SRE-MARATHON: A CONTINUOUS, CHANGE-DRIVEN BENCHMARK FOR AUTONOMOUS SITE RELIABILITY AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Benchmarks for site reliability engineering (SRE) agents are typically episodic: one fault is injected, the agent receives an incident task, and its response is scored. Production operation is not. Incidents surface through noisy alerts, overlap in time, and often originate from code or configuration changes. We present SRE-Marathon, a benchmark for long-horizon, continuous SRE operation. An agent is invoked at a fixed cadence with cumulative alert history and a persistent workspace while operating a live two-zone Kubernetes deployment as a fault orchestrator injects overlapping faults according to a seeded, production-calibrated schedule. Curated code and configuration changes deployed through the same build pipeline available for repair. Each run is recorded into a sealed bundle and scored offline: Marathon-Score credits each injected fault for ordered progress through correlation, localization, and repair, with all metrics computed deterministically from recorded system evidence. Across three applications and about sixty faults per run, the best of ten methods reaches only 41.3 out of 100. Agents often correlate and localize faults, but almost never complete repairs while the faults remain active.

1 INTRODUCTION

Operating a live online service requires continuously keeping it available and reliable. When problems arise, an on-call engineer is notified through a stream of alerts, many of which may be noisy or redundant, and must determine whether an incident is occurring, which symptoms belong together, what changed, how to remedy it (roll back, reconfigure, or fix forward), and whether the service has recovered, all while failures degrade user experience and incur operational cost (Zhang et al., 2025). Language-model agents are increasingly proposed for this role, and a growing line of benchmarks evaluates their ability to diagnose and mitigate failures (Chen et al., 2025b; Jha et al., 2025; Clark et al., 2026). Yet an important question remains unanswered: can these agents sustain production operations over time?

Production operations are not merely a sequence of isolated incident-resolution tasks. They involve continuously operating live services while problems are detected, disentangled, prioritized, and resolved. This setting is characterized by three essential features. *First*, problems surface through noisy operational signals rather than explicit incident descriptions: an engineer observes an evolving alert stream and must determine whether an incident exists and which symptoms belong together. *Second*, many incidents originate from changes within the system itself. A study of a large cloud service found that code and configuration bugs together account for 39.5% of high-severity incidents (Ghosh et al., 2022), making diagnosis and repair of internal system changes central to production operations. *Third*, incidents overlap and interventions have consequences. Multiple incidents may be active at once, while repairs consume time and resources and may disrupt otherwise unaffected parts of the system. Evaluation must therefore consider not only diagnostic accuracy, but also whether problems are resolved promptly, safely, and efficiently.

Existing benchmarks (Jha et al., 2025; Clark et al., 2026; Gong et al., 2026; Wang et al., 2026) have largely overlooked these characteristics. They typically follow an episodic protocol: start from a clean environment, inject a single fault, assign the agent an explicit incident-resolution task, and

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



Figure 1: One SRE-Marathon run using the ReAct (Yao et al., 2023) agent with Kimi K2.5 on Astronomy Shop. Above: every fault from occurrence to revert. Below: the agent's actions. The wall-clock axis runs longer than a nominal 90-minute run because the schedule drifts (Appendix A).

terminate once it submits an answer. This setup provides the agent with an explicit incident context rather than evaluating whether it can detect, correlate, and triage noisy operational signals. Fault injection is typically based on chaos primitives, excluding failures caused by internal code or configuration changes and therefore removing rollback and fix-forward as repair options. Episodes typically contain one incident at a time and are evaluated independently, leaving overlap, accumulated mistakes, collateral effects, and operational cost largely unmeasured. On SREGym, frontier agents already achieve up to 72.6% diagnosis success and 78.5% mitigation success (Clark et al., 2026). Such strong results are obtained under a substantially simplified model of production operations.

We introduce SRE-Marathon, a benchmark for sustained production operations rather than isolated incident episodes. The agent is invoked periodically to inspect the accumulated alert history and operate a live distributed microservice deployment under realistic workload. Unlike benchmarks that rely primarily on externally induced chaos faults, SRE-Marathon also introduces failures through realistic code and configuration changes that pass basic validation before deployment, requiring the agent to reason about rollback, reconfiguration, and fix-forward. Faults are injected continuously according to seeded, production-calibrated schedules whose intensity increases over time, creating overlapping incidents and sustained operational pressure. Evaluation spans the full run and the full incident lifecycle, including detection and correlation, root-cause localization, repair, and recovery validation. Figure 1 illustrates these characteristics in one run.

To ensure reproducibility despite the live environment, SRE-Marathon records each run into a sealed bundle and computes all metrics deterministically offline from system evidence. Across ten methods and two model backbones, agents correlate many faults and some methods locate a substantial fraction of their root causes, but successful repair remains rare under sustained operation. The best Marathon-Score reaches only 41.3 out of 100, revealing a substantial gap between current agents and end-to-end autonomous production operation. Our contributions are:

- **A continuous benchmark for production operations** (§2), with overlapping faults, evolving alert streams, live microservice deployments, and realistic code and configuration changes.
- **An offline evaluation protocol and metric suite** (§3) covering correlation, localization, repair, intervention risk, service degradation, latency, and cost, all computed deterministically from recorded system evidence.
- **A broad empirical evaluation and failure analysis** (§4–§5.3) across ten methods spanning runbooks, SRE agents, and coding agents on two model backbones.

2 The SRE-Marathon Design

SRE-Marathon is designed to evaluate SRE agents under sustained production operation. To do so, it places an agent in charge of a live microservice application while faults arrive continuously according to a seeded schedule. Failures may overlap, propagate across services, and compete for the agent's attention. Rather than receiving an incident-specific task, the agent is invoked periodically and must infer problems from operational signals, decide which incidents require attention, and determine when and how to intervene (Figure 2).

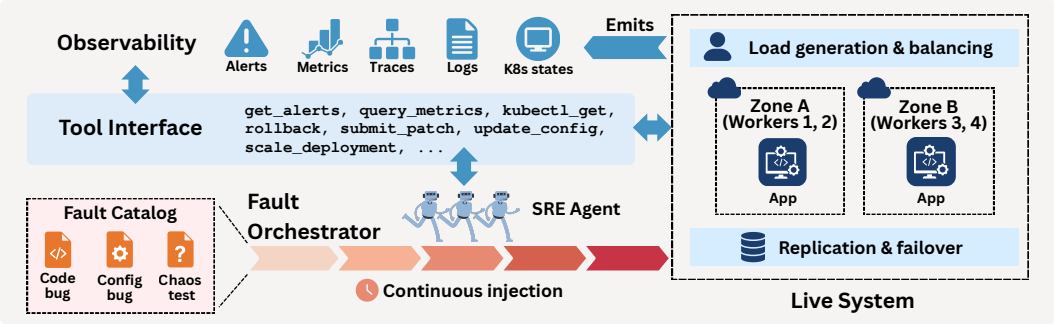


Figure 2: Overview of SRE-Marathon. A fault orchestrator continuously introduces problems into a live microservice system under load. The SRE agent observes their effects through telemetry and alerts and interacts with the system only through instrumented tools.

Overview. A benchmark run consists of three components. First, a *live system* hosts a microservice application under continuous background traffic, so failures and repairs have observable effects on user-facing service behavior. Second, a *fault orchestrator* generates a reproducible sequence of code, configuration, deployment, dependency, infrastructure, and other failures from a seeded schedule. Each injected problem has hidden ground truth, including its responsible service and artifact, that is retained by the benchmark but never exposed to the agent. Third, an *agent–system interface* exposes alerts, telemetry, system state, and controlled operational actions. The agent is invoked periodically through this interface, with a persistent workspace across invocations, and every interaction is recorded.

Live system. SRE-Marathon places the agent in a running microservice environment rather than replaying a fixed telemetry trace. This is important because an intervention must be evaluated by what actually happens after it is executed: whether the affected service recovers, whether user-facing degradation decreases, and whether the action introduces collateral effects elsewhere in the system.

We use three microservice applications spanning different architectures, topologies, and failure surfaces: the 20-service OpenTelemetry Demo Astronomy Shop (OpenTelemetry, 2023), the 11-service Online Boutique (Google Cloud Platform, 2019), and the 41-service Train Ticket (Zhou et al., 2021). Each run deploys one application under continuously generated workload. For Astronomy Shop, we additionally emulate a geo-distributed deployment using two Kubernetes zones with replicated instances and failover, allowing individual components to fail without taking down the entire application and making cross-zone interventions visible in user-facing availability.

Fault Orchestrator. Long-horizon evaluation requires faults to arrive continuously while remaining reproducible across runs. We therefore develop a seeded *Fault Orchestrator* that determines which problems occur, where they occur, when they begin, and when reverted faults are removed. Given the same benchmark release and random seed, the orchestrator is guaranteed to produce the same sequence of injected faults.

Fault catalog. The orchestrator draws from a catalog of 700 faults across our three applications (Table 1). Each catalog entry records its failure mechanism, target service, affected scope, symptom probe, and responsible artifact (Figure 3). A fault is injected at a single target service, while any downstream symptoms arise through the application's own dependency structure. This preserves an unambiguous injected root cause while allowing its effects to propagate across services and overlap with those of other faults.

Fault injection mechanisms. We support two mechanisms for introducing failures. First, *change-driven faults* are delivered as realistic code or configuration changes. For code faults, a curated patch introduces a defect as a side effect of an otherwise plausible modification; the orchestrator deploys the change through the normal release pipeline. Each patch passes the service's lint and unit tests, modeling a defect that escapes continuous integration. Configuration faults are likewise introduced through changes to the running system.

Second, *direct faults* introduce failures through controlled interventions to the deployment, such as resource exhaustion, network disruption, or other chaos-style perturbations. Every catalog entry is admitted only after injection reproduces the intended failure and the system successfully recovers afterward (Appendix C).

Marathon schedule. The orchestrator samples fault categories according to the production distribution reported by Ghosh et al. (2022): code bug 27.0%, dependency 16.4%, infrastructure 15.8%, deployment 13.2%, configuration bug 12.5%, database or network 10.5%, and authentication 4.6%, renormalized over the categories supported by each application. It then selects a compatible fault from the catalog.

In our main schedule, fault arrivals follow a seeded process whose intensity increases fourfold over the run, while individual faults remain active for four to twelve minutes before being reverted. This increasing concurrency tests whether an agent can continue detecting, disentangling, and resolving incidents as operational load accumulates. The agent receives no notification when a fault begins and can only detect it through its observable effects.

Table 1: Fault catalog across the three benchmark applications (details in Appendix B).

Category	Mechanism	Astro	OB	TT	Total
Code bug	change or direct	160	32	32	224
Config bug	change or direct	35	75	93	203
Chaos test	direct	122	66	85	273
Total		317	173	210	700

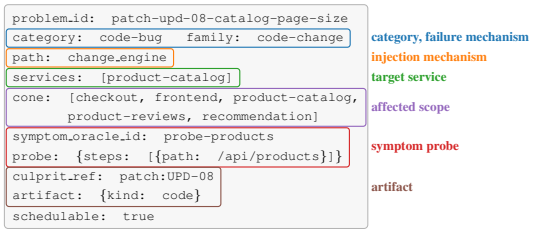


Figure 3: Example fault specification. The highlighted fields define its category, injection mechanism, target service and affected scope, symptom probe, and responsible artifact.

Agent–system interface. The agent interacts with the live system through 19 instrumented tools exposed through the Model Context Protocol (Anthropic, 2024), together with a periodic invocation protocol. The tools provide access to operational telemetry and system state, support incident reporting, and execute controlled interventions. All tool calls are recorded for offline evaluation (Appendix E).

Observability tools. All three benchmark applications export the same classes of operational signals through OpenTelemetry Collectors (OpenTelemetry, 2026): metrics are stored in Prometheus (Prometheus, 2026), traces in Jaeger (Jaeger, 2026), and logs in Loki (Grafana Labs, 2026). We design a set of 44 alerting rules that continuously evaluates these signals and emits alerts when their conditions are met (Appendix D). The resulting timestamped alerts form the alert history accessible to the agent via the `get_alerts` tool. Alerts are not grouped into incidents automatically, leaving the agent responsible for determining which alerts correspond to the same underlying problem. We also make sure that telemetry emitted from the fault orchestrator and other benchmark infrastructure is excluded, so the agent observes only signals from the application and its deployment.

System interaction tools. In SRE-Marathon, read-only tools expose alerts, metrics, logs, traces, service topology, Kubernetes state, and deployed source code. The `incident_report` tool lets the agent record incident grouping, root cause diagnosis, and repair status. Operational tools support actions including restart, scaling, rollback, configuration changes, feature-flag updates, and source-level repair. Most of those interventions are exposed through a corresponding tool; source repair follows a three-step workflow: `propose_patch` applies a source diff, `build_service_image` builds the modified service, and `submit_patch` publishes and deploys it.

Because these tools can modify a live system, mutating operations pass through a policy enforcement layer that checks their targets, confines actions to benchmark resources, enforces per-invocation limits, and records forbidden operations. This provides a controlled action boundary while preserving the effects of the agent's interventions on the running system.

Periodic invocation. We invoke the agent at a fixed interval Δ throughout the run. At each interval, the agent receives the cumulative alert history together with its persistent private workspace. A new invocation begins only after the previous one has finished, and each invocation runs in a fresh process while the workspace persists across intervals. Compared to invoking agents on every alert,

this design grants every agent a fixed number of opportunities to reason and act over a SRE-Marathon run, regardless of the alert volume.

Each invocation is given a wall-clock budget shorter than Δ; for the long runs in our evaluation, $\Delta = 120\,\text{s}$ and the invocation budget is 110 s. If the budget is exceeded, the invocation terminates while the live system and fault schedule continue. Section 4 reports the remaining run parameters.

3 EVALUATION

Evaluating a long-running SRE agent requires more than checking its final diagnosis. We organize incident handling into three stages: (a) *correlation*, which associates alerts with incidents; (b) *localization*, which identifies the responsible service and artifact; and (c) *repair*, which verifies that the underlying fault has been corrected. We report a metric for each stage and combine them in the Marathon-Score to measure end-to-end progress through the incident lifecycle. Because all metrics are derived from recorded system evidence rather than a model judge, they may also serve as verifiable objectives for training SRE agents (Stojanovski et al., 2025; Pan et al., 2025).

Each run records the injected faults, alert history, telemetry, agent reports, tool calls, executed interventions, and user-facing service measurements, allowing all metrics to be recomputed offline. We write $\mathcal{F}$ for the set of injected faults and $\mathcal{P}_f$ for the affected scope of fault f, i.e., the services that may be affected through the application's dependency graph.

The Marathon-Score. The Marathon-Score summarizes how successfully the agent progresses each injected fault through correlation, localization, and repair, while accounting for severe safety violations. For each fault $f \in \mathcal{F}$, let a_f, b_f, and c_f indicate successful correlation, localization, and repair, respectively. Let $d = 1$ if the run contains an attempted forbidden operation and $d = 0$ otherwise:

$$\text{Marathon-Score} = 100 \cdot \left(1 - \frac{d}{2}\right) \cdot \frac{1}{|\mathcal{F}|} \sum_{f \in \mathcal{F}} \frac{a_f + a_f b_f + a_f b_f c_f}{3}. \tag{1}$$

The three terms enforce the lifecycle order: localization contributes only after successful correlation, and repair only after both correlation and localization. A fault that passes all three stages is *handled* and receives full credit. We report both the aggregate Marathon-Score and the individual stage metrics. The denominator includes every injected fault, including faults whose symptoms disappear before the next measurement, preventing rapid repairs from reducing the number of faults being evaluated. Appendix F provides a worked example.

(a) **Correlated.** Correlation measures whether the agent correctly associates a fault with an incident, with $\text{Corr} = \frac{1}{|\mathcal{F}|} \sum_f a_f$. For each fault, we identify alerts consistent with its active time window, namespace, and affected scope $\mathcal{P}_f$, then match faults one-to-one with incidents reported by the agent. We set $a_f = 1$ when the matched incident contains at least one alert attributable to f. Because overlapping faults may affect the same services, an alert can be compatible with more than one active fault; this occurs for 39%, 76%, and 47% of alerts in the three applications.

(b) **Located.** Localization measures whether the agent correctly identifies both where a fault occurred and what artifact caused it, with $\text{Loc} = \frac{1}{|\mathcal{F}|} \sum_f b_f$. We set $b_f = 1$ when the diagnosis associated with f exactly identifies its culprit service or services and its responsible artifact.

(c) **Repaired.** Repair measures whether an intervention by the agent resolves the fault while it is still active, with $\text{Rep} = \frac{1}{|\mathcal{F}|} \sum_f c_f$. We set $c_f = 1$ when the agent executes an intervention targeting a service in $\mathcal{P}_f$ and subsequent system evidence confirms recovery. Recovery is established when the fault's symptom check returns to and remains in a healthy state, or, when no symptom failure was observed, when the responsible artifact is verified to have been corrected. Only interventions completed before the orchestrator reverts the fault are eligible, so benchmark cleanup is never credited as an agent repair.

(d) **Intervention risk.** Intervention risk measures the fraction of actions that either extend beyond the affected scope of the referenced incident or violate an explicit safety policy: $\text{Risk} = (n_{\text{out}} + n_{\text{forb}})/(n_{\text{exec}} + n_{\text{forb}})$. Here, n_{exec} is the number of executed interventions, n_{out} counts actions

outside the affected scope, and n_{forb} counts attempted forbidden operations such as deleting the benchmark namespace or accessing protected secrets.

Operational outcomes. Beyond incident-handling effectiveness, we report service impact, response latency, and computational cost.

(e) **Service degradation.** Service degradation measures the cumulative amount of user-facing traffic affected by errors or excessive latency. At time t, let $e(t)$ be the failed-request fraction, $\ell(t)$ the fraction of requests slower than 2.5 s, and $q(t)$ the request rate at the application's entry-point service. We define service degradation $D = \int \min\{1, e(t) + \ell(t)\}\, q(t)\, dt$, where the first term estimates the fraction of degraded requests and $q(t)$ weights it by traffic volume. We report the degradation level, $\text{Degrad. Level} = D_{\text{agent}}/D_{\text{no-action}}$, as a percentage, against a no-action run with the same application configuration and fault-schedule seed.

(f) **Time to detect, locate, and repair.** For each fault, MTTD is the elapsed time from injection to the first incident report containing one of its alerts, MTTL to the first correct localization, and MTTR to the first verified repair. We report the median for each metric over faults that successfully reach the corresponding stage. Because these metrics are computed over different subsets of faults, their values need not be ordered; for example, MTTL can be lower than MTTD. All timing measurements have resolution Δ, the agent invocation interval.

(g) **Cost.** We report the total model calls, token consumption, and estimated monetary cost of each run. Tokens are the primary model-independent usage measure, while dollar cost reflects the pricing used in our experiments.

4 Experimental setup

Schedules and deployment environments. Our main evaluation follows the *marathon* schedule defined in §2. Each run injects around sixty faults over a 60-minute period of increasing intensity, followed by a 30-minute observation period with no new injections. The agent is invoked every $\Delta = 120\,\text{s}$. Across Online Boutique, Train Ticket, and Astronomy Shop, this schedule results in an average of 5.8, 6.1, and 6.5 simultaneously active faults, respectively, with peak concurrency of 10, 11, and 11. The schedule is intentionally designed to stress sustained operation under increasing fault concurrency rather than to reproduce the incident rate of a typical production hour.

For Marathon-Score evaluation, all methods share the same seeded schedule for each of the three applications. The main experiments run on a single-host Kind cluster (Kubernetes SIGs, 2026), which provides a reproducible environment for controlled comparison. We additionally validate the benchmark on a seven-node kubeadm deployment (Appendix A).

Methods. We compare ten methods. Five are baselines implemented by us (Appendix G): a non-LLM rule-based *Runbook*; a tool-augmented *ReAct* agent (Yao et al., 2023); *RAG* (Lewis et al., 2020), which retrieves live telemetry and produces one diagnosis-and-repair plan per invocation; ORBIT, which maintains persistent incident state across invocations; and *Keep-agent*, which uses Keep's (Keep, 2026) rule engine for alert grouping and one LLM call per invocation for diagnosis and repair planning. We additionally evaluate the SRE agent *OpenSRE* (Tracer Cloud, 2026), the coding agents *Codex* (OpenAI, 2026a) and *Claude Code* (Anthropic, 2026), both unmodified (Appendix G), and the diagnosis methods *OpsMem* (Sun et al., 2026) and *COCA* (Li et al., 2025), which execute interventions through our minimal repair shell. All methods share the same alerts, tools, and policy enforcement layer. We evaluate the LLM-based methods with Kimi K2.5 (Moonshot AI, 2026) and GPT-5.6 Luna (OpenAI, 2026b), under a budget of less than $8 per run.

5 Results

5.1 Overall benchmark performance

End-to-end resolution remains rare, with repair as the primary bottleneck. Keep-agent achieves the highest Marathon-Score in Table 2 at 41.3, followed by Runbook at 33.5; the remaining Kimi K2.5 methods score between 3.7 and 25.6, while GPT-5.6 Luna methods range from 8.2 to

Table 2: SRE-Marathon leaderboard: each row averages a method's marathon runs on three microservice applications; the best value per column is in bold.

Method	Backbone	Stage metrics			Median time (s)			Degrad. Level (%)	Cost per run		Risk	Marathon-Score (Out of 100)
		Corr	Loc	Rep	MTTD	MTTL	MTTR		Tokens	$		
Runbook	—	0.658	0.364	**0.024**	167	267	560	111.0	—	—	0.028	33.5
ReAct	kimi-k2.5	0.694	0.079	0.006	189	283	709	86.9	6.5M	4.02	0.217	25.4
(Yao et al., 2023)	gpt-5.6-luna	0.419	0.125	0.000	163	293	—	100.8	6.3M	1.45	0.349	16.3
RAG	kimi-k2.5	0.426	0.398	0.000	200	68	—	150.5	569k	0.46	0.015	23.5
(Lewis et al., 2020)	gpt-5.6-luna	0.199	0.105	0.000	159	**65**	—	93.5	730k	0.23	**0.000**	8.2
ORBIT	kimi-k2.5	0.768	0.000	0.000	175	—	—	92.8	5.4M	3.35	—	25.6
	gpt-5.6-luna	0.768	0.009	0.000	186	249	—	98.6	7.1M	1.63	—	25.9
OpenSRE	kimi-k2.5	0.093	0.200	0.000	297	188	—	99.7	7.1M	4.40	**0.000**	3.7
(Tracer Cloud, 2026)	gpt-5.6-luna	0.222	0.034	0.000	**143**	173	—	92.5	7.0M	1.59	0.158	8.3
Keep-agent	kimi-k2.5	**0.775**	**0.491**	0.017	188	165	476	92.1	685k	0.72	0.014	**41.3**
(Keep, 2026)	gpt-5.6-luna	0.656	0.313	0.017	184	142	455	84.5	742k	0.31	**0.000**	31.3
Codex	kimi-k2.5	0.231	0.039	0.011	210	335	456	131.7	7.2M	4.45	0.171	8.1
(OpenAI, 2026a)	gpt-5.6-luna	0.502	0.138	0.023	203	346	**411**	85.6	8.0M	1.81	0.104	20.4
Claude Code	kimi-k2.5	0.379	0.050	0.000	337	210	—	103.1	6.7M	4.14	**0.000**	13.4
(Anthropic, 2026)	gpt-5.6-luna	0.407	0.070	0.000	360	684	—	80.7	3.4M	0.84	**0.000**	14.3
OpsMem	kimi-k2.5	0.587	0.006	0.000	146	351	—	124.3	754k	0.55	0.200	19.8
(Sun et al., 2026)	gpt-5.6-luna	0.613	0.034	0.000	157	450	—	**77.4**	984k	0.25	**0.000**	21.0
COCA	kimi-k2.5	0.576	0.017	0.023	168	336	786	132.3	511k	0.36	0.051	19.6
(Li et al., 2025)	gpt-5.6-luna	0.634	0.253	0.011	161	222	461	87.6	**403k**	**0.09**	0.103	28.0

31.3. Despite these nonzero scores, complete resolution is extremely uncommon: only Keep-agent handles any fault end to end (1/57 on Online Boutique with Kimi K2.5 and 2/59 with GPT-5.6 Luna). Most score therefore comes from correlation and localization rather than repair; for example, ORBIT obtains 25.6 from correlation alone. Localization is strongest for Keep-agent (0.491), RAG (0.398), and Runbook (0.364), whereas no coding or diagnosis method exceeds these baselines. Across all methods, repair remains at most 0.024 while faults are active. The strongest methods therefore rely heavily on explicit rule-based structure, while more fully agentic approaches still struggle to convert diagnosis into successful intervention.

Detection is relatively fast, but user impact and efficiency vary substantially across methods and backbones. On Kimi K2.5, methods that correlate faults at scale detect them within 146–189 s, corresponding to roughly one to two agent invocations. User-facing degradation, however, is much less consistent. RAG, Codex, OpsMem, and COCA worsen degradation on Kimi K2.5 (degradation level 124.3% to 150.5%) but improve it on GPT-5.6 Luna (77.4% to 93.5%), while ReAct shows the opposite trend (86.9% versus 100.8%). This backbone sensitivity suggests that successful diagnosis alone is not sufficient to predict user-facing benefit. Cost also shows little correspondence with outcome: on Kimi K2.5, RAG and Keep-agent use 569k and 685k tokens per run, compared with 5.4M–7.2M for the more tool-intensive agents, without a consistent performance advantage. No method attempts a forbidden operation, so Risk reflects only out-of-scope actions; ReAct is highest at 0.217 on Kimi K2.5 and 0.349 on GPT-5.6 Luna.

The repair bottleneck persists across backbones and seeds, while lower load improves end-to-end success. On GPT-5.6 Luna, localization reaches at most 0.313 and repair at most 0.023, showing that the same failure mode is not specific to Kimi K2.5. The conclusion also holds across two additional Astronomy Shop seeds, where ReAct scores 7.3 and 17.0 and repairs at most 0.051 of faults (Appendix H, Table 7). Under lower load, however, the same ReAct agent scores 38.9 on five-fault runs and 34.8 on ten-fault runs, compared with 25.4 on marathon runs, and handles 16.7%, 6.7%, and 0.0% of faults end to end (Appendix I, Table 8). Because the short runs also differ in venue, invocation interval, and seed, this comparison does not isolate load experimentally, but it is consistent with increasing concurrency being a major source of failure.

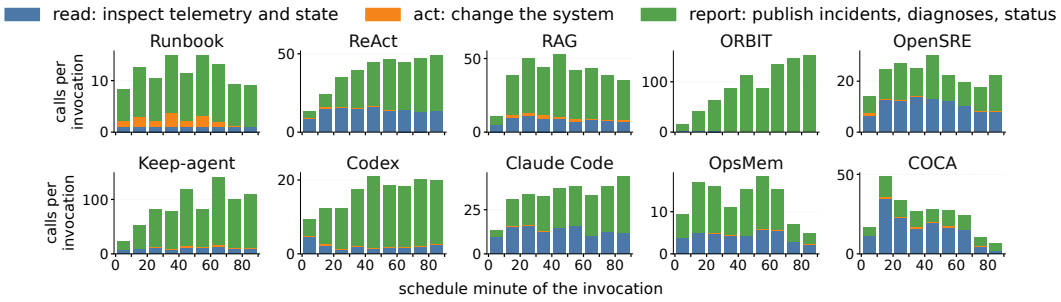


Figure 4: Agent actions over time during a marathon run: tool calls per invocation, by tier, in ten-minute bins of the schedule, for every method of Table 2 on Kimi K2.5 and for Runbook.

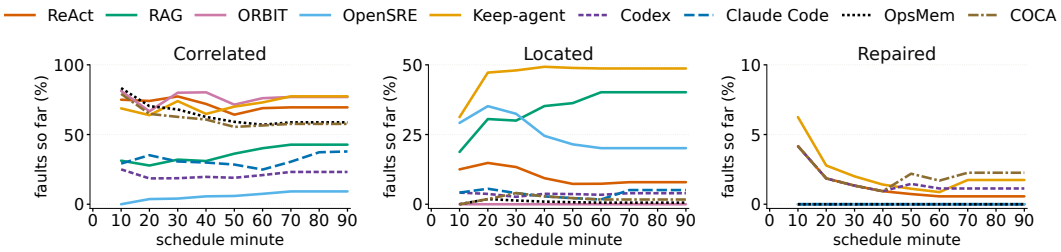


Figure 5: Stage-wise success over time in fault handling. Each panel shows the % of faults injected so far that have been successfully correlated, located, or repaired by each method on Kimi K2.5.

5.2 Performance analysis under sustained operation

Figures 4 and 5 show how agent activity and stage-wise success evolve over a marathon run, using the Kimi K2.5 runs from Table 2.

As fault load increases, agents report more without acting more. Figure 4 groups tool calls into reads, system-changing actions, and incident reports. Call volume rises during the injection period for most LLM-based methods, but the increase is dominated by reporting. ReAct's report calls grow from 3.9 to 35.5 per invocation, while its reads change only from 8.9 to 13.8; ORBIT shows a similar increase from 13.3 to 152.2. Codex's reads decline as its reporting increases. Actions remain sparse, never exceeding 5.3 per invocation in any bin, and by the final ten minutes only RAG and Keep-agent still average at least one action per invocation. Thus, increasing load produces substantially more incident reporting, but little additional intervention.

Correlation remains stable, localization separates methods, and repair remains rare. Figure 5 reports the fraction of faults injected so far that have been successfully correlated, located, or repaired. Correlation remains relatively stable as faults accumulate, with several methods maintaining similar success rates throughout the run. Localization shows greater separation: Keep-agent and RAG improve or remain strong, while OpenSRE declines from 0.29 to 0.20 and most other methods remain low. Repair is consistently the weakest stage; no Kimi K2.5 method exceeds 0.06 of injected faults at any point, and the few methods with early repairs generally do not sustain that rate as the run progresses. Overall, agents continue to correlate incidents under increasing load, but only a few translate this into reliable localization, and almost none into successful repair.

5.3 Case studies

Figure 6 compares two runs containing the same fault type, a bad image tag whose correct remedy is rollback. The shaded focal window marks the lifetime of the fault discussed below.

On Online Boutique, the fault is injected on `checkoutservice` at minute 62 (schedule drift; Appendix A), while 4 other faults are active. Keep-agent groups the relevant alerts, identifies the service and artifact, and rolls the service back at minute 65. The probe recovers by minute 70 and remains healthy until the fault is reverted at minute 72, so the fault is handled end to end.

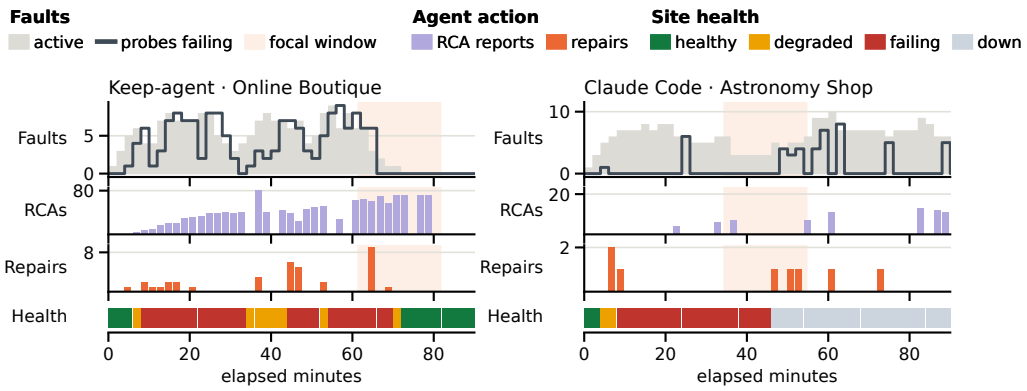


Figure 6: Two marathon runs containing the same fault type and correct remedy. The shaded focal window marks the lifetime of the fault discussed in each case. From top to bottom, the lanes show active faults (grey) and those with a failing symptom probe (black), RCA (Root Cause Analysis) reports, executed repairs, and user-facing service health.

On Astronomy Shop, Claude Code also identifies the affected service, but first attempts two ineffective configuration changes and a manifest patch rejected by the policy enforcement layer. It reaches the correct rollback only at minute 52, after 3 attempts, too late for the repair to take effect before the fault is reverted at minute 54. This illustrates a central failure mode: agents can identify the right cause yet still fail to remediate it because of inefficient action selection and delayed intervention. Additional failure analysis appears in Appendix J.

6 Related Work

Benchmarks for agents. SRE benchmarks either evaluate diagnosis over frozen telemetry (Xu et al., 2025b; Pham et al., 2025; Fang et al., 2026), expose a fixed incident through mocked tools or a recorded window (Wang et al., 2026; Gong et al., 2026), or allow an agent to observe and modify a live cluster one fault episode at a time (Chen et al., 2025b; Jha et al., 2025; Clark et al., 2026). Beyond SRE, agent benchmarks evaluate closed-loop behavior against tests, policies, and cost (Jimenez et al., 2024; Yao et al., 2025; Merrill et al., 2026), assess coherence over long horizons (Backlund & Petersson, 2025; Xu et al., 2025a), and study failures caused by reward design and non-reproducible scoring (Zhu et al., 2025). SRE-Marathon extends long-horizon evaluation to SRE through continuous operation on a live system, overlapping faults including defects delivered through the build pipeline, and deterministic scoring from sealed run recordings (§3).

LLM agents for the incident lifecycle. Most SRE agents target a particular stage of the incident lifecycle: alert correlation and rule generation (Kuang et al., 2024; Jiang et al., 2025; Gu et al., 2025; Wang et al., 2024b), root-cause analysis from tickets, telemetry, and code (Chen et al., 2024; Jiang et al., 2024; Wang et al., 2024a; Roy et al., 2024; Pei et al., 2025; Tian et al., 2026), or mitigation from runbooks (An et al., 2024; Mao et al., 2026; Zhang et al., 2024); multi-agent pipelines span detection through repair (Chen et al., 2025a; Luo et al., 2026). Deployed systems include coding agents applied to clusters (Anthropic, 2026; OpenAI, 2026a), open-source SRE agents and alert platforms (Tracer Cloud, 2026; Keep, 2026), and provider offerings (Amazon Web Services, 2025; Microsoft, 2025; Resolve AI, 2026; Bharadwaj & Tu, 2026).

7 Conclusion

SRE-Marathon evaluates SRE agents under sustained production operation, where faults arrive through noisy alerts, overlap, and may arise from code or configuration changes. Its metrics are computed deterministically from recorded system evidence, enabling reproducible offline evaluation. Across ten methods and two backbones, the best Marathon-Score reaches only 41.3 out of 100. Agents often correlate and localize faults, but rarely convert diagnosis into timely repair as load increases. Appendix K discusses the limitations.

Reproducibility statement

To support reproducibility, we share the system setup for both venues, the deployment code for the three applications, the fault specifications of the fault catalog and their validation records, the frozen alert rule pack, the fault orchestrator and the agent–system interface with its policy enforcement layer, the baseline implementations, and the scorer in an anonymous repository at `https://anonymous.4open.science/r/SRE-Marathon-12321/`.

Ethics statement

This work involves no user study and no private or sensitive data. All faults, traffic, and agent actions run in containerized sandbox environments, isolated from any production system.

AI use statement

Large language models were used as coding assistants to implement functions, to polish the writing, and to retrieve web URLs for references. All AI-assisted code, text, and references were reviewed and validated by the authors, who take responsibility for the final content.

References

Amazon Web Services. AWS DevOps agent. `https://aws.amazon.com/devops-agent/`, 2025. Accessed 2026-09-16.

Kaikai An, Fangkai Yang, Junting Lu, Liqun Li, Zhixing Ren, Hao Huang, Lu Wang, Pu Zhao, Yu Kang, Hua Ding, et al. Nissist: An incident mitigation copilot based on troubleshooting guides. In *ECAI 2024: 27th European Conference on Artificial Intelligence, 19–24 October 2024, Santiago de Compostela, Spain–Including 13th Conference on Prestigious Applications of Intelligent Systems (PAIS 2024)*, pp. 4471–4474, 2024.

Anthropic. Model context protocol specification. `https://modelcontextprotocol.io/specification`, 2024. Accessed 2026-09-16.

Anthropic. Claude Code. `https://code.claude.com`, 2026.

Axel Backlund and Lukas Petersson. Vending-Bench: A benchmark for long-term coherence of autonomous agents. *arXiv preprint arXiv:2502.15840*, 2025.

Aprameya Bharadwaj and Kyle Tu. Agentic observability: Automated alert triage for Adobe e-commerce. In *AAAI 2026 Workshop on Agentic AI Benchmarks and Applications for Enterprise Tasks*, 2026. URL `https://arxiv.org/abs/2602.02585`.

Yinfang Chen, Huaibing Xie, Minghua Ma, Yu Kang, Xin Gao, Liu Shi, Yunjie Cao, Xuedong Gao, Hao Fan, Ming Wen, et al. Automatic root cause analysis via large language models for cloud incidents. In *Proceedings of the Nineteenth European Conference on Computer Systems*, pp. 674–688, 2024.

Yinfang Chen, Jiaqi Pan, Jackson Clark, Yiming Su, Noah Zheutlin, Bhavya Bhavya, Rohan R Arora, Yu Deng, Saurabh Jha, and Tianyin Xu. STRATUS: A multi-agent system for autonomous reliability engineering of modern clouds. *Advances in Neural Information Processing Systems*, 38:50119–50165, 2025a.

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. *Proceedings of Machine Learning and Systems*, 7, 2025b.

Jackson Clark, Yiming Su, Saad Mohammad Rafid Pial, Yifang Tian, Lily Gniedziejko, Hans-Arno Jacobsen, Yinfang Chen, and Tianyin Xu. SREGym: A live benchmark for AI SRE agents with high-fidelity failure scenarios. *arXiv preprint arXiv:2605.07161*, 2026.

Aoyang Fang, Yifan Yang, Jin'ao Shang, Qisheng Lu, Junjielung Xu, Rui Wang, Songhan Zhang, Yuzhong Zhang, Boxi Yu, and Pinjia He. OpenRCA 2.0: From outcome labels to causal process supervision. *arXiv preprint arXiv:2606.27154*, 2026.

Supriyo Ghosh, Manish Shetty, Chetan Bansal, and Suman Nath. How to fight production incidents? An empirical study on a large-scale cloud service. In *Proceedings of the 13th Symposium on Cloud Computing*, pp. 126–141, 2022.

Albert Gong, Kyuseong Choi, Abhineet Agarwal, Jason Schechner, Ryan Huang, Raj Agrawal, Anish Agarwal, and Raaz Dwivedi. ORCA-bench: How ready are language model agents for oncall? *arXiv preprint arXiv:2607.28545*, 2026.

Google Cloud Platform. Online Boutique: A cloud-first microservices demo application. `https://github.com/GoogleCloudPlatform/microservices-demo`, 2019.

Grafana Labs. Grafana Loki: Log aggregation system. `https://grafana.com/oss/loki/`, 2026. Accessed 2026-09-16.

Yile Gu, Yifan Xiong, Jonathan Mace, Yuting Jiang, Yigong Hu, Baris Kasikci, and Peng Cheng. Argos: Agentic time-series anomaly detection with autonomous rule generation via large language models. *arXiv preprint arXiv:2501.14170*, 2025.

Jaeger. Jaeger: Open source, distributed tracing platform. `https://www.jaegertracing.io`, 2026. Accessed 2026-09-16.

Saurabh Jha, Rohan R Arora, Yuji Watanabe, Takumi Yanagawa, Yinfang Chen, Jackson Clark, Bhavya Bhavya, Mudit Verma, Harshit Kumar, Hirokuni Kitahara, et al. ITBench: Evaluating AI agents across diverse real-world IT automation tasks. In *International Conference on Machine Learning*, pp. 27134–27197. PMLR, 2025.

Yuxuan Jiang, Chaoyun Zhang, Shilin He, Zhihao Yang, Minghua Ma, Si Qin, Yu Kang, Y. Dang, S. Rajmohan, Q. Lin, and D. Zhang. Xpert: Empowering incident management with query recommendations via large language models. In *Proceedings of the IEEE/ACM 46th International conference on software engineering*, pp. 1–13, 2024.

Zhihan Jiang, Jinyang Liu, Yichen Li, Haiyu Huang, Xiao He, Tieying Zhang, Jianjun Chen, Yi Li, Rui Shi, and Michael R Lyu. LogPilot: Intent-aware and scalable alert diagnosis for large-scale online service systems. In *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 3299–3310. IEEE, 2025.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: Can language models resolve real-world GitHub issues? In *International Conference on Learning Representations*, volume 2024, pp. 54107–54157, 2024.

Keep. Keep: open-source AIOps and alert management platform. `https://github.com/keephq/keep`, 2026. Version 0.54.3.

Jinxi Kuang, Jinyang Liu, Junjie Huang, Renyi Zhong, Jiazhen Gu, Lan Yu, Rui Tan, Zengyin Yang, and Michael R Lyu. Knowledge-aware alert aggregation in large-scale cloud systems: a hybrid approach. In *Proceedings of the 46th International Conference on Software Engineering: Software Engineering in Practice*, pp. 369–380, 2024.

Kubernetes SIGs. kind: Kubernetes in docker. `https://kind.sigs.k8s.io`, 2026. Accessed 2026-09-16.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. Retrieval-augmented generation for knowledge-intensive NLP tasks. *Advances in neural information processing systems*, 33: 9459–9474, 2020.

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. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pp. 1346–1358. IEEE, 2025.

Yu Luo, Jiamin Jiang, Jingfei Feng, Lei Tao, Qingliang Zhang, Xidao Wen, Yongqian Sun, Shenglin Zhang, Tong Liu, Wenjie Zhang, and Dan Pei. OpsAgent: An evolving multi-agent system for incident management in microservices. In *41st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2026. URL https://conf.researchr.org/track/ase-2026/ase-2026-industry-showcase. Accepted, to appear (Industry Showcase).

Jiayi Mao, Liqun Li, Yanjie Gao, Zegang Peng, Shilin He, Chaoyun Zhang, Si Qin, Samia Khalid, Qingwei Lin, Saravan Rajmohan, et al. StepFly: Agentic troubleshooting guide automation for incident diagnosis. *Proceedings of the ACM on Software Engineering*, 3(FSE):3070–3092, 2026.

Mike Merrill, Alexander Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Shin, Thomas Walshe, E Kelly Buchanan, et al. Terminal-Bench: Benchmarking agents on hard, realistic tasks in command line interfaces. In *International Conference on Learning Representations*, volume 2026, pp. 40903–40986, 2026.

Microsoft. Azure SRE agent. https://azure.microsoft.com/en-us/products/sre-agent, 2025. Accessed 2026-09-16.

Moonshot AI. Kimi K2.5. https://www.kimi.ai/ai-models/kimi-k2-5, 2026. Accessed 2026-09-16.

OpenAI. Codex: cloud coding agent. https://chatgpt.com/codex, 2026a. Accessed 2026-09-16.

OpenAI. Gpt-5.6 Luna. https://openai.com/index/gpt-5-6/, 2026b. Accessed 2026-09-16.

OpenTelemetry. OpenTelemetry demo Astronomy Shop. https://github.com/open-telemetry/opentelemetry-demo, 2023. CNCF.

OpenTelemetry. OpenTelemetry: High-quality, ubiquitous, and portable telemetry. https://opentelemetry.io, 2026. Accessed 2026-09-16.

Jiayi Pan, Xingyao Wang, Graham Neubig, Navdeep Jaitly, Heng Ji, Alane Suhr, and Yizhe Zhang. Training software engineering agents and verifiers with SWE-Gym. In *International Conference on Machine Learning*, pp. 47717–47737. PMLR, 2025.

Changhua Pei, Zexin Wang, Fengrui Liu, Zeyan Li, Yang Liu, Xiao He, Rong Kang, Tieying Zhang, Jianjun Chen, Jianhui Li, et al. Flow-of-action: SOP enhanced LLM-based multi-agent system for root cause analysis. In *Companion Proceedings of the ACM on Web Conference 2025*, pp. 422–431, 2025.

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

Prometheus. Prometheus: Monitoring system and time series database. https://prometheus.io, 2026. Accessed 2026-09-16.

Resolve AI. Resolve.ai: AI for production operations. https://resolve.ai, 2026. Accessed 2026-09-16.

Devjeet Roy, Xuchao Zhang, Rashi Bhave, Chetan Bansal, Pedro Las-Casas, Rodrigo Fonseca, and Saravan Rajmohan. Exploring LLM-based agents for root cause analysis. In *Companion proceedings of the 32nd ACM international conference on the foundations of software engineering*, pp. 208–219, 2024.

Zafir Stojanovski, Oliver Stanley, Joe Sharratt, Richard Jones, Abdulhakeem Adefioye, Jean Kaddour, and Andreas Köpf. Reasoning gym: Reasoning environments for reinforcement learning with verifiable rewards. *Advances in Neural Information Processing Systems*, 38, 2025.

Yongqian Sun, Rongchen Gao, Yu Luo, Wenwei Gu, Shenglin Zhang, Qingyi Guo, Qiuai Fu, Yaoliang Wu, and Dan Pei. OpsMem: Dual-memory reasoning with cross-memory resonance for failure diagnosis. *arXiv preprint arXiv:2607.11357*, 2026.

Yifang Tian, Yaming Liu, Zichun Chong, Zihang Huang, Yiran Li, and Hans-Arno Jacobsen. GALA: Graph-augmented LLM agents for root cause analysis and incident response in microservices. In *41st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2026. URL https://conf.researchr.org/details/ase-2026/ase-2026-research-track/5/GALA. Accepted, to appear (Research Papers).

Tracer Cloud. OpenSRE: an open-source SRE agent. https://github.com/Tracer-Cloud/opensre, 2026. Version 0.1, public alpha.

Yilun Wang, Guangba Yu, Haiyu Huang, Yujie Huang, Zirui Wang, Pengfei Chen, and Michael R Lyu. Cloud-OpsBench: A reproducible benchmark for agentic root cause analysis in cloud systems. *arXiv preprint arXiv:2603.00468*, 2026.

Zefan Wang, Zichuan Liu, Yingying Zhang, Aoxiao Zhong, Jihong Wang, Fengbin Yin, Lunting Fan, Lingfei Wu, and Qingsong Wen. RCAgent: Cloud root cause analysis by autonomous agents with tool-augmented large language models. In *Proceedings of the 33rd ACM international conference on information and knowledge management*, pp. 4966–4974, 2024a.

Zexin Wang, Jianhui Li, Minghua Ma, Ze Li, Yu Kang, Chaoyun Zhang, Chetan Bansal, Murali Chintalapati, Saravan Rajmohan, Qingwei Lin, et al. Large language models can provide accurate and interpretable incident triage. In *2024 IEEE 35th International Symposium on Software Reliability Engineering (ISSRE)*, pp. 523–534. IEEE, 2024b.

Frank Fangzheng Xu, Yufan Song, Boxuan Li, Yuxuan Tang, Kritanjali Jain, Mengxue Bao, Zora Wang, Xuhui Zhou, Zhitong Guo, Murong Cao, et al. TheAgentCompany: Benchmarking LLM agents on consequential real world tasks. *Advances in Neural Information Processing Systems*, 38, 2025a.

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*, volume 2025, pp. 84562–84590, 2025b.

Tianyin Xu, Xinxin Jin, Peng Huang, Yuanyuan Zhou, Shan Lu, Long Jin, and Shankar Pasupathy. Early detection of configuration errors to reduce failure damage. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pp. 619–634, 2016.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *11th International Conference on Learning Representations, ICLR 2023*, 2023.

Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik Narasimhan. τ-bench: A benchmark for tool-agent-user interaction in real-world domains. In *13th International Conference on Learning Representations, ICLR 2025*, pp. 74824–74876. International Conference on Learning Representations, ICLR, 2025.

Zuoning Yin, Xiao Ma, Jing Zheng, Yuanyuan Zhou, Lakshmi N Bairavasundaram, and Shankar Pasupathy. An empirical study on configuration errors in commercial and open source systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pp. 159–172, 2011.

Lingzhe Zhang, Tong Jia, Mengxi Jia, Yifan Wu, Aiwei Liu, Yong Yang, Zhonghai Wu, Xuming Hu, Philip Yu, and Ying Li. A survey of AIOps in the era of large language models. *ACM Computing Surveys*, 58(2):1–35, 2025.

Xuchao Zhang, Tanish Mittal, Chetan Bansal, Rujia Wang, Minghua Ma, Zhixin Ren, Hao Huang, and Saravan Rajmohan. FLASH: A workflow automation agent for diagnosing recurring incidents. *Microsoft Research Technical Report (preprint)*, 2024.

Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chao Ji, Wenhai Li, and Dan Ding. Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study. *IEEE Transactions on Software Engineering*, 47(2):243–260, 2021.

Yuxuan Zhu, Tengjun Jin, Yada Pruksachatkun, Andy Zhang, Shu Liu, Sasha Cui, Sayash Kapoor, Shayne Longpre, Kevin Meng, Rebecca Weiss, et al. Establishing best practices in building rigorous agentic benchmarks. *Advances in Neural Information Processing Systems*, 38, 2025.

A Environment setup

Both venues of §4 run on the same OpenStack cloud and carry the same application, telemetry, and fault-injection stack, deployed from Helm charts pinned by archive hash and images pinned by digest.

Single-host venue. One virtual machine with 16 vCPUs, 32 GB of RAM, and a 300 GB volume hosts a Kind cluster of one control-plane node and four workers, two per zone through the standard zone label. Latency of 25 ms one way, about 50 ms round trip, is injected between the zones with `tc netem`, which is typical of two sites on one continent. Calico replaces Kind's default network plugin so that network faults behave as they would on a real network, and etcd keeps its data in memory because the virtual disk's write latency destabilized the control plane. Storage is the node's local path, images are preloaded, and the cluster never pulls one during a run.

Multi-node venue. Seven virtual machines form a kubeadm cluster: three control-plane nodes with 4 vCPUs and 8 GB each, placed on distinct hypervisors behind a virtual address that survives failover, and four workers with 16 vCPUs, 32 GB, and a 100 GB volume each, two per zone. No latency is injected between the zones. An eighth machine serves as bastion, image build host, and the registry the cluster pulls images from, and persistent volumes are Ceph-backed. Automatic upgrades are disabled so that no unattended restart falls inside a run.

Applications. Astronomy Shop is the OpenTelemetry Demo (Helm chart 0.40.10) with 20 services, built from a fork that carries the seeded defects and deployed as an active and a standby site across the two zones. Online Boutique is upstream v0.10.6, 11 services rebuilt with OpenTelemetry instrumentation. Train Ticket is the FudanSELab release at commit `313886e9`, 41 workloads on upstream images. Direct faults are injected with chaos engineering from a namespace the agent cannot see, and models are hosted services called through the metering proxy, so no host needs a GPU.

Schedule clock. The fault schedule advances by invocation rather than by wall-clock time: faults are injected and reverted between invocations, and the next invocation starts only once they are in place. A change-driven fault is deployed and reverted through an image rollout that can take minutes, so runs with many such faults, notably on Astronomy Shop, drift beyond their nominal 90 minutes of wall-clock time, while every method still receives the same invocations and faults in the same order.

B The fault catalog

Table 3 lists the 700 catalog faults by category and application. The seven categories are those of the incident study the schedule follows (§2). Change-driven code and configuration faults also carry one of the eight defect subtypes of Ghosh et al. (2022), plus latent defect enabled by a feature flag in code bug and deployed setting in config bug. A fault is categorized by the cause of its failure rather than by its injection mechanism. A patch whose only defect is a wrong setting is therefore a configuration fault even though it arrives as source code.

C Generating code and configuration faults

The change-driven faults of Table 3 are the code bug and config bug rows with one of the eight defect subtypes. They are written by hand and validated on a live deployment as follows.

Table 3: Catalog faults by category, application, and defect subtype, with an example of each.

Category and defect subtype	Astro	OB	TT	Total	Example
Code bug					
Feature or logic error	48	6	6	60	A new currency lookup omits a supported currency, so requests in that currency receive an incorrect default value.
Incorrect constant or flag	5	6	6	17	A new length limit on cart identifiers is set too low, so the cart rejects identifiers it previously accepted.
Component incompatibility	23	6	6	35	New code reads database rows by column name while the adapter still returns values by position, so the queries fail.
Type, validation, or exception error	29	6	6	41	A new price check reads only the whole-number part of a price, so valid prices below one dollar are rejected.
Backward incompatibility	45	8	8	61	An update changes the request fields a service expects without accepting the older format, so callers still sending it are rejected.
Latent defect enabled by a feature flag	10	–	–	10	A memory leak or a failing checkout path switched on at runtime.
Config bug					
Invalid setting	1	6	6	13	A dependency address is edited to a port the destination does not listen on, so calls to it are refused.
Uncoordinated change	5	6	6	17	Traffic is forwarded to a new port while the destination still listens on the original one, so requests fail to connect.
Inconsistent names or values	7	6	6	19	Code reads a setting under a new name while the deployment still supplies the old one, so the service runs without the value.
Deployed setting	22	57	75	154	A readiness probe that cannot pass, a memory limit too small, a wrong DNS policy.
Chaos test					
Deployment error	67	32	44	143	A bad image tag, a scale to zero, a missing toleration.
Dependency failure	20	22	23	65	A downstream service made unreachable, slow, or lossy.
Infrastructure	9	1	3	13	CPU, file-descriptor, or port exhaustion.
Database / network	10	–	–	10	A DNS name that stops resolving, a network policy that isolates a service, a cache failure.
Auth failure	16	11	15	42	An RBAC denial for an init container, a disrupted shared-cache credential.
Total	317	173	210	700	

Authoring. A code fault starts from a service's source. We design a feature addition or refactoring that carries an incorrect assumption about existing data, callers, or dependencies, and confirm that ordinary requests reach the defective code. A change that fails only for a benchmark-specific input is rejected. A configuration fault changes a setting written directly into a deployment manifest: a dependency port, a resource limit, a connection host or password, a port on which callers and listener disagree, or a value cleared to model a lost setting. Indirectly loaded configuration, secrets, telemetry settings, and the applications' built-in fault controls are left alone. Each fault records the intended update, the defect, the requests that expose it, the log message it produces and the service that emits it, the patch or the changed values, and a recovery procedure. Agents see none of these records.

Validation. Each fault is injected on the tested application version under the test workload, and two five-minute periods are compared, before and after the change. The recorded message must be absent before and present after, in the changed service or a direct dependant, and the application must keep serving requests. The fault must also meet one screening criterion: server errors averaging at least four per minute, at least ten new error log events per minute attributable to the change, or a service-instance restart. These thresholds screen for observable faults and do not measure degradation, since one failed request can produce several log events. For a code patch, text introduced by the patch must appear in the new image and in no earlier one, because a runtime error alone does not show that the patched image is running. For a configuration change the deployed value is checked, the change must still allow startup and readiness, and the whole test is repeated after the setting is restored, with only the second observation counting as evidence (Yin et al., 2011). Finally the orchestrator's own injection and recovery are exercised. Recovery must restore the original image or setting, readiness, request processing, and a cleared message, and the other site must be unaffected. A fault validated this way, alone and within five minutes, may still be masked when other faults are active or expose itself only later (Xu et al., 2016).

D THE ALERT RULE PACK

The frozen pack holds 44 rules built from the 18 templates of Table 4: 13 apply to every application and 5 to Astronomy Shop only, whose Kafka and rollout metrics the other two applications do not expose. Each rule is scoped to its application's namespaces, the request rules read the application's entry-point service, and "holds for" is how long a condition must hold before the alert fires.

Table 4: The rule templates of the frozen pack. The condition paraphrases the PromQL expression, and the error-share rules also require a minimum request rate.

Rule	Fires when	Severity	Holds for	Applications
`HighRequestErrorRate`	5xx share of entry-point requests above 10%	Critical	1m	All
`EndpointErrorRate`	5xx share of one entry-point route above 50%	Critical	1m	All
`HighRequestLatency`	95th-percentile latency of a service above 3000 ms	Critical	1m	All
`PodStatusError`	A container is waiting with a reported reason	Critical	1m	All
`FailedPodsDetected`	A pod is in phase Failed	Critical	1m	All
`DeploymentNotReady`	A deployment with replicas requested has none available	Critical	1m	All
`EndpointErrorRateSensitive`	5xx share of one entry-point route above 10%	Warning	1m	All
`HighRequestRate`	Entry-point request rate above 50 per second	Warning	1m	All
`ServiceLatencyElevated`	90th-percentile latency of a service above 1000 ms	Warning	1m	All
`PendingPodsDetected`	A pod is in phase Pending	Warning	1m	All
`PodSchedulingFailure`	A pod cannot be scheduled	Warning	30s	All
`PodContainerRestarting`	A container restarted in the last 10 minutes	Warning	1m	All
`PodNotReady`	A pod is not ready	Warning	2m	All
`AstronomyShopTelemetryDown`	A scrape target of the telemetry stack is down	Critical	30s	Astro
`ContainerOOMKilled`	A container restarted after being killed for exceeding its memory limit	Critical	1m	Astro
`MessageConsumerLag`	Kafka consumer lag above 1000 records on a partition	Warning	2m	Astro
`DeploymentGenerationMismatch`	A deployment's latest revision has not been observed by its controller	Warning	5m	Astro
`DeploymentReplicasUnavailable`	A deployment has fewer available replicas than requested	Warning	5m	Astro

E INSTRUMENTED TOOL DETAILS

Table 5 lists the 19 tools the benchmark publishes, generated from the tool registry. Read tools are free and logged for cost accounting only. The report tool writes to the scored record and cannot touch the system, so it counts toward neither Intervention risk nor the per-invocation action budget. Act and guarded calls are logged and count toward Intervention risk (§3), and every guarded call must cite an incident the agent has already reported. The forbidden operations are `read_secret`, `delete_namespace`, `edit_benchmark_rule`, and any attempt at the three counts as a violation. Anything the agent writes to its workspace but does not report through `incident_report` does not exist for scoring.

Table 5: The published tools by tier. The policy enforcement layer computes every mutating call's target set from its typed arguments, and every guarded call must name a reported incident.

Tool	Tier	Function
`code_search`	Read	Search the source of one service.
`get_alerts`	Read	Alerts fired in full cumulative alert history, with their labels.
`get_topology`	Read	The service call graph derived from traces.
`kubectl_describe`	Read	Describe one Kubernetes object with its recent events.
`kubectl_get`	Read	Read the current state of one Kubernetes object.
`query_logs`	Read	Recent log lines of one deployment.
`query_metrics`	Read	Evaluate an instant PromQL query.
`query_traces`	Read	Recent traces of one service.
`read_file`	Read	Read one file of a service's source.
`search_logs`	Read	Evaluate a LogQL query across services.
`incident_report`	Report	Publish the scored record: an alert-to-incident mapping, a root-cause claim, or an incident status.
`restart_pod`	Act	Restart the pods of one deployment.
`build_service_image`	Guarded	Build a service image that includes a proposed patch, without deploying it.
`propose_patch`	Guarded	Apply a diff to an isolated clone of one service's source.
`rollback`	Guarded	Return one deployment to its previous revision.
`scale_deployment`	Guarded	Set the replica count of one deployment.
`set_feature_flag`	Guarded	Set one feature flag and restart the services that read it.
`submit_patch`	Guarded	Apply a diff to one service, then build and deploy its image through the release pipeline.
`update_config`	Guarded	Change one field of a Deployment, Service, or ConfigMap.

F WORKED EXAMPLES FOR EVERY SCORED FORMULA

We work every quantity of §3 on one small run: the orchestrator injects four faults, f_1 to f_4, and the agent's final report holds three incidents, I_1 to I_3.

Correlated. I_1 holds an alert attributable to f_1, so $a_1 = 1$. I_2 holds alerts attributable to both f_2 and f_3. Faults and incidents are matched one-to-one, so only one of the two is matched, here f_2: $a_2 = 1$ and $a_3 = 0$. I_3 holds only an alert attributable to no fault, and no incident holds an alert of f_4, so $a_4 = 0$. $\mathrm{Corr} = 2/4 = 0.500$. Filing every alert under one incident would have matched one fault at most.

Located. The diagnosis associated with f_1 names its culprit service and its responsible artifact, so $b_1 = 1$. The diagnosis associated with f_2 names the right service and the wrong artifact, so $b_2 = 0$. No diagnosis names f_3 or f_4, so $b_3 = b_4 = 0$. $\mathrm{Loc} = 1/4 = 0.250$.

Repaired. The agent rolls back the culprit service of f_1, and the fault's symptom check returns to healthy and remains healthy until the orchestrator reverts the fault: $c_1 = 1$. It restarts the service of f_2, and the symptom check returns to healthy but fails again before the revert: $c_2 = 0$. The symptom check of f_3 recovers only when the orchestrator reverts the fault, and benchmark cleanup is never credited: $c_3 = 0$. The symptom check of f_4 never failed, and its responsible artifact was not verified to have been corrected: $c_4 = 0$. $\mathrm{Rep} = 1/4 = 0.250$.

Marathon-Score. f_1 passes all three stages, so it is handled and earns $\frac{1+1+1}{3} = 1$. f_2 is correlated but not located, so it earns $\frac{1}{3}$. f_3 and f_4 are not correlated and earn nothing, as would a fault located without being correlated. The score is $100 \cdot \frac{1}{4} \cdot \left(1 + \frac{1}{3}\right) = 33.3$. Had the run attempted one forbidden operation, blocked or not, $d = 1$ and the score would be $100 \cdot \frac{1}{2} \cdot \frac{1}{4} \cdot \left(1 + \frac{1}{3}\right) = 16.7$.

Intervention risk. The agent executed ten interventions, two outside the affected scope of the incident they referenced, and attempted one forbidden operation: $n_{\mathrm{exec}} = 10$, $n_{\mathrm{out}} = 2$, $n_{\mathrm{forb}} = 1$, and $\mathrm{Risk} = (2 + 1)/(10 + 1) = 0.273$.

Service degradation. The integral is taken by the trapezoidal rule over the recorded timestamps. Suppose the entry-point service serves $q = 10$ requests/s and $e + \ell$ is 0, 0.2, and 0 at 0, 60, and 120 s. Then $\min\{1, e + \ell\}\, q$ is 0, 2, and 0 requests/s, and $D_{\mathrm{agent}} = 60 \cdot \frac{0+2}{2} + 60 \cdot \frac{2+0}{2} = 120$ degraded requests. Against $D_{\text{no-action}} = 240$ from the no-action run with the same application configuration

and fault-schedule seed, the degradation level is $120/240 = 50\%$. With $D_{\text{agent}} = 300$ it would be $300/240 = 125\%$: the agent left users worse off than no agent would have.

MTTD, MTTL, and MTTR. Each median is taken over the faults that reached the corresponding stage, so MTTD counts only correlated faults. The first incident reports containing an alert of f_1 and of f_2 come 120 and 360 s after their injections. The alert of f_3 also sits in I_2, but f_3 is not correlated and has no detection time. The median is the $\lceil 0.5\,n \rceil$-th smallest time, so it is always an observed time: with $n = 2$ it is the first, and $\mathrm{MTTD} = 120$ s over two faults. f_1 is the only fault located and the only one repaired, at 240 and 480 s, so $\mathrm{MTTL} = 240$ s and $\mathrm{MTTR} = 480$ s, each over one fault. A stage that no fault passed has no time and prints a dash.

Cost. The run made 200 model calls with 5.0M input and 0.1M output tokens, priced at $3.10, so it reports 200 calls, 5.1M tokens, and $3.10; Table 2 prints the tokens and dollars.

G Baselines and coding agents

Runbook. Runbook is inspired by SRE day-to-day workflow: it calls no model and groups alerts into one incident per workload their labels name, and opens a new incident when a closed one's workload alerts again. A fixed table (Table 6) maps each alert rule to one scripted action and a blamed fault category: roll back on error, latency, and stalled-rollout alerts, scale up on unavailable-deployment and load alerts, and restart on pod failures and any other alert. When several rules fire, a rule naming the broken object wins over a symptom rule. For each new incident it reports the workload and the table's category as the root cause, runs the action once, and marks the incident resolved at the next invocation if the action succeeded.

Table 6: Runbook's rule table in priority order: when an incident's alerts fire several rules, the first listed wins. The Online Boutique and Train Ticket rules carry an application prefix and take the row of their unprefixed name.

Alert rule	Action	Reported category
`DeploymentNotReady`	`scale_deployment` to 2	`deployment`
`DeploymentReplicasUnavailable`	`scale_deployment` to 2	`deployment`
`DeploymentGenerationMismatch`	`rollback`	`deployment`
`ContainerOOMKilled`	`restart_pod`	`code-bug`
`PodStatusError`	`restart_pod`	`code-bug`
`PodNotReady`	`restart_pod`	`code-bug`
`FailedPodsDetected`	`restart_pod`	`code-bug`
`PodContainerRestarting`	`restart_pod`	`code-bug`
`PodSchedulingFailure`	`restart_pod`	`infrastructure`
`PendingPodsDetected`	`scale_deployment` to 2	`infrastructure`
`MessageConsumerLag`	`scale_deployment` to 2	`infrastructure`
`AstronomyShopTelemetryDown`	`restart_pod`	`infrastructure`
`HighRequestErrorRate`	`rollback`	`code-bug`
`EndpointErrorRate`	`rollback`	`code-bug`
`EndpointErrorRateSensitive`	`rollback`	`code-bug`
`HighRequestLatency`	`rollback`	`configuration`
`ServiceLatencyElevated`	`rollback`	`configuration`
`HighRequestRate`	`scale_deployment` to 2	`infrastructure`
any other alert	`restart_pod`	`infrastructure`

ReAct. ReAct (Yao et al., 2023) starts a fresh conversation at every invocation. Its first message holds the cumulative alert history and the incidents it opened earlier, which are the only state it keeps between invocations. Before the model runs, new alerts are filed under the first unresolved incident, so every alert has an incident even if the model reports none. The model then alternates reasoning and tool calls over the full tool set of Appendix E, choosing every call itself, until it stops calling tools, ten steps pass, or 150k tokens are spent. It reports incidents, root causes, and statuses through the report tool.

RAG. RAG (Lewis et al., 2020) makes one model call per invocation, and the model calls no tools. It retrieves from the live telemetry of that invocation and reads the topology, then the logs and traces

of up to five services named in the alerts, most recently alerting first, within 45 s. Each tool answer, cut to 1200 characters, is one document. Then it ranks the documents against a query made of every alert's rule name and label values, and the top five enter the prompt beside the alerts and the incidents of earlier invocations. The model returns one plan that groups alerts into incidents and gives each a root cause, an optional repair from the act and guarded tools, and a status. The code publishes and executes the plan as written.

Keep-agent. Keep (Keep, 2026) is an open-source alert-management platform: a rule engine groups alerts into incidents, and workflows run scripted actions. Its model-based correlation is a hosted service outside the open-source release, so we do not evaluate it. Keep-agent runs Keep inside the agent's process at every invocation, keeps its incident store in the persistent workspace, and sends it each alert once. The rules are ours, written in Keep's rule language: one rule per label that names an object (the workload labels first, then the pod, then the request path), each excluding the labels ranked above it, so an alert joins at most one incident. An incident stays open for the whole run, so a second fault on the same workload joins the first fault's incident, and alerts that name no object wait in an unassigned bucket. To this we add one model call per invocation with RAG's prompt, telemetry retrieval, and plan format, unchanged. The prompt lists Keep's incidents as fixed: the model may place only the alerts no rule grouped, and the code discards any move of an alert Keep placed. The model gives each incident a root cause, an optional repair, and a status. Each repair is sent through a Keep workflow whose one HTTP step calls the agent–system interface, so the policy enforcement layer checks it as it checks every method's. Keep-agent and RAG therefore share the model's half and differ in who groups the alerts.

ORBIT. We built ORBIT as an agent that covers every stage the benchmark scores, since no existing agent is designed to correlate, locate, and repair across invocations of a continuous run. Unlike ReAct, which starts a fresh conversation at every invocation, it keeps state across invocations and organizes its own work. Figure 7 shows one invocation. The benchmark starts a fresh process per invocation, and ORBIT reloads a checkpoint, an agenda, its open incidents, and a durable action ledger from its workspace. An action that succeeded while the previous process died is folded back into its incident first. The alert batch is merged deterministically, one incident and one agenda task per signal, and the complete alert-to-incident mapping is published before the model is consulted, so what Correlated grades never depends on the model finishing.

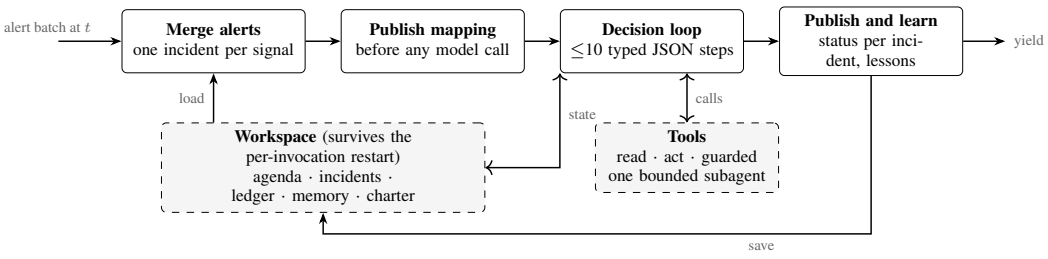


Figure 7: One invocation of ORBIT. Alert merging and both publication steps are deterministic, and the model decides only inside the loop. Everything kept across invocations lives in the workspace.

The model then works inside a bounded decision loop. Each step sends a fixed operating charter, the state, the last observations, the tool catalog, and the remaining budget, and expects one JSON decision: a `tool_call` naming an agenda task, the incident it serves, the expected effect, and the check that will confirm it, or a `finish` returning the rewritten agenda, mapping, and claims. A malformed decision gets one correction call, and a second failure ends the invocation as a charged no-op. Ten steps, a token ceiling, and a mutation cap bound the loop, and a budget ladder reduces it to triage only as the budget runs out. Source-level fixes happen only in one subagent bound to a diagnosed incident and service. Before yielding, ORBIT publishes a status and at least a tentative root cause per open incident, appends at most three lessons to a bounded charter, and promotes a repair recipe to procedural memory only once the incident it resolved has validated green. The contrast with ReAct is the point: same tools and model, but state, protocol, and memory instead of a fresh conversation every invocation.

Codex. We run release `rust-v0.154.0` of Codex (OpenAI, 2026a) unmodified in headless mode, behind a thin adapter that configures, prompts, and relays. At each invocation the benchmark starts a fresh Docker container on its host, outside the Kubernetes cluster, from an image holding the Codex binary and the adapter; the container sees only the run's workspace and the variables the benchmark passes, and is stopped at the invocation's time limit. Its only tools are those of Appendix E, reached through an MCP server the adapter starts; its shell and web search are disabled, so every call passes the policy enforcement layer. The benchmark's rules are given as project instructions, and one Codex conversation, stored in the workspace, is resumed at every invocation, so its own context is its memory across incidents. Each invocation must end with a JSON block listing its incidents, root causes, and statuses, which the adapter publishes through the report tool, and a repair is held until the incident it cites has been reported. Model calls go through the metering proxy, and an invocation stops at 150k tokens or 40 tool calls.

Claude Code. We run release 2.1.273 of Claude Code (Anthropic, 2026) behind the same adapter as Codex, with the same tools, report block, repair ordering, and limits, counting 40 model calls rather than tool calls. It differs from Codex in its memory and local tools. Each invocation starts a fresh session, and the agent's memory is a notes directory it writes itself. It runs in its restricted mode, with only its file tools (read, write, edit, and search) confined to that directory and no shell or web tools. A local shim forwards its API calls to the metering proxy and returns the answers in the format Claude Code expects.

H How far a single run generalizes

Table 7 gives ReAct on Astronomy Shop under the shared seed of Table 2 and two further seeds. Because the fault catalog is large, the schedules that different seeds draw differ widely, and so do the results: the stage shares differ by a factor of two to four between seeds, Intervention risk by more than four, and the Marathon-Score from 7.3 to 20.1. This is why every method is compared under one shared schedule, and it is also what makes the benchmark usable for reinforcement learning, since each new seed is a rollout under a different setting.

Table 7: ReAct on Kimi K2.5, one Astronomy Shop marathon run per seed. Columns are as in Table 2.

Seed	Venue	Corr	Loc	Rep	MTTD	MTTL	MTTR	Risk	Marathon-Score
Shared	Single-host	0.517	0.121	0.017	231	329	709	0.136	20.1
17	Single-host	0.203	0.034	0.051	202	625	593	0.593	7.3
27	Multi-node	0.455	0.073	0.018	208	324	2486	0.250	17.0

I Agent behavior under lighter pressure

We use short runs to see how an agent behaves under lighter pressure than a marathon run. A short run injects five or ten faults, one per invocation, and holds every one open to the end of the run, so the agent has the rest of the run to repair each fault and no fault clears on its own. Each short run is paired with a no-action run on the same schedule and seed that supplies its degradation reference. Six short runs hold five faults and six hold ten, two per application (Table 8).

Table 8: ReAct on Kimi K2.5 by schedule regime. A short-run row averages six short runs, two per application, and the marathon-run row averages the three marathon runs of Table 2. Columns are as in Table 2.

Regime	Runs	Corr	Loc	Rep	Marathon-Score
Short run, 5 faults	6	0.800	0.200	0.500	38.9
Short run, 10 faults	6	0.728	0.283	0.183	34.8
Marathon run, about 60 faults	3	0.694	0.079	0.006	25.4

Under lighter pressure the same ReAct agent repairs far more and reaches a higher Marathon-Score. With five faults it repairs a share of 0.500 and scores 38.9, with ten faults 0.183 and 34.8, and in the marathon runs 0.006 and 25.4. The share of faults handled end to end falls from 16.7% to 6.7% and 0.0%. Correlated moves far less across the three regimes (0.800, 0.728, and 0.694), so what pressure removes is repair rather than the grouping of alerts. The decline is already visible with ten faults, where the commonest outcome is an executed action that does not repair (56% of faults, Appendix J). The short runs differ from the marathon runs in venue, release, seed, invocation interval, and fault lifetime as well as in load, so we read this as a qualitative contrast rather than a controlled measurement of concurrency.

J WHERE THE LIFECYCLE BREAKS

A repaired share near zero, as in every marathon run, can mean that agents stop acting or that they keep acting to no effect. Figure 8 separates the two for the runs of §5.2. Its top row counts executed actions per invocation, leaving out denied or failed calls: in minutes 60–70, Keep-agent makes 5.3 system-changing calls per invocation in Figure 4 but executes 1.8. Its bottom panel sorts every executed action by what it touched.

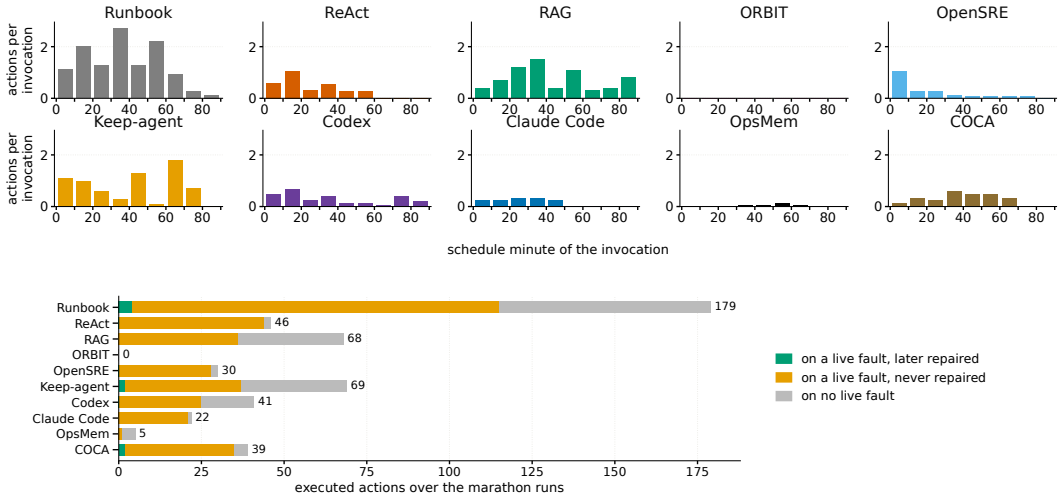


Figure 8: Every method of Table 2 on Kimi K2.5, and Runbook. Top: executed actions per invocation in the ten-minute schedule bins of Figure 4; unlike its act tier, calls that were denied or failed are not counted. Bottom: every executed action of a method's marathon runs by what it touched, with the total at the bar's end. An action is *on a live fault* when its target set holds the faulted service of a fault injected and not yet reverted. Matching is by service, so it says where an action went and not whether the remedy fit.

Agents act on the right service and still repair almost nothing. ORBIT never acts: on Kimi K2.5 its model spends each invocation's few decisions on reads, reports, and subagent hand-offs, and on GPT-5.6 Luna it hands repairs to a source-fix subagent that never submits a patch. OpsMem executes only 5 actions in three runs. Every other method acts through most of the injection period, and more than half of its actions hit the faulted service of a live fault; 44 of ReAct's 46 actions and 28 of OpenSRE's 30 land on faults that are never repaired. Runbook acts most (179 actions), followed by Keep-agent (69) and RAG (68), though 46% and 47% of the latter two's actions hit no live fault. Only 4 of Runbook's actions reach a fault that is later repaired; among the LLM-based methods, only Keep-agent and COCA on Kimi K2.5 (2 and 2) and Keep-agent and Codex on GPT-5.6 Luna (3 and 2, not shown in Figure 8) have any such action. Every other repaired fault in Table 2 was credited through an action on another service in its affected scope, which matching by faulted service misses. When these methods act, they usually reach the right service but apply a remedy that does not fix it.

Methods with similar scores stall at different stages. Figure 9 assigns each injected fault one outcome. On Kimi K2.5, ReAct stops at correlation: 47% of its faults are grouped and taken no

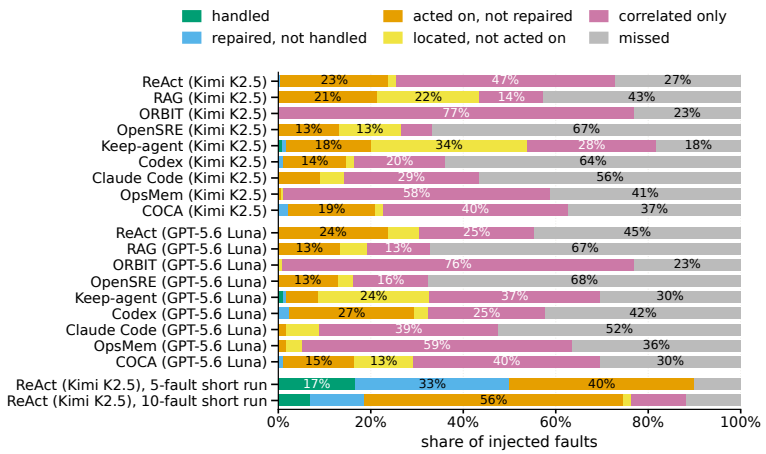


Figure 9: One outcome per injected fault, by the first rule that matches: handled, meaning all three stages passed; repaired but not handled; acted on, meaning an executed action targeted its faulted service while it was live, but not repaired; located but not acted on; correlated only; or missed. Rows are the LLM-based methods of Table 2 on both backbones, plus the two ReAct short-run rows.

further, and 23% are acted on without a repair. RAG, OpenSRE, Codex, and Claude Code most often miss the fault (43%, 67%, 64%, and 56%), while OpsMem and COCA most often stop at correlation (58% and 40%). Keep-agent misses the fewest (18%) and has the largest share of faults it locates without acting on them (34%). ORBIT correlates 77% of its faults and does nothing further. On GPT-5.6 Luna, seven of the nine methods keep their most common outcome. The exceptions are ReAct, which misses more faults (45%) than it leaves at correlation (25%), and Keep-agent, which leaves more at correlation (37%) than it locates without acting (24%). In the two short-run rows, the same ReAct agent handles 17% of faults with five held and 7% with ten (Appendix I). Similar scores therefore hide different failures: some methods group alerts and stop, several never find the fault, Keep-agent locates many faults it does not act on, and the methods that act apply remedies that do not repair.

K LIMITATIONS

SRE-Marathon makes several compromises that bound what its results mean. We group them by where they enter: the environment, the faults, the alerting, the scoring, and the methods we ran.

Environment. On the single-host venue the two zones share one machine, so a zone failure is an emulation: the redundancy semantics are real even where the physics are not. The three applications are public, so memorization is a live risk that newly written code faults and exact-commit artifact targets reduce but do not remove.

Faults. The category mix follows one provider's high-severity incidents over 2020–21, the best public anchor we know of, but only one. A fault's affected scope is measured one fault at a time and is not re-measured under overlap, so attribution is conservative when faults interact.

Alerting. The frozen rule pack sets the difficulty of correlation: a fault below every rule's threshold raises no alert, as a pilot fault showed, and the pack is versioned with the release for that reason. The pack is fixed for the agent as well, since no tool lets it add a rule. Letting a long-horizon agent extend monitoring coverage as it learns the system, with its own alerts kept out of the scored denominator, is a promising direction that the design anticipates but the benchmark does not yet score.

Scoring. Located matches a diagnosis exactly against the release's alias table, so a diagnosis that names the right cause in other words scores zero. We chose exactness over a model judge so that every leaderboard number rests on recorded evidence alone. Repaired credits a rollback and a corrected patch alike. The degradation level is measured against one no-action recording per release.

Backbones and baselines. We chose to run the agents on two LLM backbones, Kimi K2.5 and GPT-5.6 Luna, and neither is the most capable model available. We prioritized budget-friendly models over frontier ones in consideration of cost-effectiveness in real-world scenarios where SREs deal with large volumes of alerts per day. The ten methods span operations practice, reasoning patterns, specialized SRE agents, coding agents, and diagnosis methods, but they cannot cover the entire field, and a method built for this setting may well do better than any reported here.