Research Square

Preprints are preliminary reports that have not undergone peer review.
They should not be considered conclusive, used to inform clinical practice,
or referenced by the media as validated information.

AgentSentinel: A Multi-Agent Framework for Autonomous Root Cause Analysis and Self-Healing in Distributed Systems

HAZAIFA TAHIR

hazaifatahir245@gmail.com

Lahore https://orcid.org/0009-0009-5586-9540

Saboor Ahmed

lahore garrison university

Research Article

Keywords: Multi-Agent Systems,Large Language Models (LLMs) , Self-Healing Systems , Root Cause Analysis (RCA) , Anomaly Detection , AIOps

Posted Date: November 12th, 2026

DOI: https://doi.org/10.21203/rs.3.rs-11690322/v1

License: This work is licensed under a Creative Commons Attribution 4.0 International License.
Read Full License

Additional Declarations: The authors declare no competing interests.

AgentSentinel: A Multi-Agent Framework for Autonomous Root Cause Analysis and Self-Healing in Distributed Systems

Hazaifa Tahir[1], Saboor Ahmed[1] [1]
[1]*Department of Computer Science*, *Lahore Garrison University*, Lahore, Pakistan
Email: hazaifatahir245@gmail.com, fa24-bscs-400@cs.lgu.edu.pk

***Abstract*—Failures in distributed systems propagate rapidly across interdependent components, yet widely adopted observability platforms such as Prometheus and Grafana provide only detection and alerting capabilities, leaving remediation entirely to human operators. This paper presents AgentSentinel, an autonomous multi-agent framework that detects anomalies, determines their root cause using large language models (LLMs), and executes targeted repairs on live infrastructure while maintaining fallback escalation paths. The framework comprises three specialized agents. The Observer Agent monitors service logs and metrics using a weighted multi-signal scoring model. The Reasoning Agent employs a ReAct-based [2] loop with an LLM and a service dependency graph to isolate the true root cause from downstream symptoms. The Repair Agent validates proposed actions against an allowlist of reversible operations and executes them on the running system. A feedback mechanism termed Reinforcement-Informed Learning from Memory (RILFM) persists resolved incident records and surfaces them as few-shot examples for future reasoning cycles. The framework was evaluated in a controlled, single-run feasibility study on a Docker Compose testbed across five fault scenarios: memory leak, service crash, CPU spike, database connection failure, and network timeout. All five faults were detected and remediated in a single ReAct iteration, with a mean time to detect (MTTD) of 1.98 seconds, a mean time to repair (MTTR) of 5.58 seconds, and zero false positives during fault-free monitoring. These results establish architectural feasibility in a controlled testbed rather than production-scale readiness.**

***Index Terms*—multi-agent systems, large language models, self-healing systems, root cause analysis, anomaly detection, AIOps**

I. Introduction

Contemporary software infrastructure predominantly relies on distributed architectures in which containerized microservices communicate via REST APIs and message queues. Such architectures provide high scalability but introduce significant fault propagation risk: a failure in a single component can cascade across dependent services, resulting in system-wide outages. For large-scale deployments, unplanned downtime incurs costs measured in millions of dollars per hour, and production incident resolution typically requires tens of minutes to several hours [1].

Modern observability tooling addresses only the detection dimension of this problem. Platforms such as Prometheus, Grafana, and PagerDuty generate alerts when anomalies are detected but provide no automated remediation capability. Upon receiving an alert, an on-call engineer must manually connect to the affected system, analyze logs, identify the root cause, and apply a fix — a process that is both time-consuming and error-prone.

Recent advances in large language models (LLMs) have substantially expanded what automated systems can accomplish. Models such as GPT-4, Llama, and Gemini are capable of parsing raw log data, inferring causal relationships, and proposing remediation steps in natural language [6], [7]. The ReAct framework [2] extends this capability by interleaving reasoning and tool invocation in a closed loop, which is well-suited to live infrastructure debugging. Multi-agent frameworks such as AutoGen [8] and LangGraph provide the orchestration layer needed to distribute this work across specialized agents.

This paper introduces AgentSentinel, a multi-agent framework that integrates fault detection, LLM-based root cause analysis, and automated repair into a single closed-loop pipeline. We present the system as a proof of concept: the goal is to validate the architecture in a controlled testbed, not to claim production-scale readiness, which we leave to future work.

The contributions of this work are as follows:

- A three-agent closed-loop design integrating anomaly detection, LLM-based root cause analysis, and automated repair into a unified pipeline.
- A ReAct reasoning configuration that incorporates a service dependency graph, enabling the LLM to distinguish actual root causes from downstream symptoms.
- RILFM, a feedback mechanism that stores resolved incident records and reuses them as few-shot examples in subsequent reasoning prompts.
- An initial single-run feasibility evaluation across five fault types on a functional Docker Compose testbed.

II. Literature Review

A. Anomaly Detection in Distributed Systems

Traditional anomaly detection relied on static, threshold-based rules that function adequately in simple environments but are insufficient for the complexity and dynamism of modern distributed systems.

Machine learning approaches have improved detection fidelity considerably. DeepLog [4] employs an LSTM to model

normal log sequences and flag deviations. LogBERT [5] applies BERT-based contextual embeddings to the same problem, achieving richer semantic representations. Both approaches are effective at detection but terminate at the alert stage, providing no diagnostic or remediation capability.

B. LLMs for Log Analysis and Repair

LLMs are capable of processing unstructured, noisy log data and producing human-readable explanations of observed failures. LogPrompt [6] demonstrates that zero-shot prompting achieves competitive performance on log parsing tasks without task-specific fine-tuning, which makes deployment practical.

Research on LLM-based code repair [7] demonstrates that these models can identify defects and generate patches across a wide range of programming languages. A key limitation of existing work in this area is that the models operate in isolation: they can propose a fix but cannot apply it to a live system, verify whether it resolved the issue, or attempt an alternative if it did not.

C. Agentic AI and Multi-Agent Frameworks

Agent frameworks have begun to close this gap. The ReAct framework [2] alternates between reasoning and tool invocation, a pattern well-matched to iterative debugging tasks. AutoGen [8] and LangGraph extend this to collaborative multi-agent settings. While these frameworks have been applied to coding assistance and data processing tasks, we are not aware of prior work that deploys them for live infrastructure management.

D. Self-Healing Systems and AIOps

The concept of self-healing systems originates from the autonomic computing paradigm of the early 2000s [1]. Early implementations employed hardcoded recovery scripts that performed reliably on anticipated failure modes but could not generalize to novel conditions.

Contemporary AIOps platforms, such as Moogsoft and IBM Watson AIOps, apply machine learning to alert correlation and noise reduction. These platforms provide decision support but continue to delegate final remediation decisions to human operators. AgentSentinel distinguishes itself from this class of tools by having the LLM both select and autonomously execute a course of action within a constrained, validated action space.

E. Comparison of Existing Approaches

Table I situates AgentSentinel relative to representative prior work across the four stages of fault management.

The majority of existing approaches address only one or two stages of the fault management lifecycle. AgentSentinel is the first system, to the authors' knowledge, to integrate detection, diagnosis, repair, and a learning cycle within a single unified framework.

TABLE I
COMPARISON OF EXISTING APPROACHES

Approach	Detect	Diagnose	Repair	Learns
DeepLog [4]	Yes	No	No	No
LogBERT [5]	Yes	No	No	No
LogPrompt [6]	Yes	Partial	No	No
Rule-Based Scripts	Yes	No	Partial	No
AIOps Platforms	Yes	Partial	No	Partial
AgentSentinel	**Yes**	**Yes**	**Yes**	**Yes**

F. Research Gap

Existing tools address individual stages of the fault management lifecycle in isolation. Anomaly detection systems identify problems without resolving them. LLM-based tools reason about causes but cannot act on live infrastructure. Agentic frameworks support autonomous action but have not been deployed in infrastructure contexts. AIOps platforms reduce alert noise but retain human operators in the remediation loop. AgentSentinel is designed to bridge all four stages within one closed-loop system.

III. PROPOSED METHODOLOGY

A. Architecture Overview

AgentSentinel comprises three agents: Observer, Reasoning, and Repair. Under normal operating conditions, only the Observer is active. Upon anomaly detection, the full pipeline is triggered: the fault is diagnosed, a repair is executed, and the outcome is evaluated. A successful repair returns the system to the monitoring state; a failed repair re-enters the loop with additional context. All three agents share access to a service dependency graph and the RILFM memory store. The complete architecture is illustrated in Fig. 1.

B. Observer Agent

The Observer Agent continuously monitors all services by polling logs and metrics at fixed intervals and applying a weighted scoring model to each service. Four signals are tracked: CPU utilization exceeding 85% (weight 30), memory utilization exceeding 90% (weight 30), error rate exceeding 50% (weight 25), and request latency exceeding 3000 ms (weight 15). A cumulative score of 40 or higher triggers an anomaly classification. The thresholds reflect standard monitoring practice and are candidates for environment-specific tuning in future work.

Upon anomaly confirmation, the Observer packages a fault report containing the service identifier, associated logs, metric snapshot, and timestamp, and forwards it to the Reasoning Agent. The scoring procedure is described in Algorithm 1.

C. Reasoning Agent

The Reasoning Agent performs root cause diagnosis by executing a ReAct loop with an LLM. Each iteration consists of a reasoning phase, in which the model applies chain-of-thought analysis [3] to generate diagnostic hypotheses, followed by an action phase in which the agent queries the

Algorithm 1 Observer Agent Anomaly Scoring

Require: Metrics snapshot M for service s
Ensure: Anomaly alert or continue monitoring

$score \leftarrow 0$
if $M.cpu > 85\%$ **then**
  $score \mathrel{+}= 30$
end if
if $M.memory > 90\%$ **then**
  $score \mathrel{+}= 30$
end if
if $M.error_rate > 50\%$ **then**
  $score \mathrel{+}= 25$
end if
if $M.latency > 3000\,\text{ms}$ **then**
  $score \mathrel{+}= 15$
end if
if $score \geq 40$ **then**
  Package fault report $(s,\ logs,\ M,\ timestamp)$
  Forward to Reasoning Agent
else
  Continue monitoring loop
end if

dependency graph and the RILFM memory store to gather supporting evidence.

The dependency graph encodes inter-service relationships, enabling the agent to trace fault propagation paths. For instance, if `order_service` fails, the graph indicates that `api_gateway` will exhibit secondary failures due to its dependency on `order_service`, while `order_service` itself depends on `database` and `payment_service`. Without this structural context, the LLM may misattribute root cause to a downstream symptom rather than the originating service.

Each reasoning prompt comprises four components: a system role establishing the LLM as an infrastructure reliability expert, the fault report from the Observer, the relevant subgraph of the dependency graph, and any matching prior incident records from RILFM as few-shot examples. The LLM output is structured JSON specifying the target service, action type (restart, patch, scale, or flush), and action parameters. Structured output is required for deterministic parsing and safety validation by the Repair Agent.

Llama-3.1-70B was selected via the Groq API for this implementation, offering a balance between reasoning quality and inference latency. The architecture is model-agnostic; any LLM supporting structured JSON output and a sufficient context window is compatible. A comparative evaluation of model choices — including GPT-4, Claude, Gemini, and smaller open-weight models — in terms of accuracy and cost is identified as future work.

D. Repair Agent and Safety Validation

The Repair Agent receives the structured JSON repair plan and executes it on the live system. Supported actions include restarting Docker containers with revised resource limits, applying LLM-generated configuration patches, scaling services horizontally by adding replicas, and flushing network caches.

Because the system acts autonomously on live infrastructure, every proposed repair plan passes through a safety validation gate before execution. The gate enforces three checks. First, an *action allowlist*: the Repair Agent can only invoke a fixed set of pre-implemented remediation primitives (restart, patch, scale, flush); the LLM selects among these primitives via structured fields and at no point supplies free-form shell commands, so arbitrary or unanticipated operations cannot reach the execution layer. Second, *schema and target validation*: the JSON plan is rejected if the action type is outside the allowlist, if the target service identifier does not exist in the service registry, or if required parameters are missing or malformed. Third, a *reversibility classification*: each allowlisted primitive is statically tagged as reversible; operations that would be destructive or difficult to roll back (e.g., data deletion, schema changes, irreversible configuration overwrites) are deliberately not implemented as primitives and any plan requesting them is rejected and returned to the Reasoning Agent with an explanatory message. Configuration patches are additionally bounded to service-level parameters (e.g., resource limits, connection pool sizes) rather than application code.

Following execution, the Repair Agent verifies the outcome by polling the target service's health endpoint before reporting success. Upon successful execution, it logs the full incident record — including fault type, affected service, action taken, executed command, and timestamps — and forwards it to the RILFM feedback loop. Combined with the bounded retry-and-escalation policy of Section III-E, this ensures the autonomous action space remains constrained and human oversight is retained for any failure mode the system cannot resolve.

E. RILFM Feedback Loop

Following a repair action, the Observer monitors system metrics for a defined observation window. If all metrics return to normal ranges and no new anomalies are detected, the repair is classified as successful and the incident record is persisted to the RILFM store. Subsequent incidents involving the same service or fault type can retrieve this record as a few-shot example, potentially accelerating correct diagnosis.

If metrics do not recover within the observation window, the repair is classified as failed. The failure record is appended to the Reasoning Agent's context for a subsequent attempt. After three consecutive failed attempts, the system escalates to a human operator with a complete incident report. The full loop is described in Algorithm 2. We have not yet directly measured how much RILFM improves diagnostic accuracy over repeated incidents; this is the first experiment we intend to run next.

Algorithm 2 RILFM Feedback and Retry Loop

Require: Repair executed; observation window W; max retries $K = 3$

$attempt \leftarrow 1$
while $attempt \leq K$ **do**
  Monitor metrics for duration W
  if metrics normal **and** no new anomalies **then**
    Mark repair **successful**
    Save incident record to RILFM memory store
    return — resume monitoring
  else
    Mark repair **failed**
    Append failure record to Reasoning Agent context
    Reasoning Agent generates new repair plan
    Repair Agent executes new plan
    $attempt \mathrel{+}= 1$
  end if
end while
Escalate to human operator with full incident report

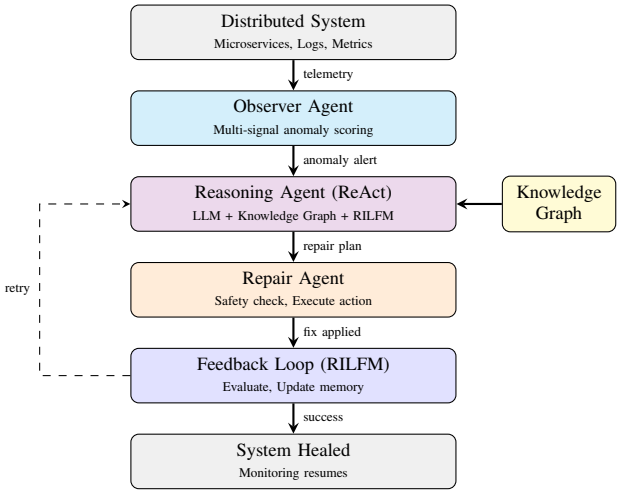


Fig. 1. AgentSentinel architecture: closed-loop multi-agent self-healing pipeline.

IV. Experimental Setup

A. Environment

AgentSentinel was evaluated on a Docker Compose testbed running on a single local machine. The testbed comprises four services: an API Gateway, an Auth Service, an Order Service, and a Payment Service, communicating via REST and message queues in a configuration representative of a microservice architecture. All services expose Prometheus metrics, which are scraped at regular intervals.

The service dependency graph is implemented as a Python dictionary. The API Gateway depends on the Auth and Order services; the Order Service depends on the Payment Service and the database. The Reasoning Agent consults this graph at runtime to determine the scope of impact and identify the originating fault source.

B. Fault Scenarios

Five distinct fault types were injected to assess the system's coverage across different failure modes:

- **Memory Leak:** Memory utilization of `order_service` was elevated to 97% with an 82% error rate, simulating a heap exhaustion condition.
- **Service Crash:** `auth_service` was terminated via an unhandled NullPointerException, causing the API Gateway to return HTTP 503 responses on authentication routes.
- **CPU Spike:** A synthetic batch workload was applied to `payment_service`, driving CPU utilization to 98% and exhausting the thread pool.
- **Database Connection Failure:** The database connection pool was drained, causing simultaneous failures across all three dependent services to test cascading fault detection.
- **Network Timeout:** Artificial latency was injected between the API Gateway and `order_service`, causing requests to exceed timeout thresholds.

C. Evaluation Approach

This evaluation is conducted as a feasibility study to verify that the proposed architecture can complete the full detect-diagnose-repair cycle across diverse fault types in a controlled testbed; it is not intended to demonstrate production-scale readiness. Each fault scenario was executed once. Single-run results do not permit estimation of variance or statistical confidence; obtaining reliable per-scenario distributions through repeated execution is the immediate next step for this line of work.

The following metrics are reported:

- **MTTD:** Elapsed time from fault injection to anomaly alert.
- **MTTR:** Elapsed time from anomaly alert to confirmed service recovery.
- **Outcome:** Whether the agent produced a correct diagnosis and the repair action restored normal operation.
- **False Positives (FP):** Spurious alerts generated during a 60-second fault-free observation period preceding each injection.

D. Reference Comparisons

For context, results are reported alongside three operational references. The human operator reference corresponds to the industry mean MTTR of 10–20 minutes for on-call incident response [1]. Rule-based automation typically resolves known alert types in 3–5 minutes. Monitoring-only deployments generate alerts without automated remediation. These reference values are drawn from published industry benchmarks and were not measured under matched conditions; they are presented as illustrative context only and must not be read as controlled baselines or direct performance comparisons.

E. Implementation

The system is implemented in Python. LangGraph is used for agent orchestration, Llama-3.1-70B via the Groq API

serves as the Reasoning Agent LLM, the dependency graph is implemented as a Python dictionary, Prometheus provides metrics collection, Docker Compose hosts the testbed, and RILFM memory is persisted in a JSON file.

V. Results and Discussion

A. Results

Table II presents the single-run feasibility results for all five fault scenarios, including the false positive count observed during each fault-free monitoring window.

TABLE II
FEASIBILITY RESULTS (SINGLE RUN PER SCENARIO)

Fault Type	Service	MTTD (s)	MTTR (s)	Iter.	FP	Outcome
Memory Leak	order_svc	1.9	4.4	1	0	Healed
Service Crash	auth_svc	1.8	5.1	1	0	Healed
CPU Spike	payment_svc	2.1	6.3	1	0	Healed
DB Failure	database	2.3	7.2	1	0	Healed
Net. Timeout	api_gateway	1.8	4.9	1	0	Healed
Avg.	**—**	**1.98**	**5.58**	**1.0**	**0**	**5/5**

AgentSentinel completed the full detect-diagnose-repair pipeline for all five scenarios in a single ReAct iteration without human intervention. Detection latency remained below 2.5 seconds across all scenarios. The worst-case end-to-end recovery time was 7.2 seconds. No false positive alerts were observed during any of the 60-second fault-free monitoring windows. Because every scenario was resolved on the first attempt, the retry and human-escalation branch of Algorithm 2 was never exercised in this evaluation; its behavior under repair failure therefore remains experimentally unvalidated. As these results are based on single executions, variance cannot be estimated, and no claims regarding robustness or consistency are made.

B. Reference Context

Table III places AgentSentinel's measured MTTR alongside the three illustrative operational references described in Section IV-D.

TABLE III
ILLUSTRATIVE REFERENCE CONTEXT (NOT MATCHED CONDITIONS)

Approach	Reference MTTR	Human Required
Manual (Human Operator)	∼14 min	Yes
Rule-Based Scripts	∼4.5 min	Partial
Monitoring Only	N/A	Yes
AgentSentinel (ours)	**5.58 s**	**No**

The observed MTTR is substantially lower than both reference points. However, these figures are not directly comparable: the reference values derive from published industry benchmarks rather than controlled measurements, and AgentSentinel's numbers are based on single-run results from a minimal four-service testbed. The table is provided solely as illustrative context for the timescales involved, not as a controlled performance comparison.

C. Discussion

The five evaluated scenarios span qualitatively distinct fault classes — isolated service failures, cascading multi-service failures, resource exhaustion, and network degradation — and the same three-agent pipeline handled all of them without fault-specific customization. This generality — rather than the raw timing numbers — is what we consider the main architectural takeaway from the evaluation.

The database connection failure scenario produced the longest recovery time (7.2 s). This is attributable to the structural position of the database at the base of the dependency graph: its failure caused simultaneous error manifestation across three dependent services, requiring the Reasoning Agent to identify a shared upstream cause rather than treating each downstream service independently. That the agent reached the correct diagnosis in a single iteration suggests the dependency graph is providing meaningful structural context, though a single trial is insufficient to confirm this reliably.

The memory leak (4.4 s) and network timeout (4.9 s) scenarios produced the fastest recoveries, consistent with their relatively unambiguous log signatures and straightforward remediation actions (container restart and cache flush, respectively). The service crash (5.1 s) and CPU spike (6.3 s) required moderately longer recovery; the CPU spike in particular necessitates horizontal scaling rather than a simple restart, which requires additional reasoning steps.

Collectively, the results support the architectural hypothesis that combining rapid anomaly detection, LLM-based reasoning with structural dependency context, and automated repair in a closed loop is a viable approach to autonomous infrastructure management within a controlled testbed. These results constitute a proof-of-concept demonstration; more extensive experimentation is required before stronger empirical claims can be made.

D. Role of the Dependency Graph

A notable observation during evaluation was the apparent contribution of the service dependency graph to diagnostic accuracy. Without structural context, the LLM must infer service relationships from log content alone, which is unreliable when multiple services exhibit simultaneous error symptoms. With the dependency subgraph injected into the reasoning prompt, the agent has explicit knowledge of upstream and downstream relationships for the affected service. This was most apparent in the database failure scenario, in which log evidence alone suggested three independent failures; the graph enabled the agent to correctly identify the single shared upstream dependency as the root cause. A formal graph-enabled versus graph-disabled ablation, quantifying how much the graph actually improves root cause attribution, remains an open task that we consider essential before stronger claims can be made.

VI. Threats to Validity

A. Internal Validity

Each fault scenario was executed once. Single-run results cannot establish whether observed outcomes are consistent or

attributable to favorable conditions in a specific trial. Statistical confidence intervals cannot be computed. Furthermore, faults were injected individually in a controlled environment, providing distinct signatures. Real production faults frequently co-occur and produce overlapping, ambiguous signals that are substantially more challenging to diagnose. The single-iteration success rate reported here should be interpreted as a feasibility indicator, not an empirical accuracy estimate.

B. External Validity

The testbed comprises four services on a single local machine. Production environments typically involve hundreds of services, dynamic autoscaling, continuously evolving topologies, and orders of magnitude greater log volume. Whether the scoring thresholds, dependency graph approach, and RILFM memory mechanism generalize to production scale has not been investigated, and no such generalization is claimed in this work.

C. Construct Validity

MTTD and MTTR are measured directly within the experimental environment and are internally consistent for this setup. The human operator and rule-based script reference values are drawn from published industry reports [1] and were not measured under conditions matched to this evaluation. They serve as illustrative context only; no speedup claim relative to these references is asserted as a controlled finding.

D. RILFM Evaluation

The RILFM mechanism is fully implemented and described, but its effectiveness has not been empirically validated. A rigorous evaluation would require repeated injection of the same fault with and without prior memory records and comparison of diagnostic accuracy and resolution time. Of all the outstanding experimental tasks, we regard this one as the most important.

VII. Conclusion

This paper presented AgentSentinel, a multi-agent framework for autonomous fault detection, root cause analysis, and repair in distributed systems. The Observer Agent monitors service metrics using a weighted multi-signal scoring model to identify anomalies. The Reasoning Agent applies an LLM with the ReAct pattern and a service dependency graph to determine the actual root cause, as distinct from downstream symptoms. The Repair Agent validates proposed actions against an allowlist of reversible primitives and executes them on the live system. The RILFM mechanism persists resolved incident records to accelerate diagnosis of recurring faults.

The core design principle is that detection alone is insufficient, and reasoning without actuation is insufficient; all three stages — detect, diagnose, repair — must be integrated in a closed loop with a memory of past resolutions. Existing tools in this space address one or two of these stages but not all four.

Evaluation on five fault types in a Docker Compose testbed demonstrated that the full pipeline completed successfully for all scenarios, with a mean MTTD of 1.98 s and a mean MTTR of 5.58 s and no false positives during fault-free monitoring. The results are treated as a proof of concept demonstration in a controlled feasibility setting rather than a definitive empirical finding.

Our immediate next steps are empirical: repeating each scenario 20–30 times to obtain statistically reliable estimates of MTTD, MTTR, and success rate; measuring RILFM directly through with-memory versus without-memory comparisons on recurring faults; and running a graph-enabled versus graph-disabled ablation of the dependency graph. Beyond that, we plan to broaden the fault taxonomy with scenarios such as disk exhaustion, configuration corruption, and dependency version conflicts; scale the testbed to a Kubernetes deployment with a larger service count; replace the JSON memory store with a vector database to improve retrieval quality; extend the Repair Agent with rollback and canary deployment capabilities; and compare AgentSentinel against other LLM-based AIOps approaches under matched experimental conditions.

Acknowledgment

The authors gratefully acknowledge the vision, support, and guidance of the IEEE Industrial Electronics Society in conducting the Generative AI Hackathon under the leadership of Daswin De Silva and Lakshitha Gunasekara. The authors also thank the anonymous reviewers for their constructive feedback, which substantially improved this manuscript.

References

[1] D. Patterson, A. Brown, P. Broadwell, G. Candea, M. Chen, J. Cutler, P. Enriquez, A. Fox, E. Kiciman, M. Merzbacher, D. Oppenheimer, N. Sastry, W. Tetzlaff, J. Traupman, and N. Treuhaft, "Recovery-oriented computing (ROC): Motivation, definition, techniques, and case studies," UC Berkeley Technical Report UCB/CSD-02-1175, 2002.

[2] 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. International Conference on Learning Representations (ICLR)*, 2023.

[3] J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, E. Chi, Q. Le, and D. Zhou, "Chain-of-thought prompting elicits reasoning in large language models," in *Advances in Neural Information Processing Systems*, vol. 35, 2022.

[4] M. Du, F. Li, G. Zheng, and V. Srikumar, "DeepLog: Anomaly detection and diagnosis from system logs through deep learning," in *Proc. ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pp. 1285–1298, 2017.

[5] H. Guo, S. Yuan, and X. Wu, "LogBERT: Log anomaly detection via BERT," in *Proc. International Joint Conference on Neural Networks (IJCNN)*, pp. 1–8, 2021.

[6] Y. Liu, X. Ge, K. Tantithamthavorn, and C. Li, "LogPrompt: Prompt engineering towards zero-shot and interpretable log analysis," *arXiv preprint arXiv:2308.07610*, 2023.

[7] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman et al., "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[8] Q. Wu, G. Bansal, J. Zhang, Y. Wu, B. Li, E. Zhu, L. Jiang, X. Zhang, S. Liu, R. Awadallah, and C. Wang, "AutoGen: Enabling next-gen LLM applications via multi-agent conversation," *arXiv preprint arXiv:2308.08155*, 2023.