

SLS-BENCH: A Benchmark for Incident Log Summarization with Synthetic Observability Data

Valerii Iakovlev Markus Spanring Martin Flechl

Dynatrace Research

{valerii.iakovlev, markus.spanring, martin.flechl}@dynatrace.com

Abstract

As IT systems grow in complexity, automated agent-driven incident response becomes increasingly critical. A key capability for these agents is summarizing large log streams into human-readable reports. Further progress requires meaningfully evaluating this function, yet existing benchmarks leave end-to-end summarization from large log streams underexplored due to the prohibitive cost and complexity of collecting real-world data. We introduce SLS-BENCH, a benchmark for incident log summarization featuring 90 tasks grounded in real-world incident reports, each with a synthetic log stream and a reference summary. To construct SLS-BENCH, we developed a data generation pipeline that converts incident reports into structured incident models, which guide the generation of synthetic logs and reference summaries. This pipeline combines narrative grounding, explicit incident modeling, generator-verifier loops, and statistical comparisons with real-world log streams to produce realistic and controllable observability data. Our evaluation of 14 LLMs on SLS-BENCH shows that while frontier models can produce useful summaries, they do so at substantial cost and latency, and open-weight models lag significantly. SLS-BENCH provides a framework for systematic evaluation of log analysis agents and a scalable blueprint for future synthetic observability benchmarks.

1 Introduction

Modern IT systems continuously generate large volumes of logs, which become essential diagnostic assets during system incidents. Diagnosing an incident requires operators to sift through vast amounts of log data, identify relevant entries, reconstruct the failure trajectory across distributed components, and distill their findings into a coherent incident summary. This is one of the most impactful yet least automated log analysis tasks: it remains tedious, time-consuming, and heavily reliant on domain expertise. Agentic workflows powered by large language models (LLMs) promise to streamline this process by navigating logs, surfacing the relevant entries, and drafting summaries. Operators can then verify and refine these summaries, enabling faster incident resolution. However, whether current agents can analyze incident logs and produce comprehensive high-quality summaries remains an important open question that existing benchmarks are not fully equipped to answer.

A benchmark for incident log summarization should reflect conditions that operators face in practice: large log volumes, diverse systems and failure types, realistic failure scenarios, and tasks that require comprehensive, structured incident summaries. To our knowledge, no existing benchmark covers all four properties, leaving this end-to-end task largely unevaluated. Prior benchmarks simplify at least one dimension, for example by using short log snippets, narrow task formulations, or keyword-style outputs, rather than requiring an agent to identify a failure, trace its evolution across components, and produce a summary an operator would write after analyzing an incident. This gap persists because

Code: github.com/dynatrace-research/sls-bench Data: huggingface.co/datasets/dynatrace-oss/SLS-Bench

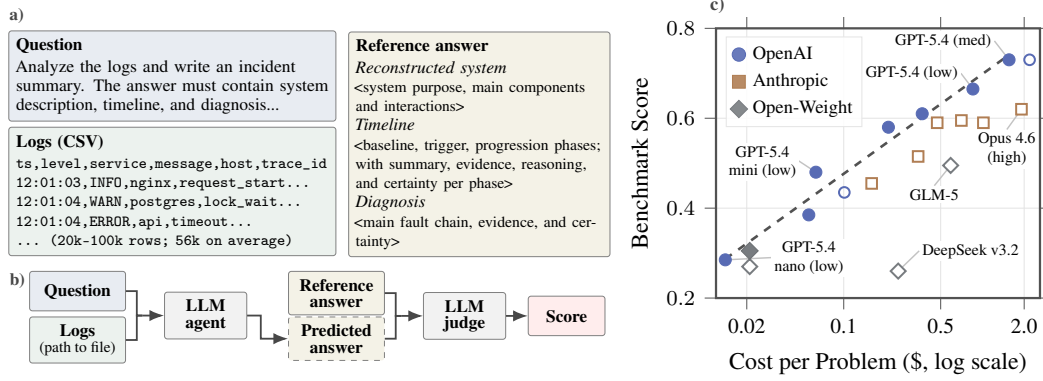


Figure 1: **SLS-BENCH task format, evaluation protocol, and cost-quality trade-off.** **a)** Each benchmark problem combines a question, a large log file, and a reference answer. The question is fixed across problems, while the logs and answers are problem-specific. **b)** The agent reads the question and logs, and writes its own incident summary that is scored against the reference by an LLM judge. **c)** Benchmark scores versus cost per problem. Filled points define the Pareto frontier, with an approximately logarithmic trend with respect to cost.

collecting such data at scale is prohibitively expensive, requiring access to diverse production systems, observed failures, relevant logs, and substantial expert effort to produce reference summaries.

In this work, we address this gap with a report-grounded synthetic benchmark. Rather than collecting expensive production logs and human-written summaries, we generate them synthetically by grounding them in real-world incident reports. Hundreds of such reports are publicly available, and they typically describe the system components involved, the system topology, the technology stack, the failure symptoms, the timeline of the incident, and the root cause. These are some of the core ingredients needed to construct a log stream that reflects a real system failure.

Turning these reports into benchmark tasks is non-trivial. Two main challenges exist: (1) translating high-level incident reports into detailed log streams that capture both overall incident dynamics and low-level realism (e.g., log structure and distribution), and (2) generating comprehensive reference summaries that faithfully reflect the log stream, from the overarching incident narrative down to specific log-level evidence.

We address these challenges and make the following contributions:

- **SLS-BENCH.** We introduce the Synthetic Log Summarization benchmark (SLS-BENCH), consisting of 90 tasks based on curated incident reports. Each task contains a large log file (56k logs on average), a fixed question, and a comprehensive reference summary (Fig. 1), and is grounded in a unique incident report covering a distinct system and incident. To our knowledge, SLS-BENCH is the first benchmark for incident log summarization that combines large log streams, diverse systems and incidents, structured reference summaries, and detailed multi-dimensional evaluation.
- **Synthetic data pipeline.** At the core of SLS-BENCH is our novel report-grounded pipeline that converts real-world incident reports into synthetic logs and reference summaries, enabling benchmark construction at this scale. To ensure high quality of the generated data, we use incident models as guides for realism and validity, generator-verifier loops with strict algorithmic and rubric-based verification, and measures to ensure verifier stability. Furthermore, we demonstrate that our generated logs capture key statistical and structural properties of real-world log streams.
- **Evaluation.** We evaluate 14 open-weight and proprietary LLMs on SLS-BENCH and find that frontier models tend to achieve strong summary quality but at significant cost and latency, while open-weight models lag substantially in quality. Score breakdowns show that top-scoring models are good at high-level incident reconstruction, but weaker at capturing detailed incident progression and supporting claims with evidence. We further analyze frontier-model trajectories and identify opportunities to reduce cost and latency, including artifact pre-computation and task parallelization.

2 Related Work

Task-level log analytics. Automated log analysis has traditionally been organized around specialized subtasks. For instance, log parsers convert raw messages into structured templates [He et al., 2017, Dai et al., 2020, Huo et al., 2021, Liu et al., 2022]; anomaly- and failure-detection methods identify unusual patterns within log sequences [Du et al., 2017, Meng et al., 2019, Zhang et al., 2019, Guo et al., 2021]; and root-cause analysis (RCA) methods localize faulty components or connections in distributed systems [Chen et al., 2014, Ma et al., 2020, Wu et al., 2020, Lin et al., 2024]. While these are essential building blocks, their outputs remain highly granular: a parser returns a template, a detector returns an anomaly score, and an RCA system returns a list of candidate faults. They do not produce the comprehensive, human-readable summaries that we focus on in this work.

LLMs for log analysis. LLMs have significantly expanded the scope of tasks a single method can handle. In-context learning has been successfully applied to log parsing and anomaly detection [Le and Zhang, 2023, Jiang et al., 2023a, Xu et al., 2024, Wu et al., 2024, Liu et al., 2023, Qi et al., 2023], while other research fine-tunes LLMs on log data to address multiple analytics tasks simultaneously [Huang et al., 2023, Ji et al., 2024, Liu et al., 2024]. More recent methodologies incorporate expert-like analysis workflows and stepwise reasoning [Ma et al., 2025]. Despite these advancements, evaluation protocols continue to center on limited contexts and brief outputs, falling short of full incident reconstruction and summarization.

Agentic log analysis. Agentic systems expand upon these foundations by managing diagnostic workflows end-to-end. For example, RCAgent [Wang et al., 2023] predicts root causes, evidence, and solutions for cloud incidents; Flow-of-Action [Pei et al., 2025] executes multi-step RCA; AetherLog [Cui et al., 2025] performs log-based RCA utilizing knowledge graphs. Additionally, systems like TrioXpert [Sun et al., 2025] and COCA [Li et al., 2025] handle anomaly triage, localize faulty components, and generate brief root-cause summaries from issue reports. Yet, the evaluation of these systems remains constrained to localized components and short explanations, lacking the scale required to reconstruct complete post-incident reports from raw log streams.

Log analytics benchmarks. While existing benchmarks have grown in both scale and complexity, their evaluation targets remain heavily constrained. Datasets like Loghub-2.0 [Jiang et al., 2023b] provide millions of log lines but restrict tasks to template prediction. Others, such as LogQA [Huang et al., 2023] and NLPLog [Ji et al., 2024], pose questions answerable from a single log entry; LogSummary [Meng et al., 2023] and LogEval [Cui et al., 2024] summarize windows of up to twenty logs into keywords or triples; and LogLM [Liu et al., 2024] spans multiple log analysis tasks with at most ten log entries per instance. Recent large-scale telemetry benchmarks, including RCAEval [Pham et al., 2025] and OpenRCA [Xu et al., 2025], span metrics, logs, and traces, scoring agents on top-k predictions of the root-cause service or a handful of categorical fields like failure type. Even environments featuring live fault-injection, such as AIOpsLab [Chen et al., 2025] and Cloud-OpsBench [Wang et al., 2026], add deterministic state snapshots and score agents per-task on detection, localization, or mitigation success. In every case, evaluation targets small, task-specific outputs. None demand the comprehensive, stream-level failure summaries that SLS-BENCH requires.

3 Benchmark

Each SLS-BENCH problem consists of a question, a log file, and a reference answer (Fig. 1a). The question is fixed across all problems, while the log file and reference answer are produced from a unique incident report for each problem. The question asks an agent to analyze the log file and produce an incident summary. The summary must include the inferred system structure, a phase-by-phase walkthrough of system behaviors, and a diagnosis with root cause, failure chain, evidence, and confidence. The produced answer is scored against the reference answer by an LLM judge. This section describes the pipeline used to construct the tasks: Sec. 3.1 describes the main components, and Sec. 3.2 describes how they are generated. Sec. 3.3 describes the LLM judge. Throughout this section, we use LLMs to assist with report collection and filtering, generate and verify incident models, simulator code, and reference answers, and create judge calibration data.

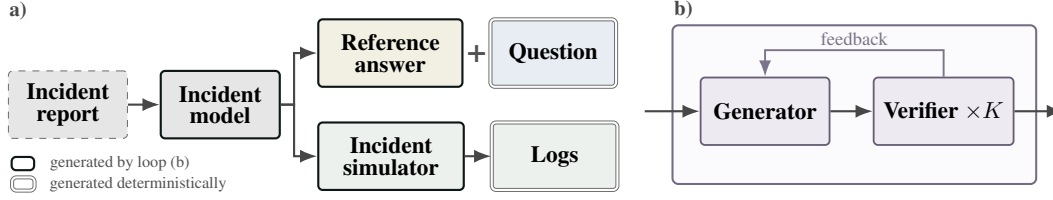


Figure 2: **Problem generation pipeline.** a) Each problem is seeded with a unique incident report. The report is converted into an incident model, which is then used to generate both a reference answer and an incident simulator; running the simulator produces the log file. The question template is fixed and shared across problems. Together, the question, logs, and reference answer constitute one benchmark problem. Solid outlines mark objects produced by the generator-verifier loop; double outlines mark deterministic outputs. b) The generator-verifier loop alternates between proposal and verification: the generator proposes an object, the verifier checks it, and any failed check returns feedback for revision. An object is accepted only after passing verification in K consecutive rounds.

3.1 Pipeline Components

Incident reports. Incident reports, written by engineers after system failures, provide the real-world grounding for our pipeline. High-quality reports typically describe the affected system components, their interactions, the technology stack, observed symptoms, the failure timeline, and the root cause. Using these reports helps our synthetic data cover diverse and realistic system architectures, failure types, and incident progressions. We collected 343 publicly available incident reports through manual and LLM-assisted web search and human screening, then de-duplicated them using Jaccard similarity (threshold 0.8), yielding 289 unique reports. We then applied an LLM-based filter designed to select reports with rich real-world grounding, such as clear descriptions of the system, failure, timeline, and symptoms. This filtering process isolated 90 high-quality reports that seed our benchmark problems. The collection process, filtering prompt, report distributions, and licensing details are provided in App. A.

To validate the LLM filter, we manually annotated 56 of the 289 unique reports (25 high-quality, 31 low-quality) and evaluated models from the GPT-5.2 family on filtering precision. GPT-5.2 (medium) achieved the highest precision (91%) and was therefore selected as our filtering model. We assessed the filter’s reliability by running it twice across all 289 reports, resulting in 266 identical verdicts (Cohen’s $\kappa = 0.81$), indicating strong consistency [McHugh, 2012].

Incident model. While incident reports describe the overall shape and progression of an incident, they frequently omit critical low-level details necessary for log and summary generation, such as comprehensive component lists, inter-component interactions, and detailed log emission behaviors during the incident. We infer these missing details from the provided report context.

We introduce the *incident model* (Fig. 3) as a structured framework for interpreting the incident reports and systematically inferring the missing information from them in a consistent, controllable, and verifiable manner. The incident model is the core of our pipeline and drives the generation of both incident logs and summaries (Fig. 2a). It comprises three main parts that we introduce next.

First, the *system structure* (Fig. 3a) defines a static graph of log-emitting components, specifying the system’s topology and per-component behavior. For each component, we define its name, interactions with other components, and the log templates it emits. A log template specifies the structure of a log line using fixed and variable slots (e.g., `user * logged in`). Each variable takes its allowed values, such as an integer range, a predefined set, or a regular expression (e.g., an IP address). Emitting a log line involves instantiating these variables with concrete values (e.g., `user Alice logged in`) and appending the result to the log stream. Components can represent individual processes, services, or sub-systems, and can be assigned to multiple hosts to simulate distributed environments.

Second, *log emission mechanisms* (Fig. 3b) govern which templates are emitted and when:

1. *Burst emissions* generate a specified number of logs from a single template within a short interval, modeling events such as configuration changes, restarts, or transient errors.
2. *Background emissions* generate logs from a single template at a fixed rate, modeling recurring activity such as health checks or baseline traffic.

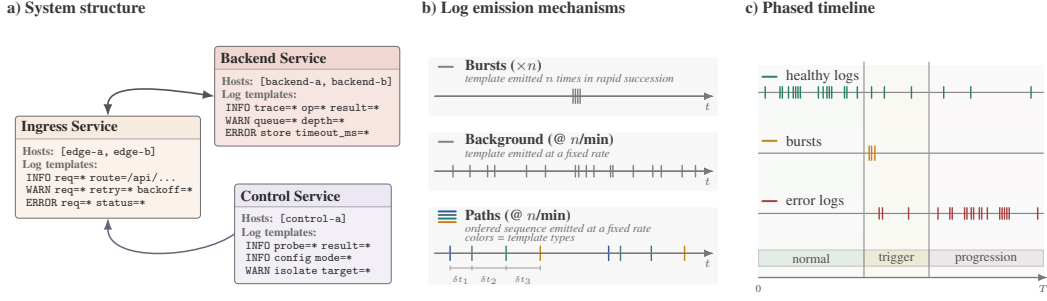


Figure 3: **Incident model.** **a) System structure:** a static graph of log-emitting components, with example log templates and edges indicating interactions. **b) Log emission mechanisms:** logs are emitted as bursts, as background activity, or as ordered paths corresponding to requests traversing multiple components. **a) Phased timeline:** emission behavior changes across normal, trigger, and progression phases, allowing log rates, error activity, and path latencies to evolve over time.

3. *Path emissions* generate an ordered sequence of logs across multiple templates at a fixed rate with per-hop latencies, used to simulate requests traversing multiple system components.

We selected these mechanisms based on template-emission patterns observed in real log streams. App. B.1 specifies their parameters and details their definitions within the incident model.

Third, the *phased timeline* (Fig. 3c) defines the temporal structure of the incident, partitioned into three phase types: a *normal* phase establishing the baseline behavior, a *trigger* phase introducing the failure-inducing event, and one or more *progression* phases detailing the failure’s evolution. Each phase specifies active emission mechanisms and their respective intensities, allowing log templates to emerge, vanish, or change emission rate as the incident progresses. For path emissions, phases also modulate $p50$ and $p95$ inter-hop latencies. Emission properties remain constant within a given phase, creating stepwise shifts in log patterns when transitioning between phases.

In summary, the incident model formally specifies the system topology, available log templates, and temporal evolution of log emission patterns. This framework allows us to translate diverse, often underspecified real-world incident reports into a unified, well-specified representation. Missing information is inferred in a manner consistent with the overall model, enabling controlled data generation, systematic evaluation, and clear interfaces for downstream pipeline stages.

We leverage an LLM to translate raw incident reports into these structured incident models. The LLM is instructed to preserve the core dynamics and structure of the original incident while coherently inferring missing elements, such as adding plausible system components and defining appropriate log templates for specific technologies. Rather than attempting to reconstruct the entire production architecture, we focus on inferring the minimal system subset necessary to contextualize the failure. The quality of these generated models is subsequently validated using a combination of algorithmic checks and LLM-based verifiers (Sec. 3.2). The final incident models are serialized as YAML files (see App. B.1 for an example), capturing all three components discussed above. Aggregate statistics characterizing these generated models are provided in App. B.4.

Incident simulator and log generation. While the incident model specifies the system topology and phase-dependent emission mechanisms, it does not yield a concrete log stream. Specifically, it abstracts away exact timestamps, emitted log identities, realized template variables, and the granular execution of encoded behaviors. To instantiate a concrete log stream, the incident model must be executed by a simulator that operationalizes these high-level specifications into low-level details. For each incident model, we leverage an LLM to generate a deterministic Python-based simulator that adheres to the model specification and coordinates the following low-level aspects to produce logs:

- **Timing and counts.** The simulator translates emission mechanisms into discrete events with precise timestamps. For each phase, it determines the scheduling of bursts, background, and path emissions, and enforces specified inter-log latencies along request paths.
- **Variable values.** Upon emission, each template variable is populated with a value from its defined domain. These values are dynamically restricted to be contextually appropriate

for the current incident phase (e.g., HTTP status codes remaining in the 2xx range during normal operation, but shifting to 4xx and 5xx during a failure).

- **Variable coherence.** The simulator ensures that variable values remain logically valid and mutually consistent throughout a path execution. For instance, a log message indicating a connection timeout must not co-occur with a variable indicating a fast connection time; similarly, if a request path registers a failure, subsequent logs within that same path cannot report a successful completion.

The resulting simulator deterministically outputs a CSV file containing the log stream, typically ranging from 20k to 100k logs (56k on average). Artifact quality is enforced via algorithmic and LLM verifiers (Sec. 3.2). A complete simulator example is provided in App. B.5.

Question and reference answer. Every problem in SLS-BENCH uses a standardized question requiring an agent to analyze the logs and produce an incident summary with four components: (i) incident highlight bullets; (ii) system structure and behavior; (iii) phase-by-phase walkthrough detailing observed behaviors, interpretations, evidence, and confidence; and (iv) an overarching diagnosis specifying the root cause, causal chain, evidence, and confidence level (see App. D).

We generate reference summaries directly from the incident model. Because the model compactly encapsulates all information about the incident, we instruct an LLM to translate this structured representation into a precise narrative that answers the provided question to produce the reference summary. However, deriving summaries from the incident model risks incorporating “invisible” ground truth—internal model mechanics (e.g., a silent hardware fault) that never surface in the log stream. To prevent this, the generator is strictly instructed to describe only the log-observable implications of internal processes, a constraint natively supported by the incident model’s explicit mapping of behaviors to log emissions. The resulting 90 reference summaries range from 1,900 to 3,300 words (2,400 on average). Their quality is assessed using an LLM verifier (Sec. 3.2).

3.2 Pipeline Component Generation

This section describes the generation pipeline for incident models, simulators, and reference answers introduced in Sec. 3.1, and covers the design choices that promote data quality and log realism. The code and prompts used in the pipeline are provided in our code repository.

Generation approach. We generate all benchmark artifacts using the generator–verifier loop in Fig. 2b. First, a generator proposes an object, which a verifier evaluates against object-specific criteria. If no violations are detected, the object is accepted; otherwise, the verifier provides constructive feedback, prompting the generator to revise the object. We allow up to 20 such iterations per object. If the generator fails to produce a valid artifact within this budget, the process restarts from scratch.

Verification depends on the artifact type. Incident models and simulators undergo both algorithmic and semantic validation. Algorithmic verifiers enforce output schema adherence, structural validity, and internal consistency. For incident models, LLM verifiers receive the source report and check faithfulness to its stated facts and the plausibility of inferred details (App. B.2); simulator verifiers check adherence to the incident model. Reference answers, whose requirements are predominantly semantic (e.g., narrative faithfulness to the incident model), are evaluated solely by an LLM verifier.

Producing high-fidelity objects requires multiple independent verifier passes. As Fig. 4 demonstrates, objects accepted after only a single pass ($K = 1$) frequently fail subsequent verifications (i.e., have low pass rate). Manual inspection reveals that $K = 1$ failures typically stem from genuine requirement violations that degrade quality. Conversely, failures at higher thresholds (e.g., $K = 10$) are often artifacts of overly rigid rubric interpretations that do not meaningfully impact object utility. Thus, consecutive verifications yield verifier-stable objects that, once accepted, are highly likely to pass again.

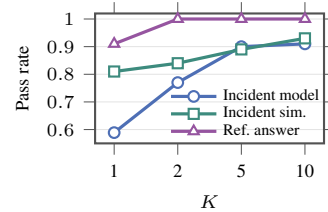


Figure 4: Increasing the verifier threshold K improves the pass rate.

We employ GPT-5.2 (high) for both generation and verification. Strong verifiers are crucial for detecting subtle constraint violations; weaker generators often struggle to incorporate feedback from a stronger verifier within the iteration budget, resulting in long or failed object generation.

To evaluate the sensitivity of the LLM verifiers, we manually injected non-trivial requirement violations into 50 accepted objects ($K = 10$) for each artifact type. The verifiers successfully detected 98% of these injected errors, confirming their high sensitivity to constraint violations.

Generated log realism. While high-level realism is anchored by the source incident reports, low-level realism relies on the LLM generating contextually appropriate log templates. To assess this, we compare SLS-BENCH against Loghub-2.0 (real logs from 14 systems) across metrics capturing message form, template diversity, and local stream structure (Fig. 5; see App. C for definitions).

SLS-BENCH matches Loghub-2.0 in message length, variables per template, and variable cardinality, indicating that our individual log lines exhibit realistic structure and variable diversity. However, three notable differences emerge:

1. SLS-BENCH contains fewer unique templates per file. This is an intentional consequence of simulating the minimal sub-system relevant to the incident, rather than an entire production environment laden with unrelated background noise.
2. While both datasets display heavy-tailed rank-frequency curves, SLS-BENCH’s curves are flatter, indicating a more even distribution of template occurrences rather than extreme concentration in a few dominant templates.
3. Adjacent-template predictability is lower in SLS-BENCH. Because our synthetic streams interleave diverse background activities, concurrent request paths, and phase-specific failure bursts, they avoid the highly repetitive, predictable template cycles characteristic of many Loghub-2.0 datasets.

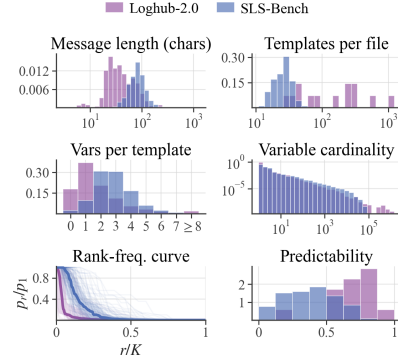


Figure 5: Comparing Loghub-2.0 and SLS-BENCH on representative statistics.

Overall, SLS-BENCH achieves message-level realism, with minor deviations in template diversity and sequence predictability directly attributable to our incident-centric generation design. However, high-level properties such as ambiguity, incomplete evidence, and conflicting signals in the log stream are harder to formalize and quantify; realism along these axes relies on grounding in real-world incident reports and adherence to internally consistent incident models (App. B.2).

3.3 LLM Judge Scoring

To evaluate agent performance, we employ an LLM-as-a-judge paradigm. The judge receives the standardized question, the reference summary, and the agent’s predicted summary, and assigns one of five labels: *mismatch*, *weak match*, *partial match*, *good match*, or *strong match*. The grading rubric spans seven dimensions evaluating both structural compliance and content accuracy, and produces the final score by combining the per-dimension scores (see App. E for full definitions).

To select a judge, we constructed a calibration dataset utilizing 10 benchmark problems. For each problem, we generated five candidate predictions corresponding to the five rubric labels. The *strong match* was generated as an LLM paraphrase of the reference answer. *Good*, *partial*, and *weak* matches were derived by progressively degrading this paraphrase to intentionally violate specific rubric thresholds. The *mismatch* baseline was created by combining a problem with a reference answer from an entirely different incident.

Based on the achieved calibration accuracy of 90%, GPT-5.2 (medium) was selected as the primary judge for the evaluation. However, large-scale benchmark runs revealed a slight strictness bias, where

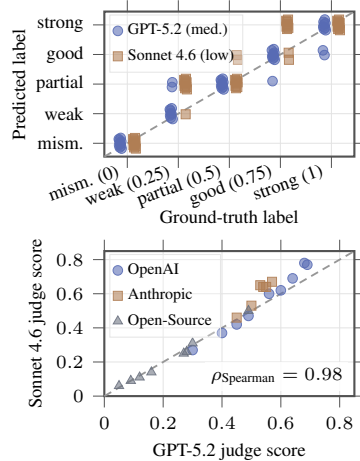


Figure 6: **Top:** Judge prediction vs. ground-truth on calibration set. **Bottom:** Judge agreement across evaluated LLMs.

Table 1: **Benchmark scores with cost and time per problem.** Score is mean $\pm$ SEM, time is mean $\pm$ SD; cost and time are averages per problem. Open-weight model parameters are total/active for mixture-of-expert models. A $\star$ denotes Pareto-optimal models under the cost–score trade-off (Fig. 13). We use the sample variance formula to compute standard deviation and standard error of the mean.

Model	Score ($\uparrow$)	Cost [\$] ($\downarrow$)	Time [Min] ($\downarrow$)	Score/\$	Score/Min
<i>OpenAI</i>					
GPT-5.4 (high)	0.73 $\pm$ 0.01	2.19	9.5 $\pm$ 1.6	0.33	0.08
GPT-5.4 (medium) $\star$	0.73 $\pm$ 0.01	1.56	6.9 $\pm$ 1.0	0.47	0.10
GPT-5.4 (low) $\star$	0.67 $\pm$ 0.01	0.85	3.4 $\pm$ 0.5	0.79	0.19
GPT-5.4 Mini (high) $\star$	0.61 $\pm$ 0.01	0.37	5.2 $\pm$ 1.3	1.65	0.11
GPT-5.4 Mini (medium) $\star$	0.57 $\pm$ 0.01	0.21	3.0 $\pm$ 0.7	2.71	0.18
GPT-5.4 Mini (low) $\star$	0.48 $\pm$ 0.01	0.06	0.9 $\pm$ 0.1	8.00	0.53
GPT-5.4 Nano (high)	0.44 $\pm$ 0.01	0.10	7.0 $\pm$ 1.7	4.40	0.06
GPT-5.4 Nano (medium) $\star$	0.39 $\pm$ 0.01	0.06	3.7 $\pm$ 0.9	6.50	0.09
GPT-5.4 Nano (low) $\star$	0.28 $\pm$ 0.01	0.01	1.1 $\pm$ 0.2	28.00	0.22
<i>Anthropic</i>					
Opus 4.6 (high)	0.61 $\pm$ 0.01	1.91	9.9 $\pm$ 2.9	0.32	0.06
Opus 4.6 (medium)	0.59 $\pm$ 0.01	0.71	4.1 $\pm$ 0.8	0.83	0.13
Sonnet 4.6 (medium)	0.57 $\pm$ 0.01	1.02	11.1 $\pm$ 3.5	0.56	0.05
Sonnet 4.6 (low)	0.58 $\pm$ 0.01	0.47	5.0 $\pm$ 1.4	1.23	0.11
Opus 4.6 (low)	0.53 $\pm$ 0.01	0.34	2.5 $\pm$ 0.5	1.56	0.21
Haiku 4.5	0.46 $\pm$ 0.01	0.16	2.4 $\pm$ 0.5	2.88	0.19
<i>Open-Weight</i>					
GLM-5 (744B/40B)	0.49 $\pm$ 0.01	0.59	7.9 $\pm$ 3.5	0.83	0.05
MiniMax-M2 (230B/10B) $\star$	0.29 $\pm$ 0.01	0.02	3.4 $\pm$ 1.1	14.50	0.06
Nemotron 3 Super (120B/12B)	0.29 $\pm$ 0.02	0.02	7.8 $\pm$ 5.1	14.50	0.03
DeepSeek V3.2 (685B/37B)	0.27 $\pm$ 0.02	0.25	11.8 $\pm$ 6.1	1.08	0.02
Mistral Large 3 (675B/41B)	0.12 $\pm$ 0.01	0.02	0.8 $\pm$ 0.4	6.00	0.13
Ministral 3 (14B)	0.12 $\pm$ 0.01	0.02	1.8 $\pm$ 1.1	6.00	0.05
Qwen3-Next (80B/3B)	0.10 $\pm$ 0.01	0.02	3.7 $\pm$ 1.9	5.00	0.02
Nemotron 3 Nano (30B/3.5B)	0.06 $\pm$ 0.01	0.02	5.1 $\pm$ 5.1	3.00	0.01

GPT-5.2 occasionally penalized harmless additions, such as agents reporting extra log-value details. Refining the prompt rubric did not mitigate this behavior. To counterbalance this strictness and reduce model-family bias, we introduced a second judge: Sonnet 4.6 (low). Sonnet demonstrated greater leniency toward harmless additions, though it was comparatively more prone to missing genuine rubric violations, resulting in a lower independent calibration accuracy (62%).

To compute the final score, we map the five rubric labels to continuous values (0, 0.25, 0.50, 0.75, and 1.00) and average the predictions of both judges. Fig. 6 (top) illustrates per-label calibration performance: both judges reliably identify *mismatch*, *partial match*, and *strong match* categories, while GPT-5.2 excels at distinguishing the boundary between *weak match* and *good match*. Fig. 6 (bottom) shows that across 14 evaluated LLMs, the mean scores of the two judges differ by no more than 0.12, and their final model rankings remain highly correlated ($\rho_{\text{Spearman}} = 0.98$). We further test judge robustness, agreement with human evaluation, and external log analytics benchmarks in App. J.

4 Results

To establish a baseline on SLS-BENCH, we evaluate an LLM agent equipped with a stateful Python interpreter for analyzing logs using pandas [McKinney, 2010] and standard Python packages. Our evaluation includes closed-source frontier models, specifically the GPT-5.4 [OpenAI, 2026] and Claude 4.x models [Anthropic, 2026], and a diverse set of open-weight models, including Ministral 3 14B and Mistral Large 3 [Mistral AI, 2025], Nemotron 3 Nano and Nemotron 3 Super [NVIDIA, 2025], Qwen3-Next-80B-A3B [Qwen Team, 2025], MiniMax-M2 [MiniMax, 2025], GLM-5 [GLM-5 Team, 2026], and DeepSeek-V3.2 [DeepSeek-AI, 2025]. All prompts, the agent implementation, and evaluation scripts are open-sourced in our repository, with further details provided in App. G.

Main results. Table 1 presents evaluation results for all models. We report the mean score across the 90 benchmark problems, along with the average monetary cost and runtime per problem. OpenAI models define most of the Pareto frontier in the cost–performance trade-off space, with MiniMax-M2

also Pareto-optimal at low cost. Among closed-source models, performance generally scales with model size and reasoning effort. However, within any given size class, models using lower reasoning effort offer the best cost- and time-efficiency ratios. Open-weight models currently lag behind their closed-source counterparts, with GLM-5 achieving the strongest open-weight score. Our LLM judge rubric maps *good match* to 0.75, which we use as the threshold for a reliable incident reconstruction. The strongest models approach this threshold, with GPT-5.4 medium and high both reaching 0.73 on average, requiring \$1.56–\$2.19 and 6.9–9.5 minutes per problem on average. App. H shows model rankings are stable under different tool-call budgets and that the scores increase with the budget.

Disaggregated score analysis. Fig. 7 shows the disaggregation of the overall score into its seven constituent rubric dimensions. Per-axis scores generally improve with the overall score, model capability, and reasoning effort. The score breakdown shows why the task remains difficult. *Diagnosis*, which evaluates high-level incident chain reconstruction, is nearing saturation at the maximum score of 1. In contrast, *timeline progression* and *support alignment*, which assess whether high-level reconstructions are supported by granular log evidence, remain unsaturated and typically plateau in the 0.7–0.8 range. Error analysis of GPT-5.4 (medium) predictions with the lowest support alignment scores shows that the main failure modes involve omitting or misinterpreting the critical evidence used by the reference answer.

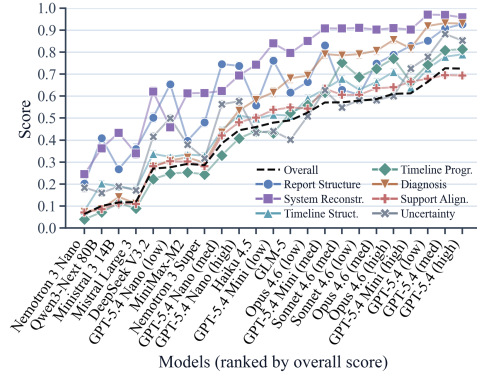


Figure 7: Overall score split into its constituent axes as evaluated by the LLM judge.

Efficiency analysis. Top-scoring LLMs can generate useful incident summaries, but they do so with significant cost and latency overheads. To identify optimization opportunities, we analyzed 10 reasoning trajectories from GPT-5.4 (medium). These trajectories consistently begin with a generic orientation phase (Fig. 8, top), during which the agent uses the Python interpreter to inspect the log schema, size, time span, unique hosts and services, errors, and templates, while also constructing temporal views (e.g., per-minute volume, severity trends) to localize the incident. Across the analyzed trajectories, 21% (75 out of 361) of the Python tool outputs were precomputable, suggesting that this context could be injected prior to the agent’s analysis to reduce API calls. Furthermore, measured by tool-output count, the initial orientation and planning prefix accounts for roughly 30% of an average trajectory. After this prefix, the agent typically branches into independent analytical streams that could be effectively parallelized (Fig. 8, bottom). Further details are available in App. I.

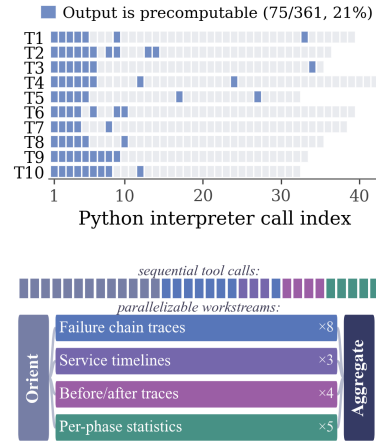


Figure 8: **Top:** Precomputable Python tool outputs. **Bottom:** After initial orientation, sequential tool calls can be parallelized into independent streams.

5 Conclusions

In this work, we introduced SLS-BENCH, a benchmark for evaluating LLM agents on incident log summarization from large-scale log streams. To construct SLS-BENCH, we developed a synthetic data generation pipeline that converts incident reports into synthetic log streams and reference incident summaries. This pipeline demonstrates how real-world grounding, explicit incident modeling, strict validation, and statistical comparisons with real log streams can be used to produce realistic and controllable observability data. The resulting benchmark contains 90 distinct tasks, each derived from a unique real-world incident report. Our evaluation of 14 LLMs shows that current frontier models can produce viable incident summaries in this setting, but remain costly and slow, while open-weight models lag substantially behind in summary quality.

SLS-BENCH currently has three main limitations. First, coverage is limited to 90 publicly available incident reports; future versions can expand this pool with new public, private, or synthetic incident

reports. Second, the incident model caps the complexity of the behavior each benchmark problem represents. While we include basic modeling components to represent realistic system behaviors, future work can extend the model to target richer systems, additional failure modes, or specific behaviors, such as background noise and the incidental complexity of complete production environments. Third, logs and reference summaries are generated from the same incident model, and judges score predictions against the references without accessing the logs. Evaluation therefore depends on the references faithfully representing the evidence available in the generated logs. Judge-robustness tests and a small human study support this scoring approach, and rank agreement with o1ly-bench and OpenRCA provides evidence of relevance to related observability tasks (App. J). Evaluating direct transfer of SLS-BENCH scores to real incident log summarization is a natural next step in extending this validation.

Ultimately, SLS-BENCH enables the systematic evaluation and improvement of LLM agents on realistic incident summarization tasks, while our pipeline offers a systematic and scalable blueprint for building future synthetic observability datasets. To support reproducible generation, evaluation, and future extensions, we have publicly released our data and code.

References

- Anthropic. System Card: Claude Opus 4.6. <https://www-cdn.anthropic.com/14e4fb01875d2a69f646fa5e574dea2b1c0ff7b5.pdf>, February 2026. Accessed: 2026-04-28.
- Harrison Chase. Langchain, October 2022. URL <https://github.com/langchain-ai/langchain>.
- Pengfei Chen, Yong Qi, Pengfei Zheng, and Di Hou. Causeinfer: Automatic and distributed performance diagnosis with hierarchical causality graph in large distributed systems. *IEEE INFOCOM 2014 - IEEE Conference on Computer Communications*, pages 1887–1895, 2014. URL <https://api.semanticscholar.org/CorpusID:12285010>.
- 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, 2025. URL <https://arxiv.org/abs/2501.06706>.
- Tianyu Cui, Shiyu Ma, Ziang Chen, Tong Xiao, Shimin Tao, Yilun Liu, Shenglin Zhang, Duoming Lin, Changchang Liu, Yuzhe Cai, Weibin Meng, Yongqian Sun, and Dan Pei. Logeval: A comprehensive benchmark suite for large language models in log analysis, 2024. URL <https://arxiv.org/abs/2407.01896>.
- Tianyu Cui, Ruowei Fu, Changchang Liu, Yuhe Ji, Wenwei Gu, Shenglin Zhang, Yongqian Sun, and Dan Pei. Aetherlog: Log-based root cause analysis by integrating large language models with knowledge graphs. *2025 IEEE 36th International Symposium on Software Reliability Engineering (ISSRE)*, pages 49–60, 2025. URL <https://api.semanticscholar.org/CorpusID:283013959>.
- Hetong Dai, Heng Li, Che-Shao Chen, Weiyi Shang, and Tse-Hsun Peter Chen. Logram: Efficient log parsing using n-gram dictionaries. *IEEE Transactions on Software Engineering*, 48:879–892, 2020. URL <https://api.semanticscholar.org/CorpusID:210116650>.
- DeepSeek-AI. DeepSeek-V3.2: Pushing the frontier of open large language models. 2025. URL <https://arxiv.org/abs/2512.02556>.
- Min Du, Feifei Li, Guineng Zheng, and Vivek Srikumar. Deeplog: Anomaly detection and diagnosis from system logs through deep learning. *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017. URL <https://api.semanticscholar.org/CorpusID:4232579>.
- GLM-5 Team. GLM-5: from vibe coding to agentic engineering. 2026. URL <https://arxiv.org/abs/2602.15763>.

- Haixuan Guo, Shuhan Yuan, and Xintao Wu. Logbert: Log anomaly detection via bert. *2021 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8, 2021. URL <https://api.semanticscholar.org/CorpusID:232146842>.
- Pinjia He, Jieming Zhu, Zibin Zheng, and Michael R. Lyu. Drain: An online log parsing approach with fixed depth tree. *2017 IEEE International Conference on Web Services (ICWS)*, pages 33–40, 2017. URL <https://api.semanticscholar.org/CorpusID:4776668>.
- Shaohan Huang, Yi Liu, Carol Fung, Jiaying Qi, Hailong Yang, and Zhongzhi Luan. Logqa: Question answering in unstructured logs. *ArXiv*, abs/2303.11715, 2023. URL <https://api.semanticscholar.org/CorpusID:257636673>.
- Yintong Huo, Yuxin Su, Cheryl Lee, and Michael R. Lyu. Semparser: A semantic parser for log analytics. *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 881–893, 2021. URL <https://api.semanticscholar.org/CorpusID:245425163>.
- Yuhe Ji, Yilun Liu, Feiyu Yao, Minggui He, Shimin Tao, Xiaofeng Zhao, Chang Su, Xinhua Yang, Weibin Meng, Yuming Xie, Boxing Chen, Shenglin Zhang, and Yongqian Sun. Adapting large language models to log analysis with interpretable domain knowledge. *Proceedings of the 34th ACM International Conference on Information and Knowledge Management*, 2024. URL <https://api.semanticscholar.org/CorpusID:274437676>.
- Zhihan Jiang, Jinyang Liu, Zhuangbin Chen, Yichen Li, Junjie Huang, Yintong Huo, Pinjia He, Jiazhen Gu, and Michael R. Lyu. Lilac: Log parsing using llms with adaptive parsing cache. *Proceedings of the ACM on Software Engineering*, 1:137 – 160, 2023a. URL <https://api.semanticscholar.org/CorpusID:263608268>.
- Zhihan Jiang, Jinyang Liu, Junjie Huang, Yichen Li, Yintong Huo, Jia-Yuan Gu, Zhuangbin Chen, Jieming Zhu, and Michael R. Lyu. A large-scale evaluation for log parsing techniques: How far are we? *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2023b. URL <https://api.semanticscholar.org/CorpusID:261049500>.
- Van-Hoang Le and Hongyu Zhang. Log parsing with prompt-based few-shot learning. *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 2438–2449, 2023. URL <https://api.semanticscholar.org/CorpusID:256868552>.
- Yichen Li, Yulun Wu, Jinyang Liu, Zhihan Jiang, Zhuangbin Chen, Guangba Yu, and Michael R. Lyu. Coca: Generative root cause analysis for distributed systems with code knowledge. *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pages 1346–1358, 2025. URL <https://api.semanticscholar.org/CorpusID:277452340>.
- Cheng-Ming Lin, Ching Chang, Wei-Yao Wang, Kuang-Da Wang, and Wenjie Peng. Root cause analysis in microservice using neural granger causal discovery. In *AAAI Conference on Artificial Intelligence*, 2024. URL <https://api.semanticscholar.org/CorpusID:267406336>.
- Yilun Liu, Shimin Tao, Weibin Meng, Jingyu Wang, Wenbing Ma, Yuhang Chen, Yanqing Zhao, Hao Yang, and Yanfei Jiang. Interpretable online log analysis using large language models with prompt strategies. *2024 IEEE/ACM 32nd International Conference on Program Comprehension (ICPC)*, pages 35–46, 2023. URL <https://api.semanticscholar.org/CorpusID:260900274>.
- Yilun Liu, Yuhe Ji, Shimin Tao, Minggui He, Weibin Meng, Shenglin Zhang, Yongqian Sun, Yuming Xie, Boxing Chen, and Hao Yang. Loglm: From task-based to instruction-based automated log analysis. *2025 IEEE/ACM 47th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 401–412, 2024. URL <https://api.semanticscholar.org/CorpusID:273345350>.
- Yudong Liu, Xu Zhang, Shilin He, Hongyu Zhang, Liqun Li, Yu Kang, Yong Xu, Minghua Ma, Qingwei Lin, Yingnong Dang, S. Rajmohan, and Dongmei Zhang. Uniparser: A unified log parser for heterogeneous log data. *Proceedings of the ACM Web Conference 2022*, 2022. URL <https://api.semanticscholar.org/CorpusID:246822534>.

- Lipeng Ma, Yixuan Li, Weidong Yang, Mingjie Zhou, Xinyi Liu, Ben Fei, Shuhao Li, Xiaoyan Sun, Sihang Jiang, and Yanghua Xiao. Logreasoner: Empowering llms with expert-like coarse-to-fine reasoning for automated log analysis. *ArXiv*, abs/2509.20798, 2025. URL <https://api.semanticscholar.org/CorpusID:281675718>.
- Meng Ma, Jingmin Xu, Yuan Wang, Pengfei Chen, Zonghua Zhang, and Ping Wang. Automap: Diagnose your microservice-based web applications automatically. *Proceedings of The Web Conference 2020*, 2020. URL <https://api.semanticscholar.org/CorpusID:215839626>.
- Mary L. McHugh. Interrater reliability: the kappa statistic. *Biochemia Medica*, 22(3):276–282, 2012. URL <https://pmc.ncbi.nlm.nih.gov/articles/PMC3900052/>. Published 2012 Oct 15.
- Wes McKinney. Data structures for statistical computing in python. In Stéfan van der Walt and Jarrod Millman, editors, *Proceedings of the 9th Python in Science Conference*, pages 56–61, 2010. doi: 10.25080/Majora-92bf1922-00a.
- Weibin Meng, Ying Liu, Yichen Zhu, Shenglin Zhang, Dan Pei, Yuqing Liu, Yihao Chen, Ruizhi Zhang, Shimin Tao, Pei Sun, and Rong Zhou. Loganomaly: Unsupervised detection of sequential and quantitative anomalies in unstructured logs. In *International Joint Conference on Artificial Intelligence*, 2019. URL <https://api.semanticscholar.org/CorpusID:199466044>.
- Weibin Meng, Federico Zaiter, Yuzhe Zhang, Ying Liu, Shenglin Zhang, Shimin Tao, Yichen Zhu, Tao Han, Yongpeng Zhao, En Wang, Yuzhi Zhang, and Dan Pei. Logsummary: Unstructured log summarization for software systems. *IEEE Transactions on Network and Service Management*, 20: 3803–3815, 2023. URL <https://api.semanticscholar.org/CorpusID:256141561>.
- MiniMax. MiniMax-M2: An efficient model for the agentic era. <https://www.minimax.io/news/minimax-m2>, 2025. Code and weights: <https://github.com/MiniMax-AI/MiniMax-M2>; accessed 2026-04-28.
- Mistral AI. Introducing Mistral 3. <https://mistral.ai/news/mistral-3>, December 2025. Accessed: 2026-04-28.
- NVIDIA. NVIDIA Nemotron 3: Efficient and open intelligence. 2025. URL <https://arxiv.org/abs/2512.20856>.
- OpenAI. Introducing GPT-5.4. <https://openai.com/index/introducing-gpt-5-4/>, March 2026. Accessed: 2026-04-28.
- Changhua Pei, Zexin Wang, Fengrui Liu, Zeyan Li, Yang Liu, Xiao He, Rong Kang, Tiejing Zhang, Jianjun Chen, Jianhui Li, Gaogang Xie, and Dan Pei. Flow-of-action: Sop enhanced llm-based multi-agent system for root cause analysis. *Companion Proceedings of the ACM on Web Conference 2025*, 2025. URL <https://api.semanticscholar.org/CorpusID:276287523>.
- Luan Pham, Hongyu Zhang, Huong Ha, Flora Salim, and Xiuzhen Zhang. Rcaeval: A benchmark for root cause analysis of microservice systems with telemetry data. In *Companion Proceedings of the ACM on Web Conference 2025*, pages 777–780, 2025. doi: 10.1145/3701716.3715290. URL <https://doi.org/10.1145/3701716.3715290>.
- Jiaxing Qi, Shaohan Huang, Zhongzhi Luan, Carol J. Fung, Hailong Yang, and Depei Qian. Loggpt: Exploring chatgpt for log-based anomaly detection. *2023 IEEE International Conference on High Performance Computing & Communications, Data Science & Systems, Smart City & Dependability in Sensor, Cloud & Big Data Systems & Application (HPCC/DSS/SmartCity/DependSys)*, pages 273–280, 2023. URL <https://api.semanticscholar.org/CorpusID:261530676>.
- Qwen Team. Qwen3 technical report. 2025. URL <https://arxiv.org/abs/2505.09388>.
- Yongqian Sun, Yu Luo, Xidao Wen, Yuan Yuan, Xiaohui Nie, Shenglin Zhang, Tong Liu, and Xi Luo. Trioxpert: An automated incident management framework for microservice system. *ArXiv*, abs/2506.10043, 2025. URL <https://api.semanticscholar.org/CorpusID:279318271>.
- Yilun Wang, Guangba Yu, Haiyu Huang, Zirui Wang, Yujie Huang, Pengfei Chen, and Michael R. Lyu. Cloud-opsbench: A reproducible benchmark for agentic root cause analysis in cloud systems, 2026. URL <https://arxiv.org/abs/2603.00468>.

- Zefan Wang, Zichuan Liu, Yingying Zhang, Aoxiao Zhong, Lunting Fan, Lingfei Wu, and Qingsong Wen. Rcagent: Cloud root cause analysis by autonomous agents with tool-augmented large language models. *Proceedings of the 33rd ACM International Conference on Information and Knowledge Management*, 2023. URL <https://api.semanticscholar.org/CorpusID:264487299>.
- Li Wu, Johan Tordsson, Erik Elmroth, and Odej Kao. Microrca: Root cause localization of performance issues in microservices. *NOMS 2020 - 2020 IEEE/IFIP Network Operations and Management Symposium*, pages 1–9, 2020. URL <https://api.semanticscholar.org/CorpusID:213256910>.
- Yifan Wu, Siyu Yu, and Ying Li. Log parsing using llms with self-generated in-context learning and self-correction. *2025 IEEE/ACM 33rd International Conference on Program Comprehension (ICPC)*, pages 01–12, 2024. URL <https://api.semanticscholar.org/CorpusID:270258306>.
- Junjielong Xu, Ruichun Yang, Yintong Huo, Chengyu Zhang, and Pinjia He. Divlog: Log parsing with prompt enhanced in-context learning. *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*, pages 2457–2468, 2024. URL <https://api.semanticscholar.org/CorpusID:269123195>.
- 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 *International Conference on Learning Representations*, 2025. URL <https://api.semanticscholar.org/CorpusID:278497745>.
- Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. *ArXiv*, abs/2210.03629, 2022. URL <https://api.semanticscholar.org/CorpusID:252762395>.
- Xu Zhang, Yong Xu, Qingwei Lin, Bo Qiao, Hongyu Zhang, Yingnong Dang, Chunyu Xie, Xinsheng Yang, Qian Cheng, Ze Li, Junjie Chen, Xiaoting He, Randolph Yao, Jian-Guang Lou, Murali Chintalapati, Shen Furao, and Dongmei Zhang. Robust log-based anomaly detection on unstable log data. *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019. URL <https://api.semanticscholar.org/CorpusID:191140040>.

Appendix Overview

The following appendices provide supplementary technical details to support the main paper. Appendix A describes incident report collection, filtering, and licensing, and Appendix B details the incident model and the simulator. Appendix C provides additional statistical analyses. Appendix D presents the fixed question prompt and an example reference answer, while Appendix E provides a granular breakdown of our scoring methodology and label mappings. Appendix F reports token usage and costs across the generator and verifier steps of the data generation pipeline. Appendix G describes our agent setup and inference configuration. Appendix H reports tool-call-budget sensitivity, and Appendix I analyzes agent trajectories to identify cost and latency optimization opportunities. Appendix J reports judge robustness, human evaluation, and comparisons with external benchmarks.

A Incident Report Collection, Filtering, and Licensing

A.1 Collection and preprocessing

We collected candidate URLs through manual and LLM-assisted web search using broad queries such as “incident report”, “postmortem”, “outage postmortem”, and “failure analysis”. Each candidate page was inspected by a human for apparent relevance before its URL was added to a pool of 659 sources. After removing exact duplicate URLs, we retrieved the publicly available pages programmatically and used Trafilatura to extract relevant text from HTML, producing 343 candidate reports. Content-based de-duplication using Jaccard similarity (threshold 0.8) reduced this set to 289 reports, from which the LLM filter selected 90 (Sec. 3.1).

A.2 Filtering

The incident report filter receives the candidate report text as the user message and uses the following system prompt:

Classify the document as exactly one label: GOOD_POSTMORTEM or BAD_POSTMORTEM.

Strict rule

Label GOOD_POSTMORTEM ONLY if ALL requirements (1-7) below are explicitly satisfied in the text.
If anything is missing, vague, implied, or not tied to concrete evidence, label BAD_POSTMORTEM.

The main goal is to find postmortems that allow us to infer the system structure and write a scenario that explains what happened and why, so that we can use the structure and scenario to write a Python script that generates logs that could have been observed before, during, and after the incident described.

Requirements

1) Postmortem (post-incident analysis)

Must be a retrospective/postmortem describing an incident after the fact (not just a status update, announcement, changelog, forum thread, Q&A, or article).

2) Cloud/distributed service failure

Must be about a failure of a backend/service hosted in a cloud-based or distributed environment.

Reject if it is mainly: client/desktop issue, purely organizational/process, security-only without operational failure symptoms, single-host/local issue, etc.

3) Explains what happened, why, and observed system behaviors

Must explicitly describe:

- what happened (impact + concrete symptoms),
- why it happened (root cause or causal chain; not speculation-only),
- observed system behaviors during the incident (specific technical behaviors, not generic "degraded").

4) Clear system/component details (names + role)

Must name the relevant system/components AND describe what they do (role/functionality) so the system or parts of the system relevant for the incident can be inferred.

Generic labels like "the service", "backend", "infrastructure" without named components and roles are insufficient.

5) Sufficient detail to reconstruct the scenario over time

Must provide enough detail to reconstruct how the failure unfolded over time:

- either timestamps OR an ordered sequence of events,
- AND
- clear progression (detection -> investigation/triage -> mitigation -> recovery) with concrete event descriptions (not just "we fixed it").

6) Log-reflectable incident (explicit incident-time signals)
The problem must be clearly reflectable in system logs/traces (not only business impact).
Must include at least TWO explicit incident-time signals that are specific enough to log, such as:

- explicit error codes/types (e.g., HTTP 5xx/429, gRPC status, DB error, specific exception),
- explicit timeouts/connection failures/retry storms,
- crash/restart/OOM/resource exhaustion with a described symptom,
- failed health checks, leader elections, deployment/rollback failures,
- queue lag/backlog growth, throttling/rate limiting.

If signals are only generic (e.g., "errors occurred", "performance issues") without concrete loggable manifestations, reject.

7) Metrics-based detection (or explicitly metric-convertible)
Must explicitly state EITHER:

- the incident was detected via metrics/monitoring/alerts/anomaly (error rate/latency/saturation/SLO/etc.), OR
- a concrete numeric or categorical signal that is easily and directly convertible to a metric is explicitly mentioned (e.g., "HTTP 500 rate increased", "p95 latency rose", "CPU hit 100%", "queue depth grew", "timeouts spiked").

If detection is only via user reports/support tickets and the text does not explicitly provide a metric-like signal, label BAD_POSTMORTEM.

Output (JSON only)
Return only:

```

{
  "label": "GOOD_POSTMORTEM | BAD_POSTMORTEM",
  "missing": [],
  "reasons": [
    "Reason 1: concrete evidence or explicit gap",
    "Reason 2: concrete evidence or explicit gap"
  ]
}

```

Output field rules

- If label is GOOD_POSTMORTEM: "missing" must be [].
- If label is BAD_POSTMORTEM: "missing" must list every failed check using only: ["postmortem", "cloud_distributed", "what_happened", "why", "observed_behaviors", "system_structure", "timeline", "log_signals", "metrics_signal"]
- "reasons": 1-6 short bullets. Each must quote or tightly paraphrase evidence, or state exactly what is absent. No extra commentary.

Fig. 9 illustrates the distribution of failure types and organizations across the 90 curated reports. For organization counts, we drop counts below 3 for clarity.

We see that certain failure types and organizations appear over-represented. However, manual verification of reports from the top five organizations confirmed the uniqueness of their systems and failure types. We found only one exception (Cloudflare) where a single incident was described twice, first at a high level, then in detail; we retained both due to their substantially different content.

A.3 Licensing

The released artifacts consist of generated incident models, simulators, synthetic log streams, reference answers, benchmark questions, and metadata. We provide the source report URLs for attribution and provenance, but do not redistribute report text, HTML, or images.

We release the benchmark under CC BY 4.0 and the code under Apache 2.0. The benchmark may be used, modified, and redistributed, including commercially, subject to attribution and the license terms. These licenses apply only to the artifacts and code for which we hold the applicable rights; the linked reports remain subject to their publishers' rights and terms.

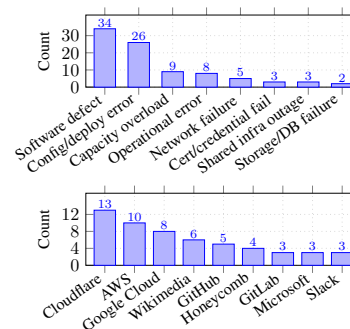


Figure 9: **Incident report statistics.** By type (top) and organization (bottom).

B Incident Model and Simulator Details

B.1 Terminology and incident-model format

The *incident model* is an abstraction that can be implemented in different ways, depending on the use case and requirements. We implement it as two YAML files called `system_description` and `scenario`. We chose the YAML format because it could be reliably generated by LLMs and enabled structural verification due to fixed pre-defined schema. These two objects separately define the system description and phased timeline, while overlapping on the emission mechanisms, as defined in Sec. 3.1. Specifically:

- `system_description`: Defines the static architecture of the system. It outlines the components, connectivity, log templates, background emissions, and request/path flows (detailing how these services interact).
- `scenario`: Defines the dynamic timeline of the incident. It details the initial trigger, the chronological sequence of failure events, the observable system behaviors, and the final resolution or mitigation steps.

To clarify the terminology mapping between the manuscript text and the released artifacts (codebase and prompts):

- *Incident model* (in the paper) $\equiv$ `system_description` + `scenario` (in prompts and as stored model representations) or `SYSTEM` + `SCENARIO` (in Python log generation scripts).
- *Incident simulator* (in the paper) $\equiv$ *log-generation script* (in the prompts).

Complete generator and verifier prompts for incident model generation, as well as all system description and scenario objects, can be found in our code and dataset repositories.

Emission mechanisms and parameters. Each mechanism is defined by the templates it uses and its timing rule:

- *Burst*. A single template is instantiated a specified number of times within a short interval. This models point events such as configuration changes, restarts, or transient error storms. Each `one_shots` entry in `scenario` specifies the template reference (`ref`), number of messages (`count`), and emitting hosts (`hosts`); its associated event determines the burst’s timing.
- *Background*. A single template is instantiated at a fixed rate for the duration of a phase, either per host or globally across the component’s hosts. This models steady-state activity such as health checks, heartbeats, or baseline traffic. In `system_description`, the component’s `beh` entries specify the template (`id`), rate in logs per minute (`per_min`), and scope (`scope`: `per_host` or `global`).
- *Path*. An ordered sequence of templates spanning multiple components is instantiated as a unit at a fixed request rate, with consecutive hops separated by sampled latencies. This models a request flow producing correlated logs, which share a trace identifier when tracing is enabled. Each request-flow entry under `flows` in `system_description` specifies the component path (`path`), ordered template sequence (`emit`), rate in requests per minute (`rpm`), and per-hop latency parameters in milliseconds (`latency_ms`), given as median and 95th-percentile values.

Scenario events select active request flows and scale emission rates and path latencies using `rate_multipliers` and `latency_multipliers`, respectively. Rates and latency parameters remain fixed until the next event.

Below are the `system_description` and `scenario` YAML schemas with comments specifying the role of different components. Some fields that are not directly related to the incident model specification are omitted for brevity.

```
<system_description>
sys:
  id: <system_name> # stable incident-model identifier
  desc: <incident-specific system description>
```

```

states: {n: normal, f: failure} # benchmark instances use normal/failure phases

components:
- id: <component_id> # machine-readable component identifier
  name: <human-readable name>
  svc: <service-name-or-null> # service label written into logs
  hosts: [<host1>, ...] # hostnames available to the simulator
  desc: <component role>
  to: # directed connectivity used for paths and system diagrams
    - dst: <component_id>
      proto: <https|grpc|kafka|redis|jdbc|tcp|...>
      desc: <edge role>
  logs: # log templates emitted by the component
    <log_id>:
      lvl: <DEBUG|INFO|WARN|ERROR|CRITICAL>
      msg: "<message template with {variables}>" # rendered log message
      vars: # variable types and domains
        <var>: {k: <i|f|ch|uuid|hex|ip|str>, v: <domain>}
      state_vars: # phase-specific variable domains
        n: {<var>: {k: <type>, v: <normal-domain>}} # normal phase domains
        f: {<var>: {k: <type>, v: <failure-domain>}} # failure phase domains
  beh: # background emissions
    n:
      emit:
        - id: <log_id>
          per_min: <float> # background emission rate
          scope: <per_host|global>
    f:
      emit:
        - id: <log_id>
          per_min: <float>
          scope: <per_host|global>

tracing:
  on: <true|false>
  origins: [<component_id>, ...]
  trace_id: {k: hex, v: 32}

flows: # path emissions
  n:
    req:
      - id: <normal_flow_id>
        rpm: <requests_per_minute>
        path: [<component_id>, ...] # logical request path through the system
        emit: [<component_id.log_id>, ...] # emitted logs along that path
        latency_ms: [[<p50>, <p95>], ...] # per-hop latency
        trace: <true|false>
    f:
      req:
        - id: <failure_flow_or_variant_id>
          ...
  assumptions:
    - <adaptation, traffic, volume, or visibility note>
</system_description>

<scenario>
scenario:
  id: <incident_id>
  title: <brief incident title>
  summary: <normal purpose, trigger, manifestation, impact>
  trigger_desc: <external action or precondition that caused the incident>
  time:
    total_minutes: <2..60>
  phases:
    n: {start_min: 0, end_min: <t>}
    f: {start_min: <t>, end_min: <total_minutes>}
  phases: # phased timeline
    n:
      flows: [<flow_id>, ...] # normal request families active before failure
    f:
      events:
        - order: <int>
          at_min: <minute> # event time; starts a new failure segment
          component: <component_id>
          event: <observed change>
          why: <mechanism and observability>
          flows: [<flow_id>, ...] # active/requested flow variants after this event
          rate_multipliers: {<flow_or_log_id>: <float>, ...} # flow IDs or component.log_id keys
          latency_multipliers: {<flow_id>: {p50: <float>, p95: <float>}, ...}
          one_shots: # burst emissions
            - ref: <component_id.log_id>

```

```

count: <int>
hosts: [<host>, ...]
</scenario>

```

B.2 High-level fidelity, consistency, and realism of the log streams

While low-level log realism can be formalized and quantified using basic statistical properties and comparisons with open-source production logs (App. C), high-level realism is harder to evaluate due to its ambiguity and lack of ground-truth production log streams with gradations of the properties of interest. We promote such high-level realism by producing log streams based on incident models grounded in real-world incident reports that provide realistic system structures, root causes, failure evolution, and observable evidence, and ensuring the incident models’ internal consistency and alignment with their corresponding incident reports. All prompts, along with generation and verification pipelines, are available in our code repo.

Faithfulness and consistency. We verify that each incident model preserves the structure and dynamics described in its source report. The LLM verifier receives both the report and the generated model, checking the reported failure mechanism, progression, symptoms, and component roles. It also assesses the plausibility and internal consistency of inferred components and failure dynamics, and whether the model contains enough detail to represent the incident. Algorithmic checks validate connectivity and references to log templates, hosts, and request flows. Acceptance requires ten consecutive verification passes (Sec. 3.2).

Observability and uncertainty. While incident models specify detailed internal system behaviors, adherence to the failure specified in the source report might require that not all aspects of such behaviors can be observed in logs. We represent partial observability by specifying (in the incident model) which internal system behaviors produce observable log evidence. The resulting log stream may leave some causes or mechanisms unresolved. Reference summaries are constrained to the observable effects and must express uncertainty where the evidence does not determine a conclusion. For example, the reference answer in App. D.2 identifies database saturation but leaves the cause of the overload unresolved.

Concurrent and distributed behavior. The incident model represents distributed incident dynamics through dependencies, concurrent request paths, and phase-dependent behavior across components and hosts. Interleaving path, background, and burst emissions allows normal activity and failure symptoms to coexist in the log stream. For example, the incident model below includes a web dashboard and an API for build agents that share a database. The API continues operating during the dashboard outage, and partial recovery produces successful and failed health checks. These overlapping signals require interpreting observations in the context of the affected components and incident phase.

Scope of realism. The incident model determines the range and complexity of behaviors represented in each log stream. We model the minimal subsystem needed to explain each incident, limiting unrelated background noise and incidental production complexity (Sec. 5).

B.3 Concrete incident-model example: 0b9e6bdf

Figures 10 and 11 visualize the system structure and incident progression timeline for the incident model 0b9e6bdf. While these figures provide an intuitive, high-level overview of the system state and incident timeline, the YAML excerpt below demonstrates how this data is actually represented and used in our pipeline. For brevity and readability, the full sequence is truncated in this snippet—specifically, intermediate events 2 and 4 have been omitted. This allows us to focus on the core failure mechanisms, their observable effects, and the ultimate resolution steps.

The YAML excerpt below shows the `system_description` and `scenario` for the incident model 0b9e6bdf. Some logs, flows, and fields are omitted for brevity.

```

sys:
  id: buildkite_dashboard_2016
  desc: >
    Hosted CI service with a web dashboard behind an ELB, Rails app instances,

```

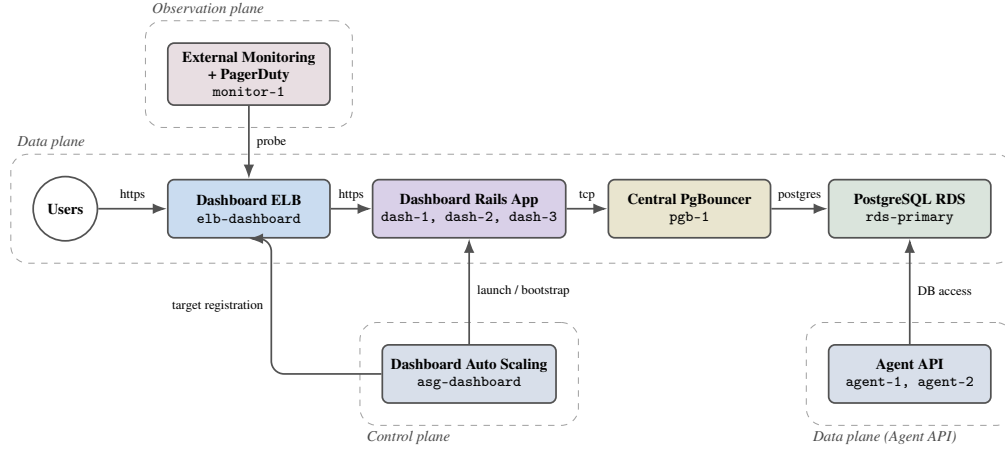


Figure 10: Compact system topology for incident model 0b9e6bdf, annotated by architectural plane. The dashboard data plane comprises the user and the middle-row components. The Agent API operates as an independent data-plane workload, sharing only the PostgreSQL database with the primary dashboard path. Auxiliary systems are positioned off the main request path, with external monitoring located in the observation plane and the Auto Scaling Group in the control plane.

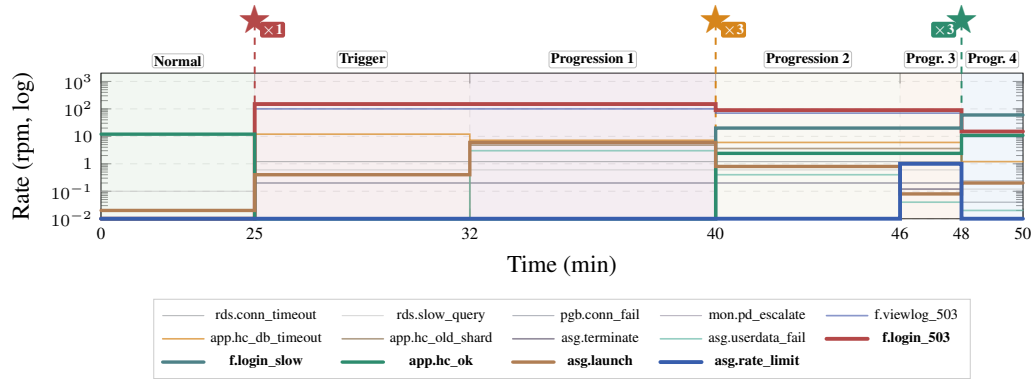


Figure 11: Compact timeline of incident model 0b9e6bdf. The figure illustrates the normal phase and all five ordered events of the 50-minute scenario. Thick lines highlight the dominant incident trajectory, thin lines represent secondary background activities, and stars denote one-shot log bursts at events 1, 3, and 5. Log and flow IDs are abbreviated for brevity.

```

central PgBouncer, and PostgreSQL. The dashboard degrades, but the Agent API
continues polling throughout the incident.

states: {n: normal, f: failure}

components:
- id: dashboard_elb
  name: Dashboard ELB
  hosts: [elb-dashboard]
  to: [{dst: dashboard_app, proto: https}]
  logs:
    req_no_healthy:
      lvl: WARN
      msg: "ELB request {method} {url_path} -> 503 no_healthy_backends latency_ms={lat_ms} req={req_id}"
    hc_result_500:
      lvl: WARN
      msg: "ELB healthcheck result target={target} status=500 latency_ms={lat_ms} req={req_id}"
- id: dashboard_app
  name: Dashboard Rails App
  hosts: [dash-1, dash-2, dash-3]
  to:
    - {dst: pgbouncer_central, proto: tcp}

```

```

- {dst: dashboard_elb, proto: https}
logs:
  req_started:
    lvl: INFO
    msg: "Started {method} {url_path} controller={controller} action={action} req={req_id}"
  healthcheck_db_timeout:
    lvl: ERROR
    msg: "Healthcheck failed: database ping timeout waited_ms={waited_ms} shard=two req={req_id}"
  healthcheck_old_shard_error:
    lvl: ERROR
    msg: "Healthcheck failed: database shard missing shard=one err=\"{err}\" req={req_id}"

- id: pgbouncer_central
  name: Central PgBouncer
  hosts: [pgb-1]
  to: [{dst: postgres_rds, proto: postgres}]
  logs:
    pool_stats:
      lvl: INFO
      msg: "pgbouncer stats db=buildkite active={active} waiting={waiting} max_client_conn={max_conn}
        avg_wait_ms={avg_wait_ms}"

- id: postgres_rds
  name: PostgreSQL RDS
  hosts: [rds-primary]
  to:
    - {dst: pgbouncer_central, proto: postgres}
    - {dst: dashboard_app, proto: postgres}
    - {dst: agent_api, proto: postgres}
  logs:
    rds_metrics:
      lvl: INFO
      msg: "rds metrics cpu_pct={cpu_pct} conn_used={conn_used} qps={qps} profile={profile}"

- id: agent_api
  name: Agent API
  hosts: [agent-1, agent-2]
  to: [{dst: postgres_rds, proto: postgres}]
  logs:
    agent_req:
      lvl: INFO
      msg: "agent-api {method} {url_path} -> {status} dur_ms={dur_ms} req={req_id}"

- id: autoscaling
  name: Dashboard Auto Scaling
  hosts: [asg-dashboard]
  to:
    - {dst: dashboard_elb, proto: aws-api}
    - {dst: dashboard_app, proto: ec2-bootstrap}
  logs:
    userdata_fetch_rev_failed:
      lvl: ERROR
      msg: "userdata fetch deploy revision failed url=https://buildkite.com/_secret/version err=\"{err}
    }\"
    ami_rollout_started:
      lvl: INFO
      msg: "ami rollout started ami={ami_id} revision_source={source} reason=\"{reason}\"

- id: monitoring
  name: External Monitoring + PagerDuty
  hosts: [monitor-1]
  to: [{dst: dashboard_elb, proto: https}]
  logs:
    pagerduty_incident:
      lvl: ERROR
      msg: "pagerduty incident opened service=dashboard severity=high incident_id={incident_id}"

flows:
  n:
    req:
      - id: dashboard_login
        rpm: 120.0
        path: [dashboard_elb, dashboard_app, pgbouncer_central,
          postgres_rds, dashboard_app, dashboard_elb]
        emit: [dashboard_app.req_started,
          dashboard_app.req_completed_ok,
          dashboard_elb.req_ok]
        latency_ms: [[1, 10], [30, 800], [1, 20]]
      - id: dashboard_elb_healthcheck
        rpm: 12.0
        path: [dashboard_elb, dashboard_app, dashboard_elb]

```

```

        emit: [dashboard_elb.hc_access,
               dashboard_app.healthcheck_ok,
               dashboard_elb.hc_result_ok]
- id: agent_job_poll
  rpm: 300.0
  path: [agent_api, postgres_rds]
  emit: [agent_api.agent_req]

f:
  req:
    - id: dashboard_login_503
      rpm: 150.0
      path: [dashboard_elb]
      emit: [dashboard_elb.req_no_healthy]
    - id: dashboard_elb_healthcheck_fail_db
      rpm: 12.0
      path: [dashboard_elb, dashboard_app, dashboard_elb]
      emit: [dashboard_elb.hc_access,
             dashboard_app.healthcheck_db_timeout,
             dashboard_elb.hc_result_500]
    - id: dashboard_elb_healthcheck_fail_old_shard
      rpm: 12.0
      path: [dashboard_elb, dashboard_app, dashboard_elb]
      emit: [dashboard_elb.hc_access,
             dashboard_app.healthcheck_old_shard_error,
             dashboard_elb.hc_result_500]

scenario:
  id: buildkite_outage_aug_2016_compressed
  time:
    total_minutes: 50
  phases:
    n: {start_min: 0, end_min: 25}
    f: {start_min: 25, end_min: 50}
  phases:
    f:
      events:
        - order: 1
          at_min: 25
          component: postgres_rds
          event: >
            PostgreSQL saturation begins; DB-dependent dashboard health checks fail,
            and the ELB starts returning 503 because targets are drained.
          flows: [dashboard_elb_healthcheck_fail_db, dashboard_login_503,
                  dashboard_view_build_log_503, agent_job_poll]
          rate_multipliers:
            dashboard_login_slow: 0.0
            dashboard_elb_healthcheck_fail_old_shard: 0.0
            dashboard_elb_healthcheck_ok: 0.0
            autoscaling_userdata_fetch_rev_failed: 0.0
            autoscaling_launch_failed_rate_limit: 0.0
            postgres_rds.rds_metrics_recovered: 0.0
            pgbouncer_central.pool_stats_recovered: 0.0
          latency_multipliers:
            dashboard_elb_healthcheck_fail_db: {p50: 1.2, p95: 1.2}
            agent_job_poll: {p50: 1.1, p95: 1.2}
          one_shots:
            - {ref: monitoring.pagerduty_incident, count: 1, hosts: [monitor-1]}

        - order: 3
          at_min: 40
          component: autoscaling
          event: >
            Operators lock desired capacity and register borrowed instances to the ELB;
            some health checks recover, but many dashboard requests still fail or run slowly.
          flows: [dashboard_elb_healthcheck_ok, dashboard_elb_healthcheck_fail_db,
                  dashboard_elb_healthcheck_fail_old_shard, dashboard_login_slow,
                  dashboard_login_503, dashboard_view_build_log_503]
          rate_multipliers:
            dashboard_login_503: 0.6
            dashboard_view_build_log_503: 0.7
            dashboard_login_slow: 1.0
            dashboard_elb_healthcheck_fail_db: 0.5
            dashboard_elb_healthcheck_fail_old_shard: 0.3
            dashboard_elb_healthcheck_ok: 0.2
            autoscaling_instance_launch: 2.0
            autoscaling_instance_terminate: 2.0
            autoscaling_userdata_fetch_rev_failed: 2.0
          latency_multipliers:
            dashboard_login_slow: {p50: 1.2, p95: 1.3}
          one_shots:

```

```

- {ref: autoscaling.desired_capacity_set, count: 1, hosts: [asg-dashboard]}
- {ref: autoscaling.register_instance_to_elb, count: 2, hosts: [asg-dashboard]}

- order: 5
at_min: 48
component: postgres_rds
event: >
  Operators upgrade RDS, roll out a fixed AMI bootstrap path, and roll back
  pooling topology; health checks mostly recover and 503s drop sharply.
flows: [dashboard_elb_healthcheck_ok, dashboard_elb_healthcheck_fail_db,
  dashboard_login_slow, dashboard_login_503, dashboard_view_build_log_503]
rate_multipliers:
  dashboard_elb_healthcheck_ok: 0.9
  dashboard_elb_healthcheck_fail_db: 0.1
  dashboard_elb_healthcheck_fail_old_shard: 0.0
  dashboard_login_503: 0.1
  dashboard_view_build_log_503: 0.1
  dashboard_login_slow: 3.0
  postgres_rds.rds_metrics: 0.0
  postgres_rds.rds_metrics_recovered: 1.0
  postgres_rds.connection_timeout: 0.2
  postgres_rds.slow_query: 0.2
  pgbouncer_central.pool_stats: 0.0
  pgbouncer_central.pool_stats_recovered: 1.0
  pgbouncer_central.server_connect_failed: 0.2
  autoscaling.userdata_fetch_rev_failed: 0.1
  autoscaling.instance_launch: 0.5
  autoscaling.instance_terminate: 0.5
  autoscaling.launch_failed_rate_limit: 0.2
latency_multipliers:
  dashboard_login_slow: {p50: 0.7, p95: 0.6}
  agent_job_poll: {p50: 0.95, p95: 0.95}
one_shots:
- {ref: postgres_rds.modify_instance, count: 1, hosts: [rds-primary]}
- {ref: autoscaling.ami_rollout_started, count: 1, hosts: [asg-dashboard]}
- {ref: pgbouncer_central.config_change, count: 1, hosts: [pgb-1]}

```

B.4 Incident model statistics

Table 2 presents the summary statistics for the generated incident models. We deliberately design our generation prompts to target a compact incident-specific model that is small enough for reliable generation and verification by the LLM, yet sufficient to capture realistic system structures and incident progression. As detailed in the table, the generated models exhibit significant variance and scale, with some instances reaching up to 211 distinct hosts and 180 variable slots, demonstrating the robust generation capabilities of our pipeline. The median accepted model has six components, 27 log templates, eight request-flow variants, and four failure events.

Table 2: Structural statistics of the 90 accepted incident models.

Quantity	Mean	SD	Median	Range
Components	6.11	1.45	6.0	3–10
Hosts	16.38	23.66	12.5	7–211
Component edges	7.91	2.90	7.0	3–16
Log templates	26.96	7.85	27.0	12–46
Variable slots	81.84	25.28	78.0	42–180
Flow variants	8.34	3.44	8.0	3–18
Normal flow variants	2.76	1.34	3.0	1–7
Failure flow variants	5.59	2.32	5.0	2–11
Background emission sources	14.28	5.54	14.0	5–28
Failure events	4.22	1.13	4.0	2–7
Total scenario minutes	48.21	8.87	50.0	24–60
Normal-phase minutes	22.04	4.06	21.5	12–30
Failure-phase minutes	26.17	6.48	25.0	12–45

B.5 Simulator example and generated logs

For each accepted incident model, we generate a deterministic Python script that embeds the model and converts the scenario events into piecewise-constant failure segments. Within these segments,

the script instantiates request-flow families, emits background logs and one-shot events, sorts all rows chronologically, and outputs a standard CSV format (timestamp, level, message, trace_id, service, host). To give a representative example, executing the simulator for incident 0b9e6bdf produces 43368 log message lines. The following excerpt summarizes the core logic: it derives failure segments from the scenario, applies rate and latency controls, and emits background logs, one-shot event markers, and request-flow instances inside each segment.

```
def derive_failure_segments() -> List[Dict[str, Any]]:
    fstart = SCENARIO["scenario"]["time"]["phases"]["f"]["start_min"]
    fend = SCENARIO["scenario"]["time"]["phases"]["f"]["end_min"]
    events = sorted(SCENARIO["scenario"]["phases"]["f"]["events"],
                    key=lambda e: (e["at_min"], e["order"]))

    boundaries = sorted(set([fstart] + [e["at_min"] for e in events] + [fend]))
    rate_flow, rate_bg, lat_flow = {}, {}, {}
    segs = []
    events_by_min: Dict[int, List[Dict[str, Any]]] = {}
    for e in events:
        events_by_min.setdefault(int(e["at_min"]), []).append(e)

    for s, e in zip(boundaries[:-1], boundaries[1:]):
        for ev in events_by_min.get(int(s), []):
            for k, m in (ev.get("rate_multipliers") or {}).items():
                (rate_bg if "." in k else rate_flow)[k] = float(m)
            for fk, mult in (ev.get("latency_multipliers") or {}).items():
                lat_flow[fk] = {"p50": float(mult.get("p50", 1.0)),
                               "p95": float(mult.get("p95", 1.0))}

        segs.append({
            "start_min": int(s),
            "end_min": int(e),
            "rate_flow": dict(rate_flow),
            "rate_bg": dict(rate_bg),
            "lat_flow": dict(lat_flow),
        })
    return segs

FAILURE_SEGS = derive_failure_segments()

def simulate() -> pd.DataFrame:
    base_time = datetime(2016, 8, 10, 12, 0, 0, tzinfo=timezone.utc)
    rows: List[LogRow] = []

    # Normal background emissions and normal flows.
    ...

    # Failure segments: one-shots, background, then flow instances.
    for seg in FAILURE_SEGS:
        seg_start_dt = base_time + timedelta(minutes=seg["start_min"])
        seg_end_dt = base_time + timedelta(minutes=seg["end_min"])

        for os in seg.get("one_shots", []):
            emit_log(rows, ..., os["ref"], "f", bound={}, host_override=host)

        for cid, cinfo in COMP.items():
            for src in cinfo.get("beh", {}).get("f", {}).get("emit", []):
                mult = get_rate_multiplier_bg(seg, f"{cid}.{src['id']}")
                ...
                emit_log(rows, t, f"{cid}.{src['id']}", "f", bound={}, host_override=host)

        for fdef in SYSTEM["flows"]["f"]["req"]:
            flow_id = fdef["id"]
            eff_rpm = float(fdef["rpm"]) * get_rate_multiplier_flow(seg, flow_id)
            ...
            lat_mult = get_flow_latency_multiplier(seg, flow_id)
            simulate_flow_instance(rows, "f", fdef, st, lat_mult, inst_key)

    df = pd.DataFrame(...).sort_values("timestamp", kind="mergesort")
    df["timestamp"] = df["timestamp"].apply(iso_z)
    return df[["timestamp", "level", "message", "trace_id", "service", "host"]]
```

The generated logs provide a concrete realization of the incident model, translating the abstract topology and timeline into an observable diagnostic artifact. The short excerpt below shows normal operation, the onset of the dashboard outage, the later bootstrap failure on replacement instances, and recovery attempts near the end of the window.

```
timestamp,level,message,trace_id,service,host
2016-08-10T12:00:00.213Z,INFO,Started GET /login controller=sessions action=new req=099e73940896feb9,,
buildkite-web,dash-2
```

```

2016-08-10T12:00:00.243Z,INFO,Completed 200 in 30ms req=099e73940896feb9,,buildkite-web,dash-2
2016-08-10T12:00:00.282Z,INFO,ELB request GET /login -> 200 target=dash-2 latency_ms=70 req=099
e73940896feb9,,aws-elb,elb-dashboard
2016-08-10T12:25:00.686Z,ERROR,pagerduty incident opened service=dashboard severity=high incident_id=2722
cd23,,monitoring,monitor-1
2016-08-10T12:25:02.641Z,ERROR,Healthcheck failed: database ping timeout waited_ms=200 shard=two req=3390
cb390814cd36,,buildkite-web,dash-1
2016-08-10T12:25:02.685Z,WARN,ELB healthcheck result target=dash-1 status=500 latency_ms=244 req=3390
cb390814cd36,,aws-elb,elb-dashboard
2016-08-10T12:25:03.027Z,WARN,ELB request GET /login -> 503 no_healthy_backends latency_ms=15 req=81
d5a8c0b5b71350,,aws-elb,elb-dashboard
2016-08-10T12:32:09.807Z,ERROR,"userdata fetch deploy revision failed url=https://buildkite.com/_secret/
version err=\"connection refused\"",,aws-asg,asg-dashboard
2016-08-10T12:32:19.060Z,ERROR,"Healthcheck failed: database shard missing shard=one err=\"relation does
not exist\" req=344f67da016bb76d",,buildkite-web,dash-3
2016-08-10T12:48:00.459Z,INFO,"pgbouncer config change pooling_mode=per_host reason=\"
rollback_from_central\"",,pgbouncer,pgb-1
2016-08-10T12:48:00.472Z,INFO,"ami rollout started ami=i-a5f658ea revision_source=s3_file reason=\"
fix_bootstrap_revision_source\"",,aws-asg,asg-dashboard

```

C Log-Statistics Comparison Details

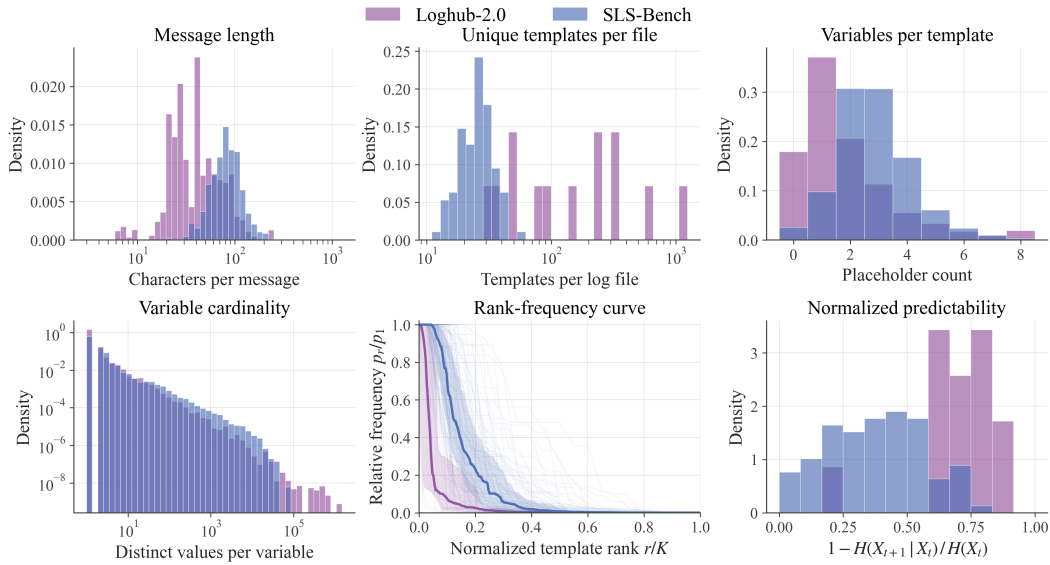


Figure 12: Distributional comparison between Loghub-2.0 and SLS-BENCH. Panels show message length, unique templates per file, variables per template, variable cardinality, normalized template rank–frequency curves, and adjacent-template predictability.

We compare SLS-BENCH logs with Loghub-2.0 log files from the Hadoop, HDFS, OpenStack, Spark, Zookeeper, BGL, Thunderbird, HPC, Proxifier, Apache, OpenSSH, HealthApp, Mac, and Linux datasets. For Loghub-2.0, templates are the released `EventTemplate` annotations. For SLS-BENCH, templates come from the incident-model system descriptions.

Fig. 12 shows six distributions. *Message length* is the number of characters in the emitted message text. *Unique templates per file* counts the distinct observed templates in each log stream. *Variables per template* counts the number of placeholder slots in each observed template. *Variable cardinality* counts the number of distinct observed values for each template slot within a file.

The *rank–frequency* panel summarizes how log volume is distributed across templates. For each file, we sort templates by descending frequency and plot $(r/K, p_r/p_1)$, where K is the number of observed templates, r is the one-indexed rank, p_r is the empirical share of the template at rank r , and p_1 is the share of the most frequent template. The panel overlays the per-file curves and the dataset-level median with their interquartile range band. Adjacent-template *normalized predictability* measures local stream regularity using the observed template sequence: $1 - H(X_{t+1} | X_t) / H(X_t)$, where X_t is the template at position t and H is Shannon entropy. Values near one indicate that the

next template is almost determined by the current template; values near zero indicate weak one-step predictability.

Comparing the distributions for the two datasets reveals the following similarities and differences:

- The message length distributions overlap significantly, with SLS-BENCH being narrower and on average longer. In part, this is due to our log messages containing more variables and often including various identifiers such as trace or request IDs that are typically long strings of random characters.
- The most notable difference is the number of unique templates. While for some Loghub-2.0 systems the number of templates overlaps with SLS-BENCH, there are many cases with hundreds of unique templates (notably Mac and Linux systems that have 626 and 338 templates, respectively). This discrepancy is expected as a consequence of our design choice to infer only the minimal relevant system subset that represents and explains the incident. This prevents our incident model from having multiple components unrelated to the failure and contributing their templates to the log stream. This aspect can be easily adjusted by injecting real or synthetic log templates unrelated to the incident into the log stream to reduce the signal-to-noise ratio.
- The number of variables, while being slightly larger on average for our logs, significantly overlaps with Loghub-2.0, indicating realistic log template structure. Similarly, variable cardinality distributions have very similar shapes across the datasets, indicating that, similarly to real logs, the majority of variables are only instantiated with a few unique values. Only a small fraction has a large number of unique values.
- Another notable difference is in how templates are used in real and our synthetic logs. The rank–frequency curves show that both SLS-BENCH and Loghub-2.0 have distinct heavy-tailed curves, indicating that there is a relatively small fraction of templates that are used most often, and a long tail of templates that are used very rarely. Real system logs, however, demonstrate a notably steeper drop, indicating that a few templates dominate the real-world log streams. SLS-BENCH, on the other hand, shows larger template utilization without extreme skewness towards a small fraction of dominant templates. These results are consistent with the previous result on the number of unique templates in both datasets.
- Finally, we compare the local structure of SLS-BENCH and Loghub-2.0 in terms of normalized predictability. We see that real log streams tend to be much more predictable than SLS-BENCH log messages. This is consistent with rank–frequency results where a few dominant templates occupy most of the log stream, making it more monotone and predictable. Our log messages, on the other hand, overlap bursts, background, and path emissions without repeating the same log template multiple times, making our synthetic log streams less locally predictable.

D Fixed Question and Example Reference Answer

Each benchmark problem uses the same question shown in App. D.1. For problem 0b9e6bdf, the answer is given in App. D.2; it is shortened to keep the appendix compact, while the released benchmark files contain the full reference answer.

D.1 Fixed question

Analyze the provided log file and write an incident investigation report.

Use the logs to reconstruct what system this is, how it normally behaves, when the behavior changes, and the single best diagnosis of the incident. Base the report on evidence visible in the logs plus reasonable domain knowledge. When the logs do not fully determine a conclusion, state the best-supported interpretation and make the uncertainty explicit.

Use exactly this structure:

Incident synopsis

Give a short bullet list with one bullet per phase in chronological order. Each bullet should name the phase and summarize its main observable state or change in one sentence.

Reconstructed system

Use exactly these '###' subsections in this order:

- Inferred purpose
- Main services/components
- Main interactions

Describe the system only to the extent the logs support it.

Prioritize the core request or data path. Mention monitoring, probes, operator tooling, deployment or restore tooling, and other support-plane actors only when they are diagnostically central.

Prefer names and roles that are visible in service names, hostnames, trace-linked interactions, or message text. Do not introduce hidden component labels or concrete external dependencies unless the logs support them.

When a role, dependency, or interaction is inferred rather than explicit, use appropriately hedged language such as ‘appears to’, ‘likely’, or ‘suggests’.

Timeline

Partition the full log into phases with [inclusive, exclusive) time ranges in minutes from the start of the file. Include the normal baseline, the trigger phase, and each subsequent failure-progression phase. Compare each phase primarily to the immediately preceding phase, but you may also refer explicitly to earlier phases when that helps explain the current one.

Use one ‘###’ heading per phase in chronological order:

- ‘### Baseline -- [start, end)’
- ‘### Trigger -- [start, end)’
- ‘### Progression 1 -- [start, end)’
- ‘### Progression 2 -- [start, end)’
- and so on as needed

Under each phase heading, use exactly these ‘####’ subsections in this order:

- Summary
- Evidence
- Reasoning

For each phase:

- ‘Summary’ should state the main phase-level claim in 2--3 sentences. For non-baseline phases, make the main visible change relative to the immediately preceding phase clear. Do not turn this subsection into an evidence dump.
- ‘Evidence’ should stay anchored to the log surface. Cite the concrete evidence that supports the phase reading: representative log templates with “...” placeholders, request or trace-linked sequences, retry patterns, bursty operational markers, service/host patterns, severity mix, and major appearance/disappearance or rate/timing shifts. For each cited item, say what is directly visible and why it is relevant. Keep ‘Evidence’ close to the log surface rather than turning it into a mini-diagnosis.
- ‘Reasoning’ should explain what the stated evidence means and how it supports the summary. It may use domain knowledge and reasonable speculation, but it must stay tied to stated evidence from the current phase and may also refer explicitly to earlier phases when that materially clarifies the claim. Begin this subsection with exactly one line of the form ‘Certainty: highly likely’, ‘Certainty: moderately supported’, or ‘Certainty: speculative’. Use the certainty label that matches the weakest material inferential step in the subsection. Use ‘highly likely’ when the conclusion is directly visible in the logs and follows from them without extra reasoning beyond what the logs explicitly show. Use ‘moderately supported’ when the evidence is only partially visible or when ordinary domain knowledge or a bridging inference is needed. Use ‘speculative’ when the evidence is indirect or incomplete and multiple plausible explanations remain. In the baseline, use this subsection to infer the normal system structure and behavior. In later phases, use it to explain what most likely changed operationally, how that affects the system, and what it says about the incident progression.

Primary diagnosis

Use exactly these ‘####’ subsections in this order:

- Summary
- Evidence
- Reasoning

For ‘## Primary diagnosis’:

- ‘Summary’ should state only the core fault chain in plain terms, preferably in 1--2 sentences. Keep amplifiers, mitigation steps, and recovery detail out of this subsection unless they are part of the core diagnosis itself.
- ‘Evidence’ should cite the main cross-phase evidence that supports that diagnosis. For each cited item, say what is directly visible and why it is relevant.
- ‘Reasoning’ should explain why that evidence makes this diagnosis the best-supported interpretation. Put amplifiers, mitigation nuance, recovery detail, material alternatives, and unresolved ambiguities here when they matter, and begin with exactly one certainty line using the same scale as above. Use the certainty label that matches the weakest material inferential step in this subsection.

Additional requirements:

- When describing log templates, replace variable parts with “...”
- If the same observable request-flow signature appears again later, reference the earlier description and describe only the observable differences
- Use qualitative language for rate and latency changes unless “at least 2x” is warranted
- Write like an engineer producing a handoff-quality investigation note

D.2 Example reference answer excerpt

Incident synopsis

- Baseline -- Dashboard and Agent API operate normally: ELB routes to healthy Rails targets, health checks return 200, and PostgreSQL/PgBouncer show low contention.
- Trigger -- PostgreSQL saturates: connection timeouts and slow queries appear, dashboard ‘/healthcheck’ fails with DB ping timeouts, ELB drains targets, and user requests flip to widespread 503s.
- Progression 1 -- Auto Scaling churn accelerates and recovery fails: new instances fail bootstrap (can’t fetch deploy revision) and many health checks now fail with a “missing shard” error.
- Progression 2 -- Partial mitigation yields partial service: desired capacity is locked and instances are registered to the ELB, producing a mix of 200/500 health checks, fewer 503s, and some successful-but-slow logins.
- Progression 3 -- Scaling hits EC2 API limits: ‘RequestLimitExceeded’ appears, successful launches/bootstraps drop, and the system remains degraded with continued dashboard errors.
- Progression 4 -- Major recovery actions applied: RDS is modified, AMI rollout and PgBouncer config change occur, DB/pgbouncer metrics shift to a “recovered” profile, health checks mostly succeed, but latency and intermittent errors persist.

Reconstructed system

Inferred purpose

A hosted CI service with a web "dashboard" (logins, build pages/log viewing, docs) served behind an AWS load balancer, plus a separate Agent API for build agents polling for jobs.

Main services/components

- 'aws-elb' ('elb-dashboard'): Front door/load balancer for the dashboard; emits access and health-check logs.
- 'buildkite-web' ('dash-...'): Rails-like dashboard application instances; handles user endpoints and '/healthcheck'.
- 'pgbouncer' ('pgb-1'): Connection pooler in front of the primary database; emits pool stats and connection failure warnings.
- 'aws-rds-postgres' ('rds-primary'): PostgreSQL primary; emits periodic "metrics" snapshots plus connection timeout / slow-query warnings.
- 'buildkite-agent-api' ('agent-...'): API for build agents; emits request logs and explicit retry markers.
- 'aws-asg' ('asg-dashboards'): Auto Scaling activity for dashboard instances; emits launch/terminate plus bootstrap/EC2 API error markers.
- 'monitoring' ('monitor-1'): Synthetic probes and PagerDuty events that detect/record unavailability.

Main interactions

- Dashboard request path appears to be: client -> 'aws-elb' -> 'buildkite-web' -> (DB access via 'pgbouncer') -> 'aws-rds-postgres' -> back to 'aws-elb', with the ELB also running periodic '/healthcheck' probes against each 'buildkite-web' instance.
- The Agent API appears to talk directly to the same PostgreSQL ('aws-rds-postgres') and will retry on failures.
- Auto Scaling launches/terminates dashboard instances and registers them to the ELB; instance bootstrapping appears to depend on fetching a deploy revision from a dashboard URL (inferred from bootstrap error logs referencing an HTTPS URL on the same domain).

Timeline
[Phase-by-phase timeline omitted for space; the released benchmark file contains all phase sections.]

Primary diagnosis

Summary

PostgreSQL saturation caused the dashboard's DB-dependent '/healthcheck' to fail, leading the ELB to mark all dashboard targets unhealthy and return widespread 503s; recovery via Auto Scaling failed because new instances could not bootstrap (deploy revision fetch depended on the down site) and many ran an outdated health check referencing a missing DB shard.

Evidence

- Cross-service health-check failure and ELB draining symptoms:
 - App: "Healthcheck failed: database ping timeout waited_ms=... shard=two req=..."
 - ELB: "ELB healthcheck result target=... status=500 ..."
 - ELB user impact: "ELB request ... -> 503 no_healthy_backends ..."
- The sequence shows health checks failing and user traffic failing due to no healthy backends.
- Database overload and connectivity failures coincide with health-check failures:
 - "rds metrics ... profile=..." plus frequent "postgres connection timeout ..." and "slow query ..."
 - PgBouncer elevated waiting: "pgbouncer stats ... waiting=... avg_wait_ms=..."
- Auto Scaling's inability to replace capacity is visible and tied to bootstrap/version issues:
 - "asg launch instance=..." followed by "asg terminate instance=... reason="failed_elb_healthcheck""
 - "userdata fetch deploy revision failed url=https://buildkite.com/_secret/version err=..."
 - Distinct app health-check failure: "Healthcheck failed: database shard missing shard=one err=..." req=..."
- Late improvements align with DB + bootstrap/pooling fixes:
 - "rds modify instance_id=buildkite from=... to=... apply_immediately=true"
 - "ami rollout started ... revision_source=s3_file ..."
 - "pgbouncer config change pooling_mode=per_host ..."
- followed by "rds metrics ... profile=recovered", "pgbouncer stats ... profile=recovered", and mostly-200 ELB health checks.

Reasoning

Certainty: moderately supported

The strongest, most direct chain in the logs is: DB overload (metrics/timeouts/slow queries) -> app '/healthcheck' explicitly failing due to DB ping timeout -> ELB recording 500 health-check results -> ELB returning 503 "no healthy backends" for user requests. That explains the initial outage state with only standard domain knowledge about how load balancer health checks gate routing. The prolonged inability to recover via scaling is best explained by the combination of (a) repeated launch/terminate cycles due to failed ELB health checks, (b) repeated bootstrap failures to fetch a deploy revision from a dashboard URL (a circular dependency under outage), and (c) a second, distinct health-check failure mode "database shard missing shard=one," which is most consistent with instances running mismatched/old code or configuration against a decommissioned shard. Alternatives like a pure ELB misconfiguration or a non-DB app crash are less consistent with the explicit DB timeout/shard-missing messages and the DB-side saturation warnings. The remaining uncertainty is the exact root cause of the DB saturation (e.g., traffic spike vs capacity reduction vs pooling change), because the logs show the saturation clearly but do not, by themselves, prove why the DB became overloaded--only that it did and that it dominated health-check and request behavior during the incident.

E LLM Judge Rubric

The judge receives the fixed question, the reference answer, and the predicted answer. The question defines the required report structure and certainty scale, while the reference answer defines the target incident story: the primary system, baseline behavior, trigger, progression, diagnosis, and uncertainty. The judge does not see the logs and is not allowed to use outside information.

The rubric scores seven axes using four labels: *yes*, *partial minor*, *partial major*, and *no*. *Yes* means the axis matches cleanly. *Partial minor* means the answer is mostly correct with only a local or non-central issue. *Partial major* means there is a material but bounded defect. *No* means the axis is absent or materially wrong. The axes separate report structure from incident semantics so that formatting errors are not confused with wrong diagnosis or chronology. Axis definitions are provided in Table 3.

The rubric allows useful extra specificity. Exact values, identifiers, hostnames, timestamps, counts, or log-template fragments are not penalized merely because the reference states the same point qualitatively. Such details are acceptable when they remain consistent with the reference incident story and are supported by the prediction’s own observations. Conflicting or unsupported specificity is scored under support alignment; overconfidence about an unresolved trigger, dependency direction, or diagnosis is scored under uncertainty unless it changes the causal story itself. The judge records only the deepest independent mismatches and material extra claims, rather than counting one root error repeatedly.

The judge then assigns one of five ordered overall labels (Table 4) by combining the per-axis scores based on instructions provided in the rubric. Diagnosis and timeline progression are the main story anchors. Support alignment and uncertainty determine whether the recovered story is properly supported and stated with the right level of confidence. System reconstruction provides context, and report structure determines whether the answer is a complete substitute for the reference.

For overall evaluation, we map the labels *mismatch*, *weak match*, *partial match*, *good match*, and *strong match* to the numerical values 0, 0.25, 0.50, 0.75, and 1.00, respectively, as described in Sec. 3.3. When converting per-axis labels to per-axis scores, we map *no*, *partial major*, *partial minor*, and *yes* to 0.0, 0.33, 0.67, and 1.0, respectively.

The complete judge rubric is available in our code repository.

Table 3: Evaluation axes used by the LLM judge.

Axis	Definition
Structure	Compliance with the required report form: headings, subsections, certainty lines, and one synopsis bullet per reference phase. Semantic correctness is scored on the other axes.
System reconstruction	Recovery of the same main system purpose, actors, and interactions. Extra plausible names or implementation details are allowed unless they change the main system, recenter a dependent component, or reverse a key dependency.
Timeline structure	Agreement with the reference phase count, order, and time ranges. Small boundary drift and reference-preserving phase splits or merges are minor unless they hide an important state transition.
Timeline progression	Recovery of the same baseline, trigger, major phase-to-phase changes, and final observable state. Missing or contradicting a major progression step is material.
Diagnosis	Recovery of the same core fault chain. A more specific restatement is acceptable when it stays within the reference diagnosis and does not turn an unresolved mechanism into a required causal step.
Support alignment	Whether the cited observations and reasoning support the same qualitative conclusions as the reference. This axis handles extra exactness, magnitude claims, local sequencing, and other log-surface details.
Uncertainty	Match to the reference hedging on material uncertainty about the trigger, progression, a key dependency direction, or the main diagnosis. Mere log-surface precision is ignored on this axis.

Table 4: Overall verdicts used by the LLM judge.

Verdict	Definition
Strong match	A faithful substitute for the reference: the answer is semantically and structurally equivalent to the reference and covers all material content.
Good match	A reliable recovery of the reference, but not a full substitute. The main incident story is the same, and any remaining semantic issues are minor and non-material.
Partial match	A useful answer that gets the main causal story broadly right, but a material bounded defect or an accumulation of smaller semantic defects prevents using it as a reliable substitute.
Weak match	The reference is still recognizable, but major defects dominate. The answer preserves fragments, observations, or broad phases, but does not cleanly recover the incident story.
Mismatch	The prediction tells a materially different incident story and is not usable as a recovery of the reference. This includes switching to a different primary system, trigger, or fault chain.

F Dataset Construction Cost

Table 5 shows the total benchmark token usage and cost breakdown. We use the following rates for the GPT-5.2 (high) model:

- Input tokens: \$1.75 per million tokens;
- Cached input tokens: \$0.18 per million tokens;
- Output tokens: \$14 per million tokens.

We report token usage and costs for each stage of the pipeline, as well as a breakdown over the generator and verifier steps within each stage.

Table 5: Token usage and cost breakdown across the three stages of the data generation pipeline. Each stage reports the generator and the verifier usage and cost. Cached tokens are a subset of the input tokens, and reasoning tokens are a subset of the output tokens.

Stage	Category	Calls	Tokens				Cost [\$]			
			Input	Cached	Output	Reasoning	Input	Cached	Output	Total
Incident model	Generator	643	11,458,543	3,187,968	8,254,739	2,754,053	14.47	0.57	115.57	130.61
	Verifier	2,730	44,687,360	18,951,040	14,351,580	13,511,415	45.04	3.41	200.92	249.37
	<i>Subtotal</i>	<i>3,373</i>	<i>56,145,903</i>	<i>22,139,008</i>	<i>22,606,319</i>	<i>16,265,468</i>	<i>59.51</i>	<i>3.99</i>	<i>316.49</i>	<i>379.99</i>
Incident simulator	Generator	373	9,188,128	2,078,464	6,793,850	1,837,210	12.44	0.37	95.11	107.93
	Verifier	1,855	47,917,225	19,464,960	12,079,385	11,704,445	49.79	3.50	169.11	222.41
	<i>Subtotal</i>	<i>2,228</i>	<i>57,105,353</i>	<i>21,543,424</i>	<i>18,873,235</i>	<i>13,541,655</i>	<i>62.23</i>	<i>3.88</i>	<i>264.23</i>	<i>330.34</i>
Reference answer	Generator	180	3,167,550	877,184	1,044,137	103,753	4.01	0.16	14.62	18.78
	Verifier	1,350	27,855,395	16,322,560	1,688,260	1,410,370	20.18	2.94	23.64	46.76
	<i>Subtotal</i>	<i>1,530</i>	<i>31,022,945</i>	<i>17,199,744</i>	<i>2,732,397</i>	<i>1,514,123</i>	<i>24.19</i>	<i>3.10</i>	<i>38.25</i>	<i>65.54</i>
Pipeline total		7,131	144,274,201	60,882,176	44,211,951	31,321,246	145.93	10.97	618.97	775.87

We note the asymmetry between generation and verification costs, with verification being around twice as expensive. The majority of the generation cost comes from feedback iterations where incorporation of every update requires the generator to completely re-generate the object with that update included. Improving the efficiency of feedback handling during generation (e.g., via in-place editing) could substantially reduce the generation costs, further increasing the asymmetry between generation and verification expenses.

G Evaluation Details

We use LangChain [Chase, 2022] to implement our ReAct-style [Yao et al., 2022] agent that iteratively receives the input, reasons about the next action, and produces the next action as a tool call. The agent has access to a stateful Python interpreter that imports pandas [McKinney, 2010] and has access to all standard Python packages. This tool allows the agent to iteratively investigate the log data without rewriting the code in every new iteration.

We access all evaluated OpenAI models on Microsoft Azure; all other models (Anthropic and open-weight) are accessed via Amazon Bedrock. We use default sampling parameters for all models (temperature=1.0, top-p=1.0). We use a high (500k) output token limit per API request, and allow up to 200 tool calls per problem. For OpenAI models, we use default Azure input caching. Amazon Bedrock provides caching only for Anthropic models but not for the open-weight models.

For all models, we use the following system prompt:

You are a log analysis agent. Your task is to answer a user’s question about log data.

Tools

python_notebook - executes Python in a stateful IPython kernel. State (variables, imports, DataFrames) is preserved across calls. pandas is pre-imported as pd.

Log Data

A single CSV log file is available at {logs_path}. Load it with `pd.read_csv(...)` when needed.

Rules

- Do NOT print large objects from `python_notebook` directly. Assign to variables, check sizes, and inspect in smaller parts to keep tool output concise.
- Do NOT create, modify, or delete any files.
- When you have the answer, respond with only the final answer text; no explanations or auxiliary text - just the complete final answer.

The user prompt consists of the question template shown in App. D.

H Tool-call-budget sensitivity

We test how the number of Python interpreter calls available to the agent affects scores and model rankings. We evaluate seven model configurations spanning the full-benchmark score range of 0.06–0.73 on a random sample of 30 SLS-BENCH problems.

Setup. We use budgets of 0, 5, 10, 25, and 50 calls. The agent knows the total budget, receives the remaining budget with each tool output, and receives a `Budget reached` message when the budget is exhausted. Each tool output is capped at 12,000 characters and 1,000 lines to prevent oversized outputs at low budgets.

Scores. Table 6 reports mean scores for each budget. All models score zero without tool calls. Scores generally improve as the budget increases, and the model ordering is preserved across all nonzero budgets.

Table 6: **SLS-BENCH scores by tool-call budget.** Values are means over the 30-problem sample.

Model	Effort	Budget 0	Budget 5	Budget 10	Budget 25	Budget 50
Nemotron 3 Nano	N/A	0.00	0.02	0.04	0.06	0.07
DeepSeek V3.2	N/A	0.00	0.09	0.17	0.20	0.29
GPT-5.4 Nano	medium	0.00	0.19	0.31	0.38	0.40
Haiku 4.5	N/A	0.00	0.32	0.40	0.41	0.43
Opus 4.6	low	0.00	0.44	0.45	0.50	0.50
GPT-5.4	low	0.00	0.56	0.62	0.63	0.65
GPT-5.4	high	0.00	0.58	0.64	0.70	0.71

Budget use. Table 7 reports the mean number of calls used per problem. With 50 calls available, DeepSeek V3.2 and GPT-5.4 Nano (medium) use about 37 calls on average, and Haiku 4.5 and Opus 4.6 (low) use about 16 and 11, respectively, indicating budget under-utilization.

Table 7: **Tool calls used by budget.** Mean number of Python interpreter calls per problem.

Model	Effort	Budget 0	Budget 5	Budget 10	Budget 25	Budget 50
Nemotron 3 Nano	N/A	–	4.23	6.57	12.73	19.53
DeepSeek V3.2	N/A	–	5.00	9.59	20.87	37.10
GPT-5.4 Nano	medium	–	5.00	10.00	24.93	36.83
Haiku 4.5	N/A	–	4.07	7.30	13.70	15.50
Opus 4.6	low	–	4.23	6.93	10.40	10.97
GPT-5.4	low	–	5.00	9.73	20.00	22.37
GPT-5.4	high	–	5.00	9.83	23.43	35.52

Tool-output size. Table 8 reports average output lengths in characters, with multipliers relative to the five-call budget. Per-call output size generally decreases as the budget grows, reaching 0.49–0.91 times the five-call value at a budget of 50. This pattern is consistent with concentrating log inspection into fewer, larger outputs when the budget is scarce.

Table 8: **Tool-output size by budget.** Mean characters per tool output; parenthesized multipliers are relative to the five-call budget.

Model	Effort	Budget 0	Budget 5	Budget 10	Budget 25	Budget 50
Nemotron 3 Nano	N/A	–	957 (1.00×)	1020 (1.07×)	852 (0.89×)	845 (0.88×)
DeepSeek V3.2	N/A	–	1270 (1.00×)	1066 (0.84×)	728 (0.57×)	634 (0.50×)
GPT-5.4 Nano	medium	–	1152 (1.00×)	1096 (0.95×)	1160 (1.01×)	1054 (0.91×)
Haiku 4.5	N/A	–	3743 (1.00×)	3669 (0.98×)	3104 (0.83×)	3257 (0.87×)
Opus 4.6	low	–	3887 (1.00×)	2688 (0.69×)	2333 (0.60×)	2335 (0.60×)
GPT-5.4	low	–	6629 (1.00×)	5577 (0.84×)	4126 (0.62×)	3904 (0.59×)
GPT-5.4	high	–	7301 (1.00×)	5904 (0.81×)	4193 (0.57×)	3570 (0.49×)

I Efficiency Analysis of Agent Trajectories

We reviewed ten GPT-5.4 (medium) trajectories and used a Python tool-output event as the unit of analysis. This unit matches the cost and latency mechanism of the agent: each output corresponds to a completed interpreter call whose result is passed back to the model. Counting the raw trajectories gives 361 such tool outputs in total.

Precomputable artifacts. We mark an output as precomputable when it is a log-derived artifact that can be produced before the agent commits to an incident hypothesis, phase boundary, selected time window, selected trace, or final diagnosis. These artifacts are the same kind of information an engineer usually asks for when first opening an unfamiliar log: schema and scale, timestamp range, service and host inventories, severity distributions, representative messages, global error and template summaries, and coarse temporal views. Under this definition, 75 of 361 (21%) of the observed interpreter outputs are precomputable. The trajectory-level counts are shown in Table 9.

The agent is repeatedly computing generic summaries of the raw log. Instead, they could be pre-computed once per log file and provided as context before the agent begins the open-ended investigation. The agent can then spend its first reasoning steps interpreting the precomputed views rather than discovering the file format and rebuilding common summaries.

Orientation-and-planning prefix. For the parallelism analysis, we define the orientation prefix as the initial contiguous segment of tool outputs used to establish the shared investigation state before the main evidence-collection fan-out. This segment includes generic log orientation artifacts, such as schema, service inventory, message vocabulary, severity mix, templates, and coarse temporal views, together with early anchor-finding calls that identify phase boundaries, first anomalies, or candidate causal components. The prefix ends immediately when the agent begins executing downstream tasks in order to substantiate, quantify, or cross-check the incident explanation.

With this definition, the orientation-and-planning prefix accounts for 108 of 361 outputs (29.9%). The prefix is not identical to the precomputable set: some precomputable summaries appear later, and some early anchor-finding calls are incident-specific rather than generic.

After this prefix, the dominant pattern is fan-out. The agent typically has enough information to separate analyses such as service-level timelines, non-INFO template breakdowns, phase-level statistics, trace or request examples, support-plane event timelines, and recovery checks. These streams use the same log messages as input but do not depend on each other’s intermediate outputs. A parallel implementation could issue these tasks concurrently, then aggregate the results for the final report synthesis.

Trajectory parallelization example. The bottom panel of Fig. 8 uses T9 (0a0114df) as a concrete example of the fan-out pattern. The reconstruction is code-level rather than merely narrative: outputs 1–13 form the shared orientation-and-planning prefix, and each later Python output is assigned to a

Table 9: **Trajectory-level efficiency counts.** Tool outputs are Python interpreter outputs in the raw trajectory. Precomputable outputs are generic log artifacts available before agent reasoning. Prefix outputs are the initial orientation-and-planning calls before the main evidence-collection fan-out.

Trajectory	Problem ID	Tool outputs	Precomputable	Prefix	Prefix fraction
T1	03a9b743	39	7	10	25.6%
T2	1cddedc3	36	10	10	27.8%
T3	0ff91f6a	35	7	9	25.7%
T4	0a7c69ff	42	8	11	26.2%
T5	2d21ffed	32	7	11	34.4%
T6	0b9e6bdf	39	7	11	28.2%
T7	2ee83142	38	5	12	31.6%
T8	1c25d7c3	35	6	13	37.1%
T9	0a0114df	33	9	13	39.4%
T10	2d3c2d7e	32	9	8	25.0%
Total	—	361	75	108	—

Table 10: **Code-level construction of Fig. 8 (bottom) for T9.** Calls are Python tool-output events. The bottom four rows match the stream labels in the figure and sum to the 20 outputs after the 13-call orientation prefix.

Stage / stream	Calls	Code functionality	Concrete role in summary construction
Orientation and planning	13	Load and parse the dataframe; compute schema, level/service/template summaries, coarse per-minute views, and short anomaly windows; build reusable service slices and phase anchors.	Establishes the shared state used by every stream: <code>rtm-3</code> websocket failures and a <code>SIGSEGV</code> crash near 12:20; later session-validation/history failures tied to DB pressure near 12:27; and mitigation/recovery near 12:34.
Failure-chain traces	8	Select anomalous request families—failed websocket upgrades, rejected upgrades, and history warnings—then read all rows with the same <code>trace_id</code> .	Produces request-level chains: failed upgrades target <code>rtm-3</code> ; rejected upgrades pass through <code>session validate timeout</code> with <code>db_err=pool_exhausted</code> ; history warnings line up with app-api timeouts and gateway 503/504 responses.
Service timelines	3	Group service-specific slices by minute or phase using parsed fields such as <code>rtm_target</code> , HTTP status, DB counters, and monitor target.	Shows component evolution: in [12:20, 12:27), <code>rtm-3</code> has 420 failed upgrades and 0 successful upgrades while other RTM targets continue serving; later timelines expose DB/app-api degradation.
Before/after traces	4	Sample representative healthy traces before the incident and recovery traces after mitigation, then read the complete trace rows.	Gives contrastive examples: normal history, websocket, and message-ingest paths before the crash, plus a successful <code>rtm-3</code> websocket handshake after the 12:34 mitigation boundary.
Per-phase statistics	5	Loop over phase windows and count levels, templates, HTTP statuses, websocket outcomes, API events, and DB event types.	Compresses the trajectory into phase evidence: baseline behavior, RTM-targeted trigger, DB/auth/API degradation, and partial recovery.

stream only when it can be run from that shared state without consuming another post-prefix stream’s intermediate result. The shared state consists of the parsed log messages, service-specific views (`edge-gw`, `rtm`, `app-api`, `db`, and `monitor`), extracted fields such as `trace_id`, `rtm_target`, HTTP status, error kind, and DB counters, plus the three incident anchors: the `rtm-3` crash/failure window around 12:20, DB/authentication degradation around 12:27, and the mitigation boundary around 12:34. Table 10 expands the four stream families shown in the figure.

The resulting dependency graph is a fan-out/fan-in computation. Once the orientation state is fixed, target accounting does not need trace reconstruction, trace reconstruction does not need phase rollups, and recovery checks do not need DB timeline summaries. A parallel implementation can therefore issue the four stream families concurrently and aggregate their outputs before final report synthesis, without changing the evidence used by the serialized trajectory.

J Benchmark and Evaluation Validity

J.1 Judge robustness

To ensure our LLM judges grade responses based on substance and meaning, rather than superficial adherence to the ground-truth section structure or wording, we test the judges from Sec. 3.3 on reference summaries modified to preserve or alter their meaning. Each evaluation pairs an original reference with its modified version as the prediction. We measure lexical overlap using word-based Jaccard similarity.

Meaning-preserving changes. We use Opus 4.6 to rephrase all 90 reference summaries. The model is instructed to preserve the meaning, substance, and main report sections required by the benchmark question while changing wording and internal structure as much as possible. We manually verify ten rephrasings for meaning preservation and substantial rewording. The average word-Jaccard similarity between the original summaries and their reworded versions is 0.56, indicating low overlap.

Meaning-altering changes. We also generate LLM-modified summaries that change the meaning and substance of the ground-truth summaries, while preserving their structure and lexical content as much as possible. Their average word-Jaccard similarity is 0.87, indicating high overlap. Similarly, we manually inspect ten modified versions to ensure adequate task execution.

Results. Table 11 reports the scores for both experiments. Meaning-preserving rephrasings (same meaning, different structure) receive an average score of 0.99, and meaning-altering changes (different meaning, same structure) receive 0.17. Both judges retain near-maximal scores under rephrasing and strongly penalize substantive changes. These experiments suggest that our LLM judges grade responses based on their meaning and substance rather than superficial structure or lexical matching.

Table 11: **Judge robustness to reference modifications.** Word-Jaccard is the average lexical similarity between original and modified summaries. Judge scores use GPT-5.2 (medium) and Sonnet 4.6 (low); Average is their mean, as used for SLS-BENCH scoring.

Modification	Word-Jaccard	GPT-5.2	Sonnet 4.6	Average
Meaning-preserving	0.56	0.98	0.99	0.99
Meaning-altering	0.87	0.14	0.20	0.17

J.2 Human evaluation

LLM judges grade predicted summaries by comparing them to the reference summaries only, without involving the log streams, which helps to make evaluation simpler, and reduce its time and cost. Here, we test if this evaluation approach correlates with how human evaluators would grade the predicted summaries against the actual log streams from which they were produced.

We compare LLM judge scores with human assessments of predicted summaries against the logs. We select five models closest to five equally spaced scores across the full-benchmark score range of 0.06–0.73. We randomly sample five SLS-BENCH problems and use the same problems for every model, yielding 25 predicted summaries.

Evaluation procedure. One human evaluator scores the summaries using a condensed, rule-based version of the LLM judge rubric (App. E). The condensed rubric retains the core evaluation criteria and omits fine-grained requirements to reduce evaluation time. The evaluator also receives the incident model as a concise overview of the log stream’s structure to accelerate grading. All assessments are required to be based on the corresponding log streams. Model identities are unknown to the evaluator.

Results and scope. Table 12 reports mean scores over the five shared problems. The combined SLS-BENCH scores and human scores produce identical model rankings. Higher human scores for the strongest models are consistent with the condensed rubric’s greater leniency. The study is limited to one evaluator and five problems, with a condensed rubric and access to the incident model. Use of the condensed rubric by the human evaluator also limits direct comparison of the absolute scores.

However, the results broadly indicate that our evaluation approach aligns with human grading of the predicted summaries based on log streams.

Table 12: **LLM judge and human evaluation scores.** Both columns are means over the same **five** sampled SLS-BENCH problems. The SLS-BENCH score averages the two LLM judges.

Model	SLS-BENCH score	Human score
Nemotron 3 Nano	0.00	0.10
DeepSeek V3.2	0.10	0.20
GPT-5.4 Nano (medium)	0.38	0.35
Opus 4.6 (low)	0.53	0.65
GPT-5.4 (high)	0.63	0.95

J.3 Comparison with external benchmarks

We compare SLS-BENCH model rankings with results on o11y-bench and OpenRCA to assess their agreement across independent observability tasks.

o11y-bench. Grafana Labs’ o11y-bench¹ evaluates agents on observability and site reliability engineering workflows in a live Grafana stack, including querying metrics, logs, and traces and investigating incidents. An LLM judge grades tasks using task-specific rubrics. The Pass³ score is the fraction of tasks solved in all three runs. We use published leaderboard scores for OpenAI and Anthropic models and evaluate the open-weight models ourselves.

Table 13 compares scores for 15 model configurations. The overall Spearman rank correlation is 0.81, with correlations of 0.83 for OpenAI, 1.00 for Anthropic, and 0.90 for open-weight models. A notable ranking difference is GPT-5.4 Nano (high), which scores above both GPT-5.4 Mini configurations on o11y-bench. The Anthropic ordering is preserved exactly.

Table 13: **SLS-BENCH and o11y-bench scores.** A superscript * marks o11y-bench scores from the published leaderboard; unmarked o11y-bench scores are from our runs.

Model	SLS-BENCH score	o11y-bench Pass ³
<i>OpenAI</i>		
GPT-5.4 (high)	0.73	0.63*
GPT-5.4 (low)	0.67	0.56*
GPT-5.4 Mini (high)	0.61	0.29*
GPT-5.4 Mini (low)	0.48	0.21*
GPT-5.4 Nano (high)	0.44	0.48*
GPT-5.4 Nano (low)	0.28	0.19*
<i>Anthropic</i>		
Opus 4.6 (high)	0.61	0.59*
Sonnet 4.6 (low)	0.58	0.51*
Opus 4.6 (low)	0.53	0.49*
Haiku 4.5 (off)	0.46	0.48*
<i>Open-Weight</i>		
GLM-5 (medium)	0.49	0.38
MiniMax-M2 (medium)	0.29	0.29
DeepSeek V3.2 (medium)	0.27	0.32
Ministral 3 (medium)	0.12	0.13
Nemotron 3 Nano (medium)	0.06	0.06

OpenRCA. OpenRCA [Xu et al., 2025] evaluates root-cause localization from real production telemetry, scoring categorical outputs for failure time, component, and reason. We use its Telemetry subset of 51 problems and select five models spread across the SLS-BENCH score range. GPT-5.4, GPT-5.4 Nano, and Sonnet 4.6 use low reasoning effort to limit evaluation time.

¹<https://github.com/grafana/o11y-bench>

Table 14 reports OpenRCA’s Correct and Partial scores. The Partial scores have a Spearman rank correlation of 0.90 with SLS-BENCH. The main ranking difference is Haiku 4.5, which scores above GPT-5.4 Nano (low) on SLS-BENCH and below it on both OpenRCA metrics.

Table 14: **SLS-BENCH and OpenRCA scores.** OpenRCA results cover the 51-problem Telemetry subset; SLS-BENCH scores are from the full benchmark.

Model	SLS-BENCH score	RCA Correct	RCA Partial
GPT-5.4 (low)	0.67	0.14	0.22
Sonnet 4.6 (low)	0.58	0.14	0.21
Haiku 4.5	0.46	0.04	0.08
GPT-5.4 Nano (low)	0.28	0.12	0.20
Nemotron 3 Nano	0.06	0.02	0.07

Because both benchmarks measure different tasks and grading criteria than SLS-BENCH and because the OpenRCA evaluation covers only five models on 51 cases, we treat these as directional rank comparisons rather than absolute-score comparisons.

However, strong rank agreement with two independent real-world benchmarks provides evidence that SLS-BENCH performance is predictive of agent performance on real data and on log analytics tasks that differ from SLS-BENCH itself.

K Extra Figures

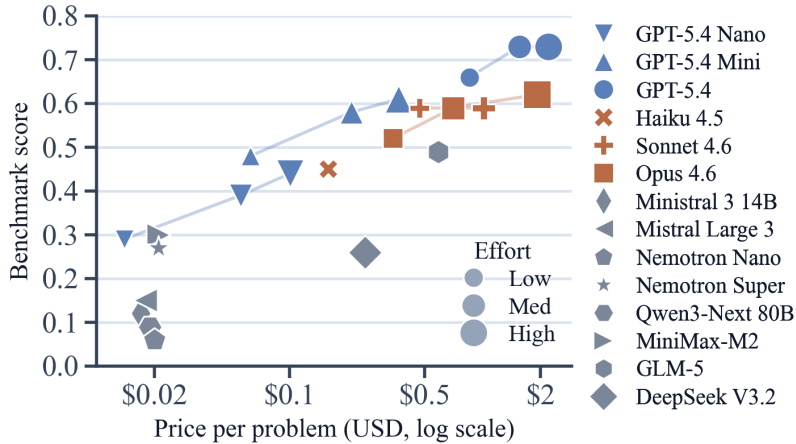


Figure 13: Benchmark score vs. average cost per problem for all evaluated models.

NeurIPS Paper Checklist

1. Claims

Question: Do the main claims made in the abstract and introduction accurately reflect the paper’s contributions and scope?

Answer: [Yes]

Justification: Benchmark and data generation pipeline are presented in Section 3. Evaluations are presented in Section 4.

Guidelines:

- The answer [N/A] means that the abstract and introduction do not include the claims made in the paper.
- The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A [No] or [N/A] answer to this question will not be perceived well by the reviewers.
- The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
- It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. Limitations

Question: Does the paper discuss the limitations of the work performed by the authors?

Answer: [Yes]

Justification: We discuss limitations in Section 3 and Section 5.

Guidelines:

- The answer [N/A] means that the paper has no limitation while the answer [No] means that the paper has limitations, but those are not discussed in the paper.
- The authors are encouraged to create a separate “Limitations” section in their paper.
- The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
- The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
- The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
- The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
- If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
- While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren’t acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. Theory assumptions and proofs

Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

Answer: [N/A]

Justification: The paper does not include theoretical results.

Guidelines:

- The answer [N/A] means that the paper does not include theoretical results.
- All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
- All assumptions should be clearly stated or referenced in the statement of any theorems.
- The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
- Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
- Theorems and Lemmas that the proof relies upon should be properly referenced.

4. Experimental result reproducibility

Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

Answer: [Yes]

Justification: We provide all information, data, and code to reproduce the main results. Our released code contains all necessary information to run the generation and evaluation pipelines. Also, our released data can be directly used with the code to reproduce the main evaluation results.

Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- If the paper includes experiments, a [No] answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
- If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
- Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general, releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
- While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
 - (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
 - (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
 - (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
 - (d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. Open access to data and code

Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

Answer: [Yes]

Justification: For double-blind review purposes, our submission provides links to anonymized data and code repositories containing necessary information to use them. For the public version of the paper we will provide de-anonymized links to our data and code repositories directly in the paper.

Guidelines:

- The answer [N/A] means that paper does not include experiments requiring code.
- Please see the NeurIPS code and data submission guidelines (<https://neurips.cc/public/guides/CodeSubmissionPolicy>) for more details.
- While we encourage the release of code and data, we understand that this might not be possible, so [No] is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
- The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (<https://neurips.cc/public/guides/CodeSubmissionPolicy>) for more details.
- The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
- The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
- At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
- Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. Experimental setting/details

Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer) necessary to understand the results?

Answer: [Yes]

Justification: Our benchmark has only one split that was used to obtain the reported results. The LLM providers and generation parameters are described in the Appendix.

Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
- The full details can be provided either with the code, in appendix, or as supplemental material.

7. Experiment statistical significance

Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

Answer: [Yes]

Justification: Table 1 provides standard error of the mean for the score, and standard deviation for the agent runtime.

Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- The authors should answer [Yes] if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.

- The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
- The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
- The assumptions made should be given (e.g., Normally distributed errors).
- It should be clear whether the error bar is the standard deviation or the standard error of the mean.
- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g., negative error rates).
- If error bars are reported in tables or plots, the authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. Experiments compute resources

Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

Answer: [Yes]

Justification: We provide the dataset construction costs in Table 5 as well as the cost per problem for each evaluated model in Table 1.

Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
- The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
- The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. Code of ethics

Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics <https://neurips.cc/public/EthicsGuidelines>?

Answer: [Yes]

Justification: Our research adheres to the Code of Ethics.

Guidelines:

- The answer [N/A] means that the authors have not reviewed the NeurIPS Code of Ethics.
- If the authors answer [No], they should explain the special circumstances that require a deviation from the Code of Ethics.
- The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. Broader impacts

Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

Answer: [Yes]

Justification: We discuss positive impact of our research in the Introduction. We do not see immediate negative societal impact of our research.

Guidelines:

- The answer [N/A] means that there is no societal impact of the work performed.
- If the authors answer [N/A] or [No], they should explain why their work has no societal impact or why the paper does not address societal impact.
- Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
- The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate Deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
- The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
- If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. Safeguards

Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pre-trained language models, image generators, or scraped datasets)?

Answer: [N/A]

Justification: Our presented work does not have a high risk of misuse.

Guidelines:

- The answer [N/A] means that the paper poses no such risks.
- Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
- Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
- We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. Licenses for existing assets

Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

Answer: [Yes]

Justification: Our benchmark consists of synthetic data grounded in publicly available real-world incident reports; we provide URLs to these reports in our code repository and Croissant metadata.

Guidelines:

- The answer [N/A] means that the paper does not use existing assets.
- The authors should cite the original paper that produced the code package or dataset.
- The authors should state which version of the asset is used and, if possible, include a URL.
- The name of the license (e.g., CC-BY 4.0) should be included for each asset.

- For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
- If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, paperswithcode.com/datasets has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
- For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
- If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

Answer: [\[Yes\]](#)

Justification: We provide all necessary documentation in this paper and the released code and data repositories.

Guidelines:

- The answer [\[N/A\]](#) means that the paper does not release new assets.
- Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
- The paper should discuss whether and how consent was obtained from people whose asset is used.
- At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

Answer: [\[N/A\]](#)

Justification: The paper does not involve crowdsourcing nor research with human subjects.

Guidelines:

- The answer [\[N/A\]](#) means that the paper does not involve crowdsourcing nor research with human subjects.
- Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
- According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

Answer: [\[N/A\]](#)

Justification: The paper does not involve crowdsourcing nor research with human subjects.

Guidelines:

- The answer [\[N/A\]](#) means that the paper does not involve crowdsourcing nor research with human subjects.

- Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
- We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
- For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does *not* impact the core methodology, scientific rigor, or originality of the research, declaration is not required.

Answer: [\[Yes\]](#)

Justification: We use LLMs as integral parts of our synthetic data generation pipeline (generators and verifiers). We also use LLM judges to evaluate the predicted answers against the ground truth answers. We provide details about the usage of the LLMs in [Section 3](#) and [Appendix](#). We also open-source all the code and prompts used for data generation, validation, and answer scoring.

Guidelines:

- The answer [\[N/A\]](#) means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
- Please refer to our LLM policy in the NeurIPS handbook for what should or should not be described.