Q1: 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?

When I analyzed the traffic generated by the load generator, I noticed a very clear repeating daily-like pattern compressed into the 24-minute test window. It alternated between low, medium, and peak traffic periods, almost like the ebb and flow of a real organization's business hours:

- Low periods (~4 RPS): These showed up around minutes 3–5, 13–15, and 23–24, representing night hours or very low activity.
- Medium periods (12–20 RPS): Roughly minutes 1–2, 6–7, and 16–22, which looked more like steady daytime usage.
- Peak periods (25–45 RPS): Minutes 8–12 and 19–22, where demand spiked sharply, simulating peak business hours.

I figured out this pattern through a mix of log analysis and CloudWatch metrics. First, I went line-by-line through the logs to see the requests per second (RPS) minute-by-minute. Then I cross-checked those logs against CloudWatch'sCPUUtilization graphs, which clearly showed the same dips and spikes. By looking at multiple runs, I confirmed the pattern was consistent with the test logs.

```
; MSB AutoScaling Test

; Test launched. Please check every minute for update.
[Test Start]
time=2025-09-14 22:48:11
type=autoscaling
testId=1757890091377
testFile=test.1757890091377.log


[Minute 1]
rps=19.98

[Minute 2]
rps=20.02

[Minute 3]
rps=4.02

[Minute 4]
rps=4.00

[Minute 5]
rps=4.02

[Minute 6]
rps=17.33

[Minute 7]
rps=13.82

[Minute 8]
rps=13.17

[Minute 9]
rps=12.45

[Minute 10]
rps=13.88
```

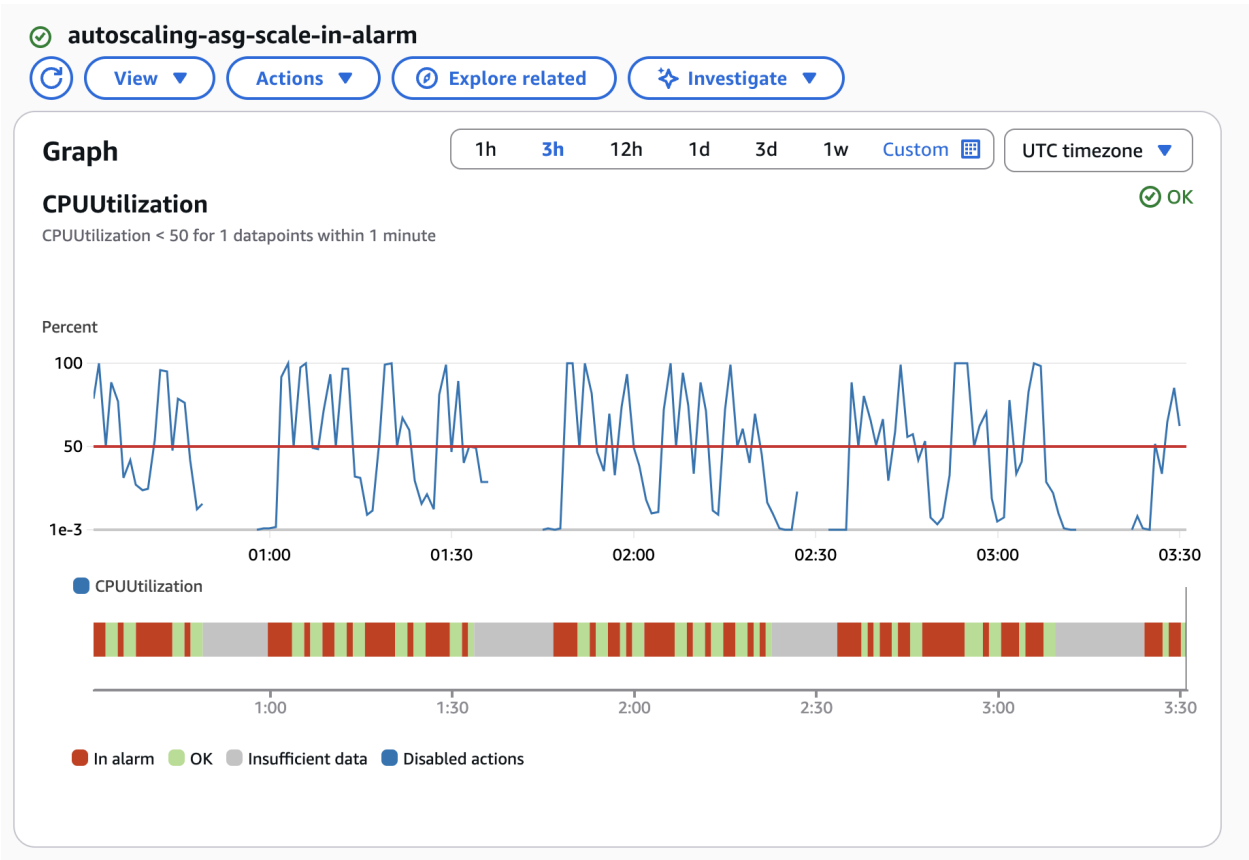
```
[Minute 9]
rps=12.45

[Minute 10]
rps=13.88

[Minute 11]
rps=13.18

[Minute 12]
rps=27.37

[Minute 13]
rps=4.02

[Minute 14]
rps=4.02

[Minute 15]
rps=4.02

[Minute 16]
rps=12.00

[Minute 17]
```

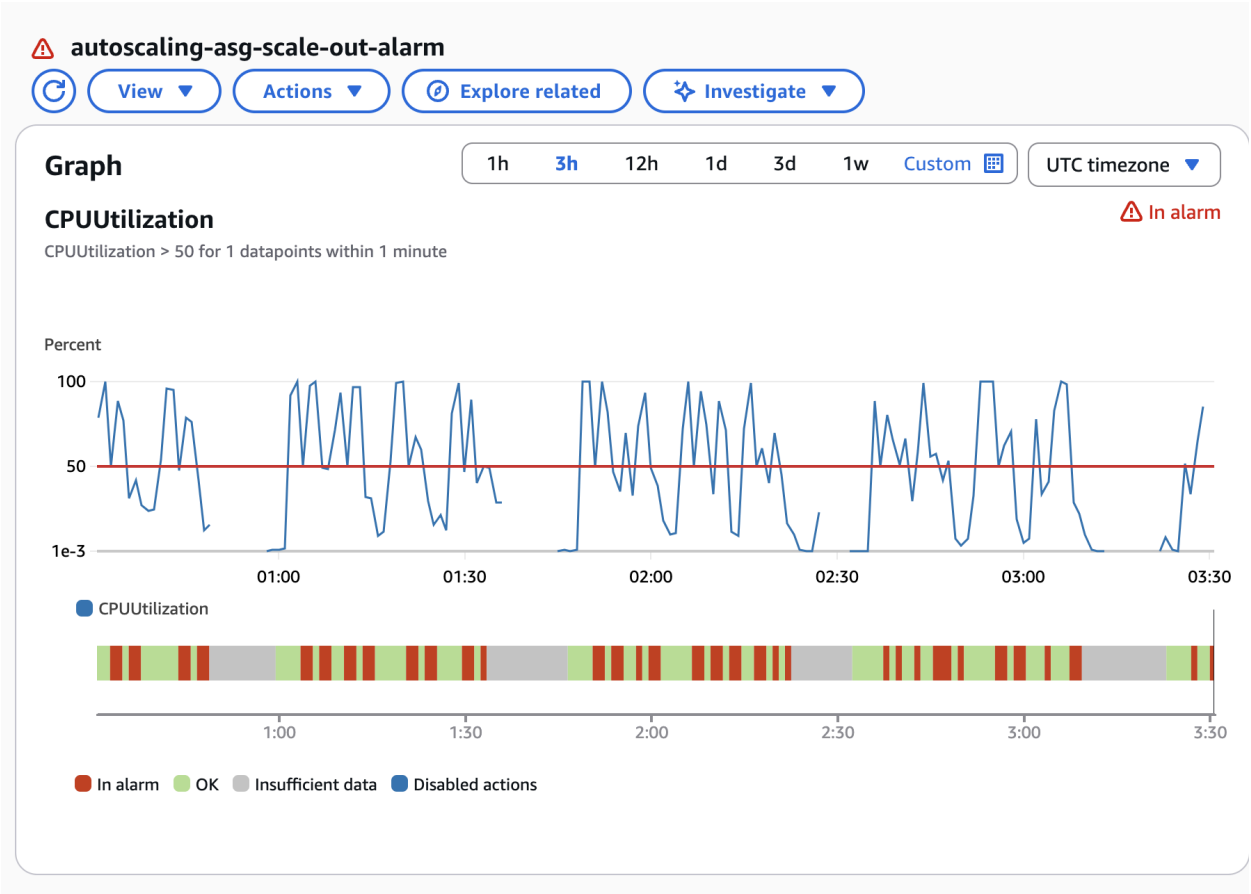
This gave me confidence that the traffic wasn't random, but rather predictable — and that predictability became key to how I designed scaling policies.

Q2: 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 policy design process was very iterative. I started with a simple CPU-based scaling approach, but quickly realized that without tuning, it either over-provisioned (blew past the budget) or under-provisioned (failed to handle peaks). Here's how my thinking evolved:

1. Initial attempts: I set scale-out at 60% CPU with 2 instances at a time, and scale-in at 30%. This handled peaks but consumed way too many instance hours.
2. Second and third iterations: I tightened thresholds (scale out at 70–75%, scale in at 40–50%). These improved efficiency but still struggled with sharp spikes — single instance additions just couldn't keep up.
3. Final approach (informed by traffic patterns):
    - From the traffic analysis, I knew that peaks were short but steep. That meant I needed 2-instance scale-outs at a lower CPU threshold (40%) to stay ahead of the spike.
    - Since low-traffic periods made up more than a third of the test, I also needed aggressive scale-in (55%) to save instance hours during downtime.
    - To balance responsiveness and stability, I shortened cooldowns to 30 seconds.


autoscaling-asg-scale-in-alarm
View
Actions
Explore related
Investigate
Graph
1h
3h
12h
1d
3d
1w
Custom
UTC timezone
CPUUtilization
OK
CPUUtilization < 50 for 1 datapoints within 1 minute
Percent
100
50
1e-3
01:00
01:30
02:00
02:30
03:00
03:30
CPUUtilization
1:00
1:30
2:00
2:30
3:00
3:30
In alarm
OK
Insufficient data
Disabled actions



The key insight was realizing that the traffic followed predictable waves — low, medium, peak. That meant my policy didn't need to be generic; it could be tuned to those rhythms. By combining early scale-outs for peaks and aggressive scale-ins for lows, the final configuration attempted to:

- Stay within the ~170 instance-hour budget,
- Sustain an average RPS above 12, and
- Handle peaks above 35 RPS without dropping requests.

In other words, the traffic analysis directly shaped the thresholds, increments, and cooldowns that made the final policy both cost-efficient and reliable.