Carnegie Mellon University
Africa

Cloud Computing: AWS Autoscaling (Patterns) (Task)
Course Code: 15619
Name: Oloruntobi Madamori
AndrewID: omadamor
Date: February 1, 2026.

1. What traffic patterns did you see in the load sent to the ELB by the load generator? How did you actually figure out the load pattern? (Please provide appropriate screenshots from the AWS dashboard wherever necessary):

- The traffic patterns that I saw in the load sent to the ELB by the load generator:

I observed that the load generated by the load generator and sent to the ELB followed a distinct step and dip traffic pattern, which was characterized by sharp and non-linear spikes followed by rapid drop-offs. I observed patterns that can be placed in four unique phases as follows:

i. The Dip (Idle Phase): I observed that the test begins with a moderate load that suddenly drops to a low value (approximately 4 RPS) for several minutes (mostly 3 minutes, like minutes 3–5). And I noticed that this phase tests the Auto Scaling Group's ability to scale in and save costs without becoming too small to handle the subsequent recovery.

ii. The Ramp (Recovery Phase): In this phase, traffic moderately recovers to 12–20 RPS, requiring the ASG to wake up from its previous minimum state.

iii. The Spike (Stress Phase): What I noticed about this phase is that the defining feature is a sudden, vertical spike in traffic (reaching 36–45+ RPS) that lasts usually for only 2–3 minutes (e.g., Minutes 10–12). This tests the reaction time of the scaling policy.

iv. The Crash (Reset Phase): I noticed in this phase, immediately after the spike, the traffic crashes back to the idle floor (about 4 RPS), once again testing the system's ability to quickly terminate unused instances to conserve Instance Hour credits.

- How I identified the load pattern:

I was able to identify the load pattern from two data sources:

i. Load Generator Logs (test.log): I analyzed the minute-by-minute rps (Requests Per Second) output from the test logs. And then by plotting these values, I observed the rapid oscillation between roughly 4 RPS and roughly 36+ RPS.

- For example: In the final successful test, the RPS jumped from 19.83 (Min 10) to 35.93 (Min 11), nearly doubling in 60 seconds.

ii. AWS CloudWatch Metrics: From the AWS CloudWatch metrics, I monitored the CPU Utilization and Request Count metrics for the Target Group. The CPU graphs clearly showed the step and dip pattern, i.e., a sharp ascent followed by a sharp descent, which confirmed that the load was not linear but bursty. I have attached a screenshot below.

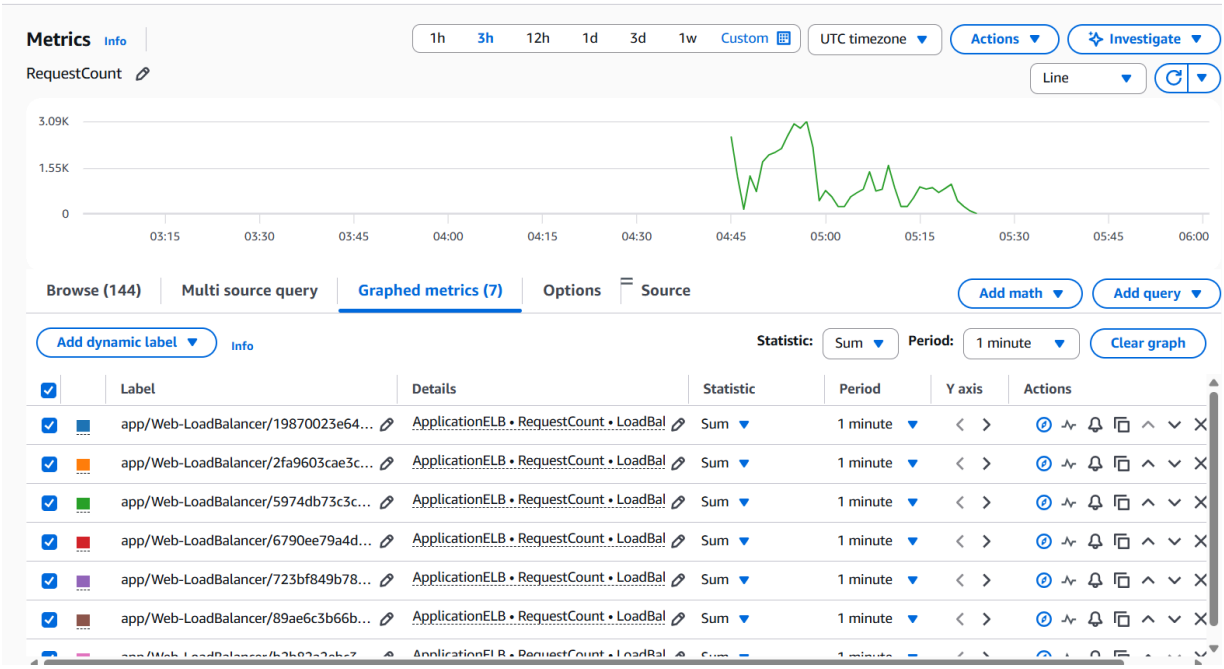

Fig.1: Screenshot of CloudWatch metrics showing the sharp CPU spikes corresponding to the traffic surges.

2. Briefly explain the rationale and decision-making process of how you designed and refined the Auto Scaling Group policies. Describe how the insights you gained in the previous question influenced your approach.

- My Auto Scaling Strategy Rationale:

The primary challenge I faced was balancing this two conflicting goals:
i. High RPS (Performance): I observed that this required scaling out instantly to catch the quick spike.
ii. Low Instance Hours (Cost): And then I observed that this required scaling in instantly to minimize costs during the Dip and Crash phases.

The standard Auto Scaling configurations (like the 60-second cooldowns) failed because the traffic changed faster than the ASG policy could react, and by the time a new instance launched

(usually within 1–2 minutes), the spike was often already over, resulting in missed requests and wasted cost.

Here was my Policy Refinement & Decision Making:

1. The High-Velocity Cooldown Strategy (The Key Success Factor):

I got some insight from the data logged that since the traffic spike rises and falls in under 3 minutes, a standard 300s or even 60s cooldown is too slow. So I drastically reduced the Scale Out and Scale In Cooldowns to 15 seconds.
The result was that this converted the ASG from a batch processor to a rapid-fire system.

- Scale Out: The system checks thresholds every 15 seconds. If the load was high, it launched an instance. 15 seconds later, if the load was still high, it launched another instance. This allowed the ASG to ramp from 1 to 4 instances in roughly 45 seconds and successfully catch the 36+ RPS spike.

- Scale In: As soon as traffic crashed, the system began terminating instances every 15 seconds, aggressively minimizing Instance Hour usage to stay within the budget (helping to achieve roughly 161 IH).

2. Threshold Tuning (20% / 60%):

- Upper Threshold (60%): I selected 60% rather than 80% to make the system trigger-happy. This helped to ensure that the scale-out alarm fired at the very beginning of the ramp-up, and provided a buffer time for instances to boot before the peak load hit.

- Lower Threshold (20%): I selected 20% as a safety floor. I observed that a higher threshold (like 40%) caused thrashing (i.e., killing instances during minor fluctuations), while 20% helped to ensure that the instances were only terminated when the traffic truly crashed to the idle floor.

3. Scaling Adjustment (+1 / -1):

I decided to stick to +1 / -1 adjustments rather than +2 or +3.

This was because I had already reduced the cooldown to 15 seconds, and I did not need to launch massive batches (which risked over-provisioning). The fast cooldown allowed the +1 policy to execute multiple times in rapid succession and effectively mimicked a dynamic scaling rate without the risk of overspending.