How to Fight Production Incidents? An Empirical Study on a Large-scale Cloud Service

Supriyo Ghosh, Manish Shetty, Chetan Bansal, Suman Nath
{supriyoghosh,t-mamola,chetanb,sumann}@microsoft.com
Microsoft

ABSTRACT

Production incidents in today's large-scale cloud services can be extremely expensive in terms of customer impacts and engineering resources required to mitigate them. Despite continuous reliability efforts, cloud services still experience severe incidents due to various root-causes. Worse, many of these incidents last for a long period as existing techniques and practices fail to quickly detect and mitigate them. To better understand the problems, we carefully study hundreds of recent high severity incidents and their postmortems in Microsoft-Teams, a large-scale distributed cloud based service used by hundreds of millions of users. We answer: (a) why the incidents occurred and how they were resolved, (b) what the gaps were in current processes which caused delayed response, and (c) what automation could help make the services resilient. Finally, we uncover interesting insights by a novel multi-dimensional analysis that correlates different troubleshooting stages (detection, root-causing and mitigation), and provide guidance on how to tackle complex incidents through automation or testing at different granularity.

CCS CONCEPTS

• **General and reference** → **Empirical studies**; **Reliability**.

KEYWORDS

Incident Management, Empirical Study, Reliability, Distributed Systems.

SoCC '22, November 7–11, 2022, San Francisco, CA, USA

ACM ISBN 978-1-4503-9414-7/22/11.
https://doi.org/10.1145/3542929.3563482

ACM Reference Format:
Supriyo Ghosh, Manish Shetty, Chetan Bansal, Suman Nath. 2022. How to Fight Production Incidents? An Empirical Study on a Large-scale Cloud Service. In *SoCC '22: ACM Symposium on Cloud Computing (SoCC '22), November 7–11, 2022, San Francisco, CA, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3542929.3563482

1 INTRODUCTION

Production failures, or *incidents*, which adversely affect the customers, are inevitable in large-scale cloud services. They can be extremely expensive in terms of customer impact and engineering resources required to resolve them. Service providers, therefore, continuously try to minimize the frequency and impact of such incidents. A first step in doing so is an in-depth understanding of the life-cycle of incidents happening in production. Understanding the common root-causes can help develop techniques that can identify and fix them before production, thereby preventing potential incidents. Understanding how much engineering effort each resolution stage consumes can help identify and optimize bottlenecks in the process, thereby reducing the overall impact duration. Empirical studies have been an important tool to gain such deep understanding.

In recent years, several empirical studies were reported for cloud systems. A recent work [21] analyzed production incidents from Microsoft Azure to identify common root-causes and resolution strategies. However, the paper does not provide the complete picture since it was *limited only to incidents caused by software bugs*. As we show, there are many types of root-causes other than software bugs. Another paper [13] analyzes cloud outages *as observed from outside* (e.g., through news articles) and identifies root-causes and impacts. However, fidelity of the data source is limited; e.g., it includes only publicly-announced incidents, it does not describe how incidents were detected and mitigated, etc. Several other empirical studies [12, 17, 22, 34, 35] analyze bugs from open source applications that were found during in-house reviewing/testing and production uses. However, they are usually limited to specific types of software bugs and to systems much smaller than the cloud services we consider. Moreover, due to the nature of their data sources, these studies cannot provide deep understanding of production incidents.

In this paper, we systematically study 152 high severity production incidents that happened during a recent 12-months period in a large-scale distributed cloud service, Microsoft-Teams[1] which is used by more than 250 million users worldwide. It is a complex web-scale distributed service built on top of the Azure IaaS infrastructure and powering messaging, calling and meeting services for the real time communication productivity application from Microsoft. It is deployed across the world in tens of cloud regions and has a fairly complex set of dependencies comprising of storage, networking, authentication, etc. Since Microsoft-Teams powers real-time communication, reliability is paramount. This is the first study which extensively analyzes and provides end-to-end characterization of the reliability issues in a web-scale customer facing service. Also, unlike previous studies, we look at incidents caused by many types of root causes including software bugs, and analyze how the incidents are detected and mitigated. More importantly, by looking at the entire life-cycle of the incidents, we identify important correlation between detection, root-causing, and mitigation of the incidents. In comparison to prior works that focused on one axis only, such correlation uncovers important insights that provide new opportunities to improve reliability of large-scale cloud services.

We first independently look at the three major stages of an incident: detection, root-causing, and mitigation. Quick detection of an incident is crucial to limit its impact and usually such quick detection is achieved with automated monitors. In our study, roughly half of the incidents were successfully detected by existing monitors. In many cases, monitors existed but failed to detect the incidents. We find that simple measures such as collecting additional telemetry, fixing bugs in existing monitors, adjusting monitors' thresholds, or adding monitors on existing telemetry can improve effectiveness of automated monitoring by 50%, which motivates future research on intelligently and dynamically reconfiguring existing monitoring thresholds.

We then investigate the common root-causes behind the incidents. While software bugs are a common root-cause, a big portion (roughly half) of the incidents are caused due to non-code related issues such as infrastructure capacity issues, manual deployment errors, and expired certificates. This shows the importance of our investigation of non-code related root causes (unlike [21]). Based on our analysis, we propose a taxonomy of common root-causes.

We then look at how incidents are mitigated. Interestingly, a vast majority (> 90%) of the incidents are mitigated without code-change[2]. Common mitigation strategies include rollback (i.e., reverting service to an earlier version), infrastructure and configuration change, rebooting micro-services, etc.

In order to limit the impact of an incident, it is important that it is detected and mitigated quickly. However, some incidents take much longer than others. We investigate the root-causes behind such delays. Detection is delayed when automated monitors fail to detect them, e.g., due to bugs in the monitor, missing telemetry, and incorrect thresholds and granularity. Mitigation is delayed mainly due to slow deployment of a fix (e.g., due to lack of on-call engineer's (OCE's) permission to deploy), poor documentation/understanding of applying mitigation, and having humans in the loop. These points to several techniques that can potentially expedite detection and mitigation such as fine-tuning monitors and adding additional testing, improving documentation and deployment/coding practice, automated mitigation such as automatic certificate renewal, traffic-failover, auto scale, etc.

The above analyses of individual stages are useful; however they miss important insights based on correlation among different stages. For example, the above analyses do not answer what types of root-causes are behind the incidents that are the hardest to automatically detect and quickly mitigate. An important contribution of this work is a multi-dimensional analysis that correlates various resolution stages at different granularities. Our analysis makes several important observations, including (1) incidents caused by software bugs and external dependencies take longer to detect due to poor monitoring. This highlights the need of practical tools for fine-grained, in-situ system observability[15]. (2) Incidents caused by some root-cause categories are quick to mitigate *after their root-cause categories are determined*. For example, incidents caused by configuration bugs and certificate expiration can be quickly mitigated by rollback and configuration update, respectively. This suggests that the overall mitigation time of incidents caused by these categories can potentially be reduced with tools that, given an incident, can quickly identify its coarse-grained root-cause category. (3) Incidents caused by some root-causes are inherently hard to monitor automatically (e.g., that requires monitoring global states). This suggests that developers should invest more in testing to uncover those root-cause categories before production, thereby avoiding such incidents. In summary, we make the following contributions in this paper.

(1) We analyze 152 high-severity production incidents from a large-scale cloud service serving hundreds of millions of customers. Our study differs from prior works in several important ways:
   - We consider incidents caused not only by software bugs, but also by various other non-code-related root causes.

[1] https://www.microsoft.com/en-us/microsoft-teams

[2] Note that *mitigation* is often a temporary measure to ensure that the service continues to operate. Many incidents are later *fixed* with more permanent measures such as code fix.

- We analyze not only the root-causes of the incidents, but also how they are detected and mitigated and why these tasks sometimes take undesirably long time.
- We analyze reflections and lessons learnt by OCEs to identify potential opportunities to improve reliability of cloud services.

(2) We present a novel multi-dimensional analysis that correlates different troubleshooting stages (detection, root-causing, and mitigation) at different granularity. Our analysis uncovers important insights that can be valuable in improving reliability of large-scale cloud services.

The rest of the paper is organized as follows: In Section 2, we discuss the methodology of the empirical study. In Section 3, we investigate the root causes and mitigation steps of the incidents. Section 4 discusses the delay in responding to incidents, while in Section 5, we go over the lessons learnt to make our services resilient to incidents. In Section 6, we analyze patterns via multi-dimensional analysis. We discuss the related work in Section 7 and then conclude the paper.

2 METHODOLOGY

2.1 Incident Selection in Our Study

Production incidents at Microsoft can be reported by users (internal or external) or by automated system watchdogs that continuously monitor system health telemetry data to identify anomalous behaviors. Every production incident is recorded in the incident database, along with information such as incident, root cause and mitigation title and descriptions, OCEs' discussion, severity-level tag, work items issued to developer teams (if any), the incident impact duration and a set of "why" questions related to the incident response from postmortems, e.g., why the detection, mitigation or engagement was delayed, why the existing service resiliency failed and what automation could be used to make the service resilient to similar future adverse scenarios, etc. We sample and study 152 high severity incidents (severity 0, 1 and 2) that happened in Microsoft-Teams during the 12 months period from May 15, 2021 to May 15, 2022 that satisfies all the following conditions:

- The incident severity level (2 or less) indicates that it led to failure of some component of the service and impacted several tenants and customers.
- The incident has been resolved or mitigated, and a root cause description is associated with the incident report.
- The incident has a complete postmortem report with extensive information on detection and mitigation steps.

The severity of incidents ranges from 0 to 4. Severity 3 and 4 incidents are non-critical, low impact and non-paging, and the majority of such incidents are transient and auto mitigated. So, we considered only incidents with severity <= 2 (2% of total incidents). Among the 152 incidents, we observe that 30% of incidents in our study have a severity level of "0" or "1" (only one incident has a "0" severity level) and the rest 70% of incidents have a severity level of "2". It should be noted that not all incidents in our study impacted external customers or users. Many of the incidents were reported by internal users or automated watchdogs, and were resolved before impacting external customers.

2.2 Categorization Strategy

In this work, we study high severity incidents to understand not just what caused them, but also how they were mitigated, the challenges faced, and strategies for optimal incident management in the future. Typically, an incident goes through three main phases: (1) *detection* – incident is detected by a service monitor or reported by a customer, (2) *root cause analysis* – investigation by on-call engineers to identify the root cause, and (3) *mitigation* – following procedures and executing steps to mitigate the impact of the incident. To holistically capture this process, we first identify **six factors** to study (as shown in Table 1), that impact effective incident management. For each of these factors, we then manually populated a summarized description using the incident report and the corresponding postmortem report. Using these summaries, we develop a taxonomy of categories for each of the six factors using an open coding approach [30]. We then categorize each incident using these taxonomies and analyze their distributions. Below we describe the procedure followed in detail:

Table 1: Factors used to study incidents and their mitigation

Study Factor	Description
Root Cause	What issue caused the incident?
Mitigation Steps	What steps were performed to restore service health?
Detection Failure	Why did monitoring not detect the incident?
Mitigation Failure	What challenges delayed incident mitigation?
Automation Opportunities	What automation can help improve service resilience?
Lessons for Resiliency	What lessons were learnt about the service's behavior and improving resiliency?

Setup. We start by randomly sampling and splitting our 152 incident dataset into three subsets: (1) *taxonomy set*: 60 incidents, (2) *validation set*: 30 incidents, and (3) *test set*: 62 incidents. Then, the first two authors of this work used an open coding [30] approach to label the *taxonomy set* independently. First, they assigned a category to each factor of an incident that best described the summary. E.g., if the root cause summary was "an incorrectly unset configuration to fetch all logs", one can categorize it as a configuration bug. Subsequently, they examined the categories and settled on a common taxonomy for each factor. Next, they labeled the *validation set* to ensure no new categories arose and had another discussion to define a more refined understanding of each category. Lastly, they annotated the *test set* and used Cohen's kappa [9] to find the inter-annotator agreement score. We observe near-perfect agreements for each of the six taxonomies: *Root Cause*: 0.94, *Mitigation*: 0.945, *Detection Failure*: 0.88, *Mitigation Failure*: 0.94, *Automation Opportunities*: 0.936, *Lessons for Resiliency*: 0.98. It should be noted that we do not double count an incident into multiple categories. For instance, if an incident is caused by multiple root causes, then the disagreement is resolved by picking (1) the most specific category that (2) first occured in the summary. Using this approach, the annotators created a complete categorized dataset of 152 incidents, which was used for all analyses described in the following sections.

2.3 Threats to Validity

The results from our study should be interpreted with our methodology and the properties of Microsoft-Teams in mind. The insights generated from root cause, mitigation strategy and automation opportunities may not replicate in other cloud systems, as Microsoft uses a wide variety of effective tools and techniques to proactively eliminate many types of bugs, and simultaneously many automation tools are in operation to automatically mitigate several types of incidents before impacting customers. Our insights may not be generalized across all the incidents in Microsoft that we have not considered in our study and may not represent the behaviour of other public cloud services. Even within Microsoft-Teams, we have filtered out about 35% of recent high severity incidents that did not have a complete postmortem report with detailed root cause and mitigation description, which may lead us to miss unprecedented incidents with limited information.

3 WHAT CAUSES INCIDENTS AND HOW WERE THEY MITIGATED?

In this section, we investigate what are the common categories of root causes behind high severity production incidents, and how those incidents are resolved.

3.1 What are the Root Causes?

Each incident report contains a root cause title associated with a detailed description. From these descriptions and OCEs' discussion, we identify the root cause and categorize them into 7 types (see Figure 1).

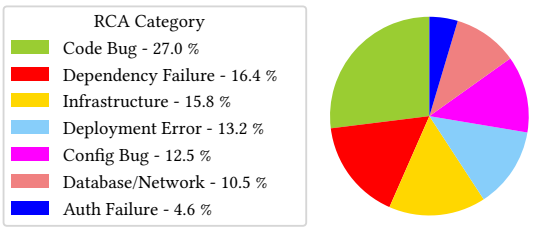

Figure 1: Breakdown of root cause categories

Finding#1: While 40% incidents were root caused to code or configuration bugs, a majority (60%) were caused due to non-code related issues in infrastructure, deployment, and service dependencies.

Code bugs. Code related bugs are the majority contributor to cloud service incidents. We observe the following code bugs in our study: (1) Code change or buggy features (24.4% of total code bugs): These issues arise when developers update existing codebases or deploy a new feature that passes existing test cases but contains faulty code. For example, a recent code change might disable a feature for some use cases because a predefined check erroneously failed, but other scenarios require the feature. Another instance is a code change introducing a buggy feature that did not support specific types of infrastructure or users. (2) Flags and constants (24.4% of code bugs): These types of code bugs arise when a feature flag is set wrongly (but passed the testing due to inadequate scenario testing) that disable some of the features to function properly. Another prevalent error in this category arises when a threshold parameter value for service health check or performance metric is set incorrectly. (3) Code dependency (19.5% of code bugs): Due to inter-dependencies between different components in large cloud services, a code change in one component can break the functionality of dependent components or features. A code dependency error usually occurs when a change in one component fails to handle additional attributes in the legacy code of a dependent component. (4) Datatype, validation, and exception handling (17.1% of code bugs): These are typical errors developers introduce in the code either by misunderstanding the datatype of a column in a database or by using bad error handling logic. Validation errors occur when a code change fails to validate tokens and certificates. (5) Backward code compatibility (14.6% of code bugs): Backward compatibility errors emerge when a code update breaks

the existing feature. E.g., in an API code change where a blob entry had been renamed but one place was missed which prevented flow of information reaching to the user interface side.

Infrastructure issues. Large cloud services are built upon a complex hierarchical infrastructure comprising nodes, clusters, and data centers that host tightly coupled resources, including CPU, memory, and networks. Due to the continuous operation of cloud systems, the performance of these resources often deteriorates. We divide these infrastructure degradation issues into the following categories: (1) CPU capacity (33.3%): The CPU utilization drastically increased because an infrastructure change reduced the CPU capacity or a specific app or request is using a lot of CPU, which leads to failure of a particular node/cluster and subsequently service degradation. (2) Capacity issue due to high traffic (41.7%): When a service (internal or external) receives unusually high amount of requests or traffic, the existing infrastructure usually cannot handle the load, so it starts throttling and a high latency or delay is observed. (3) Infrastructure scaling (16.7%): In some cases, only a slice of production clusters is used for operation, which causes over-utilization of the resources, and therefore applications dependent on these resources fail. (4) Infrastructure maintenance (8.3%): During infrastructure maintenance or migration, the cache information is unintentionally deleted, due to which endpoint users cannot connect to the service.

Deployment errors. Operator mistakes during deployment are a common phenomenon in cloud services [25, 33]. In our study, we observe three types of deployment errors. (1) Certificate management (55.0%): During deployment, operators either used old certificates that have already expired or used incorrect certificates, due to which the service cannot fetch correct authentication tokens. Another common issue is certificate auto-rotation, which requires changing some settings of the certificate issue policy, which is ignored during deployment. (2) Faulty deployment & patching (25.0%): Operators deployed a wrong patch that breaks some existing features or creates an incorrect dependency on another component. E.g., someone wrongly deployed a patch that made the UI point to a newer version of a dependent service that didn't exist yet. (3) Human Error (20.0%): This problem emerges when an operator makes mistakes with manual steps during the deployment.

Configuration bugs. To manage multiple tightly coupled components in a cloud service, operators employ many configuration settings that need to sync correctly to keep the service running smoothly. Here, we observe that configuration mismanagement is a common phenomenon. These configuration management errors can be categorized into three types: (1) Misconfiguration issue (47.4%): Operators either make mistakes with the configuration setting or use a bad configuration setting that does not follow standard requirements, which leads to degradation of service quality (traffic or latency increased) in specific regions or failure of dependent services. (2) Configuration change (42.1%): When a change is deployed to a configuration setting to adopt new scenarios or use cases without carefully analyzing its dependency, it leads to the failure of other components using the same configuration parameters. (3) Configuration sync (10.5%): If two or more flighting configuration settings operate within the service, then the updated configuration could expire, and the old configuration is used that fails to fetch the right tokens.

Dependency failures. In a large cloud system, multiple services run simultaneously and share dependency among each other. However, the OCEs for different services usually operate in silos. So, if the root cause comes from a dependent service, then the OCEs escalate those issues to the partner team. E.g., the small number of hardware issues that affected Microsoft-Teams were classified as "dependency failure" since the issues are handled by Azure IaaS engineering team that Microsoft-Teams depends on. We refer to those problems as dependency failure and categorized them into four types: (1) Version incompatibility (24.0%): When the partner team deploys a new version or build, it can create a backward code compatibility issue with other dependent codes. The new source code version may also not be compatible with other external features. (2) Service health (20.0%): If the service quality of a dependent service degrades, the down-streaming task also fails. (3) External code change (28.0%): If the partner team or remote service introduces an erroneous change, deploys a faulty package, or updates some configuration setting that has a dependency on the configuration of Microsoft-Teams, then the functionality of Microsoft-Teams is impacted. (4) Feature dependency (28.0%): The partner team rolls out a new feature or replaces some content of an existing feature that is not recognized or synced with the configuration used by the service.

Database or network problems. Although it overlaps with Infrastructure issues, we created a separate category as we identified significant number of database or network related problems. Database or network problems are mostly capacity related when the cloud system cannot handle higher than normal or expected user request and throttle user's request. Among these incidents, 25% are related to network latency due to high round trip time (RTT) or latency spike in a dependent service. Around 31% of incidents occur due to network availability or connectivity issues due to which thread services were taking longer to process request and generating timeout errors. Other 44% incidents in this category occurred due to two types of database related issues: (1) 25% incidents happened due to outages of database that impacted the file operations; and (2) 19% incidents happened

due to insufficient scaling of database capacity after user request load increased.

Authentication failures. In a large-scale cloud service, authentication failures are unavoidable. The authentication errors emerge due to access policy change or race condition between additional tokens rolled out during a new build deployment. We can broadly categorize them into three types: (1) Authorization issues (42.9%): users do not have permissions to access the service, either the new deployment requires more permission than the tested beta version, or users presented multiple tokens and the strongest token is invalid; (2) Certificate rotation (28.6%): due to rotation of new keys, tokens or certificates, making calls to the service failed; and (3) Authentication errors (28.6%): due to access policy change, the authentication request to the service failed.

3.2 What are the Mitigation Steps?

In this section, we describe the common categories of mitigation strategies employed to tackle the high severity incidents. Each incident and its corresponding postmortem report have a title and a detailed description of the mitigation steps. From these descriptions, OCEs' discussion, and work items, we identified the exact mitigation method, and categorized them to 7 types (see Figure 2).

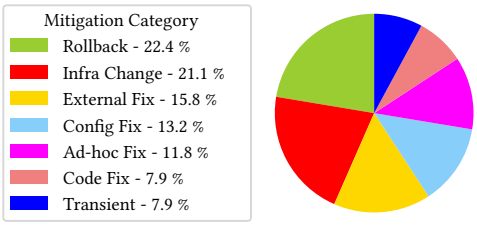

Figure 2: Breakdown of categories for mitigation steps

Finding#2: Although 40% incidents were caused by code/configuration bugs, nearly 80% of incidents were mitigated without a code or configuration fix.

Rollback. Facing time pressure, a common strategy to mitigate the incidents is to revert the changes to an older and stable version. This method is usually popular as it helps to recover the service quickly. We observe three types of rollback strategy in our study: (1) Rollback of code change (35%): carried out by the developer team or external partner team either by reverting the pull request or by reverting the code change (e.g., fixing the flag values according to the previous stable code); (2) Rollback of configuration change (24%): manually reverting the bad configuration change that was recently deployed; and (3) Rollback of new build (41%): either by redeploying an older stable build or pinning the users to the previous build that did not contain the issue.

Infrastructure change. Making an infrastructure change is a frequently used mitigation method as it can quickly recover the service, especially for capacity and throttling issues. Among these infrastructural changes, about 44% of incidents use traffic failover to another healthy service component. The traffic rerouting could be accomplished in 3 ways: (1) failover to another healthy node (16%); (2) failover to another healthy cluster (9%); and (3) fail-over to another cloud region (19%). Other infrastructural changes (56%) are accomplished via node scaling or node reboot operations. Among these, about 31% of the incidents are mitigated by upscaling the node infrastructure to tackle overutilization problems, and 10% use node downscaling strategy (e.g., deleting incorrectly provisioned nodes). The rest, 15% of infrastructural changes are performed by restarting the faulty or unhealthy nodes (sometimes after rerouting the traffic to another node).

Finding#3: Mitigation via roll back, infrastructure scaling, and traffic failover account for more than 40% of incidents, indicating their popularity for quick mitigation.

Configuration fix. To fix the majority of configuration errors and authentication failures, operators manually fix bugs in certificates or configuration files to restore the service. A quarter of the incidents in this category are mitigated by either creating a new certificate or rotating a renewed certificate that syncs with existing requirements. 20% of the incidents in this category are solved by changing and redeploying the erroneous configuration files, and the other 20% of incidents are mitigated by restoring the previous steady configuration files. 25% of the incidents in this category are solved by fixing the faulty features by performing one of the following actions: (1) disabling the new feature, (2) reverting the feature change, or (3) failover to other similar but stable feature. The remaining 10% of the configuration fixes are performed by syncing the dependent configuration files for different services.

Code fix. Updating and fixing a buggy code with additional scenario testing is a typical and frequent resolution methodology [21]. However, facing tight time constraints, a relatively low number of incidents are resolved with a code fix in our study. Among these incidents, about 42% are resolved with code change, i.e., by identifying the bug in the code and rolling out a fix. If the issue is related to bad setting of binary flags and constant values (magic number problem), then they are solved by changing the magic numbers (17% of total code fixes). 25% of the code fixes are implemented with additional exception handling logic, and for the rest 17% of incidents, an entire code module or abstract method is added to include new resources (e.g., certificates) that are rolled out recently.

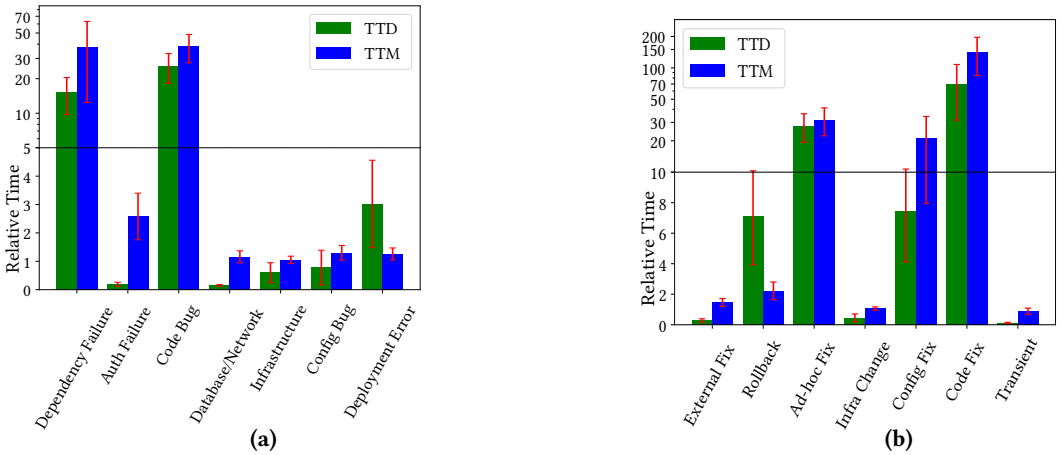

Figure 3: Average TTD and TTM for (a) different root cause categories; and (b) different categories of mitigation steps. (Y-axis shows the normalized time with the median of time to detect or mitigate of all incidents as 1; for each category, 10% outliers on the high and low side are removed; Y-axis numbers beyond the black horizontal line are shown in log scale).

External fix. After examining the root causes carefully, the internal OCEs might realize that the fix needs to be executed by the partner team – we refer to these resolution methods as external fixes. Among the external fixes, 29% of the cases, the partner team simply rolled back the recent changes, including code/configuration change and deployment of a new build. For 17% of these external fixes, the partner team identified the bug in code/configuration and manually fixed them. For the rest 54% of external fixes, the partner team executes a wide variety of mitigation steps ranging from fixing metadata to rebooting nodes/clusters to traffic rerouting, and sometimes sending notifications to customer/users asking for disabling unsupported plugins.

Finding#4: Even among incidents triaged to external teams, only a minority 17% of incidents were mitigated using a code/configuration fix.

Ad-hoc fix. When the root cause is complex and OCEs are not familiar with the issue, they execute a series of ad-hoc commands. Depending on the outcome of previous steps, further mitigation steps are taken. These types of fixes are called "Hotfix", which could be run by the internal OCEs or escalated to a partner team. E.g., a "Hotfix" was rolled out by the authentication service to use correct certificate or update certificate public keys.

Transient. These incidents are triggered by automated watchdogs if certain health check metrics crossed a predefined threshold. We refer to these incidents as false alarms. When infrastructural conditions stabilize (e.g., network connectivity restored), these incidents get automatically mitigated and the system recovers. Specifically, these incidents are auto mitigated for mainly for 3 reasons: (a) Auto healing of infrastructure (e.g., network recovered); (b) Updating or restarting the app from the user side resolved the problem; or (c) Service health metric automatically recovered.

4 WHAT CAUSES DELAY IN RESPONSE?

In this section, we analyze the response times in terms of time to detect (TTD) and time to mitigate (TTM) to understand the critical root causes and mitigation types. TTM is computed as the difference between the time when an incident is mitigated and the time when an incident is detected. Furthermore, we analyze the common reasons behind detection and mitigation delays by collecting information from the OCEs' comments.

4.1 Response Time Analysis

Figure 3(a) compares the normalized TTD and TTM among incidents caused by different types of root causes, with the median of union of TTD and TTM for all the incidents as "1". In addition, for each category of root causes, we removed 10% of outliers from high and low side. For each category, we show the average TTD and TTM along with standard error (denoted by red capped line). The detection and mitigation time for both code bugs and dependency failure is significantly higher than other root cause types. For authentication failures and database/network related issues, the detection process was quick but the mitigation time was relatively longer. On the other hand, for deployment errors, detection took longer than the mitigation time.

Finding#5: The time-to-detect code bugs and dependency failures is significantly higher than other root causes, indicating inherent difficulties in monitoring such incidents.

Figure 3(b) compares the normalized TTD and TTM among incidents caused by different types of resolution steps. Similar to the analysis with root cause, we removed 10% of outlier incidents for this experiment. As expected, the detection and mitigation times for the transient and infrastructural change category were quick. Similarly, the mitigation times for incidents with rollback as a fix were low. Interestingly, the detection and mitigation times for the external fix were significantly low as most of the mitigation steps were either rollback or infrastructural changes. As code or configuration fixes require manual effort, the detection and mitigation time were longer for these incidents. Similarly, as expected, the ad-hoc fixes took longer time to mitigate the incidents in comparison to rollback or infrastructural changes.

Finding#6: Manually fixing code and configuration take significantly higher time-to-mitigate, when compared to rolling back changes. This supports the popularity of the latter method for mitigation.

4.2 Reasons for Delay in Detection

In this section, we discuss how the incidents are detected and why existing watchdogs failed to detect all incidents automatically.

4.2.1 ***How incidents are detected?*** Unlike traditional software systems, we observe that about 55% of the incidents were detected by the automated watchdogs. These automated watchdogs are deployed by developers of Microsoft to monitor system health data (e.g., latency, CPU usage, memory usage) continuously and fire alarm once a metric goes beyond configured healthy threshold. In the next section we discuss the key reasons for which existing watchdogs failed to detect the rest of the incidents. Among other 45% incidents, 29% incidents are reported by the external users or customers, and 10% are identified by the partner teams within Microsoft. The remaining 6% incidents are detected by the Microsoft-Teams's service team itself.

4.2.2 ***Why automated watchdogs failed?*** We now study the reasons behind a detection failure from OCEs' perspective. Incident postmortem reports mostly contain an explanation for the detection failure if the incident was manually reported by an internal or external customer. From these descriptions, we identified the key reason behind a detection failure and categorized them into 7 types (see Figure 4).

Monitor bugs. In Microsoft, several automated watchdogs are deployed that continuously monitor various performance metrics. In most cases, the operators set static rules that if performance metrics crossed a predefined threshold, an alert is triggered with the details of anomalous health data. A quarter of monitor issues arise as the overall failure

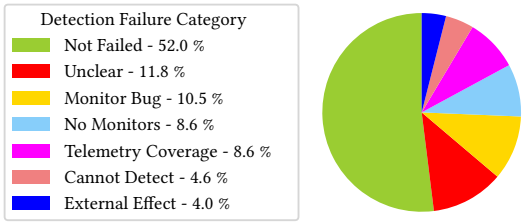

Figure 4: Breakdown of categories for detection failure

rate was below a high alert threshold. Another quarter of monitor bugs are related to misdiagnosis of the severity level of an incident by the watchdog. Rest 50% of the monitor bugs appeared because testing failed to identify that monitor had buggy features or bad configuration settings.

Telemetry coverage. Albeit having an automated watchdog for a particular component, the detection failure occurs as the watchdog does not record some specific telemetry data. We broadly categorized these telemetry coverage failures into three buckets: (1) environment related (31%): additional telemetry data is required for specific type of cloud environment; (2) service health related (38%): additional alerts for resource utilization metrics (e.g., high CPU usage or service down alerts) are needed; and (3) scenario related (31%): additional scenario-based alerts (e.g., HTTP scenarios, or telemetry data for a specific error code) are missing.

External effect or hard to detect. If the incident is caused due to dependency on the partner team, and the detection was delayed because of partner application failed to detect the anomaly from their side, we refer to them as "External Effect". E.g., the monitors all behaved as expected from the service side, but authentication service needs to add more monitors specifically for partner service latency. On the other hand, if the incident was detected by the customers or partner team through manual scenario testing (e.g., a deployment issue), we refer to them as "Cannot Detect" category, as these incidents are hard to recognize by the automated watchdogs.

No watchdogs. Albeit having a large number of watchdogs, we encounter incidents for which no automated watchdog was present to detect anomalies for a relevant performance metric. In some cases, the incident was not detected as the monitor was missing on the partner application side.

Not failed or Unclear. More than half of the incidents were successfully triggered by the existing automated watchdogs. If the OCEs left delay failure field empty in postmortem, we refer to them as "Unclear" category.

4.2.3 ***Detection failure time analysis.*** In Figure 5(a), we demonstrate the average time to detect an incident for different types of detection failures along with standard error, after removing 10% of outliers from high and low side. The detection time was highest for the incidents where failure is

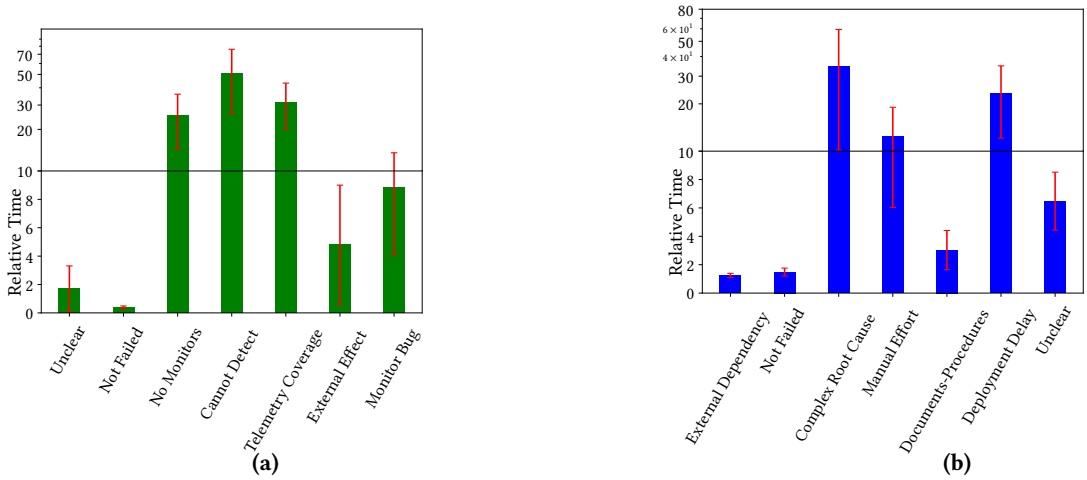

Figure 5: (a) Detection time for different types of detection failure; and (b) Mitigation time for different types of mitigation delay. (Y-axis shows the normalized time with the median of time to detect or mitigate of all incidents as 1; for each category, 10% outliers on the high and low side are removed; Y-axis numbers beyond the black horizontal line are shown in log scale).

complex and hard to recognize, albeit having a low proportion of such incidents. As expected, the detection times were significantly high in the absence of automated watchdogs, or where the monitors collect a limited amount of telemetry data.

Finding#7: ≈17% of incidents either lacked monitors or telemetry coverage, both of which result in significant detection delays.

4.3 Reasons for Delay in Mitigation

Incident postmortem reports contain an explanation for mitigation failure if the total impact time after detection is reasonably high. From these descriptions, we identified the key reason behind a mitigation failure and categorized them into 7 types (see Figure 6).

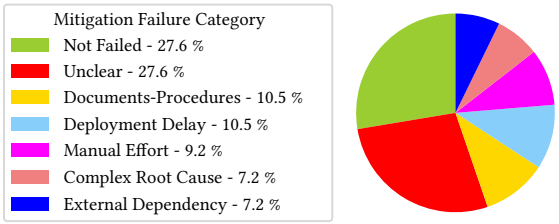

Figure 6: Distribution of mitigation failures

Deployment delay. Once a fix is deployed, it took longer time for the service to recover. The deployment delay occurs for three reasons: (1) external manual approval (25%): the OCEs needed additional approvals to execute a fix or change; (1) permission issues (25%): assigned OCEs did not have required permission to access the required log files or to disable the faulty features; and (3) slow change in the system (50%): it took longer time for the configuration change or node scaling or rolling out of a "hotfix" to take effect.

Documentation and procedures. Mitigation process was delayed because of poor documentations or infrastructural setup, which can be categorized into three broad types: (1) OCE knowledge gap (19%): OCEs have not acted swiftly when the severity level was low or the similar incident resolution method was not communicated among OCEs properly; (2) Troubleshooting guide (TSG) quality (50%): OCEs followed existing TSGs to resolve the incident, but could not find clear and specific recommendation like when to declare an outage, or which team to engage for code change; and (3) Poor infrastructure setup (31%): proper setup was not available for auto-scaling of nodes, auto-failover of traffic, or automatically rollback codes in a specific environment.

Complex root Cause. Debugging the problem and identifying the actual root cause was a painful and lengthy process. We observe three reasons for delay in root cause identification: (1) rare occurrence (18%): either the problem came from a new feature or that particular scenario is infrequent; (2) no telemetry information (27%): it was difficult to capture scope of the problem as required performance metrics were not logged; and (3) debugging problem (55%): it took time to figure out bugs in code change.

Finding#8: While complex root causes can affect time-to-mitigate, 30% of incidents had mitigation delays even after identifying the root cause due to poor documentation, procedures, and manual deployment steps.

Manual effort or external dependency. Mitigation process was delayed because it involves manual steps or errors in three ways: (1) OCE avalanche (14%): multiple OCEs simultaneously executed several steps without having a proper communication; (2) misdiagnosis (29%): the problem was not properly diagnosed initially, which leads to delay in executing wrong resolution methods; and (3) deployment and configuration errors (36%): it requires manual effort to scale the infrastructure or renew certificates. Another challenge faced by the internal OCEs when there was communication gap with the partner team and engagement was delayed, which we refer to as as "External dependency".

Not failed or Unclear. If the mitigation process was quick as per the OCEs' expectation, then we refer to them as "Not Failed" category. If the OCEs indicated that they do not understand the reasons behind the delay in the mitigation process, we refer to them as "Unclear" category.

4.3.1 ***Mitigation failure time analysis.*** In Figure 5(b), we compare the average time to mitigate an incident for different types of mitigation failures along with standard error, after removing 10% of outliers from high and low side. The incidents with complex root cause took longest time to mitigate, as the OCEs' do not have proper documentation or trouble shooting guides to resolve these incidents. As expected, the mitigation took longer if there are deployment related delays. Mitigation also gets delayed if any manual step is involved in the process.

5 LESSONS LEARNT FOR RESILIENCY

In this section, we discuss the potential automation opportunities to make the service resilient against future adversarial scenarios and the lessons learnt by the OCEs while tackling high severity incidents.

5.1 Automation Opportunities for Future

In each incident postmortem report, OCEs provide their expertise opinion on what automation could be added in the service to make them resilient to similar failures in the future. By reading these descriptions, we categorize these automation strategies into 6 types (see Figure 7).

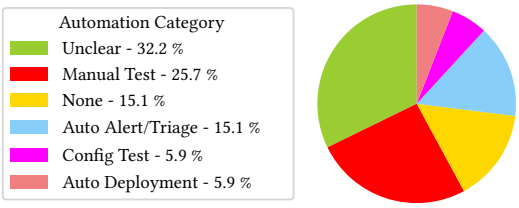

Figure 7: Potential automation opportunities

Manual and configuration test. Majority of the automation suggestions provided by the OCEs are related to improve a variety of testing methodologies: (1) performance test (12.8%): use chaos engineering techniques for load or stress testing; system test (23.1%): end-to-end testing to verify system functionality; scenario test (30.8%): test with corner and infrequent cases for specific type of users or deployment environment; validation test (12.8%): new tests to eliminate authentication ambiguity; integration test (7.7%): to identify partner dependency failure; and unit test (12.8%): new tests to verify few features or backward code compatibility. Another category of testing suggestion was related to configuration test which indicates that all boundary conditions should be tested before rolling out any configuration change.

Automated alert/triage. To eliminate detection failures, OCEs suggested three types of methods to automate the alerts or triaging process: (1) fine-tuning monitors (30%): properly tune the threshold parameters for the monitors so that it fire alerts sooner when the impact is low, or add new performance tools and anomaly detection rule, e.g., setting up an alert about certificate being close to expiry; (2) improve health checks (52%): need to add more monitors and increase telemetry coverage for existing monitors, e.g., add telemetry data for latency or add new alert system for service worker activation; and (3) automated triaging (18%): set up automatic escalation method between OCEs and partner team.

Finding#9: Improving testing was a popular choice for automation opportunities, over monitoring, indicating a need to reduce incidents by identifying issues before they reach production services.

Automated deployment. We observe two types of automation suggestion for the deployment: (1) Automation of failover (44%): build automation to automatically swap the traffic manager when this issue is detected; and (2) automated release (56%): fully automate the release pipeline to avoid any human intervention that will eliminate delay or error in manual steps.

Unclear or no automation required. In this scenario, OCEs mentioned that additional automation will not be helpful for service resiliency. We broadly divide these comments into three types: (1) Adequate automation exists (30%): either enough monitors and test cases are already in place, or additional automation will not be able to catch the specific incident; (2) Automation is hard (18%): if it is a production network or capacity issue, then setting up monitors or test cases is very challenging and; (3) Automation not applicable (52%): the problem lies in partner applications. If OCEs indicated that they are not sure about any potential automation opportunities that can tackle a particular incident, we refer to them as "Unclear" category.

5.2 Discussion on Lessons Learnt for Future

Similar to automation suggestions, in each incident postmortem report, OCEs provide their perspective on what is learnt for the future from the incident. By reading these descriptions, we categorize these future lessons learnt into 7 types (see Figure 8).

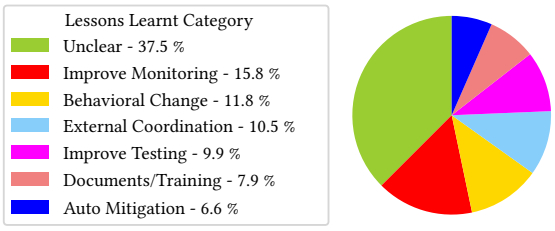

Figure 8: Breakdown of categories for lessons learnt

Improve monitoring and testing. In the postmortem, the OCEs provide suggestions to improve the existing monitoring system and testing process. For monitoring, we observe 3 types of suggestions: (1) Add telemetry (54%): update existing watchdogs to proactively track growth in performance and memory usage to detect anomalies for some specific scenarios, or create new watchdogs to fire alerts for misconfiguration and production release problems; (2) Fix alerts (33%): fix monitor bugs by revisiting the threshold parameters; and (3) Better dashboard (13%): enhance monitoring dashboard to track low volume scenarios. Similar to automated testing suggestions mentioned in Section 5.1, several types of testing lessons ranging from monitor testing (13%) to system testing (33%) to configuration testing (20%) to scenario testing (20%). We observe a new type called version testing (27%), where the suggestion was to have a strong resiliency with the legacy codebase by running proper test cases for compatibility check while upgrading to a new version.

Behavioral change. OCEs mentioned a variety of behavioural changes in incident management and engineering practices to make the service resilient. We broadly categorized them into 4 types: (1) Deployment practice (33%): operators should exercise caution while rolling out configuration change or certificates in sensitive environment; (2) Programming practice (34%): developers need to follow caution while turning on flags in some scenario, and they should carefully block all unsupported scenarios to avoid customer impact; (3) RCA and mitigation practice (22%): OCEs should carefully check that they have right permissions and knowledge before executing any rollback or restart operation in sensitive environments; and (4) Monitoring practice (11%): OCEs should pay attention in health dashboard and re-evaluate capacity measure before changing incident severity level.

Training and documentation. While solving an incident, OCEs often employ existing documentations that were used to solve similar incidents. We find suggestions for improving these documentations in four ways: (1) better TSG (50%)): existing TSGs should include details about specific telemetry or test process, and they should be carefully reviewed after major changes; (2) better postmortems (17%): eliminate dependency among postmortems to quickly get insights from historical similar incident postmortems; and (3) better API documentation (33%): improve documents for certain existing tools, their repair items and missing test coverage, to help the partner teams.

Finding#10: While improving monitoring/testing accounts for majority of the lessons learnt, a significant ≈20% feedback indicated improved documentation, training, and practices for better incident management and service resiliency.

Automated mitigation. For many of the infrastructure and authentication failure related incidents, the suggestion was to eliminate human intervention and automate the entire mitigation pipeline. Half of these suggestions were related to certificate management: (1) Certificate renewal (30%): certificate rotations and renewals should be fully automated without any manual intervention; and (2) Audit (20%): certificate updates, testing and validation should be tracked and audited periodically. Other half of these suggestions were related to infrastructure management: (1) traffic failover (20%): automate traffic manager to health check backend nodes and reroute traffic after removing unhealthy nodes; and (2) node scaling (30%): re-evaluate capacity numbers frequent and set up automated scaling mechanism for faster mitigation.

External coordination. For incidents related to features that have dependency on external or partner team, we need to ensure that there is an explicit coordination mechanism and partner team properly acknowledge mitigation steps. We observe two specific lessons for this category: (1) better coordination (56%): establish a clear communication channel with partners who will be impacted when rolling out breaking changes, or eliminate dependency where applicable (e.g., removing authentication service dependency and directly fetching client tokens); and (2) better escalation (44%): need better escalation mechanism to proactively reach the partner team (e.g., by declaring outage early or designing automatic escalation method).

6 MULTI-DIMENSIONAL INCIDENT ANALYSIS

In the previous sections, we analyze high severity incidents from various individual dimensions such as *Root Causes*, *Mitigation Strategies*, and more. While this single-axis view unveils frequently used approaches and improvement opportunities, incident management is usually extraordinarily

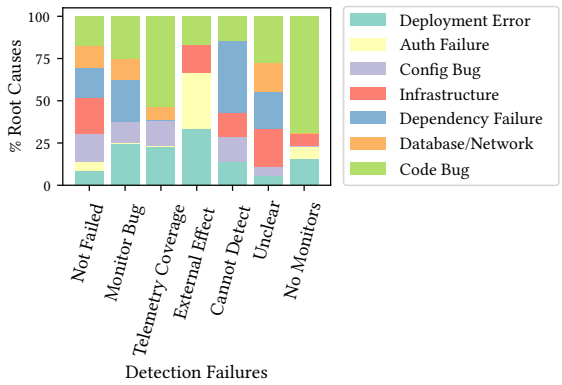

Figure 9: Detection Failures vs Root Causes

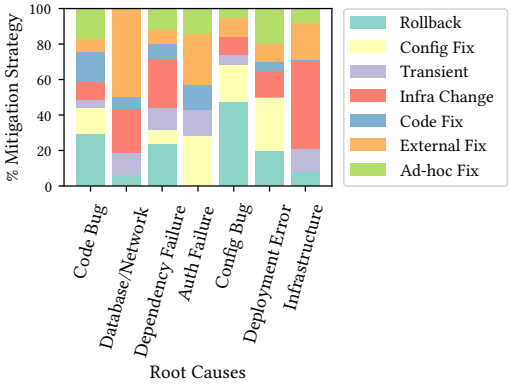

Figure 10: Root Causes vs Mitigation Strategies

complex. A single cloud service incident can involve multiple components, dependencies, teams, and on-call engineers. In such a setting, engineers handling alerts usually have a partial view of the big picture (fog-of-war) [19] and hence, react to the root cause with strategies that provide the quickest mitigation.

Example: Let's consider a customer-reported incident attributed to a recent code change. While the code change was bug-free, it exposed an incompatibility bug due to a missing null parameter check. However, alerts did not fire due to an unrelated bug in the corresponding monitors of the service. This resulted in the code being deployed beyond test rings to production rings. Given this, on-call engineers could not rollback the code change and instead applied a hot-fix to mitigate the issue.

As seen, this is an involved decision-making procedure, connecting factors such as root cause, dependencies, detection failures, and more to decide the mitigation steps. To understand such complex aspects, in this section, we look at incidents from a multi-dimensional view by analyzing the distribution of correlated factors. First, we use the Chi-Square test [26] to determine whether the association between two qualitative factors (say Root Cause and Mitigation) is statistically significant. Here, we set the null hypothesis (H_0) as the independence of two factors. We then reject H_0 (i.e. dependent factors) if the p-value of $\chi^2 <= \alpha$, where α is a chosen significance level of 0.05 (95% confidence interval). With this, we identify only the dependent factors and analyze the distribution of incidents against them. Below we discuss insights from some of our analyses and their implications (⚗) for industry and future research.

Detection Failure and Root cause. In Figure 9, we look at what *Root Causes* are associated with various kinds of *Detection Failures*. Here, we observe that a notable 70% of incidents where no monitors existed for the scenario were related to code bugs. Similarly, a significant 54% of incidents with poor telemetry coverage were also root caused to code bugs. This indicates an inherent difficulty in monitoring and detecting service health regressions due to bugs introduced via code changes. ⚗: Therefore, it seems essential to significantly test code changes before rolling out, than depend on monitoring to maintain service reliability.

> 💡 **Finding#11:** 70% of incidents with no monitors were root caused to code bugs, i.e., it is inherently difficult to monitor regressions introduced due to code changes. ⇒ For code changes, we should improve testing rather than relying on monitoring.

Additionally, we find incidents that monitors cannot detect today because they involve dependencies outside the scope of the service. Here, 42% of incidents that monitors cannot detect were associated with failures in dependency services. This indicates issues with today's monitors being localized to a service's code base, i.e., lacking a global view. ⚗: For large-scale clouds, future monitors will need to capture dependencies across services and propagate telemetry to all interacting services for faster triaging and root cause analysis. For example, related monitors can be connected and pass messages about failing services to each other.

> 💡 **Finding#12:** 42% of incidents that cannot be detected by monitoring today, were associated with dependency failures ⇒ There is a need to introduce/increase monitoring coverage and observability across related services.

Root Cause and Mitigation. In Figure 10 we analyze what *Mitigation Strategies* were used for each type of *Root Cause*. As shown, rollback- which involves reverting a build, release, code change, etc.- is one of the most popular mitigation strategies. Particularly, we observe 47% of configuration bugs mitigated with a rollback, when compared to a lesser 21% mitigated with an actual configuration fix. From Figure 3(b), we can attribute this to a rollback taking significantly less time for mitigation than a configuration fix. However, it also suggests that configuration bugs are predominantly seen in

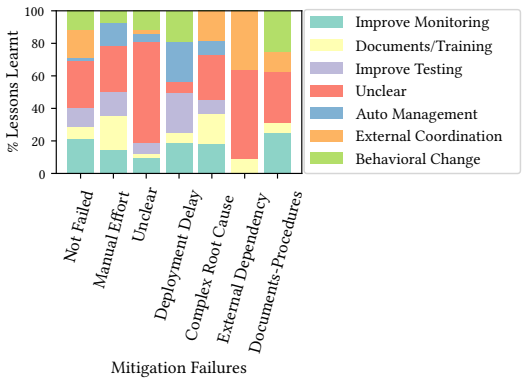

Figure 11: Mitigation Failures vs Lessons Learnt

updates to previously correct configurations, rather than configurations that unexpectedly start failing in production due to external changes such as workload, traffic, and dependencies. 🧪: Hence, a large portion of misconfigurations can be identified if tested. A de facto approach to handling misconfigurations is configuration validation – checking specified properties of configuration values. While useful, it is disconnected from the dynamic logic of the service's source code [32]. Another solution is traditional software testing, which is not customized to handle all possible configuration values and explore combinations of multiple configurations [32]. Therefore, we need to develop more systematic methods for configuration testing, that not just validate values but also test syntactically/semantically valid configuration changes that may result in unexpected service behaviour [31].

> 💡 **Finding#13:** 47% of configuration bugs mitigated with a rollback compared to a lesser 21% mitigated with a configuration fix; i.e., A large portion of misconfigurations are due to recent changes ⇒ They can be identified by rigorous configuration testing.

Also, we find that a significant 30% of incidents with deployment errors were mitigated with a configuration fix. These incidents mainly deal with deployed service certificates expiring and are mitigated by a configuration update with a new certificate version. 🧪: Here, we believe that better monitoring for certificate expiry and mechanisms for auto-renewal of certificates can improve reliability. Lastly, we observe that for 50% of database and network failures, client services depend on the external team to fix the failing infrastructure. 🧪: While this is effective, client service reliability can be further improved with auto-failover mechanisms to use alternative networks or database replicas configured during deployment.

Mitigation Failure and Lessons. Next, we look at the *Lessons* learnt by on-call engineers that could resolve various *Mitigation Failures* faced. From Figure 11, we see that in a notable 21% cases, improving Documentation and OCE training can reduce the manual effort for incident mitigation. Prior work has shown that such documents, especially troubleshooting guides (TSGs), have critical quality issues (like completeness, correctness, broken links, and readability) and are not well maintained [1, 29]. Also, TSGs frequently aid in mitigating incidents for which mitigation steps are known apriori – however, the steps are still manual. 🧪: Here, we believe two methods can help: (1) *Quality Testing*: frequently monitoring the quality of these documents using various metrics such as readability, dwell time, time-to-mitigate, up-to-dateness, etc., and (2) *Automation*: automating these manual TSGs into workflows (like jupyter notebooks) that can be executed with minimal intervention.

> 💡 **Finding#14:** 21% of incidents where manual effort delayed mitigation, expected improvements in documentation and training. ⇒ Just like with source code, we need to design new metrics and methods to monitor documentation quality. Also, automating repeating mitigation tasks can reduce manual effort and on-call fatigue.

Correspondingly, for mitigation failures due to poor Documentation and Procedures, we find that 25% expected improvements in monitoring. The most crucial issue we find here is that they can lack coverage – they don't capture dependency failures or have high thresholds and hence miss detection while the impact is low. 🧪: As previously stated, we can improve this by providing monitors global views across service dependencies and introducing clear procedures for frequently reviewing monitor thresholds. Lastly, as shown in Figure 5, even after identifying the root cause and mitigation steps, deployment delays due to manual steps result in significantly high time-to-mitigate. 🧪: To alleviate deployment delays, as in Figure 11, we see that in 25% cases, some automated mitigation steps for deployed services (e.g., certificate-rotation, traffic-failover, auto-scaling, etc.) can significantly help. As a result, this motivates future research on self-healing clouds to dynamically manage infrastructure on workload changes and other deployment activities like certificate management. A key aspect to focus on here will be connecting service monitoring to automated workflows that perform infrastructure and deployment actions.

> 💡 **Finding#15:** 25% of incidents where mitigation delay was due to manual deployment steps, expected automated mitigation steps to manage service infrastructure (like traffic-failover, node reboot, and auto-scaling).

Detection Failures and Automation. In Figure 12, we look at the distribution of *Automation Opportunities* identified

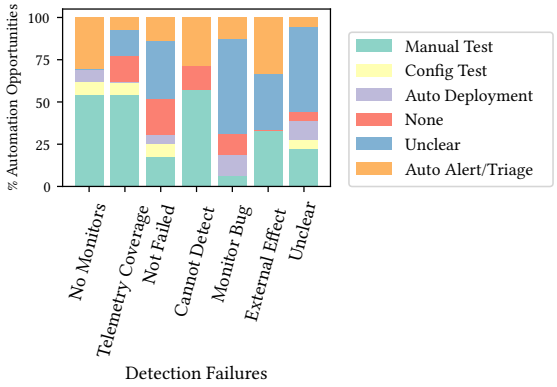

Figure 12: Detection Failures vs Automation Opportunities

by on-call engineers against *Detection Failures* that affected proactive detection of incidents. Here, a noteworthy 57% of incidents that monitors cannot detect were associated with improved manual testing over improving automated alerts (23%). Similarly, a significant 53% of incidents that had no monitors or lacked telemetry coverage, were also linked with improved manual testing. ⚗: This shows that on-call engineers find it beneficial to practice a "Shift Left" behavior that identifies incident resulting bugs before they reach production. Hence, we believe that automated testing tools can considerably help in reducing the burden of manual testing and increasing the usage of testing over monitoring for cloud reliability.

💡 **Finding#16:** In more than 50% of incidents that monitors could not detect, OCEs expected an improvement in manual testing over automated alerts (23%). ⇒ Strongly enforcing that a "Shift Left" practice with automated tools to aid testing can reduce on-call effort and expenses – both customer impact and engineering.

7 RELATED WORK

Empirical studies of incidents - There has been significant amount of prior work which has focused on analysis of incidents and outages in production systems. The prior work can be categorized into: (a) studies focused on particular type of production issues [2, 5, 11, 18, 22, 36] and, (b) study of production issues in specific services or systems [4, 10, 13, 14, 21, 34, 37]. In the first category, prior work has studied a large number of failure types such as scalability bugs [17], network partitioning failures [2], crash recovery bugs [11], exception handling [5], upgrade failures [36], partial failures [22] and task scheduling failures [8]. Alquraan et al. [2] analyzed over 130 failure reports from 25 widely used systems to characterize the impact and root causes of network partitioning failures. Similarly, Gao et al. [11] studied 103 bugs from open-source distributed systems which were caused by node crash recovery issues. However, our work is most closely related to the second category since we do a holistic analysis and characterization of 152 production issues which occurred over a span of a year. Liu et al. [21] analyzed production incidents from Microsoft Azure to analyze the type of software bugs and the mitigation process for 112 high severity incidents. Yuan et al. [34] studied 198 user reported failures in big-data systems to understand why catastrophic failures are caused and how they can be prevented. Martino et al. [10] characterize failures of a business data processing platform by using system's event log data. Unlike prior work, we do an end-to-end analysis and characterization of *all* type of production incidents in a web-scale service used by hundreds of millions of users. We study not only the root cause of incidents but also how they were detected, mitigated and the current gaps. Lastly, we do a novel correlation across these dimensions to uncover patterns and useful insights for improving incident management.

Incident management - Incident management has recently become a popular research direction in the software engineering community. There are significant challenges in incident management such as automated triaging [6, 7], diagnosis [3, 23, 24], safe deployment [20] and mitigation [16]. Chen et al. [6] conducted a large-scale empirical study which showed that incidents are frequently mis-triaged and it can lead to a delay of up to 10X with significant customer and revenue impact. Jiang et al. [16] proposed a system for recommending troubleshooting guides for incidents to reduce the mitigation time. Recent efforts [27, 28] have also focused on structured knowledge extraction from incident reports. Our work is complimentary to these prior works and the insights and learnings from this work will motivate future research into improving existing diagnosis tools and build new tools and techniques for specific categories of incidents.

8 CONCLUSION

This paper presents a comprehensive study about root causes, detection, and mitigation strategies of unprecedented high severity incidents in a cloud service serving hundreds of millions of customers. By analyzing reflections and lessons learnt by the OCEs, we identified potential automation opportunities at different granularity of root causes and mitigation strategies to improve resiliency of cloud services. Finally, our multi-dimensional correlation analysis between different stages of incident life-cycle uncovers important insights for improving reliability of large cloud services, which will provide guidance for future academic and industrial research in the field of incident management.

REFERENCES

[1] Emad Aghajani, Csaba Nagy, Mario Linares-Vásquez, Laura Moreno, Gabriele Bavota, Michele Lanza, and David C Shepherd. 2020. Software documentation: the practitioners' perspective. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 590–601.

[2] Ahmed Alquraan, Hatem Takruri, Mohammed Alfatafta, and Samer Al-Kiswany. 2018. An Analysis of {Network-Partitioning} Failures in Cloud Systems. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 51–68.

[3] Chetan Bansal, Sundararajan Renganathan, Ashima Asudani, Olivier Midy, and Mathru Janakiraman. 2020. DeCaf: Diagnosing and Triaging Performance Issues in Large-Scale Cloud Services. In *2020 IEEE/ACM 42nd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*.

[4] Ayush Bhardwaj, Zhenyu Zhou, and Theophilus A Benson. 2021. A Comprehensive Study of Bugs in Software Defined Networks. In *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 101–115.

[5] Haicheng Chen, Wensheng Dou, Yanyan Jiang, and Feng Qin. 2019. Understanding exception-related bugs in large-scale cloud systems. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 339–351.

[6] J. Chen, X. He, Q. Lin, Y. Xu, H. Zhang, D. Hao, F. Gao, Z. Xu, Y. Dang, and D. Zhang. 2019. 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)*. 111–120.

[7] J. Chen, X. He, Q. Lin, H. Zhang, D. Hao, F. Gao, Z. Xu, Y. Dang, and D. Zhang. 2019. Continuous Incident Triage for Large-Scale Online Service Systems. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 364–375.

[8] Xin Chen, Charng-Da Lu, and Karthik Pattabiraman. 2014. Failure analysis of jobs in compute clouds: A google cluster case study. In *2014 IEEE 25th International Symposium on Software Reliability Engineering*. IEEE, 167–177.

[9] Jacob Cohen. 1960. A coefficient of agreement for nominal scales. *Educational and psychological measurement* 20, 1 (1960), 37–46.

[10] Catello Di Martino, Zbigniew Kalbarczyk, Ravishankar K Iyer, Geetika Goel, Santonu Sarkar, and Rajeshwari Ganesan. 2014. Characterization of operational failures from a business data processing saas platform. In *Companion Proceedings of the 36th International Conference on Software Engineering*. 195–204.

[11] Yu Gao, Wensheng Dou, Feng Qin, Chushu Gao, Dong Wang, Jun Wei, Ruirui Huang, Li Zhou, and Yongming Wu. 2018. An empirical study on crash recovery bugs in large-scale distributed systems. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 539–550.

[12] Haryadi S Gunawi, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, Thanh Do, Jeffry Adityatama, Kurnia J Eliazar, Agung Laksono, Jeffrey F Lukman, Vincentius Martin, et al. 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.

[13] Haryadi S Gunawi, Mingzhe Hao, Riza O Suminto, Agung Laksono, Anang D Satria, Jeffry Adityatama, and Kurnia J Eliazar. 2016. Why does the cloud stop computing? lessons from hundreds of service outages. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*. 1–16.

[14] Jian Huang, Xuechen Zhang, and Karsten Schwan. 2015. Understanding issue correlations: a case study of the hadoop system. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*. 2–15.

[15] Peng Huang, Chuanxiong Guo, Jacob R. Lorch, Lidong Zhou, and Yingnong Dang. 2018. Capturing and Enhancing In Situ System Observability for Failure Detection. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI '18)*. USENIX Association, Carlsbad, CA, 1–16. https://www.usenix.org/conference/osdi18/presentation/huang

[16] Jiajun Jiang, Weihai Lu, Junjie Chen, Qingwei Lin, Pu Zhao, Yu Kang, Hongyu Zhang, Yingfei Xiong, Feng Gao, Zhangwei Xu, et al. 2020. 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*. 1410–1420.

[17] 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.

[18] Tanakorn Leesatapornwongsa, Cesar A Stuardo, Riza O Suminto, Huan Ke, Jeffrey F Lukman, and Haryadi S Gunawi. 2017. Scalability bugs: When 100-node testing is not enough. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*. 24–29.

[19] Liqun Li, Xu Zhang, Xin Zhao, Hongyu Zhang, Yu Kang, Pu Zhao, Bo Qiao, Shilin He, Pochian Lee, Jeffrey Sun, et al. 2021. Fighting the fog of war: Automated incident detection for cloud systems. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. 131–146.

[20] Ze Li, Qian Cheng, Ken Hsieh, Yingnong Dang, Peng Huang, Pankaj Singh, Xinsheng Yang, Qingwei Lin, Youjiang Wu, Sebastien Levy, et al. 2020. Gandalf: An Intelligent,{End-To-End} Analytics Service for Safe Deployment in {Large-Scale} Cloud Infrastructure. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. 389–402.

[21] Haopeng Liu, Shan Lu, Madan Musuvathi, and Suman Nath. 2019. What bugs cause production cloud incidents?. In *Proceedings of the Workshop on Hot Topics in Operating Systems*. 155–162.

[22] Chang Lou, Peng Huang, and Scott Smith. 2020. Understanding, detecting and localizing partial failures in large system software. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. 559–574.

[23] Chen Luo, Jian-Guang Lou, Qingwei Lin, Qiang Fu, Rui Ding, Dongmei Zhang, and Zhe Wang. 2014. Correlating events with time series for incident diagnosis. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*. 1583–1592.

[24] Vinod Nair, Ameya Raul, Shwetabh Khanduja, Vikas Bahirwani, Qihong Shao, Sundararajan Sellamanickam, Sathiya Keerthi, Steve Herbert, and Sudheer Dhulipalla. 2015. Learning a hierarchical monitoring system for detecting and diagnosing service issues. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 2029–2038.

[25] David Oppenheimer, Archana Ganapathi, and David A Patterson. 2003. Why do Internet services fail, and what can be done about it?. In *4th Usenix Symposium on Internet Technologies and Systems (USITS 03)*.

[26] Karl Pearson. 1900. X. On the criterion that a given system of deviations from the probable in the case of a correlated system of variables is such that it can be reasonably supposed to have arisen from random sampling. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science* 50, 302 (1900), 157–175.

[27] Amrita Saha and Steven CH Hoi. 2022. Mining Root Cause Knowledge from Cloud Service Incident Investigations for AIOps. *arXiv preprint arXiv:2204.11598* (2022).

[28] Manish Shetty, Chetan Bansal, Sumit Kumar, Nikitha Rao, Nachiappan Nagappan, and Thomas Zimmermann. 2021. Neural knowledge extraction from cloud service incidents. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 218–227.

[29] Manish Shetty, Chetan Bansal, Sai Pramod Upadhyayula, Arjun Radhakrishna, and Anurag Gupta. 2022. AutoTSG: Learning and Synthesis for Incident Troubleshooting. *arXiv preprint arXiv:2205.13457* (2022).

[30] Anselm Strauss and Juliet M Corbin. 1997. *Grounded theory in practice*. Sage.

[31] Xudong Sun, Runxiang Cheng, Jianyan Chen, Elaine Ang, Owolabi Legunsen, and Tianyin Xu. 2020. Testing configuration changes in context to prevent production failures. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 735–751.

[32] Tianyin Xu and Owolabi Legunsen. 2019. Configuration Testing: Testing Configuration Values as Code and with Code. *arXiv preprint arXiv:1905.12195* (2019).

[33] Tianyin Xu, Jiaqi Zhang, Peng Huang, Jing Zheng, Tianwei Sheng, Ding Yuan, Yuanyuan Zhou, and Shankar Pasupathy. 2013. Do not blame users for misconfigurations. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. 244–259.

[34] 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.

[35] Yuanliang Zhang, Haochen He, Owolabi Legunsen, Shanshan Li, Wei Dong, and Tianyin Xu. 2021. An evolutionary study of configuration design and implementation in cloud systems. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 188–200.

[36] 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.

[37] Feng Zhu, Lijie Xu, Gang Ma, Shuping Ji, Jie Wang, Gang Wang, Hongyi Zhang, Kun Wan, Mingming Wang, Xingchao Zhang, et al. 2022. An Empirical Study on Quality Issues of eBay's Big Data SQL Analytics Platform. (2022).