GraphWord Application

by

José Gabriel Reyes Rodríguez
Jia Hao Yang

January 2025

Contents

Purpose of the Application

The main aim of this project is to start developing skills in the creation of distributed infrastructures to create an API service formed by three different modules, where we are going to face the difficulties that a distributed architecture entails.

The proposed API service consists in querying different characteristics of a word graph, where each word is connected by one difference between them, for example, the word sea would be connected to the word see, but it wouldn't be directly connected with the word ocean because it differs more than one character from the original word.

In addition to our main goal of learning about implementing a functional distributed infrastructure, we also are going to test out how to create a git flow for better working coordination and also try to add a basic work of DevOps for our project to face the difficulties that this entails in order to increase our general knowledge.

Application Structure

This chapter provides a comprehensive overview of the application's core functionalities and the specific responsibilities assigned to each module. Understanding these components is crucial for recognizing how the system operates efficiently and effectively.

The application's data flow is designed to enable seamless processing from data acquisition to analysis, following these key steps:

1. **Book Downloading:** Automatically retrieve books from the Gutenberg Project, a vast repository of free eBooks.
2. **File Processing:** Analyze the downloaded files by extracting words between 3 to 5 characters in length, focusing on identifying meaningful word patterns.
3. **Graph Generation:** Construct a weighted graph where nodes represent extracted words, and edges illustrate relationships based on word co-occurrence or other relevant metrics.
4. **Graph Querying:** Execute various queries on the generated graph to uncover insights, detect patterns, and analyze word relationships.

This modular design ensures scalability and flexibility, facilitating efficient data processing and analysis.

To maintain the graph's relevance and prevent exponential growth, the process of downloading books and generating the graph is automated to run daily at 3 a.m. During this process, the previous set of books and the corresponding graph are deleted, allowing for the creation of a fresh graph each day. This approach ensures the generation of new optimal paths and up-to-date analytical results.

Crawler Module

The **Crawler Module** is responsible for downloading books from the Gutenberg Project, a renowned digital library of free eBooks. This module randomly selects and downloads 20 books in English, storing them locally for further processing.

After downloading, the module extracts the textual content of each book and filters words with a length between 3 and 5 characters. These words, along with their frequency counts, are stored in a structured file for subsequent analysis.

The decision to restrict the word length was made to optimize storage usage and ensure efficient processing within the deployment environment, avoiding unnecessary resource consumption.

Graph Generator Module

The **Graph Generator Module** is responsible for constructing a graph based on the processed file containing words and their frequency counts. Leveraging this data, the module generates the graph structure outlined in the *Purpose of the Application* section.

Once generated, the graph is efficiently stored for future access by the **API Module**, enabling seamless querying and advanced analysis.

For the graph construction, we opted to use the **NetworkX** library instead of **Neo4j** due to the limitations imposed by Neo4j's free plan. This constraint also influenced the decision to restrict the number of words included in the graph, ensuring optimized performance and resource efficiency.

API Module

This module is responsible for handling various user requests through a web service. The service supports the following endpoints:

- **/shortest-path/**: This endpoint calculates the shortest path between two nodes (words) in the graph using Dijkstra's algorithm.
- **/all-paths/**: This endpoint returns all possible paths between two nodes (words) in the graph.
- **/longest-path/**: This endpoint identifies the longest path within the graph, providing insights into the most extensive connections between nodes.
- **/clusters/**: This endpoint identifies clusters or communities within the graph, grouping nodes that are highly interconnected.
- **/high-connectivity-nodes/**: This endpoint returns the top n nodes with the highest connectivity, showcasing key nodes that serve as major hubs in the graph. The number of nodes (n) is configurable by the user.
- **/nodes-by-degree/**: This endpoint selects nodes based on their degree, filtering nodes with a degree between a specified minimum and optional maximum value. It is particularly useful for analyzing nodes with specific levels of connectivity.
- **/isolated-nodes/**: This endpoint retrieves all isolated nodes in the graph, which are nodes with no connections to other nodes.

- **/all-nodes/**: This endpoint returns all the nodes in the graph. If the graph is not loaded into memory, an appropriate error message is returned.
- **/all-edges/**: This endpoint retrieves all edges in the graph, including their associated weights or attributes. If the graph is not loaded into memory, an appropriate error message is returned.

The API ensures efficient and user-friendly access to graph-based operations. It leverages in-memory graph data, providing fast response times and scalability. Additionally, the endpoints are designed to handle a wide range of analytical tasks, from identifying relationships to extracting key insights about the graph's structure.

Tests Designed

The tests designed to ensure the correct deployment and functionality of the project are relatively straightforward. They were primarily developed using the unittest framework, a standard Python library for code testing. Tests have been implemented for each module of the application, including a set of tests for the Crawler module, another for the Graph module, and a separate set for the API module.

The tests verify the proper initialization of methods and class instances, as well as cover various use cases and scenarios. For example, they check whether a file exists or does not exist before being created, or whether a graph is already present, among other conditions.

Additionally, these tests are automatically executed using GitHub Actions with every push or commit made to the repository. It is worth noting that we are uncertain whether this fully meets the conditions for proper CI/CD implementation. Regardless, the tests run automatically, and the code successfully passes all test cases

Technologies used

As previously mentioned, the technologies implemented were GitHub Actions and the unittest framework.

GitHub Actions is a powerful automation tool integrated into GitHub, enabling the creation and execution of workflows for tasks such as continuous integration (CI) and continuous deployment (CD). It allows developers to automatically test, build, and deploy their code with every push or commit.

unittest is a built-in Python framework designed for testing code. It provides a robust structure to create and run unit tests, ensuring that individual components of the application, such as methods or classes, function as expected

Deployment Structure

Technologies used

The technologies chosen for the implementation of the project are as follows: Python, as the primary programming language, will be used to handle all application logic. For the deployment of the AWS infrastructure, LocalStack and Terraform have been selected as the key tools. LocalStack was chosen because it provides a robust and flexible way to emulate AWS services locally, which aligns well with our development and testing needs. Terraform, on the other hand, is used as an infrastructure-as-code (IaC) tool to define and manage the infrastructure. By integrating Terraform with LocalStack, the infrastructure can be provisioned and configured consistently and efficiently, ensuring streamlined deployment and version control of the environment.

Implemented Infrastructure

Due to the limitations encountered while working with AWS Academy accounts in a sandbox environment, we faced several challenges when deploying the infrastructure. As a result, we decided to maintain the project on **LocalStack**, utilizing locally hosted resources that can be replicated. The final infrastructure implemented consists of the following components:

- **Amazon EventBridge:** Triggers the execution of the infrastructure, automating the continuous update of our application.
- **Amazon S3 Bucket:** Stores all downloaded and generated elements produced by the application.
- **Lambda Function (Download and Processing):** Triggered by EventBridge, this function executes the book downloading and processing module. We chose AWS Lambda for this process due to its relatively short runtime within the application, as it is executed only once per day.
- **Amazon SQS Queue (Processing Completion):** Notifies the completion of the previous Lambda function's process.
- **Lambda Function (Graph Generation):** Triggered by the SQS queue, this function generates a new graph based on the processed data from the recently downloaded books. Again, we selected AWS Lambda for this task due to the low frequency of data updates and its manageable volume, making it ideal for serverless execution.

- **Amazon SQS Queue (Graph Update Notification):** Notifies when the graph has been updated, prompting the API service to launch a separate thread to refresh the graph used for query operations.

- **Simulated EC2 Instance (API Service):** Hosts the API service responsible for handling various queries. It also listens to messages from the previous SQS queue to update the in-memory graph retrieved from the S3 bucket. It is important to note that, due to working with the Docker image of **LocalStack Community**, we had to simulate an EC2 instance using a Linux-based Docker image. Both containers were connected to the same network, allowing the simulated EC2 to access LocalStack resources.

- **Cross-Platform Consideration:** This simulated EC2 instance was designed for deployment on Windows systems, which may cause compatibility issues on other operating systems.

- **Terraform Integration:** The simulated EC2 instance behaves similarly to an actual AWS EC2 instance. It can be launched and managed through Terraform, enabling the API module to run without manual intervention.

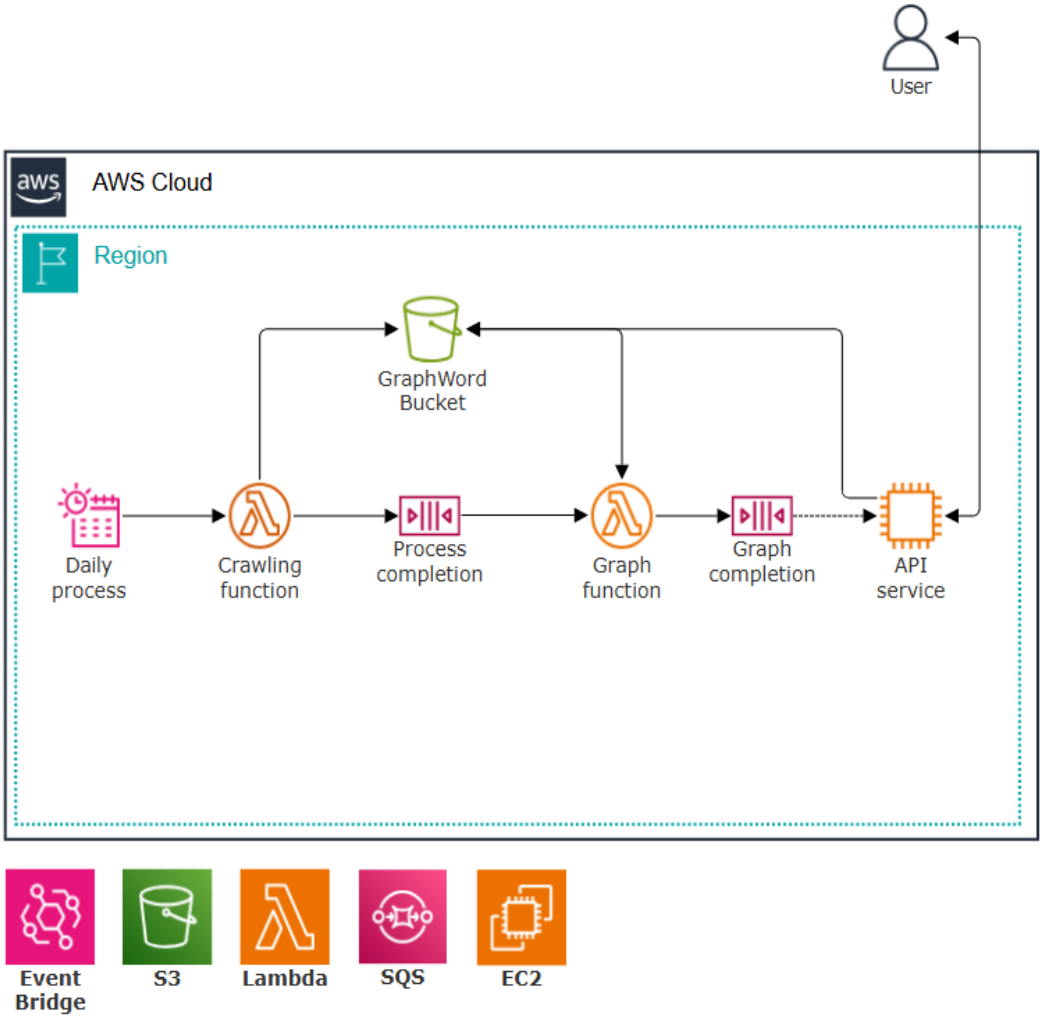

Figure 1: Implemented Infrastructure

Desired Implementation

After acquiring a deeper understanding of AWS and its capabilities throughout the development of this project, we have identified several improvements that would optimize the infrastructure and enhance the application's performance. The ideal infrastructure would consist of the following components:

- **Book Downloading and Processing:** Depending on the volume and frequency of book downloads, transitioning from a Lambda function to an EC2 instance could be more cost-effective for handling large-scale operations. To provide flexibility, this module should be designed for deployment on both Lambda and EC2 instances, allowing for comparative analysis of performance and cost-efficiency. Additionally, the existing **EventBridge** and **SQS** integration would be maintained to manage task scheduling and communication between services. The EventBridge would continue to trigger the download and processing workflows, while SQS would facilitate asynchronous task coordination.

- **Graph Generation:** Due to its event-driven nature and infrequent execution, the graph generation process is well-suited for deployment as a Lambda function. However, similar to the downloading and processing module, we are open to migrating this service to an EC2 instance if performance demands increase or cost optimization requires it.

- **AWS Neptune Integration:** Integrating **Amazon Neptune**, AWS's fully managed graph database service, would significantly improve data accessibility and security for the graph data. Storing the generated graphs in Neptune would

provide seamless integration with the API service, enable more efficient querying, and enhance data protection through built-in encryption and automated backups.

- **API Service:** To ensure high availability, scalability, and fault tolerance, the API service should be deployed using an **Auto Scaling Group (ASG)**. This configuration would allow the system to automatically adjust the number of EC2 instances in response to user demand. The Auto Scaling Group would be paired with an **Elastic Load Balancer (ELB)** to efficiently distribute incoming traffic across the active instances. Additionally, configuring a static IP address (via **Elastic IP**) would provide clients with a stable and consistent endpoint, eliminating the need to reference individual instance IP addresses.

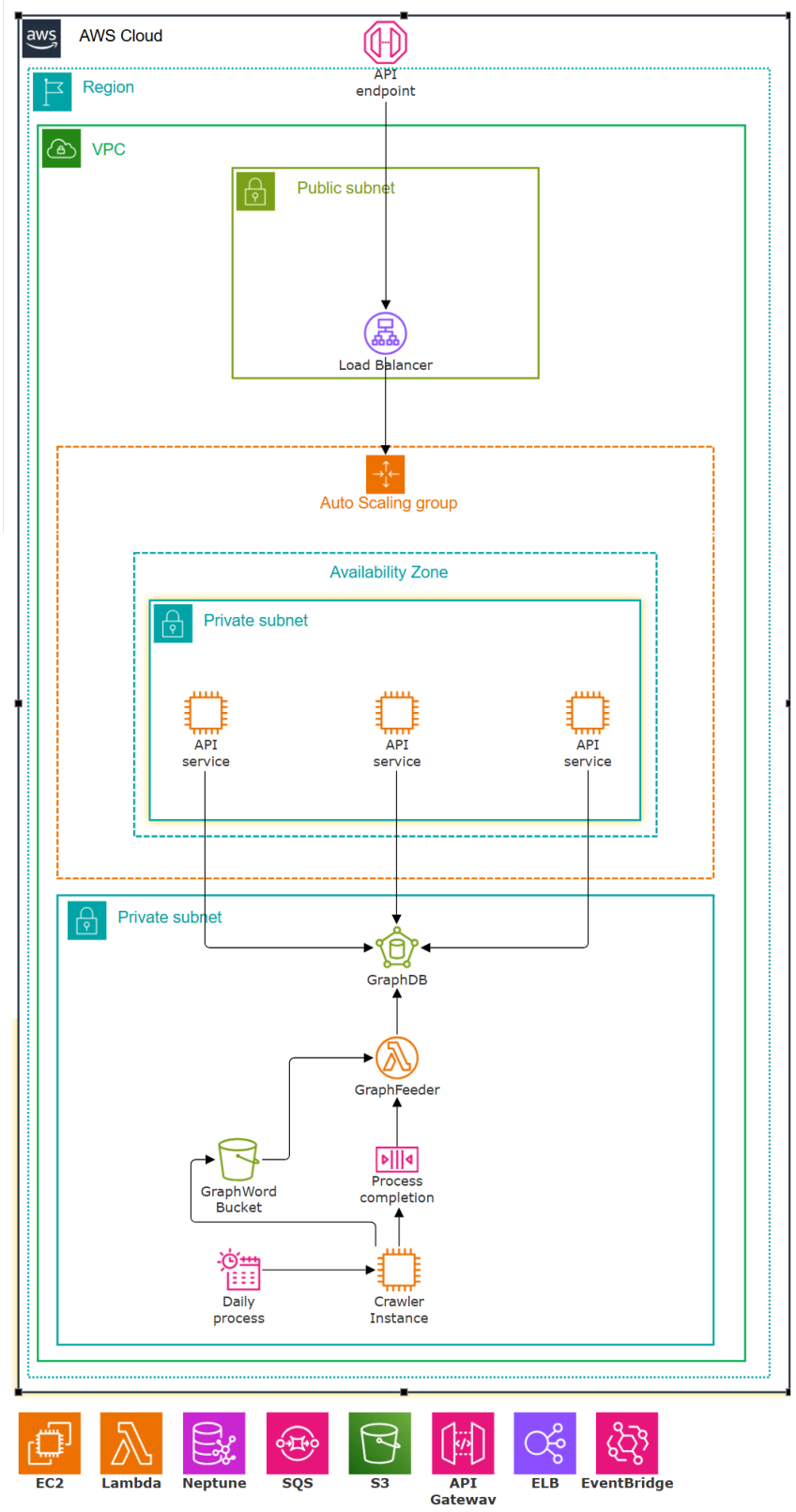

Figure 2: Desired Implementation

Problems Encountered and Solutions

Aws Academy Sandbox Environment

During the development of this project, we encountered several challenges that primarily stemmed from the restrictions imposed by the AWS environment used. These limitations complicated the deployment process, leading us to opt for deployment using **LocalStack**. This solution allowed us to simulate a production-like environment.

LocalStack Limits

Additionally, since the free version of LocalStack does not support the creation of EC2 instances, we had to manually simulate one using a Docker image. This workaround allowed us to mimic the EC2 functionality and proceed with testing and deployment. Furthermore, we weren't able to recreate an VPC in our infrastructure.

Compatibility Between Operating Systems

Another issue arose when downloading the necessary dependencies for the developed Lambda functions. Working on a Windows system led to the installation of Windows-specific dependencies, which were incompatible with the Linux environment expected by AWS Lambda. To resolve this, we utilized a Linux-based Docker image to correctly download and package the required dependencies, ensuring compatibility with the Lambda runtime environment.

CI/CD Difficulties

Another challenge we encountered was during the development of the CI/CD components. We faced significant difficulties understanding and implementing these processes, which led us to postpone their development until we could gain a clearer understanding of their functionality.

Additionally, the complexity of launching our simulated EC2 instance using **Terraform** with the **tflocal** command posed another obstacle, further delaying the Continuous Deployment process.

Future Work

Looking ahead, several enhancements and extensions can be implemented to improve the performance, scalability, and reliability of the application. The following points outline the primary areas of focus for future development:

1. Code Review and Refactoring

Conducting a thorough review of the project's codebase after some time will be crucial for identifying areas for improvement. This process will focus on:

- **Code Refactoring:** Simplifying and optimizing the existing code to improve readability, maintainability, and performance.
- **Modularization:** Breaking down large components into smaller, reusable modules to enhance scalability and ease future feature integration.
- **Error Handling:** Implementing robust error handling and logging mechanisms to ensure better debugging and system reliability.

- **Unit and Integration Testing:** Expanding the test coverage to validate individual modules and the overall system flow, ensuring long-term stability.
- **Containerization:** Dockerizing the application to ensure consistent deployments across different environments.

2. Full Deployment on AWS

Migrating the entire infrastructure to AWS would significantly enhance the system's robustness and scalability. Future work in this area includes:

- **Security Groups and IAM Roles:** Configuring fine-grained permissions and access controls to enhance security.
- **VPC, Subnets, and Networking:** Setting up a production-grade Virtual Private Cloud (VPC) with properly segmented public and private subnets for secure and efficient networking.
- **Elastic Load Balancer (ELB):** Integrating a fully managed load balancer to distribute traffic across multiple instances, improving availability and fault tolerance.
- **Auto Scaling Groups (ASG):** Implementing auto-scaling policies to dynamically adjust computing resources based on demand, optimizing performance and cost.

3. AWS Neptune Integration

Incorporating **Amazon Neptune** as a fully managed graph database would improve the scalability and performance of graph operations:

- **Graph Storage:** Storing the generated graphs in Neptune to support more complex queries and faster access.
- **Enhanced Security:** Leveraging Neptune's built-in encryption and access controls to secure graph data.

4. Monitoring and Logging

Implementing monitoring and logging solutions will ensure better observability and system health management:

- **CloudWatch Integration:** Monitoring metrics, setting alarms, and logging application activity using AWS CloudWatch.
- **Error Detection:** Early detection of anomalies and automatic alerts for performance issues.

5. Performance Optimization

Optimizing the application's performance will be essential as the system scales:

- **Parallel Processing:** Enhancing parallelism in data processing tasks to reduce execution time.
- **Cost Optimization:** Analyzing AWS cost reports and adjusting resources for cost efficiency.

6. Scalability and Multi-Region Deployment

To ensure global availability and reliability, scaling the application across multiple AWS regions could be explored:

- **Global Deployment:** Distributing services across regions to reduce latency and improve resilience.
- **Disaster Recovery:** Implementing backup and failover mechanisms to minimize downtime.

By focusing on these future improvements, we think that the application will evolve into a more scalable, secure, and efficient system capable of handling larger datasets and more complex workloads.

Knowledge Acquired

By the end of this project, we have significantly expanded our knowledge of distributed architectures, both in their design and implementation. This experience has allowed us to deeply understand the challenges and immense potential that come with developing such complex systems.

Through the development of this project, we have achieved the following:

- **Advanced Python Development:** We have deepened our understanding of structured Python development, progressively refining our problem-solving and coding strategies. This project has served as a solid foundation for improving our programming practices, enabling us to revisit the repository in the future to identify new ways to refactor the code, making it more readable and maintainable.
- **Enhanced Docker Proficiency:** By using Docker to build and manage our infrastructure, we have greatly expanded our knowledge of containerization technologies. This has allowed us to gain hands-on experience in creating isolated and scalable environments for deploying applications.

- **Cloud Computing Insights:** We have gained valuable insights into the possibilities offered by cloud computing for large-scale application development. Additionally, we have learned about the essential components required to build robust and scalable infrastructures, unveiling a vast world of opportunities in cloud-based solutions.

- **Testing and Code Quality:** We have improved our understanding of integrating testing frameworks into the development workflow. By implementing tests for the developed code, we have aimed to achieve cleaner, more reliable, and higher-quality software.

This collective learning has equipped us with valuable technical skills and a broader perspective on designing scalable and efficient systems for real-world applications.