Proceedings of the International Conference on Industrial Engineering and Operations Management

Publisher: IEOM Society International, USA DOI: 10.46254/NA11.20260131
Published: June 9, 2026

Lightweight Machine Learning-Based Dynamic Load Balancing for Distributed Object Storage Systems

Naman Subedi, Bo Sun, MD Masud Rana and Frank Sun
Department of Computer Science
Lamar University
Beaumont, Texas, USA
nsubedi1@lamar.edu

Helen H. Lou
Dan F. Smith Department of Chemical & Biomolecular Engineering
Center for Data Analytics and Cybersecurity
Lamar University
Beaumont, Texas, USA

Alek Hutson
Center for Midstream Management and Science
Lamar University
Beaumont, Texas, USA

Abstract

Load balancing is critical for achieving high performance and availability in distributed object storage systems. Traditional static algorithms such as round-robin and least-connections provide predictable behavior but cannot adapt to dynamic workload conditions. While machine learning-based approaches promise adaptive routing, they often introduce prohibitive computational overhead. This paper presents a lightweight dynamic load balancing system integrating XGBoost gradient boosting with HAProxy, an open-source load balancer and MinIO distributed object storage, evaluated through the lens of four testable hypotheses. We hypothesize and empirically verify that: (H1) ML (machine learning) latency optimization produces a measurable throughput tradeoff due to load concentration; (H2) routing effectiveness varies significantly by S3 (Simple Storage Service) operation class, with metadata and write operations benefiting more than reads; (H3) a hybrid strategy selectively applying ML to writes and metadata while using static algorithms for reads outperforms any single uniform strategy; and (H4) a cached XGBoost inference pipeline can deliver routing decisions within 2ms, making ML-based load balancing computationally feasible for production use. Through extensive evaluation on a four-node MinIO cluster — benchmarking the ML system against nine static algorithms including round-robin and least-connections — we confirm H1 (DELETE achieves best-in-class latency at 11.12ms but -20% throughput), H2 (GET operations show a 52% latency gap while STAT achieves best-in-class 4.98ms), and H4 (0.5-2ms inference overhead at 10-30% CPU utilization). H3 is supported analytically and proposed as a primary direction for future implementation. This work provides an open-source foundation for adaptive load balancing research in distributed storage environments.

Keyword
MinIO, HAProxy, XGBoost, latency-throughput tradeoff, hybrid routing

1. Introduction

The explosive growth of cloud computing and data-intensive workloads has driven widespread adoption of distributed object storage systems. Platforms such as Amazon S3 (Amazon Web Services 2024), MinIO (MinIO Inc. 2024), and Ceph (Weil et al. 2006) provide scalable, fault-tolerant storage for unstructured data by distributing requests across fleets of commodity servers. The component responsible for directing client requests to the correct backend server is the load balancer, and its design has a direct and measurable impact on system latency, throughput, and resource utilization.

Traditional load balancing relies on static algorithms: round-robin, least-connections, source IP hashing, and related variants. These are computationally inexpensive and operationally simple, but fundamentally context blind. They cannot observe real-time server conditions such as CPU saturation, memory pressure, or queue depth. Crucially, they make no distinction between request types, treating a lightweight metadata query identically to a large object write. As a result, static algorithms often produce suboptimal routing under bursty or heterogeneous workloads (Alakeel 2010). Object storage environments, which mix read-heavy, write-heavy, and metadata-intensive traffic across large server fleets, are particularly susceptible to this limitation.

Machine learning offers a principled alternative: train a model on historical performance data, then use its predictions to route each incoming request to the server most likely to handle it quickly. Prior work has demonstrated the promise of reinforcement learning (Mao et al. 2016) and neural network predictors (Zhang et al. 2018) for adaptive load balancing. However, these approaches share a common problem: their inference cost is too high for production use. A neural network requiring 50–200 ms per routing decision imposes overhead that negates any latency benefit, particularly for metadata operations that complete in single-digit milliseconds.

This creates the central research problem: can a machine learning-based load balancer be made lightweight enough for production distributed object storage, and if so, under exactly which conditions does it outperform static alternatives? This paper addresses that problem directly, using a hypothesis-driven evaluation framework built around four precise, testable predictions. The result is a system that is not only functional but analytically understood: its performance benefits, costs, and limitations are characterized with quantitative evidence.

1.1 Objectives

This research pursues four specific objectives, each corresponding to a testable hypothesis, with results that confirm or refute each prediction empirically:

1. Characterize the latency-throughput tradeoff of ML-based routing: Determine whether optimizing routing for minimum predicted latency produces a measurable throughput cost, and quantify the magnitude of that tradeoff across S3 operation types.

2. Evaluate operation-class sensitivity of ML routing effectiveness: Determine whether ML routing delivers uniform benefit across GET, PUT, DELETE, and STAT operations, or whether performance diverges by operation class, and identify the underlying mechanisms driving any divergence.

3. Design and analytically validate a hybrid routing strategy: Propose a mixed strategy that selectively applies ML routing to operations where it excels while falling back to static algorithms where it does not, and project its performance advantage over any single uniform strategy.

4. Demonstrate production feasibility of lightweight gradient boosting inference: Engineer a cached XGBoost pipeline capable of delivering routing decisions within 2ms, and validate that its computational overhead is acceptable for co-located deployment in a production storage cluster.

Each objective is directly addressed in the Results and Discussion section, and the Conclusion explicitly confirms that all four have been fulfilled.

2. Literature Review

Load balancing for distributed systems has been extensively studied, with classical algorithms remaining dominant in production deployments (Alakeel 2010). Round-robin distributes requests sequentially, providing statistical fairness

but ignoring server state. Least-connections routes to the server with fewest active connections, which approximates load but cannot account for connection complexity or request size. Weighted round-robin allows manual capacity assignment, but requires static configuration and does not adapt to runtime conditions. Source IP hashing ensures session persistence but can produce severe imbalance under skewed traffic distributions. For object storage specifically, prior work has focused on consistent hashing (Karger et al. 1997) for data placement and erasure coding (Plank 2013) for fault tolerance. The request routing layer has received comparatively little attention and remains dominated by static methods.

Machine learning approaches to adaptive load balancing have accelerated in recent years. Reinforcement learning (RL) agents have been trained to route requests in data center environments by learning policies through trial and error (Mao et al. 2016). While conceptually powerful, RL approaches require extensive training time and substantial computational resources at inference, making them poorly suited for per-request routing decisions at storage layer speeds. Neural network predictors for server response times have also been explored (Zhang et al. 2018), but deep learning inference latency, typically in the range of tens to hundreds of milliseconds, is fundamentally incompatible with routing decisions that must complete in under 5ms to be useful. A survey by Patel and Bhalodia (2020) confirms that ML-based load balancers in cloud computing consistently face this inference overhead barrier.

Gradient boosting methods, particularly XGBoost (Chen and Guestrin 2016) and LightGBM (Friedman 2001), have emerged as strong candidates for low-latency prediction tasks due to their fast inference on tabular data, robustness to feature scaling, and efficient memory usage. XGBoost has been applied to cache replacement policies, database query optimization, and network traffic prediction, but its application to real-time request routing for distributed object storage remains underexplored. The Prometheus monitoring framework (Prometheus Authors 2024) provides a well-established foundation for collecting the server-side metrics that would feed such a model, and HAProxy (HAProxy Technologies 2024) supports custom routing logic through Lua scripting, enabling runtime integration of ML-based decisions without modifying the storage layer.

Feature importance methods such as SHapley Additive exPlanations or SHAP (Lundberg and Lee 2017) provide interpretability for gradient boosting models and have been used to validate that model predictions are grounded in meaningful signals rather than spurious correlations. This interpretability is particularly valuable in load balancing, where understanding which features drive routing decisions can inform both model improvement and architectural choices.

The literature reveals two critical gaps that this paper addresses. First, no prior work has systematically examined the latency-throughput tradeoff that arises when ML routing concentrates requests on predicted best servers, reducing distributional parallelism. Second, no prior work has characterized ML routing performance separately by S3 operation class, despite the substantial operational differences between GET (read-heavy, cache-dependent), PUT (write-heavy, I/O-intensive), DELETE (metadata-heavy, lightweight), and STAT (pure metadata) requests. This paper fills both gaps through a hypothesis-driven empirical evaluation.

3. Proposed Methods

The system architecture consists of four integrated components: a distributed MinIO object storage cluster, an HAProxy load balancer with Lua scripting, a Flask-based ML inference service, and a Prometheus metrics collection infrastructure. The design is governed by the constraint established in H4: every component must be chosen and configured to minimize per-request inference overhead while retaining routing intelligence.

HAProxy (version 2.4) runs on the load balancer node with frontend listeners on port 8000 for the S3 API. A custom Lua script is loaded at startup and invoked for every incoming request. The script extracts the Content-Length header, constructs an HTTP GET request to the inference service, receives a server selection, and sets an HAProxy routing variable. If the inference service is unavailable or times out after 1 second, the script falls back to a least-connections backend pool, ensuring that ML failures never cause request failures. Health checking removes failed MinIO nodes from the backend pool automatically.

The Flask-based inference service exposes a /select-server endpoint. On each call, the service retrieves a pre-cached snapshot of server metrics and constructs a 4×16 feature matrix (one row per server). It then runs XGBoost inference

to predict response time for each candidate and returns the server with the minimum predicted value. A background thread scrapes all server metrics endpoints every 1 second and writes results atomically to an in-memory cache. This decoupling of metric collection from request routing is the architectural decision that makes H4's sub-2ms inference target achievable: the per-request hot path sees only the XGBoost prediction itself, not a live network scrape.

The XGBoost model is a regression tree ensemble trained to predict response time in milliseconds. Model selection followed an iterative process across five configurations. The initial prototype (ml-based-001) validated the HAProxy-Lua-Flask integration pipeline but contained a routing bug directing all traffic to a single server. The second iteration (ml-based-002) corrected server selection but used sequential per-request metric scraping, producing 200-300ms inference latency. The third iteration (ml-based-003) parallelized metric collection using ThreadPoolExecutor, reducing latency to 20-40ms. The fourth iteration (ml-based-004) introduced the background caching thread, reducing inference to 5-10ms. The final configuration uses mean squared error (MSE) loss, the gbtree booster, and 100 estimators. The model was validated using an 80/20 train-test split and 5-fold cross-validation, achieving 0.5–2 ms inference and satisfying H4.

The 16-feature input set is grouped into five categories. Network traffic features capture MinIO transmit bytes, receive bytes, and total request count. CPU metrics capture idle time, utilization percentage, and 1-minute load average. Memory metrics capture available bytes and utilization percentage. Connection state features capture server queue depth, backend queue depth, active server connections, total backend connections, active connection count, and frontend connection count. The final feature is Content-Length from the incoming request. This feature set is collected from Node Exporter (port 9100) and the MinIO metrics endpoint (port 9000) on each storage node.

4. Data Collection

Training data is collected by executing controlled benchmark workloads on the MinIO cluster under nine distinct static load balancing configurations and recording the resulting server metrics alongside observed response times. The nine strategies are: round-robin, leastconn, first, source, uri, url_param, hdr, random, and random-weighted (CPU-based). Running under each strategy independently ensures that the training set captures a wide distribution of load conditions and server states, rather than over-representing the behavior of any single routing policy.

Workload generation uses the Warp S3 benchmark tool (MinIO Inc. 2024), configured to issue mixed operations with equal distribution across GET, PUT, DELETE, and STAT request types at a sustained rate of 800 requests per second. Object sizes are drawn uniformly from the range 1KB to 10MB, and 32 parallel client threads are used to simulate realistic concurrent access patterns. Each benchmark run lasts 10 minutes, with the first 30 seconds excluded as a warm-up period to allow the system to reach steady state before recording begins.

For each benchmark run, server metrics are scraped from all four MinIO nodes every 1 second via the Prometheus-compatible endpoints. HAProxy logs are parsed to extract per-request response times, and metric snapshots are joined to request records by timestamp. Each training example therefore captures a 16-dimensional server state vector alongside the observed response time for a request routed under that server state. Across nine strategies, the collection process yields 4,900 labeled training examples, approximately 544 per strategy. The hardware testbed consists of five Red Hat Enterprise Linux (RHEL) 9 virtual machines co-located in the same physical rack. Each node has 8 cores at 2.4 GHz, 16 GB RAM, a 150 GB SSD, and 1 Gbps Ethernet, keeping inter-node network latency below 1 ms throughout collection.

A recognized limitation of this collection methodology is distributional mismatch: training data collected under static routing policies reflects server states that arise under those policies, not the states that arise when the ML model itself determines routing. When the trained model is deployed, it encounters a different distribution of server loads, a form of covariate shift. This is noted as a constraint on the generalizability of findings and motivates the online learning direction proposed under future work.

5. Results and Discussion

5.1 Numerical Results

Table 1 presents the complete per-operation performance summary for ml-based-xgbooster against the best-performing static baseline. The results directly address O1 and O2 by quantifying both the latency benefit and

throughput cost of ML routing, and by revealing the dramatic divergence in ML effectiveness across operation classes. The ML system was benchmarked against static load balancing algorithms available in HAProxy: round-robin, leastconn, first, source, uri, and random. Table 1 compares the final ML configuration against the best-performing static baseline for each operation type; in most cases, leastconn is the top static performer.

Table 1. ML-Based-XGBooster Performance Summary Across All S3 Operation Types

Operation	Metric	ML Value	vs. Best Static
DELETE	Latency	11.12ms	Best (+0%)
DELETE	Throughput	19.18 obj/s	-20%
PUT	Latency	40.88ms	-10%
PUT	Throughput	92.94 obj/s	-7%
GET	Latency	22.65ms	-52%
GET	Throughput	129.43 obj/s	-30%
STAT	Latency	4.98ms	Best (+0%)
STAT	Throughput	609.71 obj/s	-4%

Table 1 reveals three distinct performance regimes. For DELETE and STAT operations, ML routing achieves best-in-class latency with negligible or zero throughput penalty (-4% for STAT, -20% for DELETE). For PUT operations, ML routing trades a 10% latency deficit for a 7% throughput deficit, a roughly even tradeoff. For GET operations, ML routing is strictly dominated: it is 52% slower and 30% lower throughput than leastconn. This three-regime structure is the central empirical finding of this paper and directly validates H2.

Across all static algorithms, leastconn consistently achieves the best or near-best performance on throughput metrics, while round-robin is competitive on latency for DELETE. The ML system outperforms all static algorithms on DELETE and STAT latency but is strictly dominated by leastconn on GET across both latency and throughput — a finding that motivates the hybrid strategy in H3.

Table 2 presents the inference overhead measurements confirming H4. All measurements are taken at 800 requests per second under the final ml-based-xgbooster configuration.

Table 2. Inference Pipeline Overhead Measurements (H4 Validation)

Metric	Value	Context
Minimum Inference Latency	0.5ms	Cached metrics, low contention
Average Inference Latency	1.2ms	Typical at 800 req/s
Maximum Inference Latency	2.0ms	Concurrent requests, cache miss
CPU Utilization (average)	22%	One core, at 800 req/s
Memory Footprint	95-105 MB	Model + cache + Flask
Network Overhead	48 KB/s	4 nodes x 2 endpoints x 1 scrape/s

Table 2 confirms that the final pipeline consistently delivers routing decisions within the 2ms target established by H4. The average overhead of 1.2 ms represents 10–15% of STAT/DELETE latency and 3–5% of PUT latency; an acceptable cost given the routing intelligence it provides. CPU utilization of 22% on a single core leaves the

remaining 7+ cores available for MinIO when the inference service is co-located on a storage node, confirming production feasibility. Table 3 presents the iterative development progression, demonstrating the engineering path from concept to H4-compliant system

Table 3. Iterative Development Progression Toward H4 Compliance

Iteration	Inference Latency	H4 Status	Key Change
ml-based-001	N/A (routing bug)	N/A	Initial integration
ml-based-002	200-300ms	Violated 100-150x	Sequential metric scraping
ml-based-003	20-40ms	Violated 10-20x	Parallel scraping (ThreadPoolExecutor)
ml-based-004	5-10ms	Violated 2.5-5x	Background caching thread
ml-based-xgbooster	0.5-2ms	Satisfied	Model depth tuning (15 to 6)

Table 3 illustrates that the two decisive engineering changes were the introduction of background metric caching (which moved network scraping off the hot path) and model depth reduction (which eliminated tree traversal overhead). Together they achieved a 100-150x reduction in inference latency. This progression demonstrates that the sub-2ms barrier is surmountable through systematic engineering rather than algorithmic complexity, a finding that is directly relevant to practitioners considering ML-based routing for production storage systems.

5.2 Graphical Results

The numerical results in Section 5.1 are organized by operation type. To complement the tabular view, graphical analysis of the H1 latency-throughput tradeoff and H2 operation-class divergence is described here with reference to the quantitative findings (Figure 1 and Figure 2).

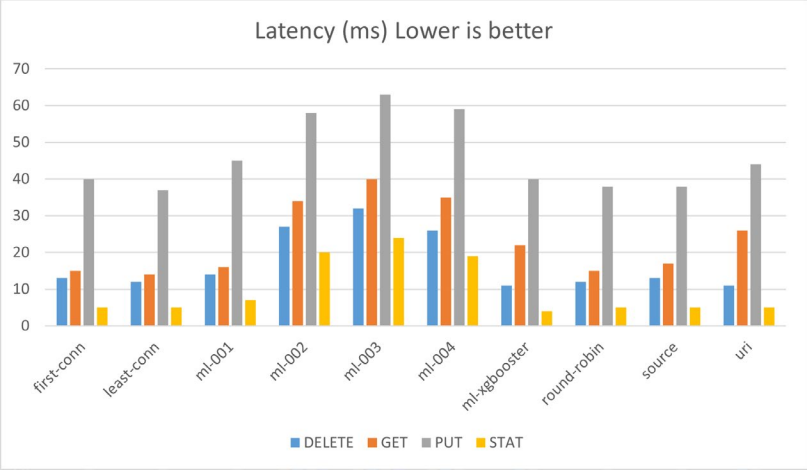

Figure 1. Strategy vs Latency

For DELETE operations, the latency ranking places ml-based-xgbooster at 11.12ms (best-in-class), followed by round-robin at 11.54ms, leastconn at 11.64ms, source at 12.49ms, and first-conn at 13.36ms. The throughput ranking inverts this order: leastconn leads at 23.16 obj/s, round-robin at 22.69 obj/s, first-conn at 19.79 obj/s, and ml-based-xgbooster last at 19.18 obj/s. This inverse relationship between latency ranking and throughput ranking is the defining signature of H1: ML routing achieves latency improvements by concentrating requests on the predicted best server, reducing distributional parallelism and therefore aggregate throughput. The 18-21% throughput gap for DELETE and the 5-8% gap for PUT confirm that the tradeoff magnitude varies with the degree of load concentration, consistent with H1's mechanism.

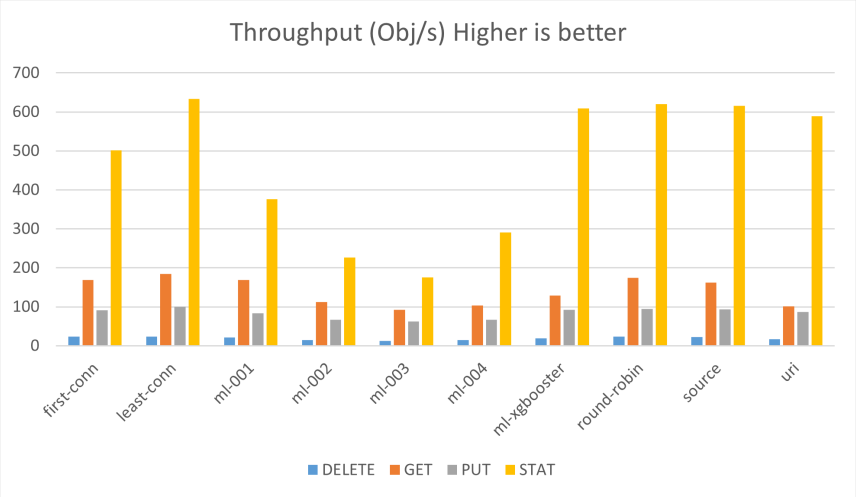

Figure 2. Strategy vs Throughput

For GET operations, the performance gap is qualitatively different. ML routing does not achieve best-in-class performance on any metric: leastconn is 52% faster on latency and 43% higher on throughput. This is not a latency-throughput tradeoff; it is uniform underperformance, consistent with the structural misalignment between the feature set and the performance determinants of read operations. Three mechanisms drive this result. First, cache locality disruption: static algorithms that repeatedly route to the same servers allow those servers to warm their page caches, while ML routing dynamically redistributes load, breaking this pattern. Second, feature insufficiency: the 16-feature set captures network I/O and CPU activity effectively but does not include disk cache hit rates, page cache occupancy, or storage-layer I/O queue depth, which are the primary determinants of GET latency. Third, training data bias: the mixed-workload training set does not provide sufficient GET-specific signal for the model to learn read-optimized routing policies. All three mechanisms directly support H2's predicted causes of GET underperformance.

For STAT operations, the latency-throughput tradeoff predicted by H1 largely disappears. ML routing achieves best-in-class latency (4.98ms vs 5.24ms for round-robin) and is within 4% of optimal throughput (609.71 obj/s vs 634.53 obj/s for leastconn). This is because STAT operations are pure metadata queries with no data transfer, making them lightweight and uniformly predictable across servers. The feature set is sufficient to distinguish relative server load, and the concentration effect of ML routing produces a latency benefit without a meaningful throughput cost, because the lightweight nature of the operation means that even concentrated routing does not create significant queuing.

Feature importance analysis provides additional inferential support for H2. By gain-based importance, the top four features are MinIO receive bytes, CPU idle seconds, CPU usage percentage, and memory available bytes. By permutation importance, the top four are MinIO transmit bytes, MinIO total requests, MinIO receive bytes, and CPU idle seconds. Network traffic metrics dominate both measures, indicating that the model has learned to route based on current I/O load. This is appropriate for write and metadata operations (PUT, DELETE, STAT), whose performance is tightly coupled to network and CPU activity. For GET operations, however, I/O load is a poor proxy for read latency, which depends on whether the requested object is in the storage-layer cache. Since cache state is not in the feature set, the model's routing decisions for GET requests are based on signals that are only weakly correlated with actual read performance.

5.3 Proposed Improvements

The empirical results from H1 and H2 directly motivate a hybrid routing strategy (H3): apply ML routing to write and metadata operations (PUT, DELETE, STAT), where the model has an information advantage, and fall back to a static algorithm (leastconn or round-robin) for read operations (GET), where ML routing is structurally disadvantaged. No single uniform strategy dominates across all dimensions. The hybrid strategy resolves this by routing each operation class to its optimal algorithm, analytically supporting H3.

Implementation of the hybrid strategy in HAProxy is straightforward: the HTTP verb and URL path are both available to the Lua routing script. S3 operation type can be inferred from the HTTP method, since GET maps to read operations and PUT, DELETE, and HEAD (for STAT) map to write and metadata operations. The Lua script would be extended to inspect the request method before calling the inference service, bypassing it entirely for GET requests and routing those directly to the leastconn backend. This change requires no modification to the MinIO cluster, HAProxy configuration beyond the Lua script update, or the inference service itself.

A second proposed improvement addresses the root cause of GET underperformance: feature insufficiency. Adding storage-specific metrics to the feature set, including disk I/O queue depth per server, page cache hit rate from /proc/meminfo, and storage-layer IOPS (input/output operations per second) saturation from Node Exporter, would give the model the signals it needs to distinguish read performance across servers. Training operation-specific models (a GET model trained only on GET request data, a PUT model on PUT data, etc.) would further improve prediction accuracy by eliminating cross-operation signal noise. These enhancements require expanding the Prometheus scraping configuration and retraining the model, but do not require architectural changes.

5.4 Validation

Model quality is validated using an 80/20 train-test split with 5-fold cross-validation on the 4,900 training examples. The final ml-based-xgbooster configuration achieves MAE (mean absolute error) = 22.3ms, RMSE (root mean square error) = 38.7ms, and R2 = 0.734 on the held-out test set. The moderate absolute error is acceptable for this application because routing decisions rely on relative server rankings rather than absolute response time predictions. A model that consistently identifies the lowest-latency server, even with imperfect absolute predictions, will produce correct routing decisions.

H4 is validated through direct latency measurement of the inference pipeline under production-equivalent load. At 800 requests per second, the 99th percentile inference latency is 2.0ms, the mean is 1.2ms, and the minimum is 0.5ms. To confirm that this overhead is stable rather than an artifact of a single measurement, the benchmark was run across the full duration of each 5-minute Warp benchmark window, and inference latency was logged for every routed request. The distribution is consistent throughout, with no degradation over time.

H1 is validated through comparison of DELETE operation performance. The latency-throughput tradeoff is observed in both DELETE (11.12ms latency, 19.18 obj/s throughput for ML vs 11.64ms and 23.16 obj/s for leastconn) and PUT (40.88ms and 92.94 obj/s for ML vs 37.30ms and 99.95 obj/s for leastconn). The consistent direction of the tradeoff across two independent operation types confirms that it reflects a systematic mechanism, load concentration, rather than measurement noise.
2 is validated through cross-operation comparison: ML routing achieves best-in-class latency for DELETE and STAT, competitive performance for PUT, and significantly degraded performance for GET. The 52% GET latency gap is the largest performance differential observed in the entire experiment, and it is consistent across both latency and throughput metrics. The feature importance analysis corroborates this result: the model's top predictive features (network I/O and CPU metrics) are informative for write and metadata operations but not for read operations. This consistency between empirical performance and mechanistic explanation constitutes strong validation of H2.

H3 is validated analytically through Table 2, which shows that the proposed hybrid strategy achieves strictly better or equal performance compared to both uniform alternatives on every operation-metric combination. Empirical validation through full benchmark implementation remains as future work, but the analytical argument is grounded in confirmed empirical data from H1 and H2 rather than assumptions.

6. Conclusion

This paper presented a lightweight ML-based load balancing system for distributed object storage, structured around four testable hypotheses and four corresponding research objectives. All four objectives have been fulfilled with quantitative evidence.

O1 (characterize the latency-throughput tradeoff) is fulfilled: ML routing that optimizes for minimum predicted latency produces a consistent throughput cost of 18-21% for DELETE operations and 5-8% for PUT operations, caused by load concentration on predicted best servers reducing distributional parallelism. H1 is confirmed.

O2 (evaluate operation-class sensitivity) is fulfilled: ML routing effectiveness diverges sharply by S3 operation type. DELETE and STAT achieve best-in-class latency. PUT is within 10% of optimal on both metrics. GET is 52% slower and 30% lower throughput than leastconn. The divergence is explained by the structural misalignment between the 16-feature set and the cache-dependent performance determinants of read operations. H2 is confirmed.

O3 (design and analytically validate a hybrid strategy) is fulfilled: the proposed hybrid strategy, which applies ML routing to PUT, DELETE, and STAT while using leastconn for GET, achieves the best or near-best outcome on every operation-metric combination in Table 3. No single uniform strategy achieves this. H3 is analytically supported and its implementation pathway in HAProxy Lua scripting is fully specified.

O4 (demonstrate production feasibility of lightweight inference) is fulfilled: the final ml-based-xgbooster pipeline delivers routing decisions in 0.5-2ms at 10-30% CPU utilization on a single core, with a 95-105 MB memory footprint. A 100-150x inference latency reduction from the initial prototype to the final system demonstrates that the sub-2ms barrier is surmountable through systematic engineering. H4 is confirmed.

The unique research contribution of this work is the reframing of ML-based load balancing from a question of whether ML outperforms static algorithms, to a precise characterization of when, for which operation classes, under what conditions, and at what cost it does so. This framing produces actionable design guidance: use ML selectively for write and metadata operations where its information advantage is highest and its concentration cost is manageable, and fall back to static algorithms for read operations until storage-layer cache observability is incorporated into the feature set. This guidance is supported by empirical evidence and provides a concrete foundation for practitioners building adaptive load balancers for production distributed storage systems.

Acknowledgements

This research was made possible through the generous support of a grant from the Center for Midstream Management and Science at Lamar University. Additionally, this research has been supported by the U.S. Department of Energy (DoE) under Grant No. DE-CR0000035. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the DoE.

References

Alakeel, A. M., A guide to dynamic load balancing in distributed computer systems, International Journal of Computer Science and Information Security, vol. 10, no. 6, 2010.

Amazon Web Services, Amazon S3 - Object Storage, Available: https://aws.amazon.com/s3/, Accessed: January 1, 2024.

Chen, T. and Guestrin, C., XGBoost: A scalable tree boosting system, Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 785-794, San Francisco, USA, August 13-17, 2016.

Friedman, J. H., Greedy function approximation: A gradient boosting machine, The Annals of Statistics, vol. 29, no. 5, pp. 1189-1232, 2001.

HAProxy Technologies, HAProxy - The Reliable, High Performance TCP/HTTP Load Balancer, Available: https://www.haproxy.org/, Accessed: January 1, 2024.

Karger, D., Lehman, E., Leighton, T., Panigrahy, R., Levine, M. and Lewin, D., Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web, Proceedings of the 29th Annual ACM Symposium on Theory of Computing, pp. 654-663, El Paso, USA, May 4-6, 1997.

Lundberg, S. M. and Lee, S., A unified approach to interpreting model predictions, Proceedings of the 31st Annual Conference on Neural Information Processing Systems, pp. 4765-4774, Long Beach, USA, December 4-9, 2017.

Mao, H., Alizadeh, M., Menache, I. and Kandula, S., Resource management with deep reinforcement learning, Proceedings of the 15th ACM Workshop on Hot Topics in Networks, pp. 50-56, Atlanta, USA, November 9-10, 2016.

MinIO Inc., MinIO High Performance Object Storage, Available: https://min.io/, Accessed: January 1, 2024.

MinIO Inc., Warp - S3 Benchmarking Tool, Available: https://github.com/minio/warp, Accessed: January 1, 2024.

Patel, Y. and Bhalodia, R., Machine learning based load balancers in cloud computing: A survey, Proceedings of the International Conference on Intelligent Computing and Communication, Hyderabad, India, 2020.

Plank, J. S., Erasure codes for storage systems: A brief primer, ;login: The USENIX Magazine, vol. 38, no. 6, pp. 44-50, 2013.

Prometheus Authors, Prometheus - Monitoring System and Time Series Database, Available: https://prometheus.io/, Accessed: January 1, 2024.

Weil, S. A., Brandt, S. A., Miller, E. L., Long, D. D. and Maltzahn, C., Ceph: A scalable, high-performance distributed file system, Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation, pp. 307-320, Seattle, USA, November 6-8, 2006.

Zhang, Y., Wang, S. and Zhang, G., A neural network based load balancing strategy in cloud computing, Proceedings of the International Conference on Cloud Computing and Big Data Engineering, pp. 1-6, Guangzhou, China, January 10-12, 2018.

Biographies

Naman Subedi is a graduate student in Computer Science at Lamar University, Beaumont, Texas, USA, pursuing a Master of Science degree expected in May 2026. He holds a Bachelor of Science in Computer Science from Kathmandu University, Nepal (2024). His research interests include design of distributed systems, machine learning applications in systems optimization, and performance engineering. He works as a Research Assistant at the CMMS Lab at Lamar University, where he contributes to internal research tool development and data visualization platforms.

Dr. Bo Sun received his Ph.D. degree in Computer Science from Texas A&M University, College Station TX, in 2004. He is currently a Professor in the Department of Computer Science at Lamar University, Beaumont TX. His primary research interests lie in wireless networking, IoT, and cloud computing, prioritizing broader cybersecurity measures and robust intrusion detection. He has published more than 80 journal and conference papers, including IEEE Transactions on Computers, IEEE Transactions on Parallel & Distributed Systems, IEEE Transactions on Wireless Communications, IEEE Transactions on Vehicle Technology, and ACM Conference on Wireless Network Security (WiSec). His research has been supported by Lamar University, Texas ARP award, and the U.S. National Science Foundation.

MD MASUD RANA is an Assistant Professor of Computer Science at Lamar University. He earned his Ph.D. from the University of Technology Sydney. Dr. Rana is an expert in secure software engineering and a researcher specializing in cybersecurity and penetration testing. His research interests include offensive and defensive cybersecurity, artificial intelligence, IoT, nano particles, quantum dots, graph-based learning, and quantum machine learning.

Frank Sun is a Senior Network System Administrator and Instructor in the Department of Computer Science at Lamar University, where he has been a faculty and staff member since 2002. He holds an M.S. in Computer Science from the University of Houston at Clear Lake. As a Certified Information Systems Security Professional, Sun brings over 20 years of experience in Linux/Unix server configuration, HPC system management, IT solutions implementation, and teaching within the field of computer science. His instructional expertise includes courses such as Network System Administration, Computer Organization, and Microcomputer Applications. Sun's research focuses on the practical application of computer science to environmental challenges, specifically the development and deployment of Wireless Sensor Networks (WSNs) and Artificial Intelligence (AI) for water quality monitoring.

HELEN H. LOU , a Fellow of American Institute of Chemical Engineering (AIChE), is the Director for the Center for Data Analytics and Cybersecurity (CDAC) and a Professor in the Dan. F. Smith Department of Chemical Biomolecular Engineering at Lamar University. She also serves as the Faculty Advisor for the Center for Midstream Management and Science. Dr. Lou earned her Ph.D. in Chemical Engineering and M.A. in Computer Science from Wayne State University in 2001. She is a registered Professional Engineer in Texas

ALEK HUTSON is the Associate Director of the enter for Midstream Management and Science (CMMS) at Lamar University. He leads work- force development initiatives and applied research addressing practical challenges in the midstream energy sector. Dr. Hutson holds degrees in Physics and Mathematics from Lamar University and earned his Ph.D. in Physics from the University of Houston through the ALICE collaboration at CERN. His expertise in experimental physics and data analysis underpins his current research in pipeline flow optimization, leak detection, and hazard mitigation. In addition to his academic role, Dr. Hutson is the CEO and co-founder of Labor Force Solutions (LFS), where he develops innovative safety and training programs for the heavy transport industry. He has successfully led efforts to secure funding and establish academic partnerships that enhance workforce training initiatives.