

Cloud Run VS App Engine

喚醒時間簡報

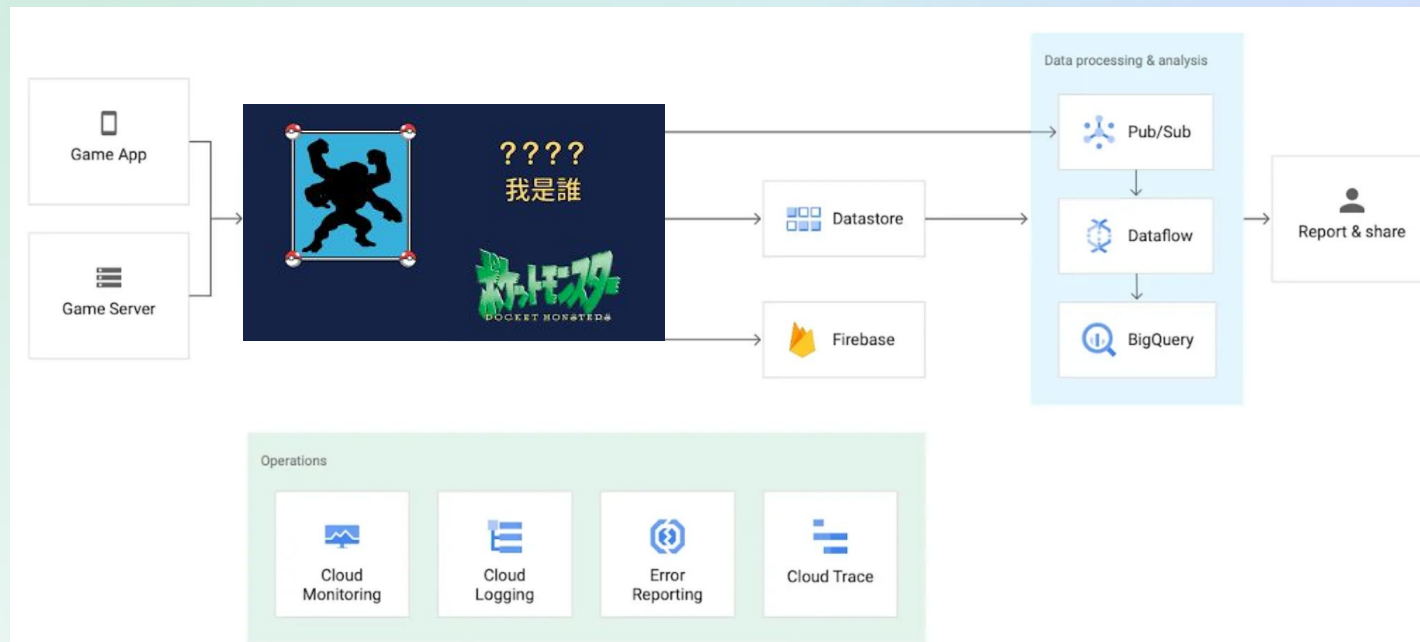
Charlie

目錄

1. 實驗目的
2. 關於喚醒時間
3. Cloud Run vs App Engine
4. 實驗設置
 1. 部署服務
 2. Dockerfile
 3. image差異
 4. 測試方法
5. 結果數據呈現
6. Summary



- 情景：我要開發一個Web應用，App Engine跟Cloud Run要怎麼選擇？兩者在喚醒時間上有什麼差異！？

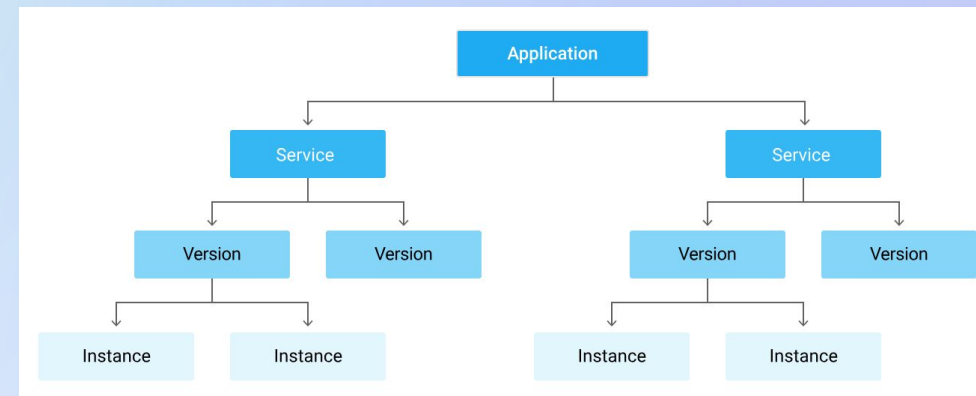


什麼是喚醒時間？

- 打GCP Cloud Run/App Engine API回應時間
=網路處理(DNS, TLS handshake...)+**冷啟動時間**+伺服器處理時間
- 冷啟動=新的容器實例被創建並首次處理請求所需的時間
- 評估重點:效能瓶頸 & 時間規模Scale

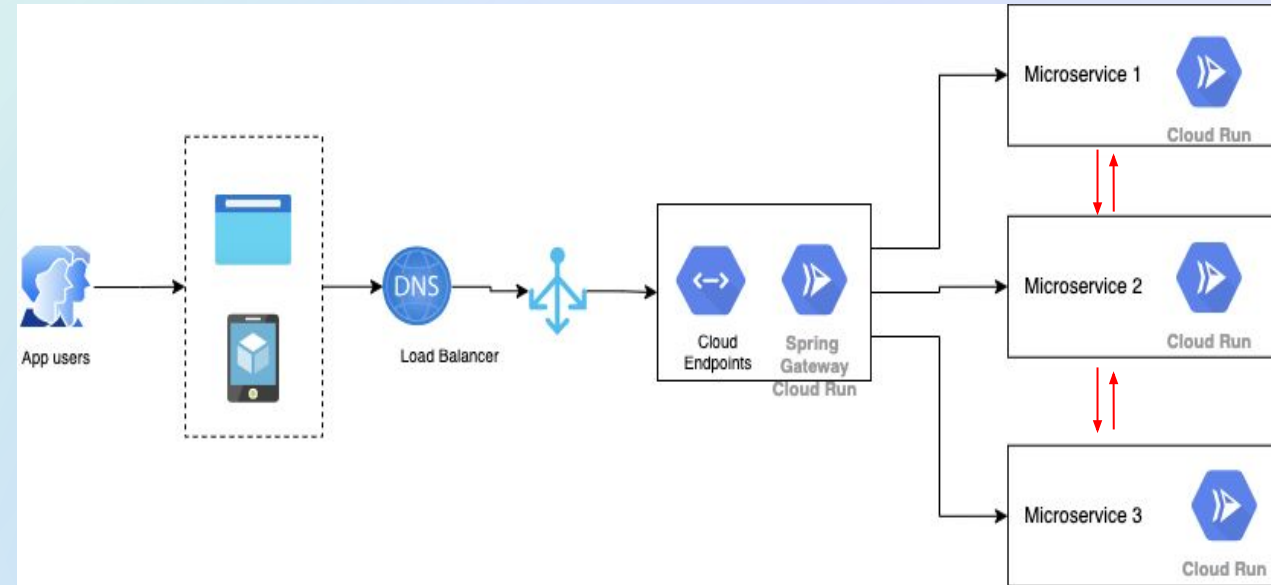


- 全託管Serverless的PaaS環境：無需管理底層
- 2008年發布
- 選擇語言開發框架Web應用
- 根據需求自動伸縮實例(auto-scaling)
- 分配流量到不同的版本(金絲雀部署)
- 標準模式VS彈性模式



服務介紹: Cloud Run

- 2019年發布: **Bringing Serverless to Containers !**
- 支援任何語言、任何標準的 Container images
- 自動伸縮 & 分流流量
- 適用更多場景:
 - 容器化工作流程
 - 靈活性高
- 以request計價, 費用低

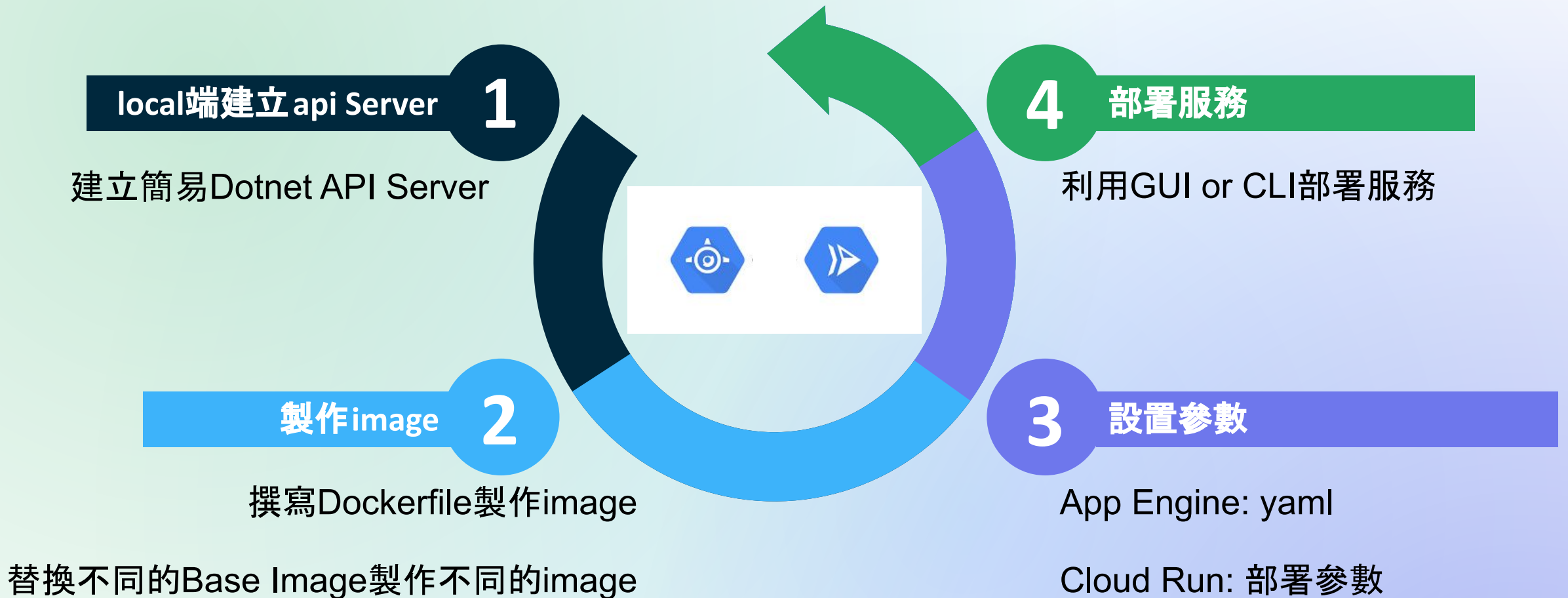


實驗環境與機制 說明

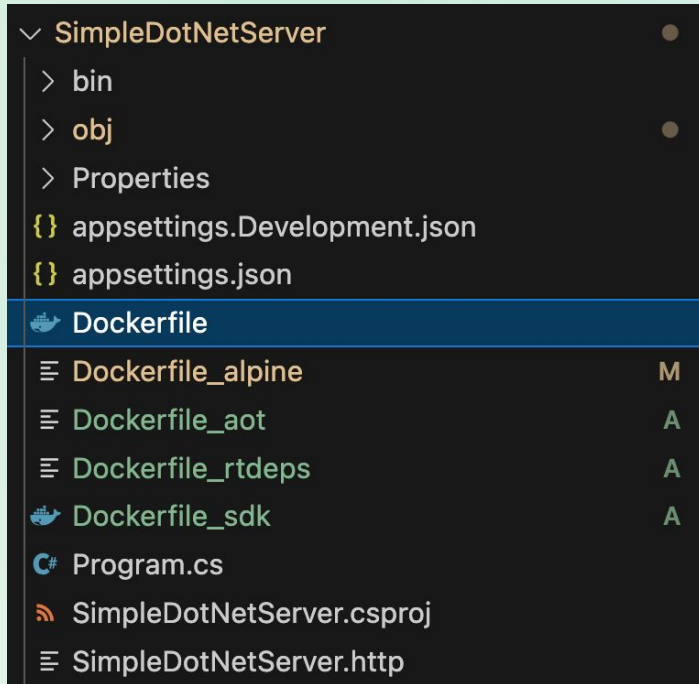
| | App Engine彈性模式 | Cloud Run | App Engine標準模式 |
|------|---------------------------|--------------------|--------------------------------|
| 運行地點 | Google Compute Engine 的VM | Cloud Run | Google 的受控沙盒環境 |
| 運行內容 | 支援原始碼部署以及自定義image部署 | | 只支援特定程式語言原始碼 不支援自定義image(註) |
| 閒置狀態 | 不會進入閒置狀態(scale to 0) | 預設15分鐘, 設定CPU來調整行為 | 預設15分鐘 |
| 回應時間 | 服務處理時間 | 冷啟動時間+服務處理時間 | |
| 費用 | 按照實例規格收費 | 依照cpu使用量收費 | 按照實例規格收費 |
| 實例擴展 | 最少要有一個運行實例 | 可以擴展到0實例 | |
| 跨區 | 無法跨區 | 可以跨區 | 無法跨區 |

註: 支援的語言: Python、Java、Node.js、Go、Ruby、PHP

實驗設置：如何設置 GCP Cloud Run/App Engine服務



實驗設置: 設置服務 - 利用 Dockerfile 建立 image



```
1 # 基礎運行環境
2 FROM mcr.microsoft.com/dotnet/aspnet:8.0 AS base
3 WORKDIR /app
4 EXPOSE 80
5
6 # 建置環境
7 FROM mcr.microsoft.com/dotnet/sdk:8.0 AS build
8 WORKDIR /src
9 | You, 1 秒前 • Uncommitted changes
10 COPY *.csproj ./
11 RUN dotnet restore
12 COPY . .
13 RUN dotnet publish -c Release -o /app/publish
14
15 # 最終階段，將應用程式部署到運行環境
16 FROM base AS final
17 WORKDIR /app
18 COPY --from=build /app/publish .
19 ENTRYPOINT ["dotnet", "SimpleDotNetServer.dll"]
```

- Muti-stage 構建方法：
- 建構階段：此階段是對原始碼進行檢查及編譯，建立執行檔。
- 運行階段：只包含運行所需的最小基礎映像，從而優化應用程序的體積和效能。
- 本次實驗利用不同的編譯方法以及運行映像基底作為操縱變因，測試喚醒時間的影響

在程式目錄內建立 Dockerfile

定義容器要如何建立及運行

實驗設置: Image

- 本次實驗選用image種類
 - SDK
 - Runtime
 - Runtime-dependency
 - Alpine
 - Alpine-AOT
- AOT vs JIT 編成機器碼的時機
 - Ahead-of-Time
 - Just-in-Time

SDK=Runtime+開發工具庫

Runtime=Runtime-deps+執行
.NET 應用程序所需的環境

Runtime-deps=運行所
需的最基本系統依賴

只能運行self-contained程式

AOT(Ahead-of-Time) vs JIT(Just-in-Time)

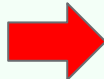
| | AOT | JIT |
|------|-----------------------|-----------------------------|
| 編譯時機 | 程式構建編譯 | 一邊執行一邊編譯 |
| 運行模式 | 編譯的Binary Code可以直接執行 | 編譯的機器碼在執行時才會編譯成Binary Code |
| 表現 | 更快的啟動時間 | 更靈活 |
| 平台相容 | 不同平台需要不同編譯方式 | Runtime時編譯出符合機器的Binary Code |
| 程式大小 | 大 | 小 |
| 記憶體量 | 大 | 小 |
| 語言 | Java, Python | C/C++ |
| 其他特點 | 新技術, 但是在.NET有許多不支援的功能 | 成熟技術, 相容性高 |

實驗設置: 不同 image 的差異

| 特性 | .NET Runtime | .NET Runtime Dependencies | .NET SDK | Alpine | Alpine AOT |
|--------|---------------|---------------------------|-----------------------|-------------------------|------------------------|
| 內容 | 完整的 .NET 運行時庫 | 僅運行 .NET 應用所需的系統庫 | 完整的 SDK 工具集, 包括編譯器和工具 | 基於 Alpine 的輕量級 .NET 運行時 | 針對 AOT 編譯優化的 Alpine 映像 |
| 作業系統大小 | 中等 | 最小 | 最大 | 小 | 小 |
| 用途 | 運行框架依賴的應用程式 | self-contained 的應用程式 | 構建和編譯應用程式 | 運行對大小敏感的應用 | 運行 AOT 編譯的應用 |
| 作業系統 | Debian | Debian | Debian | Alpine | Alpine |
| 啟動時間 | 較快 | 快 | 慢 | 較快 | 最快 |
| 適用場景 | 一般生產環境 | 資源受限環境 | 開發和構建環境 | 容器化微服務 | 需要快速啟動的服務 |
| 全球化支持 | 完整 | 有限 | 完整 | 有限 | 有限 |
| JIT 編譯 | 是 | 是 | 是 | 是 | 否 |
| AOT 編譯 | 否 | 否 | 支持編譯 | 否 | 是 |



實驗設置:規格與費用模型

| | App Engine standard environment | App Engine flexible environment | Cloud Run | | | | | | | | | | | | | | | | | | |
|------------------------|---|--|--|-------|--------|------|-----|-------|------|----|-------|------|---|-------|---------|---|-----|----|---|-----|--------------|
| Compute resources | | | | | | | | | | | | | | | | | | | | | |
| vCPU |  | <table><thead><tr><th>Instance class</th><th>vCPU*</th><th>Memory</th></tr></thead><tbody><tr><td>F/B1</td><td>.25</td><td>384MB</td></tr><tr><td>F/B2</td><td>.5</td><td>768MB</td></tr><tr><td>F/B4</td><td>1</td><td>1.5GB</td></tr><tr><td>F/B4_1G</td><td>1</td><td>3GB</td></tr><tr><td>B8</td><td>2</td><td>3GB</td></tr></tbody></table> | Instance class | vCPU* | Memory | F/B1 | .25 | 384MB | F/B2 | .5 | 768MB | F/B4 | 1 | 1.5GB | F/B4_1G | 1 | 3GB | B8 | 2 | 3GB | Up to 8 vCPU |
| Instance class | | vCPU* | Memory | | | | | | | | | | | | | | | | | | |
| F/B1 | | .25 | 384MB | | | | | | | | | | | | | | | | | | |
| F/B2 | | .5 | 768MB | | | | | | | | | | | | | | | | | | |
| F/B4 | | 1 | 1.5GB | | | | | | | | | | | | | | | | | | |
| F/B4_1G | | 1 | 3GB | | | | | | | | | | | | | | | | | | |
| B8 | 2 | 3GB | | | | | | | | | | | | | | | | | | | |
| Memory | | | Up to 32GB | | | | | | | | | | | | | | | | | | |
| Pricing model | | | | | | | | | | | | | | | | | | | | | |
| Per-request fee | No | | No, when <u>CPU is always allocated</u> . Yes, when <u>CPU is only allocated</u> during request processing. | | | | | | | | | | | | | | | | | | |
| Idle minimum instances | Same cost as active instances | | Lower cost for idle minimum instances | | | | | | | | | | | | | | | | | | |

說明: Cloud run的CPU Always-on選項達到App Engine的不閒置相同行為

說明: Cloud run的CPU Always-on選項可以達到App Engine的不閒置相同行為



實驗設置: 設置服務

Cloud Run(CLI參數控制)

部署指令範例

```
gcloud run deploy simple-dotnet-server \
  --image gcr.io/[YOUR_PROJECT_ID]/simple-dotnet-server
  --platform managed \
  --region [YOUR_REGION] \
  --allow-unauthenticated
```

VS

App Engine(yaml檔控制+CLI)

yaml檔案範例

```
1 runtime: custom
2 env: flex
3
4 # service: runtime
5
6 instance_class: F1 # 使用 F1 實例類型
7
8 automatic_scaling:
9   max_num_instances: 1
10 network:
11   name: cdckh-poc-network
```

部署指令範例

```
gcloud app deploy ${PATH_TO_YAML} --image-url=${IMAGE_URL}
```

更多部署細節參考

TFS WIKI: Training -> GCP

實驗設置:測試方法

1. 使用curl獲得回應時間
2. 撰寫bash腳本蒐集輸出
3. 使用crontab排程定時測試
4. 使用python產出結果分析

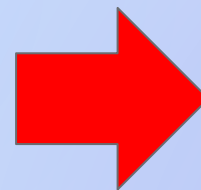
更多測試細節參考

TFS WIKI: Training -> GCP

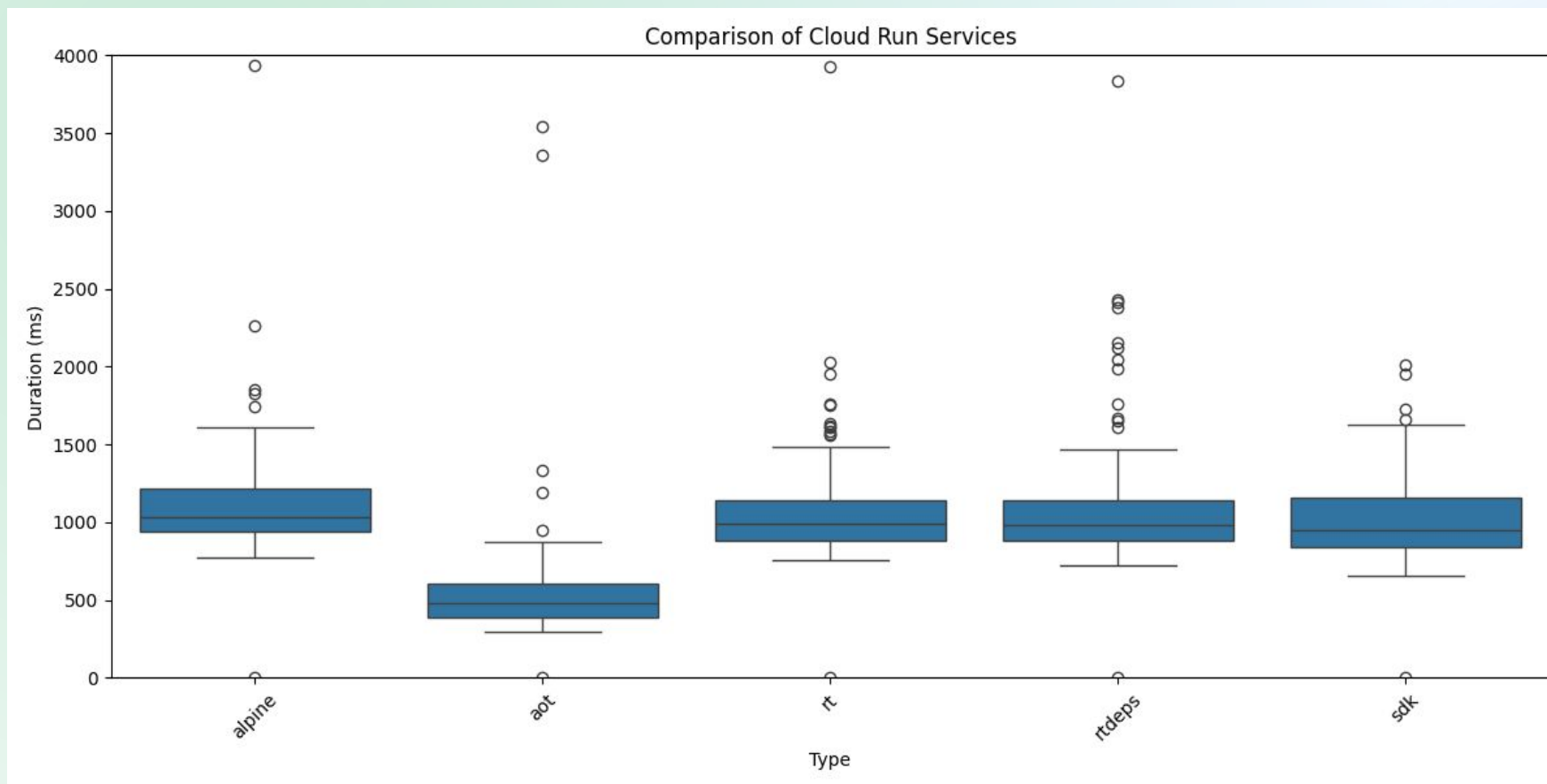
```
64 # 第一次請求
65 REQUEST_TIMESTAMP=$(date +"%Y-%m-%d %H:%M:%S")
66 FIRST_TIME=$(curl -w "%{time_total}\n" -o /dev/null -s $URL)
67 echo "[$REQUEST_TIMESTAMP] First request: $FIRST_TIME s" >> $OUTPUT_FILE
68
69 echo "多次測試結果已記錄到 $OUTPUT_FILE"
```



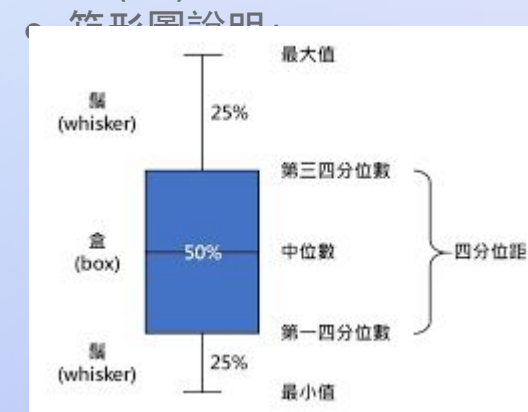
```
~ crontab -l
*/30 9-17 * * 1-5 /Users/charliecheng/cathay/ct_wakeup/multi_test.sh
```



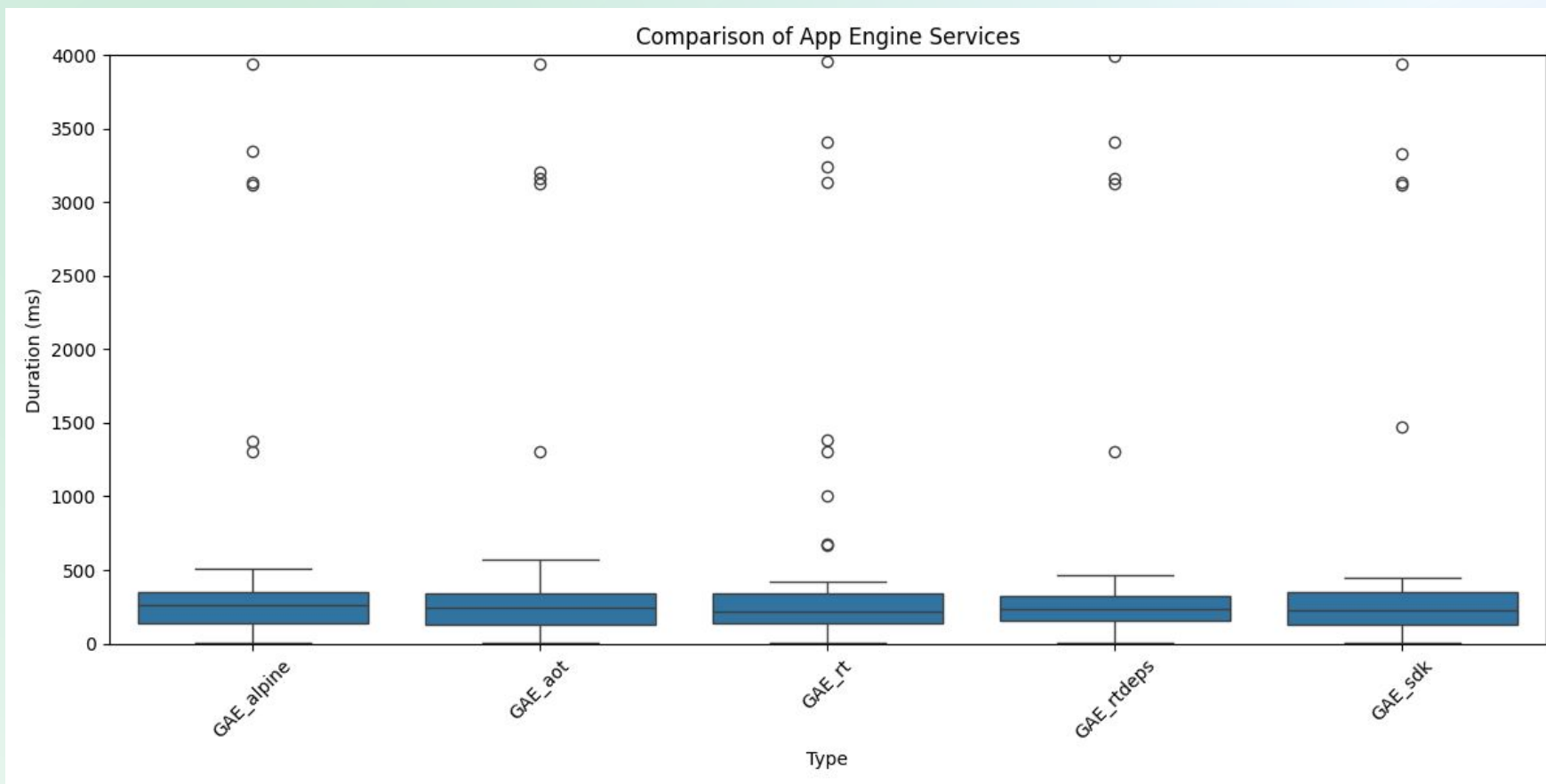
實驗結果：不同 image 結果數據統計圖 -Cloud Run



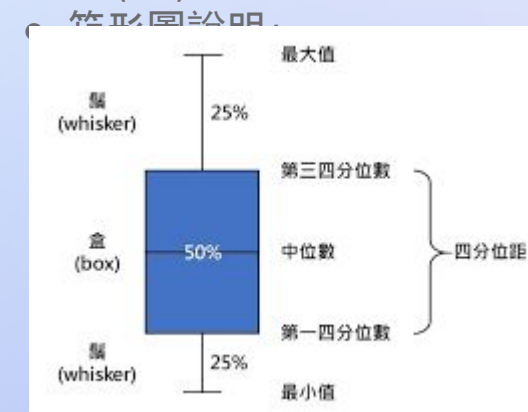
- 實驗樣本數：200次
- 回應時間：1秒
- X軸 (Type): 不同image的種類
- Y軸 (Duration, ms): 代表回應喚醒時間，單位是毫秒 (ms)。



實驗結果：不同 image 結果數據統計圖 -App Engine彈性模式



- 實驗樣本數：200次
- 回應時間：0.2秒
- X軸 (Type): 不同image的種類
- Y軸 (Duration, ms): 代表回應喚醒時間，單位是毫秒 (ms)。



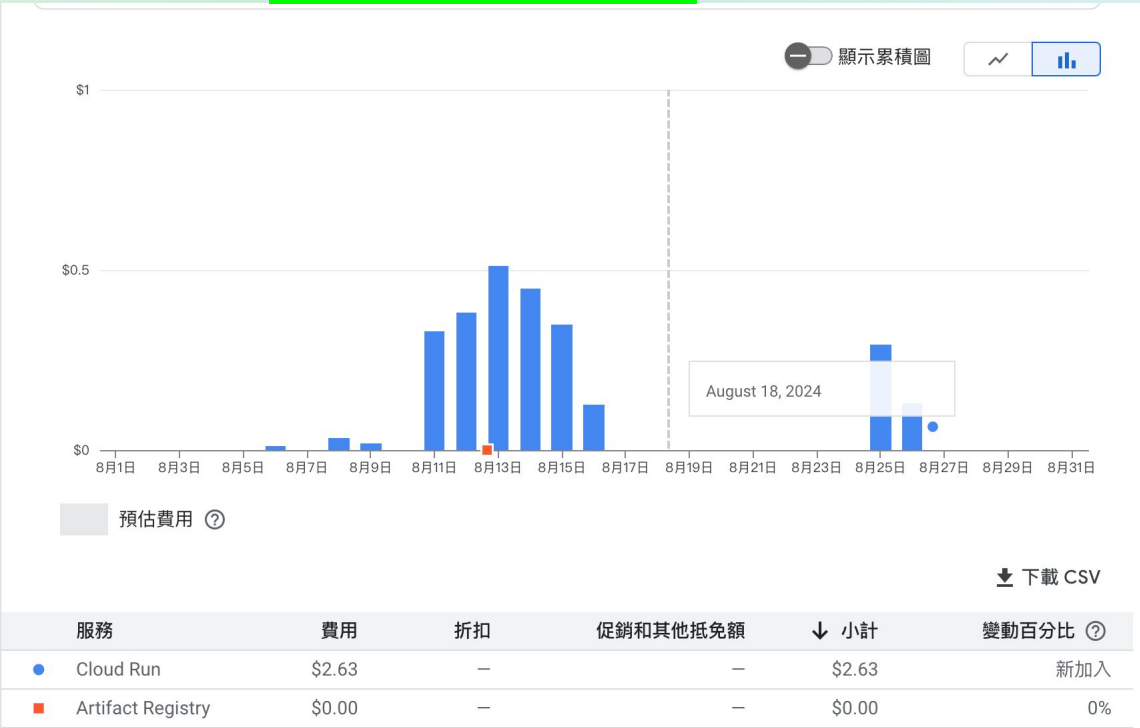
實驗結果：費用

Cloud Run

VS

App Engine(彈性模式)

\$0.3 USD/Day



\$7.8 USD/Day



對於喚醒時間，可以做得更好嗎？



喚醒時間優化方法: (App Engine)

- warmup request(App Engine):
 - 設置方法:
 - 配置yaml檔案的warmup參數
 - 在系統controller中配置
- 在yaml檔中配置參數
 - min_instance: 最小啟動實例
 - max_concurrent_requests: 最大同時流量
 - 註: Cloud Run使用參數設置

```
1 runtime: custom
2 env: flex
3
4 # service: runtime
5
6 instance_class: F1 # 使用 F1 實例類型
7
8 automatic_scaling:
9   max_num_instances: 1
10 network:
11   name: cdckh-poc-network
12
13 # warmup request
14 inbound_services:
15 - warmup
16
17 # 自動擴展參數設置
18 automatic_scaling:
19   target_cpu_utilization: 0.65
20   min_instances: 5
21   max_instances: 100
22   min_pending_latency: 30ms
23   max_pending_latency: automatic
24   max_concurrent_requests: 50
```

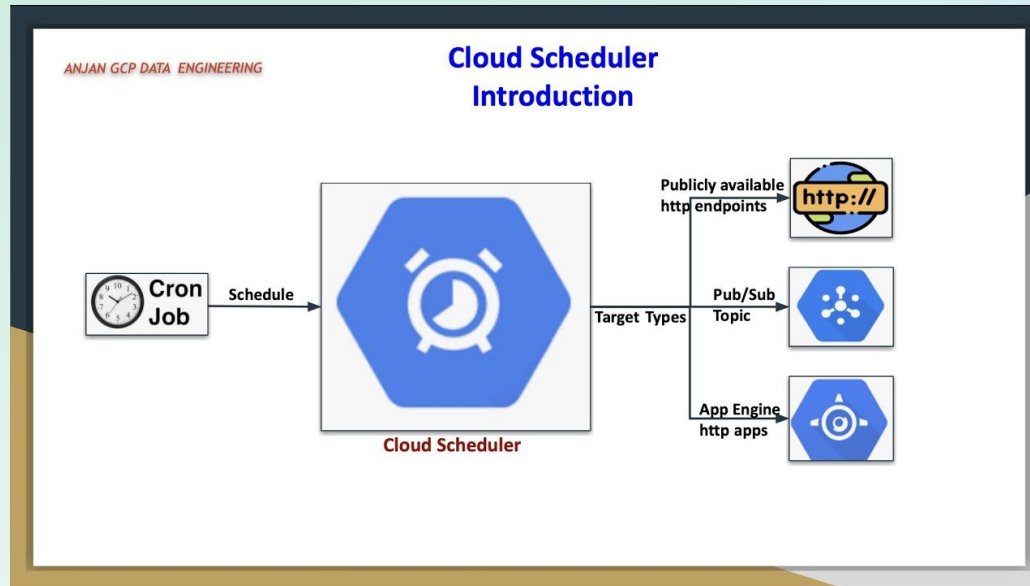
```
@app.route("/_ah/warmup")
def warmup():
    """Served stub function returning no content.

    Your warmup logic can be implemented here (e.g. set up a database connection pool)

    Returns:
        An empty string, an HTTP code 200, and an empty object.
    """
    return "", 200, {}
```

喚醒時間優化方法：排程喚醒

- 在yaml檔案中配置liveness_check
- 使用Cloud Scheduler設置API喚醒



```
liveness_check:  
  path: "/liveness_check"  
  check_interval_sec: 30  
  timeout_sec: 4  
  failure_threshold: 2  
  success_threshold: 2
```


結論



Thank you!
Q&A Time

