Amazon ECSを活用した、転職サービス「ミイダス」のAWS 移行事例

パーソルキャリア株式会社
ミイダスカンパニー

吉元 裕人

吉元 裕人(よしもと　ひろひと)

- パーソルキャリア株式会社
  ミイダスカンパニー
- Backend Engineer
  Infrastructure / DevOps Engineer
- 2015年から現職へ転職
  転職サービス『ミイダス』にローンチから携わる
- Golang、Docker💖

- **AWS Summit 2017では、弊社HITO-Managerチームが登壇しました📣**

  cf. https://d1.awsstatic.com/events/jp/2017/summit/slide/D3T5-6.pdf

ミイダス

2018年1月末、
ミイダスをあるクラウドサービスの仮想マシン群からAWSに移行しました🎉

プロジェクトの進め方や事例を紹介します
導入や移行の参考にしていただけたら幸いです

ミィダス

アジェンダ

- なぜ移行したか
- 検証／設計
- 構築
- リリース
- 運用
- まとめ

ミイダスって……

この発表より前に聞いたことあります？

ミイダスとは

- 2015年11月オープンの転職支援サービス

- 17万人以上の転職者ユーザと1万社以上の企業をマッチング(2018年5月現在)

ミイダスとは

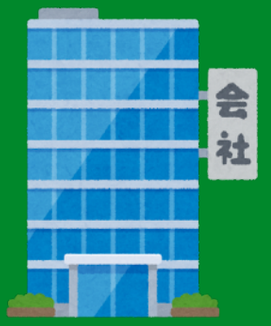
ミイダス

求職者

登録
希望条件登録
想定オファー年収算出
トーク
オファー閲覧
適性チェック

企業

登録
ユーザ検索
オファー送信
トーク



求職者のメリット

- オファー実績から、登録前にどのようなオファーが届くかわかる
- 書類選考通過済みのオファーのみが届く
- 登録したらすぐにオファーが届く
- 設定した希望条件で自動選別

企業のメリット

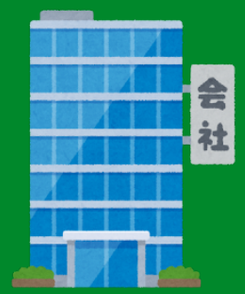
- 細かい条件で候補者を検索できる
- 高速に検索できる
- 採用にかける労力も費用も低コスト

サービス全体像

求職者

WebUI

企業



WebUI

サービス運用者

WebUI

共通API

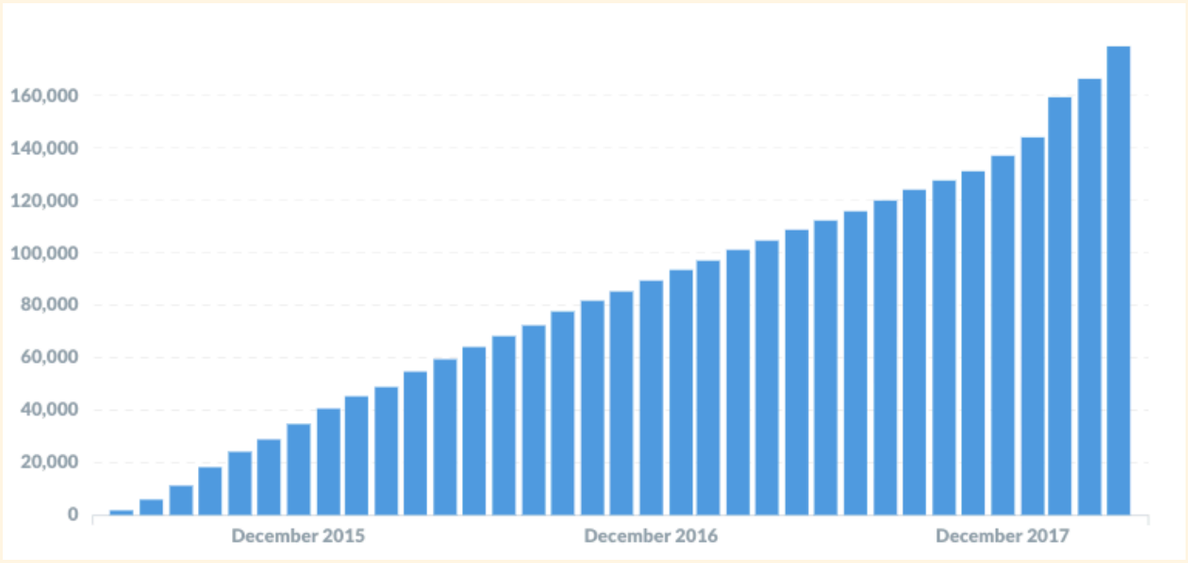
バッチ

イラスト: いらすとや

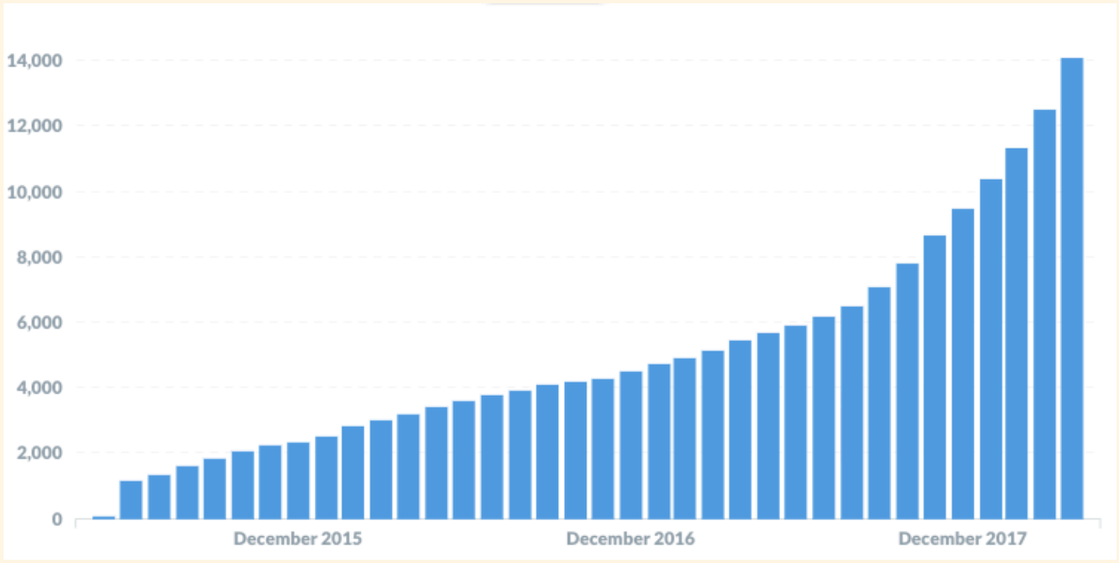
https://www.irasutoya.com/

成長スピード

ユーザ数



企業数



- ピーク時アクセス: 200 req/s
- 2年で企業数3倍、ユーザ28倍の成長スピード☺️
- ユーザと企業間に蓄積されるデータも<指数関数的>に増加😨

そもそも

- 2014年社内の事業創出制度からプロジェクトスタート
- スピード重視での実装
- 運用費用もローコストが求められていた
- アプリケーションエンジニアがインフラも担当

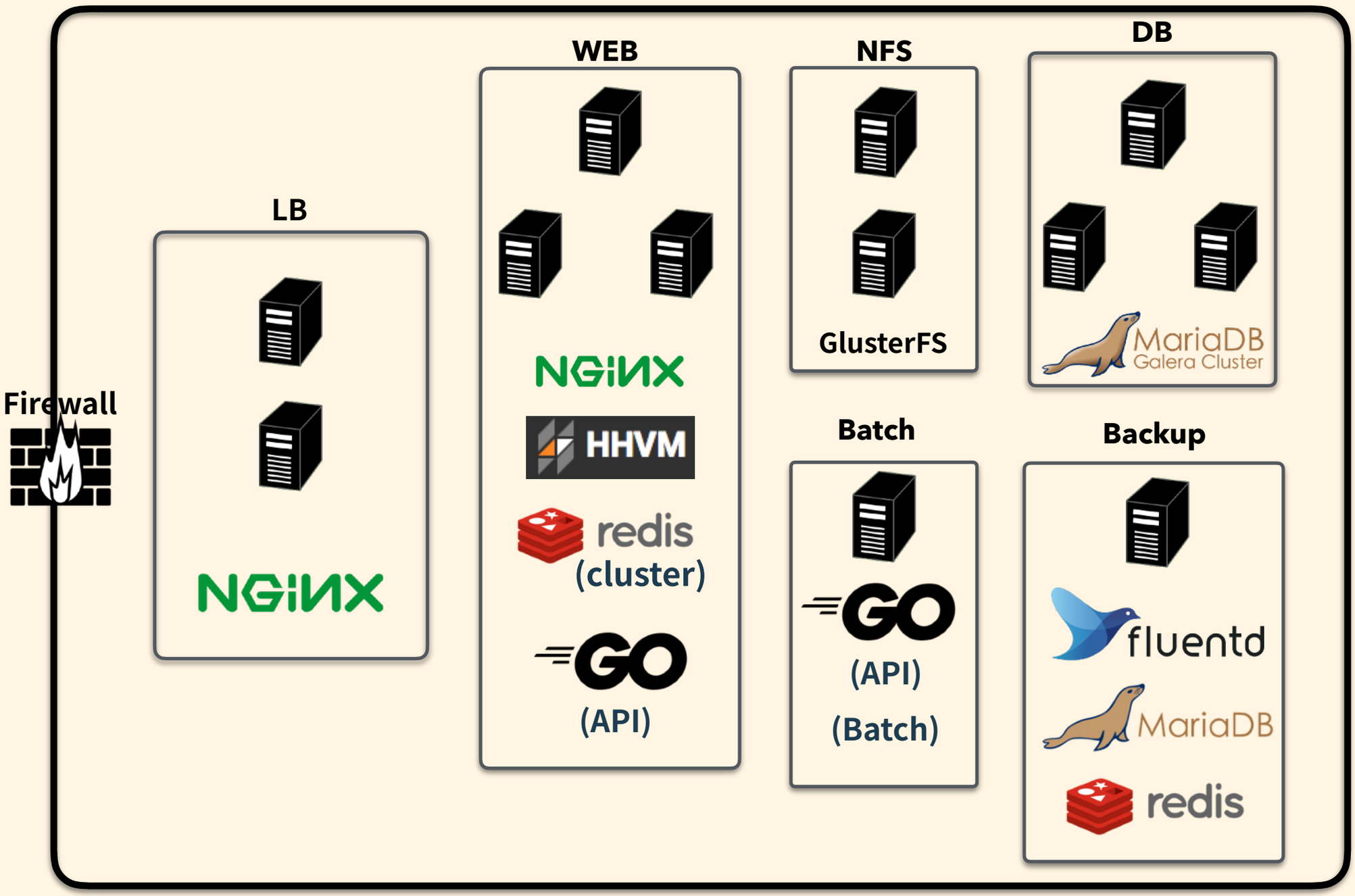
何が起こるか？

- サービス成長による変化 (嬉しい悲鳴)
  - データ量が増える
  - バックアップデータも増える
  - バッチ実行時間も増える
  - CPU/メモリ利用率も上がる

何が起こるか？

- データ・ログ・バックアップデータの増加
  - 毎月いくつかのサーバのディスク容量拡張
    - ➡ 面倒な手動オペレーション🔧
    - ➡ クリティカルなサーバを扱うときはサービスメンテが必要🚧
- サーバのスケールアップ/アウトの発生
  - ➡ スケールアップと共に設定変更が必要
  - ➡ スケールアウトしたサーバのクラスタノードのせいで全体が不安定に
- [歴史的な理由]でサーバの構築が複雑に

元の構成



Monitoring

ZABBIX

Deployment

Backup

この構成の不満

- サーバ上が整理されていない
  - 同じ機能が別のサーバでも動いている
    - イメージバックアップの弊害
  - ミドルウェアが乱立
  - チューニングしづらい
- バックアップサーバに処理が集中
- ほぼ単一のサブネット上に配置

ミィダス

⚠️積もる不満⚠️

課題感

- 今後の成長に備えた構成にしたい
    - ユーザの拡大(求職者/企業)
    - メディアに取り上げられる際のスパイクを防ぐ
        - ➡ スケールアップ/アウトを手軽に行いたい
- 手のかかる作業をやめて開発に集中したい
    - マネージメントサービスを活用したい
    - 突発作業でもオペレーションミスが起きないものにしたい
- 構成と構築をシンプルにしたい

課題を解決するために、移行を提案

移行プロジェクト

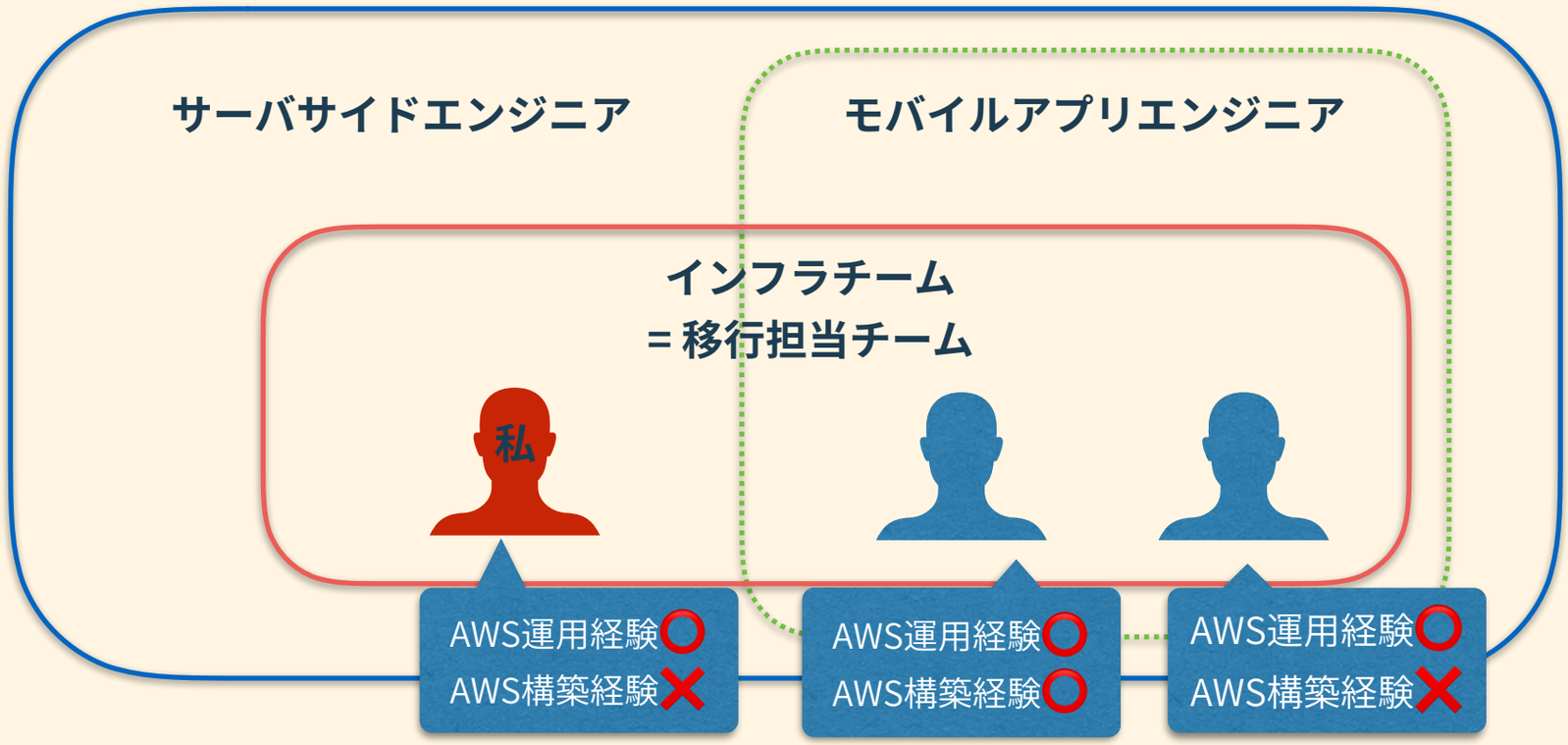
- 2017年8月にスタート

検証／設計	構築	リリース
• チーム組成 • 移行先の選定 • 構成の方針策定 • 検証 • 設計 • 設計レビュー	• 環境を1セット構築 • 動作確認テストケース作成 • 動作確認 • 開発／ステージング環境の作成 • 各環境の動作確認	• リリース手順作成 • リハーサル • メンテナンス告知(社内外) • リリース作業

ミイダス

検証／設計

チーム組成



- 移行担当チーム：インフラエンジニア3人
  - 通常は他の役割と兼任
    - 移行期間は主担当(私)はほぼインフラ業務に専念
    - チームメンバーは、随時レビューしつつ、構築フェーズから合流

クラウドベンダーの選定

- AWSとGCPで比較
  - 費用の試算

    ➡ カタログスペック対費用はあまり差がない
  - 利用したいマネージドサービスの多さでAWSに軍配
- 既にAWSを一部利用していた
  - CloudSearch, API Gateway, Lambdaの利用中
- 社内基幹システムのAWS移行
  - ビジネスサポートを契約済み/他部署の利用

総合的に評価し
AWSを利用することに

設計の方針

- 既存のアプリケーションにはなるべく手を入れない
  - 開発のスピードを落とさないことが目的
- 役割を整理してシンプルな構成にする
- 可能な限りマネージドサービスを利用
- 運用費用の削減が目的ではない
  - インフラコストに見合うエンジニア工数の削減が目的

検証&設計

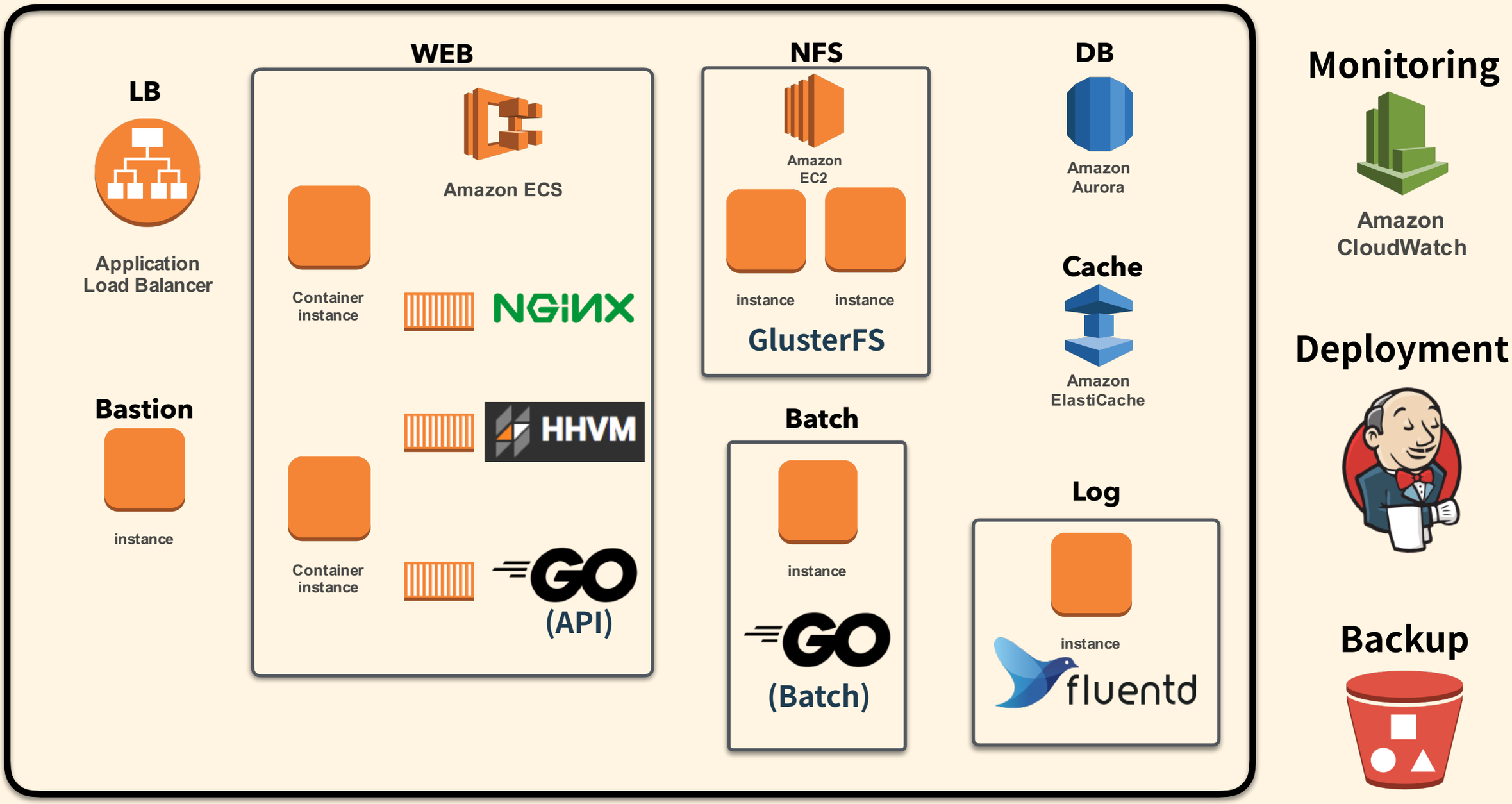
- 環境を作って、懸念点を潰していった
    - 使い勝手
    - 既存アプリケーションとの互換性
    - セキュリティ
- 構成がまとまってから、資料を作成して担当のAWSソリューションアーキテクトの方にレビューいただいた
- フィードバックをもとにさらに設計を修正
    - 全社でEnterpriseサポートを契約

AWS Loft Tokyo🎉

10月
オープン予定

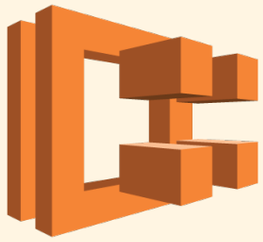
cf. https://aws.amazon.com/jp/blogs/startup/announcing_loft_tokyo/

今の構成


LB
Application Load Balancer
Bastion
instance
WEB
Amazon ECS
Container instance
NGINX
HHVM
Container instance
GO
(API)
NFS
Amazon EC2
instance
instance
GlusterFS
Batch
instance
GO
(Batch)
DB
Amazon Aurora
Cache
Amazon ElastiCache
Log
instance
fluentd
Monitoring
Amazon CloudWatch
Deployment
Backup

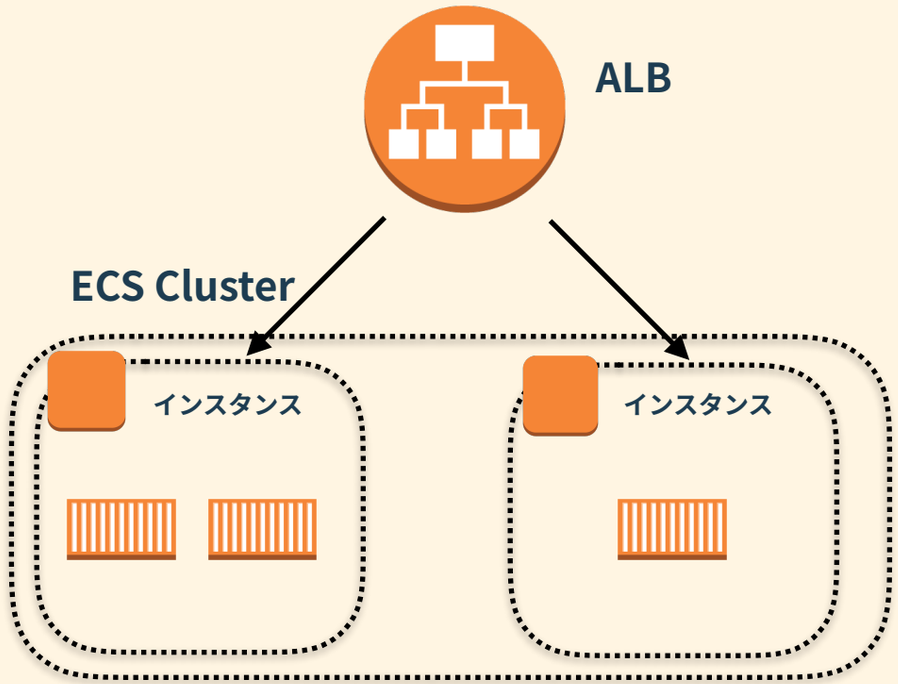
機能	移行前	移行後	マネージド	変更
Load Balancer	NGINX	Application Load Balancer	○	○
WEB Server	Server	Amazon ECS	▲	○
DB	MariaDB Galera Cluster	Amazon Aurora	○	○
DNS	-	Amazon Route53	○	○
In-Memory Cache	Redis Cluster	Amazon ElastiCache Redis	○	○
Monitoring	Zabbix	Amazon Cloudwatch Amazon SNS	○	○
Log Server	Server	Amazon EC2 instance Cloudwatch Logs	▲	○
Batch Server	Server	Amazon EC2 instance	×	×
NFS Server	Server GlusterFS	Amazon EC2 instance GlusterFS	×	×

Amazon ECS
(Elastic Container Service)



Amazon ECS

コンテナのデプロイ
管理・監視
オートスケール
ターゲットグループへの追加



- あらかじめ設定したECSクラスタ(もしくはFargate)に対して、
  タスクを実行させたり、実行状況を管理することができるコントロールプレーン
- コンテナが動く実際のリソースはEC2
- コンテナのネットワークやマウント領域などを指定する「タスク」を設定
- ECSクラスタに「タスク」をもとにデプロイや起動数、オートスケールやターゲットグループへの紐付けを指定できる、「サービス」を設定
- ECSクラスタを作成するとCloudFormationスタックが立ち上がる

EC2 vs ECS

取扱	EC2/仮想マシン	ECS
ミドルウェアの管理	Ansible	Docker Image
ソースデプロイ	rsync	Docker Image
バックアップ単位	AMI	Docker Image
オートスケール	インスタンスが増減	ECSタスクが増減 コンテナインスタンスが増減

- **最初はEC2でやるつもりだった**
  - **構成がシンプルになる**
  - **依存関係が明確になる**
  - **各所でメリットが多いと判断、利用することに**

ECSへの変更

- 良かったこと
  - そもそも開発端末で動く環境としてDockerを利用していたため、設定が流用できた
  - ローカルの開発環境から本番環境まで、動作環境の統一
    - インフラエンジニア以外もミドルウェアメンテナンスに協力しやすい
- クセのあるミドルウェアでも、コンテナで導入しやすい
  - Debian/Ubuntuしか公式サポートされていないHHVM
- ヘルスチェックによってコンテナが再起動して復帰する。プロセス監視が不要

ECS実際に運用してみて

- クラスタ作って、動くようにするまではコストかかる
  - CloudFormationに対する慣れ
    - ECSクラスタへの起動設定やAuto Scalingグループへの設定が、CloudFormationで上書きされる
  - 設定箇所が多くてつらい
    - ECS-CLIでdocker-compose.ymlを用いてタスクの作成を行える
      cf. https://docs.aws.amazon.com/ja_jp/AmazonECS/latest/developerguide/cmd-ecs-cli-compose.html
      - ECSへのロックインを防ぐメリットも

ECSを実際に運用してみて

- デプロイ手順が増え、遅く感じる
  - 以前：
    - 単純にrsyncでのソースデプロイ
  - ECS：
    - Dockerイメージビルド
    - ECR(Elastic Container Register)にプッシュ
    - プッシュしたイメージでタスク作成
    - タスクをサービスに紐づけ
    - サービスがコンテナを切り替える
    - コンテナの切り替えが終わる

ECSを実際に運用してみて

- 仮想マシンで管理してた頃より、管理も楽になった😀

構成を変更しなかった点

- Batchサーバ(EC2 Instance + Go Binary)
    - 一部バッチにサーバにステートを持って連携をする処理があったため
- 代替案を検討中
    - AWS Batch
    - ECS スケジュールタスク
    - SQS+Lambda

構成を変更しなかった点

- NFSサーバ(EC2 Instance + GlusterFS)
  - S3+Goofysを検討した
    - ソリューションアーキテクトの方「おすすめしない」
      - ➡ファイル操作時にS3のHEADやLIST APIを結構叩く
      - ➡(--cheapオプションもあるが速度とのトレードオフ)
- Amazon Elastic File System(EFS)の東京リージョン待ち
  - いくつものEC2から同時にアクセスでき、容量が自動的にスケールするフルマネージドのファイルストレージ

EFS 東京リージョン🎉

cf. https://aws.amazon.com/jp/blogs/news/amazon-elastic-file-system-efs-nrt/

設計で参考にした資料

- AWS クラウドサービス活用資料集
  - 公式に日本語ドキュメントがあって心強い
  - 行き詰ったときなどに読んでいた
    https://aws.amazon.com/jp/aws-jp-introduction/
- Developers.IO
  - 新しいサービスの資料が早くてありがたい
    https://dev.classmethod.jp/

ミィダス

構築

構築

- 2017年10月～12月
- 設計通りに構築して、構築手順を作成した後、テストケースを作成し、その環境の上で正しくアプリケーションが動くか確認した

開発/ステージング環境の構築

- 構築手順と実際に動く環境を参考に、チームメンバーそれぞれが環境を作成し、お互いにレビュー
  - チームメンバーのキャッチアップが目的
  - 必然的に全てのサービスを触ることになり、実践的に仕組みを学べた

ミィダス

リリース

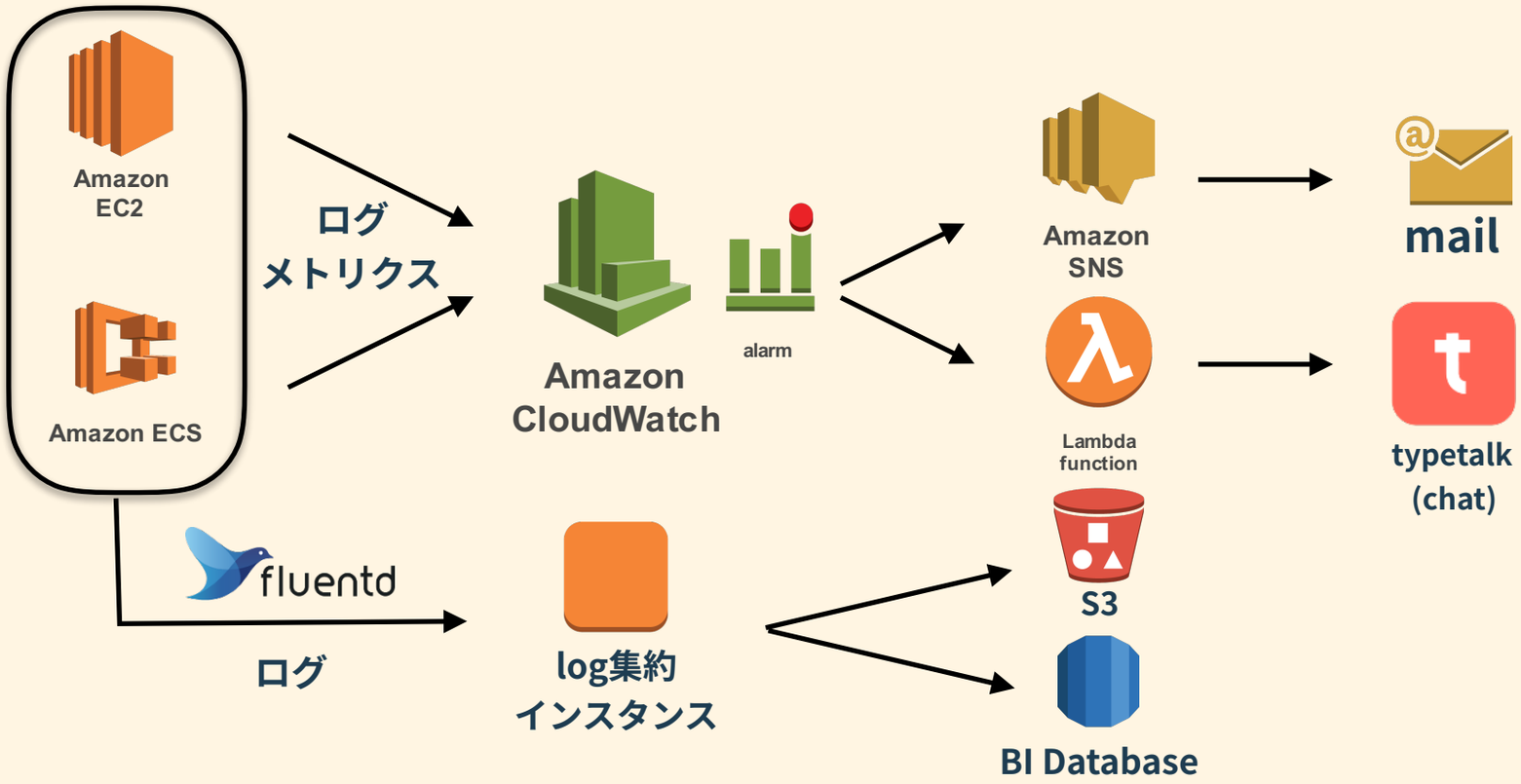
移行作業

- 2018/1下旬
- リハーサルを2回して、手順と時間を把握
- 夜22時～翌17時までサービス停止
- DB/Redisデータダンプ取得&AWSに投入
  - この手順が長く、交代で対応
- ユーザアップロードデータ取得&AWSに投入
- 確認

ミイダス

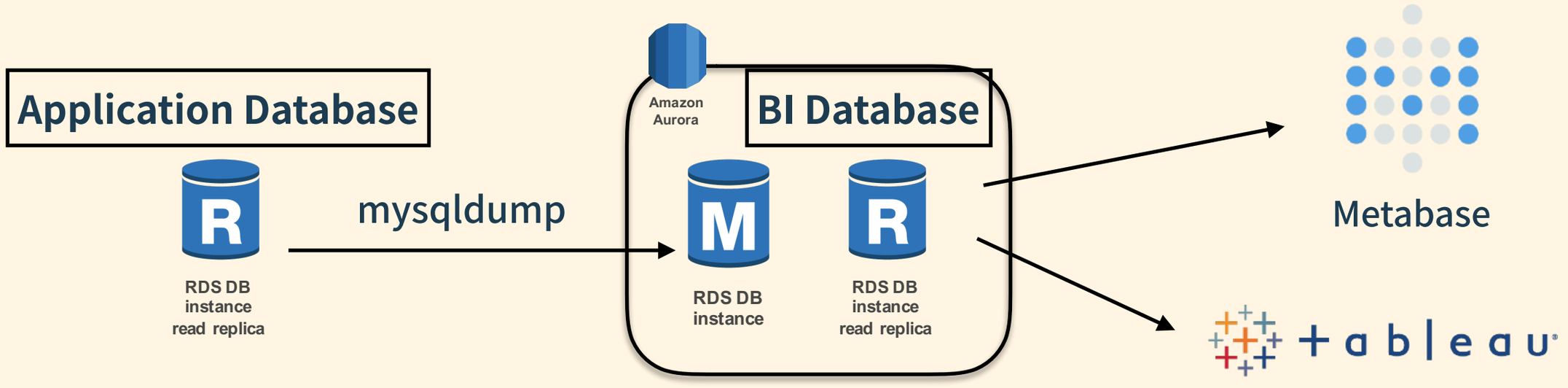
運用

ログ・監視



- ログ：CloudWatch LogsとLogサーバ両方に収集
- メトリクス：CloudWatchに集約
- アラート：alarmを設定し、Amazon SNSとLambdaで通知

BI

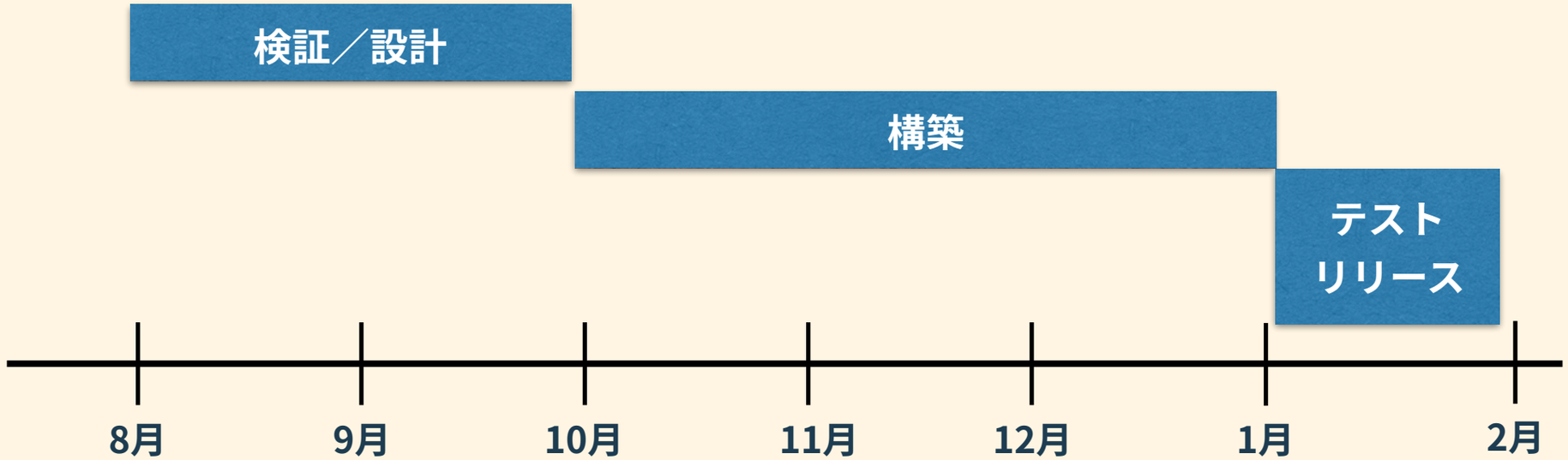


- MetabaseでKPI可視化
- Tableauで分析
- 定期的にメインDBのReaderノードからダンプファイルを出力してBI用DBに投入
- AWS Database Migration Serviceで運用していたが、しばらく粘ったが挫折
  - UTF8MB4に非対応
  - インデックスを作成するタイミング

ミイダス

結果

実際のスケジュール



- 2017/8
  - 検証開始
- 2017/9
  - 設計、サンプル構築
- 2017/10-
  - 構築
- 2017/12
  - 開発環境の作成、チーム内キャッチアップ
- 2018/1
  - テスト
- 2018/1下旬
  - リリース

移行の作業工数

- 検証・設計
  - 1人で4ヶ月
- 構築
  - 3人で2ヶ月
- テスト
  - 2人で1ヶ月

- ざっくり12人月程度

費用

- 移行前より費用は増えた
  - 役割を分割して、フルマネージドに載せた
  - 可用性を考慮し、マルチAZ構成にした部分
- 費用対効果を考えると妥当な金額に

ミィダス

移行して良くなったこと

改善されたところ

- メンテナンスが楽に
  - メンテ対象が減った
    - 16台の仮想サーバ→10台のEC2インスタンス(1環境につき)
    - 減った分はフルマネージドに移行
    - インフラ由来のアラートは明らかに減った
- ECSは管理コストが低い
  - コンテナ内にトラブルがあったら、自動的に立ち上がりなおす
  - ミドルウェアはDockerの管理だけ

課題は解決できた？

- 今後の成長に備えた構成

  ➡ ECSやマネージメントサービスで可能

- 手のかかる作業をやめて開発に集中したい

  ➡ マネージメントサービスの利用

  ➡ オペレーションミスが起きにくい

- 構成と構築をシンプルにしたい

  ➡ 機能ごとにまとまり、シンプルになった

ミイダス

大満足😄

ミィダス

これからも
サービスと共に成長していく
インフラを
AWSで構築していきます！

We Are Hiring!!!

- ミイダスカンパニー
  - インフラエンジニア
    - AWS, Docker, Infrastructure as Code,CI/CD
  - バックエンドエンジニア
    - Go,PHP(Hack)
  - フロントエンドエンジニア
    - JSX, React, Redux, Babel
- More info
  - 📫 > hirohito.yoshimoto@persol.co.jp
  - 🔎 > [エンジニア パーソルキャリア]

ミイダス

ご清聴ありがとうございました