ChatWork の
新メッセージングシステムを支える技術

AWS Dev Day Tokyo 2017

加藤潤一 / 大村伸吾

chatwork

自己紹介

加藤潤一

- コアテクノロジー開発室
- Scala / ドメイン駆動設計 / OAuth2
- Falconでは全体設計&アプリケーション担当

大村伸吾

- コアテクノロジー開発室
- Scala / 分散システム / 機械学習
- Falconでは全体設計&インフラ担当

パート1 Falconアーキテクチャ詳解（加藤）

パート2 ChatWorkにおけるDevOps改善（大村）

パート1

Falconアーキテクチャ詳解

アーキテクチャ刷新の背景

- 開発 2010年〜。内製F/Wを用いた社内向けアプリケーション。
  既存システムに相乗り。2013年に公開サービス化

- 商機を逃さないためにひたすら開発の日々。その代償として技術的負債が積み上がった。
  増え続けるデータと負荷は社内システムをベースにしたアーキテクチャでは限界を迎えた

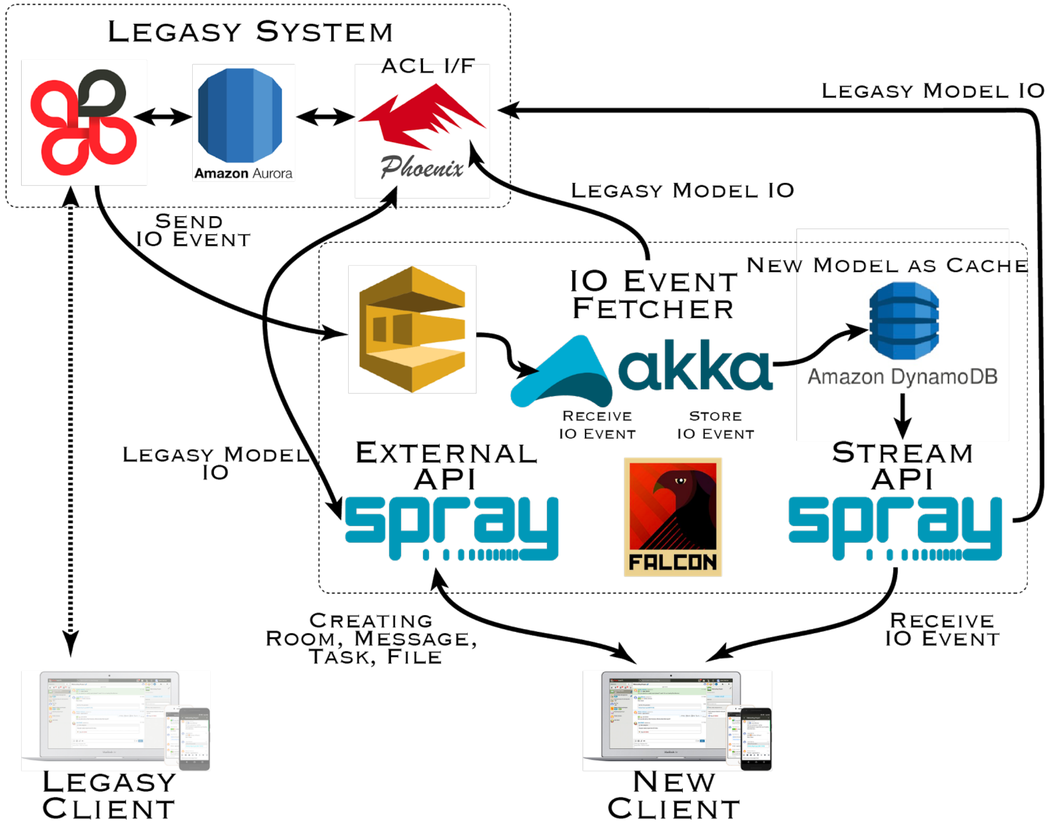
- 総コード量30万行以上。共通処理を行う中心的なクラスは1クラスで1万行

新アーキテクチャへの道のり

- その後 SPoF対策などの改善は継続的に行ったが根本対策はできなかった。

  方針を変えてシステムを刷新することになった

- システム刷新に向けて、最初に**ライブマイグレーションプラン**(右図)を選択した。
  - 既存システムへの影響を最小限にするために
    - 既存システムのコードを極力変更しない
    - 無停止マイグレーション
  - スコープはチャットルームに関するすべての機能(ルーム, メッセージ, タスク, ファイル, コンタクト)



- 一年半ほど開発を続けたが、**様々な問題(プロジェクトマネジメント、スコープ、パフォーマンスなど)**が起き、プロジェクトを再起動した…

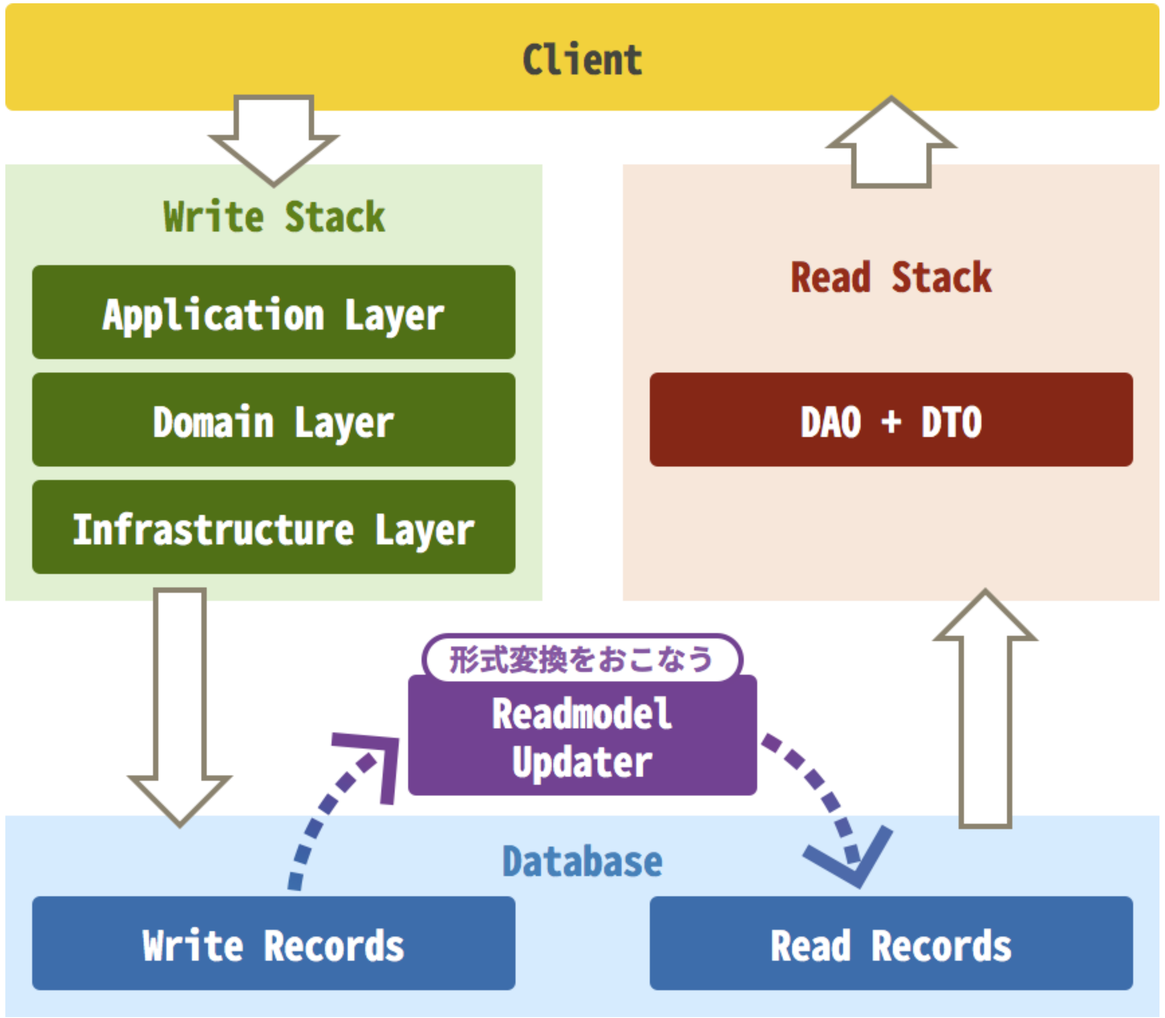
- 根本的に戦略を変えて**新しいアーキテクチャ**に移行することにした
  それから約1年間の開発期間を経て、**大規模なデータ移行**を行い、2016年末に無事リリース

新アーキテクチャの方針

- ビジネスに最も影響の大きいメッセージのみのスコープに絞る
  - メッセージ数の推移は、**2015年 5億件 ➡ 2016年 10億件 ➡ 2017年 20億件**
  - メッセージ数は、指数関数的に急増している
- 保守性を維持するために、ドメイン駆動設計は継続
- リアクティブシステム(Akka)をベースにした CQRS + ESを採用
  - 高スループット・低レイテンシを実現。
    現行システムの2倍程度の同時接続数とスループットを実現すること
  - 可能な限り障害に対して自己回復力を持つ
- 低コストであること(システム規模とコストの相関が線形未満)
- **コンセプト検証(POC)を必須とする**
  クネビンフレームワークの複雑な課題への対応(探査 ➡ 知覚 ➡ 対応のアジャイルプロセス)

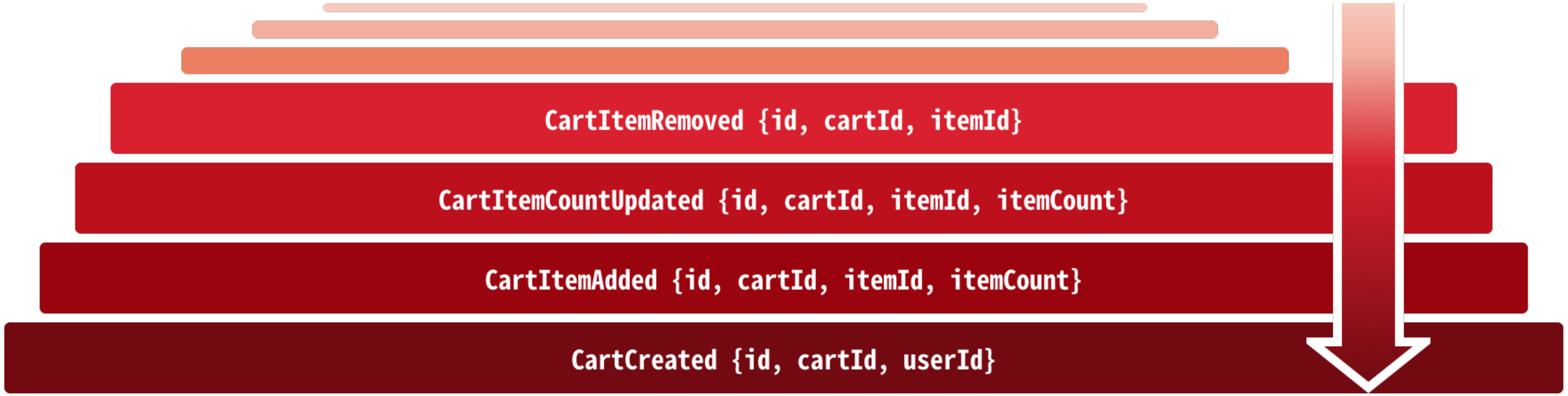
CQRSとは

- Command and Query Responsibility Segregation
  - コマンド・クエリ責務分離
  - 2010年 Greg Young氏によって考案されたアーキテクチャパターン
- Command-Query Separation = CQS
  - コマンドクエリ分離原則
  - 「あらゆるメソッドは、**アクションを実行するコマンド**か、**呼び出し元にデータを返すクエリ**かのいずれかであって、**両方を行ってはならない**。これは、**質問することで回答を変化させてはならない**ということだ」
  - 1997年 Bertrand Meyer氏によって考案された設計原則
- CQSをアーキテクチャに適用したのがCQRS

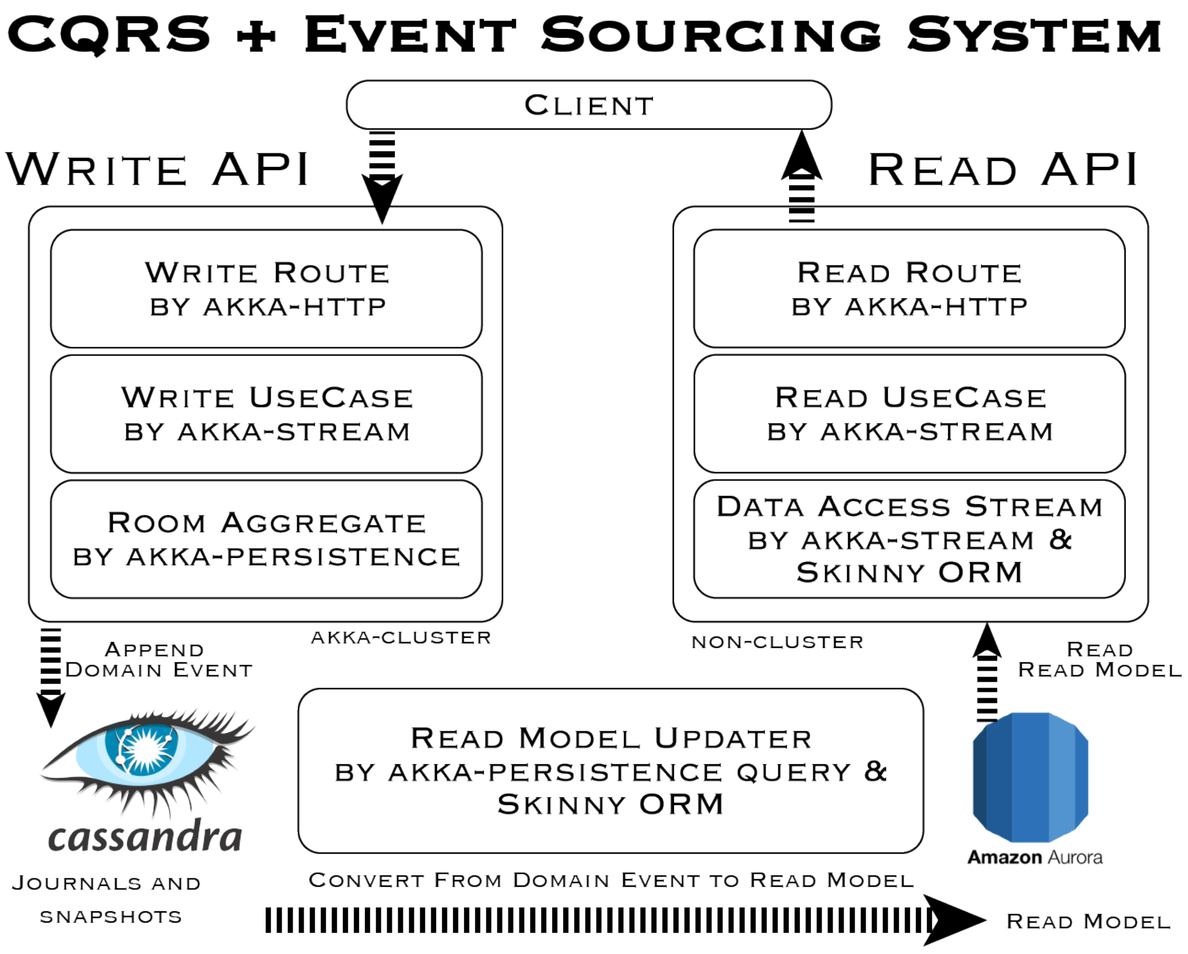


Event Sourcingとは

- Greg Young氏の考案
- Event Sourcing(以下 ES)とは、データではなく何かしらの出来事＝ドメインイベントをモデリングの主役にすること
- ドメインモデルをデータとして格納するのではなく、発生するドメインイベントのすべてを永続化する(原則的に追加書き込み)。モデルの状態は、イベントの列を再生することで得られる
- ESは、CQRSを劇的に改良する



初期のCQRS+ESモデル(POC時)



Write API

- ドメイン駆動設計
- ドメインイベントをストレージに書き込む

Read Model Updater

- ドメインイベントをリードモデルに変換

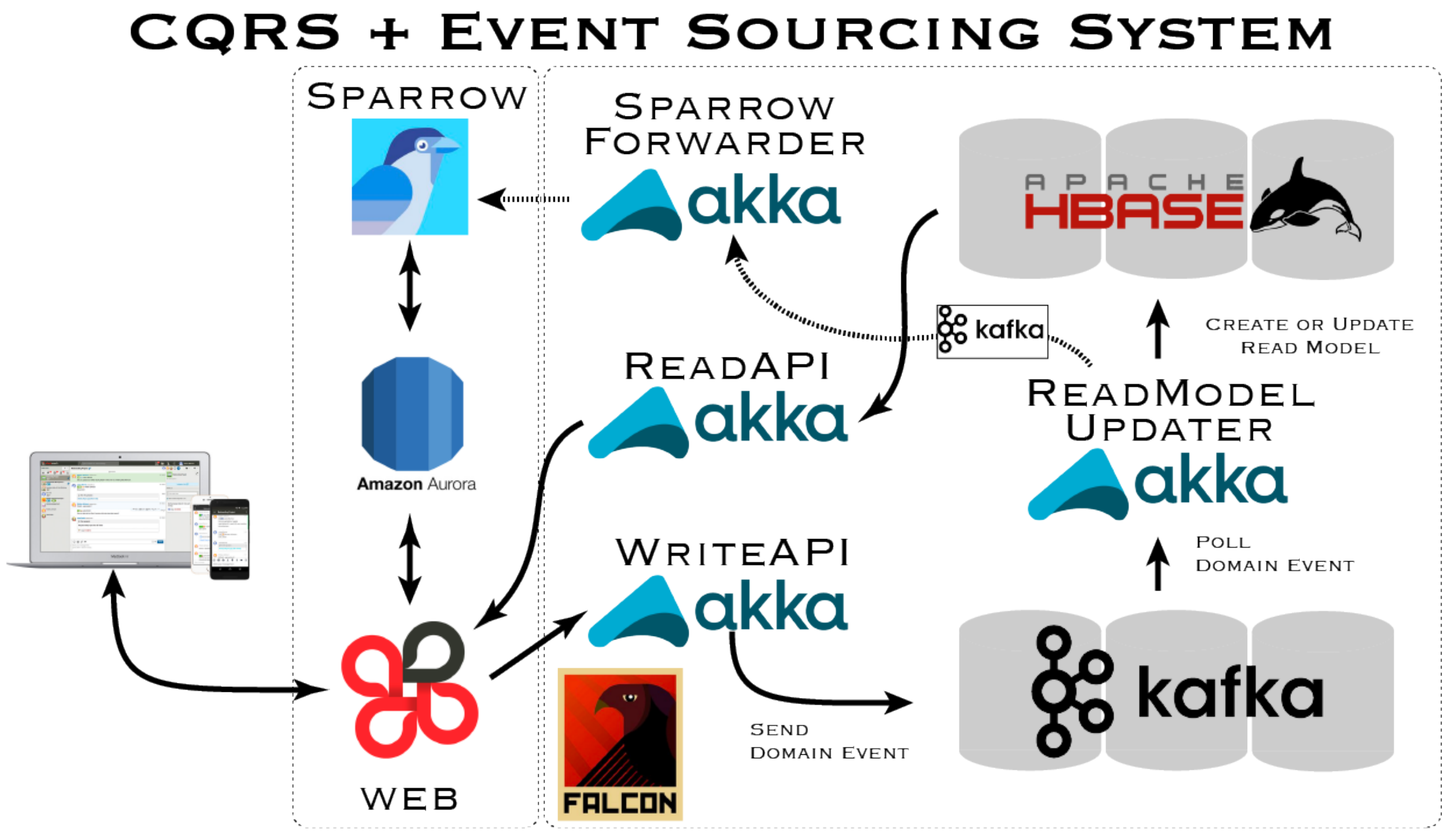
Read API

- クエリ駆動設計

検証結果

- インスタンス数とスループットの関係は、ほとんど線形でスケールアウトすることがわかった

最終的なアーキテクチャ



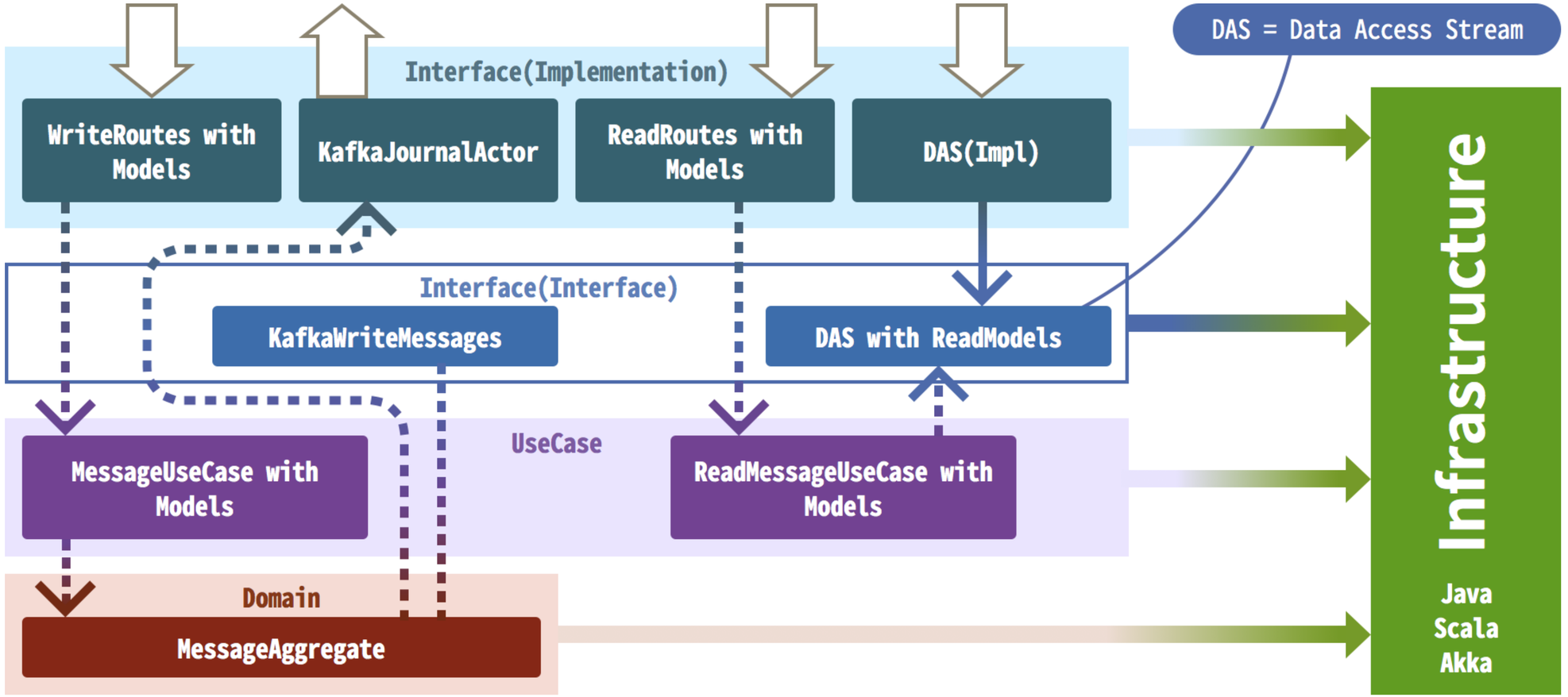
- POCの結果を踏まえて、CQRS+ESとリアクティブシステムをベースにした本番想定のアーキテクチャを構築した
- Falconは、既存システムのためのメッセージングバックエンドサービス
- 想定どおり、ローコストでハイパフォーマンスなアーキテクチャを実現

コンポーネント構成

分類	コンポーネント	ミドルウェア/スタック	概要
ストレージ	Write DB	Kafka	パーティションが1Room内のイベントの連続性を保証する。
	Read DB	HBase	ReadModelを永続化。Rowキーは、”RoomId + MessageId”
アプリケーション	Write API	akka-http, circe, akka-stream, kafka	クライアントからメッセージの投稿・更新を受け付け、 Kafkaにドメインイベントをエンキューする
	Read API	akka-http, circe, hbase	メッセージの取得・検索を受け付け、 HBaseからRead Modelをクライアントを返す。
	Read Model Updater	akka-actor, akka-stream, kafka-stream	Kafkaからドメインイベントをデキューし、HBaseにRead Modelを構築する。 その後、Sparrow Forwarderのために再びKafkaにエンキューする。
	Sparrow Forwarder	akka-actor, kafka	Kafkaからドメインイベントをデキューし、Sparrow APIをコールする。

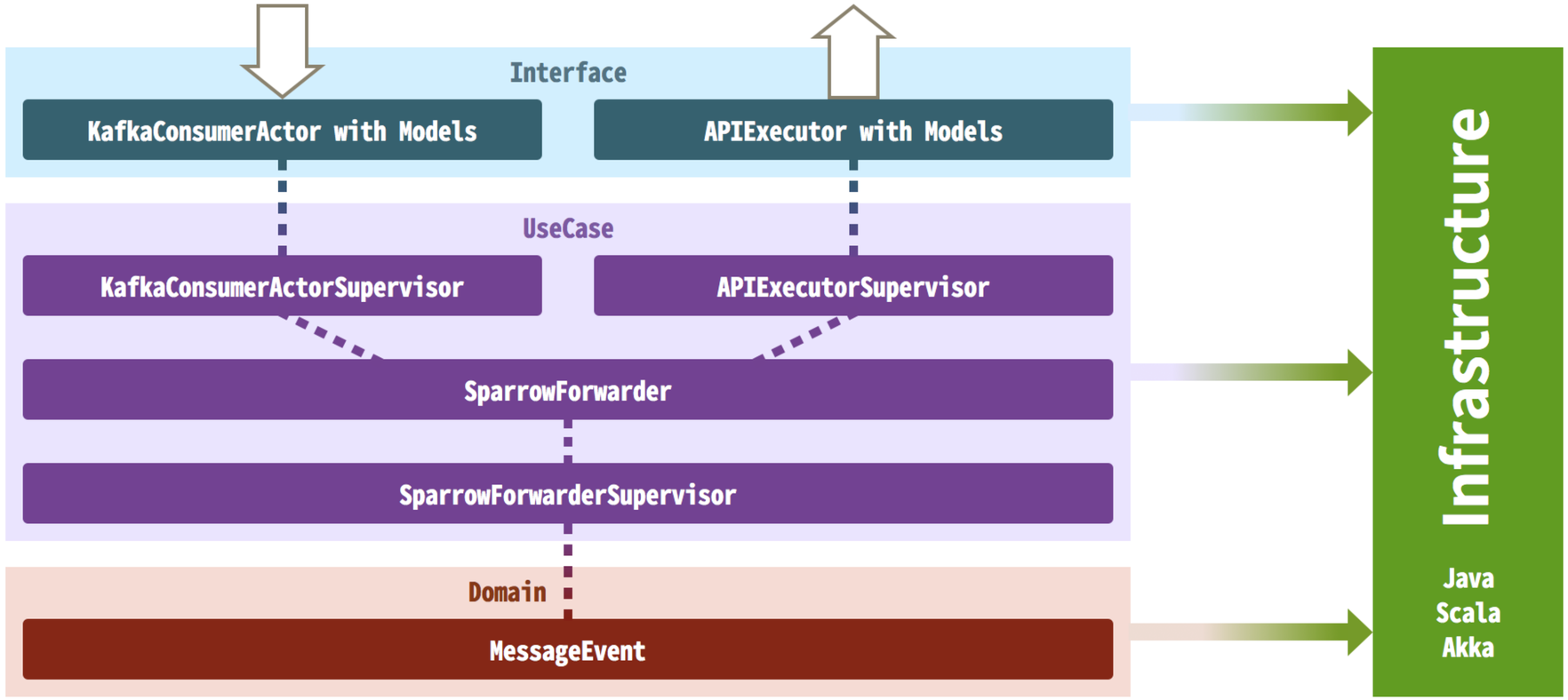
アプリケーションアーキテクチャ

API Serverのレイヤー構造と主要なオブジェクト



ヘキサゴナルアーキテクチャ(DIPを適用したレイヤ化アーキテクチャ)を採用。

SparrowForwarderのレイヤー構造と主要なオブジェクト



- 同一トランザクションの後続処理をPHP側で行うために、必要なコンポーネント
- ReadModelの構築が完了した後に、そのイベントをSparrowに伝える
- ヒエラルキー化されたアクターが相互に連携する

Falconの実績から学ぶ
”障害に強いアクター”を設計する方法

Error Kernel pattern

- In a supervision hierarchy, keep important application state or functionality near the root while delegating risky operations towards the leaves. - Reactive Design Patterns

- スーパービジョンヒエラルキーでは、**危険な作業をヒエラルキーの末端に委譲する**と同時に、**ルート近くに重要なアプリケーション状態や機能を維持する**

- このパターンは、Erlangで数十年にわたって確立されている。
  Jonas Boner氏(LightBend社 CTO)がAkkaを実装するための主なインスピレーションの1つ

出典: Reactive Design Patterns, Jonas Boner, Dr. Roland Kuhn, Brian Hanafee, Jamie Allen

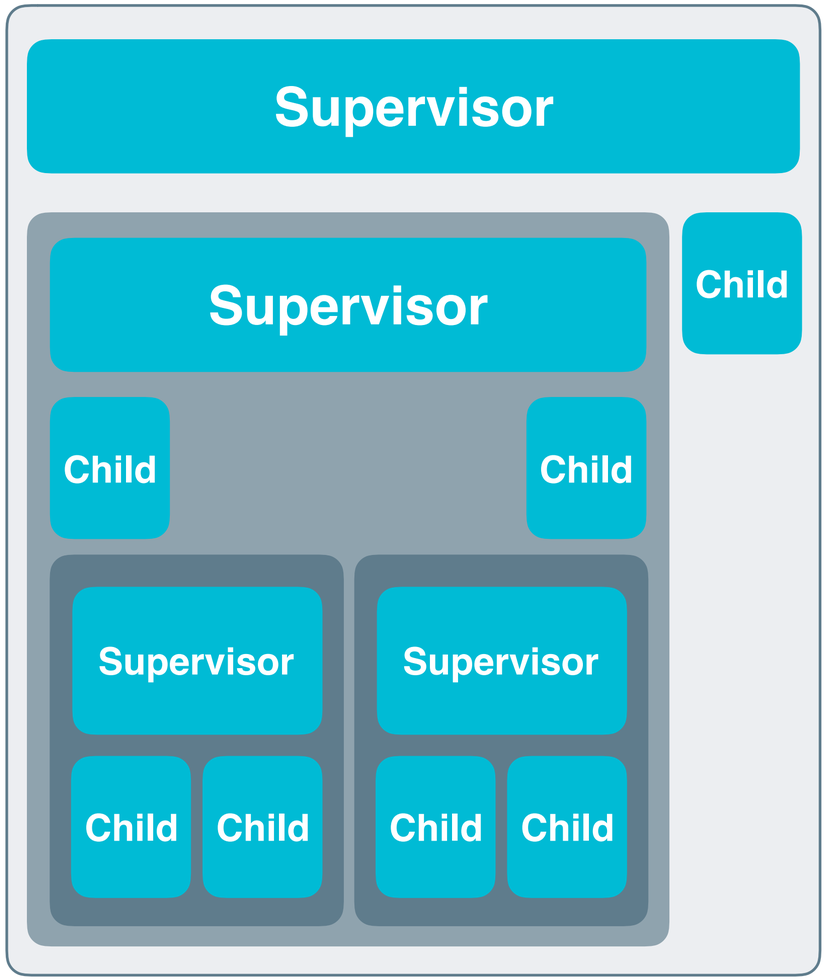
Let-it-crash pattern

- Prefer a full component restart to internal failure handling. - Reactive Design Patterns

- 内部エラーハンドリングよりも、完全なコンポーネント再起動を推奨する。
  - 内部エラーからの回復する機構は、故障した部分を十分に分離できない。
    コンポーネント内部のすべてが障害によって影響を受ける可能性がある
  - **スーパーバイザに障害への対応は委任し、システムの一部を再起動して復旧する**
  - **階層的な再起動ベースの障害処理により、障害モデルを大幅に簡素化できる**。
    また、**予期しない障害に遭遇しても生き残る可能性が増す**

出典: Reactive Design Patterns, Jonas Boner, Dr. Roland Kuhn, Brian Hanafee, Jamie Allen

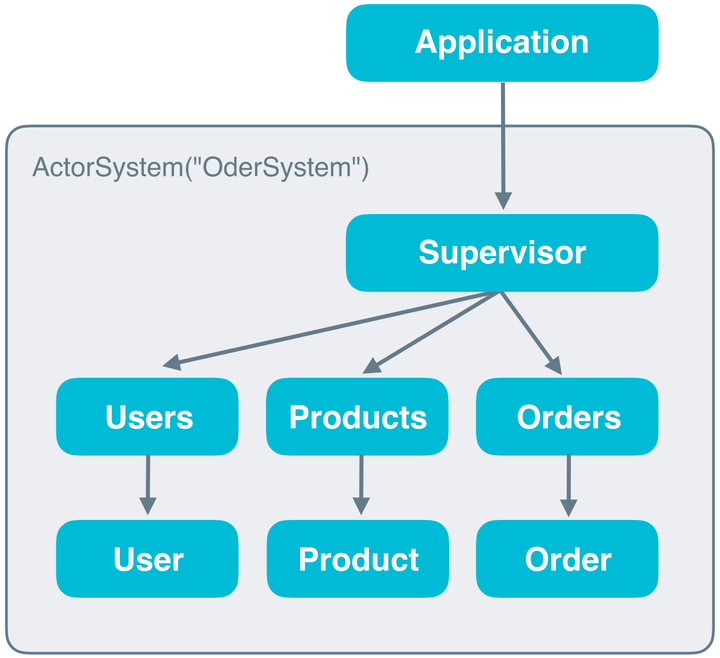
アクターのスーパービジョンヒエラルキー

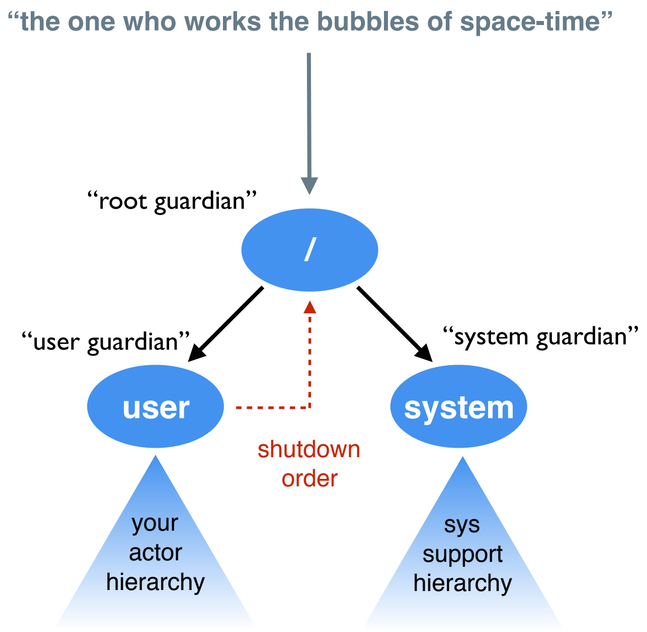


- 親と子のアクターの構造(木構造)がある。親は子の生成や監督を行う
- **親によって作られた子アクターは、スーパーバイザの監督下に置かれる**。子アクターが破棄されると、その子アクターに対する親の監督責任も終了する
- **クラッシュする可能性が高いアクターは、可能な限りヒエラルキーの下層に配置する。下層で起きた障害は、上位までのヒエラルキーが管理・エスカレーションが可能**。障害を起こした子アクターは親の指示によって再起動・停止など行う

出典: Akka in Action, Raymond Roestenburg, Rob Bakker, Rob Williams

Akkaでのスーパービジョンヒエラルキー



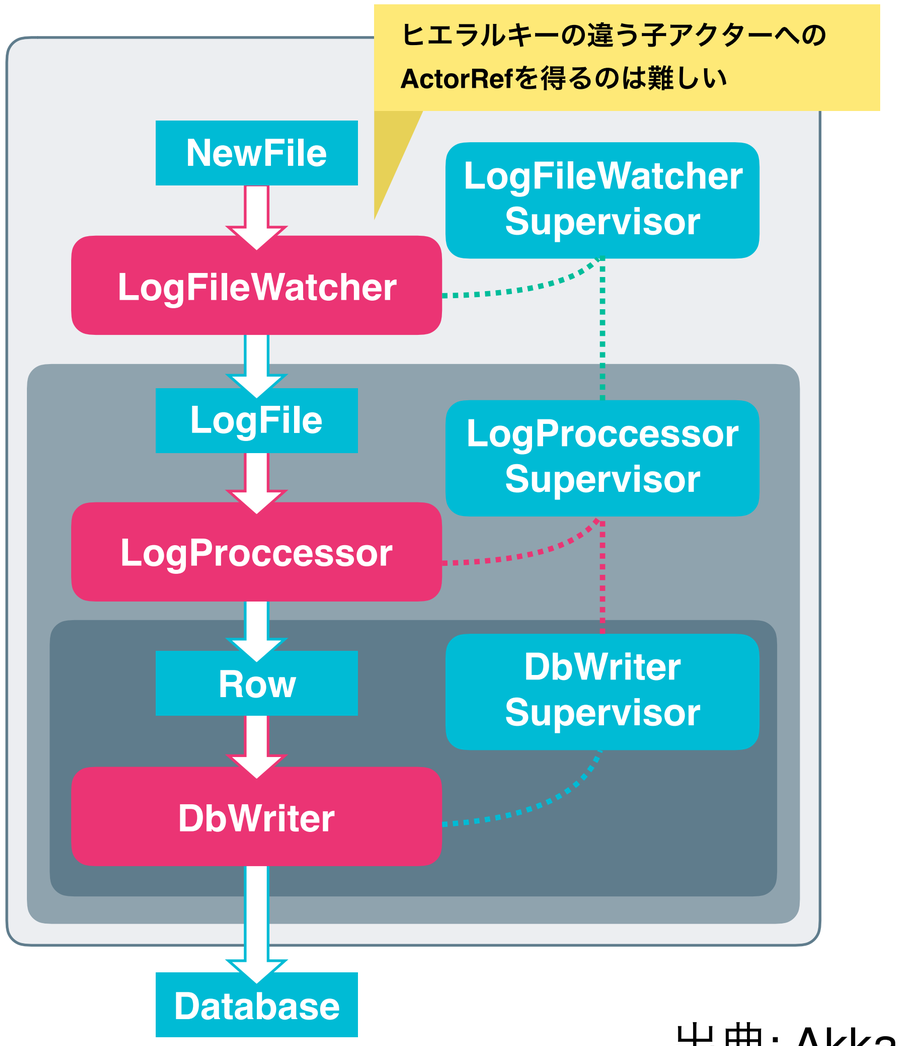


- 最初にActorSystemが作られる。ActorSystemを使って最初のスーパーバイザを作る。スーパーバイザが 子アクターを作ることでヒエラルキーを構築する

- 実際には、アプリケーション用のアクターはuser guardian配下に所属する

出典: akka.io - Supervision and Monitoring

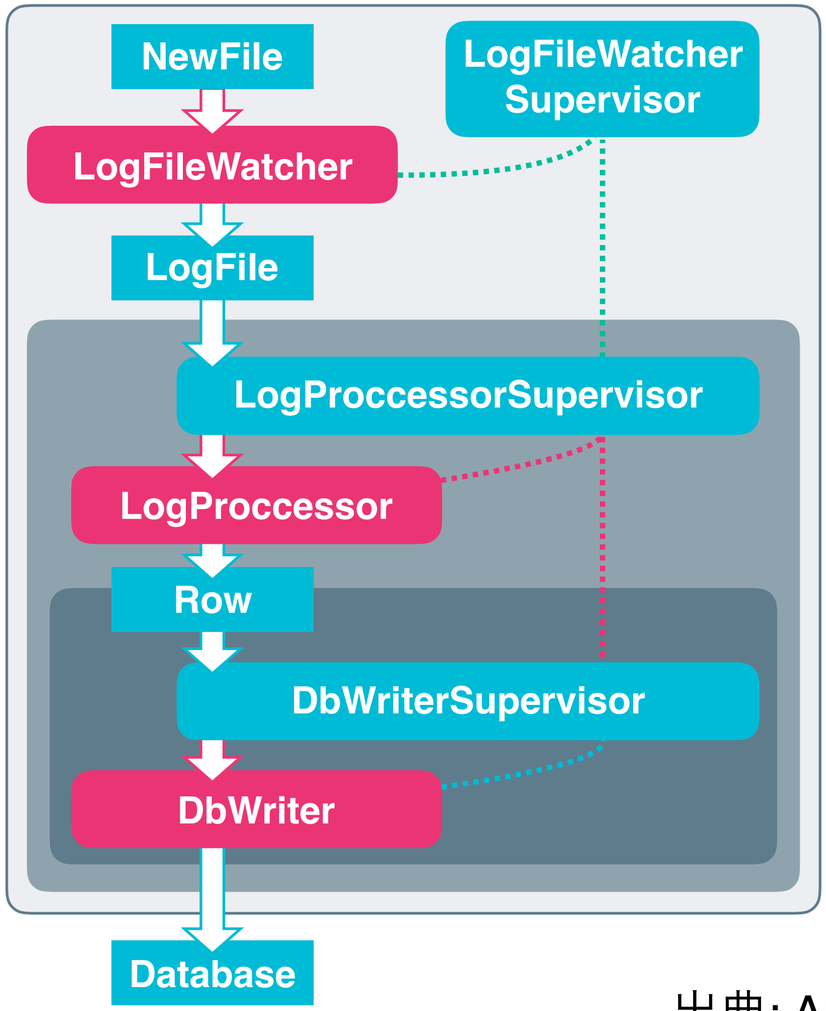
スーパービジョンヒエラルキーの実装パターン1



- ログファイルを監視➡メモリ上にロード➡行に分解➡DBに書き込む例。子アクターが隠蔽されていない実装パターン
- 利点は、各アクターが相互に直接通信すること。スーパーバイザは監督業務とインスタンス作成のみ
- 欠点は、**再起動しか使えないこと**と、**メッセージがデッドレターに送られて失われてしまう可能性があること**。親のスーパーバイザはメッセージフローから分離されてしまう

出典: Akka in Actionより編集, Raymond Roestenburg, Rob Bakker, Rob Williams

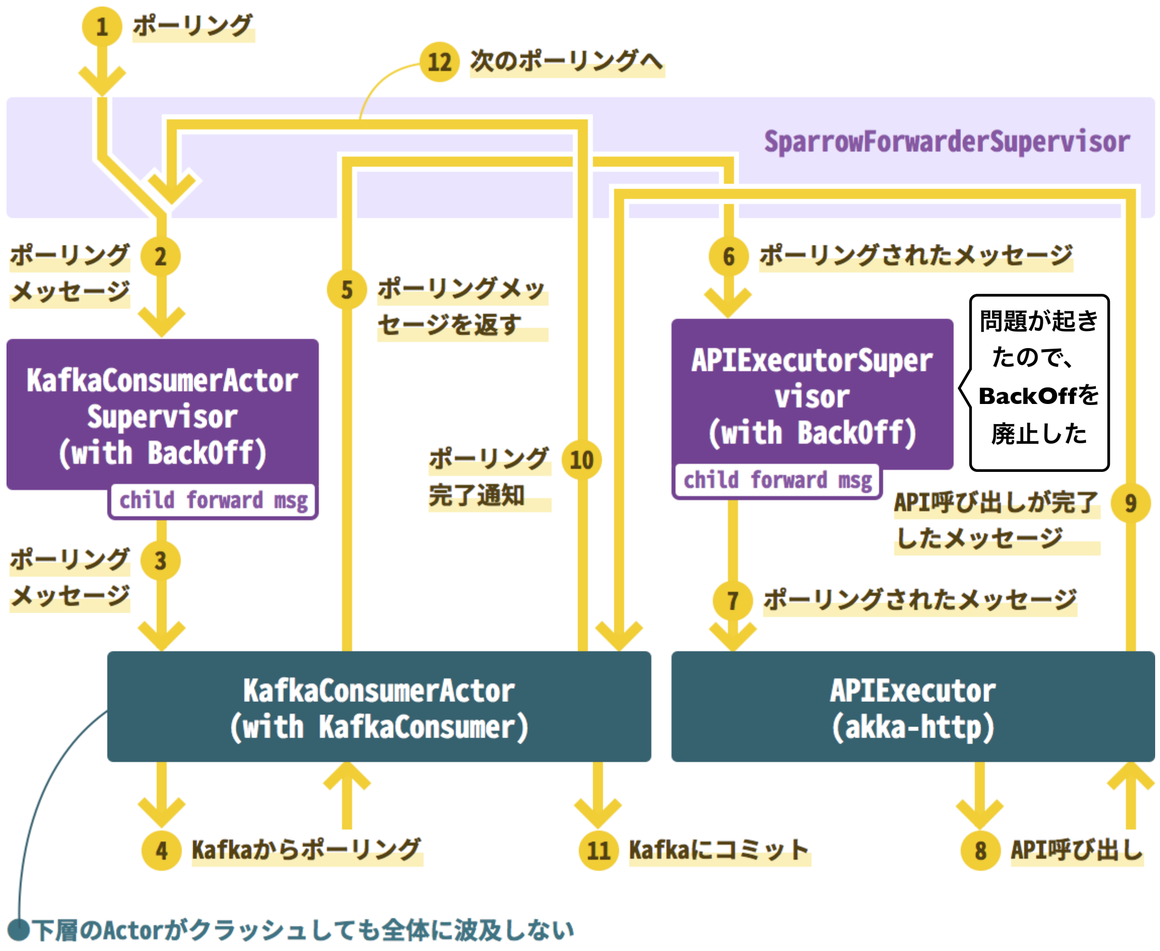
スーパービジョンヒエラルキーの実装パターン2



- 先ほどと同じ処理フローだが、中間にスーパーバイザを挟み子アクターを隠蔽する実装パターン
- スーパーバイザは単なる生成や監督ではなく、**間接参照として、すべてのメッセージを単に透過的にフォワードする**。
  スーパーバイザは子アクターを終了したり、外部のアクターとは無関係に新しいものを生み出したりできる
- 先の例と比べてメッセージフローとヒエラルキーのギャップがない

出典: Akka in Actionより編集, Raymond Roestenburg, Rob Bakker, Rob Williams

Sparrow Forwarderのスーパービジョンヒエラルキー



- 3階層のヒエラルキー。中間のスーパーバイザはExponential BackOffをサポートする
- 下位層のアクターはいつでもクラッシュしてもよい。子アクターで例外が発生すると再起動を命じる。BackOff後にプロセスを最初からやり直す
- APIコールも同様にBackOffしていたがKafkaのセッションタイムタイムアウトが発生するため、BackOffしないようにした。全体の処理フローに影響を与えないようにBackOffすることが求められる

Falconアーキテクチャ詳解のサマリ

- アプリケーション開発での成功要因
  - KPI貢献度が高いスコープの選択とPOCの導入
  - リアクティブシステムとCQRS + ESの採用

- 今回のプロジェクトの効果
  - 未知の技術を採用しながらも厳選した検証を踏まえ、リアクティブシステムとCQRS + ESの特徴をあわせ持つ、ハイパフォーマンスで障害に強いシステムを構築することができた

パート2

ChatWorkにおけるDevOps改善

モチベーション

- ChatWork内に大きなメッセージング用のサブシステムを構築する(miroservice的に切り出す)なら
- これまでのインフラ運用課題も解決したい
  - 主な対象：テスト・リリースパイプライン、実行インフラ
- 開発チームが自身でリリースサイクルを周し、運用までできるような状況を作ってDevOpsを加速させたい

困っていたこと

- 現行システムのインフラ構成超概要
  - EC2ベース(php)のサーバ群
  - Jenkinsによるデプロイジョブ群
  - Capistranoを使ったデプロイ
  - Fabricによる設定管理

- 困っていたこと
  - リリース作業が完全に自動化されておらずリリースが大変
  - デプロイフローの改修が大変&CIサーバの運用負荷高い
  - 負荷試験の手間が惜しい(もっとカジュアルにやりたい)

達成できたこと

Falcon開発チームが

負荷試験を通ったアプリコードを維持でき

チームだけで好きなタイミングでリリースし

アラート対応までチームで対応できる

ようになった

行った3つの大きな施策

実行環境をKubernetesに



CIサーバをConcourse CIに


負荷テストツールの自動化

実行環境をKubernetesに

- 採用した主な理由(技術面)
  - コンテナオーケストレーションとしてはデファクト & Production Ready な機能が豊富
  - HelmというKubernetesのパッケージングシステムがある
  - サーバコスト削減見込める
  - 開発がものすごく活発
  - kube-aws のプライマリメンテナ @mumoshu さんが社内にいる
  - ローカル開発環境minikubeがあってカンタンにローカルで起動できる

実行環境をKubernetesに

- 採用した主な理由(運用面)
  - インフラチームと開発チームの責務が分離できる
  - インフラチームはKubenetes運用に専念
    - 小さく保てる
  - 開発チームは自分達でリリースを掌握できる

Kubernetesがもたらした責務の分離

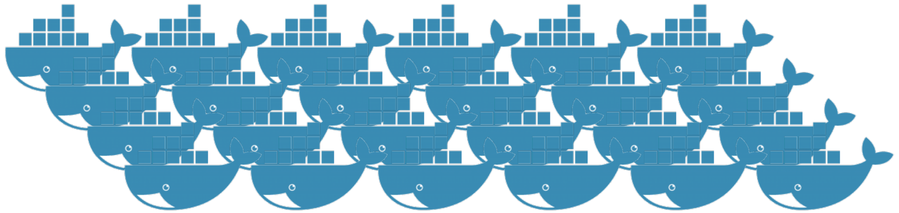
アプリ + インフラ

- 必要なAWSリソースはterraform管理
  - アプリ: PR 作成
  - インフラ: レビュー & 適用

etc.

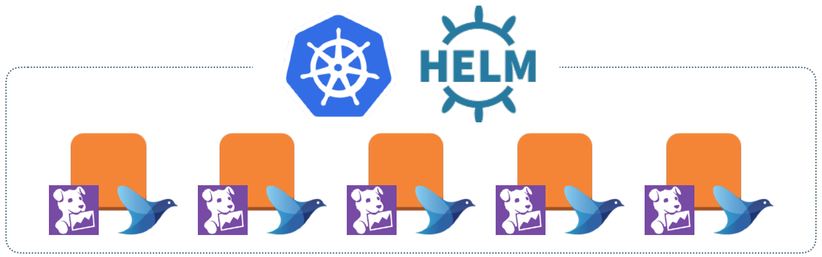
アプリ

- Kubernetes/Helm APIをでいつでもデプロイ(開発者 ≒ 運用者)
- アラート対応もチーム自身で処理可能



インフラ

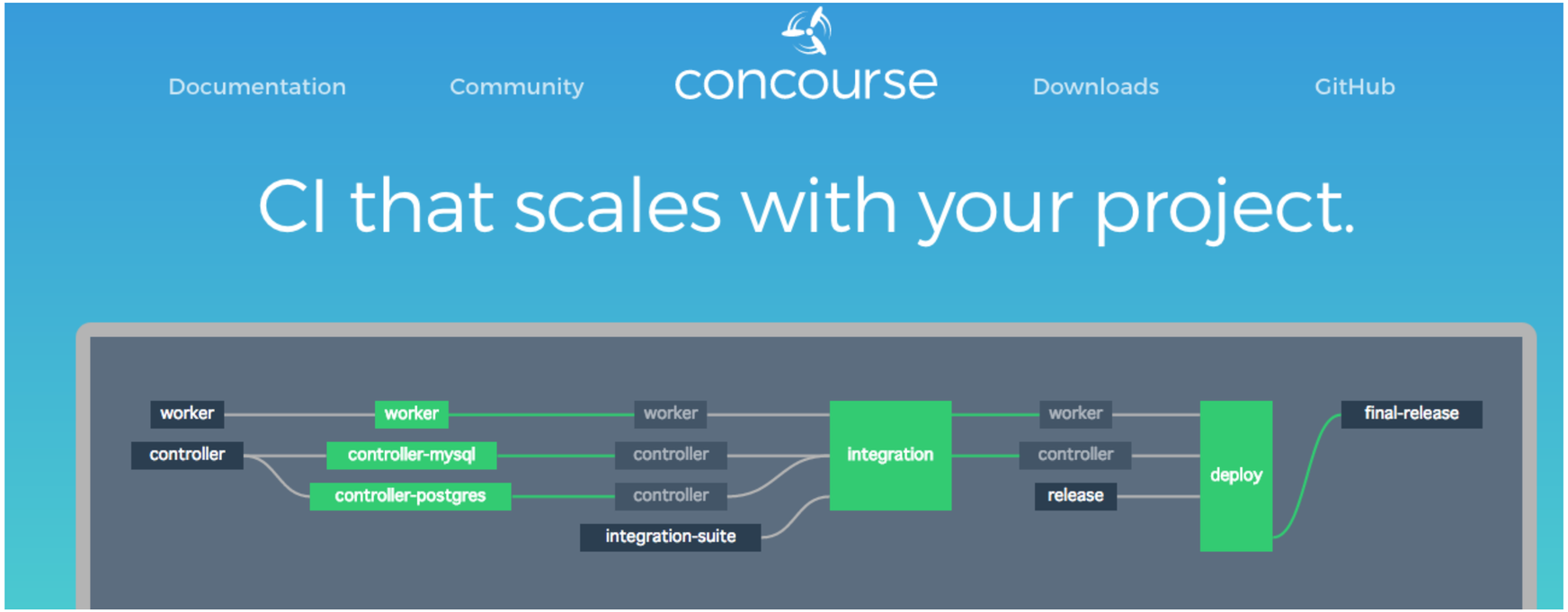
- 安定したインフラ提供にフォーカス
  - kubernetes API, Helm API (Tiller)
  - ログ収集基盤, メトリクス収集基盤
- kube-aws を使ってkubernetesをデプロイ



CIサーバにConcourse CIを採用

- Concourse CIの特徴
- Consourse CIを導入してよかったこと
- FalconのCI/CDパイプライン

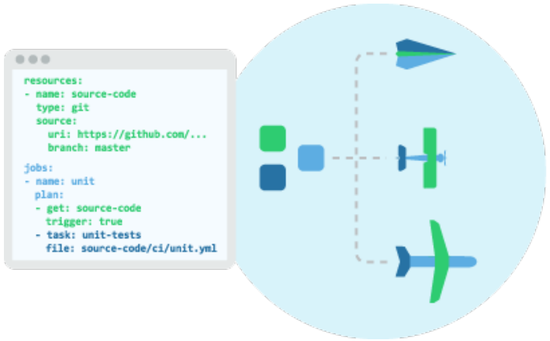
Concourse CIとは



https://concourse.ci

Concourse CIの特徴とメリット

Simple and Scalable

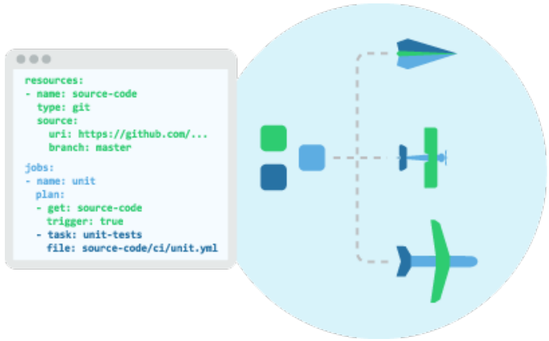


Dependable Results

Integration

Streamlined-UI

Simple & Scalable (1/4)



✓パイプラインが一級市民

✓定義は全てYAML

➡ **Pipeline as Codes可能**

Workerはインスタンス追加するだけ

reference: https://concourse.ci

Dependable Results (Reproducible Pipeline) (2/4)

✓ パイプラインはステートレス

➡ 実行に再現性がある

➡ ローカル開発したパイプラインを本番CIサーバにそのままデプロイ

✓ Taskの実行環境がコンテナで分離されている

➡ Concourseサーバにplugin必要ないので管理が楽

➡ 独自imageで実行環境自由自在

reference: https://concourse.ci

Integration, Streamlined-UI (3,4/4)

Integration

✓ **拡張可能なResource**

➡ **ChatWork通知も容易に独自拡張**

- マルチテナント対応
- 柔軟な認証方法
  - Basic認証, OAuth(Github etc.)

Streamlined-UI

- ビルドパイプラインの状態を
  リアルタイムで監視可能なUI
- 入力テキストボックスの無いUI

reference: https://concourse.ci

Concourse CIにして特によかったこと(1/2)

✓CIサーバの運用が非常に楽になった

- 基本的にパイプラインの定義に全てが記述される
  - taskやresourceの定義もcontainer image
- サーバはpipeline駆動するだけでよいのでpluginの構成管理が必要ない
- Databaseの運用は必要だが、保存データは主にパイプライン定義やビルド履歴のみ
  - RDSを使えば主な運用タスクはカバーしてもらえるので負担が少ない
- Production Readyなデプロイは少し注意が必要
  - 公式ではBOSHが推奨
  - @mumoshuさんと一緒にconcourse-awsというterraformのthin wrapperを開発&公開中

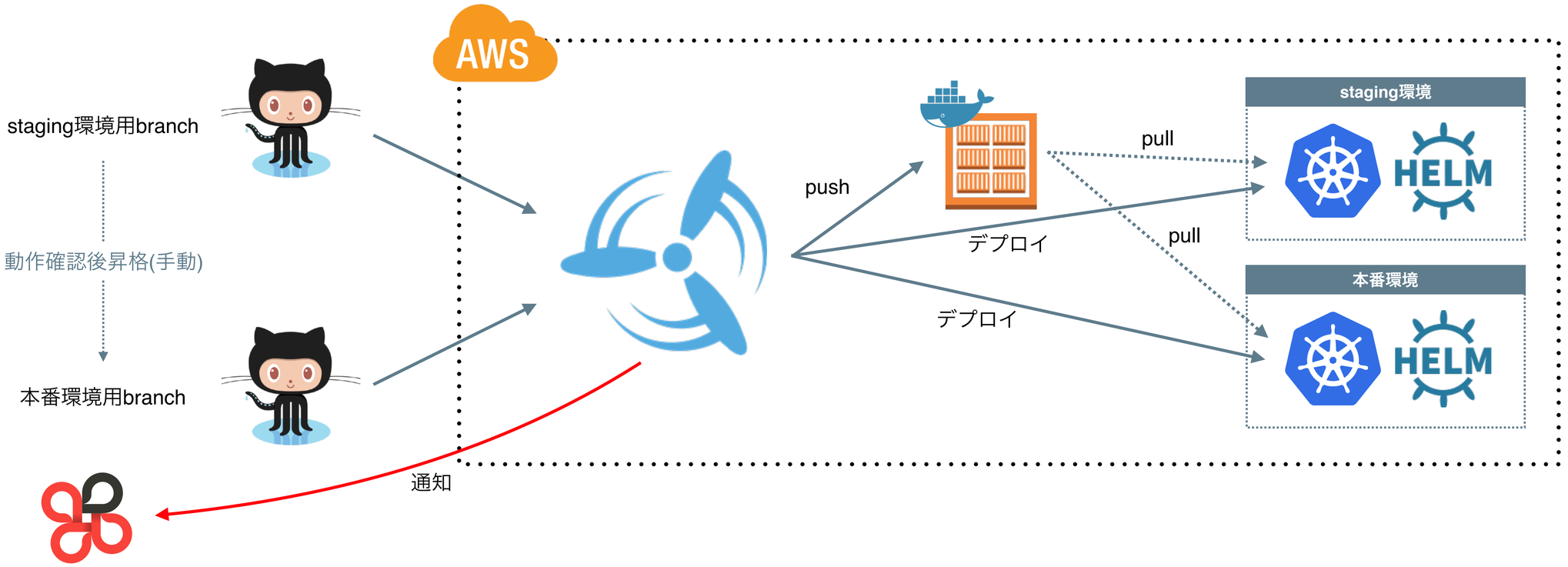
Concourse CIにして特によかったこと(2/2)

✓ローカル開発したパイプラインがそのまま本番にデプロイできる

- パイプラインはReproducibleなので、外部(git, docker registry, kubernetes等)に状態を準備すればどこでも再現が可能

- ローカル開発環境も充実
  - concourse は ‘vagrant up’でカンタンにローカル環境を起動可能
  - kubernetes も minikube というローカル開発環境がある

FalconのCI/CDパイプライン



- Gitlab flow with Environment Branches を採用
  - 環境毎のブランチを作って昇格させるモデル
- 通知は chatwork-notification-resource を実装

負荷テストツールの自動化

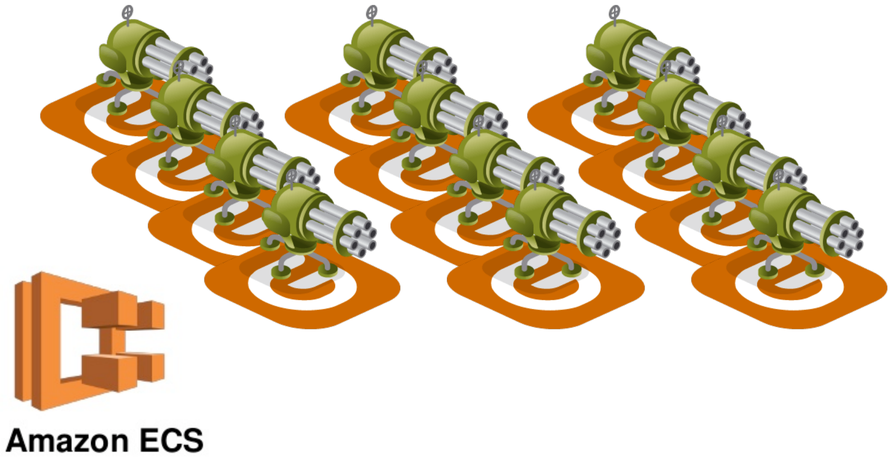
- モチベーション**(避けたい)**
  - **開発大詰めで負荷テストでパフォーマンスが出ない**

- じゃぁどうする？
  - **これまで以上にカジュアルに負荷テストをかけられる環境を作りたい**
    - FullBok利用することで構築は自動化出来ていたが実行自動化がマダだった
    - 呼び出しに依存関係があるようなシナリオをGUIで作成するのが手間だった

負荷テストツールの自動化

- どうやる？（どうやった)
  - **Amazon ECS + Gatlingを使った負荷テスト自動化ツールの導入**

- どうなった？
  - **負荷シナリオがコンテナ化されて負荷実行まで自動化**
  - **負荷テストを通ったコードの維持が楽になった**

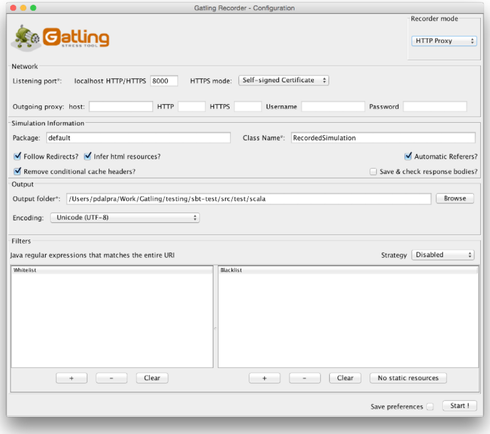


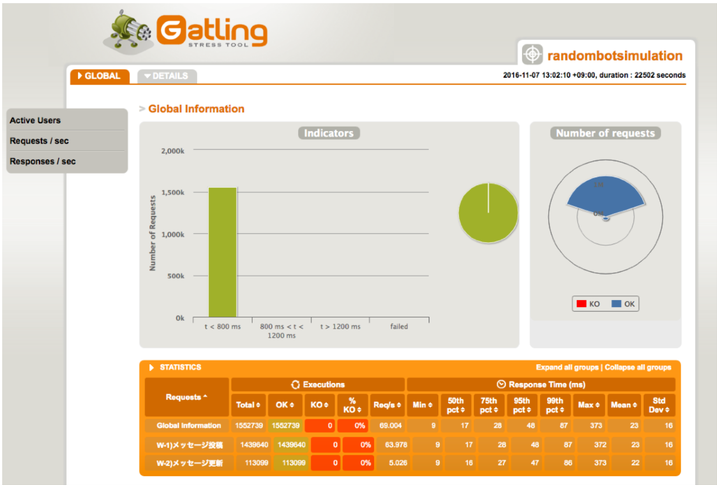
Gatlingとは

- Scala/Akkaで実装された負荷テストツール
- シナリオベース
- 複数エンドポイント対応
- DSLでシナリオがコードで書ける
- GUIからもシナリオつくれる
- InteractiveなHTMLレポート

```
object Search {

  val search = exec(http("Home")
    .get("/"))
    .pause(7)
    .exec(http("Search")
    .get("/computers?f=macbook"))
```



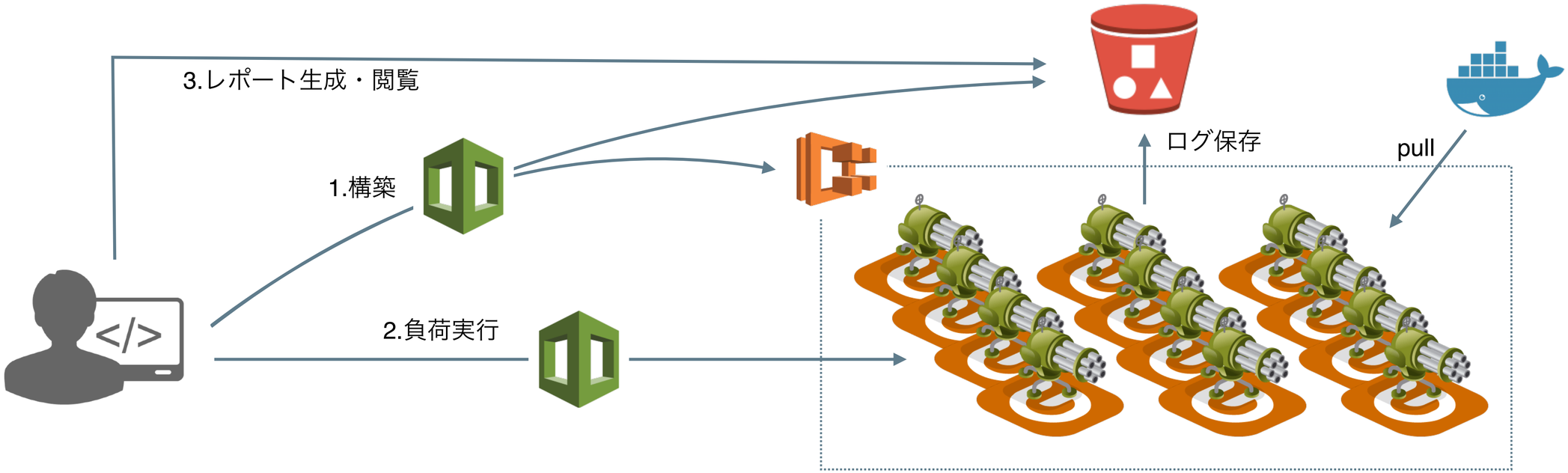


Gatlingとは

- モチベーション**(⬅この部分のサポートツールを実装)**
  - クラスタ実行
    - 複数インスタンスで実行すればなんとかできる
  - 複数レポートのaggregation
  - ただし
    - Gatling Frontline という商用版あり
    - scale-out cookbook も参考になる

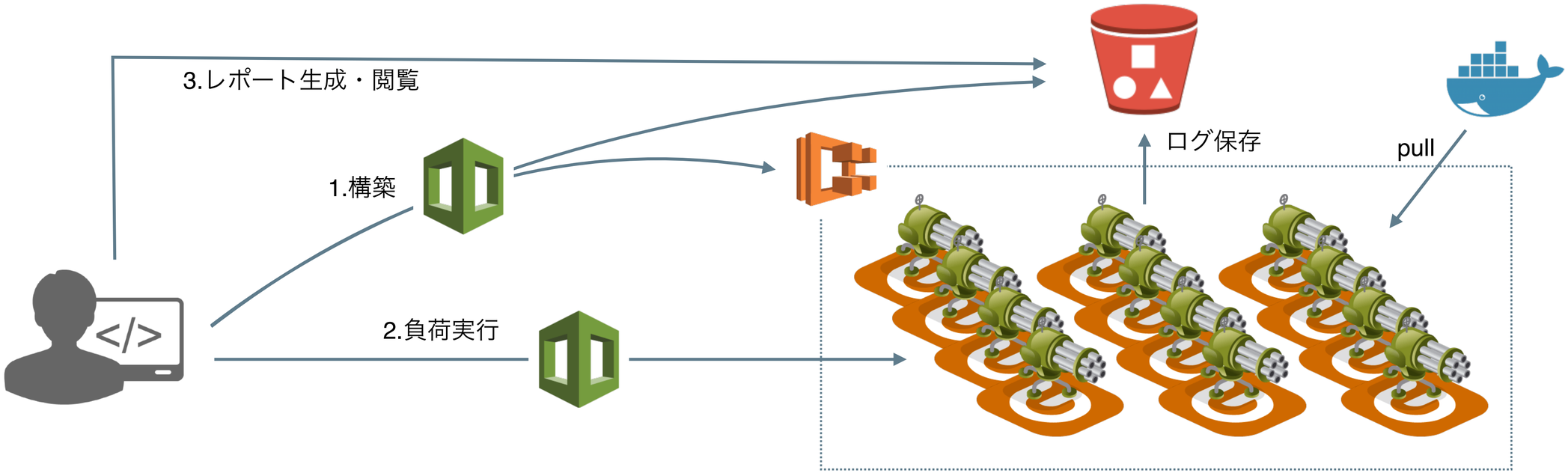
ECS + Gatlingをつかった負荷テストツール

- CloudFormationテンプレート
  - ECSクラスタ+S3バケット，ECSタスク
- Gatling LogをS3へ書き出すためのベースコンテナイメージ
- S3のログから集約レポートを生成するツール



ECS + Gatlingをつかった負荷テストツール

- 負荷シナリオをコンテナイメージ化しておけば幾つかコマンドを叩くだけで負荷実行からレポート作成まで自動化
- 現段階では内部向けツールなので、将来的にOSSとして公開予定



DevOps改善サマリ

実行環境Kubernetes&Helm化、CIサーバConcourse CI化、
負荷試験自動化ツール導入によって

Falcon開発チームが

負荷試験を通ったアプリコードを

開発チームだけで、自身の好きなタイミングでリリース

Falconシステムのアラート対応までチームで対応できる

ようになった

今後

- このプラクティスを既存ChatWorkチームに拡充
- インフラチームをできるだけ小さく保ち続けたい

ご清聴ありがとうございました。

We’re Hiring!!

http://corp.chatwork.com/ja/recruit/

chatwork