Cmpe48A Final Report

Deniz Ünal, Ersel Çanakçılı

December 2024

Contents

Github Repo
Demo Video

1 Introduction

1.1 Project

Blogsite is a simple and easy-to-use platform where you can share your thoughts, ideas, and stories. You can write your own blog posts, read what others have shared, and leave comments to join the conversation. It also allows you to keep your posts up-to-date by adding new information or making changes whenever you want. Blogsite is the perfect place to express yourself and connect with others through blogging.

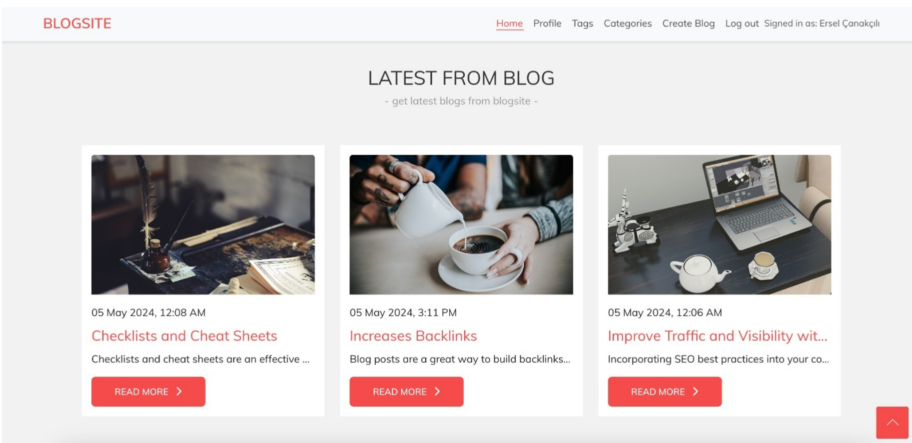


Figure 1: Screenshot from the project we have deployed.

1.2 Architecture

2 Cloud Components

We have used several products from Google Cloud Platform, these are:

- Google Cloud SQL
- Google Kubernetes Engine
- Google Cloud Functions
- Google Cloud Storage
- Google Compute Engine

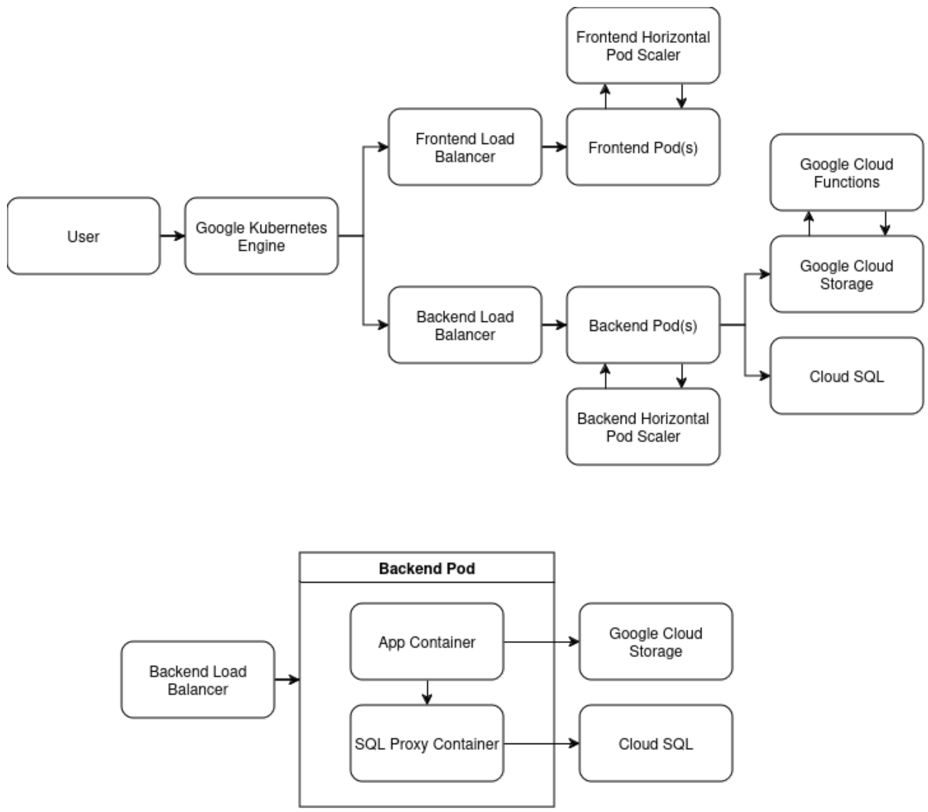


Figure 2: Our cloud architecture for the project.

2.0.1 CloudSQL

We used CloudSQL to store our database. PostgreSQL used as our SQL of choice, and we have tuned CloudSQL parameters to see find the optimal configuration.

2.0.2 Google Kubernetes Engine

We chose Google Kubernetes Engine (GKE) for our project because it offers many benefits, such as load balancing, auto-scaling, and more. In our Kubernetes nodes, we used two different types of pods, one for the back-end and one for the front-end, to efficiently scale based on the needs of each component.

2.0.3 Google Cloud Storage

We used Google Cloud Storage because it is easy to use, safe, and can store lots of data. It helps us manage files without worrying about losing them. It also offers different options to save money and keeps our data secure.

2.0.4 Cloud Functions

We used Cloud Functions to resize the images the users are uploading to their posts. We added a trigger that detects whenever an object (an image file in our case) is finalized (created) to our storage bucket, and calls our resizing function. Our resizing function takes the image in the bucket, resizes it to 256x256 image size, and replaces the original with the resized one. This use case is not realistic since we are warping the image if the original image ratio is not 1:1 but since our main subject is not the application itself but the our architecture in the cloud and the deployment itself we didn't see the necessity to create an overly complex function. Obviously we can increase the computation load with a more complex function but since we are in the free tier, it could cost too much and we find our function is in sufficient complexity.

3 System Parameters

We have several parameters in our architecture to adjust to optimize cost and performance. These parameters are:

- CloudSQL vCPU
- CloudSQL RAM size
- Kubernetes Engine node vCPU
- Kubernetes Engine node RAM
- Kubernetes Engine node count
- Kubernetes Engine pod count
- CPU limit for each Kubernetes pod
- Memory limit for each Kubernetes pod

We have done tests to tune these parameters, but since our budget and time are limited, we have restricted our test parameters. In our tests we have set node size equal to the pod size and let the pod acquire the total amount of resources available to a single pod (note that in our repository there is are resource limits set by our configurations but we have commented them out during testing). Also, we only used e2-medium node with 2 vCPU and 4 GB of memory for our nodes in our cluster. We used 4 different configurations for our CloudSQL machine, however two of them proved to be unnecessarily powerful for our use case. Those are 1 vCPU and 3.75 GB of RAM, 2 vCPU and 8 GB of RAM, 4 vCPU and 16 GB of RAM and lastly 8 vCPU and 32 GB of RAM.

4 Results

4.1 Network Traffic

We tried our tests with 3 different loads. They are 64 users, 256 users, and 1024 users. We used +1 user per second as our ramp up parameter for 64 and 256 users and +2 user per second for 1024 users. We let the test run for a while after the ramp up duration has ended to see the behavior in maximum traffic.

We are aware that 1024 user is a little on the lighter side in terms of network traffic. However, since the original implementation we have found in the github is not well implemented and the django framework itself is also on the slower side we couldn't increase it more without getting too much failure in the system. We believe that this is not because of our faulty cloud configurations, but rather from the project itself since the app runs slow even when we run it in our own workstations without any containerization.

Figure 3: 64 Users

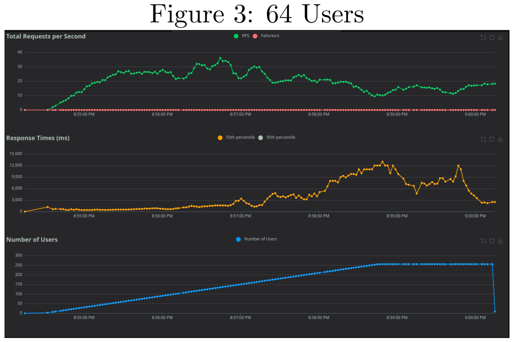


Figure 4: 256 Users

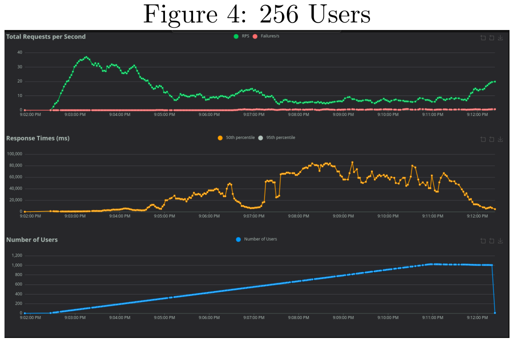


Figure 5: 1024 Users

Figure 6: Test Results With Different Loads.

4.2 SQL Bottleneck

We have performed our all tests with different CloudSQL machine configurations and found out that SQL can be the bottleneck with some loads. We have used our most powerful Kubernetes configuration to make sure the bottleneck is not on the Kuberenetes side. Here are our test results with 5 nodes with 2 vCPU and 4 GB of RAM with one pod in each with different SQL Configurations:

4.2.1 Low Load - 64 Users

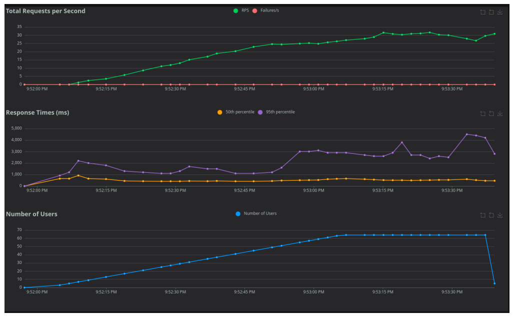


Figure 7: With 1 vCPU and 3.75 GB RAM

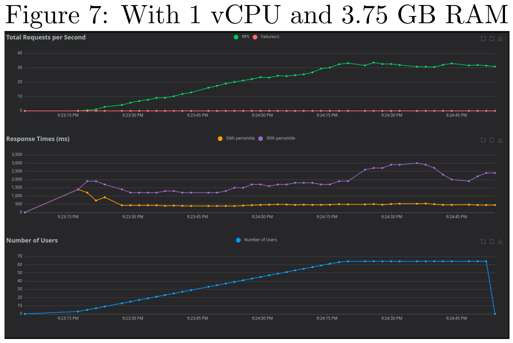


Figure 8: With 2 vCPU and 3.75 GB RAM

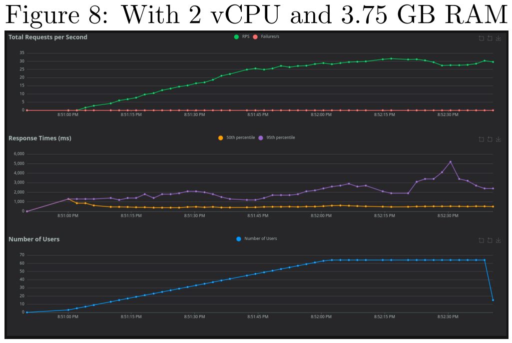


Figure 9: With 4 vCPU and 16 GB RAM

Figure 10: Test Results With Different SQL Configurations.

As it can be seen there are not much performance difference with each other. This is expected since 64 user is quite a low amount of traffic and even 1 vCPU with 3.75 GB RAM will easily suffice. But we will see the performance effect as we increase the load.

4.2.2 Medium Load - 256 Users

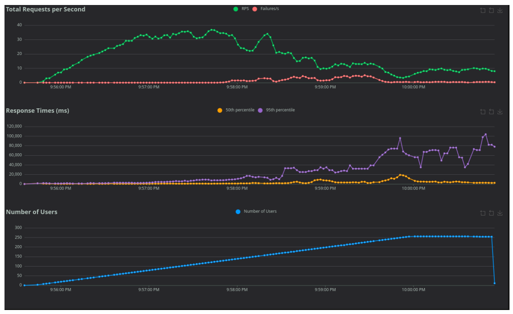


Figure 11: With 1 vCPU and 3.75 GB RAM

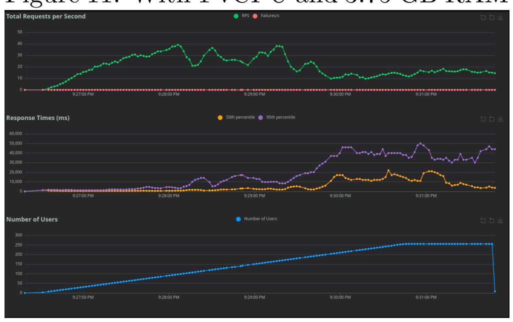


Figure 12: With 2 vCPU and 3.75 GB RAM

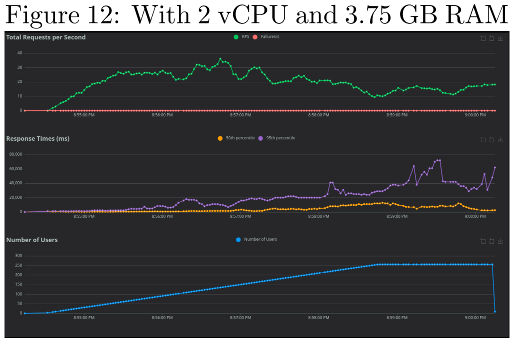


Figure 13: With 4 vCPU and 16 GB RAM

Figure 14: Test Results With Different SQL Configurations.

Now we can see that 1 vCPU and 3.75 GB RAM is starting to create a bottleneck as it has lower performance (higher latency) compared to two other options. The two other options still have similar performance meaning that 2 vCPU and 8 GB RAM is still enough for this amount of load.

4.2.3 High Load - 1024 Users

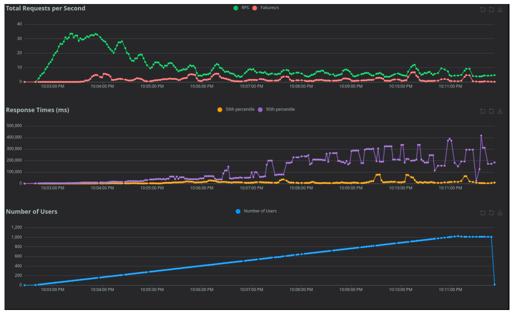


Figure 15: With 1 vCPU and 3.75 GB RAM

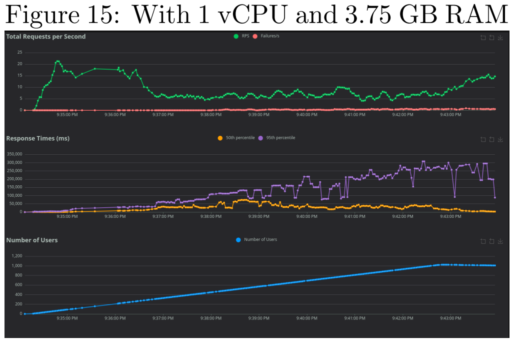


Figure 16: With 2 vCPU and 3.75 GB RAM

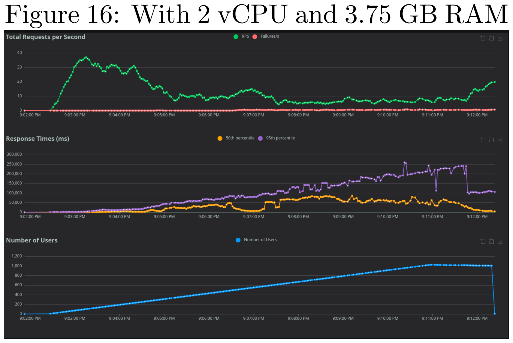


Figure 17: With 4 vCPU and 16 GB RAM

Figure 18: Test Results With Different SQL Configurations.

As we increase the load to 1024 user the difference gets bigger and easier to distinguish. And we can also see it in the failure counts. Since the SQL server couldn't handle the increasing load we got a lot of 500 (internal server errors). We can see the failure count in the tables 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
GET	/api/blog	735	162	13000	189000	262000	42967.11	406	413853	32964.45	0.2	0
GET	/api/blogs/author/[id]/	1219	155	1000	65000	237000	10699.88	373	358587	215.05	0.9	0
POST	/api/create	487	60	2600	43000	175000	11205.51	531	371837	903.41	0.8	0
DELETE	/api/delete/	367	40	1500	40000	287000	11134.78	346	346089	15.8	0.3	0
POST	/api/register	853	116	7400	73000	177000	18082.54	679	440489	134.63	0.1	0.1
POST	/api/token/	598	59	6300	66000	97000	14824.91	613	403091	454.52	0.4	0
GET	/api/user	1649	300	1000	75000	246000	13449.09	332	451738	186.22	2	0
	Aggregated	5908	892	2700	79000	237000	17033.65	332	451738	4338.27	4.7	0.1

Figure 19: With 1 vCPU and 3.75 GB RAM

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
GET	/api/blog	418	12	48000	220000	356000	81031.34	1500	431848	55728.45	0.5	0.2
GET	/api/blogs/author/[id]/	1189	18	8600	76000	241000	22398.7	372	386732	101.54	4.5	0
POST	/api/create	536	4	11000	85000	110000	24202.78	620	367766	1005.58	2.3	0
DELETE	/api/delete/	461	12	3000	69000	233000	12999.35	344	299679	3.77	2.4	0
POST	/api/register	835	47	10000	77000	295000	23001.44	744	346999	133.68	0.2	0.2
POST	/api/token/	679	17	28000	89000	294000	41676.49	604	382231	480.6	0.3	0.1
GET	/api/user	1340	22	8700	79000	252000	23941.68	337	389498	195.21	4.5	0
	Aggregated	5458	132	11000	90000	277000	29141.61	337	431848	4517.31	14.7	0.5

Figure 20: With 2 vCPU and 3.75 GB RAM

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
GET	/api/blog	753	18	15000	173000	212000	44857.95	442	293911	29178.45	0.3	0
GET	/api/blogs/author/[id]/	1783	25	4000	94000	165000	20962.8	368	391514	139.05	6.7	0
POST	/api/create	811	9	4500	98000	129000	21978.96	633	332012	1000.65	3.3	0
DELETE	/api/delete/	741	22	2300	63000	175000	12230.81	343	306744	4.11	3.2	0.1
POST	/api/register	886	49	11000	47000	259000	20068.6	739	356894	133.17	0.3	0.3
POST	/api/token/	744	14	47000	111000	251000	48388.7	611	359763	482.09	0	0
GET	/api/user	1954	35	3900	98000	178000	23259.3	332	400000	194.55	6.1	0.3
	Aggregated	7672	172	6400	104000	199000	25713.41	332	400000	3114.01	19.9	0.7

Figure 21: With 4 vCPU and 16 GB RAM

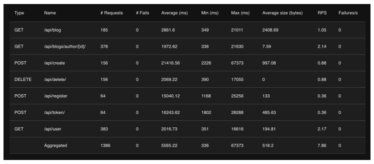
Figure 22: Test Results With Different SQL Configurations.

As we can see we have much more failures in 1 vCPU case compared to two other and we can see the other two performed well meaning the SQL is not the bottleneck here. We could increase the load even more to see where 2 vCPU starts to become unsufficient but the Kubernetes become the bottleneck after this much load and we can not increase the computing power that much in Kubernetes with free tier limits.

4.3 Pod Bottleneck

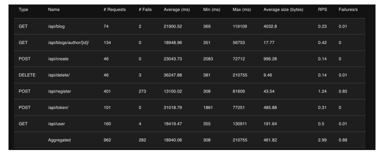
4.3.1 1 Node - 1 Pod

In a system with 64 users, the response time is poor, but no failures are observed. However, when the number of users increases to 256, both the failure rate and response time significantly worsen.



Type	Name	# Requests	# Fails	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	RPS	Failures/s
GET	/api/blog	185	0	2861.6	349	21011	2408.69	1.05	0
GET	/api/blogs/author/[id]/	378	0	1972.62	336	21630	7.59	2.14	0
POST	/api/create	156	0	21416.56	2226	67373	997.08	0.88	0
DELETE	/api/delete/	156	0	2069.22	390	17055	0	0.88	0
POST	/api/register	64	0	15040.12	1168	25256	133	0.36	0
POST	/api/token/	64	0	16243.62	1802	28288	485.63	0.36	0
GET	/api/user	383	0	2016.73	351	16616	194.81	2.17	0
	Aggregated	1386	0	5565.22	336	67373	518.2	7.86	0

Figure 23: 64 Users (Low Traffic)



Type	Name	# Requests	# Fails	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	RPS	Failures/s
GET	/api/blog	74	2	21900.52	369	119109	4032.8	0.23	0.01
GET	/api/blogs/author/[id]/	134	0	18948.96	351	56753	17.77	0.42	0
POST	/api/create	46	0	23043.73	2083	72712	996.28	0.14	0
DELETE	/api/delete/	46	3	36247.88	381	210755	9.46	0.14	0.01
POST	/api/register	401	273	13100.02	308	81609	43.54	1.24	0.85
POST	/api/token/	101	0	31018.79	1861	77251	485.88	0.31	0
GET	/api/user	160	4	18419.47	355	130911	191.64	0.5	0.01
	Aggregated	962	282	18940.06	308	210755	461.82	2.99	0.88

Figure 24: 256 Users (Medium Traffic)

Figure 25: Statistics

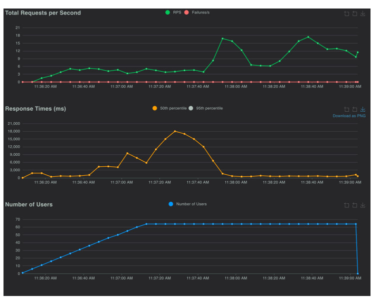


Figure 26: 64 Users (Low Traffic)

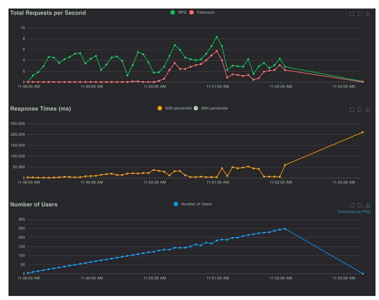


Figure 27: 256 Users (Medium Traffic)

Figure 28: Graphs

4.3.2 3 Nodes - 3 Pods

When using 3 nodes and 3 pods, the failure rate decreases significantly at 256 users. However, at 1024 users, which represents high traffic for our system, both the response time and the failure rate increase substantially. This demonstrates that by increasing the number of pods, we can reduce the response time and failure rate. This is because more pods allow us to utilize additional resources, such as CPU and memory, which help the system handle higher traffic more efficiently.

Type	Name	# Requests	# Fails	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	RPS	Failures/s
GET	/api/blog	586	21	11319.99	352	226533	20879.06	1.41	0.05
GET	/api/blogs/author/[id]/	1158	38	4190.46	330	195837	141.34	2.79	0.09
POST	/api/create	528	16	39582.62	1014	186911	988.75	1.27	0.04
DELETE	/api/delete/	473	13	4951.76	346	229592	3.99	1.14	0.03
POST	/api/register	260	10	7700.98	801	203890	133.46	0.63	0.02
POST	/api/token/	246	8	4585.33	581	179081	476.5	0.59	0.02
GET	/api/user	1253	40	3723.35	328	223329	193.63	3.02	0.1
	Aggregated	4504	146	9441.27	328	229592	2956.77	10.87	0.35

Figure 29: 256 Users (Medium Traffic)

Type	Name	# Requests	# Fails	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	RPS	Failures/s
GET	/api/blog	479	144	28158.28	347	305261	23408.3	0.85	0.25
GET	/api/blogs/author/[id]/	739	124	13637.15	329	354945	118.9	1.31	0.22
POST	/api/create	276	44	28454.1	1557	420899	876.19	0.49	0.08
DELETE	/api/delete/	199	36	17221.52	342	301552	26.23	0.35	0.06
POST	/api/register	1120	458	29387.14	328	313793	97.13	1.98	0.81
POST	/api/token/	522	56	23007.72	562	347948	451.5	0.92	0.1
GET	/api/user	1157	336	20199.64	331	241726	180.45	2.05	0.59
	Aggregated	4492	1198	22960.97	328	420899	2693.84	7.95	2.12

Figure 30: 1024 Users (High Traffic)

Figure 31: Statistics

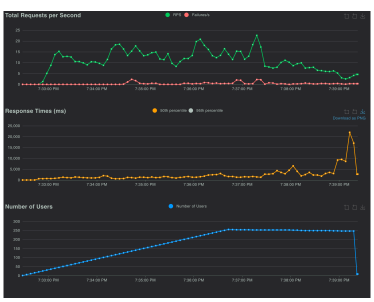


Figure 32: 256 Users (Medium Traffic)

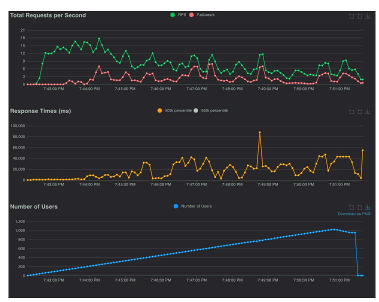


Figure 33: 1024 Users (High Traffic)

Figure 34: Graphs