UNIVERSIDAD DE LAS PALMAS DE GRAN CANARIA

Lexical Graph Exploration

Grado en Ciencia e Ingeniería de Datos

Tecnología de Servicios para la Ciencia de Datos

Gerardo León Quintana
Jaime Ballesteros Dominguez

January 17, 2025

Index

Introduction

Context

In the course *Technologies of Services for Data Science*, the focus is on leveraging cloud services —particularly Amazon Web Services (AWS)—and integrating continuous integration and continuous delivery (CI/CD) practices using DevOps tools. This project integrates both aspects by combining cloud services and DevOps automation to create a scalable and distributed system.

The primary goal of the project is to design a system capable of managing and performing graph operations, which are crucial data structures for modeling complex relationships. Graphs are widely used in data science to represent social networks, transportation systems, biological structures, and more. Through this project, students develop practical skills that can be applied to real-world data science scenarios.

By implementing this solution on a distributed architecture in AWS, the project emphasizes efficient scalability to handle large volumes of data while ensuring the system's flexibility and robustness [1]. The integration of DevOps practices guarantees that the system's deployment and integration are automated and reproducible.

Problem

The core problem addressed in this project is the construction and management of word-based graphs. In these graphs, each node represents a word, and an edge connects two words that differ by only one letter. For example, the words *dig* and *dog* are connected because they differ by a single letter.

Initially, the system will handle graphs generated from 3-letter words, enabling straightforward model validation. To demonstrate scalability, the system will be extended to handle words of greater length, increasing the graph's size and complexity. This approach allows for analyzing how the graph's size and density evolve with the expansion of word sets and their connections.

The challenges include:

- Efficiently generating large lexical graphs.
- Implementing scalable algorithms for graph operations.
- Deploying and maintaining the system on AWS with CI/CD integration.

Objectives

The project aims to achieve the following objectives:

- **Graph Construction:** Develop a system that constructs a graph from a given set of words, where each word is a node connected to other words differing by a single letter. This includes:
  - Creating dictionaries from various sources to build word sets.
  - Apply filters to include only meaningful and pronounceable words.
- **API Development:** Implement a cloud-based API to allow programmatic interaction with the graph for real-time analysis. The API will support operations such as:
  - *Shortest Path:* Calculate the shortest path between two words using algorithms such as Dijkstra or A *.
  - *All Paths:* Retrieve all possible paths between two words.
  - *Maximum Distance:* Find the longest non-cyclic path between two nodes.
  - *Cluster Detection:* Identify densely connected subgraphs.
  - *High-Degree Nodes:* Detect nodes with the most connections.
  - *Isolated Nodes:* Identify nodes without any connections.
- **Cloud Deployment:** Deploy the solution on AWS using scalable services and automate the deployment process through CI/CD pipelines.

Methodology

Development Tools and Technologies

The Lexical Graph Exploration project was designed to seamlessly integrate software development and infrastructure deployment. The tools and technologies employed include:

- **Python** was the primary programming language used to develop the core functionality. It provided flexibility for implementing complex algorithms for graph construction and analysis, such as shortest path searches, clustering, and connectivity checks.
- **Terraform** was utilized to provision and manage the infrastructure as code (IaC), allowing for a reproducible and scalable deployment setups [2].
- **LocalStack** due to permission limitations in AWS Academy accounts, LocalStack was adopted as a local cloud service emulator [3]. It enabled the simulation of AWS services for development and testing without relying on live AWS infrastructure.

System Workflow

The methodology followed a structured workflow:

1. **Infrastructure Setup:** Terraform scripts were written to define the cloud infrastructure, including Amazon S3 buckets, AWS Lambda functions, and Amazon SQS queues, all simulated via LocalStack.
2. **Data Collection with Web Crawler:** A custom Python Web Crawler was developed to automate the download of books from Project Gutenberg, providing continuous textual data for graph generation.
3. **Graph Construction:** Python scripts processed the datasets to generate graphs where nodes represent words and edges connect words differing by one letter.
4. **API Integration:** A RESTful API was implemented to allow interaction with the graph, supporting operations such as the search for the shortest path and the analysis of isolated nodes.
5. **Testing and Validation:** The system was thoroughly tested in the LocalStack environment to validate functionality and scalability.
6. **Automation with CI/CD:** CI/CD pipelines automated testing and deployment, ensuring continuous integration and seamless updates.

System Design and Architecture

System Overview

The Lexical Graph Exploration system is designed with a modular and scalable architecture that integrates data collection, graph processing, and API interaction. The system consists of two main modules:

- **Crawler Module:** Responsible for downloading books from the Project Gutenberg website and storing them in an Amazon S3 bucket.
- **API Module:** Provides endpoints for users to interact with the generated graphs and perform various analyses.

Architecture

The architecture is built on a cloud infrastructure simulated using LocalStack due to AWS Academy account permission constraints. The system follows a distributed design to ensure scalability and reliability. The key components and their interactions are as follows:

1. **Crawler Module:** A Python-based web crawler downloads books from the Project Gutenberg website and uploads them to the S3 bucket `lgex-download-bucket`.
2. **S3 to SQS Integration:** The `lgex-download-bucket` is configured to send notifications to an Amazon SQS queue (`lgex-queue`) when new files are uploaded.
3. **Batch Processing with Lambda:** Once three files are uploaded, the SQS queue triggers the Lambda function `lgex-batch-processor`. This function processes the newly uploaded files, generates a lexical graph, serializes it as a pickle file, and uploads it to another S3 bucket, `lgex-app-bucket`.
4. **API Module:** The API module loads the graph from `lgex-app-bucket` and exposes endpoints to perform operations such as shortest path searches, cluster detection, and node connectivity analysis.

Used Technologies

- **Python** Used for implementing the web crawler, AWS Lambda functions, and backend logic for graph operations.
- **Terraform** Infrastructure as Code (IaC) tool used to automate the deployment of AWS resources.
- **LocalStack** Emulates AWS cloud services locally, enabling full testing and development without relying on AWS due to permission constraints.
- **AWS Services (Simulated):** Key services include Amazon S3, AWS Lambda, and Amazon SQS, all emulated via LocalStack.

Infrastructure

The system infrastructure was provisioned using Terraform and includes the following components:

- **Amazon S3 Buckets:**
  - `lgex-download-bucket`: Stores raw text files downloaded by the crawler.
  - `lgex-app-bucket`: Stores the generated graph in a serialized pickle format.
- **Amazon SQS Queue:** `lgex-queue` manages communication between S3 and the Lambda function, triggering the batch processor after three files are uploaded.
- **AWS Lambda Function:** `lgex-batch-processor` processes uploaded files and generates the lexical graph.
- **IAM Role and Policies:** Manage secure access permissions for Lambda to interact with S3 and SQS.

Ideal Infrastructure Design

In an ideal production environment without AWS Academy permission constraints, both the Crawler and API modules would be deployed in instances AWS EC2 for better scalability and resource management. However, due to LocalStack's lack of EC2 support, these components were not included in the current infrastructure setup.

Figure [1] illustrates the ideal cloud-based infrastructure envisioned for the Lexical Graph Exploration system. This design leverages various AWS services to ensure scalability, reliability, and efficient processing of large datasets.

Infrastructure Components and Justification

The system architecture is divided into two main subnets: a **Public Subnet** and a **Private Subnet**, each hosting specific components to balance accessibility and security.

- **Application Load Balancer (ALB):** Positioned within the Public Subnet, the ALB efficiently distributes the incoming HTTP/HTTPS requests from users to backend services. Ensure scalability by handling traffic spikes and improves system resilience.
- **Public Subnet:** Hosts the ALB, allowing external users to interact with the system securely while isolating internal processing components in the Private Subnet.
- **Crawler Module (`lgex-crawler`):** Deployed in the Private Subnet as a EC2 instance. This module downloads books from the Project Gutenberg website and uploads them to the S3 bucket `lgex-download-bucket` for further processing.
- **API Module (`lgex-api`):** Also deployed in the Private Subnet as a EC2 instance, this service interfaces with the `lgex-app-bucket` to load and expose graph analysis endpoints. Its isolation ensures sensitive data remains secure.
- **Amazon S3 Buckets:** Two S3 buckets manage data storage:
  - `lgex-download-bucket`: Stores raw downloaded data from the crawler.
  - `lgex-app-bucket`: Stores processed lexical graphs for API consumption.

- **Amazon SQS (`lgex-queue`):** Facilitates asynchronous communication between the crawler and the processing layer. It queues notifications when new files are uploaded, decoupling components and improving system resilience.
- **AWS Lambda (`lgex-batch-processor`):** This serverless function is triggered by the SQS queue after detecting multiple uploads. It processes the data, generates the lexical graph, and uploads the result to the `lgex-app-bucket`. Using Lambda eliminates the need for dedicated servers, reducing costs and simplifying scaling.

Advantages of the Ideal Architecture

This infrastructure design offers several advantages:

- **Scalability:** The use of ALB, SQS, and Lambda enables the system to handle varying loads without manual intervention.
- **Security:** By segregating public and private resources, sensitive operations are isolated within the Private Subnet, reducing the attack surface.
- **Cost-efficiency:** Serverless components such as Lambda scale automatically and only incur costs when executed, optimizing resource usage.
- **Reliability:** Decoupled components via SQS and stateless services increase fault tolerance and system reliability.

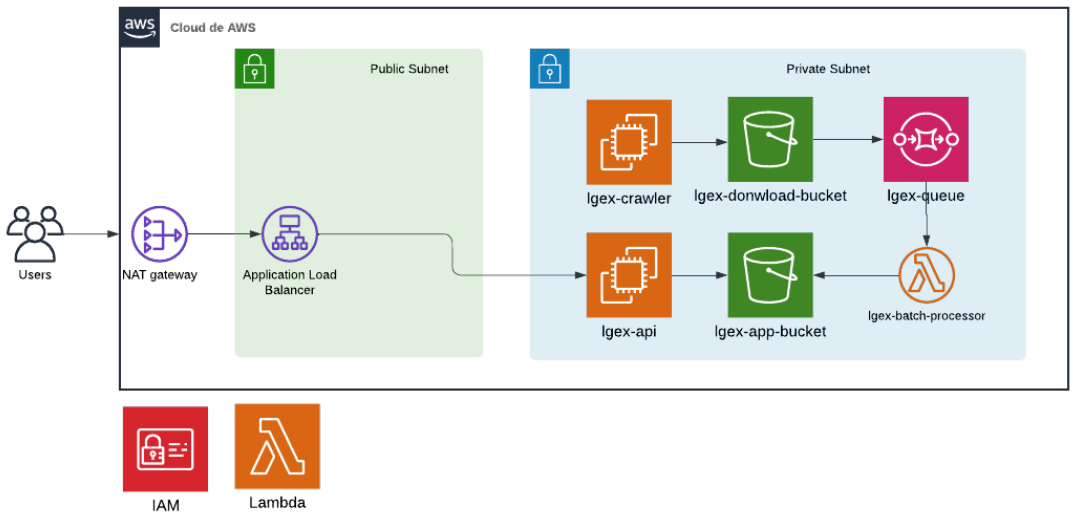

Figure 1: Ideal Cloud-Based System Architecture for Lexical Graph Exploration

Data Flow

1. The crawler downloads books from Project Gutenberg and uploads them to `lgex-download-bucket`.
2. Amazon S3 triggers an event notification to Amazon SQS after each file upload.
3. Once three files are uploaded, the `lgex-batch-processor` Lambda function is invoked.

4. The Lambda function processes the files, constructs a lexical graph, and stores it in `lgex-app-bucket`.
5. The API module loads the graph and provides endpoints for user interaction.

Implementation

Crawler Module

The Crawler Module automates the retrieval and storage of textual data from the Project Gutenberg website. It is implemented as a modular and extensible Python package named `crawler`, designed for high cohesion and maintainability.

- **Crawler Base Class:** The `Crawler` class serves as the foundational structure for specialized crawlers. It defines common behavior and interfaces for web crawling tasks, ensuring seamless integration with other system components.
- **Gutenberg Crawler:** The `GutenbergCrawler` class extends the `Crawler` base class to specifically download books from Project Gutenberg. It efficiently handles HTTP requests, content parsing, and file downloads.
- **AWS Handler:** The `AWSHandler` class abstracts AWS service interactions using the `boto3` client. It encapsulates Amazon S3 operations, streamlining the upload of downloaded books to the `lgex-download-bucket` and enabling smooth communication with AWS resources.
- **Controller:** The `Controller` class integrates the functionality of `GutenbergCrawler` and `AWSHandler`. Its `execute()` method orchestrates the downloading of books and their upload to S3, ensuring efficient and reliable data collection.

Batch Processing with Lambda

Batch processing is implemented as an AWS Lambda function responsible for constructing the lexical graph. The logic is organized into two Python packages: `graph_builder` and `graph_exporter`.

- **Graph Builder Package:** The `graph_builder` package handles the creation of lexical graphs.
  - `GraphBuilder`: Provides a base class defining a consistent interface for graph construction.
  - `WeightGraphBuilder`: Implements the logic for building weighted lexical graphs, where nodes represent words and edges signify word similarity.
- **Graph Exporter Package:** Manages the storage of generated graphs.
  - `GraphExporter`: Defines the framework for exporting graphs.
  - `PickleGraphExporter`: Serializes the generated graph into a `.pkl` file and uploads it to the `lgex-app-bucket` for efficient storage and retrieval.
- **Text Analyzer:** The `TextAnalyzer` class processes downloaded books, extracting word frequencies and relationships using Python's `Counter`. This data form the foundation for building lexical graphs.

API Module

The API Module enables user interaction with the generated lexical graphs, providing a user-friendly interface for querying and analyzing data.

- **Search API:** The `SearchAPI` class defines the API endpoints and initializes the server. Provides accessible endpoints for querying the lexical graph and performing analytical operations.
- **API Feeder:** The `APIFeeder` class loads the serialized graph from the `lgex-app-bucket`, ensuring that the API operates on the most current graph data.
- **Search Algorithms Package:** Contains algorithms for in-depth graph analysis, including:
  - Shortest path searches
  - Cluster detection
  - Identification of highly connected nodes
- **API Controller:** The `ApiController` class manages the execution of the API. Initializes the `SearchAPI` and provides the `execute()` method to launch the service, handling run-time exceptions for stability.

Conclusion

The Lexical Graph Exploration project set out to design and implement a scalable system for constructing and analyzing lexical graphs using cloud technologies and DevOps practices. Although the primary objective was to deploy this system on Amazon Web Services (AWS) to leverage its full scalability, certain constraints—particularly related to AWS Academy permissions—necessitated the use of **LocalStack** as an alternative.

Despite this limitation, the project successfully achieved a functional approximation of the intended cloud-based architecture. By emulating AWS services through LocalStack, we were able to:

- Automate infrastructure deployment using Terraform for reproducibility and scalability.
- Develop and test core system components, including the web crawler, batch processing pipeline, and API module.
- Implement graph construction and analysis operations efficiently in a simulated cloud environment.

This approach allowed us to validate the system design and ensure its components could work together as intended, providing a solid foundation for future deployment on real AWS infrastructure. The modularity of the system and the use of DevOps automation have demonstrated the potential for scalability, flexibility, and robustness in handling large datasets.

Overall, the project effectively bridges the gap between concept and implementation, offering a scalable framework that can be fully realized on AWS in the future.

Future Work

While the Lexical Graph Exploration project successfully demonstrated a scalable system architecture in a simulated cloud environment using LocalStack, several opportunities exist to enhance and fully realize the system's potential on real cloud infrastructure. Future developments can focus on the following areas:

- **Full Deployment on AWS:** Transition the system from LocalStack to live AWS services. Deploy core components—including the web crawler, batch processing pipeline, and API module—using services like EC2, Lambda, S3, and SQS to achieve true cloud scalability and reliability.
- **Performance Optimization:** Optimize the system to handle larger datasets and more complex lexical graphs. This can include implementing distributed computing solutions, such as AWS EMR or Apache Spark, to improve processing speed and efficiency.
- **Real-Time Data Processing:** Incorporate real-time data ingestion and processing using AWS services like Kinesis or Kafka. This would allow continuous updates to the lexical graph as new data becomes available.
- **Web-Based Visualization Interface:** Develop an interactive web interface for visualizing and exploring the lexical graphs. This would make the system more user-friendly and accessible to non-technical users.
- **Advanced Graph Analysis:** Integrate more sophisticated graph algorithms, such as community detection, semantic similarity measures, and weighted graph operations, to support deeper linguistic analysis.
- **Security and Access Control:** Implement robust security measures, including IAM roles, encryption, and API authentication, to protect data and ensure secure access when deployed on AWS.
- **Multi-language Support:** Expand the system to handle multilingual datasets, enabling cross-lingual lexical analysis by incorporating additional language processing tools and datasets.
- **Scalable CI/CD Pipelines:** Enhance the CI/CD workflow to support automated testing, deployment, and monitoring on AWS, ensuring reliable updates and maintenance of the system.

Bibliography

1. Amazon Web Services, *AWS Documentation*, Available at: https://docs.aws.amazon.com/
2. HashiCorp, *Terraform Documentation*, Available at: https://developer.hashicorp.com/terraform/docs
3. LocalStack, *LocalStack Documentation*, Available at: https://docs.localstack.cloud/