SREGym: A Live Training Ground for AI SRE Agents with High-Fidelity Failure Drills

Jackson Clark
University of Illinois
Urbana-Champaign
Urbana, USA
jclark58@illinois.edu

Yiming Su
University of Illinois
Urbana-Champaign
Urbana, USA
yiming34@illinois.edu

Saad Mohammad Rafid Pial
Bangladesh University of Engineering
and Technology
Dhaka, Bangladesh
saadmrp@gmail.com

Lily Gniedziejko
University of Illinois
Urbana-Champaign
Urbana, USA
lilyg3@illinois.edu

Tianyin Xu
University of Illinois
Urbana-Champaign
Urbana, USA
tyxu@illinois.edu

Abstract

SREGym is a new benchmark for AI-driven SRE (Site Reliability Engineering) techniques for diagnosing and mitigating production failures. SREGym provides a live training ground where high-fidelity failure drills are emulated through fault injectors. SREGym differs from existing SRE benchmarks such as AIOpsLab and ITBench in its realization of comprehensive, high-fidelity failure drills. SREGym implements an extensible software architecture that orchestrates fault injectors and simulators across system stacks, with new capabilities: (1) simulating low-level faults in OS kernels and hardware, (2) coordinating multiple concurrent events into compound drills, and (3) composing noises to model production environments. We demonstrate how to use and extend SREGym and present three representative cases of how AI agents tackle SREGym problems.

CCS Concepts

• General and reference → Measurement; **• Computer systems organization → Cloud computing**; **• Information systems → Language models**.

ACM Reference Format:
Jackson Clark, Yiming Su, Saad Mohammad Rafid Pial, Lily Gniedziejko, and Tianyin Xu. 2026. SREGym: A Live Training Ground for AI SRE Agents with High-Fidelity Failure Drills. In *Proceedings of the 1st ACM Conference on Agentic and AI Systems (CAIS '26), May 26, 2026, San Jose, CA, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3786335.3813208

1 Introduction

With unprecedented advances of AI and agentic technologies, agentic systems are increasingly used for operating large-scale production systems. A key focus area is Site Reliability Engineering, or SRE, where AI agents automatically diagnose and mitigate system failures to limit their damage and prevent outages. Today, SRE agents are developed by almost all companies that offer cloud services or manage large-scale systems [6, 10, 11, 14, 15]. Recently, many agentic SRE startups are found, indicating that AI for SRE is an active space of innovations [8, 12, 17, 21, 22].

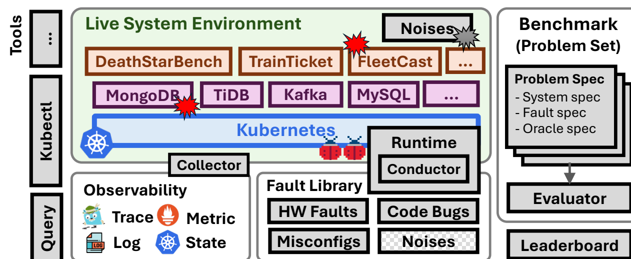


Figure 1: Overview of the SREGym framework. The core SREGym components are in grey.

Surprisingly, unlike other related areas such as AI for code [38], math [9], and tool uses [39], there is no mature, standard benchmark for SRE agents. Consequently, it is hard to comprehensively evaluate different agentic SRE technologies, understand the capabilities and limitations of AI models (to benefit training and fine-tuning), and (for practitioners) compare different agentic SRE products. As benchmarks are key to advancing system intelligence [29], there is a strong desire for high-quality agentic SRE benchmarks, in a similar way to how SWE-bench [38] serves coding agents.

To this end, we have been developing SREGym, an AI-native SRE benchmark for evaluating agentic SRE technologies. Figure 1 gives a high-level overview of SREGym. SREGym adopts the practice of its predecessors (our prior work), AIOpsLab [27] and ITBench [37], to construct *live* system environments using real-world platforms and system stacks (e.g., Kubernetes and cloud-native stacks); it injects different faults, ranging from misconfigurations, software bugs, to hardware faults and misoperations, to create failure scenarios which are *problems* for SRE agents to solve. The agents are provided with tools for querying observability data (for diagnosis) and for changing system configuration and/or code (for mitigation).

The design of SREGym focuses on realizing *comprehensive*, *high-fidelity* failure drills, which is the fundamental missing piece of

This work is licensed under a Creative Commons Attribution 4.0 International License.
CAIS '26, San Jose, CA, USA
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2415-2/26/05
https://doi.org/10.1145/3786335.3813208

existing SRE benchmarks [27, 37, 45]. Existing benchmarks are limited to simple failure scenarios, where each problem is created by injecting a single fault via changing configuration files or crashing application containers in a clean environment. However, such failure drills cannot represent the characteristics of real-world failures in production environments. A main goal of SREGym is to model complex, noisy, and eventful production failures and thus achieve high-fidelity and relevance of the failure drills. For example, SREGym models *noisy* production environments where root-cause faults of the target failures are mingled with other concurrent, low-impact faults (i.e., noises). Moreover, SREGym models faults in low-level system stacks in hardware [34], operating systems [28, 41], and Kubernetes [43], which propagates and manifests in sophisticated failures. In addition, SREGym models correlated failures such as metastable behavior [25] that has no immediate crash symptoms.

To model high-fidelity production failure scenarios, SREGym implements an extensible software architecture that orchestrates fault injectors and event emulators across system stacks, with new capabilities: (1) simulating low-level faults in OS kernels and hardware, (2) coordinating multiple concurrent events into compound failure drills, and (3) composing noises to model production environments. The ability of coordinating faults and events at runtime is key to construct high-fidelity failures, and in our experience, is hard to implement in existing benchmarks which directly invoke local fault injectors (e.g., Ansible playbooks [19] in ITBench).

SREGym's design also emphasizes usability and extensibility. Although usability and extensibility may not reflect academic novelty, they are critically important for SREGym as a community-driven benchmark. Unlike AI-for-code benchmarks like SWE-bench [38] that can leverage existing GitHub issues and pull requests in open-source projects, SRE problems require careful engineering to construct production-like environments and failure scenarios. A key feature of SREGym is its programming interface which helps users to construct new, customized SRE problems by orchestrating different building blocks (see §3).

This paper gives a brief overview of SREGym, with an emphasis on the elements we plan to demonstrate at the conference. Specifically, we plan to first demonstrate how SREGym works as an AI SRE benchmark, including live system environments it exposes and its agent interface (§2). We will then demonstrate how to create a new SRE problem and add it into SREGym (§3). Lastly, we will demonstrate three new SRE problems including a hardware issue (bad sectors), a metastable failure, and concurrent failures, and show how an SRE agent tackles these problems (§4).

2 SREGym: A User's Perspective

SREGym exposes a live system environment where end-user applications are running on a Kubernetes platform with cloud-native stacks for observability and management. Each problem in SREGym creates a unique failure drill using *coordinated* fault injection and event emulation (e.g., for adding noises). SRE agents under evaluation were asked to diagnose and mitigate the target failures using the tools exposed to them—the agents can analyze the system and troubleshoot the failures through observability data (e.g., traces, logs, metrics, and telemetry) and change system states to mitigate failures using Kubernetes' command-line interface (`kubectl`).

SREGym supports different agentic SRE use cases, including *the proactive mode* where the agent needs to detect, diagnose, and mitigate the failures (as in AIOpsLab [27]) and *the reactive mode* where the agent reacts to alerts (as in ITBench [37]). Different from AIOpsLab that artificially breaks SRE tasks down into a hardcoded sequential workflow of four steps (detection, localization, root-cause analysis, and mitigation), SREGym treats agents as autonomous entities and does not restrict the problem-solving strategy or behavior of evaluated agents.

In SREGym, failure drills (i.e., problems) are independently and automatically graded. Each problem encodes the oracles to check that the systems and applications are in healthy states and/or the injected faults are eliminated (see §3). These oracles prevent reward hacking or incomplete mitigation that only suppress alerts without improving the system states (common in alert-driven benckmarks like ITBench). For diagnosis, we currently use an LLM as the judge to evaluate the agent's understanding of the failure based on the relevance to predefined root-cause descriptions.

SREGym can be deployed on emulated Kubernetes clusters using Kind [5] or Minikube [3], which provides a rapid, cost-effective testing and debugging environment for agent development on laptops and personal computers. An SREGym problem deployment (running OpenTelemetry Astronomy Shop [7]) on a clean-slate laptop with 32GB memory and 16-core CPU takes less than 5 minutes. In our demonstration, we will use SREGym on Kind (Kubernetes in Docker) to show the usability and accessibility.

SREGym can also scale to a real Kubernetes cluster across multiple machines or virtual machines, allowing us to create large failure drills (e.g., those that require geo-distributed systems).

2.1 System Environments

SREGym exposes a Kubernetes-based system environment, where applications are deployed in Linux containers. Currently, SREGym supports microservice applications including Hotel Reservation and Social Networks in DeathStarBench [30], Train Ticket [4], Astronomy Shop [7], as well as a few other applications implemented by the SREGym team (e.g., a satellite orbit simulator). These applications interact with data systems such as TiDB, MongoDB, and Kafka, which are further managed by Kubernetes operators [31]. The applications, systems, and operators are deployed using the Helm package manager [2]; SREGym supports any real-world cloud-native applications and systems with Kubernetes manifests or Helm charts. For example, deploying the custom observability data-streaming controller [16] of Resolve AI, a commercial AI SRE product [21], takes only one `kubectl` command.

SREGym provides standard observability backends, including Prometheus for metrics, Loki for logs, and Jaeger for distributed traces. Most agentic SRE products provide built-in API endpoint integrations with these observability backends (e.g., [20]).

Figure 2 demonstrates the live system environment created by SREGym. The application is a satellite orbit simulator that constantly produces simulated satellite orbit data on a Kubernetes cluster. The simulator periodically writes the orbit data to a TiDB database deployed in the Kubernetes cluster with a TiDB operator [18], a management program that simplifies the creation and coordination of the database components. We inject a faulty configuration into the deployment by using a nonexistent container

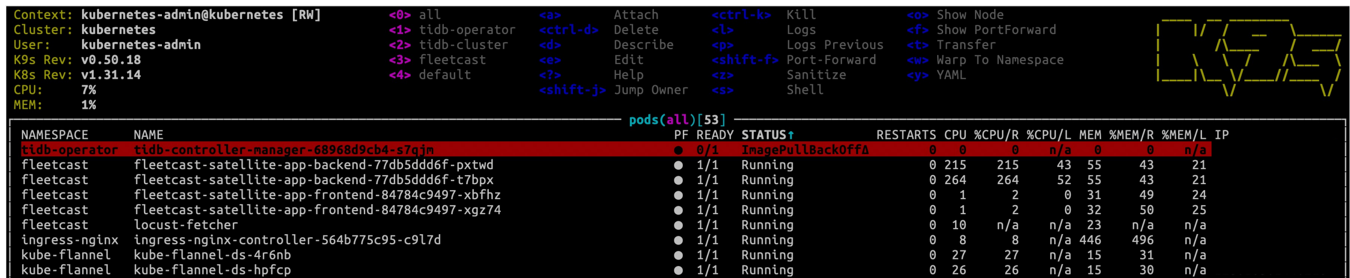


Figure 2: Visualization of a live system environment created by SREGym, where a satellite orbit simulator writes data to a TiDB deployment. The TiDB operator pod is under `ImagePullBackOff` state due to an injected fault (a misconfigured image).

image `pingcap/tidb-operatorr` (with an extra `r`). As shown in Figure 2, the TiDB operator pod gets stuck in the `ImagePullBackOff` state due to the misconfigured image name.

2.2 Agent Interface

The interface between AI agents and its environment has evolved from actively prompting models for tool calls to passively exposed MCP servers and APIs. Unlike AIOpsLab [27], SREGym does not require the agent to explicitly take hardcoded steps or actions. Once the setup finishes, SREGym waits for a `submit` API call from the agent to signal the end of its problem-solving process. Decoupling the agent interface from the benchmark allows SREGym to support diverse agent architectures—in comparison, AIOpsLab only supports ReAct agents. In our demonstration, we will use both a simple ReAct demo agent and Stratus [26], a state-of-the-art agentic SRE system that uses the multi-agent architecture.

SREGym provides MCP (Model Context Protocol) servers for observability backends and Kubernetes to allow autonomous interactions between the agent and the cluster. Specifically, SREGym provides a Prometheus MCP server for metrics, a Loki MCP server for logs, a Jaeger MCP server for traces, and a Kubernetes MCP server for interfacing Kubernetes. They support standard observability actions, such as querying metrics for Prometheus, querying logs for Loki, and querying traces for Jaeger. Kubernetes MCP server serves as a wrapper around the `kubectl` CLI tool, allowing the agent to execute any command on the cluster. Besides MCP servers, SREGym also expose observability endpoints, offering freedom to design customized agentic SRE tools.

2.3 Fault and Noise Injectors

SREGym provides diverse fault injectors as listed in Table 1. These fault injectors include widely used ones and SREGym-specific ones (e.g., the Khaos tool that uses eBPF to simulate various kernel and hardware faults). More fault injectors can be easily integrated. These faults are injected at different layers in the system stack and are manifested through different symptoms. The fault injectors can be used and orchestrated to form specific failure drills (see §3). A key principle is to inject *realistic* faults instead of random faults which are unlikely to happen in practice. For example, SREGym avoids deploying fault injection services (e.g., a chaos engineering pod) directly into the deployed environment, which would easily be identified by AI agents under evaluation. Moreover, software faults (e.g., bugs and misconfigurations) must be realistic instead of randomly fuzzed values.

Table 1: Fault and noise injectors in SREGym.

Mechanism	Simulated Faults
Kill a process or a pod	Fail-stop behavior
Stress hardware (`stress-ng` [1])	Fail-slow behavior [33]
Fail a system call via eBPF	OS and hardware faults [28, 34, 41]
Fail a disk sector (`dm-dust` [13])	Sector errors in disk drives [24, 42]
Inject a fault in deploy.yaml	Service mis-deployment
Inject a fault to application config	Application misconfiguration [44]
Inject a fault to Kubernetes config	Kubernetes misconfiguration
Use buggy application code	Buggy application code
Use buggy application operator	Buggy operator [31, 32]
Increase client loads	Service overload [40]
Inject noises into logs/metrics	Noisy observability data
Create zombie resources	Expired/stale resources
Schedule periodical maintenance	Expected cluster churn

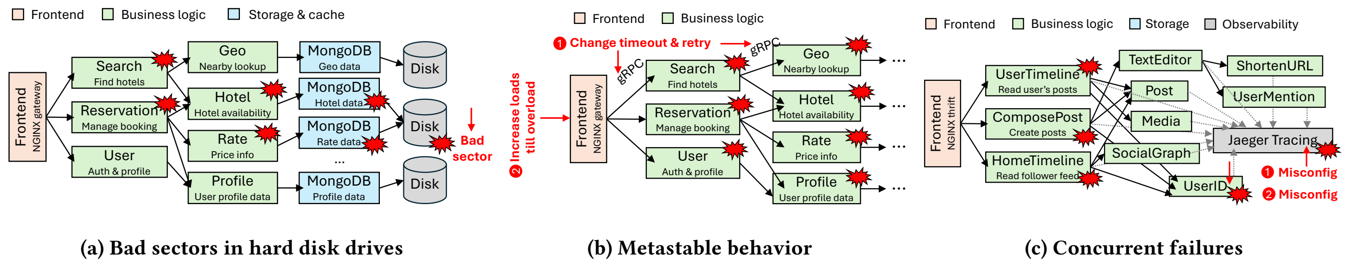
```python
class K8STargetPortMisconfig(Problem):
    def __init__(self):
        self.app = SocialNetwork()          Application
        self.namespace = self.app.namespace
        super().__init__(app=self.app, namespace=self.namespace)

        # === Attach evaluation oracles ===                  Oracles
        self.root_cause = ("The service user-service has a misconfigured target port "
                           "(9999 instead of 9090), causing connection failures.")
        self.diagnosis_oracle = LLMAsAJudgeOracle(problem=self, expected=self.root_cause)
        self.mitigation_oracle = TargetPortMisconfigMitigationOracle(problem=self)

        self.app.create_workload()

    @mark_fault_injected
    def inject_fault(self):                                   Faults
        injector = VirtualizationFaultInjector(namespace=self.namespace)
        injector._inject(
            fault_type="misconfig_k8s",
            microservices="user-service",
        )
        print(f"[FAULT INJECTED] {self.faulty_service} misconfigured")
```

Figure 3: Defining a problem in SREGym, which injects a faulty port misconfiguration in a microservice application

SREGym also provides several noise injectors to emulate noisy environments (Table 1), as real-world production environments are often noisy and dynamic. Note that the fault injectors can also be used to create noises in the form of unrelated failures, where noises are faults that are not root causes of target incidents.

3 Creating a New Problem

A key focus of SREGym is to make it easy to create new problems—an SRE benchmark is only useful with comprehensive, diverse problems that simulate real-world failures. Figure 3 shows an example



(a) Bad sectors in hard disk drives (b) Metastable behavior (c) Concurrent failures

Figure 4: The SREGym problems we will use in the demonstration where the Stratus SRE agent will be applied to tackle them. The problems, as well as the SRE agent behavior, are described in §4. The service graphs are simplified for clarity.

of a problem which injects a faulty port configuration into a Social Network application [30], which we will use in the demonstration.

A problem defines the following components (Figure 3):

- **Deployed applications and systems.** The problem can deploy any supported applications such as the Social Network application from DeathStarBench [30] in Figure 3. The problem can also deploy other concurrently running systems such as Kubernetes operators for managing the applications (which is required by the problem in Figure 1). The applications or systems expose components for fault injection, such as request-listening ports, specified container image, heartbeat message API endpoint, etc.
- **Fault/noise injection.** A problem can specify one or multiple faults or noises to inject into the deployed applications and/or systems using fault injectors (see §2.3). Fault injection is defined by the fault injectors, the target components, and other fault specifications (e.g., the faulty value and parameters).
- **Evaluation oracles.** After the faults are injected, SREGym then launches the SRE agent and waits for the agent to invoke the `submit` API which triggers the evaluation oracles. Each problem must define oracles to evaluate the diagnosis and mitigation results. SREGym uses LLM-as-a-judge as the default diagnosis oracle. The mitigation oracle, on the other hand, is based on system and application states to avoid reward hacking.

4 Solving Problems with Agentic SRE

We plan to demonstrate three representative problems (from 87 problems) in SREGym and show how SRE agents tackle these problems. These three problems are enabled by SREGym's new fault injectors and their orchestration, and thus are hard to do in other existing benchmarks such as ITBench and AIOpsLab. We will use our Stratus SRE agent as a state-of-the-art SRE agent [26] with Claude Opus 4.6 as one of the best performing LLMs.

Bad sectors in hard disk drives. SREGym can simulate low-level OS and hardware faults. We demonstrate one problem which simulates bad sectors of hard disk drives [24, 42]. SREGym uses existing fault injector `dm-dust` which emulates the behavior of bad sectors at arbitrary locations. The injected sector faults propagate and manifest as I/O-related system call failures when the application attempts to read and write data from the disk. Figure 4a shows the problem where the bad sectors affect the deployed Hotel Reservation application from DeathStarBench [30] when it attempts to read data from MongoDB.

Metastable behavior. SREGym can simulate metastable failures [25, 35], self-sustaining congestive collapses in which the system degrades in response to transient events (e.g., a load surge) but fail to recover after the trigger is removed. Recent studies report that metastable failures are hard to diagnose and mitigate, because they do not manifest through crashing behavior [35, 36]. We created three metastable failure drills in SREGym by integrating the Blueprint tool [23] and will demonstrate one of them as depicted in Figure 4b. In this problem, we first set overly aggressive gRPC configurations for connection timeout (50ms) and retry count (30). We then push the Hotel Reservation application from DeathStarBench [30] into a vulnerable state with a high load of 3000 requests per second. Finally, we trigger the metastable failure by a transient CPU stress so all RPC requests are timed out and retried at the same time, causing a retry storm. SREGym's ability to coordinate different events are key to creating the problem.

Concurrent failures. We will demonstrate a problem that composes multiple fault injectors to create concurrent failures. Figure 4c depicts the problem where two faults are injected into a Social Network application: (1) a scheduler misconfiguration that makes an observability service pod un-scheduable, and (2) a network misconfiguration that fails user requests. This problem evaluates whether SRE agents understand failure severity—the SRE agent should prioritize the network misconfiguration as it directly affects service availability, while the scheduler misconfiguration only affects the observability service which is internal and invisible to end users. We show that, through fault composition, SREGym can help create different problems by orchestrating multiple fault injectors and noise emulators, and the composed failure drills are arguably more realistic than problems with single fault.

5 Artifact

We are actively maintaining SREGym as an open-source project at https://github.com/SREGym/SREGym. The project contains sufficient documents and provides 87 built-in problems of various kinds of failure drills. We also provide a demo artifact in the project: https://github.com/SREGym/SREGym/tree/demo, for users who do not have API keys of LLM services.

Acknowledgement

We thank CAIS reviewers for insightful comments. SREGym is supported by the Laude Institute. We thank all the contributors to SREGym and users who provided us with valuable feedback.

References

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

[2] Helm - The package manager for Kubernetes. https://helm.sh/, 2015.

[3] minikube - Run Kubernetes locally. https://minikube.sigs.k8s.io/docs/, 2016.

[4] Train Ticket - A Benchmark Microservice System. https://github.com/FudanSELab/train-ticket, 2018.

[5] kind - Kubernetes IN Docker. https://kind.sigs.k8s.io/, 2021.

[6] Free to be SRE — how to use generative AI to code, test and troubleshoot your systems. https://cloud.google.com/blog/products/devops-sre/learn-how-generative-ai-can-help-with-sre-tasks/, 2024.

[7] 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.

[8] SRE.ai - AI DevOps Agents. https://www.ycombinator.com/companies/sre-ai, 2024.

[9] AIME 2025 Benchmark Leaderboard. https://artificialanalysis.ai/evaluations/aime-2025, 2025.

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

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

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

[13] dm-dust. https://docs.kernel.org/admin-guide/device-mapper/dm-dust.html, 2025.

[14] IBM Cloud - AIOps solutions. https://www.ibm.com/solutions/aiops, 2025.

[15] Introducing Bits AI SRE, your AI on-call teammate. https://www.datadoghq.com/blog/bits-ai-sre/, 2025.

[16] Resolve Satellite. https://docs.resolve.ai/resolve-satellite, 2025.

[17] Rootly AI - AI SRE agents that resolve your hardest incidents. https://rootly.com/, 2025.

[18] TiDB Operator Overview. https://docs.pingcap.com/tidb-in-kubernetes/stable/tidb-operator-overview/, 2025.

[19] Ansible Playbooks. https://docs.ansible.com/projects/ansible/latest/playbook_guide/playbooks_intro.html, 2026.

[20] Resolve AI - Extensive library of integrations. https://resolve.ai/integrations, 2026.

[21] Resolve.ai | AI for prod. https://resolve.ai, 2026.

[22] The AI SRE for even the most complex incidents. https://traversal.com/, 2026.

[23] Anand, V., Garg, D., Kaufmann, A., and Mace, J. Blueprint: A Toolchain for Highly-Reconfigurable Microservice Applications. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP'23)* (Oct. 2023).

[24] Bairavasundaram, L. N., Goodson, G. R., Pasupathy, S., and Schindler, J. 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).

[25] Bronson, N., Aghayev, A., Charapko, A., and Zhu, T. Metastable failures in distributed systems. In *Proceedings of the ACM SIGOPS 21st Workshop on Hot Topics in Operating Systems (HotOS'21)* (June 2021).

[26] Chen, Y., Pan, J., Clark, J., Su, Y., Zheutlin, N., Bhavya, B., Arora, R., Deng, Y., Jha, S., and Xu, T. 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)* (2025).

[27] Chen, Y., Shetty, M., Somashekar, G., Ma, M., Simmhan, Y., Mace, J., Bansal, C., Wang, R., and Rajmohan, S. 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).

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

[29] Feng, X., Cheng, P., Chen, Q., Lu, S., Liang, C.-J. M., Stoica, B. A., Guo, Z., Xu, J., Xu, T., and Zhou, L. Defining System Intelligence. https://www.sigops.org/2025/defining-system-intelligence, Nov. 2025.

[30] Gan, Y., Zhang, Y., Cheng, D., Shetty, A., Rathi, P., Katarki, N., Bruno, A., Hu, J., Ritchken, B., Jackson, B., Hu, K., Pancholi, M., He, Y., Clancy, B., Colen, C., Wen, F., Leung, C., Wang, S., Zaruvinsky, L., Espinosa, M., Lin, R., Liu, Z., Padilla, J., and Delimitrou, C. 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).

[31] Gu, J. T., Sun, X., Zhang, W., Jiang, Y., Wang, C., Vaziri, M., Legunsen, O., and Xu, T. 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).

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

[33] Gunawi, H. S., Suminto, R. O., Sears, R., Golliher, C., Sundararaman, S., Lin, X., Emami, T., Sheng, W., Bidokhti, N., McCaffrey, C., Grider, G., Fields, P. M., Harms, K., Ross, R. B., Jacobson, A., Ricci, R., Webb, K., Alvaro, P., Runesha, H. B., Hao, M., and Li, H. 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).

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

[35] Huang, L., Magnusson, M., Muralikrishna, A. B., Estyak, S., Isaacs, R., Aghayev, A., Zhu, T., and Charapko, A. Metastable Failures in the Wild. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI'22)* (July 2022).

[36] Isaacs, R., Alvaro, P., Majumdar, R., Muniswamy-Reddy, K.-K., Salamati, M., and Soudjani, S. Analyzing Metastable Failures. In *Proceedings of the ACM SIGOPS 21st Workshop on Hot Topics in Operating Systems (HotOS'25)* (May 2025).

[37] Jha, S., Arora, R. R., Watanabe, Y., Yanagawa, T., Chen, Y., Clark, J., Bhavya, B., Verma, M., Kumar, H., Kitahara, H., Zheutlin, N., Takano, S., Pathak, D., George, F., Wu, X., Turkkan, B. O., Vanloo, G., Nidd, M., Dai, T., Chatterjee, O., Gupta, P., Samanta, S., Aggarwal, P., Lee, R., wook Ahn, J., Kar, D., Paradkar, A., Deng, Y., Moogi, P., Mohapatra, P., Abe, N., Narayanaswami, C., Xu, T., Varshney, L. R., Mahindru, R., Sailer, A., Shwartz, L., Sow, D., Fuller, N. C. M., and Puri, R. ITBench: Evaluating AI Agents across Diverse Real-World IT Automation Tasks. In *Proceedings of the 42th International Conference on Machine Learning (ICML'25)* (May 2025).

[38] Jimenez, C. E., Yang, J., Wettig, A., Yao, S., Pei, K., Press, O., and Narasimhan, K. R. SWE-bench: Can Language Models Resolve Real-world Github Issues? In *Proceedings of the 12th International Conference on Learning Representations (ICLR'24)* (Mar. 2024).

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

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

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

[42] Schroeder, B., Damouras, S., and Gill, P. 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).

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

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

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