OPENRCA 2.0: PROCESS-LEVEL EVALUATION OF LLM AGENTS FOR MICROSERVICE ROOT CAUSE ANALYSIS

Anonymous authors
Paper under double-blind review

ABSTRACT

Root cause analysis (RCA) poses a holistic test of LLM agentic capabilities, such as long-context understanding, multi-step reasoning, and tool use. However, existing RCA benchmarks label only the root cause, leaving the agent's reasoning about how the fault produces the observed symptoms unverified. To enable process-level evaluation, we introduce PAVE, a step-wise labeling protocol that uses deliberately injected faults as known interventions to reconstruct verified propagation paths. PAVE has access to fault-injection records unavailable to the agent, enabling it to verify propagation paths from known causes to observed symptoms. Applying PAVE yields OpenRCA 2.0, an RCA benchmark for process-level evaluation of LLM agents, comprising 454 instances across 3 microservice systems. Across 16 frontier LLMs, agents recover the complete set of root causes with an average exact-match accuracy of only 47.6%. Analysis of agent trajectories shows that, for 97.1% of missed root causes, the faulty service had already appeared in a query or tool result. Process-level evaluation reveals that agents often attribute failures to services that interact with the faulty service or are affected by it.

1 INTRODUCTION

LLM-based coding agents have made rapid progress in automating code generation and testing. When the resulting software fails after deployment, *root cause analysis* (RCA) becomes critical for identifying the causes from runtime observations and guiding recovery to limit economic losses (Xu et al., 2025; Chen et al., 2025b;a; Zhang et al., 2025a). Like coding agents debugging a failure, RCA agents reason from observed symptoms to their underlying causes. In the read-only setting studied here, RCA agents must diagnose failures from recorded runtime observations without modifying or rerunning the deployed system to test hypotheses. Diagnosis is particularly challenging in microservice systems, where an application consists of services that communicate over the network and a fault in one service can produce symptoms in others (Zhou et al., 2018). The agent investigates these symptoms using monitoring data (*telemetry*): logs of runtime events, metrics such as response time, and traces recording how requests pass between services.

To evaluate RCA agents, existing benchmarks use *fault injection* to create test cases by deliberately introducing faults into known system components (Pham et al., 2025; Fang et al., 2025; Chen et al., 2025b). The agent receives monitoring data collected during each test case as input, and the component targeted by fault injection serves as the root-cause label. Current evaluation practices check whether the agent's answer matches this label, leaving its reasoning about how the fault produces the observed symptoms unverified (Figure 1). For example, the same incorrect answer may result from overlooking the faulty service or misinterpreting the evidence collected about it. Suppose a slow payment service causes requests to a checkout service to time out. An agent may blame the checkout service either because it does not examine the payment service or because it mistakenly interprets the payment delay as a consequence of the checkout failure.

Distinguishing these failure modes requires *process-level evaluation* of the agent's tool queries and its explanation of how the fault caused the observed symptoms. Checking these explanations requires ground-truth propagation paths that record how a fault spreads between services to produce

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

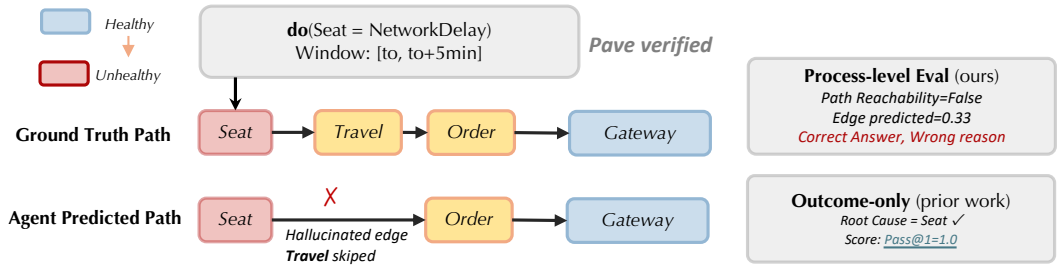

Figure 1: Process-level evaluation on the running NetworkDelay failure in Seat service. From the known intervention (top), PAVE reconstructs the verified propagation path Seat→Travel→Order→Gateway (middle). The agent identifies the correct root cause but reports a propagation graph that skips Travel (bottom): outcome-only evaluation scores this as a success, whereas process-level evaluation locates the step where the diagnosis departs from the verified propagation path.

the observed symptoms. With only monitoring data, annotators face the same ambiguity as RCA agents because different faults can produce similar symptoms. Annotators have access to fault-injection records unavailable to RCA agents, enabling them to verify propagation paths from known faults against monitoring data (Pearl, 2009; Liu et al., 2026).

We introduce PAVE (Path Annotation via Verified Effects), a step-wise labeling protocol that uses this information advantage to reconstruct verified propagation paths. PAVE first checks monitoring data from the targeted component to determine whether the injected fault produced an observable effect. It then accepts a candidate propagation path only when each step is supported by the observed behavior of the services involved. Each step must be consistent with how the services interact, such as a slow payment service delaying the checkout service that calls it. PAVE checks whether each service on the path shows significant changes from its behavior before fault injection and whether each effect begins no earlier than its presumed cause (Section 2.3).

Applying PAVE to 3 microservice systems yields OpenRCA 2.0, a benchmark of 454 instances annotated with verified propagation paths for process-level evaluation of LLM agents. The benchmark covers 23 fault types and records an average of 3.9 verified propagation steps between services per instance.

Across 16 frontier LLMs, agents recover the complete set of root causes with an average exact-match accuracy of only 47.6% (Section 3.2). Analysis of agent trajectories shows that, for 97.1% of missed root causes, the faulty service had already appeared in a query or tool result. Yet for 40.1% of missed root causes, the agent did not obtain the evidence needed to identify the fault, most often a comparison with the service's normal behavior. Process-level evaluation reveals that agents often incorrectly attribute failures to services that interact with the faulty service or are affected by it (Section 3.3).

Our contributions are:

- PAVE, a step-wise labeling protocol that uses injected faults as known interventions to reconstruct verified propagation paths (Section 2).
- OpenRCA 2.0, a benchmark that provides verified propagation paths for 454 instances across 3 microservice systems to support process-level evaluation of LLM agents (Section 3.1).
- A process-level evaluation of 16 frontier LLMs that analyzes agents' tool queries and checks their causal explanations against verified propagation paths to characterize diagnostic failures (Sections 3.2 and 3.3).

2 PAVE: Path Annotation via Verified Effects

A microservice system can be represented as a directed dependency graph over services, pods (which run service containers), and hosts (the machines they run on) (Figure 2a). Faults propagate horizontally through service calls, where a remote procedure call (RPC) fails or slows down, and vertically

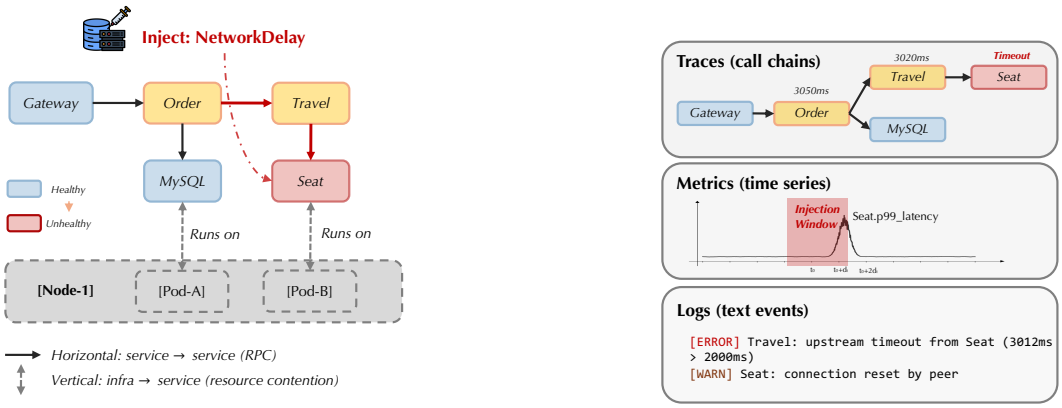

Figure 2: The RCA setting. **(a)** A microservice system forms a two-layer dependency graph. The arrows show the direction of service calls. A fault injected at Seat propagates in the opposite direction along the RPC chain (Seat → Travel → Order) and may also propagate vertically through shared infrastructure (e.g., co-located pods). **(b)** The agent observes only telemetry (traces, metrics, logs) and must reason backward to the originating fault; the intervention itself is hidden.

through shared infrastructure, where services on the same host compete for CPU, memory, or network capacity. The diagnosing agent sees only recorded monitoring data (traces, metrics, and logs; Figure 2b) and reasons backward from these observations to the originating fault.

We now describe the PAVE pipeline (Figure 3). Given fault injection records and the system dependency graph, it reconstructs verified propagation paths in two phases that together enforce three conditions on every admitted path. *Structural Pruning* (Section 2.2) enumerates candidate paths consistent with the system's dependencies and a set of fault-propagation rules, enforcing *mechanism conformance*. *Causal Verification* (Section 2.3) then checks each candidate path against monitoring data collected before injection, enforcing *statistical deviation* and *timing alignment* jointly. It also verifies whether each injected fault took effect, so faults without an observable effect contribute neither a label nor a path.

For each injection, PAVE takes as input the dependency graph $\mathcal{G}$, raw monitoring data $\mathbf{O}^{(t)}$ over the injection window, and the recorded intervention $do(v_{root})$ (target component, fault type, and injection time). Its output is a set of verified propagation paths Π^*. Phase 1 builds an intermediate candidate set Π_{cand} from the dependency graph and rule set; Phase 2 yields $\Pi^* \subseteq \Pi_{cand}$ by verifying these candidates against monitoring data. Each injection in OpenRCA 2.0 carries Π^* as its step-wise annotation.

2.1 Setup: Forward Verification from a Known Intervention

We model a microservice system as a directed dependency graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ over services, pods, and hosts (Figure 2a), with edges encoding service-call or resource dependencies (channels and modalities in Appendix B.5). Fault propagation is a structural causal process over $\mathcal{G}$ (Pearl, 2009): anomalies propagate along edges with finite delay (full equation in Eq. (2), Appendix D.1). An RCA agent observes only monitoring data $\mathbf{O}_{obs}$ and jointly infers the dependency structure, the root cause, and the propagation path, reasoning *backward* from observed effects to their most plausible cause. PAVE, by contrast, has access to the intervention $do(v_{root})$ (target component, fault type, injection window $[t_0, t_0+\Delta t]$, and parameters), giving it a strictly richer information set:

$$\underbrace{\{\mathbf{O}_{obs}\}}_{\text{Agent}} \subset \underbrace{\{\mathbf{O}_{obs},\, do(v_{root})\}}_{\text{PAVE}} \tag{1}$$

In words, the annotator sees everything the agent sees, plus the record of what was injected. Given the known intervention, the annotator accepts a candidate path in $\mathcal{G}$ only when the observed anomalies are consistent with the injected fault propagating along that path. The verified propagation graph $\mathcal{G}^*$ (formed by the edges of all paths in Π^*) is the process-level ground truth (GT) against which agent reasoning is evaluated.

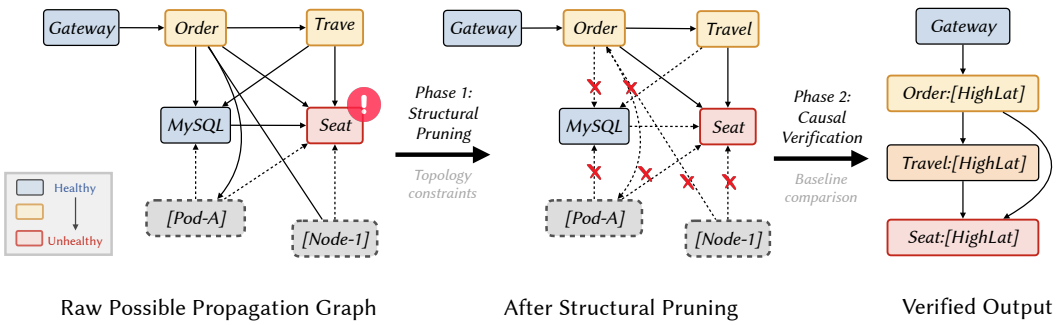

Figure 3: **Two-phase verification filters a candidate graph to recover verified propagation paths.** The TrainTicket example uses a fault injected at *Seat*. Arrows point from the calling service to the service it calls, opposite to the fault propagation shown here. Dotted edges represent infrastructure relations. *Input.* The candidate graph includes observed anomalies and potential dependencies among services, pods, and hosts. *Phase 1 (Structural Pruning).* Edges marked $\times$ are removed because no propagation rule admits them (e.g., *Pod-A→Order*). *Phase 2 (Causal Verification).* Each remaining hop is checked against the pre-injection baseline using records of the caller's requests. The retained paths show how *Seat*'s increased latency affects *Order*, both directly and through *Travel*, and extend to the healthy entry service *Gateway*. Phase 1 checks whether a step is allowed by the propagation rules, and Phase 2 checks whether monitoring data support that step. Faults without a verified effect in the root component's monitoring data are excluded from the label, and the released annotation is the union of the retained paths.

2.2 Phase 1: Structural Pruning

Phase 1 produces a candidate path set Π_{cand} by enforcing topological constraints from the system dependency graph. Raw telemetry $\mathbf{O}^{(t)}$ is first summarized per component and time step as a symbolic state from a small alphabet Σ via a mapping Ψ that depends on the node type (e.g., `HighLatency` for service nodes, `CPU_Throttled` for pods; formal definition in Appendix D.2). A propagation rule set $\mathcal{R}$ admits only transitions between (type, state) pairs $(\theta_i{:}s_i) \xrightarrow{\text{channel}} (\theta_j{:}s_j)$ backed by a known mechanism (competition for shared resources, or an RPC timeout cascade in which a slow service causes requests from services calling it to time out). A depth-first search over a time-expanded graph, whose nodes are (component, state, time) triples (v, s, t) and whose edges move forward in time, unrolls circular call dependencies into acyclic chains, yielding Π_{cand}. The state alphabet, full rule table, and time-expansion details are in Appendix D.

2.3 Phase 2: Causal Verification

Phase 2 produces the verified propagation path set Π^* by separating genuine propagation from coincidental disturbances (e.g., periodic scheduled jobs) that occur during the injection window. For each injection, we use monitoring data collected immediately before injection, after the freshly deployed system has stabilized under normal load, as a *Reference Baseline* ($\mathcal{D}_{base}$) approximating the system's behavior without the fault, $P(\cdot \mid do(v_{root}{=}\emptyset))$. We determine each node's state by comparing its observed metrics with this baseline using distributional tests tailored to each metric type (Appendix D.8). A candidate path $\pi \in \Pi_{cand}$ is *verified* only when (i) every service on the path except the injected one is anomalous and every hop between services has supporting call evidence, (ii) every edge conforms to $\mathcal{R}$, and (iii) each downstream anomaly begins no earlier than its upstream cause. For each hop, the call evidence shows that the calling service's requests to the service it depends on take longer or fail more often than before injection.

Effect verification. A recorded injection does not establish that the fault took effect because the affected code may receive no requests or another fault may prevent requests from reaching the targeted component. Before any path is built, PAVE therefore tests each injected fault against the root component's monitoring data using tests matched to the fault family, such as missing or failing server spans (records of handled requests) for a stopped pod, degraded calls on the injected link for a network fault, or rising resource use for a stress fault. When the injection record specifies a parameter

Table 1: **OpenRCA 2.0 by system.** The three systems differ in size (9–44 services), language stack, and typical propagation depth, so the 454 instances cover distinct architectures rather than resampling one. Cluster configuration in Appendix D.4; collection and curation details in Appendix A.

Property	TrainTicket	OTel Demo	Hotel Res.	Total
Services in system	44	15	9	
Implementation	Java/REST	polyglot/gRPC+Kafka	Go/gRPC	
Evaluable instances	274	32	148	454
Distinct chaos primary kinds	21	7	5	23
Mean longest propagation path	2.5	2.3	1.4	2.1
Avg. causal edges per instance	5.3	3.0	1.5	3.9

such as an added delay, an observed shift matching that parameter is accepted directly. Faults with a service-level effect form the *required* labels; a fault without an observable effect contributes neither a label nor a path, and a fault whose effect stays on its own resources is kept as an *optional* label.

Baseline. Deviations are judged against two references, with the Reference Baseline $\mathcal{D}_{base}$ setting the normal level and the pooled normal windows of the same system estimating how much each signal fluctuates without a fault. If a request flow stops before injection, its Reference Baseline is restricted to the period before that flow stopped. A deviation counts only when it exceeds both reference thresholds, filtering out fluctuations also observed in fault-free windows. An instance enters OpenRCA 2.0 only when at least one injected fault has a service-level effect; its label lists the faults that took effect, and Π^* holds their verified propagation paths.

3 Experiments

Process-level ground truth shows where LLM-based RCA agents fail: across 16 frontier LLMs, missed root-cause services almost always appeared in queries or tool results, and wrong answers mostly lie on the faulty call or its verified propagation graph.

3.1 Experimental Setup

This subsection specifies the three microservice systems we run, the fault-injection campaigns that produce the raw pool, the curation that yields OpenRCA 2.0, and the agent design under evaluation.

Systems and Faults. We evaluate on 3 architecturally distinct microservice systems: *TrainTicket* (Zhou et al., 2018), *Hotel Reservation* from DeathStarBench (Dea, 2024), and the *OpenTelemetry Demo* (OTel Demo) application (Ope, 2026), spanning different programming languages and synchronous/asynchronous communication patterns; deployment specifications are in Appendix B. For each system we run fault-injection workloads covering 23 fault types across application, resource, network, and pod-level faults (per-system taxonomies in Appendix C.2); for TrainTicket we build on the released dataset of Fang et al. (2025) and augment it with new campaigns on the other two systems. From a candidate pool of 1,464 injections, we retain 500 instances through structural filtering and stratified selection by (system, fault family, root-cause service), with a cap on the number selected from each group. We then apply PAVE (Section 2) to remove injected faults without observable effects, yielding OpenRCA 2.0 with 454 instances (full procedure in Appendix A). Table 1 reports per-system breakdowns; the Datasheet is in Appendix A.

Ground Truth. Verified propagation paths are extracted automatically using the pipeline described in Section 2; their reliability is measured by the blind per-edge audit in Appendix E.1.

Agent Design. All evaluated agents follow a tool-augmented ReAct-style (Yao et al., 2022) architecture, adapted from the Deep Research agent (Lin, 2025). To isolate LLM reasoning capability from tooling effects, all models share an identical tool suite over the released telemetry and identical prompt templates; no dependency graph is provided. Each diagnosis is returned as structured output: root-cause claims and propagation edges, each paired with a re-executable SQL query as evidence (full schema, tool definitions, and prompts in Appendix F and F.3).

Table 2: Agents locate a correct root-cause service (AnySvc) far more often than the exact root-cause set (EM), and Edge F1 trails Node F1 for every model. Outcome- and process-level evaluation on OpenRCA 2.0 pooled across 3 systems (n=454, percentages). **Bold** marks the per-column maximum. Each subscript is the half-width of a case-level percentile bootstrap 95% confidence interval (B=10,000 resamples). Metric definitions in Section 3.2; per-system breakdown in Appendix Table 8.

	Outcome					*Process*			
Model	EM↑	F1↑	Precision↑	Recall↑	AnySvc↑	PR↑	Node F1↑	Edge F1↑	SQL Exec↑
Claude Opus 5	**68.5**$_{\pm 4.2}$	**70.7**$_{\pm 4.1}$	**70.4**$_{\pm 4.1}$	**71.7**$_{\pm 4.1}$	93.2$_{\pm 2.3}$	90.4$_{\pm 2.7}$	90.9$_{\pm 1.2}$	83.9$_{\pm 1.9}$	97.6$_{\pm 0.7}$
Claude Fable 5	65.2$_{\pm 4.4}$	66.5$_{\pm 4.3}$	66.3$_{\pm 4.3}$	67.0$_{\pm 4.3}$	91.4$_{\pm 2.5}$	89.6$_{\pm 2.9}$	91.1$_{\pm 1.3}$	85.8$_{\pm 1.9}$	99.6$_{\pm 0.2}$
GPT-5.6 Sol	65.2$_{\pm 4.4}$	66.5$_{\pm 4.3}$	66.4$_{\pm 4.4}$	66.9$_{\pm 4.4}$	90.5$_{\pm 2.8}$	84.2$_{\pm 3.4}$	85.8$_{\pm 1.7}$	78.7$_{\pm 2.5}$	**99.8**$_{\pm 0.2}$
Grok 4.6	60.8$_{\pm 4.6}$	62.5$_{\pm 4.5}$	62.3$_{\pm 4.5}$	63.0$_{\pm 4.5}$	**95.6**$_{\pm 1.9}$	**92.5**$_{\pm 2.5}$	**91.9**$_{\pm 1.1}$	**87.2**$_{\pm 1.7}$	97.8$_{\pm 0.8}$
Gemini 3.7 Flash	58.4$_{\pm 4.5}$	58.6$_{\pm 4.5}$	58.7$_{\pm 4.5}$	58.6$_{\pm 4.5}$	91.2$_{\pm 2.6}$	88.9$_{\pm 2.9}$	89.1$_{\pm 1.4}$	82.6$_{\pm 2.1}$	99.4$_{\pm 0.5}$
Hy4 Preview	54.6$_{\pm 4.6}$	56.3$_{\pm 4.5}$	56.1$_{\pm 4.5}$	56.8$_{\pm 4.6}$	94.7$_{\pm 2.1}$	92.2$_{\pm 2.5}$	90.7$_{\pm 1.2}$	85.1$_{\pm 1.8}$	98.8$_{\pm 0.5}$
Kimi K3	53.7$_{\pm 4.5}$	55.1$_{\pm 4.5}$	54.9$_{\pm 4.5}$	55.5$_{\pm 4.5}$	91.9$_{\pm 2.5}$	89.6$_{\pm 2.9}$	89.3$_{\pm 1.4}$	82.9$_{\pm 2.1}$	99.6$_{\pm 0.3}$
Claude Sonnet 5	52.2$_{\pm 4.6}$	53.6$_{\pm 4.6}$	53.4$_{\pm 4.6}$	54.0$_{\pm 4.7}$	91.2$_{\pm 2.5}$	86.4$_{\pm 3.3}$	89.4$_{\pm 1.3}$	81.8$_{\pm 2.2}$	97.8$_{\pm 0.6}$
Seed 2.1 Pro	40.5$_{\pm 4.5}$	42.5$_{\pm 4.4}$	42.2$_{\pm 4.4}$	43.3$_{\pm 4.5}$	82.8$_{\pm 3.4}$	79.5$_{\pm 3.9}$	82.5$_{\pm 1.9}$	71.5$_{\pm 2.8}$	96.3$_{\pm 0.8}$
Muse Spark 1.2	39.2$_{\pm 4.4}$	42.1$_{\pm 4.4}$	41.5$_{\pm 4.4}$	43.4$_{\pm 4.5}$	81.5$_{\pm 3.5}$	73.4$_{\pm 4.1}$	82.1$_{\pm 2.0}$	67.9$_{\pm 3.0}$	98.2$_{\pm 0.7}$
MiniMax M3	39.2$_{\pm 4.5}$	41.7$_{\pm 4.5}$	41.2$_{\pm 4.4}$	43.0$_{\pm 4.6}$	79.7$_{\pm 3.7}$	68.9$_{\pm 4.4}$	79.2$_{\pm 2.1}$	64.1$_{\pm 3.3}$	94.9$_{\pm 1.1}$
MiMo 2.5 Pro	38.5$_{\pm 4.4}$	42.6$_{\pm 4.4}$	41.6$_{\pm 4.4}$	44.6$_{\pm 4.5}$	83.5$_{\pm 3.4}$	77.1$_{\pm 4.0}$	81.3$_{\pm 2.0}$	70.2$_{\pm 2.9}$	90.3$_{\pm 1.6}$
DeepSeek V4 Flash	38.1$_{\pm 4.5}$	43.7$_{\pm 4.4}$	42.5$_{\pm 4.4}$	46.3$_{\pm 4.6}$	89.4$_{\pm 2.9}$	85.2$_{\pm 3.4}$	85.0$_{\pm 1.8}$	73.5$_{\pm 2.6}$	90.3$_{\pm 1.5}$
GLM 5.3	34.4$_{\pm 4.4}$	45.6$_{\pm 4.2}$	42.8$_{\pm 4.1}$	51.4$_{\pm 4.5}$	91.9$_{\pm 2.5}$	86.4$_{\pm 3.3}$	86.0$_{\pm 1.6}$	75.1$_{\pm 2.6}$	96.8$_{\pm 0.7}$
Step 3.7 Flash	29.6$_{\pm 4.2}$	31.5$_{\pm 4.2}$	31.1$_{\pm 4.2}$	32.6$_{\pm 4.3}$	77.5$_{\pm 3.8}$	70.8$_{\pm 4.4}$	75.2$_{\pm 2.3}$	61.4$_{\pm 3.3}$	92.1$_{\pm 1.7}$
Qwen3.8-Max	24.0$_{\pm 4.0}$	43.7$_{\pm 3.9}$	38.9$_{\pm 3.8}$	53.6$_{\pm 4.6}$	93.8$_{\pm 2.2}$	88.9$_{\pm 2.9}$	87.9$_{\pm 1.3}$	80.8$_{\pm 2.1}$	98.9$_{\pm 0.4}$

3.2 Benchmark Utility: Process-Level Evaluation of RCA Agents

This subsection grades 16 frontier LLMs against OpenRCA 2.0 at the outcome and process levels and reads two patterns out of the pooled results: agents locate a correct service far more often than the full root-cause set (Finding 1), and Edge F1 is lower than Node F1 for every model (Finding 2).

Experimental Design. We evaluate the RCA agent architecture above with 16 frontier LLMs as backbones, including both closed-source and open-source models from 12 providers (Table 2). Each model receives monitoring data (traces, metrics, and logs) and is asked to diagnose the root cause while explaining the failure propagation as a causal graph.

Evaluation Metrics. We grade each diagnosis at two complementary levels (formal definitions in Appendix G). The *outcome level* scores the agent's predicted root-cause set against ground truth: *Exact Match* (EM) requires the predicted set to match ground truth exactly; *F1*, *Precision*, and *Recall* are computed over (service, fault_kind) pairs for each instance and then averaged; *AnySvc* is the case-level rate of naming at least one correct root-cause service ignoring fault kind, the most lenient outcome score and a service-only relaxation of Recall. The *process level* scores the agent's causal explanation. *Path Reachability* (PR) is satisfied when the agent names a correct root-cause service and its predicted graph contains a directed path from that service to a ground-truth alarm node, with at least one edge and every edge belonging to $\mathcal{G}^*$. An alarm node is a component observed to behave abnormally during the injection window. *Node F1* and *Edge F1* compare the nodes and directed edges of the predicted graph $\hat{\mathcal{G}}$ with those of $\mathcal{G}^*$ using exact matching. *SQL Exec* is the fraction of queries cited as evidence that execute successfully and return at least one row, computed for each instance and then averaged. It does not assess whether the returned rows support the claims. Outcome metrics score only the required labels (Section 2.3), and reporting an optional label is neither required nor penalized. Graph-based metrics exclude instances with no edges in $\mathcal{G}^*$. PR is computed on the 425 instances with an alarm downstream of a root cause.

Results. Two patterns govern the pooled results in Table 2. First, agents identify at least one correct root-cause service in 88.7% of cases, but recover the exact root-cause set in only 47.6% (Finding 1). Second, mean Edge F1 (77.0%) trails mean Node F1 (86.1%), and the gap holds for every model (Finding 2). The pooled scores are lifted by Hotel Reservation, whose instances contain only 1.5 verified propagation edges on average (vs. 5.3 on TrainTicket) and where the average model reaches 86.5% EM; on TrainTicket, the largest system, the strongest model reaches only 53.3% EM and the average model 26.6% (per-system leaderboards in Appendix Table 8). Every cell carries a case-level bootstrap 95% CI; under a paired bootstrap most adjacent rows are not significantly ordered, so the

table supports tiers rather than a strict ranking, whereas the Node F1 to Edge F1 gap is significant for every model (Appendix H.3).

(1) Agents locate a correct service far more often than the full root-cause set. AnySvc reaches 88.7% on average, whereas Recall over (service, fault kind) pairs reaches 53.2% and EM 47.6%. The drop mainly reflects errors in identifying the fault kind. Agents identify the correct service but assign the wrong fault kind for 34.6% of required root causes, and fail to identify the service for 11.9%. Section 3.3 traces the missed root causes through the agents' trajectories.

(2) Edge F1 is lower than Node F1 for every model. Agents identify the participating services more reliably than the dependencies that actually carried the fault (86.1% vs. 77.0% on average; the gap is significant for every model, Appendix H.3). Among services present in both the predicted graph and $\mathcal{G}^*$, edge precision is 95.7%, edge recall is 89.8%, and direction accuracy is 99.9%. These results suggest that errors in identifying the affected services contribute substantially to the gap between Node F1 and Edge F1. Correct outcomes do not imply correct explanations. Among exact-match answers, 29.1% have propagation graphs that differ from $\mathcal{G}^*$, and 22.4% omit edges from $\mathcal{G}^*$.

3.3 Failure Analysis

For each required root cause missed by an agent (885 across the 16 models), we link the agent's queries to their results. We record whether the agent retrieved monitoring data from the root-cause service and whether the results contained the evidence needed to identify the fault (Appendix E.2). When the agent reports a different root-cause service, we also record that service's position relative to the verified propagation graph. A separate sample of 23 missed root causes was manually labelled with the automatic labels hidden. The manual and automatic labels agree on how the root-cause service was examined in 100% of cases and on the categories of services reported instead in 78%.

(1) Agents usually retrieve data from the faulty service even in failed diagnoses. Queries execute successfully at similar rates in incorrect and correct diagnoses (87.0% vs. 85.9%). SQL Exec scores are also similar between the two groups (96.5% vs. 97.1%). For 97.1% of missed root causes, at least one query returned monitoring data from the faulty service.

(2) Insufficient diagnostic evidence. For 58.1% of missed root causes, the query results contained the evidence needed to identify the fault, while for 40.1% they lacked necessary evidence. The most common missing element was a comparison with the service's normal behavior (28.0% of missed root causes).

(3) Misattribution to related services. Setting aside predictions of other valid root causes, 67.6% of wrong answers name either the service at the other end of the faulty call or a service reached by fault propagation from the missed root cause in $\mathcal{G}^*$. These two categories account for 44.7% and 23.0% of wrong answers, respectively. For HTTP faults, agents may incorrectly blame the service receiving the request even when its own records show successful responses. Among wrong answers, 21.6% name a service that was already anomalous before injection, and 24.5% name an unrelated service. An answer may fall into multiple categories because it can name several services.

Case Study: The Other Endpoint of the Faulty Call. Figure 4 shows the most common pattern. An HTTP fault replaces the replies ts-auth receives from ts-verification-code with 503; ts-auth fails 99.9% of these calls while ts-verification-code answers every request with 200. Monitoring data identifies which side of the call is at fault: ts-ui-dashboard receives 200 on all its calls to ts-verification-code during the fault, indicating that ts-verification-code is healthy and the failure lies on ts-auth's side. Claude Opus 5 queries both sides, cites the contrast between ts-verification-code's 200 and ts-auth's 503 as evidence, and still reports ts-verification-code as the root cause, as do 12 of the 16 evaluated models. Outcome-only evaluation scores these answers like any wrong guess; process-level evaluation shows that the evidence was examined and the causal direction was inverted.

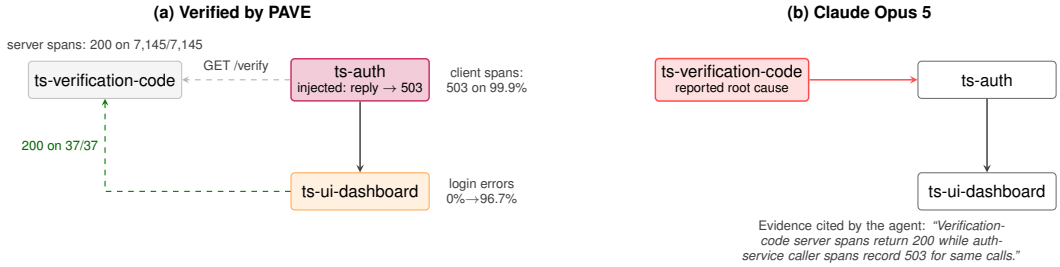

Figure 4: **Agents see the evidence but misjudge the causal direction.** An HTTP fault replaces the replies that ts-auth receives from ts-verification-code with 503, while the callee's own server spans stay healthy. (a) PAVE verifies the effect on ts-auth and its propagation to ts-ui-dashboard. The telemetry also decides which side of the call is at fault: ts-verification-code's other caller, ts-ui-dashboard, receives 200 on all its calls, so only replies travelling to ts-auth fail. (b) Claude Opus 5 cites the contrast between the callee's 200 and the caller's 503, yet reports the callee as the root cause; 12 of the 16 evaluated models make the same attribution.

Table 3: Differentiation of OpenRCA 2.0 from prior RCA benchmarks. "✓" = present; "–" = absent or not reported as a fixed-size benchmark; "part." = partial form (e.g., issue-type tag without a propagation graph). Instance counts are reported as in each source.

Benchmark	#Sys	#Inst.	Outcome	**Process**	LLM eval
OpenRCA 1.0 (Xu et al., 2025)	3	335	✓	–	✓
AIOpsLab (Chen et al., 2025b)	2	48	✓	part.	✓
RCAEval (Pham et al., 2025)	3	735	✓	–	–
Stratus (Chen et al., 2025a)	3	–	✓	–	✓
Fang et al. (2025)	1	8137	✓	–	–
OpenRCA 2.0 (ours)	3	454	✓	✓	✓

4 Related Work

4.1 From Classification-Based Ranking to Generative Reasoning

The formulation of Root Cause Analysis (RCA) has undergone a fundamental shift. Traditionally, RCA was cast as a *multi-class classification problem*, where each candidate component is scored and ranked by its likelihood of being the root cause, using heuristic algorithms (e.g., Random Walk (Yu et al., 2021)) or supervised graph neural networks (Zhang et al., 2025b). While effective for locating faults, these "black-box" approaches lack interpretability and fail to capture the sequential propagation mechanism. Recent LLM-based approaches adopt a *generative reasoning paradigm* (Xu et al., 2025; Chen et al., 2025b;a; Zhang et al., 2025a), treating RCA as a causal narrative generation task. However, the underlying data supervision has not kept pace with advances in model reasoning. Existing benchmarks (Pham et al., 2025; Fang et al., 2025; Chen et al., 2025b) still rely on outcome-based labels inherited from the classification era, recording only *which* component is the root cause without annotating *how* the fault propagates. This forces reasoning agents to optimize for the final answer directly, bypassing the logical derivation steps and encouraging the memorization of spurious correlations rather than learning robust causal dynamics.

4.2 Causal Structure Learning under Intervention

Our work intersects with causal discovery, particularly in dynamic systems. Classical observational methods (e.g., PC algorithm (Spirtes et al., 2000)) struggle to recover Directed Acyclic Graphs (DAGs) in distributed systems due to unobserved confounders and high-dimensional noise (Ikram et al., 2022; Han et al., 2025). Pearl's hierarchy of causation posits that interventional data ($do(x)$) is essential for distinguishing true causality from mere correlation (Pearl, 2009). While some RCA methods attempt to infer causal graphs relying solely on observational monitoring data (Li et al., 2022; Yu et al., 2023), they lack the *identifiability* guarantees that controlled interventions provide. PAVE distinguishes itself by adopting an *interventional* rather than purely observational perspective

during annotation. While prior benchmarks use only the identity of the injected component as the ground-truth label, PAVE uses the full intervention specification ($do(v_{root})$) to verify propagation paths. This effectively transforms the intractable structure learning problem into a tractable verification task, filtering out injections without observable effects and confounding noise that plague purely observational datasets.

4.3 Process Supervision and Verifiability

In the broader LLM landscape, focus is shifting from outcome-based supervision (Outcome Reward Models, ORMs) to *Process Supervision* (Process Reward Models, PRMs) (Lightman et al., 2023; Uesato et al., 2022; Zeng et al., 2025). Research in mathematical reasoning and code generation demonstrates that providing step-by-step verification signals significantly reduces hallucinations and improves generalization (Yin et al., 2025; Yan et al., 2025). However, applying process supervision to system diagnostics is hindered by the *verification gap*: unlike math (where proofs are deterministic) or code (where unit tests exist), operational faults lack a natural "compiler" to verify reasoning steps. Current RCA benchmarks like AIOpsLab (Chen et al., 2025b) or OpenRCA (Xu et al., 2025) rely on human-curated scenarios or heuristically filtered logs, which are hard to scale and lack granular process labels. PAVE addresses this verification gap for system diagnostics by synthesizing verifiable failure propagation paths rooted in system topology and causal interventions, producing the first large-scale process-level ground truth for *evaluating* diagnostic reasoning in complex systems. The resulting step-wise annotations also open a path toward training PRMs in the reliability domain, though we leave this exploration to future work.

Differentiation summary. Table 3 compares OpenRCA 2.0 to five prior RCA benchmarks on five axes. Among these benchmarks, only OpenRCA 2.0 records verified propagation paths, which support the analysis of agents' diagnostic errors in Section 3.3.

5 Assumptions and Threats to Validity

Input assumptions. The protocol relies on four falsifiable conditions: a real controlled intervention $do(v_{root})$ rather than an inferred label, a service-layer dependency graph complete enough that genuine horizontal propagation survives structural pruning, observable fault effects in traces, metrics, or logs, and a pre-injection baseline that approximates the no-intervention distribution closely enough for baseline-relative anomaly screening to discriminate.

Internal threats. Two limitations remain in the pipeline's individual checks: the rule set $\mathcal{R}$ may silently prune propagation mechanisms outside its current vocabulary, and the per-node anomaly screen is calibrated toward low false negatives (calibration split in Appendix E.3) and can therefore incorrectly flag individual nodes as anomalous. Requiring rule consistency, call evidence against the baseline, and timing alignment together reduces spurious edges but does not rule them out. Two services affected by a common cause may still pass all three checks if they call each other. In multi-fault instances, the protocol builds paths for each fault separately and does not determine each fault's contribution to a shared symptom. In a blind audit of 28 instances, 98.3% of the annotated propagation edges were judged correct (Appendix E.1).

External transfer. Reported numbers apply to the tested testbeds; transferring them to production deployments requires calibration. Step-wise ground truth established from complete monitoring data can be reused to evaluate agents with subsampled traces, missing data types, or shorter observation windows.

6 Conclusion

We introduced PAVE, a step-wise labeling protocol that uses injected faults as known interventions to reconstruct verified propagation paths. Applying it to 3 microservice systems yields OpenRCA 2.0, 454 instances annotated with verified propagation paths. Process-level evaluation of 16 frontier LLMs on this benchmark traces each missed root cause through the agent's trajectory and shows that agents often attribute failures to services that interact with the faulty service or are affected by it. More broadly, step-wise causal ground truth helps locate where a diagnosis went wrong, providing information needed to improve diagnostic agents.

REFERENCES

Deathstarbench. https://github.com/delimitrou/DeathStarBench/tree/master, 2024. Accessed: 2026-02-10.

Opentelemetry demo. https://github.com/open-telemetry/opentelemetry-demo, 2026. Accessed: 2026-05-05.

Yinfang Chen, Jiaqi Pan, Jackson Clark, Yiming Su, Noah Zheutlin, Bhavya Bhavya, Rohan Arora, Yu Deng, Saurabh Jha, and Tianyin Xu. Stratus: A multi-agent system for autonomous reliability engineering of modern clouds. *arXiv preprint arXiv:2506.02009v2*, 05 2025a. URL https://arxiv.org/abs/2506.02009v2.

Yinfang Chen, Manish Shetty, Gagan Somashekar, Minghua Ma, Yogesh Simmhan, Jonathan Mace, Chetan Bansal, Rujia Wang, and Saravan Rajmohan. Aiopslab: A holistic framework to evaluate ai agents for enabling autonomous clouds. In *MLSys*, 2025b. URL https://openreview.net/forum?id=3EXBLwGxtq.

Aoyang Fang, Songhan Zhang, Yifan Yang, Haotong Wu, Junjielong Xu, Xuyang Wang, Rui Wang, Manyi Wang, Qisheng Lu, and Pinjia He. Rethinking the evaluation of microservice rca with a fault propagation-aware benchmark. *arXiv preprint arXiv:2510.04711v2*, 10 2025. URL https://arxiv.org/abs/2510.04711v2.

Xiao Han, Saima Absar, Lu Zhang, and Shuhan Yuan. Root cause analysis of anomalies in multivariate time series through granger causal discovery. In *The Thirteenth International Conference on Learning Representations*, 2025.

Azam Ikram, Sarthak Chakraborty, Subrata Mitra, Shiv Saini, Saurabh Bagchi, and Murat Kocaoglu. Root cause analysis of failures in microservices through causal discovery. *Advances in Neural Information Processing Systems*, 35:31158–31170, 2022.

Zeyan Li, Nengwen Zhao, Shenglin Zhang, Yongqian Sun, Pengfei Chen, Xidao Wen, Minghua Ma, and Dan Pei. Constructing large-scale real-world benchmark datasets for aiops. *arXiv preprint arXiv:2208.03938v1*, 08 2022. URL https://arxiv.org/abs/2208.03938v1.

Hunter Lightman, Vineet Kosaraju, Yuri Burda, Harrison Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. Let's verify step by step. In *The Twelfth International Conference on Learning Representations*, 2023.

Paichun Lin. Self-balancing agentic AI: Test-time diffusion and context engineering re-imagined for deep research. https://github.com/thinkdepthai/Deep_Research, 2025.

Wei Liu, Siya Qi, Yali Du, and Yulan He. Self-play only evolves when self-synthetic pipeline ensures learnable information gain. *arXiv preprint arXiv:2603.02218v1*, 02 2026. URL https://arxiv.org/abs/2603.02218v1.

Judea Pearl. *Causality*. Cambridge university press, 2009.

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*, pp. 777–780, 2025.

Peter Spirtes, Clark N Glymour, and Richard Scheines. *Causation, prediction, and search*. MIT press, 2000.

Jonathan Uesato, Nate Kushman, Ramana Kumar, Francis Song, Noah Siegel, Lisa Wang, Antonia Creswell, Geoffrey Irving, and Irina Higgins. Solving math word problems with process- and outcome-based feedback. *arXiv preprint arXiv:2211.14275v1*, 11 2022. URL https://arxiv.org/abs/2211.14275v1.

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.

Yibo Yan, Jiamin Su, Jianxiang He, Fangteng Fu, Xu Zheng, Yuanhuiyi Lyu, Kun Wang, Shen Wang, Qingsong Wen, and Xuming Hu. A survey of mathematical reasoning in the era of multimodal large language model: Benchmark, method & challenges. In *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 11798–11827, 2025.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *The eleventh international conference on learning representations*, 2022.

Zhangyue Yin, Qiushi Sun, Zhiyuan Zeng, Qinyuan Cheng, Xipeng Qiu, and Xuan-Jing Huang. Dynamic and generalizable process reward modeling. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 4203–4233, 2025.

Guangba Yu, Pengfei Chen, Hongyang Chen, Zijie Guan, Zicheng Huang, Linxiao Jing, Tianjun Weng, Xinmeng Sun, and Xiaoyun Li. Microrank: End-to-end latency issue localization with extended spectrum analysis in microservice environments. In *Proceedings of the Web Conference 2021*, pp. 3087–3098, 2021.

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

Thomas Zeng, Shuibai Zhang, Shutong Wu, Christian Classen, Daewon Chae, Ethan Ewer, Minjae Lee, Heeju Kim, Wonjun Kang, Jackson Kunde, et al. Versaprm: Multi-domain process reward model via synthetic reasoning data. In *Forty-second International Conference on Machine Learning*, 2025.

Lingzhe Zhang, Yunpeng Zhai, Tong Jia, Chiming Duan, Siyu Yu, Jinyang Gao, Bolin Ding, Zhonghai Wu, and Ying Li. Thinkfl: Self-refining failure localization for microservice systems via reinforcement fine-tuning. *ACM Transactions on Software Engineering and Methodology*, 2025a. doi: 10.1145/3789262. URL https://doi.org/10.1145/3789262.

Songhan Zhang, Aoyang Fang, Yifan Yang, Ruiyi Cheng, Xiaoying Tang, and Pinjia He. Dynacausal: Dynamic causality-aware root cause analysis for distributed microservices. *arXiv preprint arXiv:2510.22613v1*, 10 2025b. URL https://arxiv.org/abs/2510.22613v1.

Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chao Ji, Wenhai Li, and Dan Ding. Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study. *IEEE Transactions on Software Engineering*, 47(2):243–260, 2018.

Appendix organization. This appendix is organized into five parts. *Part I* (Section A) is the dataset-side documentation: Datasheet, including a reproducibility section. *Part II* (Sections B–C) specifies the benchmark testbed. *Part III* (Section D) expands the PAVE pipeline compressed in Section 2. *Part IV* (Section E) reports the blind per-edge audit of the released paths, the failure-attribution method, and the calibration split used by the per-node anomaly screen. *Part V* (Sections F–G) documents the evaluated RCA agents and the formal metric definitions.

A Datasheet for Datasets

A.1 Motivation

OpenRCA 2.0 is designed for *process-level* evaluation of LLM-based RCA agents. The goal is to assess not only whether an agent identifies the correct root cause, but also where a failed diagnosis went wrong. Existing RCA benchmarks typically supervise only the final outcome label, which cannot distinguish an agent that never examined the faulty service from one that examined it and misjudged the causal direction. No dedicated external funding was used specifically to compile the released benchmark artifact beyond the broader research effort supporting this study.

A.2 Composition

Each instance contains three components: (i) multimodal pre- and post-injection telemetry, including distributed traces, metric time series, and structured logs in parquet format; (ii) a structured ground-truth label specifying the originating system, the injected component, and the fault type; and (iii) a step-wise causal propagation graph over services, pods, and spans (Section 2). OpenRCA 2.0 contains 454 evaluable instances from 3 architecturally distinct microservice systems and covers 23 fault types, with per-system breakdowns in Table 1. Curation proceeds in three steps. From a candidate pool of 1,464 injections, structural filtering based on the earlier annotation removes instances whose service graph is cyclic, whose longest propagation path spans at most one service, or whose root is a frontend or load-generator entry point. Two-pass stratified selection by (system, fault family, root-cause service) then retains 500 instances, with a cap on the number selected from each group and candidates ordered by (longest path, edge count, number of services). Of these, 10 instances were subsequently replaced through manual selection. The final filtering stage excludes 33 instances without a service-level fault effect and 4 without any observable fault effect, based on PAVE, along with 6 instances excluded through manual review. Because structural filtering used the earlier annotation, some released instances do not meet the original criteria. In the released labels, 4 instances have a frontend root, 1 has a cyclic service graph, 129 have a longest path of one hop, and 22 have no service edge. The resulting evaluation set is curated for balanced coverage and is not intended to be statistically representative of production incidents.

The current release does not define train/development/test splits. Retained instances are distributed as complete telemetry packs rather than field-pruned records, so field-level missingness or redaction is not a concern within the released trace, metric, or log data. Because the benchmark consists of synthetic operational telemetry generated under controlled load, it contains no PII or human-subject data; people-related subpopulation, identity, and sensitive-attribute considerations are therefore not applicable.

A.3 Collection Process

Faults are injected on three microservice systems: TrainTicket (Zhou et al., 2018), the OpenTelemetry Demo application (Ope, 2026), and Hotel Reservation from DeathStarBench (Dea, 2024). These interventions are run with Chaos Mesh on a shared 7-node Kubernetes cluster; per-system topology and deployment details are given in Section B, and the cluster configuration in Section D.4. For each intervention, a fresh system instance is deployed and warmed up, a pre-injection baseline window is recorded, the fault is applied over $[t_0, t_0+\Delta t]$, and post-injection telemetry is then collected. Step-wise causal labels are produced automatically by the PAVE pipeline (Sections 2.2, 2.3) and are further checked by the forward verification protocol described later in this appendix.

In a blind audit of 28 instances, 98.3% of the annotated propagation edges were judged correct, and the annotations covered 85.3% of the edges judged real (protocol in Appendix E.1). The authors

conducted the audit with the assistance of LLM agents. No crowdworkers or paid external annotators were used. Because the benchmark is built from synthetic system telemetry rather than data about natural persons, consent, participant notification, and formal human-subject review considerations do not apply.

A.4 Preprocessing, Cleaning, Labeling

Telemetry is exported through OpenTelemetry into standardized parquet schemas, with one file per modality per period. The released benchmark contains only instances with at least one injected fault whose effect PAVE verifies at the service level (Section 2.3); faults without an observable effect are removed from the label, and instances with none left are dropped during pipeline-level analysis. The released package bundles the evaluation telemetry together with the derived labels and propagation graphs, so the released artifact is the processed benchmark itself rather than an undocumented intermediate product.

Cleaning is performed at the instance and fault level: injected faults without an observable effect are removed from the label, instances without any fault that has a service-level effect are excluded, and log lines emitted by the fault-injection tooling itself are removed from the released logs because they would reveal the injection target; no other field of the retained telemetry is edited. The software responsible for preprocessing, filtering, and label construction is the released PAVE pipeline described in Section D. Pipeline reliability rests on the conjunctive verification design analyzed in Section D.8 and is measured by the blind per-edge audit reported in Appendix E.1.

A.5 Uses

Intended uses include (i) process-level evaluation of LLM-based RCA agents and (ii) training process reward models in the reliability domain, where the step-wise edges provide natural supervision targets. At the time of submission, the benchmark is used only in this study as the primary evaluation suite for process-level RCA assessment.

The benchmark is not intended for production-readiness certification of any specific RCA agent, for cross-architecture generalization claims to service-mesh, serverless, or fully event-driven systems beyond the three included testbeds, or for claims about faults that leave no telemetry signature. It should also not be used as a training corpus for the *evaluated* RCA agents themselves, as doing so would create direct test-set leakage.

A.6 Distribution and License

An anonymized preview of the benchmark accompanies the supplementary submission, and the anonymized full benchmark package is provided in the submission materials. Upon acceptance, the full benchmark artifact and the PAVE pipeline will be released publicly with archival metadata and integrity checksums. The released benchmark is a derived artifact constructed from a curated subset of the upstream fault-injection runs together with the additional annotations and causal-process labels introduced in this work. The dataset is released under **CC-BY-SA 4.0**, and the pipeline code under **Apache 2.0**. To the best of our knowledge, no export-control or comparable regulatory restrictions apply to the released artifact.

A.7 Maintenance

To preserve anonymity in the submission version, direct maintainer contact details are omitted here and will be restored upon acceptance. Corrections, additional fault types, and cross-system extensions will be managed through versioned releases, and older public versions will remain archived for reproducibility. After public release, questions and fixes can be tracked through the GitHub issue tracker and pull requests. External contributions will be reviewed for schema compatibility, reproducibility, and annotation consistency before inclusion in an official release.

A.8 REPRODUCIBILITY

The complete PAVE pipeline source is released alongside the dataset, with explicit random seeds for every stochastic component (DFS tie-breaking, baseline-window selection, fault-injection scheduling) so the numbers in this paper are regeneratable; pipeline hyperparameters and detector thresholds are tabulated in Sections D.5 and D.8, and cluster configuration is in Section D.4. A single-entry-point `reproduce.sh` accompanies the anonymized submission artifact and regenerates the Section 3.2 numbers on a downloadable instance subset; full-dataset re-evaluation requires API quota for the 16 models documented in Section F.5. No pre-trained models are released as part of the benchmark.

B BENCHMARK SYSTEMS AND TOPOLOGY

This section details the microservice systems used in our benchmark, including their architecture, scale, and the service dependency topology that defines the structural prior $\mathcal{G}$ for causal path extraction.

B.1 SYSTEMS OVERVIEW

The benchmark spans 3 open-source microservice testbeds chosen to cover distinct propagation regimes: TrainTicket (Zhou et al., 2018) for deep, layered Java/REST call chains; the OpenTelemetry Demo application (Ope, 2026) for polyglot gRPC services with asynchronous Kafka legs; and Hotel Reservation from DeathStarBench (Dea, 2024) for shallow, wide Go/gRPC services with shared in-memory caches. Together they cover three communication styles (synchronous REST, synchronous gRPC, and asynchronous queueing) and four data-store classes (relational, document, key-value cache, and message log), so the recovered ground-truth paths reflect propagation patterns that no single system would expose on its own. The subsections below summarize the topology of each system; deployment manifests, image tags, and commit hashes are released alongside the dataset.

B.2 TRAINTICKET SERVICE DEPENDENCY TOPOLOGY

TrainTicket's 44 microservices follow a layered architecture centered on REST-style service-to-service calls, with RabbitMQ used for asynchronous messaging and a shared MySQL instance supporting many backend services. At a coarse level, requests flow from user-facing entry services through gateway and orchestration layers into domain services and then into shared infrastructure, creating long dependency chains and frequent opportunities for multi-hop propagation.

The system exhibits several properties that make RCA challenging:

- **Deep call chains**: Booking operations traverse up to 6 service layers (Entry → Gateway → Orchestration → Domain → Foundation → Infrastructure).
- **High fan-in**: Foundation services (`Station`, `Route`, `Train`) are shared by many upstream callers, creating complex dependency patterns where a single failure can manifest in multiple symptom locations.
- **Shared infrastructure**: 23 services connect to a shared MySQL 5.7 instance, introducing vertical propagation paths from database-level faults to application-level errors.
- **Circular dependencies**: Some services exhibit mutual dependencies (e.g., `Travel` ↔ `Seat`) that require temporal unrolling for causal analysis.

B.3 OPENTELEMETRY DEMO SERVICE DEPENDENCY TOPOLOGY

The OpenTelemetry Demo is a polyglot e-commerce reference application comprising 15 application services implemented across Go, Python, Java, .NET, Ruby, and JavaScript. The deployed stack also includes supporting components such as Kafka, Postgres, valkey/Redis, and observability infrastructure, which are not counted as application services. Inter-service communication uses gRPC for synchronous request paths (e.g., `cart` → `checkout` → `payment`) and Kafka for asynchronous fan-out (e.g., `checkout` → `accounting` / `fraud-detection`); state is held in Postgres, valkey/Redis, and the Kafka log itself.

Properties relevant for fault propagation:

- **Polyglot error semantics**: each language stack surfaces failures differently (Java stack traces, Go panics, Python tracebacks, .NET exceptions), exercising the rule set across heterogeneous log formats.
- **Synchronous + asynchronous mix**: faults injected upstream of the Kafka boundary propagate as queue back-pressure rather than RPC timeouts, producing a propagation signature absent from the REST-only TrainTicket.
- **Shared infrastructure**: Postgres and valkey are accessed by multiple services; an injected database fault produces correlated anomalies across otherwise unrelated request paths.

B.4 Hotel Reservation Service Dependency Topology

Hotel Reservation, drawn from DeathStarBench (Dea, 2024), consists of 9 Go services that communicate via gRPC and rely on MongoDB and Memcached.

Properties relevant for fault propagation:

- **Shallow but wide call chains**: the maximum service-to-service depth is 3 hops (`frontend` → `search` → `profile`/`geo`/`rate`), with high fan-out at the search service.
- **In-memory cache layer**: Memcached sits between every read-heavy service and its MongoDB backing store; cache faults degrade many downstream services in parallel.
- **Terse error surface**: Go services return gRPC status codes directly rather than wrapping errors in stack traces, so log signals are sparser and trace-side evidence carries a comparatively larger share of the verification load.

B.5 Propagation Channels and Telemetry Modalities

The dependency graph $\mathcal{G}$ supports two propagation channels that the verification pipeline must distinguish. *Horizontal propagation* follows RPC call chains: a slow `Order` service causes its upstream callers to time out, producing cascading latency spikes along the call graph. *Vertical propagation* crosses abstraction layers: CPU throttling on a Kubernetes node degrades all co-located services simultaneously, even those with no mutual RPC dependency. A single injection can trigger both channels, creating multi-hop paths that mix horizontal and vertical edges.

These channels leave distinct signatures in the three telemetry modalities, which Phase 2 verification (Section 2.3) exploits. *Traces* record per-request call trees with span-level latencies and status codes; horizontal propagation manifests as anomalous parent→child span timing that reveals the cascade direction. *Metrics* are per-service and per-host time series (CPU usage, p99 latency, error rate); vertical propagation appears as correlated anomalies among co-located services sharing a throttled resource. *Logs* are semi-structured text events (e.g., `[ERROR] upstream timeout`) that supply causal context absent from numerical signals alone.

C Root Cause Taxonomy

This section formalizes the ground truth label schema and provides the complete fault type catalog.

C.1 Ground Truth Schema

Each fault injection instance is annotated with a structured ground truth label that specifies the root cause at multiple granularity levels. Table 4 shows the schema fields; cardinalities vary by system, and TrainTicket values are shown as concrete examples.

The multi-level annotation enables evaluation at different abstraction levels. Service-level evaluation ($|\mathcal{V}| = 44$) is the most commonly reported granularity in prior work. Finer-grained levels (function, span) provide more precise localization but increase label space complexity substantially.

Table 4: Ground truth label schema. Each fault injection instance is annotated with a structured label across multiple granularity levels. Cardinalities are system-dependent; example values are drawn from TrainTicket for concreteness.

Field	Description	Cardinality	Example Value
`service`	Target microservice	system-dependent	`ts-order-service`
`pod`	Target Kubernetes Pod	system-dependent	`ts-order-service-7b8f4-xk9z2`
`container`	Target container within Pod	system-dependent	`ts-order-service`
`function`	Target function or handler (code-level faults)	system-dependent	`com.order.service.impl.create`
`span`	Target trace span (HTTP faults)	system-dependent	`POST /api/v1/orderservice/order`
`metric`	Anomalous golden signal	system-dependent	`http.server.request.duration`

Table 5: Fault type taxonomy across 7 Chaos Mesh CRD categories. "Applicable Levels" lists the ground-truth fields relevant to each fault type.

Category	Fault Type	Parameters	Target Layer	Applicable Levels	Propagation Channel
PodChaos	PodKill	Kill signal (SIGKILL)	Infrastructure	service, pod	Vertical
	PodFailure	Unavailability duration	Infrastructure	service, pod	Vertical
	ContainerKill	Target container	Infrastructure	service, pod, container	Vertical
StressChaos	CPUStress	Core count, load %	Infrastructure	service, pod	Vertical
	MemoryStress	Memory size (MB)	Infrastructure	service, pod	Vertical
HTTPChaos	HTTPRequestAbort	Abort code, target path	Application	service, span	Horizontal
	HTTPResponseAbort	Abort code, target path	Application	service, span	Horizontal
	HTTPRequestDelay	Delay (ms), target path	Application	service, span	Horizontal
	HTTPResponseDelay	Delay (ms), target path	Application	service, span	Horizontal
	HTTPResponseReplaceBody	Body type {empty, random}	Application	service, span	Horizontal
	HTTPResponsePatchBody	Patch content	Application	service, span	Horizontal
	HTTPRequestReplacePath	New path	Application	service, span	Horizontal
	HTTPRequestReplaceMethod	Method {GET, POST, ...}	Application	service, span	Horizontal
	HTTPResponseReplaceCode	Status code	Application	service, span	Horizontal
DNSChaos	DNSError	Target hostname	Infrastructure	service, pod	Vertical
	DNSRandom	Target hostname	Infrastructure	service, pod	Vertical
NetworkChaos	NetworkDelay	Delay (ms), jitter	Infrastructure	service, pod	Vertical + Horizontal
	NetworkLoss	Loss rate (%)	Infrastructure	service, pod	Vertical + Horizontal
	NetworkDuplicate	Duplicate rate (%)	Infrastructure	service, pod	Vertical + Horizontal
	NetworkCorrupt	Corrupt rate (%)	Infrastructure	service, pod	Vertical + Horizontal
	NetworkBandwidth	Rate (bps)	Infrastructure	service, pod	Vertical + Horizontal
	NetworkPartition	Partition direction	Infrastructure	service, pod	Vertical + Horizontal
TimeChaos	TimeSkew	Offset (ms)	Infrastructure	service, pod	Vertical
JVMChaos	JVMLatency	Delay (ms), target class.method	Application	service, function	Horizontal
	JVMReturn	Return value, type {String, Int}	Application	service, function	Horizontal
	JVMException	Exception class, target method	Application	service, function	Horizontal
	JVMGarbageCollector	—	Application	service, pod	Vertical
	JVMCPUStress	CPU count	Application	service, pod	Vertical
	JVMMemoryStress	Type {heap, stack}, size	Application	service, pod	Vertical
	JVMMySQLLatency	Delay (ms), target method	Application	service, function	Horizontal
	JVMMySQLException	Exception, target method	Application	service, function	Horizontal

C.2 Complete Fault Type Catalog

Table 5 presents the fault types implemented in our chaos engineering framework, organized by Chaos Mesh CRD category. Each fault type is classified by:

- **Target layer**: Whether the fault targets infrastructure resources (Pod, network, CPU/memory) or application logic (HTTP endpoints, JVM methods, database queries).
- **Applicable ground truth levels**: Which fields of the ground truth schema are relevant. Infrastructure faults are labeled at the `service`/`pod` level, while application faults include `function` or `span` annotations.
- **Propagation channel**: Whether the fault propagates horizontally (between logical entities via RPC) or vertically (from infrastructure to logical layers via resource contention), or both.

C.3 Observable Fault-Kind Vocabulary for Evaluation

The chaos-mesh API names in Table 5 describe *how* a fault is injected, not what an agent observes from telemetry; in many cases distinct injection mechanisms produce indistinguishable signatures in the parquet data (for example `HTTPRequestAbort` and `HTTPResponseAbort` both surface as request failures from the client side, and `PodKill` and `ContainerKill` both manifest as a brief gap in the affected service's trace emission). An agent therefore cannot recover the raw chaos-mesh API name from the released telemetry alone. We define an observable fault-kind vocabulary

Table 6: Mapping from raw `chaos_type` (Table 5) to the observable fault-kind labels used at evaluation time. Rows listing multiple source identifiers are many-to-one collapses where the listed mechanisms produce indistinguishable telemetry signatures; the remaining rows map one-to-one.

Fault kind	Source `chaos_type`(s)	Primary observation channel
Pod / container lifecycle		
`pod_failure`	`PodKill`, `ContainerKill`	trace emission gap, brief, recovers in window
`pod_unavailable`	`PodFailure`	trace emission gap, sustained to window end
L3/L4 network (direction recorded in GT)		
`network_delay`	`NetworkDelay`	`container.network.*`, trace latency tail
`network_loss`	`NetworkLoss`	`container.network.*`, trace retry pattern
`network_partition`	`NetworkPartition`	`container.network.*`, timeout share approaches one
`network_corrupt`	`NetworkCorrupt`	`container.network.*`, TCP retransmit signal
`network_duplicate`	`NetworkDuplicate`	`container.network.*`, duplicate-ack pattern
`network_bandwidth_limit`	`NetworkBandwidth`	`container.network.*`, throughput drop
HTTP application layer		
`http_aborted`	`HTTPRequestAbort`, `HTTPResponseAbort`	HTTP span status, request-failure rate
`http_slow`	`HTTPRequestDelay`, `HTTPResponseDelay`	HTTP span duration distribution
`http_payload_modified`	`HTTPResponseReplaceBody`, `HTTPResponsePatchBody`, `HTTPRequestReplacePath`, `HTTPRequestReplaceMethod`	response body / path / method differs from baseline
`http_response_status_modified`	`HTTPResponseReplaceCode`	HTTP span status-code distribution
Resource exhaustion		
`cpu_stress`	`CPUStress`	`container.cpu.*` saturated
`mem_stress`	`MemoryStress`	`container.memory.*` saturated
`jvm_thread_cpu_stress`	`JVMCPUStress`	`jvm.cpu.*` elevated, container CPU normal
`jvm_heap_stress`	`JVMMemoryStress`	`jvm.memory.used` climbs to limit
`jvm_gc_pressure`	`JVMGarbageCollector`	`jvm.gc.duration` histogram spikes
Code-level JVM		
`jvm_method_exception`	`JVMException`	log exception class on a specific method
`jvm_jdbc_exception`	`JVMMySQLException`	exception on a JDBC client call
`jvm_method_latency`	`JVMLatency`	elevated span duration on a method
`jvm_jdbc_latency`	`JVMMySQLLatency`	elevated span duration on a JDBC call
`jvm_method_mutated`	`JVMReturn`, `JVMRuntimeMutator`	method returns a value differing from baseline
DNS / clock		
`dns_resolution_failed`	`DNSError`, `DNSChaos`	DNS error log entries on the affected service
`dns_resolution_wrong`	`DNSRandom`	resolution succeeds but returns wrong target
`clock_skew`	`TimeSkew`, `TimeChaos`	cross-service timestamp drift in correlated logs

by relabeling each chaos mechanism according to the feature signature it produces in metrics, traces, and logs, and judge an agent correct when its predicted label matches this observable label rather than the underlying API name. Table 6 lists the mapping from raw `chaos_type` to observable label together with the primary signal carrying each label; the same projection is applied at ground-truth extraction and at agent-prediction matching so both sides of the comparison share one vocabulary.

D PAVE Pipeline and Experimental Details

This appendix details (i) the pipeline internals compressed in Section 2 for space (Sections D.1–D.3, plus the rule set, hyperparameters, and detector thresholds in Sections D.6, D.5, D.8), and (ii) the operational and calibration details supporting reproduction (Sections D.4–D.8).

D.1 Structural Causal Model

Fault propagation over $\mathcal{G}$ (Pearl, 2009) is realized at each node as

$$S_i^{(t)} = f_i\big(\mathbf{S}_{Pa(v_i)}^{(<t)},\, U_i^{(t)}\big), \tag{2}$$

where $\mathbf{S}_{Pa(v_i)}^{(<t)}$ are the recent states of v_i's causal parents and $U_i^{(t)}$ is an exogenous shock. The agent's observation $\mathbf{O}_{obs} \subset \mathbf{O}_{true}$ is a strict subset of system state (instrumentation gaps, sampling), which is the source of the information asymmetry in inequality (1).

D.2 Phase 1: Discrete State Abstraction Ψ

We project high-dimensional raw telemetry $\mathbf{O}^{(t)} \in \mathbb{R}^d$ onto a discrete state alphabet Σ via a node-type-aware mapping $\Psi(O_i, \theta(v_i)) \to s_i \in \Sigma$, where $\theta(v_i)$ is the type of node v_i (matching the type tag θ_i used in the Phase 1 rule transitions of Section 2.2). For *logical nodes* (services, spans), states capture performance violations such as `HighLatency` and `HighErrorRate`; for *infrastructure nodes* (pods, containers, hosts), states reflect resource exhaustion such as `CPU_Throttled` and `MemoryPressure`. This abstraction filters high-frequency noise (U_i), retaining only semantically meaningful events as inputs to the rule-filtering and DFS stages.

D.3 Phase 1: Time-Expanded Heterogeneous Graph and Constrained DFS

To enumerate candidate paths Π_{cand}, we construct a Time-Expanded Heterogeneous Graph whose nodes are tuples (v, s, t): a system node $v \in \mathcal{V}$ with state $s \in \Sigma$ activated at time t. Edges connect tuples whose state transitions are admitted by the propagation rule set $\mathcal{R}$ (Section D.6) and whose times are causally ordered. A constrained Depth-First Search starting from the injection

point unrolls circular dependencies (common in microservice meshes) into acyclic chains, with the configured maximum path length and per-node revisit budget (Section D.5). Curation uses the earlier annotation to filter out cyclic service graphs (Section A). In the released benchmark, temporal unrolling handles cycles among spans within a service, as well as the service-level cycle in 1 instance. The resulting paths are topologically valid sequences of state transitions; statistical and temporal verification (Section 2.3) then admits or rejects them.

D.4 Cluster Configuration

Our testbed runs on Kubernetes v1.29 with the following specifications:

- **Cluster**: 1 control-plane node (`master1`, 48 vCPU, 64GB RAM) and 6 worker nodes running Debian 12 with containerd 1.7.5; `worker1`–`worker3` each provide 32 vCPU and 64GB RAM, while `worker4`–`worker6` each provide 128 vCPU and 128GB RAM
- **Systems hosted**: TrainTicket (44 Java/Spring services), OpenTelemetry Demo (15 polyglot services), and Hotel Reservation (9 Go services), each deployed in separate Kubernetes namespaces on the shared cluster
- **Service-count convention**: Reported per-system service counts are application `Deployments` obtained via `kubectl get deploy` on the running benchmark, excluding load generators, the OpenTelemetry collector, datastores, message brokers, and the TrainTicket UI dashboard
- **Deployment versions**: Exact system commit hashes and image tags for the three deployments are pinned in the archived manifests released with the benchmark
- **Traffic load**: Synthetic user sessions are generated by per-system load generators after a warm-up period, with sustained request streams used to establish pre-injection baselines and expose post-injection propagation
- **Monitoring stack**: OpenTelemetry-based collection with per-system telemetry export to standardized parquet artifacts, together with cluster-level collectors and Prometheus-based metric scraping
- **Fault injection**: Chaos Mesh with programmatic CRD generation for application, resource, network, and pod-level interventions
- **Stateful backends**: Across the three systems, the benchmark exercises MySQL, PostgreSQL, MongoDB, Memcached/valkey, RabbitMQ, and Kafka-backed propagation paths

D.5 PAVE Hyperparameters

Key parameters for the PAVE annotation pipeline:

- **State Detection Window** Δ: 3 s for span/service-level metrics; 5 s for pod, container, and machine-level metrics
- **State Activation**: Dual detection combines fixed utilization thresholds (>80% CPU/memory) with Z-score baseline comparison ($Z > 3\sigma$ for critical anomalies), plus adaptive multiplier thresholds for latency metrics based on coefficient of variation (CV)
- **Maximum Path Length**: 5 hops (default), with diamond-shaped revisits allowed (max 2 visits per node)
- **Rule Confidence**: Default 0.8 per propagation rule, with per-rule overrides

D.6 Propagation Rule Set

Table 7 lists the complete rule set $\mathcal{R}$ used in Phase 1 (Section 2.2). Each rule specifies the source entity's type and state, the intermediate entities on the path, and the destination entity's type and possible states. A rule is retained only when a known mechanism links the source and destination. We group rules into four categories.

Vertical rules (container→span, pod→span) describe how a failed, throttled, or otherwise impaired container or pod affects its service's request spans through the Kubernetes hierarchy (Container $\subset$ Pod $\subset$ Service). **Structural rules** describe dependencies between containers, pods, and services. A failed container impairs its pod, a failed pod disables its containers, and a service becomes unavailable when its only pod fails. **First-hop rules** (service→span) carry a service-level fault into the service's own spans. **Horizontal rules** (callee span→caller span) describe how a slow, failing, missing, or unresponsive service affects the services that call it.

Table 7: Complete propagation rule set $\mathcal{R}$ (18 rules). *Src*/*Dst* give entity type and states; *Path* lists the intermediate entities of multi-hop rules, which act as structural connectors and may be in any listed state. Rules marked with a label fire only when the metric adapter attaches that label to the source.

Src : States	Path	Dst : States	Mechanism
Vertical: container → span			
container : {unavailable}	pod : {unavailable, degraded, healthy, unknown} → service →	span : {missing, unavailable, erroring}	Process termination
container : {degraded}	pod : {degraded, healthy, unknown} → service →	span : {slow, erroring}	Resource contention
container : {erroring}	pod : {unavailable, degraded, erroring, healthy, unknown} → service →	span : {erroring, slow, missing}	Application error in the container
container : {slow}	pod : {slow, degraded, healthy, unknown} → service →	span : {slow, missing}	Application latency in the container
Vertical: pod → span			
pod : {unavailable}	service →	span : {missing, unavailable, erroring}	No replica serves requests
pod : {degraded}	service →	span : {slow, erroring}	Resource pressure
pod : {degraded}	service →	span : {slow, erroring}	GC pauses (label `frequent_gc`)
pod : {degraded}	service →	span : {slow, erroring}	Heap pressure (label `high_heap_pressure`)
Structural: containment and roll-up			
container : {unavailable}	—	pod : {degraded, unavailable, erroring}	Container failure degrades its pod
container : {unavailable}	—	pod : {unavailable, degraded, erroring}	OOM kill (label `oom_killed`)
pod : {unavailable}	—	container : {unavailable, erroring}	Pod failure takes its containers
pod : {unavailable}	—	service : {unavailable, degraded, erroring}	Only backend of the service is gone
First hop: service → span			
service : {slow, erroring, degraded, unavailable}	—	span : {slow, erroring, missing, unavailable}	Service fault surfaces in its spans
service : {silent}	—	span : {silent, missing}	Isolated service emits no spans
Horizontal: callee span → caller span			
span : {slow}	—	span : {slow, erroring}	Callee slowdown
span : {erroring}	—	span : {erroring, slow}	Error propagation
span : {unavailable, missing}	—	span : {erroring, unavailable, missing}	Missing or unavailable callee
span : {silent}	—	span : {silent, erroring, missing}	Silent callee

Intermediate entities in multi-hop rules, such as the pod and service between a container and its spans, provide structural connections. They can be in any state listed by the rule, including *healthy* and *unknown*, and are collapsed in the service-level graph used for scoring. Three rules apply only when the metric adapter assigns a specific condition label (`frequent_gc`, `high_heap_pressure`, `oom_killed`) to the source entity. The rule set is intentionally broad, and Phase 2 (Section 2.3) removes hops unsupported by monitoring data.

D.7 Scalability and Computational Cost

All experiments were conducted on a virtual machine with 32 CPU cores and 64GB RAM. The path extraction process for a single fault injection scenario, covering a 15-minute telemetry window ($\sim$1GB of metrics and traces), completes in an average of 3.5 minutes. Computational complexity scales with the number of active anomalous entities rather than total system components, demonstrating practical scalability.

D.8 Anomaly Detection Thresholds

The per-node anomaly screen in Phase 2 (Section 2.3) uses three detector families, each matched to the distributional characteristics of the underlying metric.

Z-score detector (for approximately Gaussian metrics such as CPU and memory utilization). A metric value x is flagged if its Z-score $z = (x - \mu_{base})/\max(\sigma_{base}, \epsilon)$ exceeds a threshold, where μ_{base} and σ_{base} are the baseline mean and standard deviation and $\epsilon = 10^{-6}$ prevents division by zero. We use three severity levels: *critical* ($z > 3.0$), *warning* ($z > 2.0$), and *moderate* ($z > 1.5$).

Percentile detector (for heavy-tailed metrics such as error counts). Instead of parametric assumptions, anomalies are flagged relative to empirical baseline percentiles: *critical* if the value exceeds the baseline P99.9 or 2×P99; *warning* if it exceeds P99 or 1.5×P99; *moderate* if it exceeds P90 or 1.2×P99.

Adaptive latency detector (for request latency, which spans orders of magnitude across services). The threshold multiplier is $\tau = (\tau_{cv\text{-}base} + \tau_{cv\text{-}range} \cdot (1 - e^{-cv/\alpha})) \cdot (l_{ref}/l_{base})^{\beta}$, where cv is the baseline coefficient of variation, l_{base} the baseline mean latency, and l_{ref} a reference latency. This formulation encodes the principle that low-baseline services (e.g., 1 ms mean) tolerate large relative spikes with minimal user impact, while high-baseline services (e.g., 5 s mean) require tighter thresholds.

Design rationale. All three detectors are intentionally permissive at the per-node level: they aim to avoid false negatives (missing a genuinely affected node) at the cost of admitting some false positives (flagging coincidental deviations). The combined verification checks reduce false positives by requiring anomalous service states, call evidence for every hop between services, conformance to

$\mathcal{R}$, and temporal ordering, as specified in Section 2.3. This layered design decouples the sensitivity of individual detectors from the overall precision of the ground truth.

Why the checks are combined. Each check can admit spurious edges when used alone. A possible dependency may not carry the fault, an anomaly may be coincidental, and temporal order does not establish a causal link. Requiring all of them removes any edge that fails one. The call structure constrains both the propagation rules and timing, so the checks are not independent. We assess annotation precision through the blind audit (Appendix E.1) without claiming an error bound for the combined checks. Passing the checks establishes consistency with the recorded intervention without proving that every edge is causal. For example, a common cause affecting two services that call each other may satisfy all three checks.

E Audit Protocol and Calibration

This appendix details the blind per-edge audit of the released paths, the trajectory-based failure attribution (Section 3.3), and the calibration split used by the per-node anomaly screen. The verification checks in Section D.8 reduce spurious edges but do not provide an error bound. The audit below assesses the released annotations.

E.1 Blind Per-Edge Audit

Goal. Measure the precision and recall of the released service-level propagation edges independently of the pipeline that produced them.

Sampling. The audit sample contains 30 instances. We randomly selected 15 single-fault and 13 multi-fault instances that had not undergone manual adjudication, using a fixed seed, and added 2 instances where a preliminary review disagreed with the labels. Two instances, one randomly selected and one flagged by disagreement, were subsequently sent to the pipeline developers as bug reports and excluded from the audit, leaving 28 instances. Of these, 25 remain in the released benchmark, while the other three have no fault with a service-level effect and contribute no edge.

Independence. Isolation from pipeline development was partial. One of the 28 instances had been used to test the pipeline during development. The first-round findings informed fixes for unstable baselines and missing intermediate hops, as well as the choice among pipeline versions. Edges added by these revisions were reviewed blind on the same instances. The reported results describe the released labels on these audited instances and do not estimate pipeline error on unseen instances.

Protocol. For each instance, the auditors received information about the injected faults and an unlabelled pool of candidate service edges from several annotation versions. Edges were sorted by name, with their annotation sources hidden. Auditors had SQL access to the raw telemetry of both windows and no access to any annotation. Each candidate edge $u \to v$ was judged Real when v calls u and v's calls to u, or v's requests that depend on u, degrade in the fault window relative to the baseline with an onset no earlier than u's; Not Real when the call exists but v is not affected through u (for example, when a closed-loop load generator throttles every service alike); No Call when neither service calls the other; and Unclear otherwise. Auditors also listed up to three clearly real edges missing from the pool; these count toward recall. Edges added by later pipeline revisions were audited in a follow-up blind round under the same protocol.

Auditors. The authors conducted the audit with the assistance of LLM agents. All verdicts, with the queries and numbers behind them, are released with the benchmark (`data/audit`).

Re-check and author review. The 138 audited edges comprise the released edges from 25 instances and the real edges missing from those annotations. A second blind round re-reviewed these edges against the monitoring data, without access to the labels or first-round verdicts. This round compared anomaly onset with the injection time and examined the caller's requests to other services. The two rounds agree on 134 of the 138 edges. The authors resolved the 4 disputed edges using injection records and monitoring data. Three lie on the injected network link, where the intervention degrades the calls. The fourth follows a container restart caused by JVM memory stress, after which the caller's requests receive no response. All four were judged real. Using a fixed seed, the authors randomly sampled 20 edges on which both rounds agreed and checked them against call statistics

from the raw traces. They confirmed all sampled verdicts. The final verdicts, the rulings, and the checked sample are released with the audit (`data/audit/recheck`).

Result. Under the final audit verdicts, the annotated edges have 98.3% precision (116 of 118 edges; 95% CI 94.0–100.0) and 85.3% recall (116 of 136 true edges; 95% CI 75.9–94.1). The confidence intervals are computed by bootstrapping over instances. The corresponding first-round precision and recall are 97.5% and 85.2%. Recall measures coverage of edges judged real in the candidate pool or identified as missing by the auditors. Treating every UNCLEAR candidate as real lowers recall to 83.5%. The two false positives, both in one instance, are an edge whose slowdown began before the injection and the edge downstream of it. Most missed edges are hops that the call-evidence check could not confirm, such as calls that did not complete, or alternative paths to services already reached in $\mathcal{G}^*$. These omissions mean that agents can lose Edge F1 for predicting real propagation edges absent from $\mathcal{G}^*$.

E.2 TRAJECTORY-BASED FAILURE ATTRIBUTION

Section 3.3 classifies every required root cause an agent missed, using only the verified ground truth and the agent's own trajectory (`scripts/process_eval.py`).

Unit and linking. The unit of analysis is a (model, instance, required root cause) triple. Of the 7455 triples, 53.5% match both service and fault kind, 34.6% identify the service with an incorrect fault kind, and 11.9% miss the service. The analysis below focuses on the last group. Each tool call is linked to its result by its call identifier. A query is counted as retrieving the root-cause service's data when it succeeds and returns rows from that service. For network faults, rows from either endpoint count.

Diagnostic evidence. For a missed root cause, the query results are classified as sufficient when they contain the evidence specified for that fault family. Pod faults require a lifecycle metric or the service's span counts in both the fault and normal windows. Resource faults require the relevant resource metric in both windows. Network, HTTP, and JVM faults require trace records showing the fault's effect on latency, errors, or request volume, with a normal-window comparison for latency and volume. HTTP faults additionally require records from the other endpoint of the call, and JVM faults require method-level spans.

What was reported instead. Predictions corresponding to another valid root cause are set aside, including other required or optional faults and accepted endpoints. Each remaining wrongly reported service is classified, in order of precedence, as the *other endpoint* of the injected call, a service *downstream* of the missed root cause along directed edges of $\mathcal{G}^*$, a service in $\mathcal{G}^*$ not reachable from that root, a service *already anomalous before injection*, or an *unrelated* service. A service is classified as already anomalous if its normal window contains a server error rate $\geq 1\%$, server p95 latency ≥ 1 s, or at least five ERROR log lines.

Validation. After the rules were fixed, a random sample of 23 missed root causes was manually labelled using the same category definitions, with the automatic labels hidden. The review used the trajectories and monitoring data directly. The manual and automatic labels agree on how the root-cause service was examined in 100% of cases and on the categories of services reported instead in 78%. Disagreements mainly concern the boundary between *already anomalous* and *unrelated*, where the automatic rule is conservative. Two limits apply: the query tool truncates results to about ten rows, so a targeted query can return no row for the service; and when a fault's effect is an absence (vanished spans), the anomaly is visible only if the agent compares the fault window with the normal window.

E.3 BASELINE CAL-EVAL SPLIT

For each instance, the pre-injection baseline window $\mathcal{D}_{base}$ is split into two halves: the first calibrates per-node thresholds (Z-score parameters, percentile cutoffs, adaptive-latency multipliers; see Section D.8), the second evaluates them. Calibrating and evaluating on the same window inflates the per-node admission rate because the screen would fit the noise it then tests against; the split prevents this overfit. Default calibration set size is half the baseline window length (~ 7.5 minutes).

F LLM-BASED RCA AGENT DETAILS

This section details the evaluation framework, agent architecture, and prompt design for the LLM-based RCA agents evaluated in our benchmark.

F.1 AGENT EVALUATION FRAMEWORK

Each LLM agent receives a datapack containing multimodal telemetry (traces, metrics, logs in parquet format) and produces a structured CausalGraph: a directed graph with annotated root causes, affected nodes, and causal edges. Evaluation operates at two levels: outcome-level (root cause accuracy) and process-level (causal edge and node fidelity against ground truth). The graph-level metrics complement the trajectory-based failure attribution of Section 3.3.

F.2 AGENT ARCHITECTURE

All agents follow a tool-augmented LLM architecture adapted from the Deep Research agent (Lin, 2025), operating in a ReAct-style (Yao et al., 2022) loop in which the final answer is itself a tool call:

1. **`llm_call`** (Decision Node): The LLM reads the incident description, prior tool observations, and its own reflections, then issues tool calls. It calls `submit_findings` to submit its diagnosis for validation.
2. **`tool_node`** (Execution Node): Executes non-terminal tool calls asynchronously and returns their observations, including any tool errors. It reminds the agent of the turns used and remaining, with stronger reminders as the limit approaches.
3. **`submit_node`** (Validation Node): Validates the arguments of `submit_findings` against the answer requirements (Section F.3). A valid answer ends the run. Validation errors are returned to the LLM as a tool result, allowing it to revise the answer and continue the loop.
4. **`force_submit`**: When the turn cap is reached without a submission, the runtime instructs the LLM to call `submit_findings` on its next response.

The control flow follows:

$$\texttt{START} \to \texttt{llm_call} \xrightarrow{\texttt{tool_calls}} \texttt{tool_node} \to \texttt{llm_call} \to \cdots \xrightarrow{\texttt{submit_findings}} \texttt{submit_node} \to \texttt{END}$$

Tool suite. The agent has access to five tools:

- `list_tables_in_directory`: Discovers available parquet files and their metadata (row/-column counts).
- `get_schema`: Inspects the column names, types, and row count of a parquet file before querying.
- `query_parquet_files`: Executes SQL queries over parquet files (supports SELECT, WHERE, JOIN, aggregations).
- `think_tool`: A reflection tool that records the agent's internal reasoning. The prompt instructs the agent to call this tool after each query to summarize findings and plan the next investigation step.
- `submit_findings`: The terminal tool; its arguments are the final answer (Section F.3).

Budget. One turn is one LLM response, regardless of the number of tools it calls. The prompt recommends about 10 turns, and the runtime requires submission when the 20-turn limit is reached. The 294 runs that reached this limit missed 20.7% of required root causes, compared with 11.5% for the other runs. All models share the same tool definitions and identical API retry logic (exponential backoff with at most 5 retries).

F.3 OUTPUT SCHEMAS: GROUND TRUTH AND AGENT ANSWER

Evaluation uses two distinct schemas: the released verified propagation graph in each case directory, and the structured JSON the agent emits. The ground-truth schema preserves per-node states and onset timestamps for process-level metrics; the agent schema is a smaller, verifiable contract built from `(service, fault_kind)` root-cause claims, propagation edges, and SQL evidence.

Ground-truth representation (released artifact). Each case directory ships a `causal_graph.json` file in the `CausalGraph` schema below. This is the per-case process-level label produced by the PAVE pipeline (Section 2). Every node carries a controlled-vocabulary state and an onset timestamp, so the process-level metrics in Section G can be computed directly from this representation.

SCHEMA Ground-truth CausalGraph JSON


```
{
  "nodes": [
    {"component": "ts-order-service",
     "state": ["HIGH_ERROR_RATE"],
     "timestamp": 1718234567000000000},
    {"component": "ts-travel-service",
     "state": ["TIMEOUT"],
     "timestamp": 1718234569000000000},
    {"component": "ts-station-service",
     "state": ["UNAVAILABLE"],
     "timestamp": 1718234565000000000}
  ],
  "edges": [
    {"source": "ts-station-service",
     "target": "ts-travel-service"},
    {"source": "ts-travel-service",
     "target": "ts-order-service"}
  ],
  "root_causes": [
    {"component": "ts-station-service",
     "state": ["UNAVAILABLE"],
     "timestamp": 1718234565000000000}
  ],
  "alarm_nodes": [
    {"component": "span|loadgenerator::POST /api/order"}
  ]
}
```


The four top-level fields are:

- **`nodes`**: all involved components, each tagged with abnormal state(s) and an onset timestamp.
- **`edges`**: directed causal arcs; an edge (`source`, `target`) means the failure of `source` caused the failure of `target`.
- **`root_causes`**: the subset of `nodes` that initiated the cascade.
- **`alarm_nodes`**: the downstream nodes observed to behave abnormally during the injection window, used as targets for path-reachability scoring.

In the example, `ts-station-service` became `UNAVAILABLE` at t_0, propagating to `ts-travel-service` (`TIMEOUT`) and then `ts-order-service` (`HIGH_ERROR_RATE`), yielding a three-hop path.

Agent answer schema. The agent emits a strict-JSON `AgentRCAOutput` with two flat lists. `root_causes` contains entries of the form (`service`, `fault_kind`, `evidence[]`), where `fault_kind` is drawn from a controlled enum and `evidence` is non-empty. `propagation` contains directed (`from`, `to`) edges, each also with non-empty `evidence`. The schema is uniform across single-fault and hybrid cases; the agent fills `root_causes` with as many entries as the evidence supports.

SCHEMA Agent answer (`AgentRCAOutput`)


```
{
  "root_causes": [
    {"service": "ts-station-service",
     "fault_kind": "pod_unavailable",
```

```
      "evidence": [
        {"kind": "metric",
         "sql": "SELECT time, value FROM abnormal_metrics WHERE
    service_name = 'ts-station-service' AND metric LIKE 'pod_phase'
    ORDER BY time",
         "claim": "ts-station-service pod is in NotReady throughout the
     window"}
        ]}
  ],
  "propagation": [
    {"from": "ts-station-service", "to": "ts-travel-service",
     "evidence": [
       {"kind": "trace",
        "sql": "SELECT span_name, attr.status_code, COUNT(*) FROM
    abnormal_traces WHERE service_name = 'ts-travel-service' AND
    span_name LIKE '%station%' GROUP BY 1,2",
        "claim": "ts-travel-service calls into ts-station-service
    return TIMEOUT"}
      ]}
  ]
}
```

Each `Evidence` item declares (i) its modality (`kind` $\in$ {`metric`,`trace`,`log`}), (ii) the actual DuckDB `sql` it claims is supported by the case parquets, and (iii) a $\leq$20-word `claim` stating what the rows demonstrate. The contract requires every `root_cause` and every `propagation` edge to carry at least one evidence item, and the SQL must be re-executable by the judge against the case directory. Service names are normalized at match time (lowercase, hyphens and underscores stripped) to absorb naming-style drift.

Both schemas feed the evaluation pipeline: the `(service, fault_kind)` multiset and the propagation edge set are extracted from the agent answer, while their counterparts plus the alarm-node set are read off the ground-truth graph. Section G defines the per-axis comparisons.

F.4 Prompt Design

The agent runs a single tool-calling prompt. The system prompt (`RCA_ANALYSIS_SP`) specifies the goal, available data, tools, turn budget, and investigation procedure, and incorporates the answer requirements from the SDK when constructed. The final answer is passed as the arguments of `submit_findings`, keeping investigation and submission within the same tool-calling interface. The prompt below is the one used for every scored run.

F.4.1 System and User Prompts

System: RCA_ANALYSIS_SP

```
You are a Root Cause Analysis (RCA) expert investigating a
microservices incident.

For context, today's date is {date}.

## Goal

Identify the root cause(s) of the SLO violation from the telemetry,
then call the submit_findings tool with your final answer.  There may
be more than one root cause.  Let the data tell you how many faults
there are and where they sit.

## Available data

Parquets in this case directory:

• abnormal_metrics.parquet, abnormal_traces.parquet,
abnormal_logs.parquet

• abnormal_metrics_histogram.parquet, abnormal_metrics_sum.parquet

• normal_metrics.parquet, normal_traces.parquet, normal_logs.parquet
```

Common columns:

• **metrics**: time, metric, value, service_name, attr.k8s.pod.name, attr.k8s.namespace.name

• **metrics_sum**: same shape; carries jvm.cpu.*, jvm.memory.*, jvm.thread.count, jvm.gc.*, container.cpu.*, container.memory.*

• **metrics_histogram**: time, metric, service_name, count, sum, min, max, attr.jvm.gc.action, attr.jvm.gc.name

• **traces**: time, trace_id, span_id, parent_span_id, span_name, service_name, duration, attr.status_code, attr.http.request.method, attr.http.response.status_code

• **logs**: time, trace_id, span_id, level, service_name, message

Tools

1. query_parquet_files: DuckDB SQL on parquets in this case dir.

2. list_tables_in_directory: list parquets.

3. get_schema: column types of a parquet.

4. think_tool: REQUIRED after each query; summarize, plan next step.

5. submit_findings: TERMINAL. Call this exactly ONCE when you are ready to commit to a root cause. The arguments are the final v2 RCA envelope (see schema below). Do NOT call any other tool in the same message as submit_findings. After this call, the run ends.

Hard limits

• Interaction budget: aim for ∼{soft_turns} turns; extend if the evidence genuinely warrants it. Hard cap is {max_turns} turns; the runtime will force a stop and demand submit_findings there.

• One turn = one assistant message you produce, regardless of how many tools it calls. Calling 3 tools in parallel in one message is still 1 turn (and is preferred over 3 sequential turns when the queries don't depend on each other).

• Spend the budget efficiently: list_tables_in_directory once, get_schema on the files you actually plan to query, then spend the rest on query_parquet_files. Batch independent queries in a single turn instead of doing them one-by-one.

Investigation playbook

1. list_tables_in_directory to confirm the parquet files.

2. get_schema on the relevant ones (start with abnormal_traces).

3. Diff abnormal vs normal: error rates, latency, status codes, log levels.

4. Trace the call chain (parent_span_id → span_id) to find the earliest service whose own work, not its dependency's, went wrong.

5. Decide every root cause and every propagation edge. More than one root cause is possible; note each separately when evidence supports it.

6. Call submit_findings with the envelope.

Final-answer schema

The submit_findings tool's arguments ARE the v2 RCA envelope. Follow the rules below verbatim; they are enforced by the validator and any violation triggers a rejection tool_result asking you to fix the call.

{agent_contract} (spliced canonical block; see "Agent Contract" below.)

Notes on calling submit_findings:

• root_causes must be a non-empty list. Even if you are uncertain, commit to the most likely service from the evidence rather than leaving it empty.

- Each root_cause carries service, fault_kind (one enum value), and ≥1 evidence item with kind/sql/claim.
- propagation is optional; omit when you have no edge-level evidence.

USER: RCA_ANALYSIS_UP

```
{incident_description}
```

In the scored runs {soft_turns} and {max_turns} are 10 and 20.

F.4.2 AGENT CONTRACT (SPLICED INTO THE SYSTEM PROMPT)

The system prompt does not define its own output schema or fault-kind vocabulary. Both are pulled at agent-startup from the SDK, so the contract version follows the SDK version automatically. The block below is the canonical text spliced in via `{agent_contract}`; it states the JSON shape of `AgentRCAOutput` (Section F.3), enumerates the `fault_kind` vocabulary, and lists the field, SQL, and hard rules.

SPLICED: agent_contract

```
## Output schema (STRICT JSON, no markdown / commentary)
JSON shape of AgentRCAOutput as in Section F.3:  root_causes[] of
{service, fault_kind, evidence[]}, and propagation[] of {from, to,
evidence[]}, where every evidence is {kind, sql, claim}.
## fault_kind enum (pick exactly one per root_cause)
pod_failure, pod_unavailable, network_delay, network_loss,
network_partition, network_corrupt, network_duplicate,
network_bandwidth_limit, http_aborted, http_slow,
http_payload_modified, http_response_status_modified, cpu_stress,
jvm_thread_cpu_stress, mem_stress, jvm_heap_stress, jvm_gc_pressure,
jvm_method_exception, jvm_jdbc_exception, jvm_method_latency,
jvm_jdbc_latency, jvm_method_mutated, dns_resolution_failed,
dns_resolution_wrong, clock_skew
## Field rules
• Service names must match strings present in the data; do not invent.
• One entry per distinct root cause; do NOT collapse multiple distinct
faults into a single root_cause.
## SQL rules
• DuckDB on this case dir's parquets.
• SELECT only; one statement per sql field.  No DDL/DML/ATTACH/etc.
• Reference parquets via read_parquet('<file>.parquet') with bare
filenames, OR directly FROM abnormal_traces (each *.parquet is mounted
as a same-named view).
• Each evidence.sql MUST be the actual SQL, not a description.
• Each evidence.claim is the ≤20-word natural-language assertion the
SQL rows back.
## Hard rules
• EVERY root_cause carries ≥1 evidence (real runnable SQL).
• EVERY propagation edge carries ≥1 evidence.  Do not emit edges you
cannot back with a concrete trace or metric query.
```

F.4.3 KEY DESIGN CHOICES

- **Answer as a tool call.** The arguments of `submit_findings` contain the final answer and are validated against the evaluator's answer requirements. Validation errors are returned as tool results for the agent to address, keeping submission and revision within the tool-calling interface.
- **Controlled fault-kind vocabulary.** The `fault_kind` enum is observable-effect-based: two chaos primitives whose telemetry is indistinguishable share a kind, and two whose signals are dis-

tinguishable get separate kinds. This lets the matcher score classification (*fault_kind_accuracy*) deterministically without an LLM-as-judge.
- **Mandatory SQL evidence.** Every root-cause or propagation claim cites at least one executable DuckDB query. The judge re-executes these queries on the case data, and SQL Exec measures the fraction that execute successfully and return rows (Section G.2.2). Whether the returned rows support the claims is not scored.
- **No topology leakage.** The agent is *not* given the service dependency graph; service relations must be inferred from trace data, testing investigative ability rather than pattern matching over a known graph.
- **Paired normal/abnormal telemetry.** Both pre- and post-injection parquets ship in the case directory, so the agent can run differential queries (error rates, latency tails, log volumes) on identical schemas.

F.5 Model Configurations

We evaluate 16 frontier LLMs spanning 12 model families from Anthropic, Google, OpenAI, Moonshot, DeepSeek, Alibaba, Zhipu, ByteDance, Xiaomi, Tencent, MiniMax, and StepFun (full list in Table 2).

All models use identical system prompts, tool definitions, and temperature (0.7) for reproducibility.

G Evaluation Protocol and Metric Definitions

We organize our evaluation metrics into two levels: *outcome* (set classification of the predicted root-cause set against ground truth) and *process* (causal reasoning quality on the predicted graph plus evidence executability), complemented by *dataset difficulty* signals that characterize each instance.

Claim-to-Metric Mapping. Each headline claim in Section 3.2 is anchored to a specific metric defined below:

- *Agents locate a correct service far more often than the full root-cause set* (Finding 1): measured by AnySvc against Recall and EM in Table 2.
- *Edge F1 lags Node F1 for every model* (Finding 2): measured by the per-row Node F1 vs. Edge F1 columns and the paired gaps in Table 12.
- *Retrieving data from the faulty service does not ensure sufficient diagnostic evidence* (Section 3.3): assessed through the query-result and failure-attribution analyses (Section E.2).
- *Per-system difficulty is uneven* (Appendix Section H.1): measured by the spread of EM and PR across the TrainTicket / OTel Demo / Hotel Reservation rows of Appendix Table 8.

G.1 Outcome-Level Metrics

The outcome-level metrics grade only the agent's predicted root-cause set against the ground-truth root-cause set, ignoring the predicted causal graph. Let $\hat{R}_q$ denote the set of (service, fault_kind) pairs the agent emits as root causes for case q, R_q^* the corresponding ground-truth set, and $\mathcal{Q}$ the set of all evaluation queries. Service names are normalized as in Section G.2 before comparison; for network-class faults, both endpoints of the netem rule are accepted because the rule is observable on either side.

AnySvc (any service hit, kind-agnostic).

$$\text{AnySvc} = \frac{1}{|\mathcal{Q}|} \sum_{q \in \mathcal{Q}} \mathbb{1}\Big[\exists\, (\hat{s}, \hat{k}) \in \hat{R}_q,\ (s^*, k^*) \in R_q^* : \hat{s} = s^* \Big] \tag{3}$$

AnySvc is satisfied when at least one predicted root-cause service appears in the ground truth, irrespective of the predicted fault kind.

Precision, Recall, F1 on (service, fault_kind) pairs. For each case q, the per-case precision and recall on the pair set are

$$\text{P}_q = \frac{|\hat{R}_q \cap R_q^*|}{|\hat{R}_q|}, \quad \text{R}_q = \frac{|\hat{R}_q \cap R_q^*|}{|R_q^*|}, \quad \text{F1}_q = \frac{2\,\text{P}_q\,\text{R}_q}{\text{P}_q + \text{R}_q} \tag{4}$$

where intersection uses HIT-and-equivalence semantics (a predicted pair counts as matched if there is a ground-truth pair with the same normalized service and an equivalent fault_kind in the equivalence vocabulary of Section C.3). We report the mean of P_q, R_q, and $F1_q$ over $\mathcal{Q}$. Recall captures fractional coverage of the ground-truth pairs and is therefore a strictly finer measurement than AnySvc, which only checks whether at least one service is named.

Exact Match (EM).

$$\text{EM} = \frac{1}{|\mathcal{Q}|} \sum_{q \in \mathcal{Q}} \mathbb{1}\left[\hat{R}_q = R^*_q \right] \tag{5}$$

EM requires the predicted root-cause set to coincide with the ground truth as sets, penalizing both missing and spurious entries.

G.2 Process-Level Metrics

These metrics evaluate the quality of the agent's causal reasoning by comparing the predicted CausalGraph against the verified propagation graph $\mathcal{G}^*$. We decompose the evaluation into *primary* metrics (service-level, deterministic matching) and *secondary* metrics (component-level, with LLM-assisted semantic matching for name normalization).

Loadgen filtering. Edges incident on the synthetic load-generator endpoint are removed from both $\hat{\mathcal{E}}_q$ and $\mathcal{E}^*_q$ before any graph-shape metric (Edge F1, Node F1, Path Reachability) is computed. Loadgen is an evaluation-time client and not part of the system under test, so neither admitting nor omitting loadgen-touching edges should be scored.

G.2.1 Primary Metrics (Service-Level)

All primary metrics operate on service-level representations. Service names are normalized (lowercased, `ts-` prefix stripped, hyphens removed) before comparison.

Edge F1. Let $\hat{\mathcal{E}}$ and $\mathcal{E}^*$ be the service-level edge sets of the predicted graph and the verified propagation graph $\mathcal{G}^*$, respectively:

$$\text{Edge Precision} = \frac{|\hat{\mathcal{E}} \cap \mathcal{E}^*|}{|\hat{\mathcal{E}}|}, \quad \text{Edge Recall} = \frac{|\hat{\mathcal{E}} \cap \mathcal{E}^*|}{|\mathcal{E}^*|} \tag{6}$$

Edge F1 is the harmonic mean.

Node F1. Let $\hat{\mathcal{V}}$ and $\mathcal{V}^*$ be the service-level node sets:

$$\text{Node Precision} = \frac{|\hat{\mathcal{V}} \cap \mathcal{V}^*|}{|\hat{\mathcal{V}}|}, \quad \text{Node Recall} = \frac{|\hat{\mathcal{V}} \cap \mathcal{V}^*|}{|\mathcal{V}^*|} \tag{7}$$

Node F1 measures whether the agent correctly identified the set of affected services.

Boundary case. When both the predicted and ground-truth sets are empty for a metric (e.g., both graphs have zero edges), we assign a perfect score of 1.0, since the agent correctly predicted the absence of that element.

G.2.2 SQL Exec

The agent answer schema (Section F.3) ties every claim to at least one piece of SQL evidence. Every `evidence.sql` in the agent answer is re-executed on a fresh in-memory DuckDB connection with each `*.parquet` of the case directory mounted as a same-named view. The result is one of three statuses: `OK` (executed, ≥ 1 row), `EMPTY` (executed, 0 rows), or `SQL_ERROR` (parse or runtime failure). Per case,

$$\text{SQL Exec} = \frac{|\{e \in \mathcal{E}_{\text{ev},q} : \text{status}(e) = \texttt{OK}\}|}{|\mathcal{E}_{\text{ev},q}|} \tag{8}$$

where $\mathcal{E}_{\text{ev},q}$ is the union of all evidence items in q's root_causes and propagation edges. This axis is mechanical; whether the rows actually back the claim is not scored here. The benchmark score is the mean of the per-instance ratios. An answer with no evidence items scores 0.

G.2.3 Diagnostic Reasoning Axes

Path Reachability (PR). Path reachability measures the fraction of applicable instances in which the agent identifies a correct root-cause service and connects it to a ground-truth alarm through predicted edges also present in $\mathcal{G}^*$.

$$\text{PR} = \frac{1}{|\mathcal{Q}_{PR}|} \sum_{q \in \mathcal{Q}_{PR}} \mathbb{1}\left[\exists\, \hat{r}_q \in \hat{R}_q \cap R_q^*,\ \hat{s}_q \in A_q^* \setminus R_q^* : \hat{r}_q \rightsquigarrow \hat{s}_q \text{ in } (\mathcal{V}, \hat{\mathcal{E}}_q \cap \mathcal{E}_q^*)\right] \tag{9}$$

where $\hat{R}_q$ denotes the set of root-cause services predicted by the agent, and R_q^* and A_q^* denote the ground-truth root-cause and alarm-service sets. $\hat{\mathcal{E}}_q$ and $\mathcal{E}_q^*$ are the predicted and verified directed service-edge sets, and $\rightsquigarrow$ denotes a directed path containing at least one edge. A qualifying path follows the propagation direction, uses only edges in $\mathcal{G}^*$, and ends at an alarm outside the root-cause set. $\mathcal{Q}_{PR}$ contains the 425 instances with an alarm outside the root-cause set. For network faults, either endpoint can serve as the starting service, as in the outcome metrics. The main-table PR checks the starting service without requiring a correct fault kind. On $\mathcal{Q}_{PR}$, PR is upper-bounded by AnySvc. The HIT-anchored variant (Appendix Table 11) additionally requires a correct fault kind and is upper-bounded by the share of instances with at least one HIT. Regression checks for this definition are in `scripts/test_path_reach.py`.

H Auxiliary Evaluation Tables

H.1 Per-system breakdown

Table 8 expands the pooled main-paper table (Table 2) into per-system rows for TrainTicket, OTel Demo, and Hotel Reservation. The pooled **All** row of each model matches the corresponding row in the main table.

The per-system spread is wide enough to dominate the cross-model spread on the easier metrics. On Hotel Reservation (148 cases, 9 services, mean longest path 1.4 hops) Claude Opus 5 reaches 100.0% AnySvc and 98.0% EM, while on TrainTicket (274 cases, 44 services, mean longest path 2.5 hops) the same model drops to 89.1% AnySvc and 53.3% EM; OpenTelemetry Demo (32 cases, 15 services, 2.3 hops) sits between, at 96.9% AnySvc and 62.5% EM. Both service count and chain depth track the difficulty ordering in the same direction. Finding (2) in Section 3.2 suggests that errors in identifying affected services contribute to the gap between Node F1 and Edge F1. Longer propagation paths include more services that an agent may miss. Pooled scores across systems with disparate chain depth are therefore sensitive to system-mix manipulation; the per-system rows below are the direct read for any narrow-regime comparison.

H.2 HIT-anchored Path Reachability

Table 11 reports HIT-anchored PR, which requires the starting root-cause claim to match both the service and the fault kind. The main-table PR (Table 2) accepts a correct starting service even when the predicted fault kind is wrong.

H.3 Statistical Robustness of the Main Table

Every cell of Table 2 carries a case-level percentile bootstrap 95% CI. Because all models are scored on the same instances, comparisons between two cells are tested on per-case differences (paired bootstrap), which is better powered than comparing two marginal CIs. Table 12 applies this test in two directions.

Within a model, the Node F1 to Edge F1 gap has a 95% CI strictly above zero for every model, and for every one of the 37 models we evaluated, including the previous generations in Tables 9 and 10. *Across models*, the column leader is significantly ahead of most of the field, but most adjacent rows are not significantly ordered. Table 2 therefore supports statements about tiers and about the gap between metric levels, not a strict order between neighbouring models.

Table 8: Per-system breakdown of Table 2 (TrainTicket, OTel Demo, Hotel Reservation; pooled aggregate in the **All** row; percentages).

		Outcome					*Process*			
Model	System	EM↑	F1↑	Precision↑	Recall↑	AnySvc↑	PR↑	Node F1↑	Edge F1↑	SQL Exec↑
Claude Opus 5	TrainTicket	53.3	55.6	55.4	56.4	89.1	85.9	88.4	79.0	98.3
	OTel Demo	62.5	70.3	68.2	75.0	96.9	88.5	90.0	85.4	98.1
	Hotel Res.	98.0	98.9	98.6	99.3	100.0	99.3	95.8	92.9	96.2
	All	**68.5**	**70.7**	**70.4**	**71.7**	**93.2**	**90.4**	**90.9**	**83.9**	**97.6**
Claude Fable 5	TrainTicket	48.2	49.5	49.5	49.8	87.6	84.4	87.3	78.9	99.6
	OTel Demo	62.5	67.7	66.1	71.9	87.5	92.3	86.8	83.0	99.6
	Hotel Res.	97.3	97.6	97.6	97.6	99.3	99.3	99.1	99.5	99.7
	All	**65.2**	**66.5**	**66.3**	**67.0**	**91.4**	**89.6**	**91.1**	**85.8**	**99.6**
GPT-5.6 Sol	TrainTicket	44.9	46.5	46.5	46.9	84.7	80.6	80.6	69.7	99.8
	OTel Demo	81.2	83.3	82.8	84.4	96.9	42.3	80.1	73.5	99.2
	Hotel Res.	99.3	99.8	99.7	100.0	100.0	99.3	97.0	97.2	99.9
	All	**65.2**	**66.5**	**66.4**	**66.9**	**90.5**	**84.2**	**85.8**	**78.7**	**99.8**
Grok 4.6	TrainTicket	39.4	41.5	41.4	42.0	93.1	90.9	89.9	83.1	98.1
	OTel Demo	68.8	75.0	73.4	78.1	96.9	73.1	87.4	81.7	95.9
	Hotel Res.	98.6	98.6	98.6	98.6	100.0	99.3	96.5	96.1	97.8
	All	**60.8**	**62.5**	**62.3**	**63.0**	**95.6**	**92.5**	**91.9**	**87.2**	**97.8**
Gemini 3.7 Flash	TrainTicket	36.9	37.1	37.2	37.0	87.6	84.4	84.7	74.8	99.2
	OTel Demo	65.6	65.6	65.6	65.6	81.2	80.8	84.2	77.4	99.6
	Hotel Res.	96.6	97.0	97.0	97.0	100.0	99.3	98.6	98.8	99.8
	All	**58.4**	**58.6**	**58.7**	**58.6**	**91.2**	**88.9**	**89.1**	**82.6**	**99.4**
Hy4 Preview	TrainTicket	29.6	31.8	31.6	32.5	91.6	88.6	87.6	79.4	98.4
	OTel Demo	71.9	76.0	75.0	78.1	96.9	92.3	88.2	83.9	99.4
	Hotel Res.	97.3	97.3	97.3	97.3	100.0	99.3	97.1	96.3	99.3
	All	**54.6**	**56.3**	**56.1**	**56.8**	**94.7**	**92.2**	**90.7**	**85.1**	**98.8**
Kimi K3	TrainTicket	32.1	33.9	33.7	34.5	89.4	85.9	85.1	75.0	99.7
	OTel Demo	46.9	49.0	48.4	50.0	84.4	76.9	84.6	78.3	98.9
	Hotel Res.	95.3	95.6	95.6	95.6	98.0	99.3	98.4	98.8	99.4
	All	**53.7**	**55.1**	**54.9**	**55.5**	**91.9**	**89.6**	**89.3**	**82.9**	**99.6**
Claude Sonnet 5	TrainTicket	27.7	29.3	29.2	29.7	86.1	81.4	85.0	73.2	97.9
	OTel Demo	53.1	55.2	54.7	56.2	93.8	69.2	85.2	77.5	98.3
	Hotel Res.	97.3	98.1	98.0	98.3	100.0	99.3	98.7	99.3	97.5
	All	**52.2**	**53.6**	**53.4**	**54.0**	**91.2**	**86.4**	**89.4**	**81.8**	**97.8**
Seed 2.1 Pro	TrainTicket	13.5	16.4	16.0	17.5	76.3	70.0	75.5	59.2	95.9
	OTel Demo	31.2	32.8	32.3	34.4	75.0	80.8	77.1	68.7	96.4
	Hotel Res.	92.6	92.9	92.9	92.9	96.6	97.8	97.2	95.9	97.1
	All	**40.5**	**42.5**	**42.2**	**43.3**	**82.8**	**79.5**	**82.5**	**71.5**	**96.3**
Muse Spark 1.2	TrainTicket	13.1	15.8	15.3	17.0	74.1	62.7	75.0	54.4	98.5
	OTel Demo	31.2	39.6	37.5	43.8	75.0	53.8	74.6	59.4	97.7
	Hotel Res.	89.2	91.3	90.9	92.2	96.6	97.8	97.4	95.8	97.8
	All	**39.2**	**42.1**	**41.5**	**43.4**	**81.5**	**73.4**	**82.1**	**67.9**	**98.2**
MiniMax M3	TrainTicket	12.0	15.2	14.5	17.0	72.3	55.3	70.9	48.0	94.6
	OTel Demo	31.2	35.4	34.4	37.5	68.8	53.8	73.4	61.7	95.9
	Hotel Res.	91.2	92.0	91.9	92.2	95.9	97.8	96.3	95.4	95.2
	All	**39.2**	**41.7**	**41.2**	**43.0**	**79.7**	**68.9**	**79.2**	**64.1**	**94.9**
MiMo 2.5 Pro	TrainTicket	13.5	17.6	16.5	19.7	78.1	67.9	75.0	59.2	91.3
	OTel Demo	31.2	33.3	32.8	34.4	65.6	57.7	73.7	63.7	83.5
	Hotel Res.	86.5	90.9	89.9	92.9	97.3	98.5	94.9	92.6	90.0
	All	**38.5**	**42.6**	**41.6**	**44.6**	**83.5**	**77.1**	**81.3**	**70.2**	**90.3**
DeepSeek V4 Flash	TrainTicket	19.3	23.6	22.6	25.5	85.0	77.6	80.8	64.7	92.3
	OTel Demo	37.5	42.9	41.4	46.9	84.4	88.5	80.3	73.4	87.0
	Hotel Res.	73.0	81.0	79.4	84.5	98.6	99.3	94.2	90.6	87.4
	All	**38.1**	**43.7**	**42.5**	**46.3**	**89.4**	**85.2**	**85.0**	**73.5**	**90.3**
GLM 5.3	TrainTicket	19.3	24.8	23.5	27.7	88.7	83.3	82.4	68.8	96.5
	OTel Demo	40.6	49.0	46.9	53.1	90.6	80.8	80.2	69.4	93.9
	Hotel Res.	60.8	83.4	77.8	94.9	98.0	93.4	94.2	88.5	97.9
	All	**34.4**	**45.6**	**42.8**	**51.4**	**91.9**	**86.4**	**86.0**	**75.1**	**96.8**
Step 3.7 Flash	TrainTicket	9.5	11.1	10.6	12.4	71.2	60.1	67.5	48.8	92.3
	OTel Demo	28.1	28.1	28.1	28.1	59.4	57.7	67.9	55.7	88.6
	Hotel Res.	67.3	70.2	70.0	71.1	93.2	94.1	91.6	86.9	92.4
	All	**29.6**	**31.5**	**31.1**	**32.6**	**77.5**	**70.8**	**75.2**	**61.4**	**92.1**
Qwen3.8-Max	TrainTicket	13.9	23.4	21.2	28.1	90.5	84.8	83.1	72.1	98.8
	OTel Demo	18.8	49.0	41.1	65.6	93.8	76.9	83.4	77.3	99.2
	Hotel Res.	43.9	80.3	71.3	98.3	100.0	99.3	98.3	98.2	98.9
	All	**24.0**	**43.7**	**38.9**	**53.6**	**93.8**	**88.9**	**87.9**	**80.8**	**98.9**

Table 9: Per-system results for previous-generation models (part 1 of 2; same layout as Table 8).

		Outcome					*Process*			
Model	System	EM↑	F1↑	Precision↑	Recall↑	AnySvc↑	PR↑	Node F1↑	Edge F1↑	SQL Exec↑
Claude Opus 4.8	TrainTicket	24.5	25.2	25.4	25.2	81.8	67.7	77.3	58.1	99.1
	OTel Demo	46.9	55.2	53.1	59.4	84.4	80.8	78.5	66.6	98.5
	Hotel Res.	95.3	95.6	95.6	95.6	99.3	98.5	98.5	99.3	97.8
	All	**49.1**	**50.3**	**50.2**	**50.6**	**87.7**	**78.4**	**84.1**	**71.8**	**98.7**
Claude Opus 4.7	TrainTicket	19.7	20.7	20.8	20.8	83.9	66.9	77.4	58.3	98.5
	OTel Demo	50.0	54.2	53.1	56.2	78.1	69.2	75.3	63.5	97.2
	Hotel Res.	91.9	92.7	92.6	92.9	96.6	96.3	97.6	98.2	97.2
	All	**45.4**	**46.5**	**46.5**	**46.8**	**87.7**	**76.5**	**83.7**	**71.4**	**98.0**
Claude Opus 4.6	TrainTicket	26.6	27.6	27.6	27.7	83.9	79.5	85.0	72.1	97.9
	OTel Demo	59.4	59.4	59.4	59.4	93.8	76.9	80.9	74.1	92.4
	Hotel Res.	92.6	93.8	93.6	94.3	97.3	99.3	99.3	98.3	92.5
	All	**50.4**	**51.4**	**51.3**	**51.7**	**89.0**	**85.6**	**89.3**	**80.6**	**95.8**
Claude Sonnet 4.6	TrainTicket	19.7	22.3	21.8	23.5	83.2	73.8	78.9	63.4	97.2
	OTel Demo	43.8	45.8	45.3	46.9	87.5	80.8	79.8	74.2	94.1
	Hotel Res.	91.9	92.7	92.6	92.9	96.6	98.5	98.3	98.6	93.7
	All	**44.9**	**46.9**	**46.5**	**47.8**	**87.9**	**82.1**	**85.2**	**75.3**	**95.8**
Gemini 3.1 Pro	TrainTicket	22.6	23.3	23.5	23.2	82.1	76.0	79.3	65.0	96.4
	OTel Demo	43.8	43.8	43.8	43.8	68.8	34.6	72.4	63.0	97.0
	Hotel Res.	91.9	92.3	92.2	92.6	97.3	98.5	96.7	95.9	98.6
	All	**46.7**	**47.2**	**47.4**	**47.2**	**86.1**	**80.7**	**84.4**	**74.7**	**97.2**
Gemini 3.5 Flash	TrainTicket	22.3	24.2	24.1	24.8	77.0	65.8	72.0	56.2	96.9
	OTel Demo	43.8	43.8	43.8	43.8	62.5	65.4	69.0	61.0	98.2
	Hotel Res.	92.6	93.4	93.6	93.2	97.3	97.8	97.3	97.1	98.3
	All	**46.7**	**48.1**	**48.1**	**48.5**	**82.6**	**76.0**	**79.9**	**69.6**	**97.4**
GPT-5.5	TrainTicket	24.5	25.4	25.5	25.4	85.4	82.9	82.2	72.2	99.4
	OTel Demo	56.2	60.4	59.4	62.5	93.8	80.8	79.2	72.3	100.0
	Hotel Res.	96.6	97.0	97.0	97.0	100.0	99.3	98.0	98.7	99.6
	All	**50.2**	**51.2**	**51.2**	**51.3**	**90.7**	**88.0**	**87.1**	**80.7**	**99.5**
GPT-5.4	TrainTicket	24.5	25.9	25.9	26.1	86.5	81.7	81.5	70.4	99.1
	OTel Demo	62.5	62.5	62.5	62.5	87.5	46.2	70.5	61.8	97.4
	Hotel Res.	95.9	96.3	96.3	96.3	99.3	99.3	94.5	92.9	99.6
	All	**50.4**	**51.4**	**51.4**	**51.5**	**90.7**	**85.2**	**84.9**	**77.0**	**99.1**
Kimi K2.6	TrainTicket	25.9	26.6	26.5	26.8	81.4	76.0	80.2	67.2	97.3
	OTel Demo	53.1	59.4	57.8	62.5	90.6	69.2	78.2	67.5	97.5
	Hotel Res.	91.9	92.7	92.6	92.9	97.3	98.5	96.4	96.1	98.7
	All	**49.3**	**50.4**	**50.2**	**50.9**	**87.2**	**82.8**	**85.2**	**76.5**	**97.8**
Kimi K2.7 Code	TrainTicket	17.5	19.2	19.2	19.5	76.6	68.7	74.4	58.2	96.9
	OTel Demo	28.1	30.2	29.7	31.2	65.6	53.8	76.1	63.7	96.9
	Hotel Res.	89.9	91.1	91.1	91.6	95.3	97.1	93.4	91.4	96.7
	All	**41.9**	**43.4**	**43.4**	**43.8**	**81.9**	**76.9**	**80.6**	**69.2**	**96.9**
Qwen3.7-Max	TrainTicket	17.5	20.5	19.8	21.9	87.6	84.4	80.8	72.5	97.7
	OTel Demo	40.6	49.0	46.9	53.1	78.1	80.8	81.5	75.4	95.7
	Hotel Res.	85.1	92.7	90.9	96.3	99.3	99.3	93.7	90.4	91.4
	All	**41.2**	**46.0**	**44.9**	**48.3**	**90.7**	**88.9**	**85.0**	**78.4**	**95.5**

Table 10: Per-system results for previous-generation models (part 2 of 2; same layout as Table 8).

		Outcome					*Process*			
Model	System	EM↑	F1↑	Precision↑	Recall↑	AnySvc↑	PR↑	Node F1↑	Edge F1↑	SQL Exec↑
Qwen3.7-Plus	TrainTicket	12.8	18.6	17.1	21.9	76.3	70.6	74.1	60.3	97.5
	OTel Demo	25.0	38.5	34.9	46.9	68.8	57.7	67.3	60.0	96.0
	Hotel Res.	77.7	84.8	83.1	88.2	98.6	99.3	94.3	93.1	95.3
	All	**34.8**	**41.6**	**39.9**	**45.3**	**83.0**	**79.0**	**80.1**	**70.8**	**96.7**
Qwen3.6-Max	TrainTicket	18.6	21.5	20.8	22.8	79.9	76.8	81.3	69.4	97.0
	OTel Demo	56.2	58.3	57.8	59.4	75.0	80.8	79.8	71.5	94.3
	Hotel Res.	85.8	91.1	90.5	92.9	97.3	97.8	94.0	91.7	90.8
	All	**43.2**	**46.8**	**46.1**	**48.2**	**85.2**	**83.8**	**85.2**	**76.7**	**94.8**
Qwen3.6-Plus	TrainTicket	13.9	14.6	14.4	15.0	69.3	65.1	73.4	57.4	90.5
	OTel Demo	31.2	33.3	32.8	34.4	53.1	68.0	70.7	63.4	88.5
	Hotel Res.	91.9	92.6	92.6	92.6	98.0	98.5	92.1	88.5	90.2
	All	**40.5**	**41.3**	**41.2**	**41.6**	**77.5**	**76.1**	**79.2**	**67.8**	**90.3**
GLM 5.2	TrainTicket	17.5	20.5	19.9	21.7	83.6	76.8	78.0	62.8	95.7
	OTel Demo	37.5	45.8	43.8	50.0	75.0	65.4	74.4	63.2	96.2
	Hotel Res.	94.6	95.4	95.3	95.6	98.6	98.5	95.4	93.5	95.4
	All	**44.1**	**46.7**	**46.1**	**47.8**	**87.9**	**83.1**	**83.3**	**72.7**	**95.6**
GLM 5.1	TrainTicket	13.9	15.7	15.2	17.0	83.9	76.4	80.1	64.8	93.3
	OTel Demo	34.4	40.6	39.1	43.8	84.4	84.6	77.3	72.1	93.0
	Hotel Res.	89.9	91.1	90.8	91.9	98.6	98.5	92.7	88.0	95.3
	All	**40.1**	**42.0**	**41.5**	**43.3**	**88.8**	**84.0**	**83.9**	**72.7**	**93.9**
Seed 2.0 Pro	TrainTicket	11.3	13.9	13.4	15.0	74.5	69.6	78.3	60.3	93.3
	OTel Demo	15.6	17.7	17.2	18.8	46.9	46.2	65.0	52.2	96.1
	Hotel Res.	62.2	64.8	64.9	64.9	93.9	96.3	90.6	83.8	97.5
	All	**28.2**	**30.7**	**30.5**	**31.5**	**78.9**	**76.7**	**81.4**	**67.3**	**94.8**
Seed 2.0 Lite	TrainTicket	6.9	8.2	7.8	8.8	62.8	51.3	67.8	47.1	96.1
	OTel Demo	15.6	17.7	17.2	18.8	43.8	42.3	63.9	49.7	93.5
	Hotel Res.	73.0	76.5	76.7	77.0	87.8	89.7	89.9	83.0	97.2
	All	**29.1**	**31.1**	**30.9**	**31.7**	**69.6**	**63.1**	**74.6**	**58.8**	**96.3**
Hunyuan 3.0	TrainTicket	6.9	12.4	11.0	15.3	65.3	54.2	63.8	45.2	88.5
	OTel Demo	25.0	37.5	34.0	46.9	65.6	46.2	65.7	52.0	81.7
	Hotel Res.	69.6	78.6	76.5	82.8	96.6	96.3	92.5	87.4	94.7
	All	**28.6**	**35.7**	**34.0**	**39.5**	**75.6**	**67.3**	**73.1**	**59.1**	**90.1**
Hunyuan 3.0 Taiji	TrainTicket	6.6	11.8	10.4	15.0	65.3	55.6	61.0	42.9	90.5
	OTel Demo	12.5	26.2	22.7	34.4	53.1	53.8	65.2	49.1	88.9
	Hotel Res.	52.0	73.5	68.6	83.8	97.3	98.5	92.4	88.0	93.3
	All	**21.8**	**32.9**	**30.2**	**38.8**	**74.9**	**69.4**	**71.3**	**57.7**	**91.3**
MiniMax M2.7	TrainTicket	9.5	11.6	11.3	12.6	66.1	43.1	65.2	42.6	94.0
	OTel Demo	9.4	9.4	9.4	9.4	34.4	23.1	57.7	44.2	91.9
	Hotel Res.	61.5	62.6	62.5	62.8	83.8	68.1	81.7	66.7	91.5
	All	**26.4**	**28.1**	**27.8**	**28.7**	**69.6**	**49.9**	**69.9**	**50.4**	**93.1**

Table 11: HIT-anchored Path Reachability (PR^{HIT}) per (model, system). PR^{HIT} is upper-bounded by the share of instances with at least one HIT. Table 2 reports AnySvc-anchored PR.

Model	TrainTicket	OTel Demo	Hotel Res.	**All**
Claude Opus 5	55.5	69.2	98.5	**70.1**
Claude Fable 5	49.4	76.9	98.5	**66.8**
GPT-5.6 Sol	44.9	38.5	99.3	**61.9**
Grok 4.6	42.6	61.5	97.8	**61.4**
Gemini 3.7 Flash	36.1	69.2	97.1	**57.6**
Hy4 Preview	32.7	76.9	97.1	**56.0**
Kimi K3	35.0	50.0	97.8	**56.0**
Claude Sonnet 5	28.5	42.3	98.5	**51.8**
Seed 2.1 Pro	16.0	38.5	95.6	**42.8**
Muse Spark 1.2	15.2	38.5	94.9	**42.1**
MiniMax M3	12.6	34.6	94.9	**40.3**
MiMo 2.5 Pro	17.9	30.8	94.9	**43.4**
DeepSeek V4 Flash	23.6	53.8	85.3	**45.2**
GLM 5.3	27.0	50.0	91.2	**48.9**
Step 3.7 Flash	11.0	26.9	72.6	**31.6**
Qwen3.8-Max	26.6	57.7	98.5	**51.5**

Table 12: Paired case-level bootstrap over the 454 instances (10,000 resamples). **Top:** the within-model Node F1 to Edge F1 gap with 95% CIs, sorted by the Node F1 to Edge F1 gap. **Bottom:** cross-model ordering in Table 2.

Model	Node F1 − Edge F1	95% CI
MiniMax M3	+15.2	[+13.4, +16.9]
Muse Spark 1.2	+14.1	[+12.6, +15.7]
Step 3.7 Flash	+13.8	[+12.3, +15.5]
DeepSeek V4 Flash	+11.5	[+10.1, +12.9]
MiMo 2.5 Pro	+11.1	[+9.7, +12.5]
Seed 2.1 Pro	+11.0	[+9.7, +12.3]
GLM 5.3	+10.9	[+9.5, +12.3]
Claude Sonnet 5	+7.6	[+6.5, +8.7]
Qwen3.8-Max	+7.2	[+6.0, +8.3]
GPT-5.6 Sol	+7.1	[+6.1, +8.1]
Claude Opus 5	+7.0	[+6.0, +8.0]
Gemini 3.7 Flash	+6.5	[+5.6, +7.4]
Kimi K3	+6.4	[+5.5, +7.4]
Hy4 Preview	+5.6	[+4.7, +6.4]
Claude Fable 5	+5.3	[+4.4, +6.2]
Grok 4.6	+4.7	[+3.8, +5.5]

	EM	F1	AnySvc	PR	Node F1	Edge F1	SQL Exec
Adjacent pairs significantly ordered	5/15	5/15	1/15	1/15	5/15	4/15	2/15
Models significantly below the leader (of 15)	14	15	13	13	14	15	12

H.4 Run-to-Run Variance

The confidence intervals in Table 2 bound case sampling but not model-side sampling. To bound the latter, we repeated the full evaluation pipeline eight times with Seed 2.0 Pro on a 50-case random subsample of OpenRCA 2.0, holding the prompt, tools, output schema, and temperature fixed; all 400 rollouts completed. These runs predate the final release of the ground truth and judge used for Table 2, so their levels are not comparable to the main table; the quantity of interest is their spread. Table 13 reports the spread. At 50 cases, case sampling alone moves a metric by several points, so the spread in Table 13 is an upper bound on the run-to-run component rather than an estimate of it.

Table 13: Eight repeated runs of the full pipeline (Seed 2.0 Pro, 50-case random subsample; prompt, tools, schema, and temperature held fixed). All values in percent.

	EM	F1	AnySvc	PR	Node F1	Edge F1	SQL Exec
Mean over 8 runs	15.2	25.6	83.2	68.2	67.3	46.4	92.6
Standard deviation	1.5	2.4	3.8	5.6	1.9	2.7	1.3
Min	12.0	22.7	78.0	62.0	65.8	42.5	91.0
Max	16.0	29.5	88.0	80.0	70.6	50.2	94.7

H.5 Inference Cost

Table 14 reports what one diagnosis costs under the shared harness. Classical rankers such as random walk (Yu et al., 2021) or graph neural rankers (Zhang et al., 2025b) cost orders of magnitude less at inference, but they emit a ranked list rather than a causal derivation, so the process-level metrics have nothing to score on them. We do not report wall-clock time: it is dominated by provider-side serving throughput and queueing rather than by the reasoning the task requires, so it would compare API endpoints rather than models.

Table 14: Per-instance inference cost over the 454 instances under the shared harness, sorted by mean tokens. Tokens count prompt and output, with cached prompt tokens counted once; the mean over models is 0.49M tokens per instance.

Model	LLM calls	Output tokens	Tokens (mean)	Tokens (median)
GPT-5.6 Sol	15.2	120.2k	1.88M	1.78M
DeepSeek V4 Flash	19.4	16.4k	0.65M	0.66M
Claude Opus 5	14.4	32.7k	0.61M	0.58M
Hy4 Preview	16.1	19.6k	0.50M	0.49M
Gemini 3.7 Flash	17.2	15.2k	0.49M	0.45M
Grok 4.6	12.2	20.0k	0.46M	0.42M
Claude Sonnet 5	13.4	27.5k	0.44M	0.40M
Step 3.7 Flash	17.8	27.9k	0.42M	0.40M
Qwen3.8-Max	16.2	24.2k	0.41M	0.39M
Seed 2.1 Pro	17.7	10.8k	0.33M	0.34M
MiniMax M3	15.0	11.2k	0.29M	0.28M
MiMo 2.5 Pro	12.5	12.1k	0.29M	0.26M
GLM 5.3	15.9	15.2k	0.28M	0.27M
Muse Spark 1.2	19.9	20.2k	0.27M	0.25M
Claude Fable 5	9.2	19.5k	0.24M	0.20M
Kimi K3	13.1	12.4k	0.24M	0.21M

I Use of Large Language Models

We used LLMs to polish the writing of this paper and to assist the authors in checking the data (Appendix E.1). The authors reviewed all AI-assisted work and take responsibility for the content of this paper.