KTH ROYAL INSTITUTE
OF TECHNOLOGY

Degree Project in Computer Science and Engineering

Second cycle, 30 credits

Simplifying Software Testing in Microservice Architectures through Service Dependency Graphs

MARCUS ALEVÄRN

Stockholm, Sweden, 2023

Simplifying Software Testing in Microservice Architectures through Service Dependency Graphs

MARCUS ALEVÄRN

Master's Programme, Computer Science, 120 credits
Date: July 5, 2023

Supervisors: Wafaa Mushtaq, Hannes Fornander
Examiner: Elena Troubitsyna
School of Electrical Engineering and Computer Science
Host company: Marginalen Bank
Swedish title: Förenkla mjukvarutestningen i mikrotjänstarkitekturer genom tjänsteberoendegrafer

Abstract

A popular architecture for developing large-scale systems is the microservice architecture, which is currently in use by companies such as Amazon, LinkedIn, and Uber. The are many benefits of the microservice architecture with respect to maintainability, resilience, and scalability. However, despite these benefits, the microservice architecture presents its own unique set of challenges, particularly related to software testing. Software testing is exacerbated in the microservice architecture due to its complexity and distributed nature. To mitigate this problem, this project work investigated the use of a graph-based visualization system to simplify the software testing process of microservice systems. More specifically, the role of the visualization system was to provide an analysis platform for identifying the root cause of failing test cases. The developed visualization system was evaluated in a usability test with 22 participants. Each participant was asked to use the visualization system to solve five tasks. The average participant could on average solve 70.9% of the five tasks correctly with an average effort rating of 3.5, on a scale from one to ten. The perceived average satisfaction of the visualization system was 8.0, also on a scale from one to ten. The project work concludes that graph-based visualization systems can simplify the process of identifying the root cause of failing test cases for at least five different error types. The visualization system is an effective analysis tool that enables users to follow communication flows and pinpoint problematic areas. However, the results also show that the visualization system cannot automatically identify the root cause of failing test cases. Manual analysis and an adequate understanding of the microservice system are still necessary.

Keywords

Microservice architecture, Service Dependency Graph, Software testing

Sammanfattning

En populär arkitektur för att utveckla storskaliga system är mikrotjänstarkitekturen, som för närvarande används av företag som Amazon, LinkedIn och Uber. Det finns många fördelar med mikrotjänstarkitekturen med avseende på underhållbarhet, motståndskraft och skalbarhet. Men trots dessa fördelar presenterar mikrotjänstarkitekturen sin egen unika uppsättning utmaningar, särskilt med hänsyn till mjukvarutestningen. Mjukvarutestningen försvåras i mikrotjänstarkitekturen på grund av dess komplexitet och distribuerade natur. För att mildra detta problem undersökte detta projektarbete användningen av ett grafbaserat visualiseringssystem för att förenkla mjukvarutestprocessen för mikrotjänstsystem. Mer specifikt var visualiseringssystemets roll att tillhandahålla en analysplattform för att identifiera grundorsaken till misslyckade testfall. Det utvecklade visualiseringssystemet utvärderades i ett användbarhetstest med 22 deltagare. Varje deltagare ombads att använda visualiseringssystemet för att lösa fem uppgifter. Den genomsnittliga deltagaren kunde i genomsnitt lösa 70.9% av de fem uppgifterna korrekt med ett genomsnittligt ansträngningsbetyg på 3.5, på en skala från ett till tio. Den upplevda genomsnittliga nöjdheten med visualiseringssystemet var 8.0, också på en skala från ett till tio. Projektarbetet drar slutsatsen att grafbaserade visualiseringssystem kan förenkla processen att identifiera grundorsaken till misslyckade testfall för minst fem olika feltyper. Visualiseringssystemet är ett effektivt analysverktyg som gör det möjligt för användare att följa kommunikationsflöden och peka ut problemområden. Men resultaten visar också att visualiseringssystemet inte automatiskt kan identifiera grundorsaken till misslyckade testfall. Manuell analys och en grundlig förståelse av mikrotjänstsystemet är fortfarande nödvändigt.

Nyckelord

Mikrotjänstarkitektur, Tjänsteberoendegraf, Mjukvarutestning

Acknowledgments

I would like to express my gratitude to KTH for giving me this invaluable opportunity. In addition, thanks to Marginalen Bank for hosting my master's thesis. A special thanks to my supervisors Hannes Fornander and Wafaa Mushtaq, who regularly provided me with guidance and feedback on my work.

Stockholm, July 2023
Marcus Alevärn

Contents

List of Figures

List of Tables

List of acronyms and abbreviations

API	Application Programming Interface
CD	Continuous Delivery
CI	Continuous Integration
CM	Continuous Monitoring
gRPC	Google Remote Procedure Call
HTTP	Hypertext Transfer Protocol
IoC	Inversion of Control
JSON	JavaScript Object Notation
QoS	Quality of Service
REST	Representational State Transfer
SDG	Service Dependency Graph
SOA	Service-Oriented Architecture
UAT	User Acceptance Testing
URL	Uniform Resource Locator

Chapter 1

Introduction

A popular architecture for developing large-scale systems is the microservice architecture, which is currently in use by companies such as Amazon, LinkedIn, Uber [1], and the Swedish bank Marginalen Bank. The microservice architecture breaks down complex software systems into self-contained services. Each service has a single responsibility and can be independently deployed, scaled, and tested [2]. This is a significant difference to the monolithic architectural style, where the entire application is developed as a single, unified entity. The microservice architecture facilitates maintainability, resilience, and scalability [3]. Despite these benefits, the microservice architecture presents its own unique set of challenges, particularly related to software testing [4, 5, 6, 7]. This thesis aims to investigate a graph-based approach to simplify and enhance the software testing process of a microservice system.

1.1 Background

Software testing is a necessity to achieve correct and reliable software, especially in today's world when many software systems are large and complex. The microservice architecture organizes an application as a distributed system, which is inherently more difficult to test than a monolithic system [5]. In the microservice architecture, services communicate with other services through a network, often using Representational State Transfer (REST) and a lightweight communication protocol such as Hypertext Transfer Protocol (HTTP) [8]. This establishes inter-dependencies among services, as some services rely on information or functionality provided by other services. Thus, testing each service in isolation is generally insufficient

for verifying the correctness of a microservice system. Consequently, development teams working with microservice systems often write integration and end-to-end tests that evaluate the behavior of multiple services together [6, 7]. Testing microservice systems can be challenging due to several factors, including interdependent services, cloud infrastructure, and external service dependencies [7].

This thesis is conducted on behalf of Marginalen Bank, a Swedish financial institution that is currently encountering challenges related to software testing. The bank maintains and develops a microservice system with more than 60 services written in `.NET`. To overcome the difficulties of software testing, this thesis aims to explore the possibilities and limitations of utilizing an Service Dependency Graph (SDG)-based method to simplify the software testing process of microservice systems. An SDG is a directed graph that visualizes inter-dependencies between services. Software testing large microservice systems can be challenging and time-consuming, making this research topic relevant to many organizations involved in microservice development.

1.2 Problem

Software testing is necessary to ensure that software functions as expected. This is especially important for software related to vital societal applications such as banking. As organizations switch from the traditional monolithic architecture to the microservice architecture, new and unique challenges arise, particularly with respect to software testing. The distributed and dynamic nature of a microservice system makes it more difficult to understand and debug failing test cases. Finding the root cause of a failing test case may require the investigation of logs or traces, which can be tedious work. The existing research on how useful an SDG-based method can be in identifying the root cause of failed test cases is limited. Therefore, this thesis aims to answer the research question of whether the use of an SDG-based method can simplify the software testing process of microservice systems.

1.3 Purpose

The purpose of this thesis is twofold. Firstly, Marginalen Bank would want to gain insight into how SDG-based methods can assist them in the software testing process of their microservice system. Secondly, there is only limited research on how SDG-based methods can simplify the software testing process

of microservice systems. Thus, the thesis should provide relevant up-to-date knowledge to the research community.

1.4 Goals

To satisfy both Marginalen Bank and the research community, the work has been divided into three concrete goals:

- *Proof-of-concept:* A proof-of-concept of an SDG-based method will be developed. The purpose of the proof-of-concept is to produce a general model that delineates a reasonable approach that highlights what is possible to achieve with an SDG-based method in relation to software testing microservice systems.
- *Prototype:* A prototype, based on the proof-of-concept, will be developed. The prototype is a realization of the proof-of-concept particularly built for being compatible with the architecture and frameworks used by Marginalen Bank. Consequently, the prototype is not a general model and will only work with services written in `.NET` and software tests that utilize the testing framework `NUnit`.
- *Evaluation:* To answer the research question the prototype is evaluated using an evaluation methodology known as *usability testing*. Usability testing involves the examination of the prototype by individuals who serve as representative users. Specifically, for this study, the representative users will consist of developers and quality assurance engineers from Marginalen Bank.

1.5 Research Methodology

The prototype was evaluated in a usability test by representative users from Marginalen Bank. In total, 22 participants took part in the study. Each participant was asked to use the prototype to solve five tasks. In each task, a test case had failed and the participant was asked to use the prototype to identify the root cause of the failed test. Five different types of errors that can occur in a microservice system were tested. The microservice system that was used in the usability test was fictional. The data collected from the study was the number of tasks correctly solved and the subjective effort required to solve each task. Lastly, the participants were asked if they perceived the prototype

as useful and if they believed that it could be useful in a real-world commercial microservice system.

1.6 Delimitations

A set of limitations was adopted to limit the scope of the project work and to make it feasible within the time constraints:

- The implementation will only work with services that are communicating over the HTTP protocol.
- Only five different error types that can occur in a microservice system were tested in the usability test.
- Only one prototype is developed from the proof-of-concept specifically designed to work for `.NET` and `NUnit`.

1.7 Structure of the Thesis

Chapter 2 presents relevant background information about the microservice architecture, software testing, and also related work. Chapter 3 describes the methods used in this project work. In particular, the usability test that was carried out. Chapter 4 describes the proof-of-concept developed in this project. Chapter 5 highlights the key features of the prototype. Chapter 6 presents the results. Chapter 7 presents a discussion. Lastly, chapter 8 presents conclusions and a proposal for future work.

Chapter 2

Background

This chapter covers the background information needed to understand the context of this project work. A brief overview of the HTTP protocol is given in section 2.1. See section 2.2 for details on the microservice architecture and its advantages, challenges, and common technologies and features. Furthermore, see section 2.3 for a brief discussion on software testing. In addition, see section 2.4 for an explanation of SDG. An important ordering for distributed systems known as causal order is presented in section 2.5. Information about OpenTelemetry is presented in section 2.6. Section 2.7 covers Marginalen Bank and section 2.8 explains usability testing. Lastly, see section 2.9 for related work.

2.1 HTTP

HTTP is an application-layer protocol that is widely used as a communication format for web clients and servers. In the context of microservice systems, the HTTP protocol can be used to allow services to expose their functionality through well-defined HTTP-based Application Programming Interfaces (APIs). The full technical specification of the HTTP protocol can be found in RFC 7231 [9].

There are two different types of HTTP messages, namely, requests and responses. An HTTP request must specify an HTTP method, such as GET, POST, or DELETE, which determines the desired action of the request. In addition, a Uniform Resource Locator (URL) to target a specific server resource. Both HTTP requests and responses can also include a set of headers and a body. A common data exchange format that is used to store data in the HTTP body is JavaScript Object Notation (JSON). A technical specification

of the JSON format can be read in RFC 8259 [10].

Unique to HTTP response messages is that they must contain a three-digit status code that indicates if the request could be fulfilled or not. A common status code is, for example, 200, which indicates that the request succeeded.

2.2 Microservices

The microservice architecture has evolved from the Service-Oriented Architecture (SOA). Both architectural styles share the fundamental principle that an application should be structured as a collection of loosely-coupled services rather than as a monolith. In addition, the idea that services should be organized around business capabilities [11]. However, the microservice architecture offers features more suitable for cloud computing. For example, each service in the microservice architecture should be independently deployable and scalable and not share its resources such as containers, caches, and databases with other software components [11]. Thus in the microservice architecture, a database can be seen as belonging to a service. The service restricts how the database can be accessed with the help of an API. See Figure 2.1 for a primitive high-level example of a banking application that is implemented as a microservice system with three separate services.

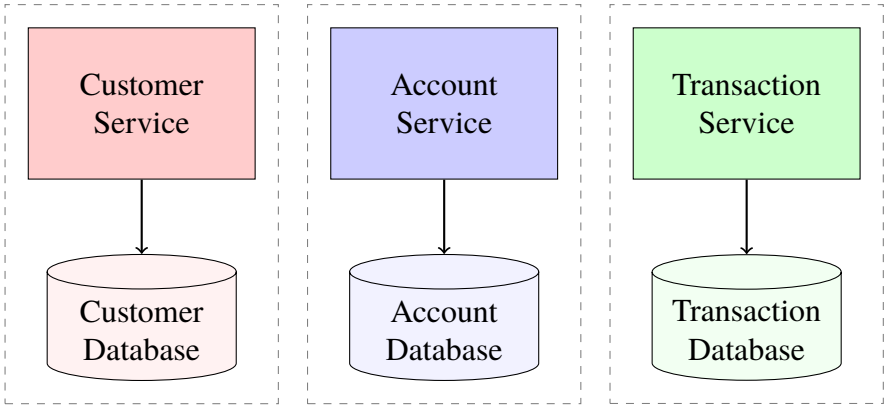

Figure 2.1: Example of a Microservice System

The first service in Figure 2.1 is the `Customer Service`, which is responsible for handling customer-related functionality such as managing customer data. The second service is the `Account Service`, which is responsible for account-related functionality such as creating and closing accounts and checking balances. Lastly, the `Transaction Service`

is responsible for transaction-related functionality such as depositing and withdrawing funds and issuing payments. Each service has a unique responsibility and maintains its own database. Thus, there is a distinct separation between services and their responsibilities, which allows for each service to be individually deployed, scaled, and tested. However, the services will need to communicate with each other in order to perform the tasks that one expects a banking system to be able to handle. For example, when a transaction is initiated, the `Transaction Service` would need to verify that the account from which the funds are being withdrawn is active and has sufficient funds. This information would need to be obtained from the `Account Service`. In addition, when a new account is created, there must also exist a customer record that the new account can belong to. Hence, there will need to be communication between `Account Service` and `Customer Service`. By defining an API for each service, such as a REST API based on HTTP, a communication protocol can be established between the services, enabling them to interact with one another.

2.2.1 Common Technologies and Features

Many different technologies are widely employed when developing microservice systems. Perhaps the most well-known technology is *Docker*, which is a virtualization platform that enables software to be packaged and executed inside containers [12]. Docker makes it possible to efficiently execute software in a variety of environments and has therefore gained a lot of popularity in the era of cloud computing. Another technology often used in conjunction with Docker is a container orchestration platform such as *Kubernetes* that can manage Docker containers and provide features such as load-balancing and automatic scaling. There are many different technologies out there that provide abstractions and functionality to microservice systems. However, some of the most widely known features that microservice systems are equipped with are circuit breaker, service discovery, and API gateway [13].

The circuit breaker is a failure-handling mechanism that avoids cascading effects in a microservice system. The circuit breaker ensures that if a service fails and becomes unresponsive then all the dependent services should still be responsive. This is implemented by quickly realizing that a service has failed and stopping the communication between dependent services and the failed service. Consequently, this reduces the load on the failed service and thus will provide better conditions for recovery. [13]

Another feature implemented in most microservice systems is a service

discovery. The idea behind a service discovery is to provide a standardized way for services to locate other services. In other words, make services addressable. This is relevant if the microservice system is hosted in the cloud, which can replicate and relocate services at runtime. In such a scenario, a service discovery will be needed to enable communication between services. [13]

Lastly, the API gateway is an entry point for the microservice system. It provides access to many APIs and be customized to fit the needs of different clients. For example, some clients may want to use an API that is less network intensive. Furthermore, the API gateway is often equipped with load balancing, service discovery, monitoring, and security. [13]

2.2.2 DevOps

DevOps is a collective term for all practices, processes, and tools that aim to streamline the software development lifecycle and reduce the time it takes to release software changes into production [14]. Common DevOps practices are Continuous Integration (CI), Continuous Delivery (CD), and Continuous Monitoring (CM). CI is the first step towards CD and enables developers to regularly merge their code changes into a central repository [14]. Each code change is then automatically built and tested to ensure that the new code does not break the existing system. CD is a practice that makes it possible to automatically deploy software to any environment. The idea is to continuously deliver software to a production-like environment, where the software can be tested, verified, and finally released to production when ready. Another common practice is CM which provides developers with performance metrics and can help detect operational anomalies [14].

DevOps is important to successfully develop, test and deploy microservice systems due to their highly distributed nature. A microservice system may involve many separate services, each of which must be independently developed, deployed, and managed. This can create challenges concerning the coordination between different teams to ensure that each service is tested and deployed properly. Effective DevOps practices can alleviate this burden and make the development lifecycle less cumbersome.

2.2.3 Advantages

Migrating from a monolithic architectural style to a microservice architecture presents both benefits and drawbacks. The first advantage is an increased

degree of maintainability because each service is independent with a single responsibility [1]. The separation into individual services that communicate over a network also avoids technology lock-in [14] as each service can be developed in a unique programming language and use a different type of data storage. The third benefit is scalability since each service can be individually scaled to quickly adapt to changes in demand [3]. A monolithic system does not bring this flexibility, and must often be scaled as a whole, which may be a more complex process and lead to more hardware usage [3]. The fourth benefit is resilience because the failure of one service does not necessarily impact the entire system [3]. Hence, parts of the system may function properly even though some services have failed. The fifth benefit is that the microservice architecture improves time to market by allowing for more continuous deliveries [1]. In addition, the microservice architecture promotes agile development by allowing agile teams to structure their work around services [1].

2.2.4 Challenges

Although the advantages of the microservice architecture are many, one should not think that the microservice architecture will automatically solve all software design problems. The microservice architecture, like any architecture, presents its own unique set of challenges. It can be difficult to structure a large application into a set of well-defined services, where each service handles a specific responsibility and provides a suitable interface [8]. If the design of the microservice system is not well-planned it can cause an increase in network communication between services [8], which will have a negative impact on the overall performance of the system. Over time as new features are added to the system and existing ones are modified, then design principles may break and the initial decomposition of the system may not be appropriate. This is known as architectural degradation, which can lead to the system being unfeasible to maintain [15].

Other common challenges in the microservice architecture are related to monitoring, deployment, testing, versioning, and deprecation [1]. A single microservice can be individually tested with ease. However, as the number of microservices and the number of connections between them grow, performing tests that consider more than one microservice can become a complex task [1]. Several different sources have highlighted software testing as a significant challenge in the microservice architecture [1, 4, 5, 6, 7]. A distributed system, such as a microservice system, is inherently more difficult

to test than a monolithic system [5]. The complexity of software testing is exacerbated by interdependent services, cloud infrastructure, and external service dependencies [7]. Consequently, different automatic testing tools [16, 17, 18, 19, 20] have been proposed to simplify and optimize software testing.

2.3 Software Testing

Software testing involves the activities that check the correctness, completeness, and accuracy of software [21]. A common software testing method is to dynamically verify that a software system produces the expected behavior on a finite set of test cases [22]. In the context of microservice systems, it is crucial that test automation is in place to regularly check that the system is working as expected. Test automation is often part of the CI/CD pipeline. According to Mike Cohn, there are three different levels that should be covered in test automation, he illustrates this in his *test automation pyramid* [23], see Figure 2.2.

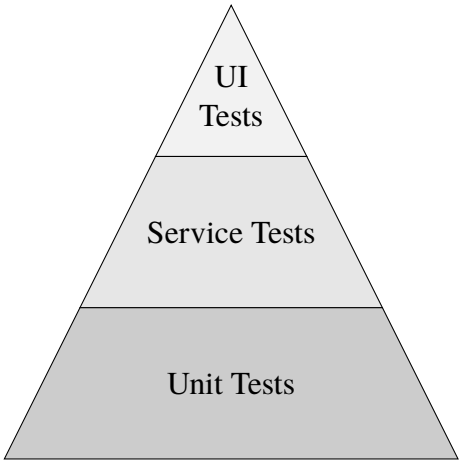

Figure 2.2: Mike Cohn's Test Automation Pyramid

The bottom layer in the pyramid involves unit tests, which are designed to test a single unit of code in isolation. Unit tests are easy to write, maintain, and quick to execute, as they only test one specific piece of functionality. The middle layer covers automated service tests for APIs and integration. These tests are designed to test the interactions between different units of code and the integration of those units into a larger system. Lastly, the top layer entails end-to-end tests that cover a complete system requirement. In general, as we

go up the layers in the pyramid, tests become harder to maintain, cover more and more units in combination, and take longer time to execute. [24]

2.4 Service Dependency Graph

An SDG is a directed graph that visualizes the dependencies among services in a microservice system. Thus, each node in an SDG represents a service and each directed edge indicates a dependency relationship between two services. A service that depends on another service makes at least one request to that service. More mathematically precise, an SDG can be defined as in equation 2.1.

$$SDG = (V, E) \tag{2.1}$$

Where V is a set of nodes and E is a set of ordered pairs. If $(u, v) \in E$ then there exists a directed edge between the nodes u and v.

See Figure 2.3 for a concrete example of an SDG that visualizes the dependencies between four different services.

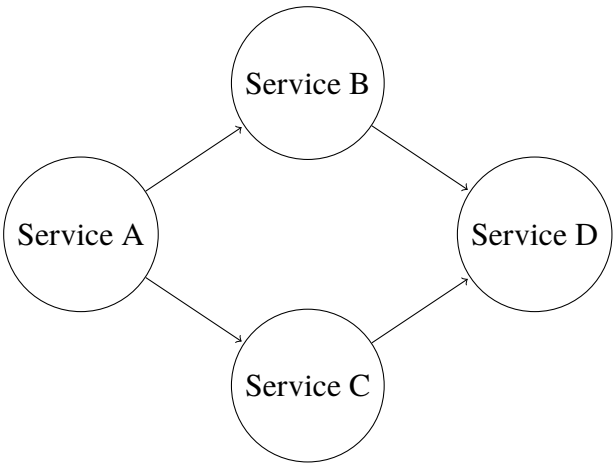

Figure 2.3: Example of a Service Dependency Graph

From Figure 2.3 we can see that `Service D` is independent as no edges are originating from it. In contrast, `Service B` and `Service C` are both dependent on `Service D`. Lastly, `Service A` is directly dependent on both `Service B` and `Service C`, but also indirectly dependent on `Service D`.

The purpose of an SDG is to provide an analysis tool and visual reconstruction of the microservice system to assist developers in identifying problems such as anti-patterns, architectural degradation, and anomalies.

There are three common approaches to constructing an SDG: static analysis, dynamic analysis, and manual analysis. In static analysis, the SDG is constructed from artifacts available before deployment. Such artifacts could be source code, configuration files, and documentation. The benefit of static analysis is that it can be performed before actual deployment, which is not the case for dynamic analysis. In dynamic analysis, a broad variety of different data sources can be used to create the SDG such as runtime traces, logs, and network traffic. Lastly, manual analysis consists of manually constructing the SDG by letting a human analyze the code, which can be an important step in validating the results. However, it is not an automated process. Thus, static analysis and dynamic analysis are two feasible options in terms of maintaining a high degree of automation while manual analysis is not. [25]

2.5 Causal Order

An important topic in distributed systems is how to track the ordering of events. It is not as simple as using clocks due to inherent problems such as clock drift and clock skew. Causal order is concerned with how the occurrence of certain events may affect other events in the future [26]. An event is said to causally precede another event if it occurs before the other event and can be considered a cause of that event. For example, when a service initiates a request to another service, the act of sending the request causally precedes the corresponding response that the receiving service will generate and send in reply.

Causal order is a type of partial order, which means that it satisfies the properties of reflexivity, transitivity, and antisymmetry. Not all events in a causal order can be compared pairwise, meaning that some events may be considered concurrent or independent of each other. Thus, a causal order is not a total order because a total order ensures that all elements are pairwise comparable.

2.6 OpenTelemetry

OpenTelemetry is a vendor-agnostic framework for collecting telemetry data such as traces, metrics, and logs [27]. The goal is to enable observability in distributed systems. OpenTelemetry is an open standard and has support for

instrumentation in several programming languages such as C#, Java, and C++, to name a few. In addition, OpenTelemetry has support for both automatic and manual instrumentation. In automatic instrumentation, telemetry data is automatically captured from popular libraries during runtime. This telemetry data could be HTTP requests, HTTP responses, database calls, and so on. The difference between automatic and manual instrumentation is that in automatic instrumentation the developer is not required to modify the application's source code to start capturing telemetry data. Tracing in OpenTelemetry is achieved by collecting objects, known as *spans*. A span represents a unit of work and contains the following information:

- *Operation name*
- *Parent span id (empty for root spans)*
- *Start and end timestamps*
- *Span context*: Contains information needed to reference spans, such as span id and trace id
- *Attributes*: A list of key-value pairs.
- *Span events*: A set of zero or more events, each event has a name, timestamp, and attribute list
- *Span links*: A set of links to zero or more causally-related spans
- *Span status*: Can be used to indicate an error or normal operation

Spans that share the same trace id belong to the same trace. A trace can be seen as a set of events that were triggered by a single logical operation. The single logical operation can for example be an HTTP GET request to a service in a microservice system. Important to note is that a causal relationship between spans can be established with the help of the parent span id or span links.

2.7 Marginalen Bank

Marginalen Bank is a Swedish bank that maintains and develops a microservice system with more than 60 services written in `.NET`. The bank has a development team and a quality assurance team. The development team is responsible for developing the microservice system and writing unit tests

to verify that individual functionality works correctly. The quality assurance team implements end-to-end tests that verify that the microservice system satisfies all business requirements. To ease development, Marginalen Bank utilizes a continuous delivery pipeline consisting of three stages: test, User Acceptance Testing (UAT), and production. All the developed code gets built in the test environment and unit tests are executed. In the UAT environment, end-to-end tests are scheduled regularly every night. The last environment is the production environment, which maintains the microservice system that is commercially deployed.

2.8 Usability Testing

In usability testing, a product is evaluated with the help of representative users. Each user is asked to use the product to complete a set of tasks while the product owner watches, listens, measures time, and takes notes. The goal of usability testing is to measure how well the system fulfills effectiveness, efficiency, and subjective user satisfaction [28]. Effectiveness is how well the user can utilize the system to solve the tasks at hand. Moreover, efficiency is how much effort and/or time the user must spend to solve the problems using the system. Lastly, user satisfaction is how subjectively useful the user finds the system and the general attitude towards the system. The motivation for using usability testing is that it is considered a reliable way to estimate users' performance and subjective satisfaction with a product [29]. It is often used to evaluate interactive systems such as web and mobile applications [29, 28]. The number of participants that is needed in a usability test to get accurate results depends on several factors. However, previous work [30], has developed a mathematical model to estimate the number of participants i required to discover a specific number of usability problems that exist in a product, see equation 2.2.

$$\text{Discovered}(i) = N(1 - (1 - \lambda)^i) \tag{2.2}$$

Where N is the total number of usability problems that exist in the product and λ is the probability of finding the average usability problem when doing the usability test with a single average participant. Both N and λ depend on the product and must be estimated as per product basis. However, the authors in [30] computed the mean values of N and λ from 11 studies and got the following mean values $N = 41$ and $\lambda = 0.31$ which can act as a very approximate rule of thumb. Using these mean values we get the graph that can

be seen in Figure 2.4. Observing the graph we can see that more than 80% of all usability problems can be discovered with only 5 participants. As already noted, this should only be seen as a very approximate rule of thumb and the actual number of participants needed will vary depending on the project.

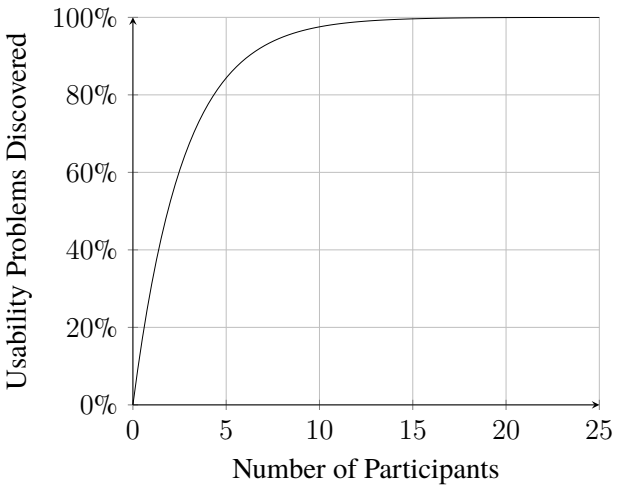

Figure 2.4: Percentage of Usability Problems Discovered Depending on the Number of Participants Using the Mean Values $N = 41$ and $\lambda = 0.31$

2.9 Related Work

See section 2.9.1 and section 2.9.2 for two microservice problems that have been solved with graph-based methods. These solutions highlight the utility of graph-based methods to solve problems related to the microservice architecture. Similarly, this project work will also focus on solving a problem related to the microservice architecture, and also use a graph-based approach to do so. However, this project work will focus on the specific problem of software testing, which is different from the two problems mentioned in section 2.9.1 and section 2.9.2. Lastly, see section 2.9.3 for previous work on software testing the microservice architecture, which is similar to the problem this thesis is addressing.

2.9.1 Composition and Decomposition

Composition algorithms in the context of distributed systems are about combining services to achieve a certain goal or task. Several service composition algorithms use graph-based methods to achieve their goals.

For example, [31] uses an SDG-based algorithm to find the k-top service compositions that are optimal with respect to Quality of Service (QoS). Another example is [32], which uses a graph-based method to find a service composition that offers a feature not provided by any individual service alone.

The opposite of composition is decomposition. The goal of decomposition algorithms is to break down a monolithic application into smaller, independent services. This can be done by constructing a dependency graph from the source code and applying clustering techniques and evaluation metrics to identify individual services [33, 34].

2.9.2 Design Errors

Design errors and anti-patterns in microservice systems have also caught the interest of the research community. A cyclic dependency is an example of an anti-pattern, which occurs when two or more services depend on each other in a circular manner. Anti-patterns are considered bad design practices and should be avoided to maintain the quality of microservice systems. Several sources have constructed graph-based models from microservice systems and applied graph algorithms to identify anti-patterns [19, 20, 35, 36, 37]. Both [35, 36] study degree centrality and clustering coefficient as possible graph metrics to discover design problems. For example, degree centrality can be used as an indication to locate bottleneck services [36]. Furthermore, cyclic dependencies can automatically be identified by constructing an SDG from the microservice system and applying Tarjan's Strongly Connected Component algorithm [20].

2.9.3 Software Testing

Two graph-based methods to simplify and optimize software testing has been identified [19, 20]. Both papers propose a model that takes the source code of microservices as input and produces a complete SDG over the microservice system. To build the SDG the authors employed the built-in reflection library in the Java language. The proposed model in the first paper [19] calculates test coverage and visualizes test results directly in the SDG. The test coverage metric can assist developers in understanding the quality of the developed system. Furthermore, the test result visualization in the SDG makes it possible to see the complete path each test has taken through the system. This visualization can assist developers in finding the root error of a failing test case.

The model in the second paper [20] adds functionality related to regression testing. For example, the model can retrieve only the necessary tests to rerun due to a code change, which allows for regression test optimization. The order of the test to rerun is decided based on a priority-based algorithm. This strategy shows a significant reduction in the number of tests that need a rerun.

There are some similarities and differences between this project work and the work presented in [19, 20]. This project work is based on the idea to visualize test results in a graph, which is also covered in the mentioned two papers. However, this project work will provide more information in the visualization and therefore provide a more comprehensive tool for identifying the root cause of failing test cases. Furthermore, the proposed solution in this project work does not build the entire SDG using reflection, instead utilizes dynamic analysis to build a partially complete SDG as each test case is executed. This project work takes a different approach because it is not always necessary to visualize a complete SDG to debug a single test case.

Another paper [16] proposes a method for simplifying regression testing by integrating it into continuous delivery steps. Hence, this method does not rely on an SDG. Instead, the idea is to compare the input-output pairs of the production environment compared to the development environment where we want to perform regression testing. The work is not graph-based and is therefore different from the work covered in this project work.

Work has also been put on automatic generation of test cases based on formal microservice specifications [17]. In addition, architectures to make test frameworks easier to maintain and reusable [18]. These topics are different from the topic of this project work. This project work is about providing a graph-based visualization tool for debugging and analyzing test cases and not about automatic test generation or how to design the infrastructure of test frameworks.

Chapter 3

Methods

This thesis aims to answer the research question of whether the use of an SDG-based method can simplify the software testing process of a microservice system. See section 3.1 for the research process utilized to answer the research question. See section 3.2 for more information on the selected evaluation methodology.

3.1 Research Process

1. *Literature study:* The essential findings of the literature study is covered in chapter 2. The literature study should provide enough background knowledge to be able to understand the difficulties of inter-dependencies and software testing microservice systems. Furthermore, different approaches that have been taken to solve similar problems related to the microservice system. The literature study should provide enough background knowledge to make it possible to develop the proof-of-concept, which should be a general model that tackles the problem at hand.

2. *Proof-of-concept:* The proof-of-concept should be based on the findings of the literature study. The proof-of-concept is covered in chapter 4. The proof-of-concept should be a general model that is platform-independent and later used as a basis for constructing the prototype. The role of the proof-of-concept is to answer general design problems, such as how the SDG should be built using one of the methods discussed in Section 2.4. The proof-of-concept should highlight features and functionality that the system should have. The reason for developing

the proof-of-concept before the prototype is to increase the likelihood of thinking through the functionality required by the system before actually implementing it. Thus, saving time that may have been wasted in the development of faulty, unnecessary, or difficult functionality. Another purpose of the proof-of-concept is to illustrate a general model that solves the problem. This is different from the prototype, which is bounded to the technology and architecture of Marginalen Bank.

3. *Prototype:* Based on the proof-of-concept the prototype is developed. The prototype is covered in chapter 5. The prototype is an actual implementation of the proof-of-concept and is tailored to the technology and architecture of Marginalen Bank. Thus, the prototype is less general compared to the proof-of-concept but will instead showcase that the proof-of-concept can be implemented and make it possible to evaluate the system in the usability test.

4. *Prototype evaluation:* The prototype was evaluated in a usability test with developers and quality assurance engineers from Marginalen Bank. See section 3.2 for more information on how the usability test was configured and carried out.

3.2 Usability Testing

To answer the research question a usability test was conducted. In total, 22 participants took part in the usability test. Out of these 22, 18 worked with development and the remaining 4 worked with quality assurance. In the usability test, both quantitative and qualitative data were gathered to measure how well the prototype fulfills effectiveness, efficiency, and user satisfaction.

3.2.1 Data Collection

A similar project conducted usability testing remotely by hosting their web application and sending out a questionnaire to users to enable the collection of both quantitative and qualitative data [38]. This project work also collected data remotely by hosting the web application and sending out a questionnaire. The questionnaire was created in Google Forms. Two different user groups from Marginalen Bank, namely developers and quality assurance engineers participated in the usability test. In total, 18 developers and 4 quality assurance engineers took part in the study. Each participant was asked to use the

prototype to solve five tasks while answering questions in the questionnaire. Thus, all data were collected remotely via Google Forms.

3.2.2 Questionnaire

The questionnaire was structured into three distinct sections. The first section explained the purpose of the prototype and highlighted its main features. Information on the fictional microservice system that is used in these five tasks was also given. The second section of the questionnaire consisted of five tasks. Each task had a brief problem statement and two questions. The first question was a single-choice question that asked the respondent to select the correct alternative from a list of five alternatives in total. The purpose of this question was to measure effectiveness, i.e. how well the user can utilize the system to solve the tasks at hand. The second question asked the user to rate the level of effort on a scale from one to ten that was required to solve the task with the prototype. The purpose of this question was to measure efficiency, i.e. how much effort the user must spend to solve the task using the system.

The last section of the questionnaire collected quantitative data and also qualitative data in the form of free text. If the user had entered a low or high level of effort on any task, then the user was asked to motivate why it was easy or hard to solve that task using the prototype. Furthermore, the user was asked to rate the satisfaction of using the prototype on a scale from one to ten and motivate the answer in free text. In addition, answer a yes or no question if the user believed that the prototype could be useful in a real-world microservice system and motivate that answer in free text.

3.2.3 Fictional Microservice System

All five tasks were performed on the same fictional microservice system that was developed to be used in the five tasks. The fictional microservice system was for a made-up consultant firm that offered consulting services in three different domains: it, legal, and financial. Each domain had at least one service that maintained information about the completed projects within that domain. There was also another dedicated service to summarize the income from all completed projects. See Table 3.1 for the name of all services and a description of their endpoints.

Service	**Endpoints**
SoftwareConsultingService	*/GetCompletedProjects* returns a JSON array with all the completed **software** projects
HardwareConsultingService	*/GetCompletedProjects* returns a JSON array with all the completed **hardware** projects.
ITConsultingService	*/GetCompletedProjects* returns a JSON array with all the completed **software and hardware** projects
LegalConsultingService	*/GetCompletedProjects* returns a JSON array with all the completed **legal** projects
AccountingConsultingService	*/GetCompletedProjects* returns a JSON array with all the completed **accounting** projects
InvestmentConsultingService	*/GetCompletedProjects* returns a JSON array with all the completed **investment** projects
FinancialConsultingService	*/GetCompletedProjects* returns a JSON array with all the completed **accounting and investment** projects
SummarizeIncomeService	*/GetTotalIncome* returns a JSON object with the total income from all completed projects.

Table 3.1: The Fictional Microservice System with Its Endpoints

The services from Table 3.1 are depending on each other as illustrated by the SDG in Figure 3.1. ITConsultingSevice invokes SoftwareConsultingService and HardwareConsultingService and aggregates their completed projects into one list and returns that to SummarizeIncomeService. The same principle is used again in FinancialConsultingSevice which invokes AccountingConsultingService and InvestmentConsultingService and aggregates their completed projects into one list and returns that to SummarizeIncomeService.

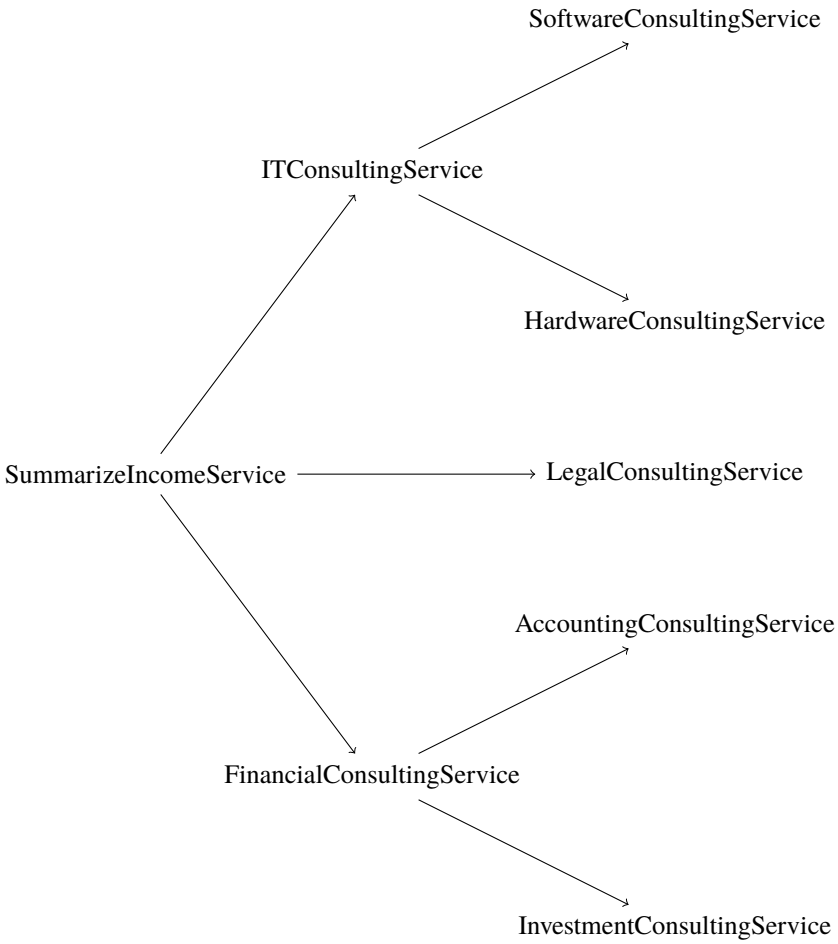

Figure 3.1: Service Dependency Graph of the Fictional Microservice System

The reason why ITConsultingService and FinancialConsultingService were added to this fictional microservice system was to increase the length of the longest path in the SDG. In Figure 3.1 it is clear that the longest path is of length two. It is desirable to have some depth in the SDG that is used in the usability test. This is due to the fact that in real microservice systems, the invocation chains can be long and deep and therefore it would not be reasonable to have an SDG that is too shallow in the usability test.

3.2.4 Tasks

The goal of each task was for the participant to identify the root cause of a failing test case. The same fictional microservice system described in the previous section was used in all five tasks. All tests were written for

SummarizeIncomeService. In each task, a unique error or bug was introduced in any of the eight services listed in Table 3.1. The following errors were tested in the five tasks:

1. **Service down:** InvestmentConsultingService is down and unresponsive, which results in FinancialConsultingService and SummarizeIncomeService returning an HTTP status code of 500.

2. **Misspelled URL:** SummarizeIncomeService had misspelled the URL that was used to get the completed projects from FinancialConsultingService. This resulted in FinancialConsultingService returning a status code of 404 and SummarizeIncomeService returning a status code of 500.

3. **Invalid response data:** HardwareConsultingService returned an invalid response body with a value of type string when ITConsultingService expected that value to be of type number. Thus, both ITConsultingService and SummarizeIncomeService returned a status code of 500.

4. **Bad request data and missing input validation:** SummarizeIncomeService did not properly validate a query parameter that had an invalid value which resulted in the parameter being passed down to ITConsultingService and then down to SoftwareConsultingService. Improper or missing input validation is not only a bug it can also be an attack vector for adversaries known as an *injection attack* [39].

5. **Missing response data:** LegalConsultingService returned zero completed projects, which made SummarizeIncomeService compute the wrong expected total income. This can happen due to several reasons, there might have been problems with the underlying database yielding an empty response, for instance.

3.2.5 Data Analysis

Both quantitative and qualitative data were collected from the usability test. The data analysis focused on measuring the effectiveness, efficiency, and user satisfaction of the prototype. For each task, the number of correctly solved tasks, as well as popular incorrect alternatives that were selected by many participants, were analyzed.

Furthermore, the amount of effort required to solve each task according to the participants was also analyzed. Each participant entered an effort rating on a scale from one and ten. This data was visualized in a bar chart and the average, median, and variance were also computed. In the last part of the questionnaire the participant could motivate why they had entered a low or high effort rating, these answers were also analyzed to see common patterns of what features of the prototype reduced the effort and what features or lack of features increased the effort.

The last section of the questionnaire aimed to assess the perceived user satisfaction and usefulness of the prototype in a real-world microservice system. Participants were asked to rate the usefulness on a scale from one to ten and were encouraged to provide their reasons in free text. The number of participants who believed the prototype could be useful in a real-world microservice system, along with their motivations, were analyzed to see common and similar reasons. Additionally, any criticisms or missing features brought up by participants that would hinder the prototype's usefulness in a real-world microservice system were also analyzed.

Chapter 4

Proof-of-concept

This proof-of-concept is a general solution for microservice systems that enhances the debugging experience of end-to-end tests that cover more than a single service. The aim is to provide a visualization based on an SDG to enable debuggers and developers to locate the cause of failing test cases. In addition, verify that the execution of each test case generates the correct communication between services. Thus, the goal is to visualize the SDG and also make it possible to inspect all HTTP request-response pairs that are being sent between services. For simplicity, the proof-of-concept will be limited to only work for HTTP-based communication.

This proof-of-concept aims to tackle two significant challenges that impede the software testing process for microservice systems, namely, inter-service dependencies and third-party services [7]. For a microservice system to work properly, the communication between all internal services and third-party services must be correct. The proof-of-concept will provide a visualization platform to visualize the communication between services to assist in finding incorrect communication. The proof-of-concept will be based on the open standard OpenTelemetry which enables tracing in distributed systems.

4.1 Architecture

The overall architecture of the proof-of-concept is illustrated in Figure 4.1. The test client will run a test case by sending at least one request to the microservice system. As the test case is executed spans will be generated by the test client and the microservice system. In total, three different span types can be generated and collected as discussed in section 4.2. To

enable the generation of spans, both the test client and all the services in the microservice system must have activated OpenTelemetry instrumentation which is discussed in section 4.3. This will enable the generation of spans that can be transmitted to the collector over the Google Remote Procedure Call (gRPC) protocol. The collector retrieves all these spans and stores them. On-demand, the collector can build the SDG for a specific test case using the algorithm explained in section 4.4. The visualization system can retrieve the SDG from the collector by making a certain HTTP request. When the visualization system has received the graph from the collector, then it can draw the graph on the screen and allow the user to interact with it as discussed in section 4.5.

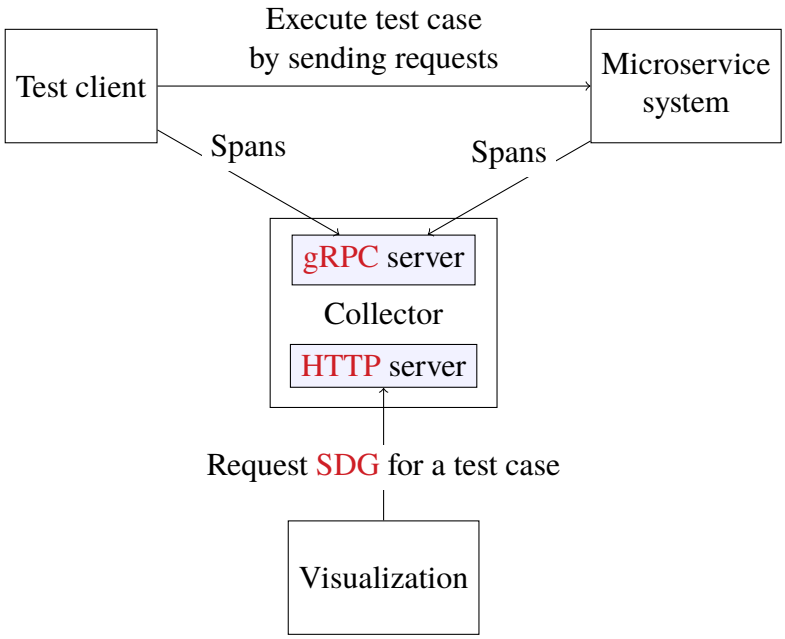

Figure 4.1: Overview of the Proof-of-concept Architecture

4.2 Span Types

In this work, three different span types are used. These three span types are distinguished from one another with the help of the *operational name*, see section 2.6 for a reminder of the span format. It does not really matter what specific operational name that is assigned to each span type as long as they are different. However, the following naming convention will be used in the report when referring to the three span types:

- *Client span*: Is generated every time an HTTP request is sent.
- *Server span*: Is generated every time an HTTP response is sent.
- *Test span*: Is generated every time a test case is executed in the test client.

Depending on the span type, different type-specific information may be stored in the span. For instance, in client and server spans, HTTP headers and bodies will be stored. Furthermore, in test spans, information on the name of the test case and if it passed or failed will be stored.

4.3 Instrumentation

To enable the generation of spans instrumentation must be activated. Instrumentation should be activated in all services in the microservice system as discussed in section 4.3.1. In addition, instrumentation should be enabled in the test client as well, which is explained in section 4.3.2.

4.3.1 Services

The first step is to enable instrumentation among all services in the microservice system. The request and response between all services need to be captured. This can be achieved with automatic instrumentation libraries for OpenTelemetry that automatically generates a span every time an HTTP request or response is sent. Thus, using automatic instrumentation libraries it should be possible to generate both client spans and server spans as listed in section 4.2.

By default, automatic instrumentation libraries do not necessarily capture the HTTP header and body that need to be stored in client and server spans. However, the OpenTelemtetry format has support for adding additional fields as key-value pairs that can be stored under the *attribute* section of the span format. Thus, it is possible to add custom key-value pairs such as *request_body*, *request_headers*, *response_headers*, and *response_body*. Then modify the source code of the automatic instrumentation libraries to ensure that these custom key-value pairs are included when the span is generated. Another more straightforward approach is to check the API of these automatic instrumentation libraries. For example, it may be possible to pass a callback function with some custom code that is called every time the span is about to

be generated. This could enable the retrieval of HTTP headers and bodies and the insertion of them as key-value pairs in the span.

4.3.2 Test client

The second step is to enable instrumentation in the test client that initiates the end-to-end test. The test client will send requests to the microservice system. Thus, the test client should be able to generate client spans that capture information about the outgoing requests. This is enabled with automatic instrumentation libraries as described previously. Furthermore, every time the test client runs a test case a test span should be generated that contains the name of the test and if the test passed or not. If no automatic instrumentation library exists for the test framework that is used, then it is possible to enable this with manual instrumentation in OpenTelemetry. This requires more work as this must be integrated with the testing framework that is used. The name of the test case and if the test passed or not shall also be added as key-value pairs under the attribute section of the span format.

4.4 SDG Generation

The actual generation of the SDG is performed in the collector in Figure 4.1. The collector stores client, server, and test spans. For every executed test case, the collector should store a unique test span. Thus, the span id of the test span can be used to uniquely identify every test case. Furthermore, since the test span is the first span that is generated in every test case, then it will become the root span that casually precedes all other client and server spans that are generated during the execution of the test case. Consequently, this will create a distributed trace where the test span is the root span of the trace. Hence, by looking at the trace id of the test span it is possible to filter out all client and server spans that belong to the same test case from all other spans that do not. This filter mechanism will ensure that the proof-of-concept still works even though test cases may be executed in parallel or other clients may be interacting with the microservice system at the same time.

See algorithm 1 for the pseudocode of the SDG generation algorithm. The algorithm takes a list of spans of all three types and the span id of the test span for the test case to build the SDG for. The algorithm returns the SDG as an ordered pair (V, E) as defined in equation 2.1.

Algorithm 1 Pseudocode for generating the SDG given a list of $spans$ and the span id of the test span that corresponds to a test case.

```
1:  function GENERATESDG(spans, id)
2:      V ← {}
3:      E ← {}
4:      t_span ← find the test span with span id equal to id
5:      c_spans ← find all client spans with trace id equal to t_span.traceId
6:      s_spans ← find all server spans with trace id equal to t_span.traceId
7:      for c_span ∈ c_spans do
8:          from = c_span.serviceName
9:          match ← find the span in s_spans with parent id equal to c_span.id
10:         if match exists then
11:             to ← match.serviceName          ▷ An internal service responded
12:         else
13:             to ← c_span.peerName            ▷ Third-party or unresponsive service
14:         V ← V ∪ {from, to}
15:         E ← E ∪ {(from, to)}
16:     return (V, E)
```

The lines 2-3 in algorithm 1 initialize an empty graph. The lines 4-6 find the test span, all client spans, and all server spans, respectively that belong to the test case to build the SDG for. On line 7, a loop starts iterating over all client spans in the test case. These spans correspond to all outgoing requests from services in the microservice system. The name of the service that made the outgoing request is stored in the $from$ variable on line 8. Then on line 9, an attempt is made to look up the matching server span that was generated in the service that got the request and sent back the response. The matching server span is found if its parent id is equal to the span id of the client span c_{span}. On lines 10-11, a check is made if the server span $match$ exists and if so the name of the service that responded to the request is stored in the variable to. On the other hand, on lines 12-13, there existed no matching server span for the client span. Thus, it may be the case that the response came from a third-party service that does not provide the collector with any spans, or it could be the case that the service that should have gotten the request is down and unresponsive and therefore did not provide the collector with any spans. In either case, the peer name of the client span is stored in the to variable. The peer name is the domain part of the URL that the request was sent to. The last lines of the loop 14-15 add the two nodes $from$ and to to the set of nodes V, and also add the edge $(from, to)$ to the set of directed edges E.

Note that an actual implementation should contain some more details

compared to the pseudocode. For example, there is no error handling in the pseudocode for the case if the test span does not exist, which may happen if the provided id is invalid. In addition, in each edge, it is insufficient to only store the name of the two services. Additional information should be stored together with the edge such as the request-response pairs that was generated between the two services. This information already exists in the client span and the matching server span. Consequently, this information can be extracted from the two spans and stored together with the directed edge.

4.5 Visualization

By collecting spans it is possible to generate the SDG for every test case. However, the SDG still needs to be visualized in an understandable manner and provide enough information to be useful. How the graph should be drawn to be graspable is explained in section 4.5.1. Furthermore, two novel features added to the visualization system of the SDG to possibly make the SDG more straightforward to work with are presented in section 4.5.2 and section 4.5.3.

4.5.1 Graph Drawing

The visualization system should be an interactive application that the user can use to see the SDG for a test case and inspect the communication between services. The SDG should be drawn on a 2D surface in such a way that it increases readability and avoids clutter to allow the user to follow dependencies among services. For SDGs that have a clear hierarchical structure, like the SDG in Figure 3.1, a hierarchical graph drawing algorithm like the Sugiyama Algorithm is suitable. The Sugiyama Algorithm draws nodes on parallel lines without overlapping, each line represents a level in the hierarchy, and all edges point in the same direction [40]. Thus, the algorithm enhances the understandability of the graph's flow and the relationships between nodes that belong to the same hierarchy.

In this proof-of-concept, an assumption is made that there is a clear hierarchical structure in the SDGs and therefore a single hierarchical graph drawing algorithm will be used. However, for SDGs that lack hierarchical structure other graph drawing algorithms such as the force-directed graph drawing algorithms might be a better option.

4.5.2 Clickable Edges

In the visualization system of the SDG, all edges should be clickable with the mouse. When a user clicks on a specific edge, a list containing all the HTTP request-response pairs exchanged between the two services connected by that edge should be displayed on the screen. This interactive feature serves the purpose of enabling users to quickly examine the communication between two services that are involved in the test. The user should also be able to inspect each request-response pair and see the information in that request and response, such as the HTTP method, target URL, status code, request and response headers and bodies, to name a few.

An alternative solution would be to provide a comprehensive list with all HTTP request-response pairs that are generated between any services during the test. However, that list could be extensive and the user may find it difficult to find the request-response pairs between two services in a long list. Of course, a hybrid approach is also possible where the visualization system has support for both clickable edges and can show a full list with all the HTTP request-response pairs captured during the test. However, for simplicity, in this proof-of-concept, there will only be support for clickable edges.

4.5.3 Edge Coloring

Up until now, the SDG visualizes all dependencies between services that were captured during the execution of a test. However, the user is still required to manually search the graph to identify where the root cause of the problem might be. To remedy this problem and streamline the debugging process, the visualization system will incorporate colored edges in the graph to indicate problematic areas. An edge in the SDG that contains at least one HTTP request-response pair with a status code of 400 or above will be colored red. This should provide users with visual cues and quickly draw their attention to areas where issues may have occurred during the test. HTTP codes of 400 or above are reserved for client and server errors. Consequently, it makes sense to highlight edges that contain HTTP status codes of 400 or above.

However, note that this is not always the case that an HTTP status code of 400 or above indicates a problem. For example, in negative tests that check that the software behaves correctly when provided with invalid or unexpected inputs, then a status code of 400 or above can be the expected and correct response code in a test. For positive tests, however, then a status code of 400 or above is more likely to indicate something wrong as positive tests aim to test the expected or intended behavior of the system under valid conditions and

inputs, and then error codes should not be returned.

A more elaborate solution might use edge coloring during positive tests and turn off this feature during negative tests. However, that requires the system to be aware of whether the test is positive or negative which can be difficult to identify. Thus, for simplicity, the proof-of-concept will stick with always having this feature on, even though the system is visualizing a negative test.

Chapter 5

Prototype

The prototype is a realization of the proof-of-concept tailor-made to be compatible with the architecture and technology stack in use by Marginalen Bank. See section 5.1 for how instrumentation is enabled in each individual service and test client. Furthermore, see section 5.2 for a brief overview of how the collector was implemented. Lastly, see section 5.3 for the implementation of the visualization system.

5.1 Instrumentation

In the prototype, OpenTelemetry instrumentation is realized using two different approaches depending on if it is enabled in a service or a test client. In both cases, instrumentation is activated with only a small amount of code modification, which makes the prototype feasible to implement on a large system. See section 5.1.1 and 5.1.2 for how instrumentation was activated in the services and test client, respectively.

5.1.1 Services

In Marginalen Bank, each service is implemented using `ASP.NET`. A software design pattern called dependency injection is widely used in `ASP.NET`. Dependency injection achieves Inversion of Control (IoC) by relying on a dependency injection container. The dependency injection container is responsible for creating instances of dependencies and injecting them into the constructors of components that require them [41]. Thus, this removes the need for each component that requires the dependency to create an instance of it.

To activate OpenTelemetry instrumentation in each service the dependency injection container was used. See Figure 5.1, for an example of the code modification that needs to be done to each service to activate instrumentation. A custom dependency is added to the dependency injection container `services`. Some additional information should also be supplied, such as the service name (e.g., ”Service A”) and the endpoint of the collector (e.g., ”https://collector.net”). The service name that is supplied here will be the name that is shown in the visualization system. Rather than hardcoding this information directly in the code as it is done in Figure 5.1, it's recommended to store it in a configuration file like *appsettings.json*.

```
services.AddOpenTelemetryExtension(new OpenTelemetryExtensionOptions {
    Name = "Service A",
    CollectorEndpoint = "https://collector.net"
});
```

Figure 5.1: Enable Instrumentation in Each Service with Dependency Injection

To summarize, to enable instrumentation in all services in the microservice system, one line of code needs to be added per service.

5.1.2 Test Client

All end-to-end tests in Marginalen Bank are written in `.NET` using the `NUnit` testing framework. Thus, instrumentation was activated by providing each test class with the `[OpenTelemetryExtension]` attribute, see Figure 5.2. This will ensure that every test case such as `TestA` and `TestB` generates a test span when they are executed.

```
[TestFixture]
[OpenTelemetryExtension]
public class ExampleTests
{
    [Test]
    public void TestA()
    {
        ...
    }

    [Test]
    public void TestB()
    {
        ...
    }
}
```

Figure 5.2: Enable Instrumentation in the Test Client with a Custom Attribute

However, this was not enough to get a working solution due to a problem with threads. As of now, a background thread is running that generates and sends spans to the collector. What can happen is that the main thread that is executing all test cases can finish and quit before the thread that sends spans has finished. Then the collector will not receive all spans. To solve this problem a special class needs to be added once to every test project that after all tests have been executed waits for all the spans to be sent to the collector. This class can be seen in Figure 5.3. Similar to how instrumentation was activated for services, the test client also needs to be given a name and a collector endpoint to send the spans to.

```
[SetUpFixture]
public class OpenTelemetryExtensionSetup
{
    [OneTimeSetUp]
    public void Setup()
    {
        OpenTelemetryExtensionNUnitSetup.Setup(
            new OpenTelemetryExtensionOptions {
                Name = "TestClient",
                CollectorEndpoint = "https://collector.net"
            }
        );
    }

    [OneTimeTearDown]
    public void TearDown()
    {
        OpenTelemetryExtensionNUnitSetup.Teardown();
    }
}
```

Figure 5.3: Code to Ensure That the Spans Are Sent to the Collector Before Exiting the Test Program

5.1.3 Limitations

Two automatic instrumentation libraries were used to generate the client and server spans, namely:

- `OpenTelemetry.Instrumentation.AspNetCore`,
- and `OpenTelemetry.Instrumentation.Http`.

It was possible to extend the spans generated from these two libraries with HTTP headers and bodies. However, due to limitations in the API of these libraries, it was only possible to include the response headers and the response body in the client span and the request headers and the request body in the server span. Consequently, the visualization system will be able to show both the HTTP request and response headers and bodies if both the client and matching server span have been collected. However, if a service communicates with a third-party service then only a client span will be collected and thus only the response headers and body from the third-party service will be available and not the request headers and body.

5.2 Collector

A custom collector was implemented in `.NET` that runs both a gRPC server and also an HTTP server as illustrated in Figure 4.1. The gRPC server

collects all spans and the HTTP server has two endpoints that are used by the visualization system:

- `/GetTestSpans/`: Returns a JSON array with all the test spans, each test span is for a specific test that has been executed, and contains information such as the span id, name of the test case, and if it passed or not. The span id can then be used to get the SDG for a specific test case.
- `/GetServiceDependencyGraph/{testSpanId}`: Given the span id of the test span, this endpoint will run the SDG generation algorithm described in section 4.4 and return the SDG of the test case in JSON format.

5.3 Visualization

The visualization system for the prototype is a web-based application written in React. The application uses D3.js for visualizing the graph. D3.js is a library that has been used by related research to visualize the SDG [19, 20]. Additionally, a library called dagrejs is used to lay out the nodes of the graph using a hierarchical graph drawing algorithm.

5.3.1 Overview

The visualization system consists of four sections, see Figure 5.4 for a complete view of the visualization system. The top section has a button that enables the user to be able to view all test cases that the collector has collected and makes it possible to select a specific test case to inspect the SDG for, see section 5.3.2 for more information on this feature. The middle left section is the actual SDG for a specific test case. Each node represents a service or a test client and each edge represents a dependency. In Figure 5.4, it is possible to see that the node for the test client has a different shape to distinguish it from the other services. The middle right section contains a list of all services and dependencies. If an edge is clicked in the SDG, then a list with all the request-response pairs for that edge is opened in this panel. At the bottom, there is a console view that prints some information about what is happening, for instance, that a test case was loaded, or if there is a red edge in the visualization system.

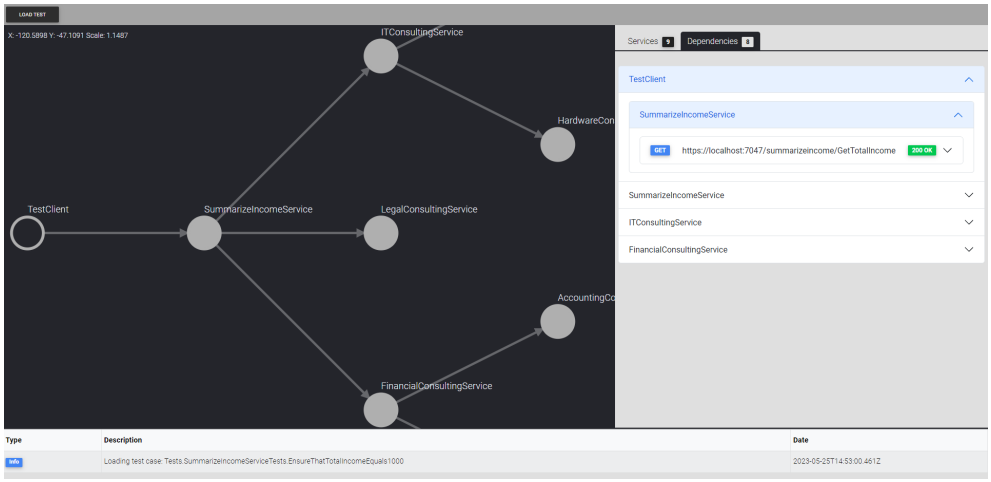

Figure 5.4: Example Image of the Visualization System

5.3.2 Select Test Case

If the button in the top section, see Figure 5.4, is pressed then a table that looks like the table in Figure 5.5 will appear. All test cases are fetched from the collector using the `/GetTestSpans/` endpoint and displayed in the table. It is possible to see the name of the test case and if the test case passed or failed. For each test case, it is possible to press the blue button called ”Inspect” which will start fetching the SDG from the collector using the endpoint `/GetServiceDependencyGraph/{testSpanId}`.

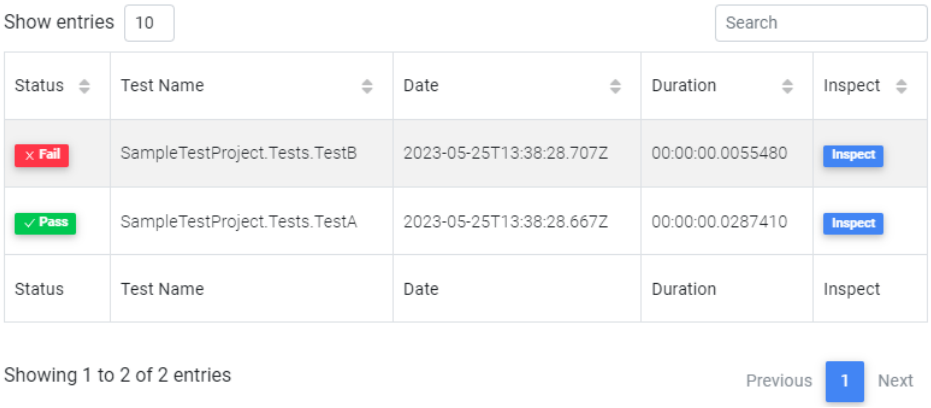

Figure 5.5: Example Table for Selecting a Test Case in the Visualization System

5.3.3 Clickable Edges

As described in the proof-of-concept by clicking on edges it should be possible to see the communication between the two services connected by the edge. Thus, this was implemented by making the edges clickable and if an edge was clicked then a list was automatically opened up in the middle right-hand section of the visualization system.

5.3.4 Edge Coloring

Another feature also described in the proof-of-concept, was to color edges red that had at least one HTTP request-response pair with an HTTP status code of 400 or above. See Figure 5.6 for an example of this. The prototype extended this by not only making the edge red but also showing the status code of the request-response pair that had a status code of 400 or above. Note, that there can be many request-response pairs with a status code of 400 or above. Currently, the algorithm selects the first one that is found and displays the status code of that. However, another alternative would be to prioritize the request-response pairs with status codes of 400 or above and select the most important one, or the one that gives the most information.

There is one special case when the edge is also colored red that does not depend on a HTTP status code of 400 or above. This is when a connection to a service fails due to a service being unresponsive. Then the edge will also be colored red and given the edge label ”Connection Refused”, to indicate that the connection could never be established with the unresponsive service.

Figure 5.6: Example of Colored Edge in the Visualization System

Chapter 6

Results

In total 22 participated in the usability test. Out of these, 18 (81.8%) worked with development and 4 (18.2%) worked with quality assurance. To see the result from task 1, 2, 3, 4, and 5 view section 6.1, section 6.2, section 6.3, section 6.4, and section 6.5, respectively. To see the aggregated effectiveness results from all tasks see section 6.6. To read the aggregated efficiency results from all tasks see section 6.7. Finally, the results from the user satisfaction and if the prototype is considered useful in a real-world microservice system are covered in section 6.8 and section 6.9, respectively.

6.1 Task 1

In the first task, the error was that InvestmentConsultingService was down and unresponsive. The following three alternatives were the most commonly chosen by participants:

- 72.7% of participants selected the correct alternative, which was: ”InvestmentConsultingService is down and does not respond to requests.”
- 22.7% of participants selected an incorrect alternative: ”FinancialConsultingService is missing the necessary authorization header.”
- 4.5% of participants selected another incorrect alternative: ”SummarizeIncomeService is misconfigured and is unable to communicate with any of the other services.”

See Figure 6.1, for how the participants perceived the required effort to solve the task on a scale from one to ten.

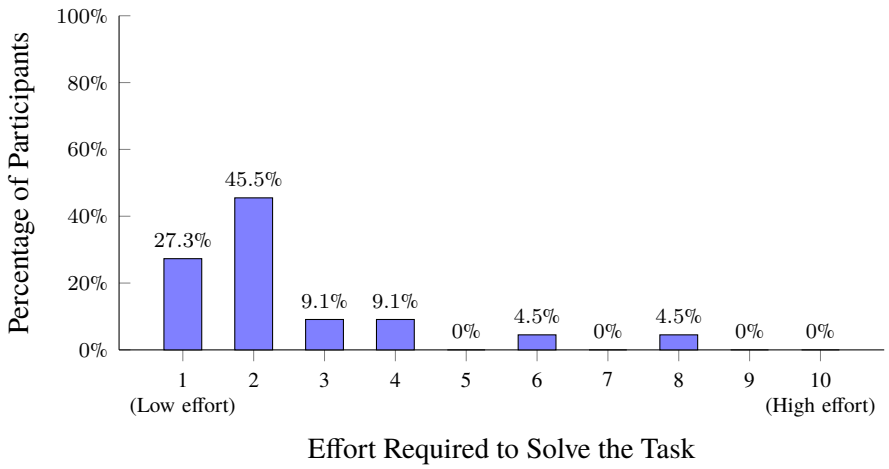

Figure 6.1: Bar Chart Showing the Effort Distribution for Task 1

6.2 Task 2

In the second task, the error was that SummarizeIncomeService had misspelled the URL to FinancialConsultingService. The following three alternatives were the most commonly chosen by participants:

- 81.8% of participants selected the correct alternative, which was: ”SummarizeIncomeService is using an invalid URL to get completed projects from FinancialConsultingService.”
- 9.1% of participants selected an incorrect alternative: ”FinancialConsultingService is experiencing database connectivity issues, causing incomplete data to be returned.”
- 9.1% of participants selected another incorrect alternative: ”FinancialConsultingService is using a different authentication protocol than the other services.”

See Figure 6.2, for how the participants perceived the required effort to solve the task on a scale from one to ten.

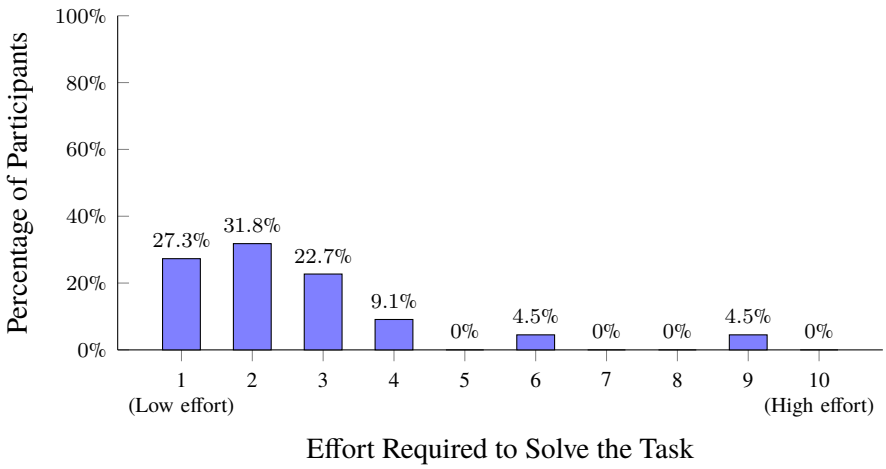

Figure 6.2: Bar Chart Showing the Effort Distribution for Task 2

6.3 Task 3

In the third task, the error was that HardwareConsultingService returned an invalid response body to ITConsultingService. The following three alternatives were the most commonly chosen by participants:

- 59.1% of participants selected the correct alternative, which was: ”HardwareConsultingService is returning data in a format that is not supported by ITConsultingService.”
- 22.7% of participants selected an incorrect alternative: ”SoftwareConsultingService is returning data in a format that is not supported by ITConsultingService.”
- 13.6% of participants selected another incorrect alternative: ”ITConsultingService sent a bad request to SoftwareConsultingService.”

See Figure 6.3, for how the participants perceived the required effort to solve the task on a scale from one to ten.

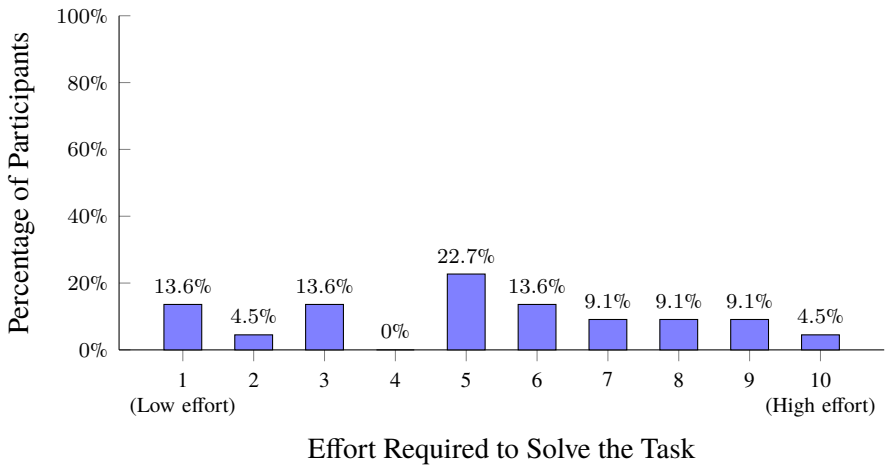

Figure 6.3: Bar Chart Showing the Effort Distribution for Task 3

6.4 Task 4

In the fourth task, the error was that SummarizeIncomeService did not do proper input validation of a query parameter that had an invalid value. The following three alternatives were the most commonly chosen by participants:

- 59.1% of participants selected the correct alternative, which was: ”SummarizeIncomeService does not do proper input validation of the query parameter ”year”.”
- 27.3% of participants selected an incorrect alternative: ”ITConsultingService sent a bad request to SoftwareConsultingService.”
- 9.1% of participants selected another incorrect alternative: ”ITConsultingService does not do proper input validation of the query parameter ”year”.”

See Figure 6.4, for how the participants perceived the required effort to solve the task on a scale from one to ten.

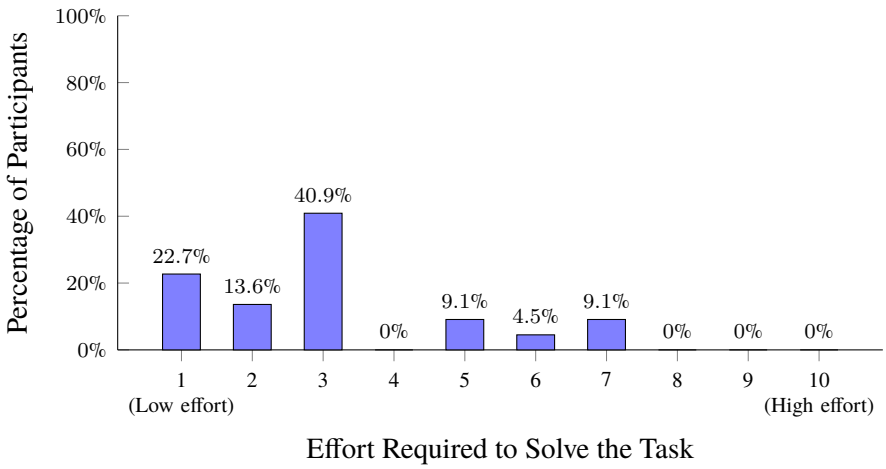

Figure 6.4: Bar Chart Showing the Effort Distribution for Task 4

6.5 Task 5

In the fifth task, the error was that LegalConsultingService did not do return any completed projects which caused SummarizeIncomeService to compute the wrong total income. The following three alternatives were the most commonly chosen by participants:

- 81.8% of participants selected the correct alternative, which was: ”LegalConsultingService does not return any completed projects at all.”
- 13.6% of participants selected an incorrect alternative: ”FinancialConsultingService does only return completed projects from AccountingConsultingService and not InvestmentConsultingService.”
- 4.5% of participants selected another incorrect alternative: ”ITConsultingService does not return any completed projects at all.”

See Figure 6.5, for how the participants perceived the required effort to solve the task on a scale from one to ten.

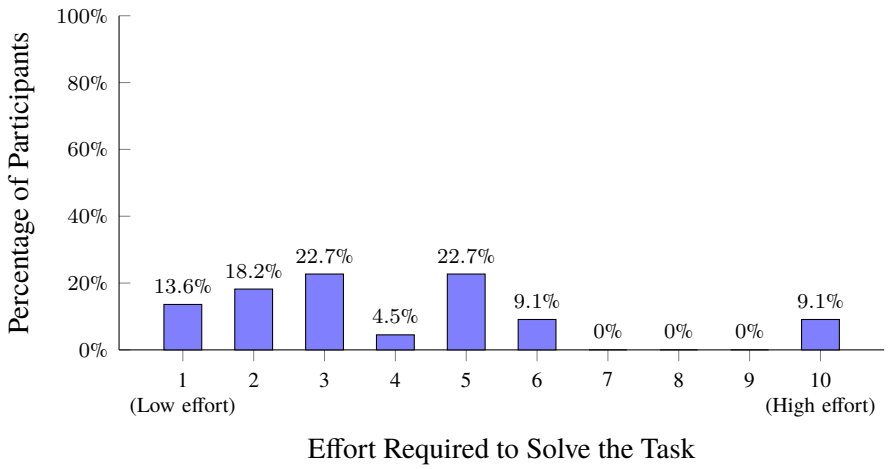

Figure 6.5: Bar Chart Showing the Effort Distribution for Task 5

6.6 Effectiveness

Effectiveness is how well the user can utilize the system to solve the tasks at hand. For each task, the percentage of the participants that were able to solve the task is listed in Table 6.1.

Task	**Percentage of Participants That Solved the Task**
1	72.7%
2	81.8%
3	**59.1%**
4	**59.1%**
5	81.8%

Table 6.1: Aggregated Correctness Results from All Tasks

On average, the average participant solved 70.9% of all tasks correctly. In Table 6.1, it is clear that task 2 and task 3 had the lowest correctness rate of 59.1%. While task 2 and task 5 had the highest correctness rate of 81.8%.

6.7 Efficiency

Efficiency is how much effort and/or time the user must spend to solve each task using the system. See the aggregated results from all tasks in Table 6.2.

In Table 6.2, task 3 and task 5 do stand out with a higher mean and also a higher spread. Task 4 had slightly higher mean and median values compared to task 1 and task 2. Task 1 and task 2 had the lowest mean values. The average participant was required to use an average effort level of 3.5 on a scale from one to ten in order to solve an average task.

Task	**Mean**	**Median**	**Variance**
1	2.5	2.0	3.0
2	2.6	2.0	3.6
3	**5.2**	**5.0**	**7.3**
4	3.1	3.0	3.4
5	**4.0**	**3.0**	**6.3**

Table 6.2: Aggregated Effort Results from All Tasks

When participants were asked to motivate why a task required low effort or high effort to solve with the help of the visualization system the following answers were mentioned to describe why a task required low effort:

- The red edge color gave a good hint of where the problematic area was.
- It felt easy and quick to retrieve information about requests and responses, which made the tasks solvable.
- The easy tasks could be solved directly by just looking at the SDG.
- It was easy to follow the call chain and find all input and output data, this was enough to figure out what was wrong.

The following answers were mentioned to describe why a task required high effort:

- It was sometimes required to go through many dependencies especially if no edge was colored red, which was the case in task 5.
- Misinterpreted what an edge in the graph meant.
- Not enough visualization to solve the problem quickly.

6.8 User Satisfaction

When asked if the user felt satisfied when using the system to solve the tasks the user answered on a scale from one to ten as shown in Figure 6.6.

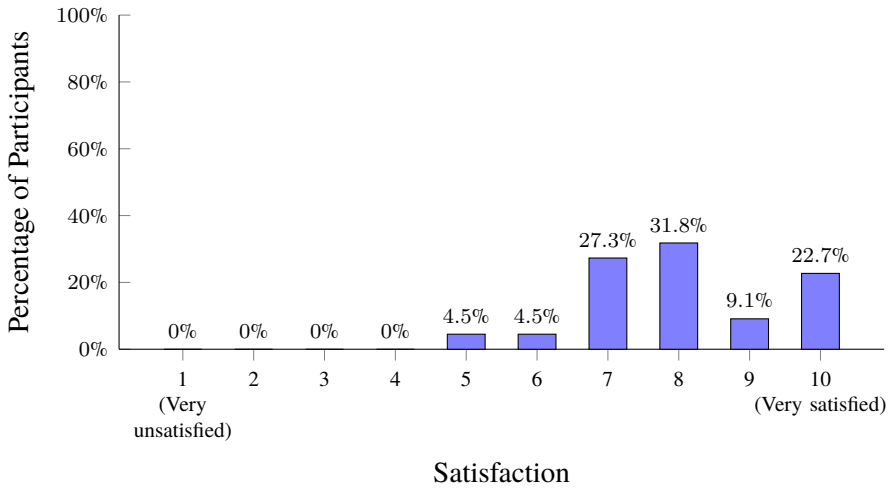

Figure 6.6: Bar Chart Showing the User Satisfaction Distribution

The mean user satisfaction score was 8.0. Additionally, the median user satisfaction score was also 8.0. The variance of the user satisfaction scores was 2.0.

The participants were asked to motivate why they gave the visualization system a high satisfaction score or a low satisfaction score. The following reasons were mentioned for why they gave the visualization system a high satisfaction score:

- It gives a great overview of all the call chains and makes it possible to quickly inspect suspicious calls.
- Enables the inspection of services deep down the call chain.
- Very user-friendly interface.
- The visualization makes it easy to track routes and follow the flow between services.

The following reasons were mentioned for why they gave the visualization system a lower satisfaction score:

- Do not know if the visualization system would work well in a real-world scenario.
- Cannot visualize common types of communication such as gRPC communication and communication with service buses.
- The user interface can be improved, for instance by making it more straightforward to find the body of HTTP messages.
- Some flows can be difficult to understand.

6.9 Usefulness

Out of all participants, 95.5% thought that the visualization system could be useful in a commercial real-world microservice system, and the remaining 4.5% thought that the visualization system could not be useful. The same reasons as highlighted in the previous section on why the visualization system was satisfying to work with were used as a motivation for why the visualization system could be useful in a real-world microservice system. Additionally, some other reasons to motivate the usefulness of the visualization system were brought forward:

- This visualization system can remove tedious work, which developers do sometimes such as following request-response pairs in logs.
- This tool can reduce the time required to find the root cause of failed test cases.
- This visualization system is more bare-bone compared to other monitoring systems, which means that it is possible to learn it quickly and use it without spending a lot of time.

Concerning the critique and why the visualization system would not work well in a real-world microservice system. The most common critique was that real-world microservice systems contain more complex forms of communication. For example, asynchronous communication with service buses. This is according to some participants a crucial feature that needs to be added to the visualization system for it to be useful in real-world microservice systems.

Chapter 7

Discussion

7.1 Methods

Usability testing is a suitable evaluation methodology for this project work because the visualization system of the prototype is an interactive web application. Remote usability testing makes it possible to save time and get more data points. The downside is that it is harder to monitor how the user interacts with the web application and ask follow-up questions. In the usability test, participation was voluntary, resulting in a potential bias towards individuals who were interested in participating. It is possible that those who volunteered may have been more familiar with or experienced in using similar tools.

Five different error categories were tested in the evaluation of this master thesis. In a real-world setting, there are of course many more different error types that can occur, and there can also occur many errors at the same time. The reason behind an error may also be sophisticated and it is not necessary that the visualization system can pinpoint exactly what the root cause of a failing test case is. For example, the error in task 1 was that a service was down and unresponsive. The visualization system can pinpoint which service that is failing in a call chain. However, it will not provide the reason for why the service is down. This can be due to several reasons e.g. misconfiguration, server failure, etc. The reason for the service being down must be analyzed using other means. However, knowing which service the problem exists in should not be neglected as irrelevant information, as this provides a crucial first step in the analysis of the error.

The fictional microservice system used in the usability test allowed for the testing of five different error types that can occur in any microservice

system. However, it is important to note that the ease of identifying these errors using the visualization system may differ in a real-world microservice system. Since the fictional microservice system was specifically designed for the usability test, it may provide a more straightforward means of identifying and visualizing the errors. In contrast, a real-world microservice system could involve more complex interactions and dependencies, making the identification of errors through visualization more challenging.

It is a case to be made about the assignments being too simple. For each task, the participant was provided with five alternatives, which could enable the participant to use the method of exclusion to arrive at the correct alternative without having a good grasp of what went wrong in the microservice system. To mitigate this problem, alternatives were crafted in such a way as to make it difficult to directly exclude alternatives. For example, in some tasks several alternatives had to do with the error but only the correct alternative covered the root cause of why the error occurred in the first place.

To evaluate the prototype other methods were considered for instance A/B testing. However, to use such a methodology it is required to have something to compare the prototype against. In the case of this masterwork, a good baseline could not be identified. I could not find an open-source or publicly available tool like the visualization system developed in this paper. There are monitoring systems like Jaeger and Zipkin that can build an SDG from OpenTelemetry traces, however, they do not enable the user to see the HTTP headers or body being sent. In addition, they do not filter the SDG to only contain the communication that belongs to a specific test case. Thus, they are very different products and cannot be compared fairly. One option would be to compare the prototype against collected logs of the communication between services. However, then one must decide how to structure and format the logs and in what order they should be presented to the user. All these factors can severely affect how well the logs can be read and understood. Consequently, it was decided that comparing the visualization system to a collection of logs can also result in an unfair comparison.

The visualization system was evaluated using usability testing with respect to effectiveness, efficiency, and user satisfaction. However, efficiency was subjectively measured by letting the user provide a value between one to ten to rate how much effort that was required to solve the task. A more objective measurement would be to collect time instead. Time could indicate how much effort a certain task required. However, this was not used due to the fact that Google Forms has no official support for capturing time. In addition, time can also introduce stress for the participant and make him or her more likely

to use strategies to exclude alternatives and guess the correct answer, which is not desirable when measuring how many tasks that can be solved with the prototype.

7.2 Implementation

There are some features of the implementation that should be discussed. First of all, if this implementation is used in a real-world microservice system, then the collector and instrumentation should be activated in a separate environment from the production environment. For instance, this should only be run in a separate test or UAT environment mainly due to two reasons. The first reason is that it becomes a significant performance bottleneck to capture all HTTP requests and responses, especially the headers and bodies of all HTTP messages. This will significantly degrade the general performance of the microservice system, which is something to be avoided in a production system. The second reason is due to security and integrity. If this system were to be run in a production environment then private information such as personal data and access tokens would be captured and stored in a centralized collector. This would be a prime target for cybercriminals to get access to all HTTP communication in a microservice system.

In the proof-of-concept and the prototype, there is only a single collector that collects all spans. This collector can become a bottleneck and may even crash if the load is too high. In this master thesis, a single collector was only considered because it is enough to evaluate the prototype. However, if the system should be used for real, one should consider using many collectors and providing a load balancer between them to ensure that a single collector does not become a bottleneck in the microservice system. The way the SDG is built may also need to change if all spans are not accessible by a single collector anymore.

7.3 Results

The results show that the visualization system has great potential to be a useful tool to identify the root cause of failing test cases in microservice systems. From the results, the average participant solved 70.9% of all tasks on average, and the average effort required to solve an average task was 3.5 on a scale from one to ten. These tasks involved five different error categories, namely: service is down, misspelled URL, bad response data, bad request data, and

missing response data.

Both task 1 and task 2 required a low average effort rate and had a high average correctness rate. These two tasks are discussed briefly in section 7.3.1. The lowest correctness rate was for task 3 and task 4. Task 3 was also considered the task that required the most amount of effort to be solved according to the average participant. See section 7.3.2 and section 7.3.3 for a discussion on task 3 and task 4, respectively. Lastly, see section 7.3.4 for task 5 which required the second highest average effort to be solved according to participants.

7.3.1 Task 1 and 2

The error in task 1 was that a service was down and the error in task 2 was a misspelled URL. Compared to the other tasks task 1 and task 2 were the tasks that required the lowest average effort to be solved according to participants. Motivations for the low effort according to the participants had to do with the red edges that highlighted the problematic areas and the fact that just observing the SDG was enough to figure out the root cause of the failing test case. Consequently, these two tasks did require a small amount of manual work as the error could be directly identified by observing the SDG. According to the results, task 1 had a slightly lower average correctness rate compared to task 2.

See Figure 7.1 for the SDG that the participant saw in task 1. The participants need to identify that the connection from FinancialConsultingService to InvestementConsultingService could not be established due to InvestementConsultingService being down and unresponsive, which is what the red edge with the label ”Connection Refused” indicates. However, 22.7% selected the wrong alternative: ”FinancialConsultingService is missing the necessary authorization header”. However, if that were the case then instead of ”Connection Refused” the edge label would be ”401 Unauthorized”. Thus, one logical conclusion for why so many picked the wrong alternative could be that they did not understand what ”Connection Refused” meant due to it not being a standardized way to express that a service is down. This indicates that the name of edge labels impacts how well the visualization system works.

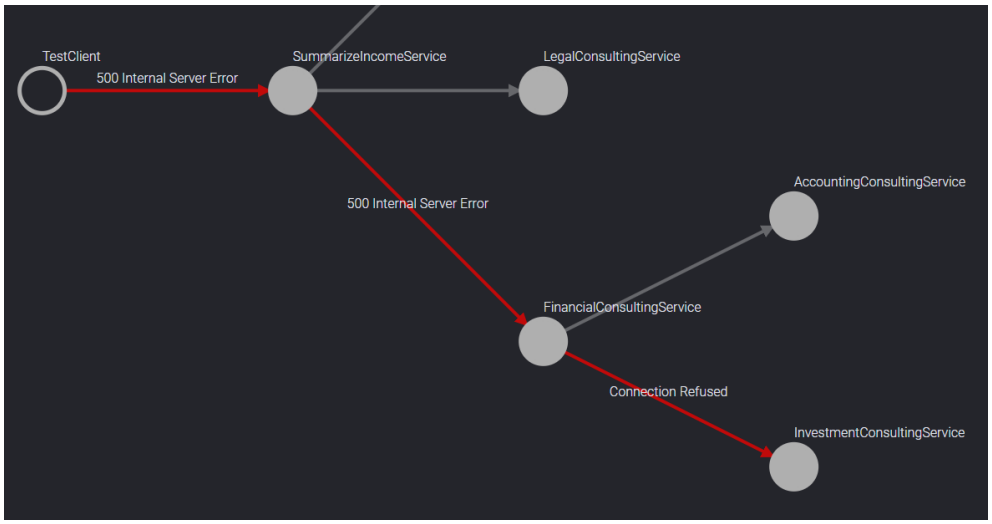

Figure 7.1: The SDG the User Saw When Trying to Solve Task 1

7.3.2 Task 3

Task 3, had the highest mean effort and the lowest average correctness rate. Thus, this task was likely the hardest task that the participant had to solve. Task 3 covered bad response data which required the user to inspect the response body of at least two services to find out which one returned the invalid response data. Two factors can be identified that made this task tricky. See Figure 7.2, for what the user saw when trying to solve this task. The error is that HardwareConsultingService returns a bad response body to ITConsultingService, which in turn made ITConsultingService crash and return a response code of 500. The red edge can appear a bit misleading here because it can make the user focus on the dependency between SummarizeIncomeService and ITConsultingService, which is not where the root cause of the problem resides. By clicking on the dependency between SummarizeIncomeService and ITConsultingService it is possible to see an error message that would indicate that there is a format error that made ITConsultingService crash. This should provide enough of a hint to make the user look up the responses from both SoftwareConsultingService and HardwareConsultingService to see if they are returning something strange. One should then identify that HardwareConsultingService is returning projects where the income is formatted as a string with a dollar sign. The error message previously read, also points out that there was a problem with converting a string to a double due to the dollar sign in the string.

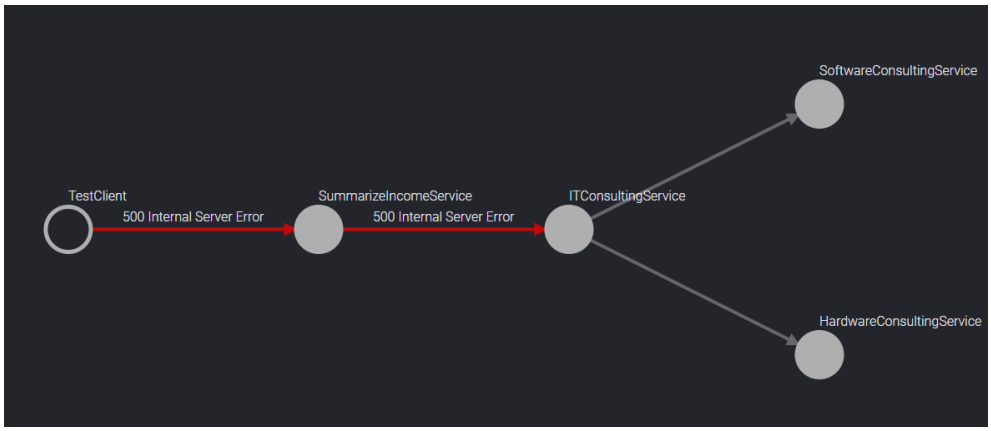

Figure 7.2: The SDG the User Saw When Trying to Solve Task 3

In task 3, 22.7% picked the option ”SoftwareConsultingService is returning data in a format that is not supported by ITConsultingService”, which indicates that they knew that there were a problem with a response body, however, they did not properly investigate both the response body from SoftwareConsultingService and HardwareConsultingService. Perhaps they assumed that it must come from the service that is contacted by ITConsultingService first, which is not the case. Consequently, this highlights that the visualization is not a quick fix nor an automatic analysis tool that finds the error automatically for the user. With this tool, the user is still required to read error messages and go through request-response pairs between dependencies.

7.3.3 Task 4

Task 4, had similarly to task 3 also the lowest average correctness rate. Task 4, covered a negative test, which makes the task a bit special as all the other tasks covered positive tasks. In task 4, the test was that SummarizeIncomeService should return ”400 Bad Request” when an optional query parameter is used with an invalid value. The error was that SummarizeIncomeService did not do proper input validation of the query parameter and therefore instead of directly returning a status code of 400, the invalid query parameter was passed down to services deep down the call chain. Finally, when SoftwareConsultingService received the query parameter with an invalid value, a status code of 400 was returned, which generated a cascading effect where both ITConsultingService and SummarizeIncomeService returned a status code of 500 as illustrated in Figure 7.3.

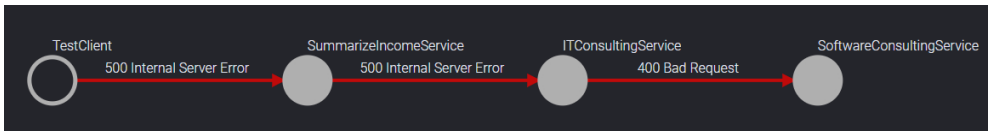

Figure 7.3: The SDG the User Saw When Trying to Solve Task 4

The participant in this case must spot that SummarizeIncomeService does not do proper input validation of the query parameter since the parameter is passed down to ITConsultingService. This can be observed by following the requests being made from the TestClient and seeing how the query parameter is allowed through until it reaches SoftwareConsultingService which luckily did do proper input validation. The most popular wrong alternative picked by 27.3% of the participants was ”ITConsultingService sent a bad request to SoftwareConsultingService”, which indeed is part of the problem, however, it is not the root cause of the problem. The query parameter should never have reached ITConsultingService, it should have been stopped directly by SummarizeIncomeService, which is why this was an incorrect alternative. This shows that it is not always the case that the root cause of an error can be identified by looking deep down the call chain and taking the original bad response message that made the microservice system crash. Some additional grasp of how the system is supposed to behave is also needed to accurately understand the root cause of the failed test case.

7.3.4 Task 5

Task 5, had the second highest mean effort of 4.0 according to participants. However, at the same time, task 5 got one of the highest correctness rates of 81.8%. The reason participants thought this task required a high amount of effort was due to the fact that there were no red edges in the visualization, as can be seen in Figure 7.4.

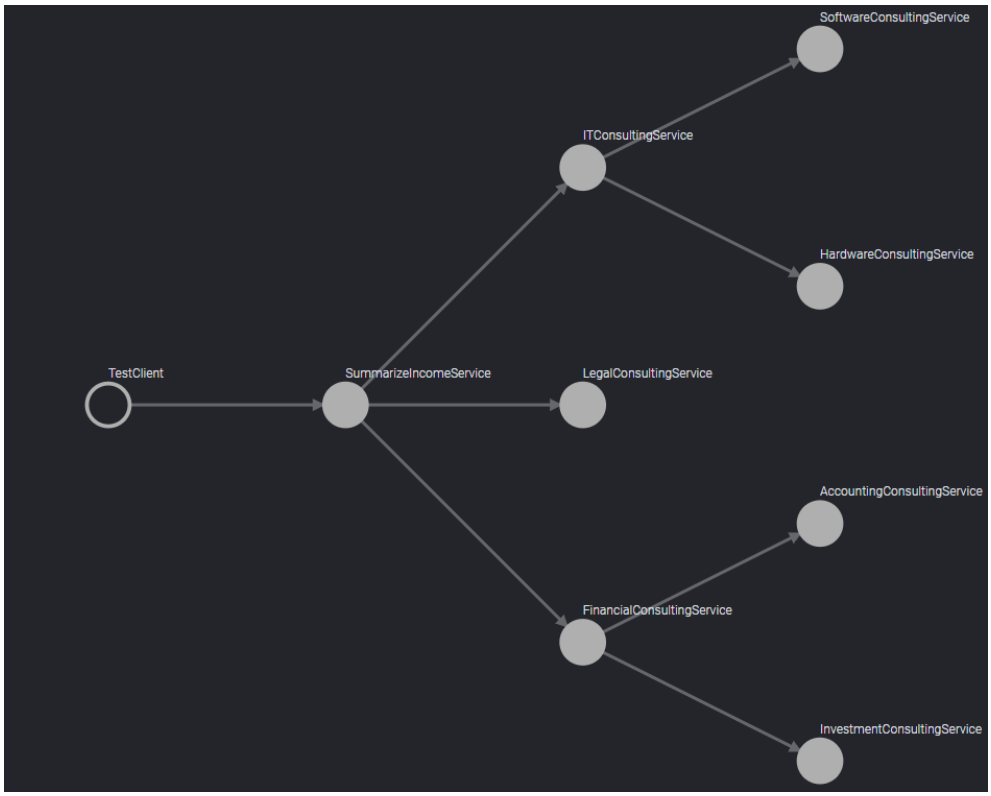

Figure 7.4: The SDG the User Saw When Trying to Solve Task 5

Without additional visualization, the user is left to go over possible all dependencies in the worst case and check all request-response pairs to figure out which service that is returning an empty response. This is a tedious task. It is possible to extend the visualization system to provide features that would enable the user to find these types of errors more quickly. For instance, a search feature that allows the user to search for all request-response pairs where the response body has a certain format.

Chapter 8

Conclusions and Future work

In this project work, a novel SDG-based method was developed to investigate the research question of whether the use of an SDG-based method can simplify the software testing process of a microservice system. The implementation was divided into a proof-of-concept and a realization of the proof-of-concept which was referred to as the prototype. The proof-of-concept was a general model that can work for many different microservice systems, and the prototype was a realization of the proof-of-concept specifically designed to work with services written in `.NET` together with the testing framework `NUnit`. To answer the research question the prototype was evaluated in a usability test with developers and quality assurance engineers from Marginalen Bank. The results, from the usability test, show that some error categories such as service down and misspelled URL can be directly identified in the visualization system just by observing the SDG. However, more complex error categories such as invalid request and response data require more manual work and going through request-response pairs.

The participants thought that the tasks that could be directly solved by observing the SDG required on average less effort than the tasks that required clicking around in the SDG and observing request-response pairs. A common reason that was highlighted to make the user satisfied and the tasks low effort was how problematic areas were marked as red in the SDG, and how the SDG allowed the user to understand connections and follow call chains. However, in the tasks, especially in task 3 and task 4 a minority of the participants seemed to rely too much on the SDG and skipped analyzing the request-response pairs and therefore came up with the wrong answer. Consequently, this highlights the fact that just observing the SDG is insufficient to identify the root cause of all problems. An understanding of the system and manual analysis of request-

response pairs in the visualization system is still necessary. In addition, too much emphasis on red edges should not be taken. Instead, they should just provide a hint of where to start analyzing in the microservice system.

In conclusion, SDG-based methods can simplify the process of finding the root cause of failing test cases for at least five different error types covered in this project work. SDG-based methods can provide a graspable visualization that enables the user to follow communication flows and understand how services depend on each other in a test case. In complex microservice systems with many dependencies, this can be an invaluable feature. Additional visualization features such as red edges can reduce the required effort by removing the burden of having to analyze every request-response pair. However, the prototype brought forward in this project work, shows that an SDG alone cannot automatically identify the root cause of failing test cases. Manual analysis is still required but the SDG can provide a debugging platform that simplifies the job of the user.

In future work, it would be interesting to see how well the prototype and SDG-based methods in general can simplify the software testing process for real-world microservice systems. Furthermore, more types of errors than the five discussed in this project work should be considered in the future. In addition, how SDG-based methods work when there are multiple errors in the same test. For example, if two services are down at the same time. As highlighted by participants in the usability study, there are many more communication protocols that microservice systems can use, not only HTTP communication. Consequently, it would be interesting to add support for additional communication protocols such as gRPC and service bus protocols.

References

[1] X. Larrucea, I. Santamaria, R. Colomo-Palacios, and C. Ebert, "Microservices," *IEEE Software*, vol. 35, no. 3, pp. 96–100, May 2018. doi: 10.1109/MS.2018.2141030 Conference Name: IEEE Software. [Pages 1 and 9.]

[2] J. Thönes, "Microservices," *IEEE Software*, vol. 32, no. 1, pp. 116–116, Jan. 2015. doi: 10.1109/MS.2015.11 Conference Name: IEEE Software. [Page 1.]

[3] O. Al-Debagy and P. Martinek, "A Comparative Review of Microservices and Monolithic Architectures," in *2018 IEEE 18th International Symposium on Computational Intelligence and Informatics (CINTI)*, Nov. 2018. doi: 10.1109/CINTI.2018.8928192 pp. 000 149–000 154, iSSN: 2471-9269. [Pages 1 and 9.]

[4] J. P. Sotomayor, S. C. Allala, P. Alt, J. Phillips, T. M. King, and P. J. Clarke, "Comparison of Runtime Testing Tools for Microservices," in *2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)*, vol. 2, Jul. 2019. doi: 10.1109/COMPSAC.2019.10232 pp. 356–361, iSSN: 0730-3157. [Pages 1 and 9.]

[5] S. Baškarada, V. Nguyen, and A. Koronios, "Architecting Microservices: Practical Opportunities and Challenges," *Journal of Computer Information Systems*, vol. 60, no. 5, pp. 428–436, Sep. 2020. doi: 10.1080/08874417.2018.1520056 Publisher: Taylor & Francis _eprint: https://doi.org/10.1080/08874417.2018.1520056. [Online]. Available: https://doi.org/10.1080/08874417.2018.1520056 [Pages 1, 9, and 10.]

[6] M. Waseem, P. Liang, G. Márquez, and A. D. Salle, "Testing Microservices Architecture-Based Applications: A Systematic Mapping Study," in *2020 27th Asia-Pacific Software Engineering Conference*

(APSEC), Dec. 2020. doi: 10.1109/APSEC51365.2020.00020 pp. 119–128, iSSN: 2640-0715. [Pages 1, 2, and 9.]

[7] M. Waseem, P. Liang, M. Shahin, A. Di Salle, and G. Márquez, "Design, monitoring, and testing of microservices systems: The practitioners' perspective," *Journal of Systems and Software*, vol. 182, p. 111061, Dec. 2021. doi: 10.1016/j.jss.2021.111061. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121221001588 [Pages 1, 2, 9, 10, and 27.]

[8] P. Jamshidi, C. Pahl, N. C. Mendonça, J. Lewis, and S. Tilkov, "Microservices: The Journey So Far and Challenges Ahead," *IEEE Software*, vol. 35, no. 3, pp. 24–35, May 2018. doi: 10.1109/MS.2018.2141039 Conference Name: IEEE Software. [Pages 1 and 9.]

[9] R. T. Fielding and J. Reschke, "Hypertext transfer protocol (HTTP/1.1): Semantics and content," num Pages: 101. [Online]. Available: https://datatracker.ietf.org/doc/rfc7231 [Page 5.]

[10] T. Bray, "The JavaScript object notation (JSON) data interchange format," num Pages: 16. [Online]. Available: https://datatracker.ietf.org/doc/rfc8259 [Page 6.]

[11] Z. Xiao, I. Wijegunaratne, and X. Qiang, "Reflections on SOA and Microservices," in *2016 4th International Conference on Enterprise Systems (ES)*, Nov. 2016. doi: 10.1109/ES.2016.14 pp. 60–67. [Page 6.]

[12] N. Kratzke, "About Microservices, Containers and their Underestimated Impact on Network Performance," Sep. 2017, arXiv:1710.04049 [cs]. [Online]. Available: http://arxiv.org/abs/1710.04049 [Page 7.]

[13] F. Montesi and J. Weber, "Circuit Breakers, Discovery, and API Gateways in Microservices," Sep. 2016, arXiv:1609.05830 [cs]. [Online]. Available: http://arxiv.org/abs/1609.05830 [Pages 7 and 8.]

[14] A. Balalaie, A. Heydarnoori, and P. Jamshidi, "Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture," *IEEE Software*, vol. 33, no. 3, pp. 42–52, May 2016. doi: 10.1109/MS.2016.64 Conference Name: IEEE Software. [Pages 8 and 9.]

[15] D. R. Apolinário and B. B. de França, "A method for monitoring the coupling evolution of microservice-based architectures," *Journal*

of the Brazilian Computer Society, vol. 27, no. 1, p. 17, Dec. 2021. doi: 10.1186/s13173-021-00120-y. [Online]. Available: https://doi.org/10.1186/s13173-021-00120-y [Page 9.]

[16] M. J. Kargar and A. Hanifizade, "Automation of regression test in microservice architecture," in *2018 4th International Conference on Web Research (ICWR)*, Apr. 2018. doi: 10.1109/ICWR.2018.8387249 pp. 133–137. [Pages 10 and 17.]

[17] J. G. Quenum and S. Aknine, "Towards Executable Specifications for Microservices," in *2018 IEEE International Conference on Services Computing (SCC)*, Jul. 2018. doi: 10.1109/SCC.2018.00013 pp. 41–48, iSSN: 2474-2473. [Pages 10 and 17.]

[18] M. Rahman and J. Gao, "A Reusable Automated Acceptance Testing Architecture for Microservices in Behavior-Driven Development," in *2015 IEEE Symposium on Service-Oriented System Engineering*, Mar. 2015. doi: 10.1109/SOSE.2015.55 pp. 321–325. [Pages 10 and 17.]

[19] S.-P. Ma, C.-Y. Fan, Y. Chuang, W.-T. Lee, S.-J. Lee, and N.-L. Hsueh, "Using Service Dependency Graph to Analyze and Test Microservices," in *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, vol. 02, Jul. 2018. doi: 10.1109/COMPSAC.2018.10207 pp. 81–86, iSSN: 0730-3157. [Pages 10, 16, 17, and 39.]

[20] S.-P. Ma, C.-Y. Fan, Y. Chuang, I.-H. Liu, and C.-W. Lan, "Graph-based and scenario-driven microservice analysis, retrieval, and testing," *Future Generation Computer Systems*, vol. 100, pp. 724–735, Nov. 2019. doi: 10.1016/j.future.2019.05.048. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X19302614 [Pages 10, 16, 17, and 39.]

[21] A. Uddin and A. Anand, "Importance of Software Testing in the Process of Software Development," *International Journal for Scientfic Research and Development*, pp. 2321–0613, Jan. 2019. [Page 10.]

[22] P. Bourque, R. Dupuis, A. Abran, J. Moore, and L. Tripp, "The guide to the Software Engineering Body of Knowledge," *IEEE Software*, vol. 16, no. 6, pp. 35–44, Nov. 1999. doi: 10.1109/52.805471 Conference Name: IEEE Software. [Page 10.]

[23] M. Cohn, *Succeeding with agile : software development using Scrum*, ser. The Addison-Wesley signature series. Upper Saddle River, N.J: Addison-Wesley, 2010. ISBN 0-321-57936-4 Publication Title: Succeeding with agile : software development using Scrum. [Page 10.]

[24] N. Radziwill and G. Freeman, "Reframing the Test Pyramid for Digitally Transformed Organizations," Nov. 2020, arXiv:2011.00655 [cs]. [Online]. Available: http://arxiv.org/abs/2011.00655 [Page 11.]

[25] T. Cerny, A. S. Abdelfattah, V. Bushong, A. Al Maruf, and D. Taibi, "Microservice Architecture Reconstruction and Visualization Techniques: A Review," in *2022 IEEE International Conference on Service-Oriented System Engineering (SOSE)*, Aug. 2022. doi: 10.1109/SOSE55356.2022.00011 pp. 39–48, iSSN: 2642-6587. [Page 12.]

[26] X. Défago, "Causal Order, Logical Clocks, State Machine Replication," in *Encyclopedia of Algorithms*, M.-Y. Kao, Ed. Boston, MA: Springer US, 2008, pp. 129–131. ISBN 978-0-387-30162-4. [Online]. Available: https://doi.org/10.1007/978-0-387-30162-4_65 [Page 12.]

[27] G. Leffler, "OpenTelemetry and observability: What, why, and why now?" Sydney: USENIX Association, Dec. 2022. [Page 12.]

[28] J. M. C. Bastien, "Usability testing: a review of some methodological and technical aspects of the method," *International Journal of Medical Informatics*, vol. 79, no. 4, pp. e18–e23, 2010. doi: https://doi.org/10.1016/j.ijmedinf.2008.12.004. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1386505608002098 [Page 14.]

[29] A. M. Wichansky, "Usability testing in 2000 and beyond," *Ergonomics*, vol. 43, no. 7, pp. 998–1006, 2000. doi: 10.1080/001401300409170. [Online]. Available: https://doi.org/10.1080/001401300409170 [Page 14.]

[30] J. Nielsen and T. K. Landauer, "A mathematical model of the finding of usability problems," in *Proceedings of the INTERACT '93 and CHI '93 Conference on Human Factors in Computing Systems*, ser. CHI '93. New York, NY, USA: Association for Computing Machinery, May 1993. doi: 10.1145/169059.169166. ISBN 978-0-89791-575-5 pp. 206–213.

[Online]. Available: https://dl.acm.org/doi/10.1145/169059.169166 [Page 14.]

[31] B. Zhang, K. Wen, J. Lu, and M. Zhong, "A Top-K QoS-Optimal Service Composition Approach Based on Service Dependency Graph," *Journal of Organizational and End User Computing (JOEUC)*, vol. 33, no. 3, pp. 50–68, May 2021. doi: 10.4018/JOEUC.20210501.oa4 Publisher: IGI Global. [Online]. Available: https://www.igi-global.com/article/a-top-k-qos-optimal-service-composition-approach-based-on-service-dependency-graph/www.igi-global.com/article/a-top-k-qos-optimal-service-composition-approach-based-on-service-dependency-graph/276376 [Page 16.]

[32] S. Hashemian and F. Mavaddat, "A graph-based approach to Web services composition," in *The 2005 Symposium on Applications and the Internet*, Feb. 2005. doi: 10.1109/SAINT.2005.4 pp. 183–189. [Page 16.]

[33] A. Santos and H. Paula, "Microservice decomposition and evaluation using dependency graph and silhouette coefficient," in *15th Brazilian Symposium on Software Components, Architectures, and Reuse*, ser. SBCARS '21. New York, NY, USA: Association for Computing Machinery, Oct. 2021. doi: 10.1145/3483899.3483908. ISBN 978-1-4503-8419-3 pp. 51–60. [Online]. Available: https://doi.org/10.1145/3483899.3483908 [Page 16.]

[34] O. Al-Debagy and P. Martinek, "Dependencies-based microservices decomposition method," *International Journal of Computers and Applications*, vol. 44, no. 9, pp. 814–821, Sep. 2022. doi: 10.1080/1206212X.2021.1915444 Publisher: Taylor & Francis _eprint: https://doi.org/10.1080/1206212X.2021.1915444. [Online]. Available: https://doi.org/10.1080/1206212X.2021.1915444 [Page 16.]

[35] E. Gaidels and M. Kirikova, "Service Dependency Graph Analysis in Microservice Architecture," in *Perspectives in Business Informatics Research*, ser. Lecture Notes in Business Information Processing, R. A. Buchmann, A. Polini, B. Johansson, and D. Karagiannis, Eds. Cham: Springer International Publishing, 2020. doi: 10.1007/978-3-030-61140-8_9. ISBN 978-3-030-61140-8 pp. 128–139. [Page 16.]

[36] I. U. P. Gamage and I. Perera, "Using dependency graph and graph theory concepts to identify anti-patterns in a microservices system: A tool-

based approach," in *2021 Moratuwa Engineering Research Conference (MERCon)*, Jul. 2021. doi: 10.1109/MERCon52712.2021.9525743 pp. 699–704, iSSN: 2691-364X. [Page 16.]

[37] A. Al Maruf, A. Bakhtin, T. Cerny, and D. Taibi, "Using Microservice Telemetry Data for System Dynamic Analysis," in *2022 IEEE International Conference on Service-Oriented System Engineering (SOSE)*, Aug. 2022. doi: 10.1109/SOSE55356.2022.00010 pp. 29–38, iSSN: 2642-6587. [Page 16.]

[38] B. Cemellini, R. Thompson, P. Van Oosterom, and M. De Vries, "Usability testing of a web-based 3D Cadastral visualization system," in *Proceedings of the 6th International FIG Workshop on 3D Cadastres, Delft, The Netherlands*, 2018, pp. 1–5. [Page 20.]

[39] M. Bach-Nutman, "Understanding The Top 10 OWASP Vulnerabilities," 2020, _eprint: 2012.09960. [Page 24.]

[40] N. Nikolov, "Sugiyama Algorithm," Jan. 2016, pp. 2162–2166. [Page 32.]

[41] K. Larkin, S. Smith, S. Addie, and B. Dahler, "Dependency injection in ASP.NET Core," May 2023. [Online]. Available: https://learn.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection [Page 35.]

TRITA-EECS-EX-2023:552