Google Cloud

SmartHR が少人数で Google Cloud へ (ほぼ) 全移行を完了させた方法

株式会社 SmartHR エンジニア

藤村 宗彦

自己紹介

藤村 宗彦
株式会社SmartHR
エンジニア

2018 年に SmartHR 入社後、従業員連携プロダクトを開発する傍ら、EM、インフラストラクチャーの管理、社内統制といった様々な管理・統制整備対応に従事

今回の GoogleCloud 移行プロジェクトをリード

話すこと

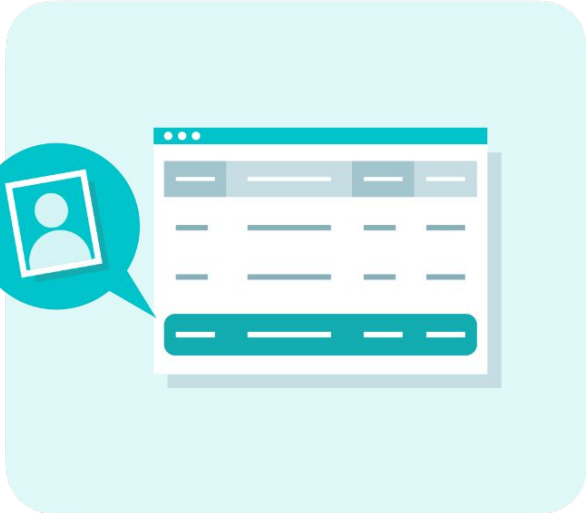
1. 弊社のサービスについて
2. 移行前のアーキテクチャーと課題について
3. 移行後のアーキテクチャーについて
4. 少人数の移行でやったこと
5. 今後の展望

弊社のサービスについて

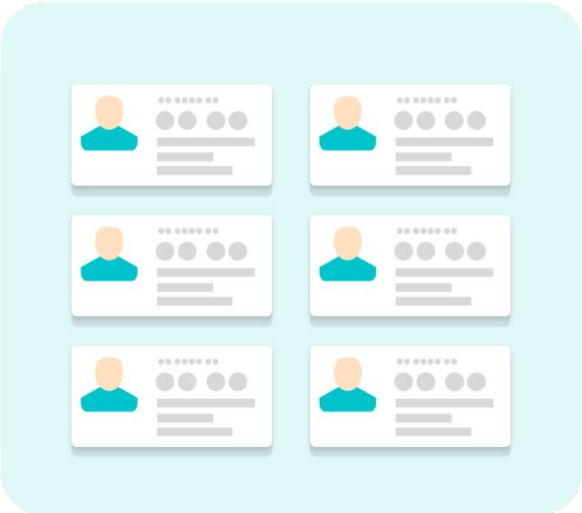
SmartHR は人事・労務業務の効率化を通じて

生産性の向上・働きたい職場環境の創出を目的とした

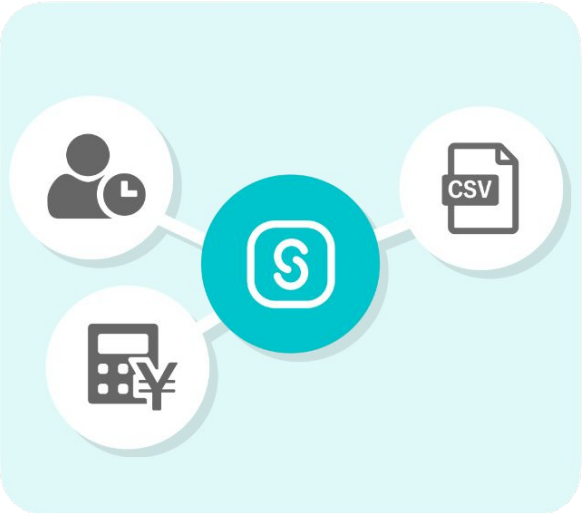
クラウド型ソフトウェアです。



人事・労務の業務効率化



人事データの一元化



人事データの活用

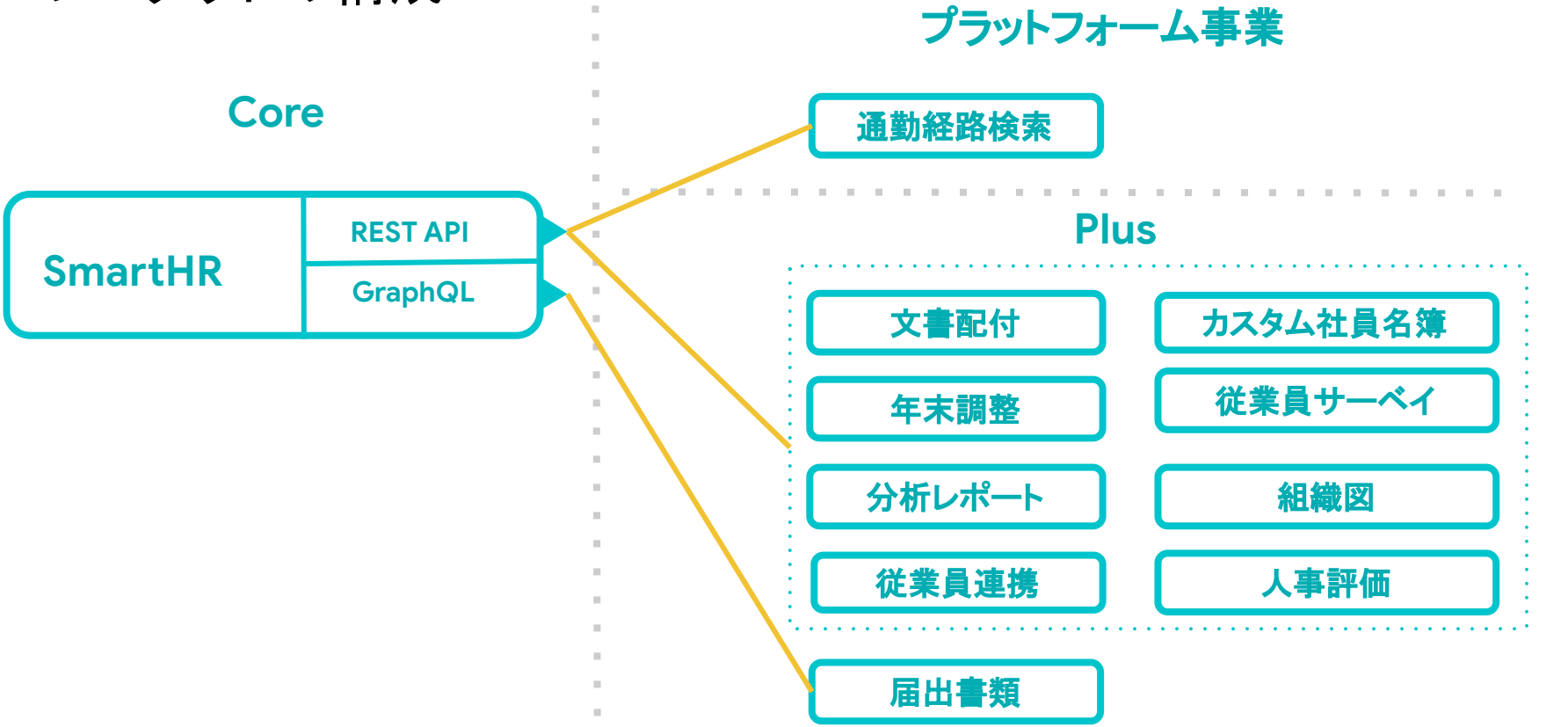
主な機能

集まる

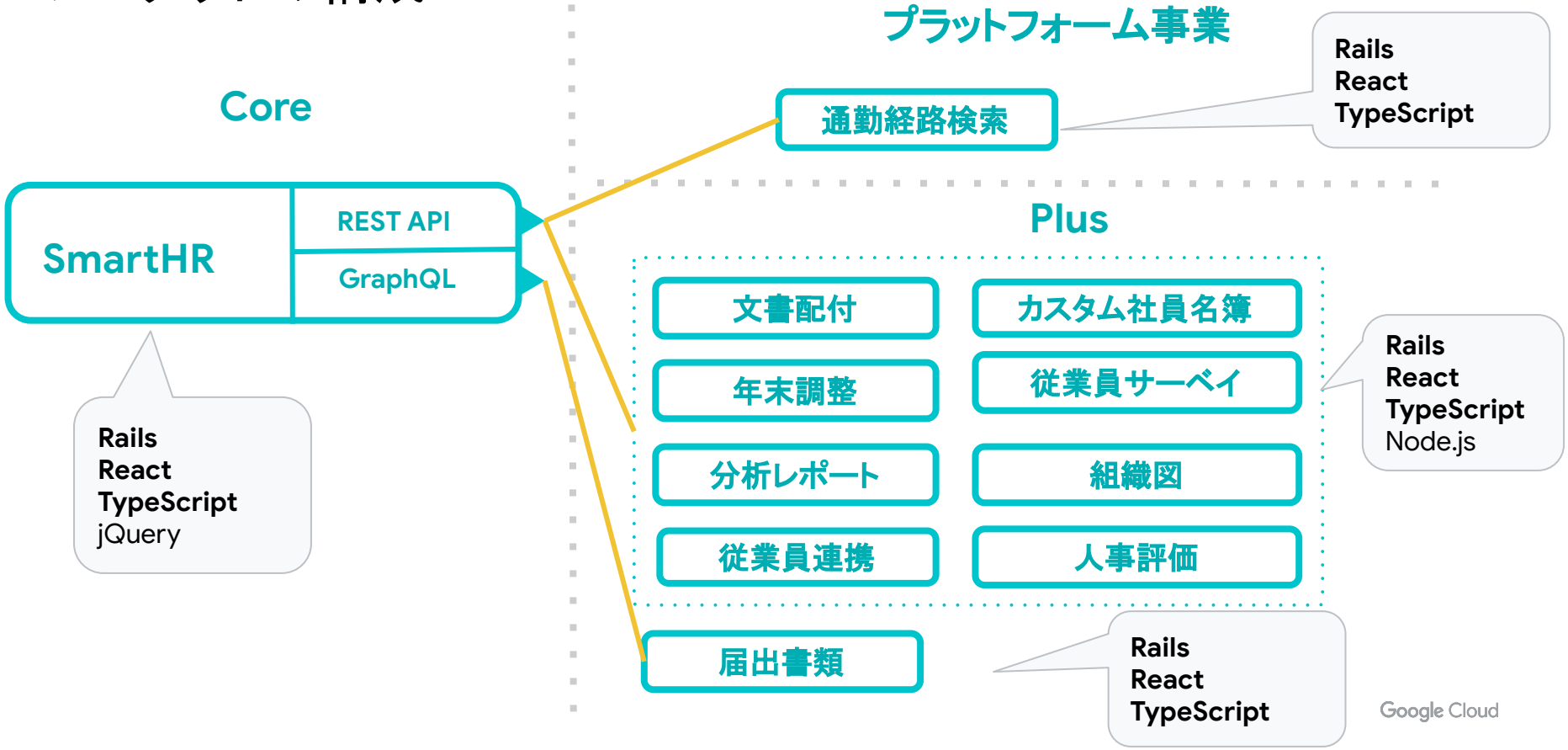
蓄まる

活用できる

プロダクトの構成

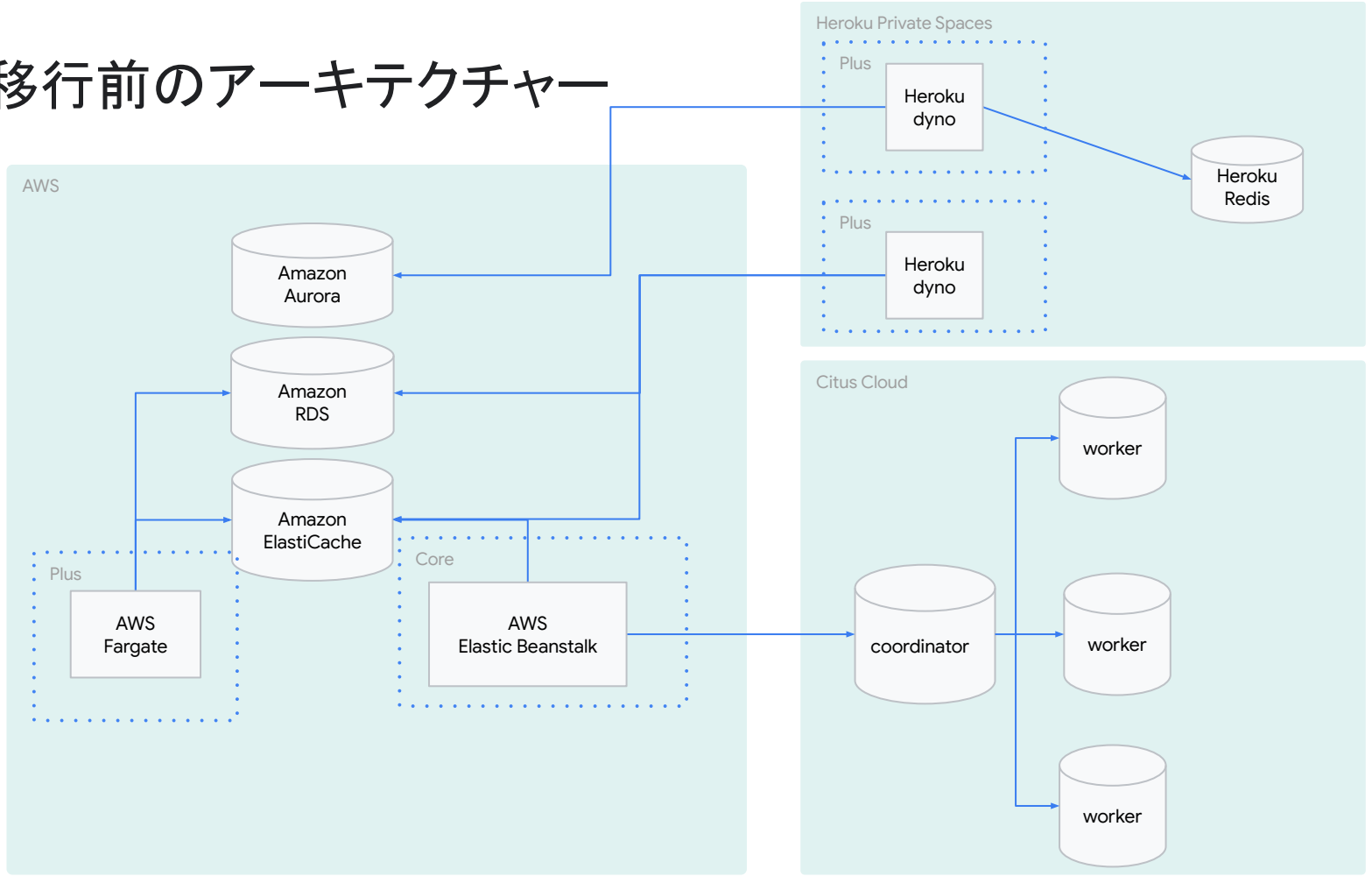

プラットフォーム事業
Core
SmartHR
REST API
GraphQL
通勤経路検索
Plus
文書配付
カスタム社員名簿
年末調整
従業員サーベイ
分析レポート
組織図
従業員連携
人事評価
届出書類

プロダクトの構成



移行前のアーキテクチャーについて

移行前のアーキテクチャー



移行前の 5 つの課題

01 | Citus Cloud (DBaaS) のサービス終了に伴う対応

02 | SOC2 Type1 取得による内部統制の対応

03 | プロダクト横断利用で適切な権限分離がなされていない AWS

04 | インフラの構成管理

05 | 統一感の無いインフラストラクチャーの運用

Google Cloud を採用した 5 つの理由

01 | 導入実績の豊富さやコンテナ化されたサービスが扱いやすい

02 | プロダクトの開発や運用支援ツール、API が十分に揃っている

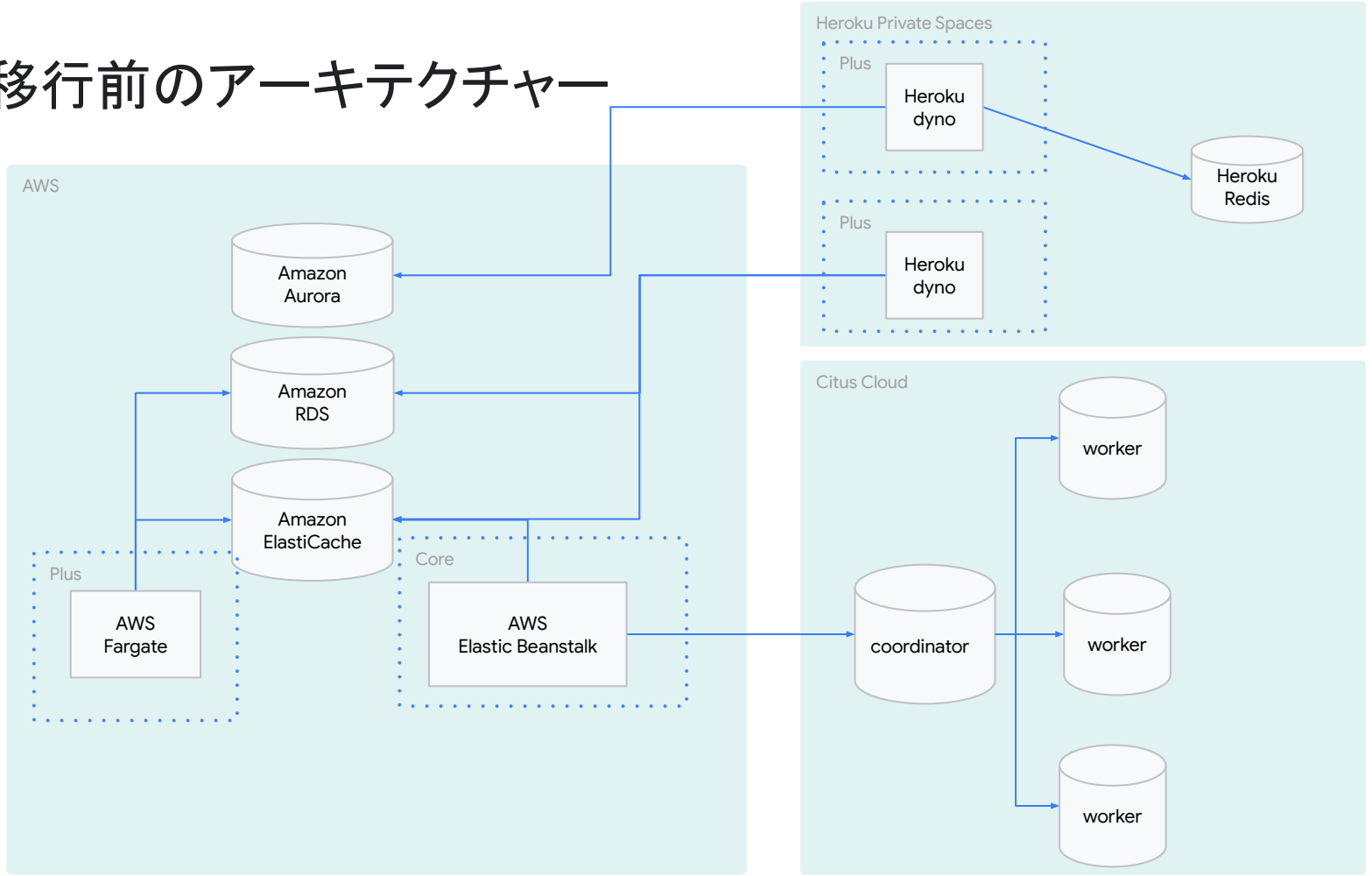
03 | アカウントの管理・権限などの統制が非常にやりやすい

04 | 社内利用の分析系ツール・基盤との親和性が非常に高い

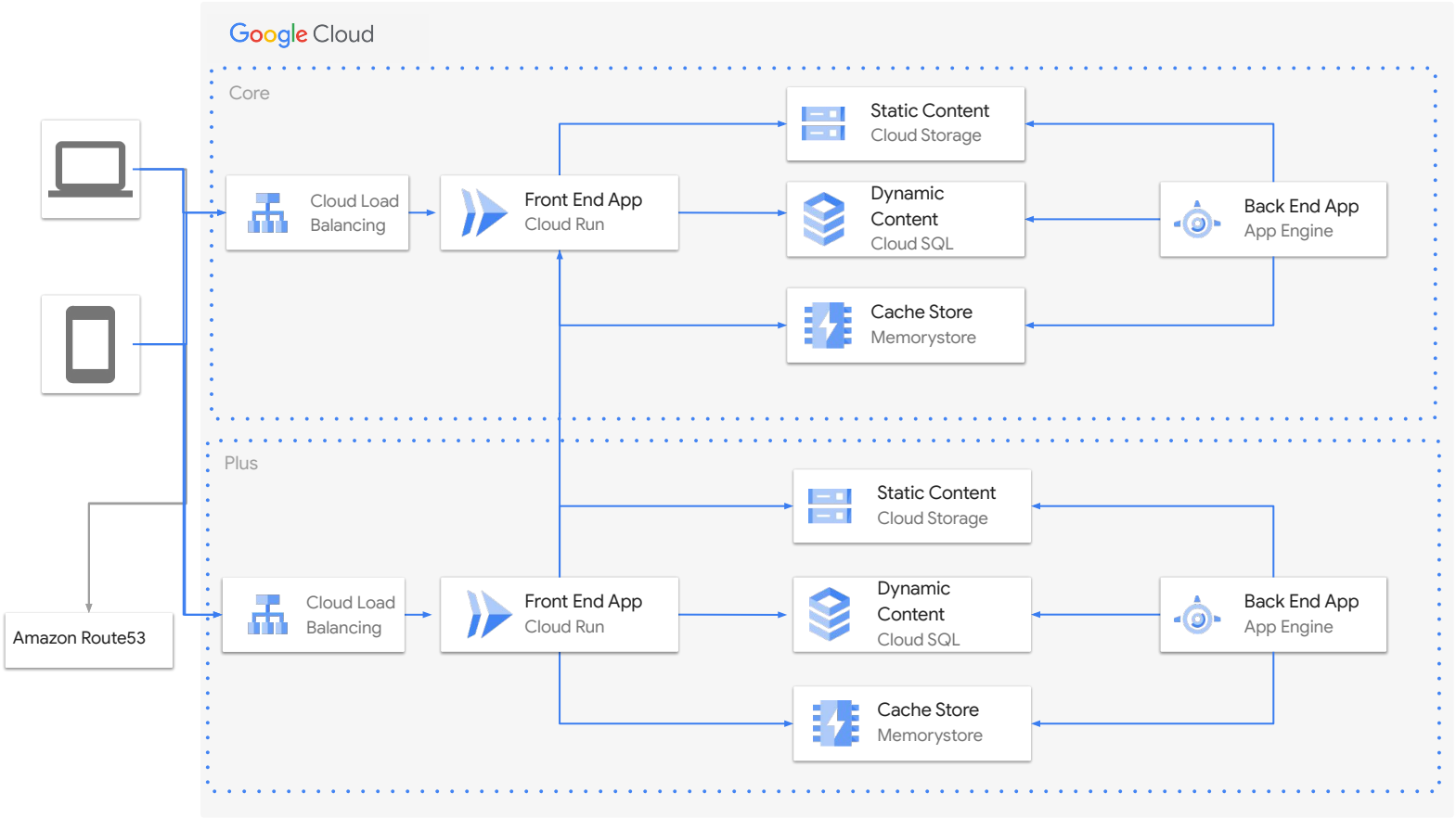
05 | NW の柔軟性が非常に高い

移行後のアーキテクチャーについて

移行前のアーキテクチャー



移行後のアーキテクチャー



主に採用したもの

Cloud Run

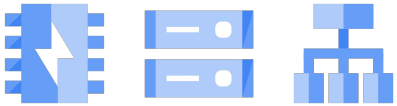
- Front End App
- Rails アプリケーションをホスト

AppEngine (flexible)

- Back End App
- Sidekiq アプリケーションをホスト
- オペレーション用のSSH

Cloud SQL for PostgreSQL

- Dynamic Content の永続化



その他

- Cache Store に Memorystore for Redis
- Static Content に Cloud Storage
- LB に Cloud Load Balancing

Cloud Run を採用した 5 つの理由

01

シンプルな Rails アプリケーション

02

移行後もインフラストラクチャーの運用コストを下げつつ、統一的な運用ができる

03

サービスの特性上、スパイクアクセスでも十分にスケールできる必要があった

04

AppEngine に LB を付与した場合、サーバレスNEGをデプロイ毎に構築しなおす必要があり、デリバリーのフローがやや煩雑になる懸念があった

05

機能拡充ペースが早いだけでなく、学習コストもそれほど高くない

AppEngine を併用している 4 つの理由

01

移行検討時、Cloud Run の Always on CPU が無かったため非同期処理の実行環境として採用が難しかった

02

従業員情報の取り込みや書類データの作成といった高負荷な処理によって、長時間の処理実行やマシンパワーを要する処理があったため、Cloud Run は不向きだった

03

サービス運用や問い合わせ調査にあたり、サービス仕様上履歴データやビジネスロジックを介してデータを調査する必要があるため、REPL (SSH コンソール)が必須だった

04

ノウハウが多々存在するので、移行における不確実性や運用における学習コストの削減を見込める

GKE を採用しなかった 5 つの理由

01

Google Cloud に対する学習コストに加えて Kubernetes の学習コストが上乗せになり、移行の不確実性が上がることが予見できた

02

細かくカスタムできる反面、ネットワークやセキュリティ対応、キャパシティプランなど考えることや運用で対応することが非常に多くなるので、少人数チームでは対処が難しい

03

チームごとに様々なクラスター運用やカスマイズが想定できたため、構成の統一や人のスケールが難しい

04

Kubernetes 自体の機能やアップデートが非常に早いので、少人数のチームでは対応ができない

05

移行検討時、そもそもGKE Autopilot が存在しなかった

少人数の移行でやったこと

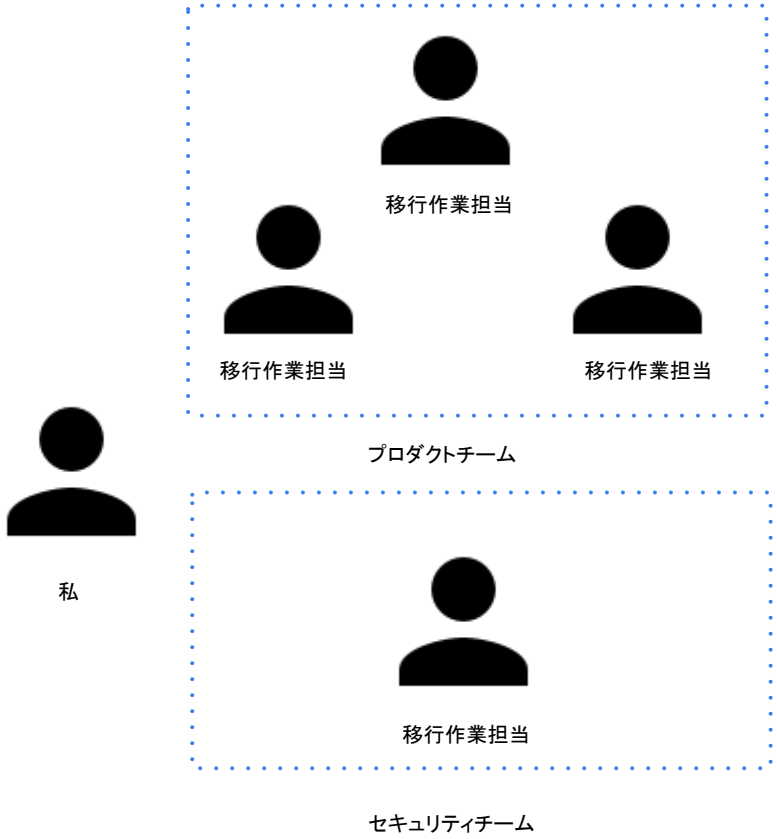
少人数チーム

01 | チーム体制

02 | チーム内の役割

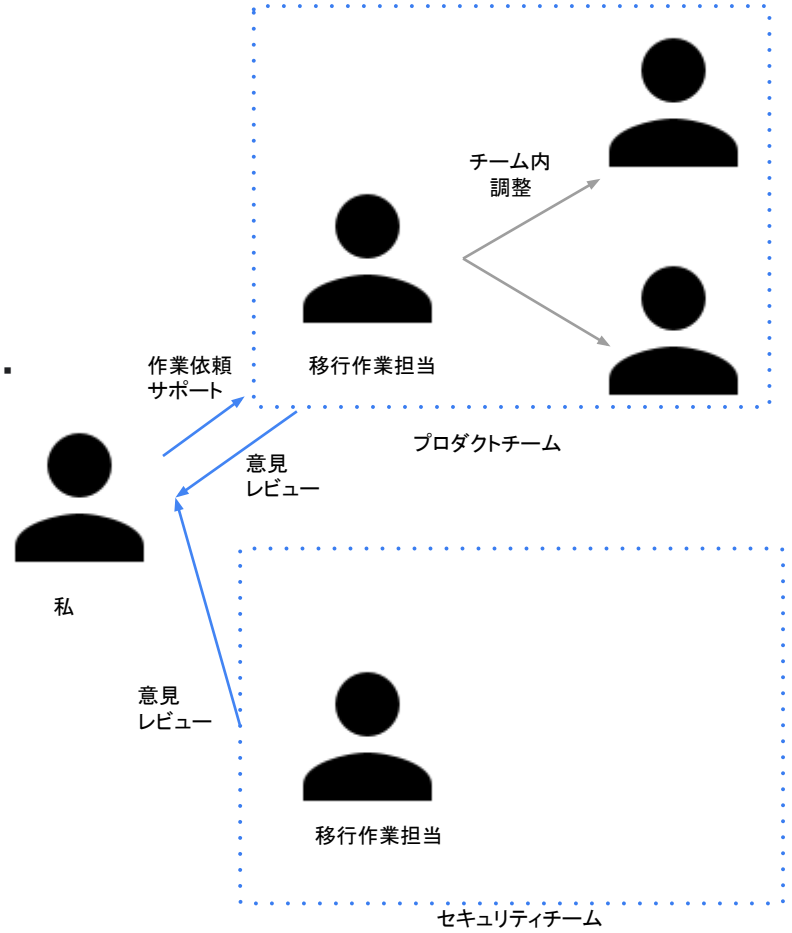
チームの体制

- 移行プロジェクトの作業の専任として、私
- プロダクトチーム内の移行作業担当（1~4 名）
- セキュリティチームの移行作業担当（1 名）



チームの役割

- 私
  - 全体の設計・構築・移行計画・サポートの対応実施
  - 各プロダクト/セキュリティチームから設計・運用の意見・レビューを吸い上げて反映
  - ハンズオンによる Google Cloud の知見共有
- プロダクトチーム
  - 移行計画を元に構築・移行作業
  - チーム内の調整を実施
- セキュリティチーム
  - セキュリティ設計・運用



移行までのスケジュール

2019 / 08

Citus Cloud の新規受付停止による
SmartHR Core 機能 のデータベース
移行検討

2020 / 10

Google Cloud へ移行決定

2021 / 01

Citus Cloud の終了アナウンスによって年内に
SmartHR Core 機能 の移行計画と対応必須化

2021 / 04 ~ 2021 / 12

従業員連携以外の SmartHR plus 機能
を随時 Google Cloud へ移行

2022 / 01 / 08

SmartHR Core 機能 の
Google Cloud へ移行

2022 / 01 /30

Citus Cloud 終了

2022 / 02 ~

SmartHR plus のうち従業員連携機能
を Google Cloud へ移行開始

対応内容

01 | 組織整備

02 | インフラの構成管理対応

03 | CI / CD の整理

組織整備

監査対応

- リソース変更を追跡できるよう監査ログの取得対応
- 長期保存できるようにエクスポート対応
- 監査ログの調査対応
  - セキュリティチームのみの開示
  - 閲覧・調査用ドキュメントの作成
- ログモニタリングと検知の対応
  - アクセス権限の変更
  - 不正リクエスト

リソース整備

- リソース階層の決定
- 組織ポリシーの決定
- 課金設定
- 乱立したリソースの削除・隔離対応
- プロダクト個別のプロジェクト作成による、アクセスと権限を分離対応

https://cloud.google.com/docs/enterprise/best-practices-for-enterprise-organizations#audit-trail
https://cloud.google.com/docs/enterprise/best-practices-for-enterprise-organizations#billing_and_management

インフラの構成管理対応

Terraform による構成管理

元々、構成管理の一部で使用

機能追従が早いことやコードレビューできるため、移行において全面的に導入

各 Google Cloud サービスコンポーネントごとにセキュリティ設定や有効が必須な内容をデフォルトで設定したモジュールを作成

利用側はパラメータを設定するだけで、社内推奨設定が有効な状態の環境構築ができるように対応

Terraform Cloud による適用

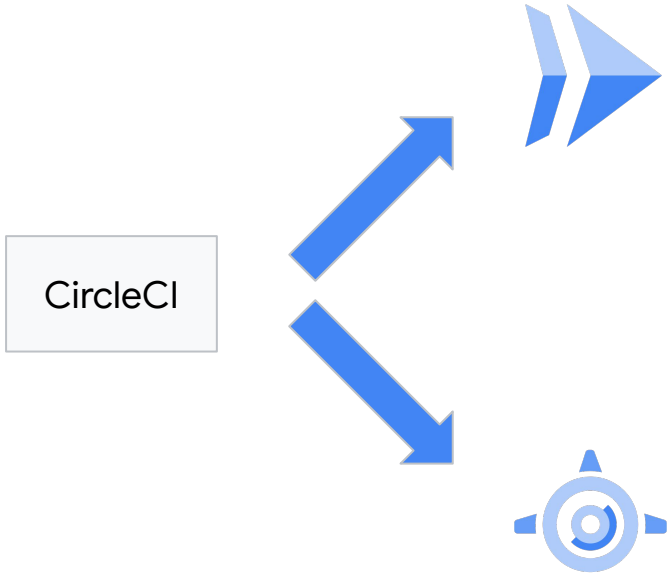
SOC2 Type1 を取得しているため、システムに対する変更・承認履歴が必須

加えて、Terraform を利用することで起きがちなStateの管理、実行のバッティング問題も解消できるため、導入

CI / CD の整理

- AWS / Heroku に向けたデプロイから切り替える必要があるので対応
- CI / CD に造詣のあるメンバーが、フローの型を用意
- 型を各プロダクトに展開することで、迷うことなく移行対応を実施

殆どのプロダクトで、デリバリーフローを統一



移行作業の内容

01 | Amazon ElastiCache / Heroku Redis から Memorystore for Redis

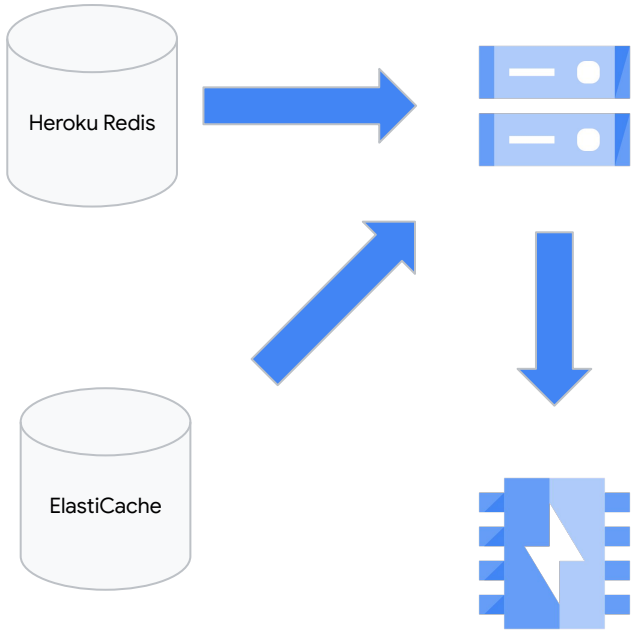
02 | S3 から Cloud Storage

03 | Amazon RDS から Cloud SQL

04 | Citus Cloud から Cloud SQL

Amazon ElastiCache / Heroku Redis から Memorystore for Redis への移行

- rdb ファイルのエクスポート機能が ElastiCache にあるので、こちらでエクスポートを実施
- Memorystore に rdb ファイル のインポート機能があるのでこれを使いデータを移行
- Heroku Redis には rdb ファイルのエクスポート機能が存在しない為、 redis-cli をセットアップして対処



https://docs.aws.amazon.com/ja_jp/AmazonElastiCache/latest/red-ug/backups-exporting.html
https://cloud.google.com/memorystore/docs/redis/import-data

S3 から Cloud Storage への移行

Storage Transfer Service による転送

S3 からの転送は全て Transfer Service を利用

約 1,000 万件、12TB の規模のバケットが約 2 時間で転送

一つのファイルが巨大な場合、転送に時間がかかるため、別の手段の検討が必須

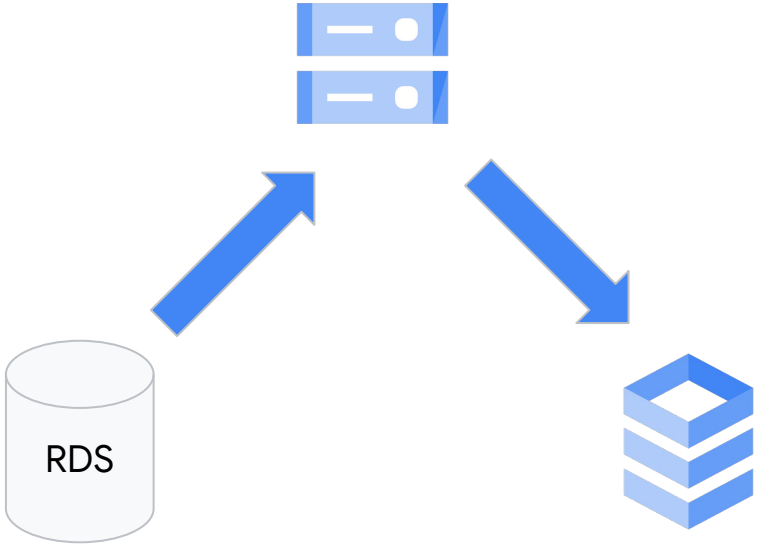
S3 Batch Operations と AWS Lambda による検証

転送後のファイルに破損や欠損が無いか確認

転送元オブジェクトを正とし、転送後の全てのオブジェクトに対して MD5 のチェックを実施

Amazon RDS から Cloud SQL への移行

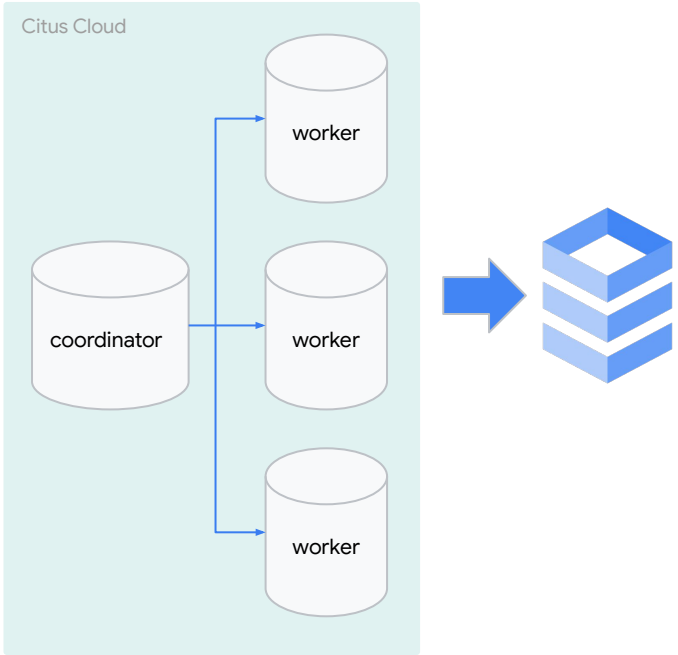
- 稼働中の RDS に操作が必要だったので Database Migration Service は未使用
- 一番シンプルで理解しやすい pg_dump を利用
- dump データを Cloud SQL のインポート機能を使用してデータを移行
- PITR やログのフラグを有効状態でインポートするとストレージサイズの増加やインポート時間がかかるので注意



https://cloud.google.com/sql/docs/postgres/import-export/import-export-sql#import

Citus Cloud から Cloud SQL への移行

- Cloud Spanner という案もあったが、懸念が残るため不採用
  - 設計やデータの変更が必須そうだった
  - 検討時、PostgreSQL interface や ActiveRecord の対応も無かった
- シャーディングが無い PostgreSQL でも動かせることが分かっていたので、CloudSQL を採用



Citus Cloud から Cloud SQL への移行

- pg_dump / pg_restore でデータ移行を実施
- coordinator にデータを集約できたが、実行時間とストレージ容量の問題で除外
- データ数が非常に多いため、予め CLUSTER コマンドで排他制約のインデックス順に揃えることで、pg_restore 時の I/O 処理を削減、高速化対応を実施
- パフォーマンスの劣化に関しては、想定リクエストの負荷試験、リソースの調整や実行計画の見直し、INDEX の貼り直しで対処し、劣化を最小限にするように対応
- テナントレベルのデータ分離は、アプリ層に「activerecord-multi-tenant」を利用することで対処

https://www.citusdata.com/blog/2021/02/06/citus-tips-how-to-undistribute-a-distributed-postgres-table/
https://www.postgresql.jp/document/13/html/sql-cluster.html
https://github.com/citusdata/activerecord-multi-tenant

運用のナレッジ

01 | Cloud Run で Rails アプリを稼働させるポイント

02 | Operations suite の活用

03 | Cloud NAT の注意点

Cloud Run で Rails アプリを稼働させるポイント

1. 高負荷によるコールドスタートを避けるため、最小台数に余裕をもたせる
2. スケールアウトを見越して最小台数と最大台数に幅をもたせる

https://cloud.google.com/run/docs/tips/general#using_minimum_instances_to_reduce_cold_starts

Cloud Run で Rails アプリを稼働させるポイント

Rails アプリは Puma を使用するため、
1 コンテナインスタンスのリクエスト数は
WEB_CONCURRENCY(CPU 数) と
RAILS_MAX_THREADS の値で決まります

Cloud Run の**同時実行数は、リクエスト数の範囲内に収めます**

```
threads_count = ENV.fetch('RAILS_MAX_THREADS', 5)
threads threads_count, threads_count
port ENV.fetch('PORT', 3000)
environment ENV.fetch('RAILS_ENV') { 'development' }
workers ENV.fetch('WEB_CONCURRENCY', 2)
worker_timeout 180
worker_boot_timeout 180
preload_app!
```

Cloud Run で Rails アプリを稼働させるポイント

本番稼働させる前の負荷試験

Rails アプリケーション	Cloud Run	Cloud SQL
○ WEB_CONCURRENCY : 2 ○ RAILS_MAX_THREAD : 24 ○ コネクションプール数 : RAILS_MAX_THREAD の値と同じ値	○ vCPU: 2 ○ メモリ: 8GB ○ 同時実行数 : 40 ○ 最小台数 : 15 ○ 最大台数 : 100	○ メモリ: 53GB ○ 最大コネクション 数: 600

コンテナが受け付けられるリクエスト数以内に同時実行数を設定

https://cloud.google.com/sql/docs/postgres/flags

Cloud Run で Rails アプリを稼働させるポイント

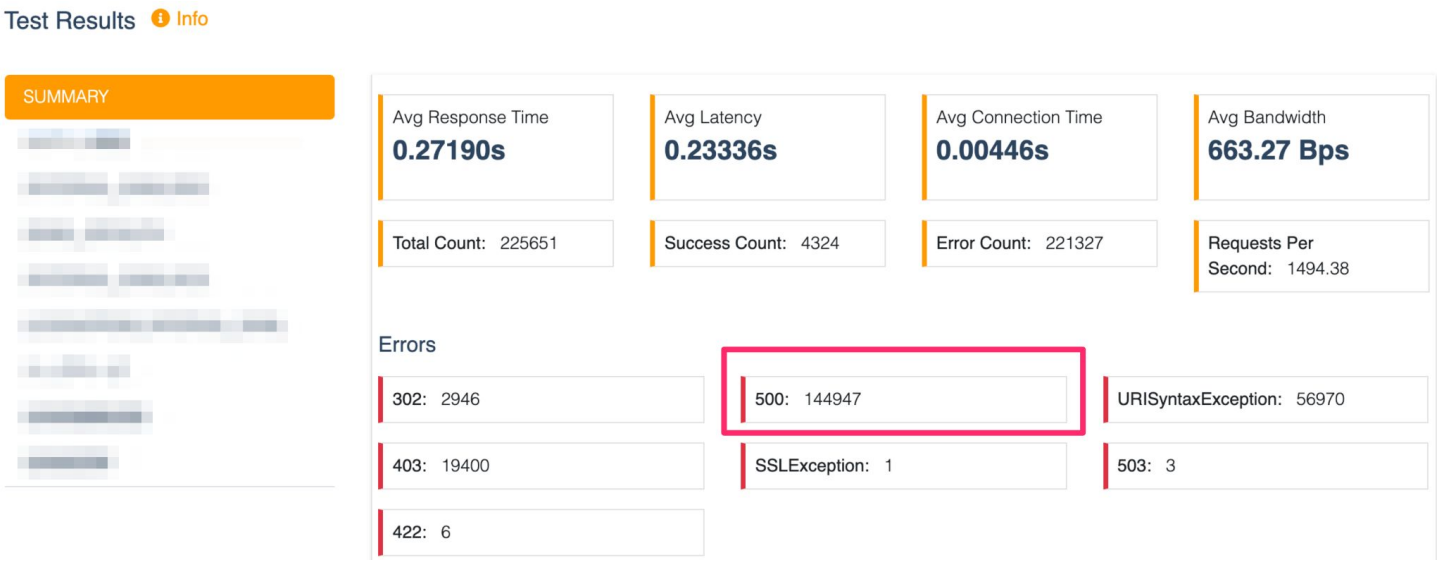
本番稼働させる前の負荷試験

Rails アプリケーション	Cloud Run	Cloud SQL
○ WEB_CONCURRENCY : 2 ○ RAILS_MAX_THREAD : 24 ○ コネクションプール数 : RAILS_MAX_THREAD の値と同じ値	○ vCPU: 2 ○ メモリ: 8GB ○ 同時実行数 : 40 ○ 最小台数 : 15 ○ 最大台数 : 100	○ メモリ: 53GB ○ 最大コネクション数 : 600

1 コンテナあたり 48 コネクションを Cloud SQL に対して貼る

https://cloud.google.com/sql/docs/postgres/flags

Cloud Run で Rails アプリを稼働させるポイント



1 コンテナあたりのコネクション数が多く、データベース側のコネクションが枯渇したことが原因

Cloud Run で Rails アプリを稼働させるポイント

1. Cloud SQL のコネクション数の上限を超えないようにする
    a. 1コンテナが同時に処理できる最大リクエスト数 (= 同時実行数)の値
    b. 最大台数の値

Cloud Run のインスタンス数は負荷に応じてオートスケールするため、
スケールする分も考慮してコネクション数・同時実行数を考えておく必要がある

https://cloud.google.com/run/docs/about-concurrency

Operations suite の活用

Cloud Monitoring

- モニターしておいたほうがよい項目のカスタムダッシュボードを用意
- アラートや外形監視も標準で用意されているので活用

Cloud Logging

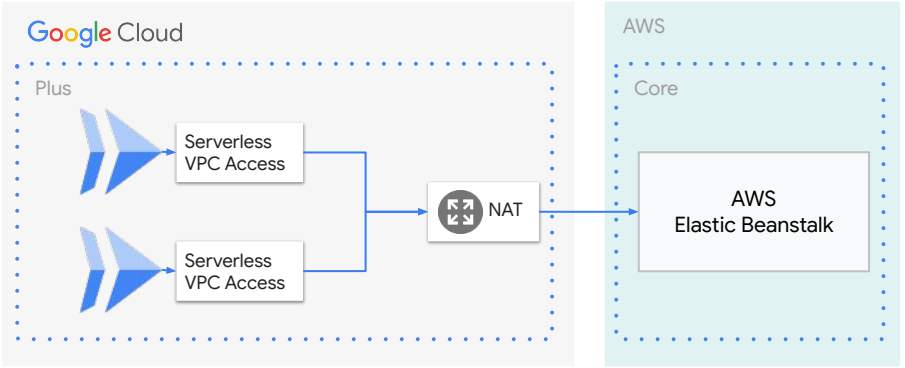
- プロダクトのログを全て集約
- 構造化ログになるようにログライブラリを自作

その他

- プロダクト横断で、状況を確認する際にNew Relic を活用
- アプリケーションエラー監視・通知にSentry を活用

Cloud NAT の注意点

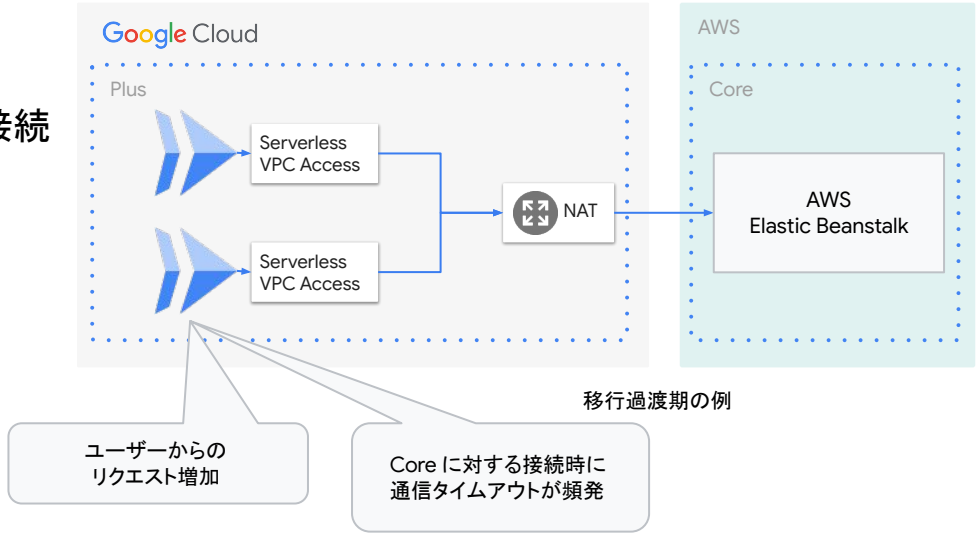
- Cloud Run から外部にアクセスする場合に Cloud NAT を使用



移行過渡期の例

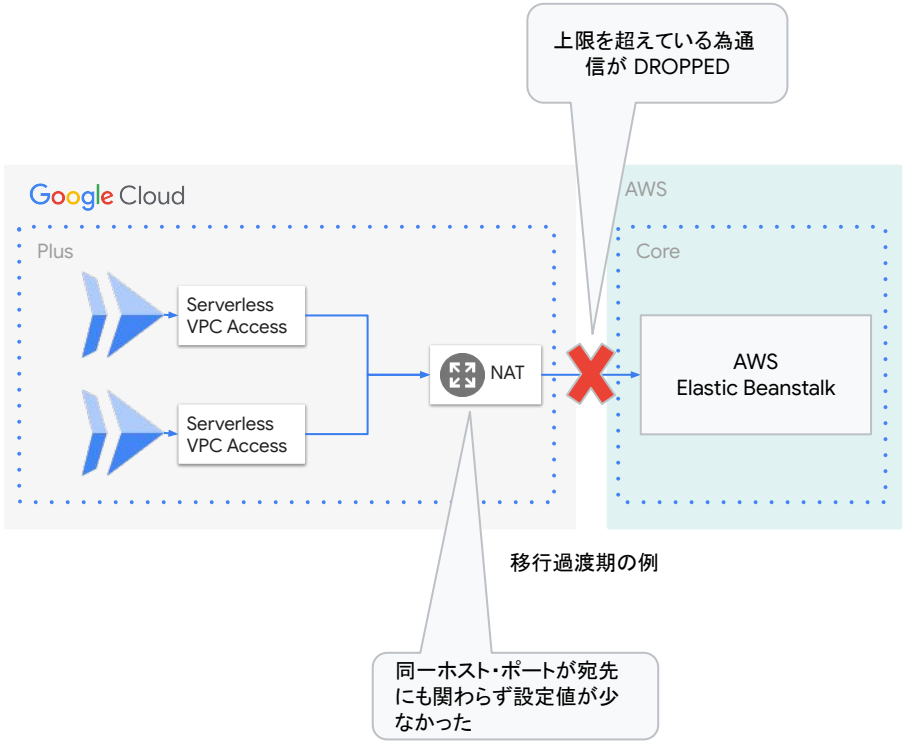
Cloud NATの注意点

- Cloud Run から外部にアクセスする場合に Cloud NAT を使用
- 移行過渡期に Plus から Core の接続に対して接続タイムアウトが頻発する事象が起きた



Cloud NATの注意点

- Cloud Run から外部にアクセスする場合に Cloud NAT を使用
- 移行過渡期に Plus から Core の接続に対して接続タイムアウトが頻発する事象が起きた
- 接続先が同じホストとポート番号の組に対して、デフォルト設定は「 VM あたり 最小 64 ポートしか使えない 」のが原因
- サーバレス VPC アクセス は最大 10 台までスケールするため、ドキュメントの「ポート予約手順」を参考に最適値を算出して解消



https://cloud.google.com/nat/docs/ports-and-addresses#port-reservation-examples

少人数での移行まとめ

- プロダクト/セキュリティチームと連携し、少人数の移行チーム体制を整えることで機能開発を止めることなく移行対応ができた
- プロジェクトを分割することで、プロダクトごとに権限管理ができるようになった
- インフラの構成管理とプロダクトの実行環境の統一ができた
- 移行のノウハウや運用のナレッジをプロダクトチーム間で共有したことで、ほとんどのプロダクトを無事に Google Cloud へ移行を完了できた
- ポイントを押さえることで、 Rails アプリケーションを Cloud Run で動かすことができた

今後の展望

今後の展望

- データを活用しやすいように分析基盤の再構築
- プロダクトのさらなる安定稼働のための QA 環境構築
- データベース周りの負荷軽減対策
- PostgreSQL の Row Level Security の導入
- コンテナ環境のさらなる改善


We Are Hiring!!
SmartHR 採用
検索

Thank you.

Google Cloud