International Journal of Scientific Research & Engineering Trends
Volume 12, Issue 5, Sep-Oct-2026, ISSN (Online): 2395-566X

AI-Powered Observability in Cloud-Native DevOps: LSTM-Based Anomaly Detection for Kubernetes Microservices

Parav Sharma, Rajesh Chauhan, Akshay Bhardwaj
Department of Computer Science & Engineering
University Institute of Technology, Himachal Pradesh University
Shimla, India

Abstract — In today's complex cloud-native microservice architectures, the traditional rule-based monitoring approaches are insufficient for system reliability. The solution to this is the integration of Artificial Intelligence (AI) into DevOps, often known as AIOps. This paper introduces an AI-powered observability framework that combines the LSTM (Long Short-Term Memory) autoencoder with the Prometheus-Grafana observability stack for anomaly detection in Kubernetes-based microservice environments. The system collects real-time CPU utilization metrics from different microservices, trains an LSTM model on metric records, and detects anomalies using reconstruction error thresholding. Under controlled CPU stress injection, the characteristic scale of reconstruction error, measured as the mean plus two standard deviations of each condition's own reconstruction errors, is approximately 27 times higher than under baseline conditions, indicating that the model's reconstruction error responds strongly to abnormal workload patterns. Because a single training run may not be representative, the model is retrained ten times on a 6,407-timestep dataset; the reconstruction-error threshold is 0.000676 +/- 0.000038 and the anomaly rate 4.38% +/- 0.26% (mean +/- SD), a coefficient of variation of 5.6%, indicating that the result is reproducible rather than an artifact of one random initialization. The trained model is then deployed as a continuous detector that scores live metrics every 60 seconds; in a controlled test it flagged all 17 readings taken while an injected workload was active and returned to normal after its removal. The LSTM-based model detects anomalies without relying on fixed alert boundaries, offering a more adaptive alternative to static threshold approaches and enabling proactive detection of system abnormalities in cloud-native DevOps environments.

Keywords— AIOps, anomaly detection, cloud-native, LSTM autoencoder, DevOps, Kubernetes, microservice, Prometheus-Grafana, observability, time-series, metric.

I. INTRODUCTION

Today's organizations progressively deploy software systems as collections of loosely coupled microservices that Kubernetes manages[11]. The cloud-native environment provides scalability and development flexibility, but it also comes with operational complexity: Normally, a production environment runs hundreds of independent services, which can generate a vast (thousands) amount of telemetry signals per second[11].

Traditional monitoring techniques like static threshold alerts and manual dashboards are not capable of this scale and dynamism[1]. They generate many false alerts, which cause alert fatigue that leads the operations team to ignore them completely[13]. Microservices are connected, so they can not detect failure patterns that spread over dependent microservices, because a fixed rule on one metric has no knowledge of the whole system[14]. The entire approach is reactive by nature; the anomaly is detected after the damage is done[1].

AIOps (Artificial Intelligence for IT Operations) is an important approach to building a smart, automated observability system. Manchana [1] proposed a four-stage model that defines how organizations can move from simply reactive (reacting to failures) to predicting and automatically fixing them before users are affected. This is a clear theoretical foundation for our work. Some studies have also shown that adding ML (Machine Learning) to cloud-native monitoring tools can meaningfully reduce the MTTD (Mean Time to Detect) and MTTR (Mean Time to Resolve) [2].

This paper makes the following contributions:

- A reference architecture that brings forth Prometheus[4], Grafana[5], and an LSTM-based model to detect anomalies in a Kubernetes cluster[11].
- An LSTM autoencoder[8] is built and trained on real CPU utilization metrics, which are collected from three microservice applications that run on Kubernetes.
- A comparison of reconstruction-error magnitude between baseline and controlled stress-injection conditions, in

which the mean plus two standard deviations of reconstruction error, computed independently on each condition's own data, was approximately 27 times larger under stress (0.701 versus 0.025).

- A repeatable open-source experimental setup on commodity hardware, making it accessible for other researchers without cloud infrastructure or budget.
- A ten-run stability analysis on a 6,407-timestep dataset, reporting the threshold and anomaly rate as distributions rather than single values (coefficient of variation 5.6%), together with a comparison across training durations showing that longer training lowers the threshold but degrades its stability.
- A live deployment of the trained model as a continuous detector, evaluated end-to-end against an injected workload, including its behaviour during the transition back to baseline.

II. RELATED WORK

1. Observability in Cloud-Native Systems

Observability is built on three pillars: metrics, logs, and traces. Metrics are numerical quantifiers for memory and CPU utilization. Logs are text records of every event. Traces track the journey of a single request as it moves across multiple microservices. The CNCF (Cloud Native Computing Foundation)[3] has formally standardized these three fundamental pillars. In a Kubernetes environment, Prometheus has become the most widely used system for collecting metrics[4]. It stores data as a time series and provides a query language known as PromQL.

Grafana is commonly used alongside Prometheus to display these metrics as a visual dashboard and set up alerts when something goes wrong[5]. Together, Prometheus and Grafana have become the foundation of today's cloud-native observability stack.

2. AIOps and Anomaly Detection

Manchana[1] proposed a four-stage theoretical model to help organizations improve their IT monitoring over time. The four stages are: reactive detection, where the system doesn't react until something fails; proactive prevention, where issues are identified before they have an impact on the user; predictive forecasting, where the technology predicts failures before they occur; and automated remediation, where the system heals itself without the need for human intervention. Our work is in the second stage, proactive detection, where the objective is to use an AI model to identify anomalies early.

A 2025 study[2] specifically addressed how AI can enhance observability in cloud-native DevOps environments. Using real Kubernetes telemetry data, the study evaluated three different types of ML (Machine Learning) algorithms: Isolation Forest, One-Class SVM, and LSTM autoencoders[2]. The findings showed that all three algorithms reduced MTTD (Mean Time To Detect) and MTTR (Mean Time To Resolve) in comparison to the traditional rule-based monitoring[2]. However, the LSTM-based model outperforms the other two models for the time-series workload, which exhibits temporal dependencies[2]. This finding directly influenced our decision to use an LSTM autoencoder as the primary detection model.

3. LSTM for Time-Series Anomaly Detection

Hochreiter and Schmidhuber introduced Long Short-Term Memory (LSTM) networks back in 1997[7]. They solved the vanishing gradient problem, a well-known limitation of recurrent neural networks (RNNs)[7]. LSTM introduced the gated memory cells that selectively retain or discard sequential information over time and learn patterns. This feature makes LSTM well-suited for time-series data because it captures long-range temporal dependencies. The autoencoder version of LSTM is frequently used for anomaly detection[8]. This model is trained only on normal data and learns to compress and reconstruct the input sequence. When a sequence is not accurately reconstructed, it is flagged as an anomaly.

The main focus of prior work done on an LSTM autoencoder for IT operations was on log anomaly detection[15]. This paper extends the framework for CPU metrics, which are collected from a Kubernetes microservice environment, showing that the LSTM autoencoder works well with system-level anomaly detection.

III. SYSTEM ARCHITECTURE

There are four integrated layers for the AI-powered observability system shown in Fig. 1.

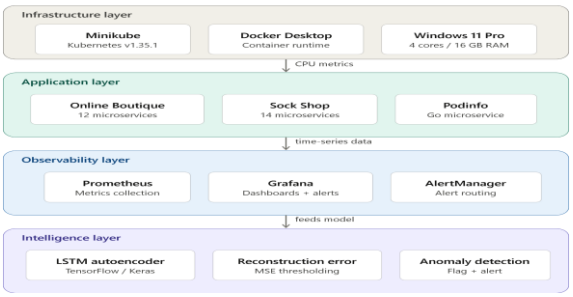

Fig. 1. Proposed AI-powered observability architecture for Kubernetes microservices.

1. Infrastructure Layer
For this experiment, a single-node Kubernetes cluster is configured with Minikube and the Docker driver[11]. For that, we have allocated four cores, 7,500 MB of RAM, and 20 GB of disk space. Minikube was chosen strategically. Cloud-based Kubernetes is expensive and hard to replicate. In contrast, Minikube can run on any laptop and behaves the same way. Any researcher can copy this setup without worrying about the cloud account and budget. Although it is not adequate for large-scale production, it is sufficient for research and study work.

2. Application Layer
Realistic workload metrics were successfully obtained through the deployment of three microservices-based applications. Google's Online Boutique[6] is an open-source e-commerce demo application built from 12 independent microservices, which are written in different languages. It was deployed into a dedicated 'boutique' Kubernetes namespace. Sock Shop[9], a retail microservice demo by Weaveworks, is also deployed with a separate namespace, contributing 14 more microservices. Podinfo is a lightweight service built in Go[10], which is also included. What makes this setup particularly useful is the load generator built into Online Boutique, which continuously simulates realistic user traffic without any manual effort.

3. Observability Layer
Prometheus manages metrics collection by deploying the kube-prometheus-stack Helm chart[4]. This chart deploys multiple components simultaneously, such as Grafana for dashboards, AlertManager for alert routing, kube-state-metrics for cluster-level metrics, and node-exporter for hardware data. Every 15 seconds, Prometheus collects data from every running microservice. A Python data pipeline was written to extract time-series metrics from Prometheus by querying its HTTP API and saving them to a structured CSV file for model training.

4. Intelligence Layer
The LSTM autoencoder is a neural network that learns to compress and reconstruct input sequences. It uses TensorFlow and Keras for anomaly detection[12]. It reconstructs input sequences of CPU utilization values and flags anomalies based on reconstruction error. The model was trained only on normal data and never saw any injected anomalies during training. This is the fundamental concept of an autoencoder.

A threshold was set, which is the mean of the reconstruction error plus two standard deviations. It is set to determine whether a sequence is anomalous. This is a standard statistical cutoff[8]. In the primary, single-application experiment, this statistic was computed independently on each condition's own reconstruction errors, giving 0.025801 under normal conditions and 0.701963 under stress injection, so the characteristic scale of reconstruction error was approximately 27 times larger under stress (Table IV). In a separate combined three-application experiment (Table V), the threshold was 0.077348, with three anomalies detected among 43 test sequences.

This unsupervised approach requires no labeled anomaly data for training, making it practical for real systems where labeled anomaly data rarely exists. The same architecture is used throughout; only the input signal differs between the batch experiments and the live deployment described in Section V-G, where a rate-based formulation is required for the model to remain valid as the cluster continues to run.

IV. METHODOLOGY

1. Data Collection
Two collection procedures are used in this work. The primary experiments (Sections V-A to V-D) query the container_cpu_usage_seconds_total Prometheus metric over one-hour windows at 15-second step intervals, summed across the target namespaces. Two datasets were collected this way: a baseline dataset under normal operating conditions, and an anomaly dataset following injection of a deliberate CPU stress workload (an infinite loop process) into the cluster.

The stability analysis (Section V-E) and the live deployment (Section V-G) use a second procedure. There the query is avg(rate(container_cpu_usage_seconds_total[1m])) evaluated per namespace and then averaged across namespaces, collected over a six-hour window.

Two changes matter here. The rate form is bounded, whereas the cumulative counter grows without limit and therefore drifts outside the range a model was trained on. Averaging rather than summing keeps the signal comparable when the number of reporting containers changes, which occurs whenever pods restart. Timesteps at which any namespace failed to report were discarded so that every observation is composed of the same three namespace averages.

2. Data Preprocessing
Raw metric records are aggregated by timestamp to produce a univariate time-series of total CPU utilization. Values are normalized to [0, 1] using Min-Max scaling. Sliding windows of length 30 (representing 7.5 minutes of metric history) are extracted as model input sequences, yielding overlapping sequence datasets for training and evaluation.

3. LSTM Autoencoder Architecture

The anomaly detection model is implemented as a symmetric LSTM autoencoder. The encoder compresses input sequences into a 64-dimensional latent representation; the decoder reconstructs the original sequence from this representation. Table I details the model architecture.

Table 1. LSTM Autoencoder Architecture

Layer	Type	Output	Params
Encoder	LSTM (64, ReLU)	(None, 64)	16,896
Bridge	RepeatVector(30)	(None,30,64)	0
Decoder	LSTM (64, ReLU)	(None,30,64)	33,024
Output	TimeDist. Dense	(None,30,1)	65
Total	—	—	49,985

4. Training and Threshold Setting

The model is trained using the Adam optimizer with mean squared error (MSE) loss over 20 epochs, batch size 32, with an 80/20 train/test split and 10% validation holdout. The anomaly threshold T is defined as:

$$T = \mu(MSE) + 2\sigma(MSE)$$

where μ and σ are the mean and standard deviation of reconstruction errors on the training set. This two-sigma threshold balances sensitivity and specificity for anomaly detection.

V. EXPERIMENTAL RESULTS

1. Experimental Setup

Table 2. Experimental Environment

Component	Specification
OS	Windows 11 Pro 25H2
CPU	4 Cores / 8 Logical Proc.
RAM	16 GB (7,500 MB to Minikube)
Kubernetes	Minikube v1.38.1 / K8s v1.35.1
Python	3.13 / TensorFlow 2.x
Microservices	12 (Online Boutique)

Table II reports the environment for the primary, single-application experiment (Online Boutique, 12 microservices). The combined three-application configuration described in Section III-B, which additionally includes Sock Shop and Podinfo, is evaluated separately in Table V.

2. Dataset Statistics

Table 3. Dataset Summary

Dataset	Records	CPU Range
Baseline (Normal)	2,824	0.09 – 0.5
Stress Injected	3,014	0.09 – 40.09

The 400x CPU utilization spike introduced by the stress workload (0.09 to 40.09 normalized units) provides a clearly distinguishable anomaly signal for model evaluation.

3. Training Performance

Training loss for the autoencoder decreased consistently from 0.0277 at epoch 1 to 0.000483 at epoch 20. The validation loss followed a similar trend to end at 0.0088, indicating good generalization without overfitting. The model took around 5 minutes to train on a CPU; thus, it is deployable without a GPU requirement.

4. Anomaly Detection Results

Table 4. Anomaly Detection Results

Condition	Thresh.	Anom.	Rate
Normal (Baseline)	0.025801	2/42	4.76%
Stress Injected	0.701963	3/43	6.97%
Error-scale ratio (stress/normal)	27x	—	—

The combined three-application experiment is visualized in Fig. 2, which shows the reconstruction error with detected anomalies (top), training and validation loss (middle), and CPU usage per application namespace (bottom).

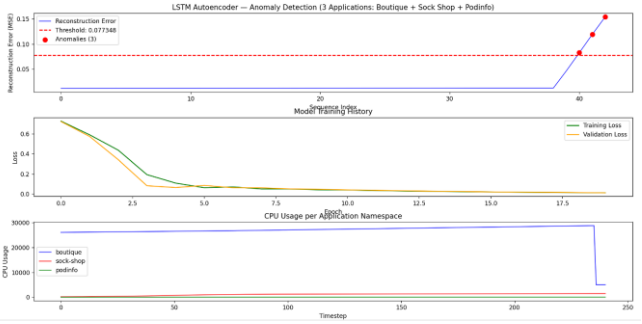

Fig. 2. Combined three-application results: reconstruction error and detected anomalies (top), training and validation loss (middle), and CPU usage per application namespace (bottom).

The main finding of this experiment is a large difference in reconstruction-error magnitude between the two conditions: the mean plus two standard deviations of reconstruction error was

0.025801 at baseline and 0.701963 under stress injection, approximately 27 times larger. Each value was computed independently on that condition's own reconstruction errors, so the comparison describes the characteristic scale of error in each condition; it does not represent a single alert threshold, learned from normal data, that rose in response to stress. For the same reason, the similar flagged fractions in Table IV (4.76% and 6.97%) mainly reflect the self-referenced cutoff, which marks a broadly similar share of any distribution scored against itself, rather than detection performance. Evaluation against a fixed threshold learned from normal data is reported for the stability analysis in Section V-E and for the live deployment in Section V-G.

Table 5. Multi-Application Experiment Summary

Application	Records	Thresh.	Anom.
Online Boutique	3,128	—	—
Sock Shop	3,374	—	—
Podinfo	241	—	—
Combined (3)	6,743	0.0773	3/43

5. Stability Across Training Runs

Reporting a single training run leaves open whether the result depends on the random weight initialization. To address this, the model was retrained ten times on a dataset of 6,407 timesteps (5,101 training and 1,276 test sequences) collected over a 24-hour window. The dataset, scaler, sequence construction, train/test split, architecture and hyperparameters were held identical across runs; only the random weight initialization and the validation shuffling differed. Table VI reports every run individually.

Table 6. Per-Run Results Across Ten Training Runs

Run	Threshold	Anomalies	Rate (%)
1	0.000731	57 / 1276	4.47
2	0.000677	57 / 1276	4.47
3	0.000634	55 / 1276	4.31
4	0.000702	57 / 1276	4.47
5	0.000680	54 / 1276	4.23
6	0.000717	49 / 1276	3.84
7	0.000616	61 / 1276	4.78
8	0.000694	58 / 1276	4.55
9	0.000621	55 / 1276	4.31
10	0.000685	56 / 1276	4.39
Mean	0.000676	55.9 / 1276	4.38
SD	0.000038	3.3	0.26

The threshold is 0.000676 +/- 0.000038, a coefficient of variation of 5.6%, and the anomaly rate is 4.38% +/- 0.26%. Individual thresholds span 0.000616 to 0.000731 and every run flagged between 49 and 61 of the 1,276 test sequences. As Table VII below shows, this 20-epoch configuration also had lower run-to-run variability (CV = 5.6%) than the 40- and 60-epoch configurations (CV = 16.0% each).

Because the reconstruction error falls throughout training, the number of epochs might be expected to affect this variability. Ten runs were therefore repeated at 40 and 60 epochs on the same data; Table VII compares the three settings.

Table 7. Effect of Training Duration on Stability

Epochs	Mean thr.	SD	CV (%)	Rate (%)
20	0.000676	0.000038	5.6	4.38
40	0.000572	0.000092	16.0	4.51
60	0.000531	0.000085	16.0	4.70

Training longer reduces the mean threshold, from 0.000676 at 20 epochs to 0.000531 at 60, which is expected since the model reconstructs the training distribution more closely. Stability, however, moves in the opposite direction: the coefficient of variation roughly triples, from 5.6% to 16.0%. Extended training allows each run to settle into a different local minimum, so although reconstruction errors become smaller they also become less consistent between runs, and the threshold derived from them inherits that spread. Twenty epochs was retained on this basis.

One methodological observation is worth recording. An initial version of this comparison used three runs per setting rather than ten and produced coefficients of variation of 14.7%, 68.5% and 10.2% at 20, 40 and 60 epochs, which would have supported the opposite conclusion. The 68.5% figure was driven by a single outlying run. Variance estimates drawn from small samples are themselves highly variable, and conclusions about stability drawn from a handful of runs should be treated with caution.

The earlier multi-application experiment (Table V) was also repeated: two runs on a 6,743-record dataset gave thresholds of 0.077348 and 0.080474 with 3 of 43 sequences flagged in both cases, and a third run on a separately collected 6,507-record dataset gave 0.055239 with 2 of 43. The lower third value coincides with a partially degraded cluster in which two Online Boutique services were repeatedly restarting, so that variation reflects workload differences in addition to initialization. Characterizing sensitivity to workload and cluster state, as

distinct from initialization, remains open and is noted in Section VI-B.

6. Comparison with Threshold-Based Alerting

Traditional static thresholds require manual configuration with predefined alert boundaries. For example, using static thresholds on the 95th percentile of the baseline CPU usage will lead to (A) missing out alerts for those anomalies which are within the historical maxima OR (B) too many false positives during the spike periods.

Whereas LSTM Autoencoder dynamically learned the normal operating envelope by adapting itself to temporal patterns. Thus, manual configuration is not required in case of LSTMs as well. This comparison is conceptual: a direct empirical benchmark of the LSTM autoencoder against an implemented static-threshold detector on the same dataset was not performed in this study and is identified as future work.

7. Real-Time Detection Under Controlled Stress Injection

The batch experiments above establish that the model separates normal from anomalous CPU behaviour, but they operate on data collected in advance. To evaluate whether the same approach functions as a live detector, the trained model was deployed in a continuous inference loop that queries Prometheus every 60 seconds, scores the most recent 30-timestep window, and compares the reconstruction error against the threshold learned at training time. The model is loaded once and is not retrained during operation, preserving its original notion of normal behaviour.

For this deployment the input signal was changed from the cumulative counter used in Section V-D to a rate-based signal, avg(rate(container_cpu_usage_seconds_total[1m])), averaged across the three application namespaces. A cumulative counter grows without bound, so live values quickly exceed the range seen during training; the rate form is bounded and remains comparable over time. The detector was deployed using an earlier model instance trained on 1,367 timesteps of this signal, prior to the extended collection reported in Section V-E; its operating threshold was 0.004172. Ratios reported below are expressed relative to that instance's threshold, so they describe the detector's behaviour in service rather than the absolute error values of the final model.

A controlled stress workload, a busy-loop process running in a dedicated pod, was injected into the Online Boutique namespace and removed approximately 17 minutes later. Table VIII summarises detector behaviour across the three phases of the test.

Table 8. Live Detection During Stress Injection

Phase	Readings	CPU (cores)	Flagged
Baseline	9	0.0041-0.0065	0 / 9
Stress injected	17	0.0212-0.0322	17 / 17
Post-removal	10	0.0046-0.0234	8 / 10

To summarise detection performance for this trial, the injection window (from 14:44 to removal at approximately 15:01 on 2026-09-06) was treated as ground-truth positive and every other reading in the same session as ground-truth negative. Table IX gives the resulting confusion matrix. All 17 readings inside the window were flagged (17 true positives, no false negatives), and 8 of the 19 readings outside it were flagged (8 false positives, 11 true negatives). This gives a precision of 0.68 (17/25), a recall of 1.00 (17/17), an F1-score of 0.81, and a false positive rate of 0.42 (8/19). All eight false positives occurred in the post-removal phase, consistent with the transition effect discussed below; none of the nine baseline readings was flagged. These figures are computed over a single controlled trial containing one injected fault and 36 readings, not over a comprehensively labeled dataset, and they describe this trial rather than providing a general estimate of detector accuracy.

Table 9. Confusion Matrix for the Live Stress Test

Actual	Flagged	Not flagged	Total
Stress window	17 (TP)	0 (FN)	17
Outside window	8 (FP)	11 (TN)	19

During the baseline phase the mean CPU rate remained near 0.005 cores and the reconstruction error stayed between 0.34x and 0.45x of the threshold, with no readings flagged. Injection produced an immediate step change: CPU rose approximately six-fold to 0.031 cores and the first elevated reading was flagged within one polling interval. All 17 readings taken while the workload was active exceeded the threshold, peaking at 11.79x during the transition.

Detector behaviour after removal is worth noting. Although CPU returned to its baseline within one interval, the reconstruction error rose further, peaking at 22.74x of the threshold roughly three minutes after removal, before returning below the threshold at 0.29x. This occurs because the 30-timestep input window spans 7.5 minutes and therefore still contained elevated values alongside the returning baseline; the model had not been trained on such an abrupt transition. The behaviour is arguably desirable, since a sudden collapse in workload is itself an event that merits attention, but it does

mean the detector signals a change of regime rather than a sustained high-load condition specifically.

This deployment addresses one of the directions previously identified as future work. It demonstrates that the trained model operates as a live detector without modification to its architecture, and that end-to-end latency from workload change to alert is bounded by the polling interval. It remains a single controlled trial on a single-node cluster, and the limitations noted in Section VI-B continue to apply.

VI. DISCUSSION

These experiments show that LSTM-based autoencoder models are effective for detecting anomalies from cloud-native microservices metrics via unsupervised machine learning methods. Our methodology runs on commodity hardware, trains in minutes, and uses tools widely deployed by organizations working with cloud native architectures like Prometheus and Kubernetes.

This analysis indicates that the approximately 27-fold difference in reconstruction-error scale between baseline and stress conditions represents a strong signal for CPU anomalies of this sort, even if the model was trained on 3,014 records using just a lightweight model of size 49,985 parameters.

Limitations: It is worth emphasizing that this study uses univariate metrics on a single-node cluster. There are many ways to complicate things further with multivariate, multi-node metric relationships in production. This particular anomaly was also designed to be deliberately obvious; subtle performance degradation could call for much longer sequence windows or ensembles. Three further limitations should be noted. First, while Section V-E characterizes sensitivity to random initialization over ten runs, sensitivity to workload and cluster state is not characterized: the primary result in Table IV comes from a single experimental run, and the variation observed across the multi-application runs confounds initialization effects with differences in the underlying cluster condition. Second, while Section V-F discusses static threshold-based alerting conceptually, no static-threshold detector was implemented and run against the same dataset for direct empirical comparison; the claimed advantages of the LSTM-based approach are therefore based on qualitative reasoning rather than a head-to-head benchmark. Third, the live deployment reported in Section V-G is a single controlled trial against one injected workload; the false-positive rate over extended operation and the detector's sensitivity to more subtle load changes were not characterized. Addressing these limitations through workload-varied trials, an implemented baseline comparison, and longer-running live evaluation is left for future work.

VII. CONCLUSION

This paper presented an AI-powered observability framework for cloud-native DevOps environments, integrating LSTM autoencoder-based anomaly detection with the Prometheus-Grafana observability stack on Kubernetes. The proposed system was implemented, evaluated, and validated through controlled anomaly injection experiments on a 12-microservice Kubernetes deployment.

Key results include: (1) successful training convergence of an LSTM autoencoder on real Kubernetes CPU metrics; (2) reconstruction error of a markedly larger scale under anomaly injection, with the mean plus two standard deviations, computed independently for each condition, approximately 27 times higher than at baseline; (3) anomaly detection without labeled training data; (4) a ten-run analysis establishing that the threshold (0.000676 +/- 0.000038) and anomaly rate (4.38% +/- 0.26%) are reproducible across random initializations; and (5) an end-to-end live deployment in which every reading taken during an injected workload was flagged. These results support the broader transition from reactive to proactive observability in cloud-native DevOps, advancing the AIOps maturity model proposed by Manchana[1].

Section V-G demonstrates a first step toward real-time operation, but that deployment routes alerts to a standalone dashboard rather than to Kubernetes-native tooling; integrating detections into AlertManager so they reach existing on-call workflows remains outstanding. Future work will also extend this framework to multivariate metrics (CPU, memory, network, latency) and to evaluation on large-scale production clusters. Two further directions follow directly from the limitations noted above: implementing a static-threshold detector for direct empirical comparison against the LSTM autoencoder on the same dataset, and repeating the experiments across varied workloads and cluster states, since Section V-E characterizes sensitivity to initialization but not to the underlying operating conditions.

Acknowledgment
The author thanks Dr. Rajesh Chauhan for his guidance and support throughout this research. This work was conducted as part of the M.Tech programme at the University Institute of Technology, Himachal Pradesh University, Shimla. The author acknowledges the use of Claude (Anthropic) for language editing, structural organization, and drafting assistance in the Introduction, Related Work, System Architecture,

Methodology, Results, Discussion, and Conclusion sections of this manuscript. The tool was used to help organize technical explanations, improve clarity and grammar, and assist in drafting text describing the author's own experimental work. All experimental design, implementation, data collection, and results reported in this paper are the author's original work; the AI system was not used to generate, fabricate, or alter any experimental data or results.

REFERENCES

1. R. Manchana, "AI-Powered Observability: A Journey from Reactive to Proactive, Predictive, and Automated," IJSR, vol. 13, no. 8, pp. 1745–1755, 2024. doi: 10.21275/SR24820054419
2. "From Reactive to Proactive: AI-Driven Observability in Cloud-Native DevOps," International Journal of Innovative Research in Science, Engineering and Technology (IJIRSET), e-ISSN: 2319-8753, p-ISSN: 2347-6710, 2025. Available: https://www.researchgate.net/publication/395242566
3. Cloud Native Computing Foundation, "CNCF Observability Whitepaper," 2023. Available: https://github.com/cncf/tag-observability
4. Prometheus Authors, "Prometheus Documentation," 2024. Available: https://prometheus.io/docs/
5. Grafana Labs, "Grafana Documentation," 2024. Available: https://grafana.com/docs/
6. Google LLC, "Online Boutique: Cloud-Native Microservices Demo," 2024. Available: https://github.com/GoogleCloudPlatform/microservices-demo
7. S. Hochreiter and J. Schmidhuber, "Long Short-Term Memory," Neural Computation, vol. 9, no. 8, pp. 1735–1780, 1997.
8. P. Malhotra et al., "LSTM-based Encoder-Decoder for Multi-sensor Anomaly Detection," ICML Workshop, 2016. Available: https://arxiv.org/abs/1607.00148
9. Weaveworks, "Sock Shop: A Microservices Demo Application," 2024. Available: https://github.com/microservices-demo/microservices-demo
10. S. Prodan, "Podinfo: Go Microservice Template," 2024. Available: https://github.com/stefanprodan/podinfo
11. Kubernetes Authors, "Kubernetes Documentation," 2024. Available: https://kubernetes.io/docs/
12. M. Abadi et al., "TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems," arXiv:1603.04467, 2016.
13. A. Leivadeas and M. Falkner, "Mitigating Alert Fatigue in Cloud Monitoring Systems: A Machine Learning Perspective," Computer Networks, vol. 250, 2024. doi: 10.1016/j.comnet.2024.110543
14. J. Chen et al., "MTG_CD: Multi-scale Learnable Transformation Graph for Fault Classification and Diagnosis in Microservices," Journal of Cloud Computing, 2024. doi: 10.1186/s13677-024-00666-0
15. M. Du, F. Li, G. Zheng, and V. Srikumar, "DeepLog: Anomaly Detection and Diagnosis from System Logs through Deep Learning," in Proc. ACM SIGSAC Conf. on Computer and Communications Security (CCS), 2017, pp. 1285-1298.