Università degli Studi dell'Aquila

Enhancing Trace Visualizations for Microservices Performance Analysis

Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica

Corso di Laurea Magistrale in Informatica

Candidate

Jessica Leone
ID number 275331

Thesis Advisor

Prof. Giovanni Stilo

Co-Advisor

Luca Traini

Academic Year 2022/2023

Enhancing Trace Visualizations for Microservices Performance Analysis
Master's thesis. Università degli Studi dell'Aquila

This thesis has been typeset by LaTeX and the uaqthesis class.

Author's email: jessica.leone@student.univaq.it

Acknowledgments

This work is supported by Territori Aperti (a project funded by Fondo Territori, Lavoro e Conoscenza CGIL CISL UIL)

Abstract

Analysis of microservices' performances 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 enabling at once the analysis of the performances 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 requests recurrent characteristics and their relation w.r.t. the end-to-end performance behavior. Through an extensive evaluation of 33 datasets generated from an established open-source microservices system, we show that VAMP can effectively help detect the execution time deviations of specific RPCs that produce degradation of end-to-end performance. Furthermore, VAMP enables the identification of meaningful structural patterns in end-to-end requests and their relation with microservice performance behaviors.

Video URL: https://youtu.be/qMVOMt06EJE

Contents

List of Figures

Chapter 1

Introduction

Microservices have revolutionized the software industry, introducing a novel paradigm for structuring software development processes [43]. This approach involves independent teams responsible for the entire software development lifecycle, delivering loosely coupled and independently deployable services [44, 43] each being responsible for a small subset of the applications functionality [21]. The modular nature of microservices aligns well with the need for rapid software updates and enhancements, which are crucial for maintaining a competitive edge [48].

However, the adoption of microservices also presents challenges, with ensuring adequate software performance being one of the primary concerns. Proactive performance assurance, such as pre-production testing [27, 57, 34] , is difficult due to the inherent complexity of these systems [59, 54]. Time and resource constraints further limit performance assurance efforts, as there is significant pressure to deliver fast-to-market [57, 48]. Additionally, microservices systems exhibit emergent performance behavior in the field, influenced by the complex interactions of multiple independent services and machines [59]. These systems also face constant software changes and variable workloads, making them susceptible to unforeseen performance regressions [4, 58, 59].

To address these challenges, there has been a growing interest in the concept of *observability* [42], which refers to the ability to gain a comprehensive understanding of the system's performance by analyzing its logs, traces, and metrics. Distributed tracing has emerged as a popular observability tool of microservices systems [41, 46],

capturing the flow of causally-related events within these systems [49]. These tools, fetch or receive trace data from complex distributed systems, such as microservices, and process this data before presenting it to the user using more readable charts and graphs. However, existing distributed tracing tools have been criticized for their limited support in analyzing system-wide performance behavior [14, 46]. They often require switching between different visualization tools, leading to a cumbersome and time-consuming process [14]. Current tracing visualizations primarily focus on individual request analysis, providing swimlane visualizations as the canonical way to visualize individual requests, lacking the ability to compare and analyze the performance behavior of the entire request corpus [3, 46, 14]. observing the performance of individual end-to-end requests might result in misleading insights if not contextualized appropriately [14]. Indeed, a request's response time can only be deemed anomalous when compared with other requests of the same type [3]. Additionally, engineers are often more inclined to investigate recurrent response time trends rather than focusing on the performance of individual requests [46]. 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.

In this thesis, we introduce VAMP, an innovative visual analytics tool designed to enhance the performance analysis of microservices systems. VAMP builds upon the concept of distributed tracing and offers two main visualization components. The first is an interactive tree that illustrates the workflow of multiple end-to-end requests in terms of Remote Procedure Calls (RPCs). The second is an interactive histogram representing the performance behavior of the analyzed requests. These visual components facilitate the understanding of how specific system performance behaviors relate to the characteristics of RPC execution paths.

We evaluate VAMP using 33 datasets derived from the widely used TrainTicket microservices system [64]. Our findings demonstrate that VAMP enables the identification of significant and recurring request characteristics associated with

specific end-to-end response time behaviors.

1.1 Organization of the Thesis

This thesis is organised as follows:

1. In chapter 2, we introduce the background knowledge. In particular, we describe microservice architectures describing how to analyze the performance of these systems and the importance of using distributed tracing tools. Then we introduce the concept of Visual Analytics making a survey of the used techniques for visualization and interaction with them. Finally we present existing visualization techniques for microservices systems.

2. In chapter 3, we present the motivations behind the development of VAMP providing and overview of the main limitations of existing systems.

3. In chapter 4, we introduce VAMP. We describe the components of the tool and how is possible to interact with each one. We then present the dashboard and describe how we have implemented the tool.

4. In chapter 5, we describe the experimental analisys of the tool by first describing the methodology we have used and then we report and discuss the results obtained from the evaluation.

5. Chapter 6 concludes the thesis and provides some future works.

Chapter 2

Background

This chapter introduces the necessary background knowledge for understanding the subsequent chapters of the thesis, where VAMP is presented: a novel visual analytics tool developed to facilitate performance analysis in microservices architecture.

The chapter begins by introducing the microservices architecture, its impact on the software industry and explains the fundamental principles of microservices.

Next, we explore the challenges associated with ensuring adequate software performance in microservices architecture, and we present common tools for microservices observability.

Following that, we introduce the concept of visual analytics. We describe its principles and the typical process of Visual Analytics used as a starting point for the development of VAMP.

Finally, we present existing visualization techniques for microservices systems.

2.1 Microservices architecture

In line with the latest trends in Software Engineering, systems are progressively growing in size and becoming more distributed. This necessitates the adoption of new solutions and development patterns [6]. One such approach that has emerged in recent years is the adoption of microservices architecture. Microservices are an architectural style for building large software applications by breaking them down into smaller, independent services. Each microservice focuses on performing a

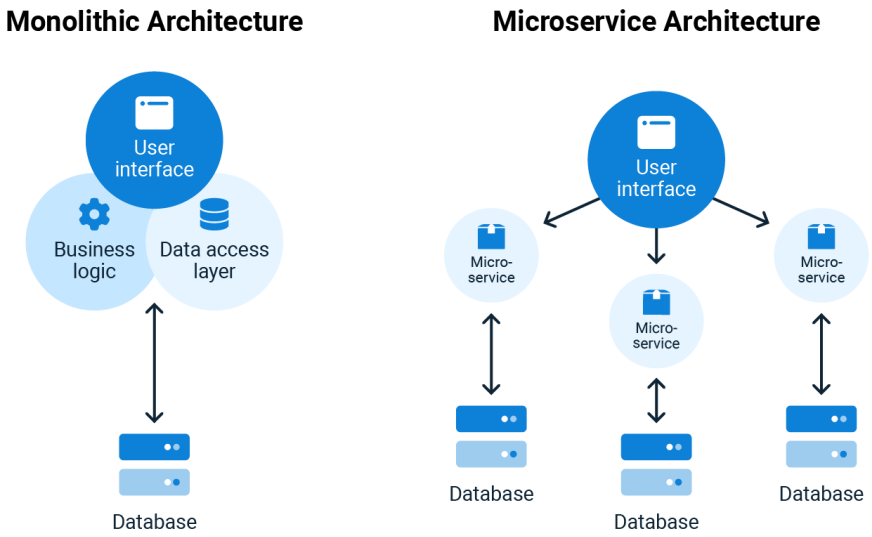


Figure 2.1. Microservices architecture

specific task, representing a small business capability.

Microservices emphasize using the most appropriate programming language and technology for each task. They can be written in different languages and technologies, depending on the specific requirements. Communication between microservices is achieved through language-neutral APIs, typically based on HTTP, such as REST. This approach allows for leveraging existing skills or choosing the optimal language for each microservice.

One key characteristic of microservices is their small and focused nature. There is no strict rule on the size of a microservice, but a commonly referenced guideline is the "Two-Pizza Team" rule [15], which suggests that a microservice should be small enough to be developed and maintained by a team that can be fed with two pizzas. Treating each microservice as an individual application or product, they have their own source code repository and delivery pipeline.

Loose coupling is another important aspect of microservices. Each microservice can be deployed independently without coordination with other services. This enables frequent and rapid deployments, allowing for the quick delivery of new features and capabilities.

The concept of bounded context is crucial in microservices. A microservice should not have knowledge about the internal implementation or architecture of other microservices. If a microservice needs to know about another service, it indicates a violation of bounded context. Keeping the interfaces small and minimizing dependencies between microservices helps maintain a clear separation of concerns.

However, it is important to strike a balance in microservice granularity. Overly granular services or excessive dependencies can introduce latency issues, which need to be considered and addressed in the design and implementation of microservices.

One of the key principles of a microservices architecture is the idea of service autonomy. Each microservice operates as an independent entity and has its own dedicated database, tailored to its specific needs. This decentralized approach allows each service to make independent decisions about the technology stack, libraries, and frameworks it uses, without being constrained by other services. As a result, different services within the same application can be developed using different programming languages or frameworks, based on what is most suitable for their requirements.

Microservices architecture presents different advantages:

- Agile and rapid development, as each microservice can be developed, tested, and deployed independently by a dedicated team. This allows for faster release cycles and the ability to quickly adapt to changing business needs.

- Ability to make targeted bug fixes or updates. Since each microservice represents a specific business function, developers can make changes or improvements to a particular service without having to redeploy the entire application. This granularity of updates reduces the risk of introducing new bugs or disruptions to other parts of the system.

- Reusability of microservices: because each service focuses on a specific business capability, it can be reused within the same application or even in different contexts. This promotes code reusability, reduces duplication of effort, and facilitates the sharing of functionality across various projects or teams.

- Scalability: with the ability to instantiate and allocate resources on-demand, services can scale horizontally to handle increased workload or traffic. This elas-

ticity allows the system to adapt dynamically to changing demands, ensuring optimal performance and resource utilization.

- the distributed nature of microservices architecture enhances fault tolerance and eliminates single points of failure. Since services are decoupled and isolated, the failure of one service does not impact the entire system. Errors can be more easily detected and isolated, enabling efficient troubleshooting and minimizing the impact on the overall application.

However, it's important to note that adopting a microservices architecture also introduces additional complexities. Managing the inter-service communication, ensuring data consistency across multiple databases, and coordinating deployment and versioning of different services require careful planning and robust infrastructure. Proper monitoring and observability mechanisms are crucial to gain insights into the behavior and performance of individual services as well as the overall system.

2.1.1 Performance analysis in microservices

Microservice systems present unique challenges when it comes to performance analysis. These systems are characterized by their extremely small-grained and complex interactions among microservices, as well as the intricate configurations of their runtime environments. The execution of a microservices system involves a significant number of asynchronous interactions, often resulting in complex invocation chains [63].

Understanding the system's communication topology and reasoning about the concurrent activities of system hosts can be difficult. When systems fails, log data is often the only information available [62], so one common approach to performance analysis is the examination of logs.

Logs are detailed records of activities that occur within a microservice. They can be generated for various events such as errors, exceptions and contain different types of information such as error messages, warnings, debugging information, timestamps, user actions, system events and so on. Logs provide a comprehensive view of the activities and events happening in the system and can be useful for identifying errors,

detect performance anomalies and help identify possible cause problems.

However, logs often contain a mix of different types of information and do not have a predefined structure. In many systems, they are generated by various components or applications without a standardized format or organization in capturing and storing these data.

Therefore, analyzing system logs in a distributed environment requires reconciling logs from multiple hosts, dealing with the challenges of non-synchronized clocks, and deciphering the encoded execution flow [7].

One of the defining characteristics of microservice applications is their ability to scale, with dozens to thousands of microservices running on hundreds to tens of thousands of machines [65]. While these systems typically have simple interfaces and exhibit quick response times, maintaining optimal performance levels over time is a complex task. Unexpected performance degradations arise frequently, and addressing them requires substantial human effort[50].

When a performance degradation occurs in a microservice system, the first crucial step is to identify the root cause. It could stem from any of the system's software components, unexpected interactions between these components, or slowdowns in the network connecting them. Traditionally, the process of identifying the cause has relied on ad-hoc manual approaches. Developers would rely on raw performance data collected from individual components and attempt to analyze it to pinpoint the issue.

However, as microservice systems have grown in scale and complexity, these ad-hoc processes have become less viable. Analyzing system logs, which is a standard approach, has become a tedious and intricate task.

Another common approach in analyzing the performance of distributed systems is to consider the performance behavior of multiple traces. This analysis is often represented using a histogram, which illustrates the distribution of latencies for various requests. Within this histogram, it is possible to identify peaks, known as *modes* which represent distinct groups or patterns of latency behavior.

Modes in a latency histogram indicate different performance characteristics or behaviors within the system. Each mode typically represents a specific set of

conditions or factors that influence the latency of requests. By identifying and analyzing these modes, developers and system administrators can gain insights into the different performance profiles or scenarios within the system [46].

For example, a bimodal distribution in the latency histogram may indicate the presence of two distinct performance patterns. This could suggest that there are two different paths or workflows within the system, each with its own latency behavior. By understanding these modes, one can investigate the underlying causes and potentially optimize the system accordingly.

Modes in the latency histogram can also help in visually identifying potential outliers or anomalies. For example, plotting a histogram or a density plot of the data and observing modes can highlight regions where there is a high density of values, while outliers may appear as isolated points or fall outside these dense regions.

2.1.2 Distributed tracing tools

As seen in the previous sections, microservice architectures involve deploying numerous service instances that can be dynamically created and destroyed based on workload demands. Each microservice performs a specific function and communicates with other services over the network. With this distributed and decentralized nature, it becomes challenging to gain visibility into how the system as a whole is functioning.

To understand microservice systems' behaviors and troubleshoot their problems, it is essential to ensure observability of these systems [36]. Observability refers to the capability of gaining insights into the internal workings of a system by systematically collecting and analyzing relevant data. It involves monitoring and analyzing various signals, logs, metrics, and traces from a system to understand its behavior, performance, and potential issues. In the context of microservices, observability is crucial due to the complex and dynamic nature of these systems.

Distributed tracing tools are recognized as important means to achieve observability in microservice systems. Distributed tracing allows us to track the flow of requests as they traverse through various microservices [6]. It captures information about each service's involvement in request processing, including timestamps, dependencies, and any encountered errors or delays.

By analyzing these traces, operators and developers can gain valuable insights into the behavior and performance of the microservices. They can identify bottlenecks, latency issues, and potential areas for optimization. Distributed tracing also facilitates troubleshooting by providing a detailed view of the request's journey, making it easier to pinpoint and analyze the root cause of any problems that may arise.

Distributed tracing systems continuously collect traces from a microservices system using the standard defined by OpenTracing [1]. According to this standard, traces are defined implicitly by their spans. A span represents a unit of work or an operation within a system. It could be a single function call, a database query, an RPC call, an HTTP request, or any other discrete operation that occurs within a microservice system.

A span has different information:

- A parent span.
- A span name (operation name).
- A span kind.
- Start and end time.
- A status that reports whether operation succeeded or failed.
- A set of key-value attributes describing the operation.
- A timeline of events.
- A list of links to other spans.
- A span context that propagates trace ID and other data between different services.

Each operation within a microservice generates such a span, each with its own temporal and contextual information.

In a distributed system, with a high volume of requests involving numerous microservices, collecting and managing tracing data can become complex. Without

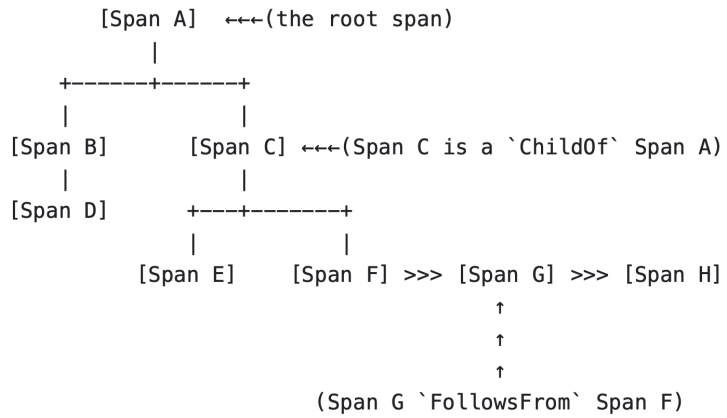


Figure 2.2. Causal relationships between Spans in a single Trace [1]

proper correlation mechanisms, it would be difficult to reconstruct the flow of requests and understand the interaction between microservices.

In this regard, OpenTracing provides trace correlation mechanisms, which allows to reconstruct the complete flow of requests through the different microservices.

In OpenTracing, each trace is identified by a unique Trace ID. This Trace ID is propagated along the request path and is used to correlate spans, which represent operations within microservices. Each span has its own Span ID, which allows you to link related spans within a trace. The tracing context, which contains the Trace ID and Span ID, is propagated between microservices to allow spans to be created and correctly associated with the current trace. This tracing context propagation allows you to create a hierarchical trace structure as shown in Figure 2.2, where each span can be linked to its parent and child span. Through this hierarchical structure, it is possible to reconstruct the causal flow of operations and understand the path of requests through the microservices.

2.2 Visual Analytics

In this thesis we present VAMP, a visual analytics tool for microservices performance analysis.

For this reason, the concept of Visual Analytics is closely related to this work.

In this section we provide an overview of the concept of Visual Analytics.

2.2.1 Principles of Visual Analytics

Visual Analytics is *"the science of analytical reasoning facilitated by interactive visual interfaces"* [10]. Is an integrated approach that combines visualization, algorithmic data analysis, human-computer interaction and analytical reasoning. It exploits visualization as a tool to integrate human cognition, perception abilities and human intelligence into the data-analysis process to obtain explanaible results.

Visual Analytics gives higher priority on analyzing data and discovering knowledge in data, rather than just presenting and understanding the data [12].

According to the definition given in [10], the core idea of Visual Analytics is to integrate the human cognitive, perceptual and reasoning abilities, along with their knowledge, into the analysis process. The ultimate goal of this integration is to gain insights from complex data that are challenging to explore using only visualization or analysis techniques in isolation.

Visual Analytics builds on the principle of "Overview First, Zoom/Filter, Details on Demand" [52] to facilitate effective data exploration and analysis. This principle emphasizes the importance of providing users with an initial overview of the data, allowing them to grasp the overall patterns and trends before delving into specific details. It enables users to progressively zoom in on areas of interest, apply filters to refine the view, and request more detailed information as needed. The "Overview First" aspect of this principle acknowledges that users often need a high-level understanding of the dataset before they can effectively analyze it. Visual Analytics tools provide visualizations that offer a broad view of the data, capturing its overall structure and distribution. This overview allows users to identify general patterns, outliers, and potential relationships, serving as a starting point for further exploration. Once users have obtained the initial overview, they can apply the "Zoom/Filter" approach to focus on specific subsets or aspects of the data. They can zoom in on particular regions of interest or filter the data based on specific criteria, narrowing down their analysis to relevant subsets. By interacting with the visualizations, users can navigate through different levels of detail, revealing more specific information

and gaining deeper insights. The "Details on Demand" component of the principle recognizes the importance of providing users with the ability to access more detailed information when necessary. Users can interact with the visualizations to request additional details about specific data points, such as numerical values, annotations, or underlying data sources. This on-demand approach ensures that users can access the desired level of information precisely when they need it, without overwhelming them with excessive details. The "Overview First, Zoom/Filter, Details on Demand" principle supports an iterative and exploratory analysis process. It empowers users to navigate through data, focus on areas of interest, and progressively drill down to gain a deeper understanding. By combining an initial overview with interactive zooming, filtering, and on-demand details, Visual Analytics tools provide users with a flexible and user-centric environment for data exploration, analysis, and insight generation. Overall, the "Overview First, Zoom/Filter, Details on Demand" principle forms a fundamental basis for the design and implementation of Visual Analytics tools. It guides the development of interactive visualizations that allow users to navigate complex datasets effectively, uncover meaningful patterns, and make data-driven decisions.

2.2.2 Visual Analytics Process

The Visual Analytics (VA) process involves a systematic and iterative approach to data analysis, exploration, and insight generation. It combines visualizations, analytical techniques, and user interaction to support effective decision-making. Figure 2.3 presents the typical steps involved in the VA process:

1. Data Acquisition and Preprocessing:The first step in the VA process is to acquire the relevant data for analysis. This may involve collecting data from various sources, such as databases, APIs, or files. Once the data is collected, it needs to be preprocessed and cleaned to ensure data quality and compatibility with the VA tool. This includes tasks such as data cleansing, integration, transformation, and formatting.

2. Analytical Techniques and Algorithms: Analytical techniques are employed

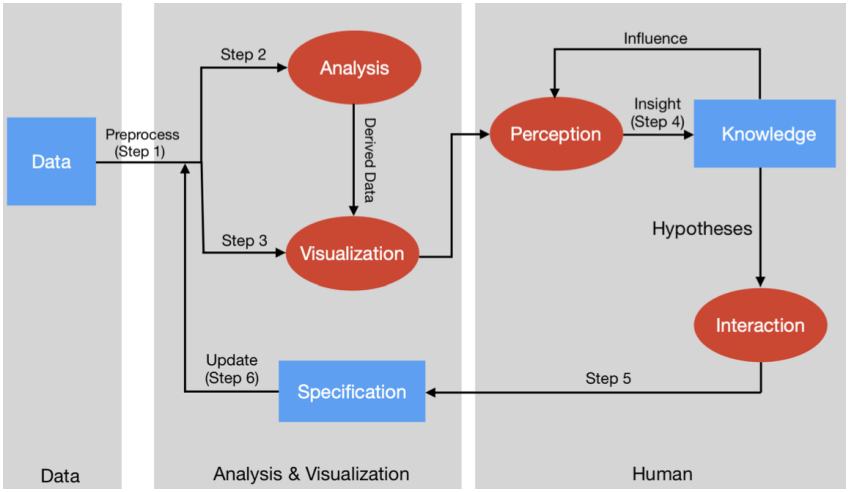


Figure 2.3. Visual Analytics process [12]

to analyze the data within the VA framework. These techniques can include statistical analysis, machine learning algorithms, data mining methods, or domain-specific analytical models. The integration of analytical techniques within the VA process allows users to perform advanced computations and derive meaningful insights from the data.

3. Visualization: In this step, visualizations are designed to represent the data effectively. The choice of visualization techniques and types depends on the characteristics of the data and the analytical goals. The design should consider the appropriate visual encodings, layout, color schemes, and interactivity to facilitate data exploration and analysis. The visualizations should align with the "Overview First, Zoom/Filter, Details on Demand" principle to provide users with a comprehensive view of the data.

4. Knowledge Discovery and Insight Generation: Based on the interactive exploration and application of analytical techniques, users can discover knowledge and generate insights. This involves identifying patterns, relationships, correlations, outliers, or trends that were not initially apparent. Visualizations and analytical results are combined to support the understanding of complex data and facilitate the extraction of actionable insights.

5. Interaction: Users interact with the visualizations to explore the data at

different levels of detail. They can zoom in and out, apply filters, select data subsets, and perform interactive operations to focus on specific areas of interest. The goal is to identify patterns, trends, anomalies, and potential insights. Users can perform visual queries, refine their exploration based on the initial findings, and dynamically adjust the visualizations to gain deeper insights.

6. Visualization update: Iterative Analysis: The VA process is typically iterative, allowing users to refine their analysis, explore alternative hypotheses, and address new questions or requirements. Users can go back to previous steps, modify visualizations, adjust parameters, or apply different analytical techniques based on their evolving understanding and exploration of the data. The iterative nature of the process allows for deeper insights and a more comprehensive analysis.

The process follows an iterative loop, continuously cycling from step 4 to step 6 until sufficient insight is gained from decision-making problem-solving. It emphasizes the integration of human capabilities and computational techniques, leveraging the strengths of both to extract knowledge and gain deeper insights from complex data. For this reason, the classinc way of visually exploring data defined in [52] *"Overview first, Zoom/Filter, Details on demand"*, needs to be extended to the Visual Analytics Mantra: *"Analyze first-Show the important-Zoom,filter and analyze Further-Details on demand"* [30].

Since the Visual Analytics Process is based on visualization and interaction with it, the succes of the process depends on:

1. the breadth of the collection of visualization techniques
2. the consistency of the design of the vies
3. the ability to interactively remap data attributes to visualizations attributes
4. the set of functions to interact with visualizations and the capabilities that these functions offer to support the reasoning process. [12]

Visualizations techniques can be classified based on [31]:

1. Data Types: Visualization techniques can be categorized based on the type of data being visualized. This can include unidimensional data like temporal data, bidimensional data like geographic maps and relational tables, text and hypertext, hierarchies, graphs, etc.

2. Visualization Techniques: Techniques can be classified based on the specific visualization methods used, such as 2D/3D visualizations, geometrically transformed displays, tree maps, etc.

3. Interaction Techniques: This category encompasses the interactive aspects of visualization, including techniques like interactive projection, zooming, distortion, etc. These techniques enable users to manipulate and explore the visualized data.

It is important to note that these three dimensions of classification—the type of data, visualization techniques, and interaction techniques—are considered orthogonal. Orthogonality means that any visualization technique can be combined with any interaction or distortion technique for any type of data.

Furthermore, it is worth noting that specific systems can be designed to support different types of data and can employ a combination of visualization and interaction techniques.

Additionally, there are two main categories of visualization [32]:

1. Data Visualization: is the study of representing data in some systematic form, including attributes and variables for the unit of information. This category focuses on visualizing raw data to represent its underlying patterns, trends, or relationships. It often involves the use of charts, graphs, plots, and other visual representations to convey information (e.g. bubble chart, scatter plot, histogram, pie chart...).

2. Information Visualization: is a research domain that concentrates on the use of visualization methods to assist people understanding data and evaluate or analyze data. This category is concerned with visualizing complex information, such as large datasets or interconnected networks, to support exploration, analysis, and understanding. Information visualization often employs techniques

like network visualizations, tree maps, scatter plots, etc., to reveal insights and relationships in the data.

There are several ways to interact with data visualizations:

1. ZOOMING: This interaction allows users to zoom in or out on specific areas of a visualization to examine details or obtain a broader view. It helps users focus on specific regions of interest and navigate through the data.

2. OVERVIEW + DETAIL: This technique involves displaying multiple views simultaneously, combining an overview of the entire dataset with detailed views of specific portions. It enables users to maintain context while exploring specific areas in more detail.

3. FISHEYE: The fisheye technique expands or magnifies a focused area within the overall view. It provides a distorted but detailed representation of the selected region, allowing users to see fine-grained details while preserving context.

4. IDENTIFICATION: This interaction displays an identifying label when the mouse hovers over a specific area or element in the visualization. It provides additional information or metadata about the selected item, enhancing the user's understanding.

5. LINKING: Linking allows the connection of selected elements across multiple visualizations or graphs. When an element is selected in one view, it highlights or shows related elements in other linked views. This technique helps users understand relationships and dependencies between different data points.

6. BRUSHING: allows users to select or highlight specific data points or regions of interest within a visualization by using a pointing device (e.g., mouse cursor) to draw a brush-like selection.

Visualizations and interaction techniques can indeed be combined to create interactive dashboards. An interactive dashboard provides a dynamic and immersive experience for users, allowing them to explore data, analyze trends, and gain insights through interactive visualizations.

Interaction techniques play a critical role in enhancing the usability and interactivity of the dashboard. They enable users to manipulate the visualizations, drill down into details, filter and sort data, and perform various actions to explore different aspects of the data. By interacting with the visualizations, users can dynamically change parameters, apply filters, and interactively explore the data from different angles, gaining deeper insights and understanding.

By combining visualizations with interactive capabilities, an interactive dashboard empowers users to actively engage with the data, uncover patterns, and gain insights in real-time. It promotes an iterative and exploratory approach to data analysis, allowing users to ask questions, test hypotheses, and make data-driven decisions effectively.

2.3 Microservices visualizations

2.3.1 Visualizations in practice

Once traces have been collected and processed, distributed tracing tools presents this data to users using more readable charts and diagrams [6]. The most widely used visualization technique is the swimlane view [14, 29, 53]. It allows users to manually investigate individual traces in detail as part of troubleshooting tasks. In the swimlane view, spans (representing individual operations) are depicted horizontally and sorted vertically, creating a timeline of a single request as shown in Figure 2.4.

Figure 2.4. Swimlane visualization [55].

In the swimlane view, the relationships between spans are typically depicted implicitly based on vertical ordering and indentation of span names. Some visualizations may explicitly show relationships using lines connecting spans [49].

While the swimlane view is the most pervasive visualization approach in distributed tracing, some distributed tracing tools also provide aggregate visualizations for analyzing large volumes of traces. These visualizations aim to capture the system's behavior as a whole. For instance, Canopy [29] allows operators to extract per-trace metrics into a tabular form, which can be queried using standard time-series database interfaces.

Another example is the dependency graph visualization of Jaeger [55], which shows the relationships and interactions between services, based on the traced data. This visualization helps in understanding the dependencies, communication patterns and overall architecture of the system. Jaeger [55] also enables the comparison of structural aspects of two requests [20].

Commercial APM tools [2], such as Dynatrace [25], AppDynamics [24], or Instana [23], provide features that support aggregated analysis of end-to-end requests. The Service Flow feature offered by Dynatrace [26] aims to provide aggregate workflows of end-to-end requests, along with their associated characteristics. This feature helps users analyze the relationship between request characteristics and end-to-end performance behavior.

All these trace visualizations, along with the swimlane view, offer various ways to explore and analyze distributed trace data, allowing users to troubleshoot issues, optimize performance, and make informed decisions. The choice of visualization depends on the specific use case and the insights users seek to extract from the trace data.

2.3.2 Research on visualizations

Prior research on visualization approaches for distributed systems has primarly concentrated on the analysis of individual requests or comparison between two requests.

ShiViz [7] offer a time-space diagram visualization of logged distributed executions,

explicitly representing the 'happens before' relationship between events. ShiViz contains features to support the following 3 main system understanding tasks:

- ORDERING OF EVENTS: ShiViz space-time diagram explicitly but compactly represents the relative ordering of events among hosts in the system, capturing the concurrency between events. ShiViz allows the developer to simplify the graph by transforming it to eliminate information that is not needed for the current task.
- MODELS OF INTERACTION: allows you to build subgraphs of events and search for them in the space-time diagram.
- COMPARISON OF MULTIPLE EXECUTIONS: Ability to present two execution graphs side-by-side to help developers compare these executions. ShiViz includes algorithms to highlight differences between execution pairs and supports grouping executions based on features.

The study of Sambavisan *et al.* [50] focus specifically on comparing two request-flow traces with a side-by-side view, difference view and animation between them. [51].

TraVista [3] is a tool designed for debugging performance issues in a single trace. TraVista extends the popular single trace Gantt chart visualization with three types of aggregate data-metric, temporal, and structure data, to contextualize the performance of the offending trace across all traces.

The research conducted by Davidson and Mace [13] highlights the significance of visualization in the field of systems research. They emphasize the need for continued exploration and development of visualization techniques in this domain. The research by Davidson *et al.* [14] also includes a qualitative interview study that specifically focuses on identifying limitations of current distributed tracing tools. The findings from this study not only reveal the limitations of current distributed tracing tools but also highlight the areas where further research and development are required. One such area is the visualization of distributed tracing data, where innovative visualization techniques can play a crucial role in enabling effective analysis and comprehension of complex system behaviors.

To address the challenge of specifying and analyzing transient behavior in microservices systems, TransVis [5] utilizes novel human-computer interaction methods to make the tasks of specifying transient behavior as a non-functional requirement and analyzing its fulfillment more accessible. The approach incorporates chatbot interactions and visualizations of the system's resilience to facilitate these tasks.

Chapter 3

Motivation

Distributed tracing tools encounter numerous challenges that hinder users from effectively troubleshooting and analyzing system performance [14]. One of the primary issues is the lack of seamless integration between multiple user interfaces and tools, requiring manual access to relevant data and impeding efficient navigation.

As we have seen in previous section, often users are more interested in investigate recurrent response time trends rather than focusing on the performance of individual requests [46]. 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 [46, 33, 11].

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 [55] and Kibana [18] [14]. 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 when using current tools.

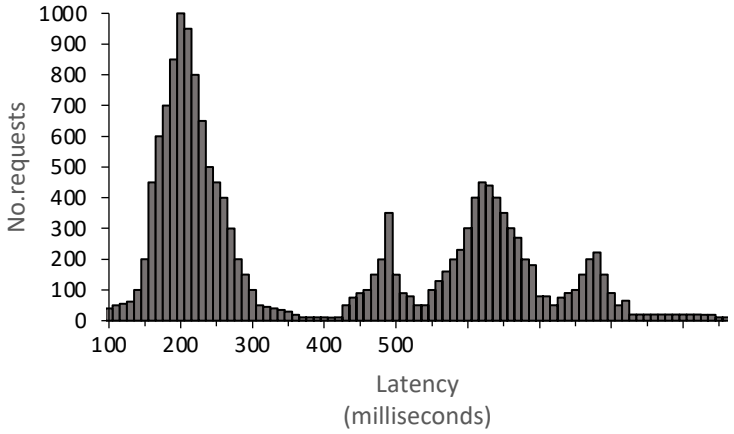


Figure 3.1. End-to-end response time distribution.

Consider the scenario illustrated in Figure 3.1 , 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. 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.

To address this, there is a need for a more interconnected approach that allows

for direct movement between tools. Moreover, existing tools often present trace data in a generic manner, failing to consider the broader context of the anomaly being investigated.

In addition to these limitations, Davidson *et al.* have pointed out other limitations of current distributed tracing systems by proposing in their research [14] a qualitative interview study that specifically focuses on identifying limitations of current distributed tracing tools.

A significant challenge of these systems is the overwhelming amount of information presented to users, leading to a high cognitive load. Current tools offer limited customization options, resulting in cluttered visualizations that include irrelevant data. Novice users often struggle with information overload, while important information may be hidden or buried deep within the tools, necessitating repetitive actions and manual effort.

Troubleshooting workflows also face obstacles as certain interactions are not adequately supported or require ad-hoc workarounds. For instance, accessing trace data programmatically for custom analysis can be difficult. Tasks like aggregate analysis, comparing and navigating traces, and searching for specific attributes within traces are not well-supported. Users often lack the ability to undo actions or easily compare traces, resorting to cumbersome methods such as opening multiple browser windows.

Another challenge lies in the development of these tools. Often, developers focus on specific visualization approaches while disregarding existing approaches that users may prefer. Additionally, there is limited feedback on tool usage and improvement, resulting in a disconnection between users and developers.

Data quality poses further challenges related to the longevity and presentation of trace data. Tracing backends typically retain data for a limited period, making it challenging to reproduce analyses. Users also encounter issues with malformed data and the impact of sampling policies on data accuracy. Existing tools are primarily designed to visualize ideal trace data, but troubleshooting requires analyzing traces that deviate from the norm, which can be challenging with certain visualization systems.

These challenges also extend to building aggregate analysis tools for tracing data. Determining the relevant dimensions or subset of data for analysis is not straightforward [3, 46]. Effectively visualizing high-dimensional trace data and designing efficient backend systems for filtering and aggregating data remain ongoing challenges in the development of aggregate analysis tools.

Addressing these challenges necessitates improvements in workflow integration, customization options, supported interactions, visualization approaches, data quality, and the development of effective tools for aggregate analysis. By overcoming these obstacles, distributed tracing tools can enhance users' ability to efficiently troubleshoot and analyze distributed systems.

Chapter 4

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 chapter, following the typical process of Visual Analytics we present VAMP. We start explaining the data acquisition and preprocessing phase, then we present the VAMP dashboard describing its primary visual components and the interaction modalities. Finally we outline the architecture and implementation details of VAMP.

4.1 VAMP architecture

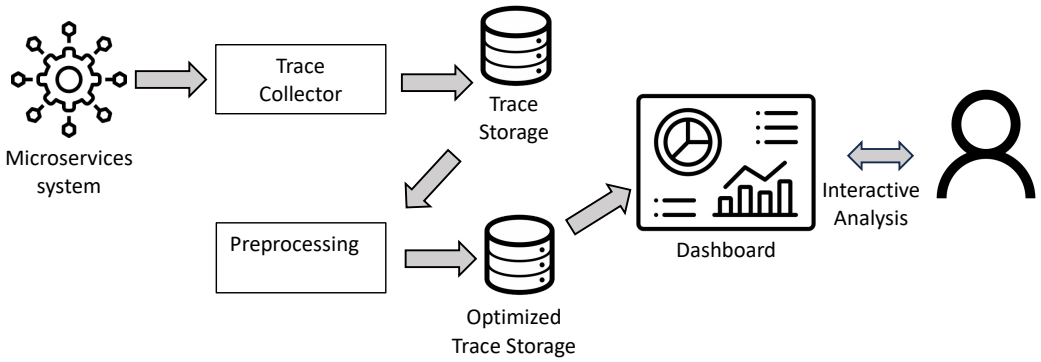


Figure 4.1. VAMP's Workflow

Figure 4.1 outlines the key architecture components of VAMP. Accordingly to the Visual Analytics process, the first step of VAMP is the acquisition of the data.

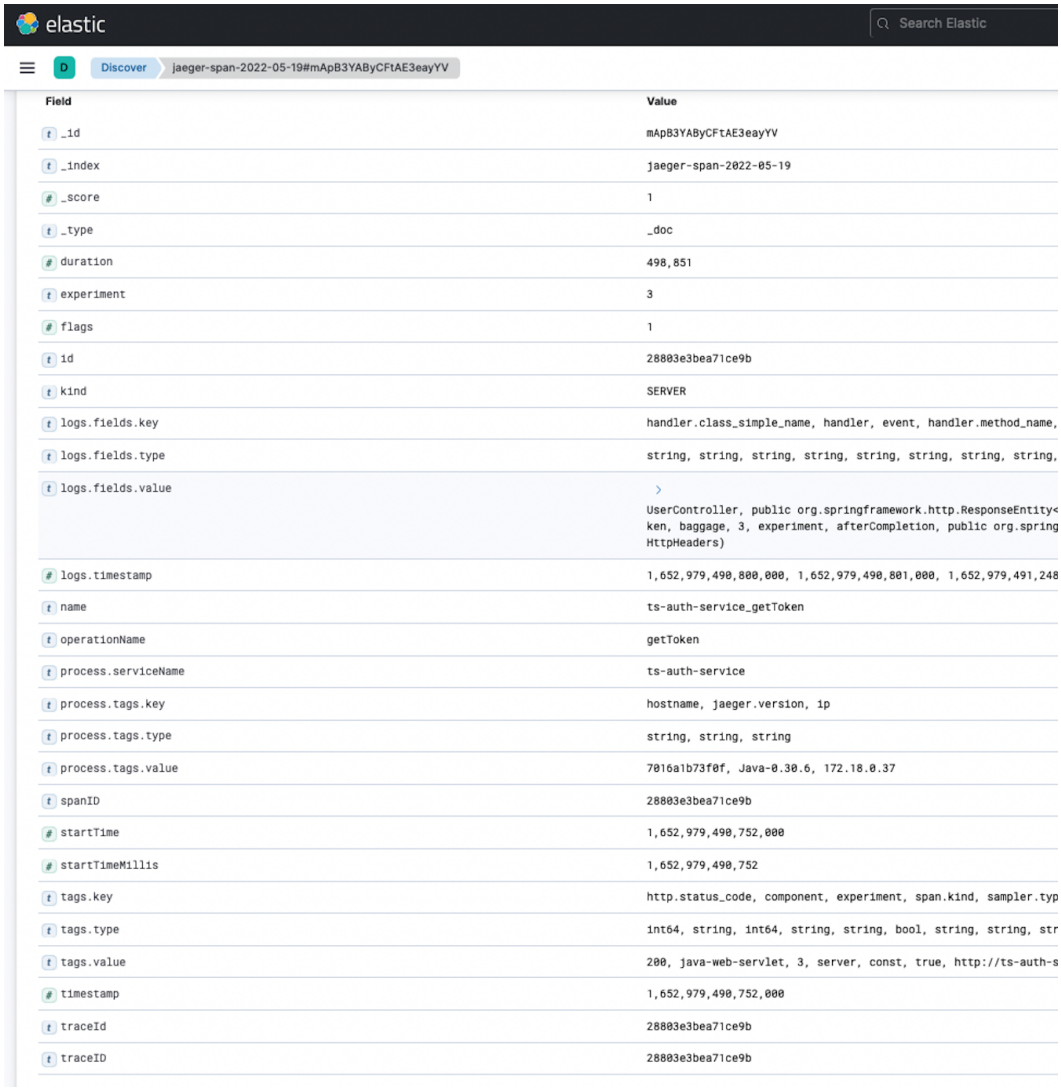
As depicted in figure, there is a *Trace Collector* (i.e., Jaeger [55]) continuously collects traces from the microservices system and stores them in a Trace Storage (i.e., Elasticsearch [17]) using the standard Jaeger format which is based on the OpenTracing specification [1] described in the previous chapter. In our TraceStorage, each span within a Jaeger trace is considered as a separate entity and stored as a separate document in Elasticsearch. Each document corresponds to a single span and includes all the relevant information about the span such as spanID, operationName, timestamp, traceId and other metadata as shown in Figure 4.2. Given the large volume of data collected each day, we have devised a *Preprocessing* step to enhance the efficiency of interaction with VAMP. The preprocessed traces are stored in an *Optimized Trace Storage* based on MongoDB. Finally, the Dashboard app directly query the Optimized Trace Storage in order to efficiently generate visualizations.

4.2 Preprocessing

The 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), all the spans that compose it are retrieved by querying the Trace Storage. Then a recursive function is used to traverse the trace hierarchy and construct the RPC execution path for each span. The function consider one span at time and by following the parent-child relationship, traverse the trace hierarchy backward until the root is reached. This happens when the function process a span whose parentId field is null. At that point, the function returns the RPC execution path constructed appending at each step the name of the current span to the existing path.

Once all the RPC execution paths of a trace have been created, these are stored in a collection in the *Optimized Trace Storage* based on MongoDB. In this collection, each RPC execution path will be uniquely inserted: if a trace has two or more spans with the same RPC execution path, in the collection we will have a single document referring to this RPC execution path for the considered trace. Each document stores other fileds in addition to the path name: the traceID, the number of occurrences of the path in the trace, the observed execution time, the timestamp and the name of



Field	Value
_id	mApB3YAByCFtAE3eayYV
_index	jaeger-span-2022-05-19
_score	1
_type	_doc
duration	498,851
experiment	3
flags	1
id	28803e3bea71ce9b
kind	SERVER
logs.fields.key	handler.class_simple_name, handler, event, handler.method_name,
logs.fields.type	string, string, string, string, string, string, string, string,
logs.fields.value	UserController, public org.springframework.http.ResponseEntity< ken, baggage, 3, experiment, afterCompletion, public org.spring HttpHeaders)
logs.timestamp	1,652,979,490,800,000, 1,652,979,490,801,000, 1,652,979,491,248
name	ts-auth-service_getToken
operationName	getToken
process.serviceName	ts-auth-service
process.tags.key	hostname, jaeger.version, ip
process.tags.type	string, string, string
process.tags.value	7016a1b73f0f, Java-0.30.6, 172.18.0.37
spanID	28803e3bea71ce9b
startTime	1,652,979,490,752,000
startTimeMillis	1,652,979,490,752
tags.key	http.status_code, component, experiment, span.kind, sampler.typ
tags.type	int64, string, int64, string, string, bool, string, string, str
tags.value	200, java-web-servlet, 3, server, const, true, http://ts-auth-s
timestamp	1,652,979,490,752,000
traceId	28803e3bea71ce9b
traceID	28803e3bea71ce9b

Figure 4.2. Elasticsearch document

the root RPC. A document in MongoDB will appear as in the Figure 4.3

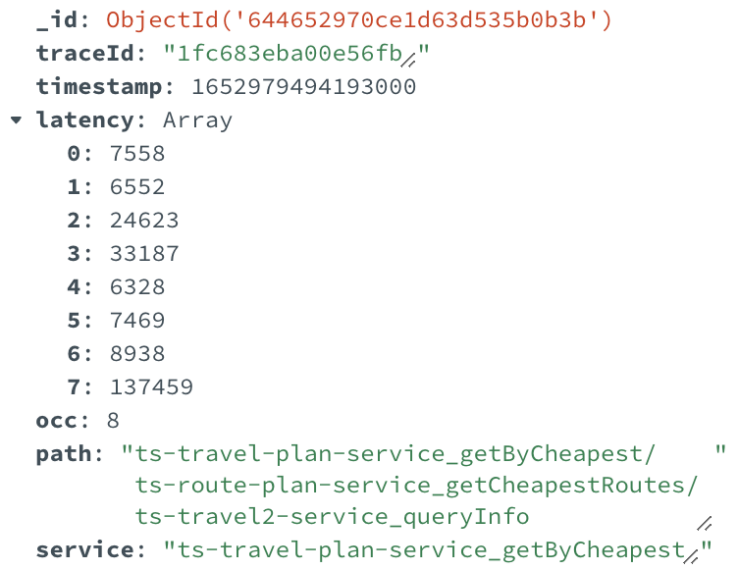


Figure 4.3. Path collection document in MongoDB

We can see that the *'occ'* field contain a number representing the number of occurrences of that RPC execution path in the trace denoted by the traceId field and the *'latency'* field is a list with as many elements as the number of occurrences of the RPC execution path, where each element represent the execution time observed for each occurence of the RPC execution path.

For example in Figure 4.3, the occ field indicates that in the trace identified by the *'traceId'* field there are 8 occurrences of the RPC execution path *"ts-travel-plan-service_getByCheapest/ts-route-plan-service_getCheapestRoutes /ts-travel2-service_queryInfo"* and the 8 values of the latency list respectively represent the latency for each of the 8 observed RPC execution paths.

Figure 4.4. Latency collection document

Similarly, VAMP stores the end-to-end response time values, along with related

information, in a separate MongoDB collection. This information includes the root RPC, the trce ID, the response time value, and the timestamp (Figure 4.4).

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 Figure 4.1, the *Dashboard* app directly query the *Optimized Trace Storage* in order to efficiently generate visualizations.

4.3 Visual Components

Once the data has been collected, it must be presented in a way that provide users with a comprehensive view of the data.

VAMP presents data using two main interactive visualization: a *tree* and a *histogram*.

4.3.1 Tree

This visualization component takes inspiration from the Jaeger comparison tool [55], 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 Figure 4.5.

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 Figure 4.5 the node labeled as $\texttt{RPC}_E$ represents the execution path $\texttt{RPC}_A \rightarrow \texttt{RPC}_B \rightarrow \texttt{RPC}_E$. As can be observed by the figure, the same RPC can appear in multiple nodes (*e.g.,* $\texttt{RPC}_F$), 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 $\texttt{RPC}_A$ invokes the $\texttt{RPC}_D$ multiple times, there

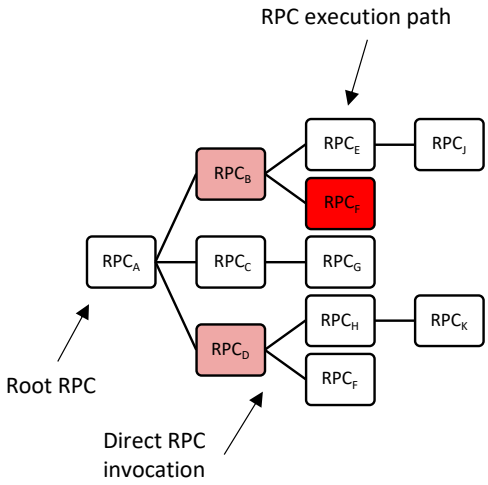


Figure 4.5. Tree

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 Coefficient of Variation (CV) [19], 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 [16], 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 attributes values. The shade of color gradually transitions from white to red as the CV value increases.

4.3.2 Histogram

The VAMP histogram component (shown in 4.6) depicts a traditional distribution plot of the end-to-end response time.

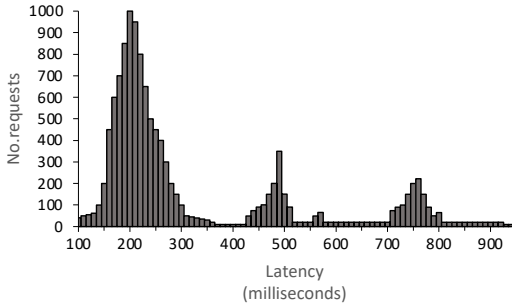


Figure 4.6. Histogram

These kinds of visualizations are frequently used in practice for performance analysis, and are provided by several tools, *e.g.,* Kibana [18]. According to recent research [14], 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 the identification of specific performance behaviors that are worthy of investigation. The user can identify "modes" in the response time distribution, which indicate meaningful recurring performance behaviors , and starts a targeted investigation on these requests, as we will detail in the subsequent subsection.

4.4 Interactions

The core insight behind VAMP is to make explicit the relationship between RPC attribute values, such as the *frequency* of RPC invocation within a request or the associated *execution time*, and the end-to-end response time.

To do this, the tool includes several features and interactions typical of visual

analytics. One of the key interactions is the link between the two main visualizations: the RPC execution tree and the end-to-end response time histogram. This linking allows users to explore the relationship between RPC attributes and response time in both directions. Through direct analysis, users can interact with the tree to examine how specific attribute values, related to a particular RPC execution path, affect end-to-end response time. This type of analysis helps identify which attributes can significantly impact response time and understand how they vary along execution paths. On the other hand, through backward analysis, users can investigate how specific end-to-end response time behaviors are associated with certain RPC attribute values. This helps identify which attributes may be related to which response time patterns and better understand the causes of slower or faster performance. In this case, VAMP provides the brushing interaction, which allows users to highlight a region of interest in the response time histogram. By focusing on a specific area of the histogram, users can explore the RPC attributes associated with that specific region and analyze its characteristics in detail. In addition, VAMP also offers the overview + detail interaction mode. Users can interact with a node of the RPC execution tree to maintain an overview of this visualization while simultaneously viewing detailed information about the specific RPC execution path through different visualizations triggered by the interaction. These different interaction modes allow users to explore the relationship between RPC attributes and end-to-end response time in a flexible and in-depth way, making it easier to analyze and understand the collected data.

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

4.4.1 Forward analysis

Figure 4.7 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

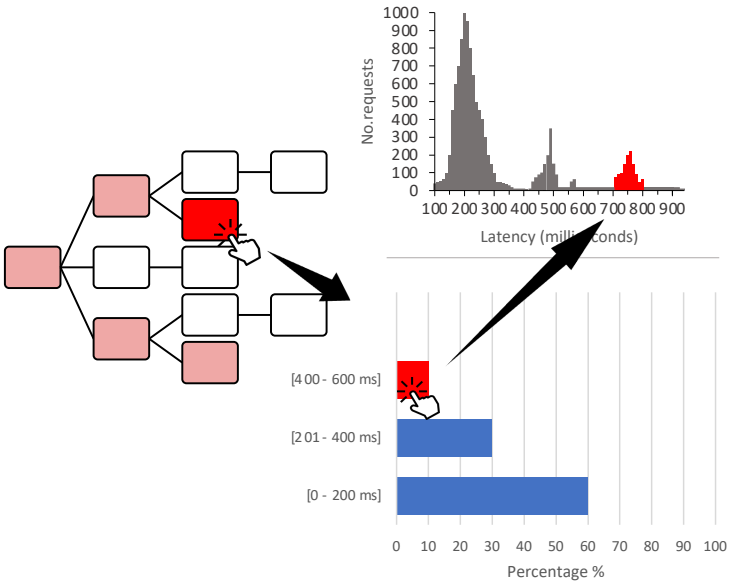


Figure 4.7. Forward Analysis

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 Figure 4.7.

Each bar refer to 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 [39]. 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 [47]. Each bar represents a meaningful recurring execution time behavior, and the user can click on each bar to see how this behavior reflect 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 Figure 4.7, we can observe that when the selected RPC has 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 kind of relationship would have been way more challenging when using present tools. It is worth to notice the same process also applies when analyzing different RPC attributes, such as *occurences*.

4.4.2 Backward analysis

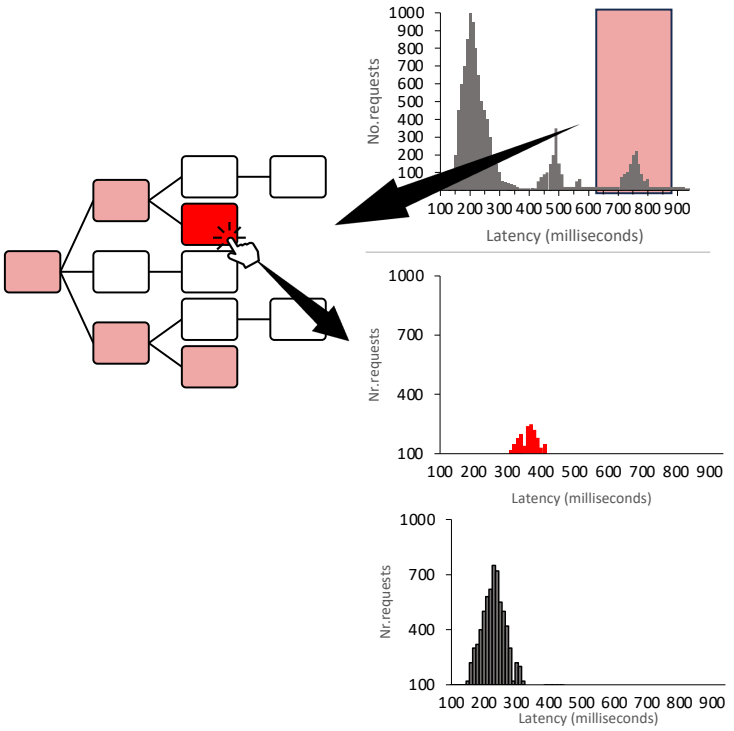


Figure 4.8. Backward Analysis

In the backward analysis, the user can start its investigation directly from the histogram component. The user can select a specific range of end-to-end response time using a slider selector, as shown in Figure 4.8. This selection trigger an update in tree component 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 others requests. A red node indicates that the corresponding RPC execution path show considerably different attributes 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 attributes values, and therefore a weak relation. We quantify the degree of divergence using Kullback-Leibler divergence [9], where values close to 0 indicates nearly identical distributions (white), while values close to ≥ 1 indicates highly different distributions

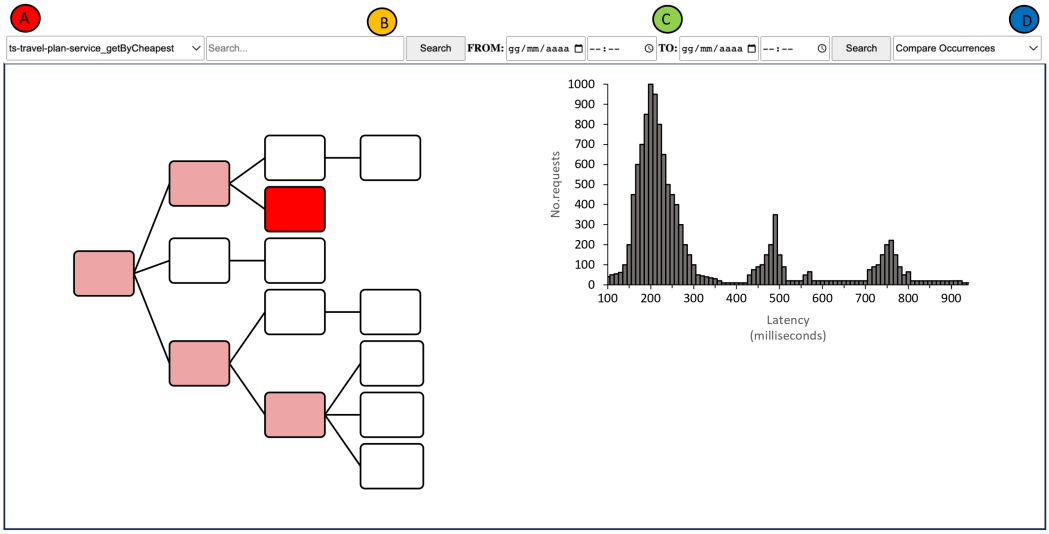


Figure 4.9. VAMP's Dashboard

(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.

4.5 Dashboard

Figure 4.9 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 corners, respectively. The space in the bottom-right is intentionally left blank and will 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

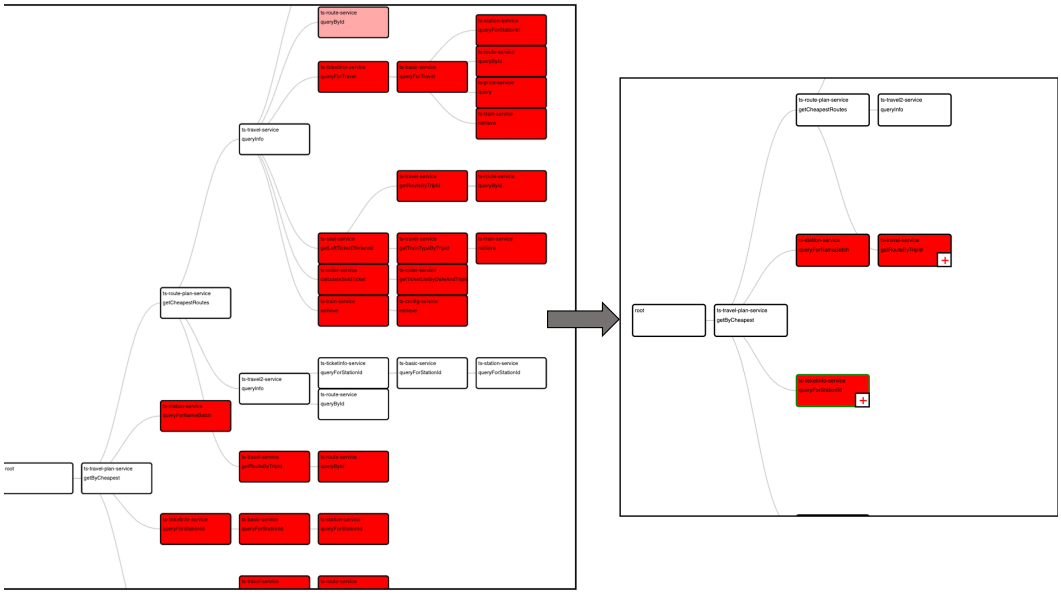


Figure 4.10. Double-click interaction

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 tool, VAMP support 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 corresponding node. As we can see in Figure 4.10, on the root node of the collapsed subtree, will appear a "+" denoting the fact that this node can be expandend. This symbol will have the color of the node in the subtree with greater variability in order to not lose the information that in the collapsed subtree there may be some nodes that show high variability in the

corresppnding attribute values.

4.6 Implementation

VAMP currently supports distributed traces stored in the Jaeger [55] format using Elasticsearch [17] as *Trace Storage* and preprocessed traces stored in MongoDB as *Optimized Trace Storage*, but they can be easily extended to other technologies. The dashboard and visual components have been developed using D3.js [8], which handles the visualization rendering, and Flask [45], which serves as the backend service. The preprocessing scripts are implemented in Python.

Chapter 5

Evaluation

Our experimental evaluation centre around one main research question:

To what extent does VAMP *support performance analysis?*

In this chapter, we first describe the methodology used to gather the answer. Then, we report and discuss the results of the experimental evaluation.

5.1 Methodology

We apply VAMP on 33 datasets generated from TrainTicket[64]. At the time of writing, TrainTicket is the largest and most complex open source microservice-based system (within our knowledge), and it has been widely used in previous software engineering research [56, 61, 36, 60, 22, 65]. TrainTicket provides typical train ticket booking functionalities such as ticket enquiry, reservation, payment, change, and user notification. It involves 41 microservices implemented in four different programming languages (i.e. Java, Python, Node.js, and Go), and it utilizes Jaeger [55] and Elasticsearch [17] for collecting and storing distributed traces.

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. The datasets used in our study fall into two categories.

The first category of datasets is generated using a methodology similar to that presented in [56, 11]. As shown in Figure 5.1, initially, the system's source code is modified to inject random performance issues. Following this, load testing sessions

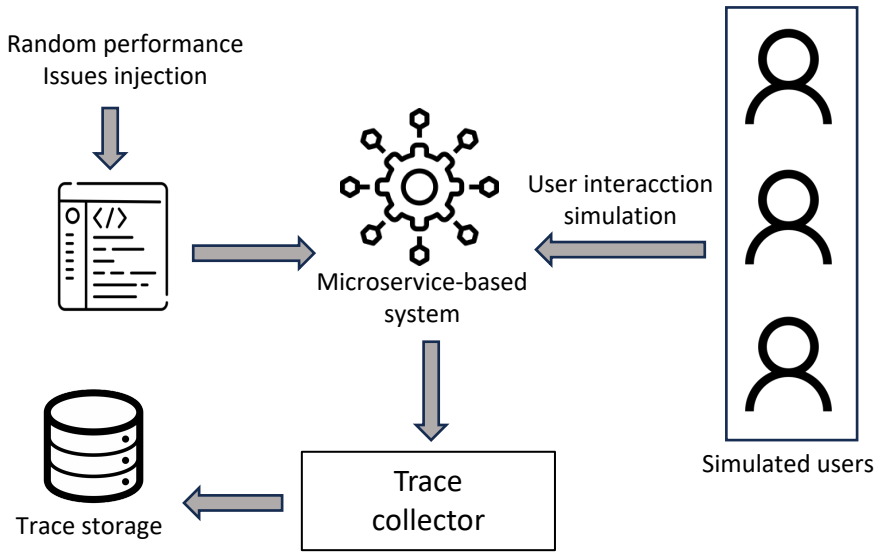


Figure 5.1. First kind dataset generation process

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 [11, 56, 33]. 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 [28]. 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 our previous work [56].

The second kind of datasets does not involve any performance issue injection,

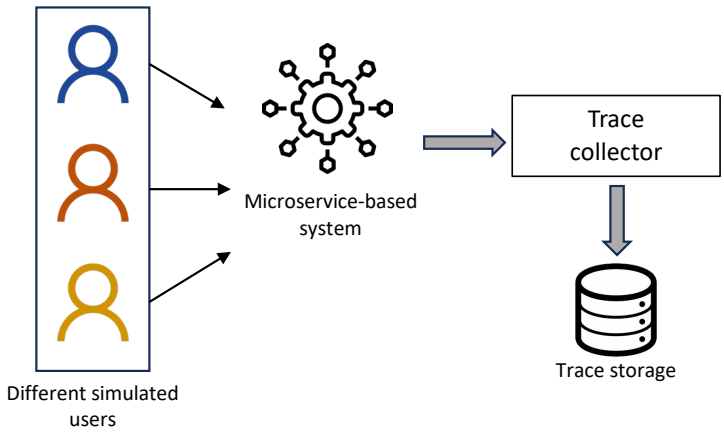


Figure 5.2. Second kind dataset generation process

but it is generated using a more elaborate workload generator as shown in Figure 5.2. Similarly to recent studies [38, 37] 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 user may only visit the homepage and subsequently search trains for some random locations, while others may first login 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 [4]. To this aim, we slightly modified *PPTAM*, 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. This process leads to 13 distinct datasets, each one related to a different API.

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 $1 \leq i \leq 13$.

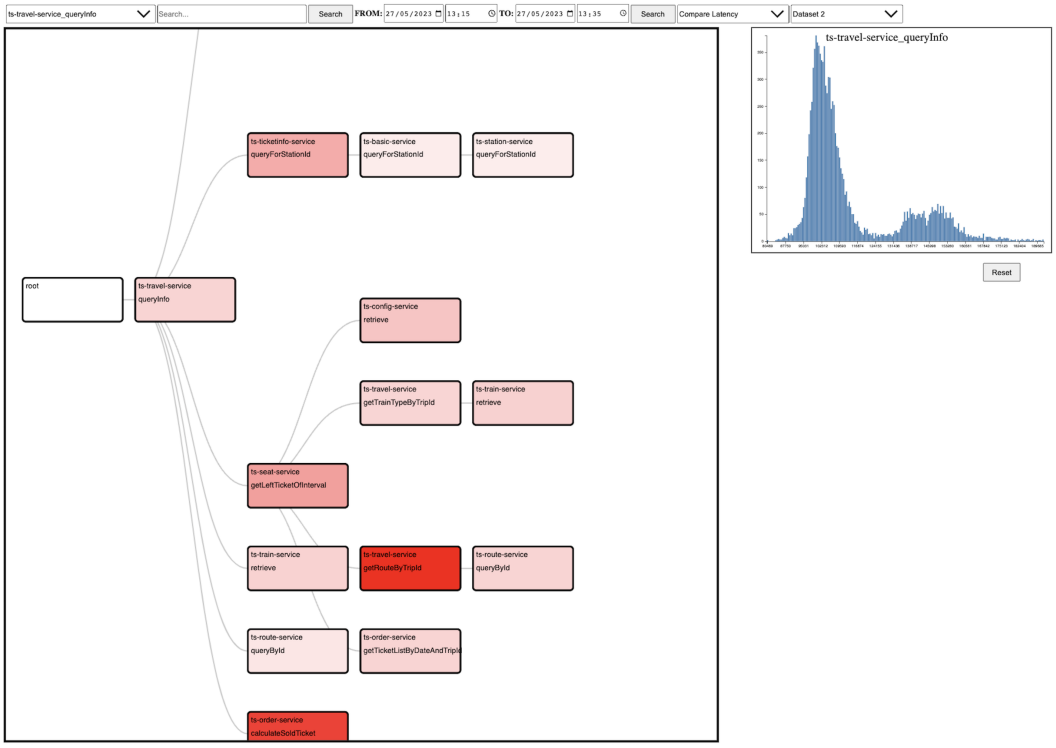
To assess the effectiveness of our approach, two authors conducted manual inspections of the 33 distributed trace datasets using VAMP. For the first type of

dataset, our evaluation focused on determining the extent to which VAMP facilitated the identification of the injected performance issues. Conversely, for the second type of dataset, our evaluation centered around assessing VAMP's ability to support the understanding of specific end-to-end response time behaviors.

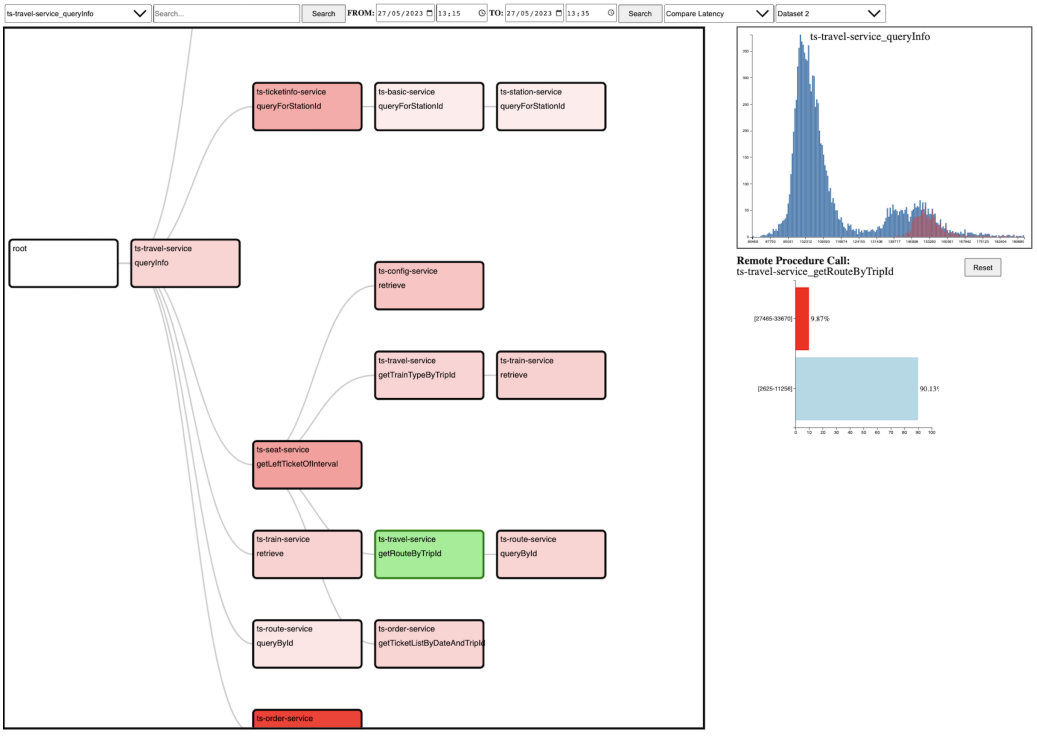
5.2 Results

VAMP has proven to be remarkably effective in identifying performance issues, 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 pinpoint the issues. 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 [35].

Figure 5.3 showcases an example of *forward analysis* using the dataset $\widehat{D}_2$. As depicted in Figure 5.3a, 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 Figure 5.3b reveals that the selected RPC execution path (highlighted in green) exhibits two distinct execution time behaviors: in 9.87% of the requests, the RPC `ts-travel-service_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



(a)



(b)

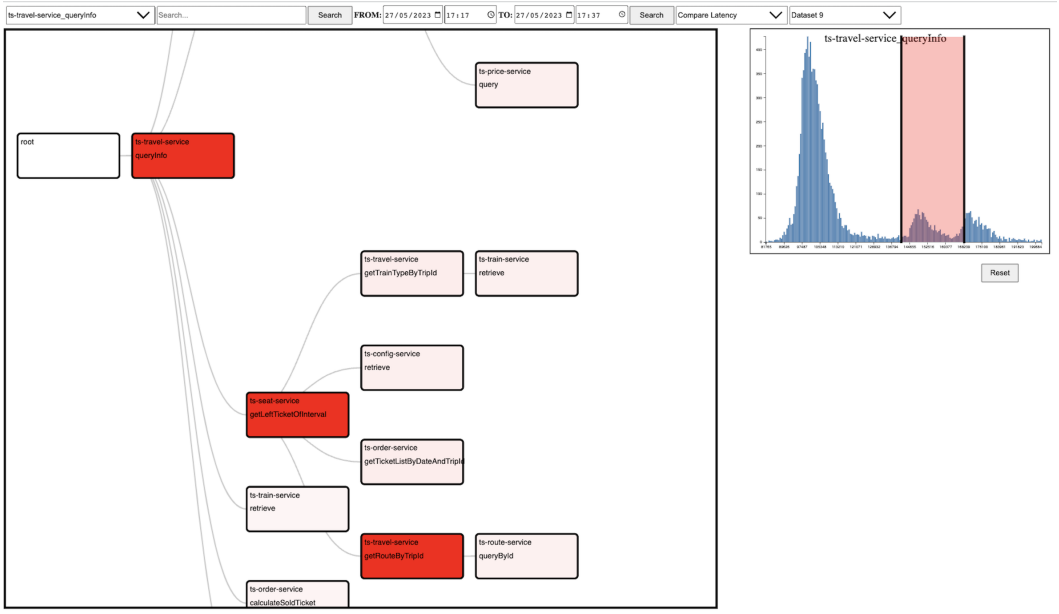
Figure 5.3. *Forward analysis* on *execution time* for dataset $\widehat{D}_2$.

end-to-end response time distribution, as shown by the red highlight in the histogram. Our analysis confirmed that the performance issue had indeed impacted the RPC `ts-travel-service_getRouteByTripId`.

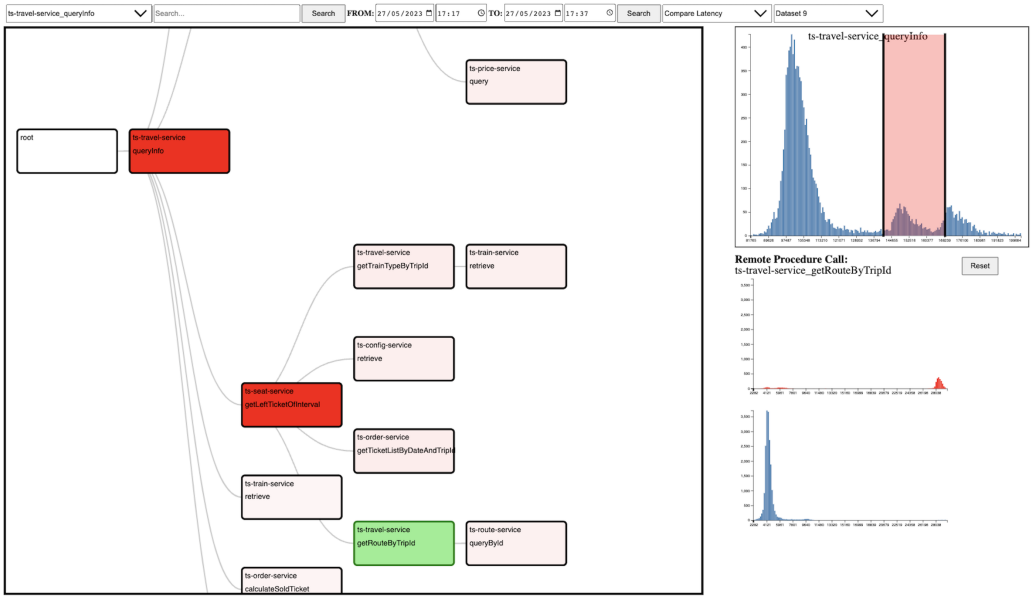
Figure 5.4 offers another example of how VAMP allows users to rapidly identify the RPC responsible for a particular end-to-end response time deviation. Specifically, this figure demonstrates an instance of a VAMP *backward analysis* using the dataset $\widehat{D}_9$. The screenshot in Figure 5.4 shows 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 Figure 5.4b displays the investigation of one of these node (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 `ts-travel-service_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 notable feature of VAMP is its ability to swiftly debunk fluctuations in RPC execution time that have no impact on the end-to-end response time For instance, Figure 5.5 illustrates two distinct execution time behaviors in the selected RPC `ts-order-service_calculateSoldTicket`: one ranging between 33.42 and 55.95 milliseconds, and another between 1.03 and 14.96 milliseconds. Upon inspecting both these behaviors through VAMP, no significant correlation with the end-to-end response time was found. As illustrated in Figures 5.5a and 5.5b, 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 our evaluation, VAMP generally exhibited similar effectiveness with both *forward* and *backward analysis*. However, there are certain scenarios where *forward analysis* proved to be more effective, particularly when multiple performance issues

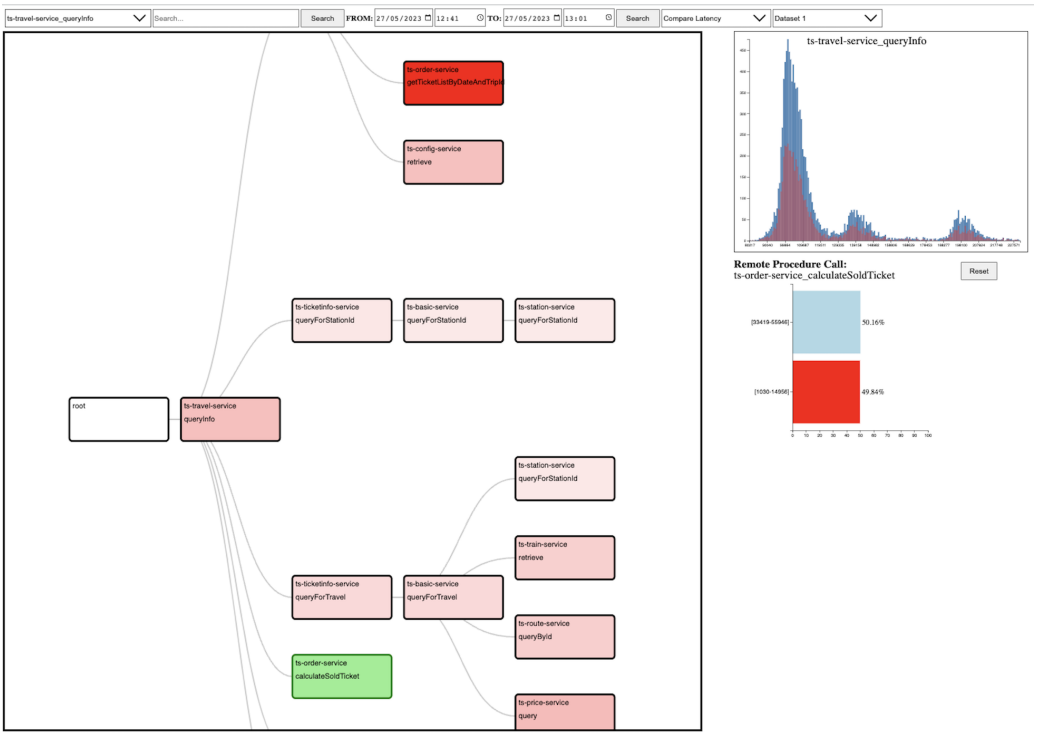


(a)

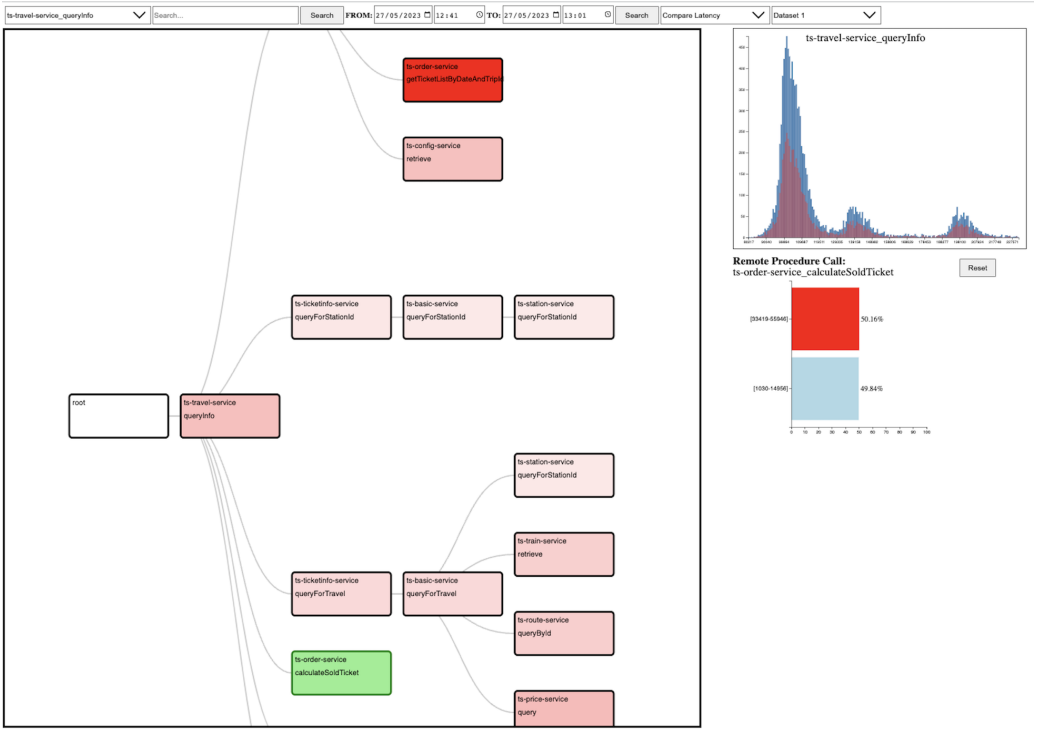


(b)

Figure 5.4. *Backward analysis* on *execution time* for dataset $\widehat{D}_9$.

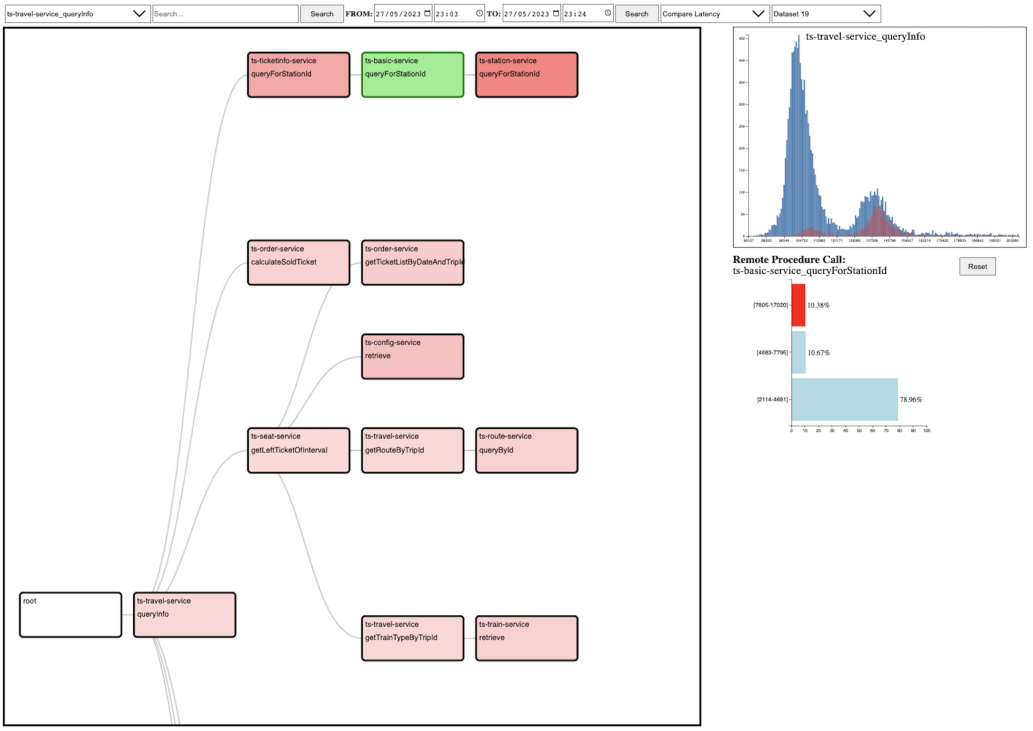


(a)

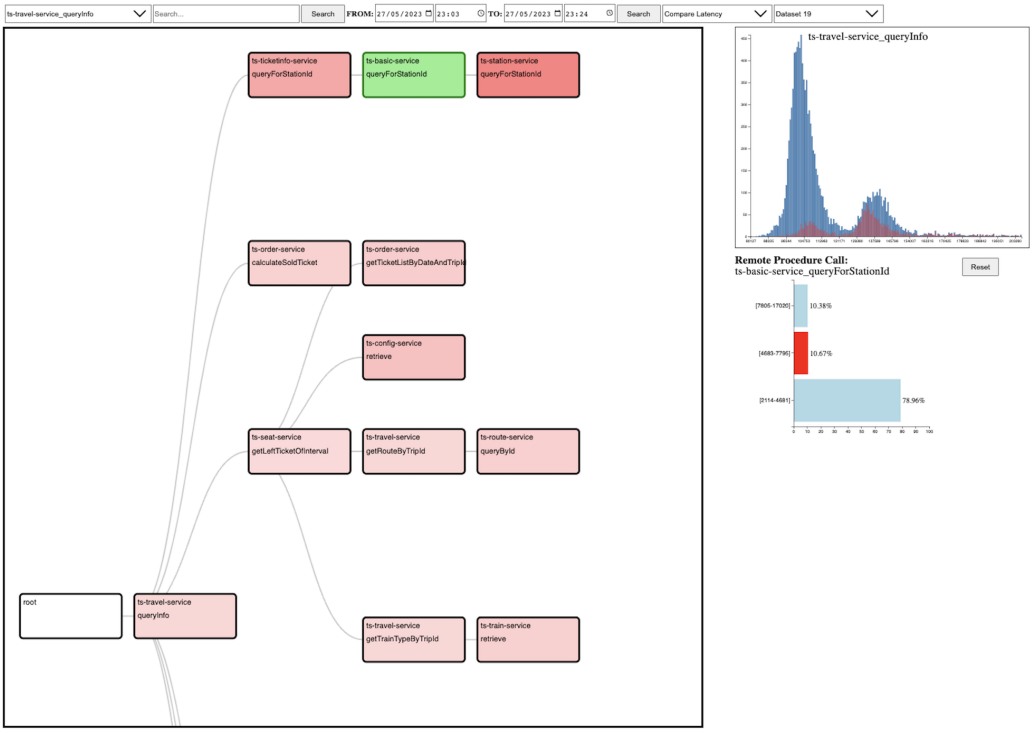


(b)

Figure 5.5. *Forward analysis* on *execution time* for dataset $\widehat{D}_1$.



(a)



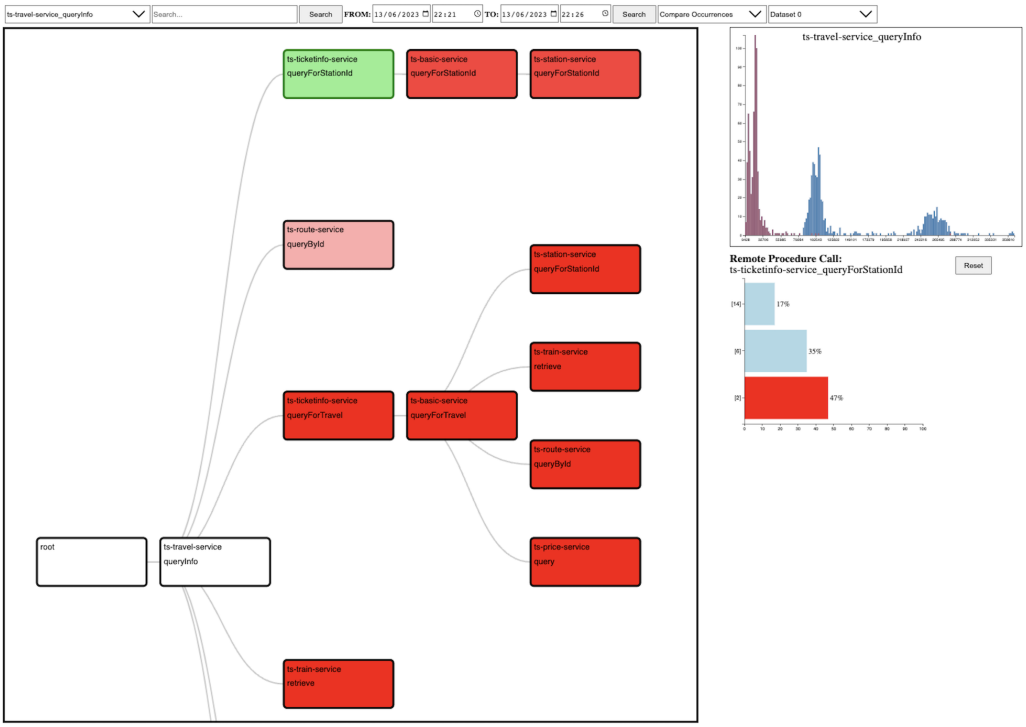
(b)

Figure 5.6. *Forward analysis* on *execution time* for dataset $\widehat{D}_{19}$.

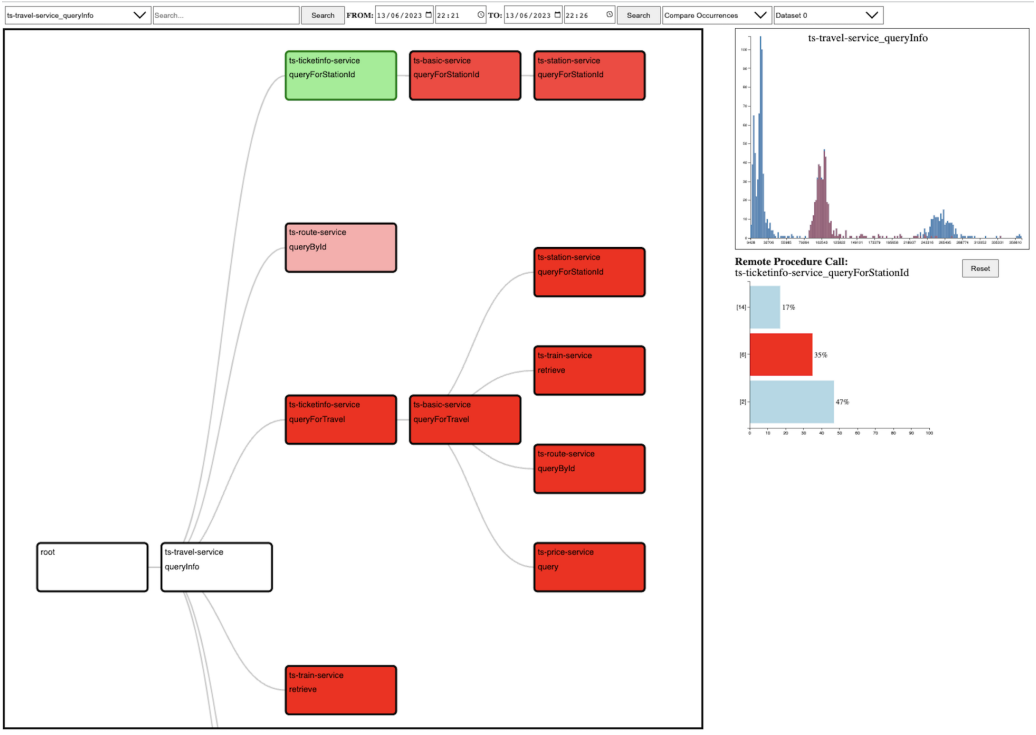
lead to an increase in end-to-end response time that overlaps within the same range. Within our datasets, we encountered two such scenarios, specifically in $\widehat{D}4$ and $\widehat{D}_{19}$. Figure 5.6 showcases an example of *forward analysis* on $\widehat{D}_{19}$. As can be seen, the RPC `ts-basic-service_queryForStationId` (highlighted in green) shows three distinct execution time behaviors. Two of these behaviors lead to shift in the end-to-end response time behavior, which falls in a similar range of the distribution. This can be noticed when examining Figure 5.6a, which highlights the region of the response time distribution when `ts-basic-service_queryForStationId` has execution time ranging between 7.84 and 17.02 milliseconds, and Figure 5.6b, which does the same for execution times between 4.67 and 7.81 milliseconds. Upon a more detailed investigation into one of the RPCs called by `ts-basic-service_queryForStationId`, specifically `ts-station-service_queryForStationId`, we found that one of these behaviors was caused by an issue in the called RPC, while the other was due to an issue within the RPC itself. When using *backward analysis*, identifying these issues proved to be significantly more challenging. This fact suggests that when multiple issues cause an increase in end-to-end response time within the same range, *forward analysis* may be the more suitable approach. However, it is worth noting that these scenarios are typically more complex to analyze and often necessitate a greater number of interactions with the tool.

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 `ts-travel-plan-service_getByCheapest`, while the second dataset, D_2, comprises requests initiated from `ts-travel-service_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. Figure 5.7 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 Figure 5.7c, the right-most

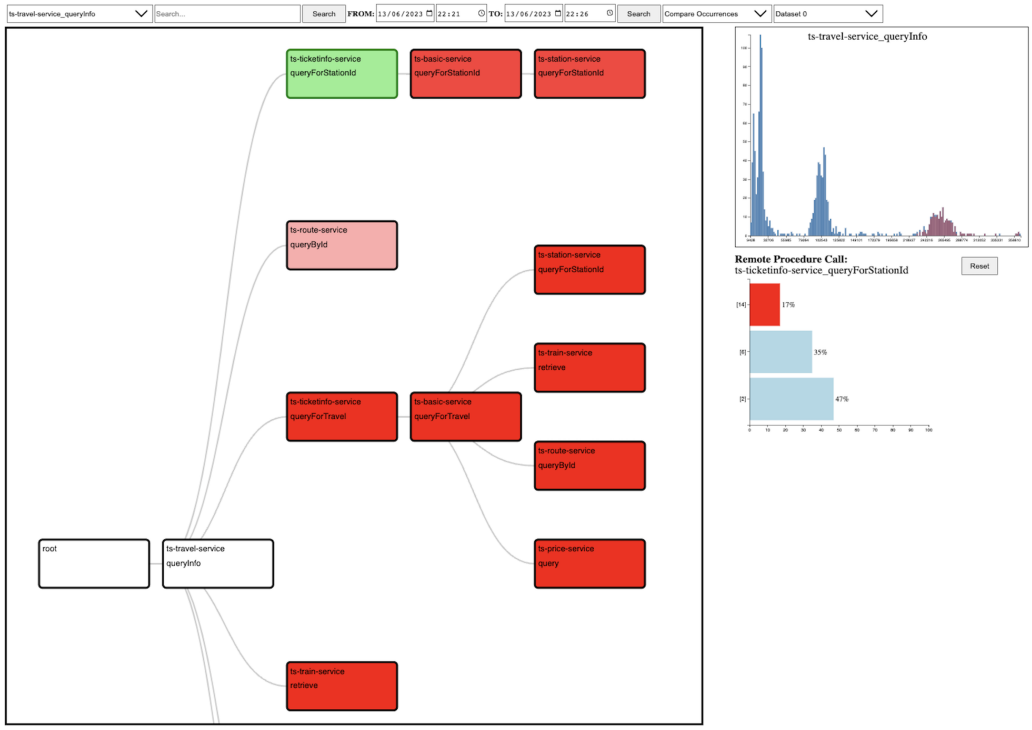
mode is characterized by 14 invocations of the path `ts-travel-service_queryInfo` → `ts-ticketinfo-service_queryForStationId`. Similarly, the center mode is characterized by 6 invocations of this path, while the left-most mode is marked by 2 invocations. Uncovering such patterns using traditional observability tools would have been notably challenging.


ts-travel-service_queryInfo
Search...
Search
FROM: 13/06/2023 22:21
TO: 13/06/2023 22:26
Search
Compare Occurrences
Dataset 0
ts-ticketinfo-service queryForStationId
ts-basic-service queryForStationId
ts-station-service queryForStationId
ts-route-service queryById
ts-station-service queryForStationId
ts-train-service retrieve
ts-ticketinfo-service queryForTravel
ts-basic-service queryForTravel
ts-route-service queryById
root
ts-travel-service queryInfo
ts-price-service query
ts-train-service retrieve
ts-travel-service_queryInfo
Remote Procedure Call:
ts-ticketinfo-service_queryForStationId
Reset
[14]
17%
[8]
35%
[3]
47%

(a)


ts-travel-service_queryInfo
Search...
Search
FROM: 13/06/2023 22:21
TO: 13/06/2023 22:26
Search
Compare Occurrences
Dataset 0
ts-ticketinfo-service queryForStationId
ts-basic-service queryForStationId
ts-station-service queryForStationId
ts-route-service queryById
ts-station-service queryForStationId
ts-train-service retrieve
ts-ticketinfo-service queryForTravel
ts-basic-service queryForTravel
ts-route-service queryById
root
ts-travel-service queryInfo
ts-price-service query
ts-train-service retrieve
ts-travel-service_queryInfo
Remote Procedure Call:
ts-ticketinfo-service_queryForStationId
Reset
[14]
17%
[8]
35%
[3]
47%

(b)



(c)

Figure 5.7. *Forward analysis* on *frequency* for dataset D_2.

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. In both datasets utilized for our evaluation, we successfully disclosed the impact of the frequency of each RPC execution path on end-to-end response time.

Chapter 6

Conclusions

In this thesis, we have analysed the key concept behind the microservices architeture. We have studied how to analyse the performance of this systems and the importance of using distributed tracing tools for this task.

Then we have analysed the limitations of existing tools and, starting from them we have proposed 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. The tool enable at once the analysis of the performances of multiple ent-to-end requests of microservice system eliminating the need to switch between different tools and visualizations. By providing the possibility to interact with the proposed visualizations, VAMP facilitates the analysis of request recurrent characteristics and their relation w.r.t. the end-to.end performance behavior.

To asses the effectiveness of our approach, we have conducted an extensive evaluation using 33 datasets generated from an established open-source microservices system, we show that VAMP can be effectively employed to identify (i) the RPCs impacted by delays, (ii) the variations in RPC execution time caused by the issue, and (iii) the specific range of end-to-end response time related to the issue. Furthermore, our findings demonstrate how VAMP can be used to better understand the relationship between the frequency of specific RPC execution paths and end-to-end response time.

For future work, we aim to broaden the capabilities of VAMP to include additional RPC attributes, such as HTTP headers. Furthermore, 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 [40]. Finally, we plan to deploy VAMP as a service for the SoBigData research infrastructure and community[1].

[1]SoBigData is a research infrastructure that has the goal of enhancing interdisciplinary and innovative research on the multiple aspects of social complexity by combining data and model-driven approaches. SoBigData emphasizes the concept of *responsible data science*. Consequently, SoBigData RI develops methodologies and approaches to put into practice the FAIR (Findable, Accessible, Interoperable, and Reusable) and FACT (Fair, Accurate, Confidential, and Transparent) principles. For additional information, please visit www.sobigdata.eu.

Bibliography

[1] Opentracing specification council: The opentracing data model specification. https://opentracing.io/specification, 2019.

[2] Tarek M. Ahmed, Cor-Paul Bezemer, Tse-Hsun Chen, Ahmed E. Hassan, and Weiyi Shang. 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*, MSR '16, page 1–12, New York, NY, USA, 2016. Association for Computing Machinery.

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

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

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

[6] Andre Bento, Jaime Correia, Ricardo Filipe, Filipe Araujo, and Jorge Cardoso. Automated analysis of distributed tracing: Challenges and research directions. *Journal of Grid Computing*, 19:1–15, 2021.

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

[8] Mike Bostock. D3: Data-driven documents, 2023. "Accessed 2023-01-21 19:45".

[9] Stephen Boyd and Lieven Vandenberghe. *Convex Optimization*. Cambridge University Press, 2004.

[10] Kristin A Cook and James J Thomas. Illuminating the path: The research and development agenda for visual analytics. Technical report, Pacific Northwest National Lab.(PNNL), Richland, WA (United States), 2005.

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

[12] Wenqiang Cui. Visual analytics: A comprehensive overview. *IEEE access*, 7:81555–81573, 2019.

[13] Thomas Davidson and Jonathan Mace. See it to believe it? The role of visualisation in systems research. In *Proceedings of the 13th Symposium on Cloud Computing*, SoCC '22, pages 419–428, New York, NY, USA, November 2022. Association for Computing Machinery.

[14] Thomas Davidson, Emily Wall, and Jonathan Mace. A qualitative interview study of distributed tracing visualisation: A characterisation of challenges and opportunities. *IEEE Transactions on Visualization and Computer Graphics*, pages 1–12, 2023.

[15] Shahir Daya, Nguyen Van Duy, Kameswara Eati, Carlos M Ferreira, Dejan Glozic, Vasfi Gucer, Manav Gupta, Sunil Joshi, Valerie Lampkin, Marcelo

Martins, et al. *Microservices from theory to practice: creating applications in IBM Bluemix using the microservices approach.* IBM Redbooks, 2016.

[16] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communications of the ACM*, 56:74–80, 2013.

[17] Elastic. Elasticsearch: The official distributed search & analytics engine, 2023. "Accessed 2023-01-15 18:38".

[18] Elastic. Kibana: Explore, Visualize, Discover Data, 2023. "Accessed 2023-01-15 17:38".

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

[20] Joe Farro. Trace comparisons arrive in Jaeger 1.7, October 2018. "Accessed 2023-01-15 18:16".

[21] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, et al. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 3–18, 2019.

[22] Xiaofeng Guo, Xin Peng, Hanzhang Wang, Wanxue Li, Huai Jiang, Dan Ding, Tao Xie, and Liangfei Su. Graph-based trace analysis for microservice architecture understanding and problem diagnosis. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, page 1387–1397, New York, NY, USA, 2020. Association for Computing Machinery.

[23] IBM. Instana: Automated Observability, 2023. "Accessed 2023-01-16 14:55".

[24] Cisco Systems Inc. Appdynamics: Observability Platform, 2023. "Accessed 2023-01-16 14:53".

[25] Dynatrace Inc. Dynatrace: Modern cloud done right, 2023. "Accessed 2023-01-16 14:50".

[26] Dynatrace Inc. Dynatrace. service flow, 2023. "Accessed 2023-02-14 14:10:59".

[27] Zhen Ming Jiang and Ahmed E Hassan. A survey on load testing of large-scale software systems. *IEEE Transactions on Software Engineering*, 41(11):1091–1118, 2015.

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

[29] 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. Canopy: An End-to-End Performance Tracing And Analysis System. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 34–50, New York, NY, USA, 2017. Association for Computing Machinery.

[30] Daniel A Keim, Florian Mansmann, Daniela Oelke, and Hartmut Ziegler. Visual analytics: Combining automated discovery with interactive visualizations. In *Discovery Science: 11th International Conference, DS 2008, Budapest, Hungary, October 13-16, 2008. Proceedings 11*, pages 2–14. Springer, 2008.

[31] Daniel A Keim and Matthew O Ward. Visual data mining techniques. 2002.

[32] Muzammil Khan and Sarwar Shah Khan. Data and information visualization methods, and interactive mechanisms: A survey. *International Journal of Computer Applications*, 34(1):1–14, 2011.

[33] Darja Krushevskaja and Mark Sandler. Understanding latency variations of black box services. In *Proceedings of the 22nd International Conference on World Wide Web*, WWW '13, page 703–714, New York, NY, USA, 2013. Association for Computing Machinery.

[34] Christoph Laaber and Philipp Leitner. An evaluation of open-source software microbenchmark suites for continuous performance assessment. In *Proceedings*

of the 15th International Conference on Mining Software Repositories, MSR '18, page 119–130, New York, NY, USA, 2018. Association for Computing Machinery.

[35] Jessica Leone, Luca Traini, Giovanni Stilo, and Antinisca Di Marco. Vamp - replication package. https://github.com/Jessicaleone/VAMP, 2023.

[36] Bowen Li, Xin Peng, Qilin Xiang, Hanzhang Wang, Tao Xie, Jun Sun, and Xuanzhe Liu. Enjoy your observability: an industrial survey of microservice tracing and analysis. *Empirical Software Engineering*, 27(1):25, 2021.

[37] Lizhi Liao, Jinfu Chen, Heng Li, Yi Zeng, Weiyi Shang, Jianmei Guo, Catalin Sporea, Andrei Toma, and Sarah Sajedi. Using black-box performance models to detect performance regressions under varying workloads: An empirical study. *Empirical Softw. Engg.*, 25(5):4130–4160, sep 2020.

[38] Lizhi Liao, Jinfu Chen, Heng Li, Yi Zeng, Weiyi Shang, Catalin Sporea, Andrei Toma, and Sarah Sajedi. Locating performance regression root causes in the field operations of web-based systems: An experience report. *IEEE Transactions on Software Engineering*, pages 1–1, 2021.

[39] S. Lloyd. Least squares quantization in pcm. *IEEE Transactions on Information Theory*, 28(2):129–137, 1982.

[40] Shutian Luo, Huanle Xu, Chengzhi Lu, Kejiang Ye, Guoyao Xu, Liping Zhang, Yu Ding, Jian He, and Chengzhong Xu. Characterizing microservice dependency and performance: Alibaba trace analysis. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 412–426, 2021.

[41] Jonathan Mace. End-to-End Tracing: Adoption and Use Cases. Survey, Brown University, 2017.

[42] C. Majors, L. Fong-Jones, and G. Miranda. *Observability Engineering: Achieving Production Excellence*. O'Reilly Media, Incorporated, 2022.

[43] Sam Newman. *Building microservices*. " O'Reilly Media, Inc.", 2021.

[44] Charlene O'Hanlon. A conversation with werner vogels: Learning from the amazon technology platform: Many think of amazon as' that hugely successful online bookstore.'you would expect amazon cto werner vogels to embrace this distinction, but in fact it causes him some concern. *Queue*, 4(4):14–22, 2006.

[45] Pallets. Web development, one drop at time, 2023. "Accessed 2023-01-21 19:52".

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

[47] Peter J. Rousseeuw. Silhouettes: A graphical aid to the interpretation and validation of cluster analysis. *Journal of Computational and Applied Mathematics*, 20:53–65, 1987.

[48] Julia Rubin and Martin Rinard. The challenges of staying together while moving fast: An exploratory study. In *Proceedings of the 38th International Conference on Software Engineering*, pages 982–993, 2016.

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

[50] Raja R Sambasivan, Ilari Shafer, Michelle L Mazurek, and Gregory R Ganger. Visualizing request-flow comparison to aid performance diagnosis in distributed systems. *IEEE transactions on visualization and computer graphics*, 19(12):2466–2475, 2013.

[51] Raja R. Sambasivan, Alice X. Zheng, Michael De Rosa, Elie Krevat, Spencer Whitman, Michael Stroucken, William Wang, Lianghong Xu, and Gregory R. Ganger. Diagnosing performance changes by comparing request flows. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, page 43–56, USA, 2011. USENIX Association.

[52] Ben Shneiderman. The eyes have it: A task by data type taxonomy for information visualizations. In *Proceedings 1996 IEEE symposium on visual languages*, pages 336–343. IEEE, 1996.

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

[54] Cindy Sridharan. Testing microservices, the sane way, December 2017. "Accessed 2023-01-16 11:14".

[55] Uber Technologies. Jaeger: Open source, end-to-end distributed tracing, 2023. "Accessed 2023-01-15 14:10:59".

[56] Luca Traini and Vittorio Cortellessa. Delag: Using multi-objective optimization to enhance the detection of latency degradation patterns in service-based systems. *IEEE Transactions on Software Engineering*, pages 1–28, 2023.

[57] Luca Traini, Vittorio Cortellessa, Daniele Di Pompeo, and Michele Tucci. Towards effective assessment of steady state performance in java software: are we there yet? *Empirical Software Engineering*, 28(1):13, 2022.

[58] Luca Traini, Daniele Di Pompeo, Michele Tucci, Bin Lin, Simone Scalabrino, Gabriele Bavota, Michele Lanza, Rocco Oliveto, and Vittorio Cortellessa. How software refactoring impacts execution time. *ACM Trans. Softw. Eng. Methodol.*, 31(2), dec 2021.

[59] Kaushik Veeraraghavan, Justin Meza, David Chou, Wonho Kim, Sonia Margulis, Scott Michelson, Rajesh Nishtala, Daniel Obenshain, Dmitri Perelman, and Yee Jiun Song. Kraken: Leveraging live traffic tests to identify and resolve resource utilization bottlenecks in large scale web services. In *OSDI*, volume 16, pages 635–651, 2016.

[60] Tianyi Yang, Jiacheng Shen, Yuxin Su, Xiao Ling, Yongqiang Yang, and Michael R. Lyu. Aid: Efficient prediction of aggregated intensity of dependency

in large-scale cloud systems. In *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering*, ASE '21, page 653–665. IEEE Press, 2022.

[61] Chenxi Zhang, Xin Peng, Chaofeng Sha, Ke Zhang, Zhenqing Fu, Xiya Wu, Qingwei Lin, and Dongmei Zhang. Deeptralog: Trace-log combined microservice anomaly detection through graph-based deep learning. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, pages 623–634, 2022.

[62] Xu Zhao, Kirk Rodrigues, Yu Luo, Michael Stumm, Ding Yuan, and Yuanyuan Zhou. Log20: Fully automated optimal placement of log printing statements under specified overhead threshold. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 565–581, 2017.

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

[64] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chao Ji, Wenhai Li, and Dan Ding. Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study. *IEEE Transactions on Software Engineering*, 47(2):243–260, 2021.

[65] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chao Ji, Dewei Liu, Qilin Xiang, and Chuan He. 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*, pages 683–694, 2019.