

Machine-Learning-Based Detection and Mitigation of CPU Fragmentation in Virtualized Environments

Abdulrazaq Mamud
am47969@georgiasouthern.edu
Georgia Southern University
Statesboro, GA, USA

Hayden Wimmer
hwimmer@georgiasouthern.edu
Georgia Southern University
Statesboro, GA, USA

Carl Rebman Jr.
carlr@sandiego.edu
University of San Diego
San Diego, CA, 92110, USA

ABSTRACT

This study examines CPU resource fragmentation in a controlled virtual machine hosting latency-critical microservices. CPU fragmentation is treated as an experimentally induced resource-allocation condition; the 100-ms p95 threshold defines an SLA-degraded performance state rather than an independent measure of fragmentation. A Random Forest classifier was used as a proof-of-concept component within a closed-loop controller that dynamically adjusted Docker cpuset allocations. The original evaluation reported 96.40% accuracy and an AUC of 0.9906 on 1,501 test observations; because latency-derived predictors overlap with the p95-based target, these values likely overstate independent fragmentation-detection performance. The stronger empirical result is the mitigation response: median CPU imbalance decreased from approximately 33 to 8, while p95 latency narrowed from approximately 95-130 ms before intervention to approximately 85-100 ms afterward. These results support the feasibility of fine-grained cpuset reallocation within the evaluated environment, while leakage-resistant, run-level validation is required before broader generalization.

Keywords: CPU fragmentation, virtualized environments, machine learning, CPU affinity, microservices, resource management.

Machine-Learning-Based Detection and Mitigation of CPU Fragmentation in Virtualized Environments

Abdulrazaq Mamud, Hayden Wimmer and Carl Redman Jr.

1. INTRODUCTION

Modern web-application performance depends on both application logic and the virtualized infrastructure that hosts it. Even when aggregate CPU utilization is moderate, uneven workload distribution across virtual CPUs (vCPUs) can create localized pressure, increase tail latency, and produce recurring Service Level Agreement (SLA) violations. Conventional dashboards can therefore show available capacity while users experience degraded responsiveness.

This study distinguishes four related concepts. CPU fragmentation denotes an experimentally induced condition in which a workload has uneven effective access to the CPU resources assigned to the virtual machine. Scheduling imbalance describes uneven execution across those resources; vCPU contention is competition among runnable workloads for CPU time; and CPU imbalance is the observed dispersion of per-core utilization, measured here as its standard deviation. The 100-ms p95 threshold denotes an SLA-degraded application state, not CPU fragmentation itself.

This problem is relevant to latency-sensitive information systems because infrastructure decisions affect application performance, SLA compliance, capacity management, and resource efficiency. Load balancing, horizontal scaling, and container orchestration distribute work among services or hosts, but they do not necessarily correct uneven CPU access within an individual VM.

The study evaluates a machine-learning-assisted mechanism for recognizing and mitigating an experimentally induced degraded operating state. Two CPU-bound Flask services behind an HAProxy load balancer were deployed in a self-hosted VMware VM. Docker's cpuset-cpus mechanism created constrained and expanded affinity conditions, and runtime telemetry supported a Random Forest classifier and dynamic CPU-core reassignment. Unlike conventional autoscaling, which provisions additional containers or VMs, the proposed micro-autoscaling controller adjusts CPU affinity within one VM. It operates through an observe-predict-scale-validate-revert loop: expand CPU allocations when a degraded state is detected, verify recovery, and gradually release excess cores after performance stabilizes.

The contribution is the integration of lightweight telemetry, proof-of-concept state classification, and closed-loop intra-VM cpuset adjustment. The mitigation mechanism, rather than classifier accuracy alone, is the principal empirical contribution.

Two research questions guide the study. They are as follows:

RQ1: *Within the controlled testbed, how effectively does the Random Forest reproduce the constructed SLA-degraded operating state from runtime telemetry?*

RQ2: *After such a state is recognized, to what extent does dynamic cpuset reallocation reduce CPU imbalance and restore p95 latency toward the experimental SLA threshold?*

From an Information Systems perspective, the study treats CPU allocation as an operational decision problem linking system telemetry and infrastructure controls to observable service-level outcomes. The results are interpreted as feasibility evidence for the evaluated environment rather than evidence of production effectiveness.

2. RELATED WORK

Virtualization and containerization shape microservice performance through both available capacity and resource mapping. Potdar et al. (2020) found that Docker can approach bare-metal performance with

lower overhead than traditional VMs, while Shah et al. (2021) showed that vCPU configuration, isolation, and core mapping can materially affect otherwise similar workloads.

CPU-affinity research likewise shows that pinning decisions affect cache locality, context switching, and performance predictability (GhatrehSamani et al., 2020; Podzimek et al., 2015). At broader infrastructure levels, fragmentation and VM-placement studies examine how uneven or stranded resources reduce utilization and efficiency (Mishra & Bellur, 2016; Ji et al., 2018; Miao & Chen, 2015).

Machine-learning-based resource controllers have been applied to horizontal and vertical scaling, latency prediction, quota management, and proactive allocation (Cano et al., 2018; Wajahat et al., 2019; Xu et al., 2022; Rahman & Lama, 2019; Choi et al., 2021; Cai et al., 2023; Wang et al., 2024; Sachidananda & Sivaraman, 2024). These systems operate at different control levels and use different signals; Table 1 compares them with the present study's closed-loop, intra-VM cpuset reassignment.

The reviewed literature establishes the importance of CPU topology, affinity, placement, and adaptive resource management. The present study addresses the narrower design space of using application and per-vCPU telemetry to trigger CPU-affinity changes among already-running containers inside one VM. It complements rather than replaces VM-level provisioning and application-level autoscaling.

Study	Management level	Primary telemetry or method	Mitigation or decision	Intra-VM cpuset reassignment
GhatrehSamani et al. (2020)	Host/container affinity	CPU pinning and performance	Pinning-strategy evaluation	No closed loop
Mishra & Bellur (2016); Ji et al. (2018)	Cloud infrastructure	Resource placement and fragmentation	Placement or defragmentation	No
Rahman & Lama (2019); Cai et al. (2023)	Microservice/service	Tail-latency prediction	Provisioning or scaling	No
Choi et al. (2021); Xu et al. (2022)	Service/cluster	Predictive or reinforcement learning	Autoscaling	No
Wang et al. (2024); Sachidananda & Sivaraman (2024)	Service/cluster	Application-aware SLO signals	Quota/resource management	No
Present study	Containers within one VM	Latency plus per-vCPU utilization	Closed-loop cpuset expansion/contraction	Yes

Table 1. Comparison of CPU Resource Management Approaches

Prior studies span CPU pinning, VM/vCPU allocation, and application autoscaling. The present approach differs in control granularity: it uses latency and per-vCPU utilization within one VM and selectively reassigns container CPU affinity. This comparison defines the study's contribution without claiming superiority over prior systems.

3. METHODOLOGY

The methodology combines controlled load testing, high-resolution telemetry, proof-of-concept classification, and automated runtime adaptation. CPU fragmentation was induced through cpuset constraints, while the 100-ms p95 threshold identified the application-level SLA state used in the original classifier evaluation. The workflow included testbed construction, data collection, Random Forest

classification, dynamic cpuset control, and mitigation assessment under varied workloads and CPU assignments.

3.1 Conceptual Framework

The framework separates the manipulated resource condition from its observed consequences. Constrained cpuset assignments can produce uneven effective access to vCPUs; CPU imbalance indicates uneven utilization; contention can develop on heavily used vCPUs; and p95 latency can then exceed the SLA threshold. The experiment evaluates whether a closed-loop controller can respond to that degraded state.

Four elements shaped the design:

- **Telemetry-driven visibility:** p50 and p95 latency, total CPU utilization, per-vCPU utilization, and CPU imbalance provide complementary performance and resource views.
- **State classification:** a Random Forest combines latency and CPU telemetry to recognize the constructed SLA state. Because the target and several predictors share latency information, the model is a proof-of-concept state classifier rather than an independent fragmentation detector.
- **Direct CPU-core reallocation:** Docker's cpuset-cpus interface changes Linux CPU affinity for running containers without provisioning another VM or restarting the application.
- **Closed-loop control:** the controller repeatedly observes telemetry, predicts state, expands or contracts cpusets, and verifies recovery; hysteresis limits oscillation.

3.2 Experimental Environment

Experiments were conducted on one workstation with an AMD Ryzen 7 6800H processor (8 physical cores and 16 logical threads) running Windows 11 Pro 10.0.26200 (Build 26200). VMware Workstation 17 Pro 17.6.0 (build 24238078) hosted one Ubuntu 22.04 LTS VM with Linux kernel 7.0.0-31-generic, 6 vCPUs, and 8 GB RAM. Docker Engine v29, Python 3.12.3, and scikit-learn 1.8.0 supported the services, workload, and Random Forest implementation.

SMT was disabled, no other user applications were running, and VMware CPU scheduling and frequency management remained at default settings. No explicit host-level pinning or physical-core isolation was used. Six vCPUs provided sufficient capacity for two CPU-bound services and HAProxy while retaining constraints for controlled affinity experiments; this was a testbed choice, not a general fragmentation threshold. The testbed contained two CPU-bound Flask services (frag_web1 and frag_web2) behind an HAProxy round-robin load balancer. Each service performed approximately 50 ms of CPU work per request. This synthetic design exposed scheduling effects while limiting I/O and network variation. Figure 1 illustrates the architecture. The self-hosted environment provided administrative control over guest and container CPU affinity. It does not represent a constrained public-cloud tenant that lacks control over host-level affinity or physical CPU placement.

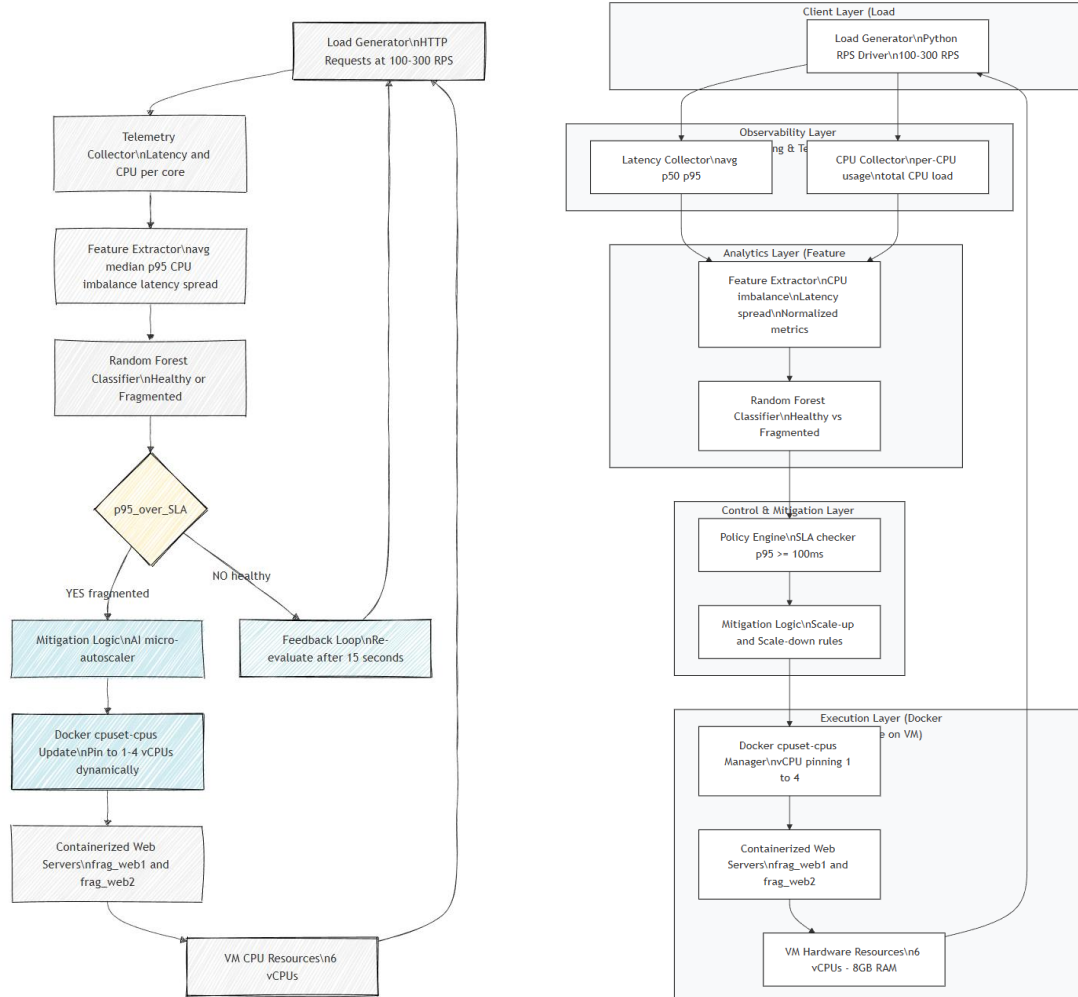


Figure 1: Overall System Architecture

Containers were assigned to specific vCPU subsets (for example, 0, 1, or 0-3) to create constrained and expanded affinity conditions. An asynchronous Python generator used concurrency 20 at target rates of 100, 150, 200, 250, and 300 requests per second for 30 seconds per run. Average, median, and p95 latency, total CPU utilization, and per-vCPU utilization were recorded.

Data collection produced approximately 5,000 one-second observations across repeated sessions with varied workloads, 1-4 assigned vCPUs, constrained and expanded cpusets, recovery, scale-down, and near-saturation conditions. These rows are temporally related measurements rather than 5,000 independent replications. The exact number of independent sessions was not separately retained. Each observation included average latency (lat_avg_ms), median latency (lat_p50_ms), p95 latency (lat_p95_ms), total CPU utilization, and per-core utilization. CPU imbalance was the standard deviation of per-core utilization; latency spread was p95 minus average latency.

The dataset included SLA-compliant operation, SLA-degraded operation, recovery after scale-up, scale-down transitions, high-load conditions, and transient cpuset changes. CPU fragmentation was induced through the managed cpuset condition. For the original classifier, however, the operational class label incorporated the 100-ms p95 SLA boundary. The resource condition and application outcome are conceptually distinct, but the original label and latency-derived predictors are not statistically independent.

For the operational classification used in this study, fragmentation labels were assigned according to a p95 latency threshold of 100 ms. Observations with $p95 \geq 100$ ms were labeled fragmented; those

below the threshold were labeled healthy. The 95th percentile is widely adopted as an SLA boundary because it captures scheduler-induced queuing delays more effectively than the median (p50) while remaining more stable than p99 (Choi et al., 2021; Cai et al., 2023). Preliminary experiments consistently showed performance degradation once p95 exceeded this value. The 100 ms threshold therefore defines the operational boundary of the performance-state classification and does not imply that latency itself constitutes the underlying CPU resource condition. Observations with elevated CPU imbalance but p95 below 100 ms were accordingly not labeled fragmented, although they may represent precursor resource-imbalance states.

The 100-ms threshold was selected during preliminary testing as a consistent experimental SLA boundary, not as a universal production SLA. Its selection may introduce threshold-selection bias, and the observed latency and imbalance ranges may depend on processor topology, frequency scaling, workload, and application requirements. Data checks addressed missing values, per-core formatting, outliers, correlations, repeated-run distributions, and representative intervals. These checks support internal consistency but do not eliminate target-feature dependency or temporal dependence.

3.3 Machine Learning Model Development

The original Random Forest used average latency, median latency, CPU imbalance, and latency spread. The archived evaluation reports a 70/30 train-test split and 1,501 test observations (569 SLA-compliant and 932 SLA-degraded). Accuracy, precision, recall, F1, confusion-matrix, ROC-AUC, calibration, and precision-recall measures were calculated.

Because latency spread algebraically contains p95 and the target incorporates the p95 threshold, the reported performance cannot establish that the model independently learned CPU fragmentation. It demonstrates discrimination of the constructed SLA state within the experimental dataset. A leakage-resistant evaluation would define ground truth independently from latency and compare CPU-only, latency-only, and combined feature sets.

The randomized split also permits temporally adjacent samples from the same run to appear in both partitions. Reported metrics may therefore be optimistic relative to experiment-level or time-separated validation and are not evidence of deployment-time generalization. During each control cycle, the system generated workload, collected telemetry, classified the operating state, adjusted cpuset allocations when triggered, verified SLA recovery, and gradually removed excess cores after several compliant cycles.

Scaling was synchronized across both services, and hysteresis limited rapid reversals. The present study does not compare this controller with no intervention, fixed cpusets, a p95-only rule, or a CPU-imbalance rule; it therefore evaluates feasibility of the integrated mechanism rather than incremental benefit attributable to machine learning.

4. RESULTS

The results document two linked components: the proof-of-concept classifier and the runtime cpuset controller. Classifier metrics are retained with explicit qualifications because the target shares information with latency-derived predictors. The before-and-after mitigation response provides the more direct evidence concerning cpuset reallocation.

4.1 Classification Performance

The Random Forest closely reproduced the constructed SLA classes using average latency, median latency, latency spread, and CPU imbalance. Table 2 reports the archived evaluation for 1,501 temporally related test observations.

Class	Precision	Recall	F1-Score	Support
SLA-compliant (0)	0.94	0.97	0.95	569
SLA-degraded (1)	0.98	0.96	0.97	932
Accuracy	-	-	0.96	1501
Macro Average	0.96	0.97	0.96	1501
Weighted Average	0.97	0.96	0.96	1501

Table 2. Classification Performance

As shown in Table 2, F1-scores were 0.9534 for the SLA-compliant class and 0.9707 for the SLA-degraded class. These values describe the original evaluation procedure and do not demonstrate independent fragmentation detection or generalization to unseen runs.

4.2 Confusion Matrix

Figure 2 reports 16 SLA-compliant observations classified as degraded and 37 degraded observations classified as compliant. These counts show close agreement with the constructed p95-based label within the archived test set.

4.3 ROC Curve and AUC Performance

The original evaluation produced an ROC-AUC of 0.9906. Target-feature overlap likely inflates this result, so Figure 3 documents the implemented proof of concept rather than leakage-free generalization.

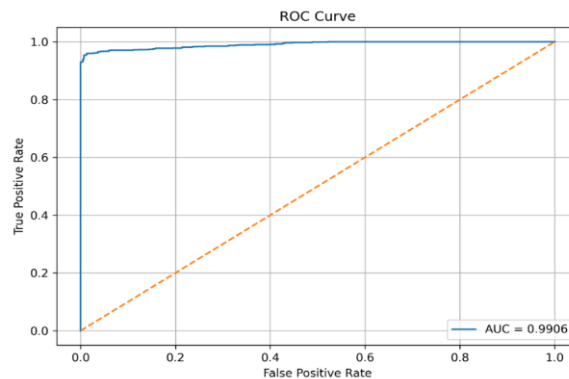


Figure 3: ROC Curve

4.4 Analysis of Misclassification Patterns

The 53 errors occurred near the boundary between the constructed SLA classes. Observations below 100 ms remain SLA-compliant under this labeling scheme even when CPU imbalance is elevated; they are not relabeled as fragmented. Without retained run identifiers, errors cannot be reliably stratified by transition, workload, or cpuset configuration.

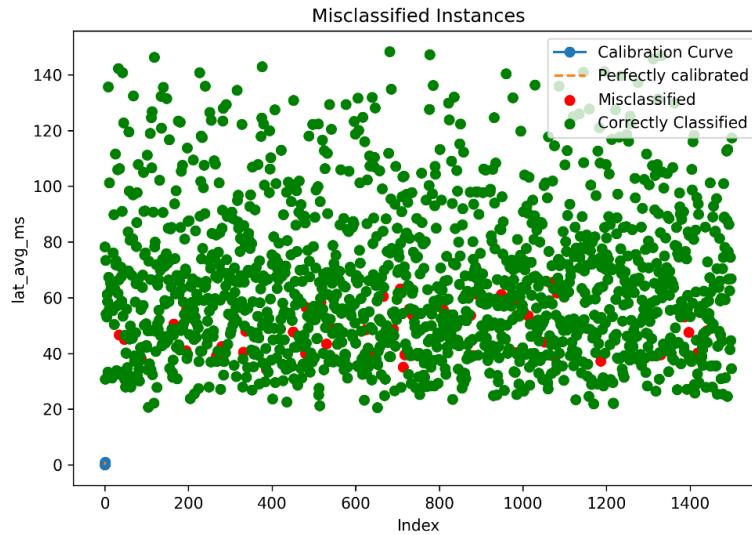


Figure 4. Classification Results: Correctly Classified and Misclassified Observations

4.5 Feature Importance and Model Interpretability

Average latency, latency spread, and median latency dominated feature importance, while CPU imbalance contributed less. This pattern is consistent with reconstruction of a latency-defined target and does not establish detection from CPU behavior alone. CPU-only, latency-only, and combined-feature ablation remains necessary.

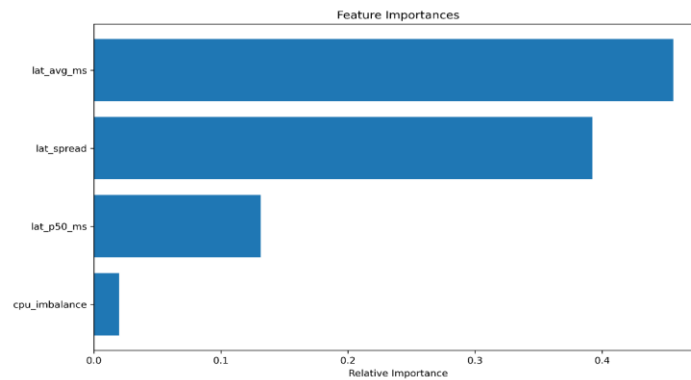


Figure 5. Feature Importance

4.6 Calibration Analysis

Figure 6 documents probability calibration against the constructed SLA label. Calibration does not resolve the target-feature dependency or temporal dependence and is interpreted only as a characteristic of the original implementation.

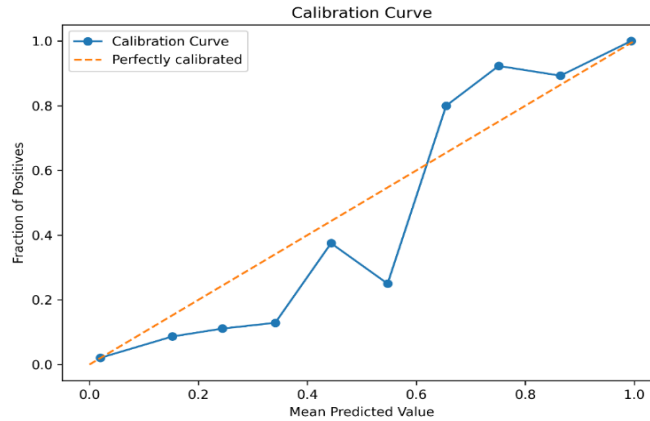


Figure 6. Calibration Analysis

4.7 Precision-Recall Analysis

The SLA-degraded class is the majority class (932 of 1,501 observations). The precision-recall curve provides another descriptive view of agreement with the constructed label, subject to the same limitations.

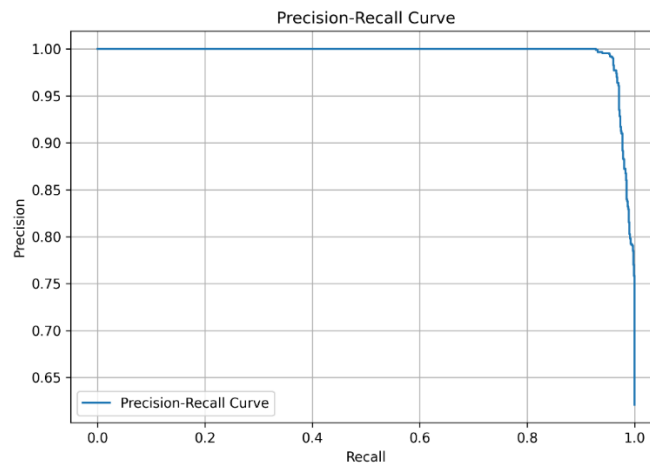


Figure 7. Precision-Recall Curve

4.8 Mitigation Response

When constrained CPU affinity and increasing workload pushed p95 beyond 100 ms, the integrated controller expanded Docker cpuset allocations. Work was then distributed across additional vCPUs, and both CPU imbalance and tail latency declined. Without a rule-based baseline, this result supports feasibility of closed-loop cpuset mitigation but not superiority of the Random Forest trigger. After reallocation, median CPU imbalance decreased from approximately 33 to 8, and the observed p95 range narrowed from approximately 95-130 ms to 85-100 ms. Fewer SLA violations and a narrower latency profile were observed. Scale-up actions were followed by stabilization intervals, while downscaling required several consecutive compliant cycles. No persistent oscillation appeared in the reported trace. Repeated-run analysis is needed to determine how consistently this behavior generalizes.

5. CPU IMBALANCE BEFORE AND AFTER MITIGATION

CPU imbalance was calculated as the standard deviation of utilization across available vCPUs at each sampling interval. Elevated imbalance is an observable indicator associated with the induced cpuset condition, not a definitive signature of fragmentation. Before mitigation, median CPU imbalance was approximately 33, with reported values generally spanning 25-40. After cpuset expansion, the median was approximately 8 and the range narrowed to about 5-12. These descriptive changes are consistent with more even resource distribution after intervention. Because the archived analysis does not provide independent-run sample counts, confidence intervals, effect sizes, or a formal statistical comparison, the result is evidence from the evaluated trace rather than definitive causal confirmation. Figure 8 presents the comparison.

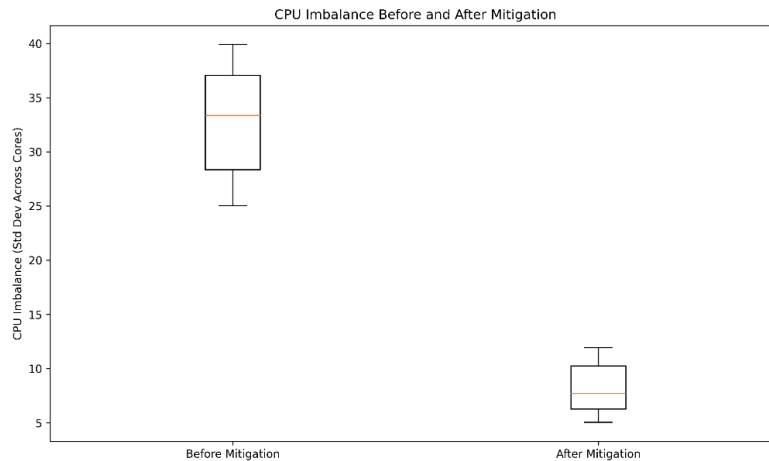


Figure 8. CPU imbalance before and after mitigation

5.1 p95 Latency Before and After Mitigation

Before intervention, p95 latency ranged approximately from 95 to 130 ms and frequently crossed the 100-ms experimental SLA threshold. After intervention, it narrowed to approximately 85-100 ms with fewer threshold violations. The concurrent reduction in CPU imbalance and p95 latency is consistent with improved scheduling balance following cpuset reallocation.

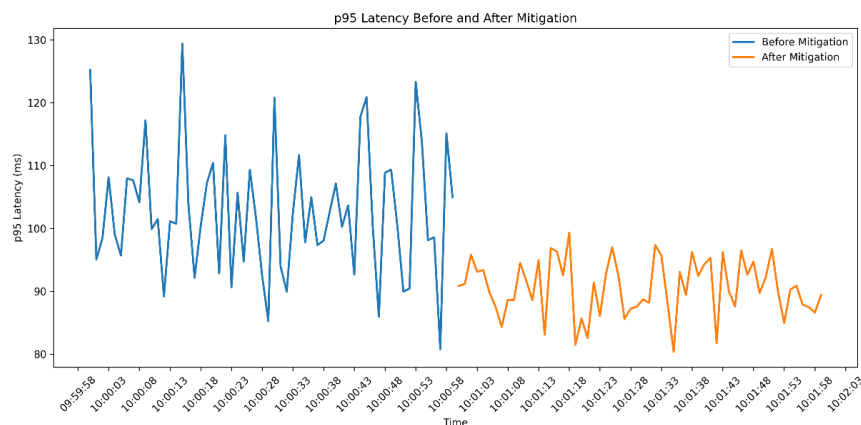


Figure 9. p95 Latency Before and After Mitigation

The time series shows a shift from a higher, more variable latency range to a lower, narrower range after intervention. This is descriptive evidence of improved service-level performance in the evaluated trace.

5.2 p95 Latency During Control Cycles

Figure 10 shows p95 latency across the reported control cycles. During the constrained phase, latency ranged approximately from 95 to 110 ms with intermittent SLA violations. As CPU allocation expanded, variation decreased; after stabilization, p95 remained largely between 85 and 95 ms. No persistent oscillation is visible in this trace.

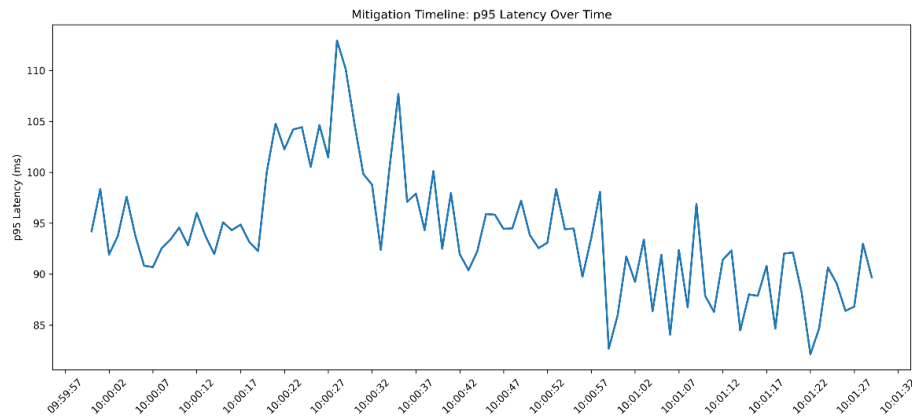


Figure 10. Mitigation Timeline: p95 Latency Over Time

6. DISCUSSION

The study provides feasibility evidence for a closed-loop, intra-VM mitigation design that links lightweight telemetry to dynamic cpuset reallocation. The classifier closely reproduced the constructed SLA state, but target-feature dependency and temporally dependent evaluation prevent the metrics from establishing independent fragmentation detection. The clearer contribution is the observed mitigation response: CPU imbalance and p95 latency declined after cpuset expansion.

6.1 Summary of Research Findings

Three findings emerge within the study's limits. First, constrained CPU affinity coincided with uneven per-core utilization and elevated tail latency. Second, the Random Forest closely reproduced the p95-defined SLA state, although latency-derived features make that result partly circular. Third, dynamic cpuset reallocation was followed by a decrease in median CPU imbalance from approximately 33 to 8 and a narrowing of p95 latency from approximately 95-130 ms to 85-100 ms. The reported 96.40% accuracy and 0.9906 ROC-AUC describe the original controlled implementation only. Leakage-resistant validation requires an independently defined target, complete experimental runs held out from training, uncertainty estimates across runs, and comparison with simple rule-based baselines.

The mitigation trace indicates that CPU-core reallocation can correct the evaluated imbalance condition without adding containers or VMs. Hysteresis also coincided with stable behavior in the reported trace. These observations establish feasibility within one synthetic, self-hosted environment rather than production robustness.

6.2 Systems-Design Implications

The results highlight a distinction between aggregate capacity and access to that capacity. Average CPU utilization can miss localized congestion, while joint examination of per-core utilization and tail latency can reveal operational states that warrant intervention. This finding motivates closer integration between application-performance monitoring and guest-level resource controls.

6.3 Operational Implications

The framework uses user-space telemetry and guest-level Docker controls rather than hypervisor modification. Where administrators possess cpuset authority, it may complement horizontal scaling through fine-grained adjustment inside an existing VM. Applicability to managed public-cloud tenants, mixed workloads, and larger production deployments remain to be demonstrated.

7. LIMITATIONS AND FUTURE WORK

The study used one physical laptop-class host, one self-hosted VMware VM, and synthetic CPU-bound Flask services. Processor topology, default frequency management, the absence of host-level isolation, and the experimental 100-ms threshold may affect the observed ranges. The workload excludes databases, caching, I/O, asynchronous operations, and network variability and does not represent a constrained public-cloud tenant.

The mitigation analysis is descriptive and lacks independent-run sample counts, confidence intervals, effect sizes, and matched controller baselines. It establishes feasibility of the observed cpuset response, not incremental value over no intervention, fixed allocation, a p95-only rule, or a CPU-imbalance rule. The classifier target incorporates p95 latency while latency-derived variables are predictors, and the randomized split does not separate complete experimental runs. These choices likely inflate classification performance.

Future work should define ground truth from imposed scheduling or cpuset conditions independently of latency; hold out complete runs, workloads, or configurations; compare CPU-only, latency-only, and combined features; benchmark non-ML controllers; report repeated-run uncertainty; and evaluate mixed services, server-class hardware, multiple VMs, and larger deployments.

8. CONCLUSION

This study presents a proof-of-concept, closed-loop mechanism for responding to an SLA-degraded operating state through guest-level CPU-affinity adjustment. The archived Random Forest results show close agreement with a p95-based label but do not demonstrate independent CPU-fragmentation detection because latency-derived predictors overlap with the target and the evaluation contains temporally related observations.

The mitigation trace provides the stronger evidence: after cpuset reallocation, median CPU imbalance decreased from approximately 33 to 8 and p95 latency narrowed from approximately 95-130 ms to approximately 85-100 ms. These findings support feasibility within the evaluated environment. Leakage-resistant run-level validation, repeated trial statistics, and rule-based controller comparisons are required before broader claims can be made.

9. REFERENCES

- Cai, B., Wang, B., Yang, M., & Guo, Q. (2023). AutoMan: Resource-efficient provisioning with tail latency guarantees for microservices. *Future Generation Computer Systems*, 143, 61–75.
- Cano, I., Chen, L., Fonseca, P., Chen, T., Cheah, C., Gupta, K., Chandra, R., & Krishnamurthy, A. (2018). ADARES: Adaptive resource management for virtual machines. *arXiv preprint arXiv:1812.01837*.
- Choi, B., Park, J., Lee, C., & Han, D. (2021). pHPA: A proactive autoscaling framework for microservice chain. In *Proceedings of the 5th Asia-Pacific Workshop on Networking (APNet)* (pp. 65–71). ACM.
- GhatrehSamani, D., Denninnart, C., Bacik, J., & Amini Salehi, M. (2020). The art of CPU-pinning: Evaluating and improving the performance of virtualization and containerization platforms. In *Proceedings of the 49th International Conference on Parallel Processing (ICPP)* (pp. 1–11). ACM.
- Ji, S., Li, M. D., Ji, N., & Li, B. (2018). An online virtual machine placement algorithm in an over-committed cloud. In *Proceedings of the 2018 IEEE International Conference on Cloud Engineering (IC2E)* (pp. 96–106). IEEE.

- Miao, T., & Chen, H. (2015). FlexCore: Dynamic virtual machine scheduling using VCPU ballooning. *Tsinghua Science and Technology*, 20(1), 7–16.
- Mishra, M., & Bellur, U. (2016). De-fragmenting the cloud. In *Proceedings of the 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)* (pp. 511–520). IEEE.
- Podzimek, A., Bulej, L., Chen, L. Y., Binder, W., & Tůma, P. (2015). Analyzing the impact of CPU pinning and partial CPU loads on performance and energy efficiency. In *Proceedings of the 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)* (pp. 1–10). IEEE.
- Potdar, A. M., Narayan, N. D. G., Kengond, S., & Mulla, M. M. (2020). Performance evaluation of Docker container and virtual machine. *Procedia Computer Science*, 171, 1419–1428.
- Rahman, J., & Lama, P. (2019). Predicting the end-to-end tail latency of containerized microservices in the cloud. In *Proceedings of the 2019 IEEE International Conference on Cloud Engineering (IC2E)* (pp. 192–202). IEEE.
- Sachidananda, V., & Sivaraman, A. (2024). Erlang: Application-aware autoscaling for cloud microservices. In *Proceedings of the 19th European Conference on Computer Systems (EuroSys)* (pp. 888–923). ACM.
- Semerikov, S. O., Zubov, D., Kupin, A., Kosei, M., & Holiver, V. (2024). Models and technologies for autoscaling based on machine learning for microservices architecture. In *Proceedings of the 8th International Conference on Computational Linguistics and Intelligent Systems (CoLInS 2024)*, CEUR Workshop Proceedings (Vol. 3664, pp. 316–330).
- Shah, S. A. R., Shah, A. A., Baker, T., Buyya, R., & Rawat, D. B. (2021). Benchmarking and performance evaluations on various configurations of virtual machine and containers for cloud-based scientific workloads. *Applied Sciences*, 11(3), Article 993.
- Sobieraj, M., Cichoń, P., & Wydra, P. J. (2024). Docker performance evaluation across operating systems. *Applied Sciences*, 14(15), Article 6672.
- Wajahat, M., Karve, A., Kochut, A., & Gandhi, A. (2019). MLscale: A machine learning based application-agnostic autoscaler. *Sustainable Computing: Informatics and Systems*, 22, 287–299.
- Wang, Z., Li, P., Liang, C.-J. M., Wu, F., & Yan, F. Y. (2024). Autothrottle: A practical bi-level approach to resource management for SLO-targeted microservices. In *Proceedings of the 21st USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (pp. 479–496). USENIX Association.
- Xu, M., Xu, Y., Zhou, P., Zhao, P., Xu, W., Gao, D., Ren, J., & Ren, S. (2022). CoScal: Multifaceted scaling of microservices with reinforcement learning. *IEEE Transactions on Network and Service Management*, 19(4), 3995–4009.