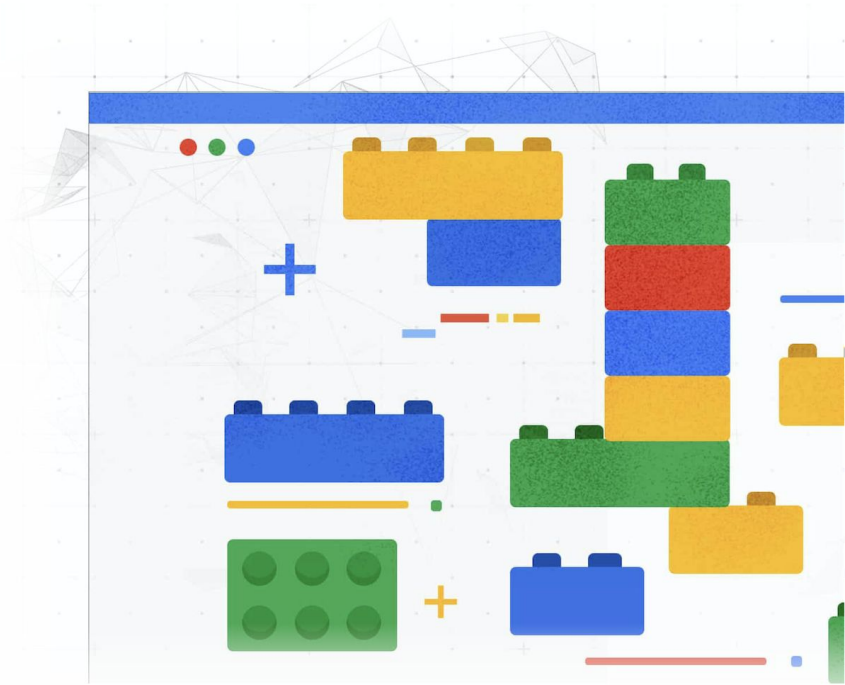
Cloud Run を
最大限使いこなすには
~パフォーマンス チューニング Deep Dive~

グーグル・クラウド・ジャパン合同会社
Application Modernization Specialist
塚越 啓介、頼兼 孝幸、篠原 一徳



Disclaimer

本セッションはデモと
トークが中心です。

アジェンダ

- ✔ Cloud Run を振り返る
- ✔ パフォーマンス チューニング デモ
- ✔ まとめ

Cloud Run を振り返る

01

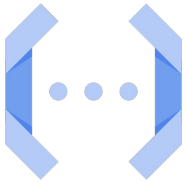
Cloud Run

Knative がベースのサーバーレス

サーバーレスのアジリティを
コンテナ化したアプリに

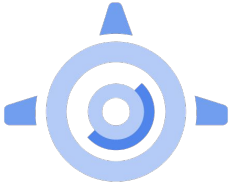
GCP のサーバーレス コンピュート

Functions



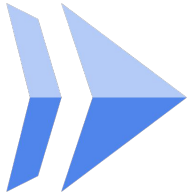
Cloud Functions

Apps



App Engine

Containers



Cloud Run

Cloud Run の主な特徴

高速なデプロイ

ステートレスなコンテナ

高速に 0 to N スケール

数秒でデプロイし URL を付与

サーバーレス・ネイティブ

管理するサーバーはなし
コードに集中

言語やライブラリの制約なし

きっちり使った分だけお支払い

高いポータビリティ

どこでも同じ Developer Experience
フルマネージでも GKE のクラスタ上でも

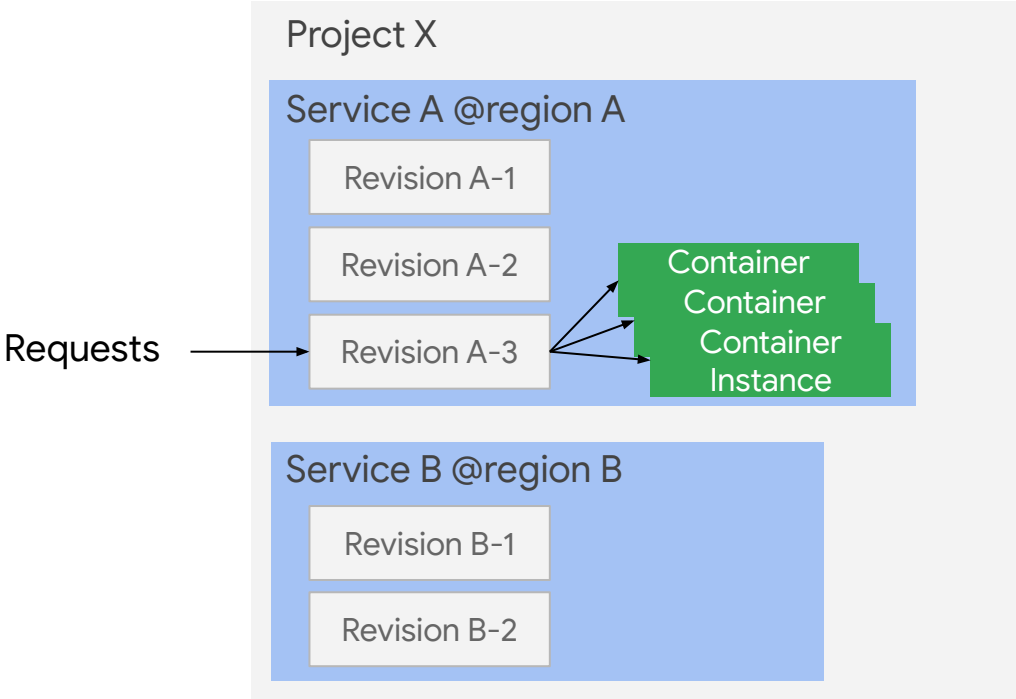
Knative API の一貫性

ロックインの排除

Cloud Run だとコンテナをシンプルに使える

Kubernetes	Cloud Run
コードを書く	**コードを書く**
$ docker build	$ docker build
$ docker push	$ docker push
$ kubectl apply -f deployment.yml	$ gcloud run deploy..
$ kubectl apply -f service.yaml	
$ kubectl apply -f hautoscal.yaml	
他にも監視・ロギングの設定とか..	

Cloud Run のリソースモデル



Service

Cloud Run の主リソース
Service 毎に Endpoint を提供
自動で設定される a.run.app ドメイン、
もしくはカスタム ドメインが選択可能

Revision

デプロイするごとに生成される
コンテナ イメージとデプロイ時に指定される
環境変数やパラメーターから構成される

Container Instance

実際にリクエストを受けるコンテナ、
リクエスト の数に応じて自動的にスケール

Cloud Run を使ってみたけど
期待したレスポンスタイムが
出ないという経験はないでしょうか？

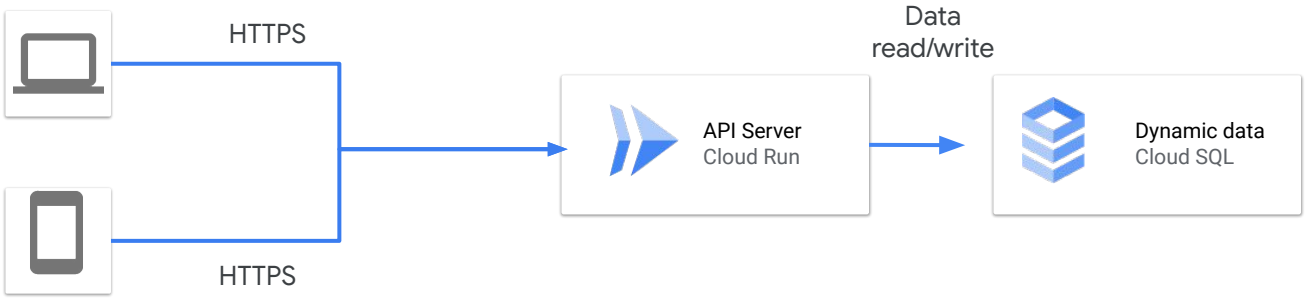
パフォーマンス
チューニング
デモ

02

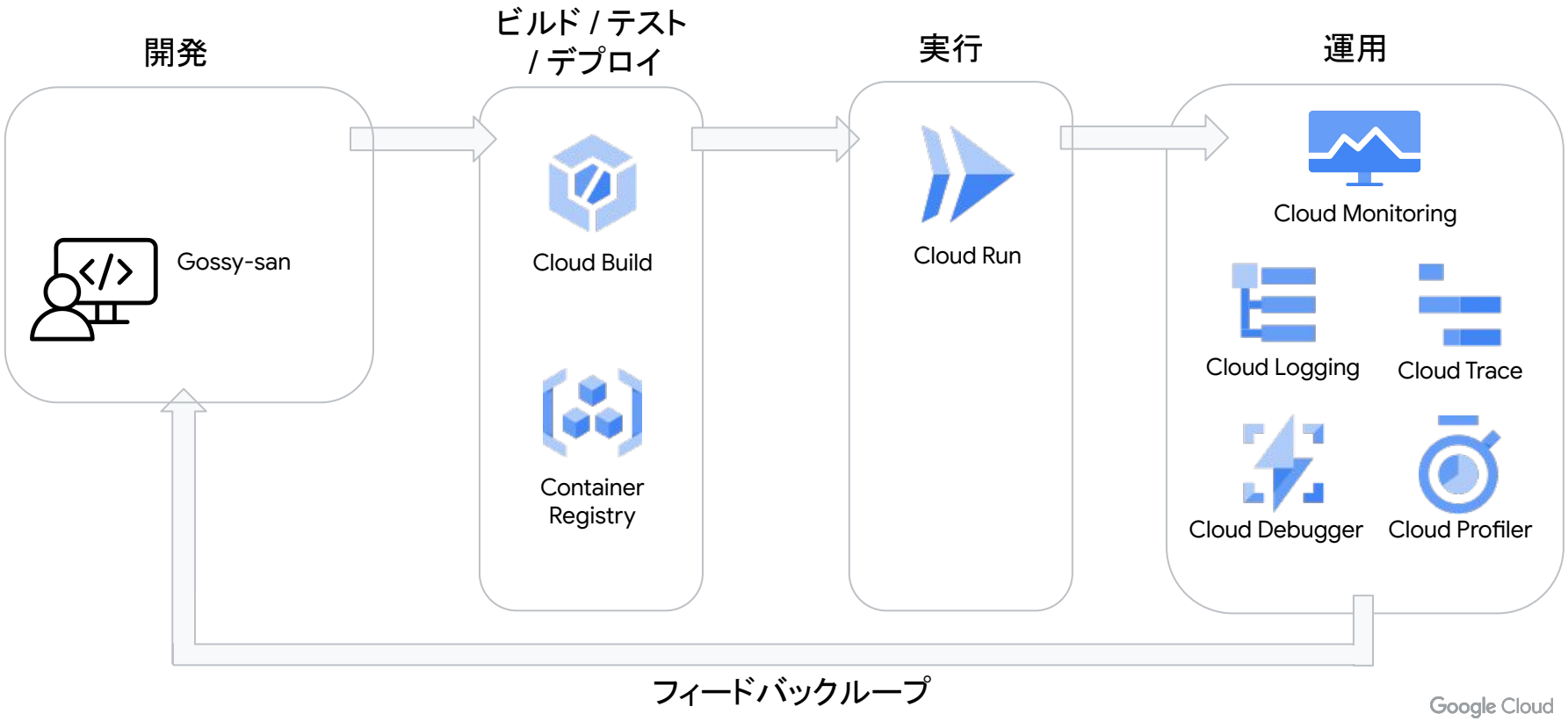
アンチパターンてんこ盛りサンプルアプリ構成

シンプルな Rest API を提供するアプリケーション

データベースは Cloud SQL を利用



ビルド、デプロイのワークフロー



パフォーマンス チューニング どこから手を付けていくか

1. Cloud Run の設定
2. アプリケーションの作り
3. DBの問題

本セッションで説明

1. Cloud Run の設定を見直していく

コンテナに最適な CPU / メモリ が設定されているか？

Cloud Run で設定できるコンテナのスペック

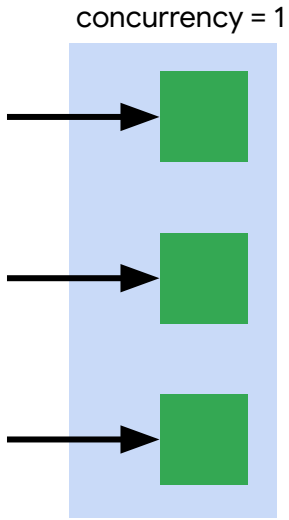
- CPU
  - デフォルト 1vCPU
  - 変更可、1vCPU, 2vCPU, 4vCPU から選択
  - 4vCPU の場合はメモリを 2GB 以上選択する必要あり
- メモリ
  - デフォルト 256 MB
  - 変更可、最小 128 MB 〜 最大 8 GB

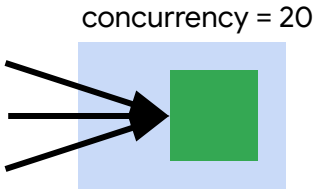
Concurrency（コンカレンシー）

Concurrency とは同時の 1 つの Container Instance に投げられるリクエストの最大数。

Google Cloud Functions など一般的な FaaS は
一度に 1 つのリクエストしかハンドルできない。
なので "concurrency = 1".

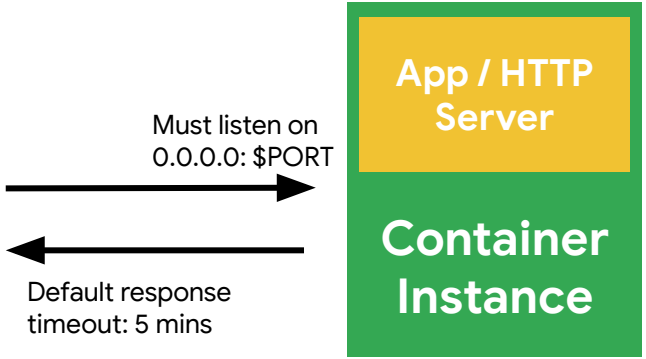
Cloud Run の場合、concurrency の値を 1 から 250 まで設定できる（default: 80）ので、1 つの Container Instance で同時に複数のリクエストを処理することができる。





レスポンスタイムアウトは最適化されているか？

- レスポンス タイムアウトはデフォルト 5 分 (GA: 最大 15 分 、Beta: 最大 60 分)
- レスポンスタイム内にレスポンスを返さなかった場合は、504エラーを返す

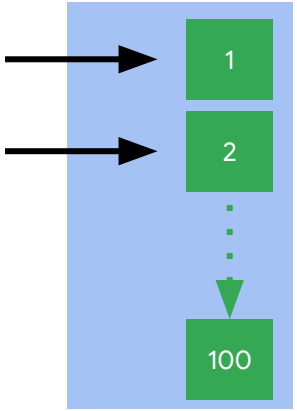


最大コンテナ数は最適化されているか？

Max instances を指定することで課金の最適化と DB のコネクション枯渇などを防ぐ

```
$ gcloud run deploy servicea \
--image gcr.io/cloudrun/hello \
--concurrency 20 \
--max-instances 100
```

リクエスト数増加に伴い
最大 100 Container Instance まで
スケールアウト

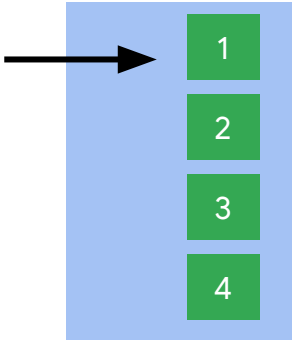


Beta

Cold Start を減らしてみる

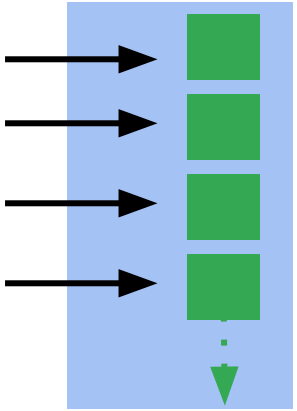
Minimum instances を指定することで Cold Start の影響を小さくすることができる

常時起動する Container Instance 数を
指定しておく




```
$ gcloud alpha run deploy servicea \
--image gcr.io/cloudrun/hello
--min-instances=4
```

リクエストがスパイクした時に
Cold Start の影響を小さくできる



2. アプリケーションの作りを見直していく

DB の Connection pool をグローバルに持つ

Connection pool はグローバルに作成し、各リクエストがDBへのコネクションを使い回せるようにする。

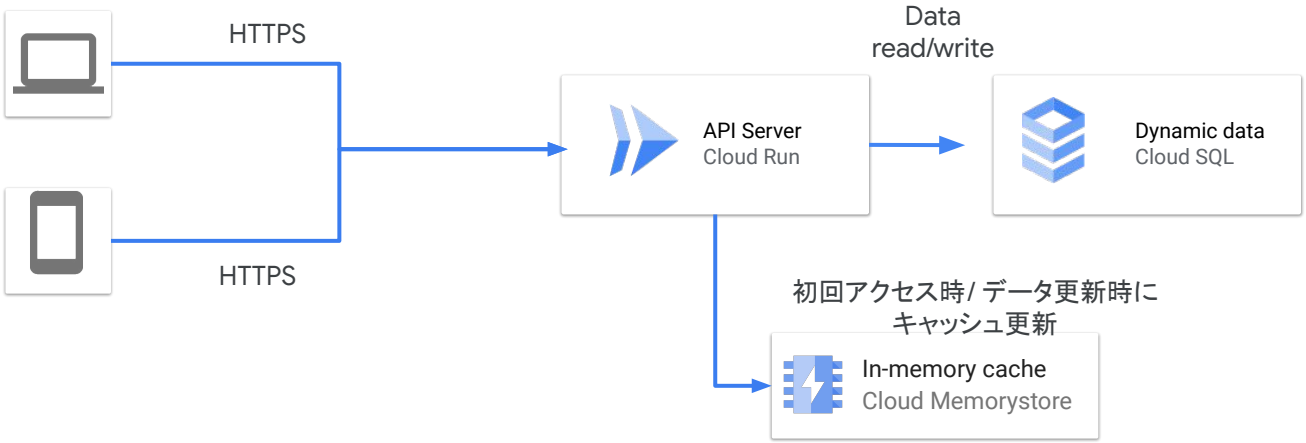
```
db = None


@app.before_first_request
def preparation():
    global db
    db = db or init_db_connection()

@app.route("/", methods=["GET"])
def index():
    with db.connect() as conn:
        results = conn.execute(
            "SELECT *************"
        ).fetchall()
    ....
    ....
    return render_template('index.html', **context)
```

インメモリ キャッシュの利用

読み込み頻度が高く、書き込み頻度が低いデータ(例えばマスターデータ等) は
アプリ本体で極力ロードせず、インメモリキャッシュ等を利用する。



コンテナ イメージの最適化

軽量なベースイメージを使うと良いのか

- Cloud Run では、コンテナ イメージのサイズは、コールドスタートやリクエストの**処理時間に影響することはない**(脆弱性の懸念は残る)
- 軽量イメージを作成することで、汎用性やデプロイ時間を短縮
  distroless、**alpine**、**scratch** といった軽量のベースイメージを利用

余計なファイルをイメージに含めない

- 不要な静的ファイルや、テストディレクトリが含まれていないか確認
- ***.dockerignore*** を作成し、不要なファイルをコンテナに含めない

まとめ

03

Cloud Run のレスポンスタイムの改善には以下の観点を最適化していくことが重要

Cloud Run の設定

- CPU / メモリ
- Concurrency（同時実行数）
- タイムアウト
- 最大、最小インスタンス数

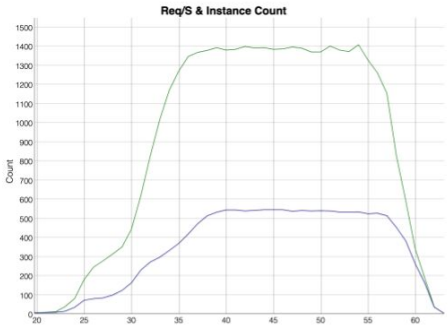
アプリケーションの作り

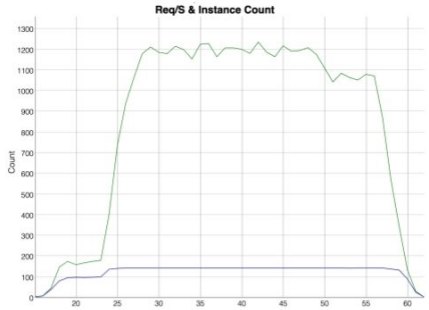
- DB connection pool を関数単位で生成しない
- マスタデータは、Redis などの外部メモリに保持して読み込む（最初は DB からロード）
- コンテナ イメージの最適化（軽量化）

同時実行数、CPU、メモリをどうやって最適化していくのか

1. 負荷テストツールを使用する。予想される負荷と同時実行の設定で、サービスの動作が安定しているかを確認
2. パフォーマンスが低下している場合は、サービスを改善し、同時実行数を減らす。CPU やメモリも並行して調整

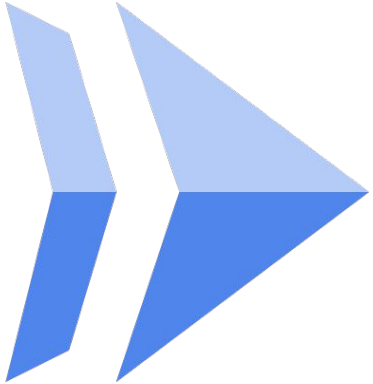
最適な設定値は、アプリの特性や言語特性にも依存する。地道な調整が大事





同時実行数の調整による
コンテナ インスタンス数の
最適化

緑線：リクエスト数
青線：インスタンス数



Happy ハイパフォーマンス Cloud Run

Thank you