ロマサガRSの大規模負荷を処理する Amazon ECS & Docker運用知見

用語統一 (※ ガイドラインに則るため)

AWS：Amazon Web Services
EC2：Amazon Elastic Compute Cloud
ECS：Amazon Elastic Container Service
Fargate：AWS Fargate
EKS：Amazon EKS
ECR：Amazon Elastic Container Registry
AutoScaling：AWS Auto Scaling
RDS：Amazon Relational Database Service
Aurora：Amazon Aurora
DynamoDB：Amazon DynamoDB
EFS：Amazon Elastic File System
ALB：Application Load Balancer
CloudFormation：AWS CloudFormation
S3：Amazon Simple Storage Service
Lambda：AWS Lambda

CloudFront：Amazon CloudFront
CodeBuild：AWS CodeBuild
CodeDeploy：AWS CodeDeploy
CodePipeline：AWS CodePipeline
CloudWatch：Amazon CloudWatch
CloudWatchLogs：Amazon CloudWatch Logs
CloudWatchEvents：Amazon CloudWatch Events
CloudWatchAlarm：Amazon CloudWatch Alarm
SSM：AWS Systems Manager
SNS：Amazon Simple Notification Service
VPC：Amazon Virtual Private Cloud
WAF：AWS WAF
Athena：Amazon Athena
Route 53：Amazon Route 53
IAM：AWS Identity and Access Management

自己紹介

氏名

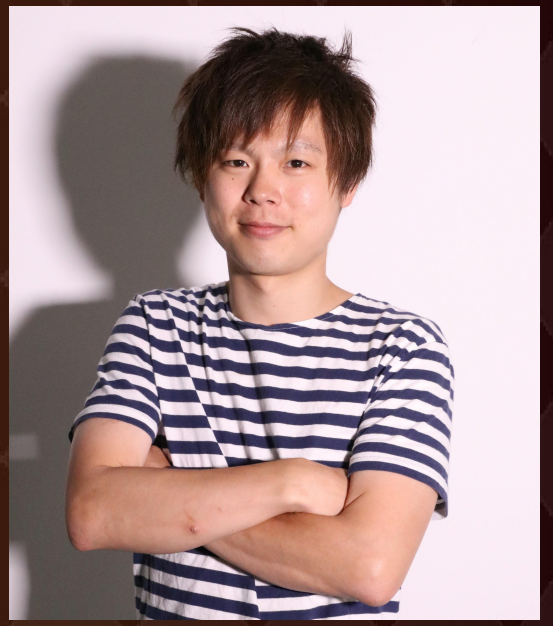
- 駒井祐人

経歴

- SIer
  - ミッションクリティカルシステム設計/構築
- アカツキ
  - リリースに関わったゲームは5～6タイトル
  - Store上位タイトルの開発/運用

個人

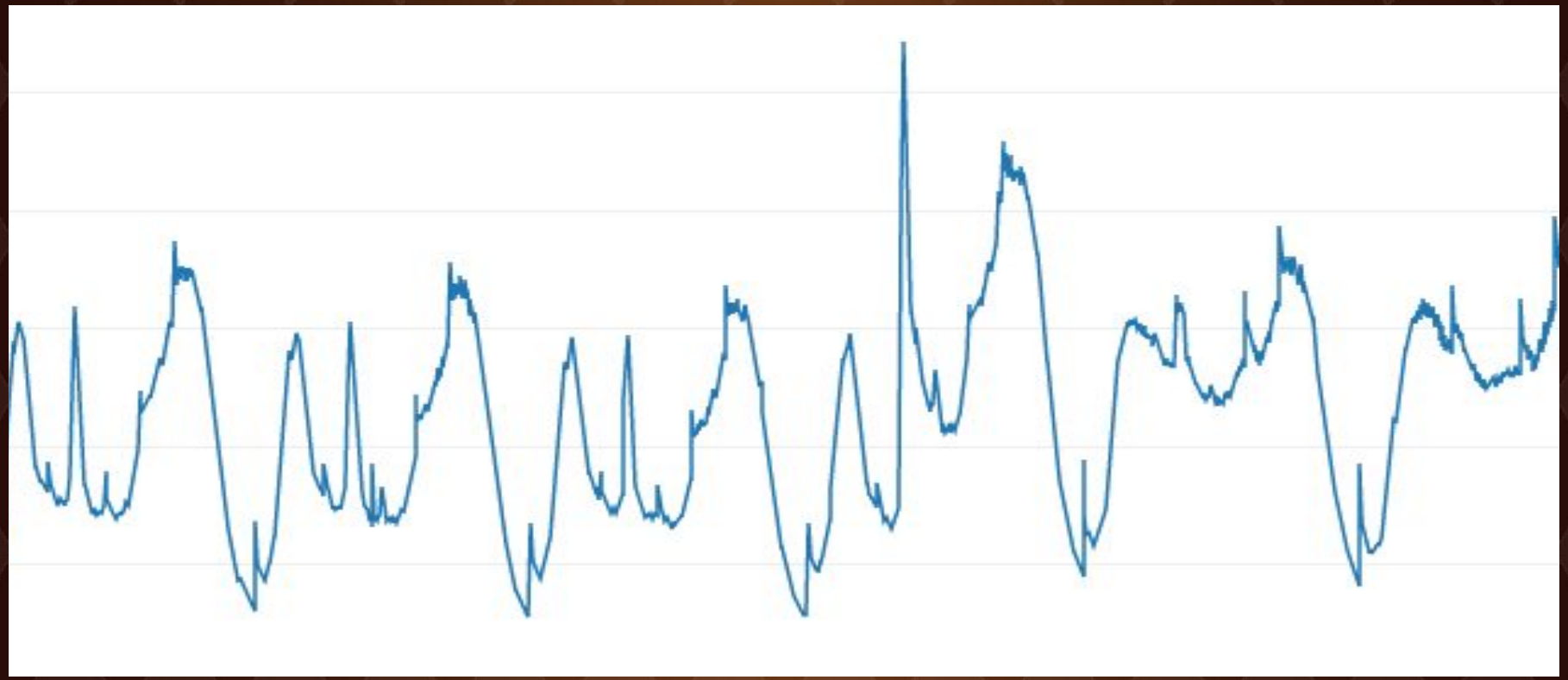
  - VTuber管理人
  - ハッカソン運営



@e__koma

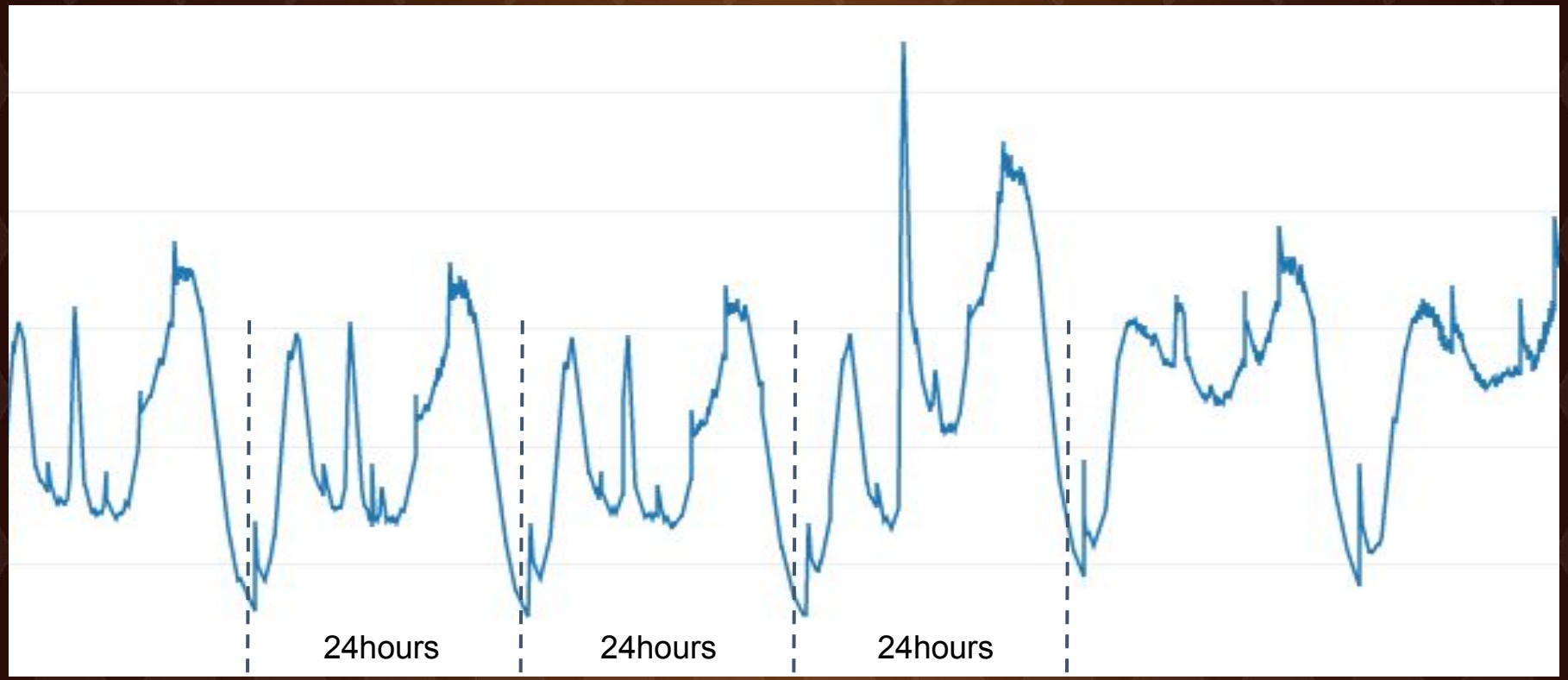
ゲームシステムの特性（例）

トラフィック量の増減



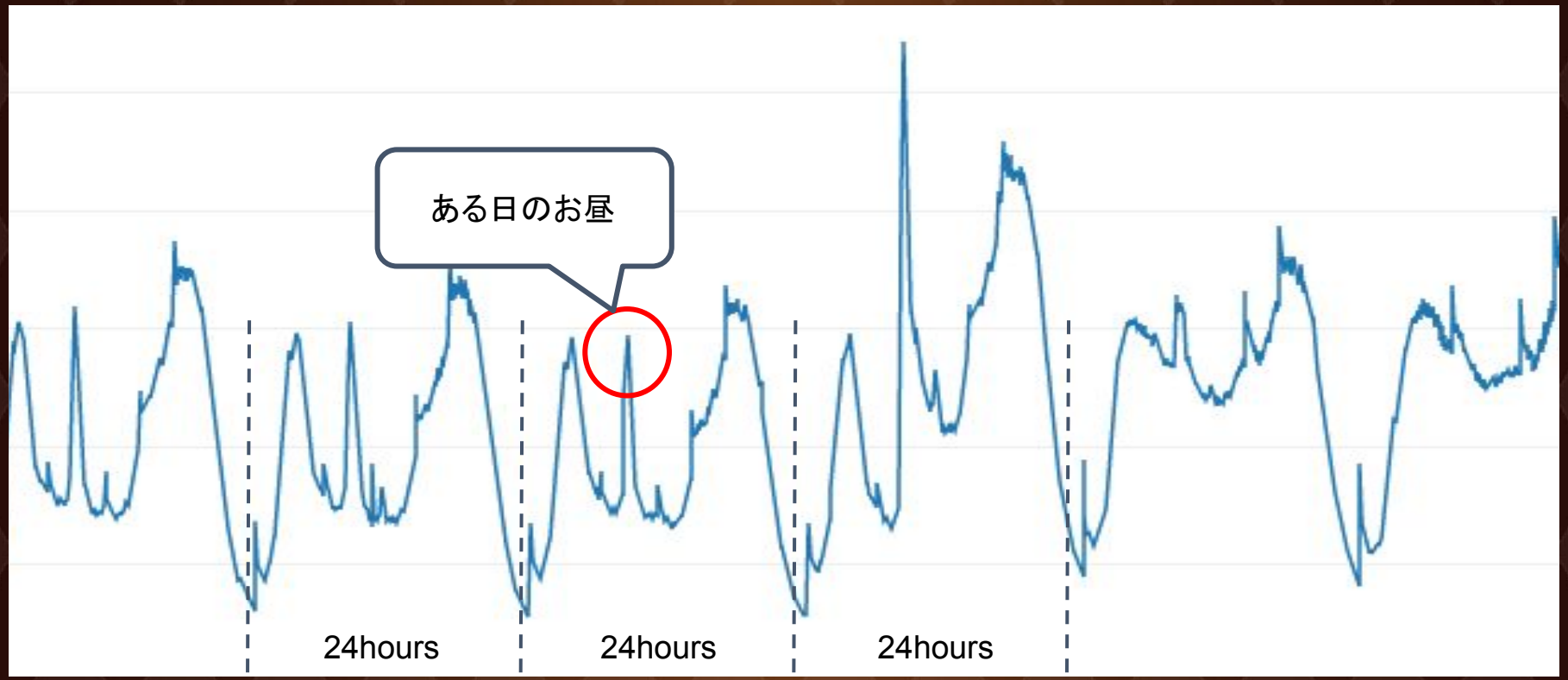
ゲームシステムの特性（例）

トラフィック量の増減



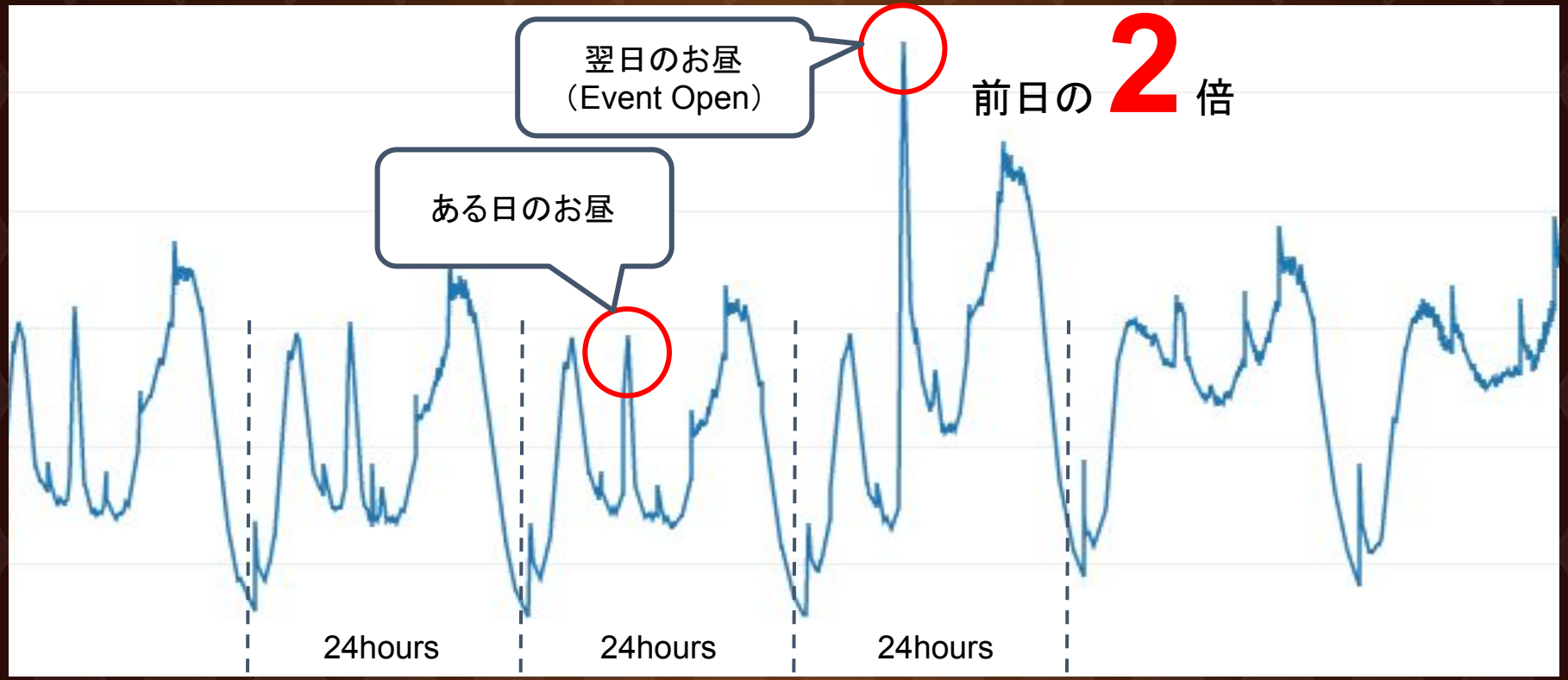
ゲームシステムの特性（例）

トラフィック量の増減



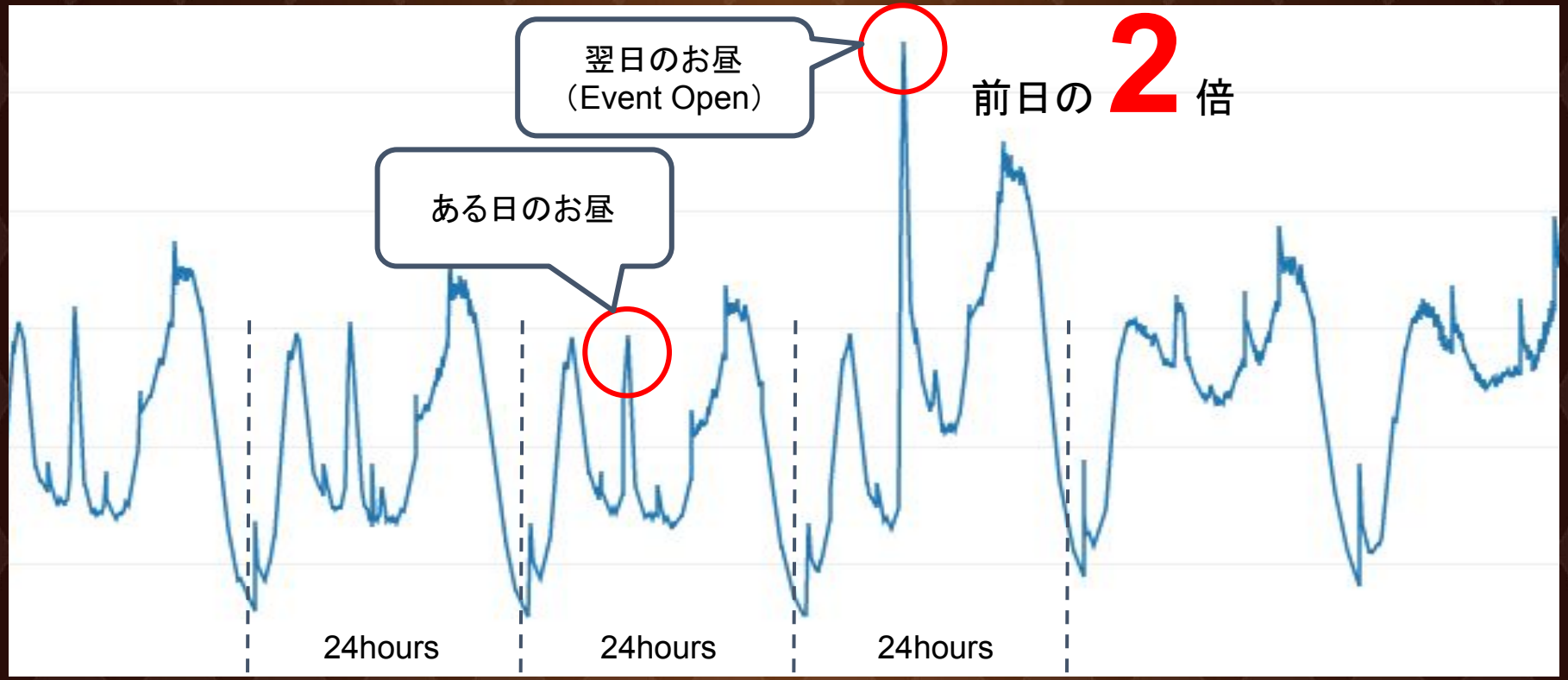
ゲームシステムの特性（例）

トラフィック量の増減



ゲームシステムの特性（例）

トラフィック量の増減



最大ピークが読みづらく、スケールするシステムが求められる

今日話すこと

1. アーキテクチャ編
   - アーキテクチャ全体像、構成要素の紹介
2. 負荷対策編
   - 負荷テスト事例
   - ECSスケーリングの工夫
3. 運用の工夫編
   - デリバリーパイプライン
   - サーバレス運用の工夫
   - 自動復旧

ロマンシング サガ リ・ユニバースとは

2018/12に(株)スクウェア・エニックスからリリースされた
23年ぶりの「ロマンシング サガ」完全新作
『ロマンシング サガ3』から300年後の世界が舞台

シリーズの枠を越えて展開されるオリジナルストーリー＆歴代のキャラクターたち！
ロマサガのバトルの面白さはそのままに、スマートフォンならではの手軽さで楽しめる！

ロマンシング サガ リ・ユニバースとは

リリース1ヶ月以内に

10,000,000

downloads

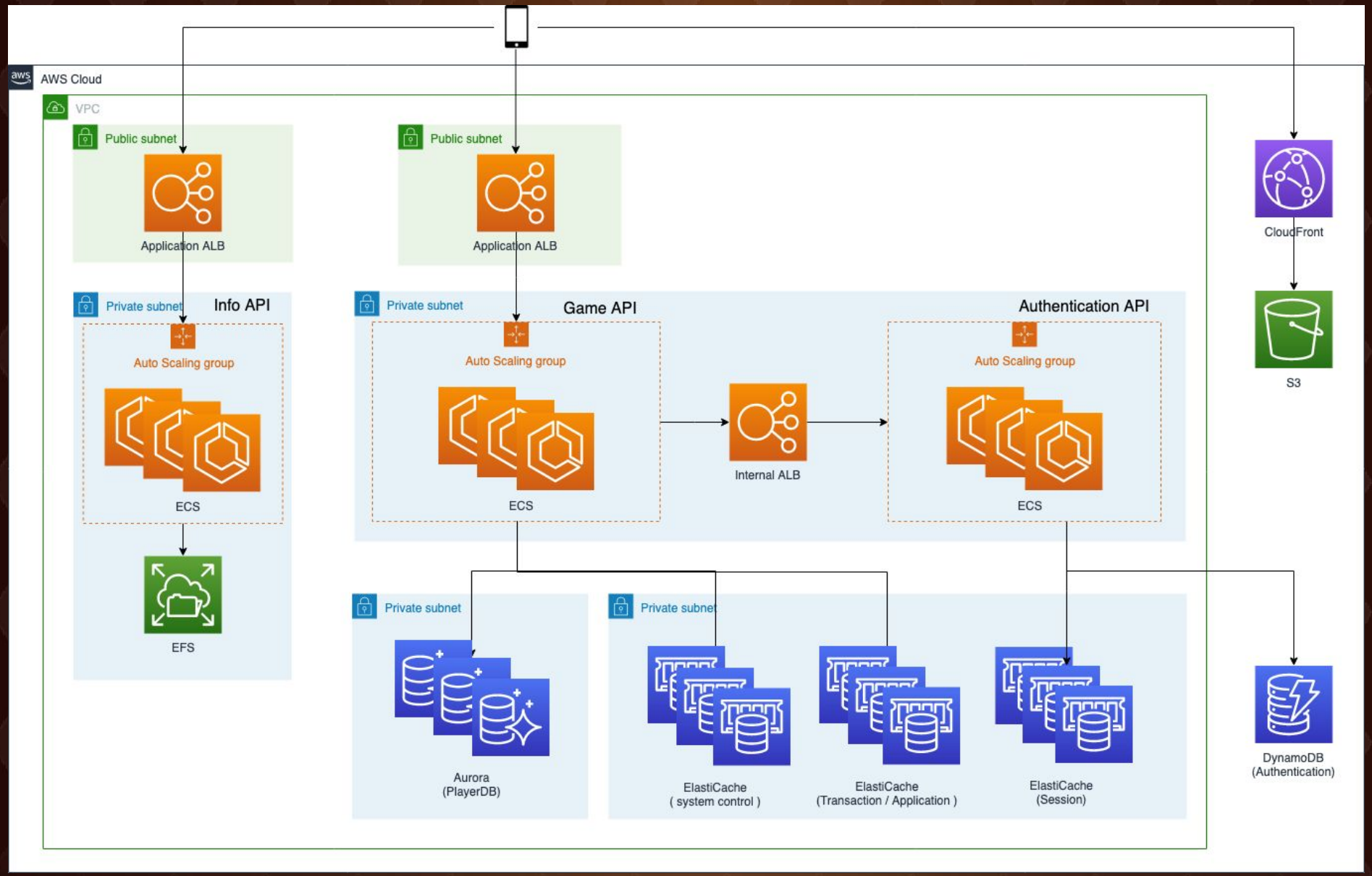
1. アーキテクチャ編

1章ではなすこと

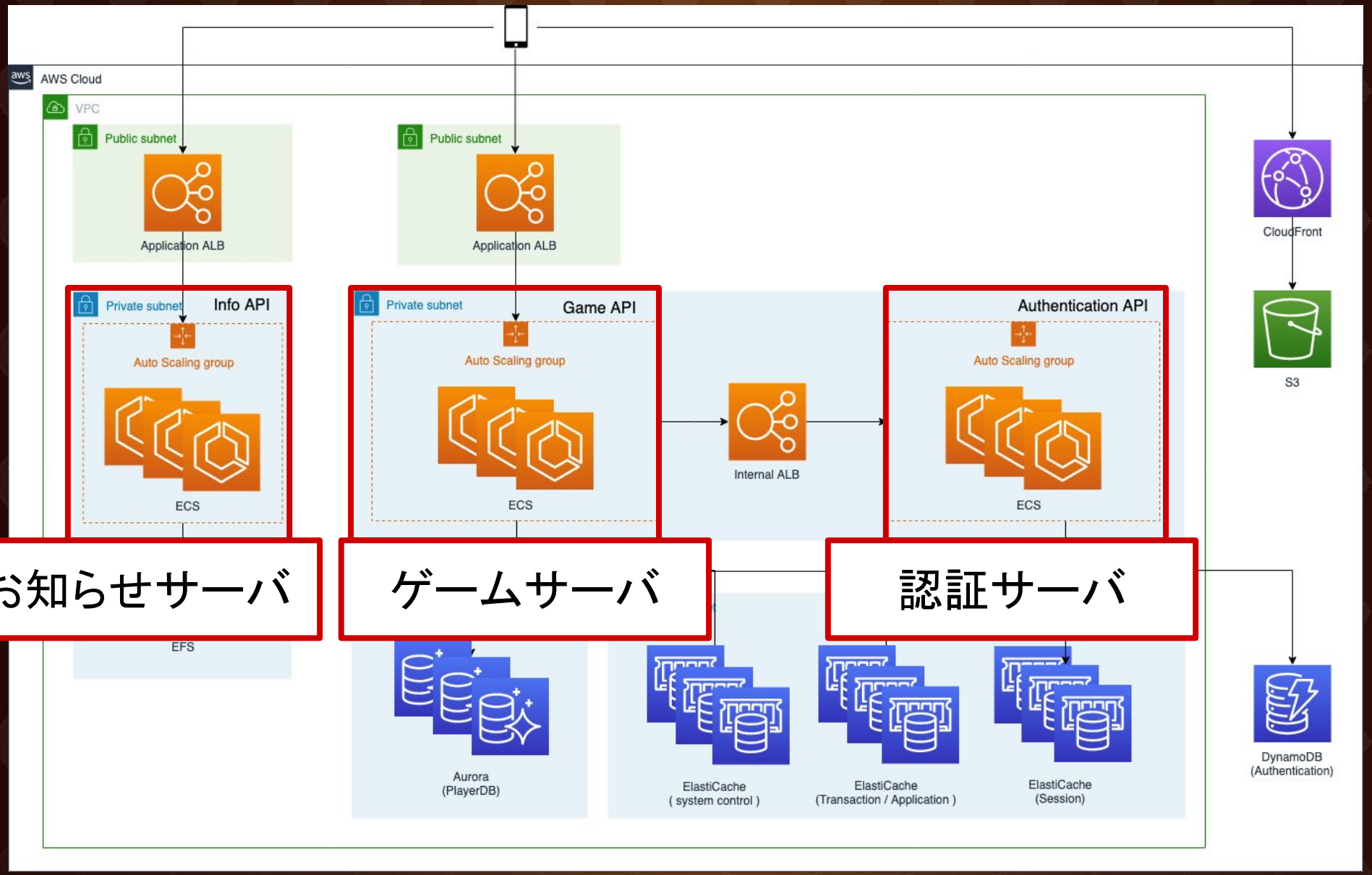
1. アーキテクチャ全体像
   - アーキテクチャ図
2. アーキテクチャ構成要素
   - ソフトウェア群
   - インフラ構成管理
   - プロセス構成管理
   - ロギング / 監視 / 分析

アーキテクチャ全体像

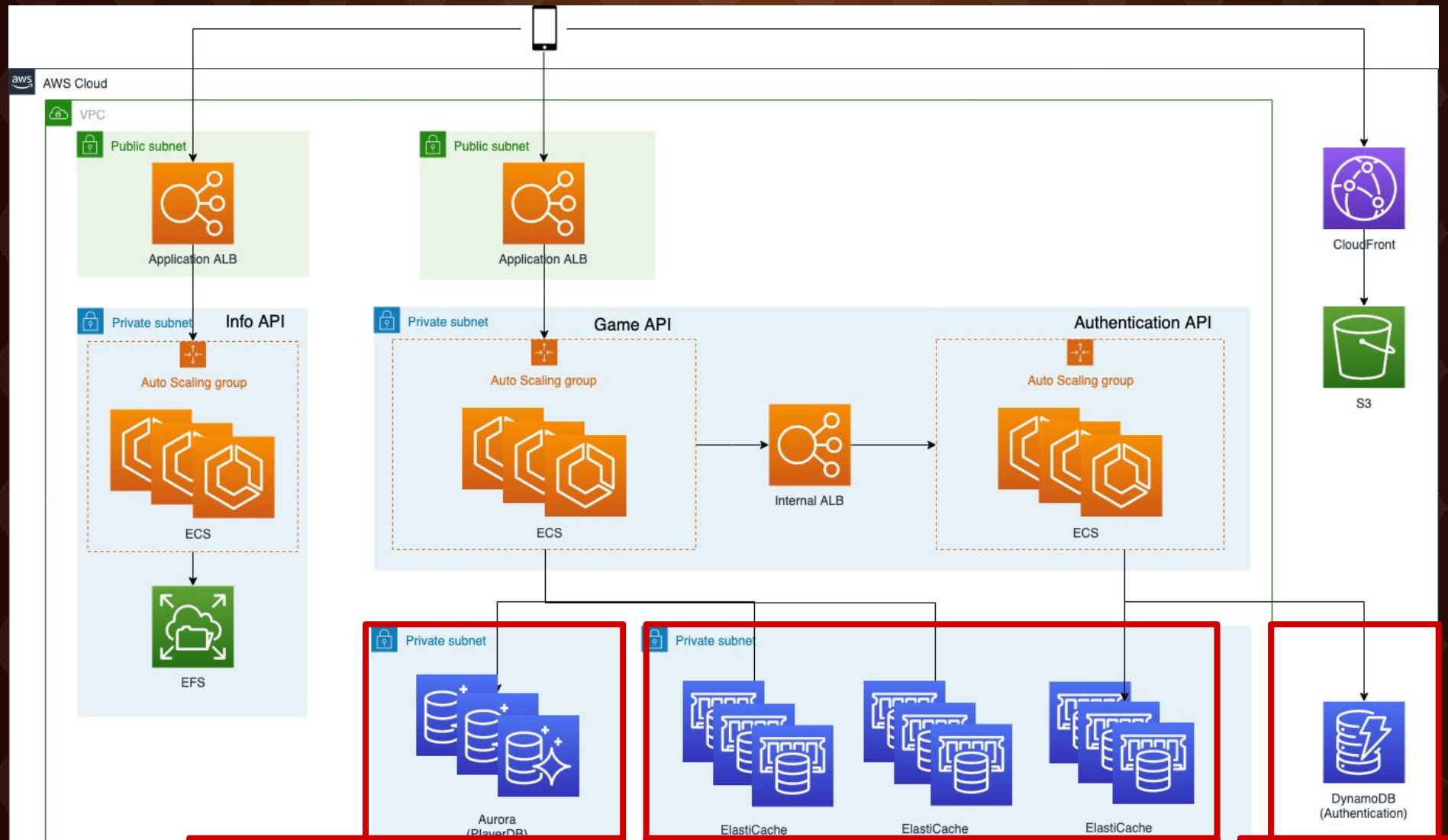
アーキテクチャ全体像


AWS Cloud
VPC
Public subnet
Application ALB
Public subnet
Application ALB
CloudFront
S3
Private subnet
Info API
Auto Scaling group
ECS
EFS
Private subnet
Game API
Auto Scaling group
ECS
Internal ALB
Authentication API
Auto Scaling group
ECS
Private subnet
Aurora
(PlayerDB)
Private subnet
ElastiCache
(system control)
ElastiCache
(Transaction / Application)
ElastiCache
(Session)
DynamoDB
(Authentication)

アーキテクチャ全体像

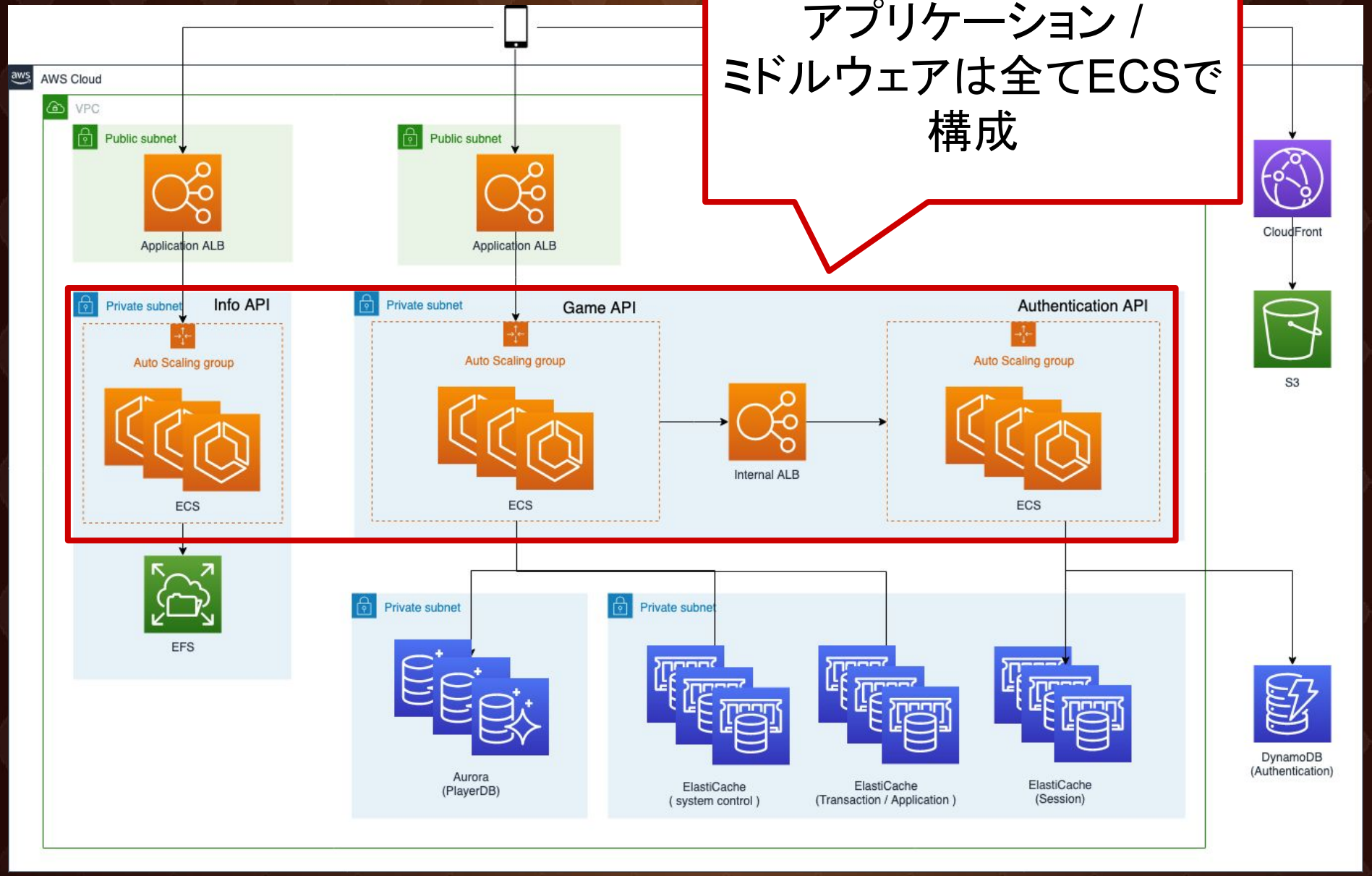

AWS Cloud
VPC
Public subnet
Application ALB
Public subnet
Application ALB
CloudFront
S3
Private subnet
Info API
Auto Scaling group
ECS
Private subnet
Game API
Auto Scaling group
ECS
Internal ALB
Authentication API
Auto Scaling group
ECS
お知らせサーバ
ゲームサーバ
認証サーバ
EFS
Aurora
(PlayerDB)
ElastiCache
(system control)
ElastiCache
(Transaction / Application)
ElastiCache
(Session)
DynamoDB
(Authentication)

アーキテクチャ全体像

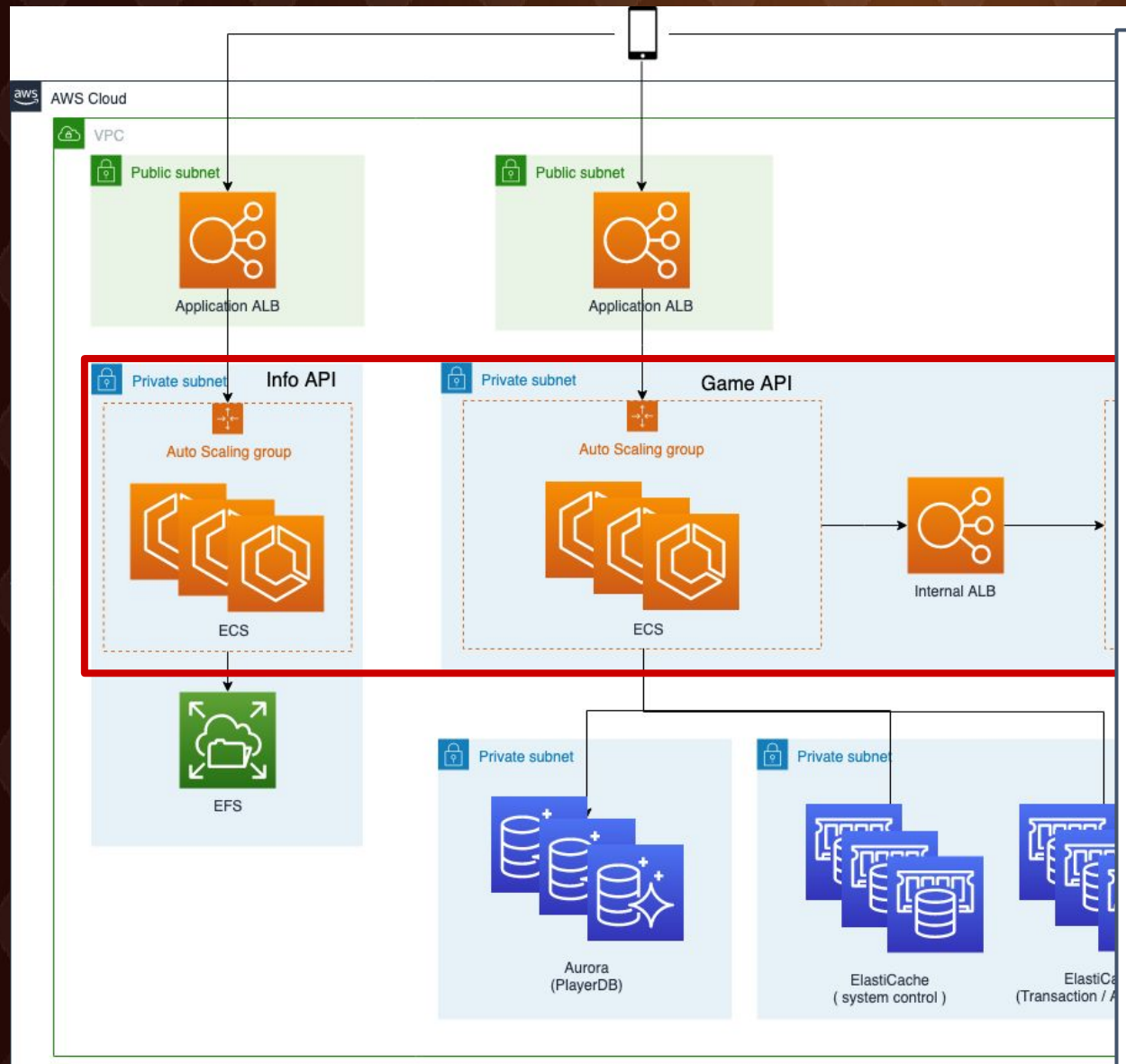

AWS Cloud
VPC
Public subnet
Application ALB
Public subnet
Application ALB
CloudFront
S3
Private subnet
Info API
Auto Scaling group
ECS
EFS
Private subnet
Game API
Auto Scaling group
ECS
Internal ALB
Authentication API
Auto Scaling group
ECS
Private subnet
Aurora
Private subnet
ElastiCache
ElastiCache
ElastiCache
DynamoDB
(Authentication)
プレイヤーデータ
session / cache
認証データ

アーキテクチャ全体像

アプリケーション /
ミドルウェアは全てECSで
構成



アーキテクチャ全体像



ゲームサーバ / 認証サーバ

NGINX elixir fluentd DATADOG

お知らせサーバ

NGINX python

運用ツール等

python LOCUST

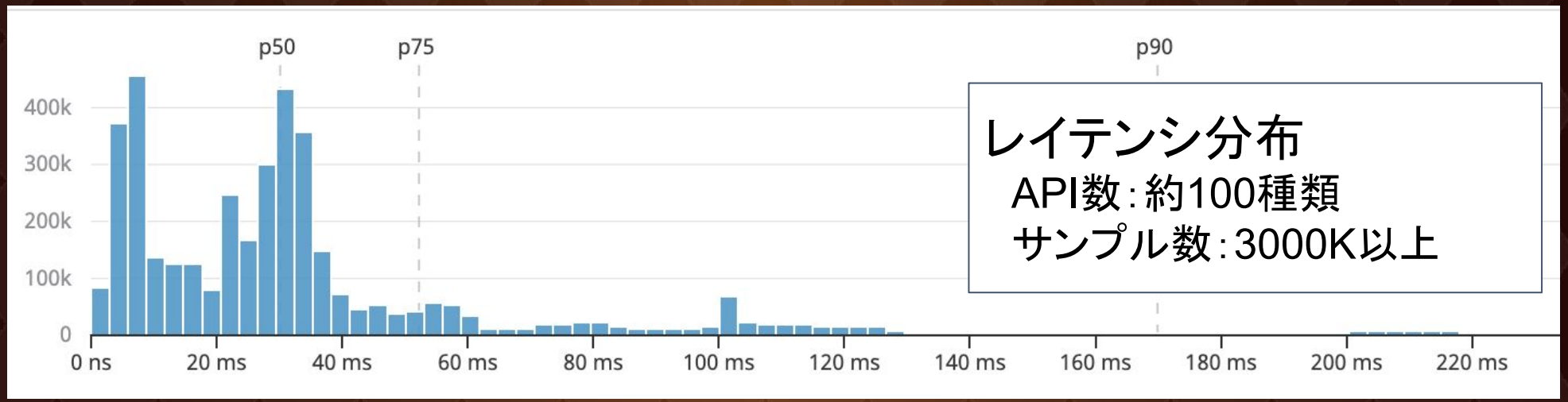
構成管理するプロセスは全てDockerで運用

アーキテクチャ構成要素

アーキテクチャ構成要素

サーバサイドの開発言語はElixir

- 並列性が高く、十分に高速



※ 参考：ロマサガRSにおけるElixir サーバー開発実践
～生産性を上げてゲームの面白さに注力～

https://speakerdeck.com/elixirfest/romasakars-niokeru-elixir-sahakai-fa-shi-jian-sheng-chan-xing-woshang-ketekemufalsemian-bai-sanizhu-li

アーキテクチャ構成要素

インフラ構成管理

- 99%のAWSリソースをCloudFormationで管理
  - 当然、環境変更のオペレーションミスは**0**件
  - 複数の開発環境間での差分がなくなる
  - 新しい開発環境も1コマンドで出来上がる
- kumogata/kumogata2でコード管理
  - Ruby製 CloudFormationラッパーツール
  - 条件分岐などのコードが書ける
  - 秘匿情報を環境変数化してSSMから取得など

アーキテクチャ構成要素

プロセス構成管理

- 100% Docker運用
  - 構成管理するプロセスは全てDocker化している
  - 安定したデプロイ
    - デプロイ / スケーリングトラブルはリリース後1度もない
  - デプロイしているものと同じImageをローカルで起動
- オーケストレーションはECSを採用
  - デプロイ / デプロイロールバック
  - オートスケーリング
  - オートヒーリング
  - 本番リリース時は、EKSはまだ東京GAではなかった

アーキテクチャ構成要素

Fargateは？

- Pros
  - サーバレス
  - 簡単にスケールする
- Cons
  - スケール時間
    - EC2のスケールアウト時間と変わらず（採用当時はもっと遅かった）
  - ロギング
    - 当時はCloudWatchLogsのみのサポートだった

一部のサーバはFargateで本番運用したがEC2/ECSに戻した。

アーキテクチャ構成要素

ロギング

- CloudWatchLogs
  - エラーのみフィルタしたアプリケーション / ミドルウェアログ
- Datadog Logs Management
  - エラーのみフィルタしたミドルウェアログ
- S3
  - アクセスログ等。Athenaで分析。
- その他
  - ゲームプレイヤーの行動ログ
  - 今日は話せないデータウェアハウス

アーキテクチャ構成要素

監視

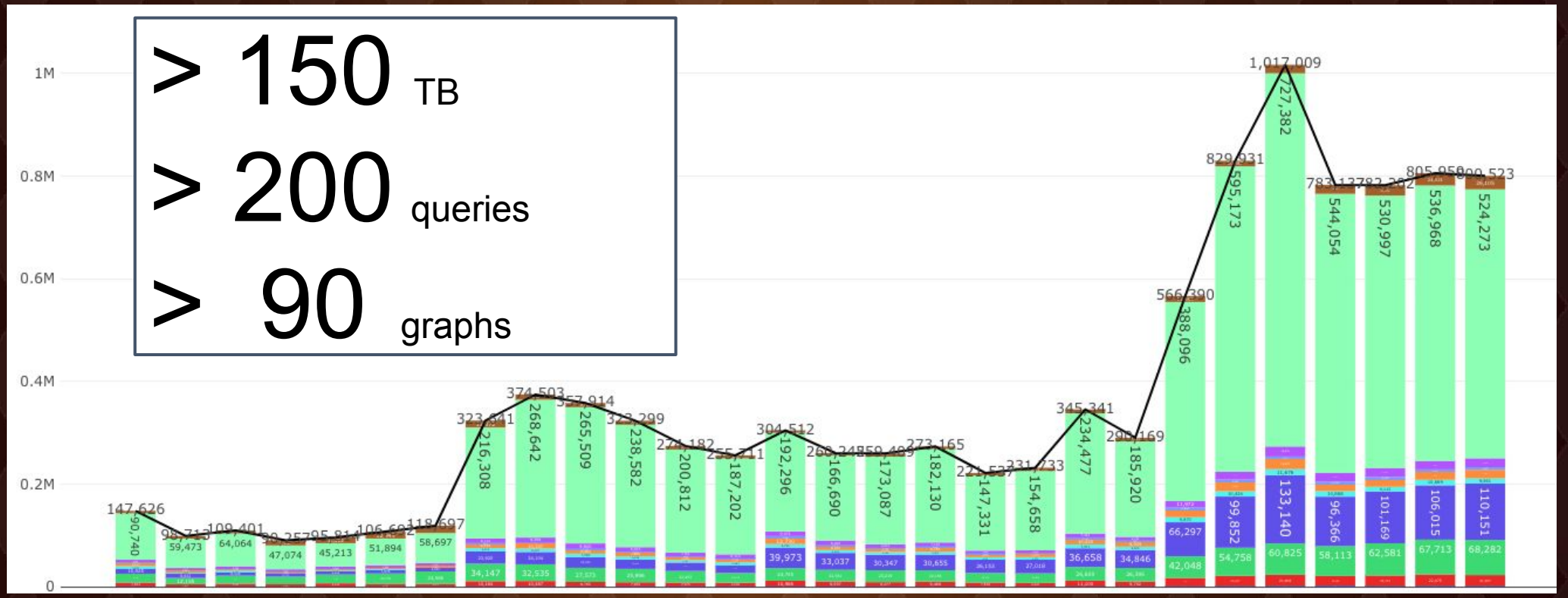
- CloudWatch
  - エラー監視、生死監視、リソース監視、ログ監視
- Datadog Infrastructure
  - エラー監視、生死監視、リソース監視、プロセス監視
- Datadog APM
  - 性能監視
- Datadog Logs Management
  - ログ監視
- Sentry
  - エラー監視

自動アラート、一部自動復旧 (※3章で少し触れます)

アーキテクチャ構成要素

分析

行動ログをRedashで可視化



1章まとめ

ロマサガRSのアーキテクチャ紹介

- 100%Docker運用
  - 構成管理するプロセスは全てDocker化
  - 安定したデプロイ/スケーリングを実現できている
- ほぼ100%CloudFormation管理
  - 環境構築/変更に伴うリスクは限りなく抑えられている
- ロギングや監視は各種サービスを導入し足りない機能を補っている

2. 負荷対策編

2章ではなすこと

1. 負荷テスト事例

2. ECSスケーリングの工夫

負荷テスト事例

負荷対策の心得

- 大前提として負荷テスト & 改善フェーズは必ず確保する
  - ギリギリに実施すると間に合わない可能性
- テストごとの目的 / ゴールを設定する
  - 負荷テストをして負荷を確認することが目的ではない

負荷対策編

負荷テストツール

- Locustを採用
  - シナリオのメンテナンス性を重視
    - Pythonで可読性が高く、多くのエンジニアが読み書きできる
    - レスポンスの結果をparse、計算結果を次のリクエストに渡すといった複雑なシナリオも簡単に書ける
  - Locust自体もDocker化し1コマンドでスケール
    - 数十コア巨漢インスタンス1台でslave50も簡単に起動
  - さらっと10000req/sとか出せる

目的ごとの負荷テスト

- 単性能テスト
- 過負荷テスト
- 高負荷テスト
- 障害テスト
- 長時間テスト

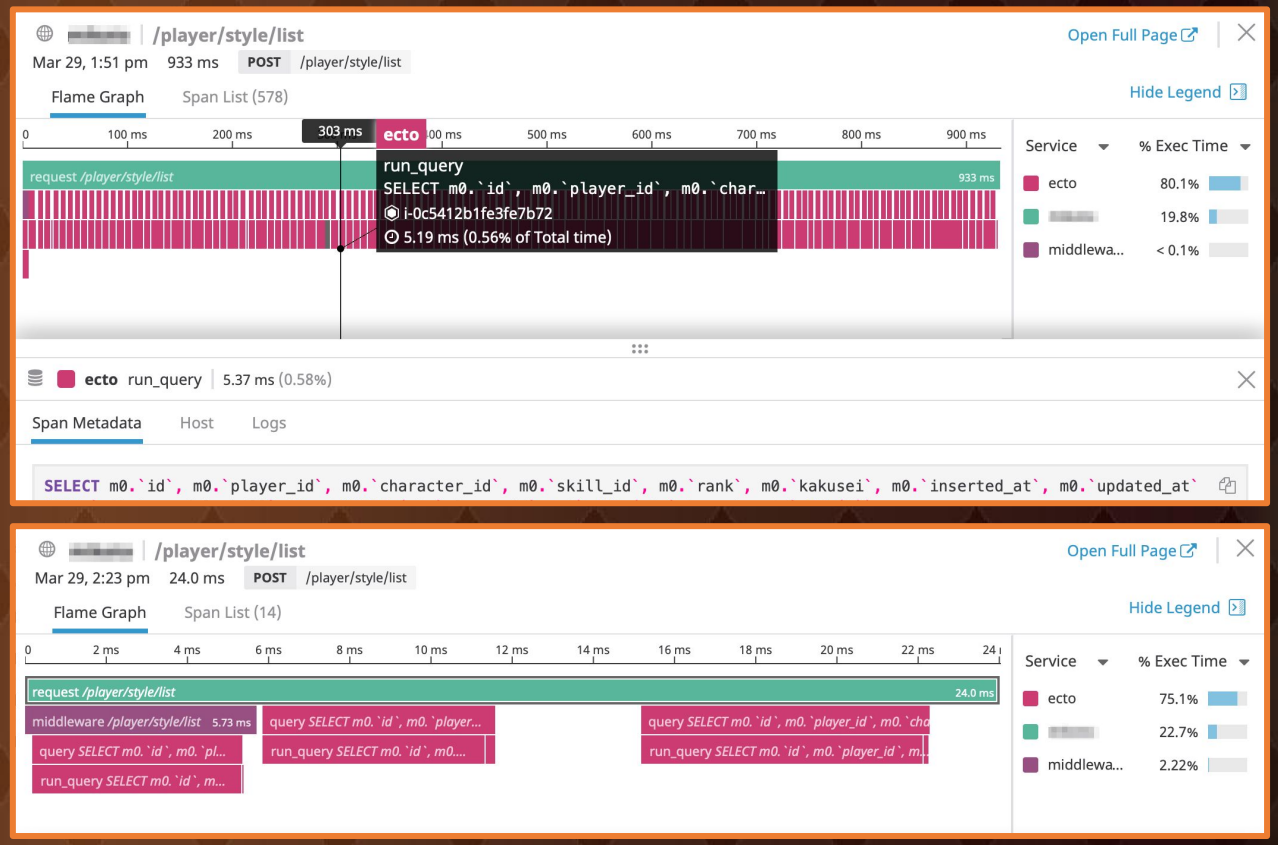
負荷対策編

単性能テスト

- アプリケーション特性をAPMで分析 & 改善
- 改善の繰り返し
- 低負荷でOK

N+1クエリ検出

改善



負荷対策編

過負荷テスト

- 各コンポーネントに過負荷をかける
  - 例）EC2にだけ過負荷
    RDSにだけ過負荷
    ※ 負荷テストシナリオに沿って負荷をかける

- ボトルネック、エラーになる箇所、原因を特定し修正
  - 例）EC2のCPUUtilization 70%ぐらいからエラーが出始める
    →どこかのパラメータの制約に引っかかっていないか？

負荷対策編

高負荷テスト

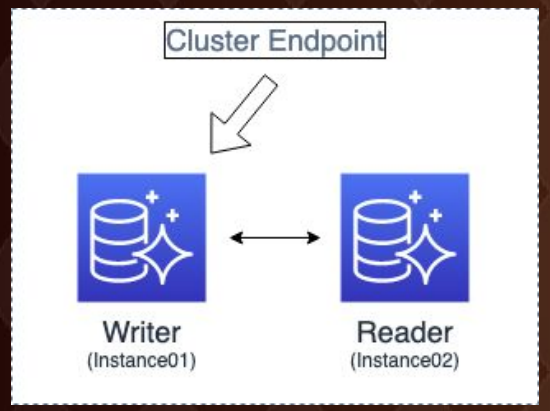
- スケールアウトの確認
  - EC2 / ECSスケールアウト確認
  - DB / ElastiCacheの垂直 / 水平分割の確認
    - リソースが分散されるか
- シナリオごとの負荷特性を確認
  - リセマラシナリオ
  - 特定の更新APIを大量に引くシナリオ
- システムサイジング

負荷対策編

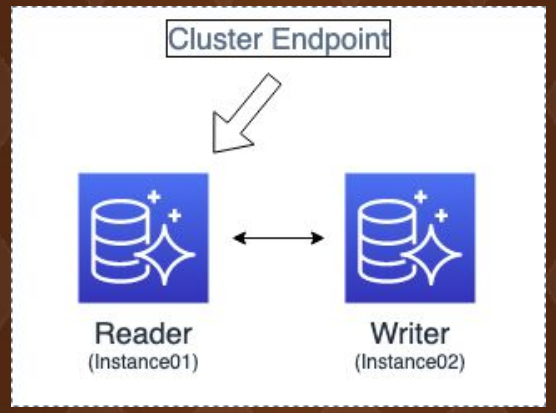
障害テスト

- 一部のコンポーネントに意図的に障害を起こして、サービスが継続するか確認する
  - 例：負荷テスト中にAuroraをfailoverさせる
    - Cluster EndPointのDNS切り替えが未完了時に再接続するとReader側のDNSをキャッシュしてしまう問題などを検知

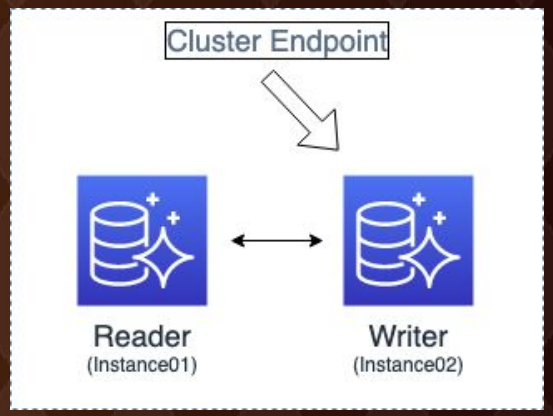
failover前



failover直後



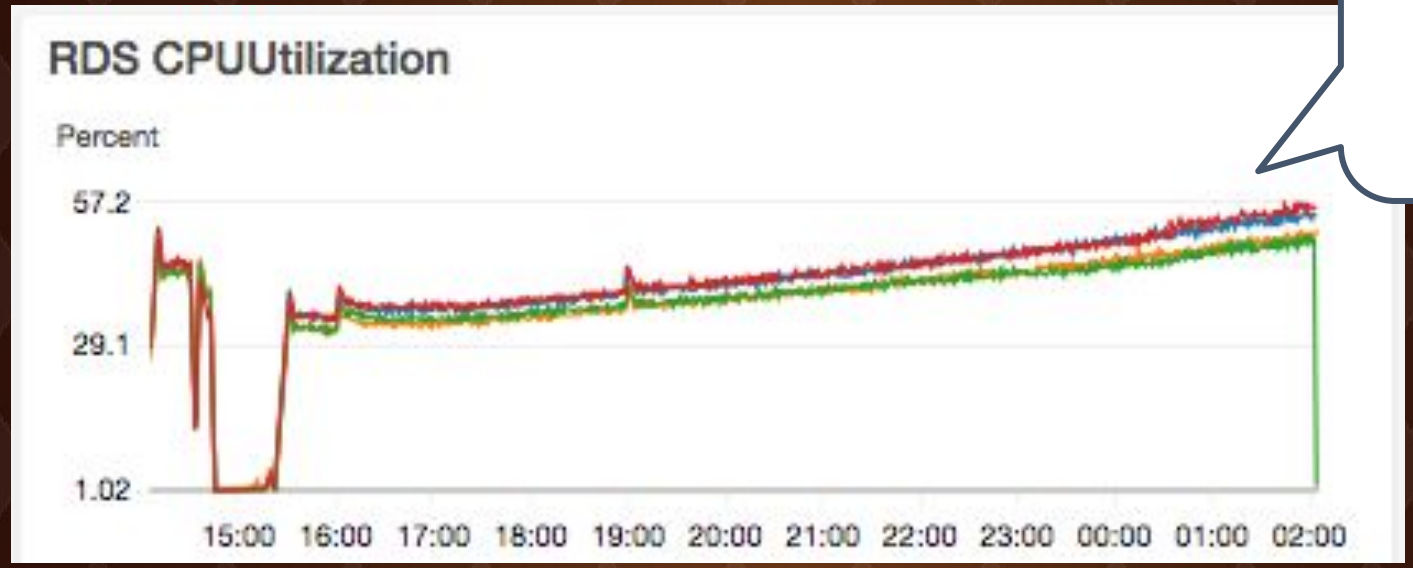
failover後



負荷対策編

長時間テスト

- 文字通り一晩中など、長時間かける。
  - メモリリークやfd枯渇などを検知できる可能性
  - CloudWatchLogs料金の肌感覚



O(N)のように見える
（危険信号）

負荷対策編

負荷テストのPDCAを高速化する工夫

- 素早くスケールする負荷テストツール
- プレイヤー資産を最大限付与するデバッグAPI
  - データを用意をする必要がなくなり気軽に負荷テストがかけれる
  - プレイヤー資産がないとN+1も検知しづらい。
- 何度でも壊して作り治せるインフラ(CloudFormation)
- メトリクス監視のCloudWatchダッシュボード

負荷対策編

実際にテストをしたからこそ発見できた問題

- N+1問題
- DBのパラメータ上限
- 長時間負荷によるSlowQuery検知
- 巨大レスポンスによるネットワーク帯域圧迫
- Aurora failover時のDNS切り替えが遅い問題
- 他いろいろ…

よくあるものを踏むべくして踏んでますが、
これらの多くををリリース前に検知 & 修正できたことが
安定運用につながったと思います

負荷対策編

リリース後

700

containers (max)

100 %

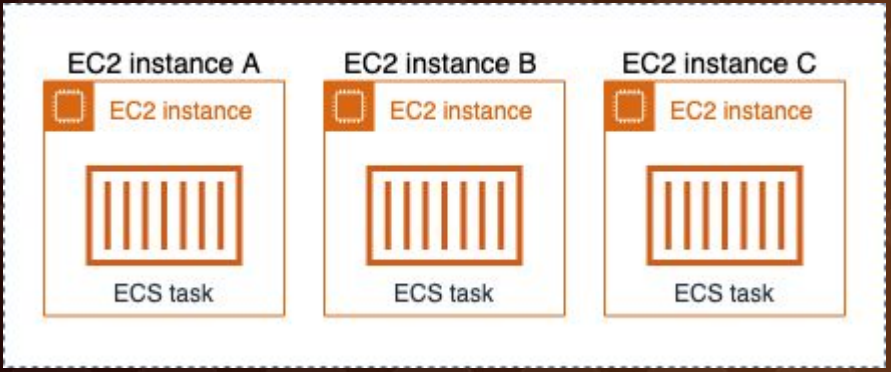
availability

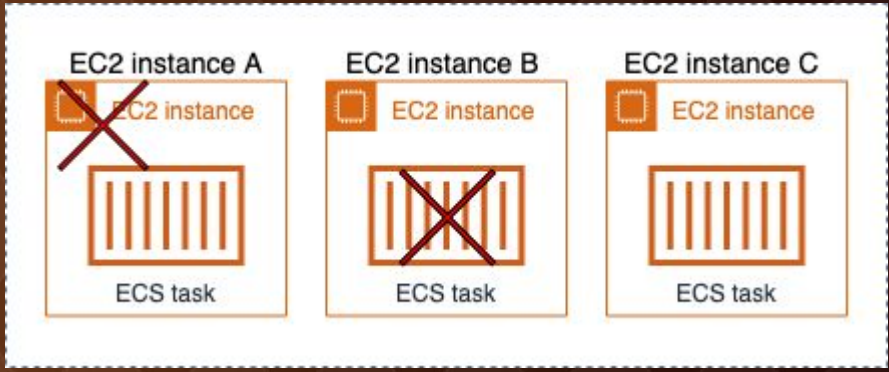
ECSスケーリングの工夫

ECSスケーリングの工夫

ECS ApplicationAutoScalingの課題

- スケールアウト
  - スケールアウトするにはEC2リソースが用意されていることが前提
  - EC2とECSの2つのAutoScaling設定
- スケールイン
  - スケールインするEC2上のECSタスクを停止させたいが、
    停止するEC2とECSタスクが一致する保証がない





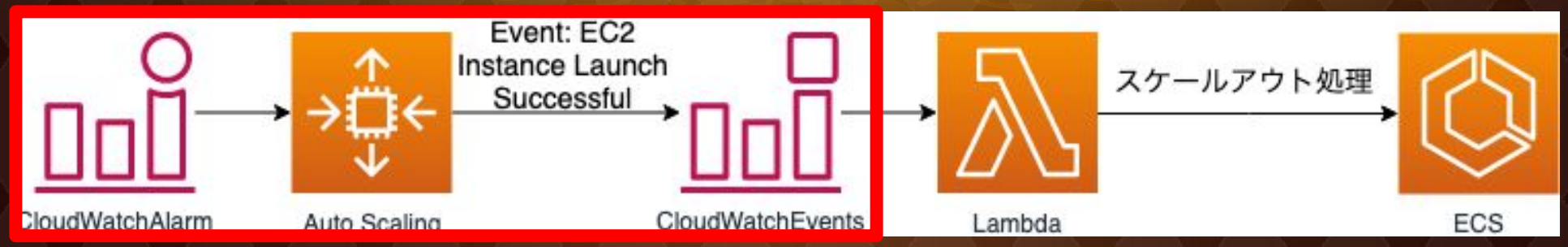
ECSスケーリングの工夫

スケールアウト

※ 1EC2インスタンス ＝ 1ECSサービスを前提としています

ECSスケーリングの工夫

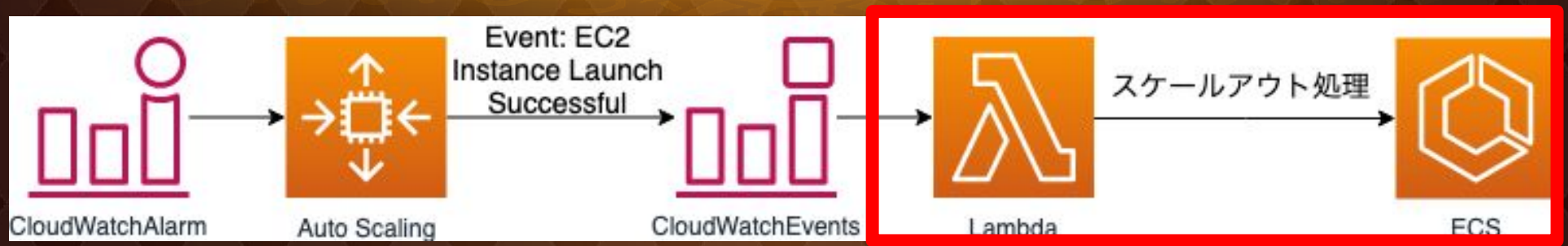
スケールアウト



- CloudWatchAlarmによってEC2がスケールアウト
- EC2起動完了のイベントをHookしてLambda実行

ECSスケーリングの工夫

スケールアウト



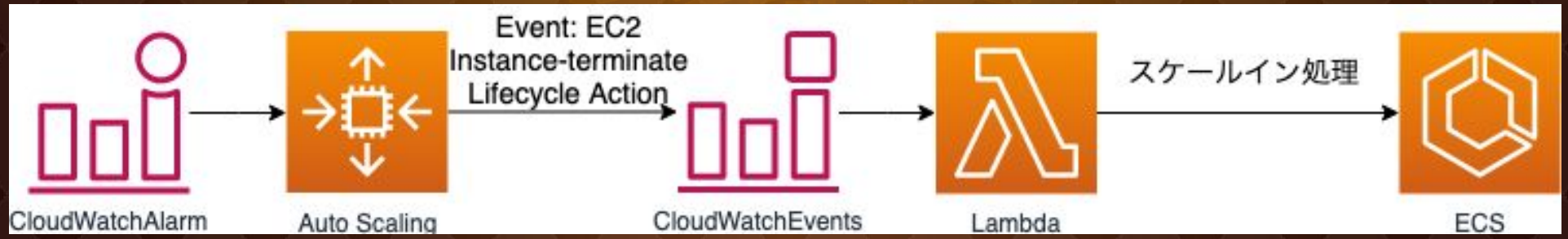
- AutoScalingのDesired Capacity値を取得して
  ECSタスク数をDesired Capacityに一致させる

```
autoscaling_group_name = event['detail']['AutoScalingGroupName']
autoscaling = autoscaling_client.describe_auto_scaling_groups(AutoScalingGroupNames=[autoscaling_group_name])
desired_capacity = autoscaling['AutoScalingGroups'][0]['DesiredCapacity']

ecs_client.update_service(cluster=cluster, service=service, desiredCount=desired_capacity,)
```

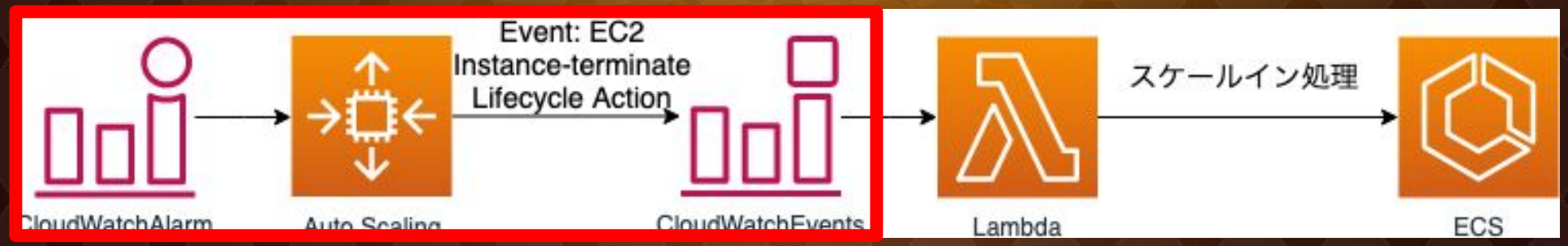
ECSスケーリングの工夫

スケールイン



ECSスケーリングの工夫

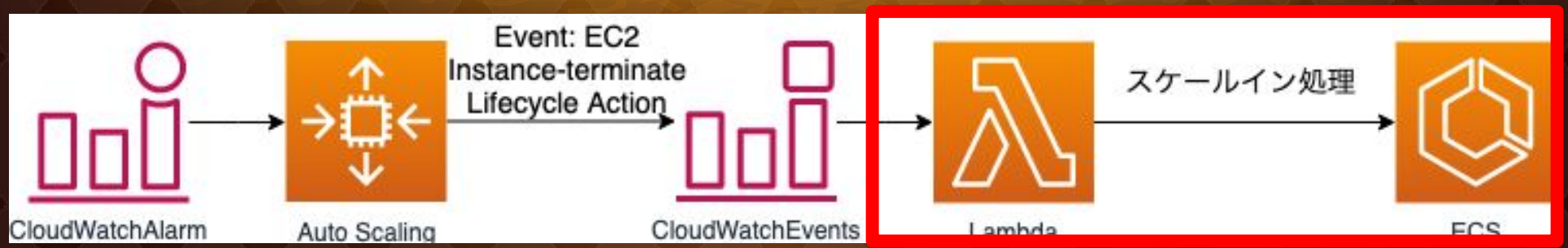
スケールイン



- EC2 AutoScalingにEC2_INSTANCE_TERMINATINGのLifeCycleHookを設定しておく
- CloudWatchAlarmによってEC2がスケールインした際に、Terminating:Wait状態に移行
- EC2_INSTANCE_TERMINATINGのイベントをHookしてLambda実行

ECSスケーリングの工夫

スケールイン



- EC2 instance idからECS container instance idを特定
- コンテナインスタンスをDRAINING
- DRAINING完了まで待つ
- ECSタスクの停止
- deregister container Instance
- complete Lifecycle Action
- update Service

```
ecs_client.update_container_instances_state(
    cluster=cluster,
    containerInstances=[container_instance_id],
    status='DRAINING'
)
wait_container_instance_draining(ec2_instance_id, autoscaling_group_name)

ecs_client.stop_task(
    cluster=cluster,
    task=task_arn,
    reason='scale in from lambda'
)

ecs_client.deregister_container_instance(
    cluster=cluster,
    containerInstance=container_instance_id,
    force=True
)
```

ECSスケーリングの工夫

スケーリングの工夫によって実現していること

- EC2のCloudWatchAlarm だけでECSスケーリング
- 停止したいEC2インスタンスのECSタスクを停止
- スケールアウト時間は2分強
- スケールアウト/インによるトラブルなし

2章まとめ

負荷対策の事例紹介

- 負荷テストパターンおよび工夫
  - 負荷テストを繰り返し実施できる仕組みを整備し、目的ごとの負荷テストを行うことで、多くの問題をリリース前に検知 & 修正
- ECSのスケーリング
  - CloudWatchEvents + Lambdaを有効活用し、ECSスケーリングの課題を解決

3. 運用の工夫編

3章ではなすこと

1. デリバリーパイプライン事例の紹介

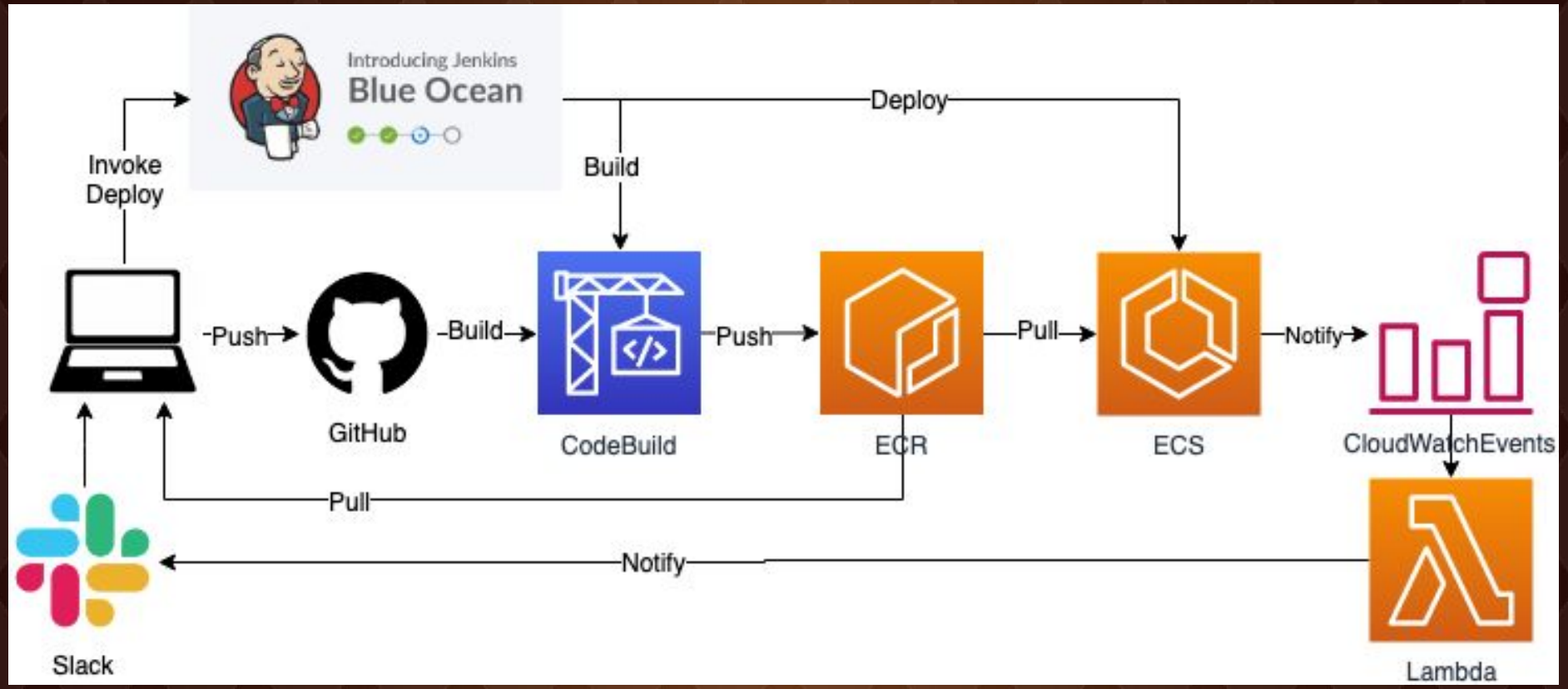
2. サーバレス運用の工夫

3. 自動復旧

デリバリーパイプライン

デリバリーパイプライン

ECSのパイプライン事例



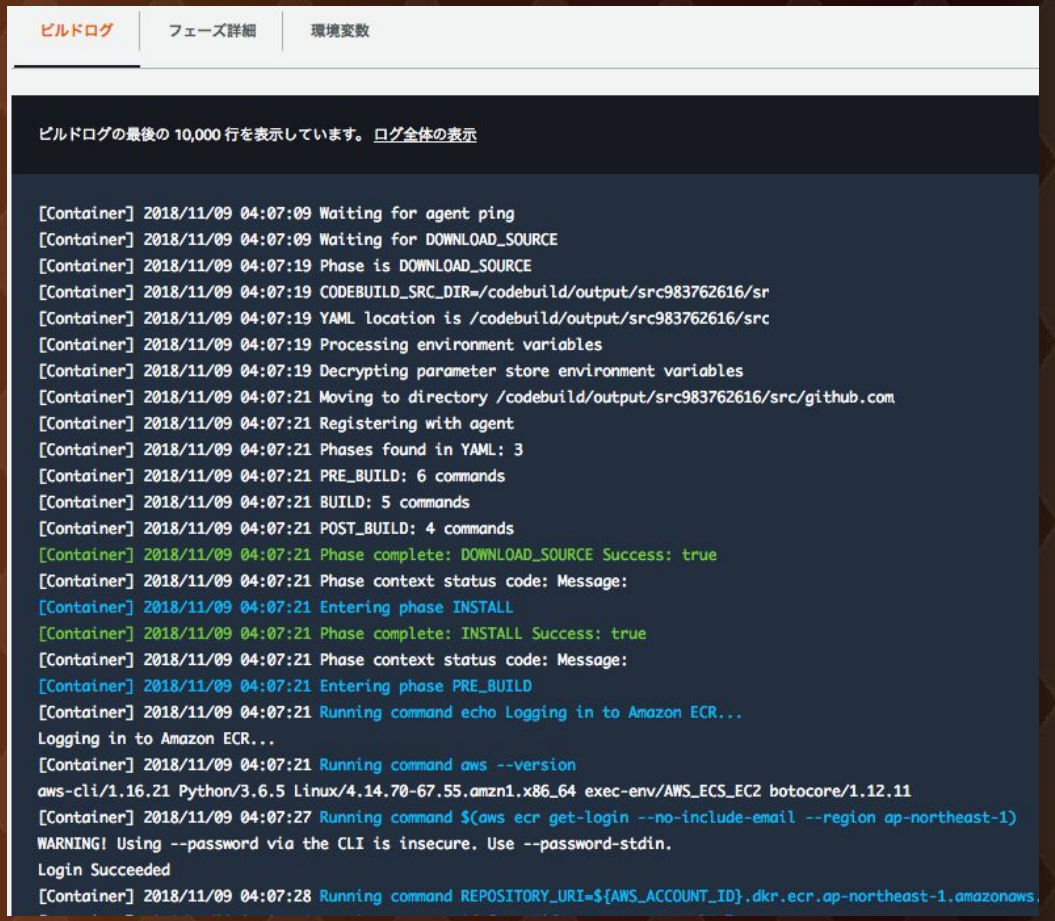
デリバリーパイプライン

パイプライン構成要素

ビルド環境：CodeBuild

- 並列性が高い
  - 複数Imageの同時ビルドが可能
- コスト
  - 従量課金で安い

- その他
  - SSMパラメータストア連携が便利
  - サーバレス
  - VPC内で実行可

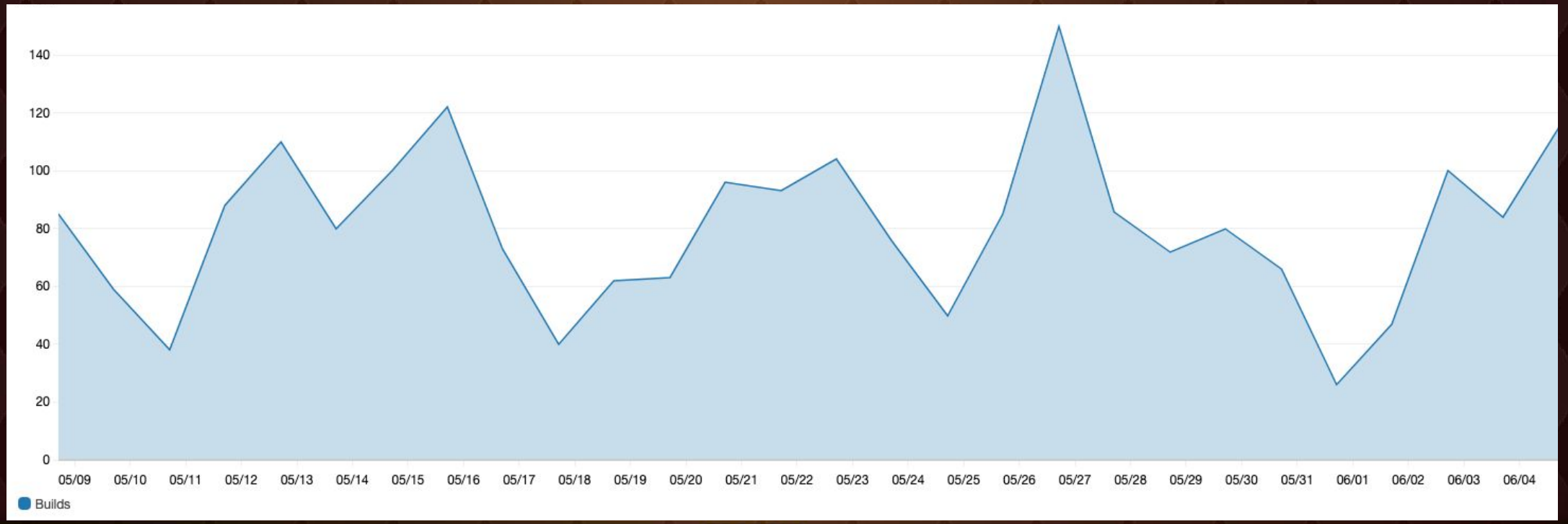


デリバリーパイプライン

並列性とコスト

>100 ビルド/day　　<$100 / month

※ Compute Types: medium

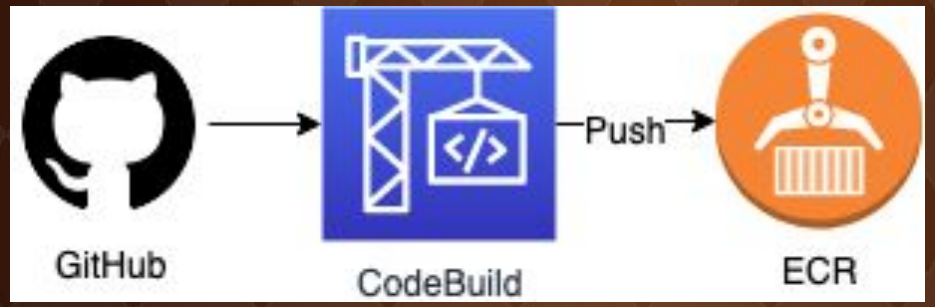


サーバレス運用の工夫

サーバレス運用の工夫

サーバレスバッチ

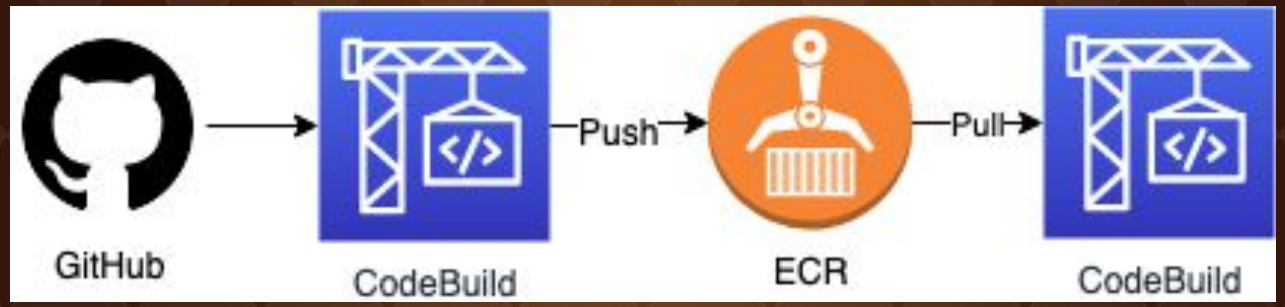
普通のCodeBuildの使い方



サーバレス運用の工夫

サーバレスバッチ

ビルド済みのDocker Imageを再利用することでCodeBuildをバッチ処理の実行環境として利用できる



※ 実装方法の詳細は https://speakerdeck.com/yutokomai/ecs-fargate-build-on-aws-codebuild

サーバレス運用の工夫

サーバレスバッチ

メリット

- ビルド / コンパイル済みのアプリケーションを再利用可能
- バッチサーバレス
- コスト減
  - EC2：24時間起動 (c5.xlarge: 4vCPU / 8Gメモリ)
    → **$154** / month （ + EBS料金 ）
  - CodeBuild：30 min/day (medium: 4vCPU / 7Gメモリ)
    → **$9** / month

サーバレス運用の工夫

サーバレスバッチ

デメリット

- プロビジョニングに30～40秒余計な時間

実際に使っている例

- アイテムドロップシミュレータ
  - 5～8 min
- DB並列マイグレーション
  - 例: 10 min * 並列数

自動復旧

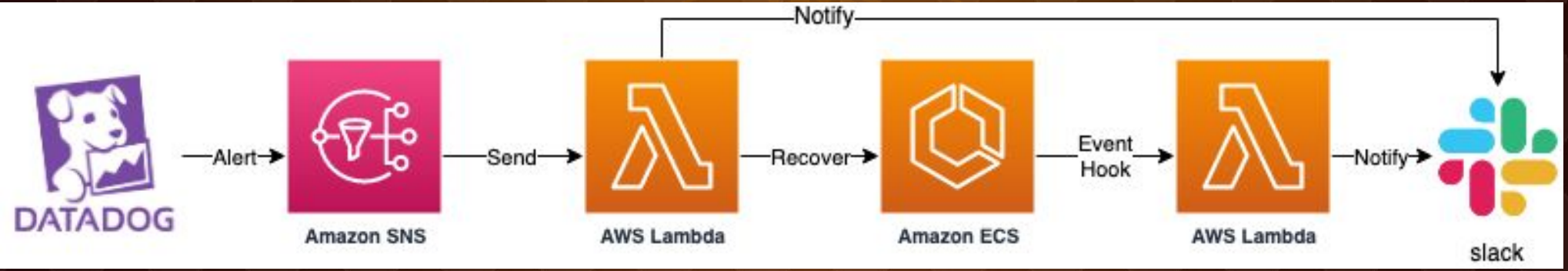
自動復旧

自動復旧

～ 自動復旧(auto-healing)はアラート疲れを避ける素晴らしい方法です。システムが巨大なら、それは必須と言っても過言ではありません ～

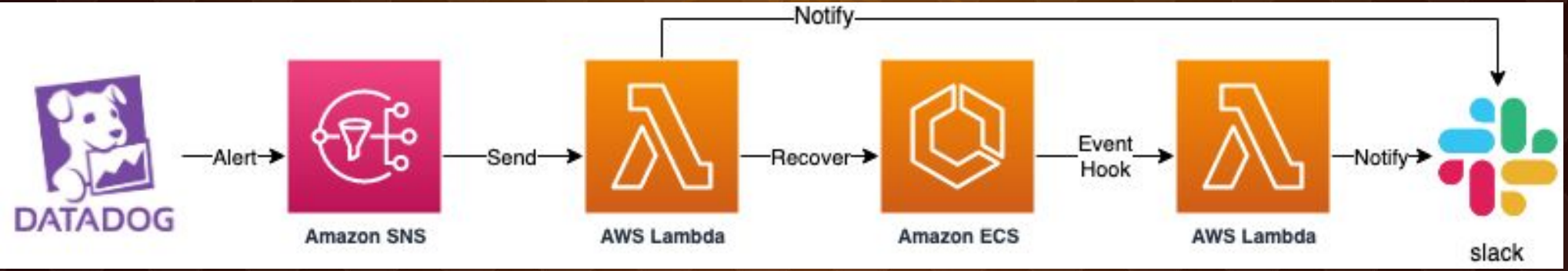
自動復旧

自動復旧(例)


Notify
DATADOG
Alert
Amazon SNS
Send
AWS Lambda
Recover
Amazon ECS
Event
Hook
AWS Lambda
Notify
slack

自動復旧

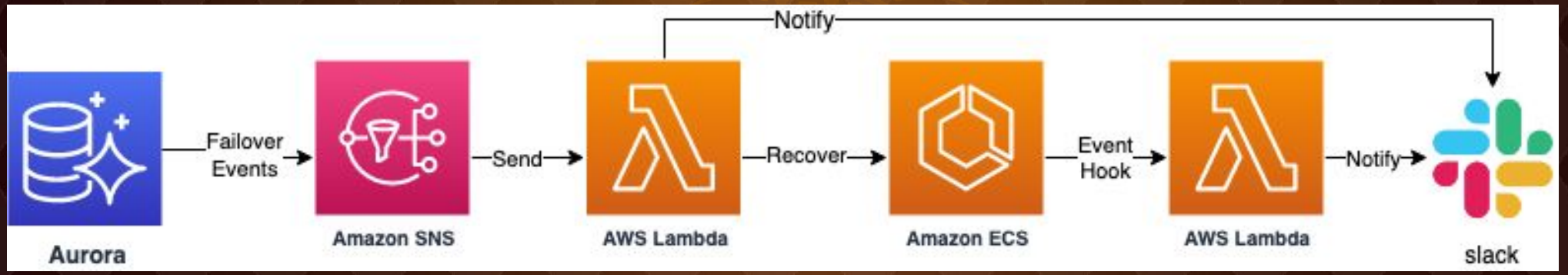
自動復旧（例）



- Datadog InfrastructureがAlertを検知
  - 例：メモリ枯渇
- SNSにメッセージを飛ばし、Lambdaでparse(EC2特定)
- 対象のEC2インスタンス上のECSタスクのみ再起動
- Slackに通知

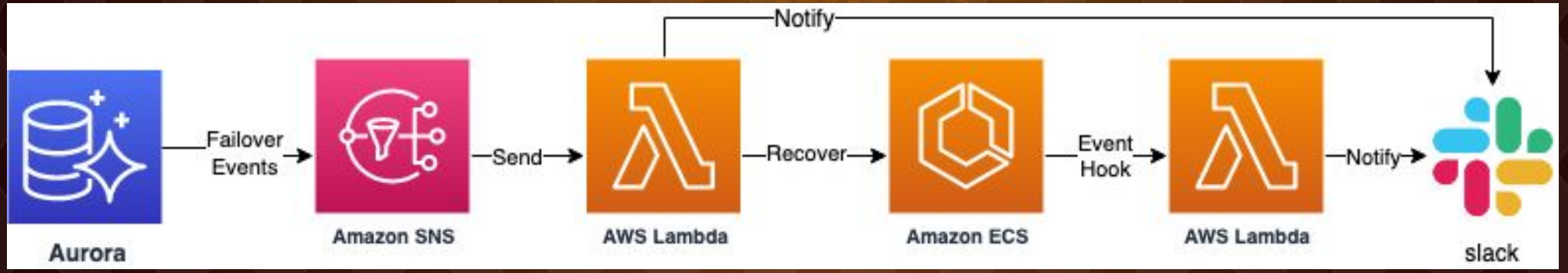
自動復旧

自動復旧(例)


Notify
Aurora
Failover Events
Amazon SNS
Send
AWS Lambda
Recover
Amazon ECS
Event Hook
AWS Lambda
Notify
slack

自動復旧

自動復旧（例）



- AuroraのfailoverイベントをRDSイベントサブスクリプションからSNSにメッセージを飛ばし、LambdaからECSを再起動
- Slackに通知

3章まとめ

運用の工夫紹介

- デリバリーパイプライン
  - CodeBuildを利用して並列性高くデリバリーできている
- CodeBuildを使用したサーバレス運用
  - 多くのバッチ処理をサーバレスで運用できている
- 自動復旧の事例を紹介
  - Datadog + Lambdaを利用し自動復旧。
    運用工数削減に繋がっている。

全体まとめ

1. アーキテクチャ編
   - ロマサガRSで採用している構成要素の紹介
   - 構成管理は全てCloudFormation化、Docker化しており
     環境変更、デプロイトラブルは一度もなく運用できている

2. 負荷対策編
   - 負荷対策のために実際に行った負荷テストパターンを複数紹介した

3. 運用の工夫編
   - サーバレス運用、自動復旧により、運用削減に注力している

ご清聴ありがとうございました