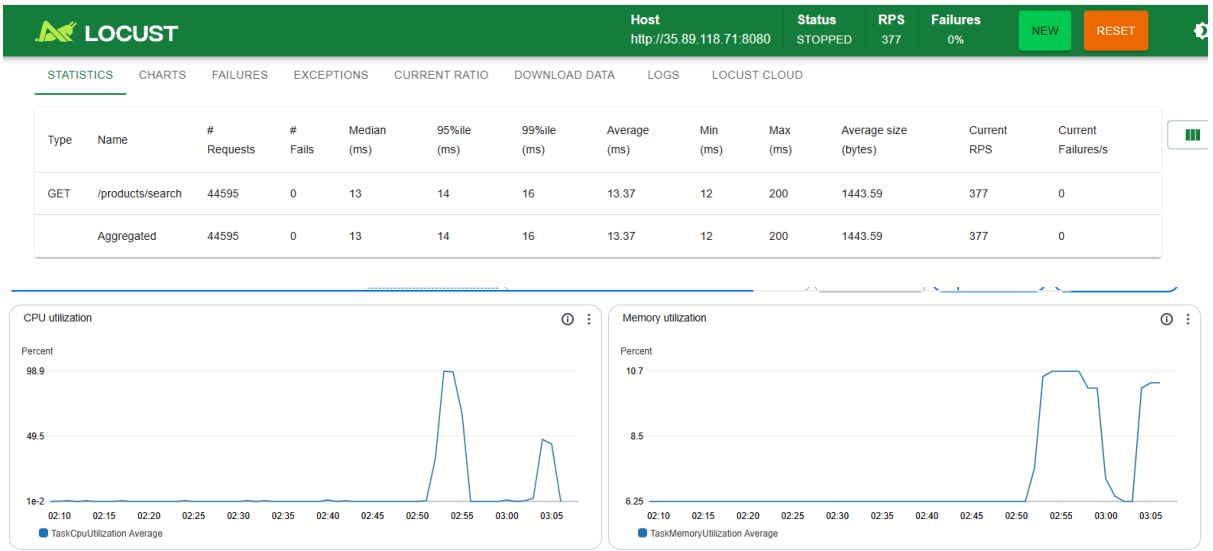
# Part 2 Report — Results

## 1. Behavior When Load Increased

Two Locust load tests were conducted on a single ECS task running the `/products/search` endpoint:

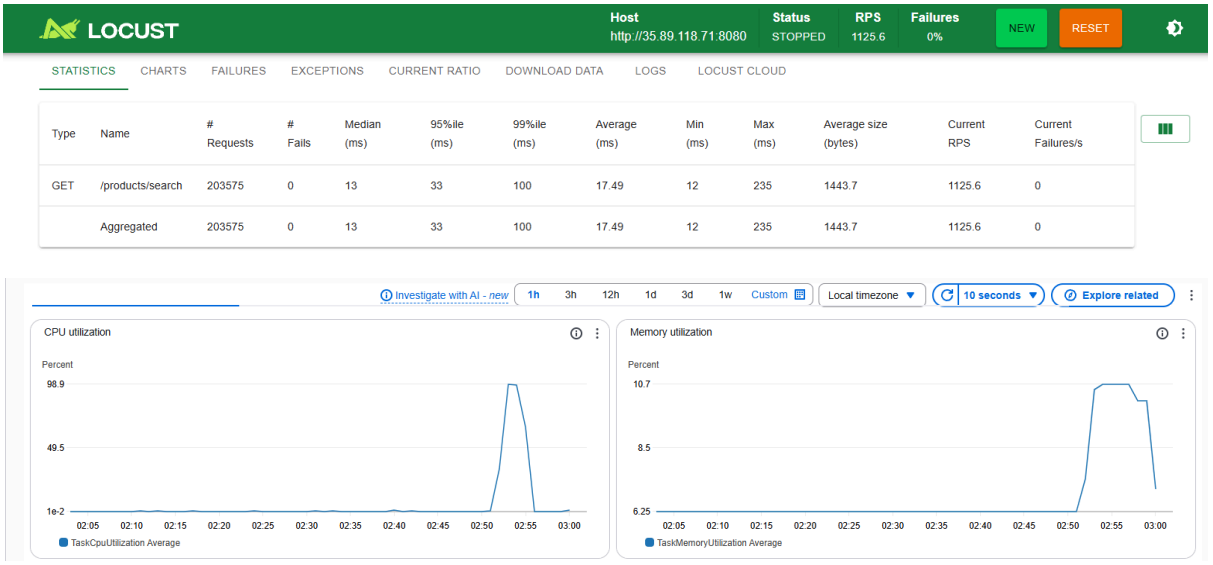
- **Test 1:** 5 users, 5 spawn rate, 2 minutes
- **Test 2:** 20 users, 20 spawn rate, 3 minutes

Under the **5-user test**, the system remained very stable:




- Average latency: **13.37 ms**
- 95th percentile: **14 ms**
- Requests per second (RPS): **~377**
- CPU utilization stayed very low, peaking around **10%**, with memory usage at ~8–10%.

Under the **20-user test**, the system handled over **203,000 requests**:




- Average latency: **17.49 ms**
- 95th percentile: **33 ms**
- RPS increased to **~1125**, showing excellent throughput
- CPU utilization spiked sharply to **~99%**, indicating full resource saturation
- Memory remained low (~10%), confirming the workload is CPU-bound, not memory-intensive.

2. So this seems like a limited compute resource for me since cpu utilization spikes to 99 percent. To improve, I believe there are two options, one is vertical scaling which is to have more cpu(eg. from 216 to 512), and another is horizontal scaling which is have multiple tasks to handle the search loop just like what we did in the map-reduce hw.

3. CloudWatch provided real-time visibility into resource utilization:

- **CPU utilization** clearly visualized the saturation threshold.
- **Memory utilization** confirmed consistent usage, validating that scaling decisions should be based on CPU, not memory metrics.

But I also found out that it often has latency when displaying cpu usage. When my locust test finished, it started to show data.

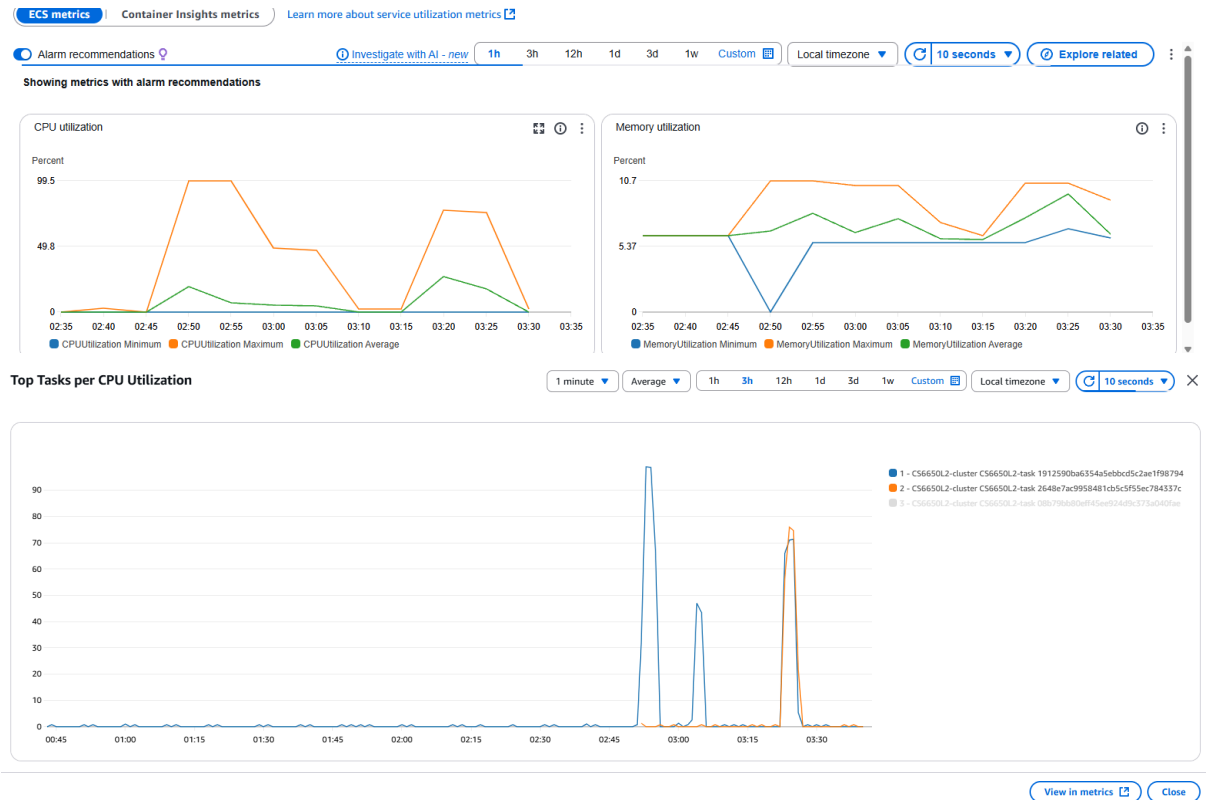
# Part 3 Report — Horizontal Scaling (Load Balancer + ECS Auto Scaling)

## 1. Test Setup

This test reused the exact load pattern that previously maxed out a single task in Part 2:

- **20 users, 20 spawn rate, 3 minutes** via Locust
- Host: **Application Load Balancer (ALB)** pointing to the ECS service
- ECS service configured with **Auto Scaling Target Tracking** on **CPU utilization = 70 %**, minimum 2 tasks, maximum 4 tasks

## 2. System Behavior Under Load




From the ECS → CloudWatch metrics:

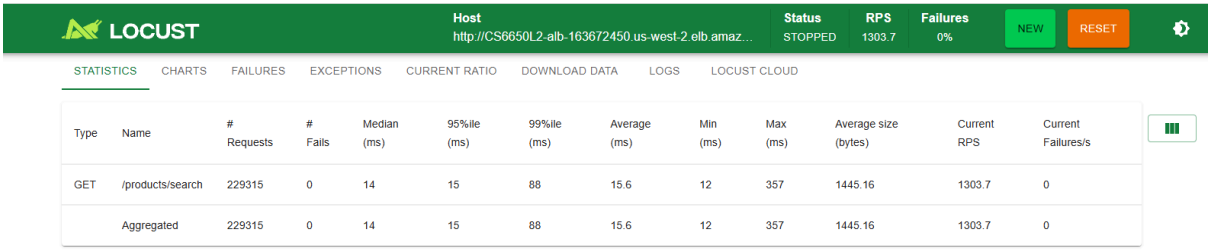
- **CPU Utilization:** peaked around 99 % on individual tasks for part 2 stress test set up, but now with alb and auto scaling, it averaged ~60–70 % overall, because traffic was split across 2 running tasks. And we did not see 50 percent cpu utilization for each task. I believe it is because when running on a single task, 99 % is limiting the capacity, and this leads to full use of cpu when we scale to two tasks.
- **Memory Utilization:** steady at ~10 %, confirming the workload remains CPU-bound.

- **Scaling Activity:**

| Name | State | Last state update (Local) | Conditions | Actions |
|---|---|---|---|---|
| TargetTracking-service/CS6650L2-cluster/CS6650L2-AlarmLow-534e7add-4612-4bef-be52-efa103ab04ef | OK | 2025-10-13 03:27:23 | CPUUtilization < 63 for 15 datapoints within 15 minutes | Actions enabled |

  no new task was launched, as the *average* CPU metric across tasks stayed below the 70 % target for the full evaluation period(it alarms the ecs when a certain data point confirms that cpu runs above 70, and in this test case for this short time, I believe it is possible that we can not see a trigger to more tasks).

- **Locust Results:**




| 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 | /products/search | 229315 | 0 | 14 | 15 | 88 | 15.6 | 12 | 357 | 1445.16 | 1303.7 | 0 |
| | Aggregated | 229315 | 0 | 14 | 15 | 88 | 15.6 | 12 | 357 | 1445.16 | 1303.7 | 0 |

  - Requests = **229 k**, Avg Latency ≈ 15.6 ms (95th %ile 15 ms)
  - RPS ≈ **1 300** with 0 failures

Compared to Part 2 (single task → 99 % CPU → 17 ms latency), the system stayed healthy and performant because the ALB distributed requests evenly across 2 containers.

## 3. How Scaling Decisions Are Calculated

ECS Target-Tracking policies work by sampling **CloudWatch CPUUtilization** metrics every minute.
Each **data point** represents the *average CPU* across all running tasks.
In this setup:

- **Scale-Out:** triggered only if > 70 % for 3 consecutive data points (~3 minutes)
- **Scale-In:** triggered if < 63 % for 15 data points (~15 minutes)

Because the two tasks shared the load almost evenly, the average CPU stayed below 70 %, so no scale-out alarm was fired.
This demonstrates that starting with 2 tasks already gave sufficient compute headroom for the tested load.

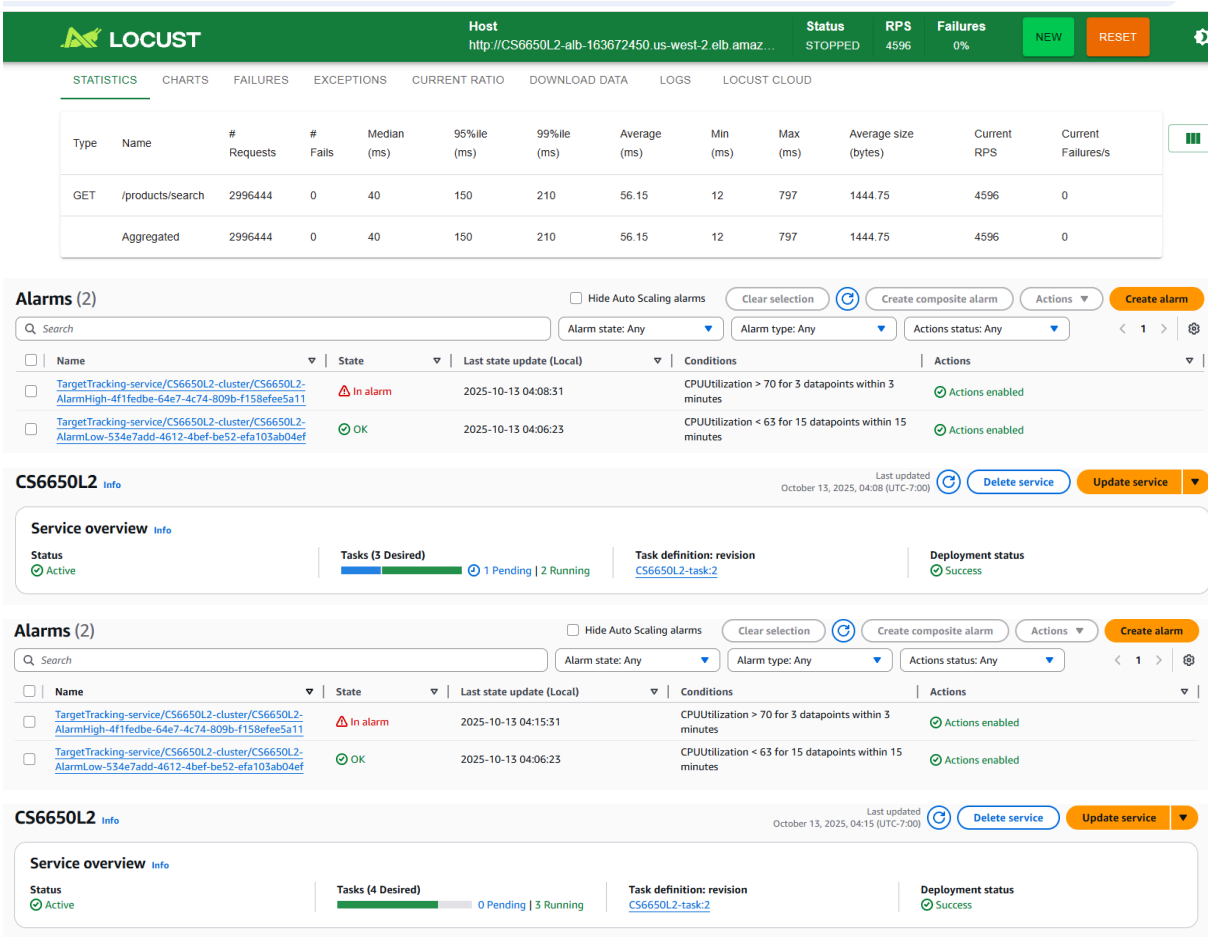
## 4. More tests

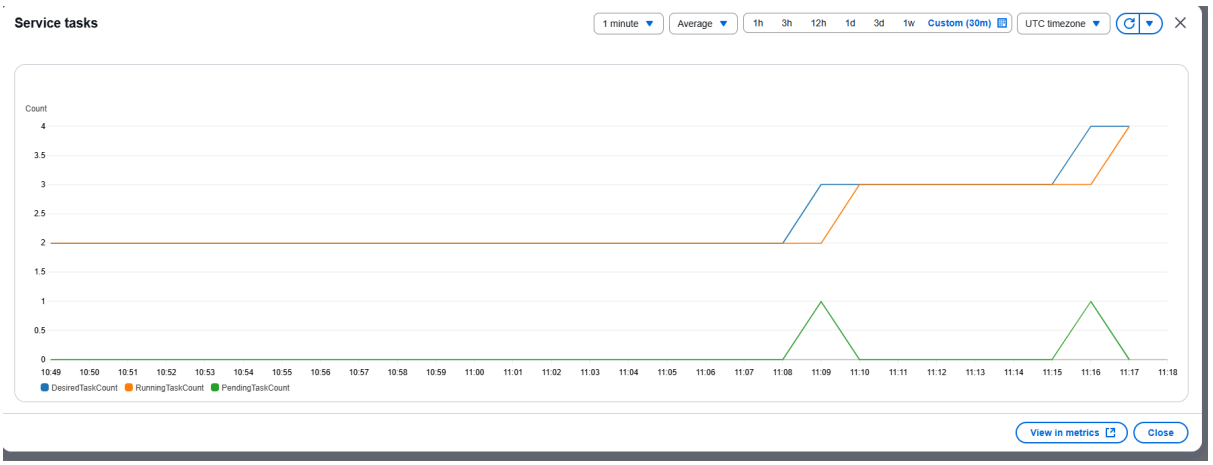
This test set up as below:

- **200 users, 200 spawn rate, 15 minutes** via Locust
- Host: **Application Load Balancer (ALB)** pointing to the ECS service

- ECS service configured with **Auto Scaling Target Tracking** on **CPU utilization = 70 %**, minimum 2 tasks, maximum 4 tasks, **cool down** 300 seconds.

Results:







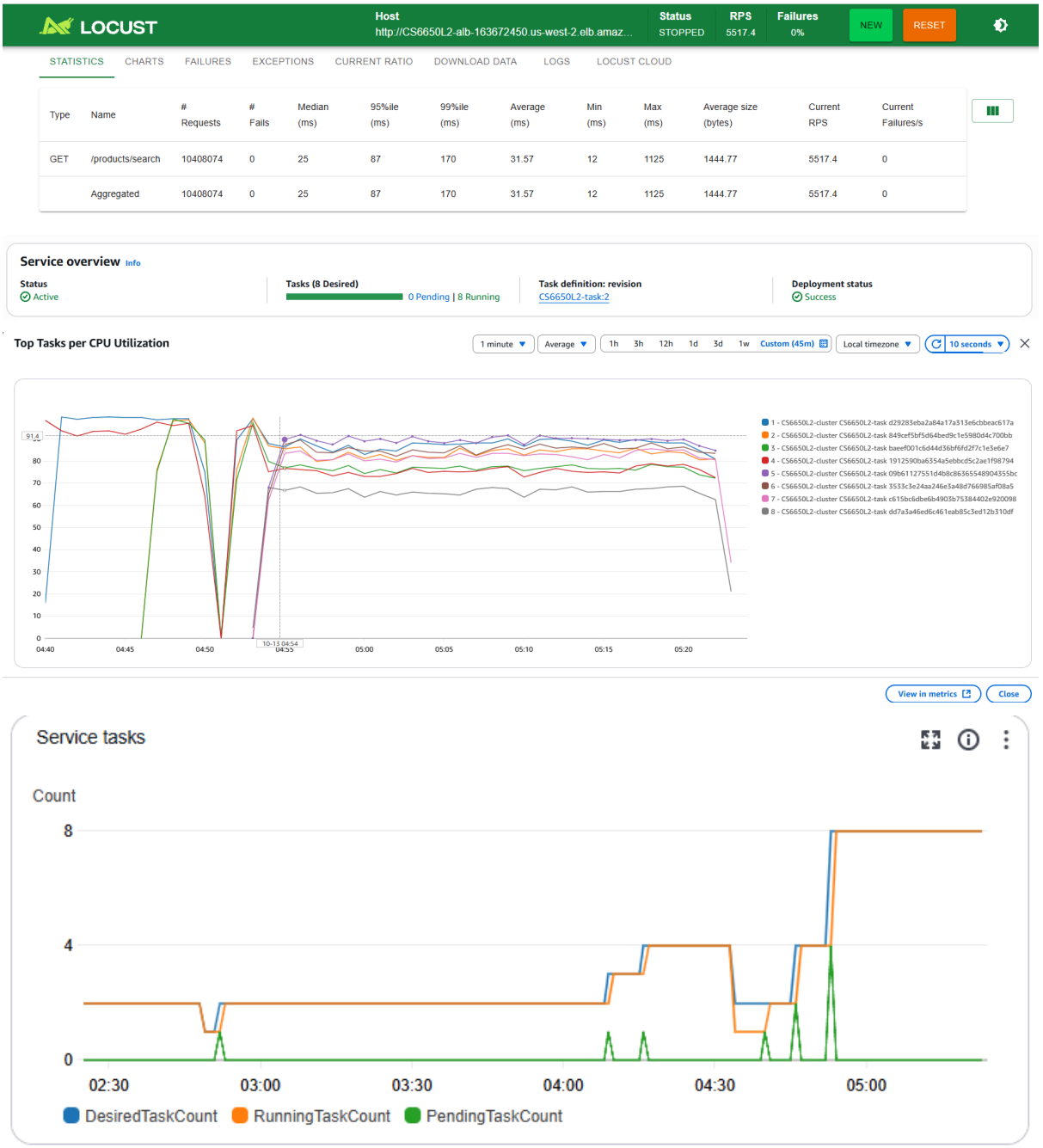
From the above experiment for stress test, we can see it steady scaling in 3-5 mins which matches with the scaling policy, cooldown plus the time ecs dealing with requests. And we can see it still reaches the system's bottleneck, this leads to latency in response rate and a non-linear increase of RPS.

So I decided to increase the tasks to see how it improves the response rate and throughput.

This test set up as below:

- **200 users, 200 spawn rate, 15 minutes** via Locust
- Host: **Application Load Balancer (ALB)** pointing to the ECS service
- ECS service configured with **Auto Scaling Target Tracking** on **CPU utilization = 70 %**, minimum 2 tasks, maximum 8 tasks, **cool down** 120 seconds, **target** at 50.

Results:




Increasing the max tasks for auto scaling shows an increase in the throughput which did not surprise me since the previous test is limiting the capacity, although I do believe 8 tasks still can not handle all the requests, it still shows 20% increase in throughput and 44% in latency.

## 5. More analysis

1.Explain how the system solved your Part II bottleneck

The ALB evenly routed incoming traffic between healthy ECS tasks in the target group.

This distribution reduced the CPU load on individual containers, preventing the sharp spikes seen in Part II.

ECS Auto Scaling ensured that when the CPU exceeded the 70% threshold for several consecutive data points, new tasks were automatically launched to absorb excess load.

## 2.Describe the role of each component (ALB, Target Group, Auto Scaling)

ALB: Routes incoming HTTP requests evenly across all running ECS tasks. Continuously performs health checks to remove failing instances from service.

Target Group: The logical grouping of ECS tasks registered with the ALB. It allows automatic registration and deregistration of tasks during scale-in and scale-out events.

Auto Scaling: Monitors average CPU utilization across tasks. Adds or removes tasks dynamically to maintain performance targets. The service ensures that the desired count matches current load.

## 3. Tradeoffs

Fault tolerance, when we vertical scale(having one big container with massive cpu power and one task), if that task fails due to some reasons, the whole system will be down, but using horizontal scaling, even if one task fails, others will pick up his work.

Easy to adjust, horizontal scale can easily scale in or scale out by monitoring cpu usage and I doubt vertical scale can do the same.

But vertical scale is simpler than horizontal scale since you do not need to set up that much stuff, just only need to keep adding cpu power to one container. And also I believe it will reduce many latencies since everything happens in one.