Documentación
Carrera de Pods
v5

Índice

Propósito general

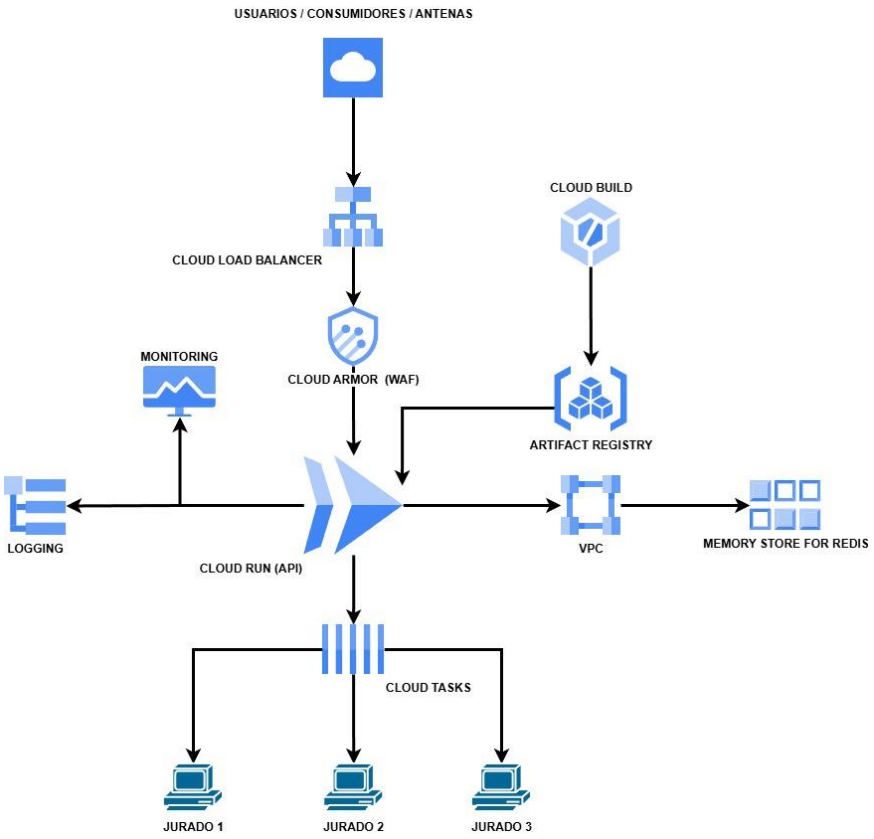
El propósito de este documento es registrar la información referida al diseño, desarrollo e implementación del proyecto “Carrera de Pods”. El proyecto alojado en la plataforma de Google Cloud Platform (GCP) satisface todos los requerimientos solicitados en Carrera de Pods v5.pdf.

El flujo general del sistema es el siguiente:

1. Las antenas envían información parcial mediante /podhealth_split/{antena}.
2. Redis almacena temporalmente los datos asociados a cada POD.
3. El cliente consulta /podhealth_split/{nombrePod}.
4. El sistema:
    - Recupera los valores en Redis
    - Triangula la posición
    - Determina las métricas
    - Devuelve resultado al cliente
    - Crea 3 tareas en Cloud Tasks para notificar a los jurados
5. Cloud Tasks reintenta hasta que cada jurado confirma recepción.

Arquitectura

Para el diseño de la arquitectura del proyecto se utilizaron los servicios de la plataforma Google Cloud Platform (GCP). La arquitectura de los servicios utilizados para satisfacer todos los requerimientos es la siguiente



Servicios utilizados en GCP

Cloud Run: Es el componente central de la arquitectura final. Ejecuta nuestra imagen de contenedor (Dockerfile) de FastAPI. Provee la escalabilidad automática para manejar las 4000 RPM.

Memorystore for Redis: Es el almacén de estado externo. Guarda los fragmentos de información de los pods recibidos en las antenas (POST) hasta que el GET los consulta. También se guarda el registro de antenas con su nombre y ubicación, permitiendo hacer un ABM de antenas.

Google Container Registry (GCR) / Artifact Registry: Repositorio donde se almacena la imagen de Docker (gcr.io/carrera-pods-gcossa/carrera-pods) después de construirla. Cloud Run toma la imagen de aquí para crear las instancias.

Cloud Load Balancing: Provee la IP pública estática y distribuye el tráfico (4000 RPM) a las instancias de Cloud Run.

Virtual Private Cloud (VPC): Este servicio crea un "puente" (nombrado carrera-pods-conector) que permite a Cloud Run comunicarse con la red VPC (donde está Redis).

Cloud Tasks: El servicio de cola que garantiza la entrega a los 3 jurados. El GET publica 3 tareas en la cola (juror-queue) y Cloud Tasks se encarga de reintentar las llamadas al worker (/tasks/notify_juror) si los servicios externos fallan.

Cloud Build: El servicio que "construye" la aplicación. Utilizado en la llamada con el comando *gcloud builds submit* para construir nuestro Dockerfile.

Cloud Logging: Herramienta de depuración principal.

Cloud Monitoring: El servicio donde monitorearíamos los KPIs (Tasa de Error, Latencia de API, Latencia de Redis, Profundidad de Cola de Cloud Tasks).

Cloud Armor: El Firewall de Aplicaciones Web (WAF) conectado al Balanceador de Carga para proteger la API contra ataques comunes (como Inyección SQL).

Diseño de código

El proyecto está desarrollado en Python haciendo uso del framework FastAPI, debido a su simplicidad y gran escalabilidad. La API tiene dos funciones principales con la lógica necesaria para resolver las demandas de los usuarios / sistemas y endpoints desde los cuales se acceden a la misma

Funciones

```
async defObtenerPosicionPod(distancias, antenasNombres)
```

Objetivo

Obtener la posición de un POD triangulando las distancias que hay entre el POD y cada una de las antenas.

Entrada

- **Distancias:** Lista de distancias entre las antenas y el POD respectivamente. Ej.: [250, 435,213]
- **AntenasNombres:** Lista de nombre de antenas, utilizado para buscar la posición de cada una de las antenas en redis.

Salida

- Tupla (X, Y) de un POD Se deben utilizar 3 antenas y las mimas deben contener sus posiciones correspondientes.

Restricciones

- Si una antena es eliminada, la información que tenía sobre un POD es invalida (a pesar de la información se haya recopilado antes de borrar la misma), ya que no se puede determinar la posición de la antena.
- Si se pasan como dato de entrada menos de 3 antenas, se lanza un error 500, ya que se requiere la distancia de 3 antenas para la triangulación.
- Si se pasan más de 3 antenas, solo se consideran las 3 primeras.

```
 def ObtenerMetricasPod(mensajes)
```

Objetivo

Devolver las métricas (temperatura de motores anti gravitación, nivel de energía, temperatura de baterías, porcentaje de averías) de un POD. Para ello se analizan los mensajes recibidos en 3 antenas. Considerando que los mensajes pueden estar **desfasados** y que las métricas temperatura de motores y temperatura de baterías **tienen la misma unidad**, se determinó que una métrica es válida solo si de los 3 mensajes 2 de ellos tienen el mismo valor **y unidad** de métrica. Por ejemplo:

"antena0" → ["590C","","","60%"]
"antena1" →["","1MWh","","60%"]
"antena2" → ["590C","","110C, ","",""]

Si las antenas reciben esos mensajes cada una respectivamente, el valor de la temperatura de motores de puede determinar (porque 2 mensajes coinciden con el valor y unidad 590C), lo mismo que el porcentaje de baterías (60%). Sin embargo, la métrica de temperatura de baterías no puede determinarse, ya que de los 3 mensajes solo uno reporto un valor, pero podría ser un desfasaje, ya que el resto de los mensajes no recibió información.

Entrada

Recibe una lista de mensajes → [["120C","","","60%"], ["","1MWh","","60%"], ["90C","","110C, ","",""]]

Salida

Devuelve una lista con las métricas obtenidas. En caso de no poder haber determinado alguna, se considera "". → ["590C","1MWh","","60%"]

Validación de datos de entrada

Para la validación de datos se utilizó Pydantic, y se definió las clases que representan la estructura de los payloads de entrada.

```python
class DatosAntena(BaseModel):
    name: str
    pod: str
    distance: float
    metrics: list[str]

class InfoAntenas(BaseModel):
    antenas: list[DatosAntena]

class DataPod(BaseModel):
    pod: str
    distance: float
    message: list[str]

class Jurado(BaseModel):
    urlJurado: str # La URL del jurado externo al que llamar
    payload: dict  # El resultado que se le envia

class Antena(BaseModel):
    name: str
    position: list[int, int]
```

Endpoints

`@app.post("/podhealth/")` →Recibe un payload con la información varias antenas juntas
`@app.post("/podhealth_split/{antena_name}")` → Recibe la información recopilada por una antena
`@app.post("/tasks/notificarJurado")` → utilizado internamente por GCP para informar a los jurados
`@app.get("/podhealth_split/{nombrePod}")` → Devuelve la posición y métricas de un POD determinado
`@app.get("/datosCapturadosPod/{nombrePod` → Devuelve la información cruda recopilada por diversas antenas para un POD determinado
`@app.post("/registrarAntena/")`→ Permite agregar una antena nueva
`@app.delete("/eliminarAntena/{nombreAntena}")` → Elimina una antena
`@app.delete("/podhealth_split/{nombrePod}")`→Permite eliminar toda la información recopilada por las diversas antenas para un POD determinado

Monitoreo

Con el fin de garantizar una alta disponibilidad de los servicios, se deben armar dashboards de ciertos parámetros sensible como:

- Picos de consultas
- Cantidad de fallos en el envío de tareas a jurados
- Consumo de CPU para el procesamiento de información

Configuración del entorno

Ambiente local

Para poder deployar el ambiente en una PC local a modo de testing o desarrollo, se deben seguir los siguientes pasos:

1) **Instalar librerías necesarias:** pip install –m fastapi, uvicorn[standard], numpy, Pydantyic, typing, json
2) **Levantar redis en un contenedor de docker:** docker run --name redis-local -p 6379:6379 -d redis
3) **Ejecutar API:** fastapi dev main.py

Google Cloud Platform

Como se mencionó previamente, el proyecto se encuentra alojado en GCP, más precisamente en la IP http://136.110.196.219 (agregar /docs al final de la ruta para entrar al Swagger y probar los endpoints). Para poder configurar dicho entorno se puede hacer a través de comandos CMD en el siguiente orden:

1) **Construir imagen:** gcloud builds submit --tag gcr.io/carrera-pods-gcossa/carrera-pods

2) **Crear VPC:** gcloud compute networks vpc-access connectors create carrera-pods-conector --region us-central1 --network default --range 10.8.0.0/28

3) **Levantar el servicio de Redis:** gcloud run deploy carrera-pods-api --image gcr.io/carrera-pods-gcossa/carrera-pods --platform managed --region us-central1 --allow-unauthenticated --vpc-connector carrera-pods-conector --set-env-vars REDIS_HOST=10.2.0.3

4) **Unificar todos los servicios (api, redis, vpc, task):** gcloud run deploy carrera-pods-api --image gcr.io/carrera-pods-gcossa/carrera-pods --platform managed --region us-central1 --allow-unauthenticated --vpc-connector carrera-pods-conector --set-env-vars REDIS_HOST=10.2.0.3,PROJECT_ID=carrera-pods-gcossa,QUEUE_ID=juror-queue,LOCATION_ID=us-central1,SERVICE_URL=https://carrera-pods-api-79403090209.us-central1.run.app

NOTA: estas configuraciones son válidas en el proyecto *carrera-pods-gcossa* creado previamente en GCP y configurado con las integraciones de los servicios pertinentes. Para un proyecto nuevo en GPC los comandos son los mismos, pero variaran los nombres y variables de entorno (como REDIS_HOST) del proyecto en cuestión.

Anexo

Para la realización de este proyecto se hizo uso de la IA Gemini, con el modelo 2.5Pro, utilizando la herramienta de “Aprendizaje guiado” con el objetivo de obtener una guía para resolver los problemas y no una respuesta concreta sin contexto. Se utilizó principalmente para resolver el posicionamiento de los PODS haciendo uso de trilateración y los lineamientos de los requerimientos no funcionales, ya que al no estar familiarizado con la plataforma Google Cloud Platform, se recurrió al uso de la IA mencionada. Algunos de los prompts utilizados fueron:

- Si fueras un desarrollador backend y tu API necesita resolver 4000 RPM, que servicios de GCP utilizarías para satisfacer este requerimiento?
- Dame un ejemplo de como configurarías Google Tasks, de modo que pueda enviar respuestas a varios servicios y reintentar su envió de forma recurrente.
- Si tuvieras que determinar la posición de un objeto y tuvieras la posición de tres antenas en un plano de ejes cartesianos, que lógica utilizarías?