Studycafe: A Cloud-Based Gamified Study Tracking App

Anastasia Deniz Durak, Rana Erkan, Emre Bülbül

Github Repository: https://github.com/Anastasia-Deniz/CS436-Project

I. INTRODUCTION

The StudyCafe is a gamified study tracking application designed to increase user productivity and motivation. The main functionality of StudyCafe provides registered users to set study goals with specific durations and receive rewards, a virtual coffee. By integrating gamification elements, the application aims to make the study process more game-like.

The application is built using a full stack architecture: a React frontend and a Django backend. The frontend is hosted on a private IP and the backend services are distributed across two servers to ensure load balancing and fault tolerance. A Load Balancer & Reverse Proxy, configured using Nginx, manages incoming traffic and routes requests to the appropriate backend services. Additionally, a Google SQL (PostgreSQL) database is used for persistent data storage, ensuring reliable and efficient data management. To further enhance the application's capabilities, a serverless function has been integrated to handle user reward processing.

II. ARCHITECTURE DESIGN

The architecture of StudyCafe uses the following components of Google Cloud Platform (GCP):

Virtual Private Cloud (VPC)

- **VPC Configuration**: A Virtual Private Cloud (VPC) is created to provide an isolated environment for all the resources. The VPC includes a subnet with an IP address range of 10.10.0.0/28, ensuring that all devices within this network have private IPs in this range.

Virtual Machines (VMs)

- **Load Balancer & Reverse Proxy VM**: This VM is configured with **Nginx** to act as a load balancer and reverse proxy. It is the only VM with a public IP address, serving as the gateway to access other VMs. This configuration helps manage traffic efficiently and secure internal components from direct internet access.
- **Frontend VM**: Hosts the React frontend application. It has a private IP and interacts with the load balancer to serve user requests.
- **Backend-1 VM**: Runs one instance of the Django backend. It processes user requests, handles application logic, and communicates with the database and other services.
- **Backend-2 VM**: Runs another instance of the Django backend to ensure load balancing and high availability.

Network Configuration

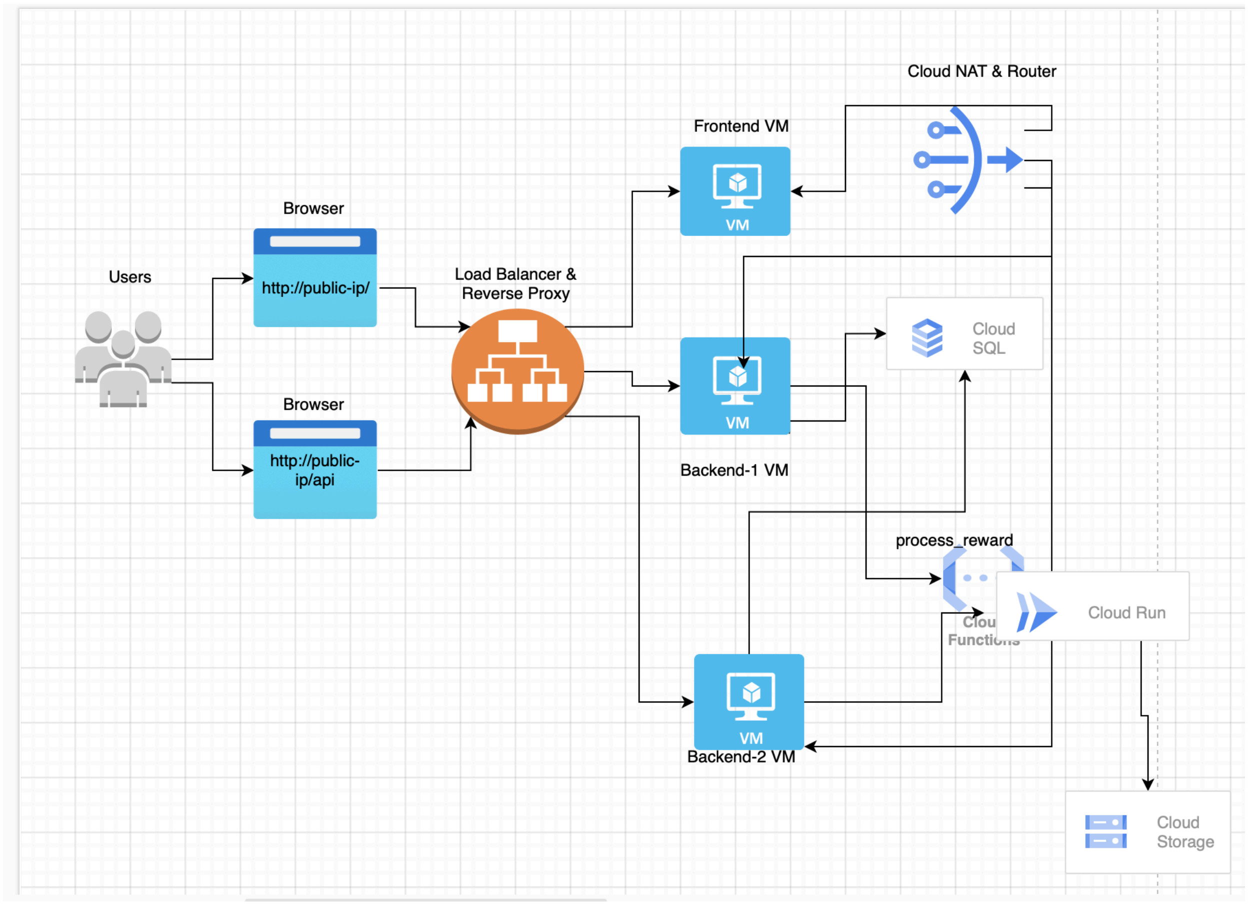
- **Cloud NAT and Router**: To enable internet access for the VMs with private IPs, a Cloud NAT (Network Address Translation) is set up along with a router. This allows the VMs to access external resources for updates and other internet-dependent tasks without exposing their private IPs.

Database

- **Google SQL (PostgreSQL)**: A managed database service that provides a robust and scalable PostgreSQL database. It stores all the persistent data for the application, including user information, study goals, and rewards. The database is connected to the backend VMs through private IPs, ensuring secure and efficient data transactions.

Serverless Functions and Associated Services

- **Cloud Functions**: A serverless function, **process_rewards**, is used to handle the processing of user rewards. This function is triggered by specific events, such as the completion of study goals, and processes the rewards asynchronously. Using Cloud Functions ensures that these tasks do not burden the backend servers, improving overall system performance and scalability.
- **Google Cloud Run**: Cloud Run is used to deploy containerized microservices that work in conjunction with Cloud Functions. For example, the **process_rewards** function can leverage Cloud Run to execute complex reward calculations or other auxiliary tasks efficiently.
- **Google Cloud Storage**: Cloud Storage is used to store various assets required by Cloud Functions, such as configuration files, logs, and backups. This storage ensures that the serverless functions can access necessary data and persist results or logs as needed.



III. EXPERIMENT DESIGN AND METHODOLOGY

Experimentation Tool

To optimize the application and to identify performance bottlenecks Locust is chosen which is an open source testing tool to help developers and testers. Locust is chosen due to its scalability and flexibility, it stimulates concurrent users interacting with an application using Python. Users, represented by Python classes, execute tasks like making HTTP requests, these tasks are defined as methods. The execution of these tasks, which is managed by Locust, controls the behavior patterns of a number of users while collecting metrics such as response times, failure rates, and requests per second during the test.

Stress Testing

The stress tests were designed to assess the application's performance under different load conditions by stimulating the following user behaviors and scenarios:

Normal Load: Such as creating goals, stimulating typical user interactions.
Peak Load: Simulating a sudden surge in user activity, like promotional events or product launches.
Sustained Load: Over an extended period continuous user activity is stimulated to evaluate stability and resource usage.

Specific Scenarios and User Behaviors:

- **Creating Users:** Users creating, registering new accounts.
- **Logging In:** Users logs into the system.
- **Creating Goals:** Frequently creating new tasks (goals) by users.
- **Fetching Goals:** User fetches goals.
- **Concurrent Users:** To test the application's scalability and performance limits increasing the number of users.

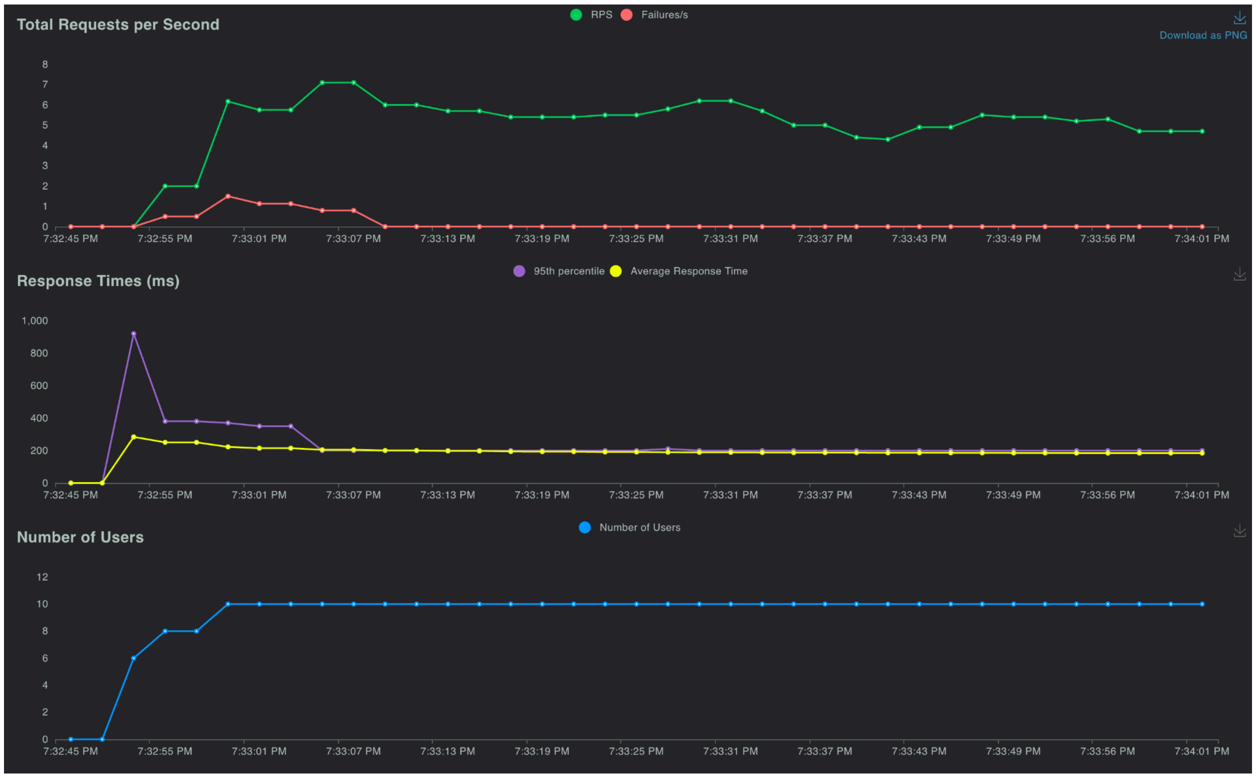
Explanation of the Metrics Collected During the Tests:

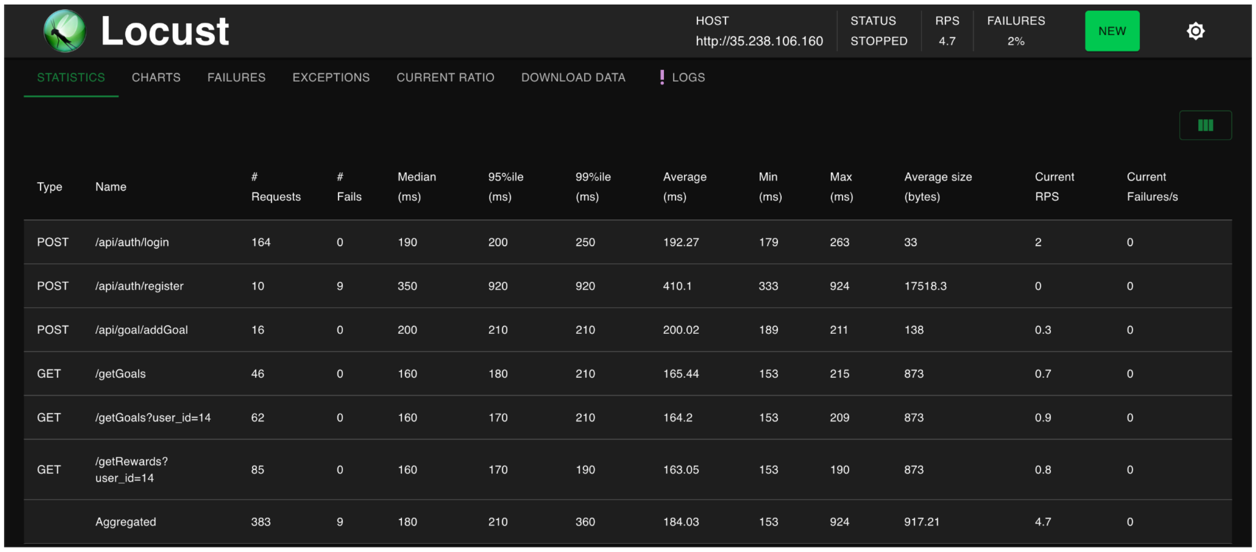
During the stress tests, the following metrics were collected to evaluate and assess the application's performance:

Response Time: The time duration for the application to respond to user requests. Lower response times signify better performance.
Request Rate: The number of requests handled per second. Higher request rates demonstrates the application's capacity to handle increased load.
Failure Rate: The percentage of requests that result in errors. A lower failure rate shows a greater reliability and stability of the system.
Resource Utilization: Tracking CPU, network usage and memory to spot the potential bottlenecks and optimize resource allocation.

IV. RESULTS

Run #1



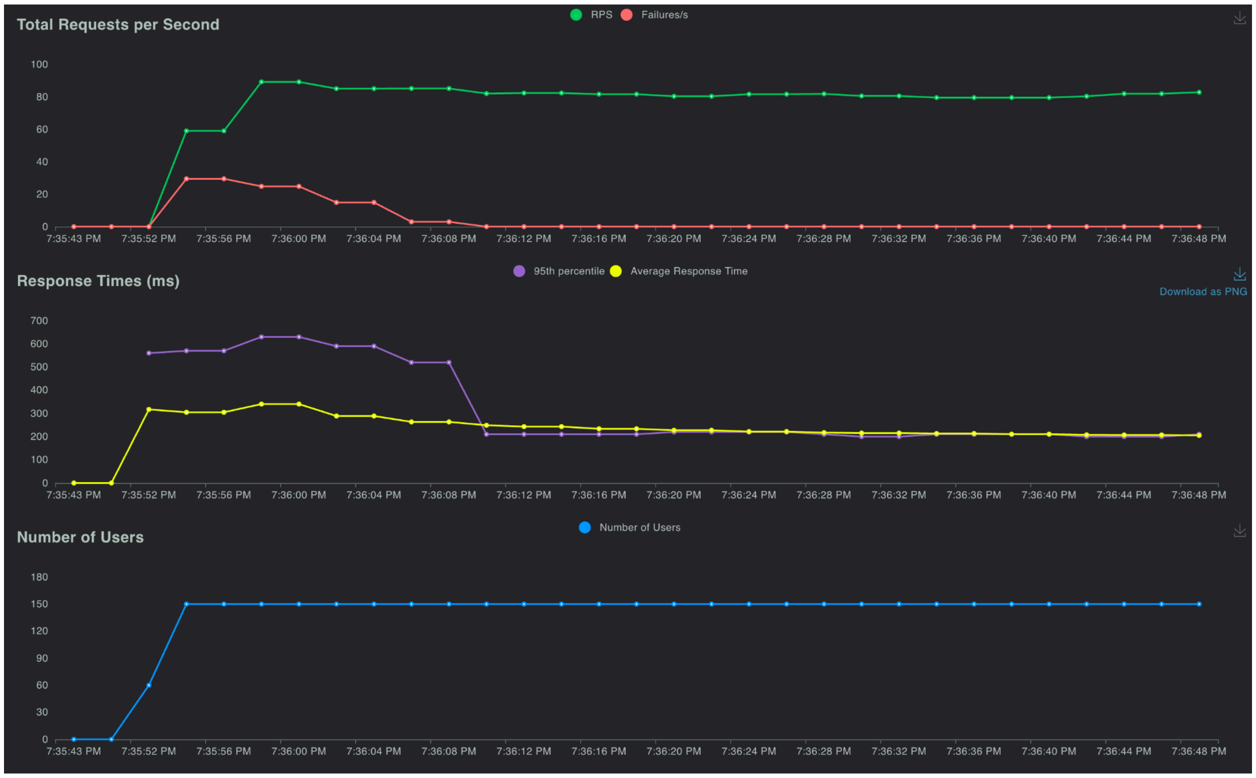


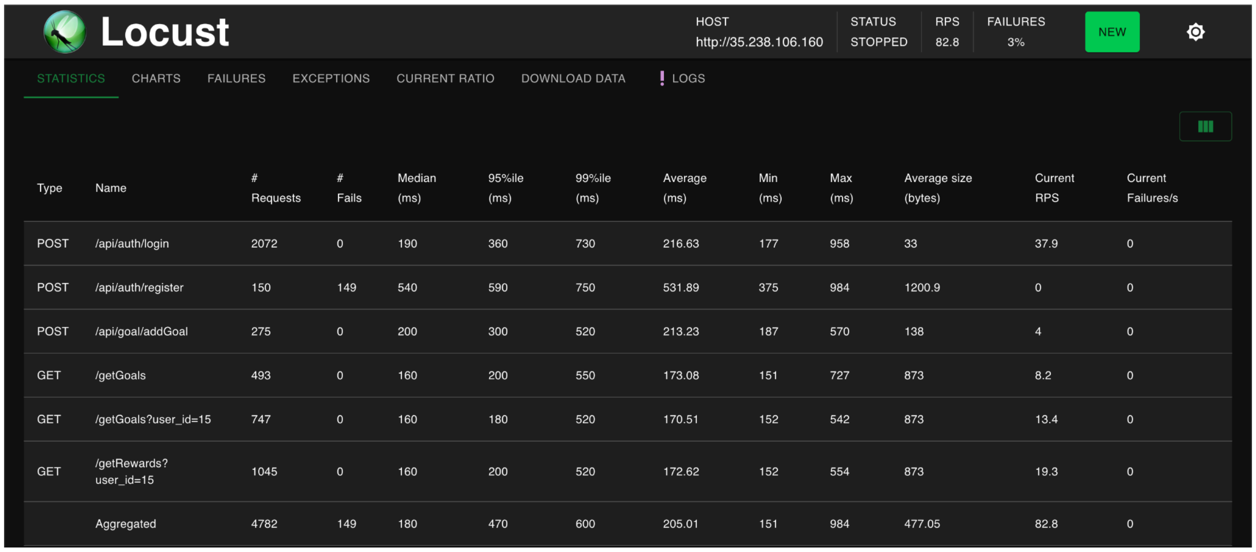
HOST http://35.238.106.160 STATUS STOPPED RPS 4.7 FAILURES 2%

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	/api/auth/login	164	0	190	200	250	192.27	179	263	33	2	0
POST	/api/auth/register	10	9	350	920	920	410.1	333	924	17518.3	0	0
POST	/api/goal/addGoal	16	0	200	210	210	200.02	189	211	138	0.3	0
GET	/getGoals	46	0	160	180	210	165.44	153	215	873	0.7	0
GET	/getGoals?user_id=14	62	0	160	170	210	164.2	153	209	873	0.9	0
GET	/getRewards?user_id=14	85	0	160	170	190	163.05	153	190	873	0.8	0
	Aggregated	383	9	180	210	360	184.03	153	924	917.21	4.7	0

Number of users(peak concurrency) = 10, Ramp up(users started/seconds) = 2

Run #2



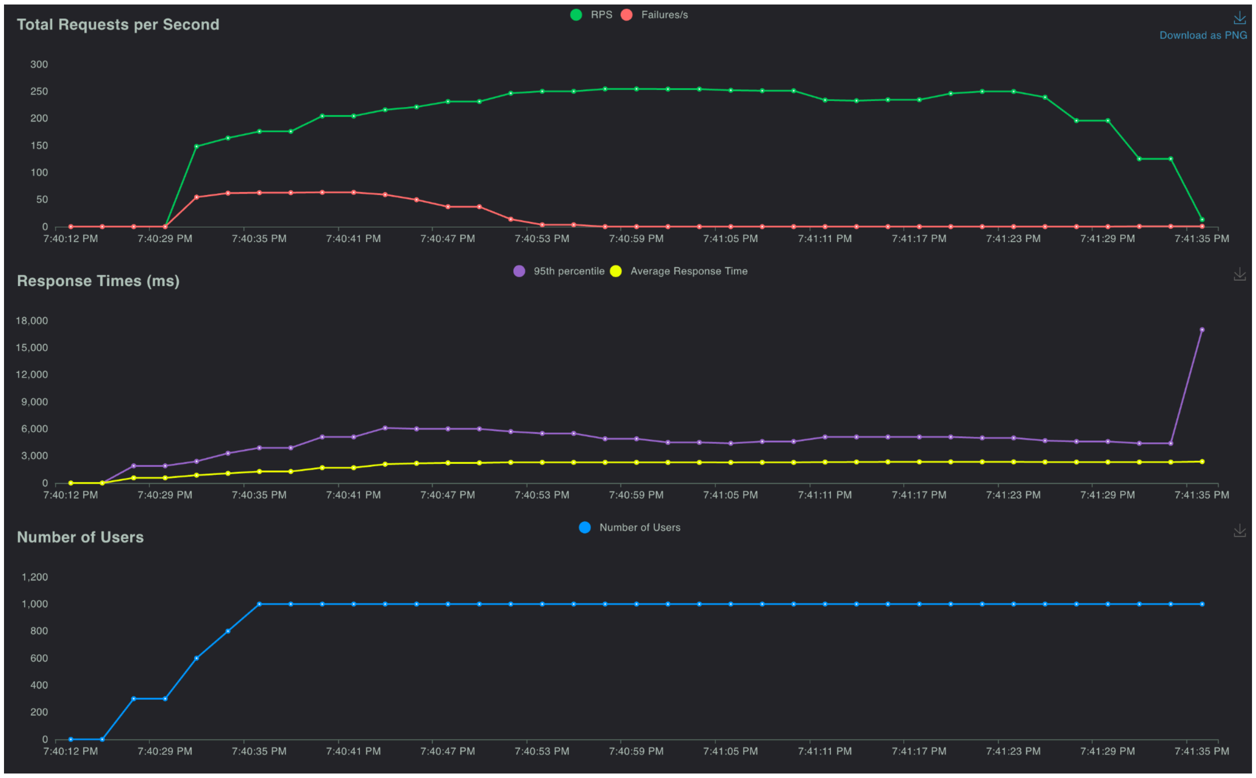


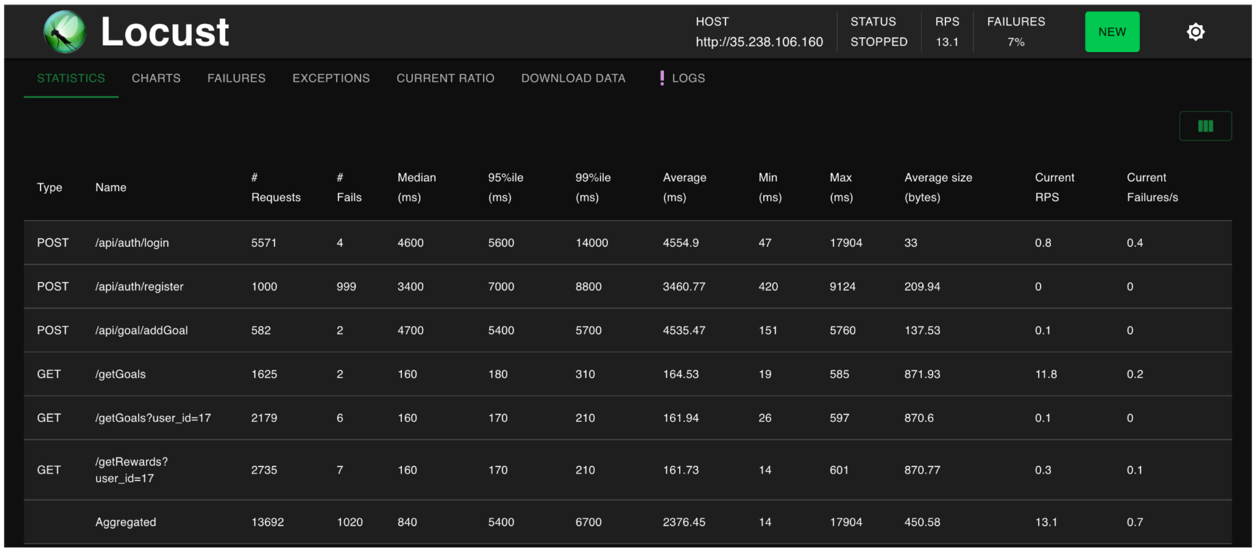
HOST http://35.238.106.160 | STATUS STOPPED | RPS 82.8 | FAILURES 3%

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	/api/auth/login	2072	0	190	360	730	216.63	177	958	33	37.9	0
POST	/api/auth/register	150	149	540	590	750	531.89	375	984	1200.9	0	0
POST	/api/goal/addGoal	275	0	200	300	520	213.23	187	570	138	4	0
GET	/getGoals	493	0	160	200	550	173.08	151	727	873	8.2	0
GET	/getGoals?user_id=15	747	0	160	180	520	170.51	152	542	873	13.4	0
GET	/getRewards?user_id=15	1045	0	160	200	520	172.62	152	554	873	19.3	0
	Aggregated	4782	149	180	470	600	205.01	151	984	477.05	82.8	0

Number of users(peak concurrency) = 150, Ramp up(users started/seconds) = 30

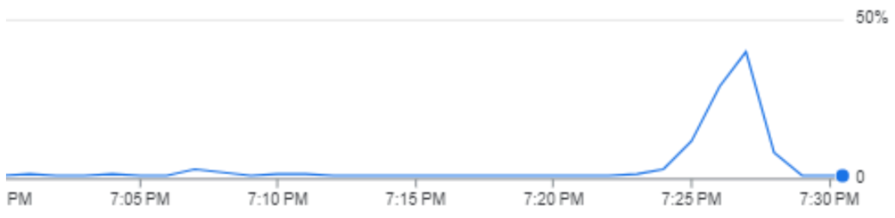
Run #3



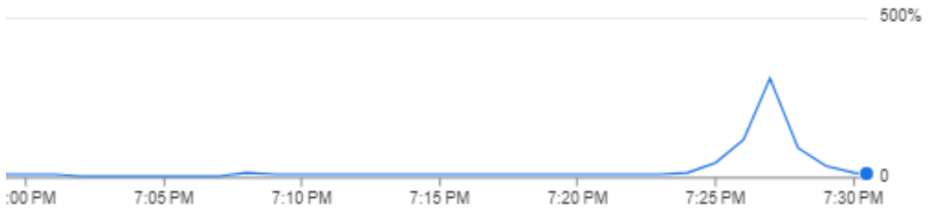


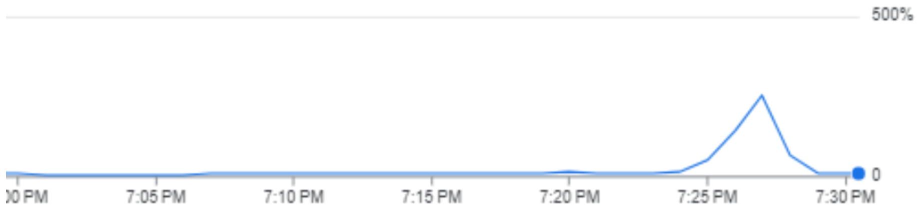
Type	Name	# Requests	# Fails	Median (ms)	95%ile (ms)	99%ile (ms)	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	Current RPS	Current Failures/s
POST	/api/auth/login	5571	4	4600	5600	14000	4554.9	47	17904	33	0.8	0.4
POST	/api/auth/register	1000	999	3400	7000	8800	3460.77	420	9124	209.94	0	0
POST	/api/goal/addGoal	582	2	4700	5400	5700	4535.47	151	5760	137.53	0.1	0
GET	/getGoals	1625	2	160	180	310	164.53	19	585	871.93	11.8	0.2
GET	/getGoals?user_id=17	2179	6	160	170	210	161.94	26	597	870.6	0.1	0
GET	/getRewards?user_id=17	2735	7	160	170	210	161.73	14	601	870.77	0.3	0.1
	Aggregated	13692	1020	840	5400	6700	2376.45	14	17904	450.58	13.1	0.7

Number of users(peak concurrency) = 1000, Ramp up(users started/seconds) = 100



CPU utilization of the load balancer and reverse proxy VM





CPU utilizations of backend VMs

1. Total Requests per Second:

Concurrency 10, 2 Users/Second:
The graph shows a stable increase in the number of requests per second, with minimum failures it peaked around 7 RPS with. Without any significant problems system handles the load well.

Concurrency 150, 30 Users/Second:
The graph shows indicates a rapid increase in the number of requests per second, stabilizing around 90 RPS. The system can handle a moderate load due to the relatively low failures per second.

Concurrency 1000, 100 Users/Second:
The graph shows a significant spike in requests per second, it peaked at approximately 250 RPS. Failures per second increase notably, it suggests that the system is struggling to handle this high load.

2. Response Times (ms):

Concurrency 10, 2 Users/Second:
Under low load it indicates a successful performance. The average response time remains low and stable, around 200 ms, with occasional spikes in the 95th percentile up to 400 ms.

Concurrency 150, 30 Users/Second:
Average response time increases to around 400 ms, with the 95th percentile showing peaks up to 700 ms. This indicates the system is experiencing some delays but still performing adequately under moderate load.

Concurrency 1000, 100 Users/Second:
Average response time significantly increases, with the 95th percentile peaking at over 15,000 ms, indicating substantial delays and performance degradation under high load.

3. Number of Users:

Concurrency 10, 2 Users/Second:
The number of users increases steadily and stabilizes at 10 users. This indicates the test setup correctly matches the specified concurrency and user start rate.

Concurrency 150, 30 Users/Second:
The number of users increases rapidly and stabilizes at 150 users, aligning with the test parameters.

Concurrency 1000, 100 Users/Second:
The number of users spikes quickly, reaching the maximum of 1000 users, as expected. The high load significantly stresses the system.

4. CPU Utilization:

Concurrency 10, 2 Users/Second:
The graph shows a stable increase in the number of requests per second, peaking around 7 RPS for backend VMs with minimal failures. The system handles the load well without significant problems.

Concurrency 150, 30 Users/Second:

The graph indicates a rapid increase in the number of requests per second, stabilizing around 90 RPS for backend VMs. The system can handle a moderate load with relatively low failures per second.

Concurrency 1000, 100 Users/Second:
The graph demonstrates a significant spike in requests per second, peaking at approximately 250 RPS for backend VMs. Failures per second increase notably, suggesting that the system is struggling to handle this high load.

V. FINDINGS

Under the varying loads the test revealed key insights in to the system performance

Low Concurrency: The system handles low concurrency well, maintaining low response times and no failures (since failures for registrations are excluded because the same person tries to sign up again after initial registration). It is well-suited for environments with light traffic. CPU utilization remains minimal, indicating efficient resource use.

Moderate Concurrency: The system shows reasonable performance under moderate load, with increased but manageable response times. This indicates the system is capable of handling moderate traffic with occasional performance hits. CPU utilization peaks moderately but recovers quickly, demonstrating good load management.

High Concurrency: The system struggles under high load, evidenced by significant increases in response times and the number of failures. This indicates that the system needs optimization or scaling to handle high traffic volumes effectively. CPU utilization spikes significantly, suggesting that the system is heavily stressed and requires additional resources or optimizations to maintain performance.

VI. DISCUSSION

The stress tests highlighted that while the application performs well under low to moderate loads, however in high concurrency it faces significant challenges. The discussion part, recommends the following actions to ensure stable performance at higher user loads:

1. Scaling Infrastructure: Increasing the capacity of the infrastructure to support more concurrent users.
2. Optimizing Code: Improving the efficiency of the application's code to reduce response times and handle more requests per second.

3. Advanced Load Balancing: Implementing improved load balancing techniques to evenly distribute the load across servers.
4. CPU Utilization Monitoring: Continuously monitoring CPU utilization to identify and address potential bottlenecks in real-time.

Addressing these can significantly improve the application's performance under high-traffic conditions, and it ensures reliability and stability. This comprehensive evaluation and discussion serves as a foundation for future enhancements, leading to a robust and high-performing product.

VII. CONCLUSION

In conclusion, project The StudyCafe has successfully developed a cloud-based, gamified study tracking application that enhances productivity and motivation for users. It applies the most modern cloud technologies to ensure robustness, scalability, and friendliness toward its users. The stress test indicated great performance under conditions of low to moderate load, though it signals that possibly the application should be optimized against greater concurrency levels. This has therefore indicated the application is in a position to handle conventional behaviors of users, including the creation and fetching of goals and had yielded valuable insights with respect to the scalability and stability of the application. Scaling the infrastructure, optimizing the code, and use of more advanced load-balancing techniques may be necessary for better performance, especially under high loads. In general, the project The StudyCafe has managed to build a reliable foundation for a scalable application, a foundation upon which further improvements can be built to meet ever-increasing demands.