ULPGC
Universidad de
Las Palmas de
Gran Canaria

Project Design and Architecture

Tecnologías de Servicios para Ciencia de Datos

CreamsCode

Carlos Mendoza Eggers & Joel Clemente López Cabrera

Contents

1 Introduction

In the last months, we have been immersed in the development of a robust and scalable architecture designed to handle complex data workflows. This project revolves around creating seamless integration between various services, each playing a pivotal role in data collection, storage, processing, and analysis. The overarching goal is to provide an efficient, graph-based API that enables users to query and manipulate complex relationships stored in a graph database.

The architecture is composed of several interconnected components. These include the Graph API, which serves as the primary interface for users; the Neo4j database, which manages graph storage and processing; the Datamart, responsible for aggregating and processing data; and the Datalake, which ensures long-term storage of both raw and processed data. Additionally, the Collector service was developed to ingest raw data from external sources and push it into the Datalake. Together, these components form the backbone of the system, designed to support highly interactive and data-driven workflows.

One of the most significant challenges of the project was deploying the entire system infrastructure on AWS. Building a cloud-native solution required extensive planning and implementation to ensure efficient communication between services, reliability, and scalability. From provisioning Virtual Private Clouds (VPCs) to configuring services, the focus was on creating a secure and high-performing environment for the application. Moreover, Terraform was used extensively to automate infrastructure provisioning, allowing reproducibility and minimizing manual configuration errors.

By leveraging a modular and containerized architecture and integrating technologies such as Spring Boot, Neo4j and Hazelcast, the system ensures that each service operates independently while seamlessly integrating into the larger ecosystem. These efforts culminated in a foundational system capable of meeting current requirements.

This document outlines the progress achieved, focusing on the technical milestones, the deployment strategies employed on AWS, and the next steps necessary to fully realize our vision for the ideal system. Through this effort, we have demonstrated a commitment to clean architecture, maintainability, and scalability, ensuring a solid foundation for long-term growth.

2 Original Idea

To start the development of the project, we decided to test the logic of the services locally. The final code differs from the code proposed in the initial prototype because we had to adapt to the problems we encountered along the way.

The main differences are:

- Use of Docker for service servers: For testing it locally, the easiest way to deploy the services was using Docker. At the moment we moved to AWS, we decided not to add an additional abstraction layer for deploying the services, as we considered it unnecessary. Instead, we opted to deploy the services natively in the intances.

- Changes in the programming language for some applications: Initially, we planned to write all the code in Python. However, some services performed very slowly computationally. We believe this was due to the nature of the language, which is not as optimized for these types of operations. Specifically, we moved the entire implementation of Hazelcast, Neo4j, and the API to Java for convenience. We observed a significant performance improvement after switching from Python to Java.

The deployment of the local architecture can be found in the following repository: **Link to GitHub**(note that some parts are not updated as we created separate repositories for each service).

3 Services Overview

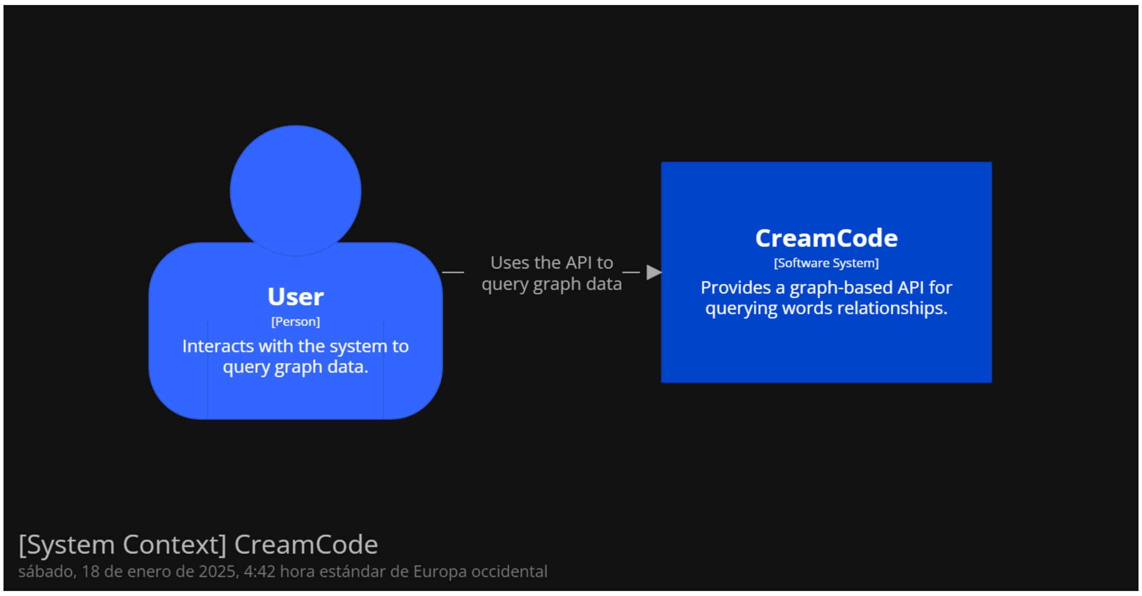
User
[Person]
Interacts with the system to query graph data.
Uses the API to query graph data
CreamCode
[Software System]
Provides a graph-based API for querying words relationships.
[System Context] CreamCode
sábado, 18 de enero de 2025, 4:42 hora estándar de Europa occidental

Figure 1: System Context Diagram

The system context overview highlights the interaction between the User and the CreamCode system, which provides a graph-based API for querying word relationships. The User interacts with the Graph API, the main interface of the system, to query graph data stored in the Neo4j graph database. Thanks to this setup, the user can interact and perform queries such as getting the shortest path between nodes or getting the nodes with the highest degree.

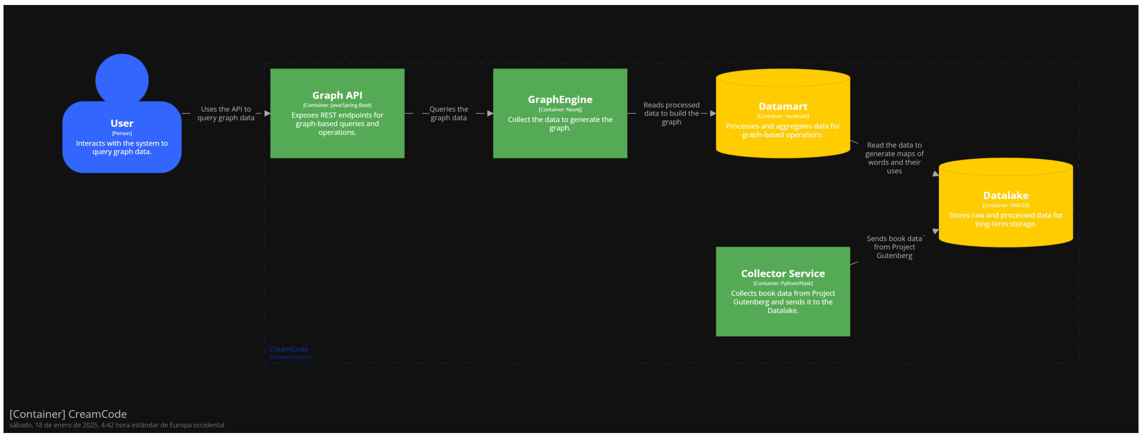
User
[Person]
Interacts with the system to query graph data.
Uses the API to query graph data
Graph API
[Container: Java/Spring Boot]
Exposes REST endpoints for graph-based queries and operations.
Queries the graph data
GraphEngine
[Container: Neo4j]
Collect the data to generate the graph.
Reads processed data to build the graph
Datamart
[Container: Hazelcast]
Processes and aggregates data for graph-based operations.
Read the data to generate maps of words and their uses
Datalake
[Container: AWS S3]
Stores raw and processed data for long-term storage.
Sends book data from Project Gutenberg
Collector Service
[Container: Python/Flask]
Collects book data from Project Gutenberg and sends it to the Datalake.
CreamCode
[Software System]
[Container] CreamCode
sábado, 18 de enero de 2025, 4:42 hora estándar de Europa occidental

Figure 2: Container Diagram

Within the CreamCode system, several services collaborate to manage data flow, processing, and storage:

- **Graph API:** This API, built using Java and Spring Boot, exposes REST endpoints for users to interact with the graph database. It acts as the main access point for querying graph data.
- **Graph Engine:** Powered by Neo4j, the graph engine processes and collects data from the datamart to construct the graph. It is the backbone of the system, enabling graph-based operations by integrating with other components.
- **Datamart:** Using Hazelcast, the datamart processes and aggregates data for efficient graph operations. It serves as a dynamic in-memory layer between the Graph Engine and the Datalake, ensuring quick data access and transformations.
- **Datalake:** The datalake provides long-term storage for raw and processed data. It plays a crucial role in maintaining scalability and ensuring that historical data remains accessible for future processing needs.
- **Collector Service:** This service gathers raw data from Project Gutenberg that will be processed afterwards. It ensures the continuous ingestion of fresh data for further processing.

Together, these services form a cohesive system that supports the user's ability to explore complex word relationships through an intuitive API. The system context diagram illustrates this interaction, while the detailed architecture shows how the services are integrated and designed to create a modular and efficient platform.

4 Services Architecture

4.1 Collector

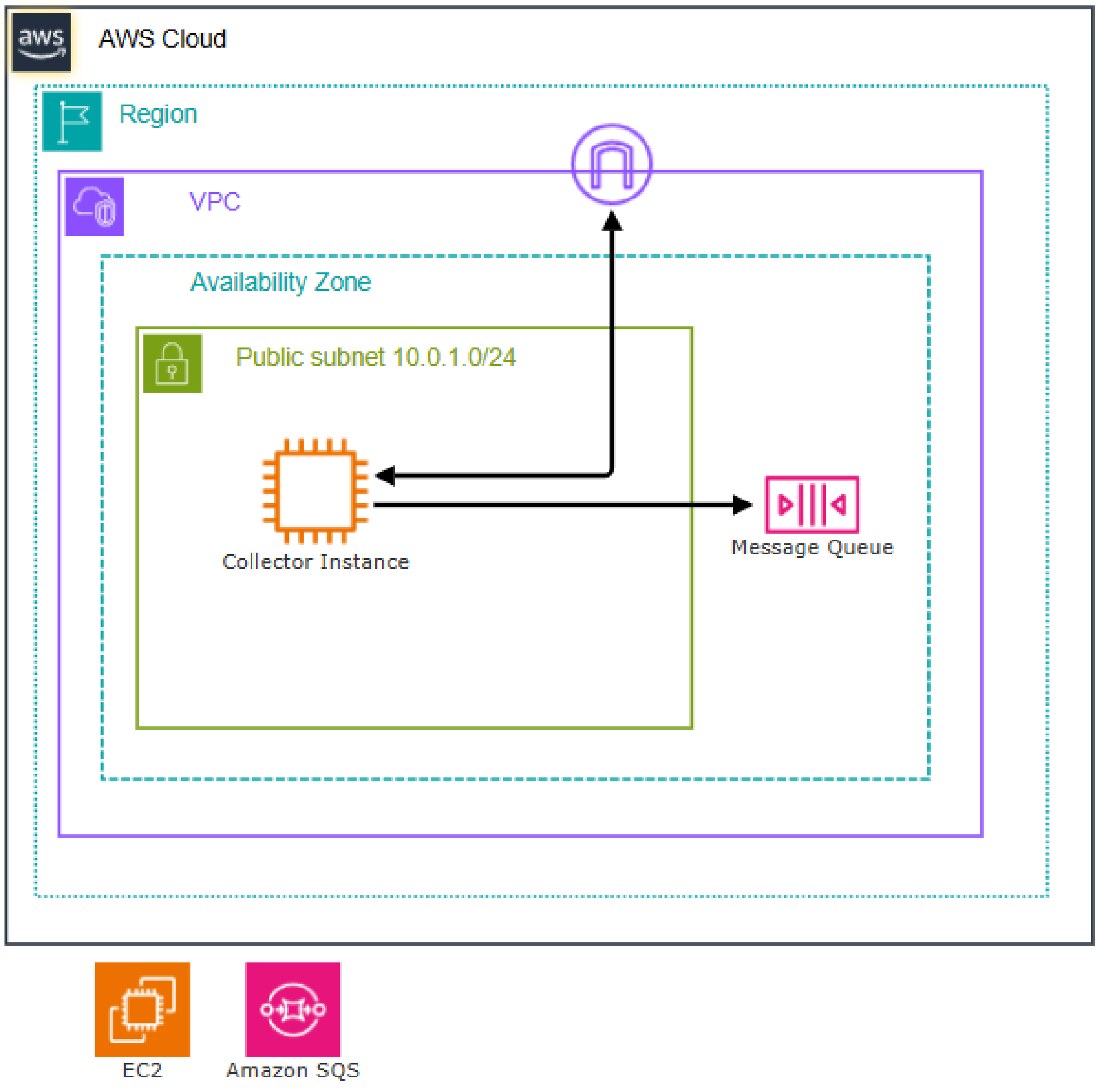

Figure 3: Collector

Function

The Collector Architecture (Figure 3) is designed as a core component of the data ingestion process within the AWS cloud environment. It is responsible for gathering raw data from Project Gutenberg, and forwarding it downstream for further processing. The architecture ensures reliability and seamless integration between ingestion and storage layers, leveraging key AWS services like EC2 and Amazon SQS. Operating within a secure Virtual Private Cloud (VPC) and a specific Availability Zone (us-east1), it fetches, processes, and queues data for subsequent transformation and analysis.

Responsibilities

1. **Scraper:** The Collector Instance (EC2) retrieves raw data (books) from Project Gutenberg, compresses the information into a message, and then passes it to the SQS queue, ensuring consistent and reliable input to the system.

2. **Networking and Connectivity:** The Internet Gateway enables the Collector Instance to connect securely to external networks for data collection.

3. **Data Buffering:** The Message Queue (Amazon SQS) temporarily stores ingested data processed by the Collector Instance, decoupling the ingestion process from downstream services to improve fault tolerance.

4.2 Datalake

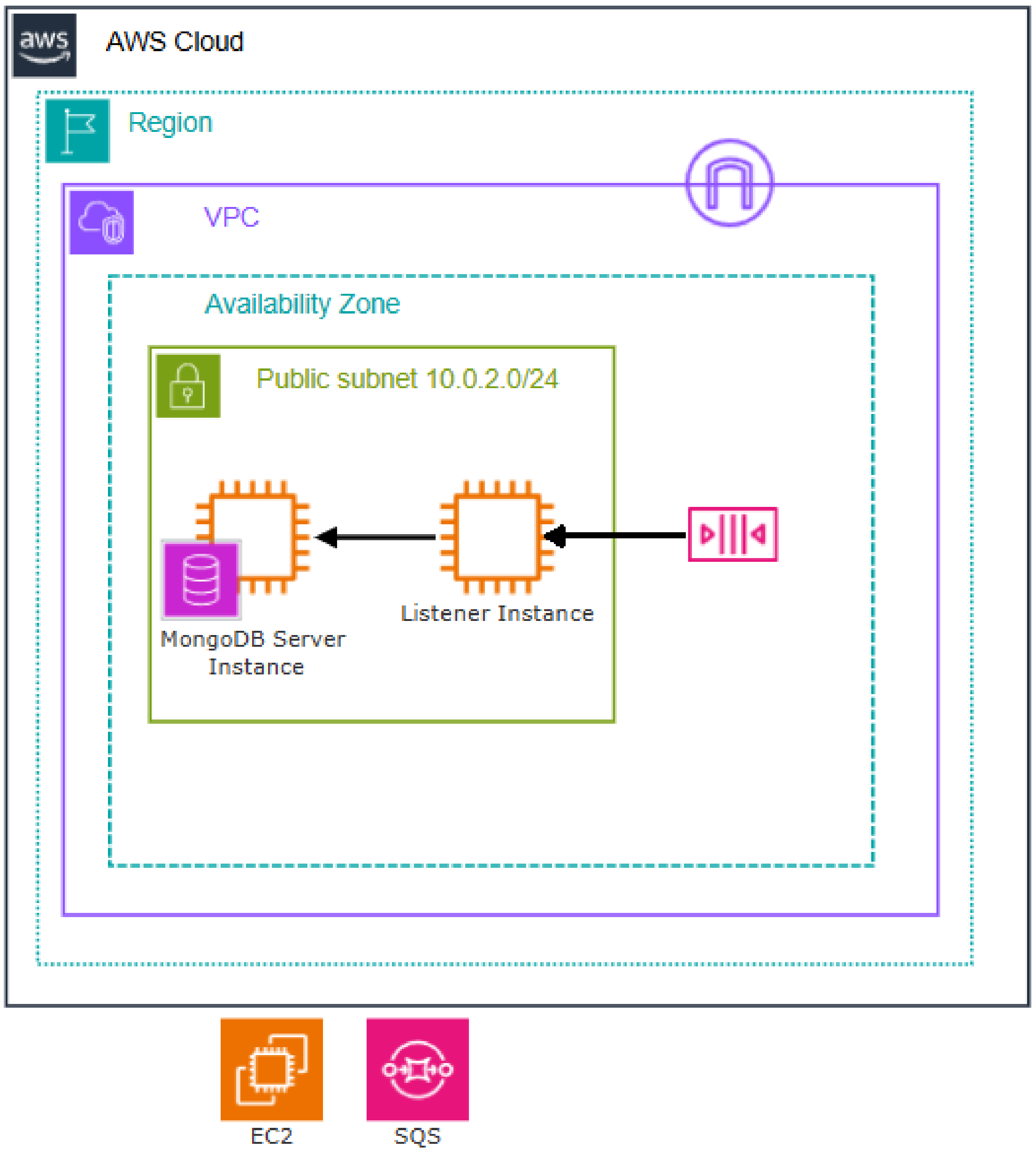

Figure 4: Datalake

Function

The Datalake architecture (Figure 4) is responsible for storing raw and processed data for long-term use while ensuring accessibility for downstream systems. It acts as a centralized repository for all collected and transformed data. The architecture resides within a specific AWS Cloud Region, ensuring compliance with regional regulations and proximity to related services. It is encapsulated within a VPC (Virtual Private Cloud) to provide isolation and security for its components. Additionally, it utilizes a Public Subnet (10.0.2.0/24) to enable network access for the EC2 instances and controlled communication with external systems and AWS services.

Responsibilities

1. **Amazon SQS:** The Simple Queue Service acts as a messaging layer to decouple components of the data pipeline. This corresponds to the queue where the Collector has sent the compressed messages.

2. **Listener Instance:** This EC2 instance listens for messages in the SQS (Simple Queue Service), consumes these messages, decompresses them, and processes the data to be stored in the MongoDB database. It ensures the data ingestion pipeline runs continuously and efficiently.

3. **MongoDB Server Instance:** This EC2 instance is configured to host a MongoDB database, where processed and structured data is securely stored. It provides a robust and scalable NoSQL database that facilitates subsequent operations and analysis.

4.3 Datamart

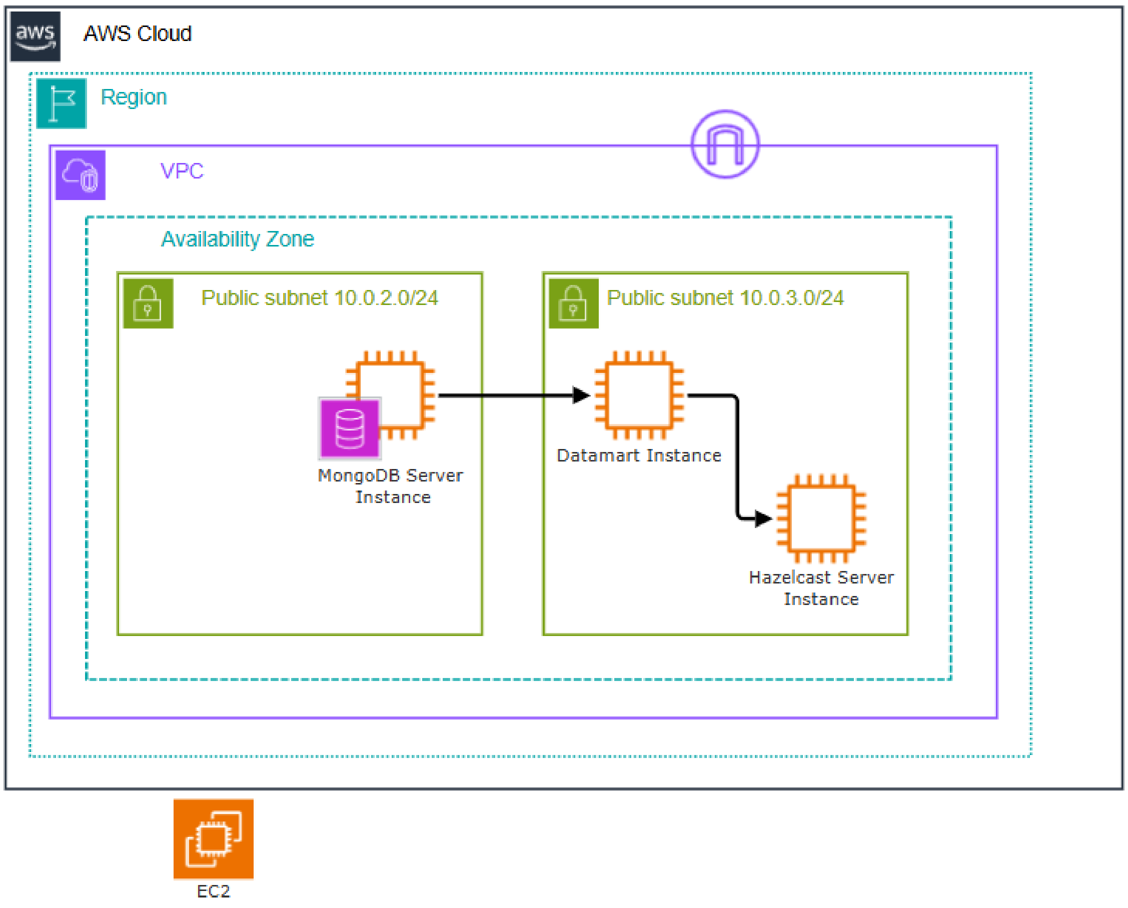

Figure 5: Datamart

Function

The Datamart (Figure 5) serves as the intermediate layer in the data pipeline, aggregating and transforming data stored in the Datalake for use in graph-related operations. It processes and optimizes data for queries and analytics performed by downstream systems such as the Graph API. The architecture is encapsulated within a VPC (Virtual Private Cloud), ensuring secure and isolated communication between its components. It spans across two public subnets (10.0.2.0/24 and 10.0.3.0/24), providing network access for instances like MongoDB Server, Datamart Instance, and Hazelcast Server. This setup ensures efficient data processing and in-memory caching capabilities to enhance performance.

We are aware that this service could have been omitted, and that the data ingestion could have been performed directly from MongoDB to the Neo4j database. However, we were determined to revisit Hazelcast, as we attempted to use it last year for our Big Data project but were unable to implement it successfully. Additionally, considering potential scalability, we saw Hazelcast as a highly feasible solution to reduce computation times effectively.

Responsibilities

1. **Datamart Instance:** This EC2 instance is responsible for aggregating and transforming data retrieved from the MongoDB Server. It processes the data to meet the requirements of graph-related operations, like word usages.

2. **Hazelcast Server Instance:** This EC2 instance acts as an in-memory caching service. It enables fast access to frequently used data and optimizes performance for real-time queries.

4.4 GraphEngine

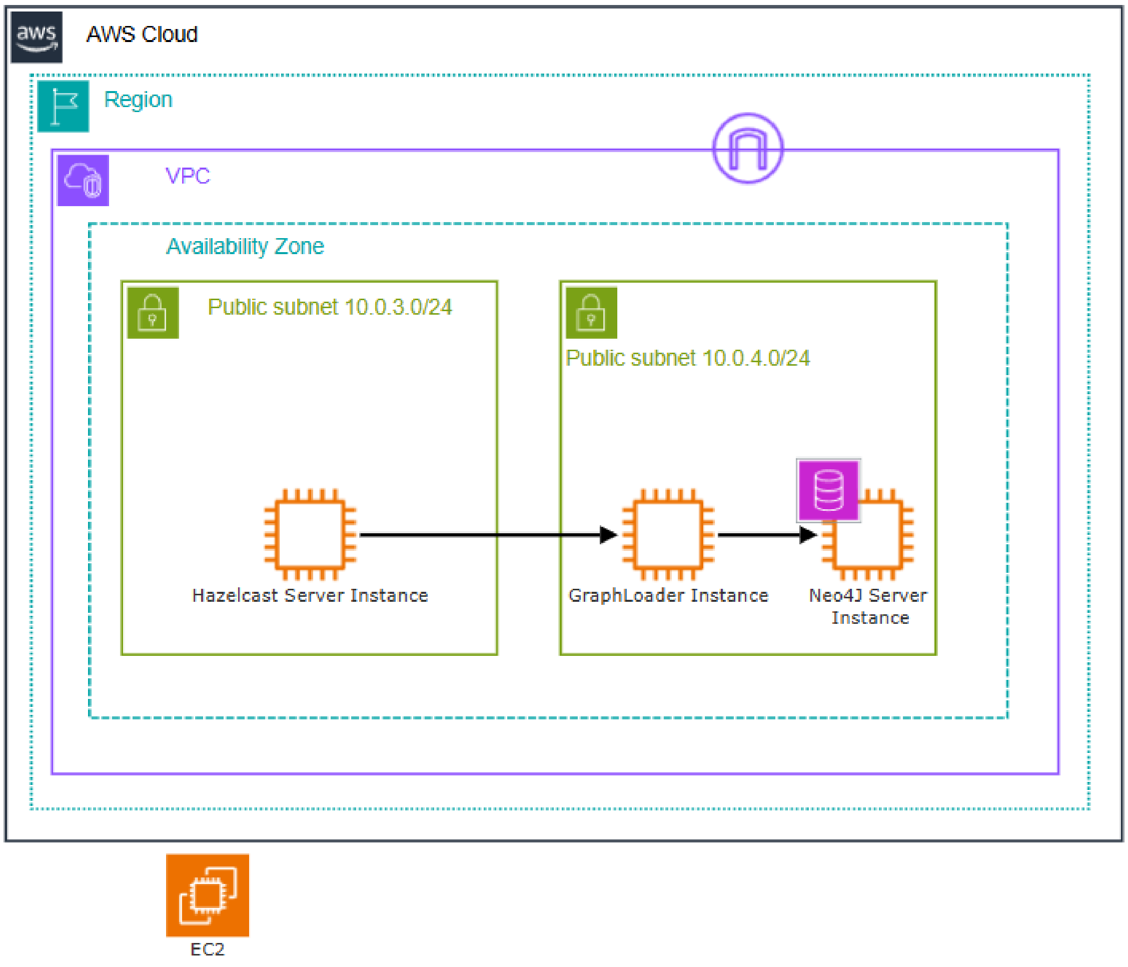

Figure 6: GraphEngine

Function

The Graph Engine (Figure 6) is the central processing unit of the architecture, tasked with generating and maintaining the graph data for efficient querying and analysis. It orchestrates communication between in-memory caching (Hazelcast), graph loading, and the graph database (Neo4j). Deployed within a VPC (Virtual Private Cloud) for security and isolation, the architecture spans across two public subnets: 10.0.3.0/24 for in-memory caching and 10.0.4.0/24 for the graph loading and database instances. This separation ensures modularity and optimized performance for graph operations.

Responsibilities

1. **GraphLoader Instance:** This EC2 instance acts as the intermediary for data transformation and insertion into the Neo4j database. It retrieves processed data from Hazelcast, translates it into a format suitable for graph storage, and inserts the information into the Neo4j server.

2. **Neo4j Server Instance:** This instance hosts the Neo4j database, the core repository for graph data. It enables complex relationship queries, leveraging the graph data generated by the pipeline.

4.5 API

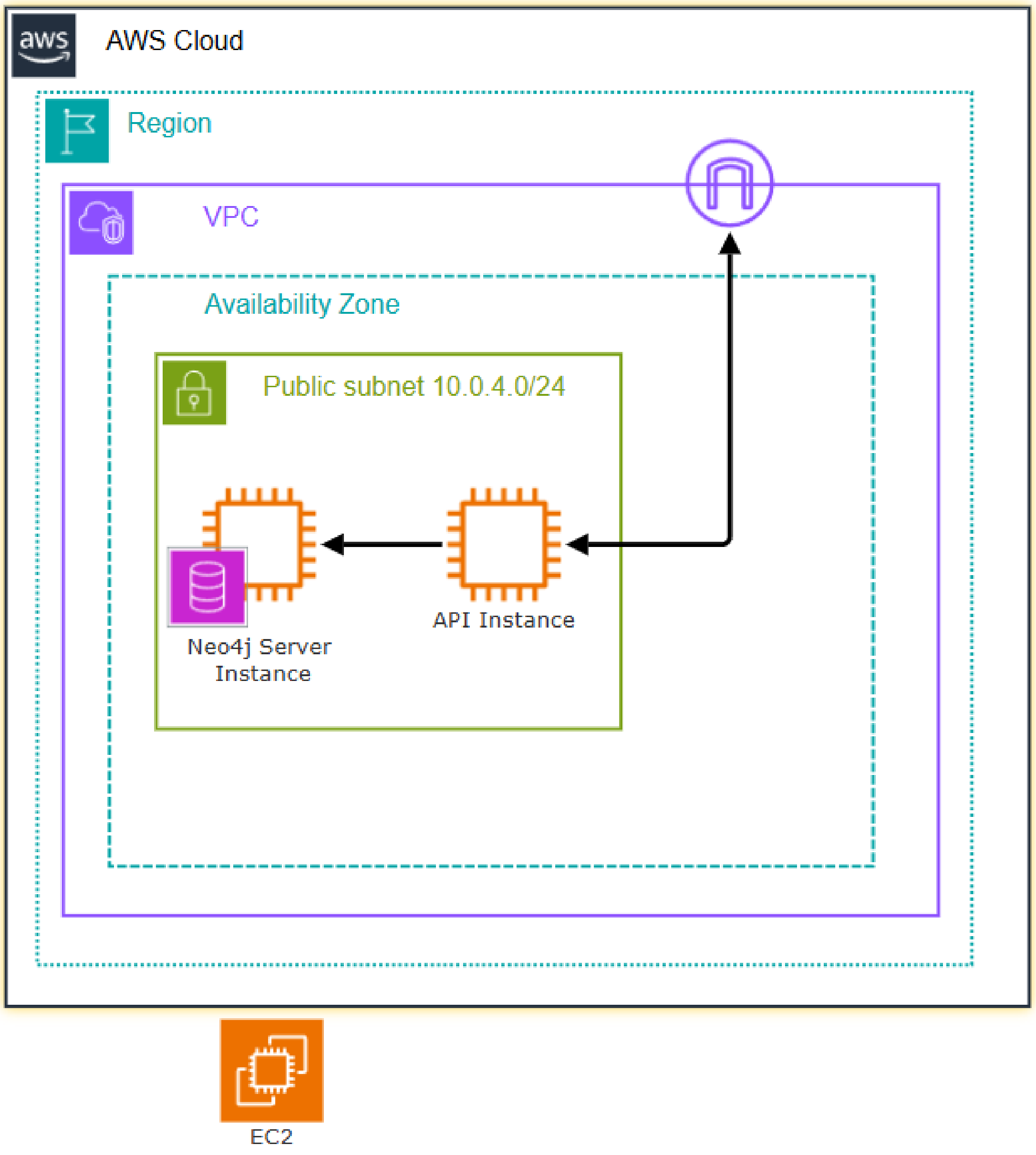

Figure 7: API

Function

The API (Figure 7) is the primary interface for users to interact with graph data. This API facilitates querying relationships stored in the Neo4j Server Instance. Deployed within the 10.0.4.0/24 public subnet in an AWS Virtual Private Cloud (VPC), it ensures secure and controlled access to the graph database. The API is designed to handle high-performance queries, enabling users to explore complex data relationships efficiently.

Responsibilities

1. **API Instance:** This EC2 instance is responsible for making requests directly to the instance hosting the Neo4j database.

   API Endpoint Descriptions:

   - /shortest-path{GET /api/shortest-path?source=word1&target=word2}:
     Purpose: Finds the shortest path between two nodes in the graph based on their word attributes.
   - /all-paths{GET /api/all-paths?source=word1&target=word2}:
     Purpose: Retrieves all paths between two nodes within a given depth while ensuring no nodes repeat within a single path.
   - /isolated-nodes{GET /api/isolated-nodes}:
     Purpose: Identifies nodes in the graph that do not have any relationships with other nodes.
   - /longest-path{GET /api/longest-path?source=word1&target=word2}:
     Purpose: Finds the longest path between two nodes without repeating intermediate nodes.
   - /clusters{GET /api/clusters}:
     Purpose: Discovers clusters of nodes that are connected within the graph.
   - /high-degree-nodes{GET /api/high-degree-nodes}:
     Purpose: Finds nodes with a high degree of connectivity.
   - /nodes-by-degree/{degree}{GET /api/nodes-by-degree/{degree}}:
     Purpose: Retrieves all nodes with a specific degree of connectivity.

4.6 Project Infrastructure

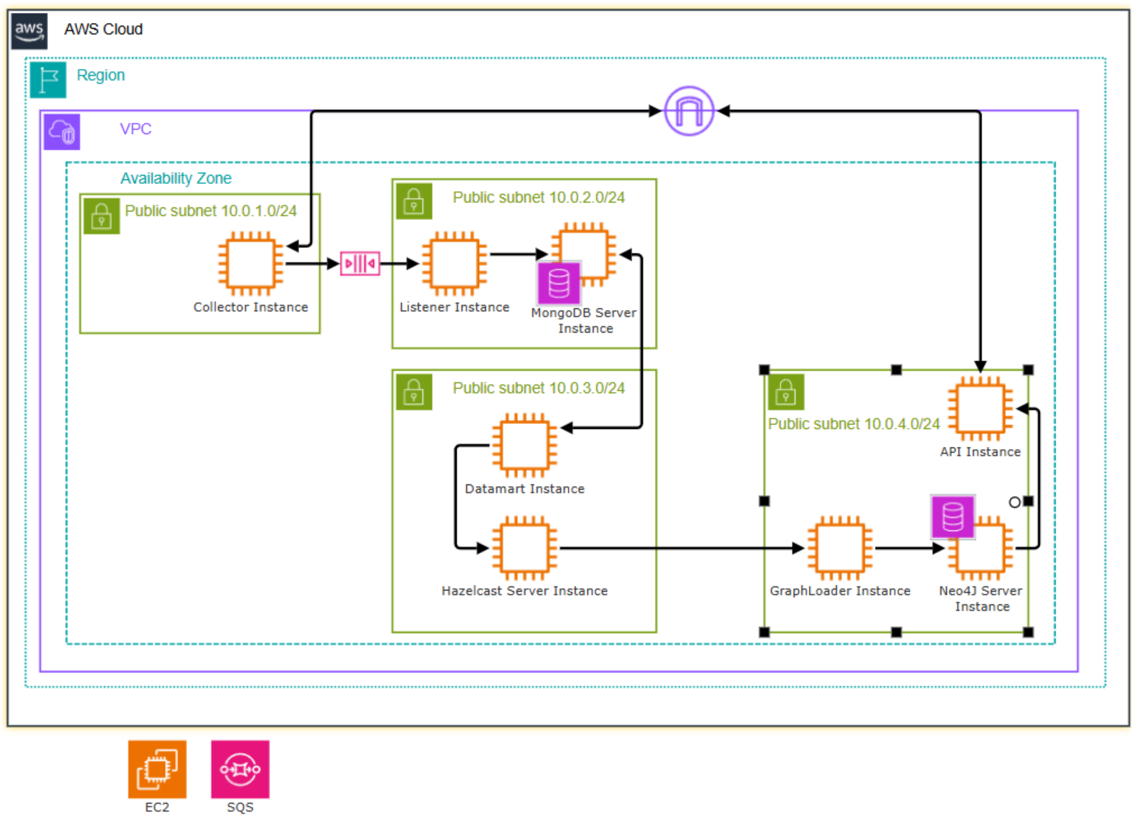

Figure 8: General Infrastructure Overview

Finally, Figure 8 shows the global breakdown of all the integrated and coupled services.

5 Future Ideas and Final Thoughts

To conclude this report, we would like to justify some design decisions and also discuss potential improvements or future ideas.

Due to the limitations of the Sandbox environment, we had to make a very important decision when we started deploying the project on AWS. Initially, we envisioned a queueing system to insert messages into the database. However, the Sandboxes that allowed the use of Amazon SQS did not support the use of Lambdas, so we ultimately had to implement most of the services within EC2 instances. Some of these services could have been deployed on a Lambda, eliminating the need to use EC2.

We also attempted to deploy a sharded cluster of MongoDB, but it turned out to be complex, and we lost a significant amount of time on it. As a result, we ultimately discarded this possibility.

Another aspect to highlight for future improvements is that some queries from the API are performed slowly.

The current system represents a significant achievement in building a modular and distributed architecture for managing graph-based data workflows. However, scalability remains a key area for improvement. As the project grows, ensuring that the system can efficiently handle larger datasets and increased user demands will be critical to its long-term success. We were unable to fully explore this feature since the execution performance is not poor. We believe that this system is easily scalable for most of its services using AWS, and that the greatest complexity would lie in configuring the scalability of the databases used in the project.

This journey highlights the importance of building systems with scalability in mind and the value of continuous improvement to align with emerging needs and technologies. With a clear focus on scalability and performance optimization, the project is well-positioned to achieve its full potential.