Guardian of the Resiliency: Detecting Erroneous Software Changes Before They Make Your Microservice System Less Fault-Resilient

Guanglei He[†§], Xiaohui Nie[‡], Ruming Tang[¶], Kun Wang[†§], Zhaoyang Yu[†§], Xidao Wen[¶], Kanglin Yin[¶],Dan Pei[†§]
[†]Tsinghua University [‡]Computer Network Information Center, Chinese Academy of Sciences [¶]BizSeer
[§]Beijing National Research Center for Information Science and Technology (BNRist)

***Abstract*— The microservice system's resilience is crucial for ensuring the quality of service. Nowadays, software changes are frequent and error-prone, and erroneous software changes could reduce microservice systems' resilience to handle faults, leading to service failures and negatively impacting user experience. To better understand erroneous software changes, we conducted an empirical study on 256 real-world incidents from four famous microservice systems. Our quantitative results indicate that 37.87% of erroneous software changes make the microservice systems less fault-resilient; that is, when a fault (*e.g.* , network fluctuation, high CPU usage, *etc*.) happens in the system after the software change, the services are more likely to experience failures. We refer to these software changes as Erroneous Software Changes that Reduce fault Resilience(*ESCR*). Traditional methods struggle to detect *ESCR*s effectively because the occurrence of faults is unpredictable and can hardly be in their post-change monitoring windows. In this paper, we propose a novel framework named *ResilienceGuardian*, aiming to detect *ESCR*s before they make microservice systems less fault-resilient. The key idea is utilizing fault injection techniques to evaluate systems' fault resilience in the staging environment and then training lightweight classifiers of KPI segment pairs to detect *ESCR*s. The performance of *ResilienceGuardian* is systematically evaluated on three datasets with various faults and erroneous software changes. The results show that *ResilienceGuardian* significantly outperforms all the baselines with a 0.9 F1-score in identifying *ESCR*s and reduces the training time by 56.23% to 97.53%. Besides, *ResilienceGuardian* can achieve minute-level *ESCR* detection in large-scale microservice systems.**

I. Introduction

Microservice architecture has become a staple in developing large-scale online service systems due to its scalability and flexibility. This architecture supports frequent software changes, including bug fixes, configuration adjustments, and the introduction of new features. However, the **fault resilience** of these systems can be compromised by **erroneous software changes** such as misconfigurations or coding errors. As a result, the quality of service provided by microservice systems is at risk, with services becoming more susceptible to failures in the face of unexpected faults like *network fluctuations*, *high CPU usage*, *etc*.. The study [1] reveals that major online service providers, including Google, implement over 10,000 software changes daily. Furthermore, another study [2] reports that approximately 70% of service incidents are attributable to erroneous software changes. Given the frequency of software updates and their potential to introduce latent defects, it is crucial to detect erroneous software changes to maintain the quality of service. Current research predominantly focuses on identifying erroneous software changes by monitoring system performance post-deployment [3, 4, 5]. These methods have proven effective in detecting **E**rroneous **S**oftware **C**hanges that incur immediate **F**ailures following deployment, referred to as *ESCF*s. However, these monitoring approaches are less effective at detecting **E**rroneous **S**oftware **C**hanges that **R**educe fault **R**esilience (*ESCR*), undermining a service's ability to withstand faults. Compared to *ESCF*s, *ESCR*s are unique because they do not impact service quality after the deployment unless the systems experience faults. For one instance in Google [6], the system functioned correctly after deploying an improper reduction of redundant resources for hours until an unexpected workload spike triggered a failure. This incident exemplifies the distinction between *ESCF*s, which cause immediate post-deployment failures, and *ESCR*s, which does not manifest until a specific fault disrupts the system. The difficulty in determining an adequate monitoring duration to cover the occurrence of faults with the balance of overhead makes identifying *ESCR*s particularly challenging.

To develop our idea, we conducted an empirical study of 256 real-world incidents gathered from four widely used microservice systems over six years [7]. Our empirical study reveals the following key observations: (1) *ESCRs are prevalent in microservice systems.* They constitute a significant portion of erroneous software changes, which reduce the systems' fault resilience and compromise service quality when faults disrupt the system. (2) *Faults are frequent and unpredictable in microservice environments*, emphasizing the importance of avoiding *ESCR*s to maintain fault resilience. (3) *Analyzing Key Performance Indicators (KPIs) proves to be an effective strategy for identifying software errors.* However, the large volume of KPIs renders manual analysis impractical. Besides, the uncertainty of fault occurrences in the production environment hinders the detection of *ESCR*s through KPI monitoring. Based on the observations above, our core idea is to *deliberately inject faults into the microservice system*

‡ Xiaohui Nie is the corresponding author.

in a staging environment and use machine learning models to assess fault resilience by comparing pre-change and post-change fault-affected KPIs, thereby detecting ESCRs. With fault injection in the staging environment set up to closely replicate the production environment, we can thoroughly test a system's fault resilience and allow for the early detection of *ESCR*s before production deployment. Below, we summarize the key challenges and our solutions.

(1) **Lack of training data**: Insufficient abnormal data hinders the collection of training data reflecting the impacts of *ESCR*s under various faults. In the empirical study, only 256 incidents were meticulously recorded over six years. Even when utilizing fault injection to build datasets, the absence of domain knowledge (labels) regarding *ESCR*s at the KPI level impedes the training process. To address this challenge, we employ data augmentation to create enriched training datasets with KPI-level labels.

(2) **Complex KPI patterns**: Typically, dozens of faults are injected to assess microservice systems' fault resilience [8]. As faults can affect the pattern of KPIs, injected faults can significantly amplify the complexity of KPI patterns by tenfold or even a hundredfold, requiring a robust model to manage this increased complexity. To tackle this challenge, we intuitively train a separate model for each fault to decrease the KPI complexity faced by each model, achieving advanced performance in detecting *ESCR*s (Section IV-B). Besides, our model focuses on a simpler feature of KPI variations instead of KPI pattern features adopted by previous methods [3, 5]. Specifically, we reframe the comparison of pre-change and post-change KPIs to classify whether the KPI exhibits significant variations post-deployment. In this way, a suite of fault-specific lightweight classifiers is trained to accommodate the extensive size of KPI patterns, albeit exacerbating the third challenge.

(3) Significant overhead : One software change typically demands millions of KPIs to be checked in real-world microservice systems [9]. The overhead of training dozens of fault-specific models with such massive KPIs is substantial, requiring one GPU to run continuously for several months. Besides, in the detection phase, each fault injection requires analysis of massive KPIs, constituting a huge detection space of #fault $\times$ #KPI (tens of millions). To address this challenge, in the training phase, we utilize transfer learning to reduce the training overhead, allowing one fault-specific classifier to be transferred to support other faults at a lower cost. In contrast, the detection overhead is unavoidable. In the detection phase, we propose a highly parallelized strategy to accelerate the detection process. In this way, we effectively handle training and detection overhead, streamlining the overall process.

In summary, we propose a novel framework named *ResilienceGuardian* to identify *ESCR*s in a staging environment through fault injection and KPI analysis. *ResilienceGuardian* uses data augmentation to generate training datasets and build a suite of fault-specific classifiers to handle complex KPI patterns. Furthermore, we propose the use of transfer learning and parallelism strategies to deal with the significant overhead. This work makes three major contributions:

- We conducted a comprehensive empirical study of real-world incidents, uncovering that *ESCR*s significantly affect a microservice system's fault resilience. Our observations motivate the design of *ResilienceGuardian*, which is the first attempt to address *ESCR* identification to the best of our knowledge.
- Our framework, *ResilienceGuardian*, enables the early detection of *ESCR*s before they impact the fault resilience of microservice systems in production. To identify *ESCR*s, *ResilienceGuardian* employs fault-specific classifiers to analyze injected faults in a staging environment. We introduce a data augmentation technique to aid classifier training, along with transfer learning and parallelism strategies to manage the overhead in large-scale microservice systems.
- *ResilienceGuardian* is systematically evaluated on two well-known microservice systems [10, 11]. Extended by several key techniques, *ResilienceGuardian* significantly surpasses baselines, achieving an average F1-score of 0.90 in identifying *ESCR*s. *ResilienceGuardian* reduces the training time by 56.23% to 97.53% compared to baselines and supports **minute-level** detection in large-scale microservice systems that contain millions of KPIs.

II. Empirical Study and Problem Statement

A. Empirical Study

To gain insights into the nature of *ESCR*s, we conducted an empirical analysis of 256 real-world incidents spanning six years. This data was sourced from microservice systems used by leading organizations such as Google, AWS, GitHub, and GitLab [7].

1) Study Method: To facilitate our investigation of *ESCR*s, we categorized incidents by underlying causes. This task was carried out by two experts in microservices. Throughout the labeling process, 98 incidents were excluded from our original set of 354 due to the lack of clear root cause information. Compared to the previous study [7], our empirical study mainly focuses on the following research questions:

- **RQ1:** Are *ESCR*s popular in erroneous software changes?
- **RQ2:** Are faults common in microservice systems?
- **RQ3:** What data can be used to identify *ESCR*s?

2) The Observations for Research Questions: We present three key observations corresponding to the research questions.

Observation 1: ESCRs are prevalent in microservice systems. Erroneous software changes emerge as the primary cause of service failures in production. Among the 256 analyzed incidents, 66.02% of incidents are caused by erroneous software changes. Our analysis revealed that 37.87% of the erroneous software changes qualified as *ESCR*s, which decreased the fault resilience of microservice systems, leading to failures under the occurrence of faults. For instance, an *ESCR* that impairs the functionality of a backup node does not instantly lead to a service failure, as the primary node continues to operate normally. However, when a fault happens in the primary node, causing it to fail, service failure will happen

TABLE I: Typical *ESCR*s and their distribution in 256 incidents

Category	Description	Percentage
I - Insufficient redundancy	Improper reduction of redundant resources requested under faults	29.69%
II - Backup node misconfiguration	Configuring invalid backup nodes under faults	23.44%
III - Problematic request configuration	Permitting problematic requests to perform bad operations	32.81%
IV - Erroneous forwarding strategy	Misconfiguring forwarding strategies for requests under faults	14.06%

because the compromised backup node is incapable of taking over the services. We summarized the typical *ESCR*s in 256 incidents into four categories, as presented in Table I.

Observation 2: Faults are frequent and unpredictable in microservice environments. Microservice systems operate in environments prone to frequent and unpredictable faults. Our study of 256 incidents revealed that 57.42% were related to faults occurring in production. These incidents were not only related to *ESCR*s but also directly resulted from faults due to either increased fault severity or flaws in the original system design. We list typical faults in Table II. Introducing *ESCR*s into this volatile environment poses significant risks, potentially compromising the service quality [8].

TABLE II: Typical faults in the collected incidents

Category	Fault
Problematic workload	*F1* - Workload spike *F2* - Expired requests
Network fluctuation	*F3* - Network delay *F4* - Network packet loss *F5* - Network packet duplication
Host failure	*F6* - Abnormal CPU usage *F7* - Abnormal memory usage *F8* - Disk I/O failure

Observation 3: Analyzing KPIs proves to be an effective strategy for identifying software errors. In these incidents, we observed widespread use of Key Performance Indicator (KPI) monitoring to detect service failures. Typically, two categories of KPIs are employed: business KPIs (*e.g.* request success rate, request duration, *etc.*) and machine KPIs (*e.g.* CPU usage, memory usage, I/O time, *etc.*) [5]. Previous studies [5, 9] utilize these KPIs to identify erroneous software changes. Machine learning techniques have been investigated in these methods to analyze millions of KPIs in real-world microservice systems. Yet, current research struggles to determine an adequate monitoring duration to cover the occurrence of faults with the balance of overhead to detect *ESCR*s.

In summary, these three observations motivate the identification of *ESCR*s with the utilization of fault injection to assess fault resilience. Considering the considerable risk of deploying *ESCR*s in production, it is necessary to propose an automated approach to identify *ESCR*s accurately and efficiently before they make the microservice system less fault-resilient.

B. Problem Statement

We utilize fault injections in a staging environment with KPI monitoring to identify *ESCR*s before their production deployment. The problem statement is as follows: **For a given software change sc, we evaluate it in a staging environment by injecting a set of faults $F = \{f_1, f_2, ..., f_n\}$, which is automated by KPI analysis using a suite of fault-specific classifiers $C = \{c_{f_1}, c_{f_2}, ..., c_{f_n}\}$. The basic task of c_f is to classify a KPI segment pair (s'_f, s^{sc}_f), where s'_f denotes the pre-change KPI segment in the duration of fault f, while s^{sc}_f denotes the post-change KPI segment after deploying sc. The output of c_f indicates the probability that abnormal KPI variations occur. The classification results of C are subsequently aggregated to assess the impacts of sc on fault resilience, aiming to determine whether sc is an *ESCR*.** Therefore, operators can make decisions to prevent the deployment of *ESCR*s in production.

TABLE III: Three KPI segment pairs for one KPI

Name	Segment 1	Segment 2
p_1	pre-change s'_f in d'_f	post-change s^{sc}_f in d^{sc}_f
p_2	pre-change s'_n in d'_n	pre-change s'_f in d'_f
p_3	post-change s^{sc}_n in d^{sc}_n	post-change s^{sc}_f in d^{sc}_f

III. Framework Design

A. Framework Overview

Fig. 1 presents the overview of *ResilienceGuardian*, which is composed of two phases: offline training and online detection. In the offline training phase, we train a suite of classifiers C for a given microservice system in the staging environment. First, we collect data in the microservice system as pre-change KPI segments and utilize data augmentation to generate pseudo-labeled datasets for each fault. Then, we build a suite of fault-specific classifiers C for the detection phases. Each software change, denoted as sc, initiates a single online detection phase. First, we deploy sc and collect data to obtain post-change KPI segments. The segments are combined with pre-change ones to form KPI segment pairs for classification. Subsequently, classification results are aggregated to identify *ESCR*s, which are further organized as a report to assist operators in making final decisions. Changes determined as normal during the detection phase of *ResilienceGuardian* are eligible for deployment in production environments. In the next, we present the key components of *ResilienceGuardian* in detail.

B. Data Collection

We use an open-source monitoring toolkit, Prometheus [12], to collect KPI data. As faults lead to new KPI patterns, we collect one KPI dataset d_f for each injected fault f. In the training phase, *ResilienceGuardian* collects a normal KPI dataset d'_n

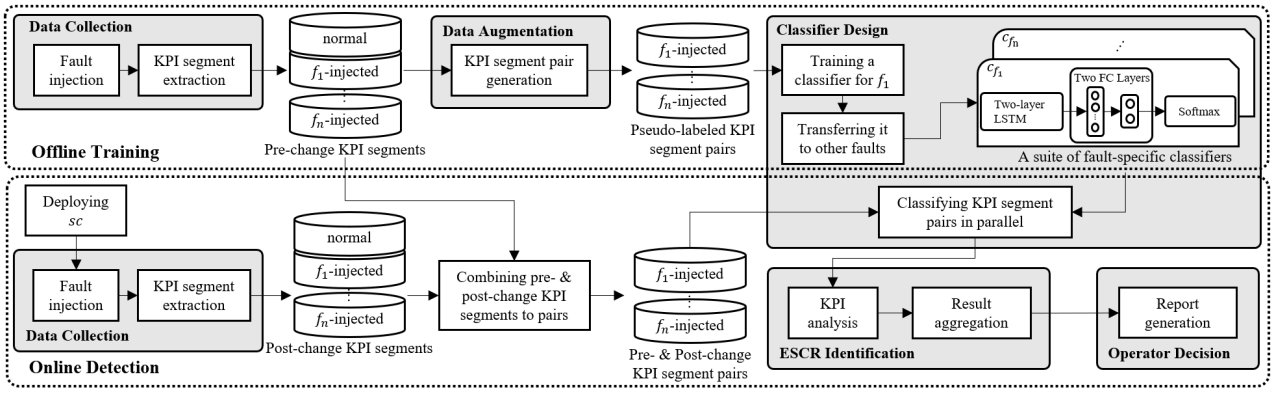


Fig. 1: Overview of *ResilienceGuardian*

and fault-injected KPI datasets $D' = \left\{d'_{f_1}, d'_{f_2}, ..., d'_{f_n}\right\}$. In the detection phase, d^{sc}_n and $D^{sc} = \left\{d^{sc}_{f_1}, d^{sc}_{f_2}, ..., d^{sc}_{f_n}\right\}$ are collected after deploying sc. Then, for each fault, we combine pre-change and post-change data to build a dataset consisting of KPI segment pairs, where one KPI corresponds to a set of three pairs $P = \{p_1, p_2, p_3\}$, as shown in Table III. In addition to the basic task of classifying p_1 (Section II-B), we incorporate the classification of p_2 and p_3 as two supplementary tasks for c_f to support the identification of *ESCR*s described in Section III-E.

1) Fault Injection: In *ResilienceGuardian*, fault injections must encompass typical faults to reveal fault resilience. *ResilienceGuardian* applies a popular fault injection tool, ChaosBlade [13], due to its capability of injecting all the typical faults listed in Table II, as well as its rapid deployment capabilities. Recent research provides strategies to generate fault sets automatically [8]. *ResilienceGuardian* can integrate these strategies to facilitate injections. For a fault set $F = \{f_1, f_2, ..., f_n\}$, a fault f is configured as a 3-tuple (i, t_s, d), where i denotes a fault in Table II, t_s indicates the start time point of the injection, and d represents the injection duration. The shape of the workload, which determines KPI patterns, can potentially impede the analysis of an *ESCR*'s impacts on KPI patterns. Thus, we employ an open-source load testing tool, Locust [14], to manage the workload in the staging environment. In detail, we replay the same workload on a fixed period T. Specifying t_s and d sets the time period of the injection corresponding to a particular segment of workload. The injection duration d is configured according to specific requirements, varying from minutes to hours. A certain interval must elapse between injections to ensure that the system attains a stable, normal state before each subsequent injection.

2) KPI Segment Extraction: In a d, we record the KPI segment s corresponding to the fault duration instead of an entire time series to avoid additional noise and computational overhead. Specifically, for a KPI time series $X = \{x_1, x_2, ..., x_w\}$, we extract the segment $s = \left\{x_{t_s}, x_{t_s+1}, ..., x_{t_s+d}, x_{t_s+d+1}, ..., x_{t_s+d+\lceil 0.5d \rceil}\right\}$. The rationale for appending $\left\{x_{t_s+d+1}, x_{t_s+d+2}, ..., x_{t_s+d+\lceil 0.5d \rceil}\right\}$ lies in the phenomenon where certain faults exert a sustained influence even after the injection ends.

C. Data Augmentation

Training a classifier requires a labeled KPI dataset; nonetheless, manually labeling massive KPIs is excessively time-consuming, constraining the broad application of *ResilienceGuardian*. We utilize data augmentation to generate KPI segment pairs with pseudo-labels. For each classifier c_f, we generate a labeled dataset from d'_f. A labeled KPI pair is composed of a s'_f in d'_f and a new segment generated from s'_f. We aim at classifiers capable of recognizing variations in KPI segment pairs; thus, the generation of new KPI segments is supposed to represent the potential variations.

TABLE IV: Descriptions of *ESCR*s designed in this paper

Category	Description
I	*E1* - Reducing CPU resources improperly *E2* - Reducing memory resources improperly
II	*E3* - Configuring insufficient CPU resources *E4* - Configuring insufficient memory resources
III	*E5* - Permitting expired requests to access invalid databases *E6* - Permitting expired requests to invoke death loops
IV	*E7* - Interrupting the forwarding of requests *E8* - Forwarding requests to invalid backup nodes

1) A study of KPI variations: To investigate variations in KPI segment pairs, we used two widely used benchmark microservice systems, HipsterShop [10] and Train-Ticket [11], to generate 200 instances of *ESCR*s derived from real-world incidents, as outlined in Table I. Table IV gives detailed descriptions of the *ESCR*s designed by us. For each *ESCR*, we generated 10 instances in HipsterShop and 15 instances in Train-Ticket. The study comprised two steps. First, we collected d'_n and D' in the microservice system. Then, we deployed *ESCR* instances (sc) respectively. After each deployment of sc, we collected d^{sc}_n and D^{sc}. For each *ESCR*, we constructed P (Table III) for each KPI. We formed a dataset with all the KPI pairs and applied OmniCluster [15] to cluster the dataset. The output clusters contained KPI pairs with similar variations, where we observed 6 categories [16] of variations illustrated in Fig. 2.

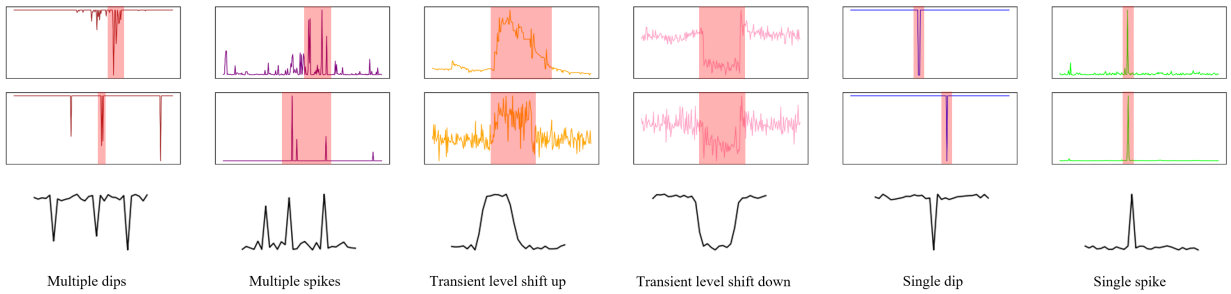


Fig. 2: Six categories of variations in KPI segment pairs and some examples. Red areas denote *ESCR*s' impacts on KPIs.

2) KPI Segment Pair Generation: Fig. 3 shows the generation process of one KPI segment pair, containing two steps. First, we randomly select a KPI segment and inject noise into it to generate a new one with varied patterns, constituting a KPI segment pair. Then, noise intensity scaling is performed on generated pairs to build the training dataset. Below, we present the process of noise injection and noise intensity scaling.

Noise injection. A KPI time series' specific characteristics, noise intensity [9], is utilized to measure the amount of injected noise. In the staging environment, KPIs exhibit periodicity because we replay the workload on a fixed period T (Section III-B1). Noise intensity NI denotes the mean of the standard deviations over all periods at each time point. For a time series $X = \{x_0, x_1, x_2, ..., x_{T-1}, x_{0+T}, ..., x_{K\times T-1}\}$ containing K periods, NI_X is calculated as Eq. (1) shows.

$$NI_X = \frac{1}{T}\sum_{i=0}^{T-1}\sqrt{\frac{1}{K}\sum_{j=0}^{K-1}\left(x_{i+j\times T} - \frac{1}{K}\sum_{j=0}^{K-1} x_{i+j\times T}\right)^2} \tag{1}$$

We configure weak noise as $(0\times NI, 3\times NI]$, and strong noise as $(3\times NI, 10\times NI]$ referring to k-σ rule [17] widely used in industry. Strong noise is injected in positive instances to induce variations in the modified segment. On the contrary, negative instances are generated by injecting weak noise. We designed two injectors, I^p and I^n, to inject noise of specific patterns. I^p generates positive instances by injecting strong noise whose patterns match with the 6 categories of variations in Fig. 2. To enhance the variety of generated segments, we select a random subset of variations when injecting noises into a specific segment instead of considering 6 categories mutually exclusive. The generated segments comprehensively reflect the potential variations, assuring the validity and quality of training datasets. I^n injects Gaussian noise configured by a distribution $N \sim (0, NI^2)$ to generate negative instances, aiming to simulate random noise in a time series [9, 16].

Noise intensity scaling. Because KPIs in a microservice system might have NIs that vary tens of times, the different amount of injected noise leads to the low performance of one classifier to classify all KPI pairs with distinct NIs [9]. To solve this problem, we propose noise intensity scaling to mitigate the impact of NI variations. Specifically, we set the default noise intensity, $NI_{default}$. For a KPI segment $\{x_0, x_1, ..., x_i, ..., x_m\}$, the scaling process is expressed as Eq. (2), where NI_X denotes NI of a KPI time series X from which the segment derives.

$$x_{i\text{-}scaled} = x_i \times \frac{NI_{default}}{NI_X} \tag{2}$$

The scaled segments replace the original ones to constitute the training set. This technique enables a classifier to classify KPI segment pairs with diverse NIs. The configuration of $NI_{default}$ is to standardize the NI across different KPIs; thus, the specific value of $NI_{default}$ does not affect the effectiveness of noise intensity scaling. In *ResilienceGuardian*, we set $NI_{default}$ to 1.5.

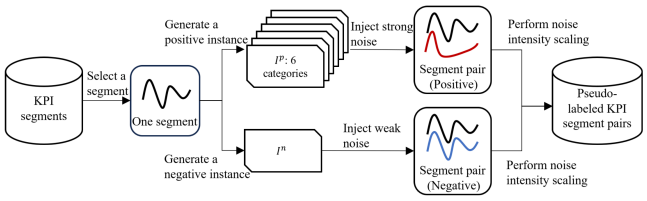


Fig. 3: Generate labeled pairs for one KPI segment

D. Classifier Design

The classifier design within *ResilienceGuardian* is tailored to address the heightened complexity of KPI patterns resulting from fault injections, with particular emphasis on managing the substantial training and detection overhead.

1) Classification Model: We employ a fault-specific strategy to train individual classifiers for each fault, resulting in a suite of classifiers capable of effectively handling complex KPI patterns induced by injected faults. Besides, the strategy makes it convenient to incorporate or remove classifiers to align with F, exhibiting flexibility in maintaining the fault set F to adapt to the evolution of microservice systems. We design a compact deep-learning model for classifiers to mitigate the training overhead. A lightweight network, long short-term memory (LSTM), is adopted owing to its advanced capability to process complex sequential data [5, 18]. This choice yields satisfying performance in identifying *ESCR*s as detailed in Section IV-B; thus, more complex transformer-based networks are not explored. The component of classifier design in Fig. 1 illustrates the classification model. In the model, a two-layer LSTM is adopted to extract fixed-length features from a KPI segment pair. The features are then fed into fully connected (FC) layers to output the probability of the positive label.

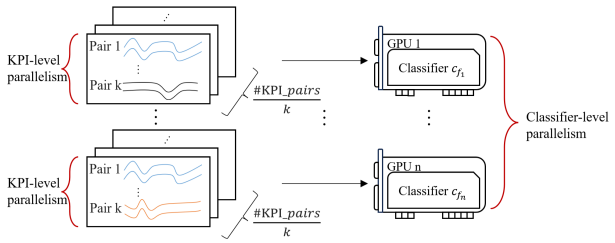


Fig. 4: A parallelism strategy, where n classifiers work in parallel and each classifier classifies k KPI pairs in parallel.

The value of probability quantifies variations in a pair's two segments, facilitating the subsequent identification of *ESCR*s (Section III-E). In the training process, binary cross-entropy losses are adopted.

2) Transfer Learning: A comprehensive assessment of fault resilience in a microservice system typically necessitates a fault set F comprising dozens of faults [8]. Implementing the fault-specific strategy by directly training classifiers separately amplifies the training overhead by tens of times, which is unacceptable as the overhead of training one classifier is substantial in real-world microservice systems housing millions of KPIs. We propose a transfer learning technique to mitigate the training overhead. Section IV-D2 evaluates the utilization of one fault-specific classifier for other faults, where although the performance is degraded, the classifier still outperforms most baselines. This result indicates that the classifier acquires the ability to extract features of KPI pair variations across faults, as the generated training dataset encompasses a diverse range of variations. Thus, the key to avoiding performance degradation is to exclude interfering information regarding specific KPI patterns in the features output from LSTM, which can be accomplished by fine-tuning the parameters of subsequent FC layers. This inspires us to utilize a smaller fine-tuning dataset to transfer a fault-specific classifier for application to other faults. Initially, one classifier c_f is trained for a fault f. Then, c_f is transferred to classifiers for other faults. Specifically, we freeze c_f's parameters in LSTM layers and fine-tune the FC layers.

3) Parallelism Strategy: To manage the significant detection overhead, the classification model is designed to classify each KPI segment pair independently, which enables a highly parallelized strategy for accelerating the detection. Furthermore, the detailed outputs of abnormal KPIs can assist engineers in conducting a deeper analysis of defects in *ESCR*s and guiding subsequent development efforts. The strategy is configured as a 2-tuple (k, n), where k denotes a classifier processes k KPI pairs in parallel, and n denotes n classifiers work in parallel. The value of k depends on the size of GPU memory resources and the length of KPI segments. Engineers can configure n to equal the number of available GPUs. Fig. 4 illustrates the application of a strategy (k, n).

E. ESCR Identification

KPI variations cannot be used directly as *ESCR*'s indicators because variations might be attributed to a normal change. Certain changes enhancing fault resilience also influence KPI patterns during fault occurrences. In *ResilienceGuardian*, the identification of *ESCR*s is composed of two phases: KPI-level analysis and result aggregation.

1) KPI-level analysis: In the first phase, for each injected fault f, we analyze the classification results of each KPI's three pairs (Table III) and divide KPIs into 4 categories. The first phase contains two steps. First, the classification of p_1 is performed for every KPI, whose output O_{p_1} quantifies KPI pattern variations that reflect the impacts of software changes. We assign a KPI to category C_1 if its O_{p_1} does not exceed a threshold δ, indicating that the KPI's pattern remains consistent after deploying the software change. The default value of δ is set to 0.5, given that O_{p_1} is a probability value ranging from 0 to 1. We conduct the second step for other KPIs in which variations exist, containing classifications of p_2 and p_3. O_{p_2} denotes the impacts of faults on the pre-change microservice system. O_{p_3} denotes the impacts of faults after deploying the software change. In O_{p_2} and O_{p_3}, a smaller value indicates that patterns are more similar, reflecting a more resilient microservice system. Hence, the lack of a significant difference ($|O_{p_2} - O_{p_3}| < \sigma$) between O_{p_2} and O_{p_3} means that the software change affects KPI patterns rather than the system's resilience (C_2). Such a change is also acceptable. The value of σ is determined by the dispersion of O_{p_2} and O_{p_3}; thus, we utilize k-σ rule [17] to set σ to $3 \times \min(\sigma_{O_{p_2}}, \sigma_{O_{p_3}})$, where $\sigma_{O_{p_2}}$ and $\sigma_{O_{p_2}}$ denote the standard deviations of O_{p_2} and O_{p_3}. A greater value for O_{p_2} compared to O_{p_3} indicates that the change enhances resilience (C_3), whereas the inverse suggests that resilience is compromised by the change (C_4).

2) Result aggregation: In the second phase, we aggregate the results to identify *ESCR*s. We calculate a vulnerability score vs_f for each member f in the fault set F, resulting in $VS = \{vs_{f_1}, vs_{f_2}, ..., vs_{f_n}\}$. The vs derives from aggregating the classification outputs of KPIs categorized into C_4. Eq. (3) shows the calculation of vs_f.

$$vs_f = \sum^{KPI \in C_4} \frac{|O_{p_2} - O_{p_3}|}{\sigma} O_{p_1} \tag{3}$$

The magnitude of $|O_{p_2} - O_{p_3}|$ signifies the extent to which variations are associated with resilience degradation, thereby being utilized to scale O_{p_1}. A threshold θ is set to determine whether a vs_f indicates an *ESCR*. Owing to the skewed distribution of vs_fs, the *knee-point* method [19] can be applied to automatically pick the knee-point as θ. If one of vs_fs in VS exceeds θ, the software change is identified as an *ESCR*.

F. Operator Decision

Despite the results recommended by *ResilienceGuardian*, automated algorithms could not make the final decision. Operators should confirm these results to avoid false positives and negatives. Following each detection phase, *ResilienceGuardian* records the KPI-level analysis and the aggregation result to generate a detection report of the evaluated software change, aiming to facilitate the manual decision process. Fig. 5 shows an example of the report for an *ESCR*.

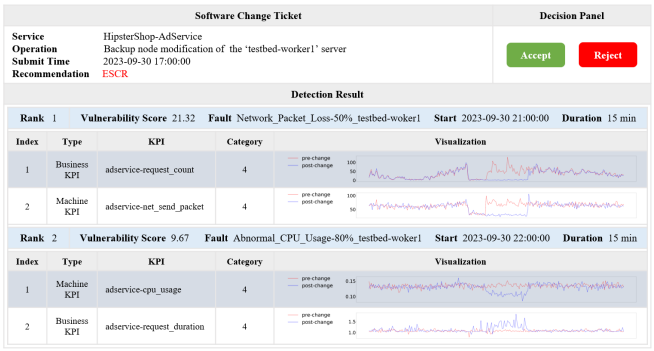


Fig. 5: A demo of report generated by *ResilienceGuardian*

IV. EVALUATION

To evaluate the performance of *ResilienceGuardian*, we conducted experiments to answer four research questions:

- **RQ1**: What is the performance of *ResilienceGuardian* in identifying *ESCR*s?
- **RQ2**: What is the performance of the classification model in *ResilienceGuardian*?
- **RQ3**: What is the contribution of key components to *ResilienceGuardian*'s performance?
- **RQ4**: What is the impact of related hyperparameters?

A. Experiment Setup

1) Dataset and Metrics: Three datasets are used in the evaluation.

Dataset $\mathcal{A}$ and Dataset $\mathcal{B}$. We adopt dataset $\mathcal{A}$ and dataset $\mathcal{B}$ to evaluate *ResilienceGuardian*'s performance in identifying *ESCR*s. These datasets are collected from two widely adopted benchmark microservice systems, HipsterShop [10] and Train-Ticket [11], respectively. The sizes of the monitored KPI sets configured in HipsterShop and Train-Ticket are 100 and 410. The corresponding F is composed of 10 typical faults in production. The construction of dataset $\mathcal{A}$ and dataset $\mathcal{B}$ uses the data collection component in *ResilienceGuardian*, consisting of D', D^{sc}, d'_n, and d^{sc}_n. Dataset $\mathcal{A}$ has 80 *ESCR*s and 50 normal changes in HipsterShop, containing 286,000 KPI segments in total. Dataset $\mathcal{B}$ has 120 *ESCR*s and 75 normal changes in Train-Ticket, containing 1,758,900 KPI segments in total. The design of these *ESCR*s has been presented in Section III-C1. With the utilization of fault injection in *ResilienceGuardian*, baseline approaches can detect *ESCR*s by analyzing the fault-affected KPI data in these two datasets.

Dataset $\mathcal{C}$. The third dataset is the UEA classification archive [20], a popular choice for evaluating classification algorithms. It consists of 30 datasets, representing a variety of critical classification domains. We adopt dataset $\mathcal{C}$ to demonstrate the classification ability of *ResilienceGuardian*.

Metrics. We adopt the precision (P), recall (R), and F1-score (F1) to evaluate the effectiveness of approaches. To evaluate the overhead of *ResilienceGuardian*, the training time of models and the average duration of the detection phase for each software change is additionally considered. When employing the UEA archive in RQ2, we use a **C**ritical **D**ifference (CD) diagram [21], which is a powerful tool for performance comparison over multiple datasets [22]. The horizontal coordinates of the CD diagram indicate the mean ranks of the models for all datasets, where lower ranks correspond to higher performance. The solid line is created by Wilcoxon-Holm tests, indicating no significant difference in performance among the models crossed by that line. The running time compared in the CD diagram refers to the duration to complete the classification of the entire UEA archive.

2) Implementations and Parameters: We conducted all experiments on Ubuntu 20.04.6 LTS with an Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40GHz, a 64-bit operating system, and one NVIDIA GeForce RTX 3090 GPU. All methods were implemented in Python. The parameters of *ResilienceGuardian* are as follows. We generated 1000 positive and negative instances for each KPI to build the training dataset, while the fine-tuning dataset contained 10 positive and negative instances per KPI. For the classification model, the hidden sizes of two-layer LSTM were configured as 100 and 60, respectively. The hidden sizes of the two FC layers were set to 30 and 2. During training, the batch size was 1000, the number of epochs was 100, and the Adam optimizer was adopted with a learning rate of 0.001. In the detection phase, we configured parallelism strategies to process all KPI pairs in each change simultaneously due to the small scale of HipsterShop and Train-Ticket. For the compared baselines, we used the parameters provided in their papers and open-source codes. In addition, we configured a uniform early stopping strategy for the deep classification models to avoid overfitting.

B. RQ1: ResilienceGuardian's performance for ESCR Identification

We used dataset $\mathcal{A}$ and dataset $\mathcal{B}$ to perform the evaluation.

1) Baselines: The compared approaches could be divided into two categories by deriving domains.

Erroneous software change identification. **Gandalf** [3] and **SCWarn** [5] detect anomalies attributed to *ESCR*s after deployments in the fault-affected KPIs. Gandalf implements an anomaly detection component whose design is not clear; thus, we followed prior studies [5, 9] to use HoltWinters. SCWarn predicts post-change KPIs using a multi-modal LSTM and compares predictions with real ones. Differing from preceding approaches, **Kontrast** [9] utilizes contrastive learning to directly compare pre-change and post-change KPIs.

Anomaly Detection. We chose three baseline approaches that can be adapted for identifying *ESCR*s. **Lumos** [4] uses a t-test to compare pre-change and post-change KPIs [5, 23]. **Donut** [24] builds a generative model to compute the reconstruction probability of post-change KPIs. **Telemanom** [18] uses LSTM to predict post-change KPIs.

2) Experiment results: ResilienceGuardian outperforms all other approaches, as shown in Table V, achieving 0.90 and 0.89 F1-score in two microservice systems. In comparison, the optimal baseline, Kontrast, achieves an average F1-score of 0.82. Furthermore, *ResilienceGuardian* achieves a training time reduction of at least 56.23% on both two datasets,

TABLE V: Performance of approaches on dataset $\mathcal{A}$ and dataset $\mathcal{B}$

Task	Approach	Dataset $\mathcal{A}$ P	R	F1	Training(min)	Detection(s)	Dataset $\mathcal{B}$ P	R	F1	Training(min)	Detection(s)
ESCR identification	Gandalf	0.74	0.68	0.71	87.32	34.32	0.72	0.66	0.69	355.21	31.09
	SCWarn	0.64	0.59	0.61	23.91	9.35	0.69	0.65	0.67	77.36	38.18
	Kontrast	0.88	0.81	0.84	290.74	***0.10***	0.82	0.79	0.80	693.33	***0.11***
	Lumos	0.55	0.70	0.62	-	15.12	0.63	0.59	0.61	-	19.07
	Donut	0.78	0.54	0.64	327.86	17.24	0.72	0.67	0.69	1368.57	21.38
	Telemanom	0.59	0.67	0.63	197.31	5.24	0.55	0.58	0.56	814.89	5.32
	ResilienceGuardian	***0.91***	***0.89***	***0.90***	***8.32***	0.12	***0.87***	***0.92***	***0.89***	***33.86***	0.12
Ablation study	*ResilienceGuardian*$_{\text{pre}}$	0.79	0.83	0.81	9.42	0.12	0.81	0.78	0.79	35.42	0.12
	ResilienceGuardian$_{\text{cate}}$	0.77	0.87	0.82	20.85	0.13	0.83	0.80	0.81	68.54	0.14
	ResilienceGuardian$_{\text{one}}$	0.83	0.79	0.81	6.88	0.12	0.76	0.87	0.81	31.09	0.13
	ResilienceGuardian$_{\text{all}}$	0.94	0.91	0.92	68.79	0.12	0.93	0.95	0.94	310.92	0.12

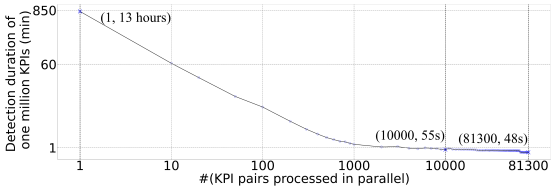
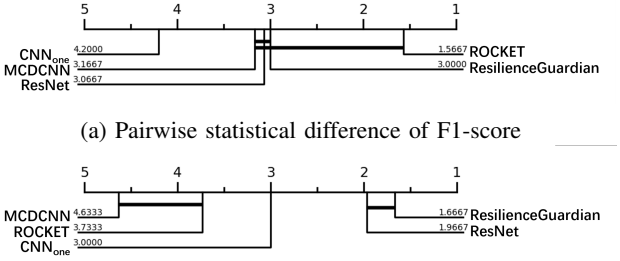


Fig. 6: The relationship between the number of KPI pairs processed in parallel and the detection duration



(a) Pairwise statistical difference of F1-score

(b) Pairwise statistical difference of running time

Fig. 7: Critical difference diagrams on dataset $\mathcal{C}$

This is primarily attributed to the lightweight classifiers design and transfer learning, which significantly facilitates *ResilienceGuardian*'s deployment in large-scale microservice systems. When considering the average duration of a single software change's detection phase, both *ResilienceGuardian* and Kontrast surpass other approaches, achieving an approximate processing time of 0.1 seconds per change. To further demonstrate the advanced scalability of *ResilienceGuardian* utilizing parallelism strategies, we build a dataset containing one million KPIs by duplicating KPI segment pairs. Fig. 6 illustrates the relationship between the number of KPI pairs processed in parallel and the detection duration of the dataset, with both axes logarithmically scaled. We only explore strategies with n set to 1 due to the satisfactory performance achievable with a single 3090 GPU. Empirical tests demonstrate that a 3090 GPU can classify approximately up to 81,300 KPI pairs in parallel, corresponding to a $(2710, 1)$ strategy, as one KPI leads to 30 pairs in *ResilienceGuardian*. The optimal strategy accomplishes the detection phase within a mere 48 seconds, whereas the serial strategy demands nearly 13 hours. In conclusion, *ResilienceGuardian* is a competitive approach for accurately and timely identifying *ESCR*s before their deployments, thereby ensuring the fault resilience of microservice systems in production. Moreover, the CPU and memory consumption of data collection and fault injections in *ResilienceGuardian* is less than 5% for each pod in the microservices, indicating that it does not impact the operation of microservices.

C. RQ2: ResilienceGuardian for Classification

The fundamental task in *ResilienceGuardian* is to classify KPI segment pairs, which is crucial for identifying *ESCR*s.

1) Baselines: The evaluation involves the following baselines: **ROCKET** [25] is a SOTA non-deep classifier [22] that utilizes random convolution kernels to obtain feature maps. **ResNet** [26] is a SOTA deep classifier [22] that comprises a deep architecture containing 9 convolutional layers connected by residual shortcuts. **MCDCNN** [27] is a traditional deep CNN that applies convolutions independently to each dimension of the time series. In addition, we implemented **CNN$_{\text{one}}$** by simply replacing LSTM with CNN in *ResilienceGuardian*'s model. The primary difference between CNN$_{\text{one}}$ and MCDCNN is that CNN$_{\text{one}}$ applies one convolution to all dimensions of the input. We added MCDCNN and CNN$_{\text{one}}$ to demonstrate the benefits of utilizing LSTM.

2) Experiment results: Fig. 7a indicates no significant difference in performance between *ResilienceGuardian* and ROCKET in F1-score. While ROCKET outperforms other deep learning approaches, it becomes computationally intensive to achieve outstanding accuracy, as shown in Fig. 7b. Consequently, it is impractical for large-scale microservice systems containing millions of KPIs. *ResilienceGuardian* and ResNet are similarly efficient and significantly better than ROCKET. On average, ROCKET requires 9 times longer for classification tasks than *ResilienceGuardian*. Furthermore, the superiority of *ResilienceGuardian* over MCDCNN and CNN$_{\text{one}}$ verifies the adoption of LSTM for classification purposes. In summary, the findings from dataset $\mathcal{C}$ demonstrate that *ResilienceGuardian* is generally competitive in typical classification tasks, attributed to a satisfactory balance between effectiveness and efficiency.

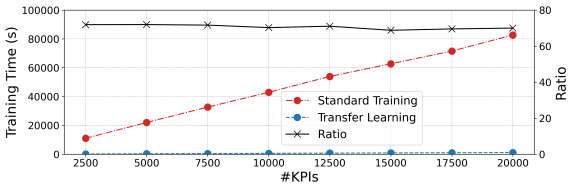
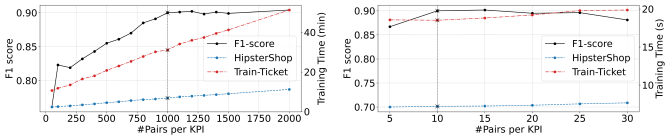


Fig. 8: The acceleration ratio of transfer learning



(a) Hyperparameter-α (b) Hyperparameter-β

Fig. 9: Impacts of α and β on two microservice systems

D. RQ3: Ablation Study of Key Components

1) Data Augmentation: To evaluate the design of data augmentation in *ResilienceGuardian*, we compared it with the SOTA augmentation technique proposed in Kontrast. There are two primary distinctions: First, compared with the predefined injector I^{pre} in Kontrast, I^p and I^n in *ResilienceGuardian* are crafted to inject noise aligned with variations identified in clustering research on KPI pairs. Second, while Kontrast divides KPIs into 5 categories to mitigate the diversity of NI and generates a dataset for each category, we employ noise intensity scaling to produce a single dataset. Therefore, we evaluated two variants of *ResilienceGuardian*. *ResilienceGuardian*$_{\text{pre}}$ is built by replacing I^p and I^n with I^{pre}. *ResilienceGuardian*$_{\text{cate}}$ is built by replacing the scaling technique with the classifier. As listed in Table V, *ResilienceGuardian*$_{\text{pre}}$ achieves an average F1-score of 0.80, validating the superiority of our injector design. *ResilienceGuardian*$_{\text{cate}}$ requires nearly double the time of *ResilienceGuardian* to achieve an average F1-score of 0.82. We present an analysis using HipsterShop as a case study. According to Kontrast's design, a total of 400,000 KPI pairs are generated for 5 training datasets, while utilizing the scaling technique generates one dataset containing 200,000 pairs, leading to a reduction in the training time. Besides, the set of 100 KPIs monitored in HipsterShop contains dozens of distinct NI, rendering the division into 5 NI categories coarse-grained. Scaling KPI pairs to a uniform $NI_{default}$ demonstrates better performance.

2) Transfer Learning: We constructed two variants of *ResilienceGuardian* to demonstrate the benefits of utilizing transfer learning. *ResilienceGuardian*$_{\text{one}}$ trains one fault-specific classifier and employs it for all faults, while *ResilienceGuardian*$_{\text{all}}$ trains the suite of classifiers without leveraging transfer learning. *ResilienceGuardian* achieves a better F1-score (0.9) than *ResilienceGuardian*$_{\text{one}}$'s F1-score (0.8) and only increases 1.44 minutes of training time. Compared with *ResilienceGuardian*$_{\text{all}}$, *ResilienceGuardian* can reduce 88.51% training time and only decrease the F1-score by 3% on average. These observations prove that the design of transfer learning in Section III-D2 can highly speed up the model training process. Building upon the parameters outlined in Section IV-A2, we expanded the training set through duplication to further investigate the benefits of implementing transfer learning within a larger microservice system containing more KPIs. Fig. 8 illustrates that the acceleration ratio of transfer learning stabilizes at approximately 70 times that of standard training as the number of KPIs increases. Therefore, we could calculate the approximate acceleration ratio as $\frac{\#F\times 70}{(\#F-1)\times 1+70}$ for a complete training phase. The effect of transfer learning increases as the fault set F expands. In dataset $\mathcal{A}$ and dataset $\mathcal{B}$ where the size of F is 10, the calculated ratio is 8.86, approximately equal to the real average ratio of 8.73.

E. RQ4: Impacts of Hyperparameters

In *ResilienceGuardian*, the size of training and fine-tuning datasets affects the effectiveness of corresponding classifiers and determines the training overhead. Two hyperparameters configure the size of these generated datasets.

1) #Pairs per KPI for Training (α): α determines the size of the training dataset. Fig. 9a shows that both the F1-score and training time of one classifier increase as α becomes larger. The performance is relatively stable when α exceeds 1000. Thus, α is set to 1000 in this paper.

2) #Pairs per KPI for Transfer Learning (β): β determines the size of the fine-tuning dataset. Fig. 9b illustrates that small datasets are sufficient for transfer learning, as the F1-score is insensitive to small values of β. This finding verifies that transfer learning can be accomplished with minimal investment. Thus, we set β to 10 in this paper.

V. DISCUSSION

There are two primary limitations of *ResilienceGuardian*. The first is the lack of utilizing multiple data sources. Existing research has explored the feasibility of using traces and logs to facilitate the identification of erroneous software changes [3, 5]. The potential benefit of using multiple data sources can be studied in the future. The second is the risk that the performance of detecting *ESCR*s in the staging environment is not necessarily equivalent to that in the production environment. However, *ResilienceGuardian* can also work in the production environment where fault injection is acceptable. Therefore, subsequent research has two directions: (1) Controlling the risks associated with change deployment and fault injection in production. (2) Evaluating and reducing disparities between the staging and production environment.

VI. RELATED WORK

A. Fault Injection

In the current research, several tools have been proposed to inject faults into microservice systems [13, 28, 29]. These tools support injections of an extensive range of faults, capable of covering typical faults in production. The blast radius is an important feature to characterize the potential scope and impact of injected faults. Recent investigations advocate for the incorporation of multiple blast radii, spanning service [29],

service instance [13], and request levels [28]. However, achieving the utmost granularity with request-level injections may necessitate modifications tailored to the target microservice system [28], hindering widespread adoption.

B. Erroneous Software Change Identification

Current research identifies erroneous software changes by comparing a system's pre-change and post-change behaviors [3, 4, 5, 9]. These approaches extract expected patterns of post-change behaviors from pre-change data, such as KPIs and logs [3, 5]. Gandalf [3] and SCWarn [5] convert the problem to anomaly detection and predict the post-change behaviors, while Lumos [4] and Kontrast [9] directly compare the pre-change and post-change behavior patterns.

VII. CONCLUSION

This paper conducts an empirical study on 256 real-world incidents from several enterprises. Our quantitative analysis reveals the importance of avoiding *ESCR*s to maintain the quality of service. Furthermore, the observations motivate us to propose a novel framework named *ResilienceGuardian*, which aims to identify *ESCR*s in a staging environment through fault injection and parallel KPI classification. The output of *ResilienceGuardian* can effectively assist engineers in carrying out proactive operations to prevent *ESCR*s with greater ease. In the evaluation, *ResilienceGuardian* achieves outstanding F1-score in identifying *ESCR*s, outperforming SOTA baselines. Due to several novel model designs, *ResilienceGuardian* can significantly reduce training overhead and support minute-level *ESCR* detection in large-scale microservice systems.

ACKNOWLEDGMENT

This work is supported by the National Natural Science Foundation of China under Grant 62072264, and the Beijing National Research Center for Information Science and Technology (BNRist) key projects.

REFERENCES

[1] A. Singleton, "The economics of microservices," *IEEE Cloud Computing*, vol. 3, no. 5, pp. 16–20, 2016.

[2] B. Beyer, C. Jones, J. Petoff, and N. R. Murphy, *Site reliability engineering: How Google runs production systems*. " O'Reilly Media, Inc.", 2016.

[3] Z. Li, Q. Cheng, K. Hsieh, Y. Dang, P. Huang, P. Singh, X. Yang, Q. Lin, Y. Wu, S. Levy *et al.*, "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)*, 2020, pp. 389–402.

[4] J. Pool, E. Beyrami, V. Gopal, A. Aazami, J. Gupchup, J. Rowland, B. Li, P. Kanani, R. Cutler, and J. Gehrke, "Lumos: A library for diagnosing metric regressions in web-scale applications," in *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2020, pp. 2562–2570.

[5] N. Zhao, J. Chen, Z. Yu, H. Wang, J. Li, B. Qiu, H. Xu, W. Zhang, K. Sui, and D. Pei, "Identifying bad software changes via multimodal anomaly detection for online service systems," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 527–539.

[6] Google cloud. https://status.cloud.google.com/summary, accessed 2023-10-26.

[7] X. Li, G. Yu, P. Chen, H. Chen, and Z. Chen, "Going through the life cycle of faults in clouds: Guidelines on fault handling," in *2022 IEEE 33rd International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2022, pp. 121–132.

[8] P. Alvaro, K. Andrus, C. Sanden, C. Rosenthal, A. Basiri, and L. Hochstein, "Automating failure testing research at internet scale," in *Proceedings of the Seventh ACM Symposium on Cloud Computing*, 2016, pp. 17–28.

[9] X. Wang, K. Yin, Q. Ouyang, X. Wen, S. Zhang, W. Zhang, L. Cao, J. Han, X. Jin, and D. Pei, "Identifying erroneous software changes through self-supervised contrastive learning on time series data," in *2022 IEEE 33rd International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2022, pp. 366–377.

[10] Hipstershop. https://github.com/GoogleCloudPlatform/microservices-demo, accessed 2023-10-26.

[11] Train-ticket. https://github.com/FudanSELab/train-ticket, accessed 2023-10-26.

[12] Prometheus. https://prometheus.io/, accessed 2023-10-26.

[13] Chaosblade. https://github.com/chaosblade-io/chaosblade, accessed 2023-10-26.

[14] Locust. https://locust.io/, accessed 2023-10-26.

[15] S. Zhang, D. Li, Z. Zhong, J. Zhu, M. Liang, J. Luo, Y. Sun, Y. Su, S. Xia, Z. Hu *et al.*, "Robust system instance clustering for large-scale web services," in *Proceedings of the ACM Web Conference 2022*, 2022, pp. 1785–1796.

[16] C. Wu, N. Zhao, L. Wang, X. Yang, S. Li, M. Zhang, X. Jin, X. Wen, X. Nie, W. Zhang *et al.*, "Identifying root-cause metrics for incident diagnosis in online service systems," in *2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2021, pp. 91–102.

[17] 3-sigma rule. https://en.wikipedia.org/wiki/68-95-99.7_rule, accessed 2023-10-26.

[18] K. Hundman, V. Constantinou, C. Laporte, I. Colwell, and T. Soderstrom, "Detecting spacecraft anomalies using lstms and nonparametric dynamic thresholding," in *Proceedings of the 24th ACM SIGKDD international conference on knowledge discovery & data mining*, 2018, pp. 387–395.

[19] V. Satopaa, J. Albrecht, D. Irwin, and B. Raghavan, "Finding a" kneedle" in a haystack: Detecting knee points in system behavior," in *2011 31st international conference on distributed computing systems workshops*. IEEE, 2011, pp. 166–171.

[20] A. Bagnall, H. A. Dau, J. Lines, M. Flynn, J. Large, A. Bostrom, P. Southam, and E. Keogh, "The uea multivariate time series classification archive, 2018," *arXiv preprint arXiv:1811.00075*, 2018.

[21] J. Demšar, "Statistical comparisons of classifiers over multiple data sets," *The Journal of Machine learning research*, vol. 7, pp. 1–30, 2006.

[22] A. P. Ruiz, M. Flynn, J. Large, M. Middlehurst, and A. Bagnall, "The great multivariate time series classification bake off: a review and experimental evaluation of recent algorithmic advances," *Data Mining and Knowledge Discovery*, vol. 35, no. 2, pp. 401–449, 2021.

[23] S. Levy, R. Yao, Y. Wu, Y. Dang, P. Huang, Z. Mu, P. Zhao, T. Ramani, N. Govindaraju, X. Li *et al.*, "Predictive and adaptive failure mitigation to avert production cloud {VM} interruptions," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020, pp. 1155–1170.

[24] H. Xu, W. Chen, N. Zhao, Z. Li, J. Bu, Z. Li, Y. Liu, Y. Zhao, D. Pei, Y. Feng *et al.*, "Unsupervised anomaly detection via variational auto-encoder for seasonal kpis in web applications," in *Proceedings of the 2018 world wide web conference*, 2018, pp. 187–196.

[25] A. Dempster, F. Petitjean, and G. I. Webb, "Rocket: exceptionally fast and accurate time series classification using random convolutional kernels," *Data Mining and Knowledge Discovery*, vol. 34, no. 5, pp. 1454–1495, 2020.

[26] Z. Wang, W. Yan, and T. Oates, "Time series classification from scratch with deep neural networks: A strong baseline," in *2017 International joint conference on neural networks (IJCNN)*. IEEE, 2017, pp. 1578–1585.

[27] Y. Zheng, Q. Liu, E. Chen, Y. Ge, and J. L. Zhao, "Exploiting multi-channels deep convolutional neural networks for multivariate time series classification," *Frontiers of Computer Science*, vol. 10, pp. 96–112, 2016.

[28] J. Zhang, R. Ferydouni, A. Montana, D. Bittman, and P. Alvaro, "3milebeach: A tracer with teeth," in *Proceedings of the ACM Symposium on Cloud Computing*, 2021, pp. 458–472.

[29] C. S. Meiklejohn, A. Estrada, Y. Song, H. Miller, and R. Padhye, "Service-level fault injection testing," in *Proceedings of the ACM Symposium on Cloud Computing*, 2021, pp. 388–402.