ChangeRCA: Finding Root Causes from Software Changes in Large Online Systems

GUANGBA YU, Sun Yat-sen University, China
PENGFEI CHEN, Sun Yat-sen University, China
ZILONG HE, Sun Yat-sen University, China
QIUYU YAN, Tencent, China
YU LUO, Tencent, China
FANGYUAN LI, Tencent, China
ZIBIN ZHENG, Sun Yat-sen University, China

In large-scale online service systems, the occurrence of software changes is inevitable and frequent. Despite rigorous pre-deployment testing practices, the presence of defective software changes in the online environment cannot be completely eliminated. Consequently, there is a pressing need for automated techniques that can effectively identify these defective changes. However, the current abnormal change detection (ACD) approaches fall short in accurately pinpointing defective changes, primarily due to their disregard for the propagation of faults. To address the limitations of ACD, we propose a novel concept called root cause change analysis (RCCA) to identify the underlying root causes of change-inducing incidents. In order to apply the RCCA concept to practical scenarios, we have devised an intelligent RCCA framework named *ChangeRCA*. This framework aims to localize the defective change associated with change-inducing incidents among multiple changes. To assess the effectiveness of *ChangeRCA*, we have conducted an extensive evaluation utilizing a real-world dataset from WeChat and a simulated dataset encompassing 81 diverse defective changes. The evaluation results demonstrate that *ChangeRCA* outperforms the state-of-the-art ACD approaches, achieving an impressive Top-1 Hit Rate of 85% and significantly reducing the time required to identify defective changes.

CCS Concepts: • **Software and its engineering → Maintaining software**; **Software reliability**; **Software performance**.

Additional Key Words and Phrases: Software Change, Root Cause Analysis, Online Systems

ACM Reference Format:
Guangba Yu, Pengfei Chen, Zilong He, Qiuyu Yan, Yu Luo, Fangyuan Li, and Zibin Zheng. 2024. ChangeRCA: Finding Root Causes from Software Changes in Large Online Systems. *Proc. ACM Softw. Eng.* 1, FSE, Article 2 (July 2024), 23 pages. https://doi.org/10.1145/3643728

1 INTRODUCTION

In modern software engineering, monolithic applications are experiencing a transformation towards microservice architecture to enhance agility and reduce the time of new feature releases by enabling continuous integration and continuous delivery (CI/CD). Despite the implementation of rigorous

Authors' addresses: Guangba Yu, Sun Yat-sen University, Guangzhou, China, yugb5@mail2.sysu.edu.cn; Pengfei Chen, Sun Yat-sen University, Guangzhou, China, chenpf7@mail.sysu.edu.cn; Zilong He, Sun Yat-sen University, Guangzhou, China, hezlong@mail2.sysu.edu.cn; Qiuyu Yan, Tencent, Shenzhen, China, ireneyan@tencent.com; Yu Luo, Tencent, Shenzhen, China, zekaluo@tencent.com; Fangyuan Li, Tencent, Shenzhen, China, leiffyli@tencent.com; Zibin Zheng, Sun Yat-sen University, Zhuhai, China, zhzibin@mail.sysu.edu.cn.

ACM 2994-970X/2024/7-ART2
https://doi.org/10.1145/3643728

testing methodologies prior to deployment and the utilization of canary release strategies [33], bugs persist in the online environment due to the inherent heterogeneity of hardware and software systems, as well as the complex interactions between various components [62]. Empirical studies conducted on incidents generated in popular cloud platforms reveal that more than 40% of these incidents are directly correlated to software changes [13, 22]. Therefore, improper management of these changes can cause severe consequences, negatively impacting user experience and business revenue [1, 11, 16, 28, 39].

Consequently, it is imperative for Site Reliability Engineers (SREs) to closely monitor services that undergo changes using Key Performance Indicators (KPIs) to promptly identify and mitigate defective changes during their earliest stages. Early identification of a defective change enables software developers to halt the detrimental change and promptly initiate a rollback. However, a larger online systems typically contains thousands of service and the frequency of software changes in distributed systems can reach thousands of deployments per day [38]. The manual identification of defective changes by SREs is not only time-consuming but also prone to errors.

Significant efforts have been made to automate the mitigation of the impact caused by defective changes, with a focus on either abnormal change detection (ACD) [4, 30, 49, 57, 60, 63] or root cause analysis (RCA) [17, 23, 35, 51, 52, 56, 61]. However, we identify two primary limitations of existing ACD or RCA approaches in localize defective changes from massive changes.

- **The overlook of anomalous propagation of defective changes in ACD.** Existing ACD approaches pay close attention to determine the presence of a defective change. These approaches typically assess the abnormality of individual changes within a single service, without taking into account the potential impact of abnormal change propagation from other services within the system. Existing ACD approaches fall short in identifying defective changes in complex scenarios that involve abnormal change propagation (details presented in § 3.3.3).
- **Insufficient exploitation of change data in RCA.** Most existing RCA approaches typically utilize system runtime information (e.g., metrics) as input while neglecting the corresponding change data (e.g., change flow). As a result, these approaches tend to localize root causes at the service level rather than at the fine-grained change level. Once RCA results are obtained, SREs are required to manually analyze which changes are responsible for the incidents, resulting in a prolonged time to identify (TTI). A more comprehensive RCA analysis should be conducted for change-inducing incidents to establish a clear linkage back to defective changes.

RCCA. To overcome the aforementioned limitations, we propose a novel concept called root cause change analysis (RCCA) to identify the root cause changes of change-inducing incidents. In this study, a root cause change refers to the defective change that triggers an incident. RCCA takes into account the interdependencies among changes across different services, allowing for a comprehensive analysis to localize root cause changes.

ChangeRCA. To apply the RCCA concept to practical scenarios, we design an intelligent RCCA framework, *ChangeRCA*, to localize the defective change of change-inducing incidents among multiple changes. The key idea of *ChangeRCA* is to combine valuable information from the difference between pre-change and post-change instances, change flow, and service dependency graph to perform RCCA effectively. When an incident occurs, suppose that *ChangeRCA* is triggered with RCA service $\mathcal{S}_{rca}$, *ChangeRCA* encompasses three primary stages. Stage ① *Defective Canary Change Identifier* (§ 5.2) ascertains whether the incident is induced by a defective canary change. If not, stage ② *Non-change Fault Identifier* (§ 5.3) investigates whether the incident is induced by other non-change faults. If neither of the previous stages identifies the cause, stage ③ *Suspicious Change Scorer* (§ 5.4) identifies the suspicious changes within the service dependency graph of $\mathcal{S}_{rca}$ for SREs to check, ultimately helping SREs to localize the root cause change.

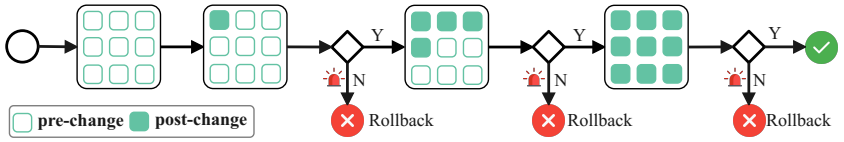


Fig. 1. Workflow of a canary software change in industry.

Results. We conduct a comprehensive study on a real-world dataset from a large-scale online system WeChat and a simulated dataset from a widely-used microservice benchmark OnlineBoutique (OB) [15] to evaluate the performance of *ChangeRCA*. Our experimental results demonstrate that *ChangeRCA* can identify defective changes with 85% *HR*@1 (Top-1 Hit-Rate), 96% *HR*@3, and outperform the ACD methods by 20%∼28% in *HR*@1. Furthermore, *ChangeRCA* can locate 90% of defective changes in less than 3 minutes in WeChat, a 90% reduction compared to ACD approaches.

Contributions. This study makes the following contributions:

- We propose a novel concept called root cause change analysis (RCCA) to identify the root cause changes of change-inducing incidents. This concept takes into account the interdependencies among changes across different services, providing a comprehensive approach for RCCA (§ 4.1).
- We introduce a novel approach called *ChangeRCA* that effectively identifies root cause changes from a large number of normal changes. *ChangeRCA* assists SREs in focusing their attention on defective changes and taking appropriate actions to prevent further service outages (§ 5).
- Extensive experiments show that *ChangeRCA* outperforms all the start-of-the-art approaches, achieving 96% *HR*@3 and effectively reducing the identification time. We have made our tool and OnlineBoutique data available on GitHub [6] (§ 6).

2 BACKGROUND AND MOTIVATION

2.1 Software Canary Release

In online service systems, software changes are both frequent and inevitable, necessitated by the introduction of novel features, resolution of extant bugs, adaptation to evolving environmental conditions, and enhancement of overall performance. To mitigate the inherent risks associated with deploying new software versions, the canary release strategy has emerged as a prevalent technique in software deployment [33]. This approach entails the incremental deployment of modifications to a select group of users or a limited number of servers before the comprehensive implementation across the entire user base or infrastructure. As illustrated in Fig. 1, the canary release process for a given service typically encompasses the following sequential steps,

(1) Developers deploy the new version of a service to a restricted number of users or servers.
(2) Developers monitor the performance, stability, and user feedback of the post-change version within the canary environment.
(3) If the post-change version exhibits suboptimal performance or issues are identified, developers initiate a rollback of the post-change version to rectify the problem. Alternatively, developers expand the release to more users or servers.
(4) Upon successful testing and confirmation of stability, the post-change version is deployed across the entire user base or infrastructure step-by-step.

Introducing the change to actual production traffic enables developers to identify problems that might not be visible in testing frameworks. The canary change strategy enables developers to minimize the ramifications of potential issues arising from the new software version by confining exposure to a narrowly defined group of users.

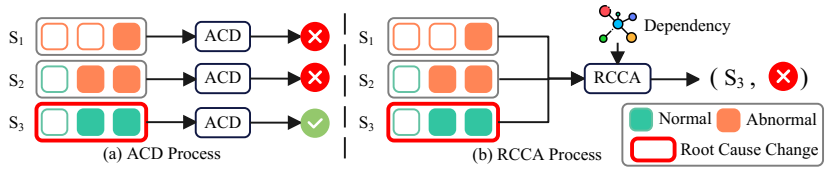


Fig. 2. Comparison between abnormal change detection (ACD) and root cause change analysis (RCCA).

2.2 Comparison between ACD and RCCA

As the state-of-the-art defective change identification technique, current abnormal change detection (ACD) approaches [50, 60, 63] typically employ continuous monitoring of KPIs or other failure signals for pre- and post-change instances. Subsequently, they quantify the disparities in KPIs between these groups to identify defective changes and provide recommendations for decision-making, such as "go" or "no-go". In this study, we present the formulation of the ACD process as follows: given a system change action C planned to be rolled out on service $\mathcal{S}$, the ACD approach is utilized to detect whether C introduces anomalies to $\mathcal{S}$. If $\mathcal{S}$ is identified as exhibiting abnormal behavior, the change is promptly terminated and rolled back.

In real-world applications, it is common to have multiple change actions occurring within a condensed timeframe. As shown in Fig. 2 (a), when confronted with multiple change actions, current ACD approaches detect software changes individually, disregarding service dependencies and fault propagation across software changes. Therefore, such ACD approaches are prone to false positives when confronted with silent defective changes and fault propagation. For example, the abnormal behavior of S_1 and S_2 in Fig. 2 (a) is caused by fault propagation from silent defective change of S_3. Current ACD approach (e.g., SCWarn [63]) will consider the changes in S_1 and S_2, which exhibit anomalies, as anomalous changes, and consider the real defective change of S_3 as normal changes. We delve into the specifics of false localization cases in § 3.3.3.

To address the limitations of ACD approaches, we propose a groundbreaking concept for identifying root cause changes, namely root cause change analysis (RCCA). We define a root cause change as the defective change that causes an incident. As shown in Fig. 2 (b), RCCA adopts a holistic perspective of applications and considers the interferences between multiple changes to identify the most suspicious ones. We formalize the RCCA process in § 4.1. Compared to conventional ACD approaches, RCCA takes into account the interdependencies among changes across various services, thereby facilitating a more comprehensive analysis for localizing root cause changes and making rollback decisions.

2.3 Relationship between RCA and RCCA

Root cause analysis (RCA) is a crucial process in system analysis aimed at identifying the component or module responsible for deviating from expected behavior. By localizing the root cause, SREs can gain a deeper understanding of the underlying issue and initiate appropriate remedial actions. However, existing RCA approaches [17, 23, 52, 54, 56] primarily rely on observability data, such as metrics and traces, while often neglecting the significance of change data. Consequently, these RCA approaches may struggle to accurately pinpoint the defective changes in scenarios where failures are induced by system changes. Consequently, SREs frequently find themselves having to proceed manually to the next stage of locating the defective changes.

RCCA extends the capabilities of RCA by specifically focusing on identifying the changes responsible for anomalies. By incorporating both observability and change data as input, RCCA provides a more granular perspective, enabling SREs to determine the specific change actions that require examination or potential rollback to rectify the problem.

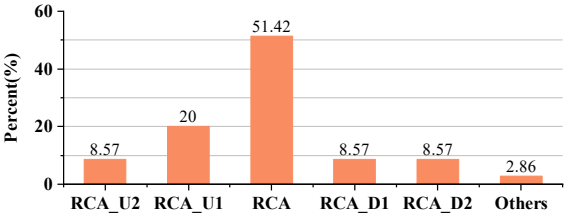


Fig. 3. Differences between automatic RCA result and defective change service.

Table 1. Distribution of change-inducing incidents in WeChat and ACD results.

Type	Funnel	SCWarn	Gandalf
Model	0/2	2/2	0/2
Resource	2/4	4/4	3/4
Configuration	2/4	2/4	3/4
Backend	13/20	13/20	14/20
Total	17/30	21/30	20/30

To illustrate the relationship between RCA and RCCA, we present a comparative analysis in RCA services and defective change services of change-inducing incidents from WeChat in a whole year of 2022. Figure 3 shows the differences between these services. The RCA system employed in WeChat adopts a graph neural network model and a PageRank algorithm with a flexible transition matrix to localize suspicious services. In Fig. 3, the label "RCA" indicates instances where the RCA results align with the defective change service. "RCA_Ui" and "RCA_Di" indicate that the defective change service is either an upstream service or a downstream service with a depth of i to the RCA service.

As depicted in Fig. 3, we find that in 51% of incidents, the RCA services coincide with the defective change services. However, these RCA services do not explicitly consider their association with defective changes. In the remaining incidents, approximately 46% of defective change services are either upstream or downstream services of the RCA services within a depth of 2. These observations highlight the interdependence of RCA and RCCA: **RCA can effectively assist RCCA in narrowing down the search scope for defective changes.** Rather than designing a comprehensive end-to-end RCCA algorithm, it is more efficient and cost-effective for enterprises to leverage the results obtained from existing RCA approaches and design an RCCA framework specifically tailored for defective changes.

3 STUDY ON ACD APPROACHES

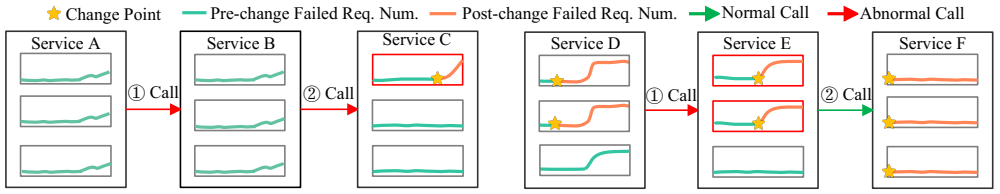
In this section, we aim to evaluate the performance of the current ACD approaches using real change-inducing incidents and investigate whether the limitations of ACD approaches have a significant impact on change localization. Specifically, we try to answer the following research questions (RQs),

- **RQ1**: How does ACD perform on the real dataset regarding number of defective changes detected?
- **RQ2**: Why does ACD succeed in detecting some defective changes?
- **RQ3**: Why does ACD fail to detect some defective changes?

3.1 Data Collection

To conduct our study, we first collect 30 real-world change-inducing incidents from WeChat, a large-scale online system widely used by billions of users worldwide. WeChat is a representative online system that covers a variety of application scenarios (e.g., instant messaging, social media, and mobile payment) and is implemented in several programming languages [53]. The frequency of software change in WeChat can reach thousands of deployments per day. These incidents encompass diverse domains such as instant messaging, social media, and mobile payment, and encompass various change scenarios, including 2 model changes, 4 resource changes, 4 configuration changes, and 20 backend changes.

To facilitate incident analysis and prevent their recurrence, engineers responsible for these incidents are obligated to document the entire fault-handling process in the form of post-mortems. These post-mortems contain crucial information such as fault manifestation, alert time, automatic



(a) Incident caused by config change of Service C (b) Incident caused by backend change of Service E

Fig. 4. Successfully-identified cases. Real service name and instance counts are anonymized due to privacy.

RCA result, final confirmed RCA results by the responsible engineers, and other pertinent data. Regarding root cause change labeling, three authors independently label the root cause change of each incident by carefully analyzing the troubleshooting steps and incident reason descriptions documented in the post-mortems. The process is conducted under the supervision of the "Cohen's Kappa coefficient" [41] to ensure reliability and consistency. In cases where disagreements arise, consultations are pursued until a consensus is reached.

3.2 Start-of-the-Art ACD Approaches

We explore the performance of the following three state-of-the-art ACD approaches on real datasets.

- **FUNNEL** [60] leverages the singular spectrum transform algorithm to detect performance changes and employs the Difference-in-Differences (DiD) [40] approach to detect defective change. DiD allows for a comparative analysis of the system's behavior before and after a change.
- **SCWarn** [63] keeps monitoring the multi-source observability data of changed services. It utilizes a multivariate Long Short-Term Memory (LSTM) [18] approach to capture temporal dependencies and patterns so as to detect anomalies associated with defective changes.
- **Gandalf** [24] monitors various fault signals (e.g., OS events) and uses Holt-Winters forecasting [7] to detect error instances. If a significant number of error instances occur after a service change, Gandalf correlates the anomaly with the specific change based on the change time.

3.3 Results and Analysis

3.3.1 ***RQ1: Number of Successfully-identified Defective Changes.*** As supported by the empirical results presented in Table 1, the ACD methods, namely FUNNEL, SCWarn, and Gandalf, exhibit the capability to identify defective changes in a range of change-inducing incidents. Specifically, FUNNEL successfully detects 17 out of 30 incidents, accounting for 56.67% of the cases. SCWarn and Gandalf correctly identify 20 and 21 defective changes, respectively. To gain deeper insights into the performance shortcomings of ACD approaches within the context of the WeChat scenario, we aim to scrutinize the individual cases where these methods either succeed or fail in identifying defective changes.

3.3.2 ***RQ2: Successfully-identified Defective Changes.*** The successfully-identified changes are predominantly achieved through the comparative analysis of KPI differences between pre- and post-change instances.

Successfully-identified Case I: In Fig. 4 (a), an inconsistency arises in the new configuration files of service C between the test and production environments due to a developer error. Consequently, the post-change instances of service C fail to handle incoming requests, leading to an escalation in failed requests for post-change instances. Conversely, the pre-change instances of C behave stable. By leveraging FUNNEL, SCWarn, and Gandalf, the defective change of C is successfully pinpointed by comparing the KPI differences between pre- and post-change instances.

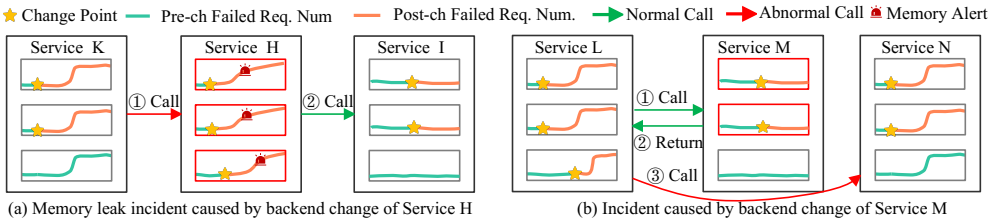


Fig. 5. Failed-identified cases. Real service name and instance count are anonymized due to privacy.

Successfully-identified Case II: In Fig. 4 (b), a defective code segment is introduced to service E, causing the post-change instances to crash. As depicted in Fig. 4 (b), the defective change manifests as an increase in failed requests for the post-change instances of E. Given the evident disparities between the pre- and post-change instances, FUNNEL, SCWarn, and Gandalf effectively detect this defective change. However, it is important to note that three ACD approaches also flag the change of D as a defective change, because the failed requests to D also increase due to fault propagation.

3.3.3 ***RQ3: Failed-identified Defective Changes.*** For failed-identified changes, most of them do not exhibit noticeable differences in KPIs between the pre- and post-change instances.

Failed-identified Case I: In Fig. 5 (a), a defective code segment was integrated into a new version of service H. This code segment introduced a slow memory leak, which caused H not to behave abnormally until 12 hours after the change was completed. However, existing approaches such as FUNNEL and SCWarn primarily focus on detecting change failures during the canary change phase, disregarding failures that occur after the changes have been fully implemented. Moreover, once a change is completed, all instances of the service become post-change instances, making it difficult for the existing ACD approaches to identify the defective change by comparing pre-change and post-change instance groups.

Failed-identified Case II: In Fig. 5 (b), after a software change of service M, it modified its return value type. Service L called M to obtain the return value and used it to call N. However, the new return value type was incompatible with the requirements of service N, resulting in a failed call from N to L. In this case, ACD approaches are unable to detect the defective change in M because its post-change instances do not exhibit any KPI or signal anomalies. Thus, solely considering the KPIs and signals of an individual service is insufficient to identify silent defective changes that do not behave any KPI or signal anomalies.

To sum up, it is evident that existing ACD approaches face challenges in detecting defective changes that do not manifest significant KPI differences and fail to account for failures occurring after the completion of changes. Additionally, relying solely on KPIs and signals of updated service proves inadequate for identifying silent defective changes. Analyzing the impact of silent defective changes on upstream and downstream based the service dependency graph and thus inferring the root cause change is a promising solution, which is an important motivation for this paper.

3.4 Enlightenment

According to the findings of above RQs, we could infer some guidelines for designing RCCA:

(1) **Utilizing comparative analysis clues.** By examining the differences in KPIs or failure signals between pre- and post-change instances, anomalies brought about by the change can be identified. This comparative analysis serves as an effective means to identify defective changes that manifested themselves during the canary change phase.
(2) **Considering service dependency graphs.** Analyzing interactions between services within the system is important to understand the impact of a defective change on the overall system. By

incorporating service dependency into process, it becomes possible to identify silent defective change and reduce false positives.

(3) **Pay more attention to silent changes.** Even if a service does not exhibit apparent anomalies, its changes are still at risk of being defective. The presence of a silent defective change can propagate to its upstream or downstream service through service interaction.

(4) **Monitoring completed changes.** Certain defective changes may not immediately reveal their impact following the update but rather emerge after the change completion. Consequently, continuous monitoring subsequent to change completion becomes indispensable to identify these latent defective changes effectively.

4 A NEW RCCA FRAMWORK: CHANGERCA

Inspired by the findings of study on ACD, we propose a new concept for identifying root cause changes, namely *Root Cause Change Identification (RCCA)*. To apply the RCCA concept to practical scenarios, we design an intelligent RCCA framework, *ChangeRCA*, for large-scale online systems.

4.1 RCCA Problem Definition

Consider a large-scale online system consisting of N services and the dependency graph $\mathcal{G}$, where service $\mathcal{S}_n$ has a change ticket set represented by $C_n = \{c_n^1, ..., c_n^m\}$. Here, c_n^m denotes the mth change ticket of $\mathcal{S}_n$. Each change ticket documents the entire process of a single change, including details such as change time and change status for each service instance. For an instance $\mathcal{S}_n^i$ of $\mathcal{S}_n$, the KPI time series are given by $\mathcal{K}_n^i = \{k_n^{i1}, ..., k_n^{ij}\}$, with k_n^{ij} representing the time series of a KPI j of $\mathcal{S}_n^i$.

The primary objective of RCCA is to determine if an incident is induced by defective changes and estimate the suspiciousness score for suspicious changes. To achieve this, we formalize RCCA based on a parameterized model $\mathcal{F} : (\mathcal{S}, \mathcal{C}, \mathcal{K}, \mathcal{G}) \rightarrow (\mathcal{T}, Score)$. $\mathcal{T}$ is a three-category indicator, where 0 denotes a defective canary change, 1 signifies a non-change fault, and 2 represents a possible defective change. The suspiciousness score, denoted by $Score$, is represented by $Score = [score_1, ..., score_M]$. If $\mathcal{T}$ equals 0, $Score$ contains only defective canary changes. If $\mathcal{T}$ equals 1, $Score$ contains no items. Otherwise, $Score$ is a sorted list containing the most suspicious changes.

4.2 Overview of ChangeRCA.

Figure 6 illustrates the overall analysis process of *ChangeRCA*. *ChangeRCA* is invoked by the *RCCA Trigger* (§ 5.1), typically implemented as a fault diagnosis system. Once RCCA Trigger identifies that service $\mathcal{S}_{rca}$ is suspicious and initiates the *ChangeRCA* process, the framework progresses through three cascade stages, each addressing specific fault mitigation objectives.

Stage ①: *Defective Canary Change Identifier* (§ 5.2) determines whether the anomaly in $\mathcal{S}_{rca}$ is caused by a defective canary change. This is accomplished by comparing the KPIs of pre- and post-change instances. In the event of a defective canary change, *ChangeRCA* presents the canary change ticket and recommends a rollback to SREs.

Stage ②: If no defective changes are identified in the previous stage, *Non-change Fault Identifier* (§ 5.3) is utilized to determine whether $\mathcal{S}_{rca}$ is affected by a non-change fault (e.g., inadequate resources). This identification facilitates appropriate mitigation actions (e.g., resource scaling).

Stage ③: In the absence of the aforementioned faults, *Suspicious Change Scorer* (§ 5.4) integrates information from change tickets, dependency graphs, and KPIs to produce an ordered list of suspicious change tickets. This prioritized list enables SREs to efficiently examine and investigate suspicious change tickets, aiding in effective problem resolution.

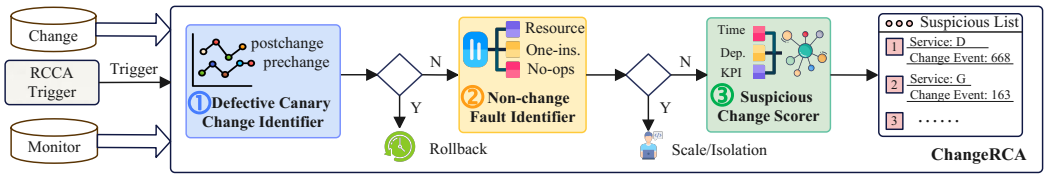


Fig. 6. ChangeRCA framework.

5 DETAIL DESIGN

5.1 RCCA Trigger

As discussed in § 2.3, RCCA extends the capabilities of RCA, which is typically performed at the service level, by specifically focusing on identifying defective changes. In this study, we adopt state-of-the-art RCA approaches as the *RCCA Trigger*. The default RCA approach is GIED [17] which adopts a graph neural network model and a PageRank algorithm with a flexible transition matrix to localize suspicious services. We choose GIED because it has been proven effective in industrial systems.

There are several reasons why we choose to use existing RCA approaches instead of designing a new one. Firstly, designing a new RCA method for industrial systems from scratch can be a complex and time-consuming process. Given the constraints of paper length, providing an in-depth discussion on RCA design is beyond the scope of this paper. Secondly, the existing RCA methods [17, 42] have already undergone extensive testing and have been proven effective at the service level in industrial systems. As depicted in Fig. 3, the state-of-the-art RCA method can effectively assist *RCCA* in narrowing down the search scope for defective changes.

Moreover, *ChangeRCA* can be readily integrated with different RCA approaches. In § 6.5, we will discuss the impact of different RCA approaches on the effectiveness of *ChangeRCA*.

5.2 Defective Canary Change Identifier

Upon activation by the suspicious service $\mathcal{S}_{rca}$, stage ① introduces the Defective Canary Change Identifier, hereafter referred to as Canary Identifier. Its primary objective is to detect whether a canary change has occurred in $\mathcal{S}_{rca}$ and further determine if the canary change is defective.

The core idea underlying Canary Identifier is as follows. If an incident is caused by defective changes, these defects will only affect the post-change instances while leaving the pre-change instances unaffected. Conversely, incidents caused by other factors will impact both pre- and post-change instances. By evaluating the significance of the difference between pre- and post-change instances, we can determine if an incident is attributable to a defective change. For example, as shown in Fig. 4 (b), a defective change in E results in a surge in the number of failed requests. Notably, the post-change instances (marked with a pentagram) exhibit a spike in failed requests, while the pre-change instance remains stable. Consequently, the likelihood that the KPI changes are caused by a defective change is high.

Drawing inspiration from Enlightenment (1) in § 3.4, our study leverages the concept of split testing to assess whether significant differences exist between KPIs of the treatment group (i.e., post-change instances) and the control group (i.e., pre-change instances). Difference-in-Differences (DiD) [40] is a widely recognized econometric technique, which enables a comparative analysis between the treatment and control groups, facilitating the estimation of the causal effect resulting from a intervention. Consequently, existing works such as FUNNEL [60] have employed DiD to quantify the discrepancies between pre- and post-change instances.

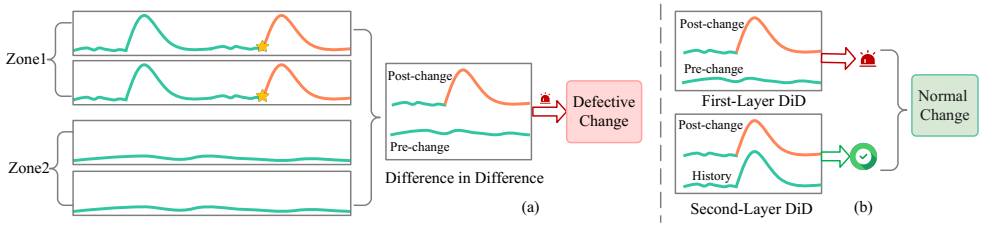


Fig. 7. Example of different instances of the same service in two zones with distinct seasonal patterns.

Nevertheless, DiD assumes that any pre-existing seasonal patterns remain consistent across both the treatment and control groups. However, one service in different zones (e.g., Asia or Europe) may have different seasonal patterns. In such situations, estimated treatment effects may become confounded by these variations, subsequently leading to biased outcomes. For instance, as shown in Fig. 7, the post-change instances in zone1 experience a pronounced seasonal surge during a specific period, while the pre-change instances in zone2 remain unaffected. In this case, applying DiD without accounting for these confounding seasonal variations can yield misleading results.

To address this concern and enhance the reliability of causal inference, *ChangeRCA* incorporates a cascade DiD framework, which consists of two layers of DiD. The first-layer DiD quantifies the discrepancies between pre- and post-change instances. If first discrepancies is significant, the second-layer DiD compares the KPI differences between post-change instances and their historical contemporaries. If second discrepancies is insignificant, *ChangeRCA* determines that the anomaly was induced by period. Otherwise, *ChangeRCA* determines the anomaly was induced by the defective change. Specifically, we adopt the same approach as GIED [17], setting the historical contemporaneous period to the previous day.

According to the first-layer DiD framework, suppose that $K_{pre}(k|C = 0)$, $K_{post}(k|C = 0)$ represent the KPIs before change C at pre- and post-change instances respectively. Similarly, $K_{pre}(k|C = 1)$, $K_{post}(k|C = 1)$ denote the KPIs after change C at pre- and post-change instances respectively. To obtain the standard errors and significance levels for the DiD estimator, we use a linear parametric model to model KPIs values. Then the first-layer DiD is calculated as follows,

$$d(post, pre) = \{E[K_{post}(k|C = 1)] - E[K_{post}(k|C = 0)]\} - \{E[K_{pre}(k|C = 1)] - E[K_{pre}(k|C = 0)]\}. \tag{1}$$

If the KPI changes are caused by factors other than software changes, there should be no significant change in the relative performance between the pre- and post-change instances, resulting in $d(post, pre)$ being close to zero. Conversely, if $d(post, pre) \ll 0$ or $d(post, pre) \gg 0$, it indicates a high likelihood that the performance changes are caused by a software change.

Subsequently, *ChangeRCA* calculates the p-value associated with $d(post, pre)$. We select a significance level λ (with a default value of $\lambda = 0.05$) for the hypothesis test. If the p-value of $d(post, pre)$ is less than λ, the DiD estimate is deemed statistically significant.

For the second-layer DiD framework, $K_{his}(k|C = 0)$ represents the KPIs before change C on the previous day, and $K_{his}(k|C = 1)$ represents the KPIs after change C on the previous day. The second-layer DiD is calculated as follows,

$$d(post, his) = \{E[K_{post}(k|C = 1)] - E[K_{post}(k|C = 0)]\} - \{E[K_{his}(k|C = 1)] - E[K_{his}(k|C = 0)]\}. \tag{2}$$

If the KPI changes are periodical such as system daily routines, there should be no significant change in the relative performance between the post-change and contemporaneous period, leading to $d(post, his)$ being close to zero. Conversely, if $d(post, his) \ll 0$ or $d(post, his) \gg 0$, it indicates a low likelihood that the performance changes are caused by a periodical factor. We use the same p-value method as the first-layer DiD to estimate significance.

For a post-change instance, if $d(post, pre)$ is significant while $d(post, his)$ is not significant, the post-change instance would be determined as an abnormal instance induced by a defective change. After processing each post-change instance and each KPI of a service in parallel, *ChangeRCA* obtains the number of abnormal post-change instances N_{apost}. Finally, we calculate the suspicious score of the change, denoted as $Score_C$, by calculating the percentage of anomalies among the post-change instances,

$$Score_C = \frac{N_{apost}}{N_{post}}, \tag{3}$$

where N_{post} denotes the total number of post-change instances. A score threshold η (default value of 0.8) is set for determining defective canary changes. If $Score_C$ is greater than η, *ChangeRCA* concludes that $\mathcal{T}$ equals 0 and recommends SREs to rollback the defective change. This approach eliminates the need to consider other changes, significantly reducing computational overhead.

However, Canary Identifier is not a silver bullet. It cannot localize the root cause change for the failed-identified case I and II (§ 3.3.3), where the KPIs of H appeared abnormal after the change and the KPIs of M remained stable after the change. This is why *ChangeRCA* requires stage ② and ③.

5.3 Non-change Fault Identifier

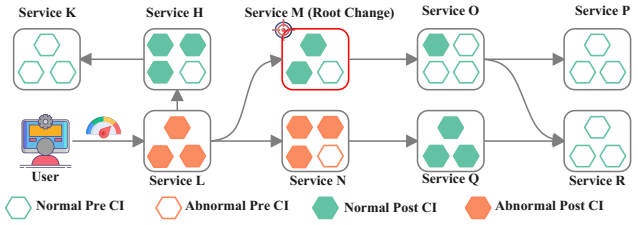
If $\mathcal{S}_{rca}$ cannot be attributed to the defective canary change, it is customary for SREs to investigate whether $\mathcal{S}_{rca}$ is caused by other non-change faults. Consequently, in stage ②, *Non-change Fault Identifier* module is employed within *ChangeRCA* to eliminate non-change faults. This prevents SREs from examining change tickets associated with faults unrelated to changes.

From the perspective of SREs, resource-related faults, one-instance faults, and non-ops faults are the three most prevalent types of non-change faults [43]. Therefore, three pluggable fault identification modules, namely *Resource Fault Identifier*, *One-instance Fault Identifier*, and *Non-Ops Fault Identifier*, are introduced within the ChangeRCA framework to provide actionable suggestions for SREs. Additionally, the pluggable nature of Non-change Fault Identifier module allows for easy integration of additional fault identification modules, such as fault sketching [25], offering flexibility in enhancing its capabilities.

5.3.1 Resource Fault Identifier. Resource faults occur when a service's resources, such as CPU, memory, or disk, reach maximum capacity or utilization, resulting in performance degradation or service failure. Resource Fault Identifier is responsible for detecting resource faults by analyzing the resource utilization rate of services and establishing utilization thresholds. These thresholds indicate whether a service has reached saturation. Empirically, for resource-sensitive services like online shopping platforms, a lower threshold, such as 50%, may be set for the resource utilization rate. In other cases, a higher threshold, such as 80%, may be more appropriate. If a resource fault is detected, *ChangeRCA* reports to SREs and provides recommendations for resource scaling.

5.3.2 One-instance Fault Identifier. A one-instance fault, occurs when only a few service instances experience issues, while the remaining instances continue to operate normally. These faults can be caused by various factors, such as hardware failures, affecting only a limited number of instances. One-instance Fault Identifier employs the DiD approach described in § 5.2 to obtain the abnormal instance number N_a. If $N_a \leq N_{\min}$, One-instance Fault Identifier determines that a one-instance fault caused the alert. Empirically, $N_{\min}$ is set to 2 in WeChat, as SREs believe that more than 2 abnormal instances warrant further investigation. If a one-instance fault is identified, auto-isolation of faulty instances can quickly alleviate the issue.

5.3.3 Non-Ops Fault Identifier. Non-ops faults refer to faults that automatically recover to normal states without any human intervention [22, 43]. Incidents induced by defective changes typically exhibit non-transient behavior, lasting for an extended period without human intervention. On the other hand, transient anomalies, such as temporary network issues, are more likely to be considered



RCA System Ticket

RCA Time	RCA Service
2023-03-03 16:16:06	Service L

Service Change Flow

Change Time	Ticket	Instance
2023-06-06 16:15:06	12345	Service M-2
2023-06-06 16:13:09	12345	Service M-1
2023-06-06 11:08:08	12343	Service N-2
2023-06-06 10:36:56	12340	Service L-3

Fig. 8. Part of dependency graph and change flow of the case in Fig 5(b). Service H, L, M, N, O and Q undergo software changes. Service M encounters a defective change and propagates faults to L and M.

as noises. Therefore, Non-Ops Fault Identifier filters out faults if the abnormal behavior is back to normal state before performing RCCA.

Overall, the above fault identifiers primarily focus on filtering out faults that are clearly identified as non-change faults. By filtering out these clearly identified non-change faults in the early stages, the subsequent stage can focus on investigating and determining faults that may or may not be change faults. While the above fault identifiersare empirically developed, they fulfill the lightweight design requirements and the fault type are easy to reason about since they are based on domain knowledge and empirical debugging experience. This approach allows for a more efficient effort.

5.4 Suspicious Change Scorer

The previous stages provide a mechanism for SREs to identify and promptly address faults by categorizing them into specific fault types. However, there are situations where certain faults cannot be definitively classified into any particular category. For instance, Fig. 8 presents a partial dependency graph and change flow related to the scenario depicted in Fig. 5 (b). In this example, services C, F, and G do not exhibit typical characteristics of defective canary changes and non-change faults. As a result, these faults cannot be effectively handled within stage ① and ②.

Li *et al.* [22] conducted a study on 354 publicly available postmortems from three large-scale cloud platforms, revealing that over 46% of the incidents were caused by software changes. Motivated by this finding and corroborated by SREs, it is reasonable to prioritize investigating whether an incident is attributed to a defective change when the fault type cannot be definitively determined. Consequently, drawing inspiration from Enlightenment (2), (3), and (4), unlike in stage ①, where only canary changes of $\mathcal{S}_{rca}$ are considered, Suspicious Change Scorer module assumes that the most recent changes for all services in the dependency graph of $\mathcal{S}_{rca}$ are potentially defective changes. In stage ③, Suspicious Change Scorer takes these changes as input and identifies the change with the highest suspicion level.

The key idea behind Suspicious Change Scorer is to amalgamate valuable clues from abnormal KPIs of service instances, change time, and dependency graph to perform RCCA. *ChangeRCA* does not solely rely on KPIs, as some defective changes may not manifest themselves as KPI anomalies (e.g., service M in Fig. 8). Fortunately, SREs discovered that the system failed merely 5 minutes after the change in service M. Moreover, by that time, more than 5 hours had passed since the last change in services L and N. As a result, the SREs deduced that this fault was associated with the change of M. Consequently, M should be prioritized for inspection. This case exemplifies the contribution of time differences to RCCA. To incorporate these three pieces of evidence, Suspicious Changes Scorer comprises three components: *KPI Scorer*, *Dependency Scorer*, and *Time Scorer*.

5.4.1 KPI Scorer. KPI Scorer evaluates the relationship between KPIs and defective changes. For canary changes, the two-layer DiD method used in the Canary Identifier module is employed to obtain the number of abnormal instances N_{apost}. For completed changes, the second-layer DiD method in the Canary Identifier module is utilized to determine the number of all abnormal

instances $N_{\mathcal{S}a}$. A service with a higher number of abnormal instances will receive a higher score. The KPI score of service $\mathcal{S}$ can be computed as follows,

$$Score_K = \frac{N_{\mathcal{S}a}}{N_{\mathcal{S}}}, \tag{4}$$

where $N_{\mathcal{S}}$ represents the number of service instances of $\mathcal{S}$. For canary changes, $N_{\mathcal{S}a} = N_{apost}$.

5.4.2 Dependency Scorer. As discussed in § 2.3, services dependent on $\mathcal{S}_{rca}$ are more likely to be the defective services. The purpose of Dependency Scorer is to evaluate the proximity of a service to $\mathcal{S}_{rca}$ in terms of their dependency relationship. Dependency Scorer assigns scores to services based on their level of connection to $\mathcal{S}_{rca}$. Services that are directly connected to $\mathcal{S}_{rca}$ receive higher scores, while those that are further away receive lower scores.

To calculate the dependency score for a service $\mathcal{S}$, Dependency Scorer considers the transitive closure of the dependency graph. The transitive closure captures all the direct and indirect dependencies between services. The score assigned to service $\mathcal{S}$ is computed as follows,

$$Score_D = \frac{tier_{max}}{tier_{max} + \text{Tier}(\mathcal{S}, \mathcal{S}_{rca})}, \tag{5}$$

where $tier_{max}$ is the maximum tier in the dependency graph of $\mathcal{S}rca$, and $\text{Tier}(\mathcal{S}, \mathcal{S}rca)$ is the tier from $\mathcal{S}$ to $\mathcal{S}_{rca}$. If $\mathcal{S}$ is $\mathcal{S}_{rca}$, $\text{Tier}(\mathcal{S}, \mathcal{S}_{rca}) = 0$. If $\text{Tier}(\mathcal{S}, \mathcal{S}_{rca}) > \text{tier}_{max}$, $\mathcal{S}$ would not be considered. tier_{max} is set to 2 based on the observations in Fig. 3 that 98% of services with defective changes are located within a two-tier range of calls from $\mathcal{S}_{rca}$ and confirmed by SREs.

5.4.3 Time Scorer. Time Scorer takes into account the timing of changes to evaluate their potential impact on a fault. It considers the time duration between the changes in different services and the occurrence of the incident. The intuition is that if a fault occurs shortly after a change in a specific service, that change is more likely to be the root cause of the incident. Conversely, if a considerable amount of time has passed since the last change in a service, it is less likely to be responsible for the fault. This intuition is inspired by previous work [24] and has been confirmed in most cases by SREs, aligning with industry practices. Considering the change time is important to identify silent changes.

To incorporate time differences into the scoring process, Time Scorer defines time windows for analysis. These time windows determine the duration within which changes are considered relevant for RCCA. When detecting an anomaly at time $time_a$, and a service $\mathcal{S}$ that releases its latest change at $time_c$, the time difference $time_a - time_c$ is evaluated to determine if it falls within the defined time window. If the time difference is within the time window, it is considered to have a positive effect on the RCCA. The time score of service $\mathcal{S}$ can be computed as follows:

$$Score_T = \frac{\sum_m \text{TimeScorer}\left(\text{time}_a, \text{time}_c \mid TW_j\right)}{m}, \tag{6}$$

where m denotes the number of time windows. Each time window is defined as $TW_j = TW_{base} \times 2^j$, where TW_{base} is the base duration for the time windows. The TimeScorer function determines whether the time difference $\text{time}_a - \text{time}_c$ falls within the jth time window (TW_j). If the time difference is less than TW_j, TimeScorer is set to 1, indicating that the change occurred within the relevant time frame. Otherwise, if $\text{time}_a - \text{time}_c \geq TW_j$, TimeScorer is set to 0, indicating that the change occurred outside the relevant time frame. The choice of m and TW_{base} for the time windows depends on the specific characteristics of the system being analyzed. In this study, m is set to 8 and TW_{base} is set to 30 minutes by default based on the change policy defined in WeChat.

After obtaining the individual scores $Score_{iK}$, $Score_{iD}$, and $Score_{iT}$ for change i using the KPI Scorer, Dependency Scorer, and Time Scorer, respectively, Suspicious Changes Scorer module employs a hierarchical weighted average approach to combine these scores and derive the final

suspicious score. If change i is a canary change and its $Score_{iK}$ exceeds the threshold value η, it is inferred to be a defective change based on Canary Identifier. In this case, the KPI score alone is considered sufficient evidence to infer faults, and the change is assigned a higher priority. For other cases, the weighted average approach defined in Equ. 7 is applied to calculate the final score.

$$Score_i = \begin{cases} 3 + Score_{iK} & if\ Score_{iK} > \eta \text{ and } i \text{ is canary change} \\ \alpha Score_{iK} + \beta Score_{iD} + \gamma Score_{iT} & Otherwise \end{cases} \tag{7}$$

The weights α, β, and γ represent the importance given to the KPI Scorer, Dependency Scorer, and Time Scorer, respectively. In this study, we set α, β, and γ equal to 1 in Equ. 7 because SREs believe that each scoring component contributes equally to the final suspicious score calculation.

Finally, after computing the final scores for all changes, an indicator $\mathcal{T} = 2$ and an ordered score list $Score = [score_1, ..., score_M]$ is generated, where M represents the total number of changes. The change with the highest final score is considered the most likely to be the defective one, indicating a higher priority for inspection and localization by the SREs.

6 EXPERIMENT EVALUATION

In this section, we aim to evaluate *ChangeRCA* to answer the following research questions (RQs):

- **RQ4:** How effective is *ChangeRCA* in RCCA? (§ 6.2)
- **RQ5:** How useful are the dependency graph and different rankers for RCCA? (§ 6.3)
- **RQ6:** Can *ChangeRCA* reduce the identification time of change-inducing incidents? (§ 6.4)
- **RQ7:** How do RCA approaches affect the effectiveness of *ChangeRCA*? (§ 6.5)
- **RQ8:** How do parameters affect the effectiveness of *ChangeRCA*? (§ 6.6)

6.1 Experiment Setup

6.1.1 Datasets. To evaluate the effectiveness of *ChangeRCA*, we utilize two datasets from real-world production systems from WeChat and a microservice benchmark system OnlineBoutique [15]. These datasets comprise a total of 81 incidents, encompassing various scenarios.

Real-world System Dataset $\mathcal{A}$: We collect change-inducing incidents from the production environment of WeChat for 3 months. With the assistance of SRE teams, we construct a labeled dataset consisting of 30 change-inducing incidents. Further details regarding data collection and change types can be found in § 3.1. Due to the limitations on the persistence time of KPI data in WeChat, we are unable to collect more change-inducing incidents

Benchmark System Dataset $\mathcal{B}$. We deploy an open-source microservice benchmark OnlineBoutique (OB) on a Kubernetes platform with 12 nodes. OB simulates an e-commerce microservice system where users can browse, view, and purchase products. OB have been widely used in many previous studies [17, 23, 52, 55]. Based on previous studies [9, 65], we carefully designed and simulated 12 types of defective changes in the OB. Table 2 provides an overview of the 12 fault types of defective changes we designed. Additionally, we also incorporate normal changes into the dataset to ensure its diversity. For each case, we randomly select one defective change and 2 to 5 normal changes on different services, resulting in a varied and comprehensive dataset. $\mathcal{B}$ consists of 51 incidents in total. The dataset $\mathcal{B}$ is available at [6].

Each case in the dataset includes the necessary input for *ChangeRCA*, such as the RCA service, dependency graph, change flow, and KPI data. These data can be easily obtained from a production environment. For example, obtaining the dependency graph can be achieved through service meshes (e.g., Istio [19]) or programming frameworks (e.g., Spring Boot [37]). Change flow information is typically stored in CI/CD tools (e.g., Gitlab [14]). Regarding the KPIs, we primarily focus on the request success ratio, which reflects the proportion of successfully processed requests. Request

Table 2. Fault type of defective changes injected on the benchmark OnlineBoutique for evaluation.

Change Type	Fault Type
Defective Backend Change	Miss Func. Call, Wrong Except. Handle, Miss Param. Value, Wrong Return Value, Wrong Sql, Wrong Cache, Wrong Param. Order
Defective Config. Change	Wrong Port, Config. File Inconsistent, Wrong Access Key
Defective Resource Change	Memory Leak, Unsatisfactory Resource Allocation

latency is intentionally excluded from consideration as excessive latency can lead to request timeouts, which would also be reflected in a decrease in the success ratio.

6.1.2 Evaluation Measurements. For RQ4, RQ5 and RQ7, we apply two widely used measurements [36, 52, 66] to validate the performance of *ChanegRCA* and ACD approaches.

- **Hit Rate of Top-k (*HR@k*)** refers to the probability that root cause change is within the top-k results. $HR@k = \frac{1}{N}\sum_{i=1}^{N}\left(c_i \in \text{Score}^i_{[1:k]}\right)$, where c_i is the ground-truth root cause change for the i-th incident, $C^i_{[1:k]}$ is top-k list of the i-th incident, and N is the number of all incidents. We set $k = \{1, 3, 5\}$ because a survey [20] found that more than 73% operators only consider Top-5 ranked elements. The higher $HR@k$ is better.
- **Exam Score (*ES*)** refers to the average count of false-positive changes that have to be excluded manually by SREs before locating the actual defective change. If the root cause is outside the Top-5, we assign a default false-positive candidate count of 10 for it. The smaller *ES* is better.

For RQ6, we introduce **Time To Identify (TTI)** to evaluate the identification time reduction brought by *ChangeRCA*. TTI is the time SREs require to identify the root causes after detecting a fault [22]. We analyzed historical incidents and found that the time for SREs to inspect a change ticket manually is approximately 2 minutes. If the root cause change is located at the Top-1 position, the TTI is equal to the execution time of approaches. Otherwise, the TTI is the sum of the execution time of approaches and review time of false-positive changes.

6.1.3 Implementation and Settings. We implement a prototype of *ChangeRCA* using Python 3.6, with the source code available at [6]. Our experiments were conducted on a virtual machine equipped with a 32-core AMD EPYC 7K62 Processor (2.6 GHz) and 32 GB memory, running Tencent Linux 3.2. We set the p-value λ to 0.05 and the score threshold η to 0.8 by default. The default RCA approach is GIED [17] because it has been proven effective in industrial systems. The influence of RCA, λ and η will discuss in § 6.5 and § 6.6.

6.2 RQ4: Effectiveness Result in RCCA

Table 3 presents the comparative results between *ChangeRCA* and ACD approaches on $\mathcal{A}$ and $\mathcal{B}$. From Table 3, we can observe that *ChangeRCA* outperforms all the ACD approaches significantly in both two dataset and achieves high accuracy in *HR*@1 (85.78%), *HR*@3 (96%) and *HR*@5 (96.67%) on average. These results indicate that *ChangeRCA* is able to successfully identify the correct root cause change in the majority of cases. The excellent performance of *ChangeRCA* is mainly attributed to the fact that *ChangeRCA* considers the interdependencies among changes across various services.

In contrast, the ACD approaches achieve lower hit rate. FUNNEL, SCWarn, and Gandalf achieve 57.75%, 65.39%, 65.69% *HR*@1 on average. *ChangeRCA* surpasses ACD approaches by 20% to 28% in *HR*@1, demonstrating its ability to accurately identify defective changes. Furthermore, the *ES* of *ChangeRCA* is 1.48, while the *ES* for FUNNEL, SCWarn, and Gandalf are 4.33, 3.95, 3.88, reflecting a

Table 3. Root cause change analysis results on Dataset $\mathcal{A}$ and $\mathcal{B}$.

Data	Approach	HR@1	⇑HR@1(%)	HR@3	⇑HR@3(%)	HR@5	⇑HR@5(%)	ES	⇓ES(%)
$\mathcal{A}$	**ChangeRCA**	**83.33**	-	**90.00**	-	**93.33**	-	**1.83**	-
	FUNNEL [60]	56.67	**47.04**	73.33	**22.73**	76.67	**21.72**	3.96	**53.78**
	SCWarn [63]	70.00	19.04	80.00	12.5	80.00	16.66	3.43	46.64
	Gandalf [24]	66.67	24.98	76.67	17.38	80.00	16.66	3.6	49.16
	w/o graph	56.67	47.04	73.33	22.73	76.67	21.72	3.96	53.78
	w/o $Score_K$	73.33	13.64	80.00	12.5	90.00	3.7	2.63	30.42
	w/o $Score_D$	70.00	19.04	80.00	12.5	80.00	16.66	3.43	46.65
	w/o $Score_T$	60.00	38.89	83.33	8.0	86.67	7.68	2.73	32.97
$\mathcal{B}$	**ChangeRCA**	**88.23**	-	**100.0**	-	**100.0**	-	**1.13**	-
	FUNNEL [60]	58.82	50	58.82	**70**	58.82	**70**	4.70	75.06
	SCWarn [63]	60.78	45.16	60.78	64.53	62.74	59.39	4.47	67.06
	Gandalf [24]	64.70	36.67	64.70	54.56	72.54	37.85	4.16	62.95
	w/o graph	37.25	**136.86**	58.82	70	58.82	70	4.92	**77.03**
	w/o $Score_K$	72.54	21.63	98.03	2.04	100.0	0	1.39	18.71
	w/o $Score_D$	80.39	9.75	98.03	2.04	100.0	0	1.27	11.02
	w/o $Score_T$	82.35	7.14	96.07	4.09	100.0	0	1.31	13.74

reduction of 62% to 65%. These results illustrate that SREs need to examine fewer false positive changes with *ChangeRCA*.

We also investigated the reasons behind the suboptimal performance of the ACD approaches. FUNNEL and SCWarn fail to account for valuable information from service dependency and change flow, rendering them incapable of handling change-inducing incidents caused by fault propagation of silent changes. Although Gandalf considers change flow, it is designed to identify the defective change in the canary change phase, making it challenging to identify the defective change after the change is complete. In contrast, *ChangeRCA* leverages useful information from the abnormal KPI of service instances, change time, and dependency graph to perform RCCA. This enables *ChangeRCA* to handle a wider range of defective changes.

In summary, *ChangeRCA* is very effective in RCCA, achieving 20% to 28% improvement in *HR*@1 and reducing 62% to 65% in *ES* compared to ACD approaches.

6.3 RQ5: Ablation Result in RCCA

We conducte an ablation study to explore the contributions of the dependency graph and different scorers in *ChangeRCA*. We derived several variants of *ChangeRCA* to analyze their impact,

- w/o graph: *ChangeRCA* does not uses dependency graph and only uses Canary Identifier § 5.2.
- w/o $Score_K$: *ChangeRCA* removes the KPI Scorer module in § 5.4.1.
- w/o $Score_D$: *ChangeRCA* removes the Dependency Scorer module in § 5.4.2.
- w/o $Score_T$: *ChangeRCA* removes the Time Scorer module in § 5.4.3.

The ablation study results are displayed in Table 3. It is evident that the dependency graph and each scorer contributes to the effectiveness of *ChangeRCA* as *ChangeRCA* achieves the best performance. However, the extent of their contributions varies. Specifically, *KPI Scorer* contributes the least, as *ChangeRCA* w/o $Score_K$ ranks second-best. This is because most defective changes that cause KPI changes are already captured by Canary Identifier in stage ①. Additionally, we observe a significant performance degradation of *ChangeRCA* w/o graph, particularly in *HR*@1, since dependency graph is crucial information that greatly contributes to the identification of defective changes that do not manifest their own anomalies.

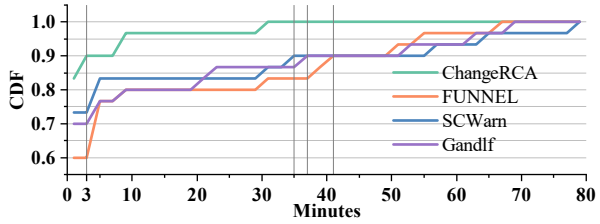


Fig. 9. CDF of TTI in change-inducing incidents.

Table 4. Results of *ChangeRCA* triggered by different RCA approaches.

RCA Approach	HR@1(%)	HR@3(%)	HR@5(%)	ES
GIED [17]	**88.23**	**100.0**	**100.0**	**1.13**
Microscope [26]	74.50	96.07	98.03	1.49
MicroRCA [47]	88.23	96.07	98.03	1.37

6.4 RQ6: Time to Identification Result in RCCA

Rapid identification of defective change is essential for timely incident mitigation in online systems upon receiving an alert. In this section, we compare the TTI between *ChangeRCA* and ACD approaches to demonstrate that ChangeRCA reduces the time required to identify defective changes. Figure 9 displays the CDF of TTI for *ChangeRCA* and ACD approaches in $\mathcal{A}$. As illustrated in Fig. 9, *ChangeRCA* can locate 90% of defective changes in less than 3 minutes, compared to 35, 37, and 41 minutes for SCWarn, Gandalf, and FUNNEL, resulting in a more than 90% reduction. This result indicates that SREs can localize defective changes more rapidly with the assistance of *ChangeRCA*, thereby accelerating incident mitigation. Due to page limitations, we do not show the results on the $\mathcal{B}$ dataset, but the results are consistent across the two datasets.

6.5 RQ7: Impact of RCA on *ChangeRCA*

We evaluate the impact of different RCA approaches on the performance of *ChangeRCA* by using the results of three state-of-the-art RCA approaches as input. The following three RCA approaches are considered,

- **GIED** [17] adopts a graph neural network model and a PageRank algorithm with a flexible transition matrix to localize suspicious services.
- **Microscope** [26] is an RCA approach that identifies root cause services based on the correlation of metrics from frontend to downstream services along the dependency graph.
- **MicroRCA** [47] presents an RCA approach that localizes suspicious services by leveraging a PageRank method on an extracted anomaly sub-graph.

Table 4 displays the results of *ChangeRCA* triggered by the outputs of different RCA approaches on dataset $\mathcal{B}$. We do not use dataset $\mathcal{A}$ here because Microscope and MicroRCA validated their effectiveness in simulated benchmarks rather than real industry scenarios. It is not practical to directly apply methods that work well in small-scale systems to complex real-world scenarios.

From Table 4, we observe that different RCA algorithms have a substantial effect on *HR*@1, where the GIED result is 13.73% higher than the Microscope result. However, the impact on *HR*@3 and *HR*@5 is relatively small. This discrepancy arises because GIED exhibits higher accuracy than Microscope in localizing root causes at the service level. Despite the less accurate localization provided by Microscope, *ChangeRCA* is still able to rank anomalous changes within the top three in most cases by considering the service dependency graph. This demonstrates that *ChangeRCA* can

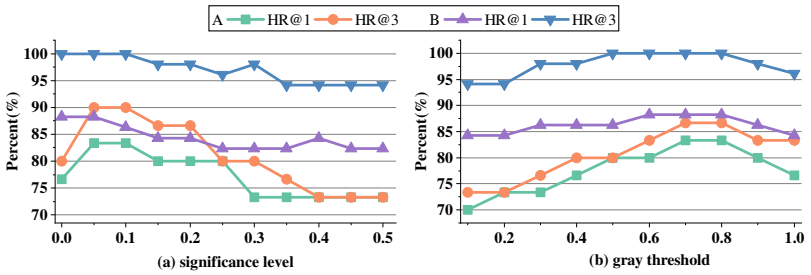


Fig. 10. The effect of parameters in *ChangeRCA*

compensate for the limitations of the RCA approaches and effectively identify root cause changes by leveraging the additional information from the dependency graph.

6.6 RQ8: Parameter Sensitivity

The significant level λ and score threshold η are two important parameters in *ChangeRCA*. Figure 10 (a) presents its effect the value of λ on *HR*@1 and *HR*@3. In hypothesis test, λ usually can take a value between 0.01 and 0.2. Bigger λ usually leads to lower *HR*@1 and *HR*@3, as it is easier to cause false positives with more loose conditions. According to the results, $\lambda = 0.05$ achieves the best result in terms of both *HR*@1 and *HR*@3. Figure 10 (b) presents the effect of the value of η in on *HR*@1 and *HR*@3. Clearly, with small η, the threshold is relatively loose, leading to lower *HR*@1 and *HR*@3. This is because *ChangeRCA* will generate some false positive changes caused by noises. According to the Fig. 10 (b), $\eta = 0.7$ or $\eta = 0.8$ achieves the best results of *HR*@1 and *HR*@3. In practice, the best configuration of η and λ highly depends on the characteristics of the dataset.

7 DISCUSSION

7.1 Limitations and Future Work

ChangeRCA relies on the comparison between pre- and post-change instances. However, when releasing a new service, it has no pre-change instances. When removing a service, it has no post-change instances. Consequently, *ChangeRCA* struggles to accurately identify both types of defective changes. In the future, we need to optimize *ChangeRCA* for these two special changes.

Furthermore, an important limitation of the current *ChangeRCA* implementation is the lack of user feedback incorporation. This absence restricts *ChangeRCA*'s ability to precisely pinpoint defective changes that are solely perceived by users. To overcome this limitation, we intend to explore the inclusion of user feedback results in *ChangeRCA* as part of our future work.

7.2 Threats to Validity

The threats to the internal validity mainly lie in the implementation of ACD and RCA approaches. To mitigate threats related to the implementation of ACD and RCA approaches, we have taken several steps. For Funnel, Gandalf, and Microscope, as these approaches lack publicly available implementations, we have implemented them ourselves based on the details provided in their respective papers. This approach ensures that our implementation aligns with the original designs and methodologies outlined in the literature. Conversely, for SCWarn, MicroRCA, and GIED, we have directly utilized their open-source code, which enhances the credibility and reliability of our study by leveraging established and validated implementations.

Threats to external validity primarily pertain to the generalizability of the study findings to a wider population. The study subjects in evaluations consist of one real-world system and one microservice benchmark. While these subjects may not fully represent all online service systems,

we believe our approach is general enough for two reasons: (1) WeChat encompasses large online systems that host various types of services, such as instant messaging, shopping, payment, and video. By considering a diverse range of services. (2) The defective changes introduced in WeChat are derived from real cases observed in the production environment. These real-world defective changes reflect the complexity and diversity of fault scenarios that can occur in production systems. Moreover, to mitigate the potential limitations of fault scenarios in the microservice benchmark, we have injected defective changes from OpenStack [9]. By incorporating these real cases, our evaluation captures representative fault scenarios that enhance the generalizability of the results.

8 RELATED WORK

Change-inducing incident study. Considerable efforts have been devoted to analyzing change-inducing incidents [3, 10, 12, 22, 34, 38, 62]. Zhang *et al.* [62] conduct an in-depth study of 123 real-world change incidents, shedding lights on the severity, underlying causes, exposing conditions, and resolution strategies of change incidents. They observe that only 37% of the software change bugs were caught before corresponding versions were released to public, with the majority (63%) exposed in production. Tudor *et al.* [12] analyze 55 upgrade failures from three systems to build an upgrade-centric fault model, which concentrates on the effects of upgrade procedural mistakes rather than software defects. Li *et al.* [22] studied 354 publicly available post-mortems collected in three large-scale cloud. One of their findings is that 46.6% of incidents involve software changes. An *et al.* [2] examine the characteristics of commits that lead to crashes in Mozilla Firefox. They found that crash-inducing commits are often submitted by developers with less experience. However, current online systems commonly experience partial failures rather than crash [27].

Identifying defective changes before deployment. Many prior work on identifying defective changes by software test [8, 21, 45, 58, 59], reliability auditing [4, 49, 57], and system verification [5, 32, 62]. Rex [32] utilizes a combination of machine learning and program analysis to learn change-rules that capture correlations of bugs and misconfigurations. PCHECK [49] analyzes the source code and automatically generates configuration checking code (called checkers) to detect latent configuration (LC) errors. Such software misconfiguration detection tools focused on configuration logic, rather than failures caused by common dependencies. DUPChecker [62] is a static tool used to search for incompatibility on data of enum types and data defined using serialization libraries. Raghav *et al.* [4] proactively assess the risk associated with software changes by extracting linkages between changes and change-inducing incidents. However, defective changes would remain concealed during the testing phase prior to software changes due to unknown differences between production and development environments.

Identifying defective changes after deployment. Efforts to identify defective changes after deployment are crucial because some bugs and errors can go undetected due to discrepancies between testing and the production environment [24, 29–31, 44, 60, 63]. FUNNEL [60] detects performance changes in the impact set by the singular spectrum transform algorithm, and determines performance changes induced by software changes using a difference-in-difference method. Gandalf [24], Kontrast [44] and CORNET [29] use the idea of the treatment group (where the change was implemented) versus the control group (where the change was not implemented) comparison to check where the current change is defective. SCWarn [63] draws support from multi-modal learning to identify abnormal changes from heterogeneous multi-source data. However, existing defective software change identification approaches majorly regard this problem as an anomaly detection task, utilizing anomaly detection algorithms to apply to this problem directly. In addition, there are some crash-inducing changes localization approaches [46, 48, 64] that rely on crash reports to localize defective change. However, they are inadequate for handling partial failures that

do not result in a system crash or do not generate crash reports. Hence, an RRCA approach tailored to handle post-deployment change failures is essential for online systems.

9 CONCLUSION

In this study, we propose a novel concept called RCCA to identify the root cause changes of change-inducing incidents from massive changes across different services. To make RCCA applicable in practical scenarios, we propose an intelligent RCCA approach named *ChangeRCA*. The primary objective of *ChangeRCA* is to accurately localize the defective change among numerous changes associated with change-inducing incidents, thereby expediting the incident resolution process. We have conducted an extensive evaluation of *ChangeRCA* on both a real-world dataset from WeChat and a simulated dataset encompassing diverse defective changes. Our evaluation results demonstrate that *ChangeRCA* significantly outperforms all existing methods, achieving a *HR*@3 of 96% and effectively reducing the time required for identifying defective changes.

Data Availability. The implementation of *ChangeRCA* and OB data are available at [6].

ACKNOWLEDGMENTS

We greatly appreciate the insightful feedback from the anonymous reviewers. The research is supported by the National Natural Science Foundation of China (No.62272495) and the Guangdong Basic and Applied Basic Research Foundation (No.2023B1515020054), and sponsored by Tencent Rhino-Bird Research Elite Program. The corresponding author is Pengfei Chen.

REFERENCES

[1] Amazon. 2017. Summary of the Amazon S3 Service Disruption in the Northern Virginia (US-EAST-1) Region. https://aws.amazon.com/message/41926/. Accessed February 6, 2023.
[2] Le An and Foutse Khomh. 2015. An Empirical Study of Crash-inducing Commits in Mozilla Firefox. In *PROMISE 2015*. ACM, 5:1–5:10. https://doi.org/10.1145/2810146.2810152
[3] Ibrahim Assi, Rami Tailakh, and Abdel Salam Sayyad. 2021. Survey on software changes: reasons and remedies. *International Arab Journal of Information Technology* 18, 2 (2021), 248–260. https://doi.org/10.34028/iajit/18/2/14
[4] Raghav Batta, Larisa Shwartz, Michael Nidd, Amar Prakash Azad, and Harshit Kumar. 2021. A system for proactive risk assessment of application changes in cloud operations. In *CLOUD 2021*. IEEE, 112–123. https://doi.org/10.1109/CLOUD53861.2021.00025
[5] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. 2017. A General Approach to Network Configuration Verification. In *SIGCOMM 2017*. ACM, 155–168. https://doi.org/10.1145/3098822.3098834
[6] ChangeRCA. 2024. ChangeRCA. https://github.com/IntelligentDDS/ChangeRCA. Accessed Feb. 6, 2024.
[7] C. Chatfield. 1978. The Holt-Winters Forecasting Procedure. *Journal of the Royal Statistical Society. Series C (Applied Statistics)* 27, 3 (1978), 264–279. https://doi.org/10.2307/2347162
[8] Runxiang Cheng, Lingming Zhang, Darko Marinov, and Tianyin Xu. 2021. Test-case prioritization for configuration testing. In *ISSTA 2021*. ACM, 452–465. https://doi.org/10.1145/3460319.3464810
[9] Domenico Cotroneo, Luigi De Simone, Pietro Liguori, Roberto Natella, and Nematollah Bidokhti. 2019. How Bad Can a Bug Get? An Empirical Analysis of Software Failures in the OpenStack Cloud Computing Platform. In *ESEC/FSE 2019*. ACM, 200–211. https://doi.org/10.1145/3338906.3338916
[10] Jose Luis de la Vara, Markus Borg, Krzysztof Wnuk, and Leon Moonen. 2016. An Industrial Survey of Safety Evidence Change Impact Analysis Practice. *IEEE TSE* 42, 12 (2016), 1095–1117. https://doi.org/10.1109/TSE.2016.2553032
[11] Dropbox. 2014. Dropbox change failure. https://dropbox.tech/infrastructure/outage-post-mortem. Accessed February 6, 2023.
[12] Tudor Dumitraș and Priya Narasimhan. 2009. Why Do Upgrades Fail and What Can We Do about It? Toward Dependable, Online Upgrades in Enterprise System. In *Middleware 2009*. Springer-Verlag, 20 pages. https://doi.org/10.5555/1656980.1657005
[13] Supriyo Ghosh, Manish Shetty, Chetan Bansal, and Suman Nath. 2022. How to fight production incidents?: an empirical study on a large-scale cloud service. In *SoCC 2022*. ACM, 126–141. https://doi.org/10.1145/3542929.3563482
[14] Gitlab. 2023. Gitlab. https://gitlab.com. Accessed Sep. 6, 2023.
[15] GoogleCloudPlatform. 2023. OnlineBoutique. https://github.com/GoogleCloudPlatform/microservices-demo. Accessed Sep.6, 2023.

[16] Lynn Greiner. 2020. The great 2020 Gmail outage: A tale of two blackouts, and lessons learned. https://www.itworldcanada.com/article/the-great-2020-gmail-outage-a-tale-of-two-blackouts-and-lessons-learned/439924. Accessed February 6, 2023.
[17] Zilong He, Pengfei Chen, Yu Luo, Qiuyu Yan, Hongyang Chen, Guangba Yu, and Fangyuan Li. 2022. Graph based Incident Extraction and Diagnosis in Large-Scale Online Systems. In *ASE 2022*. ACM, 48:1–48:13. https://doi.org/10.1145/3551349.3556904
[18] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long Short-Term Memory. *Neural Computation* 9, 8 (1997), 1735–1780. https://doi.org/10.1162/neco.1997.9.8.1735
[19] Istio. 2023. Istio. https://istio.io. Accessed Sep. 6, 2023.
[20] Pavneet Singh Kochhar, Xin Xia, David Lo, and Shanping Li. 2016. Practitioners' expectations on automated fault localization. In *ISSTA 2016*. ACM, 165–176. https://doi.org/10.1145/3510003.3510152
[21] Yigit Küçük, Tim A. D. Henderson, and Andy Podgurski. 2021. Improving Fault Localization by Integrating Value and Predicate Based Causal Inference Techniques. In *ICSE 2021*. IEEE, 649–660. https://doi.org/10.1109/ICSE43902.2021.00066
[22] Xiaoyun Li, Guangba Yu, Pengfei Chen, Hongyang Chen, and Zhekang Chen. 2022. Going through the Life Cycle of Faults in Clouds: Guidelines on Fault Handling. In *ISSRE 2022*. IEEE, 121–132. https://doi.org/10.1109/ISSRE55969.2022.00022
[23] Yufeng Li, Guangba Yu, Pengfei Chen, Chuanfu Zhang, and Zibin Zheng. 2022. MicroSketch: Lightweight and Adaptive Sketch Based Performance Issue Detection and Localization in Microservice Systems. In *ICSOC 2022 (Lecture Notes in Computer Science, Vol. 13740)*. Springer, 219–236. https://doi.org/10.1007/978-3-031-20984-0_15
[24] Ze Li, Qian Cheng, Ken Hsieh, Yingnong Dang, Peng Huang, Pankaj Singh, Xinsheng Yang, Qingwei Lin, Youjiang Wu, Sebastien Levy, and Murali Chintalapati. 2020. Gandalf: An Intelligent, End-To-End Analytics Service for Safe Deployment in Large-Scale Cloud Infrastructure. In *NSDI 2020*. USENIX Association, 389–402. https://www.usenix.org/conference/nsdi20/presentation/li
[25] Zeyan Li, Nengwen Zhao, Mingjie Li, Xianglin Lu, Lixin Wang, Dongdong Chang, Xiaohui Nie, Li Cao, Wenchi Zhang, Kaixin Sui, Yanhua Wang, Xu Du, Guoqiang Duan, and Dan Pei. 2022. Actionable and Interpretable Fault Localization for Recurring Failures in Online Service Systems. In *ESEC/FSE 2022*. ACM, 996–1008. https://doi.org/10.1145/3540250.3549092
[26] Jinjin Lin, Pengfei Chen, and Zibin Zheng. 2018. Microscope: Pinpoint Performance Issues with Causal Graphs in Micro-service Environments. In *ICSOC 2018*. Springer, 3–20. https://doi.org/10.1007/978-3-030-03596-9_1
[27] Chang Lou, Peng Huang, and Scott Smith. 2020. Understanding, Detecting and Localizing Partial Failures in Large System Software. In *NSDI 2020*. USENIX Association, 559–574. https://www.usenix.org/conference/nsdi20/presentation/lou
[28] Kim Lyons. 2021. Facebook says 'configuration change' caused some users to be logged out unexpectedly. https://www.theverge.com/2021/1/23/22245842/facebook-logged-out-configuration-change-ios-app-security. Accessed February 6, 2024.
[29] Ajay Mahimkar, Carlos Eduardo de Andrade, Rakesh Sinha, and Giritharan Rana. 2021. A Composition Framework for Change Management. In *SIGCOMM 2021*. ACM, 788–806. https://doi.org/10.1145/3452296.3472901
[30] Ajay Mahimkar, Zihui Ge, Jia Wang, Jennifer Yates, Yin Zhang, Joanne Emmons, Brian Huntley, and Mark Stockert. 2011. Rapid detection of maintenance induced changes in service performance. In *Co-NEXT 2011*. ACM, 13. https://doi.org/10.1145/2079296.2079309
[31] Ajay Anil Mahimkar, Han Hee Song, Zihui Ge, Aman Shaikh, Jia Wang, Jennifer Yates, Yin Zhang, and Joanne Emmons. 2010. Detecting the performance impact of upgrades in large operational networks. In *SIGCOMM 2010*. ACM, 303–314. https://doi.org/10.1145/1851182.1851219
[32] Sonu Mehta, Ranjita Bhagwan, Rahul Kumar, Chetan Bansal, Chandra Shekhar Maddila, Balasubramanyan Ashok, Sumit Asthana, Christian Bird, and Aditya Kumar. 2022. Rex: Preventing Bugs and Misconfiguration in Large Services Using Correlated Change Analysis. In *NSDI 2020*. USENIX Association, 435–448. https://www.usenix.org/conference/nsdi20/presentation/mehta
[33] Nabor C. Mendonca, Pooyan Jamshidi, David Garlan, and Claus Pahl. 2021. Developing Self-Adaptive Microservice Systems: Challenges and Directions. *IEEE Software* 38, 2 (2021), 70–79. https://doi.org/10.1109/MS.2019.2955937
[34] Amit Kumar Mondal, Kevin A. Schneider, Banani Roy, and Chanchal K. Roy. 2022. A survey of software architectural change detection and categorization techniques. *Elsevier JSS* 194 (2022), 111505. https://doi.org/10.1016/j.jss.2022.111505
[35] Yicheng Pan, Meng Ma, Xinrui Jiang, and Ping Wang. 2021. Faster, deeper, easier: crowdsourcing diagnosis of microservice kernel failure from user space. In *ISSTA 2021*. ACM, 646–657. https://doi.org/10.1145/3460319.3464805
[36] Spencer Pearson, José Campos, René Just, Gordon Fraser, Rui Abreu, Michael D. Ernst, Deric Pang, and Benjamin Keller. 2017. Evaluating and improving fault localization. In *ICSE 2017*. IEEE / ACM, 609–620. https://doi.org/10.1109/ICSE.2017.62

[37] Pivotal. 2023. Spring Boot. https://spring.io. Accessed Sep. 6, 2023.
[38] Tony Savor, Mitchell Douglas, Michael Gentili, Laurie A. Williams, Kent L. Beck, and Michael Stumm. 2016. Continuous deployment at Facebook and OANDA. In *ICSE Companion 2016*. ACM, 21–30. https://doi.org/10.1145/2889160.2889223
[39] Toby Sterling. 2021. Fastly blames software bug for major global internet outage. https://www.reuters.com/business/media-telecom/fastly-blames-software-bug-major-global-internet-outage-2021-06-09/. Accessed February 6, 2023.
[40] Elizabeth A Stuart, Haiden A Huskamp, Kenneth Duckworth, Jeffrey Simmons, Zirui Song, Michael E Chernew, and Colleen L Barry. 2014. Using propensity scores in difference-in-differences models to estimate the effects of a policy change. *Health Services and Outcomes Research Methodology* 14 (2014), 166–182. https://doi.org/10.1007/s10742-014-0123-z
[41] Anthony J Viera, Joanne M Garrett, et al. 2005. Understanding interobserver agreement: the kappa statistic. *Fam med* 37, 5 (2005), 360–363. https://pubmed.ncbi.nlm.nih.gov/15883903/
[42] Hanzhang Wang, Zhengkai Wu, Huai Jiang, Yichao Huang, Jiamu Wang, Selcuk Kopru, and Tao Xie. 2021. Groot: An event-graph-based approach for root cause analysis in industrial settings. In *ASE 2021*. IEEE, 419–429. https://doi.org/10.1109/ASE51524.2021.9678708
[43] Lu Wang, Pu Zhao, Chao Du, Chuan Luo, Mengna Su, Fangkai Yang, Yudong Liu, Qingwei Lin, Min Wang, Yingnong Dang, Hongyu Zhang, Saravan Rajmohan, and Dongmei Zhang. 2022. NENYA: Cascade Reinforcement Learning for Cost-Aware Failure Mitigation at Microsoft 365. In *KDD 2022*. ACM, 4032–4040. https://doi.org/10.1145/3534678.3539127
[44] Xuanrun Wang, Kanglin Yin, Qianyu Ouyang, Xidao Wen, Shenglin Zhang, Wenchi Zhang, Li Cao, Jiuxue Han, Xing Jin, and Dan Pei. 2022. Identifying Erroneous Software Changes through Self-Supervised Contrastiv Learning on Time Series Data. In *ISSRE 2022*. IEEE, 366–377. https://doi.org/10.1109/ISSRE55969.2022.00043
[45] Ming Wen, Junjie Chen, Yongqiang Tian, Rongxin Wu, Dan Hao, Shi Han, and Shing-Chi Cheung. 2021. Historical Spectrum Based Fault Localization. *IEEE TSE* 47, 11 (2021), 2348–2368. https://doi.org/10.1109/TSE.2019.2948158
[46] Ming Wen, Rongxin Wu, and Shing-Chi Cheung. 2016. Locus: locating bugs from software changes. In *ASE*. ACM, 262–273. https://doi.org/10.1145/2970276.2970359
[47] Li Wu, Johan Tordsson, Erik Elmroth, and Odej Kao. 2020. MicroRCA: Root Cause Localization of Performance Issues in Microservices. In *NOMS 2020*. IEEE/IFIP, 1–9. https://doi.org/10.1109/NOMS47738.2020.9110353
[48] Rongxin Wu, Ming Wen, Shing-Chi Cheung, and Hongyu Zhang. 2018. ChangeLocator: locate crash-inducing changes based on crash reports. In *ICSE 2018*. ACM, 536. https://doi.org/10.1145/3180155.3182516
[49] Tianyin Xu, Xinxin Jin, Peng Huang, Yuanyuan Zhou, Shan Lu, Long Jin, and Shankar Pasupathy. 2016. Early Detection of Configuration Errors to Reduce Failure Damage. In *OSDI 2016*. USENIX Association, 619–634. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/xu
[50] Yong Xu, Xu Zhang, Chuan Luo, Si Qin, Rohit Pandey, Chao Du, Qingwei Lin, Yingnong Dang, and Andrew Zhou. 2021. CARE: Infusing Causal Aware Thinking to Root Cause Analysis in Cloud System. In *HAOC 2021*. ACM, 1–3. https://doi.org/10.1145/3447851.3458737
[51] Zihao Ye, Pengfei Chen, and Guangba Yu. 2021. T-Rank: A Lightweight Spectrum based Fault Localization Approach for Microservice Systems. In *CCGrid 2021*. IEEE/ACM, 416–425. https://doi.org/10.1109/CCGrid51090.2021.00051
[52] Guangba Yu, Pengfei Chen, Hongyang Chen, Zijie Guan, Zicheng Huang, Linxiao Jing, Tianjun Weng, Xinmeng Sun, and Xiaoyun Li. 2021. MicroRank: End-to-End Latency Issue Localization with Extended Spectrum Analysis in Microservice Environments. In *WWW 2021*. ACM, 3087–3098. https://doi.org/10.1145/3442381.3449905
[53] Guangba Yu, Pengfei Chen, Pairui Li, Tianjun Weng, Haibing Zheng, and Yuetang Deng. 2023. LogReducer: Identify and Reduce Log Hotspots inKernel on the Fly. In *ICSE 2023*. IEEE, 1763–1775. https://doi.org/10.1109/ICSE48619.2023.00151
[54] Guangba Yu, Pengfei Chen, Yufeng Li, Hongyang Chen, Xiaoyun Li, and Zibin Zheng. 2023. Nezha: Interpretable Fine-Grained Root Causes Analysis for Microservices on Multi-Modal Observability Data. In *ESEC/FSE 2023*. ACM, 553–565. https://doi.org/10.1145/3611643.3616249
[55] Guangba Yu, Pengfei Chen, and Zibin Zheng. 2019. Microscaler: Automatic Scaling for Microservices with an Online Learning Approach. In *ICWS 2019*. IEEE, 68–75. https://doi.org/10.1109/ICWS.2019.00023
[56] Guangba Yu, Zicheng Huang, and Pengfei Chen. 2021. TraceRank: Abnormal service localization with dis-aggregated end-to-end tracing data in cloud native systems. *Journal of Software: Evolution and Process* 35, 10 (2021), e2413. https://doi.org/10.1002/smr.2413
[57] Ennan Zhai, Ang Chen, Ruzica Piskac, Mahesh Balakrishnan, Bingchuan Tian, Bo Song, and Haoliang Zhang. 2020. Check before You Change: Preventing Correlated Failures in Service Updates. In *NSDI 2020*. USENIX Association, 575–589. https://www.usenix.org/conference/nsdi20/presentation/zhai
[58] Chen Zhang, Bihuan Chen, Xin Peng, and Wenyun Zhao. 2022. Buildsheriff: Change-Aware Test Failure Triage for Continuous Integration Builds. In *ICSE 2022*. ACM, 312–324. https://doi.org/10.1145/3510003.3510132
[59] Mengshi Zhang, Xia Li, Lingming Zhang, and Sarfraz Khurshid. 2017. Boosting spectrum-based fault localization using PageRank. In *ISSTA 2017*. ACM, 261–272. https://doi.org/10.1145/3092703.3092731

[60] Shenglin Zhang, Ying Liu, Dan Pei, Yu Chen, Xianping Qu, Shimin Tao, and Zhi Zang. 2015. Rapid and robust impact assessment of software changes in large internet-based services. In *CoNEXT 2015*. ACM, 1–13. https://doi.org/10.1145/2716281.2836087

[61] Yingying Zhang, Zhengxiong Guan, Huajie Qian, Leili Xu, Hengbo Liu, Qingsong Wen, Liang Sun, Junwei Jiang, Lunting Fan, and Min Ke. 2021. CloudRCA: A Root Cause Analysis Framework for Cloud Computing Platforms. In *CIKM 2021*. ACM, 4373–4382. https://doi.org/10.1145/3459637.3481903

[62] 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 *SOSP 2021*. ACM, 116–131. https://doi.org/10.1145/3477132.3483577

[63] Nengwen Zhao, Junjie Chen, Zhaoyang Yu, Honglin Wang, Jiesong Li, Bin Qiu, Hongyu Xu, Wenchi Zhang, Kaixin Sui, and Dan Pei. 2021. Identifying bad software changes via multimodal anomaly detection for online service systems. In *ESEC/FSE 2021*. ACM, 527–539. https://doi.org/10.1145/3468264.3468543

[64] Yujin Zhao, Ling Jiang, Ye Tao, Songlin Zhang, Changlong Wu, Tong Jia, Xiaosong Huang, Ying Li, and Zhonghai Wu. 2023. Identifying Root-Cause Changes for User-Reported Incidents in Online Service Systems. In *ISSRE 2023*. IEEE, 287–297. https://doi.org/10.1109/ISSRE59848.2023.00028

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

[66] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chao Ji, Dewei Liu, Qilin Xiang, and Chuan He. 2019. Latent error prediction and fault localization for microservice applications by learning from system trace logs. In *ESEC/FSE 2019*. ACM, 683–694. https://doi.org/10.1145/3338906.3338961

Received 2023-09-29; accepted 2024-01-23