GraphWord Project: Building a Scalable and Distributed Architecture

Andrea Mayor Gómez
Laura Lasso García
University of Las Palmas de Gran Canaria

January 17, 2025

Contents

1 Introduction **3**
1.1 Context . . . 3
1.2 General Objective of the Project . . . 3

2 Theoretical Foundation **4**
2.1 Graphs in Data Science . . . 4
2.1.1 Graph Concept and Representation . . . 4
2.1.2 Applications of Graphs . . . 4
2.2 Techniques and Processing Algorithms . . . 4
2.2.1 Shortest Path . . . 4
2.2.2 All Paths Between Two Nodes . . . 5
2.2.3 Longest Path in the Graph . . . 5
2.2.4 Cluster Identification . . . 5
2.2.5 Nodes with High Connectivity . . . 5
2.2.6 Node Selection by Connectivity Degree . . . 5
2.2.7 Isolated Nodes . . . 5
2.2.8 Directed vs. Undirected Graphs . . . 6

3 Problem Definition **7**

4 Project Objectives **8**
4.1 Specific Objectives . . . 8

5 Tools Used **10**
5.1 Infrastructure Tools . . . 10
5.2 Development Tools . . . 10
5.3 CI/CD Tools . . . 10
5.4 Monitoring and Testing Tools . . . 10

6 System Implementation **11**
6.1 Word Graph Construction . . . 11

6.1.1 Process Flow 11
6.2 API Functionality 12
6.2.1 API Endpoints 12
6.3 Statistics API Service 14
6.4 System Flow Diagram 14
6.4.1 Context Diagram 14
6.4.2 Container Diagram 14

7 Architecture Design 16
7.1 Architecture Description 16
7.1.1 EC2 Instances 16
7.1.2 Load Balancer (ALB) 17
7.1.3 SQS Queues for Synchronization 17
7.1.4 S3 Repositories for File Storage 18
7.2 Diagrams 18
7.2.1 Application and Infrastructure Diagrams 18

8 CI/CD Workflow 22

9 Testing and Validation 25
9.1 Integration Testing 25
9.2 Performance Testing 26

10 Results and Conclusions 27
10.1 Obtained Results 27
10.1.1 Performance Graphs Visualized in Grafana 27
10.1.2 Response Times and Comparisons 27
10.2 Lessons Learned 28
10.3 Challenges and Solutions 28
10.3.1 AWS Permission Issues 28
10.3.2 Data Synchronization in S3 Buckets 28
10.4 Conclusions 29

11 Recommendations and Future Work 30
11.1 Improvements in Search Algorithms 30
11.2 Implementation of a Caching System 30
11.3 Implementation of an Autoscaling Group 31
11.4 Implementation of Prometheus and Grafana 31
11.5 Optimization of Data Management 31
11.6 Future Work on Graph Processing 32

12 Bibliography 33

Chapter 1

Introduction

1.1 Context

Managing complex relationships is one of the main challenges in data science. Graphs are essential data structures for modeling these relationships and are used in applications such as social networks, transportation networks, and recommendation systems. This project aims to implement a distributed system in AWS capable of handling large volumes of data, performing advanced operations on graphs, and automating its deployment through DevOps.

1.2 General Objective of the Project

- Design a distributed architecture in AWS to manage word graphs.
- Implement an API to analyze and query relationships between words.
- Develop a CI/CD pipeline to automate testing, deployment, and monitoring.

Chapter 2

Theoretical Foundation

2.1 Graphs in Data Science

2.1.1 Graph Concept and Representation

A graph $G = (V, E)$ is a structure composed of a set of vertices V (nodes) and a set of edges E (connections between nodes).

2.1.2 Applications of Graphs

- **Social Networks**: Nodes represent users, and edges represent their interactions.
- **Transportation Networks**: Cities as nodes and routes as edges.
- **Energy Networks**: Power stations as nodes and transmission lines as edges.
- **Recommendation Systems**: Users and products as nodes, interactions as edges.

2.2 Techniques and Processing Algorithms

2.2.1 Shortest Path

Dijkstra and A* algorithms were implemented to calculate optimal routes in the graph. These implementations find the shortest path between two words based on their connectivity, optimizing distance calculations. Dijkstra is used for weighted graphs with positive weights, while A* combines heuristics to optimize searches in large graphs.

2.2.2 All Paths Between Two Nodes

A functionality was developed to return all possible paths between two nodes in the graph. This was achieved using depth-first search (DFS) algorithms, which generate an exhaustive set of possible paths while respecting constraints such as maximum depth and a limit on the number of returned paths.

2.2.3 Longest Path in the Graph

To determine the longest acyclic path between two nodes, algorithms based on *Topological Sorting* were implemented, useful for directed acyclic graphs (DAGs). This analysis identifies the maximum-length routes in specific graphs, such as those representing hierarchies or dependencies.

2.2.4 Cluster Identification

The identification of densely connected components was performed using algorithms to detect strongly connected components (SCC). These algorithms, such as Tarjan's or Kosaraju's, identify subsets of nodes where every node is reachable from any other node within the subgraph. This is essential for detecting natural clusters in the data.

2.2.5 Nodes with High Connectivity

A functionality was implemented to identify nodes with high connectivity by calculating both in-degree and out-degree in directed graphs. This allows the analysis of the relative importance of nodes in the graph, identifying central elements within the network.

2.2.6 Node Selection by Connectivity Degree

To select nodes with a specific number of connections, the degree (number of edges connected to a node) was used as a filter criterion. This operation is useful for customized analyses, such as finding nodes with a minimum degree to be considered relevant.

2.2.7 Isolated Nodes

The functionality to return nodes without connections uses a graph search to identify nodes with a degree equal to zero. This analysis helps detect elements disconnected from the system, which may represent incomplete data or errors in the model.

2.2.8 Directed vs. Undirected Graphs

The use of `nx.DiGraph()` from the NetworkX library was justified to represent directed relationships in the graph. This enables modeling relationships with specific directions, such as dependencies or information flows, necessary for operations like shortest paths, identifying nodes with high connectivity, and detecting clusters. However, in analyses where direction is irrelevant, the graph can be converted to undirected using `nx.Graph()`.

Chapter 3

Problem Definition

- Construction of a word graph where each word connects to another if they differ by one letter.
- Scalability of the system as the length of words increases.

Chapter 4

Project Objectives

4.1 Specific Objectives

- Build dictionaries of filtered words from normalized texts.
- Create a graph where nodes represent words and edges represent connections based on one-letter differences.
- Implement an API to provide the following functionalities:
  - `/shortest-path`: Calculates the shortest path between two words using algorithms like Dijkstra or A*.
    - **Parameters:** Source node and destination node.
  - `/all-paths`: Returns all possible paths between two words, showing different connection alternatives.
    - **Parameters:** Source node, destination node, and optionally parameters like maximum depth and maximum number of paths.
  - `/maximum-distance`: Calculates the maximum possible distance between two connected nodes in the graph.
    - **Parameters:** None, or specify nodes to limit the calculation.
  - `/clusters`: Identifies densely connected subgraphs (clusters) within the graph.
    - **Parameters:** None.
  - `/high-connectivity-nodes`: Identifies nodes with a high degree of connectivity.
    - **Parameters:** Minimum connection threshold (optional).

- `/nodes-by-degree`: Selects nodes with a specific degree of connectivity.
  - **Parameters:** Desired degree of connectivity.
- `/isolated-nodes`: Returns nodes with no connections (isolated nodes).
  - **Parameters:** None.
- `/reset-graph`: Resets the graph to its original state loaded from stored data.
  - **Parameters:** None.

Chapter 5

Tools Used

5.1 Infrastructure Tools

- **AWS:**
  - EC2: Execution of the API and related tasks.
  - S3: File storage.
  - SQS: Synchronization of messages between instances.
  - ALB: Load balancing.

5.2 Development Tools

- Python: Implementation of the API logic.
- NetworkX: Library for graph operations.
- Flask: Web framework for the API.

5.3 CI/CD Tools

- GitHub Actions: Automation of testing and deployment.

5.4 Monitoring and Testing Tools

- Locust: Performance testing.
- Prometheus: Metrics usage.

Chapter 6

System Implementation

6.1 Word Graph Construction

The system begins with the downloading of books from the **Gutenberg** project. These books are stored and processed in an orderly manner to create the word graph.

6.1.1 Process Flow

1. **Crawler (Download and Normalization):** The instance responsible for downloading accesses the books via the internet, downloads them, and normalizes the content into text files, removing stopwords and unnecessary characters.

2. **Storage in Datalake:** The normalized files are stored in the S3 bucket named `datalake` in a folder corresponding to the current date.

3. **Dictionary Creation:** A second EC2 instance monitors the `datalake` S3 bucket, receives messages through an SQS queue, and processes the files to generate a dictionary of words with their respective frequencies.

4. **Storage in Datamart Dictionary:** The generated dictionary is saved in the `datamart_dictionary` S3 bucket.

5. **Graph Generation:** Another instance receives messages from the SQS queue of `datamart_dictionary` and builds a graph where the nodes are words, and the edges represent one-letter changes between words.

6. **Storage in Datamart Graph:** The graph is stored in the `datamart_graph` S3 bucket for use by the API instances.

7. **Event Processing:** The penultimate instance receives messages from the SQS queue monitoring the events folder in the datalake bucket and processes all events registered via API requests. Then, the final instance, using messages from the SQS queue of the `datamart_stats` bucket, updates the event-related information and serves the API.

6.2 API Functionality

The four instances running the Flask API allow the management and querying of the word graph through the following endpoints:

6.2.1 API Endpoints

-
  ```
  /shortest-path?origin=<node>&destination=<node>
  ```

  Returns the shortest path between two words, calculated using Dijkstra's algorithm. **Example Usage:**

  ```
  GET /shortest-path?origin=word1&destination=word2
  ```

-
  ```
  /all-paths?origin=<node>&destination=<node>
  &max_depth=<number>&max_paths=<number>
  ```

  Returns all possible paths between two words in the graph, allowing optional limits for maximum length and number of returned paths. By default, the maximum length is 5, and up to 50 paths are returned. **Example Usage:**

  ```
  GET
     /all-paths?origin=word1&destination=word2&max_depth=6
  &max_paths=30
  ```

-
  ```
  /maximum-distance
  ```

  Calculates and returns the maximum distance between nodes in the graph, indicating the pair of words farthest apart in terms of connections. **Example Usage:**

  ```
  GET /maximum-distance
  ```

-
  ```
  /clusters
  ```

Identifies graph clusters, showing strongly connected components. **Example Usage:**

```
GET /clusters
```

-
  ```
  /high-connectivity-nodes?min=<number>
  ```

  Lists nodes with at least `min` connections. **Example Usage:**

  ```
  GET /high-connectivity-nodes?min=2
  ```

-
  ```
  /nodes-by-degree?degree=<number>
  ```

  Returns nodes with a specific degree, i.e., those with exactly the specified number of connections. **Example Usage:**

  ```
  GET /nodes-by-degree?degree=3
  ```

-
  ```
  /isolated-nodes
  ```

  Lists isolated nodes, i.e., nodes with no connections to others in the graph. **Example Usage:**

  ```
  GET /isolated-nodes
  ```

-
  ```
  /health
  ```

  Checks if the API is functioning properly and responds with a 200 code if everything is fine. **Example Usage:**

  ```
  GET /health
  ```

-
  ```
  /filter-graph?min=<length>&max=<length>
  ```

  Filters the graph by word length and displays the filtered nodes and edges. **Example Usage:**

  ```
  GET /filter-graph?min=3&max=6
  ```

-
  ```
  /reset-graph
  ```

  Restores the original graph by reloading the file from the `datamart_graph` S3 bucket. **Example Usage:**

  ```
  GET /reset-graph
  ```

6.3 Statistics API Service

The last EC2 instance runs the statistics API:

- This instance subscribes to the SQS queue for `datamart_stats` and downloads the most recent files.
- It allows querying aggregated data and performance metrics through endpoints such as:
  - `/stats-endpoint-count`: Number of requests per endpoint.
  - `/average-response-time`: Average response time.
  - `/top-errors`: Top recorded errors.

6.4 System Flow Diagram

The presented diagrams illustrate the interaction between components:

6.4.1 Context Diagram

It shows the main interaction between the user, the `GraphWord` system, and external sources (internet for book downloads).

6.4.2 Container Diagram

It exposes the main containers of the system:

- **Crawler:** Downloads and normalizes text files.
- **Dictionary Builder:** Processes files and generates dictionaries.
- **Graph Builder:** Creates the graph structure and stores it.
- **Graph Query API:** Handles graph-related queries.
- **Stat Builder:** Processes events and generates statistics.
- **Stat Query API:** Allows querying system statistics.

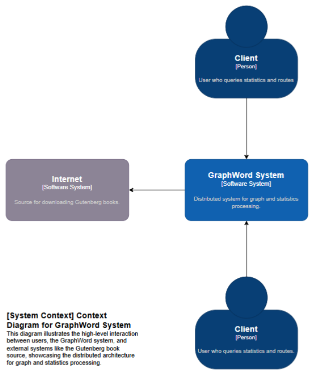
Client
[Person]
User who queries statistics and routes
Internet
[Software System]
Source for downloading Gutenberg books.
GraphWord System
[Software System]
Distributed system for graph and statistics processing.
[System Context] Context Diagram for GraphWord System
This diagram illustrates the high-level interaction between users, the GraphWord system, and external systems like the Gutenberg book source, showcasing the distributed architecture for graph and statistics processing.
Client
[Person]
User who queries statistics and routes.

Figure 6.1: Context diagram.

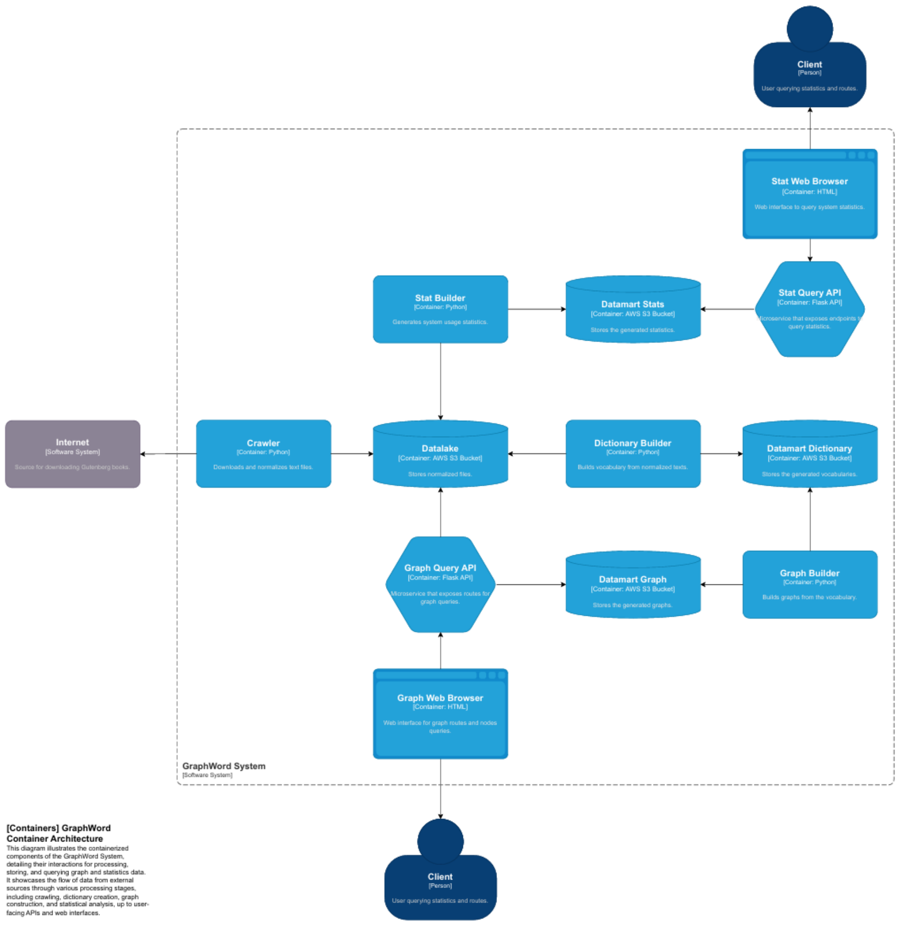
Client
[Person]
User querying statistics and routes.
Stat Web Browser
[Container: HTML]
Web interface to query system statistics.
Stat Builder
[Container: Python]
Generates system usage statistics.
Datamart Stats
[Container: AWS S3 Bucket]
Stores the generated statistics.
Stat Query API
[Container: Flask API]
Microservice that exposes endpoints to query statistics.
Internet
[Software System]
Source for downloading Gutenberg books.
Crawler
[Container: Python]
Downloads and normalizes text files.
Datalake
[Container: AWS S3 Bucket]
Stores normalized files.
Dictionary Builder
[Container: Python]
Builds vocabulary from normalized texts.
Datamart Dictionary
[Container: AWS S3 Bucket]
Stores the generated vocabularies.
Graph Query API
[Container: Flask API]
Microservice that exposes routes for graph queries.
Datamart Graph
[Container: AWS S3 Bucket]
Stores the generated graphs.
Graph Builder
[Container: Python]
Builds graphs from the vocabulary.
Graph Web Browser
[Container: HTML]
Web interface for graph routes and nodes queries.
GraphWord System
[Software System]
[Containers] GraphWord Container Architecture
This diagram illustrates the containerized components of the GraphWord System, detailing their interactions for processing, storing, and querying graph and statistics data. It showcases the flow of data from external sources through various processing stages, including crawling, dictionary creation, graph construction, and statistical analysis, up to user-facing APIs and web interfaces.
Client
[Person]
User querying statistics and routes.

Figure 6.2: Container diagram.

Chapter 7

Architecture Design

7.1 Architecture Description

The project is designed with a distributed and scalable architecture that leverages AWS services to implement an efficient data processing flow. The architecture is based on a set of EC2 instances that work in synchronization via SQS queues and store processed data in S3 buckets. Below is the description of each main component:

7.1.1 EC2 Instances

The system consists of 9 EC2 instances with different roles and responsibilities:

- **Instance 1 (Book Download):** This instance downloads books from external sources (e.g., Project Gutenberg) and saves them in the S3 bucket named `datalake-graph-ulpgc`, in a folder corresponding to the current date.
- **Instance 2 (Dictionary Processing):** This instance reads books stored in the `datalake`, processes the content to generate dictionaries of meaningful words, and saves the dictionaries in the `datamart-dictionary-ulpgc` bucket.
- **Instance 3 (Graph Generation):** This instance processes the dictionaries saved in the `datamart-dictionary-ulpgc` bucket and creates a graph with nodes representing words. The graph is stored in the `datamart-graph-ulpgc` bucket.

- **Instances 4, 5, 6, and 7 (API Execution):** These four instances run the Flask API service, enabling queries on the graph, such as calculating shortest paths, finding all paths between two nodes, calculating maximum distance, identifying clusters, and detecting isolated nodes. Traffic to these instances is distributed using an **Application Load Balancer (ALB)** to ensure high availability and balanced request distribution.

- **Instance 8 (Event Processing):** This instance downloads events stored in the `datalake` related to API requests and processes the data to generate statistics. The statistics are saved in the `datamart-stats-ulpgc1` bucket.

- **Instance 9 (Statistics API):** This final instance runs a Flask API service for statistics, allowing queries on metrics and processed event data stored in `datamart-stats-ulpgc`.

7.1.2 Load Balancer (ALB)

The **Application Load Balancer (ALB)** is responsible for distributing incoming requests to the four instances running the main API. The ALB ensures load is distributed evenly, preventing overload on a single instance and guaranteeing high availability.

7.1.3 SQS Queues for Synchronization

SQS queues are used to coordinate data flow between the various instances and S3 buckets:

- **SQS Queue for `datalake` (Books):** Monitors the folder corresponding to the current date in the `datalake-graph-ulpgc` bucket and notifies the dictionary processing instance when new books are added.

- **SQS Queue for `datalake` (Events):** Monitors the `events` folder and notifies the event processing instance when new API requests are logged.

- **SQS Queue for `datamart-dictionary-ulpgc`:** Notifies the graph generation instance when new dictionaries are added to the bucket.

- **SQS Queue for `datamart-graph-ulpgc`:** Notifies the API instances when the graph in the bucket is updated.

- **SQS Queue for `datamart-stats-ulpgc`:** Notifies the statistics API instance when new event reports are generated.

7.1.4 S3 Repositories for File Storage

Different S3 buckets are used to store files at various stages of the processing flow:

- `datalake-graph-ulpgc`: Stores downloaded books and events logged from API requests.
- `datamart-dictionary-ulpgc`: Stores dictionaries generated by the processing instance.
- `datamart-graph-ulpgc`: Stores graphs created from the dictionaries.
- `datamart-stats-ulpgc`: Stores event reports and generated statistics.
- `graph-code-ulpgc`: Stores the .py files that each instance needs to execute.

7.2 Diagrams

7.2.1 Application and Infrastructure Diagrams

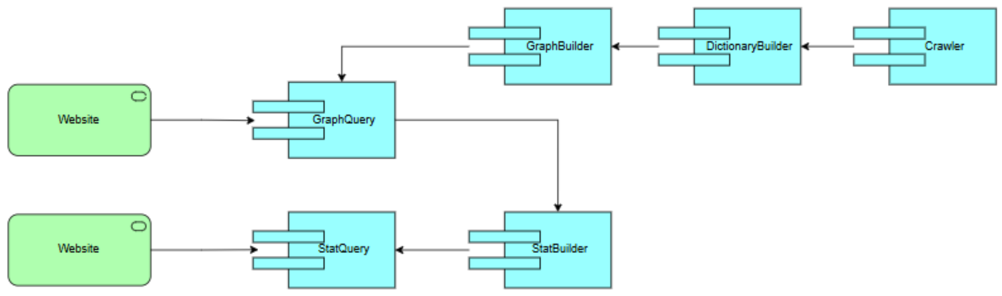

Figure 7.1: Application diagram.

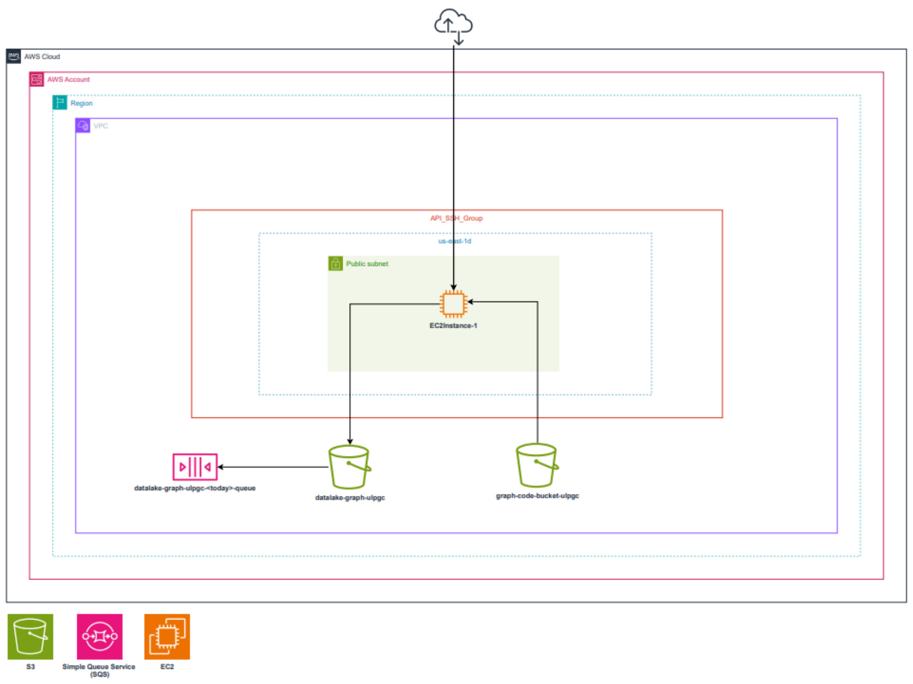

Figure 7.2: Crawler infrastructure.

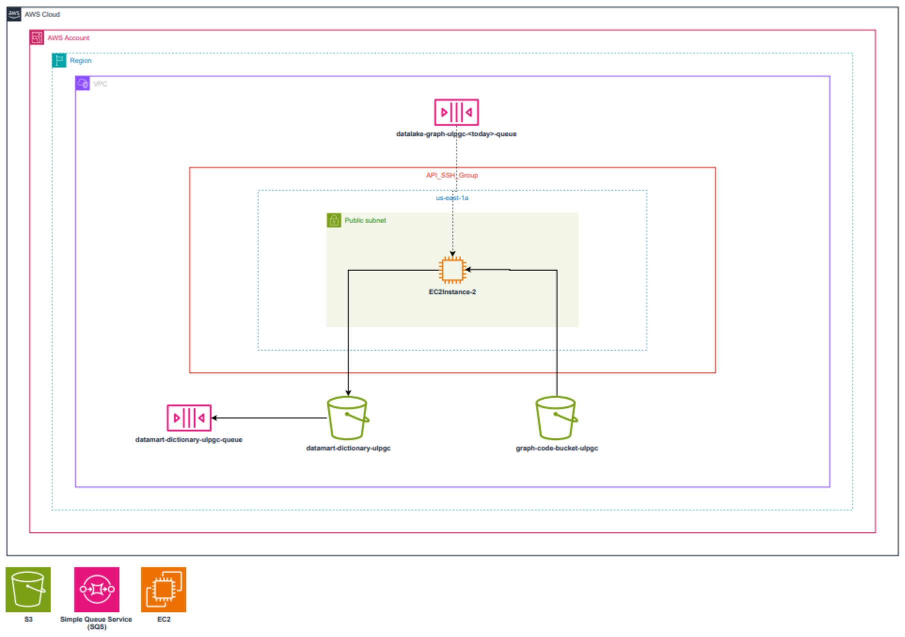
AWS Cloud
AWS Account
Region
VPC
datalake-graph-ulpgc-<today>-queue
API_SSH_Group
us-east-1a
Public subnet
EC2Instance-2
datamart-dictionary-ulpgc-queue
datamart-dictionary-ulpgc
graph-code-bucket-ulpgc
S3
Simple Queue Service (SQS)
EC2

Figure 7.3: DictionaryBuilder infrastructure.

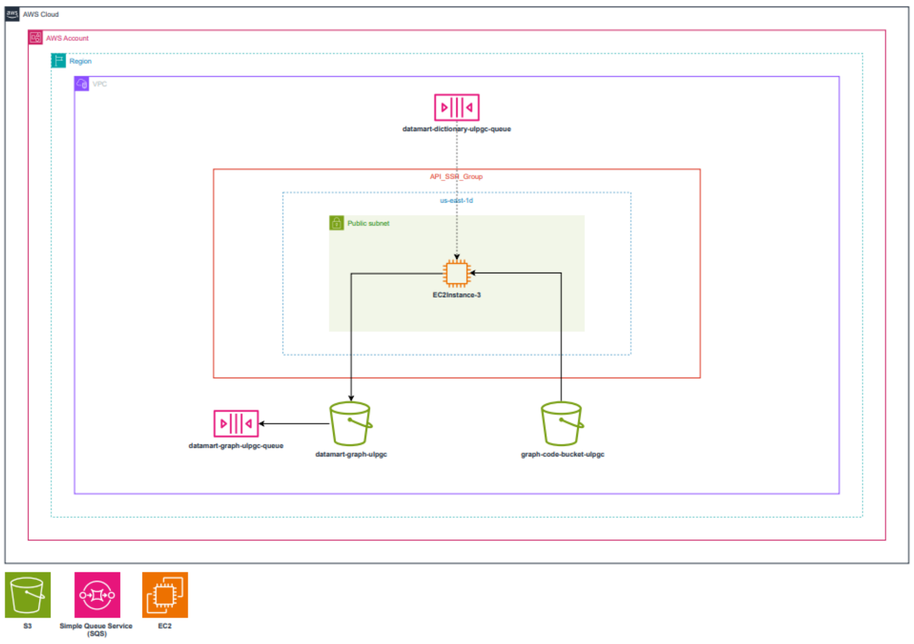

Figure 7.4: GraphBuilder infrastructure.

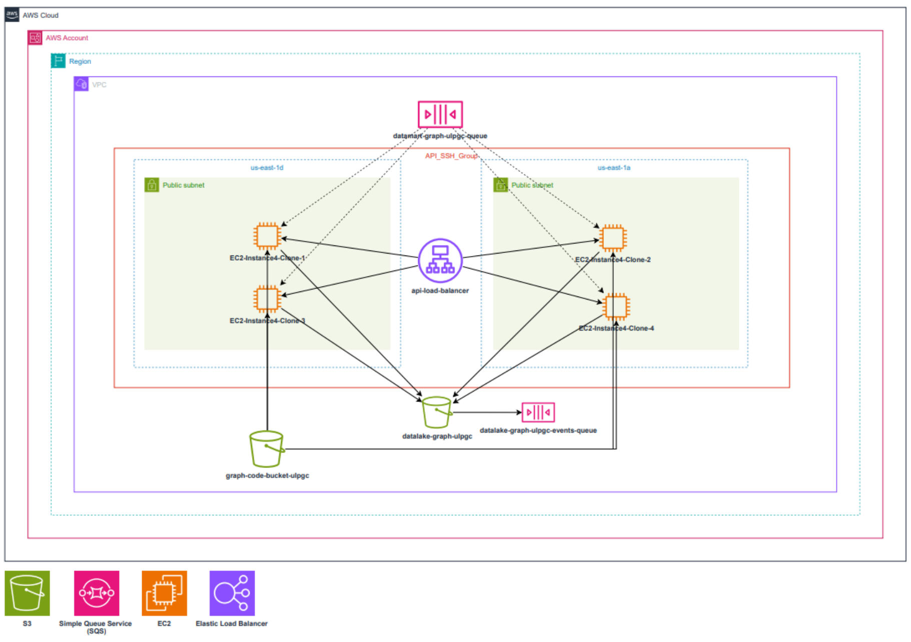

Figure 7.5: GraphQuery infrastructure.

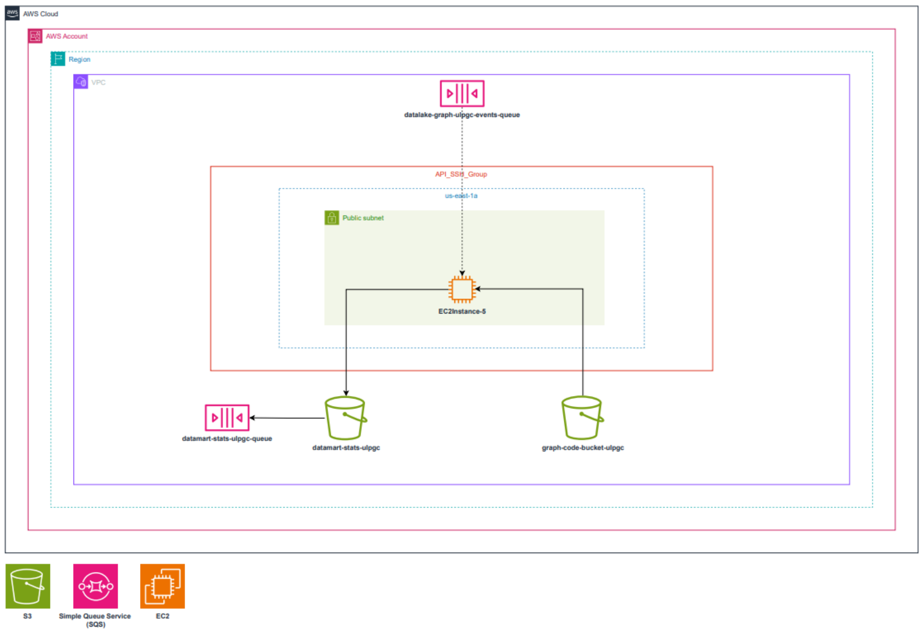

Figure 7.6: StatBuilder infrastructure.

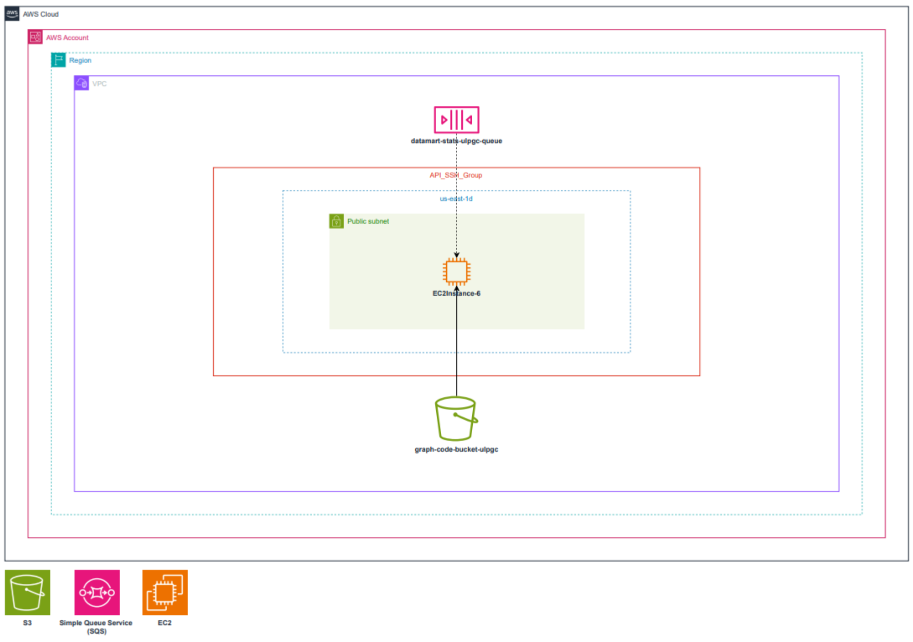

Figure 7.7: StatQuery infrastructure.

Chapter 8

CI/CD Workflow

The `.yml` file describes the workflow of a CI/CD pipeline designed to manage the infrastructure and ensure code quality through automated testing and deployments. The main steps executed by the file are:

- **Dependency Installation:** Necessary tools such as Terraform and AWS CLI are installed, along with specific libraries for testing and monitoring, such as `locust`, `prometheus_client`, and `jq`.
- **Infrastructure Configuration:** Using Terraform, the pipeline initializes, plans, and applies the required changes to manage AWS resources. This includes creating S3 buckets, configuring API Gateway, and defining the necessary IAM roles for the system.
- **Integration and Performance Testing:** Integration tests are executed to validate the correct functioning of the API. Additionally, performance tests carried out with Locust evaluate the system's capacity under simulated load, with metrics exposed in formats supported by Prometheus.
- **Error Notification:** If any failures occur during critical steps, such as applying Terraform changes or running tests, the pipeline halts, and an error notification is sent.
- **Automated Deployment on AWS:** Once the tests are validated, the pipeline ensures automated deployment on AWS by executing `terraform apply`. It verifies that the changes are correct before proceeding to the next step.
- **Infrastructure Lifecycle Management:** After completing the tests and monitoring, the pipeline removes provisional resources using `terraform destroy` to avoid unnecessary additional costs.

- **Release Creation:** After completing the main workflow, a release is created in GitHub with detailed deployment information.
- **Merge to `master` Branch:** The changes implemented in the `develop` branch are automatically merged into `master`, ensuring the environments are synchronized. During this process, Terraform is re-executed to ensure consistency.

This workflow is illustrated in the following Mermaid diagram.

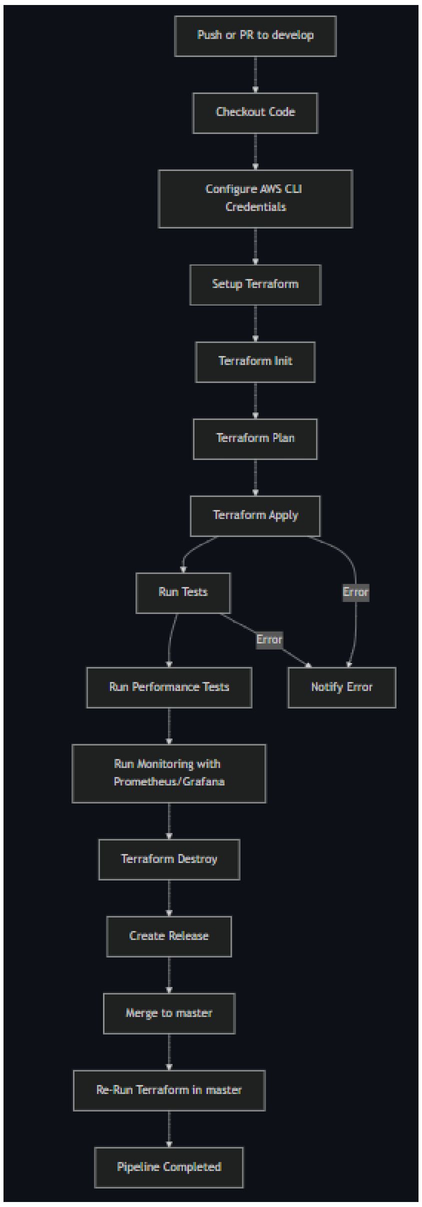

Figure 8.1: Mermaid Diagram.

Chapter 9

Testing and Validation

9.1 Integration Testing

Integration tests were performed using the `test_endpoints.py` file, which validates the interaction between the different components and endpoints of the deployed API. These tests covered the following key aspects:

- Validation of the root endpoint (`/`) to ensure the service is active and responds correctly.
- Testing shortest paths (`/shortest-path`), ensuring proper responses for both successful cases and expected errors (e.g., non-existent nodes).
- Testing all paths (`/all-paths`) with optional parameters such as `max_depth` and `max_paths`.
- Verification of maximum distances (`/maximum-distance`), connectivity (`/clusters`, `/high-connectivity-nodes`), and node degrees (`/nodes-by-degree`).
- Analysis of isolated nodes (`/isolated-nodes`) and graph filtering (`/filter-graph`).
- Restoring the graph to its initial state (`/reset-graph`) and verifying the overall system status (`/health`).

These tests ensured that the API functioned correctly in an integration environment, interacting seamlessly with AWS components and system logic.

9.2 Performance Testing

Performance tests were conducted using `Locust` with the `locustfile.py` script, designed to measure the system's capacity under load and its response to multiple concurrent users. These tests included:

- Configuration of custom metrics with `prometheus_client` to capture response times (`REQUEST_TIME`) and the total number of requests made (`TOTAL_REQUESTS`).
- Execution of tests for critical endpoints such as:
  - `/health` to verify availability.
  - `/shortest-path` and `/all-paths` to measure performance in path calculations.
  - `/maximum-distance`, `/clusters`, and `/nodes-by-degree` to validate complex graph computations.
- Configuration of concurrent users with a wait time between requests (`between(30, 40)`) to simulate realistic traffic.

Chapter 10

Results and Conclusions

10.1 Obtained Results

Throughout the project, we successfully implemented a distributed and scalable system for graph management and analysis based on AWS. The main results obtained are as follows:

10.1.1 Performance Graphs Visualized in Grafana

Using Locust along with Prometheus metrics, we captured and visualized key metrics related to API and EC2 instance performance:

- **Response Times:** Measured the average, minimum, and maximum response times for each endpoint, identifying the most used ones and those with higher processing times.
- **Resource Usage:** Monitored CPU, memory, and network usage on EC2 instances to optimize their configuration.
- **Performance Under Load:** Evaluated how the API performed under high concurrent request volumes through **Locust** testing.

10.1.2 Response Times and Comparisons

Performance tests with graphs of different sizes and configurations revealed:

- Response times for `/shortest-path` and `/clusters` are fast for small graphs but increase significantly as the graph size grows.
- The `/all-paths` endpoint exhibited the highest response times due to the high computational cost of calculating all paths between two nodes.

- Query optimization with parameters such as `max_depth` and `max_paths` reduced execution time by approximately 50% for complex queries.

10.2 Lessons Learned

The project provided valuable lessons about the implementation and management of distributed systems on AWS:

- **Permission Management:** During the setup of resources like S3, SQS, and EC2, we faced issues related to limited permissions on the student AWS account, preventing the implementation of an Autoscaling Group and delaying initial configuration.
- **Component Synchronization:** Ensuring that the different EC2 instances and SQS queues worked in sync was challenging. This required constant monitoring and the implementation of robust notification messages.
- **Query Optimization:** Processing large graphs with algorithms like Dijkstra and A* posed scalability issues. We learned to optimize queries by introducing limits and data preprocessing.

10.3 Challenges and Solutions

10.3.1 AWS Permission Issues

The lack of advanced permissions on the student account was a significant limitation:

- We could not implement an Autoscaling Group to dynamically manage the number of EC2 instances.
- **Solution:** We manually configured the number of instances to simulate a scalable environment.

10.3.2 Data Synchronization in S3 Buckets

Initially, we faced issues synchronizing data between different buckets due to delayed messages in SQS:

- **Solution:** We implemented wait times (retries) and periodic validations to ensure data availability before proceeding with processing.

10.4 Conclusions

The project achieved the proposed objectives, demonstrating the feasibility of a distributed system for graph management and statistics:

- A functional and scalable API was built, enabling complex graph analysis.
- A CI/CD pipeline was implemented in GitHub Actions, automating deployment and testing on AWS.
- Monitoring tools and performance tests were shown to be essential for ensuring system stability and efficiency.

Chapter 11

Recommendations and Future Work

11.1 Improvements in Search Algorithms

The system currently uses algorithms such as Dijkstra and A* to calculate shortest paths and optimal routes in the graph. While these algorithms are efficient, they could be further optimized to handle large-scale graphs. Possible improvements include:

- **Advanced Heuristics:** Explore smarter heuristics in the A* algorithm to reduce the number of nodes visited.
- **Graph Preprocessing:** Apply preprocessing techniques to store intermediate results and speed up repetitive queries.
- **Implementation of Parallel Algorithms:** Divide the search into subprocesses to leverage multi-core architectures.
- **GPU Utilization:** Incorporate parallel processing on GPUs for extremely large graphs.

11.2 Implementation of a Caching System

The current system computes queries in real-time, which can be costly for repetitive queries or large graphs. A caching system could:

- Store results of frequent queries, such as routes between popular nodes or preprocessed clusters.
- Significantly reduce response times by reusing previous calculations.

11.3 Implementation of an Autoscaling Group

The project currently operates with a fixed number of EC2 instances. To improve scalability and optimize resource usage:

- Configure an **Autoscaling Group** to dynamically adjust the number of instances based on system load.
- Define scalability metrics based on the number of API requests or the CPU/memory usage of the instances.
- Although not implemented due to restrictions in the student AWS account, this improvement would be essential for a production environment.

11.4 Implementation of Prometheus and Grafana

While the project includes performance testing with Locust, full integration with Prometheus and Grafana for advanced monitoring has not been fully implemented. Possible improvements include:

- Configure **Prometheus** to collect system usage metrics such as response times and errors per endpoint.
- Create dashboards in **Grafana** to visualize key metrics and identify performance bottlenecks.
- Generate real-time alerts for issues such as increased response times or frequent errors.

11.5 Optimization of Data Management

The use of S3 buckets to store data and synchronize instances can be improved:

- Optimize bucket structure to handle large volumes of data.
- Implement object versioning in S3 to maintain a history of changes.
- Use AWS `S3 Event Notifications` more efficiently to reduce latency between dependent processes.

11.6 Future Work on Graph Processing

- Implement graph reduction techniques to minimize the number of nodes and edges processed without affecting results.
- Incorporate support for dynamic graphs that change in real-time as words are added or removed from the system.
- Extend API functionality to allow for more complex analyses, such as community detection or identification of critical bridges in the graph.

Chapter 12

Bibliography

- Official AWS documentation.
- Libraries: Flask, NetworkX, Locust.