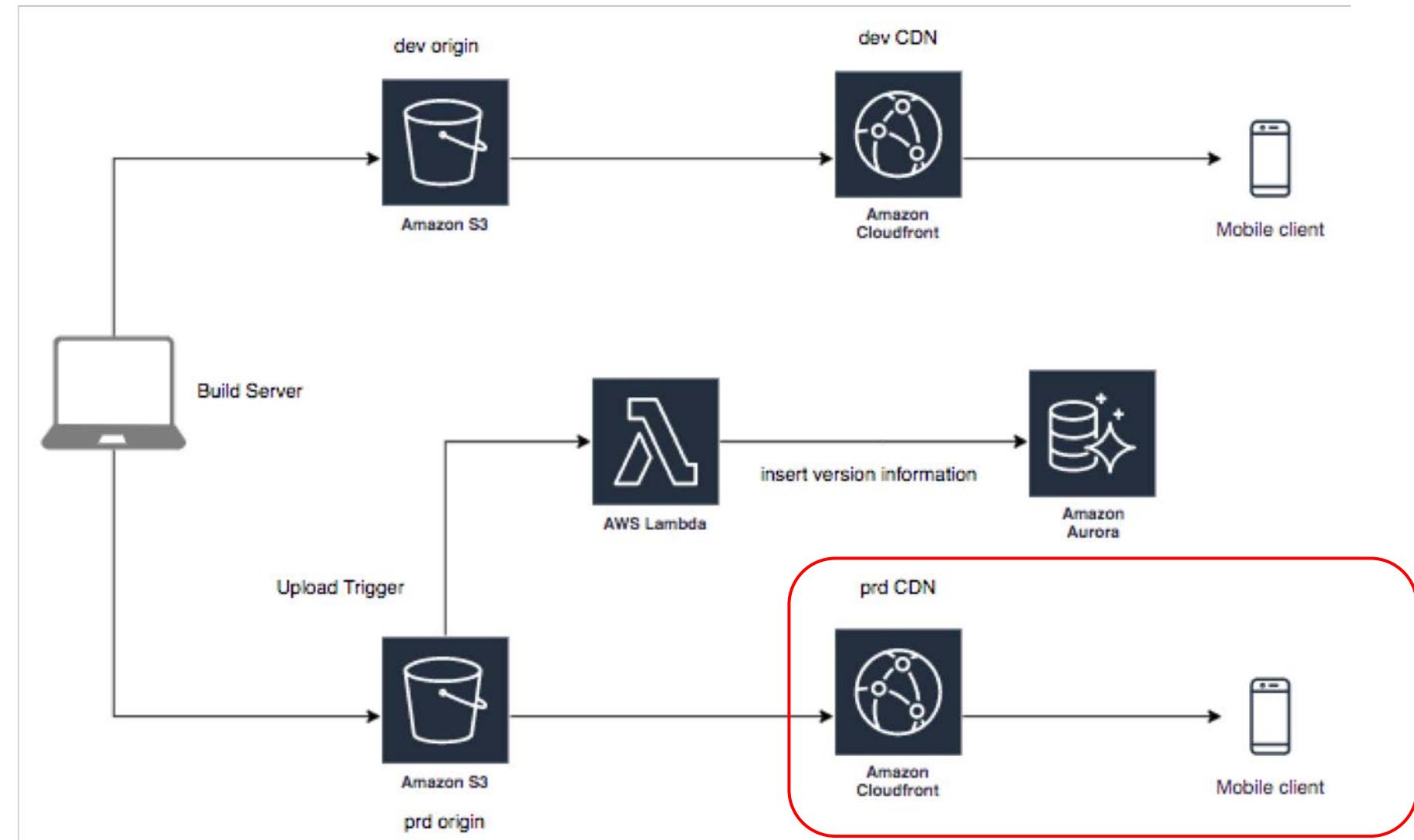
移設後のアセットバンドルデプロイ

- ① Build Server (jenkins) のビルドシーケンスの管理
- ② 本番用ビルドの最終ステップで S3 へアップロードする
- ③ S3 へのアップロードをトリガーにしてアプリケーションとアセットバンドルの対応バージョン情報を DB にインサート
- ④ アプリケーションが対応のバージョンを起動時に確認して、CDN 経由でダウンロードして使用



移設後のアセットバンドルデプロイ

- ① Build Server (jenkins) のビルドシーケンスの管理
- ② 本番用ビルドの最終ステップで S3 へアップロードする
- ③ S3 へのアップロードをトリガーにしてアプリケーションとアセットバンドルの対応バージョン情報を DB にインサート
- ④ アプリケーションが対応のバージョンを起動時に確認して、CDN 経由でダウンロードして使用



デプロイ自動化

1. アセットバンドル自動化

2. 画像デプロイ自動化

3. アプリケーションソースデプロイ効率化

画像デプロイ自動化

新しく配信する画像のパスを予想させないためハッシュ化が必須

<移設前>

動的に画像を探して配信

- ①ハッシュ名でリクエスト
- ②アンハッシュ +
コンテンツボディの取得
- ③ハッシュ名で
コンテンツボディは
オリジナルのまま送信

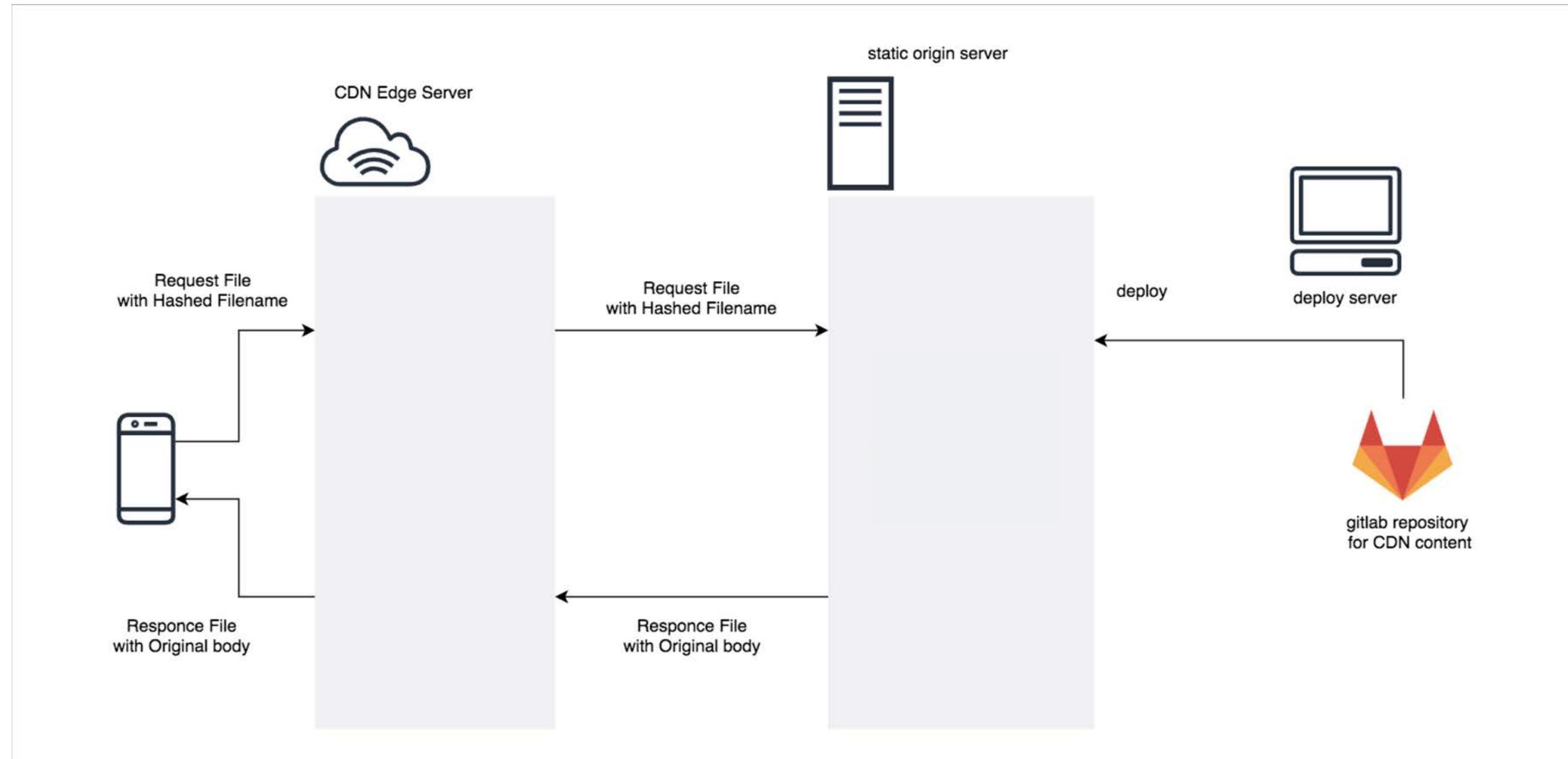
<移設後>

静的に変更

- ① git push -> gitlab CI
- ② gitlab CI が Lambda を実行
- ③ Lambda は ソースバケットから
データを読み取り、
ターゲットバケットに
名前をハッシュ化して保存

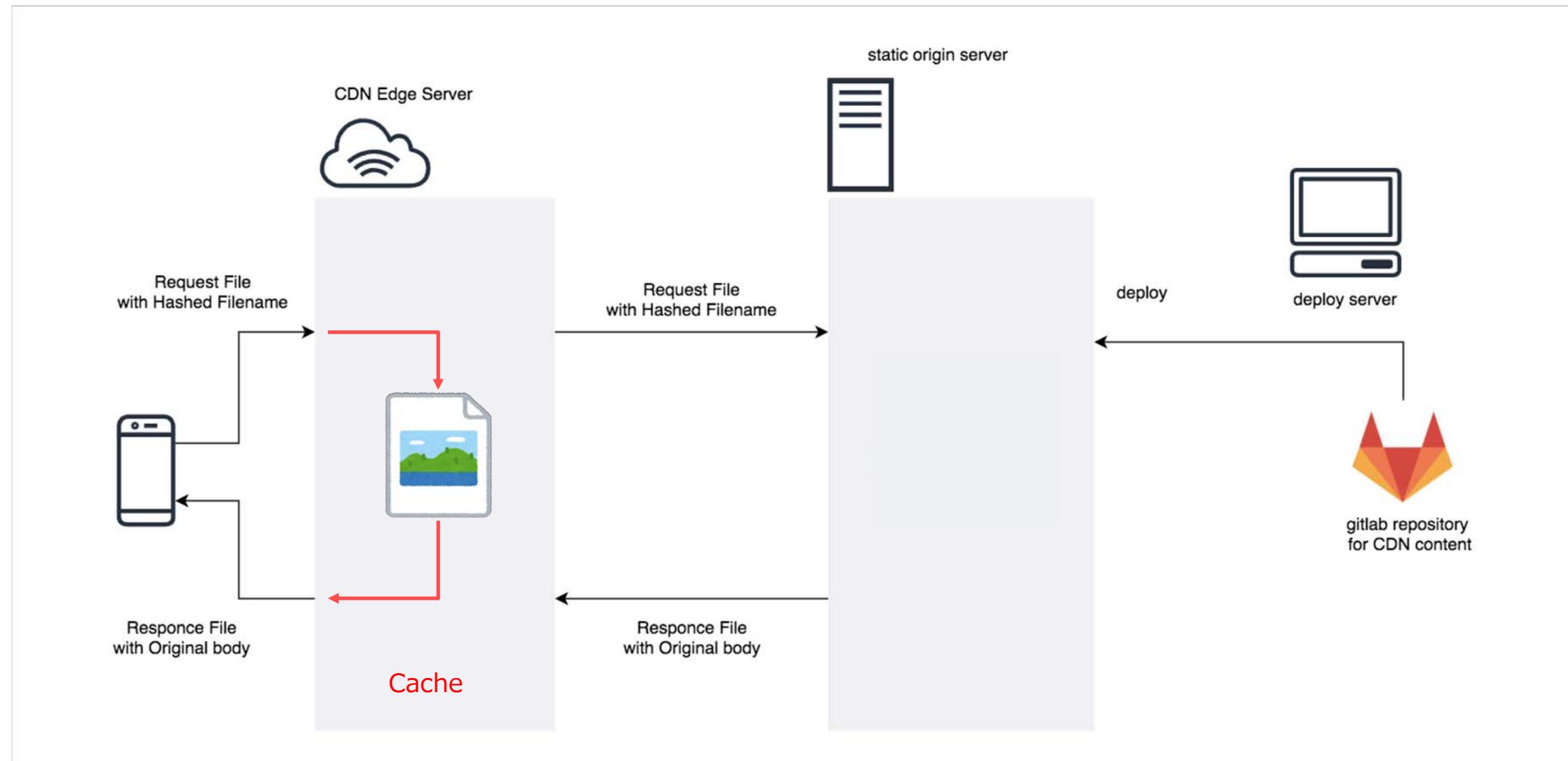
移設前の画像配信システム

オリジンサーバーへのアクセスがある度に、動的に変換して返す



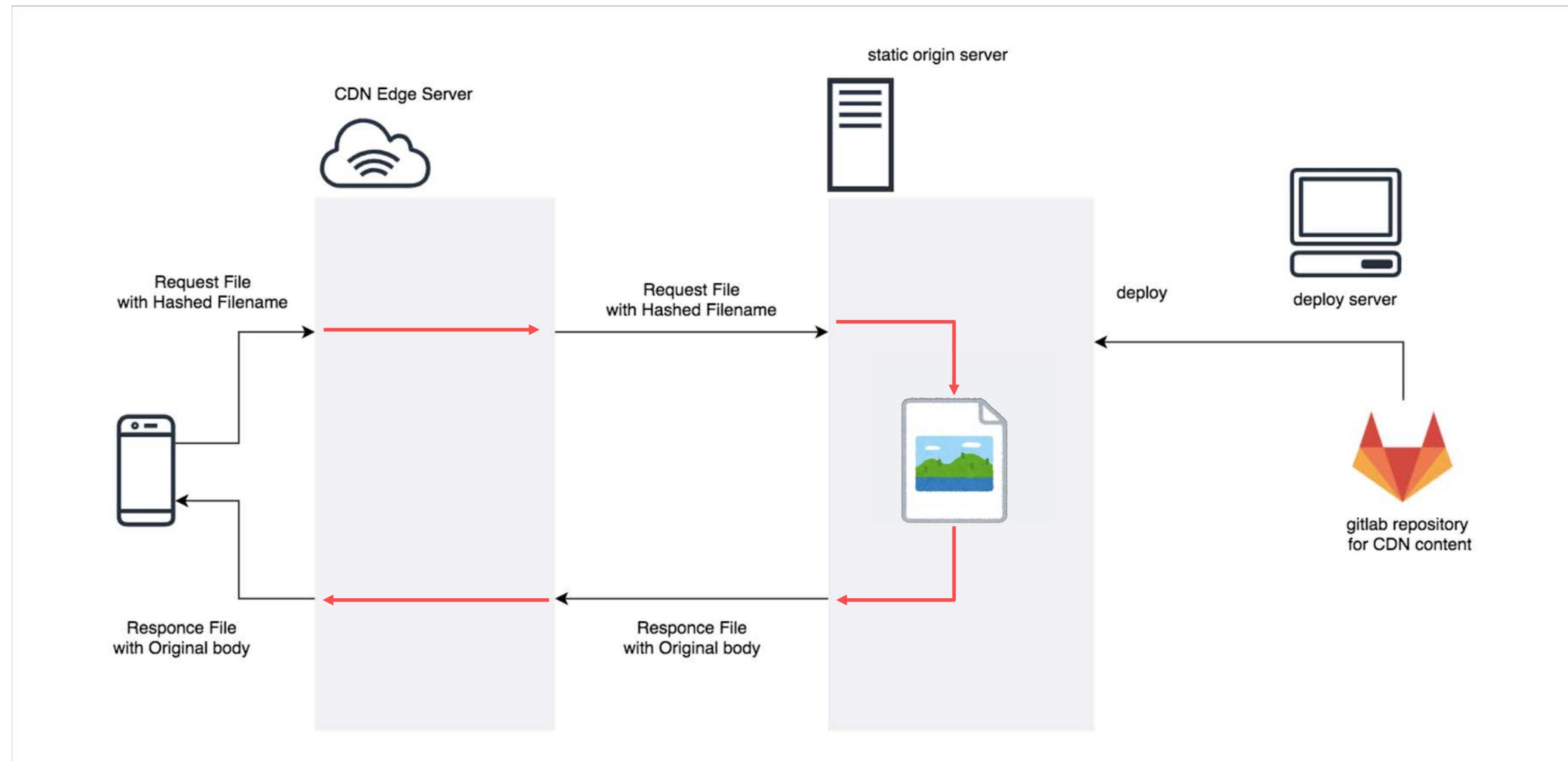
移設前の画像配信システム

オリジンサーバーへのアクセスがある度に、動的に変換して返す



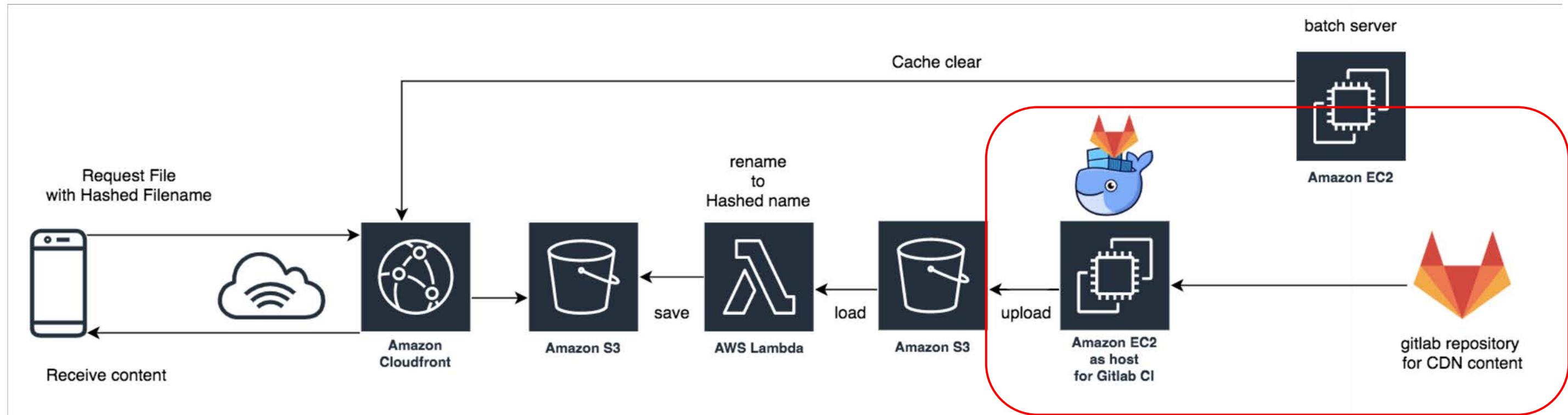
移設前の画像配信システム

オリジンサーバーへのアクセスがある度に、動的に変換して返す



移設後の画像配信システム

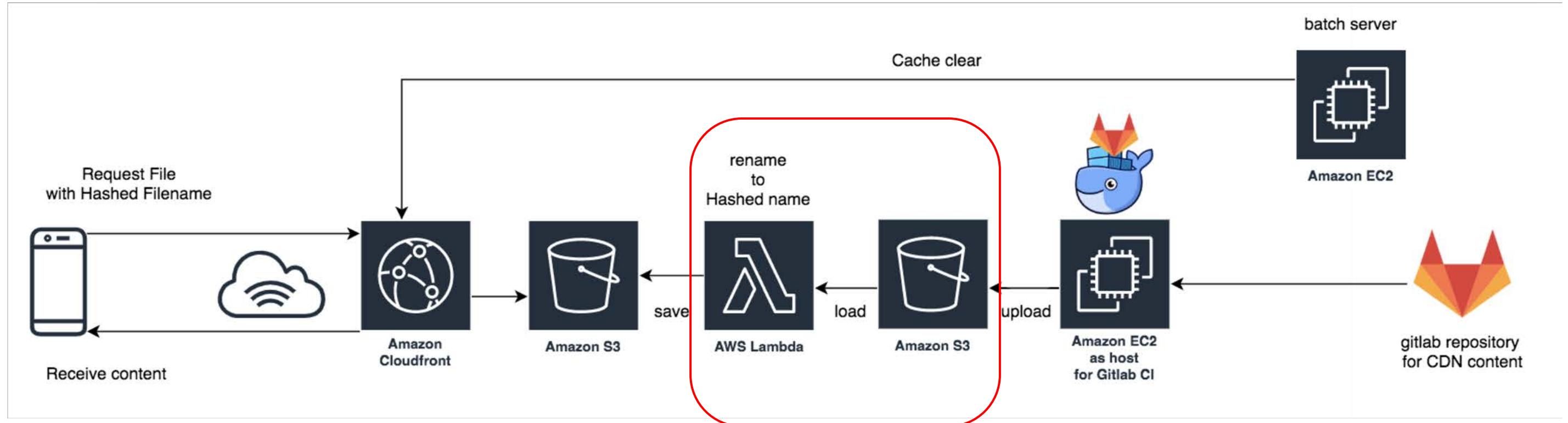
あらかじめハッシュ化された名前のファイルを用意しておいて返す



push をトリガーに CI runner が S3 にファイルをアップロードする

移設後の画像配信システム

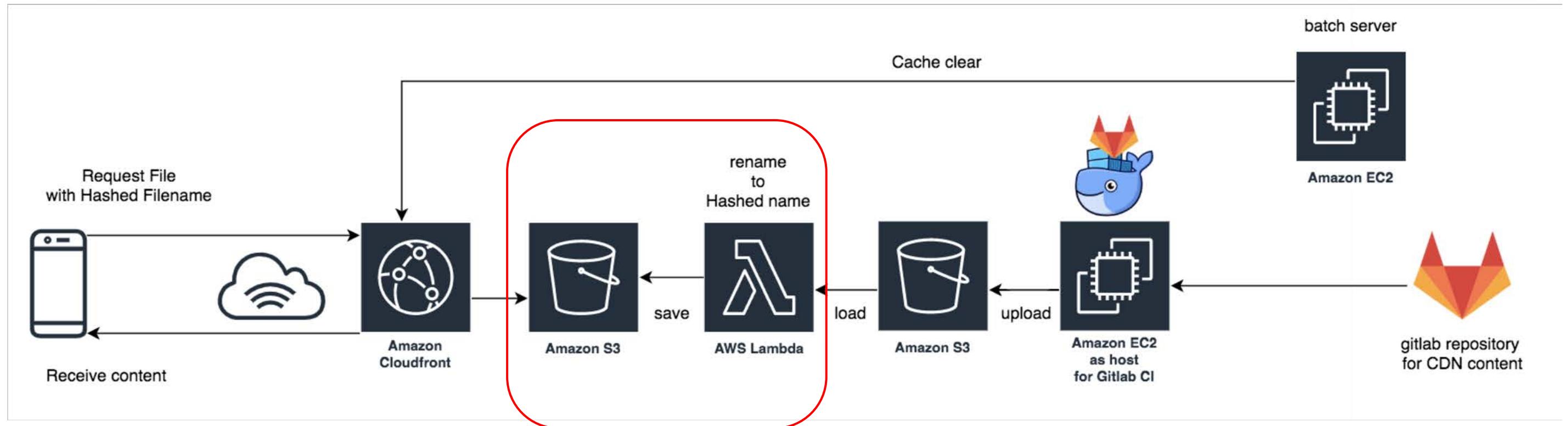
あらかじめハッシュ化された名前のファイルを用意しておいて返す



CI サーバーが Lambda を呼び出す
Lambda は画像ファイルを S3 から取得する

移設後の画像配信システム

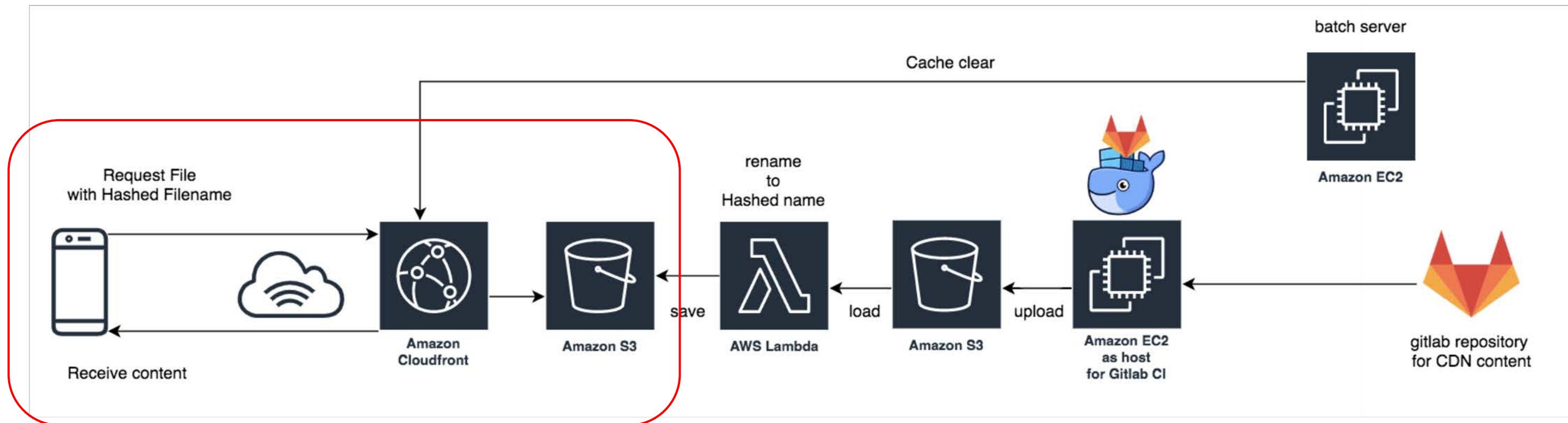
あらかじめハッシュ化された名前のファイルを用意しておいて返す



Lambda は画像ファイルを rename して
CDN の origin となっているバケットへ保存する

移設後の画像配信システム

あらかじめハッシュ化された名前のファイルを用意しておいて返す



CDN経由でアプリケーションから利用することができる

デプロイ自動化

1. アセットバンドル自動化
2. 画像デプロイ自動化
3. アプリケーションソースデプロイ効率化

アプリケーションソースデプロイ

デプロイ方法の変更で、**デプロイ実行時間が最大で1/9になった！**

<移設前>

デプロイ方法

- scp での直列 rsync で配布

デプロイ対象管理

- 手動でリストに追加

デプロイ時間

- 台数が多くなるほど
実行時間が増える

<移設後>

デプロイ方法

- 並列 pull型

デプロイ対象管理

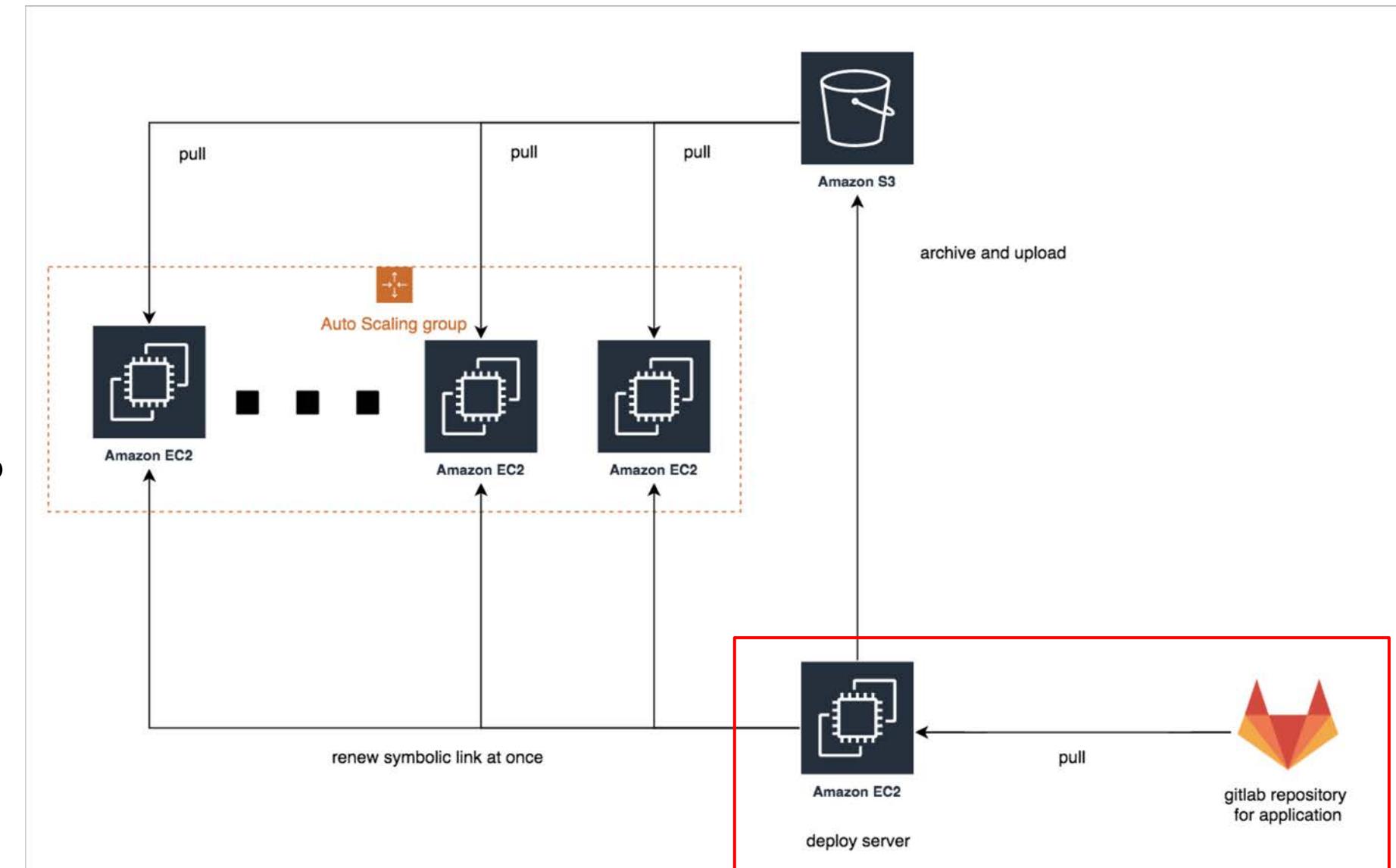
- タグによる自動判別

デプロイ時間

- **台数が増えても
実行時間は変わらない**

移設後のアプリケーションソースデプロイ

- ① git から最新版を pull
- ② 圧縮して S3 へアップロードする
- ③ S3 から一斉にダウンロードして解凍する
- ④ 全てのサーバーで解凍完了したらシンボリックリンクを変更して全サーバーで同時にリリース



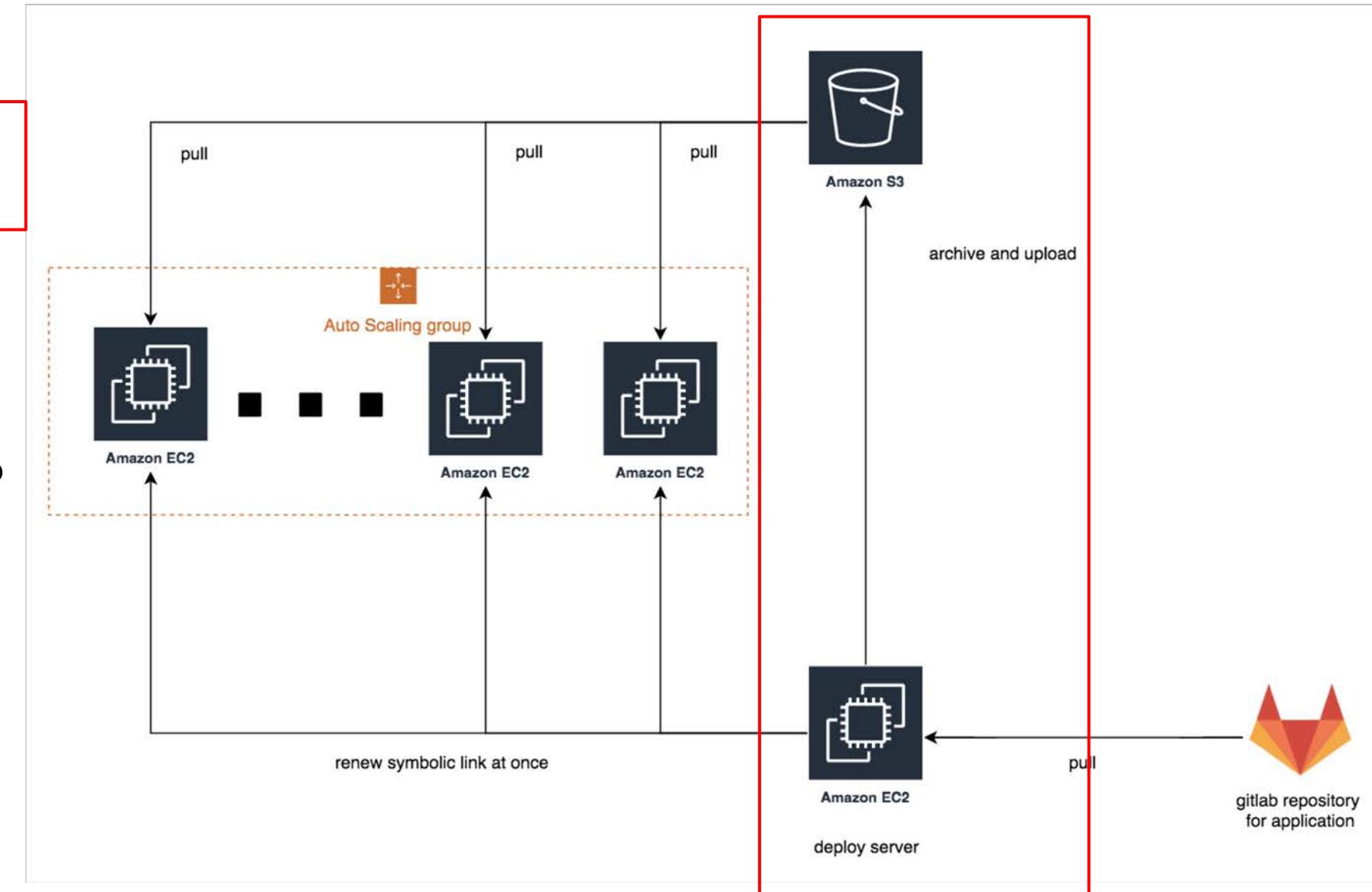
移設後のアプリケーションソースデプロイ

① git から最新版を pull

② 圧縮して S3 へアップロードする

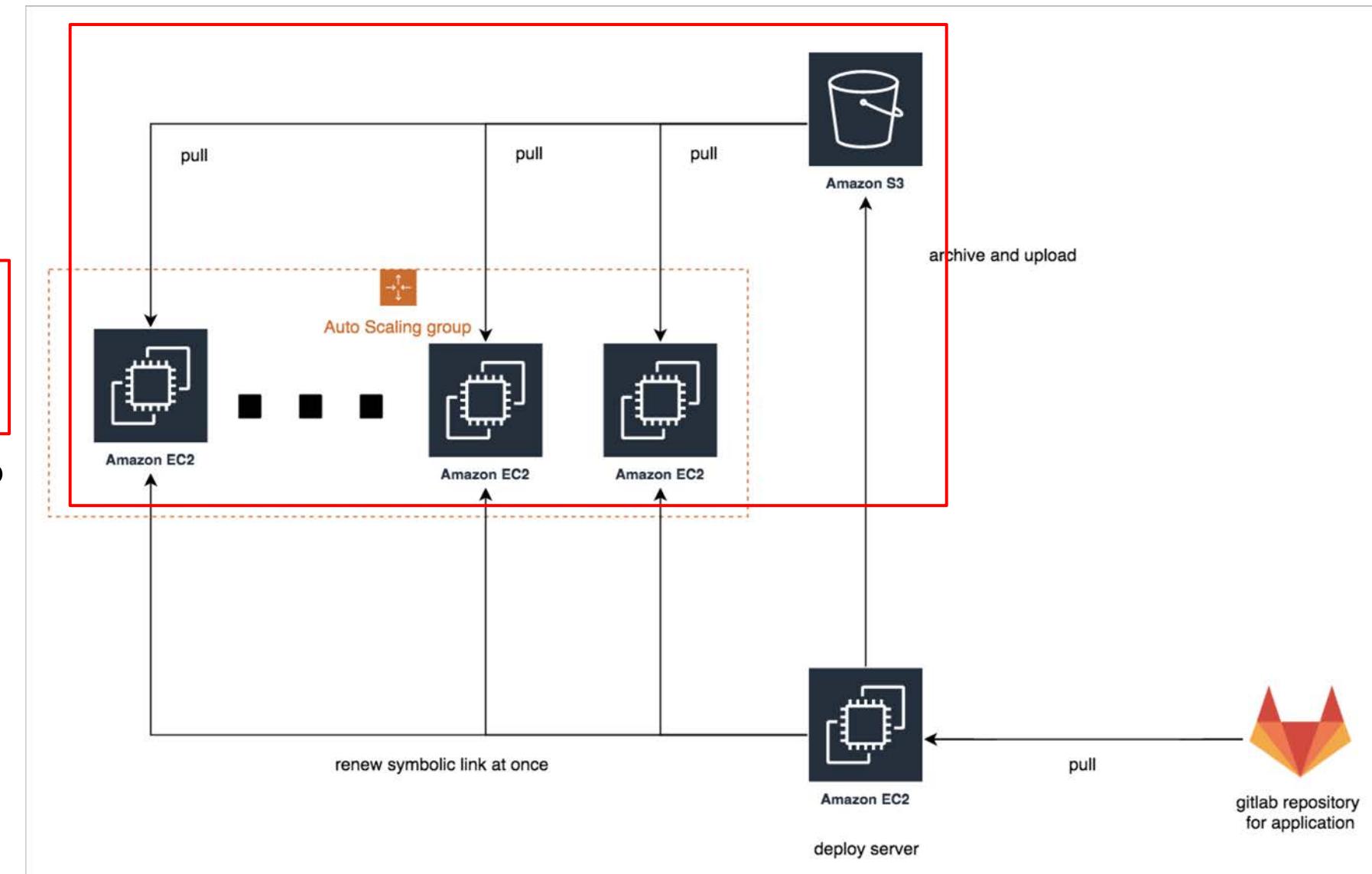
③ S3 から一斉にダウンロードして
解凍する

④ 全てのサーバーで解凍完了したら
シンボリックリンクを変更して
全サーバーで同時にリリース



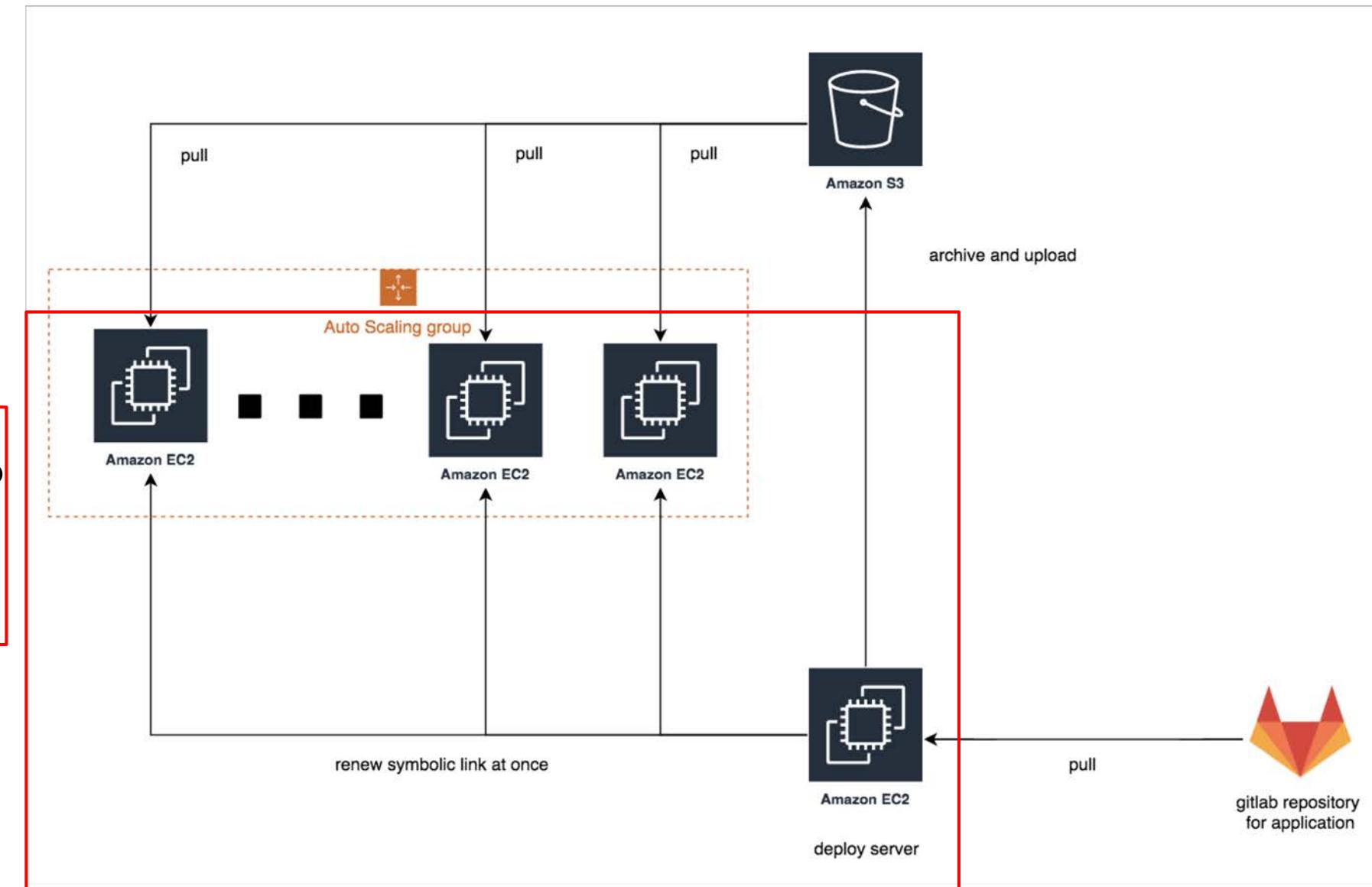
移設後のアプリケーションソースデプロイ

- ① git から最新版を pull
- ② 圧縮して S3 へアップロードする
- ③ S3 から一斉にダウンロードして解凍する
- ④ 全てのサーバーで解凍完了したらシンボリックリンクを変更して全サーバーで同時にリリース



移設後のアプリケーションソースデプロイ

- ① git から最新版を pull
- ② 圧縮して S3 へアップロードする
- ③ S3 から一斉にダウンロードして解凍する
- ④ 全てのサーバーで解凍完了したらシンボリックリンクを変更して全サーバーで同時にリリース



どう解決したか

- デプロイ自動化
- 接続先管理自動化
- 分析基盤 効率化

接続先管理自動化

ソースコードに依存していた Memcache 接続情報とマッピングを自動認識するように変更

<移設前>

接続情報

- ・ソースコードに記述

マッピング

- ・ソースコードに記述

障害発生時

- ・デプロイして変更する

<移設後>

接続情報

- ・AWS API で取得
Redis に書き込み
web サーバにキャッシュ

マッピング

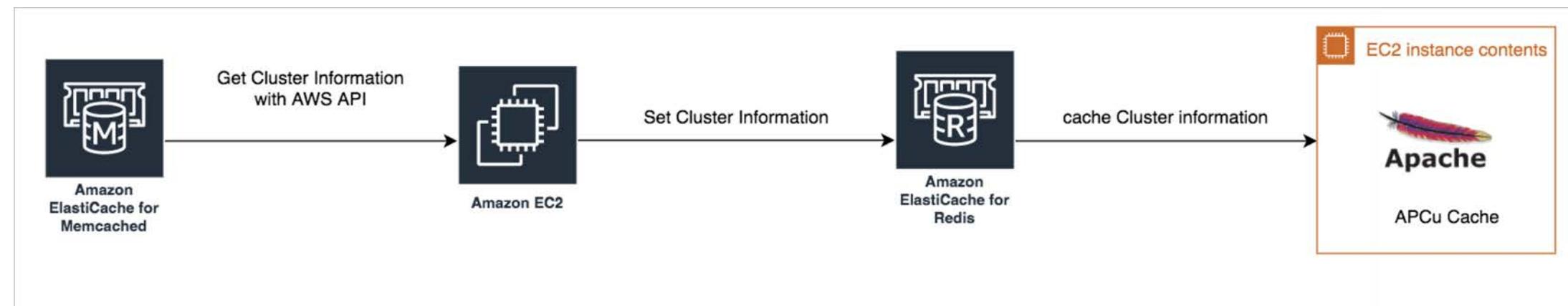
- ・consistent hashing

障害発生時

- ・consistent hashing なので
作業は不要

接続先管理自動化

application config に書かれていた接続情報をRedis に記録
mapping を consistent hashing に変更



- AWS API を用いた ElastiCache クラスタ情報取得が RateLimit を超えないように Redis にキャッシュする
- Redis へアクセスするコスト低減のために web サーバの local cache (APCu) にもキャッシュする

どう解決したか

- デプロイ自動化
- 接続先管理自動化
- 分析基盤 効率化

分析基盤

複雑な仕組みで億単位のレコードのテーブルをコピーしていた

<移設前>

データコピー

- ・数時間かけて、複雑な処理を行う
- ・数十億レコードを毎晩コピー

システム

- ・専用のシステムで管理／運用コストが高い

分析処理

- ・深夜から数時間かけて実行
- ・複雑な処理がコピー時間に依存

<移設後>

データコピー

- ・1日分のデータのみに絞ったテーブルもある
- ・必要なテーブルのみに絞る

システム

- ・サーバーレスで構成、管理／運用コスト低減

分析処理

- ・15分程度で終了
- ・複雑な処理と依存がなくなった

分析基盤

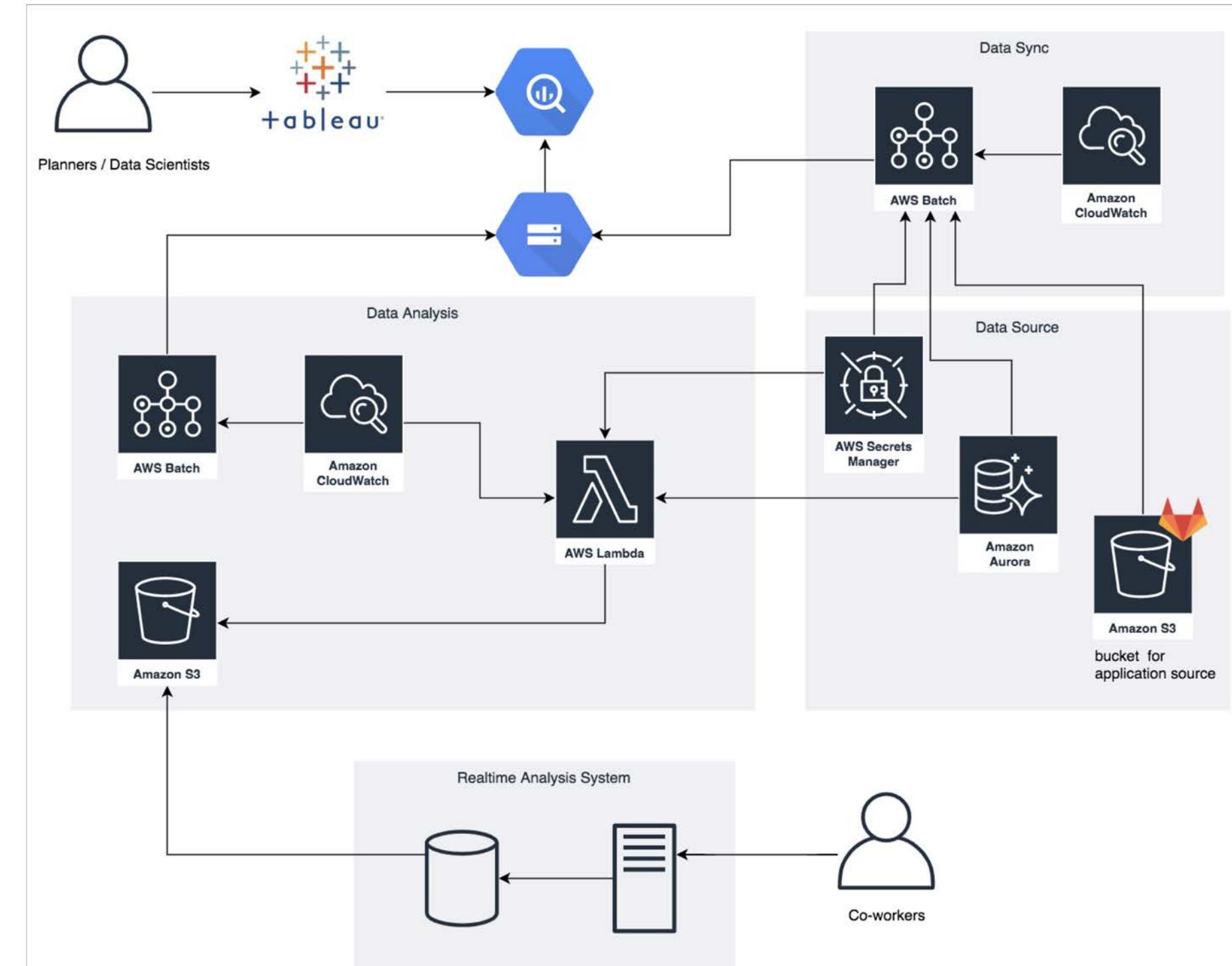
夜間バッチ

- ・データのコピー

リアルタイム分析

- ・Lambda で自動化

過去の課題を解消しつつ、
サーバーレスで構成



その他移設で工夫したところ

A.単一ゾーン構成

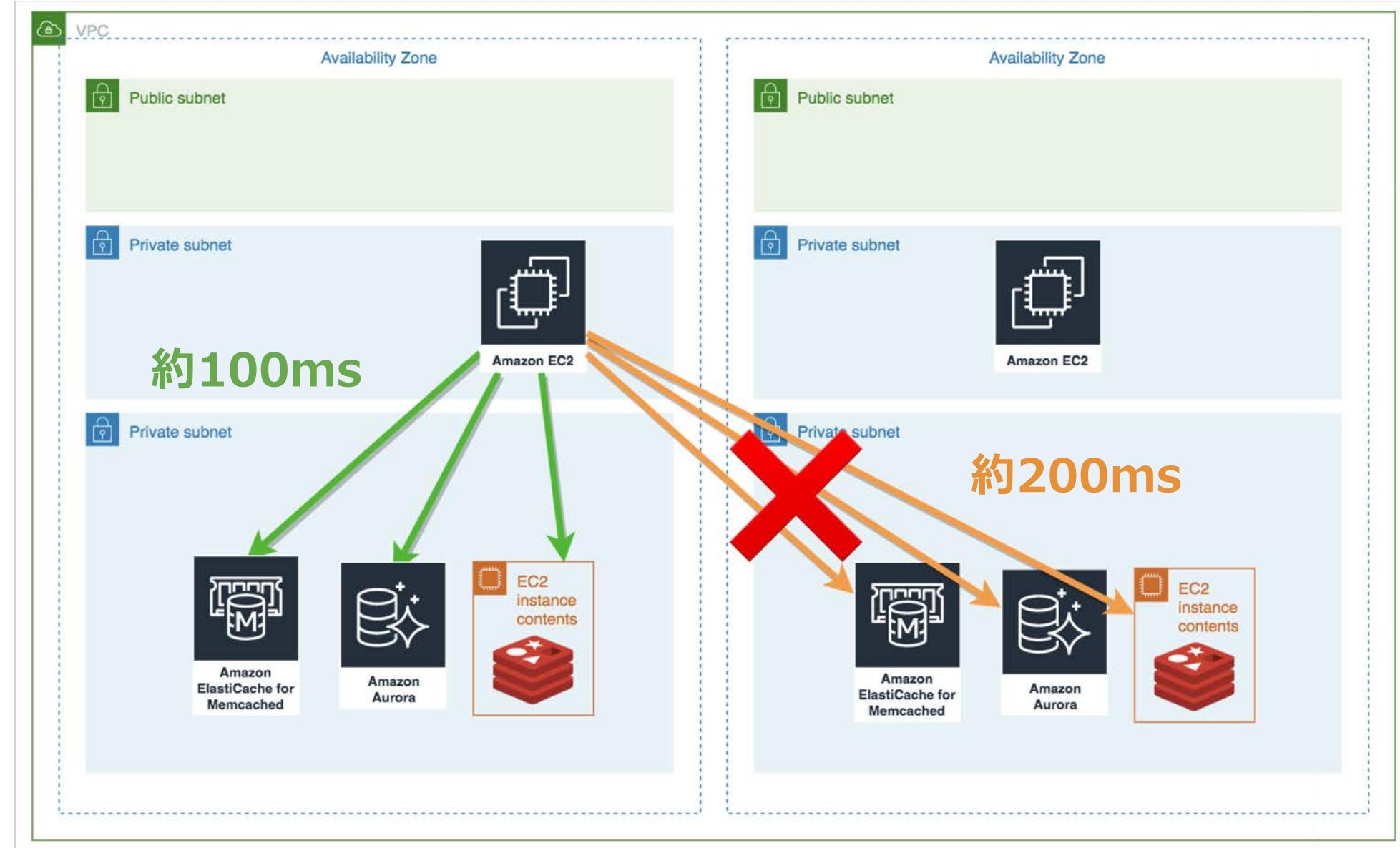
B.IEM

単一ゾーン構成

全リソースを
单一ゾーン構成にしている

ゲーム事業では
「数百ミリ秒で遅い！」
と言われる世界

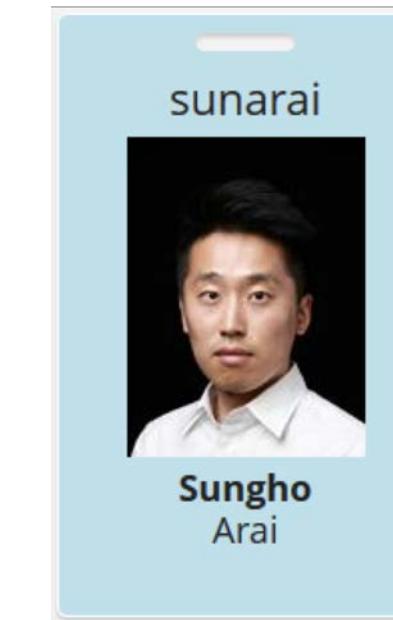
ユーザの体感を考えて
元々のデータセンターと
同じシングルAZで構成する



AWS Infrastructure Event Management (IEM)

ビジネスを左右する重要な時期にガイダンスやリアルタイムのサポートを受けることができます！

毎週の定例を行い、技術相談にのっていただきました。



ありがとうございました！

アジェンダ

自己紹介/会社概要

AWS への移設の背景

旧環境での課題と AWS での解決方法

今後の展望

ドメイン管理

ドメイン管理

- Route 53 に移行したい
- 開発環境、ステージング環境、
負荷試験環境は移行済み

メリット：

- DNS failover
- CloudFront, ACM との組み合わせで
証明書の取得・自動更新が可能！



Redis 構成変更

現状

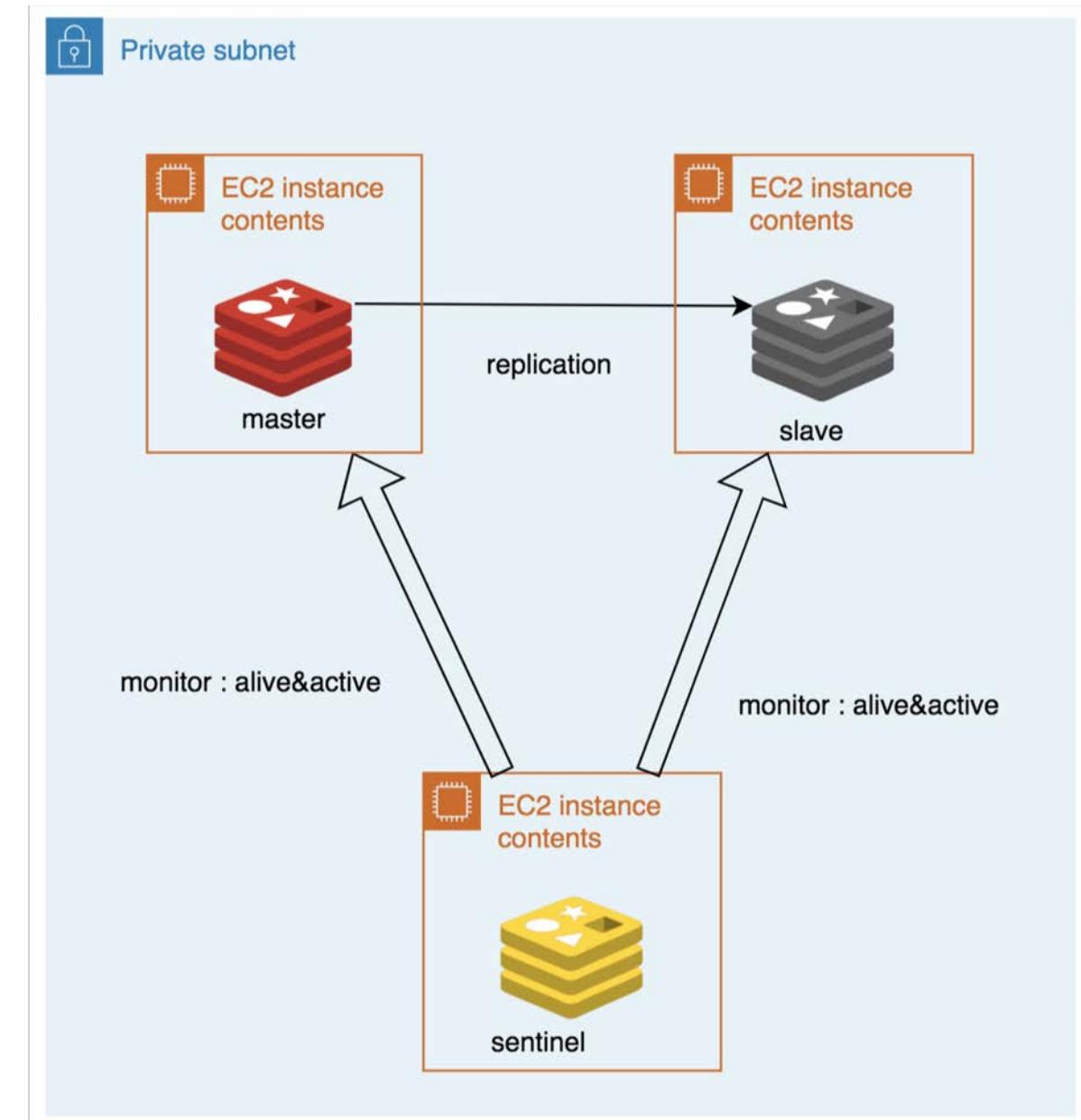
Redis 約500ポート 用途としては5種類程度

構成

- Active 1台 (SPOF)
- Active-Standby + backup の3台1組のクラスタ

やりたいこと

- 管理コストの低減と SPOF の排除
(台数削減、ポート数削減)
- failover 及び 復旧の高速化



エラーバジェットによるリスク管理

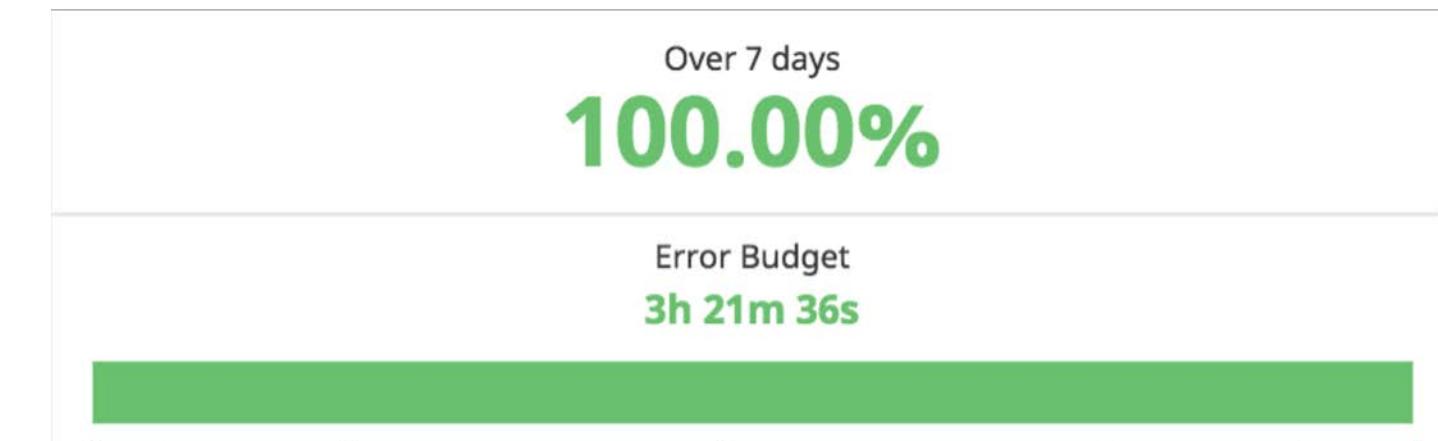
トイレにかかっていた時間を安定稼働するための改善に使う

性能や安定度などを必要な情報から数値化して目標管理したい

SLI = 指標

SLO = SLI + 目標

SLA = SLO + 罰金(罰則)



まとめ

オンプレミスから AWS へ移設しました。

移設とその後の改善の効果

- 平均レスポンス時間を **1/3** に低減
- コストを **1/3** に削減
- 運用効率の改善

今後の展望

- Route 53、Redis などの課題解決
- エラーバジェットによるリスク管理

Spot Instance の本番導入などを進めつつ、開発チームと連携して安定稼働に尽力します！

SRE メンバー募集集中

我々と共に安定運用に取り組む
メンバーを募集しています。
気になる方はお声かけください！



<https://www.sumzap.co.jp/recruit/>

aws SUMMIT



SREngine

