「家族アルバム みてね」における運用管理・オブザーバビリティの全貌

@IT 運用管理セミナー 2024 夏　基調講演 2

Vantageスタジオ みてねプロダクト開発部 プラットフォームグループ
清水 勲

自己紹介

清水 勲 @isaoshimizu
家族アルバム みてね Engineering Manager（SRE/CRE/Security）

新卒入社　　**ミクシィ（現MIXI）入社**　　**2022年1月～EM**

SIer時代（受託・自社開発）	SNS「mixi」	モンスターストライクなど	家族アルバム みてね
2003年	2011年	2014年	2018年
C/C++/C#/PHP/Python/iOS/AWS	Fedora/MySQL/LXC/OpenStack	Linux/MySQL/Ruby/ハイブリッド&マルチクラウド	AWS/MySQL/Ruby/マネジメント

プライベート

- 週末は社会人吹奏楽団での活動（楽団長、トロンボーン、たまに指揮）
- 音楽とキャンプとクラフトビールが好き
- New Relic User Group (NRUG) 運営

アジェンダ

1. MIXI GROUPの事業領域
2. 家族アルバム みてね についてご紹介
3. 家族アルバム みてねのインフラの変遷
4. オブザーバビリティの事例
5. SREチームの日々の運用
6. ポストモーテム運用
7. 現在の運用課題
8. まとめ

1 MIXI GROUPの事業領域

MIXI GROUPの事業領域

エモーションと
コミュニケーションで
「心もつながる」場と機会を
創造し続けます。

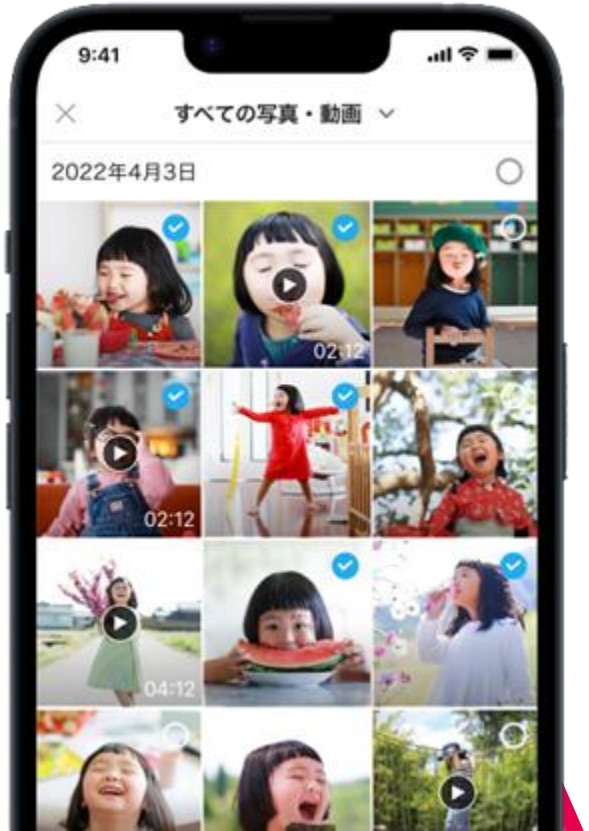
MIXI GROUPは、
ただ「つながればいい」という効率的な機能の提供ではなく、
歓喜や興奮、温かな思い、幸せ、居心地の良さの共有を通じて、
その先に、もっと深くて濃く豊かな、心のつながりを生み出すような、
サービスの開発・提供を目指しています。

現在、スポーツ・ライフスタイル・デジタルエンターテインメント
の3つの領域で事業を展開しており、
それぞれの主な事業内容は右の通りです。

また、近年の投資活動の拡大と重要性を勘案し、
FY2023からはスタートアップやファンド出資等の投資活動を事業化しました。

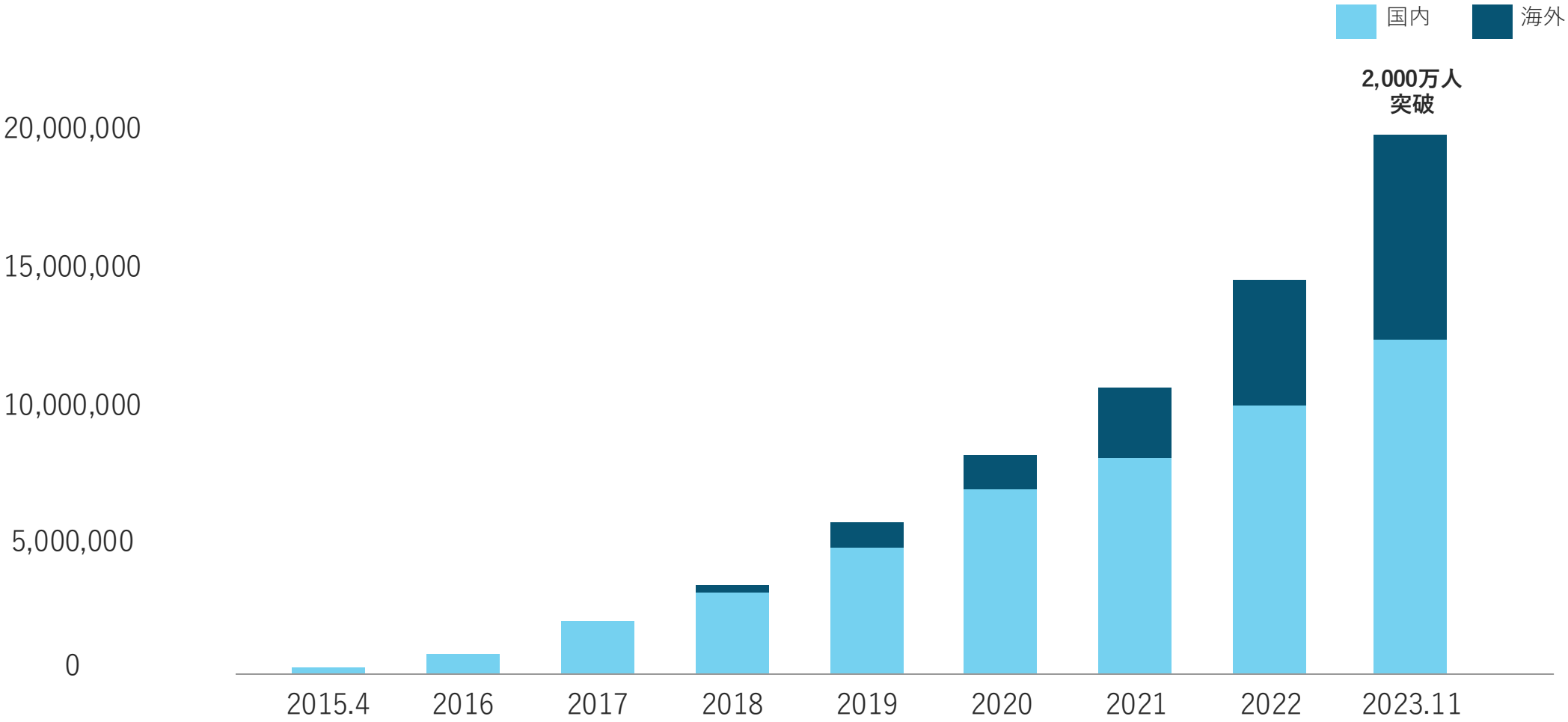
2 家族アルバム みてね についてご紹介

家族アルバム みてねはスマホで撮った子どもの写真や動画を家族と共有し、
コミュニケーションして楽しむ家族アルバムサービスです。



家族アルバム みてねの利用者数推移

2015年にリリース。7言語・175の国と地域で2,000万人以上の方にご利用いただいています。



※ iOS・Android™ アプリ登録者数、ブラウザ版登録者数の合計

写真関連商品

フォトブック・写真プリントで思い出をもっと楽しく。

みてねみまもりGPS

頼もしく成長していくお子さまを、みてねが一緒にみまもります。

運用事例をお話する前に知っておきたい

3 家族アルバム みてねのインフラの変遷

家族アルバム みてねのインフラの変遷

VMからコンテナへ

VMベースでの課題（特にChefとの組み合わせにおいて）

- デプロイ速度が遅い
  - 特にChefの適用に時間がかかる
- オートスケール速度が遅い
  - スケールアウトが遅い
- めったにVMが作り直されないことで古いOSやミドルウェアが残り続ける
- リソース効率の悪さ
  - インスタンスタイプの細かな調整に限界がある、手間がかかる
- cronサーバーのSPOF
  - スタンバイ機の準備はあるが、手作業での入れ替え作業は面倒

などなど

さらにAWS(OpsWorks)の制約による課題

- デプロイやスケール条件の設定などをおこなうためにWeb UIを操作することによる属人性、手間の多さ（職人が生まれやすい）
- EC2のコスト（スポットインスタンスが使えなかった）
- OpsWorks(Chef)における不具合のような挙動、改善がなかなか進まなかった

結果、Amazon EKS（Kubernetes）に移行しました
（4年越しプロジェクト）

詳細は「4年間のEKS移行の取り組みを振り返って」をご覧ください

https://gihyo.jp/article/2022/11/mitene-02eks

Kubernetes環境

家族アルバム みてねとKubernetes

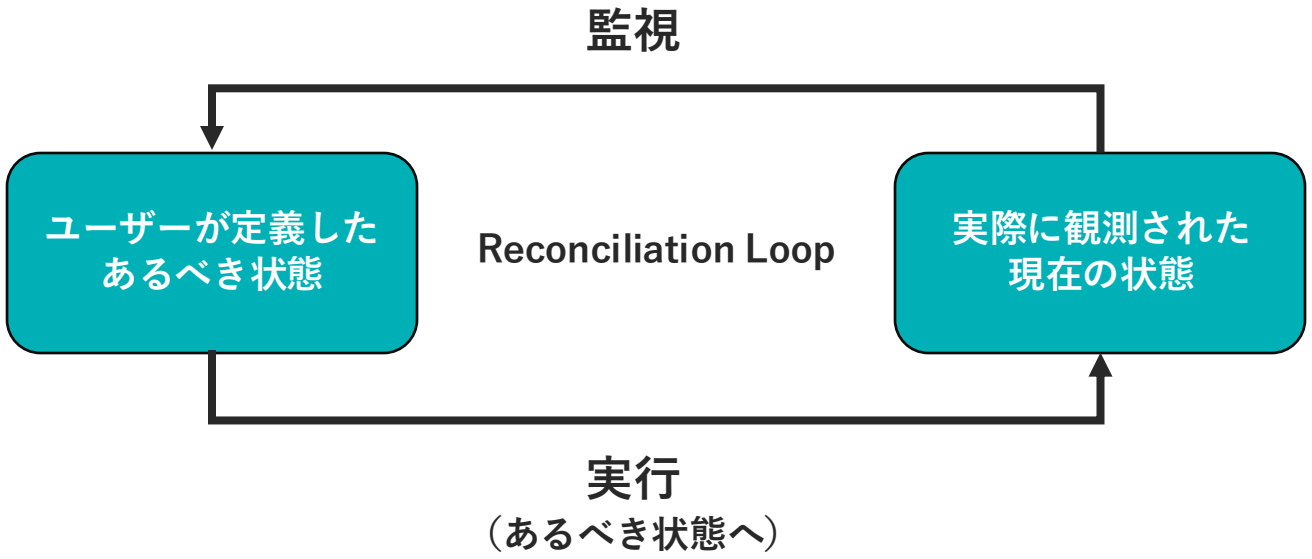
- 現在、すべてのサーバーアプリケーションがKubernetes(Amazon EKS)上で稼働している
- 開発環境、本番環境にそれぞれクラスタが1つずつ
- アプリケーションの種類によってネームスペースを分けている
- ノードとポッドの規模感 ※本資料作成時において
  - ノード数: 50〜800
  - ポッド数: 1,000〜14,000
  - オートスケールによって大きく変動し、時期によって差がある

Amazon Elastic Kubernetes Service (Amazon EKS)

Kubernetesの良いところ

Reconciliation Loop リコンシリエーション ループ（リコンサイルループ）

- あるべき状態と実際のシステムの状態を比較して、差分があればその差分をなくすための処理が実行される
  - 例えば、必要とするPod数が10と定義していて、もし何らかの原因でPod数が8になってしまった場合、Pod数を10にするための処理がおこなわれる



Kubernetesの良いところ

IaC (Infrastructure as Code)

- Kubernetesにデプロイするアプリケーションの定義、リソース、オートスケール条件などが宣言的にコードに記述される（YAML）
- 様々なリソースをHelm Chartとしてパッケージング

マニフェスト（YAML）の例

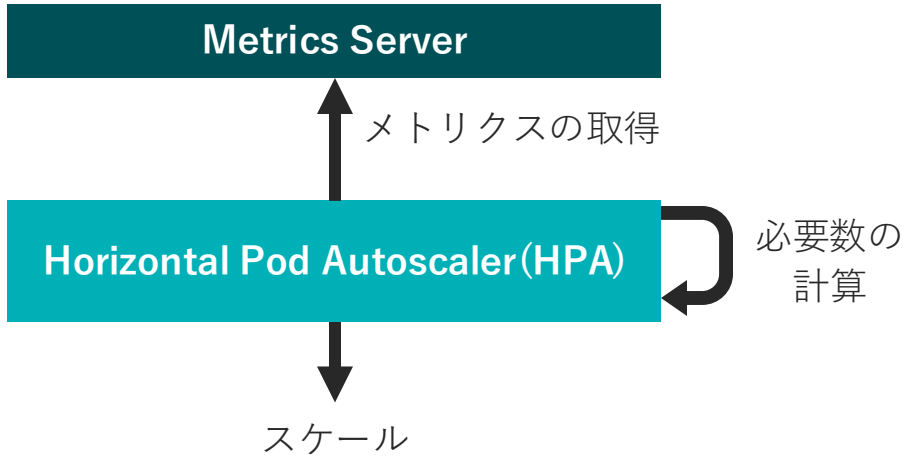
```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mitene
spec:
selector:
    matchLabels:
      app: mitene
  strategy:
    type: RollingUpdate
...
```

Kubernetesの良いところ

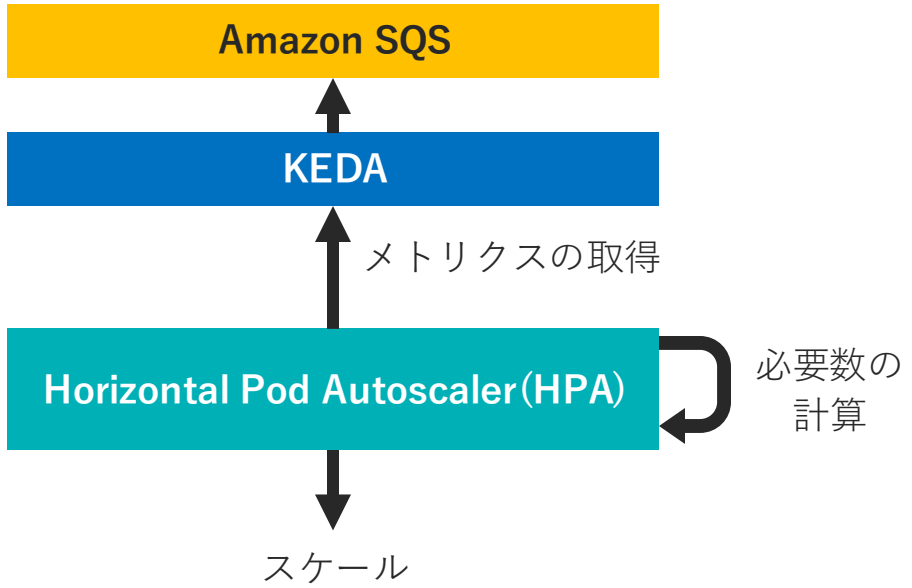
オートスケールの柔軟性

- 基本はCPU、メモリの利用状況に応じたスケール
- 外部のメトリクスの状況に応じたスケール（Amazon SQSやAmazon CloudWatch、Prometheusなど）
  - OSSである **KEDA** https://keda.sh/ を利用して実現

CPU、メモリをベースとしたスケール



KEDAを利用したスケール



Kubernetesにおける課題

クラスターバージョンのアップグレード作業

だいたい半年に1回くらいの頻度で実施している
（新バージョンリリース後14ヶ月間は標準サポートされる）

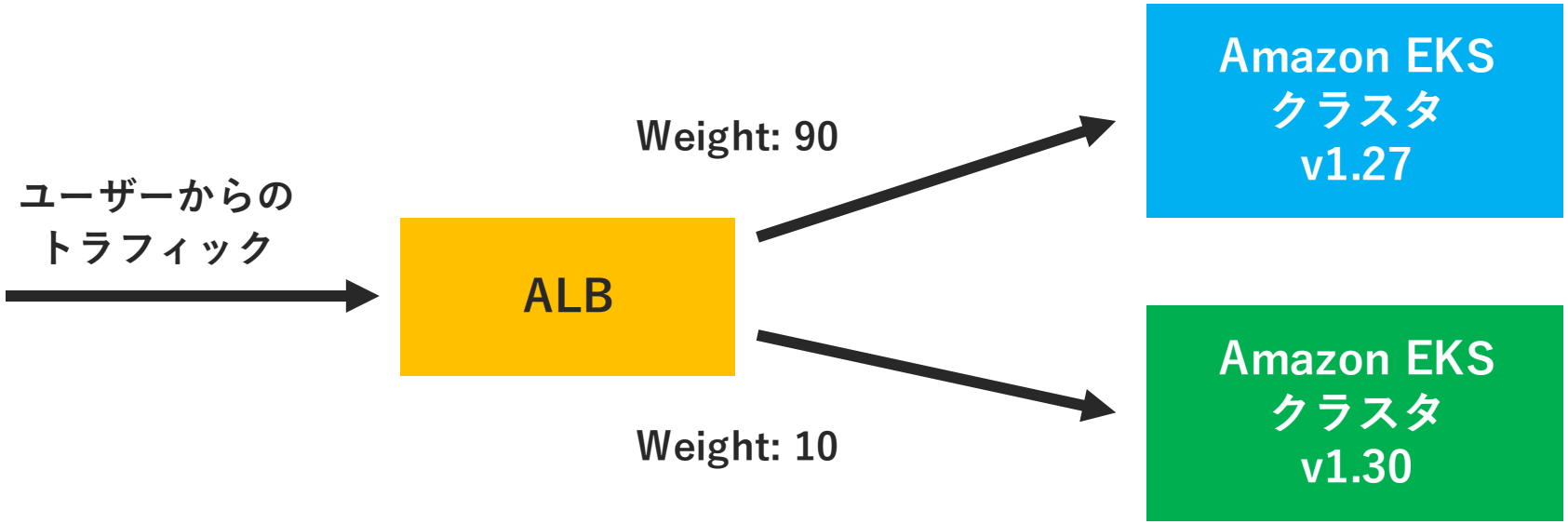
アップグレード作業の大まかな流れ

1. 新たに新バージョンのKubernetesのクラスタを立てる
2. 新クラスタにアプリケーションをデプロイ
3. 新クラスタに徐々にトラフィックを流す
4. 新クラスタ側のジョブワーカーを起動する
5. 旧クラスタ側のジョブワーカーを停止する
6. 旧クラスタを削除する

Kubernetesにおける課題

クラスターバージョンのアップグレード作業

- APIバージョンの変更や廃止への追従（Helm Chartを最新にするなどの対応）
- アップデート時のリスクを減らすためにBlue-Green方式を採用
- 回数をこなすごとに手順が改善されてきている



徐々に新しいクラスタへ移行していく

CI/CD環境

CI/CD環境

GitHub Actions, CircleCI

- RuboCopによるコードの静的チェック
  - コーディングルールへの準拠、リファクタリングの警告など
- Dangerによるコードの静的チェック
  - 非推奨事項の指摘など
- ユニットテスト
  - RSpec
- コンテナイメージのビルドとプッシュ
  - Docker Build、ECRへのプッシュ

Argo CD

- KubernetesのためのGitOpsツール
- テストとビルドが通ったあと、新たなイメージを使うようにマニフェストを書き換えることで新しいイメージがデプロイされる（Podが新たなイメージで起動するように置き換わる）

IaC (Infrastructure as Code) 環境

IaC (Infrastructure as Code) 環境

2018年からTerraformを利用してきました（それまではほとんどが手作業）

Terraformの適用対象

- AWS
- Google Cloud
- GitHub
- New Relic

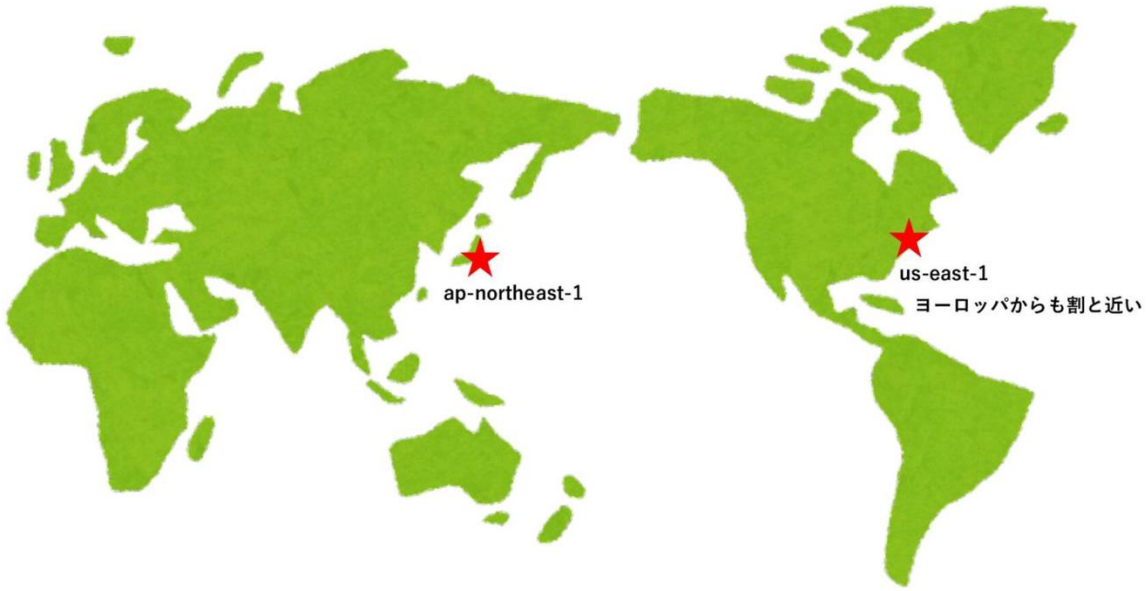
Terraform実行環境の変遷

- 2018年(導入時): GitHubとCircleCIの連携（IAMユーザーアクセスキー）
- 2018年後半: GitHubとAWS CodeBuildの連携（IAMロール）
- **EKS移行後～現在: GitHub Actions → 自作ツール → AWS Lambda → AWS CodeBuild**
  - セキュアであることと使い勝手の良さを考えた結果の構成
    - Slackとの連携によって多くの開発者が手軽に利用できる
  - 一部のサービスではTerraform Cloudを利用

マルチリージョン構成

マルチリージョン構成

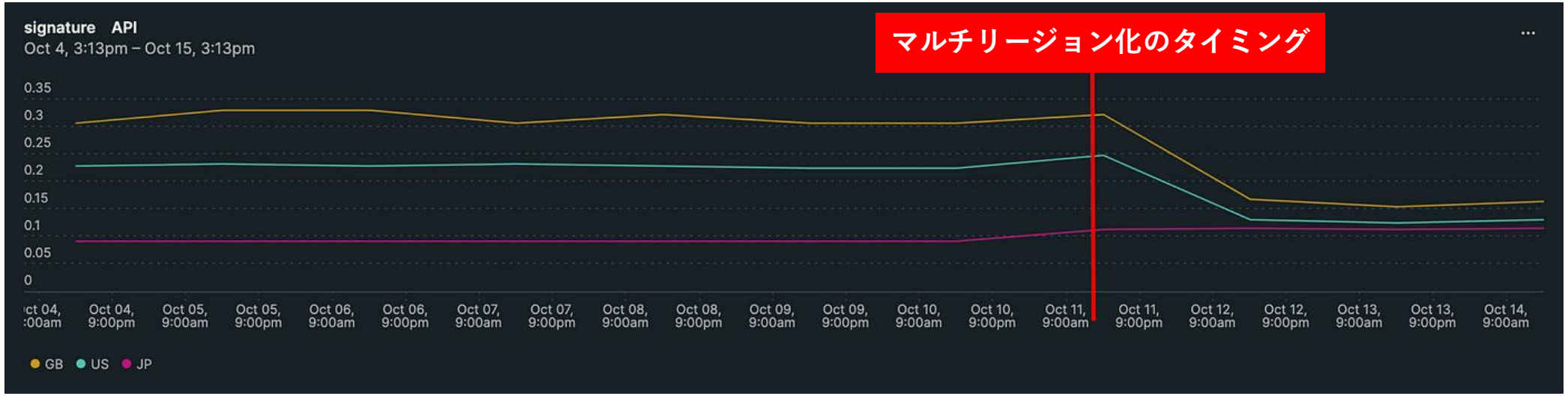
- リリースから7年くらいは**東京リージョン（ap-northeast-1）のみ**を利用してきた
- 海外ユーザーの利用体験をもっと良くしたかった（**ディザスタリカバリの観点ではない**）
- しかしどのくらい良くないのかわからないという課題があった
  - New Relic Mobileを導入することで各国・地域のユーザーのレスポンスタイムを計測できた
- **バージニア北部リージョン（us-east-1）を追加**することでレスポンスタイムを改善したい
  - USからもヨーロッパ、EU各国からも地理的に有利（少なくとも東京よりも近い）



マルチリージョン構成

マルチリージョンによる効果

アクセス頻度の高いAPIをユーザーから近いリージョン（東京orバージニア北部）から提供することでレスポンスタイムを大きく改善することができている



アメリカは日本とそこまで差のない速度まで改善できた！

ヨーロッパ（イギリス）の速度も2倍程度まで速くすることができた！

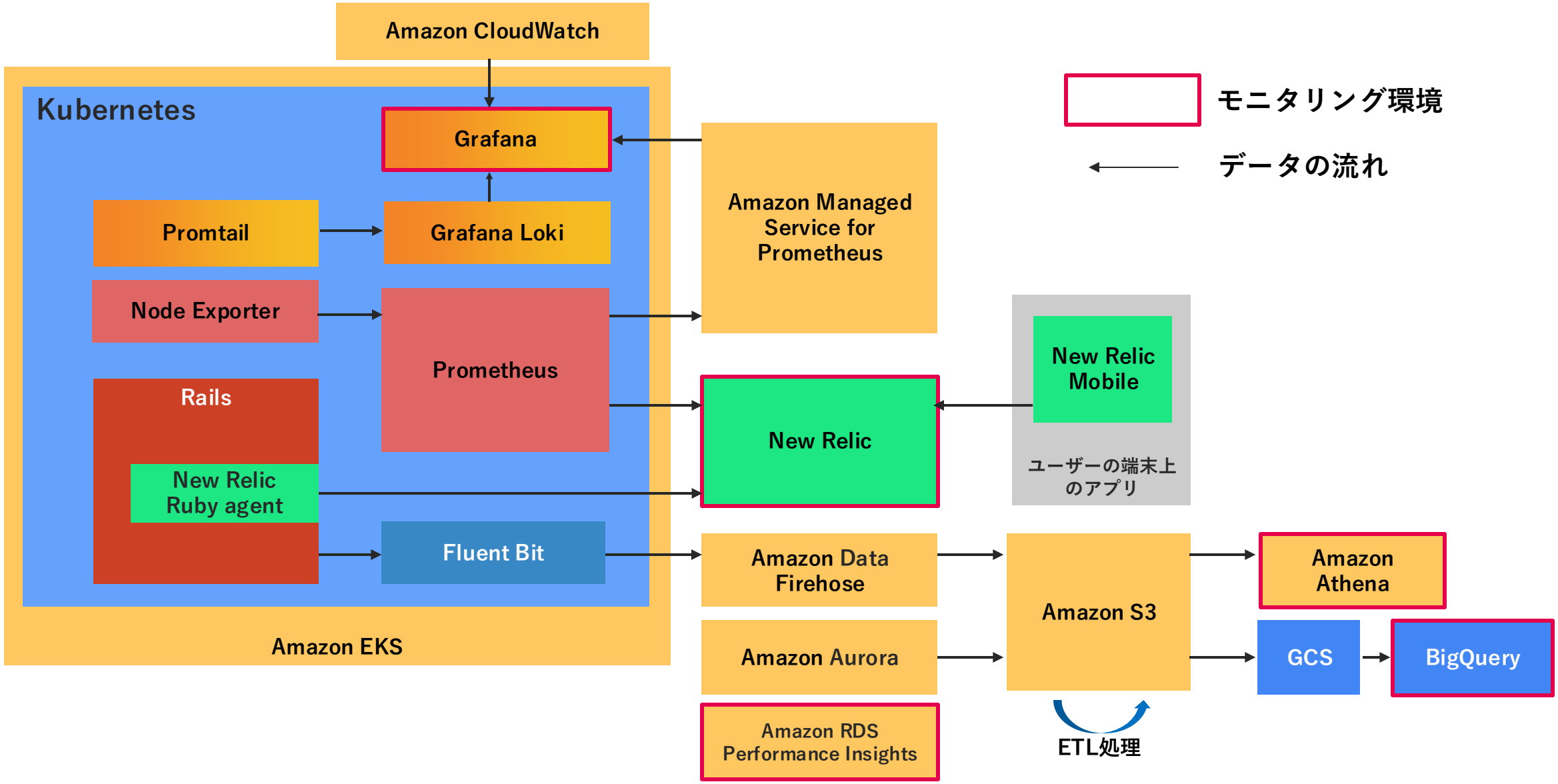
（参考）家族アルバム みてね のインフラをマルチリージョン化しました
https://team-blog.mitene.us/mitene-infra-multi-region-614717f0162d

[この章のまとめ] インフラの変遷の経験からの学び

- Kubernetesに移行し、マルチリージョンにしたことでサービスの信頼性が高まった
- ただし、Kubernetesの利用は必ずしも正解とは限らない（銀の弾丸ではない）
  - サービスと組織の規模、チームメンバーのスキルなどを踏まえて検討することが大事
  - Kubernetesの定期アップグレードの運用負荷はツールや仕組みによってある程度緩和できるが、決して軽いものではない（クラウド各社のマネージドサービスによって運用負荷の違いはある）
  - 今ではKubernetesにまつわる知見がかなり豊富になったので学びやすさはある（ブログ、書籍、勉強会）
  - サービス初期やスタートアップにおいてはAWSであればAmazon ECSも良い選択肢
- CI/CDやIaCは様々な選択肢がある
  - OSSを選ぶ場合は、要件に合っているか、開発が活発であるか（機能開発、バグフィックス、脆弱性の対応など）の観点で選ぶのがおすすめ（GitHubのスター数も一つの目安）
  - IaCはコードの書きやすさ/読みやすさだけでなく、実行環境の運用のしやすさ、セキュリティも大事
- クラウドのリージョンの選び方
  - 最初からサービスをグローバルで展開する場合は北米（とくに東海岸）を中心にするのも1つの選択肢（US、EUからの距離的に有利であるため）。アジア中心であれば東京リージョンで十分。

4 オブザーバビリティの事例

オブザーバビリティの構成



AWSにおけるオブザーバビリティ

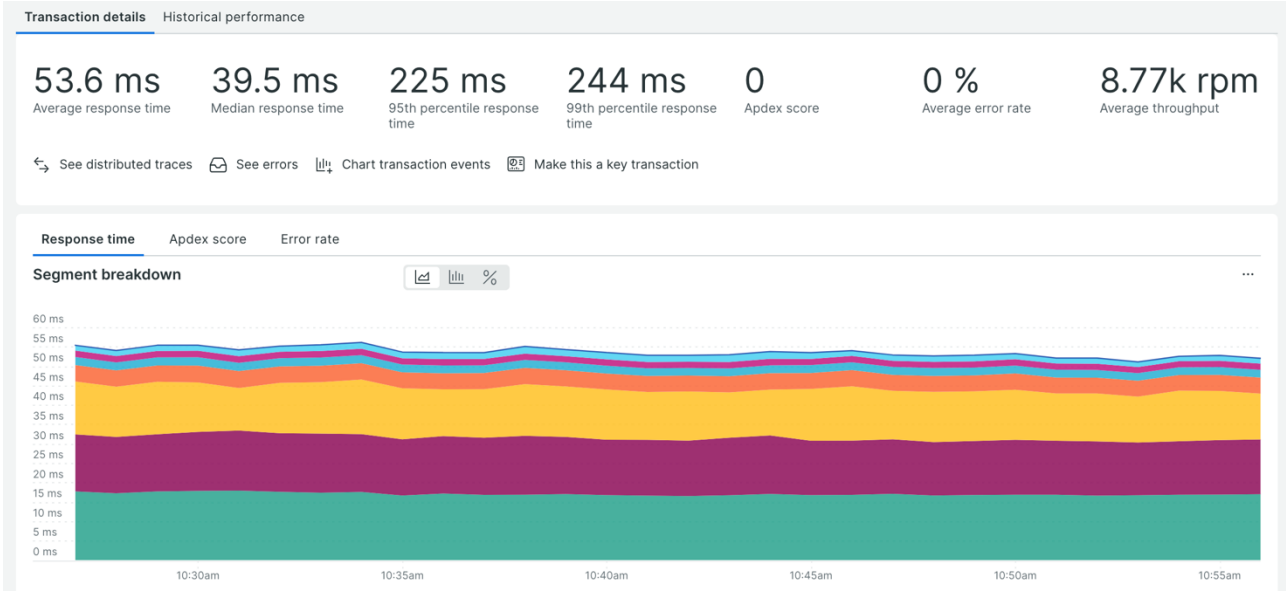
Amazon CloudWatchをどう使っているか

- AWSの各種サービスのメトリクスの確認
  - ALBのリクエスト数、処理バイト数、エラー数
  - AuroraのCPU負荷、メモリ、接続数、バッファキャッシュヒット率、レイテンシー、IOPSなど
  - ElastiCacheのCPU負荷、メモリ、接続数
  - SNSの通知数、失敗数
  - SESの送信数
  - SQSのメッセージ数
  - S3のバケットサイズ、オブジェクト数

New Relic APM / Mobileによるオブザーバビリティ

なぜAPMが便利か

- Webアプリケーションの状態を即時に知ることができる
- 1台ずつサーバーのアプリケーションログをチェックするのは非効率、ミスが起きやすい
- あらゆる処理の性能、エラー状況について誰もが把握できる
- 開発者はデプロイ後に想定外の問題が発生していないかどうか確認できる



[この章のまとめ] オブザーバビリティの経験からの学び

- みてねではコストを最適化するためにAPM、インフラのメトリクス、ログのツールを分けている
  - KubernetesではPrometheusとの相性がよいが、マネージドサービス（Amazon Managed Service for Prometheus）の併用によってさらに運用しやすく
  - ログはS3を基本についてコストを最適化（Grafana LokiのストレージにもS3を利用）
- オブザーバビリティを実現するSaaSは高機能で便利なものが多いが、サービスの規模によってコストが大きくなりやすいので要注意
  - SaaSによって、ホスト数、ユーザー数、データ量など課金のされ方は様々
- 監視やログの確認のために繰り返される手作業が多い場合は、信頼性に影響を及ぼす可能性がある
  - 問題の特定と対策までの時間がかかりサービスに影響がある場合は、SaaSやツールの活用を検討する
  - APMは利用している言語やフレームワークによって対応状況や実装の方法が異なるので要注意

5 SREチームの日々の運用

毎日ダッシュボードをチームメンバーで確認する

- チームの朝会でGrafanaダッシュボードをチームメンバー全員で眺める
  - ダッシュボードはメンバー誰でもアクセス可能
- **細かく見るのではなくトレンドの変化に注目**
- **あまり長い時間はかけない（だいたい3〜5分くらい）**
- グラフに想定外の動きがあれば調査、対策して信頼性向上につなげる
- 重要なメトリクスの意味や数値が表すことについて共通理解を得られる効果も

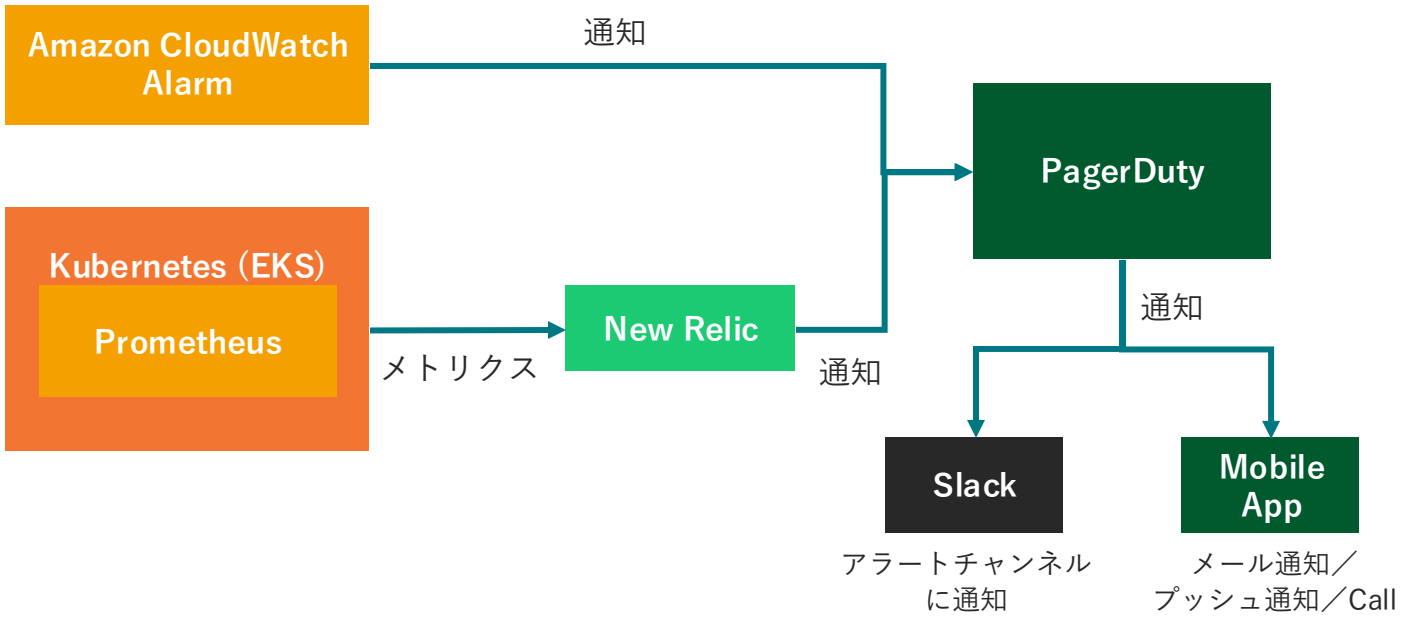
実際にSREチームが毎朝見ている
ダッシュボード（ぼかしています）

データベース負荷の監視

- データベースの問題はサービスに大きな影響を与えてしまう可能性がある
- データベース負荷のオブザーバビリティはとても重要
- Amazon Aurora for MySQLの負荷に関わるメトリクス
  - Grafana経由でのCloudWatchメトリクス
    - CPU利用率、AAS(Average active sessions)、QPS、IOPS、レイテンシ、レプリカラグ、メモリ、バッファキャッシュヒット率、RollbackSegmentHistoryListLength、AuroraSlowHandshakeCountなど
  - Amazon RDS Performance Insights
    - AAS(Average active sessions)
    - WriterのCOMMIT量とその内訳
  - New Relic APMのSlow Query

アラートへの対応

- 営業時間内に発生したアラートはチーム全員で対応する
- アラートはPagerDutyとSlackへ通知される
- 既知のものはランブックに従って対応、未知のものは調査から
- ユーザーに影響があるものは速やかに全体へ周知
  - 周知内容：発生時刻、影響範囲、起きていること、対応状況など



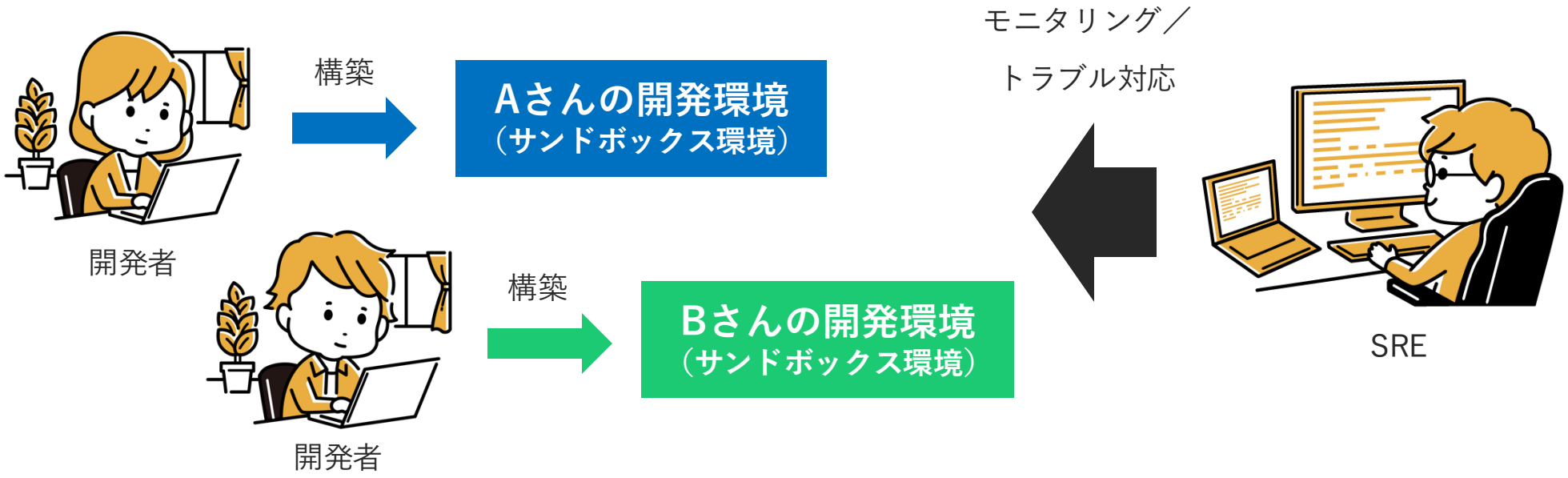
PagerDuty（インシデントマネジメント）の活用

オンコール体制と当番制度

- 1週間交代制のオンコール当番制度（週ごとに1人体制）
  - 営業時間外（19:00-10:00、土日祝日）
  - 当番の手当あり
- エスカレーションポリシー
  - 1: 当番の1人
  - 2: 当番以外のチームメンバー
  - 3: マネージャー
- インテグレーション（PagerDutyの連携元）
  - New Relic
  - Amazon CloudWatch Alarm

開発環境（サンドボックス環境、CI/CD環境）の整備・サポート

- みてねでは開発者一人ひとりに専用の開発環境（サンドボックス）が提供されている
- 開発者ごとにYAMLファイル（SSH公開鍵を定義）を書いてGitHubにプッシュするだけでサンドボックス環境が構築される
- すべての開発者が快適に開発できるようにSREがさまざまなサポート
  - CPU、メモリリソースの調整
  - ツールとドキュメントの整備
  - トラブルシューティング対応



開発者からの各種相談対応

- AWS、各種インフラ相談、質問
- 開発環境、CI環境のトラブルシューティング
- Terraformのコードレビュー依頼

相談の例をいくつか

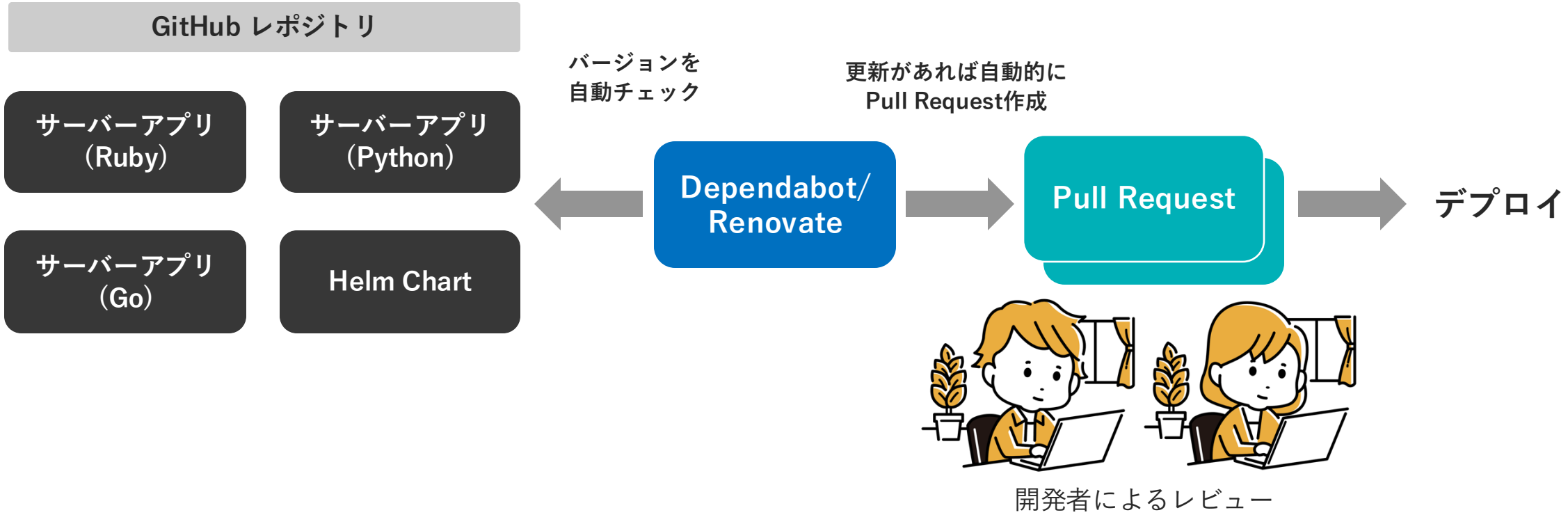
Rails.logger.error で記録されてるログを見たいんですが、 Athena で見ることはできますか？

S3の画像を更新したのですが、CloudFrontのキャッシュが効いているようで古い画像が表示されてしまいます。
バケットは ◯ ◯ なのですが、 CloudFront 側のどのキャッシュをクリアすればよいのかわかりますか？

◯ ◯の問題で、この差分を試したいと思います。レビューをお願いできますか？
https://github.com/.../terraform/pull/...

脆弱性対応

- DependabotやRenovateによるライブラリのアップデート
  - 例えばRuby Gem、Go パッケージ、Python ライブラリ、Helm Chartなど
- 定期的に自動的にPull Requestが作成される
- 毎週の定例で担当者をアサインしてPull Requestを確認、マージ



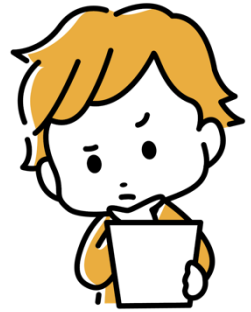
[この章のまとめ] 日々の運用からの学び

- グラフの見方やメトリクスが意味するものはチーム全体で理解をしておく
- データベースのオブザーバビリティは重要だが、メトリクスを理解するためにはデータベースのエンジンの特徴や仕組みについて深い理解が求められることがある
  - 例えば、MySQLであればInnoDBに関わるメトリクス
  - メトリクスからクエリの改善が必要な場合、InnoDBのインデックスや実行計画などの知識が求められる
- 運用系の相談や依頼はなるべく自己解決できるように整備していくことが大事
- アラートの発報は最小限にすることを心がける
  - 対応の必要の必要があるもの、対応の緊急性があるものに対してアラートする
  - 深夜早朝のバッチ処理は本当にその時間帯に実行しなければいけないかどうか
    - バッチ処理を起因に発生しうるアラートはオンコール当番の負荷がかなり高いため
  - オンコール当番の負荷は常に大きいと考え、アラートの再発防止を優先する（チームで合意する）
- ソフトウェアの脆弱性の対応は欠かせないものだが、古いバージョンのサポートがなくなるリスクもあるため、いずれにしろライブラリはアップデートし続けなければならない

6 ポストモーテム運用

ポストモーテム運用

- ポストモーテム（Postmortem）はいわゆる障害報告書
  - オライリージャパンの「SRE サイトリライアビリティエンジニアリング ―Googleの信頼性を支えるエンジニアリングチーム」15章「ポストモーテムの文化：失敗からの学び」で取り上げられている
- みてねではユーザーになんらか影響のあった際にポストモーテムを作成することが多い
  - ユーザー影響がない場合でも作られることはある（なんらかの失敗やミスなど）
- みてねでの実績としては年間30-40件ほど書かれている
- Notionにポストモーテムのデータベースがあり、テンプレートに従って誰でも作成可能



ポストモーテムの構成（テンプレートの構成）については

次のスライドで解説します!

ポストモーテムの構成①

- **タイトル**
- **サマリ**
  - どういう問題が発生したのかを簡略に
- **影響**
  - ユーザー影響・収益への影響・CSへの影響・etc...
- **発生要因**
  - 問題が発生するきっかけとなった要因
- **根本原因**
  - 問題が発生しうる状態になった原因
- **検知**
  - 問題が発生したことに気付いた経緯
- **暫定対応**
  - 発生した問題に対する暫定対応

ポストモーテムの構成②

■ 再発防止策

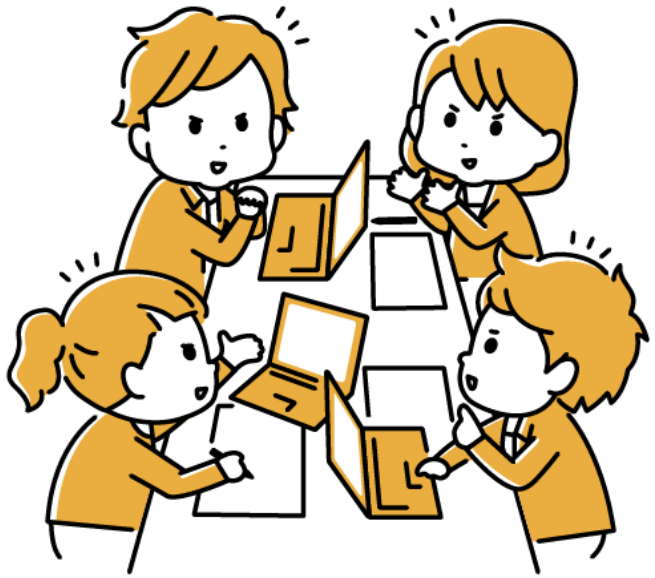
- ■ 予防（障害の再発をポジティブに防ぐにはどうしたらよいか）
- ■ 検出（同様の障害を正確に検出するまでの時間を減らすにはどうするべきか）
- ■ 緩和（次回この種の障害が起きたときの深刻度や影響度の％を減らすにはどうしたらいいか）
- ■ 修正（次回障害が検出されたときにどうすればより速く回復できるか）

人間は「修正」できない。環境を修正しなければならない。

■ 教訓

- ■ うまくいったこと
- ■ うまくいかなかったこと
- ■ 幸運だったこと
- ■ 所感

このあたりは必須項目ではなく、もしあれば程度



ポストモーテムの構成③

■ タイムライン

- 状態: 発生・発見・対応・復旧
- 時間
- 対応者
- 事象
- 備考

タイムラインの例

状態	時間	対応者	事象	備考
発生	11:20	@isao.shimizu	手順に従って◯◯を実行	SlackのURL
発見	11:45	@isao.shimizu	エラーが出ていることに気づく	SlackのURL
対応	11:49	@isao.shimizu	◯◯の停止を実行	SlackのURL
復旧	11:51		◯◯が停止されエラーがおさまった	SlackのURL

[この章のまとめ] ポストモーテム運用からの学び

- ポストモーテムは常に作りやすく、漏れなく記述できるようにテンプレートを活用する
  - テンプレートは作って終わりでなく改善を続ける
  - 組織のすべてのメンバーが閲覧・編集できるようにする
- 障害対応に限らず、失敗の記録と再発防止に役立つ
  - エンジニアに限らず組織全体で活用でき学びを得ることができる
- **誰かを非難するものであってはならない**
- 再発防止は人間の能力に頼ったものではなく、仕組みで防止する
  - 人間はミスをするものという大前提がある
- ポストモーテムは作って終わりではなく、様々なステークホルダーに共有する
  - プロダクトオーナーやビジネスサイドのメンバーにも知ってもらう

7 現在の運用課題

現在の運用課題

- 自動化しているところもあるがまだまだ手動なところはある
- Kubernetes（EKS）のアップデートはもっと楽にしたい、簡素にしたい（AWSにも期待したい）
- もっと効率化したい、開発生産性を高めたい
  - インシデントマネジメント、ポストモーテムの作成
    - 例えば生成AI（LLM）やPagerDutyをもっと活用するなど
  - CI/CD
    - GitHub Actions、CircleCIのチューニング
    - テストの高速化
    - イメージビルドの高速化
    - コストの最適化
- 開発環境におけるトラブルを減らしたい（開発者からの相談がそこそこある）
- IaCの適用範囲を広げたい
  - 例えば、Grafana、PagerDutyなどの手作業を減らしたい、Pull Requestベースで変更記録を残したい

8 まとめ

まとめ

- 家族アルバム みてねにおける運用事例をご紹介しました。なんらか参考になれば幸いです。
- プロダクトや組織のフェーズによって最適な運用手段は変わってきます。
- ハードウェアやソフトウェアの進化によって、数年後には全く変わった運用になっているかもしれません。

過去の技術記事、採用情報など

みてね チームブログ	https://team-blog.mitene.us/
みてね×gihyo.jpスペシャル	https://gihyo.jp/list/group/みてね×gihyo.jpスペシャル
みてね 採用情報	https://team.mitene.us/
MIXI GROUP新卒・中途採用情報	https://mixigroup-recruit.mixi.co.jp/
MIXI ENGINEERS Xアカウント	https://x.com/mixi_engineers

心もつなごう。

MIXI