LAMPで構成されたモバイルゲームのバックエンドのKubernetesへの移行事例

泊 久信, 中村裕也
株式会社SNK

The Future Is Now
SNK®

資料は以下URLから今すぐダウンロード

https://bit.ly/cedec2019snk

CEDiLにも公開します

自己紹介

- 泊　久信
  - 2016 年からSNK
  - コンシューマゲームのプログラム、およびモバイルゲームのインフラを担当
  - 最近はゲームエンジンを作りたいと思うことが多くなった

- 中村 裕也
  - 2016 年からSNK
  - モバイルゲームのインフラ専任　運用をする人
  - Kubernetes（GKE）を通じて最近少々 GCP贔屓

あらすじ

- はじめに
  - 旧環境の構成と自動化へのあゆみ
  - 旧環境の問題点とKubernetes
- Kubernetesで動かすまで
  - コンテナ
  - Pod
  - クラスタ
  - マネージドサービス
  - ロードバランサ
- Kubernetesでうごいてから
  - 運用のいろいろ
- まとめ

はじめに

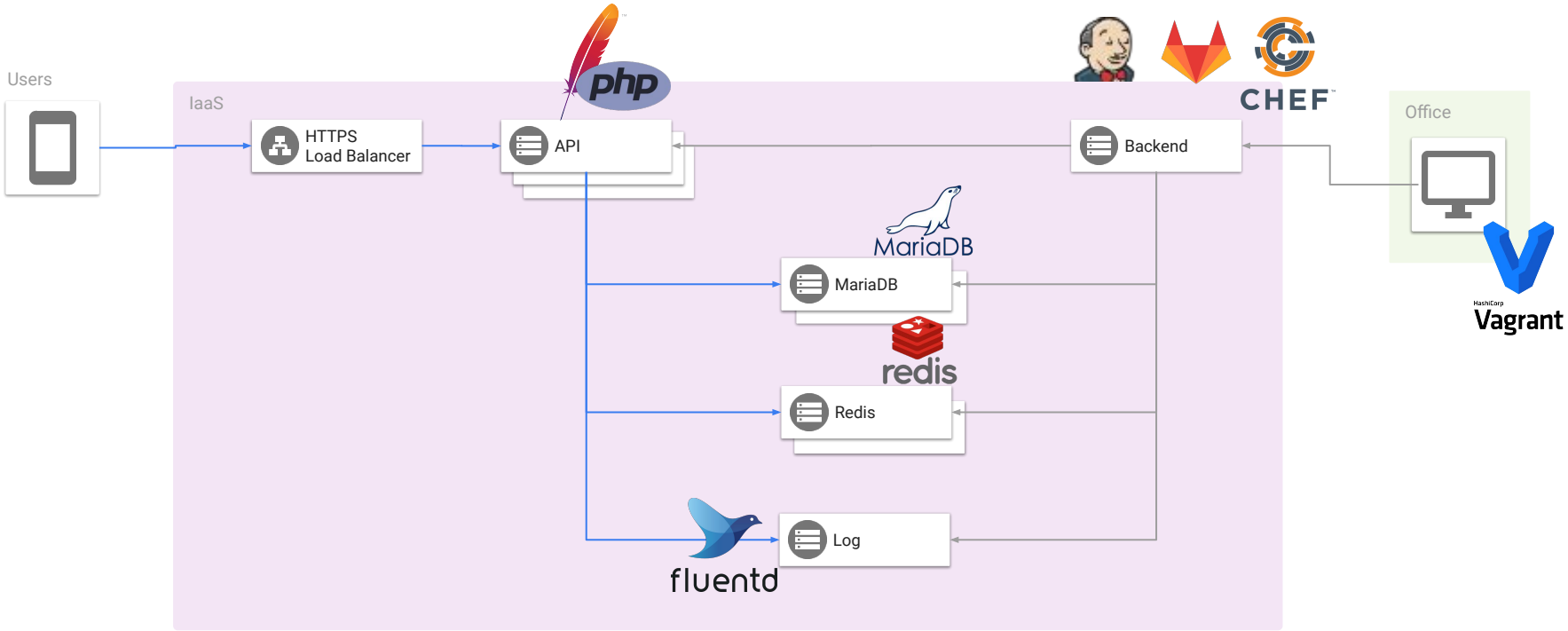
- Kubernetes　流行っているから　使いたい
  - Kubernetes: サーバーの構成とか管理を自動でやってくれる便利な仕組みらしい！！

でも

- 慣れ親しんだ構成のインフラとプログラムがある
- モバイルゲームの要件をうまくあわせ込めるか？
  - 審査とアップデート
  - 複数バージョンを同時に開発・デバッグ

私たちのモバイルゲームの全体像

旧環境の構成概要


Users
IaaS
HTTPS
Load Balancer
API
Backend
CHEF
Office
MariaDB
MariaDB
redis
Redis
Log
fluentd
HashiCorp
Vagrant

旧環境でも自動化は進んでいました

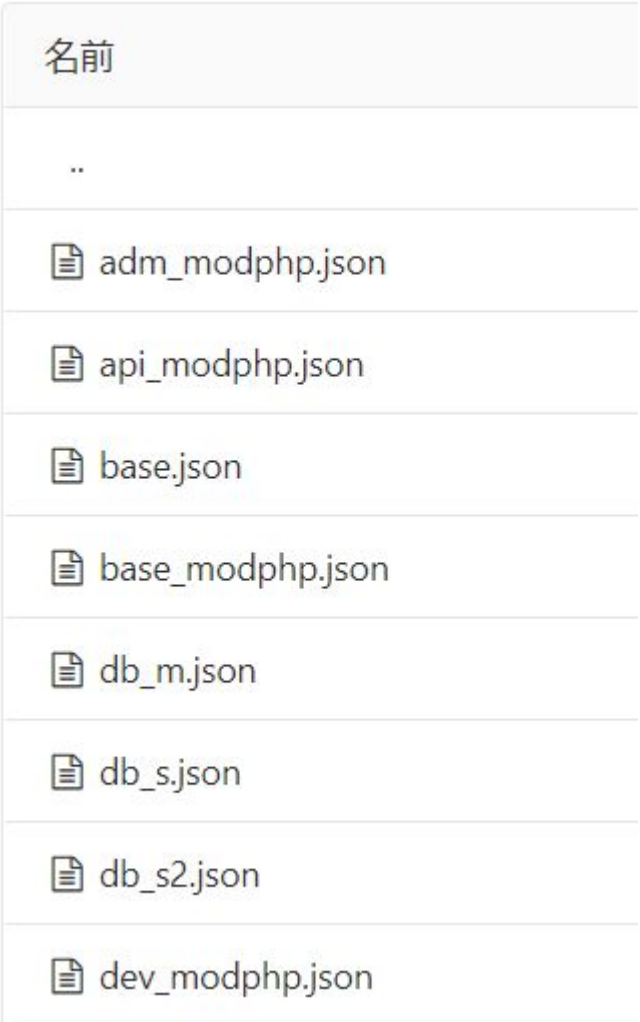
- 環境構築をChef で自動化

	A	B
1		
2		
3	**OS設定**	
4		
5	ホスト名登録	
6	`#  vi /etc/hosts`	
7		
8	`127.0.0.1  localhost loca`	
9		
10	`#  vi /etc/sysconfig/network`	
11		
12	`HOSTNAME=localhost.loc`	
13	`HOSTNAME=banner`	
14		
15	`#  hostname banner`	
16		

memcached.rb 319バイト

```
1  %w(memcached memcached-devel).each do |pkg|
2          package pkg do
3                  action  :install
4          end
5  end
6
7  cookbook_file '/etc/sysconfig/memcached' do
8          source  'memcached/memcached'
9          owner   'root'
```

旧



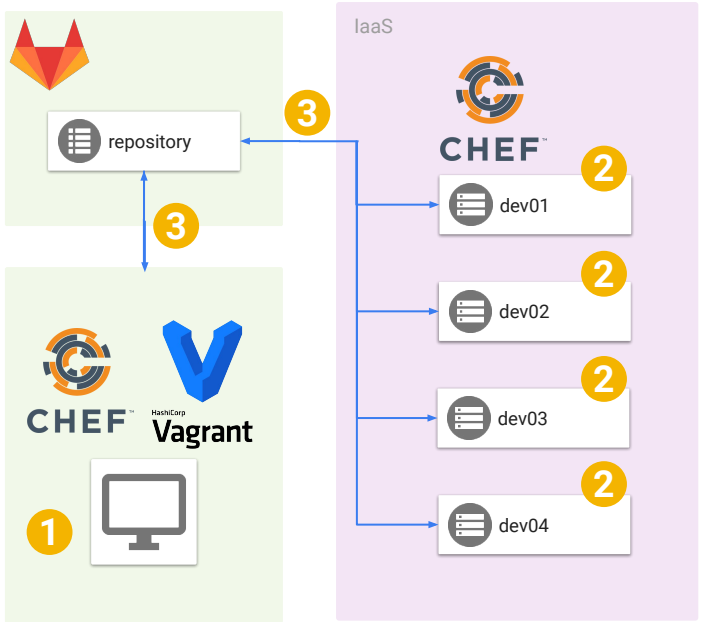
①各サーバのroleを作成しレシピ化

②レシピをリポジトリ管理

③レシピをサーバへ展開

④各サーバ上でroleを実行

旧環境の開発環境



①ローカル環境はVagrant+Chefで構築

②サーバー環境は開発用roleでChef実行

③アプリケーションプログラムは各環境からGitlabと連携

旧環境の問題点

- サーバのスケールアップは容易だがスケールアウトは躊躇
  - 構成管理、冪等性の担保が不安は Chefで解消したが、増やす際は人間の作業が発生
- 障害時のフェールオーバー等、特殊なオペレーションが手動
  - 状況による対応になるため経験と技量が必要

管理コストを下げることはできたが、手動運用は残る

我々のアプローチ

- 新タイトルのローンチに合わせてKubernetes を全面採用
- サーバーサイドのコードは旧来のLAMP のものを改良・修正して利用

2015	2016	2017	2018	2019	2020

タイトル：大進撃RPG！シスタークエスト
配信日：2015年2月26日

タイトル：METAL SLUG ATTACK
配信日：2016年2月15日

LAMP 手作り系

LAMP 自動

タイトル：君はヒーロー
配信日：2017年8月7日

Kubernetes

タイトル：KOFクロニクル
OBT開始：2019年6月17日

Kubernetesをどこで使うか

- Kubernetes が使えるクラウドはいくつかある

- コンテナ運用の実績が一番長そうなGoogle Cloud Platform を使うことに

Kubernetes で動かすまで

開発手法

- 設計、構築、試験、を繰り返し行った
  - 知らないことがおおいのでまずやってみるところから始めた
  - 技術の全体像が分かってなくても、とりあえず部品から作ってみる
- 「とりあえず動かす」の積み重ね

- 今は全体がそこそこ見えるようになった

コンテナ

まずはコンテナ化

- 最初に開発環境をコンテナ化した
  - 一通り動かすためのすべての部品が揃っている、小さな環境
  - 手元のDocker を入れた環境で作業
- 開発環境の内容
  - Apache+PHP+プログラム　自分たちで作成
  - Memcached
  - MariaDB　Docker Hub のイメージをそのまま利用
  - Redis
- ひとまず動いて、アプリケーションエラーが出るまではすぐ

コンテナ化概要

- アプリのリポジトリにDockerfile が同居
  - Chef 運用時はインフラは別リポジトリだった

- デプロイ時に必ずコンテナビルドを行う
- ビルド時の外部依存があって、リリース・開発が遅れたくない！！
  - パッケージ配布サーバーがダウンしてるなどの要因

- 外部ライブラリ、パッケージ等はベースイメージで管理
- 通常リリース時のコンテナビルドはソースのコピーだけ

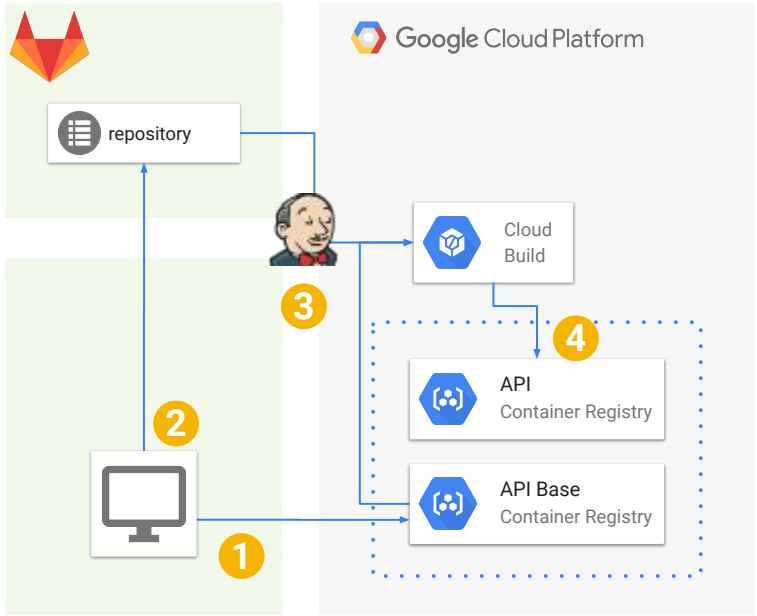
Baseイメージ

```
FROM php:7.2.8-apache

# install prerequisites
RUN apt-get update && \
        apt-get install --no-install-recommends -y libmemcached-dev zlib1g-dev && \
```

アプリバージョンごとのイメージ

```
FROM gcr.io/                /kofabase:latest
ARG KOFA_REV=master

# Copy apache configurations
COPY conf/apache2.conf /etc/apache2/apache2.conf
```

コンテナのビルド 通常時



①Apache+PHP+依存ライブラリのビルド時に外部アクセスが必要な部分をBaseコンテナとして作成

②Dockerfileを含むアプリケーションをGitlabへコミット

③Jenkinsからコンテナのビルドを実行

④Cloud Buildでビルドしたコンテナを Container Registryへ保管

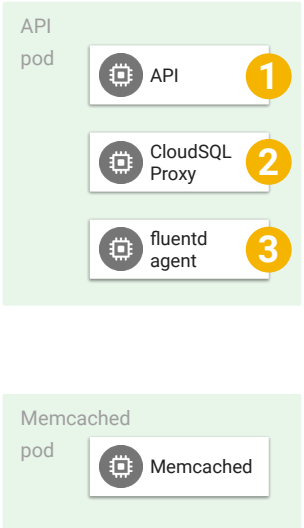
Pod

Kubernetes のPod

- Pod
  - いくつかのコンテナの集まり
  - 旧環境で言う仮想マシン1台分と考える
  - Podが増えたり減ったりする
- Apache+PHP+自前プログラム　のコンテナが中心
  - 動作に必要なサービスのコンテナをそのまわりに配置できる

- YAMLで配置方法・接続方法を宣言
  - YAMLもサーバープログラムのソースに同居

Podの構成



①Apache+PHP+自前プログラム

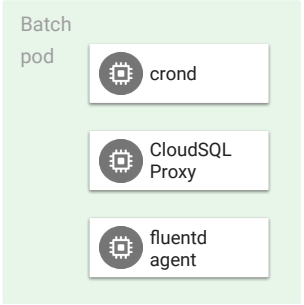
②APIコンテナがMariaDBへ接続するためのProxy

③ログを取得

Pod の困ったところ

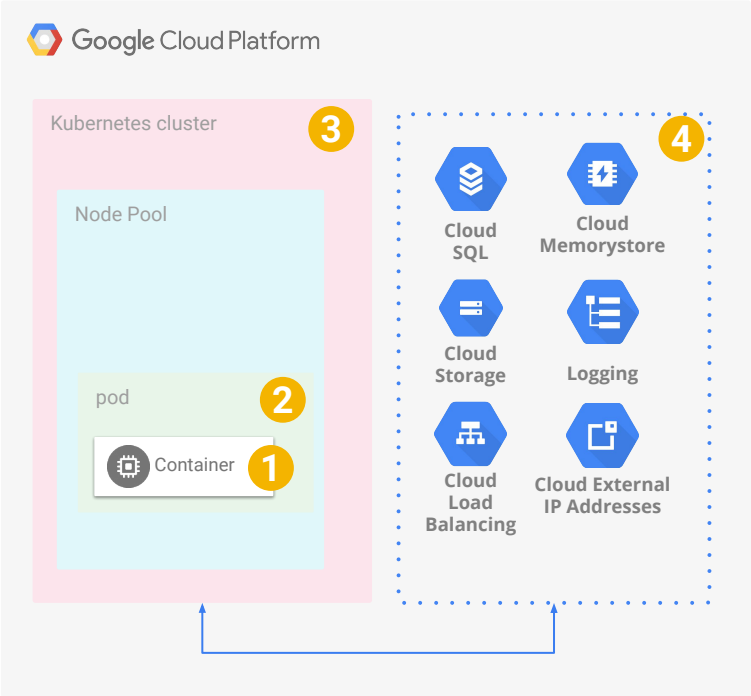
- バッチ処理をどこでやるか
  - 報酬付与
  - ランキング処理　など、数十のバッチが存在
- Cronjob で定期的にPod を動かす？
  - 記述がかなり冗長

- 今回はcrond を動かすコンテナを用意
  - デプロイタイミングによって不具合が起きるか？
- 将来的にはCronjob を利用する方向に変更したい



クラスタ

Podはどこで走るか - クラスタ



クラスタはNode Pool をもつ。

Node Pool はVMの集まり。

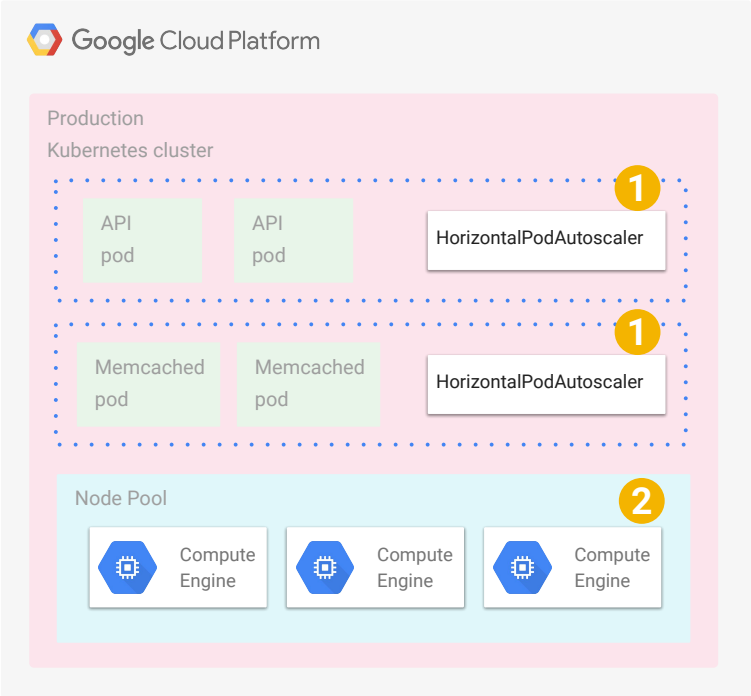
①コンテナ

②コンテナのまとまりであるPod

③Podのまとまりであるクラスタ

④外部コンポーネントとの連携

オートスケールの構成



①Pod毎にオートスケール設定

②ノードプールのオートスケール設定

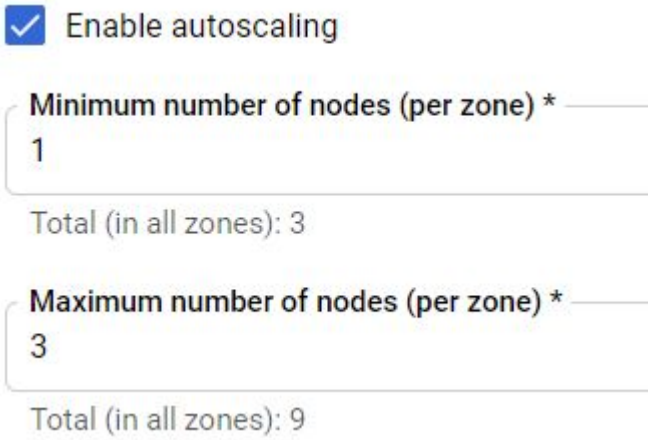
```
metrics:
- type: Resource
  resource:
    name: cpu
    targetAverageUtilization: 66
```

オートスケールの構成

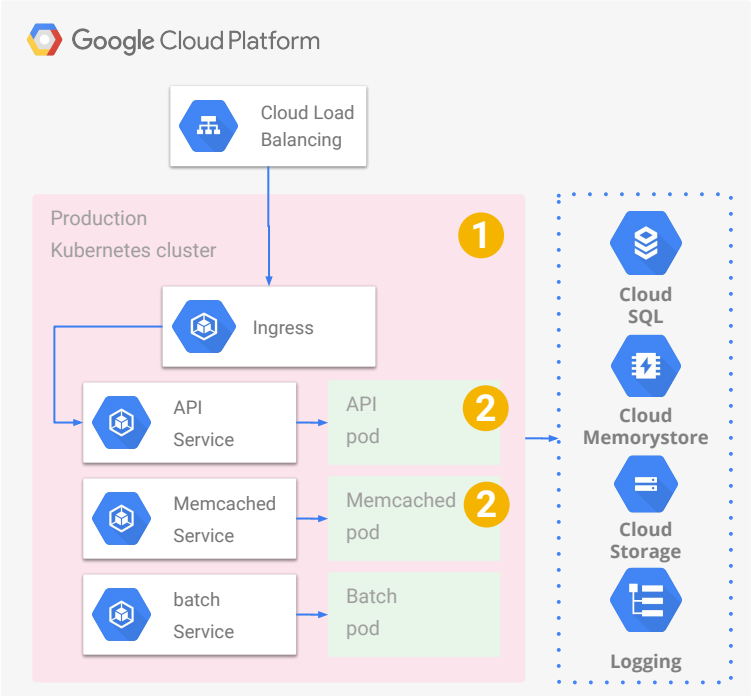
```
resources:
  requests:
    cpu: 1m
    memory: "32M"
  limits:
    cpu: 333m
    memory: "256M"
```

- 負荷状況でPodが増える
- Podの各コンテナには最低保障のリソースがセットされている
  - Podが入らなくなるとノード(VM)が増える →課金
- オートスケーラーを(自分より)信頼する
  - はじめにちゃんと動いていることが確認できれば信頼して任せて運用負荷を軽減する

- ノードのスケールは明示的に最大数を設定
  - 人気ゲームとはいえ、急に流行り出したら本当に流行っているのかお伺いを立てたい



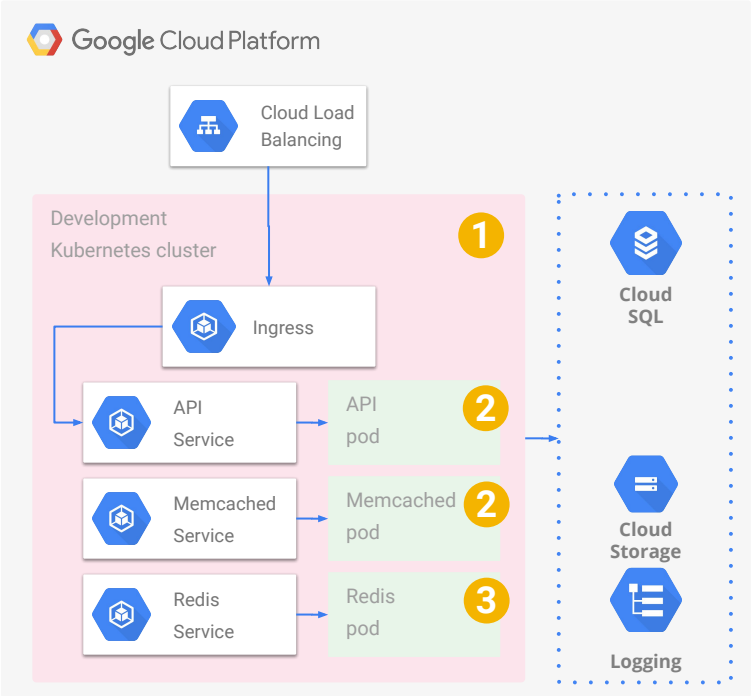
Kubernetesクラスタの構成　本番系



①クラスタのノードはオートスケールを有効化

②API、MemcachedのPodはオートスケールを設定

Kubernetesクラスタの構成　開発系



①クラスタのノードはオートスケールを有効化

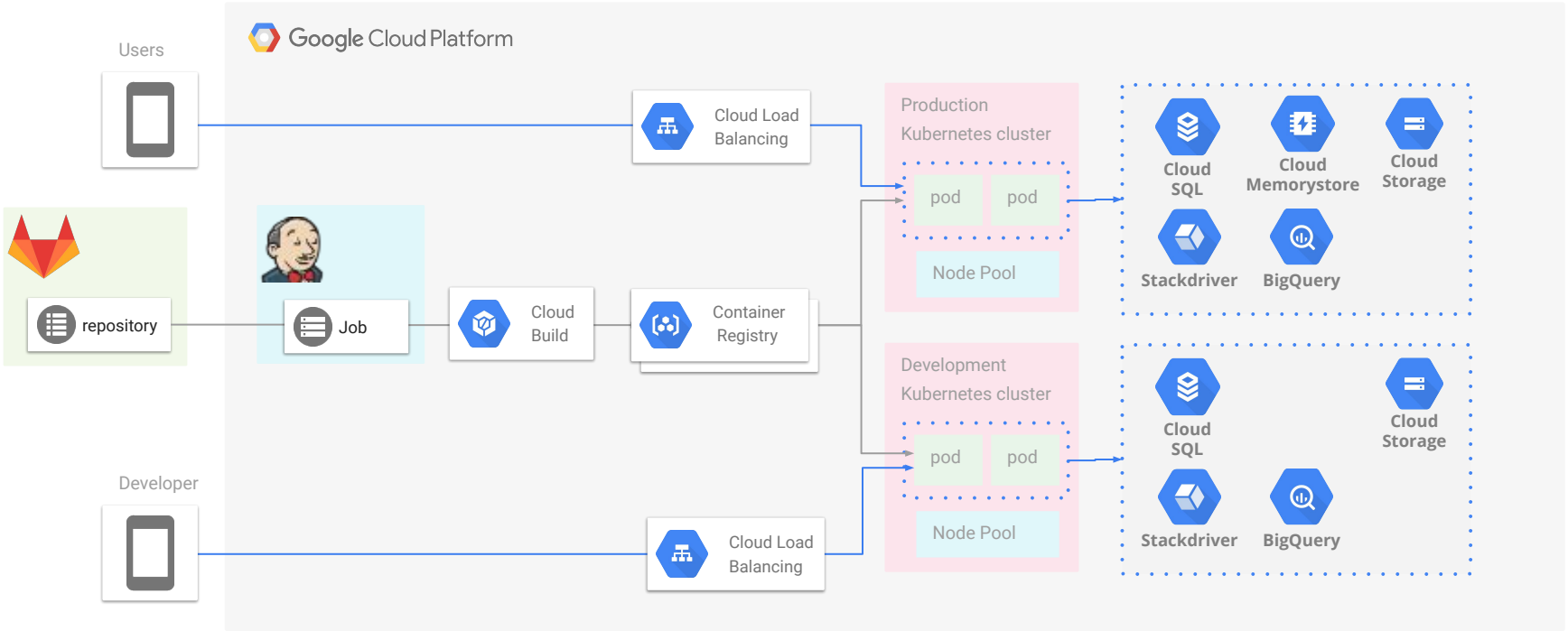
②Podはオートスケールさせない

③Redisはコンテナ

Kubernetesクラスタの構成

- 開発環境でのいたずらにより本番環境へ影響を与えたくなかった
- コマンドラインではクラスタ毎に認証が必要
  - ヒューマンエラーの抑止につながる
- コストを考慮してRedisはコンテナにした

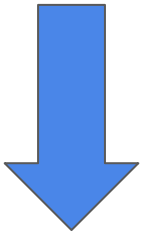
全体像


Google Cloud Platform
Users
Developer
repository
Job
Cloud Build
Container Registry
Cloud Load Balancing
Cloud Load Balancing
Production Kubernetes cluster
pod
pod
Node Pool
Development Kubernetes cluster
pod
pod
Node Pool
Cloud SQL
Cloud Memorystore
Cloud Storage
Stackdriver
BigQuery
Cloud SQL
Cloud Storage
Stackdriver
BigQuery

マネージドサービスの利用

永続的データの扱い

- 永続させたいデータ
  - ログ
  - お客様情報（所有メダル数、アイテム保持状態）
- Pod は勝手に消えたり増えたりする
- I/O性能の不安からも、Pod にMariaDB は配置しないことにした

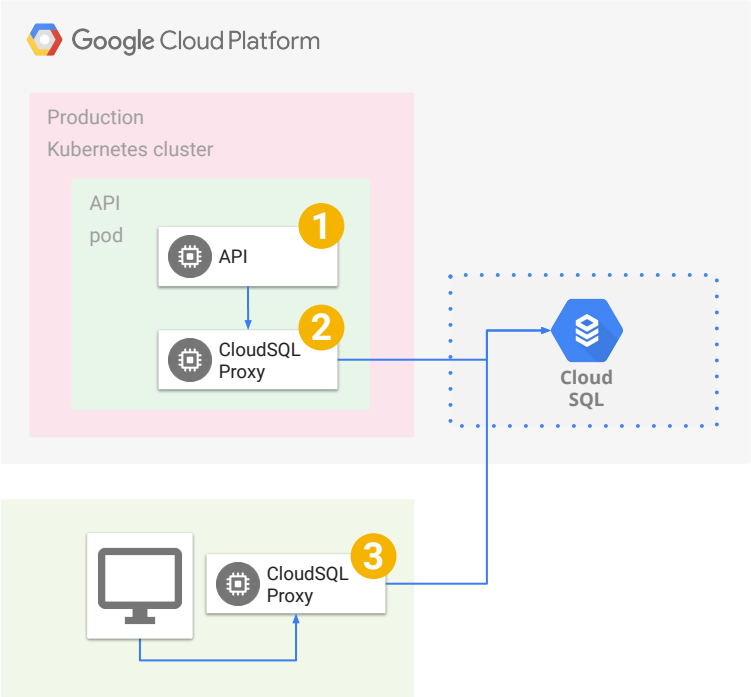


- 永続的データはマネージドサービスを利用することに

私たちが使ったマネージドサービス

- CloudSQL
  - MariaDB 互換のデータベース
  - よいところ
    - 難しいことをしなくても実際動くフェイルオーバー構成ができる
  - イマイチなところ
    - メンテナンスでダウンタイムがある
- Cloud Memorystore
  - Redis 互換のサービス
  - よいところ
    - 難しいことをしなくても実際動くフェイルオーバー構成ができる
  - イマイチなところ
    - 自分たちの使い方だとちょっと高価？

CloudSQLへの接続



①APIコンテナからは127.0.0.1で
CloudSQL Proxyへ接続

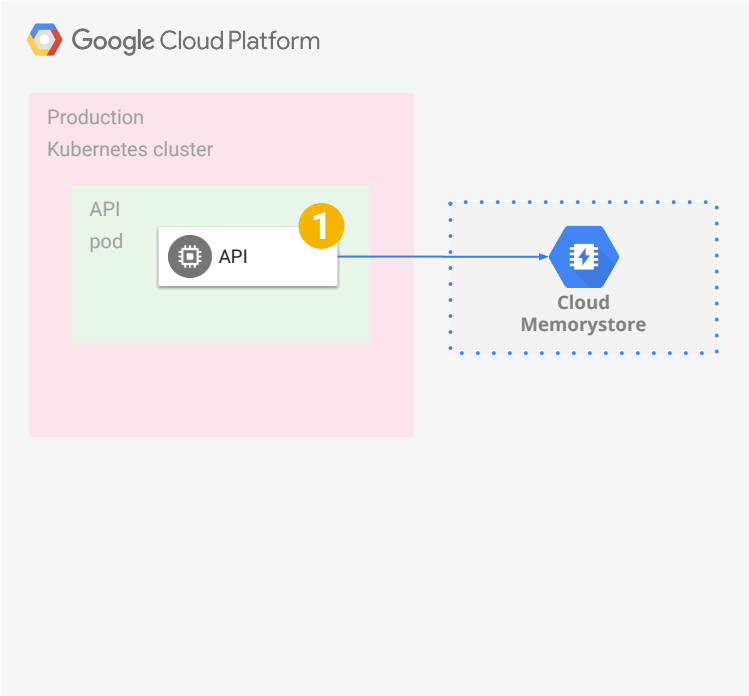
②CloudSQL ProxyがCloudSQLへ接続

③ローカル環境からでもCloudSQL Proxy
が使える

CloudSQLへの接続

- プログラマが接続先を意識する必要がなくなった
  - CloudSQL Proxyがインスタンスの認証情報を持っている
  - 接続するデータベース名は環境変数で持っている
  - 開発環境から本番DBへ接続するような過ちはなくなる
- ローカルからクライアントツールを使った運用があった
  - 以前は自前のSSHトンネルを試用
  - WindowsでもCloudSQL Proxyは便利

Cloud Memorystoreへの接続



①APIコンテナからCloud Memorystoreの
IPアドレスを指定して接続

Cloud Memorystoreへの接続

- コンテナからCloud Memorystoreへ接続できなくて困った
  - Podに割り当たるアドレス範囲が GCPネットワークと違うためルーティングできないらしい
  - VPCネイティブなクラスタを作成することで解決

- プロトタイプ段階で、マネージド含めすべての部品を使って試して見る必要あり

ロードバランサ連携

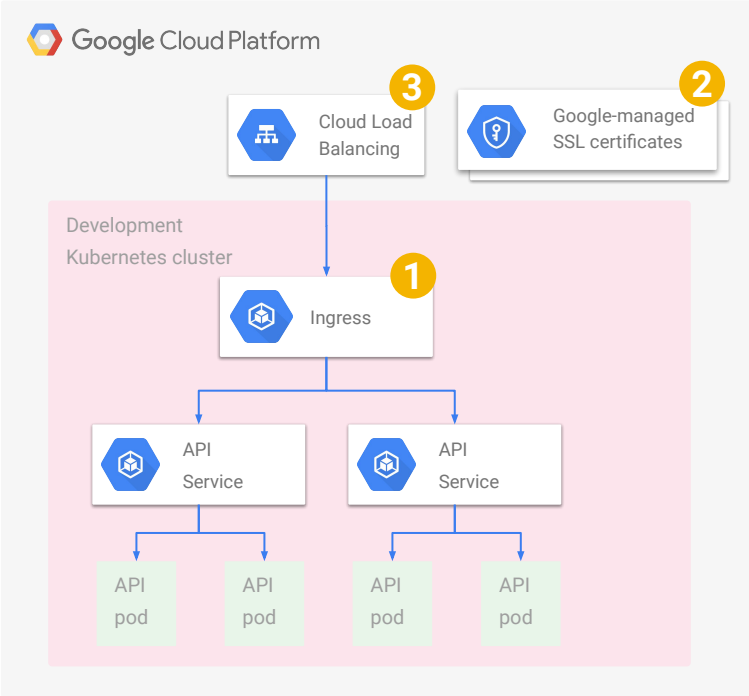
通信の入り口 Ingress

- Ingress はL7 ロードバランサをyaml で設定できる仕組み
  - yaml はソースコードと同居
- GCP の場合、TLS を終端してくれる機能がある

- Pod にはサービスの定義を介して接続される

```
spec:
  type: "NodePort"
  ports:
  - protocol: "TCP"
    port: 11211
  selector:
    app: "[redacted]-memcached"
```

ロードバランサと証明書の構成



①ホスト名でServiceへ振り分ける

②ホスト名分の証明書を取得

③取得した証明書を割り当てる

Ingress イマイチなところ

Ingress はGoogle Cloud Load Balancing の抽象化で、すべてのところに手が届くわけではなかった

- Ingress側の設定でIPv4、IPv6の2つのIPアドレスをLBに割当てられなかった
  - IngressでIPv4を設定しGCPコンソール上でIPv6を追加で設定

- LBに設定できる証明書は10枚までという制約
  - 複数バージョンを同時に開発するため、環境は10以上
    - 環境ごとにエンドポイントが 2 つある
  - 証明書の数に合わせてLBを追加した

Kubernetes で動いてから

運用の話題

- Kubernetes、オートスケールという新しい機能を使い始めた

- 新しくGoogle Cloud Platformを使い始めた
  - マネージドサービスがたくさんあるらしい

運用の観点で見ると新しいことが増えることは不安

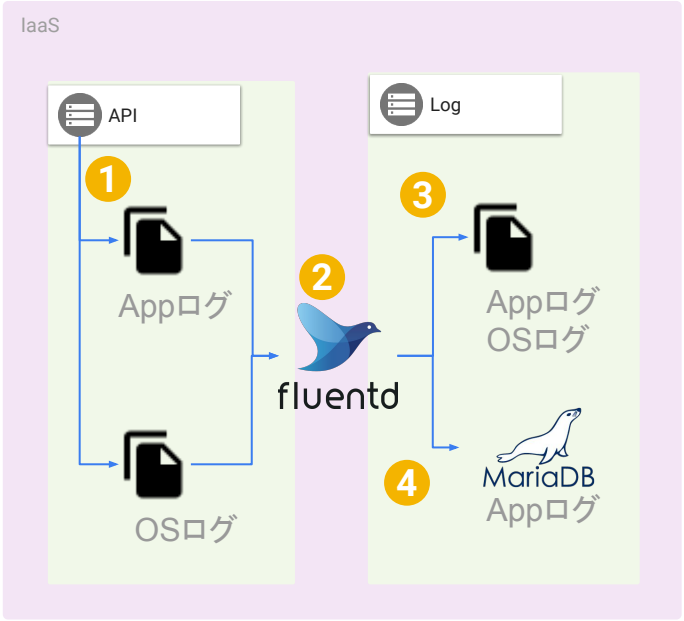
運用の話題

- 各運用項目における新旧比較

運用項目	旧環境	新環境
ログ収集、分析	△	◎
システム監視	○	◎
アプリ更新	○	○
システムアップデート	△	○

従来環境で手間かかっていた部分が解決できた

旧環境のログの取り扱い



①各ログはファイルで出力

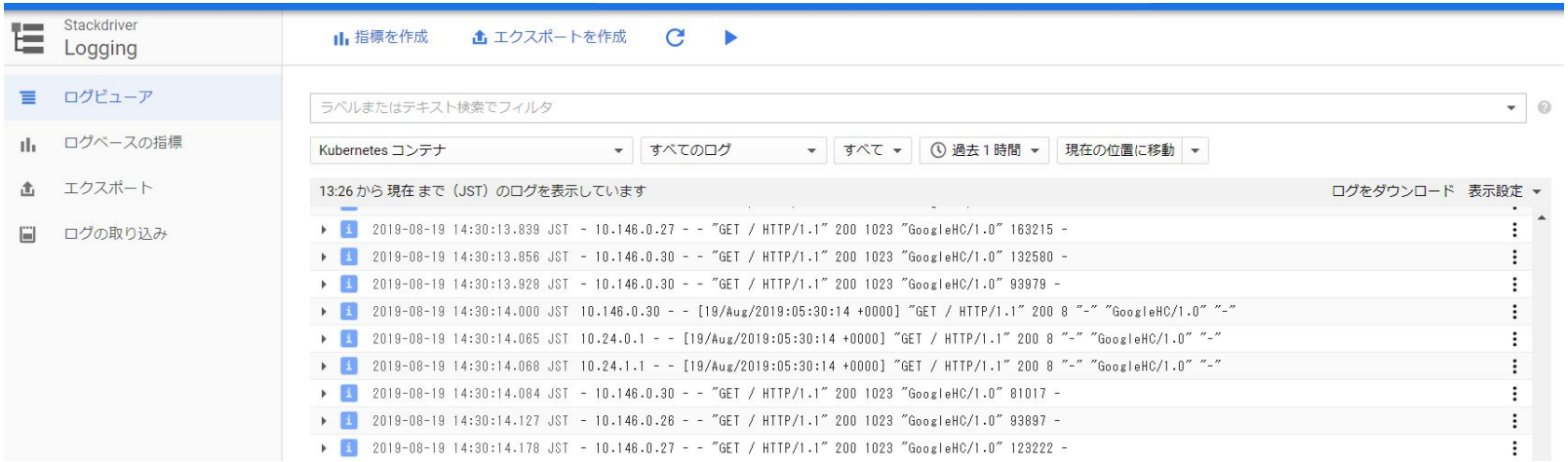
②fluentdに定義したログを収集

③収集したログをファイルで保管

④Appログはデータベースに保管し分析に利用する

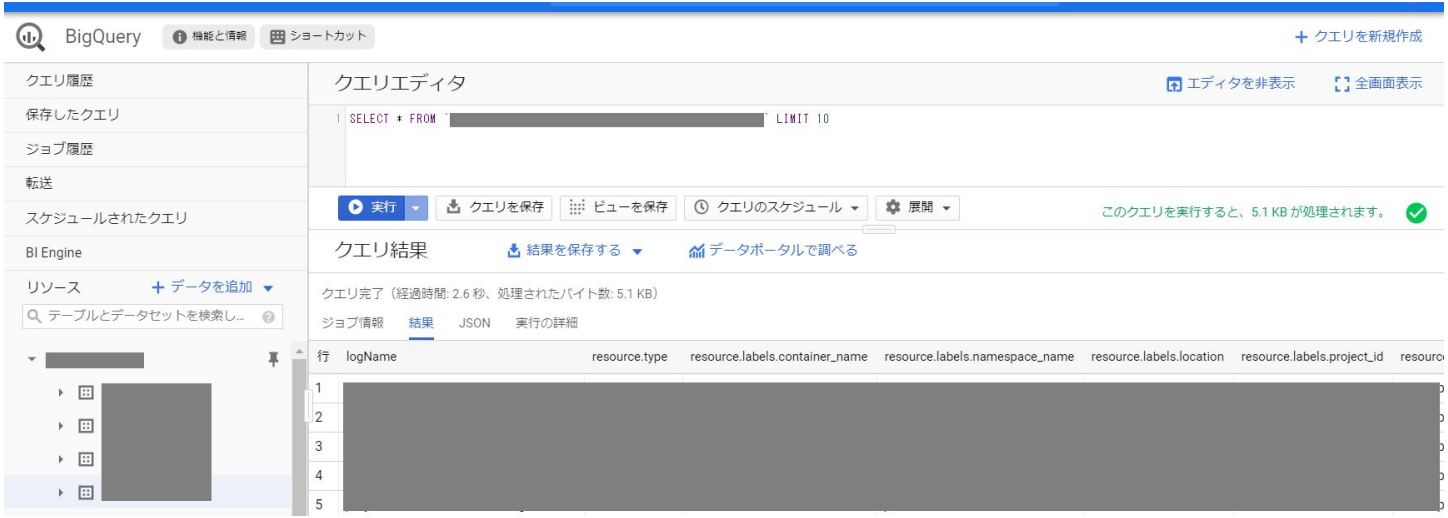
ログで使う便利な仕組み

- Stackdriver Logging
  - グーグルクラウドのサービス
  - ログをいったん受け取る仕組み
  - 30 日分のログを持っておける

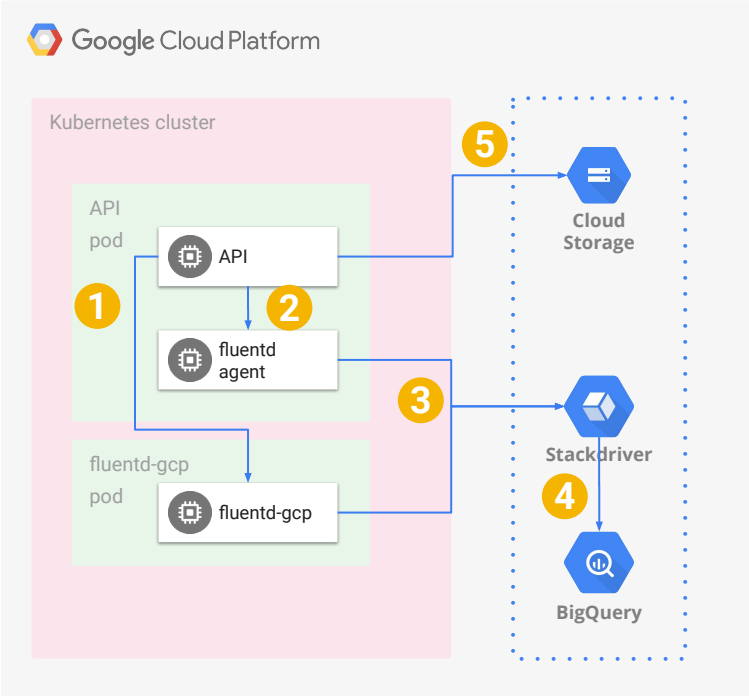


ログで使う便利な仕組み

- BigQuery
  - グーグルクラウドのキラーアプリ
  - Stackdriver で受け取ったログを持っておくのに利用
  - クエリが思ってるより早い



ログ収集



①標準出力・エラー

②ファイルログ

③fluentdでキャッチしたものはすべて
Stackdriverへ渡す

④分析対象のログはBigQueryへ渡す

⑤保管目的のログはCloudStorageへ渡す

ログ収集

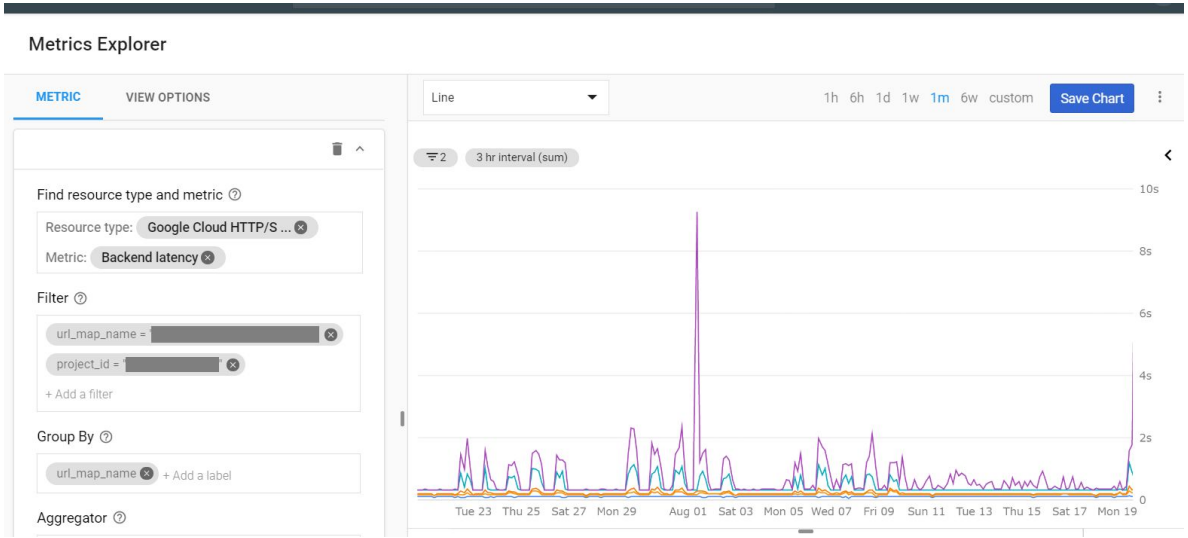
- ログをファイルで出すという慣習
  - できるものは標準出力に変更してもらった
  - ファイル出力から変更できなものは自前の fluentdで補う
- 長期保管が目的のログは直接CloudStorageへ渡してもらうようにした
- BigQueryへ渡すログはログの種類ごとに設定
  - ログが追加されるたびに設定が必要なため運用に組み込む必要がある

ファイルログもカバーできる

オートスケールにフィットした構成が苦労なく実現　〇

監視

- Stackdriver Monitoring
  - Kubernetesやグーグルクラウド上の資源のメトリクスを自動収集してくれる
  - ログとも連携できるので任意のログでアラートを飛ばせる



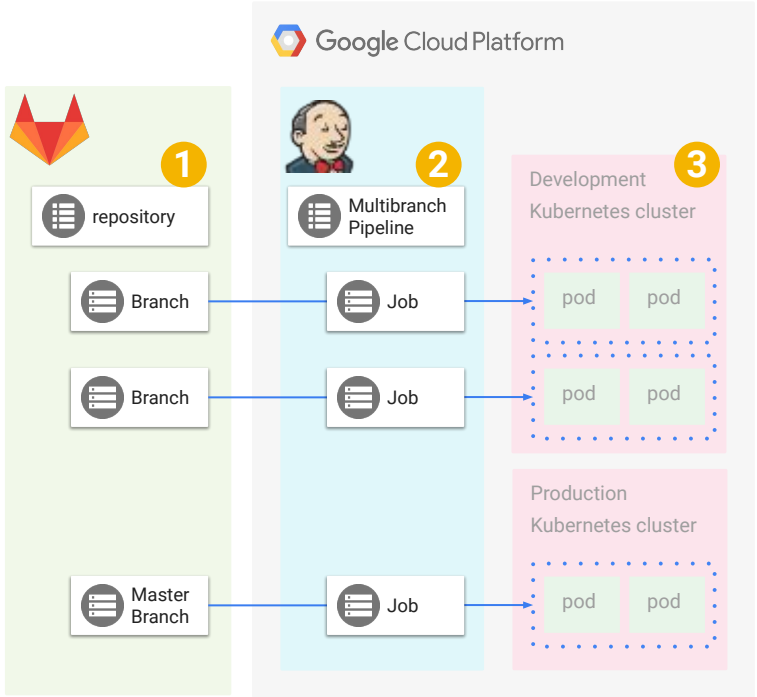
監視

- マネージドサービスを信頼した監視
  - Kubernetes配下にあるノードは重視しない
  - ノードCPUが高負荷になればオートスケールする
- サービス全体として正常であることを要監視
  - アップタイム
  - トータルレイテンシ

ノードの増減等により構成が変わっても監視の再設定は不要になった

従来、ZABBIXを使用しておりこの部分が大変だった

アプリバージョンごとの開発環境



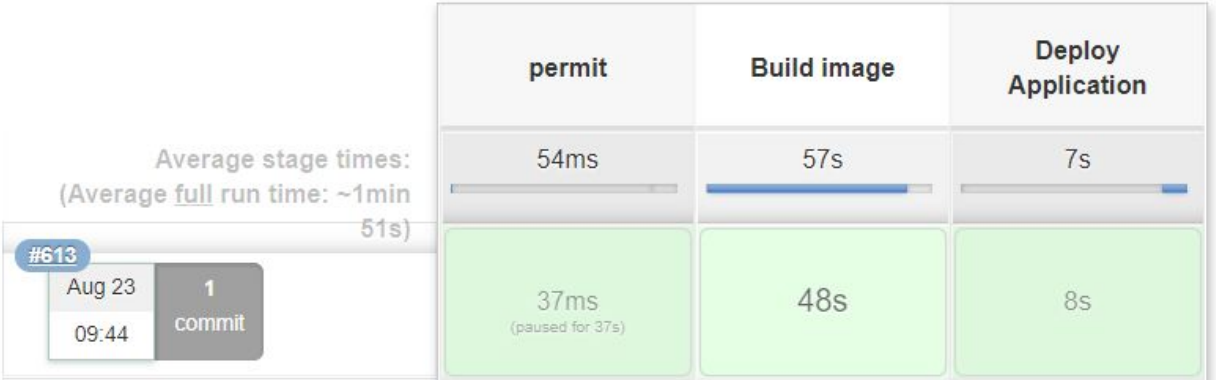
①アプリバージョンごとのブランチ

②各ブランチに対応したジョブを実行

③ブランチに対応した環境

アプリバージョンごとの開発環境

- 環境変数を利用してブランチに合わせたマニフェストファイルを実行
  - sh("kubectl --namespace=development apply -f conf/k8s/deployments/${env.BRANCH_NAME}.yaml")
- JenkinsからGoogle Cloud Build、kubectlを実行
  - Jenkinsでコンテナのビルドからデプロイまで一括でやる
  - 利用頻度が高いのでボタン一つが楽
- 将来的にはコミット一発でデプロイまでできるようにしたい



その他の運用

- Kubernetes 自体のアップデート
  - VM の管理は自動化された
  - Kubernetes の更新が追加された！！
  - クラスタのアップデートは手動で開始
  - Pod をうまいこと構成してあるなら無停止でできるという
    - 我々のモバイルアプリでは、今の段階では無停止で更新することはない

その他の運用

- バックアップ
  - Podは壊れたら作り直す、の方針でバックアップしない
  - CloudSQLなどのマネージドサービスは自動バックアップ
  - クラスタ作成時に使用するREST APIかgcloud コマンドラインはバックアップしておく

gcloud コマンドライン

これは選択したパラメータを含む gcloud コマンドラインです。

```
gcloud beta container --project "[redacted]" clusters create "standard-cluster-1" --zone "us-central1-a" --no-
enable-basic-auth --cluster-version "1.12.8-gke.10" --machine-type "n1-standard-1" --image-type "COS" --disk-type
 "pd-standard" --disk-size "100" --scopes "https://www.googleapis.com/auth/devstorage.read_only","https://www.googl
eapis.com/auth/logging.write","https://www.googleapis.com/auth/monitoring","https://www.googleapis.com/auth/service
control","https://www.googleapis.com/auth/service.management.readonly","https://www.googleapis.com/auth/trace.appen
d" --num-nodes "3" --enable-cloud-logging --enable-cloud-monitoring --enable-ip-alias --network "projects/neon-net-
203702/global/networks/default" --subnetwork "projects/[redacted]/regions/us-central1/subnetworks/default" --d
efault-max-pods-per-node "110" --addons HorizontalPodAutoscaling,HttpLoadBalancing --enable-autoupgrade --enable-au
torepair
```

まとめ

The Future Is Now
SNK®

結論

- 旧環境のサーバーサイドのプログラムと構成を基に、Kubernetes を活用した環境を構築した
- オートスケールを中心に、環境の再現性をふくめ多くのメリットがある
- オートスケールという要素が増えたことを考慮しても運用は以前よりはるかに楽になった
- 開発時の使い方や、コンテナの仕組みにあわせてプログラマから協力を得る必要あり
- Kubernetes 環境は変化が激しい
    - 来年には今よりさらに使いやすくなっているはず

謝辞

グーグルクラウドの皆様

環境構築に関して様々なサポートを頂戴し大変感謝しております

弊社開発担当の皆様

Kubernetes on GCPの環境にプログラムを合わせてくれたおかげで

新しい構成の実現に成功しました


The Future Is Now
SNK
R