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.

Auditing an LLM AIOps Agent Against Its Own Safety Mechanisms

Muhammad Saad Hussain

muhammadsaadhussain49@gmail.com

The University of Lahore

Muhammad Abubakar Sultan

The University of Lahore

Alishba Farrukh

The University of Lahore

Research Article

Keywords:

Posted Date: November 26th, 2026

DOI: https://doi.org/10.21203/rs.3.rs-11179117/v1

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

Additional Declarations: No competing interests reported.

Auditing an LLM AIOps Agent Against Its Own Safety Mechanisms

Muhammad Saad Hussain[1*], Muhammad Abubakar Sultan[1] and Alishba Farrukh[1]

[1*]Department of Computer Science, The University of Lahore, Sargodha, Pakistan.

*Corresponding author(s). E-mail(s): muhammadsaadhussain49@gmail.com;
Contributing authors: abubakrmuhammad259@gmail.com; alishba.farrukh@cs.uol.edu.pk;

Abstract

LLM agents for cloud incident management increasingly propose remedial actions and place them behind safety mechanisms, such as self-consistency voting, calibrated confidence thresholds and command allow-lists, that are validated against diagnostic performance. Whether they govern the actions themselves is rarely measured. We audit a deployed multi-signal LLM agent for Kubernetes against its own safety machinery, using 190 remediation plans from six injected fault classes, a pre-registered rubric, dry-run validation, execution against live faults and replay through a second model. Five-way voting changed no verdict relative to a single sample. A verdict-validated confidence gate admitted every plan in the scenario where 54 of 60 plans addressed the wrong service. The system's allow-list accepted 26.8% of its proposals. No state-changing command specified a namespace, so none executed as written; 96.3% did once it was supplied. Against live faults, seven of ten CPU-saturation proposals appeared to succeed, each only because it triggered a rolling update that destroyed the faulted pod. A second model was unanimous and confident in all 13 of its errors, and deleting the interpretive notes the pipeline adds to its context reversed 15 of its 16 annotated verdicts, with no change in 13 controls. The agent did decline to act on a self-healing fault in 29 of 32 cycles. Each mechanism behaves as specified; the specifications were never checked against the agent's output or its context. We recommend reporting the single-sample counterfactual for voting, recording how an execution outcome was reached, and testing context assertions by ablation.

Keywords: AIOps, LLM agents, Kubernetes remediation, safety mechanisms, self-consistency, evaluation methodology

1 Introduction

Large language model agents are increasingly proposed for cloud incident management, where they ingest observability data, diagnose faults and generate remediation actions. Much of the evaluation of such systems concerns diagnosis: detection accuracy, root-cause localisation, latency. Recent benchmarks have begun to evaluate remediation directly, executing an agent's actions against an injected incident and scoring whether it resolves [1, 2]. What remains largely unexamined is the machinery that deployed systems place between an agent's proposal and its execution.

That machinery matters because it carries the safety argument. Systems that generate `kubectl` commands, infrastructure changes or API calls typically place them behind sampling-based verdict stabilisation, calibrated confidence thresholds, and lists of permitted operations, and present

those mechanisms as the reason the system may be trusted to act. The mechanisms are usually validated against diagnostic performance. Whether a confidence score that predicts correct verdicts also predicts correct actions, or whether a permit list admits what the agent actually proposes, are separate empirical questions. Benchmark agents typically act directly, so these questions do not arise there; and an outcome score, when an agent fails, does not say whether it reasoned wrongly, emitted an unusable command, or had a sound action refused by its own guardrail.

We ask these questions of one deployed system. Using the published records of a multi-signal LLM AIOps agent evaluated on a live Kubernetes cluster [3], we measure what its safety mechanisms do with the actions it generates: the quality of those actions, whether its commands would execute, whether its self-consistency voting changes any decision, whether its confidence gate tracks action quality, what its own command allow-list does with its proposals, and, by executing a sample against live injected faults, whether the proposals resolve the fault at all.

The results point one way. Voting changed no verdict; the confidence gate admitted every proposal in the scenario where targeting was worst; the allow-list accepted about a quarter of the agent's proposals; no command executed as written; and the proposals that appeared to clear a live fault did so only by destroying the pod that contained it. A second model, given the same contexts, erred unanimously and confidently, and its verdicts turned on single interpretive sentences the pipeline had added. None of these mechanisms malfunctions. Each behaves as specified, but the specifications were written against an idealised account of what the agent would produce, and the agent's output was never measured against them. We argue this is a recurring design pattern rather than an implementation accident, and one that grows more consequential as such systems move toward autonomous execution.

The study also reports a positive result that conventional evaluation would not surface: on a fault Kubernetes repairs unaided, the agent proposed diagnostics rather than intervention in 29 of 32 cycles. Neither detection metrics nor remediation metrics that presume action is warranted can register that restraint.

Our contributions are:

1. **A pre-registered, rater-free rubric** for scoring LLM-generated Kubernetes remediation against injection-controlled ground truth, scoring target and action separately and penalising over-action on self-healing faults.
2. **An audit of a deployed agent's proposals against its own safety mechanisms and against live faults**, decomposing where its actions fail: in targeting, in executability, at its permit list, and in efficacy when executed.
3. **Evidence that self-consistency voting is inert for structured classification**, with a single-sample counterfactual that any system logging individual votes can compute, shown to hold in cycles whose context names no verdict and for a second model.
4. **Evidence that diagnosis-level calibration does not transfer to action quality**, with a demonstration that pooled analyses of this relationship across fault classes are confounded by scenario composition, and that voting and confidence certify stable errors rather than catching them.
5. **A replay of the system's own command allow-list** against its own proposals, separating rejections that reflect the guard working from those that reflect a permit list specified without reference to generated behaviour.
6. **Two methodological checks for evaluating remediation agents**: recording whether an injected fault's host was replaced, without which our execution results would have read as 70% success rather than none; and flagging context sentences that assert a conclusion, so that the pipeline's judgments are not credited to the model.
7. **A replay harness and a context ablation.** The stored contexts are replayed through the source system's unmodified agent code, with fidelity established by reproducing all 42 source verdicts sampled. Replayed through a second model and then stripped of the pipeline's interpretive notes, they show that removing those notes, and nothing else, reversed 15 of that model's 16 annotated verdicts.

The system under study is prior work by two of the present authors, and we state this plainly. The present paper evaluates properties the earlier study did not, using the same published dataset, and the two should be read as a pair. Section 3

describes the system only as far as needed to interpret the results, and Section 7 discusses the limits of studying a single framework.

Although the system operates on cloud-native infrastructure, the mechanisms we examine belong to a general class: runtime controls placed between a machine-generated proposal and its execution, meant to make that proposal safe for a human to approve. They appear wherever autonomy is introduced incrementally into operational systems, including mission-critical ones, and the assurance case for that autonomy typically rests on them. Our findings concern how such controls should be validated before they can carry that argument.

Section 2 reviews related work. Section 3 describes the system, Section 4 the methods, and Section 5 the results. Section 6 discusses them, Section 7 sets out limitations, and Section 8 concludes.

2 Related Work

2.1 LLM agents for cloud incident management

LLM agents are now applied across the incident lifecycle. STRATUS coordinates specialised agents for failure detection, diagnosis and mitigation in a multi-agent system for autonomous site reliability engineering [4]. RCAgent applies tool-augmented LLM agents to cloud root cause analysis [5], and TrioXpert combines modality-specific processing of metrics, logs and traces with LLM reasoning for incident management [6]. The source system of this study belongs to the same family: it fuses metrics, traces and logs into a structured context and asks an LLM agent for a verdict, a root cause and risk-tagged remediation [3].

Evaluation in this line of work has centred on diagnosis: detection accuracy, localisation accuracy and latency. These systems establish that LLM agents can diagnose faults in realistic cloud environments. Our concern is what happens downstream of the diagnosis.

2.2 Benchmarking remediation

Remediation has recently become a benchmark target in its own right, and the strongest such benchmark overlaps substantially with our setting.

MicroRemed introduces end-to-end microservice remediation as a task: given a diagnosis report, an LLM generates an executable Ansible playbook, which the benchmark runs and verifies by checking whether the injected failure was repaired [2]. It covers 421 fault-recovery pairs across seven failure types, including CPU and memory saturation, network delay and pod failure, on three microservice systems, one of which is Online Boutique, the application used in the system we study. Nine LLMs are evaluated under a one-shot baseline and a multi-agent framework with probing and reflection. The best configuration reaches 38.94% overall remediation accuracy, and none exceeds 50% even at the easiest difficulty level.

AIOpsLab provides a framework for evaluating AIOps agents against live Kubernetes environments through an Agent-Cloud Interface, with problems spanning detection, localisation, root cause analysis and mitigation, including deliberate no-fault controls [1]. Its reported mitigation results are the weakest of the four task types. ITBench evaluates agents on site reliability, compliance and financial operations scenarios; in its published version, agents resolve 11.4% of SRE scenarios [7].

MicroRemed's evaluation principle is deliberately execution-based: outcomes are judged by whether the system recovered, not by the form of the generated output. We regard that as the right standard for measuring remediation effectiveness and do not claim to improve on it. Our question is different.

An execution-based benchmark measures the end of a pipeline. It evaluates an agent built for the benchmark, acting directly on the environment, and reports whether the incident resolved. It does not, and does not attempt to, decompose a failure: whether the agent reasoned about the wrong service, emitted an unusable command, or had a correct action refused by a guardrail around it. The last case cannot arise in benchmark settings, because benchmark agents act directly. It arises in deployed systems, which interpose machinery between proposal and execution: confidence thresholds, permitted-action lists, sampling-based verdict stabilisation and operator confirmation.

That machinery carries the safety argument for such systems, and it is what we audit.

The two kinds of study also answer to different populations, and their numbers should not be compared. Our 42.1% fully-correct rate scores a single surfaced command against a fault-class rubric; MicroRemed's 38.94% executes a generated playbook and verifies recovery. Where remediation effectiveness is the question, execution-based evaluation is the appropriate instrument. Our own execution study (Section 5.6) follows that standard, and adds one requirement to it: recording whether the injected fault's host was replaced, without which a configuration change's side effect is scored as a fix.

MicroRemed's ablation is independently relevant. Removing its reflection mechanism costs 7.16% accuracy on average, while removing its probe agent costs only 1.57%, and in several configurations removing the probe agent improves accuracy, which the authors attribute to excessive probing adding noise. Mechanisms added to agent pipelines for reliability contribute unevenly, and their contribution can be measured.

2.3 Self-consistency and its validity conditions

Self-consistency replaces greedy decoding in chain-of-thought prompting with a sample-and-marginalise procedure: the model generates several reasoning paths and the most consistent final answer is selected [8]. Reported gains are substantial, for example +17.9% on GSM8K and +12.2% on AQuA, on free-form arithmetic, commonsense and symbolic reasoning tasks.

The method's stated rationale is specific. It rests on the intuition that a complex reasoning problem typically admits several distinct ways of reaching its single correct answer, and the authors observe that the more deliberate analysis a problem requires, the more diverse the reasoning paths that can recover the answer. The mechanism consumes that diversity: where samples agree, marginalising over them returns what the first sample gave. The original authors also report the method to be robust to sampling strategies and parameters. Our argument is therefore not that a temperature setting disables it, but that a task may not supply the diversity the method needs, however it is sampled.

Structured-output variants provide indirect support: recent work adds domain-specific validation and aggregates semantically equivalent structured outputs before voting, since exact-match voting over structured generations counts equivalent answers as distinct [9]. To our knowledge, no published AIOps agent reports what its verdicts would have been with a single sample. Section 5.3 computes exactly that.

2.4 Confidence calibration and execution gating

Calibration asks whether a model's stated confidence matches its empirical correctness. The most directly applicable technique is verbalised confidence elicitation, in which the model returns an answer together with a self-reported confidence [10], the mechanism the system studied here uses. Its dominant application is selective prediction: answer when confident, abstain otherwise, trading coverage against error on the answered subset [11]. In that frame the decision a confidence score governs is whether to emit an answer, and the quantity validated is whether the answer is right.

Recent work questions how tightly verbal confidence is coupled to what a model then does. Wang et al. find that frontier models give confidence that appears calibrated yet fail to translate it into cost-sensitive decisions, almost never abstaining even when the penalty structure favours it [12]. Kumaran reports that verbal confidence predicts a model's later commit-or-abstain decision substantially better than it predicts whether the answer is correct, across eight models that include `gpt-oss-120b`, the model of the system we study [13].

Both studies concern question answering, where the decision is whether to state an answer. Agent systems inherit the mechanism but change the decision. A confidence threshold in an operational agent gates whether a generated *action* is surfaced for execution. The score is still validated against verdict correctness, but what it releases is a command, a different object from the verdict it was validated on. Whether a score calibrated for the first purpose tells us anything about the second is a question neither the calibration literature nor the decision-coupling studies address,

because in their setting the answer and the output are the same thing. Section 5.4 addresses it in one deployed system.

2.5 Positioning

This is an evaluation paper, not a systems paper. We propose no new architecture and report no improvement. The contribution is a measurement instrument, its application to a deployed system's proposals and to their execution against live faults, and an argument that the class of mechanism it examines, runtime controls placed between an agent's proposal and its execution, is systematically under-evaluated.

The system under study is prior work by two of the present authors [3]. We evaluate properties the earlier study does not, on the same published dataset, and discuss the limits of a single-system result in Section 7.

3 System Under Study

This section describes only what is needed to interpret the results. Full architectural detail, the detection evaluation and the root-cause analysis are in the source study [3].

Pipeline. A monitoring loop runs every 30 seconds against a two-node Kind cluster (Kubernetes v1.32.2) hosting six services from the Google microservices-demo application. Each cycle queries Prometheus for per-service CPU and memory metrics, Jaeger for latency and error-span statistics, and Loki for log patterns. Modality-specific processing, namely Isolation Forest scoring for metrics, threshold analysis for traces and Drain3 template mining for logs, produces anomaly flags that are fused into a single structured context object, including a natural-language summary for the model.

Agent. When the context flags an anomaly, a Runtime Anomaly Agent is invoked. It sends the model a fixed system prompt, the fused summary, and a short history, and receives a verdict (`ANOMALY_CONFIRMED`, `FALSE_POSITIVE` or `NORMAL`), a confidence score, affected services, a natural-language root cause, and remediation steps each tagged LOW, MEDIUM or HIGH risk. The model is `openai/gpt-oss-120b`, served through a hosted API at temperature 0.1 with a 3,000-token output ceiling, under a mandatory JSON response schema.

The history holds the two most recent agent-invoking cycles, each as the cycle's summary and the resulting majority verdict. The pipeline runs as one process per experimental run, so history accumulates across cycles within a run and resets between runs.

Context annotations. The context builder does more than report measurements. In certain conditions it appends interpretive notes to the summary, and these matter for reading our results.

- When a service's CPU reaches 80% of its limit, a note states that this is resource saturation rather than idle behaviour.
- When a pod has recently restarted, a note says the reading may reflect start-up behaviour, and adds that a restart explanation is still a confirmed anomaly (`verdict=ANOMALY_CONFIRMED`), and that `FALSE_POSITIVE` should be used only on specific contrary evidence.
- When memory rises well above its baseline, a note describes a memory-pressure event and states that it is a confirmed anomaly (`verdict=ANOMALY_CONFIRMED`), not a false positive.
- When error logs reference a dependency, a note invites the model to consider whether that dependency, rather than the service reporting the error, is the true root cause.

The second and third kinds name a verdict label outright. Forty-nine of the 190 agent-invoking cycles carry such a sentence: all 29 in S3, 12 of 32 in S2, and 8 of 60 in S4. S1, S5 and S6 carry none. The fourth kind does not name a verdict but points at a root cause, and appears in S5. Each note is a separate summary line beginning `NOTE:`. We describe them because they bear on three of our analyses: the voting result (Section 5.3), which we test separately on uninstructed cycles; the S5 targeting result (Section 5.1), which the dependency note assists; and the ablation of Section 5.7, which removes them.

Three mechanisms are the subject of this paper.

Self-consistency voting. The agent queries the model five times per cycle and takes the majority

verdict and the mean confidence. Every individual vote is recorded, which makes the counterfactual analysis in Section 5.3 possible without re-execution.

Confidence gating. Proposals are surfaced for rapid operator confirmation only when the verdict is confirmed and mean confidence reaches $\theta = 0.85$. Medium- and high-risk steps escalate to a human regardless of confidence. Nothing executes without explicit operator confirmation; the source system has no autonomous execution path.

Command allow-list. A validation module applied after operator approval requires each command to match a permitted (verb, resource) pair, to reference one of six known services, and to contain no shell metacharacters. The permitted state-changing pairs are `rollout restart`, `scale deployment`, `patch configmap` and `rollout undo`, corresponding to the four remediation actions in the system's architecture diagram.

Dataset. Six scenarios, namely CPU stress, pod crash, memory pressure, network latency, cascading dependency failure and a fault-free control, each executed in five independent runs, for 330 monitoring cycles in total. Each cycle record contains the full fused context, the agent's complete output including every individual vote, the pipeline's final status, and the injected fault's target service and class. The agent was invoked in 190 of the 330 cycles, and every invocation produced at least one LOW-risk step. These 190 form the analysis set.

4 Method

We evaluate what the source system generates but does not measure: the quality of the proposed remediation, whether its commands would execute and whether they resolve the fault when executed, the marginal value of voting, the relation between confidence and the action it releases, and what the system's own allow-list does with the proposals. Except where stated, everything is computed from the published cycle records.

4.1 Scoring rubric

Remediation plans have no standard evaluation instrument. We define one, fix it in advance, and publish it, so that any assignment can be disputed and the metrics recomputed. It scores two independent dimensions.

Target correctness is binary: does the step act on the service responsible for the fault? It is judged against the `expected_root_cause_service` recorded at injection time, which requires no judgment because the fault was deliberately introduced.

Action appropriateness has three tiers. *Appropriate* (2): the action addresses the fault's mechanism. *Partial* (1): it relieves symptoms, incidentally clears the fault, or is reasonable triage that resolves nothing. *Inappropriate* (0): it does not address the fault or acts on an unrelated subsystem. A plan is **fully correct** only if the target is correct and the action Appropriate.

Tiers are assigned per fault class, because the same verb can address one mechanism and be blunt against another: a restart is Appropriate for memory pressure, where it discards accumulated heap state, and Partial for CPU saturation, where it prevents nothing. Table 1 gives the complete mapping.

Two scenarios deliberately reward inaction. In S2 Kubernetes replaces the crashed pod unaided, so diagnostics score Appropriate and state-changing actions Partial or Inappropriate; in S6 no fault exists. An agent that always proposes remediation scores poorly on both.

The rubric is mechanical and needs no human raters, so we report no inter-rater statistic and claim no independent validation; its authority rests on being fixed before scoring and published in full. Two tier definitions were tightened after an initial inspection of the data but before any metric was computed, and both changes are recorded in the rubric document.

4.2 Scope of scoring

We score the **first LOW-risk step** of each plan: the step the framework surfaces for rapid operator confirmation. Medium- and high-risk steps escalate to a human regardless of confidence, and the framework makes no autonomy claim about them. Reported figures therefore describe what the safety design exposes, not the whole plan.

Table 1 Action tier mapping by fault class. Tier 2 = Appropriate, 1 = Partial, 0 = Inappropriate. Tiers track whether an action addresses the fault's mechanism, not the verb itself.

Fault class	Tier 2	Tier 1	Tier 0
S1 CPU stress	vertical, horizontal scaling	restart, diagnostic	rollback, config patch
S2 Pod crash	diagnostic	restart	scaling, rollback, patch
S3 Memory pressure	restart, vertical scaling	horizontal, diagnostic	rollback, config patch
S4 Network latency	restart	diagnostic	scaling, rollback, patch
S5 Cascading fail.	restore correct target	inspect correct target	any action, wrong target
S6 Noisy baseline	no action, diagnostic	—	any state-changing action

4.3 Executability

Each state-changing command is submitted to `kubectl --dry-run=client`, which checks syntax, flags and resource resolution without applying anything. Read-only verbs and commands containing shell substitution are excluded. Where a verb rejects the dry-run flag, the command is downgraded to an existence check on its target rather than counted as a failure.

Each command is evaluated **twice**: exactly as emitted, and with the workload namespace supplied. The first measures what an operator pasting the command would experience; the second separates namespace resolution from every other failure mode. Commands are tokenised with shell quoting rules, since several pass JSON patches in single quotes; an earlier whitespace tokeniser rejected valid patches, and its correction is recorded in the repository. Validation used the Kind cluster on which the source experiments ran.

4.4 Voting analysis

Because all five votes are stored, the counterfactual needs no re-execution: we recompute the majority over the first vote alone ($K = 1$) and the first three ($K = 3$) and compare each with the deployed $K = 5$ verdict. This measures voting's effect on decisions, which drive the pipeline's branching, not on mean confidence, which necessarily differs at $K = 1$.

Verdict instructions. Because some contexts name a verdict label (Section 3), we split cycles by a mechanical rule fixed before counting: a context is *instructed* if any sentence in its summary contains `verdict=`, `ANOMALY_CONFIRMED` or `FALSE_POSITIVE`. The rule matches three distinct sentences across the dataset, listed in the repository, and every voting statistic is reported separately for instructed and uninstructed cycles.

4.5 Confidence analysis

We test whether confidence, validated in the source study against verdict correctness, also relates to the surfaced action, using target correctness and freedom from unresolved placeholders as outcomes. The placeholder check is a coarser proxy than the dry-run of Section 4.3, and the two should not be compared.

Analyses are stratified by scenario. Scenarios differ systematically in both confidence and action quality, so a pooled comparison of gate-passing against gate-failing plans can reproduce scenario composition rather than any property of confidence. We report pooled figures alongside stratified ones, because the gap is itself a methodological result. Correlations are reported only for S1 ($n = 58$) and S4 ($n = 60$).

4.6 Allow-list replay

We replay the framework's allow-list against the agent's own proposals, using a verbatim transcription of the deployed module, reproduced **as deployed rather than as intended**, including a singular/plural mismatch that rejects some read-only commands it was meant to permit. Correcting it would measure a gate that does not exist. Rejections are reported in separate classes: the guard working as designed (shell metacharacters), actions never enumerated, and defects in the allow-list itself.

4.7 Execution study

Dry-run establishes that a command would be accepted, not that it would help. We therefore executed a sample of proposals against live injected faults. The protocol was committed to the repository before any run, and every later change was recorded as a dated amendment stating what had and had not been observed when it was made.

Scope. S1 (CPU stress on `cartservice`) and S4 (network latency on `frontend`). S2 is excluded because Kubernetes self-heals, S5 because every first LOW-risk step is read-only, S6 because no fault exists. S3 was excluded mid-study, before any S3 outcome was observed: its memory signal reflects only 8–23% of the allocated amount, so no threshold reliably separates a present fault from an absent one.

Sample. For each scenario, the first ten distinct eligible commands in extraction order (excluding read-only verbs, shell substitutions and unresolved placeholders), plus three no-action controls in which the fault is injected and nothing applied. Commands are applied as emitted, with only the namespace supplied.

Procedure. Each run restores recorded baseline state, requires two consecutive healthy readings, injects the fault with the source study's method, confirms it is present, applies the proposal, and observes for five minutes at 30-second intervals. A fault counts as cleared after two consecutive readings below threshold. A run aborted at baseline or confirmation is attempted once more and otherwise logged as excluded.

Measurement. S1 uses the maximum CPU across all `cartservice` replicas, restricted to the application container: faulted at 40% of the CPU limit or above, cleared below 20%. The maximum ensures that idle replicas added by scaling cannot dilute a still-saturated pod. The source system's collector was not used, because it reads only the first series Prometheus returns and so, with several replicas, an arbitrary pod. S4 uses the median end-to-end latency of `frontend` traces over the preceding minute: faulted at 250 ms or above, cleared below 100 ms. The protocol originally specified p99 at 500 ms; the no-fault baseline, measured before any S4 outcome, showed p99 near 4 s (dominated by calls to services absent from this deployment) against a median of 22 ms, and the thresholds were set from that baseline alone.

Pod replacement. Both faults live inside the target pod, so any command that rewrites the pod template triggers a rolling update that removes the fault regardless of the command's content. We record the target pod's name before each proposal and whether it survives observation; clearance *without* replacement measures whether a proposal addressed the fault. This classification was fixed before any run.

Collateral effects are recorded when a non-target pod becomes not-Ready or restarts during observation. Readiness was sampled every 30 seconds, so this measure is a lower bound.

4.8 Model replay

Every cycle record stores the context object the agent received. To compare models on identical inputs, we replay those contexts through the source system's unmodified `analyze()` method, changing only the model name; the system prompt, five votes, retry logic, majority rule and history mechanism are those of the source experiments. Each run uses a fresh agent, so history accumulates within a run and resets between runs as in the original. Progress is checkpointed after every cycle, including the agent's exact history, so an interrupted replay resumes with history identical to an uninterrupted one.

Fidelity. We replayed the source model, `gpt-oss-120b`, on 42 cycles: run 1 of every scenario, plus the cycle in run 5 of S6 that holds the dataset's only `FALSE_POSITIVE`. The replay reproduced all 42 source verdicts, including that one, on which fidelity for the minority class rests.

Second model. We replayed `qwen3.8-27b` on the same 42 cycles. The scope, and a rule to report whatever portion completed, were fixed before the replay began. The model's free-tier limit on output tokens per minute is below the source system's 3,000-token ceiling, so each call is capped at 1,000 tokens. A response exceeding the cap fails to parse and is recorded as a schema failure, so truncation is counted rather than hidden.

Context ablation. Some contexts carry an interpretive NOTE line appended by the pipeline (Section 3). The second model's verdicts coincided with these lines, but each NOTE is triggered by the very readings it interprets, so the replay alone cannot separate the two. We therefore replayed every run containing a NOTE a second time, deleting every summary line that begins with `NOTE:` and changing nothing else: numeric fields, the anomaly JSON appended to the prompt, all other summary lines and all settings are those of the unablated replay. Cycles without a NOTE are replayed on identical text and serve as controls for run-to-run variation. The protocol, including the selection rule, outcomes and interpretation, was committed to the repository before any ablated

call, and states that it was motivated by the replay results.

4.9 Reproducibility

The rubric, analysis scripts, extracted commands, per-command validation records, execution-study protocol with amendments and raw observations, and replay outputs are available at https://github.com/saadhussain07/llm-aiops-safety-audit, archived as version 1.0.0 at https://doi.org/10.5281/zenodo.22974350. The source cycle data remains unmodified in the source study's archive (https://doi.org/10.5281/zenodo.22816805). The scoring, voting, confidence and allow-list analyses need only the cycle records; the executability and execution studies need a Kubernetes cluster; the replay needs access to the model API.

Use of AI tools. The two language models named above are objects of study. Separately, the authors used Claude (Anthropic) to help write and test the analysis and experiment scripts under their direction, to search for and summarise related literature, each summary checked against its source, and to help structure and draft the manuscript from the authors' results. The authors ran all code, verified every reported figure against the repository records, and take full responsibility for the content.

5 Results

Of the 330 monitoring cycles in the dataset, 190 invoked the agent and produced a verdict with at least one LOW-risk remediation step; these form the unit of analysis throughout. The remaining 140 raised no anomaly and never invoked the agent. We score the first LOW-risk step of each plan (Section 4.2).

5.1 Remediation quality

Pooled, 42.1% of plans are fully correct (correct target and Appropriate action), with a mean action tier of 1.11 out of 2 (Table 2). Three results stand out.

Restraint on self-healing faults. S2 (Pod Crash) scores highest, at 84.4% fully correct. Kubernetes replaces the crashed pod unaided, so the correct response is to verify recovery rather than act, and the rubric penalises over-action. The agent largely avoids it: 29 of 32 first steps are diagnostic (`describe`, `get`, `top`, `logs`).

Diagnosis without repair. S5 (Cascading Failure) scores 70% target accuracy but 0% fully correct. The agent identifies `redis-cart`, the deleted dependency, in seven of ten plans rather than the two services that surface the errors, helped by a context note that the error logs reference redis (Section 3). Yet every first LOW-risk step is `kubectl get pods -l app=redis-cart`: the agent localises the failure and then proposes to look at it, which the rubric scores Partial.

Remediation aimed at the loudest signal. S4 (Network Latency) scores 10% target accuracy. The fault is a 500 ms delay on `frontend`; 54 of 60 first steps propose capacity actions against `cartservice`, such as `kubectl scale --replicas` or `kubectl set resources --limits=cpu`. The agent's narratives attribute the incident to CPU saturation in `cartservice`, the downstream pressure through which the source study's ablation showed S4 is partly detected, and the remediation follows that diagnosis consistently.

Across all scenarios, 40.0% of scored steps are read-only diagnostics.

5.2 Executability

Of the 190 scored steps, 77 are read-only and 31 cannot be tested by dry-run (shell substitution, `edit`, `exec`), leaving 82 state-changing commands, each evaluated as emitted and with the workload namespace supplied.

None of the 82 commands executes as written; with the namespace supplied, 79 (96.3%) do. Of the as-written failures, 78 are `RESOURCE_MISSING`: the command is well formed and names a real deployment, but resolves against the wrong namespace. The three that fail even with the namespace are two unresolved placeholders and one `apply` of a manifest that does not exist. No state-changing proposal specifies a namespace, so each resolves against whatever the operator's kubectl context happens to be:

```
$ kubectl scale deployment/cartservice \
    --replicas=3 --dry-run=client
error: no objects passed to scale
deployments.apps "cartservice" not found

$ kubectl scale deployment/cartservice \
```

Table 2 Remediation quality of the first LOW-risk step. *Fully correct* requires correct target and an Appropriate action. S6 has no injected target. S5 ($n = 10$) and S6 ($n = 1$) support no rate claim.

Scenario	n	Target	Approp.	Mean tier	Fully correct	Placeholder
S1 CPU stress	58	100.0%	72.4%	1.62	72.4%	22.4%
S2 Pod crash	32	93.8%	90.6%	1.81	84.4%	93.8%
S3 Memory pressure	29	100.0%	31.0%	1.31	31.0%	34.5%
S4 Network latency	60	10.0%	5.0%	0.20	1.7%	11.7%
S5 Cascading failure	10	70.0%	0.0%	0.70	0.0%	60.0%
S6 Noisy baseline	1	n/a	100.0%	2.00	100.0%	0.0%
Pooled	**190**	**68.8%**	**44.2%**	**1.11**	**42.1%**	**34.7%**

Table 3 Dry-run executability of state-changing commands ($n = 82$ of 190; read-only and untestable commands excluded), as emitted and with the workload namespace supplied. S2 contributes one command; no rate is reported for it.

Scenario	n	As written	Namespace supplied
S1 CPU stress	28	0.0%	96.4%
S2 Pod crash	1	0.0%	—
S3 Memory pressure	9	0.0%	100.0%
S4 Network latency	44	0.0%	95.5%
Pooled	**82**	**0.0%**	**96.3%**

```
    --replicas=3 -n microservices-demo \
    --dry-run=client
deployment.apps/cartservice scaled (dry run)
```

The claim is narrow: the commands are not malformed, and an operator whose context already pointed at the workload namespace would find most of them work. Correctness is delegated to ambient state the agent neither observes nor mentions.

5.3 Self-consistency voting

Because every vote is stored, we recompute the majority over the first vote alone (K = 1) and the first three (K = 3) and compare each with the deployed K = 5 verdict.

Neither K = 1 nor K = 3 differs from K = 5 on any of the 190 cycles, in any scenario. In 173 cycles (91.1%) all five votes agree; of the other 17, eleven are 4/4 (one vote failed at the API), five split 4/5 and one 3/5, and in every split the first vote matches the majority. Voting multiplies model calls by five and changed no decision.

Verdict instructions do not explain this. Some contexts name a verdict label; in S3, every agent-invoking context states *"This is a confirmed anomaly (verdict=ANOMALY_CONFIRMED), not a false positive"* (Section 3). One might suspect the votes agree because the prompt dictates the answer, so we split cycles by a mechanical rule (Section 4.4).

Forty-nine cycles (26%) are instructed, concentrated in S3 (all 29), S2 (12 of 32) and S4 (8 of 60). In the 141 uninstructed cycles, the first vote matched the majority in every case. Instructions do raise agreement: all six cycles in which votes genuinely split are uninstructed. But the K = 1 result holds without them. This is what self-consistency's own rationale predicts for a task that returns one of three verdicts under a fixed schema, leaving little room for the alternative reasoning paths the method aggregates over [8].

5.4 Confidence and action quality

The source system validates confidence against verdict correctness (point-biserial $r = 0.47$, $p < 0.0001$, $n = 190$) and surfaces proposals for rapid confirmation at $\theta = 0.85$. We ask whether that validation extends to the surfaced action.

A pooled analysis is confounded, and the confound is instructive. Pooled, gate-passing plans appear *less* likely to target the correct service than gate-failing ones (61.5% versus 95.1%). Stratified by scenario the effect vanishes (+0 pp in S1, −2 pp in S2, +0 pp in S3, −8 pp in S5; S4 cannot be tested because every plan passes). The pooled difference arises only because S4 contributes 60 high-confidence, poorly targeted plans to the passing group. A weaker association survives for placeholder-free commands in S1 (+18 pp, $r = +0.41$) and S3 (+17 pp), absent in S2 and reversed in S5; we draw no conclusion from it.

Table 4 Self-consistency voting, split by whether the context names a verdict label. *Unanimous*: all usable votes agree. K=1 *differs*: the first vote alone would have given a different verdict from the five-vote majority. K=3 also differs on no cycle.

Scenario	Context	n	Unanimous	K=1 differs
S1 CPU stress	uninstructed	58	56 (97%)	0
S2 Pod crash	instructed	12	12 (100%)	0
	uninstructed	20	19 (95%)	0
S3 Memory pressure	instructed	29	29 (100%)	0
S4 Network latency	instructed	8	8 (100%)	0
	uninstructed	52	52 (100%)	0
S5 Cascading failure	uninstructed	10	8 (80%)	0
S6 Noisy baseline	uninstructed	1	0 (0%)	0
Pooled	**instructed**	**49**	**49 (100%)**	**0**
	uninstructed	**141**	**135 (96%)**	**0**

Table 5 Scenario composition. S4 has the highest mean confidence, clears the $\theta = 0.85$ gate on every plan, and has the worst targeting. S2 inverts all three. *Placeholder-free*: the command contains no unresolved placeholder; this is a coarser measure than the dry-run validation of Table 3.

Scenario	n	Gate pass	Placeholder-free	Correct target	Mean conf.
S1 CPU stress	58	71%	78%	100%	0.869
S2 Pod crash	32	44%	6%	94%	0.838
S3 Memory pressure	29	93%	66%	100%	0.889
S4 Network latency	**60**	**100%**	**88%**	**10%**	**0.919**
S5 Cascading failure	10	60%	40%	70%	0.861

The substantive result is the scenario composition (Table 5). S4 has the highest mean confidence (0.919), clears the gate on every plan, and has the worst targeting (10%). S2 is its mirror: the lowest confidence (0.838), the lowest gate-pass rate (44%), and 94% targeting. The agent is most confident where it is most wrong about what to act on.

5.5 The framework's own safety allow-list

The framework's second gate, applied after operator approval, permits four state-changing (verb, resource) pairs, `rollout restart`, `scale deployment`, `patch configmap` and `rollout undo`, matching the four actions in the architecture diagram, plus six diagnostic pairs, and rejects shell metacharacters.

26.8% of proposals are accepted, with per-scenario rates between 20% and 33%: a consistent mismatch, not one fault class behaving unusually. The rejections fall into classes that must not be conflated.

The guard working as intended (61). These contain shell metacharacters, mostly command substitutions such as `--replicas=$(kubectl get deployment ...)`. Blocking them is correct.

Actions never enumerated (58). `kubectl set resources` accounts for 34, with `logs deployment` (9), `patch deployment` (8), `edit deployment` (4), `apply` (2) and `autoscale deployment` (1). These are ordinary remediation actions; `set resources` is the most common state-changing command the agent produces, and it is not on the list.

A defect in the gate (10). The allow-list permits `("get", "pod")` and `("top", "pod")`; the agent writes `get pods` and `top pods`, and these read-only commands are rejected by a singular/plural mismatch.

The remaining 10 are unparseable or not kubectl commands. Setting aside the 61 the guard correctly blocks, the gate rejects 78 of the remaining 129 (60%). The permit list was enumerated from the architecture diagram rather than from measured agent behaviour, and the two were never reconciled.

5.6 Do the proposals resolve the fault?

We executed a sample of proposals against live injected faults (Section 4.7). Both faults live inside the target pod, so any command that rewrites the pod template (`set resources`, template `patch`, `rollout restart`) destroys the pod and removes

Table 6 The framework's own command allow-list replayed against the agent's proposals: acceptance by scenario, and rejection classes as a share of all 190 proposals. The classes are reported separately because only the second reflects a mismatch between specification and generated behaviour.

Scenario	n	Accepted
S1 CPU stress	58	13 (22.4%)
S2 Pod crash	32	7 (21.9%)
S3 Memory pressure	29	9 (31.0%)
S4 Network latency	60	20 (33.3%)
S5 Cascading failure	10	2 (20.0%)
S6 Noisy baseline	1	0 (0.0%)
Pooled	**190**	**51 (26.8%)**
Outcome of all 190 proposals		
Accepted	51	26.8%
Rejected: shell metacharacters (guard as designed)	61	32.1%
Rejected: action never enumerated	58	30.5%
Rejected: defect in allow-list (singular/plural)	10	5.3%
Rejected: unparseable or not `kubectl`	10	5.3%

the fault regardless of its content. Clearance *without* replacement of the target pod is therefore the measure of whether a proposal addressed the fault.

No proposal in either scenario resolved a fault by addressing it.

S1. Seven of ten proposals cleared the fault; all seven replaced the target pod. The three that left it running, two `scale` and one `autoscale`, did not clear it, matching the three no-action controls. Clearance times cluster tightly within each session (121.9–122.4 s for three consecutive runs; 60.7–60.9 s for three after a cluster restart), the signature of a fixed rolling-update sequence rather than of a remedy taking effect. The agent's narratives attribute the saturation to a workload spike and prescribe for load; against the injected runaway process, neither more replicas nor a higher limit reduces CPU.

S4. No proposal cleared the fault, because all ten act on `cartservice` while the delay sits on `frontend`; this includes the one restart proposal. S4 confirms the targeting result of Section 5.1 at the level of execution.

Rubric and execution measure different things. S1 scored 72.4% fully correct yet resolved nothing: the rubric credits targeting and action type, neither of which entails efficacy. A naive count of whether the fault went away would report 70% success for S1; recording whether the faulted pod survived reduces it to zero.

5.7 A second model, and what its context decides

We replayed the stored contexts of 42 cycles through the unmodified agent with `qwen3.8-27b` (Section 4.8). All 42 completed; two S4 cycles lost votes to connection errors and were decided on two and three usable votes, and one vote in the replay failed the schema.

The second model agrees with the source on 29 of 42 verdicts. All 13 disagreements run one way: the source confirmed an anomaly and the second model called it a false positive. The faults were injected in every one of these cycles, so all 13 are errors. Each was unanimous, and each carried a confidence of 0.85 or above. The model's reasoning is coherent: with `cartservice` at 37–63% of its CPU limit it cites the remaining headroom, the absence of trace or log anomalies, and a near-idle baseline behind ratios of 94 to 161 times normal. It does so beneath a context line, present in every cycle, instructing that "a small absolute number can still represent full resource saturation".

Voting behaves as it did for the source model. Every cycle is unanimous among its usable votes, so the first vote alone matches the majority in all 42. In S4 the second model targets `currencyservice` in 10 of 12 cycles, where the source targeted `cartservice`; the fault is on `frontend`. The two models are wrong about the same incident in different ways, each with high confidence. In S5 both localise `redis-cart`, and both surface `kubectl get pods` against it as the first step.

The 13 errors are exactly the cycles whose context carries no NOTE. In S1 the model rejects cycles 1–8 and confirms cycles 9–12, the four in which CPU crossed 80% of the limit and the pipeline appended "this is resource saturation". In S2 it confirms cycles 1–4, whose restart

Table 7 Execution study. Proposals were applied with the namespace supplied to live injected faults; controls received no action. A command that rewrites the pod template replaces the pod and removes an in-pod fault by construction, so *cleared without replacement* is the measure of whether a proposal addressed the fault. Four runs were excluded before observation (Section 4.7).

Scenario	Type	n	Cleared	Pod replaced	Cleared w/o replacement
S1 CPU stress	control	3	0	0	0
	proposal	10	7	7	**0**
S4 Network latency	control	3	0	0	0
	proposal	10	0	0	**0**

Table 8 Second model (`qwen3.8-27b`) replayed on the stored contexts of run 1 of every scenario and the source study's only FALSE_POSITIVE cycle (S6, run 5). *Agrees*: same verdict as the source model. *NOTE*: cycles whose context carries an interpretive NOTE line. *Reversed*: NOTE cycles whose majority verdict changed when only the NOTE lines were removed. *Controls changed*: cycles without a NOTE whose verdict changed on the identical replay.

Scenario	n	Agrees	NOTE	Reversed	Controls changed
S1 CPU stress	12	4	4	4	0 of 8
S2 Pod crash	9	4	4	4	0 of 5
S3 Memory pressure	6	6	6	6	—
S4 Network latency	12	12	0	—	—
S5 Cascading failure	2	2	2	1	—
S6 Noisy baseline	1	1	0	—	—
Total	**42**	**29**	**16**	**15**	**0 of 13**

note names the verdict, and rejects cycles 5–9, which have none. Because each note is triggered by the reading it interprets, this coincidence could reflect the readings alone.

Removing the NOTE lines reversed 15 of 16 verdicts; none of 13 controls changed. Every reversal ran from confirmed anomaly to false positive. In S1 the model rejected cycles at 80.6%, 90.6%, 96.1% and 101.6% of the CPU limit once the note was gone, in the last case reasoning that "the absolute CPU usage is extremely low (0.2032 cores)". In S2 it still reported from the appended JSON that the pod was 37 seconds old and had restarted, and called the reading a start-up artefact: the fact survived the ablation, and what the note had supplied was the rule that a restart counts as an anomaly. All six S3 verdicts reversed. In S5 the cycle in which two services logged errors naming `redis-cart:6379` held its verdict and its target; the cycle with a single service reversed. The only split vote in the ablation, 4/5 in S1, again had the first vote on the majority side.

The S1 note names no verdict label, so our mechanical rule of Section 4.4 counts those cycles as uninstructed. For this model it was as decisive as the notes that do.

6 Discussion

6.1 Mechanisms specified, not measured

The system studied here layers three mechanisms over its telemetry pipeline, each meant to make an LLM agent safe to act on: voting to stabilise the verdict, a confidence threshold to decide which proposals reach the operator, and a command allow-list to constrain what may execute after approval. Each is a reasonable design choice with precedent in the literature. Measured against the behaviour it governs, each fails to deliver what its presence implies.

None of these is a coding error in the ordinary sense. Each mechanism does what it was specified to do. The specifications were written against an idealised picture of the agent's behaviour: the four actions in the design diagram, the reasoning benchmarks on which self-consistency helped, the calibration study showing that confidence tracks correctness. The agent is an open-ended generator whose actual output is an empirical question. Where the two are never reconciled, the mechanism becomes decorative: present, documented, and not doing the work its presence implies.

6.2 Confidence gates the wrong object

The source system's calibration is sound: confidence predicts verdict correctness at r = 0.47 over the same 190 cycles. Yet S4 records the highest mean confidence of any scenario, clears the gate on every plan, and targets the correct service in 10% of them. There is no contradiction. In S4 the verdict is correct, and the confidence reflects strong evidence for it. What the score does not say is whether the action attached to the verdict addresses the cause, and the gate, reading only the score, admits the action on the strength of the diagnosis.

This is a transfer failure, not a calibration failure. Validating a confidence signal against verdict correctness licenses gating verdicts, not actions. Recent work finds verbal confidence weakly coupled to what a model then does [12, 13]; our case is the step beyond that. Here the confidence is warranted, the gate fires correctly on a correct verdict, and the action it releases is still aimed at the wrong service.

6.3 Consistency is not correctness

Voting and confidence share a limitation. Both measure agreement: voting between samples, confidence of the model with itself. Both can therefore catch error that varies between samples, and neither can catch error that is stable. Of S4's 60 cycles, 52 carry no verdict instruction, and all 52 are unanimous. In the scenario with the highest mean confidence in the dataset, independent samples agree cycle after cycle on remediation aimed at a service that is not at fault. Voting passes this and so does the gate; they certify the mistake rather than catch it.

Self-consistency is a remedy for variance: it marginalises over diverse reasoning paths [8]. The dominant failure in this dataset is not variance but bias, a context that foregrounds the wrong signal and is read the same way every time. Against that failure, more samples add cost and no protection.

The second model shows the same limitation without the cover of a correct verdict. Its 13 errors were each unanimous and each carried confidence at or above the threshold. On identical inputs the two models returned opposite verdicts on those cycles, each unanimously. Each model agrees with itself; what matters is agreement with the fault, and neither mechanism measures that.

6.4 The context decides more than the model

Three results that look like model failures trace back to the context object the pipeline builds.

What it foregrounds. In S4 the agent attributes the incident to CPU saturation in `cartservice` and acts on it consistently. The CPU pressure is the downstream consequence of the delay injected on `frontend`, and the fused context presents it as the most salient anomaly. The agent reasons coherently from salience to cause. In S1 it identifies CPU saturation correctly, assumes a load-driven cause, and prescribes for load. A context that reports which services are anomalous, without representing which anomaly precedes which or what is consuming a resource, invites an agent to act on the loudest reading and the most common explanation. The second model, reading the same S4 contexts, blamed `currencyservice` instead: two models, two wrong services, one context that never says which anomaly came first. Where the context does supply causal structure, as the redis note does in S5, both models localise the failed dependency.

What it omits. No state-changing proposal specifies a namespace, because the context never names one. That agents emit unusable commands is documented [1]; what this result adds is a cheaply fixed cause. Every one of these commands reads correctly, 82 of 82 fail, and 79 succeed once one field is supplied. The gap was invisible to inspection and obvious on execution.

What it asserts. In some conditions the context tells the model what to conclude, for instance that a memory-pressure event "is a confirmed anomaly (verdict=ANOMALY_CONFIRMED)". When the prompt states the verdict, a correct verdict measures instruction-following, not judgment. For the second model the ablation makes this literal: deleting the NOTE lines reversed 15 of its 16 verdicts, and none of 13 controls. Naming a label was not necessary. The S1 note asserts only that the reading "is resource saturation", and it was as decisive as the notes that name the verdict. With the note removed the facts stayed in the prompt, a CPU above its limit or a pod 37 seconds old, and the model reported them; what the note

had supplied was a sentence telling it how to read them. A rule that searches for verdict labels, like ours in Section 4.4, therefore finds only the most explicit assertions. We did not ablate the source model, which confirmed S1 without any note, so the degree of dependence is model-specific; but the source study's S3 detection results should be read with the annotation in view.

The annotations were not written arbitrarily. The source code records that the restart note was added after the source model mis-explained a restart. A context tuned to correct one model's errors became, for another, the determinant of its verdict.

An LLM agent's apparent competence is therefore partly a property of the pipeline around it. Evaluations should state what the context asserts and omits, not only what the model outputs, and report results separately where the context names the answer. A mechanical rule such as ours gives a lower bound. The direct test is the ablation of Section 5.7: delete the assertions and replay, which costs little wherever contexts are logged.

6.5 Surface success and genuine repair

Counting whether the injected fault went away, seven of ten S1 proposals succeed. Recording whether the faulted pod survived, none do. A benchmark that scored outcomes alone would have reported the agent as largely effective at resolving CPU saturation, when it resolved nothing by addressing it.

The confound is general. Fault injection commonly places the fault inside a container, and any remediation that recreates the container removes it, whatever the remediation says. Outcome-based evaluation is the right standard for remediation [2], but it should record how an outcome was reached: at minimum, whether the injected fault's host was replaced. The check is cheap, and without it an evaluation can score the side effect of a configuration change as a fix.

6.6 Restraint

Not all results are negative. S2 injects a pod crash that Kubernetes repairs unaided, and the agent largely responds as it should: 29 of 32 first steps are diagnostic, the highest fully-correct rate in the set.

No-fault controls, as in AIOpsLab [1] and the source study, test whether an agent declines to raise an incident when nothing is wrong. S2 tests something they cannot: an anomaly is genuinely present, the agent is right to confirm it, and the question is whether it then recognises that the platform has already resolved it. Detection metrics score a restart and a diagnostic identically after a correct confirmation. We suggest that where remediation is scored, self-healing faults be included and scored so that unnecessary action is penalised.

6.7 Implications for safety-gated automation

The results separate two kinds of validation that practice tends to conflate. *Decision-level* validation asks whether the agent's judgment is correct; the source study addresses it, with the caveat of Section 6.4 that the context sometimes supplies the verdict. *Action-level* validation asks whether what the agent proposes is correct, executable, permitted and effective. In this dataset the two are close to independent: the diagnoses are largely right, while the actions miss the faulted service in the hardest scenario, depend on context they do not specify, are refused by the system's own permit list in three of five cases not blocked for shell hazards, and, when executed, resolve nothing by addressing the fault.

For advisory systems, decision-level validation may suffice, because a human reads the proposal and supplies the missing judgment. For systems moving toward autonomous execution, it does not: the operator's confirmation is often the only thing between a well-calibrated diagnosis and a badly targeted action, and it is the step autonomy removes.

The same holds wherever autonomy is introduced incrementally into an operational system and the assurance case rests on runtime controls between a machine-generated proposal and its execution. Such a case should answer four questions, each only empirically: whether the confidence signal gating action has been validated against action quality; whether the permitted actions have been checked against what the generator actually produces; whether proposals have been executed in the target environment rather than reviewed for plausibility; and whether the

context supplied to the model asserts the conclusions its outputs are then credited with reaching.

7 Limitations

Single system. Every result is measured on one framework, one prompt design and one two-node Kind cluster running six services, and all but Section 5.7 on one model. The structural claim of Section 6.1, that safety mechanisms specified from architecture diverge from measured generator behaviour, is an interpretation of a single case. We expect it to generalise because the design pattern is common, but we have not shown that; the natural test is the same analysis applied to an agent built by another group.

Sample size. The 190 scored plans suffice for the pooled figures and for S1 (n = 58), S2 (32), S3 (29) and S4 (60), but not for S5 (10) or S6 (1), where we avoid rates. The execution study comprises 26 runs; its results are reported as counts, and no rate from it carries a meaningful interval.

Surfaced step only. Scoring covers the first LOW-risk step (Section 4.2). Excellent later steps earn nothing, and a badly wrong higher-risk step never counts against the fully-correct rate.

Scope of the execution study. A cleared fault means the injected condition stopped being measurable, not that the service was restored to a correct configuration. The pod-replacement confound is a property of the injection method: faults arising from traffic, configuration or infrastructure would survive a pod replacement. Our conclusions concern these injected faults, not all faults of these classes. Three of 260 readings were empty owing to transient loss of monitoring connectivity; an empty reading resets the clearance counter, so it can conceal a clearance but never produce one.

Mid-study amendments. Two of the four protocol amendments were made during the study. S3 was excluded after its first run failed fault confirmation, because the protocol's thresholds had assumed full reflection of the injected memory where the source study records 8–23%. The S4 metric was changed from p99 to median latency after the no-fault baseline showed a p99 near 4 s. In both cases no outcome for that scenario had been observed, and each amendment records what had and had not been seen when it was made.

Context annotations. In 49 of the 190 cycles the context names a verdict label (Section 3). Voting is analysed separately for instructed and uninstructed cycles, and the central result holds on the latter; the other analyses are not split this way. The S3 verdicts, which the context supplied in every cycle, and the S5 targeting, which a dependency note assisted, reflect the pipeline as well as the model; Section 5.7 shows how far this goes for a second model.

Rubric tiers are judgment calls. Practitioners may disagree with specific tier assignments. The most contestable, crediting a restart under network latency, holds only because the injected delay lives in the pod's own network namespace, the property that also drives the pod-replacement confound. We publish the complete mapping so that any disagreement can be localised to a named tier and the metrics recomputed. The rubric was reviewed before use by a practising DevOps engineer who is related to the first author and took no part in the system's design; we report this as a check on wording, not as independent validation.

Defects in our own measurement. Two defects were found and corrected before the results reported here. The executability script's whitespace tokeniser invalidated quoted JSON patches, understating namespace-supplied executability as 87.8% rather than 96.3%. The source system's metric collector reads an arbitrary pod when several replicas exist; it was not used in the execution study, and it did not affect the source study, in which every service ran a single replica.

Shared dataset. These analyses use the cycle records published with the source study. The questions differ, since that work evaluates detection and root-cause attribution and this one the actions generated downstream of them, but the observations are the same, and the two papers should be read as a pair.

Model replay and ablation. The second model was replayed on 42 cycles, one run per scenario, and the ablation on the 29 of those that belong to runs containing a NOTE, each once. The counts are clear-cut, 15 of 16 against 0 of 13, but they come from one model and one replicate, and we did not ablate the source model. Fidelity on the minority verdict rests on a single cycle. A hosted model may change over time; the replays measure each model as served at replay time. The second model's output was capped at 1,000 tokens;

one vote in its replay failed the schema. The ablation was designed after the replay had shown the coincidence it tests, and the protocol says so.

8 Conclusion

LLM agents for cloud incident management are increasingly designed to propose or take remedial action, and the mechanisms that make this acceptable (sampling-based reliability, calibrated confidence and constrained execution) are usually validated against detection and diagnosis. We measured them against the actions they govern, using 190 remediation plans from a multi-signal LLM AIOps agent across six injected fault classes.

Self-consistency voting at K = 5 changed no verdict relative to a single sample, including in the 141 cycles whose context named no verdict. A confidence gate validated against verdict correctness admitted every proposal in the scenario with the worst targeting, where 54 of 60 plans addressed the wrong service. An allow-list specified from the architecture accepted 26.8% of the agent's proposals and refused its most common state-changing action. No state-changing command specified a namespace, so none executed as written, while 96.3% executed once it was supplied. Executed against live faults, seven of ten CPU-saturation proposals appeared to succeed, each only by triggering a rolling update that destroyed the pod containing the fault. Replayed on the same contexts, a second model was unanimous and confident in each of its 13 errors, and deleting single interpretive sentences from its context reversed 15 of its 16 annotated verdicts while leaving all 13 unannotated controls unchanged.

None of these is a malfunction: each mechanism does what it was specified to do, against an idealised account of the agent's behaviour that was never measured. The same agent also showed a capability conventional evaluation misses: on a fault Kubernetes repairs unaided, it proposed diagnostics rather than intervention in 29 of 32 cycles.

We make four recommendations. Systems that report self-consistency as a reliability mechanism should also report the K = 1 counterfactual, which is trivial wherever samples are logged. Evaluations of action-generating agents should execute proposals rather than inspect them: here the gap between a command that reads correctly and one that runs was a single absent field. Outcome-based evaluation should record how an outcome was reached, at minimum whether the injected fault's host was replaced. And evaluations should state what the context asserts, and test it by deleting the assertions and replaying, so that the pipeline's conclusions are not credited to the model.

Acknowledgements. We thank Muhammad Azhar Javaid (Punjab Information Technology Board, Lahore) for reviewing the remediation rubric before it was used.

Declarations

Funding: This research received no external funding.

Competing interests: The authors declare no competing interests.

Ethics approval: Not applicable. This study did not involve human subjects, animal subjects, or personally identifiable data.

Relationship to prior work: The system evaluated in this paper was developed by two of the authors (M.S.H. and A.F.) and is described in [3], which is under review at another journal. The present paper addresses a distinct research question using the same published dataset.

Data and code availability: The analysis code, scoring rubric, execution-study protocol and amendments, raw observations and replay outputs are available at https://github.com/saadhussain07/llm-aiops-safety-audit, archived as version 1.0.0 (DOI: 10.5281/zenodo.22974350). The source cycle records are published with the source study at https://github.com/saadhussain07/fgcs-aiops-llm-agent (DOI: 10.5281/zenodo.22816805) and are used here unmodified.

Author contributions (CRediT): **Muhammad Saad Hussain:** Conceptualization, Methodology, Software, Validation, Formal analysis, Investigation, Data curation, Writing – original draft, Writing – review & editing, Visualization. **Muhammad Abubakar Sultan:** Formal analysis, Writing – review & editing. **Alishba Farrukh:** Supervision, Writing – review & editing.

Use of generative AI: Described in the Methods (Section 4.9).

References

[1] Chen, Y., Shetty, M., Somashekar, G., Ma, M., Simmhan, Y., Mace, J., Bansal, C., Wang, R., Rajmohan, S.: AIOpsLab: A holistic framework to evaluate AI agents for enabling autonomous clouds. In: Proceedings of Machine Learning and Systems (MLSys), vol. 7 (2025)

[2] Zhang, L., Zhai, Y., Jia, T., Duan, C., He, M., Pan, L., Liu, Z., Ding, B., Li, Y.: MicroRemed: Benchmarking LLMs in Microservices Remediation. arXiv:2511.01166 (2025)

[3] Hussain, M.S., Farrukh, A.: A telemetry-driven LLM-agent framework for fault detection and advisory remediation in cloud-native Kubernetes systems. Future Generation Computer Systems (2026). Under review, manuscript FGCS-D-26-05108

[4] Chen, Y., Pan, J., Clark, J., Su, Y., Zheutlin, N., Bhavya, Arora, R., Deng, Y., Jha, S., Xu, T.: STRATUS: A multi-agent system for autonomous reliability engineering of modern clouds. In: Advances in Neural Information Processing Systems (NeurIPS) (2025)

[5] Wang, Z., *et al.*: RCAgent: Cloud root cause analysis by autonomous agents with tool-augmented large language models. In: Proceedings of the ACM International Conference on Information and Knowledge Management (CIKM), pp. 4966–4974 (2024)

[6] Sun, Y., Luo, Y., Wen, X., Yuan, Y., Nie, X., Zhang, S., Liu, T., Luo, X.: TrioXpert: An Automated Incident Management Framework for Microservice Systems. arXiv:2506.10043 (2025)

[7] Jha, S., Arora, R., Watanabe, Y., Yanagawa, T., Chen, Y., Clark, J., Bhavya, B., Verma, M., Kumar, H., *et al.*: ITBench: Evaluating AI agents across diverse real-world IT automation tasks. In: Proceedings of the International Conference on Machine Learning (ICML). PMLR, vol. 267 (2025)

[8] Wang, X., Wei, J., Schuurmans, D., Le, Q., Chi, E., Narang, S., Chowdhery, A., Zhou, D.: Self-consistency improves chain of thought reasoning in language models. In: International Conference on Learning Representations (ICLR) (2023)

[9] Xu, J., Huang, T., Zhang, K.: Structured Self-Consistency: A Multi-Task Evaluation of LLMs on VirtualHome. arXiv:2602.00611 (2026). https://arxiv.org/abs/2602.00611

[10] Xiong, M., Hu, Z., Lu, X., Li, Y., Fu, J., He, J., Hooi, B.: Can LLMs express their uncertainty? an empirical evaluation of confidence elicitation in LLMs. In: International Conference on Learning Representations (ICLR) (2024). https://arxiv.org/abs/2306.13063

[11] Geifman, Y., El-Yaniv, R.: Selective classification for deep neural networks. In: Advances in Neural Information Processing Systems (NeurIPS) (2017)

[12] Wang, J., Zhou, Y., Devic, S., Fu, D.: Are LLM Decisions Faithful to Verbal Confidence? arXiv:2601.07767 (2026)

[13] Kumaran, D.: Reported Confidence in LLMs Tracks Commitment More Than Correctness. arXiv:2606.29490 (2026)