

Proyecto 1 - IC4302 Bases de Datos II

Introduction

Following is the documentation for 'Proyecto 1' of the course IC4302 - Databases II. This project assignment focuses on developing an application that integrates multiple technologies, including MariaDB, Elasticsearch, RabbitMQ, and the Hugging Face API. The application is designed to handle various tasks such as loading and processing datasets, generating text embeddings, and interacting with these databases. To optimize performance, the system includes caching using Memcached, and it is instrumented with Prometheus to monitor critical metrics such as HTTP request counts, query response times, object processing times, and cache efficiency.

The application is deployed in a Kubernetes environment using Helm Charts, ensuring scalability, manageability, and observability. Each component is containerized using Docker, packaging only the necessary dependencies and database interactions. For performance evaluation, the system collects Prometheus metrics and displays them on Grafana dashboards, enabling real-time monitoring and analysis of the system's performance under different workloads.

Additionally, a React-based UI is implemented, allowing users to interact with the system in a user-friendly way. The UI supports functionality such as user registration and login, submitting prompts to query songs using vector search on Elasticsearch, and interacting with friends through a social feed. Users can search for song-related prompts, follow friends, and manage their profiles and posts. The UI is deployed as a Kubernetes Deployment and exposed via NodePort for external access. The UI interacts with the backend API to perform various tasks, such as querying the database, processing text, and caching results.


Team Members

- Victor Aymerich
- Anthony Barrantes
- Fabricio Solis
- Melanie Wong
- Pavel Zamora

Next up, you will find the requirements to run the application, the steps to execute it, testing examples and the recommendations and conclusions we have gathered from the project.

Requirements

The requirements for the project are the following:

- Create an user in [DockerHub](#)
- Install [Docker Desktop](#), if you are using MacOS, please make sure you select the right installer for your CPU architecture.
- Open Docker Desktop, go to **Settings > Kubernetes** and enable Kubernetes. 

- Install [Kubectl](#)
- Install [Helm](#)
- Install [Visual Studio Code](#)
- Install [Lens](#)

Building the docker images

There is a script to build the docker images, to execute it in a bash shell execute:

```
cd ./P1/docker  
./build.sh nereo08
```

Change **nereo08** to your DockerHub username

Please take a look on the script contents to make sure you understand what is done under the hood. This script will build the images for the components of the project, including the database, the API, the cache, and the monitoring components.

Helm Charts

Configure

- Open the file **P1/charts/app/values.yaml**
- Replace **nereo08** by your DockerHub username

```
config:  
  docker_registry: nereo08
```

Install

Execute:

```
cd ./P1/charts  
./install.sh
```

Here we are installing the components in the Kubernetes cluster. The script will install the databases, the API, the cache, the backend and fronted, plus the monitoring components. It is important to have the DockerHub username set correctly in the script to ensure the correct components are installed.

Uninstall

Execute:

```
cd ./P1/charts
./uninstall.sh
```

This script will uninstall the components from the Kubernetes cluster. It is important to have the DockerHub username set correctly in the script to ensure the correct components are removed.

Access Debug Pod

```
# copy the name that says debug from the following command
kubectl get pods
# then replace debug-844bb45d6f-9jt45 by that name
kubectl exec --stdin --tty debug-844bb45d6f-9jt45 -- /bin/bash
```

In case you need to access the debug pod to check the logs or execute some commands, you can use the previous command to access it.

How to Test

To test the whole project together, you can follow the next steps: First you need to do the build and install steps, in this time you have to choose the components that will be used for this you have to change in TC1\charts\app\values.yaml, after that to verify that the project is working correctly you can follow the next steps:

- Check Docker images are running, you can check this in the Docker Desktop application. In this area it is important to check the different images, you can check the logs of the images to see if there are any errors. You should check the logs of the S3 Crawler, Backend API, Hugging Face API, Ingest, the MariaDB, Memcached, Prometheus, and Grafana images. You can check the logs of the images by clicking on the image and then clicking on the logs button. Y
- After that you can check in Lens that the pods are running correctly, here you can check the logs of the pods to see if there are any errors and the status of the pods.
- Once you have checked all the logs and the data is being processed correctly, you can consider to go to Grafana to see the metrics of the project. To do this you should forward the port of the Grafana service called **grafana-deployment** to you local machine. From here you can access the Grafana dashboards by going to <http://localhost:> (Port you decide) and logging in with the credentials **admin** and the password found in the GF_SECURITY_ADMIN_PASSWORD variable which can be found while observing the logs of the Grafana pod.
- Once you enter Grafana, you can see various dashboards displaying metrics for different components of the project. The databases being monitored include MariaDB, Elasticsearch, and Memcached, all of which are critical to the system's performance. These metrics provide insights into the behavior and efficiency of the databases and caching systems. Additionally, each of the

Python components, such as the Hugging Face API, S3 Crawler, Backend API and Ingest service, have dedicated dashboards that track their respective metrics. These include the number of requests, object processing times, and error rates, providing full visibility into how each part of the system is functioning. This allows for thorough monitoring of all the critical processes in real time. These metrics are made possible by Prometheus, which continuously scrapes data from the components, stores it in its internal database, and then feeds this information to Grafana for visualization. Monitoring is a crucial part of this project, as it helps identify potential bottlenecks and areas for optimization, offering a real-time overview of the system's performance.

- The frontend is also available for testing, you can access it by forwarding the port of the frontend service called **frontend-deployment** to your local machine. From here you can access the frontend by going to `http://localhost:` (Port you decide) and interact with the UI. The frontend provides a user-friendly interface for users to interact with the system, including features such as user registration, login, and social feed. Users can submit prompts to query songs using vector search on Elasticsearch, follow friends, and manage their profiles and posts. The frontend interacts with the backend API to perform various tasks, such as querying the database, processing text, and caching results.

Recommendations and Conclusions

Recommendations

To successfully complete the project, the following recommendations are provided to help you navigate the different components and technologies involved:

1. Learn How to Use a Kubernetes Cluster

- **Recommendation:** It is very important to get a basic understanding of Kubernetes, what is their function and how it works, as the whole project is designed to run in a Kubernetes cluster.
- **Reason:** Knowing how to deploy and manage the application in Kubernetes will help you run the project in a real-world environment.

2. Learn How to Use Helm Charts

- **Recommendation:** Learn how to use Helm charts, which are used to deploy the application in Kubernetes. It is relevant to understand how they work and what is their function so you can deploy the application correctly.
- **Reason:** The project uses Helm charts to deploy the application in Kubernetes. Knowing how to use Helm charts will help you manage the deployment of the application.

3. Get Familiar with MariaDB

- **Recommendation:** Learn how to access and use MariaDB, which is the main database used. You should learn the basics, including knowing how to connect to it, view tables, and check data for example.

- **Reason:** You will need to know how to connect to MariaDB to ensure certain components are working correctly, as well as to verify that data is being updated and stored properly.

4. Get Familiar with Elasticsearch Indexing

- **Recommendation:** Study the basics of Elasticsearch, focusing on how to create and manage indices where data will be stored.
- **Reason:** Understanding how to create and manage indices in Elasticsearch is crucial for storing and retrieving data efficiently.

5. Monitor Resource Usage

- **Recommendation:** Keep an eye on resource usage (CPU, memory) while running the script to ensure it operates within acceptable limits.
- **Reason:** Monitoring helps prevent performance bottlenecks and ensures that the script runs efficiently, particularly for large datasets.

6. Document Configuration and Steps

- **Recommendation:** Document the configuration parameters and execution steps clearly for future reference and ease of use.
- **Reason:** Clear documentation helps users understand the setup and execution process, making it easier to troubleshoot and replicate the environment.

7. Understand the Data Schema

- **Recommendation:** Familiarize yourself with the data schema and relationships between tables to write efficient queries.
- **Reason:** Understanding the data schema helps optimize queries and ensures accurate results when fetching data from the databases.

8. Learn How to Use RabbitMQ

- **Recommendation:** Learn how to use RabbitMQ, which is used generate messages and communicate between components.
- **Reason:** Understanding how to use RabbitMQ will help you send messages to the Ingest service and process data efficiently.

9. Understand Prometheus and how to use it

- **Recommendation:** Learn how to use Prometheus to monitor the different components of the project and how to scrape the metrics.
- **Reason:** Prometheus is a key component of the project as it captures the metrics of the different components and stores them in its database.

10. Understand Grafana and the consequent Dashboards

- **Recommendation:** Learn how to use Grafana to visualize the metrics of the different components of the project.

- **Reason:** Grafana is a key component of the project as it allows you to see the metrics of the different components and see how they are performing in real time.

11. Understand Memcached

- **Recommendation:** Learn how to use Memcached, which is used as a cache system in the project.
- **Reason:** Understanding how to use Memcached will help you cache data and understand how it impacts the performance of the application.

12. Understand React

- **Recommendation:** Learn how to use React, which is used to create the frontend of the project.
- **Reason:** Understanding how to use React will help you interact with the UI and understand how the frontend interacts with the backend.

Conclusions

1. Optimal Technology Integration: The integration of diverse technologies, helps create a application that uses optimal solutions for different tasks, such as data storage, retrieval, caching, and monitoring. Various components permit the application to develop different functionalities and using the best technology for each task.
2. Containerized Scalability: The use of Kubernetes allows the system to scale horizontally, adding or removing instances as needed to handle varying loads. This ensures that the project can support high levels of concurrency and data throughput, essential for real-time applications.
3. Efficient Data Handling: The use of MariaDB allows handling large volumes of data. Additionally, integrating Elasticsearch into the project provides options for fast and scalable searches in even larger or distributed databases.
4. Advanced Observability and Metrics: The integration with Prometheus and Grafana allows for observability of the system's performance metrics, including object and row processing times, error rates, and system loads. This level of monitoring is essential for maintaining high system reliability and optimizing resource usage.
5. Interactive User Interface: The project provides a user-friendly interface for users to interact with the system, including features such as user registration, login, and social feed. Users can submit prompts to query songs using vector search on Elasticsearch, follow friends, and manage their profiles and posts. Vector search is a very powerful algorithm that allows users to search for songs based on the similarity of the lyrics, providing a unique and efficient way to discover new music.
6. Efficient Embedding Generation: The project incorporates Hugging Face's API to generate embeddings for song lyrics, making the use of machine learning models practical for vector-based searches. This brings advanced AI-driven functionality to the project, enhancing search precision and user experience.

7. Scalable and Reliable Caching: The project uses Memcached as a caching system to store frequently accessed data, reducing the load on the databases and improving response times. Caching is essential for optimizing performance and ensuring a smooth user experience, especially for read-heavy applications.
8. Efficient Data Handling: The project effectively manages data from external sources, such as S3 buckets. It ensures that data is processed, embedded with vector search capabilities, and seamlessly stored in Elasticsearch, maintaining data integrity and scalability.
9. Flexible and Extensible Design: The application's architecture allows for easy integration of new features or components. Whether it's adding more data sources, expanding the Elasticsearch index, or integrating additional APIs, the design supports continuous development and flexibility.
10. Logging Usage: The project uses logging to track the performance of the different components, helping to identify issues and optimize the system. Logging is essential in the real world for monitoring the system's behavior and ensuring that it operates efficiently. It provides valuable information for troubleshooting and performance tuning.
11. Real-Time Execution: The project is designed to process data in real time, ensuring that new data is ingested, processed, and stored efficiently. This real-time processing capability is essential for applications that require up-to-date information and fast response times.
12. Connectivity and Component Interaction: The project demonstrates effective communication between components, such as RabbitMQ messages in the S3 Crawler triggering data processing in the Ingest service which connects to the Hugging Face API to generate embeddings. Also, the frontend interacts with the backend API to perform various tasks, such as querying the database, processing text, and caching results. This seamless interaction between components ensures that the system functions as a cohesive unit, delivering the desired functionality to users.

Components

Databases:

The databases used in the project are MariaDB and ElasticSearch. These databases are widely used in the industry and offer features for storing and managing data.

MariaDB is a popular open-source relational database management system that is compatible with MySQL. It provides excellent performance, scalability, and reliability. MariaDB ensures data integrity and consistency. It also offers advanced features such as replication, clustering, and high availability, making it suitable for handling large volumes of data.

Elasticsearch is also a popular NoSQL database that is used for full-text search and analytics. It is designed for real-time search and analysis of large volumes of data. It offers features like full-text search, aggregations, and spatial search, making it suitable for handling complex data structures and performing advanced queries.

MariaDB is well-suited for handling complex data structures and performing complex queries. They offer strong data consistency, reliability, and security. The choice between the two databases depends on specific project requirements, familiarity with the technology, and the need for specific features or compatibility with existing systems. On the other hand, Elasticsearch is ideal for full-text search and analytics, providing fast and scalable search capabilities for large volumes of data.

Overall, the combination of MariaDB and ElasticSearch in this project ensures efficient and reliable data storage and retrieval, enabling the application to handle large amounts of data effectively.

The structure of the databases is designed to store data related to songs, users, and objects. The MariaDB database stores user information and object data, while the Elasticsearch database stores song titles, artists, and lyrics.

MariaDB has the following tables:

- **processed_objects:** Stores information about processed objects, including the `object_key` and the `processed` status.
- **users:** Contains user-related information such as `user_id`, `name`, `username`, `password`, `friends`, `email`, `biography`, `created_at`, and `updated_at`.
- **friends:** Manages friendships between users, storing `friend_id`, `user_id`, `friend_user_id`, `created_at`, and `updated_at`.
- **prompts:** Holds data about user prompts, including `prompt_id`, `user_id`, `likes`, `prompt`, `created_at`, and `updated_at`.
- **likes:** Keeps track of likes on prompts, storing `like_id`, `user_id`, `prompt_id`, and `created_at`.

Data Loading Scripts:

As mentioned before, Elasticsearch and MariaDB are populated with certain data each. Elasticsearch is populated with songs and MariaDB is populated with Users and Objects data. The data inserting scripts are written in Python and use the Elasticsearch and MariaDB Python libraries to interact with the databases. Elasticsearch is populated with song data, which includes song titles, artists, and lyrics, allowing for efficient retrieval and search capabilities. In contrast, MariaDB is populated with user and object data, such as login information, user profiles, and object details. The scripts in charge of loading the data into the databases are optimized to handle large volumes of data efficiently, ensuring data integrity throughout the loading process. The scripts help assure that both databases are populated with the necessary data for the application to function correctly.

S3 Crawler:

The S3-RabbitMQ Processor is a Python-based application responsible for managing data flow between AWS S3 and RabbitMQ. Upon execution, it lists all objects stored in a specified S3 bucket, filters them using a predefined prefix, and sends their object keys as messages to a RabbitMQ queue. These keys are processed to trigger downstream tasks, such as data extraction or transformation. Prometheus metrics are integrated into the app, tracking the total number of S3 objects processed and the time taken for each execution cycle. It starts by fetching environment variables for configuring AWS credentials, S3 bucket names, and RabbitMQ connection details. For each valid object in S3, the application sends the object key to RabbitMQ using a specified queue. Metrics such as the total number

of objects processed and the time taken for each operation are tracked and exposed for Prometheus monitoring. The application runs as a long-running process and is designed to be deployed in a Kubernetes environment where all configuration details are injected through environment variables. The application also supports logging for error handling, providing clear insights into operational statuses and error states. This system efficiently handles S3 objects, ensures message queuing, and offers robust observability through Prometheus, making it a key part of the larger data pipeline.

Hugging Face API:

The Hugging Face is a python application that implements an API through Flask. It uses the Hugging Face Sentence Transformers model to encode text into embeddings. It also integrates Prometheus monitoring to track metrics. The model "all-mpnet-base-v2" is loaded using Sentence Transformers and is used later to encode text into embeddings. It exposes metrics for Prometheus like minimum, maximum and average time for requests and the total amount of requests. This application measures the time taken for each request and updates prometheus metrics accordingly. The application has two endpoints: /encode and /status. Encode takes POST requests with a JSON body containing the key "text". If the text is missing, it returns an error message. The Sentence Transformers model encodes the text as an embedding and returns it as a JSON. Status is a GET endpoint that encodes the hardcoded string "El sistema está funcionando correctamente." and returns the embedding. This is used to check that the system is working correctly. This API provides a service for encoding text using a Hugging Face model, while integrating Prometheus for monitoring and performance tracking.

Ingest:

The Ingest application is a Python component responsible for processing data from RabbitMQ messages. When it receives a message, it first checks MariaDB to see if the object has been previously processed; if so, the message is ignored. If the object is new, it downloads the data from an S3 bucket and processes it as a CSV file. For each row in the CSV, an embedding for the "lyrics" field is generated using the Hugging Face API and added to each row as a new field named "embeddings," formatted as a dense vector for Elasticsearch. The document, including the embeddings, is then added to Elasticsearch in the index named "songs," and MariaDB is updated to indicate that the object has been processed. This application runs as a Kubernetes deployment and all necessary configurations are injected as environment variables. Additionally, it exposes various metrics for Prometheus monitoring, including the maximum, minimum, and average processing times for both objects and rows, as well as the total number of objects and rows processed, and the number of rows with errors. The ingest is a critical component of the project, as it processes incoming data from RabbitMQ messages, generates embeddings for the lyrics field using the Hugging Face API, and updates the data in both Elasticsearch and MariaDB. The ingest ensures that new data is processed efficiently and accurately, enabling users to search and retrieve song data effectively.

Backend API:

This is an API developed in Flask, that starts in the port localhost:31000 ever since you do the install.sh script. It's main purpose is to interact with the databases and provide the necessary functionality for the frontend to work. It uses several libraries to enhance its functionality. Key libraries include:

- **Flask-CORS**: Allows communication between different domains through CORS (Cross-Origin Resource Sharing).
- **Requests**: Used to efficiently make HTTP requests.
- **MariaDB**: Facilitates the connection and manipulation of MariaDB databases.
- **Elasticsearch**: Used to perform advanced searches and data management in this NoSQL database.

The project structure includes:

- **app.py**: For the central configuration of the application.
- **config.py**: To manage environment variables.
- **database.py**: To establish the connection with MariaDB.

The API routes are organized within the **routes** directory, and additional utilities are contained in **utils.py** to support complementary operations like the vector search. The backend is easily deployable through Docker using a Dockerfile, which facilitates its implementation and scalability in production environments.

Endpoints

- **/register (POST)**: Used to register a new user by sending the necessary user data.
- **/login (POST)**: Used for logging in an existing user by providing login credentials (username and password).
- **/followOrUnfollow (POST)**: Allows a user to follow or unfollow another user.
- **/find (POST)**: Used to search for users based on their name or username.
- **/isFriend (POST)**: Checks if a particular user is a friend of the logged-in user.
- **/friends (POST)**: Retrieves a list of the logged-in user's friends.
- **/likeOrUnlike (POST)**: Used to like or unlike a specific post (prompt).
- **/hasLiked (POST)**: Checks if a user has liked a specific post.
- **/feed (POST)**: Returns a feed of posts from the user's friends and themselves.
- **/search (POST)**: Searches for posts based on a query string provided by the user.
- **/prompt (POST)**: Fetches the top n results of a specific prompt.
- **/postPrompt (POST)**: Used to submit a new post or prompt.
- **/editPrompt (POST)**: Allows the editing of an existing post or prompt.
- **/deletePrompt (POST)**: Deletes a specific post or prompt.
- **/profile (POST)**: Fetches a user's information, including their posts and profile details.

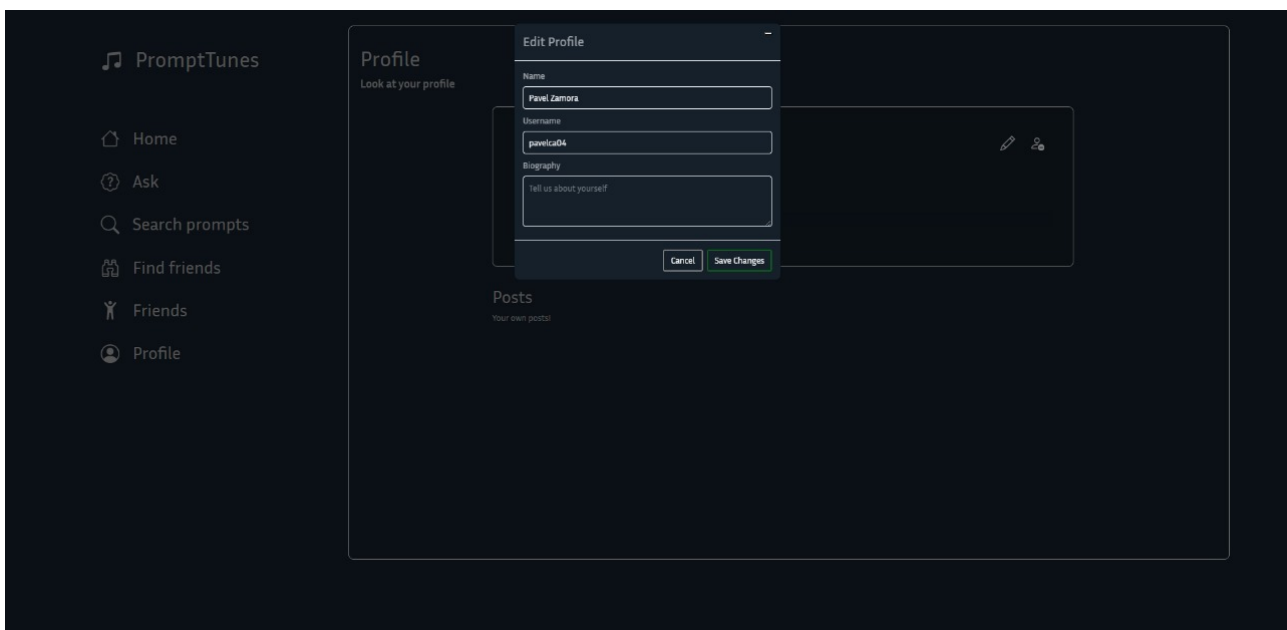
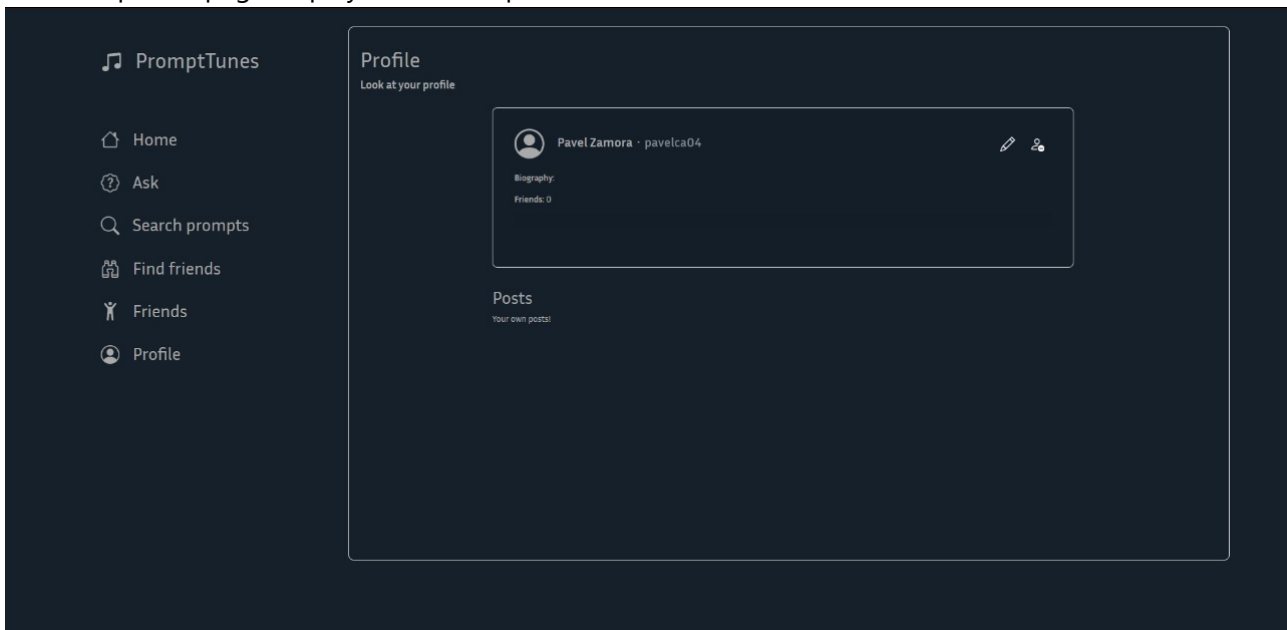
- **/updateProfile (POST)**: Allows users to update their profile information, such as username, bio, or other details.

UI:

The frontend for the application is built using React and runs on port `localhost:30080` on your machine. This frontend is organized into various pages, each serving a specific function within the application. Below is an overview of the main pages:

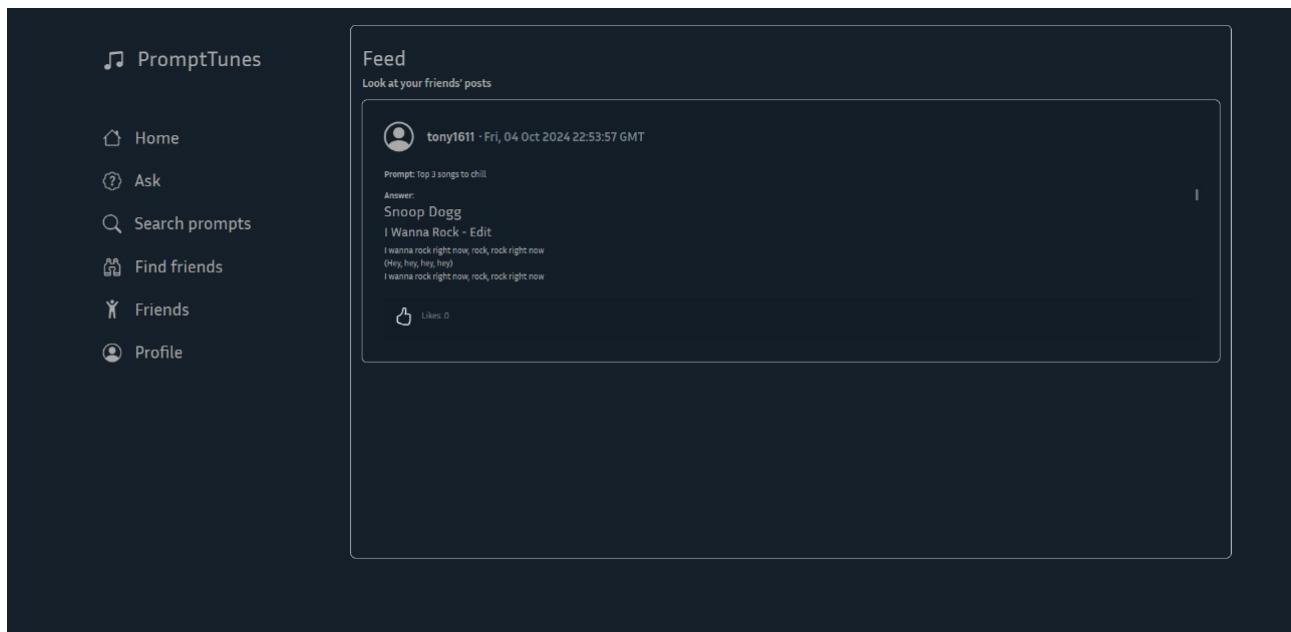
User Profile

The user profile page displays the user's personal information.



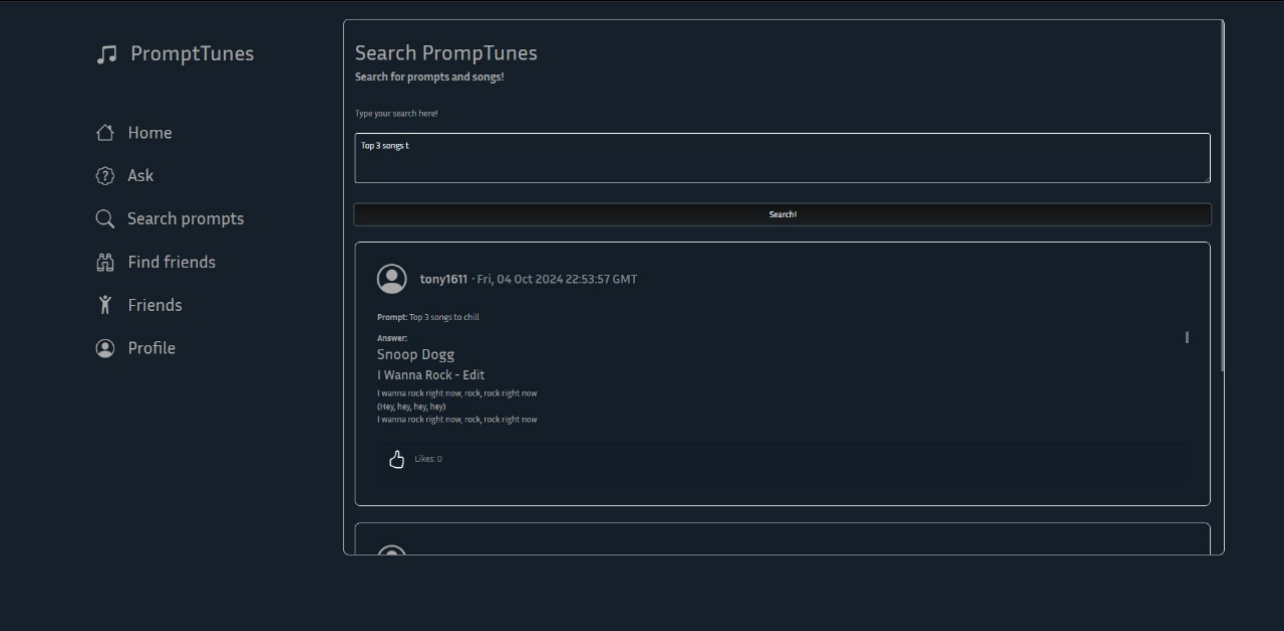
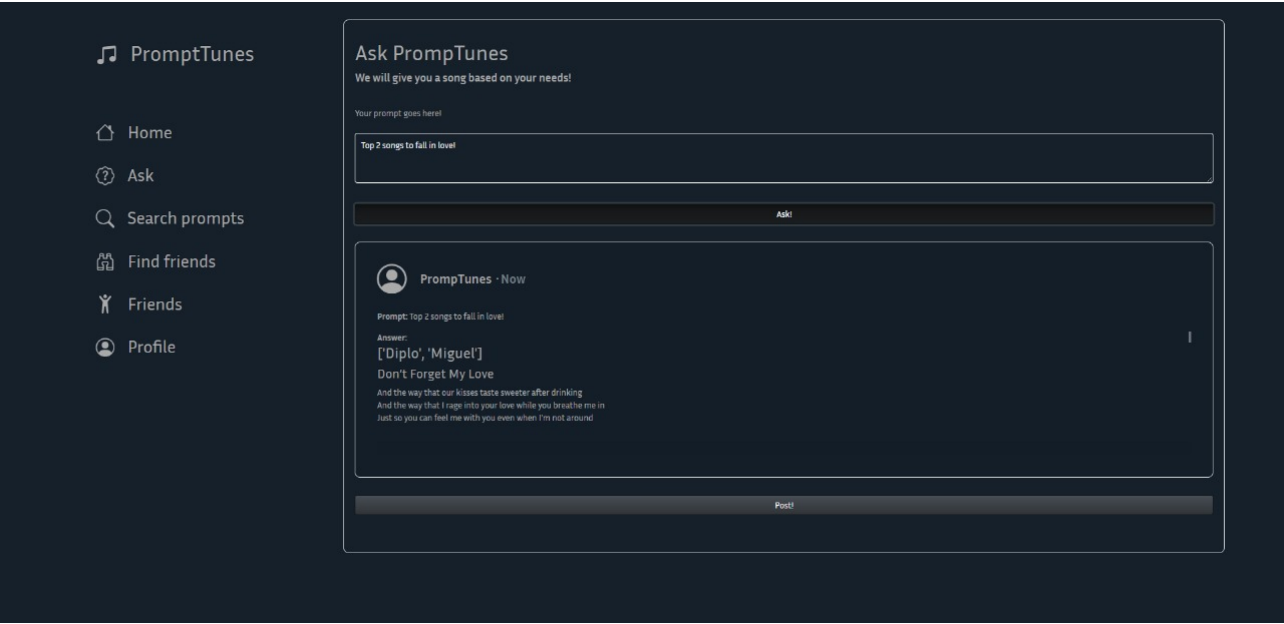
Social Feed

The social feed page aggregates posts from the user's friends and displays them in a chronological order. Users can like, comment, and share posts directly from this feed.



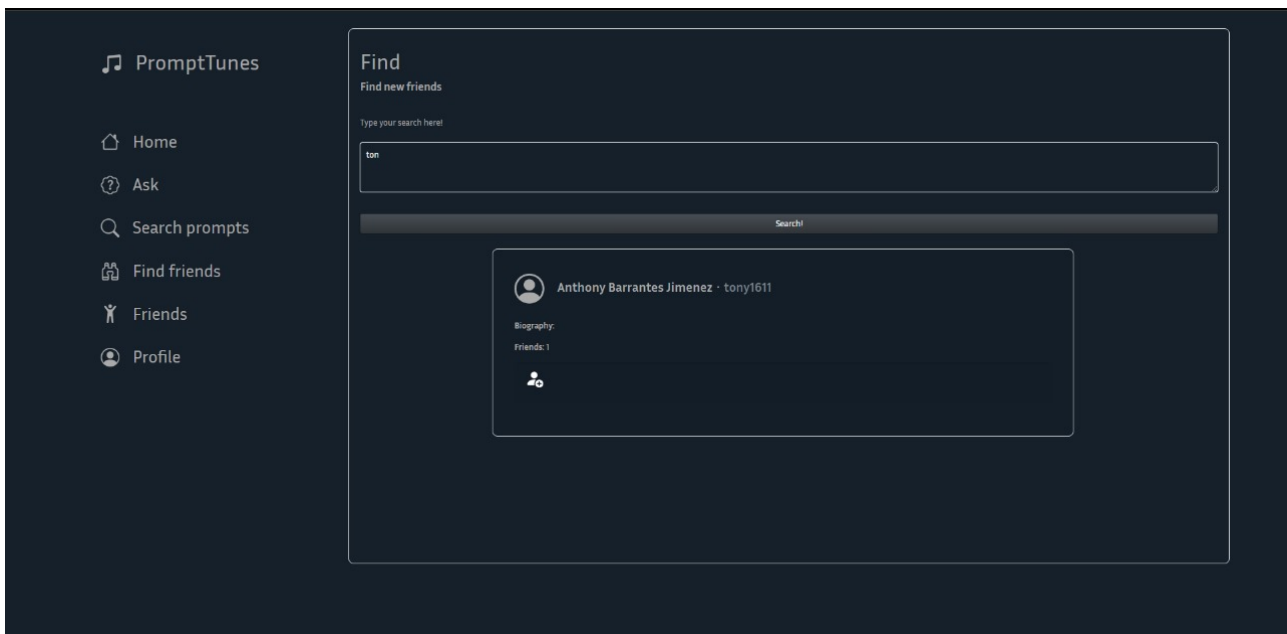
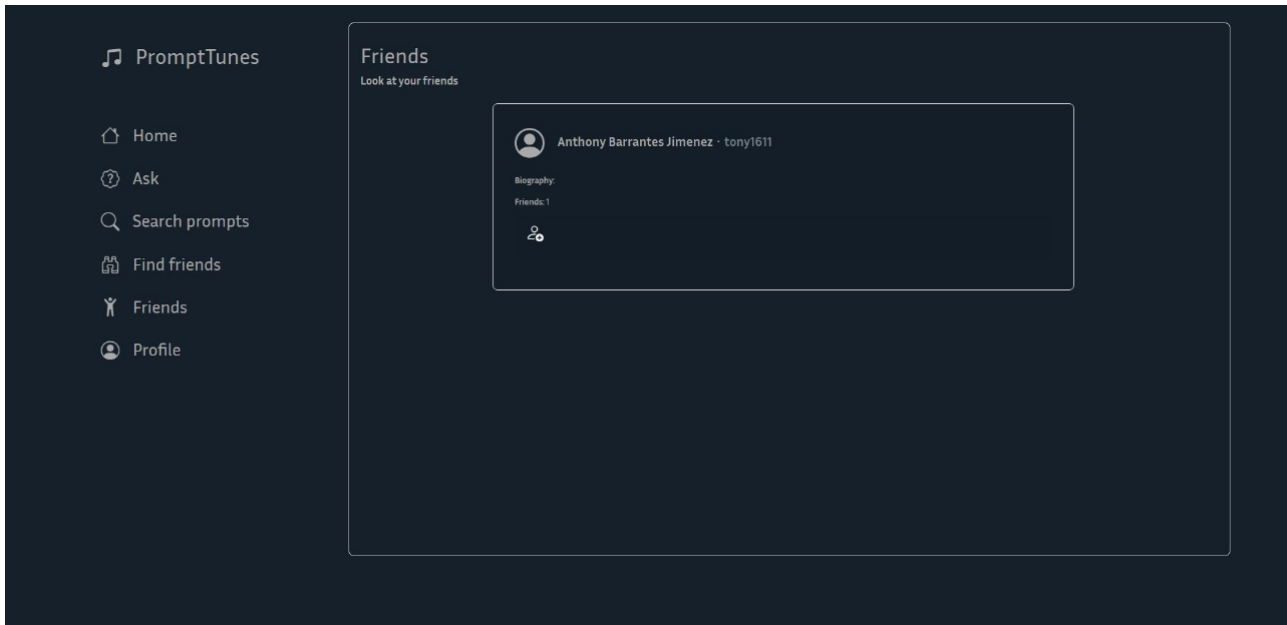
Search Page

The search page allows users to search for songs using vector search on Elasticsearch. Users can enter prompts to find songs based on lyrics or other criteria.



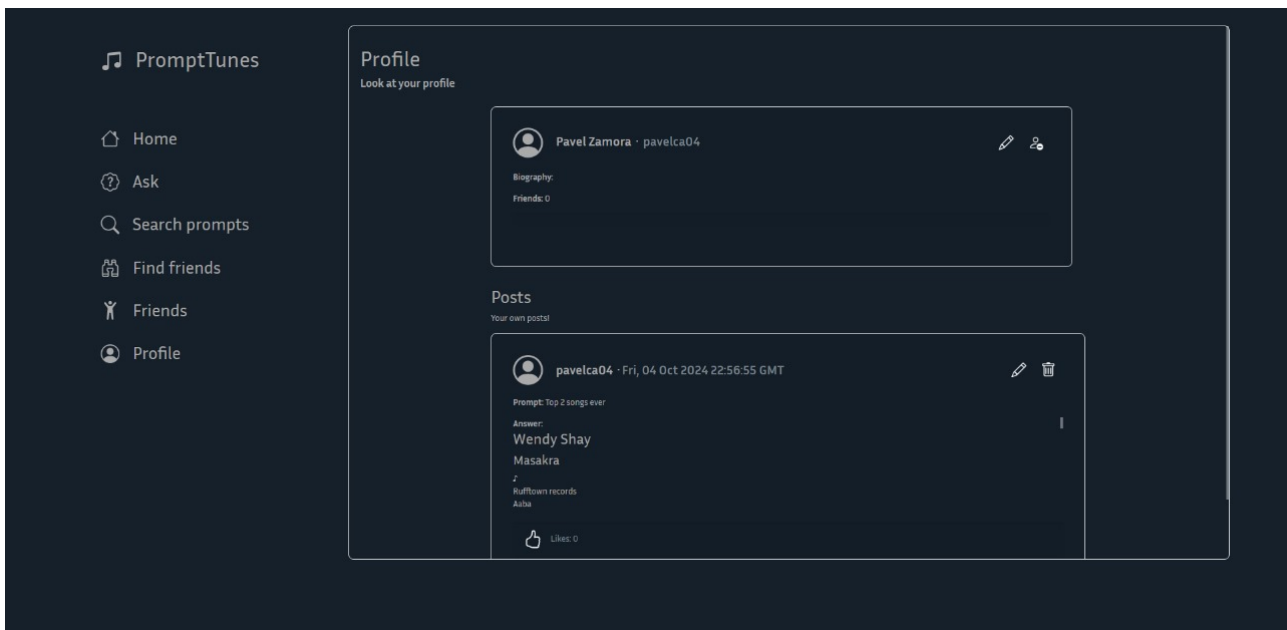
Friends

The friends page displays a list of the user's friends.



Post Creation

The post creation page allows users to create new posts or prompts. Users can enter text and submit it to the system.



Each of these pages is designed to provide a user-friendly experience, ensuring that users can easily navigate and interact with the application.

Prometheus:

Prometheus is an open-source monitoring and alerting software that is used to collect and store metrics from various components of the project. Prometheus scrapes metrics from the different components, such as databases, API, cache systems, and monitoring tools, and stores them in a time-series database. It provides a specific query language, called PromQL, to retrieve and analyze metrics, enabling users to monitor the performance and health of the system. Prometheus offers features like service discovery, multi-dimensional data model, and powerful queries, making it suitable for monitoring complex environments. In this case Prometheus is used to monitor the different components of the project and store the metrics in its database. It is configured to scrape the metrics with the goal of visualizing them in Grafana. It is configured with automatic service discovery due to the use of technologies like MariaDB which for example count with an integrated exporter that allows Prometheus to scrape the metrics of the databases automatically. In the route `TC1/charts/databases/values.yaml` you can see the configuration of the databases to be scraped by Prometheus. The `servicemonitor` resource is used to configure the scraping of the metrics of the different components of the project. As this is enabled in the Helm chart, Prometheus will automatically scrape the metrics of the different components of the project.

Grafana:

Grafana is an open-source analytics and monitoring platform that is used to visualize the metrics collected by Prometheus. Grafana provides a user-friendly interface to create dashboards and panels that display the metrics in a visually appealing way. It offers a wide range of visualization options, such as graphs, tables, and gauges, allowing users to customize the dashboards according to their needs. Grafana supports various data sources but basis in Prometheus. In this project, Grafana is used to create dashboards that display the metrics of the different components, such as databases, API and cache systems. The dashboards provide insights into the performance and health of the system, allowing

users to identify bottlenecks, anomalies, or areas for optimization. Grafana is configured to connect to Prometheus as a data source, enabling it to retrieve the metrics stored in the Prometheus database and visualize them in the dashboards. The dashboards are designed to display key performance indicators, such as HTTP request counts, query response times, cache efficiency, and resource usage, helping users monitor the system in real time and make informed decisions based on the data.

In this project, talking specifically about Grafana we can find seven main dashboards, which are:

- **MariaDB Dashboard:** This dashboard provides a comprehensive view of MariaDB's performance and health. It tracks various metrics, helping users monitor database activity, identify potential bottlenecks, and optimize overall efficiency.
- **Elasticsearch Dashboard:** Designed to give insights into Elasticsearch's health, this dashboard presents a range of performance metrics. Users can easily track the system's efficiency and address potential issues.
- **Memcached Dashboard:** Focused on the health of the Memcached system, this dashboard offers a variety of performance metrics, enabling users to monitor and enhance caching effectiveness.
- **Hugging Face API Dashboard:** This dashboard provides a detailed view of the Hugging Face API's performance, including request count and maximum, minimum, and average embedding generation time.
- **S3 Crawler Dashboard:** This dashboard tracks the performance of the S3 Crawler, displaying metrics such as total processing time and objects processed.
- **Backend API Dashboard:** This dashboard offers insights into the Backend API's performance, including cache hits, cache misses, maximum, minimum, and average request processing time, and total request count per endpoint.
- **Ingest Service Dashboard:** This dashboard provides a detailed view of the Ingest Service's performance, including object and row processing times, request counts, and error rates.

Unit Testing

Unit testing is a critical aspect of software development aimed at validating individual components of the codebase in isolation. In this project, unit tests help ensure that key functions, classes, and modules behave as expected under different conditions. By detecting bugs early in the development cycle, unit testing minimizes the risk of defects making it to production, improves code quality, and enhances maintainability. Additionally, it allows for safer refactoring and easier integration of new features.

Importance of Unit Testing

- **Reliability:** Unit tests verify that each module performs its intended function, building confidence that the system will operate as designed.
- **Bug Detection:** By isolating each part of the system, unit tests make it easier to catch bugs at an early stage, before they propagate through the codebase.
- **Refactoring Support:** Well-written unit

tests ensure that future changes or optimizations do not unintentionally break existing functionality.

- **Documentation:** Unit tests serve as a form of living documentation for the code, showing how various components are expected to behave.
- **Efficiency in Development:** Unit tests provide rapid feedback, allowing developers to fix issues before they escalate into larger, more complex problems.

What modules should have unit testing

In this project, unit testing could be applying to key modules, including:

Hugging Face API

- Verifying the proper functioning of API requests and responses.
- Ensuring that the expected model outputs are returned based on the given input data.
- Testing the error handling for API call failures, timeouts, or invalid inputs.
- Ensuring metrics collection for Prometheus (e.g., API response times, successful requests).

Ingest

- Ensuring data ingestion pipelines correctly process various data formats and sizes.
- Testing individual data transformation functions for correctness.
- Validating error handling when ingestion fails (e.g., malformed data or network errors).

S3 crawler

- Testing the ability to list files from S3 buckets.
- Ensuring graceful handling of S3-specific errors.
- Testing the processing logic that acts upon the S3 data, ensuring it behaves correctly across different scenarios.

Backend API

- Verifying the correctness of API endpoints (e.g., for creating, reading, updating, and deleting resources).
- Ensuring the integration with external services, such as Elasticsearch for indexing, and MariaDB for data persistence.
- Testing error handling, such as database failures, message queue unavailability, or malformed requests.

Why Unit Testing Was Not Implemented for Certain Modules

We decided not to implement unit testing for certain modules due to the complexity of their structure and their heavy reliance on external services. While unit testing is a critical part of maintaining code quality, there are several key reasons why these particular modules were not subjected to unit tests at this stage:

Complexity of the Modules

The modules in question, such as the Hugging Face API, the ingest, the S3 crawler, and the backend API, are highly intricate. They perform a series of complex operations that involve multiple steps of data processing, external API communication, and dynamic data transformations. Due to the depth and complexity of these operations, designing effective and meaningful unit tests would require an exhaustive setup and an in-depth understanding of the module internals, which often makes short, self-contained unit tests less feasible. For these modules, a full, dedicated testing strategy that goes beyond basic unit testing is required, including integration and system-level testing.

Dependency on External Services

Many of the core functions within these modules rely on interactions with external services such as Amazon S3, RabbitMQ, Elasticsearch, or Hugging Face APIs. Unit tests are generally meant to isolate the logic of the code from its environment, but the dependency on these services makes isolation challenging. Mocking these services for unit tests can introduce added complexity and would require comprehensive simulation of their behavior, which may not fully capture the nuances of real-world interactions. Given the tight coupling to these services, integration tests that work with real or closely simulated environments are a more practical approach to ensuring functionality.

Difficulty in Simplifying the Functions

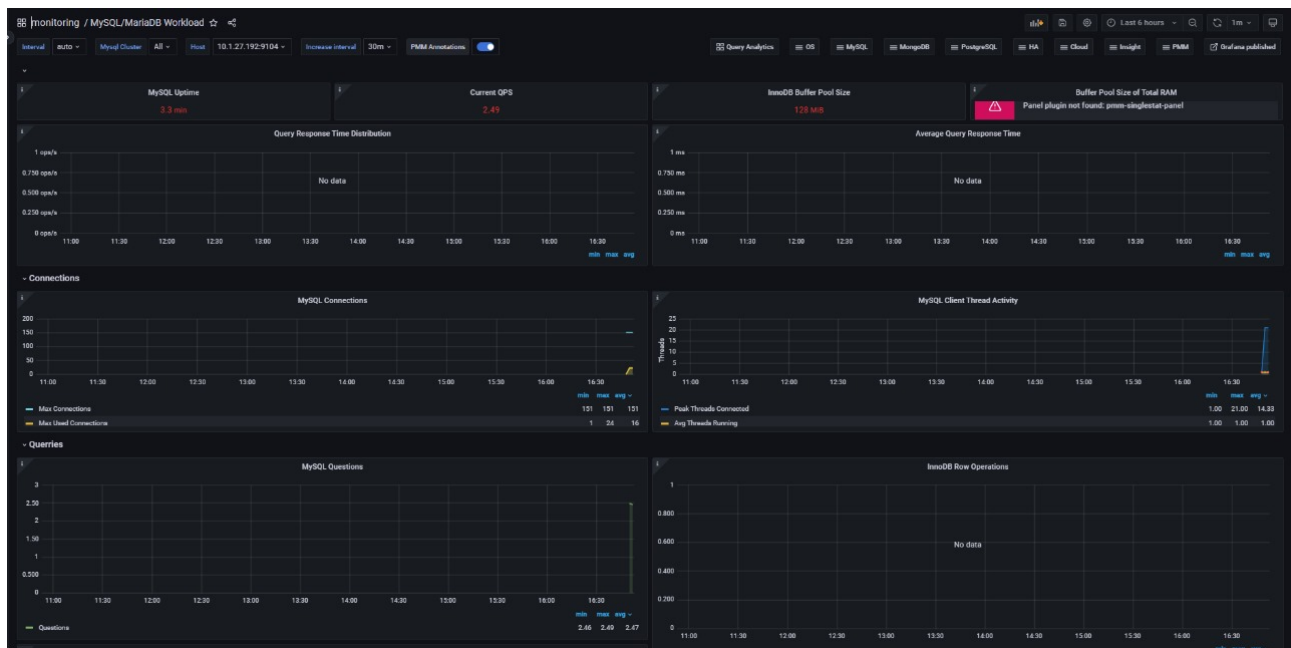
The core logic in these modules is designed to handle intricate tasks such as multi-step data processing, cross-service communication, and API responses, which do not lend themselves easily to simplification for unit testing. Breaking these down into smaller, testable units would significantly alter the code's structure and could potentially affect its performance or readability. Moreover, writing unit tests for highly complex functions would likely result in fragile tests that are tightly coupled to implementation details, reducing the maintainability of the test suite.

General Tests

The following tests were performed to evaluate the performance of the different components of the project. The tests focused on key metrics such as response times, request counts, object processing times, and cache efficiency. The results of the tests were visualized in Grafana dashboards, providing insights into the performance of the different components.

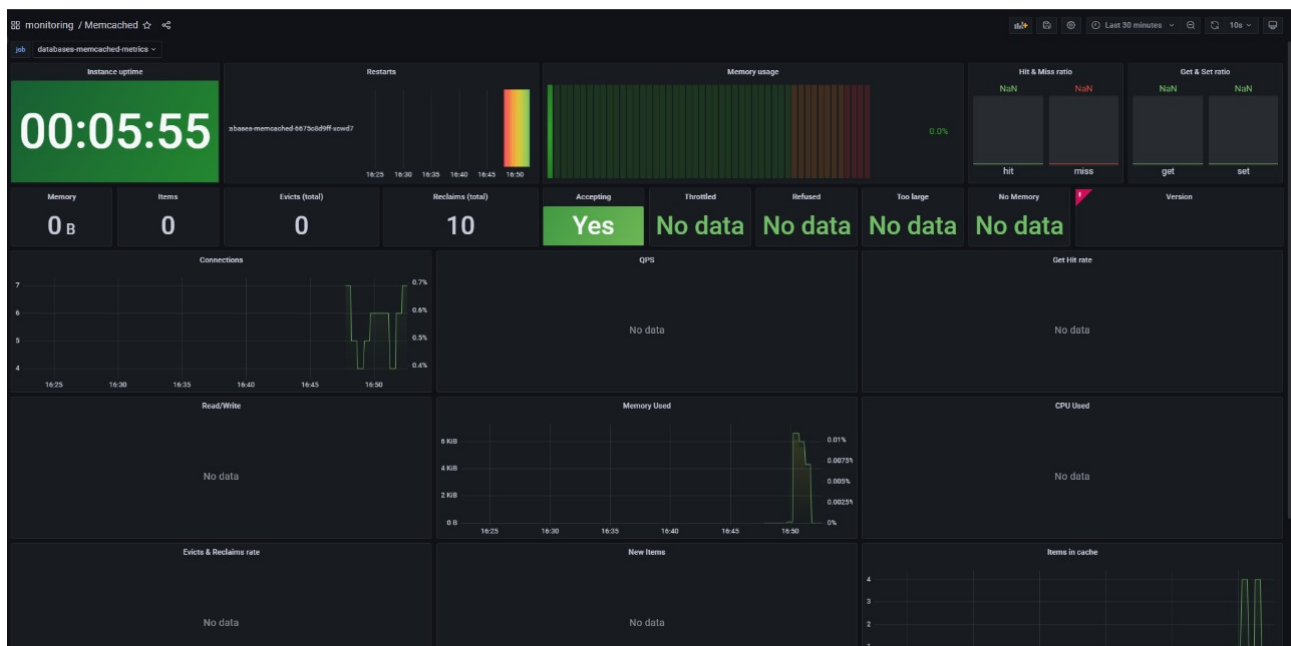
MariaDB

The MariaDB dashboard provides insights into the performance of the MariaDB database, including key metrics such as query related metrics and resource usage.



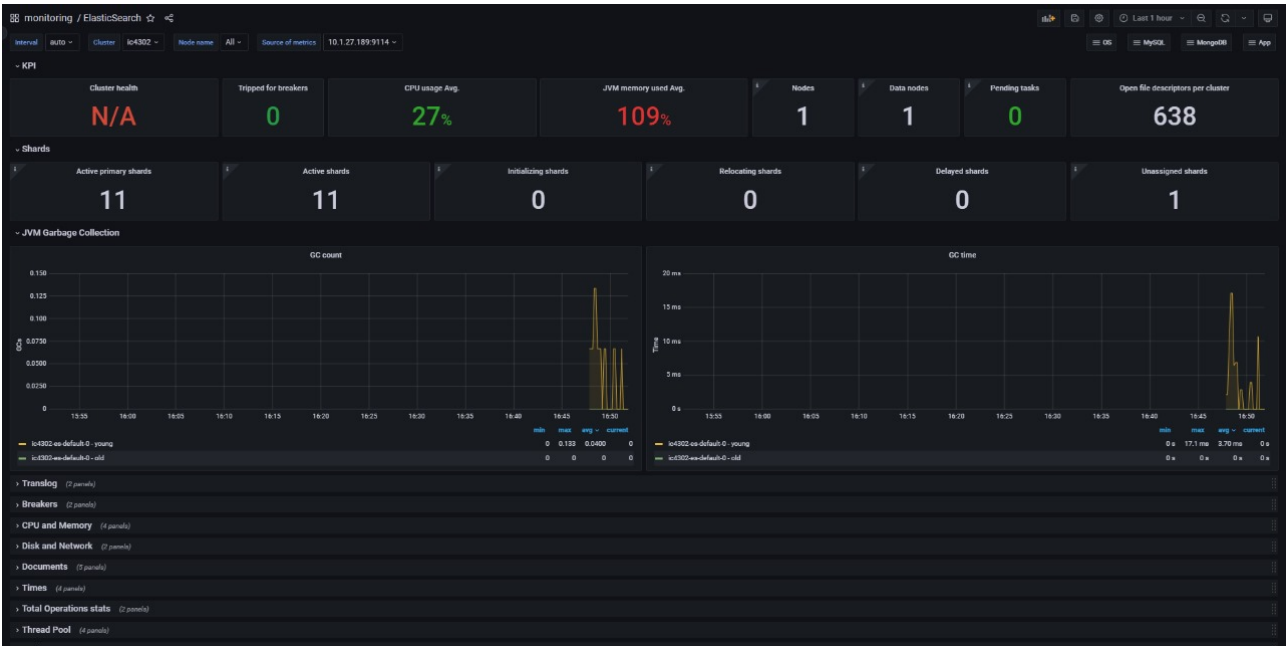
Memcached

The Memcached dashboard helps us look into the performance of the Memcached caching system. It shows us certain metrics, such as cache hits, cache misses, and overall cache efficiency.



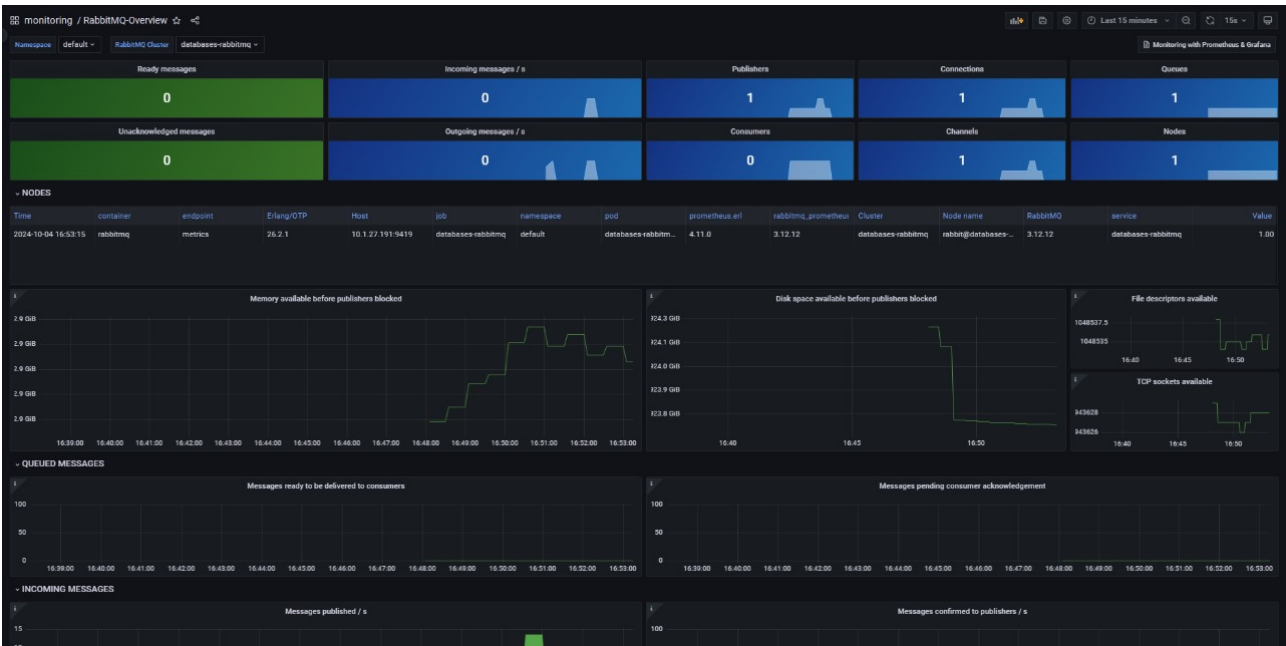
ElasticSearch

The ElasticSearch dashboard provides information about the performance of the ElasticSearch database, including certain metrics as indexing rates and resource usage.



RabbitMQ

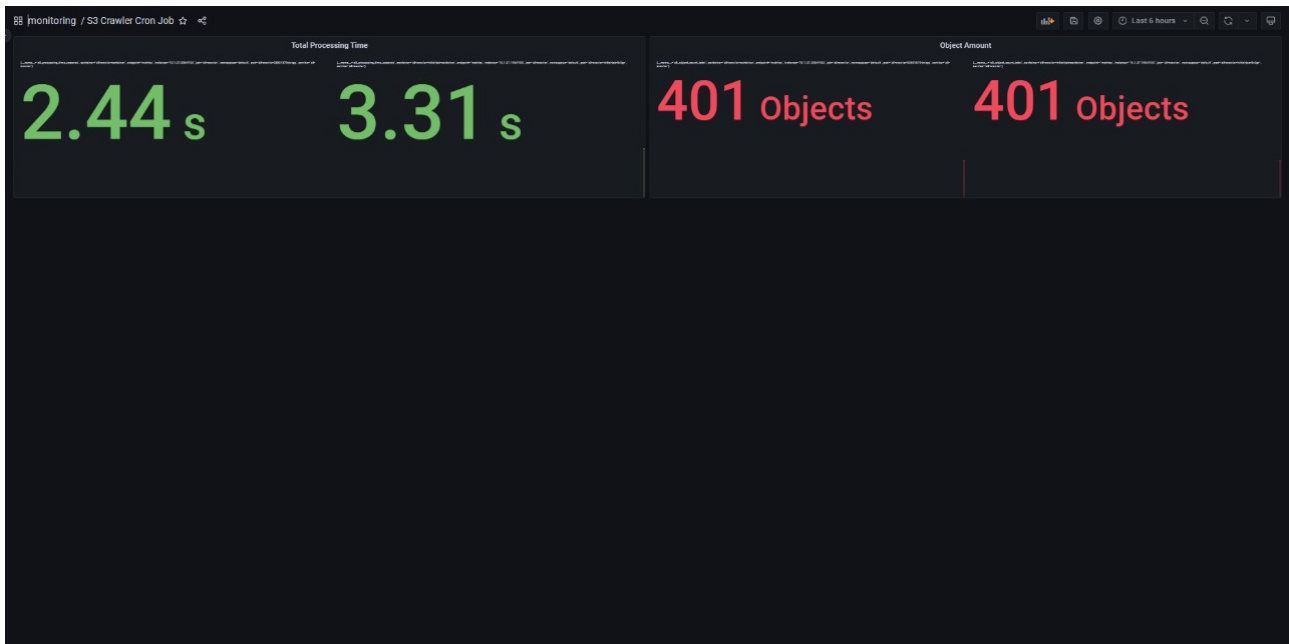
The RabbitMQ dashboard provides insights into the performance of the RabbitMQ message broker, including key metrics such as message rates, queue sizes, and resource usage.



S3 Crawler Dashboard

The S3 Crawler Dashboard provides a detailed look into the operational performance of the S3 Crawler by tracking the following metrics:

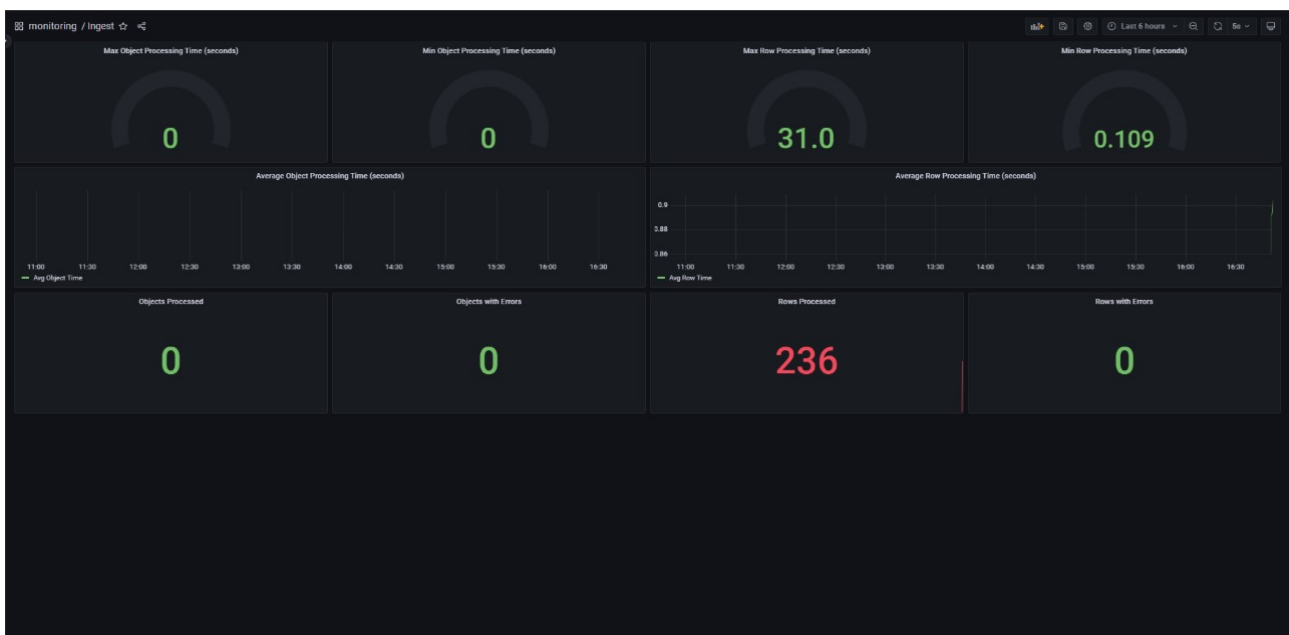
- Number of Objects: Counts the total objects processed by the crawler.
- Total Processing Time: Summarizes the total time spent processing all objects.



Ingest Dashboard

The Ingest Dashboard offers insights into how well the Ingest service is functioning, with the following metrics being monitored:

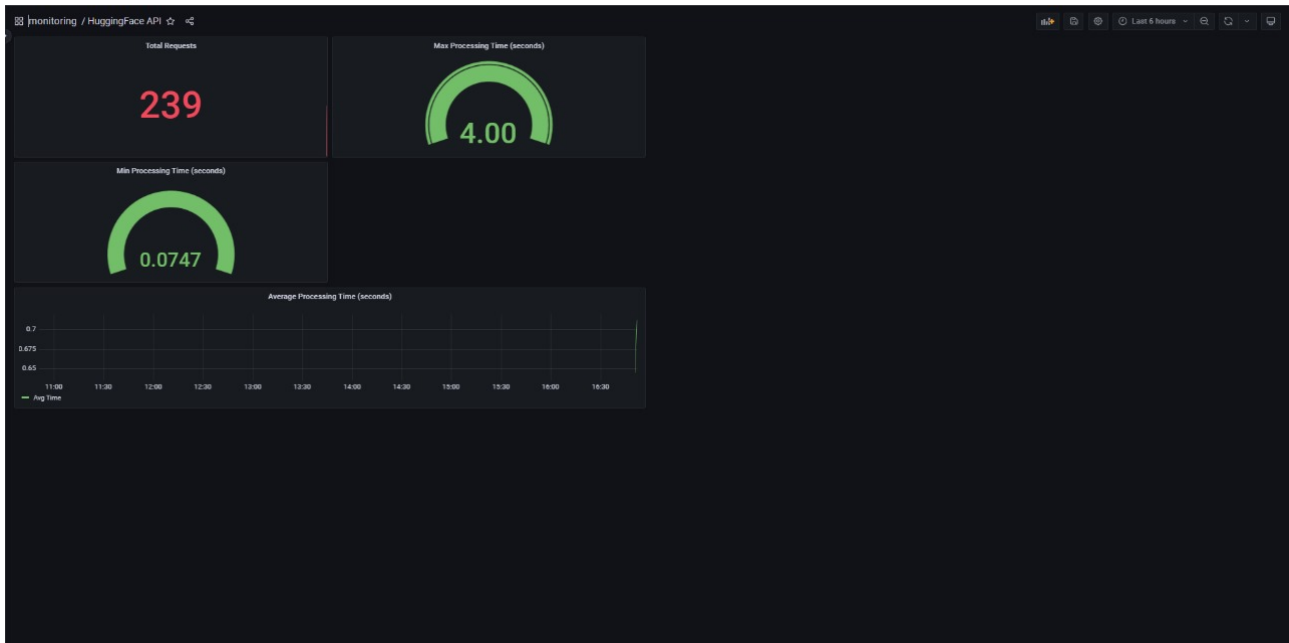
- Maximum, Minimum, and Average Processing Time for an Object: Assesses the time taken to process each object.
- Maximum, Minimum, and Average Processing Time for a Row: Evaluates the time taken to process individual rows within the objects.
- Number of Processed Objects: Tracks how many objects were successfully processed.
- Number of Processed Rows: Counts the total rows processed.
- Number of Rows with Errors: Monitors how many rows encountered errors during processing.
- Number of Objects with Errors: Tracks the total number of objects that failed to process correctly.



Hugging Face Dashboard

The Hugging Face Dashboard sheds light on the performance metrics of the Hugging Face API, including:

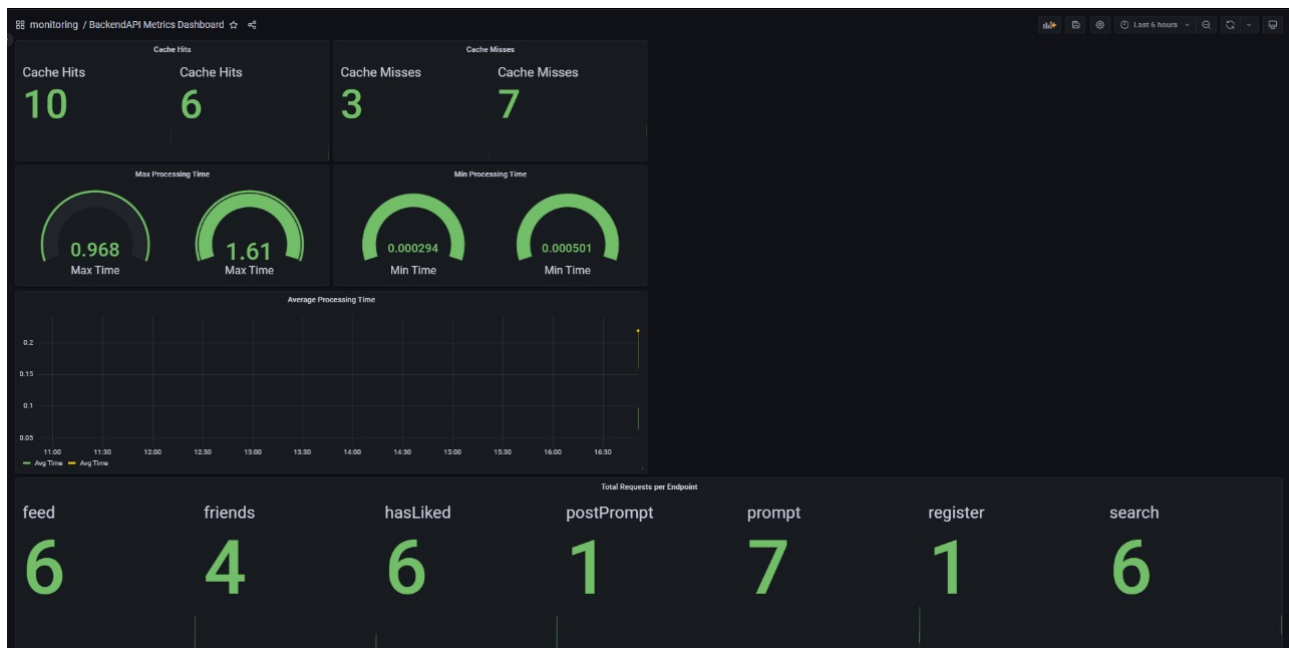
- Number of Requests: Tracks how many requests were made for generating embeddings.
- Maximum, Minimum, and Average Time for Generating Embeddings: Measures the time taken to generate embeddings, providing insights into performance and efficiency.



Backend API Dashboard

The Backend API Dashboard assesses the effectiveness of the Backend API by monitoring the following key metrics:

- Cache Hits: Counts the number of times requested data was found in the cache.
- Cache Misses: Counts the number of times requested data was not found in the cache.
- Maximum, Minimum, and Average Processing Time: Measures the performance of API requests by tracking the time taken to process each request.
- Total Requests per Endpoint: Monitors how many requests were made to each endpoint of the API.



References

- [1] "Dockerfile reference," Docker Documentation. [Online]. Available: <https://docs.docker.com/reference/dockerfile/#overview>. [Accessed: Sep. 20, 2024].
- [2] "kubectl commands," Kubernetes Documentation. [Online]. Available: <https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands>. [Accessed: Sep. 20, 2024].
- [3] "MariaDB Documentation," MariaDB Knowledge Base. [Online]. Available: <https://mariadb.com/kb/en/documentation/>. [Accessed: Sep. 20, 2024].
- [4] "Elasticsearch Documentation," Elasticsearch Documentation. [Online]. Available: <https://www.elastic.co/guide/en/elasticsearch/reference/current/index.html>. [Accessed: Sep. 20, 2024].
- [5] "Grafana Documentation," Grafana Documentation. [Online]. Available: <https://grafana.com/docs/>. [Accessed: Sep. 22, 2024].
- [6] "Custom Bootstrap Build," Bootstrap Build. [Online]. Available: <https://bootstrap.build/app/project/MjrwgQJtoYUV> [Accessed: Sep. 22, 2024].