GROUND-TRUTH LIFETIME ANALYSIS: LOCALIZING FAILURES IN AGENTIC ROOT CAUSE ANALYSIS

Anonymous authors
Paper under double-blind review

ABSTRACT

Root cause analysis (RCA) agents are judged by whether their final diagnosis is correct, which tells a developer that a run failed but not where. Recent work focuses on identifying hallucination and execution errors in reasoning traces. However, we find that many failures are well-formed on their own terms: every tool call succeeds and every claim is grounded in what the agent observed, yet the true cause was already lost somewhere earlier in the run. We therefore introduce Ground-Truth Lifetime Analysis (GLA), which localizes a failed diagnosis to the stage at which the ground truth was lost — never observed, observed but never raised as a candidate, or raised but rejected. The underlying philosophy is to *make the agent's hypotheses auditable*, so GLA instruments the agent loop to record its operations on a candidate set at every step, and attributes each failure by tracking the true cause's membership in that set, with no LLM judge. Measuring four frontier models on microservice systems, we find that one stage dominates: up to 83% of failed runs observed the relevant telemetry but never formed the true cause as a candidate, while the observation gap reaches 42% in the weakest model-system pairing. GLA further exposes how agents respond to interventions, in terms that end-to-end metrics cannot capture: the variation across models, the substantial migration of runs between stages behind an unchanged total, and the bottleneck that shifts rather than clears. Our intervention improves accuracy by up to 13.03%.

1 INTRODUCTION

Large language model (LLM) agents are emerging as a promising approach to root cause analysis (RCA) in complex software systems (Roy et al., 2024; Wang et al., 2024; Pei et al., 2025; Luo et al., 2025; Mitra, 2026; Liu et al., 2026). Given an incident, an RCA agent iteratively forms hypotheses, invokes tools or generates code to inspect telemetry, and revises its diagnosis from the returned observations, until it determines a root cause, typically a component and a reason. Unlike traditional RCA pipelines (Wang & Qi, 2024; Yu et al., 2021; 2023; Zhang et al., 2021), which execute a predefined sequence of queries and rules, the agent decides what to inspect and what to compute as its hypotheses evolve. This flexibility widens the range of incidents it can investigate, but also widens the space of ways a diagnosis can go wrong. The harness compounds this: a deployed RCA agent carries developer-injected domain knowledge and a large skill library, so an error can originate in either the model or the scaffolding around it.

Improving such an agent is hard because end-to-end accuracy is largely all a developer has to work with. Faced with a batch of wrong diagnoses, there is no principled way to tell them apart or to decide where to intervene. Recent work therefore looks inside the intermediate process, scoring a reasoning trace against annotated evidence (Fang et al., 2026; Cai et al., 2026) or identifying *hallucination* and *execution errors* such as ungrounded claims and tool call failures (Barke et al., 2026; Kim et al., 2026b; Sun et al., 2026; Zhang et al., 2025; Cemri et al., 2026). These approaches examine whether the recorded reasoning is sound and whether each action was carried out correctly.

Many of the incorrect diagnoses we examine, in our production deployment and in this study, are however well-formed on their own terms. Every tool call is valid, every claim is grounded in what the agent observed, and the chain from symptom to conclusion is coherent. What went wrong lies earlier. A relevant signal was never inspected, or a misleading rule in the skill library eliminated the true cause before it was ever weighed. Neither leaves an error for existing trace-level checks to flag.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

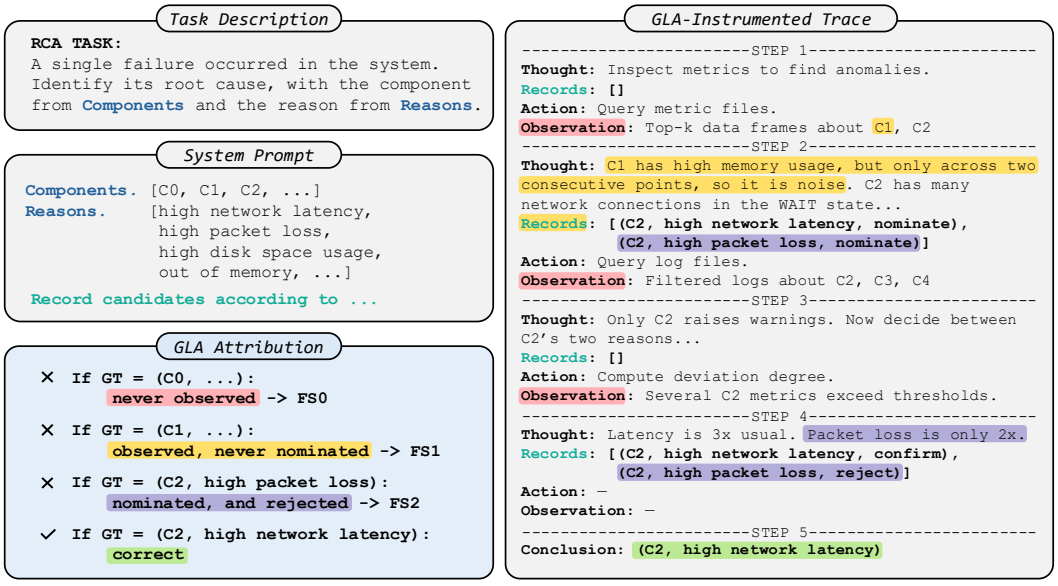


Figure 1: Overview of GLA based on a simplified diagnosis trace. GLA instruments (■) the agent loop so that each step records its operations on a candidate set. A wrong diagnosis is then attributed to the first stage the ground truth (GT) does not survive, decided by observation and GT membership in that set. Highlights mark the evidence behind each attribution.

To expose such failures, we analyze the lifetime of the true cause, asking ***at which point in the diagnostic process the ground truth was lost***. A correct diagnosis must clear three stages: observe data covering the true cause, recognize the relevant signal among concurrent and misleading ones, and settle on that cause among the candidates the agent is weighing. A diagnosis can fail at any of the three, and each calls for different interventions, targeting the mechanism behind it.

We propose to ***make the agent's hypotheses auditable in the reasoning trace***, and introduce Ground-Truth Lifetime Analysis (GLA), an analysis framework that localizes each diagnosis failure to one of these three stages (Fig. 1). As the hypotheses are often left implicit or incomplete (Wei et al., 2023; Yao et al., 2023), GLA *instruments* the agent loop to record, at each step, its operations on a *candidate set*, and tracks the membership of the true cause in it. Each failure is attributed to the first stage the ground truth does not survive — never observed, observed but never nominated as a candidate, or nominated but rejected — a partition that is exhaustive and mutually exclusive by construction. Attribution is achieved without an LLM judge, resting on that membership and the agent's own tool-call outputs. GLA requires only that the ground truth be drawn from an enumerable space rather than stated as free-form prose.

We apply GLA to four frontier models on a modified OpenRCA testbed (Xu et al., 2024). Our measurement shows that a single stage accounts for most failures: in some settings, the agent observed the relevant telemetry yet never formed the true cause as a candidate in 83% of failed runs. Meanwhile, the earlier gap, where the relevant telemetry was never observed at all, can reach 42%. GLA further exposes the challenges of optimizing an agent, in terms that end-to-end accuracy cannot capture: responses vary across models, a net change near zero hides substantial migration of runs between stages, and an intervention that relieves the stage it targets can push the bottleneck to another. Our intervention improves an agent's diagnosis accuracy by up to 13.03%, though the gain is uneven across agents.

To summarize, our key contributions are the following: (1) We introduce GLA, a framework that localizes a failed diagnosis to the stage at which the ground truth was lost, with *LLM-judge-free* attribution resting on candidate-set membership. (2) We profile stage-level failure distributions for four frontier models on three microservice systems, and explain the failure mechanisms behind them. (3) We show that GLA exposes effects of interventions that end-to-end accuracy hides, down to individual cases migrating between stages.

2 Related Work

Failure taxonomies. A body of work characterizes what failures look like in cloud RCA (Riddell et al., 2026; Kim et al., 2026a) and in multi-agent systems more generally (Cemri et al., 2025), classifying them into weak instruction adherence, evidence fabrication, hallucinated interpretations, symptom-as-cause attribution, etc. These studies focus on characterizing the behavior, with remediation not among their targets, and the causes they identify lie mostly in model capability.

Identifying hallucination and execution errors from reasoning traces. Kim et al. (2026b) evaluate reasoning soundness by checking whether intermediate claims are derivable from an evidence bank. Barke et al. (2026) check both reasoning soundness and explicit execution errors, validating constraints synthesized from the trace. Cemri et al. (2026) induce adaptive failure taxonomies from the traces of a target system. These works expose defects derivable from a conventional reasoning trace, and cannot account for wrong diagnoses that are sound at every step. In contrast, GLA explicitly tracks the hypotheses, and brings the ground truth into the analysis, which renders such runs analyzable and facilitates mitigation.

Explicit hypothesis sets. Making an agent's hypotheses or records explicit is an established way to improve its robustness and observability. Marozzo & Liò (2026) have the agent explicitly maintain competing hypotheses and update their probabilities as evidence arrives. AlSayyad et al. (2026) propose a structured logging framework for agent system observability. They share a similar philosophy with GLA, while the record schema and the analysis built atop it vary with purpose.

3 Ground-Truth Lifetime Analysis

Candidate records. An ordinary reasoning trace leaves the agent's hypothesis space implicit or incomplete (Wei et al., 2023; Yao et al., 2023). GLA therefore instruments the agent loop so that, at each step, the agent records its operations on a candidate set (Fig. 1). Each operation is a triple `(component, reason, action)`, where `component` and `reason` are drawn from an *enumerable* solution space and `action` is one of `nominate`, `confirm`, or `reject`. A step whose records violate this schema triggers a reissue. The *ground truth* also takes the same `(component, reason)` form, so its membership in the candidate set can be decided by matching. See Appendix A for details.

Failure stages. A correct diagnosis must clear three successive stages: observe data covering the true cause, recognize that cause among concurrent and misleading signals, and select it from the candidates under consideration. Thus, GLA helps localize the culprit behind an incorrect diagnosis by assigning the diagnosis to the first of these failure stages (FS) that the true cause does not survive.

FS0: Observation Failure. The agent never observes the telemetry needed to infer the ground truth.

FS1: Candidate Nomination Failure. The needed telemetry is present in the agent's observations, but the ground truth is never considered as a candidate.

FS2: Candidate Selection Failure. The ground truth is considered as a candidate, but is silently abandoned or explicitly rejected in favor of another competitor.

The three stages are mutually exclusive by construction, and attribution requires only string matching over the candidate records and the logged tool-call outputs (the agent's observations), with no LLM judge. Concretely, a wrong diagnosis is FS0 when the ground-truth `component` never appears in the agent's observation. It is FS1 when the diagnosis clears FS0 but the full ground-truth tuple `(component, reason)` is never recorded as a candidate, matched exactly with no partial credit. It is FS2 when the tuple is recorded as a candidate under any action but is not the diagnosis finally selected. We operationalize the condition for FS0 as `component` appearance rather than as sufficient evidence for the ground-truth `reason`, because doing so would require mapping each reason back to its supporting telemetry through handcrafted, deployment-specific rules that are neither unambiguous nor portable. The resulting FS0 condition is weaker but well defined, and errs toward FS1 when a run retrieved the component but not the decisive signal.

4 Where Diagnoses Lose the Ground Truth

In this section, we profile the diagnostic failures of four frontier models across three microservice systems, and explain the mechanisms behind them from an engineering perspective.

4.1 MEASUREMENT SETUP

Testbed. Our measurement is based on OpenRCA (Xu et al., 2024), a high-quality dataset of root cause analysis tasks over three microservice systems: Bank, Market, and Telecom. Each query pairs a natural-language description of an incident with a ground truth. We *normalize* every query to ask for the `(component, reason)` pair as the root cause prediction, regardless of its original task level, and restrict the measurement to queries expecting a single root cause. This leaves 261 queries: 118, 92, and 51 per system. Appendix B details the normalization.

RCA agent. The agent follows the ReAct style (Yao et al., 2023): at each step, the main LLM reads the previous observation and emits a thought, an action instruction, and its candidate records. A second LLM, the executor, carries out the action with Python code that queries a subset of the telemetry, whose volume is too large to inspect directly, and returns the result as the next observation. The loop ends when the main LLM completes the task or exhausts a 25-step budget, with a final prediction required either way. The executor could equally be developer-provided functions, API calls, or the main LLM itself; this variation does not affect our findings. We keep OpenRCA's original prompts, as they contain dataset-specific knowledge that guides the diagnosis workflow, data processing, fault identification, and root cause localization; Sec. 5 intervenes on them. While simpler than a production setup and tailored to OpenRCA, the agent's architecture and skill arrangement are standard, and the failure patterns we report match those in our own deployment on other services.

Models. We instantiate the agent with four frontier models of differing capability: OPUS-5 (Anthropic, 2026b), GPT-5.6-Sol (OpenAI, 2026), IM (a GPT-like internal model), and DeepSeek-V4-Flash (DeepSeek-AI, 2026). Each model runs at its respective default reasoning effort. Unless stated otherwise, the executor uses the same model as the main LLM.

Metrics. End-to-end *accuracy* counts a prediction as correct only when both the `component` and the `reason` match the ground truth; *score* awards partial credit of 0.5 when exactly one matches. Following Sec. 3, we also attribute every run to a failure stage or to a correct diagnosis, reporting the composition within the incorrect ones. Average total *tokens* per run measures cost. Unless stated otherwise, results are pooled over the three systems for compactness.

Measurement validity. GLA is designed to be minimally intrusive: it changes the agent only by adding the recording instruction to its prompt and a record field to each response, which the agent's existing format check enforces. Appendix C compares agents with and without GLA instrumentation. We find no systematic degradation in end-to-end accuracy, and at most a slight increase in token cost (up to $1.09\times$). Accuracy is unchanged for OPUS-5 and GPT-5.6, while IM improves from 20.7% to 26.8%, benefiting from being required to keep an explicit candidate set. We further sampled 20 runs per model and verified that their candidate records are consistent with, or a superset of, the descriptions appearing in the reasoning trace, indicating that the records faithfully reflect the agent's hypothesis space.

4.2 FAILURE COMPOSITION

Fig. 2 reports the failure composition and total failure rate of every model-system pairing.[1] The findings below proceed from the general shape of that distribution to the ways in which it moves.

> **Finding 1.** Failures concentrate at candidate formation (FS1): in most incorrect runs the ground-truth component's telemetry had already appeared in the agent's observations, yet the agent never nominated the ground truth as a candidate.

All three stages are populated in almost every model-system pairing, so failures spread across the diagnostic process rather than sitting at a single point. The distribution is nonetheless strongly skewed. As a share of incorrect runs, FS1 ranges from 39% to 83% and is the largest stage in eleven of the twelve pairings; only OPUS-5 on Telecom differs, with FS2 at 61%. This concentration exposes a bottleneck in current RCA capability: the agent either fails to retrieve the critical signals

[1] The results serve the purpose of showing the general failure composition and its trends instead of ranking models, as they are measured with the GLA-instrumented agent on the modified testbed.

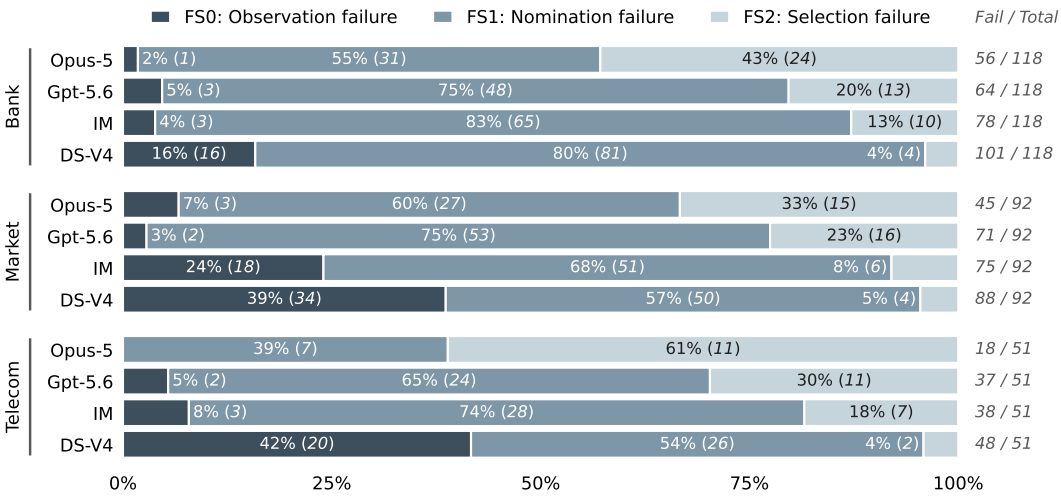


Figure 2: Failure composition by model-system pairing. Each bar splits its runs into the three stages. The share is taken over the failed runs (*Fail* on the right), with absolute counts in parentheses.

implicating the true cause, or to qualify them into candidates. The bottleneck depends not only on the model, but on the skills and tools the agent is equipped with.

Finding 2. As diagnosis capability falls, failure tends to move from the later stages of the diagnostic process toward the earlier ones.

On Bank, FS2 accounts for 43% of OPUS-5's failures, 20% of GPT-5.6's, 13% of IM's, and 4% of DS-V4's. The ordering holds in every other system. FS0 moves in the opposite direction, from 4 to 7 to 24 to 70 runs pooled across systems, and reaches 42% for DS-V4 on Telecom, where more than two fifths of failed diagnoses never observed the faulty component. A stronger agent therefore tends to lose the ground truth late, in adjudication among candidates, whereas a weaker one loses it before nomination. This is a tendency, not a law, depending on model, prompt, and task: some pairings under intervention (Sec. 5) invert it. Accordingly, composition cannot be inferred from accuracy, in either direction. On Market, GPT-5.6 and IM are close in accuracy while their FS0 shares differ eight-fold (3% vs 24%); conversely GPT-5.6 on Bank and IM on Telecom differ by 20.27 accuracy points (54/118 vs 13/51) yet their compositions agree within 4 points at every stage.

For a developer, the issue to target must therefore be measured rather than assumed. Broadening observation does little where the agent already observes the true cause and then discards it, and sharpening adjudication does little where the bottleneck lies in observation.

4.3 ENGINEERING ORIGINS OF FAILURES

Through case studies, we identify the mechanisms below. They are *non-exhaustive* and focus mainly on the engineering aspects of how the ground truth gets lost, complementing the general LLM limitations discussed in Sec. 2.

Open-ended workflow admits incomplete and error-prone exploration. Developers typically specify the diagnostic workflow in prose. This affords flexibility, but unlike code-defined pipelines that fix the sequence of queries, it leaves telemetry coverage to the agent's discretion. Given the volume, which in production spans thousands of machines, the agent inspects selectively, often fewer than ten, and readily omits the root cause site. Agents further rely on reduction strategies such as top-k filtering and `head` operations (pandas, 2026; MacKenzie & Meyering, 2026); weaker models adopt aggressive parameters ($k \leq 10$ or the first ten rows) more often, which can discard the ground truth at the earliest stage, while stronger ones analyze more systematically. The median truncation size is 30 for OPUS-5, against 10 for GPT-5.6 and IM, and 20 for DS-V4 (Appendix D). Granting the agent autonomy over query generation also admits errors that waste steps, such as format-rejected responses and unexecutable query code the model fails to fix. Weaker agents err more often, cutting

Table 1: Behaviors behind FS0 runs in Fig. 2. *Small-k*: runs containing aggressive top-k filtering or truncation ($k \leq 10$). *HitBudget*: runs that exhaust the step budget and are forced to answer. *ErrStep≥3*: runs containing ≥3 steps discarded due to errors. The last column approximates runs that exhaust the budget due to errors rather than long valid reasoning. Details in Appendix D.

Model	FS0 total	Small-k	HitBudget	HitBudget & ErrStep≥3
OPUS-5	4	0	0	0
GPT-5.6	7	5	0	0
IM	24	20	13	9
DS-V4	70	49	56	42

their exploration short or exhausting the budget and being forced to answer. Most FS0 runs exhibit these behaviors (Table 1). Incomplete exploration leads to FS0 when the ground-truth component is omitted entirely, and to FS1 when it is inspected but the critical signal is not.

Harness truncation and silent tool faults distort what the agent observes. For cost and context efficiency, truncation is standard in harness implementations; a single tool call may be capped at a few hundred output lines. The guard is reasonable in isolation, but curtails observation whenever the agent queries a large file without narrowing the result first. Agents also depend on external APIs, whose faults can be silent — worse than an outright failure, because the agent cannot tell anything went wrong. In our production deployment, a code search tool returned commit messages instead of the snippets it was designed to provide, raising no error. Such messages are rarely complete enough to convey the code change, leaving the agent to reason soundly from a distorted view and arrive at a wrong diagnosis. Concern about the reliability of such dependencies grows with deployment scale, where they are numerous and evolve rapidly. Like the previous mechanism, this one leads to both FS0 and FS1, depending on whether the ground-truth component survives the distortion.

Empirical rules encoded in prompts reject or misrank genuine signals. Developers encode diagnosis knowledge as rules that typically carry hard thresholds, even when phrased as suggestions. Yet no fixed threshold fits every incident, and seemingly reasonable rules reject genuine anomalies. Our testbed agent's prompt states: "`A fault is a consecutive sub-series of a specific component-KPI time series. . . Filter out isolated noise spikes to locate faults. . .` " This prevents the ground truth from being nominated when its sub-series spans only one or two points, violating *consecutivity*, even though the metric is sampled once per minute. A companion rule, "`If multiple faulty components are identified,. . . the root cause level should be determined by the fault that shows the most significant deviation from the threshold. . .` ", further misleads adjudication because deviations between different indicators are not comparable as a severity ordering. It also leaves deviation undefined, so the agent chooses arbitrarily among an absolute difference, a ratio, and a number of standard deviations. The mechanism leads to FS1 when a rule fires before nomination, and to FS2 when it fires after the ground truth is already a candidate.

Causally related root causes can be hard to separate from downstream evidence. Root cause analysis reasons backwards from downstream manifestations to an upstream cause, which can leave closely related causes hard to tell apart. Every agent in our testbed confuses packet loss with network latency on a set of cases where the provided signals separating them appear only in indirect metrics such as network error counts and per-state TCP connection counts, with no direct indicator of end-to-end latency or loss rate. Disk-space exhaustion and high disk I/O pose the same difficulty, being causally related and frequently concurrent. Faced with such a pair, an agent can be drawn to the wrong member. The mechanism leads to FS1 when only the wrong member is nominated, and to FS2 when both are nominated and the wrong one is selected.

Finding 3. Diagnosis failures arise not only from model capability but from the agent's engineering, and each such mechanism can surface at a different failure stage depending on how far the diagnosis has progressed when the mechanism takes effect.

Every mechanism above originates in the agent's engineering: the open-ended workflow, the harness, the prompt rules, and the available telemetry. Each can lead to more than one failure stage, and where it surfaces depends on how far the diagnosis has progressed when it takes effect, and that

Table 2: Overview of prompt interventions. +/− for added/removed prompts. PI-1 and PI-2 edit the original prompt; PI-3 and PI-4 build on PI-2 and carry the rule it adds. Details in Appendix E.

ID	Target	Description of major prompt changes
PI-1	Inter-agent context loss	Keep all evidence within the main LLM's context. `+ Do not delegate the fault qualification decision to the Executor. Have the Executor return raw evidence data alongside its summary. Apply filters only in the main LLM, not in the Executor.`
PI-2	Incomplete exploration	Encourage exploration. `+ Inspect all KPIs of each component before narrowing the search. Do not skip any KPI.`
PI-3	Rules that reject signals	Remove empirical rules about fault identification. `− Consecutivity requirement for fault definition. Filtering isolated noise spikes. Excluding near-threshold signals as false positives.`

PI-4	Rules that misrank faults	Revise how competing faults are ranked.
		`− Ranking candidate root cause levels by deviation from the threshold.`
		`+ Use deviation from the threshold only to surface anomalies. Treat a threshold breach as a fault only if the series rarely reaches that level outside the failure window.`

on the decisions the model makes along the way. A stronger model fails less often overall but does not remove these mechanisms, and in some cases the limit is absolute: where the available telemetry cannot distinguish two causes, no model can either, and a correct choice between them is a lucky guess rather than an inference. We therefore call for attention to the engineering of RCA agents, which in principle admits remedies that model capability alone does not.

5 Interventions on Failure Mechanisms

In this section, we design four prompt interventions (Table 2) to show that targeted mechanisms are remediable. Note that the space of mitigations is broader in practice — a more stable workflow, more reliable dependencies, harness and tool design, and telemetry that separates causally related faults — and the choice depends on the mechanism behind the failed diagnosis.

PI-1 and PI-2 target incomplete observation: PI-1 preserves the evidence the executor's filtering discards and the context lost between the two LLMs, and PI-2 requires the agent to inspect *all* Key Performance Indicators (KPIs) of each component before narrowing the search. PI-3 and PI-4 edit empirical rules that reject or misrank genuine signals. We locate such rules in the reasoning text of the step at which a run loses the ground truth: for FS2, the step that rejects it; for FS1, the step that promotes a candidate with the correct `component` but the wrong `reason`. PI-3 removes the rejecting rules without replacement, whereas PI-4 revises the misranking ones. Both are layered on PI-2 and carry the rule it adds, so we compare them against PI-2 rather than the original prompt.

Each intervention edits the prompt alone and is re-measured over the entire query set. We analyze their effect through end-to-end accuracy (Table 3) and a *migration matrix* (Fig. 3), cross-tabulating each run's stage before and after the intervention. Every intervention produces a significant response in at least one agent, but which agents respond, and whether they improve or degrade, varies.

Finding 4. The response to the same prompt change can vary widely by model, so an intervention validated on one model need not hold on another.

Table 3 shows this variation already in end-to-end accuracy: under each PI group, at most three of the four models shift significantly, and the shifts run in both directions. In PI-1, the first three models move by a similar amount (-4.21%, -2.68% and -3.07%) while their failure composition separates them. Under Opus-5, FS1 grows by 33 runs, almost all drawn from FS2 and correct diagnoses, so they lose the ground truth earlier than under the base prompt. Reading their traces, we find that Opus-5 takes the instruction to filter in the main LLM literally, applying the fault identification rules more strictly and rejecting the ground truth at nomination rather than passing it on; neither its base runs nor the other models under PI-1 show this. Under IM, failures instead pile into FS2 ($+30$), moving the point of loss from observation to selection. *Such variation raises a*

Table 3: Change under every intervention. The stage columns give the net change in the number of runs; the accuracy column reports the change in the share of correct diagnoses.

	PI-1				PI-2				PI-3				PI-4			
Model	FS0	FS1	FS2	Accuracy	FS0	FS1	FS2	Accuracy	FS0	FS1	FS2	Accuracy	FS0	FS1	FS2	Accuracy
OPUS-5	−3	+33	−19	−4.21%	+2	−8	+2	+1.53%	−3	+3	+3	−1.15%	−4	+6	−10	+3.07%
GPT-5.6	+8	−11	+10	−2.68%	−3	−20	+9	+5.36%	−2	+2	+16	−6.13%	−2	+8	+5	−4.21%
IM	−6	−16	+30	−3.07%	+0	−11	+15	−1.53%	+3	+3	−2	−1.53%	+9	−19	−6	+6.13%
DS-V4	−45	−4	+15	+13.03%	−22	−13	+10	+9.58%	−16	+11	+11	−2.30%	+23	+12	−8	−10.34%



Figure 3: How runs migrate across failure stages under each intervention. Each cell counts runs moving from its row's stage to its column's, with COR a correct diagnosis. *In* and *Out* sum the runs that enter and leave each stage. Green marks runs whose ground truth survived further than before; red marks those lost earlier. Saturation scales with the run count. A cell is outlined where its traffic significantly outweighs the reverse (McNemar's exact test, $p < 0.05$).

concern that a tailored skill library, common practice in production, need not carry over from one model to the next, and that each migration may require extra adaptation.

Finding 5. Accuracy moves little under many interventions, but the failure stages beneath it churn: from a quarter to over half of all runs change stage, and they move in both directions.

No intervention shifts accuracy by more than 13.03%, yet far more runs change stage: 70 (27%) in the quietest configuration (Fig. 3d) and 144 (55%) in the busiest (Fig. 3p). The same holds within a stage: FS1 under PI-3 barely moves for IM (+3), the net of 42 runs entering and 39 leaving

(Fig. 3k). A net change near zero can therefore be the residue of two large opposing flows rather than a null effect. Churn on this scale has two sources: the edit genuinely helps some diagnoses and hurts others, and the agent is nondeterministic, since the models accept no temperature setting (Anthropic, 2026a; Microsoft, 2026). Our measurement cannot separate the two, so we ground our findings in McNemar's exact test and report only the significant pairs.

Finding 6. An intervention can mitigate its target mechanism without raising accuracy, when it shifts the bottleneck to a later stage instead of clearing it.

DS-V4 gains the most from PI-1 (+13.03%) and PI-2 (+9.58%), the two interventions targeting incomplete observation. Their migration matrices (Figs. 3m and 3n) trace the gain to FS0→COR and FS1→COR, with a further 40 runs moving FS0→FS1 under each, relieving the bottleneck Fig. 2 exposes; GPT-5.6 improves the same way under PI-2, gaining 5.36% through FS1→COR (Fig. 3f). Mitigation does not guarantee the gain, however. Under the same two interventions, the runs IM frees from earlier stages land in FS2 rather than COR (Figs. 3i and 3j), and its accuracy falls in both. Broadening observation there only moves the bottleneck to selection, where the agent fails to choose among the candidates it now sees. *Yet that does not mean the edit should be abandoned*: an intervention targeting selection should be layered on top, which is how PI-3 and PI-4 build on PI-2, and PI-4 further improves IM by 6.13 accuracy points (Table 3).

Finding 7. Deleting a harmful rule need not unblock the stage it obstructed: without the knowledge it encoded, the cost reappears later in the process.

PI-3 strips the consecutivity requirement, the noise-spike filter and the near-threshold exclusion from the PI-2 prompt. Accuracy falls under all four models (-1.15%, -6.13%, -1.53%, -2.30%), though only GPT-5.6's drop is significant. No model's FS1 decreases ($+3$, $+2$, $+3$, $+11$; Table 3), so the removal does not unblock the nomination stage it targeted. What it does instead is a knowledge regression, visible in Fig. 3g: 20 runs move from a correct diagnosis to FS2 against 7 returning ($p = 0.019$), and FS2 gains 16. Deleting the rules leaves weaker criteria for what counts as a fault, and the cost lands at selection rather than nomination. Recent appeals to simplify system prompts for Claude 5 models (Shihipar, 2026) therefore do not transfer without qualification.

6 DISCUSSION

GLA localizes a failure to a stage (Sec. 3), not to the site producing it, and supplies no direct fix. That remains human work: the composition (Sec. 4.2) narrows where to look, the mechanisms suggest why the stage fails, and the developer traces the suspicious steps (Sec. 5) back to a source in the engineering (Sec. 4.3) or in a model limitation (Sec. 2), then designs the edit. For developers automating this with an LLM judge, we envision that GLA still facilitates the analysis with a firmer foundation: stage attribution, step anchors, and migration matrices beyond the raw trace.

Adopting GLA requires the solution space to be enumerable. Many settings meet this, but an agent whose output is free prose does not, and GLA does not apply to it directly. Optimizing such an agent is hard for the same reason: scoring free text is both ambiguous and subjective. Thus, a tailored, enumerable solution space is worth building for analysis alone, and GLA comes with it.

7 CONCLUSION

In this study, we present GLA, an analysis framework that localizes a wrong RCA diagnosis to one of three failure stages: observation, nomination, and selection. Our measurement shows that the loss of ground truth concentrates at observation and nomination, and that the failure composition shifts with model capability. We draw attention to the engineering factors that contribute to these failures. GLA further uncovers the challenges of intervening on them, none of which end-to-end metrics capture: the diversity of model responses, the large migration behind an unchanged total, and the bottleneck that shifts rather than clears. With GLA, developers can gain a deeper understanding of their agents and design optimizations more systematically.

AI USE STATEMENT

We used generative AI tools for polishing the writing, generating scripts for simple data analysis and plotting, and helping analyze the failure reasons behind some of the wrong diagnosis cases discussed in the paper, with the resulting case studies verified by the authors. The text of the paper was drafted by the authors, with the tools used only to improve its presentation, and the authors reviewed every generated script and checked the correctness of its output. All insights, findings, and conclusions are the authors' own. We did not use generative AI tools for ideation, design, implementation of core components, or orchestration of experiments. We take responsibility for the final content of this work, including any text, claims, or artifacts produced with the aid of generative AI.

REPRODUCIBILITY STATEMENT

The source code and the main result data will be released upon acceptance, and are provided as supplementary material during review. The release covers the dataset, the task queries, the RCA agent implementation, the GLA implementation, the data analysis scripts, and the diagnosis traces. The results for OPUS-5, GPT-5.6, and DS-V4 are intended to be reproducible, given API keys obtained from the respective commercial providers; IM is excluded from reproduction, and its run artifacts are not released.

REFERENCES

Adam AlSayyad, Kelvin Yuxiang Huang, and Richik Pal. AgentTrace: A structured logging framework for agent system observability, February 2026.

Anthropic. API usage primer for Claude. Claude Platform Documentation, 2026a. URL https://platform.claude.com/docs/en/claude_api_primer. Accessed: 2026-09-18.

Anthropic. Claude Opus 5 system card. Anthropic, July 2026b. URL https://www-cdn.anthropic.com/b514064af1408018e64b1ad24e7d5e75850b4ffd/Claude%20Opus%205%20System%20Card.pdf. Accessed: 2026-09-20.

Shraddha Barke, Arnav Goyal, Alind Khare, Avaljot Singh, Suman Nath, and Chetan Bansal. AgentRx: Diagnosing AI agent failures from execution trajectories, February 2026.

Yuanhong Cai, Xiaohui Nie, Kanglin Yin, Changhua Pei, Yongqian Sun, Shenglin Zhang, Haibin Liu, Guiyang Liu, Xidao Wen, Fang Situ, and Dan Pei. A multi-dataset benchmark for evaluating LLM agents in microservice failure diagnosis, June 2026.

Mert Cemri, Melissa Z. Pan, Shuyi Yang, Lakshya A. Agrawal, Bhavya Chopra, Rishabh Tiwari, Kurt Keutzer, Aditya Parameswaran, Dan Klein, Kannan Ramchandran, Matei Zaharia, Joseph E. Gonzalez, and Ion Stoica. Why do multi-agent LLM systems fail?, October 2025.

Mert Cemri, Andrei Cojocaru, Melissa Pan, Shu Liu, Shubham Agarwal, Alexander Krentsel, Jay Tang, Kannan Ramchandran, Joseph E. Gonzalez, Matei Zaharia, Alex Dimakis, and Ion Stoica. Fantastic adaptive taxonomies and how to use them, July 2026.

DeepSeek-AI. DeepSeek-V4-Flash. Hugging Face, 2026. URL https://huggingface.co/deepseek-ai/DeepSeek-V4-Flash. Accessed: 2026-09-21.

Aoyang Fang, Yifan Yang, Jin'ao Shang, Qisheng Lu, Junjielung Xu, Rui Wang, Songhan Zhang, Yuzhong Zhang, Boxi Yu, and Pinjia He. OpenRCA 2.0: From outcome labels to causal process supervision, June 2026.

Taeyoon Kim, Woohyeok Park, Hoyeong Yun, and Kyungyong Lee. Why do AI agents systematically fail at cloud root cause analysis?, March 2026a.

Wonjoong Kim, Sangwu Park, Yeonjun In, Sein Kim, Dongha Lee, and Chanyoung Park. Beyond the final answer: Evaluating the reasoning trajectories of tool-augmented agents, May 2026b.

Phillip Liu, John Wu, and Uttam Thakore. Taming AI infrastructure failures with agentic debugging. @Scale: Systems & Reliability, June 2026. URL https://atscaleconference.com/events/systems-reliability-2026. Accessed: 2026-09-20.

Yu Luo, Jiamin Jiang, Jingfei Feng, Lei Tao, Qingliang Zhang, Xidao Wen, Yongqian Sun, Shenglin Zhang, Jielong Huang, Nan Qi, and Dan Pei. From observability data to diagnosis: An evolving multi-agent system for incident management in cloud systems, October 2025.

David MacKenzie and Jim Meyering. head(1): Output the first part of files. GNU Coreutils Manual Page, April 2026. URL https://man7.org/linux/man-pages/man1/head.1.html. Accessed: 2026-09-20.

Fabrizio Marozzo and Pietro Liò. LLM-as-an-Investigator: Evidence-first reasoning for robust interactive problem diagnosis, June 2026.

Microsoft. Azure OpenAI reasoning models. Microsoft Foundry Documentation, 2026. URL https://learn.microsoft.com/en-us/azure/foundry/openai/how-to/reasoning. Accessed: 2026-09-18.

Gaurav Mitra. Teaching AI to fight fires: Autonomous reliability agents at Meta scale. @Scale: Systems & Reliability, June 2026. URL https://atscaleconference.com/events/systems-reliability-2026. Accessed: 2026-09-20.

OpenAI. GPT-5.6 system card. OpenAI, July 2026. URL https://deploymentsafety.openai.com/gpt-5-6. Accessed: 2026-09-20.

pandas. pandas.DataFrame.nlargest. pandas Documentation, 2026. URL https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.nlargest.html. Accessed: 2026-09-20.

Changhua Pei, Zexin Wang, Fengrui Liu, Zeyan Li, Yang Liu, Xiao He, Rong Kang, Tieying Zhang, Jianjun Chen, Jianhui Li, Gaogang Xie, and Dan Pei. Flow-of-Action: SOP enhanced LLM-based multi-agent system for root cause analysis. In *Companion Proceedings of the ACM Web Conference 2025*, 2025. doi: 10.1145/3701716.3715225.

Evelien Riddell, James Riddell, Gengyi Sun, Michał Antkiewicz, and Krzysztof Czarnecki. Stalled, biased, and confused: Uncovering reasoning failures in LLMs for cloud-based root cause analysis, January 2026.

Devjeet Roy, Xuchao Zhang, Rashi Bhave, Chetan Bansal, Pedro Las-Casas, Rodrigo Fonseca, and Saravan Rajmohan. Exploring LLM-based agents for root cause analysis, March 2024.

Thariq Shihipar. The new rules of context engineering for Claude 5 generation models. Anthropic Blog, July 2026. URL https://claude.com/blog/the-new-rules-of-context-engineering-for-claude-5-generation-models. Accessed: 2026-09-17.

Yuxi Sun, Aoqi Zuo, Haotian Xie, Wei Gao, Mingming Gong, and Jing Ma. FACT-E: Causality-inspired evaluation for trustworthy Chain-of-Thought reasoning. 2026. URL https://arxiv.org/abs/2604.10693.

Tingting Wang and Guilin Qi. A comprehensive survey on root cause analysis in (micro) services: Methodologies, challenges, and trends, July 2024.

Zefan Wang, Zichuan Liu, Yingying Zhang, Aoxiao Zhong, Jihong Wang, Fengbin Yin, Lunting Fan, Lingfei Wu, and Qingsong Wen. RCAgent: Cloud root cause analysis by autonomous agents with tool-augmented large language models. In *Proceedings of the 33rd ACM International Conference on Information and Knowledge Management*, 2024. doi: 10.1145/3627673.3680016.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. Chain-of-Thought prompting elicits reasoning in large language models, January 2023.

Junjielong Xu, Qinan Zhang, Zhiqing Zhong, Shilin He, Chaoyun Zhang, Qingwei Lin, Dan Pei, Pinjia He, Dongmei Zhang, and Qi Zhang. OpenRCA: Can large language models locate the root cause of software failures? In *The Thirteenth International Conference on Learning Representations*, October 2024.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models, March 2023.

Guangba Yu, Pengfei Chen, Hongyang Chen, Zijie Guan, Zicheng Huang, Linxiao Jing, Tianjun Weng, Xinmeng Sun, and Xiaoyun Li. MicroRank: End-to-end latency issue localization with extended spectrum analysis in microservice environments. In *Proceedings of the Web Conference 2021*, 2021. doi: 10.1145/3442381.3449905.

Guangba Yu, Pengfei Chen, Yufeng Li, Hongyang Chen, Xiaoyun Li, and Zibin Zheng. Nezha: Interpretable fine-grained root causes analysis for microservices on multi-modal observability data. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 553–565, 2023. doi: 10.1145/3611643.3616249.

Shaokun Zhang, Ming Yin, Jieyu Zhang, Jiale Liu, Zhiguang Han, Jingyang Zhang, Beibin Li, Chi Wang, Huazheng Wang, Yiran Chen, and Qingyun Wu. Which agent causes task failures and when? on automated failure attribution of LLM multi-agent systems, June 2025.

Yingying Zhang, Zhengxiong Guan, Huajie Qian, Leili Xu, Hengbo Liu, Qingsong Wen, Liang Sun, Junwei Jiang, Lunting Fan, and Min Ke. CloudRCA: A root cause analysis framework for cloud computing platforms. In *Proceedings of the 30th ACM International Conference on Information and Knowledge Management*, pp. 4373–4382, 2021. doi: 10.1145/3459637.3481903.

A GLA Implementation Details and Examples

GLA instruments the agent through its system prompt and its control loop. Listing 1 shows the prompts, with the lines GLA adds in green italics. At each step, the LLM responds in the JSON `format`, which now includes a `candidates` field[2]. The loop parses the response and validates each candidate record: its `component` and `reason` must come from the provided {`cand`} list, and its `action` must be one of the allowed values. This extends the format check the agent already performs. A response that fails either check is discarded without executing its instruction, and the next step asks the LLM to correct the format; the discarded step still counts toward the step budget. Our implementation uses `initiate`, `kill`, and `confirm` as the `action` values, corresponding to `nominate`, `reject`, and `confirm` in Sec. 3.

Listing 1: System prompts

```
system_prompt = """The issue you are going to solve is:
    {objective}
    {cand_prompt}
    Solve the issue step-by-step. In each step, your response should follow the JSON
     format below:
    {format}
    Let's begin.
"""

objective = """..."""

cand_prompt = """ In the process of reasoning, you should record the possible root
     cause components and reasons for the issue on the fly. A candidate record
     should be in the format of a tuple of (component, reason, action). The
     component and reason should be selected from the provided list: {cand}. `action
     ` must be one of the following: "initiate", "kill", "confirm". The semantics of
      a candidate record is: "In this step of analysis, I want to initiate / kill /
     confirm a hypothesis that the root cause is in `component` and the reason is `
     reason`".
"""

format = """{
    "analysis": (Your analysis of the code execution result from Executor in the last
     step, with detailed reasoning of 'what has been done' and 'what can be derived'.
     Respond 'None' if it is the first step.),
    "completed": ("True" if you believe the issue is resolved, and an answer can be
     derived in the 'instruction' field. Otherwise "False"),
    "instruction": (Your instruction for the Executor to perform via code execution in
     the next step. Do not involve complex multi-step instruction. Keep your
     instruction atomic, with clear request of 'what to do' and 'how to do'. Respond a
     summary by yourself if you believe the issue is resolved. Respond a summary by
     yourself if you believe the issue is resolved. Respond a summary by yourself if
     you believe the issue is resolved.),
    "candidates": (In this step of analysis, which candidate hypotheses do you
     initiate / confirm / kill? Format: [("component", "reason", "action"), ...].
     You don't need to maintain the full list of past candidate records, just record
      this turn.)
}
...
"""
```

Listing 2 shows the candidate records of one run. Each step holds a list of operations on candidates, reflecting the agent's hypotheses and decisions at that step. An empty list is an accepted step that records nothing, and step 11 is absent because its response failed to parse and was discarded. Half of the ten candidates follow the standard trajectory: initiated, then killed. The others move more freely. (`apache01, high disk I/O read usage`) is initiated at step 4, killed at step 5, and confirmed at

[2]The prompt specifies each record as a tuple, (`"component", "reason", "action"`). Since JSON has no tuple type, a stricter instruction would specify a list of lists, [[`"component", "reason", "action"`], `...`].

step 12; and (`Mysql01, high disk I/O read usage`) is initiated again at step 9 after being killed at step 5. (`apache01, high disk space usage`) is killed at step 8 without ever being initiated. These are valid trajectories: an agent may revisit an earlier decision, or rule out a hypothesis it never stated. We deliberately impose no further constraints, to keep the instrumentation minimally intrusive: the runtime check only verifies that each record conforms to the schema, never the order of operations on a candidate.

Listing 2: Candidate records of one run, keyed by step (0-indexed).

```
{
  "0": [],
  "1": [],
  "2": [],
  "3": [
    ["Tomcat01", "network packet loss", "initiate"]
  ],
  "4": [
    ["apache01", "high disk I/O read usage", "initiate"],
    ["Mysql01", "high disk I/O read usage", "initiate"]
  ],
  "5": [
    ["apache01", "high disk I/O read usage", "kill"],
    ["Mysql01", "high disk I/O read usage", "kill"],
    ["Tomcat01", "high CPU usage", "initiate"],
    ["Tomcat01", "high JVM CPU load", "initiate"]
  ],
  "6": [
    ["Tomcat01", "high CPU usage", "kill"],
    ["Tomcat01", "high JVM CPU load", "kill"]
  ],
  "7": [
    ["Tomcat01", "network packet loss", "kill"],
    ["Tomcat01", "network latency", "kill"]
  ],
  "8": [
    ["apache01", "high disk space usage", "kill"],
    ["Mysql01", "high disk space usage", "kill"]
  ],
  "9": [
    ["Tomcat01", "network latency", "kill"],
    ["Mysql01", "high disk I/O read usage", "initiate"]
  ],
  "10": [
    ["Redis01", "high memory usage", "initiate"],
    ["Redis01", "high CPU usage", "initiate"]
  ],
  "12": [
    ["Redis01", "high memory usage", "kill"],
    ["Redis01", "high CPU usage", "kill"],
    ["apache01", "high disk I/O read usage", "confirm"]
  ]
}
```

Listing 3 shows the `analysis` field of each step in the same run; we omit the observations for space. The ground truth is (`Mysql02, high memory usage`), the agent answers (`apache01, high disk I/O read usage`), and GLA attributes the run to FS1: the ground truth was observed but never nominated, pinpointing that `Mysql02` is first observed at step 9. The records show how it was lost. At step 9, the agent notes a mild anomaly on `Mysql02` in the metrics, but finds that traces cannot resolve the database tier and turns to logs. At step 10, it finds that logs do not cover that tier either. It then pursues the one clear in-window signal, on `Redis01`, and `Mysql02` is never nominated. This guides developers to investigate *why* `Mysql02` *is invisible in the logs*, and to decide the next action depending on whether this is a limitation of the executor's queries or a genuine gap in the logs. Likewise, in an FS2 run, where the ground truth is nominated but not selected, the records point to the step that rejects it, and developers can inspect the rules that underweight the ground truth.

Listing 3: The `analysis` field of each step in the same run, keyed by step (0-indexed).

```
====================================== Step 0 ======================================
None
====================================== Step 1 ======================================
The metric file covers the full day at 1-minute granularity, with 18 cmdb_ids (14
candidate components plus 4 docker hosts) and 349 KPIs. Next I must follow the
workflow: compute global thresholds per (component, KPI) over the whole day first,
then filter the 14:30-15:00 failure window and detect consecutive anomalies.
====================================== Step 2 ======================================
No sustained P99 breaches were found in the failure window for the 14 candidate
components. This suggests either the anomaly is a drop (below threshold) rather than a
spike, or the deviation is milder (P95-level), or the KPI affected is not a spike-type
KPI. Before loosening further, I should confirm the actual failure onset time using
the app-level business KPIs (rr, sr, cnt, mrt) which should degrade during the failure.
====================================== Step 3 ======================================
The app KPIs show a clear, synchronized anomaly: at 14:44 three services
(ServiceTest6, 7, 9) had `rr` (request rate) dip below 100% and below their whole-day
P5, while `sr` never dropped. A brief request-rate drop with no success-rate drop
across several services is typical of a transient network-level problem (e.g., packet
loss / latency) on a shared component. Since no container KPI had 3+ consecutive
points above global P99, I need to relax the detection: scan both the upper side (>=
P95) and the lower side (<= P5) for all candidate-component KPIs, allowing shorter
runs, and rank by deviation magnitude.
====================================== Step 4 ======================================
The relaxed scan produced 1083 runs, most with small deviations (noise-level). Two
things stand out:

1. apache01 `LOCALDISK-sdb_DSKRTps` (disk read transactions/s) reached 1.1167 vs.
global P95 of 0.0167 and whole-day median 0 -- a ~66x deviation, starting 14:31, i.e.
by far the largest relative deviation in the whole window. This is consistent with
'high disk I/O read usage'.

2. Mysql01 has many mild MySQL-internal workload anomalies (dirty pages,
ThreadsRunning, slow queries, scan/rows-read bursts) but their deviations are mostly
<50% and MySQL workload metrics are often just consequences of traffic, not causes.

The app-level rr dip at 14:44 (no sr drop) is a mild traffic anomaly. To decide, I
must inspect apache01's disk-related raw series in the window in detail (is it a
sustained fault or an isolated spike?) and compare against Mysql01's disk read series.
====================================== Step 5 ======================================
Key insight: the apache01 disk-read event actually begins at ~15:08, i.e. OUTSIDE the
14:30-15:00 failure window, and within the window apache01's disk-read values only sit
between P95 and P99 (mild). So apache01 high disk I/O read is a different, later
incident and should be killed as the answer for this issue. Mysql01's disk was
completely idle (all reads 0.0) during the window, so Mysql01 'high disk I/O read
usage' is also killed; its MySQL-internal metrics (ThreadsRunning=2, dirty pages,
scans) are mild and workload-driven.

What remains notable inside the window is Tomcat01: `OSLinux-CPU_CPU-0_SingleCpuidle`
dropped to 0% idle (whole-day median 91.5%, P5 = 22.5) in two runs at 14:37 and 14:56
- a 100% deviation to the floor, meaning CPU core 0 was fully saturated. Given the
sampling is on alternating minutes, those 2-point runs span ~4 minutes each. This
points to a CPU/JVM-CPU-type fault on Tomcat01. I need to inspect all Tomcat01 CPU and
JVM KPIs (CPUCpuUtil, CPULoad, per-core utilization, JVM_JVM_CPULoad, JVM heap usage,
Tomcat JVM memory percent) in the window against their whole-day thresholds to
determine whether it is 'high CPU usage' or 'high JVM CPU load'.
====================================== Step 6 ======================================
CPU, memory, JVM-CPU and JVM-heap causes are now ruled out for the 14:30-15:00 window
(all deviations tiny or whole-day sampling artifacts). The apache01 disk-read
escalation clearly begins at ~15:08, i.e. outside the stated window, so it belongs to
a later incident and cannot be the root cause of the 14:30-15:00 failure. Mysql01's
disks were completely idle in the window.
```

```
The only genuine in-window symptom found so far is at the application layer: at 14:44
the request rate (rr) of ServiceTest6, ServiceTest7 and ServiceTest9 dropped below
100% and below their whole-day P5, while success rate stayed at 100%. A short
simultaneous drop in request volume without any error increase is the classic
fingerprint of a network-level fault (packet loss or added latency) on a shared
component. I have not yet examined network KPIs. I must now check all network-related
KPIs (packets in/out, input/output errors and error percentages, bandwidth
utilization, KB/s, TCP connection counts, CLOSE_WAIT/FIN_WAIT) for the 14 candidate
components, in both directions (above P95/P99 for errors, below P5 for packet
counts/throughput which would drop under packet loss).
====================================== Step 7 ======================================
Network-level causes are now ruled out for the window: NETInErr/NETOutErr are exactly
0 on all 14 components, packet rates are stable, and the CLOSE_WAIT/FIN_WAIT 'runs'
are degenerate zero-valued series. The only mild signal was Redis01 TotalTcpConnNum
dipping to 506 vs P5=603 for two samples at 14:37 (16% deviation, likely a symptom).

So far I have examined: CPU, per-core CPU, memory percent, JVM CPU load, JVM heap,
network, and disk I/O for only apache01/apache02/Mysql01. Two candidate reasons remain
unexamined across the full component set: **high disk space usage** (FSUsedSpace /
FSAvailableSpace / FSInodeUsedPercent) and **high disk I/O read usage on the other 11
components**. A disk-space-filling fault would show a sharp monotonic rise in
FSUsedSpace within the window and would not necessarily disturb CPU/network - which
matches the near-silent metric picture we observed. I must check these next.
====================================== Step 8 ======================================
Disk space usage is now ruled out for all 14 components (all filesystem series either
flat or growing at their normal linear log-growth pace, none exceeding whole-day P99).
So far the metric analysis has ruled out: high CPU usage, high memory usage, high JVM
CPU load, JVM OOM heap, network latency, network packet loss, high disk space usage,
and high disk I/O read usage on apache01/apache02/Mysql01 within the window
(apache01's true disk event starts ~15:08, outside the window).

The only in-window symptoms are: (a) rr drop at 14:44 on ServiceTest6/7/9 with no sr
drop; (b) Mysql01 showing a persistent ThreadsRunning=2 for the whole window plus
dirty-buffer-pool pages, slow queries and full-scan bursts at 14:36 and 14:47; (c)
Redis01 TCP connection count dip at 14:37. Metrics alone are inconclusive, so per the
rules I must now move to trace analysis to see which component exhibits abnormal
latency during 14:30-15:00.
====================================== Step 9 ======================================
Trace analysis found no latency anomaly in the window (all 12 traced components
actually had window mean durations 40-60% BELOW their whole-day means, and the
fraction of spans over global P99 was under the 1% baseline everywhere). This kills
'network latency' as a cause and also indicates no Tomcat/MG/IG container was slow.
Note that Mysql01/02, Redis01/02, apache01/02 are not instrumented in traces, so the
un-traced tier remains the suspect tier.

Recapping the in-window metric evidence, the single component with by far the broadest
and most persistent set of in-window anomalies is **Mysql01**: `Innodb buffer pool
pages dirty` = 90 vs global P95 20 (day median 0) starting 14:42; `ThreadsRunning` = 2
sustained for the entire 21-sample window; `SlowQueries` run starting 14:39; bursts of
`Rows Read`/`Handler Read Rnd Next`/`Select Scan`/`Sort Scan`/`Innodb buffer pool
read requests` at 14:36 and 14:47; `CPUWio` elevated at 14:41 and 14:52; `CPULoad` run
from 14:52; and a `sdc_DSKPercentBusy` = 87.2 spike at 14:31 vs day P95 6.4 / P99
19.1. Mysql02 also shows a mild Select Scan run. This pattern (I/O wait + slow queries
+ dirty pages + device busy, with client-visible rr dips at 14:44 but no errors) is
characteristic of a storage/IO-level problem at the database host. Since traces cannot
resolve this tier, I must now use log analysis to determine the faulty component and
the resource reason.
====================================== Step 10 ======================================
Log analysis showed nothing abnormal: no error/exception logs, GC lines at only 0-1
per minute (far below the daily average, ruling out JVM memory pressure), and
apache/Tomcat access-log throughput steady and symmetric. Trace analysis also showed
better-than-average latency for all 12 traced components. Notably, both traces and
logs cover only apache01/02, Tomcat01-04, MG01/02, IG01/02 -- the Mysql and Redis
tiers are invisible to both, so a fault there would only be visible in metrics.
```

```
Reviewing what remains unexamined in detail: the `redis-Redis_6379_*` KPI family
(Redis01/Redis02) and the full `Mysql-MySQL_3306_*` family. The one clear
below-threshold in-window signal from the network scan was Redis01's TotalTcpConnNum
dropping from its usual ~606 to 506 for two consecutive samples starting 14:37 (16%
below whole-day P5) -- a loss of ~100 client connections, which would also explain the
simultaneous rr dips at 14:44 on multiple services without any error-rate increase. I
need to inspect Redis-specific KPIs to see whether Redis01/Redis02 experienced a
resource fault (memory, CPU, blocked clients, rejected connections) during the window.
====================================== Step 12 ======================================
Redis01 and Redis02 are eliminated: zero blocked clients, zero rejected connections,
zero fork stalls, linear CPU accumulation, and memory/client counts moving DOWN (a
3-minute reconnection of ~100 clients at 14:37-14:39), which is a downstream symptom
rather than a cause.

The elimination is now complete for the 14:30-15:00 window: high CPU usage and high
JVM CPU load are ruled out (CPUCpuUtil flat at 26-27%, no JVM_JVM_CPULoad flags, only
all-day per-core sampling artifacts); high memory usage and JVM OOM Heap are ruled out
(MEMUsedMemPerc drifts 1-2 points, JVM heap never above ~58%, GC log lines only 0-1
per minute, no OOM entries); network latency and network packet loss are ruled out
(NETInErr/NETOutErr exactly 0 on all 14 components, packet rates steady, no
CLOSE_WAIT/FIN_WAIT accumulation, and trace durations in the window 40-60% BELOW
whole-day means with sub-1% P99 exceedance); high disk space usage is ruled out (all
filesystem series flat or on their normal smooth linear log-growth trend, none
exceeding whole-day P99).

What remains is the single largest threshold breach found anywhere in the window, and
it is a disk-read fault on apache01. Its sdb device forms a consecutive run starting
14:31 (UTC+8): DSKRTps 1.1167 vs whole-day P95 0.0167 (~66x, the #1 relative deviation
among all 1083 detected runs) and DSKRead 118.8 vs whole-day P95 0.1333, with
DSKReadWrite 126 vs P95 10. This is followed inside the same window by a second run on
the sda device at 14:43-14:45 (DSKRead 741.6 and 787.1 vs P95 713.6; DSKRTps 22.2/21.7
vs P95 21.4) and a sustained cluster of elevated CPU I/O-wait at 14:45-14:50 (2.97,
3.67, 3.48, 4.87 vs P95 2.81), then escalates immediately after the window at
15:08-15:10 (DSKRead 4138 = 5.8x P95, DSKRTps 79.5 = 3.7x P95, device 53.6% busy vs
P95 0, CPUWio 49.84 = whole-day maximum, spreading to sdb). apache01's peer apache02
stayed completely idle on all the same disk KPIs, confirming the asymmetry is
component-specific rather than workload-wide.

The customer-visible symptoms fall exactly inside this apache01 I/O-stall interval:
the simultaneous request-rate (rr) dip on ServiceTest6/7/9 at 14:44 with success rate
held at 100% (a stall, not errors), and the 100-client Redis01 reconnection at
14:37-14:39. Per the root cause localization rule, the fault with by far the most
significant deviation from its global threshold (>> 50%) within the failure window is
apache01's disk read activity, and the root cause time is taken from the first data
point of that fault.
```

B MODIFIED TESTBED

Our testbed is built on the OpenRCA dataset (Xu et al., 2024), modified to suit our measurement goals. Table 4 compares the two.

Table 4: The original OpenRCA dataset and our testbed.

	OpenRCA	Our testbed
Queries (Bank / Market / Telecom)	136 / 148 / 51	118 / 92 / 51
Total queries	335	261
Root causes per query	one or two	one
Elements to predict	C1–C7 (varies)	component, reason (C6)

The original queries are synthesized, and their seven categories, C1 to C7, differ in which of the timestamp, the component, the reason of the root cause, or their combinations must be predicted; C6 requires the component and the reason. We make two modifications. First, we keep only queries with a single root cause. Second, we resynthesize every query as a C6 query, dropping the timestamp to meet GLA's constraints. C6 also matches practical needs, as developers care most about which component failed and why, and it is the strictest of the categories without a timestamp. The resynthesis follows the original construction procedure. We verified that the underlying incidents are distinct regardless of the original difficulty level, so the queries remain distinct tasks under C6.

C MEASUREMENT VALIDITY

Table 5 compares the end-to-end accuracy, partial-credited score, and average total tokens per run of the instrumented agents against the uninstrumented ones over the modified testbed.

Table 5: Instrumented agents (GLA) against uninstrumented ones (Base).

	Accuracy (%)		Score		Tokens (M)		
Model	GLA	Base	GLA	Base	GLA	Base	Ratio
OPUS-5	54.4	54.4	0.65	0.66	0.88	0.85	1.04×
GPT-5.6	34.1	34.1	0.47	0.46	0.92	0.97	0.95×
IM	26.8	20.7	0.35	0.30	1.38	1.27	1.09×

D Analysis of Exploration Behaviors

This appendix defines the quantities in Table 1 and reports the truncation sizes behind them. Both analyses use the base runs of each model, pooled across systems, and read two artifacts of every run: the code the executor generates at each step, and the main LLM's log.

Truncation calls. A truncation is any call that keeps only the first or largest / smallest k rows or items of a result: `head`, `tail`, `nlargest`, `nsmallest`, a slice such as `[:k]`, or an explicit top-k argument. Commented-out code is ignored. Truncations used only to preview output are included, so the counts are an upper bound on truncations that remove data from the analysis.

Columns of Table 1. The table covers the runs that GLA attributes to FS0.

- *Small-k*: runs in which at least one step contains a truncation with $k \leq 10$.
- *HitBudget*: runs that exhaust the 25-step budget, after which the main LLM must give a final answer.
- *ErrStep$\geq$3*: runs with at least three steps discarded due to an error. A step is discarded when the format check rejects the main LLM's response (unparseable JSON, a missing field, or an invalid candidate record), or when the executor exhausts its retries for generating valid Python code for the data query.

The last column (HitBudget & ErrStep$\geq$3) counts runs that satisfy both conditions. It approximates the runs that exhaust the budget because of step errors rather than a long trace of valid steps; the threshold of three is a heuristic, not a clean separation of the two causes.

Truncation sizes. Table 6 reports the k passed to every truncation call over all base runs (Fig. 2), not only the FS0 ones, at any k.

Table 6: Truncation sizes in executor code. *Calls*: truncation calls found. The five k columns give each range's share of those calls.

Model	Calls	Median k	k 1–5	k 6–10	k 11–20	k 21–50	$k > 50$	Most common k (calls)
Opus-5	806	30	5%	4%	13%	71%	6%	40 (265), 30 (228), 20 (79)
Gpt-5.6	729	10	41%	24%	13%	16%	6%	10 (154), 5 (115), 1 (104)
IM	1159	10	42%	16%	23%	14%	4%	3 (292), 20 (242), 10 (186)
DS-V4	1527	20	28%	19%	29%	13%	11%	20 (436), 10 (294), 5 (259)

E Details of Prompt Interventions

Listing 4: Prompt changes of PI-1 relative to the original prompt, shown in place: removed text struck through in red and added text in green italics, while . . . marks omitted unchanged text.

```
...
Additional Rules:
1. **DO NOT delegate the fault qualification decision to the Executor. The
Executor retrieves and computes; you decide which anomalies qualify as faults.**
The tests in `1.3. Fault identification` (isolated spike filtering and the `<=
50%` threshold breach exclusion) describe how YOU should reason over the returned
data. They are not filters to be embedded in the retrieval code.
    1.1. Never instruct the Executor to report only the rows that already pass a
fault test (e.g., 'only report faults with at least 2 consecutive points', 'only
show components whose deviation exceeds 50%'). Rows discarded inside the code
never reach you, so a criterion that turns out to be too strict cannot be
revisited without spending another Executor step.
    1.2. Instead, ask the Executor to report every point of every in-scope
'component-KPI' that breaches its own global threshold, each with its time, value,
the threshold it breached, the whole-day median of that series, and its deviation
from the threshold. Then apply the spike filtering and the magnitude test of `1.3.
Fault identification` yourself over the returned table.
    1.3. Aggregation discards evidence just as filtering does. DO NOT ask only for
a count, a maximum, or a top-N ranking when the individual points are what let you
judge consecutiveness and magnitude. Ask for the points and rank them yourself.
    1.4. When your own qualification leaves no plausible root cause candidate,
re-read the data the Executor already returned under a looser criterion (see `1.2.
Anomaly detection`) before issuing a new retrieval. Keeping the raw evidence in
the conversation means that loosening a threshold should cost no extra Executor
step.
```

Listing 5: Prompt changes of PI-2 relative to the original prompt, marked as in Listing 4.

```
...
Additional Rules:
1. During initial preprocess and anomaly discovery, **always inspect all KPIs of
each component to find the root cause.** Do not skip any KPIs. Filtering and
narrowing down the search space should be done after inspecting all KPIs.
```

Listing 6: Prompt changes of PI-3 relative to the original prompt, marked as in Listing 4. Additional Rule 1 is the rule PI-2 adds, which PI-3 carries.

```
...
- A 'fault' is ~~a consecutive sub-series of~~ an anomalous pattern in a specific
component-KPI time series and may consist of one or more data points that are not
necessarily consecutive. Thus, fault identification is the process of identifying
which components experienced faults, on which resources, and at what occurrence time
points.
~~- Filter out isolated noise spikes to locate faults.~~
- Faults where the maximum (or minimum) value in the sub-series only slightly exceeds
(or falls below) the threshold ~~(e.g., threshold breach <= 50% of the extremal)~~, it's
likely (but not necessarily) a false positive caused by random KPI fluctuations~~, and
should be excluded~~.
...
2.1. Metric analysis: Use metrics to calculate whether each KPIs of each component has
~~consecutive~~ anomalies beyond the global threshold is the fastest way to find the
faults. Since there are a large number of traces and logs, metrics analysis should
first be used to narrow down the search space of duration and components.
...
Additional Rules:
1. During initial preprocess and anomaly discovery, **always inspect all KPIs of
each component to find the root cause.** Do not skip any KPIs. Filtering and
narrowing down the search space should be done after inspecting all KPIs.
```

```
2. KPIs whose names contain both "TCP" and "WAIT" (e.g., TCP-FIN-WAIT or
TCP-CLOSE-WAIT) typically represent the number of connections in the corresponding
TCP connection state, rather than the duration of waiting. Sustained or unusual
increases in these KPIs can be evidence of network-related faults, such as network
latency or packet loss. Treat them as relevant but non-specific network signals
and correlate them with traffic, error, trace, and log evidence when
distinguishing the exact fault reason.
```

Listing 7: Prompt changes of PI-4 relative to the original prompt, marked as in Listing 4. Additional Rule 1 is the rule PI-2 adds, which PI-4 carries.

```
...
- If multiple faulty components are identified at **different levels** (e.g., some
being containers and others nodes), and all of them are potential root cause
candidates, while the issue itself describes a **single failure**, the root cause
level should be determined by the fault that ~~shows the most significant deviation from
the threshold (i.e., >> 50%)~~ is most specific to the failure window, i.e. the one
whose series least often reaches that level at other times (see `Additional Rules`
4). However, this method is only applicable to identify the root cause level, not the
root cause component. If there are multiple faulty components at the same level, you
should use traces and logs to identify the root cause component.
...
- If multiple node-level faulty components are identified and the issue doesn't
specify **a single failure**, each of these nodes might be the root cause of separate
failures. Otherwise, the predominant nodes with the most faults is the root cause
component, where a level the series reaches routinely is not a fault and must not
be counted however large its multiple (see `Additional Rules` 4). The node-level
failure do not propagate, and trace only captures communication between all containers
or all services.
...
Additional Rules:
1. During initial preprocess and anomaly discovery, **always inspect all KPIs of
each component to find the root cause.** Do not skip any KPIs. Filtering and
narrowing down the search space should be done after inspecting all KPIs.
2. **A multiple of the threshold detects an anomaly within one series; it does NOT
measure severity across series.** Different KPIs behave differently, so the
multiple a KPI can reach is fixed by its own baseline rather than by the severity
of the fault. NEVER rank faults, components, or candidate reasons by comparing
multiples across different KPIs, and never rule one out because its KPI produced a
smaller multiple than another KPI did. Rank competing breaches by rule 4 instead.
3. **When a series' global threshold is 0, no multiple exists, so compare absolute
values instead.** Judge such a series by its absolute value and by how many
consecutive samples leave its all-day constant; a counter reading zero all day and
then holding non-zero for several samples is a severe fault, not a zero-baseline
artefact. The `<= 50%` exclusion in `1.3. Fault identification` does not apply to
it.
4. **Judge a breach by whether it is specific to the failure, which is a question
about that one series alone.** Count how often the same series reaches the same
level outside the failure window, on this day and on other days. A level it
reaches routinely, especially on a schedule, is normal activity that merely
overlaps the window: exclude it however large its multiple, and do not report it
as a fault. A level reached only inside the window is highly likely to be the root
cause fault. When several breaches compete, decide between them on this.
```