Performance Evaluation of Autoscaling and Load Balancing Strategies in AWS EC2

Ammar Elzeftawy
Schulich School of Engineering
University of Calgary
Calgary, Canada
UCID: 30113872

Joseph Duong
Schulich School of Engineering
University of Calgary
Calgary, Canada
UCID: 30145210

Thevin Mahawatte
Schulich School of Engineering
University of Calgary
Calgary, Canada
UCID: 30130509

Brandon McGee
Schulich School of Engineering
University of Calgary
Calgary, Canada
UCID: 30125635

***Preamble*—Link to repository and data**
https://drive.google.com/drive/folders/1mDy1ROK4EnEGfx7MgMu8951Nt_X3w_kQ

***Abstract*—Cloud-based web applications often face challenges in maintaining performance under fluctuating traffic loads. Autoscaling and load balancing are essential mechanisms to ensure responsiveness and cost-efficiency. In this study, we evaluate the performance implications of using AWS EC2 auto scaling and different load balancing strategies on a simple web application. Using the K6 tool for load generation and InfluxDB plus AWS CloudWatch for monitoring, we simulate user traffic up to 1000 concurrent users. We compare three configurations: a static single-server setup, an auto scaled two-server system with round-robin load balancing, and an auto scaled two-server system using the least outstanding request load balancer algorithm. Results indicate significant improvements in latency and throughput when autoscaling is enabled, especially when combined with intelligent load balancing. The round-robin strategy provided most consistent performance, whereas the least outstanding request strategy handled similar loads with minor reliability trade-offs. These findings offer practical insight into default AWS auto scaling settings and their effectiveness in low-cost deployments.**

I. Introduction

Modern web applications must be scalable and resilient to handle dynamic traffic patterns. Auto scaling, dynamically adjusting computing resources based on demand, and load balancing, distributing incoming requests across servers, are widely used to maintain application performance during traffic spikes. This project investigates the performance of various auto scaling policies and load balancing strategies deployed on Amazon Web Services (AWS) Elastic Compute Cloud (EC2) instances, a foundational component in cloud infrastructure.

Amazon EC2 provides resizable virtual servers on demand, making it a cornerstone of cloud computing [1]. Cloud platforms enable organizations to scale resources elastically, avoiding high upfront costs of physical hardware. Although our study focuses on EC2's role in hosting web applications, the underlying principles extend to broader production environments. Misconfigured autoscaling strategies or poorly chosen instance types can lead to high latency, system instability, or excessive costs. While auto scaling is widely supported by major cloud providers, its real-world performance under budget constraints, especially when comparing strategies like reactive verse predictive scaling or different load balancing algorithms, remains underexplored in both academic and practical literature.

Previous studies have examined how cloud systems manage scaling with a focus on keeping performance high without inflating costs. Some research has analyzed AWS's **reactive** auto scaling, noting that while effective, it may cause delays if threshold triggers are not properly tuned [2]. Others have proposed advanced techniques such as reinforcement learning to adjust resources dynamically in response to workload patterns [3]. Practical case studies have evaluated how AWS's default load balancing algorithms, like round robin and least outstanding requests, impact system behavior during high traffic periods [4]. However, there is limited work assessing how AWS's out-of-the-box autoscaling and load balancing perform in minimal, cost-effective environments using small instance types. This project addresses that gap by empirically testing these default strategies under simulated load to evaluate their effectiveness.

We specifically pose three performance evaluation questions:

1) How does enabling autoscaling impact latency and request success rates compared to a static single server?
2) What is the performance difference between round-robin and least-outstanding-request load balancing strategies under auto scaling?
3) How do these configurations compare in terms of resource utilization (and by extension, potential cost)?

To answer these, we conduct controlled experiments on AWS using a consistent web application workload and measure key metrics such as response time, throughput, error rates, and CPU utilization.

II. Background

A. Technical Background

AWS Auto Scaling automatically adjusts the number of EC2 instances in a group based on demand, while load balancers distribute traffic among multiple instances using various algorithms (commonly round robin or least-connections/least-load methods). The default Application Load Balancer (ALB) routing algorithm is round robin, but it can be configured to use the least outstanding requests algorithm introduced by AWS in late 2019 for more adaptive balancing [5]. K6 is a modern open-source load testing tool that allows developers to script

performance tests and generate high-concurrency traffic. InfluxDB is a time-series database optimized for metrics storage, and AWS CloudWatch provides real-time monitoring and telemetry for AWS resources.

B. Related Work

Autoscaling and load balancing are critical for keeping cloud applications responsive and cost-effective as traffic fluctuates. While cloud providers offer built-in tools to manage these challenges, how well these default tools perform in practice (especially for smaller-scale setups) is not extensively documented. Sanjay et al. [2] analyzed AWS's autoscaling strategies, highlighting that reactive scaling can handle sudden traffic increases but may incur latency spikes if thresholds are not tuned appropriately. This emphasizes the importance of carefully setting autoscaling trigger conditions. In contrast, Garí et al. [3] explored advanced machine learning-based autoscaling, demonstrating that reinforcement learning algorithms can outperform traditional static policies by adapting in real time to changing traffic. However, such sophisticated solutions come with added complexity and are not readily available in standard AWS offerings.

On the practical side, Nguyen et al. [4] deployed a scalable web server using AWS's default load balancing and autoscaling. Their case study showed that even basic configurations (e.g., round-robin load balancing combined with an EC2 Auto Scaling group) can significantly improve system stability and request handling under load. While these studies provide valuable insights into both advanced and default autoscaling techniques, most focus on either theoretical models or large enterprise systems. There is a need for evaluations of AWS's default autoscaling and load balancing in smaller, budget-limited deployments. Our work builds on this literature by testing standard AWS EC2 autoscaling (with no special optimizations) and comparing two built-in load balancer algorithms, thereby providing practical performance data for developers operating within tight resource constraints.

III. Methodology

A. Experimental Setup

We deployed a simple single-page web application (built with Node.js/Express and React for the frontend) on AWS EC2 instances. The application was intentionally minimal (serving a static page) to ensure that the EC2 instance performance (CPU, etc.) is the primary factor in observed results rather than application complexity. All experiments were conducted using **t2.micro** EC2 instances (1 vCPU, 1 GiB memory) running Ubuntu 18.04, which represent the lowest-tier compute option. Using small instances and minimal application logic allows us to evaluate performance impacts of autoscaling policies in a low-cost scenario. All AWS resources were provisioned in the Canada (Central) region to minimize variability in latency.

We evaluated three deployment configurations:

1) Static Single-Server Setup: A single EC2 instance with **no** autoscaling and no external load balancer. This instance handles all incoming traffic alone. This scenario establishes a baseline for performance without any scalability mechanisms.

2) Autoscaling with Round-Robin Load Balancer: An AWS Application Load Balancer (ALB) distributes requests between two EC2 instances in an Auto Scaling group. The autoscaling policy is the default reactive policy (scale out/in based on CPU utilization threshold), and the ALB uses round-robin routing to alternate requests evenly between the two servers. This represents a typical default scaling setup.

3) Autoscaling with Least Outstanding Requests Load Balancer: Similar to configuration 2, we use two EC2 instances with the same autoscaling policy, but the ALB is configured to use the least outstanding requests algorithm. In this strategy, the load balancer routes each new request to the instance with the fewest active (ongoing) requests, aiming to keep the load balanced by work in progress. This strategy is expected to perform better if request handling times vary, as it avoids sending new requests to a busy server [5].

All tests were orchestrated from a separate client machine to avoid resource contention. We utilized a virtual machine on Cybera's Rapid Access Cloud (RAC) platform (accessed via VPN) as the load generator host. This VM ran **K6** to generate load and also hosted an **InfluxDB** container (via Docker) to collect metrics. Using a separate client ensured that the act of load generation did not consume resources on the EC2 instances under test. The InfluxDB container was started with the default configuration (exposed on port 8086) to gather test metrics in real-time. Key environment variables for K6's InfluxDB output (bucket, organization, authentication token, etc.) were configured so that K6 would push metrics to the database after each test iteration.

Each test scenario (Static, Round Robin, LOR) was executed as follows: we ran a K6 test script for **3 minutes 30 seconds** with a staged increase of virtual users (VUs) to simulate a traffic spike. Specifically, the load profile ramped from 1 up to 1000 concurrent virtual users over ~2 minutes (warm-up and ramp-up stages), sustained the peak of 1000 VUs briefly, and then ramped back down (cool-down stage). This yielded a total of roughly 75,000+ requests sent in each test. During the test, K6 recorded metrics such as response time for each request and whether any requests failed. In addition, AWS CloudWatch was used to monitor server-side metrics (CPU utilization per instance, ALB metrics, etc.) throughout the tests. We ensured that before each test the system was in a steady state (e.g., the Auto Scaling group had scaled in to the baseline of one instance in scenarios 2 and 3) to start each run under similar conditions.

B. Data Collection

We collected two categories of metrics for analysis:

1) Client-side: Performance metrics from K6 (ingested into InfluxDB), including request latency distribution (average, 95th percentile), throughput (requests per second), and success/failure counts

2) Server-side: Metrics from CloudWatch, including EC2 instance CPU utilization and the ALB's statistics (target response time and request count).

The K6/InfluxDB metrics allow detailed time-series analysis of how the system responded throughout the load test, while CloudWatch provides insight into how AWS scaling responded (e.g., when a second instance launched) and how evenly the load was balanced.

C. Summary

Our methodology combines empirical performance testing with monitoring at both the client and server side. By comparing the static vs. auto scaled scenarios, and round-robin vs. least-outstanding-request, we can directly observe the benefits and any drawbacks of each strategy under identical workload conditions. All tests were repeated twice to ensure consistency, and results were averaged when differences were negligible (variance was low).

IV. Results

A. Overall Performance Metrics

After running the load tests, we first examine aggregate performance metrics from each scenario. Table 1 summarizes the key results.

All three configurations handled ~75k+ requests over the 3.5-minute test with nearly perfect success rates. The static single server achieved a slightly lower average latency (54.5 ms) than the autoscaling cases with load balancing (which were ~55–60 ms), and it handled the load without any failures. However, the static setup's **tail latency** (95th percentile 84.0 ms) was a bit higher than that of the round-robin auto scaled setup (69.6 ms), indicating more variability under peak load when all traffic hits one small instance. The **auto scaled Round Robin** configuration had very similar average latency (~55.6 ms) to the static case, and likewise sustained 100% success rate, but it kept the high-percentile latency lower (95th percentile ~69.6 ms). This suggests that splitting traffic between two instances helped avoid the higher spikes in response time. The **auto scaled LOR** configuration showed a slightly higher average latency (59.7 ms) and a higher 95th percentile (92.0 ms). It was the only scenario that registered any request failures (0.02% failure rate, meaning only 20 out of ~75k requests failed). While 99.98% success is still extremely high, the tiny fraction of failures hints at moments of overload or brief imbalance with the LOR strategy. Throughput was essentially the same (~360 requests/sec on average) for all scenarios, as constrained by the client load generation.

In practical terms, enabling autoscaling (adding a second server under load) did not significantly reduce the average response time in our test (because even the single micro instance was able to handle 1000 concurrent lightweight requests fairly well), but it did improve the **stability** of response times (seen in the lower 95th percentile) and provided headroom such that no requests had to wait or drop. The least outstanding requests algorithm, while conceptually more adaptive, introduced a bit of overhead in our small-scale test, leading to marginally higher latency and a few timeouts. We investigate these behaviors in more detail with system metrics below.

B. Auto Scaling Behaviour and Resource Utilization (CloudWatch Insights)

To understand the above results, we inspected AWS CloudWatch metrics during the tests. In the autoscaling scenarios, a second EC2 instance was launched when the load balancer detected the high traffic (triggered by CPU utilization alarms as per the default policy). Figure 1 illustrates how CPU usage evolved on the EC2 instances across our sequential tests:

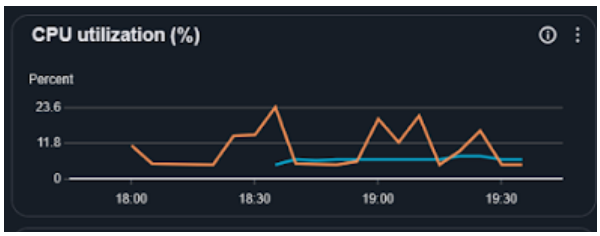

Figure 1. CPU Utilization (%) of the EC2 Instances

The **orange** line is the primary instance and the **blue** line is the second instance (launched by autoscaling). We ran the tests in this order: (i) static single-server, (ii) autoscaling with round robin, (iii) autoscaling with LOR. In the first spike (around 18:40 on the timeline), only the orange line is present, peaking at about 23.6% CPU during the single-server test. For the subsequent tests, two instances were active: during the round-robin test (middle period on the graph), both instances shared the load fairly evenly (each showing moderate CPU usage, with the orange line around 10–15% and the blue line similarly low, these appear as relatively flat lines around 10% on the combined graph). In the final spike (around 19:15), corresponding to the LOR test, we observe a **small jump in blue instance CPU usage** in Figure 2, the blue line rises slightly (around 7% at peak), while the orange instance's CPU (orange line) also remains modest. Overall, CPU utilization on both servers remained low (well under 25%) even at peak load, which is expected since the workload was not CPU-intensive. The static case pushed the single CPU to about 23.6% utilization, whereas with two instances, each CPU was under ~15%. This indicates that the

TABLE I. Summary of Performance Results under 1000 Concurrent Users

Configurations	Avg. Latency	95th Percentile Latency	Failed Request Rate	Total Requests	Success Rate
Static (1 EC2, No Auto Scaling)	54.5 ms	84.0 ms	0%	76,091	100%
Auto Scaling + Round Robin ALB	55.6 ms	69.6 ms	0%	76,028	100%
Auto Scaling + Least Outstanding ALB	59.7 ms	92.0 ms	0.02%	75,715	99.98%

autoscaling effectively split the traffic load. Figure 1 confirms that our tests were executed in the intended sequence and that the autoscaling triggered correctly for the two dynamic scenarios. The second instance (blue) was essentially idle in the first test (not yet launched), and became active in tests two and three. Notably, the CPU usage on the second instance was **very stable** (flat line) during round-robin balancing, and only in the LOR scenario did it show a distinct spike, which corresponds to that algorithm's behavior of occasionally channeling more load to the previously underutilized instance.

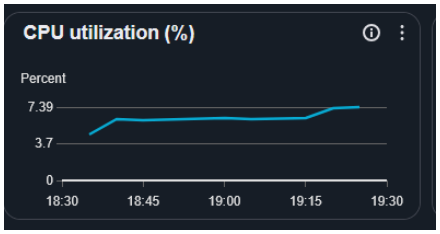

Figure 2. CPU Utilization (%) of Second EC2 Instance

We also examined the load balancer's own performance metrics. AWS CloudWatch reported the **Target Response Time** of the ALB (time the load balancer itself took to forward requests). In the round-robin test, the ALB's response time spiked to roughly **7 ms**, whereas in the LOR test it spiked to about **13 ms**. These values are low in absolute terms, but the LOR algorithm incurred nearly double the overhead on the load balancer. This makes sense because **least outstanding requests** requires more frequent evaluation of back-end queue lengths before dispatching each request, adding a small processing delay. The load balancer's **Request Count** metric confirmed that both autoscaling configurations successfully handled the surge of ~76k requests during the peak with zero dropped requests. In other words, the ALB in both cases was able to route all incoming traffic to the instances (and the tiny fraction of failures in the LOR case happened at the instance level, not due to the ALB dropping any requests).

C. Detailed Time-Series Analysis (InfluxDB Metrics)

While the aggregate metrics show end-to-end outcomes, the time-series data gives more insight into system dynamics under load. We exported and visualized the K6/InfluxDB metrics for each scenario. Each figure below plots key performance indicators over the 210-second test duration for one scenario. The metrics include:

1) ***vus****:* number of active virtual users
2) ***http_req_duration****:* request latency
3) ***http_reqs****:* request throughput
4) ***iteration_duration****:* the time for each VU to complete one test script iteration
5) ***http_req_failed****:* the count of failed requests

All metrics are plotted against the test time. We discuss two representative scenarios for brevity.

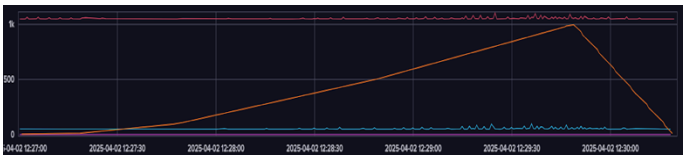

Figure 3. Performance Metrics - Static Single-Server Scenario

In this scenario, one EC2 instance handled all users. The **orange** line (VUs) steadily ramps up to 1000 concurrent users (reaching the peak at around t≈120s) and then ramps down, reflecting our load profile. The **blue** line (HTTP request duration) remains relatively flat between **50–60 ms** throughout the test, indicating consistent latency even as the load increases. We see only very minor bumps in latency during the peak, but no significant degradation. The **light pink** line (iteration duration per VU) is essentially constant (~1.05s), which means each virtual user was completing its loop (one request per iteration) at a steady pace regardless of overall load – a sign that the system did not slow down the user experience as more users joined. The **red** line (HTTP requests per unit time) climbs quickly with the user ramp-up, leveling out around ~360 requests/sec at peak, and then tapering off; this matches the expectation since more users generate more total requests until all 1000 are active. The purple line (HTTP request failures) stays at zero across the entire timeline – indeed, in the static case we recorded no failed requests. Overall, Figure 3 shows that the single micro instance handled the gradually increasing load **smoothly**: latency was stable and throughput increased linearly with users. There were no signs of resource exhaustion in this range (consistent with the CPU usage being only ~23% at max).

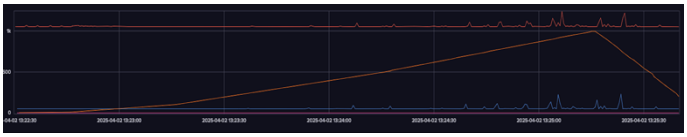

Figure 4. Performance Metrics - Auto Scaling with Least Outstanding Request

This scenario had two instances and the LOR algorithm balancing requests. The **orange** VU line has the same ramp pattern to 1000. The **blue** line here is the HTTP request duration (latency), and we notice it exhibits more frequent spikes compared to the static case. Most of the time, latency stays well below 100 ms (usually ~50–60 ms), but there are some sharp upward spikes (up to ~200–500 ms) visible around the peak load period. These correspond to occasional moments where a request took longer, likely when the LOR balancer shifted load and one instance briefly had to handle a disproportionately large share or cold start. The **red** line (throughput) is again high and relatively steady, but interestingly we see a slight dip or perturbation near the peak, coinciding with the latency spikes. This implies a brief slowdown in processing some requests (hence a momentary drop in throughput), which recovered quickly. The **light pink** line (iteration duration) remains very close to flat, meaning each user's overall cycle time did not change much; even if one request was slow, it did not dramatically affect the next iteration timing for users.

Importantly, the **purple** line (failed requests) is mostly zero but shows a couple of tiny blips (near t≈130s and 140s). These blips correspond to the 20 failed requests (0.02% of total) we noted earlier. They occurred at the height of the load and are likely due to request timeouts when the system experienced those latency spikes. After the brief failures, the system continued processing all remaining requests successfully as the load decreased.

Comparing the static and LOR scenarios: the static single-server system maintained very uniform performance until the end, whereas the LOR autoscaling system delivered **higher overall capacity** (since it had two servers) but introduced a bit of **volatility** in response times and a handful of errors under extreme load. The round-robin autoscaling scenario, seen below in Figure 5, actually combined the best of both: like the LOR case it had two servers sharing the work (so no resource saturation), but its latency plot was nearly as smooth as the static case and it experienced zero failures. In essence, the round-robin strategy produced the most **predictable** performance curve, while the LOR strategy produced a slightly less consistent curve with small hiccups, albeit still handling the load effectively.

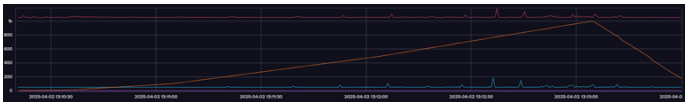

Figure 5. Performance Metrics - Auto Scaling with Round Robin

D. Additional Observations

From CloudWatch, we also looked at metrics like network traffic and CPU credit usage on the t2.micro instances. All scenarios stayed within the free CPU credit budget of the t2 instances, no CPU throttling occurred. This is evident from the steadily increasing **CPU credit balance** metric during the tests, meaning the instances continued to accrue credits (since average CPU utilization was low) even while servicing the requests. This suggests that our test, while intensive in number of requests, did not fully tax the CPU or network capacity of a micro instance due to the lightweight nature of each request. In a real-world scenario with heavier per-request processing, we might have seen higher CPU usage and potentially more performance differentiation between one vs. two instances.

We did not explicitly measure cost in this short test, but AWS's cost and usage report for the period of testing indicated a negligible cost (on the order of only $10 for all resources used, since EC2 micro instances and low hourly usage incur minimal charges). A rigorous cost analysis would require longer runs or sustained loads, but our data suggests that using two micro instances for a short duration is still very inexpensive. Autoscaling ensures that the second instance runs only when needed (minutes in our case), so the cost overhead for improved performance was minimal.

V. Conclusion and Future Work

A. Conclusion

Our experiments highlight the significant advantages of enabling auto scaling for web applications in the cloud. Even with a lightweight workload on small instances, autoscaling ensured low response times and high success rates under a load of 1000 concurrent users. Among the autoscaling configurations, the **round-robin load balancer** provided the most consistent and dependable performance. It maintained stable latency and zero errors, effectively demonstrating that a simple two-instance deployment can nearly double the handling capacity with no degradation in per-request performance. The least outstanding request strategy, while theoretically more adaptive, showed minor performance issues in our tests—slightly higher latency and a few failed requests under peak stress—indicating that the added complexity may not pay off for uniformly distributed workloads. For workloads where request processing times vary widely, however, LOR could still be beneficial to prevent any one instance from becoming a bottleneck (as also suggested in other studies [3] [5]).

B. Future Work

Our study opens several avenues for further investigation. It would be useful to compare **horizontal scaling vs. vertical scaling** (i.e., adding more small instances versus using a single more powerful instance) for reasons such as:

1) *Handle the same load*
2) *Evaluate cost-effectiveness and performance differences*

Another beneficial comparison is exploring **predictive or proactive** scaling methods (potentially using machine learning or schedule-based scaling) to show whether reacting before a surge (as opposed to the reactive scaling we used) yields better performance during sudden spikes.

A further extension is a detailed cost analysis to measure:

1) *AWS charges incurred by each strategy over longer periods*
2) *Different traffic patterns to quantify the cost-performance trade-off* (for example, whether running two instances continuously for reliability is justified by the performance gain, under various pricing models)

Additionally, testing on a larger scale (e.g., 5000+ users or using larger instance types) would provide data on how these strategies scale and at what point the single-instance approach would definitively fail.

Finally, investigating other load balancing algorithms (such as weighted round robin, IP-hash, or external solutions) and autoscaling policies (like AWS's target tracking scaling or step scaling with different thresholds) could yield a more comprehensive understanding of the landscape of options for cloud auto scaling setups. By pursuing these future directions, we can develop a more generalized set of best practices for optimizing cloud application performance under varying conditions.

REFERENCES

[1] Amazon Web Services, "What is Amazon EC2 Auto Scaling?," 2025. [Online]. Available: https://docs.aws.amazon.com/autoscaling/ec2/userguide/what-is-amazon-ec2-auto-scaling.html. [Accessed 20 April 2025].

[2] V. M. Sanjay, A. I. Kumar and S. A. Kyalkond, "Analysis of AWS Auto Scaling Strategy in Cloud Computing," International Journal of Innovative Research in Science, Engineering and Technology (IJIRSET), vol. 10, no. 12, pp. 15067-15072, 2021.

[3] Y. Garí, D. A. Monge, E. Pacini, C. Mateos and C. G. Garino, "Reinforcement Learning-based Application Autoscaling in the Cloud: A Survey," Engineering Applications of Artificial Intelligence, vol. 102, no. 2, pp. 1-19, 2021.

[4] H. S. Nguyen and K. C. Nguyen, "Load Balancing in Auto Scaling Enabled Cloud Environments," International Journal on Cloud Computing Services and Architecture, vol. 7, no. 5, pp. 15-22, 2017.

[5] Amazon Web Services, "Application Load Balancer now supports Least Outstanding Requests algorithm for load balancing requests," 25 November 2019. [Online]. Available: https://aws.amazon.com/about-aws/whats-new/2019/11/application-load-balancer-now-supports-least-outstanding-requests-algorithm-for-load-balancing-requests/. [Accessed 22 April 2025].