**Cs 436 Term Project Report**

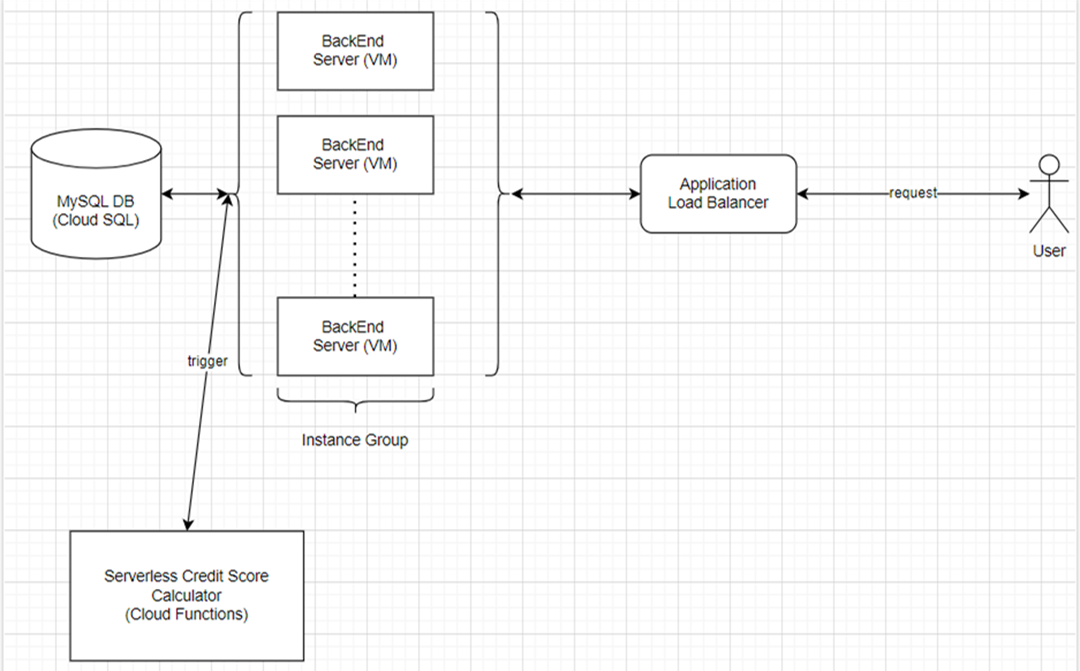
Kerim Demir

Batu Onart

Altuğ Yalım

Egemen Kılıç

**Architecture:**

****

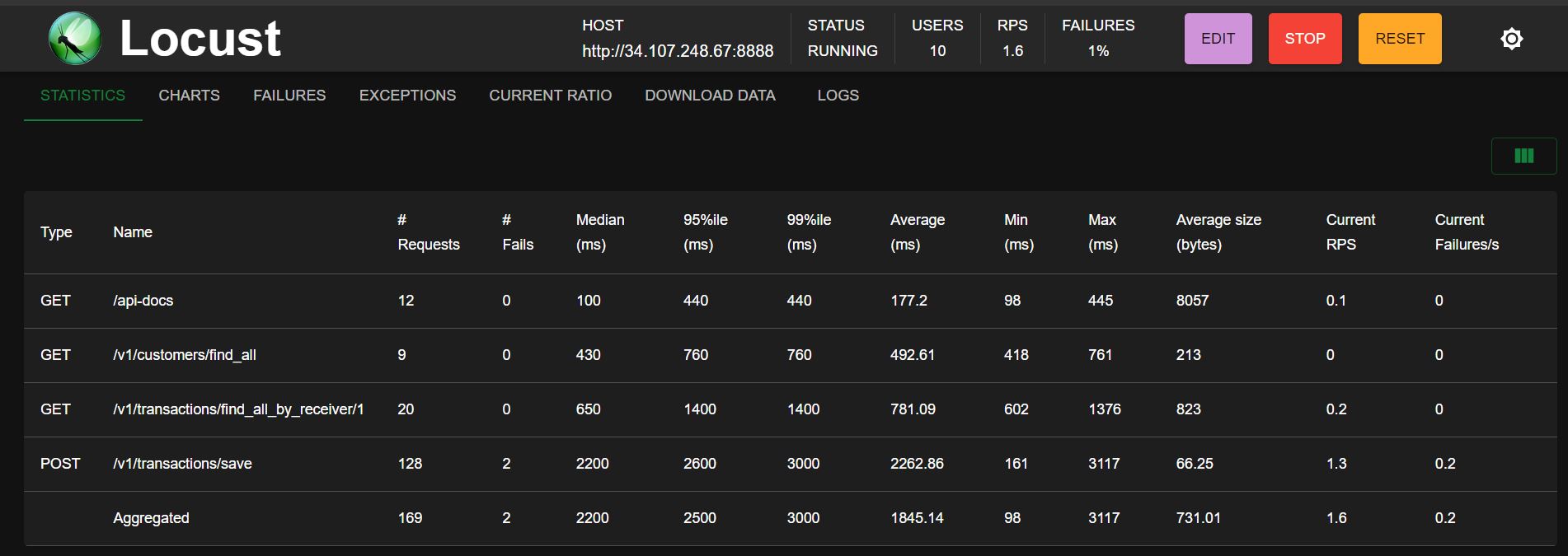
There is a single entry-point to the application which is through the application load-balancer. There is a single managed instance group that is configured to have minimum 1 and maximum 5 VMs running and the max configuration under the desired budget limit is 4 vCPU and 6GB memory for each VM. There is a single MySQL database with 2 vCPU and 8GB memory. There is also a serverless cloud function that handles a specific request to the system that has 256 MB memory.

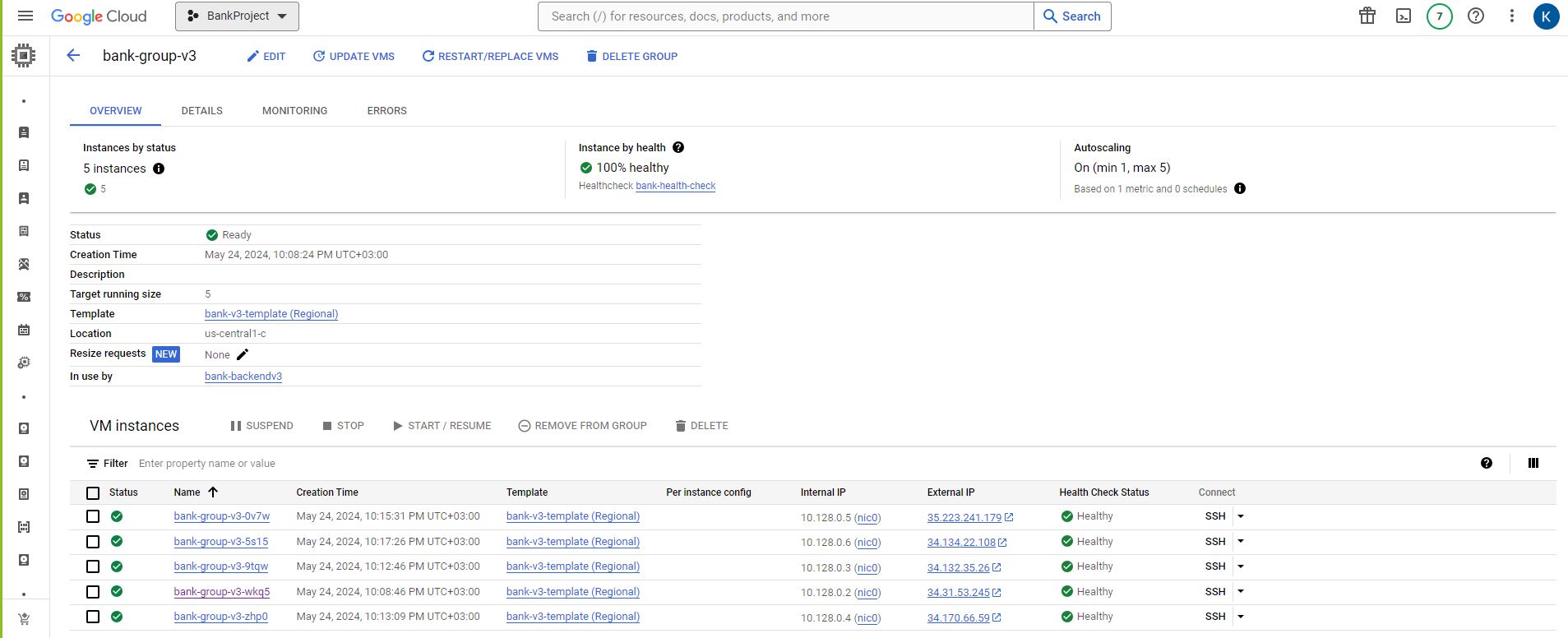
**Testing Report:**

* **Evaluated parameters:** vCPU and Memory allocation in VMs
  + With the minimum configuration that is possible for managed instance Vms that is e2-micro, when we stress test the system it is seen that the number of instances increase from 1 to 5 after a couple of seconds. However, if they are configured to have 4 vCPU and 6GB RAM, we see that only 1 instance is enough to handle the load. Results will be shown below.
* Locust was used to evaluate the system performance. (Code is in the github repo)
* **Environment setup:**
  + Create a MySQL database using Cloud SQL service. (Database schema script is available in the ddl.sql file of the backend code)
  + Create a single VM with the desired configuration that is wanted in the managed instance group VMs, while creating the initial VM choose the option to keep the disk after deletion. After configuring the VM, delete it.
    - Change the Database connection ip to the newly created ones ip in the Backend code.
    - Upload the Backend code into the VM.
    - Create a service file so that our Backend code will start running at the startup of the VM.
  + Create an image from the disk of the deleted VM.
  + Create an instance template from the image.
  + Create a managed instance group from the instance template. (min 1, max 5 instance)
  + Create a load balancer.
    - No need for a specific front-end, create a default one.
    - Configure the Backend of the load balancer to our instance group.
  + Testing environment is ready. IP of the frontend can be used to access the application.

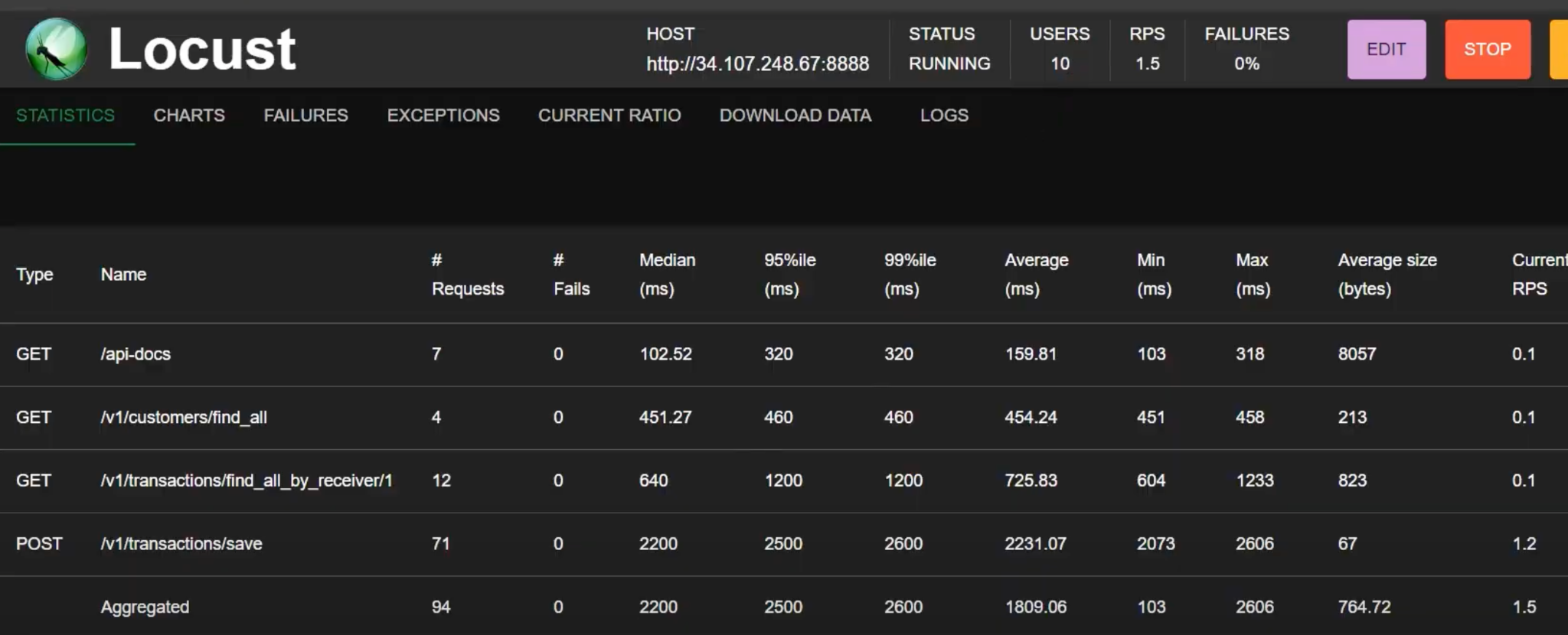
**Test Results:**

* **Minimum configuration:**
  + With minimum configuration of the VMs (e2-micro), testing the system with 10 users, we have received an average of 2231ms in our POST save function. When tested with a single Postman request we received a similar result of 2450ms. From the test results it is visible that the “save” method makes the most load on the server. Also, the second chart below shows that with minimum configuration, the VMs scale up to 5 instances when the overall load increases on the server, which we can achieve with even 10 users.

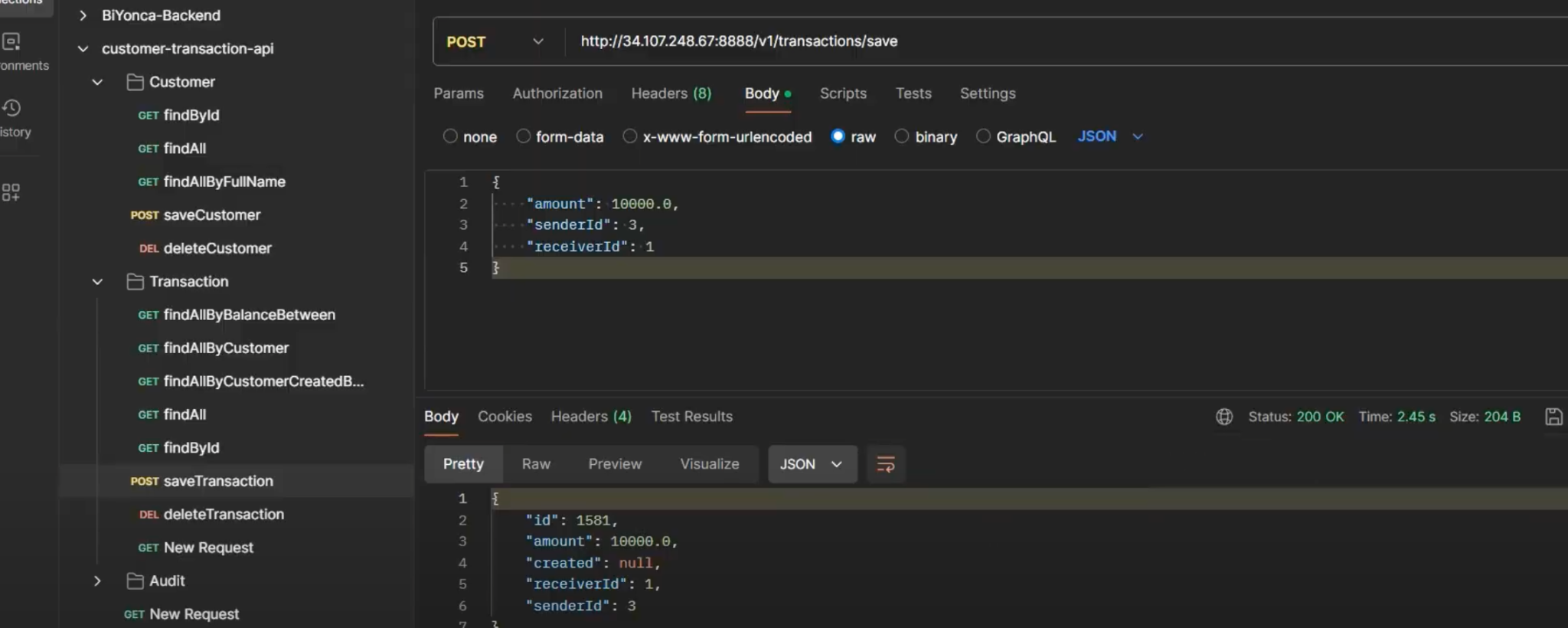
*Locust test result with 10 users (ec-2 micro)*

*The scaling of the VMs*

* **Maximum configuration:**
  + With maximum configuration possible below the budget limit (4 vCPU and 6GB RAM for VMs), testing the system with 10 users, we have received an average of 2231ms in our POST save function. When tested with a single Postman request we received a similar result of 2450ms. From the test results it is visible that the “save” method makes the most load on the server.

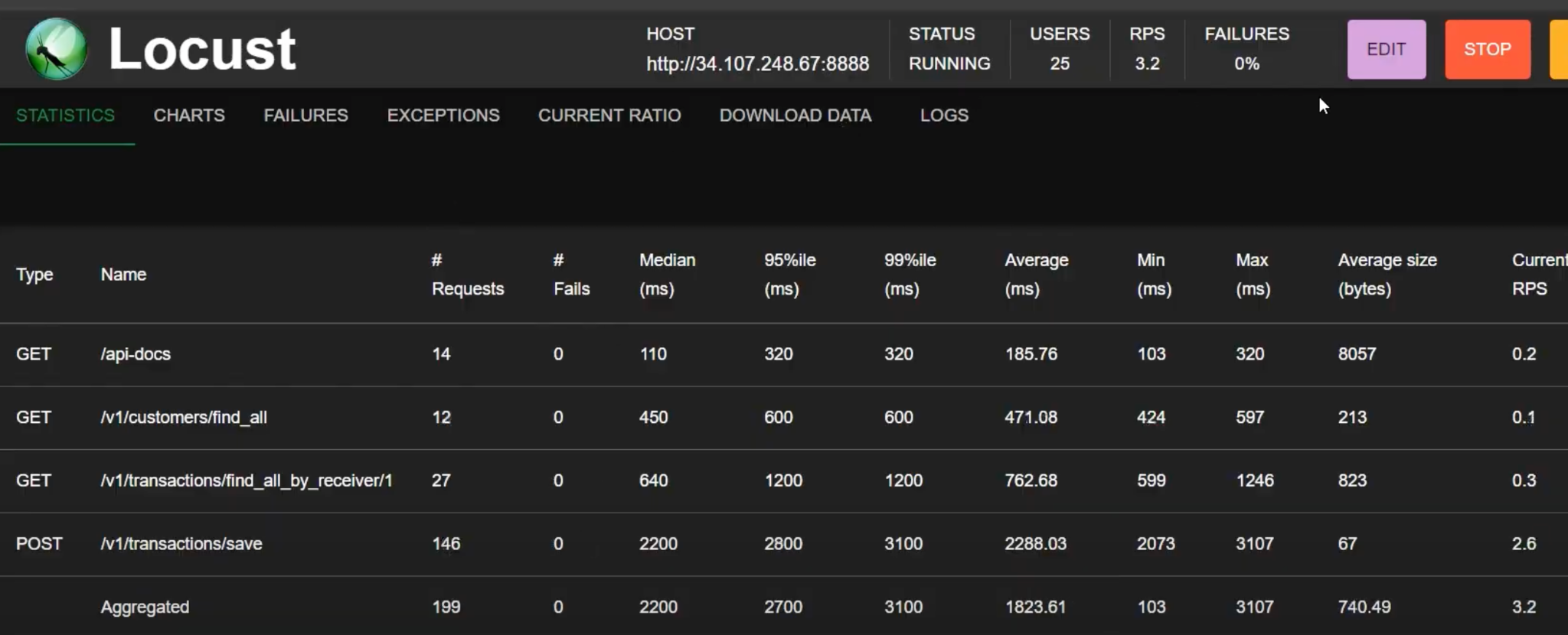


*Locust results of the test with 25 users.*



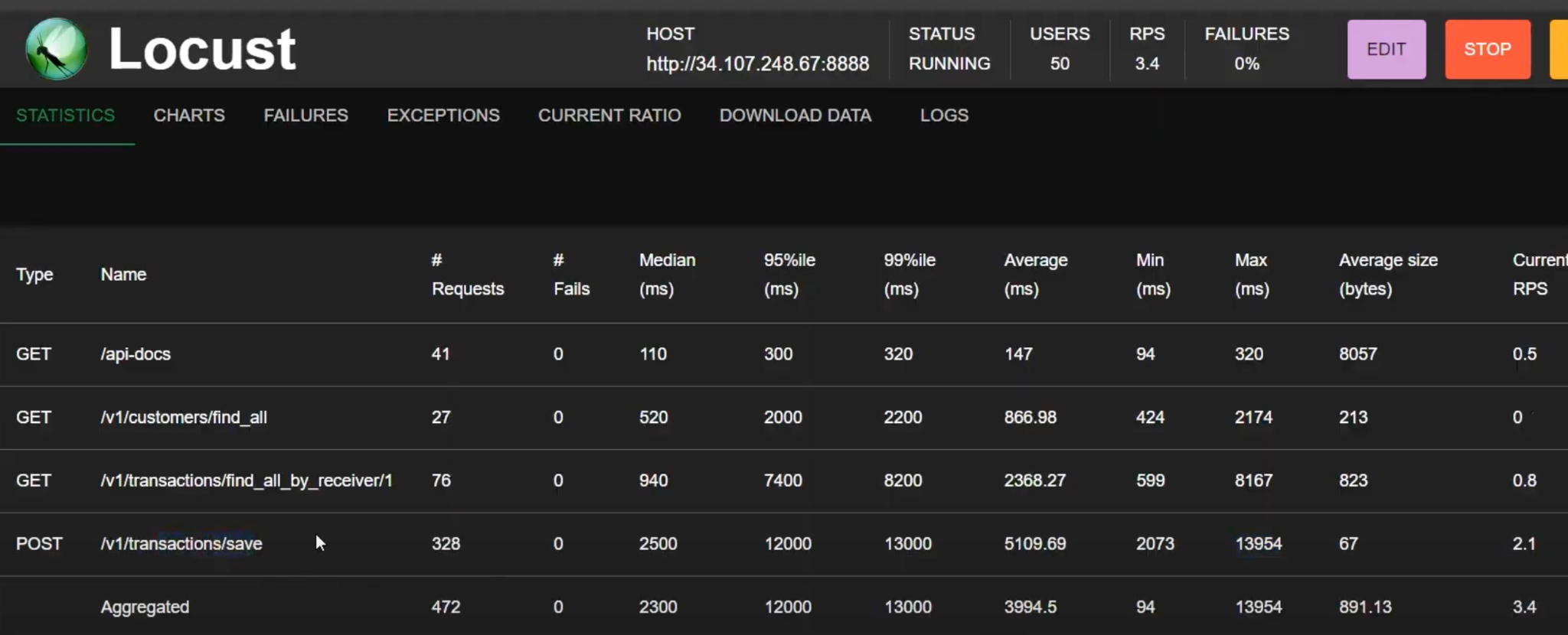
*A single test result with 25 users from the POST function.*

* With 25 users, we have seen a slight increase in the overall average of the API’s. Especially in GET find\_all functions and GET find\_all\_by\_reciever/1 functions had a slight increase in their average response time. Since we only increased the user amount by 15, this small change was expected.

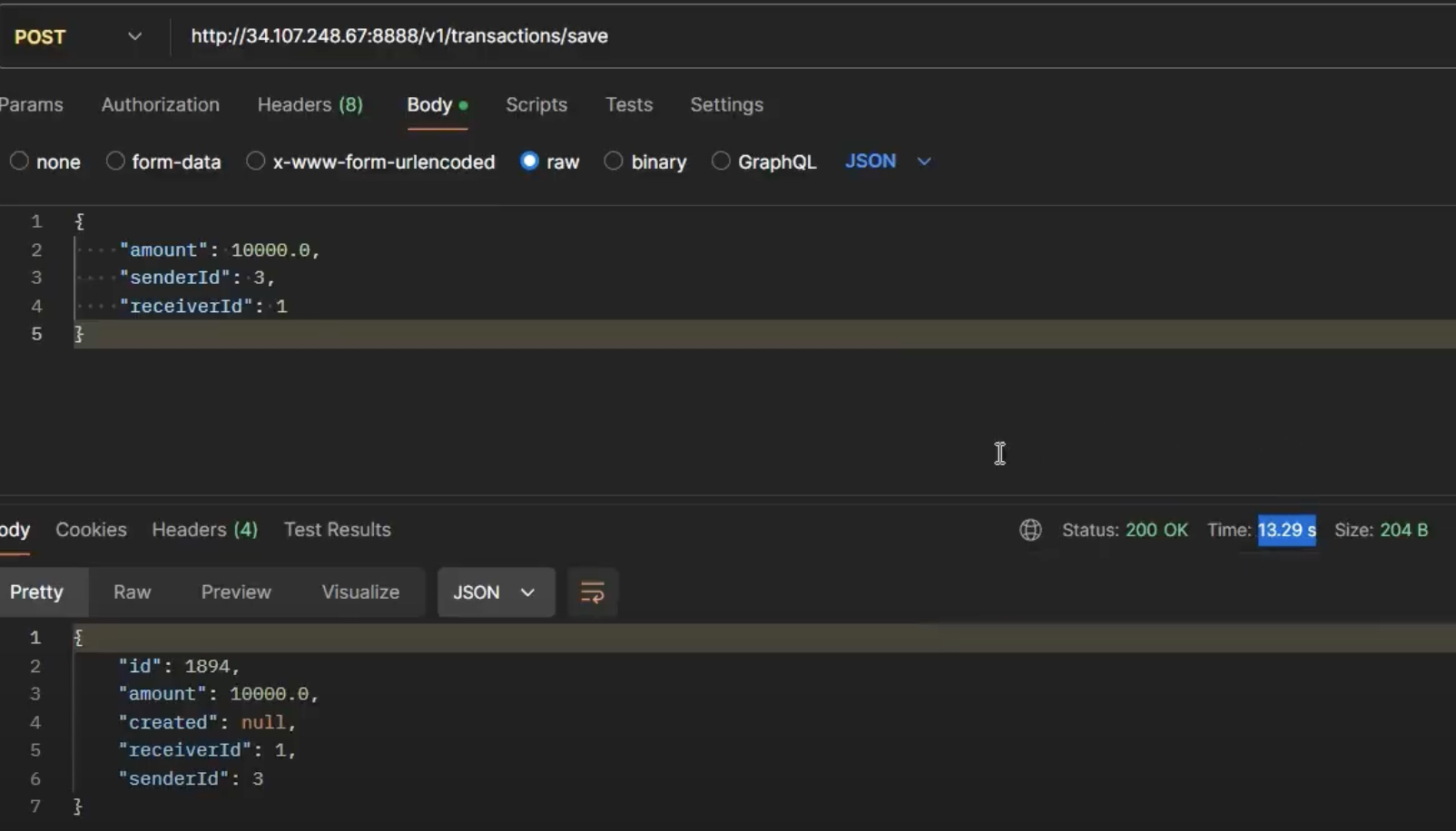


*Locust results of the test with 25 users.*

* After setting it to 50 users, we have noticed great changes in average response times. To be more specific, our POST save function almost doubled in its average response time, rising to 5109ms from the previously 2288ms. Apart from GET /api-docs, all our other API functions also doubled as well. This was again expected since we doubled the amount of users compared to the previous test.

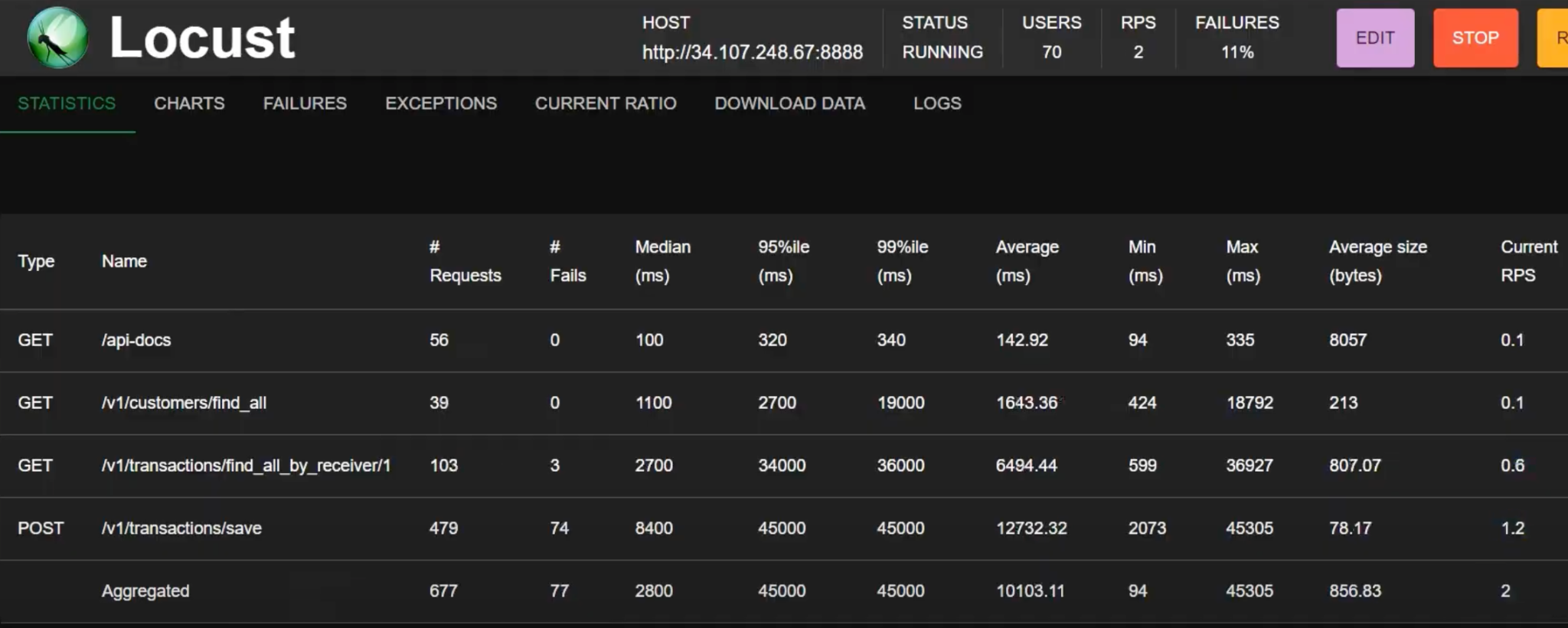


*Locust results of the test with 50 users.*

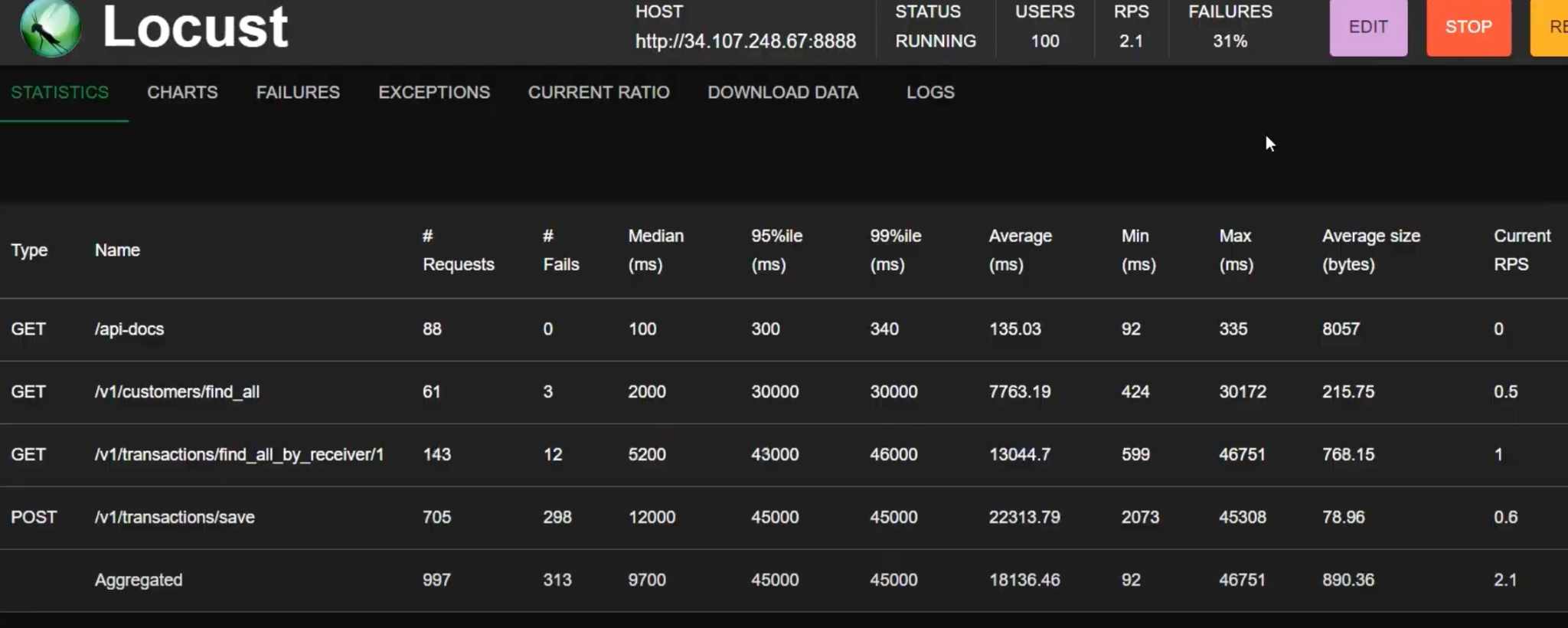


*A single test result with 50 users from the POST function.*

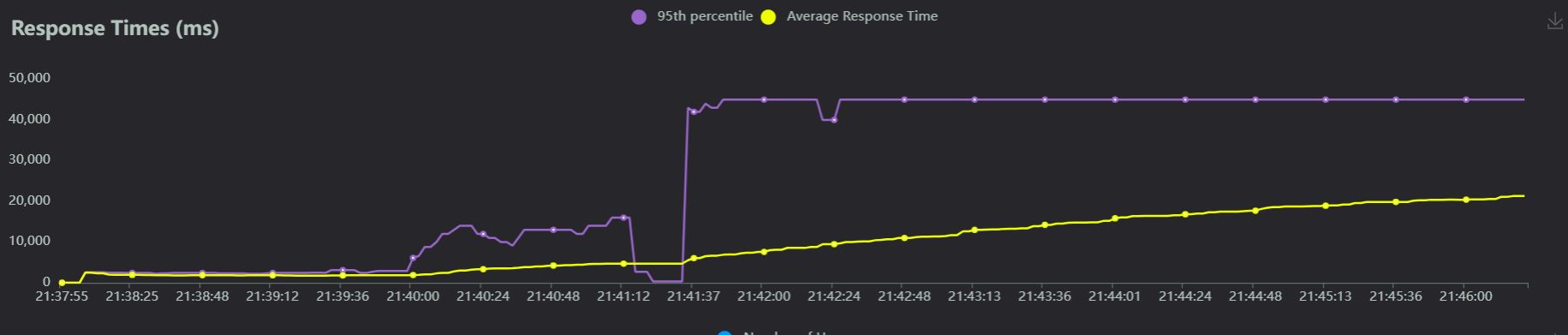
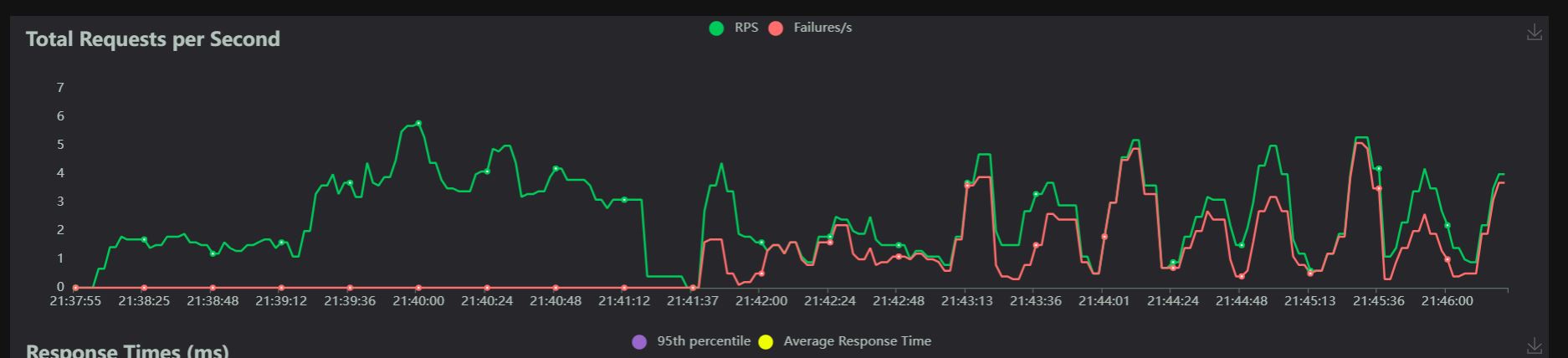
* By setting the users to 70, we have seen an even more drastic change in average response times for our API’s. Especially for the POST function, we saw an increase from 5109ms to 12732ms. Although the average is around 13 seconds, it is seen that 99% of the requests are around 45 seconds which results in timeout errors, because the load balancer is configured to maximum of 30 seconds per request which causes requests over 30 seconds to give a return results in failures. Specifically, Http error code 504.

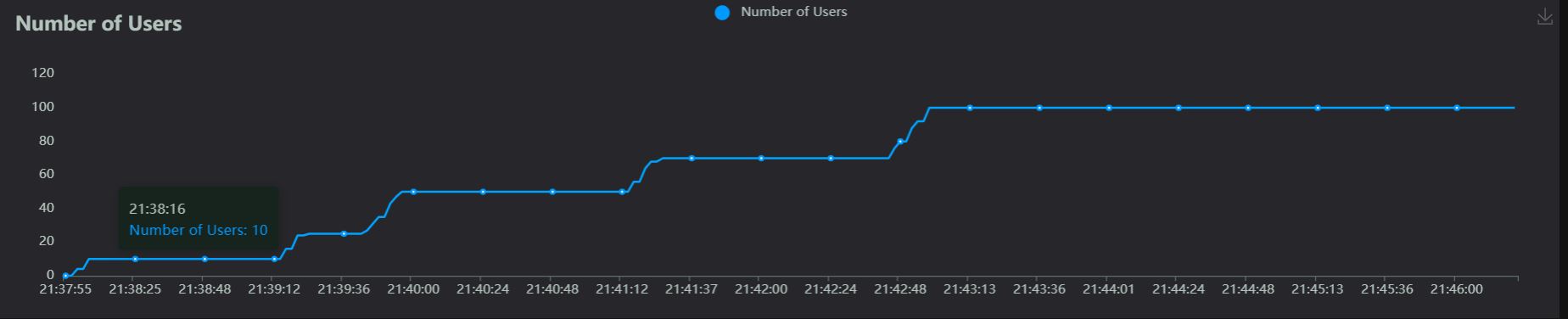
*Locust results of the test with 70 users.*

* Finally, with the 100 user test we again saw a total increase from 12732ms to 22313ms in response times for the POST function. We also noticed a great increase in failed requests. It can be seen that increasing the amount of users increases our average response times almost exponentially.

****

*Locust results of the test with 100 users.*

* From the charts below we can see the whole process of the testing as the user numbers increase.



Moreover, while doing the stress test over the maximum possible configuration of the VMs, it was observed that the number of instances managed by the instance group did not increase as the number of users increased.

To sum up, the bottleneck of the application stems from the backend code that runs in each VM. This event is also shown in the demo video as well where we saw that the CPU utilization of the single instance doesn’t even pass 30% which means the problem is in the application (backend) itself. Therefore, a viable solution could be optimizing the save method in the code or removing it from the backend and moving it to another service provided by the GCP.