DevOps … 失敗談

株式会社トライビート 2018.05.24
Satoshi Higashizaki <s-higashizaki@tribeat.com>

tribeat

アジェンダ

1. 自己紹介
2. 導入サービス概要
3. 導入コンセプト
4. 導入前のシステム・開発フローとその問題点
5. 理想像と最終結果
6. 反省点
7. まとめ

自己紹介

- 東崎 智 <Satoshi Higashizaki>
- 1975.07 ♋名古屋出身
  - おでんには味噌
- 1994.04 電気工事士
- 2003.06 未経験でIT業界参入
  - WBSエディタやISSUE管理などのプロジェクト支援ツールを自社開発
- 2013.10 転職
  - SESにて某中古車検索/情報サイト開発・保守
- 2017.01 株式会社トライビートに就職
  - https://www.wantedly.com/users/21270296
  - Webアプリケーション開発
  - PHP/Bash/Docker/AWS ...etc
  - チーフアーキテクト

サービス/環境概要

当たり障りのない感じで

- 中古車検索/情報サイト（C向け側）
- プライベートクラウド上に構築
- Webサーバ（Apache + PHP）約50台
- RDBMSサーバ（PostgreSQL）約20台
- KVS（TokyoTyrant）約2台
- NAS
- Hadoopサーバ
- etc...

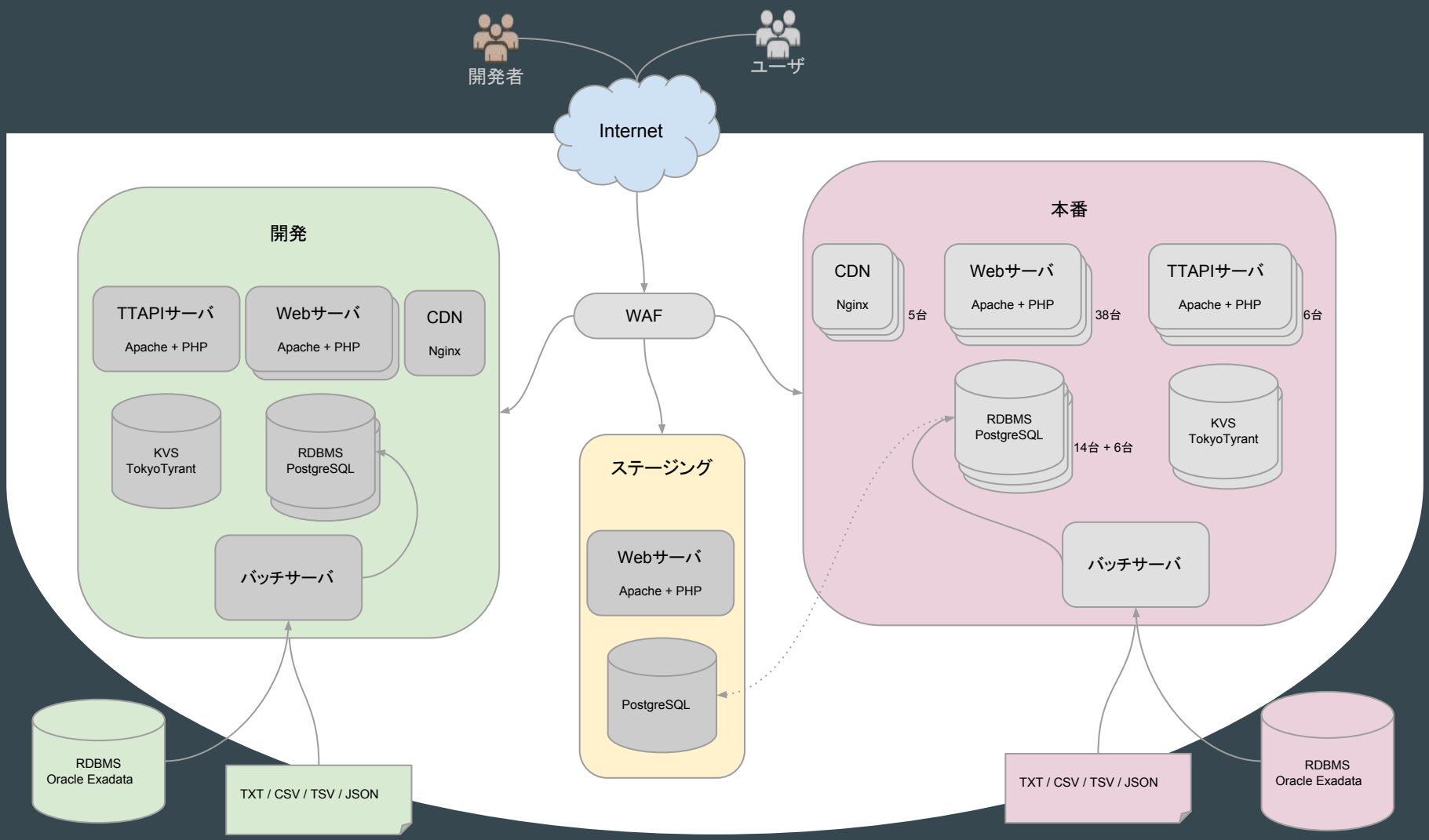
きっかけ

- 全体的なHW、OS、MW等の老朽化
- 一部実機もある環境の運用コストの問題
- 障害時の対応速度の限界
- APIファーストによる将来像の実現

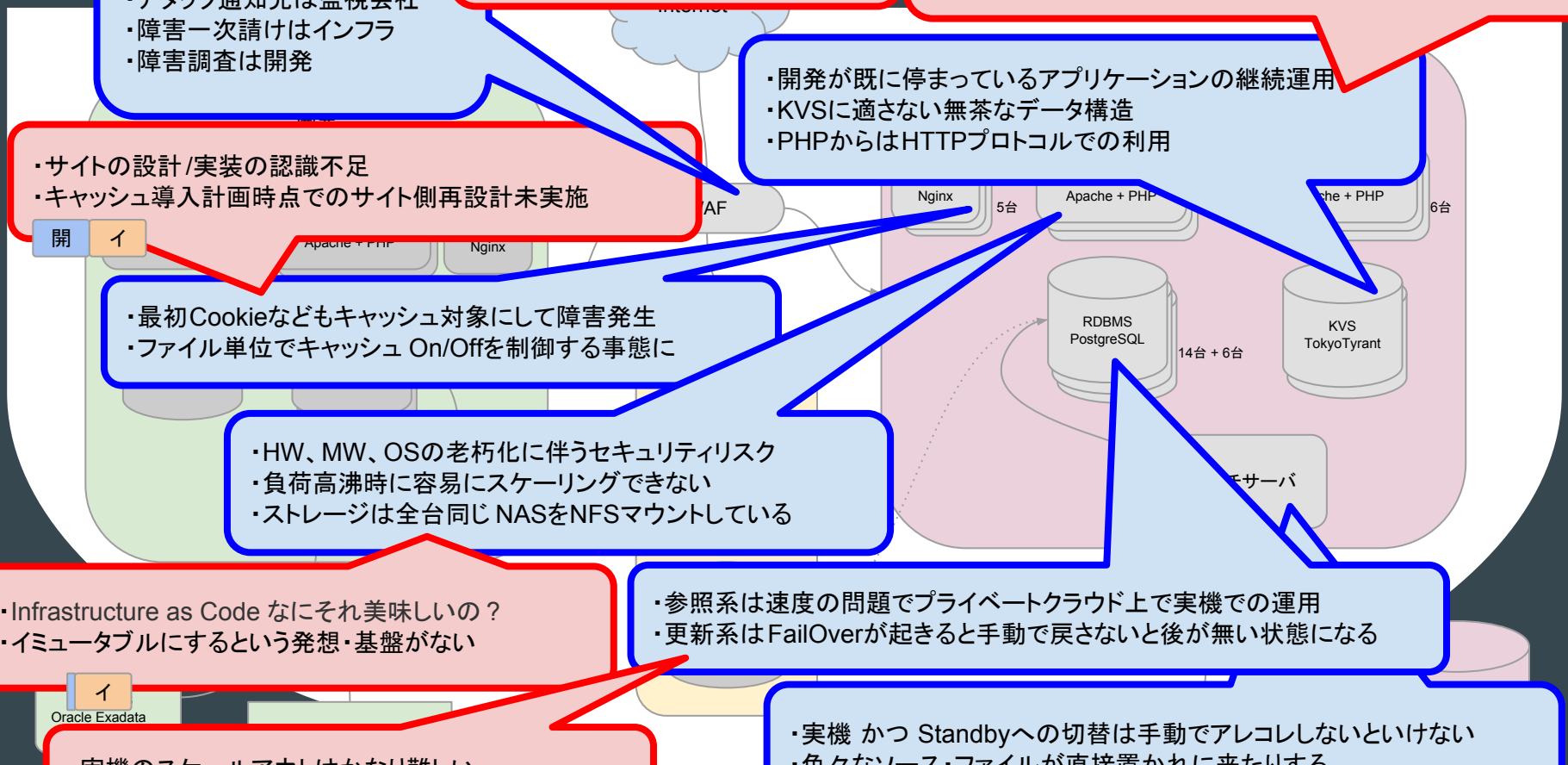
あるべきすがた

- 速い 応答性能、保守性に優れていること
- 強い 高可用性、耐障害性に優れていること
- 柔軟 変化に強い構造を持つこと
- 人系も含めた開発、運用、保守の仕組みとなっていること
- 開発・インフラ対応のコスト、スピードの改善ができること

Problem - 構成図


開発者
ユーザ
Internet
開発
TTAPIサーバ
Apache + PHP
Webサーバ
Apache + PHP
CDN
Nginx
KVS
TokyoTyrant
RDBMS
PostgreSQL
バッチサーバ
WAF
ステージング
Webサーバ
Apache + PHP
PostgreSQL
本番
CDN
Nginx
5台
Webサーバ
Apache + PHP
38台
TTAPIサーバ
Apache + PHP
6台
RDBMS
PostgreSQL
14台 + 6台
KVS
TokyoTyrant
バッチサーバ
RDBMS
Oracle Exadata
TXT / CSV / TSV / JSON
TXT / CSV / TSV / JSON
RDBMS
Oracle Exadata

Problem - 構成図



・ルール作成は開発
・ルール設定はインフラ
・アタック通知先は監視会社
・障害一次請けはインフラ
・障害調査は開発

開 イ
・セキュリティエンジニアが不在 ？
・など闇が深いので今回は割愛 ^^;

開 イ
・セキュリティリスクの放置
・KVSの特性を無視した利用で速度を犠牲
・そもそも通信のオーバヘッドが大きい方式の採用に疑問

・開発が既に停まっているアプリケーションの継続運用
・KVSに適さない無茶なデータ構造
・PHPからはHTTPプロトコルでの利用

・サイトの設計/実装の認識不足
・キャッシュ導入計画時点でのサイト側再設計未実施

開 イ
・最初Cookieなどもキャッシュ対象にして障害発生
・ファイル単位でキャッシュ On/Offを制御する事態に

・HW、MW、OSの老朽化に伴うセキュリティリスク
・負荷高沸時に容易にスケーリングできない
・ストレージは全台同じ NASをNFSマウントしている

・Infrastructure as Code なにそれ美味しいの？
・イミュータブルにするという発想・基盤がない

・参照系は速度の問題でプライベートクラウド上で実機での運用
・更新系はFailOverが起きると手動で戻さないと後が無い状態になる

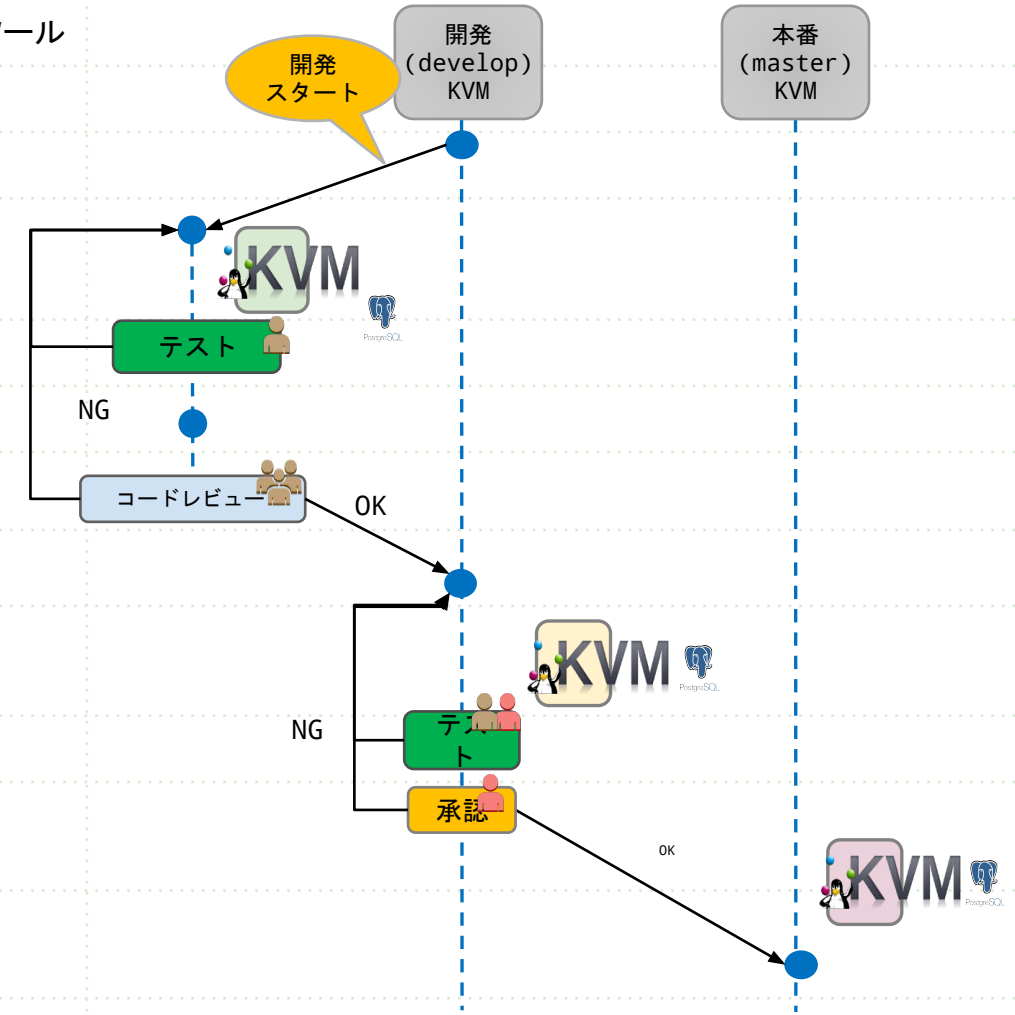
イ
・実機のスケールアウトはかなり難しい
・FailOverの切り戻しは停止が必要こともあった

・実機 かつ Standbyへの切替は手動でアレコレしないといけない
・色々なソース・ファイルが直接置かれに来たりする

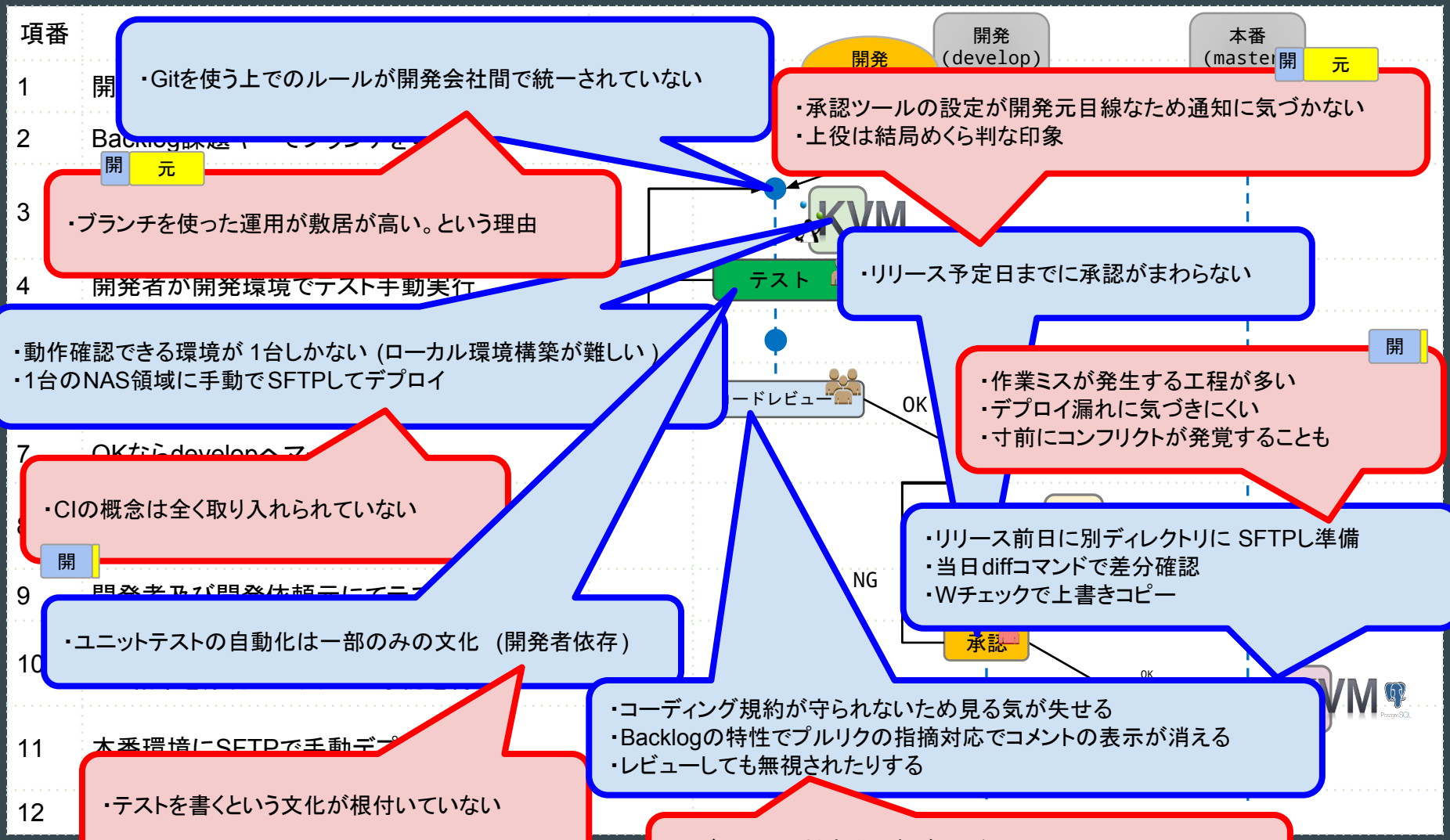
・大昔にMW、Appまで全て開発会社が構築したまま
開 イ

Problem - 開発フロー図

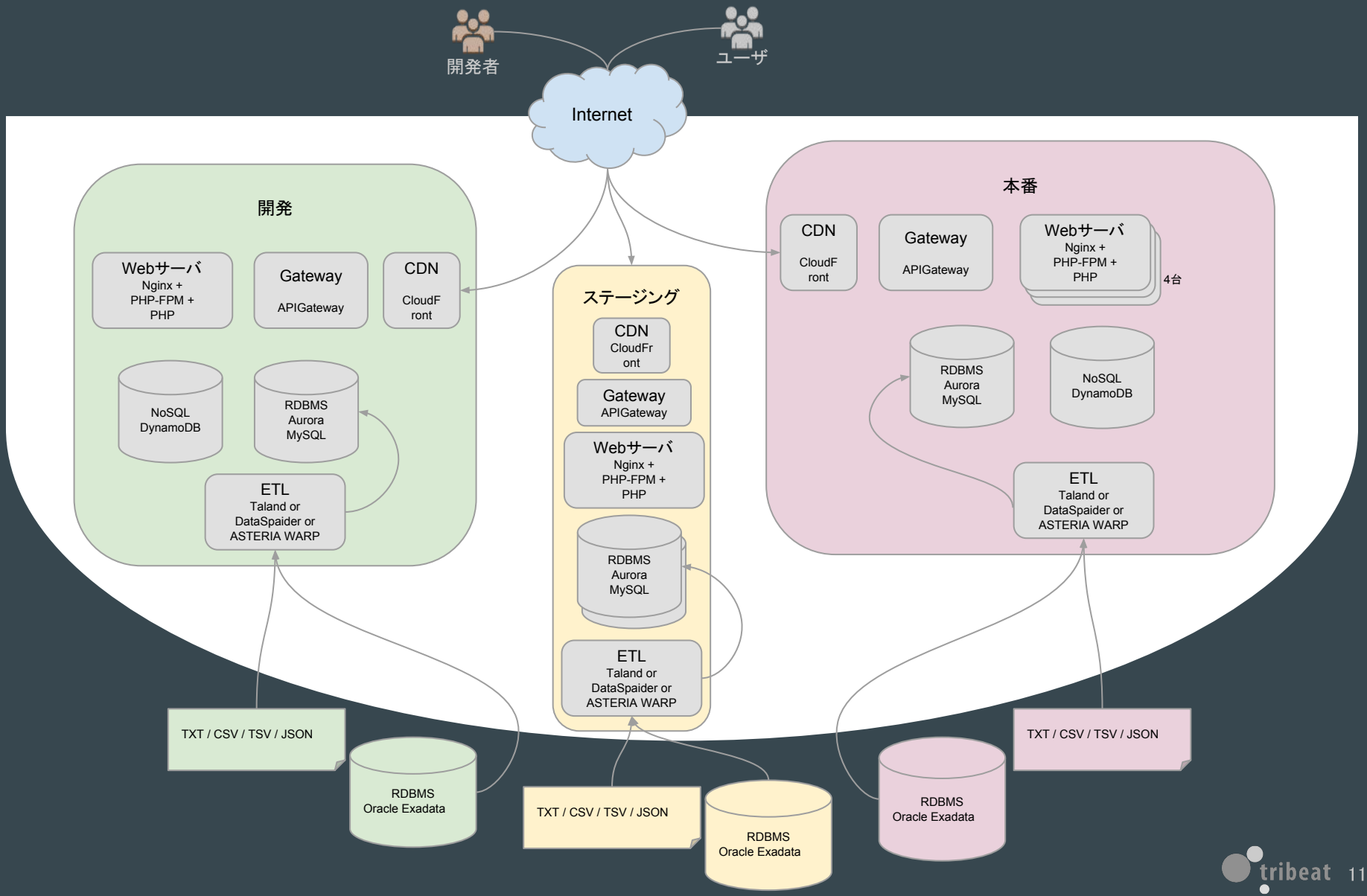
項番	内容	ツール
1	開発スタート	
2	Backlog課題キーでブランチを切る	
3	手動で開発環境へSFTPで手動デプロイ	
4	開発者が開発環境でテスト手動実行	
5	改修内容をコミット	
6	コードレビュー後developブランチにマージ	
7	OKならdevelopへマージ	
8	ステージング環境にSFTPで手動デプロイ	
9	開発者及び開発依頼元にてテストを実施	
10	独自の承認のツール(Webアプリ)にExcelのテスト結果を添付してリリース承認を得る	
11	本番環境にSFTPで手動デプロイ	
12	masterへマージ	



Problem - 開発フロー図



After? - 構成図

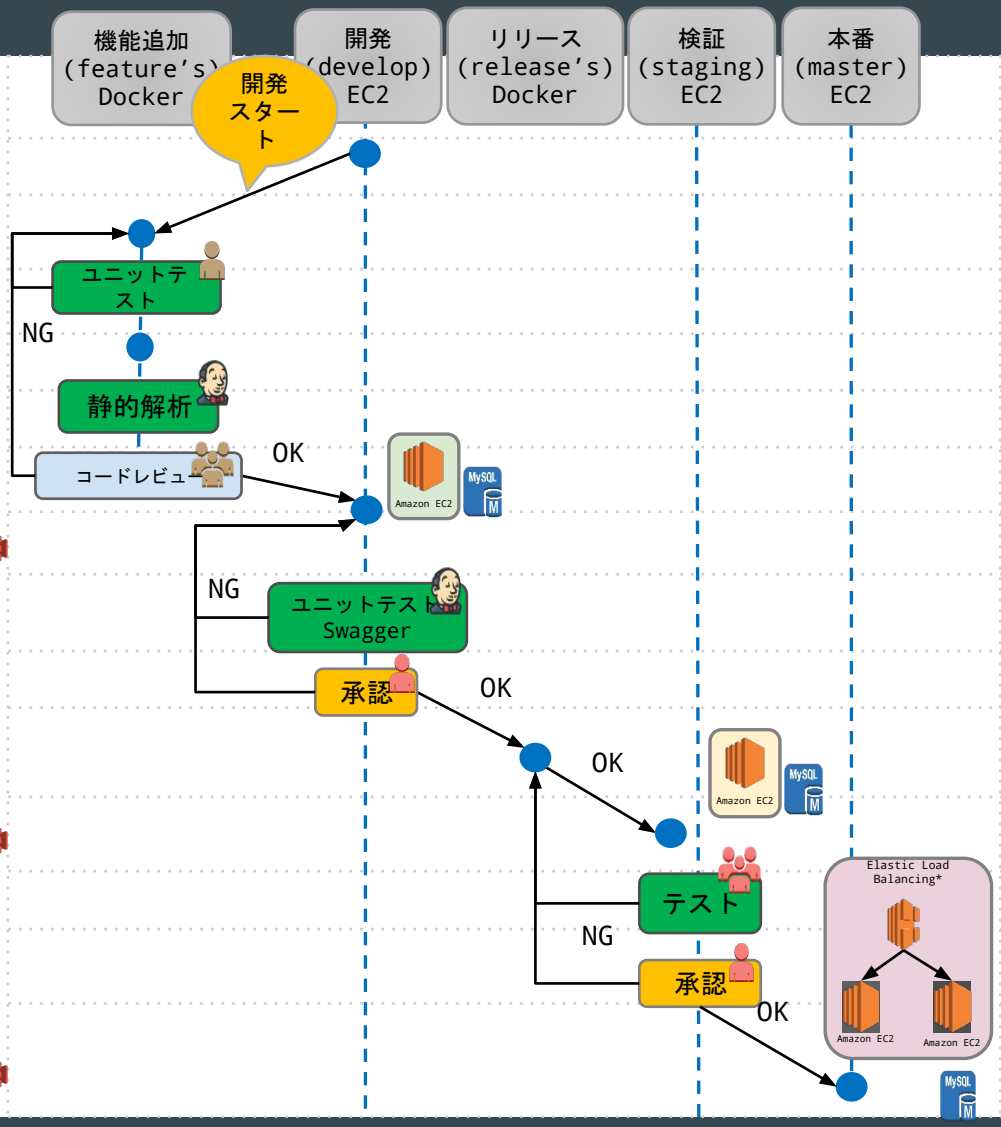

開発者
ユーザ
Internet
開発
Webサーバ
Nginx +
PHP-FPM +
PHP
Gateway
APIGateway
CDN
CloudF
ront
NoSQL
DynamoDB
RDBMS
Aurora
MySQL
ETL
Taland or
DataSpaider or
ASTERIA WARP
ステージング
CDN
CloudFr
ont
Gateway
APIGateway
Webサーバ
Nginx +
PHP-FPM +
PHP
RDBMS
Aurora
MySQL
ETL
Taland or
DataSpaider or
ASTERIA WARP
本番
CDN
CloudF
ront
Gateway
APIGateway
Webサーバ
Nginx +
PHP-FPM +
PHP
4台
RDBMS
Aurora
MySQL
NoSQL
DynamoDB
ETL
Taland or
DataSpaider or
ASTERIA WARP
TXT / CSV / TSV / JSON
RDBMS
Oracle Exadata
TXT / CSV / TSV / JSON
RDBMS
Oracle Exadata
RDBMS
Oracle Exadata
TXT / CSV / TSV / JSON

After? - 構成図


開発者
ユーザ
Internet
・全文検索エンジンを使った高速化を別チームで検証
・今回はDirectConnect経由で直接叩いちゃえ! (ヤケ気味)
・PostgreSQLからMySQLの移行に伴い速度劣化
・TokyoTyrantはしがらみが多くてひっぺがし難い
・つまりは工数が跳ね上がる
・ひとまずローンチまでは本番のDBに向けちゃおう
・性能が高いものが必要
・開発のしやすさは重要なポイント
・メーカーのサポート体制も重要
・Auroraが高価
・ASTERIA WARPのライセンスも必要
・言迷バイアスにより ASTERIA WARP を採用
・同時にETL職人も就職おめでとう!
CDN
CloudFront
Gateway
ETL
Taland or
DataSpaider or
RDBMS
Aurora
MySQL
RDBMS
Aurora
NoSQL
ASTERIA WARP
RDBMS
Oracle Exadata
TXT / CSV / TSV / JSON
RDBMS
Oracle Exadata

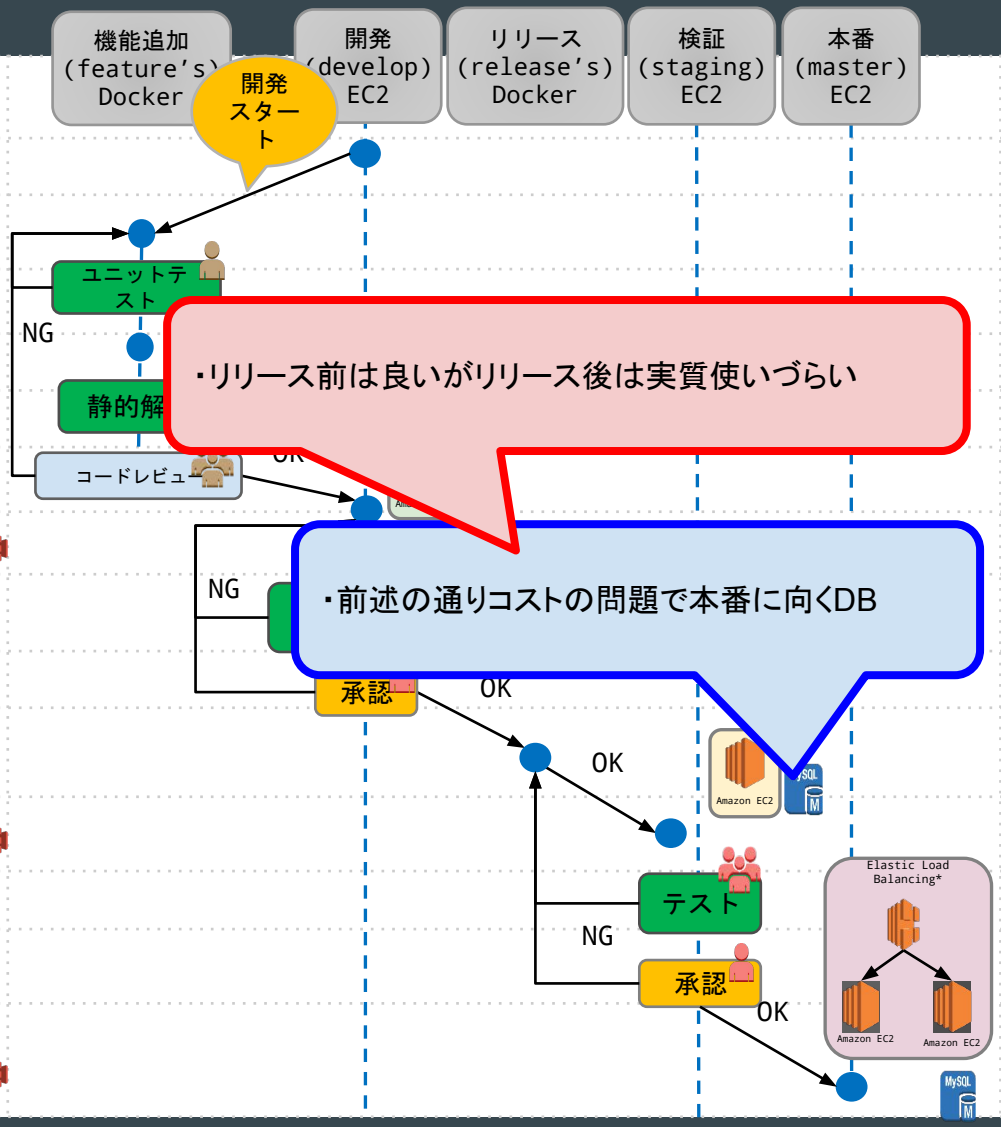
After? - 開発フロー図

項番	内容	ツール
1	開発スタート	
2	feature/Backlog課題キーでブランチを切る ローカル開発環境はDockerコンテナ	
3	開発者がローカル環境でユニットテスト手動実行	
4	改修内容をコミット	
5	CIでソースコードの静的解析と自動修正	
6	コードレビュー、2/5人がOKならdevelopブランチにマージ	
7	OKなら開発環境へマージし、デプロイ	
8	開発環境上で自動でユニットテスト & ドキュメント生成	
9	プロジェクト責任者が内容を確認、リリース判定	
10	リリースは基本週一で行っているので、その週にリリース予定の変更のみをチェリーピック or マージ	
11	まとまったリリース分をステージング環境へマージし、デプロイ	
12	開発者及び開発依頼元にてテストを実施	
13	独自の承認のツール(Webアプリ)にExcelのテスト結果を添付してリリース承認を得る	
14	ブルー・グリーンデプロイを採用。リリース予定の環境と既存の本番環境を同時に動かし、リリース予定の環境でテストOKであればLBの向きを変更してリリースを行う	

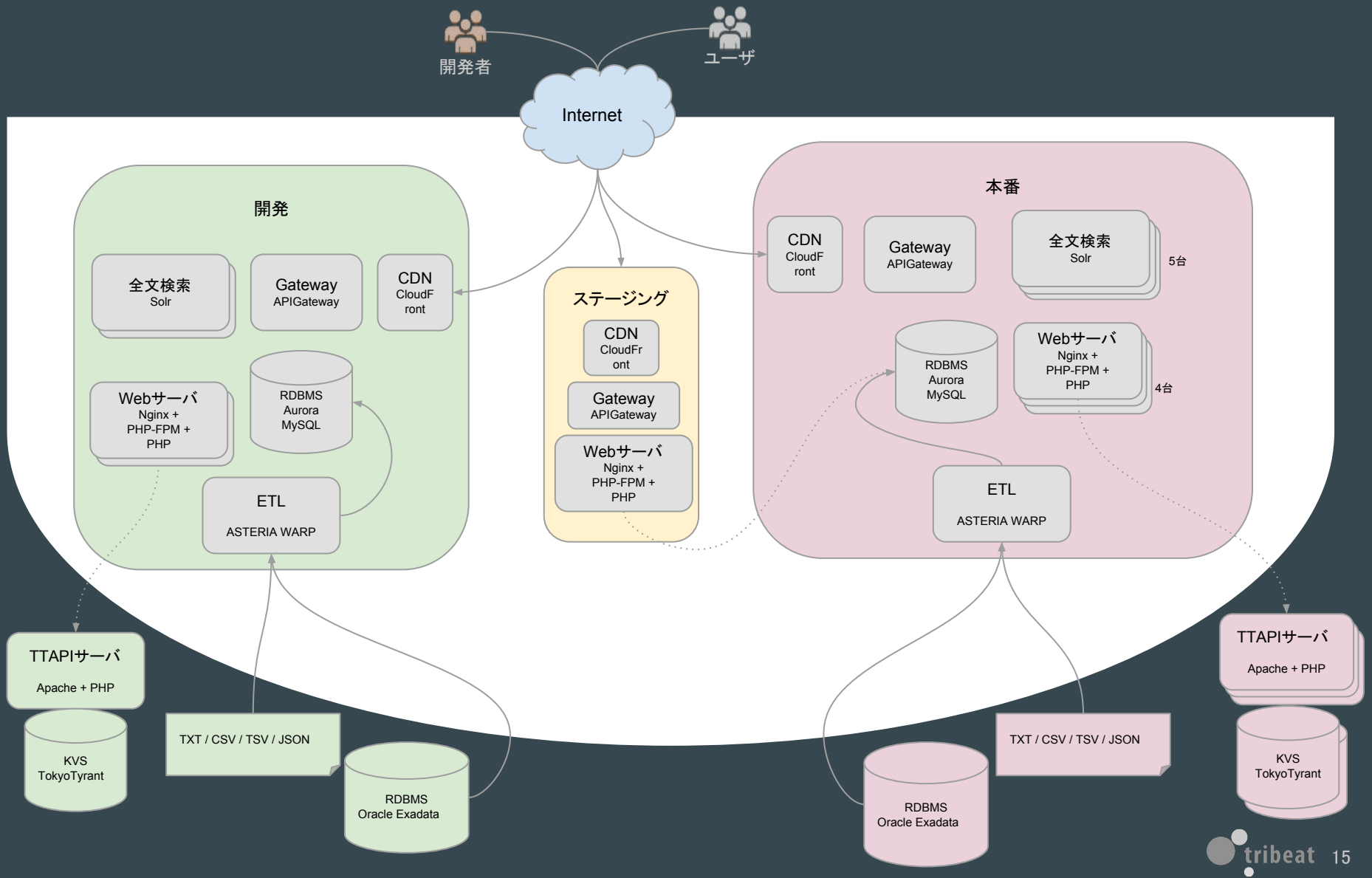


After? - 開発フロー図

項番	内容	ツール
1	開発スタート	
2	feature/Backlog課題キーでブランチを切る ローカル開発環境はDockerコンテナ	
3	開発者がローカル環境でユニットテスト手動実行	
4	改修内容をコミット	
5	CIでソースコードの静的解析と自動修正	
6	コードレビュー、2/5人がOKならdevelopブランチにマージ	
7	OKなら開発環境へマージし、デプロイ	
8	開発環境上で自動でユニットテスト & ドキュメント生成	
9	プロジェクト責任者が内容を確認、リリース判定	
10	リリースは基本週一で行っているので、その週にリリース予定の変更のみをチェリーピック or マージ	
11	まとまったリリース分をステージング環境へマージし、デプロイ	
12	開発者及び開発依頼元にてテストを実施	
13	独自の承認のツール(Webアプリ)にExcelのテスト結果を添付してリリース承認を得る	
14	ブルー・グリーンデプロイを採用。リリース予定の環境と既存の本番環境を同時に動かし、リリース予定の環境でテストOKであればLBの向きを変更してリリースを行う	

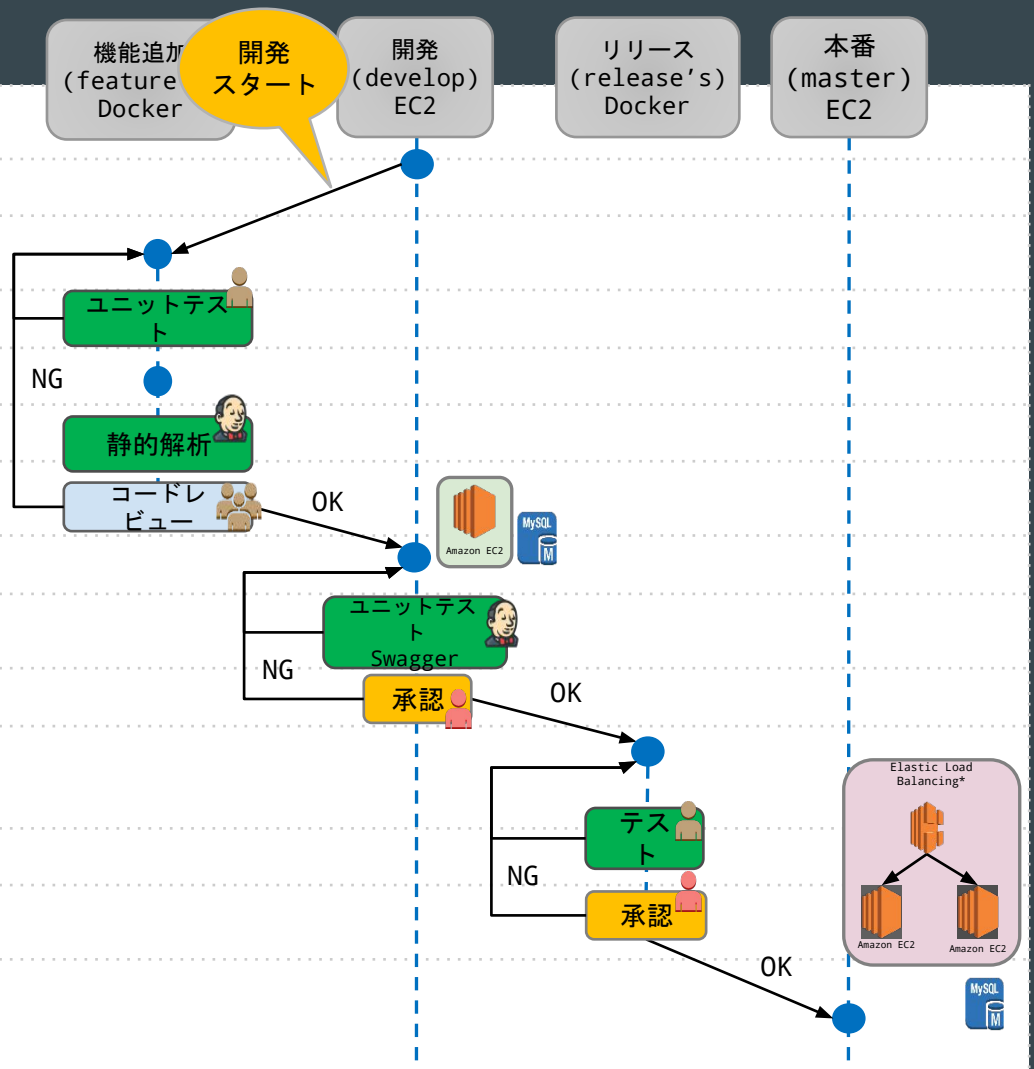


After! - 構成図


開発者
ユーザ
Internet
開発
全文検索
Solr
Gateway
APIGateway
CDN
CloudF
ront
Webサーバ
Nginx +
PHP-FPM +
PHP
RDBMS
Aurora
MySQL
ETL
ASTERIA WARP
ステージング
CDN
CloudFr
ont
Gateway
APIGateway
Webサーバ
Nginx +
PHP-FPM +
PHP
本番
CDN
CloudF
ront
Gateway
APIGateway
全文検索
Solr
5台
RDBMS
Aurora
MySQL
Webサーバ
Nginx +
PHP-FPM +
PHP
4台
ETL
ASTERIA WARP
TTAPIサーバ
Apache + PHP
KVS
TokyoTyrant
TXT / CSV / TSV / JSON
RDBMS
Oracle Exadata
TXT / CSV / TSV / JSON
RDBMS
Oracle Exadata
TTAPIサーバ
Apache + PHP
KVS
TokyoTyrant

After! - 開発フロー図

項番	内容	ツール
1	開発スタート	
2	feature/Backlog課題キーでブランチを切る ローカル開発環境はDockerコンテナ	
3	開発者がローカル環境でユニットテスト手動実行	
4	改修内容をコミット	
5	CIでソースコードの静的解析と自動修正	
6	コードレビュー、2/5人がOKならdevelopブランチにマージ	
7	OKなら開発環境へマージし、デプロイ	
8	開発環境上で自動でユニットテスト & ドキュメント生成	
9	プロジェクト責任者が内容を確認、リリース判定	
10	リリースは基本週一で行っているので、その週にリリース予定の変更のみをチェリーピック or マージ	
11	開発者にてテストを実施	
12	独自の承認のツール(Webアプリ)にExcelのテスト結果を添付してリリース承認を得る	
13	ブルー・グリーンデプロイを採用。リリース予定の環境と既存の本番環境を同時に動かし、リリース予定の環境でテストOKであればLBの向きを変更してリリースを行う。	



ツールリスト

最終的に利用したサービス/ツール **(DevOps関連のみ)**

- AWS CodeCommit（バージニア北部）
- AWS Lambda（バージニア北部）
- AWS CodeDeploy（東京リージョン）
- AWS CloudWatch（東京リージョン）
- AWS Elastic Load Balancing（東京リージョン）
- AWS Auto Scaling（東京リージョン）
- AWS Simple Notification Service（東京リージョン）
- AWS S3（東京リージョン）
- Jenkins x2（開発用、インフラ用）
- Ansible
- Swagger
- PHP Code Snifer
- PHPUnit

反省点

振り返り ~其の壱~

- CIを有効活用する設計/実装がローンチまでにできなかった
  - 開発当時UnitTestの**設計が悪かった**ため実行時間が長くデプロイに支障が出るため**封印**
    - UnitTestの開発コスト及び実行時間短縮はローンチ後に実施
    - UnitTestを分解してデプロイ時に実施するもの、定期的に夜中に実施するものなどわける などできたはず
  - 情報共有という観点でのCIツールの実装やメール通知ができなかった
    - メール内容に本番・ステージング・開発の情報が混じっている
    - メールの本文のリンク先で環境に応じて表示する情報の切替えが必要である

反省点

振り返り ~其の弐~

- Gitのdevelopブランチが汚れちゃった
  - とりあえずオーダーがあって実装したけどリリースしなかったものをmergeしたままに
  - merge時にSquashするルールとしなかったために切り戻しやcherry-pickが大変
- デプロイが高頻度でコケる
  - CodeDeployが中々の確率でエラー終了することが未解決
  - エラーの内容からAMI内のCodeDeployエージェントが古いと思われる
  - AMI作成はインフラチームの領域なのですが半年以上放置されています

反省点

振り返り ~其の参~

開発者のエゴかもしれないですが誤解を恐れずに言うと

- インフラチームの超々々保守的な スタンス/スタイル の打破が結局 できなかった
  - 結局「 自分たちのシステム 」という 認識を持ってもらえなかった
  - 従来通りオペレータ という立場でしかなく、依頼によって手を動かすポジションの 継続
  - 障害対応に関しても、「障害時にも対応するという仕事」だから動いている感
  - インフラの 作業を少なくする のは 開発チームの仕事 という認識を持たれている感
  - と、いう愚痴 ^^;)

まとめ

結局のところ

- 理想は何か？ 何が正しいのか？を考えること
- なぜクラウドコンピューティングが生まれたのか、また流行っているのか？
  - そこには理由が必ずあるはずで、それを考えること
- 知識・知見の不足によってモダンな設計ができない・しようとしない
  - 自分たちの分野、またその周辺の技術に関する知識・知見が圧倒的に少ないエンジニアが多い
  - 「新しい技能を身につけさせる、再教育する」リスキル（Reskill）という文化の確立

と、いうことをまず実践していく必要があると感じました。

まとめ

強めの言葉でごちゃごちゃと書かせていただきましたが
個人的に
とにかくDevOpsを実践するために必要なことは
関わるリソースがみんな最低限のラインに立つ努力をする必要があると感じました
専門、生業が違うことは最初からわかっています
その知識、知見にギャップがあるのは当たり前
プロジェクト立ち上げ当初はぶつかるでしょうがそれは至極健全なものだと思います
ただ、ずっとこの状態を良しとするのはダメで
何言っているかわからない イコール 意見が合わない。間違っている。
ではなく
少なくとも相手の言っている言葉がわかる努力をするべきで
それがないといくら片方が歩み寄ろうとしても
お互いにいつまでも相手を尊重することはできないはずです
結果
押し付け合い、お見合いのグレーゾーンは一向になくならないと思います
まずはこのことに自分事として気づいてもらうこと
それができて初めてDevOpsの門の前に立てるのかなと考えています
それくらい まぁ言っちゃえば 簡単ではないのかなと私は思っています

最後に

普段とくに思っていないですし DevOps語るのに言っちゃダメなのはわかっていますが

1回だけ言わせてください。

EOF