# ServiceAnomaly: An Anomaly Detection Approach in Microservices Using Distributed Traces and Profiling Metrics

Mahsa Panahandeh, Abdelwahab Hamou-Lhadj, Mohammad Hamdaqa, and James Miller

**Abstract**— Anomaly detection is an essential activity for identifying abnormal behaviours in microservice-based systems. A common approach is to model the system behavior during normal operation using either distributed traces or profiling metrics. The model is then used to detect anomalies during system operation. In this paper, we present a new anomaly detection approach, called ServiceAnomaly, for anomaly detection in microservice systems that combines distributed traces and six profiling metrics to build an annotated directed acyclic graph that characterizes the normal behaviour of the system. Unlike existing techniques, our approach captures the context propagation provided by distributed traces as a graph that is annotated with functions characterizing both linear and non-linear relationships between profiling metrics. The final annotated graph is used to detect abnormal executions during system operation. The results of applying our approach to two open-source benchmarks show that our approach detects anomalies with an F1-score up to 86%. We also show how developers can use the annotated graph to reason about the causes of anomalies.

**Index Terms**—System Observability, Distributed Traces, Anomaly Detection, Microservice Architectures, Software Reliability, AIOps

## 1 Introduction

Microservice architecture has emerged as the design of choice for the development of software systems in a wide range of industry sectors [1], [2], [3], [4]. Unlike monolithic applications, a microservice-based system is composed of a set of services that are easy to develop, deploy and maintain. Despite their benefits, these systems pose major operational challenges. The highly dynamic aspect of these systems combined with frequent maintenance changes make them prone to failures and crashes in production [5] [4].

Distributed tracing is a relatively new tracing method, designed specifically to provide observability into microservice systems. A distributed trace represents an end-to-end request and contains a series of tagged time intervals known as spans as well as the spans relationships, and optionally metadata such as logs or tags contextualizing spans [6], [7]. A span is a single unit of work carried out by a single service. Spans are also named a timed operation within a trace [7]. However, the term ”operation” does not necessarily denote operations or service functions. In this paper, we define spans at the granularity level of services. An example of a distributed trace is shown in Figure 1. This trace shows


- *M. Panahandeh is with the Department of Electrical and Computer Engineering, University of Alberta. E-mail: panahand@ualberta.ca*
- *A. Hamou-Lhadj is with the Department of Electrical and Computer Engineering, Concordia University. E-mail: wahab.hamou-lhadj@concordia.ca*
- *M. Hamdaqa is with the Department of Computer Engineering and Software Engineering, Polytechnique Montreal. E-mail: mhamdaqa@polymtl.ca*
- *J. Miller is with the Department of Electrical and Computer Engineering, University of Alberta. E-mail: jimm@ualberta.ca*


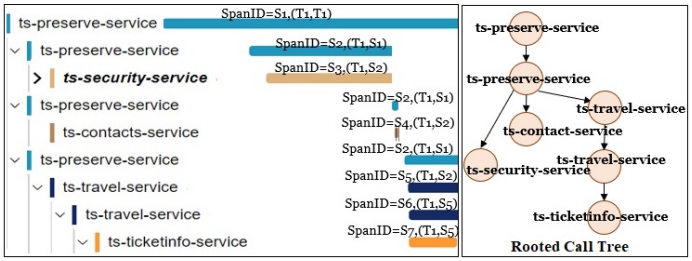


Fig. 1. A distributed trace of TrainTicket system and its corresponding rooted call tree

the services that are invoked following a user request in the TrainTicket system. This trace includes nine spans. Each span carries information including a name, a unique span ID, and a context, which consists of information including the identifier of the trace that contains the span and the parent span. A distributed trace is a directed rooted tree, which can also be represented as a directed acyclic graph (DAG), by capturing common subtrees only once [7] [8] [9]. Figure 1 shows the corresponding rotted call tree of the distributed trace. This call tree is constructed based on the context information, labelled in brackets in the figure.

Studies have shown that the analysis of distributed traces is useful for program comprehension [10], [11], fault diagnosis [12], [13], root cause analysis [14], [15], and anomaly detection [16], [17], the focus of this paper. Existing anomaly detection techniques fall into two categories. The first group of studies use log data, traces or distributed traces to model the service flow of execution [18], [19] or observe patterns within this data [20], [21], [22] for the purpose of anomaly detection. Such approaches are limited to detecting only anomalies that affect the flow of execution. The second

group relies on building a performance profile using various metrics (e.g., latency), which can later be used to indicate significant changes [23], [24], [25], [26], [27], [28], [29], [30], [31], [32]. These approaches vary depending on the applied algorithms and the use of either single or multiple metrics.

Due to the shortcomings in both categories, recent studies have attempted to combine log data and distributed traces with profiling metrics for detecting anomalies [33], [34]. For example, Meng et al. [33] use response time and a flow graph to detect anomalies in microservices caused by performance degradation. The typical metrics used in these studies include execution time [35], response time [33], [36], and latency [14]. However, the use of a single metric has its own limitations since performance problems can manifest themselves across multiple metrics or a combination of metrics.

In this paper, we propose a new anomaly detection technique, called ServiceAnomaly, which builds a model of the normal behaviour of the system using a Context Propagation Graph (CPG) and six profiling metrics. The CPG is a directed acyclic graph that models services that are invoked as user requests propagate through the system. We annotate the CPG by modelling the relationships between six profiling metrics to better characterize the expected normal behaviour of the system. The annotated CPG is used as a baseline to detect potential anomalies in traces of a running system. ServiceAnomaly also provides the possibility to compare a distributed trace that is generated in the presence of an anomaly with a model of the system's normal execution, which should help analysts conduct root cause analysis. We evaluate the effectiveness of our approach using two different benchmark systems, Teastore [37] and TrainTicket [38]. We found that our approach can identify anomaly traces with an F1-score of 85% for TeaStore and 86% for TrainTicket. Moreover, our approach provides a way to reason about the abnormal behaviour that was identified in the anomaly traces.

The paper is organized as follows: In Section 2, we provide an overview of related work. Section 3 explains our ServiceAnomaly approach. In Section 4, we list the evaluation questions and discuss the setup and execution of the experiments we used to answer them. Section 5 presents the results and answers the evaluation questions. Section 6 discusses potential threats to validity. A link to the data and reproduction package is indicated in Section 7. Lastly, we conclude the paper in Section 8 and discuss avenues for future research.

## 2 Related Work

Identifying anomalies in microservices contributes to the stability, reliability, and efficiency of microservice systems. Many approaches have been proposed in the literature. Profiling metrics, logs, and distributed traces are the three primary sources that offer valuable insights into the behaviour of microservices [39], which are essential for understanding the system and identifying and analyzing anomalies. This section outlines the different anomaly detection techniques based on the sources they use. Note that, in this section, we only report on studies that focus on microservices. There exist studies that use execution traces for anomaly detection in monolithic systems such as the work of Islam et al. [40] and Khreich et al. [41]. These studies are outside the scope of this paper.

### 2.1 Detecting anomalies using analysis of execution traces

Bao et al. [18] built a control flow graph from the execution traces extracted from the annotated source code. Then the control flow graph is labelled by a conditional probability value representing the probability of different paths. This graph is used to detect anomalies in the execution sequences. Similarly, Nandi et al. [19] proposed using a control flow graph out of log data. Then, offline template and sequence mining techniques are applied to the graph to generate baseline flows. Zhang et al. [42] have studied the combination of log data and distributed traces. They generated a model of the system using collected data and applied a deep SVDD [43] model to classify anomalous traces.

Liu et al. [21] and Du et al [20] respectively proposed a deep Bayesian network model for learning traces and Long Short-Term Memory (LSTM) for understanding log data and use these learnt models to detect deviations in the execution sequences. Given the power of distributed traces in microservices, Gogatinovsk et al. [22] [44] apply a neural network to distributed traces combined with log data to learn the spans' position and template as a baseline. These studies are limited to the detection of anomalies that manifest themselves through violations to the execution sequences of the system.

### 2.2 Detecting anomalies using analysis of profiling metrics

A considerable number of studies analyze microservices behaviour and detect anomalies by tracking changes in monitoring key performance indicators (KPIs). The most common techniques for detecting changes in the collected metrics are methods based on statistics and learning algorithms. We found that many anomaly detection approaches rely on monitoring only two metrics, mainly the response time or request time. For example, Liu et.al [45], Samir et al. [27], and Lin et al., [28] detect anomalies in the microservices by finding a significant increase in response time or latency time compared to the historical data. Nedelkoski et al. [17] employ an unsupervised deep Bayesian network model to detect changes in response time collected from distributed traces. Li et.al [46] and Wu et al. [25] [47] use clustering distributed traces to find abnormal response times that deviate from normal traces. Zuo et al. [26] suggest a sequential deep learning model applied to log data to detect a deviation in the duration of requests. Li et al [15] perform feature generation and model learning on the latency of requests extracted from log data.

However, all of these studies collect only a single performance metric. Ma et al. [48] believe that a single metric such as response time or latency is not effective enough

to detect anomalies. The importance of using more performance indicators is addressed by He et al. [29]. The authors collected 26 metrics such as CPU, memory, I/O, network, load, and disk usage. Then, they used a neural network model to learn metrics behaviour and detect anomalies. Hou et al. [30] also used a kernel density estimation and waiting moving average method on KPIs including CPU, I/O, and Network usage to model the system's performance and detected anomalies using statistical hypothesis tests. Samir et al. [31] used Spearman's rank correlation coefficient between the collected resource metrics and the number of requests to reason if the performance degradation in the response time is a symptom of an anomaly. Kohyarnejadfard et al. [32] used a multi-class support vector machine for the classification of call frequency and duration of calls into different classes with known labels of normal, CPU shortage, and memory shortage. These classes are used as baselines to assist in identifying resource-based anomalies.

Although there exist studies that use multiple metrics, they are limited to revealing a specific type of anomaly (e.g., resource-based anomalies) and do not provide reasoning about the identified anomalies. In fact, finding the root cause of the anomaly needs more investigation into the system's architecture and execution paths. This group of studies do not use the insight provided by the execution sequences such as log data, traces, and distributed traces. In the following, we review studies that exploit both metrics and execution sequences in detecting anomalies to address these challenges.

### 2.3 Detecting anomalies using analysis of execution sequences and profiling metrics

Zhang et al. [49] encoded frequent signals of log data as well as KPIs using a transformer. Then, a multi-layer perception model is used to learn signals. The learnt signals are used in predicting unexpected behaviours. This study does not elaborate on the used metrics. Moreover, the approach does not assist in understanding the system's behaviour and analyzing anomalies. Nedelkoski et al. [36] proposed an approach that uses a multimodal LSTM to build a model using traces and the response time metric and used this model for anomaly detection. This model relies on a single performance metric.

Similar to the first group of studies, a common approach for understanding the system using both metrics and execution sequences is modelling the collected data as graphs. Fu et al. [35] used generated a workflow state machine from log data. The transitions (edges) of the state machine are annotated with Gaussian functions characterizing the execution time of services. Then, anomalies are detected against the built annotated graph as a baseline if they are not matched with the graph or if their execution time does not belong to the characterized Gaussian distribution. Meng et al. [33] built a call tree from traces and response times. Anomalies are identified when an unexpected call is found by an edit distance algorithm and compared to the built tree or when a surge happens in the response time. Similarly, Xu et al. [34] proposed using a weighted directed graph extracted from traces, resource metrics, and response times representing the effect of a node in calling a neighbour node. They use the sliding window method to predict the next call and response time as a baseline for identifying anomalies.

These studies are close to our approach in terms of representing the system behaviour using a graph. However, these approaches model the microservice's performance with one or a specific number of metrics that can only enable the detection of anomalies that affect these metrics.

Yu et al. [14] clustered distributed traces based on their Euclidean distance. Then, each observed distributed trace was assigned to the corresponding cluster and then, the latency values of traces were clustered into two groups. An anomaly is detected if groups' centroids are far from each other. Wang et al. [16] used a graph-based method using distributed traces. They addressed the explanation of distributed traces by generating a call tree annotated by the services response time. Then, an online incremental clustering [50] and a tree edit distance algorithm are used to cluster the call trees and find deviations from the clusters' centres to detect anomalies affecting distributed trace structures. Also, Wang et al. detect anomalies affecting the response time by tracking a fluctuation in the response time measured by the coefficient of variance.

All the aforementioned papers rely on only a single metric along with execution sequences in understanding the system behaviour. Recently, Zhao et al. [51] proposed a failure detection method, which operates based on profiling metrics, logs, and distributed traces. The authors used a combination of a Graph Transformer Network (GTN) and a Graph Attention Network (GAT) to capture the complex correlation among various modalities of the data, including log templates, profiling metrics, and response times extracted from distributed traces. Then, they used a Gated Recurrent Unit (GRU) to learn the temporal dependencies among the data and predict the multimodal data for the upcoming moments. Failures are identified by comparing the observed data with the predicted values. Their work is based on collecting distributed traces and profiling metrics. However, their method relies on a formal model and lacks visualization tools that could aid in comprehending the system's behaviour or facilitating further insights into anomalies.

Similarly, Lee et al. [52] proposed an anomaly detection process as well as a root cause analysis approach based on KPIs, log data, and distributed traces. They used neural network techniques to acquire insights from various sources: event occurrence from log data, temporal dependencies and inter-service associations from KPIs, and latency distributions from distributed traces. A dependency graph is also extracted from invocation calls of distributed traces. Then, a graph attention network takes merged insights learned from different modalities along with the dependency graph to establish a baseline. Anomalies are detected as deviations from the baseline using a fully connected neural network binary classification algorithm. While this study is close to our approach in using both execution sequences and profiling metrics, it solely visualises dependency extracted from execution sequences. Moreover, its anomaly detection process lacks the incorporation of visualized information

with anomaly detection that aids users in comprehending the detected anomalies and comparing them with the expected behaviour of the system.

Generally, our work is different from the literature in:

- We use distributed traces jointly with six profiling metrics to provide a better characterization of the normal behaviour of the system.
- In modelling the profiling metrics, we use both linear and non-linear relationships between metrics.
- Our annotated CPG provides information beyond the detected anomaly that assists in understanding and analyzing anomalies.
- Our annotated CPG makes it possible to compare an observed trace and a snapshot of the system's performance including profiling metrics with historical data which can be used in comprehension of the system.

Table 1 briefly compares our method with most similar studies in the scope anomaly detection in microservices. The first two attributes compare the methods in terms of their inputs. We know that distributed traces provide a better insight into the microservices compared to the log data and traditional traces [6]. Moreover, using multiple metrics is more effective in detecting anomalies in the system performance [48]. However, to the best of our knowledge, the current literature does not include studies using both distributed traces and profiling metrics in detecting and understanding anomalies. The next studied attribute is the visualization of execution-based information of the system as a model that can assist in locating and better understanding anomalies [53]. We also compared the related works with our studies in terms of generating a baseline for the execution sequence (traces, distributed traces, logs) and metrics. Then, we studied the attribute of analyzing a detected anomaly if it has been provided by the different anomaly detection methods. The sign $*$ means that the attribute has been addressed partially. Finally, the online attribute shows if the proposed methods can automatically adapt to the changes in an operational environment.

## 3 The ServiceAnomaly Approach

Our approach for detecting anomalies in microservice systems, called ServiceAnomaly, relies on building an annotated Context Propagation Graph (CPG) to model the normal system execution. The annotated CPG is built by aggregating context propagation trees collected from distributed traces. A context propagation tree is a DAG of service calls, derived from the span context information, that represents a user request propagating across the system. CPG is an integration of these DAGs in which nodes are services and edges denote service invocations. The CPG is then used as a reference model to detect and analyze abnormal executions caused by faults and other undesirable behaviours. ServiceAnomaly requires two types of telemetry data as input: distributed traces and profiling metrics. This data is generated by exercising the system scenarios (or test cases) in a lab environment. The idea is to collect as much data as possible to cover various paths of the system execution. We use distributed traces to build a CPG and profiling metrics to annotate the CPG, following the steps described in Figure 2, which are discussed in more detail in the subsequent sections. The use of the annotated CPG to identify faults and their root causes is discussed in Section3.4.

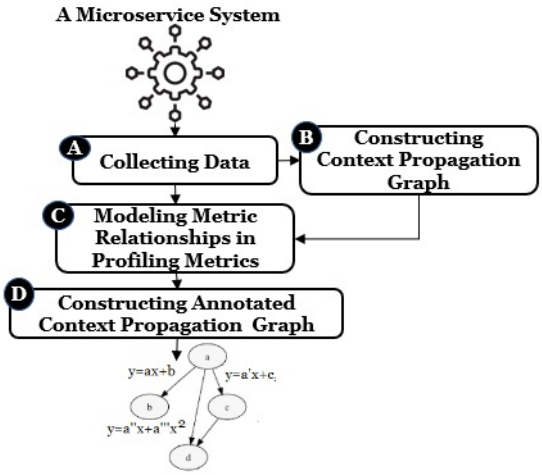


Fig. 2. ServiceAnomaly- Constructing the Annotated Context Propagation Graph

### 3.1 Collecting Data

The goal of this step is to collect distributed traces including context and profiling metrics from the target microservice application by executing the system using various requests. The metrics we focus on in this paper are the service execution time, request size, response size, service queue time, latency, and throughput, which are extensively used to assess the performance of a microservice system. These metrics are defined as follows:

- Request duration time ($m_1$): It is the time in milliseconds between the moment that a caller sends a request and the moment when the caller receives the last byte of the answer. This time is the time that a request is processed through the mesh.
- Request size ($m_2$): It is the number of bytes of the data that is sent by a caller to a callee in order to initiate a request.
- Response size ($m_3$): This metric represents the amount of data in bytes that is sent by a callee to a caller in response to a request.
- Service queue time ($m_4$): It is the time in milliseconds between the moment a request is in the push queue and the moment before it is dequeued.
- Latency ($m_5$): It is the outgoing request latency, i.e., the time in milliseconds it takes for a request to get to its destinations across the network.
- Throughput ($m_6$): It is defined as the number of processes handled by a callee within a specified time.

There are two main approaches for monitoring a microservice application in order to collect data, namely agent-based and agent-less monitoring approaches [39]. In agent-based approaches, an agent needs to be deployed on each node/server, it collects the data and pushes data to a collector. Agent-less approaches, on the other hand, do not

TABLE 1
Comparing ServiceAnomaly with similar studies in the same scope

| Attribute | Fu, 2009 [35] | Nedelkoski, 2019 [36] | Wang, 2020 [16] | Meng, 2021 [33] | Yu, 2021 [14] | Zhang, 2022 [49] | Xu, 2022. [34] | Zhao, 2023 [51] | Lee, 2023 [52] | Service-Anomaly |
|---|---|---|---|---|---|---|---|---|---|---|
| Distributed trace | - | - | + | - | + | - | - | + | + | + |
| Multiple metrics | - | - | - | - | - | + | + | + | + | + |
| Visualization | + | - | + | + | - | - | + | - | + | + |
| exe. Sequence baseline | + | + | + | + | - | - | + | + | + | + |
| Metrics baseline | + | + | - | - | - | - | + | + | + | + |
| Analyzing anomalies | * | - | * | * | - | + | * | - | - | + |
| Online method | - | - | + | + | + | - | + | + | - | - |

require any specific software installation as services are instrumented and send data to a collector directly. While agent-less approaches tend to be lightweight, agent-based approaches provide better control over the collected metrics, are more secure, and do not require code instrumentation [39]. For these reasons, we use an agent-based approach to collect data in our technique.

Figure 3 shows a high-level architecture of how the distributed traces and profiling metrics are collected. At the top level, we have monitoring systems (e.g., Prometheus [54] and Jaeger [55]) that process and visualize distributed traces and profiling metrics, collected by an Istio agent [56]. We deploy our target system on Kubernetes [57]. As shown in Figure. 3, the microservice application is run inside pods. A pod is the smallest execution unit in Kubernetes, which can be replicated. We deploy a single service in each pod. Besides running the application, we inject an Istio proxy into each pod. The Istio proxy is an agent that defines our microservice service mesh, which enables the observability of communication between services. Istio proxies mediate and control the network communication between services, collect data for the traffic passing through the service mesh, and push data to the collectors. In this paper, Istio agents collect data from pods running individual services. Therefore, the granularity of the collected data is at the level of services.

We use Jaeger [55] and Prometheus [54], two popular monitoring systems, to collect distributed traces and profiling metrics. Jaeger [55] is used for distributed tracing to track user requests, whereas Prometheus [54] is used to aggregate the profiling metrics data of the system as a whole. Jaeger is an open-source distributed tracing system that listens to the Istio agent for spans sent over the network. To generate distributed traces of the traffic, first, the tracing feature of the Istio service mesh must be activated. By Istio tracing, tracing headers are recognized by Istio and spans are automatically generated for each service in the mesh. Once a request is received, Istio assigns several headers to it including an ID and the request is passed with its ID through the services. The Jaeger platform receives the requests with their headers and specifies request correlation using IDs which form traces and correlated spans. We use a Jaeger query to gather all traces or find a specific trace. Prometheus is an open-source monitoring tool which collects metrics reported within a service mesh. It scraps endpoints including HTPP and Istio endpoints and collects metrics according to Prometheus scraping jobs described in the Prometheus configuration file. The collected profiling metrics from the microservice traffic can be queried by the Prometheus Query Language (PromQL).

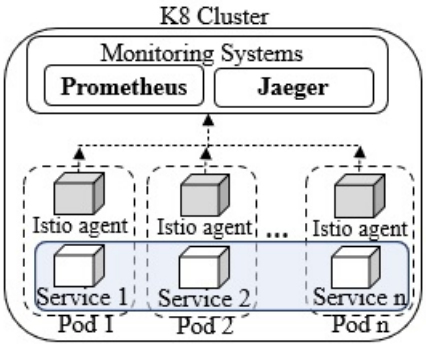


Fig. 3. Data collection architecture using monitoring systems in ServiceAnomaly

### 3.2 Constructing the Context Propagation Graph

The CPG is built from the distributed traces that are collected in the previous step. The CPG is the service call graph. However, we prefer the term "context propagation" to emphasize the fact that the graph captures the spans that are invoked as the user request propagates through the system. Figure 4 describes the process of building a CPG using three fictive traces, t1, t2, and t3. First, we start by merging the rooted call trees of the distributed traces into a context propagation tree. Then we convert the context propagation tree into an acyclic-directed graph (DAG). This is because each rooted ordered tree can be converted into a DAG by representing common subtrees only once [58]. The DAG is the final CPG.

In this paper, we merge the rooted call trees while preserving the observed service invocation paths in the collected traces. This involves merging identical sets of edges that share a common execution path. We follow the same idea when converting the context propagation tree into a DAG. Therefore, within the final CPG, we differentiate between identical services causing different execution paths. To annotate the CPG with profiling metrics relationships, Section 3.3, we create a matrix that contains all the metrics that are collected in the original traces for each edge of the DAG.

Algorithm 1 elaborates on the process of constructing a CPG in detail. Line 1 initializes an empty Tree. Next, each trace in the collected distributed traces (T) is processed as a rooted call tree using the Construct-Rooted-Call-Trees function. Then, in Line 4, the processed rooted call tree is merged with

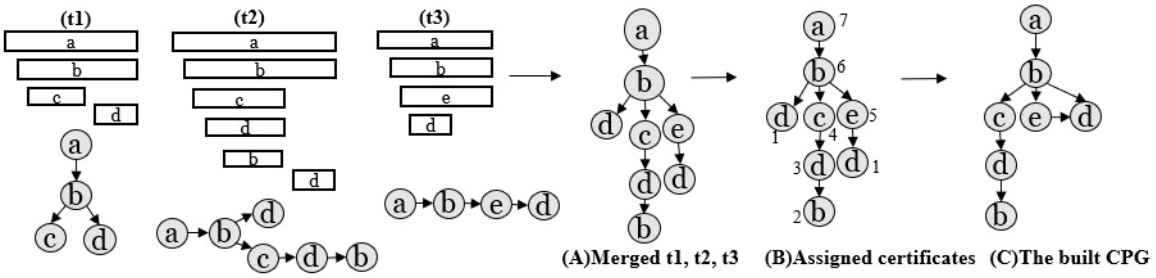


Fig. 4. Constructing a Context Propagation Graph using three fictive distributed traces. This process includes constructing root trees from distributed traces, merging them, and converting the result to a DAG

the previously processed rooted call trees using the Merge-by-Union function and the result is stored in the Tree data structure. Finally, in Line 6, the merged result Tree is fed to the Convert-to-DAG function and converted to a DAG. This DAG is the final constructed CPG.

The next lines in Algorithm 1 provide more details on each of the functions. The Construct-Rooted-Call-Tree function, explained in Lines 7-24, is designed to create rooted call trees from a given trace according to the context information propagated through its spans. It begins by initializing an empty set $\mathcal{E}$ to store the edges of the call tree. Then, for each span in the given trace, it defines an edge. Each edge is a tuple of caller and callee extracted from the span information.

According to Open Tracing [7], every span, except the root span, has a reference to its parent span, which shows the caller-callee relationship among spans. There are two types of parent-child relationships including child-of and follows-from. Edges are assigned by checking parent-child relationships of span. Lines 10 and 11 in the Construct-Rooted-Call-Tree function, check whether the span is a root one with any successor. If this condition is met, the algorithm skips to the next span to get its children's information. Next, Lines 13 and 14 process a root span with no successor that happens when a span is failed to capture its parent-child information. We process this span as an edge with a null caller. If the span is not a root span, the Construct-Rooted-Call-Tree function checks the type of parent-child relationship for the current span. If the relationship type is "FOLLOWS FROM," the algorithm searches for an existing edge in set $\mathcal{E}$ where its callee matches the parent of the current span. If such an edge is found, it means that the current span is the callee, and it has a caller span in the root call tree. The edge's caller is then assigned accordingly. If the relationship type is "CHILD OF," and the span is not a root span, it means the current span is the callee, and its caller is the span parent. After identifying the caller of the edge, the current span is assigned as the callee of the edge and the edge is appended to the list of edges $\mathcal{E}$. Finally, the Construct-Rooted-Call-Tree function returns the list of edges as output.

To merge rooted trees into one tree we adopt Messaoud et al.'s algorithm [60] that was developed to merge tree structures of XML files. The algorithm uses rules to merge two trees $t_1\langle N1, E1\rangle$ and $t_2\langle N2, E2\rangle$, where N1 and N2 are sets of nodes, and E1 and E2 are sets of edges. The Merge-by-Union function in Algorithm 1 explains how we adopted Messaoud et al.'s algorithm to merge rooted call trees. Lines 27-32 check if one tree is a subset of the other tree then the resulting merged tree will be the same as the containing tree, as the smaller tree is already included within it. For example, in Figure 4, t1 is a subset of t2. Therefore, their merged tree results in t2. Moreover, when there are different subtrees in $t_1$ and $t_2$ with the same parent, the resulting merge tree is composed of all subtrees issued from $t_1$ and $t_2$, Lines 33-38. For example, t2 and t3 of Figure 4 have different subtrees rooted at node "b". The resulting merged tree is shown in step A of Figure 4, which includes all subtrees of b. When $t_1$ and $t_2$ include the same subtrees rooted at different parents or when they do not have any edge in common, the merged tree is obtained through the union of $t_1$ and $t_2$, Line 39. Merging subtrees t1, t2, and t3 of Figure 4 results in the tree shown in Figure 4(A).

The next step is to convert the resulting rooted tree to a DAG by capturing common subtrees only once. This step is needed to reduce the number of nodes and edges of the final CPG. Hamou-Lhadj et al. [8] showed that the size of routine call traces can be significantly reduced if we can represent them as DAGs. The same principle applies to call trees of services. For example, in Figure 4(A), we can see that the subtree rooted at "d", which consists in our case of only one node, is repeated twice. To convert a rooted tree into a DAG, we use a variant of Valiente et al's algorithm [59] that is used to solve the common subexpression problem, the presence of repetitions in rooted ordered trees. Valiente et al.'s algorithm iteratively traverses the rooted tree bottom-up and assigns a certificate (a unique identifier) to each node. The assigned certificate distinguishes different nodes and is used to identify and remove redundant subpaths. First, the algorithm creates a unique signature for each node. The signature ($sig$) is defined by a triple of $sig(n) =< Label(n), sig(Left(n)), sig(Right(L)) >$ where Right(n) and Left(n) represent the right and left children of node n and Label(n) is the label of the node n. The signature determines if two subtrees are isomorphic. The algorithm stores signatures in a hash table. It checks to see if the assigned signature is already in the table, and if not, it adds the signature as well as a computed certificate to the table. Then, the certificate is assigned to the traversed node. The certificate is a positive integer between 1 to the number of nodes computed incrementally and assigned to the nodes. If the hash table includes the signature, the corresponding certificate is then returned from the table and assigned to the traversed node. Figure 4(B) shows the rooted tree with

**Algorithm 1** Constructing the Context Propagation Graph Algorithm

**Require:** $T : distributed\ traces$
Initialize Tree= $\emptyset$
**for each** $trace \in T$ **do**
    t=Construct-Rooted-Call-Trees($trace$)
    Tree= Merge-by-Union (Tree, t)
**end for**
**return** CPG= Convert-to-DAG(Tree)

**Function** Construct-Rooted-Call-Trees($trace$) **begin**
$\mathcal{E} \leftarrow \emptyset$
**for each** $span \in trace$ **do**
    Initialize $edge \leftarrow null$
    **if** $span$ is a root **and** $span$.**next** != null **then**
        **continue** ▷ Skip to the next iteration
    **end if**
    **if** $span$ is a root **and** $span$.**next** == null **then**
        $edge.caller \leftarrow null$
    **end if**
    **if** ($span.refType$ == "FOLLOWS_FROM") **then**
        $edge.caller \leftarrow \mathcal{E}.get(e.caller|e.callee = span.parent)$
    **end if**
    **if** ($span.refType$ == "CHILD_OF" **and** $span$ is not a root ) **then**
        $edge.caller \leftarrow span.parent$
    **end if**
    $edge.callee \leftarrow span$
    $\mathcal{E} += edge$
**end for**
**return** $\mathcal{E}$

**Function** Merge-by-Union($t_1 < N_1, E_1 >, t_2 < N_2, E_2 >$) **begin**
Initialize t-union
**if** $t_1 \subseteq t_2$ **then**
    t-union=$t_2$
**end if**
**if** $t_2 \subseteq t_1$ **then**
    t-union=$t_1$
**end if**
**if** $(t_1 \cap t_2 \neq \emptyset\ AND\ \exists\ n_i \in N_1 = n_j \in N_2|\ (\ parent(n_i)\ OR\ parent(n_j) = null)\ OR\ (parent(n_i) = parent(n_j))\ AND\ child(n_i) \neq child(n_j))$ **then**
    t-union$< N_3, E_3 >= t_1 \cup t_2\ where\ N_3 = N_1 \cup N_2, E_3 = E_1 \cup E_2\ where\ child(n_i)$ is connected to $\forall n_k \in t_2 | n_k = n_j\ AND\ child(n_j)$ is connected to $\forall n_k \in t_1 | n_k = n_i$
**end if**
**if** $t_1 \cap t_2 \neq \emptyset\ AND\ \exists\ n_i \in N_1 \neq n_j \in N_2|\ parent(n_i) = parent(n_j)\ AND\ child(n_i) = child(n_j)$ **then**
    t-union$< N_3, E_3 >= t_1 \cup t_2\ where\ parent(n_i) is connected to parent(n_j)\ AND\ N_3 = N_1 \cup N_2, E_3 = E_1 \cup E_2$
**else**
    t-union$< N_3, E_3 >= t_1 \cup t_2\ where\ t_1 and\ t_2$ make a forest
**end if**
**return** t-union

**Function** Convert-to-DAG($tree$): This function is based on the work of Valiente et al. [59]

the certificates. Figure 4(C) shows the final CPG that is constructed from t1, t2, and t3 as a DAG.

### 3.3 Annotating the Context Propagation Graph

The goal of this step is to annotate the CPG with the profiling metrics ($m_1$, $m_2$,...,$m_6$) to have a representative baseline model that characterizes the normal execution of the system. Our approach is to model the relationship between the metrics for each span invocation (i.e., each edge of the CPG). We want to know the relationship between any pair of metrics ($m_1$ to $m_6$) for each edge of the CPG.

There are two types of relationships among any two metrics, $m_i$ and $m_j$, linear and non-linear relationships. To identify and model the type of relationships of the metrics, we follow two steps: (a) building the profiling metrics matrix, and (b) characterizing the relationships between the metrics. Algorithm 2 elaborates on annotating the context propagation graph using these two steps. This algorithm requires the constructed CPG from the previous step and collected profiling metrics ($\mathcal{PM}$) from the Collecting data step. We explain the algorithm in the following sections.

#### 3.3.1 Building the profiling metric matrix

We start by running the system to collect enough data to help us determine the relationship between the profiling metrics. In Algorithm 2, $\mathcal{PM}$ contains all collected profiling metrics. Lines 1-2 in Algorithm 2 summarizes the construction of the profiling metrics matrix for each edge of the CPG. We use the Prometheus Query Language (PromQL) to collect the metrics for each edge of the CPG. For example, the following PromQL query returns the request duration time of the invocation characterized by the edge a→b over one day, with data points collected at 1-minute intervals:

$request\_duration\_milliseconds\{destination\_service\_name = \text{"}b\text{"}, reporter = \text{"}source\text{"}, source\_app = \text{"}a\text{"}, response\_code = \text{"}200\text{"}\}[1d : 1m]$

Applying similar queries for all six profiling metrics described in 3.1, we obtain a matrix where the number of columns equals the number of metrics, i.e., 6, and the rows represent the different observation points in time. For example, $m_{1,t0}$ refers to the observed value at $t_0$ for the first metric $m_1$ (request duration time) and $m_{6,tn}$ is the last observed value at time $t_n$ for the $m_6$.

$$\mathcal{P} = \begin{pmatrix} m_{1,t0} & m_{2,t0} & \cdots & m_{6,t0} \\ m_{1,t1} & m_{2,t1} & \cdots & a_{6,t1} \\ \vdots & \vdots & \ddots & \vdots \\ m_{1,tn} & m_{2,tn} & \cdots & m_{6,tn} \end{pmatrix} \quad (1)$$

Figure 5 shows an example of profiling metrics for the two traces T1 and T2 that are collected at times $t_0$ and $t_1$, respectively. Each line represents the collected profiling metrics based on the called span. For example, for span 'b' called by span 'a' in these two traces, we have two sets of metrics that were collected using Prometheus. Extracting metrics for 'b' called by 'a' ($a \rightarrow b$ for simplicity) results in the following $\mathcal{P}$ matrix where each column represents the values of a specific metric over time, and the rows represent the six metrics of interest. The value of each metric for $a \rightarrow b$ over times $t_0$ and $t_1$ used to build the matrix $\mathcal{P}$ are highlighted in Figure 5.

$$\mathcal{P}_{a\rightarrow b} = \begin{matrix} \\ t_0 \\ t_1 \end{matrix} \begin{matrix} m_1 & m_2 & m_3 & m_4 & m_5 & m_6 \end{matrix} \begin{pmatrix} 0.5 & 100 & 90 & 2 & 1 & 0.88 \\ 1 & 121 & 110 & 4 & 1 & 0.88 \end{pmatrix} \quad (2)$$

#### 3.3.2 Characterizing the relationships between metrics

In this step, we determine both linear and non-linear relationships between the profiling metrics. Then, we use regression functions to model the relationship between each pair of metrics for each edge of the CPG. The functions are used to annotate the CPG. In the following, we begin by identifying the linear relationships using linear correlation. We then explain the process of characterizing non-linear relationships using mutual information and non-linear Support Vector Regression (SVR).

**A) Identifying and characterizing linear relationships:**

A linear relationship is characterised by a direct connection between two variables, implying that both variables change in the same proportion [61]. We use the Pearson Correlation Coefficient (Equation 3) [61] to estimate the degree of a linear relationship between two variables, in the case of any two metrics, $m_i$ and $m_j$.

$$r_{m_i,m_j} = \frac{\sum_{t=0}^{n}(m_{i,t} - \bar{m_i})(m_{j,t} - \bar{m_j})}{\sqrt{\sum_{t=0}^{n}(m_{i,t} - \bar{m_i})^2}\sqrt{\sum_{t=0}^{n}(m_{j,t} - \bar{m_j})^2}} = \frac{Cov(m_i, m_j)}{\sigma_{m_i}\sigma_{m_j}} \quad (3)$$

where $\bar{m_i}$ and $\bar{m_j}$ are the averages of the two metric distributions, $Cov(m_i, m_j)$ is covariance of $m_i, m_j$, and $\sigma_{m_i}$ and $\sigma_{m_j}$ are respectively the standard deviations of each distribution. $r_{m_i,m_j}$ ranges from -1 to 1. A value close to -1 or 1 means that there is a strong correlation between the two variables. $r_{m_i,m_j}$ converges to 0 means that there is no linear relationship between the two variables. It should be noted that this does not mean that there is a non-linear correlation between the two variables [62], [63] since there may not be any relationships between the two variables. We will discuss non-linear correlation in the next subsection.

We need to determine a threshold beyond which we can consider two metrics are strongly linearly correlated using the Pearson coefficient. For this, we use the same threshold as the one proposed by Jiang et al. [64]. The authors used linear correlation and normalized mutual information (NMI) together, as a similarity measurement to cluster similar software applications metrics into groups. According to Jinag et al [64] two metrics $m_i, m_j$ are strongly linearly correlated if $r^2_{m_i,m_j} > 0.6$. The measurement $r^2_{m_i,m_j}$ is a positive value between 0 and 1 which specifies the proportion of the variance of $m_j$ that is explained by $m_i$ [61]. Given a matrix $\mathcal{P}$ for each edge of the CPG, we measure a symmetric matrix showing $r^2$ between all pairs of metrics in matrix $\mathcal{P}$ (Lines 3-4 in Algorithm 2).

**Algorithm 2** Annotating the Context Propagation Graph

**Require:** $\mathcal{PM}$: Collected profiling metrics
**Require:** $CPG$
1: **for each** $edge \in CPG$ **do**
2: extract $\mathcal{P}$ matrix from $\mathcal{PM}$ for $edge$
3: **for each** $m_i, m_j \in \mathcal{P}$ **do**
4: $R^2(\mathcal{P}), NMI(\mathcal{P})$
5: **if** $r^2_{m_i,m_j} > 0.6$ **then**
6: characterize $m_i, m_j$ using a linear regression function and $\epsilon_r$
7: **end if**
8: **if** $r^2_{m_i,m_j} < 0.6 and NMI > t$ **then**
9: characterize $m_i, m_j$ using a non-linear SVR regression function and $\epsilon_{svr}$
10: **end if**
11: Annotate $edge$ with linear and non-linear regression functions
12: **end for**
13: **end for**
14: ▷ Different error evaluation metrics can be computed from $\epsilon_r, \epsilon_{svr}$ to evaluate the error in the linear and non-linear regression functions

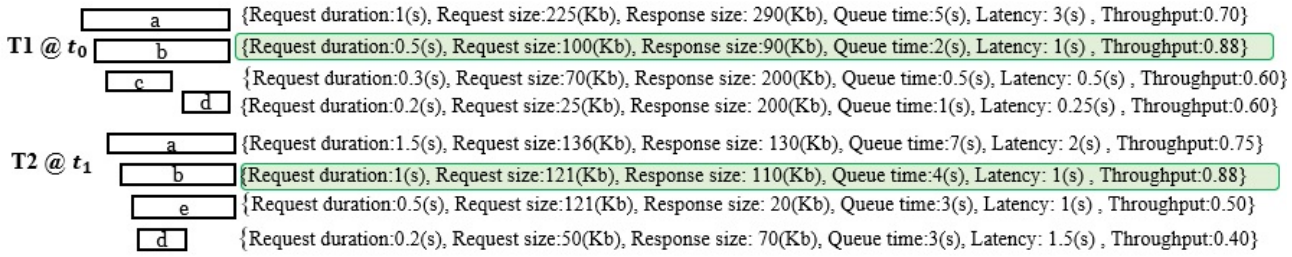


Fig. 5. Profiling metrics for traces T1, T2. Each line shows the collected six profiling metrics based on the called span.

$$\mathcal{R}^2 = \begin{pmatrix} r^2_{m_1,m_1} & r^2_{m_1,m_2} & \cdots & r^2_{m_1,m_k} \\ r^2_{m-2,m_1} & r^2_{m_2,m_2} & \cdots & r^2_{m_2,m_k} \\ \vdots & \vdots & \ddots & \vdots \\ r^2_{m-k,m_1} & r^2_{m_k,m_2} & \cdots & r^2_{m_k,m_k} \end{pmatrix} \quad (4)$$

We use linear regression to characterize the linearly correlated metrics. Linear regression attempts to find the best function that describes the linear relationship of the data [63]. The linear relationship for metrics $m_i$ and $, m_j$ is characterized using the following regression function [61]:

$$m_i = \beta_0 + \beta_1 m_j + \varepsilon_r \quad (5)$$

$m_i$ and $m_j$ are time series representing the value of these metrics at $t_0$ to $t_n$. $\beta = \begin{pmatrix} \beta_0 \\ \beta_1 \end{pmatrix}$ contains coefficients of the regression model and $\varepsilon$ is the disturbance vector or error calculated as follows:

$$\beta_1 = \frac{\sum_{t=0}^{n}(m_{it}-\bar{m_i})(m_{jt}-\bar{m_j})}{\sum_{t=0}^{n}(m_{it}-\bar{m_i})^2}, \beta_0 = \bar{m_i} - \beta_1 \bar{m_j},$$

$$\varepsilon_r = m_i - \begin{pmatrix} 1 & m_{t_0 j} \\ 1 & m_{t_1 j} \\ & \vdots \\ 1 & m_{t_n j} \end{pmatrix} \beta$$

Lines 3-6 in Algorithm 2 summarize identifying and characterizing linear relationships for each pair of metrics.

The $\epsilon_r$ vector includes the actual error values of $m_i$ or residual for each entry, i.e., $m_i - \hat{m}_i$ when $\hat{m}_i$ represents the computed value for $m_i$ using the regression function 5. There are various error evaluation metrics used to evaluate errors in regression functions, each of which computes an individual error evaluation value based on actual error values. In this study, we use the following six error evaluation metrics [65], [66] that are calculated from the vector $\epsilon_r$.

- Mean absolute error (MAE): It is a metric measuring the mean of absolute residuals. A lower MAE value corresponds to a better-fit function [65], [66]. Given a $\epsilon_r$ vector including the actual error values of metric $m_i$ for the time points $t_0$ to $t_n$, we measure MAE as follows, where $\epsilon_{rt}$ represents the actual error value at time t:

$$MAE(m_i, \hat{m}_i) = \frac{1}{n}\sum_{t=t_0}^{t_n} |\epsilon_{rt}|$$

- Mean absolute percentage error (MAPE): It is a measure defined based on the relative percentage of errors. It represents the average absolute percentage errors while a lower value shows a stronger regression fit model [65], [67]. Given a $\epsilon_r$ vector, MAPE is computed using the following equation:

$$MAPE(m_i, \hat{m}_i) = \frac{1}{n}\sum_{t=t_0}^{t_n} \left| \frac{\epsilon_{rt}}{m_{it}} \right|$$

- Mean squared error (MSE): This represents the expected value of the quadratic loss. The lower the MSE value, the more accurate the function to the actual observed value [65], [66]. The MSE is calculated as:

$$MSE(m_i, \hat{m}_i) = \frac{1}{n}\sum_{t=t_0}^{t_n}(\epsilon_{rt})^2$$

- Root mean square error (RMSE): This is the standard deviation of the residuals that shows how the regression is adjusted to the data and the quality of the model. A lower RMSE means a higher regression quality [65], [66]. RMSE is calculated as:

$$RMSE(m_i, \hat{m}_i) = \sqrt{\frac{\sum_{t=t_0}^{t_n}(\epsilon_{rt})^2}{n}}$$

- Median absolute error (MedAE): is a robust measurement to outliers which captures the median of all absolute differences between the target and the prediction [65]. The MedAE is calculated as:

$$MedAE(m_i, \hat{m}_i) = median(|\epsilon_{rt}|), t = t_0, t_1, ..., t_n$$

- Maximum error(max-error): We also use maximum error which is suggested in the literature [68]. It is a metric measuring the worst-case error between the predicted value and the true value. A low max-err means the presence of lower error values and hence a better-fit regression. The max-err is computed as follows:

$$max - err(m_i, \hat{m}_i) = max(|\epsilon_{rt}|), t = t_0, t_1, ..., t_n$$

We will discuss different error evaluation metrics and their effect on our experiments in the evaluation section (Section 4).

**B) Identifying and characterizing non-linear relationships:**

The correlation coefficient $r$ can be used to determine if two variables are linearly correlated. However, it cannot be used to deduce that two variables are not linearly correlated [62], [63]. Figure 6 shows cases where the correlation coefficient $r$ alone cannot describe non-linear relationships between variables. For example, there may not be any relationship between the two variables (as shown in Figure 6 bottom left quadrant). To identify the non-linear relationships between our metrics, we use the mutual information (MI) concept with the correlation coefficient as suggested by Jiang et al. [64]. Mutual information defined using conditional entropy and describes the amount of information that each variable provides knowing the other one. In this paper, we use Normalized Mutual Information (NMI) measurement since our collected metrics have different ranges, which is calculated using Equation 6.

$$NMI(X|Y) = \frac{H(X) - H(X|Y)}{\sqrt{H(X)H(Y)}} \quad (6)$$

The normalized mutual information for two variables of X and Y is defined as follows where H(X) is the entropy of X and $H(X|Y)$ is conditional entropy describing the uncertainty of X when the Y value is known. The mutual

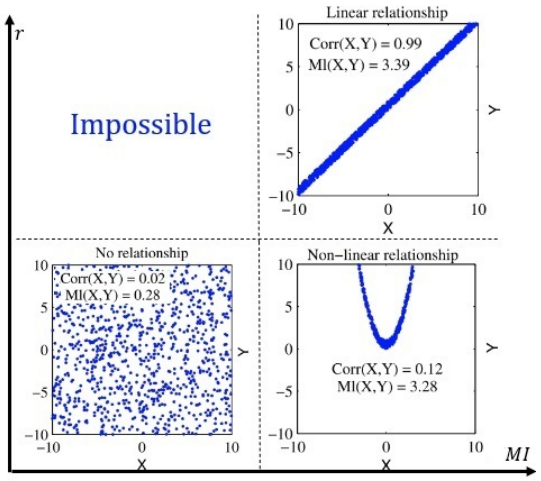


Fig. 6. Different types of the relationship between variables X and Y. The top right shows a linear relationship with high values of both the correlation and MI. In the bottom right, there is a non-linear relationship that is well captured by the MI, but the correlation does not reflect it. The bottom-left shows no relationship and both MI and correlation are low. The top left is impossible since high correlation cause high MI value, adapted from [62]

information is a symmetric value that describes the reduced uncertainty in X when Y is known. If two variables X and Y are determined together, i.e., if Y does not add uncertainty to X value, then, X and Y are dependent and the conditional entropy equals zero. In this case, NMI(X,Y) equals one that it means NMI is in its max value. The higher the NMI value, the more dependent the variables are. Similarly to the $r^2$ for a given $\mathcal{P}$ matrix, we measure an NMI matrix showing the NMI value for each pair of metrics $m_1$ to $m_6$.

$$\mathcal{NMI} = \begin{pmatrix} NMI_{m_1,m_1} & NMI_{m_1,m_2} & \cdots & NMI_{m_1,m_k} \\ NMI_{m_2,m_1} & NMI_{m_2,m_2} & \cdots & NMI_{m_2,m_k} \\ \vdots & \vdots & \ddots & \vdots \\ NMI_{m_k,m_1} & NMI_{m_k,m_2} & \cdots & NMI_{m_k,m_k} \end{pmatrix} \quad (7)$$

An NMI greater than a certain threshold, $t$ (which will be discussed later), indicates that there is either a linear or a non-linear relationship between two variables [62], [64], [69]. Jiang et al. [64] suggest using $r^2$ jointly with the NMI value to interpret the NMI magnitude and identify non-linear relationships. For a pair of metrics $m_i$ and $m_j$, if $r^2_{m_i,m_j} > 0.6$ then we conclude that $m_i$ and $m_j$ are strongly linearly correlated. On the other hand if $r^2_{m_i,m_j} < 0.6$ and $NMI >= t$ then $m_i$ and $m_j$ are non-linearly correlated. The threshold $t$ needs to be determined by analyzing the data as suggested by Jiang et al. by [64]. In this paper, we follow the same approach, explained in the next section, to identify the threshold $t$ according to our experiments.

To characterize a non-linear relationship, we use non-linear Support Vector Regression (SVR) [70], [71]. SVR uses decision boundaries along with a maximal margin to find the best-fit hyperplane between two variables including the maximum number of data points on it. These points are called support vectors (SV). Non-linear decision boundaries of SVR transform the original feature space into a high-

dimensional space in which non-linear relationships can be estimated by a linear regression model. Feature space transformation can be written as a mapping of $\varphi : m_i, m_j \rightarrow Z$ and the transformation function (kernels) is defined as: $K(x_i, x_j) = \varphi(x_i)^T \varphi(x_j)$, while $x \in m_i, m_j$. This function shows how $x_i$ is compared to $x_j$.

The non-linear SVR regression function is calculated as:

$$w = \sum_{i=1, x \in SV}^{n} (\alpha_i^* - \alpha_i) k(x_i, x) + b \tag{8}$$

Where n is the length of the data sets (number of collected metrics). The constant b is the model parameter and $\alpha = (\alpha_1, \alpha_2, ..., \alpha_n)^T, \alpha^* = (\alpha_1^*, \alpha_2^*, ..., \alpha_n^*)^T$ are model coefficients, measured by maximizing the objective function:

$$\begin{aligned} -\frac{1}{2} \sum_{i,j=1}^{n} (\alpha_i^* - \alpha_i)(\alpha_j^* - \alpha_j) k(x_i, x_j) - \\ \epsilon \sum_{i=1}^{n} (\alpha_i^* - \alpha_i) + \sum_{i=1}^{n} (\alpha_i^* - \alpha_i) y_i \end{aligned} \tag{9}$$

subject to the constraints of:

$$\sum_{i=1}^{n} (\alpha_i^* - \alpha_i) = 0, 0 \leq \alpha_i, \alpha_i^* \leq C$$

While $y_i$ is the indicator of each data point (label of $m_i$ and $m_j$) and $\epsilon$ and $C$ are known as the hyper-parameters used to minimize the fitting error for the regression.

Similarly to the $\varepsilon_r$, for each non-linear regression, there is an error vector called $\varepsilon_{SVR}$. The vector $\varepsilon_{SVR}$ includes actual error values for each entry of $m_i$ computed from the difference between an observed $m_i$ at time $t_n$ and the corresponding computed $m_i$ using equation8. We use the same error evaluation metrics of MAE, MAPE, MSE, RMSE, MedAE, and max-error measured from $\varepsilon_{SVR}$ for identified non-linear relationships.

Algorithm 2, line 4 measures NMI for each pair of metrics for every edge. Then, lines 8-10 identify and characterize non-linear relationships.

When applied to the fictive example, $r^2$ and NMI matrices are calculated for each edge of the CPG over the $\mathcal{P}$ matrices. For example, in Figure 7, in Step I, $r^2$ and NMI matrices are calculated based on the $\mathcal{P}$ metric matrix for edge $a \rightarrow b$. For real cases, we need a large sample size to confidently identify the type of relationship between the metrics. In Figure 7, high linear correlation coefficients are in green colour and high NMI values are in blue. In Step II, two linear relationships and one non-linear relationship are identified according to $r^2$ and NMI. As shown in Step II, each relationship is characterized by a regression function.

Finally, we annotate the CPG by the functions that characterize the relationships between the metrics (Line 11) in Algorithm 2. Figure 7, Step III, shows the annotated edge $a \rightarrow b$ sowing the linear and non-linear relationships. Other edges of the CPG are annotated similarly.

### 3.4 Detecting Anomalies Using the Annotated CPG

To detect an anomaly during the system execution, we collect a distributed trace (we refer to this trace as a test trace) along with profiling metrics as the system is running. Next, we follow the same steps of our method to build a CPG from the test trace and annotate it by labelling the edges with the profiling metrics. We use the term test CPG to refer to the CPG extracted from a test trace. Note that we do not need to model the linear and non-linear correlations between the metrics. For the test trace, we only need the value of the metrics. In other words, the profiling metric matrix $P$ 1 corresponding to the test trace consists of one row only. Figure 8 shows the steps for detecting anomalies using the CPG. In this figure, a fictive CPG is used for testing in the dashed-blue rectangle while an edge of $a \rightarrow b$ has been annotated with its corresponding profiling metric as an example. In real cases, all edges are annotated with their profiling metric matrix.

We compare the *test CPG* against the *baseline CPG*, which characterizes the normal behaviour of the system (see Figure 8). To achive this, we turn to graph matching. The goal is to determine whether the test CPG is an edge-inducing sub-graph of the general CPG or not. An edge-induced sub-graph includes a subset of edges and their incident vertices. We use an extended version of the VF2 algorithm [72] to check if the baseline CPG contains a sub-graph that is isomorphic to the test CPG. The test CPG is isomorphic to the general CPG if and only if there is an isomorphism mapping for edges issued from the same vertices. Since our CPG graphs are directed graphs, we use directed sub-graph matching.

In addition, we compare the test CPG against the baseline CPG to make sure that the metrics match. In this step, we check whether common edges' annotations are matched together. Edges' annotation in the test CPG is a profiling metric matrix. These matrices are tested against liner and non-linear functions annotated on the baseline CPG's edges. We check to see if the profiling metric values at the test time satisfy the linear and non-linear metrics relationship functions. To identify the degree by which a metric in the test CPG matches the functions on the baseline CPF, we need to define the margin of accepted error. As mentioned in Section 3.3.2, in this paper, we analyze six different error evaluation metrics computed for each $\epsilon_r, \epsilon_{SVR}$ vectors of linear and non-linear relationships. These metrics specify the accepted deviation margin for the profiling metric values at the test time. We will discuss the impact of the margin values on anomaly detection accuracy in Section 4.

Figure 8 shows the baseline CPG built from the fictive traces t1-t3 in a green rectangle. The edge of $a \rightarrow b$ has been annotated with the metric relationship functions and error evaluation metric as an example. Checking the test CPG against the baseline CPG includes I) checking if the test CPG is an edge-induce sub-graph of the baseline CPG; and II) if for each common edge, the relationships between the metrics on the test CPG match the annotated function on the baseline CPG. In this case, the test CPG is not a sub-graph of the baseline CPG because of node "f". Secondly, metric

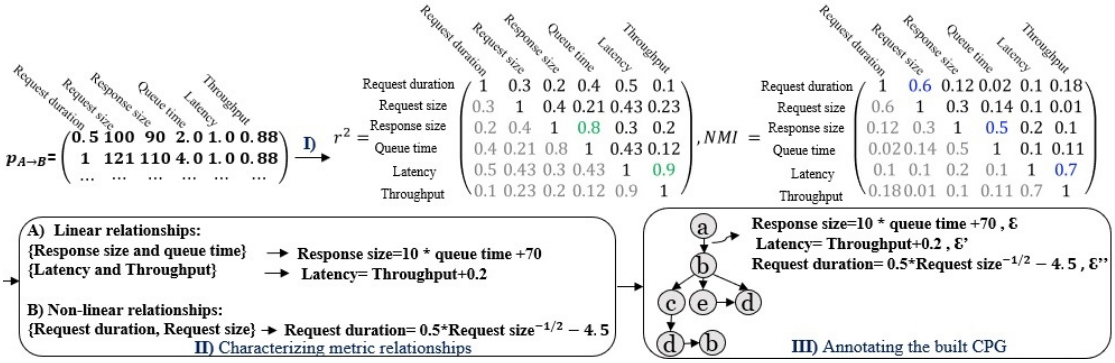


Fig. 7. Modelling metric relationships and Annotating CPG. I), $r^2$ and NMI are calculated for each pair of metrics. Next, in step II) identified relationships are characterized, and III) the CPG is annotated with the characterized relationships

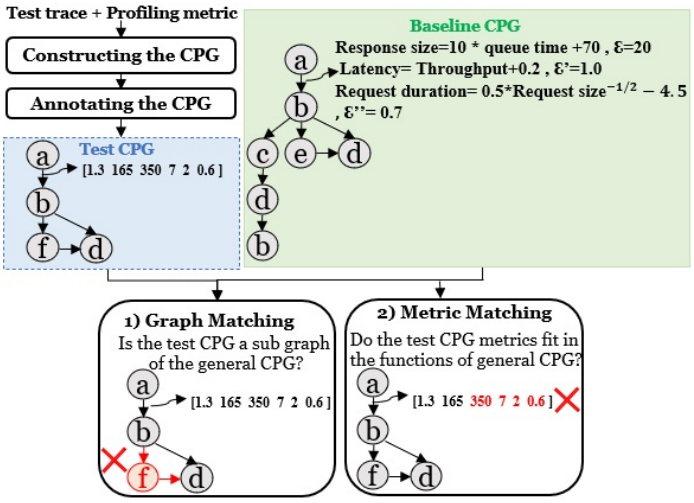


Fig. 8. Anomaly detecting process using ServiceAnomaly. A test trace and observed metrics are modelled in a test CPG. Then, the test CPG is compared with the general CPG using two steps of graph matching and metric matching

checking for the annotated instance edge of $a \rightarrow b$ results in violating two functions out of three:

$$\begin{cases} \text{Response size, queue time:} |350 - (10 * 7 + 70)| \not\leq 20 \\ \text{Latency, Throughput:} |2 - (0.6 + 0.2)| \not\leq 1.0 \\ \text{Request duration, Request Size:} |1.3 - (0.5 * 165^{0.5} - 4.5)| \leq 0.7 \end{cases}$$

We perform metric checking for all common edges between the test CPG and the baseline. Finally, according to Figure 8, any violation in graph checking and metric checking is reported to the analyst as an anomaly. We also highlight violations in the baseline CPG for further anomaly analysis. Otherwise, if the test CPG matches the baseline CPG, the test trace is recognized as a normal trace.

### 3.5 Analyzing ServiceAnomaly usage process and complexity

Figure 9 shows how our approach can be used in practice. First, the user needs to configure the microservice system to provide the necessary telemetry data for ServiceAnomaly. Our proposed architecture for data collection is detailed in Section 3.1. In this paper, we deployed microservices on Kubernetes, where Prometheus and Jaeger were added as monitoring systems into the Kubernetes cluster. However, ServiceAnomaly can be integrated with alternative architectures that offer the essential telemetry data.

To enable data collection, the user needs to a) inject Istio agents into the running Kubernetes pods to enable the data collection process, b) deploy Prometheus and Jaeger into the Kubernetes clusters and configure them to collect data from Istio agents, and c) optionally, if needed, perform instrumentation on the microservices. After the installation of Prometheus and Jaeger, additional instrumentation might be necessary for users looking to collect customized profiling metrics (e.g., the CPU usage of services) or customize span specifications. For example, Next, the collected data is acquired by ServiceAnomaly, shown in the dashed box in Figure 9, which is used to build the CPG and profiling metrics models.

During the test time, which we refer to as online setup, the user collects a test trace along with its corresponding profiling metrics using the Jaeger and Prometheus dashboards or by querying their APIs. The data collected during this test phase is then fed to ServiceAnomaly. Within the online setup, constructing the CPG, annotating the CPG, and graph matching are carried out to generate the output including the test annotated CPG and then the anomaly detection result. If an anomaly is detected, the nodes and edges that deviate from the baseline CPG, i.e., unseen nodes and edges, are highlighted. Additionally, any violated linear and nonlinear relationships are also flagged in the test CPG, along with the degree of violation for each relationship. The process of analyzing a test trace against the baseline annotated CPG can be repeated for upcoming test traces.

To analyze the computational complexity of ServiceAnomaly, we break it down into distinct computational functions. As shown in Figure 9, ServiceAnomaly operates in both offline and online phases. The offline phase of ServiceAnomaly includes a) constructing the Context Propagation Graph, b) modelling metric relationships, and c) constructing the Annotated Context Propagation Graph.

Constructing Context Propagation Graph explained by Algorithm 1 involves performing several functions for each

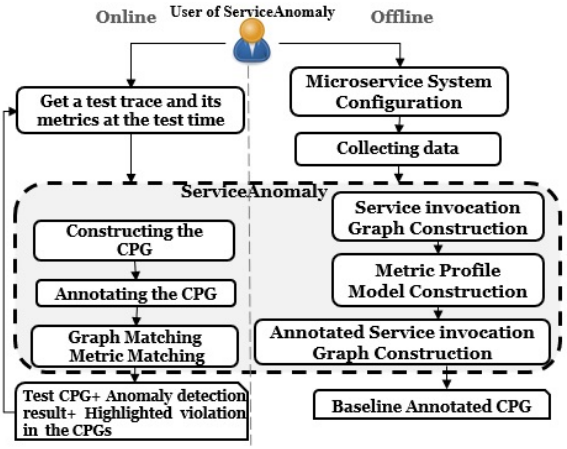


Fig. 9. User interaction with ServiceAnomaly

trace of the collected set of $T$ traces. These functions include Construct-Rooted-Call-Trees, Merge-by-Union, and Convert-to-DAG. The Construct-Rooted-Call-Trees function exhibits linear complexity, reliant on the span count within each trace. Moreover, the computation complexity of the Merge-by-Union function is derived by traversing the input rooted trees. We use a hash table structure to search through rooted trees capping the complexity of merging rooted trees at the complexity of converting them to hash maps—this conversion's complexity is linked to rooted tree size. In the worst case scenario, if $S$ denotes the service count within the longest trace and there is a call between each pair of services, the rooted tree results in $\binom{S}{2}=\frac{S(S-1)}{2}$ edges, called the tree size. Therefore, the complexity of the Merge-by-Union is primarily dictated by $O(S^2)$. For the Convert-to-DAG function, we adopt a similar strategy proposed by Hamou-Lhadj et al. [8], using a hash table to maintain certificates and signatures. According to Hamou-Lhadj et al. [8], if a tree's degree (the count of children for any node) remains bounded by a constant, the Convert-to-DAG function performs in linear time. As a result, the computation complexity of the Constructing Context Propagation Graph step drives from $O(TS^2)$ in which $T$ represents the trace count and $S$ is the number of services.

Algorithm 2 explains the steps related to modelling metric relationships and constructing the Annotated Context Propagation Graph of ServiceAnomaly in the offline setup. This algorithm requires computations of $r^2$ (equation 3), $NMI$ (equation 6), regression function (equation 5), and non-linear SVR regression function (equation 8) for each edge within the baseline CPG. The computation complexity of $r^2$ is directly proportional to the number of data points within the amassed profiling metrics, resulting in a linear function. The computation of $NMI$—as defined by Danon et al. [73]—exhibits a complexity of $O(n + K^2)$, where $n$ represents the data point count of the profiling metrics, and $k$ signifies the number of distinct categories or clusters. According to documentation [74], finding the regression function has linear complexity while the non-linear SVR regression function complexity is more than quadratic with the number of samples, approximated by Hui et al. [75] at $O(n^3)$. Therefore if the baseline CPG consists of $E$ edges, the dominant computation complexity of Algorithm 2 is $O(En^3)$.

The computational complexity of the online phase of ServiceAnomaly, which is detecting anomalies using the annotated CPG, comprises several tasks: Constructing the CPG for the test trace, Annotating the CPG, Graph Matching and Metric Matching. Given that a test trace with a constant number of spans, the complexities of all tasks are overshadowed by the task of graph matching. The graph matching algorithm for a graph with s' nodes has complexities ranging from $O(s'^2)$ to $O(s'!s')$ [76]. However, as shown by Luks [77], graph-matching algorithms demonstrate polynomial complexity in scenarios involving graphs with tree-like structures and bounded node valence. This finding is especially relevant to our study, in which a test CPG derived from a request inherently possesses a tree-like configuration with limited inter-nodal connections. In the specified context of our study, the test CPGs are characterized as labelled, directed trees sharing identical parts with the general CPG. In fact, both normal and abnormal test CPGs are expected to differ from the general CPG only in certain segments. Moreover, these CPGs are structured as Directed Acyclic Graphs (DAGs), wherein common subtrees are encapsulated singularly, thereby eliminating redundant embeddings of the entire test CPG within the general CPG. These characteristics contribute to the feasibility of graph matching, enabling efficient pruning of the search space and swift identification of matching nodes between the test CPG and the general CPG.

## 4 Experimental Setup

In this section, we explain the design of the experiment that we used to evaluate our ServiceAnomaly approach and to answer the following research questions:

- **RQ1 -** How accurate is ServiceAnomaly at detecting anomalies?
- **RQ2 -** How can the ServiceAnomaly approach be used to reason about anomalies?
- **RQ3 -** What are the limitations of ServiceAnomaly in detecting anomalies?

### 4.1 Target Systems

We use two open-source microservice applications, TeaStore[1] and TrainTicket[2]. TeaStore was developed as a benchmark to conduct experiments with performance modelling techniques. The tool emulates a tea web store for automatically-generated tea products [37]. TeaStore has six microservices including WebUI, Image-Provider, Auth, Persistence, Recommender, and a Registry service. TrainTicket is one of the academic largest open-source microservice systems to our knowledge with over 30 microservices and many service dependencies. Examples of microservices of TrainTicket include services providing ticket enquiry, reservation, food and seat selection, payment, search on travel and routes, consign, etc. [38]. TrainTicket is deployed on 56 Kubernetes pods composed of 32 logical microservices implemented in Java, node.js, Python, and Go, as well as 24 MongoDB and MySql database-related services.

1. https://github.com/DescartesResearch/TeaStore
2. https://github.com/FudanSELab/train-ticket/wiki

### 4.2 Data collection

To collect distributed traces and profiling metrics that characterize the normal behaviour of TeaStore and TrainTicket systems, we executed the load test suites that accompany these systems. The objective is to exercise as many execution paths as possible. For TeaStore, we used JMeter [78] to run the load tests. We simulated 10,000 user requests with different numbers of concurrent active users to test the system under different stress load scenarios. Finally, we used a sample of more than 3,000 successful distributed traces and 9,160 rows of six metrics collected in a one-minute window. TrainTicket comes with 15 load test scenarios written in Python and JavaScript, which we executed using a different number of concurrent active users. We collected more than 5,000 traces and 5,410 rows of metrics for TrainTicket. We used four scenarios for injecting faults into the systems based on the literature [23], [28], [79].

**Stressing the hosts:** We generated resource stress by targeting 80%-100% of the CPU and disk capacity of the node running the target service. To do this, we used Stress-ng[3], or Linux CPU exhausting test to impose an amount of CPU on the host [28], [79].

**Service latency:** We injected a 15-second latency for $\%80$ of HTTP traffic passing through the target service. We use Kiali[4] to add a rule to the Istio service mesh and control traffic transmitting across the target service. Using Kiali, we add a 15-second latency to the Istio agent running inside the target service pod. The Istio agent applies the latency to the testing service traffic [23], [28], [79].

**Service timeout:** Network connection failures might cause an error or failed request instead of a delay. To induce request failure, we used Kiali and instrument Istio traffic routing rules to inject a timeout in traffic passing through the target service. We added 5 seconds timeout for the target service, which means the target service stops answering the requests if they take longer than 5 seconds and a timeout error is returned. To trigger the timeout, we injected a 10-second latency in one of the testing service's dependencies. All injected rules are applied to the passing traffic by Istio [79].

**Service hangup:** To simulate a dead process or a service hangup, we pause the pod running the target service. Killing a pod is simulated by scaling down the Kubernetes deployment with zero replicas of the testing service's pod [23], [28].

At the test time, we execute 2 to 3 requests per second for each target system. To collect normal test traces, we continuously monitor the performance of the target system using Kiali to ensure that the system is operating as normal. For collecting anomaly traces, we inject faults into individual services, one at a time. The system performance is monitored before and after injecting a fault. To make sure that the collected trace contains a fault, we use Kiali to check the changes in the execution of the system that are caused by the fault. Figure 10 shows examples of changes in the system execution after injecting a fault.

3. https://wiki.ubuntu.com/Kernel/Reference/stress-ng
4. https://kiali.io/

In total, we injected 15 faults into TeaStore and 32 in TrainTicket. In addition to the data collected after fault injection into the target systems, we also gathered an equal amount of normal traces and profiling metrics. The collected data from normal and faulty requests are used together as our test data in evaluating our method. Table 2 shows the number of collected traces and profiling metric rows that are used to construct the baseline annotated CPG, as well as the number of normal and faulty traces used to evaluate the effectiveness of ServiceAnomaly to detect anomalies.

TABLE 2
Collected data for each case study

| Systems | Training data to construct baseline CPG | | Normal and anomaly traces used for testing | |
|---|---|---|---|---|
| | Traces | Metrics | Normal Traces | Anomaly Traces |
| TeaStore | 3,000 | 9,160 | 15 | 15 |
| TrainTicket | 5,000 | 5,410 | 32 | 32 |

### 4.3 Building the baseline annotated CPGs

We built the baseline annotated CPG for each system using the collected distributed traces and profiling metrics. Table 3 shows the number of nodes, edges, and an average number of linear and non-linear relationships per edge in the CPG. The maximum number of all possible relationships for each edge of the CPG is the number of combinations of $m$ metrics considering two metrics at a time that is $\frac{m!}{2!(m-2)!}$. In our case, at every edge, at most 15 metric relationships are possible to be characterized for each linear and non-linear type of relationship. However, it is clear that we have fewer numbers of relationships since there might not be a relationship between each pair of metrics or the relationship might not be significant to be characterized.

As mentioned in Section 3.3.2, we adopt the same process proposed by Jiang et al. [64] to interpret $r^2$ and NMI values for each target system and find linear and non-linear relationships. We consider $r^2 > 0.6$ as an indicator of a strong linear relationship as suggested by Jiang et al. [64]. To find the best $t$ threshold for NMI values, for each target system, we compute $r^2$ and NMI for all pairs of metrics and compare NMI to $r^2$. Figure 11 shows that for TeaStore when $r^2 \geq 0.6$ and there is a strong linear relationship between metrics, NMI value is at least 0.6. We know that for a linear relationship both $r^2$ and NMI are at their high values. We consider NMI $> 0.6$ as the threshold for determining a high NMI value for TeaStore. Moreover, we know that correlated data with a non-linear relationship shows a low $r^2$ and high NMI value. Therefore, when $r^2 < 0.6$ and NMI $\geq 0.6$ we identify non-linear relationships. According to Figure 11, the threshold for the TrainTicket system is NMI = 0.47 and for TeaStore is NMI=0.6.

TABLE 3
CPG characteristics for each system

| SUT | #Nodes | #Edges | Avg #Linear relations / edge | Avg #Nonlinear Relations / edge |
|---|---|---|---|---|
| TeaStore | 6 | 12 | 3.6 | 1.5 |
| TrainTicket | 32 | 83 | 4.0 | 2.95 |

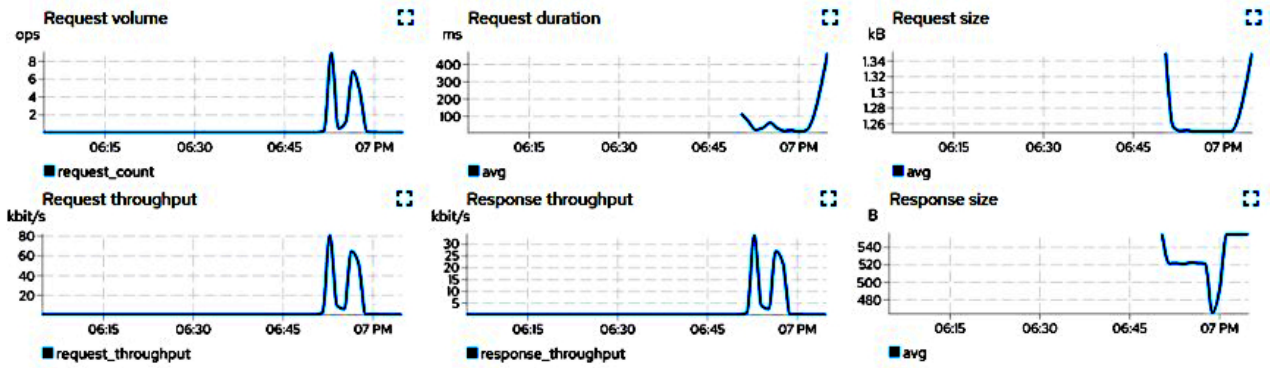


Fig. 10. Service performance change at the time of injected faults, at 7:00 PM

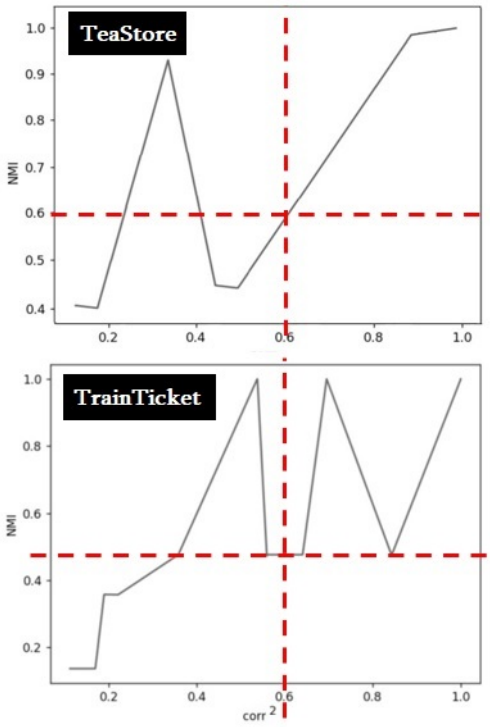


Fig. 11. $r^2$ and NMI measured for pair of metrics in TeaStore and TrainTicket

Figure 12 shows a partial view of the baseline CPG for TeaStore. The CPG is on the right side and the metric relationship functions for each edge are listed on the left side of the figure. Figure 12 shows five linear relationships. The figure also shows one non-linear relationship for the edge connecting the services "teastore-auth" and "teastore-persistence". To clearly characterize the relationships, a partial heat map for three metrics collected over these two services is shown in Figure 13. As shown, there is a high correlation between the latency and request size, highlighted by a circle, which results in a linear relationship between these metrics. Similarly, a high NMI value and low correlation between the latency and queue time, highlighted by squares in the figure, results in a non-linear relationship between latency and queue time. The low correlation coefficient and NMI values for request size and queue size indicate that these metrics do not have a relationship.

### 4.4 Evaluation Metrics

To assess the accuracy of our method, we measure the precision, recall, and F1 score. These measures are used to describe a test accuracy which detects the presence or absence of a condition [80]. In our case, the positive condition, P, is defined as the number of real anomalies, i.e. test traces collected when we injected fault/stress scenarios, and the negative condition, N, is the number of normal traces.

According to the error of the anomaly detection, different outcomes including true (T) and false (F) results can be concluded.

- True Positive (TP): An anomaly trace detected as anomaly
- False Positive (FP): A healthy trace detected as anomaly
- False Negative (FN): An anomaly trace detected as healthy
- True Negative (TN): A healthy trace detected as healthy

The precision reflects the number of correct positive predictions out of all positive predictions made, which include those falsely classified as anomalies, and it is measured as:

$$Precision = \frac{TP}{TP + FP}$$

The recall quantifies the number of positive class predictions made out of all positive cases in the data set, calculated as:

$$Recall = \frac{TP}{TP + FN}$$

F1-score is a single score that balances precision and recall and is defined as the harmonic mean of Precision and Recall:

$$F1 - score = 2 * \frac{Precision * Recall}{Precision + Recall}$$

## 5 Results

### 5.1 RQ1. How accurate is ServiceAnomaly at detecting anomalies?

The accuracy of ServicAnomaly can be affected by the choice between different error evaluation metrics since they determine the expected amount of deviation in anomaly

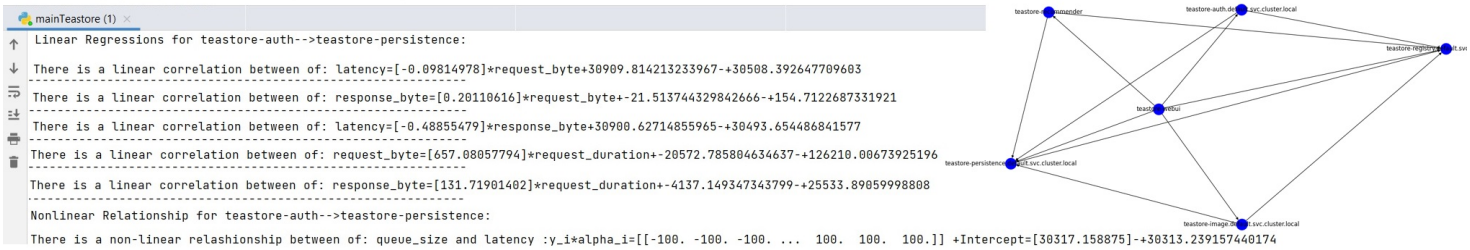


Fig. 12. A partial baseline annotated CPG for TeaStore

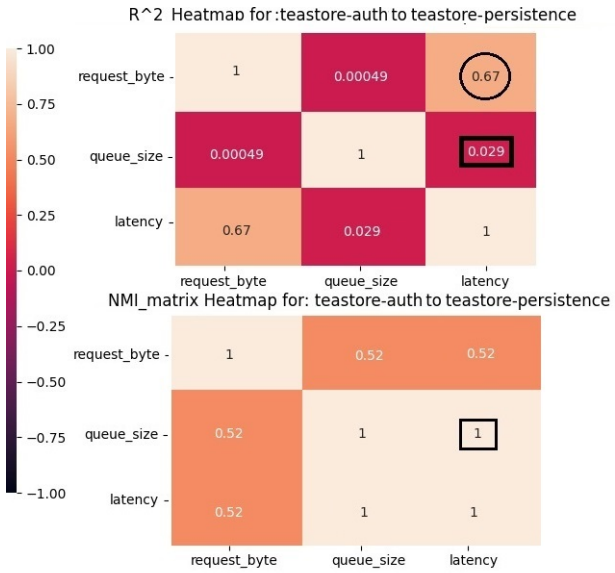


Fig. 13. A partial heat-map for an edge of TeaStore

detection. Table 4 shows precision, recall, and F1-score for both TeaStore and TrainTicket with the six different error evaluation metrics discussed in Section 3.3.2. As shown in Table 4, our approach achieves an F1-score of up to 85% for TeaStore and 86% for TrainTicket.

The choice between error evaluation metrics depends on the specific characteristics of the dataset. In our case, we have observed that using the RMSE error evaluation metric to assess the accuracy of the model yields the best results compared to other error evaluation metrics for both systems. This may be due to the fact that RMSE is sensitive to outliers and therefore penalizes large errors, resulting in a model that provides better classification. This is contrasted with MAPE, MedAE, and MAE, which are not as sensitive to outliers as RMSE [81]. Table 4 shows that models obtained using these error evaluation metrics have a very high recall (100% for TeaStore and up to 94% for TrainTicket), and a low precision (50% to 60% for TeaStore) and (48% to 58% for TrainTicket), meaning that they introduce a considerable number of false positives (i.e., healthy traces detected as anomalies). We also found that out of the 15 anomaly traces of TeaStore, 7 were detected using the CPG matching alone, and 7 were detected with metric matching with RMSE. The remaining anomaly trace was not detected as an anomaly, which explains a recall of 93%. Out of the 7 anomaly traces detected by CPG matching, 2 were affected by an injected

TABLE 4
Accuracy of ServiceAnomaly when applied to TeaStore and TrainTicket systems with different error evaluation metrics

| Target systems | Error | Precision | Recall | F1-score |
|---|---|---|---|---|
| TeaStore | MAPE | 0.50 | 1.00 | 0.67 |
| | MedAE | 0.54 | 1.00 | 0.70 |
| | MAE | 0.60 | 1.00 | 0.75 |
| | RMSE | 0.78 | 0.93 | 0.85 |
| | MSE | 0.67 | 0.67 | 0.67 |
| | max_error | 0.89 | 0.53 | 0.66 |
| TrainTicket | MAPE | 0.48 | 0.94 | 0.64 |
| | MedAE | 0.54 | 0.94 | 0.69 |
| | MAE | 0.59 | 0.91 | 0.72 |
| | RMSE | 0.79 | 0.94 | 0.86 |
| | MSE | 0.53 | 0.88 | 0.68 |
| | max_error | 0.77 | 0.72 | 0.74 |

service latency, 3 were affected by an injected service timeout, and 2 were affected by an injected service hangup scenario. Furthermore, the 7 detected anomaly traces through metric matching included 3 collected after injecting a stress scenario, 2 after injecting a latency, 1 after an injected service timeout, and 1 after an injected service hangup. We did a similar analysis on TrainTicket and found that out of 32 anomaly traces, 5 were detected using CPG matching and 25 were detected using metric matching with RMSE. Among the 5 detected anomaly traces using CPG matching, 3 were attributed to an injected service timeout scenario, one to an injected service latency, and one to an injected service hangup. Similarly, with metric matching, 25 anomaly traces were detected, where 8 were collected after injecting a service latency scenario, 7 after injecting a service timeout, 6 after stressing the host, and 4 after injecting a service hangup.

Based on our analysis, we can conclude that both CPG matching and metric matching are needed to detect a variety of faults in microservice applications.

**RQ1.** How accurate ServiceAnomaly is at detecting anomalies?
**Finding:** The results show that ServiceAnomaly can detect anomalies with an F1-score up to 85% for TeaStore and 86% for TrainTicket. The RMSE error evaluation metric yields a more accurate model for both systems compared to other error evaluation metrics. We also showed that the combination of CPG and profiling metrics is an effective way to detect different types of faults.

### 5.2 RQ2. How can the ServiceAnomaly approach be used to analyze anomalies?

In this question, we show how our approach can help developers to better understand the anomalies by exploring the annotated CPG.

In the following, we select two examples of anomaly traces from the TeaStore and TrainTicket systems respectively that were detected in the previous section and examine the causes of the anomalies using the CPG and the metrics model.

#### *5.2.1 Scenario 1: TrainTicket anomaly trace*

Figure 14-(a) shows a portion of a TrainTicket anomaly trace using Jaeger. We could not show the entire trace, which contains 28 spans showing a record of the various services involved in reserving a ticket using the TrainTicket system. To reserve a ticket, in a normal trace, first, the ”ts-ui-dashboard” calls the ”ts-preserve-service”. Then, the ”ts-preserve-service” span invokes a series of services, including ”ts-security-service”, ”ts-contact-service”, ”ts-travel-service”, and eight other services, in a sequential manner to complete the request.

Figure 14-(b) shows part of the TrainTicket's baseline CPG including the context propagation paths invoked as a result of a ticket reservation request. However, in this scenario, we simulated a timeout fault scenario into ”ts-preserve-service” using Kiali as explained in Section 3.1. We used the fault injection wizard on ”ts-preserve-service” and created a rule that triggers a timeout error after 5 seconds if the service does not receive a response within that time frame. To activate this rule, we introduced a latency to the ”ts-contact-service” using the same wizard in Kiali. This latency causes delays in the traffic forwarding from ”ts-preserve-service” to ”ts-contact-service”, which then triggers the injected timeout rule. As a result of the injected fault, the ”ts-preserve-service” span throws an error propagated to its caller, shown in Figure 14-(a), lines 1-2, 5-6, and 9. Moreover, in comparison to the observed normal paths in the baseline CPG, ”ts-preserve-service” has been repetitively called in lines 5 and 9 without any nested child.

To detect the anomalous trace using our approach, the test trace, Figure 14-(a), collected at the time of injecting the timeout into the ”ts-preserve-service” service is taken and checked against the baseline CPG. As a result, the developer will be presented by the test CPG shown in Figure 15. As we can see the matching algorithm has highlighted the unmatched nodes and edges in red colour. In the test CPG, there is a ”ts-preserve-service” node with no child that is because of repetitive calls of ”ts-preserve-service” in lines 5 and 9 of the test trace (14-(a)). Parsing the contextual information propagated through the spans of the test trace shows that one of the recalled ”ts-preserve-service” has missed contextual information. Our algorithm assigns an empty reference to this service, shown by a node with no name in Figure 15. The graph matching step finds this node and its connection unmatched with the general CPG. In addition, the metric matching step identifies linear and non-linear metric relationships that deviate from the normal metric relationships. In this case, there are eight linear and three non-linear metric relationships on seven edges that deviate from the metric relationships on the edges of the baseline CPG. These metric relationships are highlighted and reported to the developer along with the metric types, the expected value range for each metric, and the amount of deviation. Figure 15 shows part of the reported identified unmatched metric relationships.

#### *5.2.2 Scenario 2: TeaStore anomaly trace*

Figure 16-(a) shows one of the TeaStore traces that was flagged as an anomaly by our approach. This trace includes three events each of which has two spans. First, ”teastore-webui” calls ”teastore-persistence”, then it invokes a service of ”teastore-image”, and finally, ”teastore-webui” calls ”teastore-auth” to fulfil the user request. Figure 16-(b) presents a partial view of the baseline CPG for TeaStore. This graph includes the context propagation path from ”teastore-webui” to each of the other three called spans that was observed in the trace shown in Figure 16-(a). The annotations on each edge of Figure 16-(b) briefly describe the linear (L) and non-linear (NL) relationships between the profiling metrics for each edge, as well as the specific profiling metrics involved in each relationship.

To test our approach, we injected a latency into the ”teastore-recommender” service using Kiali as explained in Section 4.2. To inject a latency into the ”teastore-recommender”, we configure a fault injection rule where 80% of requests forwarding to the ”teastore-recommender” will be delayed 15 seconds. The test trace shown in Figure 16-(a) was collected after injecting this fault into ”teastore-recommender”.

As we can see in Figure 16-(c), the anomaly was caused by a violation to the metric relationships. In this scenario, the test CPG is a subgraph of the baseline CPG. Therefore, graph matching did not identify the trace as an anomaly. However, the injected fault into the trace has been detected by metric matching. In total, one linear and two non-linear relationships have deviated from this trace. Using this information the developer can do more investigations about the reason for the detected anomaly. In this scenario, the relationship between some metrics in the test CPG did not match the expected relationships on all three edges connected to the ”teastore-webui”. The developer can analyze the ”teastore-webui” service and its dependencies to uncover the causes of the anomaly. For example, comparing the test CPG (Figure 16-(c)) with the baseline CPG (Figure 16-(b)) shows that in addition to three called spans by ”teastore-webui” in the test CPG, the ”teastore-recommender” and ”teastore-registry” are also connected to the ”teastore-webui” and can be the cause of the detected anomaly. This information can not be learnt from the collected test trace. Therefore, given the result of our approach, the developer is able to learn about the reason for the detected anomaly, ”teastore-recommender” in this case.

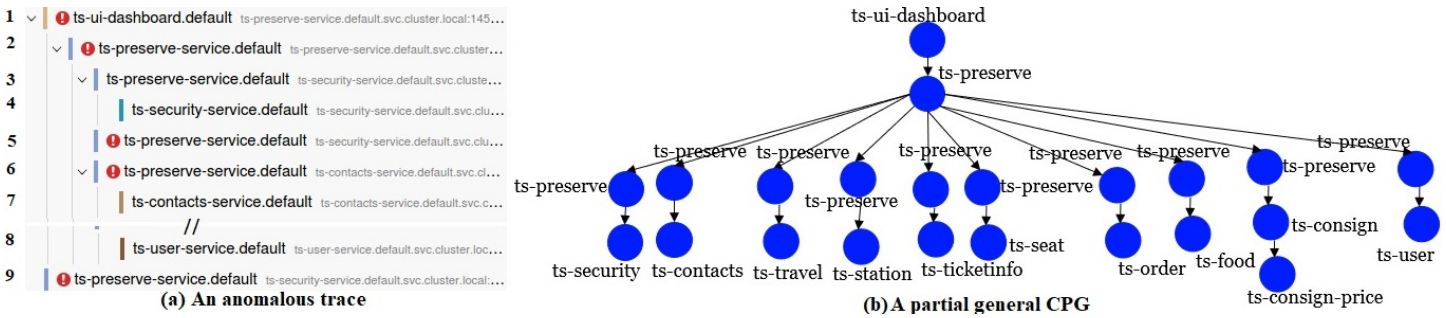


Fig. 14. (a)Scenario 1- An anomalous TrainTicket trace detected by ServiceAnomaly, (b) A partial Trainticket general CPG

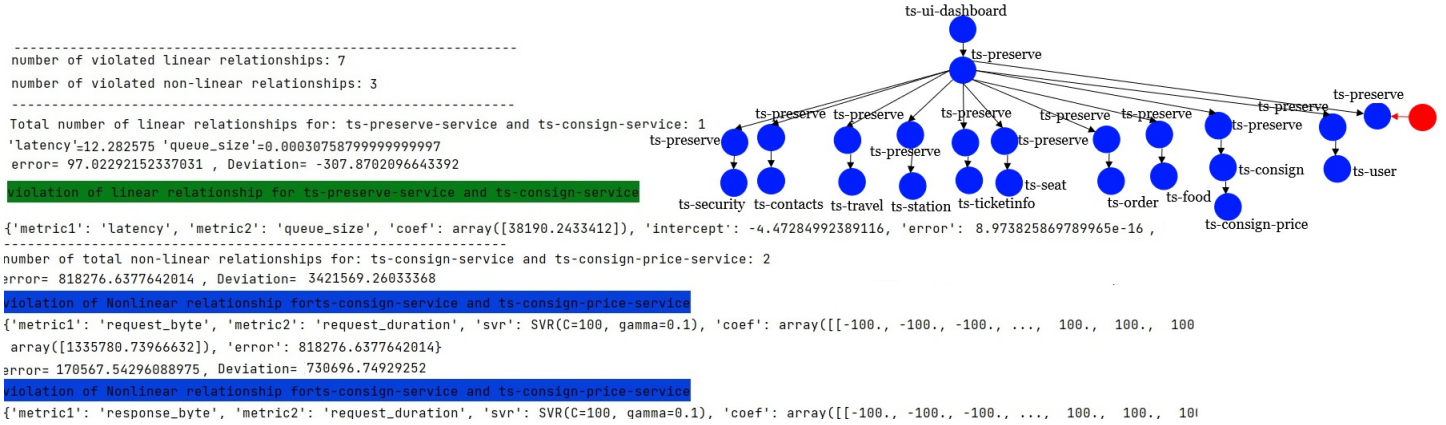


Fig. 15. Scenario 1- Test CPG result for the test trace shown in Figure 14-(a)

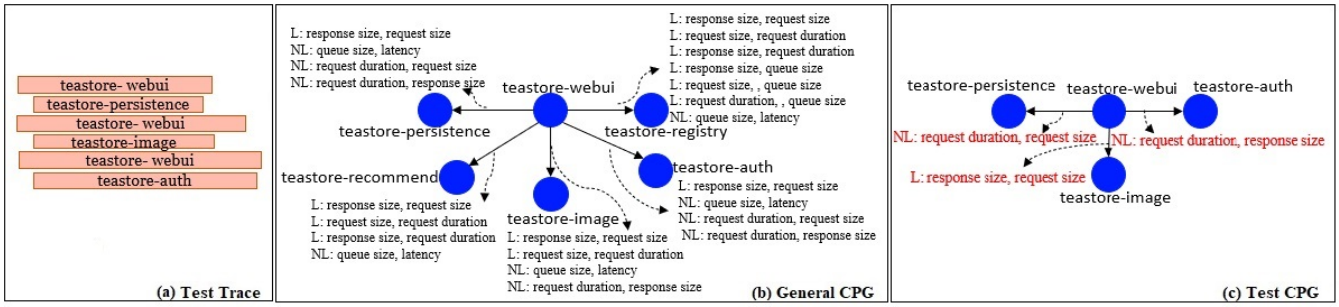


Fig. 16. Scenario2- An anomalous TeaStore trace detected by ServiceAnomaly

**RQ2.** How can the ServiceAnomaly approach be used to analyze anomalies?
**Finding:** Our approach can pinpoint places of interest where the anomalies occur using the test CPG and the baseline CPG. We illustrate this feature through two examples of anomaly traces from TeaStore and TrainTicket. Additional studies involving developers should be conducted to further assess the usefulness of our approach in root causes of anomalies.

### 5.3 RQ3. What are the limitations of ServiceAnomaly in detecting anomalies?

To identify the limitations of our approach, ServiceAnomaly, in this question, we discuss anomalies that were not detected by ServiceAnomaly and attempt to determine the causes. According to our observations, anomalies that change the CPG structure compared to the baseline CPG were successfully detected at the graph matching step. However, if the test CPG is an edge-inducing sub-graph of the baseline CPG, the anomaly might remain undetected by the metric matching.

The first reason for undetected anomalies can be found in the magnitude of deviation in the metrics affected by the anomaly. In this case, using different error evaluation

TABLE 5
Details of "teastore-webui → teastore-persistence" profiling metric relationships

| Metrics | Rel. | residual | max-error | RMSE |
|---|---|---|---|---|
| y=response size(Kb), x=request size(kb) | L | 10.9 | 25.6 | 4.7 |
| y=latency(s), x=queue time(s) | NL | $16.6 * 10^{-}3$ | 30.3 | 5.62 |
| y=request size(kb), x=duration time(s) | NL | $67.1 * 10$ | $46.9 * 10^2$ | $64.2 * 10$ |
| y=response size(kb), x=duration time(s) | NL | $20.4 * 10$ | $13.9 * 10^2$ | $18.9 * 10$ |

metrics can alter the outcome and cause the detection of the anomaly. For example, Table 5 shows profiling metric relationships for a test trace of TeaStore with an injected CPU stress that is not detected by the error evaluation metric of max-error but is detected by a smaller error evaluation metric such as RMSE.

Table 5 describes the characterized metric relationships for an edge between "teastore-webui" and "teastore-persistence". The first two columns specify the involved metrics in each characterized relationship and the type of relationship, with L denoting linear and NL denoting non-linear relationships. The third column indicates the residual value or absolute difference between the observed value of metric y and the computed value using the characterized relationships. The last two columns show the values of different error evaluation metrics, i.e., error-max and RMSE, for each metric relationship. The CPU stress can be detected when the absolute difference (third column) exceeds the expected error values (last two columns). A comparison of the third column with the expected residuals reveals that the injected CPU stress can be detected using the smaller error evaluation metric (RMSE) by one linear and two non-linear violated relationships.

The second reason for undetected anomalies is performing anomaly detection on test traces that are unaffected by the anomaly. In this case, although the injected fault scenario has changed the system performance, the injected fault has not been propagated into the collected spans of the test trace. These traces remain undetected with different experimented error evaluation metrics. For instance, Figure. 17 shows some of the performance views of Kiali for "ts-food-service.default" service in TrainTicket before and after injecting a service latency into this service. The service latency scenario has been injected at 6:26 PM, shown by a border in Figure. 17. As the monitoring dashboards show, the service performance has changed after injecting the latency. However, the collected test trace at the time of experiments does not reveal these changes. Figure. 17, shows the test trace that we collected after injecting this fault scenario in the blue colour. This test trace includes a span "ts-travel-service.default" that calls "ts-ticketinfo-service.default". Figure. 17 shows how the "ts-food-service.default" service with the injected latency is connected to the collected spans. While the "ts-food-service.default" service has been affected by the injected latency, this anomaly did not manifest itself in the test trace. Therefore, the collected test traces were missed by our approach.

We tried different test traces in this group of undetected anomalies and realized that anomalies are more likely to be detected when the collected test trace is a trace generated by requests to the affected service (i.e., the service with the injected fault) or its dependencies. Moreover, generally, collected test traces comprising more spans with a longer request propagation path are more likely to show the change caused by the injected fault.

The third group of undetected anomalies happen when the collected test trace includes spans with no relationships. It is a rare case in our experiments. However, in some cases, there might be services with low interactions with the rest of the system. Therefore, few profiling metric values can be detected for those services that it makes it challenging to identify relationships between those metrics. For example, in TeaStore, there are services of an Image provider and a Recommender calling the Persistence service only once at the system execution startup. Therefore, no relationship can be identified for them out of a few telemetry data extracted from them. Moreover, there is a possibility that there are enough collected profiling metrics for an edge but the collected metrics do not show any linear or non-linear dependency for an edge of the CPG. Therefore, collected test traces including that edge with no relationships cannot be cooperative in detecting anomalies using matching metric relationships.

**RQ3.** What are the limitations of ServiceAnomaly in detecting anomalies?
**Finding:** ServiceAnomaly cannot detect anomalies when I) The anomaly has not changed the metric values significantly to be captured by the used error evaluation metric. II) The collected test trace does not include the affected services and III) There are edges in the general CPG with no profiling metric relationship and the test trace includes spans associated with those edges.

## 6 Threats to Validity

**Internal Threats:**
The threats to internal validity concern the factors that might influence our results. To collect distributed traces and profiling metrics to characterize the normal behaviour of the system, we run both systems using the accompanying load tests. A different strategy may lead to another set of traces. However, we purposely resorted to load tests to observe as many execution paths as possible. An alternative solution would be to run the systems using execution scenarios, but this would require a good set of system features, which we do not have. Another threat to internal validity consists of the collection of anomaly traces by injecting faults into the systems. To make sure that the collected reflect the abnormal behaviour caused by the fault, we carefully observed the system to see the impact of the fault on the execution as shown in Figure 10. In addition, the profiling metrics values collected through the service mesh can be affected by the network delay. For instance, a request duration time is defined as the time between establishing a request to its completion. However, network delays may increase

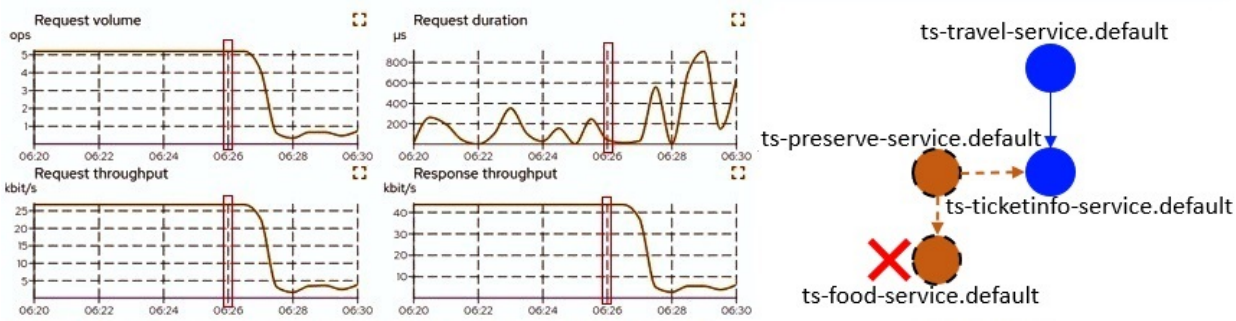


Fig. 17. An example of a service performance affected by injecting network latency at 6:26PM, A partial CPG representing the propagation path of injected fault to the services

processing time. We ignored analysing the present delay in the network since it is applied to all the metrics. However, to make sure that the profiling metrics are not affected by a long delay, we monitored our target system's health during the experiment to identify and filter any significant delays. Finally, there is a possible threats in the way we implemented ServiceAnomaly. An implementation error may have an impact on the results. To mitigate this threat, we checked the implementation very carefully and tested it on small examples. We also make the scripts and the datasets available online for verifiability[5].

**External Threats:**
The threats to the external validity of this study lie in the generalizability of the results. We evaluated our approach on two different microservices using a large number of traces. To our knowledge, the TrainTicket system is the largest open-source microservice system available. To generalize our results, we should experiment with more systems once they become available. There is also an external threat to validity related to the types of faults that were injected to generate anomaly traces. Although we covered different types of faults (service latency, stress host, etc.), we should experiment with more types of fault scenarios to claim generalizability of the approach.

**Construction Threats:**
The threats to the construction validity of our method mainly lie in modelling metric relationships. Calculating mutual information is dependent on entropy and probabilities which makes it difficult to be measured for empirical data [82]. We used the proposed binning approach for calculating entropy [83] in which the observed data is divided into k bins with $n_i$ samples in each bin and $n_i$ represents the number of occurrences of each metric value. Then, entropy is measured using the following formula for metric $m_i$:

$$entropy(m_i) = -\Sigma_{i=1}^{k} \frac{n_i}{n} log \frac{n_i}{n} \quad (10)$$

The adapted VF2 algorithm employed for graph matching is expected not to surpass polynomial time complexity, considering the specific characteristics of CPGs and the assumption that that test CPGs are expected to have identical parts with the general CPG. However, this assumption may encounter challenges under certain conditions: I) if the test CPG closely resembles the general CPG, but is not identical. i.e., when the test CPG is very similar to the general CPG in both structure and attributes, but not identical in many parts or II) the test CPG can be embedded in the general CPG in multiple ways. While the algorithm demonstrates feasibility for our tested target system, these scenarios necessitate further analysis to refine the graph-matching task. A potential solution to mitigate these complexities involves structuring anomalies, i.e. null nodes or novel patterns, as distinct patterns. By doing so, the graph-matching algorithm can specifically search for these patterns, facilitating targeted pruning based on predefined anomaly characteristics.

5. https://github.com/M-panahandeh/ServiceAnomaly

## 7 Reproduction Package

All the data used in this paper as well as the scripts can be found on Github repository using the following link: https://github.com/M-panahandeh/ServiceAnomaly.

## 8 Conclusion

Anomaly detection of microservices is a challenging task due to the complexity of these systems, the volume of logs and traces, and metrics collected during monitoring microservices. In this paper, we proposed ServiceAnomaly, a method based on modelling the context propagation as a graph, which is annotated with the linear and non-linear relationships of six profiling metrics. We evaluated our approach against two case studies of TeaStore and TrainTicket. The results show that our approach identifies traces with an accuracy of F1-score up to 86%. we also showed how ServiceAnomaly can be used to reason about anomalies, a starting point for root cause analysis. For future work, we intend to experiment with more microservice systems as they become available. We also need to integrate ServiceAnomaly in an observability tool and work with developers to examine the usefulness of this method in practice. Another important future direction is to study how the annotated CPG can be used to understand large distributed traces. A particular line of research is to examine how abstraction techniques such as those developed for traditional traces (e.g. [84]) can improve the analysis of the annotated CPG to support program comprehension, an important enabler for root cause analysis of faults. Moreover, we should investigate techniques to improve the causality and explainability of our approach. The idea is to improve ServiceAnomaly's

decisions through a feedback loop process that considers the analyst feedback. Finally, we need to study the scalability of our approach when working with large systems. For the current work, we collected traces for 3 to 7 days depending on the system to have a good coverage of the execution paths for each system. For larger systems, we may need to collect more traces, which can impact scalability.

## REFERENCES


[1] Claus Pahl. Containerization and the paas cloud. *IEEE Cloud Computing*, 2(3):24–31, 2015.

[2] Armin Balalaie, Abbas Heydarnoori, and Pooyan Jamshidi. Migrating to cloud-native architectures using microservices: an experience report. In *European Conference on Service-Oriented and Cloud Computing*, pages 201–215. Springer, 2015.

[3] Armin Balalaie, Abbas Heydarnoori, and Pooyan Jamshidi. Microservices architecture enables devops: Migration to a cloud-native architecture. *Ieee Software*, 33(3):42–52, 2016.

[4] Jacopo Soldani, Damian Andrew Tamburri, and Willem-Jan Van Den Heuvel. The pains and gains of microservices: A systematic grey literature review. *Journal of Systems and Software*, 146:215–232, 2018.

[5] Nane Kratzke and Peter-Christian Quint. Understanding cloud-native applications after 10 years of cloud computing-a systematic mapping study. *Journal of Systems and Software*, 126:1–16, 2017.

[6] A. Parker, D. Spoonhower, J. Mace, B. Sigelman, and R. Isaacs. *Distributed Tracing in Practice: Instrumenting, Analyzing, and Debugging Microservices*. O'Reilly Media, Incorporated, 2020.

[7] OpenTracing Contributors. Opentracing specification. https://opentracing.io/specification/. Accessed: February 23, 2023.

[8] Abdelwahab Hamou-Lhadj and Timothy C Lethbridge. Compression techniques to simplify the analysis of large execution traces. In *Proceedings 10th International Workshop on Program Comprehension*, pages 159–168. IEEE, 2002.

[9] Abdelwahab Hamou-Lhadj and Timothy C Lethbridge. Measuring various properties of execution traces to help build better trace analysis tools. In *Proceedings 10th International Conference on Engineering of Complex Computer Systems*, pages 559–568. IEEE, 2005.

[10] Edvin Niemi and Richard Wallin. Visualization of microservices: Mapping dependencies in a distributed architecture, 2021.

[11] Veronika Dashuber and Michael Philippsen. Trace visualization within the software city metaphor: Controlled experiments on program comprehension. *Information and Software Technology*, 150:106989, 2022.

[12] Andrew Ayers, Richard Schooler, Chris Metcalf, Anant Agarwal, Junghwan Rhee, and Emmett Witchel. Traceback: First fault diagnosis by reconstruction of distributed control flow. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 201–212, 2005.

[13] Hao Chen, Kegang Wei, An Li, Tao Wang, and Wenbo Zhang. Trace-based intelligent fault diagnosis for microservices with deep learning. In *2021 IEEE 45th Annual Computers, Software, and Applications Conference (COMPSAC)*, pages 884–893. IEEE, 2021.

[14] Guangba Yu, Zicheng Huang, and Pengfei Chen. Tracerank: Abnormal service localization with dis-aggregated end-to-end tracing data in cloud native systems. *Journal of Software: Evolution and Process*, page e2413, 2021.

[15] Richard Li, Min Du, Zheng Wang, Hyunseok Chang, Sarit Mukherjee, and Eric Eide. Longtale: Toward automatic performance anomaly explanation in microservices. In *Proceedings of the 2022 ACM/SPEC on International Conference on Performance Engineering*, pages 5–16, 2022.

[16] Tao Wang, Wenbo Zhang, Jiwei Xu, and Zeyu Gu. Workflow-aware automatic fault diagnosis for microservice-based applications with statistics. *IEEE Transactions on Network and Service Management*, 17(4):2350–2363, 2020.

[17] Sasho Nedelkoski, Jorge Cardoso, and Odej Kao. Anomaly detection and classification using distributed tracing and deep learning. In *2019 19th IEEE/ACM international symposium on cluster, cloud and grid computing (CCGRID)*, pages 241–250. IEEE, 2019.

[18] Liang Bao, Qian Li, Peiyao Lu, Jie Lu, Tongxiao Ruan, and Ke Zhang. Execution anomaly detection in large-scale systems through console log analysis. *Journal of Systems and Software*, 143:172–186, 2018.

[19] Animesh Nandi, Atri Mandal, Shubham Atreja, Gargi B Dasgupta, and Subhrajit Bhattacharya. Anomaly detection using program control flow graph mining from execution logs. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 215–224, 2016.

[20] Min Du, Feifei Li, Guineng Zheng, and Vivek Srikumar. Deeplog: Anomaly detection and diagnosis from system logs through deep learning. In *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*, pages 1285–1298, 2017.

[21] Ping Liu, Haowen Xu, Qianyu Ouyang, Rui Jiao, Zhekang Chen, Shenglin Zhang, Jiahai Yang, Linlin Mo, Jice Zeng, Wenman Xue, et al. Unsupervised detection of microservice trace anomalies through service-level deep bayesian networks. In *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*, pages 48–58. IEEE, 2020.

[22] Jasmin Bogatinovski, Sasho Nedelkoski, Jorge Cardoso, and Odej Kao. Self-supervised anomaly detection from distributed traces. In *2020 IEEE/ACM 13th International Conference on Utility and Cloud Computing (UCC)*, pages 342–347. IEEE, 2020.

[23] Sasho Nedelkoski, Jorge Cardoso, and Odej Kao. Anomaly detection and classification using distributed tracing and deep learning. In *2019 19th IEEE/ACM international symposium on cluster, cloud and grid computing (CCGRID)*, pages 241–250. IEEE, 2019.

[24] Stephen Jacob, Yuansong Qiao, Yuhang Ye, and Brian Lee. Anomalous distributed traffic: Detecting cyber security attacks amongst microservices using graph convolutional networks. *Computers & Security*, 118:102728, 2022.

[25] Li Wu, Johan Tordsson, Jasmin Bogatinovski, Erik Elmroth, and Odej Kao. Microdiag: Fine-grained performance diagnosis for microservice systems. In *2021 IEEE/ACM International Workshop on Cloud Intelligence (CloudIntelligence)*, pages 31–36. IEEE, 2021.

[26] Yuan Zuo, Yulei Wu, Geyong Min, Chengqiang Huang, and Ke Pei. An intelligent anomaly detection scheme for microservices architectures with temporal and spatial data analysis. *IEEE Transactions on Cognitive Communications and Networking*, 6(2):548–561, 2020.

[27] Areeg Samir, Nabil El Ioini, Ilenia Fronza, Hamid R Barzegar, Van Thanh Le, and Claus Pahl. Anomaly detection and analysis for reliability management clustered container architectures. *International Journal on Advances in Systems and Measurements*, 12(3\&4):247–264, 2020.

[28] Jinjin Lin, Pengfei Chen, and Zibin Zheng. Microscope: Pinpoint performance issues with causal graphs in micro-service environments. In Claus Pahl, Maja Vukovic, Jianwei Yin, and Qi Yu, editors, *Service-Oriented Computing*, pages 3–20, Cham, 2018. Springer International Publishing.

[29] Zilong He, Pengfei Chen, Xiaoyun Li, Yongfeng Wang, Guangba Yu, Cailin Chen, Xinrui Li, and Zibin Zheng. A spatiotemporal deep learning approach for unsupervised anomaly detection in cloud systems. *IEEE Transactions on Neural Networks and Learning Systems*, 2020.

[30] Chuanjia Hou, Tong Jia, Yifan Wu, Ying Li, and Jing Han. Diagnosing performance issues in microservices with heterogeneous data source. In *2021 IEEE Intl Conf on Parallel & Distributed*

*Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking (ISPA/BDCloud/SocialCom/SustainCom)*, pages 493–500. IEEE, 2021.

[31] Areeg Samir and Claus Pahl. Dla: Detecting and localizing anomalies in containerized microservice architectures using markov models. In *2019 7th International Conference on Future Internet of Things and Cloud (FiCloud)*, pages 205–213. IEEE, 2019.

[32] Iman Kohyarnejadfard, Mahsa Shakeri, and Daniel Aloise. System performance anomaly detection using tracing data analysis. In *Proceedings of the 2019 5th International Conference on Computer and Technology Applications*, pages 169–173, 2019.

[33] Lun Meng, Feng Ji, Yao Sun, and Tao Wang. Detecting anomalies in microservices with execution trace comparison. *Future Generation Computer Systems*, 116:291–301, 2021.

[34] Peng Xu, Xue Gao, and Zhongbao Zhang. Graph neural network-based anomaly detection for trace of microservices. *Available at SSRN 4111928*, 2022.

[35] Qiang Fu, Jian-Guang Lou, Yi Wang, and Jiang Li. Execution anomaly detection in distributed systems through unstructured log analysis. In *2009 ninth IEEE international conference on data mining*, pages 149–158. IEEE, 2009.

[36] Sasho Nedelkoski, Jorge Cardoso, and Odej Kao. Anomaly detection from system tracing data using multimodal deep learning. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, pages 179–186. IEEE, 2019.

[37] Jóakim von Kistowski, Simon Eismann, Norbert Schmitt, André Bauer, Johannes Grohmann, and Samuel Kounev. TeaStore: A Micro-Service Reference Application for Benchmarking, Modeling and Resource Management Research. In *Proceedings of the 26th IEEE International Symposium on the Modelling, Analysis, and Simulation of Computer and Telecommunication Systems*, MASCOTS '18, September 2018.

[38] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chao Ji, Wenhai Li, and Dan Ding. Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study. *IEEE Transactions on Software Engineering*, 47(2):243–260, 2018.

[39] Bowen Li, Xin Peng, Qilin Xiang, Hanzhang Wang, Tao Xie, Jun Sun, and Xuanzhe Liu. Enjoy your observability: an industrial survey of microservice tracing and analysis. *Empirical Software Engineering*, 27(1):1–28, 2022.

[40] Md. S. Islam, Wael Khreich, and Abdelwahab Hamou-Lhadj. Anomaly detection techniques based on kappa-pruned ensembles. *IEEE Transactions on Reliability*, 67(1):212–229, 2018.

[41] Wael Khreich, Babak Khosravifar, Abdelwahab Hamou-Lhadj, and Chamseddine Talhi. An anomaly detection system based on variable n-gram features and one-class svm. *Information and Software Technology*, 91:186–197, 2017.

[42] Chenxi Zhang, Xin Peng, Chaofeng Sha, Ke Zhang, Zhenqing Fu, Xiya Wu, Qingwei Lin, and Dongmei Zhang. Deeptralog: Trace-log combined microservice anomaly detection through graph-based deep learning. *Conference on Software Engineering (ICSE)*, 2022.

[43] Lukas Ruff, Robert Vandermeulen, Nico Goernitz, Lucas Deecke, Shoaib Ahmed Siddiqui, Alexander Binder, Emmanuel Müller, and Marius Kloft. Deep one-class classification. In *International conference on machine learning*, pages 4393–4402. PMLR, 2018.

[44] Jasmin Bogatinovski and Sasho Nedelkoski. Multi-source anomaly detection in distributed it systems. In *International Conference on Service-Oriented Computing*, pages 201–213. Springer, 2020.

[45] Dewei Liu, Chuan He, Xin Peng, Fan Lin, Chenxi Zhang, Shengfang Gong, Ziang Li, Jiayu Ou, and Zheshun Wu. Microhecl: High-efficient root cause localization in large-scale microservice systems. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 338–347. IEEE, 2021.

[46] Min Li, Dingyong Tang, Zepeng Wen, and Yunchang Cheng. Microservice anomaly detection based on tracing data using semi-supervised learning. In *2021 4th International Conference on Artificial Intelligence and Big Data (ICAIBD)*, pages 38–44. IEEE, 2021.

[47] Li Wu, Johan Tordsson, Erik Elmroth, and Odej Kao. Microrca: Root cause localization of performance issues in microservices. In *NOMS 2020-2020 IEEE/IFIP Network Operations and Management Symposium*, pages 1–9. IEEE, 2020.

[48] Meng Ma, Jingmin Xu, Yuan Wang, Pengfei Chen, Zonghua Zhang, and Ping Wang. Automap: Diagnose your microservice-based web applications automatically. In *Proceedings of The Web Conference 2020*, pages 246–258, 2020.

[49] Qixun Zhang, Tong Jia, Zhonghai Wu, Qingxin Wu, Lichun Jia, Donglei Li, Yuqing Tao, and Yutong Xiao. Fault localization for microservice applications with system logs and monitoring metrics. In *2022 7th International Conference on Cloud Computing and Big Data Analytics (ICCCBDA)*, pages 149–154. IEEE, 2022.

[50] Tao Wang, Wenbo Zhang, Chunyang Ye, Jun Wei, Hua Zhong, and Tao Huang. Fd4c: Automatic fault diagnosis framework for web applications in cloud computing. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 46(1):61–75, 2015.

[51] Chenyu Zhao, Minghua Ma, Zhenyu Zhong, Shenglin Zhang, Zhiyuan Tan, Xiao Xiong, LuLu Yu, Jiayi Feng, Yongqian Sun, Yuzhi Zhang, Dan Pei, Qingwei Lin, and Dongmei Zhang. Robust multimodal failure detection for microservice systems. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, KDD '23, page 5639–5649. Association for Computing Machinery, 2023.

[52] Cheryl Lee, Tianyi Yang, Zhuangbin Chen, Yuxin Su, and Michael R. Lyu. Eadro: An end-to-end troubleshooting framework for microservices on multi-source data. In *Proceedings of the 45th International Conference on Software Engineering*, ICSE '23, page 1750–1762. IEEE Press, 2023.

[53] Ivan Beschastnikh, Perry Liu, Albert Xing, Patty Wang, Yuriy Brun, and Michael D Ernst. Visualizing distributed system executions. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 29(2):1–38, 2020.

[54] Prometheus: From metrics to insight power your metrics and alerting with the leading open-source monitoring solution. Available athttps://prometheus.io/. Accessed: 2022-02-04.

[55] Jaeger: open source, end-to-end distributed tracing. Available athttps://www.jaegertracing.io/. Accessed: 2022-02-04.

[56] istio-simplify observability, traffic management, security, and policy with the leading service mesh. Available athttps://istio.io/ (2022-02-04).

[57] Kubernetes. Available athttps://kubernetes.io/ (2022-08-09).

[58] Philippe Flajolet, Paolo Sipala, and Jean-Marc Steyaert. Analytic variations on the common subexpression problem. In *International Colloquium on Automata, Languages, and Programming*, pages 220–234. Springer, 1990.

[59] Gabriel Alejandro Valiente Feruglio. *Simple and efficient tree pattern matching*. PhD thesis, Universitat Politècnica de Catalunya, 2000.

[60] Ines Ben Messaoud, Jamel Feki, and Gilles Zurfluh. A first step for building a document warehouse: Unification of xml documents. In *2012 Sixth International Conference on Research Challenges in Information Science (RCIS)*, pages 1–6. IEEE, 2012.

[61] Richard B Darlington and Andrew F Hayes. *Regression Analysis and Linear Models: Concepts, Applications, and Implementation*. New York, NY: Guilford, 2017.

[62] Sungyoung Lee, Young-Tack Park, Brian J d'Auriol, et al. A novel feature selection method based on normalized mutual information. *Applied Intelligence*, 37(1):100–120, 2012.

[63] David A Freedman. *Statistical models: theory and practice*. cambridge university press, 2009.

[64] Miao Jiang, Mohammad A Munawar, Thomas Reidemeister, and Paul AS Ward. Efficient fault detection and diagnosis in complex software systems with information-theoretic monitoring. *IEEE Transactions on Dependable and Secure Computing*, 8(4):510–522, 2011.

[65] Neeraj Mohan, Ruchi Singla, Priyanka Kaushal, and Seifedine Kadry. *Artificial Intelligence, Machine Learning, and Data Science Technologies: Future Impact and Well-being for Society 5.0*. CRC Press, 2021.

[66] Brad Boehmke and Brandon Greenwell. *Hands-on machine learning with R*. Chapman and Hall/CRC, 2019.

[67] Arnaud De Myttenaere, Boris Golden, Bénédicte Le Grand, and Fabrice Rossi. Mean absolute percentage error for regression models. *Neurocomputing*, 192:38–48, 2016.

[68] Eamonn Keogh, Selina Chu, David Hart, and Michael Pazzani. An online algorithm for segmenting time series. In *Proceedings 2001 IEEE international conference on data mining*, pages 289–296. IEEE, 2001.

[69] Lin Song, Peter Langfelder, and Steve Horvath. Comparison of co-expression measures: mutual information, correlation, and model based indices. *BMC bioinformatics*, 13(1):1–21, 2012.

[70] Vladimir Vapnik. The support vector method of function estimation. In *Nonlinear modeling*, pages 55–85. Springer, 1998.

[71] Vladimir Vapnik, Steven Golowich, and Alex Smola. Support vector method for function approximation, regression estimation and signal processing. *Advances in neural information processing systems*, 9, 1996.

[72] Luigi P Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. Subgraph transformations for the inexact matching of attributed relational graphs. In *Graph based representations in pattern recognition*, pages 43–52. Springer, 1998.

[73] Leon Danon, Albert Diaz-Guilera, Jordi Duch, and Alex Arenas. Comparing community structure identification. *Journal of statistical mechanics: Theory and experiment*, 2005(09):P09008, 2005.

[74] scikit learn. scikit-learn: Machine learning in python. https://scikit-learn.org/stable/index.html, 2023. Accessed: August , 2023.

[75] Yu Hui, Sun Wenzhu, Zhou Xiuzhi, Zhu Guotao, and Hu Wenting. Heuristic sample reduction based support vector regression method. In *2016 IEEE International Conference on Mechatronics and Automation*, pages 2065–2069. IEEE, 2016.

[76] Luigi P Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. A (sub) graph isomorphism algorithm for matching large graphs. *IEEE transactions on pattern analysis and machine intelligence*, 26(10):1367–1372, 2004.

[77] Eugene M Luks. Isomorphism of graphs of bounded valence can be tested in polynomial time. *Journal of computer and system sciences*, 25(1):42–65, 1982.

[78] Apache jmeter. Available athttps://jmeter.apache.org/ (2022-08-04).

[79] Álvaro Brandón, Marc Solé, Alberto Huélamo, David Solans, María S Pérez, and Victor Muntés-Mulero. Graph-based root cause analysis for service-oriented and microservice architectures. *Journal of Systems and Software*, 159:110432, 2020.

[80] Jacob Yerushalmy. Statistical problems in assessing methods of medical diagnosis, with special reference to x-ray techniques. *Public Health Reports (1896-1970)*, pages 1432–1449, 1947.

[81] H. Gzyl, S. Mayoral, and E. Gomes-Gonçalves. *Loss Data Analysis: The Maximum Entropy Approach*. De Gruyter STEM. De Gruyter, 2023.

[82] Reginald Smith. A mutual information approach to calculating nonlinearity. *Stat*, 4(1):291–303, 2015.

[83] Jan Beirlant, Edward J Dudewicz, László Györfi, Edward C Van der Meulen, et al. Nonparametric entropy estimation: An overview. *International Journal of Mathematical and Statistical Sciences*, 6(1):17–39, 1997.

[84] Heidar Pirzadeh, Sara Shanian, Abdelwahab Hamou-Lhadj, Luay Alawneh, and Arya Sharifee. Stratified sampling of execution traces: Execution phases serving as strata. *Science of Computer Programming*, 78(8):1099–1118, 2013.