Going through the Life Cycle of Faults in Clouds: Guidelines on Fault Handling

Xiaoyun Li†, Guangba Yu†, Pengfei Chen*, Hongyang Chen†, Zhekang Chen‡
*†*Sun Yat-sen University*, ‡*Bizseer*
†{lixy223, yugb5, chenhy95}@mail2.sysu.edu.cn, *chenpf7@mail.sysu.edu.cn, ‡chenzhekang@bizseer.com

***Abstract*—Faults are the primary culprits of breaking the high availability of cloud systems, even leading to costly outages. As the scale and complexity of clouds increase, it becomes extraordinarily difficult to understand, detect and diagnose faults. During outages, engineers record the detailed information of the whole life cycle of faults (i.e., fault occurrence, fault detection, fault identification, and fault mitigation) in the form of post-mortems. In this paper, we conduct a quantitative and qualitative study on 354 public post-mortems collected in three popular large-scale clouds, 97.7% of which spans from 2015 to 2021. By reviewing and analyzing post-mortems, we go through the life cycle of faults in clouds and obtain 10 major findings. Based on these findings, we further reach a series of actionable guidelines for better fault handling.**

***Index Terms*—availability, cloud computing, faults, post mortems**

I. Introduction

Today, more and more enterprises move their applications from on-premise data centers to cloud systems so as to accelerate innovation, reduce costs, and increase agility. Meanwhile, a variety of techniques, such as scalable infrastructure [1], [2], distributed storage [3], [4], load balancing [5], [6], and cluster resource management [7]–[9], have emerged to optimize the cloud systems, enabling cloud systems to hold millions of services and customers. Cloud systems are thus expected with a high availability. Faults, the culprits of failures, break down the high availability of cloud systems and lead to service performance degradation, customer drain, and even a huge economic loss.

As the scale and complexity of cloud systems grow, fault handling in large-scale clouds is much more challenging than before. Faults may occur in any components in cloud systems, propagate to other components, and eventually impact user requests. For example, a fault [10] caused by an incorrect configuration change in a critical middleware during an upgrade may propagate to user-level services, and then manifest as elevated error rates of user requests. The complex topology and propagation path further prevent fault detection and diagnosis, exacerbating the impacts of faults.

Understanding faults in-depth is therefore of the primary importance for the study of fault handling. From womb to tomb, faults generally experience a life cycle involving four stages: (i) *fault occurrence* is the birth of a fault under a specific location and environment, (ii) *fault detection* is the procedure when manifestations of a fault are detected as anomalies and raise engineers' attention, (iii) *fault identification* is the procedure when engineers correlate clues and figure out the reasons, (iv) *fault mitigation* is the procedure when engineers adopt methods and tools to alleviate and eliminate faults.

① 7/4 *Azure* ②*DevOps – Service Outage – Mitigated*

Summary of Impact: Between ③02:26 am and 03:40 am UTC on 04 Jul 2020, customers using Azure DevOps in multiple regions may have observed ④ connectivity errors to DevOps services.

Preliminary Root Cause: We identified an inadvertent error with a ⑤ configuration change in the back-end service which caused the outage.

Mitigation: We ⑥ applied a configuration update which has fully mitigated the issue.

① ***Date***	2020-07-04
② ***Impacted service***	DevOps
③ ***Time to resolve***	74 minutes
④ ***Fault manifestation***	Connectivity error
⑤ ***Root cause***	Misconfiguration
⑥ ***Mitigation action***	Apply a configuration update

Fig. 1. A simple post-mortem example from Microsoft Azure. The top half is the raw post-mortem, the bottom half is the parsed content.

Aiming at protecting cloud systems and handling faults as soon as possible, SREs engaged in many cloud systems are 24/7 on-call to respond to failures just-in-time and resolve them promptly. They are required to record a whole process of fault handling in the form of post-mortems. Therefore, post-mortems are valuable data sources to review the life cycle of faults. Some cloud vendors make part of their post-mortems public online. Fig. 1 is an example of the post-mortems from Microsoft Azure on 04 Jul 2020 [11]. The failure is caused by an error configuration inadvertently changed by humans. Engineers mitigated this failure by changing the incorrect configuration. The raw unstructured post-mortem contains critical information such as failure date, impacted service, time to resolve (TTR), fault manifestation, root cause, and mitigation action. The above raw post-mortem can transform into the structured one shown in the table of Fig. 1.

Some prior works present an empirical studies of faults in cloud systems [12]–[16]. They provided lessons learned from headline news, public post-mortems, and enclosed internal data. However, they only focused on partial life cycle of faults. In this paper, our goal is to go through a whole life cycle of faults. We conduct an empirical study of 354 public post-mortems that occurred within 9 years from 2011 to 2021 in

three large-scale cloud providers (Google Clouds, Amazon Web Services, and Microsoft Azure). We ultimately guide our research with the following research questions (RQs):

- **RQ1: What are the root causes of faults and their distribution in clouds?**
- **RQ2: How do the faults happen in clouds?**
- **RQ3: How are the faults detected in clouds?**
- **RQ4: How are the faults propagated and identified in clouds?**
- **RQ5: How are the faults mitigated in clouds?**

Our study confirms some existing findings on fault studies, and further provides in-depth insights and guidelines on fault handling. In summary, this paper makes the following contributions:

- We collect 354 public post-mortems from three popular clouds spanning from 2011 to 2021. We transform unstructured post-mortems into structured ones and open-source them[1].
- We perform a qualitative and quantitative analysis on them from the perspective of the life cycle of a fault and obtain 10 findings, especially in the scope of fault occurrence, detection, identification, and mitigation.
- We reach some guidelines on fault handling based on the above findings, especially in chaos engineering, observability, and intelligent operations.

II. Related Work

A. Failure Studies

Failure studies have been widely conducted to understand the characteristics of failures in the wild. Table I presents a comparison between this study and other important related works from multiple perspectives.

First, failure studies have been conducted for different platforms, such as high-performance clusters [17]–[19], internet services [13], [20], game development [12], clouds [14]–[16], deep learning framework [21], [22], and so on. Among them, some studies [22]–[27] aimed to identify the root cause taxonomies of software faults and discussed one specific type such as concurrency bugs [24], [28], configuration error [29], performance bugs [30], and upgrade bugs [27]. Some studies [31]–[33] conducted an empirical study to understand the characteristics of hardware faults. Only a few studies [13], [21] considered both of them. This paper present a comprehensive study on both hardware faults and software faults in clouds.

Second, failure studies analyzed parts of the life cycle of faults. Sillito et al. [15] have qualitatively analyzed thirty incidents including fifteen public post-mortems and fifteen interviews of experienced operating engineers. They discussed about how failures happened and incident responses including fault detection and fault mitigation. Liu et al. [14] carefully studied hundreds of high-severity incidents in production and provided insights such as identified root causes and related mitigation from the industry. Gunawi et al. [13] collected headline news and public post-mortems of 597 unplanned outages in 32 popular Internet services within a 7-year span.

[1]https://github.com/IntelligentDDS/Post-mortems-Analysis

TABLE I
Comparison with related works

Related work	FO	FD	FI	FM	OS	Targets
Schroeder [32]	✓	×	×	×	×	h
Wang [33]	✓	×	✓	✓	×	h
Gunawi [23]	✓	×	×	×	✓	s
Banerjee [20]	✓	×	×	×	×	s
Washburn [12]	✓	×	×	×	✓	s
Gunawi [13]	✓	×	×	✓	✓	h&s
Sillito [15]	✓	✓	×	✓	✓	s
Liu [14]	✓	×	×	✓	×	s
Chen [16]	✓	✓	×	✓	×	s
Zhang [27]	✓	×	×	×	✓	s
Chamberlin [34]	✓	×	×	×	✓	s
Ours	✓	✓	✓	✓	✓	h&s

The seven columns are (1) related work, the last name of the first author, (2) FO: fault occurrence, (3) FD: fault detection, (4) FI: fault identification, (5) FM: fault mitigation, (6) OS: whether the studied datasets is open-sourced, (7) Targets: studied objective, h means hardware faults and s means software faults.

They answered why outages occur in cloud computing environments and provided correlation analysis between root causes and fix actions, lacking deep reviews on fault detection and identification. Chen et al. [16] conducted a comprehensive empirical study of incident management practices at Microsoft including incident root causes, fault detection and mitigation. However, the enclosed data sources block further researches.

Compared with other related works, this paper collects public post-mortems and conducts a comprehensive study of faults on the whole life cycle of faults including fault occurrence, fault detection, fault identification and fault mitigation. The most similar related work is [13]. Our work collected public post-mortems, and 97.7% of that spans from 2015 to 2021, while [13] analyzed post-mortems mostly before 2015. Our work provides a full analysis of the life cycle of faults with a more detailed fault mitigation analysis and actionable fault handling guidelines especially. The data sources of this study and analyzed results are public online and can be reused for further research.

B. Incident Management

Incident management plays a significant role in an online cloud service. Lou et al. [35], [36] carried out an experience report on applying software analytics to incident management of large-scale online systems in the real world. Since it is important to be able to automatically assign an incident report to a suitable team, Chen et al. [37] presented an empirical study of incident triage. Chen et al. [38] and Gao et al. [39] are devoted to addressing improve the effectiveness of incident triage. Concerning incident mitigation, Jiang et al. [40] proposed an automated troubleshooting guide (TSG) recommendation approach, by leveraging the textual similarity between incident description and its corresponding TSG using deep learning techniques. Chen et al. [16] also presented an AIOps framework towards intelligent incident management.

III. Methodology

To understand the whole life cycle of a fault, an effective way is reviewing public post-mortems of clouds. Unstructured post-mortems are collected from public websites and then

manually transformed into structured ones. This section will describe the methodology of this process.

A. Data Collection

We consider three popular and representative cloud systems as target systems, which rank top three in Gartner's study [41]: Amazon Web Services (AWS), Microsoft Azure, Google Clouds. The details of collected datasets are shown in Table II. Cloud vendors publicly provide post-mortems on the status dashboard website, but only a small portion among them is worth analysis. So we collect 354 valuable post-mortems from them as raw datasets within a 10-year spanning from 2011 to 2021, including 14 AWS incidents, 242 Azure incidents, and 98 Google Clouds incidents. Noting that all of these data are collected and processed only for research purpose. The credentials of them are reserved by cloud vendors.

TABLE II
THE DETAILED INFORMATION OF COLLECTED DATASETS

Datasets	#incidents	source link	labels
AWS	14	[42]	aws-#
Azure	242	[11]	azure-yymmdd-#
Google Clouds	98	[43]	google-@services-#

B. Manual Labelling

Our goal is to conduct quantitative and qualitative analysis on collected data, so we first transform unstructured post-mortems into structured ones. Such transformation is also called a coding procedure. Following an open coding procedure [44], the coding process is to extract quantitative variables (e.g., impacted service, fault propagation path, root cause, time to resolve, time to detect, time to mitigate, etc.) from observations (post-mortems here).

Six experienced engineers were engaged in labeling the collected data. We take three rounds spanning five months to code the raw datasets. The first round takes around two months to sample tiny representative datasets to cover root causes as much as possible. After multiple trials of discussion and modification on the coding principle, we outline the coding methodology and ensure that all authors are consistent with it. The second round takes almost two months to code all post-mortems. Each author independently codes the post-mortems assigned to him/her and records the corner cases. The final round is cross-validation spanning one month. Three authors verified the post-mortems coded by others to guarantee correctness and consistency. The authors handled the corner cases through further discussions.

IV. THE LIFE CYCLE OF FAULTS

From womb to tomb, faults in cloud systems follow a general life cycle: fault occurrence, fault detection, fault identification, and fault mitigation. We first introduce an overview of the above four stages and then go through the whole life cycle of a fault.

A. An Overview of Time Spans Across Different Stages

Time spans across different stages are golden metrics to evaluate the importance of faults and the effectiveness of methods to handle faults in cloud systems. Fig. 2 presents the whole life cycle of a fault in cloud systems. Four stages of *Time To X (TTX)* are further explained as below:

- *Time to Detect (TTD)* is the time cloud systems take to detect a fault.
- *Time to Identify (TTI)* is the time cloud systems take to identify the root causes after detecting a fault.
- *Time to Mitigate (TTM)* is the time cloud systems take to mitigate faults. Sometimes, mitigating actions are applied without an accurate localization of root causes.
- *Time to Resolve (TTR)* is the time cloud systems take to resolve a fault. It includes the time spent on detecting a fault, identifying the root cause, and mitigating the issue.
- *Time To Failures (TTF)* is the time span between two failures, namely the uptime in cloud systems.

TABLE III
MEAN TIME SPANS ACROSS DIFFERENT STAGES IN THE LIFE CYCLE

-	MTTD	MTTI	MTTM	MTTR
Time	16.9 m	77.8 m	304.2 m	572.8 m

System availability can be considered as the proportion of uptime to runtime, hence the defined equation in Eq. 1:

$$Availability = \frac{TTF}{TTF + TTD + TTI + TTM}. \quad (1)$$

The increase of *TTF* or the decrease of any one of {*TTD, TTI, TTM*} can level up the system availability. We conduct a statistical study of *Mean Time To X (MTTX)* on our datasets, and the mean results are shown in Table III. We do not show *MTTF* since the collected datasets across cloud systems are not continuous.

Fig. 3 shows the cumulative frequency functions (CDF) of *TTX*. Detecting faults in one minute, localizing faults in five minutes, and mitigating them in ten minutes are the ideal objectives in DevOps [45]. Detecting faults in time is the key to fault handling. But only 15.7% of faults are detected in one minute. Localization time is also far from the objective, where only 14.0% reaches the goals. Mitigating faults in ten minutes only accounts for 1.9%, which means there is large progress to do to achieve the ideal objectives.

B. RQ1: What are the root causes of faults and their distribution?

> Expertise is gained by investigating why a system doesn't work.
>
> —— **Brian Redman [46]**

In our study, we consider the root cause as the direct reason leading to faults. Prior works like [13], [14], [23] have analyzed the software-related and hardware-related faults using an unstacked root cause taxonomy, while this study provides a layered taxonomy from a different perspective. The first layer is about the root cause scope. Since our study targets cloud systems, we consider root causes in software as *internal causes* while root causes out of software as *external causes*

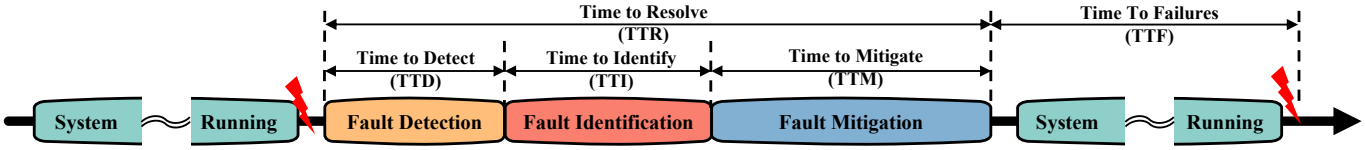
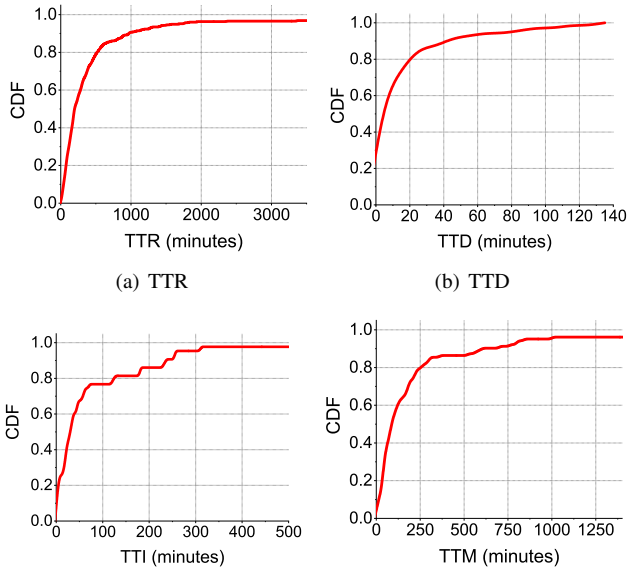


Fig. 2. The life cycle of a fault

TABLE IV
A DETAILED DESCRIPTION AND STATISTICS OF ALL ROOT CAUSES

Layer-1	Layer-2	Count	%	Description	Cases
Internal Causes	Misconfiguration	112	31.6	Refers to those software bugs caused by incorrect configurations.	azure-20200704-1
	Code change	47	13.3	Refers to those faults caused by short-term code change.	google-bq-19002
	Payload flood	28	8.0	Refers to those software bugs generating overwhelming requests.	azure-20170325-1
	Resource contention	11	3.1	Means that the shared resources are locked or congested.	aws-2
	Exception handling	7	2.0	Means that the code to handle exception fails.	azure-20200928-1
	Incompatibility	6	1.7	Means that upgraded codes are incompatible with other components.	google-gce-15065
	Others	69	19.5	Root causes cannot be assigned to any other root causes.	azure-20161207-1
External Causes	Hardware failures	60	17.0	Refers to hardware failures such as CPU, disk, network and power.	azure-20170321-1
	Insufficient resource	33	9.3	Insufficient resources (e.g., CPU, disk) are provisioned.	google-gae-17005
	Excessive flow	22	6.2	Excessive amount of requests are sent by users.	google-gcps-19001
	Third-party failures	10	2.8	Refers to the third-party failures such as dependent services.	azure-20191212-1
	Component removal	7	2.0	Refers to those faults caused by inadvertently component removal.	aws-6
	Others	5	1.4	Other external causes with vague description.	auzre-20200701-1
Unknown	/	43	12.1	Engineers can not have a definitive root cause.	azure-20170915-1



(a) TTR (b) TTD

(c) TTI (d) TTM

Fig. 3. The CDF of four stages in the life cycle of faults

in the first layer. Other faults with vague RCAs are labeled as *unknown* root causes. We present detailed descriptions and the statistics of taxonomies of all root causes in Table IV. Next, we illustrate the layered taxonomies of root causes layer by layer. Noting that the percentage in this section is calculated following the equation $Percentage(\%) = \frac{\#type}{\#faults}$. A fault may be caused by multiple root causes, consequently the sum of percentages of all root causes is more than 100%.

1) Internal Causes (IC): Internal causes, which mainly refer to software bugs, are responsible for about 79.1% of the faults. Inspired by [13], [23], we categorize software bugs into seven taxonomies including *misconfiguration*, *code change*, *payload flood*, *resource contention*, *exception handling*, *incompatibility* and *others*. A detailed description of software bugs is presented in Table IV. The most common root cause is misconfiguration (31.6%) as expected since configurations are important components in cloud systems while the least common root cause is incompatibility (1.7%). Compared with previous taxonomies of software bugs, the label *code change* is additionally added to our classification. As the complexity and flexibility of software systems increase, cloud systems are always in the active development and maintenance states, which means change is very common. Industrial engineers have disclosed that their systems have tens of thousands of changes a day. With it, *code change* accounts for 13.3% among software-related faults. Other root cause like *payload flood*, *resource contention*, and *exception handling*, accounts for 8.0%, 3.1%, 2.0%, respectively, which can not be ignored.

2) External Causes (EC): Those faults caused by factors outside cloud software systems are attributed to *external causes*. In our study, *external causes* cover 41.0% of the faults, originating from (i) hardware failures, (ii) insufficient resources, (iii) excessive flow, (iv) third-party failure, (v) component removal and (vi) others. We can observe that the type *hardware failures* (17.0%) is the most common external cause in practice. Facing the dynamically changing workload, the type *insufficient resources* (9.3%) also leads to rejection to requests, which largely affect customers. *Excessive flow* (6.2%) is possibly generated by abnormal user behaviors or a fixed pattern like morning peak. Other root causes like *third-party failures*, *component removal* only occupy 2.8% and 2.0%, respectively.

Finding 1: *Regarding the root cause distribution, 79.1% and 41.0% of faults are caused by internal and external causes, respectively. The most common causes in internal causes and external causes are misconfiguration and hardware failure, accounting for 31.6% and 17.0% of faults, respectively.*

3) Unknown Causes (Un): In some cases, engineers cannot determine the definitive root causes [azure-20170915-1]. Such cases are considered as unknown root causes.

4) Multiple Root Causes: In a dynamic and complex system, one explicit fault may be due to the combination of multiple root causes. We present the distribution of root causes number in Table V. We can observe that 23.2% of studied cases result from more than one root causes, which is a non-negligible portion in practice. The number of root causes in a fault is at most four in our study. We identify those cases caused by multiple root causes and summarize three typical forms: (i) Only one root cause may not cause failures, but under a rare condition, the combination of multiple root causes together results in failures. For example, some software bugs only manifest under specific configuration settings [azure-20180820-1]. (ii) One root cause derives from another root cause like a chain reaction. For example, a sudden increase in requests led to quota exhaustion. The high load of requests triggered an issue in the scheduling system [google-bq-19003]. (iii) Incorrect fault handling generates new faults. For example, an incorrect configuration change was inadvertently applied when handling a fault [google-bq-18036].

TABLE V
THE DISTRIBUTION OF ROOT CAUSES NUMBER

# Root Causes	1	2	3	4
# Count	272	66	15	1
%	76.8	18.6	4.2	0.3

Finding 2: *23.2% of studied faults are caused by more than one root causes, which is a non-negligible portion in practice.*

C. RQ2: How Do the Faults Happen?

> If a human operator needs to touch your system during normal operations, you have a bug.
>
> —— **Carla Geisser, Google SRE [46]**

From the perspective of the life cycle of a fault, we first need to answer the question: *How do the faults happen?* Besides the distribution of root causes, we focus more on the other important information: *What are the ongoing procedures when the faults occur? Are the faults related to human error? Why do these faults escape fault-tolerant mechanisms?*

1) Ongoing Procedures: To understand the ongoing procedures when the faults occur, we aggregate similar ongoing procedures and finally list three clustered ongoing procedures below. Fig. 4 shows the ongoing procedures distribution by the identified layer-1 root causes.

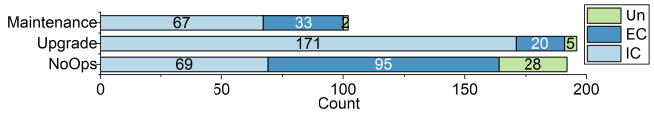


Fig. 4. The distribution of ongoing procedures

Upgrade and **Maintenance**, which are common ongoing procedures in those cloud systems under active development and operations, are responsible for 58.8% of all faults. Some cases [aws-1, gcs-18005] may go through multiple ongoing procedures. It is reasonable that changes under update and maintenance procedures bring perturbations to the running cloud systems.

Upgrade (42.9%). Indeed, an upgrade in cloud systems goes through extensive standard testing, but most faults during an upgrade do not manifest themselves in the testing and pre-production environment [azure-20180726-1, azure-20201019-1]. The production environment is different from the pre-production environment in complex interaction between components, unexpected traffic patterns, inconsistent configurations invocation, and so on. Such a gap between the production environment and testing environment may lead to failure during system upgrades. Furthermore, some cases [azure-20200928-1] show that faults in Safe Deployment Practice (SDP) make a bad deployment broadcast to many regions, exacerbating the failures. During an upgrade procedure, internal causes especially misconfiguration are the most common root causes and covers almost 46.0% faults. Also, 84.8% of faults caused by misconfiguration occurred during upgrades and maintenance. It indicates that a misconfiguration often occurs accompanied by an upgrade.

Maintenance (20.0%). Faults may occur both during routine maintenance and fault mitigation. First, faults occur during routine maintenance, like hardware maintenance (e.g., fire suppression system [azure-20170929-1], fiber [azure-20181024-1]) and software maintenance (e.g., capacity adjustment [azure-20170518-1]). Second, an error during mitigation exacerbates the faults. [google-gcdf-16001] is a typical example. During the mitigation of a lower impact performance issue, engineers introduced an erroneous configuration to pipeline orchestration, which caused validation within the orchestration component to reject all requests.

Normal Operation (NoOps), meaning that no intervention is introduced to cloud systems when system encounters faults, takes 42.7% of all faults.

Finding 3: *58.8% of faults occur during changes such as upgrades and maintenance. Faults during an upgrade may be caused by (i) the gap between production and pre-production environment in configuration, workload pattern, unusual invocation, and (ii) defects in the deployment framework. Faults during maintenance may be caused by routine maintenance and fault mitigation.*

We then analyze the relation between ongoing procedures and root causes. From Fig. 4, we can observe that around

84.7% (238) of all faults during system upgrades and maintenance are due to internal causes. Faults due to external causes account for 56.3% (95) of all faults during NoOps, and the portion is larger than the one during upgrade and maintenance. It indicates that once a qualified version of the system is deployed successfully and executed for a few days, the system is likely to run continuously and normally without changes of the external environment. External causes such as hardware failures (e.g., fiber cut, power outage), excessive flow, and insufficient resources cause faults during NoOps. It indicates that changes of external environments (e.g., workflows, dependent party, hardware) greatly affect the internal stability of cloud systems. Besides, the combination of two conditions fires some faults during normal operations.

Finding 4: *84.7% of faults during system upgrades and maintenance are due to internal causes. 56.3% of faults during normal execution are due to external causes, which is larger than the one during changes.*

2) Human Errors: Manual intervention is inevitable in the Development and Operations (DevOps) of cloud systems. According to our statistics, human errors result in 7.6% of all faults. Compared to the statistics results of 19-36% in 2003 [47], human errors are largely reduced, further confirming the results in [13]. Due to the difference in datasets, it is acceptable to obtain a biased result with 4% of all faults caused by human error. It indicates that the introduction of intelligent operations helps to reduce human errors. Human errors are mainly manifested in two formats: (i) Inadvertent error. For example, engineers in [azure-20180220-1] inadvertently recycled the power on the scale unit in production. (ii) Inefficient management. Cloud vendors have shared their experience publicly [48]. Engineers in team X had sent an email with rollout messages to notify other teams, but engineers in the other team did not notice it and made conflict operations, leading to a service outage.

Finding 5: *Faults due to human errors take up 7.6% of all faults. The introduction of intelligent operations helps to reduce human errors.*

3) Ineffective fault tolerance: Fault-tolerant mechanisms provide the ability to ensure that the system continues operating without human-involved interruption when one or more of its components fail, guaranteeing the availability of cloud systems. But what kinds of faults do escape fault-tolerant mechanisms? We zoom in these cases and provide three representative scenarios: (i) Failover to backups may encounter faults. In the case [azure-20161211-1], software issues on network routers caused routing calculations to take longer than expected during a fiber issue. The path computation slowdown caused traffic to be dropped instead of moving to the redundant fiber path. In the case [aws-10], software issues in failover mechanisms prevent the automatic failover. (ii) Potential faults may be hidden in backups. In the case [azure-20181126-1], when traffic is rerouted to the backup, an issue that occurred in the backup path resulted in traffic congestion [azure-20181126-1]. Also with low probability, multiple backups may concurrently encounter faults [azure-20200120-1]. The availability of backups may be ignored in practice since they are seldom used. (iii) Small performance issues are covered until there are no enough available replicas. In the case [azure-20211216-1], due to an internal cause, the problematic rollout was not stopped until all redundant endpoints were impacted.

Finding 6: *The failure of fault tolerance may be caused by unsuccessful failover, faulty backups, and potential performance issues.*

D. RQ3: How are the Faults Detected?

> Ways in which things go right are special cases of the ways in which things go wrong.
>
> —— **John Allspaw, Google SRE [46]**

Fault detection is the primary step of fault handling that possibly affects following handling operations. Detection methods in practice can be divided into two parts: automatic detecting and manual detecting. Rule-based monitoring methods are simple yet effective automatic detecting methods, so they are widely used in practice. For example, raising an alarm when system resource usage exceeds a preset threshold or detecting anomalous patterns like [49]. With the prevalence of machine learning, data-driven fault detection methods such as [50], [51] attract much attention but they remain largely in the realm of research. In this section, we attempt to answer how are the faults detected and dig up the corresponding information about fault detection in practice.

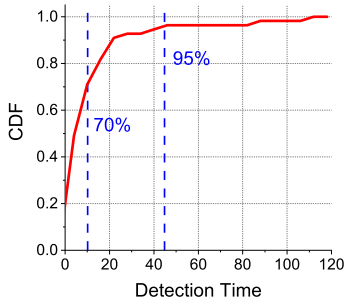


Fig. 5. The CDF of automatic detection time

In our datasets, only 38.7% post-mortems explicitly record how engineers detect faults. According to our investigation, around 93.4% of cases with detection records utilized automatic monitoring tools such as health monitoring and internal telemetry tools to detect anomalies. We can indicate that automatic monitoring methods such as rule-based and data-driven methods play an important role in practice. We present the CDF of automatic detection time in Fig. 5. Around 70% of faults can be automatically detected in 10 minutes and 95% of faults can be detected in 47 minutes. While the other cases using manual detection generally execute an upgrade or maintenance. It is acceptable since engineers are engaged in

monitoring system states carefully during changes. There are also some cases detected by analyzing user reports in practice.

> **Finding 7**: *Almost 93.4% recorded cases utilized automatic detection methods, 70% of which are detected in 10 minutes.*

E. RQ4: How are the Faults Propagated and Identified?

> If at first you don't succeed, back off exponentially.
>
> —— **Dan Sandler, Google SRE [46]**

Fault identification is to first localize faulty components or metrics, and then identify the root cause of this fault. Due to the diversity of causes, it is quite challenging to identify a specific root cause of fault automatically. So existing research methods [38], [52] in fault identification tend to localize faulty components or metrics automatically and report to engineers. Then engineers are engaged in identifying root causes manually.

Cloud systems consist of hundreds of nodes and inter connections, leading to the highly complex topology. Worsemore, cloud systems tend to generate faults logically and geographically distributed. The impacts of a fault are propagated along with the connections, leading to a more difficult fault localization.

Understanding trivial and non-trivial propagation patterns benefits the research of fault localization. We attempt to refine propagation patterns from post-mortems via manual reading and comprehension. After several rounds of discussion and verification, we conclude a high-level abstraction and deliver a better demonstration of insights into the propagation path. The vertical direction can be divided into hardware, supporting infrastructure, operating systems, virtual machines, and applications (services). Supporting infrastructure contains servers used as different roles like cache servers, job servers, and so on. The horizontal direction is labeled by different roles in cloud systems such as frontend, backend, storage, network, and middleware. All network-related devices like routers, switches, load balancers, and network-related components like DNS, VPN, and network control plane are grouped into Network. Other components like dependent services in cloud systems but not exposed to users are categorized into middleware.

TABLE VI
THE DISTRIBUTION OF PROPAGATION LENGTH

Length	1	2	3	4	5	Average
Count	42	186	96	29	1	2.3
%	11.9	52.6	27.1	8.2	0.3	-

We analyze the distribution of propagation length and show the results in Table VI. We can observe the interesting results that around 88% of all faults in our study are propagated to other components in cloud systems. It indicates that components with fault manifestation maybe not the root cause components. The propagation length is at most five in our study (cases listed below). Most faults go through two (52.6%) or three (27.1%) components before terminating. On average, faults passed through 2.3 components in our study. It is interesting but also easy to understand that the fault manifested components are usually not the ones with errors.

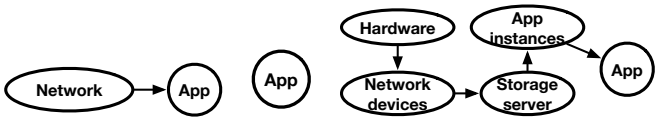
TABLE VII
THE STATISTICS OF FLOW IN PROPAGATION PATH

From \ To	app	fe	be	mw	vm	os	nw	str	si	hw
app	30	-	-	-	-	-	1	3	-	-
frontend	2	-	-	-	-	-	1	-	-	-
backend	38	1	14	9	-	-	1	3	3	1
middleware	45	1	-	1	2	-	1	6	1	-
vm	6	-	-	-	-	-	1	1	-	-
os	4	-	1	-	-	-	-	-	-	-
network	65	-	3	3	6	1	20	17	2	-
storage	73	-	2	8	-	-	3	12	1	-
su-infra	9	-	-	3	-	-	1	3	-	-
hardware	10	-	-	-	-	-	13	19	2	1
third-party	4	-	-	-	-	-	1	1	-	-

The value in table is the count of flow of "from" to "to", and "-" means that there are no flow between them. Abstraction in columns is the abbreviation of the row name. For example, "fe" refers to "frontend". "su-infra" refers to "supporting infrastructure". "third-party" is not included in the columns since no faults are propagted to third-party.

Regarding the distribution of flow patterns in propagation paths, we count the edges between two abstracted components and present the results in Table VII. By nature, faults are propagated from the bottom, like from hardware to the network, from middleware to application. So we can observe that most of the destination in propagation paths is the application layer. Besides, loops (i.e., the diagonal elements in Table VII) exist in propagation path. There are 30 cases propagated inside "app" and 20 cases propagated inside "network". However, there are also some backward propagation flows like from "app" to "network". In the case [google-BQ-18037], a new release of the BigQuery API introduced a software defect that caused the API component to return larger-than-normal responses to the BigQuery router server. The congestion further affected other services.

In addition, we show top 2 frequent fault propagation paths in Fig. 6(a) and Fig. 6(b). Case 1 (11.0%) are propagated from "network" to "app". Network in cloud systems is responsible for the connection between user-oriented services and the underlying systems. Once an error occurs in the network, it is highly possible to propagate the faults to the application layer. Case 2 (9.0%) only contains one node "app", which means that services in the application layer encounter errors with no propagation. We find that such cases are probably caused by software bugs in the application layer or unknown root causes. Unknown root causes left the fault manifestations to be found only on the application layer.



(a) Case 1 (b) Case 2 (c) Case 3

Fig. 6. Three special cases in fault propagation paths

We deliver a special case [#aws-1] with the propagation length 5 in Fig. 6(c). In this case, severe weather resulted in a power outage ("hardware"). When restoring power, DNS failed

("network devices"). Such failures led to data loss in storage servers ("storage server"), and impacted "app instances" and finally the supporting "app".

Finding 8: *On average, faults passed through 2.3 components in our study. The most common path in fault propagation is from network to application.*

F. RQ5: How are the Faults Mitigated?

> Besides black art, there is only automation and mechanization.
>
> —— **Federico García Lorca, Spanish poet and playwright [46]**

Mitigation actions are adopted to handle faults. After figuring out the stories behind the faults and relevant detection and localization methods, we desire to answer how are the faults mitigated and also to uncover frequently used mitigation actions in practice.

TABLE VIII
THE STATISTICS OF MITIGATION ACTIONS

Action Items \ Metric	Count	%	TTM (minutes)		
			Mean	Std	Med
Self-healing	27^{-}	7.6^{-}	463	865	70^{-}
Rollback	82	23.2	156^{-}	194^{-}	91
Re-action	41	11.6	275	609	86.5
Replacement	113*	32*	193	215	107
Isolation	87	24.6	241	359	125
Flow control	72	20.3	274	506	116
Scaling	47	13.3	378	584	163.5*
Fixing	90	25.4	601*	1209*	220
Others	81	22.9	1124	1761	94

The maximum and minimum along with one column except types *Others* are tagged by "*" and "$^{-}$".

1) Mitigation Actions.: We totally identify nine types of mitigation actions: (i) self-healing, (ii) rollback, (iii) re-action, (iv) replacement, (v) isolation, (vi) flow control, (vii) scaling, (viii) fixing, and (ix) others. The statistics of mitigation actions and their related TTM in our study are listed in Table VIII.

Self-healing. Three kinds of faults can be self-healed requiring no human-involved intervention, accounting for the minimum portion 7.6% of all faults. First, overloaded services return to healthy once the load is ingested. For example, the rapid pattern change resulted in a fast reconfiguration to adapt to these changes, generating a long modification queue. As the backlog of network configuration changes was automatically processed, this issue was resolved without human intervention. [google-gce-15057]. The other circumstance is inadvertent restarting or reinstalling. Cloud systems return to healthy once the operation had completed [google-gcnet-18019, azure-20180220-1]. The third kind is adopting automatic recovery mechanisms. Sometimes, the issue was self-healed by automatic recovery mechanisms without a definitive root cause [azure-20170517-1].

Rollback. If a fault occurs during some procedures such as rollouts, new feature deployments, code changes, and configuration changes, the naive but effective way to mitigate faults is to rollback [google-gce-16015]. According to our statistics, rollback spends the least time mitigating faults.

Re-action. The next mitigation action is the redo process such as restarting and redeploying components, but with a high risk to wreck the availability of cloud systems [google-gcps-17001, google-bq-19003]. In distributed systems, the leader election module is to designate a single process as the organizer of some tasks distributed among several computers. Therefore, forcing leader election is also common in distributed cloud systems to mitigate the faults of leader conflicts [google-gcnet-19020].

Replacement. Replacing the faulty component is a prompt action to mitigate faults, which is the most common mitigation action. One of these is to replace software components. With a high coverage of misconfiguration, replacing the configuration with the correct one is a fast way to handle them [azure-20200604-1, azure-20200221-1]. Clearing the full load component is also a desirable approach. For example, backlogs were generated as the overloaded traffic grows. Clearing backlog was feasible to re-enable services [google-gcic-20003]. The other one is replacing the hardware components or failover. A failure of a network component temporarily reduced network capacity, which was resolved by the replacement of the faulty hardware [google-gae-15023].

Isolation is an extraordinarily effective means when services get failed. First, it is feasible to make faulty components offline. One is to remove faulty hardware components [azure-20190110-1], and the other is to make services offline such as removing mechanisms [google-gcic-20005] and disabling services [google-csql-17017]. For example, the Denial of Service (DoS) protection mechanism was triggered by a high rate and volume of retries policy in [google-gcic-20005], causing traffic congestion. Removing the DoS protection mechanism could somehow mitigate the impact symptom. The second is to cancel tasks such as pausing faulty rollouts [google-gcnet-19007] or maintenance [azure-20181129-1], stopping buggy migration [google-gae-15025], and stopping erroneous workflow [azure-20200224-1].

Flow control is required when encountering overloaded circumstances. General mitigation actions to mitigate overloaded services are to limit rate, reduce traffic, or redirect traffic. We take [google-gcps-19001] as an example. Here the faults were caused by the overload protection mechanisms which rejected some incoming requests and delayed the processing of others. Therefore, engineers introduced a rate limit on the requests to mitigate the issue. Reducing traffic [google-gae-19007] and redirecting traffic [google-gce-17005] follow similar operations as the rate limit.

Scaling. Scaling up is to relieve the services situation of lack of capacity. Mitigation actions like scaling out and increasing capacity are usually leveraged to mitigate faults. In [google-bq-18037], increasing the network capacity of the service router server allowed API traffic to resume normally. Others like acquiring additional storage capacity [google-csql-17012] and granting additional quota [google-gcdc-19001] could also ease the burden of services.

Fixing is a regular step to eliminate faults. Patching the corresponding bugs is a common mitigation action in some cases [azure-20161130-1, google-gce-18012]. Other cases adopt roll out builds to eliminate the incompatibility of the current build

and previous build [google-gcd-19006, azure-20190501-1].

Others. Those mitigation actions are not clear and cannot be assigned to above mitigation actions.

> **Finding 9**: *The most and the least common mitigation actions are Replacement and Self-healing. Rollback spends the least time to mitigate faults while Fixing takes the most time on average.*

2) Relation between Root Causes and Mitigation Actions: Fault mitigation needs the right mitigation actions. Usually, engineers adopt mitigation actions according to the specific root causes following the guidance in SRE runbooks. To gain a better understanding of mitigation actions in practice and provide guidelines to engineers, we investigate whether there are detailed and interpretable relations between root causes and mitigation actions. As multiple root causes and mitigation actions may exist in collected datasets, we first unfold them using the Cartesian product.

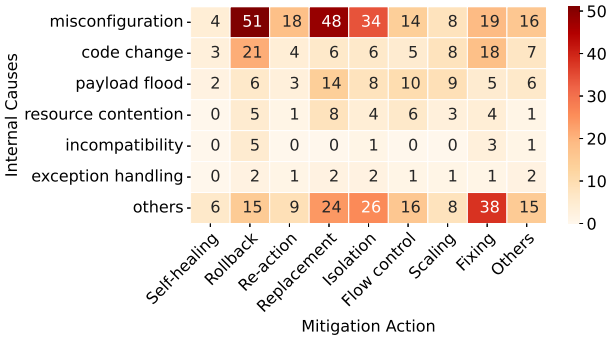


Fig. 7. The relation between mitigation actions and internal causes

The relation between mitigation actions and *internal causes* is shown as a heatmap in Fig. 7. The row refers to the type of *internal causes* and the column refers to the type of mitigation actions. For example, 51 in the first row means that 51 faults caused by misconfiguration are mitigated by *Rollback*. We can observe that in most root causes, some mitigation actions are quite conspicuous such as *Rollback* and *Replacement* for *misconfiguration*, *Rollback* and *Fixing* for *code change*. But for other faults due to root causes like *resource contention* and *exception handling*, different kinds of mitigation actions are evenly applied in fault mitigation.

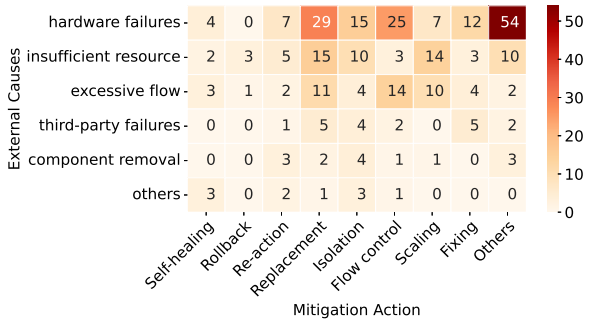


Fig. 8. The relation between mitigation actions and external causes

We also analyze the relation between mitigation actions and *external causes* in Fig. 8. We can observe that the type *Replacement* occupies 29 cases to mitigate hardware failures. It is interpretable since replacing the faulty hardware with the healthy one can mitigate faults. Also, not only one mitigation action is taken to mitigate faults. For example, to mitigate those faults caused by insufficient resources, engineers first isolate the faulty component, then scale out the capacity to hold more requests. Analogy to internal causes, there are no salient mitigation actions shown in some external causes (e.g., *component removal*, *third-party failures*).

> **Finding 10**: *Some root causes such as misconfiguration, code change, hardware failures, insufficient resource show a strong correlation with mitigation actions. But some root causes such as resource contention, exception handling, component removal, third-party failures are not.*

V. Guidelines on Fault Handling

The above sections introduce the observations from the perspective of the life cycle of faults. We passively analyze results and indicate their possible reasons. Next, we provide insights proactively by concluding guidelines from an empirical study on hundreds of post-mortems.

A. Guidelines on Chaos Engineering in Clouds

To verify the system's capability under stress, chaos engineering is proposed as a discipline of experimenting on systems in production. Mocking possible faults proactively enables cloud engineers to build timely identification mechanisms and feasible mitigation actions.

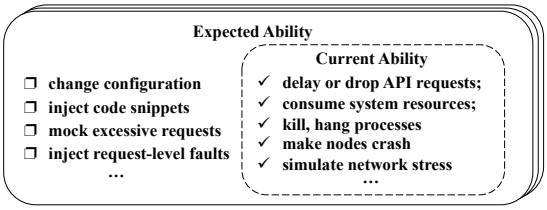


Fig. 9. Gap between current and expected ability of chaos engineering tools

The distribution of root causes in Sec. IV-B provides guidelines for simulation of fault types in production. The existing chaos engineering tools [53]–[55] provide ways to delay or drop API requests; consume system resources; kill, hang processes or make nodes crash; simulate network stress, and so on. However, the ability of the existing chaos engineering tools is far from satisfactory in face of the complex environment in modern cloud systems. Fig. 9 visualizes the gap between the ability of current chaos engineering and the expected ability in the real world. Based on the Finding 1, chaos engineering tools are expected to implement the following scenarios:

- Injecting configuration change at runtime. It is expected to change configurations in files or codes at runtime. The majority of configuration changes need to apply or restart.

- Injecting code snippets at the specific location at runtime. Although chaosblade [54] provides the way to inject code snippets in Java by modifying their bytecodes, the progress to inject faults in other programming languages like GoLang, and C++ is still at a preliminary stage. This ability can simulate faults caused by *code change*, *payload flood*, *exception handling*, *incompatibility*.
- Mocking excessive requests to systems. It is expected to replay overwhelming real-world user requests to systems based on historical user behaviors.
- Injecting request-level faults. It is expected to control the impacted scope of a fault with fine-granularity, reducing the impacts to real users. Besides, request-level faults injection helps engineers analyze manifestations of faults.

The exploration of RQ2 uncovers some unexpected scenes where faults happen. Based on Findings 3-6, chaos engineering tools are expected to implement under more scenarios:

- Testing and deployment framework. To disrupt the controller of testing and deployment framework and verify the robustness, and effectiveness of these mechanisms.
- Failover mechanisms. To break down the critical router such as modifying the target prefixes, and mock the scenario that traffics are not rerouted successfully.
- Backup components. To destroy backup components both in software and hardware and check whether they can execute in a proper state.
- Operation framework including monitoring and mitigating. To wreck system and verify whether monitoring and mitigating tools still work under destructed systems.

B. Guidelines on Observability in Clouds

Observability refers to the ability to understand and explain the state of a complex system based on the outputs of the system [56]. Cloud systems are distributed by nature and thus introduce complex and dynamically changing service dependencies. So it is imperative to make cloud systems observable for effective debugging and diagnosis. Typically in cloud systems, observability contains three fundamental components: metrics, logs, and traces. Towards better observability, we provide three guidelines based on the observations from manual comprehension:

- It is suggested to ensure the integrity of observable data. The loss of monitoring data may lead to miss alerts, which suppresses further fault handling.
- It is suggested to provide layered observability of inter-components and intra-components. Based on Finding 8, faults may propagate across multiple components. Tracing [57]–[59] is particularly effective to handle such cases and attaches a "trace id" to a request so as to identify the whole execution path of the request among inter-components and intra-components.
- It is suggested to control the granularity of observable data collection on demand. The current granularity of observable data focuses on the service level and the instance level. By nature, fine-granularity monitoring data such as in request level is more beneficial to infer the system states. Taking the overhead and observability into consideration, controlling the granularity of observable data on demand is a promising direction in clouds.

C. Guidelines on Intelligent Operations in Clouds

To ensure the reliability of complex cloud systems, software for intelligent operations are widely used in anomaly detection [60]–[62], anomaly diagnosis [63], [64], resource management [7], [65], and automatic repair [66]. Based on the above Findings, we offer some guidelines on intelligent operations after going through the life cycle of faults:

- Based on Finding 3, when encountering faults during changes, engineers are guided to identify root causes by distinguishing differences in configuration, execution flow, and data flow between offline and online. The difference between them is possibly the root cause.
- Based on Finding 4, engineers are encouraged to focus on internal causes during upgrades and maintenance. Similarly, they are encouraged to focus on monitoring the changes in external environments during normal operations. The changes in external environments may provide some clues to fault identification.
- Based on Finding 8, it is suggested to consider the context of two or three components around the alerting component to identify the root cause.
- Based on Finding 10, some root causes show a strong relation with mitigation action items. For such kinds of root causes, engineers are recommended to design automatic mitigation action recommenders, which can largely reduce TTM.

More often than not, defects in the existing intelligent operation tools deserve certain attention. There are two possible promotion guided by our study: (i) It is suggested to build an uniform intelligent operations pipeline with low overhead. From our study, issues and inconsistent pipeline [azure-20180319-1, azure-20181119-1] in the monitoring system may delay the understanding, detecting and identifying of faults. (ii) It is suggested to develop robust cloud systems to false alerts. Monitoring system is imperfect and cannot achieve 100% accuracy. Following the principle of catching anomalies as much as possible, monitoring systems are designed to attain a high recall than precision. However, a false alert may trigger the system into self-protected mode and disrupt normal executions, leading to no alarms raising or unexpected faults. A robust system is expected to behave normally though receiving incorrect instructions.

VI. Threats to Validity

The **external** threat to validity lies in our collected post-mortems. We systematically collect 354 public and valuable post-mortems from large-scale clouds. But most of them provide desensitization information. So the analyzed results in our study largely depend on the willingness of cloud vendors to disclose fault information.

The **internal** threat to validity is about the manual labeling process. Due to the complex nature of cloud faults, subjectivity may exist in the procedure of structuring post-mortems by each engineer. To alleviate the threat, our study went through multiple rounds involving independent structuring procedures,

intersected structuring procedures, and further discussion on difficult faults.

VII. Conclusion

As the scale and complexity of cloud systems increasingly grow, a comprehensive study of faults in large-scale clouds is needed. We collect and perform quantitative and qualitative analysis of 354 public post-mortems from three popular clouds. When going through the life cycle of faults, we provide a series of interesting findings and reach some guidelines for fault handling. We believe our results in this study can inspire engineers and researchers both in industry and academia.

Acknowledgements

The research is supported by the Key-Area Research and Development Program of Guangdong Province (No. 2020B010165002), the National Natural Science Foundation of China (No. U1811462), the Natural Science Foundation of Guangdong Province (No. 2019A1515012229), and the Basic and Applied Basic Research of Guangzhou (No. 202002030328). The corresponding author is Pengfei Chen.

References

[1] M. Shahrad, R. Fonseca, Í. Goiri, G. Chaudhry, P. Batum, J. Cooke, E. Laureano, C. Tresness, M. Russinovich, and R. Bianchini, "Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, 2020, pp. 205–218.

[2] A. Balalaie, A. Heydarnoori, and P. Jamshidi, "Microservices architecture enables devops: Migration to a cloud-native architecture," *Ieee Software*, vol. 33, no. 3, pp. 42–52, 2016.

[3] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, 2012, pp. 15–28.

[4] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild *et al.*, "Spanner: Google's globally distributed database," *ACM Transactions on Computer Systems (TOCS)*, vol. 31, no. 3, pp. 1–22, 2013.

[5] P. Patel, D. Bansal, L. Yuan, A. Murthy, A. Greenberg, D. A. Maltz, R. Kern, H. Kumar, M. Zikos, H. Wu *et al.*, "Ananta: Cloud scale load balancing," *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4, pp. 207–218, 2013.

[6] D. E. Eisenbud, C. Yi, C. Contavalli, C. Smith, R. Kononov, E. Mann-Hielscher, A. Cilingiroglu, B. Cheyney, W. Shang, and J. D. Hosein, "Maglev: A fast and reliable software network load balancer," in *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, 2016, pp. 523–535.

[7] Y. Zhang, W. Hua, Z. Zhou, G. E. Suh, and C. Delimitrou, "Sinan: Ml-based and qos-aware resource management for cloud microservices," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pp. 167–181.

[8] C. Delimitrou and C. Kozyrakis, "Quasar: Resource-efficient and qos-aware cluster management," *ACM SIGPLAN Notices*, vol. 49, no. 4, pp. 127–144, 2014.

[9] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at google with borg," in *Proceedings of the Tenth European Conference on Computer Systems*, 2015, pp. 1–17.

[10] "Google cloud infrastructure components incident 20005," https://status.cloud.google.com/incident/zall/20005, 2022, [Online].

[11] "Azure incidents history," https://status.azure.com/en-us/status/history/, 2022, [Online].

[12] M. Washburn Jr, P. Sathiyanarayanan, M. Nagappan, T. Zimmermann, and C. Bird, "What went right and what went wrong: an analysis of 155 postmortems from game development," in *Proceedings of the 38th International Conference on Software Engineering Companion*, 2016, pp. 280–289.

[13] H. S. Gunawi, M. Hao, R. O. Suminto, A. Laksono, A. D. Satria, J. Adityatama, and K. J. Eliazar, "Why does the cloud stop computing? lessons from hundreds of service outages," in *Proceedings of the Seventh ACM Symposium on Cloud Computing*, 2016, pp. 1–16.

[14] H. Liu, S. Lu, M. Musuvathi, and S. Nath, "What bugs cause production cloud incidents?" in *Proceedings of the Workshop on Hot Topics in Operating Systems*, 2019, pp. 155–162.

[15] J. Sillito and E. Kutomi, "Failures and fixes: A study of software system incident response," in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2020, pp. 185–195.

[16] Z. Chen, Y. Kang, L. Li, X. Zhang, H. Zhang, H. Xu, Y. Zhou, L. Yang, J. Sun, Z. Xu *et al.*, "Towards intelligent incident management: why we need it and how we make it," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 1487–1497.

[17] B. Schroeder and G. A. Gibson, "A large-scale study of failures in high-performance computing systems," *IEEE Transactions on Dependable and Secure Computing*, vol. 7, no. 4, pp. 337–351, 2010.

[18] N. El-Sayed and B. Schroeder, "Reading between the lines of failure logs: Understanding how hpc systems fail," in *2013 43rd IEEE/IFIP international conference on dependable systems and networks (DSN)*. IEEE, 2013, pp. 1–12.

[19] Y. Liang, Y. Zhang, A. Sivasubramaniam, M. Jette, and R. Sahoo, "Bluegene/l failure analysis and prediction models," in *2006 Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2006, pp. 425–434.

[20] R. Banerjee, A. Razaghpanah, L. Chiang, A. Mishra, V. Sekar, Y. Choi, and P. Gill, "Internet outages, the eyewitness accounts: Analysis of the outages mailing list," in *International Conference on Passive and Active Network Measurement*. Springer, 2015, pp. 206–219.

[21] R. Zhang, W. Xiao, H. Zhang, Y. Liu, H. Lin, and M. Yang, "An empirical study on program failures of deep learning jobs," in *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 2020, pp. 1159–1170.

[22] J. Chen, Y. Liang, Q. Shen, and J. Jiang, "Toward understanding deep learning framework bugs," *arXiv preprint arXiv:2203.04026*, 2022.

[23] H. S. Gunawi, M. Hao, T. Leesatapornwongsa, T. Patana-anake, T. Do, J. Adityatama, K. J. Eliazar, A. Laksono, J. F. Lukman, V. Martin *et al.*, "What bugs live in the cloud? a study of 3000+ issues in cloud systems," in *Proceedings of the ACM Symposium on Cloud Computing*, 2014, pp. 1–14.

[24] S. Lu, S. Park, E. Seo, and Y. Zhou, "Learning from mistakes: a comprehensive study on real world concurrency bug characteristics," in *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, 2008, pp. 329–339.

[25] P. Fonseca, C. Li, V. Singhal, and R. Rodrigues, "A study of the internal and external effects of concurrency bugs," in *2010 IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2010, pp. 221–230.

[26] T. Xu and Y. Zhou, "Systems approaches to tackling configuration errors: A survey," *ACM Computing Surveys (CSUR)*, vol. 47, no. 4, pp. 1–41, 2015.

[27] Y. Zhang, J. Yang, Z. Jin, U. Sethi, K. Rodrigues, S. Lu, and D. Yuan, "Understanding and detecting software upgrade failures in distributed systems," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, 2021, pp. 116–131.

[28] T. Leesatapornwongsa, J. F. Lukman, S. Lu, and H. S. Gunawi, "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*, 2016, pp. 517–530.

[29] T. Xu, J. Zhang, P. Huang, J. Zheng, T. Sheng, D. Yuan, Y. Zhou, and S. Pasupathy, "Do not blame users for misconfigurations," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, 2013, pp. 244–259.

[30] S. Zaman, B. Adams, and A. E. Hassan, "A qualitative study on performance bugs," in *2012 9th IEEE working conference on mining software repositories (MSR)*. IEEE, 2012, pp. 199–208.

[31] E. B. Nightingale, J. R. Douceur, and V. Orgovan, "Cycles, cells and platters: an empirical analysis of hardware failures on a million consumer pcs," in *Proceedings of the sixth conference on Computer systems*, 2011, pp. 343–356.

[32] B. Schroeder and G. A. Gibson, "Understanding disk failure rates: What does an mttf of 1,000,000 hours mean to you?" *ACM Transactions on Storage (TOS)*, vol. 3, no. 3, pp. 8–es, 2007.

[33] G. Wang, L. Zhang, and W. Xu, "What can we learn from four years of data center hardware failures?" in *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2017, pp. 25–36.

[34] B. W. Chamberlin, "How failures cascade in software systems," 2022.

[35] J.-G. Lou, Q. Lin, R. Ding, Q. Fu, D. Zhang, and T. Xie, "Software analytics for incident management of online services: An experience report," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2013, pp. 475–485.

[36] ——, "Experience report on applying software analytics in incident management of online service," *Automated Software Engineering*, vol. 24, no. 4, pp. 905–941, 2017.

[37] J. Chen, X. He, Q. Lin, Y. Xu, H. Zhang, D. Hao, F. Gao, Z. Xu, Y. Dang, and D. Zhang, "An empirical investigation of incident triage for online service systems," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 2019, pp. 111–120.

[38] J. Chen, X. He, Q. Lin, H. Zhang, D. Hao, F. Gao, Z. Xu, Y. Dang, and D. Zhang, "Continuous incident triage for large-scale online service systems," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 364–375.

[39] J. Gao, N. Yaseen, R. MacDavid, F. V. Frujeri, V. Liu, R. Bianchini, R. Aditya, X. Wang, H. Lee, D. Maltz *et al.*, "Scouts: Improving the diagnosis process through domain-customized incident routing," in *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, 2020, pp. 253–269.

[40] J. Jiang, W. Lu, J. Chen, Q. Lin, P. Zhao, Y. Kang, H. Zhang, Y. Xiong, F. Gao, Z. Xu *et al.*, "How to mitigate the incident? an effective troubleshooting guide recommendation technique for online service systems," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 1410–1420.

[41] "Gartner's study of products in cloud infrastructure and platform services market," https://www.gartner.com/reviews/market/public-cloud-iaas, 2022, [Online].

[42] "Amazon web services reports," https://aws.amazon.com/cn/premiumsupport/technology/pes/, 2022, [Online].

[43] "Google cloud status," https://status.cloud.google.com/summary, 2022, [Online].

[44] C. B. Seaman, "Qualitative methods in empirical studies of software engineering," *IEEE Transactions on software engineering*, vol. 25, no. 4, pp. 557–572, 1999.

[45] "Microservice governance technology white paper (chinese)," https://developer.aliyun.com/ebook/7565, 2022, [Online].

[46] "Google site reliability engineering books," https://sre.google/books/, 2022, [Online].

[47] D. Oppenheimer, A. Ganapathi, and D. A. Patterson, "Why do internet services fail, and what can be done about it?" in *4th Usenix Symposium on Internet Technologies and Systems (USITS 03)*, 2003.

[48] "Oral presentation of postmortem action items: Plan the work and work the plan," https://www.usenix.org/conference/srecon17americas/program/presentation/lueder, 2017, [Online].

[49] W.-K. Wong, A. Moore, G. Cooper, and M. Wagner, "Rule-based anomaly pattern detection for detecting disease outbreaks," in *AAAI/IAAI*, 2002, pp. 217–223.

[50] X. Zhang, Y. Xu, Q. Lin, B. Qiao, H. Zhang, Y. Dang, C. Xie, X. Yang, Q. Cheng, Z. Li *et al.*, "Robust log-based anomaly detection on unstable log data," in *ESEC/FSE'19: Proc. of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 2019, pp. 807–817.

[51] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan, "Detecting large-scale system problems by mining console logs," in *SOSP'09: Proc. of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*. ACM, 2009, pp. 117–132.

[52] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A survey on software fault localization," *IEEE Transactions on Software Engineering*, vol. 42, no. 8, pp. 707–740, 2016.

[53] V. Heorhiadi, S. Rajagopalan, H. Jamjoom, M. K. Reiter, and V. Sekar, "Gremlin: Systematic resilience testing of microservices," in *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2016, pp. 57–66.

[54] "Chaosblade," https://github.com/chaosblade-io/chaosblade, 2022, [Online].

[55] "Chaos monkey," https://netflix.github.io/chaosmonkey/, 2022, [Online].

[56] G. M. Charity Majors, Liz Fong-Jones, *Observability Engineering*. USA: O'Reilly Media, Inc., 2022.

[57] X. Zhou, X. Peng, T. Xie, J. Sun, C. Ji, W. Li, and D. Ding, "Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study," *IEEE Transactions on Software Engineering*, 2018.

[58] "Dynatrace," https://www.dynatrace.com/, 2022, [Online].

[59] "Zipkin," https://zipkin.io/, 2022, [Online].

[60] M. Du, F. Li, G. Zheng, and V. Srikumar, "Deeplog: Anomaly detection and diagnosis from system logs through deep learning," in *SIGSAC'17: Proc. of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2017, pp. 1285–1298.

[61] H. Ren, B. Xu, Y. Wang, C. Yi, C. Huang, X. Kou, T. Xing, M. Yang, J. Tong, and Q. Zhang, "Time-series anomaly detection service at microsoft," in *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*, 2019, pp. 3009–3017.

[62] Z. He, P. Chen, X. Li, Y. Wang, G. Yu, C. Chen, X. Li, and Z. Zheng, "A spatiotemporal deep learning approach for unsupervised anomaly detection in cloud systems," *IEEE Transactions on Neural Networks and Learning Systems*, 2020.

[63] P. Chen, Y. Qi, P. Zheng, and D. Hou, "Causeinfer: Automatic and distributed performance diagnosis with hierarchical causality graph in large distributed systems," in *IEEE INFOCOM 2014-IEEE Conference on Computer Communications*. IEEE, 2014, pp. 1887–1895.

[64] P. Wang, J. Xu, M. Ma, W. Lin, D. Pan, Y. Wang, and P. Chen, "Cloudranger: Root cause identification for cloud native systems," in *2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. IEEE, 2018, pp. 492–502.

[65] S. S. Manvi and G. K. Shyam, "Resource management for infrastructure as a service (iaas) in cloud computing: A survey," *Journal of network and computer applications*, vol. 41, pp. 424–440, 2014.

[66] X. Zhang, J. Zhai, S. Ma, and C. Shen, "Autotrainer: An automatic dnn training problem detection and repair system," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 359–371.