ORCA-BENCH: HOW READY ARE LANGUAGE MODEL AGENTS FOR ONCALL?

Anonymous authors
Paper under double-blind review

ABSTRACT

Large language models can write, patch, and search code, but oncall root cause analysis (RCA) demands something different: reasoning over noisy metrics, logs, traces, and source code, starting from ambiguous user-facing reports, often hours after the incident began. We introduce ORCA-bench, a benchmark that puts general-purpose coding agents in a production-fidelity oncall setting. ORCA-bench pairs 1,079 RCA tasks with six days of metrics, logs, and traces collected from an OpenTelemetry-instrumented microservice system under continuous simulated user load. Agents investigate this recorded history through real observability interfaces—Prometheus, Jaeger, and OpenSearch via Grafana—with full access to application source code. Tasks systematically vary report specificity, time-to-detection, and co-occurring fault scenarios. Ground-truth symptoms are curated and signed off by expert SREs, and our LLM-as-judge is independently re-scored by humans (Cohen's $\kappa_w = 0.91$). Across five frontier agents, the best RCA Accuracy is 25.3% on Medium-difficulty tasks (the realistic-input setting) and 10.0% on Hard—a gap that remains even with Claude Fable 5. The weakest model hallucinates an implausible root cause in 40% of incident reports, and removing source-code access reduces RCA accuracy and increases the hallucination rate for every evaluated model. Crucially, these are performances on a curated $50\,\text{GB}$ / six-day testbed with tasks investigated in isolation on a system whose code and instrumentation are public. Since real production systems are order of magnitudes larger, more dynamic, and more idiosyncratic, the gap we report in our controlled setting underscores the engineering work still needed before agents can be entrusted with production reliability.

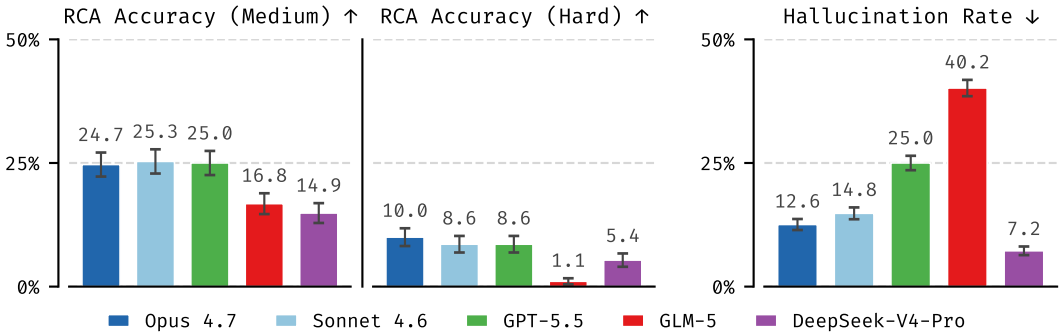


Figure 1: RCA accuracy measures whether agents identified *all* plausible root causes, and hallucination rate measures whether agents named a root cause matching *none* of the plausible root causes. We show the realistic Medium and Hard difficulties here and report Easy in Fig. 5 along with Claude Fable 5 results on a subset of tasks in Sec. 5. Error bars display ± 1 standard error across $N = 316$, $N = 280$, and $N = 884$ tasks for left, middle, and right plots, respectively.

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.

1 INTRODUCTION

Large language models (LLMs) now write, refactor, review, and debug code at scale—but they do not yet own the reliability of the systems they help build. When a production system breaks, an oncall site reliability engineer (SRE) is handed a vague natural-language complaint such as "users are having issues on the site" or "checkout is broken," often hours after the incident began. In a short period of time and under stress, they must identify which feature, deploy, or component caused the symptom—a task known as *root cause analysis* (RCA). RCA is the diagnostic prerequisite for mitigation in modern microservice deployments, and it is qualitatively unlike the bug-fix-on-a-static-repo setting on which software-engineering benchmarks (see, e.g., Deng et al., 2026; Jimenez et al., 2024) have driven recent agent progress: the input is an ambiguous report rather than a failing test; the evidence is a running microservice mesh rather than a frozen codebase; and the success criterion is a defensible diagnosis rather than a passing test. Existing benchmarks (see, e.g., Fang et al., 2026; Clark et al., 2026; Wang et al., 2026; Jha et al., 2025; Xu et al., 2025; Shetty et al., 2024) evaluate RCA by injecting a synthetic fault into a microservice cluster and asking an agent to name the root cause, but each strips out at least one of the three artifacts an oncall engineer reaches for—a telemetry interface over metrics, logs, and traces; the raw telemetry generated by services under live load; and the source code of those services—and none systematically vary the specificity of the user report or the time-to-detection (TTD) between incident onset and investigation start (Tab. 1). Only OpenRCA (Xu et al., 2025), Cloud-OpsBench (Wang et al., 2026), and OpenRCA 2.0 (Fang et al., 2026) human-verify their root-cause labels. A benchmark that takes production-style RCA seriously must therefore expose the full evidence stack, vary report specificity and TTD jointly, and validate both ground truth and judge with human review.

Contributions. We introduce ORCA-bench, a benchmark that puts general-purpose coding agents in production-fidelity oncall conditions and measures whether they can localize the root cause of a user-facing incident from telemetry, source code, and an ambiguous natural-language report.

1. **The full oncall evidence stack.** We collect six days of telemetry from the OpenTelemetry Astronomy Shop under continuous simulated user load and expose the recorded metrics, logs, and traces through Prometheus, OpenSearch, and Jaeger via Grafana, alongside full application source code—making ORCA-bench the first SRE benchmark to offer all three artifacts an oncall engineer reaches for in a single environment (Tab. 1).
2. **A context ladder where scores collapse.** 1,079 RCA tasks generated by toggling real feature flags rather than a synthetic fault library, jointly varying user-facing issue specificity (issue specificity) along an Easy/Medium/Hard ladder, TTD across 15 min–24 hr, and co-occurring fault structure across five scenario types—isolated, independent, conflicting, cascading, and sequential—designed with expert SREs (Sec. 3).
3. **Symptom-level ground truth, hand-validated.** Per-fault rubrics that enumerate the expected symptoms in metrics, logs, and traces, validated against telemetry by expert SREs; tasks are scored against the *set* of plausible root causes rather than a single label, and a 40-task ORCA-bench Verified subset has every ground-truth label and every per-model score hand-confirmed (Sec. 3 and 4).
4. **A judge humans re-scored from scratch.** A GPT-5.4-based LLM-as-judge whose per-task scores were independently re-graded by hand on ORCA-bench Verified across all five evaluated agents, attaining Cohen's $\kappa_w = 0.91$ against the mean human re-score (Sec. 4).
5. **Frontier agents are not yet ready for oncall.** Across five frontier coding agents, Opus 4.7, Sonnet 4.6, GPT-5.5, GLM-5, and DeepSeek-V4-Pro, the best RCA Accuracy is 25.3% on Medium-difficulty tasks (the realistic-input setting) and 10.0% on Hard (Fig. 1); agents hallucinate an implausible root cause in 7% (DeepSeek-V4-Pro) to 40% (GLM-5) of incident reports; and removing source-code access reduces RCA accuracy and increases the hallucination rate for every evaluated model (Sec. 5).

Implications for production readiness. ORCA-bench measures performance on a 50 GB / six-day testbed, with tasks investigated in isolation on a system whose code and instrumentation are public. The diagnostic failures observed in this controlled setting underscore the engineering work still needed before agents can be entrusted with production reliability. Generalization to larger, evolving, and private systems remains to be evaluated.

Table 1: **ORCA-bench vs. prior RCA benchmarks.** ✓ indicates the feature is supported, ✗ indicates it is not supported, and ∼ indicates partial or limited support. For scoring, ✓ denotes human-verified scoring and – denotes programmatic scoring.

Benchmark	Task complexity		Environment realness		Human verification		
	Issue specificity	TTD & offset	Telemetry interface	Source code	Symptoms	Root causes	Scoring
AIOpsLab (Shetty et al., 2024)	✗	✗	∼	✗	✗	∼	–
OpenRCA (Xu et al., 2025)	✗	✗	∼	✗	✗	✓	–
ITBench (Jha et al., 2025)	✗	✗	✓	✗	∼	∼	–
Cloud-OpsBench (Wang et al., 2026)	✗	✗	∼	✓	✓	✓	–
SREGym (Clark et al., 2026)	✗	✗	✓	✗	✗	∼	✓
OpenRCA 2.0 (Fang et al., 2026)	✗	✗	∼	✗	✓	✓	–
ORCA-bench (Ours)	✓	✓	✓	✓	✓	✓	✓

Paper organization. Sec. 2 compares ORCA-bench to prior SRE and SWE benchmarks. Sec. 3 describes the environment, task generation, and ground-truth curation. Sec. 4 defines the scoring rubric and validates the LLM scoring against human scoring. Sec. 5 reports the performance, ablation, and failure-mode analyses.

2 Related Work

Root cause analysis sits between two existing lines of work. LLM-free RCA based on causal inference and graph analysis, such as those evaluated by Pham et al. (2025), reason over structured numerical signals—metric time series, dependency graphs—but cannot ingest the natural-language complaints that initiate real investigations. Software-engineering benchmarks bring agentic LLM reasoning, but on a task that looks superficially similar to RCA while differing in every dimension that matters.

Why SWE benchmarks do not transfer. Software-engineering benchmarks (see, e.g., Deng et al., 2026; Yang et al., 2025; Bogomolov et al., 2024; Jimenez et al., 2024) evaluate LLM agents on a frozen repository given a precise task specification—typically a bug report or a failing test—and judge correctness by re-running the test suite. Production RCA replaces every one of those ingredients. The system under investigation is a live distributed deployment with dependency chains visible only through traces, time-evolving behavior visible only through metrics, and high-volume semi-structured text visible only through logs. The task input is a vague natural-language complaint that may not point at any specific service. And the success criterion is a defensible diagnosis judged against human-curated ground truth, not a binary test verdict. These differences motivate benchmarks built around ambiguous reports and interactive investigation of recorded runtime telemetry alongside application source code.

Toward this goal, SREGym (Clark et al., 2026) studies diagnosis of noisy and correlated failures in a live fault injection environment, similar to the works of Jha et al. (2025); Shetty et al. (2024). The recent Cloud-OpsBench (Wang et al., 2026) augments these benchmarks with source-code access and reproducible snapshots, meanwhile OpenRCA (Xu et al., 2025) provides historical telemetry. We summarize the differences between these benchmarks and ours in Tab. 1.

Temporal setting and issue specificity. While the existing RCA benchmarks provide different temporal contexts for diagnosis (e.g., investigations following fault injection (Clark et al., 2026; Jha et al., 2025; Shetty et al., 2024), short telemetry snapshots (Fang et al., 2026; Wang et al., 2026), and historical telemetry (Xu et al., 2025)), none study the uncertainty introduced by the user report and the timing of the investigation. In production, an investigation may begin hours after an issue starts, and the report may describe its symptoms vaguely or its onset inaccurately. ORCA-bench makes these conditions explicit task parameters: the delay between incident onset and investigation, the offset between incident onset and reported time, the form of the reported time, and the specificity of the user-facing issue. These parameters support controlled comparisons of RCA performance under different reporting conditions.

Telemetry interface and source-code access. ORCA-bench combines historical investigation with access to observability services and application source code. OpenRCA (Xu et al., 2025) supplies

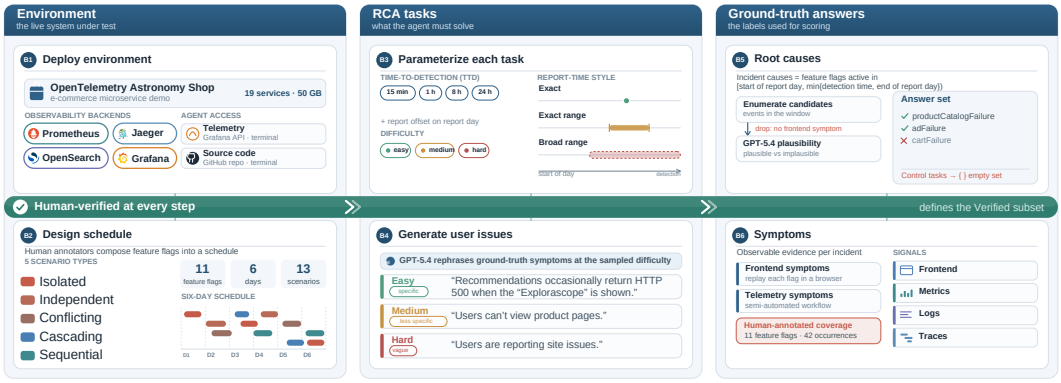


Figure 2: **Construction pipeline.** We construct ORCA-bench in six stages. and manually verify a subset of 40 tasks to obtain ORCA-bench Verified. See Sec. 3 for details.

historical telemetry as CSV files that agents can analyze using Python. However, this interface is far removed from how telemetry is queried in production, and it does not provide source-code access. ORCA-bench instead exposes metrics, logs, and traces through Grafana-backed Prometheus, OpenSearch, and Jaeger interfaces, requiring agents to formulate queries, select relevant time windows, and retrieve evidence across telemetry sources. Source-code access additionally allows agents to connect observed symptoms to implementation details. Diagnostic interfaces and source-code access are not individually unique to ORCA-bench: ITBench (Jha et al., 2025) and SREGym (Clark et al., 2026) provide production-grade telemetry interfaces, and Cloud-OpsBench (Wang et al., 2026) provides source-code access, but none combines both. Our contribution is to bring them together with systematically varied investigation delays and uncertain user reports, and to evaluate diagnoses against the multiple causes consistent with those reports.

Level of human verification. In production systems, some issues have user-facing impact, while others are silent failures. Clark et al. (2026); Jha et al. (2025); Shetty et al. (2024) inject faults live, meaning the underlying issues are known in advance, but they do not distinguish between faults with user-facing impact and those without. At a lower level, RCA requires reasoning about symptoms. Jha et al. (2025) curate ground-truth symptoms only in metrics, omitting those in logs and traces. Cloud-OpsBench (Wang et al., 2026) and OpenRCA 2.0 (Fang et al., 2026) annotate the symptoms or propagation paths of each injected fault, but these annotations are not tied to a user-facing report and do not distinguish faults with user-facing impact from silent ones. Xu et al. (2025) manually verify plausible root causes within an observation window, but do not provide ground-truth symptoms. Finally, among prior benchmarks, only SREGym (Clark et al., 2026) validates its automated RCA evaluator against human judgments.

3 ORCA-BENCH CONSTRUCTION

Below are the six stages of constructing ORCA-bench; the remainder of the section elaborates each stage.

B1. Deploy environment: run the OpenTelemetry Astronomy Shop with Prometheus, OpenSearch, and Jaeger; expose telemetry to agents via Grafana and source code via a terminal.

B2. Design schedule: with expert SREs, compose Astronomy Shop feature flags into a six-day schedule of 13 scenarios spanning five scenario types (Fig. 2).

B3. Parameterize each task: sample a report *offset* and a *time-to-detection* (TTD) from $\{15\,\text{min}, 1\,\text{h}, 8\,\text{h}, 24\,\text{h}\}$, a report-time style (*exact*, *exact range*, *broad range*), and a question issue specificity (*Easy*, *Medium*, *Hard*).

B4. Generate user issues: use GPT-5.4 to rephrase the ground-truth symptoms at the sampled issue specificity level.

B5. Generate ground-truth root causes: enumerate candidate events active in $[\text{start of report day}, \min\{\text{detection time}, \text{end of report day}\})$, drop those with no frontend

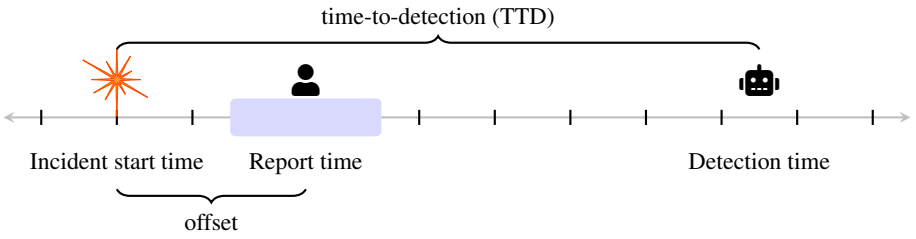


Figure 3: **Task definition.** The SRE agent is provided a user-facing issue at the report time (an *offset* after the incident start time) and begins investigation at the detection time (a *time-to-detection* after the incident start time).

symptoms, and use GPT-5.4 to mark each remaining candidate plausible/implausible against the user issue; control tasks have the empty set.

B6. Generate ground-truth symptoms: collect frontend symptoms by replaying each flag in a browser; collect telemetry symptoms via a semi-automated workflow validated by two expert SREs across 11 flags / 42 occurrences.

We have two authors verify LLM-generated root causes for 40 tasks (10 Easy, 15 Medium, and 15 Hard tasks sampled randomly and stratified by issue specificity, scenario, and offset/TTD parameterization), which we call ORCA-bench Verified.

B1. Environment. Our starting point is the OpenTelemetry Astronomy Shop, a distributed system comprising 19 microservices and spanning 13 different programming languages. OpenTelemetry standardizes the collection, transmission, and semantic conventions of telemetry data and is now supported by every major observability backend. Following the default Astronomy Shop configuration, we use Prometheus for metrics, OpenSearch for logs, and Jaeger for traces. During evaluation, SRE agents access the source code through terminal commands. As the telemetry data over the six-day period totals more than 50 GB, it is infeasible to provide direct access to the raw data for LLM agents; we therefore expose the Prometheus, OpenSearch, and Jaeger datasources through the Grafana API.

B2. Schedule. To induce events, we fire feature flags (built-in to the Astronomy Shop) over a six-day period; Tab. A.2 shows two such flags along with their symptoms. With the help of expert SREs, we design five scenario types, shown in Fig. 2, and construct a six-day schedule that contains 13 scenarios and satisfies the constraints in App. B.1 on the number of co-occurring issues, the overlap in symptoms, and the duration of incidents. The schedule, which we include in full in App. B.2, is structured around a "FAFO Friday" narrative: weekday buildup on Days 1–5, and a chaos peak on Day 6 (Friday) where six feature flags are concurrently active.

ORCA-bench comprises 1,079 tasks, each constructed from a common task instruction template and paired with ground-truth root causes and symptoms. We instantiate the template, shown in App. C, with a *report time*, *detection time*, and *user-facing issue*. For a given task, an SRE agent can only query recorded telemetry up to the task's detection time.

B3. Report and detection times. The report time is the time at which a (hypothetical) user first reports an issue on the Astronomy Shop website. We assume the issue remains open, and may continue to be reported, until the SRE agent begins investigating at the detection time. As depicted in Fig. 3, we parameterize the report time and detection time by choosing an *offset* and *time-to-detection (TTD)*, respectively, from the *incident start time*. Our task instruction, shown in App. C, makes clear to the SRE agent that the report time need not coincide with when the incident began, i.e., the report time can be different from the incident start time. We vary TTD over $\{15\,\text{min}, 1\,\text{h}, 8\,\text{h}, 24\,\text{h}\}$ and consider three styles of report time: *exact time* (e.g., "at 2:30pm"), *exact range* (e.g., "between 2pm and 3pm"), and *broad range* (e.g., "sometime in the afternoon", "sometime yesterday"). To reflect the way users specify report times in the real world, we reserve longer TTDs for the broader report time styles. Separately, we define **control** tasks in the quiet periods between days to test whether the agent can correctly identify that no incident is occurring; 195 of the 1079 total tasks are control tasks.

B4. User issues. The *issue specificity* controls the amount of context provided to the SRE agent. Intuitively, descriptions with less context are more difficult because they provide fewer leads for the SRE agent to follow. "Users are reporting site issues" is a user-facing issue with the **Hard** issue specificity; adding context about the affected component yields **Medium** issue specificity; **Easy** user-facing issues contain context about the specific symptom or error message, reflecting the real-world setting of having precisely instrumented alerts or triaging. To obtain these Easy- and Medium-issue specificity questions, we prompt GPT-5.4 to rephrase the human-curated ground-truth symptoms (see prompt in App. D).

B5. Root causes. A single incident task may admit multiple plausible root causes. This occurs when several feature flags are active in the same window, or the reduced context in the prompt means multiple root causes could plausibly explain the issue. Hence, we treat the answer for each task as the *set* of plausible root causes. For each task we first enumerate every event occurring on the report's day and before the detection time, then drop events without frontend symptoms (a user-facing issue cannot be explained by an incident with no user-visible effect). We then instruct GPT-5.4 with the prompt in App. E, which contains the user-facing issue together with each surviving candidate's rubric (description, mechanism, and frontend symptoms), to emit one verdict per candidate marking whether its user-visible effect could plausibly explain the user-facing issue. Plausible candidates are stored as the task's ground-truth answer set; for control tasks the answer is the empty set. Since the issue remains open until detection, the answer set includes every symptom-consistent incident up to the detection time, even one that begins after the report time.

B6. Symptoms. To curate the ground-truth frontend symptoms, we run the Astronomy Shop backend services using Docker, open the frontend in a browser, fire feature flags one at a time, and observe any changes on the frontend. If a frontend change is observed, we use that as our ground-truth symptom, otherwise we deem the feature flag to have no observable frontend symptom. To curate the ground-truth metrics, logs, and traces symptoms, we develop a semi-automated workflow and detail it in App. F. With the help of two expert SREs, we validated the ground-truth symptoms produced by this workflow (App. A) on the two feature flags in Tab. A.2 before scaling the procedure to 11 feature flags across all 42 occurrences in our schedule.

4 EVALUATION

The ground-truth root causes and symptoms are used as rubrics for scoring. We employ an LLM as a judge (Zheng et al., 2023; Zhou et al., 2025) to compute the evaluation metrics, including *RCA accuracy*, *RCA depth*, and *hallucination rate*. To ensure alignment, we validate LLM judge scores against scores from human annotators on the ORCA-bench Verified tasks.

E1. Scoring a single root cause. We assign a **score** on the following 0–3 scale (full prompt in App. G.1.1):

⓪ AI findings are misaligned with this rubric's mechanism and completely miss this root cause.

① AI verifies incident description: the report confirms symptoms described in the task prompt but does not investigate further toward this root cause.

② AI makes correct progress beyond symptoms: the report identifies some of this rubric's ground-truth metrics, logs, or traces and moves toward this root cause as described by the rubric, but does not fully identify it.

③ AI nails this root cause: the report correctly identifies it, this rubric's mechanism, and the supporting signals.

For more detailed analysis, we define secondary metrics in App. J. Scoring is based on content correctness only; formatting, prose style, and structural polish are explicitly excluded. Materially incorrect or misleading claims count against the per-rubric score. Empty incident RCA reports score 0 on every rubric.

E2. Computing task-level metrics. From the scores we report three core metrics per task:

(M1) **RCA accuracy**: 1 if the agent named *every* listed plausible feature flag as a root cause; 0 otherwise.

(M2) **RCA depth**: the mean of per-rubric scores. Reported as a % of the maximum (3).

(M3) **Hallucination rate**: 1 if the agent named a root cause that matches *none* of the listed plausible feature flags; 0 otherwise.

For a control task, the RCA accuracy is 1 if the agent correctly identifies the *absence* of an issue if it does not place any incident strictly inside the verified *quiet window* around the queried time detection time; an empty report is a trivial pass, and a non-empty report is judged by an LLM using the prompt in App. G.1.2. We also report three finer-grained metrics: **recall**, the fraction of a task's ground-truth root causes found in the report; **precision**, the fraction of root causes attributed in the report that are ground truth; and **hallucinated root causes**, the number of attributed root causes that are not ground truth. We provide formal definitions for all metrics in App. H.

We validated the scoring guidelines by having two authors independently re-score the RCA reports for five models (Claude Opus 4.7, Claude Sonnet 4.6, GPT-5.5, DeepSeek-V4-Pro, GLM-5) on the 32 incident and 8 control tasks in ORCA-bench Verified.[1] We then use GPT-5.4 to score the incident RCA reports using the same guidelines. Tab. 2 shows that the judge achieves strong agreement with the mean human score across all difficulty levels. We display per-task alignment for each annotator in Fig. G.1 and G.2; the two annotators agree with each other at $\rho = 0.85$, $\kappa_w = 0.83$.

Table 2: RCA depth agreement between the LLM judge (GPT-5.4) and the mean of two human annotators' scores on ORCA-bench Verified. ρ: Spearman rank correlation; κ_w: quadratic-weighted Cohen's κ.

Difficulty	N	ρ	κ_w
All	191	0.93	0.91
Easy (10 tasks)	47	0.92	0.91
Medium (15 tasks)	72	0.91	0.91
Hard (15 tasks)	72	0.89	0.89

5 RESULTS

We investigate to what extent general-purpose, language model agents can perform root cause analysis. Specifically, we use the Terminus-2 agent harness of Merrill et al. (2026), which provides the LLM access to only an interactive `tmux` session and performs context compaction when needed to prevent context overflow. We provide supplementary experiment details in App. I.

Finding #1: agents are far from being intelligent SREs. On RCA depth, which assigns partial credit for making progress toward the root causes, GPT-5.5 achieves the highest score of 48.8% across all 884 incident tasks; Claude Opus 4.7 and Claude Sonnet 4.6 achieve similar RCA depth, with DeepSeek-V4-Pro and GLM-5 trailing behind. On the stricter RCA accuracy, which requires naming all root causes correctly, the best-performing model (Claude Sonnet 4.6) achieves 30.9%. On the 32 incident tasks in ORCA-bench Verified, Claude Fable 5 achieves the highest RCA depth (RCA accuracy) of 58.2 ± 5.8% (40.6 ± 8.8%), compared to 49.2 ± 5.6% (21.9 ± 7.4%) for GPT-5.5 and 47.6 ± 5.7% (25.0 ± 7.8%) for Claude Opus 4.7 (Fig. L.1). The low RCA accuracy also persists under a different harness: on the 32 incident tasks in ORCA-bench Verified, Claude Opus 4.7 achieves 25.0% under both Terminus-2 and OpenSRE (Tracer-Cloud, 2026). We detail this comparison and its configuration differences in App. L.1. We provide the complete ORCA-bench results in Tab. J.1.

In our manual scoring of the incident RCA reports on ORCA-bench Verified, we find that (1) agents tend to lock onto louder symptoms, missing certain root causes, and (2) struggle to consistently quantify the severity increase and attribute it to distinct events. In Fig. 4, for example, GLM-5 gets distracted by a persistent background issue and fails to identify the active root cause of the user-facing isuse. Claude Opus 4.7, on the other hand, correctly identifies the `productCatalogFailure` feature flag as the root cause, tracing the chain from failed cart checkouts through `ProductCatalogService` to the enabled flag.

Finding #2: reducing input context drops RCA accuracy by 19–50 percentage points across models. Varying the issue-specificity from Easy to Hard increases the number of plausible root causes from 2.00 to 4.41. However, the increased number of root causes does not alone account for the drop in accuracy. For example, fixing the number of answers and varying issue specificity drops both average RCA accuracy and depth pooled across all models (Fig. L.4). We also find that RCA

[1] DeepSeek-V4-Pro exceeded the agent timeout for 9 tasks, so we exclude those from the validation.

Task (Medium, TTD = 15 min; ground truth: `productCatalogFailure`)

Instruction: *"Shoppers cannot use their cart. This was reported at 1:14 PM today ET."* (Agent's now: Apr 24, 2026, 1:19 PM ET)

GLM-5 — Human 0/3, GPT-5.4 judge 0/3

Summary. Between Apr 22, 10:13 UTC and Apr 24, 17:25+ UTC, browser-based simulated shoppers could not access cart functionality due to a **Playwright browser crash** in the load-generator. Over 94,000 browser-test errors occurred during the 55+ hour window. The cart service itself was operational.

Root cause. The Playwright browser integration in the load-generator lacks resilience to browser-process termination. When the browser crashes, the load-generator continues using the dead instance without recovery, causing indefinite test failures.

Claude Opus 4.7 — Human 3/3, GPT-5.4 judge 3/3

Summary. Between ∼17:06–17:25 UTC, shoppers attempting to view product `OLJCESPC7Z` or complete checkout received HTTP 500 errors because the `productCatalogFailure` feature flag was enabled. `CheckoutService/PlaceOrder` failed with *"failed to get product #OLJCESPC7Z"*.

Root cause. The `productCatalogFailure` fault-injection flag was switched on in flagd at ∼17:06 UTC, causing `ProductCatalogService/GetProduct` to return `Internal` gRPC errors for product `OLJCESPC7Z`, propagating as failed checkouts for any shopper whose cart contained that item.

Figure 4: **Excerpts from a low-scoring and high-scoring incident RCA report on the same task.** See App. K for the full, unabridged reports.

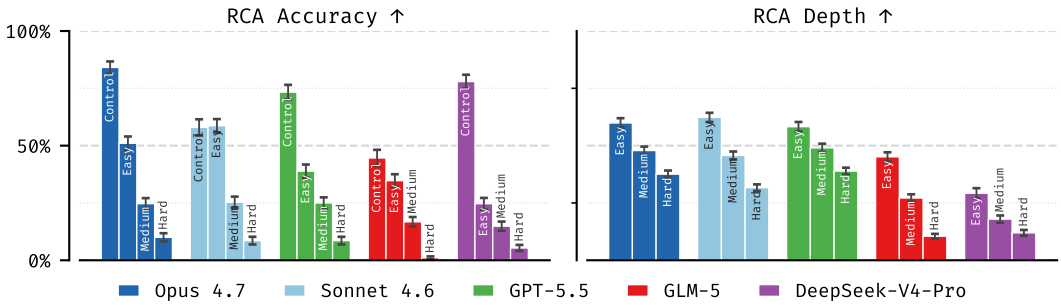


Figure 5: **RCA quality by issue specificity.** We report mean ± 1 standard error across the 195 control, 288 Easy-incident, 316 Medium-incident, 280 Hard-incident tasks.

recall, which measures how much of each cause can the agent find, drops as we go from Easy to Hard tasks across all models (Fig. L.5). In contrast, precision generally increases with issue specificity (Fig. L.6a): from Easy to Hard, it rises from 72.1% to 83.3% for Claude Opus 4.7, from 75.7% to 78.4% for Claude Sonnet 4.6, and from 71.4% to 79.5% for GPT-5.5. As the number of ground-truth root causes grows, any root cause the agent attributes is more likely to be among them. The exceptions are GLM-5 (69.0% to 30.5%) and DeepSeek-V4-Pro (38.0% to 26.0%), which produce more empty reports on harder tasks; since we assign empty reports a precision of zero, this lowers their precision on those tasks.

In sum, the best-performing model by RCA accuracy (Claude Opus 4.7) achieves only 10.0%, and no model exceeds 38.9% RCA depth (GPT-5.5). Easy tasks still leave ample room for improvement; the best-performing model (Claude Sonnet 4.6) only manages 58.7% RCA accuracy on these tasks. Our scenario on Day 6 of the incident schedule in App. B.2 has six co-occuring events: three models each identify exactly one root cause—all different ones—and miss the other five, leading to a partial RCA score of $0.5/3 = 0.167$; the fourth model (GPT-5.5) finds four of six root causes but still misses two; and the fifth model (DeepSeek-V4-Pro) finds none. On ORCA-bench Verified, we find that the gains in RCA depth and accuracy from Claude Fable 5 over Claude Opus 4.7 come primarily from Medium tasks, not Easy tasks as one might expect (see Fig. L.1).

Finding #3: removing source code drops RCA accuracy by 9–16 percentage points across models. We run all models on two versions of ORCA-bench, one with both telemetry and code access (the standard version described in Sec. 3) and one with telemetry access only (code access removed). Fig. 6 shows that without source code access, RCA accuracy drops and hallucination spikes for all models. Incident-time accuracy is relatively robust (Claude Opus 4.7: 82.6% → 79.9%), consistent with the fact that temporal information lives in telemetry signals (timestamps, metric series), not in source code. Despite the large effect of source code access on RCA quality, Fig. J.1 (App. J.4) shows that Claude Opus 4.7 and GLM-5 spend only 16% and 20% of commands on reading source

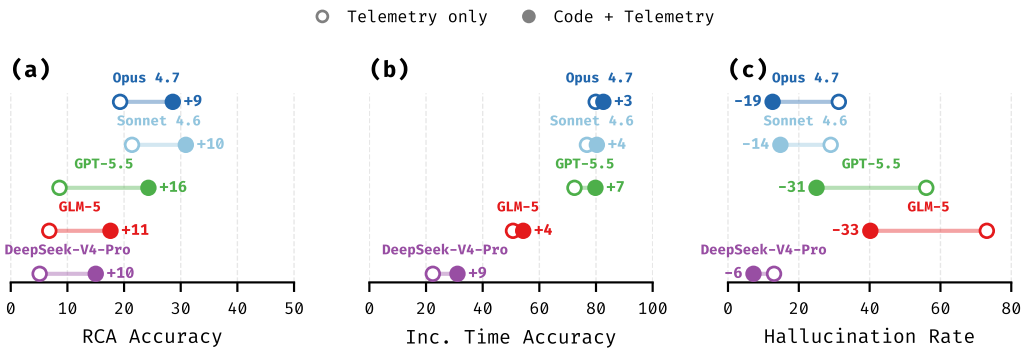


Figure 6: **RCA quality with and without code access.** We report mean ± 1 standard error across all $N = 884$ incident tasks. See App. J for the definition of Incident Time Accuracy.

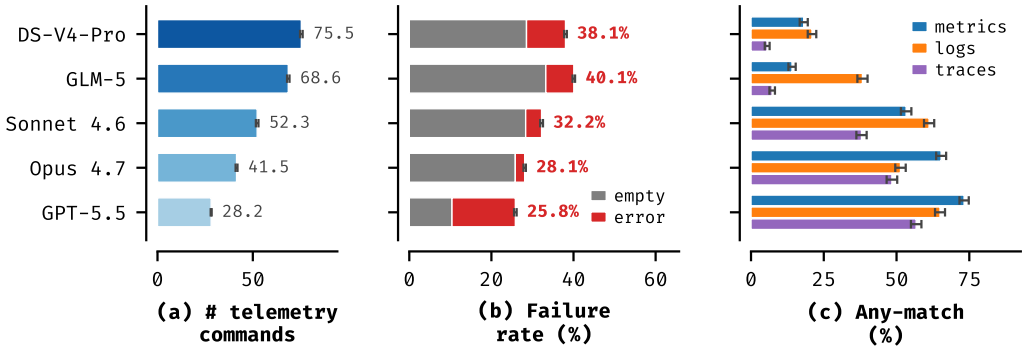


Figure 7: **Telemetry retrieval efficiency and accuracy.** We report mean ± 1 standard error across all $N = 1079$ tasks, all telemetry commands, and all tasks with ground-truth metrics/logs/traces for (a), (b), and (c), respectively. *Empty* means the call ran successfully but returned no data and *error* means the call failed. We define the Any-match metric in App. J.

code, respectively, compared to 72% and 70% on telemetry-related commands; the remaining three models are shown in Fig. J.2. Agents use source code to either explore the system at the start of their trajectories or to confirm hypotheses after querying telemetry.

Finding #4: 26–40% of telemetry calls either error out or return empty. We classify every telemetry command output as either a success, empty, or error. In addition to being the most efficient in terms of telemetry commands (see Fig. 7(a)), GPT-5.5 has the highest success rate (see Fig. 7(b)). Fig. 7(b) shows that the high telemetry failure rate for the other models is due to the fact that the commands they invoke often yield empty results. Using the ground-truth symptoms collected in ORCA-bench, we compute the percentage of trials in which *any* ground-truth metric, log, or trace was reported in the SRE agent's incident RCA report. Again, GPT-5.5 achieves the highest accuracy across all three telemetry modalities.

6 Conclusion

We introduced ORCA-bench, which evaluates language-model agents on historical incident diagnosis using telemetry collected from an OpenTelemetry-instrumented application, queried through real observability interfaces and supplemented with source-code access—all under production-style ambiguity in the user report. The headline finding is unambiguous: across five frontier agents, the best RCA Accuracy is 25.3% on Medium-difficulty tasks and 10.0% on Hard (Fig. 1), with hallucination rates from 7% to 40% and lower RCA accuracy and higher hallucination rates across all five models when source-code access is removed. On the most realistic axis we can construct in a controlled benchmark, today's agents are not ready for oncall.

AI USE STATEMENT

In this work, we used generative AI tools for (i) writing assistance—polishing prose, suggesting rephrasings, and checking grammar in portions of the manuscript; (ii) scaffolding portions of the data-processing and plotting code; and (iii) as a component of the benchmark itself: frontier LLMs draft candidate questions and ground-truth artifacts from raw telemetry (App. D and E) and serve as judges that grade agent root cause analysis reports (App. G.1.1 and G.1.2). We have reviewed all AI-assisted work. All LLM-generated text was reviewed and edited by the authors; all LLM-assisted code was reviewed and tested by the authors prior to use in the reported experiments; and LLM-generated ground-truth artifacts were validated by expert SREs and hand-verified on the 40-task ORCA-bench Verified subset, on which the LLM judge was independently re-scored by human annotators (Sec. 3 and 4). The judge prompts and rubrics are released alongside the dataset to enable independent auditing. We take responsibility for the final content of this work, including text, claims, and artifacts produced with the aid of generative AI.

REPRODUCIBILITY STATEMENT

Sec. 3 describes the incident-injection pipeline and task-construction procedure, and App. B.2, D, and E provide the full incident catalog and per-task ground-truth construction. The LLM-judge prompts and grading rubric are given in App. G.1, G.1.1, and G.1.2. App. I specifies the environment version, agent setup, evaluated models, decoding configuration, and judging protocol so that aggregate metrics reported in Sec. 5 can be reproduced within statistical error. We include the code used to reproduce our dataset construction and evaluation as well as 755 Harbor tasks in the supplementary material. Each shared Harbor task contains the environment, the task instruction, the ground-truth answers, and the LLM-judge verifier implementation. We reserve 324 of the 1,079 tasks as a private held-out set to monitor data contamination.

REFERENCES

Egor Bogomolov, Aleksandra Eliseeva, Timur Galimzyanov, Evgeniy Glukhov, Anton Shapkin, Maria Tigina, Yaroslav Golubev, Alexander Kovrigin, Arie Van Deursen, Maliheh Izadi, et al. Long code arena: a set of benchmarks for long-context code models. *arXiv preprint arXiv:2406.11612*, 2024. (Cited on page 3.)

Jackson Clark, Yiming Su, Saad Mohammad Rafid Pial, Yifang Tian, Lily Gniedziejko, Hans-Arno Jacobsen, Yinfang Chen, and Tianyin Xu. Sregym: A live benchmark for ai sre agents with high-fidelity failure scenarios. *arXiv preprint arXiv:2605.07161*, 2026. (Cited on pages 2, 3, and 4.)

Xiang Deng, Jeff Da, Edwin Pan, Yannis Yiming He, Charles Ide, Kanak Garg, Niklas Lauffer, Andrew Park, Nitin Pasari, Chetan Rane, et al. SWE-Bench Pro: Can AI agents solve long-horizon software engineering tasks? In *International Conference on Machine Learning*, 2026. (Cited on pages 2 and 3.)

Aoyang Fang, Yifan Yang, Jin'ao Shang, Qisheng Lu, Junjielong Xu, Rui Wang, Songhan Zhang, Yuzhong Zhang, Boxi Yu, and Pinjia He. OpenRCA 2.0: From outcome labels to causal process supervision. *arXiv preprint arXiv:2606.27154*, 2026. (Cited on pages 2, 3, and 4.)

Saurabh Jha, Rohan R Arora, Yuji Watanabe, Takumi Yanagawa, Yinfang Chen, Jackson Clark, Bhavya Bhavya, Mudit Verma, Harshit Kumar, Hirokuni Kitahara, et al. Itbench: Evaluating ai agents across diverse real-world it automation tasks. In *International Conference on Machine Learning*, pages 27134–27197. PMLR, 2025. (Cited on pages 2, 3, and 4.)

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues? In *International Conference on Learning Representations*, volume 2024, pages 54107–54157, 2024. (Cited on pages 2 and 3.)

Mike A. Merrill, Alexander G. Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Yeon Shin, Thomas Walshe, E. Kelly Buchanan, Junhong Shen, Guanghao Ye, Haowei

Lin, Jason Poulos, Maoyu Wang, Marianna Nezhurina, Jenia Jitsev, Di Lu, Orfeas Menis Mastromichalakis, Zhiwei Xu, Zizhao Chen, Yue Liu, Robert Zhang, Leon Liangyu Chen, Anurag Kashyap, Jan-Lucas Uslu, Jeffrey Li, Jianbo Wu, Minghao Yan, Song Bian, Vedang Sharma, Ke Sun, Steven Dillmann, Akshay Anand, Andrew Lanpouthakoun, Bardia Koopah, Changran Hu, Etash Guha, Gabriel H. S. Dreiman, Jiacheng Zhu, Karl Krauth, Li Zhong, Niklas Muennighoff, Robert Amanfu, Shangyin Tan, Shreyas Pimpalgaonkar, Tushar Aggarwal, Xiangning Lin, Xin Lan, Xuandong Zhao, Yiqing Liang, Yuanli Wang, Zilong Wang, Changzhi Zhou, David Heineman, Hange Liu, Harsh Trivedi, John Yang, Junhong Lin, Manish Shetty, Michael Yang, Nabil Omi, Negin Raoof, Shanda Li, Terry Yue Zhuo, Wuwei Lin, Yiwei Dai, Yuxin Wang, Wenhao Chai, Shang Zhou, Dariush Wahdany, Ziyu She, Jiaming Hu, Zhikang Dong, Yuxuan Zhu, Sasha Cui, Ahson Saiyed, Arinbjörn Kolbeinsson, Jesse Hu, Christopher Michael Rytting, Ryan Marten, Yixin Wang, Alex Dimakis, Andy Konwinski, and Ludwig Schmidt. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces, 2026. URL https://arxiv.org/abs/2601.11868. (Cited on pages 7 and 25.)

Luan Pham, Hongyu Zhang, Huong Ha, Flora Salim, and Xiuzhen Zhang. Rcaeval: A benchmark for root cause analysis of microservice systems with telemetry data. In *Companion Proceedings of the ACM on Web Conference 2025*, pages 777–780, 2025. (Cited on page 3.)

Manish Shetty, Yinfang Chen, Gagan Somashekar, Minghua Ma, Yogesh Simmhan, Xuchao Zhang, Jonathan Mace, Dax Vandevoorde, Pedro Las-Casas, Shachee Mishra Gupta, et al. Building ai agents for autonomous clouds: Challenges and design principles. In *Proceedings of the 2024 ACM Symposium on Cloud Computing*, pages 99–110, 2024. (Cited on pages 2, 3, and 4.)

Tracer-Cloud. OpenSRE: Build your own AI SRE agents. https://github.com/Tracer-Cloud/opensre, 2026. Software. Evaluated at commit 21273dc. (Cited on pages 7 and 36.)

Yilun Wang, Guangba Yu, Haiyu Huang, Zirui Wang, Yujie Huang, Pengfei Chen, and Michael R Lyu. Cloud-opsbench: A reproducible benchmark for agentic root cause analysis in cloud systems. *arXiv preprint arXiv:2603.00468*, 2026. (Cited on pages 2, 3, and 4.)

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*, 2025. (Cited on pages 2, 3, and 4.)

John Yang, Carlos E Jimenez, Alex L Zhang, Kilian Lieret, Joyce Yang, Xindi Wu, Ori Press, Niklas Muennighoff, Gabriel Synnaeve, Karthik R Narasimhan, et al. Swe-bench multimodal: Do ai systems generalize to visual software domains? In *The Thirteenth International Conference on Learning Representations*, 2025. (Cited on page 3.)

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, et al. Judging llm-as-a-judge with mt-bench and chatbot arena. *Advances in neural information processing systems*, 36:46595–46623, 2023. (Cited on page 6.)

Jin Peng Zhou, Séb Arnold, Nan Ding, Kilian Q Weinberger, Nan Hua, and Fei Sha. Graders should cheat: privileged information enables expert-level automated evaluations. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 16583–16601, 2025. (Cited on page 6.)

APPENDIX CONTENTS

A Feature Flags **13**

A.1 Example: Product Catalog Failure 13

A.2 Example: Recommendation Cache Failure 15

B Incident Schedule **16**

B.1 Incident Schedule Constraints 16

B.2 ORCA-bench Incident Schedule 17

C Task Instruction **18**

D Details of User-Facing Issue Generation **19**

E Details of Ground-Truth Root Cause Curation **20**

F Details of Ground-Truth Symptoms Curation **20**

G Details of LLM Judge **21**

G.1 Evaluation Prompts 21

G.2 Human Agreement with the LLM Judge 23

H Formal Metric Definitions **24**

I Supplementary Experiment Details **25**

I.1 Telemetry Outcome Classification 25

J Full Results **26**

J.1 Telemetry and Code 26

J.2 Telemetry Only 29

J.3 Token Usage by Access Level 31

J.4 Command Categories 31

K Full case-study RCA reports **33**

K.1 GLM-5 — Human 0 / 3, GPT-5.4 judge 0 / 3 33

K.2 Claude Opus 4.7 — Human 3 / 3, GPT-5.4 judge 3 / 3 34

L Additional Experiments **35**

L.1 Comparison of Agent Harnesses 36

L.2 Additional RCA Analyses 38

A FEATURE FLAGS

Tab. A.1 lists the 11 feature flags used in the ORCA-bench schedule (App. B.2), with a condensed mechanism description and the number of ground-truth symptoms in metrics, logs, and traces (union across all schedule occurrences). In App. A.1 and A.2, we provide the full mechanism and symptoms for the two feature flags in Tab. A.2.

Table A.1: **Feature flag details.** The 11 feature flags used to inject incidents in the schedule, with ground-truth symptom counts (union across all schedule occurrences). Descriptions are condensed from the per-flag rubrics in the shipped JSON dataset; metric families are counted by unique (metric family, defining labels) pairs.

Flag	Description	# Occ.	Metrics	Logs	Traces
`adFailure`	Ad service throws gRPC `UNAVAILABLE` on ∼10% of `GetAds` calls; the frontend's React Query fallback silently renders a blank ad banner.	3	9	2	1
`adHighCpu`	Four daemon threads in the ad service spin tight `Math.log` loops, pinning container CPU and `jvm_cpu_recent_utilization_ratio`.	4	11	1	0
`adManualGc`	Every `GetAds` RPC schedules a forced major GC on a recurring 10 s timer, producing stop-the-world JVM pauses on the ad service.	4	11	2	0
`cartFailure`	`CartService.EmptyCartAsync` is routed to a Valkey store pointed at an unreachable host; `EmptyCart` RPCs hang and the frontend returns HTTP 504 on `POST /api/checkout` and `DELETE /api/cart`.	3	9	3	1
`imageSlowLoad`	The frontend `ProductCard` component adds an `x-envoy-fault-delay-request` header to every image fetch; Envoy delays each `GET /images/products/*` response by several seconds.	3	0	0	0
`kafkaQueueProblems`	The checkout service duplicates each `OrderResult` $1{+}N$ times on Kafka; the accounting consumer hits Postgres `23505` unique-key violations on every duplicate.	3	7	4	0
`loadGeneratorFloodHomepage`	A Locust task in the load generator fires 100 rapid GETs at the homepage on ∼16% of cycles, spiking CPU and network on the frontend and frontend-proxy.	3	7	1	0
`paymentFailure`	The payment service throws on ∼10% (or higher variants) of `charge` calls; checkout returns gRPC `Internal` and the frontend surfaces HTTP 500 on `POST /api/checkout`.	11	10	2	1
`paymentUnreachable`	Checkout rebuilds its payment gRPC client against an unresolvable host; DNS failure returns gRPC `Unavailable` and HTTP 500 on `POST /api/checkout`.	3	11	1	1
`productCatalogFailure`	`GetProduct` returns gRPC `Internal` whenever the SKU is `OLJCESPC7Z`; cascades to HTTP 500s on `/api/products/...`, `/api/cart`, `/api/recommendations`, and `/api/checkout`.	4	12	3	3
`recommendationCacheFailure`	The recommendation service's cache-miss path calls `GetProduct` with an empty request; the catalog returns `NOT_FOUND`, surfacing as HTTP 500 on `/api/recommendations`.	1	4	1	1

A.1 EXAMPLE: PRODUCT CATALOG FAILURE

Mechanism. The feature flag is read in `main.go#L485-L495`: the product-catalog service's `GetProduct` handler calls `checkProductFailure`, which queries flagd for the `productCatalogFailure` flag only when the requested product ID is `OLJCESPC7Z`. When

Table A.2: **Example feature flags and their symptoms.**

Feature flag	Symptoms
Product Catalog Failure	**Metrics** A feature flag activates a hard-fail path in the product-catalog `GetProduct` API for SKU `OLJCESPC7Z`, returning gRPC `INTERNAL`. **Traces** The `GetProduct` span carries `feature_flag.key=productCatalogFailure` and the event "`Error: Product Catalog Fail Feature Flag Enabled`". **Logs** Envoy access logs show `GET /api/products/OLJCESPC7Z`, `GET /api/recommendations`, and `POST /api/checkout` all returning 500. **Frontend** The product page for the National Park Foundation Explorascope no longer shows product information.
Recommendation Cache Failure	**Metrics** A feature flag reroutes the recommendation service's cache-miss path to call `GetProduct` with an empty request, and the `app_recommendations_counter` flatlines to zero as `ListProducts` calls collapse. **Traces** The resulting `GetProduct` span returns `NOT_FOUND` with an empty `app.product.id`, propagating to `ListRecommendations` as gRPC `UNKNOWN`. **Logs** Recommendation-service logs alternate `cache miss`/`cache hit` entries, the latter always returning an empty `cached_ids`. **Frontend** The recommendations section fails to load (∼50% of requests, always on the first call), and renders empty on the remaining cache-hit requests.

the flag is enabled (`main.go#L439-L444`): (1) the span is marked as an error via `span.SetStatus(otelcodes.Error, msg)`, (2) a span event is added via `span.AddEvent(msg)`, and (3) a gRPC `codes.Internal` error is returned via `status.Errorf`. The frontend calls `GetProduct` for product pages, cart enrichment, recommendation enrichment, and checkout preparation—all unguarded—so the single-SKU failure cascades into HTTP 500s on `/api/products/OLJCESPC7Z`, `/api/cart`, `/api/recommendations`, and `/api/checkout`.

Frontend symptoms.

- **Product page** (`GET /api/products/OLJCESPC7Z`): returns HTTP 500 and shows no product information. Deterministic.
- **Cart page** (`GET /api/cart`): returns HTTP 500 and shows an empty or broken cart once `OLJCESPC7Z` has been added—add-to-cart succeeds silently, but the cart page's per-item `GetProduct` enrichment then fails for every item until the cart is cleared. Deterministic once the SKU is in the cart.
- **Checkout** (`POST /api/checkout`): returns HTTP 500 with *"failed to get product #OLJCESPC7Z"* whenever `OLJCESPC7Z` is in the cart. Deterministic once the SKU is in the cart.
- **Recommendations** (`GET /api/recommendations`): returns HTTP 500 instead of related products when `OLJCESPC7Z` is among the recommended IDs. Non-deterministic—depends on the randomly sampled recommendation set.

Metric families. Tab. A.3 lists the union of all metric families across the four schedule occurrences (d2, d3, d4, d6), grouped by signal layer.

Log clusters. Three log clusters are observed, all from `frontend-proxy` Envoy access logs (`event.name=proxy.access`):

- **Product page 500s** (4/4): `GET /api/products/OLJCESPC7Z` returns HTTP 500. Deterministic—fires on every request for the affected SKU.
- **Recommendations 500s** (4/4): `GET /api/recommendations` returns HTTP 500 when `OLJCESPC7Z` is among the randomly sampled recommendation IDs. Non-deterministic.
- **Checkout 500s** (2/4): `POST /api/checkout` returns HTTP 500 when `OLJCESPC7Z` is in the cart. Non-deterministic—requires the load generator to attempt checkout with the affected SKU.

Trace clusters. Three call-chain patterns are observed. All share the same smoking-gun span attributes: `feature_flag.key=productCatalogFailure`, `feature_flag.result.variant=on`; span

Table A.3: Metric signals for `productCatalogFailure` (union across 4 occurrences). Each row is a unique (metric family, defining labels) pair.

Signal Layer	Metric Family	Key Labels	Behavior
Root cause	rpc_server_duration_milliseconds	service=product-catalog, method=GetProduct, code=13	spiked
	traces_span_metrics_calls	service=product-catalog, span=GetProduct, status=ERROR	spiked
	traces_span_metrics_duration_ms	service=product-catalog, span=GetProduct, status=ERROR	spiked
	rpc_server_requests_per_rpc	service=product-catalog, method=GetProduct, code=13	spiked
Propagation	rpc_client_duration_milliseconds	service=checkout, method=GetProduct, code=13	spiked
	rpc_server_duration_milliseconds	service=checkout, method=PlaceOrder, code=13	spiked
	traces_span_metrics_calls	service=frontend, span=GetProduct, status=ERROR	spiked
	app_frontend_requests	method=GET, status=200, target=/api/products/OLJCESPC7Z	collapsed
Symptom	app_frontend_requests	method=GET, status=500, target=/api/products/OLJCESPC7Z	spiked
	http_server_duration_milliseconds	status=500	spiked
	traces_span_metrics_calls	service=frontend-proxy, span=GET, status=ERROR	spiked
Meta	feature_flag_flagd_impression	key=productCatalogFailure, variant=on	spiked

event *"Error: Product Catalog Fail Feature Flag Enabled"*; and frontend exception *"13 INTERNAL: Error: Product Catalog Fail Feature Flag Enabled"*.

- **Product page cascade** (4/4): `product-catalog:GetProduct` → `frontend:GetProduct` → `frontend:GET /api/products/{id}` → `frontend-proxy:GET` → `load-generator:GET`.
- **Recommendations cascade** (4/4): `product-catalog:GetProduct` → `frontend:GetProduct` → `frontend:GET /api/recommendations` → `frontend-proxy:GET` → `load-generator:GET`.
- **Checkout cascade** (2/4): `product-catalog:GetProduct` → `checkout:GetProduct` → `checkout:PlaceOrder` → `frontend:PlaceOrder` → `frontend:POST /api/checkout` → `frontend-proxy:POST` → `load-generator:POST`.

A.2 Example: Recommendation Cache Failure

Mechanism. The flag is read in `recommendation_server.py#L78`. When enabled, on every call the code branches into the cache-failure path: with probability 0.5 (and always on the first call, since `first_run=True`), it logs `get_product_list: cache miss` and calls `GetProduct(demo_pb2.Empty())` at line 84—i.e., with no product ID. The product-catalog service fails the lookup at `main.go#L448` and returns gRPC `NOT_FOUND` with message *"Product Not Found: "*. The exception propagates through `get_product_list` → `ListRecommendations` → the frontend's gRPC client, which receives `UNKNOWN` and returns HTTP 500 on `GET /api/recommendations`. On the cache-hit branch (probability 0.5, never on the first call), the code returns the global `cached_ids`, but this is permanently `[]` because every preceding cache-miss attempt fails before reaching the assignment on line 86. `ListRecommendations` therefore returns zero product IDs and the frontend renders an empty recommendations section.

Frontend symptoms.

- **Recommendations error** (`GET /api/recommendations`, ∼50% of requests and always on the first call): returns HTTP 500 because the cache-miss path calls `GetProduct(Empty())`, which the catalog returns as gRPC `NOT_FOUND`.
- **Recommendations empty** (`GET /api/recommendations`, ∼50% of requests, never on the first call): returns HTTP 200 with an empty recommendations section, since `cached_ids` is permanently `[]`.

Metric families. Tab. A.4 lists the metric signals for the single schedule occurrence (d6).

Log clusters. Two log clusters are observed, both INFO-level from the `recommendation` service:

- **Cache miss** (1/1): Body `get_product_list: cache miss`. Emitted at `recommendation_server.py:83` when the flag is on and `first_run=True` or `random.random() < 0.5`. Direct evidence that the flag is active. 7 entries observed.
- **Cache hit** (1/1): Body `get_product_list: cache hit`. Emitted at `recommendation_server.py:91` on the complementary branch. Returns permanently empty `cached_ids`. 8 entries observed.

Table A.4: Metric signals for `recommendationCacheFailure` (1 occurrence).

Signal Layer	Metric Family	Key Labels	Behavior
Root cause	`rpc_server_duration_milliseconds`	`service=product-catalog, method=ListProducts, code=0`	collapsed
	`traces_span_metrics_duration_ms`	`service=recommendation, span=ListProducts, status=UNSET`	collapsed
	`app_recommendations_counter`	`recommendation_type=catalog`	collapsed
Symptom	`traces_span_metrics_duration_ms`	`service=frontend, span=ListRecommendations, status=ERROR`	created
Meta	`db_sql_connection_closed_max_idle`	`service=product-catalog`	collapsed

Trace clusters. Two call-chain patterns are observed, both showing `GetProduct` called with an empty product ID:

- **NOT_FOUND cascade** (1/1, 6 traces): `product-catalog:GetProduct` → `recommendation:get_product_list` → `recommendation:ListRecommendations` → `frontend:ListRecommendations` → `frontend:GET /api/recommendations` → `frontend-proxy:GET` → `load-generator:GET`. Flag evaluation reason is `cached` on every trace.
- **NOT_FOUND cascade with flagd resolve** (1/1, 1 trace): Same cascade but prefixed by `flagd:ResolveBoolean` → `recommendation:ResolveBoolean`. Captures the initial live flag resolution (`reason=static`) before the OpenFeature SDK caches the result.

Smoking-gun span attributes shared across both clusters:

- `app.product.id=`" (empty string) and span event *"Product Not Found: "* on `product-catalog:GetProduct`
- `app.recommendation.cache_enabled=True`, `app.cache_hit=False` on `recommendation:get_product_list`
- `feature_flag.key=recommendationCacheFailure`, `feature_flag.result.variant=on`

B Incident Schedule

For the second stage of Sec. 3, we generate a 6-day schedule of feature flag on/off events to simulate realistic incident scenarios.

B.1 Incident Schedule Constraints

The 6-day ORCA-bench schedule is built from the 11 feature flags listed in Tab. A.1. Day 6 models a "FAFO Friday" chaos peak.

Constraints. The schedule is generated subject to the following constraints, developed with an expert SRE.

C0. Max concurrency: At most 6 flags may have `defaultVariant != "off"` at any given time.

C1. Peak-duration cap: There cannot be any period longer than 2 hours during which exactly 6 flags are ON simultaneously.

C2. Overnight gap: For each non-weekend transition $N-1 \to N$ with $N \in \{2,3,4,5,6,10\}$, $\text{earliest}(N) - \text{latest}(N-1) \geq 12\text{h}$ on the wall clock. For example, if day 1 has an incident from 20:00–23:00, the earliest day-2 incident must be at or after 11:00. **Weekend exception**: transitions $6 \to 7$, $7 \to 8$, and $8 \to 9$ (Friday→Saturday, Saturday→Sunday, Sunday→Monday) are exempt — chaos may bleed into the weekend and on through Monday morning with arbitrarily short gaps.

C3. Equal scenarios per type: Each of the five scenario types (Isolated, Independent, Conflicting, Cascading, Sequential) must have the same count N across the schedule. A single scenario may contribute to multiple types.

C4. Day 6 peak: Day 6 (Friday) must have the strictly highest peak concurrent-flag count of any day — the "FAFO Friday" chaos.

C5. Warm-up window: No incidents during the first 8 hours of the schedule (day 1, 00:00–07:59).

C6. Minimum incident duration: Each incident must be at least 30 minutes long.

C7. Daily flag budget: Each day can use at most 6 distinct flags (unique flags toggled at any point during the day).

C8. No inter-scenario symptom overlap within a day: For any two distinct scenarios A and B on the same day, and any $f_A \in \text{flags}(A)$, $f_B \in \text{flags}(B)$:

(a) $f_A \neq f_B$ (scenarios on the same day cannot reuse a flag), and
(b) f_A and f_B must not share an overlap edge (see Tab. B.1).

Intra-scenario overlap is still allowed — that is what defines the Conflicting, Conflicting-Cascading, and Sequential types. As a consequence, the overlap clique {`cartFailure`, `paymentFailure`, `paymentUnreachable`, `productCatalogFailure`} (plus its `recommendationCacheFailure` extension) can appear in at most one scenario per day, and similarly {`adHighCpu`, `adManualGc`} cannot be split across two scenarios on the same day.

C9. Checkout-path duration cap: Any incident on `cartFailure`, `paymentFailure`, or `paymentUnreachable` must last ≤ 2 hours (120 minutes).

C10. Global duration cap: No incident (flag activation) can last longer than 5 hours (300 minutes).

Overlap graph. Two flags share an overlap edge if they share at least one exact fingerprint (metric family plus labels, log name, frontend route, or trace call chain). Tab. B.1 enumerates the edges used by C8.

Table B.1: Overlap edges between feature flags.

Flag	Overlapping flags
`adHighCpu`	`adManualGc`
`cartFailure`	`paymentFailure`, `paymentUnreachable`, `productCatalogFailure`
`paymentFailure`	`paymentUnreachable`, `productCatalogFailure`
`paymentUnreachable`	`productCatalogFailure`
`productCatalogFailure`	`recommendationCacheFailure`
Graph-isolated (no edges):	`adFailure`, `imageSlowLoad`, `kafkaQueueProblems`, `loadGeneratorFloodHomepage`

The maximum overlap clique is {`cartFailure`, `paymentFailure`, `paymentUnreachable`, `productCatalogFailure`} (size 4, all pairs connected); it is used for sequential scenarios and the day-6 peak scenario.

B.2 ORCA-BENCH INCIDENT SCHEDULE

This appendix walks through the schedule day by day. Every event referenced below corresponds to a specific entry in the published JSON schedule files. Times are in the schedule's local 24-hour clock.

Day 1. Sunday: warm-up. A weekend shopper reports slow product images at 10:00, and `imageSlowLoad=5sec` runs until 12:00. Half an hour later, the ad service starts intermittently failing — `adFailure` is on from 12:30 to 16:00 (3.5 h), buried because Sunday traffic is low and nobody is watching. In the late afternoon, two unrelated background issues co-occur: `kafkaQueueProblems` from 16:30–19:00 and a scheduled `loadGeneratorFloodHomepage` test from 16:45–18:45 — the consumer-lag and synthetic-traffic spike happen to overlap but share no symptoms. The day ends with an evening ad-service problem: `adHighCpu` from 19:30–21:02 with `adManualGc` overlapping from 19:35–21:00, exhibiting the shared CPU/GC fingerprint that defines the conflicting-symptom case.

Day 2. Monday. A Monday-morning ad cascade unfolds: `adHighCpu` (10:00) → `adManualGc` (10:25) → `adFailure` (11:00), all contained inside the CPU window, ending in reverse 12:30 / 13:00 / 13:30. In the afternoon a catalog deploy goes through a canary:

productCatalogFailure runs 14:30–16:00, then paymentFailure escalates from 10% at 16:00 to 50% at 16:30 and is rolled back at 17:00, and cartFailure lingers from 17:00–18:30 holding stale payment references.

Day 3. Tuesday. A morning CDN-driven cascade: imageSlowLoad outer 10:00–12:30, kafkaQueueProblems middle 10:20–12:10 (image-processing queue builds), loadGeneratorFloodHomepage inner 10:45–11:45 (retry storm). Afternoon: another canary deploy — productCatalogFailure 14:30–16:00, paymentFailure 10%→50% 16:00–17:00, then paymentUnreachable 17:00–18:30 from the rolling restart.

Day 4. Wednesday. The ad-service cascade returns: adHighCpu outer 10:00–13:30, adManualGc middle 10:25–13:00, but this time kafkaQueueProblems (11:00–12:30) is the inner layer (ad-events queue saturates inside the CPU window). Afternoon canary same shape as Day 3.

Day 5. Thursday. Pre-Friday ad flakiness: adFailure outer 10:00–12:30 with imageSlowLoad middle 10:25–12:10 and loadGeneratorFloodHomepage inner 11:00–11:45 (retry storm). Afternoon cart canary fails fast — cartFailure 14:30–16:00, then paymentFailure 10%→50% rolled back at 17:00, paymentUnreachable 17:00–18:30. Things look healthy heading into the weekend.

Day 6. Friday: FAFO chaos peak. A Friday-afternoon deploy unwinds into the schedule's only six-concurrent-flag peak: adHighCpu saturates first at 12:00, adManualGc follows at 12:15 (GC storm), productCatalogFailure at 12:30 (catalog read replicas saturate), recommendationCacheFailure at 12:45 (cache invalidates), cartFailure at 13:00 (cart returns 500s), and paymentFailure begins canary at 10% at 13:15, escalating to 50% at 13:45 and 100% at 14:15 (peak failure, all-hands paged). Tail-down: productCatalogFailure recovers first at 14:15, then cartFailure and recommendationCacheFailure at 14:30, paymentFailure rolls back at 14:45, adManualGc at 15:00, and adHighCpu finally normalizes at 17:00 — a five-hour ad-CPU bleed that is the longest single-flag incident in the schedule.

C Task Instruction

Every task in ORCA-bench instantiates the following instruction template, which the SRE agent receives as its prompt. Here, {current} is the detection time (e.g., "Apr 24, 2026 at 00:28 ET"), {reported} is the styled report time (e.g., "between 4:08 PM and 5:08 PM yesterday ET" or "sometime in the afternoon today"; see B3 in Sec. 3), and {user_facing_issue} is the user-facing issue generated in App. D. The note below tells the agent that the report time is only an approximate lead: it need not coincide with when the incident began, and the agent should determine the root cause incidents and their times up to the detection time.

Task Instruction (excerpt)

```
# Root Cause Analysis
You are an expert site reliability engineer. The current time is {current}.
## Task Description
The following issue was reported {reported}:
{user_facing_issue}
NOTE:
* The time that an issue was reported is not necessarily the same as the time the incident actually began. Your
task is to pinpoint the root cause events and the times of their occurrence.
* There may be multiple root causes or no root cause at all.
## Source Code
The source code for the system is available at /app/opentelemetry-demo.
## Telemetry Tools
Please use the following tools:
- Grafana HTTP API (available at $GRAFANA_URL with username admin and password admin; the URL is set in the
GRAFANA_URL environment variable)
## Instructions
First determine whether an incident actually occurred. If no incident occurred, please write an empty file to
/app/report.md. If an incident occurred, please write a structured incident RCA report to /app/report.md with four
sections: Summary, Timeline, 5 Whys, and Remediation. Follow all section-specific rules below.
[. . . section-specific rules for Summary, Timeline, 5 Whys, and Remediation omitted . . . ]
```

D Details of User-Facing Issue Generation

For the fourth stage of Sec. 3, we prompt GPT-5.4 with reasoning effort "high" with an instantiated version of the following template:

Question Generation Prompt

```
**OpenTelemetry Demo** is composed of microservices written in different programming languages that talk to each
other over gRPC and HTTP; and a load generator which uses Locust to fake user traffic.
Here is the full architecture:
graph TD subgraph Service Diagram accounting(Accounting):::dotnet ad(Ad):::java cache[(Cache
(Valkey))] cart(Cart):::dotnet checkout(Checkout):::golang currency(Currency):::cpp email(Email):::ruby
flagd(Flagd):::golang flagd-ui(Flagd-ui):::elixir fraud-detection(Fraud Detection):::kotlin
frontend(Frontend):::typescript frontend-proxy(Frontend Proxy (Envoy)):::cpp image-provider(Image Provider
(nginx)):::cpp load-generator([Load Generator]):::python payment(Payment):::javascript product-catalog(Product
Catalog):::golang quote(Quote):::php recommendation(Recommendation):::python shipping(Shipping):::rust
queue[(queue (Kafka))]:::java react-native-app(React Native App):::typescript postgresql[(Database (PostgreSQL))]
accounting -> postgresql
ad -->|gRPC| flagd
checkout ->|gRPC| currency checkout ->|gRPC| cart checkout ->|TCP| queue
cart -> cache cart ->|gRPC| flagd
checkout ->|gRPC| payment checkout ->|HTTP| email checkout ->|gRPC| product-catalog checkout ->|HTTP| shipping
fraud-detection ->|gRPC| flagd
frontend ->|gRPC| ad frontend ->|gRPC| currency frontend ->|gRPC| cart frontend ->|gRPC| checkout frontend ->|HTTP|
shipping frontend -->|gRPC| recommendation frontend ->|gRPC| product-catalog
frontend-proxy ->|gRPC| flagd frontend-proxy ->|HTTP| frontend frontend-proxy ->|HTTP| flagd-ui frontend-proxy
->|HTTP| image-provider
payment ->|gRPC| flagd
queue ->|TCP| accounting queue ->|TCP| fraud-detection
recommendation ->|gRPC| flagd recommendation ->|gRPC| product-catalog
shipping ->|HTTP| quote
Internet ->|HTTP| frontend-proxy load-generator ->|HTTP| frontend-proxy react-native-app ->|HTTP| frontend-proxy
end
Here is the ground-truth rubric for this incident:
{current_rubric} {other_rubrics_section} Please generate the exact issues a user would face and perceive. Only
emit issues a real human user could actually notice from the UI — a visibly broken element, a clearly slow page
(multiple seconds), an error message they can see, missing or blank content, etc. Do NOT emit issues that are only
visible in metrics, logs, or numerical comparisons. Counter-example: "page latency rose from 10 ms to 190 ms" is
NOT perceivable — a sub-200ms change is invisible to humans and only shows up in dashboards.
For each affected feature area, emit one or more **medium** descriptions naming the broad feature area the user
is impacted in (one or a few alternate phrasings per area). For each distinct user-visible issue you identify,
additionally emit an **easy** variant of that same underlying issue. Granularity levels:
- **medium**: a single short clause from the customer's perspective describing the BROAD feature area affected
(e.g., "ads", "cart", "product page", "recommendations", "checkout"), NOT the specific symptom or action. MAX 7
WORDS. Must start with a user-perspective subject such as "users are...", "customers cannot...", "shoppers see...".
No product names, page names, error codes, messages, users, regions, or locales. Describe what the user actually
sees on the surface they interact with (e.g., "product page", "checkout", "cart"); do NOT name underlying technical
components or resources the user does not see (e.g., "image", "API", "service", "cache"). Examples: "users are
having issues with ads", "users are having issues with cart", "users cannot view product pages", "users are unable
to pay", "users are having payment issues", "customers cannot complete payment", "shoppers are unable to pay",
"users say product page not loading properly". Emit at least ONE medium per affected feature area. You MAY emit
a small number (typically 1–3) of alternate phrasings for the same feature area to capture different customer
voices — vary the subject ("users"/"customers"/"shoppers") and the verbs. Never emit one per individual symptom. -
**easy**: name the product / page / feature, the concrete symptom, the specific error or error-message string the
user sees, and (when applicable) the affected user, region, or locale. Example: "Product page for National Park
Foundation Explorascope is having X issue and is showing Y errors with Z message for W user".
Each **easy** variant describes one specific user-visible issue and must describe the SAME root cause (the same
feature flag) as this incident. The **medium** variants describe higher-level feature impact and are NOT bound
1:1 to the easy variants — emit them once per affected feature area, independently (you may place them before,
after, or interleaved with the easy variants). Use the lowercase granularity tokens exactly: "easy", "medium".
{prior_questions_section} Return your answer as JSON matching the provided schema.
```

where `{current_rubric}` is the ground-truth rubric for the target incident, `{other_rubrics_section}` lists the rubrics for the other feature flags in the system (used to prevent the generated questions from overlapping with unrelated incidents), and `{prior_questions_section}` lists questions already generated for the same feature flag in earlier incidents (used to encourage distinct phrasings). This prompt generates only the **Easy** and **Medium** levels of issue specificity; **Hard** user-facing issues are not LLM-generated but instead come from a fixed, flag-agnostic template ("users are reporting site issues") appended to every incident's task set. Using the feature flags in Tab. A.2, we provide examples of generated use-facing issues:

Product Catalog Failure:

- The product page for National Park Foundation Explorascope does not show product information.

- Clicking "Add to Cart" on any product page directs the user to an empty shopping cart page.

Recommendation Cache Failure:

- Product recommendations fail to load: the related/recommended products section is empty or broken because `/api/recommendations` returns 500.
- Any page or UI element that depends on the recommendations API may show a generic error instead of recommended items.

E Details of Ground-Truth Root Cause Curation

For the fifth stage of Sec. 3, we prompt GPT-5.4 with reasoning effort "high" and max tokens 16384 with the instantiated version of the following prompt template. Here, the issue is generated using the procedure in App. D.

Answer Generation Prompt

```
You are an expert SRE deciding which of several recent feature-flag-driven incidents could plausibly explain a
user-reported issue. The reported issue is short and may be flag-agnostic; multiple incidents may match.
## Reported issue
{issue}
## Candidate incidents
For each candidate below, decide whether its user-visible effect could plausibly explain the reported issue. Base
your decision on the rubric's description, mechanism, and (where present) the user-facing frontend symptoms. Be
inclusive: if the symptoms could reasonably be confused with the reported issue, mark the candidate plausible.
You must return one verdict per candidate, in the same order as listed. Each verdict has:
  - event_id: copy verbatim from the candidate.
  - plausible: true if the candidate could explain the reported issue.
  - reason: one short sentence justifying your verdict.
The order of verdicts must match the order of candidates. Do not omit any.
{candidate_blocks}
```

where `{issue}` is the user-facing issue from the task spec and `{candidate_blocks}` is the concatenation of one Markdown block per candidate event of the form

Candidate Block

```
### Candidate {i}: {event_id}
{rubric_md}
```

with `{rubric_md}` rendered from the candidate's ground-truth rubric (feature flag header, description, incident time, mechanism, and metrics / logs / frontend / traces evidence sections). The model is constrained by a JSON schema that requires exactly one verdict object per candidate, each with the fields `event_id`, `plausible`, and `reason`, in the same order as the candidates. For control tasks (where no incident is active at the reported time) we skip this LLM call entirely and emit an empty answer set.

F Details of Ground-Truth Symptoms Curation

For the sixth stage of Sec. 3, we curate the ground-truth metrics, logs, and traces for each event using a semi-automated workflow that combines LLM assistance with human verification.

Step 1: Logs and traces. We programmatically query OpenSearch and Jaeger for all logs and traces with the tag `error=true`, respectively, in the period ± 10 minutes around the ground-truth event time. The complete logs and traces for a single feature flag can fit in a single file (median 302K lines and max 1M lines for logs, median 20K lines and max 141K lines for traces). For each event, we then provide the corresponding log and trace files to Claude Code (Claude Opus 4.7 with "xhigh" effort)—along with the ground-truth feature flag, event time, source code, official OpenTelemetry documentation, and Grafana—and ask it to identify logs and traces clusters that confirm the root cause. Finally, we manually verify each result and discard ones that are inconclusive or background noise.

Step 2: Metrics. Prometheus contains more than 10K unique time series across 478 metric names in our collected data, so we cannot employ the same exhaustive "query-then-filter" approach for finding ground-truth metrics.[2] We first group the metrics into five families ("application," "infrastructure," "database," "runtime," and "telemetry pipeline"). For each family, we prompt Claude Code (Claude Opus 4.7 with "max" effort) with access to the ground-truth feature flag, event time, source code, official OpenTelemetry documentation, and Grafana as well as the ground-truth logs and traces (from the previous step) to identify candidate metrics that either spiked, collapsed, or was created ± 12 hours around ground-truth incident time. We plot each candidate metric as a time series and manually fix missclassified metrics and discard inconclusive metrics. Next, we run an LLM-assisted onset-detection step that pinpoints, for each candidate metric, the UTC timestamp at which it first deviates from its pre-incident baseline. Finally, we group the verified metrics by metric family and annotate each with one of the following signal layers using an LLM call.

Root cause. Signals directly on the service or component where the fault originates. These are the first to fire and pinpoint the broken code path (e.g., gRPC `Internal` errors on `product-catalog:GetProduct`).

Propagation. Signals on intermediate services that relay the failure to downstream consumers without originating it. They appear because the calling service does not isolate or gracefully degrade the upstream error (e.g., `checkout` receiving gRPC errors when calling `product-catalog`).

Symptom. Signals on user-facing surfaces that reflect the end-user impact. These are what an on-call engineer or an end user would notice first (e.g., HTTP 500s on the frontend's `/api/products` endpoint).

Meta. Telemetry about the telemetry infrastructure or the fault-injection mechanism itself, rather than about application behavior. Useful for confirming that a flag was active but not diagnostic on its own (e.g., `feature_flag_flagd_impression` counts).

G Details of LLM Judge

G.1 Evaluation Prompts

We use two LLM-judge prompts depending on whether the task is an incident task or a control (no-incident) task. Both prompts return a JSON object constrained by a JSON Schema delivered via OpenAI Structured Outputs.

G.1.1 Incident-task judge

The incident-task prompt is constructed dynamically from the rubric(s) attached to the task: a fixed header asking the per-rubric yes/no questions, followed by one checklist block per ground-truth metric, log, and trace cluster within each rubric, followed by a fixed scoring footer. The model returns one entry per rubric with one boolean per required field per cluster plus an integer 0–3 `score`. The empty-report short-circuit applies when the agent emitted nothing despite an incident: every rubric scores 0 without an LLM call.

Incident judge — header

```
You are an expert SRE tasked with judging the quality of an AI-generated incident RCA report.
You are given:

1. One or more ground-truth rubrics. Each describes a plausible root cause, its incident time, mechanism, and
   symptoms. Several flags may have been active in the same window; the agent is credited if it correctly
   identifies any one of these rubrics.

2. The SRE agent's incident RCA report.

## SRE Agent's Incident RCA Report
{predictions}
## Ground-Truth Rubrics
{rubrics}
## Evaluation Questions
```

[2] A time series is a stream of timestamped values for a specific metric name and set of labels.

For **each rubric** in order, answer every question with true or false. Base your answer only on evidence in the agent's report; if the report does not cite the required evidence, answer false. The output must contain one entry per rubric, in the same order, and must preserve cluster order within each rubric exactly as listed.
For each rubric:

- incident_time_within_10min: Did the agent's timeline place the incident start within ± 10 minutes of this rubric's incident_time?
- feature_flag_match: Did the agent identify this rubric's feature flag as a root cause? (The agent may name multiple flags; mark true if this flag is among them.)
- mechanism_match: Did the agent correctly explain this rubric's mechanism — how the flag propagates to the user-visible failure?
- For each metric/log/trace cluster listed for this rubric: per-cluster match flags as below.
- score: An integer 0–3 reflecting how well the report matches **this rubric specifically** (see the Per-Rubric Scoring Rubric at the end of this prompt).

The per-cluster checklist follows. One block per metric, log, and trace cluster within each rubric; empty sections are omitted. Angle brackets denote values interpolated from the rubric.

Incident judge — per-cluster checklist (schematic)

Metric clusters

- (for each metric cluster i:) Cluster i: metric_family = "<family>" — family_match: did the agent cite a query targeting metric_family <family> under a Why-step whose causal chain aligns with the rubric?

Log clusters

- (for each log cluster i:) Cluster i:
  - resource.service.name: <service_name>
  - body: <body>
  - body_match: did the agent cite log evidence whose body matches the rubric's body under a Why-step whose causal chain aligns with the rubric?
  - service_name_match: did the agent attribute the cited log evidence to the OTel service <service_name> (resource.service.name)?

Trace clusters

- (for each trace cluster i:) Cluster i:
  - call_chain: <call_chain>
  - smoking_gun: (bulleted list of the rubric's smoking-gun signals)
  - call_chain_match: did the agent cite evidence matching this service-level call chain under a Why-step whose causal chain aligns with the rubric?
  - smoking_gun_match: did the agent cite at least one of the listed smoking-gun signals?

Incident judge — per-rubric scoring footer

Per-Rubric Scoring Rubric
Assign each rubric a score from 0 to 3 reflecting how well the agent's report matches **that rubric specifically**:

- 0: AI findings are misaligned with this rubric's mechanism and completely miss this root cause
- 1: AI verifies incident description — the report confirms symptoms described in the task prompt but does not investigate further toward this root cause
- 2: AI makes correct progress beyond symptoms — the report identifies some of this rubric's ground-truth metrics, logs, or traces and moves toward this root cause, but does not fully identify it
- 3: AI nails this root cause — the report correctly identifies it, this rubric's mechanism, and the supporting signals

Scoring is based on content correctness only; do not penalize or reward for formatting, prose style, or structural polish. Materially incorrect or misleading claims count against the per-rubric score: if the report would otherwise score N for a rubric but also adds a materially wrong claim (e.g., invents an unrelated additional cause), drop that rubric's score to $N - 1$ (floor at 0). The same fabricated claim therefore lowers every rubric's score uniformly. When applying this penalty, explicitly identify in your reasoning which specific claim is materially wrong — quote or paraphrase the offending text. Do not apply the penalty without naming the offending claim; missing detail or incomplete explanation does not qualify as a materially wrong claim. Naming several of the listed plausible root causes is **not** a materially wrong claim.
Respond with ONLY a JSON object matching the provided output schema.

Per-section accuracy rollups (*all* / *any*) are derived post-hoc from the nested response: a metric cluster is considered matched iff `family_match` is true; a log cluster is matched iff both `body_match` and `service_name_match` are true; a trace cluster is matched iff both `call_chain_match` and `smoking_gun_match` are true. When a rubric section is empty (no ground-truth clusters), its rollups

are None so downstream averages normalize only over trials with ground truth in that section. The task-level overall score is the mean of the per-rubric score integers; we additionally surface the max as best_score.

G.1.2 CONTROL-TASK JUDGE

For control tasks (expected.events is empty), an empty report is a trivial pass with no LLM call. A non-empty report is judged by the prompt below, which asks a single boolean: did the agent place an incident strictly inside the quiet window? The schedule-derived quiet window is shrunk by a 10-minute margin on each defined side before being shown to the judge — the raw window is the maximal no-flag-active interval around current_time, and the margin absorbs minor agent imprecision near the boundary (e.g. claiming the incident ended a few minutes after it actually did).

Control judge

You are an expert SRE judging whether an AI-generated incident RCA report makes a false-positive claim about an incident at the queried time.
The task is a **control task**: the user queried the system at current_time during a verified quiet period in which no problem-pattern feature flag is active. The agent's snapshot includes telemetry from before current_time, so it may legitimately describe past, resolved incidents. Past incidents are **not** false positives — only claims placing an incident inside the quiet window are.
Queried time
current_time = {current_time}
Quiet window (no flag active inside)

- quiet_window_start = {quiet_window_start} (inclusive lower bound; "null" means open-ended, i.e. no neighboring incident before current_time)
- quiet_window_end = {quiet_window_end} (exclusive upper bound; "null" means open-ended, i.e. no neighboring incident after current_time)

SRE Agent's Incident RCA Report
{predictions}
Evaluation
Answer the single boolean question claims_incident_in_quiet_window:

- **true** iff the report places **any** incident — its start, ongoing period, or any reported timestamp of active impact — at a time **strictly inside** (quiet_window_start, quiet_window_end). Examples that count as true:
  - Claiming the incident is currently ongoing at current_time.
  - Reporting a timeline entry with a UTC timestamp inside the quiet window.
  - Stating that the incident started or is still active at any time inside the window.
- **false** otherwise. In particular, the following are NOT false positives:
  - Describing a past, resolved incident whose start, end, and timeline entries all fall at or before quiet_window_start.
  - Describing a future incident whose entries all fall at or after quiet_window_end.
  - Saying "no incident occurred at current_time" while describing past context.
  - **Timeline entries that merely document when the user reported the issue** (e.g. "02:04 UTC — Incident reported by user", "22:00 ET — issue reported"). These describe the moment the complaint arrived, not active system impact. Ignore such entries when applying the rule above; only entries that describe the system actively failing or degraded count.

Treat the bounds as half-open: a reported time exactly equal to quiet_window_start or exactly equal to quiet_window_end is **outside** the quiet window (no false positive). An open-ended bound (null) is equivalent to $-\infty$ / $+\infty$ respectively.
In reasoning, briefly state the verdict; if claims_incident_in_quiet_window is true, quote or paraphrase the specific offending claim from the report.
Respond with ONLY a JSON object matching the provided output schema.

A true verdict scores 0 (the agent fabricated an incident inside the quiet window); a false verdict scores 3 (the agent correctly avoided the false positive). The control judge does not contribute to the per-rubric metrics M1–M3 from Sec. 4; we report control-task accuracy separately.

G.2 HUMAN AGREEMENT WITH THE LLM JUDGE

Fig. G.1 and Fig. G.2 compare human and LLM-judge (GPT-5.4) task-level scores across all five agent models and three difficulty levels, scored by Annotators 1 and 2, respectively.

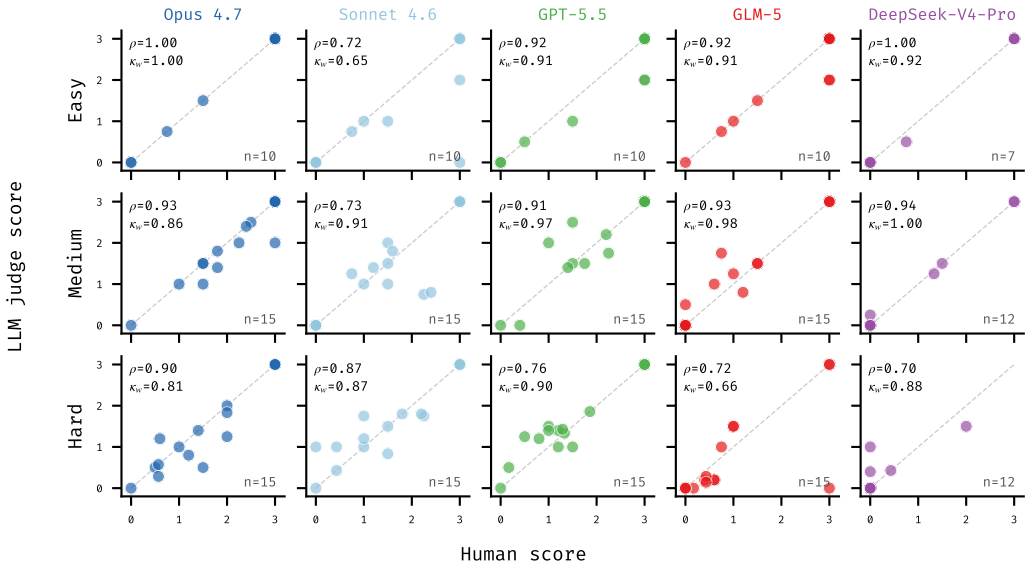


Figure G.1: **Annotator 1 vs. LLM judge (GPT-5.4) task-level scores on ORCA-bench Verified.** Each cell shows one model at one difficulty level; the dashed line marks perfect agreement. Per-cell Spearman ρ and quadratic-weighted Cohen's κ_w are shown.

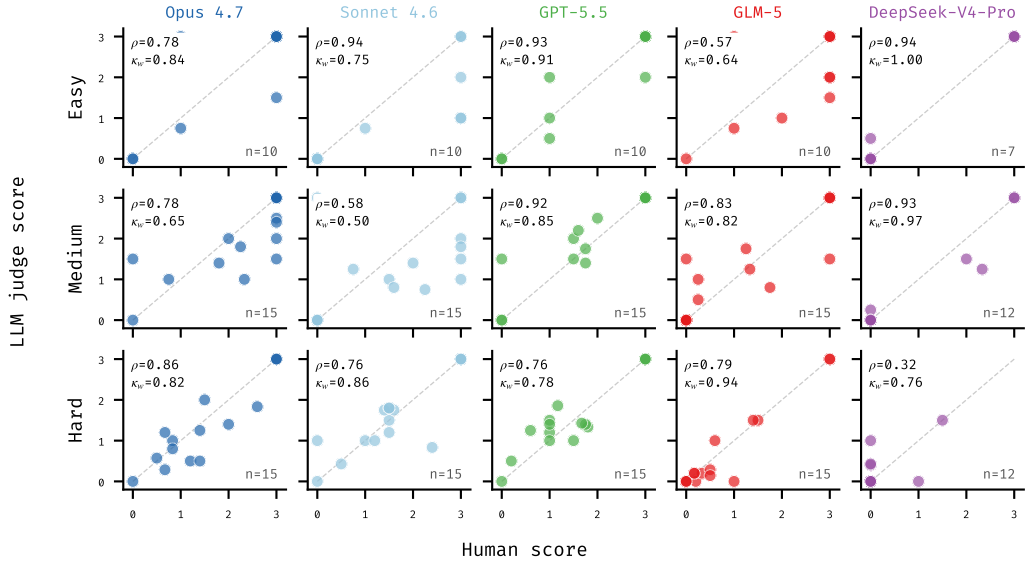


Figure G.2: **Annotator 2 vs. LLM judge (GPT-5.4) task-level scores on ORCA-bench Verified.** Same layout as Fig. G.1, scored independently by Annotator 2.

H Formal Metric Definitions

For an incident task, let G denote the set of ground-truth plausible root causes (one per active feature flag; Sec. 3) and P the set of root causes asserted in the agent's incident RCA report, as extracted by the LLM judge (App. G.1.1). The two binary task-level metrics of Sec. 4 are

$$\text{RCA accuracy} = \mathbb{I}\big[G \subseteq P\big], \tag{1}$$

$$\text{Hallucination rate} = \mathbb{I}\big[P \neq \emptyset \text{ and } P \cap G = \emptyset\big]. \tag{2}$$

Both are averaged over all incident tasks; an empty report ($P = \emptyset$) scores 0 on both. RCA accuracy does not exclude additional incorrect causes: any $P \supseteq G$ yields RCA accuracy 1.

RCA depth accounts for the intermediate cases. For each ground-truth cause $g \in G$, the judge assigns a per-rubric score $s_g \in \{0, 1, 2, 3\}$ on the scale of Sec. 4 (E2), with materially incorrect or misleading claims counting against s_g (App. G.1.1). The task-level RCA depth is $\frac{1}{|G|} \sum_{g \in G} s_g$, reported as a percentage of the maximum score 3 and averaged over tasks; an empty report has $s_g = 0$ for every g.

The two binary metrics therefore capture the extremes, while RCA depth evaluates intermediate diagnostic quality and penalizes incorrect or misleading claims. For example, predicting $P = \{A, B, C\}$ when $G = \{A, B\}$ yields RCA accuracy 1, hallucination rate 0, and a reduced RCA depth for the incorrect attribution to C.

Recall, precision, and hallucinated root causes. For these finer-grained metrics (Sec. 4), let $\widehat{P}$ denote the set of root causes that the agent's incident RCA report attributes to the incident, as extracted by GPT-5.4 from the list of all possible root causes (every feature flag). If the report names a feature flag but not its variant, we count it as a match whenever any variant of that flag appears in G. Then

$$\text{Recall} = \frac{|\widehat{P} \cap G|}{|G|}, \tag{3}$$

$$\text{Precision} = \begin{cases} |\widehat{P} \cap G| \,/\, |\widehat{P}| & \text{if } \widehat{P} \neq \emptyset, \\ 0 & \text{otherwise,} \end{cases} \tag{4}$$

$$\text{Hallucinated root causes} = |\widehat{P} \setminus G| = |\widehat{P}| - |\widehat{P} \cap G|. \tag{5}$$

All three are averaged over incident tasks. Recall is the fraction of rubrics whose root cause is found in the report, and assigning zero precision to empty reports penalizes agents that attribute no root cause at all.

I Supplementary Experiment Details

Our environment is built on the OpenTelemetry Astronomy Shop[3] at commit `fc77a6cbc0bf36ff52f21e4d59fa7b8a0f5d6733` of the `opentelemetry-demo` repository. The experiments of Sec. 5 were carried out using Python 3.13.13, Harbor 0.6.3, and an Ubuntu 22.04.5 LTS server with an Intel Xeon Gold 6548N 48-core processor and 192 GB RAM. We paired each model with the Terminus-2 agent harness of Merrill et al. (2026), set the temperature to 1, max tokens to 16384, and reasoning effort to "medium" for models that accept it (Claude Opus 4.7, Claude Sonnet 4.6, and GPT-5.5), and the agent timeout to 1 hour. We retried any failed trials, except those that raised `AgentTimeoutError` and `NonZeroAgentExitCodeError`. For judging, we used GPT-5.4 with reasoning effort "high," temperature 1, and max tokens 128,000.

I.1 Telemetry Outcome Classification

For the analysis in Fig. 7(c), we use GPT-5.4 nano with reasoning effort "high" and max tokens 16384 to classify the outcome of each telemetry tool as success, empty, or error using the prompt below. When the terminal output exceeds 200k characters, we truncate it; in that case the section header reads "Output (truncated)" rather than "Output (verbatim)".

Telemetry Outcome Classifier Prompt

```
You are evaluating a single telemetry tool call made by an AI agent during an SRE investigation. Classify it into
EXACTLY ONE of these labels:

• success: the query returned a non-empty, well-formed telemetry result.

• error: the query failed — HTTP 4xx/5xx, connection refused, malformed query, authentication failure, JSON parse
  error, exception trace, Prometheus "status": "error", curl: ( connection errors, etc.
```

[3] https://github.com/open-telemetry/opentelemetry-demo

```
- empty: the query succeeded but the result set is empty — e.g. "result": [], no log streams matched, no spans
  returned, an HTTP 200 with zero data points.

## Categories assigned to this call
{categories}
## Command
{command}
## Raw arguments
{arguments}
## {output_header}
{output}
## Notes

- The output may be a terminus terminal transcript that echoes prompts and contains text from sibling parallel
  commands within the same step. Classify based on whether THIS specific command appears to have returned data,
  errored, or returned an empty result.
- A 200 response that legitimately contains no data is empty, not error.
- A "status": "error" field inside an otherwise-200 JSON body is error.
```

J FULL RESULTS

Sec. 5 reports a subset of evaluation metrics for compactness. This appendix reports the full set of metrics defined in Sec. 4 – RCA depth, RCA accuracy, % hallucination, mechanism, incident time, and metrics/logs/traces citation rates (all/any) – broken out by access level: telemetry and code (App. J.1) and telemetry only (App. J.2). All values are mean $\pm$ 1 standard error in percent; the judge is GPT-5.4.

Per-rubric verdict fields. For each incident task, the judge evaluates the agent's RCA report against every plausible root cause (one rubric per active feature flag) and emits the following Boolean verdicts per rubric. The trial-level rollup we report below takes *any*-match across rubrics, i.e. the agent is credited if it matches at least one of the listed root causes.

- **Mechanism.** Did the report correctly explain how this rubric's feature flag propagates to the user-visible failure (a causal chain from flag to symptom)?
- **Inc. Time.** Did the report place the incident start within ± 10 minutes of this rubric's ground-truth `incident_time`?

Telemetry citation rates. Each rubric also enumerates one or more telemetry *clusters* of each signal type. A cluster counts as matched as follows:

- **Metrics cluster.** Did the report cite a query targeting the cluster's `metric_family` under a Why-step whose causal chain aligns with the rubric?
- **Logs cluster.** Did the report cite log evidence whose body matches the cluster's representative body *and* attribute it to the cluster's OTel service (`resource.service.name`)? Both conditions must hold.
- **Traces cluster.** Did the report cite evidence matching the cluster's service-level call chain *and* at least one listed smoking-gun message? Both conditions must hold.

Metrics/Logs/Traces (all) reports the fraction of trials where the agent matched *every* cluster of that signal type in some rubric; **(any)** reports the fraction where it matched at least one cluster. Rubrics that list no clusters of a given signal type are excluded from that signal's denominator.

J.1 TELEMETRY AND CODE

In this default setting, agents have access to both the codebase and the Grafana telemetry API. Tab. J.1 and J.2 report aggregate accuracy on incident and control tasks, and Tab. J.4a to J.4e stratify the incident-task results by prompt difficulty. Tab. J.3 stratifies the control-task results by prompt difficulty.

Table J.1: **Full RCA accuracy on incident tasks ($N = 884$)** (expanded version of Fig. 5). Agents have access to both code and telemetry.

	Outcome			Root cause		Telemetry citations					
Model	RCA Depth	RCA Accuracy	Hall.	Mechanism	Inc. Time	Metrics (all)	Metrics (any)	Logs (all)	Logs (any)	Traces (all)	Traces (any)
Claude Opus 4.7	48.5 ± 1.1	28.6 ± 1.5	12.6 ± 1.1	79.0 ± 1.4	82.6 ± 1.3	0.0 ± 0.0	65.3 ± 1.7	3.1 ± 0.6	51.4 ± 1.8	32.8 ± 1.7	48.4 ± 1.8
Claude Sonnet 4.6	46.7 ± 1.1	30.9 ± 1.6	14.8 ± 1.2	77.1 ± 1.4	80.3 ± 1.3	0.0 ± 0.0	53.3 ± 1.8	4.9 ± 0.8	61.2 ± 1.8	23.8 ± 1.6	38.0 ± 1.8
GPT-5.5	48.8 ± 1.1	24.3 ± 1.4	25.0 ± 1.5	72.4 ± 1.5	79.8 ± 1.4	0.1 ± 0.1	73.2 ± 1.6	17.6 ± 1.4	64.9 ± 1.7	43.1 ± 1.8	56.7 ± 1.8
DeepSeek-V4-Pro	19.7 ± 1.0	15.0 ± 1.2	7.2 ± 0.9	32.7 ± 1.6	31.1 ± 1.6	0.0 ± 0.0	18.1 ± 1.4	2.8 ± 0.6	21.0 ± 1.5	3.4 ± 0.7	5.6 ± 0.8
GLM-5	27.7 ± 1.0	17.6 ± 1.3	40.2 ± 1.6	46.3 ± 1.7	54.3 ± 1.7	0.0 ± 0.0	14.1 ± 1.3	2.2 ± 0.5	38.3 ± 1.7	3.3 ± 0.7	7.3 ± 0.9

Table J.2: **Full RCA accuracy on control tasks ($N = 195$)** (expanded version of Fig. 5). RCA Accuracy, mechanism, incident time, and telemetry citations are all 0% by construction on control tasks (no incident is present).

Model	RCA Depth
Claude Opus 4.7	84.1 ± 2.6
Claude Sonnet 4.6	57.9 ± 3.5
GPT-5.5	73.3 ± 3.2
DeepSeek-V4-Pro	77.9 ± 3.0
GLM-5	44.6 ± 3.6

Table J.3: **RCA depth on control tasks by prompt difficulty (code + telemetry), $n = 65$ per cell.** All other metrics are 0% by construction on control tasks (no incident is present).

Model	Hard	Medium	Easy
Claude Opus 4.7	80.0 ± 5.0	83.1 ± 4.7	89.2 ± 3.9
Claude Sonnet 4.6	50.8 ± 6.2	61.5 ± 6.1	61.5 ± 6.1
GPT-5.5	60.0 ± 6.1	80.0 ± 5.0	80.0 ± 5.0
DeepSeek-V4-Pro	70.8 ± 5.7	83.1 ± 4.7	80.0 ± 5.0
GLM-5	27.7 ± 5.6	50.8 ± 6.2	55.4 ± 6.2

Table J.4: **Full RCA accuracy by prompt difficulty (code + telemetry).** Per-model breakdown of incident-task accuracy stratified by prompt difficulty. All values are mean ± 1 standard error in percent; the judge is GPT-5.4.

(a) Claude Opus 4.7

		Outcome			Root cause		Telemetry citations					
Difficulty	Avg GT Events	RCA Depth	RCA Accuracy	Hall.	Mechanism	Inc. Time	Metrics (all)	Metrics (any)	Logs (all)	Logs (any)	Traces (all)	Traces (any)
Hard ($n = 280$)	4.41 ± 0.11	37.6 ± 1.6	10.0 ± 1.8	12.1 ± 2.0	81.1 ± 2.3	87.5 ± 2.0	0.0 ± 0.0	63.5 ± 2.9	3.0 ± 1.0	42.1 ± 3.0	30.3 ± 2.8	47.6 ± 3.0
Medium ($n = 316$)	2.84 ± 0.09	47.9 ± 1.7	24.7 ± 2.4	13.0 ± 1.9	76.9 ± 2.4	82.6 ± 2.1	0.0 ± 0.0	65.3 ± 2.9	2.6 ± 1.0	58.2 ± 3.0	29.7 ± 2.9	43.0 ± 3.1
Easy ($n = 288$)	2.00 ± 0.06	59.9 ± 2.1	51.0 ± 3.0	12.5 ± 2.0	79.2 ± 2.4	77.8 ± 2.5	0.0 ± 0.0	67.5 ± 3.1	3.8 ± 1.3	54.3 ± 3.3	39.3 ± 3.2	55.5 ± 3.3

(b) Claude Sonnet 4.6

		Outcome			Root cause		Telemetry citations					
Difficulty	Avg GT Events	RCA Depth	RCA Accuracy	Hall.	Mechanism	Inc. Time	Metrics (all)	Metrics (any)	Logs (all)	Logs (any)	Traces (all)	Traces (any)
Hard ($n = 280$)	4.41 ± 0.11	31.7 ± 1.4	8.6 ± 1.7	13.6 ± 2.1	72.1 ± 2.7	82.5 ± 2.3	0.0 ± 0.0	49.8 ± 3.0	3.0 ± 1.0	54.2 ± 3.0	24.0 ± 2.6	38.7 ± 3.0
Medium ($n = 316$)	2.84 ± 0.09	45.7 ± 1.7	25.3 ± 2.4	16.8 ± 2.1	74.1 ± 2.5	81.6 ± 2.2	0.0 ± 0.0	57.5 ± 3.0	5.2 ± 1.4	67.2 ± 2.9	18.8 ± 2.4	28.5 ± 2.8
Easy ($n = 288$)	2.00 ± 0.06	62.3 ± 2.0	58.7 ± 2.9	13.9 ± 2.0	85.4 ± 2.1	76.7 ± 2.5	0.0 ± 0.0	52.6 ± 3.3	6.8 ± 1.7	62.4 ± 3.2	29.3 ± 3.0	47.6 ± 3.3

(c) GPT-5.5

		Outcome			Root cause		Telemetry citations					
Difficulty	Avg GT Events	RCA Depth	RCA Accuracy	Hall.	Mechanism	Inc. Time	Metrics (all)	Metrics (any)	Logs (all)	Logs (any)	Traces (all)	Traces (any)
Hard ($n = 280$)	4.41 ± 0.11	38.9 ± 1.5	8.6 ± 1.7	22.5 ± 2.5	77.1 ± 2.5	85.0 ± 2.1	0.0 ± 0.0	72.0 ± 2.7	21.4 ± 2.5	66.1 ± 2.9	43.5 ± 3.0	54.6 ± 3.0
Medium ($n = 316$)	2.84 ± 0.09	49.0 ± 1.8	25.0 ± 2.4	23.7 ± 2.4	69.9 ± 2.6	80.1 ± 2.3	0.0 ± 0.0	76.9 ± 2.6	14.2 ± 2.1	63.4 ± 2.9	43.8 ± 3.1	56.2 ± 3.1
Easy ($n = 288$)	2.00 ± 0.06	58.3 ± 2.1	38.9 ± 2.9	28.8 ± 2.7	70.5 ± 2.7	74.3 ± 2.6	0.4 ± 0.4	70.5 ± 3.0	17.1 ± 2.5	65.4 ± 3.1	41.9 ± 3.3	59.8 ± 3.2

(d) DeepSeek-V4-Pro

		Outcome			Root cause		Telemetry citations					
Difficulty	Avg GT Events	RCA Depth	RCA Accuracy	Hall.	Mechanism	Inc. Time	Metrics (all)	Metrics (any)	Logs (all)	Logs (any)	Traces (all)	Traces (any)
Hard ($n = 280$)	4.41 ± 0.11	12.0 ± 1.3	5.4 ± 1.3	12.1 ± 2.0	27.5 ± 2.7	31.1 ± 2.8	0.0 ± 0.0	15.9 ± 2.2	3.0 ± 1.0	17.3 ± 2.3	2.2 ± 0.9	3.3 ± 1.1
Medium ($n = 316$)	2.84 ± 0.09	18.0 ± 1.6	14.9 ± 2.0	6.0 ± 1.3	29.4 ± 2.6	31.3 ± 2.6	0.0 ± 0.0	19.4 ± 2.4	2.2 ± 0.9	20.9 ± 2.5	1.6 ± 0.8	3.1 ± 1.1
Easy ($n = 288$)	2.00 ± 0.06	29.2 ± 2.2	24.7 ± 2.5	3.8 ± 1.1	41.3 ± 2.9	30.9 ± 2.7	0.0 ± 0.0	19.2 ± 2.6	3.4 ± 1.2	25.2 ± 2.8	7.0 ± 1.7	10.9 ± 2.1

(e) GLM-5

		Outcome			Root cause		Telemetry citations					
Difficulty	Avg GT Events	RCA Depth	RCA Accuracy	Hall.	Mechanism	Inc. Time	Metrics (all)	Metrics (any)	Logs (all)	Logs (any)	Traces (all)	Traces (any)
Hard ($n = 280$)	4.41 ± 0.11	10.5 ± 1.0	1.1 ± 0.6	66.1 ± 2.8	25.4 ± 2.6	48.9 ± 3.0	0.0 ± 0.0	6.6 ± 1.5	1.8 ± 0.8	29.9 ± 2.8	1.5 ± 0.7	4.4 ± 1.3
Medium ($n = 316$)	2.84 ± 0.09	27.2 ± 1.6	16.8 ± 2.1	34.2 ± 2.7	46.5 ± 2.8	57.6 ± 2.8	0.0 ± 0.0	17.5 ± 2.3	3.0 ± 1.0	45.9 ± 3.0	3.1 ± 1.1	4.3 ± 1.3
Easy ($n = 288$)	2.00 ± 0.06	45.1 ± 2.0	34.7 ± 2.8	21.5 ± 2.4	66.3 ± 2.8	55.9 ± 2.9	0.0 ± 0.0	18.8 ± 2.6	1.7 ± 0.8	39.3 ± 3.2	5.7 ± 1.5	14.0 ± 2.3

J.2 TELEMETRY ONLY

In this setting, agents have access to the Grafana telemetry API but not the codebase. Tab. J.5 and J.6 report aggregate accuracy on incident and control tasks, respectively, and Tab. J.8a to J.8e stratify the incident-task results by prompt difficulty. Tab. J.7 stratifies the control-task results by prompt difficulty.

Table J.5: **Full RCA accuracy on incident tasks, telemetry-only access ($N = 884$)** (expanded version of Fig. 6). Agents have access to telemetry but not the codebase.

	Outcome			Root cause		Telemetry citations					
Model	RCA Depth	RCA Accuracy	Hall.	Mechanism	Inc. Time	Metrics (all)	Metrics (any)	Logs (all)	Logs (any)	Traces (all)	Traces (any)
Claude Opus 4.7	38.4 ± 1.0	19.3 ± 1.3	31.3 ± 1.6	59.8 ± 1.6	79.9 ± 1.3	0.4 ± 0.2	69.5 ± 1.7	4.9 ± 0.8	48.8 ± 1.8	30.3 ± 1.7	40.3 ± 1.8
Claude Sonnet 4.6	33.4 ± 0.9	21.4 ± 1.4	29.0 ± 1.5	47.3 ± 1.7	76.8 ± 1.4	0.0 ± 0.0	57.4 ± 1.8	7.1 ± 0.9	45.0 ± 1.8	14.3 ± 1.3	22.4 ± 1.5
GPT-5.5	34.2 ± 0.9	8.6 ± 0.9	56.0 ± 1.7	42.2 ± 1.7	72.4 ± 1.5	0.1 ± 0.1	63.8 ± 1.7	16.4 ± 1.3	53.3 ± 1.8	34.9 ± 1.7	43.7 ± 1.8
DeepSeek-V4-Pro	8.3 ± 0.6	5.1 ± 0.7	13.0 ± 1.1	12.4 ± 1.1	22.4 ± 1.4	0.0 ± 0.0	14.6 ± 1.3	1.9 ± 0.5	11.8 ± 1.2	3.0 ± 0.6	4.8 ± 0.8
GLM-5	14.1 ± 0.7	6.8 ± 0.8	73.1 ± 1.5	14.8 ± 1.2	50.7 ± 1.7	0.0 ± 0.0	20.1 ± 1.4	2.8 ± 0.6	28.1 ± 1.6	2.9 ± 0.6	4.2 ± 0.7

Table J.6: **Full RCA accuracy on control tasks, telemetry-only access ($N = 195$).** RCA Accuracy, mechanism, incident time, and telemetry citations are all 0% by construction on control tasks (no incident is present).

Model	RCA Depth
Claude Opus 4.7	69.2 ± 3.3
Claude Sonnet 4.6	48.7 ± 3.6
GPT-5.5	72.3 ± 3.2
DeepSeek-V4-Pro	81.0 ± 2.8
GLM-5	27.2 ± 3.2

Table J.7: **RCA depth on control tasks by prompt difficulty, telemetry-only access ($n = 65$ per cell).** All other metrics are 0% by construction on control tasks (no incident is present).

Model	Hard	Medium	Easy
Claude Opus 4.7	47.7 ± 6.2	81.5 ± 4.8	78.5 ± 5.1
Claude Sonnet 4.6	38.5 ± 6.1	52.3 ± 6.2	55.4 ± 6.2
GPT-5.5	58.5 ± 6.2	72.3 ± 5.6	86.2 ± 4.3
DeepSeek-V4-Pro	67.7 ± 5.8	87.7 ± 4.1	87.7 ± 4.1
GLM-5	16.9 ± 4.7	27.7 ± 5.6	36.9 ± 6.0

Table J.8: **Full RCA accuracy by prompt difficulty (telemetry only).** Per-model breakdown of incident-task accuracy stratified by prompt difficulty, with telemetry access but no codebase access. All values are mean ± 1 standard error in percent; the judge is GPT-5.4.

(a) Claude Opus 4.7

		Outcome			Root cause		Telemetry citations					
Difficulty	Avg GT Events	RCA Depth	RCA Accuracy	Hall.	Mechanism	Inc. Time	Metrics (all)	Metrics (any)	Logs (all)	Logs (any)	Traces (all)	Traces (any)
Hard ($n = 280$)	4.41 ± 0.11	33.2 ± 1.3	6.8 ± 1.5	27.9 ± 2.7	68.9 ± 2.8	86.4 ± 2.1	0.4 ± 0.4	74.2 ± 2.7	3.3 ± 1.1	46.5 ± 3.0	32.1 ± 2.8	40.6 ± 3.0
Medium ($n = 316$)	2.84 ± 0.09	38.0 ± 1.7	18.7 ± 2.2	31.3 ± 2.6	56.6 ± 2.8	82.0 ± 2.2	0.4 ± 0.4	70.9 ± 2.8	4.1 ± 1.2	50.0 ± 3.1	29.3 ± 2.9	37.1 ± 3.0
Easy ($n = 288$)	2.00 ± 0.06	43.8 ± 2.0	32.3 ± 2.8	34.7 ± 2.8	54.5 ± 2.9	71.2 ± 2.7	0.4 ± 0.4	62.4 ± 3.2	7.7 ± 1.7	50.0 ± 3.3	29.3 ± 3.0	43.7 ± 3.3

(b) Claude Sonnet 4.6

		Outcome			Root cause		Telemetry citations					
Difficulty	Avg GT Events	RCA Depth	RCA Accuracy	Hall.	Mechanism	Inc. Time	Metrics (all)	Metrics (any)	Logs (all)	Logs (any)	Traces (all)	Traces (any)
Hard ($n = 280$)	4.41 ± 0.11	26.9 ± 1.3	6.1 ± 1.4	27.1 ± 2.7	46.8 ± 3.0	83.6 ± 2.2	0.0 ± 0.0	55.7 ± 3.0	5.5 ± 1.4	42.8 ± 3.0	13.7 ± 2.1	19.9 ± 2.4
Medium ($n = 316$)	2.84 ± 0.09	31.5 ± 1.4	19.3 ± 2.2	28.2 ± 2.5	43.4 ± 2.8	77.5 ± 2.4	0.0 ± 0.0	61.2 ± 3.0	5.6 ± 1.4	48.1 ± 3.1	13.7 ± 2.2	19.9 ± 2.5
Easy ($n = 288$)	2.00 ± 0.06	41.8 ± 1.9	38.5 ± 2.9	31.6 ± 2.7	52.1 ± 2.9	69.4 ± 2.7	0.0 ± 0.0	55.1 ± 3.3	10.7 ± 2.0	44.0 ± 3.3	15.7 ± 2.4	27.9 ± 3.0

(c) GPT-5.5

		Outcome			Root cause		Telemetry citations					
Difficulty	Avg GT Events	RCA Depth	RCA Accuracy	Hall.	Mechanism	Inc. Time	Metrics (all)	Metrics (any)	Logs (all)	Logs (any)	Traces (all)	Traces (any)
Hard ($n = 280$)	4.41 ± 0.11	29.3 ± 1.2	1.4 ± 0.7	54.6 ± 3.0	48.2 ± 3.0	81.1 ± 2.3	0.0 ± 0.0	64.2 ± 2.9	19.9 ± 2.4	56.8 ± 3.0	36.9 ± 2.9	43.5 ± 3.0
Medium ($n = 316$)	2.84 ± 0.09	33.3 ± 1.5	7.3 ± 1.5	54.4 ± 2.8	39.2 ± 2.8	71.8 ± 2.5	0.4 ± 0.4	64.2 ± 2.9	11.9 ± 2.0	46.3 ± 3.1	33.2 ± 2.9	42.2 ± 3.1
Easy ($n = 288$)	2.00 ± 0.06	39.8 ± 1.9	17.0 ± 2.2	59.0 ± 2.9	39.6 ± 2.9	64.6 ± 2.8	0.0 ± 0.0	62.8 ± 3.2	17.5 ± 2.5	57.3 ± 3.2	34.5 ± 3.1	45.4 ± 3.3

(d) DeepSeek-V4-Pro

		Outcome			Root cause		Telemetry citations					
Difficulty	Avg GT Events	RCA Depth	RCA Accuracy	Hall.	Mechanism	Inc. Time	Metrics (all)	Metrics (any)	Logs (all)	Logs (any)	Traces (all)	Traces (any)
Hard ($n = 280$)	4.41 ± 0.11	5.2 ± 0.8	0.0 ± 0.0	16.4 ± 2.2	9.3 ± 1.7	18.6 ± 2.3	0.0 ± 0.0	11.8 ± 2.0	1.5 ± 0.7	8.9 ± 1.7	0.7 ± 0.5	2.6 ± 1.0
Medium ($n = 316$)	2.84 ± 0.09	10.2 ± 1.1	7.0 ± 1.4	10.1 ± 1.7	15.2 ± 2.0	28.2 ± 2.5	0.0 ± 0.0	19.0 ± 2.4	1.5 ± 0.7	13.4 ± 2.1	4.3 ± 1.3	5.5 ± 1.4
Easy ($n = 288$)	2.00 ± 0.06	9.3 ± 1.2	8.0 ± 1.6	12.8 ± 2.0	12.5 ± 2.0	19.8 ± 2.4	0.0 ± 0.0	12.8 ± 2.2	3.0 ± 1.1	13.2 ± 2.2	4.4 ± 1.4	6.6 ± 1.6

(e) GLM-5

		Outcome			Root cause		Telemetry citations					
Difficulty	Avg GT Events	RCA Depth	RCA Accuracy	Hall.	Mechanism	Inc. Time	Metrics (all)	Metrics (any)	Logs (all)	Logs (any)	Traces (all)	Traces (any)
Hard ($n = 280$)	4.41 ± 0.11	7.5 ± 0.7	0.7 ± 0.5	82.9 ± 2.3	12.1 ± 2.0	55.7 ± 3.0	0.0 ± 0.0	12.2 ± 2.0	2.2 ± 0.9	28.0 ± 2.7	2.2 ± 0.9	3.3 ± 1.1
Medium ($n = 316$)	2.84 ± 0.09	13.6 ± 1.0	6.6 ± 1.4	72.2 ± 2.5	10.1 ± 1.7	53.2 ± 2.8	0.0 ± 0.0	22.4 ± 2.6	3.7 ± 1.2	31.0 ± 2.8	1.6 ± 0.8	2.0 ± 0.9
Easy ($n = 288$)	2.00 ± 0.06	20.9 ± 1.4	12.8 ± 2.0	64.6 ± 2.8	22.6 ± 2.5	43.1 ± 2.9	0.0 ± 0.0	26.5 ± 2.9	2.6 ± 1.0	24.8 ± 2.8	5.2 ± 1.5	7.9 ± 1.8

J.3 Token Usage by Access Level

Tab. J.9 reports the mean total tokens (in millions) consumed per trial under each access level.

Table J.9: **Mean total tokens per trial (M) by access level.** Same trial counts as Fig. 5 and 6.

	Code + Tel.		Tel. only	
Model	Incident ($N = 884$)	Control ($N = 195$)	Incident ($N = 884$)	Control ($N = 195$)
Claude Opus 4.7	1.67	2.60	2.01	2.27
Claude Sonnet 4.6	1.43	1.94	1.46	1.57
GPT-5.5	0.54	0.51	0.50	0.44
DeepSeek-V4-Pro	2.47	2.66	2.14	2.12
GLM-5	1.81	1.91	1.54	1.52

J.4 Command Categories

Fig. J.1 shows the command-category transition diagrams for Claude Opus 4.7 and GLM-5, and Fig. J.2 shows them for the remaining three models (Claude Sonnet 4.6, GPT-5.5, and DeepSeek-V4-Pro).

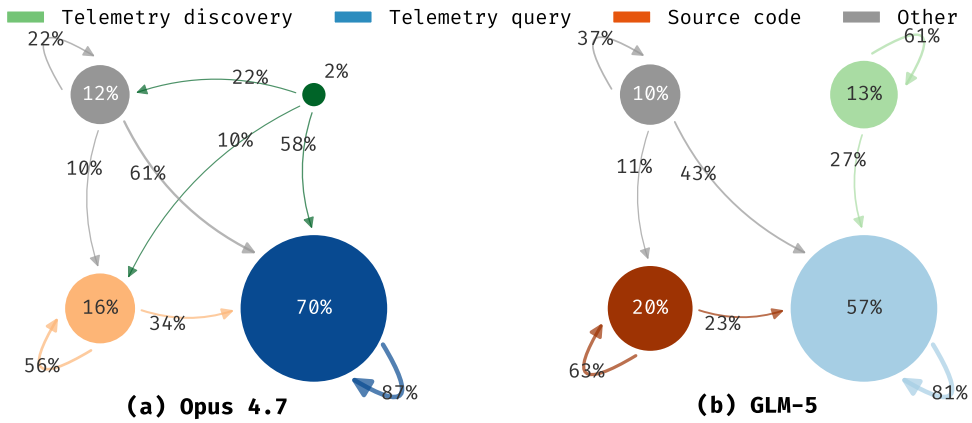


Figure J.1: **Agent behavior during investigation.** Node area is proportional to the marginal share of commands in that category. Directed edges are transition probabilities $P(\text{next} \mid \text{prev})$ between consecutive commands. For readability, we omit edges whose transition probability $P(\text{next} \mid \text{prev}) \leq 0.1$. Claude Sonnet 4.6, GPT-5.5, and DeepSeek-V4-Pro are shown in Fig. J.2.

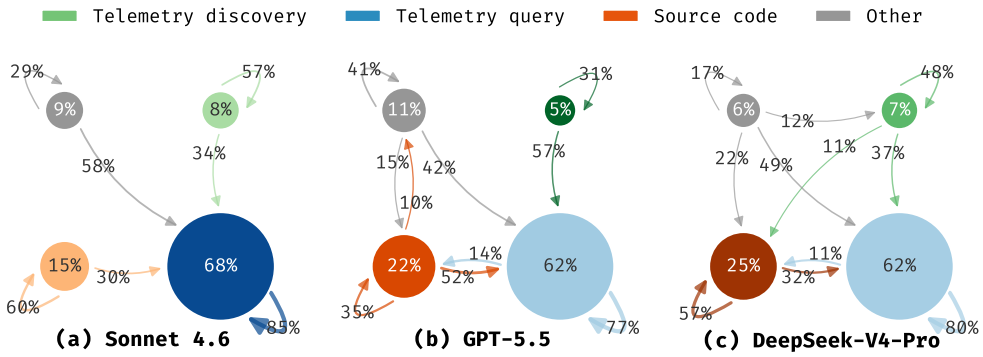


Figure J.2: **Command-category transition diagrams for the remaining models.** Companion to Fig. J.1 for the three models not shown there; see that caption for the full description of nodes, edges, and the joint-flow threshold. Terminus-2 at medium effort.

K FULL CASE-STUDY RCA REPORTS

For completeness, we reproduce the complete, unabridged incident RCA reports excerpted in Fig. 4. Both reports are for the same Medium task (user-facing issue "shoppers cannot use their cart," reported at 1:14 PM ET, TTD = 15 min, ground truth `productCatalogFailure` enabled at 1:04 PM ET). GLM-5 (App. K.1) misattributes the incident to a Playwright browser crash in the load-generator and is scored 0 / 3; Claude Opus 4.7 (App. K.2) correctly traces the failed checkouts through `ProductCatalogService/GetProduct` to the enabled `productCatalogFailure` flag and is scored 3 / 3. Both the human annotator and the GPT-5.4 judge agree on both scores.

K.1 GLM-5 — HUMAN 0 / 3, GPT-5.4 JUDGE 0 / 3

GLM-5 — RCA Report

```
# Incident RCA Report: Cart Service Unavailable

## Summary

Between April 22, 10:13 UTC and April 24, 17:25+ UTC, browser-based simulated shoppers could not access cart
      functionality due to a Playwright browser crash in the load-generator service. Over 94,000 browser test
      errors occurred during the 55+ hour incident window. The cart service itself was operational, but the
      browser-based load tests that simulate real user cart interactions failed continuously. Additionally,
      intermittent Redis connection errors affected the cart service between April 20 and April 23, compounding
      the impact.

## Timeline

- **Apr 20, 21:12 UTC** -- cart service first logged "Wasn't able to connect to redis" errors (Redis
      connectivity issues began)
- **Apr 22, 10:13 UTC** -- Playwright browser crashed in load-generator with "Target page, context or browser
      has been closed" error; browser-based cart tests began failing
- **Apr 22-24** -- Browser errors continued at rate of ~30,000-40,000 per day
- **Apr 23, 22:42 UTC** -- Last Redis connection error from cart service (Redis issue resolved)
- **Apr 24, 13:14 UTC (17:14 UTC)** -- Incident reported: "shoppers cannot use their cart"
- **Apr 24, 17:25 UTC** -- Cart service operational with no errors; browser tests still failing

## 5 Whys

**Problem statement:**
Browser-based simulated shoppers could not access cart functionality between April 22, 10:13 UTC and April 24,
      17:25+ UTC, with over 94,000 test failures. The incident was detected through load-generator error logs.

**Why 1:** Why could shoppers not use their cart?
-> The Playwright browser used by the load-generator crashed, preventing browser-based cart tests from executing
      .

Evidence: Logs from load-generator service show continuous ERROR-level entries starting April 22, 10:13:58 UTC:
```
Browser.new_context: Target page, context or browser has been closed
playwright._impl._errors.TargetClosedError: Browser.new_context: Target page, context or browser has been closed
```
Lucene query: `service:load-generator AND severity:Error AND body:"browser"`

**Why 2:** Why did the Playwright browser crash?
-> The browser process was terminated or became unresponsive, causing all subsequent browser context creation
      attempts to fail.

Evidence: Error count by day shows 23,723 errors on April 22, 40,271 on April 23, and 31,005 on April 24.
PromQL: `count_over_time({service="load-generator", severity="Error"}[1d])`

**Why 3:** Why did the browser process terminate?
-> Investigation of container metrics and logs did not reveal an explicit OOM kill or crash event. The valkey-
      cart container has only 20M memory limit which could cause resource contention, but no direct evidence of
      memory pressure was found in available telemetry.

Evidence: No OOM-related logs found. Container memory metrics showed stable ~89% usage.
PromQL: `container_memory_percent_ratio`

**Why 4:** Why wasn't the browser automatically restarted?
-> The PlaywrightUser class in locust_plugins does not implement automatic browser recovery after the browser
      process terminates unexpectedly.
```

```
Evidence: Source code at `/app/opentelemetry-demo/src/load-generator/locustfile.py` shows WebsiteBrowserUser
      class inherits from PlaywrightUser with `headless = True` but no error recovery or browser restart logic.

**Root cause:**
The Playwright browser integration in the load-generator lacks resilience to browser process termination. When
      the browser crashes or is killed, the load-generator continues attempting to use the dead browser instance
       without recovery, causing indefinite browser-based test failures.

**Unexplored branches:**
- The Redis connectivity issues (April 20-23) were resolved before the incident was reported and are not the
      primary cause of the April 24 cart issue.
- The valkey-cart 20M memory limit could be a contributing factor to intermittent Redis connectivity, but no
      direct evidence links it to the browser crash.

## Remediation

| Action | Targets | Type |
|--------|---------|------|
| Add browser health check and automatic restart logic in PlaywrightUser subclass | Why 4 | Root cause fix |
| Implement circuit breaker pattern for browser operations with automatic recovery | Why 2 | Propagation fix |
| Add alerting for browser-based test failure rate exceeding threshold | Why 1 | Detection fix |
| Increase valkey-cart memory limit from 20M to at least 64M | Unexplored | Root cause fix |
| Add monitoring for Redis connection errors in cart service | Unexplored | Detection fix |
```

K.2 Claude Opus 4.7 — Human 3 / 3, GPT-5.4 judge 3 / 3

Claude Opus 4.7 — RCA Report

```
# Incident RCA: Shoppers Unable to Use Cart

## Summary

Between approximately 17:06 UTC and the end of the observed telemetry window (~17:25 UTC) on April 24, 2026,
      shoppers attempting to view product OLJCESPC7Z, place orders containing that product, or load related
      recommendations received HTTP 500 errors, due to the `productCatalogFailure` feature flag being enabled.
      Cart-impact manifested at checkout (`POST /api/checkout`): `oteldemo.CheckoutService/PlaceOrder` failed
      with `failed to prepare order: failed to get product #"OLJCESPC7Z"`, preventing affected shoppers from
      completing purchases from their cart. The full causal chain is detailed in the 5 Whys section below.

## Timeline

- 17:06 UTC -- product-catalog `GetProduct` error counter begins climbing (581 -> 583 between 17:05:55 and
      17:06:10 UTC); `productCatalogFailure` flag now causing failures for product OLJCESPC7Z.
- 17:07 UTC -- sustained product-catalog error rate visible (~0.03-0.07 err/s) on `oteldemo.
      ProductCatalogService/GetProduct`.
- 17:08-17:25 UTC -- frontend error rate climbs; failed spans observed for `GET /api/products/{productId}` (
      OLJCESPC7Z), `GET /api/recommendations`, and `POST /api/checkout`.
- 17:14 UTC -- incident reported by shoppers ("cannot use their cart").
- 17:15 UTC -- first checkout 500 increment recorded for `/api/checkout` (counter 547 -> 548 -> 549 -> 550
      across the next ~10 minutes); checkout `PlaceOrder` spans show `failed to get product #"OLJCESPC7Z"`.
- 17:25 UTC -- end of observed telemetry; failure mode still active.

## 5 Whys

Problem statement:
Starting at 17:06 UTC on 2026-04-24, shoppers experienced HTTP 500 responses on `POST /api/checkout` (the cart-
      checkout flow) and on product/recommendation pages featuring product OLJCESPC7Z. Detected via Prometheus
      span-metrics (`traces_span_metrics_calls_total{service_name="product-catalog",status_code="
      STATUS_CODE_ERROR"}`) and corroborated by error spans in Jaeger.

Why 1: Why did checkout return 500 to shoppers? -> CheckoutService/PlaceOrder failed when preparing the order
      because the product lookup for OLJCESPC7Z returned an error.
Evidence (traces): `frontend POST /api/checkout` -> `oteldemo.CheckoutService/PlaceOrder` -> `oteldemo.
      ProductCatalogService/GetProduct` (error). Span status description on the checkout span: `failed to
      prepare order: failed to get product #"OLJCESPC7Z"`. Trace IDs: 0901ae1036f7b0b9975a4c9af1c23fd5, 5
      b6f061d4a0ad7c85f49468a620c12cc.

Why 2: Why did `ProductCatalogService/GetProduct` fail for OLJCESPC7Z? -> product-catalog deliberately returned
      `Internal` gRPC error because of an enabled fault-injection feature flag.
Evidence (traces): Span `oteldemo.ProductCatalogService/GetProduct` carries `otel.status_description="Error:
      Product Catalog Fail Feature Flag Enabled"`, `rpc.grpc.status_code=13`, `app.product.id=OLJCESPC7Z`.
```

```
Why 3: Why did product-catalog raise that error? -> The code path in `src/product-catalog/main.go` checks `
      productCatalogFailure` via OpenFeature/flagd and, when true (only for product id `OLJCESPC7Z`), returns `
      codes.Internal` with the message above. The metric counter increment from 581->583 between 17:05:55 and
      17:06:10 UTC pinpoints the moment the flag's effective value flipped to `on`.
Evidence (metrics): `sum(traces_span_metrics_calls_total{service_name="product-catalog",status_code="
      STATUS_CODE_ERROR",span_name="oteldemo.ProductCatalogService/GetProduct"})` was 581 at 17:05:55 UTC and
      583 at 17:06:10 UTC, then climbed steadily (594 by 17:11:40, 616 by 17:23:00). PromQL: `sum(rate(
      traces_span_metrics_calls_total{service_name="product-catalog",status_code="STATUS_CODE_ERROR"}[2m]))`.

Why 4: Why was `productCatalogFailure` set to `on` in production? -> The flag is an intentional chaos/demo
      toggle (`Fail product catalog service on a specific product`) shipped enabled in flagd's config. It was
      switched from its default `off` variant to `on` at ~17:06 UTC. There is no telemetry of who/what made the
       change (flagd change-event logs are not present in OpenSearch for this window), so the proximate trigger
      is a configuration change to the flagd ruleset that was not gated by a safety review or staged rollout.
Evidence (logs/config): Flag definition lives in `src/flagd/demo.flagd.json` (`productCatalogFailure`,
      defaultVariant `off`). Lucene to retrieve flagd activity in window (returned no hits, indicating the
      change was not logged to OpenSearch): `resource.service.name:"flagd" AND observedTimestamp:[2026-04-24T16
      :50:00Z TO 2026-04-24T17:15:00Z]`.

Root cause:
A fault-injection feature flag (`productCatalogFailure`) was enabled in production flagd configuration at ~17:06
       UTC, causing `ProductCatalogService/GetProduct` to return `Internal` errors for product OLJCESPC7Z,
      which propagated as failed checkouts for any shopper whose cart contained that item.

Unexplored branches:
- Recommendation and product-detail page errors (`GET /api/recommendations`, `GET /api/products/{productId}`)
      share the same root cause (fan-out to product-catalog) and were not pursued separately.
- A small number of pre-existing `Product Not Found` errors (rpc status 5) for empty product ids exist in
      baseline traffic and predate the incident; not part of this chain.
- The `cartFailure` feature flag was evaluated continuously by the cart service but no DELETE /api/cart traffic
      was present, so it did not contribute.
- Flagd does not appear to emit change-audit logs to OpenSearch in this window; identifying the operator/process
       that flipped the flag would require flagd audit telemetry that is not in the provided sources.

## Remediation

| Action | Targets | Type |
|--------|---------|------|
| Disable `productCatalogFailure` flag (set defaultVariant back to `off` and confirm via flagd) | Why 3 | Root
      cause fix |
| Gate enabling of fault-injection flags in production behind a maintenance-window/approval workflow; restrict
      who can flip chaos flags | Why 4 | Root cause fix |
| Ensure flagd emits structured change-audit logs (who/when/what flag/variant) to OpenSearch | Why 4 | Detection
       fix |
| Add a Prometheus alert on `sum(rate(traces_span_metrics_calls_total{service_name="product-catalog",status_code
      ="STATUS_CODE_ERROR"}[5m])) > 0` and on checkout 5xx rate so flag-induced failures page within minutes
      rather than ~8 minutes | Why 1, Why 3 | Detection fix |
| In CheckoutService, treat per-product GetProduct failures as a partial/degraded response (e.g., return a clear
       per-item error to the client and avoid failing the whole order if business rules permit) | Why 1 |
      Propagation fix |
| In product-catalog, if a chaos flag is enabled, return a non-Internal status (e.g., FailedPrecondition) and
      surface a banner so dependent services and operators recognize injected faults | Why 2 | Propagation fix |
```

L Additional Experiments

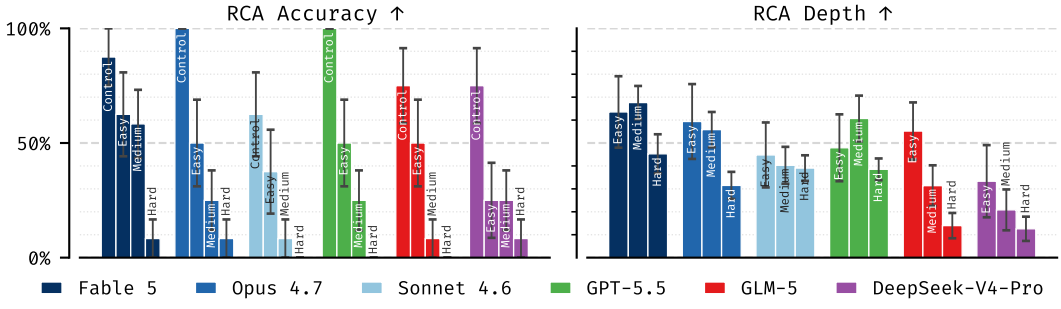


Figure L.1: **Expanded model set on ORCA-bench Verified.** We report the RCA accuracy and depth of Claude Fable 5 compared to the other models. Error bars display ± 1 standard error across the 8 control, 8 Easy, 12 Medium, 12 Hard tasks.

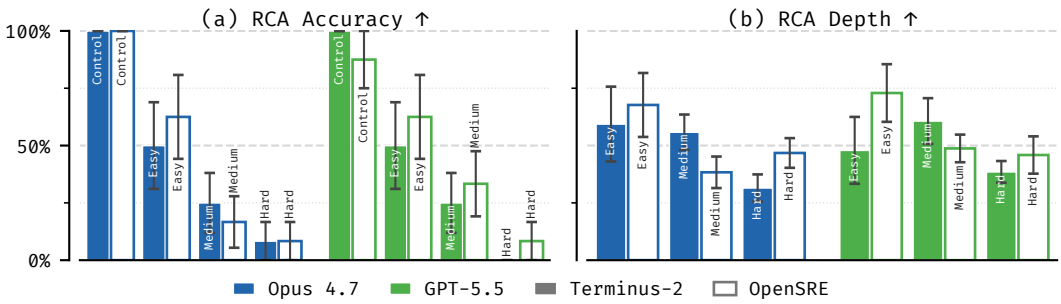


Figure L.2: **RCA quality across harnesses on ORCA-bench Verified.** (a) RCA accuracy, including controls; (b) incident RCA depth as a percentage of 3. Filled/outlined bars denote Terminus-2/OpenSRE; error bars show ±1 SE.

L.1 Comparison of Agent Harnesses

We compare Claude Opus 4.7 and GPT-5.5 under Terminus-2 and OpenSRE (Tracer-Cloud, 2026) on the same 40 tasks in ORCA-bench Verified (32 incidents and eight controls). Both generate text with temperature 1, 16,384-token output limit, medium reasoning effort, and the same verifier, reward formula, and GPT-5.4 judge. OpenSRE has specialized tools, such as `query_elasticsearch_logs` and `query_opensearch_analytics`. Tab. L.1 summarizes how the harnesses guide and constrain investigation.

Table L.1: **How the harnesses shape investigation in our evaluation.**

Aspect	Terminus-2	OpenSRE
System Prompt	General instructions for completing a task through a terminal.	Broader operational guidance, including planning and coordination.
Investigation steps	Executes one batch of commands, then inspects the result before continuing.	Can issue several tool calls in parallel within a round.
Evidence visibility	Reads the terminal screen; must page or redirect long outputs to inspect them.	Receives each command's captured output directly in the conversation.
Investigation memory	Condenses earlier investigation into a summary when context fills.	Removes selected earlier tool interactions without summarizing them.
Ending a trial	Continues until the agent declares completion or the time limit is reached.	Can also stop after too many rounds or repeated calls, potentially before a report is written.

Fig. L.2 shows higher RCA depth under OpenSRE on Easy and Hard tasks for both models, while Terminus-2 scores higher on Medium. Aggregate incident RCA accuracy is 25.0% under both harnesses for Claude Opus 4.7, and 21.9% versus 31.2% for GPT-5.5 under Terminus-2 and OpenSRE, respectively. Both solve all eight Claude Opus 4.7 controls; GPT-5.5 solves eight under Terminus-2 and seven under OpenSRE.

Fig. L.3 shows identical hallucination rates across harnesses within each model (12.5% for Claude Opus 4.7; 21.9% for GPT-5.5), but differences in evidence citation. For example, GPT-5.5 trace any-match is higher under OpenSRE (72.4% versus 48.3%), whereas Claude Opus 4.7 metric any-match is higher under Terminus-2 (73.3% versus 60.0%). These results show variation by difficulty and evidence type without a consistent advantage for either configuration.

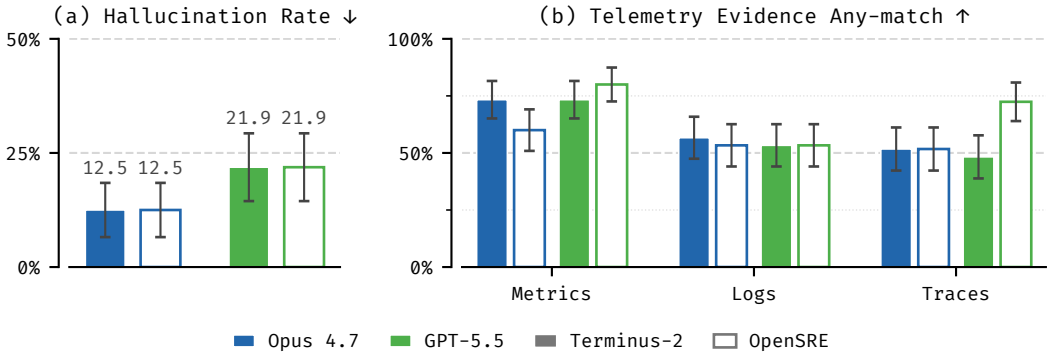
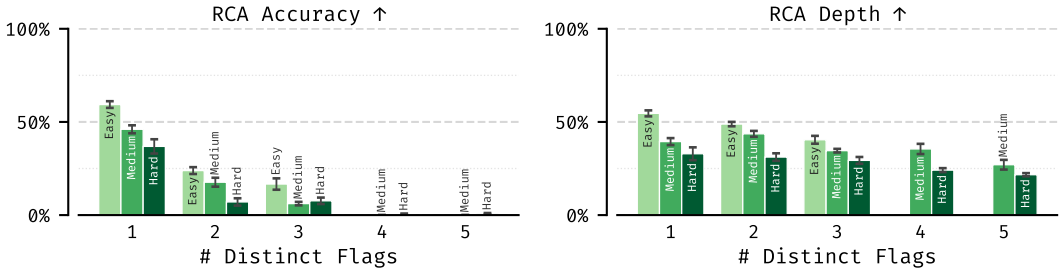


Figure L.3: **Hallucination and supporting evidence across harnesses on ORCA-bench Verified.** (a) Incident hallucination rate; (b) reports citing at least one ground-truth symptom cluster per signal type (any-match). Filled/outlined bars denote Terminus-2/OpenSRE; error bars show ± 1 SE.

L.2 Additional RCA Analyses



(a) RCA accuracy (b) RCA depth

Figure L.4: **RCA quality by number of ground-truth root causes and issue specificity.** Tasks are grouped by the number of distinct feature flags active during the incident; within each group, bars show Easy, Medium, and Hard tasks, pooled across all models. Holding the number of root causes fixed, both RCA accuracy and depth decrease as issue specificity goes from Easy to Hard. Easy tasks never have more than three distinct flags. Error bars display ± 1 standard error.

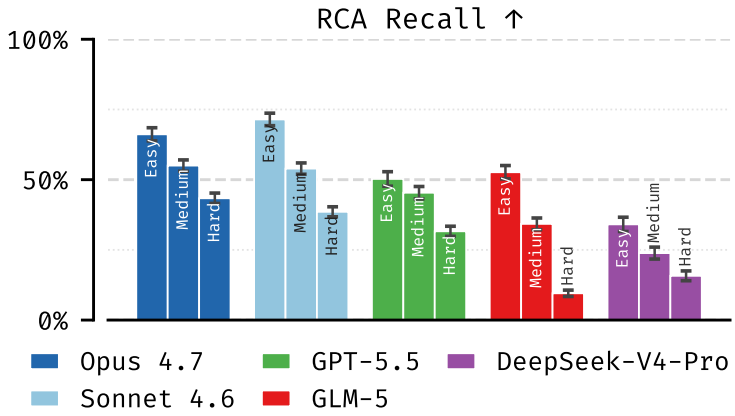


Figure L.5: **RCA recall by issue specificity.** RCA recall is the fraction of a task's ground-truth root causes that the agent identifies. For every model, recall decreases from Easy to Hard tasks. Error bars display ± 1 standard error.

Investigation timing and scenario structure. Fig. L.7 compares RCA depth across TTDs while holding report-time style and offset fixed. Mean depth increases slightly from 15 to 60 minutes, then decreases at 480 minutes for all five models. For reported-time offsets at a fixed eight-hour TTD (Tab. L.5), pooled RCA accuracy is similar when the reported interval contains or excludes the true incident time (21.0% versus 19.8%), while the hallucination rate is higher in the latter setting (17.1% versus 20.6%). Performance also varies across the scheduled scenario types (Tab. L.2): pooled accuracy ranges from 3.1% for independent scenarios to 33.8% for isolated scenarios. Tab. L.3 and L.4 provide individual-scenario and schedule-day breakdowns. These descriptive comparisons characterize variation across benchmark conditions without isolating their causal effects.

Table L.2: **RCA results by scenario type (code + telemetry).** Trials pooled over the scenarios of each type in the incident schedule (App. B.2): Isolated: `d1-i1`, `d1-i2`; Independent: `d1-n1`; Conflicting: `d1-c1`; Conflicting-Cascading: `d2-cc1`, `d4-cc2`; Independent-Cascading: `d3-ic1`, `d5-ic2`; Sequential: `d2-d1dep`, `d3-d2dep`, `d4-d3dep`, `d5-d4dep`; Mixed: `d6-big`. n is the number of trials per model. All values are mean $\pm$ 1 standard error in percent; the judge is GPT-5.4.

(a) RCA Depth, incident tasks

Scenario Type	n	Claude Opus 4.7	Claude Sonnet 4.6	GPT-5.5	DeepSeek-V4-Pro	GLM-5	All models
Isolated	52	42.3 ± 5.5	54.2 ± 5.6	53.2 ± 5.5	28.8 ± 5.4	39.4 ± 5.5	43.6 ± 2.5
Independent	26	3.2 ± 2.3	10.3 ± 4.5	13.5 ± 5.7	0.0 ± 0.0	3.8 ± 2.7	6.2 ± 1.6
Conflicting	17	22.5 ± 5.1	44.1 ± 6.5	39.2 ± 6.2	13.7 ± 5.4	23.5 ± 6.2	28.6 ± 2.9
Conflicting-Cascading	72	50.2 ± 3.5	48.1 ± 3.3	51.8 ± 3.9	22.7 ± 3.6	34.6 ± 3.8	41.5 ± 1.7
Independent-Cascading	130	27.6 ± 3.5	41.7 ± 3.8	33.3 ± 3.6	17.1 ± 3.0	23.8 ± 3.4	28.7 ± 1.6
Sequential	416	58.9 ± 1.3	51.3 ± 1.4	55.1 ± 1.3	22.4 ± 1.5	29.0 ± 1.4	43.3 ± 0.7
Mixed	171	49.8 ± 2.0	42.2 ± 2.0	49.1 ± 1.9	14.7 ± 1.8	25.2 ± 2.0	36.2 ± 1.0
All	884	48.5 ± 1.1	46.7 ± 1.1	48.8 ± 1.1	19.7 ± 1.0	27.7 ± 1.0	38.3 ± 0.5

(b) RCA Accuracy, incident tasks

Scenario Type	n	Claude Opus 4.7	Claude Sonnet 4.6	GPT-5.5	DeepSeek-V4-Pro	GLM-5	All models
Isolated	52	30.8 ± 6.5	42.3 ± 6.9	44.2 ± 7.0	25.0 ± 6.1	26.9 ± 6.2	33.8 ± 2.9
Independent	26	0.0 ± 0.0	3.8 ± 3.8	11.5 ± 6.4	0.0 ± 0.0	0.0 ± 0.0	3.1 ± 1.5
Conflicting	17	0.0 ± 0.0	23.5 ± 10.6	11.8 ± 8.1	23.5 ± 10.6	5.9 ± 5.9	12.9 ± 3.7
Conflicting-Cascading	72	23.6 ± 5.0	31.9 ± 5.5	27.8 ± 5.3	29.2 ± 5.4	26.4 ± 5.2	27.8 ± 2.4
Independent-Cascading	130	26.9 ± 3.9	41.5 ± 4.3	27.7 ± 3.9	18.5 ± 3.4	26.2 ± 3.9	28.2 ± 1.8
Sequential	416	35.6 ± 2.4	31.5 ± 2.3	24.0 ± 2.1	13.5 ± 1.7	16.6 ± 1.8	24.2 ± 0.9
Mixed	171	21.6 ± 3.2	22.2 ± 3.2	18.1 ± 3.0	8.8 ± 2.2	11.1 ± 2.4	16.4 ± 1.3
All	884	28.6 ± 1.5	30.9 ± 1.6	24.3 ± 1.4	15.0 ± 1.2	17.6 ± 1.3	23.3 ± 0.6

(c) RCA Depth, control tasks

Scenario Type	n	Claude Opus 4.7	Claude Sonnet 4.6	GPT-5.5	DeepSeek-V4-Pro	GLM-5	All models
Isolated	12	91.7 ± 8.3	58.3 ± 14.9	75.0 ± 13.1	66.7 ± 14.2	41.7 ± 14.9	66.7 ± 6.1
Independent	6	100.0 ± 0.0	83.3 ± 16.7	100.0 ± 0.0	66.7 ± 21.1	50.0 ± 22.4	80.0 ± 7.4
Conflicting	6	100.0 ± 0.0	66.7 ± 21.1	83.3 ± 16.7	83.3 ± 16.7	83.3 ± 16.7	83.3 ± 6.9
Conflicting-Cascading	24	79.2 ± 8.5	50.0 ± 10.4	66.7 ± 9.8	83.3 ± 7.8	54.2 ± 10.4	66.7 ± 4.3
Independent-Cascading	30	90.0 ± 5.6	43.3 ± 9.2	56.7 ± 9.2	76.7 ± 7.9	43.3 ± 9.2	62.0 ± 4.0
Sequential	96	77.1 ± 4.3	60.4 ± 5.0	77.1 ± 4.3	80.2 ± 4.1	41.7 ± 5.1	67.3 ± 2.1
Mixed	21	100.0 ± 0.0	66.7 ± 10.5	76.2 ± 9.5	71.4 ± 10.1	38.1 ± 10.9	70.5 ± 4.5
All	195	84.1 ± 2.6	57.9 ± 3.5	73.3 ± 3.2	77.9 ± 3.0	44.6 ± 3.6	67.6 ± 1.5

Table L.3: **RCA results by scenario (code + telemetry).** Scenario IDs are `d<day>-<id>` from the incident schedule (App. B.2); Type is the scenario's type in the schedule, except `d6-big`, which combines independent, conflicting, and cascading events and is labeled Mixed. n is the number of trials per model. All values are mean ± 1 standard error in percent; the judge is GPT-5.4.

(a) RCA Depth, incident tasks

Scenario	Type	n	Claude Opus 4.7	Claude Sonnet 4.6	GPT-5.5	DeepSeek-V4-Pro	GLM-5	All models
d1-c1	Conflicting	17	22.5 ± 5.1	44.1 ± 6.5	39.2 ± 6.2	13.7 ± 5.4	23.5 ± 6.2	28.6 ± 2.9
d1-i1	Isolated	26	20.5 ± 7.9	50.0 ± 10.0	43.6 ± 9.6	23.1 ± 8.4	23.1 ± 8.4	32.1 ± 4.1
d1-i2	Isolated	26	64.1 ± 4.8	58.3 ± 5.2	62.8 ± 4.9	34.6 ± 6.7	55.8 ± 5.4	55.1 ± 2.6
d1-n1	Independent	26	3.2 ± 2.3	10.3 ± 4.5	13.5 ± 5.7	0.0 ± 0.0	3.8 ± 2.7	6.2 ± 1.6
d2-cc1	Conflicting-Cascading	49	46.8 ± 3.5	45.9 ± 3.2	51.9 ± 4.0	19.7 ± 4.1	36.5 ± 3.9	40.2 ± 1.8
d2-d1dep	Sequential	104	53.5 ± 2.7	48.1 ± 2.9	50.6 ± 2.8	20.5 ± 2.9	30.7 ± 2.8	40.7 ± 1.4
d3-d2dep	Sequential	104	63.9 ± 2.5	54.9 ± 2.6	63.7 ± 2.6	22.3 ± 3.1	31.3 ± 2.7	47.2 ± 1.4
d3-ic1	Independent-Cascading	52	12.3 ± 4.1	36.5 ± 6.2	15.9 ± 4.7	7.9 ± 3.4	25.0 ± 5.9	19.5 ± 2.3
d4-cc2	Conflicting-Cascading	23	57.4 ± 8.1	52.8 ± 7.9	51.6 ± 8.9	29.0 ± 7.0	30.4 ± 8.4	44.2 ± 3.7
d4-d3dep	Sequential	104	58.7 ± 2.3	50.0 ± 2.7	55.8 ± 2.5	24.4 ± 3.2	28.2 ± 2.7	43.4 ± 1.4
d5-d4dep	Sequential	104	59.4 ± 2.6	52.1 ± 2.9	50.3 ± 2.5	22.5 ± 3.1	25.8 ± 2.8	42.0 ± 1.4
d5-ic2	Independent-Cascading	78	37.8 ± 4.9	45.2 ± 4.8	44.9 ± 4.8	23.3 ± 4.3	23.1 ± 4.2	34.9 ± 2.1
d6-big	Mixed	171	49.8 ± 2.0	42.2 ± 2.0	49.1 ± 1.9	14.7 ± 1.8	25.2 ± 2.0	36.2 ± 1.0
All		884	48.5 ± 1.1	46.7 ± 1.1	48.8 ± 1.1	19.7 ± 1.0	27.7 ± 1.0	38.3 ± 0.5

(b) RCA Accuracy, incident tasks

Scenario	Type	n	Claude Opus 4.7	Claude Sonnet 4.6	GPT-5.5	DeepSeek-V4-Pro	GLM-5	All models
d1-c1	Conflicting	17	0.0 ± 0.0	23.5 ± 10.6	11.8 ± 8.1	23.5 ± 10.6	5.9 ± 5.9	12.9 ± 3.7
d1-i1	Isolated	26	19.2 ± 7.9	50.0 ± 10.0	46.2 ± 10.0	23.1 ± 8.4	23.1 ± 8.4	32.3 ± 4.1
d1-i2	Isolated	26	42.3 ± 9.9	34.6 ± 9.5	42.3 ± 9.9	26.9 ± 8.9	30.8 ± 9.2	35.4 ± 4.2
d1-n1	Independent	26	0.0 ± 0.0	3.8 ± 3.8	11.5 ± 6.4	0.0 ± 0.0	0.0 ± 0.0	3.1 ± 1.5
d2-cc1	Conflicting-Cascading	49	10.2 ± 4.4	20.4 ± 5.8	22.4 ± 6.0	22.4 ± 6.0	20.4 ± 5.8	19.2 ± 2.5
d2-d1dep	Sequential	104	32.7 ± 4.6	30.8 ± 4.5	24.0 ± 4.2	10.6 ± 3.0	17.3 ± 3.7	23.1 ± 1.8
d3-d2dep	Sequential	104	43.3 ± 4.9	38.5 ± 4.8	34.6 ± 4.7	18.3 ± 3.8	17.3 ± 3.7	30.4 ± 2.0
d3-ic1	Independent-Cascading	52	11.5 ± 4.5	32.7 ± 6.6	9.6 ± 4.1	7.7 ± 3.7	25.0 ± 6.1	17.3 ± 2.4
d4-cc2	Conflicting-Cascading	23	52.2 ± 10.6	56.5 ± 10.6	39.1 ± 10.4	43.5 ± 10.6	39.1 ± 10.4	46.1 ± 4.7
d4-d3dep	Sequential	104	33.7 ± 4.7	27.9 ± 4.4	23.1 ± 4.2	12.5 ± 3.3	17.3 ± 3.7	22.9 ± 1.8
d5-d4dep	Sequential	104	32.7 ± 4.6	28.8 ± 4.5	14.4 ± 3.5	12.5 ± 3.3	14.4 ± 3.5	20.6 ± 1.8
d5-ic2	Independent-Cascading	78	37.2 ± 5.5	47.4 ± 5.7	39.7 ± 5.6	25.6 ± 5.0	26.9 ± 5.1	35.4 ± 2.4
d6-big	Mixed	171	21.6 ± 3.2	22.2 ± 3.2	18.1 ± 3.0	8.8 ± 2.2	11.1 ± 2.4	16.4 ± 1.3
All		884	28.6 ± 1.5	30.9 ± 1.6	24.3 ± 1.4	15.0 ± 1.2	17.6 ± 1.3	23.3 ± 0.6

(c) RCA Depth, control tasks

Scenario	Type	n	Claude Opus 4.7	Claude Sonnet 4.6	GPT-5.5	DeepSeek-V4-Pro	GLM-5	All models
d1-c1	Conflicting	6	100.0 ± 0.0	66.7 ± 21.1	83.3 ± 16.7	83.3 ± 16.7	83.3 ± 16.7	83.3 ± 6.9
d1-i1	Isolated	6	100.0 ± 0.0	33.3 ± 21.1	83.3 ± 16.7	50.0 ± 22.4	33.3 ± 21.1	60.0 ± 9.1
d1-i2	Isolated	6	83.3 ± 16.7	83.3 ± 16.7	66.7 ± 21.1	83.3 ± 16.7	50.0 ± 22.4	73.3 ± 8.2
d1-n1	Independent	6	100.0 ± 0.0	83.3 ± 16.7	100.0 ± 0.0	66.7 ± 21.1	50.0 ± 22.4	80.0 ± 7.4
d2-cc1	Conflicting-Cascading	12	66.7 ± 14.2	58.3 ± 14.9	66.7 ± 14.2	91.7 ± 8.3	41.7 ± 14.9	65.0 ± 6.2
d2-d1dep	Sequential	24	54.2 ± 10.4	45.8 ± 10.4	75.0 ± 9.0	83.3 ± 7.8	25.0 ± 9.0	56.7 ± 4.5
d3-d2dep	Sequential	24	87.5 ± 6.9	54.2 ± 10.4	87.5 ± 6.9	79.2 ± 8.5	50.0 ± 10.4	71.7 ± 4.1
d3-ic1	Independent-Cascading	12	83.3 ± 11.2	33.3 ± 14.2	66.7 ± 14.2	83.3 ± 11.2	50.0 ± 15.1	63.3 ± 6.3
d4-cc2	Conflicting-Cascading	12	91.7 ± 8.3	41.7 ± 14.9	66.7 ± 14.2	75.0 ± 13.1	66.7 ± 14.2	68.3 ± 6.1
d4-d3dep	Sequential	24	70.8 ± 9.5	75.0 ± 9.0	70.8 ± 9.5	79.2 ± 8.5	41.7 ± 10.3	67.5 ± 4.3
d5-d4dep	Sequential	24	95.8 ± 4.2	66.7 ± 9.8	75.0 ± 9.0	79.2 ± 8.5	50.0 ± 10.4	73.3 ± 4.1
d5-ic2	Independent-Cascading	18	94.4 ± 5.6	50.0 ± 12.1	50.0 ± 12.1	72.2 ± 10.9	38.9 ± 11.8	61.1 ± 5.2
d6-big	Mixed	21	100.0 ± 0.0	66.7 ± 10.5	76.2 ± 9.5	71.4 ± 10.1	38.1 ± 10.9	70.5 ± 4.5
All		195	84.1 ± 2.6	57.9 ± 3.5	73.3 ± 3.2	77.9 ± 3.0	44.6 ± 3.6	67.6 ± 1.5

Table L.4: **RCA results by schedule day (code + telemetry).** Trials grouped by the day of the incident-schedule scenario each task is drawn from (App. B.2); control tasks inherit the day of their scenario. n is the number of trials per model; Avg GT Events is the mean number of plausible root causes per incident task. All values are mean ± 1 standard error, in percent except Avg GT Events; the judge is GPT-5.4.

(a) RCA Depth, incident tasks

Day	n	Avg GT Events	Claude Opus 4.7	Claude Sonnet 4.6	GPT-5.5	DeepSeek-V4-Pro	GLM-5	All models
1	95	1.41 ± 0.05	28.1 ± 3.6	40.4 ± 4.0	39.8 ± 3.9	18.2 ± 3.3	26.8 ± 3.6	30.7 ± 1.7
2	153	3.49 ± 0.15	51.4 ± 2.2	47.4 ± 2.2	51.0 ± 2.3	20.3 ± 2.3	32.6 ± 2.3	40.5 ± 1.1
3	156	2.58 ± 0.11	46.7 ± 2.9	48.8 ± 2.8	47.8 ± 3.0	17.5 ± 2.4	29.2 ± 2.7	38.0 ± 1.3
4	127	3.39 ± 0.14	58.5 ± 2.4	50.5 ± 2.6	55.1 ± 2.6	25.3 ± 2.9	28.6 ± 2.7	43.6 ± 1.3
5	182	2.66 ± 0.12	50.2 ± 2.7	49.2 ± 2.6	48.0 ± 2.5	22.9 ± 2.5	24.6 ± 2.4	39.0 ± 1.2
6	171	4.22 ± 0.16	49.8 ± 2.0	42.2 ± 2.0	49.1 ± 1.9	14.7 ± 1.8	25.2 ± 2.0	36.2 ± 1.0
All	884	3.06 ± 0.06	48.5 ± 1.1	46.7 ± 1.1	48.8 ± 1.1	19.7 ± 1.0	27.7 ± 1.0	38.3 ± 0.5

(b) RCA Accuracy, incident tasks

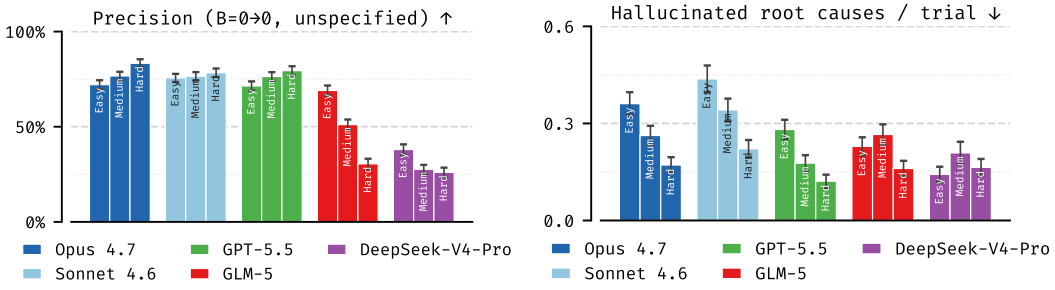
Day	n	Avg GT Events	Claude Opus 4.7	Claude Sonnet 4.6	GPT-5.5	DeepSeek-V4-Pro	GLM-5	All models
1	95	1.41 ± 0.05	16.8 ± 3.9	28.4 ± 4.7	29.5 ± 4.7	17.9 ± 4.0	15.8 ± 3.8	21.7 ± 1.9
2	153	3.49 ± 0.15	25.5 ± 3.5	27.5 ± 3.6	23.5 ± 3.4	14.4 ± 2.8	18.3 ± 3.1	21.8 ± 1.5
3	156	2.58 ± 0.11	32.7 ± 3.8	36.5 ± 3.9	26.3 ± 3.5	14.7 ± 2.8	19.9 ± 3.2	26.0 ± 1.6
4	127	3.39 ± 0.14	37.0 ± 4.3	33.1 ± 4.2	26.0 ± 3.9	18.1 ± 3.4	21.3 ± 3.6	27.1 ± 1.8
5	182	2.66 ± 0.12	34.6 ± 3.5	36.8 ± 3.6	25.3 ± 3.2	18.1 ± 2.9	19.8 ± 3.0	26.9 ± 1.5
6	171	4.22 ± 0.16	21.6 ± 3.2	22.2 ± 3.2	18.1 ± 3.0	8.8 ± 2.2	11.1 ± 2.4	16.4 ± 1.3
All	884	3.06 ± 0.06	28.6 ± 1.5	30.9 ± 1.6	24.3 ± 1.4	15.0 ± 1.2	17.6 ± 1.3	23.3 ± 0.6

(c) RCA Depth, control tasks

Day	n	Claude Opus 4.7	Claude Sonnet 4.6	GPT-5.5	DeepSeek-V4-Pro	GLM-5	All models
1	24	95.8 ± 4.2	66.7 ± 9.8	83.3 ± 7.8	70.8 ± 9.5	54.2 ± 10.4	74.2 ± 4.0
2	36	58.3 ± 8.3	50.0 ± 8.5	72.2 ± 7.6	86.1 ± 5.8	30.6 ± 7.8	59.4 ± 3.7
3	36	86.1 ± 5.8	47.2 ± 8.4	80.6 ± 6.7	80.6 ± 6.7	50.0 ± 8.5	68.9 ± 3.5
4	36	77.8 ± 7.0	63.9 ± 8.1	69.4 ± 7.8	77.8 ± 7.0	50.0 ± 8.5	67.8 ± 3.5
5	42	95.2 ± 3.3	59.5 ± 7.7	64.3 ± 7.5	76.2 ± 6.7	45.2 ± 7.8	68.1 ± 3.2
6	21	100.0 ± 0.0	66.7 ± 10.5	76.2 ± 9.5	71.4 ± 10.1	38.1 ± 10.9	70.5 ± 4.5
All	195	84.1 ± 2.6	57.9 ± 3.5	73.3 ± 3.2	77.9 ± 3.0	44.6 ± 3.6	67.6 ± 1.5

Table L.5: **RCA results by reported-time offset (code + telemetry).** `exact_range` tasks at TTD = 480 min, the only slice in which the offset between the reported and true incident time varies. The user-facing issue reports a range of ±30 min around a time offset from the true incident time: at an offset of +10 min the true incident time lies inside the reported range; at +60 min it lies outside. n is the number of trials. All values are mean ± 1 standard error in percent; the judge is GPT-5.4.

			Outcome			Root cause		Telemetry citations					
Model	Offset (min)	n	RCA Depth	RCA Accuracy	Hall.	Mechanism	Inc. Time	Metrics (all)	Metrics (any)	Logs (all)	Logs (any)	Traces (all)	Traces (any)
Claude Opus 4.7	+10 (inside)	104	45.7 ± 3.1	24.0 ± 4.2	8.7 ± 2.8	82.7 ± 3.7	82.7 ± 3.7	0.0 ± 0.0	67.7 ± 4.9	3.2 ± 1.8	51.6 ± 5.2	26.4 ± 4.6	39.6 ± 5.2
	+60 (outside)	104	46.3 ± 3.2	23.1 ± 4.2	14.4 ± 3.5	78.8 ± 4.0	80.8 ± 3.9	0.0 ± 0.0	52.7 ± 5.2	4.3 ± 2.1	52.7 ± 5.2	26.7 ± 4.7	36.7 ± 5.1
Claude Sonnet 4.6	+10 (inside)	104	45.4 ± 3.1	29.8 ± 4.5	9.6 ± 2.9	83.7 ± 3.6	83.7 ± 3.6	0.0 ± 0.0	47.3 ± 5.2	6.5 ± 2.6	66.7 ± 4.9	25.3 ± 4.6	37.4 ± 5.1
	+60 (outside)	104	45.0 ± 3.0	27.9 ± 4.4	12.5 ± 3.3	81.7 ± 3.8	79.8 ± 4.0	0.0 ± 0.0	40.9 ± 5.1	6.5 ± 2.6	67.7 ± 4.9	28.9 ± 4.8	42.2 ± 5.2
GPT-5.5	+10 (inside)	104	45.1 ± 3.1	21.2 ± 4.0	22.1 ± 4.1	74.0 ± 4.3	82.7 ± 3.7	1.1 ± 1.1	75.3 ± 4.5	22.6 ± 4.4	65.6 ± 5.0	44.0 ± 5.2	54.9 ± 5.2
	+60 (outside)	104	42.4 ± 3.1	21.2 ± 4.0	27.9 ± 4.4	73.1 ± 4.4	77.9 ± 4.1	0.0 ± 0.0	60.2 ± 5.1	17.2 ± 3.9	67.7 ± 4.9	50.0 ± 5.3	62.2 ± 5.1
DeepSeek-V4-Pro	+10 (inside)	104	18.4 ± 2.8	15.4 ± 3.6	6.7 ± 2.5	32.7 ± 4.6	33.7 ± 4.7	0.0 ± 0.0	16.1 ± 3.8	3.2 ± 1.8	23.7 ± 4.4	5.5 ± 2.4	7.7 ± 2.8
	+60 (outside)	104	19.5 ± 2.8	12.5 ± 3.3	8.7 ± 2.8	35.6 ± 4.7	34.6 ± 4.7	0.0 ± 0.0	19.4 ± 4.1	3.2 ± 1.8	20.4 ± 4.2	4.4 ± 2.2	5.6 ± 2.4
GLM-5	+10 (inside)	104	26.1 ± 3.1	14.4 ± 3.5	38.5 ± 4.8	47.1 ± 4.9	60.6 ± 4.8	0.0 ± 0.0	14.0 ± 3.6	2.2 ± 1.5	36.6 ± 5.0	6.6 ± 2.6	8.8 ± 3.0
	+60 (outside)	104	24.0 ± 2.6	14.4 ± 3.5	39.4 ± 4.8	43.3 ± 4.9	56.7 ± 4.9	0.0 ± 0.0	12.9 ± 3.5	2.2 ± 1.5	35.5 ± 5.0	3.3 ± 1.9	7.8 ± 2.8
All models	+10 (inside)	520	36.1 ± 1.4	21.0 ± 1.8	17.1 ± 1.7	64.0 ± 2.1	68.7 ± 2.0	0.2 ± 0.2	44.1 ± 2.3	7.5 ± 1.2	48.8 ± 2.3	21.5 ± 1.9	29.7 ± 2.1
	+60 (outside)	520	35.4 ± 1.4	19.8 ± 1.7	20.6 ± 1.8	62.5 ± 2.1	66.0 ± 2.1	0.0 ± 0.0	37.2 ± 2.2	6.7 ± 1.2	48.8 ± 2.3	22.7 ± 2.0	30.9 ± 2.2



(a) Precision

(b) Hallucinated root causes per task

Figure L.6: **RCA precision and hallucinated root causes by issue specificity.** Precision is the fraction of root causes attributed in the report that are ground truth (set to 0 when the report attributes none), and hallucinated root causes counts attributed root causes that are not ground truth (Sec. 4). A feature flag named without a variant matches if any variant of that flag is ground truth. Error bars display ± 1 standard error.

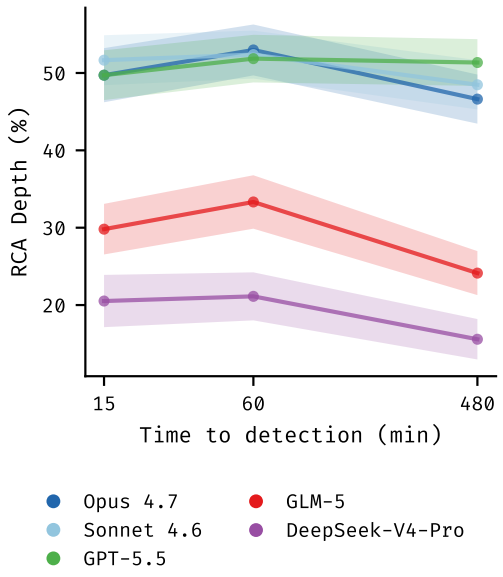


Figure L.7: **RCA depth by time-to-detection (TTD).** `exact` tasks, whose user-facing issue gives a reported time 10 min after incident onset, investigated 15, 60, or 480 min after onset (95, 100, and 104 tasks per model, respectively); the x-axis is log-scaled. Shaded bands display ± 1 standard error.