

# CS6650 – Distributed Systems Homework 6

## Performance Bottlenecks & Horizontal Scaling

AWS ECS Fargate | ALB | Auto Scaling | Terraform | Locust

## Part II — Identifying Performance Bottlenecks

### Service Setup

A Go product search service was deployed to ECS Fargate (0.25 vCPU, 512 MB). At startup it generates 100,000 products in a sync.Map. Each search checks exactly 100 products — simulating fixed-cost computation — and returns up to 20 results.

### Load Testing

Tests were run using Locust with FastHttpUser and minimal wait time (0.01–0.05s). The initial 20-user test produced only ~15% CPU — too low to observe meaningful load, as sync.Map lookups are very fast. The user count was raised to 50 to generate observable stress.

### Results

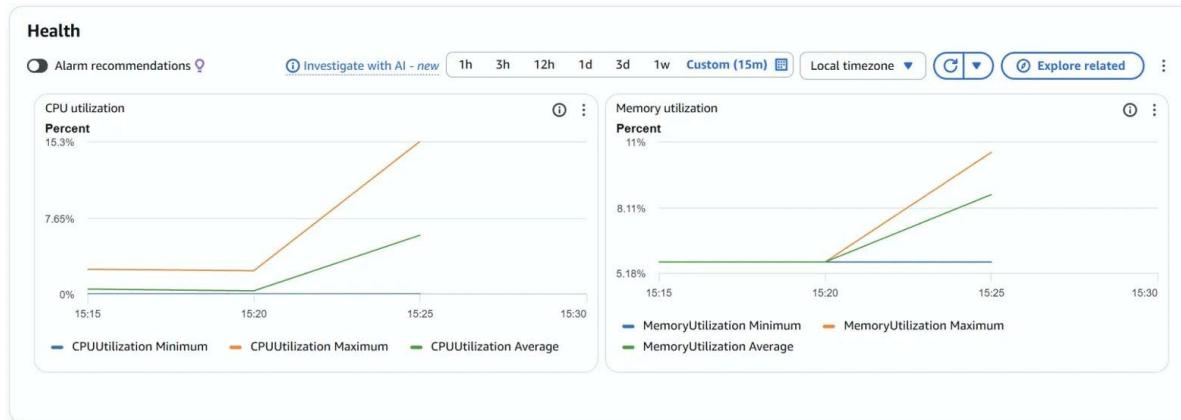


Figure 1: Baseline (5 users) — CPU peaked at 15.3%, memory stable at ~11%.

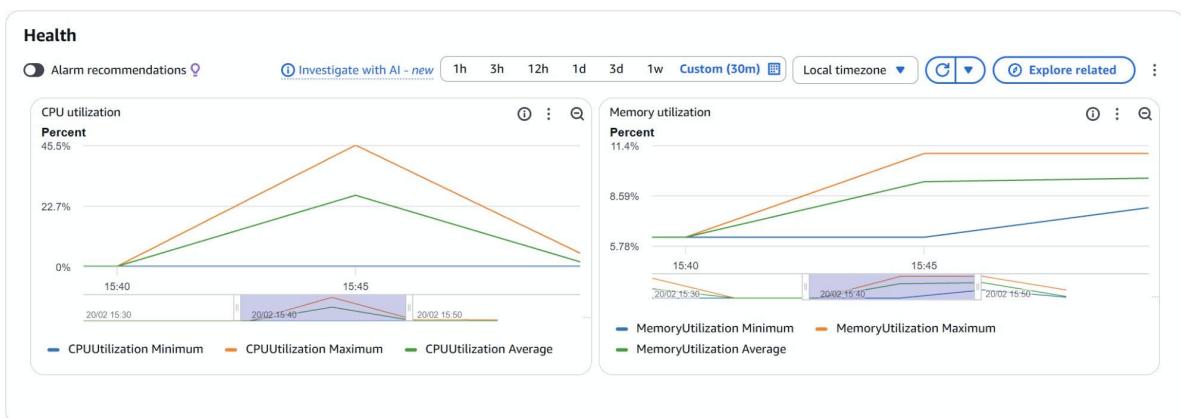


Figure 2: Stress test (50 users) — CPU peaked at 45.5%, average 22.7%. Memory remained flat.

| Metric       | 5 Users (Baseline) | 50 Users (Stress) |
|--------------|--------------------|-------------------|
| CPU Max      | 15.3%              | 45.5%             |
| CPU Average  | ~7%                | ~22.7%            |
| Memory       | ~11%               | ~11.4% (flat)     |
| Avg Response | ~12ms              | ~12ms             |
| Failures     | 0                  | 0                 |

## Analysis

CPU scaled linearly with concurrency while memory stayed flat — confirming a CPU-bound workload. Each request has a fixed compute cost (100 product string comparisons) that cannot be optimized away. Response times were stable because the service had not yet fully saturated, but at higher concurrency CPU would hit 100% and queuing would cause degradation. The solution is not better code — it is more compute power.

Doubling CPU (256 → 512 units) would roughly double throughput, but hits a ceiling and remains a single point of failure. The better answer is horizontal scaling, implemented in Part III.

## Part III — Horizontal Scaling with ALB & Auto Scaling

### Infrastructure

Three components were added via Terraform on top of the Part II service:

- Application Load Balancer (ALB) — receives all traffic and distributes requests across healthy instances
- Target Group — tracks which ECS tasks are healthy via /health checks every 30s; unhealthy instances are automatically removed from rotation
- Auto Scaling — target tracking policy: maintain average CPU at 50%, min 2 / max 4 instances, 60s cooldown

## Initial State (2 Tasks)

The screenshot shows the AWS ECS Cluster Overview for the cluster 'CS6650L2-cluster'. The cluster status is 'Active' and it has 2 registered container instances. Under the 'Tasks' tab, there are 2 running tasks, both defined by the task definition 'CS6650L2-task:6'. The tasks have been running for 34 minutes ago.

| Task                     | Last status | Desired status | Task definition | Health status | Created at     | Started by             |
|--------------------------|-------------|----------------|-----------------|---------------|----------------|------------------------|
| b064f04147aa407ebde9c... | Running     | Running        | CS6650L2-task:6 | Unknown       | 34 minutes ago | ecs-svc/67664851133... |
| e9cd42ca40484a97818d...  | Running     | Running        | CS6650L2-task:6 | Unknown       | 34 minutes ago | ecs-svc/67664851133... |

Figure 3: ECS cluster with 2 running tasks at startup (minimum instance count).

The screenshot shows the AWS Lambda Target Groups page. There is one target group named 'CS6650L2-tg' which routes requests to two healthy targets: IP addresses 172.31.60.51 and 172.31.45.234, both on port 8080.

| Name        | ARN                            | Port | Protocol | Target type | Load balancer | VPC ID                |
|-------------|--------------------------------|------|----------|-------------|---------------|-----------------------|
| CS6650L2-tg | arn:aws:elasticloadbalancin... | 8080 | HTTP     | IP          | CS6650L2-alb  | vpc-0b50bf7fc2b8cd5aa |

**Target group: CS6650L2-tg**

| IP address    | Port | Zone           | Health status | Health status details | Administrative override | Override details          | Anomaly detection |
|---------------|------|----------------|---------------|-----------------------|-------------------------|---------------------------|-------------------|
| 172.31.60.51  | 8080 | us-west-2d ... | Healthy       | -                     | No override             | No override is current... | Normal            |
| 172.31.45.234 | 8080 | us-west-2b ... | Healthy       | -                     | No override             | No override is current... | Normal            |

Figure 4: Target Group showing 2 healthy registered targets before load testing.

## Stress Test — 500 Users, 5 Minutes

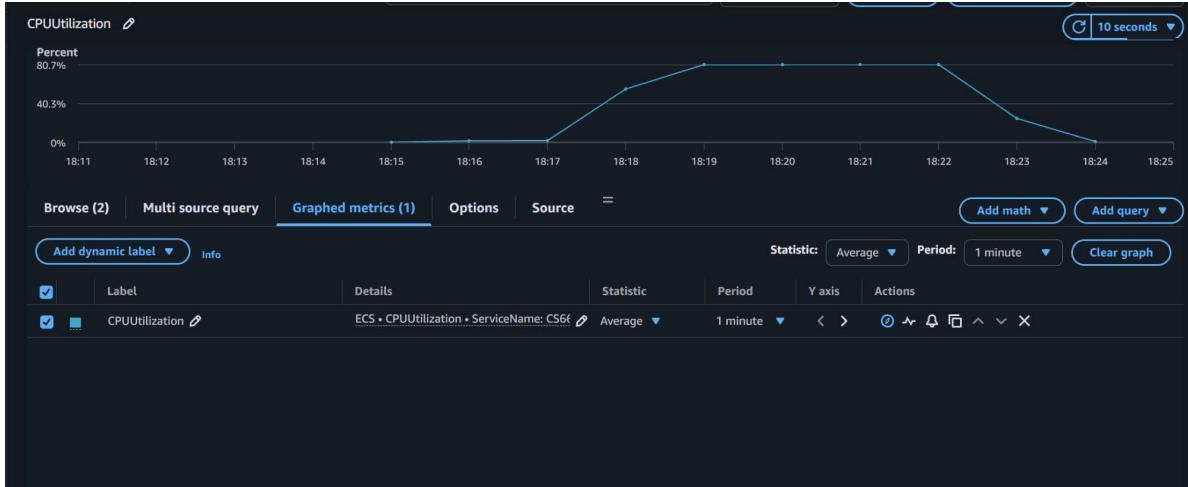


Figure 5: CloudWatch CPU during 500-user test. CPU peaked at ~80.7%, triggering Auto Scaling events.

| Scaling activities                |                          |            |  |                     |                     |  |
|-----------------------------------|--------------------------|------------|--|---------------------|---------------------|--|
| Resource ID                       | Scalable dimension       | Status     | Status message   | Start time          | End time            | Description                                |
| service/CS6650L2-cluster/CS6650L2 | ecs:service:DesiredCount | Successful | Successfully set desired count to 4. Change successfully fulfilled by ecs. | Feb 20, 2026, 18:24 | Feb 20, 2026, 18:25 | Setting desired count to 4.                |
| service/CS6650L2-cluster/CS6650L2 | ecs:service:DesiredCount | Successful | Successfully set desired count to 3. Change successfully fulfilled by ecs. | Feb 20, 2026, 18:23 | Feb 20, 2026, 18:24 | Setting desired count to 3.                |
| service/CS6650L2-cluster/CS6650L2 | ecs:service:DesiredCount | Successful | -  | Feb 20, 2026, 18:25 | -                   | Attempting to scale due to alarm triggered |

Figure 6: Auto Scaling activities — desired count increased from 2 → 3 (18:23) → 4 (18:24) as CPU exceeded 50% threshold.

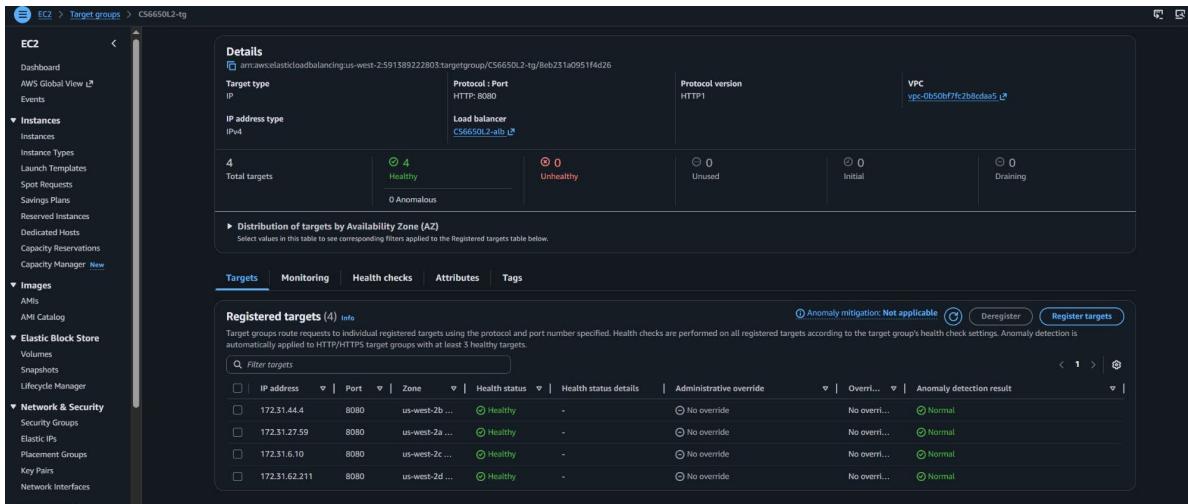


Figure 7: Target Group at peak load — 4 healthy targets across 4 availability zones.

## Key Observations

- Auto Scaling fired within ~2 minutes of sustained high CPU, incrementally adding instances

- All 4 instances registered as healthy in the Target Group and received traffic
- Response times remained stable throughout scaling events — no degradation observed
- Resilience test: manually stopping one task caused the Target Group to remove it within ~30s; Locust reported zero failures

## Vertical vs Horizontal Scaling

|                 | Vertical (Scale Up)            | Horizontal (Scale Out)               |
|-----------------|--------------------------------|--------------------------------------|
| Approach        | Larger instance (more CPU/RAM) | More instances of same size          |
| Ceiling         | Hard limit per instance        | No practical limit                   |
| Fault Tolerance | Still single point of failure  | Instance failures handled gracefully |
| Best For        | Simple, stateful workloads     | Stateless, compute-bound workloads   |
| This Service    | Would help short-term          | Correct long-term solution           |

## Conclusion

Part II identified CPU as the bottleneck for a compute-bound workload where code optimization cannot reduce per-request cost. Part III demonstrated that horizontal scaling with an ALB and Auto Scaling policy resolves this bottleneck dynamically — distributing load across multiple instances, recovering from failures automatically, and scaling back down when demand drops.