CS-436 Cloud Computing

Term Project Final Report

Group 2

Group Members:

Ata Ernam

Eren Yiğit Yaşar

Ahmet Burak Kurtulmuş

Arda Güney

Instructor:

Bahri Atay Özgövde

Date:

5/25/2024

Table Of Content:

1. Architecture Design

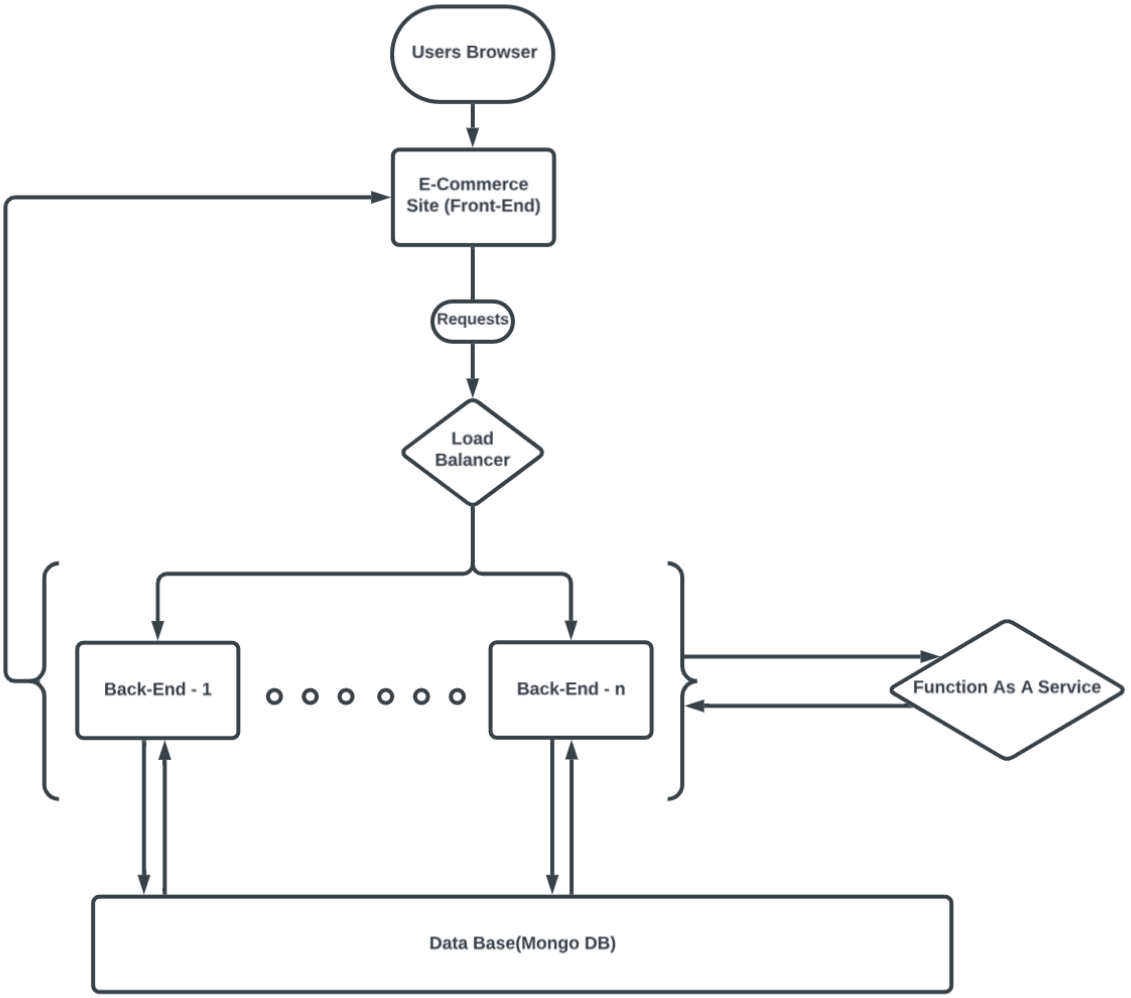

Figure 1.1 : Architecture Design

Through the scope of the whole project we worked on an e-commerce site application built by Node.Js and React. The app architecture was based on the

separation of the front-end and a back-end side. Thus two of these sides were running separately. By the light of this behaviour we also created our cloud architecture on Google Cloud Platform based on this. Firstly we separately configure and run our front-end and back-end sides of our application on different virtual machines on Google Cloud Platform. Secondly we connected our back-end side to the MongoDB database by using Mongo-DB Atlas. By this simple build we created the core of our cloud architecture.

Afterwards, to experiment by auto scaling, load balancing, server-less functions and stress testing we expanded our architecture. At first, we created our virtual machines for back-end by an instance group. Thus we could have multiple virtual machines that can be used for running our back-end to manage requests taking in consideration of the changing work-load. In order to run our application and have proper auto scaling and load balancing without manually building and running our application we added a startup script to our instance group. By that each newly generated virtual machine by the instance group would directly build and run our applications backend side and manage the requests. Secondly we added a load balancer by using the provided Load Balancing Tool of Google Cloud Platform. Thirdly we added server-less functions that would handle back-end requests by using Google Cloud Platforms, Cloud Functions tool. At last we had an architecture that would take requests from the user's browser, front-end side of the application that would send the requests to the load balancer of our cloud system. In continuation this load balancer would divide the requests to auto-scaled virtual machines and those virtual machines that handle back-end would trigger the

serverless functions when needed in our cloud system. Finally these requests would be handled by the back-end side and the necessary database Mongo DB connection thus, database related implementations would be done. Afterwards the whole application would be updated through the changes at database back to the front-end side to the user.

2. Experiment Design

In order to do experimentation on the cloud system we built, there were a couple of necessities to fulfil. Firstly we need a system that has an architecture complex enough to do different and considerable amounts of changes to observe its behaviour. To accomplish this we added auto scaling and load balancing by instance groups and Load Balancer tool in addition to a serverless function by Cloud Functions tool. Afterwards it was needed to identify the parameters to change on our system to experiment on. The parameters of virtual machines, load balancing tools and our stress tests gave us enough to experiment on. Finally by doing stress tests and observing its results after doing changes on several parameters of our system we did our experimentation.

a) System Parameters

The parameters of virtual machines, load balancing tool and our stress test tool were the parameters we used for our experiments. If we mention all them consequently starting with virtual machines, the parameters of region and machine type were the ones. Secondly for the load balancing tool of Google Cloud, maximum backend utilisation, maximum rps, scope connection draining time out, locality load balancing policy. Finally the number of users and ram up parameters of our stress test tool will be used.

b) Experimentation Tool

For an experimentation tool to do stress tests and send the requests to necessary endpoints we used Locust. By having a locust file and running it on a separate virtual machine we did the stress tests. In our locust file we both created code pieces to send api requests to our serverless function from our backend side and directly to our applications backend side. Thus by the single locust file with each test we did we were able to trigger our serverless function too. The parameters number of users and ramp up were the ones we used by locusts to do experiments on.

3. Discussion Of Results

For discussion of results we will consequently discuss the results of the experiments on the parameters of virtual machines, instance groups & load balancer and our test tool locust. For each section we will have a default set of parameters to do comparisons.

- **Discussion Of The Results On Virtual Machine Parameters**

  Our parameters to change are region and type of virtual machines for this step. Our default parameters are "*me-west-1c*" and "*medium*". We would have 3000 users and 150 ramp up for each locust test. All tests were run for intervals of 10 minutes.

❖ **Results Of "me-west-1c" and "medium":**

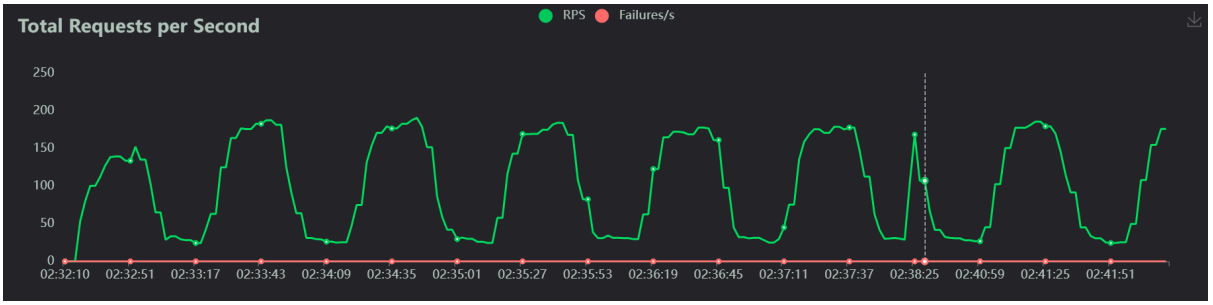

Figure 3.1 : "me-west-1c" and "medium", total requests per second

RPS at the end: 176.2

Failures/s at the end: 0

Total Failures at the end: 0%

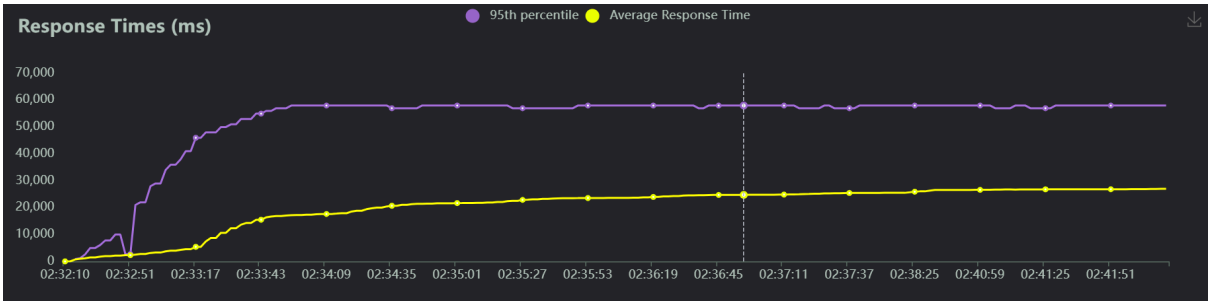

Figure 3.2 : “me-west-1c” and “medium”, response times

95th percentile at the end: 58000

Average Response Time at the end: 26992.91

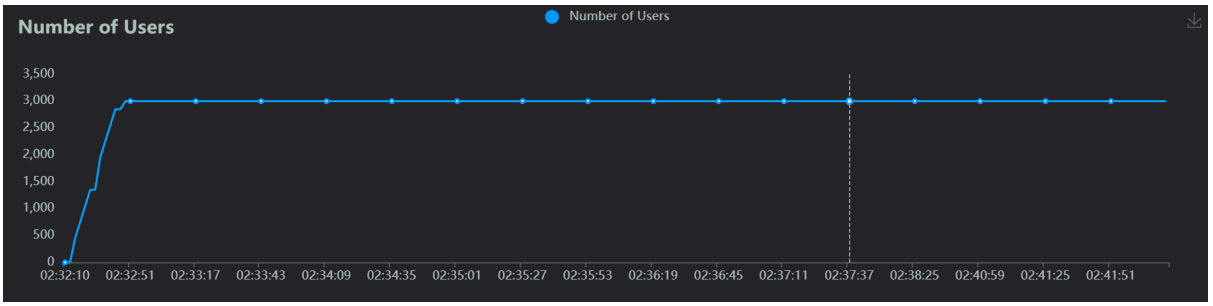

Figure 3.3 : “me-west-1c” and “medium”, number of users

- **Results Of “me-west-1c” and “small”:**

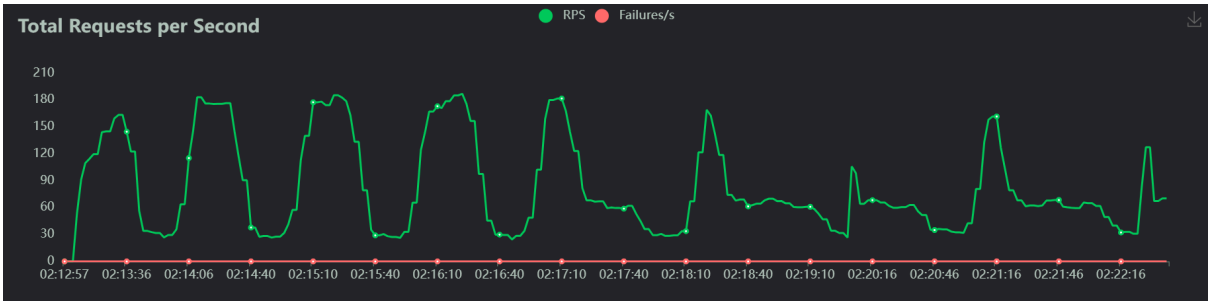

Figure 3.4 : “me-west-1c” and “small”, total requests per second

RPS at the end: 70.4

Failures/s at the end: 0

Total Failures at the end: 0%

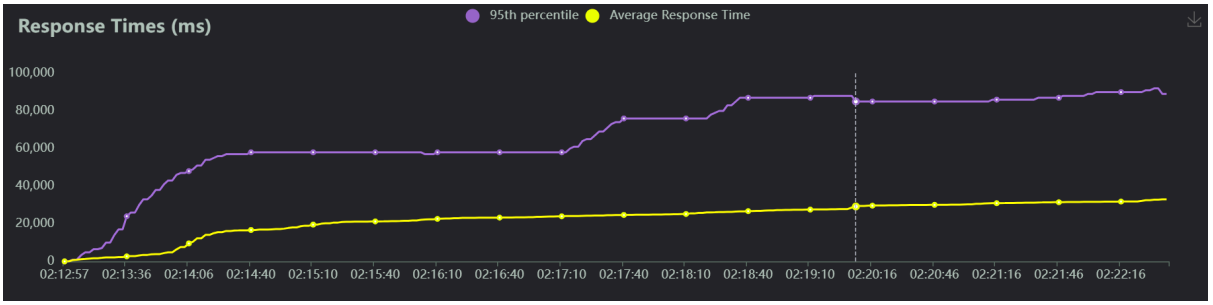

Figure 3.5 : "me-west-1c" and "small", response times

95th percentile at the end: 89000

Average Response Time at the end: 33014.73

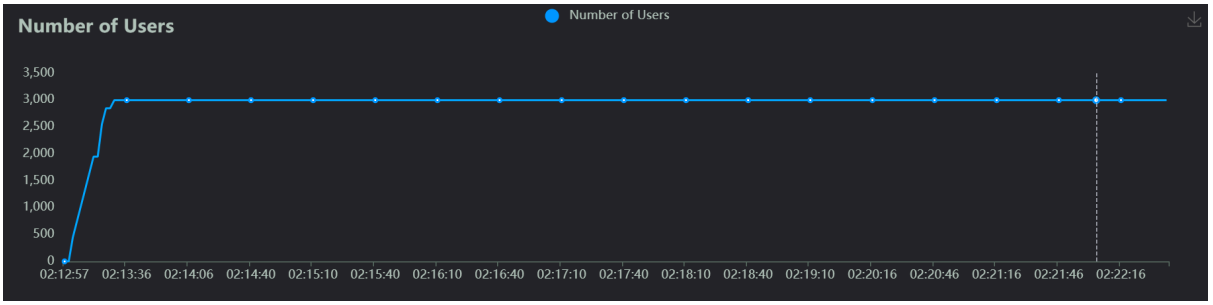

Figure 3.6 : "me-west-1c" and "small", number of users

- **Results Of "me-west-1c" and "micro":**

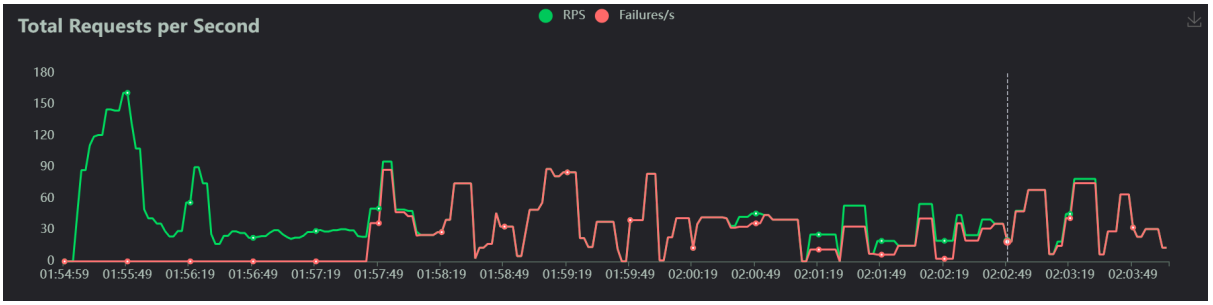

Figure 3.7 : “me-west-1c” and “micro”, total requests per second

RPS at the end: 13.1

Failures/s at the end: 13.1

Total Failures at the end: 63%

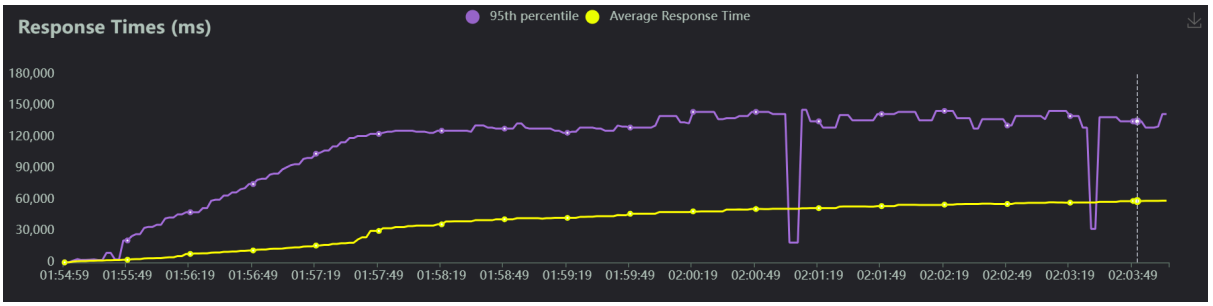

Figure 3.8 : “me-west-1c” and “micro”, response times

95th percentile at the end: 142000

Average Response Time at the end: 59056.94

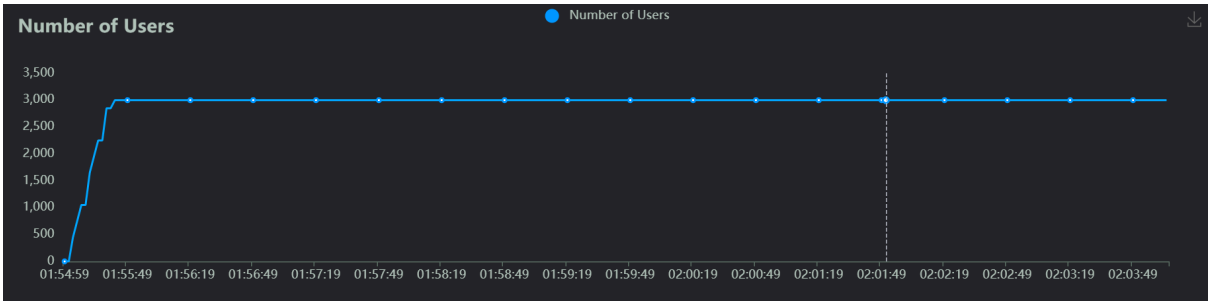

Figure 3.9 : “me-west-1c” and “micro”, number of users

By the results shown above we can conclude that the size of virtual machines directly has an effect on the success of the cloud system. As we see when the type of the virtual machine is set as micro the system can not handle the same amount of request with total success like it has done with a medium or small size of virtual machine. In addition, when we compare the medium and small sized virtual machines that both successfully handled all requests, it can be seen that a medium sized virtual machine had a lower average response time than its small sized version.

- **Results Of “usa-central-1c” and “medium”:**

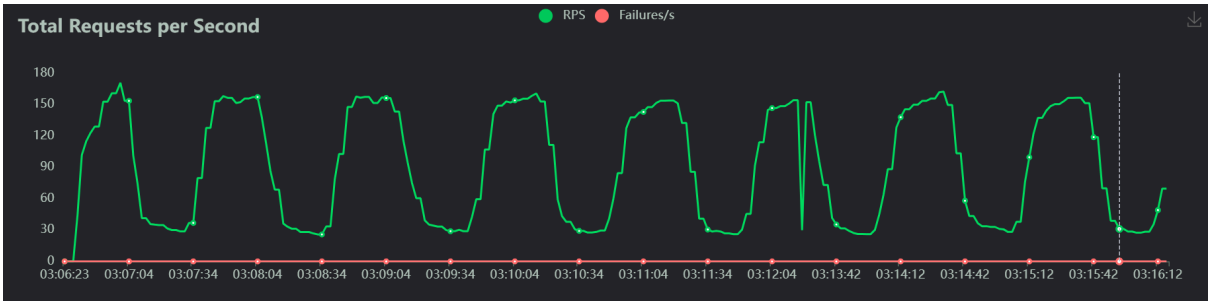

Figure 3.10 : “usa-central-1c” and “medium”, total requests per second

RPS at the end: 69.6

Failures/s at the end: 0

Total Failures at the end: 0%

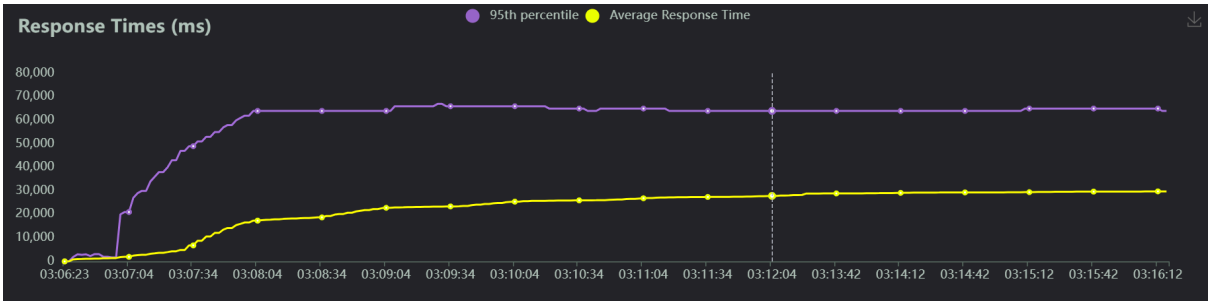

Figure 3.11 : "usa-central-1c" and "medium", response times

95th percentile at the end: 64000

Average Response Time at the end: 29773.98

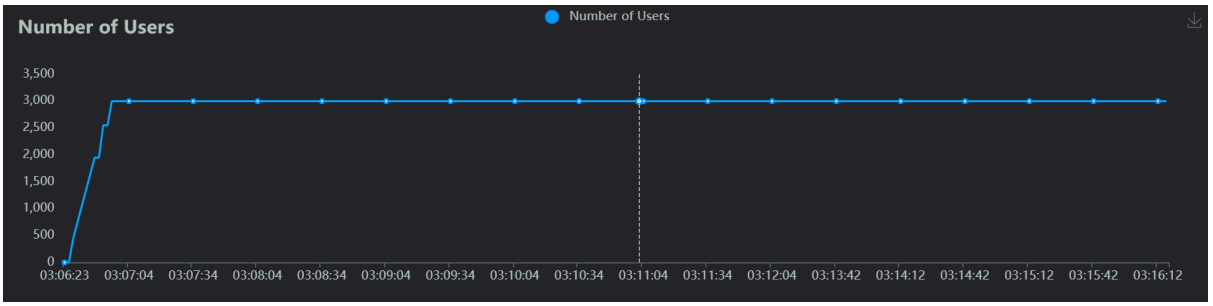

Figure 3.12 : "usa-central-1c" and "medium", number of users

By considering the medium virtual machines in our system location vise we can say that even the medium virtual machine located at Tel Aviv is closer to Turkey compared to the medium virtual machine in the USA there isn't a considerable difference. Performance of the virtual machine in Tel Aviv is slightly better but again it's not a notable difference.

- **Discussion Of The Results On Load Balancer Parameters**

Our parameters to change are time out, maximum backend utilisation, maximum rps, scope, instance group capacity, connection draining time out, locality load balancing policy for this step. Our initial parameters are "*me-west-1c*", "*medium*", "*timeout is 30*", "*maximum backend utilisation is 80*", "*maximum rps is 100*", "*scope is per instance*", "*instance group capacity is 100%*", "*connection draining timeout is 30*" and "*load balancing policy is Least Request*". All tests were run for minimum intervals of 10 minutes.

❖ **Results With Initial Parameters (What are initial configurations are explained above):**

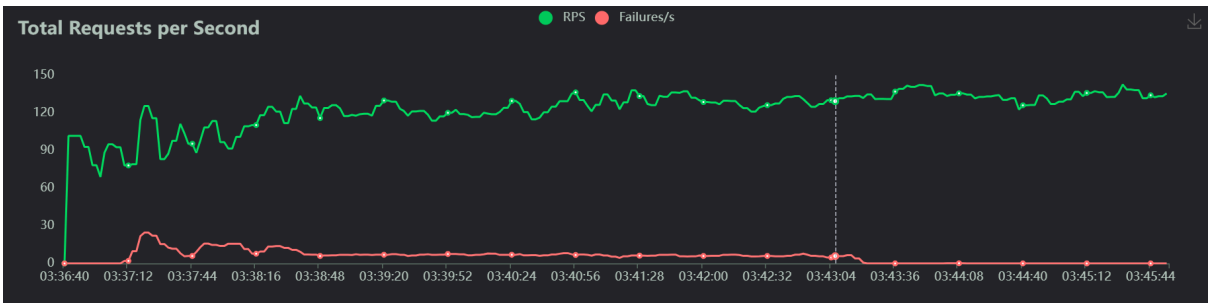

Figure 3.13 : Results with Initial Parameters, total requests per second

RPS at the end: 135.5

Failures/s at the end: 0

Total Failures at the end: 4%

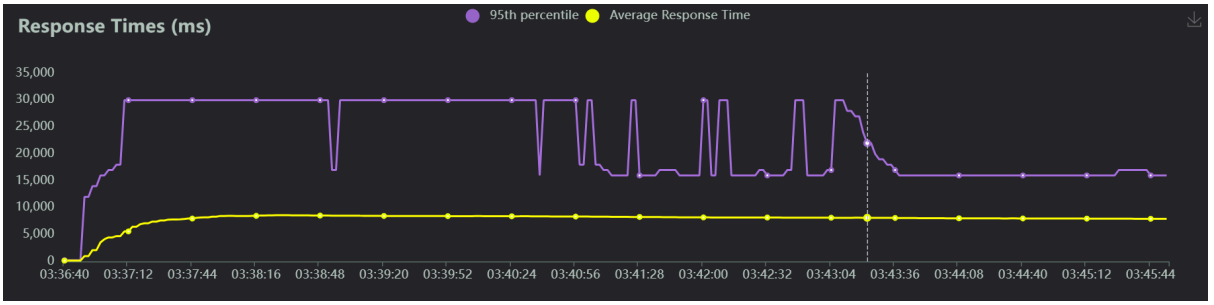

Figure 3.14 : Results with Initial Parameters, response times

95th percentile at the end: 16000

Average Response Time at the end: 7915.67

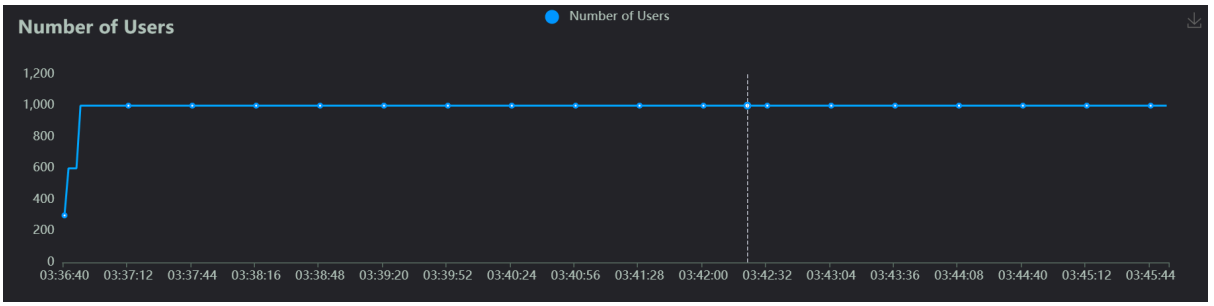

Figure 3.15 : Results with Initial Parameters, number of users

❖ **Results Of When Time Out Is 60 Seconds (All others remain as initial configurations explained above):**

➢ Note: Time out is how long to wait for the backend service to respond before considering it a failed request.

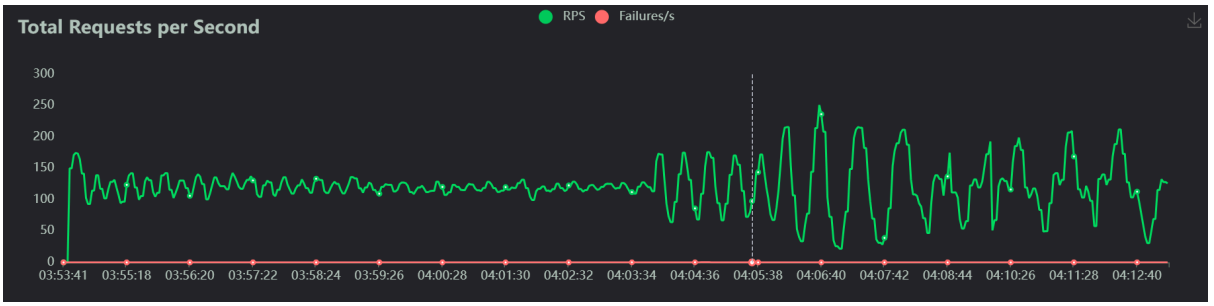

Figure 3.16 : Results Of When Time Out Is 60 Seconds, total requests per second

RPS at the end: 126

Failures/s at the end: 0.1

Total Failures at the end: 0%

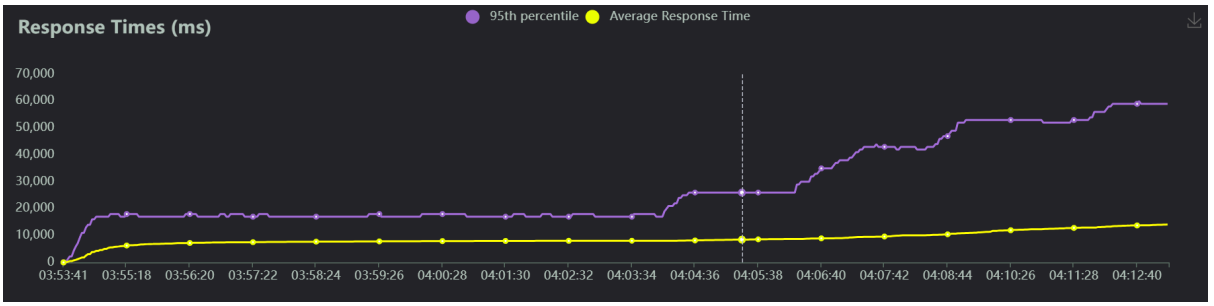

Figure 3.17 : Results Of When Time Out Is 60 Seconds, response times

95th percentile at the end: 59000

Average Response Time at the end: 14139

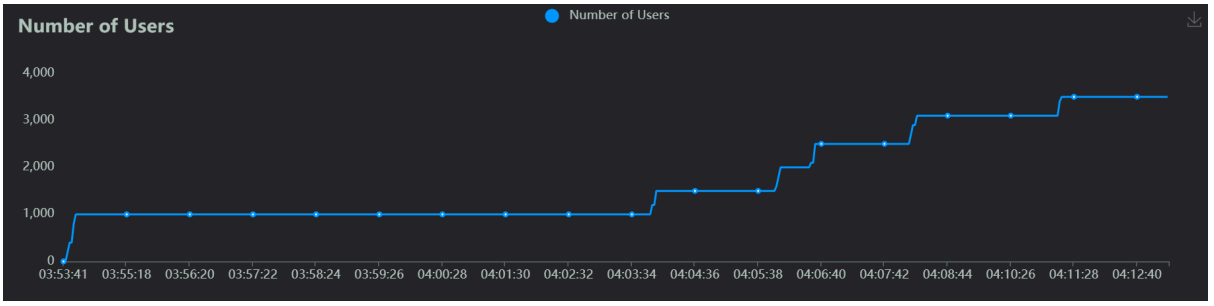

Figure 3.18 : Results Of When Time Out Is 60 Seconds, number of users

- When we increased the time out to 60 seconds we didn't face the failures we got. Moreover we can even be able to expand the number of users up to 3500 and RM as 150 which is more than the initial value of 1000 users and RM as

100. It is shown above that with the default values the system starts to give failures up to 4% percent with lower number of users and RM. In addition the initial values were the edge values for the system to work properly even 1500 users with the initial values would give near 40% errors. This clearly shows that increasing timeout value helps the system.

NOTE: After finding out that making time out as 60 gives us richer test experiments with more capacity, every test below has timeout value of 60 seconds.

- **New Default State (Updating timeout as 60 seconds, number of users as 3000 and RM as 150. All other parameters are the same as the initial mentioned above. All test below would be compared to this result.):**

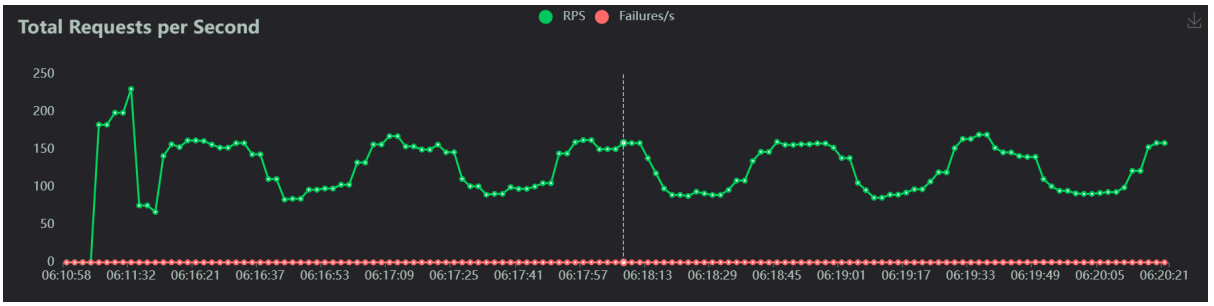

Figure 3.19 : New Default State , total requests per second

RPS at the end: 158.7

Failures/s at the end: 0

Total Failures at the end: 0%

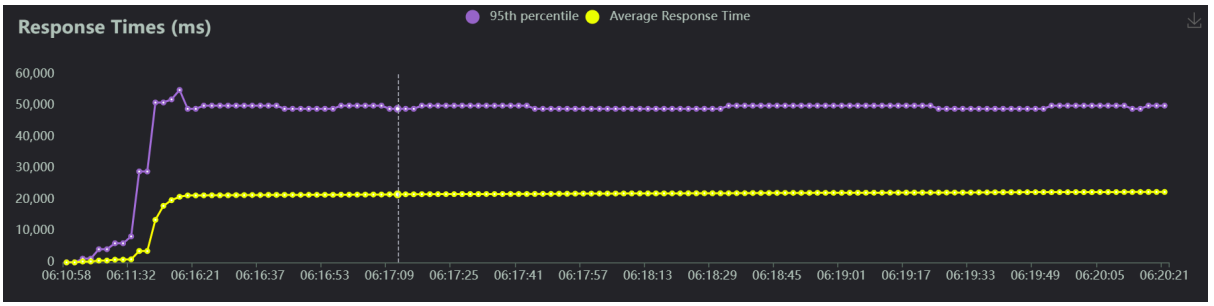

Figure 3.20 : New Default State , response times

95th percentile at the end: 50000

Average Response Time at the end: 22469.58

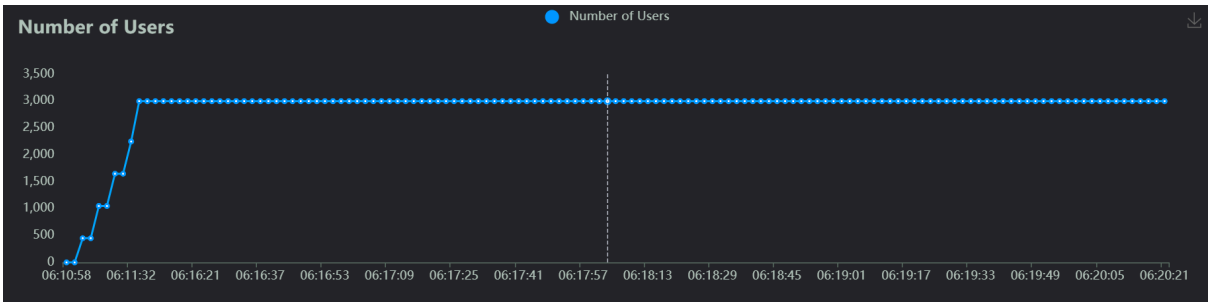

Figure 3.21 : New Default State , number of users

❖ **Results Of When Maximum Backend Utilisation Is 40 (Every other parameter than Maximum Backend Utilisation New Default State):**

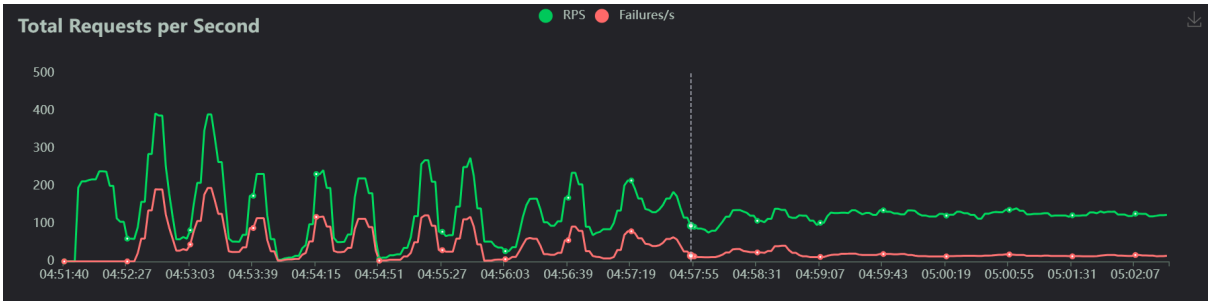

Figure 3.22 : Results Of When Maximum Backend Utilisation Is 40 , total requests per second

RPS at the end: 123.5

Failures/s at the end: 15.1

Total Failures at the end: 27%

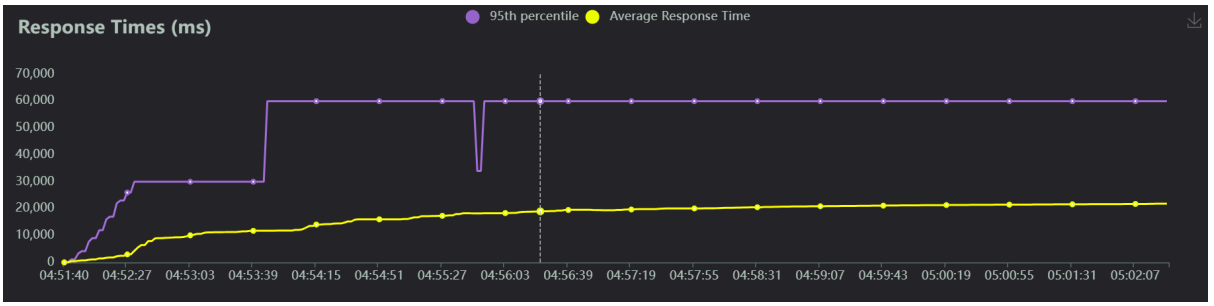

Figure 3.23 : Results Of When Maximum Backend Utilisation Is 40 , response times

95th percentile at the end: 60000

Average Response Time at the end: 21875.36

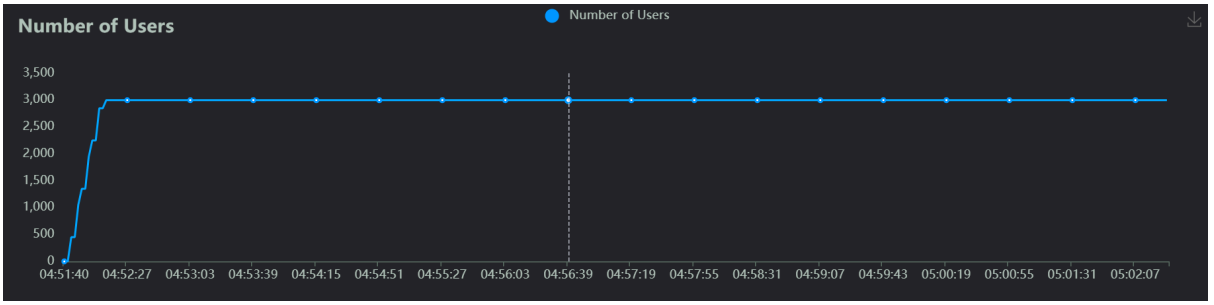

Figure 3.24 : Results Of When Maximum Backend Utilisation Is 40 , number of users

- When we decrease the maximum backend utilisation to 40% percent we observed a lot of failures. The errors were 504 Gateaway Timeout and 503 Server Errors. It is clearly that decreasing the maximum backend utilisation directly affects the server performance and lowers the performance.

❖ **Results Of When Maximum RPS Is 50 (Every other parameter than Maximum RPS are same with New Default State):**

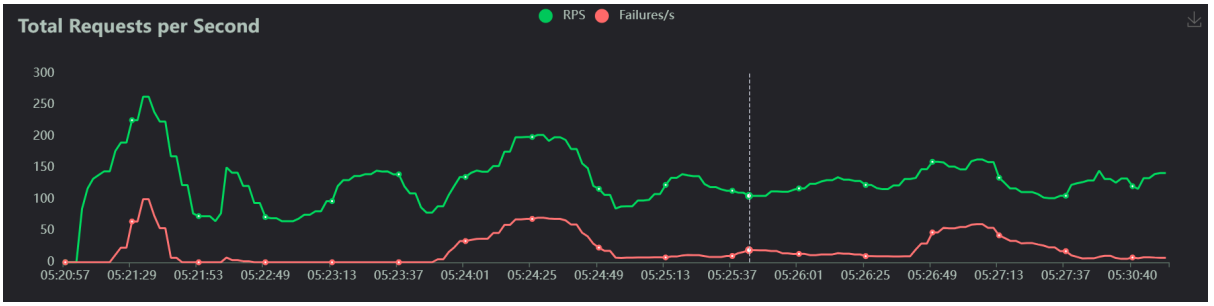

Figure 3.25 : Results Of When Maximum RPS Is 50 , total requests per second

RPS at the end: 141.8

Failures/s at the end: 7.2

Total Failures at the end: 12%

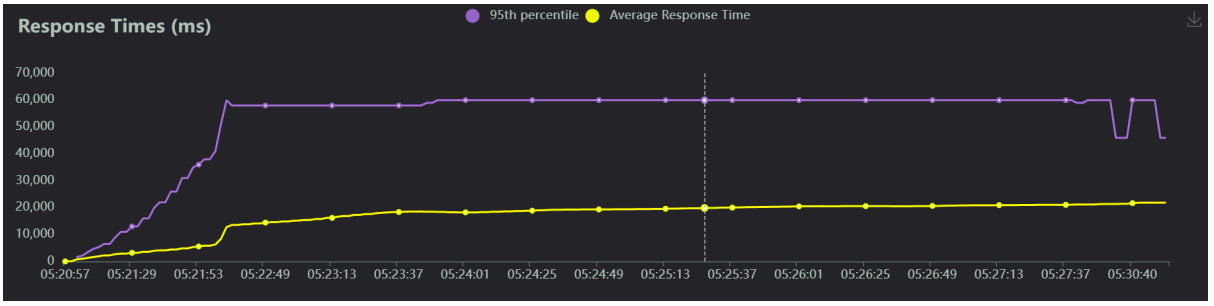

Figure 3.26 : Results Of When Maximum RPS Is 50 , response times

95th percentile at the end: 46000

Average Response Time at the end: 21844.24

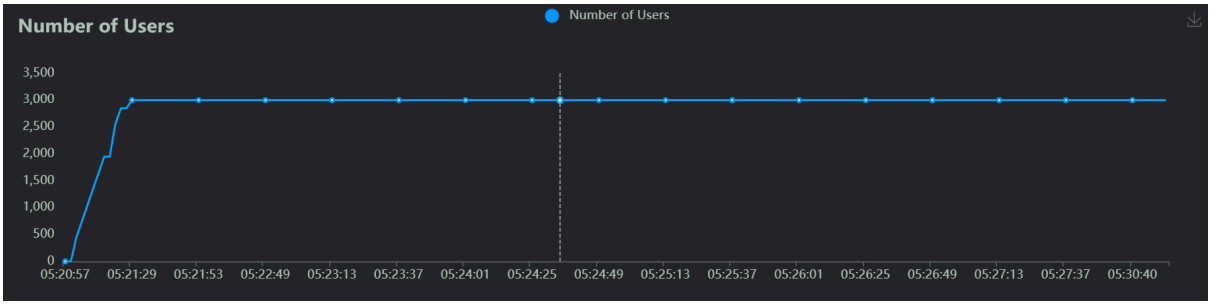

Figure 3.27 : Results Of When Maximum RPS Is 50 , number of users

❖ **Results Of When Maximum RPS Is 200 (Every other parameter than Maximum RPS are same with New Default State):**

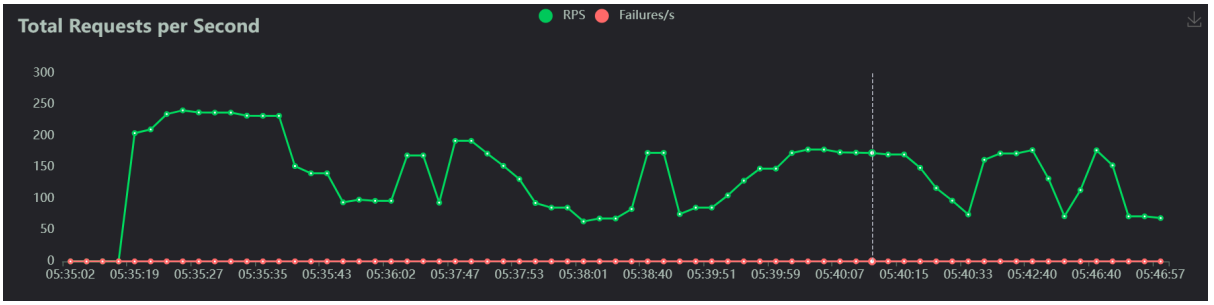

Figure 3.28 : Results Of When Maximum RPS Is 200 , total requests per second

RPS at the end: 69.1

Failures/s at the end: 0

Total Failures at the end: 0%

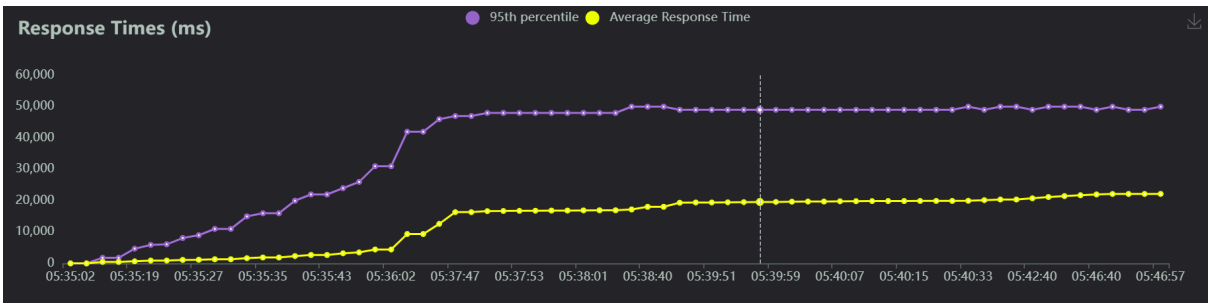

Figure 3.29 : Results Of When Maximum RPS Is 200 , response times

95th percentile at the end: 50000

Average Response Time at the end: 22163.7

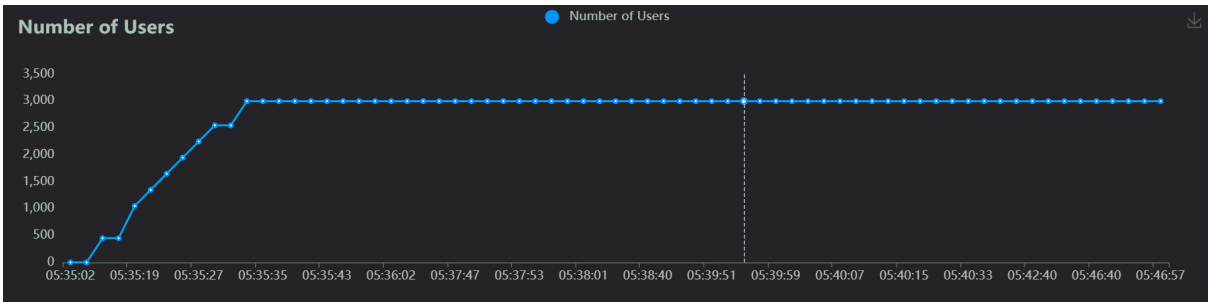

Figure 3.30 : Results Of When Maximum RPS Is 200 , number of users

As we see from the result, RPS value has a direct impact on the performance of our system. But from the results we got it can be stated that if the RPS value is over a certain threshold it does not improve the performance. When RPS is 200 the system behaves appropriately the same as the system that has a 100 RPS. On the other hand when the RPS value is decreased to 50 the system starts to give a considerable amount of failed requests.

❖ **Results With Scope as Per Group Instance (Every other parameter than Scope are same with New Default State):**

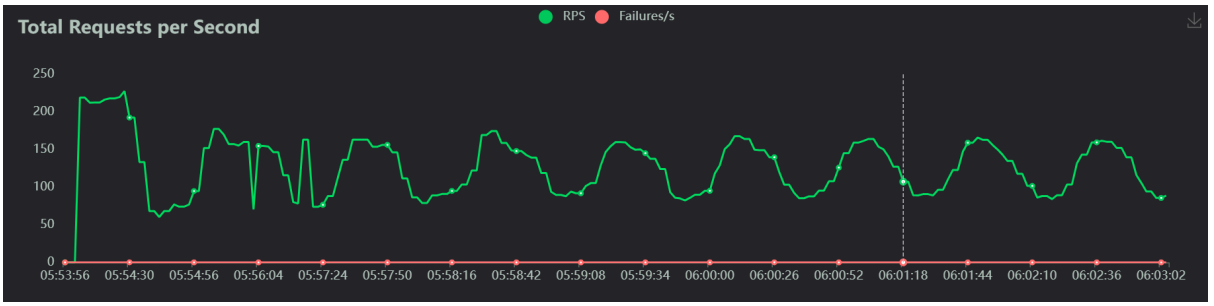

Figure 3.31 : Results With Scope as Per Group Instance , total requests per second

RPS at the end: 88.8

Failures/s at the end: 0

Total Failures at the end: 0%

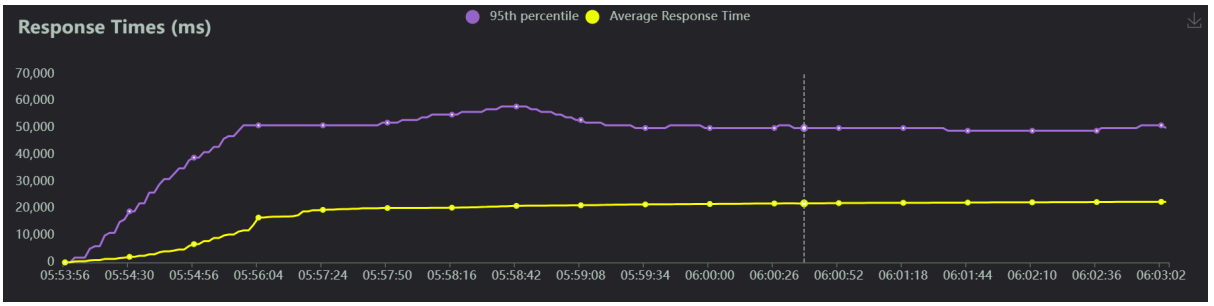

Figure 3.32 : Results With Scope as Per Group Instance , response times

95th percentile at the end: 50000

Average Response Time at the end: 22557.75

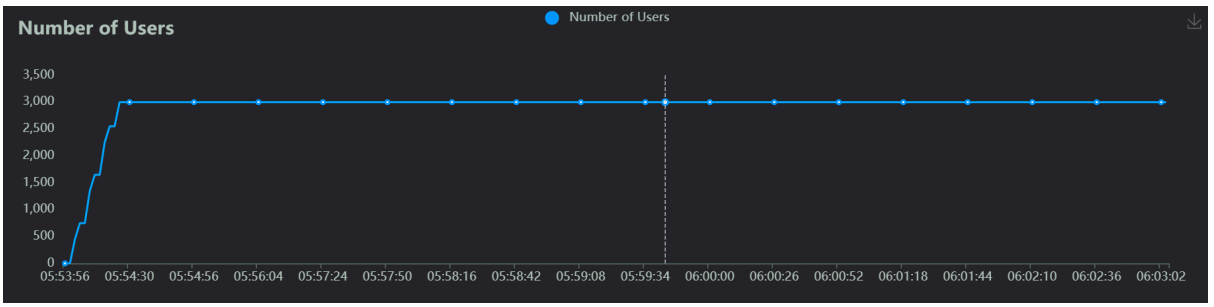

Figure 3.33 : Results With Scope as Per Group Instance , number of users

We haven't observed a noticeable difference in our system by changing this parameter.

- **Results With Instance Group Capacity Decreased to 50% (Every other parameter than Scope are same with New Default State):**

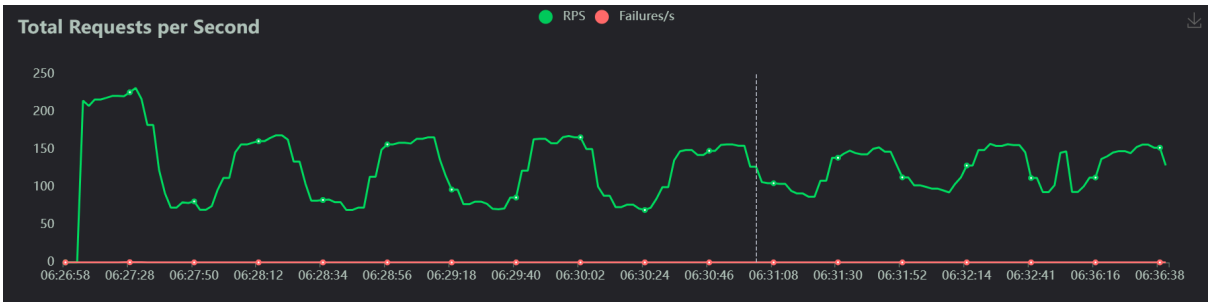

Figure 3.34 : Results With Instance Group Capacity Decreased to 50% , total requests per second

RPS at the end: 129

Failures/s at the end: 0

Total Failures at the end: 0%

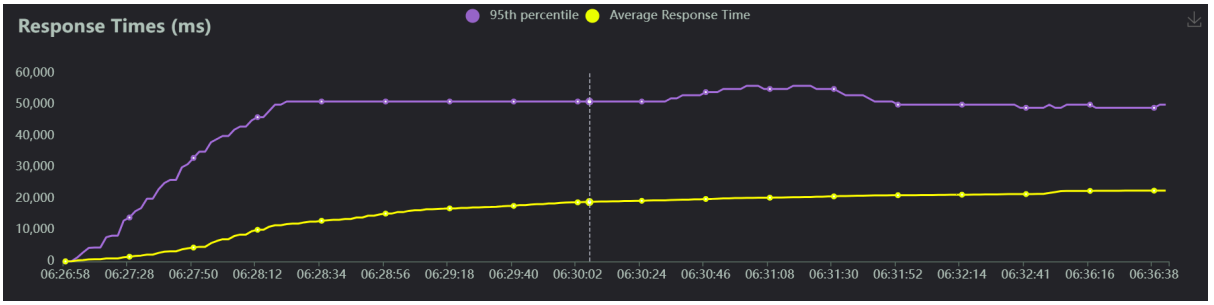

Figure 3.35 : Results With Instance Group Capacity Decreased to 50% , response times

95th percentile at the end: 50000

Average Response Time at the end: 22581

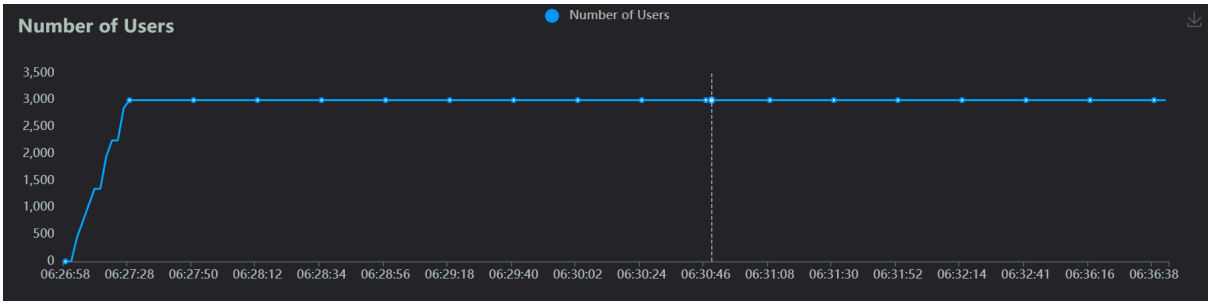

Figure 3.36 : Results With Instance Group Capacity Decreased to 50% , number of users

- **Results With Instance Group Capacity Decreased to 10% (Every other parameter than Scope are same with New Default State):**

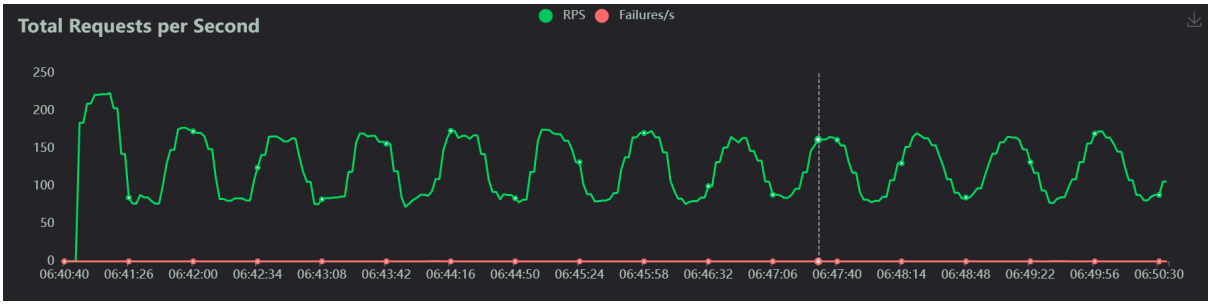

Figure 3.37 : Results With Instance Group Capacity Decreased to 10% , total requests per second

RPS at the end: 106.2

Failures/s at the end: 0

Total Failures at the end: 0%

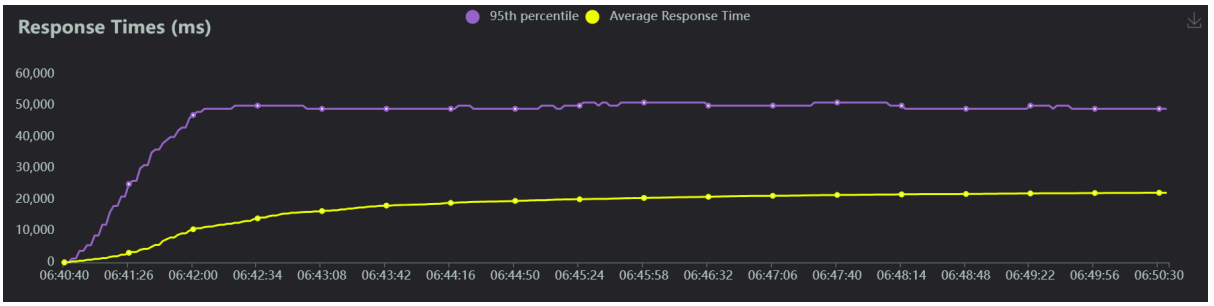

Figure 3.38 : Results With Instance Group Capacity Decreased to 10% , response times

95th percentile at the end: 49000

Average Response Time at the end: 22224.2

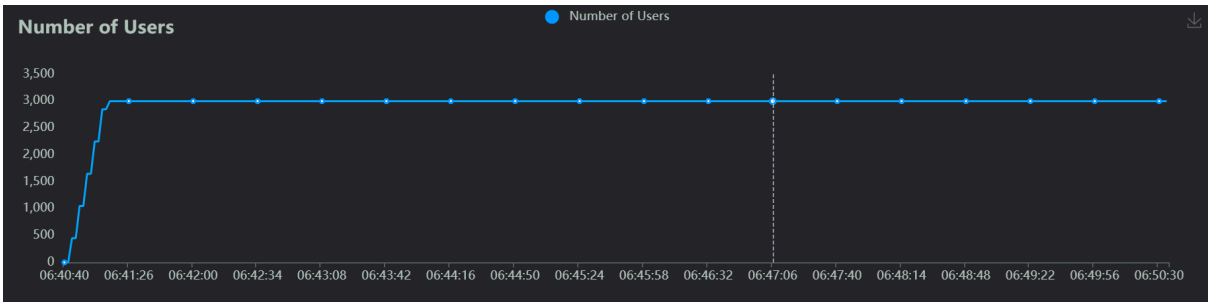

Figure 3.39 : Results With Instance Group Capacity Decreased to 10% , number of users

❖ **Results With Connection Draining Timeout Increased To 300 (Every other parameter than Scope are same with New Default State):**

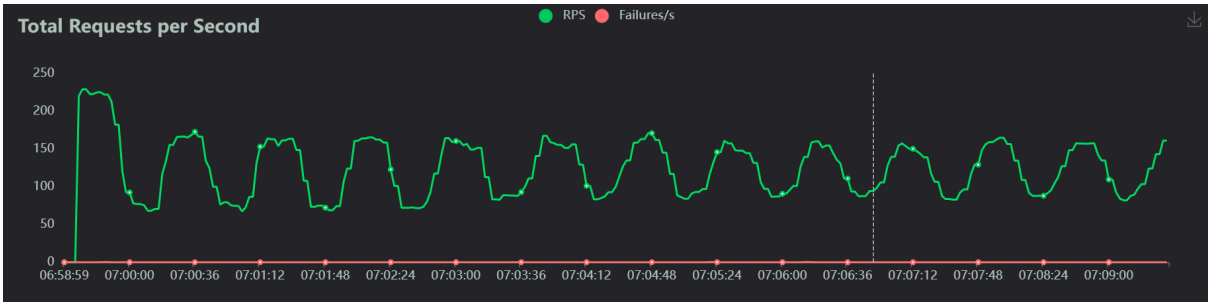

Figure 3.40 : Results With Connection Draining Timeout Increased To 300 , total requests per second

RPS at the end: 161.2

Failures/s at the end: 0

Total Failures at the end: 0%

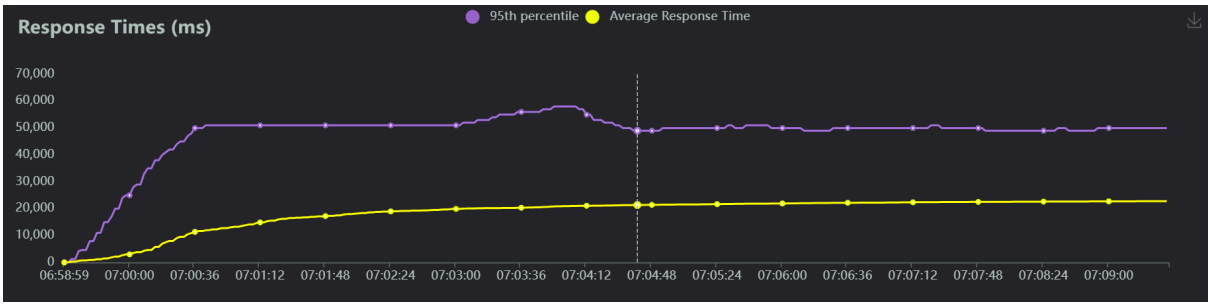

Figure 3.41 : Results With Connection Draining Timeout Increased To 300 , response times

95th percentile at the end: 50000

Average Response Time at the end: 22796.83

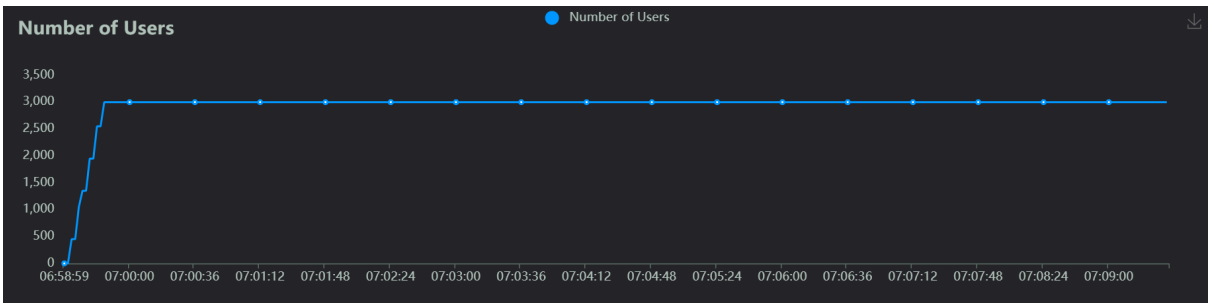

Figure 3.42 : Results With Connection Draining Timeout Increased To 300 , number of users

We haven't observed a noticeable difference in the system by changing this parameter. When it is 50 or 300 it lower or higher than the default value system behaves approximately similarly. No considerable change observed.

- **Results With Connection Draining Timeout Decreased To 3 (Every other parameter than Scope are same with New Default State):**

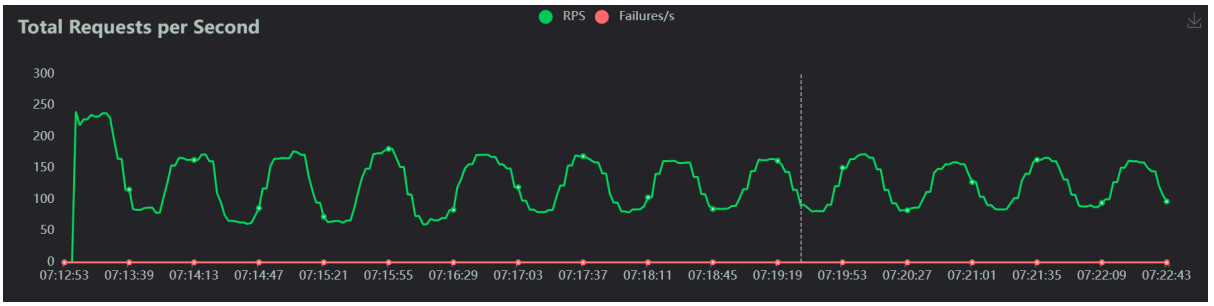

Figure 3.43 : Results With Connection Draining Timeout Decreased To 3 , total requests per second

RPS at the end: 97.2

Failures/s at the end: 0

Total Failures at the end: 0%

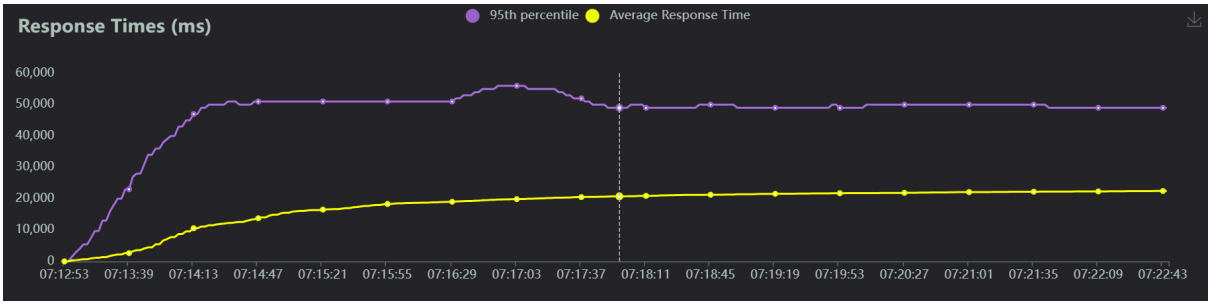

Figure 3.44 : Results With Connection Draining Timeout Decreased To 3 , response times

95th percentile at the end: 49000

Average Response Time at the end: 22463.06

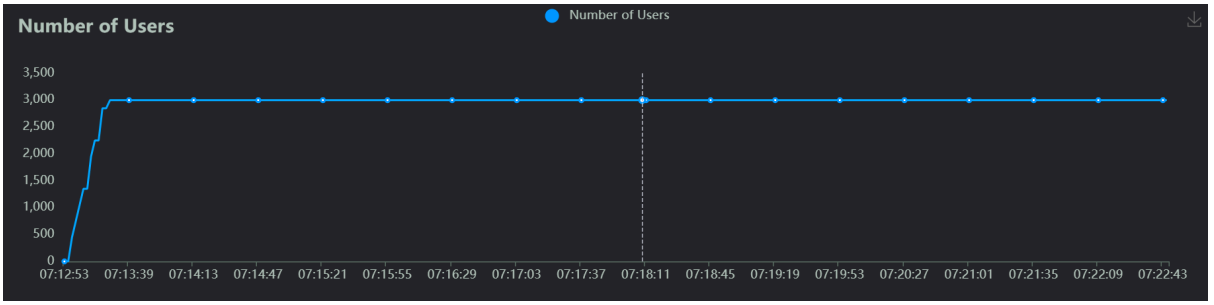

Figure 3.45 : Results With Connection Draining Timeout Decreased To 3 , number of users

We haven't observed a noticeable difference in the system by changing this parameter.

- **Results With Load Balancing Policy As Round Robin (Every other parameter than Scope are same with New Default State):**

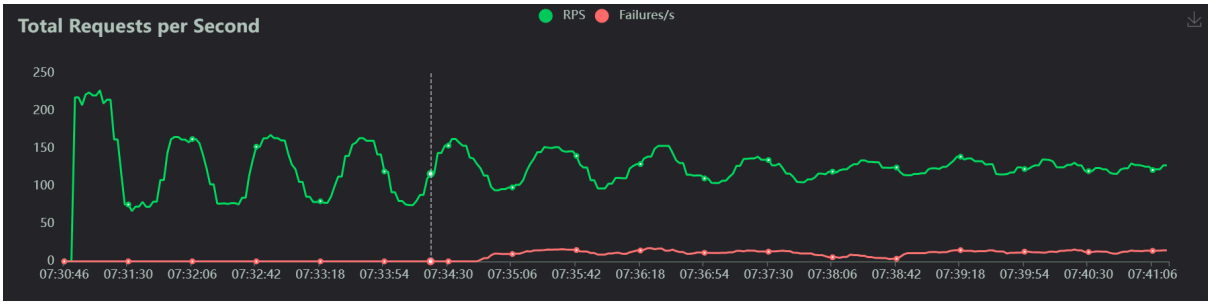

Figure 3.46 : Results With Load Balancing Policy As Round Robin , total requests per second

RPS at the end: 127.6

Failures/s at the end: 14.6

Total Failures at the end: 6%

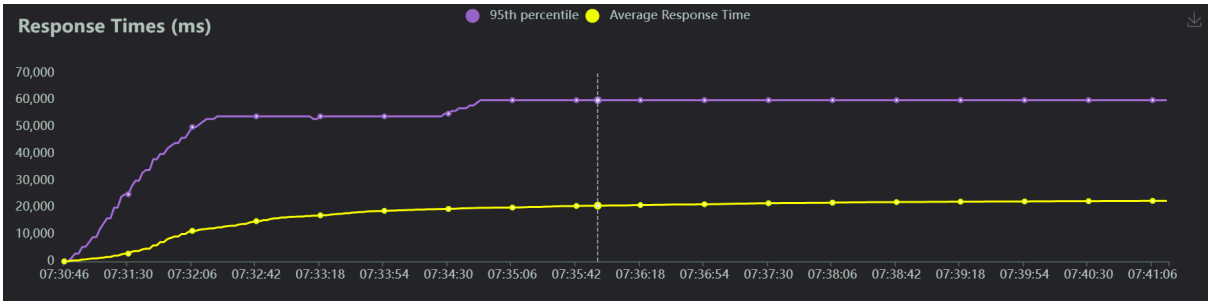

Figure 3.47 : Results With Load Balancing Policy As Round Robin , response times

95th percentile at the end: 60000

Average Response Time at the end: 22551.91

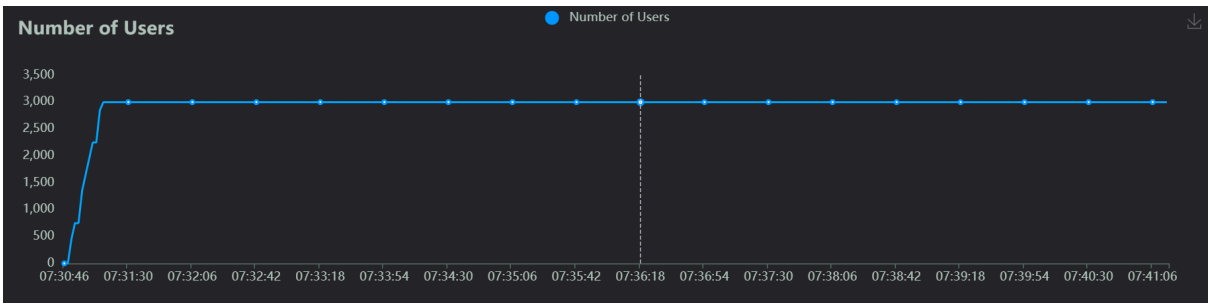

Figure 3.48 : Results With Load Balancing Policy As Round Robin, number of users

While using the Round Robin method we got http503 and http 504 errors. At the end 6% of the requests failed. On the other hand Least Request policy returned 0% error with the same user number and RM. In addition, the 95th percentile at the end increased by 10000 compared to the Least Request method but average response time stayed approximately similar.

- **Results With Load Balancing Policy As Maglev (Every other parameter than Scope are same with New Default State):**
- *Note: The load balancing algorithm used within the scope of the locality.Maglev has faster table lookup build times and host selection times, and can be a drop in replacement for the ring hash load balancer.*

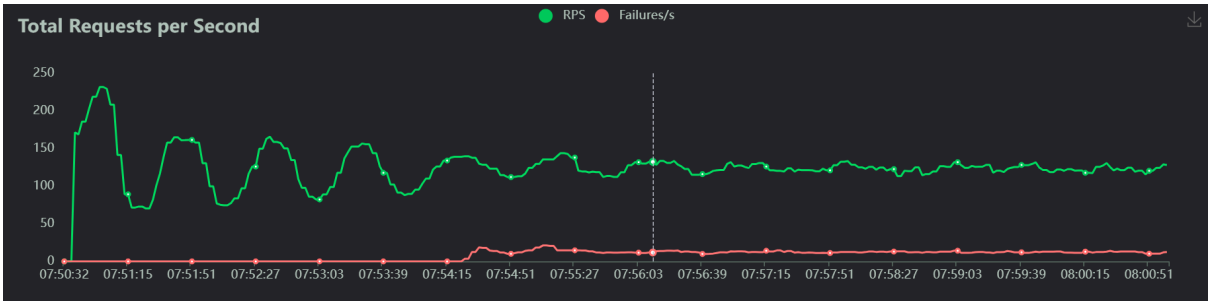

Figure 3.49 : Results With Load Balancing Policy As Maglev , total requests per second

RPS at the end: 128.1

Failures/s at the end: 12.04

Total Failures at the end: 7%

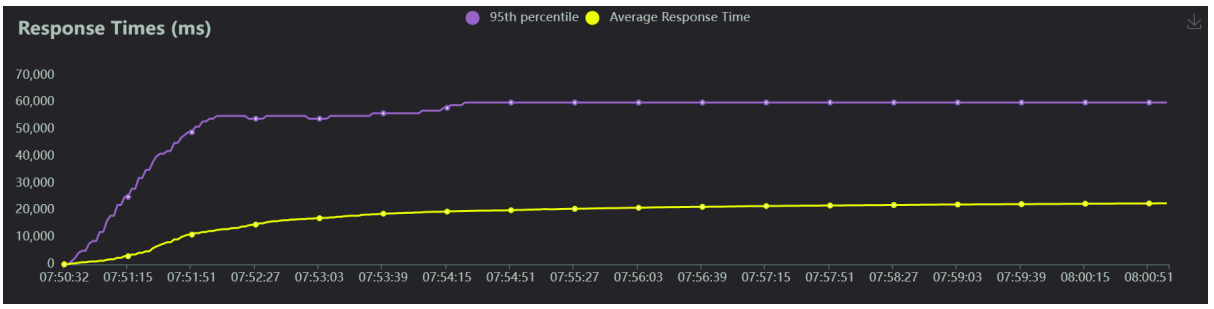

Figure 3.50 : Results With Load Balancing Policy As Maglev , response times

95th percentile at the end: 60000

Average Response Time at the end: 22593.8

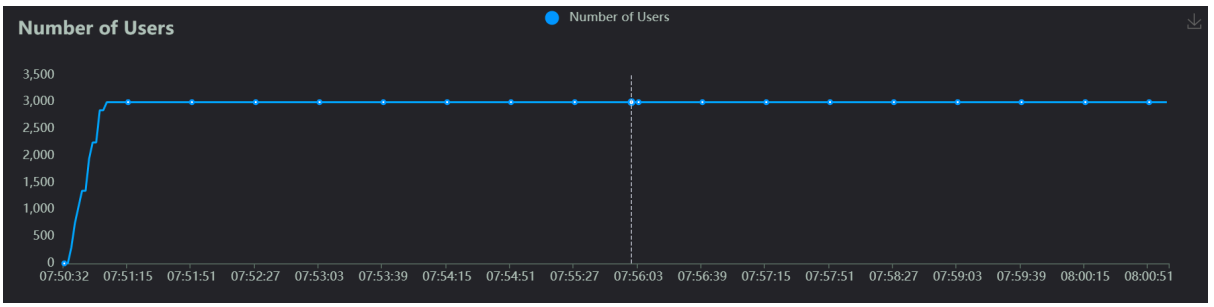

Figure 3.51 : Results With Load Balancing Policy As Maglev, number of users

While using the Maglev method we also got Http503 and Http504 errors. At the end 7% of the requests failed. On the other hand Least Request policy returned 0% error and Round Robin policy returned 6% error in total, with the same user number and RM. The 95th percentile at the end stayed exactly the same with Round Robin policy and also average response time stayed approximately similar.

- **Discussion Of The Results On Locust Parameters**

Locust facilitates load testing through two key parameters: number of concurrent users and hatch rate (users spawned per second). By strategically adjusting these values, we can control the volume of traffic directed towards our cloud system, enabling us to observe and analyze system behavior under varying load conditions.

❖ **Results With User Number As 5000 (Every other parameter than User Number are same with New Default State)**

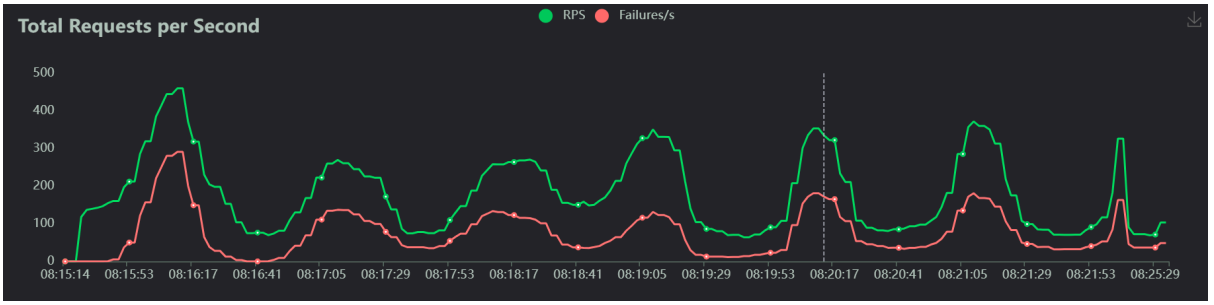

Figure 3.52 : Results With User Number As 5000 , total requests per second

RPS at the end: 89.6

Failures/s at the end: 42.8

Total Failures at the end: 45%

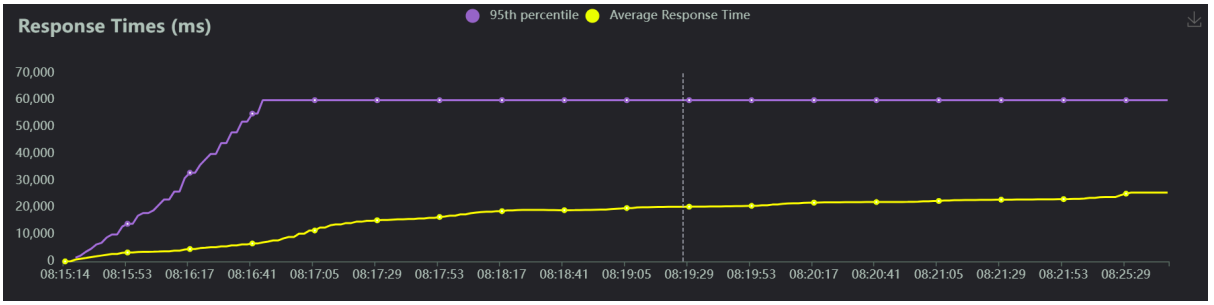

Figure 3.53 : Results With User Number As 5000 , response times

95th percentile at the end: 60000

Average Response Time at the end: 22593.8

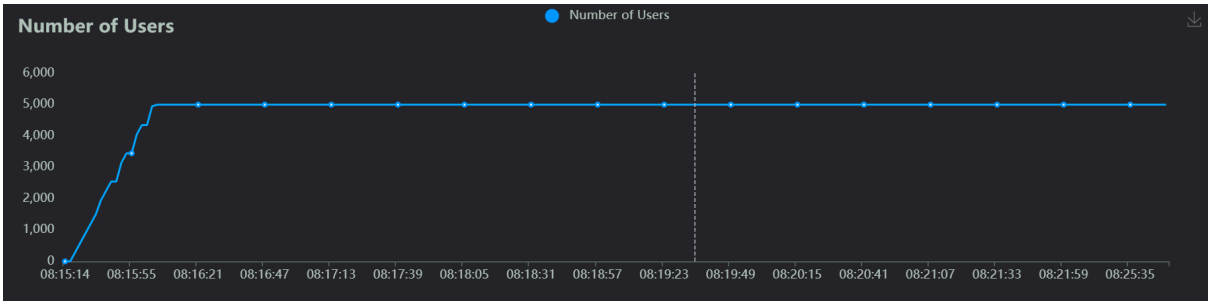

Figure 3.54 : Results With User Number As 5000, number of users

As we see, the number of users is a crucial parameter that changes the success of the system. On the very first experiments we were able to increase the number of users to handle without failure by changing parameters. Likewise in this scenario we got a 45% of failure from total requests. This indicates that the existing system can not handle such a large amount of

requests well and need adjustment. We can say from our tests that our systemS limit is between 3000 and 5000 users when we have RM as 150.

- **Results With RM As 500 (Every other parameter than RM are same with New Default State)**

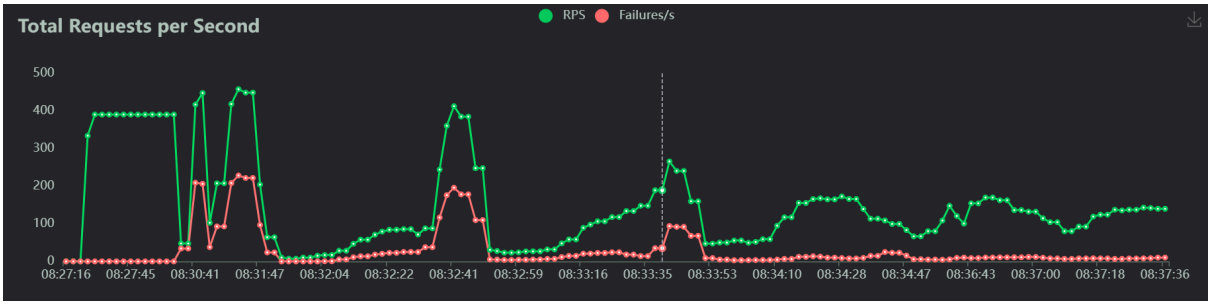

Figure 3.55 : Results With RM As 500 , total requests per second

RPS at the end: 139.9

Failures/s at the end: 10.2

Total Failures at the end: 25%

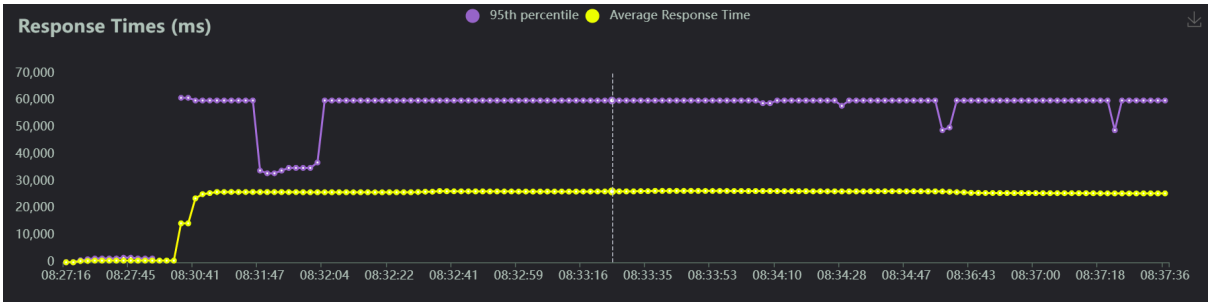

Figure 3.56 : Results With RM As 500 , response times

95th percentile at the end: 60000

Average Response Time at the end: 25534.2

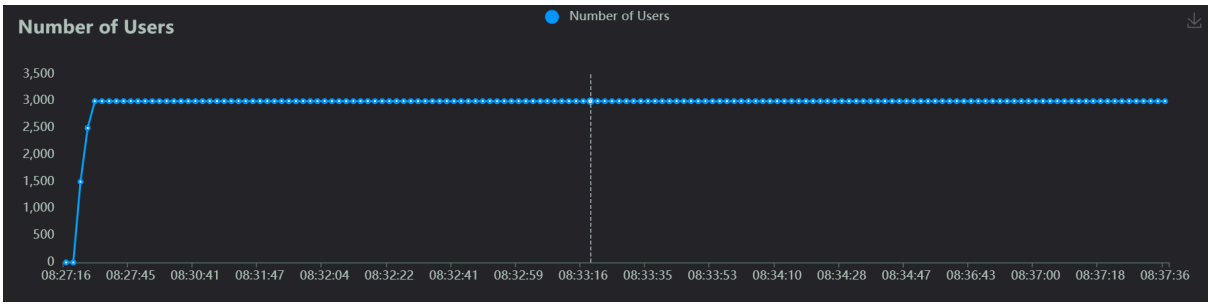

Figure 3.57 : Results With RM As 500 , number of users

Similarly, RM is a crucial parameter that changes the success of the system. Likewise in this scenario we got a 25% of failure from total requests. Which is a considerable amount of error that indicates that our system can handle such load with its current condition. We can say from our tests that our system's limit is between 150 and 500 for RM. Again, this limit can be changed by adjusting necessary parameters as we did on our first experiment by changing timeout parameters value.

- **Results Of Using VM's In Our Instance Group Directly Without Load Balancing (Every other parameter than number of users are same with New Default State. Number of users are 9000.)**

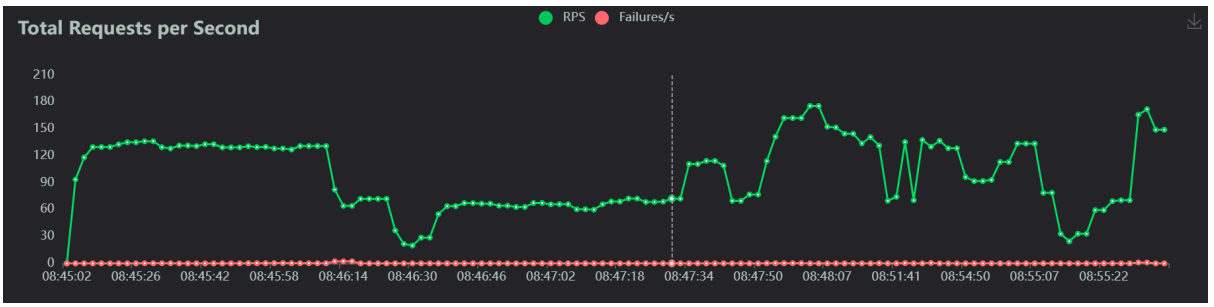

Figure 3.58 : Results Of Using VM's In Our Instance Group Directly Without Load Balancing , total requests per second

RPS at the end: 149

Failures/s at the end: 0

Total Failures at the end: 0%

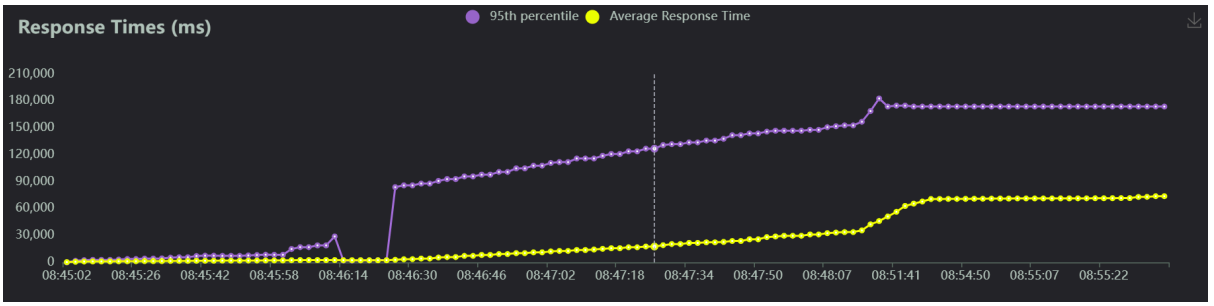

Figure 3.59 : Results Of Using VM's In Our Instance Group Directly Without Load Balancing , response times

95th percentile at the end: 174000

Average Response Time at the end: 73923.49

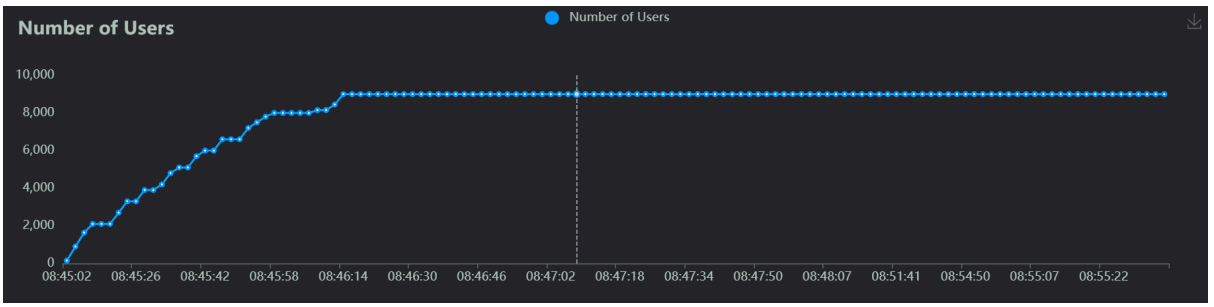

Figure 3.60 : Results Of Using VM's In Our Instance Group Directly Without Load Balancing , number of users

Our experiment compared the performance of our virtual machines in an instance group with and without load balancing. Interestingly, the system handled nearly three times more users without load balancing. This counterintuitive result suggests that the additional overhead introduced by load balancing, particularly the time spent on routing requests to the most appropriate instance, may be impacting performance under heavy traffic. While forgoing load balancing is not a recommended approach due to its critical role in ensuring high availability and scalability, this experiment highlights the potential performance implications of load balancing under specific high-traffic scenarios. Further investigation is needed to determine if optimization of the load balancing configuration can mitigate these performance bottlenecks.