安定したゲームリリースに導いたドラクエタクトの負荷テスト手法

野下 洋
株式会社 Aiming

自己紹介

- 野下 洋
- 株式会社 Aiming（2013 年入社）
  - 事業支援部 インフラ

Aiming について

Aiming について

Aiming は、オンラインゲームの企画・プロデュース・開発・運営を行う会社です。

Aiming について


CARAVAN
STORIES


DRAGON QUEST
TACT
© 2020 ARMOR PROJECT/BIRD STUDIO/SQUARE ENIX All Rights Reserved.


戦国大河


剣と魔法のログレス
いにしえの女神

Kubernetes 採用の経緯

Google Cloud 利用実績

Google Cloud
利用タイトル
※ サービス終了タイトル含む

Compute Engine メインで構成したタイトル




Google Kubernetes Engine (GKE)
利用タイトル




2012 2013 2014 2015 2016 2017 2018 2019 2020

GKE Release

Tokyo Region
Release

Kubernetes (GKE) の採用の主な理由

Docker を利用したゲーム開発

- プロジェクトメンバー (開発、QA、企画がメイン) 個人PCの開発環境化
  - GitHub + Unity + Docker で実現できた
  - 複雑な GitHub ブランチ運用が実現できた
  - 先行した実装のチェックが容易に可能となり、早期チェックを実現できた

- 本番環境も Docker (コンテナ) 化へ
  - 当時は、Kubernetes を採用した企業が増えてきており、魅力的なプロダクトだったので、すでにリリース済みのゲームを GKE 化したことからはじまった

Kubernetes (GKE) 化して特に感じたメリット

- ソースコードのデプロイ環境を容易に構築できる
  - 作成したイメージをコマンド一つで、サーバに展開できる
  - ローリングアップデートなどのデプロイ手法が k8s から提供されている

- Infrastructure as Code を実現できる
  - Infrastructure as Code が強制される
  - yaml の管理は、kustomize を利用している

- 死活監視 と 自動復旧
  - 死活監視と自動復旧を容易に実現できる

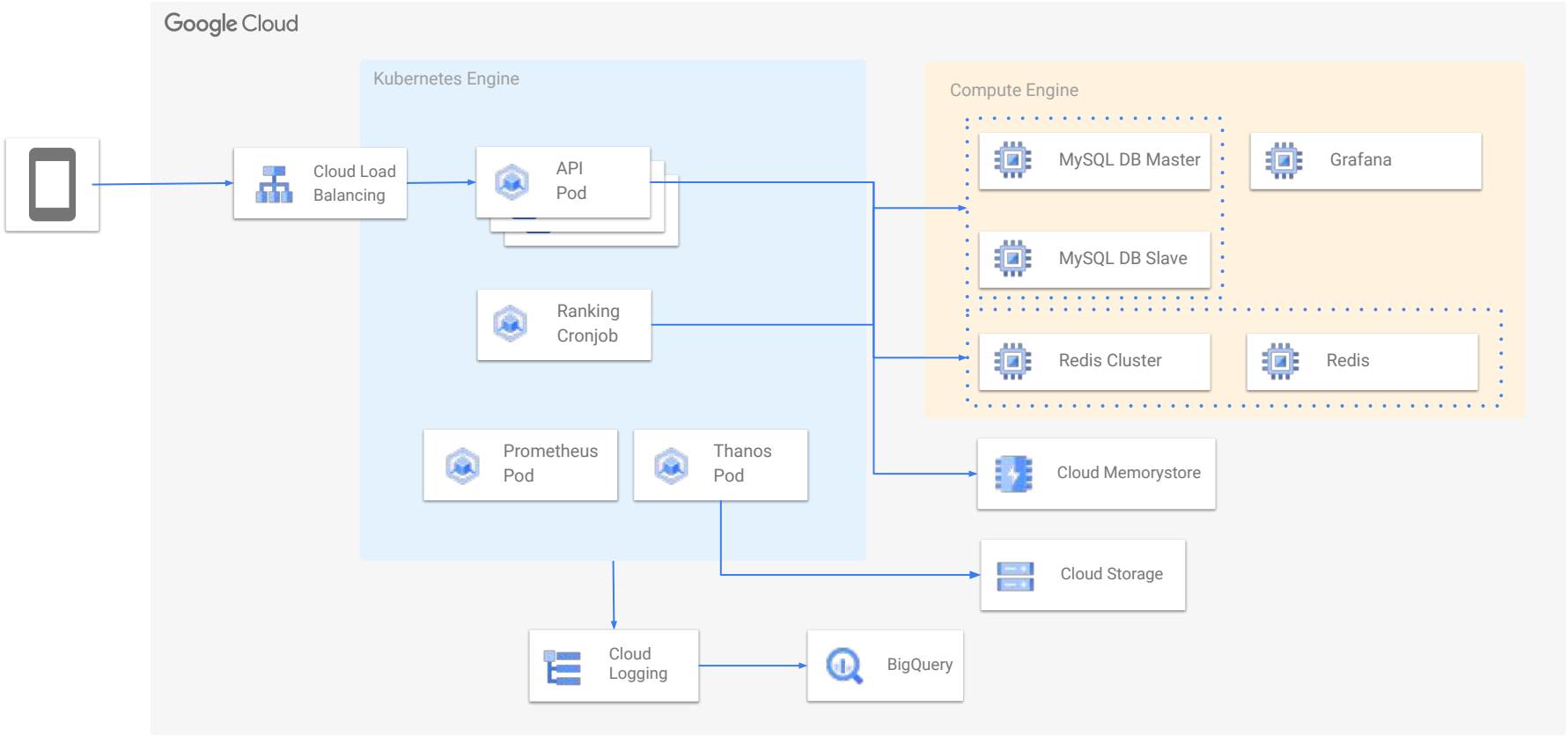
- ログ管理
  - 特にアプリケーションのログを標準出力に出力するだけで、容易に収集できる

Kubernetes 運用して大変だと感じるところ

- GKE Cluster の定期的なバージョンアップ運用
  - Cluster と Node を定期的にバージョンアップする必要がある
  - Cluster は自動で上がっていくが、Node のバージョンは手動でアップデートしている
  - Blue Green デプロイで Node Pool を別にすることで、ゲームのメンテ前に使われてない Node Pool をアップデートできるようにしている

- 学習コストが高い
  - できることが次から次へと増えていくことや構成が複雑なので、学習コストが高いです
  - 導入することで、運用面や構築面で様々なメリットを得られますが、安易に導入するでのはなく、メリットとデメリットを確認してから導入することをお勧めします

サーバ構成図


Google Cloud
Kubernetes Engine
Cloud Load Balancing
API Pod
Ranking Cronjob
Prometheus Pod
Thanos Pod
Compute Engine
MySQL DB Master
Grafana
MySQL DB Slave
Redis Cluster
Redis
Cloud Memorystore
Cloud Storage
Cloud Logging
BigQuery

サーバー構成について

- API Pod を役割に応じて、デプロイメントを分離する
  - コスト削減 (Newrelic)
  - DBへの Connection数の削減

- API Pod は、Blue / Green で Node Pool を分離する
  - Blue Green デプロイに対応するため

- DB や Redis は、Compute Engine で自前で構築
  - 性能を重視する DB / Redis については、運用実績豊富な Compute Engine を利用しています

負荷テスト

負荷テスト準備 (クライアントツール)

負荷テストクライアントツールの選定

- 負荷テスト クライアントツールの要件
  - 何度も負荷テストを実施するので、シナリオの作成を容易にできる
  - 負荷テストのユーザー規模の変更を容易にできるように、 GKE と親和性が高いツール

GKE と相性がいい、Locust を検証してみることにした

Locust について

- Python で自由にシナリオをかける
- GKE との相性がよく、容易にクライアント数を増やせる
- ブラウザで負荷テストの開始や終了やユーザー数を調整などの操作ができ、結果がリアルタイムに反映される

シナリオ作成の課題

開発している中で、段階的な負荷テストを行いたいので、シナリオの作成難易度を下げたい。。

- 独自機能の実装
  - Unity Editor 上でゲームをプレイした際の、送信リクエストの情報を記録し、その結果からシナリオを作成する独自の機能を実装した

シナリオのランダム要素について

- リリース初期を乗り越えるということにフォーカスした負荷テスト
  - リリース初期のユーザーの動きは同じ動きになりやすいので、ランダム要素がない負荷テストにした
  - ランダム要素がない負荷テストシナリオは、作成が容易

負荷テスト準備 (Prometheus + Grafana + Thanos)

- サーバリソースの可視化
  - サーバの CPU 等の負荷は、Grafana + Prometheus で計測した

- 過去のサーバリソース使用率などの比較
  - オブジェクトストレージ (GCS) に過去のデータを格納することができる Thanos を採用した

負荷テスト準備 (Sentry)

- API サーバのエラーの可視化
  - API サーバのエラーログの可視化と通知に Sentry を利用した

負荷テスト準備 (NewRelic APM)

- APIサーバのパフォーマンスの可視化
  - 各負荷テストのパフォーマンスを比較するために利用した

- レスポンスが悪化した際の原因調査
  - API のどの部分が遅かったのか？特定するために利用した

負荷テスト(スケジュール)

	2019		2020						
	11 月	12 月	1 月	2 月	3 月	4 月	5 月	6 月	7 月
マイルストーン					**CBT**				**リリース**
単体 API サーバ 負荷テスト									
ミドルウェア 負荷テスト									
負荷テスト (中規模)									
負荷テスト (大規模)									
障害テスト									

単体 API サーバ負荷テスト

1 台の API サーバに対しての負荷テスト

- 各 API のレスポンスタイムを確認し、重い API を特定し、改善する
- 1台でどのくらいのユーザー数を処理できるか、確認する
- API サーバの台数を算出し、規模が大きい負荷テストの費用を見積る
- MySQL の Slow query の確認
  - my.cnf の log_queries_not_using_indexesを有効にする
  - Slow query ログの閾値を下げる。(弊社では、0.1 sec にしている)

ミドルウェア負荷テスト

各ミドルウェア単体の負荷テストを実施

- Redis
  - Redis キーのデータサイズが大きくなると性能に対してインパクトが大きいので、現在のサイズを確認し、Memtier_benchmark というツールを用いたテストを実施した

- MySQL
  - tpcc-MySQL を用いたテストを実施した (以前の結果と比較することがメイン)

- Grafana + Prometheus + Thanos
  - Pod の数を増やし、性能に対する影響がないことを確認する
  - Grafana で 過去の古いデータを閲覧してもレスポンスが許容範囲であること

中規模ユーザー負荷テスト

数万ユーザー規模での負荷テストを実施

- APIのレスポンスタイムの再確認 (規模が大きくなったことによる劣化がどの程度か確認)
- 各サーバの負荷状況を確認
  - 負荷が高いポイントを確認する
  - スケールが難しいサーバがあった場合、負荷の状況次第では、仕組みの変更を検討をする

- DB / Redis サーバへのコネクション数の確認
  - サーバの実装によって、DB / Redis への接続数が大きく変わるので、実装を確認し、 DB / Redis 側の接続数を調整する

- Redis のメモリ使用量の確認
  - 負荷テスト中にメモリ使用量や各キーに対して ttl が設定されているか確認する

大規模ユーザー負荷テスト

Locust を利用し、数十万ユーザー規模での負荷テストを実施

- 中規模負荷テストと同じ内容を再確認
- 想定以上の規模のアクセスに対する対処方法明確にする
  - 負荷が高いサーバーのスケール方法など、対処方法を明確にする
  - 負荷が高いゲームの機能があった場合、一時的に停止できるようにする
- ユーザーデータの増加による SQL の劣化を確認する

障害テスト

- 負荷テスト中にサーバ (OS) を停止・起動
  - 実際のスマフォ端末で、ゲームプレイに支障がないことを確認する
  - 一時的な障害が発生した後でも、復旧できるか確認する

- 主なテスト内容
  - DB や Redis Cluster の Master および Slave の1台を停止・起動させる
  - API Pod の 1台を kubectl delete pod コマンドで削除するなど

負荷テストの目標値の決定について

- サンプルデータを取得し、ロードバランサーの最大リクエスト数 (rps) を確認し、ユーザー数との比率で算出した値を目標値としています
  - ユーザー1人当たりのロードバランサーに対するリクエスト数をテスト環境で確認する
  - 社内テストで、100ユーザーあたりのリクエスト数を確認する
  - CBT で、規模が大きなテストを実施した際のロードバランサーの最大リクエスト数が高い時点でプレイしているユーザー数を確認する

発生した問題と対策

内部 DNS (kube_dns)

- kube_dns への負荷対策
  - DB Redis など Compute Engine 上で動かしているサーバについては、hostalias を利用する
  - Cluster 作成時に、Node Local Cache を有効にする

- kube_dns 数の調整
  - 負荷対策後、負荷が低くなったが、デフォルトだと、サーバー台数に応じて、 kube_dns の数が増えるので、無駄に増やさないように最大数を設定して、台数が多くならないように調整した

Grafana（Prometheus）の負荷対策

- Pod 数が増加による、Grafana の遅延対策 (Grafana経由のPrometheusのクエリが複雑で重い)
  - レコーディングルールを利用することで、Prometheus が監視データを取得するタイミングで必要なデータを算出することで、Grafana のクエリを軽量化
  - デフォルトではたくさんのデータを Prometheus は取得しているが、必要最小限のデータを取得するようにした

Pod の起動時間について

- Loadbalancer は、NEG を採用しており、全ての pod に対してヘルスチェックが通るまで時間が長い。(30分くらいかかっていた。)
  - リリース後は、Blue Green デプロイを採用したため、事前に起動できるようにした
  - 負荷テスト時は、毎回サーバを全停止していたので、準備に時間がかかってしまった

今後挑戦してきたいこと

- CIと連携した小さい規模の負荷テストを継続的に実施する
- 数日レベルの長時間の負荷テストを実施する

まとめ

まとめ

- 会社として複数タイトルで利用することで、経験値をためる
- 負荷テストを早期から繰り返し実施すること
- 負荷テストのためのツールの準備
- 負荷テストの目標値の設定も重要