Faculteit Industriële Ingenieurswetenschappen

master in de industriële wetenschappen: elektronica-ICT

Masterthesis

Microservice coverage detection

Maarten Baeten

Scriptie ingediend tot het behalen van de graad van master in de industriële wetenschappen: elektronica-ICT

PROMOTOR :
Prof. dr. Kris AERTS

PROMOTOR :
Prof. Davide TAIBI

BEGELEIDER :
dr. Dario AMAROSO D ARAGONA

2022
2023

Faculteit Industriële Ingenieurswetenschappen

master in de industriële wetenschappen: elektronica-ICT

Masterthesis

Microservice coverage detection

Maarten Baeten

Scriptie ingediend tot het behalen van de graad van master in de industriële wetenschappen: elektronica-ICT

PROMOTOR :

Prof. dr. Kris AERTS

PROMOTOR :

Prof. Davide TAIBI

BEGELEIDER :

dr. Dario AMAROSO D ARAGONA

Preface

This thesis is made as the completion of the Master Engineering Technology Electronics and ICT. Yours truly has a bachelor's degree in engineering technology with a specialization in electronics-ICT from the UHasselt and the KU Leuven and this thesis is the final product of the master period. This research was held during the period of the 6th of February until the 22nd of May.

I personally chose this topic to further enhance my understanding of software engineering. The topics discussed in this thesis were slightly touched by my bachelor's study, however, the actual implementation and details fall outside the scope of the standard program. This is actually the reason why I chose this topic for my master's thesis. It should be clear that this thesis falls outside my comfort zone, however, I can proudly inform you that this thesis has been very meaningful and interesting. That is why I would like to thank my professor Kris Aerts for his valuable input and support during the entire master's period. Furthermore, I would also like to thank professor Davide Taibi from the CloudSEA.AI group at the university of Tampere for the opportunity to perform my master's thesis in cooperation with his research group. PhD. student Dario Amoroso D Aragona of CloudSEA.AI should also be mentioned for his support during the period. Last but not least, I would like to thank KU Leuven for making this exchange study possible.

Finally, I would like to thank my girlfriend, family, and friends for being helpful and supportive during my exchange program in Tampere, Hervanta.

I hope you find this thesis interesting and educational.

Maarten Baeten

Tampere, 22nd May 2023

Contents

List of Tables

List of Figures

Concepts

CloudSEA.AI	Cloud, Software Engineering, Evolution, and Assessment with AI research department at Tampere university.
MSA	Microservice architecture (MSA) is a logical structure for the design of a software program involving loosely-coupled modular components known as microservices.
Faas	Function-as-a-Service (FaaS) is a kind of cloud computing service that allows developers to build, compute, run, and manage application packages as functions without having to maintain their own infrastructure.
API	An Application Programming Interface (API) is a way for two or more computer programs to communicate with each other.
RAM	Random-Access Memory (RAM) is used as short-term memory storage for a computer's central processing unit (CPU).
REST	Representational State Transfer (REST) is a type of software architecture that was designed to ensure interoperability between different internet computer systems.
SDK	A Software Development Kit (SDK) is a set of software tools and programs provided by hardware and software vendors that developers can use to build applications for specific platforms.
DFS	Depth-first search (DFS) is an algorithm for traversing or searching tree or graph data structures.
Call-graph	A call graph (also known as a call multigraph) is a control-flow graph, which represents calling relationships between subroutines in a computer program.

Abstract in het Nederlands

Microservices zijn kleine herbruikbare en eenvoudig schaalbare componenten die zich op verschillende fysieke en virtuele plaatsen kunnen bevinden. Deze spreiding bemoeilijkt echter het verifiëren of alle services getest zijn. Deze thesis onderzoekt welke methodiek ervoor kan zorgen dat alle services geverifieerd worden. De eerste stap omvat het definiëren van de reikwijdte van de applicatie op basis van beschikbare documentatie. Dit moet resulteren in een overzicht van de microservices architectuur. Call-grahps helpen bij de visuele voorstelling hiervan. Het idee voor de servicedekking is gebaseerd op het principe van codedekking. Codedekking verifieert of alle code in een (alleenstaande) softwaretoepassing wordt uitgevoerd. Dit is niet te verwarren met testdekking die controleert of een programma voldoet aan alle functionele vereisten. Dit laatste is niet de scope van deze thesis. De voorgestelde methodiek past het Depth-First-Search (DFS) algoritme toe op de call-graphs met gekende microservices structuur om in eerste instantie (offline) het aantal nodige testen voor een volledige dekking te bepalen. De effectieve testfase zal opnieuw DFS toepassen en bijhouden hoeveel unieke test succesvol werden uitgevoerd. De verhouding is een maat voor de servicedekking. De voorgestelde methodiek is generiek toepasbaar en biedt een maatstaf ter controle van de servicedekking. De eerste stappen voor het empirisch testen werden ook genomen a.h.v. een trein-ticket microservice applicatie.

Abstract in English

Microservices are small reusable and easily scalable components that can reside in multiple physical and virtual locations. However, this distribution of the services makes verifying whether all the services are tested difficult. This thesis investigates which methodology ensures that all services are verified. The first step involves defining the scope of the application based on available documentation. This should result in an overview of the microservices architecture. Call-graphs help with the visual representation of this. The service coverage idea is based on the principle of code coverage. Code coverage verifies that all code in a (stand-alone) software application is executed. This is not to be confused with test coverage that checks whether a program meets all functional requirements. The latter is not the scope of this thesis. The proposed methodology applies the Depth-First-Search (DFS) algorithm to the call-graphs with the known microservices structure to initially determine (offline) the number of tests required for full coverage. The effective test phase will again apply DFS and track how many unique tests were successfully performed. The ratio is a measure of service coverage. The proposed methodology is generically applicable and provides a benchmark for checking service coverage. The first steps for empirical testing were also taken by means of a train-ticket microservice application.

Chapter 1

Introduction

1.1 Situation

The CloudSEA.AI research group is a research institute located at the University of Tampere. Their aim is to gain a better understanding of the advantages and disadvantages of cloud-native technologies. This enables them to support companies to implement new technologies and furthermore explain these technologies to the businesses [16]. The CloudSEA.AI group performs research on different topics such as microservice architecture (MSA), serverless functions (Function as a Service or FaaS), edge computing, software maintenance and more [16].

1.2 Problem statement

In recent years, a reduction in monolithic systems has been observed [17–19]. This is because monolithic systems usually bring a few disadvantages with them which modern businesses are struggling with [17]. Monolithic systems are less reusable, less scalable and require more deployment and restart times [4], [20]. These are only a few reasons why companies are re-architecting their monolithic system to a microservice-based system. Several large companies such as Spotify, Amazon and Netflix use a microservice-based system in order to easily deploy self-isolated, independent components that are easily scalable [4], [21]. A microservice should have a single clearly defined purpose and be loosely coupled to other services, thus independent. This comes with a few challenges. Most of the old systems are monolithic systems and require a transformation to become microservice-based. This process however, is not that simple and requires a good understanding of the actual implementation of the application [21], [22]. The decomposition of a monolithic system to a microservices-based system is usually performed manually [23], [24]. [24] offers a measurement framework to compare different decomposition methods. It is also important to test the decomposed system. This is where test coverage detection becomes important. In a microservice-based system, all the services are situated in different physical and virtual locations. Each of these services is executing calls to other services and waiting for responses. This makes it difficult to completely verify if all the application programming interfaces (APIs) are tested but also if all the different micro-services are tested in respect with each other and if all the paths between microservices are tested. Coverage tests must keep track of which functions and source code were executed during the testing phase on different levels such as unit tests, functional tests and integration tests.

1.3 Goals

The main goal of this research is to see whether there are any tools to verify if all the code is being tested during the test phase. It is important in a microservice-based system that this verification method runs over all the different APIs, all the different microservices and all the different paths within each microservice.

This research aims to answer the following questions. How can a high service coverage be established? What methods can be used to establish a clear overview of a MSA? How is it possible to verify which interactions happen between different microservices? This research aims to establish a methodology that helps answer these questions.

1.4 Methods

The first step is to perform a state of the art in related topics such as microservice testing and coverage methods. It is important to determine if there already exist some tools that perform this microservice coverage detection. This should be taken into account when developing a new tool or methodology. Then the actual case study can be deployed. To deploy the train ticket application on either a local machine or a server, an understanding of Kubernetes and KubeCTL is required. There are many different online tutorials and courses that explain the use of Kubernetes and why it is used in developing microservice-based systems. After the deployment, the tool can be developed. Since the train-ticket application is programmed in Java, the tool should also be programmed in Java including a visualization. It is therefore important to verify if there already exists such a tool. If there exists such a tool, that tool can be integrated into the coverage detection tool, if not, a visualization method should be developed. After the tool is developed, but also during the process of development, everything should be documented correctly as well in the form of a paper and poster but also in the source code of the tool.

1.5 Preview

Before determining a methodology it is important to understand the overall context. To do this it is necessary to have a clear understanding about microservice communication protocols and testing. These topics are explained in chapter 2. The general steps for implementing microservices coverage detection are explained in chapters 3. Chapter 4 gives more details on the proposed MSA coverage metric. Finally a conclusion is made in chapter 5.

Chapter 2

Literature study

2.1 Introduction

This chapter comprises the key-characteristics of a monolithic system and a microservice architecture (MSA). The key-differences, advantages and disadvantages are also listed in the next sections. Then the testing of microservices is explained on the different levels in section 2.4. It is important to have a good understanding of the concepts where a MSA differs from a monolithic architecture because this alters the way that the system is tested, at least on the integration level. Figure 2.1 shows the general difference between a monolithic architecture and a microservice architecture. The methodology proposed by this thesis focuses on the integration level. There are a lot of code coverage tools for the unit level, but no code coverage tools for the integration level exists.

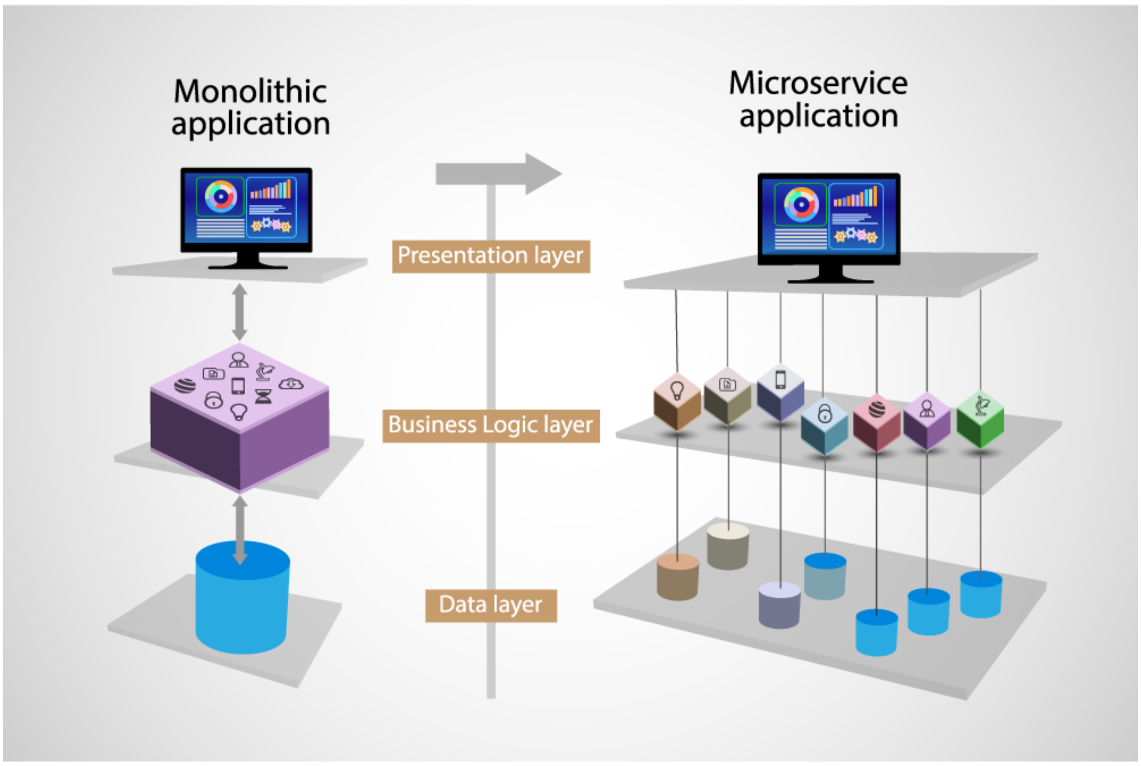


Figure 2.1: Monolithic and microservice architecture [2, p. 1]

2.2 Monolithic Software architecture

The traditional model for the design of a software system is a monolithic architecture. Monolithic, in this context, means 'composed all in one piece'. A monolithic system is usually self-contained. This means that there are no external dependencies [25]. The structure of a monolithic system is usually fixed thus each component should exist for code to be carried out by a compiler. Furthermore monolithic applications are single-tiered, which means that all the different components are gathered into one large component (see also figure 2.3 further in) [26]. Figure 2.2 shows a typical monolithic architecture.

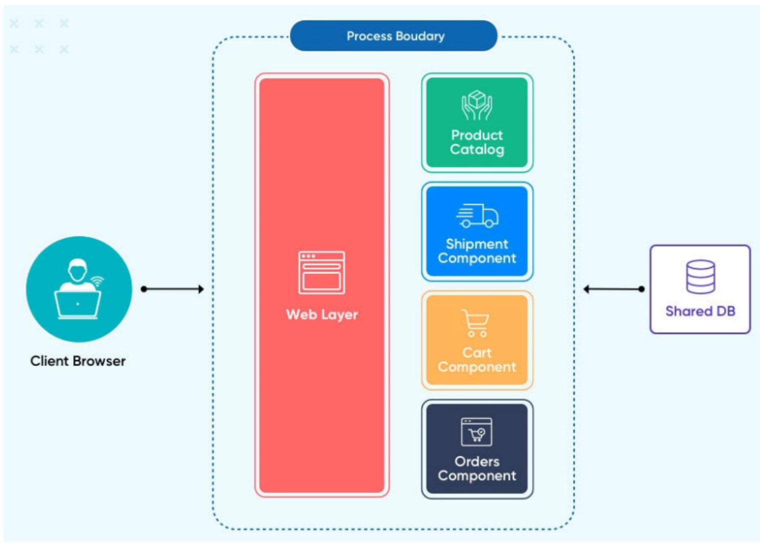


Figure 2.2: Monolithic architecture example [3, p. 1]

2.2.1 Monolithic architecture advantages and disadvantages

The list below gives an overview of the most important disadvantages of monolithic systems.

- *High code coupling*: Practice shows, that eventually most monoliths end up as a spaghetti code in some places of the monolith. This results in a high code coupling which makes the system harder to oversee and understand especially for large monoliths [27]. If a component undergoes a change, other components might require an adjustment as well. This process can be time-consuming and difficult for large-scale monolithic systems since a change in a component requires the entire system to be recompiled and tested.

- *Performance issues*: Monolithic applications are usually bound to a single database for all its services as shown in figure 2.3 [27]. This decreases the performance and flexibility of the monolith, because scaling becomes more difficult [27].

- *The cost of infrastructure*: Only the whole system can be scaled, this brings additional cost for application operability [26].

- *Lack of flexibility*: Because there is such a tight coupling between the service, it is usually the case that services inside the monolith require a specific version of a certain tool or plugin to operate correctly [28], [29]. This might not be efficient to maintain the best version of a tool, since an update of the tool requires the entire system to be adjusted, or at least a large portion of it, in order to use the newer version of the plugin [28], [29].

- *Slow speed of development*: Monolithic systems tend to have a large codebase which can lead to issues when managing all the code and adding new features over time [28]. The continuous integration and continuous deployment pipeline brings forward one of the simplest disadvantages of a monolith. In a monolith all the services with their corresponding tests are executed for each pull request [26]. If a change would happen in the source code, the entire pipeline has to be executed again [30]. This takes a lot of time, and if the pipeline for some reason fails, then it needs to be restarted all over again.

However, these disadvantages are not stopping development teams from using a monolithic system. The advantages of using a monolithic system are listed below:

- *Simplicity of development*: A monolithic software architecture is a straightforward implementation. All the source code is located in a single place [27], [26].

- *Simplicity of debugging*: Debugging is more manageable because all the source code is located in a single place. If a request has been made, it can easily been tracked down and followed throughout the different services within the monolith to find the issue [26]. However it is sometimes more difficult to isolate a bug since the code in a monolith is intertwined with itself.

- *Simplicity of testing*: There is only one thing that is tested and that is the monolith. There are no other dependencies that are not directly coupled to the service [26].

- *Low cost on early stages of the application*: All the source code is located in a single place, and a single module is deployed [26]. There is little to no overhead in early stages, not in infrastructure nor in development expenses [26].

2.3 Microservices software architecture

A MSA system consists of small, isolated services as shown in figure 2.3. Each service is responsible for its own specific task. Figure 2.4 shows an example of a MSA system with three microservices (User Service, Flight Service, Billing Service). However an actual MSA application consists of a lot more microservices. Each service can operate on its own database or multiple services can share a database as shown in figure 2.5. The type of database and the characteristics can be different for each database. This enables flexible use of databases since different services might require a different type of database to operate as efficient as possible. Microservices are loosely coupled from each other. This makes it possible to deploy and manage each of the services completely separately. Nevertheless microservices should be able to communicate with each other. If a user bought a ticket with the billing service, the billing service should inform the flight service that the corresponding user has bought a ticket for that flight, e.g. to verify that there are still seats available for that flight. Requests are usually handled by a gateway. This gateway is responsible for routing the request to the correct services. The gateway might also implement security measures to verify that the user that sends the request is authorized to perform the action. There are a number of methods to establish this type of communication which are explained in section 2.3.2.

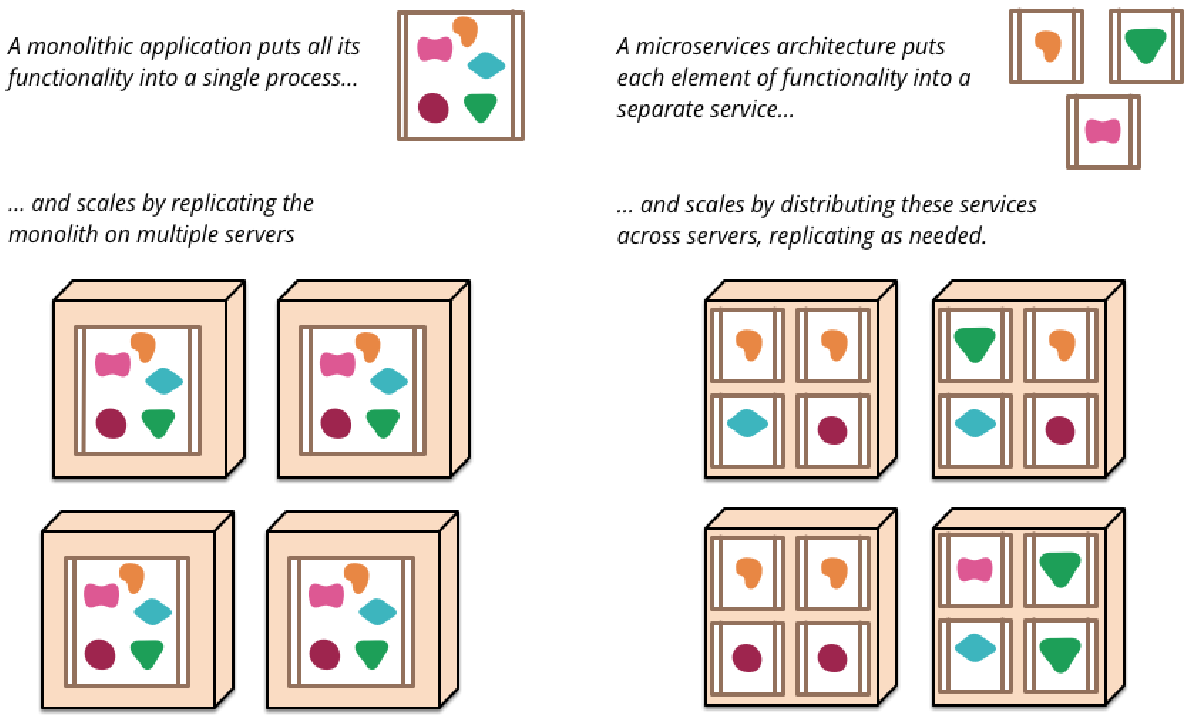

A monolithic application puts all its functionality into a single process...
A microservices architecture puts each element of functionality into a separate service...
... and scales by replicating the monolith on multiple servers
... and scales by distributing these services across servers, replicating as needed.

Figure 2.3: Monoliths and microservices [4, p. 1]

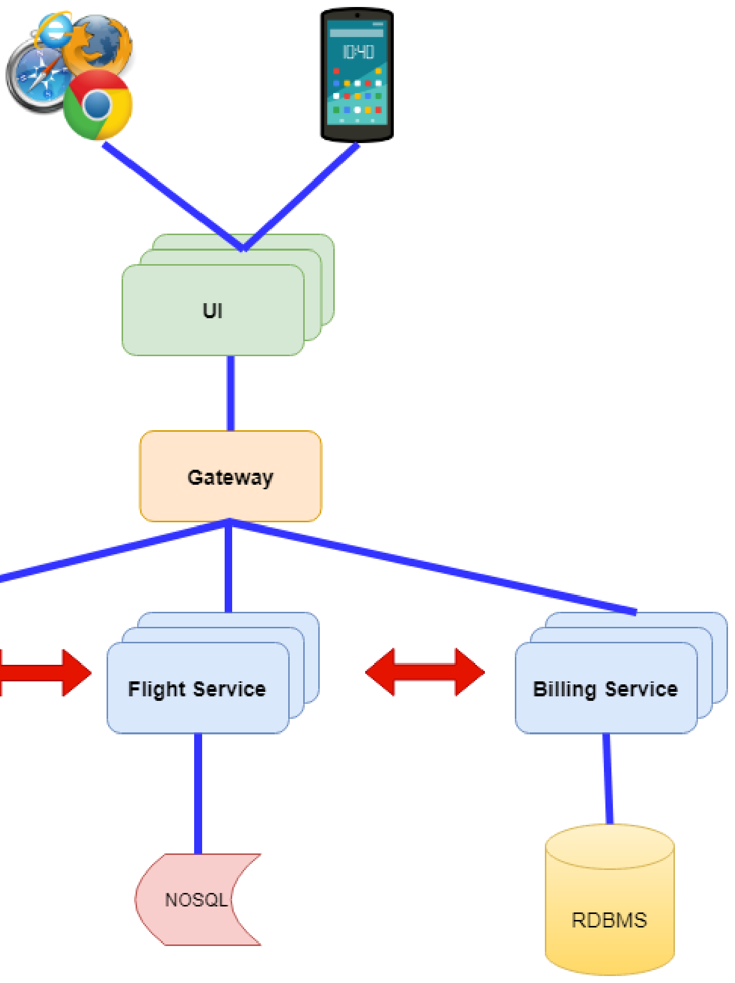


Figure 2.4: Microservice architecture example [5, p. 1]

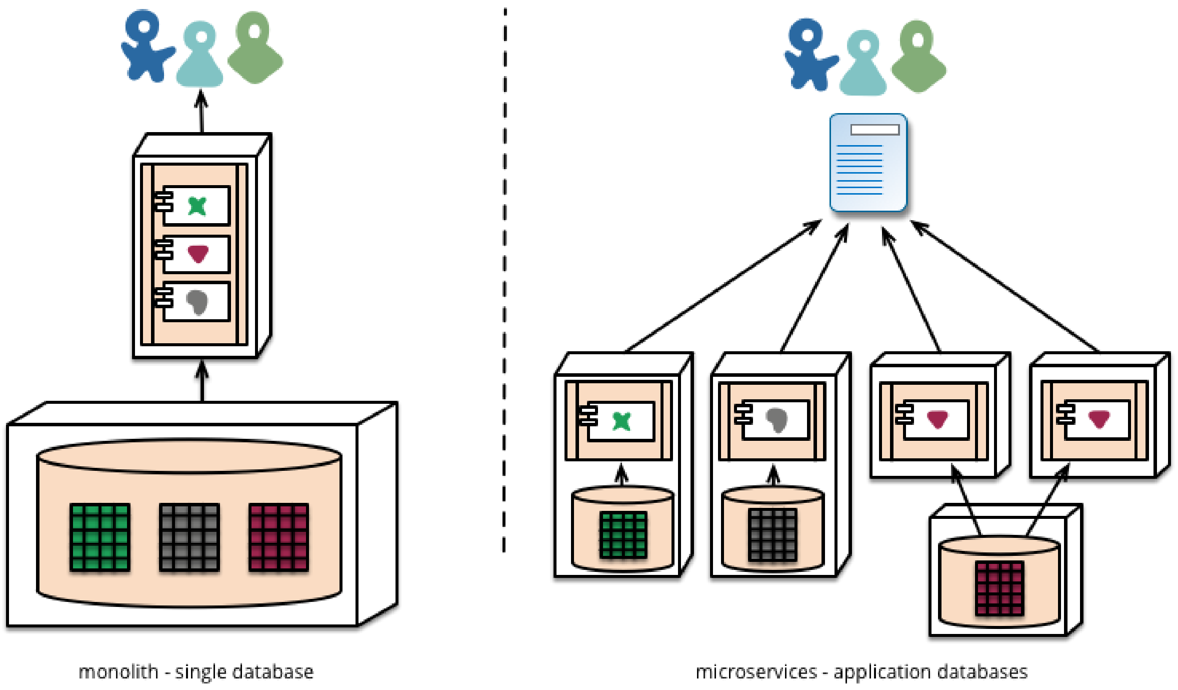

monolith - single database
microservices - application databases

Figure 2.5: Database system of monoliths and microservices [4, p. 1]

2.3.1 MSA advantages and disadvantages

The list below gives an overview of the most important disadvantages of a MSA:

- *Debugging*: MSA systems are more complex since all the services can be located on different physical and virtual locations. A single request can require the gateway to call multiple services. That makes it hard to visualize the entire system and track the request over all the used services [26]. Moreover, it might be difficult to make the distinction between a bug in the code or a bug in the communication.

- *Testing*: Testing a MSA application on the integration level is difficult due to the distributed components [20], [31].

- *Dependence*: The services usually communicate with each other in a strictly defined matter. If a service gets changed, it should not impact other services [31]. This should apply for internal services or 3rd party services. Usually when a microservice is created, there is also a documentation on how the microservices work and what input and output the microservice, requires or gives. However, when a microservice is updated, it might not always be backward compatible, and the update of the documentation might lag behind. It is therefore important to take versioning into account when creating or using microserivces.

- *Cross-cutting concerns*: Each service should contain logic for security, logging configuration, etc [5].

The most important benefits of a MSA applications are:

- *High cohesion, low coupling*: The low coupling of microservices brings forth a few advantages. Each service can be developed by a different development team because each service is independent.

- *Limited service scope*: As explained earlier, a service is usually small. This results in a service that is easier to understand. Debugging and testing are also easier within the scope of a single service [5]. This also makes the deployment time of a single service very little because the size of a service is relatively small.

- *Flexibility*: Each service can use its own database as shown in figure 2.5 with its own requirements. This can be a NoSQL, RDBMS, both or none. This loosely coupling enables the services to be developed in different programming languages and use different technologies for each service.

- *Scalability*: A MSA application allows for the independent scaling and optimization of each component. According to the load, extra containers can be deployed. The system can be scaled according to the best performance. This makes large scale MSA applications more cost effective than a monolithic system. Because when scaling a monolithics system by definition all parts must be scaled identically, whereas the services of a MSA can be scaled differently.

- *High speed of development*: There are a lot of frameworks available to be used as a MSA application (e.g. Spring boot, Eclipse Vert.X, Oracle Helidon, ...). The continuous integration and continuous deployment pipeline is also faster because services are small.

2.3.2 MSA communication protocols

The most popular communication methods are the HTTP protocol or the use of a message queuing protocol. Choosing which protocol to use should depend on the scenario and goals. The different methods can be categorized in groups according to their key characteristic:

- *Synchronous*: HTTP or HyperText Transfer Protocol is a synchronous protocol [32]. The client sends a request and waits for the service to give a response [32]. The request gives the server the necessary data it needs to customize its reply for the client device. Figure 2.6 gives an schematic overview of the http protocol. The important factor here is that the client can only continue when it receives the response.

- *Asynchronous*: AMQP (Advanced Message Queuing Protocol) or MQTT (Message Queuing Telemetry Transport) are both asynchronous protocols [32]. Here a broker is used to handle the different subscriptions and topics. If a publisher publishes to a specific topic, the broker will inform the consumers of that corresponding topic and send the message to the consumer [32]. MQTT has client/broker architecture whereas AMQP has a client or broker and client or server architecture. In contrast to AMQP, which uses response or request and publish or subscribe techniques, MQTT adheres to the abstraction of publishing and subscriptions [32]. Figure 2.7 shows the schematic overview of an AMQP. Figure 2.8 shows the schematic overview of a MQTT.

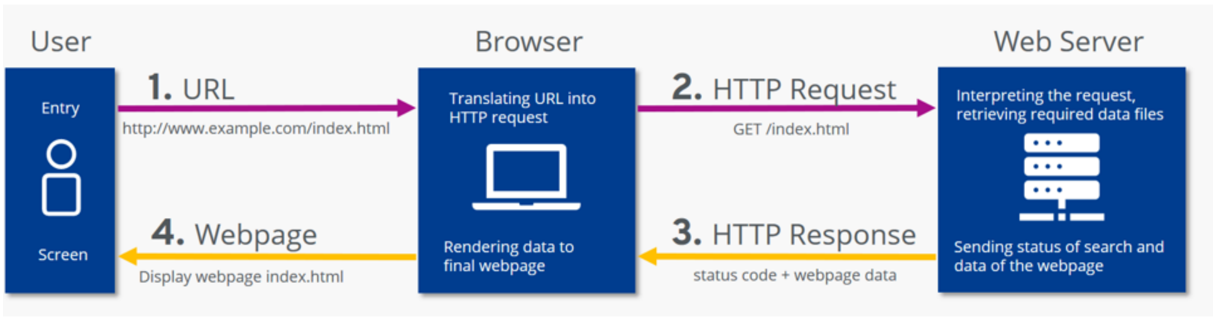

User
Browser
Web Server
Entry
Screen
1. URL
http://www.example.com/index.html
Translating URL into HTTP request
2. HTTP Request
GET /index.html
Interpreting the request, retrieving required data files
4. Webpage
Display webpage index.html
Rendering data to final webpage
3. HTTP Response
status code + webpage data
Sending status of search and data of the webpage

Figure 2.6: Diagram of the HTTP protocol communication process [6, p. 1]

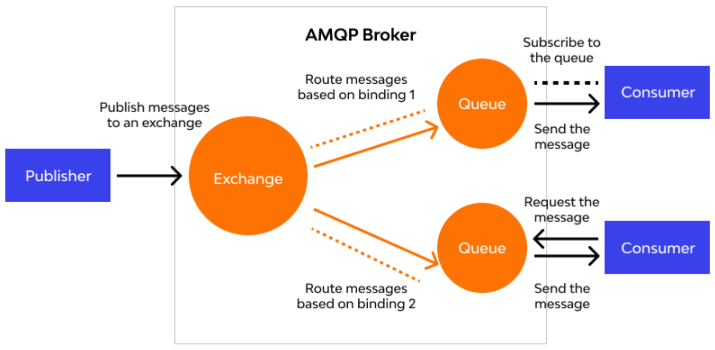


Figure 2.7: Diagram of the AMQP communication process [7, p. 1]

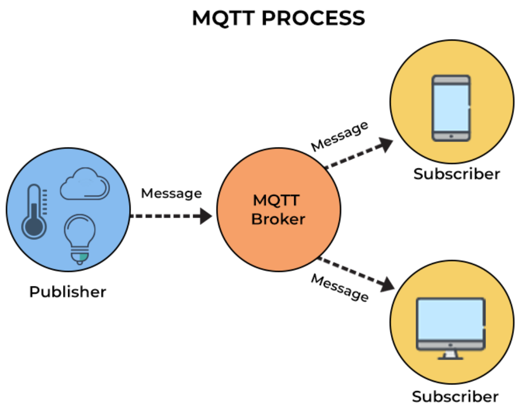


Figure 2.8: Diagram of the MQTT communication process [8, p. 1]

Another important characteristic in which communication protocols can be divided is whether the protocol allows only single or multiple receivers:

- *Single*: The request is processed by exactly one receiver or transmitter e.g. the Command Pattern [32]. In a command pattern, an object is used to store all the data necessary to carry out an action or to start an event later on.
- *Multiple*: The request is processed by multiple or no receivers. The publish/subscribe technique used in designs like the event-driven architecture is one example. When propagating data updates between various MSA through events, it is based on an event-bus interface or message broker. This is typically done through a service bus by employing topics and subscribers [32].

2.4 Testing of MSA applications

The next question to ask is how to test these microservices in order to confirm that the services work as desired. The testing of these services can be done on different levels as shown in figure 2.9. Figure 2.9 also shows two extra types of tests: component tests and contracts tests. These two test levels are usually not found in the testing pyramid for monoliths [9], [32]. This section discusses how the testing of microservices is achieved and briefly discusses each different type of test and their subcategories. Compared to monoliths, microservices are much more likely to require network calls to perform their task. This can be handled in two ways. The first option would be to accept the latency and let the service do the request. The second option would be to create some sort of test double that imitates the result. These two options divide the different types of unit tests into two main groups: Solitary unit tests and sociable unit tests. A schematical representation of the two is shown in figure 2.10. There should be a good balance between the use of solitary and sociable unit tests. Mocking things makes testing faster and reduces uncertainty. However, if too much is getting mocked, the results can be less trustworthy since sociable tests despite their downsides are more realistic.

- *Solitary*: These tests are used when the results should be deterministic. Mocks or stubs are used to isolate the service under test from external dependencies.
- *Sociable*: These tests allow the service that is being tested to call other services. Sociable tests aren't deterministic, but they give confidence in the results when used since they simulate a real scenario.

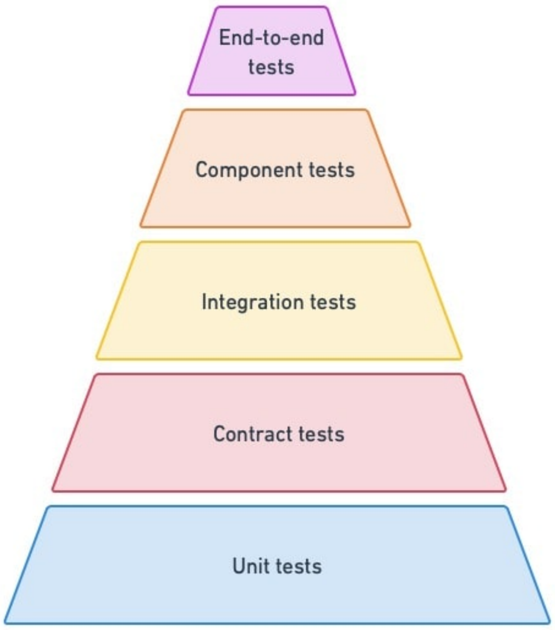


Figure 2.9: Test pyramid for microservices [9, p. 1]

- *Unit tests*: Unit tests are located at the bottom of the pyramid, this means that they are performed at the lowest level of integration. As a consequence they are the most represented tests in a system. Unit tests are also the most granular and the most numerous form of testing [9], [32]. A unit consists of a class, method, or function that is tested in isolation.

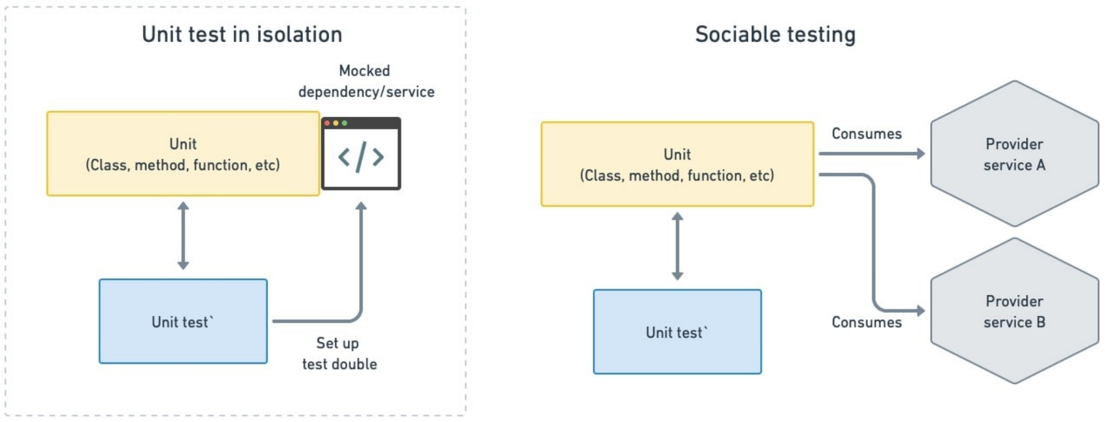


Figure 2.10: Solitary and sociable units tests [9, p. 1]

- *Contract tests*: A contract is formed whenever two services are connected through an interface. It declares all possible inputs and outputs along with data structures and constraints when the service can be used [9], [32]. Consumers and producers of services should obey the rules as stated in the contract to enable communication. Contract tests ensure that microservices comply with their arrangement. These tests verify that the inputs and outputs have the expected characteristics [9], [32]. Contract tests prove that the service has acceptable latency and performance. Contract tests can be run by the producer, consumer or both as shown in figure 2.11.
  - *Consumer-side contract tests*: During the testing phase, the microservices are connected to a fake or mocked version of the producer service. These mocks can be created by tools such as Wiremock [9], [32]. This verifies if the service can use the API as stated in the contract, or usually API docs. The consumer-side contract tests are executed by the downstream team.
  - *Producer-side contract tests*: This type of test emulates the various API requests a client can perform and ensures that the producer complies with the contract [9], [32]. Vendor testing lets developers know when they are trying to break compatibility with consumers. The producer-side contract tests are executed by the upstream team.
- *Integration tests*: Integration tests in microservices differ from the integration tests in monolithic systems. In a MSA application, the integration tests are used to identify defects when certain microservices communicate with each other. These integration tests are not to be confused with contract tests where at least one side is always a mocked version. Integration tests use real services to verify the calls between the services and their cooperation. Integration tests are used to determine if services can communicate with each other in a correct manner and use their own databases [4], [9], [32]. These tests are used to detect faults such as missing HTTP headers, mismatched request pairings, and mismatched response pairings. As such integration tests are implemented at the interface level [4], [9]. Fig 2.12 gives a schematic representation of integration tests in a system with three microservices and a database.

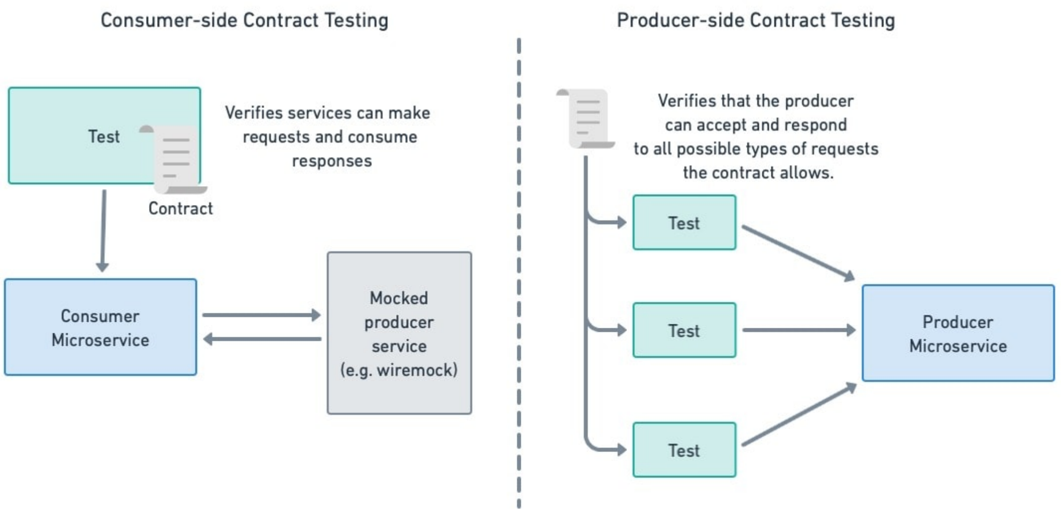

Consumer-side Contract Testing
Test
Contract
Verifies services can make requests and consume responses
Consumer Microservice
Mocked producer service (e.g. wiremock)
Producer-side Contract Testing
Verifies that the producer can accept and respond to all possible types of requests the contract allows.
Test
Test
Test
Producer Microservice

Figure 2.11: Consumer and producer contract tests [9, p. 1]

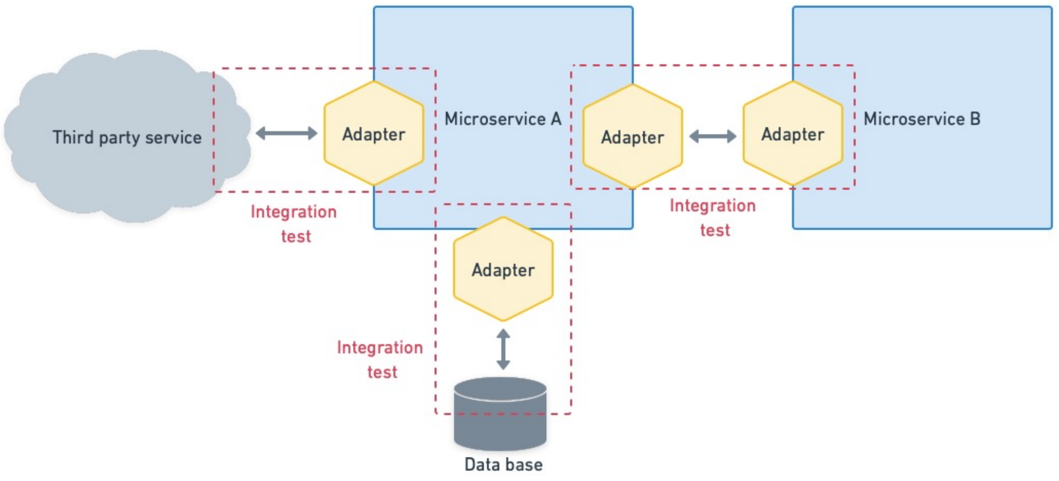


Figure 2.12: Integration tests [9, p. 1]

- *Component tests*: Component testing is a type of testing that examines the behavior of components in isolation by replacing or mocking services with simulated resources [9]. Component tests are more thorough than integration tests because they follow all different scenario paths, for example, the reaction of a component when simulated network failures and faulty requests occur [4], [9]. These tests are used to verify if the component meets the consumer's needs. Therefore it has some similarities with end-to-end testing. Component testing differs in the fact that here a component is isolated, cut-off from the rest of the system. Figure 2.16 gives a schematic overview of the difference between component and end-to-end testing. Figure 2.13 gives a schematic representation of component testing. There are two different ways for component testing which are: in-process and out-of-process. The most popular tools to write in-process component tests are Cucumber and Capybara [9].
  - *In-process*: Here the test runner is located in the same thread as the microservice. The microservice is deployed in an offline test mode where all its dependencies are mocked [9]. This allows the microservice to be executed without the network. This is only possible when the component consists of a single microservice.

- *Out-of-process*: In out-of-process testing any number of services can be combined to create a component, which can then be tested. As stated earlier, all the other dependencies outside the component are mocked. The component itself and everything inside the component remain unaltered during testing. Figure 2.15 gives a schematic representation of out-of-process component testing.

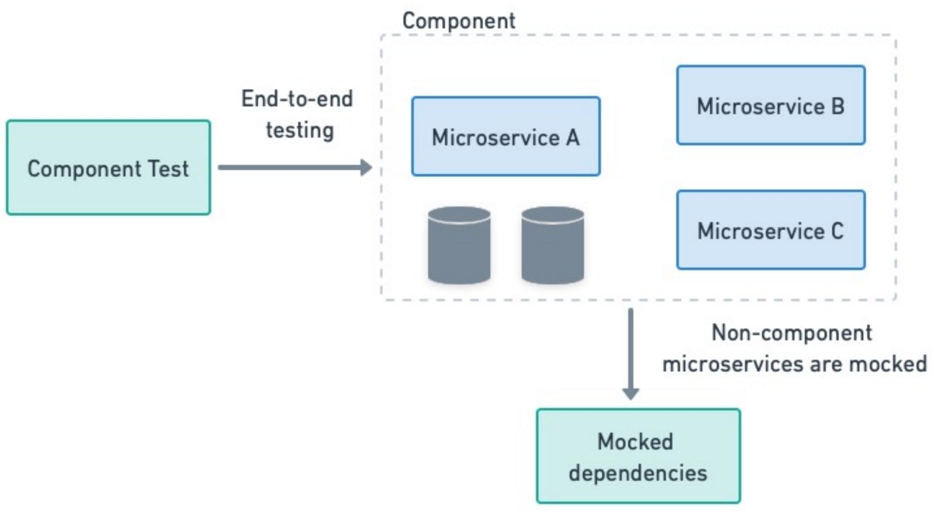


Figure 2.13: Components tests [9, p. 1]

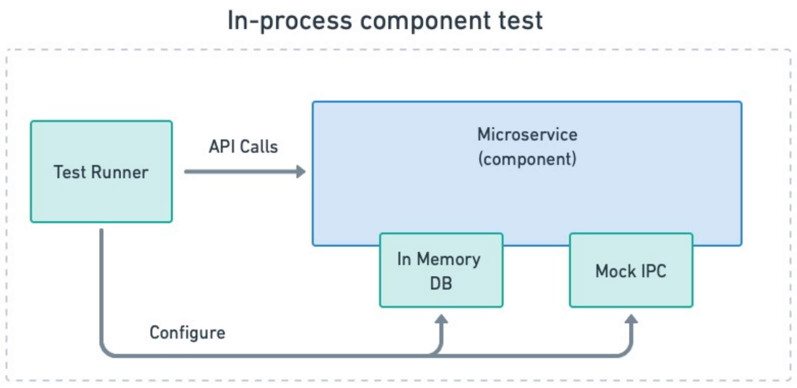


Figure 2.14: In-process component testing [9, p. 1]

- *End-to-end tests*: End-to-End (E2E) testing verifies that the system meets the user requirements and achieves their business objectives. This type of testing covers the entire system and is usually performed in an interface that is similar to the one the user would interact in. E2E tests usually consist of API and UI tests. For this testing, the system should run in an environment that is almost identical to the production implementation. Third-party services should also be included in this environment, however, this brings extra costs with it and therefore some third-party services might be mocked. Figure 2.17 gives a schematic overview of E2E testing.

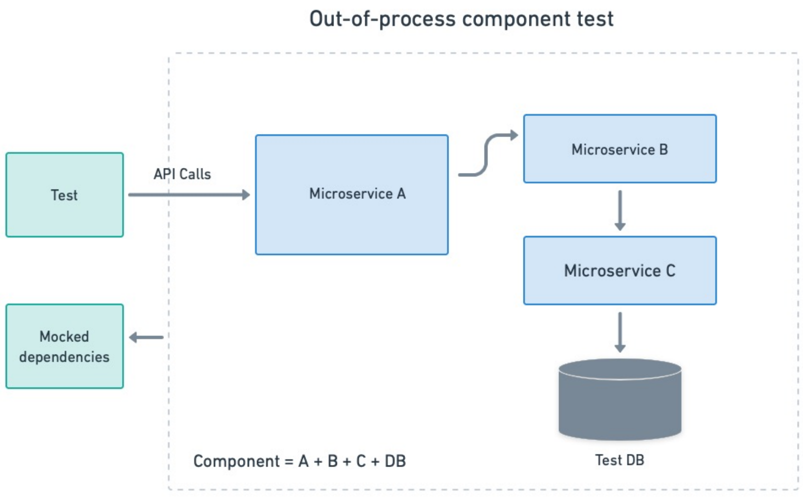


Figure 2.15: Out-of-process component testing [9, p. 1]

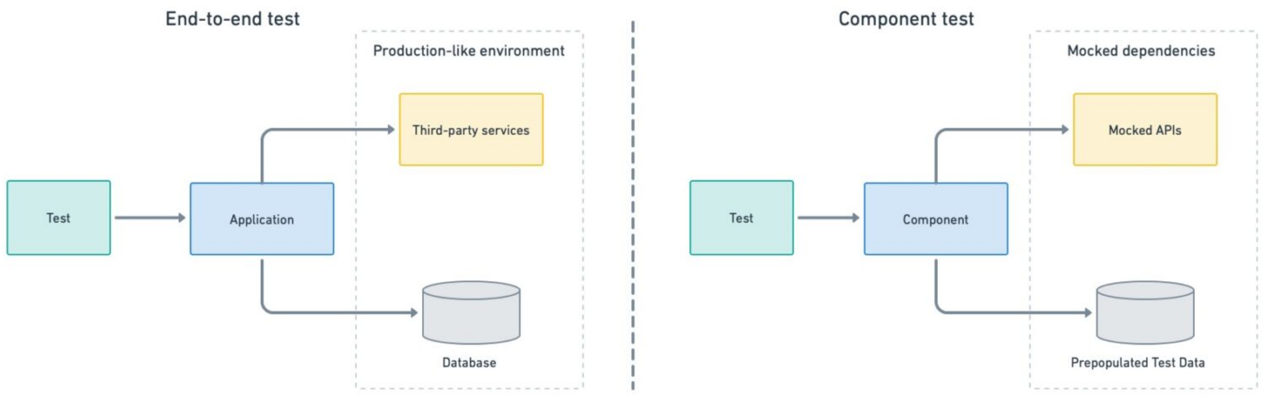


Figure 2.16: Difference between end-to-end testing and component testing [9, p. 1]

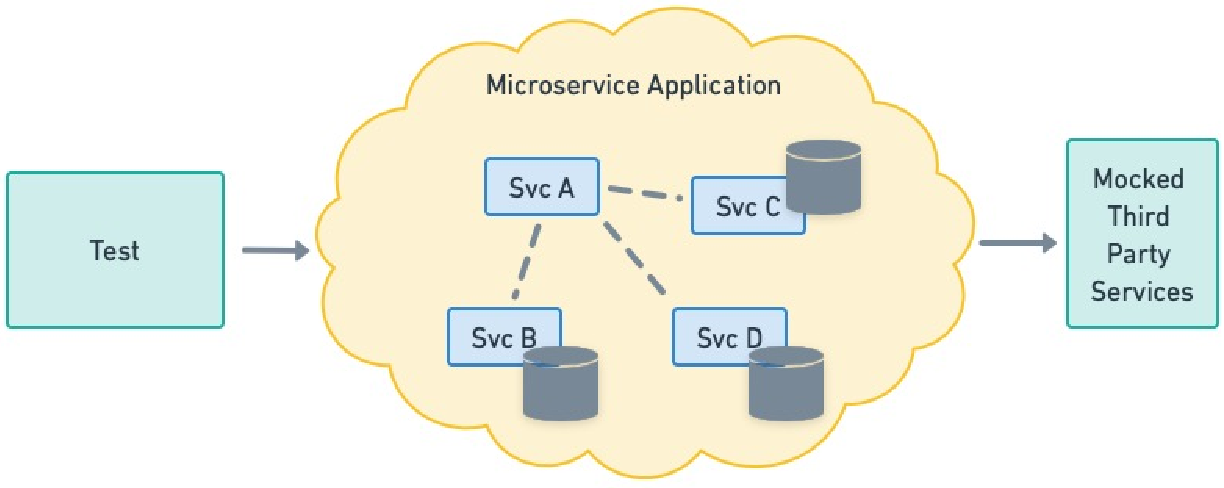


Figure 2.17: End-to-end testing [9, p. 1]

2.5 Deployment of MSA applications

MSA applications are usually deployed by containerizing the different microservices. A container is a lightweight, executable software package that bundles application code and dependencies including binary code, libraries and configuration files for simple deployment across many computer platforms [33]. These containers can be located on a physical server, in a cloud instance or both. The most popular types of containers are listed below:

- *Docker*: A well-known open-source platform that enables developers to wrap their software in a container, or separate environment. Docker makes advantage of a Linux kernel's features [33].
- *CRI-O*: A compact Red Hat-developed open-source container engine. As an alternative for Docker the runtime engine for Kubernetes, a well-known container orchestration system developed by Google, is the first implementation of CRI (Container Runtime Interface). With CRI-O, Kubernetes is compatible with any OCI (Open Container Initiative) compliant runtime [33].
- *rktlet*: An implementation of the CRI (Container Runtime Interface) for Kubernetes that competes with Docker. It makes use of CoreOS's rkt (also known as ”rocket”) as the main container runtime. To solve particular flaws in early versions of Docker, Rocket adopts a security-first strategy and makes use of a number of Linux server characteristics [33].
- *Containerd*: A daemon from the Cloud Native Computing Foundation for Linux and Windows. From image transfer through container execution and beyond, Containerd is capable of managing the complete lifespan of the container technology. Developers may utilize Kubernetes as the container runtime by using the containerd plugin cri [33].

Kubernetes is an open-source container orchestration tool that is developed by Google. A container orchestration tool allows for the system to have a high availability [10]. It also makes scaling a lot more efficient. Container orchestration tools also offer built tools such as disaster recovery tools which might be usefull. A Kubernetes architecture always includes at least one node, the master node (also known as the control plane). The control plane, or master node contains several components that helps this node orchestrate the cluster:

- *API server*: The API server manages the entrypoint to the Kubernetes cluster. This can be done by an User Interface (UI) or by Command Line Interface (CLI) [10]. The API server creates a REST interface to access.
- *Scheduler*: The task of distributing work among the multiple nodes is done by the scheduler [10]. It keeps an eye on resource availability and makes sure that a worker node's performance is within acceptable bounds [10].
- *Controller-Manager*: The controller-manager is in charge of monitoring the cluster's shared state to make sure it is functioning as it should. Actually, the controller manager is in charge of several controllers that react to different events, such when a node goes down [10].
- *Etcd*: Kubernetes employs etcd, a distributed key-value store, to exchange data about a cluster's general status. Furthermore, nodes can use the information stored there to configure themselves whenever they are produced [10].

Then any number of worker nodes can be connected to the master. A Kubernetes node controls and operates several pods. Each (worker) node has a Kubelet running on them. A Kubelet is a Kubernetes process that makes communication in the cluster possible [10]. Figure 2.18 shows the architecture of a Kubernetes cluster. Each worker node can run several containers. A node also includes a kube-proxy that is responsible for routing the traffic that is coming into the node from a service. The node also includes a container technology e.g. Docker, CRI-O, etc. A Kubernetes pod is a collection of containers and the lowest administrative unit that Kubernetes can handle. Each container in a pod has a unique IP address that is assigned to it [10]. Resources like memory and storage are shared by all the containers in a pod. This makes it possible to consider each Linux container inside a pod as a single application, just as if all the containerized processes were operating on the same host as they would in more conventional workloads [10]. When an application or service simply requires a single process to operate, it's usual to have a pod with just one container. Multi-container pods, as opposed to manually configuring shared resources amongst containers, simplify deployment setup as things become more involved and several processes must cooperate in order for things to operate properly [10]. For instance, there is a service that processed photographs and produced GIFs, one pod would include numerous containers that collaborate to resize images. One or more auxiliary (side-car) containers may be conducting batch background operations or clearing out data artifacts in the storage volume in addition to the primary container, which may be running the non-blocking microservice application that receives requests [10].

Another important concept is deployments. Deployments define the size of the application. A deployment is an instance of the application and it is possible to specify details to have multiple identical instances of these pods [10]. Deployments define the number of replications. Kubernetes will monitor the pod health and load in order to scale the system to the defined requirements [10]. Kubernetes clusters can be set up in a virtual or physical environment. Minikube can help with this process. Minikube is a minimal Kubernetes implementation that builds a single-node, basic cluster as a virtual machine on your local system. Systems running Linux, macOS, and Windows can use Minikube.

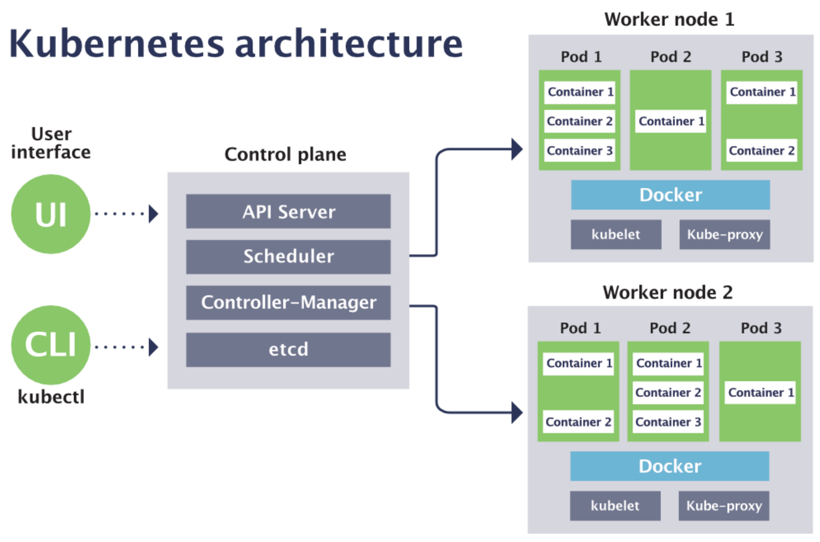


Figure 2.18: Kubernetes cluster architecture [10, p. 1]

2.6 Conclusion

In the early stages of an application, a monolithic approach can be interesting because the main goal of an early stage application is to verify whether the application is profitable. The monolithic architecture enables a quick proof of concept [26]. Customers can be brought to the system when this proof of concept is in place, and improvements can still be applied in the near future [34]. Another benefit of using a monolithic system in early stage development is that it is usually unclear what the exact specification of the system has to be. The precise requirements can therefore be determined at a later time according to the needs of the business. A monolithic system is therefore the more applicable architecture for small and rapid development because of the simplicity of testing and debugging [34].

Microservices are small reusable and easily scalable components that can reside in multiple physical and virtual locations. MSA brings additional complexity and higher costs for small projects. However, for large applications, this can be cost-effective, can influence high system performance, and speed up development [32]. Each service is responsible for its own specific task and can operate on its own database or share one with other services. This flexibility makes a MSA more interesting than monolithic architecture. However, this distribution of the services makes verifying if all the services are tested difficult. Testing can be done on the different levels. This thesis investigates which methodology ensures that all services are being tested. Thus, the methodology focuses on the integration level. There are a lot of code coverage tools for the unit level, but no code coverage tools for the integration level exist.

Chapter 3

Methodology

This chapter gives an overview of the proposed methodology to compute the coverage of a microservice application. The high level approach consists of four steps as shown in figure 3.1. They are:

- explore and define the MSA structure,
- choose (and prepare) the coverage metrics,
- deploy the MSA application and
- test at the different, appropriate levels, while monitoring the coverage metric.

This chapter mainly focusses on the first step, but for the sake of completeness also quickly addresses the following three.

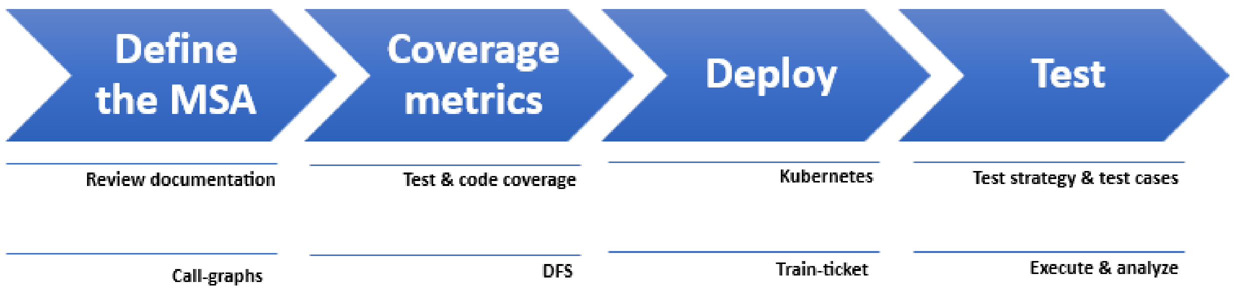


Figure 3.1: General steps to compute the coverage of a microservice application

Provided the MSA application is well documented, the MSA structure can be derived by reviewing this documentation. The aim of the first step is to build a call-graph as a representation of the MSA structure. Access to an adequate call-graph of the MSA will be imperative to apply the MSA coverage metric. Therefore, section 3.1 extensively discusses the most important factors and available tools for this process

Then, the coverage metric needs to be chosen. This thesis proposes to use a MSA coverage metric that somehow differs from both test and/or code coverage. The MSA coverage metric uses the MSA call-graph in combination with the Depth-First-Search (DFS) algorithm. Since this brings forth some new concepts, the whole of coverage detection is discussed separately in chapter 4.

The next step is to deploy the MSA application. Chapter 2 section 2.5 already explained how a microservice application can be deployed, therefore at this point section 3.3 will only give the link to the train-ticket example application.

Finally, section 3.4 concludes with an overview of test scenario's, adding a different point of view to the test levels already explained in the previous chapter. However, these test scenario's play an important role in the MSA coverage detection.

3.1 Define the MSA

Defining the MSA comprises the identification of the different microservices and their connection. To achieve this, a good understanding of the application architecture is required. This step will also expose the services that are the most critical to the overall functionality of the application. It is also important to take into account which services have the most impact on user experience (UX). There are a number of methods that can help by getting a better understanding of the application architecture and the links between different microservices. However, establishing a clear overview of the architecture can be challenging, particularly if the application is extensive and spread out across several services. Here are a number of methods listed to get a better overview of the composition of the application.

3.1.1 Review documentation

A microservices application should be documented correctly. By doing this, the developer creates a clear image of each service and its use. The documentation can then be reviewed to establish a better overview of the application structure. There are various tools that can help with this task:

- *Confluence*: Confluence is a tool for collaboration that was designed to be used for sharing, saving, and working on various projects. It gives the ability to manage teamwork more effectively by enabling you to produce project plans, record meeting minutes, and submit project needs simultaneously with other team members. Confluence also offers a simple interface and the ability to manage version control and collaboration features. Confluence is not necessarily made for microservice documentation only.
- *Swagger*: The Swagger API is a collection of free, open-source tools for programmers to create, describe, and utilize representational state transfer (REST) APIs. The tool's three parts—Swagger Editor, Swagger UI, and Swagger Codegen—are based on the OpenAPI specification.
- *RAML*: RESTful API Modeling Language also known as RAML is a method of providing a highly accessible description of practically-RESTful APIs for both people and machines. Because so few APIs genuinely adhere to all of REST's rules in the real world today, the term "practically RESTful" is used instead. RAML isn't rigid; for the time being, it concentrates on concisely specifying the resources, methods, parameters, replies, media types, and other HTTP structures that serve as the foundation for contemporary APIs that adhere to many, if not all, of the RESTful principles.
- *GitHub Wiki*: Every repository on GitHub.com has a wiki component, which is used to

contain documentation. There the developer may provide in-depth information about the project, such as how to use it, how he created it, or its guiding principles. A README file sums up the project's capabilities. For example, as explained in chapter 3 section 3.3, the general structure of the train-ticket application was given in the README, in the form of a diagram.

- *Read the Docs*: Software documentation is made simpler with Read the Docs by automatically creating, versioning, and hosting the documents. By treating it like code, developers can keep their team using the same tools and keep the documentation up to date.

3.1.2 Call-graphs

Creating a call-graph can give more insight in the application architecture, and give a better overview of the composition of the microservices. A dynamic call-graph is a record of how a program was executed, such as what is produced by a profiler. A dynamic call-graph can therefore be precise but can only represent one execution of the program. A call-graph that is meant to depict every conceivable execution of the program is called a static call-graph [35]. Static call-graph algorithms are often overestimates since determining the actual static call-graph may be an unsolvable job. This means that every call relationship that happens, as well as maybe some call relationships that would never exist in actual program runs, ares represented in the graph. Figure 3.2 gives an example of a call-graph.

The steps to create a call-graph are listed below:

1. *Identify the services*: The first step to create a call-graph is to identify which services you want to create a call-graph for and what languages these services are developed in. This is important because most call-graph tools are language specific. The identification of the services can be done by reviewing the documentation as explained earlier. This process is not always straightforward. Therefore communication with the developer team can help for identifying the services.

2. *Analyze communication patterns*: After the services are identified, it is important to analyze their communication patterns and how the services interact with each other. The most used patterns are explained in chapter 2 section 2.3. To examine the communication patterns, the source code, logs and network traffic can be inspected [35]. To make this task easier, there are a number of specialized tools that can be used as listed below. Table 3.1 gives the key-characteristics of the tools.

    - *OpenTelemetry*: A set of tools, APIs, and software development kits (SDKs) make up the open-source observability platform known as OpenTelemetry (also known as OTel). Otel gives IT teams the ability to produce, instrument, gather, and export telemetry data for analysis and to comprehend the behavior and performance of software. OTel also supports advanced features such as metric collection, logging and trace sampling. OTel can be used for Java, Python, Go, and .NET services.

    - *Jaeger*: Another open source software called Jaeger is used to track transactions across dispersed systems. It is utilized for diagnosing and monitoring sophisticated microservices setups. Jaeger provides vizualization methods for the tracing and support advanced features like anomaly detection. Jaeger supports different programming

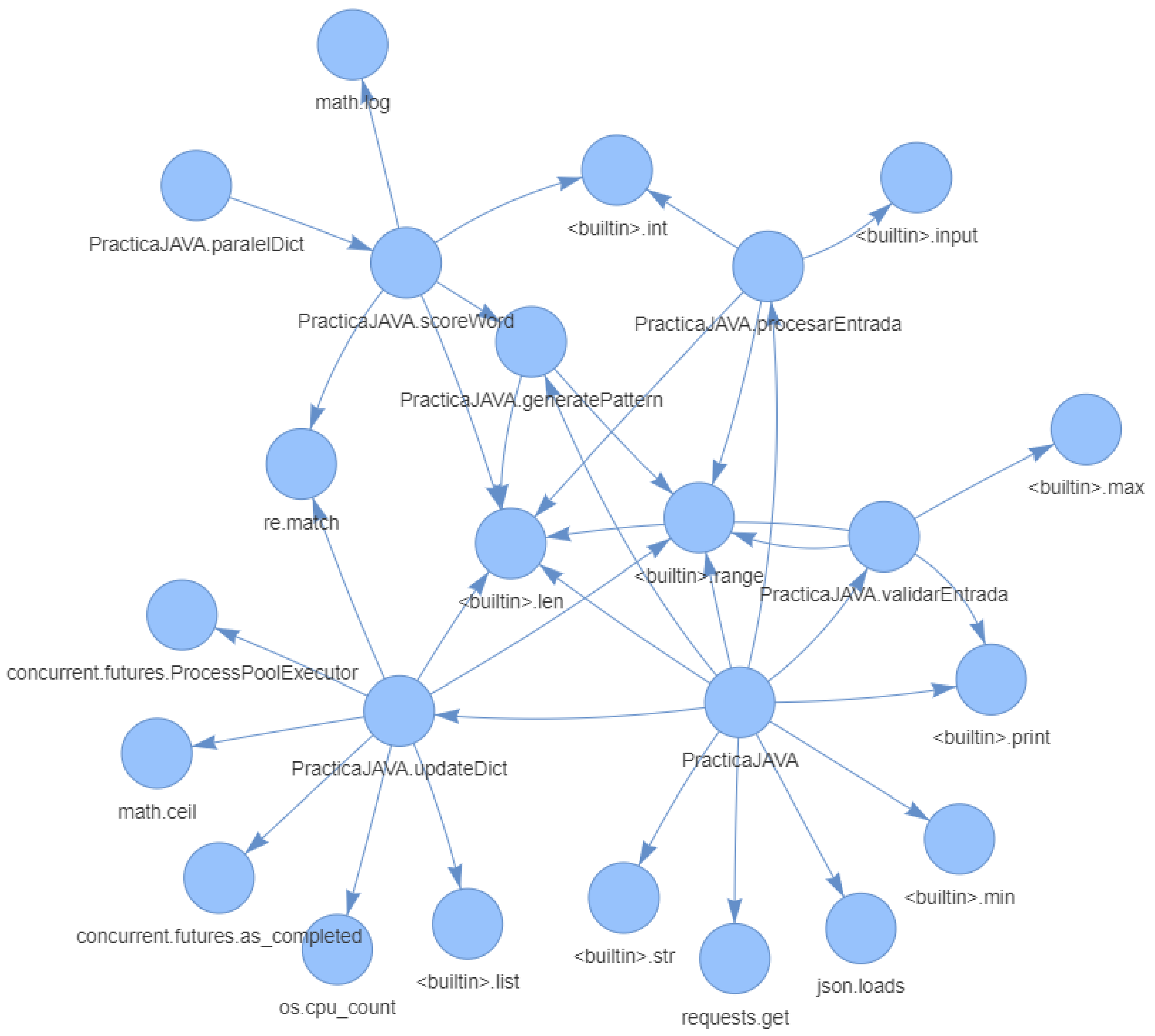


Figure 3.2: Example of a call-graph [11, p. 1]

languages such as: Java, Python, Go, and .NET.

- *Zipkin*: Zipkin assists in assembling the timing data required to resolve latency issues in service designs. The gathering and lookup of this data are features. Zipkin is an open-source distributed tracing system that support Java, Python, Go, and .NET services. Zipkin also contains advanced features such as context propagation and trace sampling.

- *AWS X-Ray*: This is a distributed tracing system provided by Amazon Web Services that requires a pay-per-use basis. AWS X-Ray is a service that gathers information about the requests that the application fulfills and offers tools to analyze, filter, and gain insights into that information in order to spot problems and areas for improvement. It is important to note that it can only be used for tracing applications running on AWS infrastructure. Integration with other AWS services including Lambda, EC2, and ECS is offered through AWS X-Ray.

Table 3.1: Different distributed tracing system tools and their characteristics

	Languages	License type
OpenTelemetry	Java, Python, Go, and .NET	No, Open-source
Jaeger	Java, Python, Go, and .NET	No, Open-source
Zipkin	Java, Python, Go, and .NET	No, Open-source
AWS X-Ray	C, C++, Fortran	Yes, pay-per-use

3. *Collect data*: In order for distributed tracing to function, the application code must be instrumented to output trace data as it responds to user requests [35, 36]. Each trace corresponds to a single request and includes details about the service that processed the request as well as any downstream services that were used. As explained earlier distributed tracing system tools gives better understanding in the communication patterns between microservices [35, 36]. The following steps describe how to collect the trace data from the tools.

   (a) *Instrument the code*: Distributed tracing system tools use instrumentation to release trace information when the program responds to queries [35, 36]. Then the output results in a trace for every request. This output includes details such as the service that was responsible for processing the request and additional downstream services that were used [35]. This is further explained in chapter 4.

   (b) *Define trace boundaries*: After the instrumentation, the boundaries of the trace should be defined. This concludes the start and end point of the request. Usually in a microservice based application this results in a trace that passes multiple services for a single request.

   (c) *Collect trace data*: The last step is to collect the trace data by using some sort of tracing agent [35, 36]. The trace data that is emitted by the application is then send over for analysis.

4. *Choose a tool*: First a tool has to be chosen to create the call-graph. Which tool to use depends on the programming language of the application, the framework that is used and the complexity of the architecture. However, it is also important to look at the licensing fee and the features that the tool brings with it. The ease of use and the integration possibilities with the environment that is used for the project are also important to be taken into account. It should be noted that in a microservice application each service can be written in a different programming language. This can make creating a call-graph with a single tool challenging. One way to work around this problem is to use a tool that supports multiple different programming languages such as Dynatrace and New Relic. Another approach is to use different tools for each different programming language and then combine these different call-graphs into one call-graph [35]. For instance, you could use Eclipse TPTP to create a call-graph for the Java service, a Python profiling tool like PyCharm for the Python service, and a Ruby profiling tool like RubyMine for the Ruby service if a microservice-oriented application has three services each written in Java, Python, and Ruby. The combined findings might then be used to build the application's overall call-graph. The list below gives an overview of the most popular tools for creating call-graphs Table 3.2 shows the different tools, their compatible programming languages

and the cost.

- *Eclipse TPTP*: Eclipse TPTP is a top-level Eclipse project for advanced Java development. Eclipse Test & Performance Tools Platform (TPTP) consists of the TPTP Platform Project, which offers a standard architecture and foundation for TPTP services. Eclipse TPTP contains tools for monitoring and logging system and application resources. The testing portion of the application lifecycle is substantially facilitated by the testing tools provided by the TPTP Testing Project for test preparation, deployment, and execution as well as execution history analysis and reporting. The TPTP Tracing and Profiling Project offers tools for tracing and profiling that may be used to gather and examine data on the performance of both distributed and single-system Java programs.

- *VisualVM*: VisualVM is a visual tool that combines JDK command-line tools and simple profiling features, created to be used throughout both development and production times. This tool is used to analyze and optimize java applications.

- *Intel VTune Amplifier*: Intel VTune Amplifier is a performance profiling tool that can show where in the code time is being spent in both serial and threaded applications. It may also quantify concurrency and pinpoint inefficiencies caused by synchronization primitives for threaded applications. Intel VTune Amplifier can be used for C, C++, and Fortran code.

- *Valgrind*: Valgrind is an instrumentation framework that is used to provide tools for dynamic analysis. Numerous memory management and threading issues may be automatically found using Valgrind tools, which can also profile your applications in great detail. Additionally, Valgrind can be used to create new tools. Valgrind can be used to create call-graphs for C, C++ and Fortran applications.

- *Dynatrace*: Dynatrace is a software-intelligence monitoring technology that promotes digital transformation by demystifying business cloud complexity [37]. The Dynatrace platform offers analysis about the performance of the application, their supporting infrastructure, and the experience of end users thanks to Davis (the Dynatrace AI causation engine) and full automation. With Dynatrace, business cloud operations can be modernized and automated, better software can be released more quickly [37].

Table 3.2: Different call-graph tools and their supported programming languages and cost

	Languages	License type
Eclipse TPTP	Java	No, Open-source
VisualVM	Java	No, Open-source
Intel VTune Amplifier	C, C++, Fortran	Yes, license
Valgrind	C, C++, Fortran	No, Open-source
Dynatrace	Non-specific	Yes, License

5. *Examine the call-graph*: In this step, the bottlenecks of the application as well as the performance issues can be identified. The steps below give an overview of how to analyze a call-graph.

   (a) *Identify and optimize the critical paths*: The critical paths are the paths that are accessed more frequently and have a significant influence on how well the application performs. These paths need to be optimized. This can be done by limiting the requests, optimizing the algorithm, caching the output, ...

   (b) *Identify and optimize the hot spots*: The hot spots are the microservices that get the most traffic and must respond quickly. It is important to optimize these hot spots. This can be done by limiting the requests, optimizing the algorithm and caching the output.

   (c) *Analyze the dependencies*: Search for and note any tight coupling or circular dependencies between the microservices [36]. These dependencies may cause performance problems or make it challenging to modify the program.

   (d) *Monitor the performance*: While improving the hot spots and critical paths, it is interesting to monitor the performance. This brings more insight into the effect of the changes that are implemented.

3.2 Coverage metrics

The next step is to choose a coverage metric. As discussed in chapter 2 section 2.4 microservices can be tested on many different levels and in many different ways. But, how much testing is enough? Should the application be tested completely, or should the most critical components of the application be prioritized? Luckily there are a number of metrics and methods for evaluating the efficacy of the tests, namely, test coverage and code coverage. Due to the similarity of their underlying concepts, both terms are occasionally used as synonyms. However, they differ from each other in several ways. Chapter 4 discusses the difference between code and test coverage, and why both metrics are important and useful to evaluate the quality and quantity of testing. Chapter 4 also explains the proposed methodology to calculate the service coverage. The Depth-First-Search (DFS) algorithm is used to achieve this goal.

3.3 Deployment of the MSA application

The following steps to be taken includes deploying the microservices in their own container as explained in chapter 2 section 2.5. Figure 3.3 shows the architecture of the microservices in the train-ticket application. However the deployment of the train-ticket service MSA application from `https://github.com/FudanSELab/train-ticket` did encounter some unidentified issues when creating the pods. The steps taken to deploy the train-ticket application are given in appendix A.

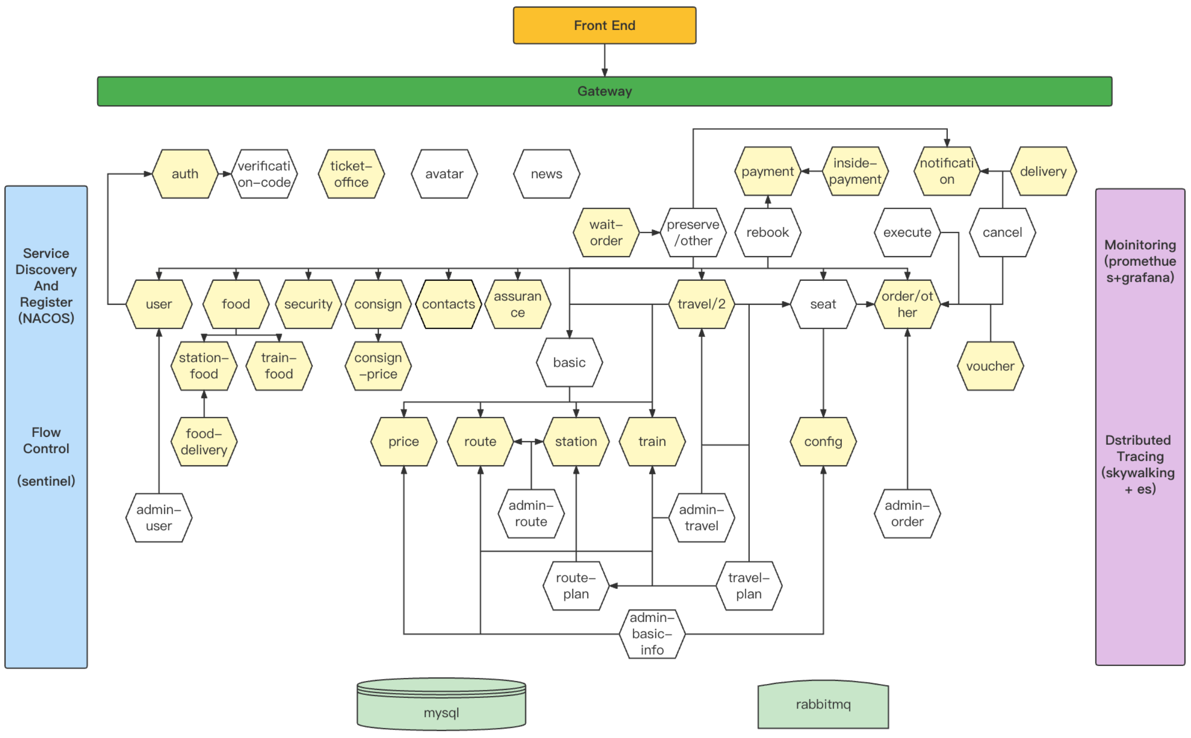


Figure 3.3: The train-ticket microservices application architecture [12, p. 1]

3.4 Testing of the MSA application

Finally the overall testing of the MSA can be done. However, before even starting to write the test, the scope that is going to be tested needs to be defined. This includes the determination on which level the application is going to be tested. As explained in chapter 2 section 2.4 the testing of microservices can be done on several levels. While integration tests examine how microservices interact with one another, unit tests are used to evaluate individual microservices in isolation. End-to-end tests validate the system's overall operation, whereas performance testing gauge the system's responsiveness to various loads. In a microservice-based system, the integration tests are used to identify defects when certain microservices communicate with each other. Integration tests use real services to verify the calls between the services and their cooperation and not mocked services like contract testing.

To start creating test cases, it is important to have a good understanding of the microservice architecture. Call-graphs and other tools explained in the previous section can help with this task. It is also important to determine their dependencies and communication protocols. The most popular communication protocols are explained in chapter 2 section 2.3. Once the testing levels are determined and the microservices structure is clear, the test cases can be defined. The test cases should include real scenarios that will occur, and not only the most possible but also the most exceptional cases [9]. Use cases or user stories can be used to define test scenarios. It is also important to determine positive and negative test scenarios. Whilst the positive scenarios describe the intended use, the negative scenarios should describe the exceptional not intended use. Test scenarios should also include error handling test cases and integration scenarios, where a microservice interacts with another service. Another important case would be a performance scenario, where the system is put under load to determine its responsiveness [9]. Security scenarios should also be taken into account. Figure 3.4 gives an overview of the most important test scenarios. The test cases should include all the different types as shown in figure 3.4.

Then the tests can be executed and should contain a component that applies the DFS algorithm (see chapter 4) and track how many unique tests were successfully performed. If some tests were not successfully performed, they should be updated before continuing with the coverage metrics. If the tests are all ran flawlessly, then it is possible to analyze the result and the measure of service coverage will be explained in chapter 4 section 4.3. If the tests all ran flawlessly, then it is possible to analyze the result and measure the service coverage as explained in chapter 4 section 4.3.

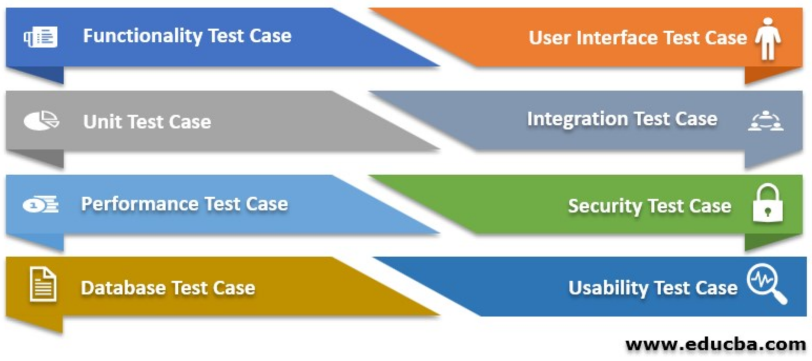


Figure 3.4: The most important test scenarios [13, p. 1]

Chapter 4

Coverage detection metrics

4.1 Test coverage

Test coverage incorporates testing the functions added as a result of the software requirements specification and the functional requirements specification [38–40]. Test coverage is a black-box testing methodology, as opposed to code coverage, which is a white-box testing methodology. Table 4.1 gives an overview of the characteristics of white box vs black box testing. The writing of test cases must ensure that all requirements from FRS (Functional Requirements Specification), SRS (Software Requirements Specification), URS (User Requirement Specification), etc. are covered to the fullest extent possible [38]. There is little to no chance of automation because the tests are derived from the aforementioned documents which are not formal enough the automatically derive tests from. [38–40]. Test coverage can be assessed using several forms of testing(, just like Code coverage). However, your business proposal will determine which form of testing you should conduct [38–40]. For instance, in user-centric online applications, usability testing, security tests, etc. may be more significant than functional tests, yet in other types of systems (like banking and finance), functional tests may take precedence. The following are a few test coverage mechanisms:

Table 4.1: Difference between black-box and white-box testing [1, p. 1].

Black-box testing	White-box testing
The black box test is a test that only considers the external behavior of the system; the internal workings of the software is not taken into account.	The white box test is a method used to test a software taking into consideration its internal functioning.
It is carried out by testers.	It is carried out by software developers.
This method is used in system testing or acceptance testing.	This method is used in unit testing or integration testing.
It is the least time consuming.	It is the most time consuming.
It is the behavior testing of the software.	It is the logic testing of the software.
It is also known as data-driven testing, functional testing, and closed box testing.	It is also known as clear box testing, code-based testing, structural testing, and transparent testing.

- *Unit testing*: At the unit/module level, this kind of testing is carried out. Bugs that are found at the unit level may not be the same problems found after integration [38].
- *Functional testing*: Functional testing involves comparing the functions and features to the specifications listed in the Functional Requirement Specification (FRS) [38].
- *Integration testing*: Since the program is tested on a system level, it is also known as system testing. Once all necessary components have been integrated, this kind of testing is conducted [38].
- *Acceptance testing*: The outcome of the acceptance testing will determine whether the product is made available to the final user or client. Before pushing the code changes from the Staging environment to the Production, the developers must receive a green light from the testers and SMEs of the web application [38].

The other crucial thing to remember is that depending on the testing level, the meaning and purpose of test coverage can change [38]. Additionally, it depends on the kind of product that is subjected to black-box testing. Metrics for measuring test coverage on e-commerce websites and mobile devices might be different. Below are some classifications:

- *Features coverage*: Here, the test cases are created so as to provide the broadest possible coverage of the aspects of the products [38]. For instance, if a tester is tasked with testing a phone dialer application, he should confirm that the length of the number being dialed is appropriate. If testing is conducted in India, the mobile number should have a maximum of 10 digits; otherwise, an error message will flash [38]. Similar to this, the product team's priorities must be followed when testing all required and optional features.

- *Risks coverage*: There is a section in every product/project requirement document that discusses the risks and mitigations related to the project [38]. Some risks must be handled during the testing process even as others (such as changes in business dynamics) are outside the purview of the planning, development, and test team [38]. For instance, while creating a company website, the server architecture should be configured such that pages can be accessed quickly [38]. The closest server should be selected to load the website depending on the location (such as the country, city, etc.) from which the website is accessed; otherwise, the overall experience would be impaired. The test team should also conduct a load test, in which a performance test is done when many users are attempting to visit the website at once, i.e. in a situation where there is heavy website traffic. Poor test findings could lead to a user experience that is below average, which is a very serious risk [38].
- *Requirements coverage*: Here, tests are designed to provide the broadest possible coverage of the requirements for the product as stated in the various Requirement Specification documents [38]. For instance, you must ensure that the default language is configured correctly while testing a pre-installed SMS application [38]. The default SMS language should be Chinese for clients in countries where English is not the predominant language, such as China, and English for all other customers, such as those in India [38].

It is possible to construct a test case that enters an email address without the @ symbol and then tries to proceed with the login in order to test the failure scenario in a straightforward email login-page [38]. This would test the login page's functioning and see if the logic for checking the email address' format complies with the requirements [38]. The advantage of test coverage is that it is an effective method for testing software features and comparing the outcomes across several specification documents (requirements, feature, product, UI/UX, etc.). Since these tests are run at the 'feature level' and not the 'code level', a manual comparison of the test results with the anticipated output is required. There is no accurate way to gauge test coverage. Hence, the results of the coverage tests can differ from tester to tester and heavily depend on the tester's domain expertise.

4.2 Code coverage

Code coverage shows the percentage of the code that is covered by the test cases, as determined by manual testing and test automation using Selenium or another framework. For instance, if your source code contains a straightforward if...else structure, the test code would need to cover both if and else to get 100% code coverage [38], [40]. The majority of code coverage techniques rely on static instrumentation, in which statements that track the execution are placed at strategic points in the code [41]. Even if the addition of instrumentation code increases the size and execution time of the entire application, the additional overhead is negligible in comparison to the data produced by the execution of the instrumented code [38], [40].

The main purpose of unit tests is to test the code at the level of individual units. Since the developer writes the unit tests, he/she has a better understanding of the tests that belong in the unit testing process [38], [40]. There will always be debate over the precise amount of tests that make up unit testing, but it does help to raise the software's overall quality. Does the test suite contain a sufficient number of test scenarios? Or should there be more tests? All of these queries can be answered through code coverage.

New features and fixes (to the defects discovered during testing) are introduced to the release cycle as the product development moves forward. This implies that in order to keep up with the software modifications made during development, the test code may also need to be modified [38], [40]. With successive release cycles, it's crucial to uphold the testing criteria that were established at the project's outset. Code coverage can be used to ensure that your tests adhere to these requirements. Typical code coverage subtypes include:

- *Branch coverage*: Branch coverage also known as decision coverage, is a technique used to guarantee that every branch that might be used in a decision-making process is actually used. As an illustration, if you use an if... else conditional statement or a do...while statement you must make sure that all of the branches—If, Else, Do, and While—are tested using the appropriate inputs in order to have a full branch coverage [38].
- *Function coverage*: Function coverage ensures that all required functions, particularly exported functions and APIs, are tested. This should also entail evaluating the functions using various input parameters because doing so would also examine the functions' internal logic. Function coverage would be 100% once every function in the code had been tested [38].
- *Statement coverage*: Statement coverage requires test code to be constructed in a way that it ensures that each executable statement in the source code is run at least once. Corner cases and border cases are included in this [38].
- *Loop Coverage*: Loop Coverage uses the strategy where every loop in the source code is run at least once. In order to make the code failsafe, it may contain some loops that could run based on the outcomes you obtained at runtime. It is crucial to test these loops as well [38].

Figure 4.1 shows two conditions, two branches and four paths. If both branches are at least tested once, then that would result in a 100% branch coverage. In the figure, this is achieved when the yellow and the blue lines are tested. For a path coverage of 100%, every single possible path should be tested. In figure 4.1 this is the case when the blue, orange, yellow and green lines are tested. This makes clear that path coverage is stronger than branch coverage

Instrumentation is a method used to check the code coverage. Instrumentation can be used to track down mistakes in the source code, inject trace information and monitor performance [40], [41]. There are various types of instrumentation, and depending on the instrumentation approach being utilized, there can be a little performance (and timing) overhead [39], [38]. The three main categories of instrumentation are as follows:

- *Code instrumentation*: Code instrumentation uses instrumentation statements that are added after the source code has been compiled. The normal toolchain should be used for compilation; when compilation is successful, instrumented assembly is produced. You can insert instrumentation lines at the beginning and end of a function, for instance, to measure the time it takes to run a specific function in your code [38], [41].
- *Runtime instrumentation*: As opposed to the code instrumentation method, information is gathered here while the code is being executed, or in the runtime environment [38], [39].
- *Intermediate code instrumentation*: In this kind of instrumentation, byte codes are added to the compiled class files to create an instrumented class [38], [39].

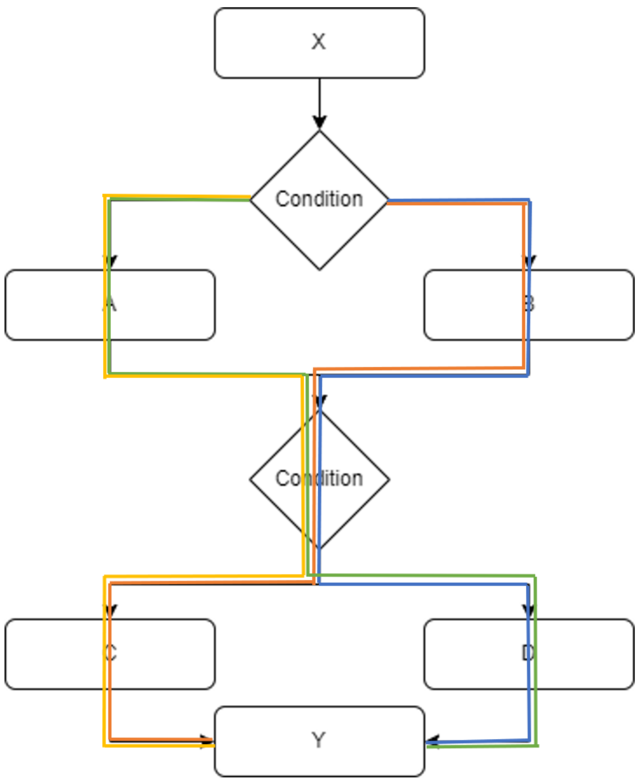


Figure 4.1: Branch and path coverage

The percentage of code that has been tested using test cases or test suites is the measurement metric for code coverage [38]. Thus, the test results may be quantified, i.e. the code coverage was 80 lines out of 100 Lines of Code. This indicates that 80% of the code is covered.

Code coverage explains how well your test code is working and how you might make the coverage better. Most code coverage tools are only capable of doing unit tests [38, 40]. It might not be possible to compare the code coverage results of one tool to another because of the diverse methodologies utilized by various tools [38]. Few tools exist that support various programming languages, such as Java, Python, C, etc. Therefore, if the developing team uses several programming languages (for the development of test code), they might need to use more than one tool [38].

The metrics used to compare the effects of code coverage and test coverage are completely dissimilar. Test coverage is determined by the features that are covered by tests, whereas code coverage is determined by the proportion of code that is covered during testing [38]. Since both test coverage and code coverage are crucial to software projects, they are frequently used together [38].

4.3 MSA coverage detection on integration level

This section gives an analytical approach on how to determine the MSA coverage on an integration level. The goal here is to verify whether all microservices are tested (called). Thus, MSA coverage is a kind of code coverage. The proposed methodology applies the Depth-First-Search (DFS) algorithm to the call-graphs at hand to establish the coverage benchmark. Since the DFS-algorithm is a key component in the MSA coverage detection, it's imperative to first explain it's working principle before continuing with it' role in the MSA coverage metric.

4.3.1 DFS

DFS generally traverses a complete graph in $O(|V|+|E|)$, where $|V|$ is the number of vertices and $|E|$ the number of edges [14]. In the case of the microservices, $|V|$ is the number of microservices and $|E|$ the number of links. To continue in this section, the terms vertices and edges will be used. There are a number of edge types that can be defined. Figure 4.2 shows a graph example with different directional edges types. DFS also works on cyclic and directional graphs which is required for a microservices-based application since the links between microservices can create a loop between microservices.

- *Tree* edges are edges that can be found in the tree after the graph was subjected to the DFS algortihm. Edge (u, v) is a tree edge if v was first discovered by exploring edge (u, v) [14].
- *Back* edges connects a vertix u to an ancestor v [14]. Self-loops are also considered to be back edges.
- *Forward* edges are non-tree edges that connects a vertex u to a descendant v [14].
- *Cross* edges are all other edges [14].

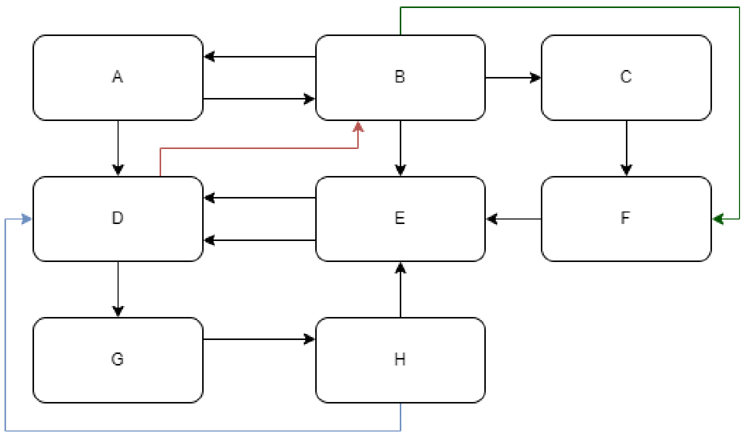


Figure 4.2: Example of a directed cyclic graph, the red line is an cross-edge, the blue line is a back edge and the green line is a forward edge, the other edges are tree edges [14].

DFS starts by selecting a random vertice v and chooses a connected edge (v, w) [14]. Then it continues down this edge until there is nowhere else to go. After that the algorithm backtracks. Let's assume the graph of vertices and the algorithm starts with vertices A as shown in figure 4.3.

1. Each vertex is first set to white by the algorithm to signify that it has not yet discovered it [14]. It also sets the parent of each vertex to be null.

2. The process starts by choosing one vertex u from the graph, giving it discovery time 0, and changing its color to gray to show that the vertex has been discovered (but isn't yet complete) [14].
3. DFS-Visit is called repeatedly for each vertex v that is a member of the set $Adj[u]$ and is still tagged as white, giving each vertex the proper discovery time $d[v]$ (the time variable is increased at each step) [14].
4. If v has no white descendants, v turns black, is given the proper completion time, and the algorithm goes back to looking at v's ancestor, $p[v]$ [14].
5. The method ends when there are no more white vertices in the network and all of vertex u's descendants are black; otherwise, a new "source" vertex is chosen from the remaining white vertices, and the process is repeated [14].

Starting from node U in figure 4.3, either node V or Y can be discovered [14]. Suppose that node V is discovered, which has a single outgoing edge to node W. W has no outgoing edges, so this node is finished, and the algorithm returns to V . From V there is no other choice, so this node is also finished and the algorithm returns to U [14]. From node U it is possible to discover Y and its descendants, and the procedure continues similarly. At stage (l) the DFS algorithm has discovered and finished nodes U, V , W, X, Y . Selecting node Q as a new starting node, the algorithm can discover the remaining nodes (in this case Z) [14].

4.3.2 Using DFS for the MSA coverage metric

The output of the DFS algorithm is a list with the visited vertices or in our case visited microservices. If DFS would be applied to a call-graph like figure 4.3 a possible output would be $[U, V, W, X, Y, Q, Z]$. This list is an overview of all the microservices in the system and will act as the denominator in the coverage ratio. It represents the number of tests required for full coverage and can be determined offline (before testing starts).

The effective test phase, implementing a test scenario from section 3.4, must now, similar to DFS, track how many unique tests were successfully performed and which microservices were addressed. E.g. if the DFS list from the test phase would look like this:$[U, V, W, Y, Q, Z]$ then it can be stated that microservice X has not been covered. In this case the ratio would be $[U, V, W, Y, Q, Z]/[U, V, W, X, Y, Q, Z]$, resulting in a MSA coverage of $6/7 \cdot 100$, which is roughly 86%.

So, to summarize, the MSA coverage metric is the ratio of the uniquely used MSA (calls) while testing in relation to the maximum possible unique MSA (calls) as determined by the DFS algorithm applied to the call-graph.

If the coverage is not a full 100% additional tests covering the missed services need to be added. The tester can check which microservice is not covered by comparing the list of the called services (nominator of ratio) to the full list determined by the DFS algorithm (denominator of the ratio).

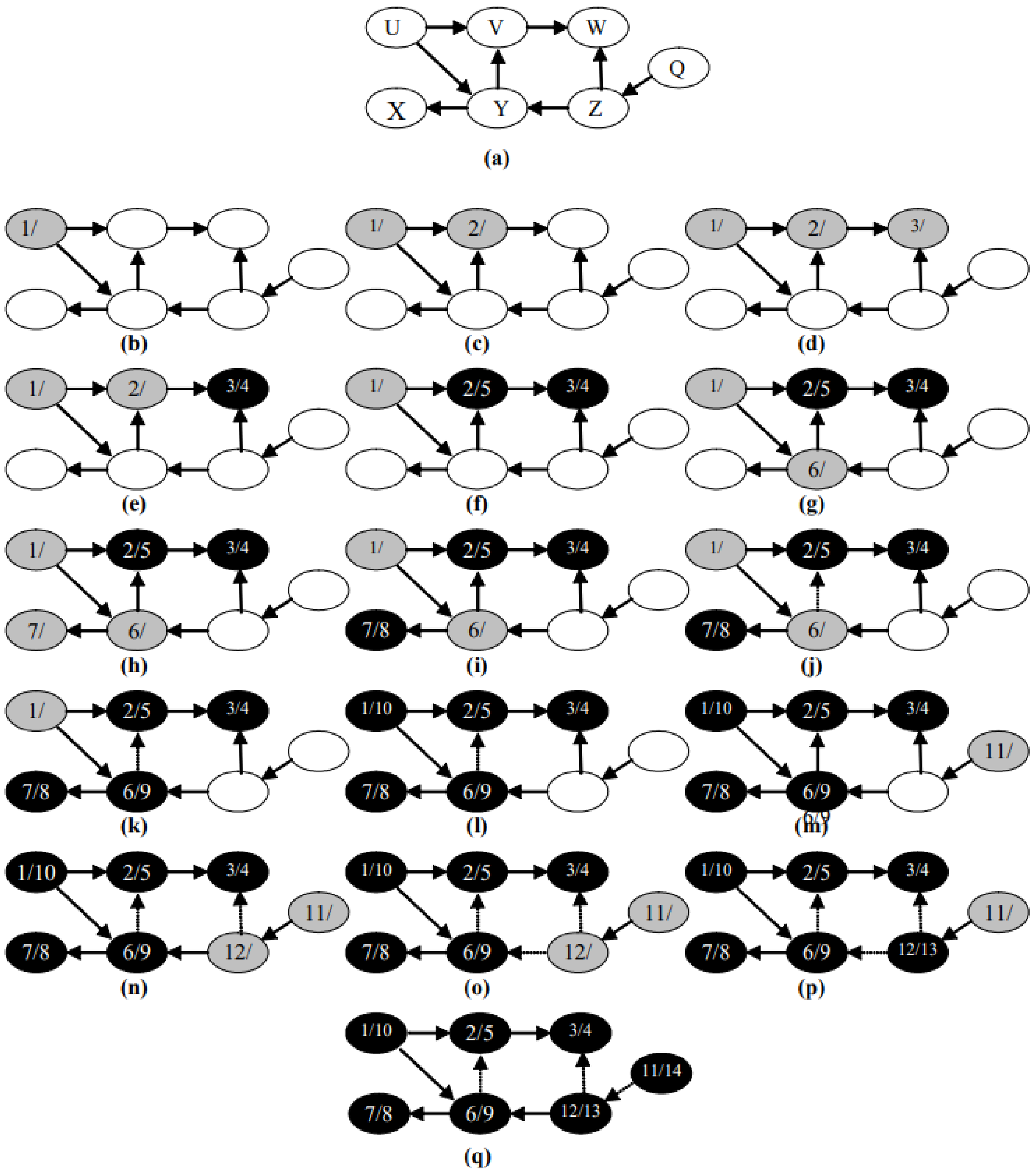


Figure 4.3: The progress of the DFS algorithm for the graph of (a). [14, p. 3]

Chapter 5

Conclusion

Microservices are small reusable and easily scalable software components that can reside in multiple physical and virtual locations. However, this distribution of the services makes verifying whether all the services are tested difficult. This thesis proposes a methodology to ensure that all services are covered while testing.

The literature study gives a comprehensive comparative overview of characteristics of monolithic versus microservice software architectures and elaborates on coverage metrics for (microservice) software testing at different levels. There are a lot of code coverage tools for the unit level, but no code coverage tools for the integration level and certainly no automatic ones, exist. Hence, this thesis focuses on microservice testing at the integration level.

The proposed methodology for applying coverage detection for a microservice architecture (MSA) application is a high-level approach consisting of four steps: define the MSA, prepare the coverage metrics, deploy and test. For each step tips and existing tools that can help with the process are presented.

The result of the first step must be a complete call-graph of the microservices structure of the application at hand. Next, the Depth-First-Search (DFS) algorithm is used to initially determine (offline) the number of tests or calls required for full coverage. Finally, during deployment and actual testing the used microservices should be tracked similar to the principles of the DFS algorithm. The ratio of the used microservice calls and the overall required ones is then the searched metric for microservice coverage. If the coverage is not a full 100% extra (functional) tests, that address the missed microservices, need to be added

Unfortunately, the deployment of the train-ticket system, as an example application, encountered unidentified server issues. Hence, the proposed methodology was not (yet) validated experimentally. This could be subject of future work.

Bibliography

[1] PractiTest, “Black box vs white box testing.” https://www.practitest.com/qa-learningcenter/resources/black-box-vs-white-box-testing/. Accessed: 2023-04-25.

[2] R. Haddad, “Migrating from monolith to microservices: How do feature flags fit in?.” https://www.flagship.io/migrating-from-monolith-to-microservices/. Accessed: 2023-05-10.

[3] A. Manchanda, “Monolithic vs microservices architecture: Which option is right for your enterprise?.” https://www.netsolutions.com/insights/monolithic-vs-microservices/. Accessed: 2023-05-10.

[4] “Microservices.” https://martinfowler.com/articles/microservices.html. Accessed: 2023-03-01.

[5] Oleksii, “How to understand microservices architecture.” https://datamify.com/architecture/how-to-understand-microservices-architecture/, Octobre 2021. Accessed: 2023-03-10.

[6] IONOS, “What is http.” https://www.ionos.com/digitalguide/hosting/technical-matters/what-is-http/, 2020. Accessed on 15 March 2023.

[7] Wallarm, “What is advanced message queuing protocol (amqp)?.” https://www.wallarm.com/what/what-is-amqp, 2021. Accessed on 15 March 2023.

[8] B. Chiradeep, “What is mqtt (mq telemetry transport)? working, types, importance, and applications.” https://www.spiceworks.com/tech/iot/articles/what-is-mqtt/, 2022. Accessed on 15 March 2023.

[9] T. Fernandez and D. Ackerson, “Testing strategies for microservices,” *Semaphore*, p. 1, 2020.

[10] J. Spaleta, “How kubernetes works.” https://sensu.io/blog/how-kubernetes-works. Accessed: 2023-05-02.

[11] D. G. Solla, “What is a call graph? and how to generate them automatically.” https://www.freecodecamp.org/news/how-to-automate-call-graph-creation/. Accessed: 2023-04-25.

[12] Anonymous, “Train ticketa benchmark microservice system.” https://github.com/FudanSELab/train-ticket#Using-Kubernetes. Accessed: 2023-04-25.

[13] T. Swati, “Types of test case.” https://www.educba.com/types-of-test-case/. Accessed: 2023-05-04.

[14] P. Charalampos, "Depth first search & directed acyclic graphs," *Graph Algorithms*, pp. 1–28, 2004.

[15] Anonymous, "Steps to deploy train-ticket." `https://ttdoc.oss-cn-hongkong.aliyuncs.com/Steps.pdf`. Accessed: 2023-02-18.

[16] "Cloudsea.ai – cloud, software engineering, evolution, and assessment with ai." `https://research.tuni.fi/cloudsea/about/`. Accessed: 2023-03-01.

[17] J. Mejia, M. Muñoz, Rocha, A. Peña, and M. Pérez-Cisneros, *Trends and Applications in Software Engineering.* Advances in Intelligent Systems and Computing, Warsaw: Springer, 2018.

[18] N. Deeraj, X. Ronghua, N. Seyed Yahya, and C. Yu, "A microservice-enabled architecture for smart surveillance using blockchain technology," *Dept. of Electrical & Computer Engineering*, pp. 1–4, 2018.

[19] N. Alshuqayran, N. Ali, and R. Evans, "A systematic mapping study in microservice architecture," *IEEE 9th International Conference on Service-Oriented Computing and Applications*, pp. 1–8, 2016.

[20] F. Tapia, M. Mora, F. Walter, A. Hernán, E. FLores, and T. Theofilos, "From monolithic systems to microservices: A comparative study of performance," *Applied sciences*, pp. 1–36, 2020.

[21] D. Taibi, V. Lenarduzzi, and C. Pahl, "Processes, motivations, and issues for migrating to microservices architectures: An empirical investigation," *IEEE Cloud Computing vol. 4 no. 5*, pp. 22–32, 2017.

[22] J. Soldani, D. A. Tamburri, and W.-j. Van Den Heuvel, "The pains and gains of microservices: A systematic grey literature," *Systems and Software*, pp. 215–232, 2018.

[23] R. Lichtenthäler, M. Prechtl, C. Schwille, T. Schwartz, P. Cezanne, and G. Wirtz, "Requirements for a model-driven cloud-native migration of monolithicweb-based applications," *SICS Software-Intensive Cyber-Physical Systems*, pp. 1–36, 2020.

[24] D. Taibi and K. Systä, "A decomposition and metric-based evaluation," *TASE - Tampere Software Engineering Research Group*, pp. 1–17, 2019.

[25] R. V. O'Connor, P. Elger, and P. M. Clarke, "Continuous software engineering—a microservices architecture perspective," *Journal of software: evolution and process*, pp. 1–12, 2017.

[26] G. Blinowski, A. Ojdowska, and A. Przybylek, "Monolithic vs. microservice architecture: A performance and scalability evaluation," *IEEE Acces*, pp. 1–8, 2021.

[27] T. Poniszewska-Maranda, J. Macioch, B. Borowska, and W. Maranda, "Mechanisms for transition from monolithic to distributed architecture in software development process," *29th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pp. 1–8, 2021.

[28] R. Abdul and G. Shahbaz A. K., "A systematic mapping study: The new age of software architecture from monolithic to microservice architecture—awareness and challenges," *Computer Applications in Engineering Education*, pp. 1–31, 2022.

[29] M. Ahmadvand and A. Ibrahim, "Requirements reconciliation for scalable and secure microservice (de)composition," *IEEE 24th International Requirements Engineering Conference Workshops*, pp. 1–6, 2016.

[30] C. Esposito, A. Castiglione, and K.-K. Raymond Choo, "Challenges in delivering software in the cloud as microservices," *IEEE Cloud Computing*, pp. 1–5, 2016.

[31] F. Auer, V. Lenarduzzi, M. Felderer, and D. Taibi, "From monolithic systems to microservices: An assessment framework," *Information and Software Technology*, pp. 1–12, 2021.

[32] C. de La Torre, B. Wagner, and M. Rousos, *.NET Microservices Architecture for Containerized .NET Applications, available.* New York: Microsoft Developer Division, .NET and Visual Studio product teams, 2022.

[33] "What is container technology?." `https://www.solarwinds.com/resources`. Accessed: 2023-05-02.

[34] Oleksii, "How to understand monolithic architecture." `https://datamify.com/architecture/how-to-understand-monolithic-architecture/`, Octobre 2021. Accessed: 2023-03-10.

[35] L. Shutian, X. Huanle, L. Chengzhi, Y. Kejiang, X. Guoyao, Z. Liping, H. Jian, and X. Chengzhong, "An in-depth study of microservice call graph and runtime performance," *IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS, VOL. 33, NO. 12*, pp. 1–14, 2022.

[36] L. Shutian, X. Huanle, L. Chengzhi, Y. Kejiang, X. Guoyao, Z. Liping, D. Yu, H. Jian, and X. Chengzhong, "Characterizing microservice dependency and performance: Alibaba trace analysis," *ACM Symposium on Cloud Computing (SoCC '21)*, pp. 1–15, 2021.

[37] Dynatrace, "What is dynatrace?." `https://www.dynatrace.com/support/help/get-started/what-is-dynatrace`. Accessed: 2023-04-25.

[38] H. Sheth, "Code coverage vs test coverage: Which is better?," December 2019. Accessed: 2023-04-12.

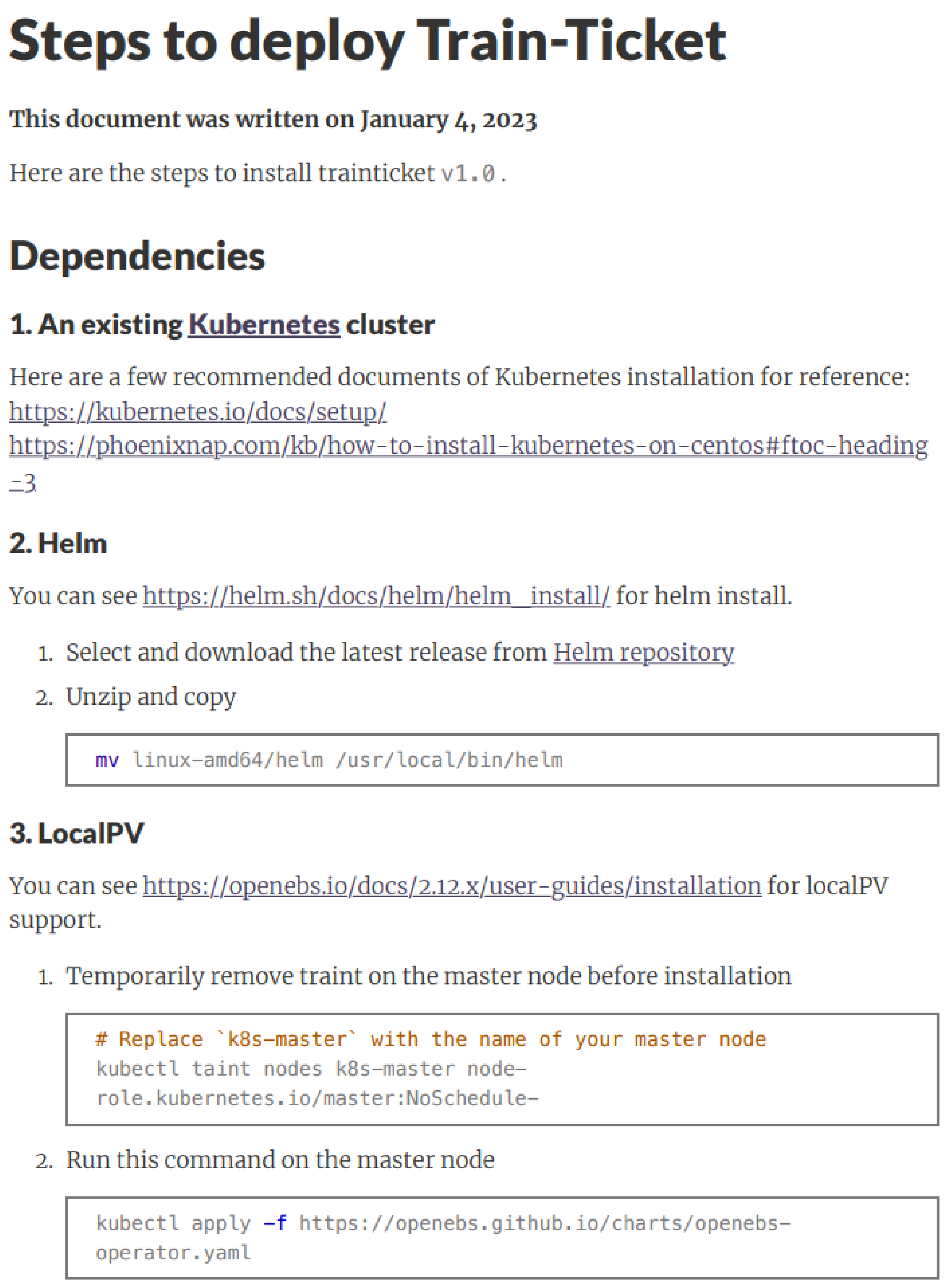
[39] H. Ferenc, G. Tamas, B. Arpad, T. David, B. Gergo, and G. Tibor, "Code coverage differences of java bytecode and source code instrumentation tools," *Software Qual J*, pp. 1–45, 2017.

[40] A. Pilgun, O. Gadyatskaya, Y. Zhauniarovich, S. Dashevskyi, A. Kushniarou, and S. Mauw, "Fine-grained code coverage measurement in automated black-box android testing," *Software Engineering and Methodology*, pp. 1–35, 2017.

[41] M. Ivanković, G. Petrović, R. Just, and G. Fraser, "Code coverage at google," *ReaerchGate*, pp. 1–10, 2019.

Appendix A

Steps for the deployment of the train-ticket application



Steps to deploy Train-Ticket

This document was written on January 4, 2023

Here are the steps to install trainticket `v1.0` .

Dependencies

1. An existing Kubernetes cluster

Here are a few recommended documents of Kubernetes installation for reference:
https://kubernetes.io/docs/setup/
https://phoenixnap.com/kb/how-to-install-kubernetes-on-centos#ftoc-heading-3

2. Helm

You can see https://helm.sh/docs/helm/helm_install/ for helm install.

1. Select and download the latest release from Helm repository
2. Unzip and copy

```
mv linux-amd64/helm /usr/local/bin/helm
```

3. LocalPV

You can see https://openebs.io/docs/2.12.x/user-guides/installation for localPV support.

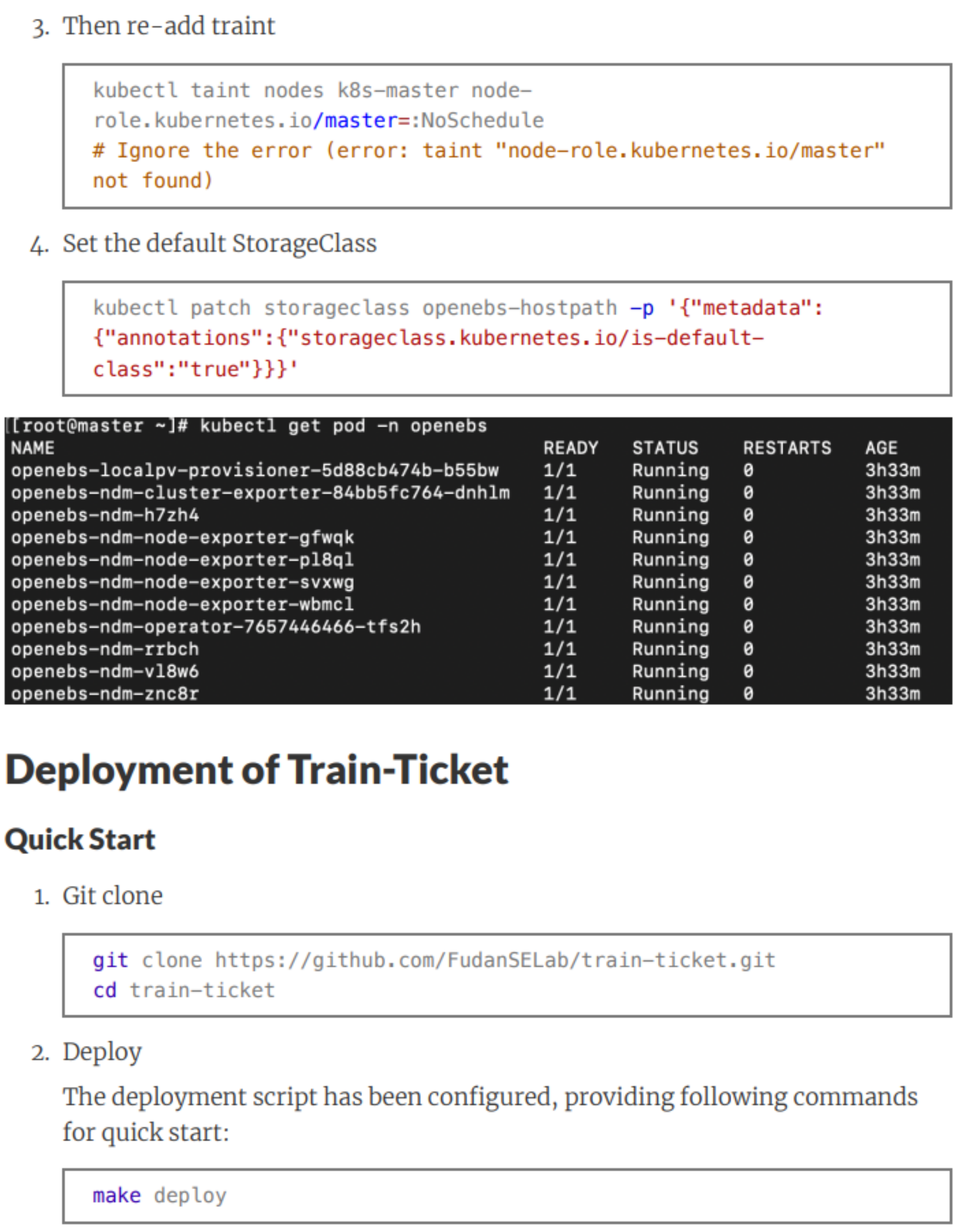
1. Temporarily remove traint on the master node before installation

```
# Replace `k8s-master` with the name of your master node
kubectl taint nodes k8s-master node-
role.kubernetes.io/master:NoSchedule-
```

2. Run this command on the master node

```
kubectl apply -f https://openebs.github.io/charts/openebs-
operator.yaml
```

Figure A.1: Steps to establish correct dependencies [15, p. 1]



3. Then re-add traint

```
kubectl taint nodes k8s-master node-
role.kubernetes.io/master=:NoSchedule
# Ignore the error (error: taint "node-role.kubernetes.io/master"
not found)
```

4. Set the default StorageClass

```
kubectl patch storageclass openebs-hostpath -p '{"metadata":
{"annotations":{"storageclass.kubernetes.io/is-default-
class":"true"}}}'
```

```
[root@master ~]# kubectl get pod -n openebs
NAME                                           READY   STATUS    RESTARTS   AGE
openebs-localpv-provisioner-5d88cb474b-b55bw   1/1     Running   0          3h33m
openebs-ndm-cluster-exporter-84bb5fc764-dnhlm  1/1     Running   0          3h33m
openebs-ndm-h7zh4                              1/1     Running   0          3h33m
openebs-ndm-node-exporter-gfwqk                1/1     Running   0          3h33m
openebs-ndm-node-exporter-pl8ql                1/1     Running   0          3h33m
openebs-ndm-node-exporter-svxwg                1/1     Running   0          3h33m
openebs-ndm-node-exporter-wbmcl                1/1     Running   0          3h33m
openebs-ndm-operator-7657446466-tfs2h          1/1     Running   0          3h33m
openebs-ndm-rrbch                              1/1     Running   0          3h33m
openebs-ndm-vl8w6                              1/1     Running   0          3h33m
openebs-ndm-znc8r                              1/1     Running   0          3h33m
```

Deployment of Train-Ticket

Quick Start

1. Git clone

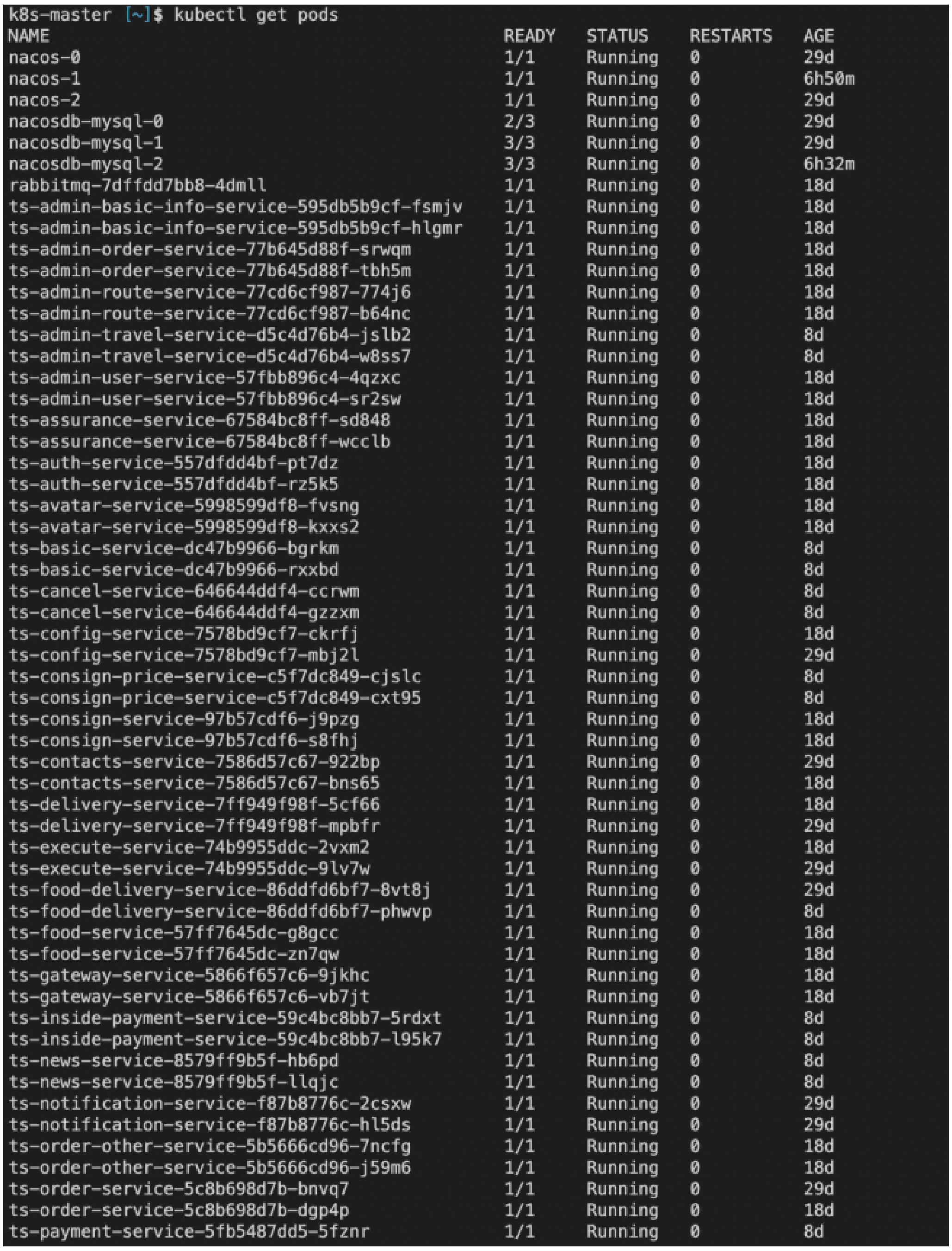
```
git clone https://github.com/FudanSELab/train-ticket.git
cd train-ticket
```

2. Deploy

The deployment script has been configured, providing following commands for quick start:

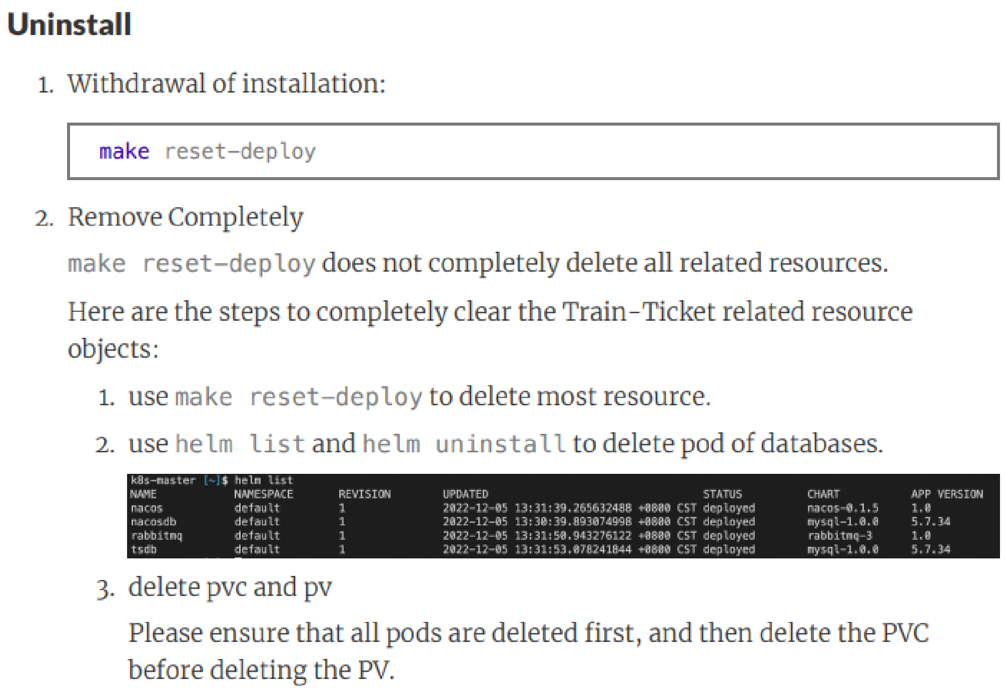
```
make deploy
```

Figure A.2: Steps to establish correct dependencies and deployment [15, p. 1]



```
k8s-master [~]$ kubectl get pods
NAME                                          READY   STATUS    RESTARTS   AGE
nacos-0                                       1/1     Running   0          29d
nacos-1                                       1/1     Running   0          6h50m
nacos-2                                       1/1     Running   0          29d
nacosdb-mysql-0                               2/3     Running   0          29d
nacosdb-mysql-1                               3/3     Running   0          29d
nacosdb-mysql-2                               3/3     Running   0          6h32m
rabbitmq-7dffdd7bb8-4dmll                     1/1     Running   0          18d
ts-admin-basic-info-service-595db5b9cf-fsmjv  1/1     Running   0          18d
ts-admin-basic-info-service-595db5b9cf-hlgmr  1/1     Running   0          18d
ts-admin-order-service-77b645d88f-srwqm       1/1     Running   0          18d
ts-admin-order-service-77b645d88f-tbh5m       1/1     Running   0          18d
ts-admin-route-service-77cd6cf987-774j6       1/1     Running   0          18d
ts-admin-route-service-77cd6cf987-b64nc       1/1     Running   0          18d
ts-admin-travel-service-d5c4d76b4-jslb2       1/1     Running   0          8d
ts-admin-travel-service-d5c4d76b4-w8ss7       1/1     Running   0          8d
ts-admin-user-service-57fbb896c4-4qzxc        1/1     Running   0          18d
ts-admin-user-service-57fbb896c4-sr2sw        1/1     Running   0          18d
ts-assurance-service-67584bc8ff-sd848         1/1     Running   0          18d
ts-assurance-service-67584bc8ff-wcclb         1/1     Running   0          18d
ts-auth-service-557dfdd4bf-pt7dz              1/1     Running   0          18d
ts-auth-service-557dfdd4bf-rz5k5              1/1     Running   0          18d
ts-avatar-service-5998599df8-fvsng            1/1     Running   0          18d
ts-avatar-service-5998599df8-kxxs2            1/1     Running   0          18d
ts-basic-service-dc47b9966-bgrkm              1/1     Running   0          8d
ts-basic-service-dc47b9966-rxxbd              1/1     Running   0          8d
ts-cancel-service-646644ddf4-ccrwm            1/1     Running   0          8d
ts-cancel-service-646644ddf4-gzzxm            1/1     Running   0          8d
ts-config-service-7578bd9cf7-ckrfj            1/1     Running   0          18d
ts-config-service-7578bd9cf7-mbj2l            1/1     Running   0          29d
ts-consign-price-service-c5f7dc849-cjslc      1/1     Running   0          8d
ts-consign-price-service-c5f7dc849-cxt95      1/1     Running   0          8d
ts-consign-service-97b57cdf6-j9pzg            1/1     Running   0          18d
ts-consign-service-97b57cdf6-s8fhj            1/1     Running   0          18d
ts-contacts-service-7586d57c67-922bp          1/1     Running   0          29d
ts-contacts-service-7586d57c67-bns65          1/1     Running   0          18d
ts-delivery-service-7ff949f98f-5cf66          1/1     Running   0          18d
ts-delivery-service-7ff949f98f-mpbfr          1/1     Running   0          29d
ts-execute-service-74b9955ddc-2vxm2           1/1     Running   0          18d
ts-execute-service-74b9955ddc-9lv7w           1/1     Running   0          29d
ts-food-delivery-service-86ddfd6bf7-8vt8j     1/1     Running   0          29d
ts-food-delivery-service-86ddfd6bf7-phwvp     1/1     Running   0          8d
ts-food-service-57ff7645dc-g8gcc              1/1     Running   0          18d
ts-food-service-57ff7645dc-zn7qw              1/1     Running   0          18d
ts-gateway-service-5866f657c6-9jkhc           1/1     Running   0          18d
ts-gateway-service-5866f657c6-vb7jt           1/1     Running   0          18d
ts-inside-payment-service-59c4bc8bb7-5rdxt    1/1     Running   0          8d
ts-inside-payment-service-59c4bc8bb7-l95k7    1/1     Running   0          8d
ts-news-service-8579ff9b5f-hb6pd              1/1     Running   0          8d
ts-news-service-8579ff9b5f-llqjc              1/1     Running   0          8d
ts-notification-service-f87b8776c-2csxw       1/1     Running   0          29d
ts-notification-service-f87b8776c-hl5ds       1/1     Running   0          29d
ts-order-other-service-5b5666cd96-7ncfg       1/1     Running   0          18d
ts-order-other-service-5b5666cd96-j59m6       1/1     Running   0          18d
ts-order-service-5c8b698d7b-bnvq7             1/1     Running   0          29d
ts-order-service-5c8b698d7b-dgp4p             1/1     Running   0          18d
ts-payment-service-5fb5487dd5-5fznr           1/1     Running   0          8d
```

Figure A.3: List of the deployed pods [15, p. 1]



Uninstall

1. Withdrawal of installation:

```
make reset-deploy
```

2. Remove Completely

   `make reset-deploy` does not completely delete all related resources.

   Here are the steps to completely clear the Train-Ticket related resource objects:

   1. use `make reset-deploy` to delete most resource.
   2. use `helm list` and `helm uninstall` to delete pod of databases.

```
k8s-master [~]$ helm list
NAME        NAMESPACE   REVISION    UPDATED                                 STATUS      CHART           APP VERSION
nacos       default     1           2022-12-05 13:31:39.265632488 +0800 CST deployed    nacos-0.1.5     1.0
nacosdb     default     1           2022-12-05 13:30:39.893074998 +0800 CST deployed    mysql-1.0.0     5.7.34
rabbitmq    default     1           2022-12-05 13:31:50.943276122 +0800 CST deployed    rabbitmq-3      1.0
tsdb        default     1           2022-12-05 13:31:53.078241844 +0800 CST deployed    mysql-1.0.0     5.7.34
```

   3. delete pvc and pv

      Please ensure that all pods are deleted first, and then delete the PVC before deleting the PV.

Figure A.4: Steps to uninstall the train-ticket application [15, p. 1]