arXiv:2605.07161v2 [cs.AI] 13 May 2026

SREGym: A Live Benchmark for AI SRE Agents with High-Fidelity Failure Scenarios

Jackson Clark[◇*] **Yiming Su**[◇*] **Saad Mohammad Rafid Pial**[◇] **Yifang Tian**[†]
Lily Gniedziejko[◇] **Hans-Arno Jacobsen**[†] **Yinfang Chen**[◇]
Tianyin Xu[◇]
University of Illinois Urbana-Champaign[◇] University of Toronto[†]

Abstract

AI agents are increasingly used to diagnose and mitigate failures in production systems, known as agentic Site Reliability Engineering (SRE). Current SRE benchmarks are limited to oversimplistic SRE tasks and are unfortunately hard to extend due to bespoke designs. We present SREGYM, a high-fidelity benchmark for SRE agents. SREGYM exposes a live system environment built atop real-world cloud-native system stacks, where high-fidelity failure scenarios are simulated through fault injectors. SREGYM models the complexity of production environments by simulating (1) a wide range of faults at different layers, (2) various ambient noises, and (3) diverse failure modes such as metastable failures and correlated failures. SREGYM is architected as a modular, extensible framework that orchestrates fault and noise injectors across stacks. SREGYM currently includes 90 realistic, challenging SRE problems. We use SREGYM to evaluate frontier agents and show that their capabilities varies significantly in addressing different kinds of failures, with up to 40% differences in end-to-end results. SREGYM is actively maintained as an open-source project and has been used by researchers and practitioners.

1 Introduction

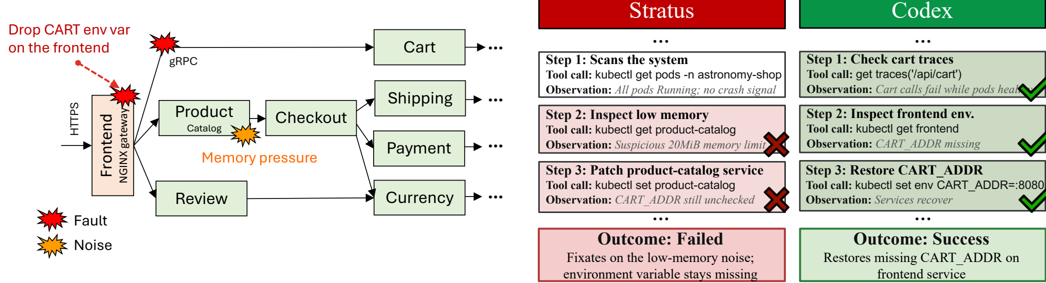
The software lifecycle extends far beyond writing code, encompassing the continuous operation of deployed systems in production. While coding agents have transformed software development [14, 20], accelerated (vibe) coding introduces reliability challenges: AI-generated code is reported to introduce 1.7× more defects than human-written code [57] and 43% of code changes made by AI escape testing and cause production issues [23]. Today, major services are already experiencing production outages caused by AI-generated code [21]. The capabilities of agentic AI must extend beyond writing code to mitigating production incidents, aka Site Reliability Engineering (SRE).

SRE requires capabilities different from coding and Software Engineering (SWE) in general. SRE agents must reason across multi-modal observability data (e.g., system configuration [68, 79], time-series metrics [49, 65], unstructured logs [81, 87], and distributed traces [28, 51, 67]), interact with domain-specific tools, and execute multi-step mitigation plans whose outcomes are only observable at runtime. These requirements make SRE a uniquely challenging domain of agentic AI.

Existing benchmarks are inadequate and fall behind the development of agentic SRE. Static Q&A datasets [24, 55] are limited to domain knowledge. Benchmarks for anomaly detection and root cause analysis (RCA) [43, 48, 77] only evaluate whether AI can detect and analyze failures in static datasets, but not whether they can resolve them. Recent efforts such as AIOpsLab [32] and ITBench [50] take a step forward by creating a live system environment with simulated failures. However, they are limited to *oversimplistic* SRE tasks. For example, they primarily focus on application-layer issues,

[*] Equal contribution.

Preprint.



(a) A failure scenario where a microservice misses an environment variable it needs to send requests to.

(b) Agent trajectories of Stratus (with Sonnet-4.6) and Codex (with GPT-5.4) in addressing the problem.

Figure 1: An SRE problem in SREGYM and the trajectories of two agents when addressing it.

whereas real-world failures have diverse root causes across the stack; faults in lower layers such as operating systems [33] and hardware [44, 62] are known to be harder to address. Moreover, they mostly simulate a single failure into an otherwise clean environment, lacking noises and unrelated events that coexist in production environments [41, 46, 54]. Unfortunately, these benchmarks are hard to extend due to their bespoke designs, e.g., hardcoding failure simulation in problem-specific scripts and lacking support for distributed event coordination.

In this paper, we present SREGYM, a high-fidelity benchmark for SRE agents. SREGYM shares the high-level principle of prior work [32, 50] to expose a *live* system environment built atop real-world cloud-native system stacks, where failures are simulated through fault injectors.[2] Differently, SREGYM models the complexity of dynamic, noisy, and eventful production environments to achieve *high-fidelity* failure scenarios that not only challenge AI but also ensure the relevance of the problems. SREGYM currently includes 90 realistic, challenging SRE problems. Figure 1 shows one problem and the corresponding agent behavior. SREGYM presents three new features:

1. **Simulating a wide range of faults across the system stack**, including hardware faults [44], OS kernel faults [33, 62], misoperations [38, 69], in addition to application issues.
2. **Simulating various ambient noises**, low-impact faults that are unrelated to the root causes of target failures, which introduce ambiguity and potentially cause distractions.
3. **Supporting diverse failure modes** such as metastable behavior [29, 45, 47] and concurrent, correlated failures [36, 82] by orchestrating distributed events (e.g., faults and noises).

SREGYM is architected as a modular, extensible framework that composes fault and noise injectors across stacks into high-fidelity failure scenarios as SRE problems. The modularity and extensibility are not only for engineering disciplines to achieve usability and maintainability (which are critical to SREGYM as a community-driven benchmark; see Appendix B), but also key enablers of its mission of a useful benchmark. For instance, noises must be composed alongside target failures and the manifestation of metastable behavior requires temporal coordination of multiple correlated faults and events. SREGYM provides a unified programming interface to curate high-quality SRE problems by mutating existing failure scenarios (e.g., by altering noises) and creating new ones.

We use SREGYM to evaluate an SRE agent (Stratus [31]) and two coding agents (Claude Code [2] and OpenAI Codex [5]), with different models (including Sonnet-4.6, GPT-5.4, and Kimi K2.5). The success rates of diagnosis and mitigation range from 38.9%–72.6% and 57.3%–78.5% across agent-model pairs, respectively. The agents show strong abilities in addressing application issues, which however drop significantly on failures rooted in other layers/patterns, with up to 40% differences in end-to-end results. For compound failures, agents tend to draw partial conclusions, missing opportunities to address them comprehensively. Similarly, agents are affected by noises in nontrivial ways. Overall, by challenging frontier AI agents and models with high-fidelity failure scenarios, SREGYM provides a foundation for advancing agentic SRE technologies toward production readiness.

SREGYM is actively maintained as an open-source project at `https://github.com/SREGym/SREGym`. It has been used by researchers and practitioners.

[2] We follow the terminology of the classic Fault-Error-Failure model [26, 52] where *faults* are root causes (e.g., software bugs, hardware malfunctions, and misconfigurations); a fault can cause abnormal behaviors referred to as *errors* which (if not handled properly) further propagate and become visible to users that are referred to as *failures*.

2 SREGYM

Building a high-fidelity SRE benchmark is challenging. The benchmark must create a realistic operational environment and simulate sophisticated failure scenarios. We enforce the following design principles in developing SREGYM (Appendix B documents our engineering practices):

- **Creating noisy and eventful environments.** A clean, quiet environment is not only unrealistic but also hard to challenge AI. However, few existing benchmarks model noises. SREGYM offers controlled noises, without compromising reliable evaluation.
- **Simulating faults, not symptoms.** We reject a common practice of existing benchmarks that use chaos engineering tools to create failure symptoms, which can only be mitigated by stopping the tools. Instead, we focus on simulating fine-grained faults.
- **Composability is key to scaling problems.** SREGYM achieves composability with support for orchestrating faults and noises into different failure scenarios. Prior work used Ansible scripts which are hard to extend or to support failure modes that require distributed events.
- **Usability and extensibility are essential.** Usability/extensibility may not reflect academic novelty, but are critical to the success of a benchmark. SREGYM is push-button and offers APIs for problem extension (which enables its users to contribute new problems).

Figure 2 provides an overview of SREGYM in terms of its components. We will present the main components in the remainder of this section.

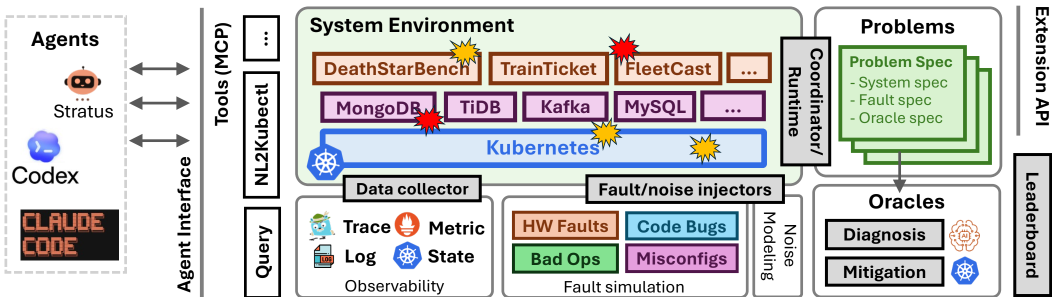


Figure 2: Overview of the SREGYM framework and benchmark suites.

2.1 Problem Definition: A User Perspective

SREGYM currently contains 90 SRE problems. A problem in SREGYM creates a failure scenario in a live production-like environment that an SRE agent must diagnose and mitigate. Formally, a problem is a four-tuple $P = (\mathcal{E}, \mathcal{I}, \mathcal{F}, \mathcal{O})$, including:

- **System environment $\mathcal{E}$.** SREGYM exposes a production-like environment that deploys user applications, backend systems, and control/management services (e.g., Kubernetes controllers).
- **Agent interface $\mathcal{I}$.** The exposed tools and APIs for agents to observe and interact with $\mathcal{E}$.
- **Faults and noises $\mathcal{F}$.** A set $\mathcal{F} = \{f_1, \ldots, f_k\}$ of one or more faults, each targeting a specific component in $\mathcal{E}$. A subset of $\mathcal{F}$ may be designated as *noises*: transient, low-impact disturbances that an agent must distinguish from the root cause(s) of the target failure.
- **Oracles $\mathcal{O} = (\mathcal{O}_d, \mathcal{O}_m)$**, where $\mathcal{O}_d$ is a diagnosis oracle and $\mathcal{O}_m$ is a mitigation oracle.

For each problem, the inputs and expected outputs of evaluated agents are described as follows.

- **Inputs.** The agent can query the system environment $\mathcal{E}$ (with $\mathcal{F}$) through standard observability modalities via the agent interface $\mathcal{I}$.
- **Expected outputs.** The agent makes two submissions for a problem. The first is a natural-language diagnosis describing the root cause of the failure. This submission triggers $\mathcal{O}_d$ (see §2.5). The second submission signals that the agent has completed its mitigation effort and triggers $\mathcal{O}_m$ based on actual system and application states. Following best practices for agent evaluation [22, 92], SREGYM uses programmatic verification whenever possible for reliable evaluation.

```python
class K8sNetworkPortMisconfig(Problem):
  def __init__(self):                                  # Application
    self.app = SocialNetwork()
    self.app.create_workload()
                                                       # Oracle
    self.root_cause = ("The user-service has
     a misconfigured network port [...]")
    self.diagnosis_oracle = LLMAsAJudgeOracle
     (problem=self, expected=self.root_cause)
    self.mitigation_oracle = MitigationOracle
     (problem=self)

  @mark_fault_injected
  def inject_fault(self):                              # Fault
    injector = NetworkPortFaultInjector(
     namespace=self.namespace)
    injector._inject(
      fault_type="port-misconfig",
      microservice="user-service")
```

Figure 3: Implementing a problem in SREGYM (noises are injected by the framework).

Table 1: Fault and noise injectors in SREGYM ("K8s" refers to Kubernetes).

Mechanism	Simulated Faults
Kill a process or a pod	Fail-stop behavior
Stress hardware [12]	Fail-slow behavior [42]
Fail syscall via eBPF	OS/hardware faults [33, 62]
Corrupt a disk sector [19]	Disk sector errors [27, 66]
Fault in deploy.yaml	Service mis-deployment
Fault in app. config	App. misconfiguration [78]
Fault in K8s config	K8s misconfiguration [88]
Use buggy app. code	Code bugs
Use buggy app. operator	Misoperations [38, 39]
Increase client loads	Service overloads [61]
Pause/restart unrelated pods	Temporal pod failures
Inject latency / drop packets	Network delay / jitter
Stress resource of nodes	Noisy neighbors [56, 63]

2.2 System Environments

SREGYM exposes a cloud-native, Kubernetes-based system environment, where applications are deployed in Docker containers. An SRE problem selects which application(s) to deploy based on the failure scenarios (§2.4). SREGYM ships a catalog of cloud-native applications, including DeathStarBench [37], Train Ticket [91], Astronomy Shop [15], and in-house applications (a satellite orbit simulator and a flight booking service), paired with backend data systems (e.g., MongoDB, TiDB, Kafka, MySQL, etc). Each application has corresponding workloads (e.g., user traffic). The applications and backend systems are managed by Kubernetes operators. The applications, systems, and operators are deployed using the Helm package manager [6]. Built atop the *de facto* cloud-native stack, the environment is highly extensible—it supports any real-world cloud-native applications and systems with Kubernetes manifests or Helm charts, and can be deployed on any mainstream hardware platform.

2.3 Agent Interface

SREGYM makes no assumption on the architectures or interaction patterns of evaluated agents, to avoid brittle coupling of agent and benchmark.[3] The benchmark exposes Model Context Protocol (MCP) servers for agents, allowing them to observe and interact with the target systems and their environment. SREGYM provides the following interfaces as MCP servers:

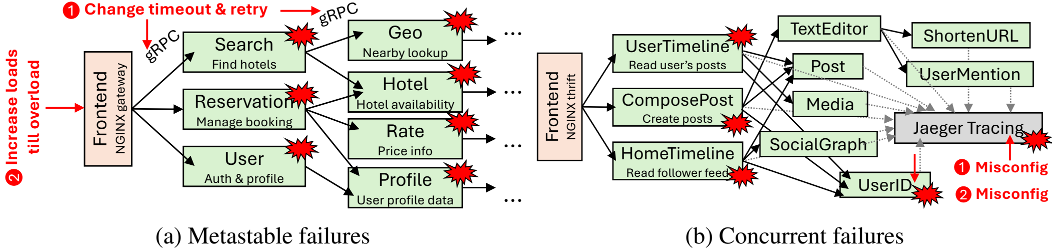
- **Metrics** for querying time-series performance metrics through Prometheus [10].
- **Logs** for searching and filtering container logs through Loki [8].
- **Traces** for inspecting request traces between system components via Jaeger [7].
- **Cluster control** for executing any commands to observe and change the system states. We currently support `kubectl` commands (Kubernetes' command-line interface).
- **Submission** for submitting diagnosis and mitigation results, which triggers evaluation.

For power users who want to design customized tools, SREGYM also exposes the raw observability endpoints and Kubernetes API endpoints for their agents to connect to.

2.4 Creating SRE Problems by Composing Faults

SREGYM offers 90 problems (with new ones being continuously added). The problems can be easily mutated, e.g., with different noise patterns. SREGYM currently provides 50 fault primitives that can be applied to 139 deployable services across 5 supported applications. With compatibility constraints (certain faults can only be applied to specific services), SREGYM offers 3,623 viable fault-component pairs, without composing noises and multiple faults, a roughly 40× multiplier over

[3] For example, AIOpsLab [32] decomposes each failure scenario into four isolated sub-problems (detection, localization, RCA, and mitigation) and scores them independently, but real-world failure handling is a single end-to-end loop in which earlier evidence and actions shape later endeavour; SREGYM instead evaluates the entire failure scenario holistically.



(a) Metastable failures (b) Concurrent failures

Figure 4: SRE problems that create scenarios with different failure modes in SREGym

90 scenarios (see Appendix C). This is a structural difference from benchmarks that ship as a fixed set of hardcoded scenarios: the curated 90 problems reflect what we have validated end-to-end, not the extent of what SREGym can express. Figure 3 shows the implementation of one SREGym problem.

Fault and Noise Simulation. Table 1 shows the faults and noises SREGym simulates and the simulation mechanisms. The faults include common ones like application misconfigurations and new ones SREGym implements (e.g., a tool that uses eBPF to simulate OS and hardware faults). More fault injectors can be directly integrated in SREGym. These faults are at different layers in the system stack and manifest via various symptoms. They require SRE agents to have comprehensive knowledge and understanding of different system components and their interactions. For example, faults injected into microservices require SRE agents to understand application logic and their dependencies [72, 73]. Faults injected into Kubernetes controllers and operators require SRE agents to understand how Kubernetes manages the cluster and applications [39, 69]. Lower-level OS and hardware faults require SRE agents to understand vertical interactions of applications, OSes, and hardware [40]. Each injector exposes a Python API; composing a compound failure reduces to invoking several injectors.

We define noises as transient, self-recovering disturbances (e.g., a pod crashing and then being rescheduled and a few dropped requests)—they are not failure root causes. SREGym injects these transient events with a configurable schedule. The agents see symptoms from both offending faults and noises and must distinguish the two types.

Failure Modes. SREGym enables developers to compose fault and noise injectors to create failure scenarios in different modes. Our current problem set covers three important failure modes.

- **Metastable failures** are self-sustaining congestive collapses in which the system degrades in response to transient events (e.g., a load surge) but fails to recover after the trigger is removed [29, 45, 47]. Metastable failures are known to be hard to diagnose, as they do not manifest in crashing behavior. Figure 4a shows a metastable failure problem we created, which (1) sets overly aggressive gRPC configurations for connection timeout (50ms) and retry count (30), (2) pushes the application (Hotel Reservation in DeathStarBench [37]) into a vulnerable state with a high load of 3000 requests per second, and (3) triggers the metastable failure by a transient CPU stress. In this way, all RPC requests are timed out and retried at the same time, causing a retry storm. SREGym's ability to coordinate different events is crucial to creating this problem.
- **Concurrent failures** are compound failures caused by multiple independent failures occurring simultaneously. Figure 4b depicts the problem where two faults are injected into the application (Social Network in DeathStarBench [37]): (1) a scheduler misconfiguration that makes an observability service pod unschedulable, and (2) a network port misconfiguration that fails the `UserID` service which further fails user requests. The problem evaluates whether SRE agents can understand and prioritize the port misconfiguration as it directly affects service availability (the scheduler misconfiguration only affects the observability service, which is invisible to end users).
- **Correlated failures** are failures in which multiple components fail at the same time because they share a common cause or dependency [36, 82]. The agent must recognize the correlations between the symptoms and connect them to the underlying root cause(s).

2.5 Evaluation Oracles

Designing oracles that fairly evaluate SRE agents is challenging. Existing SRE benchmarks use oracles that evaluate diagnosis results by requiring SRE agents to strictly match predefined labels (e.g., names of offending services and root causes) [32, 75]; such oracles are brittle because predefined labels are often ambiguous and not mutually exclusive; our experience shows that agents often report

correct results, but are incorrectly evaluated due to mismatching brittle labels. ITBench [50] proposed Normalized Topology-Aware Matching which requires fine-grained annotations of failure-propagation graphs. However, we find that such a level of laborious effort is error-prone and hard to scale.

Diagnosis Oracle. SREGYM adopts a checklist-based LLM-as-a-judge protocol [89] that decomposes diagnosis evaluation into fine-grained questions in a structured form, and produces stable verdicts across evaluators. We follow the principles of decomposing evaluation into multi-dimensional questions to improve inter-evaluator agreement [53], and grounding the rubric in domain-expert insight instead of LLM-generated criteria [35, 71]. Given a ground-truth root-cause description g and an agent-submitted diagnosis d, the diagnosis oracle assembles a prompt containing g, d, and a checklist of $N = 9$ Yes/No questions organized into $K = 3$ dimensions. An LLM evaluator returns, for each question q, a judgment $y_q \in \{0, 1\}$ together with supporting evidence and a confidence. For dimension k with a set of questions Q_k, the per-dimension score s_k is the fraction of affirmative answers: $s_k = \frac{1}{|Q_k|} \sum_{q \in Q_k} y_q$, and the *aggregated score* is a weighted sum $S = \sum_{k=1}^{K} w_k \, s_k$ (we give all dimensions equal weight, i.e., $w_k = \frac{1}{3}$). The final verdict is $\hat{v} = \mathbb{1}[S \geq \tau]$ with default threshold $\tau = \frac{7}{9}$ (which forbids a submission to pass while missing an entire dimension).

We find that decomposition into questions yields transparent, per-dimension evaluation of *where* the agent's understanding fell short. Our three dimensions are:

- **Fault localization**: Does the diagnosis identify the correct originating component, distinguishing it from downstream victims of cascading failures?
- **Fault characterization**: Does the diagnosis capture the faults and concrete details (e.g., the wrong port value, a specific environment variable)?
- **Failure scope**: Does the diagnosis avoid over- or under-attributing impact in terms of the components and symptoms?

Table 2: Pairwise inter-evaluator agreement.

Judge A	Judge B	Agree	κ
Sonnet-4.6	Human	**0.95**	**0.90**
Sonnet-4.6	GPT-5.4	0.88	0.76
Sonnet-4.6	K2.5	0.91	0.82
GPT-5.4	K2.5	**0.97**	**0.94**
GPT-5.4	Human	0.85	0.70
K2.5	Human	0.88	0.76

Table 2 validates the oracle's verdicts on a stratified sample of 100 agent diagnosis results, independently labelled by a domain expert and scored by two alternate LLM evaluators (GPT-5.4 and Kimi K2.5). The default oracle agrees with human experts at Cohen's $\kappa = 0.90$ (almost perfect agreement), while the two alternate LLM evaluators converge at $\kappa = 0.94$ and retain substantial agreement with human experts ($\kappa = 0.70$ and $= 0.76$ for for GPT-5.4 and Kimi K2.5). The convergence between LLM evaluators suggests that the decision depends primarily on checklist decomposition, not the choice of LLMs (a concern with single-model LLM-as-a-judge evaluation [89]).

Mitigation Oracle. The mitigation oracle is problem-specific to accurately reflect whether the target failure is truly mitigated. The oracle checks whether the target fault is resolved and whether the target system has recovered to a healthy state. The mitigation oracle uses both client-side observability such as user request success rate and system-side observability of application processes, Kubernetes cluster, etc. This allows us to evaluate the agent's mitigation results on complicated failure scenarios, such as metastable failures and low-level software/hardware failures.

3 Results

We use SREGYM to evaluate three AI agents powered by frontier LLMs: Stratus [31], a state-of-the-art SRE agent with two LLMs: Claude Sonnet-4.6 and Kimi-k2.5; and two coding agents: Claude Code [2] with Claude Sonnet-4.6 and Codex [5] with GPT-5.4 (gpt-5.4-2026-03-05). Each agent-model pair is evaluated with three runs per problem. We use Claude Sonnet-4.6 for the diagnosis oracle (see §2.5) consistently across the evaluation.

For noise simulation, we randomly simulate two noise patterns every five minutes, each one lasting two minutes. The noise simulation is done by the framework, not hardcoded in any problems.

Evaluation Metrics. We evaluate the agents on the success rates of diagnosis and mitigation tasks. Diagnosis success rate measures whether the agent pinpoints the root causes of target failures correctly. Mitigation success rate measures whether the agent successfully mitigates failures (verified

Table 3: Overall benchmark results on SREGYM. TTD and TTM are capped at the 1800-second agent timeout, with timed-out runs contributing the cap value, so failure cost is reflected in the mean rather than inducing survivorship bias. ■: runs with noise injected; □: runs without noise injected.

Agent	Model	Noise	Diag. (%) ↑	Mitig. (%) ↑	E2E (%) ↑	TTD (s) ↓	TTM (s) ↓	# Tokens ↓
Stratus	Sonnet-4.6	□	61.5%	**78.5%**	54.8%	**114.0**	771.1	812K
		■	51.5%	65.5%	40.2%	170.5	885.0	464K
	K2.5	□	41.3%	60.6%	32.9%	674.5	1348.8	**413K**
		■	38.9%	57.3%	30.4%	656.4	1283.2	443K
Claude Code	Sonnet-4.6	□	**72.6%**	75.6%	**60.7%**	292.5	702.0	1.47M
		■	62.6%	76.3%	53.7%	314.0	736.5	1.71M
Codex	GPT-5.4	□	70.0%	65.2%	53.3%	176.4	**376.0**	1.98M
		■	59.3%	64.0%	45.9%	218.1	397.7	1.88M

by the mitigation oracle). We also measure end-to-end (E2E) success rate, referring to cases where the agent achieves both correct diagnosis and correct mitigation on the same run. We also report Time-To-Diagnose (TTD), Time-To-Mitigate (TTM), and mean token usage per problem run.

3.1 Overall Benchmarking Results

Table 3 shows the overall results. SREGYM presents challenges to frontier AI agents and models, with diagnosis success rates ranging from 38.9% to 72.6% and mitigation success rates ranging from 57.3% to 78.5% across agent-model pairs. The SRE problems evaluate an agent's ability to reason about complex interactions between system components, infer root causes from symptoms, and effectively use tools to operate systems, which frontier agents/models still face difficulties with.

We find that agents show different characteristics in addressing SRE problems. Claude Code shows the highest end-to-end success rates, compared to Stratus and Codex. Stratus with Sonnet-4.6 has the highest mitigation success rate among all agents, attributed to its undo-and-retry mechanism [31]. On the other hand, Stratus with Kimi K2.5 has the lowest success rate, due to limited raw model capabilities.

Token Cost. Claude Code uses 3× more tokens per run than Stratus, and Codex uses 3.6× more. The reason is that coding agents are not optimized for processing the large volumes of observability data in SRE tasks. Stratus, as an SRE agent, preprocesses observability data and only prompts LLMs with relevant data.

Impact of Noises. As shown in Table 3, diagnosis success rate drops for *every* agent-model pair. Mitigation success is more robust than diagnosis under noises. Noises distract an agent's hypothesis about the root causes; on the other hand, agents tend to recover from incorrect hypotheses with self-validation (see §3.3). Inspecting trajectories, we observe that all the evaluated agents take a *greedy* approach (see Appendix D): they always treat the first plausible anomaly as the target failure (which can be a noise). In several cases, the agent did find evidence of the root cause of the target failure, but disregarded it because it was irrelevant to the noises they were (wrongly) targeting.

3.2 Results on New Failure Scenarios

Table 4 partitions SREGYM's problems by failure scenarios. We find that problems ported from existing benchmarks [32, 50] introduce limited challenges for the evaluated agents with strong models. For example, the mitigation success rate is above 80% for Stratus with Sonnet-4.6. Most of these problems are single-fault scenarios focusing on applications and do not include noises. SREGYM includes *new* problems that are built on similar failure scenarios in terms of fault families, but differ in applications and system components. As shown in Table 4, evaluated agents show similar results.

However, Table 4 shows that the agents perform significantly worse in new failure scenarios that are unique to SREGYM. The end-to-end success rates of Stratus with Sonnet-4.6, Claude Code, and Codex decrease from 63.7% to 17.9%, 60.8% to 28.2%, and 57.8% to 15.4%, respectively. These results show significant gaps in AI agents' ability to address high-fidelity failures such as those rooted

Table 4: Benchmark results partitioned into three problem types. "Ported" refers to problems directly ported from AIOpsLab/ITBench; "Similar Failures" are new problems in SREGYM that share failure patterns with "Ported"; "New failures" are faults and failure modes unique to SREGYM.

Agent	Noise	**Ported** (n=34)			**Similar Failures** (n=43)			**New Failures** (n=13)		
		Diag.	Mitig.	E2E	Diag.	Mitig.	E2E	Diag.	Mitig.	E2E
Stratus ✳	□	70.6%	**83.3%**	**63.7%**	62.8%	**80.6%**	58.9%	33.3%	59.0%	17.9%
	■	58.8%	68.6%	45.1%	55.0%	64.3%	44.2%	20.5%	30.8%	10.3%
Stratus K	□	42.2%	46.1%	27.5%	46.5%	44.2%	32.6%	15.4%	12.8%	10.3%
	■	39.2%	49.0%	27.5%	43.1%	42.3%	30.0%	17.9%	23.1%	12.8%
Claude Code	□	74.5%	71.6%	60.8%	**81.4%**	79.1%	**70.5%**	38.5%	74.4%	28.2%
	■	66.7%	73.5%	52.0%	62.8%	78.3%	56.6%	**51.3%**	**76.9%**	**48.7%**
Codex	□	**76.5%**	66.7%	57.8%	78.3%	68.2%	61.2%	25.6%	41.0%	15.4%
	■	66.7%	66.7%	51.0%	62.8%	68.2%	50.4%	28.2%	28.2%	17.9%

Table 5: Conditional mitigation (M) probability with diagnosis (D) outcome.

Agent	Noise	$P(\mathrm{M} \mid \mathrm{D})$	$P(\mathrm{M} \mid \neg\mathrm{D})$
Stratus ✳	□	0.880	0.588
Stratus ✳	■	0.738	0.506
Stratus K	□	0.690	0.351
Stratus K	■	0.708	0.396
Claude Code	□	0.798	0.500
Claude Code	■	0.834	0.571
Codex	□	0.734	0.435
Codex	■	0.773	0.431

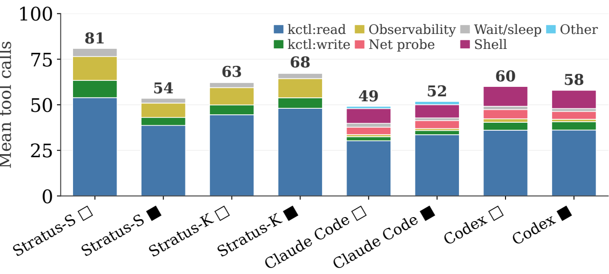


Figure 5: The average number of tool calls per SRE problem by category. S: Sonnet 4.6, K: Kimi K2.5; ■: noises.

in low-level stacks and/or caused by compound failures. We briefly describe two case studies, with more details in Appendix D.

Hardware faults. SREGYM's `latent_sector_error` problem injects intermittent errors on `read` system calls into a node. Across three runs of Stratus and Claude Code, no run produced an aggregated diagnosis score above 0.22, and fault characterization received a score of 0 in every run. All agents attribute the errors to error-handling logic inside the application. None of the runs proposed disk-level diagnostics (e.g., reading `dmesg` and inspecting `smartctl` output).

Metastable failures. The evaluated agents reliably diagnose the application-level trigger, which is visible in distributed traces and deployment configurations; however, no agent across the metastable failure problems identified *both* interacting components. In one run, Codex did locate the resource constraint but then dismissed the application trigger as a downstream artifact, producing a diagnosis that was correct on localization but wrong on scope (§2.5).

In general, we observe a lack of coherent, comprehensive understanding between the control/management plane of the cluster, the deployed systems, and the user requests. On metastable failures, agents did not connect the system-level constraints with the application-level trigger: this is required to identify how the system is induced into the metastable state. In hardware failures, agents also miss the hardware-software interactions. This lack of understanding limits the agent's ability to diagnose complex failures in SREGYM, which are arguably closer to real-world incidents.

3.3 Correlations between Diagnosis and Mitigation

When an agent succeeds at mitigation, was that success backed by a correct diagnosis, or did the agent stumble into a fix without knowing why? Table 5 shows conditional probabilities of mitigation given diagnosis outcome. Agents may pinpoint the root cause but fail in mitigation: in cases where the agent correctly diagnoses the failure, mitigation succeeds only 69%–88% of the time ($P(\mathrm{M} \mid \mathrm{D})$). When the initial diagnosis fails to address the root cause, mitigation success rate drops 23%–34% across agents. The drop shows that diagnosis is helpful in guiding the agent in its mitigation.

When the diagnosis is wrong, the agent may still successfully mitigate the failure by continuously observing and tuning the systems. Mitigation succeeds 35–59% of the time across agents when initial diagnosis is incorrect ($P(\text{M} \mid \neg\text{D})$). Note that the benchmark does not inform the agent whether its diagnosis is correct or not. We observe two common patterns: (1) the agent often pattern-matches a known symptom to mitigation actions that can successfully mitigate the failure without understanding its root cause; (2) the agent can correct its incorrect diagnosis by observing the persistence of the failure and continuously forming new hypotheses to make further attempts. For example, without noises injected, when the diagnosis is incorrect, Stratus with Sonnet-4.6 averages 3.82 attempts during mitigation, compared with 1.88 attempts with correct diagnosis (see Table 12).

3.4 Tool Usage

Figure 5 classifies tool calls in the agent trajectories (Appendix E shows more details). The majority (60–72%) of tool calls are read-only `kubectl` commands. Two commands account for around 87% of all read operations: `kubectl get` to inspect system state and `kubectl logs` to read container logs. The primary resources inspected are Pods [9], Deployments [4], and ConfigMaps [3]. Agents execute about 19–28 read commands before their first mitigation actions (see Table 9).

Agents differ in their mitigation actions. Stratus favors `kubectl patch` (39–41% of write actions) to change system state, while Claude Code and Codex prefer `kubectl rollout` (29–40% of write actions), which restarts deployments. Codex additionally relies on `kubectl run` (19–21%) to spin up ephemeral containers for in-cluster network testing and `kubectl port-forward` (11–12%) for direct service probing, which are tools Status and Claude Code rarely use.

Stratus has custom-built API tools for querying metrics, traces, and service dependency graphs, which account for 15–17% of its tool calls. Claude Code and Codex instead discover observability endpoints and construct raw `curl` commands; their observability access accounts for only 2–3% of tool calls.

4 Related Work

The advances of AI for SWE (Software Engineering) have pushed AI for SRE to be the next frontier. Recent work has made active progress on agentic SRE technologies, from root cause analysis (RCA) to failure mitigation [30, 31, 58, 59, 64, 70, 76, 83, 85, 86, 90]; meanwhile, many commercial and open-source SRE agent products are developed [11, 16–18]. Our communications with many SRE researchers and practitioners show that high-quality SRE benchmarks are highly desired.

A few benchmarks provide static datasets of Q&A [24, 55], observability data from real/synthetic sources [43, 48, 77], and system snapshots [75]. SREGYM can also be used to generate such datasets. These datasets are valuable for anomaly detection and RCA, but are fundamentally limited as they do not provide opportunities for agents to iteratively probe and observe the systems as in real-world diagnosis process. For the same reason, these benchmarks cannot support mitigation tasks.

The design of SREGYM comes from the reflection on existing live benchmarks [32, 50, 84] which are limited in their abilities to simulate high-fidelity, realistic failure scenarios, due to the lack of system support for orchestrating distributed events, low-level fault simulation and injection mechanisms, noise simulation, etc. Existing live benchmarks also suffer from engineering practices that misuse chaos engineering tools, lack protection against reward hacking, etc. (see Appendix B).

SREGYM focuses on failure analysis and mitigation and thus is complementary to other related benchmarks for deployment and regression testing [75] and terminal-based environment setup [60].

5 Discussion and Conclusion

The aspiration of SREGYM is to push the standard of AI for SRE and to unlock a whole new evolution of agentic SRE technologies. With this goal in mind, we develop SREGYM as a high-fidelity benchmark to represent existing challenges in real-world production environments and as a usable, extensible framework that can be continously evolved with new challenges for AI. Several components of SREGYM can be further enhanced and enriched, including more diverse noise modeling, more comprehensive fault simulation and failure modes, and system environments (e.g., to include edge presences)—some are being started. A clear next step is to upgrade SREGYM into a reinforcement-learning (RL) style training ground for SRE agents beyond its current problem set.

Acknowledgement

We thank everyone who has supported, helped, and contributed to SREGYM. Specifically, we thank all the contributors to SREGYM and all the users of SREGYM who give us encouragement and feedback. We thank Braden Hancock, Andy Konwinski, Brighten Godfrey, Sasa Misailovic, Xuan Feng, Lidong Zhou, and Darko Marinov for valuable feedback on the project. SREGYM is supported in part by a Slingshot grant from the Laude Institute and by NSF CNS-2145295.

References

[1] Chaos mesh: A powerful chaos engineering platform for kubernetes. `https://chaos-mesh.org`.

[2] Claude Code by Anthropic. `https://code.claude.com`.

[3] ConfigMaps. `https://kubernetes.io/docs/concepts/configuration/configmap`.

[4] Deployment. `https://kubernetes.io/docs/concepts/workloads/controllers/deployment`.

[5] Codex: Cloud coding agent. `https://chatgpt.com/codex`.

[6] Helm - The package manager for Kubernetes. `https://helm.sh`.

[7] Jaeger: open source, distributed tracing platform. `https://www.jaegertracing.io`.

[8] Loki - a horizontally-scalable, highly-available, multi-tenant log aggregation system inspired by Prometheus. `https://github.com/grafana/loki`.

[9] Pods. `https://kubernetes.io/docs/concepts/workloads/pods`.

[10] Open source metrics and monitoring for your systems and services. `https://prometheus.io`.

[11] TierZero - Agents that handle the incidents, alerts, and internal questions that fragment your team's day. `https://www.tierzero.ai/`.

[12] stress-ng - a tool to load and stress a computer system. `https://wiki.ubuntu.com/Kernel/Reference/stress-ng`, 2013.

[13] Chaosblade: An Easy to Use and Powerful Chaos Engineering Toolkit. `https://github.com/chaosblade-io/chaosblade`, 2019.

[14] Quantifying GitHub Copilot's Impact in the Enterprise with Accenture. `https://github.blog/news-insights/research/research-quantifying-github-copilots-impact-in-the-enterprise-with-accenture/`, May 2024.

[15] Otel-Demo - A microservice-based distributed system intended to illustrate the implementation of OpenTelemetry in a near real-world environment. `https://github.com/open-telemetry/opentelemetry-demo`, 2024.

[16] AWS DevOps Agent. `https://aws.amazon.com/devops-agent/`, 2025.

[17] Azure SRE Agent. `https://azure.microsoft.com/en-us/products/sre-agent`, 2025.

[18] Ciroos - Reduce toil, investigate incidents faster, and drive autonomous operations. `https://ciroos.ai/`, 2025.

[19] dm-dust - A Linux kernel module which can be used to simulate the bad blocks behavior on a physical disk. `https://docs.kernel.org/admin-guide/device-mapper/dm-dust.html`, 2025.

[20] 2025 Stack Overflow Developer Survey. `https://survey.stackoverflow.co/2025/`, 2025.

[21] Amazon tightens code controls after outages, including one caused by AI. https://www.businessinsider.com/amazon-tightens-code-controls-after-outages-including-one-ai-2026-3, Mar. 2026.

[22] Demystifying Evals for AI Agents. https://www.anthropic.com/engineering/demystifying-evals-for-ai-agents, 2026.

[23] 43% of AI-generated code changes need debugging in production, survey finds. https://venturebeat.com/technology/43-of-ai-generated-code-changes-need-debugging-in-production-survey-finds, 2026.

[24] SRE-skills-bench: LLM Benchmark for SRE Tasks. https://github.com/Rootly-AI-Labs/SRE-skills-bench, 2026.

[25] A. Alquraan, H. Takruri, M. Alfatafta, and S. Al-Kiswany. An Analysis of Network-Partitioning Failures in Cloud Systems. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI'18)*, Oct. 2018.

[26] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Transactions on Dependable and Secure Computing (TDSC)*, 1(1):1–23, Jan. 2004.

[27] L. N. Bairavasundaram, G. R. Goodson, S. Pasupathy, and J. Schindler. An Analysis of Latent Sector Errors in Disk Drives. In *Proceedings of the 2007 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'07)*, June 2007.

[28] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using Magpie for Request Extraction and Workload Modelling. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI'04)*, Dec. 2004.

[29] N. Bronson, A. Aghayev, A. Charapko, and T. Zhu. Metastable Failures in Distributed Systems. In *Proceedings of the 18th Workshop on Hot Topics in Operating Systems (HotOS'21)*, June 2021.

[30] 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'24)*, Apr. 2024.

[31] Y. Chen, J. Pan, J. Clark, Y. Su, N. Zheutlin, B. Bhavya, R. Arora, Y. Deng, S. Jha, and T. Xu. Stratus: A Multi-agent System for Autonomous Reliability Engineering of Modern Clouds. In *Proceedings of The 39th Annual Conference on Neural Information Processing Systems (NeurIPS'25)*, Dec. 2025.

[32] 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. In *Proceedings of the 8th Conference on Machine Learning and Systems (MLSys'25)*, May 2025.

[33] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An Empirical Study of Operating Systems Errors. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP'01)*, Oct. 2001.

[34] M. Chow, Y. Wang, W. Wang, A. Hailu, R. Bopardikar, B. Zhang, J. Qu, D. Meisner, S. Sonawane, Y. Zhang, R. Paim, M. Ward, I. Huang, M. McNally, D. Hodges, Z. Farkas, C. Gocmen, E. Huang, and C. Tang. ServiceLab: Preventing Tiny Performance Regressions at Hyperscale through Pre-Production Testing. In *Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI'24)*, July 2024.

[35] S. Y. Chu, J. W. Kim, and M. Y. Yi. Think Together and Work Better: Combining Humans' and LLMs' Think-Aloud Outcomes for Effective Text Evaluation. In *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems (CHI'25)*, Apr. 2025.

[36] D. Ford, F. Labelle, F. I. Popovici, M. Stokely, V.-A. Truong, L. Barroso, C. Grimes, and S. Quinlan. Availability in Globally Distributed Storage Systems. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI'10)*, Oct. 2010.

[37] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, K. Hu, M. Pancholi, Y. He, B. Clancy, C. Colen, F. Wen, C. Leung, S. Wang, L. Zaruvinsky, M. Espinosa, R. Lin, Z. Liu, J. Padilla, and C. Delimitrou. An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'19)*, Apr. 2019.

[38] J. T. Gu, X. Sun, W. Zhang, Y. Jiang, C. Wang, M. Vaziri, O. Legunsen, and T. Xu. Acto: Automatic End-to-End Testing for Operation Correctness of Cloud System Management. In *Proceedings of the 29th ACM Symposium on Operating Systems Principles (SOSP'23)*, Oct. 2023.

[39] J. T. Gu, Z. Tang, Y. Su, B. A. Stoica, X. Sun, W. X. Zheng, Y. Zhang, A. Rahman, C. Wang, and T. Xu. Who Watches the Watchers? On the Reliability of Softwarizing Cloud Application Management. In *Proceedings of the 23rd USENIX Symposium on Networked Systems Design and Implementation (NSDI'26)*, May 2026.

[40] H. S. Gunawi, C. Rubio-González, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and B. Liblit. EIO: Error Handling is Occasionally Correct. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST'08)*, Feb. 2008.

[41] H. S. Gunawi, M. Hao, R. O. Suminto, A. Laksono, A. D. Satria, J. Adityatama, and K. J. Eliazar. Why Does the Cloud Stop Computing? Lessons from Hundreds of Service Outages. In *Proceedings of the 7th ACM Symposium on Cloud Computing (SoCC'16)*, Oct. 2016.

[42] H. S. Gunawi, R. O. Suminto, R. Sears, C. Golliher, S. Sundararaman, X. Lin, T. Emami, W. Sheng, N. Bidokhti, C. McCaffrey, G. Grider, P. M. Fields, K. Harms, R. B. Ross, A. Jacobson, R. Ricci, K. Webb, P. Alvaro, H. B. Runesha, M. Hao, and H. Li. Fail-Slow at Scale: Evidence of Hardware Performance Faults in Large Production Systems. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST'18)*, Feb. 2018.

[43] S. Han, X. Hu, H. Huang, M. Jiang, and Y. Zhao. ADBench: Anomaly Detection Benchmark. In *Proceedings of the 36th Annual Conference on Neural Information Processing Systems (NeurIPS'22)*, Nov. 2022.

[44] P. H. Hochschild, P. Turner, J. C. Mogul, R. Govindaraju, P. Ranganathan, D. E. Culler, and A. Vahdat. Cores that don't count. In *Proceedings of the ACM SIGOPS 21st Workshop on Hot Topics in Operating Systems (HotOS'21)*, June 2021.

[45] L. Huang, M. Magnusson, A. B. 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)*, July 2022.

[46] 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 (HotOS'17)*, pages 150–155, May 2017.

[47] R. Isaacs, P. Alvaro, R. Majumdar, K.-K. Muniswamy-Reddy, M. Salamati, and S. Soudjani. Analyzing Metastable Failures. In *Proceedings of the ACM SIGOPS 20th Workshop on Hot Topics in Operating Systems (HotOS'25)*, May 2025.

[48] V. Jacob, F. Song, A. Stiegler, B. Rad, Y. Diao, and N. Tatbul. Exathlon: a Benchmark for Explainable Anomaly Detection Over Time Series. *Proceedings of the VLDB Endowment (VLDB'21)*, 14(11):2613–2626, July 2021.

[49] S. Jha, S. Cui, S. Banerjee, T. Xu, J. Enos, M. Showerman, Z. T. Kalbarczyk, and R. K. Iyer. Live Forensics for HPC Systems: A Case Study on Distributed Storage Systems. In *Proceedings of the International Conference for High-Performance Computing, Networking, Storage and Analysis (SC'20)*, Nov. 2020.

[50] S. Jha, R. 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, J. wook Ahn, D. Kar, 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. In *Proceedings of the 42nd International Conference on Machine Learning (ICML'25)*, May 2025.

[51] J. Kaldor, J. Mace, M. Bejda, E. Gao, W. Kuropatwa, J. O'Neill, K. W. Ong, B. Schaller, P. Shan, B. Viscomi, V. Venkataraman, K. Veeraraghavan, and Y. J. Song. Canopy: An End-to-End Performance Tracing and Analysis System. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP'17)*, Oct. 2017.

[52] J.-C. Laprie. Dependable Computing: Concepts, Limits, Challenges. In *Proceedings of the 25th IEEE International Symposium on Fault-Tolerant Computing (FTCS'95)*, June 1995.

[53] Y. Lee, J. Kim, J. Kim, H. Cho, J. Kang, P. Kang, and N. Kim. CheckEval: A Reliable LLM-as-a-Judge Framework for Evaluating Text Generation Using Checklists. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing (EMNLP'25)*, Nov. 2025.

[54] H. Liu, S. Lu, M. Musuvathi, and S. Nath. What Bugs Cause Production Cloud Incidents? In *Proceedings of the 17th Workshop on Hot Topics in Operating Systems (HotOS'19)*, May 2019.

[55] Y. Liu, C. Pei, L. Xu, B. Chen, M. Sun, Z. Zhang, Y. Sun, S. Zhang, K. Wang, H. Zhang, J. Li, G. Xie, X. Wen, X. Nie, M. Ma, and D. Pei. OpsEval: A Comprehensive IT Operations Benchmark Suite for Large Language Models. *arXiv:2310.07637*, June 2025.

[56] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis. Heracles: Improving Resource Efficiency at Scale. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA'15)*, June 2015.

[57] D. Loker. AI vs Human Code Gen Report: AI Code Creates 1.7x More Issues. `https://www.coderabbit.ai/blog/state-of-ai-vs-human-code-generation-report`, 2026.

[58] Y. Luo, J. Jiang, J. Feng, L. Tao, Q. Zhang, X. Wen, Y. Sun, S. Zhang, and D. Pei. From Observability Data to Diagnosis: An Evolving Multi-agent System for Incident Management in Cloud Systems. *arXiv:2510.24145*, Nov. 2025.

[59] J. Mao, L. Li, Y. Gao, Z. Peng, S. He, C. Zhang, S. Qin, S. Khalid, Q. Lin, S. Rajmohan, et al. StepFly: Agentic Troubleshooting Guide Automation for Incident Diagnosis. *arXiv:2510.10074*, Apr. 2026.

[60] M. A. Merrill, A. G. Shaw, N. Carlini, B. Li, H. Raj, I. Bercovich, L. Shi, J. Y. Shin, T. Walshe, E. K. Buchanan, J. Shen, G. Ye, H. Lin, J. Poulos, M. Wang, M. Nezhurina, J. Jitsev, D. Lu, O. M. Mastromichalakis, Z. Xu, Z. Chen, Y. Liu, R. Zhang, L. L. Chen, A. Kashyap, J.-L. Uslu, J. Li, J. Wu, M. Yan, S. Bian, V. Sharma, K. Sun, S. Dillmann, A. Anand, A. Lanpouthakoun, B. Koopah, C. Hu, E. Guha, G. H. S. Dreiman, J. Zhu, K. Krauth, L. Zhong, N. Muennighoff, R. Amanfu, S. Tan, S. Pimpalgaonkar, T. Aggarwal, X. Lin, X. Lan, X. Zhao, Y. Liang, Y. Wang, Z. Wang, C. Zhou, D. Heineman, H. Liu, H. Trivedi, J. Yang, J. Lin, M. Shetty, M. Yang, N. Omi, N. Raoof, S. Li, T. Y. Zhuo, W. Lin, Y. Dai, Y. Wang, W. Chai, S. Zhou, D. Wahdany, Z. She, J. Hu, Z. Dong, Y. Zhu, S. Cui, A. Saiyed, A. Kolbeinsson, J. Hu, C. M. Rytting, R. Marten, Y. Wang, A. Dimakis, A. Konwinski, and L. Schmidt. Terminal-Bench: Benchmarking Agents on Hard, Realistic Tasks in Command Line Interfaces. *arXiv:2601.11868*, Jan. 2026.

[61] J. J. Meza, T. Gowda, A. Eid, T. Ijaware, D. Chernyshev, Y. Yu, M. N. Uddin, R. Das, C. Nachiappan, S. Tran, S. Shi, T. Luo, D. K. Hong, S. Panneerselvam, H. Ragas, S. Manavski, W. Wang, and F. Richard. Defcon: Preventing Overload with Graceful Feature Degradation. In *Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI'23)*, July 2023.

[62] N. Palix, G. Thomas, S. Saha, C. Calvès, J. Lawall, and G. Muller. Faults in Linux: Ten Years Later. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'11)*, Mar. 2011.

[63] T. Patel and D. Tiwari. CLITE: Efficient and QoS-Aware Co-Location of Multiple Latency-Critical Jobs for Warehouse Scale Computers. In *Proceedings of the 2020 IEEE International Symposium on High Performance Computer Architecture (HPCA'20)*, Feb. 2020.

[64] C. Pei, Z. Wang, F. Liu, Z. Li, Y. Liu, X. He, R. Kang, T. Zhang, J. Chen, J. Li, G. Xie, and D. Pei. Flow-of-Action: SOP Enhanced LLM-Based Multi-Agent System for Root Cause Analysis. In *Companion Proceedings of the ACM on Web Conference 2025 (WWW'25)*, May 2025.

[65] T. Pelkonen, S. Franklin, J. Teller, P. Cavallaro, Q. Huang, J. Meza, and K. Veeraraghavan. Gorilla: a Fast, Scalable, In-memory Time Series Database. *Proceedings of the VLDB Endowment (VLDB'15)*, 8(12):1816–1827, Aug. 2015.

[66] B. Schroeder, S. Damouras, and P. Gill. Understanding Latent Sector Errors and How to Protect against Them. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies (FAST'10)*, Feb. 2010.

[67] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag. Dapper, a Large-Scale Distributed Systems Tracing Infrastructure. Technical Report dapper-2010-1, Google, Inc., Apr. 2010.

[68] X. Sun, R. Cheng, J. Chen, E. Ang, O. Legunsen, and T. Xu. Testing Configuration Changes in Context to Prevent Production Failures. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)*, Nov. 2020.

[69] X. Sun, W. Luo, J. T. Gu, A. Ganesan, R. Alagappan, M. Gasch, L. Suresh, and T. Xu. Automatic Reliability Testing for Cluster Management Controllers. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI'22)*, July 2022.

[70] P. Tang, S. Tang, H. Pu, Z. Miao, and Z. Wang. MicroRCA-Agent: Microservice Root Cause Analysis Method Based on Large Language Model Agents. *arXiv:2509.15635*, Sept. 2025.

[71] Y. Tian, Y. Liu, Z. Chong, Z. Huang, and H.-A. Jacobsen. GALA: Can Graph-Augmented Large Language Model Agentic Workflows Elevate Root Cause Analysis? *arXiv:2508.12472*, Aug. 2025.

[72] B. Treynor, M. Dahlin, V. Rau, and B. Beyer. The Calculus of Service Availability. *Communications of the ACM (CACM)*, 60(9):42–47, Aug. 2017.

[73] K. Veeraraghavan, J. Meza, S. Michelson, S. Panneerselvam, A. Gyori, D. Chou, S. Margulis, D. Obenshain, S. Padmanabha, A. Shah, Y. J. Song, and T. Xu. Maelstrom: Mitigating Datacenter-level Disasters by Draining Interdependent Traffic Safely and Efficiently. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI'18)*, Oct. 2018.

[74] H. Wang, Q. Mang, A. Cheung, K. Sen, and D. Song. How We Broke Top AI Agent Benchmarks: And What Comes Next. `https://rdi.berkeley.edu/blog/trustworthy-benchmarks-cont/`, 2026.

[75] Y. Wang, G. Yu, H. Huang, Z. Wang, Y. Huang, P. Chen, and M. R. Lyu. Cloud-OpsBench: A Reproducible Benchmark for Agentic Root Cause Analysis in Cloud Systems. *arXiv:2603.00468*, Feb. 2026.

[76] 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'24)*, Oct. 2024.

[77] 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 *Proceedings of the 13th International Conference on Learning Representations (ICLR'25)*, Jan. 2025.

[78] T. Xu, J. Zhang, P. Huang, J. Zheng, T. Sheng, D. Yuan, Y. Zhou, and S. Pasupathy. Do Not Blame Users for Misconfigurations. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP'13)*, Nov. 2013.

[79] T. Xu, X. Jin, P. Huang, Y. Zhou, S. Lu, L. Jin, and S. Pasupathy. Early Detection of Configuration Errors to Reduce Failure Damage. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*, Nov. 2016.

[80] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, and Y. Cao. ReAct: Synergizing Reasoning and Acting in Language Models. In *Proceedings of the 11th International Conference on Learning Representations (ICLR '23)*, May 2023.

[81] D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou, and S. Pasupathy. SherLog: Error Diagnosis by Connecting Clues from Run-Time Logs. In *Proceedings of the 15th International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS'10)*, Mar. 2010.

[82] E. Zhai, A. Chen, R. Piskac, M. Balakrishnan, B. Tian, B. Song, and H. Zhang. Check before You Change: Preventing Correlated Failures in Service Updates. In *Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI'20)*, Feb. 2020.

[83] L. Zhang, T. Jia, K. Wang, W. Hong, C. Duan, M. He, and Y. Li. Adaptive Root Cause Localization for Microservice Systems with Multi-Agent Recursion-of-Thought. *arXiv:2508.20370*, Aug. 2025.

[84] L. Zhang, Y. Zhai, T. Jia, C. Duan, M. He, L. Pan, Z. Liu, B. Ding, and Y. Li. MicroRemed: Benchmarking LLMs in Microservices Remediation. *arXiv:2511.01166*, Nov. 2025.

[85] W. Zhang, H. Guo, J. Yang, Z. Tian, Y. Zhang, Y. Chaoran, Z. Li, T. Li, X. Shi, L. Zheng, and B. Zhang. mABC: Multi-Agent Blockchain-inspired Collaboration for Root Cause Analysis in Micro-Services Architecture. In *Findings of the Association for Computational Linguistics (EMNLP'24)*, Nov. 2024.

[86] 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 (TSC)*, 18(6):4221–4233, Nov. 2025.

[87] Y. Zhang, K. Rodrigues, Y. Luo, M. Stumm, and D. Yuan. The Inflection Point Hypothesis: A Principled Debugging Approach for Locating the Root Cause of a Failure. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP'19)*, Oct. 2019.

[88] Y. Zhang, U. Paul, M. d'Amorim, and A. Rahman. Configuration Defects in Kubernetes. *arXiv:2512.05062*, Dec. 2025.

[89] L. Zheng, W.-L. Chiang, Y. Sheng, S. Zhuang, Z. Wu, Y. Zhuang, Z. Lin, Z. Li, D. Li, E. Xing, H. Zhang, J. E. Gonzalez, and I. Stoica. Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena. In *Proceedings of the 37th Annual Conference on Neural Information Processing Systems (NeurIPS'23)*, Dec. 2023.

[90] Z. Zhong, R. Fu, M. Ma, S. Zhang, Y. Sun, C. Bansal, and D. Pei. LLM-Enhanced Failure Localization in Microservices: Integrating Multi-Modal Data and Expert Interpretation. *IEEE Transactions on Services Computing (TSC)*, pages 1–14, Mar. 2026.

[91] X. Zhou, X. Peng, T. Xie, J. Sun, C. Xu, C. Ji, and W. Zhao. Benchmarking Microservice Systems for Software Engineering Research. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings (ICSE '18)*, May 2018.

[92] Y. Zhu, T. Jin, Y. Pruksachatkun, A. Zhang, S. Liu, S. Cui, S. Kapoor, S. Longpre, K. Meng, R. Weiss, F. Barez, R. Gupta, J. Dhamala, J. Merizian, M. Giulianelli, H. Coppock, C. Ududec, J. Sekhon, J. Steinhardt, A. Kellermann, S. Schwettmann, M. Zaharia, I. Stoica, P. Liang, and D. Kang. Establishing Best Practices for Building Rigorous Agentic Benchmarks. *arXiv:2507.02825*, July 2025.

A Diagnosis Evaluation Checklist

Table 6 lists the full checklist used by the diagnosis oracle (§2.5). Each question requires a Yes/No answer; the LLM evaluator returns supporting evidence and a confidence (High/Medium/Low) per question. Dimension weights and the pass threshold are set in a YAML file (current default: $w = \frac{1}{3}$).

Table 6: Full diagnosis evaluation checklist. Each question is answered Yes/No by the LLM evaluator. The per-dimension score formula and the aggregated score formula can be found in §2.5.

Dimension	ID	Question	Evaluator Hint
Fault Localization ($w = \frac{1}{3}$)	D1-Q1	Does the diagnosis name the same service, deployment, pod, node, or infrastructure component that the ground-truth identifies as the fault origin?	Compare against the target component and target resource type in the fault specification (spec).
	D1-Q2	Does the diagnosis correctly distinguish the fault origin from any secondary or cascading failure points mentioned in the ground-truth?	Check that the diagnosis points to the root-cause component, not a downstream victim.
	D1-Q3	Does the diagnosis avoid misidentifying a healthy component as the fault origin?	Verify the diagnosed component matches the fault spec's target component.
Fault Characterization ($w = \frac{1}{3}$)	D2-Q1	Does the diagnosis identify the same injected mechanism described in the ground-truth (e.g., wrong network port, missing environment variables, wrong container image, wrong selector, and memory limit)?	Match against the fault mechanism and injector method in the structured spec.
	D2-Q2	Does the diagnosis include concrete mutated details from the injection logic (e.g., environment variable, configuration value, network port, selector, container image tag, and resource limit)?	Compare concrete claims against parameters and the target mutation implied by the injector method.
	D2-Q3	Does the diagnosis avoid attributing the fault to an incorrect or unrelated fault type?	Check that the diagnosis does not conflict with the problem class, injector method, or injected parameter values.
Scope Precision ($w = \frac{1}{3}$)	D3-Q1	Does the diagnosis avoid blaming components that are not identified in the ground-truth as contributing to the fault?	Check for over-attribution: the diagnosis should not blame uninvolved components.
	D3-Q2	Does the diagnosis include all components listed in the ground-truth as contributing to or affected by the fault?	Check for under-attribution: all ground-truth components should be pointed out.
	D3-Q3	Does the diagnosis correctly describe the impact or symptom consistent with what the ground-truth states?	Compare stated impact against mechanism, parameters, and target component in the fault spec.

B Engineering Practices

While building SREGYM, we made several engineering decisions that we believe are important for any benchmark aiming to evaluate autonomous SRE agents. We document them as recommendations for future SRE benchmark developers, with the rationale behind each and, where useful, examples of departures we observed in existing benchmarks [32, 50, 75, 84].

Benefits of live environments. A natural design is to capture environment telemetry at one moment and present it to the agents as a static artifact, motivated by a desire to remove nondeterminism from

evaluation [75]. However, static artifacts are fundamentally limited. Noise and nondeterminism are intrinsic properties of the production environments that SRE agents must operate in, where signals are often partial, telemetry could race with the failures, and the environment state evolves while the agents are taking actions. Snapshots reduce the task to static log triage and eliminate the operational skills the benchmark claims to measure: forming hypotheses, issuing probes, and observing how the live environment responds. A snapshot also collapses the interactive nature of troubleshooting. Real incident response is iterative and time-ordered: an SRE engineer/agent issues a command, waits for it to take effect, watches the system's state, and decides the next step accordingly. SREGYM therefore adopts a live system environment, so the agent is evaluated against noisy, evolving environments mirroring production systems.

No restriction on the agent architecture. The space of SRE agents includes multi-agent systems with domain-specific tools (e.g., Stratus [31]) as well as general-purpose coding agents (e.g., Claude Code and Codex). AIOpsLab [32], for example, requires the evaluated agent to interact in a ReAct [80] loop mediated by an orchestrator that parses every action and exposes a fixed set of function signatures. Agents not built in the ReAct architecture would need to be ported or integrated to be evaluated on AIOpsLab. SREGYM keeps its agent interface minimal: only the `submit()` call is required. This decoupling lets us evaluate the same set of problems against architectures as different as Stratus, Claude Code, and Codex with the same benchmark framework.

Programmable runtime for synchronizing and coordinating events. The benchmark runtime must be expressive enough to schedule and synchronize events across injectors, oracles, and the agent's actions. Two classes of SREGYM problems make this requirement concrete. Metastable failures (see §D.1) pair a self-sustaining application trigger (e.g., a misconfigured `GOGC` value or a retry-storm configuration) with an infrastructure constraint (e.g., a tight namespace memory quota); the two components must be injected with the right timing and ordering to drive the system into the metastable state, and the mitigation oracle must continuously observe the environment to distinguish a transient recovery from a relapse. Noise simulation requires a separate concurrent loop that injects transient events on its own schedule while the target fault is active, so the agent sees a contemporaneous mix of distractor and target evidence. Benchmarks built on substrates like Ansible Playbooks as in ITBench [50] cannot express these coordination patterns directly and tend to defer the missing functionality to *ad hoc* shell scripts, which makes timing-sensitive faults and concurrent noises hard to control and to reproduce. SREGYM runs as a Python service that owns fault scheduling, noise scheduling, and oracle probing in a single process, so problem developers can express timing-sensitive behavior in the problem definition rather than by external, indirect scripts.

Avoiding misuses of chaos-engineering tools. Chaos engineering tools such as Chaos Mesh [1] and Chaosblade [13] are valuable for their intended use cases: perturbing a running system to test application resiliency against unexpected failures. By design, these tools inject *symptoms* (killed pods, dropped packets, throttled CPUs, latency spikes) rather than *defects*. There are no underlying faults for an SRE agent to discover, and the only valid "mitigation" is to stop the chaos injectors, which is *not* an operational skill an agent should be rewarded for learning. Several recent benchmarks conflate resiliency testing with operational evaluation: MicroRemed [84] sources its faults entirely from Chaos Mesh; Cloud-OpsBench [75] draws on Chaosblade alongside its Kubernetes misconfigurations; ITBench [50], which otherwise injects faults via direct Kubernetes manipulation, wraps a Chaos Mesh schedule for roughly 16% of its SRE scenarios (6 of 36). For those scenarios, there is no defect for the agent to fix, and the only action that "mitigates" the problem is stopping the chaos-engineering tools or deleting their schedule. SREGYM injects faults via direct Kubernetes manipulation, syscall-level eBPF probes, and operator misoperation rather than symptomatic perturbations, so every SRE problem has underlying defects for the agent to diagnose and resolve.

Protection against reward hacking. AI agents can exploit benchmark infrastructure to inflate scores without solving the underlying tasks [74]. In SRE benchmarks, the most direct manifestation is an agent that discovers and disables the fault-injection services rather than reasoning about the actual faults. Neither AIOpsLab [32] nor ITBench [50] protects against such reward hacking properly: their fault injectors run as identifiable pods in the same environment the agent inspects. A second exploit is treating alert-clearing as the success signal: the Stratus paper [31] reports that 8 of 18 ITBench mitigation problems (44%) can be "solved" by a generic pod-restart loop, where the fault injector loses track of the pod after it is restarted, the alert clears, and the agent is credited with a successful mitigation (despite taking no action on the actual defects). SREGYM hides its fault-injection plane

behind a proxy that evaluated agents have no visibility into, and uses state-based mitigation oracles that probe live environment health rather than relying on alert suppression.

C Combinatorial Coverage Details

We provide the detailed breakdown behind the combinatorial coverage reported in the preamble of §2.4. Table 7 groups SREGYM's fault primitives by the classes of target components (referred to as "target" for short) they are compatible with, and reports the number of viable (fault, target) pairs per class across the 139-pod injection-target space (plus three worker nodes for hardware-level faults).

Table 7: Fault-target compatibility classes. "# Faults" refers to the number of fault primitives in each class; since one fault primitive can be compatible with multiple target classes (e.g., a missing environment variable affects both MongoDB and non-MongoDB pods), the column does not sum to the primitive total. "Pairs" is the count of viable (fault, target) combinations, totaling 3,623.

Class	**# Faults**	**Compatible targets**	**Pairs**
Universal Kubernetes-level	25	139 pods	3,475
Storage-dependent	5	6 PVC-mounted pods	30
DaemonSet-level	1	3 daemonsets	3
Operator-level	6	5 Kubernetes applications	30
MongoDB-specific	4	18 MongoDB pods	72
Valkey-specific	2	1 pod	2
App-layer misconfiguration	1	2 services	2
Node/kernel	3	3 worker nodes	9
Total			**3,623**

The 90 curated problems we evaluate against in §3 exercise only 2.5% of the 3,623 viable (fault, target) pairs. The noise simulation and multi-fault composition further multiply the combinations. The same combinatorial structure positions SREGYM as a natural environment for reinforcement-learning rollouts, where each (fault, target) pair becomes an episode against a live, production-like system environment (§5).

D Case Studies

D.1 Metastable Failure

SREGYM includes problems that model metastable failures, where a temporary trigger pushes the system into a self-sustaining degraded state that persists even after the trigger is removed [29, 45, 47]. Each problem is a *compound fault*: an application-level trigger (e.g., a misconfigured retry policy that amplifies traffic, or a runtime flag that forces frequent garbage collection) paired with an infrastructure constraint that drives the system into a vulnerable state (e.g., a resource quota or limit that caps CPU or memory allocation). The constraint produces no errors and no failed traces; it is discoverable only through explicit inspection of deployment configuration. The trigger drives the system from the vulnerable state to the self-sustaining metastable state: system performance degrades with no explicit fail-stop symptoms. Only fixing the trigger would not mitigate the symptoms: the agent must reason about the relationship between the trigger and the sustained degraded state, then mitigate the problem by removing the infrastructure-level constraint and restarting the application, giving it a clean slate.

In our evaluation, agents reliably diagnosed the application-level trigger through trace analysis and deployment inspection, but almost never discovered the infrastructure constraint because it produces no observable symptom. In the one run where Codex *did* find the infrastructure resource constraint, it attributed the entire failure to it and dismissed the application-level misconfiguration. In no case did the agent identify the interaction between the trigger and the constraint.

The agents are observed to fix the metastable behavior at times. Agents fixed the misconfiguration and restarted the affected services; this restores the system and cleans up the compounded traffic, so the system can resume normal execution. When the agent fails to mitigate the metastable behavior,

it is often because the agent is distracted by surface-level symptoms and attributes the failure to an entirely different fault type. For example, in one run of the gc_capacity_degradation problem, where an aggressively low GOGC setting forces frequent garbage collection across all workloads in a capacity-constrained namespace, the agent never inspected GOGC at all. Instead, the agent fixated on distributed traces showing a $\sim$300 ms gap between the frontend and the profile-service, while the server-side GetProfiles handler completed in microseconds. From this pattern alone, the agent concluded that the delay must live in the network path, and submitted a root cause of a tc netem rule injecting $\sim$150 ms of latency in each direction on the profile-service pod's veth interface. This incorrect diagnosis makes two independent errors: (1) the fault type is network latency injection, and (2) the scope is a single service. As a result, the agent proposed mitigations targeting a nonexistent network rule, and never restarted the workloads to clear the metastable garbage-collection loop.

D.2 Hardware Fault

The latent_sector_error problem in SREGYM injects hardware faults into the storage devices of a physical node. A storage disk would return intermittent errors on file-reading system calls (e.g., pread()). The MongoDB databases deployed on that node would crash with "read: input/output error," as they are unable to read storage-engine-related metadata files.

Our evaluation shows that agents struggle to diagnose the hardware-related root cause. Across three Stratus, three Claude Code, and three Codex runs with no noise simulation, none produced a diagnosis score above 0.22, and D2 (fault characterization) scored 0 in every run, showing that their diagnosis never reached the correct component of the system. For example, the agents observe that the I/O errors come from the memcached deployment, which reads from the MongoDB deployment, and incorrectly attribute the failure to dropped connections from the memcached deployment. Agents also blame user workloads for overwhelming the memcached process, causing it to reset connections. Lastly, they blame the microservice application for not gracefully handling the I/O error. These conclusions never mention that the underlying hardware can be defective.

This problem exposes a specific weakness: there is a lack of understanding in how the underlying hardware can affect the application deployed atop it. Agents treat the exposed I/O error as evidence of incorrect error handling, but never suggest disk-level diagnostics that are standard SRE responses to hardware failures.

D.3 Greedy Approach in Diagnosing Failures

A recurring agent failure pattern across our evaluation is the agent taking a greedy approach in diagnosing and mitigating any first-observed anomaly in the environment. The agent first encounters a suspected but unrelated anomaly in the environment, and submits an immature diagnosis, which later misleads the mitigation phase. These suspected anomalies can be injected noises running alongside the target fault, or pre-existing application characteristics such as low memory limits in application containers.

We observe that the pattern is consistent regardless of the anomaly the agent picks up. Figure 1 shows an example using the missing_env_variable_astronomy_shop problem, where the actual fault is a dropped environment variable on the frontend component of the application. Across all runs of both Stratus and Claude Code, neither agent ever inspected the frontend component. Stratus latched onto another deployment with a low memory limit, while Claude Code focused on Jaeger tracing infrastructure. Both found plausible-looking issues, diagnosed them as the root cause, and never investigated the actual faulty component. This greedy approach affects mitigation effectiveness. After fixating on the memory limit, the agent raises it with kubectl patch or a similar edit, but clearly this fix does not address the offending fault.

A more interesting (or say pitiful) case occurs when the agent *does* encounter evidence of the offending fault but still anchors on its initial hypothesis. In the valkey_auth_disruption problem, the fault is an invalid password set on the Valkey database at runtime. The dependent microservice component crashes because it cannot authenticate to the database. Agents often attributed the crash to insufficient memory, noting that the containers had low memory limits. However, the agents also observed that the init container's TCP health check succeeded while the application-layer database connection failed. This is a discrepancy consistent with authentication failure, not resource exhaustion.

In one run, the agent mentioned the password-setting command (`requirepass`) multiple times as a candidate hypothesis but defaulted to the memory explanation.

This case study exposes a specific weakness: the agent treats the first plausible abnormality as a stopping criterion. Once a candidate root cause is written down, in the mitigation phase, the model interprets subsequent evidence as supporting it and does not generate competing hypotheses. A human SRE, in contrast, would be able to form multiple hypotheses and investigate concurrently, until the real root cause is found.

E Tool Usage Details

We provide a detailed analysis of the tool usage reported in §3.4. We classify every tool call in the agent trajectories into the following categories:

- **kubectl (read)** for read-only inspection of system state (e.g., `get`, `logs`, `describe`, `top`, `auth`).
- **kubectl (write)** for changing system state (e.g., `patch`, `rollout`, `set`, `run`, `delete`, `apply`, `scale`, `port-forward`).
- **Observability** for querying metrics, traces, and service maps. Stratus uses dedicated agent tools; Claude Code and Codex use `curl` to query Prometheus, Jaeger, or Loki endpoints.
- **Network** for connectivity probes (e.g., `curl` to application endpoints, `ss`, `netstat`, `nc`, `ping`).
- **Wait/sleep** for deliberate pausing (e.g., `wait_tool` for Stratus and `sleep` for others).
- **Shell** for general-purpose utilities (e.g., `cat`, `ls`, `grep`, `python`, `sed`, `export`).
- **Others** such as agent-framework-native tools (e.g., Claude Code's `Read`, `Agent`, `WebSearch`; Kimi's `thinking` tool).

Submission calls are excluded from all counts. Table 8 shows the tool call category breakdown per evaluation configuration. Table 9 shows the ratio of read to write commands in the system environment. Table 10 and Table 11 show the top subcommands executed in `kubectl`. All averages are computed as the mean across runs per problem, then the mean across problems.

Table 8: Tool call category breakdown.

Agent	Model	Noise	kctl(read)	kctl(write)	Observability	Network	Wait	Shell	Other
Stratus	Sonnet 4.6	□	65.8	11.9	16.6	0.0	5.6	0.0	0.0
		■	71.8	8.3	14.5	0.0	5.1	0.0	0.2
	K2.5	□	71.6	8.5	14.6	0.0	4.6	0.0	0.7
		■	72.0	8.1	14.8	0.0	4.4	0.0	0.6
Claude Code	Sonnet 4.6	□	61.5	4.6	2.4	8.3	4.2	16.5	2.5
		■	64.8	4.5	2.1	8.5	2.8	14.0	3.4
Codex	GPT-5.4	□	60.2	7.7	2.4	8.7	2.7	18.4	0.1
		■	61.6	7.6	2.7	7.4	2.7	17.8	0.2

Table 9: `kubectl` read/write analysis. Ratio is total reads divided by total writes. "Reads → First Write" is the mean number of read-only `kubectl` commands before the first mutation of system state, averaged per problem then across problems.

Agent	Model	Noise	Ratio	Reads → First Write
Stratus	Sonnet 4.6	□	5.5:1	22.7
		■	8.6:1	24.2
	K2.5	□	8.4:1	24.8
		■	8.9:1	27.6
Claude Code	Sonnet 4.6	□	13.5:1	21.3
		■	14.5:1	23.9
Codex	GPT-5.4	□	7.9:1	18.9
		■	8.1:1	21.2

Table 10: Top-3 `kubectl` read subcommands per agent (% of all read-only `kubectl` calls)

Agent	Model	Noise	1st	2nd	3rd
Stratus	Sonnet 4.6	□	get (64)	logs (18)	describe (13)
		■	get (65)	logs (17)	describe (14)
	K2.5	□	get (69)	logs (20)	describe (9)
		■	get (67)	logs (21)	describe (10)
Claude Code	Sonnet 4.6	□	get (59)	logs (23)	describe (10)
		■	get (62)	logs (19)	describe (11)
Codex	GPT-5.4	□	get (60)	logs (24)	exec (7)
		■	get (57)	logs (25)	exec (8)

Table 11: Top-5 `kubectl` write subcommands per agent (% of all mutating `kubectl` calls)

Agent	Model	Noise	1st	2nd	3rd	4th	5th
Stratus	Sonnet 4.6	□	patch (39)	set (19)	run (14)	delete (9)	rollout (8)
		■	patch (41)	set (23)	rollout (9)	delete (7)	create (6)
	K2.5	□	patch (41)	set (15)	create (13)	rollout (11)	delete (6)
		■	patch (40)	set (13)	delete (11)	create (10)	rollout (9)
Claude Code	Sonnet 4.6	□	rollout (35)	patch (32)	run (9)	delete (7)	apply (5)
		■	rollout (40)	patch (29)	delete (11)	run (7)	apply (4)
Codex	GPT-5.4	□	rollout (33)	patch (20)	run (19)	port-fwd (12)	delete (5)
		■	rollout (29)	run (21)	patch (17)	port-fwd (11)	delete (11)

Table 12: Mitigation retry analysis (Stratus variants only). The number of total mitigation attempts and mitigation reads are conditioned on whether the initial diagnosis was correct (D) or incorrect (¬ D).

			Attempts		Mitig. reads	
Agent	Model	Noise	D	¬ D	D	¬ D
Stratus	Sonnet 4.6	□	1.88	3.82	17.3	66.2
		■	1.55	1.61	13.2	21.6
	K2.5	□	1.79	1.90	19.3	26.4
		■	1.56	1.51	16.4	21.2

F Analysis of Total Tokens and End-to-End Success Rate

Figure 6 plots mean total tokens per run against end-to-end success rate for each (agent, model, noise) configuration. The plot separates two regimes. Stratus consumes 0.53M tokens per run on average (range 0.41–0.81M), while Claude Code uses 1.59M (3.0×) and Codex uses 1.93M (3.6×), consistent with Stratus's multi-agent architecture that preprocesses observability data and prompts the underlying LLM with only the filtered subset. The coding agents instead pull raw observability streams into the LLM context, which inflates token usage without a corresponding increase in end-to-end success.

Token spend does not predict success in this set. The highest end-to-end rate is achieved by Claude Code in the environment with no noise injected, but Stratus configurations land within a few points of it while spending a small fraction of the tokens. Codex spends the most tokens of any configuration and trails Claude Code in end-to-end success, indicating that on SREGym the marginal token does not buy additional success.

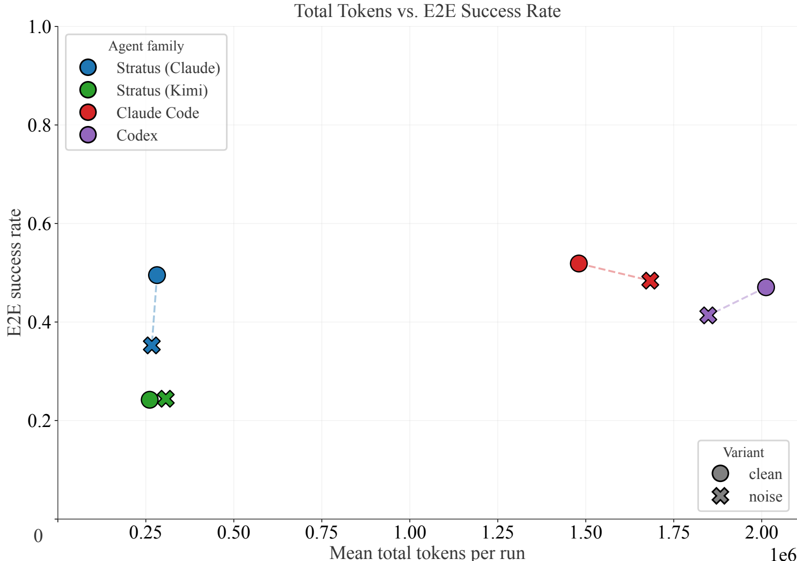


Figure 6: Mean total tokens per run versus end-to-end success rate ($P(\mathrm{D} \wedge \mathrm{M})$) for each (agent, model, noise) configuration. Circles denote the clean condition; crosses denote the noisy condition. Stratus configurations occupy a low-token / mid-success regime, while Claude Code and Codex occupy a high-token regime without a corresponding gain in end-to-end success.

Noise reduces end-to-end success rate across every agent (the noise marker for each color sits below its clean counterpart), but the token cost of noise is small. The coding agents do spend somewhat more tokens under noise as they ingest additional anomaly signals, while Stratus's token usage is essentially noise-invariant because its preprocessing layer absorbs the extra signals before they reach the LLM.

G Broader Impacts

SREGYM is a benchmark for evaluating AI agents on Site Reliability Engineering (SRE). We discuss the positive and negative societal impacts that we foresee from the work, along with mitigations.

Positive impacts. The most direct positive impact is improving the reliability of AI-assisted computing system operations. Production failures are a leading cause of service outages, financial loss, and engineer/operator burnout. A growing number of AI products are being marketed for autonomous SRE [11, 16–18]. Without realistic, mitigation-oriented benchmarks, claims about these technologies and products are difficult to verify. SREGYM is designed to evaluate AI agents' capability towards autonomous SRE. It injects faults across the full system stack (application, platform, OS, hardware), composes them with concurrent noises to model real-world production environments, and verifies mitigation against system state. By making it easier to evaluate SRE agents on realistic, high-fidelity SRE problems, we aim to raise the empirical bar that AI/agentic SRE products must clear, and to reduce the chance that under-tested agents are granted production access. A secondary positive impact is an open infrastructure for research. The composable APIs, fault and noise injectors, and MCP-based agent interface are released so that other researchers can extend SREGYM to new applications, new fault classes, and new agent architectures without rebuilding the scaffolding. In fact, we have heard from a number of colleagues who are using SREGYM in their research projects.

Negative impacts and mitigations. We considered three categories of potential negative impact. First, automation displacement: capable AI SRE agents could reduce demand for entry-level operation work, much like AI coding assistants are reshaping software development jobs. Our results, however, suggest this risk is currently distant: even frontier agents struggle on SREGYM's realistic failure scenarios (see §3 and §D). Second, misuse of fault-injection mechanisms: SREGYM's injectors could in principle be repurposed to cause harm in production systems that the user controls. We note that all of the injection primitives we use are standard, widely available tools; therefore, SREGYM does not introduce a novel attack capability beyond what these tools already expose. The benchmark

requires legitimate cluster credentials to operate, and mounting a real attack with these primitives is no easier than using the underlying tools directly. Third, over-reliance on benchmark numbers: a high SREGYM score is necessary but not sufficient evidence that an agent is safe to grant production access. Section H explicitly enumerates the assumptions and scope under which our scores should be interpreted, including applications smaller than production scale, a simplified noise model, and a small set of evaluated agent-model pairs. We encourage our users to read SREGYM results as a lower bound on the failure modes an agent can recover from, not an upper bound on the failure modes it will encounter in real-world production deployments.

H Limitations

SREGYM has limitations that shape how its results should be interpreted and extended.

Oracle Variance. The diagnosis oracle relies on an LLM evaluator, which introduces a source of variance that purely programmatic checks do not have. We mitigate this by constraining the LLM evaluator to a structured rubric with per-question answers (Appendix A) and by reporting per-dimension breakdowns so that readers can see where judgments are stable versus noisy. Despite the strong empirical reliability (see Table 2), it is still a best-available approximation for evaluating natural-language diagnosis results at scale.

Noise Modeling. The noise injected by SREGYM is a simplification of real-world production disturbances. We inject transient pod crashes and resource stress on schedules, which captures one common pattern (routine pod churn in a busy system) but does not model all sources of production noises, such as high-variance traffic anomalies [61], partial network partitions [25], or slow performance degradation caused by gradual resource exhaustion [34]. Agents that perform well on SREGYM's noise model may still be surprised by noise patterns we do not cover yet.

System Scale. SREGYM's deployed applications are substantially smaller than production systems at major cloud providers. Our largest application, Train Ticket [91], has 40 microservices, whereas production deployments at companies such as Uber, Netflix, and Meta commonly run thousands of interacting services [37, 61]. Scale affects both the diagnosis (agents face fewer candidate services to investigate) and the mitigation evaluation (fewer cross-service dependencies to reason about). Results on SREGYM may not scale linearly to systems that are orders of magnitude larger.

Environment Scope. SREGYM targets cloud-native, Kubernetes-based deployments, which are the dominant platform for modern production systems but not the only one. Workloads running on monolithic deployments or edge deployments have different failure modes that SREGYM does not currently exercise. Extending to these environments would require new fault injectors and observability integrations, which the composable architecture is designed to support.

Agent Coverage in the Evaluation. Constrained by our budget, we can only cover three agents (Stratus, Claude Code, Codex) paired with three frontier models (Claude Sonnet 4.6, Kimi K2.5, GPT-5.4) in our evaluation. This is a small sample relative to the space of possible agent architectures and LLM backbones, and our conclusions about general-purpose coding agents versus specialized SRE agents are correspondingly tentative. We release the benchmark and scoring pipeline so that the community can evaluate additional agent-model combinations and report results under the same oracles. We are committed to supporting such efforts and maintaining the leaderboard.

I Per-Problem End-to-End Results

Figures 7 and 8 show the per-problem end-to-end (E2E) success rate for each agent with and without noise injected into the environment, respectively. Problems (rows) are sorted by mean end-to-end rate across agents; agents (columns) are sorted by overall end-to-end rate. A cell value of $k/3$ indicates that k out of three runs achieved both correct diagnosis and successful mitigation.

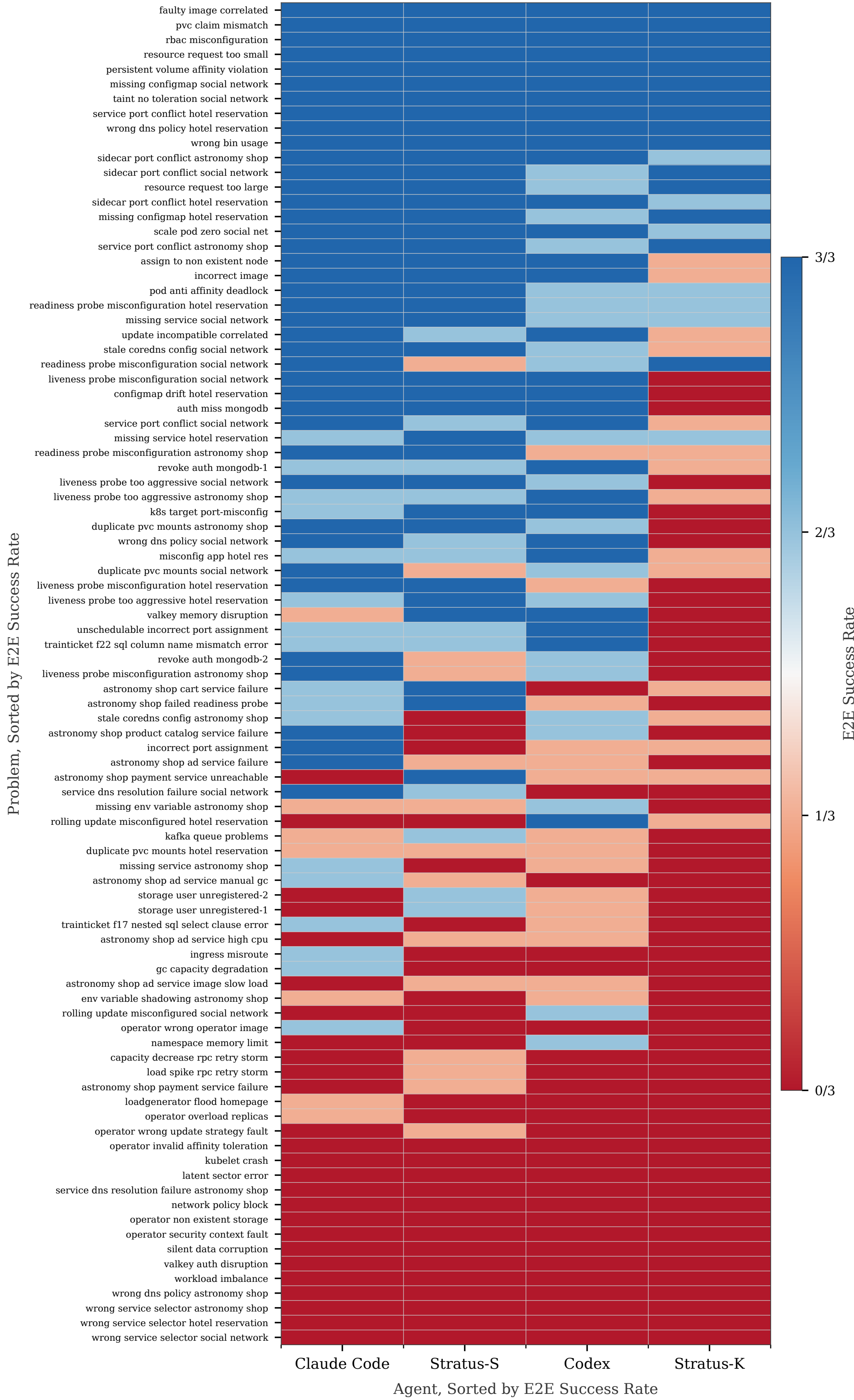


Figure 7: Per-problem end-to-end success rate with no noise injected.

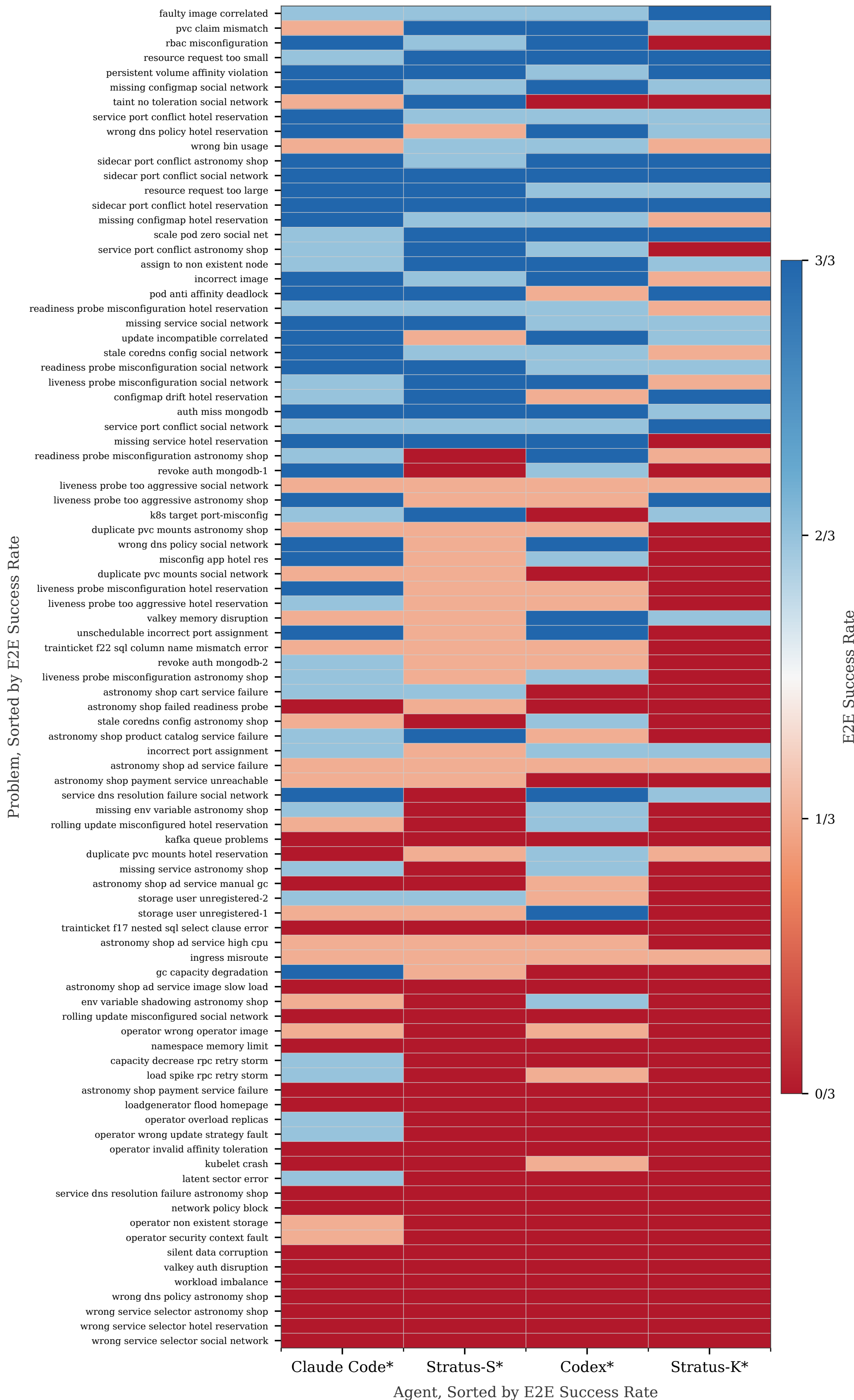


Figure 8: Per-problem end-to-end success rate under noisy conditions.