Universidad Torcuato Di Tella

MLOps

Trabajo Práctico Final

AdTech

Integrantes: Martino Boca, Gonzalo Brizuela, Gonzalo Falcon, Santiago Rodriguez

Fecha de entrega: 09/12/2024

Índice

Índice **2**
Introducción **3**
Dificultades con permisos de AWS (EC2 y ECR) **3**
Repositorio de código **5**
Pipeline: Apache Airflow **5**
Guía de Configuración y Ejecución 6
Conexión a la Instancia de AWS EC2 6
Activación de Airflow 6
Estructura del Pipeline 6
Creación de un DAG 7
Dificultades encontradas 7
Almacenamiento: Amazon S3 ..... **8**
Configuración y conexión 8
Base de datos: Amazon RDS **8**
Configuración de la base de datos 8
API **9**
Dockerización: 10
Dificultades encontradas 11
Posibles mejoras a implementar **12**
Manejo seguro de las credenciales 12
Uso de archivos .parquet en lugar de .csv 12
Reflexiones finales **12**

Introducción

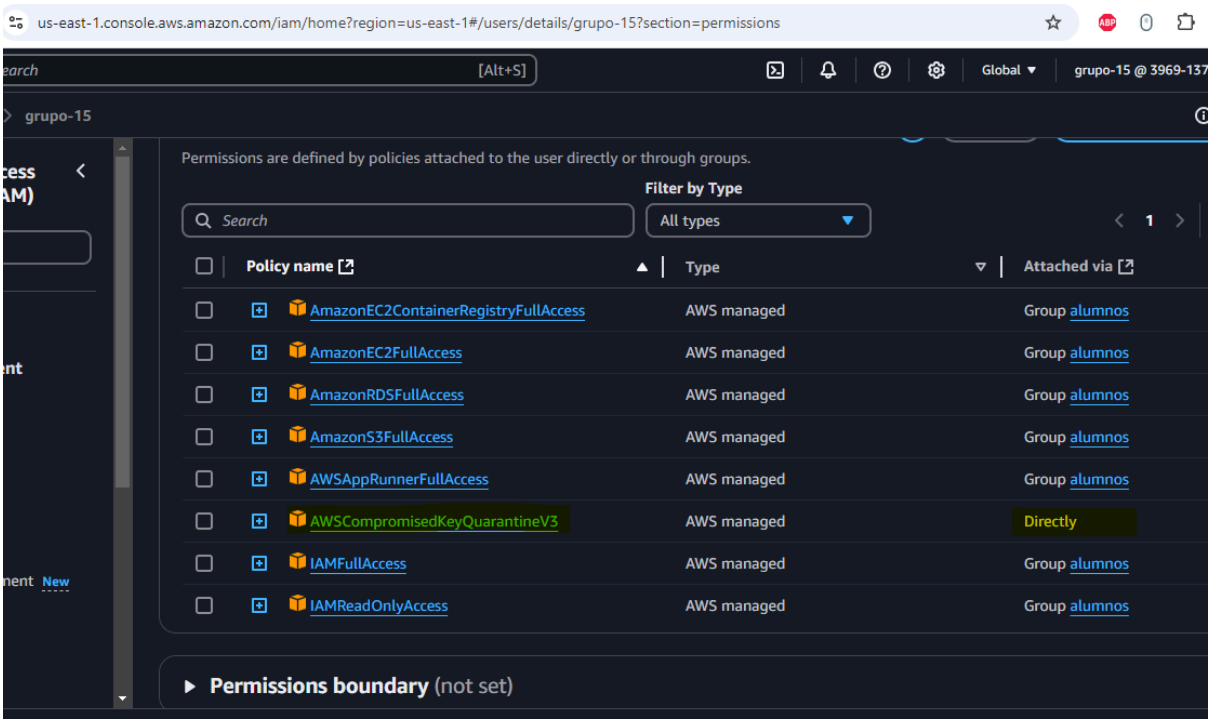
El presente informe detalla el desarrollo de un sistema de recomendación de productos en el contexto de publicidad digital, con el objetivo de optimizar la experiencia del usuario al mostrar anuncios relevantes basados en su historial de navegación. Para lograr esto, se implementó una solución que integra diversas herramientas y tecnologías, incluyendo un pipeline de procesamiento de datos utilizando Apache Airflow, una API construida con FastAPI, una base de datos PostgreSQL alojada en AWS RDS y un repositorio de archivos en AWS S3. El sistema se diseñó para recibir logs de interacciones de usuarios con productos y anuncios, procesar esta información para generar recomendaciones personalizadas, y servir estas recomendaciones a través de una interfaz accesible.

La arquitectura del sistema se compone de varios componentes clave. El pipeline, ejecutado en una instancia EC2 de AWS, se encarga de filtrar y procesar los datos crudos provenientes de un bucket S3, generando recomendaciones que se almacenan en la base de datos. La API, que está dockerizada y desplegada en AWS App Runner, permite a los clientes acceder a las recomendaciones en tiempo real mediante solicitudes HTTP. Además, se establecieron endpoints específicos para consultar estadísticas y el historial de recomendaciones. A lo largo del desarrollo del proyecto, nos encontramos con diversos errores y obstáculos técnicos que requirieron investigación y resolución creativa. Estos desafíos incluyeron problemas con la configuración del entorno, la integración entre componentes y la optimización del pipeline.

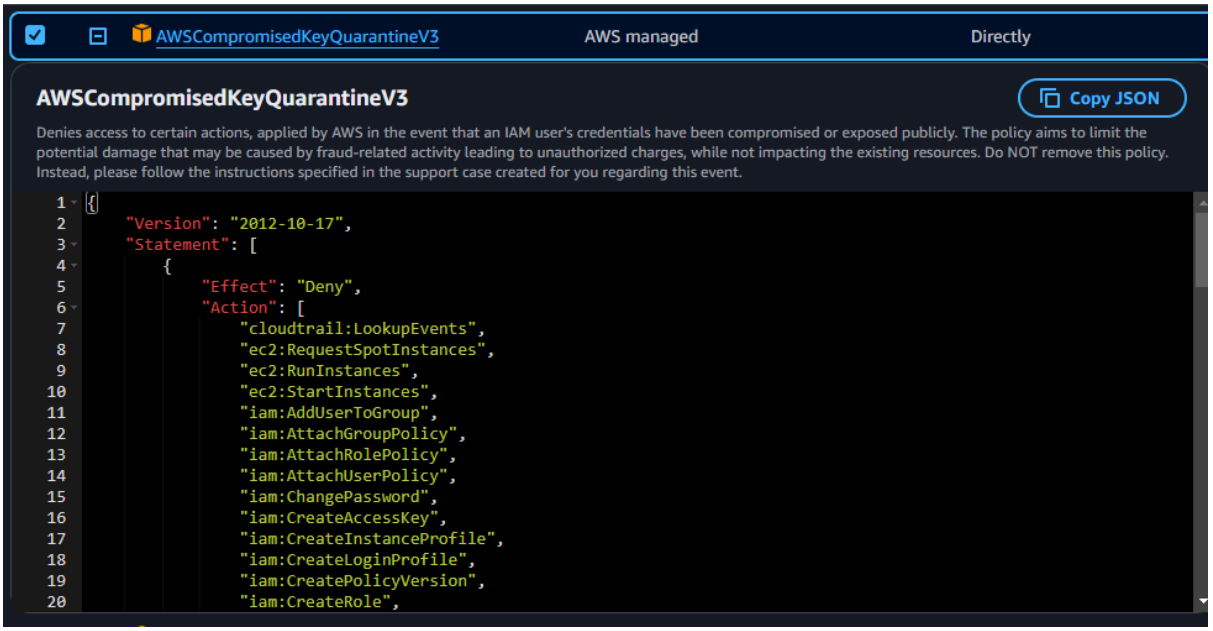
Dificultades con permisos de AWS (EC2 y ECR)

A lo largo del trabajo práctico realizamos distintas pruebas de manera local y una vez tenidas las versiones avanzadas de los códigos de la API y el DAG, buscamos probarlos en AWS. Particularmente con la instancia de EC2 logramos configurarla y hacer pruebas del Pipeline en el ambiente. En los momentos que dejamos de usar la instancia, la pausamos para no consumir recursos. El miércoles 4/12 nos encontramos con un problema que no nos permitía volver a encender la instancia:

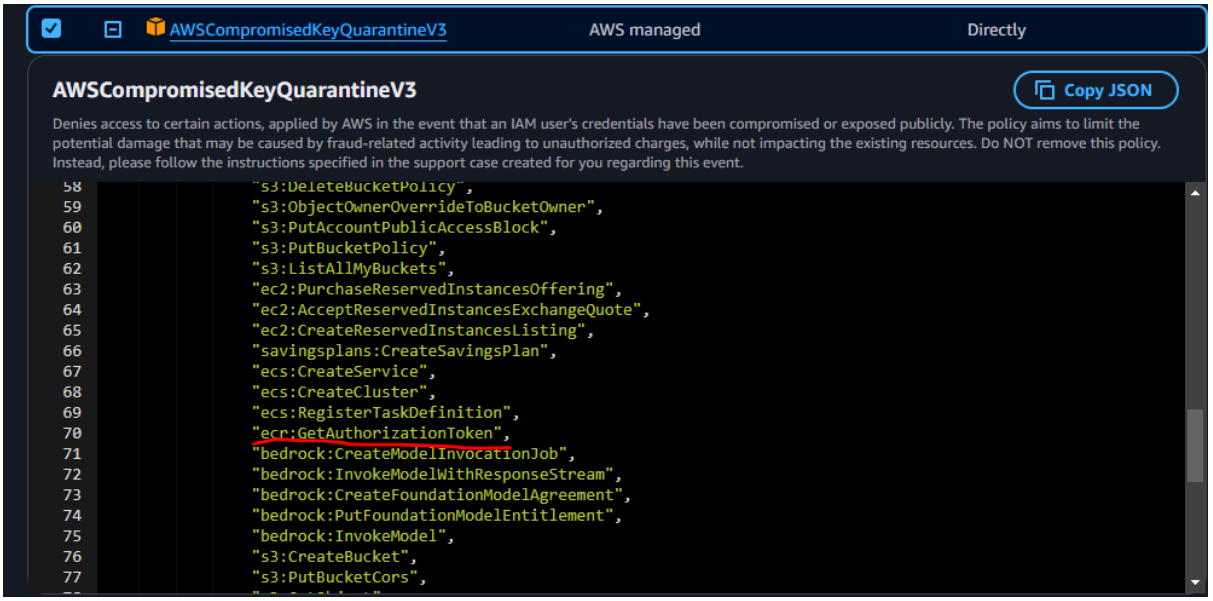
Según el mensaje de error que recibimos de AWS, vimos que se debía a un tema de permisos y por ese motivo consultamos al profesor de la materia por correo electrónico.
Como no tuvimos respuesta, insistimos en la consulta por mensaje a través del Campus Virtual. El sábado 7/12 vimos que el problema se repetía y que además nos daba error para subir una imagen de Docker a ECR:
*An error occurred (AccessDeniedException) when calling the GetAuthorizationToken operation: User: arn:aws:iam::396913735447:user/grupo-15 is not authorized to perform: ecr:GetAuthorizationToken on resource: * with an explicit deny in an identity-based policy*
Por ese motivo, revisamos las políticas de IAM y detectamos que nuestro usuario tenía una política de permisos que el resto de los usuarios no tenían: AWSCompromisedKeyQuarantineV3



Verificamos que esa política **restringe muchas acciones entre ellas correr una instancia:**



y **también verificamos que tenemos restringido el GetAuthorizationToken de ECR que nos permite loguearnos con CLI para subir en ECR la imagen de Docker:**



Por ese motivo, no pudimos avanzar en la configuración del App Runner para disponibilizar la API ni pudimos hacer nuevas pruebas sobre la instancia de EC2 con el Pipeline de Airflow.

Repositorio de código

Como repositorio de código utilizamos GitHub y creamos un proyecto prúblico: https://github.com/MartinoBoca/TP-Final-ML-Ops

Este proyecto cuenta con dos Branchs, uno con el código en python del DAG de Airflow y otro con el código en python de la API.

Pipeline: Apache Airflow

El pipeline de procesamiento de datos fue implementado utilizando Apache Airflow y desplegado en una instancia AWS EC2 (t2.small). Este pipeline se diseñó para ejecutarse diariamente y generar recomendaciones precomputadas basadas en los logs de navegación y de interacciones de usuarios con anuncios. Los resultados son almacenados en una base de datos PostgreSQL alojada en AWS RDS, desde donde la API puede acceder a ellos en tiempo real.

El flujo de datos en el pipeline puede describirse en tres etapas principales:

- Lectura de datos crudos: Los datos iniciales, que incluyen logs de vistas de productos y anuncios, junto con un listado de advertisers activos, se almacenan en un bucket de Amazon S3. El pipeline lee estos datos para iniciar el procesamiento.
- Procesamiento y filtrado: A través de varias tareas definidas en el DAG, los datos son filtrados, transformados y analizados para generar métricas clave, como los productos más vistos y los productos con mejor click-through-rate (CTR).

- Almacenamiento de resultados: Los resultados procesados se almacenan en la base de datos PostgreSQL, desde donde pueden ser accedidos por la API para servir recomendaciones a los usuarios.

Guía de Configuración y Ejecución

Conexión a la Instancia de AWS EC2

Para comenzar con la configuración de Airflow, primero se accedió a la instancia EC2 donde se ejecuta el pipeline.
Se utilizó el Key Pair PEM guardado localmente para poder conectarse a la instancia de AWS EC2 mediante SSH. Dentro de la instancia, se ingresó al directorio donde se almacenan los archivos del proyecto:
"cd grupo_15_1"
Ahi dentro, se activó el entorno virtual configurado para el proyecto:
"source bin/activate"

Activación de Airflow

Con el entorno virtual activado, se procedió a iniciar los servicios de Airflow:
"airflow webserver -D -p 8080"
Esto permitió acceder a la interfaz web de Airflow desde el navegador en la dirección:
http://3.80.108.61:8080/ (esta dirección varía según la dirección de IP de la instancia al momento de activarla), con credenciales de acceso:

- Username: admin
- Password: admin

Luego, el scheduler de Airflow, encargado de gestionar la ejecución de los DAGs, fue iniciado con el comando:
"airflow scheduler -D"

Estructura del Pipeline

El pipeline consta de las siguientes tareas principales:

- **FiltrarDatos**: Filtra los logs crudos eliminando datos relacionados con advertisers inactivos y mantiene únicamente los datos hasta el día de ejecución. En el contexto del TP los datos crudos provistos correspondían al período de ejecución del TP, pero debíamos leer solamente los datos con fecha igual o anterior a la fecha de ejecución del Pipeline, con el objetivo de calcular las métricas asumiendo que los únicos datos disponibles eran el histórico hasta la fecha de ejecución del mismo.
- **TopProducts**: Calcula los 20 productos más vistos en las páginas web de cada advertiser activo.
- **TopCTR**: Determina los 20 productos con mejor click-through-rate (CTR) por advertiser activo, basado en la relación entre clics e impresiones.
- **DBWriting**: Escribe los resultados generados por las tareas anteriores en la base de datos PostgreSQL, para que puedan ser consumidos por la API.

Estas tareas están organizadas en un DAG, que define el flujo de ejecución y las dependencias entre ellas. El DAG asegura que cada tarea se ejecute en el orden correcto y que los datos procesados en una etapa estén disponibles para las etapas posteriores.

Creación de un DAG

Con los servicios de Airflow activos, se procedió a la creación de un DAG para el pipeline. Los pasos realizados fueron:
Se accedió a la carpeta de DAGs con el comando *“cd ~/airflow/dags”.*
Ahí dentro, se creó un archivo Python para el DAG utilizando el editor nano:
“nano AdTechPipelineS3DB.py”

En la configuración del DAG tomamos la decisión de configurar una ejecución diaria a las 23 hs, con fecha de inicio 29/11/24 y configuramos el parámetro *“catchup: True”* con el objetivo de asegurar tener un histórico suficiente para que al momento de probar la API podamos traer el histórico de 7 días. Como la creación del DAG requirió de muchas iteraciones, queríamos que en cada una de esas iteraciones podamos mantener le histórico. Además en ocasiones debimos pausar la instancia de EC2 para ahorrar recursos y queríamos que al reiniciarla no perdamos alguna ejecución.

Dificultades encontradas

Durante el desarrollo del pipeline, se presentaron varios desafíos que requirieron investigación y ajustes. Al inicio, fue necesario entender la lógica detrás de los DAGs y cómo configurarlos correctamente en Airflow. Para eso, primero nos valimos de una instancia de Airflow local en nuestras computadoras de trabajo, para luego probar en EC2.
Aprendimos a definir las dependencias entre tareas y validar que las tareas se ejecuten en el orden esperado. Una dificultad asociada a esto fue el uso del parámetro *catchup: True* como indicamos anteriormente, para asegurarnos tener un histórico ininterrumpido de recomendaciones para poblar la Base de Datos. El problema con este parámetro es que las ejecuciones de las tareas de los distintos días se realizaban en simultáneo, pisando el archivo CSV generado para un día con el del día siguiente, afectando a las tareas posteriores que usaban ese archivo. De este modo, no podíamos asegurar que las recomendaciones diarias históricas fueran correctas. Para evitarlo, primero probamos con el parámetro *depends_on_past: True* para que una tarea se ejecute cuando la misma tarea en el período anterior se ejecutó correctamente. Esto no terminaba de resolver el problema, por los tiempos de ejecución de las tareas siguientes del DAG, por ese motivo recurrimos a nombrar los archivos CSV con la fecha que correspondía a la ejecución del DAG. Esto fue posible gracias al uso del parámetro *execution_date* de Airflow como input en cada una de las Tasks del DAG. Tras varias pruebas, logramos que los DAGs funcionaran correctamente consumiendo la información correspondiente al día que emulaban la ejecución de la recomendación.
Configurar la conexión con AWS S3 presentó desafíos iniciales, como la gestión de credenciales y la correcta lectura y escritura de archivos en el bucket. Fue necesario utilizar el Access Key ID y el Secret Access Key y realizar pruebas para asegurarnos de que los archivos se carguen y descarguen correctamente desde S3.
Establecer la conexión con la base de datos PostgreSQL en AWS RDS implicó configurar los parámetros de conexión, abrir los puertos necesarios en el grupo de seguridad de la

instancia RDS y ajustar las consultas SQL. Finalmente, logramos que las recomendaciones se escribieran correctamente en la base de datos. También probamos hacer pública la instancia de RDS para poder trabajar con la base de datos desde nuestra computadora y no depender de trabajar con la instancia de EC2.
Durante las pruebas, también surgieron problemas relacionados con la estructura y formato de los archivos cargados en S3. Tuvimos que ajustar las funciones de lectura y escritura para asegurarnos de que los datos fuesen procesados correctamente y que las columnas requeridas estuvieran presentes en los archivos.

Almacenamiento: Amazon S3

El almacenamiento intermedio de los datos fue implementado utilizando Amazon S3, que actúa como repositorio para los archivos crudos, resultados intermedios y datos procesados generados por el pipeline de procesamiento. Esto permitió manejar volúmenes de datos de manera eficiente y asegurar su disponibilidad para las diferentes etapas.
Durante la ejecución de las tareas del pipeline, los datos son leídos desde el bucket, procesados y luego escritos nuevamente en S3 como resultados intermedios. Finalmente, los datos procesados son utilizados por las tareas del pipeline para generar las recomendaciones que se almacenan en la base de datos.

Configuración y conexión

Para interactuar con S3, se utilizó la librería boto3, que permitió realizar operaciones como lectura y escritura de archivos en el bucket. La configuración inicial incluyó la creación del bucket en S3, la definición de políticas de acceso y la integración con el pipeline mediante las credenciales de AWS.

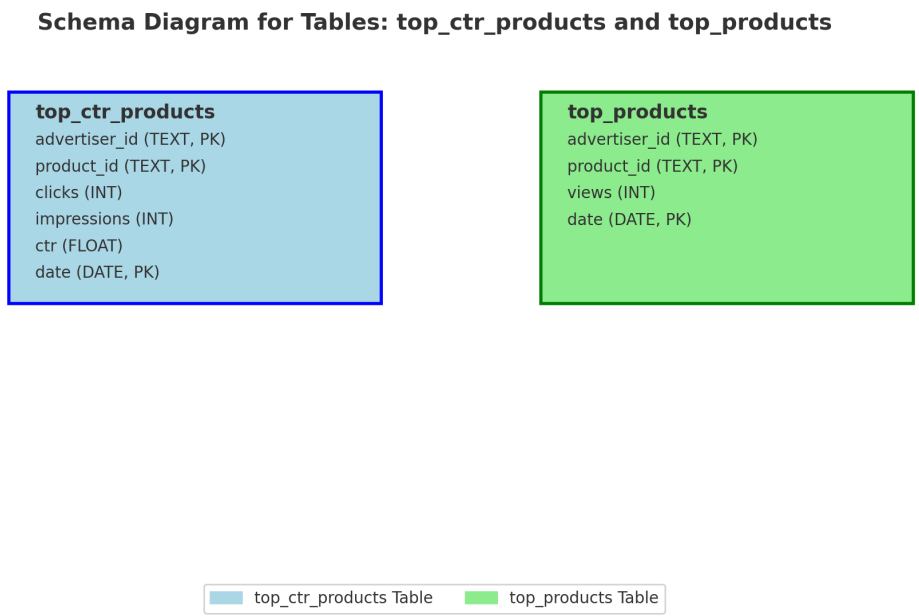
Base de datos: Amazon RDS

La base de datos utilizada en este proyecto fue implementada en Amazon RDS utilizando PostgreSQL. Su propósito principal es almacenar las recomendaciones precomputadas generadas por el pipeline de datos, para que puedan ser consumidas por la API en tiempo real. Esta elección asegura la centralización y disponibilidad de los datos, permitiendo un acceso rápido y confiable.

Configuración de la base de datos

Para configurar la base de datos en Amazon RDS, se creó una instancia de PostgreSQL con los parámetros necesarios para garantizar la compatibilidad y el rendimiento del sistema. La configuración incluyó la selección de un tamaño de instancia adecuado, la creación de un esquema inicial y la definición de las tablas requeridas para almacenar los datos de las recomendaciones. Estas tablas son:

- top_products: Almacena los 20 productos más vistos por cada advertiser activo para cada día.
- top_ctr: Contiene los 20 productos con mejor click-through-rate (CTR) por cada advertiser activo, para cada día.



El pipeline de datos y la API se conectan a la base de datos utilizando la librería psycopg2 para ejecutar consultas SQL. Durante la escritura de datos, el pipeline inserta las recomendaciones generadas en las tablas correspondientes, asegurando que estén disponibles para la API. Por otro lado, la API realiza consultas de lectura para servir las recomendaciones en tiempo real a los usuarios.

API

Para desarrollar y desplegar correctamente los endpoints que interactúan con la base de datos que contiene las recomendaciones de productos precomputadas, inicialmente se realizó un desarrollo local utilizando una base de datos de prueba para garantizar que los tres endpoints requeridos devolvieran los resultados esperados.
Los endpoints principales incluyen:

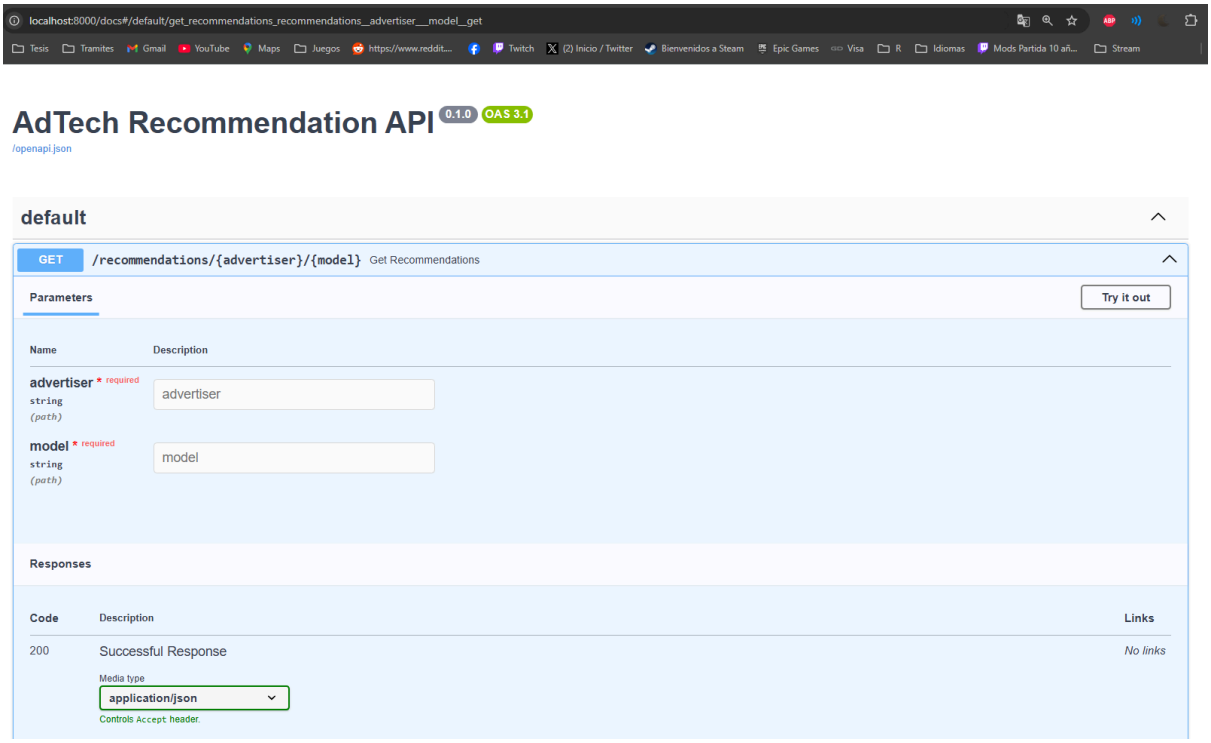
- `/recommendations/<ADV>/<Modelo>`, que devuelve las recomendaciones del día para un anunciante específico y un modelo determinado.
- `/stats/`, que proporciona estadísticas sobre las recomendaciones.
- `/history/<ADV>/`, que permite consultar el historial de recomendaciones para un anunciante en particular durante los últimos siete días.

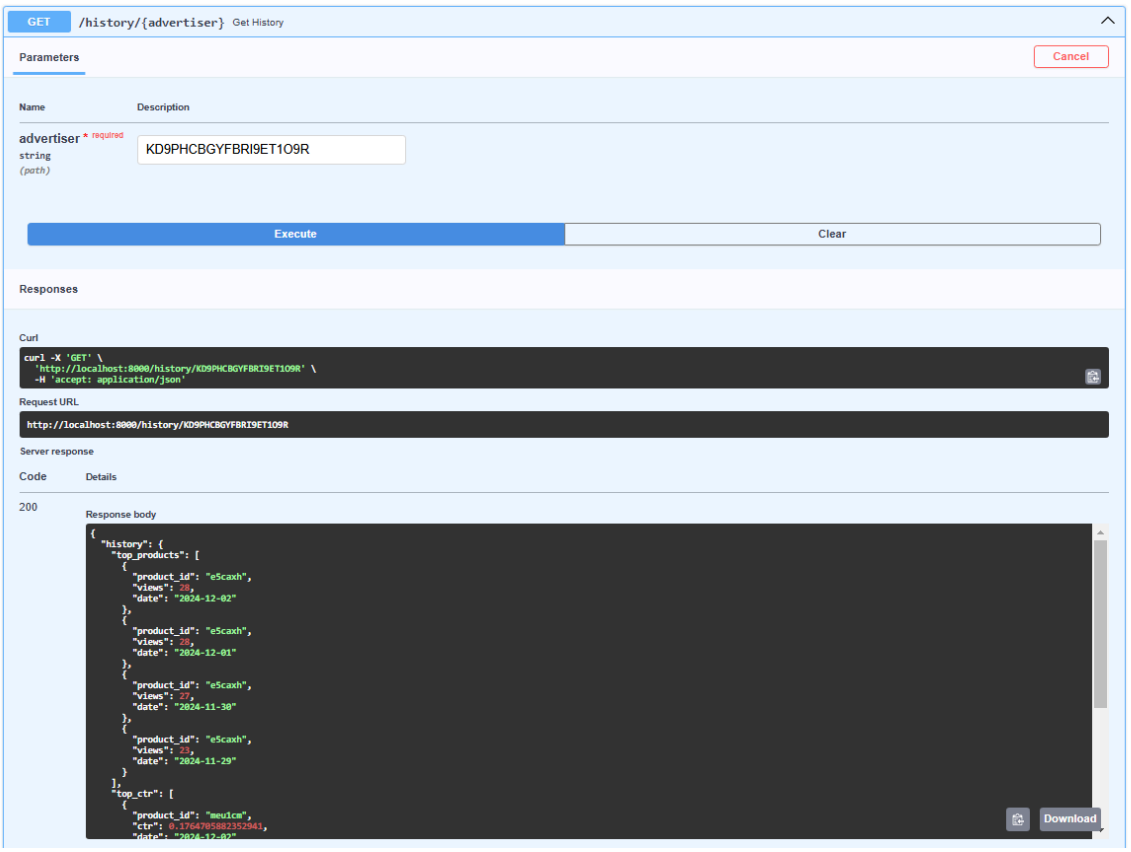
Una vez validado el funcionamiento local, se procedió a integrar la API con la base de datos alojada en AWS RDS. Para lograr esto, primero se configuró la conexión pública a la base de datos de RDS mediante la librería psycopg2, permitiendo que la API pudiera leer datos en la instancia.
Para probar los endpoints, utilizamos la herramienta de documentación de FastAPI (/docs) que genera documentación de manera automática y además permite obtener las respuestas en formato JSON. Al realizar la prueba inicial de los endpoints en forma local, las dificultades que surgieron estaban vinculadas más que nada a la estructura y diseño de la API, junto con el manejo de errores y excepciones y, posteriormente la integración a una base de datos de prueba.

Dockerización

La dockerización de la API fue un paso esencial en el proceso de despliegue, permitiendo empaquetar la aplicación y sus dependencias en un contenedor para asegurar su ejecución consistente en cualquier entorno. Para ello, se creó un archivo Dockerfile que define cómo construir la imagen de la API, utilizando como base python:3..11.10-slim-bullseye, una imagen ligera adecuada para nuestras necesidades.
En el Dockerfile, se instalaron las dependencias necesarias, incluyendo FastAPI, Uvicorn (el servidor ASGI utilizado para ejecutar la aplicación) y psycopg2. Posteriormente, se configuró el directorio de trabajo y se copió el código fuente al contenedor.
Una vez construido el contenedor, se realizaron pruebas locales para verificar que la API respondiera correctamente a las solicitudes. Esto incluyó ejecutar el contenedor en un entorno local y comprobar que los endpoints definidos funcionaran como se esperaba.

```
(test) gbrizuela@LAPTOP-CCBVKNRS:~/test/app$
(test) gbrizuela@LAPTOP-CCBVKNRS:~/test/app$ docker run -p 8000:8000 grupo_15_1:latest
INFO:     Started server process [1]
INFO:     Waiting for application startup.
INFO:     Application startup complete.
INFO:     Uvicorn running on http://0.0.0.0:8000 (Press CTRL+C to quit)
INFO:     172.17.0.1:34710 - "GET /docs HTTP/1.1" 200 OK
INFO:     172.17.0.1:34710 - "GET /openapi.json HTTP/1.1" 200 OK
INFO:     172.17.0.1:34710 - "GET /docs HTTP/1.1" 200 OK
INFO:     172.17.0.1:34710 - "GET /openapi.json HTTP/1.1" 200 OK
```





Dificultades encontradas

Uno de los errores que tuvimos fue durante el despliegue en App Runner, ya que no logramos crear el servicio debido al siguiente mensaje de error:
"12-07-2024 04:15:18 PM [AppRunner] Failed to build your application source code. Reason: Failed to execute 'build' command."
Para intentar resolver este problema, probamos diferentes comandos de instalación en el proceso de configuración del entorno, como:

- "pip install -r requirements.txt"
- "sudo yum install git -y; pip install -r requirements.txt;"

También intentamos subir la imágen de docker a ECR de AWS para luego desplegarla en App Runner, a pesar de estos intentos, el error persiste, lo que sugiere que el problema podría estar relacionado con la configuración de permisos descrita más arriba. Este inconveniente nos ha impedido completar el despliegue, a pesar de haber investigado alternativas y haber revisado documentación oficial, así como foros de soporte, para encontrar una solución adecuada.

Posibles mejoras a implementar

Detallamos algunas mejoras que no pudimos implementar y probar, dado que no podíamos acceder a la instancia de EC2 para probar, y nos demoramos haciendo muchas pruebas para habilitar el AppRunner. Más allá de que hicimos pruebas locales, no queríamos modificar el código que habíamos verificado que funcionaba en EC2.

Manejo seguro de las credenciales

Para poder hacer funcionar el código del DAG, fue necesario hardcodear las credenciales de AWS (aws_access_key_id y aws_secret_access_key) y de la base de datos en el script. Sin embargo, sabemos que esto representa un riesgo de seguridad, ya que las credenciales sensibles están expuestas directamente en el código. Una oportunidad de mejora es implementar el uso de variables de entorno o integraciones nativas de Airflow. En el caso de AWS, se puede configurar una conexión en Airflow (AWS Connection) y utilizar S3Hook en lugar de boto3 directamente, mientras que para la base de datos se puede emplear una conexión PostgreSQL utilizando PostgresHook o PostgresOperator. Esto elimina la necesidad de manejar credenciales manualmente y asegura que estén protegidas dentro del entorno gestionado de Airflow.

Uso de archivos .parquet en lugar de .csv

Para optimizar el uso de los recursos de AWS podríamos probar implementar el uso de archivos intermedios .parquet en lugar de .csv.

Reflexiones finales

En general el trabajo práctico nos ayudó a integrar las diferentes tecnologías vistas en clase. Empezar a entender y dimensionar las distintas aristas que involucran un proceso de poner en producción una solución. Poder integrar los contenidos vistos en otras materias y empezar a abandonar la perspectiva de ejecutar nuestras soluciones de manera local a trabajar en un entorno en la nube y compartido con el resto de los integrantes del grupo.
Si bien tuvimos dificultades y problemas ya mencionados. De manera local tuvimos una implementación completa combinando de cierta manera entorno virtual, airflow, trabajo mediante la consola, base de datos, la implementación de una api, etc.
Temas que quedaron fuera de la implementación creemos que es mantenimiento de la solución en producción, monitoreo, entre otros.