Agentic Failure Management of Cloud Systems

by

Yinfang Chen

Dissertation

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois Urbana-Champaign, 2026

Urbana, Illinois

Doctoral Committee:

Associate Professor Tianyin Xu, Chair
Professor Indranil Gupta
Doctor Minghua Ma, Microsoft
Associate Professor Lingming Zhang
Assistant Professor Minjia Zhang

Abstract

Cloud systems are becoming ever more critical—worldwide public cloud spending is expected to hit $723 billion in 2025. Yet failures are the norm in the cloud: outages or incidents happen every day, and downtime for large systems can run over $500,000 per hour. Despite massive investments in automation, today's cloud operations still rely heavily on human engineers.

This dissertation tackles a fundamental question: how can we embed intelligence into systems so they can autonomously detect, diagnose, and recover from failures safely, efficiently, and at scale? We answer this question by building systems that span the entire cloud incident lifecycle, from proactive prevention before incidents occur, to root-cause analysis once they happen, to a multi-agent system with safety guarantees for autonomous mitigation, alongside a benchmark that lets the research community compare agentic solutions.

In the pre-incident stage, where the goal is to prevent failures from reaching production, we present Rainmaker, a push-button reliability testing tool for cloud-backed applications. It intercepts and manipulates outbound REST API calls at the HTTP layer to inject realistic transient cloud faults. Rainmaker is built on a four-pattern bug taxonomy (no error handling, throwing unrelated exceptions, silent semantic violations, and state divergence) developed by analyzing how error handling goes wrong under the cloud-based fault model. Rainmaker turns systematic fault injection into a routine part of the development workflow.

However, real-world large-scale systems are almost never bug-free, and incidents still occur no matter how thorough the testing. In the post-incident stage, we introduce RCACopilot, an automated root-cause analysis system that adapts large language model (LLM) reasoning to cloud incident management. RCACopilot composes incident

handlers from three reusable actions (scope switching, query, and mitigation) to collect diagnostic information. It then integrates an LLM to predict the root-cause category and explain the reasoning behind the prediction.

Beyond diagnosis, we close the loop with mitigation. We present STRATUS, an LLM-based multi-agent system for autonomous Site Reliability Engineering (SRE). STRATUS orchestrates four specialized agents (detection, diagnosis, mitigation, and undo) through a deterministic state machine, with a formal safety specification we call *Transactional No-Regression* (TNR). TNR builds on classical transaction semantics to guarantee that any unsuccessful mitigation can be undone, and that the externally visible system severity never grows beyond the baseline at the time the failure was detected.

Finally, as LLMs and autonomous agents are increasingly integrated into cloud operations, the field has lacked a rigorous, reproducible benchmark to compare their effectiveness. We address this gap with AIOPSLAB, the first end-to-end benchmark suite for LLM-based agents on the full cloud incident lifecycle, covering detection, localization, root-cause analysis, and mitigation. AIOPSLAB provides realistic microservice deployments, a fault library, an Agent-Cloud Interface (ACI) that abstracts cloud complexity for agents, and an Orchestrator that turns ad-hoc agent demonstrations into reproducible experiments.

Together, the systems in this dissertation mark a first step toward autonomous troubleshooting for modern cloud systems. We conclude with a discussion and future directions toward fully autonomous clouds.

Open to everything,

attached to nothing,

with love for those around me.

Acknowledgements

Coming from a small town in China, Xinxiang, which we lovingly nickname "Chinese New York" (the character *xin* means "new", and *xiang*, with a wink, "York"), everything that has happened to me up to today still feels like a long dream. These years have brought me into contact with many wonderful people and have left me with countless memories I would not trade for anything. I would like to take a moment to thank every person who has seen me, supported me, and walked alongside me along this Ph.D. journey.

First, I would like to thank my Ph.D. advisor, Professor Tianyin Xu. I am thankful that Tianyin admitted me into the program in 2021. I still remember finding his email buried in my spam folder and rushing to write him a long reply. From that moment on, I have spent the better part of five years being "watched" by my "watchman" Tianyin in the cornfields of Urbana-Champaign. I still remember the entire weekend day he once gave up to sit and watch over me in my first semester. I still remember the conversations we had in person and online, deep into many midnights. I still remember the warmth of the holiday parties where we shared food and drink. More than an advisor, Tianyin has been a mentor and a true friend: someone who has kept me grounded and pointed me back to what I believe in through the harder stretches of this journey.

I am also deeply grateful to the rest of my doctoral committee: Professor Darko Marinov, Professor Lingming Zhang, Doctor Minghua Ma, Professor Indranil Gupta, and Professor Minjia Zhang. I owe a particular debt to Darko, whose guidance during my academic job search went well beyond what I expected. I still clearly remember the day he agreed to be my mentor for the Mavis Future Faculty Fellowship; the day I had Burger King with him and Tianyin; and the day we drank "medicine" together. He has sat through my job talk more times than I can count, helped me revise my application package and slides line by line, asked the hard questions, and pushed me to deliver the best version of every presentation I gave. Lingming has been my basketball companion

for all these years. On the court, I learned from him not only how to improve my shot but a great deal more. Off the court, he has been generous with advice on research, teaching and everything, starting with my very first project, and on the next steps of my career. Minghua was my mentor at MSRA during my gap semester. We worked together again closely during my summer internship at Microsoft Research, which remains my most unforgettable time in the U.S. Minghua has been incredibly generous in sharing his time, resources, and connections with me. Without question, he has been my best collaborator and my greatest source of support outside of UIUC along this journey. I love Indy's distributed systems course and his now-legendary raps. His slides and homeworks are perhaps the best I have ever seen at balancing humor with technical depth, and I hope I can have his teaching style in my own career. I have been both Minjia's student and his TA, and what I learned in his course proved invaluable during my own job search. Minjia is also remarkably calm, which is a quality I truly hope to develop more of myself. Each of them, in their own way, helped me see this world from angles I would not have found on my own.

I have been extraordinarily lucky to share these years with a wonderful group of labmates. My seniors taught me a lot and always lent a hand whenever I needed it: Xudong Sun and Jinghao Jia. Besides Tianyin, Xudong gave me more help in research than anyone else throughout my Ph.D. Also, he is really the one whom I trust. I learned what genuine passion for systems looks like from Jinghao, especially when listening to him talking about the Linux kernel. My close peers became my closest friends: Siyuan Chai, Tyler Gu, and Shuai Wang. We have too many stories to tell and too much fun to recall. And those who joined the group after me: Wentao Zhang, Jiyuan Zhang, Ayush Bansal, Jackson Clark, Yiming Su, Cathy Cai. I deeply appreciate their help and friendship. Every time we get together, we share great ideas, lively discussions, and plenty of laughter.

I also want to thank my collaborators: Saurabh Jha, Adam Bates, Jiaqi Pan, Jackson Clark, Yiming Su, Huaibing Xie, Haoran Yan, Jun Zeng, Anna Mazhar, Xinyu Lian, Manish Shetty, Rohan Arora, Xudong Sun, Ze Yang, Shuai Wang, Hao Kang, Yu Kang, Ming Wen, and many others. In particular, I still cannot quite believe how much Jackson has grown into a good researcher; I clearly remember meeting him for the first time when he was a course assistant in a class I was TA-ing.

I am also grateful to the many friends who helped me along the way: Chenyuan Yang, Ziqi Zhang, Shengnan Wang, Yongzhou Chen, Hao Kang, Yizhu Jiao, Xuefeng Du, Boyan Du, Zhen Han, Zhuoran Han, Shaogeng Zhang, Anna Karanika, Xinying Zheng, Chirag Shetty, Ming Chen, Youning Chen, Mingye Xiong, Jun Yang, Yuxuan Jiang, Kaizhuo Yan, Bohan Cui, Xiaojuan Ma, Kai-Siang Wang, Lilia Tang, Henry Zhu, Sarthak Chakraborty, Ahan Gupta, Hao Lin, Kai-Hsun Chen, and many others. I spent one of my best summers in Seattle with my friends (shoutout to Chenyuan, Hao, Yizhu, Xuefeng, Ruomeng, and Dujian). Playing sports with friends was the most relaxing part of the later years of my Ph.D.; so I spent another one of my best summers with Ziqi in the following year, and I really enjoy playing golf with him. Additionally, I was drawn to tennis, skiing, and poker with the help of Shengnan, Shaogeng, Zhuoran, Ming, Youning, Mingye, and Yongzhou. Without tennis, I would never have met such good friends. Lastly, I still feel bad about how often I kept Chenyuan waiting for me (patiently) at the gym, but I am also grateful for all the activities and events we have shared along the way. I may not get many more such chances, but I truly hope I will. He is the friend I will trust forever.

I want to thank professors who helped me during my job search: Tianyin Xu, Darko Marinov, Zirui (Neil) Zhao, Yiling Lou, Kexin Pei, Ang Chen, Chenfeng Xu, Tong Zhang, Xiaojing Liao, Max Fowler, Fan Lai, Sishuai Gong, Xuanhua Shi, Haibo Chen and Tao Xie. Without their help, I would never have made it through this difficult job-search period.

Special thanks to my roommates Shuai Wang, Xuhao Luo, and Junzhe Kang. I have spent so much of these years with you, and sharing a home with you made me feel safe and at ease. Thank you for always listening to me and cheering me up when I was down. I will never forget the nights we drank together and sang together. Shuai's yellow car was the first transportation I had in Champaign, and he drove me through not only the roads of this city but also some of the darkest days. Xuhao is the best roommate I have ever had, and I would gladly live with him again anytime. When I first arrived at UIUC, I knew nobody in the U.S.; Junzhe was the first friend I made here, and he supported me greatly both before and after his graduation.

I am also grateful to the friends I made in Singapore before I came to the U.S., whose friendship has stayed with me throughout my Ph.D.: Jun Zeng, Zhaoying Li, Chuqi Zhang, and Jiahao Liu. Your friendship has truly stood the test of time, and sharing my good news with you can always satisfy my greatest feeling of accomplishment. I owe a deep thanks to my advisors at NUS, Professor Zhenkai Liang and Professor Jialin Li. Your support has reached me well beyond research.

Special thanks to my friends Xupei Wang and Ning Ding. Without you, I would never have made it this far.

Finally, my deepest gratitude goes to my family. Mom and Dad: your love and steady support have been the foundation of everything I have done. Mom, you have taught me more than any Ph.D. education ever could, and your attitude toward life is what I most hope to inherit. Dad, you have always encouraged me to be independent and to pursue "reliability research" not only in systems, but in life.

To those closest to me, who have stood by me through every challenge—thank you. This dissertation would not have been possible without you.

Contents

Chapter 1 Introduction

The reliability of modern cloud systems is crucial: even a few minutes of outages can lead to significant user impacts and financial losses [1], [2]. Despite extensive engineering and research effort [3], [4], [5], [6], modern cloud systems still struggle to prevent failures before they happen and to detect, localize, and mitigate them quickly once they occur [7].

A central reason is that reliability engineering remains human-centric [8], [9]. A cloud incident typically progresses through a lifecycle of *detection*, *triage*, *diagnosis*, and *mitigation* [10], [11], [12], [13], and at nearly every stage humans are deeply involved. Developers manually anticipate which transient failures to handle or tolerate, but tests rarely cover the full space of transient faults. Once an incident reaches production, on-call engineers (OCEs) must sift through noisy telemetry to decide which alerts to act on, which tools to use, and which mitigation paths to follow. As failures are the norm in cloud systems, this human-in-the-loop practice cannot keep pace with the rate at which faults arrive. This dissertation argues that cloud failure management should become *autonomous* across the full incident lifecycle—from preventing latent bugs before they reach production to safely mitigating live failures [14], [15]—and that this progress must be made measurable. Achieving this goal requires solving four challenges.

The first challenge arises before any incident occurs. Modern cloud applications increasingly depend on cloud services such as AWS S3, Azure Storage, and CosmosDB. While this "cloud-native" practice brings clear benefits, it introduces reliability challenges that go beyond traditional software faults: HTTP requests can fail at any point along the request or response path, transient errors can be retried opaquely by the SDK, and the same logical operation can be executed more than once even when the application sees a single API call. Error-handling paths in cloud-backed applications thus form a large, deeply nested space that is exercised only under faults that are rare in development and testing, so latent bugs slip into production, where transient errors at scale eventually trigger them and cause outages. Fault injection and chaos engineering

are the natural way to expose such bugs before release [16], [17], [18], [19], [20], [21], but existing tools mostly target the cloud *services* rather than the applications that use them, and either inject faults at coarse boundaries (e.g., killing an entire service) or demand substantial fault-injection plumbing in tests. As a result, applications are rarely tested under a realistic transient-fault model. This dissertation therefore begins by systematically surfacing latent error-handling bugs in cloud-backed applications: characterizing the patterns under which such bugs arise and building a push-button reliability testing tool that exposes them automatically.

Even with better testing, real cloud systems are never bug-free, and incidents continue to surface. The first task once a failure occurs is to *diagnose* it, finding the root cause among massive volumes of noisy telemetry: distributed traces across hundreds of microservices, metrics at millisecond granularity, and logs that scale with traffic. Diagnosis today is largely manual and ad-hoc, resting on personal experience, runbooks of varying quality, and tribal knowledge that transfers poorly across services and across generations of engineers. Large language models (LLMs) offer a promising way forward: they can parse high-volume data, discern the relevant signal, and produce succinct explanations, and they can adapt to new and evolving incidents [22], [23]. A growing body of work applies them to incident diagnosis [24], [25], [26], yet these systems typically operate on a single information channel and lack the structure to combine multi-source evidence the way an experienced OCE does; and naïvely feeding all telemetry to an LLM overruns its context budget with a long tail of irrelevant data, while the model still lacks the domain-specific knowledge that cloud incident management demands.

Diagnosis identifies what went wrong, but it does not restore service. Mitigation must then take corrective action, and doing so safely is fundamentally harder than diagnosis, because mitigation changes system state, often irreversibly, and a flawed action can drive an already degraded system into a deeper failure or a complete outage. Recent LLM-based agents have begun to explore autonomous mitigation [27], [28], [29], [30], but

few focus on safety, offering no formal guarantee about what an agent may do to a live system or how the system state evolves under its actions. The third challenge is therefore to extend autonomy from diagnosis to mitigation while guaranteeing that the agent's exploration cannot make things worse.

Finally, the rapid rise of AI agents has outpaced our ability to evaluate them. Existing AIOps benchmarks tend to be either narrow in scope (a single task such as anomaly detection on metrics, over static, pre-collected data) or specific to one application domain, and most lack the live, interactive setting that agents require to plan and act over multiple steps. LLM-based agents, which reason and act dynamically, have outgrown the harnesses available to test them. The fourth challenge is to enable rigorous, reproducible evaluation of autonomous SRE agents across the full incident lifecycle.

These four challenges define the arc of this dissertation. By addressing them, this work takes a step toward autonomous, measurable, and safe failure management of modern cloud systems.

1.1 Thesis Statement

This dissertation argues that autonomous troubleshooting of modern cloud systems can be made practical through systematic improvements at both ends of the incident management lifecycle: 1) before production, systematic testing can reduce the failure surface by detecting latent bugs; 2) in production, autonomous failure management can be enabled by AI with the safety guardrails.

1.2 Contributions

This dissertation makes four contributions, organized around the cloud incident lifecycle.

1. Rainmaker: surfacing latent bugs before they reach production. We begin in the pre-incident stage, where the goal is to find and fix error-handling bugs in cloud-backed applications before they reach customers. We first conduct a study of how

error handling goes wrong under the cloud-based fault model and distill the findings into a four-pattern bug taxonomy: *no error handling*, *throwing unrelated exceptions*, *silent semantic violations*, and *state divergence*. Building on this taxonomy, we present Rainmaker, a push-button reliability testing tool that exercises an application's error-handling paths by injecting realistic transient cloud faults at the HTTP layer. Rainmaker plugs into existing test suites without any code changes, uses four fault-injection policies that together cover the taxonomized bug patterns, and selectively targets unique call sites of cloud-service APIs to keep testing tractable. Across 11 popular .NET applications, Rainmaker found 73 new error-handling bugs, of which 55 have been confirmed and 51 fixed by upstream developers, with a 1.96% false-positive rate.

2. RCACopilot: LLM-augmented root-cause analysis. While Rainmaker shrinks the bug surface before deployment, real-world cloud systems are never bug-free, and production incidents continue to occur. We turn next to the post-incident stage, where the first task is to figure out what is going wrong. We present RCACopilot, an AI-augmented root-cause analysis system that adapts LLM reasoning to cloud incident management. RCACopilot composes per-alert-type *incident handlers* from three reusable actions (*scope switching*, *query*, and *mitigation*) to collect multi-source diagnostic data, summarizes the data with an LLM to fit within prompt budgets, and then performs nearest-neighbor retrieval over historical incidents to drive automatic chain-of-thought prompting to predict the root-cause category and generate its explanation. On a real production incident dataset, RCACopilot achieves a micro-F1 score of 0.766, substantially outperforming classical machine-learning baselines and naïve LLM approaches, while remaining fast enough for use under on-call time pressure.

3. Stratus: autonomous mitigation with safety guarantees. RCACopilot accelerates diagnosis, but diagnosis alone does not restore service. To close the loop on autonomous troubleshooting, we must extend autonomy to *mitigation*, which actually changes the system state and is fundamentally harder to do safely. We present Stratus,

an LLM-based multi-agent system for autonomous Site Reliability Engineering. STRATUS orchestrates four specialized agents (detection, diagnosis, mitigation, and undo) through a deterministic state machine, and exposes the cloud to them through Agent-Cloud Interfaces. At its core is *Transactional No-Regression* (TNR), a formal safety specification built on classical transaction semantics. TNR guarantees that any unsuccessful mitigation can be *undone*, and that the externally visible system severity never grows beyond the baseline at the time the failure was detected. We implement TNR with writer-exclusivity scheduling, a stack-based undo mechanism atop Kubernetes' state-reconciliation principle, and a bounded risk window per transaction. TNR turns agentic SRE from a hope-it-works exercise into a safe-exploration loop. On mitigation problems from two benchmarks, STRATUS solves 69.2% (9/13) and 50.0% (9/18) of the problems respectively, outperforming state-of-the-art agentic baselines by at least 1.5× across multiple LLM backends.

4. AIOPSLAB: a holistic benchmark for agentic SRE. LLM-based SRE agents need a common, rigorous benchmark to be evaluated on. As our final contribution, we present AIOPSLAB, the first end-to-end benchmark suite for evaluating LLM-based agents across the full cloud incident lifecycle: detection, localization, root-cause analysis, and mitigation. AIOPSLAB formalizes each evaluation scenario and provides realistic microservice deployments, a fault library that goes beyond surface-level symptoms, an Agent-Cloud Interface (ACI) that abstracts cloud complexity for agents, and an Orchestrator that standardizes how problems are posed, executed, and evaluated. AIOPSLAB turns ad-hoc SRE agent evaluations into reproducible, comparable experiments, and is the harness on which we evaluate STRATUS and the baselines above.

The rest of this dissertation is organized as follows. Chapter 2 presents Rainmaker. Chapter 3 presents RCACOPILOT. Chapter 4 presents STRATUS. Chapter 5 presents AIOPSLAB, the benchmark used to evaluate STRATUS and the baseline agents. Chapter 6 concludes and discusses future work.

Chapter 2 Reliability Testing for Cloud-Backed Applications

Proactive defect detection is the first guardrail before failures surface. Modern applications increasingly depend on cloud services (e.g., AWS S3, Azure Storage) for various functionalities. Despite the benefits, such "cloud native" practice imposes emerging reliability challenges introduced by the fault models of opaque cloud backends and less predictable connections between the application and cloud services.

Rainmaker is a "push-button" reliability testing tool for applications that use RESTful cloud services, such as Azure Storage, Azure CosmosDB, and AWS S3. It checks whether the application under test can correctly tolerate or handle common transient errors under the cloud-based fault model (e.g., temporary service unavailability and request timeouts), and detects bugs.

A developer can directly apply Rainmaker as a "plug-and-play" tool to their existing test suites in their existing testing environments, without writing additional code or configurations. The plug-and-play nature is achieved by its fault injection mechanism—Rainmaker injects errors by intercepting outbound REST API calls made by the application to the cloud service at the HTTP layer. It includes a standalone HTTP proxy component to do the interception. For example, to inject a 5XX error on the request path of a REST call, the proxy blocks the request from reaching the service and responds with an HTTP response containing the 5XX error code. To inject a timeout on the response path, the proxy lets the request go to the service; however, on receiving the response, it introduces a delay to force a timeout at the application.

Compared with injecting faults directly into application code (e.g., in the form of exceptions), intercepting at the HTTP layer brings a number of technical benefits: (1) it allows intercepting *fine-grained* REST calls made by the application, including those triggered by SDK retries; (2) error handling in cloud-backed applications depends not only on exception types but also HTTP status codes; (3) it makes the fault injection mechanism transparent to the application under test and works irrespective of the

application language and architecture; (4) HTTP requests/responses are highly interpretable, and errors can be uniformly injected by manipulating HTTP responses, with no need to understand external dependencies (e.g., complex exception objects with multiple fields).

Usage. Rainmaker takes as input an existing testing suite that uses RESTful cloud services such as Azure Storage services or their emulators [31], and a desired coverage metric (Section 2.5). Rainmaker first installs a local HTTP proxy that can intercept and manipulate HTTP traffic to and from cloud services (or their emulators). It then selects and executes a minimal set of tests required to achieve the target coverage. As tests are executed, Rainmaker injects faults into their REST API calls according to automatic fault-injection policies. After each test is executed, Rainmaker's oracles analyze the test outcomes and raise alerts as potential bugs are detected.

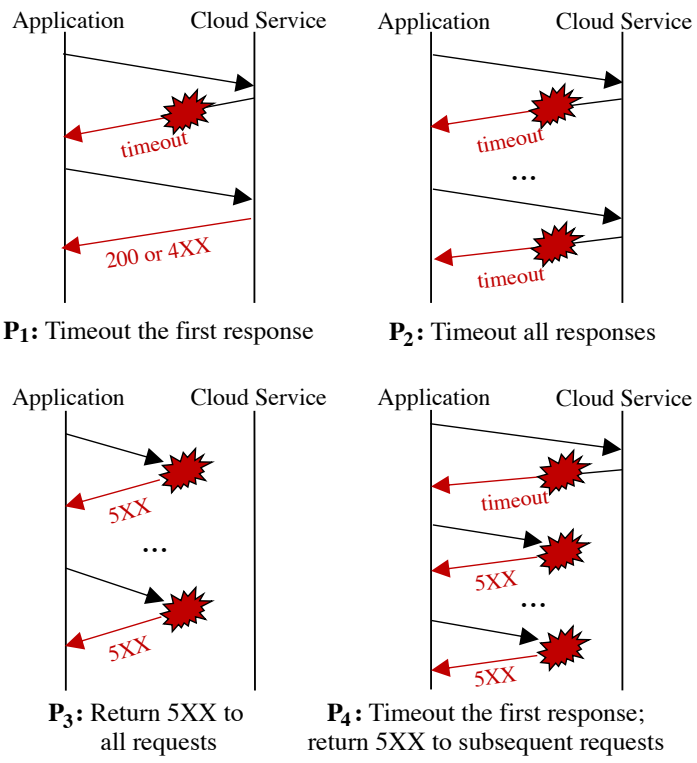


Figure 1: Fault injection policies of Rainmaker.

2.1 Background

Ideally, cloud-based programming should not be different from traditional application programming using native libraries. Unfortunately, this is rarely the case in practice: handling errors under the cloud-based fault model is challenging and error-prone. We discuss the emerging reliability challenges faced by cloud-backed applications as the background and motivation of Rainmaker.

2.1.1 Errors in Cloud-backed Applications

There are three key components related to how cloud-service-related errors are exposed to applications.

Error responses from cloud services. A request from the application can fail due to a client-side error (e.g., local network timeout) or a service-side error (e.g., temporary service unavailability). In the latter case, the service returns an error response indicating the error type. Most cloud services are RESTful, and their APIs reuse HTTP response status codes defined by the HTTP/1.1 standard. HTTP status codes 4XX and 5XX indicate "client errors" and "server errors" respectively. In addition, a cloud service can include service-specific error codes in the response payload to indicate fine-grained error types. For example, Azure Blob Storage can return 44 different error codes (e.g., `BlobImmutableDueToPolicy` and `BlobAlreadyExists`) with the same HTTP status code 409 (`Conflict`) [32]. The same practice is used by other cloud services such as CosmosDB [33], S3 [34], and SQS [35]. Applications use these codes to understand the nature of the errors and take error-handling actions accordingly.

Retry on transient errors. When a request fails due to a *transient* client- or service-side error (e.g., network timeout or server overload), an application may retry the request, hoping that it would eventually succeed. Cloud-backed applications mostly use the SDKs provided by cloud service providers to interact with the cloud services. Besides offering easy-to-use APIs, SDKs also include error-handling logic that aims to provide a

native programming experience: when a REST API call to a cloud service fails due to a transient error, the SDK tries to mask the error from the application by retrying the request [36]. Table 1 summarizes the retry policies of several representative cloud SDKs. The behavior varies widely across services, across versions of the same service, and even across the language SDKs of the same version. Besides HTTP status codes, SDKs may also retry on specific error messages (e.g., the Azure Storage SDK additionally retries on `InternalError`, `OperationTimedOut`, and `ServerBusy`).

Propagating errors up to the application. If the retry efforts fail, or if the error is of a *permanent* type, the SDK propagates the error up to the application in a way that is consistent with a native programming experience. For example, .NET and Java SDKs propagate errors to applications as exceptions.

2.1.2 Emerging Reliability Challenges

Unanticipated errors and inconsistencies. Our analysis of multiple cloud service APIs and SDKs from Azure and AWS reveals a lack of standards and behavioral consistency in all three components above, across cloud services and across SDKs from the same and different cloud providers. We observe many undocumented and inconsistent error codes returned by cloud services. The same logical error can be represented by different error codes across services: Azure Queue Storage represents the error `QueueNotFound` with code 404, while AWS SQS uses code 400 for the same error. Whether an error is masked by the SDK (via retry) and whether an error is propagated

Table 1: Retry policies of different cloud services.

SDK	Retry (HTTP codes)	API	Notes
Azure Storage (Blob/Queue/Table)	408, 429, 500, 502, 503, 504	Any	Only 429 and 503 are retried before v12.3.0.
Azure CosmosDB (HTTP mode)	403, 404, 408, 503	Read	Only enabled for multi-region (no retry for single-region).
AWS S3 / AWS SQS	500, 502, 503, 504	Any	Inconsistencies across language SDKs (e.g., Java vs. .NET).

to the application also vary widely across services, across versions of the same service, and even across language SDKs of the same version. The Azure Storage .NET SDKs before v12.3.0 only retry on error codes 429 and 503; later versions add retry for 408, 500, 502, and 504 [37]. The `DeleteMessage` API of Azure Queue propagates an exception to the application when a 404 is returned, while the `DeleteEntityAsync` API of Azure Table silently ignores 404 errors and returns success. These inconsistencies make it hard for developers to anticipate which errors might surface in their code, and lead directly to the bug patterns catalogued in the next subsection.

Retry and non-idempotent APIs. While retry is a common practice to mask transient errors, retry can introduce subtle errors of its own. A retry on a non-idempotent cloud service API can cause elusive effects such as remote data corruption, which can remain latent or lead to additional errors. Each service implements different retry logic, with no standard practice or discipline, so the semantics of SDK APIs under error conditions is often opaque and inconsistent.

Rarity and large fault space. Developers often miss error-handling bugs with small-scale, short-duration functional tests because cloud-based faults are rare. Fault-injection tools can simulate error scenarios during testing, but the key challenges lie in specifying *policies* (what faults to inject, where, and when) and *oracles* (what post-fault conditions indicate a likely bug). The space of possible policies and oracles is large, if not infinite, which motivates Rainmaker's principled, push-button approach to policy and oracle design.

2.1.3 Our Goal

Our goal is to address the emerging reliability challenges faced by cloud-backed applications by (1) systematically understanding the patterns under which error-handling bugs arise, and (2) building practical tooling that systematically tests whether a cloud-backed application can correctly handle the myriad errors that may occur during

its interactions with the cloud services it depends on. We emphasize a "push-button" technique that can be directly used by application developers in an existing testing environment, without the need to write additional code or configurations. Rainmaker is the realization of these goals, organized around three principles: testing should be *effective* at uncovering error-handling bugs, only address *valid* faults that the cloud service is documented to produce, and remain *efficient* so that high coverage can be achieved with a small number of test runs.

2.2 Bug Taxonomy

To systematically understand the patterns of error-handling bugs in cloud-backed applications and to guide the design of Rainmaker, we develop a four-class bug taxonomy that describes how error handling under the cloud-based fault model can go wrong. The taxonomy considers the transient error(s) that occur during *one* REST API call interaction initiated by the application; it does not reason about multiple independent REST API calls.

2.2.1 No Error Handling

In this pattern, the application simply does not handle certain transient errors. This can happen due to the inconsistencies discussed in Section 2.1.2: an application may not anticipate a cloud service to return a specific error, or may mistakenly expect the SDK to mask a known transient error. For example, an application using Azure Storage's .NET SDK before v12.3.0, which does not retry on certain transient error codes, may mistakenly assume that *all* transient errors are masked by the SDK and hence not handle them. As a result, some transient errors from the cloud service can lead to application crashes or other undesirable behavior.

2.2.2 Throwing Unrelated Exceptions

In this pattern, the (mis)handling of a transient error results in a new unhandled error that is *unrelated* to the root-cause fault. A common example involves request retries: when a request to a cloud service fails with a timeout, neither the SDK nor the application can determine whether the timeout occurred on the request path or on the response path. SDKs commonly treat timeouts as transient failures and retry. However, if the timeout occurred on the response path (the original request was successfully executed by the service), the retry can fail with a new error different from the original error, because the original request has invalidated the precondition of the retry. This new error, if not handled properly, can lead to undesirable effects.

Figure 2 shows an example of such a bug [38] that Rainmaker detected in Microsoft BotBuilder [39]. BotBuilder stores logs in the Azure Blob Storage service. Each log operation calls an SDK API to create a new blob. If the API call successfully creates the blob but the response times out, the Azure Storage SDK automatically retries the request. However, since the blob has already been created by the first request, the retry operation fails with a 409 (`BlobAlreadyExists`) error. The SDK propagates this *permanent* error to the application. BotBuilder does not anticipate or handle the error, which breaks the loop that uploads logs to Blob Storage and results in a loss of subsequent log data. The exception seen by the application is unrelated to the root cause (a transient timeout).

2.2.3 Silent Semantic Violations

In this pattern, mishandling of a transient error causes a semantic violation of the REST API specification, without observable symptoms. The REST call returns successfully, and so the application executes on its happy path. However, the silent semantic violation may eventually result in data loss, data corruption, or other incorrect application behavior. A common manifestation involves response timeouts: the retry operation

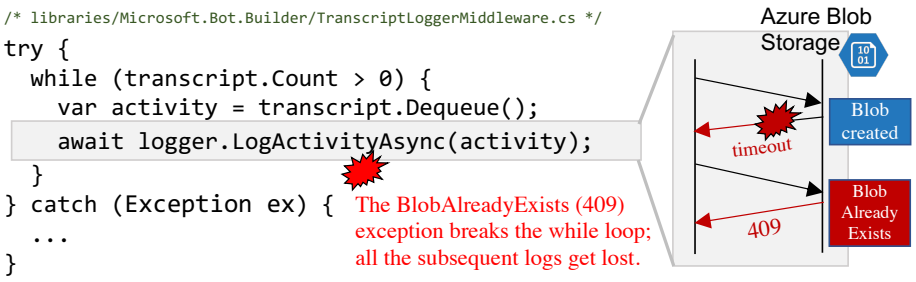


Figure 2: A bug of throwing unrelated exceptions in Microsoft BotBuilder detected by Rainmaker (confirmed and fixed). The Azure Storage SDK automatically retries on timeouts, which returns a 409 error because its precondition has been invalidated by the first request.

succeeds, the SDK successfully hides the transient error from the application, but the underlying REST API was actually executed more than once.

Figure 3 shows a silent semantic violation [40] that Rainmaker detected in Microsoft Orleans [41]. Orleans uses Azure Queue Storage to manage messages and implements `GetQueueMessage()` to dequeue *one* message from the queue, which calls the SDK API `ReceiveMessagesAsync`. The corresponding HTTP request is non-idempotent and should not be naïvely retried, because each retry changes the contents of the queue. However, if a request successfully dequeues a message but its response times out, the SDK silently retries the request multiple times, each successful retry dequeuing another message. If the last retry succeeds, the API returns the last message. The application is

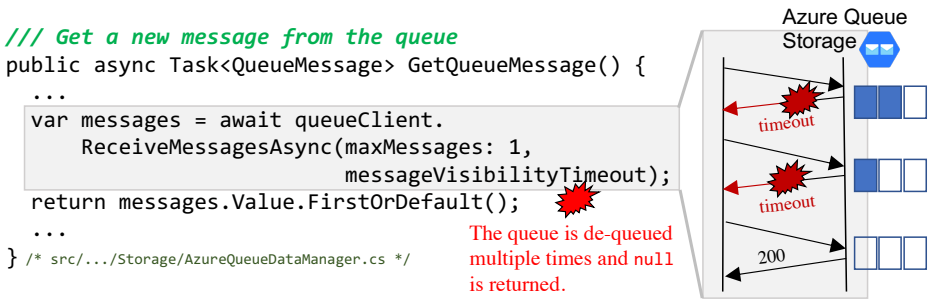


Figure 3: A silent semantic violation bug in Microsoft Orleans detected by Rainmaker. Azure Storage SDK automatically retries multiple times on timeouts, and mistakenly empties the queue.

unaware that multiple messages have been dequeued, even though the API documentation mentions this corner case. This violates the semantics of `GetQueueMessage()`, which is documented to "*get a message from the queue*". In the worst case, repeated retries can dequeue all messages from the queue, in which case the SDK API returns `null` and Orleans dereferences the `null` pointer and crashes.

2.2.4 State Divergence

In this pattern, a mishandled transient error leads to divergence between local state (in the application) and remote state (in the cloud service). There is no API semantic violation as in Section 2.2.3, but the state divergence can lead to undesired application behavior such as exceptions and resource leaks. State divergence often happens when an application *optimistically* updates a local state that is correlated with the success of a cloud API call made after the update. The bug manifests if a transient fault fails the request (so no change occurs on the cloud side) but the application does not roll back the optimistic update. The local state then makes the application behave as if the REST API call had succeeded, while it actually failed.

Figure 4 shows a state-divergence bug [42] that Rainmaker detected in Microsoft BotBuilder. BotBuilder uses Azure Blob Storage to store blob data organized into

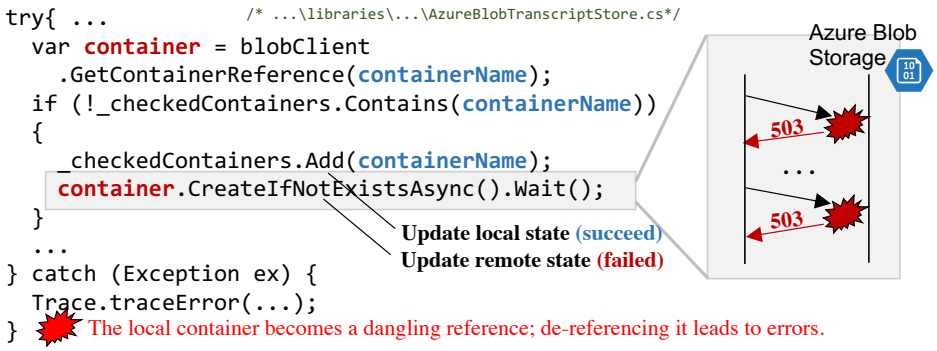


Figure 4: A local-state divergence bug in Microsoft BotBuilder detected by Rainmaker (confirmed and fixed). The Azure Storage SDK automatically retries multiple times for 503 errors, but the local state is updated before the call.

containers. To create a container, BotBuilder calls the REST API `CreateBlobContainer`. When transient errors (e.g., 503 `ServerBusy`) occur on the request path, BotBuilder swallows the exception in its `catch` block. However, BotBuilder adds the container to its local set of created containers *before* calling `CreateBlobContainer`. As a result, the local state is corrupted with a dangling container pointer, which leads to crashes when BotBuilder later dereferences the pointer (e.g., during a list operation). The bug has the same essence as file system bugs that violate update dependencies [43], but transient errors are likely more frequent than file system crashes.

State divergence can also happen when a request changes the remote state, but the application is unaware of the change. Such bugs can manifest when a transient fault breaks the *return path* of a REST call that has changed the remote state. If not handled correctly, the application would assume the call never succeeded, leading to inconsistencies of states.

2.3 Fault Injection Policy

A fault injection policy specifies *what* faults to inject and *where* (at which REST API calls) to inject them. Rainmaker's fault injection policies are designed with two main objectives.

First, the policies should be *effective*. This requires them to (1) inject only *valid* faults and (2) cover the myriad bug patterns of the taxonomy (Section 2.2). One common policy is *randomized* fault injection (e.g., selecting a random fault at a random REST call). However, randomized injection can hardly be effective. As shown in Section 2.2, to expose certain bug patterns needs multiple specific faults injected along the interaction of a REST API call—randomized injection is unlikely to hit the specifics. In fact, randomized injection cannot even guarantee valid faults. For example, returning

a 503 (`ServiceUnavailable`) error after a write request is successfully executed is invalid, because this is inconsistent with the cloud service contract.[1]

Second, the policies should enable *efficient* testing. Exhaustively injecting all possible faults at every REST call could be prohibitively expensive, because one test could issue thousands of REST API calls (see Section 2.7), and many different faults are possible for each call. This is further aggravated by the fact that each fault injection may require a separate test run because injecting the first fault might disrupt a test's subsequent execution.

We next discuss how Rainmaker achieves the two objectives in Section 2.4 and Section 2.5, respectively. Note that Rainmaker can be easily extended to support new policies.

2.4 What Faults to Inject?

Rainmaker injects transient faults that occur during the interaction of one REST API call, following the bug taxonomy (Section 2.2). However, the fault space is large even for a single REST call. This is because each of the large number of possible faults may occur on the request or the response path of the original request or subsequent retries issued by the SDK. Interestingly, we find that a small set of four policies (Figure 1) is sufficient to cover the taxonomized bug patterns, as shown in Table 2. For REST calls that do not retry, the four policies are reduced to two: (1) return a transient error code to the request, and (2) timeout the response. The four policies are:

- $\mathbf{P}_1$ (Timeout the first response). This policy forces a retry that can expose bugs related to invalidated preconditions. Since the timeout is at the response path after the request takes effect at the cloud service, the retry could trigger bugs of throwing unrelated exceptions, silent semantic violation, and/or state divergence, depending on the REST API semantics and the handling logic.

[1] In practice, a cloud service backend can have bugs to return such an inconsistent response [44]. However, we do not consider buggy cloud services.

- $\mathbf{P}_2$ (Timeout all responses). This policy presents an entirely timed-out REST call to the application, while the REST call has taken effects (potentially multiple times) at the cloud service. It could trigger bugs of silent semantic violation and state divergence.

- $\mathbf{P}_3$ (Return transient error codes to all requests). Under this policy, the REST call does not reach the cloud service as all requests are returned with transient service-side errors. The policy can potentially expose bugs of state divergence, if the local state is optimistically changed before the REST call, but not restored after the call fails.

- $\mathbf{P}_4$ (Timeout the first response and return transient error codes to all subsequent requests). This policy presents a failed REST call (with a transient error code in response) to the application, which on the contrary has been successfully executed exactly once at the cloud service. It could trigger bugs of state divergence.

Rainmaker by default injects 503 (`ServiceUnavailable`) as the transient fault(s) for request failures. For the responses, Rainmaker injects a timeout fault. These two types of transient faults are common and safe to inject, and thus are valid faults on the request and response path, respectively. In comparison, error codes like 500 (`InternalServerError`) have undefined semantics and service-side behavior. The error code can be further customized for specific cloud services and their SDKs based on their definitions of transient faults and retry policies.

Table 2: The mapping from the bug patterns to the error injection policies that can potentially expose each bug pattern.

Bug Pattern	**Fault Injection Policies**
No error handling	P_1, P_2, P_3, P_4
Throwing unrelated exception	P_1
Silent semantic violation	P_1, P_2
State divergence	P_1, P_2, P_3, P_4

Rainmaker identifies all the retried requests and their responses of each REST API call by checking the unique request ID in the HTTP header (e.g., `x-ms-client-request-id` for Azure Storage services). The request ID is provided by the SDK to identify the specific REST call request and is shared by all the subsequent retried requests and responses.

One design choice we make is to avoid encoding specifications of REST/SDK APIs in policies (e.g., idempotency of a REST API and retry behavior of an SDK API). Leveraging API information can help optimize test efficiency. However, it is known that specifications are expensive to maintain and are often incomplete and outdated in practice. Rainmaker minimizes its assumptions on the REST/SDK APIs.

2.5 Which REST API Calls to Inject Faults?

A test suite may generate an excessive number of REST calls; injecting faults into all of them can be prohibitively expensive, even with the above optimized policies (it could take several machine-months for one application). In fact, many REST calls can be redundant (e.g., invoked by the same application code location) for the purpose of covering new error-handling code. Rainmaker therefore selectively injects faults into a small number of REST calls, which achieves certain coverage metrics and optimizes testing resources.

Coverage metrics. Rainmaker supports four different coverage metrics (Table 3). While C_1 measures coverage in terms of the REST calls, C_2–C_4 involve call sites of cloud service APIs. We use the term *call site* to denote a location in the application code that invokes an SDK API that eventually makes one or more REST calls (typically, one SDK API invokes one REST API [45], [46]). Call sites reside in the application code, while REST calls are constructed by the SDK. Hence, C_2–C_4 are more intuitive to developers than C_1.

However, computing C_2–C_4 is challenging for Rainmaker and for any other tools that inject faults via a separate HTTP proxy [47], [48]. This is because the proxy process does not have visibility of call sites within the test/application process.

Making call sites available to the HTTP proxy. Rainmaker addresses this challenge with two techniques using automated instrumentation. First, Rainmaker enables a test to communicate with the HTTP proxy through headers of outgoing HTTP messages. Given test binaries, Rainmaker automatically instruments their outbound HTTP calls. An instrumented call can put its call site information in the header of an outbound HTTP request, so the HTTP proxy can retrieve the information. This can be done automatically since outbound HTTP calls are usually made with a small number of standard core APIs. For example, one needs to instrument only four HTTP client API families (e.g., `HttpClient.SendAsync`) provided by .NET core libraries to intercept outgoing HTTP calls from *all* Azure SDKs.

Second, an outbound HTTP call needs to automatically find its call site to put in the HTTP header. If the application were single-threaded, the instrumented HTTP call could identify the call site by taking the caller of the bottom-most SDK function in the call stack. But modern applications are multi-threaded, and an HTTP call usually happens asynchronously in a child thread created as a result of executing the call site. In this case, a call stack taken at an outbound HTTP call does not capture the call site that resides in a different thread.

Table 3: The coverage metrics supported by Rainmaker. We use C_4 as the default metric. Note that injecting faults in every REST call in every test is prohibitive.

	Coverage Metric
C_1	Cover all tests; for each test, select the first REST call.
C_2	Cover all unique call sites of the application code; if a call site is exercised by multiple tests, select the cheapest test.
C_3	Cover all tests and all unique call sites in a pairwise manner.
C_4	Cover all unique call sites of every test; if multiple REST calls exercise the same call site, select the first call to inject.

To solve this problem, Rainmaker utilizes inheritable thread-local storage (ITLS) supported by modern languages such as .NET [49] and Java [50]. Any data stored in the current thread's ITLS automatically propagates to all its child threads. Rainmaker automatically instruments all call sites (identified by SDK namespace) so that at runtime, they store their location information in the current thread's ITLS. When a call site eventually invokes an instrumented, outgoing HTTP call, in the same thread or in a child thread, the call retrieves the call site information from its ITLS and puts it in the HTTP header for the proxy process to examine.

Test planning. With the call site information available at the HTTP proxy, Rainmaker can inject faults according to the specified coverage metric in Table 3. Each coverage metric is a tradeoff between completeness and cost.

To plan the fault injection runs, Rainmaker first performs a *reference run* of the test suite with no fault injection. During the reference run, Rainmaker measures the time taken by each test and observes, by using its HTTP proxy, the REST calls made by different tests. It then selects the tests that issue REST calls as candidate tests for fault injection. Rainmaker then performs an offline analysis to generate test plans containing the minimum number of fault-injection target REST calls (and their tests) in order to achieve target coverages. It also outputs an approximate running time of each plan using the time of the tests in the reference run and, if any, the delays to be injected to create timeout errors. The time helps a developer choose the right coverage metric by understanding the tradeoff between completeness and cost.

Given the time taken by each test and the set of REST calls each test makes in the reference run, it is straightforward to implement the policies C_1, C_2, and C_4. For C_3, Rainmaker models it as a linear programming (LP) problem of generating a set of pairs that cover all N tests and M unique call sites (with each test covering a subset of call sites), while minimizing the total test running time (each test has different running time). It then uses an LP solver to generate a plan.

Note that test planning, including the reference run and LP solving, is done offline as a one-time effort. The results can potentially be reused across test runs in CI/CD environments.

2.6 Test Oracles

A test oracle checks whether the outcome of a fault injection test run indicates a bug. A trustworthy oracle catches different types of bugs with no false alarms so that developers can focus their investigation only on true bugs.

With the goal of being generic and widely applicable to any cloud-backed application, Rainmaker does not use any application-specific oracle. Instead, it devises a set of application-agnostic oracles on top of the existing test oracles encoded in developers' test code. These oracles are effective in identifying various types of bugs, with low false positives.

2.6.1 Exception Oracle

This oracle flags a fault-injection test outcome as a potential bug if (1) the test fails with an exception, (2) the exception is created in application code rather than in test code, and (3) the exception is inconsistent with the injected fault. We now explain the rationale behind the three conditions.

When a test fails with an exception as a result of an injected fault, it may not always mean a bug. For example, in the applications we use for our evaluation, many tests directly interact with a cloud service (e.g., to set up the test environment) but without proper error handling. Figure 5 shows an example where the same cloud API call is wrapped within a try-catch block in the application code, but not in test code. If Rainmaker injects faults into such REST calls, the test will fail with an exception. However, the failure does not indicate application bugs. Rainmaker applies the second condition to filter out test failures due to exceptions created in test code, based on the

exception call stack. Note that Rainmaker avoids injecting faults into REST calls with call sites from the test code.

However, not all test failures due to exceptions created in application code are bugs. For example, a utility method of an application may intentionally propagate an exception to the upper layer and expect it to be handled there. When a test for this utility fails due to not handling the exception, the test failure is expected, and it does not indicate a bug.

Rainmaker applies the third condition to only report bugs when the final exception that causes the test failure is inconsistent with the injected fault (by searching the injected HTTP status code or error code in the exception stack). The intuition is that, if the test fails due to an error different from the injected fault, it indicates that the fault was once handled (it shows the developer's intention to handle it), but the handling is inappropriate (at least insufficient) and causes a different error that fails the test. This oracle can capture bugs of throwing unrelated exceptions as well as silent bugs of

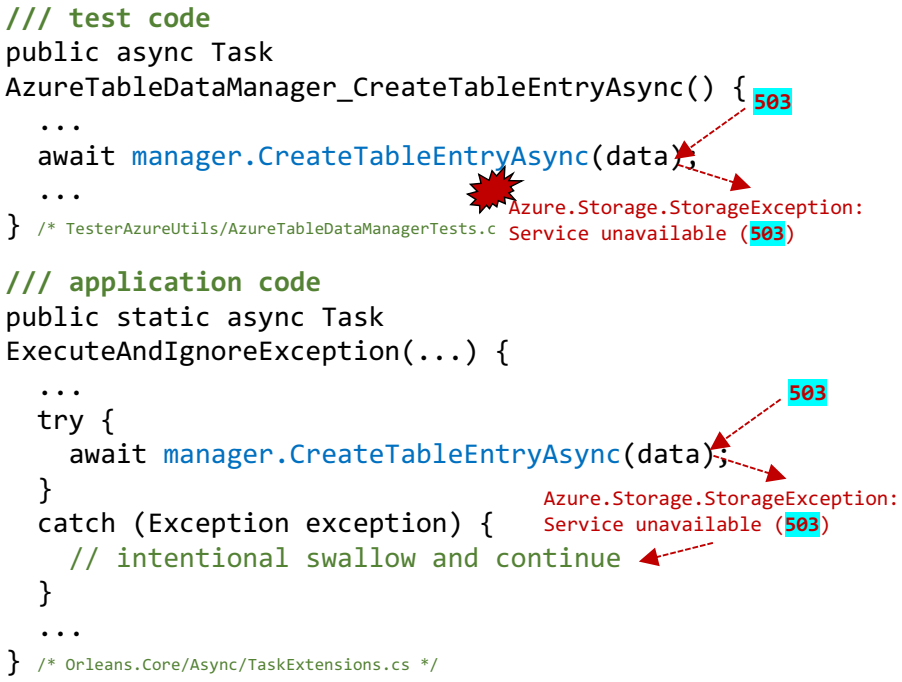


Figure 5: An example that explains why Rainmaker cannot treat all exceptions thrown by a test as a potential bug. In this example, the only usage of `CreateTableEntryAsync` is handled by the application code. The unit test does not include the error handling logic.

semantic violations and state divergence which do not cause immediate exceptions but result in exceptions eventually. Figure 6 shows such an example, where a 503 fault injected to `CreateIfNotExistsAsync` leads to a 404 exception from `UploadFromByteArrayAsync` and fails the test.

Note that the oracle is incomplete. If an application misses error handling (Section 2.2.1), exceptions exposed by the test could be a bug. However, at the unit test level, it is indeterminate.

Relaxing the oracle. With all three conditions, the oracle above is conservative. One can relax it to identify other types of likely bugs. If the developer is testing an application or running a system test (instead of a unit test), she can disable the third condition so that Rainmaker flags any failure (e.g., crash) due to exceptions from application code as a bug. This is because Rainmaker only injects transient faults that are expected to be handled gracefully.

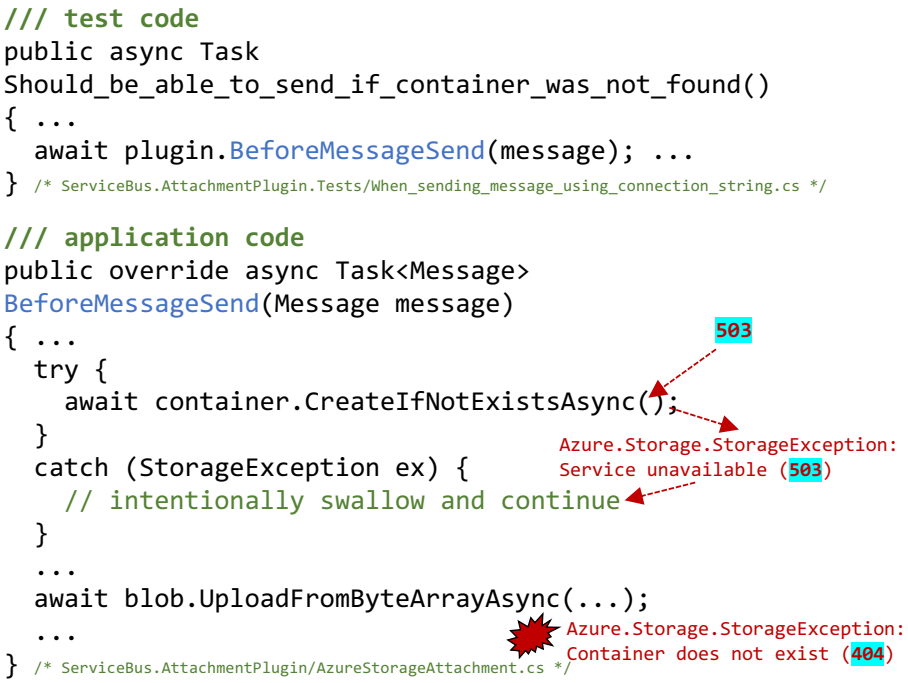


Figure 6: An exception captured by Rainmaker to detect a state-divergence bug in ServiceBus AttachmentPlugin. The exception that fails the test (404) is inconsistent with the injected fault (503).

2.6.2 Assertion Violation Oracle

This oracle flags a fault-injection test outcome as a potential bug if the test fails due to an assertion violation (*and* not an unhandled exception). Intuitively, transient faults injected by Rainmaker should not impair semantic correctness of application code; hence existing assertions should not be violated if the faults are properly handled. The assertions in test code could be brittle to fault injection [51], i.e., an assertion violation is not a bug, but caused by the fault injection changing application runtime behavior. In practice, we find such brittle assertions are small in numbers, as discussed in Section 2.7. The assertion oracle can capture bugs of silent semantic violations and state divergence. Figure 7 shows how Rainmaker leverages the existing assertion to capture a silent semantic violation in Storage.NET [52].

2.7 Evaluation

We evaluate Rainmaker on 11 popular .NET applications (Table 4) that use six different cloud services from two cloud providers, Azure and AWS. The applications include Microsoft Orleans, Microsoft BotBuilder, Microsoft Entity Framework (EF) Core, Microsoft FHIR Server, NuGet Insights, and Petabridge Alpakka, among others. These applications are mature and widely used, and many are maintained by software companies. They use Azure Blob Storage [53], Azure Queue Storage [54], Azure Table Storage [55], and Azure CosmosDB [56] from Azure, and AWS Simple Storage Service

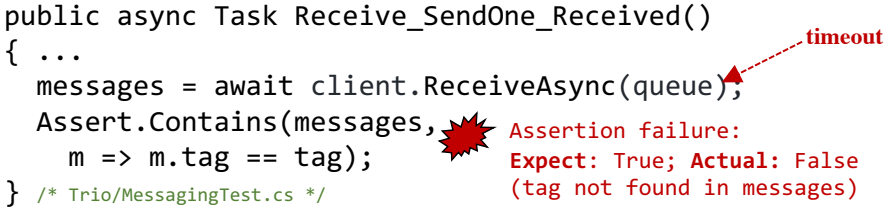


Figure 7: An assertion utilized by Rainmaker to detect a bug of silent semantic violation in Storage.NET backed by AWS SQS. When timeouts are injected, `ReceiveAsync` dequeues multiple times, causing an empty message list returned, which fails the assertion.

(S3) [57] and Simple Queue Service (SQS) [58] from AWS. We apply Rainmaker to each application's existing test suite, which contains both unit and system tests; the number of tests selected by Rainmaker for fault injection ranges from 2 to 420 across the applications. Tests against Azure cloud services run with the Azurite [31] and CosmosDB emulators [59]; AWS tests run against the real S3 and SQS services because no official emulator is available.

Bug detection. Rainmaker finds 73 new bugs across all 11 applications, spanning all four bug patterns in the taxonomy: 29 bugs of *no error handling*, 23 bugs of *throwing unrelated exceptions*, 4 bugs of *silent semantic violations*, and 17 bugs of *state divergence*. Rainmaker finds bugs in *every* evaluated application, demonstrating how error-prone it is to handle transient faults under the cloud-based programming model. We have reported 66 of these bugs upstream; as of writing, 55 have been confirmed by the developers, and 51 have been fixed. Table 5 gives the per-application breakdown.

The 73 bugs cause 2,654 test failures in total. To inspect them, we cluster test failures based on (a) the application call site that invokes the REST API where fault(s) are injected and (b) the exception stack trace in the application namespace, or assertion. For two test failures with both (a) and (b) being the same, they are considered to have the

Table 4: The cloud-backed applications used in the evaluation.

Application	**Cloud Service**	**# Stars**	**# LOC**	**# Selected Tests**
Alpakka	Queue	106	14K	9
AttachmentPlugin	Blob	67	1.3K	32
BotBuilder	Blob, Queue	758	18K	67
DistributedLock	Blob	838	17.1K	30
EF Core	CosmosDB	11.7K	842.4K	420
FHIR Server	CosmosDB	897	112.8K	202
Insights	Blob, Queue, Table	20	51.7K	147
IronPigeon	Blob	255	5.4K	7
Orleans	Blob, Queue, Table	8.8K	187K	155
Sleet	S3	276	18.7K	2
Storage.NET	Blob, Queue, S3, SQS	567	12.4K	36

same root cause, i.e., injecting to the same API call site causes the same exception stack trace in application namespace or assertion violation.

Many of the bugs Rainmaker discovered have severe real-world consequences (Table 7): externalizing unrelated exceptions to clients, operation failures that prevent resources from being created, incorrect results or states, application crashes, data loss or inaccessibility, and resource leaks. All of these consequences are triggered by transient faults against *a single* REST API call.

The four fault-injection policies (Figure 1) are complementary: no single policy uncovers all the bugs (Table 6). For example, throwing-unrelated-exception bugs such as the BotBuilder bug in Figure 2 are exposed by P_1 (timeout the first response), which lets the original request take effect at the cloud service so that the SDK's retry collides with the invalidated precondition. The test oracles are similarly complementary: silent-semantic-violation bugs emit no exception and are only caught by post-condition assertions. Rainmaker also surfaces bugs that randomized fault injection would almost certainly miss. For instance, the DistributedLock bug DistributedLock-132 [60] is only triggered when a timeout strikes the response of one specific SDK call site; in the test that exposes it issues 900+ requests in total, and only four come from that call site. Rainmaker detects the bug consistently because its call-site–targeted policy systematically exercises every unique call site (Section 2.5).

Per-pattern findings. No error handling (29 bugs). All of these bugs are in applications that use the Azure Storage SDK before v12.3.0, which does not retry timeouts, or the CosmosDB SDK, which does not retry in our single-region setting. The applications using these SDKs are expected to handle transient errors themselves but have no error handling; the bugs manifest in system tests when Rainmaker injects faults into the REST calls.

Table 5: New bugs detected by Rainmaker across the evaluated applications.

Application	No Error Handling	Throw New Exception	Semantic Violation	State Divergence	Total
Alpakka	0	0	1	1	2
AttachmentPlugin	0	0	0	2	2
BotBuilder	0	2	0	2	4
DistributedLock	0	2	0	0	2
EF Core	7	0	0	0	7
FHIR Server	11	0	0	0	11
Insights	0	10	0	0	10
IronPigeon	0	1	0	0	1
Orleans	0	5	2	11	18
Sleet	0	2	0	0	2
Storage.NET	11	1	1	1	14
Total	29	23	4	17	73

Throwing unrelated exceptions (23 bugs). Rainmaker triggers these bugs by timing out the response path of a request that has already taken effect at the cloud service. The SDK's automatic retry then collides with an invalidated precondition and produces an error code (e.g., 409 `BlobAlreadyExists`) unrelated to the original timeout. While the bug in Figure 2 is captured by an assertion on the number of created blobs, many of the other bugs are caught by Rainmaker's exception oracle, as the surfaced exception is mostly inconsistent with the injected fault.

Silent semantic violations (4 bugs). All four are caused by the SDK retrying a non-idempotent REST API (e.g., `ReceiveMessagesAsync`). Because the retried operation succeeds at the cloud service, no exception is thrown; the application continues on its happy path while the underlying operation has been performed multiple times. Rainmaker catches these bugs through its post-condition assertions on resource state, e.g., the number of queue messages.

State divergence (17 bugs). Some applications optimistically update local data structures to track remote-resource state and fail to roll back the local update when the corresponding REST call fails. Rainmaker reproduces these bugs by injecting a 5XX on the request path or a timeout on the response path, leaving the local state desynchronized

Table 6: The breakdown of the number of bugs captured by the fault injection policies and by the test oracles. For EF Core and FHIR Server, the four policies reduce to two because the CosmosDB SDK does not retry in our single-region setting. For the exception oracle ("Exp."), we differentiate unit tests and system tests. One bug can be exposed by multiple fault injection policies.

Application	**Fault Injection Policy**				**Test Oracle**		
	$\mathbf{P_1}$	$\mathbf{P_2}$	$\mathbf{P_3}$	$\mathbf{P_4}$	**Exp. (Unit)**	**Exp. (Sys)**	**Assert.**
Alpakka	2	1	1	0	0	0	2
AttachmentPlugin	0	1	2	1	2	0	0
BotBuilder	2	1	2	1	3	0	1
DistributedLock	2	0	0	0	2	0	0
EF Core	7	n/a	7	n/a	0	7	0
FHIR Server	11	n/a	11	n/a	0	11	0
Insights	10	0	0	0	10	0	0
IronPigeon	1	0	0	0	1	0	0
Orleans	17	11	11	11	16	0	2
Sleet	2	0	0	0	2	0	0
Storage.NET	13	12	12	12	2	11	1
Total	67	26	46	25	38	29	6

from the cloud. Although the inconsistencies do not immediately raise exceptions, Rainmaker still catches them when the test later throws exceptions or fails assertions.

Test-oracle precision. We measured the false-positive rate of Rainmaker's test oracles with respect to the test failures they raised. Across all 11 applications, Rainmaker reports 52 false alarms out of 2,654 test failures, for an overall false-positive rate of

Table 7: Consequences of the bugs found by Rainmaker. One bug can lead to multiple consequences.

Consequence	**Example**	**# Bugs**
Externalizing unrelated exception	IronPigeon-133: deletion operations performed on non-existent resources [61].	36
Operation failure	AttachmentPlugin-277: containers cannot be created due to transient errors [62].	35
Incorrect results or states	BotBuilder-5787: timestamp metadata of blobs is set incorrectly [63].	13
Application crash	FHIR Server-2732: FHIR Server can crash unexpectedly upon transient faults [64].	11
Data loss or inaccessibility	BotBuilder-6407: activities that should be logged are lost [38].	4
Resource leak	Orleans-7790: redundant messages were added to the queue [65].	2

1.96%. The low rate is largely due to the exception oracle, which filters out exceptions raised in test code and exceptions consistent with the injected fault (identified by searching for the injected HTTP status or error code in the exception stack); if Rainmaker directly reported the exceptions thrown by unit tests, it would have reported 3.07× more test failures. We sampled 100 of the exceptions filtered by the oracle and confirmed that none indicated a true bug.

Of the 52 false alarms, 10 came from five Insights tests that deliberately exercise invalid-request paths and assert on specific 4XX error codes; these tests fail when Rainmaker injects a 5XX on the request path because the test assertions check for the original 4XX code. Avoiding these false alarms requires understanding those REST calls, e.g., based on information from the reference run. Note that Rainmaker does not inject faults into an HTTP response that already carries an error code. The remaining 42 false alarms came from 14 Alpakka and IronPigeon tests that use small, hard-coded connection timeouts; the fault-injected delays push the tests past those timeouts, resulting in assertion failures or inconsistent exceptions. These are flaky tests [66] in the software-testing sense. Strictly speaking, flaky-test failures should not be counted as Rainmaker false alarms: detecting flaky tests is not Rainmaker's goal, and the flakiness is genuinely triggered by the injected fault (it could equally be triggered by a slow connection). We nevertheless count them conservatively in the 1.96% rate.

Efficiency. Rainmaker takes 0.57–212.77 machine hours to test each application under the most expensive coverage metric, C_4 (Table 3), as shown in Table 8. All experiments were run on a Windows 10 Pro machine with an AMD Ryzen 9 5900X 3.70 GHz CPU and 32 GB of memory. Over 93.54% of the running time is spent executing the fault-injection test runs; the remaining 0.02–16.61 hours per application go to the vanilla test run and test-plan generation. The test-plan generation itself, which uses a linear

programming solver, takes only 1.46–3.67 seconds across the applications (14–241 constraints and 18–3,214 variables) and is a one-time, offline effort.

Rainmaker's test planning avoids exhaustively injecting faults into every REST call. Figure 8 compares the number of fault-injection test runs required by each coverage metric with the exhaustive baseline. Overall, C_4 (the default coverage) reduces the number of test runs by 64.47% per application on average. For Orleans alone, C_4 eliminates 394,172 test runs (99.04%), because Orleans has stress tests that repeatedly exercise the same SDK API call site in large loops; without this reduction, Rainmaker would need 588 days to test Orleans. In terms of bug-finding effectiveness, C_4 covers all the tests and REST calls covered by C_1–C_3, and therefore detects every bug that C_1–C_3 detect.

2.8 Related Work

Our work focuses on push-button tooling to help developers address emerging reliability challenges of cloud-backed applications. The techniques that embody Rainmaker have lineages of error-handling analysis and fault injection.

Table 8: Running time of Rainmaker in machine hours under the most expensive coverage metric C_4. Test planning includes both the vanilla test run and test plan generation (the latter takes seconds).

Application	**Running Time (Machine Hours)**			**# Fault Injection**
	Test Planning	**Fault Injection**	**Total**	**Test Runs**
Alpakka	0.02	0.97	0.99	196
AttachmentPlugin	0.04	2.83	2.87	416
BotBuilder	0.32	10.36	10.68	888
DistributedLock	0.13	4.97	5.10	572
EF Core	10.65	159.60	170.25	4786
FHIR Server	16.61	196.16	212.77	6428
Insights	0.36	10.97	11.33	3056
IronPigeon	0.06	0.51	0.57	120
Orleans	2.57	53.17	55.74	3804
Sleet	0.11	0.63	0.74	72
Storage.NET	2.30	40.27	42.57	1512

- **Error-handling code analysis.** It is known that error handling is a main root cause of production failures of software systems [18], [67], [68]. Prior work developed static analysis for error-handling code to search missing logs and `TODO`s in error-handling code [67], [69], check error specifications [70], and understand error propagation [68], [71], [72].

- **Fault injection.** Prior work developed fault injection techniques for traditional software applications [17], [73], [74], [75] and for distributed backend systems (e.g., storage and data processing systems) that empower modern cloud services [16], [18], [19], [20], [21], [76], [77], [78], [79], [80], [81], [82], [83], [84]. They implicitly or explicitly target error-handling code. Moreover, many existing fault injection tools support injection at the HTTP layer [47], [48].

Unfortunately, we realized that none of the above techniques support a push-button tool that is generic, widely applicable to cloud-backed applications because they all have one or more of the following limitations. First, many fault injection tools only provide mechanisms without fully automatic policies beyond randomization or oracles beyond crashes [18], [21], [74], [76], [77], [80], [85], [86], [87], [88], [89]. Developers need to manually implement them, which is nontrivial. Second, many techniques use application

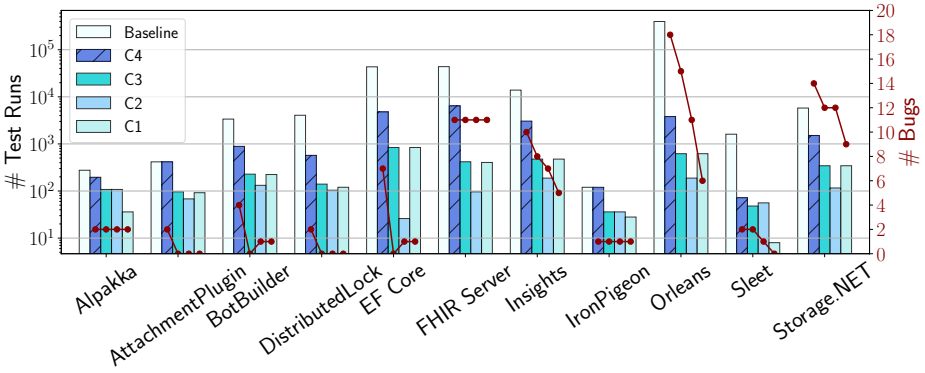


Figure 8: The number of fault-injection test runs under each coverage metric C_1–C_4 in Table 3 (bars) and the number of bugs detected by each metric (dots). Baseline refers to exhaustively injecting faults into all the REST API calls of all the tests.

or domain knowledge to devise policies and oracles [16], [19], [20], [78], [81], [82], [83], [84]. For example, fault injection for distributed databases checks consistency and isolation properties by injecting node crashes and network partitions [16], [90]. These techniques cannot be used as a common, basic utility for diverse types of applications. Third, fault injection at the program level [17], [74], [75], [91], [92] is fundamentally limited to cloud-backed applications: (1) program errors (exceptions and errno) are too coarse-grained to expose certain bug patterns (e.g., silent semantic violations) which need fine-grained injection at the HTTP layer; (2) it is nontrivial to construct exception objects—error handling of cloud-backed applications is based not only on exception types but also on HTTP status codes (Table 1); (3) few program fault injection considers cloud states, but assumes a single program. Rainmaker instead injects faults at the REST interface, effectively addressing the above three limitations.

The early form of cloud-backed applications is mobile apps that interact with cloud backends via REST APIs. Unlike modern cloud services, the cloud backend is specific to an app and is not widely used as a building block of generic application development. Most fault injection tools for mobile apps focus on GUI testing [93], [94]; few consider app-cloud interactions. Rainmaker applies to cloud-backed mobile apps.

2.9 Summary

This chapter presented Rainmaker, a push-button reliability testing tool for cloud-backed applications, together with the four-pattern bug taxonomy that motivates its design. Across 11 mature .NET applications, Rainmaker uncovered 73 new error-handling bugs (55 confirmed and 51 fixed by upstream maintainers) at a 1.96% false-positive rate, demonstrating that systematic HTTP-level fault injection can become a routine part of the cloud-application development workflow rather than a heavyweight one-off audit. The Rainmaker artifacts are publicly available at `https://github.com/xlab-uiuc/rainmaker`. We close this chapter by discussing the

practical limitations of Rainmaker and the directions in which it can be extended in the failure prevention stage.

Dependence on existing test suites. Rainmaker's effectiveness depends on the adequacy of the test suite it is given, in particular tests that interact with cloud services. Such tests are not always abundant: in Microsoft Orleans, for example, only 155 out of 7,002 tests interact with cloud services. A more common practice is to mock the cloud-service REST APIs [95]. A natural future extension is to automatically rewrite mocked tests into tests that Rainmaker can drive against emulators, broadening the test base on which Rainmaker can find bugs.

Cost of testing. Rainmaker is not cheap. To exercise multiple call sites and policies, it may run a test many times, each with a different injected fault. For large test suites this can require significant machine hours, and many bugs trigger more than one test failure during the run. Reducing this cost with test-selection techniques [96] and with incremental testing on changed code is a promising direction for making Rainmaker even cheaper to deploy.

Single-call fault model. Rainmaker's current bug taxonomy and policies target faults that occur during *one* REST API call interaction (the original request and its retries). Bugs that arise from faults across *correlated* API calls—for instance, between a write to one service and a read from another—are outside its scope. The fault space for such multi-call bugs is much larger and requires a richer fault model, which we are actively investigating.

Beyond client-side bugs. Rainmaker can also be extended to expose bugs and behavioral surprises *in cloud services themselves*. Our evaluation revealed several such surprises: the same REST API of AWS S3 has different consistency guarantees in

different regions [97], and the side effects of a non-idempotent API that returns 500 (`InternalServerError`) are often opaque even from the documentation. A natural extension is to use Rainmaker as a differential testing tool that compares the observable behavior of an application against documented service contracts.

Hard-to-fix bugs. Not every bug Rainmaker finds is easy to fix at the application layer. Response-path timeouts, for example, leave the application unable to determine whether its request was applied at the cloud service. Cleanly fixing the entire class of unrelated-exception and silent-semantic-violation bugs likely requires server-side support such as request versioning (e.g., HTTP `ETag`) or transaction-like API semantics, and SDKs that do not blindly retry non-idempotent APIs (a common practice today, as Table 1 shows).

Cross-language support. Our current prototype targets .NET applications, but the design of Rainmaker is language-agnostic. The core idea of injecting faults at an HTTP proxy is independent of the application language, and the only .NET-specific pieces of the prototype are the call-site instrumentation (which uses .NET profiling APIs [98]) and the inheritable thread-local storage that propagates call-site information through asynchronous code paths [49]. Neither facility is unique to .NET; equivalent mechanisms are available in Java and other modern runtimes, and porting Rainmaker to those ecosystems is a natural next step.

Chapter 3 Automatic Root Cause Analysis via Large Language Models for Cloud Incidents

Despite early prevention, production failures are inevitable. When they occur, on-call engineers (OCEs) resort to root cause analysis (RCA) to diagnose the issue. However, RCA is challenging because the relevant evidence, including logs, metrics, traces, and deployment history, is often scattered and noisy. When conducting RCA, engineers rely on troubleshooting guides, which are static documents that hard-code the debugging steps for engineers to refer to. These guides are slow to apply, labor-intensive, and quickly out of date.

To address these limitations, this chapter presents RCACopilot, an automated RCA system that adapts large language model (LLM) reasoning to cloud incident management. Rather than asking engineers to follow static guides by hand, RCACopilot turns the workflow of an experienced on-call engineer into an executable pipeline that runs in two stages, as shown in Figure 9: a *diagnostic information collection* stage and a *root cause prediction* stage.

Diagnostic information collection stage: This is the stage, where the incident is parsed and matched to the pre-defined incident handler. Each incident handler is tailored to a specific alert type. Upon matching the incident with the appropriate handler, RCACopilot proceeds to collect relevant diagnostic data from a variety of sources.

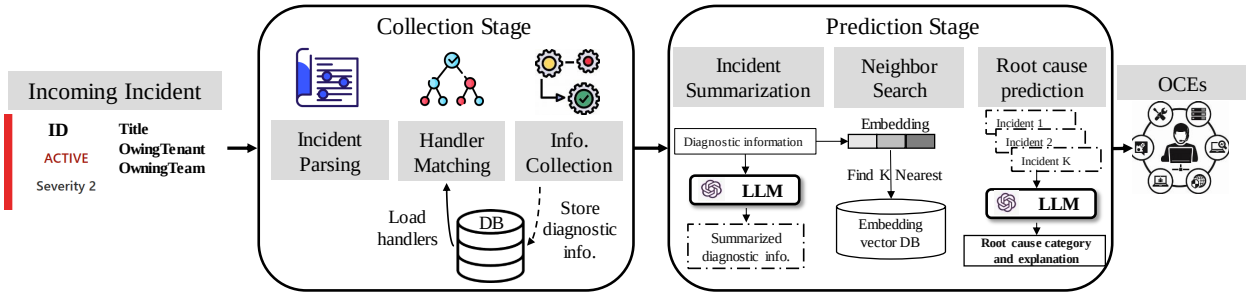


Figure 9: RCACopilot architecture.

Root cause prediction stage: Once the diagnostic information is collected, RCACopilot transitions into the root cause prediction stage. In this phase, RCACopilot applies its predictive module to determine the likely root cause category of the incident. This prediction is not a mere categorization; it is also supplemented with an explanation detailing how RCACopilot arrived at the given prediction. Subsequently, the predicted category label is presented to experienced OCEs for review.

RCACopilot aims to collect multi-source data for RCA. Specifically, for each alert type, an incident handler is constructed, comprising a series of actions to collect diagnostic information. Alert types are used to categorize alerts based on specific monitors. Incidents sharing the same alert type exhibit similar symptoms, though they may stem from different root causes.

The RCACopilot incident handler is a workflow that consists of a series of actions. Each action is a function that can be executed to collect specific diagnostic information from a target data source. OCEs can build and modify these handlers based on their expertise. The handler includes three distinct actions: *scope switching action*, *query action*, and *mitigation action*, which will be explained in Section 3.3.1. Each action generates an output, guiding the control flow of the incident handler.

3.1 Background

3.1.1 Incident Root Cause Analysis

In the realm of cloud services, an *incident* refers to any event that disrupts normal service operations or causes degradation in the quality of service. When such an incident occurs, root cause analysis (RCA) is performed to identify the underlying issue. RCA is a multi-stage process: *data collection* (gathering incident-related data from logs, metrics, traces, and alerts), *data analysis* (identifying patterns, anomalies, or correlations), and *hypothesis verification* (formulating and verifying possible root causes). Given the complexity and dynamism of cloud systems, along with the immense volume of telemetry

involved, RCA is a challenging task that demands substantial expertise and time. To put the scale into perspective, the global email service that motivates this work delivers over 150 billion messages per day; ensuring its smooth operation demands an RCA approach that is both fast and accurate.

3.1.2 Multi-Source Data: Opportunities and Challenges

Different data sources offer different perspectives on system state. Logs record event sequences, metrics reflect performance trends over time, and distributed traces reveal the propagation of requests across services. Integrating these sources gives a more comprehensive view of the system, enabling more accurate diagnosis. Multi-source data also facilitates correlation and causality analysis, which are crucial for RCA: patterns or anomalies in the relationships between sources often indicate the root cause of an incident.

Effectively leveraging multi-source data is, however, far from straightforward. The sheer volume and complexity of data from different sources can be overwhelming, making it difficult to extract meaningful signals. Different sources may also provide inconsistent or conflicting information, and real-world data is often noisy, complicating analysis and risking false conclusions.

Collecting and analyzing this data manually is not only laborious and error-prone, but can also be challenging due to varying levels of available information, what we term the "information spectrum". The "information spectrum" describes a continuum of information availability, ranging from situations with too little information to those inundated with an excess. At either end of this spectrum, RCA can become particularly challenging. The relevant information for RCA might be buried within the voluminous data, leading to an information overload for OCEs. OCEs may find it challenging to quickly pinpoint the relevant information amidst the sea of data, hindering efficient

incident resolution. Conversely, OCEs also encounter situations where they lack the necessary information to understand and address the root causes of incidents accurately.

3.1.3 Limitations of Troubleshooting Guides

Traditional Troubleshooting Guides (TSGs) represent an early attempt to leverage multi-source data: they instruct OCEs to gather and analyze data from various sources to diagnose and resolve incidents. Despite their usefulness, TSGs face several inherent limitations:

- *Manual data integration.* TSGs typically require OCEs to gather data from different sources by hand, a time-consuming and error-prone process. Even with TSG-recommendation techniques [13], this manual workflow remains a significant source of stress and burnout for OCEs.
- *Outdated information.* TSGs are static documents and often lag behind evolving systems and newly discovered root causes. New monitors, dashboards, or features are not always reflected in the TSGs that reference them.
- *Insufficient details and coverage.* High-level instructions in TSGs (e.g., "check the logs") often lack specifics, forcing OCEs into additional research and prolonging mitigation. TSGs may also overlook common checks, such as disk space, leading to partial or inadequate resolutions.

These limitations motivate the more flexible, data-driven approach taken by RCACopilot.

Different from previous work [99], which employs AI techniques to generate automated workflows from existing TSGs, our goal is to enable experienced OCEs to construct an automated pipeline for incident diagnosis. This approach allows OCEs to be directly assisted in identifying the root cause without the need to investigate intermediate diagnostic information, though they still have the option to do so.

We envision a future in which root cause analysis is predominantly automated, requiring minimal manual verification only when necessary. Our approach seeks to provide OCEs with timely, relevant, and accurate information for specific incidents, leading to more efficient RCA. By leveraging LLMs to predict the root cause category, our research aims to alleviate the stress and cognitive load associated with incident management, ultimately enhancing the efficiency and effectiveness in addressing incidents.

3.2 Insights from Production Incidents

To ground the design of RCACopilot in real-world incident behavior, we conducted a qualitative study of a one-year dataset of 653 incidents from the global email service that motivates this work. Each incident was reviewed and categorized by experienced OCEs based on its characteristics, the source of the issue, and its impact on the system. Our analysis yielded three insights that shaped RCACopilot's design. To make the discussion concrete, Table 9 lists ten representative incidents from the dataset, grouped by their root-cause category; we will refer to these by number in the insights below.

Insight 1: Determining the root cause from a single data source is hard. A single monitor alert is rarely enough to identify the underlying issue. Consider Incident 2 in Table 9, in which a single server failed to perform DNS resolution because of UDP hub-port exhaustion on a front-door machine. The monitor's initial alert reports a connection failure between the mail server and the front-door server but says nothing about DNS, leaving the root cause ambiguous. Several plausible explanations remain consistent with the alert: a network failure between the mail server and the front-door data center, a flood of connection requests depleting the front-door pool, multiple front-door crashes reducing capacity, or even a false alarm from a misconfigured monitor. Resolving such ambiguity requires correlating multiple data sources—logs, metrics, and

Table 9: Examples of cloud incidents in different root-cause categories.

No.	Sev.	Scope	Category	Occur.	Symptom	Cause
1	1	Forest	AuthCertIssue	3	Tokens for requesting services could not be created; several services reported user outages.	A previous invalid certificate overrode the existing one due to misconfiguration.
2	2	Machine	HubPortExhaustion	27	A single server failed to perform DNS resolution for incoming packets.	The UDP hub ports on the machine had been exhausted.
3	2	Forest	DeliveryHang	6	The mailbox delivery service hung for a long time.	Number of messages queued for mailbox delivery exceeded the limit.
4	2	Forest	CodeRegression	15	An SMTP authentication component's availability dropped.	Bug introduced by a recent code change.
5	2	Forest	CertForBogusTenants	11	The number of concurrent server connections exceeded a limit.	Spammers abused the system by creating bogus tenants with connectors that share a certificate domain.
6	1	Forest	MaliciousAttack	2	Forest-wide processes crashed over threshold.	An exploit was launched in remote PowerShell by serializing a malicious binary blob.
7	2	Forest	UseRouteResolution	9	Poisoned messages sent to the forest made the system unhealthy.	A configuration service was unable to update the settings, leading to crashes.
8	2	Forest	FullDisk	2	Many processes crashed and threw I/O exceptions.	A specific disk was full.
9	2	Forest	InvalidJournaling	11	Messages stuck in the submission queue for a long time.	A customer set an invalid value for the Transport config and caused a TenantSettingsNotFoundException.
10	3	Forest	DispatcherTaskCancelled	22	Normal-priority messages across a forest had been queued in submission queues for a long time.	A network problem caused the authentication service to be unreachable.

stack traces—which an OCE must do manually under time pressure. This insight motivates RCACopilot's emphasis on *multi-source* diagnostic data.

Insight 2: Incidents from similar root causes recur within a short time window. Most recurring incidents (93.80% in our dataset) reappear within 20 days, as shown in Figure 10. Take Incident 9 in Table 9, triggered by an invalid customer configuration that left messages stuck in the submission queue: incidents in this category recurred 11 times in just 15 days. Other categories such as DispatcherTaskCancelled (Incident 10) and DeliveryHang (Incident 3) recurred 22 times in a week and 6 times in a month, respectively. Several factors contribute to this temporal clustering: unresolved root causes from the initial response can let the same issue re-emerge, systemic vulnerabilities can be repeatedly exploited if not addressed, and external dependencies

can produce a cascade of similar incidents. This insight motivates RCACopilot's nearest-neighbor retrieval over historical incidents weighted by temporal proximity: recent past incidents are usually the best demonstrations for the current one.

Insight 3: Incidents with new root causes are common and harder to analyze. TSGs assume the root cause is known and documented. When a new, previously unencountered root cause appears, no TSG exists, and OCEs struggle to identify the underlying issue. In our dataset, 24.96% (163 of 653) of incidents involve a new root-cause category, as shown in Figure 11. Examples include Incident 1 (a Severity-1 misconfiguration that blocked authentication-token generation and caused widespread outages), Incident 6 (a malicious-blob exploit launched in remote PowerShell that had not been encountered before), and Incident 5 (a novel spam-abuse pattern in which spammers abused the system by creating bogus tenants with connectors sharing a certificate domain). If OCEs spend their time searching for nonexistent TSGs, the impact escalates further. This insight motivates RCACopilot's ability to generate new root-cause-category labels, with explanations, when no historical match is sufficiently close.

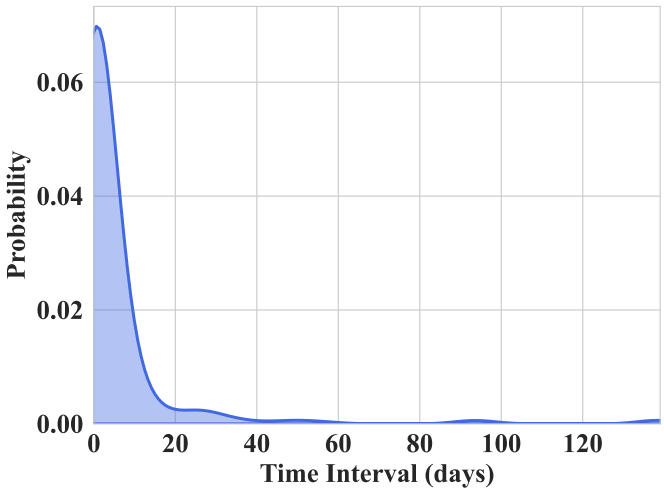


Figure 10: Recurring incidents proportion vs. time interval.

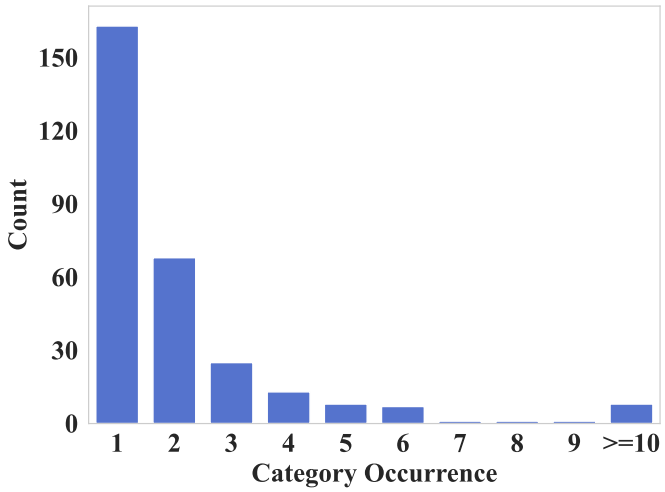


Figure 11: Distribution of incident category frequency.

3.3 Incident Handler

The decision-making process that OCEs employ when handling an incident resembles a decision tree's control flow. The root node in the incident handler is the incident alert type, which is gathered from the system monitor. We distilled OCE operations into three actions when constructing the incident handler. As OCE operations can be similar across different incident types (e.g., conducting a common disk check or query to a database), we designed RCACopilot handler actions to be reusable across all handlers. We also maintain the versions of the handlers in the database, which can be used to track their historical changes.

RCACopilot's incident handlers are constructed manually first and can be updated and modified dynamically by OCEs, allowing them to stay abreast with the most recent system changes and newly discovered root causes. For instance, when a new metric is introduced into the system, OCEs only need to construct a new action to collect the relevant data and incorporate it into the corresponding incident handler, which can ensure timely adaptation. The contribution of RCACopilot is to provide a platform to facilitate the engineers to build their own handlers. Similar to existing cloud infrastructure automation platforms such as Terraform and Ansible, which allow

engineers to manage their distributed cloud systems, RCACopilot allows engineers to build the incident handlers for their own scenarios.

3.3.1 Handler Action

RCACopilot leverages the synergy of multi-source data. The system uses predefined reusable actions in the incident handler to automatically collect relevant diagnostic information from diverse sources. The automated integration of data not only saves time but also reduces the likelihood of human error. It provides a more comprehensive view of the system state, facilitating efficient and accurate incident resolution. This significantly lightens the workload of OCEs, reducing stress and burnout, and enhancing the effectiveness of the incident resolution process. The action in the handler could be one of the following:

Scope switching action: This action facilitates precision in RCA by allowing adjustments to the data collection scope based on the specific needs of each incident. For instance, as depicted in Figure 12, if an alert originates at the 'forest' level, signifying an issue within a specific forest, and the problem type is identified as 'Busy Hub', the scope switching action can adjust the scope to the 'machine' level. This modification allows for a more fine-grained investigation, specifically assessing if a singular hub server is overly taxed.

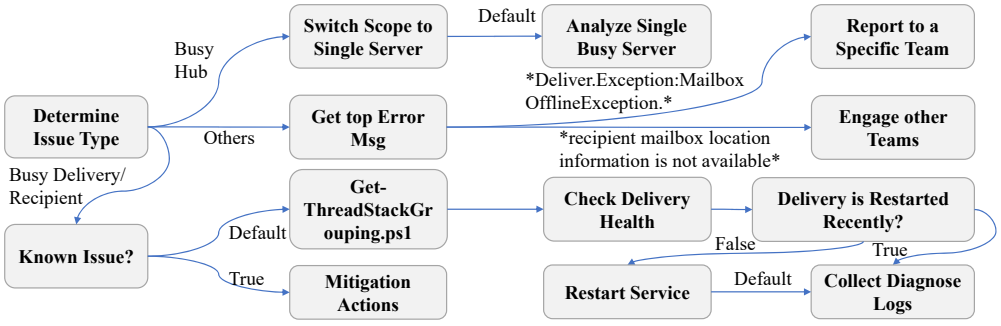


Figure 12: A RCACopilot handler for the "too many messages stuck in the delivery queue" alert.

The implementation of this action ensures that we efficiently navigate the information spectrum. When the investigation requires a more targeted approach, this action can narrow the data collection scope. Conversely, if a more holistic view is necessary, it can widen the scope, say from a single machine to an entire forest. This flexibility contributes to a more balanced and effective diagnostic data collection.

Query action: A query action can query data from different sources and output the query result as a key-value pair table. This type of action can also be hooked to executing a specific script with pre-defined parameters. Usually, scripts are internal automatic investigation tools for a service, and only the service team has access to the tools.

For instance, in Figure 12, the "Known issue?" action node queries the database to see whether the current incident is a known one or not based on its alert messages. If it is a known issue, execution flow will enter the "True" branch to give mitigation actions directly. Otherwise, a query script that can aggregate threads with the same stack traces will be executed. It will obtain an instantaneous list of the stacks on all the managed threads in the target process and then group common stacks together in order to identify potential deadlocks/blocking code paths in the process.

The query action can also output an enum value to decide the next action node to execute, e.g., after getting the top error message on the exception stack traces, i.e., the "Get top error msg" node, the next action node to be run depends on the exception type. Based on the error messages, a specific team will be reported and engaged, as shown in Figure 12.

Mitigation action: This action refers to the strategic steps suggested to alleviate an incident, such as "restart service" or "engage other teams", as depicted in Figure 12. It is important to note that handlers do not always provide exact mitigation strategies for every incident, due to handlers' pre-defined nature, which may not cover all possible situations. In cases where the incident handler is uncertain, it will offer intermediate diagnostic information to the OCEs without mitigation.

3.3.2 Multi-source Diagnostic Information

RCACopilot's diagnostic information collection stage serves as a valuable tool for OCEs by aggregating data from a myriad of sources. OCEs only need to customize the action in the handler to acquire the diagnostic information from a target source. For instance, as illustrated in Figure 13, RCACopilot can assimilate diverse data such as error logs, exception stack traces, and socket metrics related to a specific incident. The error log and exception stack trace alone do not provide sufficient insight to identify the root cause of the incident. However, when supplemented with the socket metrics, a more comprehensive picture emerges. In this example, it is clear that the UDP socket is exhausted, which is the root cause.

```
DatacenterHubOutboundProxyProbe probe log result from [MachineID].
Total Probes: 2, Failed Probes: 2
 Id  Level  Created                 Description
 --  -----  -------                 -----------
 2   Error  11/21/2022  2:04:20 AM  Probe result
 2   Error  11/21/2022  1:49:20 AM  Probe result
Failed probe error:
Name: No such host is known.
A WinSock error: 11001 encountered when connecting to host: [HOST NAME]
Count: 2
…
Exceptions:
InformativeSocketException: No such host is known.
A WinSock error: 11001 encountered when connecting to host: [HOST NAME]
at TcpClientFactory.Create(...)
at SimpleSmtpClient.Connect(...)
…
Total UDP socket count: 15276
Total UDP socket count by process and processId (top 5 only):
14923: Transport.exe, 203736
```

```
15: w3wp.exe, 102296
8: svchost.exe, 4748
7: Microsoft.Transport.Store.Worker.exe, 74060
7: Microsoft.Transport.Store.Worker.exe, 87724
```

Figure 13: Diagnostic information for hub port exhaustion.

In the case of new incidents, RCACopilot can perform a range of common checks, such as evaluating the provisioning status or analyzing thread stacks. This assists OCEs in gaining a holistic understanding of the situation. Note that the information collected is pre-defined in the actions of the RCACopilot handler, ensuring that only relevant data is gathered, thus avoiding overwhelming information that is unnecessary. By providing this comprehensive diagnostic information, RCACopilot empowers OCE teams to troubleshoot issues efficiently. They can use the gathered information as guidance to address incidents more effectively.

3.4 LLMs for Root Cause Analysis

Upon thorough investigation, each incident within our service is manually assigned a root cause category by our seasoned OCEs. OCEs will use the categories to classify the historical incidents and guide the new incoming incidents' RCA. However, reasoning over the incidents and inferring their categories is time-consuming and potentially overwhelming for OCEs, who have a tight time budget. Given this, we have identified the categorization of incident root causes as our primary downstream task.

Recently, LLMs have demonstrated remarkable capabilities in understanding the context of downstream tasks and generating relevant information from demonstrations, making them a possible choice for incident RCA. However, reasoning over the incident root cause is not a simple task, and LLMs may not be able to achieve the optimal results on long-tail or domain-specific tasks without any guidance [100], [101].

Recently, Ahmed et al. [102] proposed to fine-tune LLMs with domain-specific datasets for generating root causes of an incident just by leveraging the title and summary information available at the time of incident creation. While they have demonstrated the promise of LLMs in incident root causing, fine-tuning has several limitations: (1) as accurate RCA requires various sources of complex unstructured or semi-structured data (e.g., logs, telemetry, traces, and natural language descriptions), just using a generic title and summary might miss useful signals to reach conclusive diagnosis details; (2) fine-tuning is costly and may require a huge volume of training samples; and (3) it is challenging to continuously update a fine-tuned GPT model with the evolving nature and scope of incidents; therefore, such models are prone to generate more hallucinated results over time.

Chain-of-Thoughts (CoT) prompting is a gradient-free technique that elicits LLMs to generate intermediate reasoning steps that lead to the final answer. In few-shot CoT prompting, a few manual demonstrations are composed of a question and a reasoning chain that leads to an answer for each of them. Inspired by these ideas, diagnostic information provided by RCACopilot handlers can be used as ingredients for the reasoning process of the incidents.

3.4.1 Embedding Model

Our observation is that the *semantics of incidents can be revealed from the context in which the diagnostic information is described.* A common approach to extracting such contextual semantics involves the use of embedding models. The objective is to map the diagnostic information into an embedding space (i.e., numeric vector space), where the distances between vectors represent the semantic similarity of incidents. Choosing a computationally efficient embedding model allows us to preserve accuracy while handling a large number of incidents.

We employ FastText as our embedding model, which is efficient, insensitive to text input length, and generates dense matrices, making it easy to calculate the Euclidean distance between similar vectors. Furthermore, since our downstream task is domain-specific to incident root cause reasoning, and the incident-related information is internal to our company, we opt to train a FastText model on our historical incidents rather than using a pre-trained large language model as our embedding model, which is costly and inefficient. Additionally, we provide users with the flexibility to customize their embedding model if desired.

3.4.2 Nearest Neighbor Search

Incidents are heterogeneous, making it impractical to combine all past incidents' information for sampling due to the prompt length limitations, even after summarization. To selectively choose past cases as samples in the prompt, we design a new similarity formula:

$$Distance(a, b) = \|a - b\|_2$$

$$Similarity(a, b) = \frac{1}{1 + Distance(a, b)} \cdot e^{-\alpha\,|T(a)-T(b)|}$$

to calculate the similarity between two incidents. It first computes the Euclidean distance for every pair of incident vectors. Importantly, it also takes into account the temporal distance between incidents. Here, $T(x)$ stands for the date of incident x. This consideration of temporal distance is crucial as it influences the relevance of past incidents to the current ones. After calculating similarities, we select the top K incidents from different categories as demonstrations for the LLM. This approach ensures a diverse and representative set of incidents for effective LLM reasoning. The values of α and K have been determined as 0.3 and 5, respectively, through empirical evaluation.

3.4.3 Diagnostic Information Summary

LLMs have shown potential for automatic summarization [103]. Nonetheless, the length of the diagnostic information collected from RCACopilot handlers is often too extensive. As shown in Figure 13, the diagnostic information of an incident can have more than 2000 tokens with low readability of the log messages. The considerable number of tokens in the incident description can pose challenges for the LLM to effectively process and may introduce noise. Therefore, feeding the diagnostic information of an incident directly into the LLM to make a prediction could not be an ideal choice, let alone using the information from multiple sources. In this regard, we add another layer to leverage the LLM's ability to summarize the diagnostic information first before making the diagnosis reasoning. We construct the prompt in the way of Figure 14. We ask the LLM to summarize the diagnostic information into 120–140 words without outputting any unrelated information. This summarization process makes the diagnostic information more concise and informative, which forms the basis for the later CoT prompting. Figure 15 illustrates a more readable and concise text generated by RCACopilot, which is a summary (113 tokens) of the previous diagnostic information example in Figure 13, highlighting the key details such as the number of UDP ports used and the process utilizing the most. Specifically, we employ the tiktoken [104] tokenizer to count text tokens.

3.4.4 Prediction Prompt Construction

CoT prompting is a gradient-free technique that guides LLMs to produce intermediate reasoning steps leading to the final answer. In few-shot CoT prompting, several

> *"Please summarize the above input. Please note that the above input is incident diagnostic information. The summary results should be about 120 words, no more than 140 words, and should cover important information as much as possible. Just return the summary without any additional output."*

Figure 14: Prompt to summarize diagnostic information.

"The DatacenterHubOutboundProxyProbe has failed twice on the backend machine, with both failures due to a WinSock error 11001 indicating that the host is unknown. This error was encountered while attempting to connect to the host. The error is associated with the EOP service and has not been notified yet. The failure context suggests the same issue. ***The total UDP socket count is 15276, with the majority being used by the Transport.exe process.*** *The issue seems to be related to the SMTP connection and requires further investigation."*

Figure 15: The summarized diagnostic information.

demonstrations include a question and a reasoning chain that direct the answer. By drawing inspiration from automatically constructing the prompt to form the reasoning chains [105], we can view the summarized diagnostic information and the labeled root cause categories as questions and reasoning, so finding the nearest incident neighbor is the automatic reasoning chain construction, aligning with the CoT prompting context well. Note that we use the original incident information to do the embedding and nearest neighbor search, and use the corresponding summarized information as part of demonstrations in the prompt. We construct the prompt as in Figure 16 to ask the LLM to choose the most likely incident that has the same root cause as the current incident, and we explicitly push the LLM to reason by using "give your explanation" indications in the prompt.

Context: The following description shows the error log information of an incident. Please select the incident information that is most likely to have the same root cause and **give your explanation** (just give one answer). If not, please select the first item "Unseen incident".
Input: The DatacenterHubOutboundProxyProbe probe result from [BackEndMachine] is a failure ...
Options:
A: Unseen incident.
B: The DatacenterHubOutboundProxyProbe has failed twice ... *category:* ***HubPortExhaustion****.*
C: There are 62 managed threads in process TransportDelivery ... *category:* ***AuthCertIssue****.*

Figure 16: The prompt to predict incident category.

3.5 Implementation

We have developed and deployed RCACopilot using a combined total of 58,286 lines of code, comprising 56,129 lines of C# and 2,157 lines of Python. To facilitate the building of RCACopilot's incident handlers, we implemented the handler-construction interface as a web application. To support a new alert type, OCEs only need to add a new handler in the construction GUI based on their expertise; once constructed, the handler is stored in the database and can be modified later by adding new action nodes or removing old ones. The diagnostic information collection module of RCACopilot has been deployed across more than 30 service teams within Microsoft for over four years, demonstrating that the handler-based design is operationally workable at production scale.

3.6 Evaluation

We evaluate RCACopilot along three axes: (1) effectiveness and efficiency as an on-call system that predicts root-cause categories, (2) the contribution of RCACopilot's individual design components, and (3) deployment fitness and result trustworthiness. Across all experiments, RCACopilot achieves micro-F1 of 0.766 and macro-F1 of 0.533, outperforming all baselines at a sub-five-second per-incident inference cost; ablations confirm that the diagnostic information handler, LLM-based summarization, and chain-of-thought prompting each contribute to the final accuracy; and three independent runs of the end-to-end pipeline produce stable results, with micro-F1 above 0.70 and macro-F1 above 0.50 on every run.

Target system and dataset. We evaluate RCACopilot on a global email service that processes roughly 150 billion messages per day. The service handles the secure transmission of email between users via SMTP, IMAP, and POP3, and interacts with a large number of upstream and downstream services, making it representative of complex real-world cloud systems. We collected a one-year dataset of 653 incidents from this

service, each representing a complex issue in a large-scale, globally distributed deployment. Experienced OCEs manually label each incident with a root-cause category, providing the ground truth. We split the dataset into 75% training and 25% testing.

Models and baselines. RCACopilot uses GPT-4 (8K) as its default backbone, with GPT-3.5-turbo reported as a comparison point. We compare RCACopilot against five baselines that span the space of approaches one would consider for this task:

- **XGBoost** [106]: a parallel tree-boosting classifier widely used in networking diagnosis.
- **FastText** [107]: a lightweight text-embedding classifier that has been used in fault-injection studies for root-cause diagnosis.
- **Fine-tuned GPT** [102]: GPT-3.5 fine-tuned on the training set, predicting the category directly from the original diagnostic information (no chain-of-thought prompt).
- **GPT-4 Prompt**: a variant of RCACopilot that directly predicts the category from RCACopilot's summarized diagnostic information, without retrieved historical-incident demonstrations.
- **GPT-4 Embed.**: a variant of RCACopilot that replaces the FastText embedding model with a GPT embedding, leaving everything else unchanged.

Effectiveness and efficiency. We measure effectiveness with micro and macro F1-score. The micro-F1 aggregates the performance across all classes, weighting each sample equally; macro-F1 averages per-class performance, giving equal weight to each category and thus penalizing poor performance on rare ones. RCACopilot achieves a micro-F1 of 0.766 and a macro-F1 of 0.533 on the testing set.

As shown in Table 10, RCACopilot outperforms other approaches, and it tends to incur an acceptable higher runtime overhead. The performance of baseline approaches is poor, since multiple root cause categories exhibit a long-tail (imbalanced) distribution, as shown in Figure 11, and traditional machine learning models (FastText and XGBoost) and fine-tuning GPT models need a large amount of training data to produce accurate predictions. Directly employing a GPT-4 prompt or GPT-4 embedding approach without our design lacks domain-specific knowledge for GPT-4 to make decisions. On the contrary, RCACopilot leverages the powerful LLM to learn the domain-specific knowledge from minimal cases, so that it can achieve the best performance. Results indicate that RCACopilot not only provides higher accuracy but also maintains a reasonable level of efficiency, making it a suitable choice for incident root cause analysis. When facing incidents that RCACopilot has never seen before, RCACopilot is capable of generating a new category keyword to depict the new incident case. For example, Incident 8 in Table 9 (a full-disk failure) is a new incident case that RCACopilot had never encountered. RCACopilot's prediction component predicts it as a new category "I/O Bottleneck". Although OCEs subsequently categorize it as "DiskFull" in post-investigation, the fundamental aspects of the problem identified by RCACopilot align closely with the human-derived label. The corresponding RCACopilot's

Table 10: Effectiveness of different methods.

Method	**F1-score**		**Avg. Time (s)**	
	Micro	**Macro**	**Train.**	**Infer.**
FastText [107]	0.076	0.004	10.592	0.524
XGBoost [106]	0.022	0.009	11.581	1.211
Fine-tuned GPT [102]	0.103	0.144	3192.000	4.262
GPT-4 Prompt	0.026	0.004	–	3.251
GPT-4 Embed.	0.257	0.122	1925.000	3.522
RCACopilot (GPT-3.5)	0.761	0.505	10.562	4.221
RCACopilot (GPT-4)	**0.766**	**0.533**	10.562	4.205

explanation, illustrating how it arrived at the "I/O Bottleneck" categorization, is provided in Figure 17.

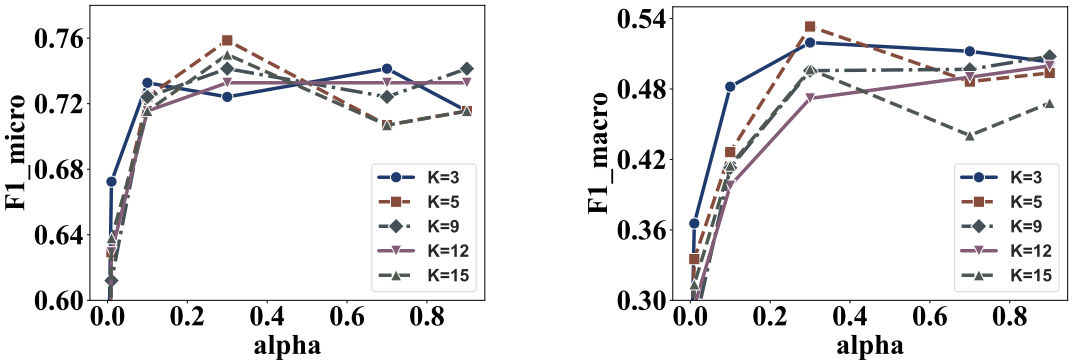
Ablation analysis. To understand how RCACopilot's individual components facilitate root cause analysis, we ablate the prompt context and the summarization and retrieval steps.

Evaluation on diagnostic information. We compare the diagnostic information collected by the handlers against two other kinds of incident-related context: AlertInfo and ActionOutput. AlertInfo includes the alert type and alert scope; the alert type is a pre-defined anomaly description from a monitor (e.g., an exception type from an external monitor), which reflects a symptom of the incident rather than its root cause, and the alert scope is the extent of the incident (e.g., a single machine). ActionOutput is the output of the executed RCACopilot handler actions, hashed as key–value pairs. As shown in Table 11, using diagnostic information alone outperforms every other source and combination on micro-F1 (0.689) and is essentially tied for the best macro-F1 (0.510, vs. 0.511 for AlertInfo combined with DiagnosticInfo). Interestingly, mixing the diagnostic information with the other sources does not enhance RCACopilot's predictive capability: an excess of information can negatively impact the LLM's prediction performance.

Evaluation on LLM summarization. As Table 11 also shows, using the summarized rather than the raw diagnostic information yields the best scores overall (micro-F1 of 0.766 and macro-F1 of 0.533), improvements of 0.077 and 0.023 respectively over the

The prediction of "I/O Bottleneck" was made based on the occurrence of System.IO.IOExceptions within crucial functions handling input/output operations, suggesting an issue with data processing. The nested exception within the DiagnosticsLog module reinforces this notion. These errors, combined with crashes on different backend machines, point to a system struggle with handling data flow.

Figure 17: RCACopilot's explanation of an incident.



(a) F1 micro.

(b) F1 macro.

Figure 18: Effectiveness of using different K and α.

unsummarized diagnostic information. The summarization step effectively condenses the information, allowing more efficient and accurate processing of incident data.

Evaluation on few-shot chain-of-thought reasoning. The GPT-4 Prompt baseline in Table 10, which predicts the category directly without any retrieved samples, achieves only 0.026 micro-F1 and 0.004 macro-F1. Figure 18 compares RCACopilot's performance under different numbers of retrieved samples K and values of α in the chain-of-thought reasoning. The best combination is $K = 5$ and $\alpha = 0.3$, which achieves the highest F1 scores. Note that more samples do not always improve RCACopilot's predictions, and the value of α plays an important role: when α is appropriate, RCACopilot better captures the time relationships between incidents, leading to more accurate predictions.

Deployment status and scale. We have deployed RCACopilot's diagnostic information collection module across more than 30 teams within Microsoft, where it has been in active use for over four years. The system is tailored to each team's specific requirements, with custom handlers built for each unique setting; not all handlers are enabled in the production environment, as some are still under development and testing. Table 12 lists the ten teams that use the most RCACopilot incident handlers. The average running time per incident ranges from 15 seconds to 841 seconds across all

teams; the highest running time is attributable to that team's large-scale and complex system infrastructure. The root cause prediction module has also been rolled out in the Transport service.

As part of our commitment to continuous improvement, we incorporate a feedback mechanism into incident notification emails to collect user perspectives from OCEs. According to the collected feedback, most OCEs expressed satisfaction with RCACopilot. Despite the manual effort involved in creating a new incident handler, OCEs find the process convenient when reusing and modifying handler actions from the database. RCACopilot saves OCEs a significant amount of time in collecting diagnostic information, triaging incidents, performing mitigation, and doing postmortem analysis.

3.7 Summary

This chapter presented RCACopilot, an automated root-cause-analysis system that adapts large language model reasoning to cloud incident management. Grounded in three insights from a one-year study of 653 incidents on a production email service, RCACopilot composes per-alert-type incident handlers from three reusable actions to collect multi-source diagnostic data, summarizes the data with an LLM to fit within prompt budgets, and predicts the root-cause category through nearest-neighbor retrieval over historical incidents combined with chain-of-thought prompting. On a real-world

Table 11: Effectiveness of different prompt context for RCACopilot. ✓$^{\text{sum.}}$ stands for the summarized diagnostic information.

Data Source			F1-score	
AlertInfo	**DiagnosticInfo**	**ActionOutput**	**Micro**	**Macro**
	✓		0.689	0.510
	✓$^{\text{sum.}}$		**0.766**	**0.533**
✓			0.379	0.245
✓	✓		0.525	0.511
✓		✓	0.431	0.247
	✓	✓	0.501	0.449
✓	✓	✓	0.440	0.349

Table 12: Teams in Microsoft using RCACopilot to automatically collect diagnostic information.

Team	Avg. exec. time (s)	# Enabled handlers
Team 1	841	213
Team 2	378	204
Team 3	106	88
Team 4	449	42
Team 5	136	41
Team 6	91	34
Team 7	449	32
Team 8	255	32
Team 9	323	31
Team 10	22	18

dataset, RCACopilot achieves micro-F1 of 0.766 and macro-F1 of 0.533 with sub-five-second inference latency, substantially outperforming classical machine-learning baselines and naïve LLM approaches; the diagnostic information collection module has been deployed in production across more than 30 service teams. We close the chapter by discussing the limitations of RCACopilot and the directions in which it can be extended.

Dependence on the LLM backbone. RCACopilot's prediction effectiveness is bounded by the reasoning ability of its underlying LLM. Our current evaluation uses OpenAI's GPT-3.5 and GPT-4 backbones, and we have not yet explored the broader landscape of available models, such as open-weight LLMs that could be fine-tuned in-house, or smaller specialized models distilled from larger ones. As LLMs continue to improve, RCACopilot is expected to benefit accordingly; conversely, its performance may vary with the specific strengths and weaknesses of whichever model it is configured to use.

Dependence on label quality. RCACopilot's accuracy is influenced not only by the model but also by the quality of the root-cause labels in its retrieval database. In this work, all labels were assigned manually by experienced OCEs in the partner team, drawing on their established practice for incident triage. Lower-quality or inconsistent

labels would weaken the nearest-neighbor signal that drives chain-of-thought reasoning. Future work includes developing semi-automated labeling assistance, label auditing, and label-disagreement resolution to keep the retrieval database healthy as it grows.

Generalization across services. The diagnosis-information-collection module of RCACopilot has been deployed across more than 30 service teams, but the prediction evaluation in this work is limited to a single email service. A valuable line of future work is to evaluate RCACopilot end-to-end across services with different telemetry signatures and incident structures, to better understand its generalizability and to identify service-specific tuning that may be required.

Coverage of the alert-driven workflow. RCACopilot's incident handler is designed to respond to alerts generated by monitors and watchdogs. When an alert type has a designated handler, RCACopilot reaches it with full coverage; however, RCACopilot's reach is constrained when monitors fail to detect an incident or when no handler exists for the alert type. Extending RCACopilot to incidents reported through customer tickets and to alert types without an existing handler are natural directions, and both can build on the same multi-source diagnostic-data collection infrastructure described in this chapter.

Stability of LLM-based predictions. LLMs can produce slightly different outputs across runs, even with the temperature set to zero. We mitigated this by running each experiment for three rounds and reporting the consistency across rounds (Section 3.6); results were stable across all rounds, but variance is real and worth being aware of when comparing LLM-based RCA systems. Reducing this variance through caching of deterministic-when-possible reasoning steps and through structured-output constraints is a promising next direction.

Chapter 4 A Multi-Agent System for Autonomous Site Reliability Engineering

Failure recovery is usually the last and most difficult stage of incident management because it requires state-changing actions that, if unsafe, can worsen failures. To address this challenge, this chapter presents STRATUS, an LLM-based agentic system that realizes autonomous SRE for cloud services through failure detection, localization, root-cause analysis, and mitigation. STRATUS differs from prior work on developing AI or agentic tools that assist human engineers [23], [24], [25], [26], [99], [108] (for data collection, summarization, and root-cause analysis). STRATUS aims to govern live production systems and mitigate failures, without human intervention.

We design STRATUS as a multi-agent system that orchestrates specialized agents of detection, diagnosis, mitigation, and undo, each responsible for one or multiple tasks (Figure 19). We chose a multi-agent design because (1) it specializes agents for SRE tasks with well-defined roles, (2) it offers customizability of individual agents and extensibility of the system, and (3) it makes it easy to reason about agent safety. STRATUS uses deterministic control-flow logic that orchestrates agents in a state machine, while leveraging LLMs in data flows that make agents intelligent and creative. We empirically compare this task-based design against an alternative role-based multi-agent design in Section 4.6.

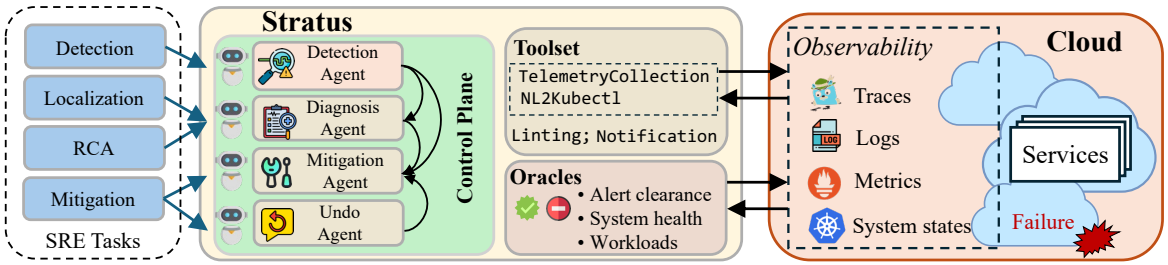


Figure 19: Overview of STRATUS, an LLM-based multi-agent system for autonomous Site Reliability Engineering (SRE) of modern cloud services.

Safety is a key challenge of SRE agents (or any agentic systems that operate critical live systems). STRATUS must not worsen the state of the target system when mitigating its failures; however, it is hard to ensure that a mitigation plan generated by the agent (or even by a human) is always successful, considering the dynamics and complexity of the system and imperfect AI models. In practice, the risk of "making things worse" is a fundamental barrier to deploying autonomous agentic techniques in high-stakes systems. We formalize a safety specification of agentic SRE, termed *Transactional No-Regression* (TNR). TNR constructs transaction semantics to ensure: (1) the agent's mitigation actions, if unsuccessful, can always be "undone"; and (2) the agent never drives the system to a state worse than the one in which the failure was detected (i.e., no regression). TNR is a safety property—it bounds how bad the system can get under the agent's actions—rather than a guarantee of eventual recovery.

We evaluate STRATUS on two SRE benchmark suites, AIOPSLAB [109] (presented in Chapter 5) and ITBench [15]. STRATUS significantly outperforms state-of-the-art agentic solutions in terms of success rate of failure mitigation problems by at least 1.5× across various models (GPT-4o, GPT-4o-mini, and Llama 3.3).

4.1 Background

STRATUS aims to realize *autonomous SRE*, which we define as an automated system for detecting failures, localizing failing components, analyzing the root causes, and mitigating the failures, with minimal human intervention. In other words, autonomous SRE systems like STRATUS imitate human engineers with agentic AI, and hence are fundamentally different from AI or agentic tools designed for assisting human engineers on specific SRE tasks [25], [26], [108]. A key differential feature of STRATUS is to automatically *mitigate* failures, beyond summarizing failure information and predicting root cause categories as in prior work. Mitigation poses unique challenges: unlike detection and diagnosis tasks that only need to observe the system, mitigation needs to

change the system state. Ensuring the safety of failure mitigation is a key challenge—mitigation actions must not cause new (nested) failures and must not drive the system to a worse state (i.e., no regression).

We model a target cloud system as an *environment* $\mathcal{E}$ characterized by:

- A set of states $S = \{s\}$, including a crash state $\perp$ where the system is completely unavailable;
- A severity metric $\mu(s^e) \colon S \cup \{\perp\} \to \mathbb{N} \cup \{\infty\}$ that represents the severity of an error state s^e.

An error state can be caused by different types of faults (root causes) [110], including software bugs [111], misconfigurations [112], and hardware issues [113], [114], [115], [116]. The severity of the error for a given state s is measured as a weighted sum of the sets of alerts (A), violations of service-level agreement (SLA) (V), and system capacity loss (unhealthy nodes) (L); it is formally defined as $\mu(s) = w_1 \cdot |A| + w_2 \cdot |V| + w_3 \cdot |L|$, where $w_i > \emptyset$ and $\mu(\perp) = \infty$ (the state $\perp$ in which the entire system is unavailable).

Just like human SRE engineers [8], STRATUS addresses the following SRE tasks in order to resolve the error in the system state:

- **Detection.** SRE must promptly detect production failures via logs, traces, and other telemetry data; detecting failures is the first step to prevent incidents.
- **Localization.** SRE must localize the faulty components so that they can be isolated to minimize the blast radius and impacts.
- **Root Cause Analysis (RCA).** SRE shall identify the root causes of the failures in order to repair the faulty components (e.g., fixing bugs and misconfigurations).
- **Mitigation.** SRE must mitigate the failures to prevent their propagation that leads to production incidents and service outages.

Note that RCA is not on the critical path of mitigation. In practice, RCA is typically done offline, while mitigation directly determines service availability [117]. A successful mitigation may not need to know the failure root causes. For example, mitigating a faulty machine can be done by migrating deployed jobs to a healthy machine [118], without knowing the faulty component; mitigating a faulty software component can be done by rebooting it [119], [120] or reverting recent changes [121].

4.2 STRATUS Design

STRATUS uses a multi-agent system design that orchestrates multiple specialized agents $\mathcal{A} = \{\alpha_i\}$ (α_i refers to an agent) to interact with $\mathcal{E}$ and mitigate any system failure represented by an error state s^e. Our implementation currently uses four agents $\mathcal{A} = \{\alpha_D, \alpha_G, \alpha_M, \alpha_U\}$:

- α_D (**Detection agent**): Observes $\mathcal{E}$ to identify failures and establishes the initial error state $s^e_\emptyset$.
- α_G (**Diagnosis agent**): Observes $\mathcal{E}$ ($s^e_\emptyset$ and telemetry) to determine the root cause(s) of the detected failure; its output is primarily analytical. α_G is responsible for both localization and RCA.
- α_M (**Mitigation agent**): Takes the diagnostic output from α_D and α_G. It is responsible for (1) devising a high-level mitigation plan, (2) decomposing the plan into a sequence of concrete mitigation actions, and (3) executing each action by issuing commands through agent tools with Agent-Computer Interfaces (ACI) [29] (see Section 4.3).
- α_U (**Undo agent**): Executes an undo sequence on $\mathcal{E}$ if a transaction executed by α_M is aborted.

Agents interact with the environment using actions from a defined *Action Space* through ACI:

- A_{read}: Read-only commands that do not change the state of $\mathcal{E}$ (e.g., `kubectl get`).
- A_{write}: Write commands that can mutate the state of $\mathcal{E}$ (e.g., `kubectl apply`).
- A_{undo}: A specialized sequence of commands executed by α_U to undo the effects of write commands.

Agents α_D and α_G are restricted to A_{read}. α_M executes actions from A_{write} as well as A_{read} for reading system states and checking pre/post conditions. α_U executes actions effectively in A_{undo} (which internally uses A_{write} commands to restore a prior state).
The system state of $\mathcal{E}$ changes based on *State-Transition Relations* [122], $R \colon S \times A_{\text{write}} \to S \cup \{\perp\}$. The undo is performed by an *operator* $U \colon S \to S$, realized by α_U.

4.2.1 Safety Specification: Transactional No-Regression (TNR)

STRATUS must guarantee safety. However, an agent's action—whether due to an imperfect understanding, a planning flaw, or hallucinations of the generative model—could inadvertently worsen an already degraded system state ($\mu(s^e_\emptyset) > \emptyset$). How can we grant autonomous agents the authority to execute sequences of potentially state-changing commands (e.g., via `kubectl`) for critical tasks like failure mitigation, without risking worse outcomes?
We formalize a safety specification of agentic SRE systems like STRATUS, termed Transactional No-Regression (TNR), atop the classic transaction abstraction [123]. TNR ensures that (1) the agent's mitigation actions, if unsuccessful, can always be "undone"; and (2) no externally visible state is ever worse, in terms of the severity metric μ, than the state observed when the failure was detected (no regression). TNR is a *safety* property: it caps the damage of an unsuccessful mitigation but does not by itself promise eventual recovery, which is a separate liveness goal.

Assumptions. We state the following assumptions, which are enforced by our implementation (Section 4.3):

A1. Writer Exclusivity (Agent-Lock or A-Lock). At most one writer agent (α_M or α_U) is scheduled to execute commands that can mutate system states at a time and has exclusive access; reader agents (α_D, α_G) do not mutate state relevant to μ. A-Lock is a readers-writer lock.

A2. Faithful Undo. The undo operator U, when invoked by α_U on s_{post} after an abort decision, restores the checkpointed s_{pre} exactly, i.e., $U(s_{\text{post}}) = s_{\text{pre}}$.

A3. Bounded Risk Window. The length k (the number of commands) of any transaction executed by α_M is bounded by a system-wide threshold K ($1 \leq k \leq K$). This limits the duration the A-Lock is held and the complexity of any single transaction.

Transaction semantics. We construct transaction semantics atop the above assumptions, where a transaction of length k ($1 \leq k \leq K$) is a sequence of read or write commands $T = (a_1, \ldots, a_k) \in (A_{\text{write}}, A_{\text{read}})^k$. A transaction encodes a mitigation plan executed by α_M while holding the A-Lock (per A1):

R1. Checkpoint. α_M records the entry state s_{pre} before a_1 is executed.

R2. Execute. α_M runs $a_1, \ldots, a_k$ sequentially. Let s_{post} be the state after a_k completes (or $\perp$ if any a_j ($1 \leq j \leq k$) causes a crash).

R3. Commit/Abort rule. α_M evaluates: *commit* if $s_{\text{post}} \neq \perp$ and $\mu(s_{\text{post}}) \leq \mu(s_{\text{pre}})$; otherwise, *abort* by instructing α_U to invoke U *once*. Per A2, this restores s_{pre} exactly. An aborted transaction leaves *no trace* on externally visible state transitions. We call the internal sequence of states visited during transaction execution, $s_{\text{pre}} \xrightarrow{a_1} s^{(1)} \xrightarrow{a_2} \ldots \xrightarrow{a_k} s_{\text{post}}$, and their corresponding severity metrics the *hidden μ-path*. Only $\mu(s_{\text{pre}})$ and, if T commits, $\mu(s_{\text{post}})$ are part of the externally *visible* state transitions that define the system's trajectory viewed by other agents.

The TNR property. Let $s^e_\emptyset$ be the error state observed by α_D when a failure occurs, with an *initial severity* $b = \mu(s^e_\emptyset)$. Then every state s in the externally visible state transitions (i.e., $s = s^e_\emptyset$, or s follows a read-only action by α_D or α_G, or s follows the completion—commit or abort—of a transaction by α_M or α_U) satisfies $\mu(s) \leq b$. The proof is by induction on the sequence of externally visible states; the inductive step uses the commit condition for committed transactions (giving $\mu(s_{\text{post}}) \leq b$) and the faithful-undo assumption A2 for aborted transactions (returning to s_{pre}, which already satisfies $\mu(s_{\text{pre}}) \leq b$).

TNR is an instance of the Alpern-Schneider safety property [124], [125]. By ensuring that the potential "damage" of an agent is capped (it cannot make the system observably worse than $s^e_\emptyset$), TNR provides a powerful tool for *safe exploration and iteration*: the agent can attempt ambitious, complex repairs, learn from outcomes (via observing μ post-transaction), and adapt its strategy without the risk of digging a deeper hole. Moreover, preventing agent-driven escalations beyond the baseline b makes the *liveness* property of eventually reaching a healthy state tenable [124], [126].

4.3 Implementation

STRATUS is implemented using the CrewAI multi-agent framework [27]. The agents are orchestrated via state-machine based control-flow logic (Figure 20). STRATUS is an extensible framework: the agents are loosely coupled and can be independently developed and customized. We currently use off-the-shelf LLMs such as GPT and Llama models

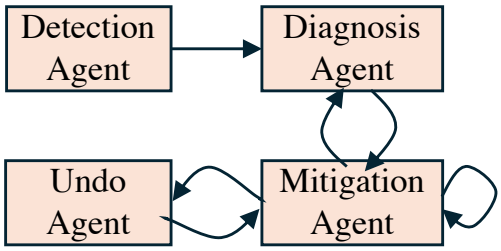


Figure 20: The state-machine–based control-flow logic.

without fine-tuning. One can also add new agents or replace existing agents. The control plane can be customized (e.g., using a different state machine) as it is decoupled from the data plane of the agents.

4.3.1 Realizing TNR in STRATUS

In STRATUS, the assumptions A1–A3 are enforced as safety invariants with the following mechanisms.

Writer Exclusivity. First, sandboxing-based confinement is implemented for each agent. The detection and diagnosis agents are not allowed to issue any commands that may mutate system states. Second, the state machine ensures mutual exclusion when a writer agent (the mitigation agent or the undo agent) is scheduled to run, i.e., the sequences of the actions of the writer agents are strictly serialized. We also instruct the agent to dry-run the commands (`kubectl --dry-run`) whenever applicable to simulate command execution and identify errors without altering the system state.

Confinement Rules. To ensure safety and predictability, each STRATUS agent operates under sandboxed confinement rules. As detailed in Table 13, detection and diagnosis agents are restricted to a Read-Only role and are only permitted to *observe* the system using commands such as `TelemetryCollection` tools. These agents are explicitly prohibited from executing any operation that alters the state of the system. To perform mitigation, STRATUS will need to use the Writer role, as changing the system state is essential for mitigation. For state-changing tasks, STRATUS performs command-level validation to verify that generated actions are syntactically correct and safe to run in production. Risky operations such as deleting namespaces are prohibited. Certain `kubectl` commands, such as `kubectl exec -it` and `kubectl edit`, launch interactive shells or editors. These commands are prevented because STRATUS cannot participate in interactive sessions, which would cause execution to hang. Additionally, granting

arbitrary shell access significantly increases the potential for unintended and unsafe behaviors. Therefore, we explicitly disallow all interactive commands.

To further enforce safety, STRATUS uses static linting to validate command syntax before execution. This validation includes rejecting shell pipes, compound commands (e.g., using `&&`, `||`, or semicolons), and other complex shell constructs. These are disallowed for two reasons: (1) there are exponentially many arbitrary shell expressions that STRATUS may generate; comprehensively verifying the safety of all possible shell expressions is not realistic. On the contrary, individual `kubectl` invocations can be easily checked. (2) We require the agent to operate incrementally, issuing and validating one command at a time, to work effectively with our safety infrastructure and confinement layers.

Faithful Undo. STRATUS ensures that every agent action has a corresponding undo operator; otherwise, the action is not allowed. STRATUS rejects destructive actions which cannot be recovered, or turns them into recoverable actions through the agent tools (e.g., a file deletion is replaced by moving the file to a backup volume, which is recoverable). A common pattern of the undo operator is system-state rollback. Modern cloud platforms such as Kubernetes [127], Borg [128], Twine [129], and ECS [130] are implemented based on *the state-reconciliation principle* [83], [131], with declarative interfaces [122], [132]. Rollback in state-reconciliation systems can be implemented by recording the state changes and later reconciling to the recorded states.

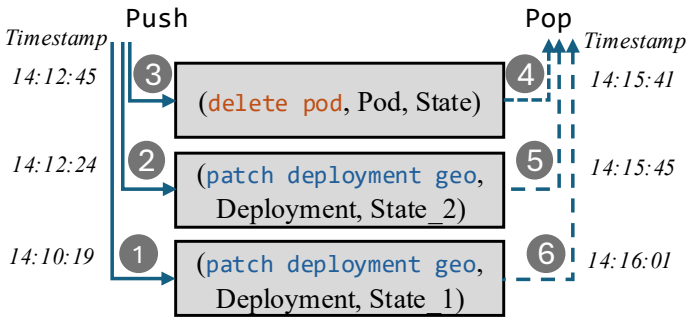


Figure 21: An example of the action stack used for reconciliation-based undo.

Table 13: Agent confinement rules in STRATUS. The role-based rules specify the actions allowed for each agent role; the command-level rules are blocked for all agents.

Category	Subcategory	Confinement	Example(s)
Agent role-based rules (allowed actions):			
Agent Role-Based	Read-Only Agents	Non-mutating commands only	`kubectl get`, `kubectl describe`, `kubectl logs`, `ls`, `cat`, `TelemetryCollection`
	Writer Agents	Sequential execution only	Only serialized state modifications are allowed
Command-level rules (blocked for all agents):			
Kubectl Subcommands	Destructive Operations	Namespace deletion	`kubectl delete namespace my-ns`
	Interactive Edit	`debug`, `edit`	`kubectl debug pod/foo`, `kubectl edit deployment/bar`
	Input from stdin	`-f -` pattern	`kubectl apply -f -`
Kubectl Flags	Interactive Terminal	`--stdin`, `--tty`, `exec -it`	`kubectl attach --tty pod/foo`, `kubectl exec -it pod/bash`
Shell Syntax	Pipelines	Pipe operator	`kubectl get pods | grep Running`
	Compound Commands	`&&`, `||`, `;`	`kubectl get pods && echo success`
	Command Substitution	`$(...)`, backticks	`kubectl delete pod $(kubectl get pods -o name)`
	Flow Control	`if`, `for`, `while`, `until`, `case`	`if [ ... ]; then ...; fi`
	Shell Functions	Function definitions	`myfunc() { ... }`

STRATUS implements a stack-based rollback mechanism, exemplified by Figure 21. The stack-based mechanism allows fine-grained rollback of related system resources (represented as state objects in Kubernetes) instead of system-wide state. Our current undo agent α_U mechanically walks the stack and performs undo, and is arguably not intelligent. The rationale for keeping α_U as an independent agent is to integrate more advanced undo policies and mechanisms (e.g., a learning-based rollback policy where the LLMs are given the choice of undoing a subset of actions in the stack instead of always rolling back to the start state).

Bounded Risk Window. The implementation is trivial. However, the window size has an impact on mitigation effectiveness. We set K to 20 based on an empirical analysis (Figure 22): we ran the evaluated agents on AIOPSLAB problems (detection, localization,

root-cause analysis, and mitigation) with total step limits set to different values (3, 5, 10, 15, 20, and 30). STRATUS (GPT-4o) improves from 0% to 53.53% after switching the step limit from 3 to 10. This shows that allowing STRATUS to take more steps can significantly boost its effectiveness. We also find that among all evaluated agents, agent performance plateaus after 15 steps, indicating that taking more steps after a certain threshold does not yield significantly better performance. The effectiveness gain can be observed on STRATUS with other LLM backends, but with lower overall accuracy, relatively scaling with the LLM's capability. Based on these observations, we set the default bounded risk window to $K = 20$.

Limitations. Realizing perfect undo for all conceivable state changes and capturing every destructive action with rule-based confinement remain practical challenges; we discuss both in Section 4.6.

4.3.2 Other Notable Implementation Details

Agent tools. We develop tools that enable agents to interact with the environment using natural language, following the Agent-Computer Interface (ACI) principles [29].

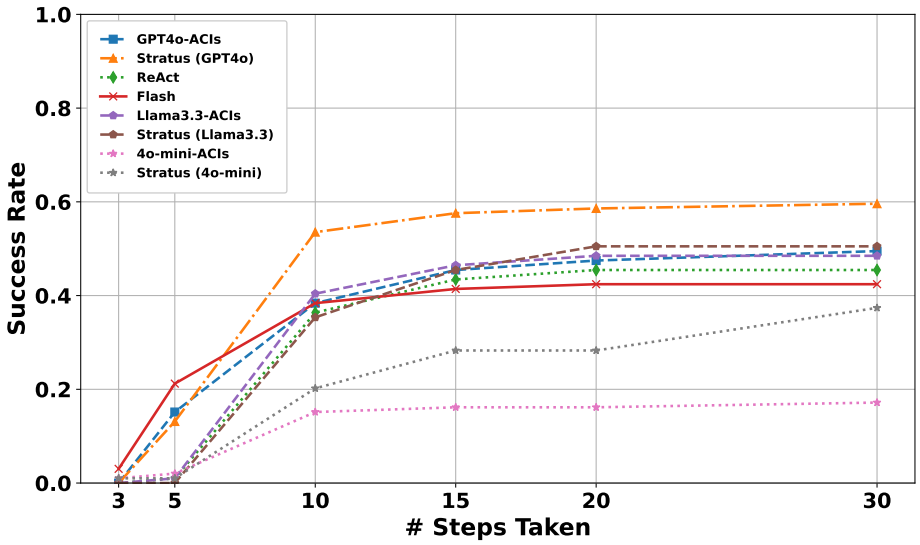


Figure 22: Impact of the step limit on the success rate of different agents in AIOPSLAB.

Specifically, we developed two sets of tools: (1) *observability tools* for agents to query different kinds of observability data such as state objects (maintained by the cloud platform like Kubernetes), system logs, distributed traces, and performance metrics; we use LLMs to summarize the observability data instead of directly feeding a large volume of data to the agents, as suggested by recent work [108]. (2) *command-line tools* that allow agents to execute commands and change system states using natural language, e.g., `NL2Kubectl` takes descriptions in natural language and generates concrete `kubectl` commands. The confinement described above is implemented as the command-line tool wrappers. STRATUS's design supports hybrid operations where writer agents can request human approval before execution, with interceptable action stacks and tool interfaces.

Bootstrapping (where shall the diagnosis agent start?). The multi-agent design makes it easy to specialize agents with different roles. We boost the diagnosis agent α_G with a bootstrapping technique that helps α_G localize the failure region. In cloud systems, failures are propagated along request-response paths [117], [133], which is reflected by distributed traces [134], [135]. STRATUS localizes the faults by constructing the call graph and establishes an initial fault localization hypothesis to guide the diagnostic process. Despite the simple idea, this fault-localization-based bootstrapping is important for large-scale systems and massive volumes of observability data.

Termination (when shall the agents stop?). Agents need to correctly determine whether the target failure is successfully resolved before stopping their actions; a perpetual agent could cause unintended effects, destabilize the system, or waste computing resources. STRATUS integrates a structured validation-and-termination approach. After each diagnosis or mitigation procedure (a series of actions), STRATUS assesses system health based on three kinds of oracles: (1) *alerts*: whether the alert that reports the target failure is cleared; (2) *user requests*: whether user requests can be successfully returned (e.g., no `5XX` or `4XX` HTTP responses); and (3) *system health*: whether system components (e.g., pods and volumes) are running in healthy states.

These oracles act as weak oracles individually, each capturing a partial view of system health. STRATUS combines multiple oracles to form a stronger oracle. If all oracles pass, α_M concludes a successful mitigation and terminates.

4.4 Evaluation

We evaluate STRATUS on two state-of-the-art benchmarks, AIOPSLAB [109] and ITBench [15]. Both benchmarks provide a live, arena-like environment where AI agents are asked to resolve problems; each problem encodes a failure in emulated cloud systems [136], [137], [138], as exemplified by Figure 23. We focus on STRATUS's ability to mitigate failures (which has dependencies on detection and localization); we also report results for the other tasks, including detection, localization, and RCA. Each benchmark takes STRATUS 15–20 machine hours to finish.

Baselines and metrics. We compare STRATUS with the reference agents offered by AIOPSLAB and ITBench, referred to as AOL-agent and ITB-agent respectively. For AIOPSLAB, we also compare STRATUS with two other agents, ReAct and Flash, included in the benchmark [109]. We fuel these agents with different LLMs, including GPT-4o,

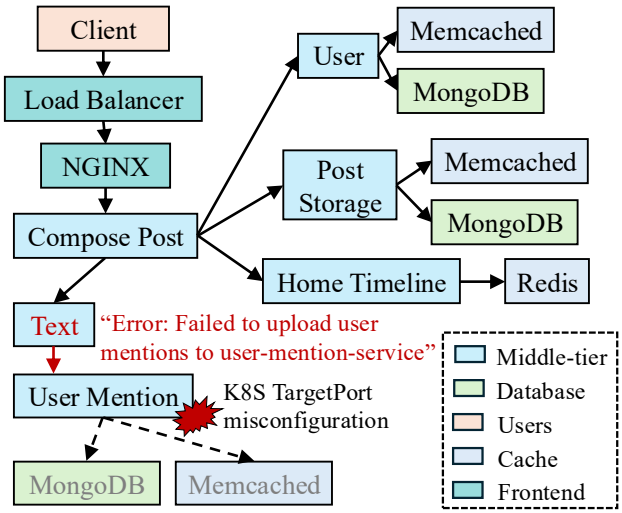


Figure 23: An example mitigation problem from the AIOpsLab benchmark.

GPT-4o-mini, and Llama 3.3. The LLM temperature is set to 0 to make the results more deterministic. We measure agentic systems with metrics: success rate (the percentage of problems being successfully solved), average time (in seconds), the number of steps to solve a problem, and monetary cost in US dollars (calculated based on token consumption).

Failure mitigation effectiveness. STRATUS (GPT-4o) significantly outperforms state-of-the-art SRE agents on mitigation problems in both AIOPSLAB and ITBench (Table 14). Concretely, it successfully solves 69.2% (9/13) and 50.0% (9/18) mitigation problems in AIOPSLAB and ITBench, respectively. This leads to improvements on the success rate by 1.5× and 5.4× over the second-best performing solution (AOL-agent and ITB-agent), respectively. The advantage of STRATUS over the other SRE agents is consistent with different models, including both GPT-4o-mini and Llama 3.3.

We carefully inspect the problem-solving trajectories of the evaluated agents. For the mitigation problems solved by STRATUS in AIOPSLAB, STRATUS correctly mitigated the triggering conditions or the root causes of the failures. In eight of the 18 ITBench problems, STRATUS exploits an observation that the injected faults do not persist after the failing pods restarted (the fault injector cannot recognize the original pod); thus, STRATUS tends to restart the failing pods one by one and thus solves the problem. Such a mitigation strategy may not work in reality, especially for failures caused by persistent faults (e.g., misconfigurations and hardware defects).

STRATUS overall takes longer to solve problems; for the hard problems, STRATUS retries multiple times to solve them—*the retry is enabled by the Transactional No-Regression safety guarantee*. In comparison, the other agents do not have the safe retry capability. In fact, they provide no safety guarantee over their mitigation actions: for the failures they fail to mitigate, these agents can leave the system in worse states than the original

error state. For the same reason, STRATUS spends more in dollar amount, but the amount is significantly cheaper than human cost.

Effectiveness of TNR-based undo-and-retry. The TNR safety property is crucial to STRATUS's effectiveness in solving complex mitigation problems. Table 15 shows this through an ablation study of STRATUS (GPT-4o). We compare STRATUS (GPT-4o) with two variants: (1) *no retry:* only attempting one mitigation path; and (2) *naïve retry without undo:* exploring a new mitigation path if the previous attempt was not successful, but conducted directly from the ending state of the last attempt instead of rolling back to the original state.

We see that (1) the ability to retry a different mitigation plan is important—only in two problems does the first mitigation plan generated by STRATUS successfully mitigate the failure; and (2) the ability to undo and roll back the system state is critical—new mitigation from an error state left by a failed attempt can hardly succeed (only one new problem is solved in this way). Intuitively, failed mitigations further worsen the system state, making the system harder and harder to save. TNR prevents such failure patterns and thus is essential to mitigation. Figure 24 shows the probability density of the number

Table 14: Effectiveness of STRATUS in solving mitigation problems of AIOPSLAB and ITBench. "4o", "mini", and "llama" refer to GPT-4o, GPT-4o-mini, and Llama 3.3, respectively. ITB-agent numbers are taken from the ITBench paper [15] ("–": not reported). The "$" column reports the average cost (in USD) of running the agent over all the mitigation tasks in each benchmark.

(a) AIOPSLAB (13 mitigation problems)

Agent	Succ.	Time (s)	Steps	$
ReAct (4o)	23.1%	46.0	23.0	0.112
Flash (4o)	38.5%	154.0	23.1	0.150
AOL-agent (4o)	46.2%	223.3	21.7	0.206
AOL-agent (mini)	7.7%	58.9	22.7	0.003
AOL-agent (llama)	15.4%	98.2	13.0	0.037
STRATUS (4o)	**69.2%**	811.9	46.3	0.877
STRATUS (mini)	23.1%	3557.9	125.7	0.036
STRATUS (llama)	23.1%	1486.9	71.8	0.360

(b) ITBench (18 mitigation problems)

Agent	Succ.	Time (s)	Steps	$
ITB-agent (4o)	9.2%	251.7	–	–
ITB-agent (llama)	5.7%	440.8	–	–
STRATUS (4o)	**50.0%**	1720.8	115.7	6.11
STRATUS (mini)	19.4%	3874.9	468.9	9.38
STRATUS (llama)	28.0%	2566.6	160.3	0.76

of retries per problem done by STRATUS (GPT-4o) in AIOPSLAB mitigation problems. STRATUS sets a limit of nine retries. In 80%+ of the problems, STRATUS retries at least once; in 30%+ of the problems, STRATUS retries no fewer than five times.

We further examine STRATUS's behavior under ablation on the ITBench benchmark, as shown in Table 16. We observe that (1) disabling retry substantially drops the success rate to 11.1%. Note that this configuration implicitly disables the undo agent, as there are no prior actions to undo. Without retries, the agent must determine the root cause and devise a mitigation plan correctly with one attempt, which is significantly challenging. (2) STRATUS achieves the same task-level success rate (solving 9/18 problems) when the undo agent is disabled. As discussed above, STRATUS exploits the observation that in 8 out of 18 problems, restarting the target pods clears the incident alerts. Since ITBench evaluates correctness primarily based on the absence of immediate alerts, this allows the agent to resolve the incident without trying more destructive, state-changing actions, where our undo mechanism will be most effective.

To better understand this, we analyzed these eight problems in detail. Among all mitigation actions issued by the agent, approximately 15.2% are pod deletion operations managed by Kubernetes Deployment, which can automatically recreate the pods. Furthermore, a significant portion (44.2%) of state-changing commands, e.g., changing a container image to an image with a hallucinated name, are easily detected and fixed by STRATUS in the subsequent run, since STRATUS has a reflection mechanism, which can help the agent to realize what went wrong in the previous run. The rest of the commands are a mix of direct restart or redeployment of the pods, which again lifts the underlying

Table 15: Ablation analysis on TNR-enabled undo-and-retry of STRATUS (GPT-4o) using the 13 mitigation problems in AIOPSLAB, which shows the importance of TNR safety.

Configuration	**Succ. Rate**	**Time (s)**	**Cost ($)**
STRATUS (4o)	69.2%	811.9	0.877
− No retry	15.4%	72.6	0.163
− Naïve retry w/o undo	23.1%	1221.5	0.929

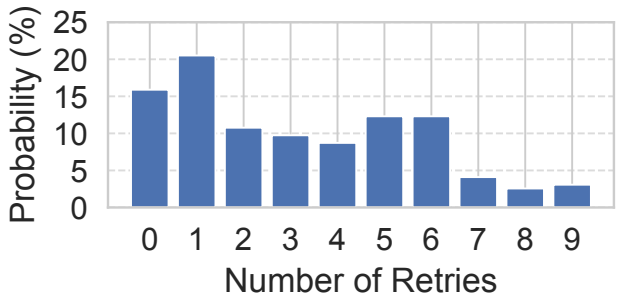


Figure 24: Probability density of the retry counts per problem.

fault directly. In this regard, the undo mechanism does not improve the success rate in ITBench. However, the absence of undo capabilities can still compromise the cloud over time. Latent or non-critical side effects from agent actions, such as misconfigurations or performance degradations that do not immediately trigger alerts, may go undetected. Without TNR, there is no guarantee ensuring that such side effects are identified and reverted.

Effectiveness on detection, localization, and RCA. We also evaluated the effectiveness of STRATUS on other SRE problems in comparison to other SRE agents (Table 17). STRATUS outperforms the reference agents on both detection and localization problems. Specifically, STRATUS with Llama and GPT-4o achieves over 90% success rate for detection. Localization and RCA problems, which require the agents to pinpoint the root-cause components (e.g., the pod) and the fault type (e.g., misconfiguration or bug), present greater challenges to agentic solutions; STRATUS (GPT-4o) has the highest

Table 16: Ablation analysis of STRATUS (GPT-4o) on the 18 mitigation problems in ITBench.

Configuration	Succ. Rate	Time (s)	Cost ($)
STRATUS (4o)	50.0%	1720.8	6.11
– No retry	11.1%	267.2	0.72
– Naïve retry w/o undo	50.0%	1711.9	5.13

success rate for localization problems and is second highest for RCA problems. Note that RCA is not a prerequisite of failure mitigation (see Section 4.1).

The relatively low RCA success rates (STRATUS included) deserve particular interpretation, because much of the gap is attributable to ambiguous evaluation metrics rather than to agent shortcomings. AIOPSLAB measures RCA success by string-matching the agent's output against pre-defined category labels, but some labels are ambiguous and not mutually exclusive. For example, on a port-misconfiguration problem, AIOPSLAB labels the root cause as "Virtualization" (layer) and "Misconfiguration" (fault type). STRATUS answers the layer as "Application" and the fault type as "Network/Storage Issue", which is arguably more precise (a port is more naturally an application-level concern than a virtualization concern, since AIOPSLAB has no traditional hypervisor-like virtualization layer). The strict label match nevertheless counts STRATUS's answer as incorrect. We are actively investigating richer RCA metrics for agentic SRE evaluation as a follow-on to this work.

Memory footprint. To understand the memory cost of running STRATUS, we profile a heavy mitigation problem (`misconfig_app_hotel_res-mitigation-1` in AIOPSLAB) that takes 13 steps to resolve. The dominant memory cost, over 530 MB, comes from loading the agent runtime and its dependencies (CrewAI and the benchmark tooling). Among

Table 17: Effectiveness of STRATUS in solving the detection, localization, and RCA problems in AIOPSLAB (the number of problems of each type is given in parentheses).

	Detection (32)			**Localization (28)**			**RCA (26)**		
Agent	**Succ.**	**Time (s)**	**$**	**Succ.**	**Time (s)**	**$**	**Succ.**	**Time (s)**	**$**
ReAct (4o)	87.5%	33.2	0.086	26.8%	59.6	0.328	23.1%	28.5	0.065
Flash (4o)	59.4%	30.7	0.013	39.3%	165.0	0.190	26.9%	30.5	0.019
AOL-agent (4o)	62.5%	14.4	0.061	46.9%	34.8	0.083	**38.5%**	12.3	0.061
AOL-agent (mini)	25.0%	43.0	0.002	9.5%	34.1	0.001	7.7%	57.7	0.003
AOL-agent (llama)	84.4%	19.8	0.019	32.1%	40.5	0.018	30.8%	13.8	0.014
STRATUS (4o)	90.6%	48.4	0.118	**51.2%**	65.3	0.126	34.6%	39.6	0.068
STRATUS (mini)	78.1%	34.4	0.010	25.0%	37.2	0.013	30.8%	279.0	0.007
STRATUS (llama)	**93.8%**	50.0	0.111	36.3%	90.5	0.112	26.9%	60.2	0.095

the four agents, the Detection agent uses the most memory because it collects and analyzes raw telemetry data (e.g., using `pandas` for trace analysis); the Mitigation agent consumes the largest *context* memory at 31.03 KB because it accumulates trajectories across steps; the Undo agent consumes the least at 5.79 KB because its rollback operations are simple. The numbers indicate that STRATUS fits comfortably within the memory budget of a single production-grade VM.

4.5 Related Work

AI and agentic research for SRE. Applying AI/ML techniques to solve SRE problems has a long history of research, as AI/ML has the potential to accommodate the large scale of modern systems and infrastructures (e.g., the clouds) [137], [139], as well as massive volumes of observability data [135]. Specialized AI/ML techniques are designed for concrete SRE tasks, such as failure detection [140], [141], [142], triage [143], [144], and diagnosis [145], [146], [147], [148]. Recently, generative AI models, as well as agentic approaches, are increasingly used to further generalize and automate these SRE tasks [24], [102], [108], [149]. However, the aforementioned efforts focus on building tools to assist human engineering (see Section 4.1). Differently, we aim for autonomous SRE. A differential feature is *failure mitigation*. Prior work mostly focuses on helping human engineers to detect and understand the failure or recommend potential mitigation documents, whereas mitigation is the main goal of STRATUS.

Safety of agentic AI systems. Safety is an emerging concern of agentic AI systems [30], [150], [151], [152]. Prior work focuses on implementing safety guardrails [153], [154], [155], [156], which aim to prevent agents from producing or acting upon harmful inputs and outputs, often through prompt-level or static analysis. However, such preventative approaches are limited in environments like cloud systems, where the system state evolves over time and side effects emerge dynamically only after

actions are executed. For example, AutoSafeCoder [153] mitigates code vulnerabilities via static checks, but such methods cannot anticipate runtime failures or cascading effects that depend on temporal system dynamics. In addition, few works provide guarantees on recovery. For example, TrustAgent [30] proposes the Agent-Constitution framework that addresses agent safety. However, it does not provide guarantees on the safety of the agent action or any recovery from unsafe actions. For SRE, this could allow failures to regress into a worse situation due to unsafe agent actions.
STRATUS addresses this challenge by enabling safe exploration with Transactional No-Regression (TNR). TNR allows agents to explore multiple action sequences safely and iteratively, giving the agent more opportunities to mitigate failures. Concurrent work has also started integrating transactional primitives into LLM-based frameworks [157]. STRATUS extends this direction by enabling rollback of both the agent and the system state, combined with a continuous severity metric to assess action outcomes. This dynamic and fine-grained safety mechanism is critical for real-world deployment of autonomous agents in evolving operational contexts like SRE.

Multi-agent systems. Multi-agent has become a common design pattern, with the development of many LLM-based multi-agent frameworks [27], [28], [158], [159], [160] and use cases [161], [162], [163], [164]. Recent work proposes multi-agent conversations [28] and debates [165], [166] that encourage diverse thinking among agents. However, our experience shows that conversations and debates are not suitable for SRE tasks which require safety reasoning and timeliness of problem solving. Hence, STRATUS uses a deterministic state machine to coordinate the agents. Recent work also reports common failure modes of multi-agent systems [167], [168], which STRATUS's deterministic orchestration and role specialization are designed to avoid.

4.6 Summary

This chapter presented STRATUS, an LLM-based multi-agent system for autonomous Site Reliability Engineering. STRATUS orchestrates four specialized agents—detection, diagnosis, mitigation, and undo—through a deterministic state machine and exposes them to the cloud through Agent-Computer Interfaces. At its core is a formal safety specification we call *Transactional No-Regression* (TNR), implemented atop a stack-based action-rollback mechanism that takes advantage of the state-reconciliation principle of modern cloud platforms. Across the AIOPSLAB and ITBench benchmarks, STRATUS significantly outperforms state-of-the-art agentic baselines on mitigation problems (by at least 1.5×) across multiple LLM backends, and the ablation study shows that this advantage is driven by TNR's safe undo-and-retry capability rather than raw model power. We close this chapter by discussing the practical limitations of STRATUS and the directions in which it can be extended.

Design choice of STRATUS's multi-agent architecture. Besides our task-based multi-agent design, we also explored a role-based approach using AutoGen [28] with conversable agents playing the roles of Application Developer, Platform Engineer, System Administrator, and Team Manager. However, this proved ineffective and inefficient. On 48 sampled AIOPSLAB problems, this design solved only 10 problems. The agents over-communicated, each with only a partial view of the system stack. With round-robin speaker selection, every agent spoke at each step, slowing decision-making and causing redundant telemetry collection as agents repeatedly verified others' observations. For example, solving the simple detection problem `user_unregistered_mongodb-detection` took 893 seconds—25× longer than STRATUS.

Perfect undo is hard in practice. Realizing perfect undo for every conceivable state change in a complex cloud environment remains a practical challenge. Some operations

involve application-specific state or external interactions whose effects cannot be captured by a simple $U(s_{\text{post}}) = s_{\text{pre}}$ operator. Fortunately, modern cloud-native systems are typically equipped with operator programs that follow the state-reconciliation principle [169], [170], which STRATUS can leverage for the bulk of common rollbacks. For the remainder, more sophisticated compensation logic (e.g., saga-style compensating transactions) is a natural future-work direction.

Confinement is rule-based and imperfect. STRATUS's writer-exclusivity guarantees rest in part on a rule-based filter that prevents reader agents from issuing state-changing commands. Rules can miss destructive actions encoded in unexpected ways (e.g., a clever shell expansion that bypasses a regex check). Recent work on agent guardrails [153], [154], [155], [156] provides increasingly powerful runtime checks; integrating these techniques with STRATUS's confinement layer is a natural way to strengthen its safety story.

Single-writer concurrency. TNR currently assumes there is only one writer agent active at a time; multiple writers are strictly serialized through the readers-writer A-Lock. Our deployment model uses one STRATUS instance per cloud system (each system in its own namespace), which makes serialization acceptable for typical production loads. STRATUS can be extended to support multiple concurrent writers through resource-aware concurrency control: a coordination controller acts as a distributed lock manager, admitting a new mitigation plan P_i only if its resource set $R(P_i)$ is disjoint from the currently locked set R_{locked}. Resources are added to and removed from R_{locked} as plans begin and end. Each admitted transaction still executes under TNR semantics, so the safety guarantee holds per-transaction, and the disjointness condition prevents read-after-write hazards across concurrent writers. We leave a full implementation and evaluation of this extension to future work.

Handling concurrent inter-dependent faults. Beyond problems from AIOpsLab and ITBench, we also evaluated Stratus on a more challenging scenario: *concurrent inter-dependent faults*. We created a new ITBench problem with a three-node quorum system (one leader, two followers) where we injected data corruptions to the followers' volumes and a crashing bug in the leader. Correct mitigation requires fixing the follower volumes first, then restarting the leader to re-establish quorum. Unfortunately, Stratus failed to localize the follower data volumes as root causes and only restarted the leader, which could not form quorum without healthy followers. However, TNR still applied—Stratus safely retried multiple times without degrading the system further. There are no multi-fault problems in the two benchmarks. To solve them, we envision enhancing Stratus to commit a transaction whenever the severity $\mu(s)$ decreases below the initial baseline b, then start a new transaction. This ensures decreasing severity between transactions, preserving progress toward eventual incident resolution without violating TNR semantics.

Beyond capability-only evaluation. The agentic-AI community has produced an explosion of capability benchmarks but very few that take *safety* as a first-class metric. We argue that future SRE benchmarks should evaluate not only whether an agent solves a problem but also whether it does so without ever driving the system into a state worse than the baseline. AIOpsLab augmented with TNR-style oracles is a step in that direction; we expect future work to refine the safety metrics, particularly for multi-fault and multi-tenant scenarios in which the notion of "the baseline" itself becomes more nuanced.

The Stratus artifacts are publicly available at `https://github.com/xlab-uiuc/stratus`.

Chapter 5 Evaluating AI Agents for Autonomous Cloud Operations

AI's rapid progress owes much to benchmarks, and there are already many. However, evaluating AI, especially agentic AI, for real-world system operations remains hard: most site reliability engineering (SRE) benchmarks use static, pre-collected data and cannot capture interactive, evolving behavior of cloud environments and the agents. SRE agents must reason and act as the system changes across a large action space, where failures can occur at any layer, and valid mitigations may span many controls. This raises a big-picture question: how effective and safe are such agents in live clouds?

As LLMs and autonomous agents are increasingly integrated into cloud operations, the lack of a standard, task-oriented evaluation framework becomes a significant barrier to progress. To bridge this gap, we propose AIOpsLab, a comprehensive framework for benchmarking AI agents in end-to-end cloud incident scenarios. AIOpsLab enables rigorous, reproducible, and extensible evaluation by simulating diverse cloud services, injecting realistic faults, and providing a standardized interface for agent interaction and measurement across the incident lifecycle.

In this chapter, we mainly discuss the design and implementation of AIOpsLab and its major component (the Orchestrator), as illustrated in Figure 25.

5.1 Background

5.1.1 Limitations of Existing AIOps Benchmarks

Existing AIOps benchmarks suffer from several limitations that make it difficult to evaluate LLM-based agents in a meaningful way. Most are *static*: they ship pre-collected datasets such as time-series Key Performance Indicators (KPIs) and system metrics [171], [172], or fixed question-answer formats [173] that look nothing like the dynamic, evolving conditions an agent would face in a live cloud. Static benchmarks cannot replicate the unpredictable interaction between incident, workload, and operator action that defines real production environments, and as a result, an agent that scores well on a static benchmark may still fail catastrophically the first time it is deployed.

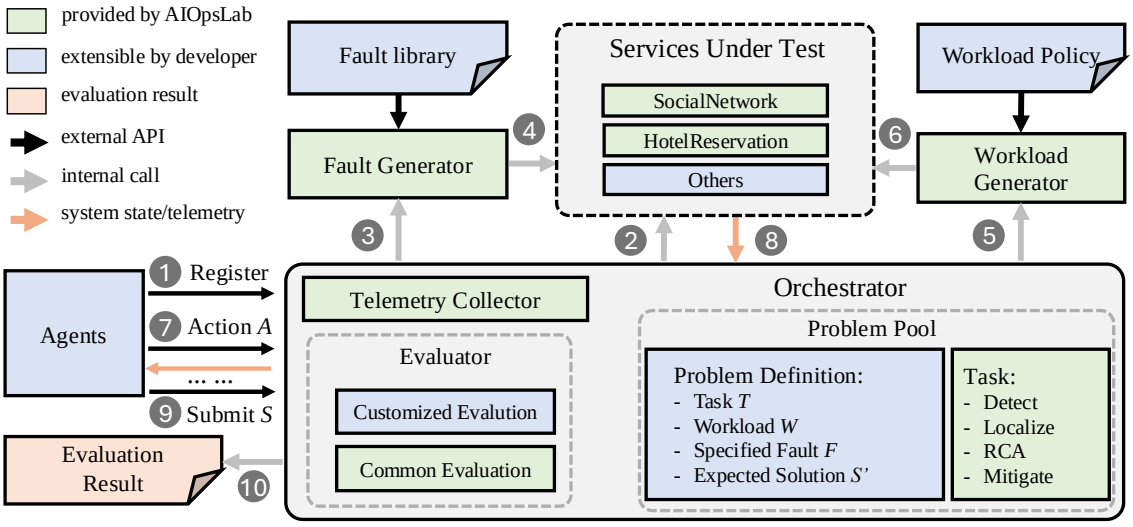


Figure 25: **Overview of AIOpsLab.** The Orchestrator coordinates interactions between various system components and serves as the Agent-Cloud Interface (ACI). Agents engage with the Orchestrator to solve tasks, receiving a problem description, instructions, and relevant APIs. The Orchestrator generates diverse problems using the Workload and Fault Generators, injecting these into applications it can deploy. The deployed service has observability, providing telemetry such as metrics, traces, and logs. 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.

A second limitation is that existing benchmarks tend to lack *clear task definitions*, particularly for the faults they introduce. A long line of fault-injection tools for distributed and cloud systems [16], [17], [18], [19], [20], [21], [73], [75], [76], [77] exists, and chaos-engineering frameworks such as ChaosBlade [174] and ChaosMesh [175] have made fault injection routine in production. However, the benchmarks built on top of these tools focus mostly on surface symptoms (pod failures, high CPU utilization, network latency) rather than on the underlying fine-grained root causes such as misconfigurations, code defects, and authorization issues [19], [20], [82], [83], [122]. Coarse-grained symptom injection cannot evaluate an agent's true ability to diagnose and mitigate a real incident, because the agent never has to reason about why the symptom appeared in the first place.

To evaluate AIOps agents fairly, the failure scenarios in a benchmark must go beyond performance and crash failures and instead reflect the realistic, multi-layered cases that production engineers actually face.

5.1.2 Goals of AIOpsLab

Motivated by these gaps, we set three goals for AIOpsLab: (1) *realistic, interactive evaluation*, in which agents reason and act over a live, evolving cloud environment rather than over a frozen dataset; (2) *full-lifecycle coverage*, where a single benchmark can evaluate detection, localization, root-cause analysis, and mitigation, so that progress on one task does not come at the cost of another; and (3) *fine-grained, root-cause-level faults*, so the benchmark exercises an agent's ability to diagnose underlying causes rather than mere symptoms. The remaining sections describe how the design of AIOpsLab realizes these goals.

5.2 Design Principles for the Agent-Cloud Interface

A key objective of AIOpsLab is to realize an effective **Agent-Cloud Interface (ACI)** between the agents under evaluation and the cloud environment. Inspired by the human/agent-computer interaction principles articulated in recent agent benchmarks [29], [176] and by industry practitioner guidance [14], the ACI should make cloud operations easy for an agent: it should let the agent communicate, take action, and receive feedback through a clear, predictable surface. Three design principles shape the ACI.

Unified interaction with cloud services. The ACI exposes a single, consistent interface for interacting with the cloud, regardless of which agent or even human operator is on the other end. Through clearly defined APIs, it surfaces the same key actions to every client: retrieving logs, gathering metrics, fetching traces, and executing commands. This standardization frees agent developers from having to learn the idiosyncrasies of each underlying service.

Real-time, actionable feedback. Every action in the ACI returns meaningful, actionable feedback to the agent, whether the action succeeded or not: error messages, system responses, and output logs are all surfaced verbatim. This feedback loop lets an agent adjust its next action on the fly, which is essential for the multi-step planning that real incident handling demands.

Simplified action space. A live cloud has a vast number of services and possible operations, and an agent that has to enumerate them is one that quickly gets lost. The ACI deliberately reduces the set of available actions to a clear, well-documented list of valid API calls (e.g., a curated subset of Kubernetes and telemetry-scraping commands). The smaller action space makes the cloud more approachable for agents while still being expressive enough to drive complex incident-resolution workflows.

5.3 Problem Definition

To support a wide range of evaluation scenarios (referred to as *problems*), which replicate realistic incidents within the cloud system, we first formalize an AIOps problem P as a tuple: $P = \langle T, C, S \rangle$, where T represents a *task*, C represents a *context*, and S represents the expected *solution* (oracle).

The task T defines the specific AIOps operation to be performed, categorized into four types: detection, localization, root cause analysis (RCA), and mitigation. We define these tasks in Table 18. Each task type is associated with success criteria and evaluation metrics. For instance, the detection task employs Time-to-Detect (TTD) to measure the time taken to detect a fault.

Level 1: Detection. The foundational requirement of any AIOps approach is detecting that something is wrong. Level-1 problems focus on the preliminary identification of unusual behavior in the system, e.g., a sudden spike in CPU usage or an unexpected drop in network throughput. Effective detection is the first line of defense, surfacing potential incidents before they escalate.

Level 2: Localization. Once an anomaly is detected, the next step is localization: identifying which component (e.g., a specific microservice) is responsible. A Level-2 problem is solved correctly when the agent points to the faulty service rather than attributing the issue to the

Table 18: Task taxonomy for AIOps agent evaluation. The lower the level, the easier the task. AIOpsLab aims to evaluate agents across all task levels with its problems.

Level	Task (# sub tasks)	Evaluation Focus
1	Detection (1)	Can the approach accurately detect anomalies or deviations?
2	Localization (1)	Can the approach pinpoint a fault's exact source (e.g., microservice)?
3	Root Cause Analysis (RCA) (2)	Can the approach determine the underlying cause of the fault?
4	Mitigation (1)	Can the approach give effective solutions to recover the environment?

system as a whole. Effective localization is critical for prompt and precise intervention and minimizes the time spent on unfocused diagnosis.

Level 3: Root Cause Analysis. Beyond surface symptoms, Level-3 problems evaluate whether an agent can determine *why* a service has failed. For example, when a microservice crashes due to an unhandled exception, the agent should identify the fault type (a code defect) and not stop at the symptom. Level-3 problems can include sub-tasks: an RCA problem typically asks both for the system level at which the fault occurs and for the fault type itself.

Level 4: Mitigation. The hardest level evaluates whether the agent can take corrective action that actually resolves the incident. The metric here is not just whether the agent stops the symptoms but whether the action it takes is specific, well-scoped, and side-effect-free. For instance, if the root cause is a misconfigured microservice, the agent should adjust that service's configuration rather than rebooting the cluster, and it must avoid both regressions in healthy components and non-termination loops.

The *context* C can be further formalized as a tuple: $C = \langle E, I \rangle$, 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, which is not shared with the agent. The problem information comprises information such as service descriptions, task descriptions, and documentation about available APIs that is directly shared 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 general state of the entire system, e.g., checking whether all of the services are up and running, after the problem is resolved, rather than 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 5.3.1. Consider the problem of localizing a Kubernetes target port misconfiguration in a social network application. AIOpsLab makes it easy to define this problem 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 fault localization, and the solution S is the microservice named "user-service", which is also the fault injection target. 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 AIOps tasks 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.

5.4 Orchestrator

AIOpsLab's Orchestrator strictly enforces the separation of concerns between the agent and the service, using a well-defined central piece, the Orchestrator. It provides a robust set of interfaces that allow seamless integration and extension of various system components.

5.4.1 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, which can prove distracting for agents and hamper performance.

The ACI specifies (1) the set of valid actions available to the agent ❼ ❾, and (2) how the service's state is conveyed back to the agent as the observation of its actions ❽.

In doing so, the ACI abstracts the cloud environment's complexity, simplifying 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 their objectives. Some APIs that AIOpsLab provides by default include `get_logs` (fetch logs), `get_metrics` (fetch metrics), `get_traces` (fetch traces), and `exec_shell` (execute shell commands after applying security policy filters).

Example 5.4.1. This example illustrates how the ACI is defined in AIOpsLab as APIs that agents can use.

```
class TaskActions:
    def get_traces(ns: str, duration: int = 5) -> str:
        """
        Collects trace data of the services from 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_traces`. On initializing a problem, the Orchestrator automatically extracts documentation from these APIs to provide as *context C* 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.

5.4.2 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 are registered with the Orchestrator, and a session starts with simple API calls passing a unique problem identifier ❶. AIOpsLab's design is highly flexible and integrates 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 that this could be a simple wrapper function around any existing agent framework.

5.4.3 Other Interfaces

Problem Initializers. As described in Section 5.3, each problem is defined with a *context C* 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 [177] to deploy the required cloud service for each problem.
As shown in Figure 25, to create realistic benchmark scenarios, the Orchestrator then interfaces with two entities: (1) a *workload generator* ❺ and (2) a *fault generator* ❸. These generators help introduce controlled service disruptions that simulate live benchmark problems. AIOpsLab currently adapts the generator from the application, e.g., the `wrk2` tool from DeathStarBench [136] and `Locust` from OpenTelemetry [138], which supports custom generation policies ❻. 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 ❹, 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 into more complex resolution strategies. We detail this fault library in Section 5.6.

Problem Evaluators. Finally, the Orchestrator plays a critical role 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 ❿. 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 sent to AIOpsLab). Additionally, AIOpsLab provides an optional qualitative evaluation of agent trajectories using LLMs-as-Judges [178].

Beyond that, all user-defined evaluation metrics specific to the problem are run. For instance, for the localization problem in Example 5.3.1, 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. All of the evaluation results will be automatically collected.

5.5 Cloud Services

AIOpsLab deploys live applications to create realistic testing environments ❷. AIOpsLab supports diverse cloud applications, including the HotelReservation and SocialNetwork applications from DeathStarBench [136] and OpenTelemetry's Astronomy Shop demo application [138]. The SocialNetwork application has 28 microservices, including Memcached, MongoDB, and Redis, that together implement several features of real-world social networking applications. The HotelReservation application, implemented with Go and gRPC, supports services like recommending and reserving hotels. The OpenTelemetry Astronomy Shop demo application contains microservices written in different programming languages, with services communicating over gRPC and HTTP. Additionally, we have also created custom applications using the TiDB Operator [179] to manage and operate TiDB clusters on Kubernetes.

5.6 Task-Oriented Fault Library

To instantiate problems across the task levels of Table 18, we use fault injection to inject faults into the system and construct a problem pool for AIOpsLab. We classify the faults into two main types: symptomatic faults and functional faults.

5.6.1 Symptomatic Faults

Symptomatic faults, such as performance degradation and crash failures, manifest as observable symptoms, such as increased latency, resource exhaustion, or service outages. These faults typically help to construct Level-1 and Level-2 tasks in the taxonomy, which can create problems that evaluate AIOps approaches' detection and localization ability. These faults provide an overview of potential problems but do not necessarily reveal the deeper, underlying root causes of issues (since they do not have one). AIOpsLab integrates the fault injection tool Chaos-Mesh [175] to inject symptomatic faults into microservice applications, and also has the faults enabled by the `flagd` [180] feature flag service in the Astronomy Shop demo application.

5.6.2 Functional Faults

As discussed in Section 5.1, the failure scenarios used to evaluate AIOps agents across tasks must go beyond simple performance or crash failures and reflect realistic cases that challenge agents, which is where functional faults come into play. Functional faults, such as misconfigurations, code bugs, and authorization issues at the application and virtualization layers, require approaches to not only detect (Level 1) and localize (Level 2) the failure but also diagnose the root cause (Level 3), such as incorrect deployment or operations, and apply the correct mitigation strategies (Level 4). For instance, one such fault revokes the admin authentication for the MongoDB database of the geographic microservice (Mongodb-geo) in the HotelReservation application. Since the Geo service relies on its backend database, errors will appear during its invocation. For each fault, AIOpsLab provides the injection function for its associated failure scenarios and offers the corresponding mitigation mechanism to recover the system from the erroneous state. Section 5.8 discusses the problem pool we instantiate from this fault library.

5.7 Observability

AIOpsLab is equipped with an extensible observability layer to provide comprehensive monitoring capabilities. AIOpsLab collects a wide array of telemetry data with its telemetry collector, including (1) traces from Jaeger [181] detailing the end-to-end paths of requests through distributed systems, (2) application logs retrieved by Kubectl, or formatted and recorded by Filebeat [182] and Logstash [183], and (3) system metrics monitored by Prometheus [184]. AIOpsLab not only supports data collection during the interaction with the LLM agent but can also export the data offline to facilitate evaluating other traditional AIOps approaches. Besides, AIOpsLab is designed to capture information from other dimensions, e.g., codebase, configuration, and cluster information. Developers can also design and expose low-level system information, such as syscall logs [185], [186], [187], to agents using AIOpsLab's interface.

5.8 Evaluation

We use AIOpsLab to evaluate four LLM-based agents and three non-LLM-based AIOps algorithms across the four task levels of Section 5.3. To keep the comparison fair, every agent is registered in AIOpsLab without any fine-tuning or benchmark-specific modification.

Problem pool. The AIOpsLab benchmark contains more than 100 problems in its problem pool. Table 19 lists the subset of faults used to instantiate problems. Due to the prohibitive time and monetary costs of evaluating all problems across every agent, we selectively evaluate 48 representative problems: 13 detection, 13 localization, 11 root cause analysis, and 11 mitigation problems. Since each RCA problem carries two sub-tasks (identifying the system level and the fault type), the agents are scored on 59 task instances in total. As shown in Table 19, most of the functional faults can be used to create problems at all of the four task levels, while the symptomatic faults (e.g., Faults 8–9) can only be used to create problems at the detection and localization levels (Level 1 and Level 2). In the detection-level task, the agents must identify the presence of faults in real time. This task is a binary classification, where the agents have to respond either "yes" if a fault is present or "no" on the contrary. The detection task (Level 1) can be made more complex, e.g., by asking the agents to label the abnormal telemetry data; however,

we keep it simple here and leave the complex tasks to other levels. The localization (Level 2) task asks the agents to specify the exact location of the fault, e.g., a service or pod name in Kubernetes. The RCA task (Level 3) requires the agents to identify (1) the system layer the fault affects and (2) the type of the fault, e.g., misconfiguration or operation error. The mitigation task (Level 4) requires the agents to *interact with the environment* to fix the fault with a series of actions, such as updating the configuration or rolling back to a previous version.
Most faults enable users to extend and create new problems easily by injecting the fault into other targets, such as services. For example, Fault 2 in AIOpsLab can be injected into 10 services by simply configuring the injection target. We select the "user-service", "text-service", and "post-storage-service" from SocialNetwork as injection targets. Injecting faults into different targets is crucial because each service may have distinct dependencies, resulting in varied fault "blast radius" or failure propagation topologies. Consequently, faults can manifest at different locations within the distributed architecture, e.g., the microservice, to help evaluate the ability of the AIOps agents since different locations may indicate distinct difficulties. Applying some faults to construct problems may require additional effort. For example, Fault 3 and Fault 4 require the users to not only prepare the scripts to trigger the admin privilege revoke or user unregistration during the testing, but also update the ConfigMap of the application in Kubernetes; and Fault 1 needs to enforce its TLS requirements through a Helm configuration update. Furthermore, some faults are designed for specific problems and are not readily adaptable, such as Fault 5, which involves an application-level code bug in the microservice's image.

Agents and baselines. The four LLM-based agents are (1) **GPT-w-Shell (GPT-3.5)** and (2) **GPT-w-Shell (GPT-4)** [188]: naïve baselines backed by GPT-3.5-turbo and GPT-4-turbo respectively, with access only to a secure shell; (3) **ReAct** [189]: an agent that extends chain-of-thought reasoning [22] by interleaving reasoning steps and actions; and (4) **FLASH** [26]: a workflow-automation agent that monitors execution status, decomposes long instructions into manageable conditional segments, and incorporates hindsight generation. Since FLASH was not publicly available at the time of writing, we develop a simplified version that retrospectively generates insights after each step. We complement these with three state-of-the-art non-LLM

Table 19: Selected faults used to instantiate the problems for evaluation in AIOpsLab. "Ext." stands for extensibility: a filled circle denotes the fault can be easily used to construct other problems; a half-filled circle denotes some manual effort is needed to create new problems; an empty circle means the fault is specific to some problems and cannot be applied to create other problems.

No.	Name	Application	Task Level	Category	Ext.	# Prob.	Description
1	AuthenticationMissing	HotelReservation	1, 2, 3, 4	Functional, Virtualization	◐	4	Missing authentication credentials cause access denial to MongoDB.
2	TargetPortMisconfig	SocialNetwork	1, 2, 3, 4	Functional, Virtualization	●	12	The service cannot connect to the specified port due to misconfiguration.
3	RevokeAuth	HotelReservation	1, 2, 3, 4	Functional, Application	◐	8	Revoked authentication causes database connection failure.
4	UserUnregistered	HotelReservation	1, 2, 3, 4	Functional, Application	◐	8	The database service has access failures after the user was unregistered.
5	BuggyAppImage	HotelReservation	1, 2, 3, 4	Functional, Application	○	4	Connection code bug in the application image causes access issues.
6	ScalePod	SocialNetwork	1, 2, 3, 4	Functional, Virtualization	●	4	Incorrect scaling operation makes the number of pods zero for a service.
7	AssignNonExistentNode	SocialNetwork	1, 2, 3, 4	Functional, Virtualization	●	4	Pod in a pending or failure status due to wrong assignment to a non-existent node.
8	NetworkLoss	HotelReservation	1, 2	Symptomatic	●	2	Network loss causes communication failures for a specific service.
9	PodFailure	HotelReservation	1, 2	Symptomatic	●	2	Service interruption due to a pod failure.
10	Noop	HotelReservation, SocialNetwork	1	–	●	2	No faults injected into the system.
11	OperatorOverload	TiDB Operator	1, 2, 3, 4	Functional, Virtualization	○	4	TiDB Kubernetes operator misconfig sets a very high number of TiDB replicas.
12	ServiceUnreachable	AstronomyShop	1, 2	Functional, Application	◐	2	Use a bad address when calling the PaymentService to make it unavailable.
13	ContainerKill	Any	1, 2	Symptomatic	●	2	A specific container is killed.
14	PVRedeployment	HotelReservation	1, 2, 3, 4	Functional, Virtualization	◐	4	Persistent volume is not correctly deleted, causing errors in redeployment.
15	WrongBinUsage	HotelReservation	1, 2, 3, 4	Functional, Virtualization	●	4	Wrong binary running in a service.

AIOps algorithms: **MKSMC** [190] for detection, **RMLAD** [191] and **PDiagnose** [192] for localization. These classical algorithms are fed telemetry data directly, exported offline by AIOpsLab's observability layer (Section 5.7), rather than interacting with the environment.

Metrics. AIOpsLab measures three things for every agent run. *Correctness* captures whether the agent successfully detects, localizes, analyzes, or mitigates the problem as expected. *Time/Steps* measures the wall-clock time and the number of agent–orchestrator interactions,

including derived metrics such as Time-to-Detect (TTD) and Time-to-Mitigate (TTM); the step count is the number of agent–orchestrator interactions, not the number of LLM calls. *Cost* reports the total tokens (input + output) generated by the agent and the orchestrator, as a proxy for the monetary cost of running the agent at scale.

Headline results. AIOpsLab surfaces clear differences between agents and clear gaps between current agentic capabilities and the demands of real operations. Among the LLM-based agents, FLASH reaches the highest overall accuracy at 59.32%, followed by ReAct (55.93%) and GPT-w-Shell with GPT-4 (49.15%); GPT-w-Shell with GPT-3.5 trails at 15.25% (Table 20). Token usage does not correlate with quality: ReAct consumes roughly 17K tokens per problem on average and is still outperformed by FLASH at roughly 6K tokens. Detection, the simplest task and the first one an AIOps agent should perform, is solved completely only by FLASH on the fault-present detection problems (Table 21); ReAct and the GPT-w-Shell baselines all miss some detection problems. Localization is harder than detection for the stronger agents, RCA and mitigation prove the most challenging, and no agent reaches even 60% on mitigation (Table 22). The classical AIOps algorithms (MKSMC, RMLAD, PDiagnose) run in one to two seconds but achieve substantially lower accuracy than the LLM-based agents on the tasks they support, illustrating the trade-off between speed and correctness that AIOpsLab is designed to expose. Together these results show that, while modern LLM-based agents can already solve a meaningful fraction of operational problems, the gap to fully autonomous, correct-by-default cloud operation is still wide and clearly measurable.

Table 20: Overall performance of the LLM-based agents in AIOpsLab: the lines of code (LoC) to register the agent, average running time, average number of steps, average tokens used, and accuracy across all problems.

Agent	LoC	Time (s)	# Steps	Tokens	Accuracy
GPT-w-Shell (GPT-4)	41	28.61	6.44	6,394.5	49.15%
GPT-w-Shell (GPT-3.5)	41	12.44	14.70	2,557.95	15.25%
ReAct	49	43.79	11.50	16,941.46	55.93%
FLASH	60	99.64	8.48	6,484.25	**59.32%**

Table 21: Agent performance on the detection and localization tasks. Input/Output is the average number of tokens given to and produced by the agent. For localization, agents may answer with a ranked list of suspect services; Acc.@1 and Acc.@3 score the top-1 and top-3 answers.

(a) Detection

Agent	Accuracy	Time (s)	# Steps	Input	Output
GPT-w-Shell (GPT-4)	69.23%	7.08	3.85	5,492	132
GPT-w-Shell (GPT-3.5)	23.07%	11.05	13.60	1,940.44	385.56
ReAct	76.92%	39.00	11.46	15,608.08	933.15
FLASH	**100%**	78.27	6.77	12,869.08	125.69
MKSMC	15.38%	1.00	N/A	N/A	N/A

(b) Localization

Agent	Acc.@3	Acc.@1	Time (s)	# Steps	Input	Output
GPT-w-Shell (GPT-4)	61.54%	**61.54%**	7.04	4.23	4,588.07	133.23
GPT-w-Shell (GPT-3.5)	30.77%	30.77%	6.26	11.92	1,784.23	217.08
ReAct	**69.23%**	53.85%	38.65	11.08	4,760.77	880.92
FLASH	61.54%	46.15%	56.60	5.77	1,875.08	123.31
PDiagnose	15.38%	15.38%	1.02	N/A	N/A	N/A
RMLAD	7.69%	7.69%	1.98	N/A	N/A	N/A

Influence of the step limit. We examine the impact of the maximum number of allowed steps on agent performance; Figure 26 shows the results. The step limit significantly affects some agents: ReAct and FLASH improve with more steps, with FLASH reaching its peak accuracy of

Table 22: Agent performance on the RCA and mitigation tasks. The classical AIOps algorithms do not support these tasks.

(a) Root cause analysis (RCA)

Agent	Accuracy	Time (s)	# Steps	Input	Output
GPT-w-Shell (GPT-4)	40.90%	8.68	4.81	4,297.91	176.18
GPT-w-Shell (GPT-3.5)	9.09%	10.06	14.00	1,495.55	406.27
ReAct	**45.45%**	32.16	8.00	16,276.09	757.27
FLASH	36.36%	59.00	6.09	1,193.90	152.45

(b) Mitigation

Agent	Accuracy	Time (s)	# Steps	Input	Output
GPT-w-Shell (GPT-4)	27.27%	99.47	13.72	10,142.55	1,060.00
GPT-w-Shell (GPT-3.5)	0%	23.78	20.00	3,178.33	967.71
ReAct	36.36%	67.18	15.54	29,211.90	1,464.90
FLASH	**54.55%**	216.41	16.09	8,469.00	760.36

59.32% at a step limit of 20, whereas for GPT-w-Shell (GPT-3.5), increasing the limit beyond 5 does not improve accuracy and merely increases token consumption. The plateauing accuracy indicates that *self-repair with environment feedback saturates quickly for AIOps problems*. This contrasts with development tasks such as code generation, where feedback from compositional tools (linters, type checkers, and test cases) helps agents improve continuously. The observation suggests the need for better task decomposition and planning for AIOps problems, improved feedback mechanisms for intermediate steps, and solutions that go beyond environment feedback and self-repair.

Agent behavior. Figure 27 shows the telemetry API usage patterns of the agents. The `get_logs` API is the most frequently used across all agents, followed by `get_metrics` and `get_traces`; however, agents diverge in their usage patterns. For example, FLASH does not use the `get_traces` API at all.

Wasting steps on unnecessary actions. Agents often waste steps on unnecessary actions, such as repeatedly calling the same API or generating invalid or non-existent API commands in loops (particularly GPT-w-Shell with GPT-3.5), leading to repeated errors in

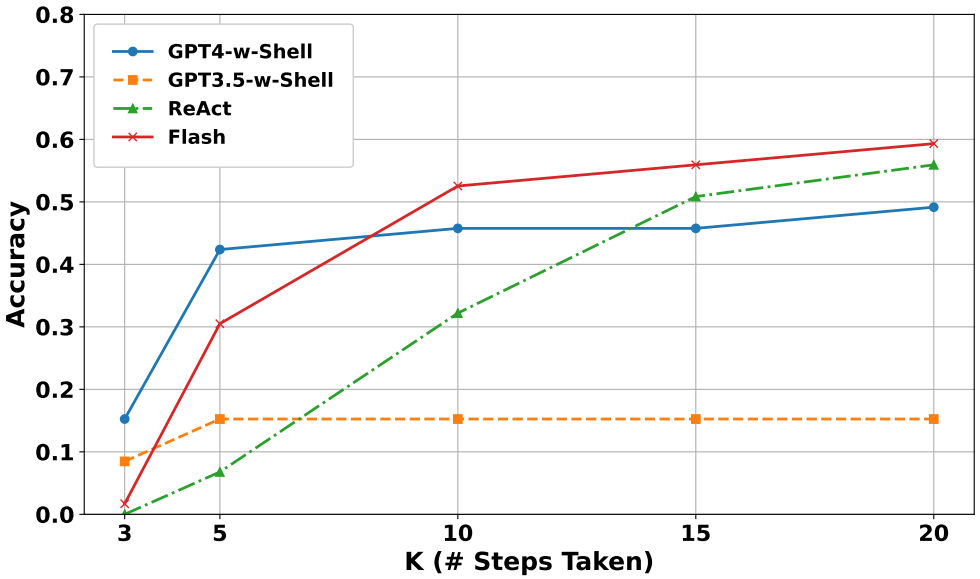


Figure 26: Agent performance vs. the number of steps taken.

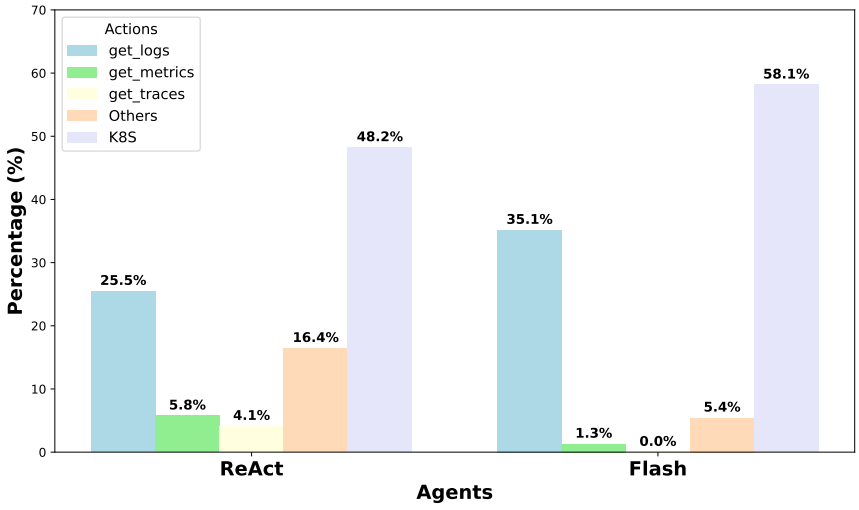


Figure 27: Telemetry API usage patterns among the agents.

execution. Agents can also occasionally attempt to execute non-existent helper scripts, such as `analyze_data.py` or `anomaly_detection.py`, despite no such Python scripts being implemented, potentially misled by examples in their own prompts. All such behaviors waste steps and tokens, and they motivate the step-limit and cost metrics that AIOpsLab reports alongside accuracy.

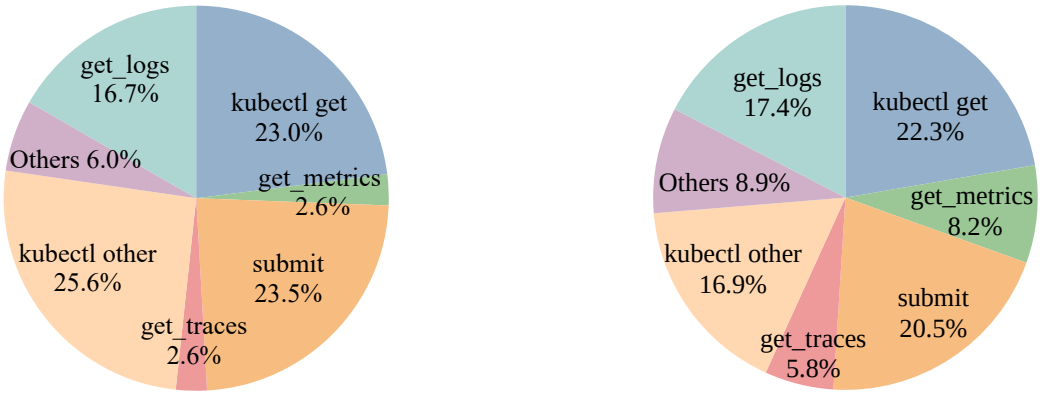
Overloaded information when consuming data. To dig deeper into the agent failure modes, we analyze the correlation between the agents' actions and the success or failure of problem-solving, as well as the distribution of actions across steps. In Figure 28, we present the distribution of actions for both successful and failed cases. Agents tend to use the `get_metrics` and `get_traces` APIs sparingly in successfully resolved problems, typically only when necessary. This is understandable, as the metrics data, e.g., CPU and memory usage, have numerous values, which are hard to directly interpret, and trace data are descriptive records of the system's dependencies, which are more comprehensible when visualized. However, agents may subsequently consume these data with a `cat` command directly, which can overwhelm the model's input context window and cause distraction and more tokens to be consumed. Consequently, using these telemetry APIs without careful consideration or analysis can add more noise into the agents' reasoning, possibly leading to token exhaustion. We expect more refined telemetry data

processing and filtering mechanisms to be implemented in the agents to avoid this issue in the future.

Invalid API usage. Agents can also struggle with improper formatting of API calls. For instance, GPT-w-Shell (GPT-3.5) consistently generates incorrect command formats (though the API name is correct), such as malformed parameters, and repeats the error in subsequent steps. In many cases, GPT-w-Shell (GPT-3.5) repeatedly responds with "I apologize for the error. Here is the API call again:" followed by the same or a similar mistake. This issue is observed as many as 14 times within a 20-step problem-solving case. Similar issues with invalid API usage are also observed in other agents. For instance, the ReAct agent occasionally generates incorrect API commands, but typically recovers by reasoning through the errors and self-correcting its commands. In the following example, ReAct uses an incorrect parameter for the `get_logs` API, but subsequently corrects it by checking the existing services in its next step.

ReAct: `get_logs("test-social-network", "Social Network")`
AIOpsLab: Error: Your service/namespace does not exist.
ReAct:
Thought: I should check the services deployed within the test-social-network namespace to ensure I use the correct service name.
Action: I will list the services in the test-social-network namespace to confirm the correct name of the service.

```
exec_shell("kubectl get services -n test-social-network")
```



(a) Successful cases. (b) Failure cases.

Figure 28: Action distribution by success and failure cases.

False positives on fault-free problems. To further evaluate the agents' detection performance, we set up two detection problems for two microservice applications (HotelReservation and SocialNetwork) where no faults exist, referred to as no-operation problems (Fault 10, Noop, in Table 19). Only GPT-w-Shell (GPT-4) correctly identifies these cases as normal system execution, while the other agents, including FLASH, report false positives, misinterpreting normal activities (e.g., standard workload generation) as anomalies.

5.9 Summary

This chapter presented AIOpsLab, the first end-to-end benchmark suite for evaluating LLM-based agents across the full cloud incident lifecycle of detection, localization, root-cause analysis, and mitigation. AIOpsLab provides realistic microservice deployments, a fault library that goes beyond surface-level symptoms to include fine-grained root causes, an Agent-Cloud Interface that abstracts cloud complexity, and a session-based Orchestrator that turns ad-hoc agent demonstrations into reproducible experiments. By evaluating four LLM-based agents and three classical AIOps algorithms across all four task levels, AIOpsLab makes the gap between current agentic capabilities and the demands of real cloud operation concrete and comparable. The AIOpsLab artifacts are publicly available at `https://github.com/microsoft/AIOpsLab`. We close this chapter by discussing the practical limitations of AIOpsLab and the directions in which it can be extended.

Granularity of evaluation oracles. AIOpsLab's task evaluators are correctness-based: an agent's answer is matched against a predefined oracle (e.g., the name of the faulty microservice or the matching fault category). This works well when an agent's reasoning either converges to the correct answer or does not, but it can miss cases where an agent answers correctly for the wrong reason. For instance, in a binary-choice detection task, an agent may identify an anomaly correctly but reference an unrelated, normal workload in its explanation. AIOpsLab's optional LLM-as-Judge mode helps surface such cases by comparing the agent's reasoning chain against the problem description, and we expect richer trajectory-level evaluation to be a productive future-work direction.

Coverage of the fault library. The current fault library focuses on application- and virtualization-level faults of the form most commonly seen in microservice incident reports: misconfigurations, code defects, authorization issues, network/storage problems, operational mistakes, and dependency failures. Many other axes of fault are within scope: hardware-level faults, multi-fault concurrent scenarios, and workload-anomaly faults that are independent of the system itself. We designed the Orchestrator's interfaces so that adding new fault types or whole new fault categories does not require modifying the benchmark core, and we expect the library to grow organically as the community contributes new scenarios.

Coverage of the agent space. Our evaluation focuses on four LLM-based agents and three classical AIOps baselines. The agentic SRE space is moving quickly: new agent frameworks, new tool-use protocols, and new model families appear frequently. AIOpsLab is designed so registering a new agent only requires implementing a small `get_action` interface; we intend the benchmark to serve as a stable evaluation harness against which future agentic systems can be compared on equal footing.

Realism of the cloud environment. AIOpsLab uses live, emulated microservice deployments rather than production cloud services, which avoids confounding evaluation with the operational quirks of any single cloud provider but also intentionally trades off some real-world fidelity. Production environments include network behavior, multi-tenancy, and operator practices that are difficult to reproduce in any benchmark. Pairing AIOpsLab with field studies of agentic SRE systems deployed in production is a natural next step.

Chapter 6 Conclusion and Future Work

This dissertation has addressed the central challenge of achieving autonomous troubleshooting for modern cloud systems. As cloud systems grow increasingly complex and incidents become routine at scale, the human-in-the-loop reliability practice that today's clouds depend on cannot keep pace with the rate at which failures arrive. To close the loop on autonomous troubleshooting, this dissertation has presented systems that span the entire cloud incident lifecycle, together with a benchmark for agentic solutions.

In the pre-incident stage, we presented Rainmaker, a push-button reliability testing tool that intercepts and manipulates outbound REST API calls at the HTTP layer to inject realistic transient cloud faults. Built on a four-pattern bug taxonomy developed by analyzing how error handling goes wrong under the cloud-based fault model, Rainmaker uncovered 73 new error-handling bugs across 11 mature .NET applications maintained by Microsoft, Petabridge, and others; 55 of these bugs have been confirmed and 51 fixed by upstream maintainers, at a 1.96% false-positive rate. Rainmaker turns systematic fault injection into a routine part of the development workflow.

In the post-incident stage, we introduced RCACopilot, an automated root-cause-analysis system that adapts large language model (LLM) reasoning to cloud incident management. RCACopilot composes per-alert-type incident handlers from three reusable actions to collect multi-source diagnostic data, summarizes that data with an LLM to fit within prompt budgets, and predicts the root-cause category through nearest-neighbor retrieval over historical incidents combined with chain-of-thought prompting. On a one-year dataset of 653 production incidents, RCACopilot achieved micro-F1 of 0.766 and macro-F1 of 0.533 with sub-five-second inference latency, substantially outperforming classical machine-learning baselines and naïve LLM approaches; its diagnostic-information-collection module has been deployed in production across more than 30 service teams.

Beyond diagnosis, we extended autonomous troubleshooting to mitigation with **Stratus**, an LLM-based multi-agent system for autonomous Site Reliability Engineering. Stratus orchestrates four specialized agents (detection, diagnosis, mitigation, and undo) through a

deterministic state machine and exposes them to the cloud through Agent-Cloud Interfaces. At its core is *Transactional No-Regression* (TNR), a formal safety specification that guarantees any unsuccessful mitigation can be undone and that the externally visible system severity never grows beyond the baseline at the time the failure was detected. STRATUS significantly outperforms state-of-the-art agentic baselines on AIOPSLAB and ITBench mitigation problems by at least 1.5× across multiple LLM backends, while ensuring that the system never enters a state worse than the one in which the failure was first observed.

Finally, to allow the research community to evaluate agentic SRE systems on equal footing, we presented **AIOPSLAB**, the first end-to-end benchmark suite for LLM-based agents on the full cloud incident lifecycle. AIOPSLAB provides realistic microservice deployments, a fault library that goes beyond surface symptoms to include fine-grained root causes, an Agent-Cloud Interface that abstracts cloud complexity, and a session-based Orchestrator that turns ad-hoc agent demonstrations into reproducible experiments. Across four LLM-based agents and three classical AIOps algorithms, AIOPSLAB makes the gap between current agentic capabilities and the demands of real cloud operation concrete and comparable.

Taken together, these contributions cover three complementary aspects of autonomous cloud troubleshooting. The failure surface can be reduced before production by systematic testing of an application's error-handling paths under realistic transient cloud faults. The failures that do reach production can be diagnosed by adapting LLM reasoning to multi-source telemetry, and they can be mitigated by a multi-agent system in which the underlying safety property is stated formally and enforced by construction. The progress of agentic SRE systems can be measured on a benchmark that covers the full incident lifecycle.

The systems and ideas presented in this dissertation contribute to the broader effort of moving cloud reliability from human-in-the-loop operation toward systems that can detect, diagnose, and recover from their own failures with substantially less human intervention.

6.1 Discussion

Societal implications. The systems in this dissertation, and STRATUS in particular, take state-changing actions on live production infrastructure with little or no human in the loop, and

this autonomy carries societal weight that a purely technical evaluation does not capture. Cloud systems are essential services in finance, healthcare, and communication, so a mitigation that an agent commits, or fails to undo, can propagate harm far beyond the immediate operator. A first concern is accountability and liability: when an autonomous agent acts on critical infrastructure and causes damage, it is unclear who is answerable: the operator who delegated control, the organization that deployed the agent, or the vendor of the underlying model, because existing operational and legal norms presume a human decision-maker who can be identified after the fact. A second is transparency and auditability: consequential autonomous actions demand a faithful, tamper-evident record of what was changed, why, and with what effect, so that incidents can be reconstructed and responsibility assigned. A third is automation bias and the erosion of operator vigilance: as agents handle a growing share of incidents successfully, the humans nominally supervising them lose situational awareness and the practiced skill to intervene at precisely the moment an agent errs in a way it cannot detect. A fourth, and the most systemic, is the concentration of operational control: if many independent systems delegate mitigation to agents built on the same models, prompts, and heuristics, their failure modes become correlated, and a single flawed reasoning pattern can trigger simultaneous, similarly wrong responses across otherwise unrelated services, turning a localized fault into a systemic one.

The human dimension. Autonomous mitigation does not remove humans from reliability engineering; it relocates them. As systems like STRATUS take over front-line work, the SRE role shifts from responder, i.e., the engineer paged at 3 a.m. to execute a runbook to supervisor and, more fundamentally, specifier of agent behavior. This relocation is where the formal guarantees of Chapter 4 ultimately rest. The same transactional mechanism described above ensures that no externally visible state ever regresses below the baseline, but only with respect to the health conditions that humans define. If that specification omits a dimension of harm, e.g., a corrupted record, a violated data-residency constraint, then the agent can commit a transaction that the guarantee certifies as no-regression while the system is, in every sense that matters operationally, worse off. The safety property is therefore conditional on the comprehensiveness of human specification: an incomplete set of health conditions yields unsafe behavior in spite of a sound

formal guarantee, and closing that gap is irreducibly human work. This frames the SRE's most consequential contribution not as keystrokes during an incident but as the up-front, auditable authoring of what must never silently regress, which draws on exactly the tacit operational knowledge that delegating routine incidents to agents, while it relieves on-call burden and toil, risks eroding over time. For the novel, ambiguous, or high-stakes incidents that fall outside an agent's competence, keeping a human in the loop remains essential, and STRATUS already supports this hybrid mode of operation: its writer agents can request human approval before execution, with interceptable action stacks, so that a human retains authority over the transactions whose consequences are hardest to specify in advance.

6.2 Future Work

The work in this dissertation has shown how AI can enhance existing cloud systems when applied as an afterthought. Looking forward, we aim to rethink the design of AI-native systems that integrate intelligence as a core part of their operation. We outline five directions for future work.

Efficient multi-agent and multi-model coordination. STRATUS introduced a multi-agent design for SRE, but coordination efficiency at scale remains an open challenge. While much recent work has optimized single-model inference and serving efficiency, the higher-level coordination across multiple agents and models has received far less attention. Two open problems are: how agents can synchronize effectively without incurring excessive or unordered communication, and how to decide when to invoke large models versus smaller ones in latency-sensitive scenarios. We plan to explore adaptive mechanisms for agent coordination and model selection that enable responsive and resource-efficient intelligence.

Safety assurance and execution confinement in operations. Safety is the cornerstone of trustworthy autonomous operation. Agents interact with external systems through API calls, for instance via the Model Context Protocol (MCP), to retrieve information or modify external system states. Without proper safeguards, such agent actions can be hazardous and may drive the system into worse states. We will continue to design mechanisms that make agent actions reversible and interpretable. Specifically, we plan to integrate rollback-aware execution

that removes side effects whenever necessary, and to develop validation oracles that continuously assess post-action system health, deciding whether the agent should retry, reflect, or escalate. These designs extend the Transactional No-Regression (TNR) property introduced in Chapter 4 to a broader class of agent-driven actions in production systems.

Extending autonomous operation beyond cloud systems. Drawing from this dissertation's work on autonomous cloud operations, we plan to explore how intelligent systems can manage cyber-physical and socio-technical environments such as IoT infrastructures, financial platforms, and robotic systems. Unlike cloud environments, these domains present additional challenges: actions can have physical or economic consequences, sensing is noisy, decisions must be both timely and safe, and computation may be limited. These differences call for a fundamental rethinking of how autonomous operation perceives, reasons, and acts when embedded within other real-world systems.

Human-AI cooperative operation. Even as operations become increasingly automated, human insight and intervention remain valuable. We plan to build collaborative autonomy in which humans and AI agents share control, context, and responsibility. Specifically, we will design interactive frameworks in which agents can explain their reasoning, quantify uncertainty, and request guidance when their confidence is low, while humans can provide corrective feedback that the agents incorporate through continual learning.

Toward more holistic evaluation platforms. Building scalable and reproducible evaluation platforms remains a central goal of our research agenda. Extending AIOpsLab (Chapter 5), we plan to develop platforms that exercise agentic AI systems under more realistic workloads and a wider range of fault types, so that deployment decisions can be made with greater confidence. Beyond operational effectiveness, future platforms should also assess security and trustworthiness, examining whether agents can withstand adversarial manipulation and mitigate security incidents such as DDoS attacks.

In summary, our future work aims to build a principled foundation for effective, safe, and efficient agentic AI.

References

[1] L. A. Barroso, U. Hölzle, and P. Ranganathan, *The Datacenter as a Computer: Designing Warehouse-Scale Machines*. 2018.

[2] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "Above the Clouds: A Berkeley View of Cloud Computing," University of California at Berkeley, Tech. Rep., Feb. 2009.

[3] J. B. Leners, H. Wu, W.-L. Hung, M. K. Aguilera, and M. Walfish, "Detecting Failures in Distributed Systems with the Falcon Spy Network," in *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP'11)*, Oct. 2011.

[4] P. Huang, C. Guo, J. R. Lorch, L. Zhou, and Y. Dang, "Capturing and Enhancing In Situ System Observability for Failure Detection," in *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI'18)*, Oct. 2018.

[5] X. Ren, S. Wang, Z. Jin, D. Lion, A. Chiu, T. Xu, and D. Yuan, "Relational Debugging — Pinpointing Root Causes of Performance Problems," in *Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI'23)*, Jul. 2023.

[6] Y. Zhang, S. Makarov, X. Ren, D. Lion, and D. Yuan, "Pensieve: Non-Intrusive Failure Reproduction for Distributed Systems using the Event Chaining Approach," in *Proceedings of the 26th ACM Symposium on Operating System Principles (SOSP'17)*, Oct. 2017.

[7] Z. Zeng, Y. Zhang, Y. Xu, M. Ma, B. Qiao, W. Zou, Q. Chen, M. Zhang, X. Zhang, H. Zhang, X. Gao, H. Fan, S. Rajmohan, Q. Lin, and D. Zhang, "TraceArk: Towards actionable performance anomaly alerting for online service

systems," in *Proceedings of the 45th IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP'23)*, 2023.

[8] B. Beyer, C. Jones, J. Petoff, and N. R. Murphy, *Site Reliability Engineering: How Google Runs Production Systems*. O'Reilly Media, Inc., Mar. 2016.

[9] B. Beyer, N. R. Murphy, D. K. Rensin, K. Kawahara, and S. Thorne, *Site Reliability Workbook: Practical Ways to Implement SRE*. O'Reilly Media Inc., Aug. 2018.

[10] M. Ma, S. Zhang, J. Chen, J. Xu, H. Li, Y. Lin, X. Nie, B. Zhou, Y. Wang, and D. Pei, "Jump-starting multivariate time series anomaly detection for online service systems," in *2021 USENIX Annual Technical Conference (ATC'21)*, 2021.

[11] C. Bansal, S. Renganathan, A. Asudani, O. Midy, and M. Janakiraman, "DeCaf: Diagnosing and triaging performance issues in large-scale cloud services," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP'20)*, 2020.

[12] C. Luo, J.-G. Lou, Q. Lin, Q. Fu, R. Ding, D. Zhang, and Z. Wang, "Correlating events with time series for incident diagnosis," in *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2014.

[13] J. Jiang, W. Lu, J. Chen, Q. Lin, P. Zhao, Y. Kang, H. Zhang, Y. Xiong, F. Gao, Z. Xu, et al., "How to mitigate the incident? an effective troubleshooting guide recommendation technique for online service systems," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'20)*, 2020.

[14] M. Shetty, Y. Chen, G. Somashekar, M. Ma, Y. Simmhan, X. Zhang, J. Mace, D. Vandevoorde, P. Las-Casas, S. M. Gupta, S. Nath, C. Bansal, and S. Rajmohan, "Building AI Agents for Autonomous Clouds: Challenges and

Design Principles," in *Proceedings of 15th ACM Symposium on Cloud Computing (SoCC'24)*, Nov. 2024.

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

[16] K. Kingsbury, *Jepsen*, `https://jepsen.io/`, 2022.

[17] P. D. Marinescu and G. Candea, "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)*, Jun. 2009.

[18] H. S. Gunawi, T. Do, P. Joshi, P. Alvaro, J. M. Hellerstein, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, K. Sen, and D. Borthakur, "Fate and Destini: A Framework for Cloud Recovery Testing," in *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI'11)*, Mar. 2011.

[19] T. Leesatapornwongsa, M. Hao, P. Joshi, J. F. Lukman, and H. S. Gunawi, "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)*, Oct. 2014.

[20] J. Lu, C. Liu, L. Li, X. Feng, F. Tan, J. Yang, and L. You, "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)*, Oct. 2019.

[21] V. Heorhiadi, S. Rajagopalan, H. Jamjoom, M. K. Reiter, and V. Sekar, "Gremlin: Systematic Resilience Testing of Microservices," in *Proceedings of the IEEE 36th International Conference on Distributed Computing Systems (ICDCS'16)*, Jun. 2016.

[22] J. Wei, X. Wang, D. Schuurmans, M. Bosma, F. Xia, E. Chi, Q. V. Le, and D. Zhou, "Chain-of-thought prompting elicits reasoning in large language models," *Advances in Neural Information Processing Systems (NeurIPS'22)*, 2022.

[23] P. Jin, S. Zhang, M. Ma, H. Li, Y. Kang, L. Li, Y. Liu, B. Qiao, C. Zhang, P. Zhao, et al., "Assess and summarize: Improve outage understanding with large language models," in *Proceedings of the Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2023.

[24] Y. Jiang, C. Zhang, S. He, Z. Yang, M. Ma, S. Qin, Y. Kang, Y. Dang, S. Rajmohan, Q. Lin, and D. Zhang, "Xpert: Empowering incident management with query recommendations via large language models," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering (ICSE'24)*, 2024.

[25] Z. Wang, Z. Liu, Y. Zhang, A. Zhong, L. Fan, L. Wu, and Q. Wen, "RCAgent: Cloud root cause analysis by autonomous agents with tool-augmented large language models," *arXiv:2310.16340*, 2023.

[26] X. Zhang, T. Mittal, C. Bansal, R. Wang, M. Ma, Z. Ren, H. Huang, and S. Rajmohan, *FLASH: A Workflow Automation Agent for Diagnosing Recurring Incidents*, `https://www.microsoft.com/en-us/research/publication/flash-a-workflow-automation-agent-for-diagnosing-recurring-incidents/`, 2024.

[27] *Crew AI*, `https://www.crewai.com/`.

[28] Q. Wu, G. Bansal, J. Zhang, Y. Wu, B. Li, E. Zhu, L. Jiang, X. Zhang, S. Zhang, J. Liu, A. H. Awadallah, R. W. White, D. Burger, and C. Wang, "AutoGen: Enabling next-gen LLM applications via multi-agent conversation framework," in *Proceedings of the 1st Conference on Language Modeling (COLM'24)*, 2024.

[29] J. Yang, C. E. Jimenez, A. Wettig, K. Lieret, S. Yao, K. Narasimhan, and O. Press, "SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering," in *Proceedings of the Thirty-eighth Annual Conference on Neural Information Processing Systems (NeurIPS'24)*, Sep. 2024.

[30] W. Hua, X. Yang, Z. Li, C. Wei, and Y. Zhang, "TrustAgent: Towards Safe and Trustworthy LLM-based Agents through Agent Constitution," *arXiv:2402.01586*, Feb. 2024.

[31] Microsoft Docs, *Use the Azurite emulator for local Azure Storage development*, `https://docs.microsoft.com/en-us/azure/storage/common/storage-use-azurite`, 2022.

[32] *Azure Blob Storage error codes*, `https://learn.microsoft.com/en-us/rest/api/storageservices/blob-service-error-codes`, 2022.

[33] *HTTP Status Codes for Azure Cosmos DB*, `https://docs.microsoft.com/en-us/rest/api/cosmos-db/http-status-codes-for-cosmosdb`, 2022.

[34] *Amazon Simple Storage Service Error Responses*, `https://docs.aws.amazon.com/AmazonS3/latest/API/ErrorResponses.html`, 2022.

[35] *Amazon Simple Queue Service Common Errors*, `https://docs.aws.amazon.com/AWSSimpleQueueService/latest/APIReference/CommonErrors.html`, 2022.

[36] Microsoft Docs, *Transient fault handling*, `https://learn.microsoft.com/en-us/azure/architecture/best-practices/transient-faults`, 2022.

[37] Azure/azure-sdk-for-net-9670, *Retry on 408, 500, 502, 504 status codes*, `https://github.com/Azure/azure-sdk-for-net/pull/9670`, 2022.

[38] botbuilder-dotnet-6407, *Activities that should be logged are missing due to transient network errors*, `https://github.com/microsoft/botbuilder-dotnet/issues/6407`, 2022.

[39] *microsoft/botbuilder-dotnet*, `https://github.com/microsoft/botbuilder-dotnet`, 2022.

[40] Orleans-7738, *Popping up queue messages may cause data loss and unexpected NullReferenceException*, `https://github.com/dotnet/orleans/issues/7738`, 2022.

[41] *dotnet/orleans*, `https://github.com/dotnet/orleans`, 2022.

[42] botbuilder-dotnet-5778, *__Checkedcontainers can become inconsistent with blob storage and further lead to unhandled exception*, `https://github.com/microsoft/botbuilder-dotnet/issues/5778`, 2021.

[43] G. R. Ganger, M. K. McKusick, C. A. N. Soules, and Y. N. Patt, "Soft Updates: A Solution to the Metadata Update Problem in File Systems," *ACM Transactions on Computer Systems (TOCS'00)*, May 2000.

[44] V. Atlidakis, P. Godefroid, and M. Polishchuk, "RESTler: Stateful REST API Fuzzing," in *Proceedings of the 37th International Conference on Software Engineering (ICSE'19)*, May 2019.

[45] Azure/azure-sdk-for-net-31001, *1-to-N Mappings between SDK API and REST APIs*, `https://github.com/Azure/azure-sdk-for-net/issues/31001`, 2022.

[46] aws-sdk-net-2102, *1-to-N Mappings between SDK API and REST APIs*, `https://github.com/aws/aws-sdk-net/discussions/2102`, 2022.

[47] Envoy Docs, *Envoy Fault Injection*, https://www.envoyproxy.io/docs/envoy/latest/api-v3/extensions/filters/http/fault/v3/fault.proto, 2022.

[48] Istio Docs, *Istio Fault Injection*, https://istio.io/latest/docs/tasks/traffic-management/fault-injection/, 2022.

[49] Microsoft Docs, *.NET CallContext Class*, https://docs.microsoft.com/en-us/dotnet/api/system.runtime.remoting.messaging.callcontext, 2022.

[50] Java API Specification, *Class InheritableThreadLocal<T>*, https://docs.oracle.com/javase/7/docs/api/java/lang/InheritableThreadLocal.html, 2022.

[51] C. Huo and J. Clause, "Improving Oracle Quality by Detecting Brittle Assertions and Unused Inputs in Tests," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'14)*, Nov. 2014.

[52] *Storage.NET*, https://github.com/aloneguid/storage, 2022.

[53] *Azure Blob Storage*, https://azure.microsoft.com/en-us/services/storage/blobs/, 2022.

[54] *Azure Queue Storage*, https://azure.microsoft.com/en-us/services/storage/queues/, 2022.

[55] *Azure Table Storage*, https://azure.microsoft.com/en-us/services/storage/tables/, 2022.

[56] *Azure Cosmos DB*, https://azure.microsoft.com/en-us/services/cosmos-db/, 2022.

[57] *Amazon S3*, https://aws.amazon.com/s3/, 2022.

[58] *Amazon SQS*, https://aws.amazon.com/sqs/, 2022.

[59] Microsoft Docs, *Install and use the Azure Cosmos DB Emulator for local development and testing*, https://learn.microsoft.com/en-us/azure/cosmos-db/local-emulator, 2022.

[60] DistributedLock-132, *Transient error leads to unhandled 409 when releasing an Azure lease*, https://github.com/madelson/DistributedLock/issues/132, 2022.

[61] IronPigeon-133, *make blob deletion tolerant of transient errors*, https://github.com/AArnott/IronPigeon/pull/133, 2022.

[62] AttachmentPlugin-277, *Transient error happening in container existence check will lead to container not found exception*, https://github.com/SeanFeldman/ServiceBus.AttachmentPlugin/issues/277, 2022.

[63] botbuilder-dotnet-5787, *Metadata of blob can be missing due to transient error which leads to unhandled exception*, https://github.com/microsoft/botbuilder-dotnet/issues/5787, 2022.

[64] FHIR Server-2732, *Retry other HTTP error codes from Cosmos DB?* https://github.com/microsoft/fhir-server/issues/2732, 2022.

[65] Orleans-7790, *Data was added repeatedly to the queue unexpectedly without any warning*, https://github.com/dotnet/orleans/issues/7790, 2022.

[66] W. Lam, K. Muşlu, H. Sajnani, and S. Thummalapenta, "A Study on the Lifecycle of Flaky Tests," in *Proceedings of the 42nd International Conference on Software Engineering (ICSE'20)*, May 2020.

[67] D. Yuan, Y. Luo, X. Zhuang, G. R. Rodrigues, X. Zhao, Y. Zhang, P. Jain, and M. Stumm, "Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems.," in *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI'14)*, 2014.

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

[69] D. Yuan, S. Park, P. Huang, Y. Liu, M. M. Lee, X. Tang, Y. Zhou, and S. Savage, "Be Conservative: Enhancing Failure Diagnosis with Proactive Logging," in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI'12)*, 2012.

[70] S. Jana, Y. Kang, S. R. Ohio, and B. Ray, "Automatically Detecting Error Handling Bugs Using Error Specifications," in *Proceedings of the 25th USENIX Security Symposium*, Aug. 2016.

[71] C. Rubio-González, H. S. Gunawi, B. Liblit, R. H. Arpaci-Dusseau, and A. C. Arpaci-Dusseau, "Error Propagation Analysis for File Systems," in *Proceedings of the ACM SIGPLAN 2009 Conference on Programming Language Design and Implementation (PLDI'09)*, Jun. 2009.

[72] W. Weimer and G. C. Necula, "Finding and Preventing Run-Time Error Handling Mistakes," in *Proceedings of the 19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'04)*, Oct. 2004.

[73] R. Banabic and G. Candea, "Fast Black-Box Testing of System Recovery Code," in *Proceedings of the 7th European Conference on Computer Systems (EuroSys'12)*, Apr. 2012.

[74] M. Christakis, P. Emmisberger, P. Godefroid, and P. Müller, "A General Framework for Dynamic Stub Injection," in *Proceedings of the 39th International Conference on Software Engineering (ICSE'17)*, May 2017.

[75] P. Zhang and S. Elbaum, "Amplifying Tests to Validate Exception Handling Code," in *Proceedings of the 34th International Conference on Software Engineering (ICSE'12)*, Jun. 2012.

[76] T. S. Pillai, V. Chidambaram, R. Alagappan, S. Al-Kiswany, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "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)*, Oct. 2014.

[77] A. Alquraan, H. Takruri, M. Alfatafta, and S. Al-Kiswany, "An analysis of network-partitioning failures in cloud systems," in *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation (OSDI'18)*, 2018.

[78] H. Chen, W. Dou, D. Wang, and F. Qin, "CoFI: Consistency-Guided Fault Injection for Cloud Systems," in *Proceedings of the 35th ACM/IEEE International Conference on Automated Software Engineering (ASE'20)*, Sep. 2020.

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

[80] X. Ju, L. Soares, K. G. Shin, K. D. Ryu, and D. D. Silva, "On Fault Resilience of OpenStack," in *Proceedings of the 12th ACM Symposium on Cloud Computing (SOCC'13)*, Oct. 2013.

[81] R. Alagappan, A. Ganesan, Y. Patel, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Correlated Crash Vulnerabilities," in *Proceedings of the*

12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16), Nov. 2016.

[82] J. Mohan, A. Martinez, S. Ponnapalli, P. Raju, and V. Chidambaram, "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)*, Oct. 2018.

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

[84] M. Canini, D. Venzano, P. Perešíni, D. Kostić, and J. Rexford, "A NICE Way to Test OpenFlow Applications," in *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI'12)*, Apr. 2012.

[85] C. S. Meiklejohn, A. Estrada, Y. Song, H. Miller, and R. Padhye, "Service-Level Fault Injection Testing," in *Proceedings of the ACM Symposium on Cloud Computing (SOCC'21)*, Nov. 2021.

[86] *AWS Fault Injection Simulator*, `https://aws.amazon.com/fis/`, 2022.

[87] G. Hunt and D. Brubacher, "Detours: Binary Interception of Win32 Functions," in *Proceedings of the 3rd USENIX Windows NT Symposium*, Jul. 1999.

[88] P. Reynolds, C. Killian, J. L. Wiener, J. C. Mogul, M. A. Shah, and A. Vahdat, "Pip: Detecting the Unexpected in Distributed Systems," in *Proceedings of the 3rd USENIX Symposium on Networked Systems Design and Implementation (NSDI'06)*, May 2006.

[89] P. D. Marinescu, R. Banabic, and G. Candea, "An Extensible Technique for High-Precision Testing of Recovery Code," in *Proceedings of the 2010 USENIX Annual Technical Conference (USENIX ATC'10)*, Jun. 2010.

[90] K. Kingsbury and P. Alvaro, "Elle: Inferring Isolation Anomalies from Experimental Observations," in *Proceedings of the VLDB Endowment*, Nov. 2020.

[91] P. D. Marinescu and G. Candea, "Efficient Testing of Recovery Code Using Fault Injection," *ACM Transactions on Computer Systems (TOCS'11)*, 2011.

[92] Z.-M. Jiang, J.-J. Bai, K. Lu, and S.-M. Hu, "Fuzzing Error Handling Code using Context-Sensitive Software Fault Injection," in *Proceedings of the 29th USENIX Security Symposium*, Aug. 2020.

[93] J. Sun, T. Su, J. Li, Z. Dong, G. Pu, T. Xie, and Z. Su, "Understanding and Finding System Setting-Related Defects in Android Apps," in *Proceedings of the 2021 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'21)*, Jul. 2021.

[94] L. Ravindranath, S. Nath, J. Padhye, and H. Balakrishnan, "Automatic and Scalable Fault Detection for Mobile Applications," in *Proceedings of the 12th International Conference on Mobile Systems, Applications, and Services (MobiSys'14)*, 2014.

[95] P. Krymets, *Unit testing and mocking with Azure SDK .NET*, `https://devblogs.microsoft.com/azure-sdk/unit-testing-and-mocking/`, 2020.

[96] S. Yoo and M. Harman, "Regression Testing Minimization, Selection and Prioritization: A Survey," *Software Testing, Verification, and Reliability*, Mar. 2012.

[97] aws-sdk-net-2084, *The retry request of PutBucket will behave differently between regions*, `https://github.com/aws/aws-sdk-net/discussions/2084`, 2022.

[98] Microsoft Docs, *.NET profiling*, `https://learn.microsoft.com/en-us/dotnet/framework/unmanaged-api/profiling/profiling-overview`, 2022.

[99] M. Shetty, C. Bansal, S. P. Upadhyayula, A. Radhakrishna, and A. Gupta, "AutoTSG: Learning and synthesis for incident troubleshooting," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'22)*, 2022.

[100] I. Chalkidis, "ChatGPT may pass the bar exam soon, but has a long way to go for the LexGLUE benchmark," *arXiv:2304.12202*, 2023.

[101] J. Kasai, Y. Kasai, K. Sakaguchi, Y. Yamada, and D. Radev, "Evaluating GPT-4 and ChatGPT on Japanese medical licensing examinations," *arXiv:2303.18027*, 2023.

[102] T. Ahmed, S. Ghosh, C. Bansal, T. Zimmermann, X. Zhang, and S. Rajmohan, "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)*, 2023.

[103] A. Mastropaolo, S. Scalabrino, N. Cooper, D. N. Palacio, D. Poshyvanyk, R. Oliveto, and G. Bavota, "Studying the usage of text-to-text transfer transformer to support code-related tasks," in *Proceedings of the 43rd International Conference on Software Engineering (ICSE'21)*, 2021.

[104] OpenAI, *Tiktoken: A Python library for tokenizing text*, `https://github.com/openai/tiktoken`, 2023.

[105] Z. Zhang, A. Zhang, M. Li, and A. Smola, "Automatic chain of thought prompting in large language models," in *The Eleventh International Conference on Learning Representations (ICLR'23)*, 2023.

[106] B. Arzani, S. Ciraci, B. T. Loo, A. Schuster, and G. Outhred, "Taking the blame game out of data centers operations with netpoirot," in *Proceedings of the 2016 ACM SIGCOMM Conference (SIGCOMM'16)*, 2016.

[107] C. Zhao, M. Ma, Z. Zhong, S. Zhang, Z. Tan, X. Xiong, L. Yu, J. Feng, Y. Sun, Y. Zhang, et al., "Robust multimodal failure detection for microservice systems," in *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, 2023.

[108] Y. Chen, H. Xie, M. Ma, Y. Kang, X. Gao, L. Shi, Y. Cao, X. Gao, H. Fan, M. Wen, J. Zeng, S. Ghosh, X. Zhang, C. Zhang, Q. Lin, S. Rajmohan, D. Zhang, and T. Xu, "Automatic Root Cause Analysis via Large Language Models for Cloud Incidents," in *Proceedings of the 19th European Conference on Computer Systems (EuroSys'24)*, Apr. 2024.

[109] Y. Chen, M. Shetty, G. Somashekar, M. Ma, Y. Simmhan, J. Mace, C. Bansal, R. Wang, and S. Rajmohan, "AIOpsLab: A Holistic Framework to Evaluate AI Agents for Enabling Autonomous Clouds," in *Proceedings of the Conference on Machine Learning and Systems (MLSys'25)*, May 2025.

[110] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic Concepts and Taxonomy of Dependable and Secure Computing," *IEEE Transactions on Dependable and Secure Computing (TDSC'04)*, Jan. 2004.

[111] H. S. Gunawi, M. Hao, T. Leesatapornwongsa, T. Patana-anake, T. Do, J. Adityatama, K. J. Eliazar, A. Laksono, J. F. Lukman, V. Martin, and A. D. Satria, "What Bugs Live in the Cloud? A Study of 3000+ Issues in Cloud Systems," in *Proceedings of the 5th ACM Symposium on Cloud Computing (SoCC'14)*, Nov. 2014.

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

[113] P. H. Hochschild, P. Turner, J. C. Mogul, R. Govindaraju, P. Ranganathan, D. E. Culler, and A. Vahdat, "Cores that don't count," in *Proceedings of the 18th Workshop on Hot Topics in Operating Systems (HotOS'21)*, Jun. 2021.

[114] J. Meza, T. Xu, K. Veeraraghavan, and O. Mutlu, "A Large Scale Study of Data Center Network Reliability," in *Proceedings of the 18th ACM Internet Measurement Conference (IMC'18)*, Oct. 2018.

[115] S. Maneas, K. Mahdaviani, T. Emami, and B. Schroeder, "A Study of SSD Reliability in Large Scale Enterprise Storage Deployments," in *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST'20)*, Feb. 2020.

[116] J. Meza, Q. Wu, S. Kumar, and O. Mutlu, "A Large-Scale Study of Flash Memory Failures in the Field," in *Proceedings of the 2015 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'15)*, Jun. 2015.

[117] B. Treynor, M. Dahlin, V. Rau, and B. Beyer, "The Calculus of Service Availability," *Communications of the ACM*, Aug. 2017.

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

[119] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou, "Rx: Treating Bugs As Allergies—A Safe Method to Survive Software Failures," in *Proceedings of the 20th Symposium on Operating System Principles (SOSP'05)*, Oct. 2005.

[120] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox, "Microreboot – A Technique for Cheap Recovery," in *Proceedings of the 6th USENIX Conference on Operating Systems Design and Implementation (OSDI'04)*, Dec. 2004.

[121] R. Bhagwan, R. Kumar, C. S. Maddila, and A. A. Philip, "Orca: Differential Bug Localization in Large-Scale Services," in *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI'18)*, Oct. 2018.

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

[123] T. Haerder and A. Reuter, "Principles of transaction-oriented database recovery," *ACM Computing Surveys*, CSUR, Dec. 1983.

[124] B. Alpern and F. B. Schneider, "Recognizing safety and liveness," *Distributed Computing*, Sep. 1987.

[125] B. Alpern, A. J. Demers, and F. B. Schneider, "Safety without stuttering," *Information Processing Letters*, 1986.

[126] B. Alpern and F. B. Schneider, "Defining Liveness," *Information Processing Letters*, 1985.

[127] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, Omega, and Kubernetes," *Communications of the ACM*, May 2016.

[128] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-Scale Cluster Management at Google with Borg," in *Proceedings of the 10th European Conference on Computer Systems (EuroSys'15)*, Apr. 2015.

[129] C. Tang, K. Yu, K. Veeraraghavan, J. Kaldor, S. Michelson, T. Kooburat, A. Anbudurai, M. Clark, K. Gogia, L. Cheng, B. Christensen, A. Gartrell, M. Khutornenko, S. Kulkarni, M. Pawlowski, T. Pelkonen, A. Rodrigues, R. Tibrewal, V. Venkatesan, and P. Zhang, "Twine: A Unified Cluster Management System for Shared Infrastructure," in *Proceedings of the 14th*

USENIX Conference on Operating Systems Design and Implementation (OSDI'20), Nov. 2020.

[130] T. Melissaris, K. Nabar, R. Radut, S. Rehmtulla, A. Shi, S. Chandrashekar, and I. Papapanagiotou, "Elastic Cloud Services: Scaling Snowflake's Control Plane," in *Proceedings of the 13th ACM Symposium on Cloud Computing (SOCC'22)*, Nov. 2022.

[131] X. Sun, L. Suresh, A. Ganesan, R. Alagappan, M. Gasch, L. Tang, and T. Xu, "Reasoning About Modern Datacenter Infrastructures Using Partial Histories," in *Proceedings of the 18th Workshop on Hot Topics in Operating Systems (HotOS'21)*, May 2021.

[132] L. Suresh, J. Loff, F. Kalim, S. A. Jyothi, N. Narodytska, L. Ryzhyk, S. Gamage, B. Oki, P. Jain, and M. Gasch, "Building Scalable and Flexible Cluster Managers Using Declarative Programming," in *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)*, Nov. 2020.

[133] A. Li, S. Lu, S. Nath, R. Padhye, and V. Sekar, "ExChain: Exception Dependency Analysis for Root Cause Diagnosis," in *Proceedings of the 21st USENIX Symposium on Networked Systems Design and Implementation (NSDI'24)*, Apr. 2024.

[134] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag, *Dapper, a Large-Scale Distributed Systems Tracing Infrastructure*, Apr. 2010.

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

[136] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, et al., "An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems," in *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'19)*, 2019.

[137] X. Zhou, X. Peng, T. Xie, J. Sun, C. Ji, W. Li, and D. Ding, "Fault Analysis and Debugging of Microservice Systems: Industrial Survey, Benchmark System, and Empirical Study," *IEEE Transactions on Software Engineering*, Dec. 2021.

[138] O. Community, *Opentelemetry astronomy shop demo*, `https://opentelemetry.io/docs/demo/`, 2025.

[139] G. Miao, L. E. Moser, X. Yan, S. Tao, Y. Chen, and N. Anerousis, "Generative Models for Ticket Resolution in Expert Networks," in *Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'10)*, Jul. 2010.

[140] R. Potharaju, J. Chan, L. Hu, C. Nita-Rotaru, M. Wang, L. Zhang, and N. Jain, "ConfSeer: Leveraging Customer Support Knowledge Bases for Automated Misconfiguration Detection," in *Proceedings of the 35th International Conference on Very Large Data Bases (VLDB'15)*, Aug. 2015.

[141] J. Zhang, L. Renganarayana, X. Zhang, N. Ge, V. Bala, T. Xu, and Y. Zhou, "EnCore: Exploiting System Environment and Correlation Information for Misconfiguration Detection," in *Proceedings of the 19th International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS'14)*, Mar. 2014.

[142] N. Palatin, A. Leizarowitz, A. Schuster, and R. Wolff, "Mining for Misconfigured Machines in Grid Systems," in *Proceedings of the 12th ACM SIGKDD*

International Conference on Knowledge Discovery and Data Mining (KDD'06), Aug. 2006.

[143] Q. Shao, Y. Chen, S. Tao, X. Yan, and N. Anerousis, "Efficient Ticket Routing by Resolution Sequence Mining," in *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'08)*, Aug. 2008.

[144] Q. Shao, Y. Chen, S. Tao, X. Yan, and N. Anerousis, "EasyTicket: a Ticket Routing Recommendation Engine for Enterprise Problem Resolution," in *Proceedings of the 34th International Conference on Very Large Data Bases (VLDB'08)*, Aug. 2008.

[145] A. Ikram, S. Chakraborty, S. Mitra, S. Saini, S. Bagchi, and M. Kocaoglu, "Root Cause Analysis of Failures in Microservices through Causal Discovery," in *Proceedings of The Thirty-Fifth Annual Conference on Neural Information Processing Systems (NeurIPS'22)*, Oct. 2022.

[146] Y. Gan, M. Liang, S. Dev, D. Lo, and C. Delimitrou, "Sage: Practical & Scalable ML-Driven Performance Debugging in Microservices," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'21)*, Apr. 2021.

[147] M. Li, Z. Li, K. Yin, X. Nie, W. Zhang, K. Sui, and D. Pei, "Causal Inference-Based Root Cause Analysis for Online Service Systems with Intervention Recognition," in *Proceedings of the 28th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'22)*, Aug. 2022.

[148] J. Mickens, M. Szummer, and D. Narayanan, "Snitch: Interactive Decision Trees for Troubleshooting Misconfigurations," in *Proceedings of the 2nd USENIX Workshop on Tackling Computer Systems Problems with Machine Learning Techniques (SYSML'07)*, Apr. 2007.

[149] Z. Yu, M. Ma, C. Zhang, S. Qin, Y. Kang, C. Bansal, S. Rajmohan, Y. Dang, C. Pei, D. Pei, Q. Lin, and D. Zhang, "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'24)*, 2024.

[150] E. Miehling, K. N. Ramamurthy, K. R. Varshney, M. Riemer, D. Bouneffouf, J. T. Richards, A. Dhurandhar, E. M. Daly, M. Hind, P. Sattigeri, D. Wei, A. Rawat, J. Gajcin, and W. Geyer, "Agentic AI Needs a Systems Theory," *arXiv:2503.00237*, Feb. 2025.

[151] N. Shinn, F. Cassano, E. Berman, A. Gopinath, K. Narasimhan, and S. Yao, "Reflexion: Language Agents with Verbal Reinforcement Learning," *arXiv:2303.11366*, Mar. 2023.

[152] Y. Dong, R. Mu, Y. Zhang, S. Sun, T. Zhang, C. Wu, G. Jin, Y. Qi, J. Hu, J. Meng, S. Bensalem, and X. Huang, "Safeguarding Large Language Models: A Survey," *arXiv:2406.02622*, Jun. 2024. arXiv: `2406.02622 [cs.CR]`. [Online]. Available: `https://arxiv.org/abs/2406.02622`

[153] A. Nunez, N. T. Islam, S. K. Jha, and P. Najafirad, "AutoSafeCoder: A Multi-Agent Framework for Securing LLM Code Generation through Static Analysis and Fuzz Testing," *arXiv:2409.10737*, Sep. 2024.

[154] H. Inan, K. Upasani, J. Chi, R. Rungta, K. Iyer, Y. Mao, M. Tontchev, Q. Hu, B. Fuller, D. Testuggine, and M. Khabsa, "Llama Guard: LLM-based Input-Output Safeguard for Human-AI Conversations," *arXiv:2312.06674*, Dec. 2023.

[155] *Guardrails AI*, `https://www.guardrailsai.com/`.

[156] H. Wang, C. M. Poskitt, and J. Sun, "AgentSpec: Customizable Runtime Enforcement for Safe and Reliable LLM Agents," in *Proceedings of the 48th IEEE/ACM International Conference on Software Engineering (ICSE'26)*, 2026.

[157] E. Y. Chang and L. Geng, "SagaLLM: Context Management, Validation, and Transaction Guarantees for Multi-Agent LLM Planning," *arXiv:2503.11951*, Mar. 2025. arXiv: `2503.11951 [cs.AI]`. [Online]. Available: `https://arxiv.org/abs/2503.11951`

[158] G. Li, H. Hammoud, H. Itani, D. Khizbullin, and B. Ghanem, "CAMEL: Communicative Agents for "Mind" Exploration of Large Language Model Society," in *Proceedings of The Thirty-Sixth Annual Conference on Neural Information Processing Systems (NeurIPS'23)*, Sep. 2023.

[159] S. Hong, M. Zhuge, J. Chen, X. Zheng, Y. Cheng, J. Wang, C. Zhang, Z. Wang, S. K. S. Yau, Z. Lin, L. Zhou, C. Ran, L. Xiao, C. Wu, and J. Schmidhuber, "MetaGPT: Meta Programming for A Multi-Agent Collaborative Framework," in *Proceedings of the International Conference on Machine Learning (ICML'24)*, Jan. 2024.

[160] *Swarm: Scalable infrastructure for multi-agent coordination*, `https://github.com/openai/swarm`, 2024.

[161] C. Qian, W. Liu, H. Liu, N. Chen, Y. Dang, J. Li, C. Yang, W. Chen, Y. Su, X. Cong, J. Xu, D. Li, Z. Liu, and M. Sun, "ChatDev: Communicative Agents for Software Development," in *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (ACL'24)*, Aug. 2024.

[162] T. Xu, Y.-F. Zhang, Z. Chu, S. Wang, and Q. Wen, "AI-Driven Virtual Teacher for Enhanced Educational Efficiency: Leveraging Large Pretrain Models for Autonomous Error Analysis and Correction," *arXiv:2409.09403*, Sep. 2024.

[163] Z. Zhou, Y. Lin, D. Jin, and Y. Li, "Large Language Model for Participatory Urban Planning," *arXiv:2402.17161*, Feb. 2024.

[164] J. Zhao, C. Zu, X. Hao, Y. Lu, W. He, Y. Ding, T. Gui, Q. Zhang, and X. Huang, "LONGAGENT: Achieving Question Answering for 128k-Token-Long Documents through Multi-Agent Collaboration," in *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing (EMNLP'24)*, Nov. 2024.

[165] Y. Du, S. Li, A. Torralba, J. B. Tenenbaum, and I. Mordatch, "Improving Factuality and Reasoning in Language Models through Multiagent Debate," *arXiv:2305.14325*, May 2023. arXiv: `2305.14325 [cs.CL]`. [Online]. Available: `https://arxiv.org/abs/2305.14325`

[166] T. Liang, Z. He, W. Jiao, X. Wang, Y. Wang, R. Wang, Y. Yang, S. Shi, and Z. Tu, "Encouraging Divergent Thinking in Large Language Models through Multi-Agent Debate," in *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing (EMNLP'24)*, Nov. 2024.

[167] S. Zhang, M. Yin, J. Zhang, J. Liu, Z. Han, J. Zhang, B. Li, C. Wang, H. Wang, Y. Chen, and Q. Wu, "Which Agent Causes Task Failures and When? On Automated Failure Attribution of LLM Multi-Agent Systems," in *Proceedings of the International Conference on Machine Learning (ICML'25)*, 2025.

[168] M. Z. Pan, M. Cemri, L. A. Agrawal, S. Yang, B. Chopra, R. Tiwari, K. Keutzer, A. Parameswaran, K. Ramchandran, D. Klein, J. E. Gonzalez, M. Zaharia, and I. Stoica, "Why Do Multiagent Systems Fail?" In *Proceedings of The Thirteenth International Conference on Learning Representations (ICLR'25)*, Mar. 2025. [Online]. Available: `https://openreview.net/forum?id=wM521FqPvI`

[169] J. Dobies and J. Wood, *Kubernetes Operators: Automating the Container Orchestration Platform*. O'Reilly Media, Inc., Feb. 2020.

[170] P. Ratis, "Lessons Learned Using the Operator Pattern to Build a Kubernetes Platform," in *USENIX SREcon*, Oct. 2021.

[171] S. Han, X. Hu, H. Huang, M. Jiang, and Y. Zhao, "ADBench: Anomaly detection benchmark," in *Thirty-sixth Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2022.

[172] V. Jacob, F. Song, A. Stiegler, Y. Diao, and N. Tatbul, "Anomalybench: An open benchmark for explainable anomaly detection," *CoRR*, 2020.

[173] Y. Liu, C. Pei, L. Xu, B. Chen, M. Sun, Z. Zhang, Y. Sun, S. Zhang, K. Wang, H. Zhang, et al., "OpsEval: A comprehensive task-oriented AIOps benchmark for large language models," *arXiv:2310.07637*, 2023.

[174] ChaosBlade Team, *ChaosBlade*, `https://github.com/chaosblade-io/chaosblade`, Accessed: 2024-07-08, 2019.

[175] ChaosMesh Authors, *Chaosmesh*, `https://chaos-mesh.org/`, Accessed: 2024-07-08, 2022.

[176] 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 (ICLR'24)*, 2024.

[177] Helm, *Helm: The package manager for kubernetes*, `https://helm.sh/`, 2024.

[178] L. Zheng, W.-L. Chiang, Y. Sheng, S. Zhuang, Z. Wu, Y. Zhuang, Z. Lin, Z. Li, D. Li, E. Xing, et al., "Judging llm-as-a-judge with mt-bench and chatbot arena," *Advances in Neural Information Processing Systems (NeurIPS'24)*, 2024.

[179] PingCAP, *Tidb operator*, `https://github.com/pingcap/tidb-operator`, 2025.

[180] OpenFeature, *Flagd*, `https://flagd.dev/`, 2025.

[181] Jaeger Authors, *Jaeger: Open source, end-to-end distributed tracing*, `https://www.jaegertracing.io`, 2024.

[182] Elasticsearch, *Lightweight shipper for logs*, `https://www.elastic.co/beats/filebeat`, 2024.

[183] Elasticsearch, *Centralize, transform & stash your data*, `https://www.elastic.co/logstash`, 2024.

[184] Prometheus Authors, *Prometheus*, `https://prometheus.io`, 2024.

[185] M. A. Inam, Y. Chen, A. Goyal, J. Liu, J. Mink, N. Michael, S. Gaur, A. Bates, and W. U. Hassan, "Sok: History is a vast early warning system: Auditing the provenance of system intrusions," in *2023 IEEE Symposium on Security and Privacy (S&P'23)*, 2023.

[186] J. Zeng, X. Wang, J. Liu, Y. Chen, Z. Liang, T.-S. Chua, and Z. L. Chua, "Shadewatcher: Recommendation-guided cyber threat analysis using system audit records," in *2022 IEEE Symposium on Security and Privacy (S&P'22)*, 2022.

[187] J. Zeng, Z. L. Chua, Y. Chen, K. Ji, Z. Liang, and J. Mao, "Watson: Abstracting behaviors from audit logs via aggregation of contextual semantics.," in *Network and Distributed System Security Symposium (NDSS'21)*, 2021.

[188] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat, et al., "GPT-4 technical report," *arXiv:2303.08774*, 2023.

[189] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, and Y. Cao, "ReAct: Synergizing reasoning and acting in language models," in *International Conference on Learning Representations (ICLR'23)*, 2023.

[190] U. Çetin and M. Tasgin, "Anomaly detection with multivariate K-sigma score using Monte Carlo," in *2020 5th International Conference on Computer Science and Engineering*, 2020.

[191] L. Wang, N. Zhao, J. Chen, P. Li, W. Zhang, and K. Sui, "Root-cause metric location for microservice systems via log anomaly detection," in *2020 IEEE international conference on web services (ICWS'20)*, 2020.

[192] C. Hou, T. Jia, Y. Wu, Y. Li, and J. Han, "Diagnosing performance issues in microservices with heterogeneous data source," in *2021 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking*, 2021.