

LLM-Powered Fully Automated Chaos Engineering: Towards Enabling Anyone to Build Resilient Software Systems at Low Cost

Daisuke Kikuta
NTT, Inc., Japan
daisuke.kikuta@ntt.com

Hiroki Ikeuchi
NTT, Inc., Japan
hiroki.ikeuchi@ntt.com

Kengo Tajiri
NTT, Inc., Japan
kengo.tajiri@ntt.com

Abstract—Chaos Engineering (CE) is an engineering technique aimed at improving the resilience of distributed systems. It involves intentionally injecting faults into a system to test its resilience, uncover weaknesses, and address them before they cause failures in production. Recent CE tools automate the execution of predefined CE experiments. However, planning such experiments and improving the system based on the experimental results still remain manual. These processes are labor-intensive and require multi-domain expertise. To address these challenges and enable anyone to build resilient systems at low cost, this paper proposes ChaosEater, a system that automates the entire CE cycle with Large Language Models (LLMs). It predefines an agentic workflow according to a systematic CE cycle and assigns subdivided processes within the workflow to LLMs. ChaosEater targets CE for software systems built on Kubernetes. Therefore, the LLMs in ChaosEater complete CE cycles through software engineering tasks, including requirement definition, code generation, testing, and debugging. We evaluate ChaosEater through case studies on small- and large-scale Kubernetes systems. The results demonstrate that it consistently completes reasonable CE cycles with significantly low time and monetary costs. Its cycles are also qualitatively validated by human engineers and LLMs.

Index Terms—Large Language Models, AI Agents, AIOps, Chaos Engineering, Failure Management, Software Systems

I. INTRODUCTION

Modern software applications are built on distributed systems, in which the entire system is composed of multiple component services. This design, known as microservice architecture [1], enables scalable and continuous deployment while supporting the integration of heterogeneous technologies. On the other hand, the complex dependencies among microservices can cause faults to propagate, and even minor faults may lead to unpredictable and chaotic behavior across the entire system. However, proactively predicting and addressing such complex behavior is challenging.

To address this and improve the resilience of distributed systems, numerous organizations, including Netflix, Amazon, and Microsoft, have recently adopted Chaos Engineering (CE) [2], [3]. Its concept is that *rather than predicting the chaotic behavior, let's observe it directly by artificially injecting faults into the system*. Based on the actual observation, we can proactively rebuild a new system that is resilient to the assumed faults. Systematically, CE cycles through four phases for a target system:

- 1) **Hypothesis**: Define steady states (i.e., normal behavior) of the system and a failure scenario. Then, make a

hypothesis that *the steady states of the system are maintained even when the failures occur in the scenario*.

- 2) **(Chaos) Experiment**: Inject the relevant faults into the system while monitoring its response behavior.
- 3) **Analysis**: Analyze the logged data and check if the hypothesis is satisfied. If so, this CE cycle is finished here. If not, move to the next *improvement* phase.
- 4) **Improvement**: Reconfigure the system to satisfy the hypothesis. The reconfigured system is tested again in (2) and (3), i.e., repeat (2) to (4) until the hypothesis is satisfied.

In recent years, several CE tools [4]–[7] have advanced the automation of chaos-experiment execution. Moreover, monitoring tools [8]–[10] enable automating metric collection, aggregation, and threshold-based testing during chaos experiments. Hence, the *experiment* and *analysis* phases have been mostly automated. On the other hand, other processes such as making a hypothesis, planning a chaos experiment to test the hypothesis, and reconfiguring the system still remain manual. These manual processes are labor-intensive and require multi-domain expertise; therefore, automating them is expected to enable anyone to build resilient systems at low cost. However, due to their complexity and generative nature, such automation has not yet been achieved by existing algorithmic approaches.

We believe that Large Language Models (LLMs) are the key to overcoming this challenge. LLMs have recently shown promising capabilities across a wide range of tasks, including general natural language processing [11], software engineering [12]–[15], and network operations [16]–[18]. Considering that CE for software systems corresponds to software engineering, these capabilities sufficiently satisfy the requirements of the manual processes in CE. Moreover, concurrent with our work, the use of LLMs for automating partial processes in software system management is also being actively explored, including root cause analysis [19]–[25], log parsing [26], [27], and remediation [28]–[30]. Given these situations, the full automation of CE cycles is now becoming feasible.

To realize this, we propose ChaosEater, an LLM-based system that automates the entire CE cycle. It predefines an agentic workflow according to the systematic CE cycle and assigns subdivided CE processes within the workflow to each LLM with a specific role. The predefined workflow ensures that the multiple LLMs collaboratively complete CE cycles

as intended. Considering the compatibility between software systems and LLMs, ChaosEater especially targets CE on Kubernetes (K8s) systems [31]. Therefore, the LLMs complete CE cycles through SE tasks, including requirement definition, code generation, testing, and debugging. In this paper, we specifically present the agentic workflow design, how LLMs perform subdivided CE processes within the workflow, and some techniques for consistently completing reasonable CE cycles. We also evaluate ChaosEater through case studies on both small- and large-scale K8s systems. The results demonstrate that it consistently completes reasonable single CE cycles with significantly low time and monetary costs (\$0.2–0.8 and 11–25min). The validity of these CE cycles is also confirmed by two human engineers and three different LLMs.

The main contributions of this paper are threefold:

- We are the first to implement an LLM-based system that automates the entire CE cycle. This implementation provides evidence for the feasibility of a new direction: the full automation of system resilience improvement.
- We release all the resources of ChaosEater.¹ This release provides a concrete foundation for subsequent works in the new direction.
- We evaluate ChaosEater quantitatively in terms of cost and stability, and qualitatively in terms of the validity of its CE processes. The results provide its fine-grained potential, limitations, and future directions.

II. PROPOSED SYSTEM: CHAOS EATER

In this section, we describe a technical overview of ChaosEater. Fig. 1 shows its simplified agentic workflow. It takes as input instructions for the CE cycle (optional) and a folder containing K8s manifests and a Skaffold configuration file [32].² It then conducts a CE cycle for these inputs through six divided phases: *pre-processing*, *hypothesis*, *experiment*, *analysis*, *improvement*, and *post-processing* phases. Finally, it outputs a summary of the completed CE cycle and a modified folder containing K8s manifests that have been reconfigured to satisfy the hypothesis defined in the *hypothesis* phase, along with their corresponding Skaffold configuration file.

In the following, we describe ChaosEater’s workflow design from input to output, breaking it down into the six phases. Note that, in this paper, we refer to each LLM assigned a specific role (i.e., a subdivided CE process) as an LLM agent, and that the underlying LLMs do not require additional fine-tuning. See the extended version [33] for implementation details, including the detailed agentic workflow, system prompt templates, graphical user interface, and system deployment.

A. Phase 0: Pre-processing

Given the user input, ChaosEater first deploys the user’s system to the K8s cluster by running the Skaffold configuration file. Then, each LLM agent sequentially fills in the

¹Available at <https://github.com/ntt-dkiku/chaos-eater>.

²K8s manifests are system configuration files that define the resources (i.e., microservices) that constitute a system, while a Skaffold configuration file defines the deployment process of those resources to a K8s cluster.

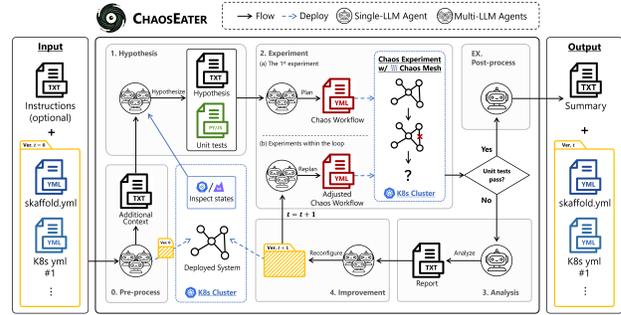


Fig. 1. The agentic workflow of ChaosEater. It autonomously completes the systematic CE cycle using LLM agents and existing tools. Note that only the representative inputs and outputs of agents are illustrated here. The two K8s clusters within the workflow refer to the same one.

implicit context of the user’s input. The filled context includes summaries of the K8s manifests, their potential issues for resilience and redundancy, and a possible application, which will serve as auxiliary information in the subsequent phases. The filtering of harmful prompts in user instructions is also performed here.

B. Phase 1: Hypothesis

The *hypothesis* phase defines the required system resilience for an assumed failure scenario. Following the principles of CE [2], ChaosEater first defines steady states and then defines a failure scenario.

Steady-state definition: Steady states are the expected and normal behavior of a system. Each steady state is defined by a pair of a state value and a threshold, and a steady state is considered satisfied when the state value meets the threshold. Therefore, the state values must be measurable outputs of the system, such as the number of active resources, error rates, and response time. Given the pre-processed user input, an LLM agent first defines a measurable state critical to maintaining the system’s application. Another agent then inspects the current value of the state using either K8s API or k6 [9]. The inspection is conducted by a Python or JavaScript script written by the agent. Based on the inspected value, an agent defines the threshold for the state, which, according to the definition of a steady state, must be satisfied under the current normal state. Finally, an agent adds threshold-based assertions to the inspection script to generate a unit test script that validates whether the steady state is satisfied (Fig. 2). These processes are repeated to list multiple steady states without duplication until an agent determines that the number of steady states is sufficient. The unit test script is used for mechanically validating the steady state during the *experiment* phase; we call this approach of having LLMs judge validity through unit test code *Validation as Code* (VaC), which ensures consistency and transparency of the LLM’s validation process.

Failure definition: Given the pre-processed user inputs and the steady states, an LLM agent proposes a failure scenario that may occur in the system (e.g., a surge in access due to a promotional campaign, cyber attack, etc.), and defines

```

(a) VaC script for K8s API (Python)
def check_podcount(label, expected_count, duration):
    consistent_count = True
    for i in range(duration):
        pods = self.v1.list_namespaced_pod(
            namespace="default",
            label_selector=label)
        pod_count = len(pods.items)
        print(f"current pod count: {pod_count}")
        consistent_count = pod_count == expected_count
        if not consistent_count:
            break
        time.sleep(1)
    assert consistent_count, "Pod count was inconsistent"
...

(b) VaC script for k6 (Javascript)
export const options = {
  vus: 10,
  duration: '10s',
  thresholds: {
    http_req_duration: ['p(95)<500'],
  },
};
export default function () {
  const res = http.get('http://example.com');
  check(res, {status was 200: (r) =>
    r.status == 200 });
  sleep(1);
}

```

Fig. 2. Examples of VaC scripts to validate steady states.

a sequence of faults that simulate the scenario. ChaosEater employs Chaos Mesh [6], a CE tool that can manage chaos experiments on K8s systems through code; therefore, the faults are selected from those supported by Chaos Mesh. After drafting faults, another agent refines parameters for each fault, such as the scope of the fault injection, the fault sub-type, and the fault strength. This stepwise fault detailing helps LLMs specify accurate parameter sets as structured JSON outputs.

At this point, the hypothesis can be reframed as *all VaC scripts pass, even when the defined faults are injected*.

C. Phase 2: (Chaos) Experiment

The *experiment* phase plans a chaos experiment to validate the hypothesis and executes it.

Experiment planning: To enable systematic planning, we divide a chaos experiment into three stages: pre-validation, fault-injection, and post-validation. In pre-validation, VaC scripts validate steady states under normal conditions. In fault-injection, faults are injected. If some steady states need to be validated concurrently, their corresponding VaC scripts are also run here. In post-validation, VaC scripts confirm the recovery of steady states after failures. Given the pre-processed user inputs and the hypothesis, an LLM agent first determines the duration of each stage. Then, other agents determine the VaC scripts and fault injections to be executed in each stage, along with their execution timing and durations. Finally, an agent writes a summary of the chaos experiment timeline, which is referenced during the *analysis* phase to identify the causes of the hypothesis rejection. The chaos experiment plan is then converted to a Chaos Mesh workflow manifest, which enables automated fault injection and hypothesis validation via VaC scripts according to the schedule defined in the manifest.

Experiment replanning: Resource types and metadata of K8s manifests may be reconfigured during the *improvement* phase. Therefore, inspection targets in VaC scripts and scopes of fault injections must be updated accordingly between the *improvement* phase and the next experiment execution. Given the original and reconfigured K8s manifests, as well as the previous plan, each LLM agent proposes new inspection targets and new scopes of fault injections. Then, a new ChaosMesh workflow manifest is generated by updating the corresponding parts in the previous one. Note that this update only makes minor adjustments to reflect the changes in K8s manifests, without altering the original intent of the chaos experiment.

Experiment execution: After the Chaos Mesh workflow manifest is generated, ChaosEater applies it to the K8s cluster. Then, the scheduled fault injections and hypothesis validation

are automatically executed by Chaos Mesh. In the meantime, ChaosEater simply waits for the experiment to complete.

D. Phase 3: Analysis

After the chaos experiment is finished, ChaosEater mechanically checks whether the VaC scripts have passed. If all of them have passed, that means the current system already satisfies the hypothesis. Therefore, ChaosEater finishes the current CE cycle at this point and moves to the *post-processing* phase. If at least one has failed, ChaosEater moves to the next *improvement* phase after analyzing the experimental results. In this analysis, given the K8s manifests, the timeline of the chaos experiments, and the list of failed VaC scripts with their logs, an LLM agent identifies the cause of the test failures and then generates a report containing the causes and countermeasures.

E. Phase 4: Improvement

The *improvement* phase reconfigures the K8s manifests to satisfy the hypothesis. Given the K8s manifests, the hypothesis, the experiment plan, and the improvement loop history, an LLM agent reconfigures the K8s manifests by replacing, creating, or deleting them, so that all the VaC scripts pass in the chaos experiment. This agent is instructed to gradually increase redundancy at each step to avoid excessive redundancy.

Improvement loop: After the reconfiguration, ChaosEater applies the reconfigured K8s manifests to the K8s cluster. They are then validated again through the *experiment* and *analysis* phases. That is, as in the systematic CE cycle, ChaosEater also repeats the *experiment*, *analysis*, *improvement* phases until the hypothesis is satisfied. We define this loop as *improvement loop*. The improvement loop history refers to the history of the experimental results, their analysis reports, and their reconfigurations within this improvement loop, which suppresses the repetition of the same reconfiguration.

F. Extra Phase: Post-processing

After the CE cycle is completed, ChaosEater finalizes its entire process by summarizing the completed CE cycle. An LLM agent summarizes the user’s input and the four completed phases. Finally, ChaosEater provides the user with the summary of the completed CE cycle and the folder containing K8s manifests that have been reconfigured to satisfy the hypothesis defined in the *hypothesis* phase, along with their Scaffold configuration file.

III. CASE STUDY

CE cycles should not be evaluated solely based on whether appropriate reconfigurations are performed; and it is equally important to evaluate whether each phase leading up to the reconfiguration is meaningful. Therefore, rather than creating a benchmark that focuses on quantitative metrics, we evaluate ChaosEater both quantitatively and qualitatively through in-depth case studies focusing on two critical cases.

The first case, NGINX, is a minimal system consisting of two K8s manifests: Pod.yml and Service.yml. The former defines a Pod including an Nginx server, and the latter defines

TABLE I
TIME AND MONETARY COSTS OF SINGLE CE CYCLES BY CHAOS EATER.

Target	Metric	All	Pre	Hyp.	Expt.	Anlys.	Imp.	Post
NGINX	Input tokens	59k	2.6k	25k	13k	4.4k	5.5k	8.2k
	Output tokens	5.9k	0.5k	2.5k	1.7k	0.6k	0.2k	0.4k
	API cost (\$)	0.21	0.01	0.09	0.05	0.02	0.02	0.02
	Time	11m	21s	2.6m	4.4m	50s	12s	21s
SOCKSHOP	Input tokens	284k	30k	150k	57k	14k	15k	18k
	Output tokens	13k	5.7k	3.8k	1.8k	0.7k	0.6k	0.5k
	API cost (\$)	0.84	0.13	0.41	0.16	0.04	0.04	0.05
	Time	25m	4.6m	4.3m	3.3m	36s	4.3m	21s

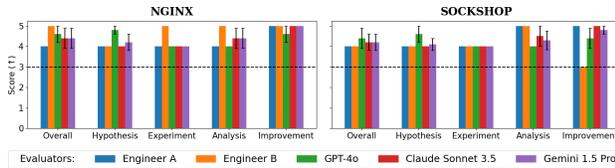


Fig. 3. Qualitative evaluation results of CE cycles for each system. A score of 3 or higher is considered a positive rating.

a Service routing TCP traffic to the Pod. To verify whether ChaosEater can improve the system when there are resilience issues, we intentionally configure the resource with a non-resilient setting: we set `restartPolicy` to `Never` in `Pod.yml`. With this configuration, once the Pod goes down, it will never restart, resulting in extended service outages. The second case, SOCKSHOP [34], is a practical and relatively large-scale e-commerce system that consists of 29 manifests. The number of replicas of all the Deployments is originally set to one. However, this setting could lead to downtime of the single replica when it goes down. To narrow down this resilience issue, we configure all Deployments to have two replicas except for `front-end-dep.yml`, which remains at one. This relatively reduces the resilience of the front-end resource.

For these cases, we validate whether ChaosEater correctly identifies and addresses these resilience issues through a reasonable CE cycle. To maintain the autonomy of ChaosEater, we input only the instruction to keep each chaos experiment within one minute (for SOCKSHOP, access methods are also input). We use `gpt-4o-2024-08-06` [35] as the underlying LLMs. To improve the reproducibility of this case study, its temperature is set to 0. We run single CE cycles for each target system five times under the same settings.

Costs and stability: Table I shows the time and monetary costs of single CE cycles for each target system. Although we do not have statistical data on the actual working time and labor costs for the same CE cycles performed by human engineers, the total costs for NGINX (\$0.21 and 11min) are obviously lower than those. For SOCKSHOP, the monetary cost increases by approximately four times (\$0.84), and the time doubled (25min). However, these values are still intuitively lower than those of human engineers. Even with the number of resources increasing by more than ten times compared to NGINX, the cost increase remains minimal, demonstrating that

ChaosEater maintains low costs even for large-scale systems. In terms of stability, ChaosEater successfully completes the CE cycle for each system without runtime errors in all five runs. It also correctly reconfigures NGINX in all five runs and SOCKSHOP in four out of five runs. Even in the non-reconfigured case, we confirm that a valid CE cycle is completed without requiring reconfigurations.

Qualitative validation: To validate CE cycles completed by ChaosEater, we select one of the five runs for each target system and qualitatively evaluate the four phases and the overall process using a five-point scale. Here, the scale is designed so that a score of 3 or higher is a positive rating. The evaluators are two external human engineers and three LLMs: GPT-4o, Claude Sonnet 3.5 [36], and Gemini Pro 1.5 [37]. The LLM evaluators evaluate each CE cycle five times with a temperature of 0, and the final score is calculated as their average. Fig. 3 shows the results of the qualitative evaluation of the two cycles conducted by the evaluators. The results show that all evaluators rated every phase above the threshold for a positive rating for both systems, demonstrating that ChaosEater completed reasonable single CE cycles. In fact, it defined Pod availability as one of the steady states and hypothesized that they are maintained even in scenarios where Pods go down, such as cyberattacks and Black Friday sales. Through these experiments, it successfully identified the issues of `restartPolicy` and the number of replicas, and solved them by replacing Pod with a Deployment resource and increasing the number of replicas. See the extended version [33] for more details of the evaluation settings, the complete outputs, and the full reviews by the evaluators.

IV. CONCLUSION, LIMITATIONS, AND FUTURE WORK

CE is labor-intensive and requires multi-domain expertise; therefore, its automation is desirable from the perspectives of cost reduction and accessibility. Furthermore, as the automatic generation of software applications by LLMs has become widespread recently, such automation is becoming even more important for ensuring the resilience of the generated applications. To address this demand, this paper provided evidence for the feasibility of the full automation of CE, suggesting a future where anyone can build resilient systems at low cost.

On the other hand, the current ChaosEater has several limitations: **1)** It can be deployed only in development environments due to security concerns; **2)** It supports only reconfiguration of K8s manifests; **3)** Its vulnerability discovery capability is limited within a single CE cycle.

In future work, we will address them in the following directions: **1)** It should control the impact range of faults and be safeguarded by a higher-level monitoring system that continuously oversees its operation; **2)** To optimally improve system resilience, it should also reconfigure frontend code (HTML, CSS, and JavaScript), application code (Python), and lower-layer code (Terraform); **3)** It should perform long-horizon vulnerability exploration across multiple CE cycles, where agents iteratively refine hypotheses based on previous cycles and the temporally changing state of the system.

REFERENCES

- [1] A. Bucchiarone, N. Dragoni, S. Dustdar, P. Lago, M. Mazzara, V. Rivera, and A. Sadovykh, Eds., *Microservices, Science and Engineering*. Springer, 2020. [Online]. Available: <https://doi.org/10.1007/978-3-030-31646-4>
- [2] A. Basiri, N. Behnam, R. de Rooij, L. Hochstein, L. Kosewski, J. Reynolds, and C. Rosenthal, "Chaos engineering," *IEEE Software*, vol. 33, no. 3, pp. 35–41, 2016.
- [3] A. Basiri, L. Hochstein, N. Jones, and H. Tucker, "Automating chaos experiments in production," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, 2019, pp. 31–40.
- [4] Netflix, "Chaos monkey," 2012. [Online]. Available: <https://github.com/Netflix/chaosmonkey>
- [5] Amazon Web Services, "Aws fault injection service," 2021. [Online]. Available: <https://aws.amazon.com/fis/>
- [6] Chaos Mesh, "Chaos mesh," 2020. [Online]. Available: <https://github.com/chaos-mesh/chaos-mesh/>
- [7] Microsoft, "Azure chaos studio," 2023. [Online]. Available: <https://azure.microsoft.com/products/chaos-studio/>
- [8] Prometheus, "Prometheus," 2012. [Online]. Available: <https://github.com/prometheus/prometheus>
- [9] Grafana Labs, "k6," 2021. [Online]. Available: <https://github.com/grafana/k6>
- [10] Cloud Native Computing Foundation, "Opentelemetry," 2019. [Online]. Available: <https://github.com/open-telemetry>
- [11] W. X. Zhao, K. Zhou, J. Li, T. Tang, X. Wang, Y. Hou, Y. Min, B. Zhang, J. Zhang, Z. Dong, Y. Du, C. Yang, Y. Chen, Z. Chen, J. Jiang, R. Ren, Y. Li, X. Tang, Z. Liu, P. Liu, J.-Y. Nie, and J.-R. Wen, "A survey of large language models," 2023. [Online]. Available: <https://arxiv.org/abs/2303.18223>
- [12] J. Jiang, F. Wang, J. Shen, S. Kim, and S. Kim, "A survey on large language models for code generation," *ACM Trans. Softw. Eng. Methodol.*, 2025. [Online]. Available: <https://doi.org/10.1145/3747588>
- [13] C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. R. Narasimhan, "SWE-bench: Can language models resolve real-world github issues?" in *The Twelfth International Conference on Learning Representations*, 2024. [Online]. Available: <https://openreview.net/forum?id=VTF8yNQm66>
- [14] J. Yang, C. E. Jimenez, A. Wettig, K. Lieret, S. Yao, K. R. Narasimhan, and O. Press, "SWE-agent: Agent-computer interfaces enable automated software engineering," in *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024. [Online]. Available: <https://openreview.net/forum?id=mXpq6ut8J3>
- [15] Cognition Labs, "Introducing devin, the first ai software engineer," 2024. [Online]. Available: <https://devin.ai>
- [16] J. Wang, L. Zhang, Y. Yang, Z. Zhuang, Q. Qi, H. Sun, L. Lu, J. Feng, and J. Liao, "Network meets chatgpt: Intent autonomous management, control and operation," *Journal of Communications and Information Networks*, vol. 8, no. 3, pp. 239–255, 2023.
- [17] Y. Huang, H. Du, X. Zhang, D. Niyato, J. Kang, Z. Xiong, S. Wang, and T. Huang, "Large language models for networking: Applications, enabling techniques, and challenges," *IEEE Network*, vol. 39, no. 1, pp. 235–242, 2025.
- [18] G. Bovenzi, F. Cerasuolo, D. Ciuonzo, D. Di Monda, I. Guarino, A. Montieri, V. Persico, and A. Pescapé, "Mapping the landscape of generative ai in network monitoring and management," *IEEE Transactions on Network and Service Management*, vol. 22, no. 3, pp. 2441–2472, 2025.
- [19] M. Bedoya, S. Palacios, D. Díaz-López, E. Laverde, and P. Nespoli, "Enhancing devsecops practice with large language models and security chaos engineering," *International Journal of Information Security*, vol. 23, pp. 3765–3788, 2024.
- [20] 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 *Eighth Conference on Machine Learning and Systems*, 2025. [Online]. Available: <https://openreview.net/forum?id=3EXBLwGxtq>
- [21] 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. Turkan, 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 *Forty-second International Conference on Machine Learning*, 2025. [Online]. Available: <https://openreview.net/forum?id=jP59rz1bZk>
- [22] Y. Han, Q. Du, Y. Huang, J. Wu, F. Tian, and C. He, "The potential of one-shot failure root cause analysis: Collaboration of the large language model and small classifier," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 931–943. [Online]. Available: <https://doi.org/10.1145/3691620.3695475>
- [23] X. Zhang, S. Ghosh, C. Bansal, R. Wang, M. Ma, Y. Kang, and S. Rajmohan, "Automated root causing of cloud incidents using in-context learning with gpt-4," in *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, ser. FSE 2024. New York, NY, USA: Association for Computing Machinery, 2024, p. 266–277. [Online]. Available: <https://doi.org/10.1145/3663529.3663846>
- [24] D. Goel, F. Husain, A. Singh, S. Ghosh, A. Parayil, C. Bansal, X. Zhang, and S. Rajmohan, "X-lifecycle learning for cloud incident management using llms," in *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, ser. FSE 2024. New York, NY, USA: Association for Computing Machinery, 2024, p. 417–428. [Online]. Available: <https://doi.org/10.1145/3663529.3663861>
- [25] D. Roy, X. Zhang, R. Bhave, C. Bansal, P. Las-Casas, R. Fonseca, and S. Rajmohan, "Exploring llm-based agents for root cause analysis," in *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, ser. FSE 2024. New York, NY, USA: Association for Computing Machinery, 2024, p. 208–219. [Online]. Available: <https://doi.org/10.1145/3663529.3663841>
- [26] V.-H. Le and H. Zhang, "Log parsing: How far can chatgpt go?" in *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '23. IEEE Press, 2023, p. 1699–1704. [Online]. Available: <https://doi.org/10.1109/ASE56229.2023.00206>
- [27] Y. Xiao, V.-H. Le, and H. Zhang, "Demonstration-free: Towards more practical log parsing with large language models," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 153–165. [Online]. Available: <https://doi.org/10.1145/3691620.3694994>
- [28] E. Malul, Y. Meidan, D. Mimran, Y. Elovici, and A. Shabtai, "Genkubesecc: Llm-based kubernetes misconfiguration detection, localization, reasoning, and remediation," 2024. [Online]. Available: <https://arxiv.org/abs/2405.19954>
- [29] K. Sarda, Z. Namrud, M. Litoiu, L. Shwartz, and I. Watts, "Leveraging large language models for the auto-remediation of microservice applications: An experimental study," in *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, ser. FSE 2024. New York, NY, USA: Association for Computing Machinery, 2024, p. 358–369. [Online]. Available: <https://doi.org/10.1145/3663529.3663855>
- [30] P. Las-Casas, A. G. Kumbhare, R. Fonseca, and S. Agarwal, "Llexus: an ai agent system for incident management," *SIGOPS Oper. Syst. Rev.*, vol. 58, no. 1, p. 23–36, 2024. [Online]. Available: <https://doi.org/10.1145/3689051.3689056>
- [31] Kubernetes, "Kubernetes," 2014. [Online]. Available: <https://github.com/kubernetes/kubernetes>
- [32] Google, "Skaffold," 2019. [Online]. Available: <https://github.com/GoogleContainerTools/skaffold>
- [33] D. Kikuta, H. Ikeuchi, and K. Tajiri, "Chaos eater: Fully automating chaos engineering with large language models," 2025. [Online]. Available: <https://arxiv.org/abs/2501.11107>
- [34] Weaveworks, "Sock shop," 2023. [Online]. Available: <https://github.com/microservices-demo/microservices-demo>
- [35] OpenAI, "Gpt-4o system card," 2024. [Online]. Available: <https://arxiv.org/abs/2410.21276>
- [36] Anthropic, "Claude 3.5 sonnet model card addendum," 2024.
- [37] Gemini Team, "Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context," 2024. [Online]. Available: <https://arxiv.org/abs/2403.05530>