GraphWord-MJ

Data Science Services Technologies
Degree in Data Science and Engineering
Las Palmas de Gran Canaria University
Maye Maye Ahmed Saleck
Joaquín Ibáñez Penalva

January 2025

1. Introduction

In the field of *Data Science Services Technologies*, we explore the use of cloud services (especially AWS) and techniques of continuous integration and delivery (CI/CD) using DevOps tools. The main assignment of the course is designed to integrate both aspects, combining the use of cloud services and automation with DevOps.

Context

This project addresses the need to manage and operate on **graphs**, a type of data structure essential for modeling complex relationships in numerous domains—examples include social networks, transportation networks, and recommendation systems. By constructing a system that builds, stores, and queries graphs in a distributed architecture (deployed on AWS), students gain hands-on experience in designing and deploying data-focused services. Continuous integration and deployment further ensure that updates to the system are automatically built, tested, and delivered.

Problem Statement

In this specific scenario, we are tasked with creating a system that constructs a graph from sets of words, where two words become connected by an edge if they differ by exactly one letter. For instance, *dig* and *dog* share an edge, as they only differ in one character. Initially, we start with words of three letters to validate the model, and then expand to larger word lengths to demonstrate scalability and the increasing complexity of the resulting graph.

Objectives

1. **Graph Construction**
    - Build and maintain dictionaries of words obtained from diverse sources (local dictionaries, Project Gutenberg books).
    - Filter words based on certain validity criteria (e.g., length, pronounceability).
    - Generate a graph (node = word, edge = one-letter difference).
2. **API Development**
    - Offer an API that allows real-time programmatic interaction with the graph.
    - Provide endpoints for minimum path calculation, all paths between two nodes, maximum distance, cluster identification, nodes by connectivity degree, and detection of isolated nodes.
    - Ensure the API runs locally for development and can also be deployed in a cloud-based environment.
3. **Deployment and Automation**
    - Implement a distributed architecture on AWS that can scale efficiently to support large amounts of data.
    - Guarantee the system remains flexible and robust through best practices in DevOps.
    - Employ continuous integration and deployment (CI/CD) pipelines to automate building, testing, and deployment.

2. Methodology

In this section, we describe the **hardware/software specifications**, **tools**, and **processes** used throughout the development of the project. We also outline the **structure** of the code and the **environments** in which it was developed.

1. Team and Hardware Specifications

1. **Team Composition**
   The project was developed by two team members, each working on different operating systems: **macOS** and **Windows.**

2. **AWS Academy Sandbox**
   Both team members had access to an **AWS Academy sandbox** environment, enabling them to launch EC2 instances, configure security groups, and test the final AWS deployment of the API. The sandbox used was from Cloud Developing course.

2. Development Environment and Tools

- **Visual Studio Code (VS Code)**
  - Chosen as the main IDE for its flexibility, extensive extension marketplace, and robust support for Python, JSON-based Terraform configurations, and other relevant file types (e.g., Markdown, YAML for CI pipelines).
  - Extensions like the *Python* plugin provided in-editor linting and debugging, while Terraform extensions helped highlight syntax and validate infrastructure scripts.
- **Python 3.x**
  - Python served as the core language for the main functionalities:
    1. Retrieving words from local dictionaries or Project Gutenberg sources.
    2. Constructing the word-based graph.
    3. Serving the API endpoints (via Flask and Gunicorn).
- **Terraform**
  - Used to define and provision AWS resources, including EC2 instances, security groups, API Gateway, etc.
  - The workflow typically involved:
    1. `terraform init` to download necessary providers.
    2. `terraform plan` to preview changes.
    3. `terraform apply` to create or modify AWS resources.
- **GitHub (Version Control & CI/CD)**
  - Code committed to a **GitHub repository**.
  - GitHub Actions configured to run tests (using `pytest`) for validating endpoints and ensuring the code remains functional.

3. Project and Code Structure

The project is logically divided into folders and scripts, each serving a distinct purpose in the **data→graph→API** pipeline:

1. **`datalake/`**
    - It is not really a datalake as the name was left over from a previous infrastructure approach, since it only stores the name of the book/file from which the words have been extracted and added to the datamart.
    - For future versions, we would consider changing this folder to metadata or something similar.
2. **`datamart/`**
    - Contains processed sets of words (e.g., `words_3.txt`, `words_4.txt`, `words_5.txt`), grouped by word length.
    - Ensures no duplicates when appending new words.
3. **`app/` Folder**
    - **`main.py`**: Coordinates the process of reading local dictionaries or downloading Project Gutenberg books, extracting words, and populating the `datamart/`.
    - **`initialize_graph.py`**: Reads all the words from `datamart/`, constructs a node-edge representation (picking up words that differ by one letter), and serializes this graph (e.g., `graph.pkl`).
    - **`api/`**: Hosts the Flask-based API (`api.py`) and any supporting modules (e.g., endpoints, initialization routines).
4. **`word_sources/`**
    - Custom Python classes that abstract the retrieval of words from different sources:
        - **Local Dictionary**: Directly from a `.txt` file. Only the local path to that file is needed. It doesn't have to be a local dictionary, it can be a book for example.
        - **Project Gutenberg**: Using a URL to download and parse text, filtering words by length and validity.
5. **`graph/`**
    - Houses the logic for building, storing, and analyzing the graph of words. Key files include:
        - **graph.py**: Defines the core `Graph` class (backed by a NetworkX graph) with methods to add nodes/edges if words differ by one letter.
        - **node.py**: Wraps each word in a `Node` object, implementing equality and hashing so they can be uniquely managed within NetworkX.
        - **graph_manager.py**: Offers a simple layer (`GraphManager`) to build a graph from a list of words and retrieve the final `Graph`.
        - **graph_analyzer.py**: Provides analytical routines, such as finding shortest paths, longest paths, clusters, and other metrics, plus a `visualize_graph` function (using matplotlib) for debugging or demos.

6. **`terraform/`**
    - Houses the Infrastructure as Code scripts:
        - **`main.tf`**: Define EC2 instance, security groups, provider settings, etc.
    - This folder is essential for provisioning the AWS environment that runs the API in production.

4. API Endpoints

The local or AWS-deployed Flask server exposes the following routes (from **`api/api.py`**):

1. **`GET /`**
    - Returns a basic welcome message and a list of available endpoints.
2. **`GET /shortest-path?word1=...&word2=...`**
    - Finds the shortest path between two words if both exist in the graph.
3. **`GET /all-paths?word1=...&word2=...&limit=10`**
    - Retrieves all simple paths (up to the specified limit) between two words.
4. **`GET /maximum-distance(?word1=..&word2=..&limit=..)`**
    - If `word1` & `word2` are given, returns the longest simple path between them (cut off by `limit` if provided).
    - Otherwise, returns the overall longest path in the entire graph.
5. **`GET /clusters`**
    - Lists connected components (clusters) in the graph.
6. **`GET /high-connectivity?degree=2`**
    - Returns nodes with degree >= `degree` (default 2).
7. **`GET /nodes-by-degree?degree=n`**
    - Returns nodes whose degree == `n`.
8. **`GET /isolated-nodes`**
    - Lists words that have no edges (isolated nodes in the graph).

5. Development Approach

1. **Local Iteration & Testing**
    - Each developer ran partial tests using `pytest` or manual curls against the local Flask server.
    - Frequent commits to GitHub for version control.
2. **Cloud Deployment**
    - On successful local tests, the team proceeded with `terraform plan` and `terraform apply`.
    - The **AWS** sandbox environment provided by AWS Academy was used for spinning up the instance, setting up Nginx + Gunicorn, and verifying the live endpoints.

3. **Time Buffer for AWS Readiness**
    - After `terraform apply`, the system allowed for **1-2 minutes** of wait time to ensure the instance had completed user_data scripts, thus preventing "Connection Refused" errors in the immediate tests.

6. Testing and Verification

- **Manual Endpoint Checks**:
    - Once the instance is up, each endpoint (`/shortest-path`, `/clusters`, etc.) is tested using `curl` or a browser.
- **Automated Tests** (Continuous Integration):
    - GitHub Actions configured to:
        1. Spin up the environment or at least run local tests.
        2. Execute `pytest` to validate the main logic, such as the graph building or some mocked API calls.
- **Observations**:
    - The key challenge was ensuring the Nginx + Gunicorn deployment is stable on the ephemeral environment and that the instance is truly ready before tests run.

7. AWS Academy Lab Usage

- Both developers used the **AWS Academy** lab environment to manage credentials, create EC2 resources, and run Terraform commands in a sandbox.
- This allowed for a practical approach to "infrastructure as code" without incurring personal AWS costs during development.

3. System Design and Architecture

In this section, we provide a high-level view of the **GraphWord-MJ** system's design—how the application is organized, how data flows from source to graph, and how the infrastructure is set up both **locally** and on **AWS**.

1. Overall Architecture

1. **Local (Development) Environment**
    - **Data Flow**:
        - The user runs `app/main.py` to either load a local dictionary or download text from Project Gutenberg.
        - Processed words end up in `datamart/`, grouped by length (e.g., `words_3.txt`).
        - `app/initialize_graph.py` then reads these files, builds a `networkx.Graph`, and serializes it into `graph.pkl`.
    - **API**:
        - A Flask-based API (`app/api/api.py`) reads `graph.pkl` on startup.
        - Gunicorn (locally or in production) can serve the Flask app.
    - **Local Testing**:
        - Developers test endpoints by hitting `http://127.0.0.1:5000/` on their macOS or Windows machine. (in macOS it was needed to adjust some things cause 5000 port was not working)
2. **AWS Deployment**
    - **Terraform**:
        - Provisions an EC2 instance, a Security Group, and optionally sets up API Gateway as a proxy for the instance.
        - The instance is spun up with an **Amazon Linux** AMI, which installs Python, Git, and Nginx.
        - A user-data script clones the GitHub repo (`graphword-mj`), installs dependencies, and starts Gunicorn behind Nginx (port 80).
    - **Infrastructure Flow**:
        - **Security Group**: Allows inbound traffic on ports **80** (HTTP) and **5000** (debug) plus SSH (22) if needed.
        - **EC2 Instance**: Pulls code from GitHub, runs `gunicorn api:app`, fronted by **Nginx**.
        - **API Gateway (Optional)**: A route in API Gateway can forward requests to the EC2's public DNS on port 80.
    - **Access**:
        - Once Terraform finishes, it outputs `ec2_public_dns`.
        - The user can `curl http://ec2-xx-xx-xx-xx.compute-1.amazonaws.com` to reach the same endpoints as local usage.

2. Key Components and Responsibilities

1. **Data Pipeline**
    - **Word Processing Scripts**:
        - `main.py` obtains words from either `LocalDictionaryWordSource` or `ProjectGutenbergWordSource`.
        - `word_manager.py` consolidates them into `datamart/`.

2. **Graph Construction**
    - **`initialize_graph.py`**: Reads the processed words, builds a **NetworkX** graph (`graph.graph`), and serializes it as `graph.pkl`.
    - **`graph/`** folder:
        - **`graph.py`**: Base `Graph` class, storing a `networkx.Graph`.
        - **`node.py`**: Simple `Node` class wrapping a single word (hashable for NetworkX).
        - **`graph_manager.py`**: A convenience builder for inserting words/edges in bulk.
        - **`graph_analyzer.py`**: Methods for shortest paths, maximum distance, clusters, or even `visualize_graph()` with matplotlib.

3. **API Service**
    - **`api/api.py`**:
        - **Load Phase**: On startup, attempts to load `graph.pkl` into memory (or logs an error if none found).
        - **Flask Endpoints**:
            - `/shortest-path`, `/all-paths`, `/maximum-distance`, `/clusters`, `/high-connectivity`, `/nodes-by-degree`, `/isolated-nodes`.
            - Requests are processed by `graph_analyzer.py` to compute the needed results.
    - **Local**: Typically run with `python3 api.py` or `gunicorn --bind 127.0.0.1:5000 api:app`.
    - **Production (AWS)**: The same API runs behind Nginx.

4. **Infrastructure (Terraform)**
    - **`terraform/main.tf`** sets up:
        - **EC2** with user-data script for installing Python, Git, Nginx, Gunicorn, and cloning the repo.
        - **Security Group** (ports 80, 5000, 22).
        - **API Gateway** integration (optionally) to forward requests to the instance's public DNS.
    - **CI/CD**: The `.github/workflows/ci.yml` uses GitHub Actions to:
        - Perform `terraform apply`,
        - Wait 90 seconds,
        - Run integration tests (`pytest` in `tests/` folder) against the newly provisioned instance.

3. Architectural Flow Summary

- **Local**:
    1. Developer runs `main.py` → Creates `datamart/` word files.
    2. Developer runs `initialize_graph.py` → Produces `graph.pkl`.
    3. `api.py` loads `graph.pkl` → Endpoints are tested at `http://127.0.0.1:5000/`.
- **AWS**:
    1. `terraform apply` → Creates EC2 with user-data.
    2. Instance boots → Clones code, starts Gunicorn + Nginx.
    3. User obtains `ec2_public_dns` → Access `http://ec2-xx-xx-xx-xx.compute-1.amazonaws.com/<endpoint>`.
    4. API responds with graph-based queries.

This design ensures **separation of concerns** (data ingestion, graph building, and API serving) while using a standard DevOps approach (Terraform for infra, GitHub Actions for CI, Nginx + Gunicorn for stable production hosting).

4. Architecture diagrams

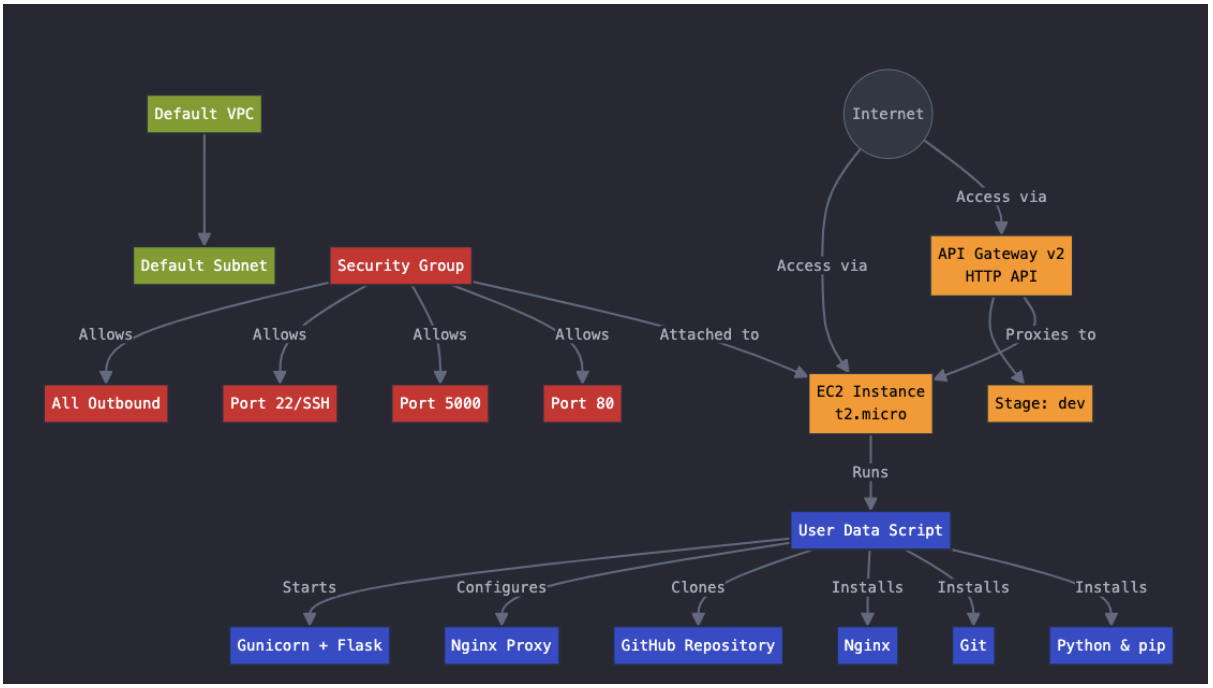

Figure 1

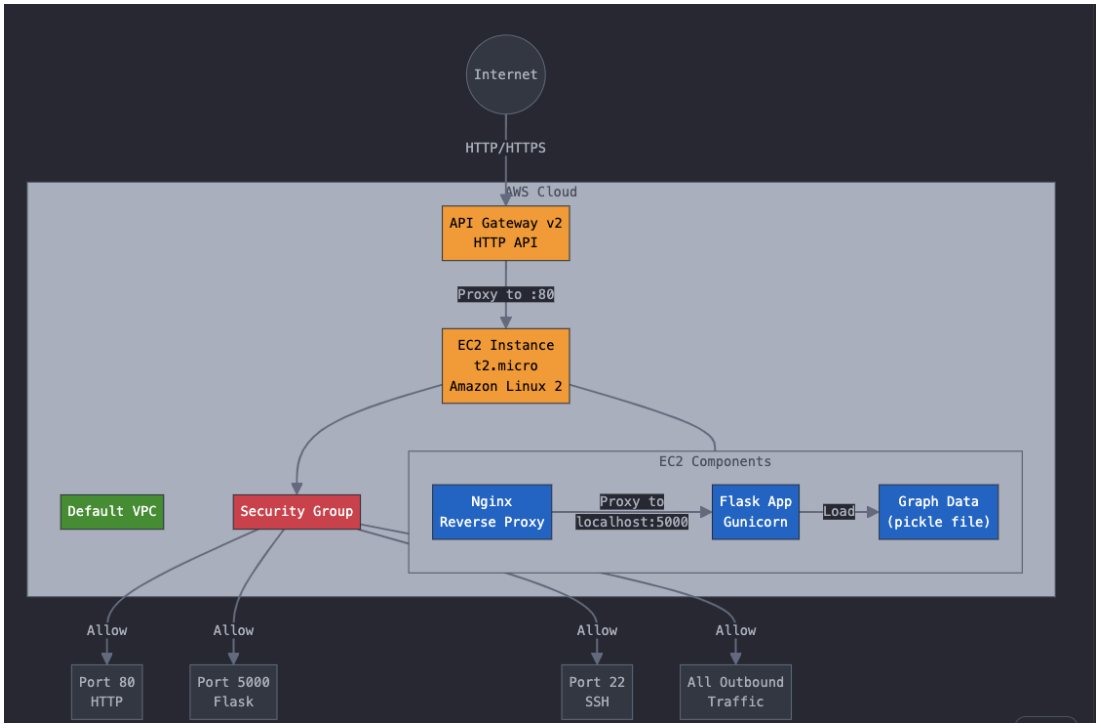

Figure 2

Explanation of Figure 1

This figure represents the complete flow from the Internet to the application deployed on AWS. It is useful for understanding the general architecture of the infrastructure. Below are its components:

Internet

- The entry point to the system. Users access the application using HTTP/HTTPS protocols.

API Gateway

- API Gateway v2 serves as the first point of contact with the AWS infrastructure.
- It proxies requests to port 80 of the EC2 instance.
- Acts as a component that abstracts the complexity of the backend and exposes the application as a public endpoint.

EC2 Instance

- A virtual server (t2.micro, based on Amazon Linux 2) hosting the application and related services.
- The EC2 instance has several internal components and specific configurations enabled:
    - **Nginx Reverse Proxy:** Receives HTTP requests and forwards them to Gunicorn.
    - **Gunicorn:** Serves as the Flask API application server on port 5000 (port 5001 on MacOS).
    - The application accesses data from a pickle file (represented as "Graph Data").

Security Groups and Network Configuration

- The figure illustrates how the Default VPC (Virtual Private Cloud) and Security Groups enable external connections:
    - **Port 80:** Opens the application to HTTP traffic.
    - **Port 5000:** Allows for testing or direct access to the Flask server.
    - **Port 22:** Enables SSH access for administration.

Explanation of Figure 2

This figure delves deeper into the technical details and key configurations of the system. Here's its breakdown:

Networking (VPC, Subnet, and Security)

- **Default VPC:** The infrastructure is deployed within the default VPC.
- **Default Subnet:** Automatically assigned to the EC2 instance to provide a public address and external access.
- **Security Group:**
    - **Allowed Ports:**
        - **22:** For SSH (server administration).
        - **5000:** For direct testing of Flask.
        - **80:** For the application in production.
    - Allows all outgoing traffic to download updates and dependencies.

EC2 Instance and Configuration with `user_data`

- Details how the EC2 instance is automatically configured upon startup:
    - **Dependency Installation:** Python, PIP, Git, and Nginx.
    - **Repository Cloning:** Fetches the source code from GitHub.
    - **Project Dependency Installation:** Uses pip to install Flask, Gunicorn, and other Python libraries.
    - **Nginx Configuration:**
        - Configured as a reverse proxy to route traffic from port 80 to the Flask server (port 5000).
    - **Flask Server Startup with Gunicorn:**
        - Launches the Flask API using Gunicorn, listening for requests on `localhost:5000`.
        - This Gunicorn instance runs in the background using `nohup`.

API Gateway

- Public traffic is routed to the EC2 instance via the API Gateway's HTTP API.
- Configures the `/` route and the proxy to forward requests to Nginx on port 80.

Finally, an image of how the GitHub Actions of the project works:

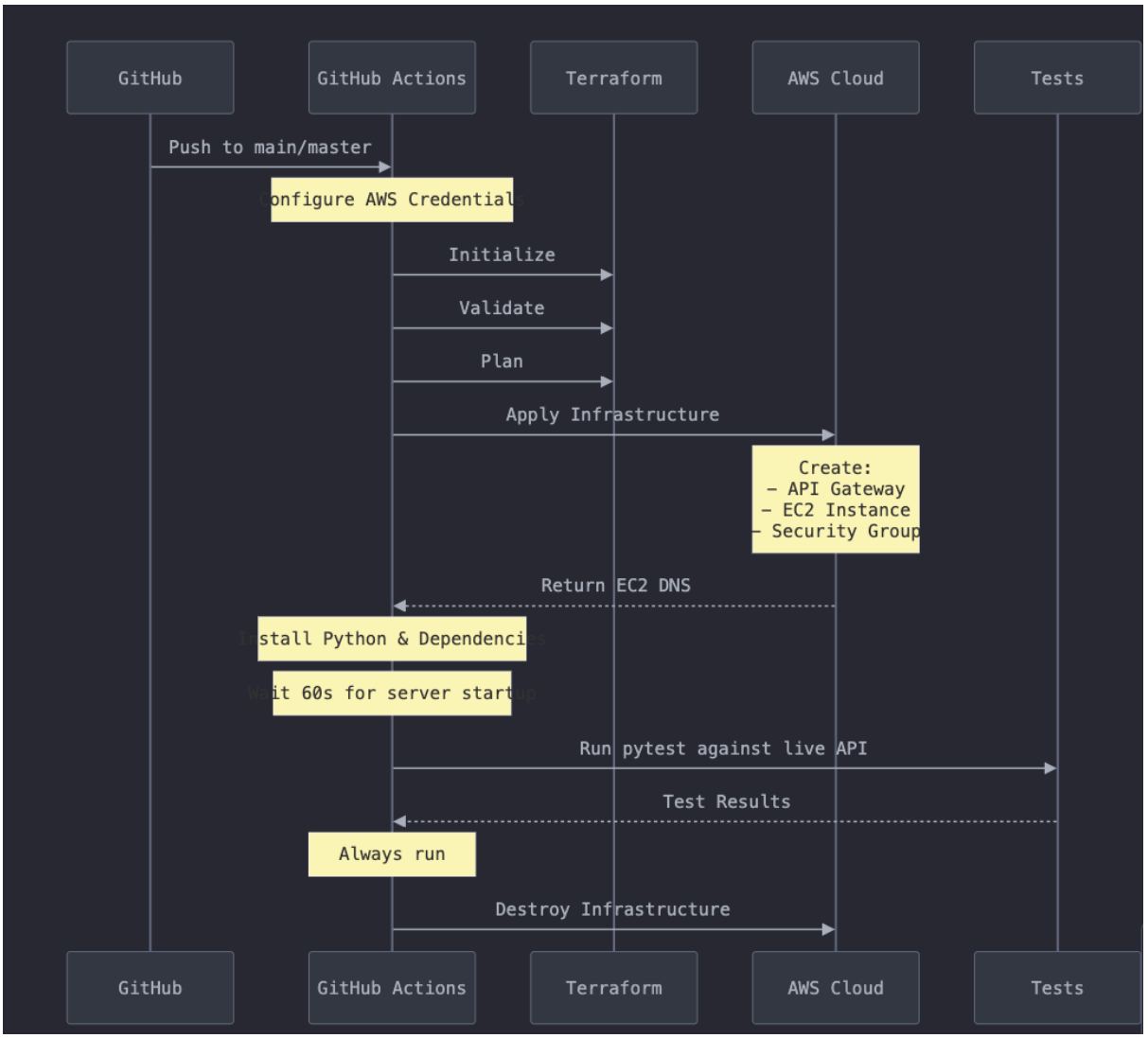

Figure 3

5. Future Work

Several enhancements and expansions could be undertaken to improve **GraphWord-MJ** and broaden its capabilities:

1. **Renaming `datalake/` Folder**
    - **Rationale**: “Datalake” currently holds raw text or dictionary files, but the name may not fully capture its purpose.
    - **Proposal**: Rename it to something like `metadata/` or `raw_input/` to be more descriptive, reflecting that it stores initial data before processing.

2. **Code Modularization and Organization**
    - **Rationale**: Although the codebase is already split into logical folders (`app/`, `graph/`, `word_sources/`), some scripts could be further compartmentalized.
    - **Proposal**: Introduce submodules or packages that isolate concerns, such as a dedicated “crawler” module, separate from the main pipeline and API modules.
3. **Automated Book Crawler**
    - **Rationale**: Currently, Project Gutenberg texts are downloaded on-demand. A scheduled or event-driven crawler would keep the system updated automatically.
    - **Proposal**:
        - Develop a Python-based crawler that runs periodically (e.g., via AWS Lambda or a cron job) to fetch new e-books from Gutenberg (or other sources).
        - Automatically process and store these in the `datamart/` (or an S3 bucket, if fully migrated to the cloud) for ingestion.
4. **AWS Cloud Integration for the Crawler**
    - **Rationale**: Running the crawler in the cloud ensures scalability, reliability, and avoids local resource constraints.
    - **Proposal**:
        - Use AWS Lambda or an EC2 scheduled job to trigger the crawler at regular intervals.
        - Store intermediate data in AWS S3 or Amazon DynamoDB, thus removing the need for local file I/O.
5. **Refining API Gateway Integration**
    - **Rationale**: API Gateway is partially set up, but some endpoints may not fully map to the correct paths. Searching for or accessing certain routes can be unintuitive.

- **Proposal**:
  - Improve the `main.tf` in Terraform to define explicit routes for each API path, ensuring we can easily find and test them.
  - Add AWS stage variables or path mappings for clarity and consistent versioning.

6. **Possible Additional Ideas**

   - **Caching and Performance**:
     - For large graphs, implement caching layers (e.g., Redis or in-memory caching) to reduce repeated path computations.
   - **Cloud File Storage**:
     - Replace local text files with an S3-based pipeline. This avoids versioning conflicts and makes the ingestion step more scalable.
   - **Continuous Integration Enhancements**:
     - Expand GitHub Actions to run advanced tests or static analysis (e.g., `pylint`) and automatically trigger the crawler on merges to `main`.
   - **Multi-Stage Infrastructure**:
     - Introduce separate dev and prod environments via Terraform workspaces or Terraform Cloud.

By addressing these items, **GraphWord-MJ** could evolve into a more scalable, robust, and feature-rich solution—supporting continuous book ingestion, fully cloud-native operations, and enhanced analytics capabilities.