

LBS Local Systems Lab: A Deterministic Simulation Platform for Comparing Load-Balancing Algorithms

1st Raksha Kardak
Computer Science and Engineering
Priyadarshini College of Engineering
Nagpur, India

2nd Shital Tiwari
Computer Science and Engineering
Priyadarshini College of Engineering
Nagpur, India

3rd Nisarga Hadke
Computer Science and Engineering
Priyadarshini College of Engineering
Nagpur, India

4th Shruti Satpute
Computer Science and Engineering
Priyadarshini College of Engineering
Nagpur, India

5th Manjiri Parate
Computer Science and Engineering
Priyadarshini College of Engineering
Nagpur, India

6th Riya Shahu
Computer Science and Engineering
Priyadarshini College of Engineering
Nagpur, India

Abstract - Cloud and distributed services receive client traffic that is unpredictable and uneven, so a poorly chosen request-routing policy can overload some servers while others sit idle, inflating response times and reducing throughput. This work presents the LBS Local Systems Lab, a loopback-only simulation platform that makes every routing decision, queue, response, and health transition visible, and that benchmarks six load-balancing policies - Round Robin, Weighted Round Robin, Least Connections, Least Response Time, IP Hash, and Random - on identical, seed-controlled workloads. A deterministic virtual mode and a measured HTTP mode share one interface, and every run is stored in a local database for replay and export. Algorithms are compared on median and 95th-percentile response time, queue wait, throughput, availability, load imbalance, and client affinity. In a steady-traffic demonstration, Round Robin served 48 of 48 requests at a 223 ms median latency with a perfectly even 16/16/16 split. Under a burst scenario, Least Connections gave the lowest 95th-percentile latency (1.83 s) and Round Robin followed closely (1.88 s), whereas Weighted Round Robin and IP Hash reached about 3.8–3.9 s. The results are relative observations for one scenario, seed, and machine, but they show concretely why no single algorithm wins in every situation.

Keywords - load balancing, request routing, round robin, weighted round robin, least connections, least response time, IP hash, random selection, distributed systems, deterministic simulation

I. INTRODUCTION

The growth of cloud computing has been fueled by an ever-rising appetite for on-demand infrastructure, elastic applications, remote storage, and internet-accessible services.

Compute, storage, platform, and software resources are now provisioned over the network in whatever quantity a given workload requires. Virtualization underpins much of this flexibility: it lets a single physical machine be partitioned into many independent virtual instances, which in turn makes far better use of the underlying hardware.

As adoption of internet-facing applications continues to climb, individual servers are routinely hit with bursts of simultaneous requests. This traffic tends to be irregular, hard to forecast, and rarely balanced across the available backend capacity. When a disproportionate share of requests lands on one server while others sit comparatively idle, the consequences include slower responses, diminished throughput, strained resource usage, weaker scalability, and, at the extreme, outages. Distributing incoming work intelligently across a server pool has therefore become a core requirement for any cloud or distributed system.

Load balancing is the mechanism that spreads incoming traffic, computation, or user requests across a set of servers or other computing resources. Done well, it keeps any single node from becoming a bottleneck and ensures the collective capacity of the pool is actually put to use. The payoffs include better overall throughput, lower latency, and support for a larger population of concurrent users. Because traffic can also be redirected around a failed or overloaded node, load balancing additionally strengthens a system's availability, fault tolerance, and long-term scalability.

Researchers and practitioners have proposed a wide spectrum of load-balancing strategies, spanning static, dynamic, hybrid, and optimization-driven designs. Static schemes route work according to rules fixed in advance and tend to work best when the environment changes slowly. Dynamic schemes instead look at the servers' present condition — active connections, current load, measured response time — and adjust their decisions accordingly, which lets them react to shifting traffic. Meta-

heuristic and AI-based methods have also been explored for this problem, though they typically add computational and engineering overhead that a simpler rule-based scheme avoids.

Motivated by these trade-offs, this project builds the LBS Local Systems Lab, a teaching and benchmarking testbed in which one workload can be routed by several load-distribution policies and the outcome inspected side by side. Simulated client requests pass through a balancer and are assigned to a pool of local servers by one of six algorithms — Round Robin, Weighted Round Robin, Least Connections, Least Response Time, IP Hash, and Random — each of which uses a different signal to choose a destination, making it possible to observe how their behavior diverges as load conditions change.

For every run the lab records median and tail response time, queue wait, throughput, availability, workload spread, and client affinity, and it draws the routing path, server queues, and health state while the run is in progress. Rather than reaching for a heavier AI-driven controller, the project runs a direct, reproducible comparison of algorithms already in common use, and it states plainly what the measurements do and do not show, with the goal of helping practitioners and students pick a suitable strategy for their own workloads.

The rest of the paper proceeds as follows. Section II surveys prior work on load-balancing methods. Section III lays out the system's design and methodology. Section IV presents and discusses the simulation results. Section V closes with conclusions and directions for future work.

II. RELATED WORK

Load balancing in cloud and distributed environments has been studied extensively, since how work is spread across servers has a direct bearing on performance, scalability, resource efficiency, and availability. The literature covers static, dynamic, hybrid, heuristic, optimization-based, and nature-inspired strategies, each differing in how it maps incoming tasks or requests onto physical machines, virtual machines, or other backend resources.

Classic algorithms remain popular largely because they are easy to reason about and implement. Round Robin cycles through the available servers in sequence and works well when those servers are roughly equal in capacity, but it can perform poorly when either the servers or the incoming requests are heterogeneous. Weighted Round Robin corrects for unequal server capacity by assigning each server a weight, so stronger machines are given a larger share of the traffic. Least Connections instead looks at how many active connections each server currently holds and sends the next request to whichever server has the fewest, which helps avoid piling more work onto an already-busy node. Least Response Time goes a step further by tracking how quickly each server has been responding and steering new requests toward the fastest one. IP Hash derives a hash from the client's IP address to pick a server, which has the side effect of routing a given client to the same server consistently — useful for maintaining session state. Random selection assigns each request to a uniformly chosen server; it keeps no state at all and therefore serves as a simple baseline against which smarter policies can be judged.

A number of other techniques — including Min-Min, Max-Min, Opportunistic Load Balancing, Throttled, and Join Idle Queue — have also been studied. While these can improve

throughput and resource use in some settings, researchers have reported drawbacks such as task starvation, sensitivity to workload characteristics, added complexity, and inconsistent performance across different deployment conditions.

Load-balancing methods are often grouped by how much they rely on the system's real-time state. Static methods work from fixed, pre-configured rules and suit environments that are stable and fairly uniform. Dynamic methods instead observe live conditions — server load, active connections, measured response times — and adapt their placement decisions as those conditions change. Hybrid designs blend the two, aiming for the responsiveness of dynamic methods without giving up all of the simplicity of static ones.

Centralized versus distributed control is another axis explored in prior work. A centralized controller offers a single, easy-to-manage point of decision-making, but it can become a scalability bottleneck as the coordination burden grows. On the more advanced end of the spectrum, nature-inspired and optimization-based methods — Ant Colony Optimization, Particle Swarm Optimization, Genetic Algorithms, Cuckoo Search, Artificial Bee Colony, and Lion Optimization among them — have been applied to harder load-balancing problems. These can find strong or near-optimal solutions in complex settings, but usually at the cost of extra computation, slower convergence, and more implementation effort.

Evaluating any of these strategies typically comes down to a shared set of metrics. Response time captures the delay between a request being submitted and its answer arriving; throughput counts how many requests a system can successfully complete in a fixed window. Scalability describes whether performance holds up as load or user count grows, and resource utilization looks at how efficiently CPU, memory, storage, and network capacity are actually used. Other metrics that show up regularly in the literature include makespan, coordination overhead, fault tolerance, availability, SLA adherence, and the number of concurrent requests a system can sustain.

Across this body of work, a consistent theme emerges: no single algorithm dominates under every workload. Which strategy performs best depends on how uniform the servers are, what the incoming requests look like, how intense the traffic is, how much spare capacity exists, and the system's current state. AI-based and meta-heuristic methods continue to be explored, but a grounded comparison of the algorithms already used in practice remains valuable for understanding their real-world behavior. This project follows that thread, implementing and comparing Round Robin, Weighted Round Robin, Least Connections, Least Response Time, IP Hash, and Random within one simulation environment so their relative strengths and weaknesses can be examined under a common set of traffic conditions.

III. METHODOLOGY

The system follows a simulation-driven, comparative approach: requests are generated or captured, routed to backend servers under a chosen algorithm, and then scored against a fixed set of performance metrics.

A. Overall Methodology

The pipeline moves through the following stages: Request Generation/Capture → Load Balancer → Algorithm Selection → Server Selection → Request Distribution → Performance

Monitoring → Comparative Analysis. Structuring the system this way makes it possible to observe how each algorithm behaves as traffic conditions are varied. Fig. 1 depicts the overall architecture, and Fig. 2 shows the corresponding process flow.

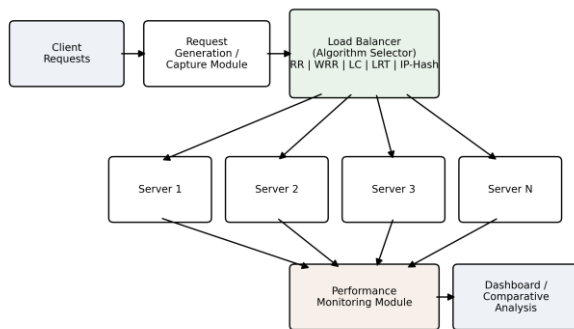


Fig. 1. Overall system architecture of the proposed load balancer simulation.

B. System Initialization

A configurable pool of local servers is set up at the start of each run. In the demonstrations reported here the pool contains three workers named Atlas, Beacon, and Comet, each with a capacity of two concurrent requests. Every server carries attributes such as an ID, rated capacity, active and queued request counts, response time, simulated CPU load, and a health status that can be toggled with Fail and Recover controls. The simulator keeps this initial state on hand as the baseline for the run.

C. Request Generation and Capture

Requests entering the system are generated from a seeded workload definition — a scenario (for example steady or burst traffic), a seed string, a request rate, and a duration — so that the same inputs always reproduce the same arrivals. Each request carries metadata — a request ID, arrival timestamp, client key, service duration, the algorithm in effect, and the server it is ultimately assigned to — before being handed off to the load balancer.

D. Load-Balancing Algorithm Selection

Users choose from among the following six algorithms to control how incoming requests are distributed.

1) Round Robin: Requests cycle through the available servers in order — Request 1 to Server 1, Request 2 to Server 2, Request 3 to Server 3, Request 4 back to Server 1, and so on.

2) Weighted Round Robin: Servers are given weights that reflect their relative capacity, so a server with a higher weight is handed a proportionally larger slice of the incoming traffic.

3) Least Connections: The simulator checks how many active connections each server is currently handling and sends the new request to whichever one has the fewest.

4) Least Response Time: Server response times are tracked continuously, and each new request goes to whichever server is currently answering fastest.

5) IP Hash: The client's IP address is hashed to a value that maps deterministically to a specific server, which keeps repeat requests from the same client landing on the same backend.

6) Random: Each request is sent to a server chosen at random from the eligible set, giving a stateless baseline that needs no load information.

E. Server Selection and Request Routing

Before the algorithm runs, the balancer builds a list of eligible servers by filtering out any that are unavailable. The algorithm then picks among the eligible servers, and the lab's explanation panel reports both the eligible set and the reason for the choice. If a failure forces a request to be sent elsewhere, the retry is counted and shown in the run summary.

F. Performance Monitoring

Throughout a run, the lab records queue wait (work start minus server queue arrival), service time (completion minus work start), and response time (completion minus original client arrival) for every request. From these it derives median and 95th-percentile response time, throughput (completed requests per run second), availability (successful client requests over generated requests), raw imbalance and capacity imbalance (variation of completed request counts, the latter normalized by server capacity), and client affinity (the share of requests reaching each client's most-used server). These measurements are saved to a local database for later analysis.

G. Comparative Performance Analysis

Every algorithm is subjected to an identical workload so the resulting performance data can be compared on equal footing. In the Compare view, one workload manifest is created once and replayed sequentially for each selected algorithm, so arrival times, request sizes, clients, and failure events remain identical; the first 800 ms of each replay is excluded from the comparison metrics. The lab supports two timing sources: a virtual mode driven by a deterministic clock, and an HTTP mode that measures real loopback requests through a gateway and worker processes. All results reported in this paper come from the virtual mode. The overall flow is summarized in Fig. 2.

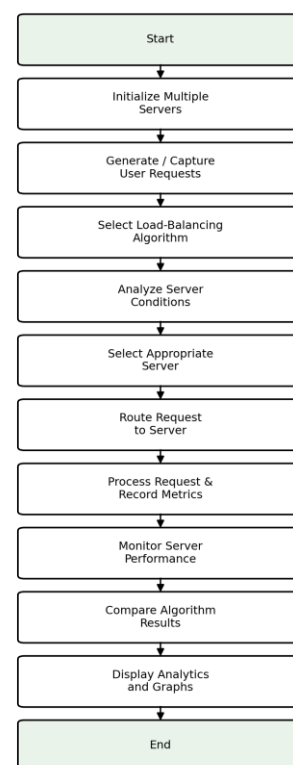


Fig. 2. Overall methodology flow of the proposed simulation system.

IV. RESULTS AND DISCUSSION

All results below are measured outputs of the LBS Local Systems Lab, captured in its deterministic virtual mode on a single machine with the lab bound to the loopback interface (127.0.0.1). Two experiments were run: a live steady-traffic demonstration using Round Robin, and a six-way comparison under burst traffic. Because the lab is designed to explain trade-offs rather than to promise production performance, the numbers are relative observations for one scenario, seed, and machine and should not be read as absolute benchmarks.

A. Experimental Setup

The lab provides a live simulation view, a comparison view, a run history, and an algorithm guide that summarizes each routing policy with a worked example, together with when it is useful and what to watch for (Fig. 3). The live simulation used Round Robin on steady traffic with seed class-demo, a rate of 6 requests per second, and a duration of 8000 ms against a pool of three workers (Atlas, Beacon, and Comet) with a capacity of two concurrent requests each. The comparison used the burst scenario with seed comparison-class and all six algorithms enabled, as configured in the panel shown in Fig. 4. A single manifest was created once and replayed sequentially, so every algorithm saw identical arrivals.

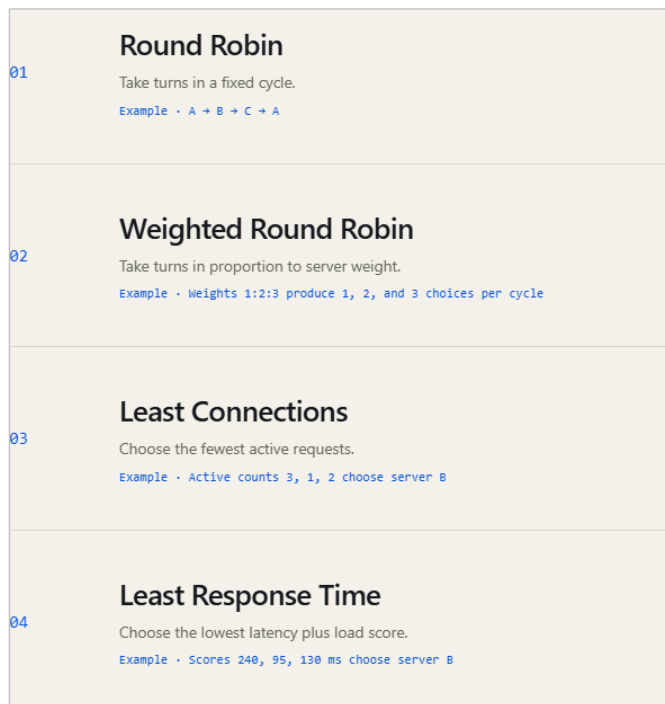


Fig. 3. Algorithm guide of the LBS Local Systems Lab: name, description, and worked example for each routing policy (first four of the six policies shown).

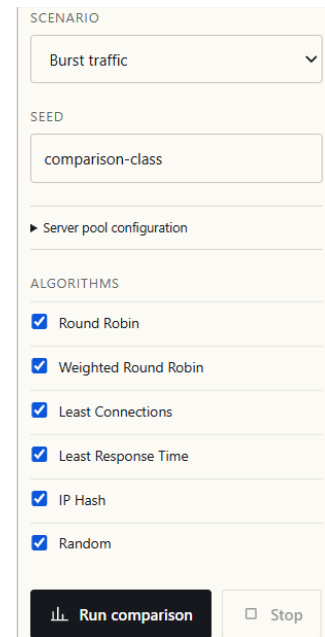


Fig. 4. Compare view configuration: burst scenario, seed comparison-class, and all six algorithms selected.

B. Live Simulation under Steady Traffic

Fig. 5 shows the live dashboard during the run. Each arriving request becomes a visible routing decision in the topology, and an explanation panel beside it names the chosen server, the eligible set, and the reason for the choice; for request #41 it reports that Beacon is the next server in the cycle. The completed run finished all 48 generated requests with 100% availability and no retries. Median latency was 223 ms, the 95th-percentile response time was 256 ms, and the 95th-percentile queue wait was 0 ms, at an effective throughput of 5.3 requests/s. Peak load was two active requests and none queued.

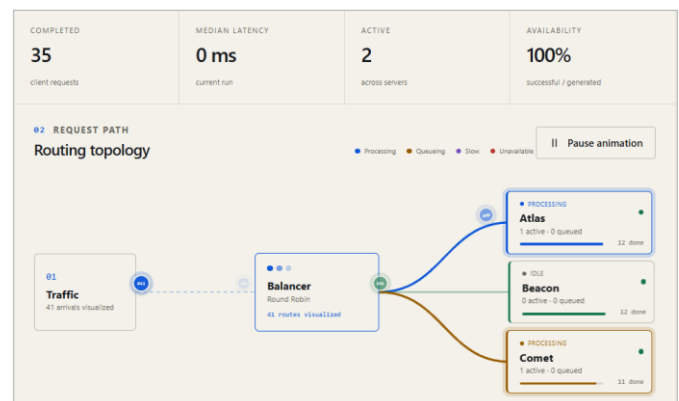


Fig. 5. Live simulation dashboard captured mid-run (35 of 48 requests completed) during the steady-traffic Round Robin run, showing headline metrics and the routing topology with per-server state.

Fig. 6 plots the cumulative requests served by each worker. The three curves climb together and finish at 16 requests each, a perfectly even 16/16/16 split with a coefficient of variation of 0, as expected for Round Robin on identical servers. Average response times per server were 219 ms (Atlas), 230 ms (Beacon), and 220 ms (Comet).

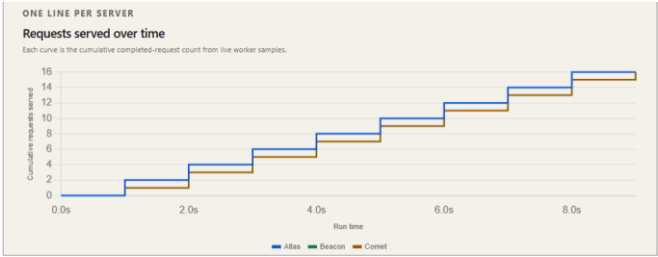


Fig. 6. Cumulative requests served over time by each server in the steady-traffic Round Robin run.

Fig. 7 shows the supporting time series. Latency peaks at about 240 ms in the first second and then settles just above 220 ms; completions per sample plateau at 6 between roughly 2 s and 8 s, matching the 6 requests/s arrival rate; and active work holds at 2 while queued and failed work stay at 0, so the pool never saturated.



Fig. 7. Supporting time series for the steady-traffic run: requests being processed per server, latency over time, completions per sample, completed distribution, and active, queued, and failed work.

C. Algorithm Comparison under Burst Traffic

Fig. 8 compares the 95th-percentile response time of the six algorithms on the same burst manifest. All six completed every request, with a 0% error rate and 100% availability, so the differences appear in latency, queueing, throughput, and distribution rather than in failures. The burst phase makes queue growth and recovery more important than average request count, so the tail of the latency distribution is the most informative comparison.

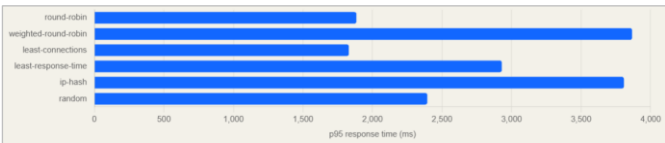


Fig. 8. 95th-percentile response time of the six algorithms on the same burst-traffic manifest.

Table I and Fig. 9 give the main metrics. Least Connections achieved the lowest 95th-percentile response time (1.83 s) and the lowest 95th-percentile queue wait (1.58 s), with Round Robin close behind (1.88 s and 1.64 s). Round Robin, Least Connections, and Random all reached the highest throughput of 16.5 requests/s. Least Response Time was intermediate, at 2.93 s p95 and 14.7 requests/s. Weighted Round Robin and IP Hash performed worst in the tail, at 3.87 s and 3.81 s p95, which is 1983 ms and 1924 ms above Round Robin, and both fell to 13.2 requests/s. Random, despite using no state at all, landed at 2.39 s p95, 510 ms above Round Robin. The lab deliberately computes no composite winner; the tail latency, availability, distribution, and scenario explanation are meant to be read together.

ALGORITHM	MEDIAN	P95	P95 QUEUE	THROUGHPUT
Round Robin The burst phase makes queue growth and recovery more important than average request count.	947 ms	1.88 s	1.64 s	16.5 r/s
Weighted Round Robin The burst phase makes queue growth and recovery more important than average request count. Its p95 was 1983 ms higher than round-robin in this run.	1.09 s	3.87 s	3.62 s	13.2 r/s
Least Connections The burst phase makes queue growth and recovery more important than average request count. Its p95 was 54 ms lower than round-robin in this run.	976 ms	1.83 s	1.58 s	16.5 r/s
Least Response Time The burst phase makes queue growth and recovery more important than average request count. Its p95 was 1046 ms higher than round-robin in this run.	1.03 s	2.93 s	2.69 s	14.7 r/s
IP Hash The burst phase makes queue growth and recovery more important than average request count. Its p95 was 1924 ms higher than round-robin in this run.	2.23 s	3.81 s	3.54 s	13.2 r/s
Random The burst phase makes queue growth and recovery more important than average request count. Its p95 was 510 ms higher than round-robin in this run.	1.05 s	2.39 s	2.16 s	16.5 r/s

Fig. 9. Left columns of the comparison table produced by the lab for the burst-traffic run (algorithm, median, p95, p95 queue wait, and throughput). The first 800 ms of each replay is excluded.

TABLE I
SUMMARY OF MEASURED RESULTS UNDER BURST TRAFFIC (SEED COMPARISON-CLASS)

Algorithm	Median (s)	P95 (s)	Thr. (req/s)	Raw imb.	Affinity (%)
Round Robin	0.95	1.88	16.5	0	34.8
Weighted RR	1.09	3.87	13.2	0.399	50.4
Least Connections	0.98	1.83	16.5	0.018	54.1
Least Resp. Time	1.03	2.93	14.7	0.221	43.7
IP Hash	2.23	3.81	13.2	0.707	100
Random	1.05	2.39	16.5	0.113	41.5

D. Workload Distribution and Client Affinity

Distribution metrics tell a complementary story. Round Robin produced an exactly even split (raw and capacity imbalance of 0), and Least Connections was nearly as even (0.018), because it always sends work to the server with the fewest active requests. Random (0.113) and Least Response Time (0.221) showed moderate variation, while Weighted Round Robin (0.399) and IP Hash (0.707) were the most uneven. In these runs raw and capacity imbalance coincide, which indicates servers of equal capacity. IP Hash's imbalance is the price of its defining property: it sent 100% of each client's requests to that client's most-used server, compared with 34.8% for Round Robin and between 41.5% and 54.1% for the remaining policies.

E. Reproducibility and Run History

Every run and comparison is saved to the local database. The run history (Fig. 10) lists each stored run with its algorithm, scenario, seed, mode, median latency, and availability, and lets a run be reopened, rerun, deleted, or exported as JSON or CSV. Median values in the saved records (for example 890 ms for Round Robin) differ slightly from those in Table I (947 ms) because the comparison metrics exclude the first 800 ms of each replay.

RUN	SCENARIO	RESULT
random f785df85 - virtual	burst seed comparison-class	993 ms median 100% available
ip-hash f831d30c - virtual	burst seed comparison-class	2.06 s median 100% available
least-response-time 42c5db0c5 - virtual	burst seed comparison-class	1.02 s median 100% available
least-connections 88cde84d - virtual	burst seed comparison-class	904 ms median 100% available
weighted-round-robin 7950be73 - virtual	burst seed comparison-class	902 ms median 100% available
round-robin b09f95c1 - virtual	burst seed comparison-class	890 ms median 100% available

Fig. 10. Run history view listing the stored runs for each algorithm under the burst scenario and seed comparison-class.

F. Discussion

Taken together, the results show that the best algorithm depends on what the workload does, and that the simple rule "state-aware beats static" holds only in part. Under burst traffic, Least Connections delivered the best tail latency and a nearly even distribution, since its active-request count is a direct, if imperfect, proxy for the queue growth that matters during a burst. Round Robin was almost as good here because the servers appear to have equal capacity; as the algorithm guide cautions, it ignores request size and server capacity, so its advantage should not be expected to carry over to heterogeneous pools. Least Response Time did not outperform these simpler policies in this run; the guide notes that it needs feedback and sensible cold-start values, which a short burst may give it little opportunity to establish. Weighted Round Robin, which applies fixed weights and does not react to live queues, and IP Hash, which ignores load entirely, were the slowest in the tail, yet IP Hash is the only policy that guarantees client affinity. These

observations line up with the trends surveyed in Section II and reinforce that the right algorithm depends on the priorities of the specific system, whether that is session persistence, tail latency, or simplicity.

Several limits apply. The results come from a single scenario, seed, and machine in virtual mode; the CPU shown in the lab is simulated; and although worker failover is demonstrated, the gateway is still one local process, so the project makes no claim of load-balancer high availability.

V. CONCLUSION AND FUTURE WORK

This paper described the LBS Local Systems Lab, a local simulation platform for comparing six widely used load-balancing strategies — Round Robin, Weighted Round Robin, Least Connections, Least Response Time, IP Hash, and Random — under a shared, reproducible evaluation framework. The lab routes seeded workloads, visualizes each routing decision, monitors queues and server health, stores every run locally, and scores each algorithm on median and tail latency, queue wait, throughput, availability, load imbalance, and client affinity. In the reported experiments, Round Robin served a steady workload evenly at a 223 ms median latency, and under burst traffic Least Connections and Round Robin delivered the lowest tail latency (1.83 s and 1.88 s p95) while Weighted Round Robin and IP Hash degraded most; IP Hash nonetheless earned its keep through full client affinity. These are relative results for one scenario, seed, and machine, and they illustrate that algorithm choice should follow the workload rather than a fixed ranking. Future work will repeat the comparison across more scenarios and seeds, add HTTP-mode measurements on real loopback traffic, extend the server pools to heterogeneous and failure-prone configurations, evaluate fault tolerance beyond a single gateway process, and explore cloud-managed deployments and lightweight AI-assisted strategies.

ACKNOWLEDGMENT

The authors gratefully acknowledge the Department for providing the computing infrastructure, laboratory access, and technical resources that made this project possible. We extend our sincere thanks to our faculty advisor and the department's teaching staff for their guidance, constructive feedback, and encouragement throughout the design, implementation, and evaluation stages of this work. We are also thankful to the College/University administration for fostering an environment that supports student-led research and experimentation in distributed systems and cloud computing. Finally, we appreciate the support of our peers, whose discussions and feedback helped refine the direction of this project, and our families, for their continued encouragement throughout the course of this work.

REFERENCES

- [1] R. Buyya, C. S. Yeo, S. Venugopal, J. Broberg, and I. Brandic, "Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility," *Future Generation Computer Systems*, vol. 25, no. 6, pp. 599–616, 2009.
- [2] N. J. Kansal and I. Chana, "Cloud load balancing techniques: A step towards green computing," *International Journal of Computer Science Issues*, vol. 9, no. 1, pp. 238–246, 2012.
- [3] T. Lu and Z. Chen, "A survey of load balancing algorithms in cloud computing," in *Proc. Int. Conf. Cloud Computing and Big Data*, 2015, pp. 1–6.

- [4] K. Mishra, S. K. Majhi, and B. Sahoo, "A comprehensive study on load balancing techniques in cloud computing," Journal of King Saud University – Computer and Information Sciences, 2021.
- [5] A. Singh, D. Juneja, and M. Malhotra, "Autonomous agent based load balancing algorithm in cloud computing," Procedia Computer Science, vol. 45, pp. 832–841, 2015.
- [6] A. M. Chandrasekaran and M. Chandrasekaran, "A survey on load balancing algorithms in cloud computing environment," Int. J. Adv. Res. Comput. Sci., vol. 8, no. 5, 2017.
- [7] M. Young, The Technical Writer's Handbook. Mill Valley, CA: University Science, 1989.