AIOpsLab: A Holistic Framework for Evaluating AI Agents for Enabling Autonomous Cloud

Yinfang Chen
UIUC
Champaign, USA

Manish Shetty
UC Berkeley
Berkeley, USA

Gagan Somashekar
Microsoft
Redmond, USA

Minghua Ma
Microsoft
Redmond, USA

Yogesh Simmhan
Microsoft and IISc
Bengaluru, India

Jonathan Mace
Microsoft Research
Redmond, USA

Chetan Bansal
Microsoft
Redmond, USA

Rujia Wang
Microsoft
Redmond, USA

Saravan Rajmohan
Microsoft
Redmond, USA

Abstract

AI for IT Operations (AIOps) aims to automate complex operational tasks, such as fault localization and root cause analysis, to reduce human workload and minimize customer impact. Traditionally, DevOps tools and AIOps algorithms focus on different isolated operational tasks. However, the rapid recent advancements in Large Language Models (LLMs) and AI agents are revolutionizing AIOps by enabling end-to-end and multitask automation. In this paper, we envision a future where AI agents can seamlessly handle different types of operational tasks across the incident lifecycle stack and achieve an autonomous, self-healing cloud systems, creating a new paradigm termed Agent for Operations, i.e., AgentOps. To realize this vision, a holistic framework is needed to empower these agents. We contribute to this goal by proposing and implementing an evaluation framework, AIOpsLab, that allows users to design, build, and assess the effectiveness and performance of agents in cloud environments with a focus on microservice scenarios. This paper first discusses the essential requirements for such a holistic framework and proposes AIOpsLab, which orchestrates applications, injects fine-grained, realistic faults, and interfaces with agents. By evaluating state-of-the-art AIOps LLM agents within AIOpsLab, we provide an analysis that offers valuable insights into their capabilities and limitations in handling complex operational tasks in cloud environments.

1 Introduction

The rapid evolution of IT applications and services has entered an era where enterprises increasingly rely on complex cloud-based hyper-scale cloud systems. Further, the adoption of the microservices architecture and serverless computing has enabled faster development and scaling by decomposing monolithic applications into independently deployable services. This flexibility allows developers to leverage cloud elasticity for more rapid innovation than ever before.

However, these architectures introduce significant challenges in managing and ensuring system reliability. The decentralization and statelessness that enhance scalability also create operational complexity. In such distributed environments, an issue can cascade into widespread outages due to complex dependencies, as demonstrated by a recent Amazon outage that cost an estimated $100 million in just one hour [55].

The burden of maintaining the seamless operation of these complex cloud environments primarily rests on the shoulders of Site Reliability Engineers (SREs) and DevOps professionals. These experts are responsible for developing, deploying, monitoring, and maintaining cloud services, often under immense pressure to minimize downtime and ensure optimal performance. Incident management typically involves multiple stages: detection, triaging, root cause analysis, and mitigation, as shown in Figure 1. Beyond these reactive measures, there is an increasing emphasis on proactive strategies such as fault prediction and preventive maintenance to avert failures before they impact the system.

Despite the proliferation of incident management tools, engineers are often overwhelmed by the sheer volume of data and the complexity of decision, making required in large-scale cloud environments. This challenge is further compounded by the trend toward real-time observability, where continuous monitoring generates vast amounts of telemetry data that must be rapidly analyzed and acted upon. The interplay of numerous microservices and the dynamic nature of cloud environments make traditional manual approaches untenable.

To address these challenges, there is a growing movement toward the adoption of AIOps (Artificial Intelligence for IT Operations). AIOps leverages AI and machine learning to automate and enhance IT operations, from monitoring and anomaly detection to fault diagnosis and recovery. The ultimate goal is to create Autonomous Clouds, where AI-driven agents can detect, localize, and mitigate faults with minimal human intervention. The concept of self-healing clouds is not new [14, 35], having been proposed over a decade ago, but the maturity of AI technologies and the emergence of AIOps have brought this vision closer to reality [16, 22, 36, 40, 49, 50, 56, 59, 61, 63].

We envision a future that extends beyond traditional AIOps capabilities, a new paradigm we term *AgentOps*. In this paradigm, AI agents can not only work on different isolated operational tasks, but also seamlessly handle multiple, cross-layer tasks across the entire operational stack. AgentOps represents an evolution where autonomous agents are not just tools but integral components of cloud operations, capable of making real-time decisions and taking actions to ensure system reliability and performance. These agents would foster truly autonomous, self-healing cloud systems that can adapt to changes and recover from failures without human intervention.

However, the journey toward fully realizing AgentOps is fraught with challenges, particularly in the design, development, and evaluation of AI-driven agents capable of handling complex operational tasks. Existing AIOps approaches often focus only isolated aspects of the incidents lifecycle, such as anomaly detection or fault localization, without having a cohesive framework that can evaluate these approaches and help decision making process to chain these algorithms or choose which agent is suitable for which scenario. Moreover, recent initiatives to leverage AIOps for cloud operations have been impeded by the reliance on proprietary services and datasets, which hampers reproducibility and the generalization of research outcomes. Other approaches employ ad hoc benchmarks and static metrics that fail to capture the dynamic and complex nature of real-world cloud services. Most of the benchmarks are text-based without enabling the agent to interact with the system environment. Additionally, the absence of standardized metrics and taxonomies for evaluating operational tasks creates difficulties in comparing the effectiveness of different AIOps agents.

These challenges highlight the need for a standardized and principled framework for testing and evaluating AIOps agents. Such a framework should enable realistic and reproducible interactions with operational tasks, allowing researchers and practitioners to benchmark their solutions against a common set of criteria. It should be flexible enough to accommodate new applications, workloads, and fault scenarios, ensuring its applicability in a rapidly evolving technological landscape. Crucially, the framework should facilitate the iterative improvement of AIOps agents by providing rich observability data and serving as a training environment where agents can learn from simulated operational tasks and refine their decision-making processes [62]. While existing tools address individual components of the AIOps lifecycle, such as observability and introspection [21, 48], application suites [15, 32], chaos engineering and fault injection [9, 10, 45], and agent-computer interfaces [57], they lack the integration necessary to support the development of holistic AIOps solutions.

Summary. This paper makes the following contributions:

- We unravel the requirements and challenges of a comprehensive framework that supports the design, development, and evaluation of autonomous AIOps agents;

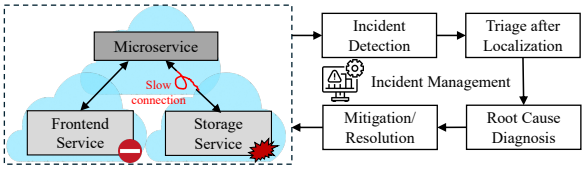

Figure 1: Cloud Incident Management Lifecycle.

- We develop a framework - AIOpsLab that combines a fault injector, workload generator, cloud-agent orchestrator, and telemetry observer to simulate production incidents and provide an agent-cloud interface for orchestrating and evaluating AIOps agents;
- We integrate 4 AIOps algorithms and 5 agents to demonstrate the application of our preliminary framework in evaluating an LLM-based agent with different types of AIOps tasks.
- We will make AIOpsLab publicly available.

2 Background and Motivation

We discuss the gaps between current AIOps approaches and existing benchmarks as the background and motivation for our work. Ideally, all of the approaches including the LLM-based AIOps agents should be evaluated in a standardized benchmark that allows easy comparison for different types of operation tasks.

2.1 AIOps Approaches

Our AIOps approaches to discuss include: *traditional AIOps approaches*, *human-driven operations*, and *LLM-based agents*.

Traditional AIOps Approaches. These approaches leverage machine learning models, statistical analysis, and rule-based automation to perform tasks like anomaly detection, log analysis, and predictive maintenance. They have proven effective at processing large amounts of structured data and identifying patterns that indicate potential issues. However, they typically function as passive tools, capable of providing insights but requiring human intervention for decision-making and mitigation.

Human-driven Operations. Incident management in cloud systems has been heavily reliant on human operators. These operators use their expertise, intuition, and manual workflows to identify, diagnose, and resolve incidents. While human-driven operations are adaptable and capable of understanding complex situations, they are inherently limited by scalability and response time.

LLM-based Agents. Recent advancements in large language models, particularly those augmented with tool usage and reasoning capabilities, have introduced a new type of AIOps approach: LLM-based agents [43, 47]. These agents go beyond language models by integrating external tools such as retrieval

systems, code interpreters, and web search engines, etc. This allows them to dynamically interact with their environment [53], enabling them to not only detect and analyze incidents but also autonomously manage the entire incident lifecycle. LLM-based agents can perform tasks across different layers of the system stack, from monitoring and anomaly detection to root cause analysis and corrective actions, providing an end-to-end solution for incident management. Their ability to adapt in real-time and collaborate with other agents or human operators makes them potentially suited for managing complex cloud infrastructures.

2.2 AIOps Benchmarks

Current benchmarks for AIOps approaches suffer from several key limitations, making it challenging to effectively evaluate and compare different AIOps approaches.

Existing benchmarks are mostly static, providing datasets such as Key Performance Indicators (KPIs) or system metrics [20, 28], i.e., time series data, or fixed question-answer format [38] that often do not reflect the dynamic nature of real-world cloud environments. These static benchmarks can be limited in fidelity, failing to replicate the unpredictable and evolving conditions found in production systems and incident management process. As a result, the evaluations may not accurately assess the robustness of AIOps agents in realistic settings.

Second, there is also a lack of clear task definitions in existing benchmarks, particularly concerning the types of faults introduced. Though there are many fault injection tools for testing the resilience of cloud systems [1, 4–6, 8, 11, 13, 19, 26, 31, 34, 39, 41, 42, 44, 46, 51, 60], many benchmarks focus solely on injecting system symptoms with existing fault injector [9, 10], such as pod failure, high CPU utilization or network latency. These coarse-grained faults existing in the existing benchmarks can only cause disruption without modeling the underlying, fine-grained root causes [11, 18, 34, 39, 44, 51]. They cannot evaluate true capabilities of AIOps agents to diagnose and mitigate root causes, limiting people to understand how well these agents perform at different types of the incident tasks. However, to evaluate AIOps agents effectively, the failure scenarios must go beyond simple performance or crash failures. Instead, they should reflect realistic cases that challenge the agents' ability.

Also, current benchmarks lack customization and extensibility, making them unsuitable for diverse user scenarios. Real-world incidents can vary significantly across different cloud environments and application domains. This lack of adaptability prevents users from tailoring benchmark scenarios to reflect their specific operational challenges.

To fully realize the potential of LLM agents in real-world AIOps, a standardized framework to easily evaluate and benchmark these agents in an online fashion is needed. By providing a common platform for evaluation, the framework would facilitate the comparison of different agents and approaches, drive advancements in the field, and accelerate the adoption of LLM agents in cloud operations.

3 AIOpsLab

In this section, we discuss the AIOpsLab design principles that can lead to a holistic and standardized framework to evaluate different agents.

3.1 Overview of AIOpsLab Design

In designing AIOpsLab as an evaluation framework for AIOps approaches, one of the central goals is to create a robust and flexible interface between agents and the cloud. The interface must accommodate a variety of users, from human engineers, AIOps algorithms to LLM agents. AIOpsLab should ensure effective communication, action-taking, and feedback from the cloud systems. Yang et al. [30] introduce *agent-computer-interface* for coding agents, inspired by human-computer interaction (HCI) principles, suggesting that agents should have a simplified interfaces for tasks. Here, we posit that agents should have a similar simplified interface to interact with services and clouds—the *Agent Cloud Interface (ACI)*.

We propose Agent-Cloud Interface (ACI) as the core component that facilitates this interaction. It acts as an orchestrator, simplifying the complexities of cloud operations for agents, enabling them to interact seamlessly with cloud services, detect and resolve issues, and adapt to dynamic environments. The following design decisions shape the ACI's structure and functionality.

Unified Interaction with Cloud Services. The ACI is designed to present a consistent, unified interface for interacting with cloud services. Whether it is a human operator or an automated AIOps tool, the interaction model remains the same, through clearly defined, documented APIs. These APIs expose key actions, such as retrieving logs, gathering metrics, executing commands, or scaling services. By standardizing these interactions, the ACI ensures that agents, regardless of their type, can engage with cloud systems without needing deep domain-specific knowledge of how individual services are implemented.

Automated Action-Oriented Feedback Another design aspect of the ACI is its ability to provide meaningful and actionable feedback for every action taken by the agent. After each request, whether successful or not, the system generates detailed feedback, including error messages, system responses, and output logs. This feedback loop ensures that agents can adjust their actions based on real-time data, learning from each step in the process.

Simplified Action Space. To make the interface more accessible to different types of agents, the ACI simplifies the action space into a clear and manageable list of valid API calls. These APIs encapsulate common cloud operations, e.g., Kubernetes commands. By reducing the complexity of available actions, the ACI makes cloud operations more approachable, particularly for AI agents that rely on a well-structured and clear

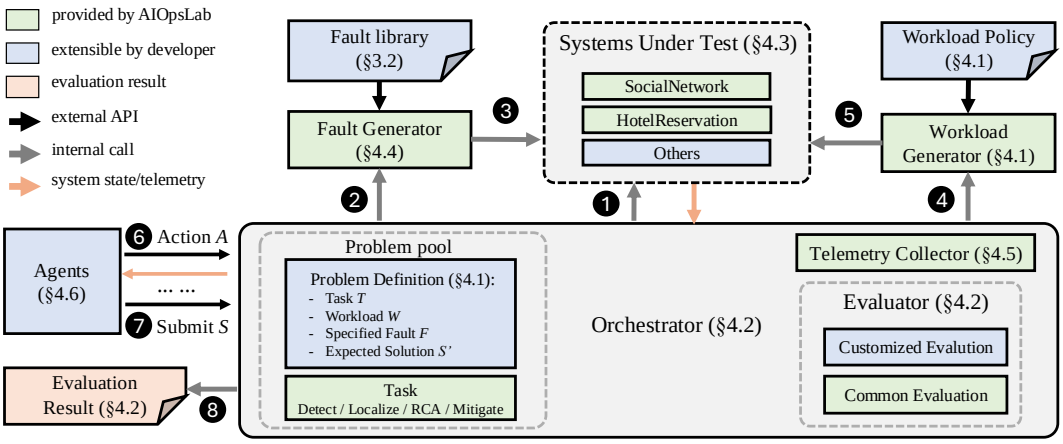

Figure 2: Overview of AIOpsLab. The Orchestrator coordinates interactions between various system components and serves as the Agent-Cloud-Interface (ACI). Agents engage with the Orchestrator to solve tasks, receiving a problem description, instructions, and relevant APIs. The Orchestrator generates diverse problems using the Workload and Fault Generators, injecting these into applications it can deploy. The deployed service has observability at multiple layers, providing telemetry, traces, and logs. The Orchestrator communicates with the service and the cloud using several tools such as `Kubernetes`, `Helm`, and even a `Shell`. Agents act via the Orchestrator, which executes them and updates the service's state. The Orchestrator evaluates the final solution using predefined metrics for the task.

action space. This design allows agents to focus on solving problems effectively without getting bogged down in technical details or irrelevant tasks, improving their overall performance in managing cloud environments.

Orchestration Between Agent and Cloud. At its core, the ACI serves as an orchestrator between the agent and the cloud environment, handling interactions and enforcing boundaries. Agents do not have direct control over the cloud infrastructure but instead interact through the ACI, which ensures that all requests are validated and that actions are performed safely.

The ACI acts as a safeguard, preventing agents from causing unintended disruptions in cloud services. By mediating agent requests, the ACI ensures that agents operate within a controlled environment, maintaining the integrity of cloud services even when managing complex tasks such as fault detection and resolution in real-time. This orchestration ensures a balanced approach to agent autonomy while maintaining cloud system reliability and safety.

3.2 Evaluation Scenarios of AIOpsLab

To effectively assess different AIOps approaches, we design evaluation scenarios that replicate realistic incidents. These scenarios involve deliberately introducing faults at various system levels, simulating the diverse operational challenges that AIOps tools and human operators must address.

We present a fault level taxonomy that categorizes faults according to different stages of the incident management lifecycle, with progressively increasing complexity. Our claim is that the higher level the fault is, the harder of the task the fault can provide to evaluate the agents. This taxonomy provides a structured framework for evaluating how AIOps approaches, be they AI agents, traditional algorithms, or human teams, detect, localize, analyze, and mitigate issues in complex cloud environments. By defining faults at multiple levels, we simulate a wide range of real-world challenges, from simple anomaly detection to advanced tasks like mitigation. Table 1 summarizes the levels of fault complexity.

Table 1: Fault Taxonomy for AIOps Agent Evaluation.

Level	Fault Type	Evaluation Focus
1	Detection	Can the approach accurately detect anomalies or deviations?
2	Localization	Can the approach pinpoint the exact source of a fault, e.g., a specific microservice?
3	Root Cause Analysis	Can the approach determine the underlying cause of the fault?
4	Topology	Can the approach identify and analyze multiple concurrent faults and their interdependencies?
5	Mitigation	Can the approach give effective solutions to recover the environment?

Level 1: Detection Ability. The foundational requirement for any AIOps approach is the ability to detect anomalies. Level 1 focuses on the preliminary identification of unusual behavior within the system. For example, detecting a sudden spike in CPU usage or an unexpected drop in network throughput indicates deviations from normal operations. Effective detection serves as the first line of defense, enabling the cloud systems to mark the symptoms and flag potential problems.

Level 2: Localization Ability. Once an anomaly is detected, the next step is to localize the fault within the system. Level

2 assesses the ability to accurately determine which component, such as a specific microservice, is malfunctioning. For instance, if a particular service is down, the approach should pinpoint that service rather than attributing the issue to the entire system. Effective localization is crucial for prompt and precise intervention, reducing the time and resources spent on diagnosing issues.

Level 3: Root Cause Analysis Ability. Level 3 evaluates whether the approach can diagnose why a service or component has failed. For example, if a microservice crashes due to an unhandled exception, the method should identify the fault type, i.e., code defect, causing the failure. This level challenges the approach to go beyond surface symptoms and provide insights into the root problem.

Level 4: Handling of Multiple Concurrent Faults. In complex cloud environments, multiple faults can occur simultaneously. Level 4 shifts the focus to detecting and understanding these concurrent faults, which may or may not be related. For example, a hardware failure in one server and a software bug in a microservice could happen at the same time, collectively leading to significant system disruption. This level evaluates whether the approach can identify multiple faults occurring concurrently and assess their combined impact on the system.

Level 5: Resolution and Mitigation. The final level assesses the ability to take effective corrective actions. Level 5 evaluates whether the approach can resolve issues by applying appropriate solutions. Key factors include the automation of the resolution process (if applicable) and the specificity and actionability of the proposed solutions. For example, if the root cause is a misconfigured microservice, the approach should adjust its configuration to restore normal operations of that specific microservice instead of re-configure everything. It is crucial to resolve the issue without introducing side effects, such as causing additional faults elsewhere in the system. Additionally, the agent must avoid getting stuck in a loop or taking an excessive amount of time, which could lead to non-termination.

4 Implementation and Use Cases

In this section, we introduce how we implement AIOpsLab, and show how AIOpsLab can be easily used with examples.

4.1 Problem Definition

To support a wide range of AIOps problems in AIOpsLab, we first formalize an AIOps problem instance P as a tuple: $P = (T, C, S)$, where T represents a *task*, C represents a *context*, and S represents the *solution*. The task T defines the specific AIOps operation to be performed, categorized into four types: detection, localization, (root cause) analysis, and mitigation. These tasks are defined as in Table 1. Each task type is associated with a set of success criteria and evaluation metrics. For instance, the detection task employs the time-to-detect (TTD) metric to measure the time taken to detect a fault.

The context C can be further formalized as a tuple: $C = (E, I)$, where E is the *operational environment* in which the problem occurs and I is the *problem information* used to describe the problem to the agent. The operational environment includes the cloud service, the fault model, and the workload model used to generate the problem instance; This is not shared with the agent and is used to evaluate the agent's performance. The problem information comprises information such as service description, task descriptions, and documentation about available APIs that is directly share with the agent. It also subsumes indirect information (including logs, metrics, and traces observed in the operational environment) that is queryable by the agent at runtime. Finally, S is the expected outcome of the task, which is used to evaluate the agent's performance. The solution is typically problem and task specific, and is carefully designed for evaluation. Note that some problems (e.g., mitigation tasks) can be solved in multiple ways. In such cases, AIOpsLab evaluates the state of the entire system after the problem is resolved, rather than focusing solely on the targeted resource where the fault was injected, because other services or resources may have been inadvertently affected during the mitigation process.

Example 4.1. Consider a problem to localize a Kubernetes target port misconfiguration in a microservices-based social network application. AIOpsLab makes it easy to define this problem instance in just a few lines by extending the `LocalizationTask` interface.

```python
from aiopslab import LocalizationTask, SocialNetwork
from aiopslab import Wrk, VirtFaultInjector

class K8STargetPortMisconf(LocalizationTask):
    def __init__(self):
        self.app = SocialNetwork()
        self.ans = "user-service"

    def start_workload(self):
        wrk = Wrk(rate=100, duration=10)
        wrk.start_workload(url=self.app.frontend_url)

    def inject_fault(self):
        inj = VirtFaultInjector(self.app.ns)
        inj.inject([self.ans], "misconfig_k8s")

    def eval(self, soln, trace, duration):
        res["TTL"] = duration
        res["success"] = is_exact_match(soln, self.ans)
        return res
```

Here the task T is localization and the solution S is the service named "user-service". The context C includes the social network application, a misconfiguration fault from AIOpsLab's fault library, and a standard workload using the `wrk` tool. AIOpsLab provides several such interfaces for all major AIOps tasks (Table 1) and allows users to add new problems by extending them. Once problems are defined, AIOpsLab can instantiate them and allow agents to interact with them using an Orchestrator that we describe next.

4.2 Orchestrator

AIOpsLab strictly enforces the separation of concerns between the Agent and the Service, using a well-defined central

piece we call the Orchestrator. It provides a robust set of interfaces that allow seamless integration and extension of various system components.

4.2.1 Agent Cloud Interface

A key responsibility of the Orchestrator is to provide a well-defined interface for the Agent to interact with the cloud environment. Typically, developers operate clouds and services with various programming (e.g., APIs, CLIs) and user interfaces (incident portals, dashboards etc.). However, existing interfaces to the cloud are not well-designed for LLMs and agents. For instance, humans can reliably ignore irrelevant information, but the same can prove distracting for agents and hamper performance.

The Orchestrator acts as the ACI, as mentioned in Section 3.1, and specifies (1) the set of valid actions available to the agent, and (2) how the service's state is conveyed back to the agent as the observation of its actions. In doing so, the Orchestrator abstracts the complexity of the cloud environment, and simplifies the agent's decision-making process. The ACI is designed to be intuitive and easy to use, with a concise list of APIs, each documented to ensure that agents can make meaningful progress towards objectives. Some APIs that AIOpsLab provides by default include `get_logs` (fetch logs from a service), `get_metrics` (fetch metrics from prometheus), `get_traces` (fetch traces from Jaeger), and `exec_shell` (execute shell commands).

Example 4.2. In this simplified example, we illustrate how the ACI is defined in AIOpsLab as APIs that agents can use.

```python
class TaskActions:
    def get_logs(ns: str, serv: str) -> str:
        """Collects log data from a pod using Kubectl.
        Args:
            ns (str): The K8S namespace.
            serv (str): The name of the service.
        Returns:
            str: Logs from the service.
        """
        try:
            pod = KubeCtl().get_pod_name(ns, f"app={serv}")
            return KubeCtl().get_pod_logs(pod, namespace)
        except Exception:
            return "Error: Service/namespace does not exist."

    def get_traces(ns: str, duration: int = 5) -> str:
        """
        Collects trace data from the service using Jaeger.
        Args:
            ns (str): The K8S namespace.
            duration (int): Duration to collect traces.
        Returns:
            str: Path to the directory where traces saved.
        """
        trace_api = TraceAPI(ns)
        end_t = datetime.now()
        start_t = end_t - timedelta(duration)
        traces = trace_api.extract_traces(start_t, end_t)
        return trace_api.save_traces(traces)
```

As shown, the ACI encapsulates complex operations behind simple APIs like `get_logs` and `get_traces`. On initializing a problem, the Orchestrator automatically extracts documentation from these APIs to provide as *context* (Section 4.1) to the agent. At runtime, agents can specify a wide range of actions on the service (e.g., scaling, redeploying, patching) by way of the Orchestrator's privileged access. Finally, the Orchestrator conveys the service's state after each action with high-quality feedback to the agent, including outputs, error messages, and tracebacks. This design maintains a consistent interface across different services, allowing agents to be easily ported across different problems.

4.2.2 Session Interface

Another key responsibility of the Orchestrator is to manage the lifecycle of the agent and the service. We implement the Orchestrator as a session-based system, where a `Session` is created for each instance of an agent solving a problem. Agents can be registered with the Orchestrator, and a session can be started with simple API calls passing a unique problem identifier. AIOpsLab is designed to be highly flexible and integratable with the growing LLM and agent framework space. Our only requirement is that the agent must implement a `get_action` method with the following signature: `async def get_action(state: str) -> str`. It takes the service's state as input from the Orchestrator and returns the next action the agent wants to take. Note, this could simply be a wrapper function around any existing agent framework.

Example 4.3. In this simplified example, we illustrate how an Agent can be onboarded to AIOpsLab.

```python
from aiopslab import Orchestrator

class Agent:
    def __init__(self, prob_desc, instructs, apis):
        self.prompt = self.set_prompt(prob_desc, instructs, apis)
        self.llm = GPT4()

    async def get_action(self, state: str) -> str:
        return self.llm.generate(self.prompt + state)

#initialize the orchestrator
orch = Orchestrator()
pid = "misconfig_app_hotel_res-mitigation-1"
prob_desc, instructs, apis = orch.init_problem(pid)

#register and evaluate the agent
agent = Agent(prob_desc, instructs, apis)
orch.register_agent(agent, name="myAgent")
asyncio.run(orch.start_problem(max_steps=10))
```

As shown on initializing a problem, the Orchestrator shares *context* necessary for the agent to solve the problem. It then iteratively polls the agent (via `get_action`) for its next action.

4.2.3 Other Interfaces

Problem Initializers. As described in Section 4.1, each problem is defined with a *context* which includes its operational environment. This environment is the service, fault, and workload conditions under which the problem occurs. Here, the Orchestrator deploys services and uses infrastructure-as-code tools like `Helm` [23] to deploy the required cloud service for each problem. We describe services already integrated into AIOpsLab in Section 4.3.

As shown in Figure 2, to create realistic benchmarks scenarios, the Orchestrator then interfaces with two pieces: (1) a *workload generator* and (2) a *fault generator*. These generators are responsible for introducing controlled service disruptions,

simulating live benchmark problems. As the workload generator, AIOpsLab currently uses the `wrk2` tool [15] which supports several workload policies and also replaying industry workloads. However, the AIOpsLab is extensible to other workload generators. For fault generation, AIOpsLab uses a custom fault library that instantiates faults across different levels of the system stack such as application and virtualization. The library contains and extends to several fine-grained and parametric faults that go beyond surface-level symptoms and engage deeper and complex resolution strategies. We describe the fault library in detail in Section 4.4.

Problem Evaluators. Finally, the Orchestrator plays a critical rols in evaluating the Agent's performance on a problem. It compares the Agent's solutions against predefined success criteria and evaluation metrics specific to each task. AIOpsLab supports several default and common metrics for each task (e.g., Time-to-detect for detection, number of steps taken, and tokens produced by an LLM-powered agent). Additionally, AIOpsLab provides qualitative evaluation of agent trajectories using LLMs-as-Judges [64]. Beyond that, all user-defined evaluation metrics specific to the problem instance are run. For instance, for the localization problem in Example 4.1, the metric `success` is defined by the agent's submission matching the fault microservice's name. Lastly, the Orchestrator maintains comprehensive logs of all agent trajectories, including actions taken and resulting system states, facilitating detailed analysis and debugging.

4.3 Services

AIOpsLab deploys live services to simulate production services. AIOpsLab is currently integrated with the HotelReservation and SocialNetwork from DeathStarBench [15] and the Google's Online Boutique application [17]. The SocialNetwork application has 28 microservices that together implement several features of real-world social networking applications. The constituent microservices are Nginx, Memcached, MongoDB, Redis, as well as microservices that implement the logic of the application. The constituent microservices are similar to the ones in the social networking application. The HotelReservation application, implemented using Go and gRPC, supports services like reserving hotels, recommending hotels based on user profiles, etc. The online boutique is a web-based e-commerce application implemented as a cloud-first microservices application. It supports activities like browsing items, adding them to the cart and purchasing them. It consists of 11 microservices implemented using different languages that communicate via gRPC.

4.4 Fault Library

To instantiate faults across different levels, we classify them into two main types, symptomatic faults and functional faults, as shown in Figure 4.

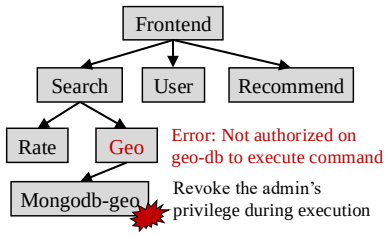

Figure 3: Revoke authentication fault example. Injection happens at Mongodb-geo service, while Geo service will be abnormal and generate error logs.

4.4.1 Symptomatic Faults

Symptomatic faults, such as performance degradation and crash failures, manifest as observable symptoms, like increased latency, resource exhaustion, or service outages. These faults typically satisfy Level 1 and Level 2 in the fault taxonomy (Table 1), which can help construct scenarios that to evaluate AIOps approaches' detection and localization ability. These faults provide an overview of potential problems but don't necessarily reveal the deeper, underlying reason of issues (since it does not have one), e.g., the root cause behind the pod failure.

AIOpsLab integrates existing fault injection tools, e.g., Chaos-Mesh [9], to inject symptomatic faults to various components within microservices. AIOpsLab currently supports five types of symptomatic faults to including CPU stress, memory stress, pod failure, request abort, and request delay.

4.4.2 Functional Faults

For Level 3 (Root Cause Analysis) faults, more fine-grained fault injection is required, which is where functional faults come into play. Functional faults, such as misconfigurations or critical software bugs, cause a system or service to fail. These faults require the AIOps approaches to not only detect and localize the failure but also diagnose the root cause, such as incorrect settings or code-level bugs, and apply the correct mitigation strategies. By using functional faults at the deeper levels of the taxonomy, we ensure that AIOps approaches especially the AI agents are tested on their ability to move beyond surface-level symptoms and engage in the more complex task of identifying and resolving the underlying causes of failures. Our fault injection methodology is designed to challenge agents across a wide range of scenarios, from application malfunctions to complex virtualization-level failures. We implement seven different categories of faults in both application and virtualization system levels. Note that users can apply these faults to different microservices to construct various evaluation test cases.

Example 4.4. In the following example, we illustrate the structure of the application-level fault injector for a revoke authentication fault and its usage example in AIOpsLab.

```
from aiopslab.generators.fault.base import FaultInjector
from aiopslab.service.apps.hotelres import HotelReservation
```

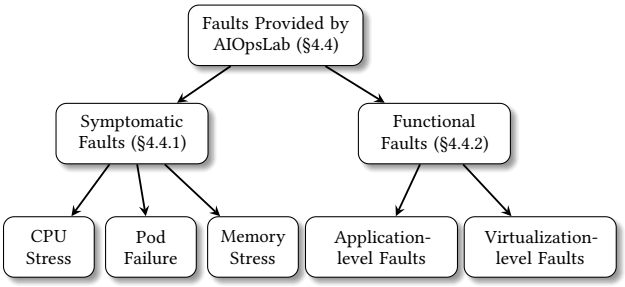

Figure 4: Fault categories to construct evaluation scenarios in AIOpsLab.

```python
class ApplicationFaultInjector(FaultInjector):
    def inject_revoke_auth(self, microservices: list[str]):
        """Inject a fault to revoke MongoDB admin privileges."""
        ...
    def recover_revoke_auth(self, microservices: list[str]):
        """Recover the revoke admin privileges fault."""
        ...

# Usage Example
class MongoDBRevokeAuth:
    def __init__(self):
        self.app = HotelReservation()

    def inject_fault(self):
        injector = ApplicationFaultInjector(ns)
        injector._inject(["mongodb-geo"], "revoke_auth")
```

Users can define problems using the existing fault library. For instance, users can specify different faulty services or even construct a task that injects multiple faults to multiple services.

Users can also customize their own faults to generate various problem tuple P, as mentioned in Section 4.1. For instance, if the user provides a general functional fault that can be applied to all of the 28 SocialNetwork's microservices, then the number of the problems will be $28 * 4 = 112$ as evaluation scenarios. Lastly, AIOpsLab not only provides the injection function for its associated failure scenarios, but also offers the corresponding mitigation mechanism to recover the system from the erroreous state.

4.5 Observability

AIOpsLab is equipped with an extensible observability layer designed to provide comprehensive monitoring capabilities. AIOpsLabcollects a wide array of telemetry data, including (1) traces from Jaeger detailing the end-to-end paths of requests through distributed systems, (2) application logs formatted and recorded by Filebeat and Logstash, and (3) system metrics monitored by Prometheus. Note that AIOpsLabnot only supports data collection during the interaction with the LLM agents, but also is able to export the data offline to facilitate the other traditional AIOps approachs to test. Besides, AIOpsLab is designed to capture information from other dimension, e.g., codebase, configuration, and cluster information. Developer can also design and expose even low-level system information such as syscall logs to agents by using AIOpsLab's interface.

5 Experimental Setup

5.1 Research Questions

- How different AIOps algorithms and agent perform in our benchmarks?
- What kind of faults are challenging for AIOps algorithms and agents?
- How flexible and extensible is the AIOpsLab?

5.2 Evaluated Approaches

5.2.1 AIOps Algorithms

AIOps has been a popular research domain, even before LLMs, with several classical approaches at various stages of the operations lifecycle. In this work, we evaluate a few state-of-the-art solutions across the board on tasks in AIOpsLab.

MKSMC. MKSMC adopts a new unsupervised anomaly detection technique based on Monte Carlo sampling to detect anomalies and enhance the interpretability of anomaly detection in multivariate data.

RMLAD. RMLAD localize failure components by computing log anomaly scores with DeepLog and assessing the mutual information between metrics and log anomalies. Metrics are ranked based on their correlation with log anomalies to predict root causes, with modality fusion managed by correlation computations.

PDiagnose. PDiagnose handles diagnosis by conducting separate anomaly detection on metrics, call chains, and logs, using a voting system to aggregate results and identify root causes through thresholds and suspicious fields.

MicroCBR. MicroCBR uses fault fingerprints and spatio-temporal knowledge graphs, integrating fault fingerprints with system topology for fault classification. It performs single-modal anomaly detection, constructs new fault fingerprints, and applies hierarchical case-based reasoning, relying heavily on accurate fault fingerprints.

5.2.2 LLM based agents

GPT-w-Shell. GPT is a family of LLMs trained on a massive corpus of data on the web. This makes them capable to do a variety of AIOps tasks such as analyzing multi-source data and even writing code or commands to mitigate issues. In all the experiments that follow we use a leading language models from the GPT family [2]. Our naive baseline (GPT-w-Shell) is a LM with access to only a secure shell.

ReAct. ReAct (Reasoning and Acting) is a framework that extends the idea of chain-of-thought [54] reasoning by introducing an interleaved reasoning-and-acting paradigm. Reasoning traces help the model induce, track, and update action plans as well as handle exceptions, while actions allow it to interface

with external sources, such as the AIOpsLab environment, to gather additional information.

TaskWeaver. TaskWeaver is a code-first agent framework that interprets user requests through code snippets to plan and execute data analytics tasks. It coordinates various plugins as functions to perform tasks, enabling seamless and efficient execution of complex data analytics workflows.

5.3 Metrics

Correctness. This metric measures the accuracy of the agent's response to faults. It evaluates whether the agent successfully detected, localized, and resolved the faults as expected. For example, in a detection task, correctness would indicate whether the agent identified the presence of an anomaly, while in a root cause analysis task, it would assess whether the agent correctly diagnosed the system level and fault type.

Time/Latency/Steps. This metric evaluates the efficiency of the AIOps agent by measuring the time and steps taken to detect and mitigate faults. It includes two key submetrics: Time-to-Detect (TTD), which is the time elapsed from the occurrence of a fault to its detection, and Time-to-Mitigate (TTM), the time taken from detection to complete resolution of the fault. Additionally, the metric assesses overall system latency and the number of steps or actions required to mitigate the issue. Lower values in TTD, TTM, latency, or fewer steps indicate a more efficient and effective agent. Optimizing for these factors ensures quicker detection and recovery, minimizing the impact of faults on the system.

Cost. The cost metric evaluates the overall resource consumption incurred by the AIOps agent. This includes average computational costs such as CPU and memory usage, operational costs like system downtime or cloud resource expenditure, and the cost of external services like LLM token usage.

6 Experimental Results

6.1 Agent Evaluation

To validate the effectiveness of the AIOpsLab framework, we integrated various AIOps agents and evaluated their performance across a series of fault scenarios and tasks, including detection, localization, root cause analysis (RCA), and mitigation. The agents were tested on a range of fault scenarios in a Kubernetes-based microservices environment. Below, we present detailed evaluations for each task, supported by experimental results.

6.1.1 Detection

In the detection task, the AIOps agents were tasked with identifying faults in real-time from a set of predefined operational anomalies. This task is critical for ensuring that faults are promptly identified before they can impact system performance or availability. Table 2 captures the agents' performance on detecting different faults. There is clear performance gap between GPT-4 and GPT-3.5 when both are provided shell access. GPT-4 successfully completed 4 out of 7 tasks (57%), while GPT-3.5 failed to complete any tasks successfully. This disparity highlights the significant improvement in problem-solving capabilities offered by more advanced language models. TaskWeaver, also using GPT-4, showed mixed results, successfully completing 3 out of 7 tasks (43%) While it performed well on certain tasks, such as detecting Kubernetes target port misconfigurations, it struggled with others due to its inability to come up with the correct solution format. The ReAct agent, utilizing GPT-4, showed impressive performance, successfully completing 6 out of 7 tasks (86%). It demonstrated consistent success across various problem types, from Kubernetes misconfigurations to MongoDB authentication issues. This high correctness score suggests that the ReAct approach, which combines reasoning and acting, is particularly effective for AIOps detection tasks.

Agent	F1	F2	F3	F4	F5	F6	F7	Correctness (%)
GPT-w-Shell (GPT-4)	✗	✓	✗	✓	✓	✗	✓	57%
GPT-w-Shell (GPT-3.5)	✗	✗	✗	✗	✗	✗	✗	0%
ReAct (GPT-4)	✓	✓	✓	✓	✗	✓	✓	86%
TaskWeaver (GPT-4)	✗	✓	✗	✗	✓	✗	✓	43%
MKSMC	✗	✗	✓	✓	✗	✗	✗	29%

Table 2: Performance on detection task. F1–F7 are the faults against which the agents are evaluated.

Agent	Avg. Time (s)	Avg. Steps	Avg. Input Tokens	Avg. Output Tokens
GPT-w-Shell (GPT-4)	7	4	8,179	118
GPT-w-Shell (GPT-3.5)	17	20	2,932	590
ReAct (GPT-4)	32	11	6,886	785
TaskWeaver (GPT-4)	195	14	3,946	283
MKSMC	1	N/A	N/A	N/A

Table 3: Agent efficiency for detection task.

The efficiency metrics of various agents, as shown in Table 3, reveal significant differences in performance and approach. GPT-w-Shell (GPT-4) demonstrated the fastest average execution time at 7 seconds, while ReAct (GPT-4) took 32 seconds, and TaskWeaver (GPT-4) required 195 seconds on average. Token usage patterns varied, with GPT-w-Shell using more input tokens (8,179) but fewer output tokens (118) compared to ReAct's 6,886 input and 785 output tokens, suggesting ReAct's more detailed reasoning process. The average number of steps taken also differed, with GPT-w-Shell at 4, ReAct at 11, and TaskWeaver at 14, indicating more complex problem-solving processes for the latter two. A specific task comparison for Kubernetes misconfiguration highlighted these differences: GPT-w-Shell failed in 10 seconds with 4 steps, ReAct succeeded in 21 seconds with 10 steps, and TaskWeaver succeeded but took 160 seconds with 13 steps. The traditional MKSMC method, while significantly faster with an average execution time of 1 seconds, only correctly identified 2 out of 7 anomalies, demonstrating a trade-off between speed and accuracy in AI-driven versus traditional approaches.

6.1.2 Localization

Agent	Correctness (%)	Avg. Time (s)	Avg. Steps	Avg. Input Tokens	Avg. Output Tokens
GPT-w-Shell (GPT-4)	71%	8	5	4,912	134
GPT-w-Shell (GPT-3.5)	0%	10	20	180	335
ReAct (GPT-4)	57%	46	13	5,768	1,066
TaskWeaver (GPT-4)	29%	150	9	108,448	81,582
PDiagnose	14%	1	N/A	N/A	N/A
RMLAD	7%	2	N/A	N/A	N/A

Table 4: Performance for localization task.

Fault localization is the process of pinpointing the exact location of the fault within the microservices architecture. The agents were evaluated based on their ability to utilize system metrics and traces to accurately localize faults.

The localization task evaluation, as presented in Table 4, reveals significant variations in performance across different agents and models. GPT-w-Shell (GPT-4) demonstrates superior performance with the highest correctness score of 71% and the most efficient execution, requiring only 8 seconds and 5 steps on average. In contrast, GPT-w-Shell with GPT-3.5 failed to localize any faults, highlighting the substantial performance gap between GPT-3.5 and GPT-4 models. ReAct (GPT-4) shows competitive performance with 57% correctness but requires significantly more time (46 seconds on average) and steps (13) compared to GPT-w-Shell (GPT-4). TaskWeaver (GPT-4) demonstrates moderate correctness at 29% but exhibits the highest resource consumption, with an average execution time of 150 seconds and extensive token usage, particularly in output tokens (81,582 on average). Traditional methods like PDiagnose and RMLAD show faster execution times (1 and 2 seconds respectively) but achieve lower correctness (14% and 7% respectively) compared to the AI-driven approaches. This underscores the trade-off between speed and accuracy in fault localization tasks. The results suggest that there's still room for improvement in correctness and achieving a balance between correctness and efficiency across all agents. This comparison provides valuable insights for selecting and optimizing AI agents for AIOps localization tasks, demonstrating the effectiveness of the evaluation framework in highlighting performance differences across various approaches.

6.1.3 Root Cause Analysis

Agent	Correctness (%)	Avg. Time (s)	Avg. Steps	Avg. Input Tokens	Avg. Output Tokens
GPT-w-Shell (GPT-4)	14%	8	5	3,198	147
GPT-w-Shell (GPT-3.5)	0%	15	20	260	570
ReAct (GPT-4)	14%	23	7	4,466	609
TaskWeaver (GPT-4)	0%	265	15	32,608	410
MicroCBR	14%	1	N/A	N/A	N/A

Table 5: Performance for root cause analysis (RCA).

The root cause analysis (RCA) task aimed to identify the fundamental cause behind observed system faults by analyzing logs, metrics, and traces, correlating events, and deducing causal relationships. Table 5 presents the performance of different agents in completing this task, revealing notable differences in execution time and efficiency.

GPT-w-Shell (GPT-4), exhibited moderate overall performance, completing the task in an average of 8 seconds with 5 steps. Although relatively fast, its correctness remained at 14%, demonstrating limited accuracy. GPT-w-Shell (GPT-3.5), on the other hand, failed to identify any root cause successfully, often reaching the maximum step limit of 20, with an average execution time of 15 seconds. ReAct (GPT-4) performed worse than GPT-w-Shell (GPT-4) in terms of speed and steps, averaging 23 seconds and 7 steps, with the same correctness rate of 14%. Despite taking longer and requiring more steps, its accuracy did not improve over GPT-w-Shell (GPT-4). TaskWeaver, also based on GPT-4, had the longest execution times, averaging 265 seconds and 15 steps. Despite the extended time for processing, TaskWeaver was unable to solve any problem correctly. MicroCBR, a case-based reasoning approach, stood out for its speed, completing tasks in just 1 second on average. However, it failed to achieve better correctness rates, matching the 14% success of GPT-w-Shell and ReAct.

This comprehensive evaluation highlights the significant variability in agent performance for RCA tasks. While some models, like MicroCBR, excel in speed, others, like TaskWeaver, struggle despite extended analysis time. The AIOpsLab framework plays a crucial role in enabling this detailed comparison, providing insights into the trade-offs between speed, steps taken, and success rates for identifying root causes in complex systems.

6.1.4 Mitigation

Agent	Correctness (%)	Avg. Time (s)	Avg. Steps	Avg. Input Tokens	Avg. Output Tokens
GPT-w-Shell (GPT4)	43	127	13	9,313	1,038
GPT-w-Shell (GPT3.5)	0	19	20	1,298	1,139
ReAct (GPT4)	43	57	15	23,433	1,227
TaskWeaver (GPT4)	29	260	16	54,757	28,587

Table 6: Agent performance for mitigation task.

The mitigation task focused on the agents' ability to not only detect issues but also perform corrective actions in real-time. As shown in Table 6, the evaluation revealed considerable variations in performance across the tested agents.

GPT-w-Shell (GPT-4) and ReAct (GPT-4) both achieved the highest correctness at 43%, completing tasks with an average time of 127 and 57 seconds, respectively. ReAct was noticeably faster but required more steps (15 steps) compared to GPT-w-Shell (GPT-4)'s 13 steps. TaskWeaver (GPT-4) showed a lower success rate at 29%, with the longest average execution time of 260 seconds and the highest average input token usage at 54,757 tokens. GPT-w-Shell (GPT-3.5), while the fastest in terms of execution time (averaging just 19 seconds), failed to succeed in any of the scenarios. It consistently reached the maximum step count of 20, highlighting its struggles to

effectively resolve the mitigation tasks. In terms of input token usage, ReAct (GPT-4) processed 23,433 tokens on average, much more than GPT-w-Shell (GPT-4) with 9,313 tokens. TaskWeaver was an outlier, processing significantly more tokens on average, yet this didn't translate into better success rates. GPT-w-Shell (GPT-3.5), despite using only 1,298 tokens on average, faced context length limitations and often failed to complete the tasks effectively.

In summary, GPT-w-Shell (GPT-4) and ReAct (GPT-4) displayed similar levels of correctness, with ReAct standing out for its faster execution times. TaskWeaver's performance was hampered by long execution times and excessive input processing, while GPT-w-Shell (GPT-3.5) demonstrated speed but at the cost of success. This comparison highlights the varying trade-offs between speed, correctness, and processing efficiency across different agents.

7 Discussion

Ease of Use for Developers. One primary objective of AIOpsLab is to ensure that developers can easily create their own incident scenarios to evaluate the agents. By offering a user-friendly interface, we lower the barrier to entry, allowing developers to focus on crafting scenarios that best reflect incidents in their systems and define what ability their agents can have. For example, developers just need to register TaskWeaver in AIOpsLab and optionally provide the tools to use, such as the scripts for anomaly detection, which developers can easily modify or extend to fit their specific needs.

Challenges with LLM Agents. While LLM agents hold potential for automating cloud operational tasks, their current implementation often introduces unnecessary complexity or leads agents astray. A recurring issue is its redundancy. LLM agents could repeatedly perform tasks or pursue ineffective strategies, e.g., reading the microservices status via `kubectl`, that complicate the process rather than resolving it. In many cases, LLM agents lack the precision or task-specific optimization needed to effectively manage cloud systems. As generalized agents, they are designed to handle a broad range of tasks but often struggle with specialized problems that require detailed system knowledge or domain-specific actions.

8 Related Work

AIOps Agents. Recent advancements in cloud management handling have increasingly integrated large language models (LLMs) to enhance anomaly detection, localization, root cause analysis, and mitigation. Methods like fine-tuned GPT [3], RCACopilot [12], RCAgent [52], MonitorAssistant[58], and Xpert [29] exemplify the application of LLMs in monitoring and analyzing system behaviors, significantly improving the accuracy of these processes. However, a notable gap in the field is the lack of a comprehensive online evaluation benchmark for assessing performance across these various tasks.

Agent benchmarks. assess the general abilities of LLMs across various tasks. These tasks evaluate LLMs' capacity for logical reasoning, general knowledge, common sense, and other similar abilities rather than being confined to a particular domain. MMLU [24, 25] is a benchmark designed to measure knowledge acquired during pretraining by evaluating models exclusively in zero-shot and few-shot settings, covering 57 subjects across STEM. HELM [33, 37] offers a comprehensive evaluation of LLMs' capabilities across multiple dimensions especially text-to-image. BIG-bench [7] comprises 204 tasks spanning a wide array of topics, with a particular focus on tasks deemed beyond the reach of current LLMs. C-Eval [27] is a comprehensive Chinese evaluation suite designed to assess Chinese LLMs' advanced knowledge and reasoning abilities. However, all of these benchmarks are text-based and static., which do not fit the interactive and dynamic operation cloud environment.

9 Conclusion

In this paper, we unravel the requirements and challenges of a comprehensive framework that supports the design, development, and evaluation of autonomous AIOps agents; and develop a prototype framework - AIOpsLab that combines a fault injector, workload generator, cloud-agent orchestrator, and telemetry observer to simulate production incidents and provide an agent-cloud interface for orchestrating and evaluating AIOps agents. We have tested 4 AIOps algorithms and 5 agents to demonstrate the application of our preliminary framework in evaluating an LLM-based agent with different types of AIOps tasks.

References

[1] 2022. Jepsen. https://jepsen.io/.

[2] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774* (2023).

[3] Toufique Ahmed, Supriyo Ghosh, Chetan Bansal, Thomas Zimmermann, Xuchao Zhang, and Saravan Rajmohan. 2023. Recommending Root-Cause and Mitigation Steps for Cloud Incidents using Large Language Models. In *Proceedings of the 45th International Conference on Software Engineering (ICSE'23)*.

[4] Ramnatthan Alagappan, Aishwarya Ganesan, Yuvraj Patel, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2016. Correlated Crash Vulnerabilities. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)*.

[5] Ahmed Alquraan, Hatem Takruri, Mohammed Alfatafta, and Samer Al-Kiswany. 2018. An Analysis of Network-Partitioning Failures in Cloud Systems. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation (OSDI'18)*.

[6] Radu Banabic and George Candea. 2012. Fast Black-Box Testing of System Recovery Code. In *Proceedings of the 7th European Conference on Computer Systems (EuroSys'12)*.

[7] BIG bench authors. 2023. Beyond the Imitation Game: Quantifying and extrapolating the capabilities of language models. *Transactions on Machine Learning Research* (2023). https://openreview.net/forum?id=uyTL5Bvosj

[8] Marco Canini, Daniele Venzano, Peter Perešíni, Dejan Kostić, and Jennifer Rexford. 2012. A NICE Way to Test OpenFlow Applications. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI'12)*.

[9] Chaos Mesh Community. [n. d.]. ChaosMesh. https://chaos-mesh.org/. Accessed: 2024-07-08.

[10] ChaosBlade Team. [n. d.]. ChaosBlade. https://github.com/chaosblade-io/chaosblade. Accessed: 2024-07-08.

[11] Haicheng Chen, Wensheng Dou, Dong Wang, and Feng Qin. 2020. CoFI: Consistency-Guided Fault Injection for Cloud Systems. In *Proceedings of the 35th ACM/IEEE International Conference on Automated Software Engineering (ASE'20)*.

[12] Yinfang Chen, Huaibing Xie, Minghua Ma, Yu Kang, Xin Gao, Liu Shi, Yunjie Cao, Xuedong Gao, Hao Fan, Ming Wen, Jun Zeng, Supriyo Ghosh, Xuchao Zhang, Chaoyun Zhang, Qingwei Lin, Saravan Rajmohan, Dongmei Zhang, and Tianyin Xu. 2024. Automatic Root Cause Analysis via Large Language Models for Cloud Incidents. In *Proceedings of the Nineteenth European Conference on Computer Systems, EuroSys 2024, Athens, Greece, April 22-25, 2024*. ACM, 674–688. https://doi.org/10.1145/3627703.3629553

[13] Maria Christakis, Patrick Emmisberger, Patrice Godefroid, and Peter Müller. 2017. A General Framework for Dynamic Stub Injection. In *Proceedings of the 39th International Conference on Software Engineering (ICSE'17)*.

[14] Yuanshun Dai, Yanping Xiang, and Gewei Zhang. 2009. Self-healing and Hybrid Diagnosis in Cloud Computing. In *Cloud Computing*, Martin Gilje Jaatun, Gansen Zhao, and Chunming Rong (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg.

[15] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, et al. 2019. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 3–18.

[16] Vaibhav Ganatra, Anjaly Parayil, Supriyo Ghosh, Yu Kang, Minghua Ma, Chetan Bansal, Suman Nath, and Jonathan Mace. 2023. Detection Is Better Than Cure: A Cloud Incidents Perspective. In *Proceedings of the 31st Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*.

[17] Google. 2024. microservices-demo. https://github.com/GoogleCloudPlatform/microservices-demo.

[18] Jiawei Tyler Gu, Xudong Sun, Wentao Zhang, Yuxuan Jiang, Chen Wang, Mandana Vaziri, Owolabi Legunsen, and Tianyin Xu. 2023. 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)*.

[19] Haryadi S. Gunawi, Thanh Do, Pallavi Joshi, Peter Alvaro, Joseph M. Hellerstein, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Koushik Sen, and Dhruba Borthakur. 2011. Fate and Destini: A Framework for Cloud Recovery Testing. In *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI'11)*.

[20] Songqiao Han, Xiyang Hu, Hailiang Huang, Minqi Jiang, and Yue Zhao. 2022. Adbench: Anomaly detection benchmark. *Advances in Neural Information Processing Systems* 35 (2022), 32142–32159.

[21] Shilin He, Botao Feng, Liqun Li, Xu Zhang, Yu Kang, Qingwei Lin, Saravan Rajmohan, and Dongmei Zhang. 2023. STEAM: Observability-Preserving Trace Sampling. Association for Computing Machinery, New York, NY, USA, 1750–1761. https://doi.org/10.1145/3611643.3613881

[22] Shilin He, Xu Zhang, Pinjia He, Yong Xu, Liqun Li, Yu Kang, Minghua Ma, Yining Wei, Yingnong Dang, Saravanakumar Rajmohan, et al. 2022. An empirical study of log analysis at Microsoft. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*.

[23] Helm. 2024. *Helm: The package manager for Kubernetes*.

[24] Dan Hendrycks, Collin Burns, Steven Basart, Andrew Critch, Jerry Li, Dawn Song, and Jacob Steinhardt. 2021. Aligning AI With Shared Human Values. *Proceedings of the International Conference on Learning Representations (ICLR)* (2021).

[25] Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. 2021. Measuring Massive Multitask Language Understanding. *Proceedings of the International Conference on Learning Representations (ICLR)* (2021).

[26] Victor Heorhiadi, Shriram Rajagopalan, Hani Jamjoom, Michael K Reiter, and Vyas Sekar. 2016. Gremlin: Systematic Resilience Testing of Microservices. In *Proceedings of the IEEE 36th International Conference on Distributed Computing Systems (ICDCS'16)*.

[27] Yuzhen Huang, Yuzhuo Bai, Zhihao Zhu, Junlei Zhang, Jinghan Zhang, Tangjun Su, Junteng Liu, Chuancheng Lv, Yikai Zhang, Jiayi Lei, Yao Fu, Maosong Sun, and Junxian He. 2023. C-Eval: A Multi-Level Multi-Discipline Chinese Evaluation Suite for Foundation Models. In *Advances in Neural Information Processing Systems*.

[28] Vincent Jacob, Fei Song, Arnaud Stiegler, Yanlei Diao, and Nesime Tatbul. 2020. Anomalybench: An open benchmark for explainable anomaly detection. *CoRR abs/2010.05073* (2020).

[29] Yuxuan Jiang, Chaoyun Zhang, Shilin He, Zhihao Yang, Minghua Ma, Si Qin, Yu Kang, Yingnong Dang, Saravan Rajmohan, Qingwei Lin, and Dongmei Zhang. 2024. Xpert: Empowering Incident Management with Query Recommendations via Large Language Models. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*. ACM, 92:1–92:13. https://doi.org/10.1145/3597503.3639081

[30] Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. 2024. SWE-bench: Can Language Models Resolve Real-world Github Issues?. In *The Twelfth International Conference on Learning Representations*. https://openreview.net/forum?id=VTF8yNQM66

[31] Xiaoen Ju, Livio Soares, Kang G. Shin, Kyung Dong Ryu, and Dilma Da Silva. 2013. On Fault Resilience of OpenStack. In *Proceedings of the 12th ACM Symposium on Cloud Computing (SOCC'13)*.

[32] Varad Kulkarni et al. 2024. XFBench: A Cross-Cloud Benchmark Suite for Evaluating FaaS Workflow Platforms. In *24th IEEE/ACM international Symposium on Cluster, Cloud and Internet Computing (CCGRID)*.

[33] Tony Lee, Michihiro Yasunaga, Chenlin Meng, Yifan Mai, Joon Sung Park, Agrim Gupta, Yunzhi Zhang, Deepak Narayanan, Hannah Benita Teufel, Marco Bellagente, Minguk Kang, Taesung Park, Jure Leskovec, Jun-Yan Zhu, Li Fei-Fei, Jiajun Wu, Stefano Ermon, and Percy Liang. 2023. Holistic Evaluation of Text-To-Image Models. arXiv:2311.04287 [cs.CV] https://arxiv.org/abs/2311.04287

[34] Tanakorn Leesatapornwongsa, Mingzhe Hao, Pallavi Joshi, Jeffrey F. Lukman, and Haryadi S. Gunawi. 2014. SAMC: Semantic-Aware Model Checking for Fast Discovery of Deep Bugs in Cloud Systems. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI'14)*.

[35] Wenrui Li, Pengcheng Zhang, and Zhongxue Yang. 2012. A Framework for Self-Healing Service Compositions in Cloud Computing Environments. In *2012 IEEE 19th International Conference on Web Services*. https://doi.org/10.1109/ICWS.2012.109

[36] Zeyan Li, Junjie Chen, Rui Jiao, Nengwen Zhao, Zhijun Wang, Shuwei Zhang, Yanjun Wu, Long Jiang, Leiqin Yan, Zikai Wang, et al. 2021. Practical root cause localization for microservice systems via trace analysis. In *2021 IEEE/ACM 29th International Symposium on Quality of Service*.

[37] Percy Liang, Rishi Bommasani, Tony Lee, Dimitris Tsipras, Dilara Soylu, Michihiro Yasunaga, Yian Zhang, Deepak Narayanan, Yuhuai Wu, Ananya Kumar, Benjamin Newman, Binhang Yuan, Bobby Yan, Ce Zhang, Christian Cosgrove, Christopher D. Manning, Christopher Ré, Diana Acosta-Navas, Drew A. Hudson, Eric Zelikman, Esin Durmus, Faisal Ladhak, Frieda Rong, Hongyu Ren, Huaxiu Yao, Jue Wang, Keshav Santhanam, Laurel Orr, Lucia Zheng, Mert Yuksekgonul, Mirac Suzgun, Nathan Kim, Neel Guha, Niladri Chatterji, Omar Khattab, Peter Henderson, Qian Huang, Ryan Chi, Sang Michael Xie, Shibani Santurkar, Surya Ganguli, Tatsunori Hashimoto, Thomas Icard, Tianyi Zhang, Vishrav Chaudhary, William Wang, Xuechen Li, Yifan Mai, Yuhui Zhang, and Yuta Koreeda. 2023. Holistic Evaluation of Language Models. arXiv:2211.09110 [cs.CL] https://arxiv.org/abs/2211.09110

[38] Yuhe Liu, Changhua Pei, Longlong Xu, Bohan Chen, Mingze Sun, Zhirui Zhang, Yongqian Sun, Shenglin Zhang, Kun Wang, Haiming Zhang, et al. 2023. OpsEval: A Comprehensive Task-Oriented AIOps Benchmark for Large Language Models. *arXiv preprint arXiv:2310.07637* (2023).

[39] Jie Lu, Chen Liu, Lian Li, Xiaobing Feng, Feng Tan, Jun Yang, and Liang You. 2019. CrashTuner: Detecting Crash-Recovery Bugs in Cloud Systems via Meta-Info Analysis. In *Proceedings of the 26th ACM Symposium on Operating System Principles (SOSP'19)*.

[40] Minghua Ma, Shenglin Zhang, Dan Pei, Xin Huang, and Hongwei Dai. 2018. Robust and rapid adaption for concept drift in software system anomaly detection. In *2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 13–24.

[41] Rupak Majumdar and Filip Niksic. 2018. Why is Random Testing Effective for Partition Tolerance Bugs?. In *Proceedings of the 45th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL'18)*.

[42] Paul D Marinescu and George Candea. 2009. LFI: A Practical and General Library-Level Fault Injector. In *Proceedings of the 39th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'09)*.

[43] Grégoire Mialon, Roberto Dessì, Maria Lomeli, Christoforos Nalmpantis, Ram Pasunuru, Roberta Raileanu, Baptiste Rozière, Timo Schick, Jane Dwivedi-Yu, Asli Celikyilmaz, et al. 2023. Augmented language models: a survey. *arXiv preprint arXiv:2302.07842* (2023).

[44] Jayashree Mohan, Ashlie Martinez, Soujanya Ponnapalli, Pandian Raju, and Vijay Chidambaram. 2018. Finding Crash-Consistency Bugs with Bounded Black-Box Crash Testing. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation (OSDI'18)*.

[45] Netflix. [n. d.]. ChaosMonkey. https://github.com/Netflix/chaosmonkey. Accessed: 2024-07-08.

[46] Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Samer Al Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2014. All File Systems Are Not Created Equal: On the Complexity of Crafting Crash-Consistent Applications. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI'14)*.

[47] Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2024. Toolformer: Language models can teach themselves to use tools. *Advances in Neural Information Processing Systems* 36 (2024).

[48] Jesper Simonsson, Long Zhang, Brice Morin, Benoit Baudry, and Martin Monperrus. 2021. Observability and chaos engineering on system calls for containerized applications in docker. *Future Generation Computer Systems* 122 (2021), 117–129.

[49] Gagan Somashekar, Anurag Dutt, Mainak Adak, Tania Lorido Botran, and Anshul Gandhi. 2024. GAMMA: Graph Neural Network-Based Multi-Bottleneck Localization for Microservices Applications. In *Proceedings of the ACM on Web Conference 2024* (Singapore, Singapore) *(WWW '24)*. Association for Computing Machinery, New York, NY, USA, 3085–3095. https://doi.org/10.1145/3589334.3645665

[50] Gagan Somashekar, Anurag Dutt, Rohith Vaddavalli, Sai Bhargav Varanasi, and Anshul Gandhi. 2022. B-MEG: Bottlenecked-Microservices Extraction Using Graph Neural Networks. In *Companion of the 2022 ACM/SPEC International Conference on Performance Engineering* (Bejing, China) *(ICPE '22)*. Association for Computing Machinery, New York, NY, USA, 7–11. https://doi.org/10.1145/3491204.3527494

[51] Xudong Sun, Wenqing Luo, Jiawei Tyler Gu, Aishwarya Ganesan, Ramnatthan Alagappan, Michael Gasch, Lalith Suresh, and Tianyin Xu. 2022. Automatic Reliability Testing for Cluster Management Controllers. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI'22)*.

[52] Zefan Wang, Zichuan Liu, Yingying Zhang, Aoxiao Zhong, Lunting Fan, Lingfei Wu, and Qingsong Wen. 2023. Rcagent: Cloud root cause analysis by autonomous agents with tool-augmented large language models. *arXiv preprint arXiv:2310.16340* (2023).

[53] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Ed Chi, Quoc Le, and Denny Zhou. 2022. Chain of thought prompting elicits reasoning in large language models. *arXiv preprint arXiv:2201.11903* (2022).

[54] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems* 35 (2022), 24824–24837.

[55] Sean Wolfe. 2018. Amazon's one hour of downtime on Prime Day may have cost it up to $100 million in lost sales. (2018). https://www.businessinsider.com/amazon-prime-day-website-issues-cost-it-millions-in-lost-sales-2018-7

[56] Zhe Xie, Haowen Xu, Wenxiao Chen, Wanxue Li, Huai Jiang, Liangfei Su, Hanzhang Wang, and Dan Pei. 2023. Unsupervised Anomaly Detection on Microservice Traces through Graph VAE. In *Proceedings of the ACM Web Conference 2023*.

[57] John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. Swe-agent: Agent-computer interfaces enable automated software engineering. *arXiv preprint arXiv:2405.15793* (2024).

[58] Zhaoyang Yu, Minghua Ma, Chaoyun Zhang, Si Qin, Yu Kang, Chetan Bansal, Saravan Rajmohan, Yingnong Dang, Changhua Pei, Dan Pei, Qingwei Lin, and Dongmei Zhang. 2024. MonitorAssistant: Simplifying Cloud Service Monitoring via Large Language Models. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering, FSE 2024, Porto de Galinhas, Brazil, July 15-19, 2024*. ACM, 38–49. https://doi.org/10.1145/3663529.3663826

[59] Zhengran Zeng, Yuqun Zhang, Yong Xu, Minghua Ma, Bo Qiao, Wentao Zou, Qingjun Chen, Meng Zhang, Xu Zhang, Hongyu Zhang, Xuedong Gao, Hao Fan, Saravan Rajmohan, Qingwei Lin, and Dongmei Zhang. 2023. TraceArk: Towards Actionable Performance Anomaly Alerting for Online Service Systems. In *To appear in Proc. of ICSE*.

[60] Pingyu Zhang and Sebastian Elbaum. 2012. Amplifying Tests to Validate Exception Handling Code. In *Proceedings of the 34th International Conference on Software Engineering (ICSE'12)*.

[61] Qiao Zhang, Guo Yu, Chuanxiong Guo, Yingnong Dang, Nick Swanson, Xinsheng Yang, Randolph Yao, , Murali Chintalapati, Arvind Krishnamurthy, and Thomas Anderson. 2018. Deepview: Virtual Disk Failure Diagnosis and Pattern Detection for Azure. In *Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI'18)*.

[62] Xuchao Zhang, Supriyo Ghosh, Chetan Bansal, Rujia Wang, Minghua Ma, Yu Kang, and Saravan Rajmohan. 2024. 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* (Porto de Galinhas, Brazil) *(FSE 2024)*. Association for Computing Machinery, New York, NY, USA, 266–277. https://doi.org/10.1145/3663529.3663846

[63] Chenyu Zhao, Minghua Ma, Zhenyu Zhong, Shenglin Zhang, Zhiyuan Tan, Xiao Xiong, LuLu Yu, Jiayi Feng, Yongqian Sun, Yuzhi Zhang, et al. 2023. Robust Multimodal Failure Detection for Microservice Systems. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*.

[64] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, et al. 2024. Judging llm-as-a-judge with mt-bench and chatbot arena. *Advances in Neural Information Processing Systems* 36 (2024).