https://github.com/egeoztass/CS436-Project.git

Nisa Erdal 28943
Ege Öztaş 28828
Furkan Yıldırım 35145
Doruk Benli 29182

1. Architecture Design

The architecture of this project comprises several key components, including Cloud Run, Compute Engine VMs, Cloud Functions, and Cloud Storage (Figure 1.1). The application is a social media platform, and this architecture is designed to handle its specific requirements, including user interactions, image storage, and data processing.

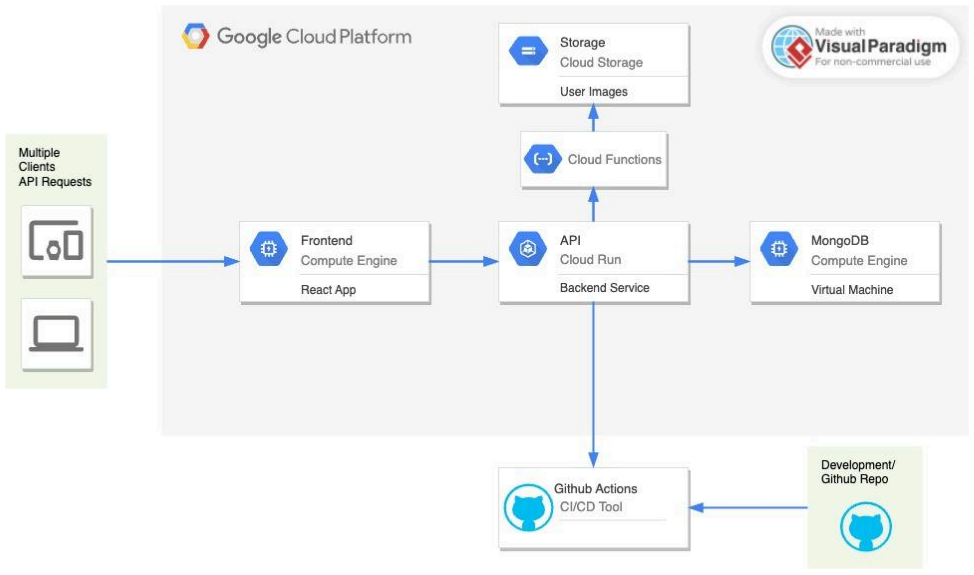


Figure 1.1: Cloud Architecture

Cloud Run is utilized for deploying and running the backend services in a serverless environment. Cloud Run's ability to automatically scale up or down based on traffic ensures efficient resource utilization and cost management. The backend application, containerized for deployment, handles incoming API requests, processes data, and communicates with the database.

For hosting the database and frontend, separate **Compute Engine VMs** are used. The database VM hosts the MongoDB database. The frontend VM serves the user interface of the social media application. Deployed with appropriate web server configurations, it handles HTTP requests and delivers a responsive interface for the users to interact with the platform.

A **Cloud Function**, written in Python FastAPI, is employed to automate the process of transferring images to Cloud Storage. Triggered by events such as new image uploads or updates, this function processes the images and transfers them to Cloud Storage. This automation ensures that images are efficiently managed and stored.

Cloud Storage serves as the scalable and cost-effective solution for storing images. It is used to store profile pictures and the posts that users upload. Images are stored in buckets, ensuring secure storage and easy retrieval when needed.

The interaction flow within this architecture begins with user requests. Users interact with the frontend application hosted on the frontend VM, which makes API calls to the backend service on Cloud Run. The backend service processes these requests, interacts with the database VM for data retrieval or storage, and performs the necessary business logic. When images are uploaded or modified, the backend triggers the cloud function. The cloud function processes the images and stores them in Cloud Storage. Metadata related to images and other application data is stored in the database on the database VM.

This architecture also incorporates a Continuous Integration and Continuous Deployment **(CI/CD)** pipeline to streamline and automate the deployment process for the social media application. Utilizing tools such as GitHub Actions and Google Cloud Build, the CI/CD pipeline automates the stages of code integration and deployment. When changes are pushed to the repository, the pipeline automatically triggers a series of actions, including setting up MongoDB and frontend VMs, building Docker images, deploying these images to Cloud Run, initializing Cloud Storage and deploying Cloud Functions.

Terraform and **Bash Scripting** is used in this project. By giving the necessary rules and roles, terraform can automatically build up and run the project on the given Google Cloud project.

CI/CD and IaaS tools build up the MongoDB VM, publish serverless functions, publish the project container to Artifact Registry and run it on Google Cloud Run.

2. Experiment Design

The experiments focus on load stress testing to evaluate the system's performance under various conditions. The primary objective is to determine the application's scalability, reliability, and response times when subjected to increasing levels of user traffic. To design these experiments, different scenarios representing real-world usage patterns will be simulated. This includes gradually increasing the number of simultaneous users, varying the frequency of user actions such as posting images or fetching data, and monitoring the system's behavior under peak load conditions. By doing so, the aim is to identify the potential bottlenecks, measure resource utilization, and ensure the application can handle the expected user load efficiently.

For effective stress testing, first, there will be a baseline testing to establish a performance benchmark under normal operating conditions, understanding the system's standard response times, throughput, and resource utilization. Incremental load testing will follow, gradually increasing the number of concurrent users to observe how the system manages increasing loads and identifying the threshold at which performance degrades. Peak load testing will simulate maximum expected traffic and beyond to assess performance during high-traffic events and pinpoint critical failure points. Key metrics such as CPU and memory usage, response times, database performance, and network latency will be monitored. Post-testing, the results will be analyzed to identify performance bottlenecks and scalability issues, guiding us in optimizing system parameters, code, and resource scaling.

3. System Parameters

These system parameters will be monitored and configured to ensure optimal performance and resilience of the application. These parameters include the specifications and configurations of the various components within our architecture. The initial configuration of these parameters are as follows:

Compute Engine VM for Database:

CPU: Configured with 2 vCPUs.
RAM: Allocated 4 GB of memory.
Storage: 10 GB balanced persistent disk

Compute Engine VM for Frontend:

CPU: Configured with 2 vCPUs.
RAM: Allocated 4 GB of memory.
Storage: 10 GB balanced persistent disk
Network: Ensured high-speed network connectivity for seamless interaction with the backend services and users.

Cloud Run for Backend Services:

Minimum Instances: Configured to maintain a minimum of 0 instances.
Maximum Instances: Set to automatically scale up to 100 instances based on traffic load to accommodate sudden spikes in user activity.
CPU and Memory per Instance: Each instance is configured with 1 vCPU and 512 MB of memory, balancing cost and performance for the backend processes.

Cloud Storage:

Storage Class: Using Standard Storage for frequently accessed data, such as user profile pictures and posts, ensuring quick retrieval times.

Cloud Function:

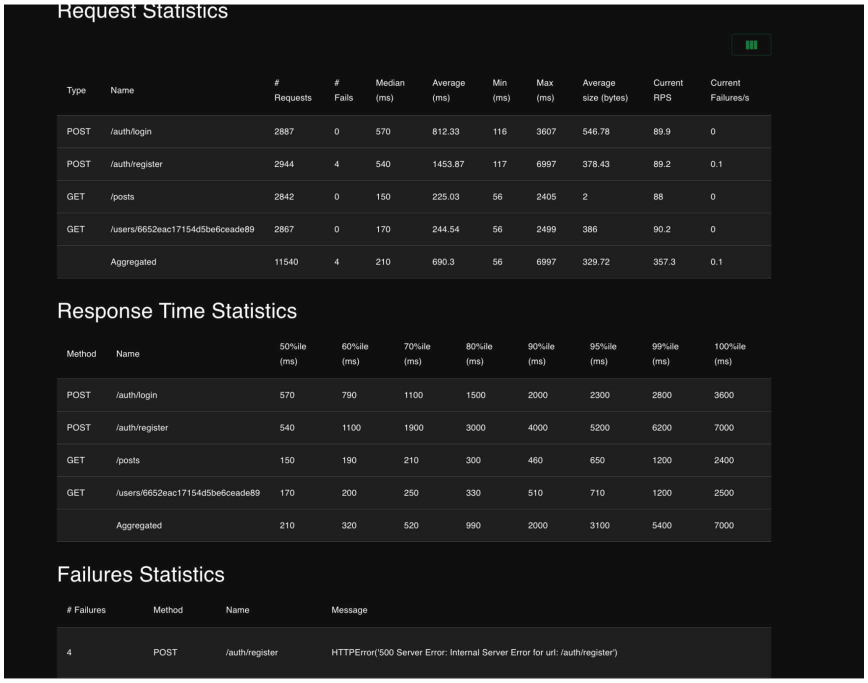
Memory Allocation: Allocated 512 MB of memory to the cloud function to handle image processing tasks efficiently.
Timeout Setting: Configured a timeout of 60 seconds to ensure timely processing and transfer of images to Cloud Storage.
Minimum Instances/Maximum Instances: Minimum 0, maximum 3 instances

4. Experimentation Tool

For the load stress testing, **Locust** is utilized to define user behavior and simulate multiple concurrent users interacting with the social media application. In the tests, scenarios where users register on the platform, upload profile pictures, log in, and interact with the application by accessing various routes such as fetching user information, posts and posting images are implemented to measure the performance of the application in increasing user activity.

5. Results

Initial test
Initial test is conducted for /auth/register, /auth/login, get /posts and get /users routes with the following locust parameters and the baseline system parameters explained above:
Number of users (peak concurrency): 500
Ramp up (users started / seconds) : 10
The system handled a substantial number of requests (11,540) with minimal failures (only 4), indicating good stability under the test conditions (Figure 5.1).



Request Statistics

Type	Name	# Requests	# Fails	Median (ms)	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	Current RPS	Current Failures/s
POST	/auth/login	2887	0	570	812.33	116	3607	546.78	89.9	0
POST	/auth/register	2944	4	540	1453.87	117	6997	378.43	89.2	0.1
GET	/posts	2842	0	150	225.03	56	2405	2	88	0
GET	/users/6652eac17154d5be6ceade89	2867	0	170	244.54	56	2499	386	90.2	0
	Aggregated	11540	4	210	690.3	56	6997	329.72	357.3	0.1

Response Time Statistics

Method	Name	50%ile (ms)	60%ile (ms)	70%ile (ms)	80%ile (ms)	90%ile (ms)	95%ile (ms)	99%ile (ms)	100%ile (ms)
POST	/auth/login	570	790	1100	1500	2000	2300	2800	3600
POST	/auth/register	540	1100	1900	3000	4000	5200	6200	7000
GET	/posts	150	190	210	300	460	650	1200	2400
GET	/users/6652eac17154d5be6ceade89	170	200	250	330	510	710	1200	2500
	Aggregated	210	320	520	990	2000	3100	5400	7000

Failures Statistics

# Failures	Method	Name	Message
4	POST	/auth/register	HTTPError('500 Server Error: Internal Server Error for url: /auth/register')

Figure 5.1: Statistics of the initial test

Average response times for all endpoints are relatively low, suggesting that the system performs well for the majority of requests (Figure 5.2).

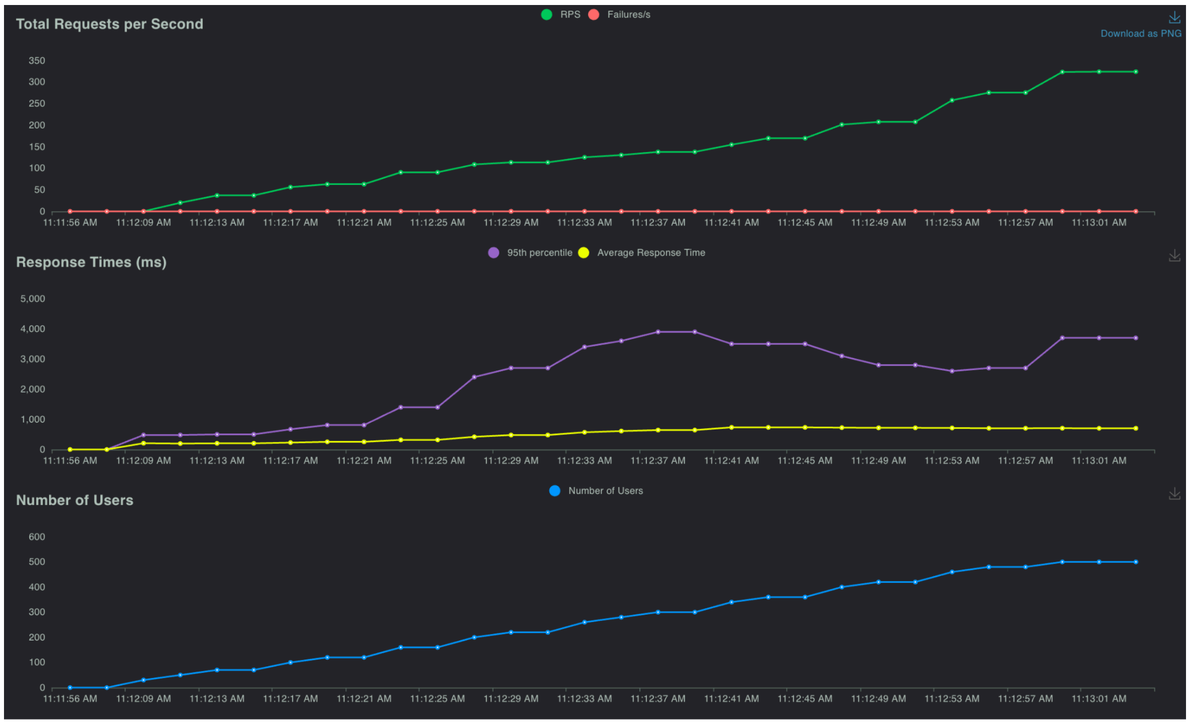


Figure 5.2:Performance graph

The median response times for the POST /auth/login and POST /auth/register endpoints are 570 ms and 540 ms, respectively. These times are within acceptable limits for the social media application.
The GET requests for /posts and /users/{id} endpoints have significantly lower median response times (150 ms and 170 ms), indicating that read operations are well-optimized.

The system was able to sustain an average of 357.3 requests per second with an aggregated average response time of 690.3 ms, which is a solid performance metric for the given load.

The CPU utilization of the database Compute Engine during this experiment was around 39%, indicating that the current resources of the compute engine is sufficient to serve the traffic in this experiment (Figure 5.3).

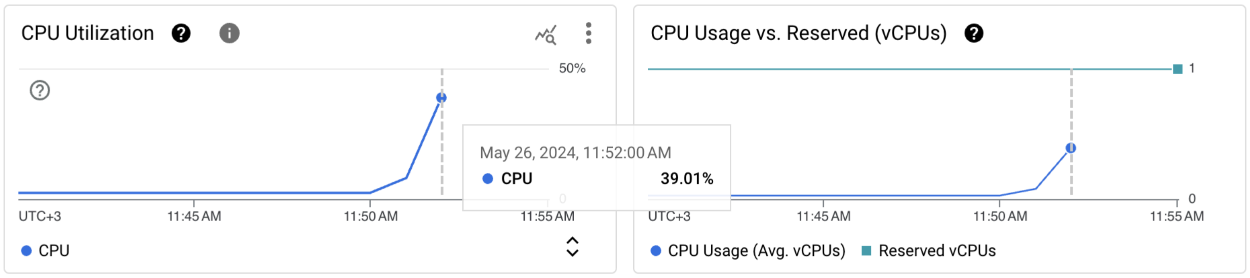


Figure 5.3

Increased number of users and ramp up

Experiment size is increased with the following parameters for the next round of experiments.
Number of peak concurrent users: 1000
Ramp up (users started / seconds) : 20
Following results are obtained (Figure 5.4). The results are quite consistent with the initial experiment, with similar performance graphics. Average response times and failure rates are within the acceptable range for both experiments.

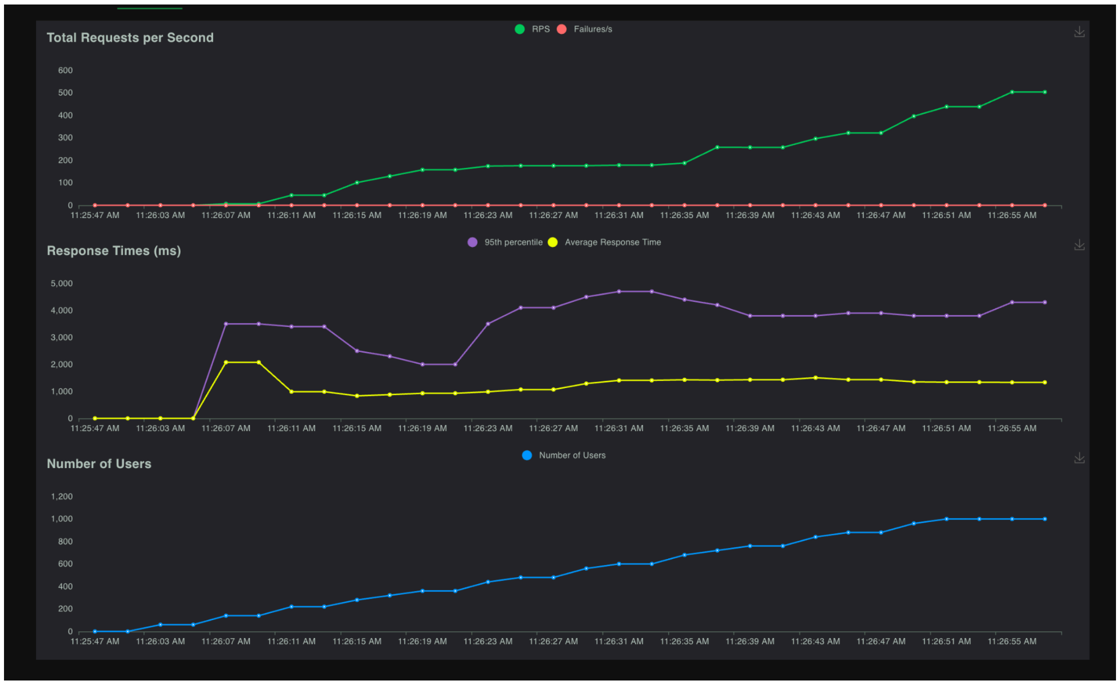
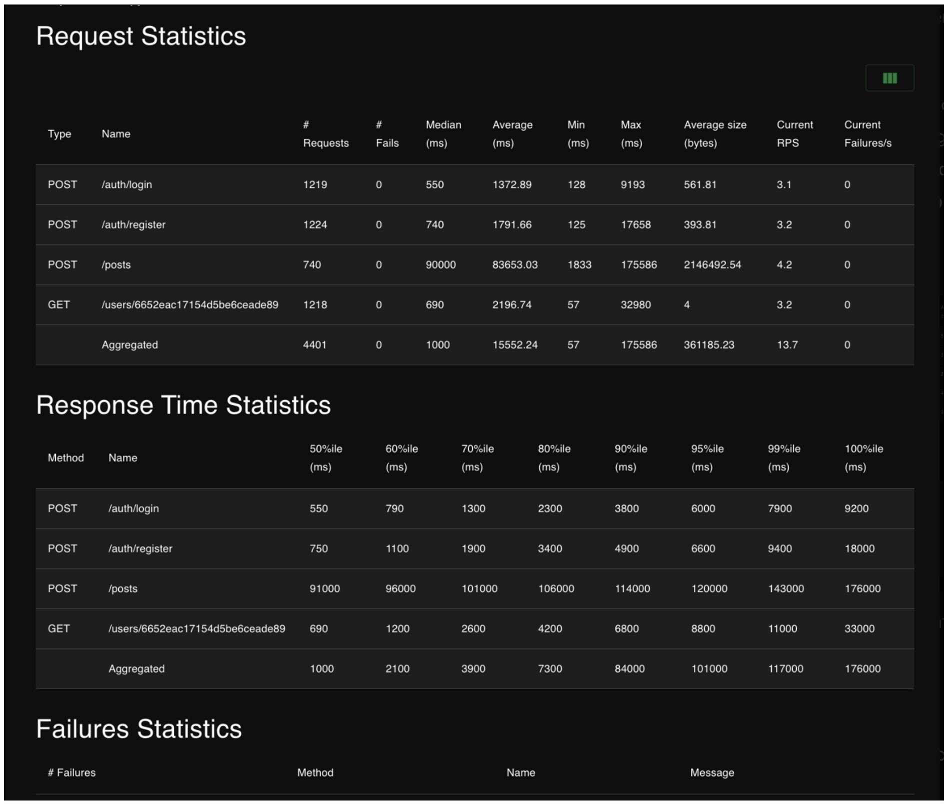


Figure 5.4

Initial test with posting images

Another experiment with the same system and locust parameters (500 concurrent users, 10 ramp up) is conducted with the addition of a post /posts route, to include posting images to the performance testing process. Post /posts route triggers a cloud function whenever a post is made with an image, and sends the image to Cloud Storage to store. Since this is a more complex process, compared to the functionalities of the other routes, more latency is expected and was observed as shown below (Figure 5.5):



Request Statistics

Type	Name	# Requests	# Fails	Median (ms)	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	Current RPS	Current Failures/s
POST	/auth/login	1219	0	550	1372.89	128	9193	561.81	3.1	0
POST	/auth/register	1224	0	740	1791.66	125	17658	393.81	3.2	0
POST	/posts	740	0	90000	83653.03	1833	175586	2146492.54	4.2	0
GET	/users/6652eac17154d5be6ceade89	1218	0	690	2196.74	57	32980	4	3.2	0
	Aggregated	4401	0	1000	15552.24	57	175586	361185.23	13.7	0

Response Time Statistics

Method	Name	50%ile (ms)	60%ile (ms)	70%ile (ms)	80%ile (ms)	90%ile (ms)	95%ile (ms)	99%ile (ms)	100%ile (ms)
POST	/auth/login	550	790	1300	2300	3800	6000	7900	9200
POST	/auth/register	750	1100	1900	3400	4900	6600	9400	18000
POST	/posts	91000	96000	101000	106000	114000	120000	143000	176000
GET	/users/6652eac17154d5be6ceade89	690	1200	2600	4200	6800	8800	11000	33000
	Aggregated	1000	2100	3900	7300	84000	101000	117000	176000

Failures Statistics

# Failures	Method	Name	Message

Figure 5.5

There are no failures in this experiment and the response times of the routes besides post /posts route are within acceptable ranges. However, the latency of post /posts operation demonstrates performance bottlenecks with median response times around 90 seconds. The latency is expected because of the cloud function trigger and Cloud Storage operations. Potential improvements are tried to be made by increasing the number of minimum instances to 1, to avoid cold starts and lower the latency. Even though no significant improvement was observed in the latency, the failure percentage was always near 0, indicating that the system can handle the requests successfully despite the latency. Below the performance graphics of this experiment can be found (Figure 5.6):

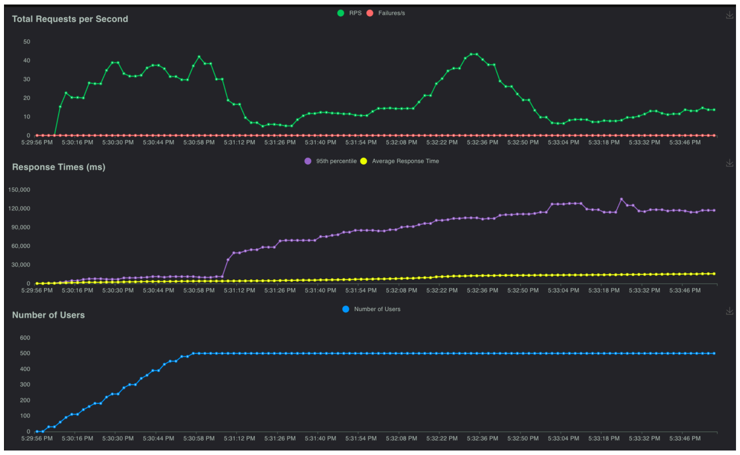
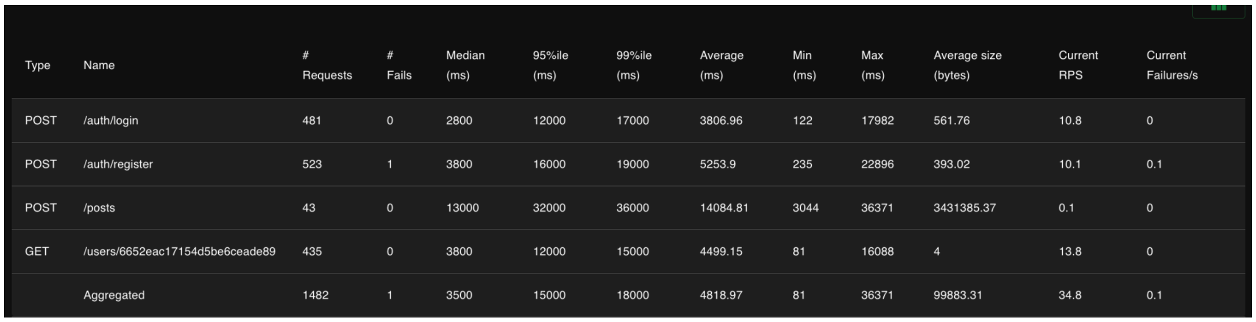


Figure 5.6

Increasing the memory of the Cloud Run containers

The same experiment is conducted by changing the Cloud Run containers' memory to 1 GiB. The results show no further improvement, with similar latencies and failure rates:

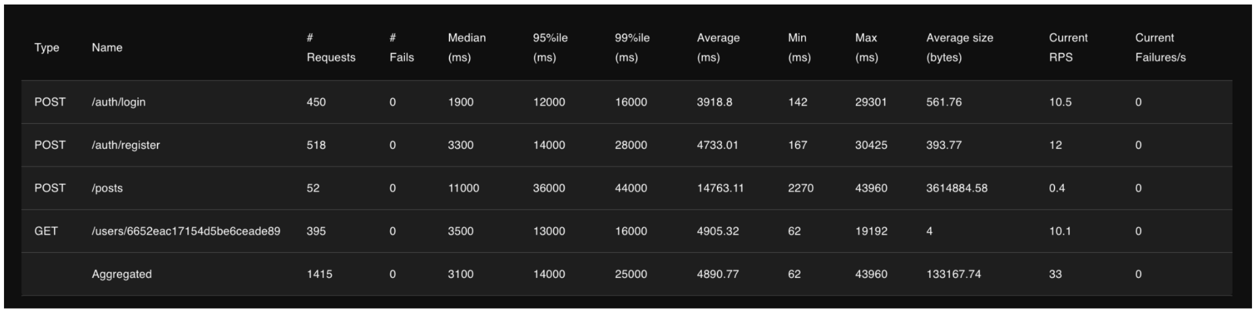


Type	Name	# Requests	# Fails	Median (ms)	95%ile (ms)	99%ile (ms)	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	Current RPS	Current Failures/s
POST	/auth/login	481	0	2800	12000	17000	3806.96	122	17982	561.76	10.8	0
POST	/auth/register	523	1	3800	16000	19000	5253.9	235	22896	393.02	10.1	0.1
POST	/posts	43	0	13000	32000	36000	14084.81	3044	36371	3431385.37	0.1	0
GET	/users/6652eac17154d5be6ceade89	435	0	3800	12000	15000	4499.15	81	16088	4	13.8	0
	Aggregated	1482	1	3500	15000	18000	4818.97	81	36371	99883.31	34.8	0.1

Figure 5.7

Increasing the CPU of the Cloud Run containers

Increased container vCPU to 2, decreased max instances to 50 because of the Google Cloud Quota limitations, observed some improvement in latency and failure rate. This improvement indicates that our social media application can perform better with the increased CPU per instance in Cloud Run. Despite the limitation of max instances under the free trial status, the improvement can be observed as below:



Type	Name	# Requests	# Fails	Median (ms)	95%ile (ms)	99%ile (ms)	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	Current RPS	Current Failures/s
POST	/auth/login	450	0	1900	12000	16000	3918.8	142	29301	561.76	10.5	0
POST	/auth/register	518	0	3300	14000	28000	4733.01	167	30425	393.77	12	0
POST	/posts	52	0	11000	36000	44000	14763.11	2270	43960	3614884.58	0.4	0
GET	/users/6652eac17154d5be6ceade89	395	0	3500	13000	16000	4905.32	62	19192	4	10.1	0
	Aggregated	1415	0	3100	14000	25000	4890.77	62	43960	133167.74	33	0

Figure 5.8

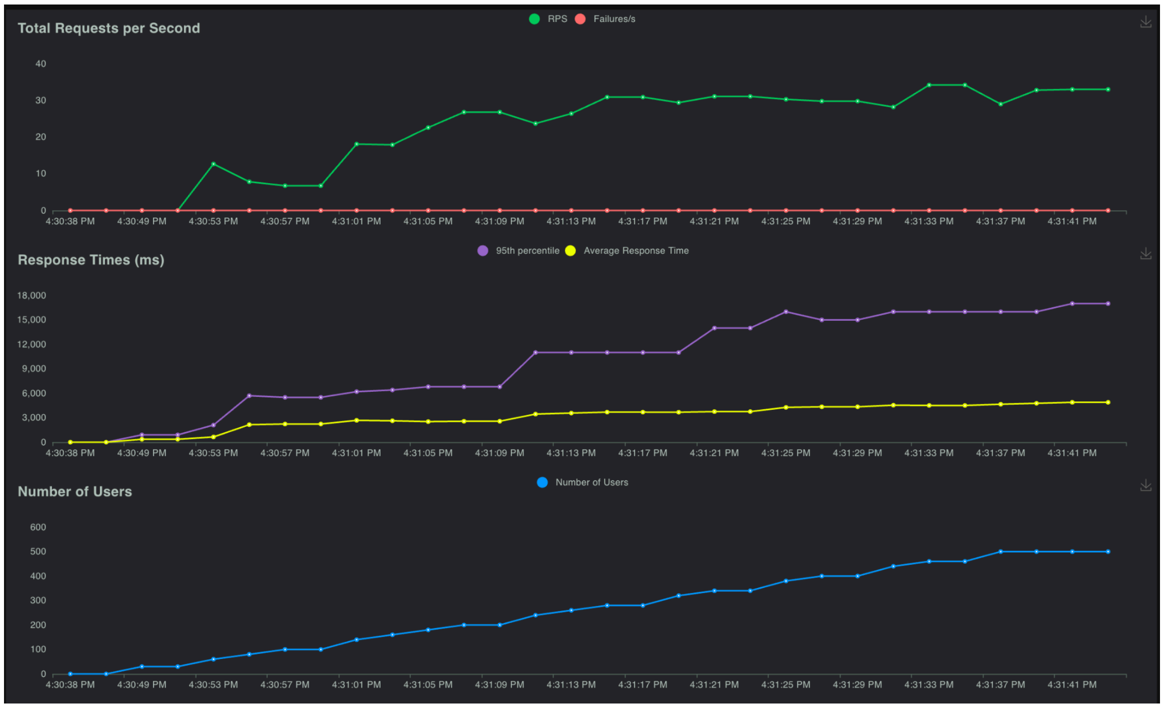


Figure 5.9

Final test with the best parameters

Since the best performance was observed with the Cloud Run: 2 vCPU, 1 GiB memory, 50 max instances, 1 min instances parameters, a final test is conducted with 2000 users, 20 ramp up with these system parameters.

The results can be seen in the graph below. Initially, the service handles the increasing load well, maintaining a stable request rate of 45-50 requests per second (RPS) with minimal failures. However, as the number of users approaches the peak, a performance degradation becomes evident. Both the 95th percentile and average response times, which initially remain low, experience dramatic spikes towards the end, with the 95th percentile reaching 85,000 milliseconds and the average response time hitting approximately 4,561 milliseconds. This suggests that the Cloud Run instances are likely reaching their resource limits. Due to the quota limitations of Cloud Run, the instance limit per region for free trial accounts cannot exceed 100, which means that it is not possible for us to increase CPUs per instance higher while keeping the number of instances.

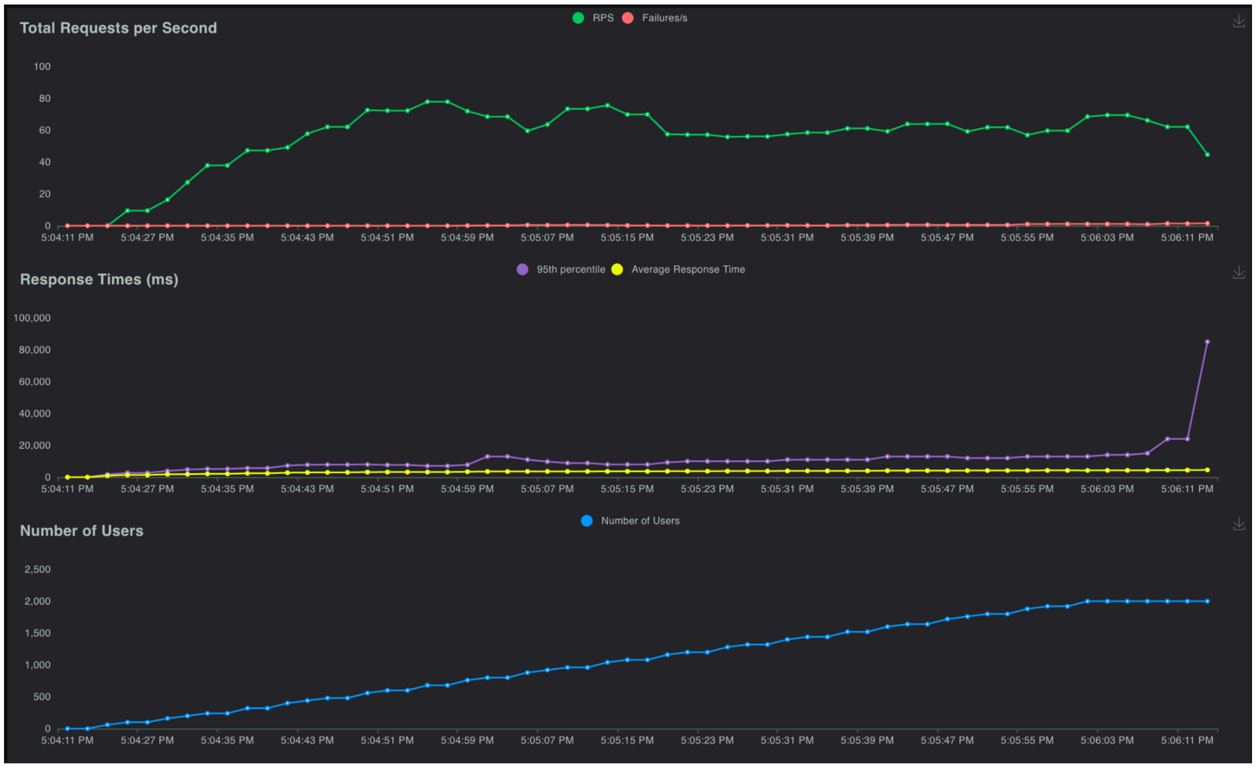


Figure 5.10

Container CPU utilization graph obtained during the tests support that the instances are using mostly all of their CPU resources allocated to them.

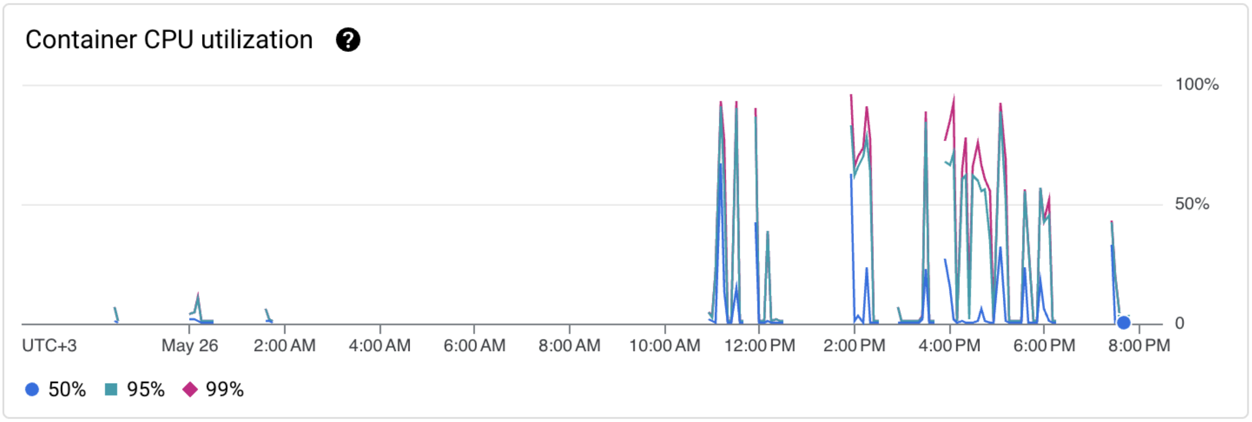

Container CPU utilization
100%
50%
0
UTC+3
May 26
2:00 AM
4:00 AM
6:00 AM
8:00 AM
10:00 AM
12:00 PM
2:00 PM
4:00 PM
6:00 PM
8:00 PM
● 50%
■ 95%
◆ 99%

Figure 5.11