CS 436: Cloud Computing
Term Project Report

Group Members:

Alper Kaan Odabaşoğlu
Barış Ulaş Çukur
Işıktan Tanış
Oktay Çelik

GitHub URL: https://github.com/BarisUlas/CS436_Project

Spring 2024
Sabancı University

1) Overview of the Whole Architecture

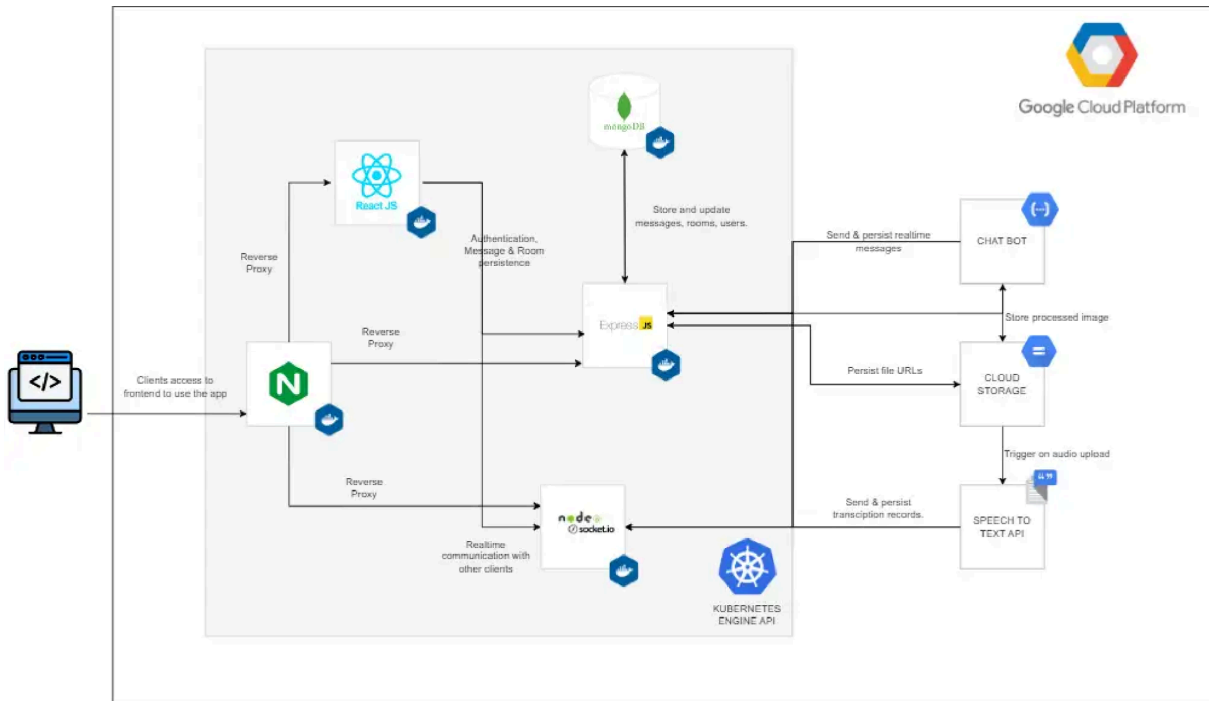

Google Cloud Platform
mongoDB
Store and update messages, rooms, users.
React JS
Reverse Proxy
Authentication, Message & Room persistence
Send & persist realtime messages
CHAT BOT
Express JS
Store processed image
Reverse Proxy
Persist file URLs
CLOUD STORAGE
Clients access to frontend to use the app
Trigger on audio upload
Reverse Proxy
Send & persist transciption records.
SPEECH TO TEXT API
node socket.io
Realtime communication with other clients
KUBERNETES ENGINE API

Figure 1: Architecture Scheme

The aim of our project is to provide users a chatting application with some additional features. The barebones of the application is a text based chatting application but we have extended it with unique features such as a Chatbot that can do image processing and audio based communication with transcripts.

Our project works with 5 containers and 3 Google Services. The containers are designed to be used in a Kubernetes cluster to enable a finer grained scaling and elasticity. Nginx is used as a reverse proxy, ReactJS as the frontend, Socket.io for enabling real-time communication, ExpressJS to handle persistent queries such as messages and authentication queries and MongoDB as the database.

In order to be concise about how different parts of the architecture interact with each other, we must examine their relationship in different use cases such as authentication and sending different types of text messages.

First of all, a new user is required to be registered to the database. This is implemented through ExpressJS and MongoDB. After acquiring the user information, the database is updated by ExpressJS and a JWT is sent to the user. The login procedure has similar interactions with the containers. After signing in the user picks another user to start a chat with. The user is selected with search queries. After the chat room is initiated the user can send start messages. There are three types of messages, text only messages, messages sent to chatbot, and voice messages that are transcribed by Google Cloud's Speech-to-Text API.

The text messages are taken from the user through React, gets duplicated to be sent to the receiving user through Socket.IO in real-time and to be saved into the database through ExpressJS. ExpressJS maintains a unique chat identifier for each chat room to save the messages into MongoDB.

Messages sent to chatbot are taken from the frontend and forwarded to ExpressJS only. This container saves the message into the database and forwards it to the Chatbot. The Chatbot is implemented as a serverless function hosted in Google Compute Functions. It has three possible actions, help, black and white conversion and blurring. The help route has minimal computational footprint as its response is a description of the available options. The black and white conversion and blurring is implemented with OpenCV functions. As it takes time to apply these filters, the response times are much higher than the help route. The resulting image is sent to Cloud Storage to be stored. The respective URL is sent back to the ExpressJS to be saved persistently. The image is sent with Socket.IO to the receiver.

The last type of messages is voice messages. The audio file and the chat identifier are taken from the frontend to be sent to the ExpressJS container. The message is saved persistently to the database and the audio recording is sent to Google's Speech-to-text API. The transmission of the audio file and the processing time that it takes to transcribe at Google's end makes this route have the largest latency when compared with other message types. After collecting the transcription, the transcript is duplicated and sent through Socket.IO to the receiver and persistent storage through ExpressJS.

2) Performance Evaluation and Testing

Our architecture broadly has 3 stages of execution, registering and logging in, creating a chat with an existing user, and sending a message. We are simulating these three stages in each execution of the script by a user created by Locust.

```
define function stressTest(N,M,pT,pI,pV):
1.    register with a random username and mail
2.    get a list of the current users
3.    pick N random users to create a chat
4.    for each users to chat with
5.        create a chat room
6.        for M number of messages:
7.            pick text message with probability pT.
8.            pick image message with probability pI.
9.            pick audio message with probability pV.
10:           send the message
11:       endfor
12:   endfor
```

Figure 2: Pseudocode of the Locust Task for a single user

The pseudo code provided gives an overview of our testing script. It has multiple parameters to simulate different levels of loads when testing. First parameter is the number of users per second. This is a parameter set through the Locust interface. It can specify the number of users to simulate per second and a spawn rate that determines the users created per second. The procedure explained in the pseudocode is the task of one user created by Locust.

The second parameter is denoted with “N” in the pseudo code. This parameter shows how many users the current user will send messages to. This drastically increases the load on the server-side as for each additional user to send messages to, the total number of messages sent gets increased by M.

The third parameter is denoted with “M” in the pseudo code. It determines how many messages a user will send to each other. This can accumulate over time to create load on each container. The lines 2 to 6 mainly exhaust the database and ExpressJS.

The last parameter is the probability distribution of the messages. The distribution of these three messages affect the real-world load greatly as message types have no homogenous computational footprint. Almost always text messages are sent and received with minimum latency. On the other hand Speech-to-text API accesses and the chatbot queries have a much larger footprint. The lines 7 to 11 test the maximum number of components at any given time.

At each step of the test we are testing different components of the architecture. For example during the registration and retrieval of information about all currently registered users is an ExpressJS and database intensive query. As each user created by Locust will accumulate over time the length of the retrieved data will increase drastically. For the chat creation process we are mainly targeting Socket.IO and ExpressJS as it is done in a parallel fashion. The message sending phase will create the most load on overall architecture as it would include external accesses to Google Cloud Services depending on the message type.

3) Experiment Design

a) Testing Parameters

We have three main levels of stress testing of our architecture. The hardness of each level is determined by the values of our Locust parameters. The main parameters that we control are the number of users a given user interacts with which is denoted as “N” in the pseudocode and the number of messages sent to each receiving user which is denoted as “M” in the pseudocode. Additionally, the probability distribution of message types is also configured as a testing parameter.

The easiest level has 20 receiving users and 10 messages for each receiving user with text dominant messages. Namely, any given message will be a text message half of the time and the rest will be divided equally. The medium level has 50 receiving users and 20 messages for each receiving user with uniform distribution of the message types. The hardest level has 100 receiving users and 30 messages for each receiving user. Since the message type that has the highest computational footprint is Voice messages, this layer has voice messages half of the time and the rest is distributed evenly.

	N	**M**	**pT**	**pV**	**pI**	**#users**	**Spawn Rate**
Low	20	10	0.50	0.25	0.25	50	5
Medium	50	20	0.33	0.33	0.33	100	10
High	100	30	0.25	0.25	0.50	150	10

Figure 3: Table of Testing Parameters

b) System Parameters

Number of replicas per pod is the fundamental determination tool for the testing system parameters. Total of 13 parameters are generated. Initial parameter is based on the experiment design which indicates whether a low, medium or high number of requests will be possessed on our system. This initial configuration is handled automatically from CI/CD with a dedicated variable provision to the pipeline. For each configuration of low, medium or high, a predetermined number of replicas per pod for ExpressJS, Socket.IO, React and Nginx is provided to the system. So, for each 3 of the configurations, 4 system parameters are provided. At the end, a total of 13 parameters are handled. Since we do not wish to handle a distributed database, a single Mongo pod is provided with one replica for each configuration. In other words, only vertical scaling is allowed for the Mongo pod. However, various different pod replicas are provided to aforementioned services. Vertical and horizontal scaling is automatically handled by the Kubernetes Engine itself through the Autopilot feature.

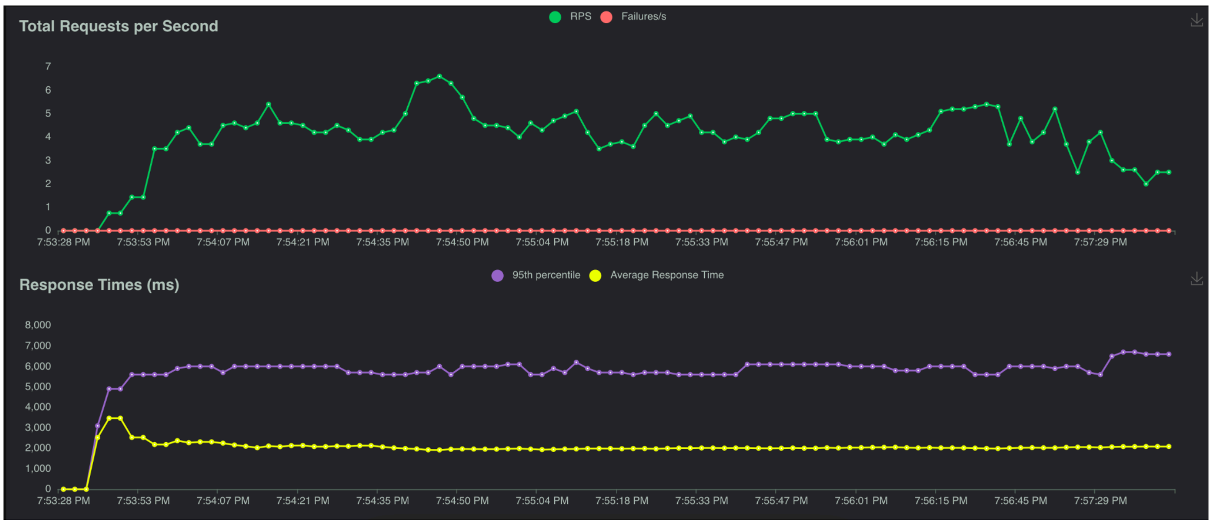
4) Scaling

Scaling of our system is mainly handled by the Kubernetes cluster itself. The nature of the Kubernetes already satisfies the auto-scaling. For MongoDB, we prevent our database to be a distributed database with proper configurations. For this, only the vertical scaling option is provided to the database. This ensures the database is scaled up by increasing the resources (CPU, memory) available to it. For other services in the architecture both horizontal and vertical scaling is provided. Different amounts of replicas per pod are provided to different services which increase the elasticity of the system. Due to the internal configurations of Kubernetes, these services could be horizontally scaled by increasing the amount of nodes. As a consequence, both horizontal and vertical scaling is internally handled by Kubernetes.

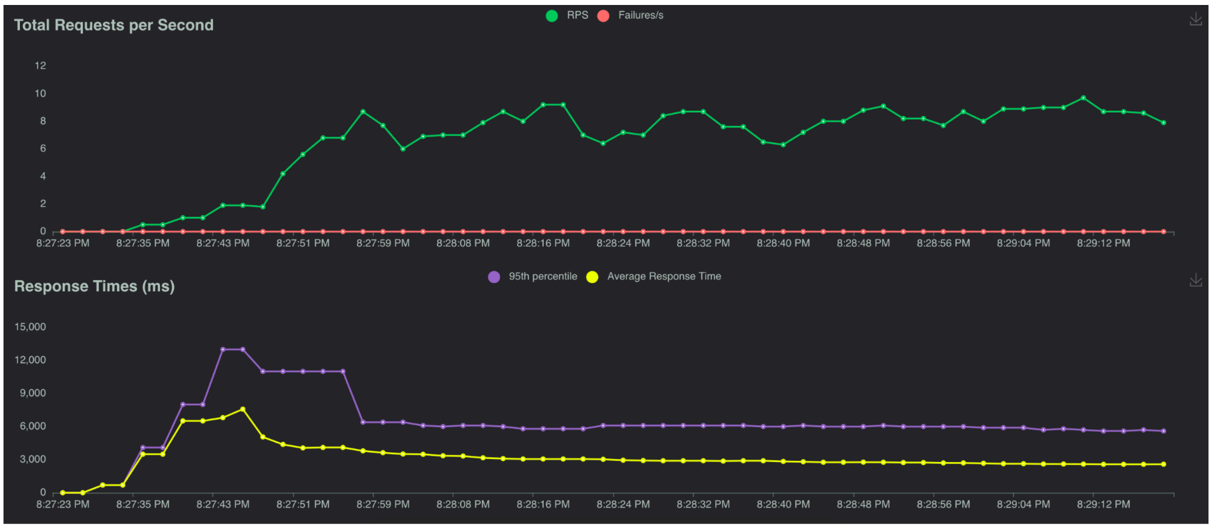
5) **Benchmark Results**

a) Low System Configuration

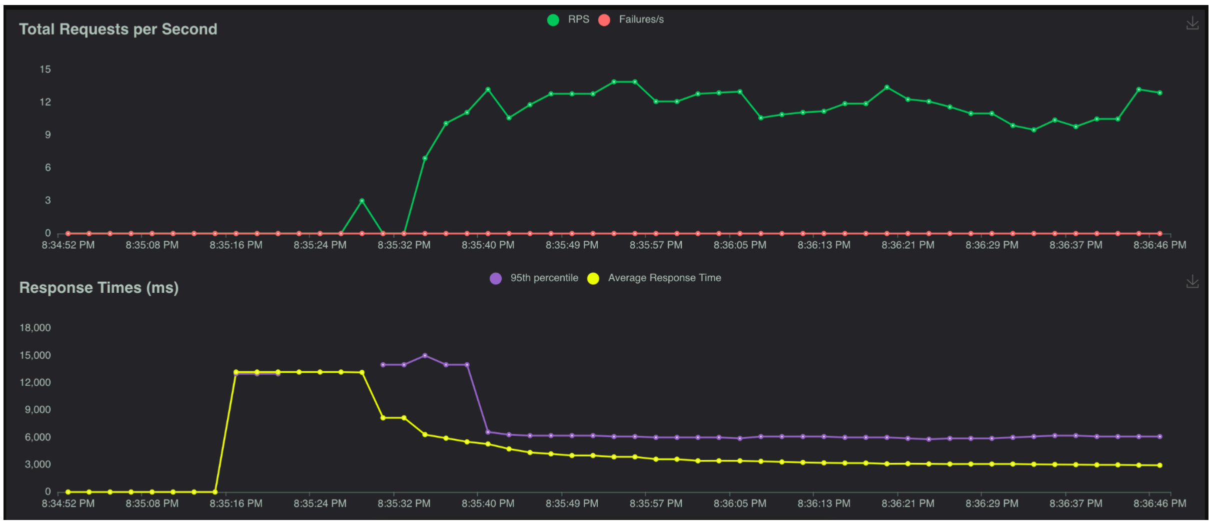
1) Low Traffic



2) Medium Traffic

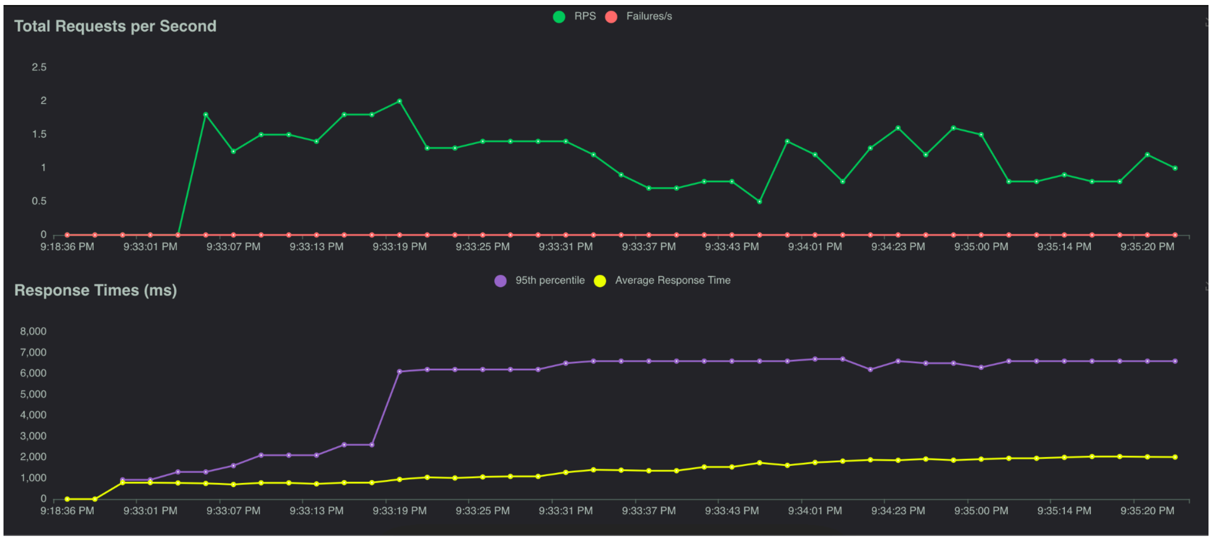


3) High Traffic

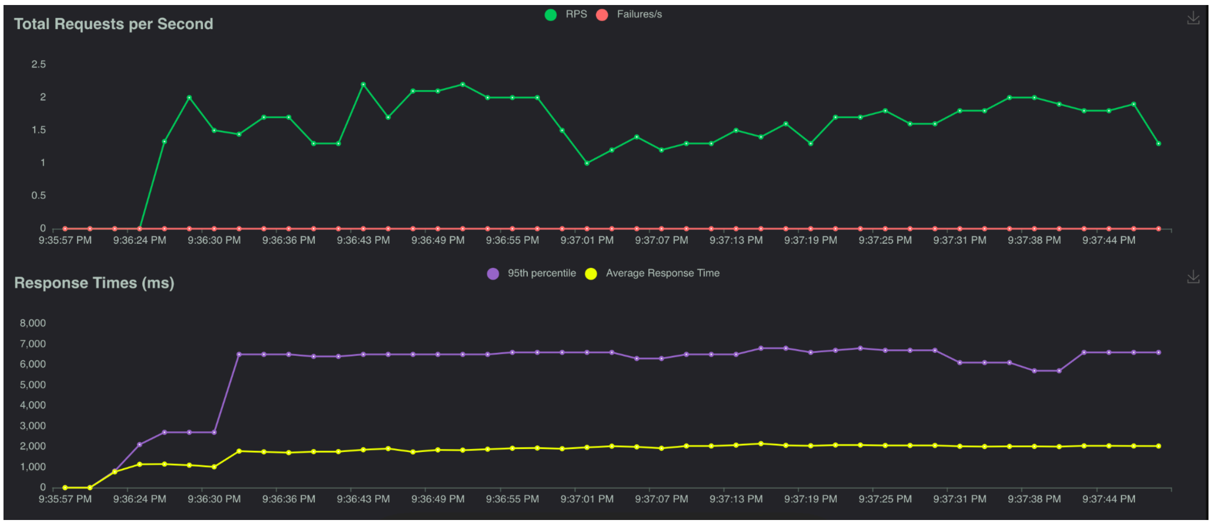


b) Medium System Configuration

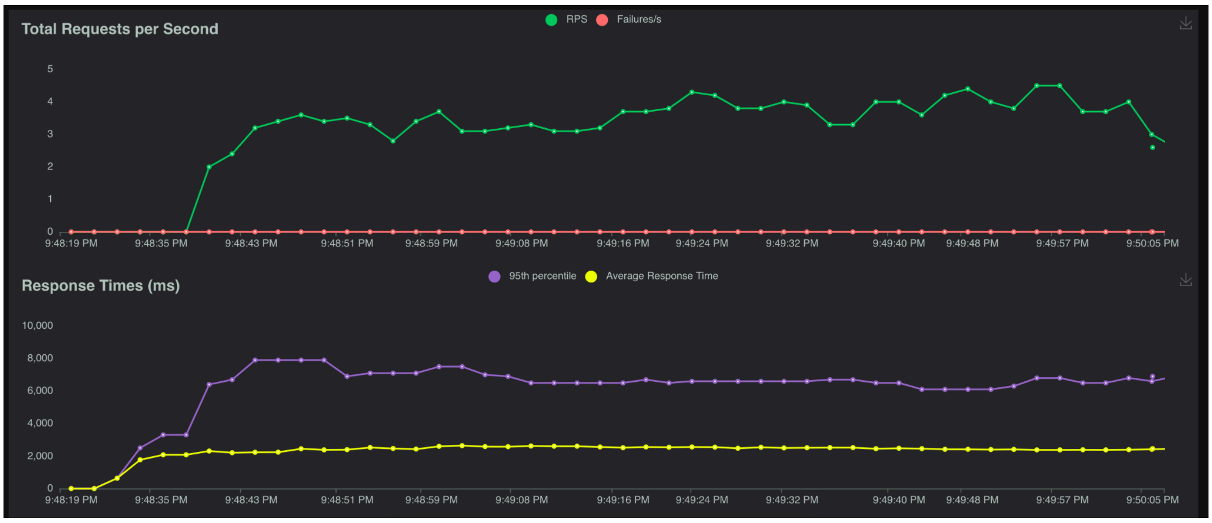
1) Low Traffic



2) Medium Traffic

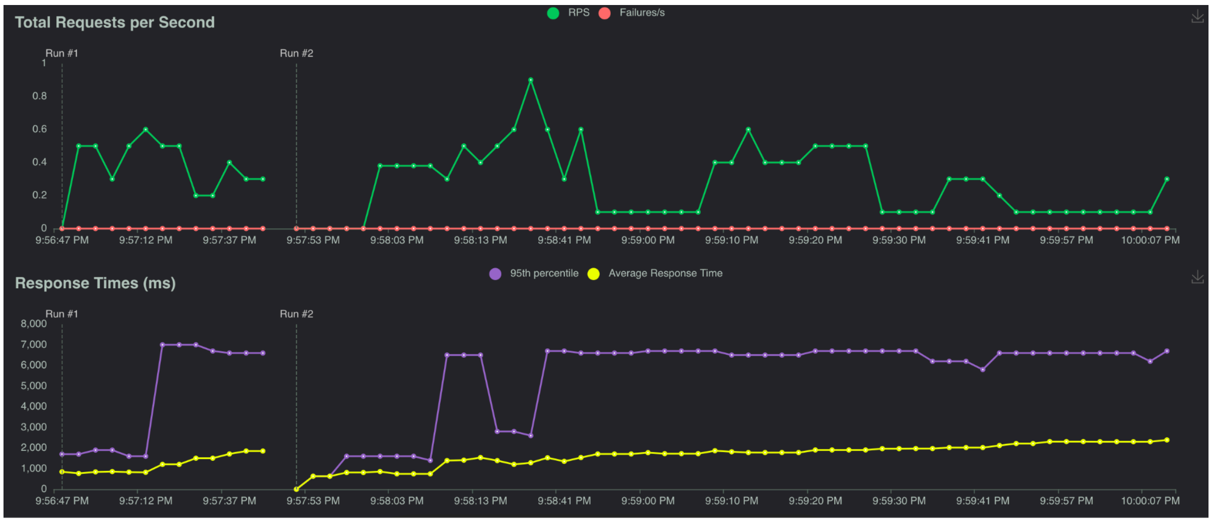


3) High Traffic

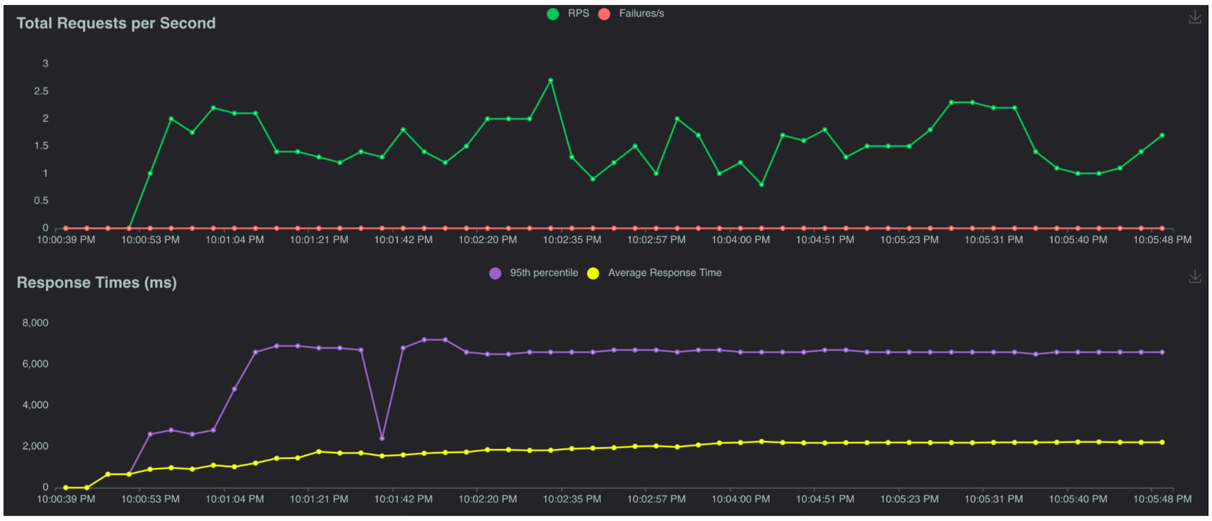


c) High System Configuration

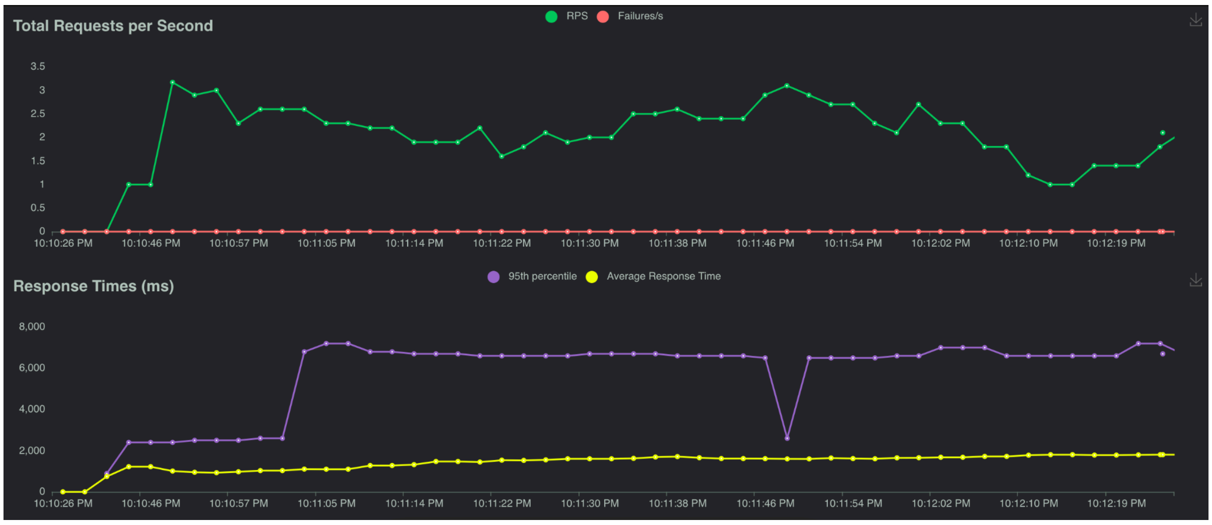
1) Low Traffic



2) Medium Traffic



3) High Traffic



6) <u>Benchmark Discussion</u>

Based on the testing parameters shared in Section 3.a, above test graphs were acquired for three configurations. Under the conditions of low system configurations, it can be seen that average response times increased up to a maximum of 12000 ms going through the low traffic to high traffic. This significant increase in response time is a direct consequence of the system's limited resources. It could be figured out that the average response time is higher than the other configurations as the elasticity of the system is lower due to the low amount of predetermined replicas of the pods under the low configuration. With fewer replicas, the system's ability to handle parallel requests diminishes, leading to higher latency.

It could be understood that more lack of resource issues can be encountered due to the low number of replicas per pod under each node. This lack of resources causes the system to struggle with workload distribution, further aggravating latency issues. This issue would result in the latency problem which can be observed on the graph. The graph clearly demonstrates that as the system's configuration advances, the average response time decreases. It is indicated that the system is delayed on average in responding in the expected way when operating under low configurations.

It can be seen that as you move from low configuration to high, the system responds faster on average and latency diminishes. This correlation highlights the system's dependency on adequate resource allocation. These results, which are exactly as we expected, reveal in a numerical context the importance of organizing the system in an elastic and scalable manner. Proper resource allocation and scaling strategies are crucial for maintaining optimal performance and minimizing response times.

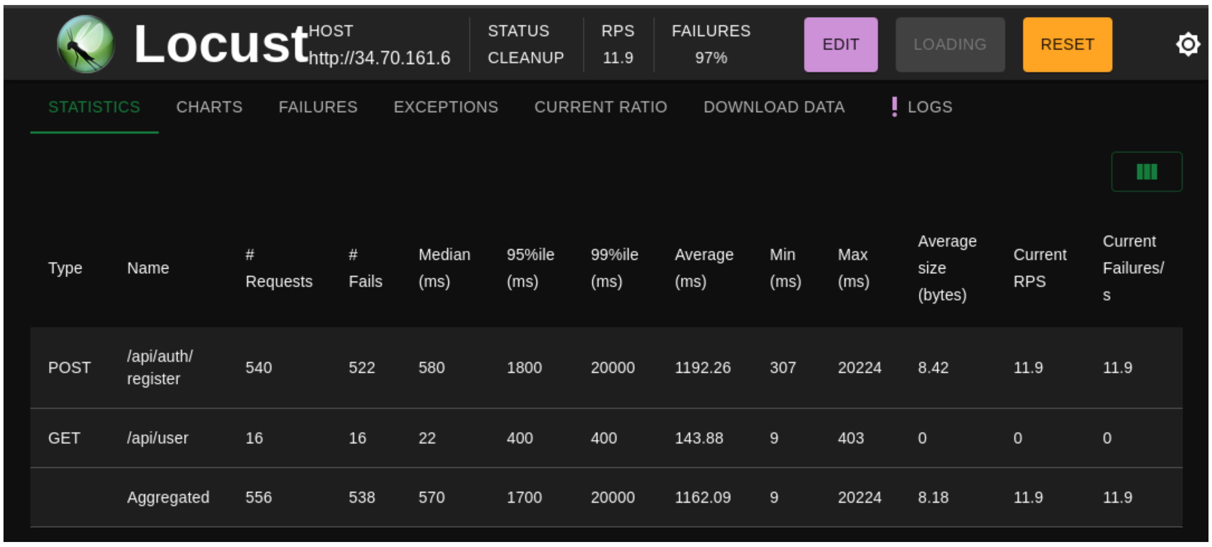
The response time results can also be seen in the Appendix 1 where each configuration option and each traffic option properly represented which bar charts. Appendix 1 is a clear bar-chart visualization of the graph results shared under the Benchmark Results.

Appendix 2 describes the hardware utilization throughout the benchmarks. We can see a clear trend that our application gets more and more computation bounded rather than hardware bounded. This is a necessary condition for scalability as one can configure their configurations based on their expected amount of responses. For example if the chat application is designed for the communication of select individuals going with the Low Configuration would be optimal. On the other hand, if the application was to be deployed globally, we would need much better configurations to elevate the load.

The most of the computational load is due to reverse proxying as it is consistently orchestrating the responses with the other containers. This is a CPU-bound process. The next container, ExpressJS, has considerable amounts of memory usage as it is consistently interacting with the Mongo database. The operations on the Express APIs are mostly related to persistent information about the users, chatrooms, and chats. This makes the request much more heavier as the number of users increases. The last bottleneck is the Socket.IO container.

The main drawback of this module is that it should keep any online user connected as a user can receive a message at any given time.

We have witnessed some real life consequences of under provisioned deployments when we were designing our test module. The lower configured nodes were using their resources not to serve the users but to send timeouts and error messages. This is eliminated with higher configurations of hardware. As seen in Appendix 1 higher configuration gives the users a better overall experience and Appendix 2 shows why that is the case. An example of an under provisioned node with high traffic is given below

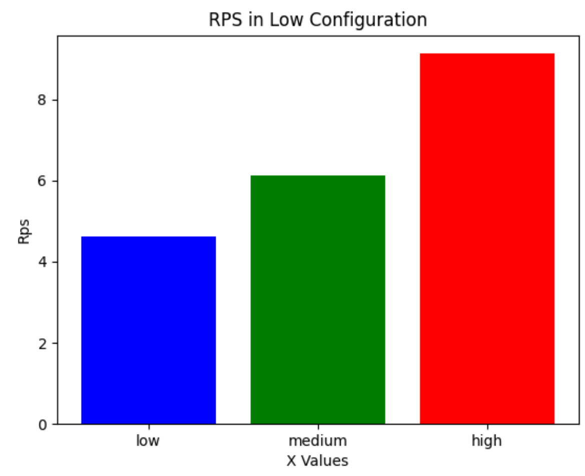


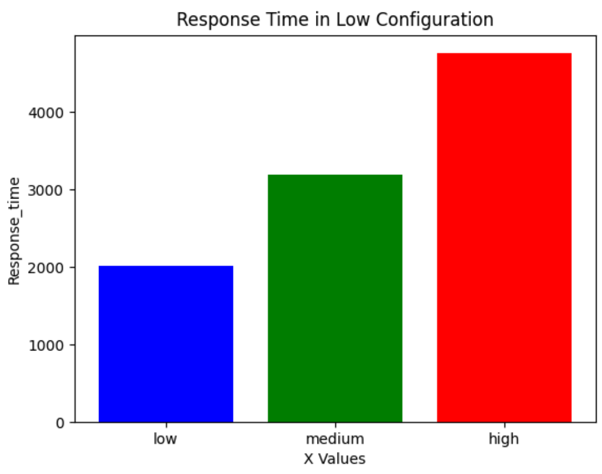
Type	Name	# Requests	# Fails	Median (ms)	95%ile (ms)	99%ile (ms)	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	Current RPS	Current Failures/s
POST	/api/auth/register	540	522	580	1800	20000	1192.26	307	20224	8.42	11.9	11.9
GET	/api/user	16	16	22	400	400	143.88	9	403	0	0	0
	Aggregated	556	538	570	1700	20000	1162.09	9	20224	8.18	11.9	11.9

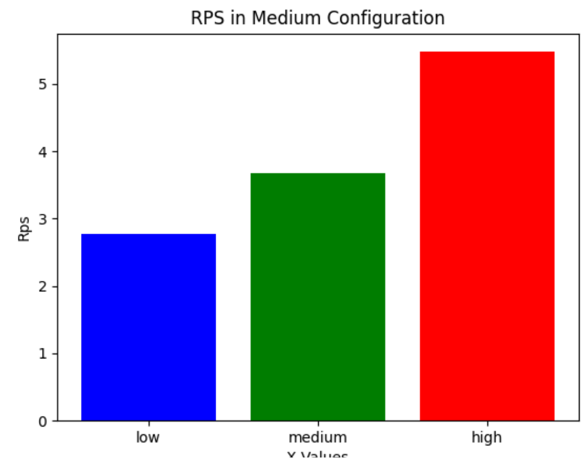
A Low Configuration with High Traffic, 10000 concurrent users with spawn rate of 50

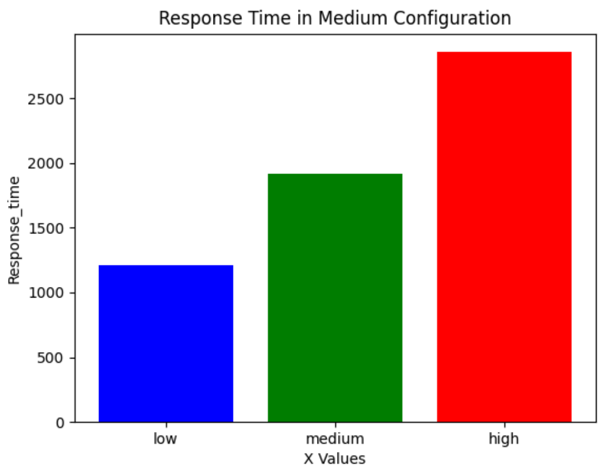
Appendix 1

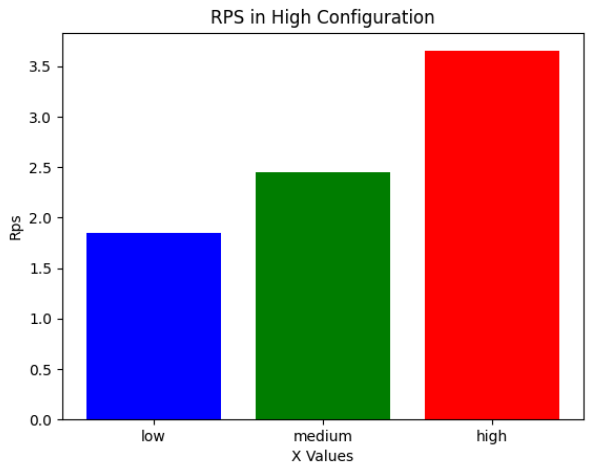
RPS and Response Times with different configurations

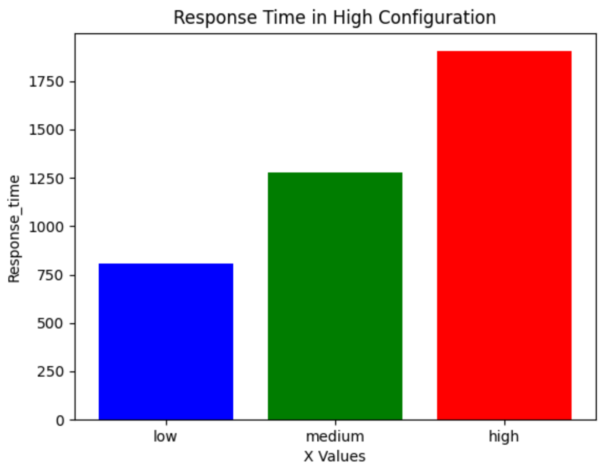






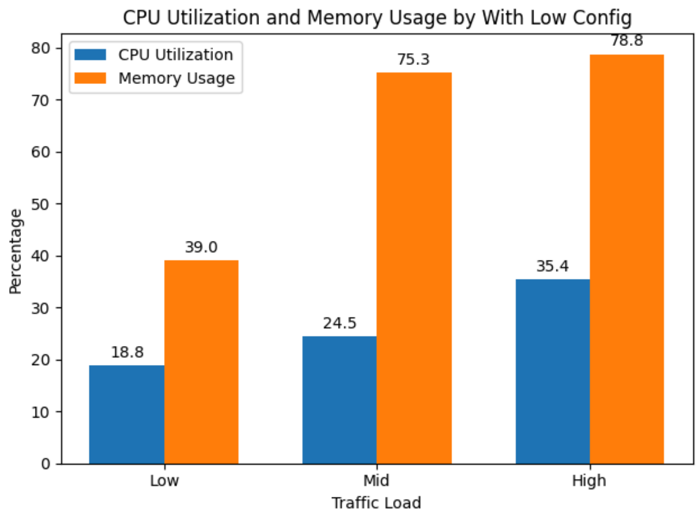


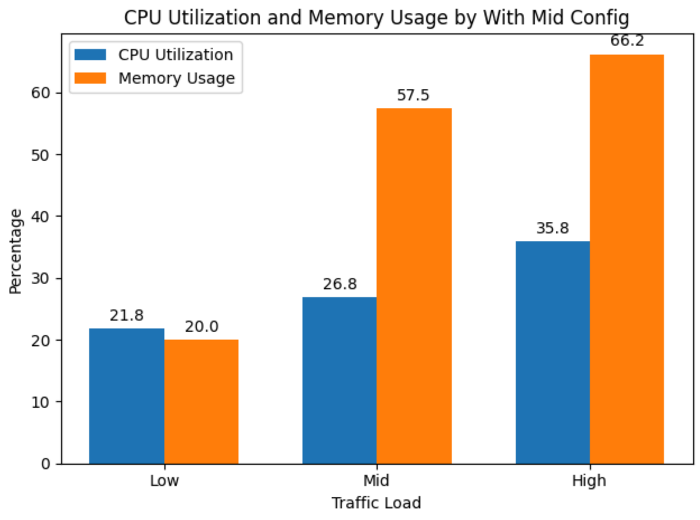


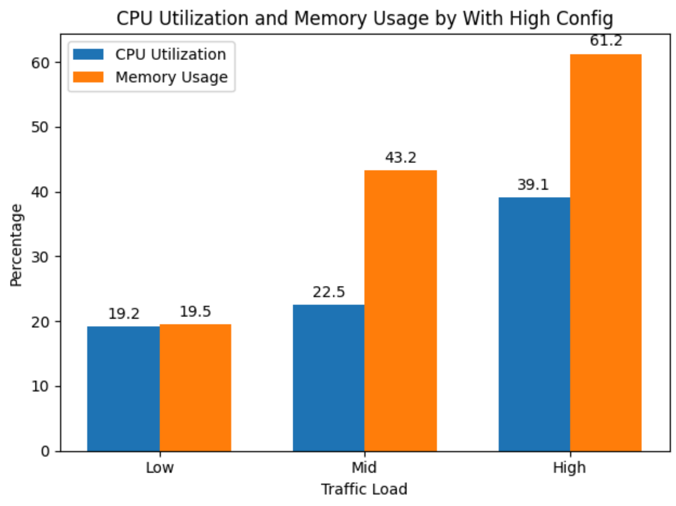


Appendix 2

CPU Utilization and Memory with different configurations


CPU Utilization and Memory Usage by With Low Config
CPU Utilization
Memory Usage
Percentage
Traffic Load
Low
Mid
High


CPU Utilization and Memory Usage by With Mid Config
CPU Utilization
Memory Usage
Percentage
Traffic Load
Low
Mid
High


CPU Utilization and Memory Usage by With High Config
CPU Utilization
Memory Usage
Percentage
Traffic Load
Low
Mid
High