A STUDY OF TESTING A MICROSERVICE SYSTEM BASED ON CODE COVERAGE

BY

SHIRDON GORSE

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2019

Urbana, Illinois

Adviser:

Professor Tao Xie

ABSTRACT

The microservice architecture is a service-oriented architecture that supports the development of small loosely-coupled software services. In contrast to the monolithic architecture, the microservice architecture is increasingly popular among developers since it allows for an easier development process, reduces the barrier to adding new technologies to a system, and increases the efficiency of a system's scalability. Testing a microservice system with a test suite (i.e., a collection of test cases) is an important means for improving the quality of the microservice system. To guide the assessment and improvement of the test suite, code-coverage information is commonly used. However, there exists no research on investigating how code-coverage information can be used to measure and improve a test suite for a microservice system. To fill this gap, we conduct an empirical study on testing a benchmark microservice system by measuring code-coverage achieved by its test suite based on a coverage-measurement infrastructure that we build. The line-coverage results show that covering a major portion of the lines not being covered is blocked by error-handling branches (i.e., branches related to error handling). Based on the code-coverage information, we explore ways of augmenting the existing test suite to achieve higher code-coverage. In particular, guided by the code-coverage information, we augment the existing test suite by adding new test cases to aim for covering those functionalities whose related code portions are not covered by the existing test suite. In addition, to covering those error-handling branches, we augment the test suite by manipulating its execution environment, i.e., mutating the messages sent by a microservice to another microservice in order to aim for covering not-covered error-handling branches. Such coverage-guided test-suite augmentation helps achieve higher code-coverage of the microservice system.

To my parents, for their love and support.

ACKNOWLEDGMENTS

I would like to thank my advisor, Professor Tao Xie, for providing me with an opportunity to join his Illinois ASE research group. I would also like to thank the microservice project team from the Illinois ASE research group for providing support to me, and being there to answer questions when they arose. Through the process of writing my thesis, I have learned and gained a completely different admiration for the research community through working on my thesis.

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION

The rise of cloud computing has caused the usage of the microservice architecture to become increasingly popular. The microservice architecture is a service-oriented architecture used to develop software systems by creating small loosely-coupled services that use the same communication mechanism, such as the Hypertext Transfer Protocol (HTTP) [1]. There are various benefits of using this architecture in contrast to the monolithic architecture. Example benefits include development agility when adding new technologies to a system, deployment efficiency since all microservices are deployed and run independently, and an increase in runtime efficiency with respect to a system's scalability [2].

Testing a microservice system is an important means to improve the quality of the microservice system. During or after the development of a microservice system, the developers or testers test the microservice system by typically creating and running a test suite, i.e., a collection of test cases falling into various categories. For example, unit tests are usually the initial test cases that the developers create for an individual microservice. However, it is difficult for such unit tests to detect faults related to interactions among microservices. Integration tests are test cases that group microservices together into a partial subsystem, and test the collaboration among these services. Finally, end-to-end tests are test cases that test the system as a whole. In contrast to unit tests, integration tests and end-to-end tests are especially useful when finding interaction-related faults.

A common approach to assessing and augmenting a test suite is to use code-coverage measurement; however, there exists no research on investigating how code-coverage information can be used to assess or improve a test suite for a microservice system. In particular, there are different types of coverage that can be collected from running a software system. For testing a service-based system such as a microservice system, high-level coverage can include service coverage, and low-level coverage can include line coverage. Service coverage is lightweight in terms of runtime overhead, and simple to collect for the system under test. In contrast, line coverage provides a finer granularity of coverage information but with higher runtime overhead. In the research literature, no research exists to study the measurement of code coverage (achieved by a test suite) such as service coverage and line coverage of a microservice system or the usage of such coverage measurement to augment the test suite.

To fill this gap of lacking studying code-coverage measurement for a microservice system, in this thesis, we conduct an empirical study on measuring code coverage achieved by an existing test suite for an open-source benchmark microservice system, TrainTicket [3]. The test suite contains 21 end-to-end test cases provided by the developers of TrainTicket.

During our study, we analyze the code-coverage information gathered before and after the test suite is run to gain insights, especially those of special interest and unique to testing a microservice system. Based on the code-coverage information, we explore ways of augmenting the existing test suite to achieve higher code coverage. In particular, one insight that we gain is that a significant portion of the lines not being covered are due to that some functionalities of the system are not exercised by the existing test suite. To exercise these missing functionalities, we augment the existing test suite by adding new test cases to aim for covering these functionalities. In addition, via analyzing the line-coverage information after the test suite is run, we find that another significant portion of the lines not being covered are blocked by error-handling branches (i.e., branches related to error handling). To execute error-handling branches, we propose a new approach to augment the test suite by altering the test-execution environment, by mutating the messages being sent by a microservice to a dependent microservice. Doing so can help execute the associated error-handling branches within the service receiving the altered message. In the end, using code coverage to guide us to augment the existing test suite, we are able to achieve higher code coverage compared to the coverage achieved by the original test suite.

1.1 THESIS GOALS AND CONTRIBUTIONS

The goals of this thesis are to fill the gap of lacking code-coverage studies when testing a microservice system, by (1) developing an infrastructure of code-coverage measurement for a microservice system; (2) using this infrastructure to gather code-coverage information of the TrainTicket system after running its existing test suite, and using this information to augment the existing test suite to achieve higher code coverage; (3) gaining insights into why certain code portions of the TrainTicket system are not covered by its existing test suite.

In summary, this thesis makes the following main contributions:

- Develop an infrastructure for measuring code coverage including service coverage and line coverage for a microservice system.
- Conduct a study by measuring code coverage achieved by the existing test suite for TrainTicket, a benchmark microservice system.
- Augment the existing test suite by adding additional test cases to cover not-covered functionalities, as guided by the collected code-coverage information.
- Augment the existing test suite by manipulating the test-execution environment to execute the not-covered error-handling branches, as guided by the collected code-coverage information.

1.2 THESIS ORGANIZATION

The rest of the thesis are organized as follows. Chapter 2 introduces the background to our study. Chapter 3 presents related work that also uses the TrainTicket benchmark system. Chapter 4 presents the design of the TrainTicket benchmark system on multiple levels from the single-service level to the entire system, along with the study setup. Chapter 5 presents the details on the study results and analysis of the study steps. Chapter 6 concludes the thesis.

CHAPTER 2: BACKGROUND

This chapter presents background information for the empirical study, including microservices, comparison of microservice and monolithic architectures, debugging of microservices, testing of microservices, system testing, common faults in microservice systems, and code coverage in microservices.

2.1 MICROSERVICES

In this section, we first discuss background information on software architecture and the service-oriented architecture, and then on microservice architecture.

2.1.1 Software Architecture and Service-Oriented Architecture

Software architecture, as defined by the IEEE Standard 1471-2000 [4], is the fundamental organization of a system embodied in its components, their relationships to each other and to the environment, and the principles guiding its design and evolution. The intended goal of the definition is to highlight that there are differences between architectural descriptions (e.g., concrete artifacts and documentation) and the conceptual abstract concept (i.e., an architecture).

The Service-Oriented Architecture (SOA) is motivated by increased attention to the principle of separation of concerns. The principle recommends separating programs into sections each of which addresses a specific concern [5], and is a design methodology that treats pieces of software as components. In SOA, components are used to realize a specific functionality, and are self-contained. These components are denoted as *services*. The logic of each service is a black box from the point of view of its dependent services, and a service may contain more than one service inside of it [6].

2.1.2 Microservice Architecture

The microservice architecture is a service-oriented pattern that supports the use of several small loosely-coupled services. It strives to remove complexities from traditional SOA by demanding that each service be completely independent in development and deployment. In other words, each service should be implemented in isolation and should be in an independent process [5]. An example microservice architecture is an e-commerce web-service system that sells home goods. A functionality in the system is to allow the user to search and order a home good. To realize this functionality, a login service, an order service, and a search service are provided by the system. The microservice architecture uses smaller services, and can improve the modularity of the service since there is less logic inside of it. If the order service in the e-commerce system is being heavily requested, it is more efficient to duplicate the logic of only the order service rather than duplicating other portions that are not being used heavily.

The opposite of the microservice architecture is known as the monolithic architecture. The monolithic architecture of a system relies on a methodology that supports a minimal number of services along with the services being tightly coupled. Thus, each service has several different functionalities that are usually tightly coupled since they are in the same service. If the e-commerce system is built using the monolithic methodology, it would be more computationally expensive to duplicate itself since it would need to duplicate multiple functionalities even when many of the services may not be used extensively. There are also usually multiple teams and individuals working on a monolithic system since it has a larger amount of logic. Microservices allow smaller development teams to each focus on a certain separate functionality in the system rather than having every team work in the same large code base.

The higher level of modularity brings the higher number of dependencies that a microservice needs to rely on. Since the functionality of the microservice is generally supposed to be small, the implementation for the functionality provided by a (micro)service would not be able to be all fit into one microservice. Thus, there is a need of communication between microservices that need to request information from other microservices. Suppose that the e-commerce system has a login service. This login service could be dependent on multiple microservices based on its functionality. It may need to have a way to process multiple different types of logins such as a user-name/password checker and a single-sign-on feature. To follow the microservice methodology, these features would need to be split into multiple microservices, and there is usually one microservice that is considered as the gateway the relies on requesting information from the other microservices to satisfy the login functionality.

2.2 MICROSERVICE ARCHITECTURE VS. MONOLITHIC ARCHITECTURE

There are various benefits and drawbacks of choosing a specific architecture to use. As stated earlier, the microservice architecture promotes the use of small loosely-coupled services used for specific functionalities to create an system highly reliant on the communication of these services. The monolithic architecture is an older architecture that was the most popular

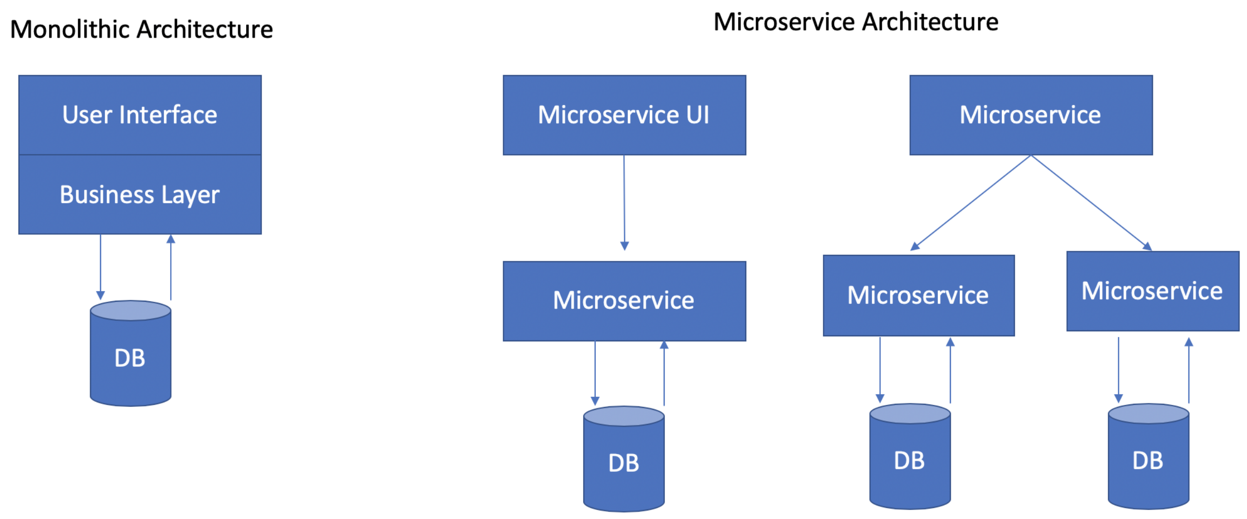


Figure 2.1: Microservice vs. Monolithic Diagram [7]

architecture before the boom of cloud computing. The monolithic architecture supports the use of a single system that encapsulates all of the system's functionalities. Figure 2.1 shows a diagram to help understand the key differences between the microservice and monolithic methodologies underlying the microservice and monolithic architectures.

2.2.1 Benefits and Drawbacks of Monolithic Architecture

There are various benefits of choosing the monolithic architecture. The systems adopting the the monolithic architecture are simple to develop since there is no distributed aspect to consider, and the communication between functionalities in a monolithic service has a simpler and faster way of communication since it is all encapsulated in one service [7]. For a monolithic service, collecting code-coverage information could be more easily done, and there is no distributed aspect to worry about when trying to equip tools to the service system. It is also simpler for the developers to deploy and scale the service system since there is only one service to deploy and duplicate if needed.

There are also various drawbacks of choosing the monolithic architecture. These drawbacks are usually related to the performance and resource usage of the system adopting the monolithic architecture. When developers create a complex service system adopting the monolithic architecture, its monolithic service has a high likelihood of becoming too large and complex to understand during the debugging process. Developers need to understand several portions of the system before making changes since a fault in a certain portion of the service system can cause the entire system to crash. The performance of the system can be compromised because of the need for the entire service to be up and running before a request can be made. To scale a certain portion of the system, the entire service needs to be duplicated, being very inefficient depending on the size of the system.

2.2.2 Benefits and Drawbacks of Microservice Architecture

There are various benefits of choosing the microservice architecture. The microservice architecture allows an easier development process since the services there are more independent. The teams that are working on each microservice do not necessarily need to communicate with other microservice developers within the same system if the endpoints are documented properly. The microservice architecture reduces the issue of adding a new technology since a microservice is completely independent of other microservices, and the microservice architecture allow developers to take more risk. As discussed earlier, the scalability of microservices can be accomplished efficiently since there is no need to duplicate other system portions not needed by the microservice that is receiving heavy traffic.

The drawbacks of choosing the microservice architecture are very similar to the drawbacks of any distributed system. The development of the service communication has become easier by the addition of many new microservice frameworks. However, it is still a functionality that the developers need to consider when writing code and making assumptions about the results from the communication of microservices within their system. As discussed throughout this thesis, testing a microservice system is far more difficult, and it is also more difficult to conduct accurate measurement of code coverage during the testing of the microservice system because of its distributed nature. Code-coverage measurement tools need to be equipped to every service individually, and the complexity increases when there are duplicate instances of a microservice. It is also more difficult to make changes to add a functionality that requires the changing of more than one service. Such code evolution may require the communication across multiple teams, and could have been done by one team if the entire system is built in a monolithic manner. The deployment of the microservice system also faces more challenges since sometimes developers have to deploy hundreds of different services depending on the size of the system. Indeed, containers and container-management tools are adopted to facilitate dealing with these issues.

In summary, there are various benefits of using the microservice architecture over the monolithic architecture but indeed the monolithic architecture tends to work better for smaller systems, while the microservice architecture tends to work better for large systems that can function in a distributed manner.

2.3 MICROSERVICE MONITORING TOOLS

Currently, there are three common approaches to monitoring microservices in a production setting [6]. The first is using basic log statements, and analyzing the collected logs using different tools to monitor the system if a failure occurs. It is the most basic type of monitoring tool, and is the simplest for developers to add to their system. Although it is the simplest and fastest to use for developers, it is still the most difficult to use when analyzing the logs of a large microservice system. Given that microservice systems are distributed, there is no centralized clock for each of the services to connect to. Therefore, there is no guarantee about the ordering of logs in a microservice system. The developers may add too many log statements, and it would be difficult to find the suspicious logs if the communication between services is high. On the other hand, there may be a lack of log statements, making it more likely that the suspicious area would not be found through the logs. Therefore, developers would lean more toward adding more log statements so that there is a higher probability of the fault being found in the logs. An example of a tool used for logging in Java is log4J (https://logging.apache.org/log4j/2.x/). Using logging alone tends to be less useful when dealing with errors that involve the communication between multiple microservices.

The second common type of tool used for monitoring microservices is a visual logging tool. It is not the simplest one for monitoring in terms of setting up the tools. But it makes searching through the logs a simpler task for developers. An example tool stack of this type is the "ELK" stack. The E stands for Elasticsearch, and is used for indexing and "stashing" the logs [8]. The L stands for Logstash, and is a server-side data-processing pipeline that ingests data from several different sources, and receives the logs from multiple microservices in our context [6]. The K stands for Kibana, and Kibana enables users to have a visual representation of their data in forms of charts and/or graphs in Elasticsearch [8]. So the logs are sent to the Logstash pipeline, Elasticsearch is used to index and search through the logs, and Kibana is used to visualize these logs.

The third common type of tool used for monitoring microservices is visual tracing analysis. Visual tracing analysis has been known to be the most helpful for developers when trying to find faults that occur when multiple microservices are communicating with each other. The tracing tool shows a visual representation for displaying the order in which each microservice is invoked. In this way, if a microservice fails because of an input received by a requesting service, it is relatively simple to find the error by sifting through the request trace. Developers can add this tool to their system by using the same ELK stack (as in the visual logging tool) and also using a distributed tracing system such as Zipkin [9] or Jaeger. In our empirical study, the microservice system under study is equipped with Zipkin.

2.4 TESTING MICROSERVICES

One of the options for testing microservices is to create unit tests for functionalities within each service. Unit testing is a level of testing where individual components are tested [10]. Therefore, unit tests are usually made for one specific microservice, and each microservice may have multiple unit tests. Unit tests are usually effective in finding logic faults in smaller units of code since unit tests are used to test single components. These types of tests are usually run without the need for the entire system to be deployed.

Another option for testing microservices is to create end-to-end tests (in end-to-end testing, which we elaborate in the next section). Systems tests are used to test the communication between services so that they do not end up creating faults. It is a higher level of testing, and usually invokes multiple microservices. End-to-end tests are effective in finding faults related to invalid interaction between services. These faults are the hardest types of faults to debug in microservice systems because of the large number of (micro)services. It is also because there are so many request traces that could occur during the running of the system. End-to-end tests require that the entire system has been deployed, because end-to-end tests are simulating how a user would interact with the system.

2.5 END-TO-END TESTING

End-to-end testing (based on end-to-end tests) is a testing approach that is used to make sure that the system as a whole meets certain goals [**?**]. Automated end-to-end testing tools can be used to simulate how a client would interact with the system under test. In particular, end-to-end tests can be created by using an end-to-end testing tool such as Selenium [11]. Selenium is a tool that can be used to automate browser interaction, and can be used to mimic how a user would interact with a web application. One can create tests for each of the expected use cases that a user is expected to use the system. One advantage of this testing process when creating these simulations manually is that one can make sure that the code portions and use cases to be tested are invoked during the process. One disadvantage is that one is not able to create these test cases without sufficient domain knowledge. It takes a certain level of understanding of the system under test, and also takes more time to create the simulations of these use cases. For example, there are most likely going to be hundreds of use cases if one is testing a banking system, and one needs to understand each of these use cases before creating the test cases to simulate each of these use cases.

2.6 COMMON FAULTS IN MICROSERVICE SYSTEMS

Although there are several benefits developers find when using the microservice architecture, there is an issue of your system being more vulnerable to certain common faults. A fault in terms of software is a flaw where the results that are found dont correspond to the expected results [12]. Functional faults are a type of fault where services malfunction, and the malfunctioning may cause other microservices to fail. Non-functional faults relate to faults that cause the reliability and performance of the services to go down [6]. There are three categories that are important when comparing microservice faults. Internal faults occur because of errors and flaws in the internal implementation of a service. Interaction faults occur from an error in the interaction and communication between multiple microservices. Environment faults occur when there is an error in the configuration of the infrastructure that the microservices are deployed on.

As discussed previously, internal faults are the easiest to be found since these faults are not caused by the communication of the microservices. Therefore, they are simpler to pinpoint during debugging, and testing through unit tests. The rule for internal faults that make them different from interaction-based faults would be that each service that is called is in the same system, and communication is synchronous [13]. Using basic log statements intelligently should be enough to find these internal errors since they are specific to one machine, and the logging order would be guaranteed as correct since all of the logs are synchronous in the same machine.

The main type of fault that is difficult to debug in microservices are interaction-based faults. The number of services in large systems cause it to be relatively impossible to understand all of the paths of communication that occur in the production systems. Asynchronous interaction faults and multi-instance faults are two types of faults that occur because of miscommunication between multiple services [13]. Asynchronous interaction faults are found when there are multiple asynchronous requests made concurrently, and the responses are returned in an unexpected order. The unexpected order may cause a functional fault in the service where the asynchronous requests were made. Testing all possible orders of these asynchronous requests would be the most complete manner to test for asynchronous interaction faults, however that is infeasible in most production environments because the number of paths grows exponentially as more services are added to the system. Multi-instance faults occur when there is a miscommunication between multiple of the same type of microservice. For scaling purposes, it may be beneficial to duplicate and load-balance a service that is being requested at a high rate. But if the communication between the gateway and the microservices is flawed, then a multi-instance fault may occur. Using service coverage which identifies different containers may be useful when trying to achieve a way to test for multi-instance faults.

2.7 CODE COVERAGE IN MICROSERVICES

Code coverage is the measurement of code that has been covered by the automated or manual test cases. Code coverage is shown as a report at the end of running the test cases. Depending on the level of coverage that is run, the report will show statistics about the coverage. There are different levels of code coverage, and each have their use cases. A higher level of coverage in terms of microservices would be service coverage. The lowest level of coverage would be line. There are other types of coverage such as class level, and function coverage that have benefits as well. Code coverage measurements are usually used to improve the quality and robustness of your test cases. The higher the percentage, the more high quality your test cases are. There may be areas of code that are not covered in your test cases that could cause a fault in a production setting when these instructions are invoked. This is why gaining more code coverage in your test cases can be correlated with a more high quality test suite.

Service coverage would measure the number of services touched by the test cases compared to all of the deployed services. The equation to calculate service coverage is given below. Since it is a higher level of coverage, it is difficult to gain more insights into the inner functionality of a service during the running of test cases. It is fairly simple to gather this level of coverage from existing tools such as distributed tracing tools. You can parse the data that is kept in the Zipkin (distributed tracing tool) logs to find how many unique services were called throughout the testing process. The simplicity of collecting service coverage makes it useful and easy to gather, however it reveals the a little amount of information regarding the quality of the test cases when compared to line coverage. But keep in mind that there are certain scenarios where line coverage is not possible. Production level systems would suffer a performance hit if they were to equip each service with a line coverage collection tool. Service coverage information can also be inferred if a system is already gathering a finer granularity of coverage.

$$ServiceCoverage = ServiceCovered/TotalServiceCount$$

Line coverage is known as one of the lower levels of coverage, and is useful in understanding the percentage of lines covered within a particular service, or code base. The equation below shows how line coverage is calculated. It is taken by dividing the number of lines covered by the test cases of a system divided by the total number of lines in the system. The equation is used to gain overall system coverage, but this could be fine tuned to gain line coverage of a single service, a class, a function, and so on. Although it leads to a much higher understanding of the quality in your test cases, it is more difficult to implement and can be much more computationally intensive depending on the tool that is used to gain the coverage. Tools that allow for line coverage are usually language specific, and used mainly for measuring coverage in unit tests. However, coverage can also be measured after the deployment of the system as well. Tools such as Jacoco for the Java language allow for on-the-fly coverage, which means it gathers coverage information while the system is running without the need for automated test cases.

$$LineCoverage = LinesCovered/TotalLinesOfCode$$

The tools that are needed for line coverage are different depending on what type of testing is being done. For unit testing, you could use a language specific tool to run the test cases along with gathering coverage information. However, gathering coverage information for a microservice system using system tests is different. First, you need to find a tool that would allow for continuous coverage without the need for automated unit tests. Second, you need to find multiple tools since a microservice system can be a polyglot system. Most coverage tools are language specific, and this is why you would need multiple in the scenario where your system does indeed use multiple languages.

CHAPTER 3: RELATED WORK

This chapter discusses previous work related to the empirical study in the thesis, with focus on two lines of previous work that uses the same microservice benchmark system, TrainTicket, as our study does.

3.1 FAULT ANALYSIS AND DEBUGGING OF MICROSERVICE SYSTEMS: INDUSTRIAL SURVEY AND EMPIRICAL STUDY

Zhou et al. [14] conduct fault analysis and debugging of microservices through an empirical study influenced by an industrial survey. They create an industrial survey sent to software practitioners to fill out regarding their experience with microservices. Based on the survey results, Zhou et al. share the difficulties faced by practitioners when they transition their systems from the monolithic architecture to the microservice architecture. Zhou et al. also show particular faults faced by practitioners at their companies. With this information, Zhou et al. decide to reproduce each of the 22 collected fault cases into their newly created open-source microservice system, TrainTicket. They also go into detail about the current approaches to debugging microservice systems along with the pros and cons of using each approach.

Zhou et al. discuss three types of debugging practices: basic log analysis, visual log analysis, and visual trace analysis. They investigate the impact of using visual tracing compared to basic log analysis by observing the time that it takes for one to find one of the 22 fault cases using one of the debugging practices. Their study shows that visual tracing is useful in finding multiple types of faults in a timely manner and these types of faults cannot be found using simpler debugging practices such as basic log analysis. Basic log analysis is effective when debugging internal faults, while being ineffective in debugging interaction faults. This finding is expected since logging in a distributed system has no guarantee of being correct since there is no synchronized clocks between the services. Here visual trace analysis is useful since it gives the developers much more information in regards to the interactions between the services. Their study shows that developers using distributed tracing tools to debug find faults in much more efficient time when dealing with interaction faults. Their study shows that the process of debugging most types of faults could benefit from using of a visual tracing tool.

Zhou et al. also contribute the development of the TrainTicket system along with seeding faults in it. At the time of the creation of this microservice system, there were no other open-source systems with a relatively large number of services as TrainTicket including 72 microservices. Although 72 microservices are not as many as systems such as Twitter and Uber, which rely on hundreds or thousands of microservices, TrainTicket is still large enough to simulate a number of industry fault cases.

3.2 DELTA DEBUGGING

In their related efforts, Zhou et al. [15] conduct delta debugging on a microservice system, particularly the TrainTicket system. The original use case of delta debugging is for systems that use the monolithic architecture. The main type of faults focused by Zhou et al. is environmental faults, such as issues in the resource allocation of the run-time environment. Suppose that a service requires 400 MB of memory, but the container is only provided 200 MB. This configuration could induce a fault since the container does not have the resources to run the source code properly. For microservice systems, Zhou et al. design the delta-debugging algorithm to minimize the initial failure-inducing deployment configurations to minimal ones.

Zhou et al. make their solution scalable by parallelizing the delta debugging process. With their infrastructure support, multiple test cases can be run on the TrainTicket system with several different infrastructure setups. Then the delta-debugging algorithm can efficiently dig and find faulty infrastructure setups in a timely manner. Their solution requires to have a powerful enough infrastructure to allow the test cases used in delta debugging to be run concurrently. The findings from their evaluation show that their delta-debugging algorithm is able to efficiently identify minimal failure-inducing configurations.

CHAPTER 4: STUDY SUBJECT, SETUP, AND METHODOLOGY

In this chapter, we discuss the study subject, study setup, and study methodologies for our empirical study on measuring code coverage achieved by an existing test suite for an open-source benchmark microservice system.

4.1 TRAINTICKET SUBJECT

For the purposes of our study, we choose an open-source benchmark microservice system, TrainTicket [3], as our study subject. TrainTicket is a railway ticketing system that uses microservices for all of its functionalities. Example functionalities of this system include booking a train ticket, paying for the ticket, and redeeming the ticket. TrainTicket has 41 business-oriented microservices, being more than any other open-source benchmark microservice systems at the time of its release. In total, the TrainTicket has 72 services. Therefore, there are 31 services that are not created by the TrainTicket developers but are also deployed for the TrainTicket system.

Services in the TrainTicket system share the same service design. Each service is created and deployed as a container. A container is a piece of software that takes code, packages it, and deploys the code along with all of its dependencies to allow the software to run properly [16]. Inside each container includes Java source code, the Jacoco runtime agent attached to the JVM provided by the jre1.8 Docker image, and a gateway for load balancing in case there are duplicates of a specific service. The container is connected to the host's OS network, which allows the container to expose itself to a port on the host's machine. This mechanism allows for the container to communicate with network protocols, most likely being the HyperText Transfer Protocol (HTTP). When a microservice makes a request to a service, the microservice uses the service's domain name along with the method that the microservices would like to invoke in the service.

Figure 4.1 shows an infrastructure-level diagram of the TrainTicket system. The lowest layer is the host server infrastructure, which is the bare bone of the host server. The next two layers are the host server OS and the OS of the virtual machine running on the host server. The next layer is the Docker daemon, which can be used to manage the cluster. The functionality offered by the Docker daemon allows to deploy all services necessary for the TrainTicket system to function. The highest layer is the containerized services, where there are 72 service containers in total when the TrainTicket system is deployed.

The TrainTicket system also provides traffic monitoring, measurement, and management

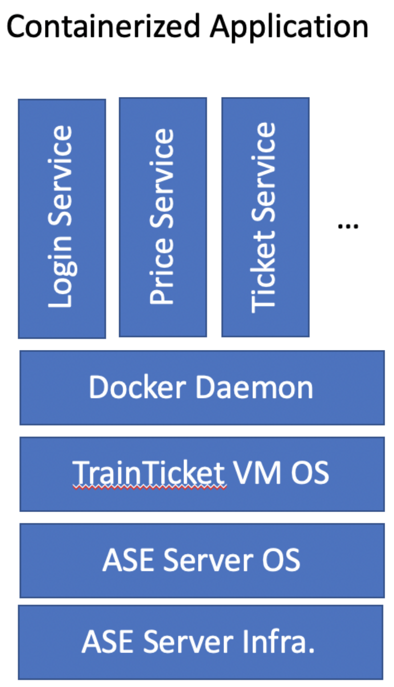


Figure 4.1: TrainTicket's Infrastructure-Level Diagram

through the use of two open-source tools: Zipkin (https://zipkin.io/) and Istio (https://istio.io/). Zipkin is a distributed tracing tool that uses logging information to gather and create distributed service-level traces. As discussed in Section 2.4, the distributed tracing tool facilitates debugging interaction faults. The developers/testers are able to view the service coverage of actions made on the microservice system along with a time-based visualization of when the services are invoked along with how long they take to respond. Zipkin also allows the developers/testers to view what are sent inside of the requests and responses between services. For microservices that communicate over HTTP, Zipkin allows the developers/testers to look inside of the GET requests and responses.

The TrainTicket services are deployed using Docker (which is a program that does OS-level virtualization to create independent containers for each of the services), and the TrainTicket containers can be managed and deployed using two different tools: Kubernetes [17] and Docker Compose [18]. Kubernetes [17] is an open-source tool for automating deployment, scaling, and management of containerized systems. Docker Compose [18] is another open-source tool used for the management of containerized systems. Docker Compose allows to spin up all of the images as containers, and also allows to duplicate these images into multiple running container instances. Docker Compose uses the discovery service along with the load-balancing service to allow requests to be evenly distributed if there are multiple running instances of the same image.

The TrainTicket system mainly uses the Spring Boot (https://spring.io/projects/spring-boot) for the framework of developing its microservices. The main language of the majority of the microservices is Java. Other used languages are Node JS, GoLang, and Python. Each of the services supports load balancing and other infrastructure logic to enable microservice scaling. If there is a time where multiple users may need to book a ticket and the ticket-reserve service is being overloaded, the use of microservices will allow only this microservice to be scaled to meet the needs without unnecessarily duplicating other microservices. Docker Compose's cluster management simplifies the process of manipulating the test infrastructure.

4.2 SYSTEM RESOURCES

We deploy TrainTicket on a Linux-based virtual machine with the configuration of OS: Ubuntu 16.04.6 LTS (Xenial Xerus); CPU: 16 cores; Memory: 32 GB. Note that to deploy the TrainTicket system, we need to reserve at least 200 MB of memory for each of the microservice containers. Since there are 72 microservices being deployed, we need a minimum memory of 14.4 GB for the virtual machine.

In our study, we use a client machine with the configuration of OS: macOS Mojave 10.14.2; CPU: 4 cores; Memory: 16 GB. Note that as long as a client machine can support a Chrome browser and Java IDE such as IntelliJ Idea to run the Selenium test cases, the IDE is expected to meet the requirements. Also, the Selenium Web Driver and the Chrome web driver shall be installed in the client machine to fulfill Selenium's dependencies.

4.3 COVERAGE-MEASUREMENT INFRASTRUCTURE

The open-source TrainTicket system does not provide tools to collect fine-grained code coverage such as line coverage. To carry out our empirical study, we add services and tools to collect line coverage besides service coverage. In particular, we add a Maven service to the cluster to allow maven commands to be made on all of the other containers. We need Maven since the code-coverage tool that we use requires a Maven plugin to dump the collected coverage information to the virtual machine. We also add another service, the SonarQube [19] service. We need SonarQube to aggregate and visualize the collected coverage information.

To collect the coverage information accurately, each container needs to be equipped with a language-specific tool that can gather coverage information inside a deployed system. Since there are 38 microservices (out of the 41 microservices) that are based on the Java language, we equip all of the Java-based microservices with Jacoco. We choose not to cover the other languages since the amount of coverage information from the remaining 3 microservices would be negligible, and some of their used languages also do not have tools that could perform the coverage collection during an end-to-end testing scenario. The Jacoco runtime agent connects to the Java Virtual Machine (JVM), and collects code-coverage information by dumping the information to the destination of our choice upon the request of a script used by us. The time that it takes for the dump request to complete (from each Jacoco runtime agent to our virtual machine with TrainTicket deployed) is about 12 seconds. It then takes 10 minutes to send all the information to our centralized code-coverage visualizer service.

We use SonarQube, a coverage visualization service to visualize the coverage information collected from the Jacoco runtime agent. SonarQube allows all of the services to communicate their code-coverage information, and each service is displayed as a project in the web service provided by SonarQube. One can dive into each service project to view the covered lines, and SonarQube also has an API that we use to collect the coverage information from the projects. The benefit of SonarQube is that all of the coverage information can be found in one centralized service. The SonarQube web application is particularly useful when one tries to manually observe the not-covered lines after the test cases are run. Any source code that has a green bar next to its line number of the source code indicates that the test cases or deployment of the TrainTicket system have covered that portion of the code base. In contrast, a not-covered line has a red bar next to its line number. SonarQube also allows the Java source code written and gathered from Jacoco to also be run through SonarQube's special language-specific bug finder. SonarQube shows certain areas in TrainTicket's source code that may be unsafe from a security perspective, along with other types of bugs.

4.3.1 Existing Test Suite

To conduct the manual testing, one can use TrainTicket's web page UI to interact with the services. For example, one can carry out the process of booking and paying for a train ticket. To support automated end-to-end testing, TrainTicket developers have written a test suite for simulating functionalities visible to the users. Running the test cases in the test suite relies on the Selenium testing tool, which can simulate a user's walking through the TrainTicket UI in the web client.

There are 21 test cases in the test suite, falling into two types of test cases. Although these test cases can all be considered end-to-end tests, there is a functionality added to the front-end but not available in a normal microservice system. The first type is a single-service test case, which invokes only one service through a request made from the web client. But there is no guarantee that only one microservice is invoked when running this test case since this microservice can have dependencies on other services. For example, the basic information service has dependencies on the configuration, station, train, contact, and price services. However, there are also services that do not have any dependencies, and for such situation, system test cases can be seen as simply unit test cases. These test cases are useful when trying to gain the most possible line coverage information since the user has a much clearer understanding of what is being directly sent to the microservice during the request.

The second type is a flow test case. The flow test case simulates how a typical user interacts with the TrainTicket system. The first flow system test case in the test suite goes through the process of logging in, searching for a ticket, reserving the ticket, paying for it, collecting it, and then entering the station with the collected ticket. The test case interacts with multiple services, and requires a response from multiple microservices. A flow test case is typically more comprehensive than a single-service test case since a flow test case simulates how a user interacts with the system, and can typically achieve higher code coverage because of the requirement of responses from multiple microservices.

Note that the TrainTicket developers had not actively run the test cases in the test suite since they initially open-sourced the TrainTicket system. In particular, assertions in many of the test cases are written for an older version of TrainTicket and not valid anymore for the latest, open-source version of TrainTicket. Some test cases are written to test deprecated functionalities that are missing in the the latest, open-source version of TrainTicket. We have to make multiple changes to the source code of the TrainTicket system in order to allow the test cases to pass.

In particular, we notice two main issues in the test cases from the test suite and have to fix these issues. First, all of the test cases are time dependent. If the machine with TrainTicket deployed is a bit slower than what the test cases expect, the delay can lead to many false-positive failures of test cases, which would pass if the assertions would have been given a bit more time to receive the expected results. An example of this issue is in the test case that goes through the flow of a user's searching for a train ticket, booking it, paying for it, and collecting the ticket. The test case originally waits for 2 seconds before checking that the user is logged in. However, our machine takes approximately 3 seconds to finish the login process, and the test case is flagged as a failure although the login part does work properly. To fix this issue, the time given by the test case for waiting should be extended to 5 seconds. For certain requests that may have taken over 30 seconds, another possible solution is to poll for the assertion every second for one minute. A case where the polling mechanism is necessary is for the cancel service since it takes a variable amount of time to cancel a ticket purchase.

Second, the test suite includes invalid test cases, which most likely are not run on the system for quite some time. The logic of certain services is no longer compatible with the expected mechanism reflected in certain test cases. For example, the basic information microservice has a test case that simulates a user's searching for a ticket with a higher number of parameters when compared to the normal user search. The normal search mechanism asks for the user to fill in the starting place, the terminal place, the date of departure, and the train type. The basic information service previously should have been able to respond to requests with higher precision based on the higher number of parameters. The parameters for the basic-service request are the trip ID, train type ID, starting station ID, middle station ID, terminal station ID, starting time, ending time, starting place, ending place, and departure time. The new version of the basic service does not have the logic to support the query with the higher amount parameters given from the web client, causing the service to crash when it receives too many parameters. To fix this issue, we have to re-implement the logic that has been deprecated in the basic information service to allow for the basic service test case request to be given a valid response, and not crash the entire search functionality afterwards.

4.4 STUDY METHODOLOGY

We take the following steps to attain the study results presented in Chapter 5:

1. Deploy the TrainTicket system in a clean state using Docker Compose, and do not run or interact with the system to make sure that we gather accurate code-coverage measurement of the system.
2. Wait for approximately 10 minutes for the services to be initialized, and then gather the code-coverage information from each Java-based microservice.
3. Send the code-coverage information to SonarQube to allow visual analysis of the line coverage after deployment.
4. Run the TrainTicket's existing test suite: first run the single-service test cases to make sure that the single-service test cases pass before running the more complex flow test cases; then run the flow test cases afterwards to complete the running of all the 21 test cases.

5. After the test cases have completed running and passed, gather the code-coverage information from each microservice again.
6. Send the code-coverage information to SonarQube to allow visual analysis of the line coverage after the test cases are run.
7. Derive the service-coverage information by comparing the line-coverage information of a microservice before and after the test cases are run. If there are previously not-covered lines being covered after the test cases are run, the microservice has been covered by the test cases.
8. Create new test cases to achieve higher service coverage, by analyzing the microservices not covered by the existing test cases, and finding a way to invoke these microservices through the web client.
9. Create new test cases to achieve higher line coverage, with the following steps:
    - Remove dead code (code that is never reachable but left in by the developers).
    - Add new test cases for exercising functionalities not covered as reflected by not-covered lines.
    - Mutate the responses from the service's dependencies to allow the error-handling branches (previously not executed as reflected by not-covered lines) to be executed.

CHAPTER 5: COVERAGE RESULTS AND ANALYSIS

In this chapter, we present and analyze the results of code-coverage information collected from the TrainTicket system. We collect code-coverage information in three steps. First, we collect the code-coverage information after the deployment of the TrainTicket system. Second, we collect the code-coverage information after running TrainTicket's existing test suite. Third, we collect the code-coverage information after augmenting TrainTicket's existing test suite.

5.1 CODE-COVERAGE RESULTS AND ANALYSIS AFTER RUNNING TRAINTICKET'S TEST SUITE

In this section, we present the results from collecting code-coverage information from running TrainTicket's existing test suite. We collect code-coverage information before the test suite is run and after the test suite is run, respectively. The two types of code coverage focused in our study are service coverage and line coverage. Table 5.1 shows the line-coverage statistics categorized by individual services. Using the line-coverage measurement before the test suite is run and after the test suite is run, respectively, we can derive the service coverage. We find that 28 out of the 38 Java-based microservices are covered after running the test suite. The service coverage after running TrainTicket's existing test suite is 73.68%.

The overall line-coverage results of the TrainTicket system is also collected before and after running the test suite. When the test suite is not run, the covered lines are 1,506 lines out of 15,148 lines in total. The overall line coverage of the TrainTicket system after deployment is 9.94%. After the test suite is run, the code-coverage data is collected and aggregated to allow the overall coverage to be calculated again. The covered lines come out as 6,127 lines out of 15,148 lines in total after running the test suite. The overall line coverage of the TrainTicket system after running the test suite is 40.45%.

5.1.1 Analysis of Code Coverage After Deployment

An interesting finding discussed earlier is the observation of lines being covered before the test suite is run. To figure out the reason behind why there are certain lines being covered, we manually analyze each of the covered services to see whether there are common code areas being executed by each of the covered services. It turns out that the main reason why there are lines covered before the test suite is run is the initialization process of the Spring framework code. We next analyze the line coverage of the login service (other services

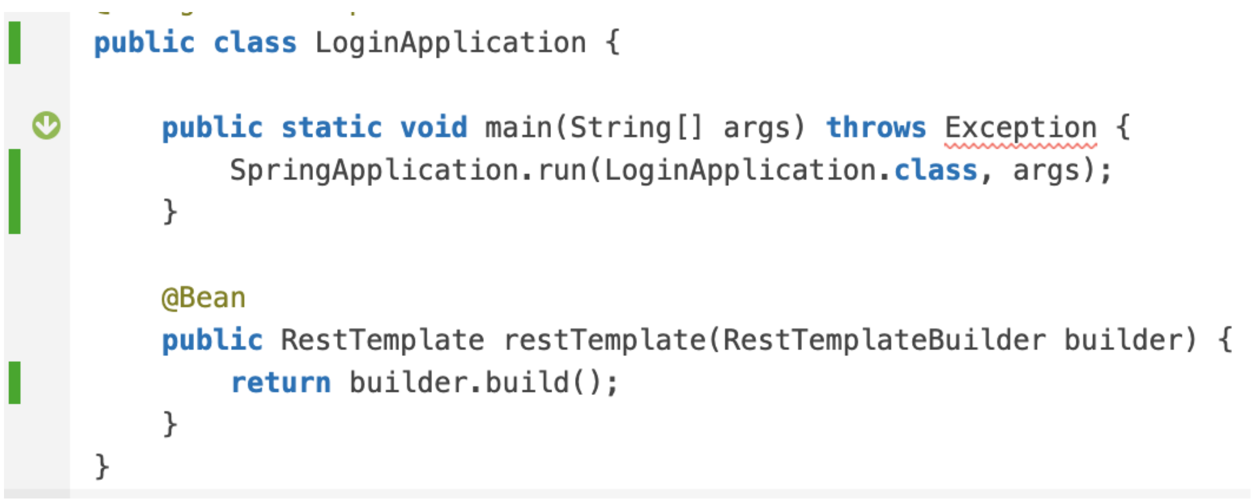
Service Name	Line Coverage (Pre-Test)	Line Coverage (Post-Test)	Difference
ts-admin-basic-info-service	1.30%	1.30%	0.00%
ts-admin-order-service	7.10%	7.10%	0.00%
ts-admin-route-service	3.90%	3.90%	0.00%
ts-admin-travel-service	2.20%	2.20%	0.00%
ts-admin-user-service	2.90%	2.90%	0.00%
ts-assurance-service	15.00%	15.00%	0.00%
ts-basic-service	1.40%	57.30%	55.90%
ts-cancel-service	1.10%	70.00%	68.90%
ts-config-service	53.80%	62.30%	8.50%
ts-consign-service	6.60%	10.80%	4.20%
ts-consign-price-service	55.10%	55.10%	0.00%
ts-contacts-service	34.40%	40.20%	5.80%
ts-execute-service	43.30%	53.40%	10.10%
ts-food-service	60.90%	60.90%	0.00%
ts-food-map-service	85.00%	85.00%	0.00%
ts-inside-payment-service	42.10%	60.60%	18.50%
ts-login-service	54.10%	55.00%	0.90%
ts-notification-service	41.10%	54.80%	13.70%
ts-order-service	37.10%	46.40%	9.30%
ts-order-other-service	10.80%	29.70%	18.90%
ts-payment-service	39.20%	69.60%	30.40%
ts-preserve-service	9.60%	59.10%	49.50%
ts-preserve-other-service	0.60%	0.60%	0.00%
ts-price-service	53.90%	64.90%	11.00%
ts-rebook-service	0.70%	44.30%	43.60%
ts-register-service	14.60%	82.90%	68.30%
ts-route-service	14.70%	61.40%	46.70%
ts-route-plan-service	1.20%	30.50%	29.30%
ts-sso-service	11.10%	57.80%	46.70%
ts-seat-service	64.80%	70.40%	5.60%
ts-security-service	32.70%	37.70%	5.00%
ts-station-service	62.20%	68.90%	6.70%
ts-ticketinfo-service	64.80%	69.80%	5.00%
ts-train-service	66.90%	76.60%	9.70%
ts-travel2-service	14.50%	26.70%	12.20%
ts-travel-service	64.70%	66.20%	1.50%
ts-travel-plan-service	1.20%	3.60%	2.30%
ts-verification-code-service	68.70%	70.10%	1.40%

Table 5.1: Microservice Line Coverage

covered after deployment are in similar situations). In Figure 5.1, there are two methods that are run in the login service after deployment. The main method is always run to create an instance of the Login Application class. This main method then calls Spring's run method, which is the initiation of the startup process for the microservice instance. This main method also creates a RestTemplate Bean. A bean is an object managed by the Spring Inversion of Control (IoC) container [20]. The RestTemplate is a client used to allow the microservice to make RESTful GET requests and responses. In summary, all of the lines of code that are run during deployment are the microservice's main method, and methods that are used to properly initialize a RESTful Spring-based microservice instance. We can see in Table 5.1 that there are no services with 0% line coverage. The line-coverage percentages are different among the services because of the difference in the initialization processes of a service's main class, and the number of lines is generally not the same in each of the services.

5.1.2 Analysis of Dependency-based Coverage

A hypothesis that we intend to study is how the dependencies of a microservice could effect its line coverage. Our hypothesis is that the number of dependencies would lead to lower


```
public class LoginApplication {

    public static void main(String[] args) throws Exception {
        SpringApplication.run(LoginApplication.class, args);
    }

    @Bean
    public RestTemplate restTemplate(RestTemplateBuilder builder) {
        return builder.build();
    }
}
```

Figure 5.1: Line Coverage of LoginApplication Class After Deployment of TrainTicket

coverage in the test suite. To do so, we count the number of dependencies that each service has, and collect the line-coverage statistics for each individual service. After we gather the data, we then create a box plot where the x-axis shows the number of dependencies, and the y-axis shows the line-coverage percentage. The box plot allows to group all of the services based on the number of dependencies that they have, and gain a clearer picture as to whether there is a correlation between the number of dependencies and the line coverage of a service.

While we create the dependency/coverage box plots, it makes the analysis more accurate when we remove the services not covered by running the test suite. After we remove these not-covered services, we attain the breakdown of how many services have a certain number of dependencies: 13 services have 0 dependencies, 3 services have 1 dependency, 3 services have 2 dependencies, 3 services have 3 dependencies, 1 service has 4 dependencies, 2 services have 5 dependencies, 1 service has 6 dependencies, 1 service has 8 dependencies, and 1 service has 13 dependencies.

The first step is to gather the line-coverage information of each service right after all the services are deployed. Figure 5.2 shows the box plot created with the coverage information after deployment when the services not deployed are removed. The highest average line coverage is in the services with 0 dependencies. The lowest average line coverage is found in the service that has 8 dependencies.

The second step is to gather the line-coverage information of each service after the test suite is run. Figure 5.3 shows the box plot with the coverage information gathered after the second step is completed. At the moment, there does not seem to be a correlation between line coverage and the number of dependencies after the initial test suite is run. But there are

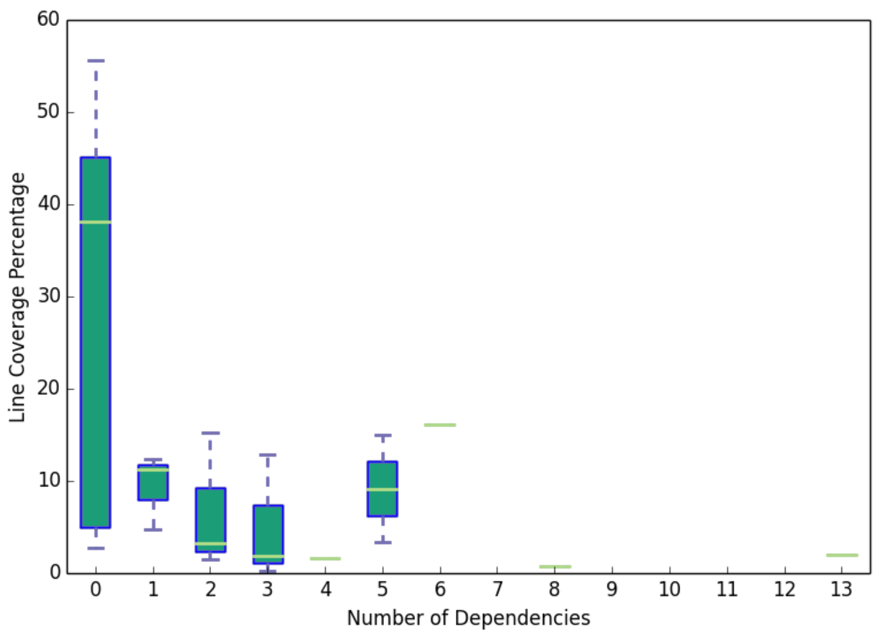


Figure 5.2: Dependency/Coverage Box Plot After Deployment

several issues that could skew this result. First, the number of services that are present in each dependency box plot is not uniform. The highest number of services in one dependency box plot is 13 while the lowest is 1 service in one dependency box plot. A more accurate result would be available if more services are assigned to each dependency category. Second, the initial test suite is not guaranteed to be of high quality, and there are still services with low line coverage after the test suite is run. The portions of not-covered lines that are due to dead code and missing functionalities may skew the results. Based on these issues, we cannot refute our initial hypothesis that there exists a correlation between line coverage found in a service and the number of dependencies in a service.

Since we do not augment every service in the third step of code-coverage collection, it would be inaccurate to create a box plot from the collected code coverage after augmenting the TrainTicket's test suite.

5.2 CODE-COVERAGE RESULTS AND ANALYSIS AFTER AUGMENTING TRAINTICKET'S TEST SUITE

To gain higher code coverage, we augment TrainTicket's test suite to make sure that all functionalities that are accessible to the client are run. For the TrainTicket system, 34% of the services that are not covered by the existing test suite are from the administrative services. There are five administrative services in the TrainTicket system, and there is a

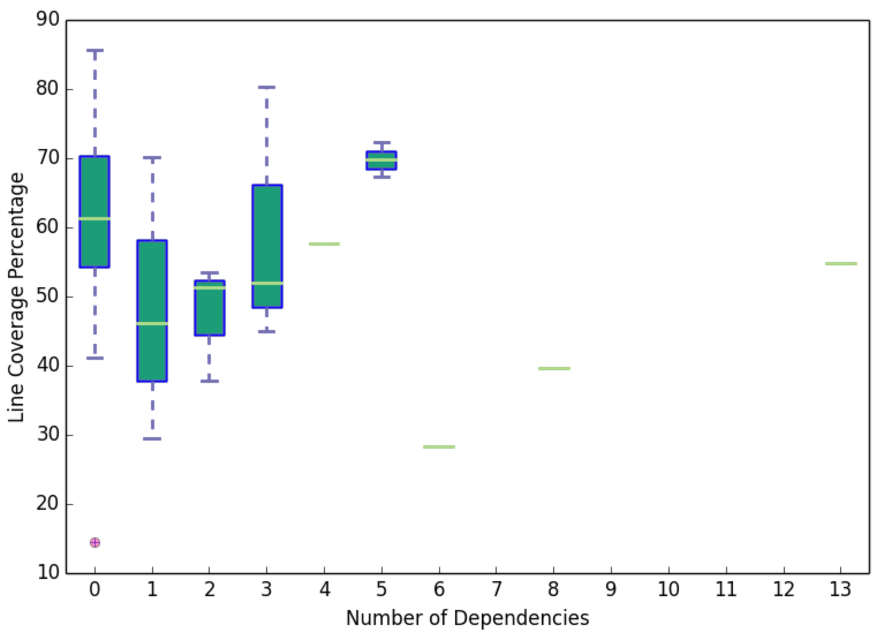
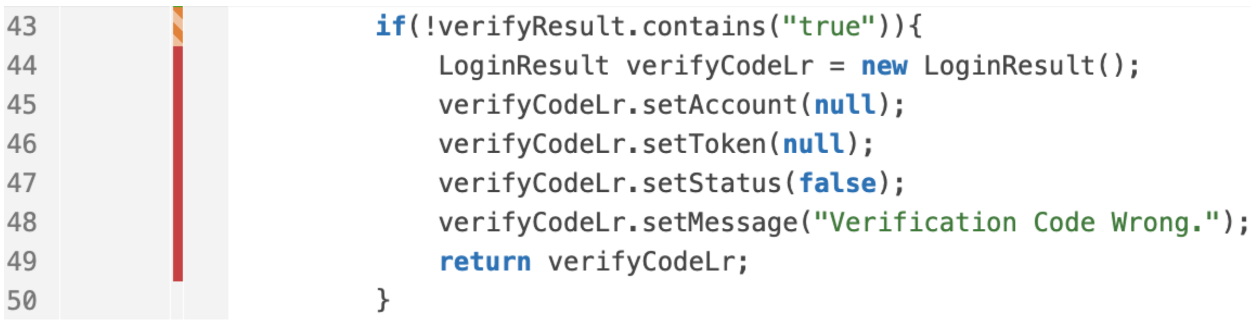


Figure 5.3: Dependency/Coverage Box Plot After Running the Test Suite

User Interface (UI) in the TrainTicket web system that allows these services to be covered. We create additional test cases that could cover these five services by interacting with them through the administrator web client. The service coverage increases from 73.8% achieved by the existing test suite to 86.8% after the augmented test suite (with the additional test cases) are run. The line coverage increases by 12.3% (52.72% line coverage) after the augmented test suite is run when compared to only the existing test suite is run (40.45% line coverage).

To augment the existing test suite based on the line-coverage data, we focus on three services rather than the entire system. The services that we choose to be invoked are the services that are invoked heavily throughout the process of running the existing test suite. We choose the login service and the basic-information service because they are invoked in every flow test case, and they also have many dependencies. We choose the ticket-station service since it has 0 dependency. We can use these 3 services to compare and gain insights into the difference of gaining coverage from a service with a high number of dependencies (login service, basic service) and a service with no dependency (ticket-station service). The first approach to increase line coverage is to remove dead code, and add new test cases for testing functionalities that are not covered by the existing test suite. After applying the first approach, the line coverage of the login service is increased from 55.0% after the existing test suite is run to 73.6%; the line coverage of the basic-information service is increased from 57.3% after the existing test suite is run to 63.4%; the line coverage of the ticket-station service is increased from 68.9% after the existing test suite is run to 93.4%. These increases



```
        if(!verifyResult.contains("true")){
            LoginResult verifyCodeLr = new LoginResult();
            verifyCodeLr.setAccount(null);
            verifyCodeLr.setToken(null);
            verifyCodeLr.setStatus(false);
            verifyCodeLr.setMessage("Verification Code Wrong.");
            return verifyCodeLr;
        }
```

Figure 5.4: Verification-Failure Conditional in Login Service

in line coverage are made without altering the testing infrastructure in any way. After using our approach of mutating dependency responses to execute error-handling branches within a service, we are able to gather the following line coverage. The line coverage of the login service is increased from 73.6% to 87.6%, and the line coverage of the basic service is increased from 63.4% to 81.5%. The line coverage of the train-station service remains at 93.4% since it does not have any dependency-based error-handling branches that can be executed.

5.2.1 Code-Coverage Analysis After Gaining Higher Code Coverage

The process of aiming to increase service coverage sheds light on the services that are particularly difficult to cover when testing the TrainTicket system. From the process of increasing service coverage, we find that the admin services are not invoked since there are no test cases that interact with the admin web client. Although service-coverage information could have been derived using line coverage, the approach of extracting the information from service coverage instead is much more lightweight on a system. In the TrainTicket system, it is particularly difficult to cover services that are used as dependencies, and not invoked directly from a functionality in the web client.

There are three insights gained after attaining more line coverage on the TrainTicket. We next focus on the login service since the insights gained through observing the login service are the same as the insights gained from the other two services. The first insight shows that the majority of the lines not covered by the existing test suite are blocked by error conditions. An example piece of code not covered because of lacking error cases is shown in Figure 5.4. Figure 5.4 is a screenshot from SonarQube, and shows a conditional used to check to make sure that the verification code is valid by making a request to the verification service. If the verification code is never received, a failure is caused in the login service. The red bar to the right of the line numbers shows that this portion of code is not covered. The screenshot is taken after the existing test suite is run on the login service. If the verification has a status of true, then the verification code is valid, and there is no need to run these lines of code. However, if the verification code is invalid, then these lines of code will be run as a manner of returning a rejection to the web page since the service is given invalid information and should no longer proceed in the login process. It turns out that the TrainTicket system has a fault where it returns true for any given verification code. It has been hard-coded on the server side to return true for any verification code, and the insight is made clear since none of the lines are triggered even after mutating the response message.

To gain more perspective on the first insight, it is useful to dive deeper to find the percentage of not-covered lines caused by lacking invalid-response test cases for the login service. In total, there are 58 not-covered lines in the login service. Through manually analyzing each of these lines, 21 out of the 58 not-covered lines are caused by not satisfying the error conditionals. In other words, 36% of the lines not covered in the login service are because of lacking faulty responses from its dependencies when the test suite is run. By understanding that the majority of lines not covered by the existing test suite is because of lacking failure cases produced, we see more motivation for mutating messages sent between microservices to force failure cases. An approach to fault injection that would work particularly well for the login service example would be mutating the responses that it receives from its dependencies. This approach would take the information from a response of a dependent service, and alter the information so the data being received by the client service is invalid [21]. By doing so, it would allow the verification code to be altered, and the verification failure condition to be entered. A similar approach can be taken by manually altering the logic of the source code in the dependent service to respond to the client service with a payload that triggers the execution of the error-handling branches. However, doing so is no longer considered as end-to-end testing since TrainTicket's backend is no longer being tested as a black box. We recommend a combination of mutating the messages between services and test cases to achieve a higher level of coverage.

The second insight is that a major portion of the lines not being covered are caused by the lack of quality within the TrainTicket test suite. The two examples that encompass the majority of the lines not covered are the following. First, the TrainTicket test suite does not do is not completely testing every functionality that is available to the user. For instance, the logout functionality is never interacted with throughout the test suite. Second, by analyzing certain classes not covered at all, we find that many of these classes would be considered as dead code. The methods in these classes are never invoked throughout the running of the test suite. Removing these classes have no effect on the functionalities of the login service.

We make the final insight during an approach to cause execution of error-handling branches by taking down a service that the client is making a request to. When attempting to get an invalid request to cause execution of the logout error-handling branch, we decide that removing the logout service should cause a timeout error, and enter the logout error-handling branch. However, the login branch actually crashes because of a run-time error since the conditional requires a variable to have an attribute to check for. But the variable that is supposed to be part of the response from the logout service is NULL because the response from the logout service is never received by the login service. There are three cases of non-functional faults that occur in just the login service because of lacking logic to allow the service to fail safely when no response is received from a service that it is making a request to. A solution to fix these faults is to create try-catch blocks to support the case of no response being returned, and add an assertion that the variable is not NULL before referencing it.

CHAPTER 6: CONCLUSION

In this thesis, we have presented an empirical study of measuring the code coverage achieved by a microservice system's test suite and augmenting the test suite to achieve higher code coverage. We use the Jacoco code-coverage tool to collect code-coverage information on an open-source benchmark microservice system, TrainTicket. The two types of code coverage focused in our study are service coverage and line coverage. The collection of the code-coverage information consists of three steps. First, we collect the code-coverage information after the deployment of the TrainTicket system. Second, we collect code-coverage information after the existing test suite has been run. Third, we collect code-coverage information after the existing test suite has been augmented to increase the code coverage.

We have made the following observations from analyzing the code coverage after deployment and after the existing test suite is run, respectively. When we collect the code-coverage information after the deployment, an interesting finding is that there are multiple covered lines of code related to the initialization process of Spring-based microservices. After the existing test suite is run, we hypothesize that there exists correlation of lower code coverage and a higher number of service-request dependencies within a microservice but such hypothesis is not confirmed by our results (Figures 5.2 and 5.3). However, we do not believe that the results are strong enough to refute our hypothesis either.

We have gained three main insights while augmenting the TrainTicket's existing test suite to achieve higher code coverage in the three chosen microservices. The first insight is that there is a high percentage of un-covered lines being blocked by error-handling branches, and it is very difficult to execute these branches through only end-to-end test cases. The second insight is that a major portion of the covered lines stem from dead code, and user functionalities that are not exercised during the execution of the existing test suite. The third insight is related to aiming to execute those error-handling branches: a non-functional fault occurs when we take down a service's dependency and making a request to the dependency.

In future work, we plan to investigate the effectiveness of automated test-generation tools such as Monkey [22] for simulating a user's actions via the front end of a microservice system. There exist coverage-measurement tools for an Android app as the front end of a microservice system. We plan to investigate whether there is a correlation between the front-end coverage and backend coverage achieved by an automated test-generation tool for Android.

REFERENCES

[1] M. Fowler and J. Lewis, “Microservices a definition of this new architectural term,” 2014. [Online]. Available: http://martinfowler.com/articles/microservices.html

[2] P. Jamshidi, C. Pahl, N. C. Mendona, J. Lewis, and S. Tilkov, “Microservices: The journey so far and challenges ahead,” *IEEE Software*, vol. 35, May 2014.

[3] “Trainticket: A benchmark microservice system,” 2019. [Online]. Available: https://github.com/FudanSELab/train-ticket

[4] M. W. Maier, D. E. Emery, and R. Hilliard, “Software architecture: Introducing IEEE standard 1471,” *IEEE Computer*, vol. 34, no. 4, pp. 107–109, 2001. [Online]. Available: https://doi.org/10.1109/2.917550

[5] N. Dragoni, S. Giallorenzo, A. Lluch-Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina, “Microservices: Yesterday, today, and tomorrow,” pp. 195–216, 2017. [Online]. Available: https://doi.org/10.1007/978-3-319-67425-4_12

[6] X. Zhou, X. Peng, T. Xie, J. Sun, C. Ji, W. Li, , and D. Ding, “Fault analysis and debugging of microservice systems: Industrial survey,” *IEEE Transaction On Software Engineering*, vol. 14, Aug. 2018.

[7] “Microservices: An introduction to monolithic vs microservices architecture (msa),” Oct. 2018. [Online]. Available: https://www.bmc.com/blogs/microservices-architecture/

[8] “What is the elk stack?” 2018. [Online]. Available: www.elastic.co/elk-stack

[9] “Zipkin web,” 2019. [Online]. Available: https://zipkin.io/

[10] “Unit testing,” 2019. [Online]. Available: www.softwaretestingfundamentals.com/unit-testing/

[11] “Selenium,” 2019. [Online]. Available: https://www.seleniumhq.org/

[12] “Faults,” 2019. [Online]. Available: www.tutorialspoint.com/software_testing_dictionary/fault.htm

[13] A. Authors, “Latent error prediction and fault localization for microservice applications by learning from system trace logs,” unpublished.

[14] X. Zhou, X. Peng, T. Xie, J. Sun, C. Ji, and W. Zhao, “Benchmarking microservice systems for software engineering research,” *International Conference on Software Engineering*, vol. 40, May 2018.

[15] X. Zhou, X. Peng, T. Xie, W. L. Jun Sun, C. Ji, and D. Ding, “Delta debugging microservice systems,” *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018.

[16] “What is a container?” 2019. [Online]. Available: https://www.docker.com/resources/what-container

[17] “Kubernetes website,” 2019. [Online]. Available: https://kubernetes.io/

[18] “Docker compose cluster management tool,” 2019. [Online]. Available: https://docs.docker.com/compose/

[19] “Sonarqube website,” Oct. 2018. [Online]. Available: https://www.sonarqube.org/

[20] “The ioc container,” 2019. [Online]. Available: https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html

[21] C. Miller and Z. N. Peterson, “Analysis of mutation and generation-based fuzzing,” *Independent Security Evaluators, Tech. Rep*, 2007.

[22] “User interface and application exerciser monkey,” 2019. [Online]. Available: https://developer.android.com/studio/test/monkey