Airflow

Cloud Computing: Servicios y Aplicaciones

José Antonio Córdoba Gómez - 77201588H

Granada, España - 8 de mayo de 2021
Máster en Ingeniería Informática

Índice general

Capítulo 1

Introducción

1.1 Entorno y despligue Apache Airflow

Para realizar la práctica he usado una instancia *CX21* de **Hertzner** situada en Frankfurt, Alemania. Al igual que las instancias de Digital Ocean, no traen unidad de swap, por lo que hemos tenido que crearle una unidad de swap para no ver nuestros contenedores muertos o el entrenamiento de nuestros modelos con airma detenidos repentinamente (recibir la señal **SIGTERM**).

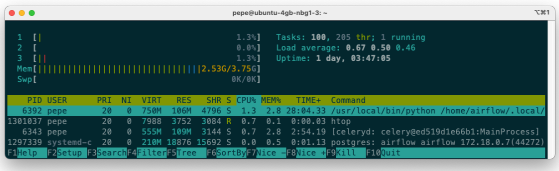

Figura 1.1: Memoria Virtual inexistente

Para ello hemos realizado este proceso:

```
sudo fallocate -l 3.5G /swapfile
sudo chmod 600 /swapfile # Lo hace w/r al propietario (Root)
sudo mkswap /swapfile # Crea el sistema de archivos
sudo swapon /swapfile
 # Lo montamos siempre
```

```
echo '/swapfile none swap sw 0 0' | sudo tee -a /etc/fstab
# Configura la prioridad de hacer swap
sudo systemctl sysctl vm.swappiness=15
```

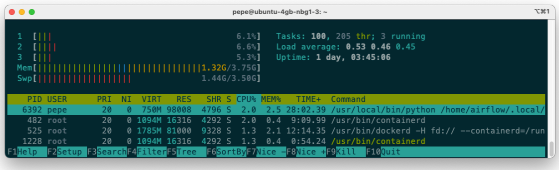

Figura 1.2: Uso de memoria virtual

El despliegue del servicio de **Apache Airflow** lo he realizado usando el fichero de composición de servicios que proporciona la propia fundación Apache en su página web [1].

Para ello hemos creado las tablas de las bases de datos y lanzado la configuración inicial con:

```
docker-compose up airflow-init
```

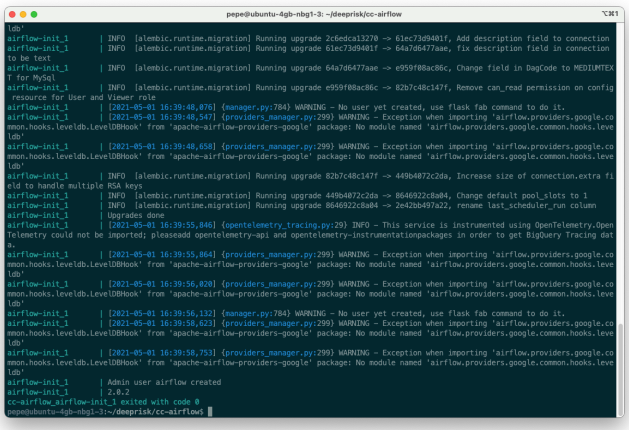

Figura 1.3: Inicialización de la configuración del servicio

Una vez terminado hemos podido lanzar el servicio de airflow como tal y lo hemos dejado como demonio:

```
docker-compose up -d && docker-compose logs -f
```

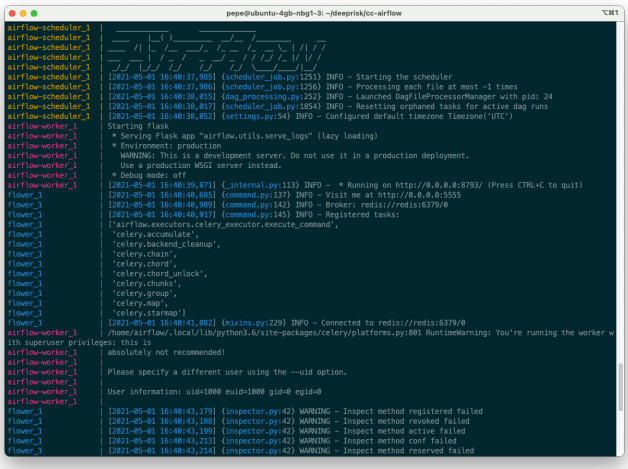

Figura 1.4: Logs de los servicios en activo

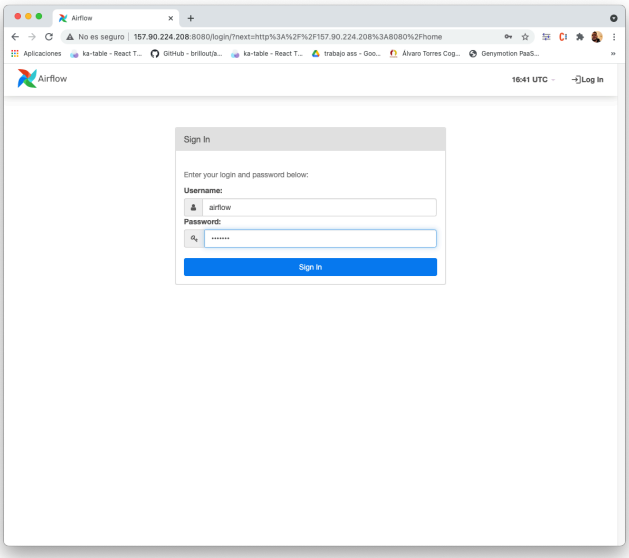

Figura 1.5: El servicio web de airflow nos responde correctamente y procedemos a ingresar con el usuario airflow

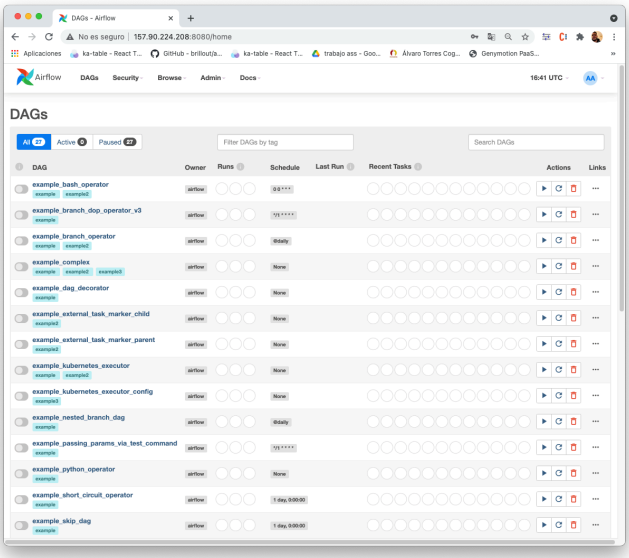

Figura 1.6: Y comprobamos que podemos interactuar con los DAGs que trae por defecto

Capítulo 2

Resolución de tareas

Para modelar un flujo de ejecución que cumpla los requisitos de la práctica se han ido realizando tareas (DAGs) minimales y unitarios con cada una de las tareas que se pueden encontrar en la entrega de esta práctica dentro de la carpeta (DAG) y que son fácilmente reconocibles por la semántica de su nombre.

Como nuestros contenedores **worker**, **flower**, y **shceduler** están basadas en una versión de debian *slim* y no contiene herramientas como **wget**, **git** o **unzip**, hemos preferido no realizar instalaciones de programas adicionales y realizar las tareas con otras herramientas como **curl** o implementando las funcionalidades en scripts de **python**.

Las únicas dependencias que tenemos que instalar serán las dependencias de nuestros scripts de python en los tres contenedores de airflow: **worker**, **flower**, y **shceduler** para que la importación, planificación y ejecución de las tareas no fallen. Para ello hemos realizado:

```
packages=$(cat requirements.txt |tr "\n" " ") && \
for i in $packages; \
do docker exec --user airflow \
cc-airflow_flower_1 pip install $i \
docker exec --user airflow  \
```

```
cc-airflow_airflow-scheduler_1  pip install $i; \
docker exec --user airflow  \
cc-airflow_airflow-worker_1  pip install $i; done
```

2.1 Obtención de datos

Primero creamos un *bash operator* que crea una carpeta temporal para que podamos guardar los datos **CreateDir**, posteriormente hemos realizado una operación *curl* para la descarga de los datos de la temperatura (**DownloadTemperatureData**) y humedad (**DownloadHumidityData**).

Hemos establecido que la obtención de los datos de humedad y temperatura tiene que ser precedidos de la tarea de creación del directorio de guardado, tal y como se puede ver en pepitoenpeligro/cc2_apache_airflow_wheater_predictor

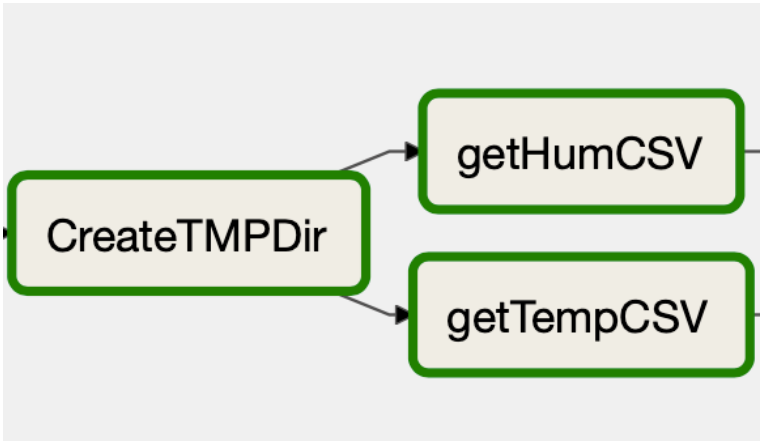

Figura 2.1: Diagrama de flujo de las tareas de obtención de datos

2.2 Preprocesado de datos

Posteriormente a lo anterior, tenemos que descomprimir los datos, ya que los descargamos como ficheros comprimidos con el algoritmo **zip**. También debemos leer los datos y generar un formato de salida del estilo **DATE;TEMP;HUM**.

Para la descompresión de los ficheros, hemos implementado la descompresión en una función de python a la que hemos llamado en la tarea **ExtractZip**

El preprocesamiento lo realizamos también a través de otro *pythonOperator* que se encarga de leer ambos *csv* con la librería *pandas*, que consiste en quedarnos con la columna de San Francisco de ambos csv, rellenar los valores perdidos con la media y exportar un *csv* con el carácter tabulador como separador, sin incluir los índices de cada fila y usando la codificación *utf-8*. Esta tarea se llama Preprocessing.

El flujo de ejecución de esta tarea compuesta la podemos encontrar de nuevo al final del DAG en pepitoenpeligro/cc2_apache_airflow_wheater_predictor

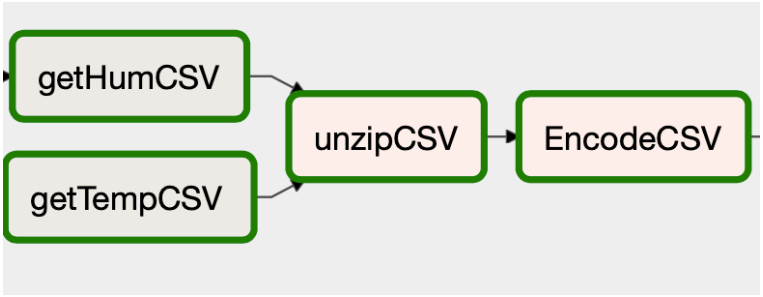

Figura 2.2: Diagrama de flujo de las tareas de preprocesamiento

2.3 Almacenamiento de datos en BD

A continuación guardamos los datos en una base de datos de Mongo, pero en nuestro caso hemos optado por usar un **DaaS** llamado **Mongo Atlas** que nos proporciona un clúster de 3 nodos réplica con 512MB de almacenamiento. Dentro hemos creado un usuario que sólo puede escribir y leer dentro de una base de datos llamada **p2Airflow** y una colección llamada **sanfrancisco**.

Esta tarea se llama SaveToMongo que incluye la llamada a una función de python que a través de la librería de *pymongo* crea una conexión a este servicio de base de datos y hace la inserción de los datos como **many**, es decir, para cada tupla de valores **fecha, temperatura y humedad** crea un documento u objeto en esa colección.

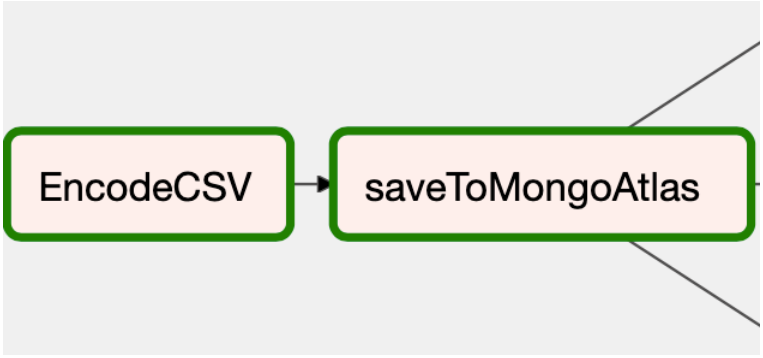

Figura 2.3: Diagrama de flujo de las tareas de almacenamiento en Mongo Atlas

2.4 Microservicios

Los modelos no los entrenamos en cada petición a los endpoints de los microservicios ya que los datos no se actualizan, pero realmente, si fuera así, tendríamos que incluir una tarea que fuera sólo de entrenamiento de los modelos. Estos modelos se entrenan una única vez y se serializan usando la librería *pickle* incluida en el **core** del lenguaje desde python3.6 y que posteriormente se comprimen como zip para no sobrepasar los 100MB máximos que puede ocupar un único fichero en GitHub.

Al principio no los comprimí y consumí la cuota gratuita de GLFS, me obligaron a realizar un pagoe de un paquete de datos de 5 para poder operar sobre esos ficheros, aunque fuera para eliminarlos, y decidí no agotar más esta vía, comprimiendo y descomprimiendo los modelos.

2.4.1 Versión 1

La primera versión del microservicio que consume el modelo de predicción de temperatura basado en el modelo ARIMA tal y como se propone en el el guión se puede encontrar en el repositorio pepitoenpeligro/cc2_weatherpredictor_v1. Dentro podemos encontrar un fichero arima_dao.py que se encarga de crear el modelo de temperatura y humedad y exportarlos en zip.

Consta de un **middleware** que se encarga de recoger la peticion que se pide para mostrarla por el log y poder diferenciarla del resto de peticiones.

A todas las respuestas que devuelva le incorporamos el **Content-Type** de **application/json**

También podemos encontrar el microservicio, implementado con el marco de desarrollo web, **Flask** server.py

Se expone al exterior por el puerto **3005**

2.4.2 Versión 2

Esta versión es muy similar a la anterior, pero en vez de usar el auto arima, usamos el modelo **SARIMAX**

Se expone al exterior por el puerto **3006**

2.4.3 Versión 3

En esta versión hacemos realizamos una petición a la API de **OpenWeather** y procesamos su respuesta para adecuarnos al formato de las espuestas de los servicios anteriores. Si miramos el controlador de la ruta 24horas podemos ver cómo en una simple función lambda se puede adecuar la respuesta de **OpenWeather** al formato anterior con poquísimo esfuerzo sintáctico.

Se expone al exterior por el puerto **3007**

2.4.4 Integración en el orquestador de tareas

Estos microservicios, una vez alojados ya en un repositorio público de GitHub, se deben de obtener (descargar), para lo cual hemos definido una tarea para cada versión del microservicio. Estas tareas se llaman DownloadService1, DownloadService2 y DownloadService3

2.5 Pruebas de unidad

Para las pruebas de unidad de los microservicios se han definido tres tareas llamadas TestServiceV1,TestServiceV2 y TestServiceV3 que consiste en lanzar el servidor y comprobar que cada *http endpoint* responde correctamente y devuelve un **status** de 200.

2.6 Flujo de tareas

El flujo de tareas en Apache Ariflow se quedaría como sigue:

```
DownloadService1  >> ExtractService1 >> TestServiceV1
DownloadService2  >> ExtractService2 >> TestServiceV2
DownloadService3  >> ExtractService3 >> TestServiceV3
```

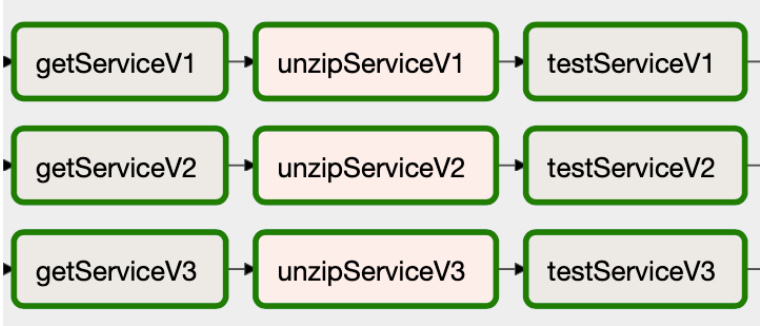

Figura 2.4: Diagrama de flujo de las tareas descarga, descompresión y testeo de los microservicios en cada versión

2.7 Despligue de los servicios

Para el despliegue de los servicios hemos construido los contenedores correspondientes de cada microservicio y los hemos puesto en ejecucicón en una máquina adicional.

Podemos encontrar las tareas como GenerarContenedorV1 y DesplegarContenedorV1, siendo extensible al resto de versiones.

Para ello, previamente hemos creado una clave *ssh* y hemos enviado la clave a la otra máquina donde desplegamos realmente los servicios. Esto lo hacemos para que la conexión se pueda hacer sin autenticación con contraseña y por tanto el acceso automático.

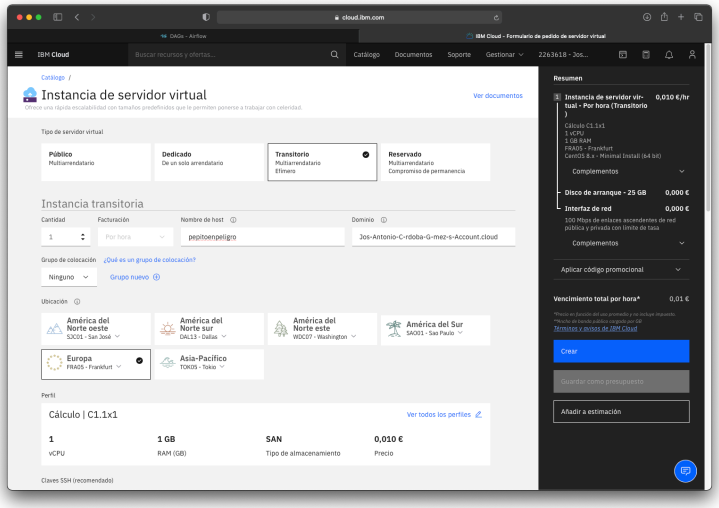

Figura 2.5: Creación de la máquina donde se va a desplegar los servicios realmente : IBMCloud

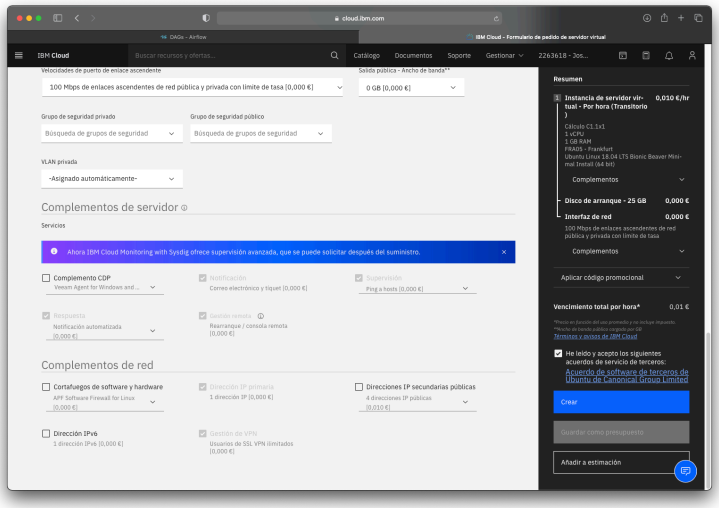

Figura 2.6: Creación de la máquina donde se va a desplegar los servicios realmente : IBMCloud

Lanzamos el flujo de tareas completamente y obtenemos el siguiente flujo:

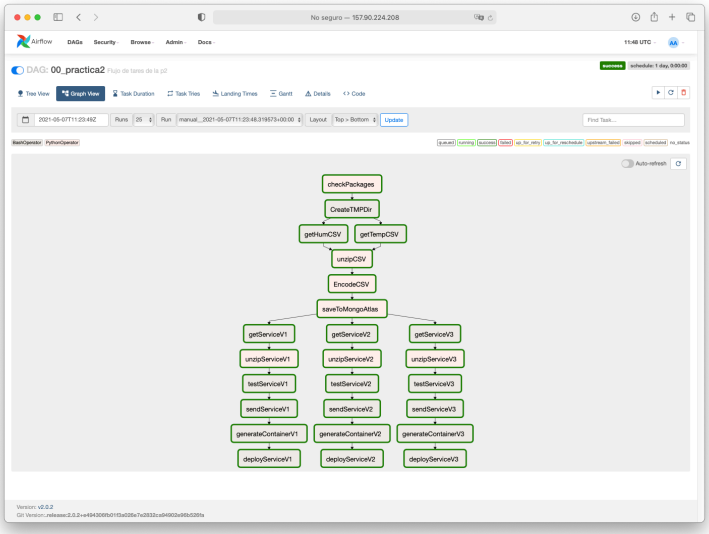

Figura 2.7: Flujo de tareas y estados

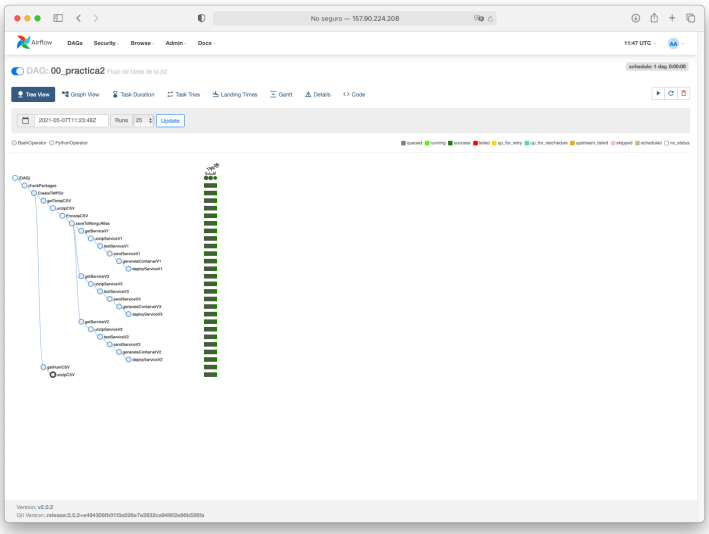

Figura 2.8: Flujo de tareas y estados

También podemos ver el diagrama de Gantt de cada tarea. Nótese que el tiempo para

desplegar los dos contenedores V2 y V3 fue excesivo porque necesitó de reintentos.

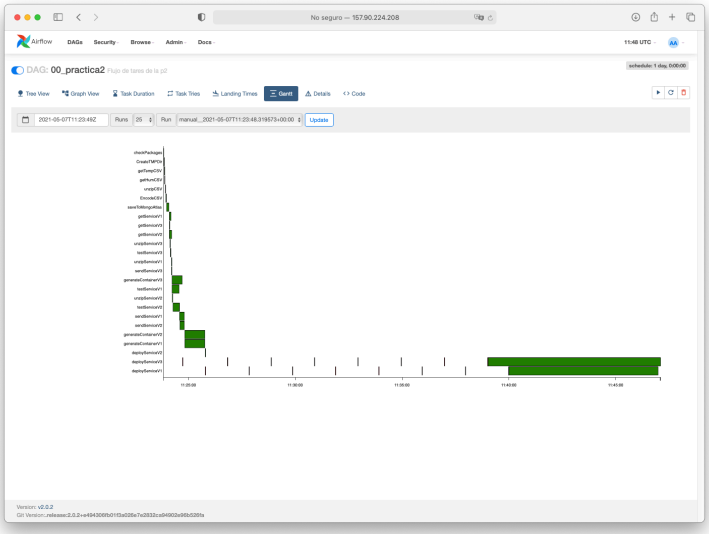

Figura 2.9: Diagrama de Gantt

Capítulo 3

Conclusiones

Un sistema de orquestación de tareas siempre ha sido una de los componentes software que me hubiera gustado conocer anteriormente para poder ofrecer soluciones computacionales a amigos y familia. Esta opción, que ofrece un servidor web muy amigable y fácil les hubiera permitido interactuar con las tareas directamente sin mi intervención, aunque la programación de las tareas las hubiese tenido que hacer yo, claramente.

Me parece un sistema muy potente y sobre todo flexible, ya que permite que el sistema de trabajo, el **worker** sea compuesto y remoto, por lo que es fácilmente escalable tanto horizontalmente como verticalmente.

Desde mi desempeño en la práctica, tengo que comentar que he usado prácticas que jamás usaría en un sistema real, como introducir la contraseña de la base de datos (aunque el usuario esté limitado a esa colección) en el código, o compartir la clave de la api, pero el tiempo disponible es el que es por desgracia.

Además, he descubierto un sistema que me permite implementar un sistema de CI/CD personalizado, que me puede servir mucho mejor que las soluciones (en planes gratuitos) en mis prácticas de empresa (al ser código confidencial).

A día de hoy, es cierto que conozco alternativas Cloud Native y Serverless como AWS CloudWatch que nos permite realizar algo similar, pero claro, como servicio y por tanto, facturado.

Bibliografía

[1] Apache Airflow Docker Compose https://airflow.apache.org/docs/apache-airflow/stable/start/docker.html