Journal of Grid Computing (2021) 19: 9
https://doi.org/10.1007/s10723-021-09551-5

Automated Analysis of Distributed Tracing: Challenges and Research Directions

Andre Bento · Jaime Correia · Ricardo Filipe · Filipe Araujo · Jorge Cardoso

Received: 10 July 2020 / Accepted: 6 February 2021 / Published online: 25 February 2021

Abstract Microservice-based architectures are gaining popularity for their benefits in software development. Distributed tracing can be used to help operators maintain observability in this highly distributed context, and find problems such as latency, and analyse their context and root cause. However, exploring and working with distributed tracing data is sometimes difficult due to its complexity and application specificity, volume of information and lack of tools. The most common and general tools available for this kind of data, focus on trace-level human-readable data visualisation. Unfortunately, these tools do not provide good ways to abstract, navigate, filter and analyse tracing data. Additionally, they do not automate or aid with trace analysis, relying on administrators to do it themselves. In this paper we propose using tracing data to extract service metrics, dependency graphs and work-flows with the objective of detecting anomalous services and operation patterns. We implemented and published open source prototype tools to process tracing data, conforming to the OpenTracing standard, and developed anomaly detection methods. We validated our tools and methods against real data provided by a major cloud provider. Results show that there is an underused wealth of actionable information that can be extracted from both metric and morphological aspects derived from tracing. In particular, our tools were able to detect anomalous behaviour and situate it both in terms of involved services, work-flows and time-frame. Furthermore, we identified some limitations of the OpenTracing format—as well as the industry accepted tracing abstractions—, and provide suggestions to test trace quality and enhance the standard.

Keywords Microservices · Autonomic analysis · Anomaly detection · Observability · Monitoring · Tracing

A. Bento (✉) · J. Correia · R. Filipe · F. Araujo · J. Cardoso
CISUC, Department of Informatics Engineering, University of Coimbra, Coimbra, Portugal
e-mail: apbento@dei.uc.pt

J. Cardoso
Huawei Munich Research Center, Munich, Germany

1 Introduction

Following modern Software Engineering trends, systems are becoming larger and more distributed, requiring new solutions and new development patterns. One approach that emerged in recent years is to decouple large monolithic components into interconnected, functionally small, components that encapsulate and provide specific logic. These components are known as microservices and have become mainstream in the enterprise software development industry [13]. Despite their organizational and technical advantages [9, 39], Fine-Grained Distributed Systems (FGDS), specifically microservices increase system complexity, thus

turning anomaly detection into a more challenging task [14].

To tackle this problem, operators resort to state observation techniques like monitoring [11], logging [19], and end-to-end tracing [41]. Monitoring consists in measuring aspects like Central Processing Unit (CPU) and hard drive usage, network latency and other infrastructure metrics around the system and components. Logging provides an overview to a discrete, event-triggered log. Tracing is similar to logging, but focuses on registering the flow of execution of the program, as requests travel through several system modules and boundaries. Distributed tracing preserves causality relationships when state is partitioned over multiple threads, processes, machines and even geographical locations. In particular for anomaly detection in FGDS, distributed tracing tools—like Jaeger [44] or Zipkin [3]—, are currently the state of the art. They are used to looking for traces that take too long to execute or exhibit other unexpected behaviours; However, due to the volume of data, this task is hard and tedious to perform, and tools fail to direct the attention of operators to notice the interesting time-frames or traces. For example, to find traces involved in an anomalous region of operation, one must manually query the distributed tracing tool based on time and annotations (developer defined properties).

To improve the state of the art, and make systems more autonomic, tracing analysis needs to be automated to produce higher order constructs that provide insights for operators. The objective is to automatically find anomalies from traces.

We developed a number of tools to process traces and used machine learning algorithms to look for anomalies. The resulting data drives operators towards anomalous locations, in the temporal and service dimensions (i.e., time-stamp and service or a particular trace), reducing the search space. In particular, we created an OpenTracing Processor (OTP) to extract metrics from traces and fed them to our *Data Analyser*, which identifies anomalies in time-series of number of in-calls, out-calls and response times. To validate our approach and tools, we used production tracing data provided by Huawei Germany, from their Cloud Platform.

Results show that our approach can identify anomalies in FGDS by time-frame and services; However, one of the most interesting results of our analysis was to realize that the OpenTracing standard was in itself a limiting factor. To start, OpenTracing lacks support tools to create and analyse dependency graphs and span trees. While Zipkin manages the latter, it does not export such data in a structured form. Extracting knowledge from this unprocessed data will require manpower proportional to its volume—to the point of being untreatable in web-scale systems. The time-stamp fields, indicating when spans start and end, are not labelled with units, leading to mistakes such as spans in a dataset, showing up in various units. In particular, in our data, we found both milliseconds and microseconds. Other parts of the specification are also too ambiguous, as they allow arbitrary key-value pairs in annotations that explode the possible ways of expressing the same measurement. This is the case of error codes, function returns, Uniform Resource Locators (URLs) and other fields that vary between spans. As a result, the trace data is not given to computational processing, requiring a data cleaning step and most of the time statistical or machine learning approaches to work around ambiguity. Finally, we also observed that trace quality varies widely. This suggests that tracing frameworks, like OpenTelemetry [6], which is currently under development, should support testability, by providing developers with concrete metrics of quality, capable of improving instrumentation and, therefore, the resulting traces.

To summarize, we make two contributions in this paper: i) we developed processing and analysis tools for OpenTracing data; and ii) we identified important limitations of OpenTracing, which might help other researchers, namely those building distributed tracing tools and standards.

The rest of the paper is organized as follows. Section 2 presents the state of the art for this research. Section 3 describes the proposed solution. Section 4 we show and evaluate the results and the strengths of this approach. Section 5 we discuss a set of limitations we found concerning both methods and standards. Section 6 concludes the paper and describes future directions.

2 State of the Art

In this section we provide some background notions, as well as an overview of similar approaches. Having only been adopted recently by the industry, the number

of anomaly detection approaches using tracing is relatively small.

2.1 Core Concepts

Distributed tracing [41] is a method that comes from traditional tracing, but applied to a distributed system at the work-flow level. Unlike simple logging, tracing must relate information from different parts of the system, to order events according to some order, like Lamport's happens-before relation [23], serving multiple purposes, such as identifying the root-cause of anomalies or perform distributed profiling, and monitor applications, especially those built using microservice architectures and, in the end, it can be used to pinpoint failures and reason about their root cause.

A number of tools and standards emerged from this concept. For example, the OpenTracing standard [33] uses baggage passing mechanisms (e.g. see [12]), to connect together a tree of scoped units of work, like threads, functions, and services. For generality, OpenTracing extended their model to support full directed acyclic graphs instead of just trees. These traces reveal the causal connections between such units of work throughout the system.

OpenTracing uses dynamic, fixed-width meta-data to propagate causality between spans, meaning that each span has a *trace identifier* common to all spans of the same trace, as well as a *span identifier* and *parent span identifier* representing parent/child relationships between spans [40]. The standard defines the format for spans and the semantic [34, 35] conventions for their content/annotations.

Usually, the span has an operation name, start timestamp, duration and some annotations regarding the operation itself. An example of a span can be a Remote Procedure Call (RPC) or Hypertext Transfer Protocol (HTTP) call annotated with source, destination and possibly user defined logs/data. We provide insight of how spans are related to each other and with time in Fig. 1. As we can see, spans spread over time, overlapping each other since nothing prevents the occurrence of multiple calls in a short period or simultaneously. From a trace like this, one may extract a span tree, as the one we show in Fig. 2.

However, this specification may not be sufficient—namely, it is not strict enough to be quantitatively tested. Furthermore, the semantic conventions are very generic and leave most decisions to the practitioner. This lead to incoherence in traces, even inside the same organization, and undermines the creation of tools to automate their analysis.

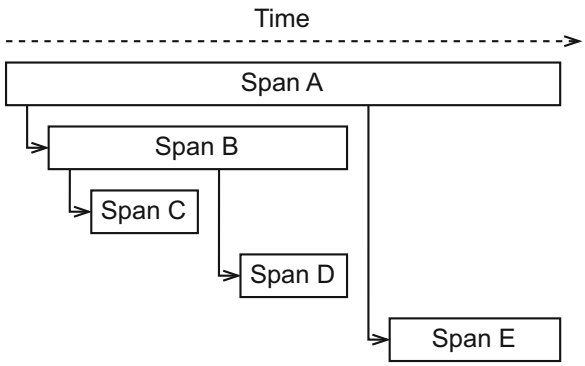

Fig. 1 Sample trace over time

From a representative set of traces and their respective span trees, we are able to extract the service dependency graph. Figure 3 shows a possible dependency graph generated from the span tree in Fig. 2. Service A, the root, directly uses services B and E, which on turn use C, D, and F; F uses G. We used dashed arrows after service E, because these dependencies do not come from the trace of Fig. 1, but from the traces of other invocations involving E. From these invocations, we can produce request work-flows. Request work-flows represent the path carried out by one request throughout services in the system. For example, from the dependency graph presented in Fig. 3, a clear work-flow is: *Service A* → *Service E* → *Service F* → *Service G*. This request work-flows can be used to trace and study service and business process interactions.

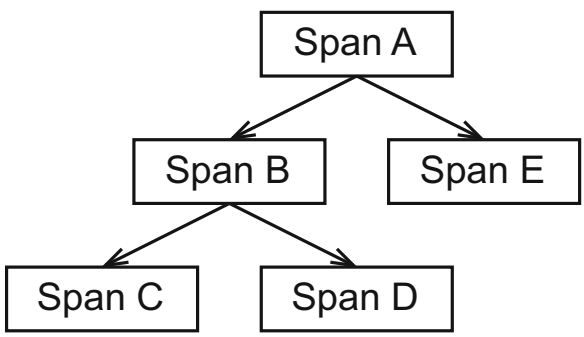

Fig. 2 Span tree example

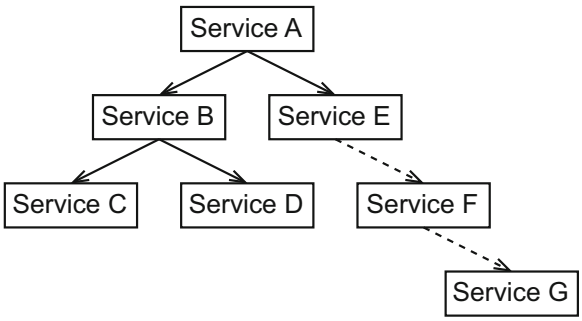

Fig. 3 Dependency graph example

2.2 Distributed Tracing Tools

Distributed tracing tools fetch or receive trace data from complex distributed systems—such as microservice-based ones—and process this data, before presenting it to the user using more readable charts and diagrams. Among other things, these tools provide the possibility to perform queries on the tracing data, e.g., by trace identifier and by time-frame. Table 1 presents a comparison of open source tracing tools.

The two tools we compare, Jaeger [44] and Zipkin [3], are very similar. Their advantages include the availability of source code, containerization, support for well known span transport technologies, and span aggregation for representation in a browser; However, they are focused on span and trace lookup, and presentation, not doing any type of automated analysis or processing. For example, they lack mechanisms capable of pinpointing anomalies in specific microservices or work-flows / requests, leaving this kind of work to operators, whom must perform manual trace and span inspection.

In summary, while generating, persisting, sorting and representing tracing data is certainly a good starting point for these tools, they still lack more advanced features for autonomic system analysis. Application Performance Monitoring Tools usually sport some analysis capability, but they are typically expensive full-stack observability suites [32].

2.3 Related Work

To contextualize our contributions, this sub-section summarizes the related work found in the literature. Automating tracing analysis has been attempted for classic tracing, where the data is usually from a single process or machine, and focuses on lower level calls, such as functions and kernel calls. In this vein, [22] present a method to detect anomalies in features extracted from Linux kernel traces. While the subject of anomaly detection from tracing features is a shared concern, our approach focuses on the distributed nature of FGDS—on its unique aspects,

Table 1 Distributed tracing tools comparison

	Jaeger [44]	Zipkin [3]
Brief description	Released as open-source by Uber Technologies. Used for monitoring and troubleshooting microservice-based distributed systems.	Helps gathering timing data needed to troubleshoot latency problems in microservice applications. It manages both the collection and lookup of data. Zipkin's design is based on the Google Dapper paper [42].
Pros	Open-source; Docker-ready; Collector interface is compatible with Zipkin protocol; Dynamic sampling rate; Browser User Interface.	Open-source; Docker-ready; Allows multiple span transport technologies (HTTP, Kafka, Scribe, AMQP); Browser User Interface.
Cons	Only supports two span transport technologies (Thrift and HTTP).	Fixed sampling rate.
Analysis	Dependency graph view; Trace comparison.	Dependency graph view.

such as morphological analysis—and the existing distributed tracing standards.

As instrumentation cost is relevant for the application of distributed tracing, there is significant research in attempting to automate or circumvent instrumentation. There are tools that attempt to automate instrumentation, either at code runtime or middle-ware levels [4, 10, 28]. Others attempt an inference-based tracing approach, statistically extracting causal order, making it transparent to the services themselves and treating them as black boxes [2]. [5, 37] do the same, with focus on systems of microservices and exploiting the observation features of the underlying platforms, such as service meshes and cluster managers. In contrast, we assume that the instrumentation effort has already been carried out, as it is gaining popularity in the industry to solve FGDS observability issues. Furthermore, this affords higher confidence in the results, especially for statistically rare work-flows or occurrences.

On the subject of tracing collection—which we approach in our suggestions to improve the usefulness of tracing for automated analysis—[25] propose Sifter, a trace sampler built to bias the sampling decision towards edge cases and rare work-flows. [21] propose Canopy, a comprehensive instrumentation, collection and analysis framework, that decouples those steps and allows dynamic feature extraction using a Domain-Specific Language (DSL).

Artificial Intelligence for Information Technology (IT) Operations (AIOps), the application of artificial intelligence to operations [18] was introduced in 2016 [26] to develop new methods to automate and enhance IT operations. Firstly, it recognizes the difficulty of manually managing distributed infrastructures and system state; secondly, the amount of data that has to be retained keeps growing, creating a plethora of problems to operators; thirdly, the infrastructure itself is becoming more distributed across geography and organizations, as evidenced by trends, like cloud-first development and fog computing. In this new field, there are a few interesting applications to tracing analysis. [30] use deep learning, trained on encoded traces, to detect anomalies with recourse to distributed tracing, in particular of cloud systems (OpenStack). This approach attempts to uncover features automatically and determine anomalous operation and traces. The amount of data needed to train these models is considerable, and is limited to classifying a trace as normal or abnormal, losing detail and interpretability, i.e., no justification for the classification. By comparison, our approach focuses on a fixed set of features, related to operation metrics, and morphology, such as connectivity degree and work-flow, and uses interpretable machine learning methods.

Looking at practical applications of tracing analysis, at IBM, [27] have achieved good results with AIOps for trace and other observability data analysis; they present a complete data processing pipeline, from ingestion to actionable insight, as well a successful evaluation on a production cloud. [7] use trace analysis together with fault injection to improve failure propagation analysis in cloud systems. Similarly, [46] developed a model to predict latent errors and localize them by learning from distributed tracing. The model was trained using data generated under a fault injection load.

3 Problem Statement and Proposed Solution

Improving observability in a large-scale distributed system serves the main purpose of driving the system towards responsiveness [20], which implies resilience [24] and elasticity [17]. Elasticity depends on the ability of the system to scale horizontally with load and on the availability of the provider to support such scaling with more (or less) resources. According to Laprie [24], resilience is:

"The persistence of service delivery that can justifiably be trusted, when facing changes".

Since, in most implementations, elasticity relies on simple direct metrics, like CPU occupation or response latencies, observing a system mostly serves to ensure resilience. In our particular case, our system was an OpenStack cluster that Huawei Research uses for testing purposes, of which we had an anonymised OpenTracing data set, limiting us to use our method in an off-line *post mortem* analysis of the cluster. Hence, based on the existing traces, and the metrics derived from them, like the number of incoming and outgoing requests, response times or service error codes, to look for threats to resilience. To achieve this goal, we asked two research questions:

1. Is there any anomalous behaviour in the system?
2. If yes, where?

One can easily see that a timely answer to these questions is very helpful for resiliency during system

operation. However, the sheer number of components and metrics, like the number of incoming and outgoing requests, response times, downtimes, error codes, and so on, requires a significant capacity to collect and process data, but above all the need to know where and what to look for. By doing the *post mortem* analysis on the tracing data, we aim to find appropriate methods to answer these questions.

The data from Huawei Research consisted of two JSON Lines (JSONL) files, one file per day of operation. Table 2 provides additional details on the data. Each file has around 200,000 Spans, composing about 70,000 Traces. This file format is an extension of the JavaScript Object Notation (JSON) file type. In JSONL, multiple JSON objects, each encoding a span, are separated by a new line character.

3.1 Solution

Algorithm 1 Algorithm for metrics extraction from tracing.

```
Data: Trace files/Trace data.
Result: Trace metrics written in the time-series
        database.
Connect to Time-Series database;
Read time_resolution, start_time and end_time
from configuration;
Read traces from trace files/trace data;
Post traces to Zipkin;
Get services from Zipkin;
Calculate time_intervals using start_time,
end_time and time_resolution;
foreach time_interval in time_intervals do
    Get service_dependencies from Zipkin;
    Build system_dependency_graph using
    service_dependencies;
    Extract graph_metrics from
    system_dependency_graph;
    foreach service in services do
        Get Spans from Zipkin;
        Aggregate Spans into Traces;
        Extract service_metrics from Traces;
    Post graph_metrics to Time-Series database;
    Post service_metrics to Time-Series database;
```

Before we could run the data analysis tool, we had to extract metrics from tracing data and write them into a time-series database. For this operation, we used Algorithm 1 which retrieves tracing data from Zipkin, links spans to rebuild *Traces* and *Service Dependency Graphs* in memory, and finally, extracts pre-defined metrics from these structures, to store them in the time-series database for visualization and analysis. Currently, OTP is extracting and analysing the following data from tracing, for a given time interval:

- Number of incoming/outgoing calls per service.
- Average response time per service.
- Changes to service neighbourhood (both for incoming and outgoing calls).

If necessary, for the sake of analysing the system, one could extract other metrics, like service connection degrees, number of services traced over time, or number of entering and departing services over time. We did not use these additional metrics in this paper, because we did not find them useful for the particular traces under analysis.

Table 2 Data set provided for this research

File date	June 28th	June 29th
Spans count	190 202	239 693
Traces count	64 394	74 331

3.2 Implementation

We followed the two-step high-level approach of Fig. 4. The tracing data feeds OTP, which derives higher order metrics from tracing data, before storing them in a time-series database. The existing tracing back-ends, can only export spans, leaving the reconstruction of traces (connecting the spans as a tree) to the user. OTP does this using Java Streaming Application Program- ming Interface (API) [36], leveraging its parallelization capabilities. Service dependency graphs, were extracted and processed using NetworkX [31], a Python-based framework for graph processing, containing a large set of graph algorithms. OpenTSB [43] was used to store the derived metrics that follow. To visualize the extracted metrics, we used Grafana [16].

The other tool aims to perform metric analysis from the time-series database. Since our data is unlabelled, i.e., it has no classification; therefore, our analysis uses unsupervised learning algorithms. We chose Isolation

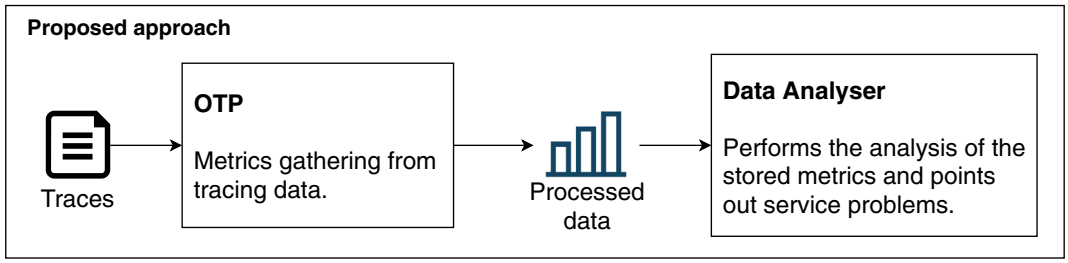

Fig. 4 Proposed solution

Forests [29], as the starting point, as it allows outlier detection in a multidimensional space. We developed this component as a collection of Python scripts in Jupyter Notebooks [38]. We used Pandas [45], to process time-series data, and Scikit-learn [8], to provide an implementation of Isolation Forests.

The main goals of this pair of components was to find the set of interesting time-frames in a large set of traces, thus relieving operators from the need to conduct unguided, sometimes exhaustive, search using Zipkin that are mostly limited to tracing visualisation features. The code and documentation of our work are available in GitHub.[1]

4 Results and Analysis

In this section we present the results gathered from the Data Analysis component presented in Section 3, to identify and locate anomalous behaviour in the system. As we worked towards this goal, the quality of the tracing data became a problem, leading us to formulate an additional research question. Furthermore, even though we had traces and were able to identify anomalous regions, we did not have access to issue reports and were unable to validate accuracy and precision.

4.1 Anomaly Detection

Figure 5 provides a representation of two time-frame samples of the same service, one for an anomalous region, and another for a non-anomalous region as tagged by our *Data Analyser* component. We set the time-series resolution to 10 minutes—to avoid intervals with too few traces—, and considered the number of incoming and outgoing requests, in conjunction with the average response time.

Figure 6 presents the comparison between detected Anomalous and Non-Anomalous time-frames in unix time stamp for a given service. This information, represented in Fig. 6, was the result of outlier detection, considering three service metrics: number of incoming requests, number of outgoing requests and average response time. Anomalies identified by the algorithm, are indicated by vertical red lines. In addition to the metrics used for anomaly detection, we include an additional time-series, which denotes the morphological changes to the service dependency graph.

As we can see in Fig. 5, the difference between anomalous and non-anomalous operation is made clear by the presence of outliers. In the anomalous samples, points form a cluster near the chart origin, with some outliers on the upper-left and down-right regions of the chart. Meanwhile, in the non-anomalous samples, there is only a clustering of points near the chart origin.

The next step of our analysis is to determine the cause for the outliers in the anomalous samples, i.e., what exactly is causing this unexpected increment in the number of incoming/outgoing requests—which accounts for load variation—and average response times, more precisely:

- Some services take longer to respond even when the system is lightly loaded, with few incoming/outgoing requests.
- Some services are receiving more incoming/outgoing requests, but still responding fast.

An elastic system should be capable of handling more requests and still reply in an expected amount of time. However, if the service quality degradation in response to increased load is to steep, this represent some error condition that we must rule out.

[1] OpenTracing Processor (OTP), https://github.com/andrepbento/OpenTracingProcessor

Algorithm 2 Work-flow type algorithm.

```
Data: Trace files/Trace data.
Result: Set of all unique work-flow types, and
        their frequency and request duration
        statistics. Formatted to Comma
        Separated Values (CSV).
Read configuration to determine analysis
time-frame;
Read traces within defined time-frame;
WorkflowSet = {};
foreach trace in Traces do
    workflow = EmptyWorkflow();
    foreach span in trace do
        if span is service call then
            workflow.addEdge(span.service,
            span.remoteServiceCall);
    if workflow not in WorkflowSet then
        workflow.requests = 1;
        workflow.requestTimeList.add(trace.duration);
        WorkflowSet.add(workflow);
    else
        workflow = WorkflowSet.get(workflow);
        workflow.requests++;
        workflow.requestTimeList.add(trace.duration);
foreach workflow in WorkflowSet do
    count = workflow.requests;
    respTime =
    mean(workflow.requestTimeList);
    writeToCsv(workflow, count, respTime);
```

Subsequently, we analysed the work-flow types. A work-flow is a class of requests, or traces, that share the same service invocation graph. Usually they represent a type of request or business process. The objective of this analysis is to understand if there is something wrong with the request work-flow paths, such as degenerate paths resulting from missing services. Algorithm 2 illustrates our approach to collect all work-flows present in the tracing data. The process to collect work-flows involves pinpointing requests between services present in the span meta-data and then storing a list of all unique graphs.

In Fig. 7 we show the most common work-flows from the Anomalous and Non-Anomalous time-frames. Given the large number of work-flows that exist in the system, we encoded them numerically.

One interesting fact to notice is that, in the anomalous regions, there are more request work-flows types. The next step would be to check what was causing this increase, by retrieving the most invoked work-flow. Unfortunately, we were unable to continue down this path, because tracing data was incomplete. The flows were not relevant for a further analysis because they were just calls between a gateway and a service. Moreover, the gateway instrumentation was incomplete; logging the type of request and service name but not the endpoint. At this point, and for this question, it is possible to say that this data set was exhaustively analysed, and an improvement of the tracing data should be the path to take.

4.2 Trace Quality Analysis

Once we concluded the impossibility of going deeper in the analysis of the tracing data, we questioned how we could measure the quality of tracing. Our approach to this question was to process the tracing data and feed it to the *Data Analysis* component, this time without using a time-series database in-between. We divided this analysis into two procedures. The first procedure checks if the spans complies with the OpenTracing specification. According to the algorithm, every span structure complies with the specification. The problem here is that the OpenTracing specification is not very strict and therefore, this testing algorithm cannot provide very accurate results. For example, the units for time-stamps are not uniform, one can use milliseconds in one field and then, in another field of a span, in the same trace, time might be in microseconds. This leads to problems in time measurements, but the specification, and the very design of the standard, make it difficult to detect computationally in a deterministic manner. We discuss a possible redefinition of the OpenTracing specification in Section 5.

The second procedure checks if tracing covers the entire time of the root spans. For a simple example, if we have a trace with a root span of 100 ms, and this root span has two children spans, one with 50 ms, the other one with 40 ms, the entire trace has a temporal coverage of $(50 + 40)/100 = 90\%$. We apply this method to every trace, and plot the results; furthermore, we split them by service, with the objective of determining the time coverage of tracing by service.

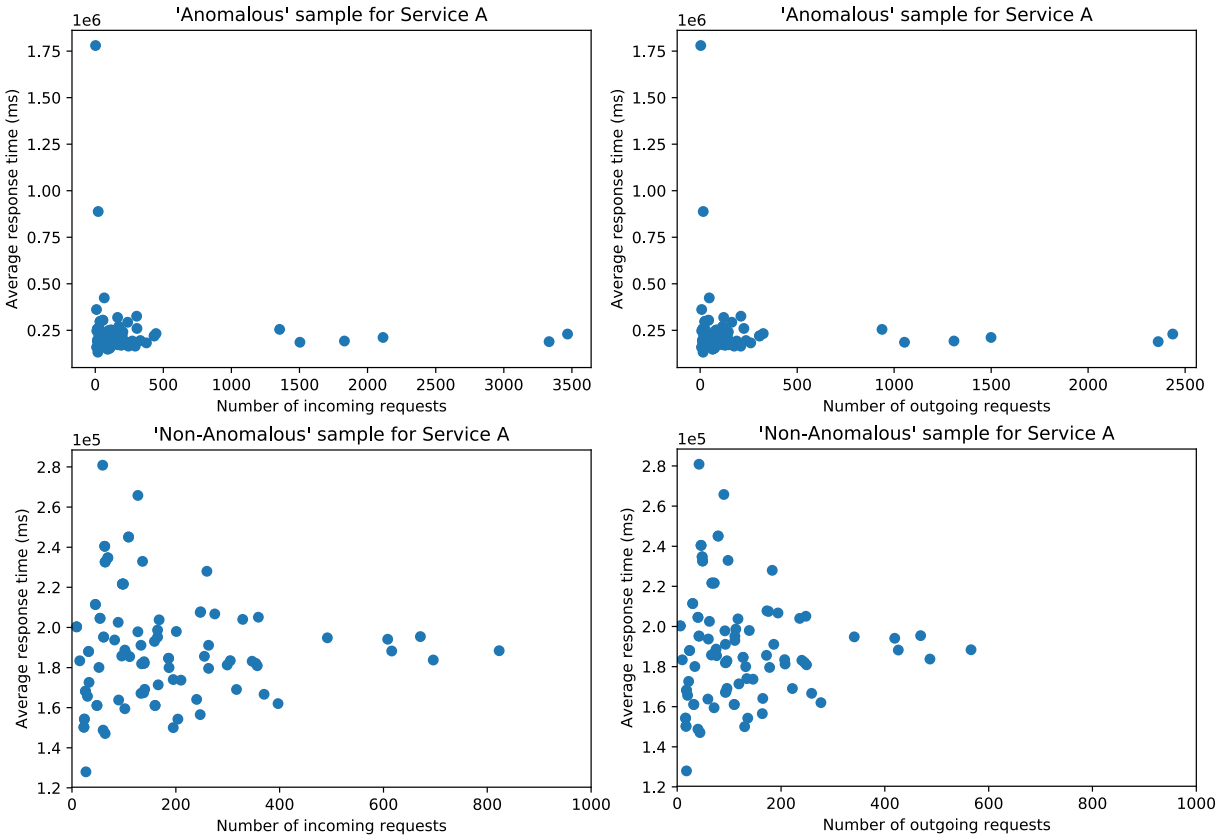

Fig. 5 Comparison between Anomalous and Non-Anomalous service time-frame regions

The results, regarding two different services are displayed in Fig. 8, depicting the coverage histograms for two different services. Each time the service shows up in a trace, we calculate the percentage of time covered, to produce the histogram. It is important to notice the good coverage level—in the 60% − 100% range. This means that coverage for this tracing could be better, but it is nonetheless good. This points to the fact that even a relatively high temporal coverage is not a sufficient quality indicator for automated anomaly detection.

5 Tracing Standard Limitations and Mitigations

The quality of our anomaly detection method was bounded by the quality of the data. Specifically, the tracing dataset presented problems in completeness and homogeneity, which we reason is a consequence of ambiguity in the tracing format specification standard. Limitations in the tooling exacerbate the aforementioned issue—there is no tool to perform tracing quality evaluation—, and made it necessary to develop tools to treat and analyse data. Furthermore, exploratory analysis of tracing data is difficult as there are no tools available for this purpose. We categorised and sub-divided this issues in Fig. 9. They can be roughly divided in three groups, data sufficiency, ontological, and tools.

Data available in the tracing dataset must be sufficient for the analysis. This means that the instrumentation needs to cover the code-base, like unit tests would have to, as well as time. As described in Section 4.2, a span should have its children spans cover as much as its time-frame as possible, limited by code granularity. Even though trace sampling becomes a necessity at scale, to avoid unacceptable overhead, it needs to be balanced against representativeness of the data and, in turn, of the intelligence extracted from it. In order to be able to capture rare events as well as enough volume in a low throughput system, sampling-rate

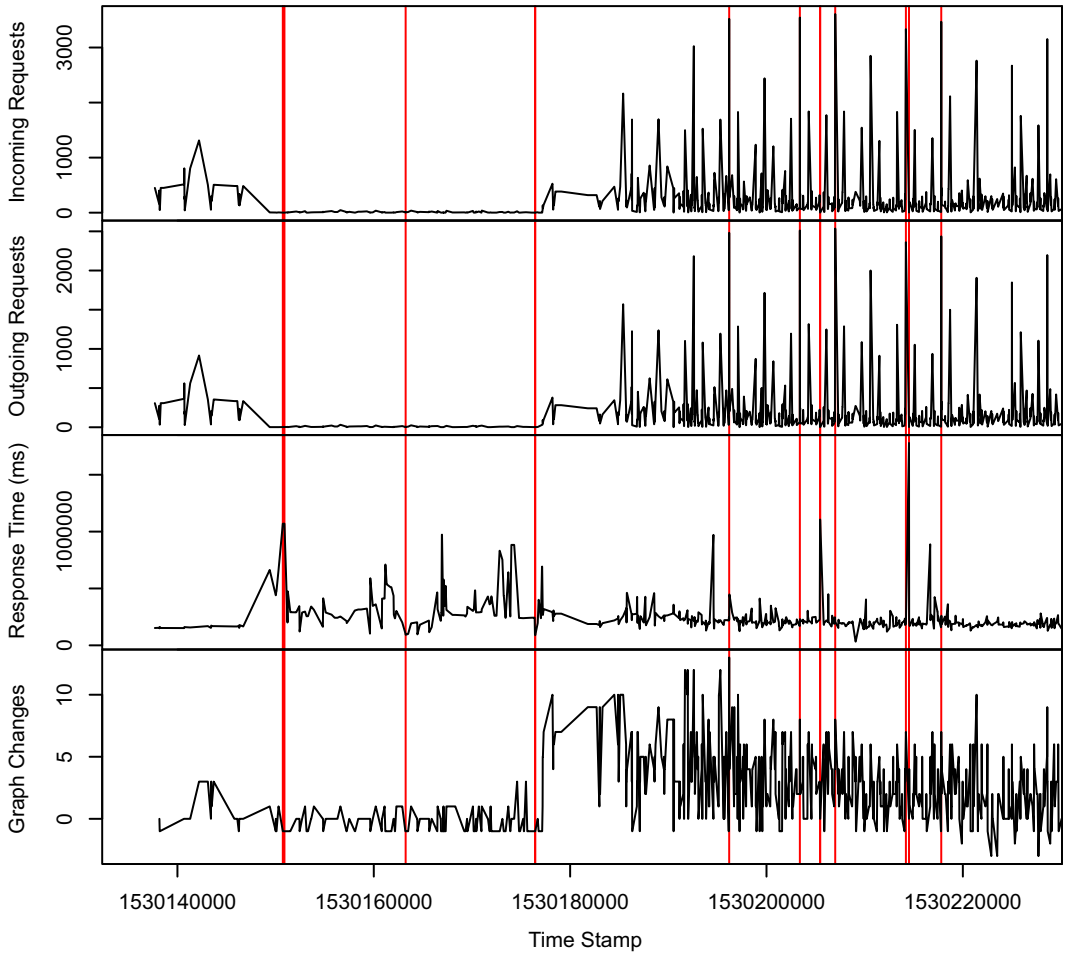

Fig. 6 Metrics over time and detected anomalies

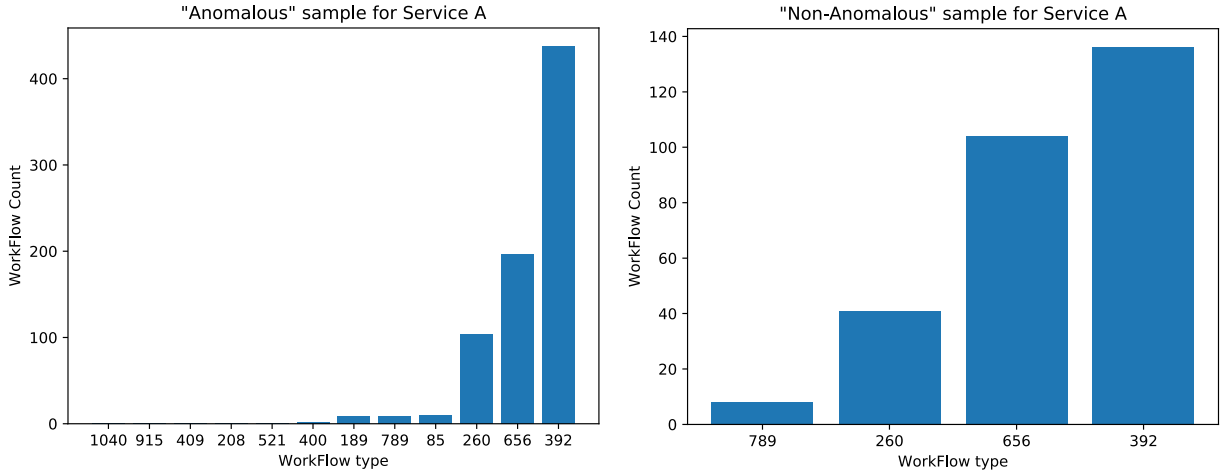

Fig. 7 Comparison between Anomalous and Non-Anomalous service work-flow types

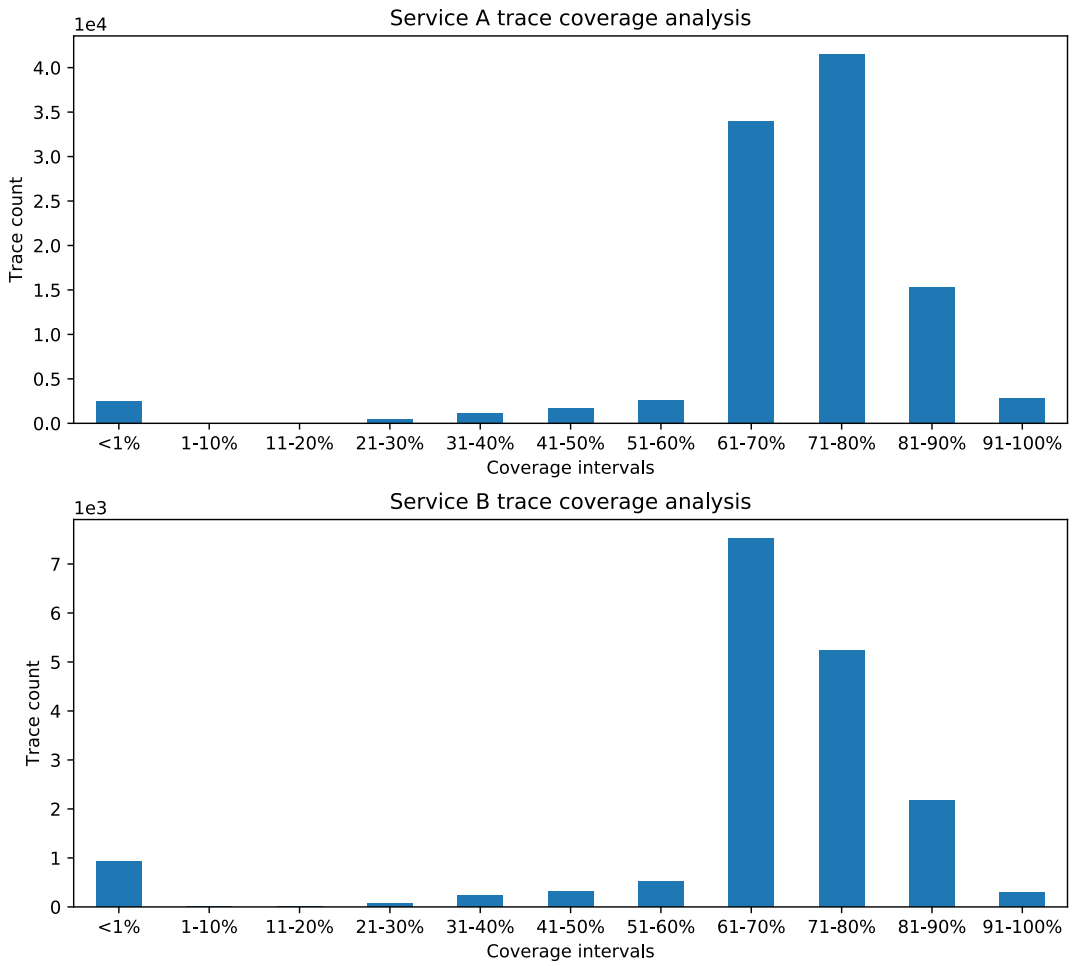

Fig. 8 Services coverage analysis

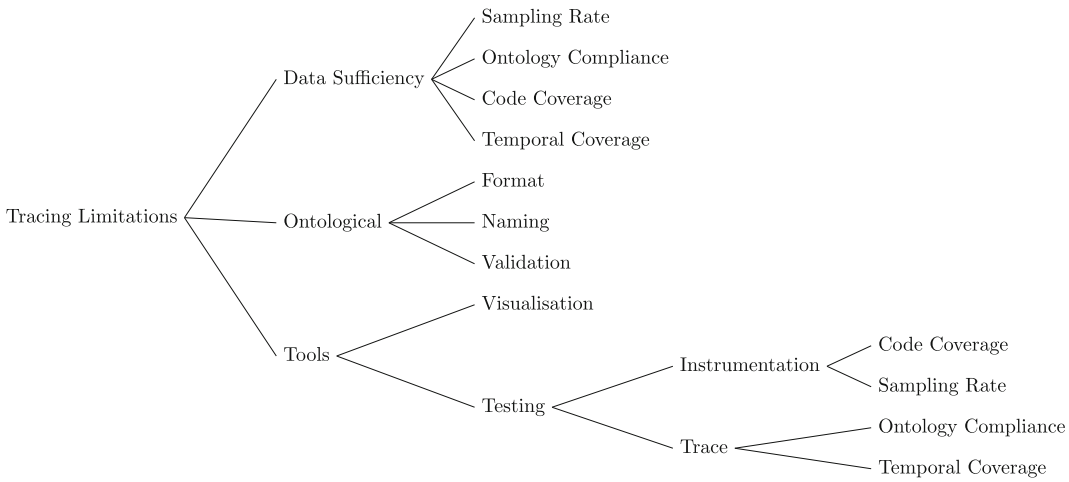

Fig. 9 Categorization of Tracing Limitations

must be adaptive, preferably staying at 100% until the overhead becomes a problem.

In addition to the previous considerations, which focused on the data set level (set of traces), it is also important to pay attention to individual traces, in other words, "internal trace quality". Traces should have enough information to clearly and uniquely identify all the components in the system, their invocation / causality relationships, as well as their context (e.g., physical computer, run-time, real time clock). Note that component, might refer to a executable artefact, run-time, service or be as granular as classes and functions.

Spans should follow an ontology that enforces a strict tracing schema upon traces, implying archetypes for format and naming, as well as validation features. With respect to format, spans parameters should follow specific descriptions, lowering the difficulty of employing statistical analysis and machine learning methods. As an example, the definition of span start and end time-stamps should be a compound type, containing a 64 bit integer numeric value and a time unit (e.g., milliseconds (ms), microseconds (μs)); in terms of validation, there should be bounds of what is an acceptable, relatively current time-stamp (e.g., ± 1 month), and an assertion that the end time-stamp is greater than the starting one.

In line with the current practices of Continuous Development (CD) / Continuous Integration (CI), there is the need to fully automate the suggested practices. To begin, to make sure that the code-base is sufficiently covered by the tracing instrumentation, developers need a tool that can be integrated in their development and quality assurance pipelines as means of enforcement. From the testing perspective, and looking at the resulting traces themselves, there should be tools capable of testing compliance with the ontology / specification and clearly output errors and warnings about any inconsistencies. To ensure that this is noted by the developers, the tracing back-end tools should, for example, refuse to ingest incongruent data. Note that this is only possible if the tracing format is sufficiently strict and well defined, leading back to the need for an ontology.

Concerning visualization, traces are not easy to present as a result of being tree structures representing a path on the dependency / architectural graph; additionally, there are no tools to filter and project results according to multiple dimensions that might be of interest (e.g., time, work-flow, dependency graph / architecture, physical infrastructure, run-time, and others).

Data exploration is another use case that would benefit from better tooling. When debugging a system or attempting to reason about its behaviour, it is useful to be able to view data in context and follow connections across different dimensions, for example, find logs or infrastructure metrics related to a particular trace, or traces from a specific work-flow. This could be implemented in a number of ways, as an example, relate logs to spans by adding *span identifiers* to logs, and relate infrastructure metrics by time-stamp and physical machine; alternatively, this could be achieved by creating a unified observation standard that collects all observability data, therefore having enough information to preserve their relation.

We believe that the industry is experiencing the same limitations, regarding tools and standards, as we did in this work, as evidenced by an ongoing open source initiative entitled *OpenTelemetry* [6]. Supported by big companies, such as *Google*, *Lightstep* and *Uber*, this project aims at creating a more comprehensive standard, merging *OpenCensus* [15] and *OpenTracing* [1], to enable the creation of reusable tools. Furthermore, so far, they have not satisfied all the requirements we uncovered, having mostly carried a merging effort between technologies, instead of redesigning them. For example, the consortium should consider trace testability as a driver for the design, laying the foundation to create a quantitative metric for trace evaluation, as well as the respective tools. Developers need a tool capable of determining if the analysis failed as a consequence of trace quality issues. The lack of automatic analysis leads to usability issues and code quality problems; instrumentation libraries do not guide programmers towards the correct way of using them (e.g., requiring explicit units for time-stamps).

6 Conclusion

Our results, and industry tendencies, reveal that tracing data is useful and required to find anomalies in large-scale distributed systems, where human cognition starts to fail; However, tracing data is hard to handle due to application specificity, complexity and sheer volume. We have used this information to detect

anomalous behaviours and locate them in services and time. We extracted metrics, from tracing data, as time-series, and then performed outlier detection analysis over a composite multi-dimensional time-series. Despite the elevated cost of analysing traces manually, issues addressed in this paper can only be identified using tracing data.

In the end, our analysis of the tracing data from the Huawei Cloud Platform, lead us to the following conclusions:

1. OpenTracing suffers from a lack of tools for data processing and visualisation.
2. The OpenTracing specification is not strict enough for automated analysis.
3. The lack of tools to control instrumentation quality jeopardizes the tracing effort.

These conclusions are valid for the newer OpenTelemetry standard, as it is partially derived from OpenTracing and OpenCensus [15].

Finally, the analysis we did on tracing quality lead us to another result. While we can use more or less complex tools and data analysis algorithms, the poor quality of traces compromises what can be achieved. Even when traces are appropriately organized and provide reasonable temporal coverage, the lack of a strict specification, together with the lack of code coverage considerably reduces the usefulness of tracing data.

As future work, we intend to generate a labelled dataset using fault and or failure injection to enable the use of supervised learning methods. Additionally, the industry definition of tracing in a distributed context is lacking compared to the classic tracing concept. One interesting path to follow is to extend tracing to include other aspects of system state and meta-data, such as monitoring and logging.

Acknowledgements This work was produced with the support of INCD funded by FCT and FEDER, under the project 01/SAICT/2016 n- 022153 and was partially carried out under the project P2020 - 31/SI/2017: AESOP — Autonomic Service Operation, supported by Portugal 2020 and UE-FEDER. This work was also partially supported by national funds through the FCT - Foundation for Science and Technology, I.P., within the scope of the project CISUC - UID/CEC/00326/2020 and by the European Social Fund, through the Regional Operational Program Centro 2020.

Data Availability The data that support the findings of this study are available from Huawei Research, but restrictions apply to the availability of these data, which were used under license for the current study, and so are not publicly available. Data are however available from the authors upon reasonable request and with permission of Huawei Research. All our source code was made available in GitHub.[2]

References

1. The OpenTracing Specification repository. https://github.com/opentracing/specification. Retrieved on Nov, 2018
2. Aguilera, M.K., Mogul, J.C., Wiener, J.L., Reynolds, P., Muthitacharoen, A.: Performance debugging for distributed systems of black boxes. ACM SIGOPS Operating Systems Review **37**(5), 74 (2003). https://doi.org/10.1145/1165389.945454
3. Apache Software Foundation: Zipkin. http://zipkin.io (2016). Retrieved on Feb, 2019
4. Ates, E., Sturmann, L., Toslali, M., Krieger, O., Megginson, R., Coskun, A.K., Sambasivan, R.R.: An automated, cross-layer instrumentation framework for diagnosing performance problems in distributed applications. In: Proceedings of the ACM Symposium on Cloud Computing - SoCC '19, pp. 165–170. ACM Press, New York (2019). https://doi.org/10.1145/3357223.3362704
5. Cinque, M., Della Corte, R., Pecchia, A.: Microservices monitoring with event logs and black box execution tracing. IEEE Trans. Serv. Comput., 1–1. https://doi.org/10.1109/TSC.2019.2940009 (2019)
6. Cloud Native Computing Foundation: OpenTelemetry: Effective observability requires high-quality telemetry. https://opentelemetry.io (2019). Retrieved on July, 2019
7. Cotroneo, D., De Simone, L., Liguori, P., Natella, R., Bidokhti, N.: Enhancing failure propagation analysis in cloud computing systems. In: 2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE), pp. 139–150. IEEE (2019). https://doi.org/10.1109/ISSRE.2019.00023
8. Cournapeau, D.: Scikit-learn - Machine learning in Python. https://github.com/scikit-learn/scikit-learn. Retrieved on Feb, 2019 (2007)
9. Dragoni, N., Giallorenzo, S., Lafuente, A.L., Mazzara, M., Montesi, F., Mustafin, R., Safina, L.: Microservices: yesterday, today, and tomorrow. In: Present and Ulterior Software Engineering, pp. 195–216 (2017). https://doi.org/10.1007/978-3-319-67425-4_12
10. Erlingsson, Ú., Peinado, M., Peter, S., Erlingsson, U., Peinado, M., Peter, S., Budiu, M.: Fay. Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles - SOSP '11 **13**(4), 311–326 (2011). https://doi.org/10.1145/2043556.2043585
11. Ewaschuk, R., Beyer, B.: Site Reliability engineering: How Google Runs Production Systems, chap. Monitoring Distributed Systems, pp. 55–66. O'Reilly Media Inc. (2016)
12. Fonseca, R., Porter, G., Katz, R.H., Shenker, S., Stoica, I.: X-trace: a pervasive network tracing framework. In: Proceedings of the 4th USENIX Conference on Networked Systems Design & Implementation

[2] OpenTracing Processor (OTP), https://github.com/andrepbento/OpenTracingProcessor

(NSDI'07), April, p. 20. USENIX Association (2007). https://doi.org/10.5555/1973430.1973450
13. Fowler, M., Lewis, J.: Microservices, a definition of this architectural term. https://martinfowler.com/articles/microservices.html. Retrieved on Sep, 2018 (2014)
14. Francesco, P.D., Malavolta, I., Lago, P.: Research on architecting microservices: trends, focus, and potential for industrial adoption. In: 2017 IEEE International Conference on Software Architecture (ICSA), pp. 21–30. IEEE (2017). https://doi.org/10.1109/ICSA.2017.24
15. Google LLC: OpenCensus. https://opencensus.io (2016). Retrieved on July, 2019
16. Grafana Labs: Grafana - The tool for beautiful metric dashboards. https://github.com/grafana/grafana (2015). Retrieved on Feb, 2019
17. Herbst, N.R., Kounev, S., Reussner, R.: Elasticity in cloud computing: what it is, and what it is not. Presented as part of the 10th International Conference on Autonomic Computing, 23–27 (2013)
18. Jacob, S.: The Rise of AIOps: How Data, Machine Learning, and AI Will Transform Performance Monitoring. https://www.appdynamics.com/blog/aiops/aiops-platforms-transform-performance-monitoring. Retrieved on Mar, 2019 (2019)
19. Janapati, S.P.R.: Distributed Logging Architecture for Microservices. https://dzone.com/articles/distributed-logging-architecture-for-microservices. Retrieved on Feb, 2019 (2017)
20. Jonas Bonér Dave Farley, R.K., Thompson, M.: The Reactive Manifesto. https://www.reactivemanifesto.org. Retrieved on Jun, 2019 (2014)
21. Kaldor, J., Mace, J., Bejda, M., Gao, E., Kuropatwa, W., O'Neill, J., Ong, K.W., Schaller, B., Shan, P., Viscomi, B., Venkataraman, V., Veeraraghavan, K., Song, Y.J.: Canopy: an end-to-end performance tracing and analysis system. In: SOSP 2017 - Proceedings of the 26th ACM Symposium on Operating Systems Principles, pp. 34–50. ACM Press, New York (2017). https://doi.org/10.1145/3132747.3132749
22. Kohyarnejadfard, I., Shakeri, M., Aloise, D.: System Performance Anomaly Detection Using Tracing Data Analysis. In: ACM International Conference Proceeding Series, vol. Part F1482, pp. 169–173. ACM Press, New York (2019). https://doi.org/10.1145/3323933.3324085
23. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Communications of the ACM **21**(7), 558–565 (1978). https://doi.org/10.1145/359545.359563. http://amturing.acm.org/p558-lamport.pdf, http://portal.acm.org/citation.cfm?doid=359545.359563
24. Laprie, J.C.: From dependability to resilience. In: 38th IEEE/IFIP Int. Conf. on Dependable Systems and Networks, pp. G8–G9 (2008)
25. Las-Casas, P., Papakerashvili, G., Anand, V., Mace, J.: Sifter: scalable sampling for distributed traces, without feature engineering. In: Proceedings of the ACM Symposium on Cloud Computing - SoCC '19, pp. 312–324. ACM Press, New York (2019). https://doi.org/10.1145/3357223.3362736
26. Lerner, A.: AIOps Platforms. https://blogs.gartner.com/andrew-lerner/2017/08/09/aiops-platforms. Retrieved on Jun, 2019 (2017)
27. Levin, A., Garion, S., Kolodner, E.K., Lorenz, D.H., Barabash, K., Kugler, M., McShane, N.: AIOps for a cloud object storage service. In: 2019 IEEE International Congress on Big Data (Bigdatacongress), pp. 165–169. IEEE (2019). https://doi.org/10.1109/BigDataCongress.2019.00036
28. Li, H., Oh, J., Oh, H., Lee, H.: Automated source code instrumentation for verifying potential vulnerabilities. IFIP Advances in Information and Communication Technology **471**, 211–226 (2016). https://doi.org/10.1007/978-3-319-33630-5_15
29. Li, S.: Time Series of Price Anomaly Detection. https://towardsdatascience.com/time-series-of-price-anomaly-detection-13586cd5ff46. Retrieved on Jan, 2019 (2019)
30. Nedelkoski, S., Cardoso, J., Kao, O.: Anomaly detection from system tracing data using multimodal deep learning. In: 2019 IEEE 12th International Conference on Cloud Computing (CLOUD), vol. 2019-July, pp. 179–186. IEEE (2019). https://doi.org/10.1109/CLOUD.2019.00038
31. NetworkX developers: NetworkX. https://networkx.github.io (2014). Retrieved on Nov, 2018
32. New Relic, Inc.: Newrelic – deliver more perfect software. https://newrelic.com (2008). Retrieved on Jan, 2021
33. OpenTracing Specification Council: The OpenTracing Data Model Specification. https://opentracing.io/specification (2019). Retrieved on Feb, 2019
34. OpenTracing Specification Council: The OpenTracing Semantic Conventions. https://github.com/opentracing/specification/blob/master/semantic_conventions.md (2019). Retrieved on Feb, 2019
35. OpenTracing Specification Council: The OpenTracing Semantic Specification. https://github.com/opentracing/specification/blob/master/specification.md (2019). Retrieved on Feb, 2019
36. Oracle: Java Stream API. https://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html (2017). Retrieved on Feb, 2019
37. Pina, F., Correia, J., Filipe, R., Araujo, F., Cardoso, J.: Nonintrusive monitoring of microservice-based systems. In: 2018 IEEE 17th International Symposium on Network Computing and Applications (NCA), pp. 1–8. IEEE (2018)
38. Project Jupyter: Jupyter Notebooks. https://jupyter.org (2015). Retrieved on Nov, 2018
39. Richardson, C.: Microservices Definition. https://microservices.io. Retrieved on Sep, 2018 (2019)
40. Sambasivan, R.R., Fonseca, R., Shafer, I., Ganger, G.R.: So, you want to trace your distributed system? Key design insights from years of practical experience. Tech. rep., Technical Report CMU-PDL-14 (2014)
41. Sambasivan, R.R., Shafer, I., Mace, J., Sigelman, B.H., Fonseca, R., Ganger, G.R.: Principled workflow-centric tracing of distributed systems. In: Proceedings of the Seventh ACM Symposium on Cloud Computing - SoCC '16, pp. 401–414. ACM Press, New York (2016). https://doi.org/10.1145/2987550.2987568
42. Sigelman, B.H., André, L., Burrows, M., Stephenson, P., Plakal, M., Beaver, D., Jaspan, S., Shanbhag, C.: Dapper, a Large-Scale Distributed Systems Tracing Infrastructure. Tech. rep., Google LLC (2010)

43. StumbleUpon, Inc: OpenTSDB. https://github.com/OpenTSDB/opentsdb (2010). Retrieved on Feb, 2019
44. Uber Technologies: Jaeger. https://www.jaegertracing.io (2017). Retrieved on Jun, 2019
45. Wes McKinney: Pandas - Flexible and powerfull time-series data analysis. https://github.com/pandas-dev/pandas (2008). Retrieved on Nov, 2018
46. Zhou, X., Peng, X., Xie, T., Sun, J., Ji, C., Liu, D., Xiang, Q., He, C.: 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 - ESEC/FSE 2019, pp. 683–694. ACM Press, New York (2019). https://doi.org/10.1145/3338906.3338961