# Anomaly Detection and Failure Root Cause Analysis in (Micro) Service-Based Cloud Applications: A Survey

JACOPO SOLDANI and ANTONIO BROGI, University of Pisa

The proliferation of services and service interactions within microservices and cloud-native applications, makes it harder to detect failures and to identify their possible root causes, which is, on the other hand crucial to promptly recover and fix applications. Various techniques have been proposed to promptly detect failures based on their symptoms, viz., observing anomalous behaviour in one or more application services, as well as to analyse logs or monitored performance of such services to determine the possible root causes for observed anomalies. The objective of this survey is to provide a structured overview and qualitative analysis of currently available techniques for anomaly detection and root cause analysis in modern multi-service applications. Some open challenges and research directions stemming out from the analysis are also discussed.

CCS Concepts: • **General and reference** → *Surveys and overviews*; • **Applied computing** → *Service-oriented architectures*; • **Computer systems organization** → *Cloud computing;*

Additional Key Words and Phrases: Microservices, multi-service applications, failure detection, anomaly detection, root cause analysis

**ACM Reference format:**
Jacopo Soldani and Antonio Brogi. 2022. Anomaly Detection and Failure Root Cause Analysis in (Micro) Service-Based Cloud Applications: A Survey. *ACM Comput. Surv.* 55, 3, Article 59 (February 2022), 39 pages.
https://doi.org/10.1145/3501297

## 1 INTRODUCTION

With the rise of microservices, multi-service applications became the de-facto standard for delivering enterprise IT applications [80]. Many big players are already delivering their core business through multi-service applications, with Amazon, Netflix, Spotify, and Twitter perhaps being the most prominent examples. Multi-service applications, like microservice-based applications, hold the promise of exploiting the potentials of cloud computing to obtain cloud-native applications composed of loosely coupled services that can be independently deployed and scaled [40].

At the same time, services and service interactions proliferate in modern multi-service applications, often consisting of hundreds of interacting services. This makes it harder to monitor the various services forming an application to detect whether they have failed, as well as to understand whether a service failed on its own or in cascade, viz., because some of the services it interacts with failed, causing the service to fail as well. The detection and understanding of failures in modern multi-service applications are actually considered a concrete "pain" by their operators [80].

Authors' address: J. Soldani and A. Brogi, University of Pisa, Dipartimento di Informatica, Largo B. Pontecorvo 3, Pisa, PI 56127, Italy; emails: {jacopo.soldani, antonio.brogi}@unipi.it.

0360-0300/2022/02-ART59 $15.00
https://doi.org/10.1145/3501297

Various techniques have been proposed to automate the detection of failures and automatically determine their possible root causes. Failure detection typically relies on identifying anomalies in the behaviour of services, which can be symptoms of their possible failures [58, 60, 91]. Once an anomaly has been detected in a multi-service application, further analyses are enacted to determine the possible root causes for such an anomaly [42, 76]. This allows application operators to determine whether the anomaly observed on service was due to the service itself, to other services underperforming or failing as well, or to environmental reasons, e.g., unforeseen peaks in user requests [63] or lack of computing resources in the runtime environment [19].

Existing techniques for anomaly detection and root cause analysis are however scattered across different pieces of literature, and often focus only on either anomaly detection or root cause analysis. This makes it complex equipping multi-service applications with a pipeline for detecting anomalies and identifying their root causes. To this end, in this article, we survey the existing techniques for detecting anomalies in multi-service applications and for identifying their possible root causes. We also discuss the instrumentation needed to apply an anomaly detection/root cause analysis technique, as well as the additional artifacts that must be provided as input to such techniques. We also highlight whether the surveyed techniques already put anomaly detection and root cause analysis in a pipeline or whether they need to be integrated into one another. In the latter case, we comment on whether the type of anomalies that can be observed with a given anomaly detection technique can be explained by a given root cause analysis technique.

We believe that our survey can provide benefits to both practitioners and researchers working with modern multi-service applications. We indeed not only help them in finding the anomaly detection and root cause analysis techniques most suited to their needs, but we also discuss some open challenges and possible research directions on the topic.

The article is organised as follows. Sections 2 and 3 introduce some terminology and discuss related work, respectively. Sections 4 and 5 survey existing techniques for anomaly detection and root cause analysis in multi-service applications, which are then jointly discussed in Section 6. Section 7 then concludes the article by outlining possible research directions stemming out from our survey.

## 2 TERMINOLOGY

Whilst *failures* denote the actual inability of a service to perform its functions [73], *anomalies* correspond to the observable symptoms for such failures [87], e.g., a service slowing its response time, reducing its throughput, or logging error events. The problem of *anomaly detection* in multi-service applications hence consists of identifying anomalies that can possibly correspond to failures affecting their services. Anomalies can be detected either at *application-level* or at *service-level*, based on whether symptoms of a failure are observed by considering the application as a whole (e.g., performance degradations or errors shown by its frontend) or by focusing on specific services.

*Root cause analysis* then consists of identifying the reasons why an application- or service-level anomaly has been observed, with the ultimate goal of providing possible reasons for the corresponding failure to occur. Root cause analysis is enacted by analysing what has actually happened while an application is running. It is hence not to be confused with debugging techniques, which look for the possible reasons for an observed anomaly by re-running applications in a testing environment and by trying to replicate the observed anomaly (e.g., as proposed by Zhou et al. [96]). Root cause analysis techniques instead only analyse information collected while the application was running (e.g., logs or monitoring data) without re-running it under different conditions.

Anomaly detection and root cause analysis are typically enacted based on runtime information collected on the services forming an application, hereafter, also called *application services*. Such information includes **Key Performance Indicators *KPIs*** monitored on application services (e.g., response time or resource consumption) and the *events* they logged. Each logged event

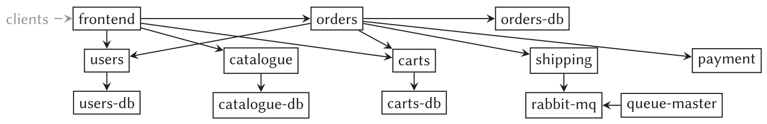

Fig. 1. Example of multi-service application topology. Boxes denote application services, whilst arrows denote interactions among application services (in black) and from external clients to application services (in gray).

provide information on something that happened to the service logging such event, e.g., whether it was experiencing some error, hence logging an *error event*, or whether it was invoking or being invoked by another service. The events logged by a service constitute the *service logs*, whereas the union of all *service logs* distributed among all application services is hereafter referred to as *application logs*.

Application logs are not to be confused with *distributed traces*, obtained by instrumenting application services to feature *distributed tracing*, a distributed logging method for multi-service applications. It consists of instrumenting application services to (i) assign each user request a unique id, (ii) pass such id to all services that are involved in processing the request, (iii) include the request id in all log messages, and (iv) record information (e.g., start time, end time) about the service invocations and internal operations performed when handling a request in service [71]. All the events generated by user requests are hence tracked in the distributed "traces", and traces of events generated by the same request can be identified based on the request id.

*Example.* Consider *Sock Shop* [89], an open-source web-based application whose topology is in Figure 1. The logs, traces, or KPIs of *Sock Shop*'s services can be used to enact anomaly detection, e.g., to detect services experiencing anomalous performances (viz., *performance anomalies*) or enacting some unexpected action (viz., *functional anomalies*). For instance, suppose that frontend is taking a lot of time to answer to a request. This may be due to external reasons (e.g., peaks of requests) or to some failure occurred in frontend, causing its anomalous behaviour. To determine whether the performance degradation is truly denoting an anomaly, the logs, traces, or KPIs produced by frontend can be processed to enact anomaly detection. This can be done *online*, if logs, traces, or KPIs are processed when they are produced, or *offline*, if they are stored and processed later on.

The above is an example of application-level anomaly, as the anomaly occurs on the frontend of *Sock Shop*. Different is the case of anomalies occurring on internal services, e.g., failures occurring on carts-db and queue-master, in which case we talk about service-level anomalies. The failure of carts-db may propagate up to frontend: carts-db is invoked by carts, which may fail in cascade when carts-db fails, and this may cause a cascading failure in frontend as well. If this happens, the service-level anomaly of carts-db would cause an application-level anomaly observed on frontend, without however distinguishing this case from that discussed above, where the anomaly was only due to frontend itself. Instead, an anomaly on queue-master cannot propagate up to frontend, as there is no invocation chain from the latter to queue-master. This means that one such anomaly would not be observed if we would enact application-level anomaly detection, as we would need a more fine-grained solution to detect service-level anomalies. One such solution would actually allow us to determine all the above discussed anomalies, by explicitly stating that each detected anomaly affects frontend, carts-db, or queue-master (rather than the whole application).

Consider now two possible anomalies for frontend, one due to an internal failure of frontend and the other due to a failure in carts-db propagating to frontend. The logs, traces, or KPIs produced by *Sock Shop*'s application services can be processed offline to determine the possible root cause for an anomaly observed on frontend, including the two mentioned above. In one case, the root cause would be something happened to the frontend, whilst in the other case, the root cause will be identified as something happened to carts-db. Depending on whether the enacted analysis can determine the possible root causes of anomalies observed on an application's frontend or on any of its services, we talk about root cause analysis of application-level anomalies or service-level anomalies, respectively. As we will see, also the granularity of identified root causes varies, spanning from just returning the service names to returning the events or KPIs produced by services that most probably caused the observed anomaly. In addition, as the possible root causes can be many, these are often ranked from the highest probable to the lowest probable, based on different heuristics.

One may hence wonder which is the "best" available technique to enact anomaly detection and root cause analysis. Unfortunately, there are no techniques fitting all possible scenarios. Whether to employ one technique or another indeed depends on various aspects, e.g., setup costs, or type and granularity of the anomalies and root causes to be identified. To ease choosing the anomaly detection and root cause analysis techniques best suited to an application's needs, we hereafter provide a first classification based on the above aspects, among others.

## 3 RELATED WORK

Anomaly detection and root cause analysis techniques have already been discussed in existing surveys; however, they differ from ours and focus either on anomaly detection or root cause analysis, often not referring to their use in multi-service applications. For instance, Chandola et al. [16] and Akoglu et al. [2] survey the research on detecting anomalies in data. In both surveys, the focus is on the generic problem of significant changes/outlier detection in data, independently of whether this is done online or offline, or what the data is about. We instead focus on online anomaly detection in a specific type of data, viz., logs, traces, or KPIs monitored on application services. This allows us to provide a more detailed description of how such techniques can detect anomalies in multi-service applications, and to compare and discuss them under dimensions peculiar to such problem, e.g., detection of application- vs. service-level anomalies, setup costs, or accuracy in dynamically changing applications. In addition, the studies surveyed by Chandola et al. and Akoglu et al. are dated 2015 at most, hence not covering various anomaly detection techniques for multi-service applications. For instance, distributed tracing-based anomaly detection techniques are not covered by Chandola et al. nor by Akoglu et al., since they have indeed been proposed after 2014, when microservices were first proposed [41] and gave rise to the use of distributed tracing in multi-service applications [80]. Finally, differently from Chandola et al. and Akoglu et al., we also survey the techniques that can be used in a pipeline with the surveyed anomaly detection techniques to determine the possible root causes for detected anomalies.

Steinder and Sethi [83], Wong et al. [90], and Sole et al. [81] instead survey solutions for determining the possible root causes for failures in software systems. Steinder and Sethi and Wong et al. however, focus on software systems different from multi-service applications, namely computer networks and monolithic applications, respectively. The survey by Sole et al. is instead closer to ours, since they focus on techniques allowing to determine the root causes of anomalies observed in multi-component software systems, therein included multi-service applications. Their survey is, however, complementary to ours: they analyse the performance and scalability of the surveyed root cause analysis techniques, whilst we focus on more qualitative aspects, e.g., which types of anomalies can be analysed, what instrumentation is required, or which actual root causes are determined. In addition, we also survey existing techniques for detecting anomalies in multi-service application, with the aim at supporting application operators in setting up their pipeline to detect anomalies in their applications and to determine the root causes of observed anomalies.

Another noteworthy study is that by Arya et al. [5], who evaluate several state-of-the-art techniques for root cause analysis, based on logs obtained from a publicly available benchmark

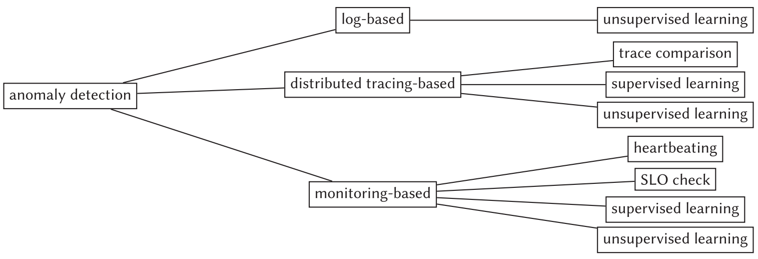

Fig. 2. A taxonomy for classifying techniques for detecting anomalies in multi-service applications.

multi-service applications. In particular, they consider Granger causality-based techniques, by providing a first quantitative evaluation of their performance and accuracy on a common dataset of application logs. They hence complement the qualitative comparison in our survey with a first result along a research direction emerging from our survey (viz., quantitatively comparing the performance and accuracy of the existing root cause analysis techniques), which we later discuss in Section 7.

In summary, to the best of our knowledge, ours is the first survey on the techniques for detecting anomalies that can be symptoms of failures in multi-service applications, while at the same time presenting the analysis techniques for determining possible root causes for such anomalies. It is also the first survey providing a qualitative analysis of such techniques in the perspective of their use with multi-service applications, e.g., whether they can detect and analyse anomalies at application- or service-level, which instrumentation they require, and how they behave when the services forming an application and their runtime environment change over time.

## 4 ANOMALY DETECTION

We hereafter survey the existing techniques for detecting anomalies in multi-service applications, viz., symptoms of failures affecting the whole application or some of its services. We organise the survey by distinguishing three main classes of techniques, following the taxonomy shown in Figure 2. We separately discuss the techniques enacting anomaly detection by directly processing the logs produced by application services (Section 4.1), from those processing the traces they produce (Section 4.2) or the KPIs monitored on such services (Section 4.3). Such a categorisation provides a first distinction on the applicability of the surveyed techniques, based on the instrumentation that a multi-service application should feature for such techniques to detect anomalies therein. Intuitively, log-based application techniques can be applied to the logs natively produced by the services forming an application. Distributed tracing-based and monitoring-based techniques instead require two different types of instrumentation to be applied to application services. The former requires application services to be instrumented to feature distributed tracing, whereas the latter requires monitoring agents to be installed alongside application services to monitor their KPIs.

### 4.1 Log-Based Anomaly Detection Techniques

Log-based online anomaly detection is typically based on unsupervised machine learning algorithms, which are used to process the logs produced by application services to detect whether some anomaly is occurring in some service. In particular, unsupervised learning algorithms are used to learn a baseline modelling of the logging behaviour of the application services in failure-free runs of an application. The baseline model is then used online to detect whether the events newly logged by a service diverge from the baseline model, hence indicating that the service suffers from some anomaly. This technique is adopted by OASIS [58] to detect application-level functional anomalies, and by Jia et al. [33] and LogSed [34] to detect functional and performance anomalies at the service-level.

OASIS [58] starts from the application logs generated in failure-free runs of the application, which are used as the training data to mine a control-flow graph modelling the baseline application behaviour under normal conditions, viz., which events should be logged by which service, and in which order. OASIS mines log templates from the textual content in the training logs, and it clusters the log templates so that all templates originated by the same log statement go in the same cluster. The nodes in the control flow graph are then determined by picking a representative template from each cluster. The edges are instead mined based on temporal co-occurrence of log templates, upon which OASIS determines the immediate successors of each template, together with branching probabilities and expected time lags between each template and its successors. Online anomaly detection is then enacted by mapping newly generated logs to their corresponding templates and by checking whether the arriving log templates adhere to the control-flow graph. If none of the templates that should follow a template is logged in the corresponding time lag, or if the actual rate of templates following a given template significantly deviates from the expected branching probabilities, OASIS considers the application as affected by a functional anomaly.

Jia et al. [33] and LogSed [34] provide two other techniques for the detection of functional or performance anomalies in multi-service applications based on the analysis of their logs. They process the logs collected in normal, failure-free runs of an application to mine, for each service, a time-weighted control flow graph modelling its internal flow. In particular, Jia et al. and LogSed learn the time-weighted control flow graph for each service, with a similar method to that in OASIS [58]: template mining is used to transform logs into templates, which are then clustered based on events they correspond to, and the log clusters are processed to infer the sequencing relationships among events. The resulting graph of events is enriched by weighting each arc with the average time passing between logging its source and target events. The topology graph and the time-weighted control flow graphs constitute the baseline model against which to compare newly logged events. This is done by mapping the events logged by each service to their corresponding template and by checking whether its actual behaviour deviates from the expected one. A service suffers from functional anomalies whenever it does not log an event that it should have been logging, or when unexpected logs occur. A service instead suffers from performance anomalies when it logs an expected event significantly later than when expected.

OASIS [58], Jia et al. [33], and LogSed [34] share the common assumption to enact anomaly detection based on supervised learning: they assume that applications log similar events in a similar order, both during the training runs and while the application is actually running in production. This is, however, not always the case since the actual behaviour of multi-service applications also depends on the conditions and load under which they are executed, which may vary, e.g., with seasonalities [52]. Modern multi-service applications are also highly dynamic in their nature, with new functionalities included by introducing new services, and outdated services removed or replaced by other services [80]. As a result, assuming that multi-service applications keep running under the same conditions as in training runs may decrease the accuracy of the enacted anomaly detection, if the runtime conditions of an application change. Such changes may indeed result in detecting false positives/negatives, depending on whether they consider the logs produced by some application services as anomalous/normal, even if these rather correspond to the new normal/anomalous logging behaviour of the application in its new runtime conditions.

## 4.2 Distributed Tracing-Based Anomaly Detection Techniques

Online anomaly detection is also enacted by instrumenting the target applications to feature additional functionalities, with the most common being distributed tracing. Distributed tracing is indeed at the basis of various techniques for anomaly detection, combined with supervised or unsupervised machine learning,[1] or with trace comparison techniques. In all such cases, the common assumption is that the traces produced by an application in training runs are consistent with those it produces at runtime, hence providing a baseline against which to compare newly produced traces. This is almost the same assumption as for the log-based anomaly detection techniques just discussed, hence bringing the same limitations: if the runtime conditions for an application vary from those in training runs, the accuracy of enacted anomaly detection may decrease.

*4.2.1 Unsupervised Learning.* TraceAnomaly [45] and Nedelkoski et al. [61] collect traces in training runs of a multi-service application and use them to train unsupervised neural networks. They then enact online anomaly detection by exploiting the trained neural networks to process the traces generated by the application while it is running in production. In particular, TraceAnomaly trains a deep Bayesian neural network with posterior flow [59, 70], which enables associating monitored traces with a likelihood to be normal, viz., not affected by performance anomalies. It also stores all seen service call paths, viz., all sequences of service interactions observed in the available traces. TraceAnomaly then enacts online anomaly detection in two steps. It first checks whether a newly produced trace contains previously unseen call paths. If this is the case, TraceAnomaly decides whether to consider the unseen call paths as functional anomalies for the application, based on whether they appear in a whitelist manually filled by the application operator to exclude false positives (e.g., unseen call paths corresponding to new interactions occurring after service upgrades). If there is no functional anomaly, the trace is passed to the deep Bayesian neural network, which associates the monitored trace with its likelihood to be normal. If the likelihood is below a given threshold, the trace denotes a performance anomaly for the application.

Nedelkoski et al. [61] train a multi-modal long-short term memory neural network [23] modelling the normal behaviour of a multi-service application, based on the traces collected in failure-free runs of the application. The long-short term memory neural network is multi-modal in the sense that it considers both the types of events logged in a trace and the response times of application services. It is obtained by combining two single-modal networks, one trained to predict the probability distribution of the events following an event in a trace, and the other trained to predict the probability distribution of the response times for a service invocation. The trained network is then used in online anomaly detection to predict the most probable events following an event and the most probable response times for a service invocation. If the event following another, or if the response time of a service are not amongst the predicted ones, the application is considered to suffer from a functional or performance anomaly, respectively.

Differently from the above-listed techniques and from most of the techniques surveyed in this section, Jin et al. provide an *offline* anomaly detection solution. Jin et al. [35] indeed allow detecting performance anomalies in multi-service applications based on the offline analysis of the traces collected through distributed tracing, by also considering the performance metrics of application services that can be directly collected from their runtime environment (e.g., CPU and memory consumption). Jin et al. first enact principal component analysis on logged traces to determine the services that may be involved in anomalous interactions. This is done by deriving a matrix-based representation of the logged traces, by reducing the obtained matrix to its principal components, and by applying a function linearly combining such principal components to elicit the services involved in anomalous traces. The performance metrics collected for such services are then processed with unsupervised learning algorithms for anomaly detection, viz., isolation

[1]Whilst unsupervised learning algorithms learn patterns directly from data, supervised learning algorithms rely on data to be labelled, viz., training examples associating given input data with desired output values [57].

forest, one-class support vector machine, local outlier factor, or $3\sigma$ [16]. Such algorithms are used to determine anomalous values for such metrics, which are used to assign an anomaly score to the services involved in anomalous traces. Jin et al. then return the list of services affected by performance anomalies, viz., the services whose anomaly score is higher than a given threshold, ranked by anomaly score.

*4.2.2 Supervised Learning.* Seer [22], Nedelkoski et al. [60], and Bogatinovski et al. [8] also enact a combination of distributed tracing and deep learning, however, following a supervised learning approach. They indeed collect traces in training runs of a multi-service application, labelled to distinguish normal runs from those where anomalies are known to have occurred on specific services. The collected traces are used to train neural networks, which are then used to enact online anomaly detection by processing the traces newly generated by the application. In particular, Seer [22] trains a deep neural network consisting of convolutional layers followed by long-short term memory layers [23]. Input and output neurons of the network correspond to application services. Input neurons are used to pass the KPIs of each service, viz., latency and outstanding requests, logged by the service itself in distributed traces, and resource consumption, obtained by interacting with the nodes where the services are running. Output neurons instead indicate which services are affected by performance anomalies. At runtime, Seer continuously feeds the trained deep neural network with the stream of monitored traces, and the network consumes such traces to determine whether some application service is affected by a performance anomaly. If this is the case, Seer interacts with the runtime of the node hosting the anomalous service to determine which computing resources are being saturated. Seer then notifies the system manager about the resource saturation happening on the node, to allow the manager to mitigate the effects of the detected performance degradation, e.g., by providing the node with additional computing resources.

Nedelkoski et al. [60] train two neural networks, viz., a variational autoencoder [39] modelling the normal behaviour of the application and a convolutional neural network [23] recognising the type of failure affecting a service when the latter is affected by a performance anomaly. The variational autoencoder is learned from the distributed traces obtained in normal runs of the application, so that the autoencoder learns to encode non-anomalous traces and to suitably reconstruct them from their encoding. If applied on an anomalous trace, instead, the autoencoder reconstruct them with significant errors, hence enabling to detect anomalies. The convolutional neural network is instead trained on the traces collected in runs of the application where specific failures were injected in specific services, so that the network can recognise which failures caused a performance anomaly in a service. In online anomaly detection, the trained networks are used in a pipeline. The outputs of the autoencoder are passed to a post-processor to exclude false positives, viz., performance anomalies affecting a service due to temporary reasons (e.g., overload of a service). If a service is considered anomalous by the post-processor as well, the anomalous trace is passed to the convolutional network to detect the type of anomaly affecting the service.

Bogatinovski et al. [8] applies a different reasoning from that of the above-discussed techniques, with the additional assumption that the logging of an event in a particular position in a trace is conditioned both by those logged earlier and by those appearing afterwards. They indeed train a self-supervised encoder-decoder neural network [23] capable of predicting the logging of an event in a given "masked" position of a trace-based on the context given by its neighbour events. The trained neural network provides a probability distribution of the possible events appearing in any masked position of an input trace, given the context of the events appearing before and after the masked event to be predicted. The trained neural network is then used to enact anomaly detection over novel traces, viz., to obtain a sorted list of events that should have been logged in each position of the trace. The obtained lists are then analysed by a post-processor, which considers a truly logged event as anomalous if it is not amongst the events predicted to appear in the corresponding position. The post-processor then computes an anomaly score for the trace (viz., the number of anomalous events divided by trace length). If the anomaly score is beyond an user-specified threshold, the trace is considered to witness a functional anomaly affecting the application.

Finally, MEPFL [97] provides another technique applying supervised machine learning to perform online anomaly detection on application instrumented to feature distributed tracing. Whilst all the above techniques directly process training and runtime traces as they are, MEPFL requires a slightly different setup: it takes as input a multi-service application and a set of automated test cases simulating user requests to load the application and to produce traces, and it requires traces to include additional information if compared to that typically included (viz., configuration of a service and status, resource consumption, and service interactions of its instances). This information is processed by a supervised learning algorithm (viz., k-nearest neighbors [3], random forests [10], or multi-layer perceptron [23]) to train classifiers determining whether a trace includes anomalies, which services are affected by such anomalies, and which types of failures most probably correspond to such anomalies. The training of the classifiers is based on traces collected in training runs of an application, still under the common assumption that, at runtime, applications behave consistently with training runs. During such training runs, the application is loaded with the input test cases, and it is run both in normal conditions and by injecting failures in its services to observe how the traces change when such type of failures affect such services.

*4.2.3 Trace Comparison.* Trace comparison is another possible technique to enact online anomaly detection on multi-service applications instrumented to feature distributed tracing. This is the technique followed by Meng et al. [54], Wang et al. [88], and Chen et al. [17], which all rely on collecting the traces that can possibly occur in a multi-service application and by then checking whether newly collected traces are similar to the collected ones. This is again done under the assumption that the application continues to behave consistently with previous runs, with the previously collected traces providing a baseline against which to compare newly collected ones. It hence also holds here that when the runtime conditions for an application vary from those when the baseline traces were collected, the accuracy of enacted anomaly detection may decrease.

Meng et al. [54] and Wang et al. [88] collect traces by testing an application in a pre-production environment, and use such traces to build a set of representative call trees. Such trees model the service invocation chains that can possibly appear in normal conditions, as they have been observed in the collected traces. Newly monitored traces are then reduced to their call trees, whose tree-edit distance from each representative call tree is computed with the RTDM algorithm [69]. If the minimum among all the computed distances is higher than a given threshold, the trace is considered as anomalous, and the service from which tree edits start is identified as the service suffering from a functional anomaly. Meng et al. and Wang et al. also enable detecting performance anomalies based on the services' response time. The response times in the traces collected in the testing phase are modelled as a matrix, whose elements represent the response time for service in each of the collected traces. Principal component analysis [37] is then enacted to reduce the matrix to its principal components, which are then used to define a function determining whether the response time of a service in newly monitored traces is anomalous. A limitation of one such approach is the computational complexity, which makes the anomaly detection enacted by Meng et al. and by Wand et al. better suited to enact offline anomaly detection, as they would be too time consuming to perform online anomaly detection on medium-/large-scale multi-service applications.

Such a limitation does not apply to the trace comparison-based anomaly detection enacted by Chen et al. [17], thanks to the fast matrix sketching algorithm they use to compare traces and detect anomalies. More precisely, Chen et al. adapt an existing matrix sketching algorithm [28] to detect response time anomalies in services in monitored traces. This is done by maintaining up-to-date a "sketch" of the high-dimensional data corresponding to the historically monitored response times of the services forming an application. The sketch is a limited set of orthogonal basis vectors that can linearly reconstruct the space containing the non-anomalous response times for each service in the formerly monitored traces. The response time for a service in a newly monitored trace is anomalous if it does not lie within or close enough to the space defined by the sketch vector, given a tolerance threshold. If this is the case, the corresponding service is considered to experience a performance anomaly. Otherwise, the sketch vectors are updated with the goal of reducing the rate of false positives/negatives by adapting them to the changing runtime conditions of an application.

## 4.3 Monitoring-Based Anomaly Detection Techniques

Online anomaly detection in multi-service applications can also be enacted by installing agents to monitor KPIs on their services and processing such KPIs to detect anomalies. The most common technique is to exploit unsupervised or supervised machine learning algorithms to process monitored KPIs, train a baseline model, and detect whether an anomaly is occurring in any application service by comparing the newly monitored KPIs against such baseline. Such an approach enables determining functional and performance anomalies at both application-level and service-level, still under the assumption that applications run in the same conditions during the training phase and at runtime. At the same time, as already observed in the previous sections, one such assumption results in the accuracy of the enacted anomaly detection potentially decreasing when the runtime conditions for an application vary, e.g., due to higher loads generated by peaks of end-users requests, or since new services are deployed to replace outdated ones.

The above limitation does not apply to other existing monitoring-based anomaly detection techniques, which rather use SLO checks and heart beating to detect anomalies, at the price of restricting the type or granularity of detected anomalies. SLO checks consist of comparing the KPIs monitored on the service acting as the application frontend against the application's SLOs: in case of SLO violations, the whole application is considered to suffer from a performance anomaly. Another possibility is to exploit self-adaptive heartbeat protocols to detect functionally anomalous services; however, requiring additional instrumentation than installing monitoring agents: application services are indeed required to comply with the employed heartbeat protocols.

*4.3.1 Unsupervised Learning.* Online anomaly detection can be enacted by exploiting unsupervised learning algorithms to process the KPIs monitored on application services in failure-free runs of the application and learn a baseline modelling of their behaviour. The baseline model is then used online to detect whether the newly monitored KPIs on a service diverge from the baseline model, hence indicating that the service suffers from some anomaly. This technique is adopted by Gulenko et al. [26], LOUD [51], MicroRCA [92], Wu et al. [91], and DLA [74], which train the baseline model by considering the KPIs monitored in training runs of the application, whilst assuming that no anomaly occurred in such runs. To do it, DLA only requires the target application deployment, whilst the others require a workload generator to be used during the training runs.

Gulenko et al. [26] and LOUD [51] are "KPI-agnostic", since they can work on any set of KPIs monitored with probes installed in the hosts where the application services are run. Gulenko et al. enable detecting performance anomalies in multi-service applications by building different baseline models for different services. The KPIs monitored on service at a given time are modelled as a vector, and the BIRCH [95] online clustering algorithm is used in an initial training phase to cluster the vectors corresponding to the monitored KPIs. At the end of the training phase, the centroids and centroid radii of the obtained clusters determine the baseline subspaces where the vectors of monitored KPIs should pertain to be classified as "normal". This enables the online processing of newly monitored KPIs for each service, by simply checking whether the corresponding vector falls within any of the baseline subspaces, viz., whether there exists a cluster whose centroid is less distant from the vector of monitored KPIs than the corresponding centroid radius. If this is not the case, the corresponding service is considered as affected by a performance anomaly.

LOUD [51] instead trains a baseline model for the whole application by pairing each monitored KPI with the corresponding service. The KPIs monitored in the offline runs of the application are passed to the IBM ITOA-PI [30] to build a KPI baseline model and a causality graph. The baseline model provides information about the average values and the acceptable variations over time for a KPI. Each node in the causality graph correspond to a service's KPI, and the edges indicate the causal relationships among KPIs, with weights indicating the probability to which changes in the source KPI can cause changes in the target KPI (computed with the Granger causality test [4]). During the online phase, LOUD again exploits the IBM ITOA-PI to detect performance anomalies: a KPI and the corresponding service are reported as anomalous if the monitored value for such KPI is outside of the acceptable variations coded in the baseline model, or if the causal relationships represented in the causality graph are not respected (e.g., when a KPI significantly changed its value, but those that should have changed as well remained unchanged).

MicroRCA [92], Wu et al. [91], and DLA [74] differ from the above-discussed techniques in their applicability: while Gulenko et al. [26] and LOUD [51] are agnostic with respect to the deployment technology and to the monitored KPIs, MicroRCA, Wu et al., and DLA are designed to work with Kubernetes (K8s) deployments and to focus on given KPIs. MicroRCA and Wu et al. consider the response time and resource consumption of application services, which they monitor by requiring the K8s deployment of multi-service applications to be instrumented as service meshes featuring Istio [31] and Prometheus [67]. The latter is used to collect KPIs from the application services. MicroRCA and Wu et al. then follow an unsupervised learning approach similar to that proposed by Gulenko et al. The KPIs monitored on each service are modelled as a vector, and the BIRCH algorithm [95] is used to cluster the vectors corresponding to KPIs monitored in normal runs of the application, whilst assuming that no anomaly occurred in such runs. The obtained clusters determine the baseline subspaces where the vectors of monitored KPIs should pertain to be classified as "normal". This enables the online processing of newly monitored KPIs for each service, by simply checking whether the corresponding vector falls within any of the baseline subspaces. If this is not the case, the service is considered as affected by a performance anomaly.

DLA [74] instead focuses on the response time of each application service, whilst also considering its possible fluctuations due to the increase/decrease of end users' transactions. It also focuses on containerised multi-service applications, whose deployment in K8s is expected to be provided by the application operator. DLA installs a monitoring agent in the VMs used to deploy the application. The agents collect the response time of the containerised services running in each VM, by associating each monitored response time with a timestamp and with the corresponding user transaction. This information is used to learn whether/how the response time varies based on the number of user transactions occurring simultaneously, based on the Spearman's rank correlation coefficient [77]. The obtained correlation is then used to check whether a variation in a newly monitored response time corresponds to an expected fluctuation due to an increase/decrease of users' transactions, or whether it actually denotes a performance anomaly affecting a service.

The above-discussed techniques train a baseline model of the application behaviour in an offline learning step, and then employ the trained model to detect anomalies while the application is running. A limitation of one such approach is that the accuracy of the enacted anomaly detection depends on the fact that the conditions under which an application is run do not change over time (e.g., no new application is deployed on the same VMs). The shared assumption is indeed that such conditions are the same during the training phase and at runtime. DLA [74] actually recognises this possible issue, also showing how the offline training can be periodically repeated to keep the baseline model updated to the new conditions under which a multi-service application is running.

CloudRanger [87] and Hora [65, 66] enact anomaly detection without relying on the above-stated assumption but still exploiting unsupervised learning. CloudRanger [87] enacts continuous learning on the frontend service of an application to learn which is its "expected" response time in the current conditions, based on the monitored response time, load, and throughput. This is done with the initial, offline training of a baseline model on historically monitored data for the application frontend. When the application is running, CloudRanger applies polynomial regression to monitored KPIs to update the baseline model to reflect the application behaviour in the dynamically changing runtime conditions. Online anomaly detection is then enacted by checking whether the response time of the application frontend significantly deviates from the expected one, based on a given tolerance threshold. At the same time, focusing on the application frontend, CloudRanger detects performance anomalies at a coarser granularity (application-level, rather than service-level), if compared with the above discussed unsupervised learning techniques [26, 51, 74, 91, 92].

Hora [65, 66] instead enacts service-level anomaly detection, by tackling the problem of online anomaly detection from a perspective different from all those discussed above. It indeed combines architectural models with statistical analysis techniques to preemptively determine the occurrence of performance anomalies in a multi-service application. Hora exploits SLAstic [85] to automatically extract a representation of the architectural entities in a multi-service application (viz., application services and nodes used to host them) and the degree to which a component depends on another. This information is used to create Bayesian networks [59] modelling the possible propagation of performance anomalies among the services forming an application. The online anomaly detection is then enacted by associating each application service with a detector, which monitors given KPIs for such service, and which analyses such monitored metrics to detect whether the currently monitored KPIs denote a performance anomaly. Since monitored metrics are time series data, detectors exploit autoregressive integrated moving average [78] to determine whether performance anomalies affect monitored services. Whenever a performance anomaly is detected on a service, Hora enacts Bayesian inference [59] to determine whether the anomaly propagates to other services. It then returns the set of application services most probably being affected by a performance anomaly.

*4.3.2 Supervised Learning.* ADS [21] and PreMiSE [52] enact online anomaly detection by exploiting supervised learning to train a baseline model on monitored KPIs. They label the KPIs monitored during training runs to denote whether they correspond to normal runs or to runs where specific failures were injected on specific services. The monitoring is enacted by installing monitoring agents in the VMs running application services and by exploiting a workload generator, which is provided by the application operator to simulate end-user requests during the training runs. ADS and PreMiSE, however, differ in setup costs: whilst PreMiSE is technology-agnostic and only requires the application deployment and workload generator, ADS requires the application deployment to be in K8s, and it also needs a module to inject failures in the services forming the target application.

ADS [21] enables detecting performance anomalies in a containerised multi-service application, given its K8s deployment, a workload generator simulating end-user requests, and modules for injecting failures in its services. ADS considers a predefined set of KPIs for each service, namely CPU, memory, and network consumption. In the training phase, the application is loaded with the workload generator. Multiple runs of the application are simulated, both in failure-free conditions and by injecting failures in the application services. The monitored KPIs are labelled as pertaining to "normal" or "anomalous" runs of the application, and they are processed to train a classifier with a supervised machine learning algorithm (viz., nearest neighbour [3], random forest [10], naïve Bayes [36], or support vector machines [75]), assuming that no more than one anomaly occurs at the same time. The trained classifier provides the baseline model used to process newly monitored KPIs to detect whether application services are experiencing some known performance anomaly.

PreMiSE [52] trains a baseline model for detecting anomalies in multi-service applications, assuming that each service runs in a different VM. Monitored KPIs are treated as time series: in an offline learning phase, PreMiSE trains a baseline modelling of the failure-free execution of the application services. The baseline model captures temporal relationships in the time series data corresponding to monitored KPIs to model trends and seasonalities, and it applies Granger causality tests [4] to determine whether the correlation among two KPIs is such that one can predict the evolution of the other. PreMiSE also trains a signature model representing the anomalous behaviour of the application in the presence of failures. This is done by injecting predefined failures in the VMs running the application services and by monitoring the corresponding changes in KPIs. The monitored KPIs are then used to train a classifier for detecting the occurrence of the anomalies corresponding to the injected failures, with one out of six supported supervised machine learning algorithms. The trained baseline model is then compared against newly monitored KPIs to enact online anomaly detection. Once an anomaly is detected, the signature model is used to classify the application service suffering from the detected anomaly and the failure it is suffering from.

*4.3.3 SLO Check.* CauseInfer [18, 19], Microscope [24, 42], and $\epsilon$-diagnosis [76] detect application-level performance anomalies by monitoring KPIs of the frontend of a multi-service application and by comparing such KPIs with the SLOs of the application. They, however, differ in their applicability, with CauseInfer designed to be technology-agnostic, whilst Microscope and $\epsilon$-diagnosis requiring the K8s deployment of the target application. CauseInfer exploits a monitoring agent to continuously monitor the response time of the service acting as frontend for an application. Microscope instead relies on the K8s deployment of an application to include monitoring agents (like Prometheus [67]), throughout which application services expose their response time. CauseInfer and Microscope then compare the KPIs monitored on the frontend of an application with that declared in the SLOs of the application. Whenever the performance of the frontend violates the SLOs, CauseInfer and Microscope consider the application as affected by a performance anomaly.

$\epsilon$-diagnosis [76] instead focuses on the tail latency of the application frontend, determined in small time windows (e.g., one minute or one second). The $\epsilon$-diagnosis system [76] indeed enacts online anomaly detection by partitioning the stream of monitored data into small-sized windows, it computes the tail latency for each window, and it then compares the tail latency with the threshold declared in the application SLOs. If the tail latency for a time window exceeds the given threshold, the whole application is considered to suffer from a performance anomaly in such a time window.

*4.3.4 Heartbeating.* M-MFSA-HDA [94] exploits heartbeating to detect service-level functional anomalies. This is done by installing a monitor periodically sending heartbeat messages to the application services, and by instrumenting such services to reply to heartbead messages within a given timeframe to be considered as fully working. If any application service does not reply to a heartbeat message within a given timeframe, it is considered to suffer from a functional anomaly. To keep a suitable tradeoff between the impact of the heartbeat protocol on application performances and the failure detection performances of the heartbeat detection system itself, M-MFSA-HDA self-adaptively changes the heartbeat rate by monitoring the CPU consumption of the nodes hosting application services, the network load, and the application workload.

Table 1. Classification of Anomaly Detection Techniques, Based on Their *Class* (L = Log-Based, DT = Distributed Tracing-Based, M = Monitoring-Based), The Applied *Method*, the *Type* (F = Functional, P = Performance) and *Granularity* of Detected Anomalies (A = Application-Level, S Service-Level), and The *input* They Need to Run

| | | | | Anomaly | | |
|---|---|---|---|---|---|---|
| **Reference** | **Class** | **Method** | **Online** | **Type** | **Gran.** | **Needed Input** |
| OASIS [58] | L | unsupervised learning | ✓ | F | A | previous and runtime logs |
| Jia et al. [33] | L | unsupervised learning | ✓ | F, P | S | previous and runtime logs |
| LogSed [34] | L | unsupervised learning | ✓ | F, P | S | previous and runtime logs |
| TraceAnomaly [45] | DT | unsupervised learning | ✓ | F, P | A | previous and runtime traces |
| Nedelkoski et al. [61] | DT | unsupervised learning | ✓ | F, P | A | previous and runtime traces |
| Jin et al. [35] | DT | unsupervised learning | | P | S | previous and runtime traces |
| Seer [22] | DT | supervised learning | ✓ | P | S | previous and runtime traces |
| Nedelkoski et al. [60] | DT | supervised learning | ✓ | P | S | previous and runtime traces |
| Bogatinovski et al. [8] | DT | supervised learning | ✓ | P | A | previous and runtime traces |
| MEPFL [97] | DT | supervised learning | ✓ | F | S | app deployment, test cases |
| Meng et al. [54] | DT | trace comparison | | F, P | S | previous and runtime traces |
| Wang et al. [88] | DT | trace comparison | | F, P | S | previous and runtime traces |
| Chen et al. [17] | DT | trace comparison | ✓ | P | S | previous and runtime traces |
| Gulenko et al. [26] | M | unsupervised learning | ✓ | P | S | app deployment, workload generator |
| LOUD [51] | M | unsupervised learning | ✓ | P | S | app deployment, workload generator |
| MicroRCA [92] | M | unsupervised learning | ✓ | P | S | K8s app deployment, workload generator |
| Wu et al. [91] | M | unsupervised learning | ✓ | P | S | K8s app deployment, workload generator |
| DLA [74] | M | unsupervised learning | ✓ | P | S | K8s app deployment |
| CloudRanger [87] | M | unsupervised learning | ✓ | P | A | app deployment, prev. monitored KPIs |
| Hora [65, 66] | M | unsupervised learning | ✓ | P | S | app deployment |
| ADS [21] | M | supervised learning | ✓ | P | S | K8s app deployment, workload generator, failure injectors |
| PreMiSE [52] | M | supervised learning | ✓ | F, P | S | app deployment, workload generator |
| CauseInfer [18, 19] | M | SLO check | ✓ | P | A | app deployment, SLOs |
| MicroScope [24, 42] | M | SLO check | ✓ | P | A | K8s app deployment, SLOs |
| $\epsilon$-diagnosis [76] | M | SLO check | ✓ | P | A | K8s app deployment, SLOs |
| M-MFSA- HDA [94] | M | heartbeating | ✓ | F | S | app deployment |

## 4.4 Summary

Table 1 recaps the surveyed anomaly detection techniques by distinguishing their classes (viz., log-based, distributed tracing-based, or monitoring-based) and the method they apply. The table also classifies the surveyed techniques based on whether they can detect functional or performance anomalies, and on the "granularity" of detected anomalies, viz., whether they just state that an application is suffering from an anomaly, or whether they distinguish which of its services are suffering from anomalies. Finally, the table provides insights on the artifacts that must be provided to the surveyed anomaly detection techniques, as input needed to actually enact anomaly detection.

Taking Table 1 as a reference, we summarise the surveyed techniques for detecting anomalies in multi-service applications. All such techniques (but for those based on SLO checks [18, 19, 24, 42, 76] or on heartbeating [94]) rely on processing data collected in training runs of the target applications. The idea is indeed to use the logs, traces, or KPIs collected in training runs of the application as a baseline against which to compare newly produced logs or traces, or newly monitored KPIs. Machine learning is the most used approach to train baseline models of the application behaviour: the logs, traces, or KPIs collected in training runs of the application are used to train baseline models with unsupervised learning algorithms [26, 33–35, 45, 51, 58, 61, 65, 66, 74, 87, 91, 92], or with supervised learning algorithms if logs, traces, or KPIs are also labelled with the failures that affected application services in the training runs [8, 21, 22, 52, 60, 97]. The trained baseline models span are clusters defining the space where newly monitored KPIs should pertain not to be considered anomalous, or classifiers/neural networks to be fed with newly monitored KPIs, logs, or traces to detect whether they denote anomalies.

An alternative approach to machine learning is trace comparison [17, 54, 88], provided that applications are instrumented to feature distributed tracing. The idea here is to store the traces

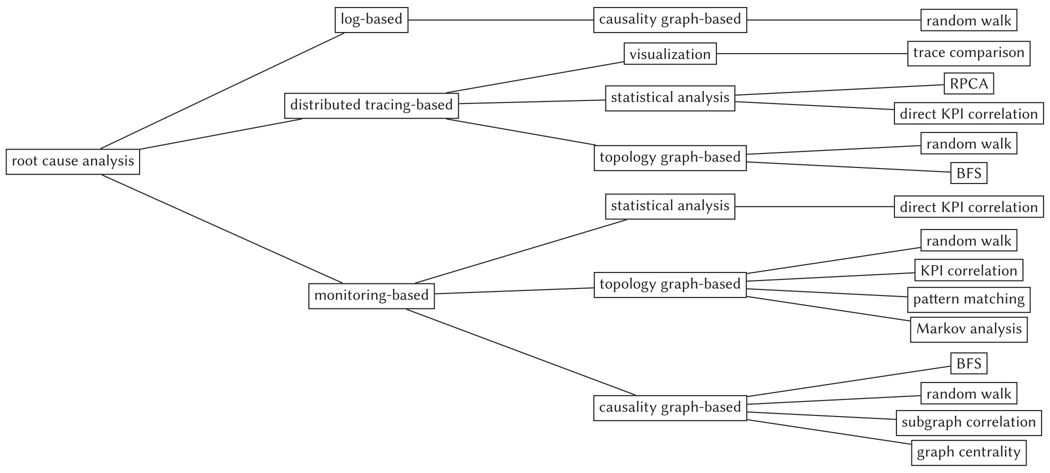

Fig. 3. A taxonomy for classifying techniques for root cause analysis in multi-service applications.

that can possibly occur in an application, and then to check whether newly generated traces are similar enough to the stored ones. This approach, however, results to be quite time consuming, bringing two out of the three techniques enacting trace comparison to be suited only for offline anomaly detection. Meng et al. [54] and Wang et al. [88] indeed explicitly state that their techniques are too time-consuming to be used online on medium-/large-scale applications.

The latter is a major limitation for Meng et al. [54] and Wang et al. [88], as well as for Jin et al. [35], with respect to all other surveyed techniques. Meng et al., Wang et al. and Jin et al. are indeed proposing techniques that can effectively detect anomalies in multi-service application only *offline*, based on the offline processing of the traces produced by their services. All other techniques can instead effectively detect anomalies also online while a multi-service application is running, hence also allowing to promptly react to corresponding failures, if needed.

## 5 ROOT CAUSE ANALYSIS

We now survey the existing techniques for determining the possible root causes for anomalies observed in multi-service applications. Following the taxonomy in Figure 3, and similarly to Section 4, we separately present the techniques directly processing the logs produced by application services (Section 5.1), from those requiring some instrumentation to processing the traces produced by application services (Section 5.2) or the KPIs monitored on such services (Section 5.3). This again provides a first distinction on the applicability of the surveyed techniques, based on which instrumentation should be enacted on a multi-service application to exploit the surveyed techniques. As we already observed in Section 4, log-based application techniques do not need any instrumentation other than native logging. Distributed tracing-based and monitoring-based techniques instead require application services to be instrumented to feature distributed tracing, or to be accompanied by monitoring agents collecting their KPIs, respectively.

The taxonomy in Figure 3 introduces an additional level between the type of input (viz., logs, traces, or monitored KPIs) and the actual method used. It indeed distinguishes those techniques directly processing such inputs, either through some visualisation support or by means of statistical analysis, from those building some support structure to analyse such inputs, such as topology graphs or causality graphs. We will follow this categorisation while surveying the considered root cause analysis techniques by discussing each category in a different subsection.

### 5.1 Log-Based Root Cause Analysis Techniques

Root cause analysis can be enacted by only considering the logs natively produced by the services forming an application (and without requiring further instrumentation like distributed tracing or monitoring agents). This is done by processing the application logs to derive a "causality graph", whose vertices model application services and whose directed arcs model that an anomaly in the source service may possibly cause an anomaly in the target service. The causality graph is then visited to determine a possible root cause for an anomaly observed on a multi-service application.

The above approach is adopted by Aggarwal et al. [1] to determine the root causes for functional anomalies observed on the frontend of a multi-service application faced by end-users when such an application fails. Aggarwal et al. consider the frontend and the application services that logged error events. They model the logs of considered services as multivariate time series, and they compute Granger causality tests [4] on such time series to determine causal dependencies among the errors logged by the corresponding services. The obtained cause-effect relations, together with an input application specification, are used to derive a causality graph. The latter is a graph whose nodes model application services, and where an arc from a service to another that the errors logged by the source service can have caused those logged by the target service. The services that, according to the causality graph, can have caused directly or indirectly the error observed on the application frontend are considered as its possible root causes. The candidate services are scored by visiting the causality graph with the random walk-based algorithm proposed by MonitorRank [38] (which we present in Section 5.2.3). Aggarwal et al. then return the highest scored candidate, which is considered the most probable root cause for the observed functional anomaly.

By cutting off all possible root causes other than the highest scored, Aggarwal et al. minimize the effort put on application operators: when the returned root cause is truly the root cause of an observed anomaly, they indeed only have to consider that possible cause, instead of browsing among other possible causes that would actually be "false positives". At the same time, cutting off other possible root causes may also result in dropping also the true cause of an observed anomaly if such cause would not be the highest-ranked. In such a case, the enacted root cause analysis would actually be useless, and this is the main reason why all other surveyed techniques accept tend to return a set of possible root causes (including false positives), often ranked so that higher ranks are assigned to the root causes that have higher chances to have caused an observed anomaly.

### 5.2 Distributed Tracing-Based Root Cause Analysis Techniques

The information available in distributed traces can be exploited to support application operators in determining the possible root causes for anomalies observed in their applications. In this perspective, the most basic technique consists of provides a systematic methodology to visually compare traces, so that application operators can manually determine what went wrong in the "anomalous traces", viz., the traces corresponding to observed anomalies. While such an approach minimizes the risk for false positives/negatives, it puts all the analysis efforts on application operators, by anyhow providing them with the guidelines and the tool support needed to enact such analysis.

The other techniques instead fully automate the root cause analysis based on two different alternatives, at the price of potentially returning false positives or excluding false negatives. One possibility is to enact statistical analyses directly on traces to detect which services experienced anomalies, which may have possibly caused the observed anomalies. The other possibility is determining the topology of an application (viz., a directed graph whose vertices model the application services and whose directed arcs model the service interactions) from available traces and to use such topology to drive the search for possible root causes of anomalies observed on an application.

*5.2.1 Visualization-Based Analysis.* Zhou et al. [98] and GMTA [27] allow application operators to manually determine the possible root causes for application-level anomalies, based on a trace comparison methodology to be enacted with the support of a trace visualisation tool. Zhou et al. propose to use ShiViz [7] to visualise the traces produced by a multi-service application as interactive time-space diagrams. This allows to determine the root causes for the traces corresponding to some anomaly observed in the application frontend by means of a pairwise comparison of traces. The idea is that the root causes for an anomalous trace are contained in a portion of such trace that is different from what contained in a successful trace for the same task, whilst shared with other anomalous traces. Zhou et al. hence propose to consider the traces corresponding to the observed anomaly and a set of reference, successful traces for the same set of tasks. They then propose to compare a successful trace and an anomalous trace corresponding to the same task. This allows obtaining a set of so-called "diff ranges", each of which corresponds to multiple consecutive events that are different between the two traces. Zhou et al. propose to repeat this process for any pair of successful and anomalous traces for the same task by always restricting the set of "diff ranges" to those that are shared among all analysed traces. The "diff ranges" obtained at the end of the process are identified as the possible root causes for the observed anomaly.

GMTA [27] provides a framework to collect, process, and visualise traces, which can be used to determine the root causes for functional anomalies observed on a multi-service application. GMTA collects the traces produced by the application services, and it automatically process them to determine interaction "paths" among services. A path is an abstraction for traces, which only models the service interactions in a trace and their order with a tree-like structure. GMTA also groups paths based on their corresponding "business flow", defined by application operators to indicate which operations could be invoked to perform a task and in which order. GMTA then allows to manually enact root cause analysis by visually comparing the paths corresponding to traces of successful executions of a given business flow with that corresponding to the anomalous trace where the functional anomaly was observed. This allows to determine whether the paths diverge and, if this is the case, which events possibly caused the paths to diverge. In addition, if the anomalous traces include error events, GMTA visually displays such errors together with the corresponding error propagation chains. This further allows to determine the services that first logged error events in the anomalous trace, hence possibly causing the observed anomaly.

Both Zhou et al. [98] and GMTA [27] allow to determine the root causes for anomalies observed on the frontend of an application, viz., the events that caused the trace to behave anomalously, and their corresponding services. They also highlight that by looking at the root cause events, or by inspecting the events logged by the corresponding services close in time to such events, application operators can determine the reasons for such services to cause the observed anomaly: whether they failed internally, because of some unexpected answer from the services they invoked, or because of environmental reasons (e.g., lack of available computing resources). This comes at the price of putting all root cause analysis efforts on application operators, who are required to *manually* enact such analysis by following the given guidelines and by exploiting the given tool support.

*5.2.2 Statistical Analysis.* CloudDiag [56] and TraceAnomaly [45] automatically determine the possible root causes of anomalies observed on the frontend of an application by directly analysing the response times in the service interactions involved in collected traces. Their aim is to determ other anomalies that may have possibly caused the observed one, based on different setups and by applying different methods. As for setups, CloudDiag must be attached to running application deployment to collect the produced traces, and it must be manually triggered by application operators when a performance anomaly is observed on the application frontend. TraceAnomaly instead directly processes the traces produced by an application, whichever is the way they are collected, and it comes as an already integrated solution that automatically enacts root cause analysis in pipeline to the functional/performance anomaly detection described in Section 4.2.1.

CloudDiag [56] processes collected traces to determine "anomalous groups", viz., groups of traces sharing the same service call tree and where the coefficient of variation [72] of response times in service interactions is above a given threshold. Anomalous groups are processed to identify the services that most contributed to the anomaly, based on Robust Principal Component Analysis (RPCA [13]). CloudDiag represents the traces in a category as a matrix, and it exploits RPCA to determine the columns corresponding to services with anomalous response times. Such services constitute the possible root causes for the observed anomaly, which are returned by CloudDiag. Root causing services are ranked based on the number of times they appeared in anomalous categories: the larger the number of times is, the more possible it is that service caused the observed anomaly.

TraceAnomaly [45] instead differently determines the possible root causes for the functional and performance anomalies it detects, based on the way they are detected. As functional anomalies are detected when previously unseen call paths occur in a trace, their root causes obviously coincides with the services starting the first interaction of the previously unseen call paths. Performance anomalies are instead detected by means of a neural network, which classifies newly produced traces as normal/anomalous. If a performance anomaly is detected on a trace, TraceAnomaly determines its possible root causes by correlating the response times of the service interactions in the anomalous trace with their average response times in normal conditions: each service interaction whose response time significantly deviates from the average response time is considered as anomalous. This results in one or more sequences of anomalous service interactions. The root causes for the detected performance anomaly are extracted from such anomalous service interactions sequences by returning the last invoked services as possible root causes without ranking them.

*5.2.3 Topology Graph-Based Analysis.* The approach of building and processing topology graphs for determining the possible root causes of performance anomalies is adopted by MonitorRank [38] and MicroHECL [44]. They indeed both rely on services to produce interaction traces, including the start and end times of service interactions, their source and target services, performance metrics (viz., latency, error count, and throughput), and the unique identifier of the corresponding user request. This information is then processed to reconstruct service invocation chains for the same user request, which are then combined to build the application topology. MonitorRank and MicroHECL, however, differ in their applicability and setup: MonitorRank determines the possible root causes for application-level anomalies, needing to get attached to the target application deployment to observe the produced traces. MicroHECL instead directly process collected traces to determine the possible root causes for service-level anomalies. MonitorRank and MicroHECL also differ in the method applied to enact root cause analysis: MonitorRank performs a random walk on the topology graph, whereas MicroHECL explores it through a **breadth-first search** (**BFS**).

MonitorRank [38] periodically processes the traces produced by application services to generate a topology graph for each time period. Based on such graphs, MonitorRank allows to determine the possible root causes of a performance anomaly observed on the application frontend in a given time period. This is done by running the personalised PageRank algorithm [32] on the topology graph corresponding to the time period of the observed anomaly to determine its root causes. MonitorRank starts from the frontend service, and it visits the graph by performing a random walk with a fixed number of random steps. At each step, the next service to visit is randomly picked in the neighbourhood of the service under visit, namely among the service under visit, those it calls, and those calling it. The pickup probability of each neighbour is proportional to its relevance to the anomaly to be explained, computed based on the correlation between its performance metrics and those of the frontend service (with such metrics available in the traces produced in the considered time period). MonitorRank then returns the list of visited services, ranked by the number of times they have been visited. The idea is that, given that services are visited based on the correlation of their performances with those of the application frontend, the more are the visits to a service, the more such service can explain the performance anomaly observed on the application frontend.

MicroHECL [44] instead builds the topology based on the traces produced in a time window of a given size, which ends at the time when a service experienced the performance anomaly whose root causes are to be determined. Based on such topology, MicroHECL determines the anomaly propagation chains that could have led to the observed anomaly. This is done by starting from the service where the anomaly was observed and by iteratively extending the anomaly propagation chains along with the possible anomaly propagation directions: at each iteration, the services that can be reached through anomaly propagation directions are included in the anomaly propagation chains if they experienced corresponding anomalies. This is checked by applying formerly trained machine learning models for offline detection of anomalies on each service, under the assumption that the application behaved consistently with the offline trained models in the considered time window. When anomaly propagation chains cannot be further extended, the services at the end of each chain are considered as possible root causes for the initial anomalous service. MicroHECL finally ranks the possible root causes based on the Pearson correlation between their performance metrics and those of the service where the performance anomaly was initially observed.

## 5.3 Monitoring-Based Root Cause Analysis Techniques

Root cause analysis is also enacted by processing the KPIs monitored on application services throughout monitoring agents installed alongside such services. The techniques enacting such a kind of analysis can be partitioned into three sub-classes, based on the method they apply to determine the possible root causes for an anomaly observed on a service. A possibility is to enact statistical analyses directly on monitored KPIs to determine other services that experienced anomalies, which may have possibly caused that actually observed. Other alternatives rely on topology/causality graphs modelling the dependencies among the services in an application, as they visit such graphs to determine which services may have possibly caused the observed anomaly.

As we will see, most of the monitoring-based root cause analysis techniques rely on correlation as the driver to identify the possible root causes of an observed anomaly. The idea is that the higher is the correlation between the KPIs monitored on an anomalous service and those monitored in parallel on other services, the higher is the probability that the latter has caused the anomaly observed on the former. However, given that correlation is not ensuring causation [12], the actual root cause for an anomaly observed in service may not be amongst the services whose monitored KPIs are highly correlated with that of the anomalous service, but rather in some other service. Spurious correlations hence impact the techniques exploiting KPI correlation to enact statistical analyses or to drive the search for root causes in a topology/causality graph. This raises the risk for false negatives, which is typically tackled by returning multiple possible root causes, often also ranked to reduce the application operators' efforts due to the false positives therein.

*5.3.1 Statistical Analysis.* The root causes of an anomaly detected on a service can be determined by identifying application services whose monitored KPIs were anomalous in parallel with the detected anomaly. This technique is enacted by $\epsilon$-diagnosis [76] and Wang et al. [86] to determine the possible root causes for the anomalies they detect, with $\epsilon$-diagnosis enacting the anomaly detection described in Section 4.3.3, whilst Wang et al. relying on an external monitoring tool [20] to detect anomalies. $\epsilon$-diagnosis considers the time series of KPIs monitored on application services, from which it extracts two same-sized samples. A sample corresponds to the time period where the frontend anomaly was detected, while the other sample corresponds to a time period where no anomaly was detected. To determine whether a KPI can have caused the detected anomaly, the similarity between the two samples is computed: if the similarity is below a given tolerance threshold, the two samples are diverse enough to consider the change in such KPI as anomalous and possibly causing the frontend anomaly. The list of anomalous KPIs, together with the corresponding services, is returned by $\epsilon$-diagnosis as the set of possible root causes.

Wang et al. [86] instead process the logs natively produced by application services and the time series of KPIs monitored on such services by devoted monitoring agents. They first train a long-short term memory neural network for each service, which models its baseline behaviour in failure-free runs. When an anomaly is observed on the application, each trained neural network processes the logs produced by the corresponding service in the time interval when the anomaly was observed. The outputs of all trained neural networks are combined to obtain a time series of anomaly degrees for the application. Wang et al. then apply mutual information [64] to determine the dependency of the anomaly degree on the KPIs monitored on application services in the same interval. The obtained correlation values are used to provide the list of KPIs that may have possibly caused the observed application anomaly, together with the services on which they have been monitored. Differently from $\epsilon$-diagnosis [76], the root causes are ranked by decreasing correlation value, to first show the services' KPIs having higher probability to have caused the observed anomaly.

PAL [63] and FChain [62] also exploit offline detection of anomalous KPIs to determine ranked lists of possible root causes for performance anomalies detected on the frontend of a multi-service application. They, however, differ from $\epsilon$-diagnosis [76] and Wu et al. [86] in the granularity of identified root causes (anomalous services, rather than anomalous services' KPIs), and since they do not come in an integrated pipeline with any anomaly detection mechanism, but rather rely on the application operator to trigger them. PAL and FChain analyse system-level KPIs (viz., CPU usage, free memory, and network traffic) monitored on application services by exploiting a combination of CUSUM (Cumulative Sum) charts and bootstrapping [6]. CUSUM charts measure the magnitude of changes for the monitored KPI values, both in the original sequence of monitored values and in "bootstraps", with the latter being permutations of the monitored KPI values obtained randomly reordering them. If the magnitude of change in the original sequence is higher than in most bootstraps, the original sequence is considered anomalous and the time when the anomaly started is identified based on its CUSUM charts. To reduce false positives, PAL and FChain check whether anomalies affected all application services, in which case the anomaly is classified as due to external reasons (e.g., workload spikes). Instead, if only a subset of application services are anomalous, they are considered as the possible root causes for the frontend anomaly, and they are returned sorted by chronologically ordering the start times of their anomalies. The earliest anomalies are indeed considered the most probable root causes, since they may have propagated from their corresponding services to other services, up to causing the observed frontend anomaly.

*5.3.2 Topology Graph-Based Analysis.* MicroRCA [92], Wu et al. [91], Sieve [84], and Brandón et al. [9] automatically reconstruct the graph modelling the topology of a running application, which they then use to drive the analysis of the possible root causes for observed anomalies. DLA [74] also exploits a topology graph to identify the possible root causes for observed anomalies but relies on application operators to provide such a graph. The above techniques, however, differ in their setup: MicroRCA, Wu et al., and DLA come in an integrated pipeline with anomaly detection (Section 4.3.1), but require the target application to be deployed with K8s. Sieve and Brandón et al.

are instead technology-agnostic but need to be triggered and provided with the anomaly whose root causes are searched for. They also differ in the type and granularity of explained anomalies, in the granularity of identified root causes, and in the methods applied over topology graphs to determine the possible root causes of observed anomalies. All such methods are, however, based on either correlation or probabilistic analyses, which are inherently being subject to false positives and false negatives. All the above-listed techniques hence mitigate the risk for false negatives by returning multiple possible root causes. Instead, the impact of false positives on application operators' efforts is mitigated only by Brandón et al., who rank the possible root causes with graph similarity-based heuristics.

*Random Walk.* MicroRCA [92] considers the status of the K8s deployment of an application when an anomaly was detected: which containerised services were running, on which nodes they were hosted, and the interactions that occurred among services. MicroRCA exploits this information to generate a topology graph, whose vertices model the running application services and the node used to host them and whose oriented arcs model service interactions/hosting. The topology graph is enriched by also associating each vertex with the time series of KPIs monitored on the corresponding service or node. MicroRCA then extracts an "anomalous subgraph" by picking the vertices of the topology graph that correspond to the services where anomalies were detected, by including the vertices and arcs corresponding to the interactions to/from the anomalous services, and by including other vertices and arcs from the topology graph so as to obtain a connected subgraph. The anomalous subgraph constitutes the search space where MicroRCA looks for the services that may have possibly caused the detected anomalies. The actual search for the root causing services is enacted by adapting the random walk-based search proposed in MonitorRank [38] (Section 5.2.3) to work with the information modelled by the anomalous subgraph.

Wu et al. [91] refine MicroRCA [91] by determining the possible root causes for detected anomalies at a finer granularity: they indeed return the KPIs monitored on application services that may have caused an anomaly, rather than the application services themselves. Wu et al. reuse the topology-driven search proposed by MicroRCA to determine the services that can have possibly caused the detected anomalies. They then process the KPIs monitored on root causing services with an autoencoder [23], a neural network that learns to encode input values and to suitably reconstruct them from their encoding. Wu et al. first train an autoencoder with values for such KPIs monitored while no anomaly was affecting the application. They then feed the trained autoencoder with the KPI values monitored on the application services when the considered anomalies were detected. The KPIs whose values are not well-reconstructed, together with their corresponding services, are considered as the possible root causes for the detected anomalies.

*Other Methods.* Sieve [84] and Brandón et al. [9] monitor network system calls to collect information on service interactions, which they then exploit to automatically reconstruct the topology of a running application. Sieve is the only monitoring- and topology graph-based solution focusing on application-level anomalies: when a performance anomaly is observed on the application frontend, Sieve exploits the topology to drive the analysis of the possible root causes for the observed anomaly. It first reduces the dimensionality of considered KPIs by removing those whose variance is too small to provide some statistical value, and by clustering the KPIs monitored on each service so as to consider only one representative KPI for each cluster, namely that closest to the centroid of the cluster. Sieve then explores the possibilities of a service's representative KPIs to influence other services' KPIs using a pairwise comparison method: each representative KPI of each service is compared with each representative KPIs of another service. The choice of which services to compare is driven by the application topology: Sieve indeed enacts a pairwise comparison of KPIs monitored only on interacting services by exploiting Granger causality tests [4] to determine whether changes in the values monitored for a service's KPI were caused by changes in those monitored for another service's KPI. The obtained cause-effect relations are used to only keep the KPIs whose changes were not caused by other services. Such KPIs, together with their corresponding services, are considered as the possible root causes for the observed application-level performance anomaly.

Brandón et al. [9] instead allow identifying the possible root causes for both functional and performance service-level anomalies. Brandón et al. enrich the derived topology by associating each service with its monitored KPIs and with the possibility of marking services as anomalous, by however relying on existing anomaly detection techniques to actually detect anomalous services. Whenever the root cause for an anomaly in a service is looked for, the proposed system search within the neighbourhood of the anomalous service. It indeed extracts a subgraph including that modelling the anomalous services and those close to it up to a given distance. The subgraph is compared with "anomaly graph patterns", provided by the application operator to model already troubleshooted anomalies whose root causing service is known. If the similarity between the extracted subgraph and an anomaly graph pattern is above a given threshold, the corresponding root causing service is included among the possible root causes for the considered anomaly. To reduce the impact of false positives on application operators' efforts, when multiple roots causing services are detected, they are ranked by the similarity between the corresponding anomaly graph patterns and the anomalous subgraph. The idea here is that the closer is an anomalous subgraph to a given anomaly graph pattern, the higher is the probability that it actually corresponds to the anomaly propagation that caused the anomaly being explained.

Differently from all above-discussed techniques, DLA [74] require the application topology to be provided as input by the application operator instead of automatically determining it. The input topology must specify the services forming an application, the containers used to run them, and the VMs where the containers are hosted, together with the communication/hosting relationships between them. DLA transforms the input topology into a Hierarchical Hidden Markov Model (HHMM) [14] by automatically determining the probabilities of an anomaly affecting a service, container, or VM to be caused by the components it relates to. The possible root causes for the detected anomaly are then determined by exploiting the obtained HHMM to compute the likelihood of the anomaly to be generated by KPI anomalies monitored at container- or VM-level. More precisely, DLA computes the path in the HHMM that has the highest probability to have caused the anomaly observed on a service. The obtained path is then used to elicit the KPIs (and corresponding components) that most probably caused the observed anomaly.

*5.3.3 Causality Graph-Based Analysis.* Various existing techniques determine the possible root causes for an anomaly observed on some application services by suitably visiting an automatically derived causality graph [18, 19, 24, 42, 43, 48–50, 87]. The vertices in the causality graph typically model application services, with each oriented arc indicating that the performance of the source service depends on that of the target service. Causality graphs are typically built by exploiting the well-known PC-algorithm [82]. More precisely, the causality graph is built by starting from a complete and undirected graph, which is refined by dropping and orienting arcs to effectively model causal dependencies among the KPIs monitored on services. The refinement is enacted by pairwise testing conditional independence between the KPIs monitored on different services: if the KPIs monitored on two services result to be conditionally independent, the arc between the services is removed. The remaining arcs are then oriented based on the structure of the graph.

The existing causality graph-based techniques, however, differ in the methods applied to process causality graphs to determine the possible root causes for observed anomalies. The most common methods are visiting the causality graph through a BFS or a random walk, but there also exist techniques enacting other causality graph-based analysis methods. In any case, being the causality graph essentially built based on the correlation between the KPIs monitored on the services forming an application, the enacted analysis may exclude false negatives or include false positives, e.g., because of spurious correlations [12]. To this end, all the monitoring- and causality graph-based root cause analysis techniques reduce the risk for false negatives by returning multiple possible root causes whilst at the same time mitigating the impact of false positives by ranking the returned root causes. The idea is again that the highest ranks are assigned to the possible root causes that most probably caused an observed anomaly. These should hence include the true cause for such an anomaly, which would be earlier checked by the application operator, saving her from wasting resources and efforts in checking the other possible root causes.

*BFS.* CauseInfer [18, 19] and Microscope [24, 42] determine the possible root causes for the response time anomalies they detect on the frontend of a multi-service application, hence providing an integrated solution for both anomaly detection and root cause analysis. They rely on monitoring agents to collect information on service interactions and their response times, with CauseInfer being technology-agnostic, whilst Microscope requiring the target application to be deployed with K8s. The monitored information is processed with the PC-algorithm [82] to build a causality graph. The causality graph is enriched by also including arcs modelling the dependency of each service on the services it invokes. CauseInfer and Microscope then enact root cause inference by recursively visiting the obtained causality graph, starting from the application frontend. For each visited service, they consider the services on which it depends: if all such services were not affected by response time anomalies, the currently visited service is considered a possible root cause. Otherwise, CauseInfer and Microscope recursively visit all services that were affected by response time anomalies to find the possible root causes for the anomaly observed on the currently visited service. To determine anomalies in the response times of visited services, CauseInfer computes an anomaly score based on CUSUM statistics [6] on the history of monitored response times. Microscope instead exploits standard deviation-based heuristics to determine whether the latest monitored response times were anomalous. When all possible root causing services have been identified, CauseInfer and Microscope return them ranked, based on their anomaly score or on the Pearson correlation between their response times and that of the application frontend, respectively.

Differently from CauseInfer [18, 19] and Microscope [24, 42], Qiu et al. [68] does not come as an integrated anomaly detection and root cause analysis solution, requiring the application operator to trigger it on the KPIs monitored on application services when an anomaly is detected. At the same time, Qiu et al. consider finer-grained performance anomalies (service-level, rather than application-level) and provide finer-grained possible root causes (root causing services' KPIs, rather than services). To determine such root causes, Qiu et al. exploit the PC-algorithm [82] to build causality graphs whose vertices correspond to KPIs monitored on application services (rather than to the services themselves). Qiu et al. also enrich the causality graph by including arcs among KPIs also when a dependency between the corresponding services is indicated in the application specification provided as input by the application operator. The arcs in the causality graph are also weighted, with each arc's weight indicating how much the source KPI influences the target KPI. The weight of each arc is computed by determining the Pearson correlation between the sequence of significant changes in the source and target KPIs, with significant changes being determined by processing the corresponding time series of monitored KPI values with the technique proposed by Luo et al. [47]. The root cause analysis is then enacted with a BFS in the causality graph, starting from the KPI whose anomaly was observed. This allows to determine all possible paths outgoing from the anomalous KPI in the causality graph, which all correspond to possible causes for the anomaly observed on such KPI. To mitigate the impact of false positives, the paths are sorted based on the sum of the weights associated with the edges in the path by also prioritizing shorter paths in the case of paths whose sum is equivalent. In this way, Qiu et al. prioritize the paths including the KPIs whose changes most probably caused the observed anomaly.

*Random Walk.* CloudRanger [87], MS-Rank [48, 49], and AutoMAP [50] determine the possible root causes for application-level performance anomalies, with CloudRanger coming as an integrated anomaly detection (Section 4.3.1) and root cause analysis solution, whereas MS-Rank and AutoMAP relying on the application operator to trigger them on monitored KPIs when an anomaly is detected. They all exploit the PC-algorithm [82] to build a causality graph by processing the monitor response times, throughput, and power, with MS-Rank and AutoMAP also considering the services' availability and resource consumption. The obtained causality graph then provides the search space where CloudRanger, MS-Rank, and AutoMAP perform a random walk to determine the possible root causes for a performance anomaly observed on the application frontend. The random walk starts from the application frontend and it consists of repeating $n$ times (with $n$ equal to the number of services in an application) a random step to visit one of the services in the neighbourhood of the currently visited service, which includes the currently visited service and the services that can be reached by traversing forward/backward arcs connected to it. At each iteration, the probability of visiting a service in the neighbourhood is proportional to the correlation of its KPIs with those of the application frontend. CloudRanger, MS-Rank, and AutoMAP rank the application services based on how many times they have been visited, under the assumption that most visited services constitute the most probable root causes for the anomaly observed on the frontend.

Differently from the above-discussed techniques [48–50, 87], MicroCause [55] determines finer-grained possible root causes (services' KPIs, rather than services) for finer-grained performance anomalies (service-level, rather than application-level). It does so by still exploiting the PC-algorithm [82] to build a causality graph whose vertices correspond to KPIs monitored on application services (rather than to the services themselves). In particular, MicroCause considers the response time monitored on application services, their error count, queries per second, and resource consumption. MicroCause then determines the KPIs that experienced anomalies and the time when their anomaly started by exploiting the SPOT algorithm [79]. MicroCause also computes an anomaly score for each KPI, essentially by measuring the magnitude of its anomalous changes. This information is then used to drive a random walk over the causality graph to determine the possible root causes for an anomalous KPI being observed on a service. MicroCause starts from the anomalous KPI and repeats a given number of random steps, each consisting of staying in the currently visited KPI or randomly visiting a KPI that can be reached by traversing forward/backward arcs connected to the currently visited KPI. At each step, the probability of visiting a KPI is based on the time and score of its anomaly, if any, and on the correlation of its values with those of the anomalous KPI whose root causes are being searched. The KPIs visited during the random walk constitute the set of possible root causes returned by MicroCause, ranked based on their anomaly time and score.

*Other Methods.* FacGraph [43] exploits the PC-algorithm [82] to build causality graphs, whilst also assuming the same frontend anomaly to be observed in multiple time intervals. It indeed exploits the PC-algorithm to build multiple causality graphs, each built on the latency and throughput monitored on application services during the different time intervals when the frontend anomaly was observed. FacGraph then searches for anomaly propagation subgraphs in the obtained causality graphs, with each subgraph having a tree-like structure and being rooted in the application frontend. The idea is that an anomaly originates in some application services and propagates from such services to the application frontend. All identified subgraphs are assigned an anomaly score based on the frequency with which they appear in the causality graphs. Only the subgraphs whose anomaly score is higher than a given threshold are kept, as they are considered as possible explanations for the performance anomaly observed on the application frontend. FacGraph then returns the set of services corresponding to leaves in the tree-like structure of each of the kept subgraphs, which constitute the possible root causes for the observed frontend anomaly. The returned root causes are also ranked by anomaly score, under the assumption that the highest scores are assigned to the services whose anomaly most probably caused the observed application-level performance anomaly.

LOUD [51] instead determines the root causes of service-level performance anomalies at a finer granularity by building causality graphs whose vertices correspond to any KPIs monitored on application services (rather than to the services themselves). In addition, LOUD comes as integrated anomaly detection (Section 4.3.1) and root cause analysis solution, hence automatically determining the possible root causes of the service-level performance anomalies it detects. LOUD relies on the IBM ITOA-PI [30] to determine the possible root causes for anomalous KPIs monitored on application services. The IBM ITOA-PI is indeed exploited to automatically build and process a causality graph. This is done under the assumption that the anomalous KPIs related to a detected anomaly are highly correlated and form a connected subgraph of the causality graph. In particular, LOUD assumes that the anomalous behaviour of the service whose KPIs are anomalous is likely to result in the anomalous behaviour of services it interacts with, either directly or through some other intermediate services. Based on this assumption, LOUD exploits graph centrality to identify the anomalous KPIs that best characterize the root cause of a detected anomaly: the KPIs with the highest centrality scores likely correspond to the services that caused the detected anomaly.

## 5.4 Summary

Table 2 recaps the surveyed root cause analysis techniques by distinguishing their classes, viz., whether they are log-based, distributed tracing-based, or monitoring-based, as well as the method they apply to determine the possible root causes for an observed anomaly. The table classifies the surveyed techniques based on whether they identify the possible root causes for functional or performance anomalies, observed at application-level or on specific services in the application. Table 2 also provides some insights on the setup required by the surveyed root cause analysis techniques, indicating whether they come already integrated with some anomaly detection technique, as well as on the required inputs. The table indeed lists the artifacts that must be provided to the surveyed root cause analysis techniques, as input needed to actually search for possible root causes of observed anomalies. Finally, Table 2 recaps the root causes identified by the surveyed techniques, by distinguishing whether they identify the services, events, or KPIs that possibly caused an observed anomaly, and whether the identified root causes are ranked, with higher ranks assigned to those that have a higher probability to have caused the observed anomaly.

Taking Table 2 as a reference, we hereafter summarise the surveyed root cause analysis techniques, all allowing to determine the possible root causes for anomalies observed on multi-service applications. Ten of the surveyed root cause analysis techniques [18, 19, 24, 42, 45, 51, 74, 76, 86, 87, 91, 92] already enact such a root cause analysis in the pipeline with anomaly detection. Whenever an anomaly is detected to affect the whole application or one of its services, a root cause analysis is automatically started to determine the possible root causes for such an anomaly. In this case, the pipeline of techniques can be used as-is to automatically detect anomalies and their possible root causes. All other techniques instead determine the possible root causes for anomalies observed with external monitoring tools, by end-users, or by application operators [1, 9, 27, 38, 43, 44, 48–50, 55, 56, 62, 63, 68, 84, 98]. By combining the information on the type and granularity of detected/analysed anomalies available in Tables 1 and 2, application operators

Table 2. Classification of Root Cause Analysis Techniques, Based on Their *Class* (L = Log-Based, DT = Distributed Tracing-Based, M = Monitoring-Based), The Applied *Method*, whether The *Detection* of Analysed Anomalies is Integrated or Done with External Tools (I = Integrated Pipelines, E = External Tools), the *Type* (F = Functional, P = Performance) and *Granularity* of Explained Anomalies (A = Application-Level, S = Service-Level), The Identified *Root Causes*, and The *Input* They Need to Run

| | | | Anomaly | | | | |
|---|---|---|---|---|---|---|---|
| | **Class** | **Method** | **Det.** | **Type** | **Gran.** | **Root Causes** | **Needed Input** |
| Aggarwal et al. [1] | L | random walk on causality graphs | E | F | A | service | runtime logs, app spec |
| Zhou et al. [98] | DT | visual trace comparison | E | F, P | A | events | previous and runtime traces |
| GMTA [27] | DT | visual trace comparison | E | F, P | A | services | previous and runtime traces |
| CloudDiag [56] | DT | RPCA on traces | E | P | A | ranked services | app deployment |
| TraceAnomaly [45] | DT | direct KPI correlation | I | F, P | A | services | previous and runtime traces |
| MonitorRank [38] | DT | random walk on topology graphs | E | P | A | ranked services | app deployment |
| MicroHECL [44] | DT | BFS on topology graphs | E | P | S | ranked services | previous and runtime traces |
| $\epsilon$-diagnosis [76] | M | direct KPI correlation | I | P | A | service KPIs | K8s deployment |
| Wang et al. [86] | M | direct KPI correlation | I | F, P | S | service KPIs | runtime logs, monitored KPIs |
| PAL [63] | M | direct KPI correlation | E | P | A | ranked services | app deployment |
| FChain [62] | M | direct KPI correlation | E | P | A | ranked services | app deployment |
| MicroRCA [92] | M | random walk on topology graphs | I | P | S | services | K8s deployment, workload generator |
| Wu et al. [91] | M | random walk on topology graphs | I | P | S | service KPIs | K8s deployment, workload generator |
| Sieve [84] | M | KPI correlation on topology graphs | E | P | A | service KPIs | app deployment |
| Brandón et al. [9] | M | pattern matching on topology graphs | E | F, P | S | ranked services | app deployment, anomaly patterns |
| DLA [74] | M | Markov analysis on topology graphs | I | P | S | service KPIs | K8s deployment, app spec |
| CauseInfer [18, 19] | M | BFS on causality graphs | I | P | A | ranked services | target app deployment |
| Microscope [24, 42] | M | BFS on causality graphs | I | P | A | ranked services | K8s app deployment |
| Qiu et al. [68] | M | BFS on causality graphs | E | P | S | ranked service KPIs | monitored KPIs, app spec |
| CloudRanger [87] | M | random walk on causality graphs | I | P | A | ranked services | app deployment, prev. monitored KPIs |
| MS-Rank [48, 49] | M | random walk on causality graphs | E | P | A | ranked services | monitored KPIs |
| AutoMAP [50] | M | random walk on causality graphs | E | P | A | ranked services | monitored KPIs |
| MicroCause [55] | M | random walk on causality graphs | E | P | S | ranked service KPIs | monitored KPIs |
| FacGraph [43] | M | correlation of causality graphs | E | P | A | ranked services | monitored KPIs |
| LOUD [51] | M | centrality on causality graphs | I | P | S | ranked service KPIs | a pp deployment, workload generator |

can determine which root cause analysis technique can be used in pipeline with a given anomaly detection technique, after applying the necessary integration to both techniques to let them successfully interoperate.

Whichever is the way of detecting anomalies, their root cause analysis is typically enacted on a graph modelling the multi-service application where anomalies have been observed (Table 2). A possibility is to visit a graph modelling the application topology, which can be automatically reconstructed by monitoring service interactions [9, 84, 91, 92] or from distributed traces [38, 44], or which must be provided as input by the application operator [74]. The most common, graph-based approach is, however, reconstructing a causality graph from the application logs [1] or from the KPIs monitored on the services forming an application [18, 19, 24, 42, 43, 48–51, 55, 68, 87], typically to model how application services influence each other. The vertices in a causality graph either model the application services [18, 19, 24, 42, 43, 48–50, 87] or the KPIs [51, 55, 68] monitored on such services, whereas each arc indicates that the performance of the target service/KPI depends on that of the source service/KPI. Causality graphs are always obtained (but for the case of LOUD [51]) by applying the PC-algorithm [82] to the time series of logged events/monitored KPIs, as such algorithm is known to allow determining causal relationships in time series, viz., by identifying whether the values in a time series can be used to predict those of another time series.

The search can then be performed by visiting the topology/causality graph, starting from the application frontend in the case of application-level anomalies [1, 18, 19, 24, 38, 42, 43, 48–50, 84, 87] or from the KPI/service where the anomaly was actually observed [44, 51, 55, 68, 91, 92]. Different methods are applied to visit the graph, the most intuitive being a BFS over the graph to identify all possible paths that could justify the observed anomaly [68]. To reduce the paths to be visited, the BFS is often stopped when no anomaly was affecting any of the services that can be reached from the currently visited service, which is hence considered a possible root cause for the observed anomaly [18, 19, 24, 42, 44]. The most common method is, however, that first proposed in MonitorRank [38] and then reused by various techniques [48–50, 55, 87, 91, 92]. In this case, the graph is visited through a random walk, where the probability of visiting service with a random step is proportional to the correlation between the performances of such service and of the service where the anomaly was observed. Rather than returning all possible root causes, random walk-based techniques return a ranked list of possible root causes, assuming that the more a service gets visited through the random walk, the more is probable that it can have caused the observed anomaly [38, 48, 49].

Other graph-based root cause analysis techniques anyhow exist, either using the application topology to drive the pairwise comparison of monitored performances [84], or based on processing the whole topology/causality graph to determine the possible root causes for an observed anomaly [9, 51, 74]. As for processing the application topology, the alternatives are twofold: enacting pattern matching to reconduct the observed anomaly to some already troubleshooted situation [9], or transforming the topology into a probabilistic model to elicit the most probable root causes for the observed anomaly [74]. In the case of causality graph, instead, graph processing corresponds to computing graph centrality indexes to rank the KPIs monitored on services, assuming that the highest ranked KPIs best characterise the observed anomaly [51].

Quite different methods are trace comparison and direct KPI correlation. The idea of trace comparison is to provide application operators with graphical support to visualise and compare traces, so that they can troubleshoot the traces corresponding to anomalous runs of an application [27, 98]. Direct KPI correlation is instead based on the idea here is that an anomaly observed on a service can be determined by the co-occurrence of other anomalies on other services. The proposed method is hence to directly process the traces produced by application services [45, 56] or their monitored KPIs [62, 63, 76, 86] to detect co-occurring anomalies, with the latter being anomalies occurring in a time interval, including the time when the initially considered anomaly was observed. Such co-occurring anomalies are considered as the possible root causes for the observed anomaly.

## 6 DISCUSSION

We hereafter discuss the anomaly detection and root cause analysis techniques we surveyed, by first focusing on the tradeoff between the anomalies/root causes they can determine and their setup costs (Section 6.1). We then elaborate on the false positives/negatives that may affect the surveyed techniques (Section 6.2), and on the need for explainability and countermeasures (Section 6.3).

### 6.1 Setup Costs vs. Detected Anomalies and Identified Root Causes

The surveyed techniques achieve a different granularity in the anomalies they can detect or explain by requiring different instrumentations and applying different methods (Tables 1 and 2). In the case of anomaly detection, finer granularity is achieved independently from the required instrumentation, as this rather depends on the enacted method. The techniques comparing newly collected KPIs, logs, or traces against a baseline model turn out to be much finer in the granularity of detected anomalies, if compared with techniques enacting SLO checks. Indeed, whilst monitoring SLOs on the frontend of an application allows to state that the whole application is suffering from a performance anomaly, the techniques based on machine learning or trace comparison often allow to detect of functional or performance anomalies at the service-level. They indeed typically indicate which of its services are suffering from functional or performance anomalies, whilst also providing information on which are the KPIs monitored on a service, the events it logged, or the portion of a trace due to which a service is considered anomalous. In the case of supervised learning-based techniques [8, 21, 22, 52, 60, 97], they also indicate which failures may possibly correspond to the symptoms denoted by the detected anomalies. All such information is precious for application operators to further investigate on detected anomalies, e.g., to check whether they correspond to services truly failing and to determine why they have failed [11]. This comes at the price of requiring to execute training runs of multi-service applications to collect logs, traces, or KPIs to train the baseline models against which to compare those produced at runtime. In the case of supervised learning-based techniques, it is also required to provide what needed to automatically inject failures, or to explicitly label the collected logs, traces, or KPIs to distinguish those corresponding to normal runs from those corresponding to runs where specific failures occurred on specific services.

In the case of root cause analysis, instead, the level of detail for considered anomalies and identified root causes is instead much deeper for techniques requiring to instrument applications to feature distributed tracing or to include monitoring agents. The log-based technique by Aggarwal et al. [1] determines a single root cause, which corresponds to the service that most probably caused the functional anomaly observed on the frontend of an application. The available trace-based and monitoring-based techniques instead determine multiple possible root causes for application-level anomalies [18, 19, 24, 27, 38, 42, 43, 45, 48–50, 56, 62, 63, 76, 84, 87, 98] or service-level anomalies [9, 44, 51, 55, 68, 74, 86, 91, 92]. They can also provide more details on the possible root causes for an observed anomaly, from the application services that may have caused it [9, 18, 19, 27, 38, 43–45, 48–50, 56, 62, 63, 87, 92] up to the actual events [98] or KPIs monitored on such services [51, 55, 68, 74, 76, 84, 86, 91]. To help application operators in troubleshooting the multiple root causes identified for an observed anomaly, some techniques also rank the possible root causing services [9, 18, 19, 24, 38, 42–44, 48–50, 56, 62, 63, 87] or KPIs [51, 55, 68]. The idea is that highest-ranked root causes are more likely to have caused an observed anomaly, hence helping application operators to earlier identify the actual root causes for the observed anomaly, hence saving them from wasting resources and efforts in troubleshooting all other possible root causes.

In both cases, setup costs are the price to pay to reach a deeper level of detail for considered anomalies and identified root causes. Instrumentation costs may be saved if using log-based anomaly detection and root cause analysis techniques, which directly work on the event logs produced by application services. Such techniques indeed only assume application services to log events, which is typically the case, whilst not requiring any further application instrumentation. This is enough to train the baseline models and enact service-level anomaly detection [33, 34], still at the price of providing what is needed to train the baseline model against which to compare newly collected logs. The same does not hold for root cause analysis, as the only available log-based technique can determine the possible root causes only for application-level anomalies [1]. Instrumentation is hence a setup cost to pay to determine the possible root causes of service-level anomalies, unless distributed tracing is already suitably configured on application services or monitoring agents are already installed to collect the necessary KPIs from such services.

Setup costs also include the artifacts to be provided as input to the surveyed techniques and the integration of the chosen anomaly detection and root cause analysis techniques. As for inputs, the techniques exploiting machine learning to detect anomalies and identifying their root causes require to run the target application in training runs [21, 26, 51, 52, 65, 66, 74, 86, 91, 92, 97], or data from previous runs of the target application, either plain [8, 17, 27, 33–35, 38, 44, 45, 54, 58, 61, 87, 88, 98] or labelled to explicitly mark the runs where failures occurred [22, 60]. The surveyed techniques also often rely on specific technologies, such as K8s [24, 42, 74, 76, 91, 92], assume the deployment of an application to be such that each service is deployed on a different VM [62, 63], or require the application operator to provide additional artifacts, e.g., a specification of the application topology [1, 68, 74], known anomaly patterns [9], workload generators [21, 26, 51, 52, 91, 92], or failure injection modules [21]. Integration is instead required if we wish to first detect anomalies and then automatically determine their possible root causes, as the selected anomaly detection and root cause analysis techniques must be put in pipeline unless they come as an already integrated pipeline, such as in the case of 10 of the surveyed techniques [18, 19, 24, 42, 45, 51, 74, 76, 86, 87, 91, 92].

In summary, the surveyed techniques differ in their applicability to existing applications as they are. They may require to instrument a multi-service application to adapt its deployment to work with a given technique, to provide additional artifacts to configure the chosen techniques to detect anomalies and identify their possible root causes, or to be integrated to enact a pipeline of anomaly detection and root cause analysis. There is hence no solution that best fits for detecting anomalies and identifying their root causes in any target application. The choice of which techniques to use rather consists of finding a suitable tradeoff between the desired granularity/type of anomalies to be considered, the level and ranking of identified root causes, and the cost for setting up chosen technique to work with a target application.

## 6.2 Accuracy

The surveyed anomaly detection techniques (but for those based on SLO checks or heart beating) share the common assumption that, at runtime, the application and its services behave similarly to the training runs. They indeed assume that applications will feature similar performances, log similar events, or perform similar interactions in a similar order. This is, however, not always the case since the actual behaviour of multi-service applications also depends on the conditions under which they are executed, which often dynamically changes [80]. For instance, if hosted on the same virtual environments but with different sets of co-hosted applications, multi-service applications may suffer from different resource contentions and feature different performances [53]. Similar considerations derive from the workload generated by end-users, which often varies with seasonalities [52]. In addition, multi-service applications can be highly dynamic in their nature, especially if based on microservices: new services can be included to feature new functionalities, or existing services can be replaced or removed as becoming outdated [80]. As a result, all techniques assuming that multi-service applications keep running under the same conditions as in training runs may lower their accuracy if the running conditions of an application change, as they may end up with detecting false positives or with suffering from false negatives. They may indeed detect functional or performance anomalies in a running application, even if these are not anomalous changes, but rather correspond to the "new normal behaviour" in the new conditions under which an application runs. They may instead consider the actual behaviour of an application as normal, even if such behaviour may denote anomalies in the latest running conditions of an application.

The problem of false positives and false negatives is not to be underestimated. Anomaly detection is indeed enacted as anomalies are possible symptoms of failures. A detected anomaly on a service hence provides a warning on the fact that such service can have possibly failed, and it may also be used to determine the type of failure affecting such service. A false positive, viz., a detected anomaly that is actually not an anomaly, would bring the application operator to waste resources and efforts on application services that could have possibly failed, e.g., to recover them or to understand why they have failed, even if this was not the case. False negatives have a different but still significant impact: if some anomalies would not be detected, the symptoms of some failures may not get detected, which in turn means that the application operator would not be warned in case such failures actually occur in a multi-service application. Both false positives and false negatives hence impact on the practical usefulness of anomaly detection techniques, as the former may result in significant wastes of resources and efforts, whereas the latter may make an anomaly detection technique useless in some cases.

Similar considerations apply to the surveyed root cause analysis techniques, whose accuracy may also be affected by the problem of false positives and false negatives. In this case, false positives correspond to the services or KPIs that are identified as possible root causes for an observed anomaly, even if they actually have nothing in common with such anomaly. The only situation in which no false positive is returned by a root cause analysis technique is when Aggarwal et al. [1] return the service that is actually causing an observed anomaly. In all other cases, false positives are there: if Aggarwal et al. [1] return a service different from that actually causing an observed anomaly, the returned service is a false positive. The other surveyed techniques instead return multiple possible root causes, often including various false positives. Such false positives generate more effort for the application operator, who is required to troubleshoot the corresponding services, even if they actually not caused the anomaly whose root causes are being searched.

False negatives are, however, always around the corner. The common driver for automatically identifying root causes with the surveyed techniques is a correlation, which is used to correlate offline detected anomalies or to drive the search within the application topology or in a causality graph. It is, however, known that correlation is however not ensuring causation [12], hence meaning that the actual root cause for an anomaly observed in a service may not be amongst the services whose behaviour is highly correlated with that of the anomalous service, but rather in some other service. Spurious correlations impact more on those techniques determining causalities by relying on correlation only, e.g., to determine which is the root cause among co-occurring anomalies, or to build a causality graph. One may hence think that topology-driven techniques are immune to spurious correlations, as they rely on the explicit modelling of the interconnections between the services forming an application and their hosting nodes. However, those techniques enacting correlation-guided random walks over the topology of an application are still prone to spurious correlations. At the same time, even if a technique enacts an extensive search over the application topology and considers all modelled dependencies, it may miss the actual root cause in some cases. For instance, if a topology is just representing service interactions, a technique analysing such topology may miss the case of performance anomalies affecting a service because other services hosted on the same node are consuming all computing resources. It may also be the case that—independently from the applied method—a root cause technique actually determines the root cause for an observed anomaly, but the latter is excluded from those returned because its estimated likelihood to have caused the observed anomaly is below a given threshold.

Mitigation actions must hence be put in place to reduce the impact of false positives and false negatives in the enacted anomaly detection and root cause analisys. For instance, in the case of anomaly detection, the baseline trained to detect anomalies should be kept up to date by adapting it to the varying conditions under which a multi-service application runs. A subset of the surveyed techniques explicitly recognise this problem, e.g., PreMiSE [52] prescribes to use of a huge amount of training data to include as many evolutions of the application as possible, therein including seasonalities of workloads, to mitigate the effects of the “training problem” as much as possible. Some of the surveyed techniques also propose solutions to address the training problem by keeping the baseline model up to date. For instance, DLA [74], Seer [22], and Wang et al. [88], among others, propose to periodically re-train the baseline model by considering newly collected data as training data. They also observe that the re-training period should be tailored as a tradeoff between the effectiveness of the trained models and the computational effort needed to re-train the model, as re-training is typically resource and time-consuming. CloudRanger [87] instead keeps the baseline continuously updated by enacting continuous learning on the services forming an application. Among trace comparison-based techniques, Chen et al. [17] actually try to address the issue with false positives by exploiting matrix sketching to keep track of the entire prior for each service. They hence compare newly collected information with all that has been observed formerly, both during the offline reference runs and while the application was running in production.

The impact of false positives and false negatives on techniques enacting anomaly detection based on supervised learning [8, 21, 22, 52, 60, 97] is even higher: the false positives and false negatives possibly caused by the training problem impact not only on anomaly detection, but also on the identification of the failure corresponding to a detected anomaly. Failure identification is indeed based on the correlation of symptoms, assuming that the same failure happening on the same service would result in a similar set of symptoms. Again, the varying conditions under which an application runs, as well as the fact that the service forming an application may dynamically change themselves, may result in the symptoms observed on a service to change over time. The failures that can possibly affect a service may also change over time, hence requiring to adapt the supervised learning-based techniques not only to the varying conditions under which an application runs, but also to the varying sets of possible failures for a service. To tackle this issue, Seer [22] re-trains the baseline modelling of the application by also considering the failures that were observed on the application services while they were running in production. Seer is actually the only technique dealing with the training problem among those enacting supervised learning. This is mainly because Seer is designed to be periodically trained on the data monitored while an application is running in production, whereas the other supervised learning-based techniques rely on predefined failures being artificially injected during the training runs.

Almost all the surveyed root cause analysis techniques instead reduce the risk for false negatives by returning multiple possible root causes, hence increasing the probability that the actual root cause is amongst those returned. This choice is mainly motivated by the fact that root cause analysis is enacted to automatically identify the actual root causes for an observed anomaly, and missing such actual root causes may make the price to enact root cause analysis not worthy to be paid. At the same time, this comes at the price of increasing the rate of false positives, hence putting more effort on the application operator [68], who must manually discern which is the actual root cause amongst those returned, if any. As we already highlighted in Section 5.3.3, to reduce the effort to be put on the application operator to identify the actual root cause among the returned ones, some techniques also rank them based on their likelihood to have caused the observed anomaly [38].

The above, however, only constitute actions to mitigate the impact of false positives/negatives on the enacted anomaly detection and root cause analysis. False positives/negatives are indeed a price to pay when detecting anomalies by comparing the runtime behaviour of an application with a baseline modelling of its normal behaviour, as well as when identifying possible root causes for an observed anomaly based on correlation or by abstracting from some relationships among the services in an application (e.g., co-hosting). A quantitative comparison of the accuracy of the surveyed techniques on given applications in given contexts would further help application operators in this direction, and it will complement our survey in supporting applications in choosing the anomaly detection and root cause analysis techniques best suited to their needs. Such a quantitative comparison is, however, outside of the scope of this survey and a direction for future work.

## 6.3 Explainability and Countermeasures

The problem of false positives/negatives is also motivated by a currently open challenge in the field of anomaly detection and root cause analysis for multi-service applications, viz., "explainability". As for anomaly detection, this is tightly coupled with the problem of explainable AI [25], given that anomaly detection is mainly enacted with machine learning-based techniques. If giving explanations for answers given by learned models is generally important, it is even more important in online anomaly detection. Explanations would indeed allow application operators to directly exclude false positives, or to exploit such explanations, e.g., to investigate the root causes for the corresponding failures, as well as to design countermeasures avoiding such failures to happen again [11].

Root cause analysis can anyhow be enacted with the techniques that we surveyed in Section 5, which, however, still suffer from the training problem potentially causing false positives or false negatives, as well as from the need for explainability. Identified root causes should be associated with explanations on why they may have caused an observed anomaly, or why they are ranked higher than other possible root causes, in the case of techniques returning a ranking of possible root causes. Such information would indeed help application operators in identifying the identified root causes that are false positives, hence needing to get excluded, or in considering the possible root causes in some order different from that given by the ranking, if they believe that their associated explanations are more likely to have caused an observed anomaly. Explanations would also help reducing false negatives, which are typically due to the fact that a root cause analysis technique excluded possible root causes if their likelihood to have caused an observed anomaly was below a given threshold. If returned root causes would be associated with explanations on why they were considered such, we may avoid cutting the returned set of possible root causes, hence reducing the risk of cutting the true causes off from those returned. Again, application operators could exploit provided explanations to directly exclude the root causes corresponding to false positives.

Explanations would also help devising countermeasures to avoid certain anomalies to happen again, e.g., avoiding an anomaly to occur again because of the same root cause. More generally, an interesting research direction would be to take advantage of the information processed to detect anomalies and identify their root causes to determine countermeasures to isolate (the failures causing) detected anomalies, e.g., by proactively activating circuit breakers/bulkheads [71] to avoid that a service anomaly propagates to other services. A first attempt in this direction is made by Seer [22], which notifies the application operator about the anomaly detected on a service and the failure that can have possibly caused it, e.g., CPU hog or memory leak. With such information, Seer envisions the application operator to be able to enact policies mitigating the effects of the failure causing a detected anomaly, e.g., by providing the node hosting an anomalous service with additional computing resources. Seer, however, does so by only relying on the information monitored on the service where the anomaly is detected, hence not considering the possibility of, e.g., failure cascades propagating up to such service and causing its observed anomaly.

The surveyed root cause analysis techniques allow to determine possible root causes for an observed anomaly, which may be the set of services that may have caused the anomaly, or the trace events or KPIs collected on such services. The identified root causes are returned to the application operator, who is in charge of further troubleshooting the potentially culprit services to determine whether/why they actually caused the observed anomaly. At the same time, the possible root causes are identified based on data collected on the services forming an application, which is processed only to enact root cause analysis. Such data could, however, be further processed to further support the application operator, e.g., to automatically extract possible countermeasures to avoid it to cause the observed anomaly [11]. For instance, if a service logged an error event that caused a cascade of errors reaching the service where the anomaly was observed, the root cause analysis technique could further process the error logs to suggest how to avoid such error propagation to happen again, e.g., by introducing circuit breakers or bulkheads [71]. If a service was responding too slowly and caused some performance anomaly in another service, this information can be used to train machine learning models to predict similar performance degradations in the root causing service, which could then be used to preemptively scale such service and avoid the performance degradation to again propagate in future runs of an application. The above are just two out of many possible examples, deserving further investigation and opening new research directions.

## 7 CONCLUSIONS

We surveyed existing techniques for anomaly detection and root cause analysis in modern multi-service applications. Other than presenting the methods applied by such techniques, we discussed the type and granularity of anomalies they can detect/analyse, their setup costs (viz., the instrumentation that application operators must apply to their applications to use such techniques, and the additional artifacts they must produce), their accuracy and applicability to dynamically changing applications, and the need for explainability and countermeasures.

We believe that our survey can provide benefits to practitioners working with modern multi-service applications. Our survey provides a first support for setting up a pipeline of anomaly detection and root cause analysis techniques, so that failures are automatically detected and analysed in multi-service applications. We indeed not only highlighted which existing techniques already come in an integrated pipeline, but we also discussed the type and granularity of anomalies they consider. This provides a baseline for application operators to identify the anomaly detection and root cause analysis techniques that—even if coming as independent solutions—could be integrated into a pipeline since they identify and explain the same type/granularity of anomalies. This, together with our discussion on the techniques' setup costs (Section 6.1), provides a first support to practitioners for choosing the most suited techniques for their needs.

Our survey can also help researchers investigating failures and their handling in modern multi-service applications, as it provides a first structured discussion of the existing research contributions for enacting anomaly detection and root cause analysis in such a kind of applications. In addition, and based on the discussion we provided in Section 6, we hereafter outline three main research challenges that deserve further investigation:

**Continuous delivery—** False positives and false negatives can negatively affect the accuracy of observed anomalies/identified root causes. They indeed both impact on the usefulness of the outcomes of an anomaly detection or root cause analysis technique: false positives result in unnecessary work for application operators, who are asked to troubleshoot services even if they are not anomalous; false negatives instead correspond to considering a service as not anomalous or not causing an observed anomaly, even if this was actually the case. In both cases, false positives/negatives can occur if the services forming an application or its runtime conditions change over time. Whilst some of the surveyed techniques already consider the issue of accuracy losses in the case of application changing over time, this is done with time-consuming processes, e.g., by re-training the machine learning models used to detect anomalies [22, 74, 88] or by relying on application operators to provide updated artifacts to identify possible root causes [9]. Re-training or manual updates, as well as the other proposed updates, are hence not suited to get continuously enacted, but rather require to find a suitable tradeoff between the update period and the accuracy of enacted anomaly detection/root cause analysis [22, 74]. However, modern software delivery practices are such that new versions of the services forming an application are continuously released [29], and cloud application deployments are such that services are often migrated from one runtime to another [15]. An interesting research direction is hence to devise anomaly detection and root cause analysis techniques that can effectively work also in presence of continuous changes in multi-service applications, either adapting the surveyed ones or by devising new ones, e.g., by exploiting the recently proposed continual learning solutions [46].

**Explainability—** Most of the surveyed techniques are not "explainable by design", meaning that, despite they effectively detect anomalies and their root causes, they are not providing explanations of why this is the case. This is typical because machine learning or correlation analyses are used to determine whether a service is anomalous or the possible root cause of a detected anomaly. At the same time, associating observed anomalies and their identified root causes with explanations of why they are considered so would allow application operators to directly exclude false positives, which in turn means that, e.g., we could avoid cutting the possible root causes for an observed anomaly to only the most probable ones. Indeed, an application operator could be provided with all possible root causes and she could later decide which deserve to be troubleshooted based on their explanations. This hence calls for anomaly detection and root cause analysis techniques that are "explainable by design", much in the same way as the need for explainability is nowadays recognized in AI [25].

**Countermeasures—** Recommending potential countermeasures to be enacted to avoid (the failures corresponding) to observed anomalies and their root causes would enable avoiding such failures to happen again in the future. This could be done both for anomaly detection and for root cause analysis. Indeed, whilst avoiding (the failure corresponding to a) detected anomaly of course needs acting on the root causes for such an anomaly, countermeasures could anyhow be taken to avoid the anomaly in a service to propagate to other services. At the same time, the possible root causes for an observed anomaly are identified based on data collected on the services forming an application, which is processed only to enact root cause analysis. However, such data could be further processed to automatically extract possible countermeasures to avoid it to cause the observed anomaly. For instance, if a service logged an error event that caused a cascade of errors reaching the service where the anomaly was observed, the root cause analysis technique could further process the error logs to suggest how to avoid such error propagation to happen again, e.g., by introducing circuit breakers or bulkheads [71]. If a service was responding too slowly and caused some performance anomaly in another service, this information can be used to train machine learning models to predict similar performance degradations in the root causing service, which could then be used to preemptively scale such service and avoid the performance degradation to again propagate in future runs of an application.

Our survey provides a qualitative on the state-of-the-art for detecting anomalies in multi-service applications and for determining their possible root causes, based on the peer-reviewed literature published on the topic. A quantitative comparison of the performance of the surveyed techniques would further support practitioners, and it definitely deserves future work. Indeed, whilst the surveyed techniques already measure their performance or accuracy, sometimes also in comparison with other existing techniques, this is typically done with experiments on different reference applications running in different environments. This makes it complex to quantitatively compare the existing anomaly detection and root cause analysis techniques, hence opening the need for studies quantitatively comparing the performance of such techniques by fixing the reference applications and runtime environments, much in a similar way as Arya et al. [5] did for a subset of existing root cause analysis techniques. Quantitative performance comparisons would complement the results of our qualitative comparison, providing practitioners with more tools to choose the anomaly detection and root cause analysis techniques most suited to their applications' requirements.

Another interesting direction for future work is understanding the gap between the state-of-the-art overviewed and analysed in this survey and the state-of-practice on the topic. There indeed exist industrial solutions for detecting anomalies and identifying their possible root causes, coming both as proprietary solutions and open-source software. A first step towards bridging such a gap requires analysing and classifying industrial solutions for anomaly detection and root cause analysis, much in a similar way as we did for FaaS platforms in our recent study [93]. This would indeed enable understanding which techniques are enacted in available industrial solutions, as well as whether they actually enact or can be integrated with those available in the state-of-the-art surveyed in this article. This information would be precious for researchers and practitioners working on anomaly detection and root cause analysis in modern multi-service applications, hence deserving further investigation, which we plan to pursue in our future work.

## REFERENCES

[1] P. Aggarwal, A. Gupta, P. Mohapatra, S. Nagar, A. Mandal, Q. Wang, and A. Paradkar. 2020. Localization of operational faults in cloud applications by mining causal dependencies in logs using golden signals. In *Proceedings of the Service-Oriented Computing (LNCS, Vol. 12632)*. Springer, Cham, 137–149. DOI: https://doi.org/10.1007/978-3-030-76352-7_17

[2] L. Akoglu, H. Tong, and D. Koutra. 2015. Graph based anomaly detection and description: A survey. *Data Mining and Knowledge Discovery* 29, 3 (2015), 626–688. DOI: https://doi.org/10.1007/s10618-014-0365-y

[3] N. S. Altman. 1992. An introduction to kernel and nearest-neighbor nonparametric regression. *The American Statistician* 46, 3 (1992), 175–185. DOI: https://doi.org/10.1080/00031305.1992.10475879

[4] A. Arnold, Y. Liu, and N. Abe. 2007. Temporal causal modeling with graphical granger methods. In *Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining 2007*. ACM, New York, 66–75. DOI: https://doi.org/10.1145/1281192.1281203

[5] V. Arya, K. Shanmugam, Pooja Aggarwal, Qing Wang, Prateeti Mohapatra, and Seema Nagar. 2021. Evaluation of causal inference techniques for AIOps. In *Proceedings of the CODS COMAD 2021*. ACM, New York, 188–192. DOI: https://doi.org/10.1145/3430984.3431027

[6] M. Basseville and I. V. Nikiforov. 1993. *Detection of Abrupt Changes: Theory and Application*. Prentice-Hall, Inc.

[7] I. Beschastnikh, P. Wang, et al. 2016. Debugging distributed systems. *Communications of the ACM* 59, 8 (July 2016), 32–37. DOI: https://doi.org/10.1145/2909480

[8] J. Bogatinovski, S. Nedelkoski, J. Cardoso, and O. Kao. 2020. Self-supervised anomaly detection from distributed traces. In *Proceedings of the 2020 IEEE/ACM 13th International Conference on Utility and Cloud Computing*. IEEE, New York, 342–347. DOI: https://doi.org/10.1109/UCC48980.2020.00054

[9] A. Brandón, M. Solé, Alberto Huélamo, David Solans, María S. Pérez, and Victor Muntés-Mulero. 2020. Graph-based root cause analysis for service-oriented and microservice architectures. *Journal of Systems and Software* 159 (2020), 110432. DOI: https://doi.org/10.1016/j.jss.2019.110432

[10] L. Breiman. 2001. Random forests. *Machine Learning* 45, 1 (2001), 5–32. DOI: https://doi.org/10.1023/A:1010933404324

[11] A. Brogi and J. Soldani. 2020. Identifying failure causalities in multi-component applications. In *Proceedings of the Software Engineering and Formal Methods (LNCS, Vol. 12226)*. Springer, Cham, 226–235. DOI: https://doi.org/10.1007/978-3-030-57506-9_17

[12] C. Calude and G. Longo. 2017. The deluge of spurious correlations in big data. *Foundations of Science* 22, 3 (2017), 595–612. DOI: https://doi.org/10.1007/s10699-016-9489-4

[13] E. J. Candès, X. Li, et al. 2009. Robust principal component analysis? *Journal of the ACM* 58, 3, Article 11 (2011), 37 pages. DOI: https://doi.org/10.1145/1970392.1970395

[14] O. Capp, E. Moulines, and T. Ryden. 2010. *Inference in Hidden Markov Models*. Springer, New York.

[15] J. Carrasco, F. Durán, and E. Pimentel. 2020. Live migration of trans-cloud applications. *Computer Standards & Interfaces* 69 (2020), 103392. DOI: https://doi.org/10.1016/j.csi.2019.103392

[16] V. Chandola, A. Banerjee, and V. Kumar. 2009. Anomaly detection: A survey. *ACM Computing Surveys* 41, 3, Article 15 (2009), 58 pages. DOI: https://doi.org/10.1145/1541880.1541882

[17] H. Chen, P. Chen, and G. Yu. 2020. A framework of virtual war room and matrix sketch-based streaming anomaly detection for microservice systems. *IEEE Access* 8 (2020), 43413–43426. DOI: https://doi.org/10.1109/ACCESS.2020.2977464

[18] P. Chen, Y. Qi, Pengfei Zheng, and Di Hou. 2014. Causeinfer: Automatic and distributed performance diagnosis with hierarchical causality graph in large distributed systems. In *Proceedings of the INFOCOM 2014 IEEE Conference on Computer Communications*. IEEE, New York, 1887–1895. DOI: https://doi.org/10.1109/INFOCOM.2014.6848128

[19] P. Chen, Y. Qi, and D. Hou. 2019. Causeinfer: Automated end-to-end performance diagnosis with hierarchical causality graph in cloud environment. *IEEE Transactions on Services Computing* 12, 2 (2019), 214–230. DOI: https://doi.org/10.1109/TSC.2016.2607739

[20] M. Du, F. Li, et al. 2017. Deeplog: Anomaly detection and diagnosis from system logs through deep learning. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, New York, 1285–1298.

[21] Q. Du, T. Xie, and Y. He. 2018. Anomaly detection and diagnosis for container-based microservices with performance monitoring. In *Proceedings of the Algorithms and Architectures for Parallel Processing (LNCS, Vol. 11337)*. Springer, Cham, 560–572. DOI: https://doi.org/10.1007/978-3-030-05063-4_42

[22] Y. Gan, Y. Zhang, Kelvin Hu, Dailun Cheng, Yuan He, Meghna Pancholi, and Christina Delimitrou. 2019. Seer: Leveraging big data to navigate the complexity of performance debugging in cloud microservices. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, New York, 19–33. DOI: https://doi.org/10.1145/3297858.3304004

[23] I. Goodfellow, Y. Bengio, and A. Courville. 2016. *Deep Learning (1st ed.)*. MIT Press, Cambridge.

[24] Z. Guan, J. Lin, and P. Chen. 2019. On anomaly detection and root cause analysis of microservice systems. In *Proceedings of the Service-Oriented Computing (LNCS, Vol. 11434)*. Springer, Cham, 465–469. DOI: https://doi.org/10.1007/978-3-030-17642-6_45

[25] R. Guidotti, A. Monreale, Salvatore Ruggieri, Franco Turini, Fosca Giannotti, and Dino Pedreschi. 2018. A survey of methods for explaining black box models. *ACM Computing Surveys* 51, 5, Article 93 (2018), 42 pages. DOI: https://doi.org/10.1145/3236009

[26] A. Gulenko, F. Schmidt, Alexander Acker, Marcel Wallschläger, Odej Kao, and Feng Liu. 2018. Detecting anomalous behavior of black-box services modeled with distance-based online clustering. In *2018 IEEE 11th International Conference on Cloud Computing*. IEEE, New York, 912–915. DOI: https://doi.org/10.1109/CLOUD.2018.00134

[27] X. Guo, X. Peng, Hanzhang Wang, Wanxue Li, Huai Jiang, Dan Ding, Tao Xie, and Liangfei Su. 2020. Graph-based trace analysis for microservice architecture understanding and problem diagnosis. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, New York, 1387–1397. DOI: https://doi.org/10.1145/3368089.3417066

[28] H. Huang and S. P. Kasiviswanathan. 2015. Streaming anomaly detection using randomized matrix sketching. *VLDB Endowment* 9, 3 (2015), 192–203. DOI: https://doi.org/10.14778/2850583.2850593

[29] J. Humble and D. Farley. 2010. *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation*. Addison-Wesley, Boston.

[30] IBM. 2021. IBM operations analytics predictive insights. Retrieved November 18th, 2021 from https://www.ibm.com/support/knowledgecenter/SSJQQ3.

[31] Istio Authors. 2020. Istio. Retrieved November 18th, 2021 from https://istio.io.

[32] G. Jeh and J. Widom. 2003. Scaling personalized web search. In *Proceedings of the 12th International Conference on World Wide Web*. ACM, New York, 271–279. DOI: https://doi.org/10.1145/775152.775191

[33] T. Jia, P. Chen, Lin Yang, Ying Li, Fanjing Meng, and Jingmin Xu. 2017. An approach for anomaly diagnosis based on hybrid graph model with logs for distributed services. In *2017 IEEE International Conference on Web Services*. IEEE, New York, 25–32. DOI: https://doi.org/10.1109/ICWS.2017.12

[34] T. Jia, L. Yang, Pengfei Chen, Ying Li, Fanjing Meng, and Jingmin Xu. 2017. Logsed: Anomaly diagnosis through mining time-weighted control flow graph in logs. In *Proceedings of the IEEE 10th International Conference on Cloud Computing*. IEEE, New York, 447–455. DOI: https://doi.org/10.1109/CLOUD.2017.64

[35] M. Jin, A. Lu, Yuanpeng Zhu, Zijiang Wen, Yubin Zhong, Zexin Zhao, Jiang Wu, Hejie Li, Hanheng He, and Fengyi Chen. 2020. An anomaly detection algorithm for microservice architecture based on robust principal component analysis. *IEEE Access* 8 (2020), 226397–226408. DOI: https://doi.org/10.1109/ACCESS.2020.3044610

[36] G. H. John and P. Langley. 1995. Estimating continuous distributions in bayesian classifiers. In *Proceedings of the 11th Conference on Uncertainty in Artificial Intelligence*. Morgan Kaufmann Publishers Inc., San Francisco, 338–345.

[37] I. Jolliffe. 2011. Principal component analysis. In *International Encyclopedia of Statistical Science*. Lovric M. (Ed.), Springer, Berlin, 1094–1096. DOI: https://doi.org/10.1007/978-3-642-04898-2_455

[38] M. Kim, R. Sumbaly, and S. Shah. 2013. Root cause detection in a service-oriented architecture. *ACM SIGMETRICS Performance Evaluation Review* 41, 1 (2013), 93–104. DOI: https://doi.org/10.1145/2494232.2465753

[39] D. P. Kingma and J. Bam. 2015. Adam: A method for stochastic optimization. In *Proceedings of the 3rd International Conference for Learning Representations*. arXiv, Cornell University, Ithaca, 1–15. Retrieved from http://arxiv.org/abs/1412.6980.

[40] N. Kratzke and P. C. Quint. 2017. Understanding cloud-native applications after 10 years of cloud computing—a systematic mapping study. *Journal of Systems and Software* 126 (2017), 1–16. DOI: https://doi.org/10.1016/j.jss.2017.01.001

[41] J. Lewis and M. Fowler. 2014. Microservices. ThoughtWorks. Retrieved from https://martinfowler.com/articles/microservices.html.

[42] J. Lin, P. Chen, and Z. Zheng. 2018. Microscope: Pinpoint performance issues with causal graphs in micro-service environments. In *Proceedings of the Service-Oriented Computing (LNCS, Vol. 11236)*. Springer, Cham, 3–20. DOI: https://doi.org/10.1007/978-3-030-03596-9_1

[43] W. Lin, M. Ma, Disheng Pan, and Ping Wang. 2018. Facgraph: Frequent anomaly correlation graph mining for root cause diagnose in micro-service architecture. In *Proceedings of the 2018 IEEE 37th International Performance Computing and Communications Conference*. IEEE, New York, 1–8. DOI: https://doi.org/10.1109/PCCC.2018.8711092

[44] D. Liu, C. He, Xin Peng, Fan Lin, Chenxi Zhang, Shengfang Gong, Ziang Li, Jiayu Ou, and Zheshun Wu. 2021. MicroHECL: High-efficient root cause localization in large-scale microservice systems. In *Proceedings of the 2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice ICSE-SEIP*. IEEE, New York, 338–347. DOI: https://doi.org/10.1109/ICSE-SEIP52600.2021.00043

[45] P. Liu, H. Xu, Q. Ouyang, R. Jiao, Z. Chen, S. Zhang, J. Yang, L. Mo, J. Zeng, W. Xue, and D. Pei. 2020. Unsupervised detection of microservice trace anomalies through service-level deep bayesian networks. In *Proceedings of the 2020*

*IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, New York, 48–58. DOI:https://doi.org/10.1109/ISSRE5003.2020.00014

[46] V. Lomonaco, L. Pellegrini, A. Cossu, A. Carta, G. Graffieti, T. L. Hayes, M. De Lange, M. Masana, J. Pomponi, G. van de Ven, and M. Mundt. 2021. Avalanche: An end-to-end library for continual learning. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*. IEEE, New York, 3595–3605. DOI:https://doi.org/10.1109/CVPRW53098.2021.00399

[47] C. Luo, J. G. Lou, Qingwei Lin, Qiang Fu, Rui Ding, Dongmei Zhang, and Zhe Wang. 2014. Correlating events with time series for incident diagnosis. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, New York, 1583–1592. DOI:https://doi.org/10.1145/2623330.2623374

[48] M. Ma, W. Lin, Disheng Pan, and Ping Wang. 2019. MS-rank: Multi-metric and self-adaptive root cause diagnosis for microservice applications. In *Proceedings of the 2019 IEEE International Conference on Web Services*. IEEE, New York, 60–67. DOI:https://doi.org/10.1109/ICWS.2019.00022

[49] M. Ma, W. Lin, Disheng Pan, and Ping Wang. 2020. Self-adaptive root cause diagnosis for large-scale microservice architecture. *IEEE IEEE Transactions on Services Computing*. DOI:https://doi.org/10.1109/TSC.2020.2993251

[50] M. Ma, J. Xu, Yuan Wang, Pengfei Chen, Zonghua Zhang, and Ping Wang. 2020. AutoMAP: Diagnose your microservice-based web applications automatically. In *Proceedings of the Web Conference*. ACM, New York, 246–258. DOI:https://doi.org/10.1145/3366423.3380111

[51] L. Mariani, C. Monni, Mauro Pezzé, Oliviero Riganelli, and Rui Xin. 2018. Localizing faults in cloud systems. In *Proceedings of the 2018 IEEE 11th International Conference on Software Testing, Verification and Validation*. IEEE, New York, 262–273. DOI:https://doi.org/10.1109/ICST.2018.00034

[52] L. Mariani, M. Pezzé, Oliviero Riganelli, and Rui Xin. 2020. Predicting failures in multi-tier distributed systems. *Journal of Systems and Software* 161 (2020), 110464. DOI:https://doi.org/10.1016/j.jss.2019.110464

[53] V. Medel, R. Tolosana-Calasanz, José Ángel Bañares, Unai Arronategui, and Omer F. Rana. 2018. Characterising resource management performance in kubernetes. *Computers & Electrical Engineering* 68 (2018), 286–297. DOI:https://doi.org/10.1016/j.compeleceng.2018.03.041

[54] L. Meng, F. Ji, Yao Sun, and Tao Wang. 2021. Detecting anomalies in microservices with execution trace comparison. *Future Generation Computer Systems* 116 (2021), 291–301. DOI:https://doi.org/10.1016/j.future.2020.10.040

[55] Y. Meng, S. Zhang, Yongqian Sun, Ruru Zhang, Zhilong Hu, Yiyin Zhang, Chenyang Jia, Zhaogang Wang, and Dan Pei. 2020. Localizing failure root causes in a microservice through causality inference. In *Proceedings of the 2020 IEEE/ACM 28th International Symposium on Quality of Service*. IEEE, New York, 1–10. DOI:https://doi.org/10.1109/IWQoS49365.2020.9213058

[56] H. Mi, H. Wang, Yangfan Zhou, Michael Rung-Tsong Lyu, and Hua Cai. 2013. Toward fine-grained, unsupervised, scalable performance diagnosis for production cloud computing systems. *IEEE Transactions on Parallel and Distributed Systems* 24, 6 (2013), 1245–1255. DOI:https://doi.org/10.1109/TPDS.2013.21

[57] T. M. Mitchell. 1997. *Machine Learning (1st ed.)*. McGraw-Hill, Inc.

[58] A. Nandi, A. Mandal, Atreja, S., G. B. Dasgupta, and S. Bhattacharya. 2016. 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*. ACM, New York, 215–224. DOI: https://doi.org/10.1145/2939672.2939712

[59] R. M. Neal. 1996. *Bayesian Learning for Neural Networks (1st ed.)*. Springer-Verlag, New York.

[60] S. Nedelkoski, J. Cardoso, and O. Kao. 2019. Anomaly detection and classification using distributed tracing and deep learning. In *Proceedings of the 2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. IEEE, New York, 241–250. DOI:https://doi.org/10.1109/CCGRID.2019.00038

[61] S. Nedelkoski, J. Cardoso, and O. Kao. 2019. Anomaly detection from system tracing data using multimodal deep learning. In *Proceedings of the 2019 IEEE 12th International Conference on Cloud Computing*. IEEE, New York, 179–186. DOI:https://doi.org/10.1109/CLOUD.2019.00038

[62] H. Nguyen, Z. Shen, Yongmin Tan, and Xiaohui Gu. 2013. FChain: Toward black-box online fault localization for cloud systems. In *Proceedings of the 2013 IEEE 33rd International Conference on Distributed Computing Systems*. IEEE, New York, 21–30. DOI:https://doi.org/10.1109/ICDCS.2013.26

[63] H. Nguyen, Y. Tan, and X. Gu. 2011. PAL: Propagation-aware anomaly localization for cloud hosted distributed applications. In *Proceedings of the Managing Large-scale Systems Via the Analysis of System Logs and the Application of Machine Learning Techniques*. ACM, New York, Article 1, 8 pages. DOI:https://doi.org/10.1145/2038633.2038634

[64] H. Peng, F. Long, and C. Ding. 2005. Feature selection based on mutual information: Criteria of max-dependency, max-relevance, and min-redundancy. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 27, 8 (Aug. 2005), 1226–1238. DOI: https://doi.org/10.1109/TPAMI.2005.159

[65] T. Pitakrat, D. Okanović, Andre Van Hoorn, and Lars Grunske. 2016. An architecture-aware approach to hierarchical online failure prediction. In *Proceedings of the 2016 12th International ACM SIGSOFT Conference on Quality of Software Architectures*. IEEE, New York, 60–69. DOI:https://doi.org/10.1109/QoSA.2016.16

[66] T. Pitakrat, D. Okanović, André van Hoorn, and Lars Grunske. 2018. Hora: Architecture-aware online failure prediction. *Journal of Systems and Software* 137 (2018), 669–685. https://doi.org/10.1016/j.jss.2017.02.041

[67] Prometheus Authors. 2021. Prometheus: Monitoring System & Time Series Database. Retrieved November 18th, 2021 from https://prometheus.io.

[68] J. Qiu, Q. Du, K. Yin, S. L. Zhang, and C. Qian. 2020. A causality mining and knowledge graph based method of root cause diagnosis for performance anomaly in cloud applications. *Applied Sciences* 10, 6, Article 2166 (2020), 19 pages. DOI: https://doi.org/10.3390/app10062166

[69] D. Reis, P. B. Golgher, A. S. Silva, and A. Laender. 2004. Automatic web news extraction using tree edit distance. In *Proceedings of the 13th International Conference on World Wide Web*. ACM, New York, 502–511. DOI: https://doi.org/10.1145/988672.988740

[70] D. J. Rezende and S. Mohamed. 2015. Variational inference with normalizing flows. In *Proceedings of the 2017 International Conference on Machine Learning*, Vol. 37. JMLR.org, Microtome Publishing, Brookline, 1530–1538.

[71] C. Richardson. 2018. *Microservices Patterns (1st ed.)*. Manning Publications, Shelter Island.

[72] N. J. Salking. 2010. Coefficient of variation. In *Proceedings of the Encyclopedia of Research Design (SAGE Research Methods)*. SAGE Publications, Washington, DC, 169–171. DOI: https://doi.org/10.4135/9781412961288.n56

[73] J. H. Saltzer and M. F. Kaashoek. 2009. *Principles of Computer System Design: An Introduction*. Morgan Kaufmann Publishers Inc., San Francisco.

[74] A. Samir and C. Pahl. 2019. DLA: Detecting and localizing anomalies in containerized microservice architectures using markov models. In *Proceedings of the 2019 7th International Conference on Future Internet of Things and Cloud*. IEEE, New York, USA, 205–213. DOI: https://doi.org/10.1109/FiCloud.2019.00036

[75] B. Schölkopf, C. J. C. Burges, and A. J. Smola. 1999. *Advances in Kernel Methods: Support Vector Learning*. MIT Press, Cambridge.

[76] H. Shan, Y. Chen, Haifeng Liu, Yunpeng Zhang, Xiao Xiao, Xiaofeng He, Min Li, and Wei Ding. 2019. $\epsilon$-Diagnosis: Unsupervised and real-time diagnosis of small-window long-tail latency in large-scale microservice platforms. In *Proceedings of the World Wide Web Conference*. ACM, New York, 3215–3222. DOI: https://doi.org/10.1145/3308558.3313653

[77] D. J. Sheskin. 2011. *Handbook of Parametric and Nonparametric Statistical Procedures (5th ed.)*. Chapman & Hall/CRC, Taylor & Francis Group, Abingdon, UK.

[78] R. H. Shumway and D. S. Stoffer. 2017. *Time Series Analysis and Its Applications (4th ed.)*. Springer, New York.

[79] A. Siffer, P. A. Fouque, et al. 2017. Anomaly detection in streams with extreme value theory. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, New York, 1067–1075. DOI: https://doi.org/10.1145/3097983.3098144

[80] J. Soldani, D. A. Tamburri, and W. J. Van Den Heuvel. 2018. The pains and gains of microservices: A systematic grey literature review. *Journal of Systems and Software* 146 (2018), 215–232. DOI: https://doi.org/10.1016/j.jss.2018.09.082

[81] M. Solé, Victor Muntés-Mulero, Annie Ibrahim Rana, and Giovani Estrada 2017. Survey on models and techniques for root-cause analysis. arXiv:1701.08546. Retrieved from https://arxiv.org/abs/1701.08546.

[82] P. Spirtes, C. Glymour, and R. Scheines. 2000. *Causation, Prediction, and Search (2nd ed.)*. MIT Press, Cambridge.

[83] M. Steinder and A. S. Sethi. 2004. A survey of fault localization techniques in computer networks. *Science of Computer Programming* 53, 2 (2004), 165–194. DOI: https://doi.org/10.1016/j.scico.2004.01.010

[84] J. Thalheim, A. Rodrigues, Istemi Ekin Akkus, Pramod Bhatotia, Ruichuan Chen, Bimal Viswanath, Lei Jiao, and Christof Fetzer. 2017. Sieve: Actionable insights from monitored metrics in distributed systems. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*. ACM, New York, 14–27. DOI: https://doi.org/10.1145/3135974.3135977

[85] A. van Hoorn. 2014. *Model-Driven Online Capacity Management for Component-Based Software Systems*. Ph.D. Dissertation. Faculty of Engineering, Kiel University, Kiel, Germany.

[86] L. Wang, N. Zhao, Junjie Chen, Pinnong Li, Wenchi Zhang, and Kaixin Sui. 2020. Root-cause metric location for microservice systems via log anomaly detection. In *Proceedings of the 2020 IEEE International Conference on Web Services*. IEEE, New York, 142–150. DOI: https://doi.org/10.1109/ICWS49710.2020.00026

[87] P. Wang, J. Xu, Meng Ma, Weilan Lin, Disheng Pan, Yuan Wang, and Pengfei Chen. 2018. Cloudranger: Root cause identification for cloud native systems. In *Proceedings of the 2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. IEEE, New York, 492–502. DOI: https://doi.org/10.1109/CCGRID.2018.00076

[88] T. Wang, W. Zhang, Jiwei Xu, and Zeyu Gu. 2020. Workflow-aware automatic fault diagnosis for microservice-based applications with statistics. *IEEE Transactions on Network and Service Management* 17, 4 (2020), 2350–2363. DOI: https://doi.org/10.1109/TNSM.2020.3022028

[89] Weaveworks and Container Solutions. 2017. Sock shop. Retrieved November 18th, 2021 from https://microservices-demo.github.io.

[90] E. W. Wong, R. Gao, Yihao Li, Rui Abreu, and Franz Wotawa. 2016. A survey on software fault localization. *IEEE Transactions on Software Engineering* 42, 8 (2016), 707–740. DOI: https://doi.org/10.1109/TSE.2016.2521368

[91] L. Wu, J. Bogatinovski, Sasho Nedelkoski, Johan Tordsson, and Odej Kao. 2020. Performance diagnosis in cloud microservices using deep learning. In *Proceedings of the International Conference on Service-Oriented Computing (LNCS, Vol. 12632)*. Springer, Cham, 85–96. DOI: https://doi.org/10.1007/978-3-030-76352-7_13

[92] L. Wu, J. Tordsson, Erik Elmroth, and Odej Kao. 2020. MicroRCA: Root cause localization of performance issues in microservices. In *NOMS 2020 IEEE/IFIP Network Operations and Management Symposium*. IEEE, New York, 1–9. DOI: https://doi.org/10.1109/NOMS47738.2020.9110353

[93] V. Yussupov, J. Soldani, Uwe Breitenbücher, Antonio Brogi, and Frank Leymann. 2021. Faasten your decisions: A classification framework and technology review of function-as-a-service platforms. *Journal of Systems and Software* 175 (2021), 110906. DOI: https://doi.org/10.1016/j.jss.2021.110906

[94] X. Zang, W. Chen, Jing Zou, Sheng Zhou, Huang Lisong, and Liang Ruigang. 2018. A fault diagnosis method for microservices based on multi-factor self-adaptive heartbeat detection algorithm. In *Proceedings of the 2018 2nd IEEE Conference on Energy Internet and Energy System Integration*. IEEE, New York, 1–6. DOI: https://doi.org/10.1109/EI2.2018.8582217

[95] T. Zhang, R. Ramakrishnan, and M. Livny. 1996. BIRCH: An efficient data clustering method for very large databases. *ACM Sigmod Record* 25 2(1996), 103–114. ACM, New York. DOI: https://doi.org/10.1145/233269.233324

[96] X. Zhou, X. Peng, Tao Xie, Jun Sun, Chao Ji, Wenhai Li, and Dan Ding. 2019. Delta debugging microservice systems with parallel optimization. *IEEE Transactions on Services Computing*. DOI: https://doi.org/10.1109/TSC.2019.2919823

[97] X. Zhou, X. Peng, Tao Xie, Jun Sun, Chao Ji, Dewei Liu, Qilin Xiang, and Chuan He. 2019. Latent error prediction and fault localization for microservice applications by learning from system trace logs. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, New York, 683–694. DOI: https://doi.org/10.1145/3338906.3338961

[98] X. Zhou, X. Peng, Tao Xie, Jun Sun, Chao Ji, Wenhai Li, and Dan Ding. 2021. Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study. *IEEE Transactions on Software Engineering* 47, 2 (2021), 243–260. DOI: https://doi.org/10.1109/TSE.2018.2887384

Received May 2021; revised October 2021; accepted November 2021