VAMP: Visual Analytics for Microservices Performance

Luca Traini
University of L'Aquila, Italy
luca.traini@univaq.it

Jessica Leone
University of L'Aquila, Italy
jessica.leone@student.univaq.it

Giovanni Stilo
University of L'Aquila, Italy
giovanni.stilo@univaq.it

Antinisca Di Marco
University of L'Aquila, Italy
antinisca.dimarco@univaq.it

ABSTRACT

Analysis of microservices' performance is a considerably challenging task due to the multifaceted nature of these systems. Each request to a microservices system might raise several Remote Procedure Calls (RPCs) to services deployed on different servers and/or containers. Existing distributed tracing tools leverage swimlane visualizations as the primary means to support performance analysis of microservices. These visualizations are particularly effective when it is needed to investigate individual end-to-end requests' performance behaviors. Still, they are substantially limited when more complex analyses are required, as when understanding the system-wide performance trends is needed.

To overcome this limitation, we introduce VAMP, an innovative visual analytics tool that enables, at once, the performance analysis of multiple end-to-end requests of a microservices system. VAMP was built around the idea that having a wide set of interactive visualizations facilitates the analyses of the recurrent characteristics of requests and their relation w.r.t. the end-to-end performance behavior. Through an evaluation of 33 datasets from an established open-source microservices system, we demonstrate how VAMP aids in identifying RPC execution time deviations with significant impact on end-to-end performance. Additionally, we show that VAMP can support in pinpointing meaningful structural patterns in end-to-end requests and their relationship with microservice performance behaviors.

CCS CONCEPTS

• **Software and its engineering** → **Software performance**; **Maintaining software**; **Software evolution**; • **Human-centered computing** → **Visual analytics**; **Visualization toolkits**.

KEYWORDS

Microservices, Distributed Tracing, Performance Analysis

ACM Reference Format:
Luca Traini, Jessica Leone, Giovanni Stilo, and Antinisca Di Marco. 2024. VAMP: Visual Analytics for Microservices Performance. In *The 39th ACM/SIGAPP Symposium on Applied Computing (SAC '24), April 8–12, 2024, Avila, Spain.* ACM, New York, NY, USA, Article 4, 10 pages. https://doi.org/10.1145/3605098.3636069

SAC '24, April 8–12, 2024, Avila, Spain

ACM ISBN 979-8-4007-0243-3/24/04.
https://doi.org/10.1145/3605098.3636069

1 INTRODUCTION

Microservices have emerged as a pivotal change in the software industry, paving the way to a novel paradigm for structuring the software development process. This novel approach entails multiple independent teams responsible "from development to deploy" [31] of loosely coupled independently deployable services [30, 31]. Due to their modular nature, microservices are particularly well-suited for the modern software industry, where rapidly releasing software updates and enhancements is a critical competitive advantage [34].

Although beneficial in many aspects, microservices also introduce new challenges, especially when it comes to maintaining consistent software performance. This complexity arises from various elements. Firstly, the inherent complexity of these systems often hinders the adoption of proactive measures for performance assurance[38, 45], such as pre-production performance testing [17, 21, 42]. Secondly, these proactive measures are often hampered by time and resource constraints due to the substantial pressure to deliver fast-to-market [34, 40]. Thirdly, microservices systems typically exhibit an emergent performance behavior in the field that is hard to predict in advance [45]. Finally, these systems undergo continuous software changes, with multiple releases occurring on a daily basis, and handle highly variable workloads [3], which make them more vulnerable to unforeseen performance regressions [43, 45].

These challenges have led to an increased interest in the concept of *observability* [29], *i.e.,* the ability to have a holistic understanding of the system's performance by analyzing its logs, traces, and metrics. Distributed tracing tools [32] are today widely used in practice to enhance observability of microservices systems [28]. These tools track and record the propagation of requests as they flow through different RPCs and services of a microservices system [35], and provide visual aids to support performance analysis of end-to-end requests, *e.g.,* swimlane visualizations [10, 37, 39].

Despite their utility, distributed tracing tools have recently been criticized for their limited support for performance analysis [10]. A common use case for these tools is the analysis of the system-wide performance behavior [32], such as understanding the response time distributions of end-to-end requests [10]. However, current distributed tracing tools often fall short in this area, necessitating a switch between various visualization tools, which can make the process cumbersome and time-consuming [10]. Indeed, they primarily focus on the analysis of individual requests, which has

limited value unless it is compared with the performance behavior of the entire corpus of requests [2, 10, 32].

In this paper, we introduce VAMP, an innovative visual analytics tool designed to enhance the performance analysis of microservices systems. VAMP extends the conceptual proposal of Leone and Traini [22]. The fundamental idea underpinning VAMP is to simplify the understanding of the relationship between request characteristics and end-to-end response time behavior through interactive charts and color-encoding techniques. VAMP comprises two main visualization components: an interactive tree that illustrates the workflow in terms of RPCs for multiple end-to-end requests, and an interactive histogram representing the end-to-end performance behavior of the requests under analysis. Interaction with these visual components aids in identifying the unique characteristics of certain RPC execution paths with respect to some specific system performance behaviors.

We evaluate VAMP using 33 datasets derived from TrainTicket [47], an open-source microservices system widely utilized in previous software engineering research [23, 41, 46]. Our findings demonstrate that VAMP enables the identification of notable and recurrent request characteristics associated with specific end-to-end response time behaviors.

A video of VAMP in action can be accessed at https://youtu.be/qMVOMt06EJE.

2 MOTIVATION

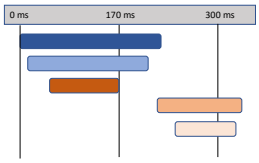


Figure 1: Swimlane visualization.

The swimlane visualisation is the canonical way to visualize individual requests within distributed tracing tools [9, 19, 37, 39]. Fig. 1 shows a representative example of a swimlane visualization. The visualization depicts a timeline of a single request, with RPCs depicted horizontally and sorted vertically to highlight their relationships. This type of visualization proves highly beneficial for performance analysis of individual requests, allowing for a detailed investigation into how each RPC affects the overall end-to-end response time.

However, these visualizations exhibit certain limitations when it comes to conducting more complex performance analyses. For instance, observing the performance of individual end-to-end requests might result in misleading insights if not contextualized appropriately [10]. Indeed, a request's response time can only be deemed anomalous when compared with other requests of the same type [2]. Additionally, engineers are often more inclined to investigate recurrent response time trends rather than focusing on the performance of individual requests [32]. Diverse end-to-end response time behaviors may be associated with specific request characteristics, such as particular RPC execution paths or RPC performance behaviors. Consequently, engineers may wish to identify these characteristics to uncover potential performance issues, and gather a more comprehensive picture of the system performance [8, 20, 32].

Distributed tracing tools currently lack sufficient support for this type of analysis, which often necessitates the concurrent use of multiple visualizations and tools, such as Jaeger [39] and Kibana [13] [10]. A naive strategy involves initially recognizing repetitive performance behaviors for further investigation, followed by the examination of individual requests to characterize relevant performance behaviors. This can be accomplished by detecting "modes" within the end-to-end response time distribution (for instance, using Kibana), which represent meaningful recurring performance behaviors. Following this, samples of requests associated with each mode can be extracted and examined individually (for instance, using Jaeger's swimlanes) to identify distinct characteristics that contribute to specific performance behaviors or modes.

However, this method can be particularly laborious as it requires manual inspection and comparison of multiple requests across diverse visualizations and tools. Moreover, even when the method is successful, it may not provide a satisfactory level of confidence. Indeed, determining the specific characteristics associated with a particular distribution mode necessitates verifying that these characteristics appear *exclusively* in requests that exhibit this particular end-to-end response time behavior. This task can be challenging by using current tools.

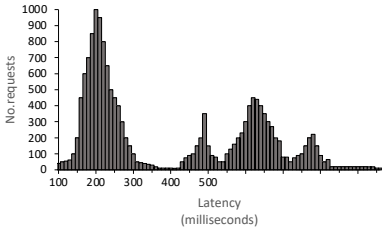
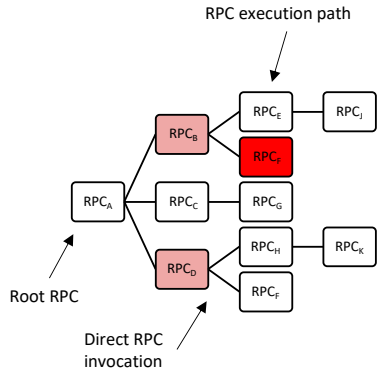


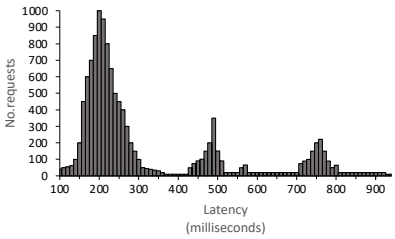
Figure 2: End-to-end response time distribution.

Consider the scenario illustrated in Fig. 2, which represents the distribution of end-to-end response times for a specific type of request, such as loading a website homepage. As can be observed in the figure, requests demonstrate four distinct response time behaviors, *i.e.,* modes. Suppose that the rightmost mode is characterized by a unique request characteristic, specifically an RPC that exhibits slower execution time[1] . That is, this specific RPC shows increased execution time in all requests belonging to the rightmost mode (*e.g.,* due to an expensive task), but not in others. With the current distributed tracing tools, identifying patterns like this can be particularly challenging. Current distributed tracing tools lack targeted methods to simplify the analysis of RPC attributes, such as execution time, and their relationship with end-to-end response time.

[1]Henceforth, the term *execution time* will be used to denote the response time of a generic RPC. Conversely, *end-to-end response time* will refer to the response time of the root RPC that triggers all subsequent RPC invocations.



(a) Tree



(b) Histogram

Figure 3: Visual Components

3 VAMP

VAMP aims to enhance performance analysis of microservices systems by simplifying the investigation of attributes pertaining to specific RPC and their relationship with end-to-end response time. In this section, we first introduce the core insights that underpin VAMP, its primary visual components, and the interaction modality. Then, we describe how these visual components fit within the VAMP dashboard, and detail the VAMP architecture and implementation.

3.1 Visual Components

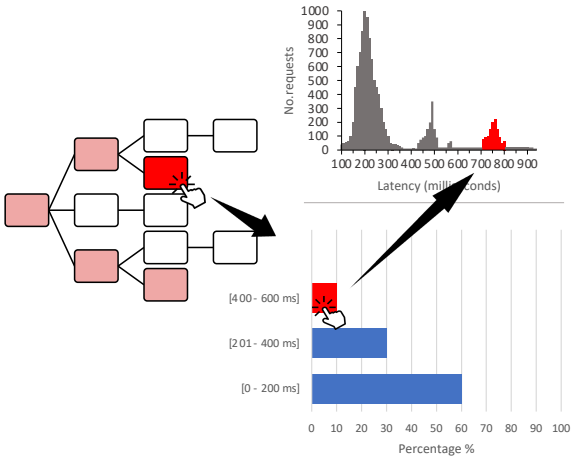
The core insight behind VAMP is to make explicit the relationship between RPC attribute values and end-to-end response time. RPC attributes could refer to several aspects, such as the *frequency* of RPC invocation within a request, or the associated *execution time*. VAMP leverages two main interactive components to highlight this relationship: a *tree* and a *histogram*. The tree provides an aggregated view of the requests' workflows in terms of RPC invocations, while the histogram displays a traditional distribution plot of the end-to-end response time. Users can interact with the tree to examine how specific attribute values, related to a particular RPC execution path, influence the end-to-end response time; we refer to this as *forward analysis*. Conversely, starting from the histogram, users can investigate how specific end-to-end response time behaviors are associated with certain RPC attribute values; this is referred to as *backward analysis*. In the following, we will first describe in detail the characteristics of these two main visual components. Then, in the subsequent subsection, we will detail the interaction modality of VAMP.

3.1.1 Tree. This visualization component takes inspiration from the Jaeger comparison tool [15], which allows users to compare two end-to-end requests and highlight their structural differences. We have redesigned this approach by extending its capabilities beyond the comparison of two requests, thereby allowing aggregated analysis of multiple end-to-end requests. In a nutshell, the VAMP tree provides an aggregated view of the RPC workflows performed by a set of end-to-end requests, as shown in Fig. 3a. Each node of the tree represents a RPC invocation within a specific execution path, where the leftmost node represents the root RPC, and edges indicate direct RPC invocation. For instance, in Fig. 3a the node labeled as RPC_E represents the execution path $\mathsf{RPC}_A \rightarrow \mathsf{RPC}_B \rightarrow \mathsf{RPC}_E$. As can be observed by the figure, the same RPC can appear in multiple nodes (*e.g.,* RPC_F), as it can be invoked within multiple different execution paths. A RPC execution path will appear in the tree if and only if it is present in at least one of the requests being analyzed. It is worth noting that when a particular RPC invokes the same RPC multiple times, this leads to a single node in the tree. In other words, if the RPC_A invokes the RPC_D multiple times, there will be only one child node referring to RPC_D.

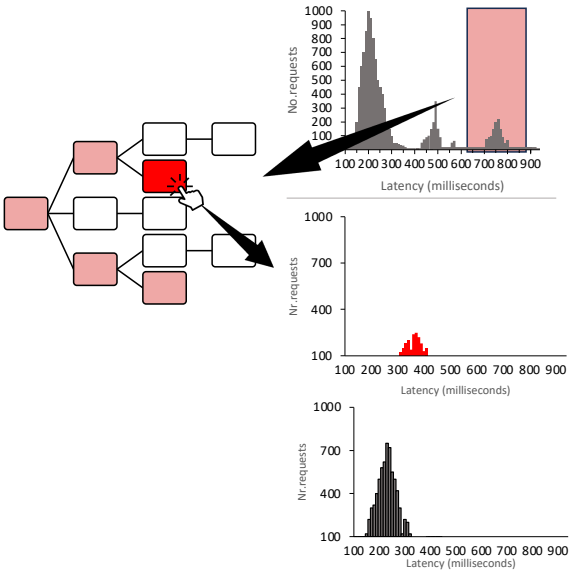
VAMP utilizes color encoding to highlight RPC execution paths that are worthy of investigation based on their attribute values. It currently supports the analysis of two kinds of attributes: *execution time* and *frequency*. The first one denotes the (average) execution time of the RPC within a specific execution path in each request, while second one indicates the path frequency, *i.e.,* how many times it occurs within each request. We use color encoding to emphasize RPC execution paths with higher variance in their attributes. The key intuition here is that RPC execution paths showing higher variance in their attributes are likely to manifest different behaviors that can potentially affect the end-to-end response time. For instance, a higher frequency of a particular RPC invocation within a request could result in a longer end-to-end response time. Or similarly, a slower RPC execution time may correspond to a prolonged end-to-end response time.

We employ a continuous color scale to depict the variability in the attribute values associated. This scale is based on the Coefficient of Variation (CV)[14], *i.e.,* a standardized measure of dispersion that is defined as the ratio of the standard deviation to the mean. As execution times in distributed systems are well known to be subject to long tails [11], when dealing with this attribute, we apply outlier filtering by removing execution times values greater than the 99^{th} percentile. A CV of 0 results in a white node, indicating no variability. On the other hand, a CV greater than or equal to 1 results in a red node, suggesting a high variability in the attribute values. The shade of color gradually transitions from white to red as the CV value increases.

3.1.2 Histogram. The VAMP histogram component (shown in Fig. 3b) depicts a traditional distribution plot of the end-to-end response time. These kinds of visualizations are frequently used in practice for performance analysis, and are provided by several tools, *e.g.,* Kibana [13]. According to recent research [10], understanding the distribution of end-to-end response times stands as core activity in modern performance analysis practice. The histogram component provided by VAMP aims to facilitate this process by supporting



(a) Forward analysis



(b) Backward analysis

Figure 4: Interaction modalities

the identification of specific performance behaviors that are worthy of investigation. The user can visually identify "modes" in the response time distribution, which indicate meaningful recurring performance behaviors, to start a targeted investigation on these requests, as we will detail in the subsequent subsection.

3.2 Interaction modalities

VAMP supports bidirectional analysis, allowing users to initiate their analysis from either the tree (*forward analysis*) or the histogram (*backward analysis*). In the following, we provide detailed descriptions of both these interaction modalities.

3.2.1 Forward analysis. Fig. 4a depicts an illustrative example of forward analysis. By examining the tree, the user can identify "suspicious" RPC execution paths that exhibit high variability in the corresponding attribute values. For instance, when analyzing the execution time attributes, the user can identify RPCs that show highly varying execution times, and, by clicking on the corresponding node, they can inspect the recurring execution time behaviors associated with the path, displayed in the form of a bar chart, as shown in Fig. 4a. Each bar refers to a specific execution time range (see y-axis labels), and it shows the percentage of requests with RPC execution time falling in that range. In order to identify meaningful ranges, we employ a widely-used clustering algorithm, namely K-means [26]. In particular, we run the algorithm on-the-fly after the user click with k ranging from 2 to 5 and we select the results showing the highest silhouette score [33]. Each bar represents a meaningful recurring execution time behavior, and the user can click on each bar to see how this behavior reflects in the end-to-end response time. This relation is shown by highlighting in red the area of the distribution that shows this particular RPC execution time behavior. For instance, in Fig. 4a, we can observe that when the selected RPC has an execution time ranging between 400 and 600 milliseconds, it can lead to end-to-end response times that range between 700 and 800 milliseconds. Understanding these kinds of relationships would have been way more challenging by using currently available tools. It is worth noticing that the same interaction modality also applies when analyzing different RPC attributes, such as *occurences*.

3.2.2 Backward analysis. In the backward analysis, the user can start its investigation directly from the histogram component. The user selects a specific range of end-to-end response time using a slider selector, as shown in Fig. 4b. This selection triggers an update in the tree component's color scheme, shifting its semantic from variability to divergence. In other words, the updated color scheme will now denote the degree of divergence in the attribute values of the selected set of requests (*i.e.,* those that show end-to-end response time in the selected range) when compared to those in other requests. A red node indicates that the corresponding RPC execution path shows considerably different attribute values in the selected requests when compared to other requests, suggesting a possible relationship between the selected end-to-end response time and the RPC execution path. Conversely, a white node indicates similar attribute values, and therefore a weak relation. We quantify the degree of divergence using Kullback-Leibler divergence [7], where values close to 0 indicate nearly identical distributions (white), while values close to or higher than 1 indicate highly different distributions (red).

The user can then delve deeper into each RPC execution time behavior by clicking on the corresponding node. This action lets appear at screen two new histograms (in the bottom right corner) representing the distributions of the execution time in the selected RPC execution path, respectively in the selected requests (in red) and in other requests (in grey). In doing so, the user can effectively analyze how particular ranges of the end-to-end response time distribution correlate with specific RPC attribute values.

3.3 Dashboard

Fig. 5 outlines the VAMP dashboard. As can be observed by the figure the two main visual components, namely the tree and the histogram, are positioned in the center-left and in the upper-right

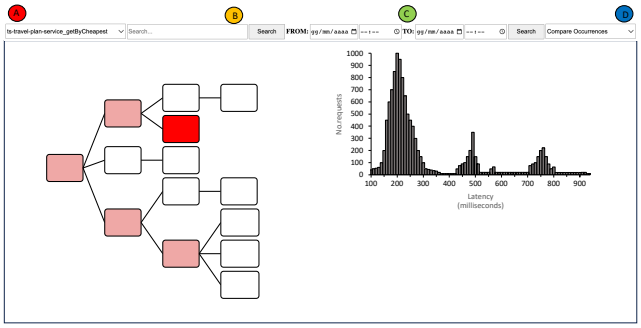


Figure 5: VAMP's Dashboard

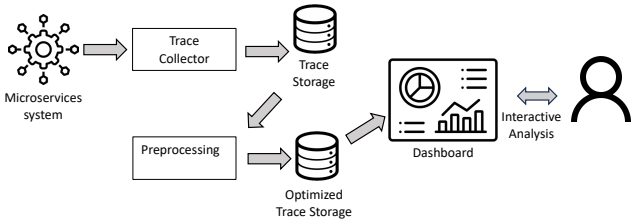


Figure 6: VAMP's Workflow

corners, respectively. The space in the bottom-right is intentionally left blank and will be used to display supplementary visualization components during the interaction, *e.g.,* the bar chart (for forward analysis) and the two histograms (for backward analysis).

It's worth noting that VAMP is specifically designed to assist in analyzing requests from the same class, *i.e.,* those originating from the same root RPC. As part of this process, the user is required to first select the root RPC and the RPC attribute (*i.e.,* execution time or path frequency) to be investigated, before proceeding with the actual analysis.

The user can select the root RPC using either a dropdown menu (A) or a search text-box (B). Similarly, the RPC attribute (execution time or frequency) to be analyzed can be selected using a dropdown menu (D). Additionally, the dashboard includes a date-time range selector (C), where the user can specify the start and end date-times. This feature allows for analyses at different time granularities (*e.g.,* monthly, weekly, and daily) or over specific time ranges known to include system anomalies.

To enhance user experience during the interaction with the tool, VAMP supports pinch gestures to enable zoom in and zoom out of the tree. In addition, it allows the user to hide the RPCs invoked within a particular execution path by double-clicking on the related node.

3.4 Architecture and Implementation

Fig. 6 outlines the key architecture components of VAMP. The *Trace Collector* (*i.e.,* Jaeger [39]) continuously collects traces from the microservices system and stores them in a *Trace Storage* (*i.e.,* Elasticsearch [12]). Given the large volume of data collected each day, we have devised a *Preprocessing* step to enhance the efficiency of interaction with VAMP. This preprocessing step operates in batches and is intended to be executed periodically (*e.g.,* hourly or daily). For each end-to-end request (*i.e.,* trace), VAMP recursively reconstructs all the involved RPC execution paths, along with their attribute values (namely, execution times and frequencies), and stores them in an *Optimized Trace Storage* based on MongoDB. Each path associated with a request is stored as a separate document in a dedicated MongoDB collection, and includes: the name of the path, the trace ID, the number of occurrences of the path in the trace, the observed execution time, the timestamp, and the name of the root RPC. Similarly, VAMP stores the end-to-end response time values, along with related information, in a separate MongoDB collection. This information includes the RPC root, the trace ID, the response time value, and the timestamp. This data reorganization allows for greater flexibility in easily and efficiently querying the data needed for the VAMP dashboard to function properly. As can be seen from Fig. 6, the *Dashboard* app directly queries the *Optimized Trace Storage* to efficiently generate visualizations.

VAMP currently supports distributed traces stored in the Jaeger [39] format using Elasticsearch [12] as *Trace Storage*, but it can be easily extended to other technologies. The dashboard and visual components have been developed using D3.js, which handles the visualization rendering, and Flask, which serves as the backend service. The preprocessing scripts are implemented in Python.

4 EVALUATION

The conducted evaluation is centered around one main research question: *To what extent does VAMP support performance analysis?* We want to understand whether VAMP can be successfully utilized to gain insights about the relationship between request attributes and end-to-end performance response time.

In the following, we first describe the methodology used to gather the answer. Then, we report and discuss the results of the experimental evaluation. Finally, we describe the threats to validity of our study.

4.1 Methodology

To achieve our study *goal*, we generate 33 datasets of distributed traces, where each dataset reflects a distinct scenario that induces a specific variation in the relationships between request attributes and end-to-end response time. Subsequently, we manually analyze each dataset using VAMP to evaluate the effectiveness of our tool in highlighting these relationships.

4.1.1 Datasets generation. The 33 datasets are generated from TrainTicket [47], which, as best as we know at the time of writing, is the largest and most complex open-source microservice-based system. TrainTicket provides a typical train ticket booking web service; it involves 41 microservices implemented in four programming languages, and it utilizes Jaeger [39] and Elasticsearch [12] for collecting and storing distributed traces. We have chosen TrainTicket as a representative case system due to its complexity and because it has been recently used in software engineering research [23, 41, 46, 47].

Each dataset of our study contains distributed traces related to one specific *root RPC* of the system, which are stored on Elasticsearch using the standard Jaeger format.

To simulate different scenarios that induce different variations in the relationship between RPC attributes and end-to-end response time, we rely on two different approaches: (i) we inject synthetic performance issues in specific RPCs to increase the overall end-to-end response time, and (ii) we use complex mixtures of varying workloads that may alter the relationships between RPC execution time/occurrences and end-to-end response time. These two distinct approaches lead to the generation of two categories of datasets.

The first category of datasets is generated using a methodology similar to the one presented in [8, 41]. Initially, the system's source code is modified to inject random performance issues. Following this, load-testing sessions are run to simulate user interactions with the system and generate distributed traces. Each injected performance issue affects approximately 10% of requests, introducing a delay into one specific RPC.

To generate a dataset, we first select two random RPCs that will be impacted by the performance issues. Subsequently, we choose a random delay to increase the end-to-end response time by x%, where $x \in \{10, 20, 30\}$. In addition, in half of the datasets, we inject a random delay of y% (with $y \in \{10, 20, 30\}$) into an asynchronous RPC, which does not produce any effect on the end-to-end response time. This is a common practice used to test the robustness of pattern detection approaches in the context of microservices systems [8, 20, 41]. After modifying the system accordingly, we conduct load-testing sessions to generate the distributed trace datasets. Each load testing session involves 20 synthetic users, simulated by Locust [18]. Each user makes a request to the system and randomly waits between 1 and 3 seconds before making the next request. Each session lasts for 20 minutes. Using this methodology, we generate 20 datasets featuring various combinations of performance issues that affect different RPCs with different delays. For a more detailed explanation of this process, we refer readers to the work of Traini and Cortellessa [41]. Due to space constraints, we do not elaborate further here.

The second kind of dataset does not involve any performance issue injection, but it is generated using a more elaborate workload generator. Similarly to recent studies [24, 25] we use load mixtures that involve multiple types of simulated users (*i.e.,* load drivers), where each user type performs different classes of requests on the system. For example, some types of users may only visit the homepage and subsequently search trains for some random locations, while others first login into the system and then book random tickets. Besides this, we also ensure that the number of simulated users per type keeps changing over time. In this way, workloads will more closely resemble real-world ones, as they generate mixtures of different classes of requests that change over time [3]. To this aim, we slightly modified *PPTAM* [4], a workload generator that involves 5 different user types, to continuously change the number of users of each type at run-time. Overall, the number of simultaneous users ranges from a minimum of 20 to a maximum of 31, and the load-testing session lasts for 1 hour. The workload fluctuations over time are randomly generated upfront. This process leads to 13 distinct datasets, each one related to a different API.

The generations of the datasets were done on a bare-metal machine running Linux Ubuntu 18.04.2 LTS on a dual Intel Xeon CPU E5-2650 v3 at 2.30 GHz, with a total of 40 cores and 80 GB of RAM. All non-mandatory background processes except SSH are disabled, and we ensured that no other users interacted with the dedicated machine during our experiments.

To enhance clarity throughout the rest of the article, we will use specific notations for different categories of datasets. Datasets characterized by performance issues (*i.e.,* first category) will be referred to as $\widehat{D}_i$, where $1 \leq i \leq 20$. Conversely, datasets free from performance issues (*i.e.,* second category) will be denoted as D_i, with $1 \leq i \leq 13$.

4.1.2 Manual analysis. To assess the effectiveness of our approach, two authors conducted manual inspections of the 33 distributed trace datasets using VAMP. Our evaluation focused on determining the extent to which VAMP facilitated the comprehension of the relationship between RPC execution time/frequency and end-to-end response time.

It is worth noting that neither author was aware of the specific performance issues or the workload variations present in each dataset. This is because the process for the dataset generation, including performance issue injections and load testing modifications, was entirely random and automated. Nonetheless, both authors were familiar with the TrainTicket system before the study.

4.2 Results

VAMP has proven to be effective in highlighting the relationship between RPC execution time and end-to-end response time, throughout all the datasets featuring injected performance issues. The analysis was straightforward for the majority of the datasets (18 out of 20), demanding minimal interaction with VAMP. In these datasets, both *forward* and *backward analysis* demonstrated comparable effectiveness, with no noticeable difference in the effort needed to understand these relationships. Due to space constraints, we are unable to present the exhaustive results of our analyses across all datasets. However, we have included a selection of representative examples that underscore both the utility and potential challenges associated with employing VAMP. Additionally, for the sake of completeness, we have made available screenshots capturing interactions with VAMP across all the datasets in a supplementary replication package [44].

Fig. 7 showcases an example of *forward analysis* using the dataset $\widehat{D}_2$. As depicted in the left screenshot, VAMP significantly streamlines the identification of the two RPCs impacted by performance issues, *i.e.,* the ones highlighted in bright red. Following this, the user can select these nodes to investigate correlations between specific RPC execution times and end-to-end response times. For example, the screenshot on the right of Fig. 7 reveals that the selected RPC execution path (highlighted in green) exhibits two distinct execution time behaviors: in 9.87% of the requests, the RPC `getRouteByTripId` has an execution time ranging from 27.46 to 33.67 milliseconds, and in the remaining 90.13% of requests, the execution time ranges from 2.62 to 11.25 milliseconds. This screenshot displays the view of VAMP during the investigation of the first behavior, that is, after clicking on the corresponding bar (highlighted in red). As evident from the figure, VAMP reveals that all the requests with an execution time ranging from 27.46 to 33.67 milliseconds in the selected RPC execution path fall within a specific region of the end-to-end response time distribution, as shown by the red highlight in the histogram. Understanding these kinds of relationships would have

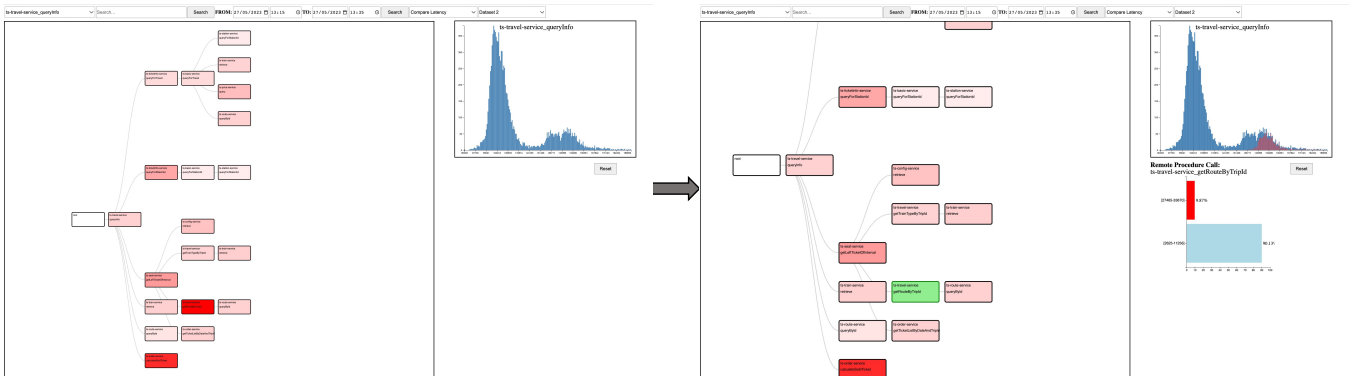


Figure 7: *Forward analysis* on *execution time* for dataset $\widehat{D}_2$.

been particularly challenging when using traditional performance analysis tools.

Fig. 8 offers another example of how VAMP allows users to rapidly identify the RPC responsible for a particular end-to-end response time deviation. Specifically, this figure demonstrates an instance of a VAMP *backward analysis* using the dataset $\widehat{D}_9$. The left screenshot shows that by selecting a specific range of end-to-end response times, the user can immediately pinpoint the RPC execution paths that display significantly divergent behavior in the execution time (highlighted in bright red). The screenshot on the right displays the investigation of one of these nodes (*i.e.,* the one highlighted in green), illustrating how VAMP assists users in comprehending the correlation between specific RPC execution times and the selected range of end-to-end response times. For instance, it is noticeable that when the RPC `getRouteByTripId` has an execution time exceeding 27 milliseconds, it results in an end-to-end response time that falls within the range of 137 and 168 milliseconds.

Another interesting aspect of VAMP is its ability to identify execution time fluctuations in RPCs that do not have influence on the end-to-end response time. For instance, Fig. 9 illustrates two distinct execution time behaviors in the selected RPC `calculateSoldTicket`: one ranging between 33.42 and 55.95 milliseconds, and another between 1.03 and 14.96 milliseconds. Through the use of VAMP, we were able to easily notice the lack of correlation between the execution time of this RPC and the end-to-end response time. As illustrated in Figures 9a and 9b, the selected execution time behaviors (*i.e.,* the bars highlighted red) are evenly distributed across the end-to-end response time distribution, implying a lack of notable correlation with specific regions of the end-to-end response time. This indicates that even if the RPC execution time varies drastically from one request to another, it does not have any significant impact on the end-to-end response time.

In two datasets, specifically $\widehat{D}_4$ and $\widehat{D}_{19}$, the analysis was more complex, requiring a higher number of interactions with VAMP. The peculiarity of these datasets was that the two performance issues led to an increased end-to-end response time that overlaps within the same range. This made the connection between the RPC execution time and end-to-end response time more challenging to understand. We did not include the specific details of these cases in the paper because of limited space, but we refer the reader to our supplementary materials [44] for the related screenshots.

With regards to the 13 datasets in the second category, we found that a substantial majority of them - precisely 11 datasets - feature a unique mode in the end-to-end response time distribution. Given the objectives of our analysis, these cases were not considered. Consequently, we used two datasets for our evaluation. The first dataset, D_1, consists of requests originating from the root RPC `getByCheapest`, while the second dataset, D_2, comprises requests initiated from `queryInfo`. VAMP enabled us to characterize the correlation between the frequency of each RPC execution path and specific modes of the end-to-end response time. Fig. 10 provides an example of this characterization, illustrating that each mode of the end-to-end distribution corresponds to a specific number of invocations of a selected RPC execution path (highlighted in green). For example, as depicted in Fig. 10c, the right-most mode is characterized by 14 invocations of the path `queryInfo` ⟶ `queryForStationId`. Similarly, the center mode is characterized by 6 invocations of this path, while the left-most mode is marked by 2 invocations. Uncovering such patterns using traditional observability tools would have been notably challenging.

Summing up, we answer our RQ as follows: VAMP proved to be effective in supporting performance analysis of microservices. In 18 out of the 20 datasets involving performance issues, we were able to rapidly identify the affected RPCs, their corresponding execution time behaviors, and their relationship with end-to-end response time. However, in a few specific cases (2 out of 20 datasets), the analysis proved to be more challenging, necessitating a greater number of interactions with VAMP. Moreover, our evaluation demonstrates how VAMP can facilitate an understanding of how structural differences in requests (*i.e.,* varying frequencies of RPC execution paths) influence end-to-end response time.

4.3 Threats to Validity

4.3.1 Construct validity. We conducted the evaluation in-house rather than using external participants. Our familiarity with the tool and the experimental setup could potentially introduce bias into the evaluation outcomes. To mitigate this threat, we generated 20 diverse scenarios randomly, each involving various combinations of performance issues. Additionally, the two authors who

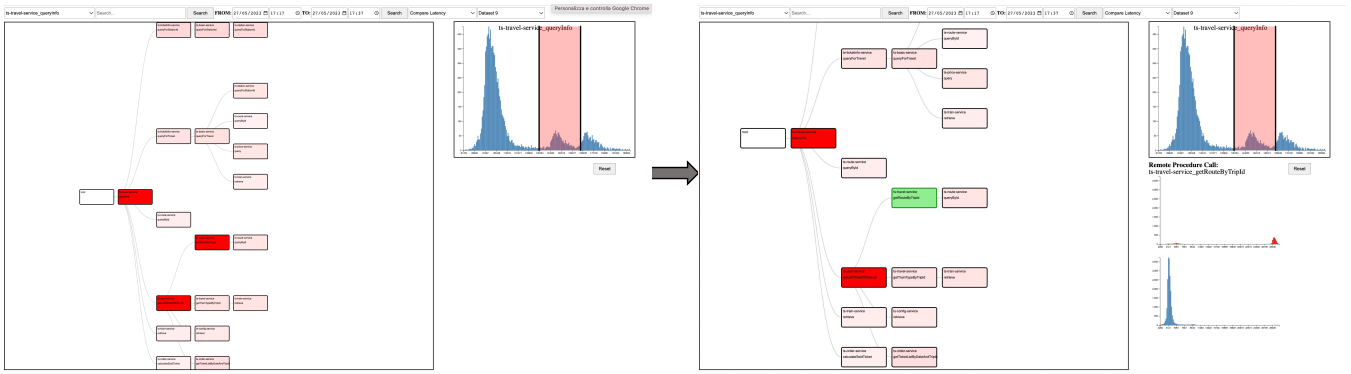
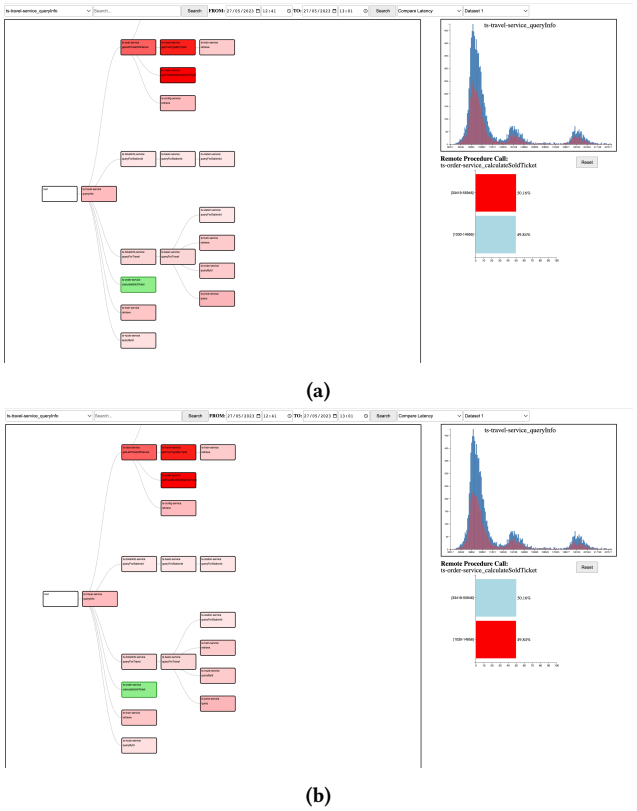


Figure 8: *Backward analysis* on *execution time* for dataset $\widehat{D}_9$.



(a)

(b)

Figure 9: *Forward analysis* on *execution time* for dataset $\widehat{D}_1$.

conducted the evaluations were kept uninformed about the RPCs affected by the performance issues. Using artificial delays as part of the evaluation may not perfectly mirror real-world performance issues. However, this methodology aligns with prevailing practices in software engineering research, as evidenced by several studies [21, 25, 27, 41]. Furthermore, in contrast to many previous studies [21, 25, 27], which typically employ a limited set of predetermined regressions with fixed magnitudes, our approach offers a more comprehensive evaluation on 20 diverse scenarios involving different combinations of RPCs and delay magnitudes

4.3.2 Internal validity. The workloads used in our experimental setup may not be representative of real-world workloads. To (partially) mitigate this limitation, we perform an additional analysis using mixtures of continuously changing workloads, generated through PPTAM [4]. Our evaluation may be subject to confirmation bias, wherein the authors may unconsciously confirm their pre-existing beliefs on the effectiveness of VAMP. Nonetheless, the results obtained using VAMP unambiguously demonstrate its effectiveness across a majority of the datasets evaluated. In the interest of transparency and to enable readers to independently assess this evidence, we have made all screenshots documenting the use of VAMP across the datasets in our study publicly available [44].

4.3.3 External validity. We cannot ensure that VAMP can achieve the same effectiveness on other datasets outside our experimental setup (*e.g.*, real world scenarios). Nevertheless, through an evaluation on 33 datasets, we have demonstrated that our approach effectively aids in the performance analysis of microservices systems. VAMP's efficiency was evaluated on datasets of varying sizes, ranging from 11181 to 22348 requests. It's worth noting that real-world microservices systems may involve a much larger volume of requests. As part of our future work, we plan to enhance the scalability of VAMP by incorporating sampling techniques and optimizing preprocessing procedures.

5 RELATED WORK

Previous research on visualization for distributed systems has primarily focused on analyzing individual requests or comparing two requests. The swimlane visualization, a widely used technique to represent individual end-to-end executions, was originally proposed by Singelman *et al.* [37]. Today, most distributed tracing tools offer this visualization. TraVista [2] enhances the standard swimlane visualization by augmenting it with information that assists users in contextualizing the performance of the analyzed request in relation to others. Beschastnikh *et al.* [6] introduced

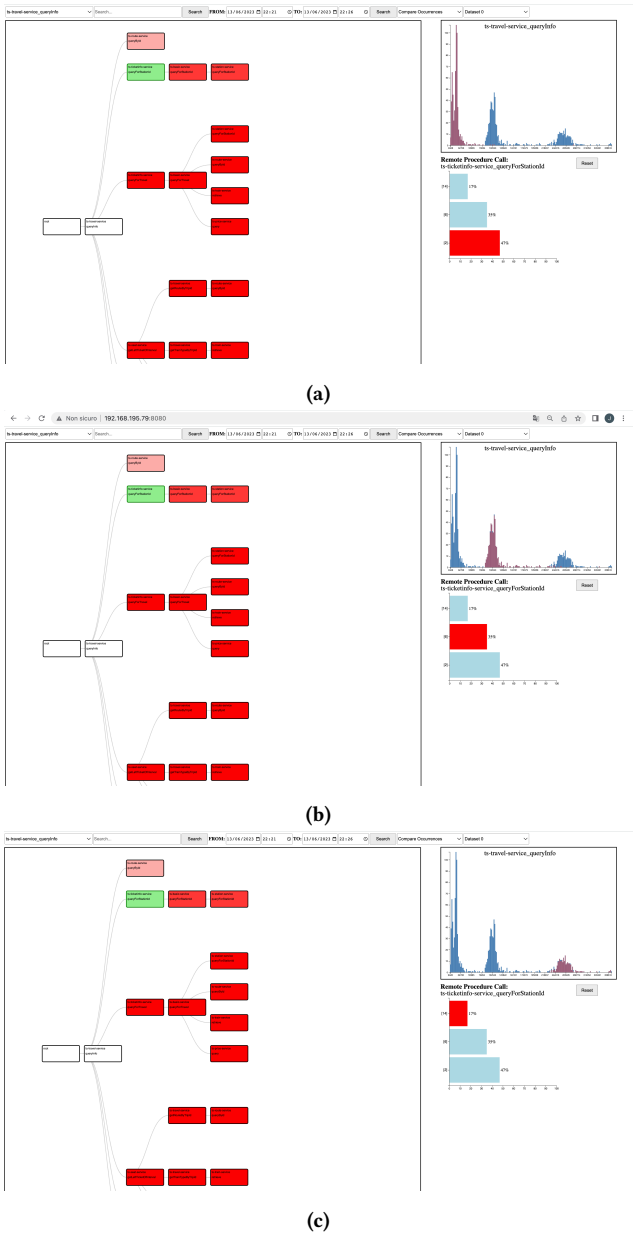


Figure 10: *Forward analysis* on *frequency* for dataset D_2.

a novel visualization tool called ShiViz, which includes an interactive time-space diagram for visualizing individual end-to-end executions of a distributed system.

Sambasivan et al. [36] studied and compared three visualization approaches (*i.e.,* side-by-side view, difference view, and animation) for comparing two request-flow traces. Jaeger [39] provides a feature to visually compare the structural characteristics of two requests [15].

Several visualizations have also been introduced to analyze the performance behaviors of multiple end-to-end requests in aggregate. One example is TransVis by Beck *et al.* [5], which provides a visualization technique for specifying and analyzing transient performance behaviors in microservice systems. Other examples of visual techniques for aggregate performance analysis can be found in commercial APM tools [1], such as Dynatrace, AppDynamics, or Instana. For instance, Dynatrace's Service Flow feature [16] allows to display aggregate workflows of end-to-end requests along with their associated characteristics.

To the best of our knowledge, despite the many existing visualization techniques for microservices performance analysis, there is still a lack of dedicated visualizations to analyze the correlation between requests' attributes and their end-to-end performance behavior, which is the goal of our study.

Other related works include the recent Davidson and Mace's survey [9], which underscores the critical role of visualization within systems research, and the qualitative interview study conducted by Davidson *et al.* [10], which highlighted the limitations of current distributed tracing tools. Davidson *et al.*'s findings involved several open research challenges that span multiple research areas, including visualization research.

6 CONCLUSION

In this paper, we presented VAMP, a novel visual analytics tool for microservices performance analysis. VAMP overcomes the limitations of current distributed tracing tools by providing a wide set of interactive visualizations that enables effective performance analysis of multiple end-to-end requests. Through an evaluation of 33 datasets generated from an established open-source microservices system, we demonstrate how VAMP can be effectively used to understand the relationship between the RPC attributes and end-to-end response time. For future work, we plan to enhance the efficiency of our tool to facilitate its transition to practice. As part of this process, we intend to validate our future improvements using real-world distributed traces from large-scale microservices systems, similar to those shared by Alibaba [27]. To aid reproducibility we provide the data and source code needed to replicate our findings [44].

ACKNOWLEDGMENTS

This work is partially supported by European Union - NextGenerationEU - National Recovery and Resilience Plan (Piano Nazionale di Ripresa e Resilienza, PNRR) - Project: "SoBigData.it - Strengthening the Italian RI for Social Mining and Big Data Analytics" - Prot. IR0000013 - Avviso n. 3264 del 28/12/2021, and by Territori Aperti (a project funded by Fondo Territori, Lavoro e Conoscenza CGIL CISL UIL).

REFERENCES

[1] Tarek M. Ahmed, Cor-Paul Bezemer, Tse-Hsun Chen, Ahmed E. Hassan, and Weiyi Shang. 2016. Studying the Effectiveness of Application Performance Management (APM) Tools for Detecting Performance Regressions for Web Applications: An Experience Report. In *Proceedings of the 13th International Conference on Mining Software Repositories* (Austin, Texas) *(MSR '16)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/2901739.2901774

[2] Vaastav Anand, Matheus Stolet, Thomas Davidson, Ivan Beschastnikh, Tamara Munzner, and Jonathan Mace. 2020. Aggregate-Driven Trace Visualizations for Performance Debugging. arXiv: 2010.13681 [cs] Number: arXiv:2010.13681.

[3] Dan Ardelean, Amer Diwan, and Chandra Erdman. 2018. Performance Analysis of Cloud Applications. In *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation (NSDI'18)*. USENIX Association, USA, 405–417. event-place: Renton, WA, USA.

[4] Alberto Avritzer, Daniel Menasché, Vilc Rufino, Barbara Russo, Andrea Janes, Vincenzo Ferme, André van Hoorn, and Henning Schulz. 2019. PPTAM: Production and Performance Testing Based Application Monitoring. In *Companion of the 2019 ACM/SPEC International Conference on Performance Engineering* (Mumbai, India) *(ICPE '19)*. Association for Computing Machinery, New York, NY, USA, 39–40. https://doi.org/10.1145/3302541.3311961

[5] Samuel Beck, Sebastian Frank, Alireza Hakamian, Leonel Merino, and André van Hoorn. 2021. TransVis: Using visualizations and chatbots for supporting transient behavior in microservice systems. In *2021 Working Conference on Software Visualization (VISSOFT)*. IEEE, 65–75.

[6] Ivan Beschastnikh, Perry Liu, Albert Xing, Patty Wang, Yuriy Brun, and Michael D. Ernst. 2020. Visualizing Distributed System Executions. *ACM Trans. Softw. Eng. Methodol.* 29, 2 (March 2020). https://doi.org/10.1145/3375633 Place: New York, NY, USA Publisher: Association for Computing Machinery.

[7] Stephen Boyd and Lieven Vandenberghe. 2004. *Convex Optimization*. Cambridge University Press. https://doi.org/10.1017/CBO9780511804441

[8] Vittorio Cortellessa and Luca Traini. 2020. Detecting Latency Degradation Patterns in Service-Based Systems. In *Proceedings of the ACM/SPEC International Conference on Performance Engineering (ICPE '20)*. Association for Computing Machinery, New York, NY, USA, 161–172. https://doi.org/10.1145/3358960.3379126

[9] Thomas Davidson and Jonathan Mace. 2022. See it to believe it? The role of visualisation in systems research. In *Proceedings of the 13th Symposium on Cloud Computing (SoCC '22)*. Association for Computing Machinery, New York, NY, USA, 419–428. https://doi.org/10.1145/3542929.3563488

[10] Thomas Davidson, Emily Wall, and Jonathan Mace. 2023. A Qualitative Interview Study of Distributed Tracing Visualisation: A Characterisation of Challenges and Opportunities. *IEEE Transactions on Visualization and Computer Graphics* (2023), 1–12. https://doi.org/10.1109/TVCG.2023.3241596

[11] Jeffrey Dean and Luiz André Barroso. 2013. The Tail at Scale. *Commun. ACM* 56 (2013), 74–80. http://cacm.acm.org/magazines/2013/2/160173-the-tail-at-scale/fulltext

[12] Elastic. 2023. Elasticsearch: The Official Distributed Search & Analytics Engine. https://www.elastic.co/elasticsearch/ "Accessed 2023-01-15 18:38".

[13] Elastic. 2023. Kibana: Explore, Visualize, Discover Data. https://www.elastic.co/kibana "Accessed 2023-01-15 17:38".

[14] Brian Everitt. 1998. *The Cambridge Dictionary of Statistics*. Cambridge University Press, Cambridge, UK.

[15] Joe Farro. 2018. Trace comparisons arrive in Jaeger 1.7. https://medium.com/jaegertracing/trace-comparisons-arrive-in-jaeger-1-7-a97ad5e2d05d "Accessed 2023-01-15 18:16".

[16] Dynatrace Inc. 2023. Dynatrace. Service Flow. https://www.dynatrace.com/platform/service-flow/ "Accessed 2023-02-14 14:10:59".

[17] Zhen Ming Jiang and Ahmed E. Hassan. 2015. A Survey on Load Testing of Large-Scale Software Systems. *IEEE Transactions on Software Engineering* 41, 11 (2015), 1091–1118. https://doi.org/10.1109/TSE.2015.2445340

[18] Heyman Jonatan, Carl Byström, Joakim Hamrén, and Hugo Heyman. 2023. An open source load testing tool. https://locust.io "Accessed 2023-06-10 13:38".

[19] Jonathan Kaldor, Jonathan Mace, Micha\l Bejda, Edison Gao, Wiktor Kuropatwa, Joe O'Neill, Kian Win Ong, Bill Schaller, Pingjia Shan, Brendan Viscomi, Vinod Venkataraman, Kaushik Veeraraghavan, and Yee Jiun Song. 2017. Canopy: An End-to-End Performance Tracing And Analysis System. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 34–50. https://doi.org/10.1145/3132747.3132749

[20] Darja Krushevskaja and Mark Sandler. 2013. Understanding Latency Variations of Black Box Services. In *Proceedings of the 22nd International Conference on World Wide Web* (Rio de Janeiro, Brazil) *(WWW '13)*. Association for Computing Machinery, New York, NY, USA, 703–714. https://doi.org/10.1145/2488388.2488450

[21] Christoph Laaber and Philipp Leitner. 2018. An Evaluation of Open-Source Software Microbenchmark Suites for Continuous Performance Assessment. In *Proceedings of the 15th International Conference on Mining Software Repositories* (Gothenburg, Sweden) *(MSR '18)*. Association for Computing Machinery, New York, NY, USA, 119–130. https://doi.org/10.1145/3196398.3196407

[22] Jessica Leone and Luca Traini. 2023. Enhancing Trace Visualizations for Microservices Performance Analysis. In *Companion of the 2023 ACM/SPEC International Conference on Performance Engineering* (Coimbra, Portugal) *(ICPE '23 Companion)*. Association for Computing Machinery, New York, NY, USA, 283–287. https://doi.org/10.1145/3578245.3584729

[23] Bowen Li, Xin Peng, Qilin Xiang, Hanzhang Wang, Tao Xie, Jun Sun, and Xuanzhe Liu. 2021. Enjoy your observability: an industrial survey of microservice tracing and analysis. *Empirical Software Engineering* 27, 1 (2021), 25. https://doi.org/10.1007/s10664-021-10063-9

[24] Lizhi Liao, Jinfu Chen, Heng Li, Yi Zeng, Weiyi Shang, Jianmei Guo, Catalin Sporea, Andrei Toma, and Sarah Sajedi. 2020. Using Black-Box Performance Models to Detect Performance Regressions under Varying Workloads: An Empirical Study. *Empirical Softw. Engg.* 25, 5 (sep 2020), 4130–4160. https://doi.org/10.1007/s10664-020-09866-z

[25] Lizhi Liao, Jinfu Chen, Heng Li, Yi Zeng, Weiyi Shang, Catalin Sporea, Andrei Toma, and Sarah Sajedi. 2021. Locating Performance Regression Root Causes in the Field Operations of Web-based Systems: An Experience Report. *IEEE Transactions on Software Engineering* (2021), 1–1. https://doi.org/10.1109/TSE.2021.3131529

[26] S. Lloyd. 1982. Least squares quantization in PCM. *IEEE Transactions on Information Theory* 28, 2 (1982), 129–137. https://doi.org/10.1109/TIT.1982.1056489

[27] Shutian Luo, Huanle Xu, Chengzhi Lu, Kejiang Ye, Guoyao Xu, Liping Zhang, Yu Ding, Jian He, and Chengzhong Xu. 2021. Characterizing Microservice Dependency and Performance: Alibaba Trace Analysis. In *Proceedings of the ACM Symposium on Cloud Computing*. 412–426.

[28] Jonathan Mace. 2017. *End-to-End Tracing: Adoption and Use Cases*. Survey. Brown University. https://cs.brown.edu/people/jcmace/papers/mace2017survey.pdf

[29] C. Majors, L. Fong-Jones, and G. Miranda. 2022. *Observability Engineering: Achieving Production Excellence*. O'Reilly Media, Incorporated. https://books.google.it/books?id=MbmLzgEACAAJ

[30] Sam Newman. 2015. *Building Microservices* (1st ed.). O'Reilly Media, Inc.

[31] Charlene O'Hanlon. 2006. A Conversation with Werner Vogels. *Queue* 4, 4 (May 2006), 14:14–14:22. https://doi.org/10.1145/1142055.1142065 Place: New York, NY, USA Publisher: ACM.

[32] Austin Parker, Daniel Spoonhower, Jonathan Mace, Ben Sigelman, and Rebecca Isaacs. 2020. *Distributed Tracing in Practice: Instrumenting, Analyzing, and Debugging Microservices*. O'Reilly Media, Incorporated.

[33] Peter J. Rousseeuw. 1987. Silhouettes: A graphical aid to the interpretation and validation of cluster analysis. *J. Comput. Appl. Math.* 20 (1987), 53–65. https://doi.org/10.1016/0377-0427(87)90125-7

[34] Julia Rubin and Martin Rinard. 2016. The Challenges of Staying Together While Moving Fast: An Exploratory Study. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. Association for Computing Machinery, New York, NY, USA, 982–993. https://doi.org/10.1145/2884781.2884871

[35] Raja R. Sambasivan, Ilari Shafer, Jonathan Mace, Benjamin H. Sigelman, Rodrigo Fonseca, and Gregory R. Ganger. 2016. Principled Workflow-centric Tracing of Distributed Systems. In *Proceedings of the Seventh ACM Symposium on Cloud Computing (SoCC '16)*. ACM, New York, NY, USA, 401–414. https://doi.org/10.1145/2987550.2987568

[36] R. R. Sambasivan, I. Shafer, M. L. Mazurek, and G. R. Ganger. 2013. Visualizing Request-Flow Comparison to Aid Performance Diagnosis in Distributed Systems. *IEEE Transactions on Visualization and Computer Graphics* 19, 12 (2013), 2466–2475. https://doi.org/10.1109/TVCG.2013.233

[37] Benjamin H. Sigelman, Luiz André Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. 2010. *Dapper, a Large-Scale Distributed Systems Tracing Infrastructure*. Technical Report. Google.

[38] Cindy Sridharan. 2017. Testing Microservices, the sane way. https://copyconstruct.medium.com/testing-microservices-the-sane-way-9bb31d158c16 "Accessed 2023-01-16 11:14".

[39] Uber Technologies. 2023. Jaeger: Open source, end-to-end distributed tracing. https://www.jaegertracing.io/ "Accessed 2023-01-15 14:10:59".

[40] Luca Traini. 2022. Exploring Performance Assurance Practices and Challenges in Agile Software Development: An Ethnographic Study. *Empirical Software Engineering* 27, 3 (2022), 74. https://doi.org/10.1007/s10664-021-10069-3 ISBN: 1573-7616.

[41] Luca Traini and Vittorio Cortellessa. 2023. DeLag: Using Multi-Objective Optimization to Enhance the Detection of Latency Degradation Patterns in Service-Based Systems. *IEEE Transactions on Software Engineering* (2023), 1–28. https://doi.org/10.1109/TSE.2023.3266041

[42] Luca Traini, Vittorio Cortellessa, Daniele Di Pompeo, and Michele Tucci. 2022. Towards effective assessment of steady state performance in Java software: are we there yet? *Empirical Software Engineering* 28, 1 (2022), 13. https://doi.org/10.1007/s10664-022-10247-x

[43] Luca Traini, Daniele Di Pompeo, Michele Tucci, Bin Lin, Simone Scalabrino, Gabriele Bavota, Michele Lanza, Rocco Oliveto, and Vittorio Cortellessa. 2021. How Software Refactoring Impacts Execution Time. *ACM Trans. Softw. Eng. Methodol.* 31, 2, Article 25 (dec 2021), 23 pages. https://doi.org/10.1145/3485136

[44] Luca Traini, Jessica Leone, Giovanni Stilo, and Antinisca Di Marco. 2023. VAMP - Replication Package. https://github.com/lucatraini/VAMP.

[45] Kaushik Veeraraghavan, Justin Meza, David Chou, Wonho Kim, Sonia Margulis, Scott Michelson, Rajesh Nishtala, Daniel Obenshain, Dmitri Perelman, and Yee Jiun Song. 2016. Kraken: Leveraging Live Traffic Tests to Identify and Resolve Resource Utilization Bottlenecks in Large Scale Web Services. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 635–651.

[46] Chenxi Zhang, Xin Peng, Chaofeng Sha, Ke Zhang, Zhenqing Fu, Xiya Wu, Qingwei Lin, and Dongmei Zhang. 2022. DeepTraLog: Trace-Log Combined Microservice Anomaly Detection through Graph-based Deep Learning. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*. 623–634. https://doi.org/10.1145/3510003.3510180

[47] Xiang Zhou, Xin 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. https://doi.org/10.1109/TSE.2018.2887384