A18: Load Testing the Web Server

First, I ran a series of pre-trial tests to get a sense of how Locust works. Then I ran 6 trials with the following user numbers and spawn rates:

Trial Number	Number of Users	Spawn Rate
Trial 1	100 users	5 users/ second
Trial 2	200 users	20 users/ second
Trial 3	400 users	20 users/ second
Trial 4	800 users	50 users/ second
Trial 5	2500 users	80 users/ second
Trial 6	350 users	100 users/ second

The resulting Locust and AWS CloudWatch charts are in the Appendix.

For each case, I kept track of the timings from when I started the test to when the Scale-in and Scale-out alarms were triggered, the auto-scaling group launching and terminating instances, and how long it took for these instances to run in the EC2 dashboard.

Observations

In general terms, I observed the following patterns:

- In most cases, the Scale-out alarm was triggered first. This usually happened at the beginning of the test.
- Immediately after the Scale-out alarm was triggered, the web auto-scaling group would launch a new instance, growing the pool from 2 to 3.
- However, the process from the instance being launched by the auto-scaling group to reaching a "Running" status in the EC2 console was not immediate. For example, in Trial #4, the Scale-out alarm and new instance were launched at 3:56 am, but the instance changed status from "Initializing" to "Running" at 3:58 am, ie. 2 minutes later.
- In two tests that were left running for longer, the Scale-in alarm was triggered during the test, such that the auto-scaling group reduced the pool from 3 to instances. Later, the Scale-out alarm was triggered again, prompting the auto-scaler to once again increase the pool to 3 instances.
- During my testing, at most 3 instances were created.
- Once I hit stop, it would take a while for the instance selected by the auto-scaler to fully terminate.
- The Scale-in alarm would go back to "OK" sometime after the instance was terminated.

Analysis

There are several factors that affect cloud elasticity in this assignment.

First, the CloudWatch alarms play a key role in determining what specific event will trigger an action in the auto-scaling group, whose role is to manage scaling (both in and out) by increasing or decreasing our pool of EC2 web server instances. The ability to increase and decrease capacity as a response to changes in application demands is known as elasticity, and it is one of the main benefits of working on the Cloud. Off-the cloud this would be costly and more complicated as one would have to provision the maximum amount of resources required for peak times. The problem with that is that resources are being wasted during off-peak times, incurring additional costs.

In this case, I set up my Scale-out alarm when the sum of requests counts exceeds 200 in one minute. I used the RequestCount [1] parameter at the ELB level instead of by availability zone or target group because I wanted the total sum of requests received by the ELB. Also, there could be any number of instances running on different availability zones, so a metric by AZ would not be very informative for my purposes.

For my Scale-In alarm, I used the TargetResponseTime[2] metric, which measures the time elapsed in seconds from which the request leaves the ELB until a response from the target is received. I measured this and graphed as an average over one minute. Since the metric is in seconds, I set it to 0.01 seconds.

The second factor is the scale policies within the auto-scaling group, which determine what action to take if one alarm is triggered. The scale-out policy is activated by the scale-out alarm and responds by adding one EC2 instance. This explains why, in my set-up, my web server farm was never larger than 3 instances - once the alarm was triggered, the auto-scaler simply added 1 instance. Throughout each test, the Scale-out alarm remained triggered instead of turning off and on again each time there were more than 200 requests for one minute, which would have resulted in more instances being created.

Based on the number of users and spawn rate, Locust simulates user behavior. From my Locust charts, the *average* number of requests per second ranged from 700-850 for most of my testing. Adding these numbers over a minute results in a *very large* number of requests. Looking at my RequestCount charts, the statistics reach up to 50k. I found it puzzling that no additional instances were created.

It is worth noting that my application was able to handle a large number of requests with 2 to 3 instances without failures in all 6 trial runs except for Trial # 5, where I tried an extreme case of 2500 users with a spawn rate of 80 users/second. The Trial 5 – Locust charts show that it's the only case where failures spiked and remained at about 20% throughout the test. My expectation with this test was to challenge the application load and see if the number of requests per second would spike even higher. I wanted to see if that would result in more instances being launched in my web server pool. This didn't happen for the reason mentioned before.

Instead, the number of requests per second was lower than the other tests that had far fewer users. The reason is that by having so many users, this effectively caused the app to fail with 500 error codes. These failures prevented more requests from being accepted by the application, akin to a *denial of service* hacking. This suggests that the current GAS design is only functional for a certain number of users at a time, beyond which other scaling policies and alarms might be needed. For example, if the Scale-out could be triggered more than once or the auto-scaling can add instances until the alarm is turned-off, this could help in a production environment where there is a large number of users.

[1] https://docs.aws.amazon.com/elasticloadbalancing/latest/classic/elb-cloudwatch-metrics.html

[2] https://docs.aws.amazon.com/elasticloadbalancing/latest/application/load-balancer-cloudwatch-metrics.html

A third factor that affects elasticity behavior is the target group health check. From my observations, takes 2-3 minutes from an instance being launched to fully running because it must first be checked as healthy. The goal of the health check is to inform the balancer which EC2 instances are available to accept requests. From the user perspective, there is transparency because it looks like the application is one single component that is continuously running while behind the scenes, there are new instances being added and removed to match the demand.

Something else I noticed was that the Scale-out alarm was triggered at the beginning of each test. Looking at the charts, this seems to be because the swarming Locust process happens very quickly at the beginning (at least with the spawn rates I tested). As a result, the maximum number of requests per second happens at the very beginning, and it remains fairly constant over the remainder of the test. This could be because of how Locust simulates user behavior. I only ran each test for a couple of minutes at a time, but none of the simulations showed significant arbitrary drops or increases in requests. In general, the number of requests was very stable, which might not necessarily be representative of reality. It could be that the GAS user demand has more drastic spikes followed by periods of inactivity, which is very different from having a constant number of requests.

The charts also show there is an *apparent* inverse correlation between requests per second and response time. For example, Trial 1 – Locust charts show that a *drop* in total requests per second is accompanied by an *increase* in response time. This is counterintuitive. I would have expected that if fewer requests are coming in, then the application would respond quickly, not slower.

I think one possible reason for this is that the scaling process happens so quickly that it can run a full cycle in a matter of milliseconds. That is, the drop in demand (measured as the number of requests per second) would *immediately* result in lower response times, which would then trigger the Scale-in alarm (if target response time falls under 10 ms for 1 minute). Then, the Scale-in alarm results in an instance being terminated from the pool, which would once again *increase* response times (given that user demand is back up).

If this process happens quickly enough, it might explain why the charts show what *appears* to be a contradictory inverse correlation between the number of requests per second and response time. Looking at the **Error! Reference source not found.** and **Error! Reference source not found.**, both seem to support this theory as both RequestCount and TargetResponseTime are moving in the same direction – downwards. In other words, there is no inverse correlation between the two, but the quick acting of the auto-scaling group behind the scenes gives it that false appearance in the Locust charts.

Trial #6 was one example in which the scale-in alarm was triggered before the test ended. I recorded the following play-by-play, which illustrates the elastic behavior I explained above:

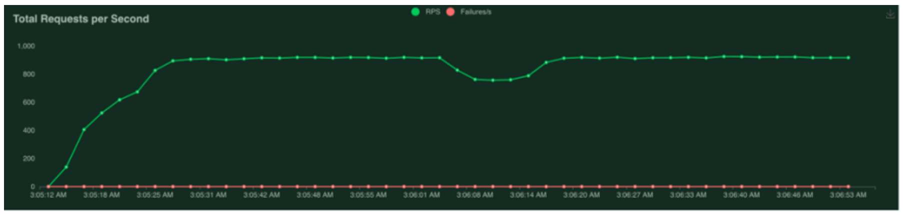
- The previous test (from Trial #5) had increased the web server pool to **3** instances
- Trial #6 started at **4:50am**
- The ***Scale-in*** alarm was triggered at **4:51 am**
    - This matches the Trial 6 – AWS CloudWatch Target Response Time chart that, at the same time, is below the threshold of 10 ms for one minute
    - As a result, at **4:53:11 am** the auto-scaling group terminated an instance → web server pool back to **2** instances
- The ***Scale-out*** alarm was triggered at **4:52 am**

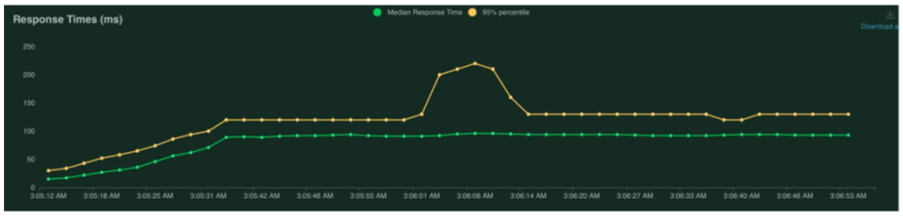
    - The auto-scaling group launched a new instance at **4:53:32 am**, increasing the pool to **3** instances
- At **4:57 am** there is a drop in Target Response time, visible in Trial 6 – AWS CloudWatch Target Response Time chart which triggered the ***Scale-in*** alarm (note that eventhough the increase in response time over the 10 ms threshold lasted only a few milliseconds, the immediate drop was enough to target the alarm again)
    - At **4:58:51 am**, the auto-scaler terminates an EC2 instance, back to **2** instances
- At **5:04 am** a new instance is launched by the auto-scaler. This is *not* because the Scale-out alarm was triggered again. This alarm remained triggered throughout the course of the test, as can be seen in Trial 6 – AWS CloudWatch Request Count chart. However, because the Scale-in alarm prompted the auto-scaler to shrink the web server pool and simultaneously, the Scale-out alarm was still turned "on", it once again prompted the auto-scaler to increase the pool from **2** to **3** instances.
- The test was stopped at **5:05 am**
- By **5:09 am**, the auto-scaler terminated an instance, going back to the minimum of **2** instances. This process took 4 minutes since the test was stopped.
- Scale-out alarm back to OK by **5:08 am**

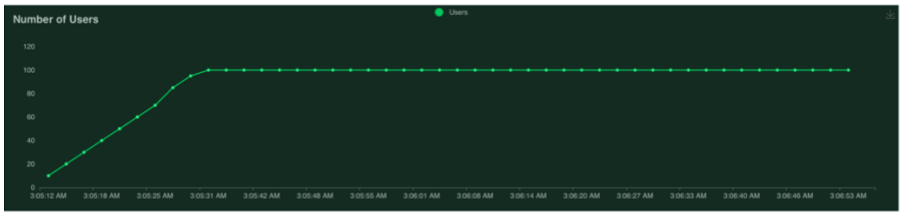
Appendix

1. Trial 1: 100 users at 5 users/ second

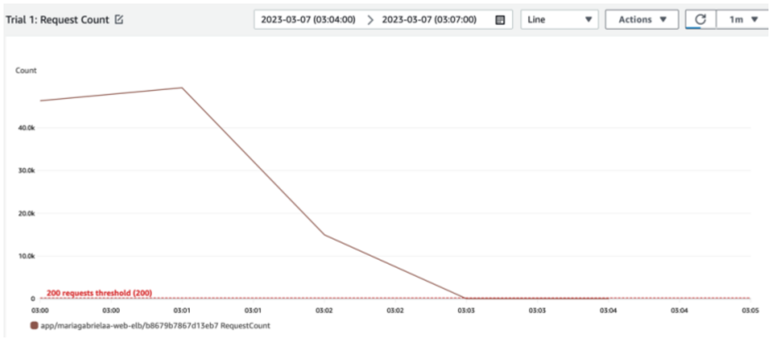
1.1 Trial 1 – Locust charts







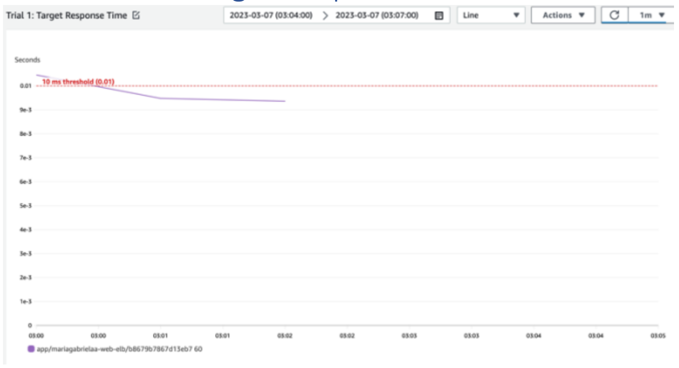
1.2 Trial 1 – AWS CloudWatch Request Count chart


Trial 1: Request Count

2023-03-07 (03:04:00) > 2023-03-07 (03:07:00)

Line

1.3 Trial 1 – AWS CloudWatch Target Response Time chart

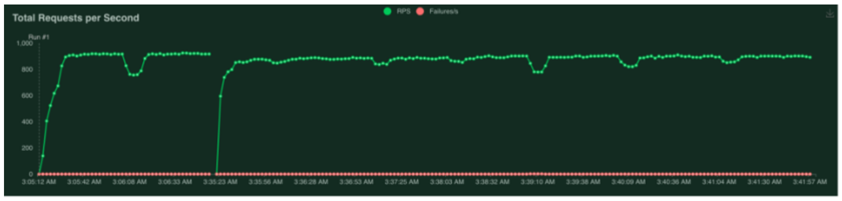

Trial 1: Target Response Time

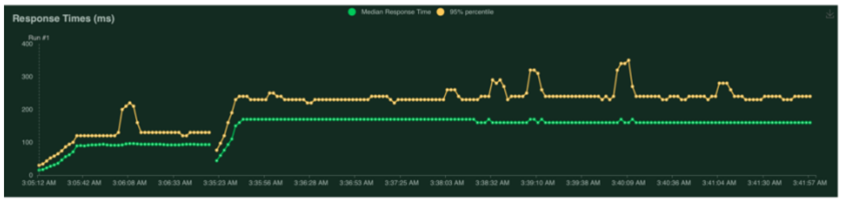
2023-03-07 (03:04:00) > 2023-03-07 (03:07:00)

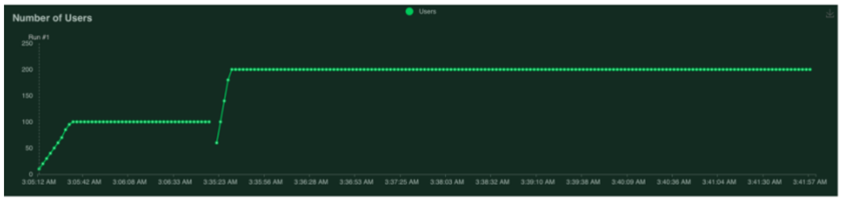
Line

2. Trial 2: 200 users at 20 users/ second

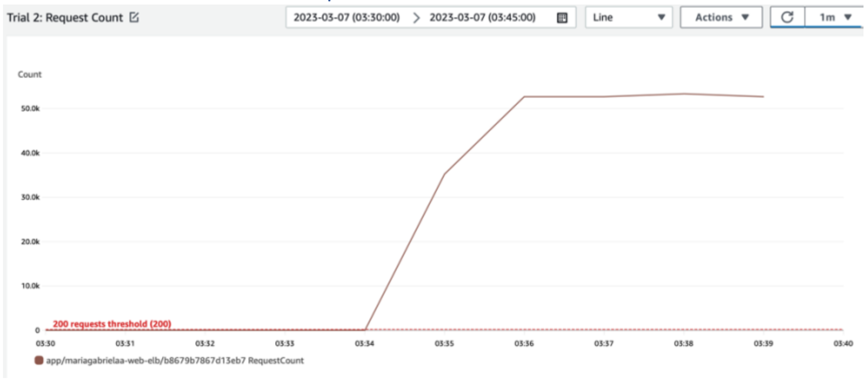
2.1 Trial 2 – Locust charts



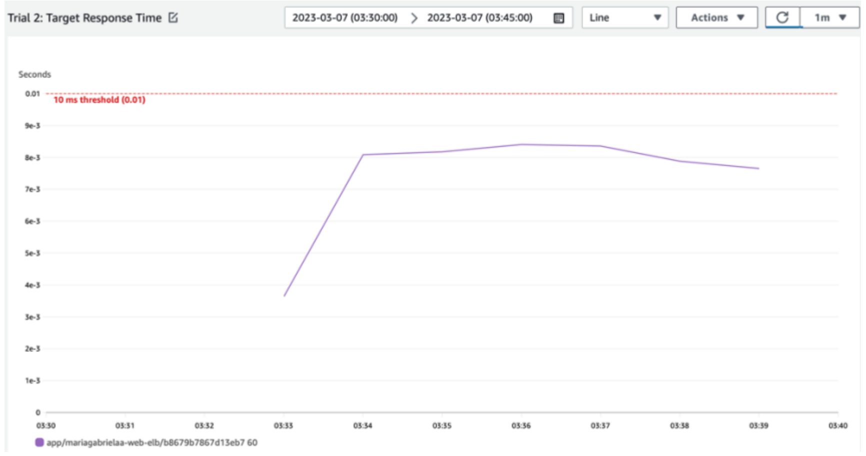




2.2 Trial 2 – AWS CloudWatch Request Count chart

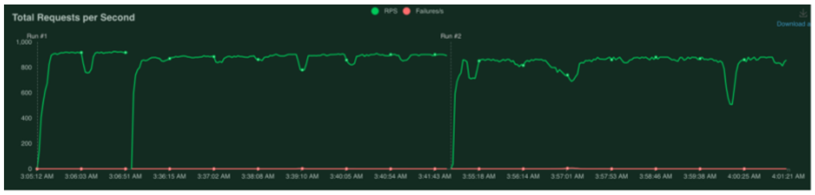


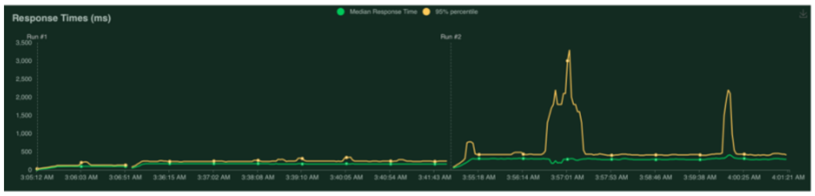
2.3 Trial 2 – AWS CloudWatch Target Response Time chart

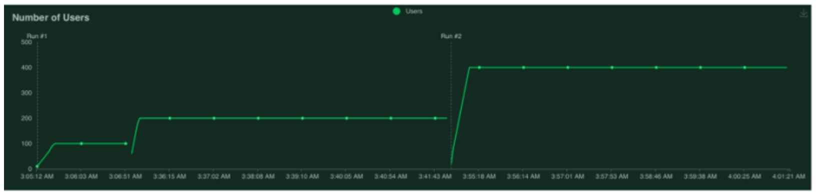


3. Trial 3: 400 users at 20 users/ second

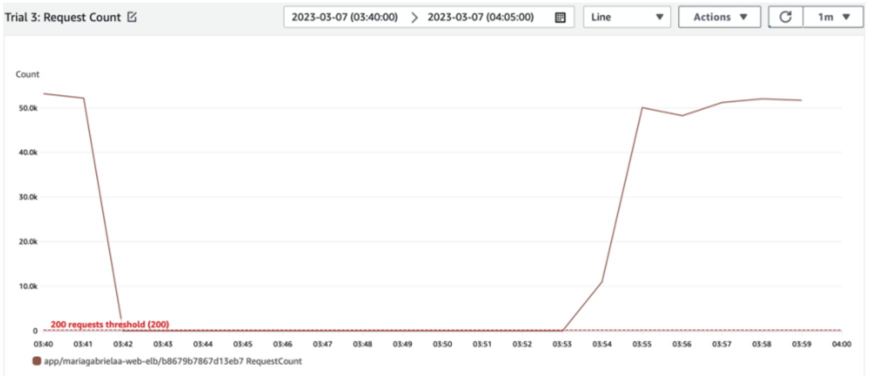
3.1 Trial 3 – Locust charts



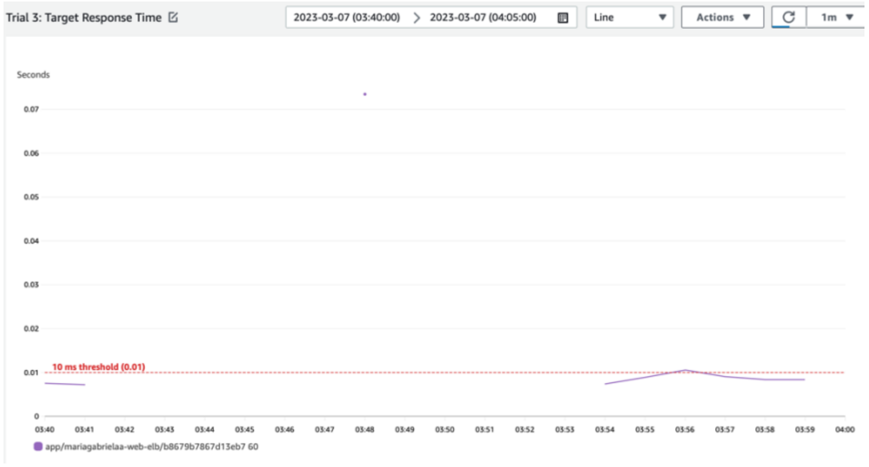




3.2 Trial 3 – Request Count chart

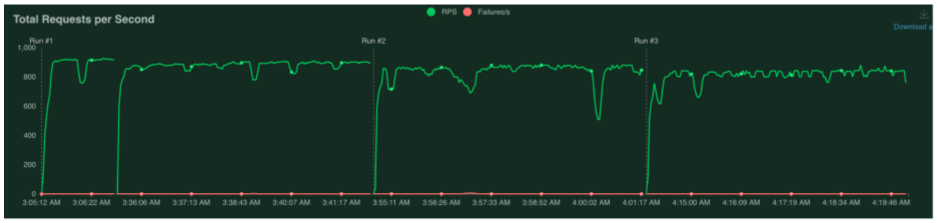


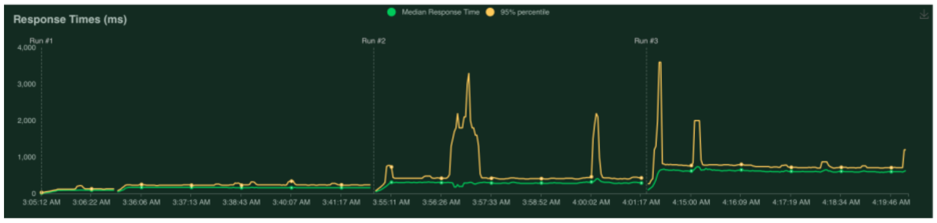
3.3 Trial 3 – AWS CloudWatch Target Response Time chart

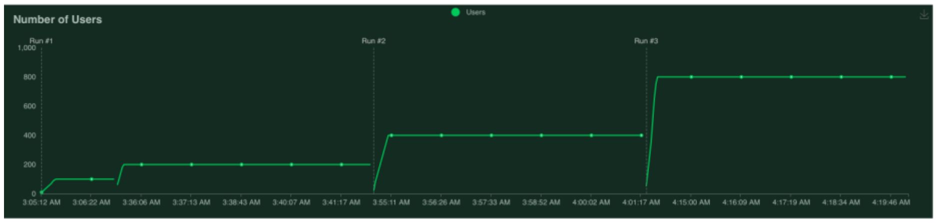


4. Trial 4: 800 users at 50 users/ second

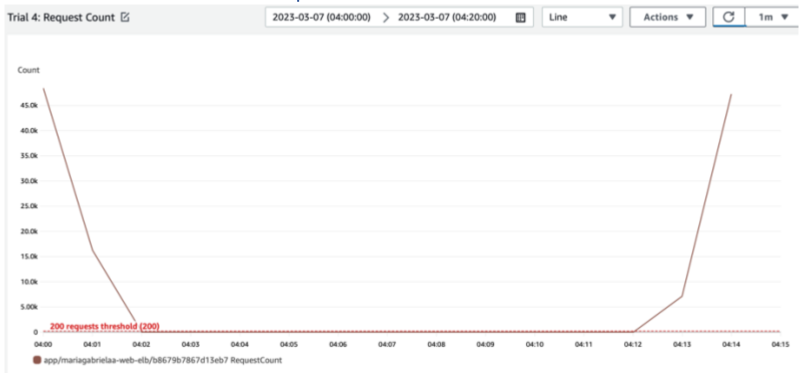
4.1 Trial 4 – Locust charts





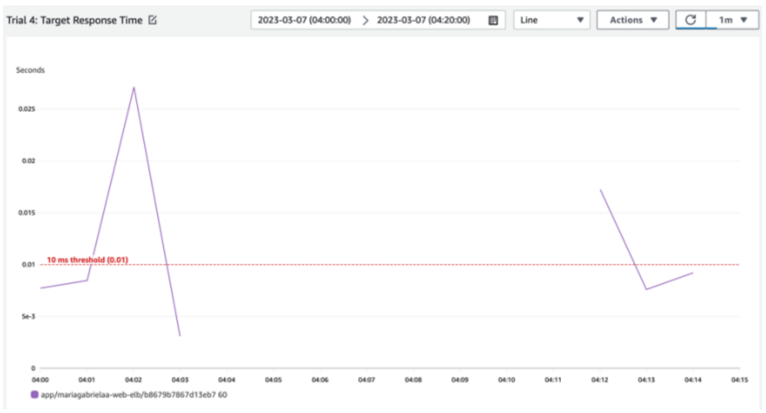


4.2 Trial 4 – AWS CloudWatch Request Count chart



Trial 4: Request Count 2023-03-07 (04:00:00) > 2023-03-07 (04:20:00) Line Actions 1m

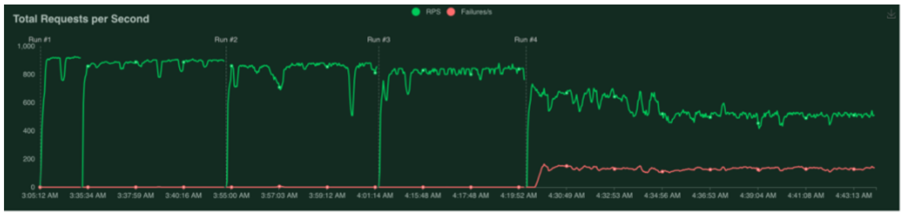
4.3 Trial 4 – AWS CloudWatch Target Response Time chart

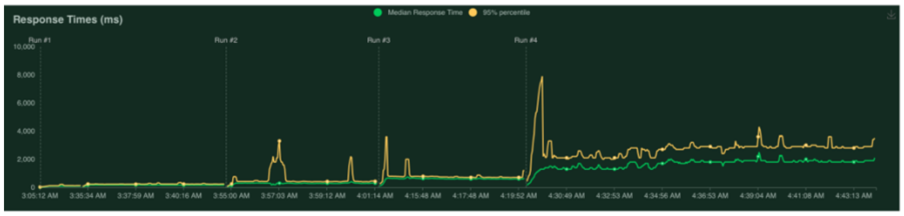


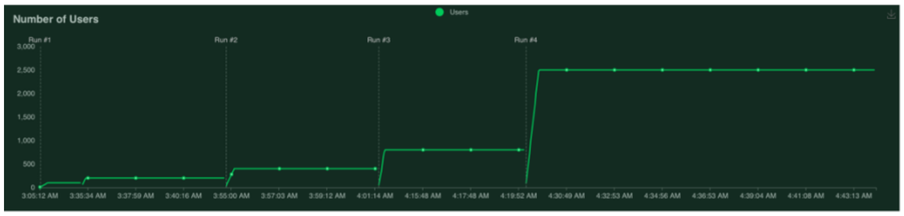
Trial 4: Target Response Time 2023-03-07 (04:00:00) > 2023-03-07 (04:20:00) Line Actions 1m

5. Trial 5: 2500 users at 80 users/ second

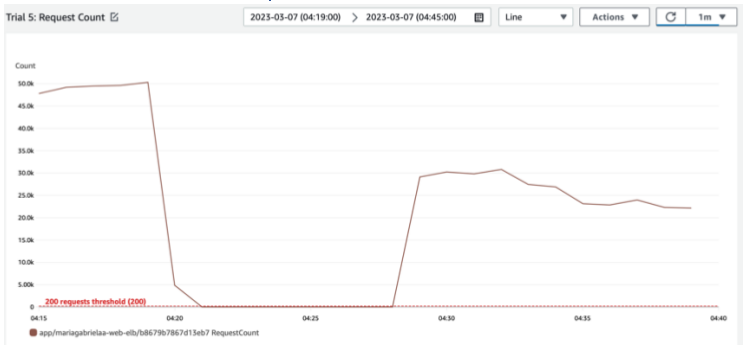
5.1 Trial 5 – Locust charts



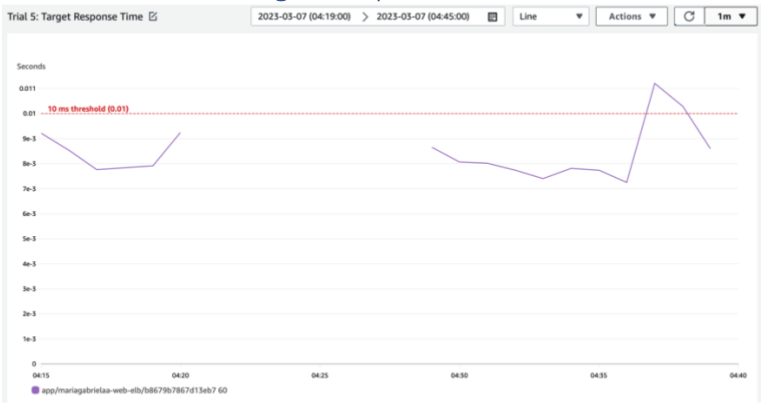




5.2 Trial 5 – AWS CloudWatch Request Count chart

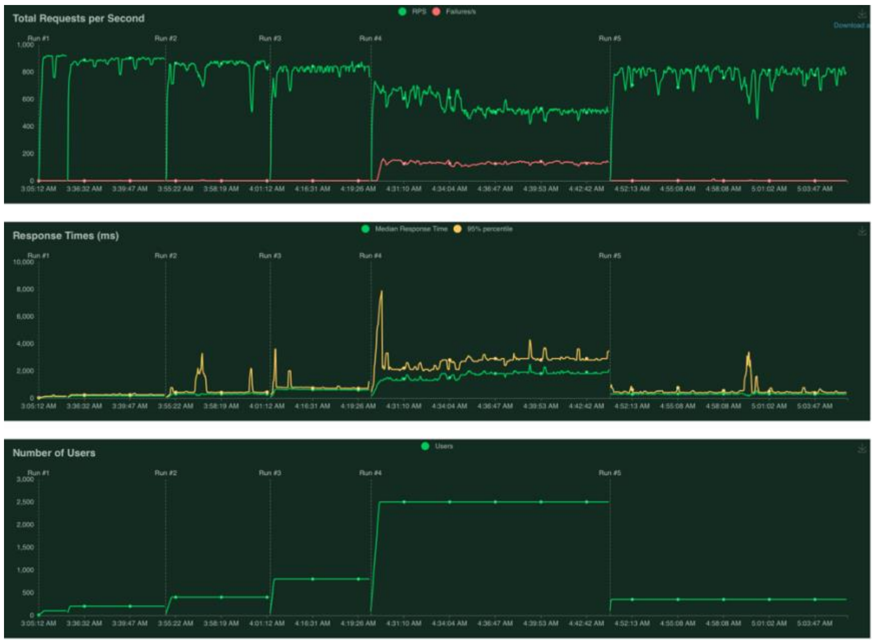


5.3 Trial 5 – AWS CloudWatch Target Response Time chart

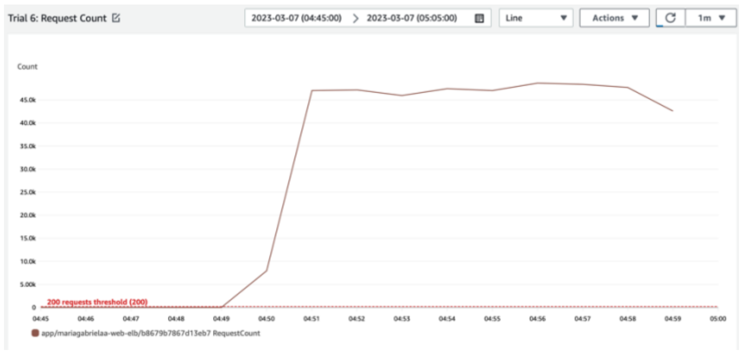


6. Trial 6: 350 users at 100 users/ second

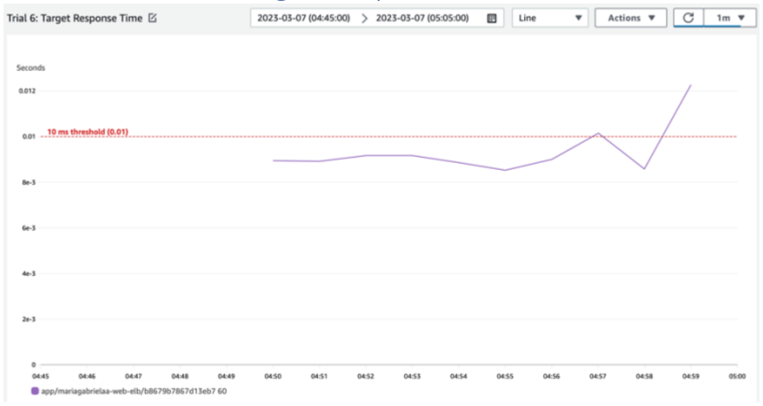
6.1 Trial 6 – Locust charts



6.2 Trial 6 – AWS CloudWatch Request Count chart



6.3 Trial 6 – AWS CloudWatch Target Response Time chart


Trial 6: Target Response Time
2023-03-07 (04:45:00)
2023-03-07 (05:05:00)
Line
Actions
1m
Seconds
10 ms threshold (0.01)
app/mariagabrielaa-web-elb/b8679b7867d13eb7 60