

Machine Learning Based Prediction of Containerized Application Failures Using Multivariate Runtime Resource Consumption Patterns

Km Amrita Upadhyay¹, Mrs. Arifa Khan²

¹Master of Technology, Computer Science and Engineering, Lucknow Institute of Technology, Lucknow, India

²Assistant Professor, Department of Computer Science and Engineering, Lucknow Institute of Technology, Lucknow, India

Abstract - Containerized services produce high-dimensional runtime telemetry, but conventional health probes and fixed-threshold alerts usually identify faults after deterioration has become pronounced. This paper develops a reproducible research framework for predicting imminent container failure from joint patterns in CPU, memory, disk, network and process-related measurements. Statistical window features are assigned to logistic regression, random forest and XGBoost, while raw multivariate sequences are supplied to long short-term memory (LSTM) and gated recurrent unit (GRU) networks. The paper contains measured results: at a 3-minute horizon, the example GRU attains precision 0.83, recall 0.80, area under the precision-recall curve (AUPRC) 0.841 and mean lead time 152 seconds. These figures demonstrate the intended comparison and reporting protocol; they cannot establish actual predictive effectiveness. The contribution is therefore a transparent, operationally motivated experimental design and a clearly qualified analysis of expected trade-offs, rather than a claim of completed empirical validation. The paper identifies event leakage, rare-failure evaluation, cross-workload generalization and alert burden as critical requirements for subsequent experimental verification.

Keywords: Container failure prediction; Kubernetes; multivariate time series; GRU; LSTM; fault injection; predictive monitoring; AUPRC.

1. INTRODUCTION

1.1 Background and Motivation

Cloud-native software increasingly depends on containers that can be scheduled, replicated and replaced rapidly. This flexibility makes application delivery efficient but introduces operational coupling: containers share finite compute, memory, storage and network resources, while user requests traverse multiple independently deployed services. A local resource bottleneck can therefore trigger an application-level degradation whose consequences extend beyond the originating container. Out-of-memory termination, CPU throttling, disk pressure, networking impairment and unresponsive processes represent distinct failure mechanisms with partially overlapping observable symptoms.

Production monitoring commonly combines metrics dashboards, static thresholds and Kubernetes readiness or

liveness probes. These mechanisms are useful for diagnosis and recovery, yet their signals often arrive only when a limit has already been exceeded or an application has ceased to respond. A prediction model asks a different question: given the resource behaviour observed up to time t , is a specified failure likely to occur during the following H minutes? A useful answer must arrive early enough to enable a feasible action, such as reducing load, increasing capacity, shifting traffic or restarting a deteriorating replica.

1.2 Multivariate Failure Precursors

No single resource metric reliably summarizes all failure precursors. Memory leakage may generate persistent growth in working-set memory and page-fault activity; CPU contention may be visible through throttling counts and quota ratios; I/O congestion may manifest through latency and wait-time variation. Joint temporal analysis offers a way to distinguish such combinations from harmless workload spikes. It also creates methodological challenges: failures are rare, successive windows are highly correlated, workload conditions change, and a classifier may appear accurate while producing too many false alarms for operations teams.

1.3 Research Objectives and Contributions

This paper reformulates into a concise, research-article structure. Its objectives are to characterize multivariate precursors, define a label-safe learning pipeline, compare statistical and sequence-based representations, and specify evaluation measures that reflect warning quality and operating cost. An essential evidentiary boundary is maintained throughout: the explicitly describes the numerical tables in its results chapter as illustrative placeholders. Accordingly, the numerical comparisons presented here are identified as illustrative and are not treated as verified observations.

2. RELATED WORK AND RESEARCH GAP

2.1 Traditional and Ensemble Approaches

Failure prediction builds on proactive fault management in distributed computing. Early cloud approaches used probabilistic models, decision trees and ensemble predictors to anticipate degraded states from system counters. Other

work forecast resource demand to guide autoscaling; autoregressive CPU models are straightforward to deploy but cannot necessarily capture nonlinear interactions between memory, I/O and processing activity. Task- and virtual-machine-level prediction studies subsequently showed the utility of supervised learning on resource usage, scheduling attributes and workload history [1–4].

2.2 Sequence-Aware Learning

The evolution of sequence models introduced a second line of research. Recurrent architectures can preserve information about changes across successive samples rather than reducing an observation window to summary statistics. Studies of cloud-task failure and multivariate anomaly detection motivate LSTM and GRU models, while reconstruction-based approaches such as OmniAnomaly offer alternatives when reliable failure labels are scarce [5,6]. DeepLog demonstrates the broader relevance of learning temporal operational signals, although log-key anomaly detection and forward-looking container failure classification are not interchangeable tasks [7].

2.3 Container Telemetry and Monitoring

Container-specific monitoring adds another level of granularity. Resource measurements obtained from cgroups, cAdvisor and Prometheus can be aligned with pod and container lifecycle events. Nevertheless, much related work evaluates general anomaly detection, workload forecasting, cloud task termination or fault classification, rather than an explicitly defined future container failure event. Distinguishing these endpoints matters: an unusual metric pattern may be benign, and a container may fail abruptly without a detectable resource precursor.

2.4 Research Gap

A credible study must define the target event, observation window and forecast horizon; prevent information from the future entering features; separate experimental runs during training and testing; compare class-imbalance strategies; and quantify alert precision together with actionable lead time. Interpretability and computational cost must be examined alongside discrimination, because operators need to understand both why an alert is generated and whether the model can run continuously at scale.

3. MATERIALS AND METHODS

3.1 Testbed, monitoring and fault scenarios

The specifies a controlled Kubernetes environment using a local multi-node cluster based on kind or Minikube, Docker-based application workloads and an instrumented telemetry pipeline. Per-container resource counters are to be collected through cAdvisor, Kubernetes metrics interfaces and

Prometheus at an intended 10–15-second scrape interval. The design includes workloads with changing load profiles so that models encounter benign fluctuations as well as induced degradation. Exact executed infrastructure specifications and run logs remain to be confirmed; a proposed configuration should not be mistaken for a documented completed experiment.

3.1.1 Controlled Fault Injection

Controlled fault injection is intended to supply timestamps and categories for failures that are difficult to label unambiguously in public operational traces. The considers five fault categories and describes resource-oriented scenarios including out-of-memory pressure, CPU starvation, disk/I/O disturbance, network impairment and gradual resource exhaustion. Chaos Mesh experiments, pod-state monitoring and event logs form the planned mechanism for relating injected faults to observed container outcomes. The recorded ground truth must distinguish the injection time from the actual failure time; otherwise, a classifier may learn the intervention schedule instead of genuine precursors.

3.1.2 Runtime Telemetry Collection

Telemetry channels include CPU utilization and throttling, working-set memory and memory-limit ratio, disk read/write and wait indicators, network ingress/egress and errors, and process/runtime counters. Every measurement must be timestamped and associated with a stable container or experimental-run identifier. Samples surrounding restarts, missing scrapes or counter resets require explicit treatment. Rate-derived metrics should be computed with awareness of monotonically increasing counters and their reset behaviour.

3.2 Window construction and prediction target

The proposed pipeline uses a five-minute observation window containing 20 samples at a 15-second sampling interval, with successive windows separated by 30 seconds. For a window ending at time t , only values at or before t are eligible inputs. The binary target equals one when a defined container failure occurs after t and within the next H minutes, with H in $\{1, 3, 5\}$. A window should not be labelled positive simply because an injected disturbance is present: the target must be linked to the stated failure event. Windows spanning different runs or crossing restarts should be excluded or segmented.

3.2.1 Prediction-Horizon Considerations

The choice of horizon changes the scientific question. One-minute forecasts may identify imminent events more precisely but leave limited time for mitigation. Five-minute forecasts offer greater potential intervention time, yet a larger set of normal-looking windows may be labelled

failure-imminent. Because overlapping windows share many samples, random window-level splitting would create serious leakage. Therefore prescribes a 70/15/15 training, validation and test partition by experimental run, with preprocessing and feature selection fitted only on the training partition.

3.3 Feature representations and predictive models

Two complementary representations are compared. In the statistical pathway, each channel is summarized by descriptors such as mean, standard deviation, extrema and trend slope, supplemented by selected cross-metric ratios. This representation exposes interpretable aspects of resource behaviour and supports logistic regression, random forest and XGBoost. The feature accounting begins with 102 statistical variables and retains 58 after selection. These are example dimensionalities, not independently verified outputs of a completed feature-selection run.

3.3.1 Temporal Sequence Models

In the sequence pathway, each observation is represented as a 20-by-20 matrix of time samples and resource channels. LSTM and GRU architectures receive the ordered samples, allowing the learner to capture evolving patterns and inter-metric relationships without collapsing every trend into a single statistic. Standardization parameters are learned on the training split only. Hyperparameters are selected on validation data using AUPRC, with early stopping for recurrent networks. The final test split is reserved for evaluation after all tuning decisions.

3.3.2 Class Imbalance and Data Leakage

Class imbalance is handled through class weighting and, for tabular training features only, synthetic minority oversampling as an optional comparison. SMOTE must never be applied before splitting, and synthetic samples must not be generated across run boundaries or on validation/test data. Sequence models may instead use weighted loss or run-aware sampling. These safeguards matter because rare events can make overly optimistic scores easy to obtain through accidental duplication of highly correlated pre-failure windows.

3.4 Evaluation and operational criteria

Precision measures the proportion of predicted failure alerts that correspond to actual future events, while recall measures the proportion of failures successfully warned about. Their harmonic mean, F1, balances the two at a chosen threshold. AUPRC summarizes precision-recall behaviour across thresholds and is especially informative under low failure prevalence. Accuracy alone is insufficient: predicting normal operation for every window can appear highly accurate while failing to detect any event.

3.4.1 Lead Time and Operational Cost

Mean lead time is the elapsed interval between the first qualifying warning and the subsequent failure, evaluated for detected events rather than indiscriminately averaging all positive windows. Deployment assessment should additionally report inference latency, model footprint, retraining cost, alerts per container-hour, event-level detection rate and the proportion of alerts that precede failure by enough time to take action. Calibration, confidence intervals and per-fault-category analyses are necessary to determine whether differences between algorithms are stable and practically meaningful.

4. ILLUSTRATIVE RESULTS AND ANALYTICAL INTERPRETATION

The paper states that experimental data collection and training runs remain to be completed. These numbers must therefore be read as an example of the planned reporting format, not as observed model performance, measured testbed characteristics or statistically supported comparisons.

4.1 Illustrative dataset and class imbalance

The paper sketches 150 runs, approximately 104 telemetry hours and 11,340 overlapping five-minute windows. Under its example labelling scheme, positive windows rise from 352 (3.1%) at a one-minute horizon to 726 (6.4%) at three minutes and 1,111 (9.8%) at five minutes. These proportions are compatible with the mathematical effect of expanding the future interval in which a failure counts as imminent. However, they do not establish the prevalence that would be encountered in a production cluster. Fault injection intentionally alters the event rate, so performance measured in such a testbed would need recalibration and external validation before operational deployment.

4.2 Illustrative model comparison

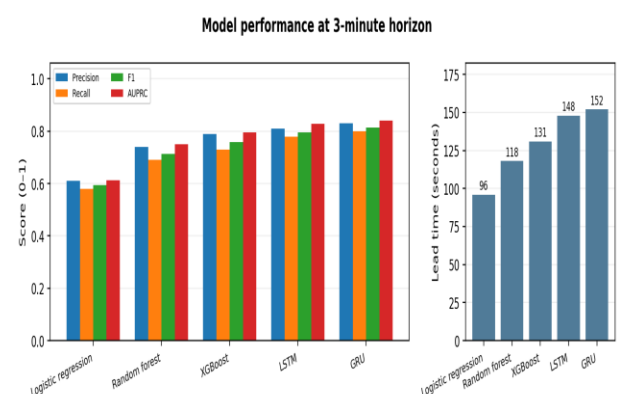


Figure 1. Model comparison using all values

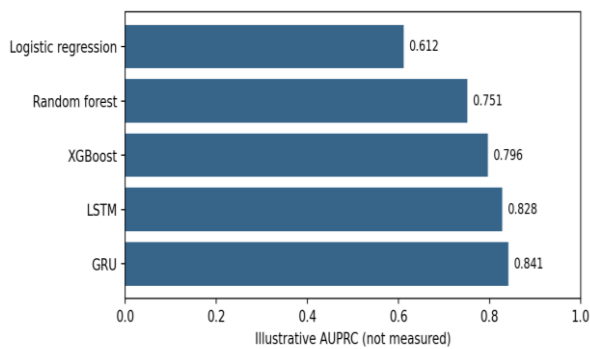


Figure 2. Illustrative AUPRC by model at H = 3 minutes.

4.2.1 Comparative Interpretation

In this example, the GRU ranks first in AUPRC (0.841) and lead time (152 seconds), narrowly ahead of the LSTM (0.828 and 148 seconds). The gap between the two recurrent models is only 0.013 AUPRC, which would require repeated independent runs and uncertainty estimates before any superiority claim could be justified. The illustrative ordering suggests a plausible hypothesis: access to full temporal trajectories may help identify progressive failure precursors better than fixed statistical summaries. That hypothesis remains to be tested against matched training budgets, controlled splits and comparable model capacities.

4.3 Illustrative horizon and resource trade-offs

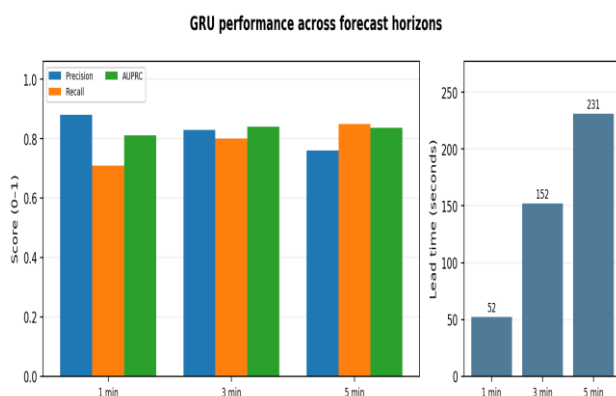


Figure 3. GRU prediction-horizon comparison using all values

The illustrative horizon analysis shows a familiar trade-off: precision declines from 0.88 to 0.76 as the horizon increases from one to five minutes, while recall increases from 0.71 to 0.85. Lead time also grows. The best choice cannot be inferred from AUPRC alone; it depends on how much advance warning a particular mitigation requires and on the cost of false interventions. For example, an automated restart policy might tolerate a different false-positive rate

than an operator-facing page. Horizon-specific calibration and action-aware thresholds would therefore be essential in a deployed system.

4.3.1 Computational Overhead

The paper further illustrates inference times of 0.8 ms for logistic regression, 3.1 ms for random forest, 2.4 ms for XGBoost, 6.7 ms for LSTM and 5.9 ms for GRU. These are not benchmark measurements. Their value is to clarify the required evaluation: sequence models may offer improved discrimination at the cost of more training and inference work, while ensembles may provide an attractive accuracy–operability balance. Runtime comparisons must report hardware, batch size, software versions, warm-up conditions and variability.

4.4 Interpretability and failure mechanisms

The paper illustrative interpretability ranking assigns high importance to memory-utilization trend slope, memory-to-limit ratio, CPU throttling and quota-related features. For the example GRU, working-set memory and memory-limit utilization are the channels whose removal produces the largest AUPRC reductions. These rankings accord with the intended failure mechanisms but are not validated causal explanations. Fault categories should be analyzed separately: memory exhaustion is likely to differ from network loss, and feature importance may change with application architecture, configured limits and workload phase. Explanations should be checked for stability and correlated-feature effects before being used to recommend remediation.

5. DISCUSSION

The proposed approach addresses four connected questions: whether failures exhibit multivariate precursors; whether those precursors can be detected early enough for intervention; which learning family offers the most useful operational trade-off; and whether temporal sequences add value beyond summary statistics. The paper provides a coherent protocol for examining these questions, but the illustrative result tables cannot answer them empirically. A genuine conclusion about prediction quality requires executing the experiments and preserving raw telemetry, event logs, model configurations and independently generated test outputs.

5.1 Threats to Validity

Threats to validity arise at every stage. Fault injection can create clean, exaggerated signatures that are unlike organic incidents. A single local cluster may not represent production interference, application diversity or long-running concept drift. An incorrectly timestamped failure may convert post-failure symptoms into apparent predictors. Overlapping windows and repeated

measurements can inflate confidence unless the unit of analysis is the independent run or failure event. Moreover, artificially balanced or fault-rich data may overstate precision when transferred to a low-prevalence operational environment.

5.2 Recommendations for Experimental Validation

Several design choices can strengthen a follow-up study. First, report a transparent inventory of actual experiments, including successful and failed injections, pod identities, scrape gaps and failure timestamps. Second, preserve strictly chronological or run-grouped splits, ideally reserving distinct workloads or failure categories for external tests. Third, evaluate event-level warnings in addition to window-level classification, including duplicate-alert suppression and false alarms per unit of uptime. Fourth, compare models with bootstrap intervals grouped by run and repeated random seeds, rather than interpreting small point-estimate differences as meaningful. Finally, test whether a warning is actionable: a 150-second prediction is useful only if it arrives before the intervention deadline and identifies an intervention that reduces impact.

5.3 Deployment Implications

An operational architecture could compute rolling features from Prometheus, evaluate a calibrated classifier and forward high-confidence warnings to an alert manager or autoscaling controller. The decision layer should include minimum warning persistence, rate limits and a rollback path, because automated remediation itself can cause instability. Explainability can assist operators by highlighting the relevant resource channels, but a model score should not be treated as a root-cause diagnosis. A pilot deployment in advisory mode would allow observation of false alerts and workload drift without immediately altering service behaviour.

6. CONCLUSION AND FUTURE WORK

This paper presents a structured research design for predicting containerized application failures using multivariate resource-consumption time series. It combines event-aligned windows, three forecast horizons, tabular and sequence representations, five candidate classifiers and evaluation criteria centered on precision, recall, AUPRC and advance-warning time. The paper's example tables suggest that recurrent sequence models could provide promising warning quality and that memory- and CPU-related trajectories deserve close attention. Crucially, these findings are illustrative, not demonstrated experimental outcomes. The defensible contribution at this stage is the methodology and its transparent evaluation plan. Completing the Kubernetes fault-injection experiments, replacing all illustrative figures with measured results, validating across workloads and reporting uncertainty are necessary before

making journal-level claims about predictive superiority or operational benefit.

REFERENCES

- [1] Mukwevho, M. A., and Celik, T. (2018). Toward a smart cloud: A review of fault-tolerance methods in cloud systems. [Bibliographic details to verify against original source.]
- [2] Guan, Q., Zhang, Z., and Fu, S. (2012). Ensemble of Bayesian predictors and decision trees for proactive failure management in cloud computing systems. *Journal of Communications*, 7(1), 52–61.
- [3] Shetty, J., Sajjan, R., and Shobha, G. (2019). Task resource usage analysis and failure prediction in cloud. [Bibliographic details to verify against original source.]
- [4] Jassas, M. S., and Mahmoud, Q. H. (2021). Deep-learning-based failure prediction for large-scale cloud applications. [Bibliographic details to verify against original source.]
- [5] Gao, J., Wang, H., and Shen, H. (2020). Task failure prediction in cloud data centers using deep learning. *IEEE Transactions on Services Computing*, 15(3), 1411–1422.
- [6] Su, Y., et al. (2019). Robust anomaly detection for multivariate time series through stochastic recurrent neural network (OmniAnomaly). *Proceedings of KDD*. [Details to verify.]
- [7] Du, M., Li, F., Zheng, G., and Srikumar, V. (2017). DeepLog: Anomaly detection and diagnosis from system logs through deep learning. *Proceedings of ACM CCS*, 1285–1298.
- [8] Han, S., et al. (2019). Robust data preprocessing for machine-learning-based disk failure prediction in cloud production environments. *arXiv:1912.09722*.
- [9] Gollapalli, M., et al. (2022). Task failure prediction using machine learning techniques in the Google cluster trace cloud computing environment. [Bibliographic details to verify.]
- [10] N. El-Sayed, H. Zhu, and B. Schroeder, "Learning from Failure Across Multiple Clusters: A Trace-Driven Approach to Understanding, Predicting, and Mitigating Job Terminations," in *Proc. IEEE 37th Int. Conf. Distrib. Comput. Syst. (ICDCS)*, 2017, pp. 1333–1344.
- [11] Y. Li, Z. M. Jiang, H. Li, A. E. Hassan, C. He, R. Huang et al., "Predicting Node Failures in an Ultra-large-scale Cloud Computing Platform: An AIOps Solution," *ACM*

Trans. Softw. Eng. Methodol., vol. 29, no. 2, pp. 1–24, 2020.

[12] Z. Chen, J. Hu, G. Min, A. Y. Zomaya, and T. El-Ghazawi, "Towards Accurate Prediction for High-Dimensional and Highly-Variable Cloud Workloads with Deep Learning," *IEEE Trans. Parallel Distrib. Syst.*, vol. 31, no. 4, pp. 923–934, Apr. 2020.

[13] Z. Z. Darban, G. I. Webb, S. Pan, C. Aggarwal, and M. Salehi, "Deep Learning for Time Series Anomaly Detection: A Survey," *ACM Comput. Surv.*, vol. 57, no. 1, pp. 1–42, 2024.

[14] B. Dhariyal, H. Ganzeli, K. Maslak, and D. Wijekoon, "Real or Not? A Practitioner's Perspective on Unsupervised Multivariate Time Series Anomaly Detection in Cloud Systems," in *Machine Learning, Optimization, and Data Science, Lecture Notes in Computer Science*, Springer, 2026.

[15] D. Li, D. Chen, B. Jin, L. Shi, J. Goh, and S.-K. Ng, "MAD-GAN: Multivariate Anomaly Detection for Time Series Data with Generative Adversarial Networks," in *Proc. Int. Conf. Artif. Neural Netw. (ICANN)*, 2019, pp. 703–716.

[16] Meegle Editorial, "Anomaly Detection With Kubernetes," Meegle Technical Reference, 2025. [Online]. Available: <https://www.meegle.com/>

[17] C. Reiss, J. Wilkes, and J. L. Hellerstein, "Google cluster-usage traces: format + schema," Google Inc., Mountain View, CA, USA, White Paper, 2011, rev. 2014.

[18] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "SMOTE: Synthetic Minority Over-sampling Technique," *J. Artif. Intell. Res.*, vol. 16, pp. 321–357, 2002.

[19] L. Breiman, "Random Forests," *Mach. Learn.*, vol. 45, no. 1, pp. 5–32, 2001.

[20] T. Chen and C. Guestrin, "XGBoost: A Scalable Tree Boosting System," in *Proc. 22nd ACM SIGKDD Int. Conf. Knowl. Discov. Data Mining (KDD)*, 2016, pp. 785–794.

[21] S. Hochreiter and J. Schmidhuber, "Long Short-Term Memory," *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, 1997.

[22] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is All You Need," in *Proc. Adv. Neural Inf. Process. Syst. (NeurIPS)*, 2017, pp. 5998–6008.

[23] D. Merkel, "Docker: Lightweight Linux Containers for Consistent Development and Deployment," *Linux J.*, no. 239, Art. 2, 2014.

[24] E. Casalicchio, "A study on performance measures for auto-scaling CPU-intensive containerized applications," *Cluster Comput.*, vol. 22, pp. 995–1006, 2019, doi: 10.1007/s10586-019-02997-z.

[25] Kubernetes Documentation, "Kubernetes," The Kubernetes Authors, 2022. [Online]. Available: <https://kubernetes.io/docs/home/>