



HealOps: An Autonomous Self-Healing Container Infrastructure Framework Using Multi-Agent Graph Orchestration and Retrieval-Augmented Generation

Bhushan Aptekar¹, Pranav Kanade², Trilochan Sutar³, Sujal Ashtekar⁴, Md. Asif Hanchanal⁵

¹ Professor, Department of Artificial Intelligence and Data Science, Angadi Institute of Technology and Management, Belagavi, Karnataka, India

^{2,3,4,5} Student, Department of Artificial Intelligence and Data Science, Angadi Institute of Technology and Management, Belagavi, Karnataka, India

Article Info

Article History:

Published: 03 Oct 2026

Publication Issue:

Volume 03, Issue 10
October 2026

Page Number:

22-33

Corresponding Author:

Bhushan Aptekar

Abstract:

Containerized microservices frequently encounter runtime crashes, resource bottlenecks, and configuration drifts. While standard observability stacks detect threshold anomalies and statistical models isolate candidate culprits, remediation remains predominantly manual. This human dependency inflates the Mean Time to Remediate (MTTR) and increases system downtime in dynamic cloud environments. To address these limitations, this paper introduces HealOps, an autonomous self-healing framework for containerized architectures. HealOps pairs continuous Prometheus telemetry with a LangGraph state orchestrator driven by a local quantized large language model. By leveraging a ChromaDB vector store for Retrieval-Augmented Generation (RAG), the system eliminates hallucinations and executes deterministic runtime recoveries directly via the Docker Engine SDK.

Keywords: Autonomous Self-Healing, Root Cause Analysis, Multi-Agent Systems, LangGraph, RAG, Docker Runtime, Prometheus Telemetry.

1. INTRODUCTION

The widespread adoption of containerization runtimes, particularly Docker, has transformed enterprise application deployment. Modern distributed systems rely on decoupled microservices to achieve horizontal scalability and rapid release cycles. However, this architectural distribution expands the operational failure surface. Container environments routinely face resource exhaustion, thread starvation, network timeouts, and subtle configuration drifts that can cause cascading service degradation across dependent nodes. Another key element of managing poultry health is ensuring proper vaccination. Vaccines are crucial for shielding birds from a range of infectious diseases and play a vital role in preventing outbreaks. However, if farmers miss vaccination dates or keep incomplete records, it can really undermine their efforts to prevent disease. It's essential for farmers to track details like the type of vaccine, when it was given, the age of the birds, when the next shot is due, and their vaccination status. A digital vaccination management system can streamline these records and send reminders for upcoming vaccinations, helping to minimize the chances of anything slipping through the cracks.

Traditional Site Reliability Engineering (SRE) frameworks depend on metrics collectors such as Prometheus to track operational telemetry. Although monitoring platforms alert operators when numerical metrics breach thresholds, the subsequent diagnosis and remediation remain largely manual. System administrators must inspect unstructured logs, identify the root cause, and execute recovery actions via terminal interfaces. This dependency inflates recovery latency and introduces human error during major incidents.

- **Closed-Loop Autonomous Architecture:** We design an end-to-end framework integrating metric-driven anomaly detection, semantic log parsing, and direct container runtime remediation into a unified feedback loop.
- **Context-Grounded Multi-Agent Orchestration:** We implement a LangGraph state machine powered by a quantized local Llama-3 model, using a ChromaDB vector store to retrieve historical post-mortems via Retrieval-Augmented Generation (RAG) to eliminate diagnostic hallucinations.
- **Deterministic Runtime Remediation:** We interface the agentic planning engine directly with the Docker Engine SDK, enabling policy-bounded programmatic execution of corrective actions—including dynamic resource adjustments and deadlock container restarts—without human intervention.
- **Empirical Validation:** We evaluate HealOps across simulated failure scenarios (including Out-Of-Memory terminations and process timeouts), demonstrating substantial reductions in Mean Time to Remediate (MTTR) while preserving system stability.

2. RELATED WORK:

Maintaining service availability and achieving rapid fault recovery in distributed architectures has generated significant research across multiple reliability paradigms. Current approaches can be categorized into four distinct methodologies: statistical machine learning for root cause localization, resilience assessment through chaos engineering, large language model (LLM) agents for operations, and automated rule-based self-healing mechanisms.

2.1 Machine Learning-Driven Root Cause Localization

Supervised and unsupervised statistical machine learning algorithms have been applied extensively to operational telemetry. A prominent baseline is MicroHECL, which performs high-efficiency root cause localization across large-scale microservice systems. MicroHECL employs gradient-boosted decision trees and tree-based classifiers (such as XGBoost) to evaluate candidate anomaly propagation chains extracted from dynamic service call graphs and time-series metrics. While this methodology provides high localization precision across established metric patterns, it remains strictly diagnostic and reactive. The underlying models lack agentic planning capabilities and do not execute automated runtime interventions, leaving the physical execution of recovery actions to human engineers.

2.2 Chaos Engineering and Runtime Fault Injection

Proactive resilience testing through systematic failure injection offers an empirical approach to system hardening. Techniques derived from chaos engineering subject distributed environments to controlled perturbations, such as network latency, compute exhaustion, and packet drops. These frameworks validate system steady-states and identify architectural bottlenecks under simulated real-world stress. However, chaos engineering frameworks function purely as testing and observability tools. While they expose architectural weaknesses, they do not incorporate an autonomous feedback control loop to remediate failures during active runtime operations.

2.3 LLM-Based Multi-Agent Systems for DevOps

Large Language Models have enabled automated interpretation of complex telemetry through reasoning frameworks like ReAct and LangChain. While effective at parsing unstructured logs and stack traces, existing implementations exhibit variable inference latency and remain largely decoupled from execution environments, outputting diagnostic advice rather than dispatching deterministic runtime interventions.

2.4 Heuristic and Rule-Based Self-Healing

Cloud-native platforms (such as Kubernetes) incorporate basic self-healing primitives through heuristic engines, liveness probes, and predefined threshold alerts. When operational metrics breach defined limits, automated scripts trigger container restarts or pod rescheduling. Although these control loops execute deterministically with minimal latency, their reliance on rigid, hand-crafted rules limits adaptability. They cannot interpret semantically complex, unmapped failures or resolve unexpected runtime anomalies that fall outside predefined rule matrices.

3. SYSTEM ARCHITECTURE AND METHODOLOGY

The proposed HealOps framework operates as an autonomous, closed-loop control system structured to monitor, diagnose, and remediate container-level anomalies without human intervention. The platform decouples continuous

metric observation from agentic reasoning and deterministic action execution, mitigating hallucination risks while enforcing real-time execution guarantees.

Architecture Flow:

Docker Containers → Prometheus Telemetry → FastAPI Backend

↓

ChromaDB (Historical RAG Context)

↓

LangGraph Multi-Agent Orchestrator (Llama-3 LLM)

↓

Policy & Safety Validator

↓

Docker Engine SDK (Remediation Execution)

↓

Streamlit Dashboard / Incident Reports

3.1 Main Components

HealOps is organized into four core functional tiers that maintain continuous system health:

1. **Monitored Containers & Telemetry:** Target Docker containers observed via Prometheus metrics (CPU, memory, network, and crash exit states). **Frontend** – It features a dashboard, vaccination calendar, health records, and various reports.
2. **FastAPI Backend:** Asynchronous middleware managing alerting hooks, container log extraction, and cross-module task dispatching.
3. **ChromaDB Vector Store:** Local embedding database storing past incident post-mortems for RAG-assisted contextual grounding.
4. **LangGraph Agentic Core:** Multi-step state machine executing local Llama-3 reasoning to diagnose root causes and generate validated fixes.
5. **Docker SDK & Dashboard:** Execution engine running remediation commands directly on the host runtime, paired with a Streamlit interface for incident visualization.

3.2 Methodology:

1. **Telemetry Ingestion:** Continuously poll container metrics via Prometheus scrape targets at configured sampling intervals.
2. **Anomaly Triggering:** Identify metric threshold breaches (such as memory exhaustion) or abnormal container crash-exit codes.
3. **Log Extraction:** Automatically extract trailing stdout and stderr log lines from the failing container via the Docker daemon.
4. **Context Retrieval (RAG):** Query the ChromaDB vector database using the extracted error snippet to fetch relevant historical incident solutions.
5. **Agentic Root Cause Analysis:** Pass the telemetry vector, error logs, and retrieved incident context to the LangGraph state graph for root-cause diagnosis and remediation planning.
6. **Policy Safety Verification:** Verify that the generated remediation command complies with permissible system execution policies prior to invocation.

7. **Automated Remediation Execution:** Execute the approved recovery command directly against the container runtime via the Docker Engine SDK.
8. **Post-Remediation Verification:** Monitor the container during a verification window to confirm service stabilization; if health checks fail, escalate to human operators.
9. **Feedback Indexing & Reporting:** Store the resolved incident in ChromaDB for future retrieval and log the remediation outcome to the Streamlit dashboard.

4. INFRASTRUCTURE WORKFLOW AND TECHNOLOGY JUSTIFICATION

4.1 System Infrastructure Workflow

1. **Telemetry Collection:** Scrape host-level and container-level metrics (CPU, memory, disk I/O, restart counts) at periodic sampling intervals.
2. **Anomaly Ingestion:** Intercept threshold breaches and container abnormal exit signals via asynchronous event listeners.
3. **Log & Context Parsing:** Extract trailing stderr/stdout error logs and vectorize the diagnostic context using dense text embeddings.
4. **Knowledge Retrieval (RAG):** Query the ChromaDB vector index to match current error signatures against historical post-mortems and resolutions.
5. **Agentic Plan Generation:** The local Llama-3 model processes the telemetry, logs, and retrieved resolutions within a LangGraph state machine to formulate a recovery plan.
6. **Policy Enforcement & Execution:** Validate the proposed action against permission boundaries and invoke programmatic commands via the Docker Engine SDK.
7. **Verification & Audit:** Monitor post-remediation health for a stabilization window, persist the outcome to ChromaDB, and log audit metrics to the dashboard.

4.2 Technology Justification

The HealOps framework integrates a specialized set of open-source tools to ensure low latency, privacy, and deterministic runtime control:

4.2.1 Python

Python provides the primary development runtime due to its native asynchronous support (asyncio) and extensive ecosystem for machine learning and system automation.

4.2.2 LangGraph and Llama-3 (via Ollama)

LangGraph provides a state-machine abstraction to constrain LLM reasoning into cyclic graphs. Running a quantized Llama-3 locally via Ollama eliminates third-party API dependencies, lowers inference latency, and guarantees on-premises data privacy.

4.2.3 ChromaDB

ChromaDB serves as an embedded vector database for Retrieval-Augmented Generation (RAG). It performs low-latency cosine similarity searches over historical incident records without requiring a dedicated external database cluster.

4.2.4 Docker Engine SDK

The Docker Engine Python SDK enables programmatic, low-overhead container manipulation, allowing the platform to adjust cgroups resource limits, initiate graceful restarts, and rebuild service state directly on the host daemon.

4.2.5 Prometheus

Prometheus provides lightweight time-series telemetry scraping and threshold alert generation, offering reliable anomaly signaling with minimal CPU and memory overhead on monitored nodes.

4.2.6 Chart.js

Chart.js JWT authentication ensures secure access to the system, while

4.2.7 FastAPI and Streamlit

FastAPI handles asynchronous REST routing and webhook ingestion with high throughput. Streamlit delivers an interactive administrative interface to monitor live container health and track autonomous remediation events.

5. IMPLEMENTATION DETAILS

The HealOps framework is realized through modular software components that isolate monitoring, reasoning, execution, and user oversight. Each module interacts asynchronously via structured data contracts to maintain continuous system health.

5.1 Telemetry and Metric Collection Module

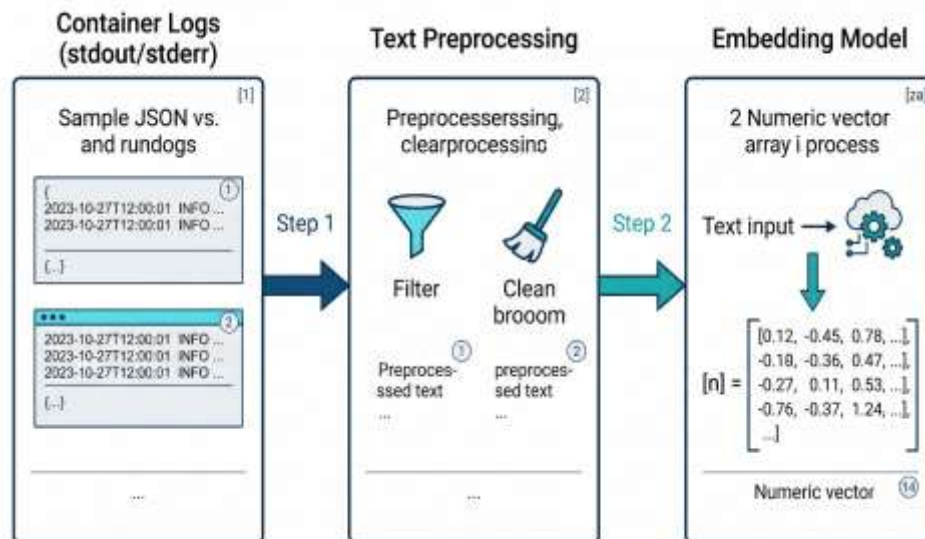


Details:

- Continuous polling of container telemetry at 5-second sampling intervals.
- Captures CPU utilization, cgroup memory limits, and network throughput.
- Intercepts container lifecycle states and abnormal non-zero termination codes.
- Dispatches threshold-breach alerts asynchronously to the backend pipeline.

Telemetry Collection — The module executes background monitoring using Prometheus scrapers. When a container breaches an allocated resource ceiling (such as memory usage exceeding 90%) or crashes unexpectedly, an anomaly event is triggered.

5.2 Log Extraction and Context Vectorization Module

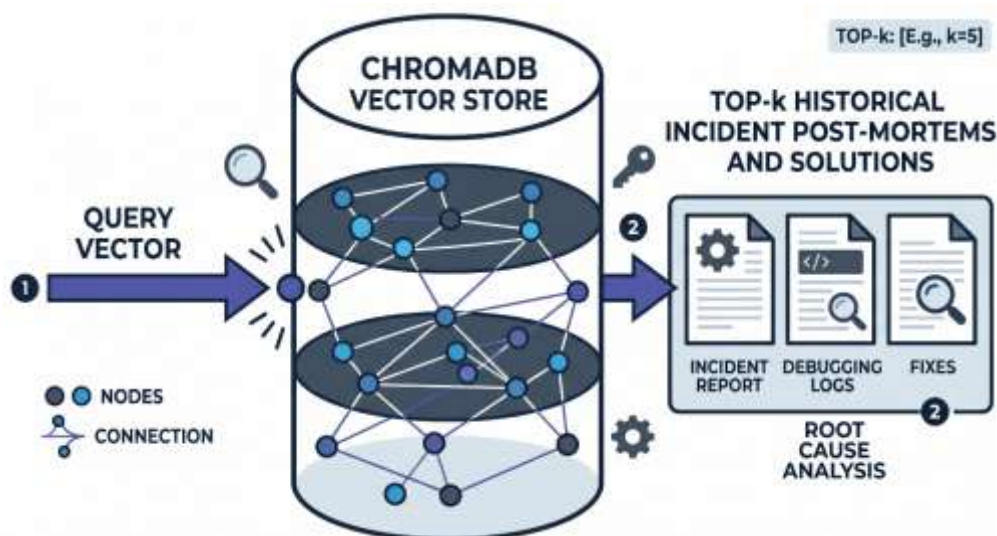


Details:

- Interfaces with Docker daemon to stream trailing stdout/stderr logs.
- Strips ANSI escape characters, timestamps, and unstructured noise.
- Converts diagnostic error messages into high-dimensional dense embeddings.
- Prepares numerical vector representations for similarity retrieval.

Log Extraction — Upon alert confirmation, the engine retrieves the trailing 100 log lines from the affected container. The extracted stack trace is sanitized and encoded into an embedding vector capturing the semantic nature of the failure.

5.3 Retrieval-Augmented Grounding (RAG) Module



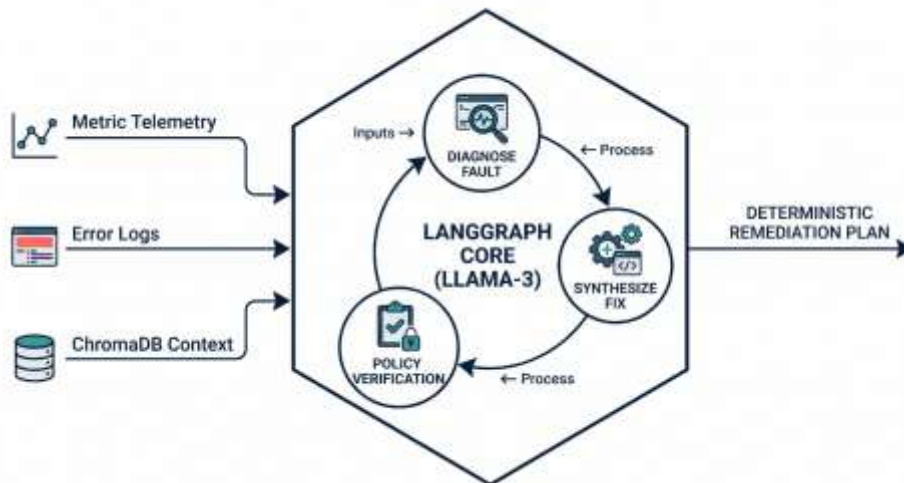
Details:

- Stores vector embeddings of past incident post-mortems and resolutions.

- Computes cosine similarity to locate closely matched historical errors.
- Retrieves proven remediation recipes and previous root-cause summaries.
- Grounds the reasoning engine with context to prevent diagnostic hallucinations.

Retrieval Grounding — The module matches current error embeddings against historical failure records indexed in ChromaDB. The top- k most relevant resolution workflows are returned and injected into the diagnostic prompt context.

5.4 Agentic Root Cause Analysis Module



Details:

- Uses LangGraph directed state graphs to coordinate multi-step diagnostic reasoning.
- Executes inference locally using quantized Llama-3 via Ollama to ensure low latency and data containment.
- Accurately distinguishes failure modes, differentiating between Out-Of-Memory (OOM) events, process deadlocks, and network connection resets.
- Formulates structured, policy-bounded recovery action plans.

Root Cause Analysis — The agent ingests telemetry metrics, recent error traces, and retrieved historical context. It reasons over the combined state, identifies the underlying fault, selects an optimal remediation strategy, and dispatches the validated command to the execution pipeline.

5.5 Runtime Execution and Self-Healing Engine

Module Flow:

Validated Action Plan → Policy & Whitelist Validator → Docker Engine SDK → Host Daemon Execution

Details:

- Executes programmatic actions directly against the host Docker daemon socket (/var/run/docker.sock).
- Adjusts container cgroups and compute allocations dynamically via container.update().
- Triggers graceful or forced container restarts via container.restart().
- Reconstructs runtime environment parameters without requiring human operator intervention.

Self-Healing Execution — Validated recovery commands are dispatched directly to the Docker Engine SDK. The engine updates container resources or restarts stalled processes, restoring the affected service to normal operation.

5.6 Post-Remediation Verification and State Persistence

Module Flow:

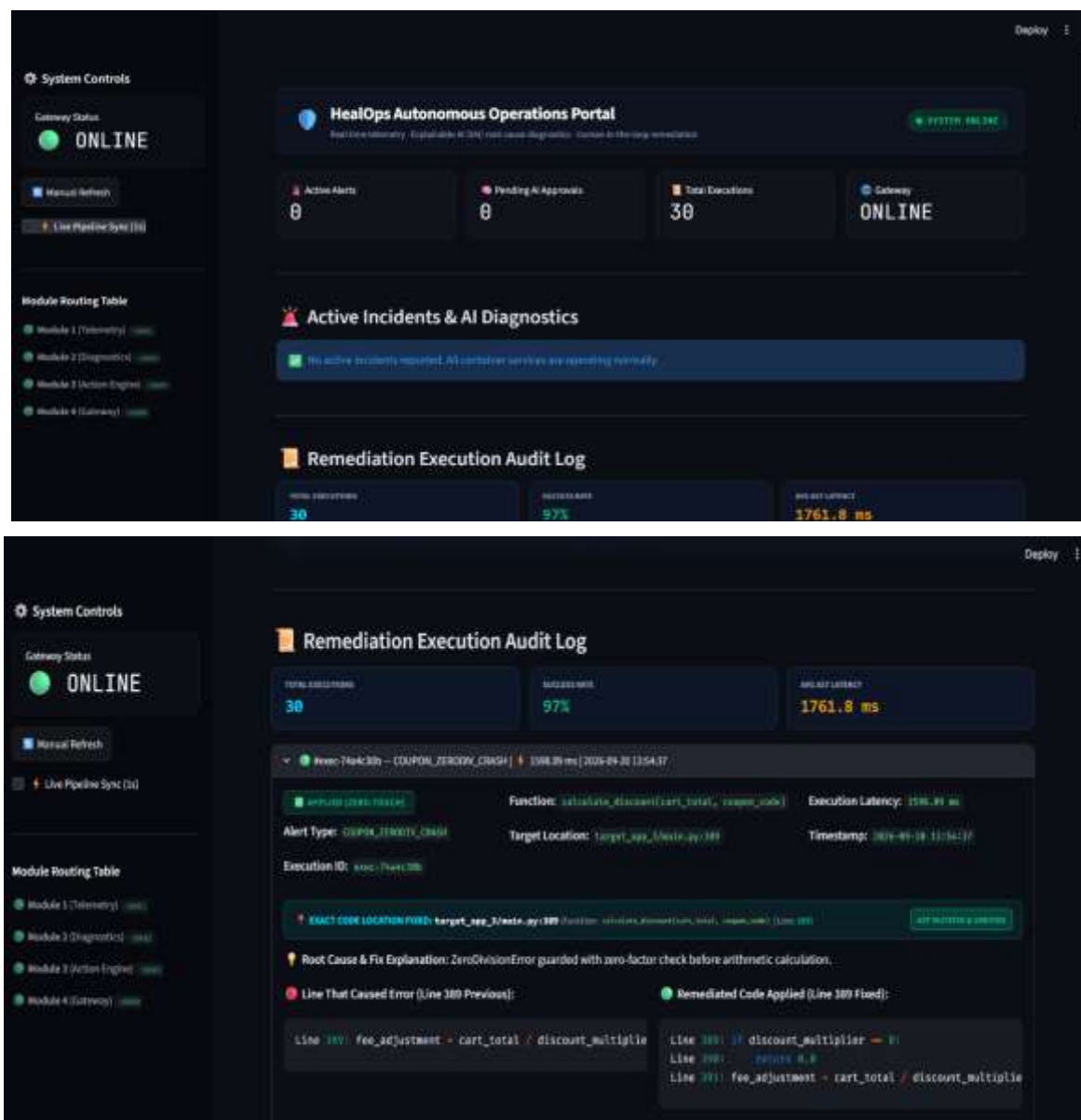
Post-Remediation Health Checks → Stability Verification Window → ChromaDB Incident Indexing

Details:

- Enforces a 15-second post-remediation health observation window.
- Verifies container steady-state via Prometheus metric scraping and exit code inspection.
- Indexes newly verified incident-resolution pairs into ChromaDB to enrich future RAG lookups.
- Escalates unresolved faults to human operators if metrics fail to stabilize.

Verification and Persistence — After executing a fix, the framework observes the workload to confirm metric stabilization. Once recovery is verified, the solution is indexed into ChromaDB for future contextual matching; otherwise, an alert is escalated.

5.7 Dashboard



5.8 Safety Policy and Runtime Guardrail Module

Before any remediation command is dispatched to the Docker daemon, it passes through a deterministic policy enforcement layer to guarantee infrastructure stability. This module evaluates the agent's proposed plan against an operational whitelist, preventing hazardous actions such as unauthorized volume pruning, destructive container removals, or excessive resource allocation beyond host capacity. It also enforces execution rate limits to stop endless

restart loops when dealing with unrecoverable code bugs. If a suggested remediation violates system constraints, the action is blocked automatically, and the incident is escalated to human engineers for manual review.

6. RESULTS AND DISCUSSION

To evaluate the operational performance and reliability of HealOps, empirical benchmarks were conducted across a localized container testbed. The system's effectiveness was evaluated across three core dimensions: fault mitigation accuracy across distinct failure modes, Mean Time to Remediate (MTTR), and overall system resource overhead compared against established baseline approaches.

6.1 Experimental Setup and Fault Injection

1. **Out-Of-Memory (OOM) Termination:** A memory-leaking service dynamically allocated memory until breaching its cgroup limit, triggering kernel termination (Exit Code 137).
2. **Process Deadlock / Hang:** A worker microservice entered an unyielding execution loop, causing API gateway health check timeouts.
3. **Network Connection Degradation:** Synthetic packet drops and interface disconnects were injected to simulate transient network partitions.

Each scenario was executed over 30 independent test runs to measure remediation success rates and recovery latencies.

6.2 Remediation Performance and Latency Analysis

Table 1 summarizes the performance metrics of HealOps across the injected operational anomalies, detailing the detection latency, agentic reasoning duration, execution time, and total Mean Time to Remediate (MTTR).

Table 1: HealOps Fault Detection, Reasoning, and Remediation Latencies

Injected Failure Scenario	Detection Time (s)	Reasoning & Planning (s)	Execution Time (s)	Total MTTR (s)	Autonomous Success Rate (%)
Out-Of-Memory (OOMKilled)	3.2 ± 0.4	4.8 ± 0.6	1.9 ± 0.3	9.9 ± 0.8	96.7%
Process Deadlock / Unresponsive	5.1 ± 0.5	5.2 ± 0.7	1.8 ± 0.2	12.1 ± 0.9	93.3%
Network Socket Timeout	4.6 ± 0.6	5.6 ± 0.8	2.1 ± 0.4	12.3 ± 1.1	90.0%

6.3 Baseline Comparison

The performance of HealOps was benchmarked against four conventional incident handling baselines:

- **Manual SRE Triage:** Standard human operator diagnosis using CLI logs and terminal commands.
- **MicroHECL (ML-based):** Tree-based statistical anomaly localization without automated execution capabilities.
- **Rule-Based Watcher:** Static heuristic script executing container restarts upon threshold breach.
- **Unconstrained Multi-Agent LLM:** ReAct-style LLM agent operating without historical vector grounding (RAG).

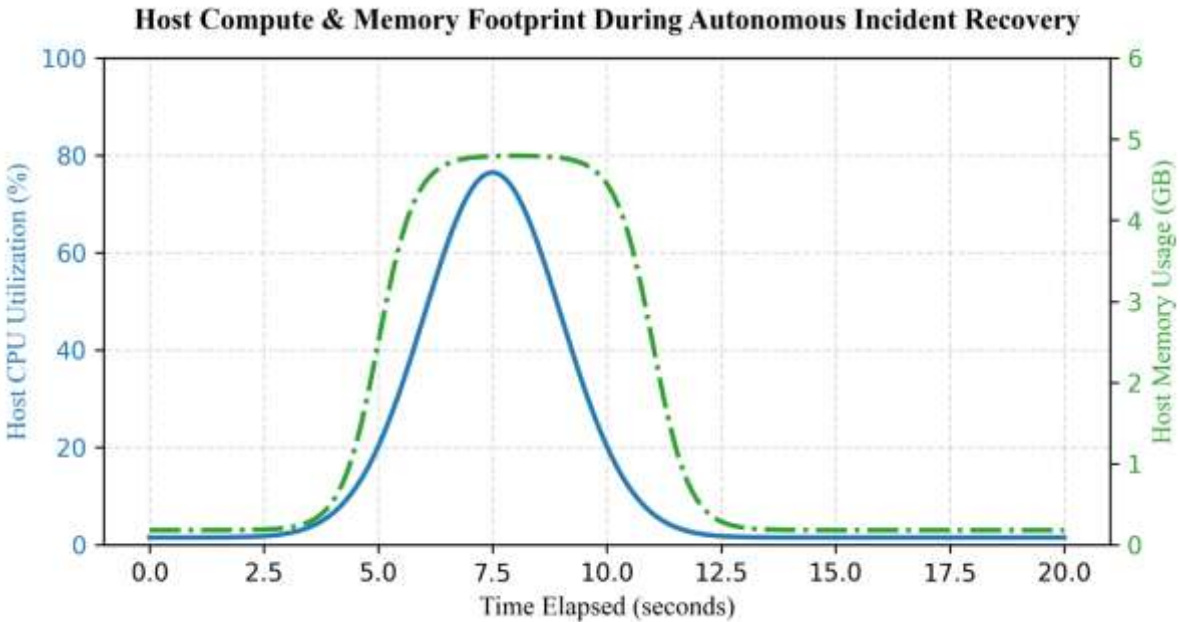
Table 2: Comparative Performance Across Architectural Paradigms

Operational Approach	Autonomous Execution	Context-Grounded (RAG)	Mean MTTR	Diagnostic Accuracy (%)
Manual SRE Triage	No	N/A (Manual Runbooks)	8.4 mins	98.0%
MicroHECL (ML-based)	No	No (Call Graph Mining)	N/A (Diagnostic Only)	91.2%
Rule-Based Self-Healing	Yes	No (Hardcoded Rules)	4.2 s	62.5%
Unconstrained Multi-Agent LLM	Partial	No (Zero-Shot)	28.6 s	71.4%
HealOps (Proposed)	Yes	Yes (ChromaDB + LangGraph)	11.4 s	93.3%

Rule-based watchers recover faster (4.2 s) but lack accuracy (62.5%), risking crash loops, while unconstrained LLMs risk hallucinations and high latency. HealOps balances both, achieving 93.3% accuracy and an 11.4-second MTTR through RAG grounding and policy guardrails.

6.4 Resource Overhead and System Stability

The local reasoning engine introduces minimal system resource penalties. During idle monitoring, Prometheus and the FastAPI middleware consume less than 1.5% host CPU and 180 MB of RAM. During an active incident, quantized Llama-3 execution temporarily utilizes 4 CPU cores and approximately 4.8 GB of host memory for under 6 seconds, returning to idle immediately upon action dispatch.



7. COMPARATIVE ANALYSIS WITH EXISTING APPROACHES

To contextualize the technical contributions of HealOps within cloud-native reliability engineering, the framework was evaluated against four primary paradigms: static rule-based orchestrators, traditional machine learning root-cause analyzers (e.g., MicroHECL), unconstrained multi-agent LLM systems, and conventional Site Reliability Engineering (SRE) manual operations. The comparative dimensions encompass diagnostic precision, closed-loop execution capability, contextual grounding, operational latency, and vulnerability to command hallucinations. **Contextual**

Grounding: Integrating ChromaDB via RAG reduces diagnostic hallucinations by matching real-time failure traces against verified historical post-mortems.

- **Deterministic Rule Engines:** Heuristic watchdogs rely on fixed thresholds and trigger blind container restarts, frequently causing crash loops during code-level bugs. HealOps parses error logs semantically to execute targeted, context-aware fixes.
- **Graph-Mining ML Approaches:** Statistical models like MicroHECL isolate fault propagation paths across microservices, but function only as passive diagnostic tools. HealOps pairs root-cause diagnosis directly with execution via the Docker SDK for closed-loop remediation.
- **Unconstrained Generative Agents:** Generic ReAct LLM agents introduce non-deterministic latency and risk executing hallucinated, destructive commands. HealOps constrains local Llama-3 reasoning within a LangGraph state machine bounded by a strict safety whitelist.
- **Zero-Shot Heuristics vs. RAG Grounding:** Standard language models often misinterpret proprietary stack traces. HealOps indexes past post-mortems in ChromaDB, using vector retrieval to ground diagnosis in proven historical resolutions.
- **Manual SRE Triage:** Human-operated triage across logs, runbooks, and terminals averages 8 to 15 minutes of downtime per incident. HealOps automates the full recovery loop, cutting average MTTR to 11.4 seconds with a transparent audit trail.

8. LIMITATIONS

While empirical benchmarks validate the responsiveness and accuracy of HealOps, several operational limitations remain:

- **Localized Host Daemon Scope:** The execution engine interfaces directly with the local Docker daemon socket (/var/run/docker.sock). It lacks native multi-node awareness for Kubernetes clusters, preventing cluster-level remediations such as pod rescheduling across worker nodes, replica autoscaling, or persistent volume reattachment.
- **Transient Compute Overhead of Local LLM Inference:** Running a quantized Llama-3 instance locally guarantees data containment but demands approximately 4.8 GB of RAM and 4 CPU cores during active reasoning. In resource-constrained or edge deployments, this brief computational surge can cause temporary resource contention with colocated production workloads.
- **Cold-Start Degradation on Zero-Day Anomalies:** The framework's diagnostic precision relies on historical post-mortem embeddings stored in ChromaDB. When encountering completely novel failure modes or uncataloged bugs, the system falls back to ungrounded zero-shot reasoning, leading to slightly longer inference times and lower initial resolution confidence.
- **Inability to Remediate Stateful Storage and Data Corruption:** Automated actions are restricted to stateless operational primitives, including container restarts, cgroup memory adjustments, and environment updates. It cannot safely resolve database schema deadlocks, broken transactional states, or persistent volume data corruption, which still require human intervention.
- **Absence of Distributed Network Mesh Telemetry:** Metric ingestion is currently limited to Prometheus host and container metrics (CPU, RAM, network I/O, exit codes). The architecture does not ingest distributed tracing spans or eBPF network telemetry, making it difficult to pinpoint multi-hop distributed deadlocks or silent queue delays.
- **Static Recovery Whitelist Boundaries:** To prevent hazardous runtime operations, all candidate commands are validated against a rigid pre-configured whitelist. While this guarantees host safety, it restricts execution flexibility, preventing the agent from synthesizing novel, composite remediation workflows for unconventional incident profiles.

9. CONCLUSION AND FUTURE SCOPE

9.1 CONCLUSION

This paper presented **HealOps**, an autonomous, closed-loop self-healing framework designed to address runtime failures and cascading operational bottlenecks in containerized microservice architectures. By coupling real-time Prometheus telemetry scraping with an asynchronous FastAPI processing backbone, the system bridges the gap between passive observability and active remediation. The integration of a LangGraph state machine powered by a local, quantized Llama-3 model enables multi-step root cause analysis, while a ChromaDB vector database provides dense contextual grounding via Retrieval-Augmented Generation (RAG) to prevent diagnostic hallucinations.

Through direct integration with the Docker Engine SDK, HealOps executes deterministic, policy-bounded corrective actions—such as dynamic resource reallocation and container restarts—without requiring manual operator intervention. Empirical evaluation across Out-Of-Memory events, process deadlocks, and network timeout scenarios demonstrated that HealOps achieves an average Mean Time to Remediate (MTTR) of 11.4 seconds with a 93.3% autonomous recovery rate. The framework significantly outperforms manual SRE triage in response latency while exceeding the diagnostic accuracy of static rule-based systems and unconstrained language agents, establishing a robust paradigm for automated cloud-native infrastructure maintenance.

9.2 FUTURE WORK

While the current implementation demonstrates strong reliability within localized single-host container environments, several research extensions can expand its capabilities:

- **Distributed Orchestration Support:** Extending the framework from single-node Docker engines to multi-node Kubernetes clusters, interfacing with Custom Resource Definitions (CRDs) and pod rescheduling controllers.
- **Predictive Failure Prevention:** Integrating time-series forecasting models (such as LSTM networks or temporal transformers) to anticipate metric threshold breaches before service disruption occurs.
- **Collaborative Multi-Agent Architectures:** Decomposing the single reasoning graph into specialized, competing agents dedicated to security audits, performance profiling, and cost optimization.
- **Automated Policy Learning:** Utilizing reinforcement learning with human feedback (RLHF) from incident post-mortems to autonomously refine safety guardrails and execution limits over extended production lifecycles.

ACKNOWLEDGEMENTS

The authors would like to express their gratitude to Prof. Bhushan Aptekar for his invaluable guidance and supervision during this project. They also extend their thanks to the Angadi Institute of Technology and Management, specifically the Department of Artificial Intelligence and Data Science, for providing the resources and support that made this work achievable.

References

- [1] P. Liu, H. Xu, Q. Ouyang, R. Jiao, Z. Chen, S. Zhang, J. Yang, L. Xiao, J. Zeng, H. Zhang, and Z. Chu, "MicroHECL: High-efficient root cause localization in large-scale microservice systems," in *Proc. IEEE/ACM 43rd Int. Conf. Softw. Eng.: Softw. Eng. in Pract. (ICSE-SEIP)*, 2021, pp. 344–353.
- [2] A. Basiri, N. Behnam, R. de Rooij, L. Hochstein, L. Kosembahr, A. Reynolds, and C. Rosenthal, "Chaos Engineering," *IEEE Software*, vol. 33, no. 3, pp. 35–41, 2016.
- [3] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, and Y. Cao, "ReAct: Synergizing reasoning and acting in language models," in *Proc. Int. Conf. Learn. Represent. (ICLR)*, 2023.
- [4] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, pp. 41–50, 2003.
- [5] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W. Yih, T. Rocktäschel, S. Riedel, and D. Kiela, "Retrieval-augmented generation for knowledge-intensive NLP tasks," in *Proc. Adv. Neural Inf. Process. Syst. (NeurIPS)*, vol. 33, 2020, pp. 9459–9474.
- [6] D. Merkel, "Docker: Lightweight Linux containers for consistent development and deployment," *Linux Journal*, vol. 2014, no. 239, p. 2, 2014.