IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 5 February 2024, accepted 1 March 2024, date of publication 7 March 2024, date of current version 14 March 2024.

Digital Object Identifier 10.1109/ACCESS.2024.3374334

RESEARCH ARTICLE

MetricSifter: Feature Reduction of Multivariate Time Series Data for Efficient Fault Localization in Cloud Applications

YUUKI TSUBOUCHI[1], (Member, IEEE), AND HIROFUMI TSURUTA[2], (Member, IEEE)
[1]SAKURA Internet Research Center, SAKURA Internet Inc., Osaka 530-0001, Japan
[2]SAKURA Internet Research Center, SAKURA Internet Inc., Fukuoka 810-0042, Japan

Corresponding author: Yuuki Tsubouchi (y-tsubouchi@sakura.ad.jp)

ABSTRACT Automated fault localization in large-scale cloud-based applications is challenging because it involves mining multivariate time series data from large volumes of operational monitoring metrics. To improve localization accuracy, automated fault localization methods incorporate feature reduction to reduce the number of monitoring metrics unrelated to a failure. However, these methods have problems with inaccuracy, either from removing too many failure-related metrics or from retaining too few failure-unrelated metrics. In this paper, we present MetricSifter, a feature reduction framework designed to accurately identify anomalous metrics caused by faults. Our framework locates a failure time window with the highest density of fault-induced change point times across monitoring metrics with a focus on their temporal proximity. Experimental results indicate that MetricSifter achieves an accuracy of 0.981, which is significantly better than the selected baseline methods. Furthermore, experiments combining various reduction methods with various localization methods demonstrate that MetricSifter improves the recall and time efficiency over the baseline methods.

INDEX TERMS AIOps, failure management, fault localization, incident response, monitoring, site reliability engineering.

I. INTRODUCTION

Online services related to social media, online gaming, e-commerce platforms, and others are increasingly integral to everyday life. The cloud applications that power these services require not only high reliability but also continuous improvement of the user experience through consistent feature updates [1]. However, the growth in the number of application components, the diversity of component types, the complexity of intercomponent dependencies, and the frequency of updates has inevitably led to reliability incidents, including outages and performance degradation [2]. Studies analyzing service provider incidents have shown that failure resolution can be a lengthy process, in some cases lasting several hours [3], [4]. Ensuring rapid failure diagnosis and mitigation is therefore critical to maintaining reliability in the face of recurrent updates.

The associate editor coordinating the review of this manuscript and approving it for publication was Yang Liu.

Engineers routinely monitor the inner workings of application systems by mining collected telemetry data, which provides insights into the causes of faults that lead to failures. In the literature, three primary types of telemetry data are distinguished: metrics, logs, and traces [5], [6]. Monitoring metrics, the most commonly accessible monitoring data, are performance metrics such as average service response time or machine CPU utilization. These monitoring metrics form a time series, and metrics are sampled at consistent intervals (e.g., every 15 seconds). Engineers build monitoring systems in advance to facilitate the instrumentation, storage, and visualization of metrics [7]. These systems allow them to perform visual inspections of extensive monitoring metrics, thereby clarifying the understanding of the operational status of the service.

Immediate fault localization through visual inspection of monitoring metrics is challenging in large-scale applications [8]. Automating the fault localization, i.e., bypassing the need for visual inspection, remains elusive for several

© 2024 The Authors. This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License.
For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

reasons. First, the large volume of anomalous metrics results from complex dependencies between components, and these dependencies allow anomalies to propagate, resulting in numerous affected metrics. Second, anomalous patterns in monitoring metrics are heterogeneous because of the variety of metric types and the specific roles of the various components, such as web servers and database servers [9]. Third, the causes of incidents within cloud applications vary widely, including internal factors such as misconfiguration, code changes, and resource contention as well as external factors such as hardware faults, insufficient resources, and excessive traffic [4]. This variety makes it difficult to predict and to detect new faults.

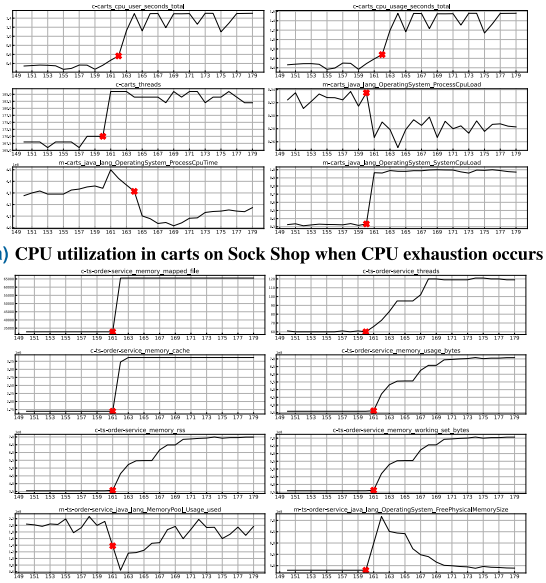
Automatic fault localization methods with statistical analysis, data mining, and machine learning for monitoring metrics have been proposed in recent years [9], [10], [11], [12], [13], [14], [15], [16], [17], [18], [19], [20], [21], [22], [23], [24]. To trace the process of anomaly propagation in application components, much of the existing research has addressed the problem of determining hidden underlying topological relations between monitoring metrics. Because this approach requires pairwise comparisons, it can quickly become computationally expensive.Wang et al. reported that as the number of monitoring metrics increases, the problem space expands, and typical fault localization methods therefore suffer from accuracy and time inefficiency [25].

Feature reduction, which is a principal approach to managing a large number of monitoring metrics, serves as a crucial preprocessing step in fault localization [25], [26]. We refer to a feature as a monitoring metric represented by a univariate time series. Fault localization methods reduce the number of metrics by selecting specific sets, either manually or automatically. Manual preselection is difficult because the most useful metrics will vary depending on the fault [17], [20]. Automated feature reduction methods can be divided into two main types: normality reduction and redundancy reduction. Normality reduction retains only those metrics identified by statistical techniques as anomalies [9], [16], [25], while redundancy reduction examines similarities (e.g., correlation) between the time series of metrics to eliminate redundancies [14], [18], [26].

Automated feature reduction methods encounter the following performance and evaluation challenges.

A. PERFORMANCE CHALLENGE

The conventional reduction methods typically over- or under-reduce the essential monitoring metrics. Redundancy reduction runs the risk of removing useful monitoring metrics from the set of root fault metrics due to their similarities. Fig. 1 illustrates cases where root fault metrics have similar time series patterns or high correlations. Conversely, normality reduction may fail to remove failure-unrelated monitoring metrics because it may incorrectly identify anomalies outside the failure time window as failure-related. Given these problems, there is a need for a more failure-oriented normality reduction method that is capable of accurately localizing the failure time window.



(a) CPU utilization in carts on Sock Shop when CPU exhaustion occurs

(b) Memory utilization in ts-order-service on Train Ticket when memory overload occurs

FIGURE 1. Examples of fault-induced change points in root fault metrics. The x-axis is the time index. These change points cluster in time because the fault-induced anomalies propagate rapidly in the faulty components. The data were samples from our experimental datasets (see Table 4). Shown here are time series of all root fault metrics when we manually injected two types of faults (CPU exhaustion and memory overload) into Sock Shop or Train Ticket, which are open-source benchmark applications. For ease of checking the change point indices, only the 150–179 range of the 0–179 x-axis is depicted.

B. EVALUATION CHALLENGE

Previous research has not established quantitative evaluation metrics for feature reduction alone. In [25], the quantitative contribution of one particular reduction method on fault localization is shown, but feature reduction itself is not evaluated. We need to make feature reduction into a quantitatively evaluable task.

In this paper, we first propose MetricSifter, a failure-oriented feature reduction framework for multivariate time series data based on unsupervised learning, to address the performance challenge. Second, to address the evaluation challenge, we quantitatively evaluate feature reduction by conventional classification evaluation metrics for a task that identifies whether each monitoring metric is failure-related or failure-unrelated. To accurately localize the failure time window, MetricSifter leverages the temporal proximity of change points induced by a fault across monitoring metrics, as shown in Fig. 1. We design the MetricSifter framework as a three-step process: offline change point detection in univariate time series data, time segmentation based on different densities of detected change points, and selection of the time segment with the highest change point density, which is estimated as the failure time window. The monitoring

metrics with change points within this time segment are identified as failure-related.

We evaluate MetricSifter through experiments on synthetic datasets for a simulation study and on real datasets from two benchmark applications, Sock Shop [27] and Train Ticket [28], [29], for an empirical study. The simulation study shows that MetricSifter achieves a classification accuracy of 0.981, outperforming the best baseline method by 0.178 on average. Experiments combining various feature reduction methods with different fault localization methods show that MetricSifter improves the recall of the top-5 root fault metrics by 0.058 to 0.224 on average over that of the baseline methods. The empirical study demonstrates that MetricSifter enhances the average top-5 recall over that of the baseline methods by 0.019 to 0.051. Moreover, MetricSifter improves the average execution time by 43.75 to 50.39% over that of the baselines with a higher improvement of top-5 recall than those without reduction.

Our contributions in this paper are summarized as follows.

- For the first time, we quantitatively evaluate different feature reduction methods on their own by formulating the feature reduction as a task of classifying whether monitoring metrics are related to a failure or not.
- We propose a feature reduction framework called MetricSifter that focuses on the proximity of change point times during a failure across monitoring metrics. Our algorithm locates a failure time window with the highest density of the change point times.
- Evaluations of MetricSifter on both synthetic and empirical datasets demonstrate its effectiveness for feature reduction in fault localization methods. Furthermore, it has the best overall impact on the accuracy and the time efficiency of fault localization compared to the baseline feature reduction methods.

Section II of this paper provides the background and our motivation. In Section III, we present the details of our feature reduction framework. In Section IV, we describe the experiments to evaluate the feature reduction performance of our framework and its impact on fault localization, and in Section V, we report the results. Section VI is a review of related work. We conclude in Section VII with a brief summary and mention of future work.

II. BACKGROUND AND MOTIVATION

A. FAULT LOCALIZATION

According to a taxonomy study for dependable computing [30], a failure and a fault are defined as follows. A **failure** is an event that occurs when an online service deviates from the correct service. This happens when a fault is propagated to the online service interface and causes the service to deviate from the correct service. The adjudged or hypothesized cause of the deviation is called a **fault**. A failure further causes an incident when it disrupts services and negatively impacts customers. We call monitoring metrics directly affected by a fault **root fault metrics**.

Fault localization is one of the processes for incident response in cloud applications. The incident response lifecycle for software system failures consists of five phases [4], [31]: (i) failure detection and incident reporting, (ii) incident triage and incident responder assignment, (iii) fault localization, (iv) failure mitigation, and (v) incident resolution. Service providers are required to immediately complete phases (i) through (iv).

B. AUTOMATED FAULT LOCALIZATION

The following are preliminaries of automated fault localization techniques based on monitoring metrics [32].

1) CATEGORIZATION

Automated fault localization is a telemetry data-driven approach that represents monitoring metrics. Metric-based fault localization methods can be divided into two main types, anomaly-degree and anomaly-propagation, which utilize fault-included anomalies on the observed time series data as the clues for localization.

(i) The anomaly-degree methods assume that monitoring metrics with a higher anomaly degree are more likely to be root fault metrics [9], [19]. With these methods, anomaly degrees are calculated for univariate time series, and metrics are ranked on the basis of these scores. Time series anomaly detection algorithms for obtaining anomaly scores have been extensively studied over the years [33], [34]. Schmidl et al. categorized such algorithms into six method types: forecasting, reconstruction, distance, encoding, distribution, and tree [33]. The existing localization approaches [9], [19] typically utilize distance or distribution methods. The distance methods compare points or subsequences using specialized distance metrics, and then identify anomalies as those with larger distances than normal subsequences. The distribution methods estimate the probability distribution of the data or fit a distribution model to the data. In this category, abnormality is judged by frequency rather than distance.

(ii) The anomaly-propagation methods localize root fault metrics by tracing the propagation of fault-induced anomalies in monitoring metrics [10], [11], [12], [13], [15], [17], [20], [21], [22], [24], [25], [26], [35]. With these methods, fault localization is attributed to a source localization problem of signal propagation in complex networks, which is a well-studied problem in the field of network science [36]. This attribution regards a source as a root fault metric and an anomaly as a signal. Ji et al.'s overview of signal propagation [36] introduced the quantification of propagating features from observational data to uncover the hidden underlying network structure. Source localization refers to finding the source of complex networks using incomplete available data based on known or discovered network structures. The methods in the anomaly-propagation family typically utilize causal discovery techniques [37], [38], which infer underlying causal relationships from observational data. Causal discover techniques can be divided into four

main types: constraint-based, score-based, those exploiting structural asymmetries, and those exploiting various forms of intervention [38]. Most fault localization methods [11], [17], [20], [21], [22], [35] adopt either an original or modified form of the PC algorithm [39], which is a typical constraint-based method concerned with conditional independence relationships and is valued as a nonparametric, scalable, and efficient algorithm for causal discovery [37]. The resulting causality graph is essentially built on the basis of the correlation between monitoring metrics. Some fault localization methods enrich the resulting graph with additional information, such as the time series correlation [20] or anomaly score [21], [22]. The most common approach to determining the possible root fault metrics is to visit the resulting graph through traversal algorithms, such as a breadth-first search, random walk, or PageRank [40]. The traversability can exclude anomaly metrics that have no path from the frontend anomalous node to the root fault metrics, i.e., those that are unrelated to the failure.

2) STARTING LOCALIZATION TRIGGERED BY DETECTING SERVICE LEVEL REDUCTION

The learning process in fault localization starts automatically at failure detection. Most fault localization methods detect failures by identifying anomalies in the times series for service level indicator (SLI) metrics [1], which measure the level of service provided to users [9], [15], [17], [20], [21], [22]. SLI metrics include response latency, error rates, and throughput. The fault localization process is initiated when SLI metrics violate the predefined normal operating conditions of the service.

3) LOOKUP TIME WINDOW SIZE

Most fault localization methods look back at a fixed time window from the failure detection time. These methods estimate the maximum possible time for a failure to occur and then input time series data within that range. Research [18] has shown that for a particular bank in China, 80% of 82 failures over a year were detected within nine minutes, with the longest taking 19 minutes. Some methods [9], [11], [12], [15], [16], [18], [19], [20] require normal time series data outside the failure time window and set a lookback window that includes this normal time window.

4) FAST LOCALIZATION PROCESSING

The conventional fault localization methods typically learn from monitoring metric data for each failure and then suggest potential fault locations. Because this learning process begins after failure detection, rapid processing is essential. This requirement arises from the scarcity of past failures in the target system.

C. PROBLEM FORMULATION

Fig. 2 illustrates how the set of monitoring metrics M at the time of a failure is categorized into three subsets: root fault metrics (M^A), metrics involved in anomaly propagation (M^B), and metrics unrelated to the fault (M^C). This classification is consistent with the fault-error-failure cycle [30], a model that describes the transition from faults to errors, ending with failures. Fig. 2 shows the process of anomaly propagation from a fault to a failure in the service. Anomaly propagation occurs through network communication or shared resources. Eventually, these anomalies affect SLI metrics, resulting in failures that negatively impact service levels. In this context, the fault localization task is defined as identifying M^A from M.

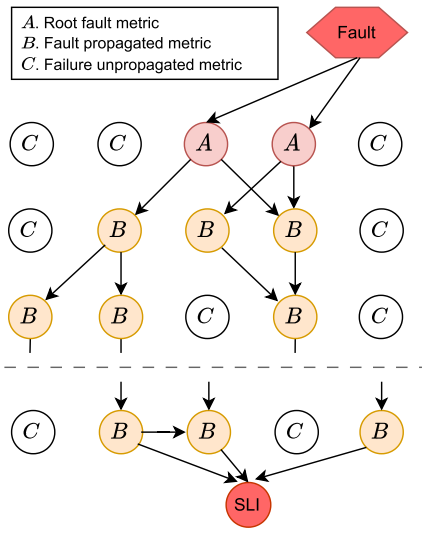


FIGURE 2. Three types of monitoring metrics on anomaly propagation for a failure. When a fault occurs, the fault-induced anomalies propagate in the system. These anomalies appear as changes in the time series of some monitoring metrics (labeled A and B), whereas other metrics (labeled C) do not have changes. Metrics labeled A are ones that directly indicate the fault, and metrics labeled B are ones that are indirectly involved in anomaly propagation. Service Level Indicator (SLI) metrics, which are one of the metrics labeled B, measure the level of service provided to users.

With feature reduction, we minimize the number of elements of M^C, which consists of metrics that either have no anomalies or have anomalies outside the failure. Removing M^C during the initial stages of localization is beneficial, as M^C can introduce noise into the fault localization process.

We write a multivariate time series at the failure detection time t as $\boldsymbol{X}_t = [\boldsymbol{x}_t^1, \boldsymbol{x}_t^2, \ldots, \boldsymbol{x}_t^{n_{\text{metric}}}]$, where $\boldsymbol{x}_t^i = [x_{t-w+1}^i, x_{t-w+2}^i, \ldots, x_t^i]$ is the univariate time series of the i^{th} monitoring metric m_i, n_{metric} is the number of metrics and w is the observation window size. Table 1 summarizes the key notations used in this paper.

Our main objective is, **once the monitoring system detects a failure, to identify the possible set of monitoring metrics $M^A \cup M^B$ from $M^A \cup M^B \cup M^C$ based on previously collected multivariate data $\boldsymbol{X}_t$ to localize M^A as soon as possible**.

D. FEATURE REDUCTION AND ITS PERFORMANCE CHALLENGES

Table 2 presents the comparison of existing feature reduction methods and MetricSifter in fault localization research based

TABLE 1. Notations.

Notation	Definition
n_{metric}	Number of monitoring metrics
w	Size of observation lookback window
$M = \{m_i \mid i = 1, 2, ...n_{\text{metric}}\}$	Metrics set of all observed metrics
$\boldsymbol{X}_t = [\boldsymbol{x}_t^1, \boldsymbol{x}_t^2, ..., \boldsymbol{x}_t^{n_{\text{metric}}}]$	Matrix of multivariate time series
$\boldsymbol{x}_t^i = [x_{t-w+1}^i, x_{t-w+2}^i, ..., x_t^i]$	Vector of univariate time series uniquely corresponding to a monitoring metric
$\mathbb{P}$	Set of vectors of change point indices uniquely corresponding to a monitoring metric
$\boldsymbol{p}^{1d} = \{p_r^{1d} \mid r = 1, 2, ...\}$	One-dimensional vector of change point indices flattened from $\mathbb{P}$
$\mathbb{S} = \{S_j \mid j = 1, 2, ...\}$	Set of segments
S_j	Segment, which is set of change point indices

$|\boldsymbol{z}|$ and $|Z|$ means the size of a vector $\boldsymbol{z}$ and a set Z, respectively.

TABLE 2. Comparison of existing works and MetricSifter.

Name	Reference	Reduction type	Data characteristic	Learning type	Multivariate analysis	Technique
FluxInfer-AD	IPCCC'20 [16]	Normality	Anomalies	Unsupervised		Distribution-based anomaly detection
BIRCH	ACSOS'21 [25]	Normality	Anomalies	Semi-supervised	✓	Distance-based anomaly detection
K-S test	ISSRE'21 [9]	Normality	Anomalies	Semi-supervised		Distribution-based anomaly detection
NSigma	CCGrid'22 [12]	Normality	Anomalies	Semi-supervised		Distribution-based anomaly detection
PairCorr	INFOCOM'14 [23]	Redundancy	Correlation similarities	Unsupervised	✓	Pairwise correlation
k-Shape	Middleware'17 [26]	Redundancy	Shape similarities	Unsupervised	✓	Partitioning-based clustering
HDBS+SBD	ICWS'22 [14]	Redundancy	Shape similarities	Unsupervised	✓	Density-based clustering
MetricSifter (ours)	N/A	Normality	Change points and their time density	Unsupervised	✓	Change point detection and segmentation

"Name" denotes the method name used in the original paper. "Reduction type" denotes the coarse typing of the reduction method. The "normality" reduction type removes monitoring metrics without anomalies, and the "redundancy" reduction type removes similar metrics with high time series similarity. "Data characteristic" denotes time series data characteristics used as clues for reduction. "Learning type" denotes the typing of necessity of training data. "Semi-supervised" method trains subseries in a normal time window. "Multivariate analysis" denotes whether the method considers other metrics for the reduction of one metric. MetricSifter is the only unsupervised method of the normality reduction type with multivariate analysis. "Technique" denotes the technical algorithm utilized in the method. Each existing normality reduction method uses either a distribution-based or distance-based algorithm in six categories of Time series anomaly detection algorithms [33] (presented in Section II-B), while each existing redundancy reduction method uses either a partitioning-based or density-based clustering algorithm in categories of time series clustering algorithms [41].

on monitoring metrics. These methods are categorized into two types: normality reduction, which involves reducing monitoring metrics without anomalies, and redundancy reduction, which involves reducing redundant monitoring metrics based on data similarities.

1) NORMALITY REDUCTION

As shown in Table 2, existing normality reduction methods focus on anomalies in univariate time series data as data characteristics using time series anomaly detection algorithms [33]. Because semi-supervised methods are required to train subseries in a normal time window, they split each univariate time series into normal and anomalous subseries based on a fixed window size. Existing methods except BIRCH in pursuit of efficiency.

BIRCH [25] utilizes BIRCH clustering [42] to cluster multivariate time series inside a normal time window. Specifically, it identifies each univariate time series as anomalous if its data during an anomalous time window is distant from any cluster. The K-S test [9], a two-sample Kolmogorov-Smirnov test (K-S test) [43], is used to detect differences between the probability distributions of these two subseries. NSigma uses the n-sigma rule of z score normalization [12]. The z score transforms the data into a standard normal distribution by using the mean and the standard deviation. FluxInfer-AD [16] smooths each univariate time series while preserving anomalies with clustering based on Gaussian mixture models [44] and then applies the three-sigma rule for anomaly detection.

2) REDUNDANCY REDUCTION

Most redundancy reduction methods are based on time series clustering, which groups together homogenous time series based on a certain similarity measure. Time series

clustering is a well-studied problem in the field of data mining [41]. According to [41], it can be divided into four components: representation, similarity or distance measurement, clustering algorithm, and clustering prototype. Redundancy reduction methods typically use raw time series as the representation and use the medoid, defined as the metric with the minimum sum of distances to others in the cluster, as the prototype. In these methods, either the Pearson correlation coefficient or shape-based distance (SBD) is typically utilized as a distance measurement. SBD measures the shape similarity between two time series by aligning their shapes, regardless of their amplitude and phase differences [45]. Clustering algorithms are generally classified into six groups: partitioning, hierarchical, grid-based, model-based, density-based clustering, and multi-step clustering algorithms [41]. Redundancy reduction methods typically use partitioning or density-based clustering.

Sieve [26] and TS-InvarNet [14] learn metric relationships during normal operation and generate online predictions by using trained models. Sieve utilizes k-Shape [45], which is a partitioning clustering algorithm that uses SBD as a distance measurement. One of the most famous density-based algorithms is DBSCAN, where a cluster is expanded if its neighbors are dense [46]. TS-InvarNet utilizes HDBS+SBD with SBD and HDBSCAN [47], an advanced algorithm that extends DBSCAN by transforming it into a hierarchical algorithm to be more adaptive to varying density clusters without the need for a predetermined main parameter. PairCorr, instead of using time series clustering, iteratively removes a monitoring metric from a pair if its Pearson correlation coefficient exceeds a predefined threshold [22].

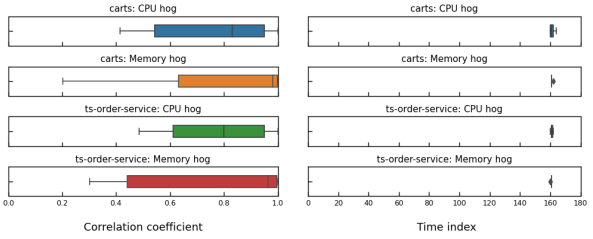
3) PERFORMANCE CHALLENGES

In practice, both normality and redundancy reduction methods still suffer from the following limitations in terms of accuracy performance.

False Positives: Normality reduction methods can lead to false positives because anomalies are also common when the system is running normally. An example of these anomalies is the periodic spikes caused by routine events such as batch job executions. These false positives prevent the removal of some monitoring metrics from M^C, which can lead to additional computation costs in the fault localization algorithm.

False Negatives: In contrast, redundancy reduction methods can lead to false negatives because sets of M^A and M^B are sometimes similar to other metrics, as shown in Fig. 1. For example, the shapes of the time series data for the number of transmitted network packets and the size of transmitted network data are often similar. Removing some monitoring metrics in $M^A \cup M^B$ can lead to a lower localization accuracy.

Because feature reduction should have minimal false negatives that erroneously remove root fault metrics, we adopt an approach to develop a normality reduction method with lower false positives. False positives can be reduced if we localize the failure time window by considering multivariate time series data characteristics, rather than arbitrarily predetermining the normal time window. In this paper, we design MetricSifter as a new framework for normality reduction in the unsupervised learning type with multivariate analysis. This approach is unique among the existing works, as shown in Table 2.



(a) Correlation coefficient (b) Fault-caused change point time

FIGURE 3. Pairwise analysis between root fault metrics. The variance of the fault-caused change point times (b) is significantly smaller than the variance of the Pearson correlation coefficients (b). Thus, the more appropriate feature for identifying root fault metrics is the fault-caused change point time. These metrics are sampled in the datasets $\mathcal{SS}$-middleware and $\mathcal{TT}$-middleware in Table 4. The change points in (b) are manually labeled by a domain expert. The horizontal axis in (b) is the time index that spans the entire observation time window. The observation window in (a) is the same as in (b). The vertical axis in (a) is continuous, and that in (b) is integers only.

III. FEATURE REDUCTION FRAMEWORK

A. KEY INSIGHTS FROM OBSERVATION

We present *fault-caused change point times* as data characteristics useful for localizing failure time window through the following observations.

Fig. 3 shows a pairwise analysis of the absolute value of the Pearson correlation coefficient and the fault-caused change point time for $M^A \cup M^B$. The absolute value of the Pearson correlation coefficient has been used as a similarity measure for monitoring metrics in previous studies [18], [20], [23]. Fig. 3(b) shows that the fault-caused change point time typically clusters around index 160, which corresponds to the fault time. Fig. 3(a) shows a wide distribution of the Pearson correlation coefficient, ranging from 0.4 to 1.0. These observations support the use of change point time as a common feature for each monitoring metric in $M^A \cup M^B$.

This observation is supported by a related field study [9], in which 13 anomalous patterns were identified in the monitoring metrics of M^A. Low correlations are found between certain pattern pairs such as spike and level shift.

The metrics in M^C either have no change points or have change points outside the failure time window. Thus, we can assume that $M^A \cup M^B$ falls within the time range where the change point times are most densely distributed. The highest density time range is estimated as the failure time window.

To identify fault-caused change point times, we need to resolve the following two problems. The first problem is offline change point detection for each univariate time series data. Change point detection is similar to anomaly detection, but anomaly detection focuses on identifying

unusual data points whereas change point detection is about identifying shifts in the statistical properties over time. Adopting change point detection effectively resolves the false positives discussed in Section II-D3 because it does not require predeterminating the normal time window. The second problem is change point time segmentation based on the distribution density of the detected change point times. The segmentation requires estimating the probability distribution of the change point times and then finding the segment boundaries to locate relatively higher density ranges of the distribution.

B. FRAMEWORK OVERVIEW

We design the MetricSifter framework to combine change point detection and segmentation to address the performance challenges based on the key insights presented in Section III-A. The MetricSifter framework consists of four sequential steps (STEP 0 to STEP 3) that produce the final output. Fig. 4 presents an overview of MetricSifter, and Fig. 5 exemplifies the critical components in STEPS 1 to 3. The steps are as follows.

STEP 0 Simple Filter: This preprocessing step filters out monitoring metrics represented by univariate time series that show no variation. The objective here is to reduce the computational cost of STEP 1. STEP 0 removes monitoring metrics corresponding to time series where all data points have equal values or where all values in the first-order difference series are equal.

STEP 1 (Change Point Detection): STEP 1 detects change points in each univariate time series data that passes the STEP 0 filter. Monitoring metrics with time series that have zero change points are further removed in this step.

STEP 2 (Change Point Segmentation): STEP 2 separates the change point indices into multiple segments based on their distribution density for each monitoring metric detected in STEP 1. The indices of segment boundaries are at the relative minima of the density estimator.

STEP 3 (Select the Largest Segment): STEP 3 selects the segment with the largest number of members from the obtained segments. The final output of MetricSifter is then the monitoring metrics with change points inside the selected segment.

MetricSifter operates independently on each set of metrics observed for the application components during STEPS 2 and 3. The output generated by STEP 3 is then used as input to the fault localization algorithm.

C. CHANGE POINTS DETECTION FOR UNIVARIATE TIME SERIES DATA

After running STEP0, STEP1 detects change points in the univariate time series data for all remaining monitoring metrics. A change point detection algorithm with low computational complexity is essential for fast processing of many monitoring metrics.

According to [48], change point detection algorithms are generally expressed as a combination of the following three elements: a cost function, a search method, and a constraint. A cost function measures homogeneity; choosing the cost function is equivalent to choosing the type of change to detect. A small value of the cost function indicates the absence of change points, whereas a larger value indicates their presence. A search method solves a discrete optimization problem on the set of change points $\mathcal{T}$ to minimize the sum of the cost function values. A constraint is put on the number of change points. If the number of change points is unknown, the optimization function introduces a penalty term $\mathrm{pen}(\mathcal{T})$ for model complexity to prevent detecting too many change points owing to overfitting. A smaller penalty detects more change points, whereas a larger penalty detects fewer or no change points.

We opted to use the mean shift model as the cost function because it has lower computational complexity than that of linear regression or autoregression [48]. In the mean shift model, it is assumed that the time series distribution is Gaussian with fixed variance. The cost function $c(\boldsymbol{x}_{\mathrm{sub}})$ is expressed as follows

$$c(\boldsymbol{x}_{\mathrm{sub}}) = \sum_{k \in I} \|x_{\mathrm{sub},k} - \bar{\boldsymbol{x}}_{\mathrm{sub}}\|_2^2 \tag{1}$$

where I is the interval of the subseries indices and $\bar{\boldsymbol{x}}_{\mathrm{sub}}$ is the sample mean of the subseries. The computational complexity of the mean shift model is $\mathcal{O}(T)$, where T is the length of the time series.

For the search method, we opted to use pruned exact linear time (Pelt) [48], [49]. Pelt significantly reduces the computational complexity of the optimization problem by applying a pruning rule when the penalty term is a linear function of the number of change points. This pruning rule allows potential change points to be neither discarded nor retained from the set of data points. Assuming that partition lengths are randomly drawn from a uniform distribution, the average computational cost of Pelt is $\mathcal{O}(C_{\mathrm{cost}}\, T\, |\mathcal{T}|)$, where T is the length of the time series, and C_{cost} is the computational cost when the cost function is called for a subseries [48].

We need to use the penalty term as the constraint because the number of change points in $\boldsymbol{x}_t^i$ is unknown in our scenario. To calibrate a linear penalty coefficient according to $\boldsymbol{x}_t^i$, we use a heuristic based on the Bayesian information criterion (BIC), which is a well-known criterion for model selection [48]. We also fit a penalty weight constant parameter (ω) to BIC as a domain-specific adjustment. The penalty function $\mathrm{pen}(\mathcal{T})$ is given by

$$\mathrm{pen}(\mathcal{T}) = \omega\, \sigma^2 \log T\ |\mathcal{T}|, \tag{2}$$

where T is the length of the time series [48]. We set $\omega = 2.5$, which gives the best accuracy from the upcoming parameter sensitivity study in Section IV-B5. If no change points are found within the constraint, the corresponding metric is removed.

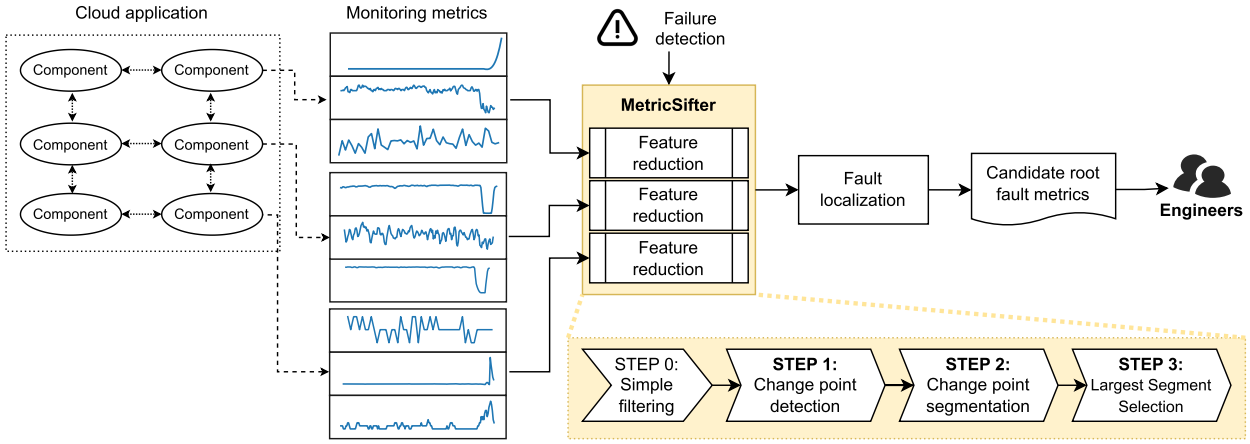


FIGURE 4. Overview of MetricSifter. Automated fault localization is a step-by-step approach that monitors various components of a cloud application, detects failures through anomalies in monitoring metrics such as SLI, reduces the complexity of the data, and pinpoints the potential faults. The final output is a set of metrics that suggest probable fault sources, which are then examined by engineers to address the problems. Feature reduction is an intermediate process of detecting a failure and localizing a fault. The MetricSifter process as feature reduction is further broken down into initial simple filtering, change point detection, segmentation of these change point times, and selection of the most significant segment likely to contain the fault.

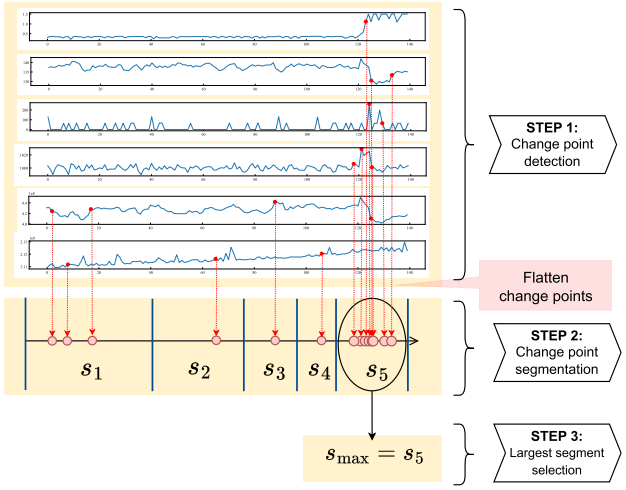


FIGURE 5. An example of feature reduction using the MetricSifter framework. STEP 1 finds one or more change points in each of the six time series data. STEP 2 first flattens the change point indices into a one-dimensional array of length 15 and then segments them into five segments (s_1 through s_5) based on their density. STEP 3 selects the largest segment, which is s_5 in this example. Even if STEP 1 erroneously detects change points outside the failure time window (such as the first and second time series from the bottom), STEPS 2 and 3 can select the accurate segment.

D. SEGMENTATION OF CHANGE POINT TIMES

Algorithm 1 outlines the procedure for segmenting change point indices. The input to Algorithm 1 is the set of change point indices $\mathbb{P}$ of unequal length obtained in Section III-C. First, $\mathbb{P}$ is transformed into a one-dimensional vector $\boldsymbol{p}^{1d}$ (line 1). Second, a probability density function (PDF) is estimated to fit $\boldsymbol{p}^{1d}$ (line 2). Finally, $\boldsymbol{p}^{1d}$ is segmented by using the relative minimum indices of the density estimates as the segment boundaries (lines 3 – 11).

Algorithm 1 Algorithm of Segmentation of Change Point Times

Input: Change points indices $\mathbb{P}$
Output: Segments $\mathbb{S}$

1: $\boldsymbol{p}^{1d} = \{p_r^{1d} \mid r = 1, 2, \ldots\} \leftarrow$ flatten $\mathbb{P}$
2: $\widehat{p}_n(x) \leftarrow$ learn PDF according to (3) from $\boldsymbol{p}^{1d}$
3: $\boldsymbol{w} \leftarrow [1, 2, \ldots, w]$
4: $\boldsymbol{e} = \{e_i \mid i = 1, 2, \ldots, w\} \leftarrow \widehat{p}_n(\boldsymbol{w})$
5: $\boldsymbol{i}^* = \{i_j^* \mid j = 1, 2, \ldots\} \leftarrow$ calculate relative minimum indices of $\boldsymbol{e}$ according to (4)
6: $\mathbb{S} = \{S_j \mid j = 1, 2, \ldots, |\boldsymbol{i}^*| + 1\} \leftarrow$ create empty $|\boldsymbol{i}^*| + 1$ segments
7: $S_1 \leftarrow \left\{p_r^{1d} \mid p_r^{1d} < \boldsymbol{w}[i_1^*],\ 1 \le r \le |\boldsymbol{p}^{1d}|\right\}$
8: **for all** $i_j^* \in \boldsymbol{i}^*$ **do**
9: $\quad S_j \leftarrow \left\{p_r^{1d} \mid \boldsymbol{w}[i_j^*] \le p_r^{1d} \le \boldsymbol{w}[i_j^* + 1],\ 1 \le r \le |\boldsymbol{p}^{1d}|\right\}$
10: **end for**
11: $S_{|S|} \leftarrow \left\{p_r^{1d} \mid p_r^{1d} \ge \boldsymbol{w}[i_{|\boldsymbol{i}^*|}^*],\ 1 \le r \le |\boldsymbol{p}^{1d}|\right\}$

To estimate the probability distribution of the flattened change point indices $\boldsymbol{p}^{1d}$, we use kernel density estimation (KDE) [50], one of the most well-known approaches to estimate the underlying PDF. KDE is nonparametric and learns the density shape directly from the data. Let $[x_1, x_2, \ldots, x_n]$ be n data points; then, KDE is given by

$$\widehat{p}_n(x) = \frac{1}{nh} \sum_{i=1}^{n} K\left(\frac{x - x_i}{h}\right), \tag{3}$$

where K is a smooth function (called the kernel function) and $h > 0$ is the smoothing bandwidth that controls the amount of smoothing.

Intuitively, KDE projects each data point into a smooth "bump" whose shape is defined by K. The sum of these

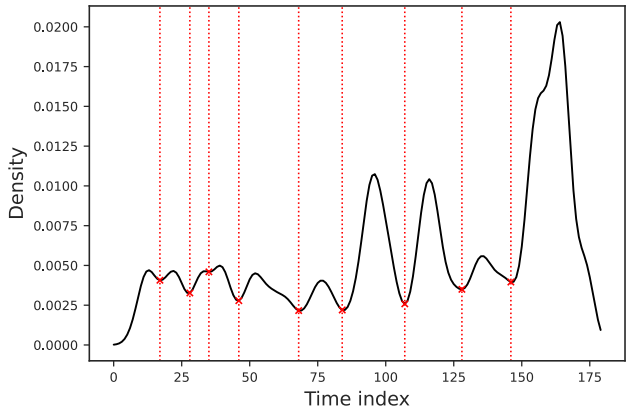


FIGURE 6. An example of segmentation based on the density distribution of change points in STEP 2. The × markers are plots of relative minima, and the vertical lines are segment boundaries. In this example, the density distribution is separated into ten segments.

bumps is the overall density estimate. High-density areas have larger density values due to overlapping bumps, whereas sparse areas have lower values. Since the kernel function mainly affects the estimation error by a constant shift and the choice of kernel function has only minimal impact, a standard Gaussian distribution is common. The bandwidth h strongly influences the estimation error [51]; however, our dataset shows a consistent performance regardless of h, as indicated by the parameter sensitivity study in Section IV-B5. We set $h = 3.5$, which provides the best accuracy in the parameter sensitivity study shown in Section IV-B5.

We obtain the discrete values $\boldsymbol{e}$ from the PDF $\widehat{p}_n(x)$ learned by $\boldsymbol{p}^{1d}$, where x is a vector of elements incremented by 1 from 1 to w.

Given $\boldsymbol{e}$, the relative minimum indices as the segment boundaries $\boldsymbol{i}^*$ are given by

$$\boldsymbol{i}^* = \{i \mid (e_i < e_{i-1}) \wedge (e_i < e_{i+1}),\ 1 < i < |\boldsymbol{e}|\}. \quad (4)$$

Since the chosen kernel function is a Gaussian-based smooth function, the relative minima are obtained as points that are preceded and succeeded by points with higher values.

Finally, the segment set $\mathbb{S}$ is obtained from $\boldsymbol{p}^{1d}$ separated by the boundaries $\boldsymbol{i}^*$. Fig. 6 illustrates the segmentation process when using change point indices from a subset of the dataset prepared for the upcoming empirical study in Section IV-C. This example shows the partitioning of the data into ten segments. The last segment, which has the highest density, contains a fault injection time at index 160.

E. SELECTION OF THE LARGEST SEGMENTS

The largest segment, $S_{\max}$, is selected from the segment set $\mathbb{S}$ obtained in Section III-D. The size of each segment is quantified by the sum of the reciprocals of the number of change points in the monitoring metrics. This segment sizing avoids overestimating the size of segments outside the failure time window.

For any given S_j in $\mathbb{S}$, let $M(S)$ be the function that returns the set of each monitoring metric with the change point indices in a given segment S. Let $N(m)$ be the function that returns the number of change point indices found in a given monitoring metric m. $S_{\max}$ is given by

$$S_{\max} = \underset{S_j \in \mathbb{S}}{\arg\max} \sum_{m \in M(S_j)} \frac{1}{N(m)}.$$

The set of monitoring metrics linked to the change point indices in $S_{\max}$ is the result produced by MetricSifter.

IV. EXPERIMENT

In this section, we evaluate the performance of the various feature reduction methods through two experiments: a simulation study based on synthetic datasets and an empirical study based on practical datasets collected from realistic mock cloud applications. The datasets and implementation can be found in our GitHub repository [52].

A. EXPERIMENTAL SETUP

In this section, we describe the experimental settings common to both the simulation and the empirical study.

1) BASELINE METHODS

We compare MetricSifter with the following representative methods from each type of normality reduction and redundancy reduction introduced in Section II-D as baselines.

For the normality reduction type, we selected NSigma, BIRCH, K-S test, and FluxInfer-AD.

For the redundancy reduction type, we selected HDBSCAN with SBD (HDBS-SBD) and HDBSCAN with Pearson correlation (HDBS-R), as redundancy reduction methods typically utilize SBD or Pearson correlation to measure similarity and k-Shape or HDBSCAN for clustering. k-Shape clustering is a method of partitional optimization clustering that repeatedly refines the clustering to determine the optimal number of clusters. However, k-Shape is not included in the baselines given the significant execution time due to its iterative nature and the impracticality for online incident response. Redundancy reduction methods are executed separately for each component since the computational complexity of clustering scales quadratically with the number of items.

An "Ideal" method, theoretically tuned to achieve 100% accuracy by using a ground truth, removes only all M^C, failure-unrelated monitoring metrics.

2) FAULT LOCALIZATION METHODS

We evaluate the accuracy of fault localization by comparing the various feature reduction methods and by combining them with different fault localization methods.

Random Selection (RS) serves as a benchmark, simulating the random selection in which engineers, in the absence of system-specific domain knowledge, continue to select monitoring metrics one by one until the root fault metrics are found.

For our experiment, we selected representative methods from each type of the anomaly-degree and anomaly-propagation methods described in Section II-A. We selected ϵ-Diagnosis from the anomaly-degree family. Most methods in the anomaly-propagation family include two steps: graph construction and scoring. For the graph construction step, we selected Call Graph (CG), which uses a static call graph to represent network interactions between components, PC [39], a constraint-based method, DirectLiNGAM [53] (LiNGAM), a function-based technique, and RCD [11], an intervention-based method. Inspired by [15], [20], and [21], PC and LiNGAM improve efficiency by using CG as prior knowledge to streamline graph construction. For the scoring step, we categorize the methods into random-walk-based and regression-based methods. We selected PageRank [40] as a random-walk-based method, following implementations in [15], [16], [24], and [25], and HT [13], which is the only regression-based approach. As an exception, RCD does not have a separate scoring phase because it treats the failure as an intervention in the causal structure graph on the root fault metrics. Combinations of graph building and scoring techniques are referred to as PC+PageRank, LiNGAM+HT, and similar expressions.

3) IMPLEMENTATION

a: FEATURE REDUCTION

We utilized Ruptures [48] for change point detection in MetricSifter, Statsmodels [54] for KDE, SciPy [55] for the K-S test and Pearson correlation, Scikit-Learn [56] for BIRCH, a Gaussian mixture model, DBSCAN, and hdbscan [57] for HSDBSCAN, and the authors' implementation for SBD.

b: FAULT LOCALIZATION

We used PyRCA [58], which is the latest fault localization library, for ϵ-Diagnosis, PC, DirectLiNGAM, HT, and RCD, and Networkx [59] for PageRank. RCD is implemented with a random processing order for monitoring metrics when constructing the causal graph, and in our approach, we averaged the results of 100 RCD models with different random seeds for its final prediction.

The hyperparameters of the methods utilized in our experiment can be found in our repository [52]. In principle, we adopted the settings described in each original paper; and for those not described, we adopted the default values from the libraries.

The execution granularity in STEPS 2 and 3 of MetricSifter is system-wide in the simulation study (Section IV-B) and microservice-level in the empirical study (Section IV-C).

We ran the feature reduction and fault localization processes on a virtual instance within SAKURA Cloud[1] with an Intel Xeon Gold 6212U 2.40 GHz CPU (20 cores) and 96 GiB RAM.

[1] https://cloud.sakura.ad.jp/

4) EVALUATION METRICS

We adopted different evaluation metrics for each stage of feature reduction and fault localization.

a: FEATURE REDUCTION

Feature reduction is a task to classify $M^A \cup M^B$, which is failure-related, and M^C, which is failure-unrelated, as explained in Section II-C. Thus, we adopted three metrics to evaluate feature reduction alone: specificity, recall, and balanced accuracy (BA). These are known as evaluation metrics of classification problems in the simulation study. In the simulation study, the labels A, B, and C were provided according to the rules of the data generation, as shown in Fig. 2. Specificity, recall and BA are given by

$$\begin{aligned} \text{Specificity} &= |\hat{M^C} \cap M^C| \,/\, |M^C| \\ \text{Recall} &= \frac{|(\hat{M^A} \cup \hat{M^B}) \cap (M^A \cup M^B)|}{|M^A \cup M^B|} \\ \text{BA} &= (\text{Specificity} + \text{Recall}) \,/\, 2, \end{aligned} \tag{5}$$

where $(\hat{M^A} \cup \hat{M^B})$ and $\hat{M^C}$ are the predicted sets of monitoring metrics by feature reduction. High specificity means that many failure-unrelated metrics are reduced while preserving the root fault metrics. High recall indicates that a significant number of root fault metrics are retained.

The empirical datasets require manual labeling because of the lack of rules. Since manual labeling is impractical for the large numbers of M^B and M^C, we manually labeled only M^A. Therefore, we utilized three evaluation metrics that can only be defined by the M^A label: reduction rate (RR), recall of M^A (Recall of Root Fault, RF), and proportion of M^A in the reduced set (Proportion of Root Fault, PF). RR, RF, and PF are given by

$$\begin{aligned} RR &= |\hat{M}_{\text{reduced}}| \,/\, |\hat{M}_{\text{step0}}| \\ RF &= |\hat{M^A}| \,/\, |M^A| \\ PF &= |\hat{M^A}| \,/\, |\hat{M}_{\text{reduced}}|, \end{aligned} \tag{6}$$

where $\hat{M}_{\text{reduced}}$ is the predicted set of monitoring metrics reduced by feature reduction and $\hat{M}_{\text{step0}}$ is the predicted set of monitoring metrics reduced by STEP0 in MetricSifter. Applying STEP0 to the baselines achieves a fair comparison with MetricSifter. PF directly indicates the size of the problem space addressed by fault localization.

Validating these metrics for evaluating feature reduction is crucial, since this paper is the first to evaluate feature reduction based on these metrics. In Section IV-B4 we evaluate whether high feature reduction metrics actually correlate with high fault localization metrics.

b: FAULT LOCALIZATION

Following evaluations from prior studies [10], [15], [20], [21], [35], fault localization performance is measured by recall of the top-k results (AC@K) and average recall (AVG@K). AC@K assesses the probability that the top-k

results contain the true root fault metrics. A higher AC@K for smaller values of k means more accurate fault localization.

AC@K for a given set of faults F is defined as

$$AC@k = \frac{1}{|F|} \sum_{f \in F} \frac{\sum_{i<k} R[i] \in V_{rc}}{\min(k, |V_{rc}|)}, \tag{7}$$

where $R[i]$ symbolizes each ranked root fault metric and V_{rc} is the set of root fault metrics. AVG@k evaluates the overall performance by averaging AC@k as follows:

$$AVG@k = \frac{1}{k} \sum_{1 \le j \le k} AC@j. \tag{8}$$

We adopted AVG@K for $k \le K = 5$ because 73% of developers expect fault localization within the top 5, as shown by surveys [60].

B. SIMULATION STUDY

1) DATA GENERATION

To generate synthetic datasets, we adopted the data generator from PyRCA, which generates multivariate time series data and a directed acyclic graph (DAG). Each node of the DAG represents the univariate time series of a monitoring metric, where edges indicate anomaly propagation directions. We generated four synthetic datasets: 50 nodes with 100 edges, 50 nodes with 200 edges, 100 nodes with 500 edges, and 100 nodes with 700 edges, denoted as $D_{\text{Sim}}^{N,E}$ (where N equals nodes and E equals edges). Each $D_{\text{Sim}}^{N,E}$ dataset contains different combinations of data generation parameters and five failures for each parameter set.

The data generator creates a random DAG that satisfies the desired number of nodes and edges. Anomalies in the DAG nodes propagate from child to parent. Normal state data precedes anomalous state data, the latter generated by introducing perturbations in selected metrics. Each generated time series contains 160 points representing the normal time window and 20 for the anomalous time window. The generation parameters are a set of anomaly types, noise types, and noise weights. More details of the data generation are provided in Appendix.

2) PERFORMANCE OF FEATURE REDUCTION

Fig. 7 shows the performance comparison of the various feature reduction methods on synthetic datasets, where 7(a), 7(b), and 7(c) illustrate the classification effectiveness for each dataset and 7(d), 7(e), and 7(f) show the balanced accuracy (BA) over different anomaly and noise types and noise weight parameters. MetricSifter outperforms the baseline methods on BA of 0.981 on average, the overarching metric of classification effectiveness. HDBS-SBD and HDBS-R, redundancy reduction methods, have lower BA scores because they erroneously remove $M^A \cup M^B$. This reduction error stems from the similarities between the time series of $M^A \cup M^B$, as highlighted in Fig. 3. However, NSigma and BIRCH, which are conservative approaches that minimize feature reduction, show higher recall rates than MetricSifter.

TABLE 3. R^2 between the reduction performance metrics and localization performance metric.

Localization method	BA	Recall	Specificity
RS	0.374	0.134	0.143
ϵ-Diagnosis	0.000	0.028	0.031
PC+PageRank	0.413	0.258	0.064
PC+HT	0.139	0.187	0.000
LiNGAM+PageRank	0.510	0.178	0.202
LiNGAM+HT	0.424	0.484	0.002
RCD	0.148	0.222	0.002

Each value is the coefficient of determination R^2 between the reduction performance metrics (BA, Recall, Specificity) and localization performance metric (AVG@5). R^2 is a measure of how well the reduction performance metrics explain the localization performance metric. R^2 ranges from 0 to 1, where 1 indicates that the reduction performance metrics perfectly explain the localization performance metric. Each performance value originates from Section IV-B2 and IV-B3. BA accounts for 13.9% to 51.0% of the variability of AVG@5, excluding the results of ϵ-Diagnosis. The reduction performance partially explains AVG@5 of the corresponding localization methods.

3) PERFORMANCE OF FAULT LOCALIZATION WITH FEATURE REDUCTION

Fig. 8 displays the contribution of the feature reduction methods to the accuracy of fault localization on the synthetic datasets. The average AVG@5 of MetricSifter is the best performance among all baselines except Ideal. The increase of the average AVG@5 of MetricSifter over the best baseline NSigma is 0.074 and the increase over the worst baseline HDBS-SBD is 0.240. However, the average AVG@5 of MetricSifter does not reach that of Ideal, and the difference is 0.029.

The difference of the average AVG@5 compared to that of ''None'' ranges from –0.101 (HDBS-SBD) to 0.167 (Ideal). The normality reduction methods (except FluxInferAD) have a higher average AVG@5 than that of ''None''. However, the redundancy reduction methods (HDBS-R and HDBS-SBD) have a lower average AVG@5 than that of ''None''. Therefore, the redundancy reduction methods are not useful in terms of localization accuracy.

These results suggest that addressing the performance challenges presented in Section II-D contributes to fault localization.

4) VALIDITY OF FEATURE REDUCTION ACCURACY METRIC

To validate our chosen evaluation metrics for feature reduction, we analyze the coefficient of determination R^2 [61] between feature reduction accuracy and fault localization recall. Table 3 shows that BA accounts for 13.9% to 51.0% of the variability of AVG@5, excluding the results of ϵ-Diagnosis. Therefore, the reduction performance partially

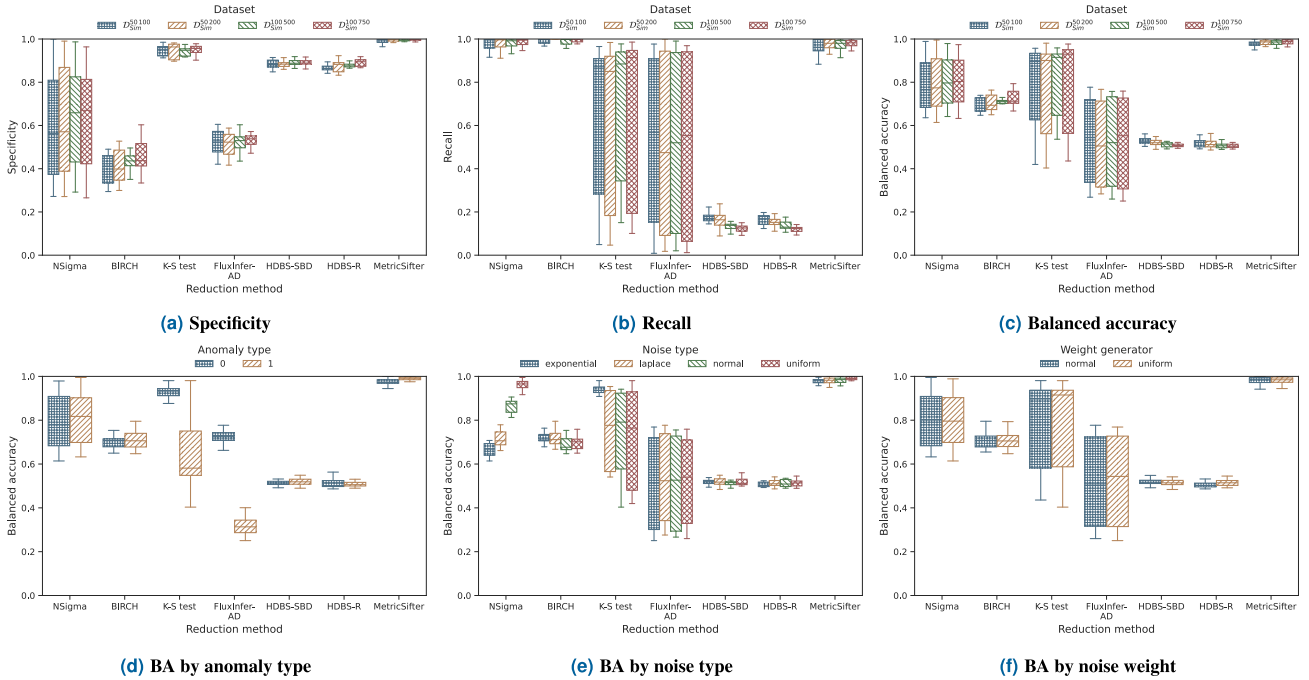


FIGURE 7. Performance of different feature reduction methods in the simulation study. (a), (b), and (c) show the specificity, recall and balanced accuracy (BA), which is a balanced metric of specificity and recall, over different datasets. (d), (e), and (f) show the BA over different anomaly and noise types and noise weight parameters. The vertical lines (whiskers) on the boxplots represent the variability outside the upper and lower quartiles, giving a sense of the distribution of performance values. (c), (d), (e), and (f) show that the overall performance of MetricSifter is better than the baseline methods on average across all datasets and parameters. (a) shows that the specificity of MetricSifter is the best score, but (b) shows that the recall of MetricSifter is equal to or less than NSigma and BIRCH.

explains the localization recall of the corresponding localization methods.

For PC+PageRank, LiNGAM+PageRank and LiNGAM+HT, the R^2 of BA is higher than that of RS. This suggests that feature reduction especially contributes to traditional statistical causal discovery methods such as PC and LiNGAM, which are widely utilized in related works. The negligible performance of ϵ-Diagnosis, approaching zero, makes meaningful analysis impossible. For the RCD and HT-based methods, the R^2 of recall is higher than that of BA. Thus, removing monitoring metrics unrelated to failures does not significantly increase the localization accuracy of these methods.

5) PARAMETER SENSITIVITY

We evaluate how changes in the main hyperparameters of MetricSifter affect the results of feature reduction. MetricSifter has the two main hyperparameters: the penalty weight parameter ω in STEP 1 and the KDE bandwidth h in STEP 2. Fig. 9(a) illustrates the variances in BA resulting from varying ω and h. BA peaks at $\omega = 2.5$ and $h = 3.5$, and it decreases significantly as ω approaches 1.0. This is because reducing the penalty values by lowering ω allows more change points to occur outside the failure time window. Fig. 9(b) and (c) indicate BA with different ω and h, respectively by anomaly type when another parameter is fixed at the peak. h is stable across different anomaly types, whereas ω is more sensitive to anomaly type 1. Note that ω has the same peak for both the anomaly types.

6) ABLATION STUDY

We conduct an ablation study to evaluate the contribution of each step of MetricSifter to the overall reduction performance. In this study, we examine the full version and the version without STEPS 2 and 3. Since the segmentation in STEP 2 depends on the change point detection in STEP 1 and since the upstream selection in STEP 3 depends on STEP 2, running STEP 2 or STEP 3 in isolation is not feasible.

Fig. 10 illustrates the variance in BA with different ω for MetricSifter and MetricSifter (w/o STEPS 2 and 3). The inclusion of STEPS 2 and 3 significantly improves BA, especially for ω values of less than 2.5, which is the optimal value shown in IV-B5. Therefore, STEPS 2 and 3 mitigate the BA loss due to the high sensitivity of ω. For ω values of 2.5 or greater, there is little difference in BA between these methods. The time series in our synthetic datasets do not have significant change points outside the failure time window. Increasing the penalty values prevents the detection of noise as change points outside the failure time window. As a result, MetricSifter achieves high BA without STEPS 2 and 3.

C. EMPIRICAL STUDY ON MICROSERVICES DATA

1) DATASET

To evaluate MetricSifter in a practical setting, we used six datasets with 132 faults from two different cloud applications of different sizes. The details of these datasets are summarized in Table 4. The code for creating these datasets is available in our GitHub repository [62].

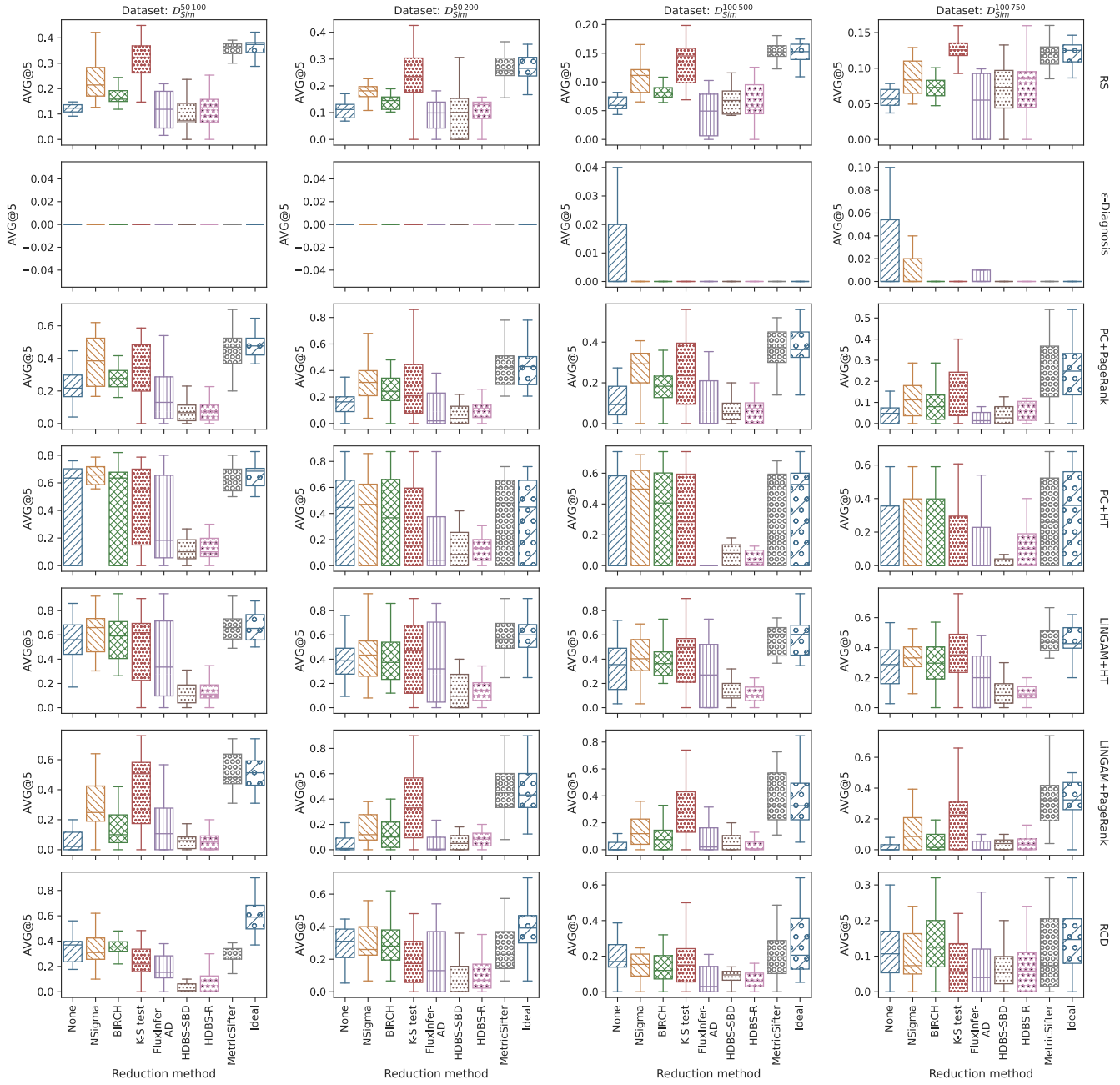


FIGURE 8. Localization accuracy in the simulation study. The vertical lines (whiskers) on the boxplots represent the variability outside the upper and lower quartiles, giving a sense of the distribution of AVG@5, which is the average recall of top-5 results. Each boxplot corresponds to different datasets and fault localization methods, showing the spread of AVG@5 achieved by each method. "None" is the baseline without feature reduction. The AVG@5 of MetricSifter is the best among all baselines except Ideal on average. The normality reduction methods (NSigma, BIRCH, K-S test) except FluxInfer-AD have a higher AVG@5 than that of "None" on average. The redundancy reduction methods (HDBS-R and HDBS-SBD) have a lower AVG@5 than that of "None" on average.

a: TESTBED

We selected two open-source benchmark applications as test settings: Sock Shop, which is a sock sales service, and Train Ticket, which is a train ticket booking service. One or both of these applications have been widely utilized in many related works [10], [11], [15], [21], [25], [63]. Sock Shop consists of seven microservices, and Train Ticket consists of 41 microservices, positioning it as one of the largest public microservices benchmarks. Both systems use polyglot programming (Java, Golang, Node.js, etc.), support databases such as MongoDB and MySQL, and use HTTP REST for interservice communication. Compared to Sock Shop, Train Ticket has longer communication paths and more exposed metrics.

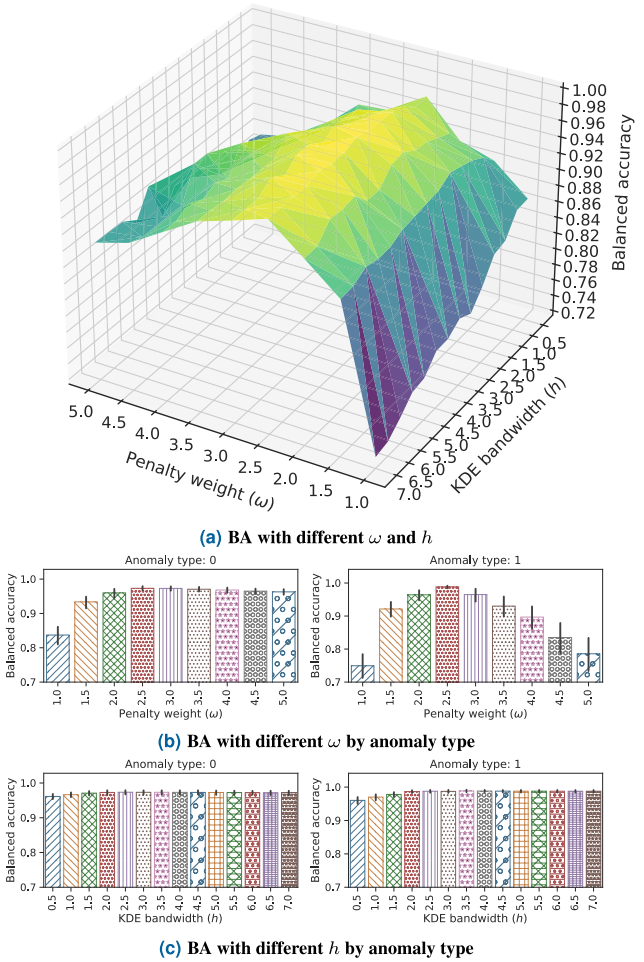


FIGURE 9. Parameter sensitivity analysis. The two hyperparameters of MetricSifter are described in Section III. ω is the penalty weight parameter in STEP 1 and h is the KDE bandwidth in STEP 2. (a) The variance of BA as the reduction performance metric with different ω and h parameters. ω is more sensitive to BA than h. (b) (c) The variance of BA with different ω and h, respectively, by anomaly type when another parameter is fixed at the peak. ω is more sensitive to anomaly type 1.

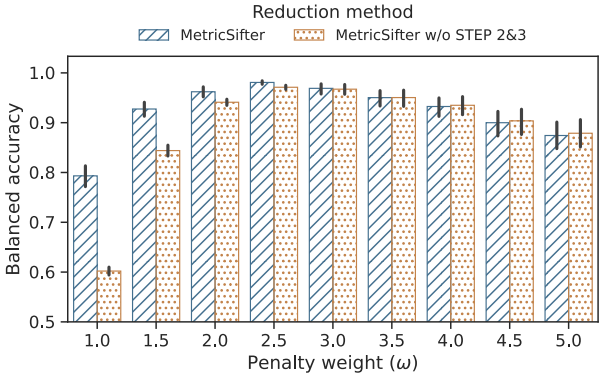


FIGURE 10. Ablation analysis with different ω. The BA of MetricSifter with and without STEP 2 and STEP 3 is compared across different penalty weights (ω) on STEP 1. The near-constant high BA across different ω suggests that MetricSifter is more robust to changes in ω than MetricSifter w/o STEPS 2 and 3. The robustness of MetricSifter to ω is due to the contribution of STEPS 2 and 3.

We deployed two separate Kubernetes [64] clusters, one for Sock Shop and one for Train Ticket, by using the Google Kubernetes Engine (GKE).[2] In a Kubernetes cluster, main nodes manage each cluster, and worker nodes host application container pods. Among the worker nodes, one handles load generation, another manages metric data storage, and the other acts as a control node. The rest (five nodes for Sock Shop and seven for Train Ticket) serve the application deployments. Worker nodes were equipped with two vCPUs and 8 GB of RAM, except for the load generator node, which had 0.5 vCPU and 2 GB of RAM. The microservice containers were maintained at a replication factor of 1.

TABLE 4. Microservices dataset summary.

Dataset	No. of failures	No. of faulty components	No. of metrics	Application
$\mathcal{SS}$-small	90	9	64/81	Sock Shop [27]
$\mathcal{SS}$-medium			184/499	
$\mathcal{SS}$-large			1312/2930	
$\mathcal{TT}$-small	42	7	383/492	Train Ticket [29]
$\mathcal{TT}$-medium			1349/6839	
$\mathcal{TT}$-large			9458/56976	

Six datasets collected from two types of microservice applications (Sock Shop and Train Ticket) used in the empirical study. The "-small" suffix indicates the dataset includes manually selected metrics in only standard pods. The "-medium" suffix means that the dataset includes only standard metrics that are collected commonly in all pods. The "-large" suffix means that the dataset includes middleware metrics that each middleware provides on its own. No. of metrics is {the mean number of unique metrics after STEP 0}/{the total number of unique metrics}. No. of failures is the number of failures in the dataset. No. of faulty components is the number of components that have faults.

b: LOAD GENERATION

To simulate concurrent user access on benchmark applications, we used Locust,[3] which allows workloads to be customized in a way that reflects actual user interactions. Workload patterns in Sock Shop mimicked a typical user journey through the site, starting from landing on the home page and moving on to browsing product listings and placing an order. The load was balanced to replicate a realistic scenario where browsing the home page and catalog is more common than completing a purchase. We loaded Train Ticket according to eight user scenarios: 1) login, 2) unauthenticated ticket search, 3) authenticated search with reservation, 4) baggage check-in for reserved ticket, 5) post-login order payment, 6) no-refund cancellation, 7) ticket issuance, and 8) post-login ticket rebooking. These scenarios were run in parallel to mimic the various activities of numerous users. The load generator sent requests at a frequency of 200–300 per second (RPS) for Sock Shop and 100–150 RPS for Train Ticket.

[2]Google Kubernetes Engine: https://cloud.google.com/kubernetes-engine
[3]Locust: https://locust.io/

c: FAULT INJECTION

While generating workloads, we injected a set of faults to emulate performance degradation in microservices applications. We utilized Litmus Chaos [65], which is capable of injecting a variety of preconfigured faults into Kubernetes-managed containers as a fault injector. We selected two types of faults that are commonly seen in related work [11], [21], [25]: CPU exhaustion and memory overload. CPU exhaustion increases the usage of all visible CPU cores of the container within a container by using the Pod CPU Hog for Litmus Chaos. Memory overload increases memory usage by 500 MB within a container by using the Pod Memory Hog for Litmus Chaos. We injected faults into nine key containers in Sock Shop (carts, carts-db, catalog, catalog-db, payment, orders, order-db, user, user-db) and seven containers in Train Ticket (ts-auth-service, ts-basic-service, ts-order-service, ts-station-service, ts-train-service, ts-travel-service, and ts-user-service). The duration of a fault injection was set to five minutes.

d: DATA COLLECTION

To collect monitoring metric data, we adopted Prometheus [66], a popular monitoring tool, to scrape and save monitoring metrics. We set the scraping interval to 15 seconds in accordance with a Prometheus user survey [67] indicating that this was the preferred length.

We obtained the last 180 data points (equivalent to 45 min) from each time series before the fault injection completion. Data were excluded only if no anomalies were detected in any of the SLI metrics. For the SLI suite, we selected a set of monitoring metrics including average response time, requests per second (RPS), and error rates collected from the Sock Shop and Train Ticket frontends.

2) PERFORMANCE OF FEATURE REDUCTION

Table 5 presents the feature reduction results. PF of MetricSifter outperforms the baselines by 22.2% to 65.3%. The redundancy reduction methods (HDBS-SBD and HDBS-R) demonstrate particular strength in RR, and the normality reduction methods (NSigma, BIRCH, K-S test, and FluxInfer-AD) lead in terms of RF and T(s). Although MetricSifter performs poorly in T(s), it significantly reduces the localization time relative to the normality reduction methods, as detailed in Section IV-C3.

PF of the normality reduction methods is higher than that of the redundancy reduction methods in all datasets. The high recall of the normality reduction indicates that the normality reduction does not remove M^A, as M^A has an anomaly during the failure. In contrast, the redundancy reduction decreases RF because the similarity of monitoring metrics in $M^A \cup M^B$ often leads to unintended reductions of M^A, as noted in Section III-A. Moreover, the redundancy reduction methods show a pronounced escalation in execution time with an increasing number of monitoring metrics per application component in contrast to the normality reduction methods. This can be traced to the computational complexity, which increases quadratically with the number of metrics.

3) PERFORMANCE OF FAULT LOCALIZATION WITH FEATURE REDUCTION

Fig. 11 shows the impact of applying feature reduction methods on the fault localization performance (for simplicity, items with T(s) exceeding 60 minutes are excluded). In terms of localization accuracy, the overall results are consistent with the simulation study described in Section IV-B3.

The average AVG@5 of MetricSifter is the best performance among all methods, with an average AVG@5 of 0.119. The difference in the average AVG@5 of MetricSifter over the best baseline K-S test is 0.014 and that over the worst baseline HDBS-R is 0.050. The difference of the average AVG@5 compared to that of ''None'' ranges from –0.030 (HDBS-R) to 0.020 (MetricSifter). The normality reduction methods have equal to or a higher average AVG@5 than that of ''None.'' However, the redundancy reduction methods are not useful in terms of localization accuracy because they have a lower average AVG@5 than that of ''None.''

The average T(s) of MetricSifter is higher than that of the redundancy reduction methods by 45.87 to 52.25% but lower than that of the normality reduction methods by 33.16 to 60.09% and that of ''None'' by 91.39%. However, T(s)s in larger datasets with the localization methods (except RCD) is over one hour, even when applying each feature reduction method shown in Fig. 11(b).

In conclusion, MetricSifter outperforms the normality reduction methods in both accuracy and time efficiency, although it is less time efficient than the redundancy reduction methods.

V. DISCUSSION

In this section, we discuss both MetricSifter alone and the overall feature reduction.

A. ADVANTAGES AND LIMITATIONS OF METRICSIFTER

1) FLEXIBILITY AS A FRAMEWORK

MetricSifter provides the flexibility to replace any of its steps with alternative methods, thereby allowing users to choose a better method for each step given their specific use cases. Therefore, our major contribution is not to present new methods for change point detection and probability density estimation but rather to design a feature reduction framework integrated with existing algorithms. In addition, the fault localization accuracy (as measured in Section IV-C3) can be further improved by incorporating a more effective fault localization method into the system.

2) HYPERPARAMETERS

MetricSifter users need to tune two hyperparameters: the penalty weight ω in STEP 1 and the KDE bandwidth h in STEP 2. In practice, even suboptimal hyperparameter settings

TABLE 5. Performance of various feature reduction methods in empirical study.

Reduction method	$\mathcal{SS}$-small				$\mathcal{SS}$-medium				$\mathcal{SS}$-large			
	RR	RF	PF	T(s)	RR	RF	PF	T(s)	RR	RF	PF	T(s)
None	0.000	**1.000**	1.55e-02	**0.309**	0.000	**1.000**	2.39e-02	**0.312**	0.000	**1.000**	5.02e-03	**0.450**
NSigma	0.130	0.988	1.77e-02	0.572	0.176	0.988	2.84e-02	0.621	0.318	0.963	6.98e-03	1.140
BIRCH	0.157	**1.000**	1.87e-02	0.375	0.182	**1.000**	2.92e-02	0.420	0.241	0.976	6.38e-03	0.943
K-S test	0.130	**1.000**	1.81e-02	0.588	0.134	**1.000**	2.76e-02	0.640	0.119	0.997	5.66e-03	1.193
FluxInfer-AD	0.130	**1.000**	1.81e-02	0.597	0.134	**1.000**	2.76e-02	0.647	0.119	0.997	5.66e-03	1.198
HDBS-SBD	0.458	0.438	1.25e-02	0.666	0.690	0.254	1.92e-02	0.758	**0.843**	0.135	3.98e-03	6.486
HDBS-R	**0.459**	0.450	1.29e-02	0.674	**0.695**	0.229	1.79e-02	0.989	0.836	0.137	3.82e-03	81.353
MetricSifter	0.324	0.975	**2.52e-02**	0.874	0.415	0.981	**4.24e-02**	1.040	0.437	0.967	**8.53e-03**	2.701

(a) Sock Shop

Reduction method	$\mathcal{TT}$-small				$\mathcal{TT}$-medium				$\mathcal{TT}$-large			
	RR	RF	PF	T(s)	RR	RF	PF	T(s)	RR	RF	PF	T(s)
None	0.000	**1.000**	3.09e-03	**0.487**	0.000	**1.000**	3.44e-03	**0.581**	0.000	**1.000**	7.27e-04	**1.604**
NSigma	0.291	**1.000**	4.40e-03	1.138	0.329	**1.000**	5.10e-03	1.493	0.468	0.989	1.36e-03	5.019
BIRCH	0.267	0.976	4.13e-03	0.720	0.271	**1.000**	4.69e-03	1.097	0.354	0.992	1.12e-03	4.680
K-S test	0.278	**1.000**	4.35e-03	1.174	0.265	**1.000**	4.64e-03	1.566	0.243	0.996	9.60e-04	5.337
FluxInfer-AD	0.278	**1.000**	4.35e-03	1.177	0.265	**1.000**	4.64e-03	1.572	0.243	0.996	9.60e-04	5.359
HDBS-SBD	0.524	0.500	3.16e-03	1.416	**0.743**	0.178	2.50e-03	2.223	**0.854**	0.132	6.64e-04	14.707
HDBS-R	0.527	0.619	4.07e-03	1.652	0.742	0.186	2.64e-03	3.091	0.846	0.129	6.26e-04	105.000
MetricSifter	**0.543**	0.929	**6.32e-03**	3.026	0.612	0.952	**8.43e-03**	4.620	0.598	0.930	**1.69e-03**	22.201

(b) Train Ticket

The three performance metrics for feature reduction (RR, RF, and PF) are defined in Section IV-A4, and the six datasets (SS-small, SS-medium, SS-large, TT-small, TT-medium, and TT-large) are listed in Table 4. "None" is only simple filtering at STEP 0. For a fair comparison, all reduction methods were run after applying STEP 0. Each score is the mean value of the scores in each dataset. "T(s)" is the mean elapsed time from the completion of STEP 0 to the completion of reduction. (a) and (b) show that MetricSifter is the best in terms of PF, which is the overarching metric of feature reduction effectiveness.

can be enough to prevent significant performance degradation of their systems, since parameter h is not sensitive to the performance and STEPS 2 and 3 reduce the performance degradation despite the high sensitivity of the parameter ω, as shown in Sections IV-B5 and IV-B6.

In contrast to the normality reduction methods (excluding FluxInfer-AD), which must specify the normal time window, MetricSifter is independent of the time window parameter. In STEP 3, MetricSifter can identify the anomalous time window instead of arbitrarily specifying the normal time window.

3) COMPUTATIONAL COST

The computational cost of MetricSifter is scaled to the number of application components, as it is designed to be applied to each component. Users can change the target unit size of an application component (such as a container or a microservice) in accordance with their system.

4) METHOD LIMITATIONS

(1) The low proximity of fault-caused change points within $M^A \cup M^B$ is at odds with the assumption from the observations described in Section III-A. In this case, MetricSifter loses some root fault metrics. (2) MetricSifter may erroneously remove root fault metrics without any detectable change points. For example, if a resource nearing full capacity can cause performance degradation, the degree of change in the resource metric is small. This limitation is not unique to MetricSifter, as normality-based reduction methods also rely on detecting anomalies within root fault metrics, and they share the same risk of missing such anomalies.

5) LIMITED DATASETS

Since the simulation and empirical studies in Section IV mimic only a fraction of the possible failure scenarios in a controlled environment, these scenarios may not fully represent those in real user systems. As such, the generalizability of our results may be limited. For example, different data parameters may not give consistent results. In our experiment, the values of three data parameters were fixed: time series length, the ratio of normal and anomalous window sizes, and the sampling rate. Changing the data parameters may result in lower accuracy, so to avoid accuracy loss, MetricSifter users may need to tune the hyperparameters.

B. LIMITATIONS OF OVERALL FEATURE REDUCTION

1) SCALABILITY WITH MONITORING METRICS VOLUME

With the empirical datasets, fault localization methods typically experience decreasing top-5 recall and increasing execution time, regardless of any feature reduction method, as indicated in Fig. 11. In the datasets with more than 1,000 monitoring metrics, the top-5 recalls fall below 0.2, which is too low to be practical. Therefore, the feature reduction methods, including MetricSifter, do not prevent the

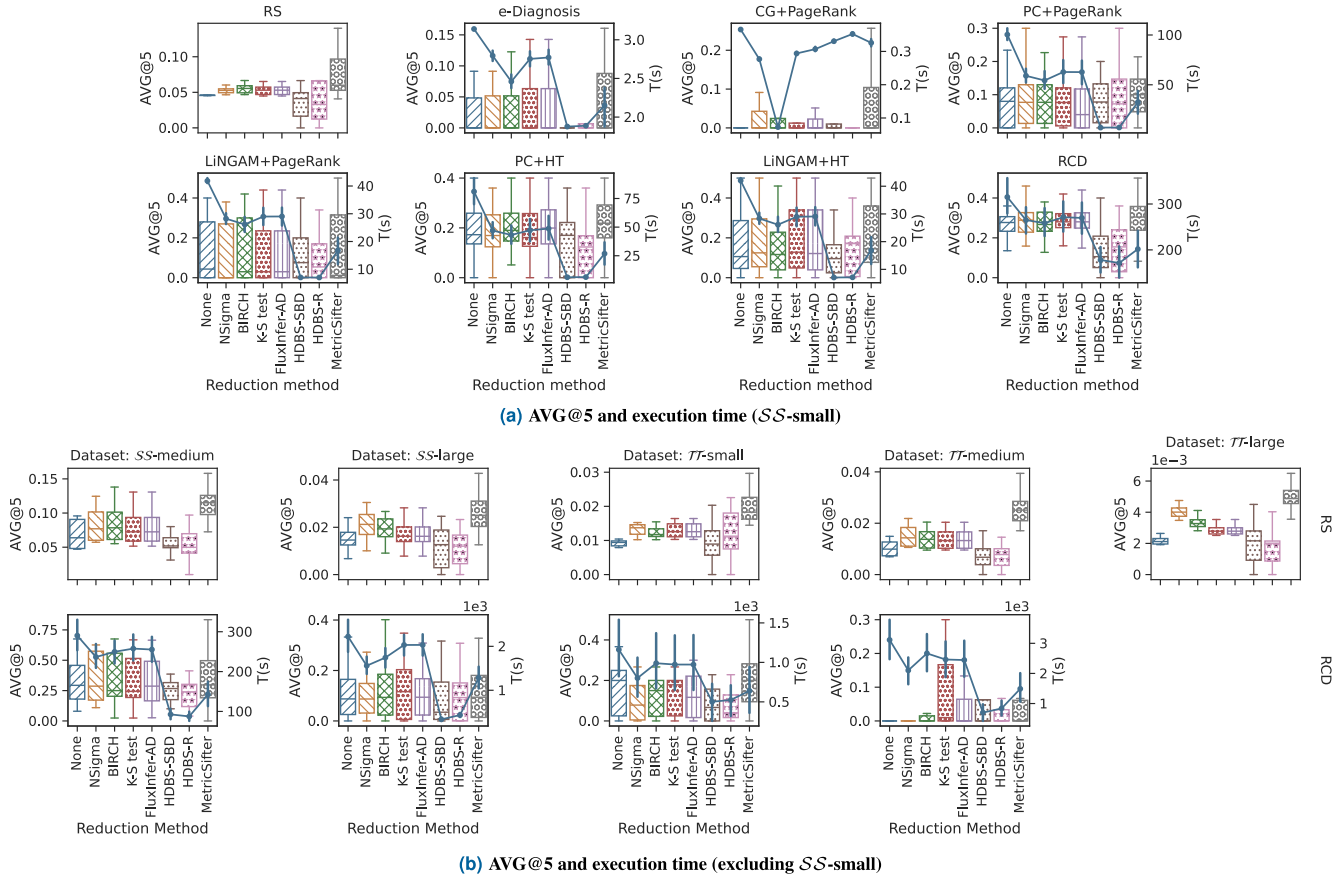


FIGURE 11. Localization performance in the empirical study. The feature reduction methods combining the fault localization methods are compared in terms of their localization performance and computational efficiency in the empirical study. The boxplots correspond to AVG@5, which is the average recall of top-5 results. The vertical lines (whiskers) on the boxplots represent the variability outside the upper and lower quartiles, giving a sense of the distribution of AVG@5. The line plots correspond to T(s). The items with T(s) exceeding 60 minutes are excluded for practicality. Execution time (T(s)) is the sum of the time spent on reduction and localization. The six datasets (*SS*-small, *SS*-medium, *SS*-large, *TT*-small, *TT*-medium, and *TT*-large) are listed in Table 4. The average AVG@5 of MetricSifter outperforms the baseline methods, and the average T(s) of MetricSifter is higher than that of the redundancy reduction methods (HDBS-R and HDBS-SBD) but lower than that of the normality reduction methods (NSigma, BIRCH, K-S test, and FluxInfer-AD).

localization accuracy from degrading with increasing metric volume. In terms of execution time, the localization methods (excluding RCD) take more than one hour on large empirical datasets. Causal learning-based methods such as PC and LiNGAM benefit marginally from parallel computing due to the inherent limitations of parallelism in statistical causal discovery. To address this scalability problem, we implement safe pruning of M^B without degrading the localization accuracy.

2) VALIDITY OF GROUND TRUTH LABELING

In the empirical study, AVG@k is sensitive to the selection of root fault metrics as ground truth. In our experiment, we marked the monitoring metrics that are directly related to the type and location of the fault as ground truth. For the memory overload, we marked memory usage metrics such as mem_used, mem_free, and mem_cached in the injected component. However, the process of injecting memory overload simultaneously increases CPU usage metrics such as cpu_total and cpu_user. It is not obvious, even for domain experts, whether to select one or both of the CPU and memory metrics. Future research in fault localization needs to examine the dependence of the evaluation metric on the ground truth labeling.

3) INCOMPATIBILITY WITH FAULT LOCALIZATION METHODS

MetricSifter and other feature reduction methods output heterogeneous monitoring metrics. Heterogeneity poses compatibility problems with certain fault localization methods. For example, AutoMAP [17] requires the same counts and types of monitoring metrics across application components, and this requirement is beyond the scope of existing automated feature reduction.

VI. RELATED WORK

Automating fault localization in cloud applications by utilizing statistics and machine learning techniques that leverage telemetry data is a growing field [32], [68]. These initiatives typically focus on microservices [69], which decompose

system functionality into numerous small, deployable units and database clusters within cloud ecosystems.

Telemetry data sources categorize the methods for automating fault localization into four groups: metric-based, log-based, trace-based, and multimodal-based. Metric-based methods localize root fault metrics by identifying anomalies in the time series of monitoring metrics or their causal relationships [9], [10], [11], [12], [13], [14], [15], [16], [17], [18], [19], [20], [21], [22], [23], [24]. Log-based methods localize faults through operational text logs during program execution [70], [71]. Trace-based methods localize faults through trace data and examine request paths through systems [63], [72]. Multimodal-based methods analyze two or more data sources (such as metrics, logs, and traces) and then delve into faults [73], [74]. Due to the widespread historical use of metric instrumentation [75], metric-based research has gained broader applicability. Therefore, in this paper we focus on metric-based methods.

Fault localization methods can be divided into two main categories: coarse-grained localization at the component level and fine-grained localization at the metric level [9], [10], [11], [15], [19], [35]. In this paper, we emphasize fine-grained localization, which is more likely to benefit from feature reduction than coarse-grained localization.

Several researchers have utilized fine-grained anomaly-degree based techniques. For example, ϵ-Diagnosis [19] uses p values obtained from a two-sample test that measures the energy distance [76] between metric data during normal and faulty states as an indicator of anomalies. PatternMatcher [9] assigns different weights to these P values by using a supervised learning approach that identifies time series patterns in root fault metrics.

Other researchers have used fine-grained anomaly-propagation based techniques. MicroCause [35] constructs a modified PC algorithm-based causal structure graph that accounts for anomaly propagation delay and traces causality via an adapted random walk method. MicroDiag [15] applies PageRank to inspect a causal structure graph built by DirectLiNGAM [53], which is a causal discovery algorithm based on a structural equation model. CausalRCA [10] also applies PageRank [40] to navigate a causal structure graph built by gradient-based causal structure learning, which detects both the linear and nonlinear causal links between monitoring metrics. RCD [11] builds a causal structure graph by performing conditional independence tests on time series data from both normal and anomalous time windows and then uses causal interventions to locate faults. To address the increased computation time caused by a larger number of metrics, RCD partitions metrics into subsets and then merges them using a divide-and-conquer model.

VII. CONCLUSION

In this paper, we presented MetricSifter, a feature reduction framework designed to enhance fault localization performance. To address the performance and evaluation challenges of previous research, we designed MetricSifter to remove only failure-unrelated monitoring metrics and to quantitatively evaluate feature reduction as a binary classification. Our key insight is that the proximity of fault-caused change point times across failure-related monitoring metrics is higher than that across failure-unrelated metrics. On the basis of this insight, we designed the failure time localization algorithm by identifying the highest density of the change point times. Our framework integrates two traditional statistical models: offline change point detection for univariate time series and probability density estimation with KDE to identify the distribution of the change point times. The results of simulations and empirical experiments demonstrate that MetricSifter outperforms several baselines in terms of its contribution to various fault localization methods. We also verify that the feature reduction accuracy is useful for explaining part of the fault localization recall with coefficient of determination analysis.

Our findings also revealed a scalability challenge with low localization recall and high computational cost for many monitoring metrics. To overcome this challenge, in future work, we plan to redesign the feature reduction to identify and remove failure-related metrics that are less affected by the fault localization recall.

APPENDIX

SYNTHETIC DATA GENERATION

Our synthetic datasets presented in Section IV-B were generated utilizing the PyRCA [58] simulator. The data generation mechanism of the PyRCA simulator is based on the following procedure. We generated normal time window data and anomalous time window data separately and then combined them as complete time series data for simulation.

The normal window data generation is formally given by

$$\mathbf{G}_i = \sum_{G_j \in \mathbf{Pa}(G_i)} D_{ij} \cdot f_i(G_j) + \beta_i \cdot noise_i, \tag{9}$$

where $\mathbf{G}_i$ is the data of the i-th node, D is the weighted adjacency matrix encoding the DAG relations, $\mathbf{Pa}(G_i)$ is the parent node of G_i, $f_i(G_j)$ is the mapping function, $noise_i$ is the underlying noise data, and β_i is the noise weight parameter. Let a nonzero G_{ij} be a causal link from the j-th to the i-th node.

$noise_i$ and a weight of G_{ij} are generated from exponential distribution $\mathcal{E}(1)$, normal distribution $\mathcal{N}(0, 1)$, uniform distribution $\mathcal{U}(-0.5, 0.5)$, or Laplace distribution $\mathcal{L}(0, 1)$. The parameter β_i is the weight based on a normal distribution for each node G_i, represented by $\text{sign}(x)(\text{abs}(x) + 0.2)$ $(x \sim \mathcal{N}(0, 1))$. Alternatively, β_i can be based on a uniform distribution in the range $(-2.0, -0.5) \cup (0.5, 2.0)$. For the function f_i, options include the identity function $f_i(x) = x$, the square function $f_i(x) = x^2$, the sine function $sin(x)$, and the hyperbolic tangent function $tanh(x)$. Because of possible overflow or underflow problems, we fixed f_i to the identity function in our experiments.

To inject a fault $\mathbf{F}$ at time t, we first randomly selected the number of root fault metrics $|\mathbf{F}|$. Since root fault metrics

are typically a small fraction of the total metrics, $|\mathbf{F} - 1|$ followed a Poisson distribution. We chose anomaly 0 and anomaly 1 as the fault injection anomaly types. Both involve injecting faults into $V_i \in \mathbf{F}$. The fault variable $fault_i$ takes a random value from a Poisson distribution for $\mathbf{G}i \in \mathbf{F}$. The fault injection for $\mathbf{G}i$ is defined as follows.

$$\mathbf{G}_i = \begin{cases} \sum D_{ij} f_i(G_j) + \beta_i \cdot noise_i + fault_i & \text{(anomaly 0)} \\ \sum D_{ij} f_i(G_j) + (\beta_i + fault_i) \cdot noise_i & \text{(anomaly 1)} \end{cases} \tag{10}$$

Anomaly 0 adds faults to the constant term, and anomaly 1 adds faults to the weight of the noise term. Computing (10) for all $\mathbf{G}$ nodes was repeated until the SLI metric $\mathbf{G}0$ was detected as an anomaly by using the three-sigma rule. After each loop, squaring $fault_i$ gradually increased the anomaly degree propagated by the fault.

REFERENCES

[1] B. Beyer, C. Jones, J. Petoff, and N. R. Murphy, *Site Reliability Engineering: How Google Runs Production Systems*. Sebastopol, CA, USA: OiReilly Media, 2016.

[2] Y. Wu, B. Chai, Y. Li, B. Liu, J. Li, Y. Yang, and W. Jiang, "An empirical study on change-induced incidents of online service systems," in *Proc. IEEE/ACM 45th Int. Conf. Softw. Eng., Softw. Eng. Pract. (ICSE-SEIP)*, May 2023, pp. 234–245.

[3] C. Nash, "The VOID report 2021," VOID, Tech. Rep. VOID-2021-1006, 2021.

[4] 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 *Proc. IEEE 33rd Int. Symp. Softw. Rel. Eng. (ISSRE)*, Oct. 2022, pp. 121–132.

[5] C. Sridharan, *Distributed Systems Observability: A Guide to Building Robust Systems*. Sebastopol, CA, USA: O'Reilly Media, 2018.

[6] Technical Advisory Group for Observability. (2022). *Observability Whitepaper*. Cloud Native Computing Foundation. [Online]. Available: https://github.com/cncf/tag-observability/blob/main/whitepaper.md

[7] Y. Tsubouchi, A. Wakisaka, K. Hamada, M. Matsuki, T. Kobayashi, H. Abe, and R. Matsumoto, "HeteroTSDB: A high performance time series database for automated data tiering in heterogeneous distributed KVSs," *IPSJ J.*, vol. 62, pp. 818–828, Dec. 2021.

[8] S. Emmons, C. Watson, and B. Gregg. (2015). *A Microscope on Microservices*. [Online]. Available: https://netflixtechblog.com/a-microscope-on-microservices-923b906103f4

[9] C. Wu, N. Zhao, L. Wang, X. Yang, S. Li, M. Zhang, X. Jin, X. Wen, X. Nie, W. Zhang, K. Sui, and D. Pei, "Identifying root-cause metrics for incident diagnosis in online service systems," in *Proc. IEEE 32nd Int. Symp. Softw. Rel. Eng. (ISSRE)*, Oct. 2021, pp. 91–102.

[10] R. Xin, P. Chen, and Z. Zhao, "CausalRCA: Causal inference based precise fine-grained root cause localization for microservice applications," *J. Syst. Softw.*, vol. 203, Sep. 2023, Art. no. 111724.

[11] A. Ikram, S. Chakraborty, S. Mitra, S. Saini, S. Bagchi, and M. Kocaoglu, "Root cause analysis of failures in microservices through causal discovery," in *Proc. Adv. Neural Inf. Process. Syst. (NeurIPS)*, vol. 35, 2022, pp. 31158–31170.

[12] X. Lu, Z. Xie, Z. Li, M. Li, X. Nie, N. Zhao, Q. Yu, S. Zhang, K. Sui, L. Zhu, and D. Pei, "Generic and robust performance diagnosis via causal inference for OLTP database systems," in *Proc. 22nd IEEE Int. Symp. Cluster, Cloud Internet Comput. (CCGrid)*, May 2022, pp. 655–664.

[13] M. Li, Z. Li, K. Yin, X. Nie, W. Zhang, K. Sui, and D. Pei, "Causal inference-based root cause analysis for online service systems with intervention recognition," in *Proc. 28th ACM SIGKDD Conf. Knowl. Discovery Data Mining*, Aug. 2022, pp. 3230–3240.

[14] Z. Hu, P. Chen, G. Yu, Z. He, and X. Li, "TS-InvarNet: Anomaly detection and localization based on tempo-spatial KPI invariants in distributed services," in *Proc. IEEE Int. Conf. Web Services (ICWS)*, Jul. 2022, pp. 109–119.

[15] L. Wu, J. Tordsson, J. Bogatinovski, E. Elmroth, and O. Kao, "MicroDiag: Fine-grained performance diagnosis for microservice systems," in *Proc. IEEE/ACM Int. Workshop Cloud Intell.*, May 2021, pp. 31–36.

[16] P. Liu, S. Zhang, Y. Sun, Y. Meng, J. Yang, and D. Pei, "FluxInfer: Automatic diagnosis of performance anomaly for online database system," in *Proc. IEEE 39th Int. Perform. Comput. Commun. Conf. (IPCCC)*, Nov. 2020, pp. 1–8.

[17] M. Ma, J. Xu, Y. Wang, P. Chen, Z. Zhang, and P. Wang, "AutoMAP: Diagnose your microservice-based web applications automatically," in *Proc. Web Conf.*, Apr. 2020, pp. 246–258.

[18] P. Liu, Y. Chen, X. Nie, J. Zhu, S. Zhang, K. Sui, M. Zhang, and D. Pei, "FluxRank: A widely-deployable framework to automatically localizing root cause machines for software service failure mitigation," in *Proc. IEEE 30th Int. Symp. Softw. Rel. Eng. (ISSRE)*, Oct. 2019, pp. 35–46.

[19] H. Shan, Y. Chen, H. Liu, Y. Zhang, X. Xiao, X. He, M. Li, and W. Ding, "ϵ-diagnosis: Unsupervised and real-time diagnosis of small-window long-tail latency in large-scale microservice platforms," in *Proc. World Wide Web Conf.*, May 2019, pp. 3215–3222.

[20] P. Wang, J. Xu, M. Ma, W. Lin, D. Pan, Y. Wang, and P. Chen, "CloudRanger: Root cause identification for cloud native systems," in *Proc. 18th IEEE/ACM Int. Symp. Cluster, Cloud Grid Comput. (CCGRID)*, May 2018, pp. 492–502.

[21] J. Lin, P. Chen, and Z. Zheng, "Microscope: Pinpoint performance issues with causal graphs in micro-service environments," in *Proc. Int. Conf. Serv. Orient. Comput.*, 2018, pp. 3–20.

[22] P. Chen, Y. Qi, and D. Hou, "CauseInfer: Automated end-to-end performance diagnosis with hierarchical causality graph in cloud environment," *IEEE Trans. Services Comput.*, vol. 12, no. 2, pp. 214–230, Mar. 2019.

[23] P. Chen, Y. Qi, P. Zheng, and D. Hou, "CauseInfer: Automatic and distributed performance diagnosis with hierarchical causality graph in large distributed systems," in *Proc. IEEE Conf. Comput. Commun.*, Apr. 2014, pp. 1887–1895.

[24] M. Kim, R. Sumbaly, and S. Shah, "Root cause detection in a service-oriented architecture," *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 41, no. 1, pp. 93–104, Jun. 2013.

[25] L. Wu, J. Tordsson, E. Elmroth, and O. Kao, "Causal inference techniques for microservice performance diagnosis: Evaluation and guiding recommendations," in *Proc. IEEE Int. Conf. Autonomic Comput. Self-Organizing Syst. (ACSOS)*, Sep. 2021, pp. 21–30.

[26] J. Thalheim, A. Rodrigues, I. E. Akkus, P. Bhatotia, R. Chen, B. Viswanath, L. Jiao, and C. Fetzer, "Sieve: Actionable insights from monitored metrics in distributed systems," in *Proc. 18th ACM/IFIP/USENIX Middleware Conf.*, Dec. 2017, pp. 14–27.

[27] Weaveworks. *Sock Shop: A Microservices Demo Application*. Accessed: Dec. 13, 2023. [Online]. Available: https://microservices-demo.github.io/

[28] X. Zhou, X. Peng, T. Xie, J. Sun, C. Ji, W. Li, and D. Ding, "Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study," *IEEE Trans. Softw. Eng.*, vol. 47, no. 2, pp. 243–260, Feb. 2021.

[29] Software Engineering Laboratory of Fudan University. *Train Ticket: A benchmark microservice system*. Accessed: Dec. 13, 2023. [Online]. Available: https://github.com/FudanSELab/train-ticket/

[30] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Trans. Dependable Secure Comput.*, vol. 1, no. 1, pp. 11–33, Jan. 2004.

[31] W. Wang, J. Chen, L. Yang, H. Zhang, P. Zhao, B. Qiao, Y. Kang, Q. Lin, S. Rajmohan, F. Gao, Z. Xu, Y. Dang, and D. Zhang, "How long will it take to mitigate this incident for online service systems?" in *Proc. IEEE 32nd Int. Symp. Softw. Rel. Eng. (ISSRE)*, Oct. 2021, pp. 36–46.

[32] J. Soldani and A. Brogi, "Anomaly detection and failure root cause analysis in (micro) service-based cloud applications: A survey," *ACM Comput. Surv.*, vol. 55, no. 3, pp. 1–39, Mar. 2023.

[33] S. Schmidl, P. Wenig, and T. Papenbrock, "Anomaly detection in time series: A comprehensive evaluation," *Proc. VLDB Endowment*, vol. 15, no. 9, pp. 1779–1797, May 2022.

[34] V. Chandola, A. Banerjee, and V. Kumar, "Anomaly detection: A survey," *ACM Comput. Surveys*, vol. 41, no. 3, pp. 1–58, Jul. 2009.

[35] Y. Meng, S. Zhang, Y. Sun, R. Zhang, Z. Hu, Y. Zhang, C. Jia, Z. Wang, and D. Pei, "Localizing failure root causes in a microservice through causality inference," in *Proc. IEEE/ACM 28th Int. Symp. Quality Service (IWQoS)*, Jun. 2020, pp. 1–10.

[36] P. Ji, J. Ye, Y. Mu, W. Lin, Y. Tian, C. Hens, M. Perc, Y. Tang, J. Sun, and J. Kurths, "Signal propagation in complex networks," *Phys. Rep.*, vol. 1017, pp. 1–96, May 2023.
[37] C. Glymour, K. Zhang, and P. Spirtes, "Review of causal discovery methods based on graphical models," *Frontiers Genet.*, vol. 10, no. 524, Jun. 2019.
[38] M. J. Vowels, N. C. Camgoz, and R. Bowden, "D'ya like DAGs? A survey on structure learning and causal discovery," *ACM Comput. Surv.*, vol. 55, no. 4, pp. 1–36, Apr. 2023.
[39] P. Spirtes, C. N. Glymour, and R. Scheines, *Causation, Prediction, and Search*. Cambridge, MA, USA: MIT Press, 2000.
[40] L. Page, S. Brin, R. Motwani, and T. Winograd, "The PageRank citation ranking: Bringing order to the web," Stanford InfoLab, Stanford, CA, USA, Tech. Rep. SIDL-WP-1999-0120, 1999.
[41] S. Aghabozorgi, A. Seyed Shirkhorshidi, and T. Ying Wah, "Time-series clustering—A decade review," *Inf. Syst.*, vol. 53, pp. 16–38, Oct. 2015.
[42] T. Zhang, R. Ramakrishnan, and M. Livny, "BIRCH: An efficient data clustering method for very large databases," in *Proc. ACM Int. Conf. Manage. Data (SIGMOD)*, 1996, pp. 103–114.
[43] F. J. Massey, "The Kolmogorov–Smirnov test for goodness of fit," *J. Amer. Stat. Assoc.*, vol. 46, no. 253, pp. 68–78, Mar. 1951.
[44] D. A. Reynolds, "Gaussian mixture models," *Encyclopedia Biometrics*, vol. 741, pp. 659–663, Jan. 2009.
[45] J. Paparrizos and L. Gravano, "k-Shape: Efficient and accurate clustering of time series," in *Proc. ACM Int. Conf. Manage. Data (SIGMOD)*, 2015, pp. 1855–1870.
[46] M. Ester, H. P. Kriegel, and X. Xu, "A density-based algorithm for discovering clusters in large spatial databases with noise," in *Proc. 2nd Int. Conf. Knowl. Discovery Data Mining*, vol. 6, 1996, pp. 226–231.
[47] R. J. G. B. Campello, D. Moulavi, and J. Sander, "Density-based clustering based on hierarchical density estimates," in *Proc. Pacific–Asia Conf. Knowl. Discovery Data Mining*, 2013, pp. 160–172.
[48] C. Truong, L. Oudre, and N. Vayatis, "Selective review of offline change point detection methods," *Signal Process.*, vol. 167, Feb. 2020, Art. no. 107299.
[49] R. Killick, P. Fearnhead, and I. A. Eckley, "Optimal detection of changepoints with a linear computational cost," *J. Amer. Stat. Assoc.*, vol. 107, no. 500, pp. 1590–1598, Dec. 2012.
[50] B. W. Silverman, *Density Estimation for Statistics and Data Analysis*, vol. 26. Boca Raton, FL, USA: CRC Press, 1986.
[51] Y.-C. Chen, "A tutorial on kernel density estimation and recent advances," *Biostatist. Epidemiol.*, vol. 1, no. 1, pp. 161–187, Jan. 2017.
[52] Y. Tsubouchi. (2023). *MetricSifter*. [Online]. Available: https://github.com/ai4sre/metricsifter
[53] S. Shimizu, T. Inazumi, Y. Sogawa, A. Hyvarinen, Y. Kawahara, T. Washio, P. O. Hoyer, K. Bollen, and P. Hoyer, "DirectLiNGAM: A direct method for learning a linear non-Gaussian structural equation model," *J. Mach. Learn. Res.*, vol. 12, pp. 1225–1248, Apr. 2011.
[54] S. Seabold and J. Perktold, "Statsmodels: Econometric and statistical modeling with Python," in *Proc. 9th Python Sci. Conf.*, vol. 57, no. 61, 2010, pp. 25010–25080.
[55] P. Virtanen et al., "SciPy 1.0: Fundamental algorithms for scientific computing in Python," *Nature Methods*, vol. 17, pp. 261–272, Feb. 2020.
[56] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and É. Duchesnay, "Scikit-learn: Machine learning in Python," *J. Mach. Learn. Res.*, vol. 12, pp. 2825–2830, Nov. 2011.
[57] L. McInnes, J. Healy, and S. Astels, "Hdbscan: Hierarchical density based clustering," *J. Open Source Softw.*, vol. 2, no. 11, p. 205, Mar. 2017.
[58] C. Liu, W. Yang, H. Mittal, M. Singh, D. Sahoo, and S. C. H. Hoi, "PyRCA: A library for metric-based root cause analysis," 2023, *arXiv:2306.11417*.
[59] A. Hagberg, P. J. Swart, and D. A. Schult, "Exploring network structure, dynamics, and function using NetworkX," Los Alamos Nat. Lab. (LANL), Los Alamos, NM, USA, Tech. Rep. LA-UR-08-5495, 2008.
[60] P. S. Kochhar, X. Xia, D. Lo, and S. Li, "Practitioners' expectations on automated fault localization," in *Proc. 25th Int. Symp. Softw. Test. Anal.*, Jul. 2016, pp. 165–176.
[61] J. L. Rodgers and W. A. Nicewander, "Thirteen ways to look at the correlation coefficient," *Amer. Statistician*, vol. 42, no. 1, pp. 59–66, Feb. 1988.
[62] Y. Tsubouchi. (2023). *Meltria*. [Online]. Available: https://github.com/ai4sre/meltria
[63] G. Yu, Z. Huang, and P. Chen, "TraceRank: Abnormal service localization with dis-aggregated end-to-end tracing data in cloud native systems," *J. Softw., Evol. Process*, vol. 35, no. 10, p. e2413, Oct. 2023.
[64] B. Burns, J. Beda, K. Hightower, and L. Evenson, *Kubernetes: Up & Running: Dive Into the Future of Infrastructure*. Sebastopol, CA, USA: O'Reilly Media, 2017.
[65] LitmusChaos Authors. *LitmusChaos: Open Source Chaos Engineering Platform*. Accessed: Dec. 13, 2023. [Online]. Available: https://litmuschaos.io/
[66] B. Brazil, *Prometheus: Up & Running: Infrastructure and Application Performance Monitoring*. Sebastopol, CA, USA: O'Reilly Media, 2018.
[67] Prometheus Authors. (2020). *Prometheus User Survey 2020*. [Online]. Available: https://docs.google.com/spreadsheets/d/1piPNpm-JaksUxHoMlJvxFOy35Mwhtt2iSLInihPPe7w/edit?usp=sharing
[68] P. Notaro, J. Cardoso, and M. Gerndt, "A survey of AIOps methods for failure management," *ACM Trans. Intell. Syst. Technol.*, vol. 12, no. 6, pp. 1–45, Dec. 2021.
[69] S. Newman, *Building Microservices*. Sebastopol, CA, USA: O'Reilly Media, 2021.
[70] Q. Lin, H. Zhang, J.-G. Lou, Y. Zhang, and X. Chen, "Log clustering based problem identification for online service systems," in *Proc. IEEE/ACM 38th Int. Conf. Softw. Eng. Companion (ICSE-C)*, May 2016, pp. 102–111.
[71] Y. Sui, Y. Zhang, J. Sun, T. Xu, S. Zhang, Z. Li, Y. Sun, F. Guo, J. Shen, Y. Zhang, D. Pei, X. Yang, and L. Yu, "LogKG: Log failure diagnosis through knowledge graph," *IEEE Trans. Services Comput.*, vol. 16, no. 5, pp. 3493–3507, Sep./Oct. 2023.
[72] J. Rios, S. Jha, and L. Shwartz, "Localizing and explaining faults in microservices using distributed tracing," in *Proc. IEEE 15th Int. Conf. Cloud Comput. (CLOUD)*, Jul. 2022, pp. 489–499.
[73] G. Yu, P. Chen, Y. Li, H. Chen, X. Li, and Z. Zheng, "Nezha: Interpretable fine-grained root causes analysis for microservices on multi-modal observability data," in *Proc. 31st ACM Joint Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, Nov. 2023, pp. 1–10.
[74] C. Lee, T. Yang, Z. Chen, Y. Su, and M. R. Lyu, "Eadro: An end-to-end troubleshooting framework for microservices on multi-source data," in *Proc. IEEE/ACM 45th Int. Conf. Softw. Eng. (ICSE)*, May 2023, pp. 1750–1762.
[75] G. Aceto, A. Botta, W. de Donato, and A. Pescapè, "Cloud monitoring: A survey," *Comput. Netw.*, vol. 57, no. 9, pp. 2093–2115, Jun. 2013.
[76] G. J. Székely and M. L. Rizzo, "Energy statistics: A class of statistics based on distances," *J. Stat. Planning Inference*, vol. 143, no. 8, pp. 1249–1272, Aug. 2013.

YUUKI TSUBOUCHI (Member, IEEE) received the Ph.D. degree from the Graduate School of Informatics, Kyoto University, Kyoto, Japan, in 2023. From 2013 to 2018, he was a Site Reliability Engineer with Hatena Company Ltd. Since 2018, he has been a Researcher with the SAKURA Internet Research Center, SAKURA Internet Inc. His research interest includes site reliability engineering for cloud computing. He is a member of ACM and IPSJ.

HIROFUMI TSURUTA (Member, IEEE) received the master's degree in materials engineering from Kyushu University, Fukuoka, Japan, in 2012. He is currently a Researcher with the SAKURA Internet Research Center, SAKURA Internet Inc. His research interests include machine learning for drug discovery and material science. He is a member of ACM.