Cloud Computing

(2023/2024)

Phase 10 - Final delivery

Group 03

Beatriz Rosa 55313, José Ricardo Ribeiro 62761,
Ayla Stehling 63327, Christopher Anaya 60566

June 5, 2024

Contents

1 Motivation and dataset characterization

This project makes use of 4 distinct datasets, whose characteristics are the following:

- Amazon Product metadata: This dataset is used on the Market Analysis microservice. Includes product metadata (descriptions, category and subcategories information, price, brand, and image features), and links of the Amazon Website Store.
- Yelp Reviews Dataset: This dataset is used on the Yelp Reviews microservice it includes cities,states, businesses information in those cities.
- Question Analysis Dataset: This dataset is used on the Question Analysis microservice and consists of the productID, questionType, answerTime, unixTime, question, answerType and answer.
- Amazon Reviews Dataset: This dataset is used in the Sentiment Analysis microservice and consists of reviews made for products on Amazon, including Product info, Reviewer info, Rating, and Text.

2 Use cases

No.	As a <role>	I want <capability>	so that <benefit>
1	As a premium user	I want to add new reviews	so that I can tell other users my opinion on products and leave feedback for the manufacturer
2	As a premium user	I want to edit reviews	so that I can add information with the longivity of the product and update accordingly
3	As a premium user	I want to delete reviews	so that I can remove the reviews which are outdated
4	As a regular user	I want to filter by productID	so that I see reviews for a specific product
5*	As a regular user	i want to filter by main category	so that i can see the most relevant categories
6*	As a regular user	I want to filter by cat. and/or brand, min/max price	so that I can find the cheapest/most expensive prod. of a brand
7*	As a regular user	I want to filter by main category	so that I can see the most popular brand on a category
8*	As a regular user	I want to filter by main category	so that I can see the best 2 products of a specific category
9	As a regular user	I want to filter by overall	so that I can choose to see only the worst reviews, or the best reviews
10	As a regular user	I want to filter by more than one aspect	so that I can combine my filters for the best search result
11	As a regular user	I want pre-defined search queries	so that I have quick access to the most common search filter combinations
12*	As a regular user	I want to filter by product	so that I can see the questions asked by clients for my product
13*	As a regular user	I want to get the time of the answer	so that I can see how relevant a question and answer is
14*	As a regular user	I want to get the types of question and answers	so that I can analyse the sorts of answers given

Table 1: User Stories

* - implemented

3 REST API

No.	Role	Path	GET	POST	PUT	DEL	Description
1	Premium User	/yelp/cities	X				Allows retrieval of cities
2	Premium User	/yelp/states	X				Allows retrieval of states
3	Regular User	/yelp/businesses	X				Allows retrieval of businesses
4	Regular User	/yelp/businesses?business_id={business_id}	X				Allows retrieval of a specific business
5	Regular User	/yelp/businesses/top-rated	X				Allows users to to see the top rated businesses
6	Regular User	/market/categories	X				Allows users to filter the main categories of products
7**	Regular User	/market/metadata?category={category}	X				Allows users to filter products by main category,brand,min.price and max. price
8	Regular User	/market/top-brand?category={category}	X				Allows users to see the top performing brand in sales in a specific category
9	Regular User	/market/top-products?category={category}	X				Allows users to see the top performing products in sales in a specific category
10	Regular User	/qa/get_string_answer		X			Allows users to see the answer to the question asked
11	Regular User	/qa/get_product_product_answer		X			Allows users to get answers to a product
12	Regular User	/qa/get_answer_type		X			Allows users to get the answer type to a given answer
13	Regular User	/qa/get_question_type		X			Allows users to get the question type to a given question
14	Regular User	/sentiment/get_review_by_key	X				Allows retrieval of review by primary key
15	Regular User	/sentiment/get_reviews_by_user_id	X				Allows retrieval of reviews by user_id
16	Regular User	/sentiment/get_reviews_by_asin	X				Allows retrieval of reviews by product asin

Table 2: API table

** and optionally:
/market/metadata?category={category}&brand={brand}&minPrice={minPrice}&maxPrice={maxPrice}

4 Application Architecture

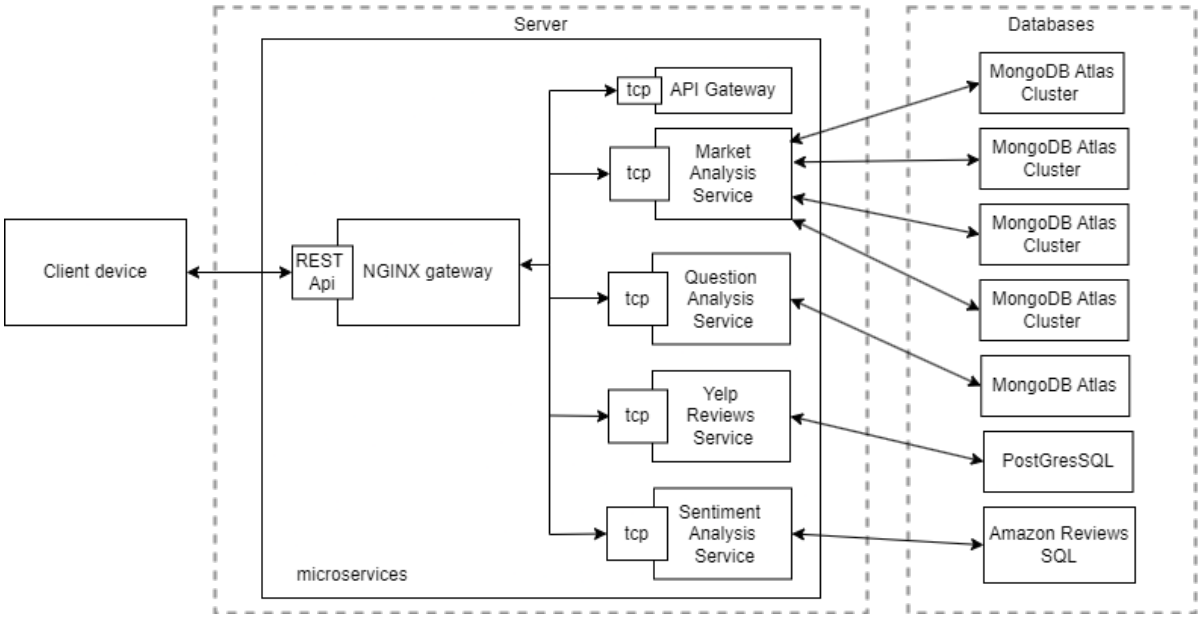

Figure 1: Application Architecture diagram

5 Technical Architecture

The application is hosted on Google Cloud Platform (GCP), having a microservices architecture with containers orchestrated by Docker and managed through Docker Compose. The architecture is designed to meet the specified non-functional requirements as follows:

Google Cloud Platform (GCP): Serves as the cloud infrastructure provider, offering a suite of services that support high availability, security, and scalability. GCP's managed service Google Kubernetes Engine (GKE) has these qualities and is be used in this project order to allow orchestration of the containers at scale.

Containers (Docker): Each microservice, namely the nginx gateway and the market analysis, yelp reviews, question analysis and sentiment analysis, is containerized using Docker. This supports scalability by allowing each microservice to be scaled independently based on demand and maintainability by simplifying updates and deployments.

Microservices Architecture: Decomposing the application into microservices (NGINX Gateway and Market Analysis, yelp reviews, question analysis and sentiment analysis) ensures scalability and availability by allowing each microservice to scale independently and remain isolated from failures in other services.

NGINX Gateway: Acts as the single entry point for all client requests, routing them to the appropriate microservice.

TCP: This protocol is used by the microservices for communication.

MongoDB Atlas & PostgreSQL: MongoDB Atlas, a fully-managed cloud database, is used for storing Amazon Products metadata across clusters for high availability and scalability. Question Analysis Microeservice also makes use of mongoDB Atlas database for the same reasons. The Yelp Reviews microservice uses PostgreSQL for storing review data efficiently.

Visualization and Monitoring Tools: Tools like MongoDB Compass, PostgreSQL, and the Docker extension for VSCode were used to monitor the databases and containers, supporting the monitoring and diagnostics requirement by providing insights into the system's state and performance.

Building, deployment and running scripts:

- `gcp_clean`: Deletes clusters in GCP and GCR (Google Container Registry) along with any Kubernetes allocated resources. It ensures that the environment is cleaned up, preventing unnecessary charges and maintaining a tidy workspace;
- `cluster_create`: This script enables necessary GCP services, creates the Kubernetes cluster and sets up the required cluster role bindings and secrets, ensuring that the infrastructure is correctly configured for deploying the microservices;

- `gcp_configure`: Creates the service account and secrets;
- `gcr_publish`: Builds and pushes docker images;
- `gcr_apply_kubernetes`: Applies deployment configurations.
- `test`: This script automates the testing process by updating the test files with the external IP of the nginx-gateway-svc and running unit tests, acceptance tests, and security tests. It ensures the application is working as expected and meets the required standards.

Load Balancing: Horizontal Pod Autoscaler (HPA) is used to automatically scale the number of pod replicas based on observed CPU utilization or other select metrics. This ensures the application can handle varying loads efficiently.

Automation: A CI/CD pipeline is set up using tools like GitHub Actions to automate the process of building, testing, and deploying the application. This ensures that changes can be integrated and deployed rapidly and reliably.

Application Tests:

The testing framework includes:

- Unit tests to validate individual components;
- Acceptance tests to ensure the system meets business requirements;
- Load tests using Locust to simulate user load and measure system performance under stress;
- Security tests using Bandit tool that generates a report on the code analysed.

Monitoring: Prometheus is used to collect metrics from the application, and Grafana is used to visualize these metrics which allows for real-time monitoring of the system's health and performance, helping in quick detection and resolution of issues.

6 Implementation

The implementation phase involved setting up the infrastructure on Google Cloud Platform (GCP), containerizing the microservices using Docker, and deploying them on Google Kubernetes Engine (GKE). The microservices include the NGINX gateway, market analysis, yelp reviews, question analysis, and sentiment analysis, each serving a specific purpose in the application.

7 Deployment

Deployment was automated using a CI/CD pipeline. The pipeline is configured to automatically build Docker images for each microservice, push them to Google Container Registry (GCR), and deploy them to GKE.

The deployment process also included setting up the necessary Kubernetes resources like services, deployments, and secrets, as well as configuring Horizontal Pod Autoscaler (HPA) to manage load balancing.

8 Test and Evaluation techniques and results

8.1 Operational Support

8.1.1 Automation

Deployment and CI/CD pipelines are fully automated using GitHub Actions.

8.1.2 Unit tests

Unit Tests were performed on the microservice Market Analysis using framework pytest for Python. these test aimed to test requests made to these microservices with unexpected inputs to measure how well the microservice was prepared to handle these situations such as forgetting an argument on a request or providing a category that does not exist. Overall it performed well in these edge situations.

Market Analysis Microservice Unit tests:

```python
def test_get_top_brand_invalid_category():
    response = requests.get(f"{BASE_URL}/market/top-brand?category=InvalidCategory")
    assert response.status_code == 404
    assert "error" in response.json()
```

Figure 2: Test that should return error with code 404

```
============================= test session starts ==============================
platform linux -- Python 3.10.12, pytest-8.2.1, pluggy-1.5.0
rootdir: /home/beatrizarosa4/cn-group03/testing
plugins: anyio-4.4.0
collected 10 items

test_market_analysis.py ..........                                       [100%]

============================== 10 passed in 8.28s ==============================
```

Figure 3: Unit tests results

8.2 Security

- Static Code Analysis: The tool Bandit was used for Python security analysis.

  Microservice market analysis

Metrics:
Total lines of code: 177
Total lines skipped (#nosec): 0

flask_debug_true: A Flask app appears to be run with debug=True, which exposes the Werkzeug debugger and allows the execution of arbitrary code.
Test ID: B201
Severity: HIGH
Confidence: MEDIUM
CWE: CWE-94
File: ../app/microservices/market-analysis/market-analysis.py
Line number: 243
More info: https://bandit.readthedocs.io/en/1.7.8/plugins/b201_flask_debug_true.html

```
242     if __name__ == '__main__':
243         app.run(debug=True, host='0.0.0.0', port=50053)
```

hardcoded_bind_all_interfaces: Possible binding to all interfaces.
Test ID: B104
Severity: MEDIUM
Confidence: MEDIUM
CWE: CWE-605
File: ../app/microservices/market-analysis/market-analysis.py
Line number: 243
More info: https://bandit.readthedocs.io/en/1.7.8/plugins/b104_hardcoded_bind_all_interfaces.html

```
242     if __name__ == '__main__':
243         app.run(debug=True, host='0.0.0.0', port=50053)
```

Figure 4: Report generated by Bandit tool after analyzing the market-analysis microservice directory

Bandit Security Analysis Results:

The Bandit analysis of the market-analysis microservice identified a high-severity issue where the Flask application is running with `debug=True`. Running a Flask application in debug mode exposes the Werkzeug debugger, which allows the execution of arbitrary code and poses a significant security risk in a production environment.

Additionally, a medium-severity issue was found where the Flask application is binding to all network interfaces using `0.0.0.0`. Binding a web application to all interfaces makes it accessible from any network, which can expose the application to unwanted access and potential attacks.

8.3 Reliability

8.3.1 Load Testing

The following analysis is based on the data from the Locust load testing tool, where microservices endpoints are tested to measure their performance under a specific load.

Market Analysis Microservice Reliability tests

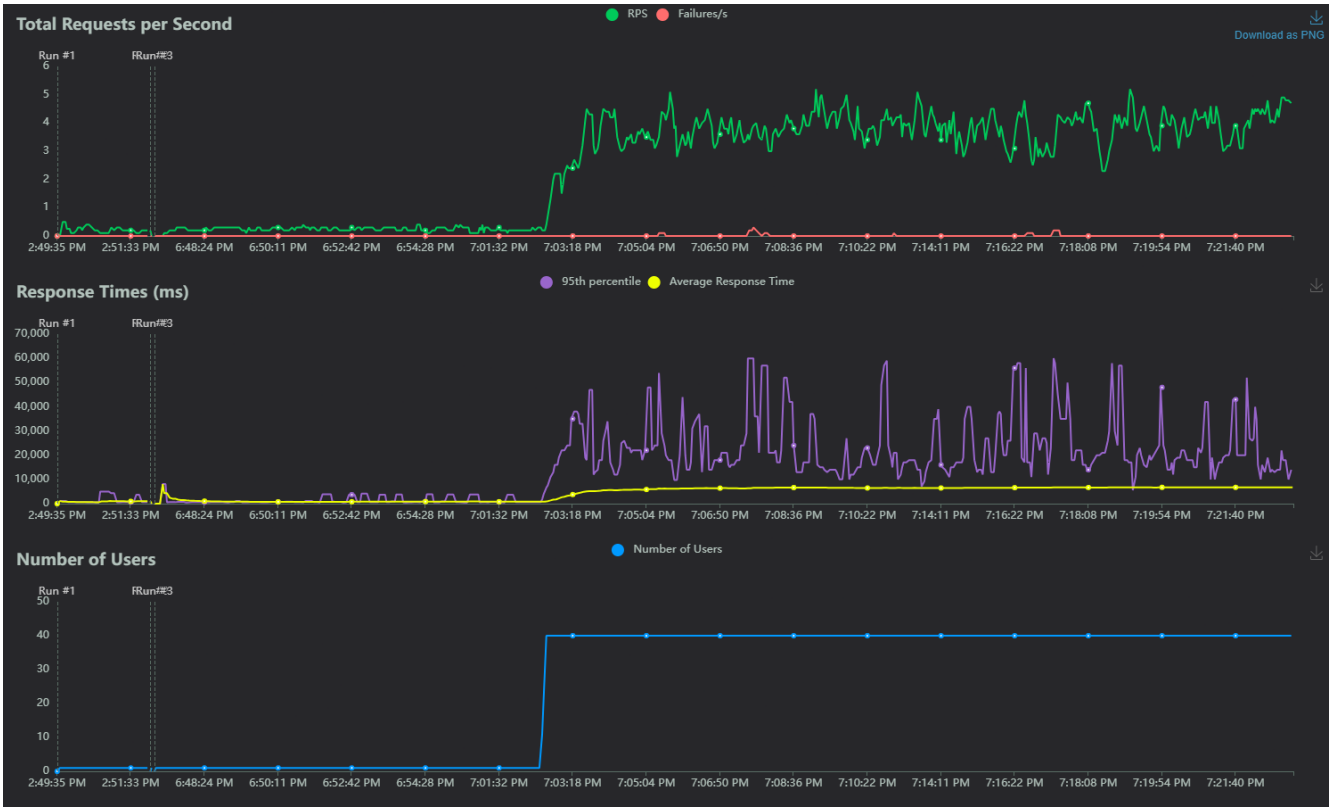

Figure 5: Locust Report Graphs for users using Market Analysis Microservice

Analysis of Locust Test Results:
In Figure 5 are the graph results computed by Locust, that was configured to run with 40 users and a ramp-up rate of 5 users per second. The test started at 7:02 PM.

- Total Requests per Second (RPS)
  - The RPS graph shows that the number of requests per second steadily increased after the test began at 7:02 PM. The peak RPS reached approximately 5.2. There were minimal failures throughout the test, indicated by the red markers. Overall the system handled the increasing load fairly well, maintaining a consistent RPS with very few failures.
- Failures: The only endpoint with recorded failures was `/market/categories`, with a total of 9 failures. The error message indicates a `504 Server Error: Gateway Time-out`, which the cause can be related to the fact that this endpoint occasionally had issues handling the load, resulting in timeout errors.

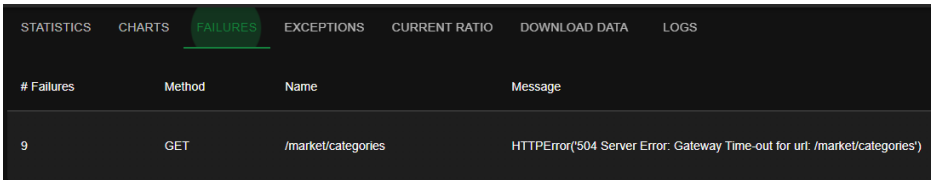

Figure 6: Failures observed by Locust during test time

- Response Times (ms)
  - The average response time (yellow line) remained relatively stable, with occasional spikes. The 95th percentile (purple line) shows some variability, indicating that some requests took significantly longer than the average, which reveals that there are occasional performance bottlenecks/issues.
- Number of Users

– The number of users increased as per the ramp-up configuration, reaching 40 users quickly and then maintaining this level throughout the test. The system was able to sustain the load of 40 concurrent users as configured without significant issues.

8.4 Performance efficiency

Prometheus and Grafana were integrated for monitoring the application. Prometheus collected metrics from the application, and Grafana provides dashboards for visualizing these metrics.

- **Prometheus:** Prometheus was configured to scrape metrics from the application and Kubernetes nodes and pods. The Prometheus configuration included various scrape jobs such as scraping metrics from the Prometheus server itself, from the Pushgateway, and from Kubernetes nodes and pods. It also specifically scraped metrics from the market analysis microservice.
- **Pushgateway:** Pushgateway was used to push custom metrics from the market analysis microservice. Metrics such as request latency, request count, and error count were collected and pushed to the Pushgateway. This allows Prometheus to scrape these metrics from the Pushgateway, providing a reliable way to collect metrics from short-lived jobs.

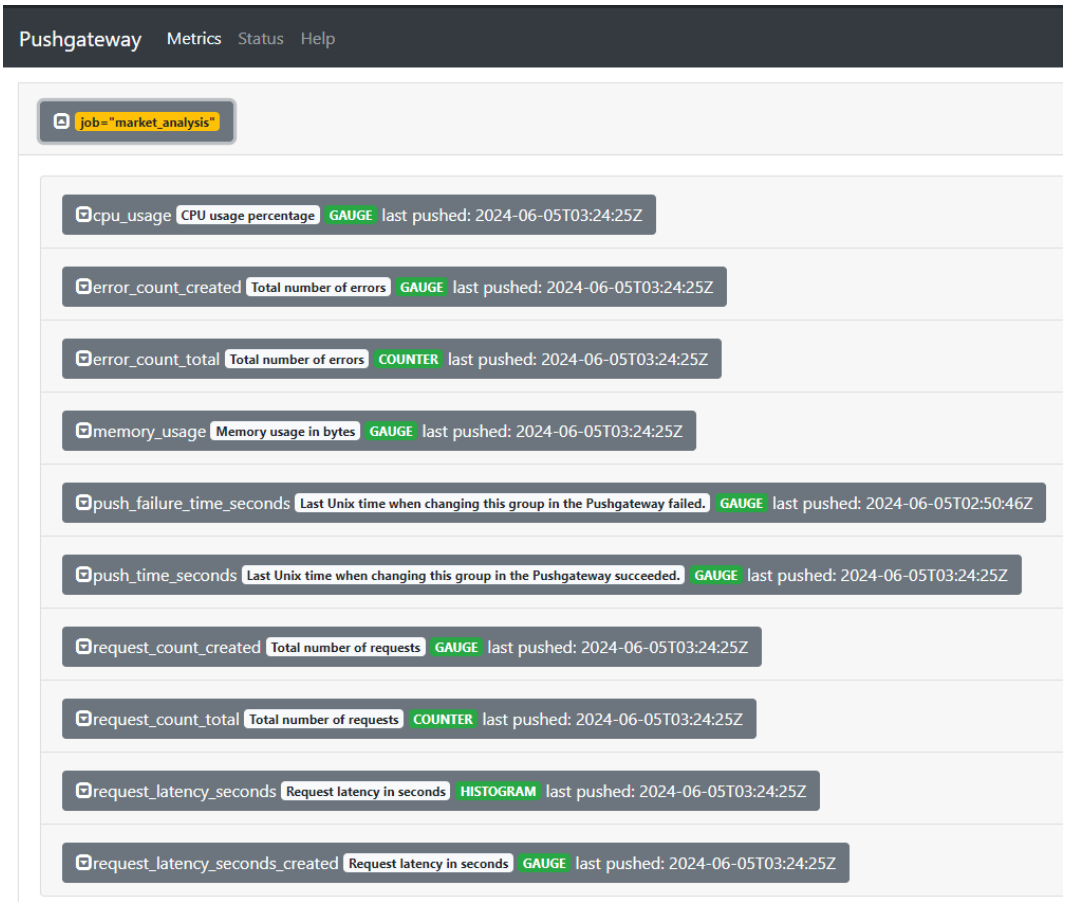

Figure 7: Metrics pushed to job:market analysis

- **Grafana:** Grafana was used to visualize the metrics collected by Prometheus. Grafana dashboards were created to monitor the application's performance, including metrics such as CPU usage, memory usage, request latency, request count, and error count. This visualization helps in understanding the system's performance and identifying potential issues.

  Monitoring scenarios provoked by Locust:

  - 100 Users; Ramp up 10 seconds

  - 150 Users; Ramp up 10 seconds

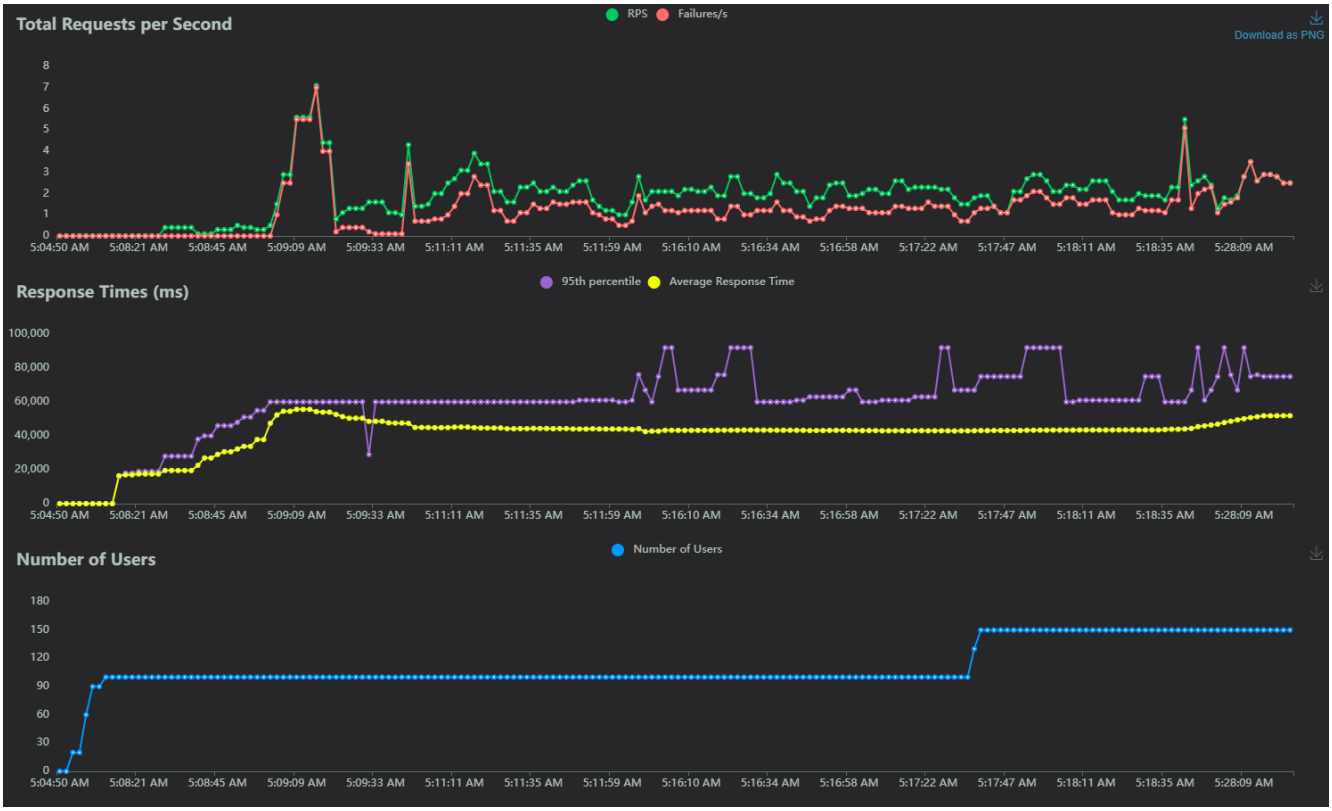

Figure 8: Locust load created for Prometheus metrics to measure

- **Metrics Monitored:** The following metrics were monitored to assess the system's performance:
  - **Request Latency:** Measured to understand the time taken to process requests, which helps in identifying performance bottlenecks.

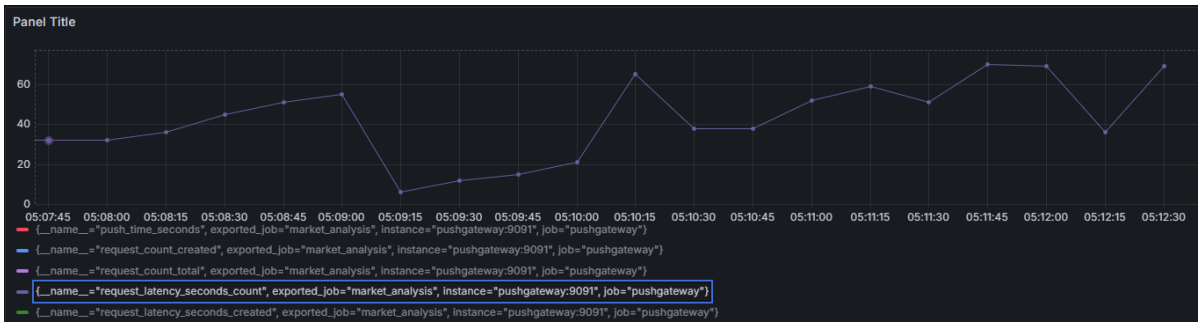

Figure 9: request_latency_seconds_count with 100 Users ; Ramp up 10 seconds

Figure 10: request_latency_seconds_count with 150 Users ; Ramp up 10 seconds

The Grafana metrics reveal that under a load of 100 users, the system maintains a relatively stable performance with minor fluctuations in request latency, averaging around the mid-40s to 60 range. However, increasing the load to 150 users results in a substantial rise in latency, with counts peaking around 350 and showing more pronounced variability. This indicates that the system experiences significant strain under heavier loads, highlighting potential bottlenecks or limitations in resource allocation. The increased latency suggests the need for optimization in database queries, resource limits, and request handling mechanisms to enhance scalability and performance efficiency.

– **Request Count:** Tracked to monitor the total number of requests received by the application.

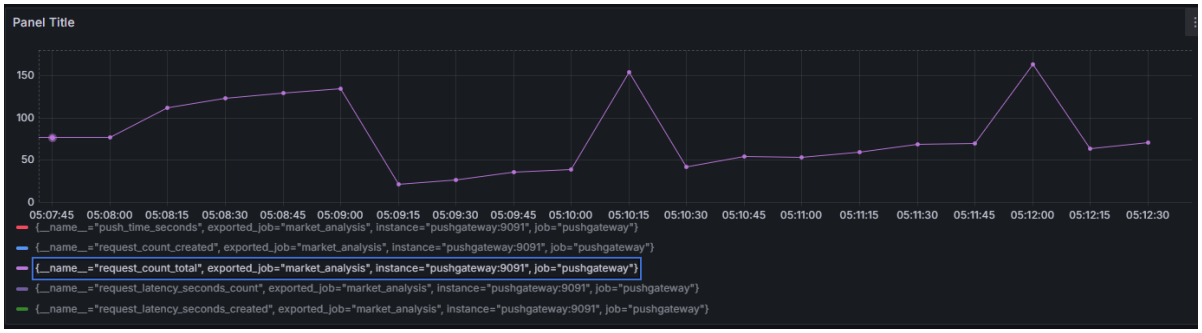

Figure 11: request_count_total with 100 Users ; Ramp up 10 seconds

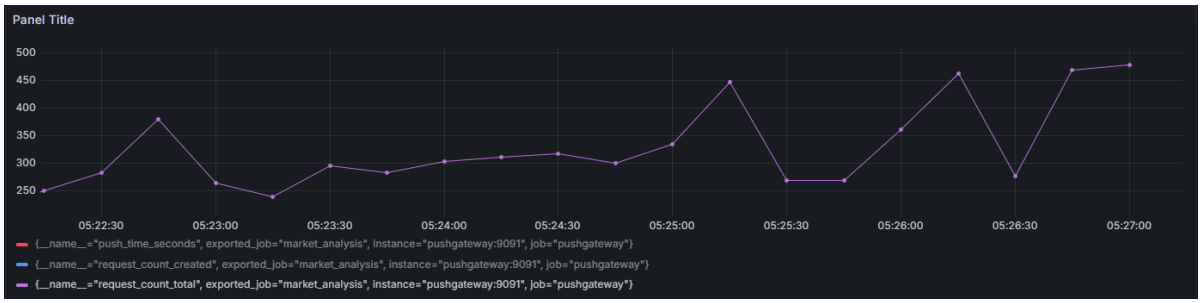

Figure 12: request_count_total with 150 Users ; Ramp up 10 seconds

In the Grafana query results for the "request_count_total" metric, a notable increase in the total number of requests can be observed when the load was increased from 100 users to 150 users in Locust. Initially, with 100 users, the request count total demonstrates a relatively steady pattern with occasional spikes, indicating consistent handling of incoming requests. However, upon increasing the load to 150 users, the graph reflects a substantial rise in the number of requests processed, with more pronounced spikes and higher peaks. This indicates that the system was able to handle the increased load by processing a significantly greater number of requests, showcasing the system's scalability and the effectiveness of the monitoring setup in capturing and visualizing performance metrics under varying loads.

– **Error Count:** Monitored to keep track of the total number of errors occurring in the application.

Figure 13: error_count_total with 100 Users ; Ramp up 10 seconds

Figure 14: error_count_total with 150 Users ; Ramp up 10 seconds

When analyzing the error count total metric as the load increased from 100 users to 150 users, we observed notable changes in the system's behavior. Initially, with 100 users, the error count remained relatively low and stable, indicating that the system was handling the load without significant issues. However, as the load increased to 150 users, the error count showed significant spikes, suggesting that the system began to experience more frequent errors under the higher load.

– **Push Time Seconds:** Measured the time taken to push metrics to the Pushgateway.

Figure 15: push_time_seconds with 100 Users; Ramp up 10 seconds

Figure 16: push_time_seconds with 150 Users; Ramp up 10 seconds

When analyzing the results of the Grafana query for the metric push_time_seconds, it is evident that the push time steadily increased as the load increased from 100 users to 150 users in Locust. This linear growth in push time indicates that as the system experienced a higher load, the time taken to push metrics to the Pushgateway also increased proportionately.