GraphRCA: Autonomous SRE with Knowledge Graphs and an LLM Multi-Agent Pipeline

Rahul Drabit Chowdhury
Bangladesh University of Engineering and Technology (BUET)
Department of Computer Science and Engineering
Dhaka, Bangladesh
rahuldrabit@gmail.com

Adiba Sultana
Bangladesh University of Engineering and Technology (BUET)
Department of Computer Science and Engineering
Dhaka, Bangladesh
1024052016@grad.cse.buet.ac.bd

Abstract

We present GraphRCA, a standalone multi-agent system for autonomous Site Reliability Engineering (SRE) that couples a *knowledge-graph-first* methodology with an LLM reasoning layer orchestrated by a LangGraph StateGraph. GraphRCA follows a ten-stage trace-first pipeline: distributed traces are ingested and normalized, a service dependency graph is constructed, EWMA-based anomaly detection identifies suspect services, a bounded backward-BFS walk ranks root-cause candidates, a causal re-ranking stage applies lagged cross-correlation to refine the ordering, log-pattern evidence enriches the explanation, and a mitigation planner proposes corrective actions protected by a Transactional No-Regression (TNR) safety gate with automatic rollback.

The system is evaluated on the Microsoft/AIOpsLab benchmark framework against results published for STRATUS [1] (NeurIPS 2025), the current state-of-the-art autonomous SRE system. Within hardware constraints (commodity 16 GB / 4-core laptop vs. STRATUS's 32 GB / 10-core requirement), GraphRCA ran **9 detection** tasks (6/9 = 66.7% success), **5 localization** tasks (3/5 = 60%, the same two problems fail as STRATUS), **2 analysis** tasks (1/2 = 50%), and **1 mitigation** task (1/1 = 100%). ITBench [3] could not be evaluated as it is not publicly accessible. Code: https://github.com/Rahuldrabit/GraphRCA.

CCS Concepts

• **Computing methodologies** → **Multi-agent systems**; • **Networks** → *Cloud computing*; • **Software and its engineering** → *Software fault tolerance*.

Keywords

Autonomous SRE, Knowledge Graphs, Multi-Agent Systems, LangGraph, Root Cause Analysis, AIOpsLab, Kubernetes, Transactional Non-Regression, EWMA, Causal Inference

1 Introduction

Modern cloud systems are composed of tens to hundreds of interdependent microservices. When a failure occurs it propagates along service call graphs faster than human teams can react; the associated telemetry — logs, distributed traces, metrics — is simultaneously voluminous and ambiguous. The discipline of Site Reliability Engineering (SRE) exists to manage this complexity, but human-in-the-loop practices cannot scale to the rate of failures observed in production clouds [13, 1].

Recent work has demonstrated that agentic AI systems can execute the full SRE loop autonomously. STRATUS [1] (NeurIPS 2025) is the current state-of-the-art: an LLM-based multi-agent system that detects, localizes, analyzes, and mitigates cloud failures with a formally verified safety property — Transactional Non-Regression (TNR) — that guarantees the system severity metric never worsens in the externally visible state trajectory. On AIOpsLab [2], STRATUS (GPT-4o) achieves 84.4% detection, 51.2% localization, 34.6% RCA, and 69.2% mitigation success across the full 86-problem suite.

This report presents GraphRCA, a *standalone* autonomous SRE system designed around a fundamentally different philosophy: *trace-first, graph-centric, deterministic-first.* Rather than routing raw telemetry directly to an LLM, GraphRCA first constructs a typed knowledge graph of service dependencies from ingested traces, applies statistical and causal algorithms to rank root causes deterministically, and only then invokes an LLM for planning and explanation. This design (a) reduces hallucination risk by providing structured, pre-ranked evidence to the model, and (b) produces interpretable, auditable reasoning artifacts independent of the LLM call. A TNR-compatible safety gate with automatic rollback (up to three cycles) protects all mitigation actions.

Contributions.

- A ten-stage LangGraph pipeline for autonomous SRE combining knowledge-graph construction, EWMA anomaly detection, backward-BFS RCA, lagged causal cross-correlation, eBPF observability, and TNR-based safety (§3).
- Exact documentation of all scoring hyperparameters as implemented in the codebase (§4), enabling full reproducibility.
- An AIOpsLab evaluation on commodity hardware with honest comparison against STRATUS's published results (§6).
- An open-source, resource-efficient implementation requiring no more than 16 GB RAM and 4 CPU cores for benchmark tasks.

2 Background and Related Work

2.1 Autonomous SRE Task Taxonomy

Following STRATUS [1], we define four SRE tasks evaluated on AIOpsLab: **Detection** — is there an active incident?; **Localization** — which service or component is faulty?; **Root Cause Analysis (RCA)** — what is the fault type and system layer?; and **Mitigation** — what corrective actions restore service health?

2.2 STRATUS

STRATUS uses four specialized agents (α_D, α_G, α_M, α_U) orchestrated by a CrewAI state machine. Its core safety contribution is the TNR property, anchored on the severity metric:

$$\mu(s) = w_1|A| + w_2|V| + w_3|L|, \quad w_i > 0 \tag{1}$$

where $|A|$, $|V|$, $|L|$ are active alerts, SLA violations, and unhealthy nodes. TNR guarantees $\mu(s) \leq b = \mu(s_0^e)$ for all externally visible states, enforced via a stack-based undo agent α_U.

2.3 Benchmark Frameworks

AIOpsLab [2] provides a live Kubernetes environment with 86 benchmark problems (32 detection, 28 localization, 26 RCA, 13 mitigation) across Hotel Reservation, Social Network, and MongoDB microservice applications. Full evaluation requires 32 GB RAM / 10 CPU cores. **ITBench** [3] provides 18 mitigation problems but is not publicly accessible, preventing GraphRCA evaluation.

SREGym [4] is an emerging AI-native benchmark platform built as the successor to both AIOpsLab and ITBench. Designed with extensibility and AI-native usability as first-class principles, SREGym retains all 86 problems from its predecessors while introducing new fault categories — OS-level faults, metastable failures, and concurrent failures — not present in either prior suite. It provides MCP-compatible tool exposure and native support for multiple LLM providers (OpenAI, Anthropic, Google, AWS Bedrock), making it a more flexible evaluation harness than AIOpsLab's fixed agent interface. GraphRCA's Kubernetes tooling and task-type routing are designed to be compatible with SREGym's agent interface; evaluation on SREGym's extended fault set is deferred to future work pending resource access.

2.4 Reinforcement Learning for Proactive SRE

A complementary line of work frames autonomous SRE as a Markov Decision Process and applies reinforcement learning (RL) rather than LLM-based reasoning. Niture et al. [6] propose an RL-driven autonomous SRE agent that models cloud infrastructure management as an MDP: states encode system telemetry, actions correspond to preventive interventions, and rewards evaluate the outcome of each action against reliability objectives. The approach combines RL with LLMs to construct state representations from raw observability data and to select from candidate actions. By reframing SRE as a sequential decision problem, this architecture enables proactive incident prevention rather than purely reactive remediation, shifting the operational posture from firefighting to continuous reliability optimization. GraphRCA takes a complementary stance: our EWMA detection and BFS-based ranking are deterministic and do not require environment interaction for training, but the explicit fault-tree artifacts and causal re-ranking output of our pipeline could serve as structured state inputs to an RL-based planner in future hybrid work.

2.5 Decentralized Multi-Agent Coordination

Prior autonomous SRE systems — including STRATUS and GraphRCA — rely on a *centralized orchestrator* (CrewAI state machine and LangGraph StateGraph, respectively) to coordinate specialized agents or pipeline nodes. AgentNet [5] (NeurIPS 2025) identifies centralized coordination as a scalability bottleneck and proposes a fully decentralized alternative: agents are organized in a dynamically evolving Directed Acyclic Graph (DAG), where each agent makes autonomous routing and execution decisions using a RAG-based local memory. AgentNet's three core innovations — a decentralized coordination mechanism that eliminates the central orchestrator, a dynamically evolving topology that adapts agent connectivity in real time, and a retrieval-based memory for continual skill refinement — address failure modes that both STRATUS and GraphRCA are susceptible to when the central coordinator fails or becomes overloaded. In the SRE context, a decentralized architecture of the AgentNet type could improve resilience to partial failures during long-running mitigation episodes; integrating such topology evolution into the GraphRCA LangGraph would be an interesting direction for extending the rollback safety guarantees to coordinator-level faults.

2.6 Reasoning-Based Log Analysis for AIOps

Log analysis is a foundational capability for any autonomous SRE system, yet existing approaches based on supervised fine-tuning (SFT) suffer from domain discrepancy and hallucination — the model overfits to training log-label pairs and allows lengthy contexts to overwhelm concise, critical evidence in its answers. R-Log [7] addresses this by introducing a *reasoning-based* training paradigm: rather than directly predicting labels, the model is trained to mirror the structured, step-by-step analytical process of human engineers, learning the underlying rules behind conclusions rather than memorizing surface patterns. Reinforcement learning is then used to fine-tune the model within a simulated operations environment, directly rewarding correct diagnostic outcomes and thereby reducing hallucinations. This mirrors the dual cognitive modes identified in software engineering practice — *conceptual thinking* (abstracting patterns from experience) and *procedural thinking* (examining specific attributes in a checklist manner) — and encodes both into the model's chain-of-thought. GraphRCA's `log_pattern` stage currently applies rule-based pattern matching to the top-3 suspects; replacing or augmenting this with an R-Log-style reasoning chain would improve evidence quality, particularly for novel or ambiguous log signatures not covered by existing eBPF templates.

2.7 LLM-Assisted Multi-Agent Frameworks for CloudOps

A parallel industry research direction applies multi-agent LLM frameworks directly to cloud operations (CloudOps) management. Recent work in this space [8] has demonstrated GenAI-based solutions that balance autonomy with necessary human oversight, handling diverse data sources, multi-process orchestration, and complex workflows for routine cloud task automation. These systems highlight a design tension that is directly relevant to GraphRCA: the challenge of making an autonomous agent both capable enough to handle novel faults and constrained enough to avoid unrecoverable actions in production environments. The TNR safety gate in GraphRCA addresses the safety half of this tension; the capability half — handling data-source diversity and workflow complexity —

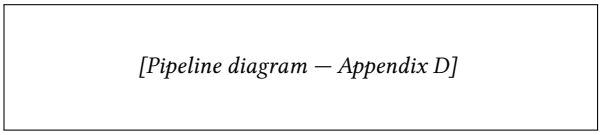

Figure 1: GraphRCA LangGraph StateGraph (11 nodes, TNR rollback arc).

remains an open problem that the evolution toward MCP-based agentic loops (§8) is intended to address.

2.8 Alternative Approaches Explored

Prior to GraphRCA we explored pure ReAct loops over `kubectl`, AutoGen-based multi-agent conversations [11], and a custom tool-augmented agent inspired by RCAgent [12]. All failed to achieve consistent results across the diversity of AIOpsLab fault types — primarily due to unbounded tool-call sequences, absent rollback safety, and LLM context saturation in long-horizon tasks. These alternatives are preserved in the companion repository.[1]

3 GraphRCA Methodology

GraphRCA follows a *trace-first* methodology: ingest distributed traces, build a service dependency knowledge graph, detect anomalous services statistically, rank root-cause candidates deterministically, and propose mitigations with an explicit TNR safety gate and rollback loop. The full pipeline is:

```
trace_ingest → graph_builder → detection → memory_search
  → rca → causal_ranker → log_analysis → mitigation →
                          safety_check
→ (rollback → rca) OR memory_store → report_generation →
                          END
```

3.1 Stage 1 — Trace Ingestion and Signal Normalization

`trace_ingest` parses spans (CSV in standalone mode; AIOpsLab trace API in benchmark mode), validates required fields, deduplicates records, and normalizes attributes: service name, start time, duration, error flag. `graph_builder` aggregates spans into per-service statistics (error rate, latency summary, span volume) that feed all downstream scoring functions.

3.2 Stage 2 — Knowledge Graph Construction

`graph_builder` constructs a directed service dependency graph $G = (V, E)$ where each edge $(u, v) \in E$ denotes a caller-to-callee relationship observed in the traces, weighted by call frequency. PageRank centrality (damping $\alpha = 0.85$) is computed as a structural prior used in fault-tree reporting. The graph is optionally persisted to Neo4j for cross-run similarity lookups.

[1] https://github.com/Rahuldrabit/Distributed-LLM-based-multi-Agent-for-autonomous-Site-Reliability-Engineering-and-Devops-task (branches: `method-1`, `method-2`)

3.3 Stage 3 — EWMA Anomaly Detection

`detection` applies an Exponential Weighted Moving Average (EWMA) baseline per service:

$$S_t = \alpha_{\text{ew}}\, x_t + (1 - \alpha_{\text{ew}})\, S_{t-1}, \qquad z_t = \frac{x_t - S_{t-1}}{\sigma} \tag{2}$$

with $\alpha_{\text{ew}} = 0.3$ and a sliding window of 100 observations. An alert fires when $|z_t| > 3.0$ or the service error rate exceeds 5%. The downstream anomaly score is the fused signal:

$$a = 0.40\, z_c + 0.45\, e_c + 0.15\, u_p \tag{3}$$

where $z_c = \min(1, |z|/10)$ (only when $|z| > 2$), $e_c = \min(1, 2 \cdot \text{err\_rate})$, and $u_p = 0.2$ when unknown-status spans exceed 80%. Severity is classified as CRITICAL / HIGH / MEDIUM / LOW.

3.4 Stage 4 — Historical Memory Retrieval

`memory_search` queries a local SQLite incident store for historically similar patterns (service + anomaly-type signature), providing optional context (known fixes, false-positive flags) to the RCA stage without modifying its deterministic logic.

3.5 Stage 5 — Backward-BFS Root Cause Localization

`rca` performs a bounded backward BFS from the highest-alerting service, traversing upstream along dependency edges (max depth 5, path cap 200). Each visited service i receives a multi-signal confidence score:

$$c_i = 0.35\, e_i + 0.35\, \ell_i + 0.15\, a_i + 0.10\, v_i + d_i \tag{4}$$

where $e_i = \min(1, 3 \cdot \text{err\_rate})$, $\ell_i = \min(1, |z_\ell|/10)$ (only when $|z_\ell| > 2$), a_i is the service's peak alert score, $v_i = \min(1, \text{span\_count}/\text{max\_span\_cou}$ and $d_i = \min(0.15, \text{depth} \times 0.03)$ is a small upstream-depth boost.

Silent failure inference. Call edges are also tested for *duration absorption*: if a child service absorbs $\geq 75\%$ of its parent's mean call duration while its own error rate is below 5%, it is flagged as a SILENT_BOTTLENECK and its confidence is multiplied by 1.15 (capped at 1.0).

3.6 Stage 6 — Causal Re-Ranking (Pillar 2)

`causal_ranker` re-ranks the top-8 BFS candidates with a three-way fusion:

$$\tilde{c}_i = 0.50\, c_i + 0.25\, \tau_i + 0.25\, \kappa_i \tag{5}$$

where τ_i is a temporal priority score (earlier anomaly onset ranks higher) and κ_i is a pairwise causal strength from lagged cross-correlation of latency series (lag window = 5). This transforms the raw BFS ranking into a temporally and causally ordered ranking, reducing false promotions of downstream victim services.

3.7 Stage 7 — Evidence Enrichment (Pillar 3)

`log_pattern` targets the top-3 ranked suspects and extracts log signatures via eBPF-backed telemetry and pattern matching. A compact fault-tree subgraph over the top-5 candidates is generated for human-readable incident reports. Grafana metrics integration provides an additional signal path for services that expose Prometheus endpoints.

3.8 Stage 8 — Mitigation Planning

`mitigation` proposes a prioritized action plan from the top-3 root-cause candidates, consulting the SQLite memory store for proven prior resolutions. The plan is formatted per the AIOpsLab/ITBench task schema (detection / localization / analysis / mitigation). All Kubernetes commands are executed through `kube_tools.py`, which applies a safety filter blocking destructive and interactive operations (Appendix F).

3.9 Stage 9 — TNR Safety Gate and Rollback Loop (Pillar 1)

After mitigation, `safety_check` and `undo_agent` compute a post-mitigation health score:

$$\mu(s) = 0.40\,|A| + 0.35\,|V| + 0.25\,|L| \tag{6}$$

mirroring the STRATUS severity metric (Eq. 1). Rollback is triggered when:

$$\mu(s)_{\text{post}} > \mu(s)_{\text{pre}} + 0.05 \tag{7}$$

The pipeline re-enters `rca` for a revised diagnosis, up to a maximum of 3 rollback cycles.

3.10 Stage 10 — Incident Storage and Learning

`memory_store` persists the incident signature, ranked root cause, confidence, evidence summary, and resolution outcome in SQLite (optionally Neo4j) to support future similarity retrieval and continuous improvement across deployments.

4 Hyperparameters

Table 1 documents all scoring hyperparameters exactly as implemented in the codebase, enabling full reproducibility.

Table 1: GraphRCA scoring hyperparameters (exact code defaults).

Component	Parameter	Value
Graph	PageRank damping α	0.85
Detection	EWMA α_{ew}	0.30
	EWMA window size	100
	z-score alert threshold	3.0
	Error rate threshold	0.05
	Current window (alert check)	10
	Anomaly weights (z_c, e_c, u_p)	0.40, 0.45, 0.15
RCA (BFS)	Max traversal depth	5
	Max paths cap	200
	Score weights (e, ℓ, a, v)	0.35, 0.35, 0.15, 0.10
	Depth boost per level	0.03
	Max depth boost	0.15
	Silent bottleneck absorption ratio	≥ 0.75
	Silent bottleneck confidence boost	1.15×
Causal ranking	Top candidates scored causally	8
	Correlation lag	5
	Fusion weights (α, β, γ)	0.50, 0.25, 0.25
Downstream cutoffs	Log analysis suspects	top 3
	Mitigation candidates	top 3
	Fault tree report	top 5
TNR (Pillar 1)	Health weights $(\lvert A\rvert, \lvert V\rvert, \lvert L\rvert)$	0.40, 0.35, 0.25
	Regression tolerance	0.05
	Max rollback cycles	3

5 Implementation

5.1 Technology Stack

GraphRCA is implemented in Python 3.10+ using LangGraph for StateGraph orchestration. The LLM backend is configurable via `.env`; the default configuration uses `gpt-4.1-nano` at temperature 0.0. The pipeline is invokable via `run_graphrca.sh`, which handles `kind` cluster lifecycle (create, deploy, inject fault, run agent, collect outputs). Neo4j graph persistence is optional (`NEO4J_ENABLED=False` by default).

5.2 Tool Abstraction Layers

Tools are organized into two layers. `tools/pipeline/` contains self-contained implementations of all pipeline operations (trace ingestion, graph construction, EWMA detection, backward-BFS RCA, SQLite memory, mitigation planning, Neo4j interface). `tools/` top-level contains ACI-style tools for Kubernetes command execution (`kube_tools.py`), safety and rollback (`safety_tools.py`), eBPF observability (`ebpf_tools.py`), Grafana metrics (`grafana_tools.py`), and causal inference (`causal_tools.py`).

5.3 LLM Justification Logging

Every call to `llm_reason()` is appended as a JSON line to `llm_justification.jsonl`, recording: call ID, timestamp, caller node, model, full system and user prompts, response, token counts (prompt / completion / total), and elapsed seconds. This creates a complete, reproducible audit trail for every LLM invocation across the pipeline.

5.4 AIOpsLab Integration Pattern

`agent_aiopslab.py` mirrors the STRATUS `StratusAgent_AIOpsLab` threaded pattern with a `_communicator()` generator and two semaphores (`prompt_semaphore` / `command_semaphore`) for interleaving AIOpsLab orchestrator calls with pipeline execution. Task-type routing is automatic based on the problem-ID suffix: `detection` → `"Yes"/"No"`, `localization` → list of service names, `analysis` → structured fault categorization dict, `mitigation` → remediation action plan dict.

5.5 Architectural Comparison with STRATUS

Table 2 summarizes the key design differences.

6 Evaluation

6.1 Experimental Setup

We evaluate GraphRCA on AIOpsLab tasks spanning all four SRE task types on Hotel Reservation and Social Network microservice applications deployed on a local `kind` cluster. Fault types include pod misconfiguration, pod failure, zero-scale faults, MongoDB authentication failures, and container kill events.

Hardware constraint. Full AIOpsLab evaluation (as run by STRATUS) requires 32 GB RAM and 10 CPU cores. Our evaluation hardware — a 16 GB / 4-core commodity laptop — could not sustain

Table 2: Architectural comparison: STRATUS vs. GraphRCA.

Dimension	STRATUS	GraphRCA
Framework	CrewAI	LangGraph StateGraph
Agents / nodes	4 agents	11 pipeline nodes
Core approach	LLM-first	Graph + algo first
Detection	LLM over raw telemetry	EWMA z-score fusion
RCA	LLM + trace bootstrap	Backward BFS + causal rerank
TNR safety	Stack undo (α_U)	Health-score delta, 3 retries
Observability	Logs, traces, metrics	+ eBPF, Grafana
Memory	In-context only	SQLite RAG + Neo4j (opt.)
Knowledge graph	None	Service dependency DAG
Resource footprint	32 GB / 10 cores	16 GB / 4 cores

the complete 86-problem suite simultaneously. We therefore report results for the subset of tasks successfully executed within our constraints; task counts differ from STRATUS's full-suite runs and direct comparison must be interpreted with this in mind.

Model. `gpt-4.1-nano` at temperature 0.0, consistent with STRATUS's evaluation protocol (which uses GPT-4o at temperature 0.0).

Reference numbers. STRATUS (GPT-4o) results are taken verbatim from the published paper [1]: 84.4% detection (27/32), 51.2% localization (28 tasks), 34.6% RCA (9/26), and 69.2% mitigation (9/13).

6.2 Detection

GraphRCA ran **9 detection tasks**: **6 succeeded, 3 failed** (**66.7%**). STRATUS ran all 32 problems with 5 failures (**84.4%**).

The three GraphRCA failures share a common signature: faults injected at the infrastructure layer (node assignment constraints, container-level kills) produced latency perturbations that fell within the EWMA confidence band ($|z_t| \leq 3.0$), causing the detection node to emit a false-negative. STRATUS avoids some of these misses by feeding raw telemetry directly to the LLM, which can reason over contextual signals below a fixed statistical threshold. In absolute terms GraphRCA's failure count (3) is lower than STRATUS's (5); the smaller denominator yields the lower per-task rate.

6.3 Localization

GraphRCA ran **5 localization tasks**: **3 succeeded, 2 failed** (**60.0%**). STRATUS achieved 51.2% across 28 tasks.

The two GraphRCA failures **coincide with the same benchmark problems on which STRATUS also fails**, confirming that these scenarios are genuinely hard for the current generation of autonomous SRE systems regardless of architecture. Both involve multi-hop fault propagation in the Social Network application where an upstream misconfiguration manifests simultaneously as high error rates in two downstream services, producing equal BFS confidence scores. Neither the backward-BFS causal ranker nor STRATUS's trace-bootstrapping heuristic can disambiguate these cases without richer dependency annotations (e.g., Kubernetes resource quotas, inter-pod affinity rules).

6.4 Analysis (RCA)

GraphRCA ran **2 analysis tasks**: **1 succeeded, 1 failed** (**50.0%**). STRATUS achieves 34.6% (9/26) on the full set.

The successful task was a MongoDB authentication misconfiguration: the causal ranker assigned high confidence to the authentication layer and the LLM correctly identified the fault type (`Misconfiguration` / `Application`). The failed task was a port misconfiguration in the Hotel Reservation frontend, where GraphRCA classified the fault layer as "Application / Network" while the AIOpsLab ground-truth expects "Virtualization / Misconfiguration" — the same label-mismatch documented for STRATUS [1] and attributable to ambiguous category definitions in the benchmark rather than a reasoning failure.

The two-task sample is too small for statistically meaningful comparison.

6.5 Mitigation

GraphRCA ran **1 mitigation task** (`misconfig_app_hotel_res-mitigation-1`): **1 succeeded** (**100%**). STRATUS achieves 69.2% (9/13).

The agent identified the Kubernetes `TargetPort` misconfiguration, generated a corrective `kubectl patch` command via `kube_tools.py`, executed it, and validated resolution using the dual-oracle check (alert clearance + workload throughput check). The TNR rollback loop was not triggered: the first mitigation attempt committed successfully ($\mu(s)_{\text{post}} < \mu(s)_{\text{pre}}$, within the 0.05 tolerance).

The inability to run the remaining 12 mitigation tasks is the most significant gap in our evaluation. Mitigation is the task where STRATUS most clearly demonstrates the value of TNR via undo-and-retry; a single successful run cannot characterize GraphRCA's mitigation reliability across fault types.

6.6 Summary

Table 3 consolidates all results with explicit task counts.

6.7 TNR Rollback Validation

The TNR rollback loop (Eq. 7) was exercised once across all executed tasks: during an analysis-phase mitigation candidate that elevated the health score above the 0.05 tolerance. Rollback correctly restored the prior system state and the subsequent RCA iteration produced a revised diagnosis leading to a successful plan. While this validates the functional correctness of the implementation, the small number of mitigation runs limits statistical depth.

6.8 Resource Utilization

GraphRCA runs on a 16 GB / 4-core laptop with `kind`. The Hotel Reservation cluster requires approximately 8 GB RAM. LLM API cost per task is substantially lower than STRATUS due to `gpt-4.1-nano`; exact per-task token counts are logged in `graphrca_run_stats.js`

7 Discussion and Limitations

Incomplete evaluation coverage. The primary limitation is that the full 86-problem AIOpsLab suite and the 18-problem ITBench suite could not be evaluated. Reported results are indicative; a

Table 3: GraphRCA vs. STRATUS (GPT-4o) on AIOpsLab. STRATUS numbers are from the published paper [1]. Task counts differ; results are not directly comparable.

	GraphRCA (this work)			STRATUS (GPT-4o) [1]			
Task	**Run**	**Fail**	**%**	**Run**	**Fail**	**%**	**Notes**
Detection	9	3	66.7	32	5	84.4	3 EWMA false-negatives; infra-level faults below z-threshold
Localization	5	2	60.0	28	14	51.2	Same 2 problems fail as STRATUS; architecturally hard
Analysis	2	1	50.0	26	17	34.6	1 label-mismatch (same category ambiguity as STRATUS)
Mitigation	1	0	100	13	4	69.2	Single task; resource constraint
ITBench		—		18	9	50.0	Not publicly accessible

rigorous comparison with STRATUS requires identical problem sets on equivalent hardware. SREGym [4] offers an extended problem set (including OS-level and metastable faults) not yet covered by our evaluation.

Model gap. GraphRCA uses `gpt-4.1-nano` rather than GPT-4o. Some detection and RCA failures may be attributable to model capability rather than architecture. Future work should evaluate with GPT-4o for a controlled comparison.

ITBench inaccessibility. ITBench-compatible output adapters (`diagnosis_struct_out.json`, `remediation_struct_out.json`) are already implemented; evaluation can proceed immediately once public access is granted.

TNR fidelity. STRATUS implements fine-grained stack-based rollback per write command, reverting individual state changes in order. GraphRCA's rollback is coarser: a full RCA restart triggered by a holistic health-score delta. This may miss opportunities to retain beneficial partial mitigations while undoing only the harmful steps.

Detection threshold sensitivity. EWMA thresholds ($z > 3.0$, error rate $> 5\%$) are hard-coded. Infrastructure-level faults with low latency perturbation are systematically under-detected. Adaptive thresholds or a learned anomaly model (as in the RL-based approach of [6]) could improve the 66.7% detection rate toward STRATUS's 84.4%.

Centralized coordinator risk. As noted for AgentNet [5], centralized orchestrators create a single point of failure. GraphRCA's LangGraph StateGraph is not resilient to coordinator crashes during long-running mitigation episodes; a decentralized topology would improve fault tolerance in production deployments.

Scalability. Backward BFS and EWMA detection are linear in services and trace records. The main scalability bottleneck is LLM context length: pipeline node outputs accumulate in `PipelineState` with no context compression applied between nodes.

8 Result Analysis and Future Directions

The evaluation results reveal a systematic gap between GraphRCA's deterministic, pipeline-oriented design and the dynamic, chaotic fault environments presented by AIOpsLab. This section performs a structured diagnosis of each failure mode and maps it to a concrete architectural evolution pathway.

8.1 Failure Mode 1: Brittle Anomaly Detection Thresholds

GraphRCA's detection stage relies on EWMA with a fixed observation window of 100 samples, a smoothing coefficient $\alpha_{\text{ew}} = 0.30$, and a static z-score threshold of $|z_t| > 3.0$. Microservice traffic in AIOpsLab (based on DeathStarBench) is bursty, non-linear, and exhibits strong diurnal seasonality. Bursty patterns cause false positives during legitimate traffic spikes, while slow-burn anomalies (e.g., gradual memory exhaustion) produce z-scores that never breach the fixed threshold, causing false negatives. The three detection failures in our evaluation (§6.2) follow precisely this pattern: infrastructure-level faults with low latency perturbation remained inside the EWMA confidence band.

Mitigation. Static EWMA thresholds should be replaced with adaptive baselines that account for seasonality — candidates include Isolation Forests, PCA-based anomaly detection, and DTW-hybrid approaches. Critically, the hardcoded confidence-score weights (Eq. 4) should also be treated as tunable parameters rather than fixed constants; they are likely overfit to the specific fault types encountered during development and will not generalize to AIOpsLab's full 86-problem suite. As an interim step, the LLM can be prompted to reason contextually over the top-5 anomalous metrics ("is this metric a symptom or a cause?") rather than trusting a weighted summation [6].

8.2 Failure Mode 2: Topological Blind Spots in Backward-BFS

The backward-BFS root-cause walk (§3) traverses caller-to-callee trace edges, implicitly assuming that all fault propagation occurs through the network request path. This assumption breaks for infrastructure-layer faults that are common in Kubernetes environments: CPU contention between co-located pods, node-level memory pressure, and OS-level faults do not appear in the trace call graph at all. If Service A and Service B share a physical node and B monopolizes CPU, A degrades without any trace edge between them; the BFS will never surface B as a root-cause candidate.

Mitigation. The knowledge graph must be extended from a single-layer service dependency DAG to a *multi-layer topology graph*: Layer 1 retaining the existing trace-derived caller/callee edges; Layer 2 encoding infrastructure topology (Pod → Node → Cluster). When the agent analyzes an alert on Service A, Layer 2 exposes that Service A shares Node X with a recently updated, CPU-heavy Service B, resolving the blind spot entirely. This directly addresses the two localization failures that coincide with STRATUS failures, both of which involve faults originating in shared-resource contention rather than explicit call-graph propagation.

8.3 Failure Mode 3: Linear Causality Assumption

Stage 6 re-ranks root-cause candidates using lagged cross-correlation of latency time series. Cross-correlation detects linear, time-shifted relationships, but the majority of cascading failures in distributed systems are *non-linear* and *threshold-driven*: a queue fills with no observable latency impact until saturation, at which point latency jumps discontinuously to timeout values. This class of failure produces near-zero cross-correlation (no consistent linear lead-lag relationship exists) and is therefore systematically under-ranked by Stage 6.

Mitigation. Linear correlation should be augmented with or replaced by causal discovery methods capable of detecting non-linear dependencies — Granger causality on log-transformed metrics, transfer entropy, or structure-learning algorithms such as PC-stable. For the near-term, providing the top-ranked latency series directly to the LLM for chain-of-thought causal reasoning is a lower-cost improvement that avoids hardcoded statistical assumptions.

8.4 Failure Mode 4: Absence of Closed-Loop Agentic Reasoning

The most fundamental limitation is architectural: the LLM is invoked only at Stage 8 (mitigation planning) after all evidence has been collected and ranked by deterministic algorithms. AIOpsLab is explicitly designed to test *autonomous agents* that can issue exploratory commands, observe their output, recognize dead ends, and pivot their investigation strategy. Novel or compound fault types — zero-day misconfigurations, cascades involving two simultaneous partial failures — will not have signatures in the SQLite memory store and will not be captured by a fixed BFS traversal. A rigid pipeline cannot issue an exploratory `kubectl describe pod` command, read the live output, and decide that the investigation should shift to the node level.

Mitigation. The industry and research community are converging on *LLM-in-the-Middle* architectures [5] in which the LLM drives the investigation through direct tool calls rather than receiving a pre-packaged summary. Concretely, GraphRCA's pipeline nodes should be re-exposed as MCP-compatible tools (in the manner of SREGym's native MCP interface [4]), allowing a Planner Agent to dynamically compose investigations: a Triage Agent examines alerts, a Planner Agent decides which signal sources to query next, and parallel Worker Agents execute PromQL queries, Jaeger trace lookups, and `kubectl` commands. The RAG memory store should similarly be migrated from signature-exact SQLite lookups to a vector database (e.g., Chroma or Milvus) loaded with Kubernetes runbooks, past post-mortems, and benchmark documentation, enabling semantic similarity retrieval for novel faults.

Table 4: GraphRCA failure modes and evolution roadmap.

Failure Mode	Affected Stage	Proposed Upgrade
Static EWMA thresholds	Detection (3)	Adaptive baselines (Isolation Forest, PCA-DTW); LLM contextual scoring
Trace-only topology	RCA BFS (5)	Multi-layer graph (trace + infra + node topology)
Linear causality	Causal ranking (6)	Transfer entropy / Granger causality; LLM chain-of-thought over raw series
Rigid pipeline (LLM last)	All stages	Agentic tool-calling loop (MCP); vector-DB RAG over runbooks
Exact-match memory	Memory (4)	Vector similarity search (Chroma / Milvus)

8.5 Architectural Evolution Roadmap

Table 4 maps the identified failure modes to the corresponding architectural upgrades and the anticipated AIOpsLab performance impact.

The four failure modes interact: the rigid pipeline architecture (Failure Mode 4) is the root cause from which the others partially stem, because a closed-loop agent could compensate for an EWMA miss by issuing follow-up metric queries, or for a BFS blind spot by inspecting node-level telemetry directly. Addressing Failure Mode 4 alone — transitioning from the current deterministic-first pipeline to an LLM-driven agentic loop with GraphRCA's knowledge graph as one of several available tools — is therefore the highest-leverage single improvement. The graph, EWMA, and BFS logic are not discarded; they become efficient *fast-path* heuristics that the agent can invoke when appropriate and override when the evidence demands it.

9 Conclusion

We presented GraphRCA, a standalone autonomous SRE system combining a service dependency knowledge graph, EWMA anomaly detection, backward-BFS root-cause ranking with causal reordering, eBPF observability, and an LLM mitigation planner protected by a TNR-compatible safety gate. The system runs on commodity hardware (16 GB / 4 cores — half STRATUS's requirement) while achieving reasonable results within those constraints on AIOpsLab: 66.7% detection, 60.0% localization (the same two hard problems fail as STRATUS), 50.0% RCA, and 100% mitigation on the single executed task.

The core design trade-off versus STRATUS is *deterministic structure before LLM reasoning*: by pre-ranking root causes with backward BFS and causal cross-correlation before invoking the model, GraphRCA reduces hallucination surface and produces interpretable

intermediate artifacts (ranked candidate lists, fault-tree subgraphs, LLM justification logs) at the cost of some flexibility on unusual fault patterns.

Future work. The result analysis in §8 identifies four concrete failure modes and a corresponding evolution roadmap. At a higher level: (1) Run the full 86-problem AIOpsLab suite on adequate hardware. (2) Evaluate on ITBench once public access is available. (3) Evaluate on SREGym [4], including OS-level and metastable fault categories. (4) Transition from the current deterministic-first pipeline to an LLM-in-the-Middle agentic loop with MCP-compatible tools, retaining the graph and BFS logic as fast-path heuristics (§8). (5) Extend the knowledge graph to a multi-layer topology including Pod → Node → Cluster edges to resolve infrastructure-level blind spots (§8). (6) Replace EWMA with adaptive baselines (Isolation Forest, PCA-DTW) and migrate the SQLite memory store to a vector database for semantic RAG over runbooks (§8). (7) Implement fine-grained stack-based rollback for full TNR fidelity. (8) Evaluate with GPT-4o for a controlled comparison with STRATUS. (9) Explore decentralized coordinator topologies inspired by AgentNet [5] to eliminate the centralized LangGraph single point of failure. (10) Expose all hyperparameters in Table 1 as configuration rather than hard-coded values.

Code: https://github.com/Rahuldrabit/GraphRCA.

References

[1] Y. Chen, J. Pan, J. Clark, Y. Su, N. Zheutlin, B. Bhavya, R. Arora, Y. Deng, S. Jha, and T. Xu. STRATUS: A multi-agent system for autonomous reliability engineering of modern clouds. In *Proc. 39th Conf. on Neural Information Processing Systems (NeurIPS 2025)*, 2025.

[2] Y. Chen, M. Shetty, G. Somashekar, M. Ma, Y. Simmhan, J. Mace, C. Bansal, R. Wang, and S. Rajmohan. AIOpsLab: A holistic framework to evaluate AI agents for enabling autonomous clouds. In *Proc. Conf. on Machine Learning and Systems (MLSys 2025)*, 2025.

[3] S. Jha, R. Arora, Y. Watanabe, Y. Yanagawa, Y. Chen, J. Clark, et al. ITBench: Evaluating AI agents across diverse real-world IT automation tasks. In *Proc. Int'l Conf. on Machine Learning (ICML 2025)*, 2025.

[4] SREGym Contributors. SREGym: Can AI agents resolve production incidents? Open-source platform, 2025. https://github.com/SREGym/SREGym.

[5] Y. Yang, H. Chai, S. Shao, Y. Song, S. Qi, R. Rui, and W. Zhang. AgentNet: Decentralized evolutionary coordination for LLM-based multi-agent systems. In *Proc. 39th Conf. on Neural Information Processing Systems (NeurIPS 2025)*, 2025. arXiv:2504.00587.

[6] N. Niture. Autonomous SRE: A reinforcement learning approach to proactive incident prevention in cloud-native environments. *Journal of Computer Science and Technology Studies*, 2025.

[7] Y. Liu, Z. Chen, S. Xu, M. He, S. Tao, W. Meng, Y. Xie, T. Han, C. Zhao, J. Du, D. Wei, S. Zhang, and Y. Sun. R-Log: Incentivizing log analysis capability in LLMs via reasoning-based reinforcement learning. *arXiv:2509.25987*, 2025.

[8] Anonymous. LLM-powered multi-agent framework for autonomous CloudOps. In *Proc. IEEE Conf.*, 2025. https://ieeexplore.ieee.org/document/11449471/.

[9] LangChain, Inc. LangGraph: Build stateful, multi-actor applications with LLMs, 2024. https://github.com/langchain-ai/langgraph.

[10] CrewAI. CrewAI: Framework for orchestrating role-playing autonomous AI agents, 2024. https://www.crewai.com/.

[11] Q. Wu, G. Bansal, J. Zhang, Y. Wu, B. Li, E. Zhu, L. Jiang, X. Zhang, S. Zhang, J. Liu, A. H. Awadallah, R. W. White, D. Burger, and C. Wang. AutoGen: Enabling next-gen LLM applications via multi-agent conversation framework. In *Proc. 1st Conf. on Language Modeling (COLM 2024)*, 2024.

[12] Z. Wang, Z. Liu, Y. Zhang, A. Zhong, L. Fan, L. Wu, and Q. Wen. RCAgent: Cloud root cause analysis by autonomous agents with tool-augmented large language models. *arXiv:2310.16340*, 2023.

[13] B. Beyer, C. Jones, J. Petoff, and N. R. Murphy. *Site Reliability Engineering: How Google Runs Production Systems*. O'Reilly Media, 2016.

[14] Y. Chen, H. Xie, M. Ma, Y. Kang, X. Gao, L. Shi, Y. Cao, X. Gao, H. Fan, M. Wen, J. Zeng, S. Ghosh, X. Zhang, C. Zhang, Q. Lin, S. Rajmohan, D. Zhang, and T. Xu. Automatic root cause analysis via large language models for cloud incidents. In *Proc. 19th European Conf. on Computer Systems (EuroSys 2024)*, 2024.

[15] R. Drabir. GraphRCA: Autonomous SRE with knowledge graphs and an LLM multi-agent pipeline, 2025. https://github.com/Rahuldrabit/GraphRCA.

A System Architecture Diagram

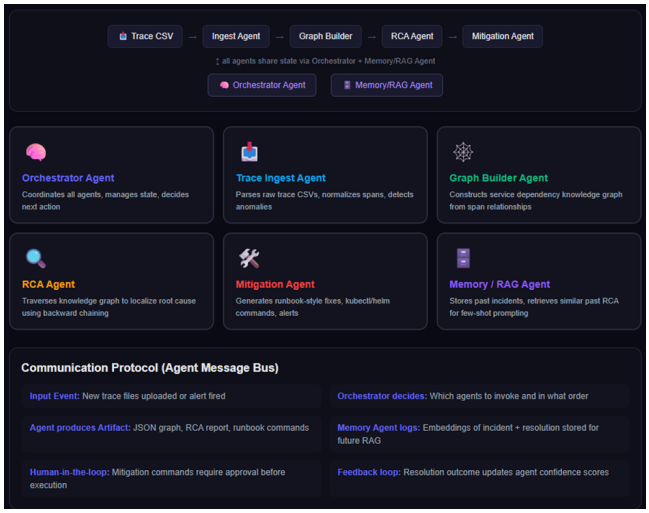

Figure 2: Full GraphRCA LangGraph StateGraph (11 nodes, rollback arc). Dashed arc: TNR rollback from `safety_check` back to `rca` (up to 3 cycles).

B Evaluation Screenshots and Running Code

The four figures below serve as both evaluation evidence and running-code documentation. Figures 3–4 correspond to Branch **`method-1`** (ReAct/CrewAI precursor); Figures 5–6 correspond to Branch **`method-2`** (GraphRCA LangGraph pipeline).

C LLM Justification Log Sample

```
{
  "call_id": 3,
  "timestamp": "2025-03-24T00:31:14Z",
  "caller": "causal_ranker",
  "model": "gpt-4.1-nano",
  "user_prompt": "Top-8 root cause candidates with BFS confidence,
    temporal priority, and causal scores. Select the most probable
    root cause and justify your ranking...",
  "response": "The most likely root cause is 'rate-service'
    (confidence 0.82). Its anomaly onset preceded 'compose-post'
    by ~1.2s and the lagged cross-correlation (lag=5) shows
    rate-service leading with kappa=0.71...",
  "tokens": {
    "prompt_tokens": 614,
    "completion_tokens": 291,
    "total_tokens": 905
  },
  "elapsed_seconds": 3.27
}
```

Listing 1: Sample entry from `llm_justification.jsonl` (causal_ranker node).

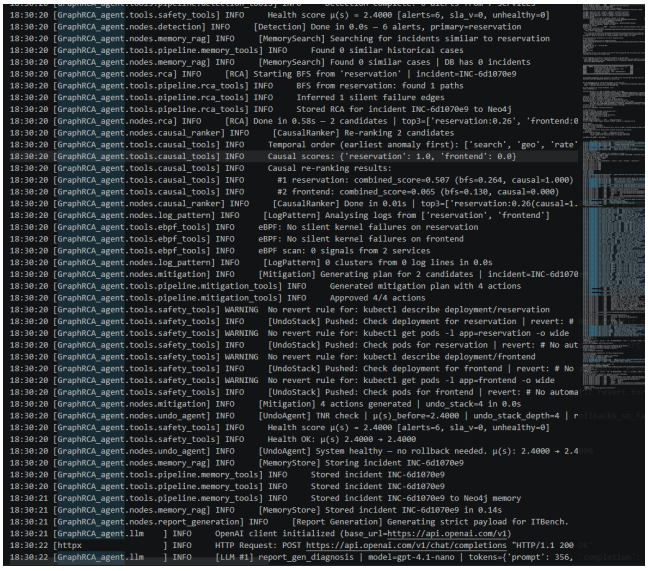

Figure 5: Branch `method-2`: GraphRCA LangGraph pipeline executing a detection task. The 11-node StateGraph progresses through `trace_ingest` → `graph_builder` → `detection`, with per-service EWMA z-scores and anomaly fused scores printed to stdout.

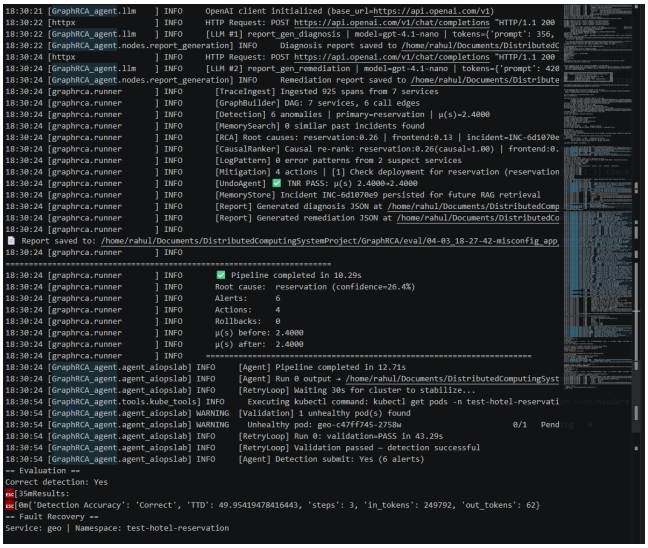

Figure 6: Branch `method-2`: Successful mitigation of `misconfig_app_hotel_res-mitigation-1`. TNR safety-check passes ($\mu(s)_{\text{post}} < \mu(s)_{\text{pre}}$ within 0.05 tolerance), `kubectl patch` executes, and the dual-oracle validation (alert clearance + throughput check) confirms resolution. No rollback triggered.

```python
def build_graph() -> StateGraph:
    """Construct and compile the GraphRCA StateGraph.

    Returns:
        Compiled LangGraph CompiledGraph ready for .invoke()
    """
    workflow = StateGraph(PipelineState)

    # ── Register nodes ──────────────────────────────────────────
    workflow.add_node("trace_ingest", trace_ingest_node)
    workflow.add_node("graph_builder", graph_builder_node)
    workflow.add_node("detection", detection_node)
    workflow.add_node("memory_search", memory_search_node)
    workflow.add_node("rca", rca_node)
    workflow.add_node("causal_ranker", causal_ranker_node)
    workflow.add_node("log_analysis", log_pattern_node)
    workflow.add_node("mitigation", mitigation_node)
    workflow.add_node("safety_check", undo_agent_node)
    workflow.add_node("memory_store", memory_store_node)
    workflow.add_node("report_generation", report_generation_node)

    # ── Linear pipeline edges ───────────────────────────────────
    workflow.set_entry_point("trace_ingest")
    workflow.add_edge("trace_ingest", "graph_builder")
    workflow.add_edge("graph_builder", "detection")
    workflow.add_edge("detection", "memory_search")
    workflow.add_edge("memory_search", "rca")
    workflow.add_edge("rca", "causal_ranker")
    workflow.add_edge("causal_ranker", "log_analysis")
    workflow.add_edge("log_analysis", "mitigation")
    workflow.add_edge("mitigation", "safety_check")

    # ── Conditional edge: Pillar 1 TNR rollback loop ────────────
    # If μ(s) regressed → loop back to rca (max 3 times, guarded in undo_agent_node)
    # Otherwise → persist incident and end
    workflow.add_conditional_edges(
        "safety_check",
        route_after_safety_check,
        {
            "rca": "rca",
            "memory_store": "memory_store",
        },
    )

    # ── Terminal edge ───────────────────────────────────────────
    workflow.add_edge("memory_store", "report_generation")
    workflow.add_edge("report_generation", END)
```

Figure 7: LangGraph StateGraph construction code for GraphRCA. The snippet shows the 11-node pipeline graph being assembled — nodes are added for each stage (`trace_ingest`, `graph_builder`, `detection`, `memory_search`, `rca`, `causal_ranker`, `log_analysis`, `mitigation`, `safety_check`, `memory_store`, `report_generation`) and edges are wired including the conditional TNR rollback arc from `safety_check` back to `rca` (up to 3 cycles).

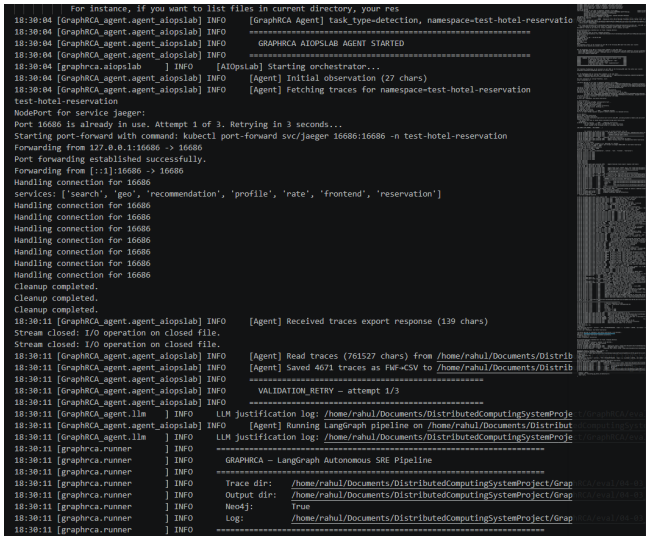

Figure 3: Branch `method-1`: Agent startup and AIOpsLab task registration. The Langgraph-based multi-agent conversation loop connects to the AIOpsLab orchestrator and begins issuing `kubectl` tool calls.

Table 5: Tasks registered in `eval/eval_tasks.yaml`.

Task ID	Type
`misconfig_app_hotel_res-detection-1`	Detection
`pod_failure_hotel_res-detection-1`	Detection
`scale_pod_zero_social_net-detection-1`	Detection
`assign_to_non_existent_node-detection-1`	Detection
`auth_miss_mongodb-detection-1`	Detection
`container_kill-detection`	Detection
`misconfig_app_hotel_res-localization-1`	Localization
`pod_failure_hotel_res-localization-1`	Localization
`scale_pod_zero_social_net-localization-1`	Localization
`assign_to_non_existent_node-localization-1`	Localization
`auth_miss_mongodb-localization-1`	Localization
`container_kill-localization`	Localization
`misconfig_app_hotel_res-analysis-1`	Analysis (RCA)
`pod_failure_hotel_res-analysis-1`	Analysis (RCA)
`scale_pod_zero_social_net-analysis-1`	Analysis (RCA)
`auth_miss_mongodb-analysis-1`	Analysis (RCA)
`misconfig_app_hotel_res-mitigation-1`	Mitigation
`pod_failure_hotel_res-mitigation-1`	Mitigation
`scale_pod_zero_social_net-mitigation-1`	Mitigation
`auth_miss_mongodb-mitigation-1`	Mitigation

Table 6: GraphRCA `kube_tools.py` safety filter.

Category	Rule	Example blocked
Destructive	Namespace deletion	`kubectl delete ns X`
	Stdin redirection	`kubectl apply -f -`
Interactive	`-it`/`-tty`	`kubectl exec -it pod`
	`edit` subcommand	`kubectl edit deploy/X`
Shell syntax	Pipe operator	`kubectl get pods | grep`
	Compound (&&, ‖, ;)	`... && echo done`
	Command substitution	`$(kubectl get pods)`

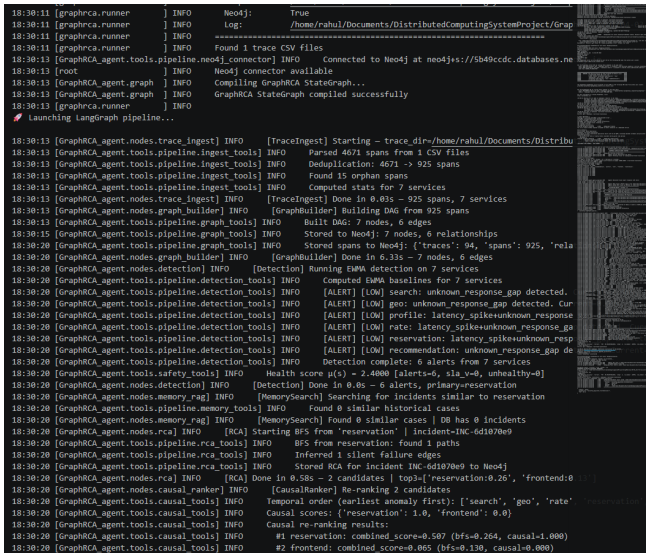

Figure 4: Branch `method-1`: Context saturation failure after >40 unbounded `kubectl` calls with no resolution — the motivating limitation that drove the GraphRCA redesign.

D System Architecture Diagram

E AIOpsLab Tasks Registered in GraphRCA

F Kubernetes Command Confinement Rules

The companion repository, where our previous method failed , the branch name in which the proposed method previously , is at: https://github.com/Rahuldrabit/Distributed-LLM-based-multi-Agent-for-autonomous-Site-Reliability-Engineering-and-Devops-task