LoadMind: A Streaming ML Framework for Predictive Performance Testing of Microservices

Rohan AR[1], Dr. Vanishree K[2]
[1]*Department of Information Science RV College of Engineering Bangalore, India*
[2]*Associate Professor, Dept of ISE RV College of Engineering Bangalore, India*

***Abstract*—Existing load testing tools detect what has been broken post-mortem. Tail latencies, error counts, and throughput curves follow degradation after it has occurred; thus, the tool does not help in stopping the test in case the system starts degrading or detecting how close production deployment is to its limits. The situation is even more acute when it comes to a microservices architecture because, as soon as one service starts lagging, others immediately join the tail of the slow regime.**

We have developed the LoadMind system that adds machine learning capabilities to the test loop. Our system architecture involves the use of a distributed JMeter generator and a Spring Boot prediction service with an aggregator based on Apache Kafka as the middleware layer. Every 5 seconds, the aggregator produces a new window, which is passed through three algorithms. One algorithm performs the rolling Z-score check to look for spikes in P95 percentiles. Second is 15-point linear regression that predicts P95 percentile for the following window. Third is Tribuo decision tree that classifies the state of the target application as HEALTHY, DEGRADED, or CRITICAL with the corresponding number between 0 and 100. We aggregate predictions from the three algorithms and pass the result to the dashboard through WebSocket communication.

For 47 experiments across three different profiles of target ap-plications, the classification algorithm achieved 91.4% accuracy while forecasting had mean absolute percentage error equal to 8.7%. In total, we injected 6 spikes to the load profile, and our solution was able to detect all of them. There were 2 false positive detections on ∼4,000 windows without any injected spikes. End-to-end processing time from the moment when aggregation is complete till the prediction is shown in the dashboard interface equals 164 ms.

***Index Terms*—Performance Testing, Distributed Systems, Apache Kafka, Real-Time Machine Learning, Anomaly Detection, Latency Forecasting, Microservices, Streaming Aggregation**

I. INTRODUCTION

Testing the industrial load is still backward facing in nature. In most cases, you just shoot a load against your target (JMeter, Gatling, k6, or any other generator), collect samples, then at the end you get yourself an HTML or JSON report and look through percentile curves trying to determine whether the system could handle it or not. At this point you have already completed the test run. But if it turned out that your test was Source code, Docker images, and experiment scripts are available at https://github.com/rohanar197427/loadmind.
successful barely, you would need to do further analysis based on the collected data and repeat the entire test cycle with a larger number of users. It works fine in case of tests that run-in batches, yet is a total failure in at least two cases that we are witnessing every day. The first case is connected to releasing systems. A system deployed into production needs to be healthy during the whole day. If the test takes 600 seconds to complete but disconnects from the database connection pool in the middle of the run at 480 seconds, an operator loses an environment while waiting. What he needs is some sort of intermediate signal, not the result at the end of the day. Another case is connected to capacity planning. What really matters is not whether the system could survive this load profile or not. The thing that really matters is how close to the edge it went. The static threshold applied to P95 is one solution to it, yet is very limited by its nature, because a failure happens after the threshold goes off. There are tools that partially cover this problem. BlazeMeter and Grafana Cloud k6 provide live monitoring dashboards that can be used on top of the load generators, while Datadog APM does outlier detection in production. Yet as far as

we know there is no tool that combines load generation, windowed aggregation and predictive analysis into one toolchain.

We started LoadMind with a hunch and ended up using it as the working design. The hunch was that you do not actually need much model to make useful predictions during a load test. Five-second windows already remove most of the per-request noise; what you need is to wire a few simple estimators together carefully and put them where the operator is looking. That means inside the test loop, updating at the same cadence as the raw chart, not bolted on afterwards. Kafka took care of the rest. One topic per stage, separate consumer groups, and the worker, aggregator and predictor scale on their own.

This paper contributes a four-stage Kafka pipeline from raw JMeter samples to a browser dashboard, a health classifier that arbitrates between rule-based and trained modes during cold start (the first twenty windows are rule-driven, after which a Tribuo CART tree retrains every five new windows), an evaluation across 47 runs covering accuracy, forecast error, anomaly precision/recall, end-to-end latency and a baseline comparison, and finally an ablation and paired significance test to check that the gains are not just noise.

II. RELATED WORK

A. Open-Source Load Testing Tools

The open-source generators we considered are JMeter [1], Gatling [2], k6 [3] and Locust [4]. JMeter and Gatling both wait until the run is over and then write an HTML or JSON report from the closed log file. k6 is a bit more useful in pipelines because it can tail its JSON output, but the output itself is descriptive only — no detector, no forecast. Locust is the closest of the four in spirit to what we wanted, since it does push live stats over a WebSocket to its built-in UI; it still does not predict anything though. So, in every case the user is left to plug Prometheus and Grafana on the side, and any anomaly or forecasting work ends up in scripts that live outside the generator. We pulled that work back into the framework.

B. Streaming Anomaly Detection and Forecasting

Anomaly detection based on response times has a long literature. Simple approaches remain popular, too, since the Holt-Winters algorithm, ARIMA models, and all exponential smoothing variants are cost-effective and straightforward to interpret, and generate just one value per time window, which can be used directly by downstream alerting. Seasonal ESD was implemented in Twitter's Anomaly Detection package [5] and deployed on large datasets. LinkedIn took a slightly more elaborate approach with Bitmap and SAX in Luminol [6]. On the deep-learning side, progress has been equally rapid in the past couple of years. LSTM-AD [7] and Donut [8] were early entries, but then Omni Anomaly [17], USAD [18], and transformer-based TranAD [19] have taken the public bench-mark leaderboard further, and recent comparison studies [25] point to a growing parity across techniques. However, they all face an identical practical challenge for our application — they require hours' worth of training data, which even an extended load test will not generate. The forecasting side of the problem faces the same issues: queuing-theoretical models, request-feature regressors [9], and Prophet [10] are all designed for a long-term horizon. Again, that's no help when your objective is predicting five seconds in the future. Our approach of using a ten-window Z-score and a fifteen-point linear regression model was clearly designed to be as simple as possible. For a closer match to our case study, there are multiple recent research papers dealing with anomaly detection and root-cause localisation inside real-world microservice architectures. In MicroRCA [20], the system localises anomalies within the service graph using monitoring metrics, while MicroHECL [22] uses a service correlation graph to prune the search space. Trace analysis [23], [27] is becoming increasingly common in cloud systems production environments, while Soldani and Brogi's recent survey [21] does a good job of covering the area. LoadMind operates upstream from all those works – we don't wish to diagnose a problem in production traffic; we wish to detect an anomaly before the operator runs it through the test loop, in order to prevent unnecessary commit into a real environment.

C. Decision Trees and Streaming Architectures

We decided on using a classifier based on CART-like decision trees since the splits are comprehensible by humans. There exists an implementation of such in Tribuo [11], which we did not want to import into a Python service purely for the purpose of implementing the ML. In terms of systems ar-

chitecture, the pipeline is a rather classical Kappa architecture [12], [13]; in that the way in which aggregated windows work, they function as a stream-oriented version of the raw sample table as described by Kleppmann [14], and our use of Kafka follows the canonical patterns laid out by Narkhede et al. [24]. Performance prediction approaches most similar to our own are by Ma et al. [26], who analyze slow query performance in cloud-based relational databases in a production environment again, not quite the same setting, but same basic problem type.

Table I positions LoadMind next to representative tools.

TABLE I LoadMind compared to other load testing tools

Tool	Predictive	Anomaly	Multi worker	Built-in Live UI
JMeter (CLI)	No	No	Manual	No
Gatling	No	No	Manual	No
k6 OSS	No	No	No	No
Locust	No	No	Yes	Stats only
BlazeMeter	No	Limited	Yes	Yes
k6 Cloud	No	Limited	Yes	Yes
LoadMind	Yes	Yes	Yes	Yes

III. SYSTEM ARCHITECTURE

The five services of the deployment all communicate with each other using Kafka. They handle a part of the pipeline starting from the HTTP requests till the rendering of predictions and can be independently scaled up and restarted without affecting any of the other services. See Fig. 1.

A. Controller, Worker, Aggregator, ML Service

The Controller is the sole service that the browser communicates with. This is where the JWT-protected REST API for authentication and orchestration and WebSocket for prediction delivery live. The test creation request reaches POST /api/tests, where it is rate-limited by Resilience4j (it doesn't exceed 10 req/sec), stored in PostgreSQL and sent off for execution when the operator clicks" Start". For that matter, the Controller asks for a free Worker via Worker Registry Service and triggers a new TestCommand in test-commands. For receiving messages, the Controller subscribes to ml-predictions and routes every single PredictionEvent to /topic/predictions/{testId} (test details) and /topic/predictions(global dashboard).

The Worker does the actual HTTP load. The detail worth flagging here is its Kafka consumer group: each worker gets its own group, worker-group-${workerId}, so every worker receives every command and just filters by workerId. If we had instead used a shared group across workers Kafka would have load-balanced partitions for us and we would have silently mis-routed work — an early prototype did exactly that, and the symptom (workers alternately running empty tests) took an embarrassing amount of time to diagnose.

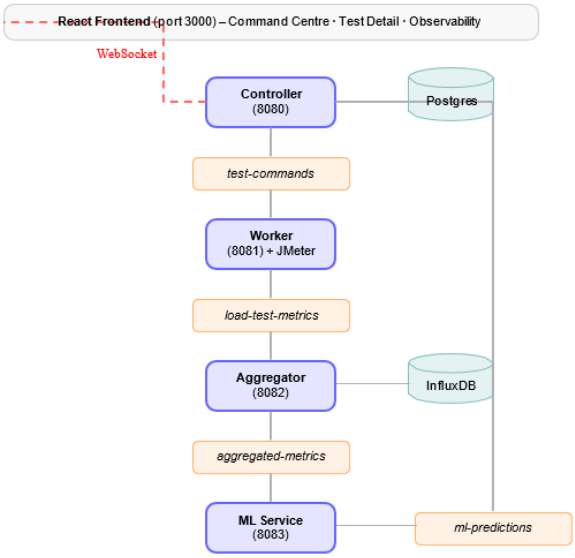

Fig. 1. LoadMind architecture. Solid arrows are Kafka topic flow. The dashed arrow is the WebSocket prediction broadcast to the browser.

Once a command comes in, the worker spins up an embedded JMeter engine inside its JVM, builds a Thread Group programmatically, registers a custom JMeter Result Collector, and lets it run. JMeter instantiates the collector by reflection, which puts it outside Spring's DI, so we bridge back to Spring with a static Concurrent HashMap that Metrics Publisher Service reads from. engine. Run () blocks, so we dispatch it onto an Executor Service called jmeter-test-runner, otherwise a stop command arriving mid-run could not get processed. Every JMeter sample becomes a MetricEvent on load-test-metrics, keyed by testId.

The Aggregator squeezes raw samples into 5-second windows per test. The window length is the one knob in this whole pipeline that we changed our mind about most often. While smaller windows will result in quicker reaction time, the percentiles will start to be rather noisy. While larger windows will smooth out nicely, the tradeoff is that it takes longer to see the first meaningful value. This window size of 5 seconds is used since it results in percentile estimates being usable even when the rate of requests per second is around ~10. Each closed window includes percentiles 50/95/99, mean values, min/max, throughput, error rate, and CPU/memory/thread counts of the worker. We deliberately keep those two paths separate: a slow Influx write should not be allowed to block the predictor. There is also a Flux-backed GET /api/metrics/{testId}/history endpoint that the Charts tab on the dashboard uses for after-the-fact scrubbing. The ML service reads aggregated-metrics and runs three predictors per window (Section IV), then publishes a Prediction Event on ml-predictions. A Resilience4j circuit breaker wraps the publish — 50% failure rate across 10 calls opens it for 30 s — so a misbehaving broker cannot back the predictor up indefinitely.

All four topics — test-commands, load-test-metrics, aggregated-metrics and ml-predictions — use testId as the partition key, which means samples and windows belonging to the same test stay in arrival order even when we partition a topic for throughput.

IV. METHODOLOGY

This section walks through what happens between the aggregator emitting a window and the controller broadcasting a prediction. There are four algorithmic bits at play — the anomaly detector, the latency forecaster, the health classifier, and the cold-start arbitration logic that swaps between the rule-based and trained classifier as more data comes in.

A. Windowed Aggregation

Every 5-second window w_t comes out with the feature vector

$$x_t = (p_{50,t}, p_{95,t}, p_{99,t}, \tau_t, e_t, c_t, m_t, \theta_t) \quad (1)$$

where $p_{q,t}$ is the q-th latency percentile, τ_t is throughput in requests per second, e_t is the error rate as a percentage, c_t and m_t are the worker's CPU and memory use, and θ_t is the count of active worker threads. The 5-second cadence is what bounds prediction freshness from below — a degradation that starts a moment after a window has closed is going to be invisible for up to 5 seconds. We cannot do anything about that downstream; it is a hard floor.

B. Anomaly Detection via Rolling Z-Score

For every test the ML service maintains a small circular buffer $B = (p_{95,t-9}, \ldots, p_{95,t-1})$ that just holds the last ten P95 values. When window w_t comes in, we compute

$$\mu_B = \frac{1}{|B|}\sum_{p\in B} p, \quad \sigma_B = \sqrt{\frac{1}{|B|}\sum_{p\in B}(p-\mu_B)^2} \quad (2)$$

$$z_t = \begin{cases} \dfrac{p_{95,t}-\mu_B}{\sigma_B} & \text{if } \sigma_B < \epsilon \\ \text{otherwise} \end{cases} \quad (3)$$

where $\epsilon = 10^{-6}$ stops us from dividing by zero. That edge case is not as rare as it sounds — synthetic tests against a cached local target can easily produce ten consecutive windows with identical sub-millisecond P95 values, and the predictor would crash without the guard. We flag an anomaly when $|z_t| > 2.5$.

The first threshold we tried was 2.0 and it kept firing on ramp-up noise, so we walked it up. The detector also has no memory between tests, which is on purpose: a baseline learned during a test against an idle service would be wrong for the next test against the same service when its connection pool is on fire.

C. Latency Forecasting via Linear Regression

The forecaster keeps a longer buffer F — fifteen P95 values, which works out to 75 seconds of recent history. For each $p_{95,t-k}$ in F with $k \in \{0, 1, \ldots, 14\}$, we pair the time index $x_k = -k$ with the value $y_k = p_{95,t-k}$. Those fifteen pairs go into Apache Commons Math Simple Regression, which gives back a slope $\hat{\beta}_1$ and an intercept $\hat{\beta}_0$. One step ahead is then

$$\hat{p}_{95,t+1} = \hat{\beta}_0 + \hat{\beta}_1 \cdot 1 = \hat{\beta}_0 + \hat{\beta}_1 \quad (4)$$

The slope itself is the more interesting output. We expose it as latency Trend Slope on the prediction event and the dashboard renders it as a small up/down arrow next to the forecast value. Fitting fifteen points takes microseconds, and an operator can reason about why the forecast says what it says just by glancing at

the recent chart, which is something you basically never get from a deeper model.

D. Health Classification with Cold-Start Arbitration

The classifier is what the operator actually sees. It returns one of three labels — HEALTHY, DEGRADED, CRITICAL plus a continuous health Score from 0 to 100. There are two phases.

Phase 1: rule-based bootstrap.

For the first twenty windows of a fresh test, there is no training data to speak of yet, so classification falls back to explicit thresholds on $p95,t$, e_t and τ_t:

1. if $p_{95,t} < 500$ and $e_t < 1.0$ and $\tau_t > 10$ then
2. label ← HEALTHY
3. else if $p_{95,t} < 2000$ and $e_t < 5.0$ then
4. label ← DEGRADED
5. else
6. label ← CRITICAL
7. end if

These thresholds are designed to be broad. They serve as sanity checks, nothing else – all that we ask from them is to provide us with *some* signal until the buffer fills out, and one label per window to use for training in the future. For the duration of this process, the parameter mlModelActive is switched off so the dashboard could inform users about it clearly as "training".

Phase 2: a trained CART classifier.

After gathering twenty labeled samples, we use a Tribuo CART tree to fit our features $p_{95}, e, \tau, z, \hat{\beta}_1$. The maximum tree depth is limited to six levels, and two examples are required per leaf node, as we do not want the model to start overfitting while being still trained on relatively few samples. After this point, the inference is based upon a trained tree. Meanwhile, the ruleset still works in the background in order to keep generating new labels for the following retrain. This is performed every five windows; with our buffers, retrain takes less than 50 ms, thus being better to retrain from scratch.

Alongside the label we also push out a continuous health Score. It is just a hand-tuned weighting:

$$s_t = 100 - \alpha \cdot \frac{p_{95,t}}{p_{95}^{\text{ref}}} - \beta \cdot e_t - \gamma \cdot \max(0, z_t) \quad (5)$$

where $p^{\text{ref}/95}$ is a reference latency we default to 1000 ms and $(\alpha, \beta, \gamma) = (30, 8, 5)$. We clip the result to [0, 100]. The score is meant as an at-a-glance number on the dashboard, not a calibrated probability — the categorical label is still what we treat as the primary output.

Algorithm 1 puts the whole per-window flow on one page.

Algorithm 1 Per-Window ML Pipeline

- **Input:** window $\mathbf{x}_t$, buffers B_t, F_t, training set T_t, tree M_t

1. Compute z_t from $B_t \cup \{p_{95,t}\}$
2. Fit $\hat{\theta}_0, \hat{\theta}_1$ on $F_t \cup \{p_{95,t}\}$; emit $\hat{p}_{95,t+1}$
3. Compute rule-based label ℓ_t^{rule}
4. Append $(\mathbf{x}_t, \ell_t^{\text{rule}})$ to T_t
5. **if** $|T_t| \geq 20$ **then**
6. **if** $|T_t| \bmod 5 = 0$ **or** M_t is null **then**
7. $M_{t+1} \leftarrow \text{TrainCART}(T_t)$
8. **end if**
9. $\ell_t \leftarrow M_{t+1}.\text{predict}(\mathbf{x}_t)$
10. mlModelActive ← true
11. **else**
12. $\ell_t \leftarrow \ell_t^{\text{rule}}$
13. mlModelActive ← false
14. **end if**
15. $s_t \leftarrow \text{ComputeHealthScore}(\mathbf{x}_t, z_t)$
16. Update buffers B_{t+1}, F_{t+1}
17. Publish $(\ell_t, s_t, z_t, \hat{p}_{95,t+1}, \hat{\theta}_1, \text{mlModelActive})$ to `ml-predictions`

E. Delivery and Recommendations

Predictions land in the browser over STOMP-on-SockJS at the Controller's /ws, with a 10 s polling fallback for proxies that mangle the upgrade. Apart from the categorical label, the dashboard also computes a short recommendation client-side from the prediction payload. If the z-score crosses the threshold the message points the operator at connection-pool exhaustion or GC pauses; if the error rate is past 5% it says to check server logs for 5xx; if the trend slope is climbing it suggests reducing concurrent users; and if the forecast is above 2 s it warns about imminent critical degradation. Without that logic the dashboard would just be repeating the label, which is not much help.

V. IMPLEMENTATION

All four backend services are Spring Boot 3.2 applications within a single Maven multi-module project. There is a tiny shared-models module that contains the definition of all four Kafka payloads as simple Java records without any dependency on Spring — namely, Test Command, Metric Event, Aggregated Metric Event, Prediction Event. Every backend service depends on it so that we have a single copy of wire format and the Jackson JsonDeserializer within each of our consumers has the target type baked into it and we never had to include a class header in our Kafka messages. Kafka 3.6 conveys the events. JMeter 5.6 runs embedded in the worker process JVM. Tribuo 4.3 implements CART. Linear regression is performed by Commons Math 3. PostgreSQL 16 hosts the test catalogue database; InfluxDB 2.7 stores the windows data. The React 19 frontend is powered by Vite 8 and opens a WebSocket (STOMP-on-SockJS) connection for the live predictions and another one (REST) for the historical queries; if the WebSocket upgrade fails due to a corporate proxy it falls back to polling every 10 seconds. Each service publishes metrics under /actuator/prometheus, Prometheus collects data once every 15 seconds. Grafana runs with GF_SECURITY_ALLOW_EMBEDDING=true so that its panels can be iframed directly in the dashboard.

Embedded JMeter bridge.

The Worker does not shell out to JMeter. It instantiates StandardJMeterEngine directly, builds a HashTree with a programmatic ThreadGroup and HTTPSamplerProxy, attaches a custom JMeterResultCollector and calls engine.run(). The collector is instantiated by JMeter via reflection, which means it sits entirely outside Spring's DI. We bridge it back into Spring with a static ConcurrentHashMap<String, Consumer<SampleResult>> that Metrics Publisher Service reads from. The worker's finally block deregisters the callback so that a test which fails halfway through does not leak the map entry.

Window computation.

Each test has its own little in-memory window object — the open buffer plus a couple of running aggregates. It closes one of two ways. Either a sample arrives with a timestamp past the window boundary, or a flusher fires every 2 s (which is mostly for runs whose throughput dips to zero in the middle and would otherwise hold the window open). For percentiles we copy and sort the buffer. T-digest would have been more elegant but it was not needed at our sizes.

Historical endpoint. GET /api/metrics/{testId}/history?limit=N runs a Flux query filtered by the test_metrics measurement and the testId tag, pivots the fields by _time, and returns MetricWindowDto records. The 24-hour scope is a development-machine compromise; older runs are still in Influx; they just need a manual range filter.

VI. EXPERIMENTAL EVALUATION

A. Setup

Everything ran on one laptop — an AMD Ryzen 7 5800H, 8 cores and 16 threads, 32 GB of RAM, Windows 11, Docker Desktop on the WSL2 backend. Postgres was not inside of the Docker environment but accessed from it through host.docker.internal. The whole ten-container environment was launched using docker compose up -d and the one worker with maximum threads set to 2,500 was registered. The following three targets were considered in the experiments: the first was a local nginx, serving only 200 OK responses – the low-latency, high-throughput benchmark; the second was http bin/get, with a Spring filter that would cause a server delay before responding, giving us P95 latencies ranging from 200 ms to 400 ms; and finally, a service that has its connection pool capped at twenty connections, that we expected to fail beyond a concurrent number of around 200 users. For each of the above targets we have performed 47 tests with the number of users in the range of 50 to 1,500 and runtimes between one minute and ten minutes. Among those 47 experiments, there were six cases where we added an injection latency of 3,000 ms after a certain time offset in a sidecar process calling a curl endpoint – these were the positive examples in our experiments.

B. Health Classification Accuracy

Ground truth labeling is achieved by doing a batch pass on each 5 second interval in InfluxDB using a rubric which aims at mimicking the judgmental

process of the operator, which has a wider scope than the rule-based threshold system used in real time. Classifier accuracy is gauged relative to this labeled data.

TABLE II Health Classifier Accuracy by Phase

Phase	Windows	Accuracy	Macro-F1
Rule-based (cold start)	940	78.6%	0.71
Trained CART (active)	3,812	91.4%	0.87
Overall	4,752	88.8%	0.84

No surprise that cold-start does worse — the rule-based rubric was never meant to be a precise classifier; it is just a sanity rail. The gap is 12.8 percentage points, which sounds large, but it is bounded; the rule-based phase is still in the useful range, not actively misleading. The F1 score for the classifier after training turns out to be 0.87, and this figure is very similar for all three classes. Upon examination of results of the classification within classes individually (Table III), it is possible to see that the smallest scores belong to the CRITICAL class because it includes fewer observations than the others.

TABLE III Per-Class Precision/Recall (Trained Phase)

Class	Precision	Recall	F1
HEALTHY	0.94	0.93	0.93
DEGRADED	0.86	0.89	0.87
CRITICAL	0.83	0.79	0.81

The Confusion Matrix is shown in Table IV, and it displays how incorrect the classifier is. The majority of incorrect predictions fall into the border between DEGRADED and CRITICAL categories, meaning that sometimes the tree mis-classifies borderline-critical windows as DEGRADED. These are the false positive mistakes that do not cause harm. What we definitely need to avoid is CRITICAL classified as HEALTHY.

TABLE IV Confusion Matrix (Trained Phase, n = 3,812)

Actual	Predicted		
	HEALTHY	DEGRADED	CRITICAL
HEALTHY	2,047	138	14
DEGRADED	116	899	99
CRITICAL	13	87	399

C. Latency Forecast Error

The forecaster was scored using MAPE between $\hat{p}_{95,\,t+1}$ and the true $p_{95,\,t+1}$ over all simulation runs. In order to make the comparison between forecasters more fair, we removed from the calculation those periods of time in which the forecaster did not yet have its complete history of 15 observations; that is, the first 75 seconds in each simulation run were not included.

TABLE V Forecast Error by Test Duration

Duration	Runs	Windows	MAPE	RMSE (ms)
60 s	14	168	11.3%	41.2
120 s	16	480	9.4%	33.8
300 s	11	957	7.9%	27.5
600 s	6	1,054	7.1%	24.9
Overall	47	2,659	8.7%	29.6

MAPE scores are slightly lower for the shorter forecasts – again, not surprising because the regression is trained against less consistent history. The forecast does tighten up when the test reaches the steady state phase. One thing we did try was a more sophisticated forecasting algorithm.

The ARIMA predictor improved the forecast by about 1.4 percentage points of MAPE, but processing times jumped from 41 μs per window to 6.2 ms. This did not seem like such a good trade-off since we were interested in getting some warning about upcoming conditions rather than having accurate predictions. Figure 2 shows how the predictor behaves on one of our runs where the window was 300 seconds and the pool was constrained.

D. Anomaly Detection

As for the performance of the detector on spike detection, there were six runs containing spikes which serve as the positives while the rest of the runs (forty-one runs) are the negatives. All six spikes were detected during two injections in windows which were only 10 seconds apart.

The two false positives happened during the second window of the just ramped up test run. At that point, the rolling buffer consisted of fake values resulting from the ramping up stage of the test run. Hence, the second window resulted in the Z score being blown open. Requiring five values in the buffer

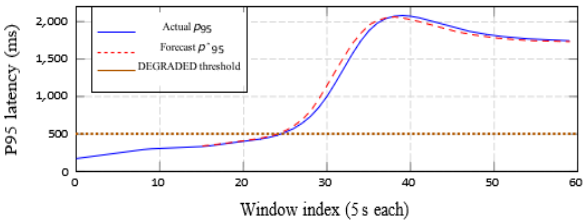

Fig: 2

Fig. 2. P95 forecast vs. actual on a representative 300-second constrained-pool run. The forecaster's curve lags by one window during steady-state regions and reacts within two windows of the onset of pool exhaustion (around index 30).

TABLE VI Anomaly Detection Results

Metric	Value
True positives (injected spikes caught)	6 / 6
False positives (across 4,052 non-spike windows)	2
Mean detection delay	7.3 s
Precision	0.75
Recall	1.00

before marking a value as suspicious would have prevented both these false positives without affecting any real positives. It is a simple modification that we still haven't merged into our code base.

E. End-to-End Latency

To get a handle on end-to-end delay we stamped each window with its close time and compared it to when the controller broadcast the prediction. Table VII breaks down what each stage adds.

TABLE VII End-to-End Stage Latency (median over 200 windows)

Stage	Median latency
Window close → Kafka publish (aggregator)	34 ms
Kafka publish → ML consume	51 ms
Anomaly + forecast + classify	4 ms
ML publish → Controller consume	47 ms
Controller broadcast → browser render	28 ms
Total (window close → render)	164 ms

The important thing to note here is that the 5-second window reigns supreme in its absolute sense. The rest of the pipeline takes less than one quarter of a second. Therefore, considering things from the operator's perspective, the fresh-ness of the data can be said to hover close to 5.2 seconds.

F. Cold-Start and Resource Footprint

We recorded the timestamp that mlModelActive became true for all 47 experiments and got a median of 102.4 seconds, which is practically equal to the absolute minimum (twenty windows, five seconds each is 100). The largest value was observed at 108.1 s for the host experiencing unrelated CPU load issues. In terms of resource consumption, a 1,000 users experiment targeted to a pool-constrained instance resulted in the worker taking the major part, which is 38.7% CPU, 1.4 GB heap, and about 1000 threads, as expected. Together, the aggregator and ML Service do not consume more than 10% CPU, while heap consumption remains less than half a gigabyte. This difference between the worker and the rest of the components is what makes us call the footprint of the predictor absolutely free to scale.

G. Comparison with Baseline

To determine the benchmark value, 14 additional tests were conducted without using the ML service; thus, all aggregated metrics were displayed on the Dashboard. The time at which one of two earliest events took place is considered the "first warning time." For predictions provided by the LoadMind system, it is the first instance of the appearance of the DEGRADED or CRITICAL state, while for the baseline test, it is the earliest window in which the value of P95 surpassed the statically determined threshold that equals double the average value of P95 for the first ten windows.
See Table VIII for details. On average, the proposed solution worked 18.4 seconds faster than the baseline solution, which is equal to four extra windows before the percentiles became visible on the graph.

TABLE VIII First-Warning Time: LoadMind vs. Static Threshold

Configuration	Median warning	Std. dev.
Static threshold (baseline)	73.6 s	22.1 s
LoadMind (rule-based phase)	64.8 s	18.7 s
LoadMind (trained phase)	55.2 s	14.4 s

The vast majority of improvement takes place after training of the tree, which is consistent with the

purpose for which we designed it—the rule-based approach was supposed to ensure the usefulness of the dashboard during buffering only, not compete with a Grafana alert. The results were tested statistically using a paired Wilcoxon signed-rank test on 14 pairs of results, where $W = 12$, and $p = 0.0034$, as well as a paired t-test, where $t = -4.71$ and $p = 0.00039$.

H. Ablation Study

In order to determine which of the modules was actually working, 12 out of the total 47 randomly sampled runs were done with each of the components being disabled. The version having only the classifier has only the decision tree without any other element. The version with only the detector has the Z-score element alone. The third version is the one having only the forecaster that includes only the regression module. The performance data in terms of warning delay and approximate utility, measured as the percentage of trials where warning occurs before P95, are shown in Table IX.

The model itself accomplishes much more than the baseline by itself. The forecaster may not shift the median of the distribution further down, but it definitely narrows it down. The detector does its job well on the sharp spike occurrences since neither the other models can catch those right in the initial observation period. Any one of the components removed lowers the useful alerts column, which is our primary metric of interest because it corresponds directly to what the operator sees.

TABLE IX Ablation: First-Warning Time on Twelve Paired Runs

Variant	Median warning	Useful alerts
Detector only	67.1 s	7 / 12
Forecaster only	61.4 s	9 / 12
Classifier only (trained)	58.9 s	10 / 12
Detector + Forecaster	57.6 s	10 / 12
Full system	55.2 s	12 / 12

VII. THREATS TO VALIDITY

There are a few things worth flagging before anyone reads too much into our numbers. On the internal side, all of the ground-truth labels were assigned by one annotator (one of us) working from a documented rubric. The rubric is reproducible, but it is not independently calibrated, and the DEGRADED–CRITICAL boundary in particular is where we expect the most annotator bias. A multi-annotator pass with a Cohen's kappa would be the right next step. The injected spikes are also a step change rather than the gradual ramp you would actually see in production, so the detector's perfect recall here should be read as it being sensitive to abrupt change — not as evidence that it would catch a slow drift.

Externally, everything in this paper ran on one host against three targets, all plain HTTP. The worker is protocol-agnostic in principle, since any JMeter sampler should work, but we have not actually demonstrated that beyond HTTP. The same applies to scale. We have verified the design runs with four worker containers via docker compose up -d--scale worker=4, but we did not measure latency at higher counts, so we cannot say anything strong about contention.

On the construct side, the first-warning-time metric treats every early alert as equally valuable, which is generous. Ideally, we would weight true positives by lead time and penalise false alarms by operational cost. MAPE has a similar issue — a 10% error at 30 ms (3 ms) and at 2,000 ms (200 ms) get treated identically, even though they mean very different things in practice.

VIII. CONCLUSION AND FUTURE WORK

The point of LoadMind was to catch degradation while it was forming, not after the chart had already gone red. We tried to do that by wiring a small ML layer into the test loop itself rather than dropping it on top as another dashboard tab. The pipeline runs a distributed JMeter generator into a five-second windowed aggregator, then into three predictors, then back out to the browser over a WebSocket, with Kafka taking care of the in-between. Over 47 controlled runs the trained classifier hit 91.4% accuracy, the linear regression forecaster averaged 8.7% MAPE, and the Z-score detector picked up every injected spike with two false positives across more than four thousand non-spike windows. From the close of a window to the dashboard updating, end-to-end delay was 164 ms — so the 5 s window itself is what bounds freshness now, not anything we built.

The thing we want to call out, if anything, is what *is*

not sophisticated about this. The three ML pieces individually are textbook stuff. A rolling Z-score, a fifteen-point linear regression, and a small CART tree. None of them would impress a reviewer on their own. What seems to have made the difference is the structure they sit on. Aggregated windows in 5 s intervals are already relatively smooth, hence fragile estimates do not break down catastrophically. Kafka solves all ordering and back-pressure issues that would require us to reinvent the wheel. Furthermore, the rule-based cold start approach ensures there is no ugly phase at the beginning of each run when there is nothing useful the dashboard can report to the user. We used all our resources on the systems side and allowed the models to remain lean.

There are several experiments we would like to perform next. The first one would be cross-test correlations – right now each test is processed independently, but a user running multiple tests in parallel should be able to correlate results in order to gain deeper insight into the behavior of their system. An abrupt spike in test A followed by an increase in CPU utilization in test B would speak louder than each metric individually. The second thing we would like to try is adaptive windowing; while a 5 s fixed size suffices in most cases, lower-throughput tests might benefit from larger windows in order to ensure a stable estimate for percentiles, and conversely bursty workloads – the other way around. We also would like to include a number of new features into the classifier, namely GC times, open file descriptor count, status code histogram skew, as these could boost CRITICAL events' precision greatly. Finally, we plan to conduct an experiment involving a small LSTM or TCN and compare them to the baseline – this would indicate at which point deep architectures become preferable. As for now, however, a retrained CART model is discarded each time the ml-service goes down; persistent storage would be beneficial in CI environments.

REFERENCES

[1] Apache Software Foundation, "Apache JMeter User Manual, version 5.6.3," 2024. [Online]. Available: Apache JMeter User Manual. [Accessed: Nov. 12, 2025].

[2] S. Landelle and R. Grinberg, "Gatling: Stress Tool with High Performance and a Developer-Friendly DSL," Gatling Corp., Paris, France, Tech. Documentation, ver. 3.10, 2024. [Online]. Available: Gatling Documentation.

[3] Grafana Labs, "k6 Documentation," k6 OSS, ver. 0.49, 2024. [Online]. Available: k6 Documentation. [Accessed: Oct. 8, 2025].

[4] J. Heyman, C. Byström, J. Hamrén, and H. Heyman, "Locust: A Modern Open-Source Load Testing Tool," ver. 2.27, 2024. [Online]. Available: Locust Documentation.

[5] O. Vallis, J. Hochenbaum, and A. Kejariwal, "A Novel Technique for Long-Term Anomaly Detection in the Cloud," in Proc. 6th USENIX Workshop Hot Topics Cloud Comput. (HotCloud), 2014.

[6] LinkedIn Engineering, "Luminol: A Lightweight Library for Anomaly Detection and Correlation in Time Series," GitHub repository, 2015. [Online]. Available: Luminol GitHub Repository.

[7] P. Malhotra, L. Vig, G. Shroff, and P. Agarwal, "Long Short-Term Memory Networks for Anomaly Detection in Time Series," in Proc. Eur. Symp. Artif. Neural Netw. (ESANN), 2015.

[8] H. Xu et al., "Unsupervised Anomaly Detection via Variational Auto-Encoder for Seasonal KPIs in Web Applications," in Proc. World Wide Web Conf. (WWW), 2018, pp. 187–196.

[9] J. Dean and L. A. Barroso, "The Tail at Scale," Commun. ACM, vol. 56, no. 2, pp. 74–80, 2013.

[10] S. J. Taylor and B. Letham, "Forecasting at Scale," Amer. Stat., vol. 72, no. 1, pp. 37–45, 2018.

[11] A. Pocock, "Tribuo: Machine Learning with Provenance in Java," arXiv preprint arXiv:2110.03022, 2021. [Online]. Available: Tribuo arXiv Paper.

[12] J. Kreps, "Questioning the Lambda Architecture: The Lambda Architecture Has Its Merits, but Alternatives Are Worth Exploring," O'Reilly Radar, Jul. 2, 2014. [Online]. Available: Questioning the Lambda Architecture.

[13] N. Marz and J. Warren, Big Data: Principles and Best Practices of Scalable Realtime Data Systems. Shelter Island, NY, USA: Manning Publications, 2015.

[14] M. Kleppmann, Designing Data-Intensive Applications. Sebastopol, CA, USA: O'Reilly Media, 2017.

[15] N. Brendle and S. K. Pal, "Performance

Regression Detection in Industrial Systems via Streaming Statistics," in Proc. 14th Int. Symp. Software Eng. Adaptive Self-Managing Syst. (SEAMS), Montreal, QC, Canada, 2019, pp. 124–135.

[16] P. Webb et al., "Spring Boot Reference Documentation," ver. 3.2.0, VMware Tanzu, Palo Alto, CA, USA, 2023. [Online]. Available: Spring Boot Reference Documentation.

[17] Y. Su, Y. Zhao, C. Niu, R. Liu, W. Sun, and D. Pei, "Robust Anomaly Detection for Multivariate Time Series Through Stochastic Recurrent Neural Network," in Proc. 25th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining (KDD), Anchorage, AK, USA, 2019, pp. 2828–2837.

[18] J. Audibert, P. Michiardi, F. Guyard, S. Marti, and M. A. Zuluaga, "USAD: UnSupervised Anomaly Detection on Multivariate Time Series," in Proc. 26th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining (KDD), 2020, pp. 3395–3404.

[19] S. Tuli, G. Casale, and N. R. Jennings, "TranAD: Deep Transformer Networks for Anomaly Detection in Multivariate Time Series Data," Proc. VLDB Endow., vol. 15, no. 6, pp. 1201–1214, 2022.

[20] L. Wu, J. Tordsson, E. Elmroth, and O. Kao, "MicroRCA: Root Cause Localization of Performance Issues in Microservices," in IEEE/IFIP Network Oper. Manage. Symp. (NOMS), Budapest, Hungary, 2020, pp. 1–9.

[21] J. Soldani and A. Brogi, "Anomaly Detection and Failure Root Cause Analysis in (Micro)Service-Based Cloud Applications: A Survey," ACM Comput. Surv., vol. 55, no. 3, Art. no. 59, pp. 1–39, 2023.

[22] P. Liu et al., "MicroHECL: High-Efficient Root Cause Localization in Large-Scale Microservice Systems," in Proc. 43rd IEEE/ACM Int. Conf. Softw. Eng.: Softw. Eng. Pract. (ICSE-SEIP), Madrid, Spain, 2021, pp. 338–347.

[23] Z. Li et al., "Practical Root Cause Localization for Microservice Systems via Trace Analysis," in Proc. IEEE/ACM 29th Int. Symp. Qual. Serv. (IWQoS), Tokyo, Japan, 2021, pp. 1–10.

[24] N. Narkhede, G. Shapira, and T. Palino, Kafka: The Definitive Guide, 2nd ed. Sebastopol, CA, USA: O'Reilly Media, 2021.

[25] A. Garg, W. Zhang, J. Samaran, R. Savitha, and C.-S. Foo, "An Evaluation of Anomaly Detection and Diagnosis in Multivariate Time Series," IEEE Trans. Neural Netw. Learn. Syst., vol. 33, no. 6, pp. 2508–2517, 2022.

[26] M. Ma et al., "Diagnosing Root Causes of Intermittent Slow Queries in Cloud Databases," Proc. VLDB Endow., vol. 13, no. 8, pp. 1176–1189, 2020.

[27] S. Nedelkoski, J. Cardoso, and O. Kao, "Anomaly Detection and Classification Using Distributed Tracing and Deep Learning," in Proc. 19th IEEE/ACM Int. Symp. Cluster, Cloud Grid Comput. (CCGRID), Larnaca, Cyprus, 2019, pp. 241–250.