DSTBench: Distributed Systems Traces-Driven Agentic Performance Debugging in Microservices

S. M. Qasim*, G. Stringhini*, A. K. Coskun*, S. Xi†, D. Altınbüken‡, A. Yazdanbakhsh‡
*Boston University, †Google, ‡Google DeepMind

***Abstract*—Operational performance faults in microservices, such as retry storms, cache stampedes, and gray network failures, arise when application code is correct but the system degrades under load due to interactions between services, infrastructure components, and runtime environments. These faults inflate latencies and amplify request fan-out, often without explicit error signals or with alerts that flag a problem without localizing it. Distributed system traces (DSTs), end-to-end records of request paths through service call graphs, carry the timing distributions, span counts, and call-graph structure needed for localization, but no single signal fully explains a given incident.**

Recent work on large language model (LLM)-based AIOps agents and interactive benchmarks such as AIOpsLab and ITBench has made substantial progress on microservice fault diagnosis. However, existing fault scenarios focus on failures with explicit error signals (connection failures, pod crashes, resource exhaustion), while diagnosing operational performance faults requires comparing observed behavior against a pre-fault baseline, a capability that prior benchmarks do not provide.

We present *DSTBench*, a benchmark and harness for evaluating LLM-based agentic performance debugging in microservices. We make two contributions. First, we introduce DSTFaults, a taxonomy of eight operational performance-fault categories derived from public postmortems and empirical studies of cloud incidents, which we reproducibly inject into four microservice applications from DeathStarBench, Astronomy Shop, and Train Ticket. For each scenario, we collect pre-injection baseline latency and trace statistics and include them in the agent's starting context, enabling baseline-relative diagnosis. Second, we design an improved agent harness with two key properties: it uses *DSToon*, a token-efficient trace serialization format that represents causal directed acyclic graphs (DAGs) in a compact, LLM-readable notation, and it introduces pagination for large telemetry responses, addressing the agent stalls we observe when trace and log volumes grow large under the production-realistic load conditions in our benchmark. For the same AIOpsLab agent, our harness (*DSToon* + pagination + trace statistics) improves detection accuracy from 18% to 80%; for the Flash agent, our harness improves detection accuracy from 62% to 88% (YAML) and achieves localization accuracy up to 27%.

***Index Terms*—microservices, performance debugging, agentic**

I. Introduction

Modern cloud applications are increasingly assembled from hundreds of loosely-coupled microservices that communicate over the network and share infrastructure such as message buses, caches, and databases [1], [2]. A single user request now traverses dozens of service boundaries, and the dominant failure mode in production is no longer the fail-stop crash but the *gray failure*: a subtle, partial degradation, tail-latency inflation, sporadic packet loss, capacity pressure, flaky I/O, or non-fatal exceptions, whose symptoms are observed by some components of the system [3]. Postmortems from large operators consistently report that such performance-class incidents account for the majority of customer-impacting outages and are the most expensive to diagnose, costing millions of dollars per hour [4]–[6].

The intrinsic difficulty of performance debugging in microservices stems from three intertwined properties. First, the *causal distance* between root cause and symptom is large: a misconfigured connection pool in a downstream authentication service can manifest as elevated checkout latency several hops away [1], [7]. Second, faults are *cascading and non-stationary*: retry storms, thundering herds following cache expiry or service recovery, and circuit-breaker oscillations transform a localized perturbation into a system-wide event whose observed signature evolves over time [8]–[10]. Third, performance failures often degrade user experience without crashing services. Aggregate metrics and error rates serve as the first line of detection but are insufficient for localization: a p99 spike or elevated error rate signals that something is wrong but not which service or call-graph path is responsible. DSTs carry the span counts, timing distributions, and call-graph topology needed to distinguish a retry storm from a cache stampede or a circuit breaker trip, yet no single telemetry signal fully explains a given incident [11]–[13].

a) The mainstream approach and its limits.: The dominant approach to debugging DSTs at scale has been *learned anomaly detection over distributed system traces*. Systems such as TraceAnomaly [14], TraceVAE [15], and CRISP [16] model the distribution of normal trace structure and latency, flag deviations across millions of traces, and have been deployed in production at major operators. This line of work is mature: it is cheap enough to run continuously, scales to production trace volumes, and forms the backbone of most automated detection pipelines today. However, these ML-based detectors are semantically constrained. A garbage-collection pause, a routine deploy, and a genuine async message lag all surface as the same anomalous deviation, producing alert volumes that overwhelm on-call engineers and offering no purchase on *the root cause* of the anomaly.

b) The emerging response and its constraints.: A more recent line of work addresses this gap directly by replacing traditional ML detectors with *large language model (LLM)-based agents* that couple chain-of-thought reasoning with tool use over telemetry APIs. RCACopilot [17], RCAgent [18], Nissist [19], FLASH [20], TAMO [21], and the systems evaluated in AIOpsLab [22] produce semantic, human-readable

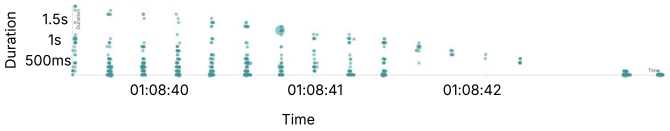


Fig. 1. Latency spikes seen during a cache stampede, when requests fall to the database after a cache miss. Each blue dot represents the latency of a trace.

diagnoses and, in restricted settings, mitigations. But this thread is still developing, and it carries two constraints that prevent it from displacing the mainstream detector. First, agents do not yet meet the accuracy thresholds required for production deployment, and the evidence is consistent across every recent benchmark designed to evaluate them. On OpenRCA [23], the strongest evaluated configuration (Claude 4.6 Opus with a purpose-built RCA agent) reaches only 36.4% end-to-end accuracy [24], a substantial improvement over the 11.3% baseline reported at publication but still far from production usefulness. ITBench [25] reports that state-of-the-art agents resolve only 13.8% of site reliability engineering (SRE) incident scenarios, with diagnosis success on hard scenarios falling to 5.0%, and AIOpsLab itself observes similarly low completion rates on its harder task levels [22]. Across these three independently constructed benchmarks, the pattern is the same: monolithic agents operating on raw telemetry struggle to cross the threshold required for production use, particularly as scenario complexity increases. Second, even if accuracy were adequate, agents are prohibitively expensive to run continuously over a production trace stream, where the mature detector pipeline operates today.

The evidence above establishes that LLM agents fall well short of production readiness, yet current benchmarks may also be systematically underexercising the fault regimes that dominate real production incidents. Two structural gaps prevent an honest assessment of what agents can and cannot do on operational performance faults.

c) The benchmarking gap.: Existing AIOps benchmarks fall short along three axes. (1) *Static datasets*, including the metric and trace corpora used by PyRCA, BARO, and RCAEval, freeze observability data at collection time and therefore preclude evaluation of *interactive* agents that must issue queries, run commands, and react to system state [13], [26], [27]. (2) *Synthetic stress workloads*, typically built from `stress-ng` CPU/memory hogs and `tc` network delays, exercise only a narrow slice of the fault space and bear little resemblance to the gray-failure modes documented in postmortems [3], [28]. (3) Recent interactive harnesses such as AIOpsLab [22] and ITBench [25] provide live microservice deployments with reproducible scenarios, but their fault catalogs are dominated by crash- and configuration-style failures (corrupted images, incorrect authentication, pod failures, memory limits, CPU stress) detectable through standard health checks and metric thresholds. These benchmarks therefore provide limited coverage of *performance pathologies* such as cache stampedes, asynchronous message lag, partial network failures, and GC stalls, where the diagnostic signature lives in trace structure rather than in error logs. More broadly, prior benchmarks largely treat performance degradation as a generic failure symptom rather than studying the specific fault mechanisms that cause it, so published agent results may not fully reflect production operational complexity: an agent excellent at locating a deliberately killed pod may be helpless against a subtle p99 latency spike.

d) The interface gap.: A complementary gap concerns how telemetry reaches the agent. Although the agent reasons over metrics, traces, and logs together, distributed traces dominate the prompt budget: a single incident can involve thousands of spans across dozens of services, and Jaeger output is overwhelmingly serialized as verbose JSON, which inflates prompt size, dilutes salient signals, and consumes a disproportionate share of the model's context window. Recent work on token-efficient prompting has shown that schema-aware, tabular serializations can recover 40–60% of the tokens spent on equivalent JSON payloads while preserving or improving downstream accuracy [29]–[31]. To our knowledge, no prior AIOps agent has systematically exploited this design space, even though distributed system traces, with their regular, columnar span structure, are nearly ideal candidates for such representations.

e) This paper.: We address both gaps with *DSTBench*[1], a benchmark and harness for evaluating LLM-based agents in more realistic gray-failure scenarios, built on the AIOpsLab orchestration substrate [22]. Departing from the coarse fault scenarios of prior AIOps benchmarks, *DSTBench* injects faults into an already-running representative workload against an established baseline performance expectation; these faults may or may not trigger metric alerts or error logs, leaving distributed trace analysis as the key clue for accurate localization. Concretely, our contributions are:

- **DSTFaults: a taxonomy of operational performance faults.** We systematically distill eight categories of operational performance faults (retry storms, latency-induced circuit breaking, asynchronous message lag, cold starts, gray network delay, gray network loss, cache stampedes, and GC stalls) from public postmortems and prior empirical studies of cloud incidents [3], [4], [6], [32], and operationalize each as a reproducible fault injector deployed within four microservice applications from DeathStarBench [1], Astronomy Shop [33], and Train Ticket [2]. For each scenario, we collect pre-injection baseline latency and trace statistics and include them in the agent's starting context, enabling baseline-relative diagnosis that prior benchmarks do not support.
- **An improved agent harness with *DSToon*.** We design a harness with two key improvements. First, it uses *DSToon*, a domain-specific serialization that converts distributed system traces into causal DAGs and then into a compact, LLM-readable notation (TOON [29]), to compress the prompt footprint of DSTs while preserving

[1]The full *DSTBench* benchmark and harness are available at https://github.com/ai4cloudops/DSTBench.

causal and topological structure. Second, it introduces pagination for large telemetry responses, addressing the agent stalls that arise when trace and log volumes grow large under the production-realistic load conditions in our benchmark. Finally, the harness provides a trace-statistics API (`get_traces_stats`) that exposes compact summaries such as latency percentiles, error rates, and span-count statistics.

Together, these contributions move LLM-agent evaluation in AIOps from crash-centric benchmarks toward operational performance debugging in realistic gray-failure scenarios. For the same AIOpsLab agent, our harness (*DSToon* + pagination + trace statistics) improves detection accuracy from 18% to 80%; for the Flash agent, our harness improves detection accuracy from 62% to 88% (YAML) and localization accuracy up to 27% as we detail in Section V.

The remainder of the paper is organized as follows. Section II reviews the AIOpsLab substrate, learned trace anomaly detection, and the LLM-agent literature on which we build. Section III details the *DSTBench* performance-fault taxonomy and its grounding in documented incidents. Section IV describes the *DSTBench* experimental setup, including benchmark applications, agent variants, and evaluation metrics. Section V reports our evaluation against baselines and ablations of *DSToon*, and Section VI concludes.

II. Background

Existing benchmarks evaluate agents on operational and software engineering tasks but leave performance fault diagnosis in live distributed systems largely unaddressed. AIOpsLab [22] provides 100 cloud incident problems across detection, localization, RCA, and mitigation, with a fault library that covers functional faults (misconfiguration, revoked auth, buggy images) and two symptomatic faults (network loss, pod failure), using raw Jaeger JSON as the trace representation. ITBench [25] targets a wider IT automation surface across SRE, compliance (CISO), and cost management (FinOps) personas, with performance-adjacent scenario classes such as `HighCPU` and `CacheFailure` included as part of SRE coverage. However, ITBench's fault injection relies on primitive infrastructure-level mechanisms (CPU and memory hogs via `stress-ng`, network delay via Linux `tc`, pod termination, and memory resource limits), which produce straightforward, easy-to-attribute symptoms rather than the complex distributed system dynamics, such as retry amplification loops or circuit breaker trips, that characterise real production performance incidents. PerfBench [34] takes a fundamentally different approach: it benchmarks agents on fixing performance bugs in static .NET code repositories, providing 81 tasks drawn from real GitHub issues where the agent must modify source code and validate improvements via BenchmarkDotNet microbenchmarks. While PerfBench addresses code-level performance inefficiencies (memory leaks, algorithmic hotspots, concurrency bugs), it operates on offline repositories rather than live distributed systems and does not involve trace-based fault localization.

A common property of faults in these benchmarks is that they produce explicit error signals: misconfigurations cause connection failures, pod terminations trigger crash restarts, CPU and memory hogs exhaust resources until processes are killed, and code-level memory leaks surface as exceptions. These error signals are sufficient to locate the fault — an agent can grep logs for `5xx` responses or `OOMKilled` events and arrive at the root cause without ever reasoning over trace structure. Standard alerting thresholds would fire on these symptoms, and the diagnostic challenge is resolution, not detection.

DSTBench targets a harder and less-covered class of SRE scenarios: faults for which conventional alerting is often insufficient for localization and root-cause analysis, and where trace structure provides the key clues needed to identify the responsible service or interaction. A retry storm inflates span counts and produces sequential timing gaps and timeout spans along a specific call path, even when latency and error-rate alerts have already fired. A latency-induced circuit breaker makes a downstream service vanish from traces entirely with no error tags emitted. A cache stampede produces synchronized latency spikes across concurrent root spans with HTTP 200 responses throughout. In each case, metric alerts and error logs may flag that something is wrong, but identifying which service, call path, or interaction is responsible requires reasoning over latency distributions, span topology, and causal call-graph structure — capabilities that existing benchmarks do not specifically exercise. Table I summarises the key differences.

III. Performance Faults

Operational performance faults are a diagnostically challenging class of failures in microservice systems: application code is correct, but the system degrades under load due to interactions between services, infrastructure, and runtime environments. Unlike hard failures (pod crashes, misconfigurations, or network outages), they manifest as inflated latencies, amplified request fan-out, or silent capacity collapse, and either produce no explicit error signals or trigger metric alerts that flag a problem without localizing it. In both cases, DSTs become the primary diagnostic surface, carrying the timing distributions, span counts, and call-graph topology that aggregate metrics and error logs cannot convey.

We further distinguish operational performance faults from *code-level* performance faults, which originate in flawed application logic (algorithmic inefficiency, memory leaks, concurrency bugs) and are resolved by changing source code. Retry amplification, cache stampedes, GC stalls, and gray network failures, by contrast, emerge from the dynamics of a distributed system in production rather than from a bug in any single service, and their diagnosis and remediation are the responsibility of site reliability engineers reasoning over runtime telemetry. This distinction separates *DSTBench* from code-level benchmarks such as PerfBench [34].

To evaluate how well AI agents handle this class of incident, we introduce *DSTFaults*, an operational performance fault

TABLE I
COMPARISON OF *DSTBench* WITH RELATED PERFORMANCE AND AIOPS BENCHMARKS.

Dimension	**AIOpsLab** [22]	**ITBench** [25]	**PerfBench** [34]	***DSTBench* (ours)**
Primary focus	Cloud incident management (SRE)	Broad IT automation (SRE, CISO, FinOps)	Code-level performance bug fixing	Runtime performance fault diagnosis in microservices
Perf. fault categories	2 (NetworkLoss, PodFailure)	5 (HighCPU, CacheFailure, HTTP faults, MemoryLimit)	5 (Memory, Concurrency, Algorithm, I/O, Build)	**8** (*DSTFaults* taxonomy)
Fault scope	Functional (misconfig, auth, image bugs) + symptomatic (network loss, pod failure)	Infrastructure-primitive: CPU/memory hogs (stress-ng), network delay (tc), pod termination, memory limits	Code-level: memory leaks, concurrency bugs, algorithmic inefficiency, I/O overhead	Application-level: retry amplification, circuit breaking, cache stampedes, network degradation, cascading pool exhaustion
Environment	Live Kubernetes cluster	Live Kubernetes cluster	Static code repos (Docker)	Live Kubernetes cluster
Telemetry	Traces, metrics, logs	Traces, metrics, logs	Microbenchmark output (BenchmarkDotNet)	Traces, metrics, logs
Trace format	Raw Jaeger JSON	NL2Traces	N/A	*DSToon* (compact DAG)
Primary fault signal	Error-based (conn. failures, crashes, 5xx)	Error-based (resource exhaustion, OOM, pod crash)	Performance metrics (execution time, memory)	Trace-structural (latency distribution, span count, topology)
Pre-fault baseline	×	×	×	✓ explicit baseline stats (mean, p50–p99) injected into agent context
Task types	Detect, Localize, RCA, Mitigate	Diagnose, Mitigate, Compliance	Fix code, validate perf. improvement	Detect, Localize, Analyze
Automated eval.	✓ (opt. LLM-as-judge)	✓ (metric-based)	✓ (BenchmarkDotNet)	✓ (incl. LLM-as-judge for analysis)

suite built on top of AIOpsLab, comprising eight faults that exercise different operational failure modes observable through distributed traces and system metrics. We select these faults by surveying production postmortems and the distributed systems reliability literature and applying three inclusion criteria: the fault must be operational (correct code misbehaving due to distributed system dynamics), must produce a distinctive and diagnosable trace signature, and must be documented in at least one real production incident. Table II summarises the taxonomy; the subsections below ground each fault in its documented occurrences and describe its trace signature. Fault-injection mechanisms within DeathStarBench, Astronomy Shop, and Train Ticket are deferred to Section IV.

1) Cache Stampede: A cache stampede occurs when a cache flush or expiry forces many concurrent requests to miss simultaneously and fall through to the backing datastore, producing synchronized latency spikes with no application-level errors. Nishtala et al. [35] identify this as a primary availability challenge in large-scale Memcache deployments; Instagram Engineering [36] document a stampede during cluster warm-up that cascaded latency across unrelated services. The trace signature is a cluster of concurrent root spans with elevated end-to-end latency arriving at the same wall-clock timestamp, with backing-store spans appearing on call paths that normally terminate at the cache layer (Figure 1).

2) Retry Storm: A retry storm arises when clients retry failed or timed-out requests against a degraded service, amplifying load and turning a recoverable perturbation into a cascading overload. Retry storms manifest in multiple ways depending on the triggering mechanism and retry policy: unsynchronized client retries at shared service boundaries [8], retry amplification across misaligned microservice call paths [37], and timeout-driven retry loops where each failed attempt increases load and induces further timeouts. Huang et al. [10] document retry-driven amplification as the most common

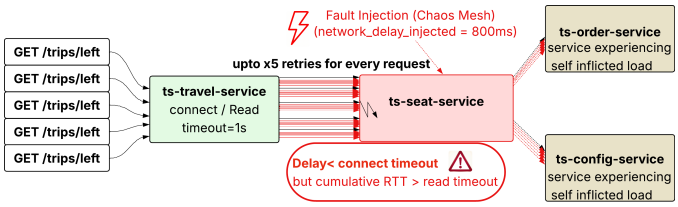


Fig. 2. Retry storm injection on the `ts-travel-service` → `ts-seat-service` call path in Train Ticket: an 800 ms Chaos Mesh delay is within the connection timeout but exceeds the read timeout, causing each failed call to retry, with each retry generating further downstream requests.

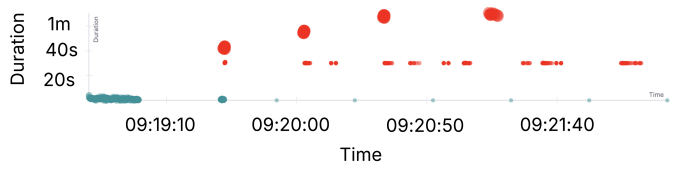


Fig. 3. Trace latencies (blue dots) in the Jaeger UI during a retry storm. The synchronized bands of elevated latency correspond to repeated retry attempts on the `ts-seat-service` call path. The red dots are traces with error spans from a read timeout exception; the size of each dot scales with the number of spans in that trace, relative to other traces.

TABLE II
DSTFaults: EIGHT PERFORMANCE FAULT CATEGORIES AND THEIR DISTRIBUTED TRACE SIGNATURES.

Category	Fault	DST Signature
Resiliency	Retry Storm	Repeated identical child subtrees per request; elevated span count with sequential timing gaps between retries.
Resiliency	Latency-Induced Circuit Breaker	Preceded by a window of high-latency spans to one service; sudden drop in span count as the circuit trips, with traces truncating early, no downstream spans, and no error tags.
Middleware	Asynchronous Message Lag	Large timestamp gap between Producer Finish and Consumer Start span; total trace duration inflates with no single slow span; no error status on either end.
Environment	Cold Starts	First requests routed to newly provisioned Java Virtual Machine (JVM) pods show substantially elevated end-to-end latency; subsequent requests on the warmed pod return to baseline; no error tags, no unusual child spans.
Network	Gray Network Delay	Uniform latency inflation on every span crossing the affected service edge; span count unchanged; no error tags on either endpoint; per-service health checks remain green.
Network	Gray Network Loss	Timeout spans and error tags on calls crossing the affected service edge; span count drops as failed requests truncate before reaching downstream dependencies.
Infrastructure	Cache Stampede	Burst of simultaneous cache-miss spans arriving at the same instant, followed by a fan-out of identical backend queries to the backing store on paths that would normally be served from cache.
Environment	GC Stall	Periodic stop-the-world pauses stall all concurrent requests to the affected JVM pod simultaneously; p99 latency spikes while p50 remains near baseline; no error tags, no downstream propagation; stall intervals correlated across concurrent root spans.

sustaining effect across 22 real production incidents from 11 organizations, appearing in over 50% of cases. The trace signature is a staircase of repeated identical child subtrees per root request, with elevated span counts, timeout spans on the affected call path, and sequential timing gaps matching the client retry interval, as shown in Figures 2 and 3.

3) Latency-Induced Circuit Breaker: A latency-induced circuit breaker trip occurs when sustained high-latency responses from a downstream service cause a load balancer to begin rejecting calls immediately, protecting the caller from thread exhaustion but silently removing the downstream service from the call graph. Dean and Barroso [38] show that high-latency episodes cascade destructively across service dependencies at scale; Sedghpour et al. [39] and Lu et al. [40] show that determining when to trip in latency-driven environments remains unsolved, since static timeout thresholds cannot adapt to gradual latency accumulation. The trace signature is counter-intuitive: root-span duration grows while span count drops, downstream spans disappear entirely, and no error tags are emitted anywhere.

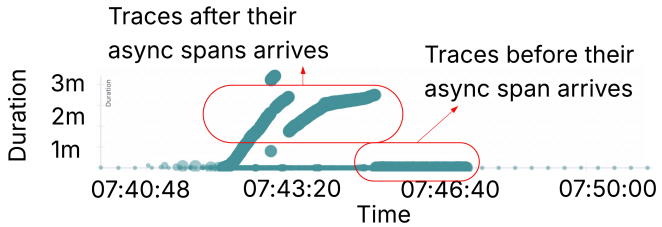


Fig. 4. Trace latencies (blue dots) in the Jaeger UI during an async message lag fault. Trace duration is low before async spans are emitted from the lagging consumer, *'ts-notification-service'* in our benchmark.

4) Asynchronous Message Lag: Asynchronous message lag occurs in queue-based or stream-based microservice systems when the consumer falls behind the producer's arrival rate, inflating end-to-end processing latency while leaving both endpoints individually healthy. Yadav [41] documents this pattern in Slack's job queue, where slow consumers cause user-visible action delays to accumulate while no service crashes or returns errors. A March 2026 GitHub incident exhibited the same dynamic at the webhook delivery pipeline: resource constraints in the consumer pushed average webhook delivery latency from roughly 5 to 160 seconds, disrupting downstream CI/CD workflows even though the surface endpoints continued accepting requests [42]. Huang et al. [10] characterize consumer-side queue lag as a sustaining mechanism in metastable failures from which the system fails to recover without intervention. The trace signature is a large timestamp gap between the producer finish span and the consumer start span, with total trace duration inflating while no single span shows elevated duration and no error tags appear on either end (Figure 4).

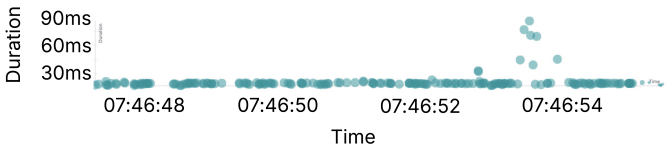


Fig. 5. Cold start latency spike during a rolling pod scale-out: newly provisioned Spring Boot pods serve requests before JVM warm-up completes, producing substantially elevated end-to-end latency before self-resolving as the runtime reaches steady state.

5) Cold Starts: Cold starts are an environmental performance fault in which a newly provisioned pod incurs elevated latency on its first requests while its runtime completes ini-

tialization; in JVM-based frameworks such as Spring Boot, class loading, JIT compilation, and dependency injection add substantial overhead before steady-state throughput is reached. Wang et al. [43] report cold start latencies ranging from hundreds of milliseconds to tens of seconds across serverless runtimes; Shahrad et al. [44] show that cold starts recur continuously in production as scale-to-zero keep-alive windows expire. The trace signature is a subset of concurrent traces to the same service with a root-span latency outlier while traces routed to warmed replicas remain at baseline, with pod-level instance metadata in the span identifying the slow replica (Figure 5).

6) Gray Network Failure: Gray network failure is a network-layer fault in which the path between two services degrades without either endpoint being internally at fault: packets are delayed or dropped in transit, leaving per-service metrics clean while end-to-end request latency inflates or requests fail to complete. Huang et al. [3] identify partial network degradation as one of the most under-detected incident classes in production clouds. We implement two variants: the delay variant uniformly slows every packet crossing the affected edge, producing inflated latency on every traversing span with span count and error status unchanged; the loss variant drops a fraction of packets, producing timeout spans and error tags with span count dropping as failed requests truncate.

7) GC Stall: A GC stall is a JVM runtime environmental fault in which forced or excessive garbage collection triggers stop-the-world pauses that freeze all application threads simultaneously. Dean and Barroso [38] identify stop-the-world GC as a primary source of tail latency at scale, noting that pause durations ranging from tens to hundreds of milliseconds create synchronized latency spikes across all in-flight requests with no application-level error or downstream signal. The trace signature is a cluster of concurrent root spans with elevated end-to-end latency sharing the same wall-clock pause window, with p99 spiking sharply while p50 remains near baseline and no child spans or error tags affected.

IV. Experimental Setup

A. Infrastructure and Testbed

We conduct all experiments on a Kubernetes cluster provisioned using `kind` (Kubernetes-in-Docker), managed via the AIOpsLab orchestration framework [22]. OpenEBS provides persistent storage, deployed as the default storage class at the start of each experiment and torn down upon completion. Alongside each application we co-deploy Prometheus to collect pod-level metrics (CPU, memory, network I/O, Istio latency and throughput), Jaeger to collect distributed traces, and Elasticsearch to collect logs. We tear down and reprovision the full observability stack per experiment to avoid cross-run interference.

B. Benchmark Applications

To ensure variety across architectures and fault surfaces, we evaluate our agents on four microservice benchmark applications, each selected to surface specific classes of performance faults. **SocialNetwork** [1] comprises Thrift-based microservices with caching layers, making it ideal for cache-related anomalies. **Train Ticket** [2] features 41 Spring Boot services communicating via synchronous HTTP and message queues, making its architecture well-suited for asynchronous message lag and retry storms. **HotelReservation** [1] contains $\sim$15 microservices routed through a centralized `user` service, facilitating the study of network-layer degradation propagation. Finally, **OpenTelemetry Astronomy Shop** [33] is a 15-service polyglot application with built-in load generation and programmatic fault injection, serving as a testbed for JVM runtime faults such as GC stalls.

C. Models and Agent Variants

We evaluate five agent variants, all backed by locally hosted models served via Ollama [45]. The backbone models are **Gemma 4 31B** [46] and **glm-4.7-flash** (30B parameters), provisioned on a dedicated pod in the New England Research Cloud backed by an **NVIDIA H100 80 GB HBM3** GPU with 80 GB of system memory. We restrict evaluation to open-weight backbones to ensure reproducibility and on-premises deployment by operators handling sensitive operational telemetry. The Ollama instance is tunneled to the AIOpsLab controller at inference time so that all agent calls are served locally. Each agent is allowed up to 100 steps per problem instance.

AIOpsLab Baseline (`ollama_aiopslab`). A standard reactive agent with access to the full AIOpsLab telemetry surface: `get_logs`, `get_metrics`, `read_metrics`, `get_traces`, `read_traces`, `exec_shell`, and `submit`. The agent serves as the ablation baseline against which all other variants are measured.

DSToon Agent (`ollama_dstoon`). Extends the baseline by replacing the baseline trace APIs with the *DSToon*-aware telemetry APIs (detailed in Section IV-E). The baseline trace methods are hidden from the agent. This reduces token consumption while preserving the structural call-graph information needed to reason about span fan-out (as in retry storms) and span absence (as in latency-induced circuit breaking).

Flash Agent Baseline [20] (`flash_ollama`). Serves as an additional baseline that augments the AIOpsLab agent with a hindsight-aware reasoning loop. Before each main LLM call, a secondary `OllamaHindsightBuilder` sub-agent reviews the last five history messages and the current observation, then advises whether to submit an answer or continue diagnosing. This advisory text is prepended to the current observation before it is passed to the main reasoning model. The Flash architecture targets premature diagnostic steps and early stopping: performance faults often produce a conclusive signal early in the trace, but agents with no self-monitoring mechanism continue issuing API calls long after the evidence is sufficient. We observed that Flash agents frequently failed to follow the required result-reporting format, leading to poor automated evaluation even when the diagnosis was potentially correct. We therefore add a reminder of the required format to the advisory text for all Flash-based agents.

Flash+DSToon Agent (`flash_ollama_dstoon`). Combines both enhancements: hindsight-augmented reasoning with the *DSToon* trace representation. Context management is handled jointly: per-observation token truncation applies a cap of `MODEL_CTX // 10`, and full conversation history is trimmed to a configurable token budget before each LLM call.

Flash+YAML Agent (`flash_ollama_yaml`). Uses the same harness as `flash_ollama_dstoon`, except that the trace representation format is *YAML* instead of *DSToon*. This variant isolates the effect of the trace serialization format from the hindsight-augmented reasoning loop, allowing us to compare the diagnostic accuracy of *YAML* (which achieves the highest single-trace localization accuracy in our format ablation in Section V-A) against *DSToon* (which trades a small accuracy margin for substantially lower token consumption) under an otherwise identical agent configuration.

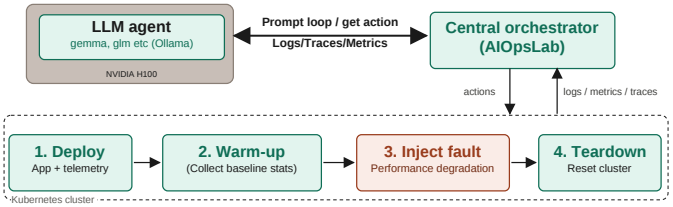


Fig. 6. Experiment lifecycle managed by the AIOpsLab `Orchestrator`, covering application deployment, fault injection, workload generation, the agent loop, evaluation, and cleanup.

D. Orchestrator Flow

Each experiment follows a fixed lifecycle managed by the AIOpsLab `Orchestrator`, as illustrated in Fig. 6:

1) **Deploy.** We install OpenEBS and Prometheus, deploy the application via Helm, and verify it with a post-deployment health check.
2) **Workload Start.** A synthetic workload is launched via `wrk2` as a Kubernetes Job to generate representative traffic for the deployed application. Asynchronous workloads are scheduled as background coroutines so subsequent stages begin without blocking.
3) **Warm-up and Baseline Collection.** The application is allowed to run under the workload until pod-level metrics and trace latency distributions stabilize, establishing a baseline performance reference against which the post-fault behavior is compared.
4) **Fault Injection.** The fault is injected (e.g., pod scale-up, Redis flush, feature flag toggle, network-delay, resource throttling).
5) **Agent Loop.** The agent receives the task description, API reference, and an initial prompt. At each step, the agent returns a single API call in a Markdown code block. The orchestrator parses the call using AST-based parsing, dispatches it to the environment, and returns the resulting observation. This repeats for up to `max_steps` steps or until a valid `submit()` is received.
6) **Evaluation.** The submitted solution is evaluated against the ground truth. Task-specific metrics are described in Section IV-F.
7) **Cleanup (Teardown).** The fault is recovered, the application namespace is deleted, and the Prometheus/OpenEBS stack is uninstalled to reset the cluster for the next experiment.

The full agent–environment interaction trace is saved as a JSON session file for analysis. Each problem in *DSTBench* carries its own workload profile, fault-injection parameters, and recovery routine; the granular per-problem configurations are documented in the benchmark's Github repository.

E. Available Telemetry APIs

During the agent loop, the agent may invoke the following observability APIs. These constitute the baseline AIOps harness:

- `get_logs(service, start, end)` — retrieves container logs from Elasticsearch.
- `get_metrics(service, start, end)` — queries Prometheus for CPU, memory, and network I/O per pod/container.
- `get_traces(service, start, end)` — fetches distributed traces from Jaeger.
- `exec_shell(cmd)` — executes a non-interactive shell command (e.g., `kubectl get pods`).

The improved *DSTBench* harness replaces the baseline trace APIs with two *DSToon*-aware endpoints that the *DSToon* agent variants use in place of `get_traces` and `read_traces`, adds `get_traces_stats` as a new trace-summary endpoint, and substitutes `read_metrics_paged` for `read_metrics`:

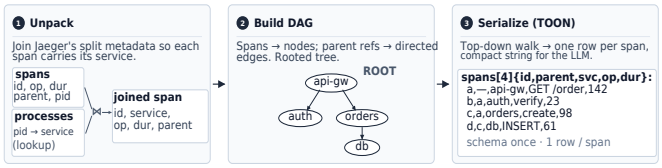


Fig. 7. *DSToon* conversion pipeline: raw Jaeger JSON traces are parsed and serialized as topologically sorted DAGs with columns for span ID, parent ID, service, operation, and duration.

- `get_traces_dstoon(namespace, duration=5)` — fetches Jaeger traces and serializes them in *DSToon* format: a topologically sorted directed acyclic graph (DAG) per trace with columns `id`, `parent`, `service`, `operation`, and `duration` (in microseconds) serialized using TOON [29]. The result is saved to a `.dstoon` file and the path is returned to the agent.
- `read_traces_dstoon(file_path, page=0)` — reads a `txt` file with 2,000-line pagination. Each block begins with a `trace_id:` header followed by a span column header and CSV rows. When the response is truncated, the agent is prompted to call `page=N+1` to retrieve the next block.

- `get_traces_stats(namespace, duration)` — returns a statistical summary of Jaeger traces, including latency percentiles, error rates, and span-count statistics, without saving to a file. This provides a compact, high-level view of trace behavior before the agent commits to reading raw span data.
- `read_metrics_paged(file_path, page=0)` — reads a Prometheus CSV metrics file with 2,000-line pagination, replacing the non-paginated `read_metrics` to prevent context window exhaustion on large metric exports.

F. Task Types and Evaluation Metrics

Each fault scenario is instantiated as three independent task types: Detection, Localization, and Analysis. We do not evaluate mitigation because a single problem admits multiple valid remediations (e.g., throttling requests or pre-provisioning for high load); we will address this in future work. In total, *DSTBench* comprises 25 problem instances: 12 Detection (8 fault-injected, one per fault category, and 4 no-fault controls, one per deployed application, used to measure false positives), 8 Localization, and 5 Analysis; the full problem list is available in our repository.

Task accuracy is the primary correctness metric, defined per task type:

- **Detection** is scored as binary correct or incorrect over fault-injected and no-fault runs of the app.
- **Localization** uses exact-match accuracy (100%) or partial credit when the ground-truth service name is contained within the agent's answer set, computed as accuracy $= \frac{1}{|\text{answer set}|} \times 100\%$. This penalizes agents that hedge by returning large service sets rather than committing to a single root cause.
- **Analysis** is scored by an LLM-as-judge (`LLMJudge`), using Gemini 2.5 Flash, that evaluates the agent's free-form diagnostic narrative against a reference ground-truth analysis. `LLMJudge` classifies each response as Correct, Partial, or Incorrect.

All tasks additionally record the number of agent–environment interaction steps and input/output token counts. Time metrics (TTD, TTL, TTA) measure wall-clock seconds from problem initialization to `submit()`.

V. Evaluation and Results

A. Trace Serialization Format Selection

To establish the effects of DST representation on LLMs, we design a controlled experiment to measure how well different serialization formats convey trace-structural information to LLMs (exp1). We randomly inject an artificial latency delay into one span of a SocialNetwork trace, then present each model with three healthy traces and one anomalous test trace serialized in a candidate format, and ask the model to localize the anomalous span. This setup isolates format comprehension from agent strategy: the only variable is how we represent the DST. Specifically, we inject random delays ranging from 625 microseconds to 20ms into random spans, whereas the average latency of healthy social media traces is 10ms. We create 180 test traces for this task and evaluate five formats across six model backbones, with results shown in Figure 8.

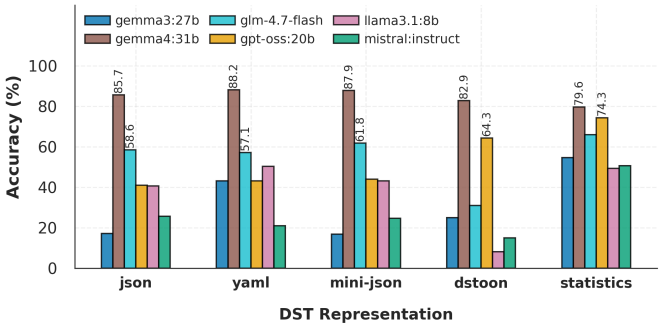


Fig. 8. Span-localization accuracy (isolated single-trace task) under five trace serialization formats across six model backbones. *YAML* achieves the highest accuracy among structural formats for the gemma4:31b model; *DSToon* preserves the trace structure and achieves similar accuracy with up to 10× less context overhead.

First, we observe that *YAML* achieves the highest span-localization accuracy among structural formats (e.g., gemma4:31b at 88.2%). However, *DSToon* achieves similarly competitive accuracy (e.g., gemma4:31b at 82.9%) while preserving the trace structure and requiring up to 10× less context overhead. Second, providing per-span latency statistics rather than raw span records (the `statistics` condition) proves effective for conveying overall latency anomaly information across all models (e.g., gemma4:31b: 79.6%; gpt-oss:20b: 74.3%), as aggregated percentiles make the deviation from normal immediately legible without requiring the model to compare raw durations across many individual spans.

To quantify and analyze why different representations affect model behavior, we compute five lexical metrics across the trace formats using *tiktoken*, normalized in Figure 9: *Symbol Density* (ratio of punctuation and symbols to total characters), *Punctuation-to-Token Ratio*, *Structural Entropy* (Shannon entropy over structural delimiters), *File Size*, and overall *Token Count*. The analysis reveals that JSON yields the highest token count, YAML and miniJson perform similarly, and *DSToon*'s compact DAG serialization dramatically minimizes token count and file size while maintaining high symbol density. This demonstrates that *DSToon* preserves the essential causal and topological structure needed for models to reason over call-graphs, without the prohibitive context overhead of other formats. Because *DSTBench* requires localization and analysis, tasks that demand reasoning over which service or call path is responsible, we select both *YAML* and *DSToon* to further evaluate our benchmark: *YAML* offers peak per-trace span-localization accuracy on the isolated single-trace task above, while *DSToon* achieves comparable diagnostic accuracy with up to 10× less context, which can be advantageous when the agent must reason over multiple anomalous and baseline traces within a fixed context budget.

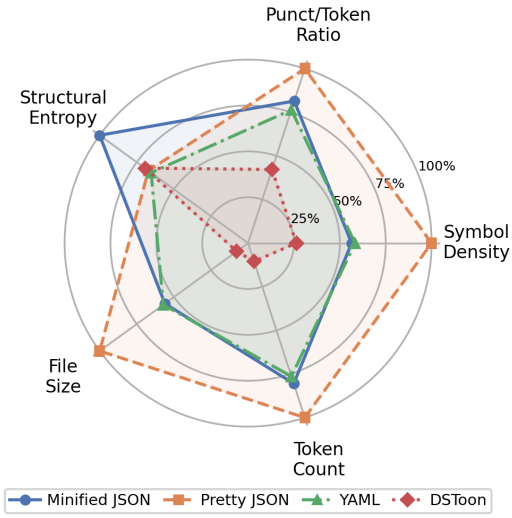


Fig. 9. Comparison of DSToon with other representation formats across normalized metrics. DSToon significantly reduces token count, file size, and structural entropy while maintaining high symbol density, making it extremely context-efficient compared to YAML and JSON variants.

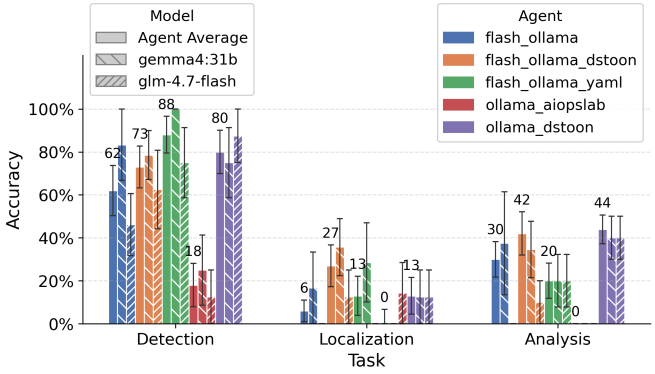


Fig. 10. Accuracy breakdown across Detection, Localization, and Analysis tasks for various agent configurations. Results are grouped by task, illustrating the performance of different model backbones and trace representation methods. Agents using our harness (which employs YAML or DSToon representations alongside pagination) demonstrate substantially improved performance compared to those relying on the default harness in AIOpsLab. Error bars denote ±1 standard error of the mean.

B. Accuracy of Agents on DSTBench

Figure 10 reports task accuracy across Detection, Localization, and Analysis for various agent configurations, averaged over 3 independent runs over all problems of *DSTBench* for each task; the full experimental setup is described in Section IV. We evaluate agents utilizing our custom harness (`ollama_dstoon`, `flash_ollama_dstoon`, and `flash_ollama_yaml`), which leverage *DSToon* or *YAML* representations combined with pagination and `get_traces_stats`, against baseline agents (`ollama_aiopslab` and `flash_ollama`), which rely on the default AIOpsLab harness. We observe that agents utilizing our custom harness consistently outperform the baseline agents. Specifically, the `ollama_aiopslab` baseline agent performs the worst overall, and log analysis reveals that it gets stuck in high-volume telemetry when reading logs and traces. In contrast, the `ollama_dstoon` agent navigates large telemetry volumes efficiently by employing pagination alongside the compact *DSToon* format. This structural advantage allows it to avoid getting stuck and perform on par with or better than the baseline `flash_ollama` agent, which utilizes a sophisticated hindsight mechanism.

For the Detection task, agents using our harness achieve significantly higher accuracy compared to the baseline, demonstrating that tools in our harness more effectively convey the presence of an anomaly. For the same AIOpsLab agent, detection accuracy rises from 18% with the default harness (`ollama_aiopslab`) to 80% with our harness (*DSToon* + pagination + `get_traces_stats`; `ollama_dstoon`); for the Flash agent, detection accuracy rises from 62% (`flash_ollama`) to 88% with *YAML* (`flash_ollama_yaml`) and 73% with *DSToon* (`flash_ollama_dstoon`). The performance gap is also present in the Localization task, which requires the agent to pinpoint the exact failing service. Here, agents using our harness improve upon the baseline's accuracy: the AIOpsLab agent jumps from 0% to 13%, and the Flash agent from 6% to 27% under *DSToon*. The `flash_ollama_yaml` and `flash_ollama_dstoon` configurations form our controlled comparison for serialization format: they share the same agent, model, and harness, and differ only in how DSTs are represented. We observe that *YAML* is stronger on Detection (88% vs. 73%), while *DSToon* is stronger on Localization (27% vs. 13%). This suggests compact structural representations are beneficial for tasks requiring precise causal reasoning over high-volume traces. However, even with our improved harness, localization remains challenging, suggesting current LLM agents are still far from production-grade RCA for operational performance faults.

Finally, in the Analysis task, agents with our harness again demonstrate superior average performance (agent-average bar combines both models), with our harness raising the AIOpsLab agent from 0% (almost always gets stuck reading telemetry) to 44%; Flash agent from 30% to 42% under *DSToon*. Analysis scores exceed Localization because `LLMJudge` awards partial credit: flagging a performance problem earns half points even without identifying the root cause.

We also observe that the `glm-4.7-flash` model generally underperforms `gemma4:31b` across tasks. However, it benefits from our harness, allowing it to achieve competitive accuracy.

C. Limitations

We observe that localizing operational faults remains challenging for current LLM agents, and accuracy varies substantially with the choice of backbone model, so our results are best read as trends rather than fixed capabilities. This uncertainty compounds on the Analysis task, where scoring relies on an LLM-as-judge (`LLMJudge`, using Gemini 2.5 Flash) whose rubric is grounded in structured criteria (root cause, failure mechanism, affected components, fault type);

we manually spot-checked a sample of judge verdicts and found them accurate, but have not conducted a systematic human-agreement study, so judge scores should still be read as a proxy for analysis quality rather than a fully validated metric. We see substantial improvements in model capabilities over time, illustrated by gemma4:31b's consistent gains over its predecessor gemma3:27b across every trace format in the isolated format-comprehension task of Figure 8. We read our results as a snapshot of where current models stand rather than a ceiling on the approach, and *DSTFaults* is designed to track that progress: a fixed taxonomy and baseline telemetry against which future, more capable models and agent harnesses can be benchmarked.

Two further limitations concern the benchmark's current implementation. Exposing shell access to an agent, while useful in a benchmark setting, is risky in production environments and would require strong sandboxing and action-level guardrails before such agents could be deployed against live systems. Separately, the *DSToon* schema used in our experiments records each span's duration but not its start offset, so inter-span timing gaps (e.g. retry intervals, producer-to-consumer lag) cannot be read directly off the DAG; we have since added a `start` column to *DSToon* in the open-sourced version to address this.

VI. Conclusion

In this paper, we presented *DSTBench*, a benchmark that addressed a gap left by prior AIOps benchmarks with coarser crash- and configuration-oriented fault scenarios. *DSTBench* provided more realistic production fault scenarios where a representative workload was already running against the application with an established baseline performance expectation, and a fault was then injected that may or may not trigger metric alerts or error logs, leaving analysis of distributed traces as the key clue for accurate diagnosis. Furthermore, we systematically studied how different trace serialization formats impacted an agent's reasoning capabilities. We introduced *DSToon*, an efficient representation that dramatically reduced token overhead while preserving essential topological and causal structures. Our evaluations demonstrated that equipping agents with structural trace representations such as *DSToon* or *YAML*, alongside paginated telemetry and trace statistics, substantially improved diagnostic accuracy over the default AIOpsLab harness. Ultimately, this work advanced AIOps agent evaluation from crash-centric incident benchmarks toward realistic operational performance debugging, where distributed traces carry the key diagnostic clues.

Acknowledgment

We thank Fatih Acun, Efe Sencan, Amin Khodaverdian, Beste Oztop, Sina Karimi, and Reza Sajjadinasab for helpful discussions on this work. We also thank the extended team at Google and Google DeepMind for their valuable contributions to this research. We thank the New England Research Cloud (NERC) and the Chameleon testbed [47] for providing the shared computing resources used in this work. This work has been partially funded by a Google Academic Research Award (2025–2026).

References

[1] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson *et al.*, "An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems," in *Proceedings of the twenty-fourth international conference on architectural support for programming languages and operating systems*, 2019, pp. 3–18.

[2] X. Zhou, X. Peng, T. Xie, J. Sun, C. Ji, W. Li, and D. Ding, "Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study," *IEEE Transactions on Software Engineering*, vol. 47, no. 2, pp. 243–260, 2018.

[3] P. Huang, C. Guo, L. Zhou, J. R. Lorch, Y. Dang, M. Chintalapati, and R. Yao, "Gray failure: The achilles' heel of cloud-scale systems," in *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, 2017, pp. 150–155.

[4] V. Ganatra, A. Parayil, S. Ghosh, Y. Kang, M. Ma, C. Bansal, S. Nath, and J. Mace, "Detection is better than cure: A cloud incidents perspective," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023, pp. 1891–1902.

[5] J. Chen, X. He, Q. Lin, Y. Xu, H. Zhang, D. Hao, F. Gao, Z. Xu, Y. Dang, and D. Zhang, "An empirical investigation of incident triage for online service systems," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 2019, pp. 111–120.

[6] H. Yan, Y. Chen, M. Ma, M. Wen, S. Lu, S. Zhang, T. Xu, R. Wang, C. Bansal, S. Rajmohan *et al.*, "An empirical study of production incidents in generative ai cloud services," in *2025 IEEE 36th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2025, pp. 359–370.

[7] Z. Liu, H. Xu, B. Hu, D. Pei, Y. Su, X. Wang, Y. Liu, Z. Wang, M. Cao, C. Wang, Y. Zhang, L. Yang, B. Pei, Y. Sun, Y. Hu, and Y. Liu, "Practical root cause localization for microservice systems via trace analysis," in *Proceedings of the 29th IEEE/ACM International Symposium on Quality of Service (IWQoS)*, 2021.

[8] B. Beyer, C. Jones, J. Petoff, and N. R. Murphy, *Site Reliability Engineering: How Google Runs Production Systems*. O'Reilly Media, 2016, chapter 22: Addressing Cascading Failures.

[9] S. Floyd and V. Jacobson, "The synchronization of periodic routing messages," *IEEE/ACM Transactions on Networking*, vol. 2, no. 2, pp. 122–136, Apr. 1994.

[10] L. Huang, M. Magnusson, A. Bangalore Muralikrishna, S. Estyak, R. Isaacs, A. Aghayev, T. Zhu, and A. Charapko, "Metastable failures in the wild," in *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI '22)*. USENIX Association, 2022, pp. 73–90.

[11] G. Yu, P. Chen, Y. Li, H. Chen, X. Li, and Z. Zheng, "Nezha: Interpretable fine-grained root causes analysis for microservices on multi-modal observability data," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 2023, pp. 553–565.

[12] C. Lee, T. Yang, Z. Chen, Y. Su, and M. R. Lyu, "Eadro: An end-to-end troubleshooting framework for microservices on multi-source data," in *Proceedings of the 45th International Conference on Software Engineering (ICSE)*. IEEE/ACM, 2023, pp. 1750–1762.

[13] L. Pham, H. Ha, and H. Zhang, "RCAEval: A benchmark for root cause analysis of microservice systems with telemetry data," in *Proceedings of the VLDB Endowment*, 2025, arXiv:2412.17015.

[14] P. Liu, H. Xu, Q. Ouyang, R. Jiao, Z. Chen, S. Zhang, J. Yang, L. Mo, J. Zeng, W. Xue, and D. Pei, "Unsupervised detection of microservice trace anomalies through service-level deep bayesian networks," in *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*, 2020, pp. 48–58.

[15] Z. Xie, H. Xu, W. Chen, W. Li, H. Jiang, L. Su, H. Wang, and D. Pei, "Unsupervised anomaly detection on microservice traces through graph vae," in *Proceedings of the ACM Web Conference 2023*, 2023, pp. 2874–2884.

[16] Z. Zhang, M. K. Ramanathan, P. Raj, A. Parwal, T. Sherwood, and M. Chabbi, "CRISP: Critical path analysis of large-scale microservice architectures," in *Proceedings of the 2022 USENIX Annual Technical Conference (USENIX ATC 22)*, Carlsbad, CA, 2022, pp. 655–672. [Online]. Available: https://www.usenix.org/conference/atc22/presentation/zhang-zhizhou

[17] Y. Chen, H. Xie, M. Ma, Y. Kang, X. Gao, L. Shi, Y. Cao, X. Gao, H. Fan, M. Wen, J. Zeng, S. Ghosh, X. Zhang, C. Zhang, Q. Lin, S. Rajmohan, D. Zhang, and T. Xu, "Automatic root cause analysis via large language models for cloud incidents," in *Proceedings of the 19th European Conference on Computer Systems (EuroSys)*. ACM, 2024, pp. 674–688.

[18] Z. Wang, Z. Liu, Y. Zhang, A. Zhong, J. Wang, F. Yin, L. Fan, L. Wu, and Q. Wen, "RCAgent: Cloud root cause analysis by autonomous agents with tool-augmented large language models," in *Proceedings of the 33rd ACM International Conference on Information and Knowledge Management (CIKM)*. ACM, 2024.

[19] K. An, F. Yang, J. Lu, L. Li, Z. Ren, H. Huang, L. Wang, P. Zhao, Y. Kang, H. Ding *et al.*, "Nissist: An incident mitigation copilot based on troubleshooting guides," *arXiv preprint arXiv:2402.17531*, 2024.

[20] X. Zhang, T. Mittal, C. Bansal, R. Wang, M. Ma, Z. Ren, H. Huang, and S. Rajmohan, "Flash: A workflow automation agent for diagnosing recurring incidents," October 2024. [Online]. Available: https://www.microsoft.com/en-us/research/publication/flash-a-workflow-automation-agent-for-diagnosing-recurring-incidents/

[21] X. Zhang, Q. Wang, M. Li, Y. Yuan, M. Xiao, F. Zhuang, and D. Yu, "Tamo:fine-grained root cause analysis via tool-assisted llm agent with multi-modality observation data in cloud-native systems," *IEEE Transactions on Services Computing*, vol. 18, no. 6, pp. 4221–4233, 2025.

[22] Y. Chen, M. Shetty, G. Somashekar, M. Ma, Y. Simmhan, J. Mace, C. Bansal, R. Wang, and S. Rajmohan, "Aiopslab: A holistic framework to evaluate ai agents for enabling autonomous clouds," *Proceedings of Machine Learning and Systems*, vol. 7, 2025.

[23] J. Xu, Q. Zhang, Z. Zhong, S. He, C. Zhang, Q. Lin, D. Pei, P. He, D. Zhang, and Q. Zhang, "Openrca: Can large language models locate the root cause of software failures?" in *The Thirteenth International Conference on Learning Representations*, 2025.

[24] Microsoft Research, "OpenRCA Leaderboard," https://microsoft.github.io/OpenRCA/, 2026, accessed: 2026-04-26.

[25] S. Jha, R. Arora, Y. Watanabe, T. Yanagawa, Y. Chen, J. Clark, B. Bhavya, M. Verma, H. Kumar, H. Kitahara, N. Zheutlin, S. Takano, D. Pathak, F. George, X. Wu, B. O. Turkkan, G. Vanloo, M. Nidd, T. Dai, O. Chatterjee, P. Gupta, S. Samanta, P. Aggarwal, R. Lee, P. Murali, J. wook Ahn, D. Kar, A. Rahane, C. Fonseca, A. Paradkar, Y. Deng, P. Moogi, P. Mohapatra, N. Abe, C. Narayanaswami, T. Xu, L. R. Varshney, R. Mahindru, A. Sailer, L. Shwartz, D. Sow, N. C. M. Fuller, and R. Puri, "Itbench: Evaluating ai agents across diverse real-world it automation tasks," 2025. [Online]. Available: https://arxiv.org/abs/2502.05352

[26] C. Liu, W. Yang, H. Mishra, A. Saha, D. Sahoo, and S. C. H. Hoi, "PyRCA: A library for metric-based root cause analysis," *arXiv preprint arXiv:2306.11417*, 2023, salesforce Research technical report; https://github.com/salesforce/PyRCA.

[27] L. Pham, H. Ha, and H. Zhang, "BARO: Robust root cause analysis for microservices via multivariate Bayesian online change point detection," in *Proceedings of the ACM on Software Engineering (FSE)*, vol. 1, no. FSE, 2024, pp. 2214–2237.

[28] Y. Yang, Y. Wu, K. Pattabiraman, L. Wang, and Y. Li, "How far have we come in detecting anomalies in distributed systems? an empirical study with a statement-level fault injection method," in *Proceedings of the 31st IEEE International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2020, pp. 59–69.

[29] J. S. Johansson and TOON Contributors, "TOON: Token-oriented object notation – compact, human-readable, schema-aware JSON for LLM prompts," https://github.com/toon-format/toon, 2024, open-source specification and benchmarks.

[30] D. C. Schmidt *et al.*, "Prompt engineering for structured data: Cross-model comparison of prompt styles for token-efficient generation," *Preprints.org*, 2024, preprint, College of William & Mary.

[31] B. Williams, "Token efficiency with structured output from language models," Microsoft Data Science and AI Engineering Blog, 2024, https://medium.com/data-science-at-microsoft/token-efficiency-with-structured-output-from-language-models-be2e51d3d9d5.

[32] D. Cotroneo, L. De Simone, P. Liguori, R. Natella, and N. Bidokhti, "How bad can a bug get? an empirical analysis of software failures in the openstack cloud computing platform," in *Proceedings of the 2019 27th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*, 2019, pp. 200–211.

[33] OpenTelemetry Authors, "OpenTelemetry astronomy shop demo," https://github.com/open-telemetry/opentelemetry-demo, 2024, reference microservices application with built-in observability and feature-flag-driven fault injection.

[34] S. Garg, R. Z. Moghaddam, and N. Sundaresan, "Perfbench: Can agents resolve real-world performance bugs?" 2025. [Online]. Available: https://arxiv.org/abs/2509.24091

[35] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani, "Scaling memcache at Facebook," in *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2013, pp. 385–398.

[36] Instagram Engineering, "Thundering herds & promises," https://instagram-engineering.com/thundering-herds-promises-82191c8af57d, 2019.

[37] J. Tavori, A. Bremler-Barr, H. Levy, and O. Lavi, "RetryGuard: Preventing self-inflicted retry storms in cloud microservices applications," 2025.

[38] J. Dean and L. A. Barroso, "The tail at scale," *Communications of the ACM*, vol. 56, no. 2, pp. 74–80, 2013.

[39] M. R. S. Sedghpour, C. Klein, and J. Tordsson, "Service mesh circuit breaker: From panic button to performance management tool," in *Proceedings of the 1st Workshop on High Availability and Observability of Cloud Systems (HAOC '21)*. ACM, 2021.

[40] R. Lu, Y. Lu, Y. Jiang, G. Xue, and P. Huang, "One-size-fits-none: Understanding and enhancing slow-fault tolerance in modern distributed systems," in *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI '25)*, 2025.

[41] S. Yadav, "Scaling Slack's job queue," Slack Engineering Blog, 2017, https://slack.engineering/scaling-slacks-job-queue/. Accessed: 2026-05-14.

[42] DevActivity, "GitHub webhook incident: Lessons in performance engineering software," DevActivity Insights, Mar. 2026, https://devactivity.com/insights/github-webhook-incident-lessons-in-performance-engineering-software/. Accessed: 2026-05-14.

[43] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift, "Peeking behind the curtains of serverless platforms," in *Proceedings of the 2018 USENIX Annual Technical Conference (USENIX ATC '18)*. USENIX Association, 2018, pp. 133–146.

[44] M. Shahrad, R. Fonseca, I. Goiri, G. Chaudhry, P. Batum, J. Cooke, E. Laureano, C. Tresness, M. Russinovich, and R. Bianchini, "Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider," in *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC '20)*. USENIX Association, 2020, pp. 205–218.

[45] Ollama, "Ollama: Get up and running with large language models locally," https://ollama.com, 2024.

[46] Google DeepMind, "Gemma 4," https://ai.google.dev/gemma/docs/core/model_card_4, 2025.

[47] K. Keahey, J. Anderson, Z. Zhen, P. Riteau, P. Ruth, D. Stanzione, M. Cevik, J. Colleran, H. S. Gunawi, C. Hammock, J. Mambretti, A. Barnes, F. Halbach, A. Rocha, and J. Stubbs, "Lessons learned from the chameleon testbed," in *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC '20)*. USENIX Association, Jul. 2020.