FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Spectrum-based Fault Localization for Microservices via Log Analysis

João Miguel Ribeiro de Castro Silva Martins

Mestrado em Engenharia Informática e Computação

Supervisors: Prof. Jácome Cunha and Prof. Rui Maranhão

July 25, 2022

Spectrum-based Fault Localization for Microservices via Log Analysis

João Miguel Ribeiro de Castro Silva Martins

Mestrado em Engenharia Informática e Computação

Approved in oral examination by the committee:

President: Prof. José Carlos Medeiros de Campos
External Examiner: Alexandre Campos Perez, PhD
Supervisor: Prof. Jácome Miguel Costa da Cunha

July 25, 2022

Abstract

Microservices is a distributed software architecture that divides the business logic of an application into more minor, fine-grained services. Due to their distributed nature, microservices have implementation challenges that make their application (and maintainability) non-trivial. One of the main issues with microservices is debugging; their differences from traditional distributed systems make the process even harder. Our focus is on developing a novel approach to the Spectrum-based Fault Localization (SFL) technique adapted to the context of microservices. Traditionally, this technique analyzes components of the code, statements, and its execution or non-execution in passing and failing test cases. From there, a coefficient algorithm is applied to each component to determine the probability of being faulty. Then the components are ranked by that probability, the most likely component to be faulty being placed at the top.

Debugging microservices is not a trivial task, and the current developments are still limited in providing time-saving solutions to developers, who still spend a lot of time manual debugging when a failure occurs. Finding a feasible solution that is accurate and capable of dealing with information density is a challenge that the complex nature of microservices imposes. Moreover, current approaches to debugging microservices are limited and often focus on monitoring performance metrics or tracing techniques that require further manual analysis. Besides that, we conclude that a combination of techniques might be beneficial to tackle the problem at hand.

Our work is towards a novel approach for debugging microservices through SFL via Log Analysis. We achieve this by using the traditional SFL technique and adapting it to the context of microservices by introducing the concept of hierarchy entities, in this case, composed of two levels, service and method entity. We obtain this information from the logs generated by the microservices, which are collected and processed. Since we cannot ensure the completeness of logs, we created the two-leveled entity hierarchy to capture valuable information. From there, the SFL technique works, as usual, analyzing the number of times each entity is or is not executed in good and faulty scenarios and computing the rank of entities with the highest probability of being faulty at the top.

The evaluation consisted of executing our tool with logs from a preexistent microservice application. We created a log processing configuration and a suite of scenarios with various faults we injected. As the selected application did not offer complete information in its logs, a common occurrence in real-life applications, it required several adaptations. We measured the tool's accuracy in each scenario given by the expected faulty entities' positions in the rank. With the application tested, we achieved a 68% global accuracy, with several scenarios above 90% accuracy, which tells us there is potential for improvement and better accuracy in scenarios with complete information.

Keywords: microservices, debugging, log analysis, spectrum-based fault localization, SFL

Resumo

Microserviços é uma arquitetura de *software* distribuído que divide a lógica de negócio de uma aplicação em serviços de menor granularidade. Devido à sua natureza distribuída, os microserviços têm os seus próprios desafios de implementação que tornam a sua aplicabilidade (e manutenção) um problema não trivial. Um dos maiores problemas com microserviços é o *debugging*; as diferenças que entre estes e sistemas distribuídos tradicionais têm tornam esse processo ainda mais difícil. O nosso foco é desenvolver uma abordagem nova da técnica de *Spectrum-based Fault Localization* (SFL) adaptada ao contexto de microserviços. Tradicionalmente, esta técnica analisa componentes do código, *statements*, e a sua execução e não execução em testes passados e falhados. Daí, um algoritmo de coeficientes é aplicado a cada componente para determinar a probabilidade de ter falhas. Posteriormente os componentes são ordenados por essa probabilidade, sendo o componente com maior probabilidade de ter falhas colocado no topo do *ranking*.

Debugging de microserviços não é uma tarefa trivial, e os desenvolvimentos atuais são ainda limitados em fornecer soluções que poupem tempo aos programadores, que ainda perdem tempo em debugging manual quando uma falha ocorre. Encontrar uma solução viável que seja exata e capaz de lidar com densidade de informação é um desafio que a natureza complexa de microserviços impõe. Além disso, abordagens atuais para fazer debugging de microserviços são limitadas e frequentemente focam-se na monitorização de métricas de *performance* ou técnicas de *tracing* que requerem análise manual adicional. Por fim concluímos que a combinação de técnicas poderá ser benéfico para enfrentar o problema em mãos.

O nosso contributo é no sentido de implementar uma abordagem nova para *debugging* microserviços por *SFL* via análise de *logs*. Para isso introduzimos o conceito de entidades hierárquicas, neste caso compostas de dois níveis, entidade de serviço e método. Obtemos a informação necessária para criar as entidades dos *logs* gerados pelos microserviços, recolhidos e processados. Uma vez que não é possível assegurar que os *logs* são e estão completos, criamos a hierarquia de dois níveis nas entidades para obter toda a informação relevante. A partir daí a técnica de *SFL* funciona como normalmente, analisando o número de vezes que cada entidade é ou não executada em cenários com e sem falhas, calculando o *ranking* das entidades com a entidade com maior probabilidade de ter falhas no topo.

A avaliação consiste em executar a nossa ferramenta com *logs* originados de uma aplicação baseada em microserviços pré-existente. Criamos uma configuração para processar os *logs* e uma bateria de cenários com um número variável de falhas injetadas manualmente. Como a aplicação selecionada não fornece informação completa nos seus *logs*, o que é uma ocorrência comum em aplicações usadas em cenários reais, foi necessário fazer várias adaptações. Medimos a exatidão da ferramenta em cada cenário baseada na posição das entidades com falhas conhecidas no *ranking*. Com a aplicação testada, obtivemos uma eficácia global de 68%, com vários cenários com exatidão superior a 90%, confirmando o potencial para melhorias e uma melhor eficácia em cenários com

informação completa.

Palavras-chave: microserviços, *debugging*, análise de *logs*, *spectrum-based fault localization*, *SFL*

Acknowledgements

First, I would like to thank my supervisors, Professors Jácome Cunha and Rui Maranhão, for all the support and counsel they provided me during the dissertation. Furthermore, a special thanks to Professor Rui Maranhão for introducing the Thesis idea to me, which allowed me to learn quite a lot about debugging.

To my parents, for supporting my interests from day one, in more ways than one, and always making sure they were present in all my endeavors, an endless thank you. Like many of my accomplishments in life, you also deserve credit for this.

A special mention to a special person, Ana: as we started college together, you helped me grow and become a much better person. A huge thank you for "sticking" with me through these five incredible, however challenging, years of university.

Finally, thank my friends for doing a great job of making my life happier. I will always be glad to have them in my life. A big thank you from my heart.

João Martins

"Yeah, Mr. White! Yeah, Science!"

Jesse Pinkman

Contents

List of Figures

List of Tables

List of Listings

Abbreviations

API	Application Programming Interface
APM	Application Performance Monitoring
CD	Continuous Delivery
CI	Continuous Integration
CPU	Central Processing Unit
DAG	Directed Acyclic Graph
DNS	Domain Name System
eBPF	extended Berkeley Packet Filter
GDB	GNU Debugger
GNU	GNU's Not Unix
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
KPI	Key Performance Indicator
PID	Process Identifier
QoS	Quality-of-Service
QR	Qualitative Reasoning
REST	REpresentional State Transfer
RPC	Remote Procedure Call
SFL	Spectrum-based Fault Localization
SOA	Service Oriented Architecture
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
VPN	Virtual Private Network

Chapter 1

Introduction

This introductory chapter presents the context and motivation for the problem at hand. In addition, we also lay out the general goals we expect to accomplish, issues to face, and main results obtained. In the end, we present the layout of this document.

1.1 Context

With fast-paced environments around software development, new tools and techniques emerge every day to face the challenges of building scalable and highly complex systems. *Microservices* is a distributed software architecture that originated from the necessity of dealing with the complexity of giant monoliths, as their tight-coupling, which makes maintenance and introducing changes difficult [21].

In sum, a microservice is a minimal component of an application that executes a particular task independently from other services, making the development and deployment of changes more effortless and scale better [34]. Of course, its issues and drawbacks make this solution unsuitable for some applications. One is debugging these systems, which requires different approaches than the traditional monoliths.

Debugging is a process essential to software development and consists of inspecting and finding an issue in the software, and it is something every programmer does. Many techniques are dedicated to different kinds of software and architectures [27].

"Every coin is two-faced," and challenges come with the advantages that microservices bring. One of them is how to debug these systems when an error occurs. As a relatively recent approach, the diversity of tools to assist this process still allows for research and new findings.

1.2 Motivation

Given the growing usage of the microservices architecture, creating and maintaining these growing systems requires more resources. Besides the monetary cost of building or buying tools that help debug the systems, there is the time spent developing, configuring, and using them. When

the number of services involved increases, the number of hours dedicated to localizing the fault increases, as surveyed by Zhou *et al*. [113]. Therefore, manually debugging becomes a severe "strain on developers' shoulders", usually with time constraints.

There are several techniques dedicated to debugging distributed systems [8] but less dedicated to microservices. In Chapter 3 (p. 17) contributions deemed relevant are explored and analyzed.

Although the application code is divided into several services, there are issues like the high network load, the dispersity of the services across different physical machines, and the parallel execution of instances for the same service. These are just some examples to demonstrate how the issue is not trivial.

1.3 Goals

Our main goal is to present a novel solution for fault localization based on log analysis and the spectrum-based fault localization (SFL) technique. In Chapter 2 (p. 5) and Chapter 3 (p. 17) we explain and exemplify the concepts here mentioned and the rationale behind using them.

In essence, the idea of our solution is to leverage the information provided by the system logs and process it into a standard format. From there, that structured log data can feed a technique developed based on SFL to obtain fault localization of components in the application code.

As the setting of microservices is usually complex systems, the tools used should interfere as little as possible with the system in production. Besides, as the scale of the products is significant, it is also crucial for the cost of these tools to be inexpensive and not become a liability. From this general perspective of the system, we target these goals:

Minimal intrusion:
Ideally, any tool the developers use to debug should not make any changes to the codebase, as it could be an entry point for *bugs* itself. If this goal is unattainable, the design of the solution should strive for minimal intrusion in the code.

Agnostic:
The solution should be agnostic of the microservice implementation. Microservices benefit from being seamlessly polyglot, so it only makes sense that our solution is as generic as possible.

High Accuracy:
The more accurate our solution is, the better. We want to pinpoint failures with the highest degree of granularity, from the microservice (calls it makes) to the class, the function, and ultimately, the line. This is, of course, the best outcome and, simultaneously, the biggest challenge.

Inexpensive:
Developers should debug their programs without worrying about their impact on the costs

both in time and computational resources (CPU usage, RAM). The objective of any debugging tool is to directly or indirectly save the developer's time; hence, the cost of deploying it should not outweigh the time saved. The proposed solution should then not become a liability with its costs. Furthermore, it should be inexpensive so as not to overload the network or delay the communications between services.

Scalable:
Since it is a hard requirement for microservices to be scalable, we aim for our solution to be scalable. Without this property, it becomes an obstacle for the developer and, therefore, quickly discarded.

The principal goals to be achieved are the **minimal intrusion**, **agnostic-oriented implementation**, and **high accuracy** of our tool. These are the minimal requirements to achieve success in our solution. The other goals, **inexpensive** and **scalable**, come as a secondary priority. The rationale is that the first three are essential to the tool's functionality, i.e., correctly pinpointing failures in generic microservices. On the other hand, the last two are not so important to achieve that, despite being essential for having the tool ready to be used in a production-level environment.

1.4 Issues

Developing microservices and debugging, in general, have associated issues, and the debugging of microservices is no exception. The particular scenario of microservices architecture raises issues in developing our debugging tool. We highlight the following:

Complexity:
Microservices are built for complex systems, so debugging them is as complex or more. The tools for debugging are then required to deal with this complexity.

Scalability:
Scalability is one of the goals of microservice architecture. Thus, any tool built for it must satisfy this goal to be usable.

Density:
Due to the high cardinality of services, the number of metrics and logs generated are a challenging obstacle. The tool needs to be prepared to filter and separate irrelevant data in each specific instance.

Infrastructure:
Starting from scratch is not feasible in our scope. Therefore, we must rely on existing infrastructures. We will build upon that infrastructure to achieve our goals, so finding the appropriate basis for our solution is essential to the success of our work. This constraint applies to the log processing mechanism and the evaluation process.

Log Processing:

Our tool requires processing logs from microservices. Due to the diversity of logs used, we must face the issue of being able to process logs without a standardized format.

Technique Incompatibility:

As explained in Chapter 3 (p. 17), the traditional approach to SFL is not well-suited to deal with executing microservices. Therefore, one of the main implementation issues is to find a way to adapt the technique to the context of microservices and leverage the log data to achieve it.

1.5 Main Results

The results we obtained are further explored in Chapter 5 (p. 40) and Chapter 6 (p. 62). To summarize, we tested the tool in a microservice-based application that provided logs with incomplete information. The main results show that we obtain an average 68% accuracy in detecting faulty components in the code, considering a variety of scenarios. We conclude that the results, while not excellent, reveal the potential for improvement and better results with scenarios with complete information.

1.6 Document Structure

The structure of the document is as follows:

- Introduction 1: In this chapter, the context of debugging microservice applications is presented, the motivation for a novel solution for fault localization, and its main objectives.
- Background 2: So that further information and concepts are provided, this chapter goes into detail regarding the two main topics of this Thesis, microservices and debugging.
- Related Work 3: This chapter explores the current state of the art regarding solutions for microservice debugging, analyzing the presented approaches, traits, and gaps.
- Debugging Tool Implementation 4: This chapter details the architecture and concrete concepts applied in the implementation of the debugging tool prototype and the concrete design choices used.
- Evaluation 5: This chapter is about presenting the evaluation setup prepared for the tool. Moreover, it is about presenting and analyzing the results in depth.
- Conclusions 6: This closing chapter concludes this dissertation with the withdrawn conclusions from the completed work and the main points to focus on in future work.

Chapter 2

Background

The Background chapter introduces fundamental concepts required to have in the context of the problem. The first part of this chapter is dedicated to diving into microservices' concepts and main ideas and explaining some of their advantages and challenges. Following that, we provide a general overview of debugging to contextualize some of the main techniques used for debugging.

2.1 Microservices

As introduced in Chapter 1 (p. 1), microservices are a contemporary architecture that divides an application *"into a set of fine-grained services, which can be independently developed, tested, and deployed"* [54]. Fowler and Lewis [21] characterize microservices architecture style as *"an approach to developing a single application as a suite of small services, each running in its process and communicating with lightweight mechanisms, often an HTTP resource API"*. In the following subsections, we present the motivation and origins of microservices, along with their main issues and usages in the industry.

2.1.1 Monoliths

Monoliths are applications that follow the traditional architecture of encapsulating the whole logic of the application in a single service. When it comes to scaling, the whole is replicated. For a long time, this architecture fit customers' needs as they were small. However, as applications grew, this architecture started showing its flaws. The high complexity, poor reliability, and limited scalability have created the need for a change that microservices would come to satisfy. Figure 2.1 displays a general view of the monolithic architecture.

The drawbacks of giant monoliths [87] can be firstly seen in the large codebases, which are intimidating, especially for new developers in a team. The code becomes hard to understand and modify, slowing development speed. Since modularity is not a hard requirement in monoliths, it eventually breaks down, increasing the coupling of the application. Also, it takes more and more time to start up the containers or components required to develop or test the application, which considerably impacts the developer's productivity.

It is tricky with large monoliths if a team has a rule of frequently deploying the application. Since they are tightly coupled, the redeployment of just one component implies the redeployment of the whole application. For example, for a team that does daily deployments, deploying an extensive application might take the whole day and render the frequent deployment useless. There is also the case when an error occurs in a single application. A quick fix must be deployed to correct the issue quickly, which becomes impossible if the whole application is redeployed.

As mentioned, scaling monoliths is troublesome because their architecture is not suited to deal with the increasing data volume. Monoliths can only scale in one dimension, horizontally, which means running copies of the application to handle the traffic volume. However, each copy accesses the same data, interfering with caching, memory consumption, and I/O usage.

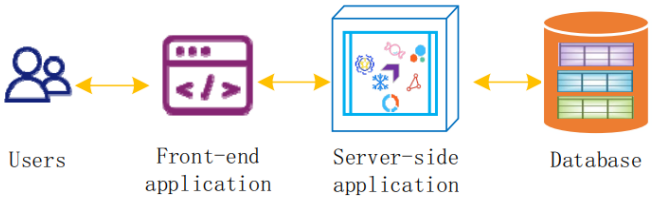

Figure 2.1: Monolithic Architecture [54]

2.1.2 SOA

Service-Oriented Architecture (SOA) is considered the precursor of microservices as they are today. Fundamentally, it would divide the backend of an application into multiple loosely coupled services, communicate with a service bus, and use the same database, as shown in Figure 2.2. Despite being an evolution of the "old" monolith, it still has some issues regarding high complexity and scalability.

SOA was the first step into de-coupling the application's business logic into separate services. However, one of the main issues [32] in this approach is the separation of concerns. Since the application code is ever-changing, the boundaries between services that were once well-defined may become blurred, and there is the risk of falling back to the one-service-does-all logic, which in the end, is regressing to the monolithic approach.

2.1.3 Microservices and Usages

Large applications today, like e-commerce and social networks, require a high functioning standard, like availability, concurrency, and scalability. Microservices have independent development, deployment, release, high concurrency and availability, and low coupling. These attributes allow them to meet the requirements of these large and complex systems. [54]

The evolution towards microservices happens with the fine-granularity of the services, as microservices divide the services into smaller, task-specific components, as displayed in Figure 2.3. This architecture *"proposes (...) loosely coupled services oriented to business responsibilities"*

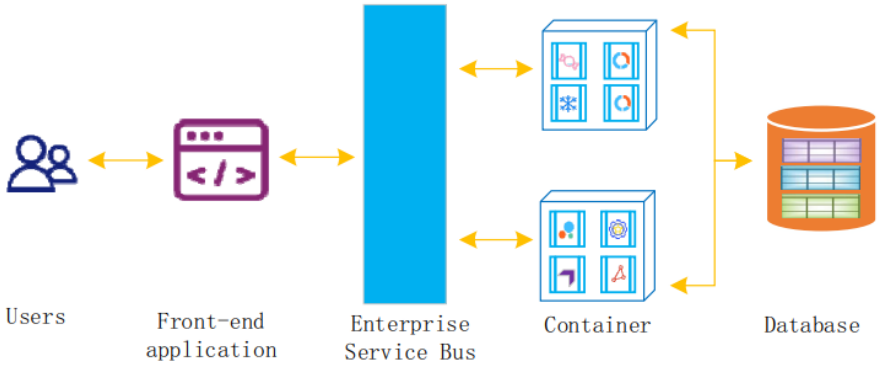

Figure 2.2: Service-Oriented Architecture [54]

[54]. The isolation and autonomy are due to a *"share-nothing philosophy"* of the architecture. It also facilitates the application of agile and DevOps practices, the automation of the infrastructure proving continuous delivery (CD) features, and decentralized data management and governance among services [22]. Nowadays, its extensively used in many companies, such as Amazon, Netflix, and Twitter. Especially Netflix, which has made many contributions in this field and uses more than 500 microservices [54].

One of the leading frameworks used for microservice development is Spring Boot [96] as it provides essential features such as REST client, database integrations, externalized configuration, and caching. Containerization is at the core of the microservice architecture, and the *de facto* standard is Docker [15], as it provides portability, flexibility, efficiency, and speed. The organization of microservices is usually managed by clusters that provide configuration management, service discovery, service registry, and load balancing by using a framework such as Spring Cloud [97], Mesos [7], and Kubernetes [48] [113].

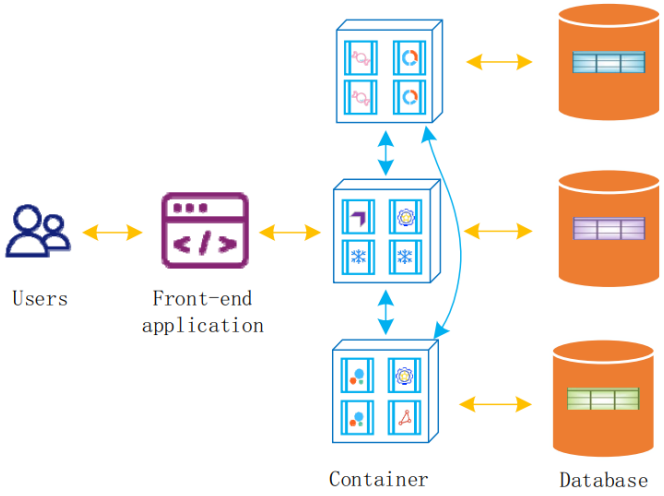

Figure 2.3: Microservices Architecture [54]

One of the main principles of microservices is the division of request processing into several

smaller tasks distributed throughout the systems' services. This requirement means they must communicate to fulfill the request in all its steps. In addition, other requirements exist to maintain the system *healthy* and monitor any abnormalities. A "simple" approach could be embedding these features into each microservices as required. However, this means duplication of functionality, heavier load on the microservices as they are no longer focused solely on the business logic they are meant to perform, and many other drawbacks that make this approach unfeasible. Nowadays, it comes down to two options, which are:

API Gateway

The API gateway approach centralizes the required microservices features (such as authentication, security, monitoring, logging, caching, and rate-limiting) in a single component. Notable examples of API gateways are Kong [47] and AmazonAPI [6].

Service Mesh

The service mesh is a dedicated infrastructure layer for facilitating and controlling service-to-service communications between microservices. Notable examples of service meshes are the ones of Istio [39], Linkerd [52], Consul [33], and Open Service Mesh [66].

Microservices also have drawbacks, some of which are not as pertinent to the problem tackled in this document. Nevertheless, despite not being critical to the system's functioning (and sometimes overlooked), proper performance is required for its maintenance and long run. However, with microservices, this has become a more critical metric, as separating the services requires more communication transactions between them to guarantee the correct execution of the program. This increased amount of messages between services also increases latency and causes speed loss, and specific applications require a tight margin for that [54].

Another most relevant issue is the debugging of these systems. Despite being a concurrent distributed system, microservices systems are more complex and dynamic, and techniques primarily applied for debugging distributed systems do not apply so well. Examples of those techniques are tracing and visualization, which are explored in more detail in the following section (*cf.* Section 2.2).

The reasoning behind the previous affirmation regarding the complexity of microservices is the following [54]:

- Microservices in a large number also run on a larger number of nodes (both physical or virtual machines), and their distribution is constantly changing, increasing communication uncertainties between microservices;
- These systems often require complex configurations, and sometimes faulty executions are related to the incorrect configuration of one or more services;
- As mentioned before, many interactions between microservices are involved, and often they are asynchronous, which leads to complex call chains and can cause runtime failures.

2.2 Debugging

Ghosh and Singh [27] state that *"debugging is a process of finding out the incorrect execution and locating the root cause of the incorrectness"*. It is a process that persists in software development, and every developer faces it as they program, no matter the size and complexity of the project. This section describes some main processes and techniques, such as the exceptional cases of debugging distributed systems and microservices.

2.2.1 Processes and Techniques

As systems become more prominent, so does the number of faults and bugs developers need to fix in different stages of the product. Nowadays, it is estimated that developers spend 20% to 40% of their time [95] and professionals tend to agree on fault locations identified through trace-based and interactive debugging, described in this section.

Generally speaking, debugging requires detecting a bug, gathering the necessary data for analysis, and investigating it to determine a root cause [14]. Besides that, developers learn from experience to use certain *tricks* or use a process to debug systematically. In Figure 2.4 it is presented a generic systematic debugging process that applies these concepts.

Debugging does not (or should not) always begin after a fault is found. Developers can use certain software development processes to prevent and locate bugs, such as unit testing, logging, and static analysis. It can be useful for complex cases (such as non-deterministic failures and memory corruption) and aid during bug location.

Other processes are used by leveraging the abilities of debuggers. They can be mighty when well used, but it is not always a trivial task, and the learning curve can be somewhat complicated. Some of these features are the following [95]:

Reverse debugging

It is essentially *"the ability to run code in reverse"*, allowing the developer to find and track statements that could influence a statement where a fault has been detected. Debuggers like gdb [28] already implement this feature.

Capture-and-replicate or Record-and-Replay

Extremely useful to debug non-deterministic faults, this technique allows the developer to capture and replicate a program's memory access or communication operations in full detail. Usually, the application is run under control until the failure is detected. Then, an execution recording is generated to be replayed afterward under a debugger to locate the statement that causes the fault. Examples of tools that implement this technique are Friday [26] and D3S [56].

Running and dead processes

Debugging a running process usually occurs when the failure is difficult to reproduce. First, the developer finds the process identifier (PID) to attach the debugger. From there, the

standard features of a debugger can be used: interrupt a stuck program, add new breakpoints, and examine values. On the other hand, debugging dead processes allows a *post-mortem* of the execution and investigation of the facts related to the crash. A widely known approach of this type is called *core dump*. This dump consists of an image of the memory associated with the process and can be retrieved to be examined under a specialized, dedicated development environment.

And some of the proeminent techniques for debugging are the following [27, 105]:

Trace-Based Debugging

It works with breakpoints to trace a fault. Breakpoints are used to pause or stop the execution of a program, allowing to examine the state and observe any anomalies at a given point. The developers can then set one or more breakpoints and repeat the process until every bug is found and fixed. There are different implementations of this technique. Trace debugging is the traditional approach, examining the state of the execution line by line, and the developer investigates manually. An omniscient debugging is a more powerful version of this, as the debugger can trace the computation in both backward and forward ways, implying substantial execution traces and low scalability. Others are algorithmic and hybrid debugging, which bring a higher level of abstraction and automation and mix the best techniques (this applies only to the hybrid technique).

Slice-Based Debugging

The program slicing technique allows obtaining a reduced form of the program by deleting statements that do not interfere with the program behavior in a specific case. *Ergo*, the resulting slice must have the same result as the original (whole) program. There are two main variants, static and dynamic slicing. The first is the original one and promotes the idea that the defect of a variable value in a statement will be found in the static slice associated with that "variable-statement pair", reducing the search to the slice alone. The second one, dynamic slicing, was created because static slicing still includes statements that could affect the variable's value. However, some slices could not be used in a specific execution, making them unnecessary in that scenario. The new variant considers only statements directly affecting a specific execution scenario statement.

Delta Debugging

It is an automated test case minimization process. In essence, the goal is to identify a failing test case and gather the set of inputs that are or can be involved in the error. From there, minimal changes to these inputs enable filtering of the ones that do not cause the error. In the end, the result is a minimal subset of inputs that ease the process of fault localization. The number of failing test cases can vary, and the complexity of the input can also vary, from runtime variables to configuration variables. It is part of the family of **state-based debugging** techniques.

Spectrum-Based Debugging

Also named spectrum-based fault localization, is a process that focuses on a particular execution trace and monitors the statements involved. SFL estimates possible fault locations based on program spectra (i.e., program entities) and their coverage status in failed and passed tests. In particular, these approaches count how often a statement is executed for failing and successful executions. A similarity coefficient is then computed, and the statements are then ranked, being statement at the top, the most likely to contain the failure.

There are many more techniques for debugging that are not mentioned in this dissertation. Some of them are used in debugging distributed systems and microservices, which is why they are discussed in the following Subsection 2.2.2 and Subsection 2.2.3.

2.2.2 Debugging Distributed Systems

Distributed systems are everywhere in our lives, whether in our online stores, chat applications, or banking applications. They allow for better scaling and performance, but they pose serious challenges to development. According to Beschastnikh *et al*. [8], they are the following:

Heterogeneity

The challenge of heterogeneity comes from the diversity of nodes that a system can have (mobile phones, laptops, server machines). Developers must manage compatibility during development but also during debugging.

Concurrency

Having simultaneous operations occurring leads to concurrency. This can introduce race conditions and deadlocks, which are difficult to debug. In addition, one must consider the packet delay and loss in network communications.

Distributed state

Having a distributed system state across multiple nodes, although removing a central point of failure and improving scalability, also requires complex node coordination to assure synchronization of the state among the nodes. Despite distributed algorithms preventing eventual inconsistencies, it is a rather difficult or maybe impossible task to rebuild the global state of the system scattered across multiple nodes. This also is an issue for bug diagnosis and validation.

Partial failures

Having a distributed architecture allows the system to have a particular grade of fault tolerance compared to traditional centralized systems. Nevertheless, achieving this tolerance is not a trivial task and requires previous thought from the developers, and the design is often complex and hard to test.

With that in mind, several techniques have been discovered and improved over the years to make this process easier for developers.

Some of them are fundamental elements of software development, such as testing. However, there are also areas dedicated to going in great depth to validate the systems, such as the case of model checking and theorem proving [8, 75].

Others are more balanced with applicability and utility for developers. To summarize:

Monitoring

One of the most used debugging techniques is monitoring. Not only for the debugging but also for performance checking, monitoring allows developers to be alerted when something goes wrong when some resource is over or underused and quickly informs the team to examine the system and understand the misbehavior. Usually is used to detect abnormalities in the system architecture, configuration, and deployment rather than the code of a specific process.

Log analysis

Log analysis works better with systems that cannot be modified and are therefore a lighter-weight approach. The reason for this is its black-box approach to collecting console logs, debugging logs, and various other log sources and examining them to find anomalous behavior. Detailed logs from large systems contain a considerable amount of relevant information, but the volume is often too heavy for developers to use, as it is overwhelming [8].

Event logging

A branch of the logging technique, event logging is the go-to tool as it is not yet possible to *single-step* concurrently through the significant number of processes in a modern distributed system. This technique consists of processes logging operational events that target system admins and reliability engineers. It can always be enabled in a production environment and provides observability, aiding developers in evaluating the application's status and interactions between processes and determining possible changes in the system's configuration. By making explicit the contents of the error messages and metrics, finding the root problem is made more accessible in a complex failure [95].

Tracing

Tracing is used to track data flow in a system across applications and servers, such as a database, web server, DNS, load balancer, or VPN. Traditionally, tracing is more efficient than Record and Replay because it focuses on a specific subset of data. However, it requires instrumenting applications and protocols to forward tracing metadata without consuming it [8].

Visualization

Due to the complexity of these systems, visualization techniques have come to the surface to face this issue and make the systems more transparent to developers. A common drawback is that they do not help a developer understand the system's underlying communication pattern nor the distributed order of messages [8]. Theia [25] is an example of a Visualization tool, as it displays the visual signature of Hadoop executions like resource usage [95].

Record and replay

As also mentioned in the previous subsection, this technique helps debug non-deterministic behaviors. The main issue is recording complex executions, which can be expensive and even change the system's behavior beneath.

2.2.3 Debugging Microservices

Microservices are distributed systems, but they are much more complex and dynamic than traditional ones. According to Zhou *et al.* [113], it is hard to establish a relation between microservices and system nodes due to the natural ability of microservices scaling (instances can be created and destroyed at any time). Therefore, some or most of the tools used to debug distributed systems might not be suitable for microservices in general.

For instance, existing debugging techniques based on breakpoints are ineffective for microservices due to the high concurrency, which can cause the same breakpoint to be reached in different executions and have different program states. In addition, the asynchronous processes, characteristic of microservices, require tracing multiple breakpoints across multiple processes, being a very challenging task.

Another example is traditional fault localization techniques, such as spectrum-based fault localization mentioned before, that are inadequate for highly concurrent and dynamic systems such as microservices. Recently there have been efforts to extend these techniques to concurrent and distributed systems. However, they require execution data at the thread level to apply such fault localization techniques. This is a highly complex task due to the dynamism of microservices instances, complicating the evaluation of the produced logs. That being said, it is natural that **runtime verification** and **debugging** are the main challenges for microservice developers, as they heavily depend on **monitoring and tracing** of the systems.

Developers also make use of **logging** techniques. As mentioned before, the simplest level is to analyze execution logs that the system "spits" to locate faults. Usually, the developers need to look over a considerable amount of logs manually, so naturally, the efficiency of this process is related to the developer's experience, both with the system and faults of the same family. Another level, more structured, is the use of **visual log analysis** that allows developers to find more easily the set of logs that they might be interested in with appropriate search operators. Furthermore, the results can be sorted and presented in different statistical charts. A common stack used for this purpose is called **ELK**, which is composed of ElasticSearch [17] (responsible for log indexing and retrieval), Logstash [20] (responsible for log collection), and Kibana [19] (responsible for visualization).

The most advanced level of analysis is **visual tracing**. Here developers inspect collected system execution traces with the support of visualization tools. Traces are accompanied by an ID and allow identifying the invocation chains among microservices.

The time developers spend debugging is directly correlated to the number of microservices involved in the fault. Having more than three microservices is enough to make this process almost two days. Depending on the service provided, it can be critical and cause a significant loss to the company.

We explored academia and industry, and our understanding is that the research on fault localization in microservices is in an early stage and is usually based on small systems, which is not a comparable benchmark for many industrial-grade microservices.

2.3 Summary

In this Background Chapter, we explored foundational concepts for our work, which relies on microservices and debugging.

Regarding microservices, in Table 2.1 it is shown a summary comparing different architectures.

Table 2.1: Comparison of different architectures [54]

Categories	***Monolithic Architecture***	***Service-Oriented Architecture***	***Microservices Architecture***
Componentization	Module	Service	Fine-grained
Component size	Big	Coarse-grain	Fine-grained
Elasticity	A single point of failure	No single point of failure	No single point of failure
Deployment	Holistic creation and deployment	Each component deployed independently	Each component deployed independently
Storage mechanism	Shared database	Shared database	Private database
Technology	The same programming language and framework	Isomorphism	Heterogeneous
Scalability	Unable to scale on demand	Scale on demand	Scale on demand

There has been an evolution of architecture as they needed to face new challenges. In the era of microservices, there is motivation to face the challenges that come with the architecture, and the gain is undeniable. Therefore, the more (and better) tools developers have to tackle these issues, the better the product will be. It is in the interest of all to keep exploring techniques and approaches to improve all phases of software development in this distributed architecture.

Debugging is the process of finding and fixing a problem in a program. With the evolution of software architectures and processes, the techniques used to debug programs followed this trend.

With the emergence of distributed systems, new challenges have arisen for developers due to the complexity of these systems. New tools have appeared to assist developers in debugging these systems, but they still consume many resources, such as time.

Microservices are the latest trend in distributed systems and come with their challenges. Some techniques such as tracing, monitoring, logging, and visualization are the microservice developers' "better" friends. However, there is much room for improvement, and research on debugging techniques for microservices is still in its early stages.

Some techniques, when combined, can be powerful allies and contribute with their strengths to tackle the complexity of debugging microservices. The case of logging and spectrum-based fault localization is the cornerstone of our solution.

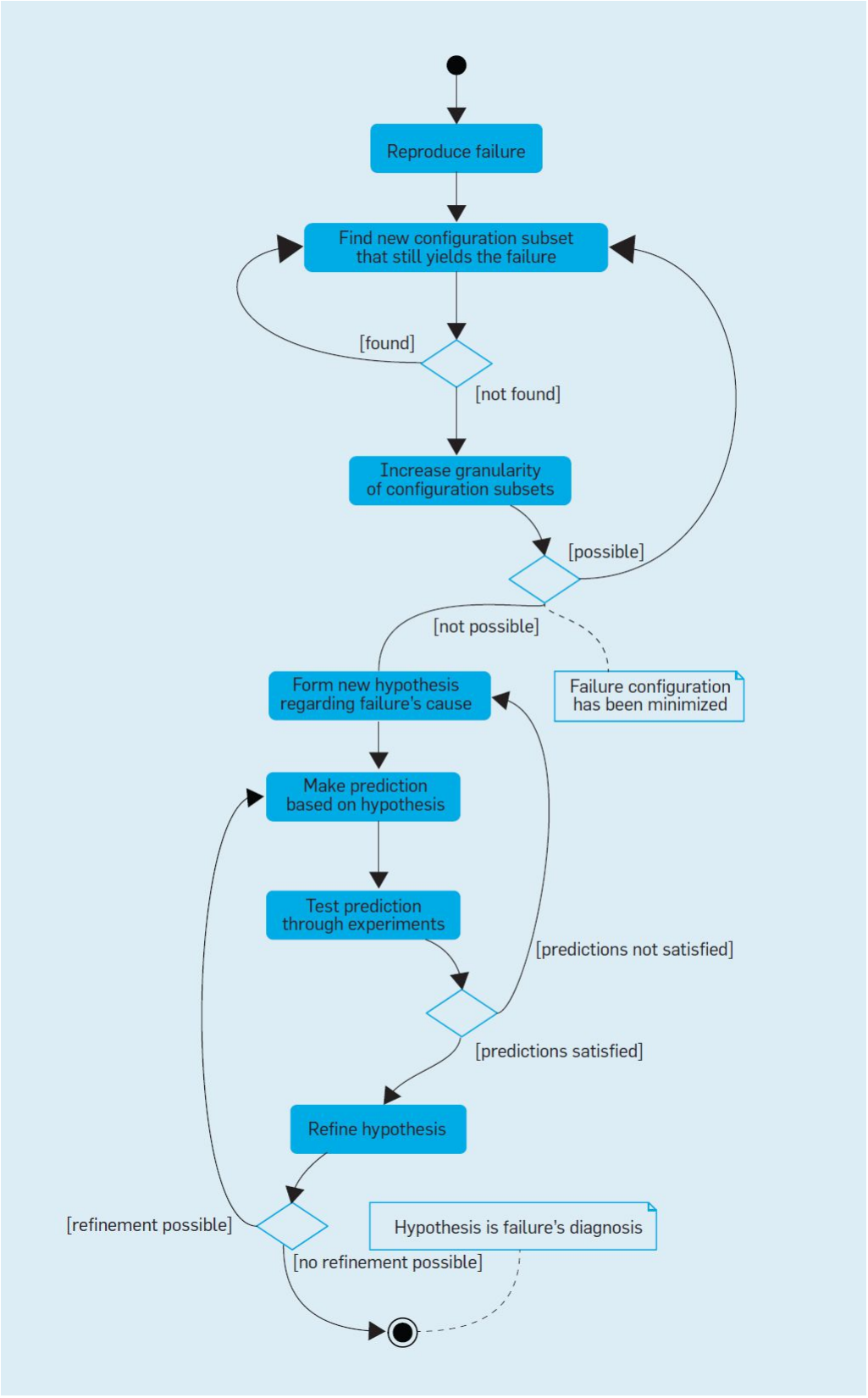

Figure 2.4: A process for systematic debugging [95]

Chapter 3

Related Work

This chapter explores concrete solutions from industry and academia dedicated to debugging microservices. We also explore debugging solutions that, despite not being aimed towards microservices directly, can be adaptable or inspirational towards our methods for developing our solution. Finally, we explore approaches that, while not debugging microservices, are useful for developers by analyzing the system and its behavior and generally indicate where a problem might be found.

Our literature review is then composed of three sections highlighting microservice debugging approaches, traditional system debugging approaches, and other helpful tools to assist the debugging process. We here consider traditional systems with a monolithic nature, whether single-threaded or multi-threaded (both concurrent and distributed).

3.1 Debugging Microservices

3.1.1 Spectrum-based Fault Localization

Spectrum-based fault localization, as mentioned in the previous chapter, estimates possible fault locations based on program spectra and their coverage status in failed and passed tests. In this section, we present an SFL solution applied to microservices.

Ye *et al.* propose T-Rank [110], a lightweight spectrum-based performance diagnosis tool, which provides a ranking for a list of microservices to localize the root cause of a failure. It leverages the SFL algorithm and a single standard metric, the latency in completing the request. This avoids the challenge of building complex dependency graphs. They apply the SFL algorithm, which does not require previous training and the causal graph of the system.

T-Rank collects tracing data in a sliding time window and condenses them into tracing chains representing the request generated by client service, from the beginning to its end. Then it labels the data by the metric representing the elapsed time of the whole request. Based on the spectrum algorithm, it creates a ranking score for a list of containers (microservices). It compares different SFL algorithms with some focus on the Ochiai algorithm [4].

It may be the case where more than one root cause exists at the same time, and the tool is only able to localize only one. The container with the second-highest score in suspiciousness may be

innocent because it is only affected by the root cause container in the same tracing chain. It needs to reduce the number of false alarms raised. It is a valuable tool for use as a support tool due to its lightweight.

SFL is an exciting approach that we further explore in the context of "traditional" systems. In T-Rank, it is explored as a performance diagnosis tool, while we intend to focus on faults in the code.

3.1.2 State-based fault localization

In *state-based fault localization* the technique leverages the state of a program at a given point of execution, usually when a failure occurs, and explores it to find the origin. A definition for a program state is that it "consists of variables and their values at a particular point during program execution, which can be a good indicator for locating program bugs." [105] One particular example of this technique, which can be applied in microservices, is the delta debugging technique.

Zhou *et al.* [115] repurpose the delta debugging algorithm on microservice systems. In delta debugging, the goal is to "minimize failure-inducing deltas of circumstances (e.g., deployment, environmental configurations) for effective debugging."

A particularity of their approach is to define a set of dimensions that bound the execution of microservice systems, to construct the failing circumstance space. The dimensions in question are *Node*, *Instance*, *Configuration*, *Sequence*, and *Input*. Therefore, the circumstances conjugate these dimensions in a specific setting.

The approach identifies failure-inducing deltas that effectively help root-cause diagnosis, but it is a rather consuming process and not trivial to implement. In [112], the same authors make an effort to optimize this process by parallelizing the delta testing tasks, which improves the efficiency of delta debugging significantly.

They are still limited by the granularity of the supported atomic deltas, and the dimensions of the circumstances are limited. In essence, it fails to consider reasonable deltas (for example, the default value not being unlimited memory) and does not consider specific relevant dimensions, such as the invocation chain. Despite the limitations, this tool is one of the few advanced fault localization techniques applied to microservices and shows great promise.

3.1.3 Record And Replay

In *record and replay* techniques, the developer records essential parts of the system execution (calls, requests, responses) and stores the data to perform a replay in a controlled environment. In this way, the developer can perform a deep inspection of the conditions that lead to a fault and manipulate the scenario toward fixing the original root cause of the fault.

In his master's dissertation, Silva [92] proposes a solution based on the Record and Replay technique for microservices. In particular, the proposed framework focus on instrumenting and recording network communications and random number generation. The premise is that recording those components is enough to replicate the previous execution deterministically. The solution

targets individual microservices, thus not having a holistic system view. It records network communications protocol-agnostic and random number generation in the JVM (Java Virtual Machine) programs.

The resulting prototype reveals a low overhead on the system execution and low intrusion on the source code. However, storing the high amount of data generated from the recording phase is an issue. This limits the tool's usage on systems with many events (mainly network events). Furthermore, the tool excludes microservices that do not run on the JVM, which, however broadly used, does not contribute to the programming-language agnostic aspect of debugging tools.

Another approach to Replay Debugging is the one that Mathur [63] proposes in his master's dissertation. In more detail, he proposes a language-agnostic replay debugging framework (for microservices) using a distributed tracing system to record network communication. He replays them on cloned service containers that run in a debug environment. One highlighted feature is an anomaly detector that uses span-level and container-level monitoring in fault symptoms detection and trace-level fault localization to find the root cause. This detection occurs from a mathematical formulation of span-level (latency) and container-level (memory and CPU usage) applied to time series data.

The author claims that the framework's overhead is *"10X less than the case in which language-specific recording tools (...) are used"* as part of the evaluations performed on the tool. It is also stated that the tool cannot reproduce all executions of the microservice application but can be a valuable tool for finding the root cause of common faults. Furthermore, the tool deals with several sources of non-deterministic faults, such as network, resources, and asynchronous requests. It does not handle non-determinism from third parties, which can happen when services call external APIs. Naturally, the responsibility of debugging an external API does not fall on the system's developer. However, it is helpful to recognize such interactions when retracing the steps that lead to abnormal behavior.

3.1.4 Live Debugging

In *live debugging* the developer performs traditional debugging, e.g., with breakpoints, on a running program, by interacting with it.

A recent tool for live debugging microservices is Squash [94], which allows debugging of running microservices, containers in a pod, service, and such in real-time. The idea is to bring the powerful functionalities of debuggers in the traditional monoliths to microservices. It aims to bridge between apps running in a Kubernetes environment and the developer's IDE. Since it is recent, it is limited to Kubernetes implementations, which cover a large part of modern microservice deployments. It also supports known debuggers, including gdb, node, and python. Besides Kubernetes platform also supports OpenShift and Istio, which provide the service mesh. Currently, the supported IDEs are VS Code, IntelliJ, and Eclipse.

Squash has a promising potential, but currently, it is elementary in functionality and still lays on the developers' shoulders most of the configuration and debugging work. A demo by the team [49] shows that despite integrating with commonly used tools such as Zipkin and Jaeger, it is

still a long way from becoming an essential tool for debugging microservices. In addition, the project seems inactive according to their official git repository. As of 2022, the latest update in the main branch dates back to 2020, and the latest release dates back to 2019, which is somewhat concerning if the project is to be kept alive.

3.1.5 Anomaly Detection

Anomaly detection encompasses detecting the symptoms that a fault cause in the system. This way, we can find the correspondence between the anomaly and the concrete failure it is causing and fix it. Anomalies can be detected at two levels: application level (when considering the symptoms of the application as a whole) or at the service level (by focusing on the symptoms detected at that level). There are also different techniques to base the detection, namely log-based techniques, distributed tracing-based techniques, and monitoring-based techniques [93].

In the case of log-based techniques, they usually leverage unsupervised machine learning algorithms. These are used to process the logs generated by the services and learn the system's behavior when there is no failure in execution. Such are the cases of [41, 42, 70], which apply the said technique. The cost of this technique is associated with the training the approach has, how much data is required, how diverse the training data is, and how accurate it is. However, they benefit from no intrusion since the log processing requires no code instrumentation.

As for techniques based on distributed tracing, they can leverage unsupervised machine learning algorithms [43, 55, 72], supervised machine learning algorithms [24, 71, 9, 114], or by comparing traces [10, 64, 101]. The first two require data training and therefore have the additional cost mentioned before. The second is trained with known anomalies to improve accuracy and have the associated additional cost. The third, and last, follows a much simpler approach by collecting the different possible traces in the system and comparing the new traces with the collected ones. It is simple and less costly than the previous ones but also less effective. Distributed tracing implies instrumentation of the code and therefore goes against the principle of no-intrusion. Configuring the tool to train data is also not a negligible effort from the developer's side.

Finally, the techniques based on monitoring use agents installed on the systems to monitor Key Performance Indicators (KPIs) and detect an anomaly in the readings. To learn the baseline of the systems, these techniques often leverage unsupervised [30, 60, 84, 88, 100, 107, 108] and supervised [16, 61] machine learning algorithms. Nevertheless, there are also ones that rely on SLO (Service-level objective) checks [12, 29, 90] and heartbeat [111], a more fundamental mechanism of checking the availability of a given service. These last two have a much broader view of the application and, therefore, are less detailed. In general, monitoring-based approaches require specific or additional configuration to set up the agents to monitor the KPIs. The cases of solutions based on SLOs or heartbeats avoid the training costs of the machine learning techniques. However, SLOs require previous monitored KPIs to generate a baseline system model, even if it is on a much smaller scale than training data sets.

These approaches based on anomaly detection are helpful and sometimes easy to deploy when the developer has/wants already tools in place to monitor the system performance, considering

external conditions not directly related to the application code. It is not the focus of our work, as we intend to focus on eliminating the faults that originated in the code. However, the conditions the tools presented work with are (almost) the same as ours, given the microservices and cloud computing contexts.

3.1.6 Root Cause Localization

The *root-cause localization* technique identifies why an anomaly or failure occurs, aiming to locate the root of the failure at the application or service level [93]. Such as in anomaly detection techniques, we separate them into three categories considering log, tracing, or monitoring nature.

Regarding log-based techniques, it leverages the application logs to generate a causality graph of the execution and extract possible candidates for the root cause from the anomaly. The case of [5] analyzes frontend errors on multi-service applications to generate a causality graph by perceiving the services as multivariate time series. They extract the "cause-effect" relation between them to create the topology, extract the causality graph, and select a subset of application services that might be the root cause. Finally, it uses another algorithm to rank them according to the root cause probability. Log-based techniques are more straightforward and might have the necessary information for the developer to decide and detect the problem.

As for the (distributed) tracing-based techniques, as explored in the previous section of anomaly detection, it requires the instrumentation of the code to leverage the tracing of the applications' requests. The approach used to analyze the traces can be separated into visualization-based [31, 113], direct [50, 55, 67] or topology-based [46, 53] analysis. In visualization analysis, the developer is presented with a visual comparison of traces, so they manually determine the root of the anomalous traces. On the other hand, in direct analysis, the root cause finding is automated with a given approach. For example, in [50], it classifies traces as anomalous through anomaly detection and considers microservices with a high ratio number of abnormal traces versus normal ones, providing a ranked subset of these microservices. Finally, there is the case of topology-based approaches where the application's topology is determined from the traces, and they leverage that to apply a specific analysis and determine the possible root causes for the anomalies observed.

Tracing techniques have a necessary degree of intrusion of the system to allow the instrumentation of the code and obtain the traces. The automated approaches have the advantage of requiring less effort from the developer to detect root causes than manual approaches, such as visualization-based. The direct analysis does not rely on the topology, which is better in cases where it is hard to extract the topology of the network or is highly connected.

Monitoring-based techniques, such as it happens with anomaly detection, rely on active monitoring of the system KPIs data collected by agents installed inside the system. The analysis provide can be on a direct basis [90, 99, 77, 76], topology-based [88, 107, 108, 116, 98], or causality graph-based [12, 29, 85, 100, 57, 58, 65, 51, 60]. The direct analysis is based on the idea that an anomaly detected on the frontend is registered by (anomalous) KPIs identifying the corresponding service. Topology-based analysis works the same way as before, driving the analysis according to

a reconstructed topology model. The causality-based analysis is quite similar to performing the analysis based on the extracted causality graph of the services.

The root-cause analysis is, in our opinion and based on the observed applications, an appropriate tool to explore when performance evaluation mechanisms are already deployed. Their focus on application or service level is limited in assisting the developer in finding the root cause in the application code, i.e., at a lower level.

3.2 Debugging Traditional Systems

3.2.1 Configuration Fault Localization

Configuration fault localization is the technique that aims to identify faults in the configuration of a particular component, which is the source of failure when given failure-inducing conditions.

In [45] the authors present a novel approach to performing configuration fault localization based on the difference of configuration parameters in components that share a resource. More specifically, they define a *Reference Configuration State* based on the "set of non-faulty probing components for each faulty component with respect to shared resources". The wrong configuration parameter is localized by obtaining the difference in the configuration of that reference state with the faulty component.

The solution has the advantage of not requiring the expertise of the domain at hand, which can be helpful sometimes and even more with complex configurations. In addition, it seems to be scalable in large systems. However, it still lacks accuracy due to the data available in resources and applications. The experiment showed that the tool could detect about 20% of the configuration faults. It is an interesting approach but completely overlooks other sources of faults, making it suitable as an additional tool to inspect microservices configuration, but it is limited to that function.

3.2.2 Slice-based Fault Localization

Program slicing is a technique to abstract a program into a reduced form by deleting irrelevant parts such that the resulting slice will still behave the same as the original program concerning certain specifications. There are two main variations of slicing, one of which is static slicing, which is the traditional approach based on the static analysis of the program. On the other hand, dynamic slicing computes the slice during the program execution and considers only statements that directly contribute to the detected faulty behavior. The main difference is that dynamic slicing reduces the set of statements of the slice even further.

Wotawa's JSDiagnosis [106] combines dynamic slicing with model-based diagnosis to achieve a more effective fault localization. The basic idea of the approach is to combine slices for faulty variables to minimize the resulting set of statements that may lead to the detected misbehavior. It states as prerequisites of the approach are the program source code and at least one test case. It dismisses the need for large test suits, which is common in other approaches of this type. It

assumes that a missing statement does not originate the fault; in that case, the tool still works, but with no guarantee of usefulness in the results. The tool collects dynamic slices for faulty variables using a given test suite against a program. From there, it constructs hitting sets, which contain at least one statement from each (dynamic) slice. The probability of a statement being faulty is based on the number of hitting sets covering that statement.

Given the nature of the tool and its implementation (focused on Java programs alone), we analyze this tool in the approach sense since it could make sense to adopt a slice-based strategy in our debugging solution. We, therefore, refer the reader to [105] for an in-depth review of the technique and the extensive literature based on it.

It is also known that techniques based on slicing-hitting-set-computation sometimes produce an undesirable ranking, including certain statements, such as constructors, which are executed in many test cases, placing them at the top [105]. Another known limitation of dynamic slicing-based techniques is their failure to capture execution omission errors, which translates into specific critical statements in a program not to be executed and ultimately lead to failures.

3.2.3 Spectrum-based Fault Localization

We previously defined *Spectrum-based fault localization* and provided an example applied to microservices. However, most of the work based on this approach has been applied to "traditional" systems. We present essential tools and refer the reader to [105] for further review of the literature regarding this technique.

Zoltar [40] is a toolset that implements spectrum-based fault localization techniques, with a highlight to the BARINEL algorithm, of the same authorship. The toolset provides the necessary infrastructure to instrument automatically the source code to produce data in run-time and analyze it afterward to return a list of candidates. It is aimed at complete automation and, therefore, can instrument the analyzed program with fault screeners, replacing test oracles at run-time. Fault screeners are generic program invariants trained to be application-specific. Some of the program spectra included by the tool are basic block hits, function hits, and def-use pairs. There are two approaches that the tool uses for fault localization. The first is based on statistics, based on the assumption that a high similarity to an error vector indicates a high probability that the corresponding parts of the software are the cause of the errors and that the computed similarity coefficients rank the parts of the program according to the likelihood of containing faults. Once more, the default coefficient (algorithm) used in the toolset is the Ochiai [4], but it mentions other coefficients the tool provides optionally. The second approach is based on a reasoning method called BARINEL, a spectrum-based, logic reasoning approach to fault localization. This approach is also able to detect multiple candidates with multiple faults. It then returns a ranked list of candidates based on a posterior probability.

It is a tool based on abstractions of program traces and is extensible by design. It does not translate directly into a practical tool to be used off the bat. However, it is worth considering design aspects while developing a technique for fault localization based on the program spectrum.

Q-SFL [82], is an SFL approach that leverages SFL with Qualitative Reasoning (QR) by introducing quantitative landmarks that partition the domains of system components into a set of qualitative descriptions. QR is a branch of Artificial Intelligence, and it describes continuous values with discrete, behavioral qualities, allowing the reasoning of a system's behavior without specific quantitative information. By considering the descriptions generated by QR SFL components, it is possible to record their coverage and then apply the traditional SFL approach by diagnosis and ranking. Thus it allows the suggestion of the most probable location of the bug and pinpoints behavioral properties that induce failures, adding value to the report produced by the algorithm.

Nevertheless, the authors concluded regarding the innovative approach that there is "no single (automated) landmarking strategy that was consistently better than the original spectra, meaning that using more intricate, white-box strategies will likely be necessary for practical applications of the approach." This goes in favor of our belief that strategies must be combined.

While the particular tools described in this section are not suitable for microservices, it is a fascinating technique to find faults in the application code and could benefit from the information that contextualizes the nature of microservices. In sum, the technique is not built for dealing with the idiosyncrasies of microservices, but it seems adaptable to such scenarios.

3.2.4 Breakpoints

One of the most traditional debugging approaches is using the use of *breakpoints*. A given statement is marked with a breakpoint, and the developer executes the program in a "special" configuration, usually called "debug mode". When the program's execution reaches the marked statement, it halts until the developer resumes the execution. In the halted state, the developer can access the program state at that moment of the execution and inspect variable values in the call stack, among other functionalities. The most famous tool for breakpoint debugging is, most likely, GDB [28], which has been used for a long time and remains actively used.

Naturally, it is impossible to employ this strategy in complex scenarios such as microservices. It is naturally overwhelming for the developer to halt the execution of an instance (not the microservice itself) and try to debug from there. Some efforts like the one mentioned in the live debugging section still fall short compared to other techniques better prepared for these scenarios.

3.3 Debugging Assistance

3.3.1 Monitoring

The *monitoring* technique consists of constantly analyzing the system performance to detect abnormal behavior. By continuously tracking performance metrics (response time, CPU, and memory usage), the developer can discover when a fault occurred, which will help find the cause.

Kmon [102] is a monitoring tool that aims to provide multiple run-time information on microservices such as latency, topology, and performance metrics with low overhead. It makes the

argument that capturing "all types of indicators with a unified data structure" can reduce the cost of aggregating indicators. This is necessary when different monitoring tools provide information in a different format. It also argues for monitoring more *fine-grained* indicators to discover additional deeper problems. An example of a *livelock* is made to exemplify that CPU and other resources' usage does not guarantee proper execution of a program. Avoiding code instrumentation makes a case for the tool collecting complex data, such as TCP messages' latency.

The results show that CPU usage of the tool is negligible, but its memory usage is not. This is defined as future work since it severely impedes its application. It also lacks a generic view of the TCP connection, thus unsuitable for services that communicate through message queues.

There are also "black-box" approaches to monitoring, which collect information about performance and calls (e.g., elapsed time of the request, content of the request, and response) without a deep knowledge of the system or a low-level insight of the functioning. This is the case of the tool proposed by Pina *et al.* [83], where they propose to log gateway activity, thus not interfering with the application code. It does not interfere with the system at the microservice level, making it a good solution for monitoring systems already in production. The information extracted allows computing the system's topology, average response time, and the load in each microservice, which also allows inferring each part's maximum capacity and Quality-of-Service (QoS).

However, this approach has some drawbacks, especially regarding causality. Because the tool does not have a fine-grained view of the system that usually tracing techniques provide, it is naturally limited in indicating faults localization, leaving the developer to explore the system deeper to achieve that. Furthermore, despite being a reasonably good solution for monitoring microservice systems, it is built around systems that resort to API gateways to serve the system communications. Therefore, it would require adaptation for service mesh,

3.3.2 Log Analysis

In the *log analysis technique*, the developer leverages the logging of the system to inspect the behavior and gather in-depth details about a specific execution.

Falcon [73] is a tool that aims to make the log analysis of distributed systems more practical and effective. It is built as an extensible pipeline, providing modularity, and can combine different logging sources without much effort. It also generates a space-time diagram of the distributed executions. This is achieved by employing a happens-before symbolic formulation and obtaining a coherent chain of events with a constraint solver.

As it is based on the logs produced by the system, it does not require instrumentation of the code, avoiding intrusion in the system code. However, the constraint solver used to perform the causal ordering cannot scale to executions with more than a couple of thousand events. Naturally, this is not suitable for production environments, let alone for microservice systems. The manipulation of execution traces is also limited, making it more challenging to use in day-to-day debugging scenarios.

Another example of log analysis tools is Horus [74], the successor of Falcon. It also "enables causally-consistent refinement of distributed system logs in a non-intrusive and scalable fashion".

The novel approach in this proposal is leveraging kernel-level probing to capture events and establish causality between logs from multiple sources at the application level. Furthermore, the events are present using a directed acyclic graph (DAG) and stored in a graph database, allowing querying and thus facilitating the debugging process.

The results show that the tool surpasses widely-adopted similar systems in pinpointing the root cause of faults and performs considerably better ("up to 3 orders of magnitude") in building causally-consistent logs of the distributed executions. It also outperforms traditional traversals in graph databases in the query computation time by reducing it up to 30 times. Despite the outstanding performance, a drawback of this system is the necessity to do a proper configuration to ensure intra-process and inter-process causality. This means that additional time must be spent configuring how events are collected in the tool and should be updated when the microservice system changes (a new microservice produces new events, so that must be considered). Furthermore, it is unclear what the support for RPC is, and it is a growing trend with microservices, especially with service meshes. Even though RPC is considered a form of inter-process communication, the physical address space is not the same, and it is not clear how the causality is established in executions in a microservice triggered by a remote call in another.

3.3.3 Tracing

The *tracing* technique correlates events by identifying the messages interchanged between processes or services, thus allowing to establish a trace of the events involved in a specific execution.

One of the first examples of distributed tracing is Dapper [91], which was proposed by Google back in 2010. Despite its age, it contributed significantly to how tracing is performed, and this work has inspired several tools that have appeared since then. The tool has contributed with new features, including sampling and achieving a degree of application-level transparency that makes it more appropriate in production. The goals of Dapper are low overhead, application-level transparency, and scalability. It records timestamped messages and events that trace the whole execution of the system. Nevertheless, it restricts its instrumentation to a small set of common libraries.

As mentioned before, one of the novel contributions of Dapper is the use of sampling towards maintaining a low overhead, especially useful in scenarios that are very sensitive to latency variation. They use a uniform sampling approach to select which data to record. They also found that, on average, 1 out of 1024 requests has enough relevant data in services with high data throughput. This would not be the case with services with lower traffic. In addition, the scenarios described in the paper are based on the uniformity of the system. This is not the case of the typical microservice architecture, nor is it the goal it tries to achieve, but the exact opposite. Microservices are meant to be inherently heterogeneous, and the work required for instrumentation would be pretty considerable, as the effort required to maintain the interoperability of the tool.

Zipkin [80] and OpenTracing [79] (which is now merged into OpenTelemetry [78]) are widely known tools in the industry that provide distributed tracing. They require the instrumentation of the application code in order to report trace data. They also support the primary protocols for

communication at the application level (level 7), such as HTTP, message queues, or RPC. The tracing headers in each request are minimized to maintain a low overhead. The identifiers in each request are sent in-band with the rest of the application communication, but the details of each request, which can be analyzed later, are sent out-of-band asynchronously. This permits tracking causal and temporal relationships of service invocations in each request.

They are potent tools for distributed tracing and are among the most used in the industry [59]. However, since it requires the instrumentation of the application code, it is not fully transparent to the user. It only gives the developer information on where a fault might be, but it does not assist in finding the root cause of the fault. This is because the state leading to the fault is not recorded with the trace (which makes sense since it is not a record and replay tool). Hence the conclusion about these tools is that they are not to be used singlehandedly but to support other tools or techniques the developer employs towards debugging and maintaining its system.

Another example of a tracing-based tool that leverages the sampling technique is Sieve [35], which focuses on systems with massive amounts of trace data. The proposal is a novel sampling technique that aims towards uncommon traces, often overlooked by typical sampling approaches, and provides helpful information. The strategy for the online sampling approach is to use the robust random cut forest, a variant of isolation forest, to detect uncommon traces. It highlights traces that distinguish themselves from others (either temporarily or structurally) and gives them higher chances of being included in the sampling. This technique helps detect uncommon traces and reduce the cost of storage that is part of the implementation of sampling.

The approach results show a low overhead, considering it is an online tool. The tool's applicability must be considered with additional tools, mainly tracing, to provide the necessary data to perform well.

A more low-level and infrastructure-specific approach is Rbinder [89], whose novelty lies in joining proxies usage and operating system syscalls monitoring to handle tracing and establishing causality between multiple requests, respectively. This is an effort to address the high heterogeneity that microservice systems possess. They aim to separate instrumentation and application code, minimizing the performance overhead. In a first step, Rbinder deploys one proxy for each microservice as a middleman (similar to the approach of service meshes) and inserts the tracing headers in each request that passes the proxy. Afterward, the service must run as a child process of the tool. It happens this way to ensure the application propagates the header given by the proxy in each request. This way, the tool achieves a transparent application of tracing.

It is a future-oriented approach to tracing, as it leverages proxies to perform tracing, but the necessity of intrusion remains a drawback. In addition, it still does not support other protocols besides HTTP, which is a limitation. The use of service mesh promotes the growing usage of RPC since it is more performant than HTTP.

To address the lack of transparency in tracing, the authors of Inkle [81] argue for a transparent tracing system based on microservices that use RPC (gRPC, in particular) to communicate. Remote Procedure Call allows performant communication between services when they require tasks performed by other services. The microservice-based systems targeted are run with Kubernetes.

The solution implements passive tracing, which requires the system to be non-intrusive. This is achieved by performing packet interception on the packets sent by the application, recording the gRPC trace data, and writing it to a log. It leverages ElasticSearch for storage and Kibana for visualization and analysis.

This tool can perform transparent tracing, but it cannot do it recursively due to its log-based approach. Ultimately request causality cannot be discovered through the tool, as the trace data related to the request does not establish a relationship between requests. Furthermore, the success rate of the trace seems to be inaccurate to this constraint.

3.3.4 Visualization

Visualization is a technique that allows the developer to view the system functioning graphically, as opposed to the traditional textual view of logs and traces. In the same way, one can read the traces of a call to understand the flow of a call; visualization tools allow one to visualize that same call flow, simplifying that process for developers and ultimately saving time.

The work of Beschastnikh *et al*. gave origin to ShiViz [8], a visualization tool dedicated to distributed systems. This tool displays distributed executions as "interactive time-space diagrams that explicitly capture distributed ordering of messages and events in the system". ShiViz establishes the *happens-before* relation between events to represent the execution more accurately. It does not guarantee total order, as some events may occur after or before others without affecting its behavior. Nevertheless, when causality is required, it can establish the relationship between events. The tool also allows rich manipulation and querying of the data to understand the flow in question. ShiViz assists developers in visualizing the events (partial) order, searching existent communication patterns, and registering causality among events. This is extremely helpful considering the nuances of distributed executions and the abnormal behavior that can emerge from there.

Notwithstanding, the tool has some limitations. First, it is not helpful to understand high-level behavior due to the low-level nature of its information ordering. It is based on logical ordering and not real-time, making it unsuitable for inspecting specific performance metrics. The tool is built as a client-side-only browser application, making it portable and appropriate for sensitive log data, but it also severely limits its scalability.

Another underlying problem in visualizing microservice traces is mapping microservices units to nodes. Since microservice instances run on containers and are dynamically created and destroyed (along with their containers), this is an issue for visualization. Furthermore, it is unknown *a priori* which instance is assigned to handle a call made towards a microservice.

3.4 Summary

In this chapter, we analyzed several tools and techniques used and developed by academia and industry to help developers debug their systems.

First, we explored debugging approaches for microservice-based systems and noticed that there is not yet a large variety of solutions implementing fault localization techniques. Such exceptions are solutions that apply delta debugging (state-based fault localization) and spectrum-based fault localization. Both of them focus on external aspects, although necessary, besides the application code and do not provide a fine-grained insight into the fault itself. We also explored record-and-replay and live debugging tools, which give developers some of the "power of debugging monoliths" back with limitations. At last, in this section, we also explored anomaly detection and root-cause analysis tools, which are essential to discuss, although not focusing on the application code due to their practicality in production scenarios.

Afterward, we explored debugging approaches for "traditional" (non-microservice-based) systems, as it contains most of the literature on debugging techniques, especially for distributed systems, of which microservices are a part. We focused on advanced fault localization techniques such as configuration, slice-based, and spectrum-based fault localization.

Finally, we discussed debugging assistance tools and approaches. Despite not being tools that actively find faults, they are widely used in industry and are essential to track the system's performance. Specifically, we explored monitoring, log analysis, tracing, and visualization tools. The first three are the primary tools of developers in most cases, including microservice-based systems [113].

In this chapter, we analyzed several tools and techniques used and developed by academia and industry to help developers debug their systems.

First, we explored debugging approaches for microservice-based systems and noticed that there is not yet a large variety of solutions implementing fault localization techniques. Such exceptions are solutions that apply delta debugging (state-based fault localization) and spectrum-based fault localization. Both of them focus on external aspects, although necessary, besides the application code and do not provide a fine-grained insight into the fault itself. We also explored record-and-replay and live debugging tools, which give developers some of the "power of debugging monoliths" back with limitations. At last, in this section, we also explored anomaly detection and root-cause analysis tools, which are essential to discuss, although not focusing on the application code due to their practicality in production scenarios.

Afterward, we explored debugging approaches for "traditional" (non-microservice-based) systems, as it contains most of the literature on debugging techniques, especially for distributed systems, of which microservices are a part. We focused on advanced fault localization techniques such as configuration, slice-based, and spectrum-based fault localization.

Finally, we discussed debugging assistance tools and approaches. Despite not being tools that actively find faults, they are widely used in industry and are essential to track the system's performance. Specifically, we explored monitoring, log analysis, tracing, and visualization tools. The first three are the primary tools of developers in most cases, including microservice-based systems [113].

To conclude this chapter, we want to point out that there are tools dedicated to different kinds of systems. Therefore one single tool could not cover all bases of debugging, and combining

tools and techniques seems to be the way to develop a novel solution. In particular, we draw the reader's attention to logging and tracing techniques, which provide rich information about the system execution at a low level. The more information is present, the more likely are we to find some critical detail that helps pinpoint a given faulty component. Our highlight regarding debugging techniques is the spectrum-based fault localization. It is a fascinating approach we intend to focus on while developing our strategy and designing our solution.

Chapter 4

Debugging Tool Implementation

In the previous chapter, we discussed the parameters of our solutions, the goals we mean to achieve, and the issues and challenges we expect to face. To go from logs of microservices systems executions to a ranking exposing possible faulty entities requires a certain degree of manipulation of the information. Besides the SFL technique itself, we soon realize that the data must be processed when it is in a plain log format. For this reason, our implementation is divided into two parts, one dedicated to processing the logs collected from the microservice system and the other dedicated to extracting entities and performing analysis to obtain the SFL rank. For the rest of the chapter, we discuss the prototype's architecture and each implementation part in its section. The code repository for our prototype is accessible at [62].

4.1 Architecture

At a high level, we see the architecture of our prototype divided into four components: log processor, entity parser, analytics, and ranking. In Figure 4.1 we present a UML components' diagram describing the relationships between the components.

The first component to address, the log processor, is a component that is not implemented from the core but instead configured and adapted to extract the log data in the proper format. The data should have a standard format, but different degrees of completeness in the entries are also expected. Therefore, each entry has minimal requirements, so the SFL technique can still be applied. In the following section, we analyze further in detail which are the mandatory and optional fields extracted from each log.

Afterward, the first component of the second part of the implementation, as mentioned before, is the entity parser. This component is responsible for parsing SFL entities from the extracted log data. Since we do not guarantee complete information, we must consider the two levels of entities that can be extracted, the service being executed and the method invoked. In Subsection 4.3.2, we detail the concrete implementation approach to adapt the extracted data to entities eligible for the SFL technique.

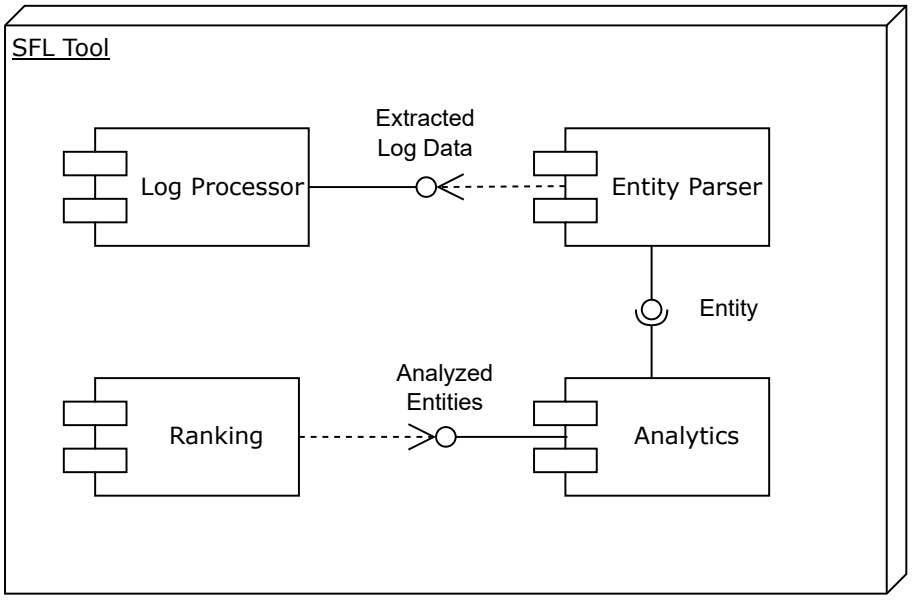

Figure 4.1: SFL Tool Architecture Component Diagram

The following component is responsible for analyzing the entity references and computing the *hit spectra* of each entity. This is the final step before applying the SFL algorithm.

The last step, performed by the ranking component, is to compute the probability of being faulty for each entity, given the SFL coefficient algorithm. Finally, the rank is created, ordering the entities by highest value (probability) and outputting the ranking to the end-user.

4.2 Logs Processor

Log processing is not a novel process, and in the observability area, several developments already exist that provide many capabilities to the users. We must analyze a considerable amount of logs and be able to parse and extract relevant information from them in a feasible time.

After some time spent researching, we encountered a tool that was able to fulfill our needs. As we already discussed in Chapter 2 (p. 5), the Logstash [20] tool provides us the means for log manipulation and streaming the data into various output formats. In addition, we realized that there was an open-source version, which we used as our baseline.

We used its Docker container image to run the tool, as it is quicker to start without any extra configurations. As we will refer to from now on as Logstash, the log processor has many capabilities beyond our scope and need. For our case, the essential features are its pipelines and configuration files, which we explain immediately.

The pipeline indicates a job that Logstash must perform, each job/pipeline being associated with a configuration file. It can be configured to distribute the available resources customarily through the active pipelines.

A configuration file indicates where the information comes from, what operations it does with it, and where does it output to. There are numerous possible input and output plugins for both input

and output, such as files, message queues, TCP/UDP servers, and many more. One can specify the flow of data with almost no restrictions. As for the operations possible to perform, they are also quite extensive. The standard data format is JSON, and from there, one can modify data values, perform conditional operations, and extract information from strings with Grok [18] patterns.

4.2.1 Configuration example

A sample configuration file can be observed in Listing 1.

In this example, the log data is retrieved from two possible sources. The first is from files in the local storage, which are located in the directory */data/file* and have the extension *.log*. The second is from a RabbitMQ server, which should be running in *localhost* with the port *5672*, and the data should be coming from the exchange *logstash-input*. As for the operations performed, a grok filter plugin is used to extract information from the logs. In fact, there are several patterns, which means that the logs expected should come in different formats. Different segments can be extracted into a field in each pattern if they match the pattern required. For example, if the first part of the log has the format of the timestamp ISO-8601, that part of the string that matched the template will be extracted to the field *timestamp*. The syntax for applying the intermediate patterns is *%{PATTERN:field_name}*, where *PATTERN* is a macro for a regular expression and *field_name* is the name of the field to be created if the data matches the pattern. Some patterns are more complex and contain others within, such as the case of *COMBINEDAPACHELOG* which captures an Apache-type log. The last operation performed on the extracted data removes some unused fields that are not attractive. Finally, the information is outputted. Since there is a conditional operation, the extracted data is only outputted if there is no tag indicating a failure parsing the logs in the grok filter. This means only the logs that fully match one of the above patterns are outputted. If that happens, the output is again the RabbitMQ server. The extracted data is sent to the exchange *logstash-output*. We obtain a powerful setup to process and extract logs into a parsed and structured object with just a couple of lines.

4.2.2 Log Processor Adaptation

Once the primary container is set up, one must create the appropriate configuration file for each scenario. This task is not heavy and provides excellent control over the manipulation of the information, making it relatively quick to obtain extracted and parsed log information from the logs.

The records are stored or output in JSON format to ease the posterior use of the extracted data since it is easily readable and manipulated.

After some consideration and experimentation, we arrived at a template of what information should be extracted from a log entry. Most of its fields are optional, as we do not expect such completeness and uniformity in the logs. However, it could hold information relevant for the developer when he is debugging from a faulty entity, so it has this information “close-by”. However, there are some mandatory fields, as we consider them the essential minimum for the SFL tool to work to create an entity reference from extracted data.

```
input {
  file {
    path => "/data/file/*.log"
  }
  rabbitmq {
    host => "localhost"
    port => 5672
    exchange => "logstash-input"
    codec => "plain"
  }
}
filter {
  grok {
    match => {
      "message" => [
        "%{TIMESTAMP_ISO8601:timestamp} %{LOGLEVEL:logLevel}
        ↪  %{JAVAFILE:className} - %{GREEDYDATA:logMessage}",
        "%{MONTHNUM:month}/%{YEAR:year} %{LOGLEVEL:loglevel} :
        ↪  \.*%{DATA:action}",
        "%{COMBINEDAPACHELOG}"
      ]
    }
  }
  mutate {
    remove_field => ["event", "@version", "log"]
  }
}
output {
  if "_grokparsefailure" not in [tags] {
    rabbitmq {
      host => "localhost"
      exchange => "logstash-output"
      port => 5672
    }
  }
}
```

Listing 1: Logstash Sample Configuration File

The provided guiding template for the outputted log format is in Listing 2.

4.3 SFL Tool

The core feature of our prototype is manipulating structured log data into components and applying the SFL technique to them. The uncertainty of log quality forces us to consider settings with

```
{
    "correlationID": "[REQUIRED] <string> of the request ID that started the invocation chain",
    "durationProcessing": "[OPTIONAL] <number> of the duration, in milliseconds, taken to fulfill the logged request/method",
    "spanID": "[OPTIONAL] <string> of the span ID of the service",
    "endpoint": "[OPTIONAL] <string> of the endpoint that invoked the logged activity",
    "httpCode": "[OPTIONAL] <integer> of the HTTP status code produced by the request",
    "instanceIP": "[OPTIONAL] <string> of the instance IP where the microservice is running",
    "methodInvocation": {
        "fileName": "[OPTIONAL] <string> of the file name/path where the method is invoked",
        "className": "[OPTIONAL] <string> of the class name/path where the method is invoked",
        "line": "[OPTIONAL] <integer> of the line of the logged activity",
        "methodName": "[OPTIONAL] <string> of the method invoked"
    },
    "logLevel": "[OPTIONAL] <string> of the log level (e.g: DEBUG, INFO, ERROR, 1, 2, 3)",
    "message": "[OPTIONAL] <string> of a message explaining the logged activity",
    "microserviceName": "[REQUIRED] <string> of the microservice name",
    "parentSpanID": "[OPTIONAL] <string> of the span ID of the parent service that invoked the current service",
    "timestamp": "[REQUIRED] <string> of the timestamp, in ISO-8601 format and in UTC time standard, when the logged activity occured",
    "user": "[OPTIONAL] <string> of the user name or id associated to the user that initiated the logged request"
}
```

Listing 2: Processed Log Template

incomplete information and adapt to it. To perform SFL, we must have the concept of an entity in the code. In the following steps, we detail step-by-step the implementation of our tool, from the raw structured log data to the SFL ranking.

4.3.1 Inputs

Production environments often have a high data flow and are automatic, i.e., the systems are deployed and executed without human interference. We chose a method of automatically receiving the parsed log data so that the tool could be integrated into a pipeline and receive continuous data streams in real-time. Our first option was using RabbitMQ [86] as a message broker for the messages containing the log data, as it is robust and well-supported.

In certain situations, using a data stream is not practical, for example, when a developer performs a static analysis of the code and logs. To keep our tool generic for multiple purposes, we also implemented file reading, allowing asynchronous analysis and smaller scenarios to be tested quicker.

4.3.2 Entity Reference Extraction and Generation

Once the log data is manipulable, the next step is to use it to construct the entity reference, which is the basis for applying the SFL technique. Considering the restraint of possible incomplete information in the logs, we implemented a two-level hierarchy for the entities, one for the service and the other for the method invocation inside the service.

In the code, there are components that, given a set of conditions, and variables, are executed. Those components, or entities, can be executed or not in passing or failing test cases. These are two foundational premises of SFL. In the context of microservices, we made two adaptions to these premises.

For the first adaptation, regarding the entities, we now consider two entities of different hierarchical levels, a service entity referred to when a service is invoked, and a method entity referred to when a method in that service is invoked. This is our response to the possibility of incomplete information. Even if it is not possible to point out the faulty method, we can point in the general direction and focus the attention on one single service in the system.

We consider executions of microservices and do not expect test cases for the second adaptation, as it is the traditional approach. Instead of passing and failing test cases, we expect "good" and "faulty" executions of the system. We consider the execution of the system every step since a request is received/generated in the system until it is fulfilled. The distinction between "good" and "faulty" executions relies on the system's behavior and if the output is considered correct or incorrect. Therefore, the tool expects two sources of log data, one for the logs regarding "good" executions and the other for the logs regarding "faulty" executions".

For each log data entry, we create a reference to the entity its mentions. At the very least, a service entity reference is always created for each log entry. The method entity reference is also created if the data contains information regarding the method invocation. This is the process for every log data entry received.

A diagram for the entity class can be observed in Figure 4.2. For each log entry, there is an instance of at least one entity, the service entity, and the method entity might also be instantiated.

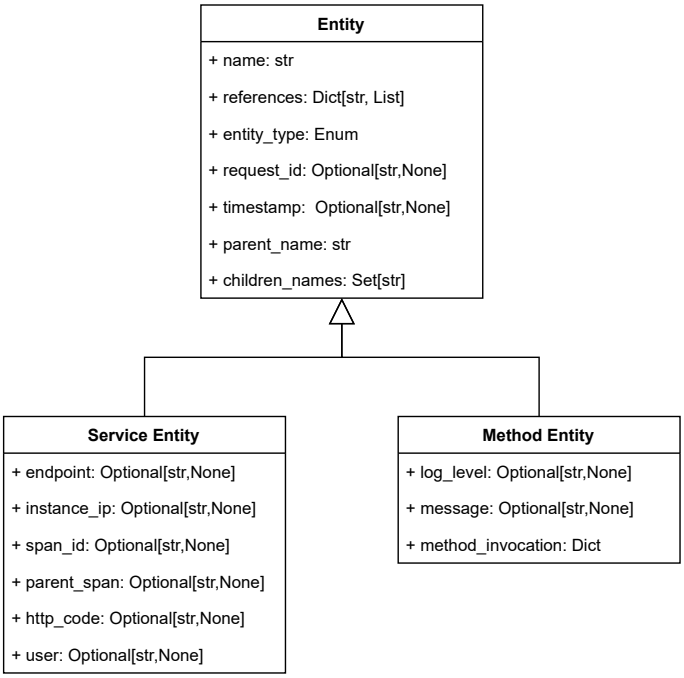

Figure 4.2: Entity Class Diagram with Python-like Typing

After collecting all the entity references, multiple references belong to the same entity because one service or method is rarely executed only once. The final step of entity extraction and generation is to merge the references into a compilation. To merge references of the same entity, they are hashed with their name, entity type, and parent name. We collect the specific information about the references and children's names during the merge because that is the different information between entity references.

The result is two sets of unique entities, with the same total number of entities present in the code and are captured by the logs, one extracted from the "good" logs and another from the "faulty" logs.

4.3.3 Entity Analytics

The analytics step of the tool is to obtain the hit spectra, a term used in SFL which means the number of unique times an entity is executed or not in "good" and "faulty" scenarios. Since we already have the entities' sets separated by the two scenarios, we perform the analytics for each.

For the "faulty" scenario entity set, we compute how many times it was executed in that scenario for each entity. At the same time, we keep track of the number of unique executions. Each unique execution is represented by its request ID or correlation ID. Nevertheless, all entity references without it are counted as single executions, as requests of their own, since they cannot be related to any other. This number of (unique) executions permits retrieving the number of non-executions for the same entity, as executions and non-executions are complementary.

The exact process is done for the "good" scenario, and in the end, for each entity, we obtain the count of unique executions and non-executions for both scenarios.

During the testing of the tool, we observed that the service entity often had a higher ranking than the method entities and consistently outranked every one of its children. Therefore, we implemented an attenuation step in the service entities' metrics in the analytics step. We explored different approaches discussed in Chapter 5 (p. 40), and the selected winner was the "average" approach. In this strategy, we traverse each service entity's children and obtain their average on each metric (number of times executed or not in "good" and "faulty" scenarios). We replace the service entity's metric value with its children's average for the same metric. This way, we can balance the service entity's impact on the ranking.

4.3.4 SFL Ranking

The final step is to obtain the SFL rank from each entity's analytics. As we explored in Chapter 3 (p. 17), there are more than one alternative to the coefficient algorithm. In this prototype we implemented several of them, namely Ochiai [13], Jaccard [11], Tarantula [44], Zoltar [2], O [69], O^P [69], Kulczynski2 [68], McCon [68], DStar (D*) [104], and Minus [109].

The tool also supports multiple metrics used by merging, currently performing the average or the median (average is the default). Since not all of them follow the standard probabilistic scale (0 to 1), we also implemented a normalization step for the metric values to be correctly merged.

Once each entity has its SFL value, we prepare the ranking list and order it with the highest valued entity on top.

This step is easily extensible, and more metrics can be added to customize the ranking operation.

4.3.5 Outputs

The ranking is outputted into a JSON file and a separate file containing the ranked entities' references. For each scenario, "good" and "faulty", its entities references are also registered in case of posterior analysis. Providing the ranking in this format is easily readable and manipulable in a programmatic way.

4.4 Summary

Our tool must be able to process information from the logs before it is possible to apply the SFL technique and generate a ranking. Our implementation has a first part dedicated to performing log processing and a second to implementing the SFL technique.

There are several options available to deal with log processing. Our choice was Logstash which has an open-source version and the necessary features for our case. We built a Docker container image for our usage and prepared the required setup for it to be used. Logstash works with pipelines that represent different jobs and can have the assigned resources dynamically allocated.

A configuration file in each job details how the information is retrieved. It can be from files, message queues, web servers, and many more. It also details operations to perform on the data read; it can be parsing the string data into meaningful variables, manipulating the existing fields, shape the output data into a structured object. It can also detail how the processed data is outputted, having the same options as it does with the input. The significant number of possibilities is a great plus for our work since it provides a generic and highly customizable way of processing log data into usable data for our tool.

Once the log data for "good" and "faulty" scenarios is available, it is processed, and for each log entry, the tool extracts an entity reference. An entity could be one of two types, service or method. The service entity represents a higher-level component of the system, and the method entity represents a child component of the service. After processing all log entries, they are compiled and merged into two sets of unique entities. Then each entity is analyzed by counting the times it is executed in each scenario. At the same time, in each scenario, the tool keeps track of unique executions, represented by the request IDs or by the number of references without them. This permits us to infer the number of times each entity has not been executed and complete the analysis. Finally, the last step is to apply the SFL coefficient algorithms to each entity's analytics and arrange them by their SFL value in descending order. The ranking is the writing on a JSON file, completing the process.

With the complete pipeline, we can now integrate it with microservice logs and produce rankings. Our tool is now ready to be tested after performing some simple test cases and fixing any lingering issues.

Chapter 5

Evaluation

Once we have implemented our prototype, we must test it to be validated. Therefore, in this chapter, we first detail the setup developed for the evaluation, and afterward, we analyze the results obtained from running the test scenarios we created.

5.1 Evaluation Setup

We created this tool to be compatible with microservice systems, and therefore we decided to test the tool with actual data and logs produced by a microservice system. The application chosen for our evaluation was Instana's [36] Robot Shop [37].

5.1.1 Test Application

The Robot Shop is a web application for the online shopping of robots composed of 12 microservices, 7 of which were created for the application. Therefore we could alter their code (the remaining services are databases and message queues). The application is also polyglot, supporting our intention to build a generic, agnostic tool.

In this blog post [38], the authors describe the application as having a single web page using AngularJS (1.x), having its resources served by Nginx, which performs as a reverse proxy for the backend microservices as well. Since the microservices are polyglot, they make use of different frameworks. MongoDB stores data of the product catalogue and registered users, MySQL is responsible for looking up shipment information, and Redis holds the data about active shopping carts. RabbitMQ is used to process the order pipeline.

Besides supporting services for databases and message queuing, the business logic is spread in these seven services:

- **User** - Deals with user accounts and order history
- **Payment** - Deals with payments and forward to payment gateways
- **Catalogue** - Deals with robots availability and storage

- **Cart** - Deals with each user's current cart
- **Shipping** - Deals with shipment costs
- **Ratings** - Deals with robot ratings
- **Web** - Deals with the shop frontend and proxy the requests to the microservices

5.1.2 Log Extraction And Processing

Upon analyzing the services' code and logs when executed, we quickly understood that some information gaps in the logs would affect how the tool could find responsible entities. First, only two of the seven services observed contained request/correlation IDs in their logs, making it insufficient to relate requests between the rest. Second, no service would log the invocated method during the log's generation. We foresaw both problems as the tool was developed, and we found alternative solutions for them.

Since having a request id is unavoidable, problem one is that we cannot assume a relation between requests. We would not be able to do it without extracting more information, which is highly intrusive. Instead, we have the worst case: every service execution has its request. This means that instead of the intended correspondence of test cases in the traditional SFL approach to requests in our approach, we have each invocation of the entity belonging to its request. Regardless, we can separate good and faulty executions of the system and, therefore, separate entities into the two categories.

As for the second problem, upon further inspection of the logs generated by the services, we detected that all produced logs were present at the endpoint they were serving. By analyzing the code, we decided that we could meet halfway and assume that the method invocations could be represented by the endpoints or, more specifically, the code executed when they are called. Most endpoints were successfully present in the injected faults, but not all. Endpoints also introduced some noise from a service logged by another service, which is not ideal to "blame" a faulty entity.

We ran the load generator already available in the application demo to produce logs. We executed each scenario for 10 minutes to provide a high number of log entries. Since the execution was local, we only had one instance for each service, but it was sufficient to produce enough logs.

Once able to configure and collect logs from the application, we prepared a configuration to extract information from the logs (as explained in Chapter 4 (p. 31)'s Log Processor section). Each service provided logs in a different format and structure, which the log processor allowed not to become a challenge.

5.1.3 Fault Injection

To assess the ability of the tool to localize faults, we collected logs from scenarios where the system was running normally, unaffected, and others where we injected faults into the code of the services.

There was no limit, at first, to how many faults we would inject into the system to evaluate the performance. However, as we see in [3], and due to the size of the microservice system, we explored the injection of faults up to 5, and the results we obtained revealed to be sufficient for conclusion making. In addition, we were curious to see if the fault's distribution in the services impacted the tool's ability to detect faults, so we ran scenarios where the faults were distributed up to 5 services. For each scenario, we executed four combinations to obtain results with different sets of services. We chose this number of combinations to provide a sufficient variety of results without losing too much time exploring every possible combination, which our time frame would not allow. Since we have from one to five fault scenarios, and they are distributed in one to five services, the result is fifteen different categories. The four combinations per category make for 60 executions in the scenario suite.

The injected faults had different natures, as the services had different settings. Some were composed of injecting null values in a variable of a request body. Others were misspelling or altering a URL used for an intermediate request, changing constant values used in conditional statements, changing initialization values of booleans, and changing returns values. Our rationale was to inject errors that would not necessarily break the system altogether but be executed, perform erroneous behavior, and test whether our tool could detect it. Examples of the injected faults are in Listings 3, 4, 5, 6, and 7.

```
103 // return all users for debugging only
104 app.get('/users', (req, res) => {
105     if(mongoConnected) {
106         usersCollection.find().toArray().then((users) => {
107             // res.json(null); <-- The fault is inserted here
108             res.json(users);
109         }).catch((e) => {
110             req.log.error('ERROR', e);
111             res.status(500).send(e);
112         });
113     } else {
114         req.log.error('database not available');
115         res.status(500).send('database not available');
116     }
117 });
```

Listing 3: Example of a null value injection in the user service in server.js

5.1.4 Scenario Generation

Each scenario has a different combination of services in which the faults have been injected. We test four scenarios for each category, classified by the number of faults and services containing them. For simplicity, we used the terminology **X**f**Y**s, in which **X** stands for the number of faults

```
$scope.addToCart = function() {
// var url = '/api/cart/add/' + currentUser.uniqueid + '/' +
 ↪  $scope.data.quantity + '/' + $scope.data.product.sku; <--
 ↪  The fault is inserted here
var url = '/api/cart/add/' + currentUser.uniqueid + '/' +
 ↪  $scope.data.product.sku + '/' + $scope.data.quantity;
console.log('addToCart', url);
...
```

Listing 4: Example of URL tampering in the web service in controller.js

```
// update/create cart
app.get('/add/:id/:sku/:qty', (req, res) => {
    // check quantity
    var qty = parseInt(req.params.qty);
    if(isNaN(qty)) {
        req.log.warn('quantity not a number');
        res.status(400).send('quantity must be a number');
        return;
    // } else if(qty < 2) { <-- The fault is inserted here
    } else if(qty < 1) {
        req.log.warn('quantity less than one');
        res.status(400).send('quantity has to be greater than
         ↪  zero');
        return;
    }
...
```

Listing 5: Example of a modification of a constant value in a conditional statement in the cart service in server.js

and **Y** for the number of services the faults were distributed. For example **4f3s** means that this category is for scenarios containing four faults distributed in three different services. In Table 5.1, we present the generated combination for each scenario and category selected randomly.

5.1.5 Performance Evaluation

To evaluate each scenario's performance, we focused on measuring the accuracy of the scenario according to the ranking outputted. We used the combination of Ochiai and Jaccard metrics, best and second-best techniques as seen in [1].

There is a set of faulty entities for each scenario we test due to the injection of faults. The faulty entity's detection accuracy depends on the distance (delta) to the top of the ranking produced. For

```python
...
# has_shipping = True <-- The fault is inserted here
has_shipping = False
for item in cart.get('items'):
    if item.get('sku') == 'SHIP':
        has_shipping = True

if cart.get('total', 0) == 0 or has_shipping == False:
    app.logger.warn('cart not valid')
    return 'cart not valid', 400
...
```

Listing 6: Example of a boolean value modification in the payment service in payment.py

```php
/**
 * @Route("/fetch/{sku}", methods={"GET"})
 */
public function get(Request $request, string $sku): Response
{
    try {
        if (!$this->ratingsService->ratingBySku($sku)) {
            throw new NotFoundHttpException("$sku not found");
        }
    } catch (\Exception $e) {
        throw new HttpException(500, $e->getMessage(), $e);
    }

    // return new JsonResponse(['avg_rating' => 1.42,
    ↪  'rating_count' => 420]); <-- The fault is inserted here
    return new
    ↪  JsonResponse($this->ratingsService->ratingBySku($sku));
}
```

Listing 7: Example of a return value modification in the ratings service in RatingsApiController.php

the fairness of the evaluation, once an entity has been evaluated, it is not considered present in the ranking for the remaining entities' evaluation. This means that if we expect the presence of two faulty entities and they occupy first and second places, their delta is zero in both cases, resulting in an accuracy of 100% for that scenario.

In addition, we also add up the accuracy of the parent (service), as the service is more easily found to be faulty than the method invocation. Notwithstanding, it is added with a weight to maintain balance.

Table 5.1: Set of scenario combinations generated for each category of number of faults and fault distribution

Category	Combinations			
1f1s	cart	catalogue	user	web
2f1s	cart	catalogue	user	web
2f2s	cart-catalogue	payment-ratings	shipping-user	web-cart
3f1s	cart	ratings	shipping	user
3f2s	cart-catalogue	payment-ratings	shipping-user	web-cart
3f3s	cart-catalogue-payment	ratings-shipping-user	web-cart-catalogue	payment-shipping-ratings
4f1s	cart	catalogue	ratings	web
4f2s	cart-catalogue	payment-ratings	shipping-user	web-cart
4f3s	cart-catalogue-payment	ratings-shipping-user	web-cart-catalogue	payment-shipping-user
4f4s	cart-catalogue-payment-ratings	shipping-user-web-cart	payment-web-catalogue-user	cart-ratings-shipping-payment
5f1s	cart	shipping	user	web
5f2s	cart-catalogue	payment-ratings	shipping-user	web-cart
5f3s	cart-catalogue-payment	ratings-shipping-user	web-cart-catalogue	payment-shipping-ratings
5f4s	cart-catalogue-payment-ratings	shipping-user-web-cart	payment-shipping-user-cart	catalogue-ratings-web-cart
5f5s	cart-catalogue-payment-ratings-shipping	user-web-cart-catalogue-payment	ratings-shipping-user-web-payment	cart-catalogue-web-ratings-payment

We consider that if the method is not present in the ranking, the parent service still points in the general direction of the method, so it must be accounted for.

We also consider that if the parent's weight is not balanced, a service found guilty will influence all the methods children that are not to be blamed.

Therefore, our conclusion was to divide the parent's accuracy by the number of entities in the ranking before adding it to the method entity's accuracy.

For each scenario, the total accuracy is the average of the faulty entity's accuracy expected to be detected.

For example, we have a test scenario with the faulty entities "A", "B" and "C", whose parents are the service entities "D", "E", and "F", respectively. After running the tool, the ranking has ten entities, and each entity occupies the corresponding place:

- Method entity "A" occupies first place
- Method entity "B" occupies fifth place
- Method entity "C" is not part of the rank
- Service entity "D" occupies second place
- Service entity "E" occupies third place
- Service entity "F" occupies sixth place

"A" has a zero delta since it occupies the first place. Therefore the accuracy of detecting "A" is 100%.

"B" has a delta of three because it occupies the fifth place (delta is four), and "A" has been processed before. Its parent, "E", has a delta of one, as it occupies the third place, and "A" has been

processed before. "B" has the accuracy of $1 - 3/10 = 0.7$, which is 70%. "E" has the accuracy of $1 - 1/10 = 0.9$, which weighed down is $0.9/10 = 0.09$, resulting in 9%. The final accuracy of "B" is 79%.

"C" is not part of the ranking, so its accuracy is 0%. However, its parent, "F", is present and has a delta of three since it occupies sixth place (delta is five), and "A" and "B" have been processed before. "F" has the accuracy of $1 - 3/10 = 0.7$, which weighed down is $0.7/10 = 0.07$, resulting in 7%. The final accuracy of "C" is 7%.

This scenario's total accuracy is the average accuracy of all faulty entities, which is 62%.

In addition to the accuracy, we registered the position in the ranking and the SFL value attributed to each faulty entity in each scenario.

As an extra step, we tracked the application's execution time in the best configuration (service attenuation with average) to benchmark it. The machine used in the evaluation process possesses 16 GB RAM and Intel™ Core i7-7700 2.8 GHz with 4 cores.

5.1.6 Additional Evaluation Points: Entity Ties

As it is common in SFL, and we confirmed so in our results, there are often ties with entity values; therefore, they have placed adjacently in the ranking. We analyzed different "tie-breaking" strategies that represent the order a developer would analyze each entity when debugging the system. Our study examined four strategies: best case, worst case, average, and "as-is". The best case means that the entity is always the first to be considered independently, so it has the highest-ranking position of the entities with the same value. For example, if the first three entities would have the same value, all of them would be considered in the first place when evaluated. The worst-case means that the entity is always the last to be considered, so it has the lowest ranking of the entities with the same value. For example, if the first three entities would have the same value, all of them would be considered in the third place when evaluated. The average case means that the entity is always in the average place, so it has the average ranking position of the entities with the same value. For example, if the first three entities had the same value, all of them would be considered in the average position, which is second, when evaluated. The "as-is" case processes the ranking as it is, without any tie-breaking considerations.

5.1.7 Additional Evaluation Points: Endpoint Coverage Percentage

Another evaluation point we considered relevant was the tool's performance with varying grades of endpoints/method invocation presence and how it impacts the accuracy if there is a threshold of necessary endpoint presence. We executed the same suite with a different number of services with endpoints as method invocations to evaluate this. We considered a presence of 100%, 75%, 50%, 25%, and 0% of the services with endpoints. We also tested along with the 0% endpoints scenario, the same suite but pointing only to services so that we can evaluate the tool's performance with only the service information available.

5.2 Results And Their Analysis

As we mentioned before, our evaluation consisted of analyzing the rankings produced by executing the tool on logs of the microservice application in different settings. From one to five faults, distributed in one to five various services, we analyzed the results in different aspects, focusing on the accuracy of each scenario. While the results in this section represent the average accuracy in each category of scenarios, the individual accuracies of each specific scenario are presented in Appendix A (p. 64).

5.2.1 Scenario Suite

Results In every run, we evaluate the accuracy of each scenario and then extract useful information from them. We compute the average accuracy for the category with **X** of faults and with **Y** of services the faults have been distributed. Furthermore, for each specific scenario and these two compilations, we compute the minimum and maximum accuracies and the average minimum and maximum of all the scenarios. As a general pointer, we also calculate the global average accuracy of all scenarios together.

In the first execution of the tool, we could not extract the method invocations in the endpoints in the logs of three services, *payment*, *shipment*, and *ratings*. This constraint implied a reduction in the number of entities identifiable in the results. The results in Table 5.2 confirm just that.

Table 5.2: Evaluation results for the scenario suite in the first log extraction

Category	Accuracy	Min Accuracy	Max Accuracy
1f	70.73%	45.60%	94.60%
2f	48.76%	2.44%	75.10%
3f	36.16%	2.40%	78.20%
4f	42.33%	2.50%	80.40%
5f	37.81%	2.50%	83.40%
1s	41.07%	2.40%	94.60%
2s	43.76%	2.40%	80.40%
3s	41.89%	2.50%	83.40%
4s	43.59%	13.20%	65.00%
5s	41.85%	29.90%	62.20%

Global average
42.34%

Category	Accuracy	Min Accuracy	Max Accuracy
1f1s	70.73%	45.60%	94.60%
2f1s	53.53%	33.90%	75.10%
2f2s	43.99%	2.44%	72.20%
3f1s	24.75%	2.40%	66.50%
3f2s	45.68%	2.40%	78.20%
3f3s	38.05%	2.50%	73.60%
4f1s	35.68%	2.50%	76.50%
4f2s	44.55%	2.50%	80.40%
4f3s	47.13%	21.50%	80.40%
4f4s	41.95%	13.20%	65.00%
5f1s	20.65%	2.50%	40.40%
5f2s	40.83%	2.50%	79.70%
5f3s	40.50%	2.50%	83.40%
5f4s	45.23%	33.70%	56.00%
5f5s	41.85%	29.90%	62.20%
	AVG	13.34%	72.28%

The results in Table 5.3 represent the evaluation on the second batch of logs extracted. We would not need another rerun of log extraction, as we extracted all the information available in this application.

Table 5.3: Evaluation results for the scenario suite in the second, and final, log extraction

Category	Accuracy	Min Accuracy	Max Accuracy
1f	75.88%	56.90%	95.90%
2f	62.05%	30.70%	84.20%
3f	64.50%	28.30%	87.80%
4f	61.73%	19.00%	84.40%
5f	59.20%	19.00%	87.80%
1s	57.64%	19.00%	95.90%
2s	66.53%	30.70%	84.40%
3s	68.07%	50.10%	86.90%
4s	58.60%	36.20%	74.10%
5s	60.70%	48.80%	69.10%

Global average
62.43%

Category	Accuracy	Min Accuracy	Max Accuracy
1f1s	75.88%	56.90%	95.90%
2f1s	58.63%	35.80%	83.40%
2f2s	65.48%	30.70%	84.20%
3f1s	60.80%	28.30%	87.80%
3f2s	69.63%	31.70%	84.20%
3f3s	63.08%	50.10%	78.90%
4f1s	51.00%	19.00%	84.40%
4f2s	67.45%	47.40%	84.40%
4f3s	72.28%	66.30%	84.40%
4f4s	56.20%	36.20%	70.60%
5f1s	41.88%	19.00%	87.80%
5f2s	63.55%	48.10%	83.90%
5f3s	68.85%	50.80%	86.90%
5f4s	61.00%	43.80%	74.10%
5f5s	60.70%	48.80%	69.10%
	AVG	40.86%	82.67%

Analysis In this first run, in Table 5.2, we understand that the more faults are introduced, the lower the accuracy. In this particular run, the accuracy drops from 70% to 38%, almost half. Here we see that the fault distribution in the services does not seem to affect the performance, as the average accuracy in all **Y**s is quite similar.

In general, the results are inferior, and this is due to the apparent lack of information in the logs for us to extract the method invocation of the endpoints.

The low results lead us to focus on the missing method invocation from the services mentioned earlier. Looking deeper into the logs, we detected and extracted endpoint information from those services, corresponding to the method invocation.

In Table 5.3, the results improved considerably in this new iteration, with the global average increasing by about 20 points. The accuracy gap between scenarios with a different number of faults has closed significantly. Now it goes from 76% to 59%, and it is worth mentioning that while the upper bound has increased slightly, the lower bound rose the most. The worst-performing scenario also improved its accuracy from 2.4% to 19%. The average bound puts the accuracy above 50% chance, with the average minimum accuracy at 41% and the average maximum accuracy at 83%.

The number of services seems to oscillate the accuracy more than previously but without any connection. It goes between 57% and 67%, a ten-point distance, peaking with the distribution among three services and having the lowest percentages in the scenarios with 1 and 4 services containing the faults.

Still in the lowest-performing scenarios, with 19% of accuracy, we analyzed the results and the logs and found the root cause for this. First, since we extracted more information from the logs than previously, the number of entities extracted before was 37 now rose to 49. Some of them are

irrelevant to the business logic and therefore not targeted by the fault injection, but extracting the remaining helpful information was the only option. The second cause for this low accuracy was that the faults injected into the web service were not properly logged. This service makes many intermediate calls for the other services and logs those calls. However, most of the calls to its endpoints are not logged, so injecting a fault in one of its endpoints would not be captured because any or most logs produced in that endpoint did not specify the endpoint it was invoked from. This phenomenon is a natural consequence of assuming endpoints as method invocations. Since this has occurred in very few scenarios, we accepted this phenomenon and the drop in accuracy.

5.2.2 Service Entity Weight Attenuation

Due to having a mixed ranking with different hierarchy levels, we found an issue in the rankings. Service entities were occupying the higher ranking levels due to the aggregation of all its children, plus any extra references that did not generate a method entity. This meant that the service entity containing a faulty method entity would be positioned above the child, therefore hurting the ranking the method entity would have.

To solve this, we had multiple ideas that we tested. For once, divide the service entity metric counts (the number of times (not) executed in good (or faulty) scenarios) by the number of children it had. Here the idea is to cut down the numbers and analyze the impact.

Another strategy was to consider the children's average for each metric count and replace the service's original value. Here the idea is that the service cannot outrank the best child it has.

The third strategy used the maximum value of the children count assigned to the service. The idea here is a softer reduction of the service impact by replacing the sum of the children's metric count with their maximum value only.

Results The results of the first, second, and third strategy implemented are in Table 5.4, Table 5.5, and Table 5.6, respectively.

Analysis In the first strategy, in Table 5.4, we observe the results when the service entities are weighted down by dividing the count (of each metric, executed/not, good/bad) by the number of children. This operation is equivalent to multiplying a factor which is the children's average, by the service's metric count. For example, being g_e the total of the service metric, and a, b, c, each of its child metric count, this means that $g\_e = a+b+c$. The average of the children is then $avg = (a+b+c)/3$. There the factor to weight would be $f = avg/g\_e = ((a+b+c)/3)/(a+b+c)$, which is equivalent to $1/3$. Therefore, multiplying $f * g\_e$ is the same as $g\_e/3$.

As analyzed, the impact is negligible, as the statistics remain closely similar. It is even slightly worse, indicating that it is not a viable option for attenuation. As observed in SFL, the reason for this is that the proportion is more relevant than the count. Entities being executed in faulty runs are more relevant than in good runs. So if one entity has five times more references in faulty runs than good ones, it will still have that proportion if the metric values are divided equally.

Table 5.4: Evaluation results for the scenario suite using the service attenuation with division

Category	Accuracy	Min Accuracy	Max Accuracy
1f	75.88%	56.90%	95.90%
2f	62.05%	30.70%	84.20%
3f	64.13%	28.30%	87.80%
4f	61.14%	19.00%	84.40%
5f	58.76%	19.00%	87.80%
1s	57.83%	19.00%	95.90%
2s	66.56%	30.70%	84.40%
3s	67.33%	49.40%	86.90%
4s	57.04%	33.50%	74.10%
5s	59.28%	46.60%	67.40%

Global average
62.05%

Category	Accuracy	Min Accuracy	Max Accuracy
1f1s	75.88%	56.90%	95.90%
2f1s	58.63%	35.80%	83.40%
2f2s	65.48%	30.70%	84.20%
3f1s	60.80%	28.30%	87.80%
3f2s	69.38%	30.70%	84.20%
3f3s	62.20%	49.40%	78.90%
4f1s	51.28%	19.00%	84.40%
4f2s	67.45%	47.40%	84.40%
4f3s	71.55%	64.10%	84.40%
4f4s	54.30%	33.50%	68.40%
5f1s	42.55%	19.00%	87.80%
5f2s	63.93%	47.40%	83.90%
5f3s	68.25%	50.10%	86.90%
5f4s	59.78%	41.10%	74.10%
5f5s	59.28%	46.60%	67.40%
	AVG	40.00%	82.41%

Table 5.5: Evaluation results for the scenario suite using the service attenuation with average

Category	Accuracy	Min Accuracy	Max Accuracy
1f	82.73%	58.40%	100.00%
2f	67.39%	32.70%	96.20%
3f	70.24%	32.20%	97.40%
4f	67.27%	20.40%	91.60%
5f	64.97%	20.50%	97.40%
1s	63.04%	20.40%	100.00%
2s	72.44%	32.70%	96.20%
3s	74.57%	55.80%	91.70%
4s	63.54%	39.60%	78.70%
5s	66.40%	52.70%	75.50%

Global average
68.14%

Category	Accuracy	Min Accuracy	Max Accuracy
1f1s	82.73%	58.40%	100.00%
2f1s	63.95%	39.30%	90.60%
2f2s	70.83%	32.70%	96.20%
3f1s	66.50%	32.20%	97.40%
3f2s	75.25%	33.70%	96.20%
3f3s	68.98%	56.50%	83.90%
4f1s	55.10%	20.40%	91.60%
4f2s	73.43%	52.30%	88.90%
4f3s	79.60%	71.90%	88.80%
4f4s	60.95%	39.60%	77.60%
5f1s	46.93%	20.50%	97.40%
5f2s	70.28%	55.10%	87.80%
5f3s	75.13%	55.80%	91.70%
5f4s	66.13%	49.90%	78.70%
5f5s	66.40%	52.70%	75.50%
	AVG	44.73%	89.49%

In the second strategy, in Table 5.5, we observed the results when we performed the service entity attenuation by replacing the service metric with the average of its children, meaning $g_e = avg = (a+b+c)/3$.

In this run, the accuracy rises, and the global average hits 68%. There is even one scenario

Table 5.6: Evaluation results for the scenario suite using the service attenuation with maximum

Category	Accuracy	Min Accuracy	Max Accuracy
1f	79.33%	56.90%	100.00%
2f	64.46%	29.90%	93.60%
3f	67.52%	30.90%	95.80%
4f	64.03%	19.00%	86.40%
5f	61.76%	19.00%	95.80%
1s	60.58%	19.00%	100.00%
2s	69.65%	29.90%	93.60%
3s	70.78%	53.00%	89.30%
4s	60.16%	37.10%	73.90%
5s	61.58%	50.30%	69.40%

Global average
65.05%

Category	Accuracy	Min Accuracy	Max Accuracy
1f1s	79.33%	56.90%	100.00%
2f1s	60.83%	36.30%	84.40%
2f2s	68.10%	29.90%	93.60%
3f1s	64.68%	31.10%	95.80%
3f2s	72.95%	30.90%	93.60%
3f3s	64.93%	53.00%	80.10%
4f1s	52.45%	19.00%	85.40%
4f2s	70.50%	48.80%	86.40%
4f3s	75.83%	67.20%	85.90%
4f4s	57.35%	37.10%	70.90%
5f1s	45.60%	19.00%	95.80%
5f2s	67.05%	49.50%	85.90%
5f3s	71.60%	53.00%	89.30%
5f4s	62.98%	46.90%	73.90%
5f5s	61.58%	50.30%	69.40%
	AVG	41.93%	86.03%

with 100%, as its faulty entity is correctly placed in the first place of the ranking. The average minimum is about 45%, and the average maximum is about 89.5%, the highest values registered until now.

This is a clear improvement on the previous approach and seems to be the best candidate thus far.

The third and last strategy, in Table 5.6, shows an improvement compared to the default situation, without attenuation of the service entities. However, it still is left behind the previous approach in all aspects. The global average is about 3% behind, the average lower bound is 42%, and the average upper bound is 86%.

We verified that the average strategy produced the best outcome and effectively attenuated the service entity weight based on the results.

5.2.3 Ranking Ties

One aspect of the ranking produced by the tool is that it does not naturally consider a tie when two entities have the same SFL value. Each entity occupies whatever position resulting from sorting the ranking list. We also evaluated entities based on their default ranking, or “as-is”. However, this might not necessarily mean the order in which the developer will observe each entity if they are tied. When evaluating an entity, we considered that, in a tie, the developer could analyze that entity first (best-case scenario), last (worst-case scenario), or consider a middle ground (average-case scenario). Therefore, we evaluated three tie-breaking strategies, besides the default “as-is”, as observed in the previous subsections.

Results Here are the results for the best-case, worst-case, and average-case scenarios. They are detailed in Table 5.7, Table 5.8, and Table 5.9, respectively.

Table 5.7: Evaluation results for the scenario suite using the best-case tie-breaking strategy

Category	Accuracy	Min Accuracy	Max Accuracy
1f	82.73%	58.40%	100.00%
2f	67.51%	32.70%	96.20%
3f	70.61%	32.20%	97.40%
4f	67.51%	20.40%	92.70%
5f	66.14%	20.50%	97.40%
1s	63.41%	20.40%	100.00%
2s	73.01%	32.70%	96.20%
3s	74.92%	57.20%	91.70%
4s	65.34%	42.90%	79.10%
5s	67.65%	55.40%	76.40%

Global average
68.71%

Category	Accuracy	Min Accuracy	Max Accuracy
1f1s	82.73%	58.40%	100.00%
2f1s	64.20%	39.30%	91.60%
2f2s	70.83%	32.70%	96.20%
3f1s	66.75%	32.20%	97.40%
3f2s	75.75%	35.70%	96.20%
3f3s	69.33%	57.90%	83.90%
4f1s	55.63%	20.40%	92.70%
4f2s	73.90%	53.70%	88.90%
4f3s	81.03%	73.00%	88.80%
4f4s	62.88%	42.90%	78.70%
5f1s	47.73%	20.50%	97.40%
5f2s	71.58%	57.20%	87.80%
5f3s	75.93%	57.20%	91.70%
5f4s	67.80%	53.10%	79.10%
5f5s	67.65%	55.40%	76.40%
	AVG	45.97%	89.79%

Table 5.8: Evaluation results for the scenario suite using the worst-case tie-breaking strategy

Category	Accuracy	Min Accuracy	Max Accuracy
1f	82.73%	58.40%	100.00%
2f	66.60%	30.60%	96.20%
3f	68.73%	28.90%	97.40%
4f	64.69%	20.40%	88.90%
5f	62.21%	20.50%	97.40%
1s	62.34%	20.40%	100.00%
2s	71.20%	28.90%	96.20%
3s	72.15%	52.30%	91.70%
4s	58.54%	30.50%	78.70%
5s	61.93%	45.30%	71.50%

Global average
66.13%

Category	Accuracy	Min Accuracy	Max Accuracy
1f1s	82.73%	58.40%	100.00%
2f1s	62.90%	39.30%	86.40%
2f2s	70.30%	30.60%	96.20%
3f1s	66.00%	32.20%	97.40%
3f2s	74.05%	28.90%	96.20%
3f3s	66.15%	52.30%	83.90%
4f1s	53.80%	20.40%	87.50%
4f2s	72.38%	50.20%	88.90%
4f3s	77.43%	67.10%	88.80%
4f4s	55.15%	30.50%	72.70%
5f1s	46.25%	20.50%	97.40%
5f2s	68.08%	50.20%	87.80%
5f3s	72.88%	53.00%	91.70%
5f4s	61.93%	41.30%	78.70%
5f5s	61.93%	45.30%	71.50%
	AVG	41.35%	88.34%

Table 5.9: Evaluation results for the scenario suite using the average-case tie-breaking strategy

Category	Accuracy	Min Accuracy	Max Accuracy
1f	82.73%	58.40%	100.00%
2f	67.13%	32.70%	96.20%
3f	69.51%	31.60%	97.40%
4f	66.38%	20.40%	91.60%
5f	63.80%	20.50%	97.40%
1s	62.89%	20.40%	100.00%
2s	72.02%	31.60%	96.20%
3s	73.48%	54.40%	91.70%
4s	61.35%	35.90%	77.40%
5s	64.33%	49.70%	73.70%

Global average
67.33%

Category	Accuracy	Min Accuracy	Max Accuracy
1f1s	82.73%	58.40%	100.00%
2f1s	63.43%	39.30%	88.50%
2f2s	70.83%	32.70%	96.20%
3f1s	66.25%	32.20%	97.40%
3f2s	74.73%	31.60%	96.20%
3f3s	67.55%	54.40%	83.90%
4f1s	55.10%	20.40%	91.60%
4f2s	72.98%	51.60%	88.90%
4f3s	78.90%	69.80%	88.80%
4f4s	58.53%	35.90%	75.40%
5f1s	46.93%	20.50%	97.40%
5f2s	69.55%	53.00%	87.80%
5f3s	74.00%	54.40%	91.70%
5f4s	64.18%	46.70%	77.40%
5f5s	64.33%	49.70%	73.70%
	AVG	43.37%	88.99%

Analysis The best-case strategy, in Table 5.7, shows a slight improvement in the accuracy overall. We see that some faulty entities do happen to tie with others in the rankings and are not always placed first, and this strategy bumps them to the best place. We also analyzed the rankings and verified that several more ties are happening in the ranking, as it is natural. Still, only a handful are the ones related to the faulty entities we evaluate. Therefore the tie-breaking in this setting is limited but still note-worthy.

In the worst-case strategy, in Table 5.8, it does the reverse, bumping tied entities to the last position. The average accuracy drops here, naturally, by about 2%. It is still not very relevant, showing that some ties spread across many entities, but it is not very impactful.

The average accuracy, in Table 5.9, also drops in the average-case strategy, but less than 1%. It is the closest strategy to the default "as-is".

These different strategies show how differently the results from our tool could be interpreted and how our evaluation fits the possibilities of a developer debugging. Analyzing these results, the default behavior seems to fall within their set boundaries, which we consider the desirable outcome.

5.2.4 Method Entity Percentage

So far, we have analyzed the tool's performance when all services provide information about the method invocation in the endpoints. As we mentioned before, a relevant challenge for the tool is to observe its performance with varying percentages of services providing this information. Therefore we filtered this in several executions to simulate its absence from the tool. This means

explicitly that in each log entry, we keep the information relative to the service but discard the information relative to the children (methods).

Results Considering only 75% of the services with endpoints (as method invocations), we allowed only this information from the cart, catalogue, payment, ratings, and shipment services, given that it is the closest fraction to that percentage (five out of seven). The results are shown in Table 5.10.

Table 5.10: Evaluation results for the scenario suite using 75% of services with endpoints as method invocations

Category	Accuracy	Min Accuracy	Max Accuracy
1f	41.00%	2.60%	93.10%
2f	45.89%	2.50%	85.90%
3f	53.69%	2.60%	84.10%
4f	50.05%	3.90%	85.90%
5f	47.30%	2.70%	79.40%
1s	43.06%	2.50%	93.10%
2s	54.55%	26.80%	84.10%
3s	56.54%	40.10%	71.80%
4s	41.84%	23.80%	67.80%
5s	43.75%	36.10%	53.00%

Global average
48.70%

Category	Accuracy	Min Accuracy	Max Accuracy
1f1s	41.00%	2.60%	93.10%
2f1s	41.70%	2.50%	85.90%
2f2s	50.08%	36.00%	82.00%
3f1s	52.65%	2.60%	79.40%
3f2s	56.23%	38.90%	84.10%
3f3s	52.20%	40.10%	57.70%
4f1s	48.20%	3.90%	85.90%
4f2s	57.73%	26.80%	84.10%
4f3s	55.90%	44.70%	66.80%
4f4s	38.38%	23.80%	61.00%
5f1s	31.75%	2.70%	79.40%
5f2s	54.18%	26.90%	68.80%
5f3s	61.53%	48.40%	71.80%
5f4s	45.30%	23.80%	67.80%
5f5s	43.75%	36.10%	53.00%
	AVG	23.99%	74.72%

Considering 50% of the services with endpoints (as method invocations), we allowed only this information from the cart, catalogue, and payment services. The results are shown in Table 5.11.

Considering only 25% of the services with endpoints (as method invocations), we allowed only this information from the cart and catalogue services. The results are shown in Table 5.12.

Considering 0% (none) of the services with endpoints (as method invocations), we filtered this information from all the services. The results are shown in Table 5.13.

The results for the service entity targeting are shown in Table 5.14

Analysis In Table 5.10, by removing 25% of the method invocations, we see a significant drop in the accuracy, below 50% in average accuracy. The poorly performant scenarios are also many more, with five scenarios below 5% accuracy. Only 11 scenarios out of 60 performed with an accuracy above 70%. In the default scenario, that number is 33.

In Table 5.11, with half of the services providing method invocations, the average accuracy floats around 30%, a steady drop from the previous scenario.

Table 5.11: Evaluation results for the scenario suite using 50% of services with endpoints as method invocations

Category	Accuracy	Min Accuracy	Max Accuracy
1f	37.33%	3.30%	89.50%
2f	34.65%	3.20%	78.20%
3f	29.74%	3.40%	76.80%
4f	33.93%	3.60%	78.20%
5f	30.08%	3.60%	68.20%
1s	27.95%	3.20%	89.50%
2s	36.63%	3.50%	76.80%
3s	35.29%	4.20%	68.20%
4s	30.29%	17.30%	51.60%
5s	29.28%	11.90%	35.50%

Global average
32.13%

Category	Accuracy	Min Accuracy	Max Accuracy
1f1s	37.33%	3.30%	89.50%
2f1s	37.58%	3.20%	78.20%
2f2s	31.73%	3.50%	74.80%
3f1s	18.80%	3.40%	63.30%
3f2s	39.10%	3.60%	76.80%
3f3s	31.33%	4.20%	54.30%
4f1s	33.25%	4.70%	78.20%
4f2s	39.20%	3.60%	76.20%
4f3s	36.40%	4.80%	63.70%
4f4s	26.85%	17.30%	42.40%
5f1s	12.78%	3.60%	37.40%
5f2s	36.48%	3.70%	63.40%
5f3s	38.15%	4.80%	68.20%
5f4s	33.73%	17.40%	51.60%
5f5s	29.28%	11.90%	35.50%
	AVG	6.20%	63.57%

Table 5.12: Evaluation results for the scenario suite using 25% of services with endpoints as method invocations

Category	Accuracy	Min Accuracy	Max Accuracy
1f	35.68%	3.50%	88.00%
2f	31.59%	3.30%	75.60%
3f	26.04%	3.50%	73.70%
4f	30.81%	3.80%	75.60%
5f	27.04%	3.80%	66.70%
1s	26.88%	3.30%	88.00%
2s	32.16%	3.70%	73.70%
3s	30.75%	4.00%	66.70%
4s	27.46%	15.00%	47.70%
5s	25.23%	5.40%	32.40%

Global average
29.03%

Category	Accuracy	Min Accuracy	Max Accuracy
1f1s	35.68%	3.50%	88.00%
2f1s	35.95%	3.30%	75.60%
2f2s	27.23%	3.70%	71.70%
3f1s	17.83%	3.50%	58.60%
3f2s	33.43%	3.80%	73.70%
3f3s	26.88%	4.50%	50.00%
4f1s	32.23%	5.20%	75.60%
4f2s	35.45%	3.80%	72.90%
4f3s	31.78%	4.00%	62.20%
4f4s	23.78%	15.10%	38.10%
5f1s	12.70%	3.80%	35.80%
5f2s	32.53%	3.90%	61.10%
5f3s	33.60%	4.70%	66.70%
5f4s	31.15%	15.00%	47.70%
5f5s	25.23%	5.40%	32.40%
	AVG	5.55%	60.67%

With just a quarter of the services providing method invocations, in Table 5.12, the drop in the accuracy is much smaller than the previous scenario with 50% of services with method invocations. It goes below 30% in average accuracy, but close by, it is not a significant change from what we

Table 5.13: Evaluation results for the scenario suite using 0% of services with endpoints as method invocations

Category	Accuracy	Min Accuracy	Max Accuracy
1f	8.60%	3.10%	12.50%
2f	9.26%	3.60%	14.40%
3f	8.82%	4.30%	14.00%
4f	11.76%	4.80%	20.80%
5f	13.86%	5.20%	22.10%
1s	9.94%	3.10%	20.80%
2s	10.28%	4.40%	17.40%
3s	12.04%	6.10%	21.70%
4s	13.28%	7.70%	22.10%
5s	16.43%	13.90%	19.10%

Global average
11.33%

Category	Accuracy	Min Accuracy	Max Accuracy
1f1s	8.60%	3.10%	12.50%
2f1s	9.90%	3.60%	14.40%
2f2s	8.63%	4.40%	12.10%
3f1s	6.70%	4.30%	9.00%
3f2s	9.58%	5.20%	13.80%
3f3s	10.18%	8.40%	14.00%
4f1s	13.18%	8.50%	20.80%
4f2s	11.15%	4.80%	16.40%
4f3s	11.38%	6.10%	17.50%
4f4s	11.33%	7.70%	13.00%
5f1s	11.33%	5.20%	20.80%
5f2s	11.75%	5.30%	17.40%
5f3s	14.58%	8.40%	21.70%
5f4s	15.23%	9.00%	22.10%
5f5s	16.43%	13.90%	19.10%
	AVG	6.53%	16.31%

Table 5.14: Evaluation results for the scenario suite targeting service entities with no method invocations

Category	Accuracy	Min Accuracy	Max Accuracy
1f	68.75%	25.00%	100.00%
2f	65.74%	25.00%	100.00%
3f	58.43%	25.00%	90.30%
4f	65.94%	33.00%	100.00%
5f	64.20%	25.00%	100.00%
1s	63.75%	25.00%	100.00%
2s	60.73%	33.00%	90.30%
3s	62.57%	42.10%	88.90%
4s	67.76%	43.20%	95.00%
5s	75.35%	63.50%	86.00%

Global average
64.02%

Category	Accuracy	Min Accuracy	Max Accuracy
1f1s	68.75%	25.00%	100.00%
2f1s	68.75%	25.00%	100.00%
2f2s	62.73%	33.00%	85.70%
3f1s	46.88%	25.00%	62.50%
3f2s	63.43%	39.30%	90.30%
3f3s	64.98%	51.00%	88.90%
4f1s	78.13%	50.00%	100.00%
4f2s	59.15%	33.00%	79.50%
4f3s	60.18%	42.10%	88.90%
4f4s	66.30%	49.90%	80.80%
5f1s	56.25%	25.00%	100.00%
5f2s	57.60%	39.30%	79.50%
5f3s	62.55%	51.00%	88.90%
5f4s	69.23%	43.20%	95.00%
5f5s	75.35%	63.50%	86.00%
	AVG	39.69%	88.40%

consider already poor results.

At last, with no services providing method invocations, in Table 5.13, every faulty entity will not be identified since no method invocations are present in the extracted information. The ob-

tained results are the weighed parent accuracy of entities having a low impact. The average accuracy is at its lowest, at 11%. The highest accuracy level registered is only 22%.

As this last case was expected to be unfruitful, we also adapted our test suite for identifying the faulty service entities instead of the methods since there would be no information pointing to them.

In this scenario, in Table 5.14, the results are not far away from the default with all the service method invocations without service weight attenuation. The significant difference is that the values observed seem more disparate, oscillating from 25% accuracy to 100%. This is only because the number of entities in the ranking is low since only the service entities are reported.

5.2.5 Execution Times

Results The execution times results, while important, they play a secondary role in our evaluation step, as our focus at this point is on the accuracy of the tool, and the machine used for testing is not appropriate for production-level microservices, as described in Subsection 5.1.5. Since the purpose of this step is to benchmark the execution times of the tool, we computed the execution times in the tool's configuration with the best accuracy results (service weight attenuation with average). They are shown in Table 5.15. In Appendix B (p. 78) we present the execution times of each individual scenario for reference.

Table 5.15: Execution times (in seconds) in the most accurate scenario suite (service weight attenuation with average).

Category	Avg	Min	Max
1f	7.0967	5.2879	8.7980
2f	6.8629	4.7905	7.8597
3f	7.3498	4.4619	8.9848
4f	6.6452	4.7722	8.0032
5f	7.2676	4.8807	12.1451
1s	6.5800	4.4619	8.7980
2s	7.3374	5.9315	9.3037
3s	7.6067	5.0214	12.1451
4s	6.8745	6.2888	7.2863
5s	6.9717	6.1931	8.4100

Global	Avg	Min	Q1 25%
	7.0527	4.4619	6.50752
	Med	Q3 75%	Max
	6.9728	7.52992	12.1451

Category	Avg	Min	Max
1f1s	7.0967	5.2879	8.7980
2f1s	6.6811	4.7905	7.7746
2f2s	7.0446	6.5107	7.8597
3f1s	6.8911	4.4619	8.5373
3f2s	7.7889	6.7925	8.9644
3f3s	7.3693	6.2890	8.9848
4f1s	5.7430	4.7722	6.8019
4f2s	7.1671	6.0035	8.0032
4f3s	7.0307	6.6583	7.3888
4f4s	6.6399	6.2888	7.0032
5f1s	6.4880	4.8807	7.4484
5f2s	7.3491	5.9315	9.3037
5f3s	8.4202	5.0214	12.1451
5f4s	7.1091	6.9070	7.2863
5f5s	6.9717	6.1931	8.4100

Analysis Regarding the execution time of our tool, in Table 5.15, we observe that it revolves around seven seconds in our test application. It is noteworthy that it encompasses the whole reading of log data (already processed), entity extraction, analysis, ranking, and completing all the outputs. Since it is replaceable and meant to be generic, we do not track the execution time of the log processor, Logstash. Ignoring the tool's accuracy, it seems that the execution time is appropriate given the time spent debugging manually, as discussed in Section 1.2, and the amount of log data and size of the codebase. Possibly the time spent in I/O operations (reading/writing files) has a more significant impact than the remaining of the tool's operation, but that would require further investigation to corroborate it.

5.3 Comparative Analysis

Besides analyzing the results individually, it is relevant to analyze other approaches' results and compare them with ours, where deemed appropriate. To the best of our knowledge, there are no other approaches to debugging microservices by applying the SFL technique via log analysis. Therefore, we analyze different approaches in either the context of debugging microservices or using the SFL technique. All the techniques are already explored as part of the Related Work, in Chapter 3 (p. 17).

Analyzing approaches dedicated to debugging microservices, we consider T-Rank [110], which also leverages the SFL technique. However, this technique focuses on root cause localization and analyzing performance metrics. In their evaluation, the authors use the EXAM score (among other metrics) to assess the approach's effectiveness. This metric represents the percentage of components observed before the faulty component is analyzed. This is complementary to the accuracy we evaluate, given the distance of the faulty component to the top of the ranking. Given that their score is 0.013, the respective accuracy is 98.3%. Strictly comparing numbers, this represents that it surpasses our approach by 30%. Notwithstanding, it is relevant to consider that tracing data is imperative in this approach, while our approach functions without it, and we have evaluated a scenario without tracing data. Furthermore, the granularity of this approach is limited to the container, which is relevant for localizing faults of other nature, such as configuration and instance. Moreover, our approach is not limited to a specific level of granularity, being able to pinpoint faults at the line of code level given that the logs contain that information.

On the subject of delta debugging, there is the approach presented in [112] that proposes a technique based on delta debugging, a type of state-based fault localization. Here different dimensions that can interfere with the proper execution of the microservice system are considered, and the goal is to determine which circumstances cause the system to malfunction. The approach considers various dimensions besides the code, such as configuration and sequence. Their results show that the tool is successful in most cases in detecting the delta responsible. There are no explicit numerical values to assess the approach's effectiveness, but comparing the granularity of both approaches is more relevant. While this approach can analyze different categories of faults in the system, it always requires further inspection to find the root cause while we can

pinpoint the fault in the code. In addition, the setup required to perform debugging with this tool is more complicated as it needs to deal with multiple categories of deltas. At the same time, our configuration is restricted to the source and processing of the logs.

Regarding anomaly detection in microservices, we analyzed the proposed algorithm in [43], which is based on machine learning. The algorithm analyzes invocation chains to mine causality and locates the root cause of anomalies. It requires information necessary to establish invocations chains of services and other KPIs, such as CPU, memory, and cache usage. Besides evaluating precision and recall, the authors also compute a score for the algorithm. This is based on the total root cause indicators score divided by the number of faults. The achieved score is 0.8304, and the precision is 90%. This approach is more appropriate than ours for detecting performance issues, whereas ours is more focused on detecting faults in the code.

In [5], the authors propose a root cause localization technique in (cloud) microservice applications. They focus on microservices' error rates (golden signals) to localize operational faults (at the microservice level), using runtime logs to infer causality. Their approach to localizing operation faults is two-fold, localizing the faulty microservice and finding the error message of that service responsible that flags the fault. Based on the causal relationships of golden signal errors and microservice errors, they produce a ranked list of possible faulty components, an output with a similar format to ours. It achieves an F-score of 88%. The focus is operation faults instead of faults in the code, so naturally, the maximum granularity level achievable is that of the microservice.

Turning the focus to SFL approaches, we analyze the approach presented in [3]. The proposed tools focus on localizing multiple faults by applying the SFL combined with model-based diagnosis. The performed evaluation considers single faults, but our focus is on multiple faults, the scenario parallel to our evaluation. In the multiple scenarios evaluated, there was a range of multiple faults used, from one to five. The metric used to assess the effectiveness of the approach was wasted effort, which is the percentage of components observed until the faulty component is reached. Another way to see it is as the complement the accuracy. Computing the average of wasted effort in all scenarios, we arrive at 8.4%. The complementary that we consider analog to accuracy is then 91.6%. Perhaps we cannot achieve the same level of accuracy in the context of microservices as the level achieved in traditional systems. Nevertheless, this result favors SFL as a valid technique to be adapted to the context of microservices.

Another SFL approach is [82], which combines Qualitative Reasoning with the SFL technique. Q-SFL *"leverages the concept of qualitative reasoning to augment the information made available to SFL techniques by qualitatively partitioning the values of data units from the system and treating each qualitative state as a new SFL component to be used when diagnosing"*. In the empiric evaluation performed, based on 167 components observed, the tool produced an average of 37.56 components evaluated before the faulty component was observed. From this, we can compute the percentage of wasted effort, $37.56/167 = 0.225$, or 22.5%. The accuracy of the tool is then 77.5%. This result is closer to ours and again reinforces the potential for improvement of our tool.

5.4 Threats to Validity

Given our evaluation process, it is also important to consider existent threats to the validity of our study. We consider two categories to assess possible threats based on [103], *internal* and *external*.

5.4.1 Internal Validity

Threats to internal validity affect the conclusions about the established causal relationship between execution and the outcome.

- **Instrumentation**: Considering that we manually and randomly inject faults, some faults may be more or less relevant in the sense that they have little or no occurrence in real-world scenarios. However, we strived to insert diverse types of faults to cover several scenarios.
- **Selection**: In the evaluation process, we considered only a single application to evaluate. The motivation was simply that it provided minimal log information and was easily manipulable in a local machine. The choice was free of any bias that could positively influence the results. However, since there is one sample, it is impossible to guarantee that the application logs are favorable, and the general case would be less so.

  Even more so, we considered an application that provides incomplete information: no tracing data and concrete method invocation. The hypothesis is that given that the missing information is present, the results would improve. However, we cannot validate this hypothesis without concrete scenarios to prove our assumption.

5.4.2 External Validity

Threats to external validity affect the ability to generalize the experiment's results to industrial practice.

- **Interaction of selection and treatment**: Considering production-level microservice systems, the size and complexity can be superior to the application used in the evaluation process. Despite the business logic being of a real-world application, the nature of microservices is to deal with a high cardinality of operations. The application selection may not be representative of the majority of real-world microservices.

5.5 Summary

After implementing the prototype, we had to test our tool and observe its performance. We chose to apply a small-grade microservice application based on an online shopping site that could be set up locally and have its source code altered for fault injection. This application did not consider request IDs in all its services, so we had to adapt our implementation and view all the available information.

We run the tool and evaluate the entities injected with a fault, their position in the ranking, and compute its detection accuracy. Each scenario can have one or more faulty entities to be discovered, and the total accuracy for each scenario is the average accuracy of its fault entities.

With a complete log extraction, we obtained a global accuracy of 63%. It is an acceptable value considering the non-optimal conditions it has with this demo application.

We detected an overweight of the service entities when compared to its children and applied a strategy to alleviate it. The winner strategy, the average of the service children, was able to bring the global accuracy up to 68%.

Regarding rank ties, we considered multiple approaches to see if the default behavior varies too much from them. Viewing the results of the best-case, worst-case, and average-case scenarios, we establish that the default behavior seems to fall within the boundaries they set, which is the desirable outcome.

Finally, our last evaluation of interest was considering varying percentages of services with endpoints as method invocations and analyzing the impact on the tool's performance. For that reason, we thought of scenarios with 100% (default), 75%, 50%, 25%, and 0% of services with method invocations. We also analyzed the tool's performance targeting the service entities when 0% of the services had method invocations extracted from the logs. We soon understood that the impact of having method invocations in the logs is vital to the healthy functioning and accuracy of the tool. By having 75% of the services with method invocations, the global accuracy drops above 50%, and with 50% of the services, it goes close to 30%. By removing all of the method invocations, we hit a low 11% of accuracy, which is the representation of the fault entity parent (service entity) weighed accuracy. By running the same scenario targeting the service entities instead, we return to the baseline of 64% of global accuracy.

The application's execution time is around seven seconds in our test scenarios. Considering the volume of log data, the codebase's size, and comparing the execution time with the time frequently spent on manual debugging, the current execution time represents an improvement.

When comparing to other approaches, we observed that, while not outperforming other techniques, their focus is not on the code, where ours is. Traditional SFL approaches have great results, leading us to believe that there is potential for improvement, given that we are exploring the usage of the technique in the context of microservices.

The validity of our evaluation process has possible internal and external threats. At the internal level, we consider that having this evaluation process applied to a single application does not guarantee that the same results extend to other applications in the same conditions. Furthermore, given the size and complexity of the application, it might not be representative of all production-level microservices.

This complete set of results allows us to draw conclusions and point to strong and weak points, which we discuss as part of the future work for this tool.

Chapter 6

Conclusions

In this final chapter, we reflect on the considerations we began the Thesis based on, the challenges faced during development, the obtained results versus the expected results, and the contributions of our work. Additionally, it is essential to be critical of our work and analyze potential threats to the work performed. Finally, based on all the conclusions, it is also necessary to delineate relevant topics to focus on as future work and maintain continuity to the project.

6.1 Final Considerations

This work focus on a particular fault localization technique applied to microservices. As part of its nature, it can be built with various options, configurations, and paradigms. We realize it is challenging to create a generic tool to be viable with all microservices while requiring the lowest information possible.

During the implementation, we faced that issue head-on, and, when possible, we strived to make it generic. Mainly when dealing with logs, we rapidly faced the reality of processing non-uniform logs and logs without complete information. The issue of logs not containing request IDs is even more specific and usually used in tracing techniques. It is essential to have this information present to be more accurate in assessing the components in the code as faulty.

However, if a system is currently legacy and, for some other reasons, it cannot be altered, one must be prepared to deal in cases where the input data is not optimal. We prepared for that and evaluated our tool in such circumstances. We tested our tool in a small-scale, apt microservice-based application without complete information. We obtained an average of 68% accuracy in overall the scenarios discussed in Chapter 5 (p. 40).

Considering the results and current accuracy in pinpointing a faulty entity, we conclude that, considering the circumstances in which the application was evaluated, this result is encouraging for the tool, and the potential for improvement is correctly assumed. Our comparative analysis also indicates that further work in our approach is reasonable to make it outstanding.

Notwithstanding the already tested and validated technique that is SFL, our approach is a novel approach to adapt this technique to the context of microservices and requires many modifications throughout the project's timeline.

The main contributions our work provides are:

- A generic tool that can perform debugging of microservices based on log analysis and the SFL technique
- A novel approach to SFL adapted to the context of microservices

6.2 Further Work

The prototype we developed has some limitations. First, as we develop a tool for evolving software, we must consider certain aspects of the implementation that could become problematic. In more detail, the communication/transaction section of the tool requires further evaluation to ensure the most efficient method to guarantee the complete delivery of information (log data). Moreover, our primary focus during the development was the accuracy of the tool's detection. We must further evaluate the tool's performance in other aspects (such as resource usage) in microservices and optimize its implementation, if necessary. However, further investigation is required to assess optimization points, mainly in I/O operations.

An interesting topic to work on is the classification of executions/requests, which separate logs. We consider our tests a manual classification given that we executed the system with and without injected faults. However, this is not the typical case of real-world microservices, and the classification of a request, and its respective logs, is not a trivial task. For future work, we consider it would be relevant to find a generic method to label executions in an automated but reliable way.

Regarding the validity threats as analyzed in Chapter 5 (p. 40), we believe that further evaluation and analysis are required to assure the tool's usability. In the first part, we consider it relevant to gather more applications suitable for the scenarios presented in the evaluation process. By having a non-homogenous group of tested applications, we can further advocate for the generalization of the tool. In the second part, some of those applications should push the tool's limits with their size and complexity. It would also be interesting to assess what is the limit, if it exists, of the number of services in the application before the tool is unable to provide meaningful outputs. An example of such applications and interesting to test in the future is TrainTicket [23], which in its total capacity would require more computational power to execute.

Appendix A

Evaluation Accuracy for Individual Scenarios Results

In this appendix, the accuracies for individual scenarios are presented. In addition to the accuracy, the ranking of the expected faulty entities and its SFL value in each scenario are presented. In scenarios with multiple faulty entities their ranking and SFL value are separated with a bar, "/", and they have they are presented in the same order in the ranking and SFL values column. For the ranking, they can have a number from one to the number of entities present in the SFL ranking, or denoted with "-" if they are not present in the ranking. The SFL value will be 0 if the entity is the last of the rank or if its not present in it.

A.1 Scenario Suite

Table A.1: Individual accuracies in the first execution of the scenario suite. The rank has 37 entities.

Category	Combination	Accuracy	Entities Ranking	Entities SFL Value	Category	Combination	Accuracy	Entities Ranking	Entities SFL Value
1f1s	cart	45.60%	22	0.026	2f1s	cart	65.10%	8/22	0.058/0.025
	catalogue	94.60%	4	0.153		catalogue	75.10%	4/19	0.154/0.042
	user	77.80%	10	0.069		user	40.00%	10/-	0.06/0
	web	64.90%	15	0.037		web	33.90%	15/-	0.035/0
2f2s	cart-catalogue	72.20%	4/21	0.159/0.039	3f1s	cart	66.50%	7/22	0.059/0.025
	payment-ratings	2.44%	-/-	0/0		ratings	2.60%	-/-	0/0
	shipping-user	40.00%	10/-	0.058/0		shipping	2.40%	-/-/-	0/0/0
	web-cart	61.30%	12/21	0.046/0.38		user	27.50%	10/-/-	0.059/0/0
3f2s	cart-catalogue	78.20%	4/7/21	0.16/0.089/0.039	3f3s	cart-catalogue-payment	48.40%	4/21/-	0.158/0.038/0
	payment-ratings	2.40%	-/-	0/0		ratings-shipping-user	27.70%	10/-/-	0.064/0/0
	shipping-user	27.60%	10/-/-	0.06/0/0		web-cart-catalogue	73.60%	4/12/21	0.153/0.045/0.037
	web-cart	74.50%	3/12/21	0.149/0.04/0.037		payment-shipping-ratings	2.50%	-/-/-	0/0/0
4f1s	cart	45.20%	7/22/-	0.07/0.03/0	4f2s	cart-catalogue	74.00%	4/7/18/21	0.156/0.088/0.043/0.039
	catalogue	76.50%	4/18	0.114/0.031		payment-ratings	2.50%	-/-/-/-	0/0/0/0
	ratings	2.50%	-/-	0/0		shipping-user	21.30%	10/-/-/-	0.055/0/0/0
	web	18.50%	15/-/-/-	0.036/0/0/0		web-cart	80.40%	3/6/12/20	0.161/0.124/0.047/0.039
4f3s	cart-catalogue-payment	59.00%	4/7/21/-	0.159/0.089/0.039/0	4f4s	cart-catalogue-payment-ratings	37.00%	4/21/-/-	0.157/0.038/0/0
	ratings-shipping-user	21.50%	10/-/-/-	0.069/0/0/0		shipping-user-web-cart	52.60%	8/12/21/-	0.077/0.047/0.039/0
	web-cart-catalogue	80.40%	4/12/21	0.152/0.045/0.037		payment-web-catalogue-user	65.00%	4/8/12/-	0.154/0.078/0.046/0
	payment-shipping-user	27.60%	10/-/-	0.064/0/0		cart-ratings-shipping-payment	13.20%	21/-/-/-	0.037/0/0/0
5f1s	cart	40.40%	7/21/31/-	0.075/0.032/0/0	5f2s	cart-catalogue	63.40%	4/7/19/21/31	0.1/0.088/0.043/0.038/0
	shipping	2.50%	-/-/-/-	0/0/0/0		payment-ratings	2.50%	-/-	0/0
	user	21.20%	10/-/-/-	0.058/0/0/0		shipping-user	17.70%	10/-/-/-/-	0.065/0/0/0/0
	web	18.50%	15/-/-/-	0.037/0/0/0		web-cart	79.70%	3/6/12/21	0.148/0.114/0.044/0.036
5f3s	cart-catalogue-payment	58.30%	4/7/19/22/-	0.12/0.066/0.033/0.029/0	5f4s	cart-catalogue-payment-ratings	48.90%	4/6/20/-/-	0.159/0.089/0.039/0/0
	ratings-shipping-user	17.80%	10/-/-/-/-	0.062/0/0/0/0		shipping-user-web-cart	42.30%	8/12/21/-/-	0.075/0.046/0.038/0/0
	web-cart-catalogue	83.40%	4/7/12/21	0.155/0.087/0.046/0.038		payment-shipping-user-cart	33.70%	8/21/-/-	0.077/0.037/0/0
	payment-shipping-ratings	2.50%	-/-/-/-	0/0/0/0		catalogue-ratings-web-cart	56.00%	4/12/20/21/-	0.153/0.045/0.042/0.037/0
5f5s	cart-catalogue-payment-ratings-shipping	29.90%	4/21/-/-/-	0.153/0.037/0/0/0					
	user-web-cart-catalogue-payment	62.20%	4/8/12/21/-	0.154/0.08/0.046/0.038/0					
	ratings-shipping-user-web-payment	30.30%	11/15/-/-/-	0.065/0.037/0/0/0					
	cart-catalogue-web-ratings-payment	45.00%	4/12/21/-/-	0.153/0.045/0.037/0/0					

Table A.2: Individual accuracies in the second execution of the scenario suite. The rank has 49 entities.

Category	Combination	Entities Ranking	Accuracy	Entities SFL Value	Category	Combination	Entities Ranking	Accuracy	Entities SFL Value
1f1s	cart	23	56.90%	0.05	2f1s	cart	7/23	73.80%	0.117/0.049
	catalogue	4	95.90%	0.257		catalogue	4/17	83.40%	0.258/0.067
	user	11	81.30%	0.106		user	11/-	41.50%	0.089/0
	web	17	69.40%	0.055		web	17/-	35.80%	0.054/0
2f2s	cart-catalogue	7/23	73.80%	0.117/0.049	3f1s	cart	6/23	74.90%	0.118/0.049
	payment-ratings	4/17	83.40%	0.258/0.067		ratings	13/38	52.20%	0.071/0.035
	shipping-user	11/-	41.50%	0.089/0		shipping	8	87.80%	0.139
	web-cart	17/-	35.80%	0.054/0		user	11/-/-	28.30%	0.088/0/0
3f2s	cart-catalogue	4/7/22	83.00%	0.262/0.145/0.061	3f3s	cart-catalogue-payment	4/22/43	57.30%	0.26/0.061/0
	payment-ratings	33/38	31.70%	0.048/0.036		ratings-shipping-user	8/11/38	66.00%	0.14/0.095/0.034
	shipping-user	9/11	84.20%	0.139/0.089		web-cart-catalogue	4/13/22	78.90%	0.256/0.073/0.059
	web-cart	3/13/22	79.60%	0.246/0.07/0.057		payment-shipping-ratings	9/33/38	50.10%	0.14/0.048/0.035
4f1s	cart	6/23/-	50.50%	0.136/0.057/0	4f2s	cart-catalogue	4/7/16/22	81.20%	0.258/0.145/0.068/0.061
	catalogue	4/16	84.40%	0.224/0.059		payment-ratings	13/34/37	47.40%	0.076/0.05/0.038
	ratings	14/39	50.10%	0.07/0.033		shipping-user	9/11/-	56.80%	0.135/0.083/0
	web	17/-/-/-	19.00%	0.054/0/0/0		web-cart	3/6/13/21	84.40%	0.256/0.196/0.072/0.06
4f3s	cart-catalogue-payment	4/7/22/43	66.30%	0.261/0.144/0.061/0	4f4s	cart-catalogue-payment-ratings	4/22/23/43	58.00%	0.259/0.06/0.051/0
	ratings-shipping-user	10/11/13/38	68.90%	0.135/0.104/0.071/0.034		shipping-user-web-cart	8/13/22/45	60.00%	0.121/0.072/0.059/0
	web-cart-catalogue	4/13/22	84.40%	0.255/0.072/0.059		payment-web-catalogue-user	4/8/13/43	70.60%	0.257/0.128/0.073/0
	payment-shipping-user	8/11/33	69.50%	0.141/0.097/0.049		cart-ratings-shipping-payment	22/23/43/45	36.20%	0.058/0.047/0/0
5f1s	cart	7/22/49/-	39.00%	0.141/0.059/0/0	5f2s	cart-catalogue	4/7/17/22/49	65.40%	0.171/0.149/0.07/0.063/0
	shipping	8	87.80%	0.143		payment-ratings	13/33/37	48.10%	0.076/0.051/0.039
	user	11/-/-/-	21.70%	0.089/0/0/0		shipping-user	9/11/-	56.80%	0.136/0.097/0
	web	17/-/-/-	19.00%	0.055/0/0/0		web-cart	3/6/13/22	83.90%	0.245/0.188/0.07/0.057
5f3s	cart-catalogue-payment	4/7/17/22/43	67.80%	0.231/0.125/0.06/0.053/0	5f4s	cart-catalogue-payment-ratings	4/6/21/23/43	66.10%	0.262/0.145/0.062/0.049/0
	ratings-shipping-user	9/11/13/37	69.90%	0.14/0.091/0.074/0.038		shipping-user-web-cart	8/13/22/45	60.00%	0.12/0.072/0.059/0
	web-cart-catalogue	4/7/13/22	86.90%	0.258/0.142/0.073/0.06		payment-shipping-user-cart	8/22/43/45	43.80%	0.124/0.058/0/0
	payment-shipping-ratings	8/33/38	50.80%	0.139/0.048/0.037		catalogue-ratings-web-cart	4/13/18/22/23	74.10%	0.256/0.073/0.067/0.06/0.048
5f5s	cart-catalogue-payment-ratings-shipping	4/22/23/43/45	48.80%	0.256/0.059/0.051/0/0					
	user-web-cart-catalogue-payment	4/8/13/22/43	69.10%	0.257/0.13/0.073/0.06/0					
	ratings-shipping-user-web-payment	8/11/18/34/38	62.10%	0.14/0.098/0.055/0.048/0.038					
	cart-catalogue-web-ratings-payment	4/13/22/23/43	62.80%	0.255/0.073/0.06/0.049/0					

A.2 Service Entity Weight Attenuation

Table A.3: Individual accuracies in the scenario suite using the service attenuation with division. The rank has 49 entities.

Category	Combination	Entities Ranking	Accuracy	Entities SFL Value	Category	Combination	Entities Ranking	Accuracy	Entities SFL Value
1f1s	cart	23	56.90%	0.05	2f1s	cart	7/23	73.80%	0.117/0.049
	catalogue	4	95.90%	0.257		catalogue	4/17	83.40%	0.258/0.067
	user	11	81.30%	0.106		user	11/-	41.50%	0.089/0
	web	17	69.40%	0.055		web	17/-	35.80%	0.054/0
2f2s	cart-catalogue	4/22	78.10%	0.262/0.062	3f1s	cart	6/23	74.90%	0.118/0.049
	payment-ratings	35/37	30.70%	0.048/0.04		ratings	13/38	52.20%	0.071/0.035
	shipping-user	9/11	84.20%	0.14/0.086		shipping	8	87.80%	0.139
	web-cart	13/22	68.90%	0.071/0.059		user	11/-/-	28.30%	0.088/0/0
3f2s	cart-catalogue	4/7/22	83.00%	0.262/0.145/0.061	3f3s	cart-catalogue-payment	4/22/47	54.50%	0.26/0.061/0
	payment-ratings	34/38	30.70%	0.048/0.036		ratings-shipping-user	8/11/38	66.00%	0.14/0.095/0.034
	shipping-user	9/11	84.20%	0.139/0.089		web-cart-catalogue	4/13/22	78.90%	0.256/0.073/0.059
	web-cart	3/13/22	79.60%	0.246/0.07/0.057		payment-shipping-ratings	9/34/38	49.40%	0.14/0.048/0.035
4f1s	cart	6/23/-	50.50%	0.136/0.057/0	4f2s	cart-catalogue	4/7/16/22	81.20%	0.258/0.145/0.068/0.061
	catalogue	4/16	84.40%	0.224/0.059		payment-ratings	13/34/37	47.40%	0.076/0.05/0.038
	ratings	14/38	51.20%	0.07/0.033		shipping-user	9/11/-	56.80%	0.135/0.083/0
	web	17/-/-/-	19.00%	0.054/0/0/0		web-cart	3/6/13/21	84.40%	0.256/0.196/0.072/0.06
4f3s	cart-catalogue-payment	4/7/22/47	64.10%	0.261/0.144/0.061/0	4f4s	cart-catalogue-payment-ratings	4/22/23/47	55.80%	0.259/0.06/0.051/0
	ratings-shipping-user	10/11/13/38	68.90%	0.135/0.104/0.071/0.034		shipping-user-web-cart	8/13/22/46	59.50%	0.121/0.072/0.059/0
	web-cart-catalogue	4/13/22	84.40%	0.255/0.072/0.059		payment-web-catalogue-user	4/8/13/47	68.40%	0.257/0.128/0.073/0
	payment-shipping-user	8/11/34	68.80%	0.141/0.097/0.049		cart-ratings-shipping-payment	22/23/46/47	33.50%	0.058/0.047/0/0
5f1s	cart	7/22/44/-	41.70%	0.141/0.059/0/0	5f2s	cart-catalogue	4/7/17/22/44	67.60%	0.171/0.149/0.07/0.063/0
	shipping	8	87.80%	0.143		payment-ratings	13/34/37	47.40%	0.076/0.051/0.039
	user	11/-/-/-	21.70%	0.089/0/0/0		shipping-user	9/11/-	56.80%	0.136/0.097/0
	web	17/-/-/-	19.00%	0.055/0/0/0		web-cart	3/6/13/22	83.90%	0.245/0.188/0.07/0.057
5f3s	cart-catalogue-payment	4/6/21/23/47	66.10%	0.231/0.125/0.06/0.053/0	5f4s	cart-catalogue-payment-ratings	4/6/21/23/43	64.40%	0.262/0.145/0.062/0.049/0
	ratings-shipping-user	9/11/13/37	69.90%	0.14/0.091/0.074/0.038		shipping-user-web-cart	8/13/22/46	59.50%	0.12/0.072/0.059/0
	web-cart-catalogue	4/7/13/22	86.90%	0.258/0.142/0.073/0.06		payment-shipping-user-cart	8/22/46/47	41.10%	0.124/0.058/0/0
	payment-shipping-ratings	8/34/38	50.10%	0.139/0.048/0.037		catalogue-ratings-web-cart	4/13/18/22/23	74.10%	0.256/0.073/0.067/0.06/0.048
5f5s	cart-catalogue-payment-ratings-shipping	4/22/23/46/47	46.60%	0.256/0.059/0.051/0/0					
	user-web-cart-catalogue-payment	4/8/13/22/47	67.40%	0.257/0.13/0.073/0.06/0					
	ratings-shipping-user-web-payment	8/11/18/34/38	62.10%	0.14/0.098/0.055/0.048/0.038					
	cart-catalogue-web-ratings-payment	4/13/22/23/47	61.00%	0.255/0.073/0.06/0.049/0					

Table A.4: Individual accuracies in the scenario suite using the service attenuation with average. The rank has 49 entities.

Category	Combination	Entities Ranking	Accuracy	Entities SFL Value	Category	Combination	Entities Ranking	Accuracy	Entities SFL Value
1f1s	cart	22	58.40%	0.05	2f1s	cart	3/22	78.40%	0.117/0.049
	catalogue	1	100.00%	0.257		catalogue	1/12	90.60%	0.258/0.067
	user	4	95.50%	0.106		user	5/-	47.50%	0.089/0
	web	13	77.00%	0.055		web	13/-	39.30%	0.054/0
2f2s	cart-catalogue	1/20	81.90%	0.262/0.062	3f1s	cart	2/22	79.40%	0.118/0.049
	payment-ratings	33/36	32.70%	0.048/0.04		ratings	7/38	57.00%	0.071/0.035
	shipping-user	3/5	96.20%	0.14/0.086		shipping	3	97.40%	0.139
	web-cart	10/21	72.50%	0.071/0.059		user	5/-/-	32.20%	0.088/0/0
3f2s	cart-catalogue	1/2/20	88.40%	0.262/0.145/0.061	3f3s	cart-catalogue-payment	1/20/40	62.00%	0.26/0.061/0
	payment-ratings	30/38	33.70%	0.048/0.036		ratings-shipping-user	3/4/38	73.50%	0.14/0.095/0.034
	shipping-user	3/5	96.20%	0.139/0.089		web-cart-catalogue	1/9/20	83.90%	0.256/0.073/0.059
	web-cart	1/10/21	82.70%	0.246/0.07/0.057		payment-shipping-ratings	3/28/38	56.50%	0.14/0.048/0.035
4f1s	cart	2/22/-	53.40%	0.136/0.057/0	4f2s	cart-catalogue	1/2/11/20	87.90%	0.258/0.145/0.068/0.061
	catalogue	1/11	91.60%	0.224/0.059		payment-ratings	7/32/36	52.30%	0.076/0.05/0.038
	ratings	9/38	55.00%	0.07/0.033		shipping-user	3/5/-	64.60%	0.135/0.083/0
	web	13/-/-/-	20.40%	0.054/0/0/0		web-cart	1/2/10/19	88.90%	0.256/0.196/0.072/0.06
4f3s	cart-catalogue-payment	1/2/20/40	71.90%	0.261/0.144/0.061/0	4f4s	cart-catalogue-payment-ratings	1/20/23/40	61.30%	0.259/0.06/0.051/0
	ratings-shipping-user	3/4/8/38	77.80%	0.135/0.104/0.071/0.034		shipping-user-web-cart	4/10/21/42	65.30%	0.121/0.072/0.059/0
	web-cart-catalogue	1/9/20	88.80%	0.255/0.072/0.059		payment-web-catalogue-user	1/3/9/40	77.60%	0.257/0.128/0.073/0
	payment-shipping-user	3/4/29	79.90%	0.141/0.097/0.049		cart-ratings-shipping-payment	21/23/40/42	39.60%	0.058/0.047/0/0
5f1s	cart	2/20/44/-	44.60%	0.141/0.059/0/0	5f2s	cart-catalogue	1/2/12/20/44	72.80%	0.171/0.149/0.07/0.063/0
	shipping	3	97.40%	0.143		payment-ratings	7/28/36	55.10%	0.076/0.051/0.039
	user	4/-/-/-	25.20%	0.089/0/0/0		shipping-user	3/4/-	65.40%	0.136/0.097/0
	web	13/-/-/-	20.50%	0.055/0/0/0		web-cart	1/2/10/21	87.80%	0.245/0.188/0.07/0.057
5f3s	cart-catalogue-payment	1/2/12/21/40	74.10%	0.231/0.125/0.06/0.053/0	5f4s	cart-catalogue-payment-ratings	1/2/19/23/40	70.10%	0.262/0.145/0.062/0.049/0
	ratings-shipping-user	3/5/7/36	78.90%	0.14/0.091/0.074/0.038		shipping-user-web-cart	3/10/21/42	65.80%	0.12/0.072/0.059/0
	web-cart-catalogue	1/2/9/20	91.70%	0.258/0.142/0.073/0.06		payment-shipping-user-cart	3/21/40/42	49.90%	0.124/0.058/0/0
	payment-shipping-ratings	3/30/37	55.80%	0.139/0.048/0.037		catalogue-ratings-web-cart	1/9/14/20/23	78.70%	0.256/0.073/0.067/0.06/0.048
5f5s	cart-catalogue-payment-ratings-shipping	1/20/23/40/42	52.70%	0.256/0.059/0.051/0/0					
	user-web-cart-catalogue-payment	1/3/9/20/40	75.50%	0.257/0.13/0.073/0.06/0					
	ratings-shipping-user-web-payment	3/4/14/29/37	70.40%	0.14/0.098/0.055/0.048/0.038					
	cart-catalogue-web-ratings-payment	1/9/20/23/40	67.00%	0.255/0.073/0.06/0.049/0					

Table A.5: Individual accuracies in the scenario suite using the service attenuation with maximum. The rank has 49 entities.

Category	Combination	Entities Ranking	Accuracy	Entities SFL Value	Category	Combination	Entities Ranking	Accuracy	Entities SFL Value
1f1s	cart	23	56.90%	0.05	2f1s	cart	4/23	76.90%	0.117/0.049
	catalogue	1	100.00%	0.257		catalogue	1/18	84.40%	0.258/0.067
	user	7	89.50%	0.106		user	7/-	45.70%	0.089/0
	web	16	70.90%	0.055		web	16/-	36.30%	0.054/0
2f2s	cart-catalogue	1/22	80.20%	0.262/0.062	3f1s	cart	3/23	78.00%	0.118/0.049
	payment-ratings	34/38	29.90%	0.048/0.04		ratings	10/39	53.80%	0.071/0.035
	shipping-user	4/7	93.60%	0.14/0.086		shipping	4	95.80%	0.139
	web-cart	13/22	68.70%	0.071/0.059		user	7/-/-	31.10%	0.088/0/0
3f2s	cart-catalogue	1/3/22	87.20%	0.262/0.145/0.061	3f3s	cart-catalogue-payment	1/22/47	55.90%	0.26/0.061/0
	payment-ratings	32/39	30.90%	0.048/0.036		ratings-shipping-user	4/7/39	70.70%	0.14/0.095/0.034
	shipping-user	4/7	93.60%	0.139/0.089		web-cart-catalogue	1/13/22	80.10%	0.256/0.073/0.059
	web-cart	1/13/22	80.10%	0.246/0.07/0.057		payment-shipping-ratings	4/32/39	53.00%	0.14/0.048/0.035
4f1s	cart	3/23/-	52.70%	0.136/0.057/0	4f2s	cart-catalogue	1/3/17/22	83.80%	0.258/0.145/0.068/0.061
	catalogue	1/17	85.40%	0.224/0.059		payment-ratings	10/33/38	48.80%	0.076/0.05/0.038
	ratings	11/39	52.70%	0.07/0.033		shipping-user	4/7/-	63.00%	0.135/0.083/0
	web	16/-/-/-	19.00%	0.054/0/0/0		web-cart	1/3/13/21	86.40%	0.256/0.196/0.072/0.06
4f3s	cart-catalogue-payment	1/3/22/47	67.20%	0.261/0.144/0.061/0	4f4s	cart-catalogue-payment-ratings	1/22/24/47	56.20%	0.259/0.06/0.051/0
	ratings-shipping-user	4/7/10/39	74.80%	0.135/0.104/0.071/0.034		shipping-user-web-cart	5/13/22/38	65.20%	0.121/0.072/0.059/0
	web-cart-catalogue	1/13/22	85.90%	0.255/0.072/0.059		payment-web-catalogue-user	1/5/13/47	70.90%	0.257/0.128/0.073/0
	payment-shipping-user	4/7/32	75.40%	0.141/0.097/0.049		cart-ratings-shipping-payment	22/24/38/47	37.10%	0.058/0.047/0/0
5f1s	cart	3/22/44/-	43.80%	0.141/0.059/0/0	5f2s	cart-catalogue	1/3/18/22/44	69.80%	0.171/0.149/0.07/0.063/0
	shipping	4	95.80%	0.143		payment-ratings	10/32/38	49.50%	0.076/0.051/0.039
	user	7/-/-/-	23.80%	0.089/0/0/0		shipping-user	4/7/-	63.00%	0.136/0.097/0
	web	16/-/-/-	19.00%	0.055/0/0/0		web-cart	1/3/13/22	85.90%	0.245/0.188/0.07/0.057
5f3s	cart-catalogue-payment	1/3/18/22/47	68.20%	0.231/0.125/0.06/0.053/0	5f4s	cart-catalogue-payment-ratings	1/3/21/24/47	65.90%	0.262/0.145/0.062/0.049/0
	ratings-shipping-user	4/7/9/38	75.90%	0.14/0.091/0.074/0.038		shipping-user-web-cart	5/13/22/38	65.20%	0.12/0.072/0.059/0
	web-cart-catalogue	1/3/13/22	89.30%	0.258/0.142/0.073/0.06		payment-shipping-user-cart	5/22/38/47	46.90%	0.124/0.058/0/0
	payment-shipping-ratings	4/32/39	53.00%	0.139/0.048/0.037		catalogue-ratings-web-cart	1/13/19/22/24	73.90%	0.256/0.073/0.067/0.06/0.048
5f5s	cart-catalogue-payment-ratings-shipping	1/22/24/38/47	50.30%	0.256/0.059/0.051/0/0					
	user-web-cart-catalogue-payment	1/5/13/22/47	69.40%	0.257/0.13/0.073/0.06/0					
	ratings-shipping-user-web-payment	4/7/17/33/39	65.30%	0.14/0.098/0.055/0.048/0.038					
	cart-catalogue-web-ratings-payment	1/13/22/24/47	61.30%	0.255/0.073/0.06/0.049/0					

A.3 Ranking Ties

Table A.6: Individual accuracies in the scenario suite using the best-case tie-breaking strategy. The rank has 49 entities.

Category	Combination	Entities Ranking	Accuracy	Entities SFL Value	Category	Combination	Entities Ranking	Accuracy	Entities SFL Value
1f1s	cart	22	58.40%	0.05	2f1s	cart	3/22	78.40%	0.117/0.049
	catalogue	1	100.00%	0.257		catalogue	1/11	91.60%	0.258/0.067
	user	4	95.50%	0.106		user	5/-	47.50%	0.089/0
	web	13	77.00%	0.055		web	13/-	39.30%	0.054/0
2f2s	cart-catalogue	1/20	81.90%	0.262/0.062	3f1s	cart	2/22	79.40%	0.118/0.049
	payment-ratings	33/36	32.70%	0.048/0.04		ratings	6/38	58.00%	0.071/0.035
	shipping-user	3/5	96.20%	0.14/0.086		shipping	3	97.40%	0.139
	web-cart	10/21	72.50%	0.071/0.059		user	5/-/-	32.20%	0.088/0/0
3f2s	cart-catalogue	1/2/20	88.40%	0.262/0.145/0.061	3f3s	cart-catalogue-payment	1/20/38	62.00%	0.26/0.061/0
	payment-ratings	28/38	35.70%	0.048/0.036		ratings-shipping-user	3/4/38	73.50%	0.14/0.095/0.034
	shipping-user	3/5	96.20%	0.139/0.089		web-cart-catalogue	1/9/20	83.90%	0.256/0.073/0.059
	web-cart	1/10/21	82.70%	0.246/0.07/0.057		payment-shipping-ratings	3/26/38	57.90%	0.14/0.048/0.035
4f1s	cart	2/22/-	53.40%	0.136/0.057/0	4f2s	cart-catalogue	1/2/10/20	88.40%	0.258/0.145/0.068/0.061
	catalogue	1/10	92.70%	0.224/0.059		payment-ratings	6/31/36	53.70%	0.076/0.05/0.038
	ratings	8/38	56.00%	0.07/0.033		shipping-user	3/5/-	64.60%	0.135/0.083/0
	web	13/-/-/-	20.40%	0.054/0/0/0		web-cart	1/2/10/19	88.90%	0.256/0.196/0.072/0.06
4f3s	cart-catalogue-payment	1/2/20/38	73.00%	0.261/0.144/0.061/0	4f4s	cart-catalogue-payment-ratings	1/20/23/38	62.40%	0.259/0.06/0.051/0
	ratings-shipping-user	3/4/7/38	78,3%	0.135/0.104/0.071/0.034		shipping-user-web-cart	4/10/21/38	67.50%	0.121/0.072/0.059/0
	web-cart-catalogue	1/9/20	88.80%	0.255/0.072/0.059		payment-web-catalogue-user	1/3/9/38	78.70%	0.257/0.128/0.073/0
	payment-shipping-user	3/4/27	81.30%	0.141/0.097/0.049		cart-ratings-shipping-payment	21/23/38/38	42.90%	0.058/0.047/0/0
5f1s	cart	2/20/38/-	47.80%	0.141/0.059/0/0	5f2s	cart-catalogue	1/2/11/20/38	75.90%	0.171/0.149/0.07/0.063/0
	shipping	3	97.40%	0.143		payment-ratings	6/26/36	57.20%	0.076/0.051/0.039
	user	4/-/-/-	25.20%	0.089/0/0/0		shipping-user	3/4/-	65.40%	0.136/0.097/0
	web	13/-/-/-	20.50%	0.055/0/0/0		web-cart	1/2/10/21	87.80%	0.245/0.188/0.07/0.057
5f3s	cart-catalogue-payment	1/2/11/21/38	75.40%	0.231/0.125/0.06/0.053/0	5f4s	cart-catalogue-payment-ratings	1/2/19/23/38	71.00%	0.262/0.145/0.062/0.049/0
	ratings-shipping-user	3/5/6/36	79.40%	0.14/0.091/0.074/0.038		shipping-user-web-cart	3/10/21/38	68.00%	0.12/0.072/0.059/0
	web-cart-catalogue	1/2/9/20	91.70%	0.258/0.142/0.073/0.06		payment-shipping-user-cart	3/21/38/38	53.10%	0.124/0.058/0/0
	payment-shipping-ratings	3/28/37	57.20%	0.139/0.048/0.037		catalogue-ratings-web-cart	1/9/13/20/23	79.10%	0.256/0.073/0.067/0.06/0.048
5f5s	cart-catalogue-payment-ratings-shipping	1/20/23/38/38	55.40%	0.256/0.059/0.051/0/0					
	user-web-cart-catalogue-payment	1/3/9/20/38	76.40%	0.257/0.13/0.073/0.06/0					
	ratings-shipping-user-web-payment	3/4/14/28/37	70.90%	0.14/0.098/0.055/0.048/0.038					
	cart-catalogue-web-ratings-payment	1/9/20/23/38	67.90%	0.255/0.073/0.06/0.049/0					

Table A.7: Individual accuracies in the scenario suite using the worst-case tie-breaking strategy. The rank has 49 entities.

Category	Combination	Entities Ranking	Accuracy	Entities SFL Value	Category	Combination	Entities Ranking	Accuracy	Entities SFL Value
1f1s	cart	22	58.40%	0.05	2f1s	cart	3/22	78.40%	0.117/0.049
	catalogue	1	100.00%	0.257		catalogue	1/16	86.40%	0.258/0.067
	user	4	95.50%	0.106		user	5/-	47.50%	0.089/0
	web	13	77.00%	0.055		web	13/-	39.30%	0.054/0
2f2s	cart-catalogue	1/20	81.90%	0.262/0.062	3f1s	cart	2/22	79.40%	0.118/0.049
	payment-ratings	34/37	30.60%	0.048/0.04		ratings	8/39	55.00%	0.071/0.035
	shipping-user	3/5	96.20%	0.14/0.086		shipping	3	97.40%	0.139
	web-cart	10/21	72.50%	0.071/0.059		user	5/-/-	32.20%	0.088/0/0
3f2s	cart-catalogue	1/2/20	88.40%	0.262/0.145/0.061	3f3s	cart-catalogue-payment	1/20/49	55.60%	0.26/0.061/0
	payment-ratings	34/39	28.90%	0.048/0.036		ratings-shipping-user	3/4/39	72.80%	0.14/0.095/0.034
	shipping-user	3/5	96.20%	0.139/0.089		web-cart-catalogue	1/9/20	83.90%	0.256/0.073/0.059
	web-cart	1/10/21	82.70%	0.246/0.07/0.057		payment-shipping-ratings	3/33/39	52.30%	0.14/0.048/0.035
4f1s	cart	2/22/-	53.40%	0.136/0.057/0	4f2s	cart-catalogue	1/2/15/20	85.80%	0.258/0.145/0.068/0.061
	catalogue	1/15	87.50%	0.224/0.059		payment-ratings	8/33/37	50.20%	0.076/0.05/0.038
	ratings	10/39	53.90%	0.07/0.033		shipping-user	3/5/-	64.60%	0.135/0.083/0
	web	13/-/-/-	20.40%	0.054/0/0/0		web-cart	1/2/10/19	88.90%	0.256/0.196/0.072/0.06
4f3s	cart-catalogue-payment	1/2/20/49	67.10%	0.261/0.144/0.061/0	4f4s	cart-catalogue-payment-ratings	1/20/24/49	55.90%	0.259/0.06/0.051/0
	ratings-shipping-user	3/4/9/39	76.70%	0.135/0.104/0.071/0.034		shipping-user-web-cart	4/10/21/49	61.50%	0.121/0.072/0.059/0
	web-cart-catalogue	1/9/20	88.80%	0.255/0.072/0.059		payment-web-catalogue-user	1/3/9/49	72.70%	0.257/0.128/0.073/0
	payment-shipping-user	3/4/33	77.10%	0.141/0.097/0.049		cart-ratings-shipping-payment	21/24/49/49	30.50%	0.058/0.047/0/0
5f1s	cart	2/20/49/-	41.90%	0.141/0.059/0/0	5f2s	cart-catalogue	1/2/15/20/49	68.90%	0.171/0.149/0.07/0.063/0
	shipping	3	97.40%	0.143		payment-ratings	8/33/37	50.20%	0.076/0.051/0.039
	user	4/-/-/-	25.20%	0.089/0/0/0		shipping-user	3/4/-	65.40%	0.136/0.097/0
	web	13/-/-/-	20.50%	0.055/0/0/0		web-cart	1/2/10/21	87.80%	0.245/0.188/0.07/0.057
5f3s	cart-catalogue-payment	1/2/16/21/49	68.40%	0.231/0.125/0.06/0.053/0	5f4s	cart-catalogue-payment-ratings	1/2/19/24/49	65.70%	0.262/0.145/0.062/0.049/0
	ratings-shipping-user	3/5/7/37	78.40%	0.14/0.091/0.074/0.038		shipping-user-web-cart	3/10/21/49	62.00%	0.12/0.072/0.059/0
	web-cart-catalogue	1/2/9/20	91.70%	0.258/0.142/0.073/0.06		payment-shipping-user-cart	3/21/49/49	41.30%	0.124/0.058/0/0
	payment-shipping-ratings	3/33/38	53.00%	0.139/0.048/0.037		catalogue-ratings-web-cart	1/9/18/20/24	78.70%	0.256/0.073/0.067/0.06/0.048
5f5s	cart-catalogue-payment-ratings-shipping	1/20/24/49/49	45.30%	0.256/0.059/0.051/0/0					
	user-web-cart-catalogue-payment	1/3/9/20/49	71.50%	0.257/0.13/0.073/0.06/0					
	ratings-shipping-user-web-payment	3/4/14/33/38	68.30%	0.14/0.098/0.055/0.048/0.038					
	cart-catalogue-web-ratings-payment	1/9/20/24/49	62.60%	0.255/0.073/0.06/0.049/0					

Table A.8: Individual accuracies in the scenario suite using the average-case tie-breaking strategy. The rank has 49 entities.

Category	Combination	Entities Ranking	Accuracy	Entities SFL Value	Category	Combination	Entities Ranking	Accuracy	Entities SFL Value
1f1s	cart	22	58.40%	0.05	2f1s	cart	3/22	78.40%	0.117/0.049
	catalogue	1	100.00%	0.257		catalogue	1/14	88.50%	0.258/0.067
	user	4	95.50%	0.106		user	5/-	47.50%	0.089/0
	web	13	77.00%	0.055		web	13/-	39.30%	0.054/0
2f2s	cart-catalogue	1/20	81.90%	0.262/0.062	3f1s	cart	2/22	79.40%	0.118/0.049
	payment-ratings	34/37	32.70%	0.048/0.04		ratings	7/39	56.00%	0.071/0.035
	shipping-user	3/5	96.20%	0.14/0.086		shipping	3	97.40%	0.139
	web-cart	10/21	72.50%	0.071/0.059		user	5/-/-	32.20%	0.088/0/0
3f2s	cart-catalogue	1/2/20	88.40%	0.262/0.145/0.061	3f3s	cart-catalogue-payment	1/20/44	59.10%	0.26/0.061/0
	payment-ratings	31/39	31.60%	0.048/0.036		ratings-shipping-user	3/4/39	72.80%	0.14/0.095/0.034
	shipping-user	3/5	96.20%	0.139/0.089		web-cart-catalogue	1/9/20	83.90%	0.256/0.073/0.059
	web-cart	1/10/21	82.70%	0.246/0.07/0.057		payment-shipping-ratings	3/30/39	54.40%	0.14/0.048/0.035
4f1s	cart	2/22/-	53.40%	0.136/0.057/0	4f2s	cart-catalogue	1/2/13/20	86.80%	0.258/0.145/0.068/0.061
	catalogue	1/13	91.60%	0.224/0.059		payment-ratings	7/32/37	51.60%	0.076/0.05/0.038
	ratings	9/39	55.00%	0.07/0.033		shipping-user	3/5/-	64.60%	0.135/0.083/0
	web	13/-/-/-	20.40%	0.054/0/0/0		web-cart	1/2/10/19	88.90%	0.256/0.196/0.072/0.06
4f3s	cart-catalogue-payment	1/2/20/44	69.80%	0.261/0.144/0.061/0	4f4s	cart-catalogue-payment-ratings	1/20/24/44	58.60%	0.259/0.06/0.051/0
	ratings-shipping-user	3/4/8/39	77.80%	0.135/0.104/0.071/0.034		shipping-user-web-cart	4/10/21/44	64.20%	0.121/0.072/0.059/0
	web-cart-catalogue	1/9/20	88.80%	0.255/0.072/0.059		payment-web-catalogue-user	1/3/9/44	75.40%	0.257/0.128/0.073/0
	payment-shipping-user	3/4/30	79.20%	0.141/0.097/0.049		cart-ratings-shipping-payment	21/24/44/44	35.90%	0.058/0.047/0/0
5f1s	cart	2/20/44/-	44.60%	0.141/0.059/0/0	5f2s	cart-catalogue	1/2/14/20/44	72.00%	0.171/0.149/0.07/0.063/0
	shipping	3	97.40%	0.143		payment-ratings	7/30/03	53.00%	0.076/0.051/0.039
	user	4/-/-/-	25.20%	0.089/0/0/0		shipping-user	3/4/-	65.40%	0.136/0.097/0
	web	13/-/-/-	20.50%	0.055/0/0/0		web-cart	1/2/10/21	87.80%	0.245/0.188/0.07/0.057
5f3s	cart-catalogue-payment	1/2/14/21/40	71.50%	0.231/0.125/0.06/0.053/0	5f4s	cart-catalogue-payment-ratings	1/2/19/24/44	67.90%	0.262/0.145/0.062/0.049/0
	ratings-shipping-user	3/5/7/37	78.40%	0.14/0.091/0.074/0.038		shipping-user-web-cart	3/10/21/44	64.70%	0.12/0.072/0.059/0
	web-cart-catalogue	1/2/9/20	91.70%	0.258/0.142/0.073/0.06		payment-shipping-user-cart	3/21/44/44	46.70%	0.124/0.058/0/0
	payment-shipping-ratings	3/31/38	54.40%	0.139/0.048/0.037		catalogue-ratings-web-cart	1/9/16/20/24	77.40%	0.256/0.073/0.067/0.06/0.048
5f5s	cart-catalogue-payment-ratings-shipping	1/20/24/44/44	49.70%	0.256/0.059/0.051/0/0					
	user-web-cart-catalogue-payment	1/3/9/20/44	73.70%	0.257/0.13/0.073/0.06/0					
	ratings-shipping-user-web-payment	3/4/14/31/38	69.10%	0.14/0.098/0.055/0.048/0.038					
	cart-catalogue-web-ratings-payment	1/9/20/24/44	64.80%	0.255/0.073/0.06/0.049/0					

A.4 Method Entity Percentage

Table A.9: Individual accuracies in the scenario suite using 75% of services with endpoints as method invocations. The rank has 30 entities.

Category	Combination	Entities Ranking	Accuracy	Entities SFL Value	Category	Combination	Entities Ranking	Accuracy	Entities SFL Value
1f1s	cart	12	64.90%	0.026	2f1s	cart	7/12	74.80%	0.058/0.025
	catalogue	4	93.10%	0.153		catalogue	4/9	85.90%	0.154/0.042
	user	-	2.60%	0		user	-/-	2.50%	0/0
	web	-	3.40%	0		web	-/-	3.60%	0/0
2f2s	cart-catalogue	4/11	82.00%	0.159/0.039	3f1s	cart	7/12	74.80%	0.059/0.025
	payment-ratings	18/19	43.40%	0.033/0.028		ratings	11/20	53.80%	0.047/0.023
	shipping-user	9/-	38.90%	0.092/0		shipping	8	79.40%	0.091
	web-cart	11/-	36.00%	0.038/0		user	-/-/-	2.60%	0/0/0
3f2s	cart-catalogue	4/7/11	84.10%	0.16/0.089/0.039	3f3s	cart-catalogue-payment	4/11/28	57.70%	0.158/0.038/0
	payment-ratings	16/20	44.80%	0.033/0.025		ratings-shipping-user	8/20/-	40.10%	0.094/0.023/0
	shipping-user	9/-	38.90%	0.092/0		web-cart-catalogue	4/11/-	56.00%	0.153/0.037/0
	web-cart	3/11/-	57.10%	0.149/0.036/0		payment-shipping-ratings	9/17/20	55.00%	0.093/0.033/0.024
4f1s	cart	7/12/-	50.90%	0.07/0.03/0	4f2s	cart-catalogue	4/7/9/11	84.10%	0.156/0.089/0.043/0.039
	catalogue	4/9	85.90%	0.114/0.031		payment-ratings	11/17/19	54.00%	0.052/0.034/0.026
	ratings	12/20	52.10%	0.046/0.022		shipping-user	9/-/-	26.80%	0.087/0/0
	web	-/-/-/-	3.90%	0/0/0/0		web-cart	3/6/11/-	66.00%	0.161/0.124/0.039/0
4f3s	cart-catalogue-payment	4/7/11/28	65.40%	0.159/0.089/0.039/0	4f4s	cart-catalogue-payment-ratings	4/11/12/28	61.00%	0.157/0.038/0.033/0
	ratings-shipping-user	10/11/20/-	46.70%	0.089/0.047/0.024/0		shipping-user-web-cart	11/25/-/-	23.80%	0.039/0/0/0
	web-cart-catalogue	4/11/-	66.80%	0.152/0.037/0		payment-web-catalogue-user	4/28/-/-	27.30%	0.154/0/0/0
	payment-shipping-user	8/16/-	44.70%	0.092/0.033/0		cart-ratings-shipping-payment	11/12/25/28	41.40%	0.037/0.3/0/0
5f1s	cart	7/11/29/-	41.00%	0.075/0.032/0/0	5f2s	cart-catalogue	4/7/9/11/29	68.80%	0.1/0.88/0.043/0.038/0
	shipping	8	79.40%	0.1		payment-ratings	11/16/19	55.20%	0.053/0.035/0.027
	user	-/-/-/-	2.70%	0/0/0/0		shipping-user	9/-/-	26.90%	0.09/0/0
	web	-/-/-/-	3.90%	0/0/0/0		web-cart	3/6/11/30	65.80%	0.148/0.114/0.036/0
5f3s	cart-catalogue-payment	4/7/9/12/28	68.60%	0.12/0.066/0.033/0.029/0	5f4s	cart-catalogue-payment-ratings	4/6/11/12/28	67.80%	0.159/0.089/0.039/0.031/0
	ratings-shipping-user	9/11/19/-	48.40%	0.095/0.051/0.026/0		shipping-user-web-cart	11/25/-/-	23.80%	0.038/0/0/0
	web-cart-catalogue	4/7/11/-	71.80%	0.155/0.087/0.038/0		payment-shipping-user-cart	11/25/28/-	25.00%	0.047/0/0/0
	payment-shipping-ratings	8/16/20	57.30%	0.089/0.032/0.025		catalogue-ratings-web-cart	4/9/11/12/-	64.60%	0.153/0.042/0.037/0.03/0
5f5s	cart-catalogue-payment-ratings-shipping	4/11/12/25/28	53.00%	0.153/0.037/0.032/0/0					
	user-web-cart-catalogue-payment	4/11/28/-/-	36.20%	0.154/0.038/0/0/0					
	ratings-shipping-user-web-payment	8/17/20/-/-	36.10%	0.092/0.033/0.036/0/0					
	cart-catalogue-web-ratings-payment	4/11/12/28/-	49.70%	0.153/0.037/0.031/0/0					

Table A.10: Individual accuracies in the scenario suite using 50% of services with endpoints as method invocations. The rank has 20 entities.

Category	Combination	Entities Ranking	Accuracy	Entities SFL Value	Category	Combination	Entities Ranking	Accuracy	Entities SFL Value
1f1s	cart	11	51.20%	0.026	2f1s	cart	7/11	63.30%	0.058/0.025
	catalogue	4	89.50%	0.153		catalogue	4/9	78.20%	0.154/0.042
	user	-	3.30%	0		user	-/-	3.20%	0/0
	web	-	5.30%	0		web	-/-	5.60%	0/0
2f2s	cart-catalogue	4/10	74.80%	0.159/0.039	3f1s	cart	7/11	63.30%	0.059/0.025
	payment-ratings	15/-	17.40%	0.033/0		ratings	-/-	4.70%	0/0
	shipping-user	-/-	3.50%	0/0		shipping	-	3.80%	0
	web-cart	10/-	31.20%	0.038/0		user	-/-/-	3.40%	0/0/0
3f2s	cart-catalogue	4/7/10	76.80%	0.16/0.089/0.039	3f3s	cart-catalogue-payment	4/10/18	54.30%	0.158/0.038/0
	payment-ratings	13/-	22.40%	0.032/0		ratings-shipping-user	-/-/-	4.20%	0/0/0
	shipping-user	-/-	3.60%	0/0		web-cart-catalogue	4/10/-	51.90%	0.153/0.037/0
	web-cart	3/10/-	53.60%	0.149/0.036/0		payment-shipping-ratings	14/-/-	14.90%	0.033/0/0
4f1s	cart	7/11/-	43.80%	0.07/0.03/0	4f2s	cart-catalogue	4/7/9/10	76.20%	0.156/0.088/0.043/0.039
	catalogue	4/9	78.20%	0.114/0.031		payment-ratings	14/-/-	15.00%	0.034/0/0
	ratings	-/-	4.70%	0/0		shipping-user	-/-/-	3.60%	0/0/0
	web	-/-/-/-	6.30%	0/0/0/0		web-cart	3/6/10/-	62.00%	0.161/0.124/0.039/0
4f3s	cart-catalogue-payment	4/7/10/18	61.00%	0.159/0.089/0.039/0	4f4s	cart-catalogue-payment-ratings	4/10/18/-	42.40%	0.157/0.038/0/0
	ratings-shipping-user	-/-/-/-	4.80%	0/0/0/0		shipping-user-web-cart	10/-/-/-	17.30%	0.039/0/0/0
	web-cart-catalogue	4/10/-	63.70%	0.152/0.037/0		payment-web-catalogue-user	4/18/-/-	28.60%	0.154/0/0/0
	payment-shipping-user	13/-/-	16.10%	0.033/0/0		cart-ratings-shipping-payment	10/18/-/-	19.10%	0.037/0/0/0
5f1s	cart	7/10/19/-	37.40%	0.075/0.032/0/0	5f2s	cart-catalogue	4/7/9/10/19	63.40%	0.1/0.088/0.043/0.038/0
	shipping	-	3.80%	0		payment-ratings	13/-/-	16.80%	0.035/0/0
	user	-/-/-/-	3.60%	0/0/0/0		shipping-user	-/-/-	3.70%	0/0/0
	web	-/-/-/-	6.30%	0/0/0/0		web-cart	3/6/10/-	62.00%	0.148/0.114/0.036/0
5f3s	cart-catalogue-payment	4/7/9/11/18	63.00%	0.12/0.066/0.033/0.029/0	5f4s	cart-catalogue-payment-ratings	4/6/10/18/-	51.60%	0.159/0.089/0.039/0/0
	ratings-shipping-user	-/-/-/-	4.80%	0/0/0/0		shipping-user-web-cart	10/-/-/-	17.40%	0.038/0/0/0
	web-cart-catalogue	4/7/10/-	68.20%	0.155/0.087/0.038/0		payment-shipping-user-cart	10/18/-/-	19.00%	0.037/0/0/0
	payment-shipping-ratings	13/-/-	16.60%	0.032/0/0		catalogue-ratings-web-cart	4/9/10/-/-	46.90%	0.153/0.042/0.037/0/0
5f5s	cart-catalogue-payment-ratings-shipping	4/10/18/-/-	34.40%	0.153/0.037/0/0/0					
	user-web-cart-catalogue-payment	4/10/18/-/-	35.30%	0.154/0.038/0/0/0					
	ratings-shipping-user-web-payment	14/-/-/-/-	11.90%	0.033/0/0/0/0					
	cart-catalogue-web-ratings-payment	4/10/18/-/-	35.50%	0.153/0.037/0/0/0					

Table A.11: Individual accuracies in the scenario suite using 25% of services with endpoints as method invocations. The rank has 17 entities.

Category	Combination	Entities Ranking	Accuracy	Entities SFL Value	Category	Combination	Entities Ranking	Accuracy	Entities SFL Value
1f1s	cart	11	45.30%	0.026	2f1s	cart	7/11	58.60%	0.058/0.025
	catalogue	4	88.00%	0.153		catalogue	4/9	75.60%	0.154/0.042
	user	-	3.50%	0		user	-/-	3.30%	0/0
	web	-	5.90%	0		web	-/-	6.30%	0/0
2f2s	cart-catalogue	4/10	71.70%	0.159/0.039	3f1s	cart	7/11	58.60%	0.059/0.025
	payment-ratings	-/-	4.60%	0/0		ratings	-/-	5.20%	0/0
	shipping-user	-/-	3.70%	0/0		shipping	-/-	4.00%	0
	web-cart	10/-	28.90%	0.038/0		user	-/-/-	3.50%	0/0/0
3f2s	cart-catalogue	4/7/10	73.70%	0.16/0.089/0.039	3f3s	cart-catalogue-payment	4/10/-	48.20%	0.158/0.038/0
	payment-ratings	-/-	4.20%	0/0		ratings-shipping-user	-/-/-	4.50%	0/0/0
	shipping-user	-/-	3.80%	0/0		web-cart-catalogue	4/10/-	50.00%	0.153/0.037/0
	web-cart	3/10/-	52.00%	0.149/0.036/0		payment-shipping-ratings	-/-/-	4.80%	0/0/0
4f1s	cart	7/11/-	40.90%	0.07/0.03/0	4f2s	cart-catalogue	4/7/9/10	72.90%	0.156/0.088/0.043/0.039
	catalogue	4/9	75.60%	0.114/0.031		payment-ratings	-/-/-	4.80%	0/0/0
	ratings	-/-	5.20%	0/0		shipping-user	-/-/-	3.80%	0/0/0
	web	-/-/-/-	7.20%	0/0/0/0		web-cart	3/6/10/-	60.30%	0.161/0.124/0.039/0
4f3s	cart-catalogue-payment	4/7/10/-	55.60%	0.159/0.089/0.039/0	4f4s	cart-catalogue-payment-ratings	4/10/-/-	38.10%	0.157/0.038/0/0
	ratings-shipping-user	-/-/-/-	5.30%	0/0/0/0		shipping-user-web-cart	10/-/-/-	16.30%	0.039/0/0/0
	web-cart-catalogue	4/10/-	62.20%	0.152/0.037/0		payment-web-catalogue-user	4/-/-/-	25.60%	0.154/0/0/0
	payment-shipping-user	-/-/-	4.00%	0/0/0		cart-ratings-shipping-payment	10/-/-/-	15.10%	0.037/0/0/0
5f1s	cart	7/10/17/-	35.80%	0.075/0.032/0/0	5f2s	cart-catalogue	4/7/9/10/17	61.10%	0.1/0.088/0.038/0
	shipping	-	4.00%	0		payment-ratings	-/-/-	4.80%	0/0/0
	user	-/-/-/-	3.80%	0/0/0/0		shipping-user	-/-/-	3.90%	0/0/0
	web	-/-/-/-	7.20%	0/0/0/0		web-cart	3/6/10/-	60.30%	0.148/0.114/0.036/0
5f3s	cart-catalogue-payment	4/7/9/11/-	57.70%	0.12/0.066/0.033/0.029/0	5f4s	cart-catalogue-payment-ratings	4/6/10/-/-	47.70%	0.159/0.089/0.039/0/0
	ratings-shipping-user	-/-/-/-	5.30%	0/0/0/0		shipping-user-web-cart	10/-/-/-	16.50%	0.038/0/0/0
	web-cart-catalogue	4/7/10/-	66.70%	0.155/0.087/0.38/0		payment-shipping-user-cart	10/-/-/-	15.00%	0.037/0/0/0
	payment-shipping-ratings	-/-/-	4.70%	0/0/0		catalogue-ratings-web-cart	4/9/10/-/-	45.40%	0.153/0.042/0.037/0/0
5f5s	cart-catalogue-payment-ratings-shipping	4/10/-/-/-	31.00%	0.153/0.037/0/0/0					
	user-web-cart-catalogue-payment	4/10/-/-/-	32.10%	0.154/0.038/0/0/0					
	ratings-shipping-user-web-payment	-/-/-/-/-	5.40%	0/0/0/0/0					
	cart-catalogue-web-ratings-payment	4/10/-/-/-	32.40%	0.153/0.037/0/0/0					

Table A.12: Individual accuracies in the scenario suite using 0% of services with endpoints as method invocations. The rank has 8 entities.

Category	Combination	Entities Ranking	Accuracy	Entities SFL Value	Category	Combination	Entities Ranking	Accuracy	Entities SFL Value
1f1s	cart	-	6.30%	0	2f1s	cart	-/-	7.20%	0/0
	catalogue	-	12.50%	0		catalogue	-/-	14.40%	0/0
	user	-	3.10%	0		user	-/-	3.60%	0/0
	web	-	12.50%	0		web	-/-	14.40%	0/0
2f2s	cart-catalogue	-/-	12.10%	0/0	3f1s	cart	-/-	7.20%	0/0
	payment-ratings	-/-	7.00%	0/0		ratings	-/-	9.00%	0/0
	shipping-user	-/-	4.40%	0/0		shipping	-	6.30%	0
	web-cart	-/-	11.00%	0/0		user	-/-/-	4.30%	0/0/0
3f2s	cart-catalogue	-/-/-	13.40%	0/0/0	3f3s	cart-catalogue-payment	-/-/-	9.40%	0/0/0
	payment-ratings	-/-	5.90%	0/0		ratings-shipping-user	-/-/-	8.40%	0/0/0
	shipping-user	-/-	5.20%	0/0		web-cart-catalogue	-/-/-	14.00%	0/0/0
	web-cart	-/-/-	13.80%	0/0/0		payment-shipping-ratings	-/-/-	8.90%	0/0/0
4f1s	cart	-/-/-	8.50%	0/0/0	4f2s	cart-catalogue	-/-/-/-	15.50%	0/0/0/0
	catalogue	-/-	14.40%	0/0		payment-ratings	-/-/-	7.90%	0/0/0
	ratings	-/-	9.00%	0/0		shipping-user	-/-/-	4.80%	0/0/0
	web	-/-/-/-	20.80%	0/0/0/0		web-cart	-/-/-/-	16.40%	0/0/0/0
4f3s	cart-catalogue-payment	-/-/-/-	11.10%	0/0/0/0	4f4s	cart-catalogue-payment-ratings	-/-/-/-	13.00%	0/0/0/0
	ratings-shipping-user	-/-/-/-	10.80%	0/0/0/0		shipping-user-web-cart	-/-/-/-	7.70%	0/0/0/0
	web-cart-catalogue	-/-/-/-	17.50%	0/0/0/0		payment-web-catalogue-user	-/-/-/-	12.90%	0/0/0/0
	payment-shipping-user	-/-/-	6.10%	0/0/0		cart-ratings-shipping-payment	-/-/-/-	11.70%	0/0/0/0
5f1s	cart	-/-/-/-	13.00%	0/0/0/0	5f2s	cart-catalogue	-/-/-/-/-	17.40%	0/0/0/0/0
	shipping	-	6.30%	0		payment-ratings	-/-/-	7.90%	0/0/0
	user	-/-/-/-	5.20%	0/0/0/0		shipping-user	-/-/-	5.30%	0/0/0
	web	-/-/-/-	20.80%	0/0/0/0		web-cart	-/-/-/-	16.40%	0/0/0/0
5f3s	cart-catalogue-payment	-/-/-/-	16.90%	0/0/0/0/0	5f4s	cart-catalogue-payment-ratings	-/-/-/-/-	17.10%	0/0/0/0/0
	ratings-shipping-user	-/-/-/-	11.30%	0/0/0/0		shipping-user-web-cart	-/-/-/-	12.70%	0/0/0/0
	web-cart-catalogue	-/-/-/-/-	21.70%	0/0/0/0/0		payment-shipping-user-cart	-/-/-/-	9.00%	0/0/0/0
	payment-shipping-ratings	-/-/-	8.40%	0/0/0		catalogue-ratings-web-cart	-/-/-/-/-	22.10%	0/0/0/0/0
5f5s	cart-catalogue-payment-ratings-shipping	-/-/-/-/-	13.90%	0/0/0/0/0					
	user-web-cart-catalogue-payment	-/-/-/-/-	19.10%	0/0/0/0/0					
	ratings-shipping-user-web-payment	-/-/-/-/-	15.20%	0/0/0/0/0					
	cart-catalogue-web-ratings-payment	-/-/-/-/-	17.50%	0/0/0/0/0					

Table A.13: Individual accuracies in the scenario suite targeting service entities with no method invocations. The rank has 8 entities.

Category	Combination	Entities Ranking	Accuracy	Entities SFL Value	Category	Combination	Entities Ranking	Accuracy	Entities SFL Value
1f1s	cart	5	50.00%	0.119	2f1s	cart	5	50.00%	0.117
	catalogue	1	100.00%	0.294		catalogue	1	100.00%	0.295
	user	7	25.00%	0.139		user	7	25.00%	0.127
	web	1	100.00%	0.287		web	1	100.00%	0.277
2f2s	cart-catalogue	1/4	85.70%	0.299/0.148	3f1s	cart	5	50.00%	0.117
	payment-ratings	4/6	52.70%	0.191/0.149		ratings	4	62.50%	0.182
	shipping-user	6/7	33.00%	0.156/0.124		shipping	5	50.00%	0.172
	web-cart	2/4	79.50%	0.283/0.231		user	7	25.00%	0.126
3f2s	cart-catalogue	1/5	78.60%	0.299/0.147	3f3s	cart-catalogue-payment	1/5/7	63.50%	0.297/0.145/0.012
	payment-ratings	4/7	45.50%	0.189/0.105		ratings-shipping-user	4/5/7	51.00%	0.183/0.156/0.132
	shipping-user	5/7	39.30%	0.154/0.127		web-cart-catalogue	1/2/5	88.90%	0.292/0.286/0.142
	web-cart	2/4	90.30%	0.276/0.225		payment-shipping-ratings	4/5/6	56.50%	0.182/0.156/0.147
4f1s	cart	5	50.00%	0.136	4f2s	cart-catalogue	1/5	78.60%	0.296/0.147
	catalogue	1	100.00%	0.258		payment-ratings	4/7	45.50%	0.192/0.108
	ratings	4	62.50%	0.177		shipping-user	6/7	33.00%	0.15/0.121
	web	1	100.00%	0.279		web-cart	2/4	79.50%	0.286/0.236
4f3s	cart-catalogue-payment	1/5/7	63.50%	0.298/0.147/0.012	4f4s	cart-catalogue-payment-ratings	1/3/5/7	73.10%	0.296/0.286/0.144/0.012
	ratings-shipping-user	4/6/7	46.20%	0.18/0.151/0.138		shipping-user-web-cart	2/4/5/8	61.40%	0.284/0.233/0.144/0.005
	web-cart-catalogue	1/2/5	88.90%	0.29/0.286/0.14		payment-web-catalogue-user	1/2/4/7	80.80%	0.293/0.289/0.152/0.012
	payment-shipping-user	5/6/7	42.10%	0.157/0.135/0.107		cart-ratings-shipping-payment	3/4/7/8	49.90%	0.245/0.227/0.012/0.005
5f1s	cart	5	50.00%	0.141	5f2s	cart-catalogue	3/5	66.10%	0.218/0.151
	shipping	5	50.00%	0.177		payment-ratings	4/7	45.50%	0.192/0.11
	user	7	25.00%	0.126		shipping-user	5/7	39.30%	0.151/0.132
	web	1	100.00%	0.287		web-cart	2/4	79.50%	0.275/0.224
5f3s	cart-catalogue-payment	2/5/7	59.30%	0.265/0.126/0.016	5f4s	cart-catalogue-payment-ratings	1/3/4/7	77.30%	0.3/0.284/0.147/0.012
	ratings-shipping-user	4/5/7	51.00%	0.188/0.156/0.129		shipping-user-web-cart	2/4/5/8	61.40%	0.281/0.232/0.144/0.005
	web-cart-catalogue	1/2/5	88.90%	0.295/0.29/0.144		payment-shipping-user-cart	4/5/7/8	43.20%	0.228/0.147/0.012/0.005
	payment-shipping-ratings	4/5/7	51.00%	0.188/0.157/0.104		catalogue-ratings-web-cart	1/2/3/5	95.00%	0.292/0.288/0.279/0.143
5f5s	cart-catalogue-payment-ratings-shipping	1/3/5/7/8	63.50%	0.292/0.285/0.142/0.012/0.005					
	user-web-cart-catalogue-payment	1/2/4/5/7	82.30%	0.293/0.289/0.154/0.144/0.012					
	ratings-shipping-user-web-payment	1/4/5/6/7	69.60%	0.29/0.188/0.174/0.135/0.106					
	cart-catalogue-web-ratings-payment	1/2/3/5/7	86.00%	0.292/0.288/0.28/0.142/0.012					

Appendix B

Most Accurate Configuration Individual Scenarios Execution Times

Table B.1: Individual execution times (in seconds) in the most accurate scenario suite (service weight attenuation with average).

Category	Combination	Execution Time (s)	Category	Combination	Execution Time (s)
1f1s	cart	5.2879	2f1s	cart	4.7905
	catalogue	6.4979		catalogue	7.4642
	user	7.8031		user	7.7746
	web	8.7980		web	6.6950
2f2s	cart-catalogue	6.5107	3f1s	cart	4.4619
	payment-ratings	7.8597		ratings	6.9475
	shipping-user	6.8757		shipping	7.6175
	web-cart	6.9324		user	8.5373
3f2s	cart-catalogue	8.0874	3f3s	cart-catalogue-payment	6.9980
	payment-ratings	6.7925		ratings-shipping-user	7.2054
	shipping-user	7.3114		web-cart-catalogue	6.2890
	web-cart	8.9644		payment-shipping-ratings	8.9848
4f1s	cart	4.7856	4f2s	cart-catalogue	6.0035
	catalogue	4.7722		payment-ratings	7.3988
	ratings	6.6124		shipping-user	7.2630
	web	6.8019		web-cart	8.0032
4f3s	cart-catalogue-payment	7.3888	4f4s	cart-catalogue-payment-ratings	6.2888
	ratings-shipping-user	6.6583		shipping-user-web-cart	6.8577
	web-cart-catalogue	6.8975		payment-web-catalogue-user	7.0032
	payment-shipping-user	7.1780		cart-ratings-shipping-payment	6.4100
5f1s	cart	4.8807	5f2s	cart-catalogue	5.9315
	shipping	7.3563		payment-ratings	9.3037
	user	6.2665		shipping-user	7.5007
	web	7.4484		web-cart	6.6604
5f3s	cart-catalogue-payment	5.0214	5f4s	cart-catalogue-payment-ratings	7.1980
	ratings-shipping-user	12.1451		shipping-user-web-cart	7.0452
	web-cart-catalogue	7.7170		payment-shipping-user-cart	7.2863
	payment-shipping-ratings	8.7974		catalogue-ratings-web-cart	6.9070
5f5s	cart-catalogue-payment-ratings-shipping	6.5735			
	user-web-cart-catalogue-payment	6.7103			
	ratings-shipping-user-web-payment	8.4100			
	cart-catalogue-web-ratings-payment	6.1931			

References

[1] Rui Abreu, Peter Zoeteweij, and Arjan J.C. Van Gemund. An evaluation of similarity coefficients for software fault localization. *Proceedings - 12th Pacific Rim International Symposium on Dependable Computing, PRDC 2006*, pages 39–46, 2006.

[2] Rui Abreu, Peter Zoeteweij, and Arjan J.C. Van Gemund. Localizing software faults simultaneously. *Proceedings - International Conference on Quality Software*, pages 367–376, 2009.

[3] Rui Abreu, Peter Zoeteweij, and Arjan J.C. Van Gemund. Spectrum-based multiple fault localization. *ASE2009 - 24th IEEE/ACM International Conference on Automated Software Engineering*, pages 88–99, 2009.

[4] Rui Abreu, Peter Zoeteweij, and Arjan J.C. van Gemund. On the accuracy of spectrum-based fault localization. In *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION (TAICPART-MUTATION 2007)*, pages 89–98, 2007.

[5] Pooja Aggarwal, Ajay Gupta, Prateeti Mohapatra, Seema Nagar, Atri Mandal, Qing Wang, and Amit Paradkar. Localization of operational faults in cloud applications by mining causal dependencies in logs using golden signals. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 12632 LNCS:137–149, 12 2020.

[6] Amazon. Amazon api gateway. Accessible at https://aws.amazon.com/api-gateway/. Accessed in January 2022.

[7] Apache. Apache mesos. Accessible at https://mesos.apache.org/. Accessed in December 2021.

[8] Ivan Beschastnikh, Patty Wang, Yuriy Brun, and Michael D. Ernst. Debugging distributed systems. *Queue*, 14:91–110, 2016.

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

[10] Hongyang Chen, Pengfei Chen, and Guangba Yu. A framework of virtual war room and matrix sketch-based streaming anomaly detection for microservice systems. *IEEE Access*, 8:43413–43426, 2020.

[11] Mike Y. Chen, Emre Kiciman, Eugene Fratkin, Armando Fox, and Eric Brewer. Pinpoint: Problem determination in large, dynamic internet services. *Proceedings of the 2002 International Conference on Dependable Systems and Networks*, pages 595–604, 2002.

[12] Pengfei Chen, Yong Qi, and Di Hou. Causeinfer: Automated end-to-end performance diagnosis with hierarchical causality graph in cloud environment. *IEEE Transactions on Services Computing*, 12(2):214–230, 2019.

[13] Andréia Da, Silva Meyer, Antonio Augusto Franco Garcia, Anete Pereira De Souza, and Cláudio Lopes De Souza. Comparison of similarity coefficients used for cluster analysis with dominant markers in maize (zea mays l). *Genetics and Molecular Biology [online]*, 27(1):83–91, 2004.

[14] Darren Dao, Jeannie Albrecht, Charles Killian, and Amin Vahdat. Live debugging of distributed systems. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 5501 LNCS:94–108, 2009.

[15] Docker. Empowering app development for developers | docker. Accessible at https://www.docker.com/. Accessed in December 2021.

[16] Qingfeng Du, Tiandi Xie, and Yu He. Anomaly detection and diagnosis for container-based microservices with performance monitoring. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 11337 LNCS:560–572, 11 2018.

[17] Elastic. Elasticsearch: The official distributed search & analytics engine. Accessible at https://www.elastic.co/elasticsearch/. Accessed in December 2021.

[18] Elastic. Grok filter plugin | logstash reference [8.2] | elastic. Accessible at https://www.elastic.co/guide/en/logstash/current/plugins-filters-grok.html. Accessed in June 2021.

[19] Elastic. Kibana: Explore, visualize, discover data. Accessible at https://www.elastic.co/kibana/. Accessed in December 2021.

[20] Elastic. Logstash: Collect, parse, transform logs. Accessible at https://www.elastic.co/logstash/. Accessed in December 2021.

[21] Martin Fowler and James Lewis. Microservices. Accessible at https://www.martinfowler.com/articles/microservices.html. Accessed in December 2021.

[22] Paolo Di Francesco, Ivano Malavolta, and Patricia Lago. Research on architecting microservices: Trends, focus, and potential for industrial adoption. *Proceedings - 2017 IEEE International Conference on Software Architecture, ICSA 2017*, pages 21–30, 5 2017.

[23] FudanSELab. Train ticket: a benchmark microservice system. Accessible at https://github.com/FudanSELab/train-ticket/. Accessed in February 2022.

[24] Yu Gan, Yanqi Zhang, Kelvin Hu, Dailun Cheng, Yuan He, Meghna Pancholi, and Christina Delimitrou. Seer: Leveraging big data to navigate the complexity of performance debugging in cloud microservices. *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 19–33, 4 2019.

[25] Elmer Garduno, Soila P. Kavulya, Jiaqi Tan, Rajeev Gandhi, and Priya Narasimhan. Theia: Visual signatures for problem diagnosis in large hadoop clusters. In *26th Large Installation*

System Administration Conference (LISA 12), pages 33–42, San Diego, CA, December 2012. USENIX Association.

[26] Dennis Geels, Gautam Altekar, Petros Maniatis, Timothy Roscoe, and Ion Stoica. Friday: Global comprehension for distributed replay. In *4th USENIX Symposium on Networked Systems Design & Implementation (NSDI 07)*, volume 7, 01 2007.

[27] Debolina Ghosh and Jagannath Singh. A systematic review on program debugging techniques. *Advances in Intelligent Systems and Computing*, 767:193–199, 2020.

[28] GNU. Gdb: The gnu project debugger. Accessible at https://sourceware.org/gdb/. Accessed in December 2021.

[29] Zijie Guan, Jinjin Lin, and Pengfei Chen. On anomaly detection and root cause analysis of microservice systems. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 11434 LNCS:465–469, 11 2018.

[30] Anton Gulenko, Florian Schmidt, Alexander Acker, Marcel Wallschläger, Odej Kao, and Feng Liu. Detecting anomalous behavior of black-box services modeled with distance-based online clustering. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, pages 912–915, 2018.

[31] 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. Association for Computing Machinery, 2020.

[32] Van Haren. Soa governance framework – background. Accessible at https://www.opengroup.org/soa/source-book/gov/p2.htm. Accessed in December 2021.

[33] HashiCorp. Consul by hashicorp. Accessible at https://www.consul.io/. Accessed in January 2022.

[34] Red Hat. What are microservices? Accessible at https://www.redhat.com/en/topics/microservices/what-are-microservices. Accessed in December 2021.

[35] Z. Huang, P. Chen, G. Yu, H. Chen, and Z. Zheng. Sieve: Attention-based sampling of end-to-end trace data in distributed microservice systems. In *2021 IEEE International Conference on Web Services (ICWS)*, pages 436–446, Los Alamitos, CA, USA, sep 2021. IEEE Computer Society.

[36] Instana. Instana - enterprise observability and apm for cloud-native applications. Accessible at https://www.instana.com/. Accessed in February 2022.

[37] Instana. Instana - sample microservice application. Accessible at https://github.com/instana/robot-shop. Accessed in June 2022.

[38] Instana. Instana - sample microservice application. Accessible at https://www.instana.com/blog/stans-robot-shop-sample-microservice-application/. Accessed in June 2022.

[39] Istio. Istio. Accessible at https://istio.io/latest/. Accessed in January 2022.

[40] Tom Janssem, Rui Abreu, and Arjan J.C. Van Gemund. Zoltar: A toolset for automatic fault localization. *ASE2009 - 24th IEEE/ACM International Conference on Automated Software Engineering*, pages 662–664, 2009.

[41] Tong Jia, Pengfei Chen, Lin Yang, Ying Li, Fanjing Meng, and Jingmin Xu. An approach for anomaly diagnosis based on hybrid graph model with logs for distributed services. In *2017 IEEE International Conference on Web Services (ICWS)*, pages 25–32, 2017.

[42] Tong Jia, Lin Yang, Pengfei Chen, Ying Li, Fanjing Meng, and Jingmin Xu. Logsed: Anomaly diagnosis through mining time-weighted control flow graph in logs. In *2017 IEEE 10th International Conference on Cloud Computing (CLOUD)*, pages 447–455, 2017.

[43] Mingxu Jin, Aoran Lv, Yuanpeng Zhu, Zijiang Wen, Yubin Zhong, Zexin Zhao, Jiang Wu, Hejie Li, Hanheng He, and Fengyi Chen. An anomaly detection algorithm for microservice architecture based on robust principal component analysis. *IEEE Access*, 8:226397–226408, 2020.

[44] James A. Jones, Mary Jean Harrold, and John Stasko. Visualization of test information to assist fault localization. *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, page 467, 2002.

[45] Kalapriya Kannan and Anuradha Bhamidipaty. A differential approach for configuration fault localization in cloud environments. *Proceedings of the IEEE International Conference on Cloud Engineering, IC2E 2013*, pages 250–257, 2013.

[46] Myunghwan Kim, Roshan Sumbaly, and Sam Shah. Root cause detection in a service-oriented architecture. *SIGMETRICS Perform. Eval. Rev.*, 41(1):93–104, jun 2013.

[47] Kong. Kong open-source api management gateway for microservices. Accessible at https://konghq.com/kong/. Accessed in January 2022.

[48] Kubernetes. Kubernetes. Accessible at https://kubernetes.io/. Accessed in December 2021.

[49] Idit Levine. Debugging microservices applications. Accessible at https://www.youtube.com/watch?v=3NGa6G_KVns. Accessed in December 2021.

[50] Zeyan Li, Junjie Chen, Rui Jiao, Nengwen Zhao, Zhijun Wang, Shuwei Zhang, Yanjun Wu, Long Jiang, Leiqin Yan, Zikai Wang, Zhekang Chen, Wenchi Zhang, Xiaohui Nie, Kaixin Sui, and Dan Pei. Practical root cause localization for microservice systems via trace analysis. *2021 IEEE/ACM 29th International Symposium on Quality of Service, IWQOS 2021*, pages 1–10, 6 2021.

[51] Weilan Lin, Meng Ma, Disheng Pan, and Ping Wang. Facgraph: Frequent anomaly correlation graph mining for root cause diagnose in micro-service architecture. In *2018 IEEE 37th International Performance Computing and Communications Conference (IPCCC)*, pages 1–8, 2018.

[52] Linkerd. Linkerd. Accessible at https://linkerd.io/. Accessed in January 2022.

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

[54] Guozhi Liu, Bi Huang, Zhihong Liang, Minmin Qin, Hua Zhou, and Zhang Li. Microservices: Architecture, container, and challenges. *Proceedings - Companion of the 2020 IEEE 20th International Conference on Software Quality, Reliability, and Security, QRS-C 2020*, pages 629–635, 12 2020.

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

[56] Xuezheng Liu, Zhenyu Guo, Xi Wang, Feibo Chen, Xiaochen Lian, Jian Tang, Ming Wu, and Zheng Zhang. D3S: Debugging deployed distributed systems. In *5th USENIX Symposium on Networked Systems Design and Implementation (NSDI 08)*, San Francisco, CA, April 2008. USENIX Association.

[57] Meng Ma, Weilan Lin, Disheng Pan, and Ping Wang. Ms-rank: Multi-metric and self-adaptive root cause diagnosis for microservice applications. In *2019 IEEE International Conference on Web Services (ICWS)*, pages 60–67, 2019.

[58] Meng Ma, Jingmin Xu, Yuan Wang, Pengfei Chen, Zonghua Zhang, and Ping Wang. *AutoMAP: Diagnose Your Microservice-Based Web Applications Automatically*, page 246–258. Association for Computing Machinery, New York, NY, USA, 2020.

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

[60] Leonardo Mariani, Cristina Monni, Mauro Pezzé, Oliviero Riganelli, and Rui Xin. Localizing faults in cloud systems. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, pages 262–273, 2018.

[61] Leonardo Mariani, Mauro Pezzè, Oliviero Riganelli, and Rui Xin. Predicting failures in multi-tier distributed systems. *Journal of Systems and Software*, 161:110464, 2020.

[62] João Martins. Spectrum-based fault localization for microservices via log analysis - code repository. Accessible at `https://github.com/joaomrcsmartins/sfl-for-microservices-via-log-analysis`. Accessed in June 2022.

[63] Mihir Mathur. Leveraging distributed tracing and container cloning for replay debugging of microservices. Master's thesis, University of California, 2020.

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

[65] Yuan Meng, Shenglin Zhang, Yongqian Sun, Ruru Zhang, Zhilong Hu, Yiyin Zhang, Chenyang Jia, Zhaogang Wang, and Dan Pei. Localizing failure root causes in a microservice through causality inference. In *2020 IEEE/ACM 28th International Symposium on Quality of Service (IWQoS)*, pages 1–10, 2020.

[66] Open Service Mesh. Open service mesh. Accessible at https://openservicemesh.io/. Accessed in January 2022.

[67] Haibo Mi, Huaimin Wang, Yangfan Zhou, Michael Rung-Tsong Lyu, and Hua Cai. Toward fine-grained, unsupervised, scalable performance diagnosis for production cloud computing systems. *IEEE Transactions on Parallel and Distributed Systems*, 24(6):1245–1255, 2013.

[68] Lee Naish, Hua Jie Lee, and Kotagiri Ramamohanarao. Spectral debugging with weights and incremental ranking. *Proceedings - Asia-Pacific Software Engineering Conference, APSEC*, pages 168–175, 2009.

[69] Lee Naish, Hua Jie Lee, and Kotagiri Ramamohanarao. A model for spectra-based software diagnosis. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 20(3), 8 2011.

[70] Animesh Nandi, Atri Mandal, Shubham Atreja, Gargi B. Dasgupta, and Subhrajit Bhattacharya. Anomaly detection using program control flow graph mining from execution logs. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, page 215–224, New York, NY, USA, 2016. Association for Computing Machinery.

[71] Sasho Nedelkoski, Jorge Cardoso, and Odej Kao. Anomaly detection and classification using distributed tracing and deep learning. In *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pages 241–250, 2019.

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

[73] Francisco Neves, Nuno Machado, and José Pereira. Falcon: A practical log-based analysis tool for distributed systems. *Proceedings - 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2018*, pages 534–541, 7 2018.

[74] Francisco Neves, Nuno Machado, Ricardo Vilaça, and José Pereira. Horus: Non-intrusive causal analysis of distributed systems logs. *Proceedings - 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2021*, pages 212–223, 6 2021.

[75] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. Use of formal methods at amazon web services, 2014.

[76] Hiep Nguyen, Zhiming Shen, Yongmin Tan, and Xiaohui Gu. Fchain: Toward black-box online fault localization for cloud systems. In *2013 IEEE 33rd International Conference on Distributed Computing Systems*, pages 21–30, 2013.

[77] Hiep Nguyen, Yongmin Tan, and Xiaohui Gu. Pal: Propagation-aware anomaly localization for cloud hosted distributed applications. In *Managing Large-Scale Systems via the Analysis of System Logs and the Application of Machine Learning Techniques*, SLAML '11. Association for Computing Machinery, 2011.

[78] OpenTelemetry. Opentelemetry. Accessible at https://opentelemetry.io/. Accessed in February 2022.

[79] OpenTracing. The opentracing project. Accessible at https://opentracing.io/. Accessed in February 2022.

[80] OpenZipkin. Openzipkin · a distributed tracing system. Accessible at https://zipkin.io/. Accessed in February 2022.

[81] Abram Perdanaputra and Achmad Imam Kistijantoro. Transparent tracing system on grpc based microservice applications running on kubernetes. *2020 7th International Conference on Advanced Informatics: Concepts, Theory and Applications, ICAICTA 2020*, pages 1–5, 9 2020.

[82] Alexandre Perez and Rui Abreu. A qualitative reasoning approach to spectrum-based fault localization. *Proceedings - International Conference on Software Engineering*, 2:372–373, 5 2018.

[83] Fabio Pina, Jaime Correia, Ricardo Filipe, Filipe Araujo, and Jorge Cardroom. Nonintrusive monitoring of microservice-based systems. *NCA 2018 - 2018 IEEE 17th International Symposium on Network Computing and Applications*, pages 1–8, 11 2018.

[84] Teerat Pitakrat, Dusan Okanovic, Andre Van Hoorn, and Lars Grunske. An architecture-aware approach to hierarchical online failure prediction. In *2016 12th International ACM SIGSOFT Conference on Quality of Software Architectures (QoSA)*, pages 60–69, 2016.

[85] Juan Qiu, Qingfeng Du, Kanglin Yin, Shuang-Li Zhang, and Chongshu Qian. A causality mining and knowledge graph based method of root cause diagnosis for performance anomaly in cloud applications. *Applied Sciences*, 10(6), 2020.

[86] RabbitMQ. Messaging that just works – rabbitmq. Accessible at https://www.rabbitmq.com. Accessed in June 2022.

[87] Chris Richardson. Monolithic architecture pattern. Accessible at https://microservices.io/patterns/monolithic.html. Accessed in December 2021.

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

[89] Matheus Santana, Adalberto Sampaio Jr, Marcos Andrade, and Nelson S Rosa. Transparent tracing of microservice-based applications. *Proceedings of the ACM Symposium on Applied Computing*, Part F147772:1252–1259, 4 2019.

[90] Huasong Shan, Yuan Chen, Haifeng Liu, Yunpeng Zhang, Xiao Xiao, Xiaofeng He, Min Li, and Wei Ding. e-diagnosis: Unsupervised and real-time diagnosis of small- window long-tail latency in large-scale microservice platforms. In *The World Wide Web Conference*, WWW '19, page 3215–3222. Association for Computing Machinery, 2019.

[91] 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.

[92] João Pedro Gomes Silva. Debugging microservices. Master's thesis, Universidade do Porto, Porto, September 2019. Accessible at https://hdl.handle.net/10216/122187.

[93] Jacopo Soldani and Antonio Brogi. Anomaly detection and failure root cause analysis in (micro) service-based cloud applications: A survey. *ACM Comput. Surv.*, 55(3), feb 2022.

[94] Solo. Squash. Accessible at https://squash.solo.io/. Accessed in January 2022.

[95] Diomidis Spinellis. Modern debugging: The art of finding a needle in a haystack. *Communications of the ACM*, 61:124–134, 11 2018.

[96] Spring. Springboot. Accessible at https://spring.io/projects/spring-boot. Accessed in December 2021.

[97] Spring. Springcloud. Accessible at https://spring.io/projects/spring-cloud. Accessed in December 2021.

[98] Jörg Thalheim, Antonio Rodrigues, Istemi Ekin Akkus, Pramod Bhatotia, Ruichuan Chen, Bimal Viswanath, Lei Jiao, and Christof Fetzer. Sieve: Actionable insights from monitored metrics in distributed systems. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*, Middleware '17, page 14–27, New York, NY, USA, 2017. Association for Computing Machinery.

[99] Lingzhi Wang, Nengwen Zhao, Junjie Chen, Pinnong Li, Wenchi Zhang, and Kaixin Sui. Root-cause metric location for microservice systems via log anomaly detection. In *2020 IEEE International Conference on Web Services (ICWS)*, pages 142–150, 2020.

[100] Ping Wang, Jingmin Xu, Meng Ma, Weilan Lin, Disheng Pan, Yuan Wang, and Pengfei Chen. Cloudranger: Root cause identification for cloud native systems. In *2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pages 492–502, 2018.

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

[102] Tianjun Weng, Wanqi Yang, Guangba Yu, Pengfei Chen, Jieqi Cui, and Chuanfu Zhang. Kmon: An in-kernel transparent monitoring system for microservice systems with ebpf. *Proceedings - 2021 IEEE/ACM International Workshop on Cloud Intelligence, CloudIntelligence 2021*, pages 25–30, 5 2021.

[103] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. Experimentation in software engineering. *Experimentation in Software Engineering*, 9783642290442:1–236, 7 2012.

[104] W. Eric Wong, Vidroha Debroy, Yihao Li, and Ruizhi Gao. Software fault localization using dstar (d*). *Proceedings of the 2012 IEEE 6th International Conference on Software Security and Reliability, SERE 2012*, pages 21–30, 2012.

[105] W. Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. A survey on software fault localization. *IEEE Transactions on Software Engineering*, 42:707–740, 8 2016.

[106] Franz Wotawa. Fault localization based on dynamic slicing and hitting-set computation. *Proceedings - International Conference on Quality Software*, pages 161–170, 2010.

[107] Li Wu, Jasmin Bogatinovski, Sasho Nedelkoski, Johan Tordsson, and Odej Kao. Performance diagnosis in cloud microservices using deep learning. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 12632 LNCS:85–96, 12 2020.

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

[109] Jian Xu, Zhenyu Zhang, W. K. Chan, T. H. Tse, and Shanping Li. A general noise-reduction framework for fault localization of java programs. *Information and Software Technology*, 55:880–896, 5 2013.

[110] Zihao Ye, Pengfei Chen, and Guangba Yu. T-rank: A lightweight spectrum based fault localization approach for microservice systems. *Proceedings - 21st IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing, CCGrid 2021*, pages 416–425, 5 2021.

[111] Xiuhuan Zang, Wei Chen, Jing Zou, Sheng Zhou, Huang Lisong, and Liang Ruigang. A fault diagnosis method for microservices based on multi-factor self-adaptive heartbeat detection algorithm. In *2018 2nd IEEE Conference on Energy Internet and Energy System Integration (EI2)*, pages 1–6, 2018.

[112] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chao Ji, Wenhai Li, and Dan Ding. Delta debugging microservice systems with parallel optimization. *IEEE Transactions on Services Computing*, 5 2019.

[113] 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*, 47:243–260, 2 2021.

[114] 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. *ESEC/FSE 2019 - Proceedings of the 2019 27th ACM Joint Meeting European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 19:683–694, 8 2019.

[115] Xiang Zhou, Jun Sun, Xin Peng, Wenhai Li, Dan Ding, Tao Xie, and Chao Ji. Delta debugging microservice systems. *ASE 2018 - Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 802–807, 9 2018.

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