On-Premise to AWS

6年物のレガシーインフラをいかに移行したか？

Who are we ?

Persol Group
Intelligence
Innovation lab.
SEEDS Company

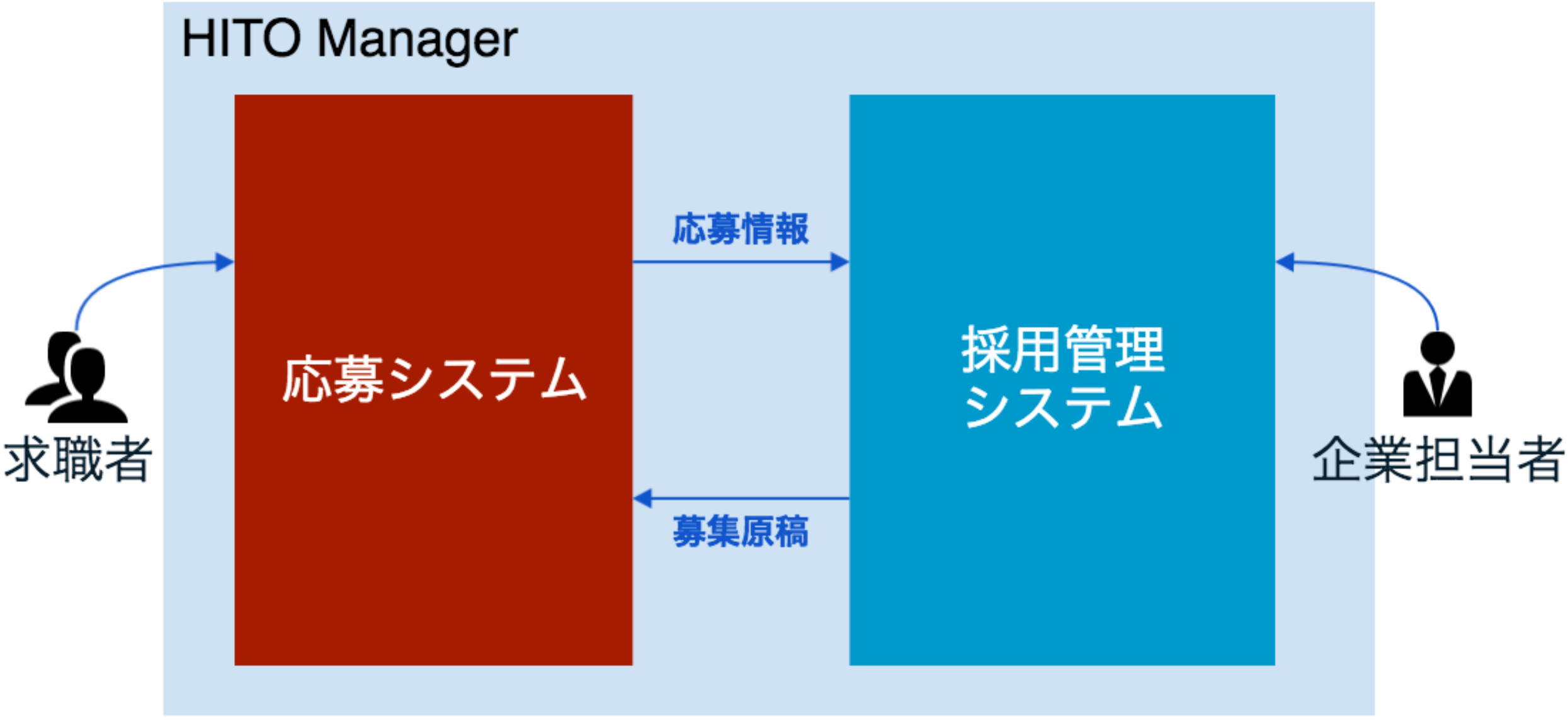
千家 啓陽

keiyo senge

- *DevOps Team*
- *Team Leader*
- *Certified Scrum Master*

HITO Manager

- アルバイト・パート採用支援/管理システム
  (ATS; Applicant Tracking System)
- *since 2010*
- 採用*HP*作成
- 応募者情報管理


HITO Manager
応募システム
応募情報
募集原稿
採用管理
システム
求職者
企業担当者

こんな話をします

- なぜ AWS へ移行したのか？
- どのように移行したのか？
- 実際に移行してどうだったのか？

なぜ AWS に移行したのか？

ビジネス背景

- *HR x Tech* 業界の競争激化
- スピードアップの必要性が高まる

2016年4月以前の課題

- 子会社におまかせ
- レガシーインフラ
- 大量の技術的負債
  - CentOS 5.x の EOL が2017年3月31日

➡ 変化に対応できない

2016年4月 DevOps チーム成立

脱・子会社 ➡ 内製化

・もっと素早いデリバリーを

・もっと高品質なサービスを

・もっと頻繁に価値検証できる環境を

➡ レガシーインフラからの脱却は必然

レガシーインフラ？

- 設定がブラックボックス化
- 構成変更申請が面倒かつ遅い
- トラブルの原因追求が打ち切られる

グループ共通のオンプレ基盤のため、改善が困難

脱・レガシーインフラ

- 設計を透明に
- 構成変更を柔軟かつ高速に
- トラブルの原因調査を詳細に

➡ 自己管理できるAWSへ

Struggle

人的リソース不足

- DevOps チーム4名
- 阻害要因
  - 通常運用
  - 障害対応
  - タスク割り込み
- AWS移行に使えるリソースはせいぜい 30%

どう戦う？

トレードオフによる取捨選択 ➡ 割り込みの極小化

- 定型作業をテクニカルサポートチームへ移譲
- 移行後でも直せるものや改善タスクは後回し
- 組織からの依頼を減らしてもらう
- 週2日 AWS専念デー

Mind Set

- 小さく素早く始める
- レガシー = ブラックボックス = リスク
  - ゼロにはできない
  - 進めながら潰す
- × エンジニアの自己満足
  - ◯ ビジネス要求に対応できる

Timeline

- 2016年11月 移行プロジェクト開始
- 2016年12月 ステージング環境構築
- 2017年2月 第1次移行
- 2017年3月 完全移行完了

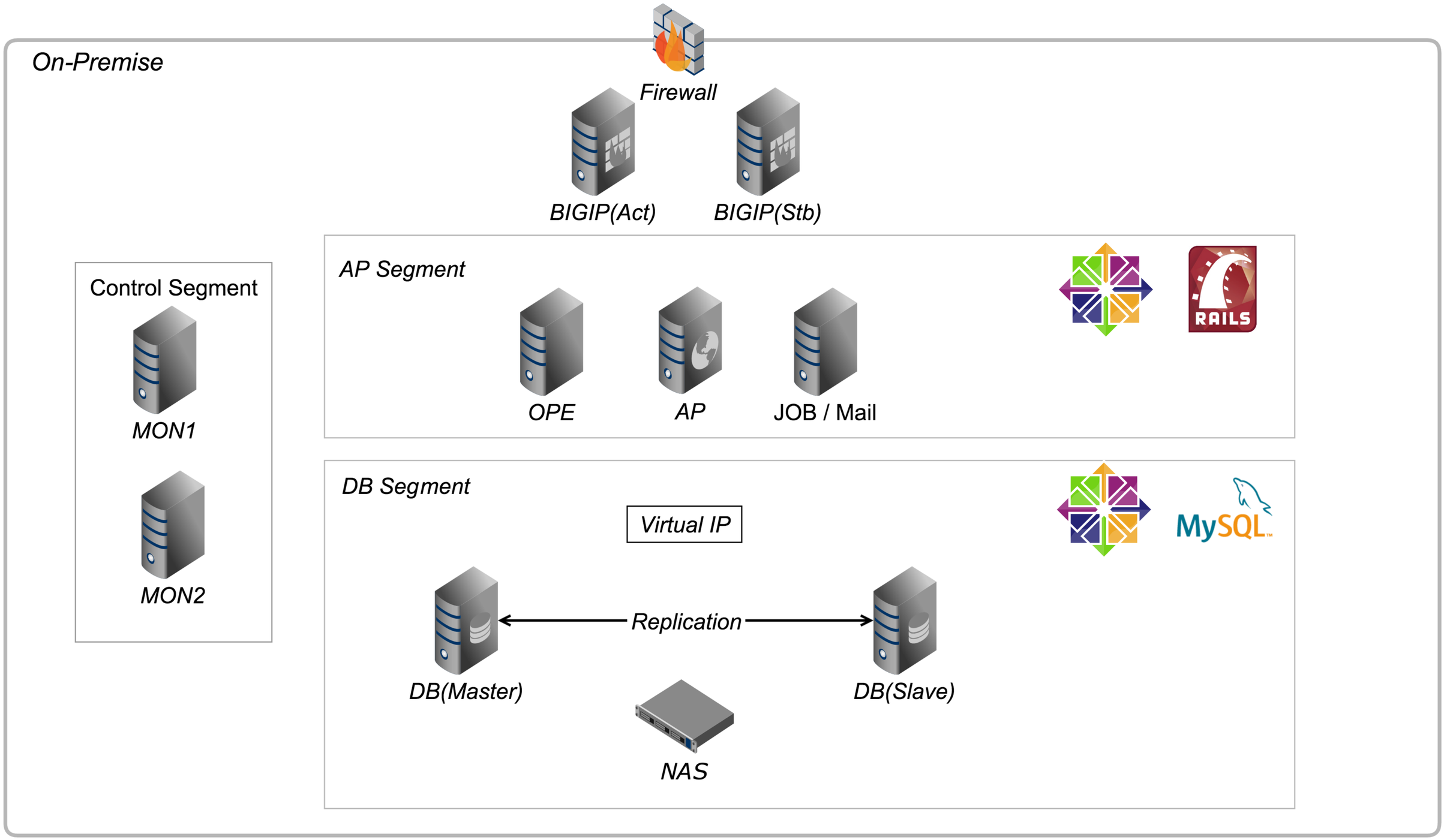
どのように移行したのか？

浅野 潤

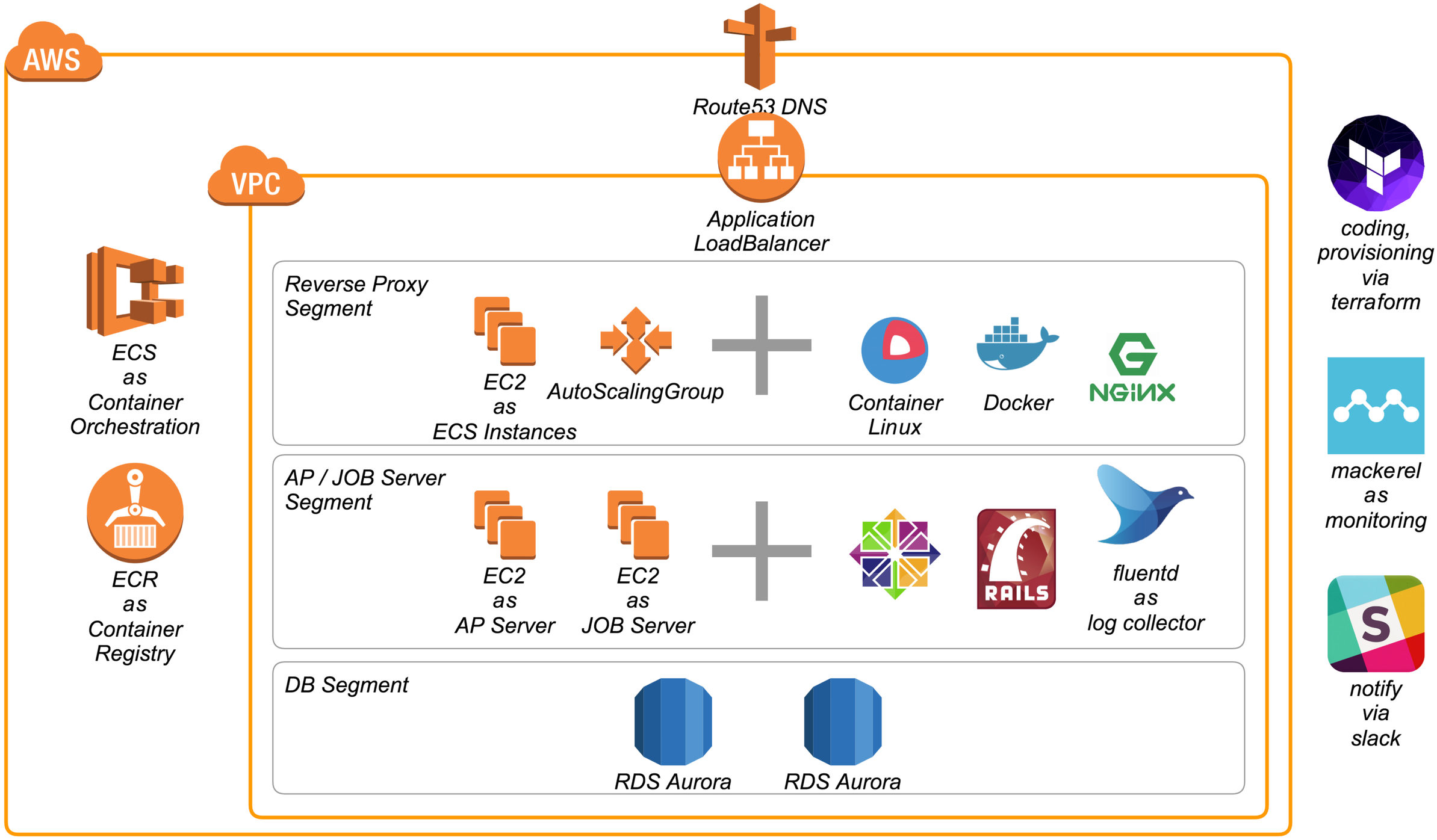
jun asano

- *X Team*
- *Tech Lead*
- 2男児父

Architecture Overview of On-Premise


On-Premise
Firewall
BIGIP(Act)
BIGIP(Stb)
Control Segment
MON1
MON2
AP Segment
OPE
AP
JOB / Mail
RAILS
DB Segment
Virtual IP
MySQL
Replication
DB(Master)
DB(Slave)
NAS

Architecture Overview of AWS


AWS
Route53 DNS
VPC
Application
LoadBalancer
ECS
as
Container
Orchestration
ECR
as
Container
Registry
Reverse Proxy
Segment
EC2
as
ECS Instances
AutoScalingGroup
Container
Linux
Docker
NGINX
AP / JOB Server
Segment
EC2
as
AP Server
EC2
as
JOB Server
RAILS
fluentd
as
log collector
DB Segment
RDS Aurora
RDS Aurora
coding,
provisioning
via
terraform
mackerel
as
monitoring
notify
via
slack

用途	On-Premise	AWS
Provisioning	なし	Terraform
Load Balancer	BIG-IP	AWS ALB & nginx with Amazon ECS
Database	MySQL	Amazon Aurora
Alert / Monitoring	Hinemos	fluentd / mackerel & slack
NFS	NAS	Amazon S3 + Goofys

Provisioning

Before

- 構成管理、なし
- インフラ設計書は稼働当初(2010年)のものが残っているだけ...
- ブラックボックス &「秘伝のタレ」化
  - インフラチームと古参メンバーに聞かないと詳細が分からない

HashiCorp

Terraform

Infrastructure as Code

- 構築対象のAWSリソースをひたすらコード化
- stagingに適用後、問題なければproductionへ
- Terraformのmodule機能を活用
  - stagingとproductionで異なる部分のみ変数化
  - 例: staging -> t2.micro, production -> m4.large

Terraform 所感

つらみ

- Version Up 頻度が高く、追いつくのが大変
- plan (aka dry-run)とapplyで結果が異なる場合があり、apply恐怖症になる

よさ

- 簡易なDSL なので、とっつきやすい
- 入り組んだAWSリソースが可視化される

Load Balancer

Before

- インフラチーム管轄のBIG-IP
- 負荷分散ができてない
  - 接続元IPで振分け先が固定されてしまう
- 煩雑なiRule定義
  - url上の顧客コードからAPサーバーへの振分けをiRuleで定義
  - 導入顧客数分の設定量に...(大量のif分岐)

AWS ALB & nginx with Amazon ECS

AWS ALB

- Application Load Balancer
- Amazon ECS との親和性が高かったため採用

nginx as reverse proxy

- 顧客コード毎のAPサーバ振分けのiRule定義が
  多すぎてAWS ALB のpath based routingで
  賄いきれない

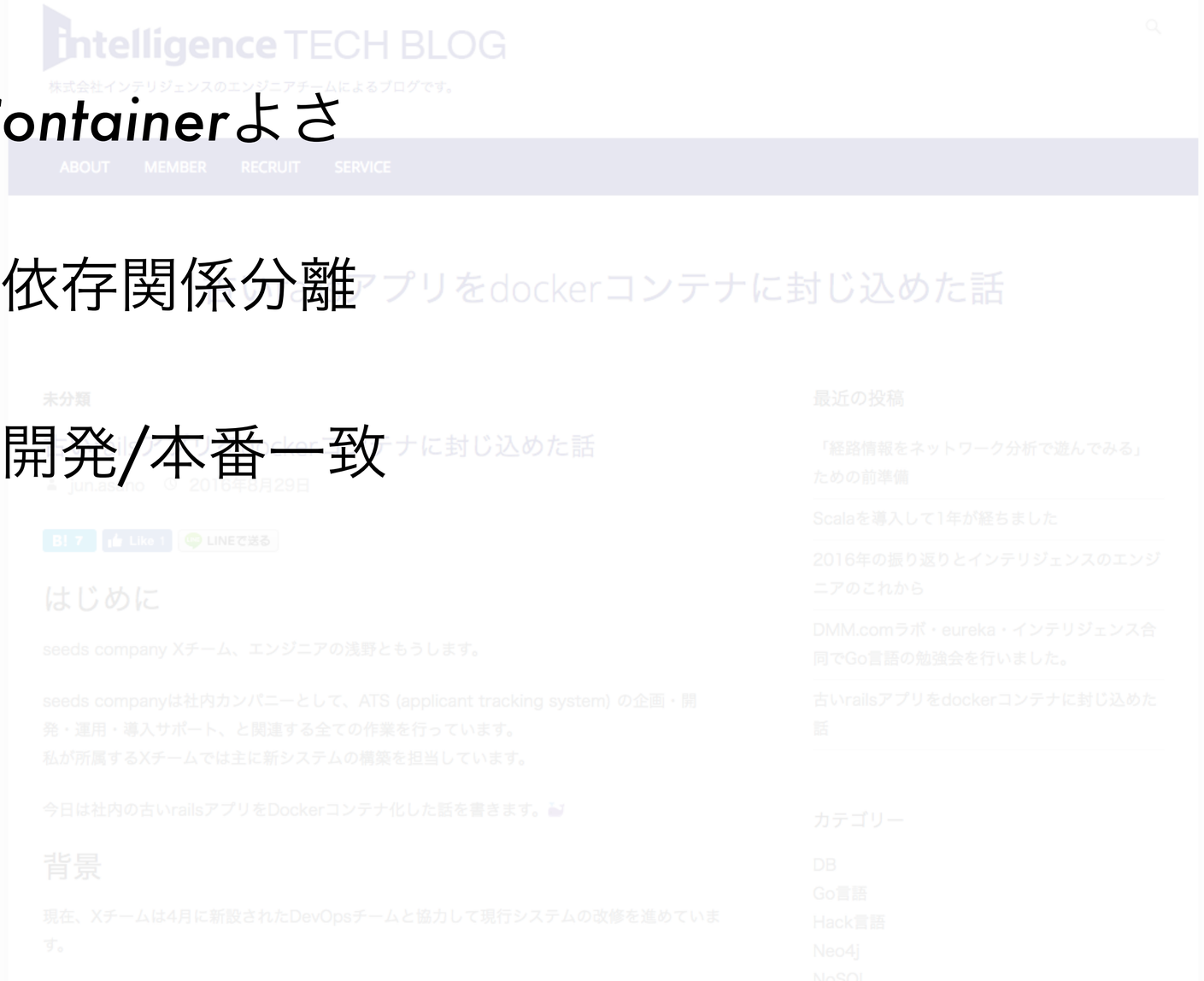
- そもそも顧客追加・変更の頻度が多いため、
  AWS ALB をあまりカジュアルにいじりたくない

Why Container ?

http://techblog.inte.co.jp/

Why Container ?

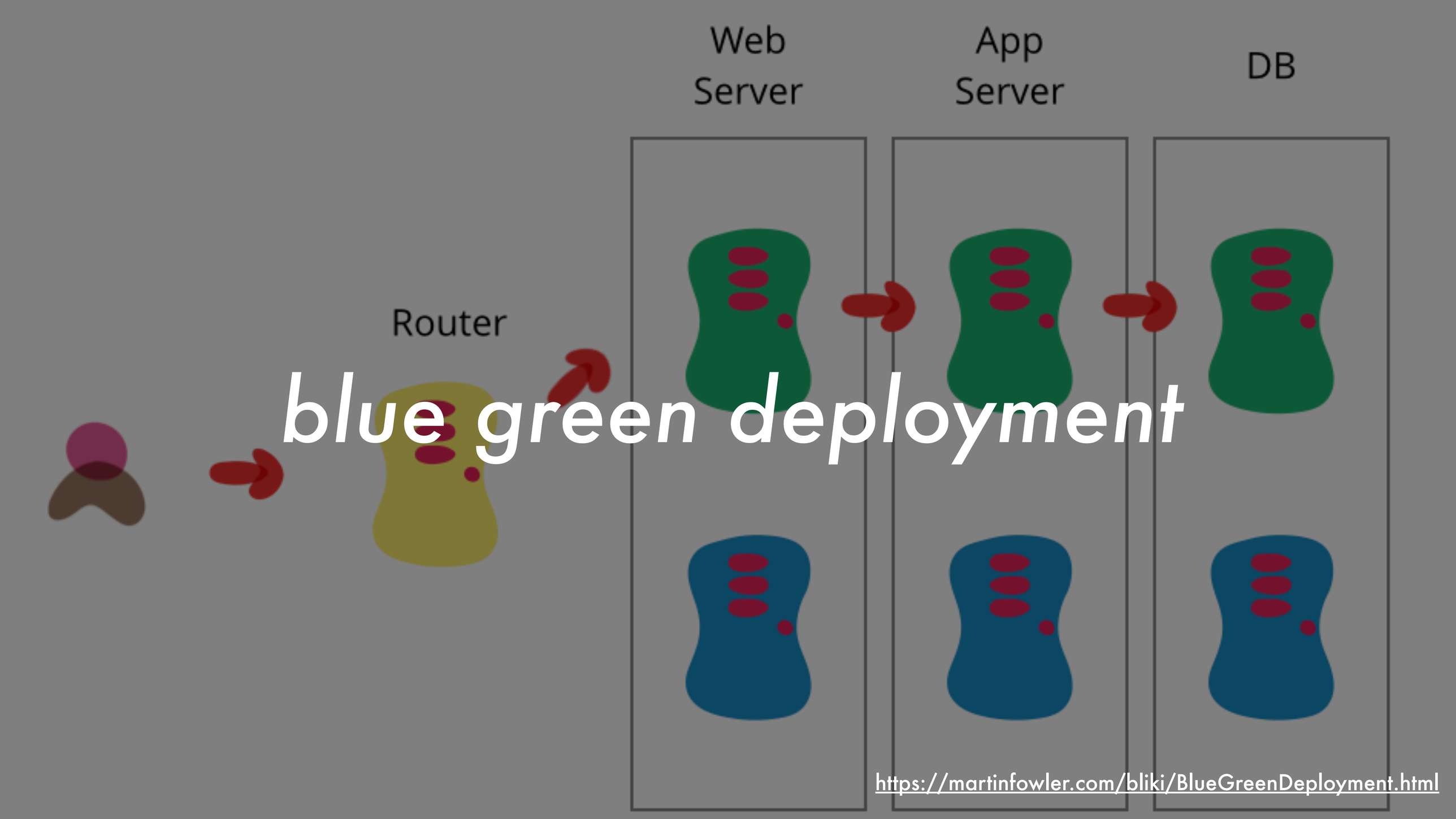
- Containerよさ
  - 依存関係分離
  - 開発/本番一致



Amazon ECS

- サービスを安定・継続的に提供する上で
  魅力的なしくみが揃っていたことが決め手

  - *app auto scaling*
  - *container auto recovery*
  - *blue green deployment*



blue green deployment

https://martinfowler.com/bliki/BlueGreenDeployment.html

3 steps for blue green deployment on Amazon ECS

```
docker push url.to.ecr.region.amazonaws.com/your-repos

aws ecs register-task-definition \
  –cli-input-json file://path/to/task-definition.json

aws ecs update-service \
  –cli-input-json file://path/to/update-service.json
```

safe & easy !

blue green deployment step 1

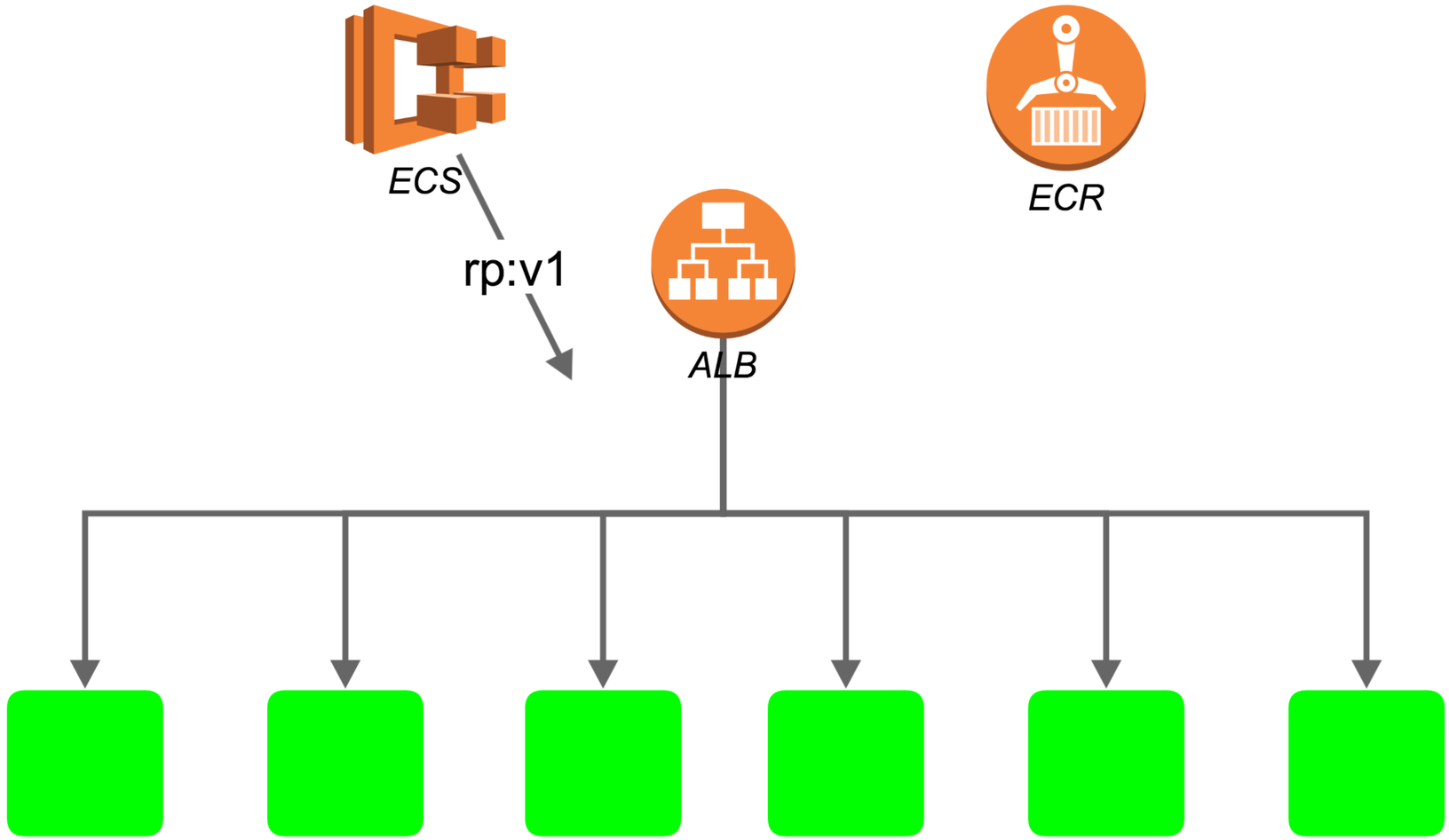
service : reverse-proxy

task-definition : rp:v1

desired count : 6

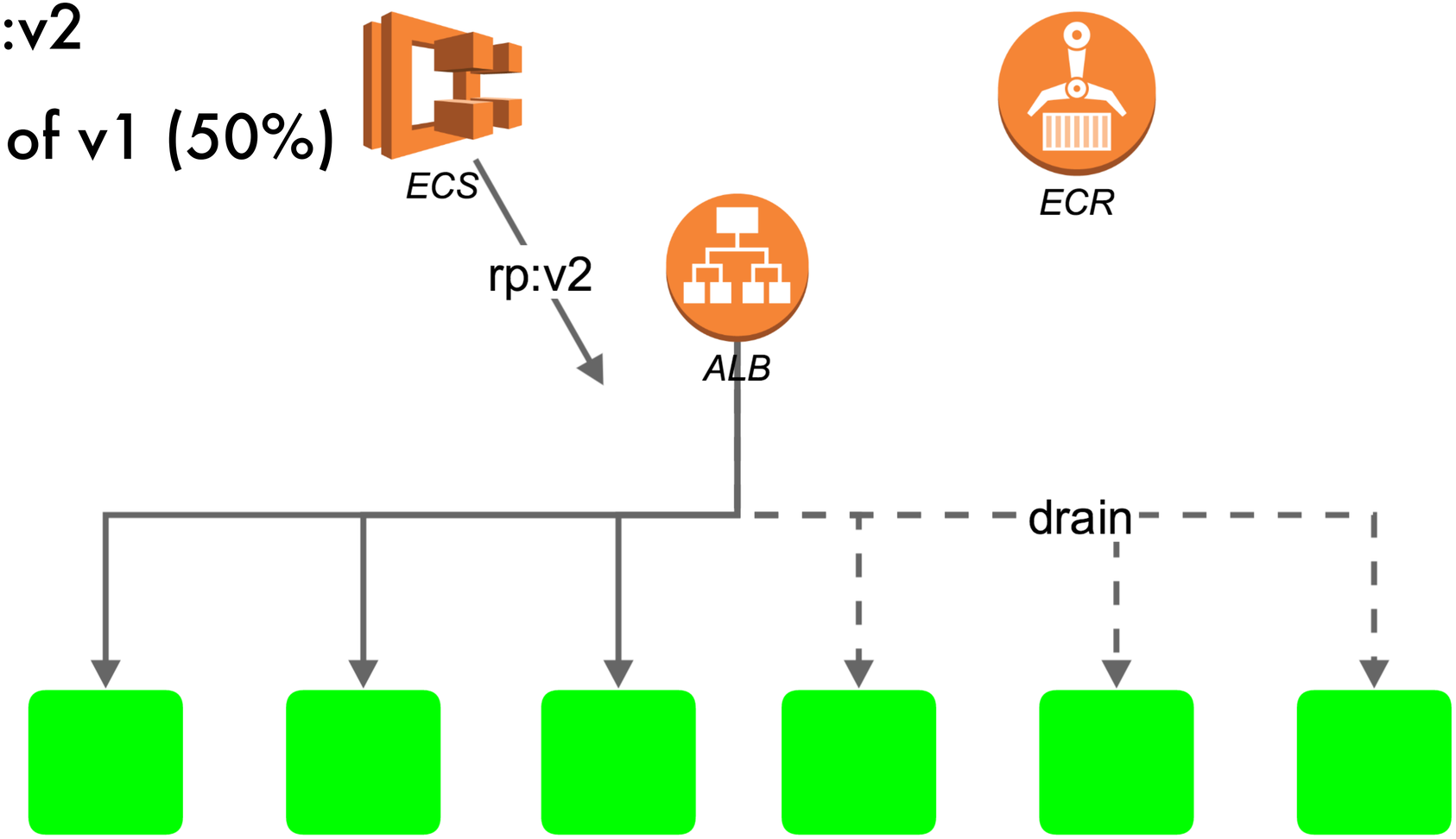
maximum % : 100

minimum healthy % : 50



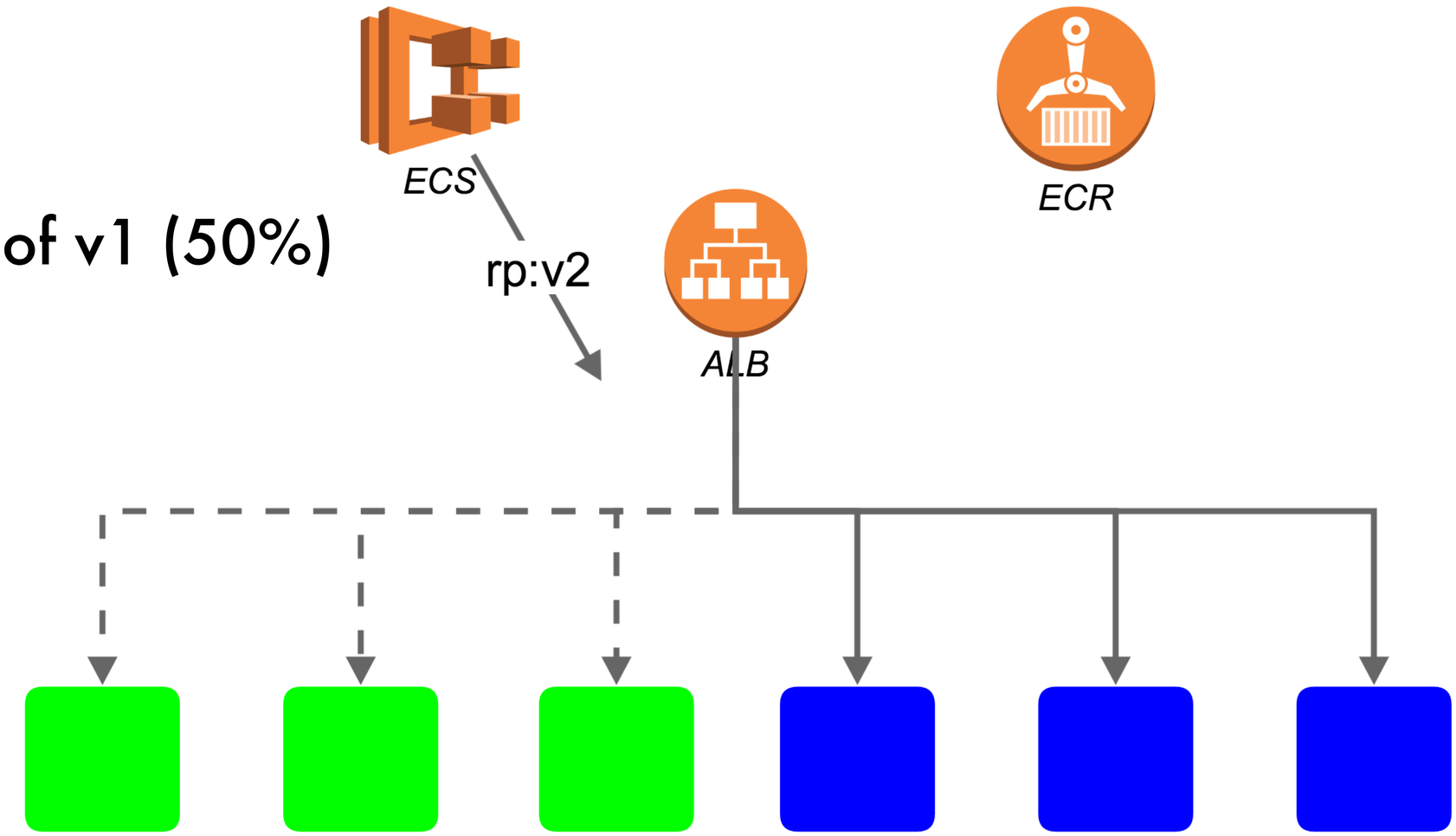
blue green deployment step 2

- task-definition: rp:v2
- drain connection of v1 (50%)



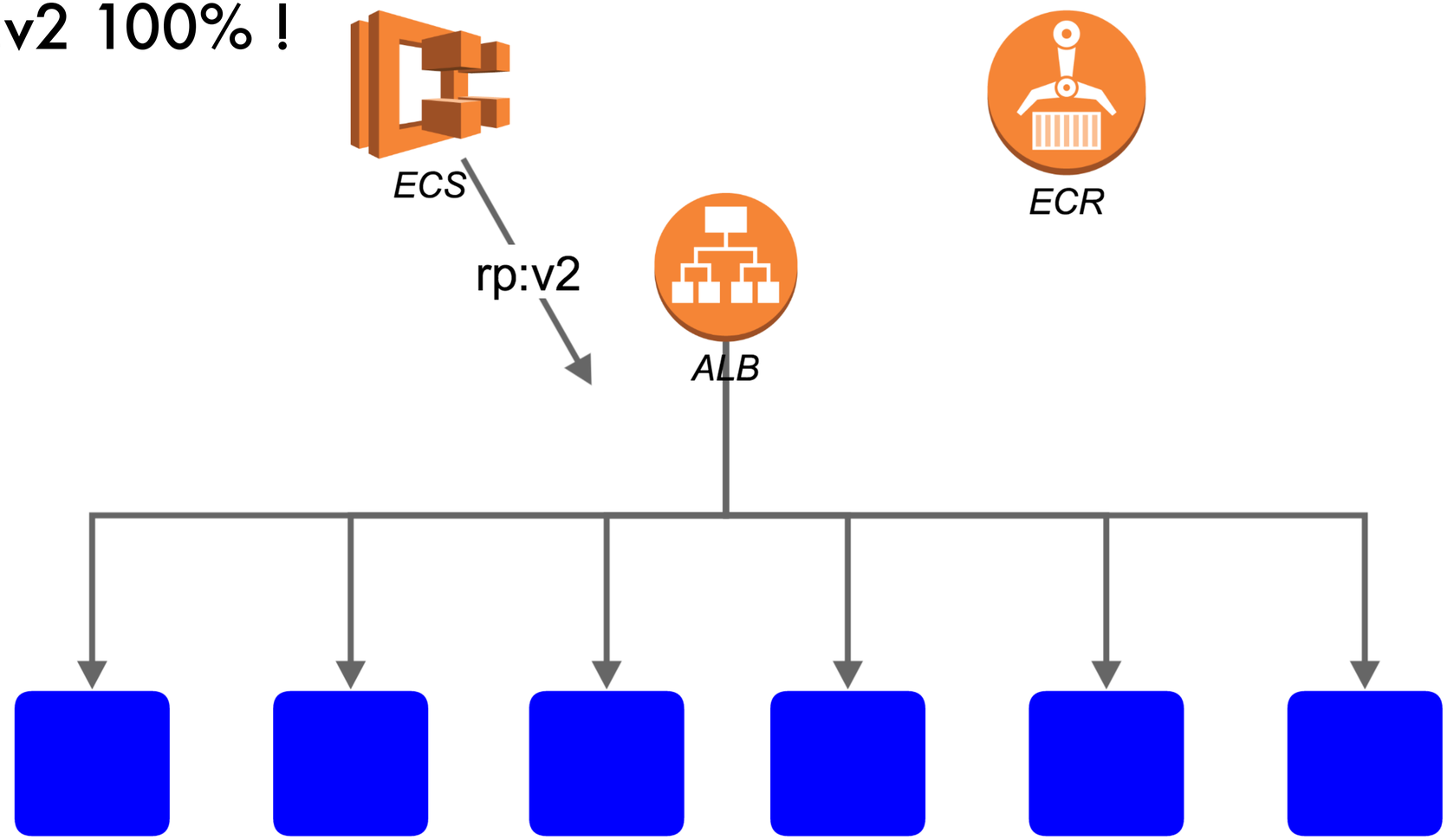
blue green deployment step 3

- start v2 service
- connect to v2
- drain connection of v1 (50%)



blue green deployment step 4

- task-definition: rp:v2 100% !



Database

Before

- 無意味なデータベース分割
  - サービス拡大に応じてデータベースが増加
  - 運用負荷を招く
- *Trouble*
  - レプリカが機能してないことが度々あった
  - そしてそれに誰も気づいていなかった...


Amazon Aurora

Amazon Aurora

- 機能単位でDBを整理し、cluster構築

staging 構築時は Amazon RDS MySQL 5.6だった

- 構築後にAmazon Auroraが
  監査ログに対応したため急遽Auroraに変更

- 性能とメンテナンス性が決め手

DB移行をどうしたか

- セキュリティポリシー上、
  On-Premiseからのレプリケーションは不可
- Downtimeは深夜のみ可
  - 小さいDBはフルダンプ&レストア
  - 作業に数時間かかるDBのみ
    差分インポートで対応

Amazon Support

手厚い *Support*

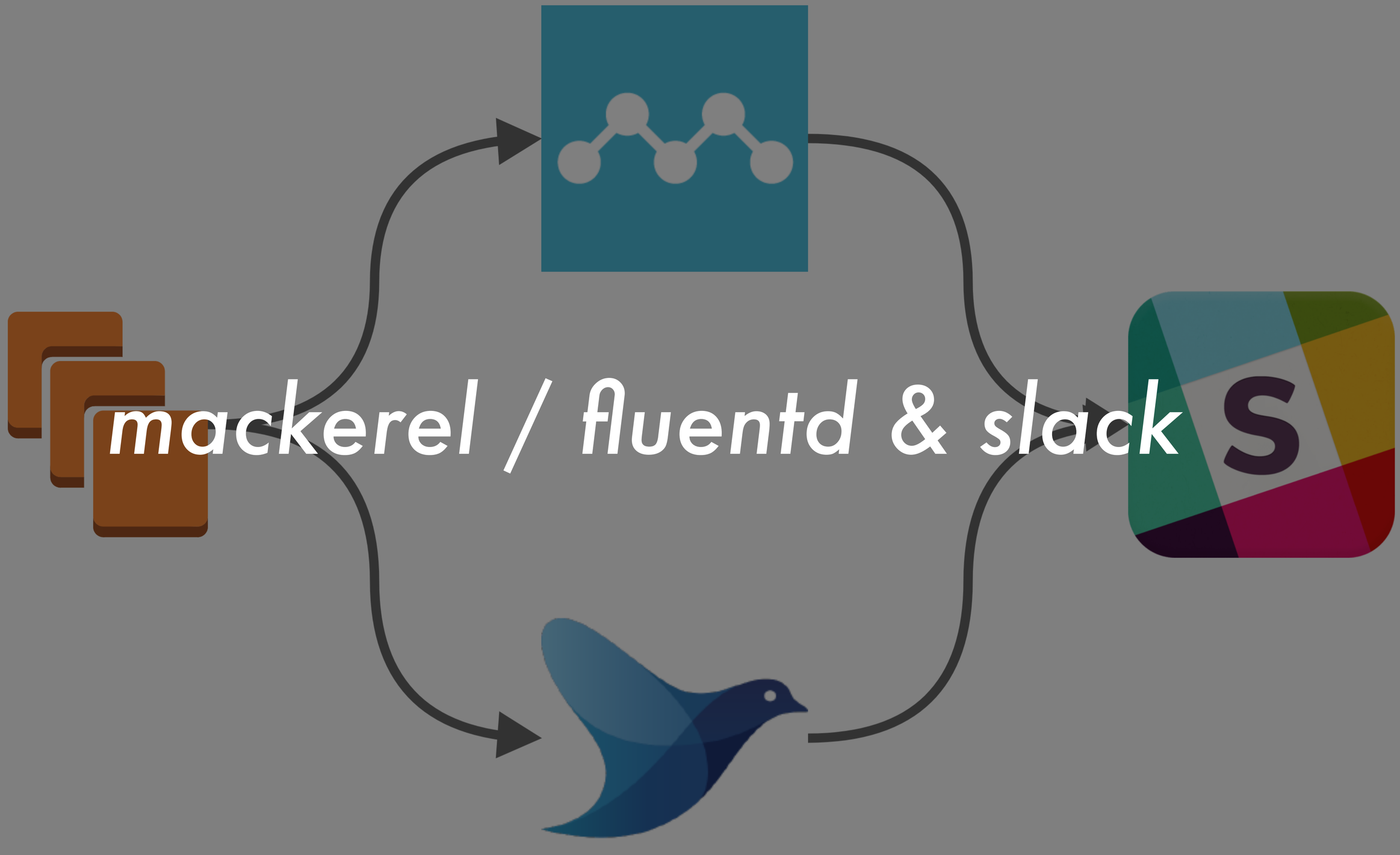
- *Aurora*に限らず、*MySQL*のダンプ＆リストア時間の節約方法などもアドバイスいただく

いつも助かってます！

Alert / Monitoring

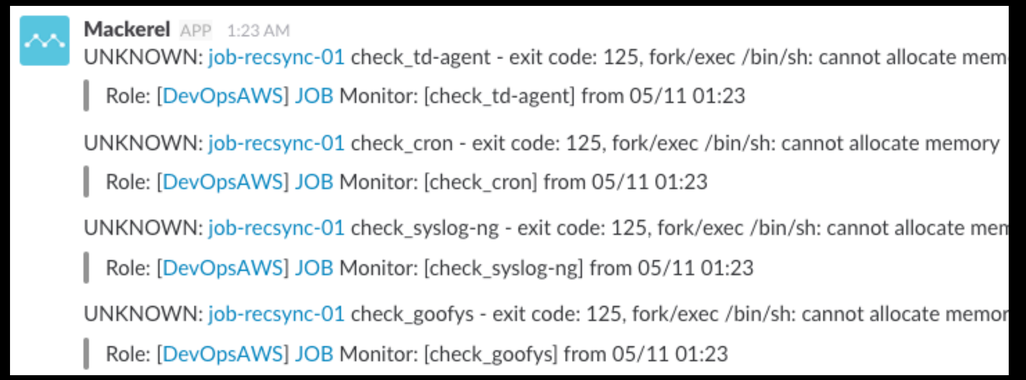
Before

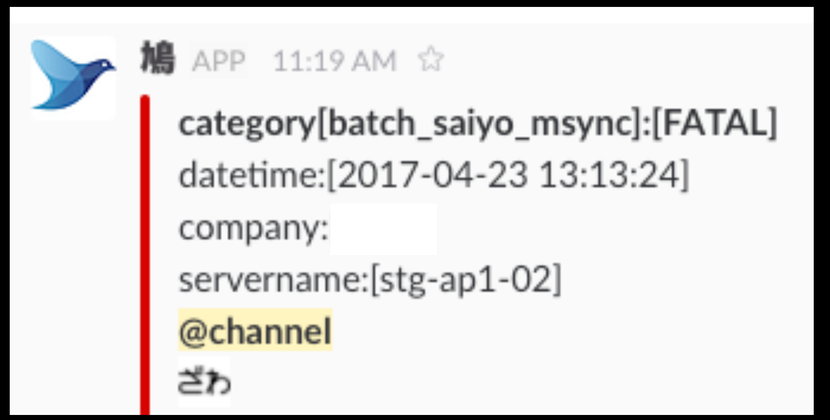
- Hinemosがオオカミ少年化
- 重要なアラートが埋もれてしまう
- 監視されるべきプロセスが漏れている
- GUIベースであるため、設定変更が手間
- Windows Server が必要


mackerel / fluentd & slack

mackerel / fluentd & slack

- 必要なAlert・Monitoringを精査、設定
  - process監視は mackerel
  - log監視は fluentd
- slack通知し、すぐ検知できるように





NFS

Before

- 全サーバーに同じNASをマウント
- 運用作業で頻繁に利用
- 設定・管理はインフラチーム任せ...

Amazon EFS

Amazon EFS

- ...
- 東京リージョン未対応 ...

Amazon S3 + Goofys

- NFSを使わない運用を検討する時間はなかった
- NASの代替策を検討
  - Goofys :
    Amazon S3をNFSとして扱える golang製tool

実際に移行してどうだったのか？

Timeline

- 2016年11月 移行プロジェクト開始
- 2016年12月 ステージング環境構築
- 2017年2月 第1次移行
- 2017年3月 完全移行完了

Mind Set

小さく素早く始める

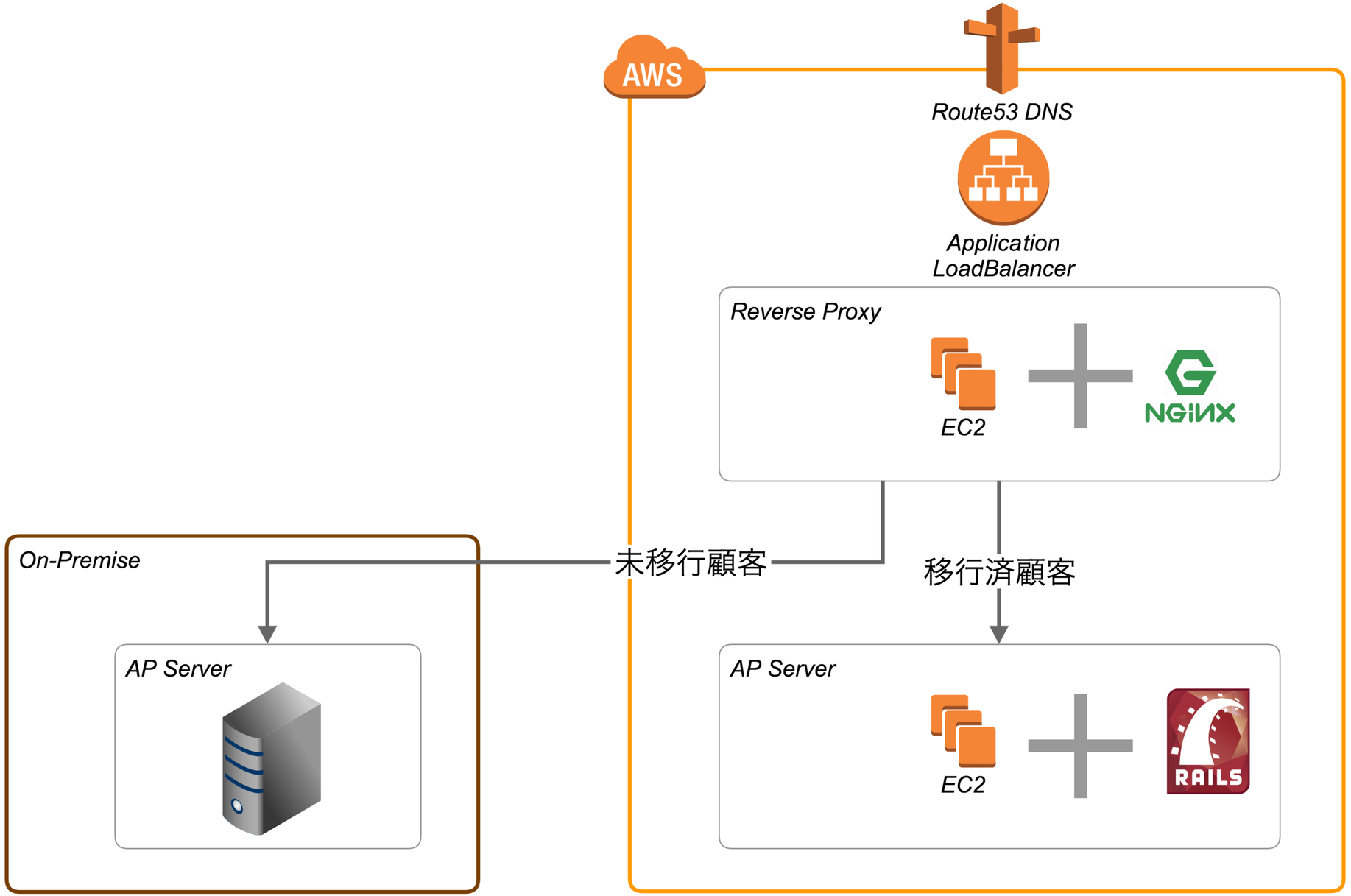
- とにかく手早く*staging*環境を
  - すぐ立て・捨てられるクラウドの利点を活かす
- 環境構築する過程でブラックボックスがクリアになり、問題点が見えてくる
  - 移行の実現可能性を早期に判断するためにも必須
- 環境構築したらテストを回す
  - *E2E*テスト、負荷テスト

段階的な移行

- 1回で全て移行するのではなく段階的に移行
- 並行稼動の設計コストは高い...
- 一方で下記の問題を検出できた
  - 移行手順の過不足
  - 移行後の運用課題
- DevOps以外のチームがAWSに慣れる

1次移行

- Domainを Amazon Route53 に移行
- nginxが顧客コードに応じて
  AWSとOn-Premise環境へ振分け


AWS
Route53 DNS
Application
LoadBalancer
Reverse Proxy
EC2
NGINX
未移行顧客
移行済顧客
On-Premise
AP Server
AP Server
EC2
RAILS

2次移行

- 移行漏れが発覚し、一度失敗
  - 切り戻しラインを定めておいたため、
    中断に伴うトラブルは無し
- 1週間後にリトライし、移行成功

移行で良くなった？

➡ *YES*

- ブラックボックス：解消 (コード化)
- 不具合調査：チーム内で完結、スピードアップ
- サーバー追加：*45*日 ➡ *1*日
- 構成変更: スケールアップ/アウトも即時対応

➡ チームが自律的に活動できるように

Summary

レガシーインフラにAWS移行は有効、ただし...

- スキルとリソースが必要
- 組織の協力も必要
- 多くのリスクやトレードオフと戦う覚悟も必要
  - ブラックボックス ≒ 地雷

Mind Set

- 小さく素早く始める
- レガシー = ブラックボックス = リスク
  - ゼロにはできない
  - 進めながら潰す
- × エンジニアの自己満足
  - ◯ ビジネス要求に対応できる

移行できれば

- 全てが透明になる
- ハンドリング可能になる
- 市場変化に対応できるようになる

We are hiring!
@ Intelligence
SEEDS company

Thank You !