Architecture system

WORDNET-NEXUS APPLICATION

January 23, 2025

Susana Suarez Mendoza
University of Las Palmas de Gran Canaria
School of Computer Engineering

Mara Pareja del Pino
University of Las Palmas de Gran Canaria
School of Computer Engineering

1 Introduction

This document presents the architecture of Wordnet-Nexus, a system designed to process Project Gutenberg books and generate word frequency graphs. The architecture is defined using the C4 model, complemented with dynamic, infrastructure, deployment diagrams, and detailed descriptions of each module. The C4 model provides a clear hierarchical view of the software architecture through four levels: context, containers, components, and code. These levels make it possible to structure and visualize the interactions of the system with its actors and the relationship between its different parts.

In addition to the structure described by the C4 model, additional representations will be included detailing the dynamic interactions between components, the arrangement of resources in the underlying infrastructure and the deployment strategy in a distributed cloud environment. The goal is to provide a comprehensive understanding of the system, from its conceptual design to its technical implementation, highlighting how each module contributes to the overall flow of data and operations.

This report starts with an overview of the system context and its main actors, followed by a more detailed analysis of the containers, components and code. Diagrams and explanations are also presented to illustrate how the modules interact and how the system operates in an efficient and scalable manner.

2 C4 Model

This document defines the architecture of the **Wordnet-Nexus** system following the C4 model, which allows one to structure and visualize software architectures in a clear and understandable way through four hierarchical levels: context, containers, components, and code. Wordnet-Nexus is a system designed to process books from Project Gutenberg and generate word frequency graphs, facilitating the analysis of texts programmatically through an API and a web interface. This report details each level of the architecture, starting with the general context of the system (Level 1) and progressively going deeper into its technical design and specific functionality. Each level is complemented by visual diagrams illustrating the interaction between actors, systems, and components.

2.1 Level 1 - System context diagram

Level 1 of the C4 model describes the overall context of the system and how it interacts with its actors and other external systems. The following details the context of the Wordnet Nexus system, a system designed to process books and generate word frequency graphs.

2.1.1 System Description

The Wordnet-Nexus system is primarily intended to process books from the Project Gutenberg collection and generate word frequency graphs. This system provides a web interface and an API to allow users to interact in a simple and efficient way.

2.1.2 External Actors and Systems

- **User:** Person who interacts with the system through the Web interface and the API, sending requests to view charts.
- **Wordnet-Nexus:** Central system that performs the main operations, including downloading books and processing data to create frequency graphs.
- **Project Gutenberg:** External system that provides the books in digital format. Wordnet-Nexus downloads these books for analysis and processing.

2.1.3 System Context Diagram

The following is a diagram representing the overall context of the Wordnet-Nexus system that highlights the interactions between the system, its actors, and external systems.

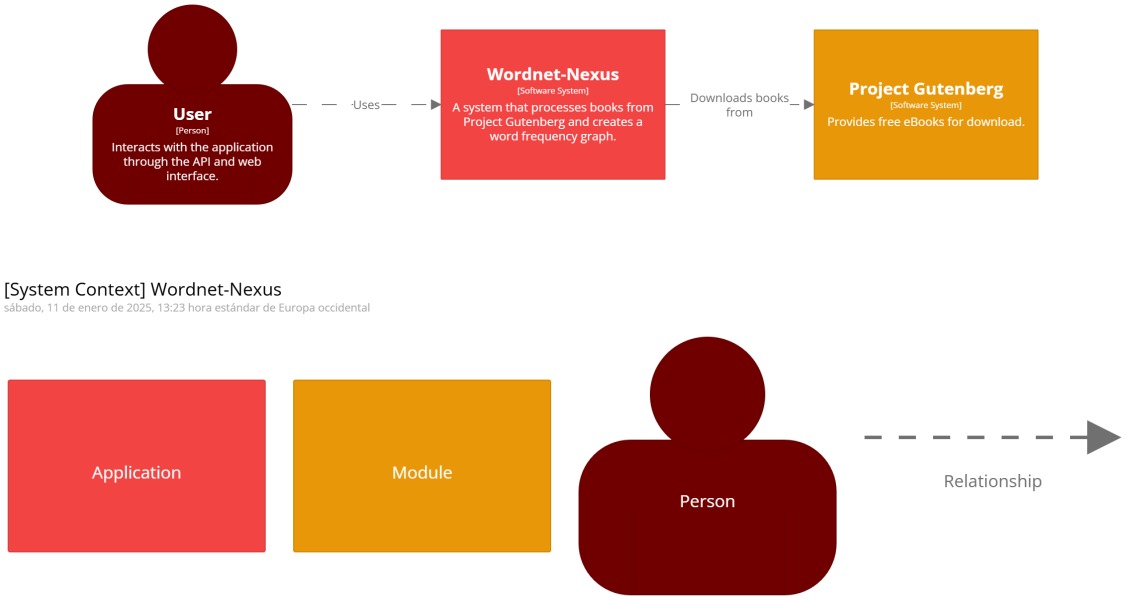

Figure 1. System context diagram.

2.2 Level 2 - Container diagram

Level 2 of the C4 model focuses on the internal structure of the system, dividing its architecture into containers that represent applications, services, databases, and other entities that are part of the system.

2.2.1 System Description

The Wordnet-Nexus system is composed of several containers that collaborate to download books from Project Gutenberg, process their content and generate word frequency graphs. Each container plays a specific role within the data flow and uses different technologies to carry out its duties.

2.2.2 System Containers

- **API Service (Python and JavaScript):** This container handles API requests and serves web pages. It acts as the main entry point for the user, managing both queries and the user interface.
- **Crawler (Java):** This module is responsible for downloading books from Project Gutenberg and uploading them to a bucket in AWS S3.
- **DatalakeBuilder (Python):** Processes the books stored in AWS S3, extracts word frequencies, and stores the resulting data in a MongoDB database.
- **GraphDrawer (Python):** Generates word frequency graphs using the data stored in MongoDB and saves the results as graphs in Neo4j.

2.2.3 Databases and External Services

- **AWS S3:** Data storage used to save the books downloaded by the Crawler.
- **MongoDB:** Database that stores processed information about word frequencies.
- **Neo4j:** Graph database where the generated frequency graphs are stored.
- **Project Gutenberg:** External system that provides digital books for analysis.

2.2.4 Key Interactions

- The **User** interacts with the system through the **API Service** container, sending requests and receiving results.
- The **Crawler** downloads books from **Project Gutenberg** and stores them in **AWS S3**.
- The **DatalakeBuilder** reads the books from **AWS S3**, extracts word frequencies, and saves the data in **MongoDB**.

- The **GraphDrawer** uses data from **MongoDB** to generate graphs and stores them in **Neo4j**.
- The **API** directly queries **Neo4j** to retrieve and display the generated graphs.

2.2.5 Container Diagram

The following is the container diagram for the **Wordnet-Nexus** system, illustrating the interaction between the various modules, databases, and external services.

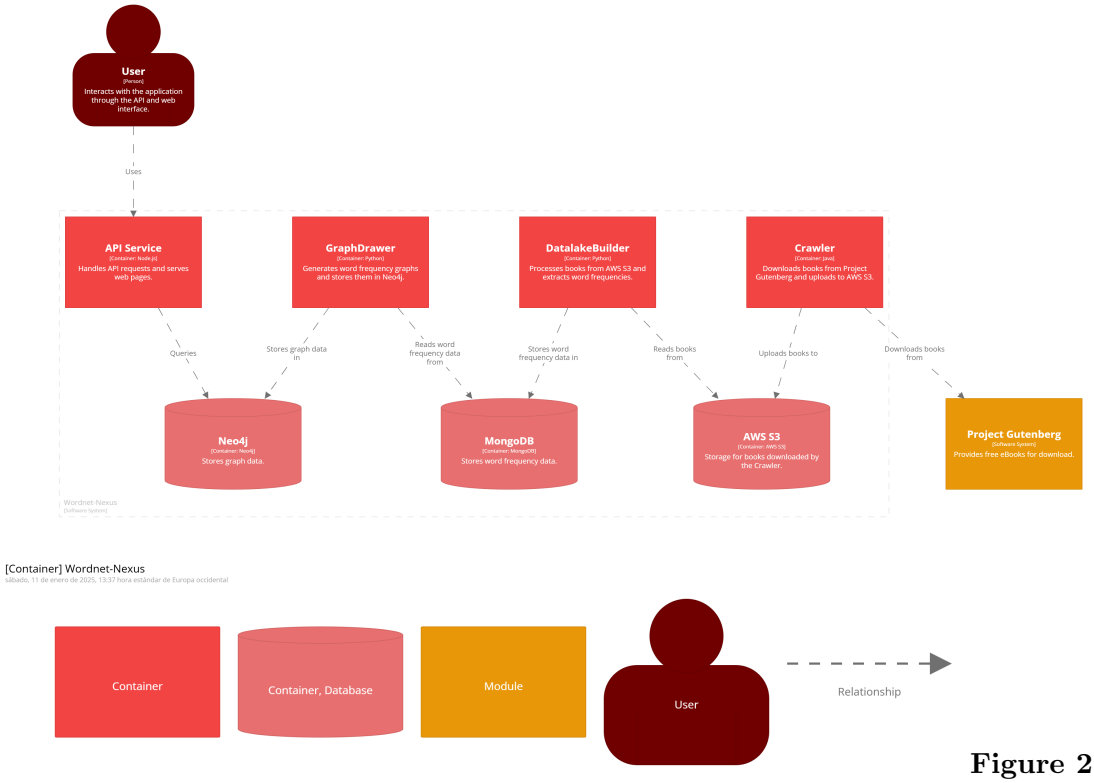

Figure 2. Container diagram.

2.3 Level 3 - Component diagram

The Level 3 C4 model details the main components within each container of the system, providing a more granular view of individual responsibilities and how they interact with each other. In this case, the internal components of the API Service container of the Wordnet-Nexus system are described, which handles user requests and performs queries on the graph database system.

2.3.1 Overview

The API Service container, implemented in Python and JavaScript, consists of several components designed to handle different types of graph-related data processing requests. An Nginx-based proxy manages incoming requests and directs them to the appropriate components for processing.

2.3.2 Components of the API Service Container

1. **Nginx-MainWeb (Proxy):** Acts as a centralized entry point, managing and redirecting HTTP requests to the appropriate submodules of the API container.
2. **AllPaths:** Returns all possible paths between two words in the graph.
3. **ShortestPath:** Calculates the shortest path between two words using efficient algorithms such as Dijkstra or A*.
4. **HighDegreeConnections:** Identifies nodes with the highest number of connections in the graph.
5. **IsolatedNodes:** Locates nodes that have no connections with other nodes in the graph.
6. **MaxDistance:** Finds the maximum distance between two nodes in the graph without including cycles.
7. **StronglyConnected:** Detects densely connected subgraphs within the main graph.
8. **NodeConnections:** Selects nodes with a specific number of connections in the graph.

2.3.3 Associated Database

Neo4j: This graph database system stores all the processed graph data. Each API component queries this database to retrieve relevant information for its operations.

2.3.4 Key Interactions

- The **User** interacts with the system by sending requests to the **Nginx-MainWeb** proxy.
- **Nginx-MainWeb** redirects requests to specific API components, such as **AllPaths** or **ShortestPath**, depending on the requested functionality.
- Each API component directly queries the **Neo4j** database to retrieve the necessary data to complete the requests.

2.3.5 Component Diagram

The following diagram illustrates the internal structure of the API Service container, detailing the components that make it up, their responsibilities, and how they interact with each other and the **Neo4j** database.

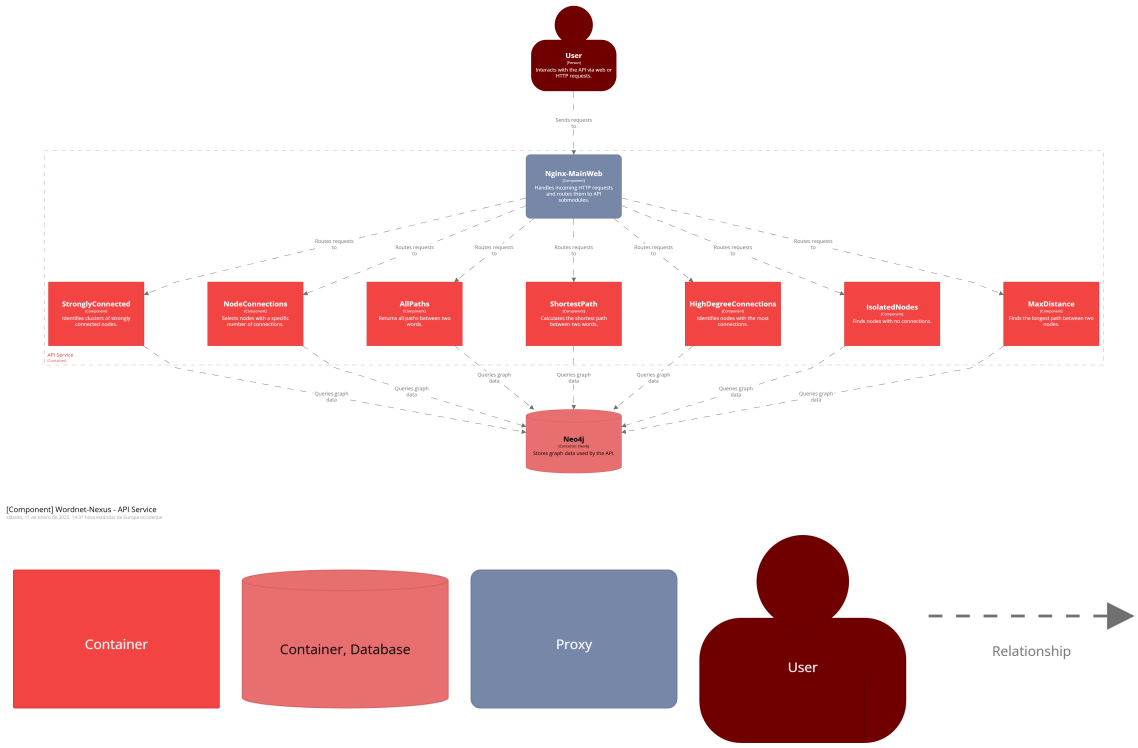

Figure 3. Component diagram.

2.4 Level 4 - Code diagram

The Level 4 of C4 model describes the implementation of individual modules, showing specific classes, methods, and relationships. This level provides a detailed view of the internal design of a module within the system. In this case, the code diagram for the AllPaths module within the API Service container is presented, which is responsible for finding all possible paths between two nodes in the graph.

2.4.1 Description of the AllPaths Module

The AllPaths module is designed to handle requests related to finding all paths between two nodes in the graph. It is composed of the following classes:

2.4.2 Class Relationships

- The `app` class encapsulates the application logic and connects to the `API` package.
- Within `API`, the `ApiInit` class initializes the blueprint and references `Routes`.
- The `Routes` class calls `QueryHandler` to perform specific queries on the Neo4j database.

2.4.3 Code Diagram

The following UML diagram illustrates the class structure, methods, and relationships within the AllPaths module.

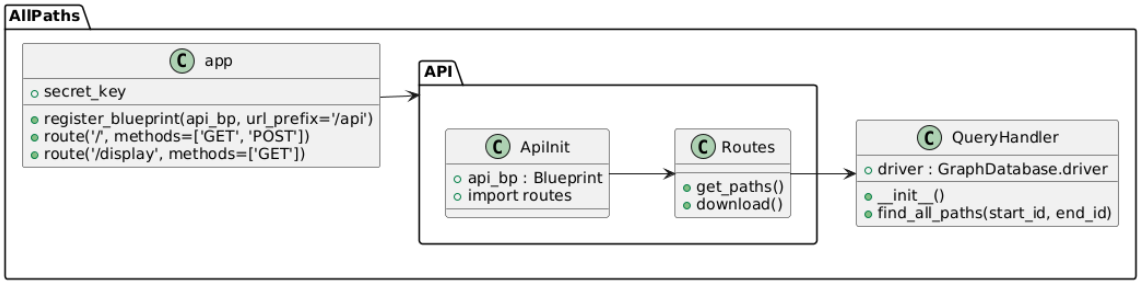

Figure 4. AllPAths Code diagram.

2.4.4 Generalisation for Other Modules

The structure presented for AllPaths is similarly replicated in other modules of the API Service container. Each module implements its own specific logic, such as shortest path search (ShortestPath) or identification of isolated nodes (IsolatedNodes), while maintaining a coherent and modular architecture.

3 Dynamic Diagram

This section describes the interaction between the different components of the system for the ShortestPath functionality, which calculates the shortest path between two words. The flow reflects the interaction between the user and the internal modules of the Wordnet-Nexus architecture.

3.1 Use Case Description

The use case begins when the user sends an HTTP request to the system to calculate the shortest path between two specific words. The dynamic flow of the request traverses the system components as follows:

1. **User:** Sends an HTTP request to calculate the shortest path.
2. **Nginx (Proxy):** Acts as a gateway, routing the request to the appropriate API service.
3. **API Service:** Processes the request, delegating the query to the ShortestPath component.
4. **Neo4j:** Stores and responds with the graph data required by the ShortestPath component.
5. **Response:** The result flows back to the user, traversing the same components in reverse order.

3.2 Dynamic Diagram

The following diagram illustrates the dynamic flow of the ShortestPath functionality:

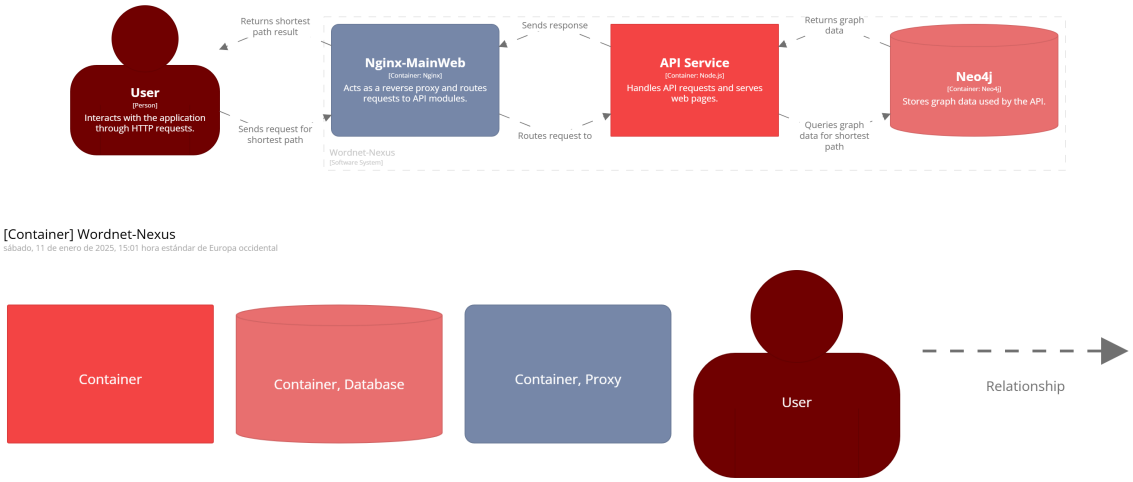

Figure 5. Dynamic Diagram.

3.3 Technical Details of the Components

1. **User (Persona):** Represents an external client interacting with the system via an HTTP interface.
2. **Nginx (Proxy):** A reverse proxy server that routes requests.
3. **API Service (Python and Javascript):** The main service that processes requests and coordinates internal components.
4. **Neo4j (Database):** A data repository that stores the word graph and performs queries.

3.4 Interaction Flow

The complete flow of the use case is described sequentially as follows:

1. The **user** sends an HTTP request via **Nginx** to find the shortest path between two words.
2. **Nginx** routes the request to the ShortestPath component in the **API Service**.
3. The ShortestPath component performs a query on the graph stored in **Neo4j**.
4. **Neo4j** returns the query data to the ShortestPath component.
5. ShortestPath composes a response with the calculation result.
6. The response is returned to the user through **Nginx**.

4 Deployment Diagram

This section details how the components are deployed in the AWS infrastructure. Each diagram illustrates the relationships between the main elements, organized into public and private subnets, highlighting the system's modular and distributed architecture.

4.1 Data Construction Infrastructure

This diagram showcases the deployment of the infrastructure for data construction, including components such as Crawler, DatalakeBuilder, and GraphDrawer, along with storage and database services (S3, MongoDB, Neo4j). These modules work together to process books, extract word frequencies, and build graphs.

4.1.1 Description

- **Public Subnet:** Contains the Nginx server acting as a proxy and the Crawler component for downloading books from Project Gutenberg and uploading them to S3.
- **Private Subnet:** Contains modules for data processing (DatalakeBuilder) and graph generation (GraphDrawer), along with the databases.

4.1.2 Key Relationships

1. The Crawler uploads downloaded books to the S3 bucket.
2. DatalakeBuilder processes the books in S3 and stores frequencies in MongoDB.
3. GraphDrawer uses frequencies from MongoDB to generate graphs and stores them in Neo4j.

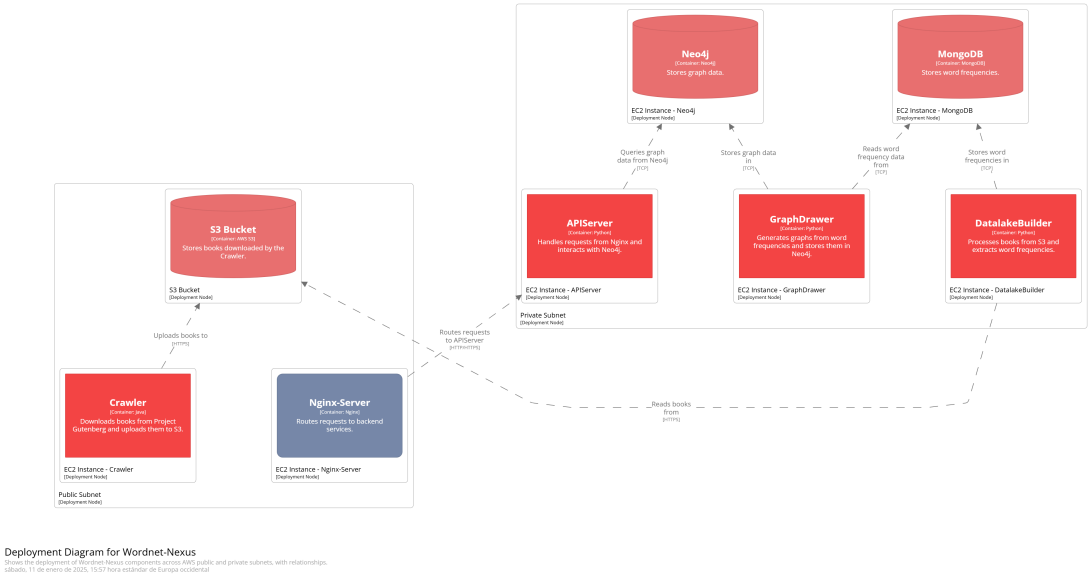

Figure 6. Deployment Diagram - Data Construction Infrastructure

4.2 API Infrastructure

The API infrastructure is divided into three main modules, each focused on specific functionalities. Each diagram illustrates how these modules are distributed between public and private subnets, connecting API components with the Neo4j database and the Nginx server.

4.2.1 API 1: AllPaths and ShortestPath Modules

The infrastructure for the AllPaths and ShortestPath modules is divided into public and private subnets, each serving specific roles in the deployment. The public subnet hosts the Nginx server, which receives incoming HTTP requests and routes them to the appropriate EC2 instance running the AllPaths and ShortestPath modules. The private subnet contains the Neo4j database, which stores the data necessary for calculating all paths or the shortest path between two words.

The main relationships in this deployment include Nginx routing HTTP requests to the API modules and the AllPaths and ShortestPath modules querying the Neo4j database to process these requests. This interaction ensures a streamlined flow of data and functionality between the components.

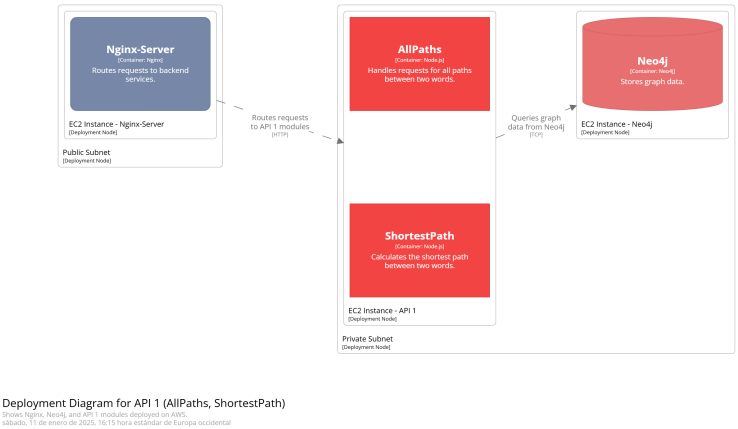

Figure 7. Deployment Diagram - AllPaths and ShortestPath Modules

4.2.2 API 2: NodeConnections and StronglyConnected Modules

The infrastructure for the NodeConnections and StronglyConnected modules operates across public and private subnets. The public subnet hosts the Nginx server, which routes incoming HTTP requests to the EC2 instance containing these modules. The private subnet houses the Neo4j database, which is utilized to identify nodes with a specific number of connections or to detect strongly connected subgraphs within the graph.

Key interactions include Nginx redirecting requests to the API modules and the modules querying Neo4j to retrieve the necessary data for processing and responding to user requests.

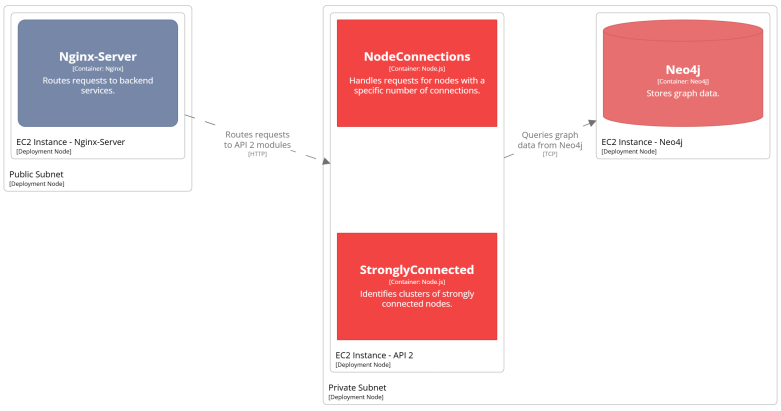

Figure 8. Deployment Diagram - NodeConnections and StronglyConnected Modules

4.2.3 API 3: HighDegreeConnections, IsolatedNodes, and MaxDistance Modules

The infrastructure for the modules handling isolated nodes, high-degree connections, and maximum distance operates on a distributed architecture across public and private subnets. The public subnet hosts the Nginx server, which routes HTTP requests to backend services in the private subnet. The private subnet includes the Neo4j database for graph data management and EC2-hosted API 3 modules, which interact with AWS Lambda functions for specific processing tasks.

- **HighDegreeConnections**: Identifies nodes with the most connections, leveraging the `LambdaHighDegree` function for advanced tasks.
- **IsolatedNodes**: Finds nodes without connections, supported by the `LambdaIsolatedNodes` function for efficient processing.
- **MaxDistance**: Calculates the longest path between two nodes using the `LambdaMaxDistance` function.

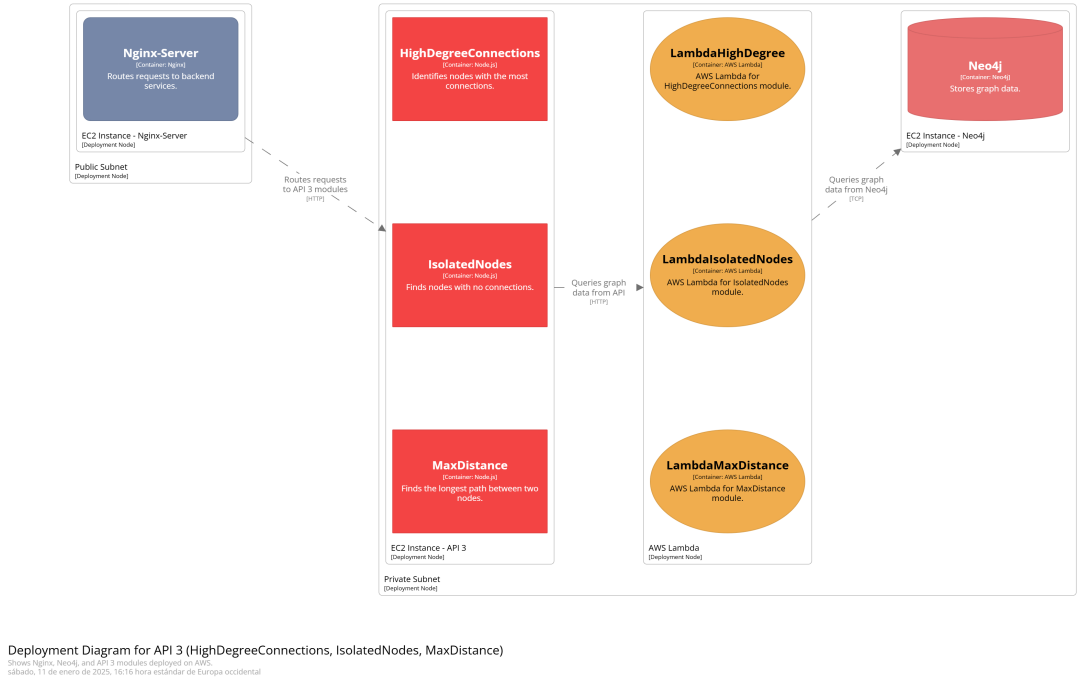

Figure 9. Deployment Diagram - HighDegreeConnections, IsolatedNodes, and MaxDistance Modules

5 Infrastructure layer

The infrastructure layer of this project is designed on a scalable and distributed architecture, leveraging AWS services to ensure flexibility and efficiency. The infrastructure is segmented into a VPC (Virtual Private Cloud) that contains two main subnets: a public subnet, hosting the Nginx server and the web crawling component (Crawler), and a private subnet, housing key services for graph construction and analysis. These services include databases such as MongoDB and Neo4j, as well as specialized modules for the generation, processing, and analysis of graphs. This design ensures the isolation of sensitive services and controlled access, optimizing both the security and performance of the cloud-based system.

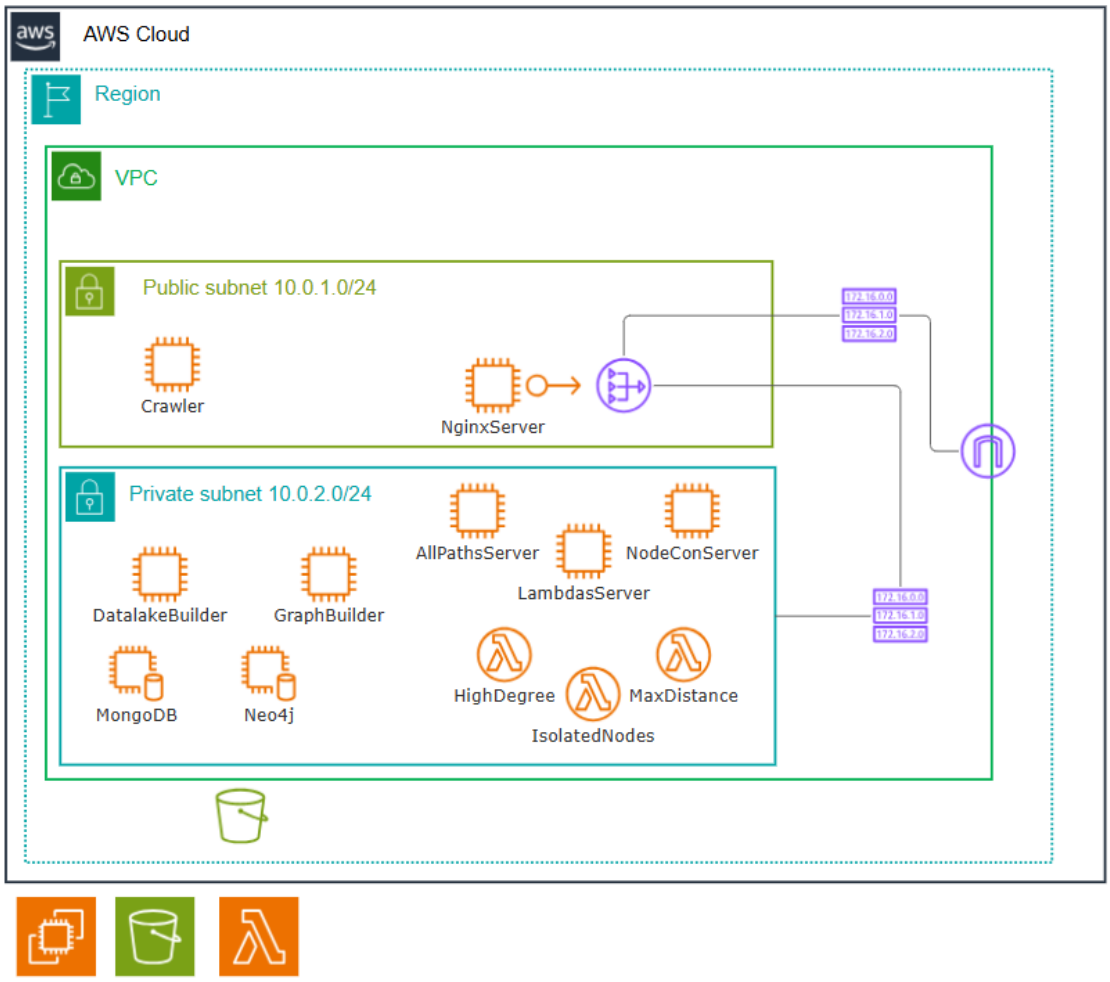

Figure 10. Infrastructure layer.

6 Detail of the Architecture Components

6.1 Crawler

6.1.1 Function:

Download books from Project Gutenberg and store them in an S3 bucket on AWS.

6.1.2 Responsibilities:

1. Connect to the Project Gutenberg API to fetch available books.
2. Download books in a compatible format (e.g., plain text).
3. Upload the downloaded books to the appropriate AWS S3 bucket.

6.2 DatalakeBuilder

6.2.1 Function:

Process the books stored in S3, extract words, and count their frequency, storing the results in MongoDB.

6.2.2 Responsibilities:

1. Download books from S3 for local processing.
2. Process each book to extract words and normalize text (e.g., remove punctuation, convert to lowercase).
3. Calculate the frequency of each word.
4. Insert the processed data into the MongoDB database.

6.3 GraphDrawer

6.3.1 Function:

Build a graph using words and their frequencies from MongoDB and store the result in Neo4j.

6.3.2 Responsibilities:

1. Query MongoDB to retrieve words and their frequencies.
2. Create nodes and edges based on the connections defined between words.
3. Calculate additional properties of the graph, in particular the weight between two words which is the average of their frequency.
4. Save the finalized graph structure in Neo4j for future queries.

6.4 API Service Modules

The API Service is composed of several modules, each designed to perform specific types of graph queries and computations. All modules follow a similar structure and responsibilities, which include designing a REST API, querying the Neo4j graph database, and displaying results through an interactive webpage. Below is a summary of the primary search functionalities performed by each module:

- **AllPaths:** Returns all possible paths between two words.
- **HighDegreeConnections:** Identifies nodes with the highest number of connections. This module uses Lambda functions to execute the search efficiently.
- **IsolatedNodes:** Lists nodes without any connections in the graph. The search is performed using AWS Lambda functions to optimize resource utilization.
- **MaxDistance:** Calculates the longest path between two nodes without cycles. This computation is executed using Lambda functions to handle complex queries dynamically.
- **NodeConnections:** Selects nodes with a specific number of connections.
- **ShortestPath:** Computes the shortest path between two words.
- **StronglyConnected:** Identifies densely connected clusters within the graph.

7 Budget by region

The design and maintenance of the distributed architecture on AWS requires the use of various cloud services. Below is a monthly and annual breakdown of the costs associated with each service, including their respective descriptions and amounts used:

Table 1: Monthly Budget by AWS Service

Service	**Monthly Cost (USD)**	**Description**
EC2	0.00	Crawler
EC2 (x2)	71.54	Datalake and Datamart
EC2 (x2)	15.04	Build Datalake and Datamart
EC2 (x3)	107.31	API
EC2	22.72	NGINX
S3	0.00	Book Bucket
VPC	524.45	NAT, Public IP, VPC, Data Transfer
AWS Lambda	7.38	Lambda Functions

Note. The total budget is 763.48 USD per month, equivalent to 9,161.76 USD per year.

8 Technologies Used

The development of this project required the integration of various technologies, tools, and dependencies, ranging from programming languages to cloud services and documentation tools. Below is a detailed breakdown of the technologies used, organized by categories:

8.1 Programming Languages

- **Python**: Main language for developing scripts, data analysis, and API implementation.
- **Java**: Used for specific modules related to processing and connectivity.
- **HTML, CSS, and JavaScript**: For creating and designing dynamic and interactive web interfaces.
- **Markdown**: Lightweight markup language used for documentation and content formatting.

8.2 Frameworks, Libraries, and Dependencies

- **Jacoco**: A tool for measuring test coverage in Java.
- **Flask**: Lightweight web framework in Python for building APIs.
- **Blueprint**: For modular structuring of Flask applications.
- **nltk**: Natural Language Processing for specific tasks in Python.
- **pymongo**: Library for interacting with MongoDB databases.
- **Cytoscape**: A library for visualization and analysis of networks and graphs.
- **Locust**: A tool for performance testing.
- **Coverage**: A Python tool for measuring code coverage during testing.
- **Unittest**: Standard library module for writing and running tests in Python.

8.2.1 Java Dependencies (Maven)

- `org.json` (v20210307): For handling data in JSON format.
- `org.jsoup` (v1.15.3): For manipulation and data extraction from HTML.
- `com.amazonaws:aws-java-sdk-s3` (v1.12.300): For interaction with Amazon S3.

- `org.junit.jupiter:junit-jupiter` (v5.8.2): Unit testing tool for Java.
- `org.mockito:mockito-core` (v4.0.0): For creating mocks in tests.

8.3 Databases

- **MongoDB**: NoSQL database for flexible data storage.
- **DynamoDB**: Distributed database service from AWS.
- **Neo4J**: Graph database for modeling and analyzing complex relationships. Queries were implemented using **Cypher**, Neo4J's declarative query language, to perform operations like finding shortest paths, identifying clusters, and analyzing node connectivity.

8.4 AWS Services

- **EC2**: Cloud computing instances to run services.
- **S3**: Object storage for unstructured data.
- **VPC**: Virtual Private Cloud for infrastructure control.
- **Elastic IP**: Static IP addresses for service connections.
- **Lambda**: Serverless functions for specific tasks.
- **Boto3**: Python SDK for interacting with AWS services.

8.5 Version Control and Automation

- **Git and GitHub**: For source code management and collaboration.
- **GitHub Actions**: For automating CI/CD processes.
- **Bash**: Scripts for automated tasks.

8.6 Design and Documentation Tools

- **Structurizr**: For creating software architecture diagrams.
- **SmartDraw**: Tool for technical and organizational diagrams.

8.7 Infrastructure and Orchestration

- **Docker**: Containers for packaging applications and their dependencies.
- **NGINX**: Web server for request management and load balancing.

9 Conclusions

The development of this project within a sandbox environment provided by AWS Academy involved certain limitations related to the permissions and resources available. These restrictions required adapting the implementations to the policies and configurations allowed within the environment, adding a practical component to managing real-world constraints in cloud infrastructure.

Throughout the project, solid knowledge of distributed architectures was acquired, understanding how to design scalable, modular, and resilient systems capable of handling large volumes of data. Furthermore, the use of advanced AWS services, such as EC2, S3, VPC, and Lambda, was explored, strengthening the ability to deploy complete solutions in the cloud. Integration and continuous delivery (CI/CD) workflows were also examined through GitHub Actions, enabling the automation of development, testing, and deployment tasks, reinforcing the importance of DevOps practices in modern development environments.

In addition, teamwork and the use of collaborative tools like GitHub enhanced skills in code management and conflict resolution. The importance of documentation and planning was highlighted through tools like Structurizr and SmartDraw, which facilitated the visualization of the systemâs architecture and design.

Finally, the use of complementary technologies, such as Neo4J for graph analysis, Flask for API development, and Docker for containerization, demonstrated how combining diverse tools can contribute to the development of robust and versatile solutions. This project not only allowed for the application of prior knowledge but also introduced new technologies and methodologies, establishing a strong foundation to tackle real-world challenges in data science and cloud-based development.