Summer Internship 2021
10 Day Tech: Infrastructure

技術部 SRE グループ @mozamimy

cookpad

本日(インフラパート)の流れ

cookpad

本日(インフラパート)の流れ

- 講義パート
- ハンズオンパート
  - Day 3 までに触ってきた minimart-api と minifinancier を AWS にデプロイします
  - 今朝 Slack の DM で送った AWS のクレデンシャルと SSH 用の秘密鍵を使います

インフラパートのテーマ

- 現在のクックパッドのインフラの概観からその変遷をたどります
  - 個々の要素技術の解説は世に溢れている
  - クックパッドではそれらをどう利用しているのか
  - クックパッドのインフラの変遷から開発スタイルがどう変わってきたのか
- ハンズオンではここまで作ったアプリを実際に AWS にデプロイします
  - Terraform や Hako / ECS といったツールを使ってみます

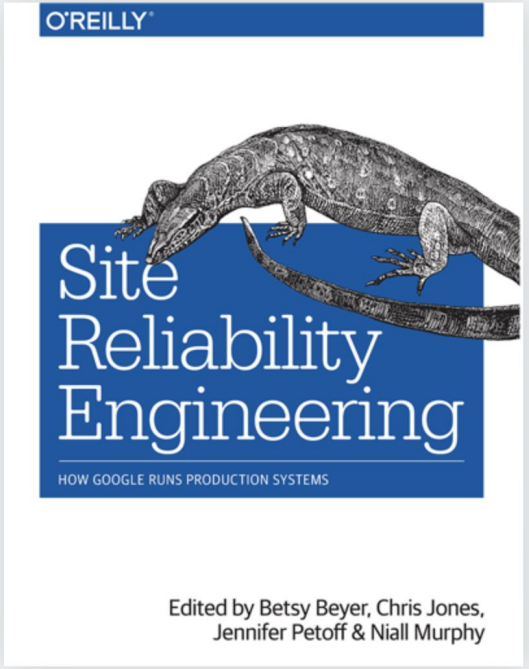
クックパッドにおける SRE とインフラ

cookpad

そもそも SRE とは

- Google が提唱
- ソフトウェアエンジニアリングで信頼性を**コントロール**
- SRE たらしめる方法論
  - SLI/SLO
  - オペレーションの自動化
  - トイルの削減
  - etc...
- 組織によって SRE 像は違う
  - (...と思う)

Site Reliability Engineering



https://sre.google/books/

クックパッドの SRE

価値を提供することに集中

レシピサービス

マート

cookpadLive

共通基盤

Kuroko2

Hako エコシステム

Barbeque

aws 各種インフラ

SREs

サービスの信頼性を担保

開発・デプロイのための仕組みを開発・運用

クックパッドのインフラの特徴

- AWS をフル活用
  - EC2 のようなプリミティブなものからマネージドサービスまで広く
- 巨大モノリシックアプリケーションから
  マイクロサービスアーキテクチャへの移行
- 高いレベルでのコスト可視化と最適化

インフラ運用の主な技術スタック

- **Hako / ECS**
  - アプリケーションを動かすための基盤
  - コンテナオーケストレーション
- **Terraform**
  - AWS リソースの管理
- SAM /CDK
  - サーバレスアーキテクチャを主軸とするアプリケーションでの　AWS リソースの管理

クックパッドの現在のインフラ概観

cookpad

非常にざっくりとした概観

- 図は別資料を参照
- クックパッドではほぼ全てのサービスが AWS 上で動いている

クックパッドマート API サーバの構成

- 図は別資料を参照
- 社内で典型的な Hako アプリケーション
  - ロードバランサ
  - Rails サーバ
  - データベース (MySQL)
  - キャッシュ・KVS (Memcached, Redis)
  - もっとも基本的でわかりやすい構成
- ビルディングブロックをいい感じに組み合わせて目的を達成する
  - EC2 のようなプリミティブなもの
  - Aurora や DynamoDB のように高レベルなもの (マネージドサービス)

Route 53

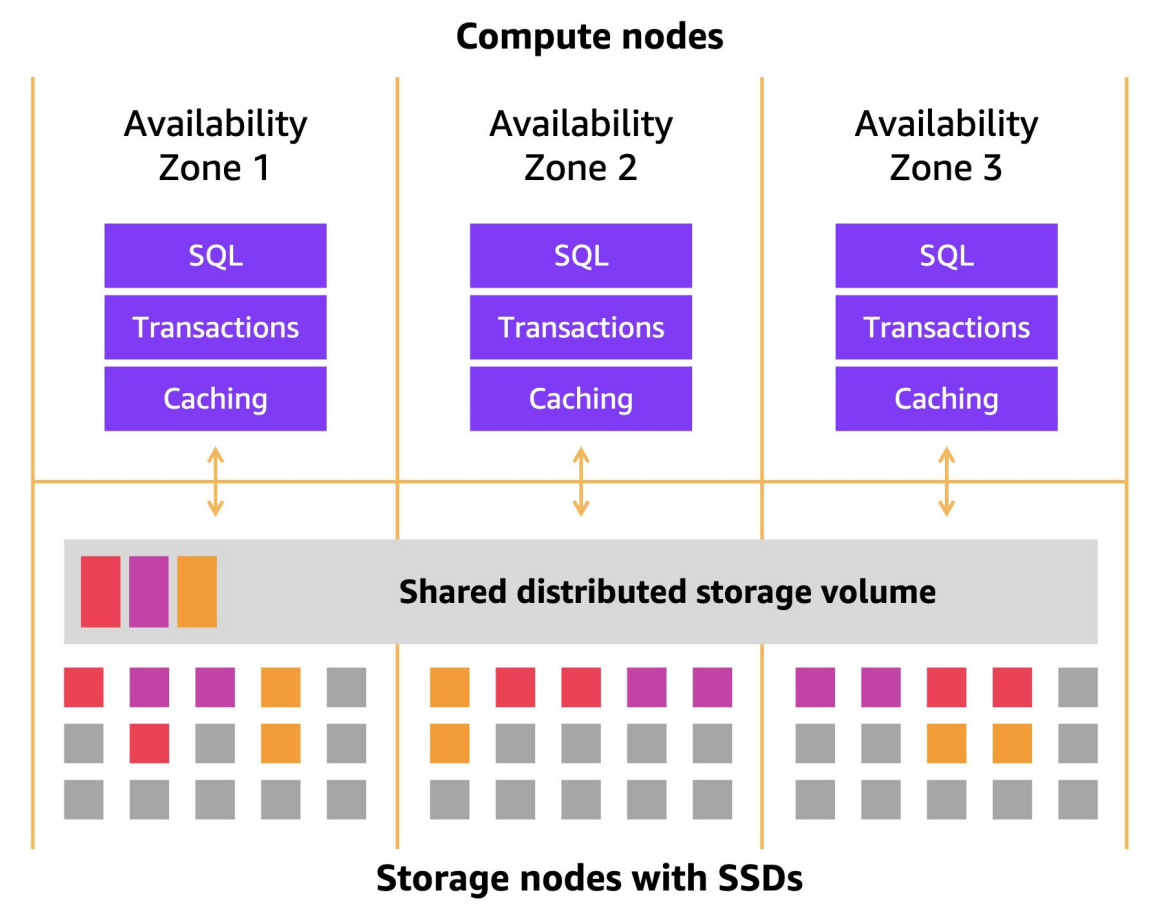
- AWS が提供する DNS サービス
- ドメイン名を IP アドレスに変換する

ALB (ロードバランサ)

- Application Load Balancer
- 負荷分散に特化したマネージド HTTP(S) プロキシ (L7)
- 受け取った HTTP リクエストを配下の EC2 インスタンスに均等に流す
- 負荷に応じていい感じにスケールする
  - ALB の裏側も EC2 インスタンス (らしい)
  - ロードバランサを真面目に自前で作るのは意外に難しい
- HTTP 通信したいときのキホン
  - API Gateway というコンポーネントもあります
- L4 (TCP) レベルでバランスできる Network Load Balancer もある

Amazon Aurora

- AWS が提供するマネージドなデータベースサーバ
- MySQL や PostgreSQL と互換性がある
  - それぞれのプロトコルでおしゃべりできる
  - ストレージエンジン・オプティマイザ・キャッシュ機構などは独自設計
- クックパッドのほとんどのサービスが Aurora を利用
- 高可用性・高耐久性
  - ストレージは 3AZ に分散
  - 計算ノードの故障も高速にフェイルオーバーする



https://pages.awscloud.com/rs/112-TZM-766/images/01_Amazon Aurora %E3%82%A2%E3%83%BC%E3%82%AD%E3%83%86%E3%82%AF%E3%83%81%E3%83%A3%E6%A6%82%E8%A6%81.pdf

ElastiCache

- AWS が提供するマネージドなキャッシュストア
- Redis と Memcached から選べる
- クラスタを単位としてノードを増減しやすい
- Redis Cluster にも 6.x エンジンで対応
- 汎用のキャッシュや Rails のセッションストアとして利用されることが多い

ECS (Elastic Container Service)

- **Docker** を利用した AWS が提供する
  コンテナオーケストレーションサービス
- **コンテナ**化されたアプリケーションのデプロイとスケール
- ALB などのサービスと協調して動く
- Fargate を利用すればクラスタを管理する必要もない

Docker とは

- **仮想化技術**の一種
- Linux 環境をまるごと **Docker イメージ**としてまとめて配布できる
- Docker が動く環境ならどこでも動かせる
  - AWS (ECS, Fargate)
  - Google Cloud (GKE)
  - もちろん手元の Linux や Windows, macOS でも

仮想化技術とは

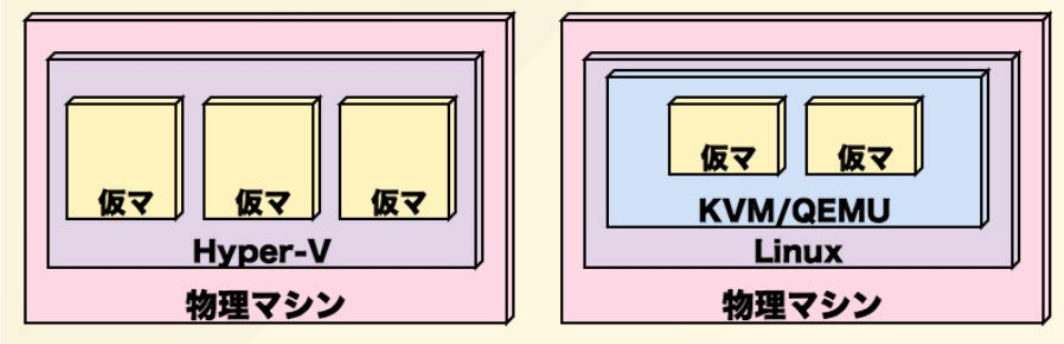
- 1 台の物理サーバを小分けにする
- 小分けした仮想的な環境で OS が動く
- 仮想マシン上の OS の上でアプリケーションが動く

仮想化によるメリット

- 物理サーバのリソースを効率よく利用できる
- 同じ環境をバーンとたくさん用意できる
  - 仮想ディスクのイメージをコピーして仮想マシンをたくさん起動する
- 耐障害性を向上できる
- データセンターで動く莫大な物理サーバ群の一部を
  必要に応じて借りられる
  - AWS・Google Cloud etc.

仮想化の方式

- 物理サーバに直接仮想化する機構を用意
- OS 上で仮想化を実現するソフトウェアを用意
- **コンテナ**と呼ばれる単位で環境を区切る



コンテナ技術

- 物理サーバで動く OS はひとつ
- アプリケーションが動く空間を区切ってリソースを制限する
  - プロセスから見えるものを制限して空間を区切る（言い換え）
- ここからは Linux と Docker に絞って話します
  - FreeBSD の jail などもある

Docker

- Docker それ自体は Linux の機能**ではない**
- cgroups や pivot_root などのプリミティブな Linux の機能のあわせ技
- Docker がやっていること
  - OS を構成するファイル一式を Docker イメージとしてパッケージ・展開
  - プリミティブな Linux の機能を使ってコンテナを実現しイメージ上でプロセスを動かす
- Docker っぽいものを作るのは意外と簡単
  - https://github.com/rrreeeyyy/container-internship

コンテナ内での制限

- CPU/メモリ/デバイス
- ファイルシステム
  - コンテナごとに特定のディレクトリを / (root) として見せる
- ネットワーク
  - コンテナごとに独立したネットワーク設定を持たせられる
- プロセステーブル
  - コンテナごとにプロセスの一覧が独立
  - つまり他のコンテナのプロセスは見えない

Docker イメージ

- OS や Rails、依存ライブラリー式を tarball としてまとめたもの
  - つまり OS の / (root) 以下のファイルが全部詰まっている
- ホスト OS に依存しないので Docker さえあればどこでも動く
- イメージの作り方
  - Dockerfile を書く
  - `docker build` コマンドを叩いてイメージを作る
  - `docker push` コマンドを叩いてイメージをアップロードする

The Twelve-Factor App

- https://12factor.net/
- Heroku による開発と環境構築に関する方法論
- コンテナ技術と相性が良い
- ベストプラクティスの例
  - コンテナに状態を持たせない、使い捨てにする
  - 環境変数を唯一の設定方法とする
  - ログは必ず標準出力に出す
  - プロセスの起動と終了は速やかに
  - デーモン化はオーケストレーションツールに任せる

コンテナオーケストレーション

- Docker でイメージを配布して動かすことはできるようになった
- ではどうやってサーバにコンテナを**いいかんじ**に配置するのか?
  - 空いてるサーバに配置する
  - コンテナが落ちたら上げ直す
  - ロードバランサと連携して外部からトラフィックを受けられるようにして負荷分散
- それらを行うのが**コンテナオーケストレーションツール**

代表的なツールたち

- Kubernetes
  - もともと Google による OSS
  - たぶん界隈で一番有名
- Amazon ECS
  - 今回はこちらを利用
  - クックパッドの本番サービスでも利用している

あたらめて ECS まわりを見てみよう

- 図は別資料を参照

レシピサービス (ブラウザ) の構成

- 図は別資料を参照
- 一見複雑に見えるが
  - Route 53 で名前解決して
  - ALB でトラフィックを受けて
  - ECS で動くコンテナにトラフィックを分散して
  - アプリケーションからはマネージドサービスなミドルウェアを利用する
  - という基本構成は変わらない
- ※ 国外サービスも含めるともうちょっと複雑ですが端折ってます
- cookpad service が別の service と通信していることに注目

マイクロサービスとは

- Kubernetes で管理されたコンテナで動いていて🤔
- 最新の技術が使われていて🤔🤔
- 規模の小さいサービスの集まり🤔🤔🤔

マイクロサービスとは

- Kubernetes で管理さ
- 最新の技術が使われてい
- 規模の小さいサービスの

マイクロサービスとは

- (Why) 組織全体のパフォーマンスを最大化するために
- (How) 開発からリリースまでのサイクルを回して
  コミュニケーションコストを下げることで
- **(What) 巨大化した組織とシステムを再編するためのアーキテクチャ**

理想のマイクロサービス

- 機能ごとにアプリケーションをサービスに分割
- サービス間では HTTP API や gRPC などで通信
- 各サービスで小さく速く開発イテレーションを回す
- 結果としてモノリスより速く価値を提供できるようになる

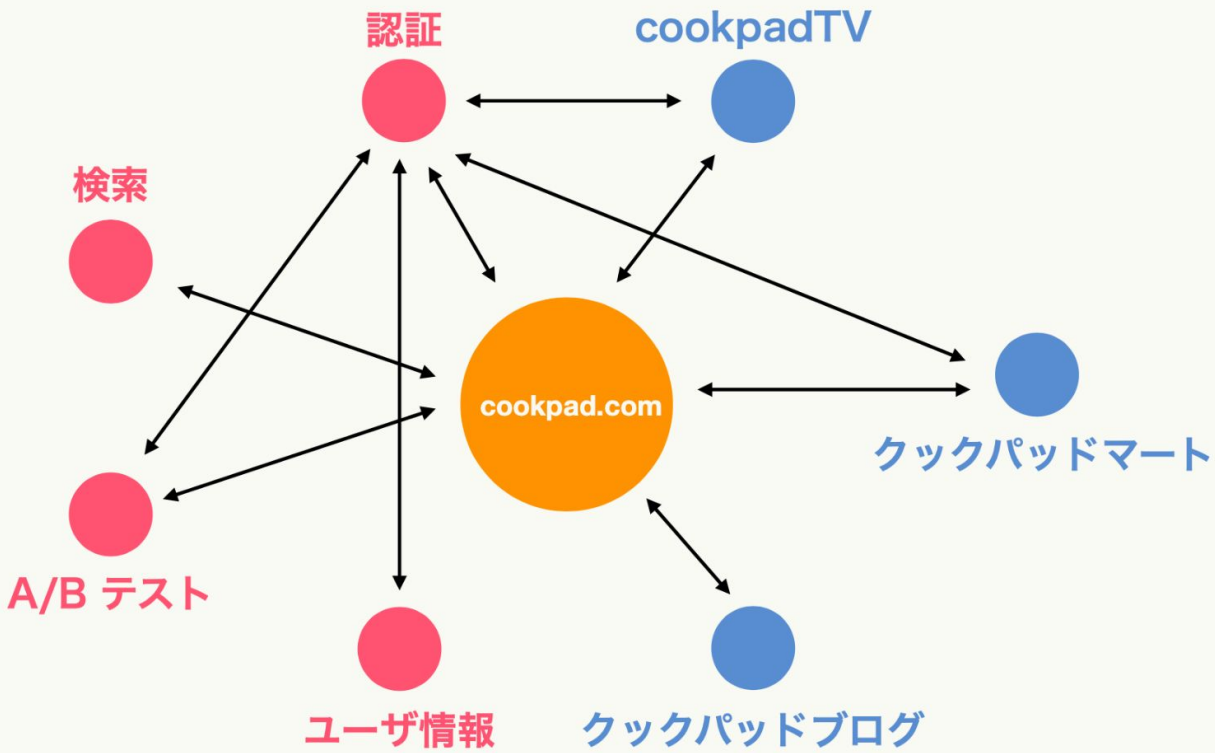
マイクロサービスとインフラ

- インフラチームはレシピサービスを見ているだけでよかった
- マイクロサービス化によって様々なインフラを支える必要が出てきた
- 個々のインフラを都度用意したり運用のサポートをするのは
  コミュニケーションコストが高くなりスケールしない
- コンテナオーケストレーションツールと AWS のマネージドサービスを
  活用してなるべく開発者がセルフサービスで新規サービスを
  立ち上げられるようにする
  - タイミングがすごく良かった

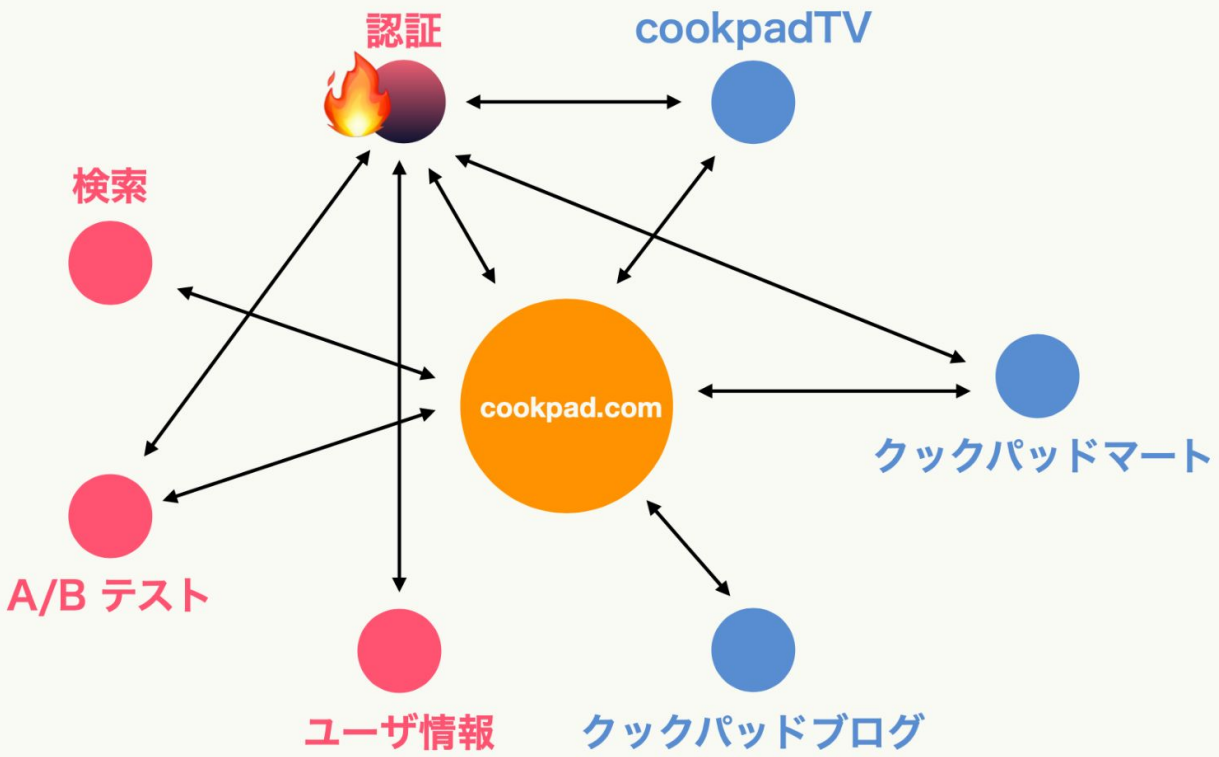
サービス間通信は難しい

- マイクロサービスは全てを解決するわけではない
- サービス間通信が新たな痛みとなる
- アプリケーション開発者の視点
  - サービス間通信を実装しなきゃ ...
  - 複数サービスへどうクエリしよう ...
  - テストどうしよう...
- SRE の視点
  - 障害がどのサービスで起こっているのか特定しづらい
  - カスケード障害の対応が大変

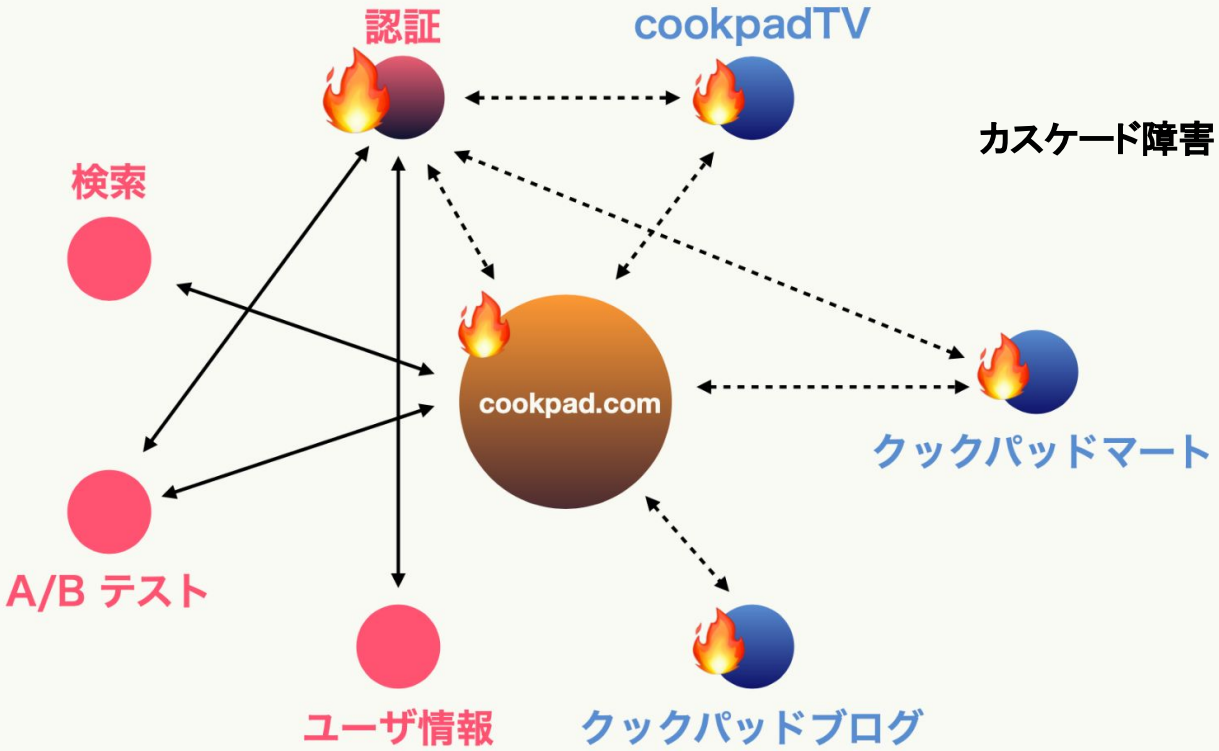
https://speakerdeck.com/itkq/a-life-of-chaos-engineering-starting-with-resilience を引用



https://speakerdeck.com/itkq/a-life-of-chaos-engineering-starting-with-resilience を引用



https://speakerdeck.com/itkq/a-life-of-chaos-engineering-starting-with-resilience を引用



サービス間通信は難しい

- 各サービスが一斉にリトライするとさらに燃え広がる
- どこどどこが通信してるのかも把握しづらい
- どこが火元なのかすぐに分からない
  - どこを直せばいいのかすぐに分からない

サービスメッシュの導入

- 以下を実現してサービス間通信の痛みを和らげる
  - 上流への適切なリトライ/タイムアウト/サーキットブレイク
  - サービスディスカバリ
    - 通信相手をいい感じに知りたい
  - オブザバビリティ(可観測性)
    - サービス間通信の状況を把握できる
    - キャパシティプランニングや障害の原因究明に役立てる
- 各サービスにプロキシを配置し、それを通して他サービスと通信
- プロキシを管理するコントロールプレーンが存在する
- 図は別資料を参照

ここまでを踏まえて...

- あらためてレシピサービスの今のインフラ構成を見てみよう
- 図は別資料を参照
- ここからは過去のインフラ構成を見ていきます

クックパッドの過去の
インフラ概観と開発スタイル

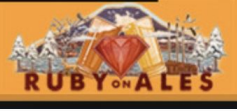
cookpad

世界最大級の Rails モノリス

- 2015 年
- https://speakerdeck.com/a_matsuda/the-recipe-for-the-worlds-largest-rails-monolith

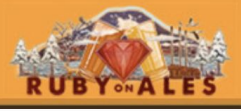
% rake stats

Name	Lines	LOC	Classes	Methods	M/C	LOC/M
Controllers	48552	39075	518	3941	7	7
Helpers	14660	12012	14	1390	99	6
Models	95193	74916	1732	8489	4	6
Mailers	2197	1757	44	204	4	6
Workers	593	501	20	31	1	14
Chanko units	11816	9732	6	247	41	37
Libraries	2781	2213	134	290	2	5
Feature specs	43536	35864	0	196	0	180
Request specs	36432	31235	0	16	0	1950
Routing specs	639	516	0	0	0	0
Controller specs	60543	50042	7	123	17	404
Helper specs	4195	3436	1	10	10	341
Model specs	75517	62368	4	72	18	864
Worker specs	862	715	0	1	0	713
Chanko unit specs	11636	9411	0	24	0	390
Library specs	22983	19202	27	131	4	144
Total	432135	352995	2507	15165	6	21



Tests

We have 20000+ RSpec examples



過去のレシピサービスのインフラ

- https://miro.com/app/board/o9J_l55WMMQ=/?moveToWidget=3074457361863487121&cot=14
- mamiya
  - 自前の高速デプロイツール
- RRRSpec
  - 並列分散テストツール
- SpotScaler
  - 安い EC2 スポットインスタンスを活用したオートスケーラー
- 基本的にいろいろ自前でやっていてモノリスを運用していた

過去のインフラで新規サービスを作るには

- インフラチームがアプリケーション実行環境を EC2 上に環境を用意
  - その他にも MySQL を用意したり
  - 必要であればオートスケーラを仕込んだり
- デプロイスクリプトを用意
  - Slack からたたけるようにしたり
- 手間がかかるしコミュニケーションコストも大きい
  - スケールしない
  - リードタイムが長い

モノリシックの問題点

大規模な変更を行えない技術的な理由としては以下のような点が挙げられます。

- コードを変更すると意図しないところが壊れる。例えばウェブサービスをいじるとガラケーの認証が壊れる。
- ライブラリが古かったとしても依存が多すぎて気軽に更新できない。
- 実行環境が非常に複雑かつ特殊で、迂闊にデータベースを追加したりできない。
- 普通のツールが動かない。例えばコードカバレージが取れない、並列テストが動かない。
- ObjectクラスやStringクラスのような非常に基本的なクラスが改変されており、普通の動きをしない。

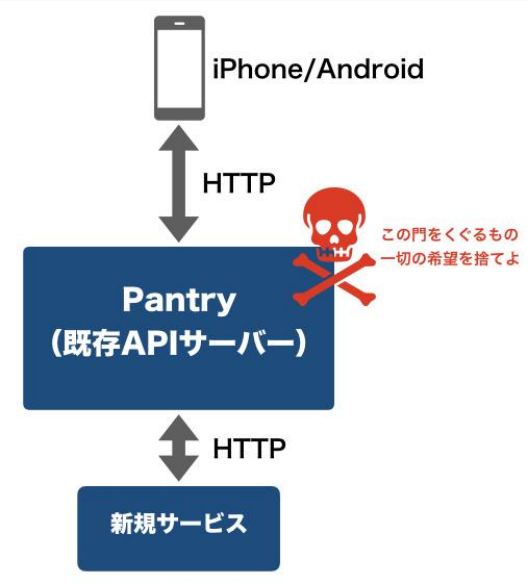
また、組織的・プロセス的な理由もあります。

- あるコードのオーナーが誰かわからない。例えばuserリソースのAPIを変更したくても誰にも相談できない。
- GitHubのissue・pull requestが多すぎてとても全部は見ていられない。通知も何も機能しない。
- 「本体」をいじる開発者が多すぎて、改善系のpull requestを作ると頻繁にコンフリクトする。

https://techlife.cookpad.com/entry/2018-odaiba-strategy

モノリシックの問題点

かと言って、既存APIサーバー（Pantry）の改修もしたくありません。図2のように、Pantryから新サービスを叩くように変更すればAPI呼び出しを1つにまとめることはできます。しかしこのPantryというサーバーは以前の記事で説明した「世界最大のモノリシックなRailsアプリケーション」であり、理由はよくわからないがとにかくこれをさわるだけで開発期間が3倍になる優れモノです。できることならいっさいPantryにさわることなく開発を終えたいわけです。



https://techlife.cookpad.com/entry/2019-orcha-bff

そして今のカタチへ

- 最初はうまくいっていたがスケールしなくなった
  - 組織が小さければモノリスのほうが良いケースもあるはず
- レシピサービス以外にもやっていきたい
- 2014 年からマイクロサービスアーキテクチャに移行開始
- そして今のカタチになった

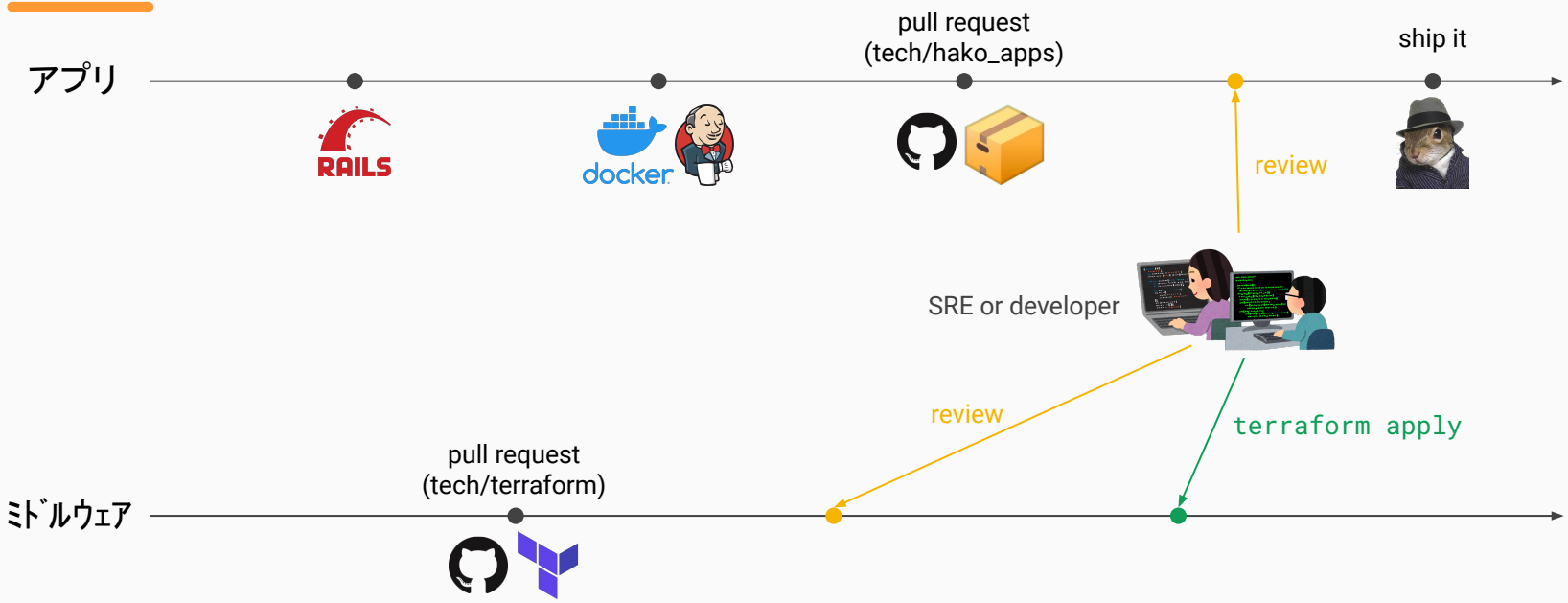
Hako エコシステムによる現在のアプリケーション開発スタイル

cookpad

インフラ運用の主な技術スタック (再掲)

- **Hako / ECS**
  - アプリケーションを動かすための基盤
  - コンテナオーケストレーション
- **Terraform**
  - AWS リソースの管理
  - Hashicorp 社によるオープンソースのツール
- 基本的にすべて pull request ベース
  - いわゆる GitOps

新規開発のフロー


アプリ
pull request
(tech/hako_apps)
ship it
RAILS
docker
review
SRE or developer
review
terraform apply
pull request
(tech/terraform)
ミドルウェア

実際の pull request をみてみよう

- 図は別資料を参照

hako-console & Grafana

- Hako エコシステムの中核といっても過言ではない
- デプロイされている Hako アプリケーションを
  - 一覧
  - 各アプリケーションに関する情報へのリンク集
  - コンテナのログやメトリクスを確認できる
  - 設定などを確認できる
- アラートを開発チームに届ける仕組みもある
- 図は別資料を参照

サービスメッシュの活用

- サービスメッシュの整備によりサービス間通信が可視化されている
- 設定も pull request ベースで
- 図は別資料を参照

AWS リソースを Terraform で管理

- Infrastructure as Code のためのツール
- 基本的に AWS のマネージドサービスを利用
  - 運用コストの軽減
  - 開発チーム自身で運用
- pull request ベースでリソースを管理したい
- 独自の linter で設定の不備を自動で指摘
- 図は別資料を参照

AWS コスト管理

- 様々なチームが様々な AWS サービスを利用することにより
  コスト管理の重要性が増してきた
- 予算内に収めることは絶対なので
- https://techlife.cookpad.com/entry/how-to-describe-infra-cost

サーバレスアーキテクチャについて

- Hako / ECS エコシステムは万能
  - Docker イメージさえあれば何でも動く
  - hako-console や Kuroko2、Barbeque といった基盤が豊富
- サーバレスアーキテクチャがより向いてる場面もある
  - S3・DynamoDB・Lambda などを活用したアーキテクチャ
  - ECS task は基本的にずっと立ち上がった状態
  - イベント駆動で処理したい場合など
    - e.g. S3 にファイルをアップロードした時に処理したい
    - ECS でもデーモンを常駐させれば可能だが ...
- CDK + CloudFormation

サーバレス実例: Ganmo

- Tofu という画像配信システムに関連
- S3 バケットにアップロードされた画像を Google Cloud Storage にコピーするシステム
- (Ganmo の design document へのリンク (外部向け資料では非公開))

サービスの立ち上げはできた、運用は...?

- デプロイ・AWS リソース管理・モニタリング・アラーティングなどが開発チーム主導でできるようになってきた
- あらためて「運用」について考えてみる
  - 本番環境でエラーが出ていたら誰がどう対処するのか ？
  - ユーザにどの程度影響があったか？(ビジネスインパクト)
  - 機能開発の手を止めてでも信頼性の改善に集中すべき ？

運用シナリオ

- Dev: 機能開発を優先して信頼性をおろそかにしてしまう
  - アプリケーションコードの品質の低下
  - ユーザへの影響
- Ops: アプリケーション起因のアラート対応
  - ドメイン知識がないので不具合の修正が困難
- 逆に過度に慎重になりすぎてユーザに価値を届けることが遅くなるパターンも考えられる

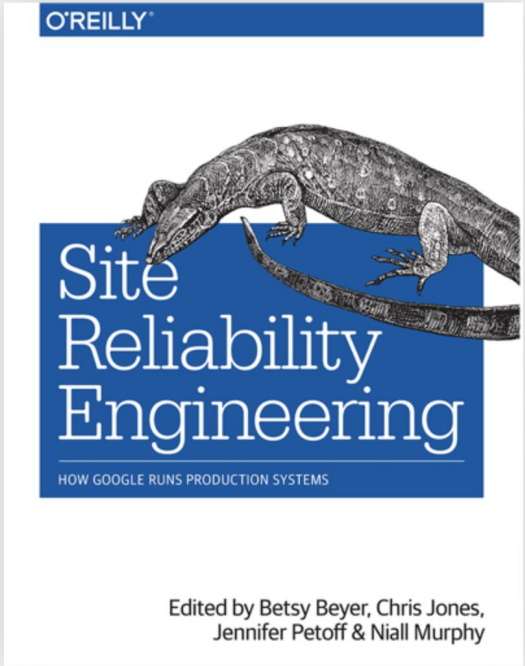
SLI / SLO

- 信頼性リスクを 0 にするのは不可能!
- 障害を受け入れる
  - ただし当然障害は少ないほうが良い
  - どのくらい受け入れられるかを定義する
  - e.g. サービスが 10 分ダウンした場合の損失は？

SLI / SLO

- SLI: サービスレベル指標
  - e.g. Uptime
- SLO: サービスレベル目標
  - e.g. 月の Uptime が 99.95%
    - つまり月 22 分ダウンしててもいい
  - ステークホルダーすべてで共通する目標
  - これを割った場合には機能開発の手を止めてでも信頼性を上げる改善に取り組む

Site Reliability Engineering



SLO があると何が嬉しいのか

- 数値に基づいた定量的なコミュニケーションができる
  - 「サービスがちょっとでも落ちるとまずいんで絶対落とさないようにしてください ！」みたいなのがなくなる
  - SRE は必要十分なコストを払って信頼性を担保する動機ができる

SLI / SLO の具体的な事例

- クックパッドマートでは実際に SLI/SLO を導入している
- (SLI/SLO が見られる Grafana ダッシュボードへのリンク (非公開))