Demystifying the Fight Against Complexity: A Comprehensive Study of Live Debugging Activities in Production Cloud Systems

P. C. Sruthi
Purdue University
West Lafayette, Indiana, USA
psruthi@purdue.edu

Zinan Guo*
Purdue University
West Lafayette, Indiana, USA
guo663@purdue.edu

Deming Chu†
Purdue University
West Lafayette, Indiana, USA
chu292@purdue.edu

Zhengyan Chen‡
University of Georgia
Athens, Georgia, USA
zychen@uga.edu

Yongle Zhang
Purdue University
West Lafayette, Indiana, USA
yonglezh@purdue.edu

ABSTRACT

Debugging in production cloud systems (or live debugging) is a critical yet challenging task for on-call developers due to the financial impact of cloud service downtime and the inherent complexity of cloud systems. Unfortunately, *how* debugging is performed, and the unique challenges faced in the production cloud environment have not been investigated in detail.

In this paper, we perform the first fine-grained, observational study of 93 real-world debugging experiences of production cloud failures in 15 widely adopted *open-source distributed systems* including distributed storage systems, databases, computing frameworks, message passing systems, and container orchestration systems. We examine each debugging experience with a fine-grained lens and categorize over 1700 debugging steps across all incidents. Our study provides a detailed picture of how developers perform various diagnosis activities including failure reproduction, anomaly analysis, program analysis, hypothesis formulation, information collection and online experiments.

Highlights of our study include: (1) Analyses of the taxonomies and distributions of both live debugging activities and the underlying reasons for hypothesis forking, which confirm the presence of expert debugging strategies in production cloud systems, and offer insights to guide the training of novice developers and the development of tools that emulate expert behavior. (2) The identification of the primary challenge in anomaly detection (or, observability) for end-to-end debugging: the collection of system-specific data (17.1% of data collected). In comparison, nearly all (96%) invariants utilized to detect anomalies are already present in existing monitoring tools. (3) The identification of the importance of online interventions (i.e., in-production experiments that alter system execution) for live debugging – they are performed as frequently as information collection – with an investigation of different types of interventions and challenges. (4) An examination of novel debugging techniques developers utilized to overcome debugging challenges inherent to or amplified in cloud systems, which offer insights for the development of enhanced debugging tools.

CCS CONCEPTS

• **Software and its engineering** → **Distributed systems organizing principles**; **Software testing and debugging**; • **Computer systems organization** → **Cloud computing**.

KEYWORDS

Debugging, Live Debugging, Distributed Systems, Cloud Computing, Observability, Study

ACM Reference Format:
P. C. Sruthi, Zinan Guo, Deming Chu, Zhengyan Chen, and Yongle Zhang. 2024. Demystifying the Fight Against Complexity: A Comprehensive Study of Live Debugging Activities in Production Cloud Systems . In *ACM Symposium on Cloud Computing (SoCC '24), November 20–22, 2024, Redmond, WA, USA.* ACM, New York, NY, USA, 20 pages. https://doi.org/10.1145/3698038.3698568

*Currently at Ernst & Young
†Work done while enrolled at Tongji University, interning at Purdue
‡Work done while enrolled at Peking University, interning at Purdue

2
SoCC '24, November 20–22, 2024, Redmond, WA, USA

ACM ISBN 979-8-4007-1286-9/24/11.
https://doi.org/10.1145/3698038.3698568

1 INTRODUCTION

Failure diagnosis and resolution in production cloud systems (or *live debugging*) is notoriously difficult and time-consuming due to their inherent complexity: developers typically need to reason about tens of thousands of concurrent processes executed on heterogeneous hardware devices [12]. This time-consuming procedure could delay failure resolution, extend service disruption, and result in customer frustration; we collected a random sample of public incidents from top cloud vendors and found that 62.5% of the duration of service disruption was spent on debugging (§ 2).

Many researchers [43, 49, 57, 72, 73, 90, 99] have investigated different aspects of debugging in single-node applications. Yet, *live debugging* in production cloud systems remains under-examined. To the best of our knowledge, only three recent works [48, 55, 114] have studied a few aspects of debugging in cloud systems, but they did not study how live debugging is performed: researchers from Microsoft studied common root causes of production incidents and corresponding detection and mitigation strategies [55] in Microsoft Azure, as well as the root cause labeling given a postmortem report [48], but they did not analyze how the root causes are diagnosed. Zhou et al. [114] focuses on evaluating the effectiveness of information (i.e., logs and traces) utilized in diagnosing microservice failures. None of them provide a detailed understanding of how developers perform various debugging activities, such as failure reproduction, anomaly detection, program analysis, hypothesis formulation, information collection and online experiments.

In this work, we perform the first study of *fine-grained* debugging activities in *end-to-end, real-world* debugging experiences in *production cloud systems*. In particular, we collect and analyze 93 detailed documents (e.g., blog posts) of developers' failure diagnosis experiences in production environments of 15 widely deployed *open-source* distributed systems (e.g., Cassandra [16], Spark [106], and Kubernetes [23]). In contrast to bug reports and postmortem reports which usually document the root cause and fix of a failure, we study the documents that *rewind* every diagnosis activity performed by the developer for a production cloud incident and how each activity changes their hypotheses. We divide each document into fine-grained steps according to different activities performed by the developer. This results in 1740 steps across all cases, with an average of approximately 19 steps per case. To ensure the quality of our analysis, we conservatively exclude documents lacking detailed debugging steps and clear descriptions of hypotheses.

Our goal is to provide a comprehensive, fine-grained understanding of various debugging activities and respective challenges in production cloud environments, providing guidance for potential solutions. We focus on the following research questions:

RQ 1. **Debugging Activities**. What debugging activities do developers perform in production cloud systems? How are they performed – what strategies and heuristics do developers adopt in each activity?

RQ 2. **Challenges**. What are the challenges faced in each diagnosis activity? What are the challenges introduced by or magnified in production cloud systems? What debugging strategies do developers adopt to overcome these challenges?

RQ 3. **Reasons for Forking Hypotheses**. Given that the core of debugging is to formulate and validate hypotheses, what causes developers to formulate multiple hypotheses?

This paper makes the following contributions:

- We present the first study of fine-grained debugging activities in end-to-end, real-world debugging experiences in production cloud systems from open-source distributed systems. Our analysis results in a set of structured documentations and *causal graphs* (§ 2) of debugging experiences. Our study reveals over a dozen informative findings with concrete implications for new research directions to assist debugging production cloud failures. All of our data is available online[1].
- Our study reveals taxonomies of both live debugging activities and strategies (Table 4), as well as underlying reasons for hypothesis forking (Table 7) in the production cloud environment. Our analysis explains expert debugging strategies and offers insights on how novice developers and debugging tools may emulate such strategies. For instance, we find that expert developers tend to (in 80.4% of cases) formulate multiple hypotheses by enumerating **immediate** causes in a chosen model or considering the correctness of the model itself. Our analysis shows how hypothesis forking is performed (§ 4.2), as well as how developers limit the scope of their reasoning (§ 4.3) so that the number of immediate causes is tractable using a variety of strategies (§ 4.4, § 6).
- To the best of our knowledge, our study is the first to quantify the requirements of effective anomaly detection for **end-to-end** debugging within production cloud environments. Despite existing monitoring tools incorporating most (96%) necessary invariants, the collection of system-specific information (17.1%) remains a primary obstacle. In addition, we investigate potential strategies to automatically collect system-specific information and provide a taxonomy of unconventional invariants.
- Our study reveals that online interventions – in-production experiments that alter system execution – are critical for

[1] https://github.com/zlab-purdue/socc-24-debugging-study

debugging purposes, demonstrating their prevalence (11.3%) to be on par with information collection (12%). We further discuss the potential to inject knobs for intervention automatically, as well as functionality- and performance-preserving guarantees to mitigate risks associated with such interventions.
- Our study reveals novel debugging techniques developers utilized to overcome debugging challenges inherent to or amplified in cloud systems. These techniques include leveraging concurrency indicators (unique anomalies that indicate concurrency bugs), educated guesses of execution frequency, and exploiting symmetry in node roles. These findings offer valuable insights for the development of enhanced debugging tools tailored to the specific demands of cloud systems.

We describe our study *methodology* in § 2, before presenting an overview of studied *debugging activities and strategies* in § 3. We investigate how developers formulate hypotheses by *explanation* and *exploration* and corresponding challenges in § 4. § 5 details our findings about how developers perform *experiments*. We discuss challenges inherent to or amplified by cloud systems, as well as *novel debugging strategies* to overcome them in § 6. We explain related work in § 7 and conclude in § 8.

2 METHODOLOGY

To understand the cost of debugging in production, we first performed a pilot study by randomly sampling 20 public postmortem reports from Google Cloud Platform (GCP) [21], as they typically record timestamps indicating when the failure is detected, debugged, and recovered from. Our study shows that on average, 10.8% of incidents' duration was spent on detection, 62.5% on debugging, and 29.2% on deployment and recovery, which highlights the need to reduce debugging time for production failures. Although these reports offer valuable insights into the high-level pattern of failure resolution processes in production cloud environments, they lack detailed descriptions of debugging steps that are essential for our study, such as how developers formulate hypotheses and conduct experiments.

	avg. min	avg. %
Detection	29.8	10.8%
Debugging	169	62.5%
Recovery	37.2	29.2%
Total	232.8	

Table 1: Incident timeline

2.1 Case Collection

To ensure that we collected representative and high-quality debugging experiences, we adopted the following case collection methodology.

System	#	%	System	#	%
Cassandra	2	2.1	Kubernetes	9	9.7
FastDFS	1	1.1	MongoDB	6	6.5
Flink	2	2.1	Redis	4	4.3
HBase	6	6.5	Spark	12	12.9
HDFS	14	15.1	TiDB	2	2.1
Hive	6	6.5	Yarn	3	3.2
Kafka	15	16.1	ZooKeeper	10	10.8
etcd	1	1.1			
			Total	**93**	**100.0**

Table 2: Distribution of cases analyzed

As shown in Table 2, we chose a large variety of widely-deployed, open-source distributed systems, including distributed storage systems (HDFS [22], FastDFS [105]), distributed databases and key-value stores (Cassandra [16], HBase [18], Hive [19], ZooKeeper [64], etcd [28], Redis [35], MongoDB [32], TiDB [36]), distributed computing frameworks (Spark [27]), a distributed resource manager (YARN [98]), distributed message passing services (Flink [17], Kafka [20]), and a container orchestration framework (Kubernetes [23]).

For each of the above systems, we scraped the Internet with keywords such as "debugging", "diagnosis", "troubleshooting", and "postmortem", combined with (1) the system's name, or (2) bug ticket identifiers on their issue trackers. Our scraping resulted in over *63,000* webpages. To ensure the quality of the studied documents, we then filtered out webpages without detailed steps for diagnosing a failure (e.g., posts that only describe a failure's root cause and fix). To ensure these debugging experiences happened in production, we further filtered out the documents which had no keywords indicating the failure was encountered in production (e.g., "production", "customer"). Finally, for each blog post in the remaining set, we found additional high-quality posts by navigating through technical blog posts from the same author (85% of our documents have a distinct author), as well as recommendations on the sites where the posts were hosted. This resulted in a total of 93 posts, that we analyzed in detail.

Case Quality: The conservative nature of our case selection process resulted in a limited proportion (7.8%) of debugging steps where the rationale for hypothesis development was not explicitly documented.

2.2 Case Characteristics

Root Cause: As shown in Table 3, our collected cases have diverse root causes. **Temporal Distribution:** 81% of these cases happened within the past five years.

Though not every document mentions the scale of their cloud and the developer's profile, many mentioned that the

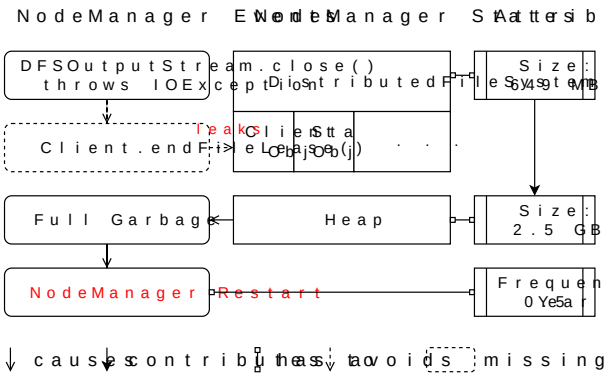

Figure 1: Causal Graph for *hdfs-5* [7]

failures happened on large-scale clouds and were diagnosed by experienced developers.

Cloud Scale: For example, *hdfs-8* [6] happened on a system with 50 billion transactions per day and 8PB data, and *kafka-6* [15] happened on a system powering over 250 applications.

Root Cause	Case (%)
Software Bug	53.4
Misconfiguration	35.0
Incorrect Input	6.8
Hardware Fault	3.9
Changed Dependency	1.0

Table 3: Root cause distribution

In addition, though root caused in one system, many resulting failures were diagnosed across multiple interdependent systems (e.g., Hive, HDFS, Yarn [5, 10, 31]). **Developer Profile:** Many authors of the posts were engineers with decades of experience from mid- to large-scale cluster companies, including Apache Project Management Committee members [1, 2, 9, 11, 14, 26]. Additionally, some posts are detailed accounts of incidents from companies with billion dollar valuations such as PagerDuty [34], a leader in monitoring and incident management and Gojek [30], a technology service provider, written by their own engineers. **Debugging Duration:** Failure cases with debugging duration were documented as taking hours to months to diagnose.

2.3 Case Analysis Method

To facilitate a fine-grained analysis of debugging activities, for each post, we organized developers' actions as a series of fine-grained *steps*, falling into the categories of activities specified in § 3. To study the relationship between debugging activities and hypotheses, we constructed a *causal graph* representing developers' hypotheses that resulted from each step, with each node representing an event or state and edges representing causal relationships. Nodes can be associated with attributes representing a quantity that needs to be explained, such as the size of a memory snapshot.

We provide an overview of our analysis with a simplified real-world example – *hdfs-5* [7], a failure caused by memory leaks in an HDFS/YARN deployment. Figure 1 shows the causal graph representing developers' final hypothesis of the failure. We separate developers' diagnosis procedure into the following steps:

Step 1 (categorized as Exploration/Anomaly Detection): The developer discovers the symptom of the failure: an occurrence of full Garbage Collection (full GC) on a NodeManager (⑧) causing a restart of the process (⑨). This phenomenon occurs every 6 months (⑩) and makes offline reproduction infeasible due to the long execution.

Step 2 (categorized as Information Collection): Fortunately, the failure happens on other nodes in the cluster. The developer captures the memory snapshot (⑦, ⑥) of a machine that experiences full GC.

Step 3 (categorized as Explanation of Quantity Contributions): They analyze the snapshot to find the objects that contribute to the high memory before full GC, and find a large number of DistributedFileSystem objects (③, ⑤).

Step 4 (categorized as Exploration (through Correlation) and Explanation (of State Transitions)): They hypothesize that there is a memory leak of DistributedFileSystem objects or their referenced objects such as Client objects (④). They inspect the source code locations cleaning up these objects with a heuristic: since it takes six months to trigger full GC, the leak should be due to execution paths that occur infrequently, such as exception code paths. They find that a close method fails to release references of Client objects correctly when exceptions occur (①, ②).

Step 5 (categorized as Consultation through Online Search): Finally, they search the system's JIRA issue tracker for relevant issues, and find a confirmation of their hypothesis.

The causal graph captures the entire hypothesis development process and is particularly useful for studying the relationship between various debugging activities and hypotheses. For example, we study what causes developers to fork hypotheses (§ 4.2) and the distribution of the challenge of collecting system-specific data in the development of hypotheses (§ 5.1.1) through queries on the causal graphs.

2.4 Threats to Validity

Our collected set of developers' documentation may not contain complete and unbiased descriptions of their diagnosis processes. For example, the developer omitted a reasoning mistake in their diagnosis for *hdfs-6* [1], which they acknowledged. Additionally, being individual accounts, debugging experience posts' language convey differing levels of detail. We attempt to account for the bias and incompleteness by filtering out those posts that do not provide detailed debugging procedures, where the reasoning is not thorough. Developers

Activity	Mechanism	
Explanation (640)	Model Analysis (640)	State Transition
		Quantity Contribution
Exploration (452)	Correlation (303)	Locality
		Execution Comparison
	Anomaly Detection (149)	Event Anomaly
		State Anomaly
		Source Code Anomaly
Experimentation (438)	Information Collection (209)	Instrumentation
		Probing
	Online Intervention (196)	
	Offline Reproduction (33)	
Miscellaneous (459)	Consultation (103)	Internet Search
		Consulting Experts
	Past Experience (9) / Unspecified (136)	
	Validation (132) / Invalidation (159)	

Table 4: Overview of diagnosis activities (with # of steps)[2]

are more likely to document and publish challenging debugging experiences and ignore trivial ones. Fortunately, this aligns with the goal of our study – to observe and analyze challenges in debugging. Our collected debugging experiences are public posts of diagnosing failures in open-source distributed systems. Internal documentation of debugging experiences about proprietary systems could reveal different characteristics. Unfortunately, publicly available postmortems from companies do not usually contain enough detail for analysis. In addition, our sampling and filtering are limited by our heuristics. We account for observer error during analysis by having multiple inspectors investigate each case, and any disagreement is discussed to reach a consensus.

3 DEBUGGING ACTIVITY OVERVIEW

Our analysis reveals a large variety of debugging activities and strategies performed in production clouds. In this section, we give an overview of the categorization (Table 4).

The core task of debugging is to find the *hypothesis* that adequately illustrates the sequence of events that leads to the failure symptom. This allows the developer to modify the system execution in such a way that the effects of the failure can be mitigated, and completely prevented in future executions. As shown in Table 4, developers develop their hypotheses mainly through *explanation* and *exploration*, and verify them through *experimentation*.

Explanation: Explanation (or *model analysis*) refers to when the developer analyzes source code or state machine style mental models to identify causes of events and states or causal links between them. Causal relationships adopted by developers include state transition causality [79], and quantity contribution.

Exploration: Developers often search relevant sources of information for clues that could indicate possible causes for the failure, without having a concrete causal relationship in mind (the possible causes are often examined later with *explanation*). Such exploration is typically performed with a combination of *correlation* to identify potential causes following locality or comparing multiple executions, and *anomaly detection* over events, program states, and source code.

Experimentation: To facilitate both exploration and explanation, developers perform a variety of experiments, which fall under three categories: *information collection*, whereby the developer probes system state and functionality or traces execution by instrumenting the system; *online interventions* that modify system execution in the production environment; and *offline reproduction* of failures.

Miscellaneous: Developers also perform *consultation* – online search through search engines or on issue trackers, and consulting experts. If the developer comes up with a hypothesis without explaining how or states that they had observed a similar symptom previously, we label the step with *unspecified* and *past experience* respectively. We have omitted a consultation section in this paper, as our analysis did not yield significantly new findings compared to existing research. Lastly, developers *validate* and *invalidate* their hypotheses through repeatedly performing all the aforementioned activities. These steps allow us to perform the analysis on reasons for hypothesis forking (§ 4.2).

4 HYPOTHESIS FORMULATION

> *"When you have eliminated all which is impossible, whatever remains, however improbable, must be the truth."*
>
> — Arthur Conan Doyle, *The Adventure of the Blanched Soldier*

This Sherlock Holmes quote captures the essence of hypothesis development strategies observed in this study – expert developers consider possible causes exhaustively, and validate them rigorously.

In this section, we first investigate the mechanisms developers employ for explanation and exploration. We then analyze their reasoning strategies which determine the breadth

[2]Developers occasionally mention multiple activities in the same sentence. In these cases, we did not separate the sentence into distinct steps for each activity, but labeled the step with multiple activities. This leads to steps belonging to multiple categories.

and depth of explanations and explorations, revealing two key strategies: (1) a broad consideration of potential causes, and (2) a tendency to focus explanation on a limited scope informed by experimental evidence. We identify general challenges in explanation and exploration, including the broad range of possible causes, as well as unique invariants and system-specific data required for effective anomaly detection. Our data also reveals challenges specific to cloud systems, which we discuss in detail in subsequent sections.

4.1 Mechanisms of Hypothesis Formulation

In this section, we present the mechanisms of hypothesis formulation and their distribution. While largely intuitive, these mechanisms are essential for obtaining a holistic view of the entire debugging process.

4.1.1 Explanation. A major portion (36.8%) of a developer's debugging process is in the *explanation* of events and states. The mechanism of explanation is characterized by the type of causal relationship utilized in the explanation. In particular, we observe two types of causal relationships, namely *state transitions* and *quantity contributions*, in the proportions shown in Table 5. State transitions (or events) denote transitions between states in a state machine model of the system. They include both source code level transitions such as control flow and data flow, as well as state transitions at a higher level of abstraction. Quantity contribution steps involve enumerating what contributes to a quantity metric such as memory size, execution time, or other program quantities.

Causal Relationship	(%)
State Transition	70.8
Quantity Contribution	29.2

Table 5: Explanation Steps

Correlation Mechanism		(%)
Locality	Spatial Locality	60.7
	Temporal Locality	27.1
Execution Comparison	Common Denominator	9.2
	Delta Identification	9.6

Table 6: Exploration step distribution. Some steps may fall under multiple categories.

4.1.2 Exploration. Another major (26% of all steps) approach to identify potential causes for events and states is *exploration*: exploring relevant sources of information (often within logs and metrics of dependent components) for anomalies that could be correlated with and possibly causing the failure. We examine the mechanism of correlation here, and defer a detailed discussion of anomaly detection to § 4.4 as the primary challenge of anomaly detection lies in the policy of selecting relevant invariants and data.

As shown in Table 6, to correlate anomalies with the failure, developers often search for anomalies in the system with the intuitive assumption that anomalous behavior located nearby (following **Locality**) may be the cause. *Spatial locality* refers to when developers examine events with a small spatial distance measured by the number of intermediate dependent components (e.g., nodes, systems), to the effect. We distinguish this from pure explanation, since developers do not reason about a specific state transition between the components but rather leverage their dependencies for correlation. *Temporal locality* refers to searching for abnormalities in recent logs or metrics close to when the failure happened (i.e., within a small temporal distance). Developers also compare failing and successful executions to find anomalies relevant to a failure (**Execution Comparison**). We label a step with *common denominator* if it focuses on identifying a shared characteristic of multiple execution paths, pieces of data, or nodes, and we label a step with *delta identification* if it focuses on comparing successful and failing executions to identify any differences between them.

4.2 Hypothesis Forking

In this section, we analyze one aspect of developers' debugging strategy – how **broadly** they reason in explanation and exploration, by answering the question: what factors contribute to the number of hypotheses that developers form? In particular, we examine the steps where developers form multiple hypotheses – a process we refer to as *hypothesis forking* because it forks multiple branches in our causal graphs (§ 2) – and analyze the underlying reasons.

Table 7 shows all the reasons we observed for hypothesis forking with their definitions. These reasons reflect the classes of hypotheses that developers consider to explain a target when debugging – the developers then exhaustively enumerate possible causes in the class until the correct one is identified.

> Finding 1. *In 80.4% of cases, developers formulated multiple hypotheses by enumerating the immediate causes of a target in a model or suspecting the correctness of the model itself.*

For a given target state or event to be explained, developers choose a model (an abstract state machine model or source code model), and enumerate all possible immediate causes with respect to that model. For example, the Data Flow category refers to immediate assignment statements to a state in the source code, Partial Failure Modes refer to the different failure condition checks (e.g., connectivity check, disk size

Hypothesis Forking Reasons			(%)
Blackbox Reasoning	Enumerating faults in (1) dependent components, (2) environment, (3) input/load, (4) config, or (5) code.		44.6
Multiple Execution Paths	Data Flow	Enumerating possible dataflows.	3.6
	Partial Failure Modes	Enumerating partial failure conditions (e.g., partitioned, disk full) for one or multiple nodes.	4.5
	Failure Recovery	Enumerating (1) recovery executed but failed or (2) recovery did not happen.	5.4
	Missing Event	Enumerating occurrence locations of a missing event and conditions leading to its omission.	7.1
	Opaque Error	Enumerating intractable immediate causes for an error (e.g., who sent a SIGKILL).	1.8
	Network Messages	Enumerating faults in (1) sending, (2) receipt, or (3) delivery of messages.	2.7
	Concurrency	Enumerating process/thread interleavings.	0.9
	Other	Enumerating multiple relatively long execution paths leading to a state/event.	17.0
Slow Performance	Channel Delay	Enumerating (1) saturated bandwidth, (2) broken channel, or (3) delay in endpoint.	2.7
	Operation Delay	Enumerating delay due to (1) slow single operation or (2) too many operations.	0.9
	Contributors	Enumerating which subset in a sequence of operations contributes most to the delay.	7.1
	Timeout Limit	Enumerating (1) performance should be optimized or (2) timeout limit should be increased.	1.8
Large Quantity	Addition/Removal	Enumerating (1) frequent addition or (2) infrequent removal causing the large quantity.	5.4
	Quantity Limit	Enumerating (1) some quantity should be reduced or (2) the quantity limit should be increased.	5.4
	Contributors	Enumerating which subset of quantity is abnormally large.	3.6
Dynamism	Reasoning about dynamic network/cluster configuration and component dependency in microservices.		1.8
Irrelevant Anomaly	Correlating an irrelevant anomaly (i.e., noise) with the failure.		6.2
Incorrect Model	Reasoning with incorrect specification of underlying components.		1.8
(In)Validation Mistake	Invalidated a hypothesis incorrectly that was later proven correct (or vice versa).		0.9

Table 7: Reasons for the formation of multiple hypotheses. A single target may have multiple reasons for branching (e.g., if they considered different models). Reasons inherent to or amplified by cloud systems are bolded.

check) returning the same error (e.g., bad node) in source code, Concurrency refers to the distinct possible interleavings around the critical section. These immediate causes can be obtained mechanically (and can be systematically taught to a novice developer) by choosing the appropriate model and tracking the state transitions. Though these immediate causes could in theory be difficult to analyze due to the large and sometimes unbounded number of enumerations, we find that developers generally limit the scope of their reasoning until the number of immediate causes is tractable, at which point they explore them exhaustively. Strategies developers use to limit the scope and scale are discussed in § 4.3 and § 6. We illustrate a subset of the reasons for forking with examples here.

hdfs-6 [1] is an example in which the developer enumerates **Partial Failure Modes** of nodes in a cluster. As shown in Figure 2, developers are diagnosing why an HDFS NameNode could not find a suitable DataNode to replicate its data (i.e., `chooseTargets()` returns an empty list). At lines 7-9, various failure conditions are checked for each DataNode, such as unavailability, limited node capacity, and overload. The developer formulates hypotheses for **all** of these possible failure conditions, and validates them by manually checking these failure conditions *in situ*.

An extreme example is an **Opaque Error**, in which an error appears to be "opaque" to a developer due to an intractable number of immediate static or dynamic causes. In

```
1 Node[] chooseTargets(int nReplicas) {
2   Node[] targets;
3   while (nReplicas > 0) { // replicas needed
4     Node node = chooseNextRandom();
5     if (node == NULL) // checked all nodes
6       break;
7     if (node.space < limit
8     || !node.isAvailable()
9     || node.load > threshold)
10      continue; // exclude a ``bad'' node
11    targets.add(node);
12    nReplicas--;
13  }
14  return targets;
15 }
```

Figure 2: Simplified code snippet for *hdfs-6* [1]

spark-1 [81], the developer was explaining an InterruptedException, which is an exception thrown by threads when they are interrupted by other threads in Java. In this case, the number of possibilities appeared intractable to the developer due to the large number of Java threads (jobs) from different services on the servers running Spark. It took more than a week for the developer to exhaustively analyze anomalies and identify the error-inducing thread in YARN.

Developers also reason exhaustively about possible mistakes in their reasoning, such as using an **Incorrect Model** and **(In)Validation Mistakes**. For example, in *hdfs-4* [14],

the developer was debugging a NullPointerException of an object retrieved from a list. They first assumed the list data structure was thread-safe, but later refuted this model after exhaustively confirming that all insertions into the list are non-null.

> Lesson 1. *The key to successful debugging is the exhaustive enumeration of all mutually exclusive immediate causes in a carefully chosen model.*

As the developer in *kube-6* [82] says, "While debugging any issue, all components across the stack should be suspects. Ruling out EKS as a possible culprit led to us wasting a lot of valuable time." Any ignorance of a possible cause could lead to a significant cost of time. Though such a breadth-first-search style of debugging has been discussed before [99] in the context of test failure debugging, we show that it is critical in debugging production cloud failures and provide a taxonomy in Table 7 as to *how* Lesson 1 may be realized at the scale of cloud failures and used to guide the training of novice developers, as well as the development of debugging tools.

In addition, the reasons for hypothesis forking reflect the challenges for debugging. Though many of the listed reasons have been investigated (e.g., missing events [107, 108]), many challenges unique to or amplified by cloud environments (in bold in Table 7) persist. We discuss developers' novel debugging strategies to address them in § 6.

4.3 Scale of Explanation

Exhaustively enumerating all possible causes, especially in explanation using model analysis, is difficult, as evidenced by extreme examples such as opaque errors. In this section, we analyze another aspect of developers' debugging strategy – what is the **scale** of explanation performed by developers? This encompasses the scope of source code examined and the depth of causal relationships investigated.

We performed two analyses: (1) in code-level explanation steps, we analyzed how much source code was used for reasoning in each case; (2) in explanation steps at a higher level of abstraction, we counted the number of causal links in each explanation step (e.g., causal links formed by RPCs, cross-component data flow) by querying our data using an LLM tool [29], which we then cross-verified manually.

In the first analysis, we found that developers reasoned about, on average 95 lines of code per case (that they mentioned in their experiences). In the second analysis, we found that developers reasoned about a few transitions (avg. 3.5 per step) in every explanation step, before switching to other debugging activities (developers often resort to anomaly detection for hints, or perform experiments to validate their hypotheses).

> Lesson 2. *Frequent verification of hypotheses is the key to scalable yet systematic debugging.*

The results of our analyses are indicative of the phenomenon that developers perform verification of hypotheses frequently to keep their reasoning within a tractable scope, though we acknowledge that our analysis methods are biased by the developers' reporting. Frequent verification [58] has been discussed as an expert debugging strategy in the test debugging context, however, our analysis is the first to confirm its existence in live debugging of production cloud failures.

4.4 Anomaly Detection

> *"The little things are infinitely the most important."*
>
> — Arthur Conan Doyle, *A Case of Identity*

Existing research on anomaly detection in cloud systems has predominantly focused on failure detection [92]. To the best of our knowledge, our study provides the first analysis of anomaly detection for the purpose of end-to-end debugging, which includes both anomalies utilized for failure detection and anomalies developers collected adaptively for root cause localization and repair.

Developers detect anomalies by checking collected data against mental models of the correct behavior of the system, according to *invariants*. In this section, we examine the *data* and *invariants* used in anomaly detection. To understand the challenges in anomaly detection, we categorize data into *system-agnostic* (e.g., metrics such as load, CPU utilization, or the presence of error logs) and *system-specific* (e.g., values concerning system-specific data structures in memory) data and invariants as *conventional* and *unconventional* invariants depending on whether popular monitoring systems such as Prometheus [85] and DataDog [46] implement them by default (e.g., threshold-based anomaly detection, simple time series analyses). For example, *zoo-12* [37] involved the occurrence of a high number of Zookeeper znodes [38] – a traditional invariant (threshold based anomaly detection) broken on system-specific data. In *hive-4* [5], the developer noticed the anomaly of one process being alive, and another process being dead (where it would not have been anomalous for either of these to have been true independently) – an example of an unconventional invariant observed on system-agnostic data (process liveness).

As shown in Table 8, developers look for anomalies in a variety of sources of information. While anomalies are mostly (82.9%) observed in system-agnostic data such as logs (34.7%) and monitoring metrics (29.7%), about one-sixth (17.1%) are system-specific internal state and events, which are harder to collect.

Category	Data Compared	(%)
Events	Errors	13.2
	Exceptions	10.1
	Warnings	3.7
	Crashes	0.9
	Log Frequency	3.1
	Other Logs	4.6
	Locking Behavior	0.9
	Other System Specific Events	7.6
State	Config	2.4
	Input/Input-metadata	1.5
	Process/Thread State	8.9
	System Specific State	9.5
	Metrics	29.7
	Output/Output-metadata	4.0

Table 8: Categories of data compared for exploration

Invariant Type	Data	Invariant Rule	(%)
Runtime Invariant	State	Existence	2.5
		Relation	6.3
		Expected Value	36.3
	Events	Unexpected Events	34.8
		Missing Events	1.8
Static Invariant (in source code)			3.5

Table 9: Invariants distribution (over all exploration steps)

Unconventional Inv.	Examples
Static Invariant	Missing statement, unintended APIs.
Process State Relation	Two processes need to be alive simultaneously.
Data Consistency	Data/metadata consistency.
Numerical Delta	Abnormal delta of two variables.
Temporal Distance	Abnormally distant/close in time.
Equality	Two variables are equal/unequal.

Table 10: Unconventional Invariants

Invariant	Data	
	System-Agnostic	System-Specific
Conventional	79.5%	16.5%
Unconventional	3.4%	0.6%

Table 11: Cross section of anomaly data and invariants

Table 9 shows our categorization of invariants: (1) runtime invariants including existence and expected values of states, the occurrence of events (a missing event is an anomaly, as is an unexpected event such as an exception), as well as more complex relationships between states and events; (2) static invariants refer to abnormal patterns in the source code (i.e., the source code showed incorrect behavior according to the developer's own system model). We consider the existence and expected values of states, the occurrence of specific events, and time series patterns as *conventional* invariants, since they are default metrics and detection strategies in popular monitoring frameworks. On the other hand, static invariants, and a subset of runtime invariants, are considered *unconventional* (Table 10).

We performed a cross-analysis of anomaly data and the runtime invariants (without static invariants) as shown in Table 11.

> Finding 2. *The majority (79.5%) of anomalies can be detected with conventional invariants and system-agnostic data. The remainder (20.5%) require system-specific data or unconventional invariants.*

Implications: Existing anomaly detection techniques should be able to automate the majority (79.5%) of anomaly detection performed by the developers. On the other hand, the remainder (20.5%) requires system-specific data or unconventional invariants, rendering widely adopted anomaly detection techniques insufficient. Among the unconventional invariants, *Process State Relation* and *Data Consistency* present a lower barrier to adoption due to the well-defined scope of the required data. Conversely, *Numerical Delta*, *Equality*, and *Temporal Distance* pose a greater challenge due to the pervasive nature of the data involved. *Static Invariants* are also challenging due to the many varied suspicious code patterns. Since the system-specific data utilized in exploration are collected through *Information Collection* in experimentation, we discuss the potential to adaptively collect this data with low overhead, in § 5.

5 EXPERIMENTATION

Developers perform experiments, namely, information collection, online interventions in the production environment, and offline reproductions, to verify and develop their hypotheses.

5.1 Information Collection

Developers collect additional information when existing runtime information (e.g., logs, metrics) are insufficient to diagnose the failure. An Information Collection (IC) step is one that involves *instrumentation* to trace execution of the program as it re-executes failure paths, or *interactive probing* of the system to collect information while the system is in a buggy state. We present the collection methods and information collected in Table 12a and Table 12b.

	Action	Steps(%)
Probing	State	64.7
	Command	13.5
Instrument	Log	3.9
	Debugger	1.0
	Sniffing	2.9
	Log Level	4.3
	Tracing	5.3
	Command	2.4
	Watchpoint	1.9

(a) IC Actions

Info.	(%)
Internal Data	23.0
Process Status	20.6
Command Result	13.9
Stack Trace	14.8
Memory Dump	9.6
Network	9.6
Filesystem	3.8
CPU	0.5
Input	1.0
Output	1.9

(b) IC data categories

Table 12: Information Collection characteristics

5.1.1 Collection of System-Specific Data. We are particularly interested in how to automatically collect system-specific data required in exploration. The first question we seek to answer is – as system-specific data accounts for a minority of data used in exploration, how does it impact the debugging process? Contrary to our initial assumption that system-

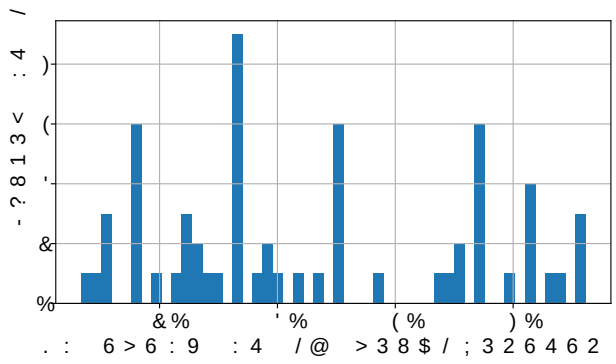

Figure 3: Distribution of system-specific information collection.

specific data collection primarily impacts the final stages of hypothesis formulation, our analysis reveals a different pattern. We investigated the distribution of system-specific data collection within the causal graph of explanation steps, examining its **position** in each hypothesis subgraph. As depicted in Figure 3, the collection of such data is dispersed throughout the entire hypothesis development process. This finding suggests that the inability to gather system-specific data can hinder debugging efforts at the earliest stages.

hbase-1 [94] (code snippet in Figure 4) shows an example of system-specific information collection. The failure happened during a read over network: the developer observed a large read being broken into smaller reads across multiple network connections. They located the reconnection at line 17. They noticed that the condition on variable `blockEnd` can only be satisfied by two data flows by analyzing source code: (1) at line 10 when a backward seek is performed, (2) at line 21

```
1 int readAt(byte dest[], int off,
2            int end, int len) {
3   seek(off);
4   read(dest, end, len);
5 }
6 void seek(long off) {
7   ...
8   if (!done) {
9     pos = off;
10    blockEnd = -1;
11  }
12 }
13 void read(byte dest[], int end, int len) {
14   while (retries > 0) {
15     try {
16       if (blockEnd == -1) {
17         // Form a new connection for read.
18       }
19     } catch (IOException e) {
20       LOG(e);
21       blockEnd = -1;
22     }
23   }
24 }
```

Figure 4: Simplified code snippet for *hbase-1* [94]

when the connection is interrupted by an IOException. As they see no IOExceptions in the logs, they decided to log the seek position (`off`) in the function `seek()` to expand on their hypothesis. Upon observing a consistent delta of 33 in the expectedly random offsets – which they insightfully identified as the size of an HBase block header – they suspected a concurrency bug resetting the read offset as the potential root cause.

The developer utilizes program analysis (data flow analysis and symbolic constraint solving) and log analysis to adaptively collect system-specific information, resulting in a much more efficient and scoped information collection process. Though some existing works [70, 101] propose solutions to collect additional information automatically for debugging using control- and data-flow analysis, developers' log placement pruning practices utilize more advanced techniques combining symbolic constraint solving (constraints on `blockEnd`), anomaly analysis, and log analysis. This offers a promising avenue for efficient adaptive information collection.

5.1.2 Other Challenges. We observe two more interesting challenges during information collection. The first challenge is a *compromised crime scene*, in which failure-related information was wiped out, often due to automated failure recovery. For example, in *kube-2* [66], when investigating whether a high log production rate could have caused the

fluentd subsystem of a Kubernetes deployment to run out of memory, the developer mentions, "Unfortunately, I do not have access to the log files from right before the issue occurred anymore, and do not have a way to dig deeper at this time".

> Finding 3. *In 8% of all information collection steps, the additional collected information existed in the original failure execution but was wiped out, often due to automated recovery.*

The other challenge is the sheer variety in the number of tools of which knowledge is required to obtain all the information needed to debug or diagnose a failure. Across all cases studied, we identified 72 unique tools used, with an average of 1 tool per case, going as high as 8 for a case such as *kube-3* [4], for a failure that spanned Kubernetes, a Docker daemon, and systemd.

5.2 Online Interventions

> *"Observation is a passive science, experimentation an active science."*
>
> — Claude Bernard

When the developer has constructed a hypothesis for the root cause, they often come up with *interventions* – actions that actively change the execution of the system, including restarting nodes or processes, updating software, code changes, and configuration changes – to fix the root cause or mitigate its consequences, *in addition to confirming or ruling out their hypothetical causal chains that lead to the symptom.*

> Finding 4. *The prevalence of online interventions (11.3%) is on par with information collection (12%), highlighting its significant role in live debugging.*

A simple step in *hdfs-11* [9] demonstrates the effectiveness of interventions in debugging. When the developer is debugging a performance issue in JournalNode writes, they say, "In order to further determine whether it is a problem with the JN service itself, we decided to restart the JN service to see if the JN lag phenomenon can disappear." When they restarted it, they found the problem still existed and ruled out a JournalNode fault as the cause.

Many interventions are performed after a developer is fairly confident of the hypothesis, where the intervention serves merely as a confirmation (e.g., deploying a fix or workaround with high confidence) instead of a debugging aid. We investigate how many interventions served a significant diagnostic purpose as in *hdfs-11* [9] above. In particular, we labeled intervention steps as using "Unconfirmed" hypotheses of the root cause if the developer did not have a plan using the intervention to eliminate the failure, and "Confirmed" if they were deploying a fix or workaround to eliminate the failure with high confidence.

Category	Example Effects	Total (%)
Configuration **U**: 53%, **C**: 47%	Disable logging Swap/Disable modules Change timeout/retry config Horizontal scaling Resource changes Swap out storage Reduce load Other system-specific	45.4
Admin/User Command **U**: 54%, **C**: 46%	Kill process Skip/Delete data Perform failover Reduce load Others system-specific	13.3
Upgrade **U**: 62%, **C**: 38%	Upgrade modules	8.2
Restart **U**: 82%, **C**: 18%	Restart component Perform failover Reclaim memory/resources	22.4
Source Code **U**: 19%, **C**: 81%	Skip corrupt data Swap algorithms Disable/deprecate functionality Enforce timeout for function Others system-specific	18.4
Hardware **U**: 0%, **C**: 100%	Replace network switch	1.0

Table 13: Category of interventions. U and C denote whether an intervention was performed with a hypothesis that was "Unconfirmed" and "Confirmed" respectively.

> Finding 5. *Over half (51.5%) of interventions were performed when developers were clearly unsure of their hypothesis, indicating that they actively use interventions as a debugging aid.*

The result shows most interventions are performed for diagnosis purposes instead of confirmation purposes. The exact behaviors tested by interventions are shown in Table 13.

5.2.1 Automated Interventions for Debugging. Our analysis (Table 13) reveals opportunities to perform automated online interventions for debugging purposes.

Finding 6. *Almost half (45.4%) of interventions are configuration changes.*

For instance, developers frequently adjust configurations for debugging purposes, despite software patches being the majority (53.4%) of root cause fixes applied to all cases. **Implication:** Adjusting failure-related configuration options is the most popular intervention strategy. However, automation of this strategy is only discussed in the context of security to avoid CVEs [62, 65] at compile time, but not live debugging cloud failures in production.

Finding 7. *A large portion (22.4%) of interventions are node restarts with 82% of them being performed before a well reasoned hypothesis is reached.*

The traditional practice – rebooting computers to resolve failures – is a common strategy for debugging purposes in cloud systems. With distributed systems having many fault-tolerant components, node restart is a very common action to verify hypotheses such as "a component is behaving abnormally", with the assumption that the reboot and failover mechanism clears the state of the component that is related to the failure. **Implication:** Automated diagnosis tools should utilize the fault-tolerance mechanism of distributed systems to perform experiments. In addition, fault-tolerance mechanisms enable cloud systems to introduce fine-grained intervention knobs for debugging purposes as long as such knobs are tolerated by fault-tolerance mechanisms (§ 5.2.2).

Finding 8. *The required interventions may not be able to be performed (3.1% of interventions).*

In addition, we observe that developers are eager to perform an intervention but are unable to due to the lack of intervention mechanisms. For example, in *kube-4* [8], the developer wished to increase thread queue capacity when faced with an error from a component that said that the requests were rejected, and the number of queued tasks was greater than the capacity. However, this variable was not configurable, and led the developers to solve the problem through other means. This highlights the need for more intervention knobs for live debugging.

5.2.2 Fine-Grained Knob Injection for Debugging. All interventions performed by developers other than source code changes and hardware fixes indicate *knobs* that developers are free to tune at runtime to test the system. Given how heavily these interventions are used, system design should

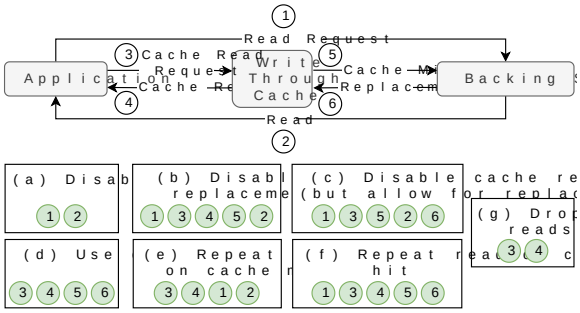

Figure 5: A hypothetical example of how different interventional knobs could be used to diagnose a failure. (a-g) represent knobs. (1-6) represent code paths. Enabling/disabling the different code paths serves to isolate the root cause.

strive to include knobs for the examples mentioned in Table 13 not just for modular design or reliability, but especially for *diagnosability* – to test more fine-grained patterns in system behavior.

Source code interventions (especially those that did not follow confident hypotheses and did not lead to fixes) indicate knobs that developers wished they had, but did not. For example, in *hdfs-8* [6], the developer is debugging a performance problem in a large Hadoop cluster that manages petabytes of data. They form a hypothesis that frequent locking of the NameNode lock is the cause. They then try a number of measures in rapid succession, such as adjusting a configuration controlling frequency of NameNode heartbeats (which could grab the NameNode lock frequently), and changing the source code to change the NameNode lock from fair mode (assigned in arrival order) to non-fair mode (entry order is unspecified, but usually has higher throughput). Being uncertain of what could be the bottleneck to be tackled, the developers made these code changes **in production** to test their hypotheses. In other cases, developers also included other interventional knobs in their source code such as skipping over corrupt items while processing in a loop (*hdfs-4* [14]), and removing locks (*hdfs-15* [13], Figure 6).

To illustrate the feasibility of fine-grained knob injection in detail, we consider a situation inspired by *kafka-14* [88] (Figure 5), wherein a memory leak occurred in the cache read. In this situation, if the developer knows that application reads are causing the leaky behavior, but does not know exactly where, they could selectively enable or disable certain parts of the code. As shown in the figure, the options enumerate ways that an application that uses a cache and backing store could selectively use different code paths to enumerate possibilities. For example, option (c) is a possibility where an application could issue a read to a cache, but also directly to the backing store, ignoring the result of

```
1 Node[] chooseTargets(int nReplicas, Type t) {
2   Node[] targets;
3   while (nReplicas > 0) { // replicas needed
4     Node node = chooseNextRandom(t);
5     if (node == NULL) // checked all nodes
6       break;
7     targets.add(node);
8     nReplicas--;
9   }
10  return targets;
11 }
12 Node chooseNextRandom(Type t) {
13   Node[] candidates;
14   for (Node n : allNodes()) {
15     if (n.hasStorageType(t)) // grabs a lock
16       candidates.add(n);
17   }
18   return random(candidates);
19 }
```

Figure 6: Simplified code snippet for *hdfs-15* [13]

the cache read. Option (g) is more aggressive, dropping any reads that are not in the cache entirely. Between all of the options, the developers can narrow down the cause to any of the single code paths.

Implication: Our analysis reveals the need for, and feasibility of knob injection to increase the diagnosability of cloud systems in production. Frameworks that can safely inject more fine-grained knobs make it much easier to evaluate hypotheses in production. Existing work and tools make use of interventions, mainly fault injection [33, 41, 84] among others [51]. However, these works either serve the purpose of testing, or are not suited for production use.

5.2.3 Safety Concerns in Interventions. Performing interventions in production environments inherently comes with certain risks, evidenced by 2.6% of documented interventions having negative consequences.

> Finding 9. *Interventions may have negative consequences (2.6% of interventions); these consequences are crucial to avoid for production systems.*

We demonstrate how this happened in *hdfs-15* [13] (Figure 6). The developer is diagnosing a performance delay in the `chooseTargets()` function, which iterates through all nodes in the cluster and randomly picks `nReplicas` nodes with specified node type `t`. It invokes the `hasStorageType()` function at line 15, which grabs a lock and releases it. The developer formulated one hypothesis – the frequent lock acquisition was causing the delay – and removed the lock operation. However, it made things worse and completely overloaded their NameNode. It was because the actual cause – too many loop iterations at line 3 due to not enough good nodes of type `t` – conflicted with their hypothesis in such a way that the intervention based on their hypothesis – removing the lock – saturated the NameNode CPU by iterating through all nodes in the cluster repeatedly.

Implication: Before performing an intervention in production (or injecting a knob), it is crucial to assess the risk of potential negative consequences if, instead, an alternative hypothesis is true. Risk assessment should be performed for any online intervention. This is a unique requirement compared to intervention-based offline test failure debugging tools [51], which are free to perform arbitrary execution modifications. *hdfs-15* [13] hints the feasibility of an automated risk assessment tool: Figure 6 is simplified from 150 lines of code, which is potentially feasible for program analysis techniques, such as symbolic execution, to analyze execution paths, formulate hypotheses, and predict consequences.

5.3 Offline Reproduction

Offline failure reproduction is one of the most important [42] prerequisites for successful debugging, yet a well-known challenging task for production failures.

> Finding 10. *Most (78.5%) failures were completely diagnosed online. Among all cases reproduced offline (21.5%), half (10.8%) were reproduced after the root cause was diagnosed.*

Bettenburg et al. [42] shows that most (80%) single-node application failures are diagnosed given reported reproduction steps. In comparison, reproduction is mostly (89.2%) unavailable before the root cause is diagnosed for production cloud system failures. This calls for better support for live debugging techniques, such as adaptive information collection and automated knob injection for interventions as discussed previously. It also motivates research on record and replay [54, 77], automated offline reproduction [110] techniques to overcome bottlenecks such as recording overhead and path explosion when analyzing long execution traces.

> Finding 11. *About a fifth (17.9%) of the successful reproductions requires ad-hoc modifications to the source code in order to enforce timing constraints.*

As shown in Table 14, in the successful reproductions, we study what special conditions developers enforce or simulate, including faults, timing constraints, and software or hardware heterogeneity.

We find that special timing constraints are enforced either through ad-hoc modifications to the source code (17.9%) or stress testing (10.7%). For example, *kafka-6* [15] required a specific condition to be held: a partition should not have a leader for a specific period of time. To enforce this in offline reproduction, they modified a heartbeat timer to not check for a leader.

Condition	Cases (%)
Fault	17.9
Timing	28.6
Heterogeneity	3.6

Table 14: Special cond.

Implication: Enforcing timing constraints through a *general* interface is needed for easy reproduction of cloud failures. Tools like IMUnit [67] and CONCURRIT [50] provide domain-specific languages to specify and enforce thread scheduling constraints for multi-threaded applications. Similar techniques are necessary to specify and enforce network message scheduling constraints for distributed systems.

6 CLOUD-RELATED CHALLENGES AND NOVEL DEBUGGING STRATEGIES

In this section, we discuss challenges inherent to or amplified by production cloud environments, as evidenced by our analysis of hypothesis forking reasons (§ 4.2), and present developers' novel debugging strategies to address such challenges.

> Finding 12. *Challenges amplified in the production cloud environment account for a major portion (black-box reasoning (44.6%) and concurrency (0.9%)) of the hypothesis forking reasons. Meanwhile, unique features of distributed systems such as distributed partial failure modes, automated failure recovery, network message across nodes, as well as dynamic network and component dependency in microservices, constitute a non-trivial portion (14.4%).*

6.1 Partial Failures

In our dataset, most failures are partial or gray failures [63, 78]: they did not involve the complete stop or crash of a process, hardware failure, or failure of a network link. This is reflected in the hypothesis forking reasons namely blackbox reasoning at component boundaries, reasoning about partial failure modes, failure recovery, and network messages.

Developers address these challenges by collecting information on component boundaries and failure recovery logic to perform anomaly detection. Such information is often not available, as evidenced by a developer's quote – "Finding when the NameNode failed over was surprisingly hard. Tools like Grafana and SmartSense don't seem to track this out of the box" (*kube-4* [8]). In fact, we find that 9.6% of information collection steps collect inter-component data. Though existing works [63, 78] have improved observability at component boundaries to detect partial failures, collecting information about failure recovery and partial failure conditions have yet to be investigated.

In some cases, partial failures manifest as components of a certain type failing, while others of that type do not. An interesting strategy developers adopted in this scenario is utilizing the symmetry of nodes, and patterns derived from them. In 14.3% of execution comparison steps in exploration, developers utilized symmetry among replicated data and nodes to find common or differing properties as potential causes. In *hdfs-11* [9], the developer observes differences in latency between multiple JournalNodes to a NameNode, to determine whether the NameNode is faulty or not.

6.2 Concurrency

The most ingenious strategies we observed were adopted by developers when debugging concurrency issues, allowing them to *short-circuit* their reasoning. When searching backwards from a failure, developers rarely consider concurrency issues initially. Instead, they utilize unique *indicators* for concurrency issues – specific patterns in runtime data that are suspected to be a result of concurrency issues such as data races and message reordering. These include **shared variables with unusual values** (*hive-6* [24], *hbase-1* [94]), observations of **simultaneous exceptions from different threads** (*hdfs-4* [14], *fastdfs-1* [3]), observing **unexpected timing relationships between logs from different threads** (in the case of *hdfs-6* [1], the developers observed seemingly unrelated logs between threads performing copy and delete happening simultaneously, repeatedly), or using a **"magic number"** as a concurrency indicator (as mentioned in *hbase-1* [94]). In the last case, the developer observed a consistent delta of 33 in the offsets, which are expected to be random. They perceptively identified this number as the size of an HBase block header – and suspected a concurrency bug resetting the read offset as the potential root cause. These indicators allow developers to consider causal explanations involving interactions between multiple threads on an *as-needed* basis, rather than considering all possible thread interleavings right from the beginning. Figure 7 shows an example of how developers tracked from multiple errors in different threads to the problematic shared variable in *fastdfs-1* [3].

> Finding 13. *Developers use concurrency indicators as clues that they should search for interleaved execution paths in a small scope.*

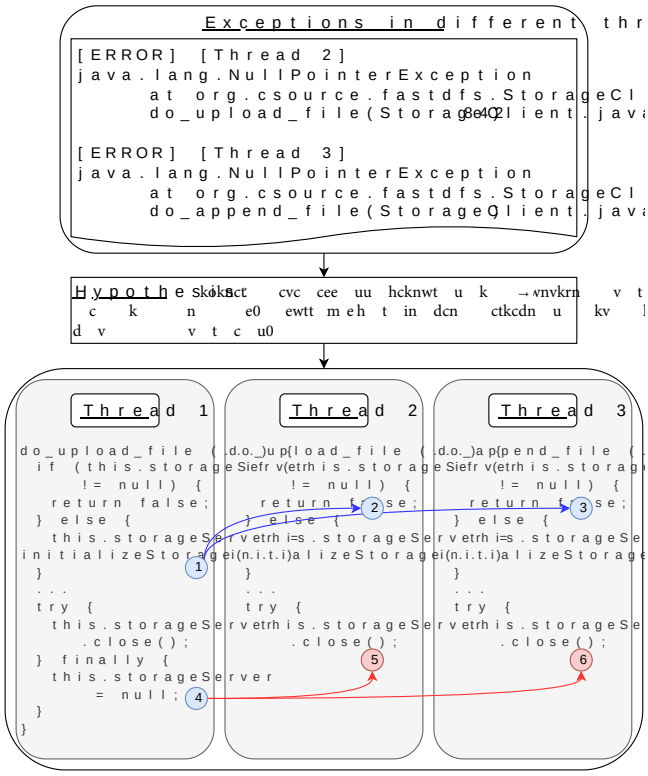

Figure 7: An example of special reasoning by which the developer uses a concurrency indicator pattern to scope the exact paths to analyze. The presence of exceptions in multiple threads allows the developer to reason that the initialization (①) of the `storageServer` variable by Thread 1 is read (②, ③) by the other threads, but is released by Thread 1 (④) before the others (⑤, ⑥), which causes the (NullPointer) exceptions.

6.3 High Availability

As cloud systems are designed to be highly available and continuously running, developers face unique challenges debugging failures that become apparent only over extended periods of time. In *hdfs-5* [7], the root cause, a memory leak, accumulates over the course of six months, resulting in a NodeManager `OutOfMemory` error. We term this as *fail-slow software failure* (different from fail-slow hardware failures [60]), which is characteristic of long-running production systems.

The developer used a unique strategy – knowledge of code execution frequency – to quickly narrow down to the potential root cause: because the `OutOfMemory` error happened every half a year, they suspected that an infrequently executed code path was leaking memory, and quickly found two such code snippets (exception handlers) that missed deallocation statements. These turned out to be the exact root cause.

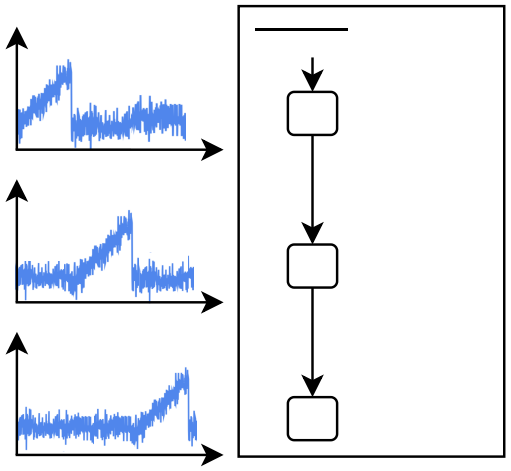

Figure 8: Utilizing time series patterns in memory utilization on multiple machines to identify cascading failures.

> FINDING 14. *Using information about execution frequency is effective for debugging fail-slow software failures.*

Implication: To the best of our knowledge, though fail-slow hardware failures [60] have been investigated recently, *fail-slow software failures* are not well-studied. How to estimate and utilize execution frequency for automated debugging is a promising yet uninvestigated technique.

6.4 Cascading Failures

Cascading failures [74, 100] and vicious cycle failures [86] are unique failures in cloud systems with severe consequences. They are often closely coupled with partial failures since they require the failure to propagate, and can form a cycle across components.

We observe that developers utilize special time series patterns to debug these kinds of failures quickly. In *etcd-3* [25] (Figure 8), there was a cycle in the system that continuously caused an instance that ran out of memory to offload its tasks to another instance, and so on. The developer initially observed the OOMs on the different nodes and started reasoning about different hypotheses regarding what could cause the code to leak resources. However they could not find a satisfactory explanation. It was only when they observed the cyclical pattern in the memory utilization and failures of the different nodes, did they realize that it was actually a cascading failure, and this helped them to quickly hone in on the correct explanation, which was that a specific heavy client operation was being shuttled from one instance to the next, causing failures.

6.5 Dynamism

Microservice systems exhibit the phenomenon of *dynamism*; complex and continuously changing environments due to the ability to quickly scale, swap out independent components or services, and develop user-defined architectures and applications. This leads to several unique challenges in debugging, as the dependencies in the system are ever-changing. One such challenge was faced in *kube-1* [61], wherein an IP assigned to the Load Balancer was changed dynamically, and the change did not propagate to other nodes and pods, which caused the developer to need to look for anomalies in network data, and infer why connectivity that should have been present was not. In *kube-6* [82], developers encountered a situation where different containers were placed in different Virtual Private Clouds, due to configuration changes, which led to a loss of connectivity between different services. While dynamism has been considered and used to develop tools for debugging [115], more complex dynamism patterns such as the relationship between the network configurations of different instances as in the case of *kube-6* [82] above are yet to be examined deeply.

7 RELATED WORK

We present related work in studies on debugging, studies on cloud failures, and debugging tools for cloud failures.

7.1 Studies on debugging

The debugging procedure has been studied extensively, in many different contexts [43, 49, 57, 72, 73, 90, 99, 114]. Most studies rely on gathering anecdotes [49], conducting interviews with, or passing out questionnaires to developers [73, 114] or students [72, 99]. More closely, the work by Zhou et al. [114] studies debugging procedures in microservice systems – however, it focuses on a coarser characterization of the steps involved, and the effectiveness of log based strategies on debugging. The work by Böhme et al. [43] studies debugging by professional developers in more detail, and derives its results from a user study of developers debugging local, single machine failures. Studying cloud production incidents leads to a different set of findings, especially those on experimentation, that are not observed when debugging local programs. Recently, researchers from Microsoft studied common root causes of production incidents and root cause labeling given a postmortem report [48] in Microsoft Azure, but they did not analyze how the root causes are diagnosed.

7.2 Studies on cloud failures

Failures in production cloud systems have been explored widely [59, 68, 71, 75, 78, 87, 103, 112], studying the characteristics of bugs, their manifestations, fixes, and so on, with some works focusing on specific categories of bugs such as performance [68], concurrency [71], partial failure [78] and upgrade [112] bugs. However, they do not study the actual debugging steps taken to diagnose or debug the failures.

7.3 Debugging tools for cloud failures

There is extensive research [40, 44, 83, 104] on debugging tools for single-node applications. Debugging techniques for cloud failures are mostly based on log analysis [39, 110, 113], distributed tracing [52, 53, 80, 91], anomaly analysis [47, 93, 96, 102, 109], and statistical debugging (along with collecting targeted information) [56, 70, 76, 95]. Others focus on utilizing special hardware support [69, 70], and checkpointing strategies [97]. Our study complements these tools by showing what information is required to make their analyses more accurate.

Recently, there is work on failure reproduction [45, 110], failure minimization [89], root cause localization [70, 104, 111], as well as interactive debugging [54, 77] for distributed systems. Our study complements them by discovering challenges not addressed and strategies not utilized in these tools.

8 CONCLUSION

This paper presents the first in-depth, observational study of live debugging in production cloud systems. We provide taxonomies of a variety of activities and strategies performed when debugging. We identify unique challenges faced in diagnosing production cloud failures and developers' novel debugging techniques to fight against them. In addition, we quantify the requirement of effective anomaly detection for end-to-end debugging, and analyze the usage of interventions for live debugging.

9 ACKNOWLEDGEMENTS

We thank Ramnatthan Alagappan, our shepherd, and the anonymous reviewers for their feedback and comments. We would also like to thank Adrian Raj and Yiqi Hu for analyzing debugging experience documents. This research is supported by NSF 2140305.

REFERENCES

[1] 2016. NameNode RepicationMonitor Exception Tracking- Hexiaoqiao. Retrieved 2022-12-10 from https://hexiaoqiao.github.io/blog/2016/09/13/namenode-repicationmonitor-exception-trace/

[2] 2016. Remember a DataNode Slow Start Problem - Blog of People on the Android Road - CSDN Blog. Retrieved 2023-04-12 from https://blog.csdn.net/Androidlushangderen/article/details/50500136

[3] 2017. It Must Be All Your Fault! - A Troubleshooting Experience of FastDFS Concurrency Problems- Pure Smile- Blog Garden. Retrieved 2023-09-04 from https://www.cnblogs.com/ityouknow/p/8123998.html

[4] 2019. k8s|A Troubleshooting-Tencent Cloud Developer Community-Tencent Cloud. Retrieved 2022-12-09 from https://cloud.tencent.com/developer/article/1444074

[5] 2019. The Task Submitted by Hive to Yarn Has Been Running Troubleshooting_hive on Spark Task Running Yarn Ui Shows Running. Retrieved 2023-04-17 from https://blog.csdn.net/u013332124/article/details/89283727

[6] 2020. Remember a Super-Trillion-Scale Hadoop NameNode Performance Troubleshooting Process. Retrieved 2023-04-15 from https://blog.csdn.net/weixin_44253169/article/details/105564433

[7] 2020. Solve the Memory Leak Problem That NodeManager Frequently Triggers FULL-GC after Running for about Half a Year. Retrieved 2022-12-09 from https://blog.csdn.net/weixin_43990680/article/details/104754341

[8] 2020. Why We Switched from Fluent-Bit to Fluentd in 2 Hours. https://prometheuskube.com/why-we-switched-from-fluent-bit-to-fluentd-in-2-hours

[9] 2021. Analysis and Troubleshooting of a HDFS JournalNode Transaction Lag Problem. Retrieved 2022-12-09 from https://blog.csdn.net/Androidlushangderen/article/details/112744149

[10] 2021. Big Data Troubleshooting Series-HIVE Stepping on the Pit-HIVE-15642_org.Apache.Hadoop.Hive.thrift_Ming Ge's IT Essay Blog-CSDN Blog. Retrieved 2023-04-12 from https://blog.csdn.net/MichaelLi916/article/details/119902075

[11] 2021. An HBase & HDFS Short-Circuit Read Odyssey. Retrieved 2024-10-12 from https://blogsarchive.apache.org/hbase/entry/an-hbase-hdfs-short-circuit

[12] 2021. Record the Upgrading Process of Hadoop Cluster with Thousands of Nodes in the Database. Retrieved 2022-12-12 from https://zhuanlan.zhihu.com/p/163352048

[13] 2021. Record the Upgrading Process of Hadoop Cluster with Thousands of Nodes in the Database (Part 3). Retrieved 2023-04-16 from https://zhuanlan.zhihu.com/p/163352048

[14] 2021. Troubleshooting a Problem That HDFS Snapshot Cannot Be Deleted_hadoop Deletion Error Has a Snapshot. Retrieved 2023-04-17 from https://blog.csdn.net/Androidlushangderen/article/details/113446906

[15] 2021. When Kafka Went Offshore. https://blog.gojek.io/when-kafka-went-offshore/

[16] 2022. Apache Cassandra | Apache Cassandra Documentation. Retrieved 2022-12-13 from https://cassandra.apache.org/_/index.html

[17] 2022. Apache Flink: Stateful Computations over Data Streams. Retrieved 2022-12-13 from https://flink.apache.org/

[18] 2022. Apache HBase – Apache HBase™ Home. Retrieved 2022-12-13 from https://hbase.apache.org/

[19] 2022. Apache Hive. Retrieved 2022-12-13 from https://hive.apache.org/

[20] 2022. Apache Kafka. Retrieved 2022-12-13 from https://kafka.apache.org/

[21] 2022. Google Cloud Computing Services. Retrieved 2022-12-13 from https://cloud.google.com/

[22] 2022. Hadoop Distributed File System (HDFS). Retrieved 2022-12-13 from https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html

[23] 2022. Kubernetes: Production-Grade Container Orchestration. Retrieved 2022-12-13 from https://kubernetes.io/

[24] 2023. Hive on Mr Job Repeated Execution Problem Troubleshooting_hive Re-Read Execution-CSDN Blog. Retrieved 2023-11-28 from https://blog.csdn.net/u013332124/article/details/106575443

[25] 2023. Remember to Troubleshoot the ETCD OOM Problem Once. Retrieved 2023-11-29 from https://zhuanlan.zhihu.com/p/571473832

[26] 2023. Twists and Turns - Remember a K8S Cluster Application Troubleshooting - JD Cloud Developer Community. Retrieved 2023-11-30 from https://developer.jdcloud.com/article/1538

[27] 2024. Apache Spark™ - Unified Engine for Large-Scale Data Analytics. Retrieved 2024-10-15 from https://spark.apache.org/

[28] 2024. Etcd. Retrieved 2024-10-15 from https://etcd.io/

[29] 2024. Gemini. Retrieved 2024-10-06 from https://gemini.google.com

[30] 2024. Gojek Super App. Retrieved 2024-10-15 from https://www.gojek.com/en-id

[31] 2024. Hive Cannot Submit to Yarn_08235.15.1 Analysis of the Problem of Slow Hive Query Caused by Slow HDFS - CSDN Blog. Retrieved 2024-10-12 from https://blog.csdn.net/weixin_42139302/article/details/112092820

[32] 2024. MongoDB: The Developer Data Platform | MongoDB. Retrieved 2024-10-15 from https://www.mongodb.com/

[33] 2024. Netflix/Chaosmonkey. Netflix, Inc.. https://github.com/Netflix/chaosmonkey

[34] 2024. PagerDuty | Real-Time Operations | Incident Response | On-Call. Retrieved 2024-10-15 from https://www.pagerduty.com/

[35] 2024. Redis - The Real-time Data Platform. Retrieved 2024-10-15 from https://redis.io/

[36] 2024. TiDB, Powered by PingCAP. Retrieved 2024-10-15 from https://www.pingcap.com/

[37] 2024. Zookeeper Once Fault Handling_exception Causing Close. Retrieved 2024-10-15 from https://blog.csdn.net/huochen1994/article/details/79288194

[38] 2024. ZooKeeper Overview. Retrieved 2024-10-16 from https://zookeeper.apache.org/doc/r3.1.2/zookeeperOver.html

[39] Marcos K Aguilera, Jeffrey C Mogul, Janet L Wiener, Patrick Reynolds, and Athicha Muthitacharoen. 2003. Performance debugging for distributed systems of black boxes. ACM SIGOPS Operating Systems Review 37, 5 (2003), 74–89.

[40] Gautam Altekar and Ion Stoica. 2009. ODR: Output-Deterministic Replay for Multicore Debugging. In Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles - SOSP '09. ACM Press, Big Sky, Montana, USA, 193. https://doi.org/10.1145/1629575.1629594

[41] Peter Alvaro, Joshua Rosen, and Joseph M. Hellerstein. 2015. Lineage-Driven Fault Injection. In Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15). Association for Computing Machinery, New York, NY, USA, 331–346. https://doi.org/10.1145/2723372.2723711

[42] Nicolas Bettenburg, Sascha Just, Adrian Schröter, Cathrin Weiss, Rahul Premraj, and Thomas Zimmermann. 2008. What Makes a Good Bug Report?. In Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering. 308–318.

[43] Marcel Böhme, Ezekiel O. Soremekun, Sudipta Chattopadhyay, Emamurho Ugherughe, and Andreas Zeller. 2017. Where Is the Bug and How Is It Fixed? An Experiment with Practitioners. In Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering. 117–128.

[44] Cristian Cadar and Koushik Sen. 2013. Symbolic Execution for Software Testing: Three Decades Later. Commun. ACM 56, 2 (Feb. 2013), 82–90. https://doi.org/10.1145/2408776.2408795

[45] Weidong Cui, Xinyang Ge, Baris Kasikci, Ben Niu, Upamanyu Sharma, Ruoyu Wang, and Insu Yun. 2018. REPT: Reverse Debugging of Failures in Deployed Software. In 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18). 17–32. https://www.usenix.org/conference/osdi18/presentation/weidong

[46] Datadog. 16:32:45 -0400 -0400. Cloud Monitoring as a Service | Datadog. Retrieved 2022-10-14 from https://www.datadoghq.com/

[47] Daniel Joseph Dean, Hiep Nguyen, and Xiaohui Gu. 2012. UBL: Unsupervised Behavior Learning for Predicting Performance Anomalies in Virtualized Cloud Systems. In Proceedings of the 9th International Conference on Autonomic Computing - ICAC '12. ACM Press, San Jose, California, USA, 191. https://doi.org/10.1145/2371536.2371572

[48] Pradeep Dogga, Chetan Bansal, Richard Costleigh, Gopinath Jayagopal, Suman Nath, and Xuchao Zhang. 2023. AutoARTS: Taxonomy, Insights and Tools for Root Cause Labelling of Incidents in Microsoft Azure. In 2023 USENIX Annual Technical Conference (USENIX ATC 23). 359–372.

[49] Marc Eisenstadt. 1997. My Hairiest Bug War Stories. Commun. ACM 40, 4 (April 1997), 30–37. https://doi.org/10.1145/248448.248456

[50] Tayfun Elmas, Jacob Burnim, George Necula, and Koushik Sen. 2013. CONCURRIT: A domain specific language for reproducing concurrency bugs. In Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation. 153–164.

[51] Anna Fariha, Suman Nath, and Alexandra Meliou. 2020. Causality-Guided Adaptive Interventional Debugging. In Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD '20). Association for Computing Machinery, New York, NY, USA, 431–446. https://doi.org/10.1145/3318464.3389694

[52] Yu Gan, Mingyu Liang, Sundar Dev, David Lo, and Christina Delimitrou. 2021. Sage: practical and scalable ML-driven performance debugging in microservices. In Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems. 135–151.

[53] Yu Gan, Yanqi Zhang, Kelvin Hu, Dailun Cheng, Yuan He, Meghna Pancholi, and Christina Delimitrou. 2019. Seer: Leveraging big data to navigate the complexity of performance debugging in cloud microservices. In Proceedings of the twenty-fourth international conference on architectural support for programming languages and operating systems. 19–33.

[54] Dennis Geels, Gautam Altekar, Petros Maniatis, Timothy Roscoe, and Ion Stoica. 2007. Friday: Global Comprehension for Distributed Replay.. In NSDI, Vol. 7. 285–298.

[55] Supriyo Ghosh, Manish Shetty, Chetan Bansal, and Suman Nath. 2022. How to fight production incidents? an empirical study on a large-scale cloud service. In Proceedings of the 13th Symposium on Cloud Computing. 126–141.

[56] Kirk Glerum, Kinshuman Kinshumann, Steve Greenberg, Gabriel Aul, Vince Orgovan, Greg Nichols, David Grant, Gretchen Loihle, and Galen Hunt. 2009. Debugging in the (Very) Large: Ten Years of Implementation and Experience. In Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP '09). Association for Computing Machinery, New York, NY, USA, 103–116. https://doi.org/10.1145/1629575.1629586

[57] John D. Gould and Paul Drongowski. 1974. An Exploratory Study of Computer Program Debugging. Human Factors 16, 3 (1974), 258–277.

[58] Leo Gugerty and Gary Olson. 1986. Debugging by skilled and novice programmers. In Proceedings of the SIGCHI conference on human factors in computing systems. 171–174.

[59] Haryadi S. Gunawi, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, Thanh Do, Jeffry Adityatama, Kurnia J. Eliazar, Agung Laksono, Jeffrey F. Lukman, and Vincentius Martin. 2014. What Bugs Live in the Cloud? A Study of 3000+ Issues in Cloud Systems. In Proceedings of the ACM Symposium on Cloud Computing. 1–14.

[60] Haryadi S Gunawi, Riza O Suminto, Russell Sears, Casey Golliher, Swaminathan Sundararaman, Xing Lin, Tim Emami, Weiguang Sheng, Nematollah Bidokhti, Caitie McCaffrey, et al. 2018. Fail-slow at scale: Evidence of hardware performance faults in large production systems. ACM Transactions on Storage (TOS) 14, 3 (2018), 1–26.

[61] Dominic Gunn. 2018. Kubernetes and the Menace ELB, the Tale of an Outage. Retrieved 2023-04-13 from https://itnext.io/kubernetes-and-the-menace-elb-the-tale-of-an-outage-c00bef678fc0

[62] David Ke Hong, Qi Alfred Chen, and Z Morley Mao. 2017. An initial investigation of protocol customization. In Proceedings of the 2017 Workshop on Forming an Ecosystem Around Software Transformation. 57–64.

[63] Peng Huang, Chuanxiong Guo, Lidong Zhou, Jacob R. Lorch, Yingnong Dang, Murali Chintalapati, and Randolph Yao. 2017. Gray Failure: The Achilles' Heel of Cloud-Scale Systems. In Proceedings of the 16th Workshop on Hot Topics in Operating Systems. ACM, Whistler BC Canada, 150–155. https://doi.org/10.1145/3102980.3103005

[64] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. 2010. ZooKeeper: Wait-free Coordination for Internet-scale Systems.. In USENIX Annual Technical Conference, Vol. 8.

[65] Muhammad Adil Inam and Wajih Ul. 2022. Forensic analysis of configuration-based attacks. In Proceedings of the 2022 Network and Distributed System Security Symposium.

[66] Keilan Jackson. 2019. Post Mortem: Kubernetes Node OOM. Retrieved 2022-12-11 from https://www.bluematador.com/blog/post-mortem-kubernetes-node-oom

[67] Vilas Jagannath, Milos Gligoric, Dongyun Jin, Grigore Rosu, and Darko Marinov. 2010. IMUnit: improved multithreaded unit testing. In Proceedings of the 3rd International Workshop on Multicore Software Engineering. 48–49.

[68] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. 2012. Understanding and Detecting Real-World Performance Bugs. In Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '12). Association for Computing Machinery, New York, NY, USA, 77–88. https://doi.org/10.1145/2254064.2254075

[69] Baris Kasikci, Weidong Cui, Xinyang Ge, and Ben Niu. 2017. Lazy Diagnosis of In-Production Concurrency Bugs. In Proceedings of the 26th Symposium on Operating Systems Principles. ACM, Shanghai China, 582–598. https://doi.org/10.1145/3132747.3132767

[70] Baris Kasikci, Benjamin Schubert, Cristiano Pereira, Gilles Pokam, and George Candea. 2015. Failure Sketching: A Technique for Automated Root Cause Diagnosis of in-Production Failures. In Proceedings of the 25th Symposium on Operating Systems Principles. ACM, Monterey California, 344–360. https://doi.org/10.1145/2815400.2815412

[71] Tanakorn Leesatapornwongsa, Jeffrey F. Lukman, Shan Lu, and Haryadi S. Gunawi. 2016. TaxDC: A Taxonomy of Non-Deterministic Concurrency Bugs in Datacenter Distributed Systems. In Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems. 517–530.

[72] Colleen M. Lewis. 2012. The Importance of Students' Attention to Program State: A Case Study of Debugging Behavior. In Proceedings of the Ninth Annual International Conference on International Computing Education Research (ICER '12). Association for Computing Machinery, New York, NY, USA, 127–134. https://doi.org/10.1145/2361276.2361301

[73] Bowen Li, Xin Peng, Qilin Xiang, Hanzhang Wang, Tao Xie, Jun Sun, and Xuanzhe Liu. 2022. Enjoy Your Observability: An Industrial Survey of Microservice Tracing and Analysis. Empirical Software Engineering 27, 1 (Jan. 2022). https://doi.org/10.1007/s10664-021-10063-9

[74] Jiaxin Li, Yuxi Chen, Haopeng Liu, Shan Lu, Yiming Zhang, Haryadi S. Gunawi, Xiaohui Gu, Xicheng Lu, and Dongsheng Li. 2018. Pcatch: Automatically Detecting Performance Cascading Bugs in Cloud Systems. In Proceedings of the Thirteenth EuroSys Conference (EuroSys '18). Association for Computing Machinery, New York, NY, USA, 1–14. https://doi.org/10.1145/3190508.3190552

[75] Sihan Li, Hucheng Zhou, Haoxiang Lin, Tian Xiao, Haibo Lin, Wei Lin, and Tao Xie. 2013. A Characteristic Study on Failures of Production Distributed Data-Parallel Programs. In 2013 35th International Conference on Software Engineering (ICSE). IEEE, 963–972.

[76] Ben Liblit, Mayur Naik, Alice X. Zheng, Alex Aiken, and Michael I. Jordan. 2005. Scalable Statistical Bug Isolation. ACM SIGPLAN Notices 40, 6 (June 2005), 15–26. https://doi.org/10.1145/1064978.1065014

[77] Xuezheng Liu, Zhenyu Guo, Xi Wang, Feibo Chen, Xiaochen Lian, Jian Tang, Ming Wu, M. Frans Kaashoek, and Zheng Zhang. 2008. D3S: Debugging Deployed Distributed Systems. In NSDI. https://www.usenix.org/event/nsdi08/tech/full_papers/liu_xuezheng/liu_xuezheng_html/

[78] Chang Lou, Peng Huang, and Scott Smith. 2020. Understanding, Detecting and Localizing Partial Failures in Large System Software.. In NSDI, Vol. 20. 559–574.

[79] David E Lowell, Subhachandra Chandra, and Peter Chen. 2000. Exploring failure transparency and the limits of generic recovery. In Fourth Symposium on Operating Systems Design and Implementation (OSDI 2000).

[80] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. 2015. Pivot Tracing: Dynamic Causal Monitoring for Distributed Systems. In Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15). Association for Computing Machinery, New York, NY, USA, 378–393. https://doi.org/10.1145/2815400.2815415

[81] David McGinnis. 2020. Debugging From The Field: The Case of the Empty Files. Retrieved 2022-12-11 from https://www.davidmcginnis.net/post/debugging-from-the-field-the-case-of-the-empty-files

[82] Yash Mehrotra. 2020. The Case of the Missing Packet: An EKS Migration Tale. Retrieved 2024-06-06 from https://yashmehrotra.com/posts/the-case-of-the-missing-packet-an-eks-migration-tale/

[83] Robert O'Callahan, Chris Jones, Nathan Froyd, Kyle Huey, Albert Noll, and Nimrod Partush. 2017. Engineering Record and Replay for Deployability. In 2017 USENIX Annual Technical Conference (USENIX ATC 17). 377–389. https://www.usenix.org/conference/atc17/technical-sessions/presentation/ocallahan

[84] Cuong Pham, Long Wang, Byung Chul Tak, Salman Baset, Chunqiang Tang, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. 2017. Failure Diagnosis for Distributed Systems Using Targeted Fault Injection. IEEE Transactions on Parallel and Distributed Systems 28, 2 (Feb. 2017), 503–516. https://doi.org/10.1109/TPDS.2016.2575829

[85] Prometheus. 2022. Prometheus - Monitoring System & Time Series Database. Retrieved 2022-10-14 from https://prometheus.io/

[86] Shangshu Qian, Wen Fan, Lin Tan, and Yongle Zhang. 2023. Vicious Cycles in Distributed Software Systems. In 2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, Luxembourg, Luxembourg, 91–103. https://doi.org/10.1109/ASE56229.2023.00032

[87] Ariel Rabkin and Randy Howard Katz. 2012. How Hadoop Clusters Break. IEEE software 30, 4 (2012), 88–94.

[88] Sid Rathi. 2021. Solving a Native Memory Leak. https://medium.com/expedia-group-tech/solving-a-native-memory-leak-71fe4b6f9463

[89] Colin Scott, Vjekoslav Brajkovic, George Necula, Arvind Krishnamurthy, and Scott Shenker. 2016. Minimizing Faulty Executions of Distributed Systems. In 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16). 291–309.

[90] Benjamin Siegmund, Michael Perscheid, Marcel Taeumel, and Robert Hirschfeld. 2014. Studying the Advancement in Debugging Practice of Professional Software Developers. In 2014 IEEE International Symposium on Software Reliability Engineering Workshops. 269–274. https://doi.org/10.1109/ISSREW.2014.36

[91] Benjamin H. Sigelman, Luiz André Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. 2010. Dapper, a Large-Scale Distributed Systems Tracing Infrastructure. (2010).

[92] Jacopo Soldani and Antonio Brogi. 2022. Anomaly detection and failure root cause analysis in (micro) service-based cloud applications: A survey. ACM Computing Surveys (CSUR) 55, 3 (2022), 1–39.

[93] Jacopo Soldani and Antonio Brogi. 2022. Anomaly Detection and Failure Root Cause Analysis in (Micro) Service-Based Cloud Applications: A Survey. Comput. Surveys 55, 3 (Feb. 2022), 59:1–59:39. https://doi.org/10.1145/3501297

[94] Hou Song. 2016. The Problem of Hbase Fast Reconnecting DataNode - Discovery and Analysis. Retrieved 2022-12-10 from http://housong.github.io/2016/hbase-reconnect-dn/

[95] Linhai Song and Shan Lu. 2014. Statistical Debugging for Real-World Performance Problems. In Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '14). Association for Computing Machinery, New York, NY, USA, 561–578. https://doi.org/10.1145/2660193.2660234

[96] Yongmin Tan, Hiep Nguyen, Zhiming Shen, Xiaohui Gu, Chitra Venkatramani, and Deepak Rajan. 2012. PREPARE: Predictive Performance Anomaly Prevention for Virtualized Cloud Systems. In 2012 IEEE 32nd International Conference on Distributed Computing Systems. IEEE, Macau, China, 285–294. https://doi.org/10.1109/ICDCS.2012.65

[97] Joseph Tucek, Shan Lu, Chengdu Huang, Spiros Xanthos, and Yuanyuan Zhou. 2007. Triage: Diagnosing Production Run Failures at the User's Site. In Symposium on Operating Systems Principles (SOSP) 2007 (SOSP '07). Association for Computing Machinery, New York, NY, USA, 131–144. https://doi.org/10.1145/1294261.1294275

[98] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. 2013. Apache Hadoop YARN: Yet Another Resource Negotiator. In Proceedings of the 4th Annual Symposium on Cloud Computing. ACM, Santa Clara California, 1–16. https://doi.org/10.1145/2523616.2523633

[99] Iris Vessey. 1985. Expertise in Debugging Computer Programs: A Process Analysis. International Journal of Man-Machine Studies 23, 5 (Nov. 1985), 459–494. https://doi.org/10.1016/S0020-7373(85)80054-7

[100] Haoyu Wang, Haiying Shen, and Zhuozhao Li. 2018. Approaches for Resilience against Cascading Failures in Cloud Datacenters. In 2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS). 706–717. https://doi.org/10.1109/ICDCS.2018.00074

[101] Lingmei Weng, Yigong Hu, Peng Huang, Jason Nieh, and Junfeng Yang. 2023. Effective Performance Issue Diagnosis with Value-Assisted Cost Profiling. In Proceedings of the Eighteenth European Conference on Computer Systems. ACM, Rome Italy, 1–17. https://doi.org/10.1145/3552326.3587444

[102] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael I. Jordan. 2009. Detecting Large-Scale System Problems by Mining Console Logs. In Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles - SOSP '09. ACM Press, Big Sky, Montana, USA, 117. https://doi.org/10.1145/1629575.1629587

[103] Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao, Yongle Zhang, Pranay U. Jain, and Michael Stumm. 2014. Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributed Data-Intensive Systems. In 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14). 249–265.

[104] Ding Yuan, Haohui Mai, Weiwei Xiong, Lin Tan, Yuanyuan Zhou, and Shankar Pasupathy. 2010. Sherlog: Error Diagnosis by Connecting Clues from Run-Time Logs. In Proceedings of the Fifteenth International Conference on Architectural Support for Programming Languages and Operating Systems. 143–154.

[105] YuQing. 2024. Happyfish100/Fastdfs. https://github.com/happyfish100/fastdfs

[106] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for in-Memory Cluster Computing. In Presented as Part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12). 15–28.

[107] Xiangyu Zhang, Neelam Gupta, and Rajiv Gupta. 2006. Locating faults through automated predicate switching. In Proceedings of the 28th International Conference on Software Engineering. 272–281.

[108] Xiangyu Zhang, Sriraman Tallam, Neelam Gupta, and Rajiv Gupta. 2007. Towards Locating Execution Omission Errors. In Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation. 415–424.

[109] Yingying Zhang, Zhengxiong Guan, Huajie Qian, Leili Xu, Hengbo Liu, Qingsong Wen, Liang Sun, Junwei Jiang, Lunting Fan, and Min Ke. 2021. CloudRCA: A Root Cause Analysis Framework for Cloud Computing Platforms. In Proceedings of the 30th ACM International Conference on Information & Knowledge Management (CIKM '21). Association for Computing Machinery, New York, NY, USA, 4373–4382. https://doi.org/10.1145/3459637.3481903

[110] Yongle Zhang, Serguei Makarov, Xiang Ren, David Lion, and Ding Yuan. 2017. Pensieve: Non-Intrusive Failure Reproduction for Distributed Systems Using the Event Chaining Approach. In Proceedings of the 26th Symposium on Operating Systems Principles. ACM, Shanghai China, 19–33. https://doi.org/10.1145/3132747.3132768

[111] Yongle Zhang, Kirk Rodrigues, Yu Luo, Michael Stumm, and Ding Yuan. 2019. The Inflection Point Hypothesis: A Principled Debugging Approach for Locating the Root Cause of a Failure. In Proceedings of the 27th ACM Symposium on Operating Systems Principles. ACM, Huntsville Ontario Canada, 131–146. https://doi.org/10.1145/3341301.3359650

[112] Yongle Zhang, Junwen Yang, Zhuqi Jin, Utsav Sethi, Kirk Rodrigues, Shan Lu, and Ding Yuan. 2021. Understanding and Detecting Software Upgrade Failures in Distributed Systems. In Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles. 116–131.

[113] Xu Zhao, Kirk Rodrigues, Yu Luo, Michael Stumm, Ding Yuan, and Yuanyuan Zhou. 2017. Log20: Fully Automated Optimal Placement of Log Printing Statements under Specified Overhead Threshold. In Proceedings of the 26th Symposium on Operating Systems Principles. ACM, Shanghai China, 565–581. https://doi.org/10.1145/3132747.3132778

[114] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chao Ji, Wenhai Li, and Dan Ding. 2021. Fault Analysis and Debugging of Microservice Systems: Industrial Survey, Benchmark System, and Empirical Study. IEEE Transactions on Software Engineering 47, 2 (Feb. 2021), 243–260. https://doi.org/10.1109/TSE.2018.2887384

[115] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chao Ji, Wenhai Li, and Dan Ding. 2022. Delta Debugging Microservice Systems with Parallel Optimization. IEEE Transactions on Services Computing 15, 1 (Jan. 2022), 16–29. https://doi.org/10.1109/TSC.2019.2919823