Universidad de Las Palmas de Gran Canaria

Escuela de Ingeniería Informática

Grado en Ciencia e Ingeniería de Datos

Flight Network Graph API

Sistema de Análisis de Redes de Vuelos mediante Grafos

Tecnologías de Servicios para Ciencia de Datos

Trabajo de Curso

Autor:
Raúl Mendoza

Fecha:
Enero 2026

Curso 2024-2025

Índice

1. Introducción **4**
1.1. Contexto del Proyecto . . . 4
1.2. Descripción del Problema . . . 4
1.3. Objetivos . . . 4
1.4. Estructura del Documento . . . 4

2. Arquitectura del Sistema **6**
2.1. Vista General . . . 6
2.2. Nivel de Aplicación . . . 6
2.2.1. Capa de Presentación . . . 6
2.2.2. Capa de Negocio . . . 6
2.2.3. Capa de Datos . . . 7
2.3. Nivel de Tecnología . . . 7
2.4. Diagrama de Componentes . . . 7
2.5. Decisiones de Diseño . . . 7

3. Tecnologías Utilizadas **9**
3.1. Lenguaje de Programación . . . 9
3.2. Bibliotecas Principales . . . 9
3.3. Servicios AWS . . . 9
3.3.1. AWS Lambda . . . 9
3.3.2. Amazon API Gateway . . . 10
3.3.3. Amazon S3 . . . 10
3.3.4. Amazon DynamoDB . . . 10
3.4. LocalStack . . . 10
3.5. Herramientas de Desarrollo . . . 11

4. Implementación del Modelo de Grafos **12**
4.1. Estructura de Datos . . . 12
4.1.1. Representación de Nodos (Aeropuertos) . . . 12
4.1.2. Representación de Aristas (Vuelos) . . . 12
4.2. Clase FlightGraph . . . 12
4.3. Algoritmos Implementados . . . 13
4.3.1. Camino Más Corto (Dijkstra) . . . 13
4.3.2. Todos los Caminos . . . 13
4.3.3. Detección de Hubs . . . 14
4.3.4. Detección de Clusters . . . 14
4.3.5. Nodos Aislados . . . 14
4.4. Datos del Sistema . . . 14

5. API REST **16**
5.1. Diseño de la API . . . 16
5.2. Endpoints Disponibles . . . 16
5.2.1. GET /airports . . . 16
5.2.2. GET /stats . . . 16
5.2.3. GET /shortest-path . . . 16
5.2.4. GET /all-paths . . . 17

5.2.5. GET /hubs . . . 17
5.2.6. GET /isolated . . . 17
5.2.7. GET /connections . . . 17
5.2.8. GET /by-degree . . . 18
5.2.9. GET /clusters . . . 18
5.2.10. GET /longest-path . . . 18
5.3. Manejo de Errores . . . 18

6. Interfaz Web **19**
6.1. Descripción General . . . 19
6.2. Características . . . 19
6.3. Secciones de la Interfaz . . . 19
6.3.1. Barra de Estadísticas . . . 19
6.3.2. Búsqueda de Rutas . . . 19
6.3.3. Visualización de Hubs . . . 19
6.3.4. Conexiones Directas . . . 19
6.3.5. Detección de Clusters . . . 20
6.4. Tecnologías del Frontend . . . 20
6.5. Comunicación con la API . . . 20

7. Infraestructura Cloud **21**
7.1. AWS SAM Template . . . 21
7.2. LocalStack Setup . . . 21
7.3. Servidor Flask para Desarrollo . . . 22

8. Pipeline CI/CD **23**
8.1. GitHub Actions . . . 23
8.2. Flujo del Pipeline . . . 24
8.3. Beneficios del CI/CD . . . 24

9. Pruebas **25**
9.1. Estrategia de Testing . . . 25
9.2. Tests del Modelo (test_graph.py) . . . 25
9.3. Tests de la API (test_api.py) . . . 25
9.4. Resultados de las Pruebas . . . 26
9.5. Cobertura de Tests . . . 26

10. Manual de Instalación y Uso **27**
10.1. Requisitos Previos . . . 27
10.2. Instalación . . . 27
10.3. Ejecución Local . . . 27
10.3.1. Opción 1: Servidor Flask . . . 27
10.3.2. Opción 2: LocalStack . . . 27
10.4. Ejecutar Tests . . . 28
10.5. Uso de la API . . . 28
10.6. Uso del Frontend . . . 28

11. Conclusiones **29**
11.1. Objetivos Alcanzados 29
11.2. Lecciones Aprendidas 29
11.3. Trabajo Futuro 29
11.4. Valoración Personal 30

12. Referencias **31**

A. Estructura del Repositorio **32**

B. Ejemplo Completo de Sesión **32**

1. Introducción

1.1. Contexto del Proyecto

En el ámbito de la ciencia de datos, los grafos constituyen una de las estructuras de datos más versátiles y potentes para modelar relaciones complejas entre entidades. Desde las redes sociales hasta los sistemas de transporte, pasando por las redes de distribución de energía, los grafos nos permiten representar y analizar conexiones que de otro modo serían difíciles de visualizar y procesar.

Este proyecto surge como respuesta a la necesidad de aplicar los conocimientos adquiridos en la asignatura de Tecnologías de Servicios para Ciencia de Datos, combinando el uso de servicios en la nube con técnicas de desarrollo modernas como DevOps y CI/CD.

1.2. Descripción del Problema

El problema que abordamos consiste en modelar una red de vuelos como un grafo, donde:

- Los **aeropuertos** se representan como **nodos** del grafo
- Los **vuelos directos** entre aeropuertos se representan como **aristas**
- La **distancia** entre aeropuertos se almacena como peso de la arista

Sobre esta estructura, implementamos una API REST que permite realizar diversas operaciones de análisis, como encontrar la ruta más corta entre dos aeropuertos, identificar los aeropuertos con más conexiones (hubs), detectar comunidades o clusters, y más.

1.3. Objetivos

Los objetivos principales de este proyecto son:

1. Diseñar e implementar un sistema basado en grafos para modelar redes de vuelos
2. Desarrollar una API REST con múltiples endpoints para operaciones sobre el grafo
3. Desplegar la solución utilizando servicios de AWS (o LocalStack como alternativa local)
4. Implementar un pipeline de CI/CD para automatizar pruebas y despliegue
5. Crear una interfaz web para interactuar visualmente con el sistema
6. Documentar la arquitectura siguiendo buenas prácticas de ingeniería de software

1.4. Estructura del Documento

Este documento se organiza de la siguiente manera:

- **Sección 2:** Describe la arquitectura del sistema desde los niveles de aplicación y tecnología

- **Sección 3:** Detalla las tecnologías y herramientas utilizadas
- **Sección 4:** Explica la implementación del modelo de grafos
- **Sección 5:** Documenta la API REST y sus endpoints
- **Sección 6:** Presenta la interfaz web desarrollada
- **Sección 7:** Describe la infraestructura cloud y LocalStack
- **Sección 8:** Explica el pipeline de CI/CD implementado
- **Sección 9:** Presenta las pruebas realizadas y sus resultados
- **Sección 10:** Proporciona un manual de instalación y uso
- **Sección 11:** Conclusiones y trabajo futuro

2. Arquitectura del Sistema

2.1. Vista General

La arquitectura del sistema sigue un patrón serverless, aprovechando los servicios gestionados de AWS para minimizar la complejidad operativa y maximizar la escalabilidad. El diseño se ha realizado pensando en la separación de responsabilidades y la facilidad de mantenimiento.

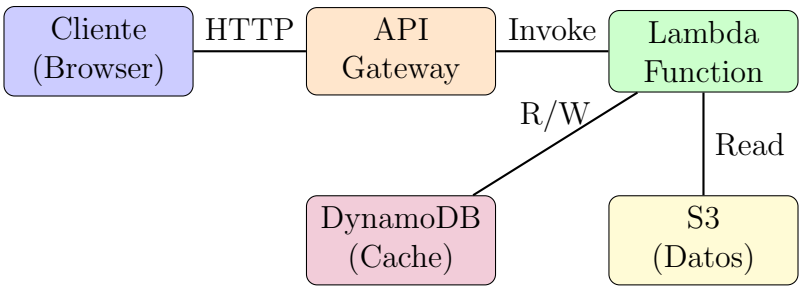

Figura 1: Arquitectura general del sistema

2.2. Nivel de Aplicación

El nivel de aplicación describe los componentes de software y sus interacciones. Nuestro sistema se compone de tres capas principales:

2.2.1. Capa de Presentación

La capa de presentación está formada por:

- **API Gateway:** Punto de entrada para todas las peticiones HTTP. Se encarga del enrutamiento, la autenticación (si se configura) y el formateo de respuestas.
- **Frontend Web:** Interfaz de usuario desarrollada en HTML/CSS/JavaScript que consume la API y presenta los datos de forma visual.

2.2.2. Capa de Negocio

La capa de negocio contiene la lógica principal de la aplicación:

- **Lambda Function (graph_operations.py):** Función serverless que procesa las peticiones, parsea los parámetros y orquesta las llamadas al modelo.
- **FlightGraph (graph.py):** Clase que encapsula el grafo y proporciona métodos para todas las operaciones de análisis.
- **Helpers (helpers.py):** Funciones auxiliares para carga de datos, formateo de respuestas y utilidades comunes.

2.2.3. Capa de Datos

La capa de datos gestiona la persistencia:

- **Archivos JSON:** Almacenan los datos de aeropuertos y vuelos en formato estructurado.
- **S3 Bucket:** En producción, los datos se almacenarían en S3 para mayor durabilidad.
- **DynamoDB:** Tabla opcional para cachear resultados de consultas frecuentes.

2.3. Nivel de Tecnología

El nivel de tecnología describe la infraestructura física y los servicios utilizados:

Capa	Servicio AWS	Alternativa Local
Compute	AWS Lambda	LocalStack Lambda
API	API Gateway	LocalStack API Gateway
Storage	Amazon S3	LocalStack S3
Database	DynamoDB	LocalStack DynamoDB
Logs	CloudWatch	Logs locales

Cuadro 1: Mapeo de servicios AWS y alternativas locales

2.4. Diagrama de Componentes

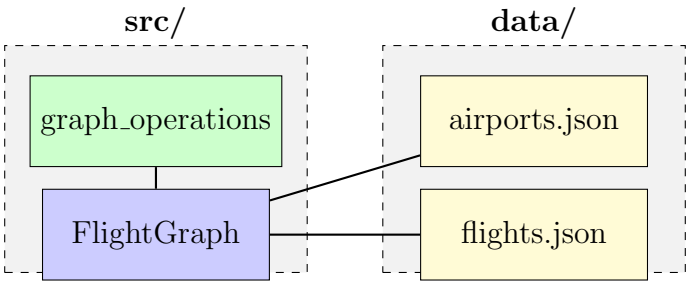

Figura 2: Diagrama de componentes principales

2.5. Decisiones de Diseño

Durante el desarrollo del proyecto, tomamos varias decisiones arquitectónicas importantes:

1. **Arquitectura Serverless:** Elegimos Lambda sobre EC2 porque el patrón de uso esperado es esporádico, y Lambda permite escalar automáticamente sin gestionar servidores.
2. **Grafo en Memoria:** El grafo se carga en memoria al iniciar la Lambda. Para grafos pequeños-medianos (como el nuestro con 15 aeropuertos), esto es más eficiente que consultar una base de datos en cada operación.

3. **NetworkX:** Utilizamos esta biblioteca por su madurez, amplia documentación y porque implementa los algoritmos que necesitamos (Dijkstra, detección de comunidades, etc.).

4. **LocalStack para Desarrollo:** En lugar de usar AWS directamente durante el desarrollo, usamos LocalStack para evitar costes y tener un entorno reproducible.

5. **API RESTful:** Seguimos las convenciones REST para que la API sea predecible y fácil de consumir desde cualquier cliente.

3. Tecnologías Utilizadas

3.1. Lenguaje de Programación

Python 3.11 fue elegido como lenguaje principal por varias razones:

- Amplio ecosistema de bibliotecas para ciencia de datos y grafos
- Soporte nativo en AWS Lambda
- Sintaxis clara y facilidad de mantenimiento
- Comunidad activa y abundante documentación

3.2. Bibliotecas Principales

Biblioteca	Versión	Propósito
NetworkX	3.2.1	Creación y manipulación de grafos. Proporciona algoritmos como Dijkstra, BFS, DFS, detección de comunidades, etc.
Boto3	1.34.0	SDK de AWS para Python. Permite interactuar con servicios como S3, DynamoDB, Lambda.
Flask	3.0.0	Framework web ligero para crear el servidor de desarrollo local y servir el frontend.
Flask-CORS	-	Extensión para habilitar Cross-Origin Resource Sharing en Flask.
Pytest	7.4.3	Framework de testing para Python. Facilita la escritura y ejecución de pruebas unitarias.
Requests	2.31.0	Biblioteca para realizar peticiones HTTP, útil para testing de integración.

Cuadro 2: Bibliotecas Python utilizadas

3.3. Servicios AWS

3.3.1. AWS Lambda

AWS Lambda es un servicio de computación serverless que ejecuta código en respuesta a eventos. Características clave:

- **Sin servidores:** No hay que aprovisionar ni gestionar servidores
- **Escalado automático:** Se escala automáticamente según la demanda
- **Pago por uso:** Solo se paga por el tiempo de ejecución
- **Integración:** Se integra nativamente con otros servicios AWS

Configuración de nuestra Lambda:

```
FlightGraphFunction:
  Type: AWS::Serverless::Function
  Properties:
    FunctionName: flight-graph-operations
    Runtime: python3.11
    MemorySize: 256
    Timeout: 30
    Handler: lambdas.graph_operations.lambda_handler
```

Listing 1: Configuración de Lambda en SAM

3.3.2. Amazon API Gateway

API Gateway actúa como puerta de entrada para las peticiones HTTP:

- Enruta las peticiones a la Lambda correspondiente
- Gestiona throttling y cuotas
- Proporciona métricas y logging
- Soporta HTTPS para comunicaciones seguras

3.3.3. Amazon S3

Simple Storage Service se utiliza para:

- Almacenar los archivos JSON con datos de aeropuertos y vuelos
- Hospedar el frontend como sitio web estático (opcional)
- Alta durabilidad (99.999999999 %)

3.3.4. Amazon DynamoDB

Base de datos NoSQL utilizada opcionalmente para:

- Cachear resultados de consultas frecuentes
- Almacenar configuraciones dinámicas
- Latencia de milisegundos

3.4. LocalStack

LocalStack es una herramienta que emula los servicios de AWS localmente:

```
version: '3.8'
services:
  localstack:
    image: localstack/localstack:latest
    container_name: localstack-flights
    ports:
      - "4566:4566"
```

```
    environment:
      - SERVICES=lambda,apigateway,s3,dynamodb
      - DEBUG=1
    volumes:
      - "./infrastructure/localstack:/etc/localstack/init/ready.d"
      - "./src:/opt/code/src"
      - "./data:/opt/code/data"
```

Listing 2: docker-compose.yml para LocalStack

Ventajas de usar LocalStack:

- Desarrollo sin costes de AWS
- Entorno reproducible para todo el equipo
- Testing de integración sin afectar producción
- Ciclos de desarrollo más rápidos

3.5. Herramientas de Desarrollo

Herramienta	**Uso**
Git	Control de versiones
GitHub	Repositorio remoto y CI/CD con GitHub Actions
Docker	Contenedorización para LocalStack
VS Code	Editor de código principal
AWS SAM	Framework para definir infraestructura serverless

Cuadro 3: Herramientas de desarrollo

4. Implementación del Modelo de Grafos

4.1. Estructura de Datos

El grafo se modela utilizando la clase `FlightGraph`, que encapsula un grafo no dirigido de NetworkX. Cada nodo representa un aeropuerto y cada arista un vuelo directo.

4.1.1. Representación de Nodos (Aeropuertos)

Cada aeropuerto se almacena como un nodo con los siguientes atributos:

```
{
    "code": "MAD",           # Codigo IATA del aeropuerto
    "name": "Adolfo Suarez Madrid-Barajas",
    "city": "Madrid",
    "country": "Spain"
}
```

Listing 3: Estructura de un aeropuerto

4.1.2. Representación de Aristas (Vuelos)

Cada vuelo se representa como una arista con peso:

```
{
    "origin": "MAD",
    "destination": "JFK",
    "distance": 5768    # Distancia en kilometros
}
```

Listing 4: Estructura de un vuelo

4.2. Clase FlightGraph

La clase principal que gestiona el grafo:

```
import networkx as nx
from typing import List, Dict, Optional

class FlightGraph:
    def __init__(self):
        self.graph = nx.Graph()

    def add_airport(self, code: str, name: str,
                    city: str, country: str) -> None:
        """Anade un aeropuerto (nodo) al grafo."""
        self.graph.add_node(code, name=name,
                            city=city, country=country)

    def add_flight(self, origin: str, destination: str,
                   distance: int) -> None:
        """Anade un vuelo (arista) entre dos aeropuertos."""
        self.graph.add_edge(origin, destination, weight=distance)

    def load_data(self, airports: List[Dict],
                  flights: List[Dict]) -> None:
```

```
        """Carga los datos de aeropuertos y vuelos al grafo."""
        for airport in airports:
            self.add_airport(
                airport["code"], airport["name"],
                airport["city"], airport["country"]
            )
        for flight in flights:
            self.add_flight(
                flight["origin"], flight["destination"],
                flight["distance"]
            )
```

Listing 5: Clase FlightGraph (extracto)

4.3. Algoritmos Implementados

4.3.1. Camino Más Corto (Dijkstra)

Para encontrar la ruta óptima entre dos aeropuertos, utilizamos el algoritmo de Dijkstra:

```
def shortest_path(self, origin: str,
                  destination: str) -> Optional[List[str]]:
    """Devuelve el camino mas corto entre dos aeropuertos."""
    try:
        return nx.shortest_path(self.graph, origin,
                                destination, weight="weight")
    except nx.NetworkXNoPath:
        return None

def shortest_path_distance(self, origin: str,
                           destination: str) -> Optional[int]:
    """Devuelve la distancia del camino mas corto."""
    try:
        return nx.shortest_path_length(self.graph, origin,
                                       destination, weight="weight")
    except nx.NetworkXNoPath:
        return None
```

Listing 6: Implementación del camino más corto

El algoritmo de Dijkstra tiene una complejidad de $O((V + E) \log V)$ donde V es el número de nodos y E el número de aristas.

4.3.2. Todos los Caminos

Para obtener todas las rutas posibles entre dos aeropuertos:

```
def all_paths(self, origin: str, destination: str,
              max_length: int = 5) -> List[List[str]]:
    """Devuelve todos los caminos entre dos aeropuertos."""
    try:
        paths = list(nx.all_simple_paths(
            self.graph, origin, destination, cutoff=max_length
        ))
        return paths
    except nx.NetworkXNoPath:
        return []
```

Listing 7: Obtención de todos los caminos

Nota: Limitamos la longitud máxima para evitar explosión combinatoria.

4.3.3. Detección de Hubs

Los hubs son los aeropuertos con mayor número de conexiones:

```
def get_hubs(self, top_n: int = 5) -> List[Dict]:
    """Devuelve los aeropuertos con mas conexiones."""
    degrees = sorted(self.graph.degree(),
                     key=lambda x: x[1], reverse=True)
    return [{"airport": code, "connections": degree}
            for code, degree in degrees[:top_n]]
```

Listing 8: Identificación de hubs

4.3.4. Detección de Clusters

Utilizamos el algoritmo de modularidad greedy para detectar comunidades:

```
def get_clusters(self) -> List[List[str]]:
    """Detecta comunidades/clusters en el grafo."""
    communities = nx.community.greedy_modularity_communities(
        self.graph
    )
    return [list(community) for community in communities]
```

Listing 9: Detección de clusters

Este algoritmo agrupa los nodos que están más densamente conectados entre sí.

4.3.5. Nodos Aislados

Aeropuertos sin conexiones directas:

```
def get_isolated_nodes(self) -> List[str]:
    """Devuelve aeropuertos sin conexiones."""
    return list(nx.isolates(self.graph))
```

Listing 10: Obtención de nodos aislados

4.4. Datos del Sistema

El sistema incluye datos de 15 aeropuertos internacionales:

Código	**Ciudad**	**País**
MAD	Madrid	España
BCN	Barcelona	España
LPA	Las Palmas	España
TFN	Tenerife	España
LHR	Londres	Reino Unido
CDG	París	Francia
FCO	Roma	Italia
FRA	Frankfurt	Alemania
AMS	Ámsterdam	Países Bajos
JFK	Nueva York	Estados Unidos
MIA	Miami	Estados Unidos
LAX	Los Ángeles	Estados Unidos
IST	Estambul	Turquía
DXB	Dubái	Emiratos Árabes
SIN	Singapur	Singapur

Cuadro 4: Aeropuertos incluidos en el sistema

Y 27 conexiones de vuelos directos entre ellos, formando una red realista de transporte aéreo.

5. API REST

5.1. Diseño de la API

La API sigue los principios REST (Representational State Transfer):

- **Stateless:** Cada petición es independiente
- **Recursos:** Endpoints organizados por recursos (airports, paths, etc.)
- **Métodos HTTP:** Uso de GET para consultas
- **JSON:** Formato de intercambio de datos

5.2. Endpoints Disponibles

5.2.1. GET /airports

Lista todos los aeropuertos del sistema.

```
{
    "airports": [
        {
            "code": "MAD",
            "name": "Adolfo Suarez Madrid-Barajas",
            "city": "Madrid",
            "country": "Spain"
        },
        ...
    ]
}
```

Listing 11: Ejemplo de respuesta /airports

5.2.2. GET /stats

Devuelve estadísticas generales del grafo.

```
{
    "total_airports": 15,
    "total_flights": 27,
    "density": 0.257,
    "is_connected": true
}
```

Listing 12: Ejemplo de respuesta /stats

5.2.3. GET /shortest-path

Calcula la ruta más corta entre dos aeropuertos.
Parámetros:

- `origin` (requerido): Código del aeropuerto origen
- `destination` (requerido): Código del aeropuerto destino

```
{
    "origin": "MAD",
    "destination": "SIN",
    "path": ["MAD", "LHR", "DXB", "SIN"],
    "distance": 12596,
    "stops": 2
}
```

Listing 13: Ejemplo: /shortest-path?origin

5.2.4. GET /all-paths

Devuelve todos los caminos posibles entre dos aeropuertos.
Parámetros:

- `origin` (requerido): Código del aeropuerto origen
- `destination` (requerido): Código del aeropuerto destino
- `max_length` (opcional, default=5): Longitud máxima del camino

5.2.5. GET /hubs

Devuelve los aeropuertos con más conexiones.
Parámetros:

- `top` (opcional, default=5): Número de hubs a devolver

```
{
    "hubs": [
        {"airport": "MAD", "connections": 8},
        {"airport": "LHR", "connections": 6},
        {"airport": "CDG", "connections": 5}
    ]
}
```

Listing 14: Ejemplo: /hubs?top

5.2.6. GET /isolated

Devuelve aeropuertos sin conexiones directas.

5.2.7. GET /connections

Devuelve las conexiones directas de un aeropuerto.
Parámetros:

- `airport` (requerido): Código del aeropuerto

```
{
    "airport": "MAD",
    "connections": ["BCN", "LPA", "TFN", "LHR", "CDG", "FCO", "JFK", "
   FRA"],
    "total": 8
}
```

Listing 15: Ejemplo: /connections?airport

5.2.8. GET /by-degree

Filtra aeropuertos por número de conexiones.
Parámetros:

- `degree` (requerido): Número exacto de conexiones

5.2.9. GET /clusters

Detecta comunidades o clusters en el grafo.

```
{
    "total_clusters": 3,
    "clusters": [
        ["BCN", "TFN", "MAD", "CDG", "FCO", "LPA"],
        ["FRA", "SIN", "AMS", "IST", "DXB", "LHR"],
        ["LAX", "JFK", "MIA"]
    ]
}
```

Listing 16: Ejemplo de respuesta /clusters

5.2.10. GET /longest-path

Devuelve el camino más largo (sin ciclos) entre dos aeropuertos.

5.3. Manejo de Errores

La API devuelve códigos HTTP estándar:

Código	Significado	Ejemplo
200	OK	Operación exitosa
400	Bad Request	Faltan parámetros requeridos
404	Not Found	No se encontró ruta o recurso
500	Server Error	Error interno del servidor

Cuadro 5: Códigos de respuesta HTTP

Ejemplo de error:

```
{
    "error": "origin and destination required"
}
```

Listing 17: Respuesta de error

6. Interfaz Web

6.1. Descripción General

Se ha desarrollado una interfaz web moderna y responsiva que permite interactuar visualmente con la API. El frontend está construido con HTML5, CSS3 y JavaScript vanilla, sin dependencias externas.

6.2. Características

- **Diseño responsivo:** Se adapta a diferentes tamaños de pantalla
- **Tema oscuro:** Interfaz con colores oscuros para reducir fatiga visual
- **Interactiva:** Actualizaciones en tiempo real sin recargar la página
- **Estadísticas en vivo:** Muestra métricas del grafo al cargar

6.3. Secciones de la Interfaz

6.3.1. Barra de Estadísticas

Muestra en tiempo real:

- Total de aeropuertos en el sistema
- Número de vuelos/conexiones
- Densidad del grafo
- Número de clusters detectados

6.3.2. Búsqueda de Rutas

Permite seleccionar aeropuerto de origen y destino para calcular:

- Ruta más corta
- Distancia total
- Número de escalas

6.3.3. Visualización de Hubs

Muestra los aeropuertos más conectados con:

- Código del aeropuerto
- Número de conexiones
- Ranking visual

6.3.4. Conexiones Directas

Permite consultar los vuelos directos desde cualquier aeropuerto.

6.3.5. Detección de Clusters

Visualiza las comunidades detectadas, agrupando aeropuertos por regiones geográficas.

6.4. Tecnologías del Frontend

- **HTML5:** Estructura semántica del contenido
- **CSS3:** Estilos modernos con variables CSS, flexbox y grid
- **JavaScript ES6+:** Lógica de interacción y llamadas a la API usando fetch
- **Google Fonts:** Tipografía Inter para mejor legibilidad

6.5. Comunicación con la API

El frontend utiliza la API Fetch de JavaScript para comunicarse con el backend:

```
async function findShortestPath() {
    const origin = document.getElementById('origin').value;
    const destination = document.getElementById('destination').value;

    const response = await fetch(
        `${API_BASE}/shortest-path?origin=${origin}&destination=${
   destination}`
    );
    const data = await response.json();

    // Mostrar resultados...
}
```

Listing 18: Ejemplo de llamada a la API

7. Infraestructura Cloud

7.1. AWS SAM Template

Utilizamos AWS SAM (Serverless Application Model) para definir la infraestructura como código:

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Description: Flight Network Graph API

Globals:
  Function:
    Timeout: 30
    Runtime: python3.11
    MemorySize: 256

Resources:
  FlightGraphFunction:
    Type: AWS::Serverless::Function
    Properties:
      FunctionName: flight-graph-operations
      CodeUri: ../../src/
      Handler: lambdas.graph_operations.lambda_handler
      Events:
        Airports:
          Type: Api
          Properties:
            Path: /airports
            Method: get
        ShortestPath:
          Type: Api
          Properties:
            Path: /shortest-path
            Method: get
        # ... mas endpoints

  FlightDataBucket:
    Type: AWS::S3::Bucket
    Properties:
      BucketName: !Sub flight-data-${AWS::AccountId}

Outputs:
  ApiUrl:
    Description: URL de la API
    Value: !Sub https://${ServerlessRestApi}.execute-api.${AWS::Region}.
   amazonaws.com/Prod/
```

Listing 19: template.yaml (extracto)

7.2. LocalStack Setup

Para desarrollo local, utilizamos un script de inicialización:

```
#!/bin/bash
echo "Configurando recursos en LocalStack..."

```

```
# Crear bucket S3 para datos
awslocal s3 mb s3://flight-data

# Subir datos al bucket
awslocal s3 cp /opt/code/data/airports.json s3://flight-data/
awslocal s3 cp /opt/code/data/flights.json s3://flight-data/

# Crear tabla DynamoDB para cache
awslocal dynamodb create-table \
    --table-name FlightCache \
    --key-schema AttributeName=route,KeyType=HASH \
    --attribute-definitions AttributeName=route,AttributeType=S \
    --billing-mode PAY_PER_REQUEST

echo "LocalStack configurado correctamente!"
```

Listing 20: setup.sh para LocalStack

7.3. Servidor Flask para Desarrollo

Para facilitar el desarrollo y testing, incluimos un servidor Flask:

```
from flask import Flask, request, jsonify, send_from_directory
from flask_cors import CORS

app = Flask(__name__, static_folder='frontend')
CORS(app)

@app.route('/')
def index():
    return send_from_directory('frontend', 'index.html')

@app.route('/airports', methods=['GET'])
def get_airports():
    return jsonify({'airports': graph.get_all_airports()})

# ... mas endpoints

if __name__ == '__main__':
    print("Flight Network Graph API")
    print(f"API: http://localhost:5000")
    app.run(debug=True, port=5000)
```

Listing 21: app.py (extracto)

8. Pipeline CI/CD

8.1. GitHub Actions

Implementamos un pipeline de integración y despliegue continuo usando GitHub Actions:

```yaml
name: CI/CD Pipeline

on:
  push:
    branches: [main]
  pull_request:
    branches: [main]

jobs:
  test:
    runs-on: ubuntu-latest

    steps:
      - uses: actions/checkout@v4

      - name: Set up Python
        uses: actions/setup-python@v5
        with:
          python-version: '3.11'

      - name: Install dependencies
        run: |
          python -m pip install --upgrade pip
          pip install -r requirements.txt

      - name: Run tests
        run: |
          pytest tests/ -v --tb=short

      - name: Run linter
        run: |
          pip install flake8
          flake8 src/ --max-line-length=100

  deploy:
    needs: test
    runs-on: ubuntu-latest
    if: github.ref == 'refs/heads/main'

    steps:
      - uses: actions/checkout@v4

      - name: Install SAM CLI
        run: pip install aws-sam-cli

      - name: Build and Deploy
        run: |
          sam build -t infrastructure/aws/template.yaml
          sam deploy --no-confirm-changeset \
                     --stack-name flight-graph-api
        env:
```

```
52            AWS_ACCESS_KEY_ID: ${{ secrets.AWS_ACCESS_KEY_ID }}
53            AWS_SECRET_ACCESS_KEY: ${{ secrets.AWS_SECRET_ACCESS_KEY }}
```

Listing 22: .github/workflows/ci-cd.yml

8.2. Flujo del Pipeline

1. **Trigger:** El pipeline se activa con cada push a main o pull request
2. **Checkout:** Descarga el código del repositorio
3. **Setup:** Configura Python 3.11
4. **Install:** Instala las dependencias del proyecto
5. **Test:** Ejecuta todos los tests con pytest
6. **Lint:** Verifica el estilo del código con flake8
7. **Deploy:** Si los tests pasan y es la rama main, despliega a AWS

8.3. Beneficios del CI/CD

- **Automatización:** No hay intervención manual en el proceso
- **Consistencia:** Cada despliegue sigue el mismo proceso
- **Calidad:** No se despliega código que no pase los tests
- **Rapidez:** Feedback inmediato sobre el estado del código
- **Trazabilidad:** Historial completo de todos los despliegues

9. Pruebas

9.1. Estrategia de Testing

Implementamos pruebas a dos niveles:

1. **Tests unitarios:** Prueban el modelo del grafo de forma aislada
2. **Tests de integración:** Prueban la API completa

9.2. Tests del Modelo (test_graph.py)

```python
import pytest
from models.graph import FlightGraph

@pytest.fixture
def sample_graph():
    """Crea un grafo de ejemplo para tests."""
    graph = FlightGraph()
    graph.add_airport("MAD", "Madrid", "Madrid", "Spain")
    graph.add_airport("BCN", "Barcelona", "Barcelona", "Spain")
    graph.add_airport("LHR", "London", "London", "UK")
    graph.add_flight("MAD", "BCN", 500)
    graph.add_flight("MAD", "LHR", 1200)
    return graph

def test_shortest_path(sample_graph):
    """Test camino mas corto."""
    path = sample_graph.shortest_path("BCN", "LHR")
    assert path == ["BCN", "MAD", "LHR"]

def test_get_hubs(sample_graph):
    """Test obtener hubs."""
    hubs = sample_graph.get_hubs(2)
    assert hubs[0]["airport"] == "MAD"
    assert hubs[0]["connections"] == 2
```

Listing 23: Tests del modelo de grafo

9.3. Tests de la API (test_api.py)

```python
def test_get_airports():
    """Test endpoint /airports."""
    event = {'path': '/airports', 'httpMethod': 'GET'}
    response = lambda_handler(event, None)

    assert response['statusCode'] == 200
    body = json.loads(response['body'])
    assert 'airports' in body
    assert len(body['airports']) > 0

def test_shortest_path():
    """Test endpoint /shortest-path."""
    event = {
        'path': '/shortest-path',
```

```
        'httpMethod': 'GET',
        'queryStringParameters': {
            'origin': 'MAD',
            'destination': 'JFK'
        }
    }
    response = lambda_handler(event, None)

    assert response['statusCode'] == 200
    body = json.loads(response['body'])
    assert 'path' in body
```

Listing 24: Tests de la API

9.4. Resultados de las Pruebas

Al ejecutar `pytest tests/ -v`, obtenemos:

```
tests/test_api.py::test_get_airports PASSED
tests/test_api.py::test_get_stats PASSED
tests/test_api.py::test_shortest_path PASSED
tests/test_api.py::test_shortest_path_missing_params PASSED
tests/test_api.py::test_get_hubs PASSED
tests/test_api.py::test_get_connections PASSED
tests/test_api.py::test_get_clusters PASSED
tests/test_api.py::test_endpoint_not_found PASSED
tests/test_graph.py::test_add_airport PASSED
tests/test_graph.py::test_add_flight PASSED
tests/test_graph.py::test_shortest_path PASSED
tests/test_graph.py::test_shortest_path_no_route PASSED
tests/test_graph.py::test_get_connections PASSED
tests/test_graph.py::test_get_isolated_nodes PASSED
tests/test_graph.py::test_get_hubs PASSED
tests/test_graph.py::test_get_nodes_by_degree PASSED
tests/test_graph.py::test_all_paths PASSED
tests/test_graph.py::test_graph_stats PASSED

================== 18 passed in 0.32s ==================
```

Listing 25: Resultados de pytest

Resultado: 18 tests pasados, 0 fallidos.

9.5. Cobertura de Tests

Los tests cubren:

- Todos los endpoints de la API
- Todas las operaciones del modelo de grafos
- Casos de error (parámetros faltantes, rutas inexistentes)
- Casos límite (nodos aislados, grafos vacíos)

10. Manual de Instalación y Uso

10.1. Requisitos Previos

- Python 3.11 o superior
- Docker y Docker Compose (para LocalStack)
- Git

10.2. Instalación

```
# 1. Clonar el repositorio
git clone https://github.com/raulmendoza21/flight-network-graph-api.git
cd flight-network-graph-api

# 2. Crear entorno virtual
python -m venv .venv

# 3. Activar entorno virtual
# Windows:
.venv\Scripts\activate
# Linux/Mac:
source .venv/bin/activate

# 4. Instalar dependencias
pip install -r requirements.txt
```

Listing 26: Pasos de instalación

10.3. Ejecución Local

10.3.1. Opción 1: Servidor Flask

```
python app.py
```

Listing 27: Ejecutar servidor Flask

Esto inicia el servidor en `http://localhost:5000`

10.3.2. Opción 2: LocalStack

```
# Iniciar LocalStack
docker-compose up -d

# Verificar que esta corriendo
docker ps
```

Listing 28: Ejecutar con LocalStack

10.4. Ejecutar Tests

```
# Todos los tests
pytest tests/ -v

# Solo tests del modelo
pytest tests/test_graph.py -v

# Solo tests de la API
pytest tests/test_api.py -v
```

Listing 29: Ejecutar tests

10.5. Uso de la API

Ejemplos de uso con curl:

```
# Listar aeropuertos
curl http://localhost:5000/airports

# Buscar ruta mas corta
curl "http://localhost:5000/shortest-path?origin=MAD&destination=JFK"

# Ver hubs
curl "http://localhost:5000/hubs?top=5"

# Ver conexiones de un aeropuerto
curl "http://localhost:5000/connections?airport=MAD"

# Detectar clusters
curl http://localhost:5000/clusters
```

Listing 30: Ejemplos de uso de la API

10.6. Uso del Frontend

1. Iniciar el servidor Flask: `python app.py`
2. Abrir navegador en `http://localhost:5000`
3. Usar la interfaz para:
   - Buscar rutas entre aeropuertos
   - Ver estadísticas del grafo
   - Consultar hubs y conexiones
   - Visualizar clusters

11. Conclusiones

11.1. Objetivos Alcanzados

Este proyecto ha cumplido satisfactoriamente todos los objetivos planteados:

1. **Sistema de grafos funcional:** Implementamos un modelo robusto capaz de representar redes de vuelos y realizar operaciones complejas sobre ellas.
2. **API REST completa:** Desarrollamos 10 endpoints que cubren todas las operaciones requeridas, desde búsqueda de rutas hasta detección de comunidades.
3. **Infraestructura cloud:** Configuramos el despliegue tanto en LocalStack (desarrollo) como en AWS (producción) usando SAM.
4. **CI/CD automatizado:** Implementamos un pipeline con GitHub Actions que ejecuta tests automáticamente y despliega a producción.
5. **Interfaz visual:** Creamos un frontend web atractivo y funcional para interactuar con el sistema.
6. **Documentación completa:** Este documento describe en detalle toda la arquitectura y decisiones de diseño.

11.2. Lecciones Aprendidas

Durante el desarrollo del proyecto, aprendimos:

- La importancia de diseñar la arquitectura antes de empezar a codificar
- Cómo utilizar servicios serverless de AWS de forma efectiva
- El valor de las pruebas automatizadas para mantener la calidad del código
- La utilidad de LocalStack para desarrollo sin incurrir en costes de AWS
- Buenas prácticas de CI/CD con GitHub Actions

11.3. Trabajo Futuro

Posibles mejoras y extensiones:

- Añadir más aeropuertos y vuelos para aumentar la complejidad del grafo
- Implementar visualización gráfica del grafo con D3.js o similar
- Añadir autenticación de usuarios
- Implementar caché en DynamoDB para mejorar rendimiento
- Crear aplicación móvil que consuma la API
- Añadir métricas adicionales como centralidad de intermediación

11.4. Valoración Personal

Este proyecto ha sido una excelente oportunidad para aplicar de forma práctica los conceptos aprendidos en la asignatura. La combinación de grafos, servicios cloud y DevOps proporciona una experiencia realista de lo que implica desarrollar aplicaciones modernas orientadas a ciencia de datos.

12. Referencias

1. AWS Lambda Documentation. https://docs.aws.amazon.com/lambda/
2. NetworkX Documentation. https://networkx.org/documentation/
3. LocalStack Documentation. https://docs.localstack.cloud/
4. Flask Documentation. https://flask.palletsprojects.com/
5. GitHub Actions Documentation. https://docs.github.com/en/actions
6. AWS SAM Documentation. https://docs.aws.amazon.com/serverless-application-model/
7. Pytest Documentation. https://docs.pytest.org/

A. Estructura del Repositorio

```
flight-network-graph-api/
|-- .github/
|   |-- workflows/
|       |-- ci-cd.yml          # Pipeline CI/CD
|-- data/
|   |-- airports.json          # Datos de aeropuertos
|   |-- flights.json           # Datos de vuelos
|-- docs/
|   |-- ARCHITECTURE.md        # Documentacion arquitectonica
|   |-- GUIA_PROYECTO.md       # Guia del proyecto
|   |-- memoria.tex            # Este documento
|-- frontend/
|   |-- index.html             # Interfaz web
|-- infrastructure/
|   |-- aws/
|   |   |-- template.yaml      # SAM template
|   |-- localstack/
|       |-- setup.sh           # Script de inicializacion
|-- src/
|   |-- lambdas/
|   |   |-- graph_operations.py  # Handler Lambda
|   |-- models/
|   |   |-- graph.py           # Modelo del grafo
|   |-- utils/
|       |-- helpers.py         # Funciones auxiliares
|-- tests/
|   |-- test_api.py            # Tests de la API
|   |-- test_graph.py          # Tests del modelo
|-- .gitignore
|-- app.py                     # Servidor Flask
|-- docker-compose.yml         # Config LocalStack
|-- README.md
|-- requirements.txt           # Dependencias Python
|-- test_local.py              # Script de prueba rapida
```

Listing 31: Estructura de carpetas del proyecto

B. Ejemplo Completo de Sesión

```
$ python test_local.py

=== Test: GET /airports ===
Status: 200
Total aeropuertos: 15

=== Test: GET /shortest-path MAD -> JFK ===
Ruta: MAD -> JFK
Distancia: 5768 km
Escalas: 0

=== Test: GET /hubs (top 5) ===
MAD: 8 conexiones
LHR: 6 conexiones
```

```
CDG: 5 conexiones
JFK: 5 conexiones
BCN: 4 conexiones

=== Test: GET /stats ===
Aeropuertos: 15
Vuelos: 27
Densidad: 0.257

=== Test: GET /clusters ===
Total clusters: 3
  Cluster 1: ['BCN', 'TFN', 'MAD', 'CDG', 'FCO', 'LPA']
  Cluster 2: ['FRA', 'SIN', 'AMS', 'IST', 'DXB', 'LHR']
  Cluster 3: ['LAX', 'JFK', 'MIA']

Todos los tests pasaron!
```

Listing 32: Ejemplo de uso del sistema