MEMORIA TÉCNICA DEL PROYECTO

Diseño e Implementación de una Arquitectura Cloud Híbrida para Sistemas de Recomendación

Asignatura: TSCD

Curso: 4º Grado en Ingeniería Informática

Autores: Jorge Cubero Toribio, Manuel Alejandro Torrealba Torrealba

1. Introducción y Contexto

El presente proyecto se enmarca en la finalización de los estudios de grado, con el objetivo de integrar conocimientos consolidados de ingeniería del software con nuevos paradigmas de **Computación en la Nube (Cloud Computing)** y **Big Data**.

Dado que el plan de estudios ha cubierto extensamente el desarrollo de aplicaciones monolíticas y bases de datos relacionales, este proyecto supone un reto tecnológico al introducir por primera vez una **arquitectura orientada a eventos (Event-Driven)** y el uso de servicios gestionados de AWS (Amazon Web Services). El sistema desarrollado simula un entorno real de ingesta de datos para una plataforma de streaming, utilizando grafos para modelar las preferencias de los usuarios.

2. Objetivos del Proyecto

El objetivo principal es diseñar un pipeline de datos desacoplado y escalable. Los objetivos específicos incluyen:

1. **Transición al Cloud:** Migrar de una mentalidad de "servidor local" a una arquitectura basada en servicios (SaaS/PaaS) utilizando **LocalStack** como entorno de simulación de AWS.
2. **Persistencia Políglota:** Implementar **Neo4j** (Base de Datos Orientada a Grafos) para resolver problemas de relaciones complejas que serían ineficientes en SQL tradicional.
3. **Infraestructura como Código (IaC):** Automatizar el despliegue de recursos (colas SQS, buckets S3) mediante scripts, evitando la configuración manual propensa a errores.
4. **Calidad del Software:** Aplicar patrones de diseño para la validación de datos y gestión de errores (Dead Letter Queues).

3. Arquitectura de la Solución

Para superar las limitaciones de las arquitecturas monolíticas tradicionales (donde un fallo en la base de datos detiene toda la aplicación), se ha diseñado una arquitectura de microservicios distribuida.

3.1. Diseño Conceptual

El sistema sigue el patrón **Producer-Consumer** con un intermediario de mensajería:

- **Capa de Ingesta (API REST):** Desarrollada en Python (FastAPI). Su única responsabilidad es recibir el dato y acusar recibo (ACK), garantizando alta disponibilidad.
- **Capa de Transporte (AWS SQS):** Se ha elegido SQS (Simple Queue Service) frente a otras opciones como Kafka por su curva de aprendizaje más ajustada y su integración nativa con el ecosistema AWS. Permite el desacoplamiento asíncrono.
- **Capa de Procesamiento (Workers):**

- *Quality Gate:* Filtra datos corruptos antes de que lleguen al almacenamiento.
- *Graph Ingester:* Transforma los mensajes JSON en consultas Cypher para Neo4j.

3.2. Diagrama de Flujo de Datos

1. Usuario envía valoración -> API Producer.
2. API -> Cola raw (AWS SQS).
3. Microservicio Quality Gate valida -> Cola clean o dlq.
4. Microservicio Ingester consume clean -> Base de Datos Neo4j.

4. Stack Tecnológico y Curva de Aprendizaje

El proyecto combina tecnologías familiares con nuevas herramientas Cloud exploradas durante la asignatura:

- **Lenguaje:** Python 3.10. Se eligió por su robustez en tratamiento de datos y la familiaridad previa con el lenguaje.
- **Contenerización:** Docker y Docker Compose. Fundamental para orquestar los distintos servicios (API, BD, AWS Simulado) en un entorno de desarrollo local.
- **Cloud (La Novedad):**
  - **LocalStack:** Ha sido la pieza clave para el aprendizaje. Nos ha permitido interactuar con las APIs reales de AWS (boto3) sin incurrir en costes ni requerir tarjetas de crédito, proporcionando un "sandbox" seguro para fallar y aprender.
  - **Servicios AWS:** Se ha profundizado en el uso de **SQS** (Colas) y **S3** (Almacenamiento de Objetos), comprendiendo conceptos como "visibilidad del mensaje", "retención" y "consistencia eventual".
- **Base de Datos:** Neo4j. Se seleccionó por ser el estándar industrial para sistemas de recomendación, permitiendo consultas semánticas ("*buscar películas que vieron amigos de mis amigos*") de forma nativa.

5. Retos de Implementación y Soluciones

Durante el desarrollo, nos enfrentamos a desafíos propios de la integración de sistemas distribuidos:

5.1. Configuración de Redes en Entornos Híbridos

Uno de los mayores obstáculos fue la comunicación entre los contenedores Docker y el sistema operativo anfitrión (Windows) al usar LocalStack.

- *Problema:* Las librerías de AWS intentaban resolver subdominios DNS (sqs.us-east-1...) que no son nativos en entornos locales, provocando errores de UnknownOperationException.
- *Solución:* Tras investigación y depuración, se configuró la estrategia de endpoints basada en rutas (SQS_ENDPOINT_STRATEGY=path) y se forzó el uso de IPs estáticas internas, demostrando la importancia de entender las redes en Docker.

5.2. Consistencia de Datos

Al tener un sistema asíncrono, surgió el problema de la integridad referencial (votar por una película que aún no existe en BD).

- *Solución:* Se implementó una lógica de verificación en el Ingester y un script de carga masiva (Batch Loader) para inicializar el grafo, separando la carga histórica del tráfico en tiempo real.

6. Validación y Pruebas

Para asegurar la robustez del sistema, se aplicaron diferentes estrategias de testing:

- **Pruebas de Carga (Stress Testing):** Utilizando **Locust**, simulamos la concurrencia de múltiples usuarios. Se observó que, gracias a la cola SQS, la API no se saturaba ni aumentaba su latencia, ya que la escritura en BD se hacía en segundo plano (buffering).
- **Integración Continua (CI):** Se configuró un pipeline básico en GitHub Actions para automatizar la validación del código en cada subida, acercándonos a las metodologías DevOps reales.

7. Conclusiones y Trabajo Futuro

Este proyecto ha servido como puente entre la teoría académica y la práctica profesional en la nube.

Se ha logrado implementar con éxito una arquitectura que, aunque desplegada en local, está lista para ser migrada a AWS real cambiando únicamente las credenciales de configuración.

Como conclusión principal, hemos comprobado cómo las arquitecturas desacopladas aumentan la resiliencia del software, y cómo herramientas como LocalStack democratizan el acceso al aprendizaje de tecnologías Cloud complejas. Como líneas futuras, se propone implementar el motor de recomendación sobre los datos ya almacenados en el grafo.