Aalto University
School of Electrical Engineering
Department of Electrical Engineering and Automation

Joel Huttunen

Microservice Testing Practices in Public Sector Software Projects

Master's Thesis
Espoo, April 18, 2017

Supervisor:	Ville Kyrki, D.Sc. (Tech.)
Advisor:	Jari Pääkkö, M.Sc. (Tech.)

Aalto University
School of Electrical Engineering
Department of Electrical Engineering and Automation

ABSTRACT OF MASTER'S THESIS

Author:	Joel Huttunen		
Title:	Microservice Testing Practices in Public Sector Software Projects		
Date:	April 18, 2017	**Pages:**	ix + 83
Professorship:	Automation Technology	**Code:**	AS-84
Supervisor:	Ville Kyrki, D.Sc. (Tech.)		
Advisor:	Jari Pääkkö, M.Sc. (Tech.)		

Online services are constantly evolving, which makes service maintainability challenging. This has led to microservice architecture, where big applications are split into smaller services in order to improve applications' maintainability, scalability, and flexibility.

However, splitting a single process application into multiple services causes the testing process to be more challenging. This Master's thesis is exploring these testing problems in a microservice context and finding practical guidance for the test implementation.

Moreover, this Master's thesis focuses on public sector software projects. Public sector software projects are clearly predefined and the provider has open information about the project's needs. Thus, the project has a clear goal and known boundaries right from the beginning.

The research approach for this study is an exploratory multiple case study consisting of three case projects. The data of the case projects were collected through semi-structural interviews and version history commit analysis.

The results of this study present a set of successful practices and recommendations for taking testing into account during a microservice oriented agile development process. Successful testing requires monitoring of the project's maturity level to focus testing resources at the right time. Additionally, the case projects brought up practical testing guidance, such as understanding of the common testing responsibility, the importance of peer review, and the value of assigning a specific tester after the project has reached its end-to-end testing phase.

Keywords: microservice, testing practices, case study, software maturity, software architecture

Language: English

Aalto-yliopisto
Sähkötekniikan korkeakoulu
Sähkötekniikan ja automaation laitos

DIPLOMITYÖN TIIVISTELMÄ

Tekijä:	Joel Huttunen		
Työn nimi:	Mikropalveluiden testauskäytännöt julkisen sektorin projekteissa		
Päivämäärä:	18. huhtikuuta 2017	**Sivumäärä:**	ix + 83
Professuuri:	Automaatiotekniikka	**Koodi:**	AS-84
Valvoja:	TkT Ville Kyrki		
Ohjaaja:	DI Jari Pääkkö		

Web-palvelut kehittyvät jatkuvasti, mikä vaikeuttaa palveluiden ylläpitoa. Yhtenä ratkaisuna on palvelun pilkkominen osiin mikropalveluiksi. Palvelun pilkkominen edistää palvelun ylläpitoa, skaalattavuutta ja joustavuutta.

Toisaalta palvelun pilkkominen mikropalveluiksi vaikeuttaa testausprosessia. Tämä diplomityö tutkii mikropalveluiden testausprosessiin liittyviä ongelmia ja etsii käytännönläheistä ohjeistuista testien toteuttamiseen mikropalveluympäristössä.

Diplomityö keskittyy julkisen sektorin mikropalveluprojekteihin, koska kaikki tässä diplomityössä käytetyt tutkimusprojektit ovat julkisen sektorin hallinnoimia. Julkisen sektorin ohjelmistoprojektit ovat selkeästi esimääriteltyjä ja projektien aineisto on avoimesti saatavilla. Tämän takia projekteilla on selkeä päämäärä ja tunnetut rajat heti projektin alussa.

Tutkimusmenetelmänä käytettiin tutkivaa case study -menetelmää. Tutkimus sisälsi kolme tutkimuskohdetta. Tutkimusdata kerättiin osittain jäsennetyillä kontekstuaalisilla haastatteluilla ja ohjelmistokoodin versiohallinnan historian analyysillä.

Tuloksena syntyi kokoelma hyväksi todettuja käytäntöjä ja suosituksia, jotka auttavat ottamaan testauksen huomioon mikropalvelun iteratiivisessa ohjelmistokehitysprosessissa. Suositeltaviksi testauskäytännöiksi havaittiin projektin maturiteetin tarkkaileminen, että testauksen resursointi voidaan tehdä oikeaan aikaan. Lisäksi, projekteista nousi esiin muita suosituksia, kuten kehitystiimin yhteisen testaamisvastuun ymmärtäminen, koodikatselmoinnin merkitys ja erillisen testaajan tärkeys, kun projektin maturiteetti on kasvanut riittävästi.

Avainsanat:	mikropalvelut, testauskäytännöt, tapaustutkimus, ohjelmiston maturiteetti, ohjelmistoarkkitehtuuri
Kieli:	Englanti

Preface

First, I would like to thank my advisor M.Sc. Jari Pääkkö for his excellent support, feedback and facilitating during the process. I want to thank my supervisor D.Sc. Ville Kyrki for his time and valuable and inspiring feedback.

I am grateful to Finnish Institution of Occupational Health and to Population Register Center for allowing me to use their projects in my Master's Thesis. I want to also thank Gofore for giving me time to write my thesis and my colleagues for their support and feedback. I am also thankful of the weekly supportive discussions with Peter Kronström and Mika Huttunen's great feedback.

I would also like to thank the experts who were involved in the interviews for this research project. Without their passionate participation and input, this thesis would not have been conducted.

Finally, I express my very profound gratitude to my family, roommates, and friends for their encouragement throughout my years of study and writing this thesis. This accomplishment would not have been possible without them. Thank you.

Helsinki, April 18th, 2017

Joel Huttunen

Abbreviations

API	Application Programming Interface
AWS	Amazon Web Services
CD	Continuous Delivery
CDC	Customer-Driven Contract
CI	Continuous Integration
CRUD	Create, Read, Update, and Delete
DEVOPS	Software Development and Information Technology Operations
E2E	End-to-End
ESF	European Social Fund
ESP	Enterprise Services Platform
ET	Exploratory Testing
FIOH	Finnish Institute of Occupational Health
GUI	Graphical User Interface
HTTP	Hypertext Transfer Protocol
JSON	JavaScript Object Notation
KAPA	National Architecture for Digital Services
REST	Representational State Transfer
RPC	Remote Procedure Calls
SOA	Service-Orientated Architecture
SOAP	Simple Object Access Protocol
SUT	System Under Test
TDD	Test-Driven Design
UAT	User Acceptance Testing
UI	User Interface
WIP	Work In Progress

Contents

List of Tables

List of Figures

Chapter 1

Introduction

This section presents a short introduction to this thesis. The section begins with Section 1.1 giving a short background of microservices. Section 1.2 presents the aims and objects of this study. It explains what kind of testing challenges microservices include. These challenges define the research problem of this thesis. The scope is restricted to public sector software projects, which are explained in Section 1.2. The methods of the case study will be shortly discussed in Section 1.3. Finally, Section 1.4 concludes with a thesis outline.

1.1 Background and Motivation

A typical piece of software consists of several relations between modules which make testing more difficult. Nowadays, web based and business oriented software are increasingly implemented by using a microservice architectural style [1]. Business orientation means that the service has a clear business goal and allows small latencies between services. For example, a service providing a form is business oriented, because it has a clear business goal - to provide a form to a user. The service could allow latencies up to 100 ms, because the user is not able to recognize the delay within this scope. [2]

Microservices can be defined as one implementation of Service Orientated Architecture (SOA) [2]. Previous common implementations of SOA have been enterprise oriented. For example, Web Services are one common implementation of SOA [4]. On the other hand, microservices have typical common features like independence, micro databases, and infrastructure automation [5]. Thus, microservices can also be described as a separate

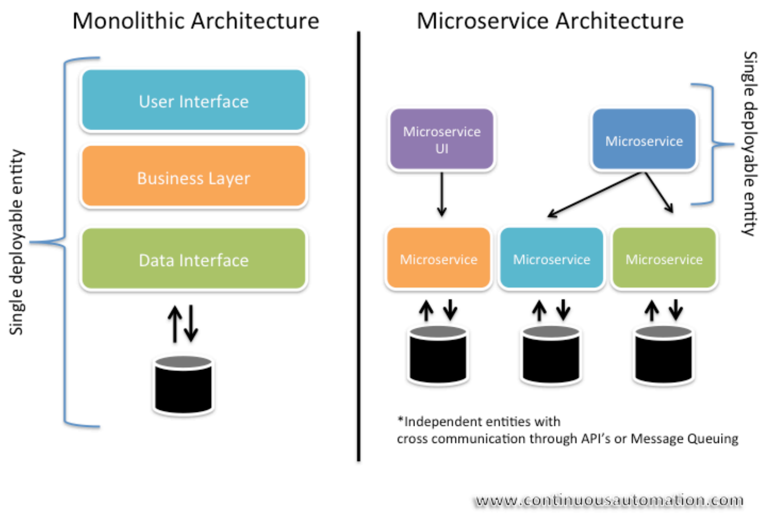

Figure 1.1: Microservice deployment modularity [3].

architecture [6]. However, definition of a microservice is still forming. Currently, microservices are defined to be small, autonomous services which have a focused purpose [2].

The goal of microservices is to break an application into smaller pieces to make the application more maintainable (see Figure 1.1). In web software development, modules, libraries or even languages out-date rapidly [1]. This has created a demand for an easily re-programmable software architecture. Microservices do not add limitations for the inner implementation of the service [1]. Furthermore, microservice architecture supports *DevOps* practices by providing small and more focused systems for the delivery process. *DevOps* consists of a set of practices which typically minimize the delivery time between development and production environments [7]. Moreover, microservices are usually developed in agile teams [1]. The teams are responsible of the whole journey of the implemented property. The journey includes planning, implementation, testing, and maintaining the solution.

Microservices are not a silver bullet which solves all problems of the service oriented software design. One simplified way to describe microservices is that it moves complexity from the modules to the network layer. Thus, the software structure should be well-known before the implementation. However, this is not guaranteed in software development. The communication structure should be consistent between the microservices, so shifting between microservices would not require unnecessarily broad programming knowledge. [2]

1.2 Research Problem

This thesis aims to study how testing should be done in public sector microservice projects. Software testing is used to ensure the quality of the service, which is impossible to measure only by software metrics [8, Chapter 8]. Thus, the thesis uses an exploratory research approach, which includes semi-structured interviews from three different public sector software projects.

The outcome of this study is a set of recommendations for different testing practices, approaches, and models that should be considered when starting a public sector microservice project. The thesis compares the findings and practices with literature recommendations.

Overall, testing is used to decrease the frequency of defects in software. Testing can be divided into two distinct goals according to Sommerville [8, Chapter 8]:

- *"To demonstrate to the developer and the customer that the software meets its requirements".*
- *"To discover situations in which the behavior of the software is incorrect or undefined."*

Testing cannot ensure that the software is bug-free or that it will behave according to specifications in every situation. Thus, software monitoring is also required, before the software is released into production. Testing allows discovery of errors but does not show their absence [8, Chapter 8]. In this study, service monitoring is considered as a testing tool, because microservices requires service monitoring. Service monitoring allows finding bugs from the software like dedicated testing.

Single process architectures, i.e., monoliths, are running as one task on top of the operating system, whereas microservices have established multiple processes to extend scalability. Thus, microservices includes distributed software environment problems in their testing, monitoring, and development environments.

Research questions

The main research problem is the following: *What kind of testing practices are software engineers recommending for public sector microservice software projects?*

The research problem is divided into smaller research questions, which are the following:

- How does testing a microservice architecture differ from testing a monolithic architecture?
- How does a microservice environment influence testing?
- Why do project testing practices differ from the recommendations?
- How does a project's release process affect testing?

1.3 Thesis Scope

The thesis scope is restricted into public sector oriented microservice architectures, which are implemented in agile teams. The thesis includes three different case projects. The case projects have varying maturity level, size, and development practices. Nevertheless, all case projects have a clearly defined goal right from the beginning like most public sector oriented software projects. Public sector projects also provide a clear framework of technologies that are recommended or even required to be used because those have been evaluation criteria in the competitive tendering. Moreover, public sector software projects have similarities in the organizational structure. For example, all case projects have a dedicated part-time product owner to facilitate the software team.

1.4 Thesis Outline

Chapter 2 explains the research approach and methods used in this study. It also describes the case projects and explains the case selection. Chapter 3 introduces the background of the research project. It includes literature and practices from other known cases. Overall, the chapter summarizes previous related work done in the area of this thesis. Chapter 4 presents the results that were obtained from the studied case projects. The cases are initially described in Chapter 1.3, but here the thesis goes more into case details.

Chapter 5 compares literature practices to the results and groups them into scalable categories. Chapter 6 answers to the research questions and makes the final conclusions. This chapter points out the main findings, challenges, and compares the findings to the literature practices.

Chapter 2

Research methodology

This section presents the research methods and case projects used in this study. Section 2.1 goes through case study research approach. Next, Section 2.2 explains the selection of the case projects. Section 2.3 presents the case projects' backgrounds. After that, Section 2.4 shows how data collection is done through interviews. Finally, Section 2.5 explains used data analysis methods and estimates their validity and limitations.

2.1 Research approach

This thesis uses an exploratory multiple case study method [9] to find answers to the research questions presented in Section 1.2. The exploratory case study method allows finding out what is happening and seeking new insights. The results can be used to identify microservice testing practices and to bring up the main microservice testing questions. These results can also be used as a hypothesis for another research. Basically, the thesis is structuring good practices for microservice testing. [9]

The case study research process has five major process steps [9]:

1. Defining research questions
2. Planning data collection
3. Collecting evidence
4. Analysis of collected data
5. Reporting

The first step is to *define research questions.* The research questions are formed around the research object which is usually a program, an entity, a person, or a group of people in a case study research. Each of these objects has complex connections to social, political, historical, and personal issues. Thus, the researcher observes the object from multiple different directions to gather a good overall picture of the case project. With a good overall picture, a researcher is able to answer the research questions, and produce convincing evidence about differences between the case projects. A case study research's interviews are aiming for quality over quantity [10].

A case study usually finds answers to questions, which begin with *"how"* or *"why"* [11]. These questions are formulated after a literature review. A literature review gives a background information about the problem, brings up insightful questions, and ensures that the research questions are relevant to the problem. A researcher can also complete a pilot study to ensure that the research approach is relevant to the problem. [10] This thesis is monitoring microservice testing as an entity through different viewpoints which came up during the interviews. The viewpoints are discussed in Section 5.

The second step is *planning the data collection.* The thesis is using multiple real-life case projects to bring variety in project size. The multiple cases should be treated as a single case in the case study methodology [10]. Thus, case projects are divided into separate sections. The conclusions of the case projects can be used as information to form overall conclusions, but the case project conclusions should be presented separately [10].

Multiple cases provide supportive theories through similar results, but they also can contrast results for predictable reasons. Moreover, the theory should be changed if the upcoming results do not work as a *theoretical framework* has predicted. Hence, a theoretical framework should have clear limitations and conditions when the framework can be used right from the beginning. [11] Furthermore, according to Bratthall and Jørgensen [12]: *"A multiple data source case study is more trustworthy than single data source case study"*. They argue that a multiple case study will never provide statistical significance, but instead, it brings many kinds of evidence, which can be linked together to support a solid conclusion.

The third step is *data collection.* A case study requires a systematic organization of the data because a researcher handles large amounts of data from multiple sources. Moreover, systematic techniques help to keep track of the original research purpose and prevent getting overwhelmed by the research data. Additionally, investigators should be trained to perform the

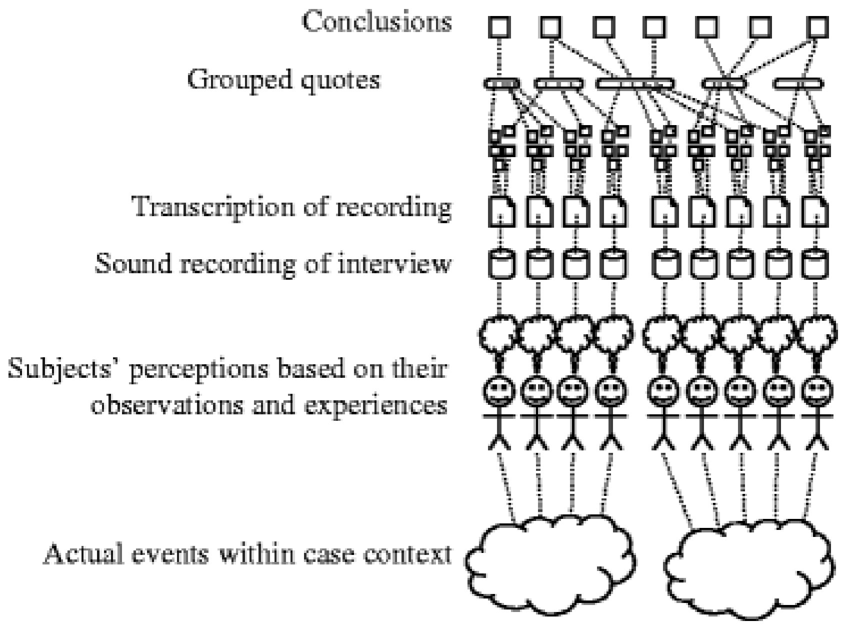
Conclusions
Grouped quotes
Transcription of recording
Sound recording of interview
Subjects' perceptions based on their observations and experiences
Actual events within case context

Figure 2.1: Interview results' analyzing process [9, sec. 5].

interviews trustworthily. [10] This thesis has only 10 interviews, so I decided to complete all case project interviews by myself. The desired skills of an investigator are the following; an investigator should be a good listener, adaptive and flexible, be able to ask questions, interpret answers, have a good understanding of the subject, and sense novelty of the answers [11].

The research data should be collected from multiple sources and stored systematically. Case studies use field notes to categorize the data for later use. However, I did not have team members to help me with the interviews, so I ended up using the projects' documentation as supportive material. Furthermore, it is important to store the raw interview material, for instance, by recording the interview. Field recordings include feelings, intuitive hunches, pose questions, and document the work in progress. These attributes warn about impending bias in the recordings. The results also point out if the inquiry needs to be reformulated. Hence, results should maintain the connection between the case project, the inquiry, and the interview. [10]

The next step is *data evaluation.* The raw research data should be examined from multiple viewpoints to generate several interpretations from the data. After this, interpretations can be used to find linkages between the research object and the outcomes. Multiple data collection methods allow data source triangulation and strengthen the findings. [10]

Researchers categorize, tabulate and recombine data to find linkages between the data. Figure 2.1 shows the general process, how conclusions are formed in a case study process. Research data grouping can be done by using different data visualization methods, for example by using arrays, matrices of categories, flow charts, or tabulating frequency events. The grouping process requires constant data cross-checking and a good understanding of the data validity. [10]

The final step is *reporting*. In reporting, researchers should pay attention to present sufficient evidence to convince the reader. The confidence of a reader can be achieved by showing that all paths of the research object have been explored, pointing out all boundaries of the study, and disclosing conflicting propositions. [10]

Case studies are commonly presented case by case in chronological order. This means that case projects are presented like stories each in their own section. The conclusions of the research are composed of the case projects' results. Finally, the case study should be reviewed by a professional. Researchers can, for example, use audience groups, journalists or participants to review their study. [10]

Overall, this research procedure is quite straightforward, but it also sets some constraints; the case study should have a clear hypothesis at the research stage. Sometimes this is hard to achieve and the data collection methods should be modified at the middle of the research. However, in a case study, changing the data collection form invalidates the previous results. Thus, in uncertain research cases, a pilot case study could ensure that the data collection method is valid. Valid data should produce evidence that allows answering the research questions. [11] Nevertheless, I had previous experience with the subject, so I felt that a pilot case study would be unnecessary. However, I piloted the research questions with one non-expert from the field.

The case study process has also some variations. For example, Eisenhardt [13] adds two research process steps between steps 4 and 5. These steps are *sharping hypothesis* and *enfolding literature*. This thesis concentrates on the basic 5 step version, but I suppose hypothesis sharping happens almost automatically during the process. The alternative hypotheses are discussed in Section 5.

The case study research procedure is built on triangulation. Triangulation means that the object should be examined from multiple perspectives.

Case studies rely primarily on qualitative data, which is less precise than quantitative data. Hence, triangulation is needed. However, qualitative data sources are broader and richer, so single data units are more important if the data is managed correctly. [14] Triangulation can be divided into four different types [14, 15]:

- *Data Source Triangulation* - use multiple data sources
- *Observer Triangulation* - use multiple observers
- *Methodological Triangulation* - combine and use multiple data collection methods
- *Theory Triangulation* - use multiple viewpoints and create alternative theories

In this case study, these triangulation types are taken into account by having multiple interviews, having access to project specific information and the results are complemented by the customer and analytics data. The research was mainly monitored from an objective perspective. Nevertheless, I had been the main software engineer of the Solmu project which caused a bit challenge to objective monitoring.

2.2 Case selection

The cases for this study were selected from a project pool of a Gofore Oy software company. An internal questionnaire was created to map out what kind of technologies and service architectures projects were using. Based on the answers, three case projects were chosen which shared the same kind of development environments but had a different project size. Two of these selected case projects were provided by Finnish Institute of Occupational Health (FIOH) and one of them was part of Population Register Center's (*fi. Väestörekisterikeskus,* VRK) services.

The initial questionnaire included over 10 different microservice projects. However, I had worked as the main developer for the Solmu project. Hence, similar case projects were chosen to compare the decisions that were made in the Solmu project. Latu was selected to be the second case project because it had a very similar architecture and I had worked closely with Latu team members. After this selection, a larger project was selected to add contrast to the research. Thus, Kapa was selected and Kapa was also recommended

Table 2.1: The studied case projects

Case	Customer	Project size (man-days)	Team size	Interviews
Solmu	FIOH	200	4	1
Latu	FIOH	10000	6	3
Kapa	VRK	100000	13 x (5-12)	6

by one of my colleagues. Kapa had a more sophisticated software structure and a large testing team.

The case projects have many similarities: all case projects are less than three years old, public service oriented, implemented mainly by Gofore Oy, and have employed a microservice approach right from the beginning. In contrast, long-term software projects tend to have more testing interest involved because a developer cannot have a full understanding of the whole system. Additionally, public service oriented software projects tend to have clear specifications right from the beginning because projects have gone through a competitive tendering. Furthermore, the case projects have applied a microservice architecture right from the beginning. Dividing an existing system requires different methods: maintaining backward compatibility at the integration phase, implementing sufficient tests before integration phase, and so on.

2.3 Overview of case projects

The case projects have similarities in team structure, organizational requirements, and in a well-defined service structure. The projects are implemented in small agile teams, where people have specialized roles. For example, in the Solmu project, we had a scrum master, a developer, a user experience designer, and a DevOps architect. The case projects are summarized in Table 2.1.

2.3.1 Case Solmu

The goal of the Solmu project is to develop a self-assessment questionnaire, the Abilitator, for measuring the work ability and functional capacity of the participants. These participants are unemployed job seekers, who are motivated through government founded projects. The project results are gathered through the Abilitator questionnaire. The Solmu project is part of the European Social Fund (ESF) Priority 5 program that supports the employability and social inclusion of groups at risk of marginalization. Solmu is funded by ESF and carried out by Finnish Institute of Occupational Health (FIOH).

Therefore, Solmu aims to support social inclusion and employability of various hard-to-reach groups in Finland. FIOH is monitoring these projects through the Abilitator questionnaire and providing analytics from the data to participants and to common use. The Abilitator service has three types of users. It has FIOH administrators, program workers, and program participants. The service also includes three different deployment environments: development, acceptance testing, and production.

The project lasted about 5 months with 4 team members so it was quite small compared to the other case projects. The service was implemented using a microservice architecture because the service will be extended in the future.

2.3.2 Case Latu

The Latu project consists of two different services: Quality portal (Laatuportaali) and Zero accidents (Nolla tapaturmaa) -forum. The services share a few background services but provide separate front-end services. Both of the services are carried out by Finnish Institute of Occupational Health. This thesis concentrates on projects' software architecture and testing practices. Thus, the services are handled as one project, because the same developer team is responsible for both of the services. The developer team size has been between 4 and 7 team members.

Laatuportaali provides information, support, and tools about occupational health care. The project aims to improve occupational health care co-operation and to provide tools for comparing and improving the quality of occupational health care services in Finland.

Nollis aims to decrease work accidents by providing a program to companies which they can join. The program includes annual meetings, tools, and metrics which can be used to prevent work accidents at workplaces.

Both services are long-term software projects which aim to be functional over a decade. Thus, a microservice architectural style was selected to ensure the maintainability and future development of the services. The services do not have many concurrent users. Both of the services are more like form filling services that gather information and provide supportive feedback to the users.

2.3.3 Case Kapa

National Architecture for Digital Services (*fi. Kansallinen Palveluarkkitehtuuri*, KAPA) is a large continuous software project, which aims to provide a public service interface to services provided by the public sector. It facilitates information transfer between organizations and services. The program involves creating a national data exchange layer, the shared service views required by citizens, companies and authorities, a new national e-identification model and national solutions for organizations and individuals. The program aims to simplify and facilitate transactions with the authorities. It promotes openness in public administration and improves quality and security of the online services. So, basically, Kapa provides an interface to different national public services. Moreover, it also adds an access-point to the end-users as well as to private and public sectors.

The thesis concentrates on the data exchange layer because its architecture differs from the other case projects. The project encloses over 13 different software teams (see Table 2.1). The team size varies between 5 and 12 members, but the average team size is about 8 people. The data exchange layer includes four different teams, so the layer is built in close relationship with related teams.

2.4 Data collection

The data for this study was collected in Spring 2017. The data were collected through semi-structured interviews which were held at the customers' premises. The interviews were recorded, so the interviewer was able to have his or hers full focus on the current interview. Persons for the interview were selected based on the project's architect's recommendations. The architect knew the project, so they were able to point out different people from the project. It was important to select people with varying backgrounds to ensure data source triangulation of the case project. The selected team members had been in the projects from the beginning, so they were able to understand the reasons behind test changes during the project.

All except one interview were conducted face-to-face, which made note taking more difficult. Actually, the interviewer preferred to stay focused and leave note-taking for later. Two recording devices were used to ensure that recording works all the time. The planned duration for the interviews was a bit over an hour, but the interviewees were asked to reserve 1.5 hours for

Table 2.2: The interview pool by projects.

Project	Time used (min)	Role
Solmu	50	Scrum Master and Architect
Latu	75	Scrum Master and Architect
Latu	40	Software Developer
Latu	60	Project Manager
Kapa	90	Software Architect
Kapa	60	Test Manager
Kapa	60	Project Manager
Kapa	40	Regression Tester
Kapa	40	Regression Tester
Kapa	40	Manual Tester

the interviews, so the interviewer would not need to hurry. The interview duration decreased generally after the first interview from a case project because the interviewer did not need to use so much time to understand the background of the case project.

The target was to get at least three interviews from each case, but in the Solmu case, it was impossible because the project did not have enough technical team members to interview. Furthermore, I could not interview myself. Relevant interviews were chosen with the project architect's help. Mostly technical long time project members were interviewed because they had the broadest knowledge of the project's history. The interviews had also two different technical groups: testers and developers. Overall, I completed all interviews which included varying roles, such as developers, testers, architects, and project managers. The diversity of the interviewed roles is presented in Table 2.2.

The interviews started to repeat themselves after a few interviews. The overlapping answers were able to specify the problem cause. Sometimes, the interviewees had not thought about the testing process problems before those were brought up in the discussion.

Two different interview templates were used to guide the semi-structural interviews. The first interview template presented in appendix A.1 focuses on technology specific questions for the developer team. The second interview template presented in appendix A.2 is for software project managers. The second template is process and team oriented whereas the first one is more software architecture specific. The templates follow a top-down approach, where the questions started from a more general level and became more precise during the interview. For example, the interview started by explaining my thesis project and after that, the interview included questions about the interviewee's background. The interview had also a few team based questions about the project's history to see if there had been any big changes during the project.

The interviewees were asked about their individual testing practices and to compare their testing practices to the project's requirements. They were also encouraged to describe the system in their own words and to tell possible improvement ideas and challenges in the case project. Interview templates were inspected and tested with my colleague beforehand to ensure that the questions are relevant to the problem.

2.5 Data analysis

The interviews were performed in two batches. The first batch included the case projects' software architects. The second interview batch included other team members. After the first interview batch, the interview notes were grouped from the recordings. The interview direction was checked to ensure that the interviews are leading to the right direction. The results were analyzed by organizing the answers into clusters. These clusters had usually some common properties and the clusters were named in order to be able to refer to them later.

At the next interview batch, mainly software developers and testers were interviewed. The answers started to repeat them self after a few interviews. When the overlapping answers were detected the interview were focused more on the details. The first batch interviews were able to clarify the fuzzy concepts that architects introduced on a high level. The developers' comments were closer to the actual implementation, which allowed to see language specific problems. On the other hand, managers pointed out the responsibility and scheduling problems.

Chapter 3

Related Work

This section presents the background of the study through a literature review. The section begins by explaining what microservices are in Section 3.1. Section 3.2 explains the microservice development environment, which includes basics of agile software development and microservice deployment. After this, 3.3 goes through testing strategies and explains the terms in a microservice context. Next, Section 3.4 describes different testing practices and connects those with a software maturity model. Finally, Section 3.5 concludes the literature review and summarizes the key findings from the literature.

3.1 Microservices Architecture Structure

There is no formal definition of microservices, but microservices tend to have some common characteristics [1]. Mainly, microservice architecture is a method of developing applications consisting of independently deployable, small, and modular services [2]. These services communicate through a well-defined, lightweight protocol to serve a business goal.

The application's requirements define the communication protocols, but usually, developers end up with using HTTP/REST with JSON or Protobuf [16]. A reason to this is that Representational State Transfer (REST) is easy-to-use and extremely well-supported [16]. However, services that require more calculating power, may end up with using Remote Procedure Calls (RPC) to provide tighter coupling between the client and server [2]. Anyway, in this thesis, we concentrate on HTTP/REST, because it is used in all the case projects and it has become clearly the most used protocol in microservices [1].

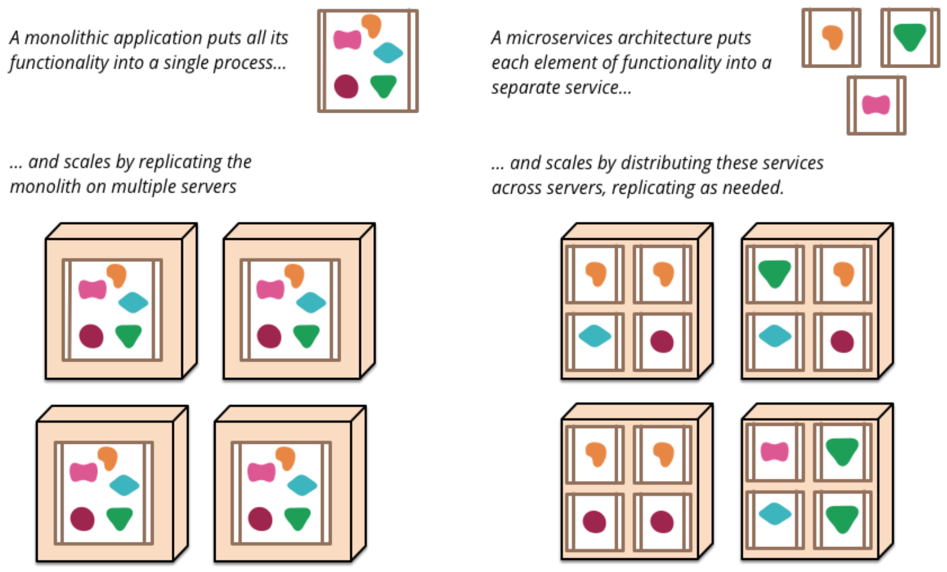
A monolithic application puts all its functionality into a single process...
... and scales by replicating the monolith on multiple servers
A microservices architecture puts each element of functionality into a separate service...
... and scales by distributing these services across servers, replicating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

Figure 3.1: Structural differences between monolith and microservice architecture [1].

In order to understand microservices, it is good to compare to its opposite; the monolithic architectural style (see Figure 3.1). A monolith application is always built as a single unit. There all modules are inside one application and built as one. This brings up a problem; small software updates require building and deploying an entirely new version of the whole application. Moreover, if you need to scale a component of an application, you may need to scale the whole application instead, because the components are usually really tightly related. [16]

Single process applications, monoliths, are usually built from small modules, but the connections between modules are tight and hard to split into separate processes. This restricts code reusability and extendability. For example, changing the programming language of a monolith is almost impossible. In web software development, coding languages and frameworks out-date rapidly, which has created a demand for a more flexible architectural style. Thus, web services have started to split their monoliths into smaller pieces - towards microservices. [1]

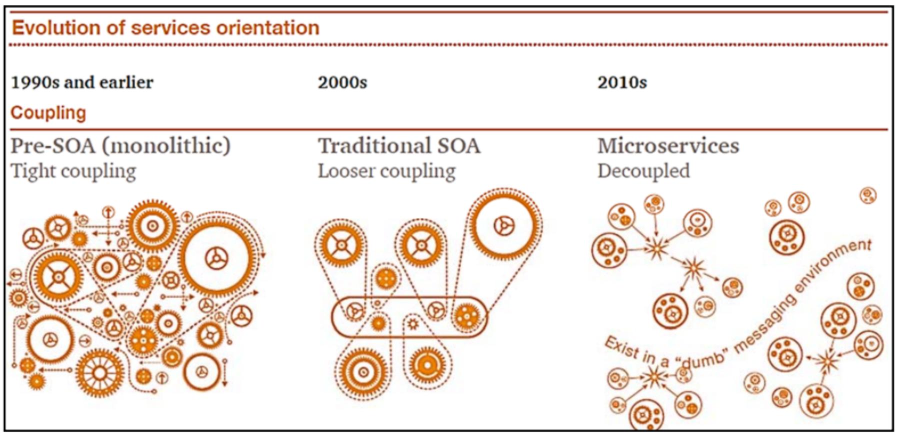

Figure 3.2: The evolution of service oriented software architectures [6].

3.1.1 Relation to Service-Orientated Architecture

Service-Orientated Architecture (SOA) became popular at the beginning of this century. It has many similarities to microservices. It is said that microservices are a modern ideal implementation of SOA [16]. The main difference between SOA and microservices is that SOA is a broader framework which includes also other implementations [16].

The best known SOA implementations are old-fashioned *Web Services*. Web Services is a middleware interoperability standard which provides a message bus and remote procedure calls between services [17]. Web Services commonly use Simple Object Access Protocol (SOAP) to transfer data over Hypertext Transfer Protocol (HTTP). The web services message bus is generally referred to as Enterprise Service Bus (ESB) because it facilitates the transfer of data among services. [18] Web services have been criticized for having a vendor middleware which might restrict service development. Furthermore, many problems which are associated with SOA are actually problems of previous SOA implementations. For example, problems like stiff communication protocols (e.g., SOAP), vendor middleware, and lack of guidance of service structuring have decreased SOA's reputation. [2, sec. 1]

Sam Newman [2, sec. 1] defines SOA in the following way: *"SOA is a design approach where multiple services collaborate to provide some end set of capabilities"*. Service here means a completely separate system process, where communication with other services goes through a network layer instead of straight method calls. Thus, these connections are described as loose connections. [16]

Table 3.1: The main differences between traditional SOA and microservice architecture. Adapted from an article [6].

Criteria	**Traditional SOA**	**Microservices**
Messaging style	Smart, but dependency-laden ESB	Dumb, fast messaging
Programming style	Imperative model	Reactive actor programming model that echoes agent-based systems
State	Stateful	Stateless
Messaging Type	Synchronous: wait to connect	Asynchronous: publish and subscribe
Databases	Large relational databases	NoSQL or micro-SQL databases blended with conventional databases

As mentioned before, SOA and microservices are trying to split a monolith into smaller maintainable pieces to promote the reusability of software. Figure 3.2 visualizes how service-oriented architecture has been evolved recently. The main changes between traditional SOA and microservices are in messaging type, programming style, application state, and database handling (see Table 3.1). However, more recent SOA implementations include also features from microservice architecture. [6]

According to Newman [2, sec. 1], SOA is a great idea, but it does not have solid guidance on how to achieve re-usability of software. Consequently, SOA fits well in situations where the developer team is homogeneous and it has already selected a service structure that will guarantee a certain level of performance. In other situations, microservices, as a more democratic framework allow a more agile framework to the teams to implement their parts without unnecessary restrictions. In many cases, microservice architecture can be used at least as an exploratory solution. [6]

3.1.2 Forerunners of Microservices

Many well-known services are using microservice architecture, including Netflix, eBay, Amazon, Twitter, PayPal [1]. Netflix has been a great example, how it is possible to convert a monolithic architecture into microservices [16]. Their online service receives more than one billion requests every day to its streaming-video API. Each API call is creating additional back-end calls in order to complete its functionality. Microservices allow easy load balancing because the microservices do not have many external relations. Thus, microservices can be run on separate machines and there can even be multiple entities of one microservice. This improves application's scalability and makes online services likes Netflix possible. [16]

Amazon is another well-known pioneer in the field [2, sec. 10]. They had a more team based approach to develop online services. They noticed that small teams can work faster than large teams. Thus, they designed the nowadays famous two-pizza teams where no team should be so big that it could not be fed by two pizzas. These small teams handled their feature's whole life-cycle from requirements until the maintenance phase. Amazon designed Amazon Web Services (AWS) for supporting the delivery process of small teams. Nowadays, AWS is used widely with microservices to provide good tooling to allow cross-functional teams to be self-sufficient [19].

3.1.3 Key Benefits

The main advantages of using microservices are according to Newman the following [2, section 1]:

- **Technology heterogeneity.** A microservice does not restrict the implementation technology of the service. Thus, the developer team is solely responsible for choosing the right technology to solve the problem. This includes choosing a programming language, databases, and related libraries. However, the team should have a common guidance to coding style to improve its services' maintainability. The microservices should use mainly the same communication protocols between services. If the communication protocols vary a lot, each module requires different middleware to convert the messages between protocols. Using multiple communication protocols or different technologies adds unnecessary complexity to the system and requires more expertise from the team. The team should try to keep the system as simple as possible to ensure its maintainability.

- **Resilience.** Building a system composed of microservices can increase the resilience of the whole system by enabling better isolation of failures. However, this requires that the service has taken fault-tolerance into account. A system comprising of microservices might work mostly, even if one service would be off-line compared to a monolithic architecture where the whole system crashes if its component gets into a fatal situation. On the other hand, a crashed microservice may cause cascading failures through the whole system if it is not well isolated. A cascading failure crashes or jams the whole microservice network. To prevent cascading errors, microservices use timeouts, bulkheads and circuit breakers to isolate failure situations. Additionally, microservices, as any other distributed system, requires active monitoring to see the status of all the services. For example, health monitoring and different types of application metrics are used to monitor the services. By these metrics, the team is able to ensure that all services are working correctly. The team can also predict upcoming problems by monitoring, for example, the load usage of the services.

- **Scaling.** Enterprise applications have to be ready for scaling. With a monolith architecture, this is expensive because the whole system needs to scaled. With microservices, scaling can be done in smaller pieces. The server can deploy multiple instances of microservices that have most of the load. For example, Amazon Web Services automatically spawns more instances of a microservice if the microservice is under heavy load. This kind of scaling allows you to run the whole application on less powerful hardware. See Figure 3.3 for how monolithic and microservice architectures differ. Each box in Figure 3.3 can be multiplied into many instances (except shared databases, which require the replication of the whole application like a monolithic architecture).
- **Ease of Deployment.** Ease of deployment actually enables the benefits of microservices. Microservices are commonly built in agile teams, which have the whole responsibility of the new functionality. This means that the team is responsible for designing, implementing, testing, and maintaining the implemented functionality. Thus, the team needs an easy way to deploy their own functionality. The deployment should be easy and fast. Faster releases allow for better tracking of the software's direction based on agile software development principles.
- **Organizational Alignment.** Conway's Law [20] states the following: *"Any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization's communication structure."* Basically, this means that communication problems

increase when a team is too large or distributed. Microservices are embracing Conway's Law by giving the whole ownership of the functionality to small agile teams. The team is in charge of developing, deploying, testing, operating, and possible restructuring of the services under their control. Hence, the team gets a suitable challenge where the team has a freedom to find a solution. A suitable challenge improves a team's motivation and enhances their working capability.

- **Composability.** Microservices allow us to divide platform specific builds into separate services. This way the application supports easily multiple platforms because a microservice can reuse main components which are not platform specific. The application supports also building the software by using different versions of the microservices to create specific releases. Therefore, the product can be composed of a subset of the microservices rather than duplicating service functionalities. On monolithic systems, reusability of the software is not guaranteed and maintaining platform specific releases is always a challenge.
- **Optimizing for Replaceability.** Microservices are like modular components; they are independently upgradeable and replaceable. Thus, there are not many restrictions to replace the whole service with newer technology. This allows keeping the services fresh and reactive to market and user requirements. For example, rewriting the service is a tempting option, when major changes are made to an old service. This happened for example in one of the research cases.

3.1.4 Base Requirements

Microservice also brings new software environment challenges to the developers, which are managing distributed systems: building support for continuous deployment, restructuring test automation, and setting up service monitoring [2]. Without environment support, a microservice loses all its benefits [16]. Microservices depend heavily on agile software development ideology, where the development feedback cycle should be short. This is not achievable if services' deployment is too slow or difficult [8].

Building a distributed system instead of a monolith can double the effort [16]. This extra work comes from implementing the communication interfaces between services, configuring multiple developments and test environments, and integrating everything into a continuous build delivery process

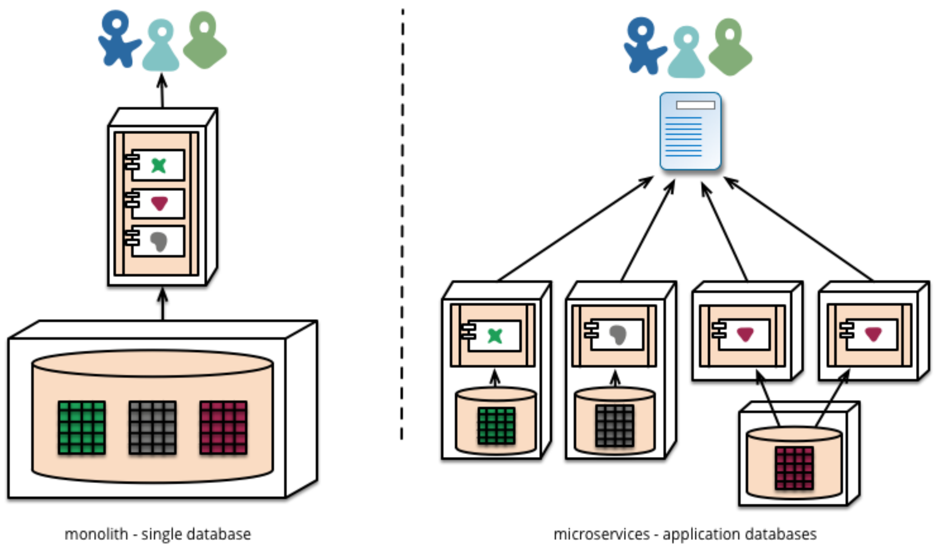
monolith - single database
microservices - application databases

Figure 3.3: The difference between a monolithic and a microservice architecture [1].

[23]. Thus, microservices includes investment into infrastructure before the project actually starts.

According to Newman [2], microservices are not solution to every online service. He says that splitting a monolith into microservices is a challenging task because usually a monolithic architecture is not designed to be split apart. There are different toolsets to help with these challenges, but a developer with a monolithic architecture background requires some time to learn these new tools. Learning to use these tools is necessary for a microservice environment because building own unique solutions is not a sustainable solution. Software development tools help to minimize monitoring, testing, and deployment delays. The team's toolset depends on the project, organization, and team's preferences. [2]

Fowler [21] brings up the following prerequisites for microservices:

- **Rapid provisioning.** The deployment process may need to fire up a new server, so the server spawning should be automated. For example, this can be achieved by using a cloud provider or using another existing provisioning system, as long as rapid deployment is achieved.

- **Service monitoring.** Failures are inevitable and noticing the failures will be more difficult in distributed systems. The service might get jammed or just lose its state in relation with the other services. Thus, monitoring is essential with microservices. The services should be easily accessible and real-time monitored. Basically, monitoring includes logging all events and parsing those into one value adding stream. For example, a minimal log implementation could record all HTTP requests from microservices into a common file [2]. A more sophisticated monitoring solution would inform about disruptions in testing and production environments [2]. The faults may be impossible to repeat without proper logging from the application side.

- **Rapid application deployment.** This involves the creation of a delivery pipeline, where a developer can easily promote selected builds of the services into production or test environments. The delivery pipeline should finish the provisioning in hours to be feasible [22].

3.1.5 When to Adopt Microservices

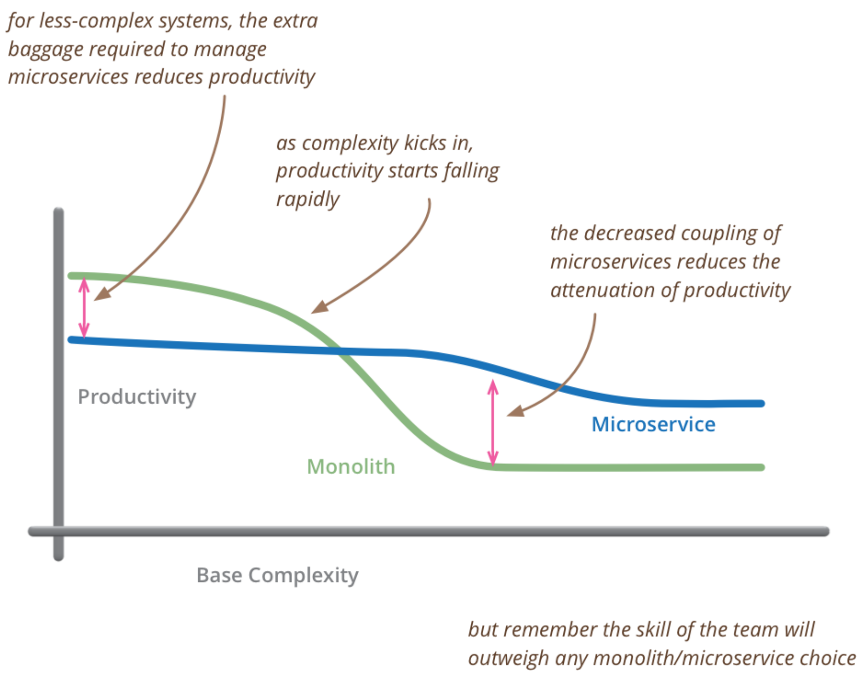

Figure 3.4: Productivity over project complexity [23].

A base rule according to Newman [23] is the following: *”You should not even consider microservices unless you have a system that’s too complex to*

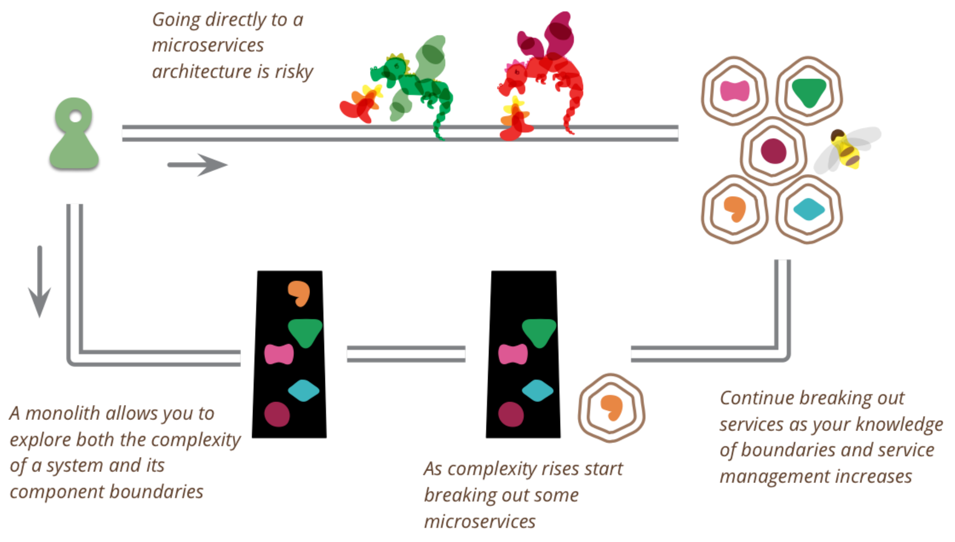
Going directly to a microservices architecture is risky
A monolith allows you to explore both the complexity of a system and its component boundaries
As complexity rises start breaking out some microservices
Continue breaking out services as your knowledge of boundaries and service management increases

Figure 3.5: Software architecture path by Martin Fowler [24].

manage as a monolith.”. Figure 3.4 shows the correlation between productivity and base complexity in monolithic and microservice architecture approaches. Therefore, many people favor a monolithic architecture, because microservice architecture adds unnecessary complexity at the beginning of the project [24]. Additionally, according to Fowler [25] and Newman [26], most of the projects should be built as monoliths, because the complexity of those projects will not cross the critical point. Thus, they recommend starting with a monolithic architecture style (see Figure 3.5), unless you are sure that the complexity of the software will be increasing and the customer is ready to invest into the future development.

Fowler [25] also argues that organizational challenges have driven teams towards microservices. For example, if the team size is too large, it becomes inefficient and loses the focus of the whole system. He points out that the main reason for moving towards microservices is that the monolith may grow too big to be modifiable and become too fragile to be deployable.

I see that Newman’s rule applies well to the situation and I agree with Fowler that the software should be divided into pieces based on team needs. Fowler is a significant microservice supporter and if he says that it is too risky to go straight to microservices I totally understand that.

Tilkov [26] disagrees with the monolithic first approach (see Figure 3.5). He says that you should reserve more time to architecture design, and delay implementation in order to be able to structure the microservices right from the start. He claims that if you start with a monolithic architecture style, the components will be extremely tightly coupled with each other, and these components are almost impossible to restructure afterward. He argues that if you are able to build a well-structured monolith, you probably do not need microservices in the first place.

Tikov has a different viewpoint, but I think that he wants to emphasize that a monolithic application does not automatically slip into microservices. Even if you are building a monolith at the start, the project should have a determined software architect who would not allow taking shortcuts between the modules. Thus, I see that people selection affects a lot to the path selection. With motivated and quality oriented developers monolithic first approach would work, but if developers are not so self-quality oriented the approach would not work because the monolithic software would end up to have too many tight relations between the modules.

On the other hand, some of the problems ascribed to monoliths are not influenced by the architectural choice. For example, a developer can do continuous delivery also when using a monolithic architecture, but many people tend to associate that with microservices. For example, Facebook has used a cookie-cutter deployment approach to achieve continuous delivery with a monolithic architecture style [25]. Cookie-cutter deployment is used when the performance of a server is not enough to deploy the software. The deployment is divided into multiple services which perform the deployment process concurrently. [27]

3.2 Microservice Environment

Microservice testing is tightly related to agile software development practices [28]. Thus, it is important to understand the deployment process of microservices. First, Section 3.2.1 explains what the lean principle is and then goes through Scrum and Kanban software development strategies. Next, Section 3.2.2 takes look into continuous integration (CI). Section 3.2.3 presents the continuous delivery (CD) process model. After these basic agile software deployment practices, Section 3.2.4 applies these deploying processes in a microservice context.

3.2.1 Agile Software Development

Microservices are usually implemented in small agile software development teams [2]. All case projects were using an agile software development process. Thus, it is important to understand the basic principles behind agile software development. Agile software development has many patterns which can be explained through the principles of lean software development [29]:

1. *Eliminate Waste*
2. *Build Quality In*
3. *Create Knowledge*
4. *Defer Commitment*
5. *Deliver Fast*
6. *Respect People*
7. *Optimize the Whole*

These principles optimize software development by decreasing waste time, offering shorter delivery times and building trust within the team [29]. Our case projects were using Scrum and Kanban agile software development guidelines to achieve effective software development processes.

Scrum [30] has specified steps for how product owner's user stories ends up in the product (see Figure 3.6). The product owner starts the process by presenting the user stories. The user stories are split into smaller tasks and the team commits to doing some of those during the next sprint. The selected tasks are moved to the sprint backlog. During the 1-4 week long sprint, the team has short (max. 15 minutes long) daily meetings where each team member answers three questions: what they did yesterday, what they will do today, and are there any problems in the way. After the sprint, the potential product increment is reviewed and the team keeps a retrospective to examine possible process improvements.

Kanban is quite similar to Scrum with a few differences [31]: Kanban does not include specific time boxes or iterations. The Kanban process is continuously iterative. The Kanban process limits the work in process (WIP) in each work category (see Figure 3.7). So, the work is limited by the tasks and the efficiency of the process is measured by the lead time. Lead time means the average time to complete one task. The process is

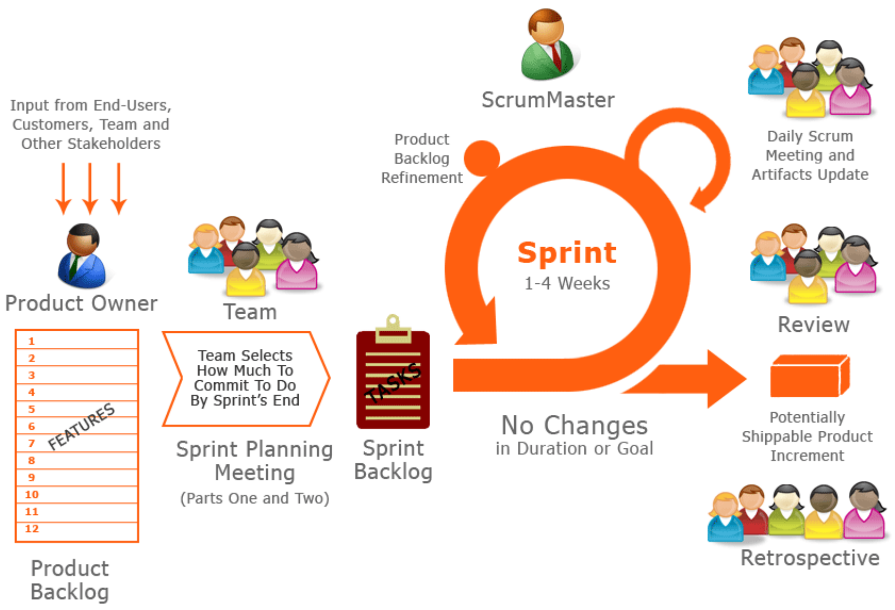
Input from End-Users, Customers, Team and Other Stakeholders
Product Owner
1
2
3
4
5
6
7
8
9
10
11
12
FEATURES
Product Backlog
Team
Team Selects How Much To Commit To Do By Sprint's End
Sprint Planning Meeting
(Parts One and Two)
TASKS
Sprint Backlog
ScrumMaster
Product Backlog Refinement
Sprint
1-4 Weeks
No Changes
in Duration or Goal
Daily Scrum Meeting and Artifacts Update
Review
Potentially Shippable Product Increment
Retrospective

Figure 3.6: Scrum software development process [30].

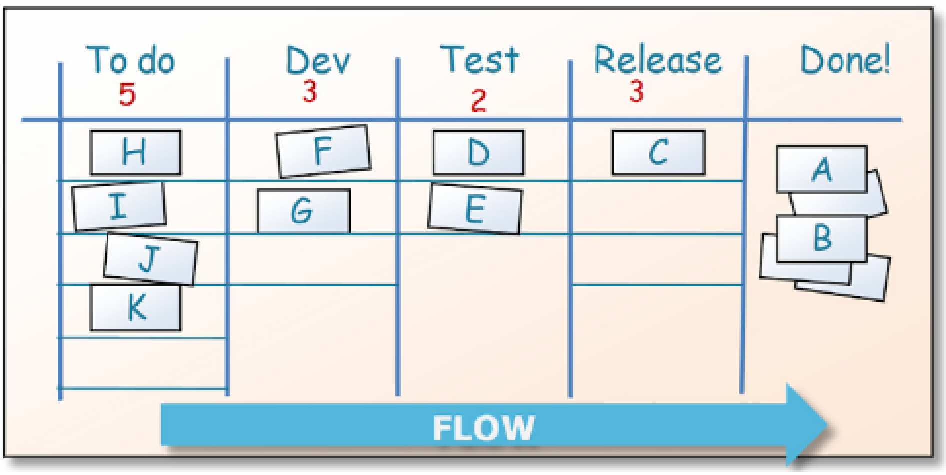

Figure 3.7: Kanban software development process [31, sec. 1].

optimized by decreasing the lead time as much as possible without sacrificing the predictability of the tasks.

However, agile software development leaves a lot of freedom, flexibility, and responsibility to the team [31]. Therefore, the teams should have broad professional competence. For example, Scrum and Kanban models are usually used in cross-functional development teams. The team shares their knowledge and aims to be self-organizing, so the team itself notifies and reacts to upcoming challenges.

3.2.2 Continuous Integration

Continuous integration (CI) is a practice to keep the shared mainline up-to-date [5]. This software development practice is used to minimize the risk of software integration. In agile software development, new code updates are committed constantly to the mainline. Continuous integration [5] allows fast integrations of the updates and prevents forming a pile of code changes. Continuous integration relies on automated tests that are run every time before the code submission is accepted. These tests ensure that the added functionality does not break any existing functionalities.

Usually, a team uses a dedicated continuous integration server, which handles the build test running and automates the process [2, sec. 6]. There are multiple frameworks supporting the modern CI methods, for example, Jenkins, TravisCI, and TeamCity offer a broad CI tool set [22].

3.2.3 Continuous Delivery

Continuous delivery (CD) is an extension of continuous integration [32]. It includes the software release process, where CI only takes along the development process. Continuous delivery [32] can be seen through a build pipeline, which declares the code submission's route to release. Figure 3.8 shows that the first testing steps are fast unit tests and the last steps are manually approved user acceptance testing (UAT) and release publication. Furthermore, continuous delivery considers every check-in as a release candidate, because all code submissions may end up released if they go through the whole pipeline. Each step has to be completed successfully in order to move on to the next stage. Nevertheless, Figure 3.8 shows that the user has the final word because even the automated tests fails the user can accept the build to UAT environment.

Benefits of this pipeline are that it helps to monitor the quality of the

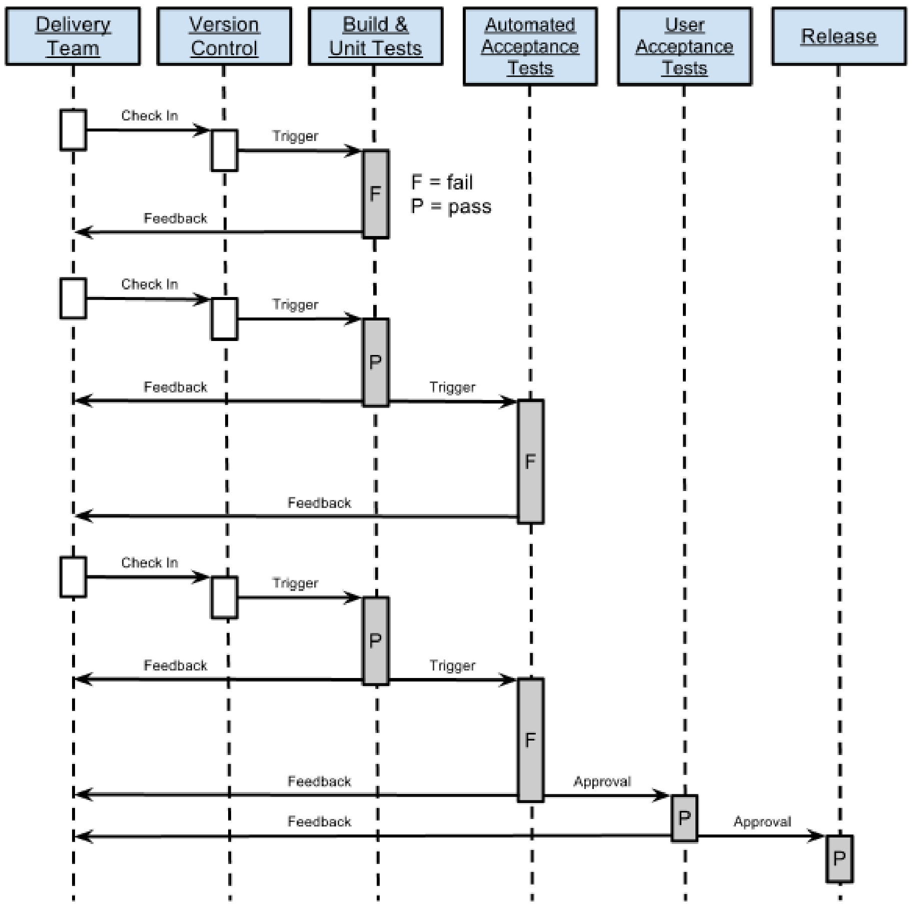

Figure 3.8: Deployment pipeline process visualized. Adapted from a book [32, sec. 5].

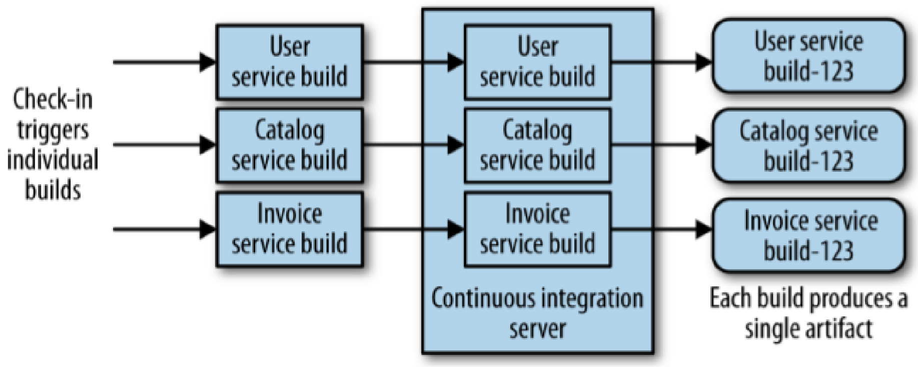
Check-in triggers individual builds
User service build
Catalog service build
Invoice service build
User service build
Catalog service build
Invoice service build
Continuous integration server
User service build-123
Catalog service build-123
Invoice service build-123
Each build produces a single artifact

Figure 3.9: Continuous integration building in microservice context. Each service has their own source code repository and builds are done independently. Adapted from Newman [2, sec. 6].

software, reduce the time taken between releases and allows constant feedback from the code submission's readiness. Disadvantages are usually the invested time for building the whole pipeline because the whole process can also be achieved by running separate scripts. [2, sec. 6]

3.2.4 Deploying microservices

As mentioned in Section 3.1.4 microservices requires a continuous delivery process in order to benefit from the architecture. Anyway, the build pipeline that a microservice requires is larger than for a normal monolithic application's build pipeline, because every microservice has to have their own build process (see Figure 3.9). According to Sam Newman [2, sec. 6] building or releasing multiple microservices at once can cause a lock-step release pattern, where the microservices have hidden couplings between each other. Thus, he recommends that each microservice has their own repository, pipeline and build provisioning. He says that separating the microservice's deployment processes decreases the risk of undiscovered dependencies.

Running multiple build processes on multiple machines requires a lot of calculating power and may be expensive to maintain. Thus, continuous integration servers are usually using virtualized containers to be able to build multiple microservices on the same machine. Docker [33] is a lightweight platform that allows an easy way of managing and deploying applications. It allows configuring everything in advance so that reproducible server configurations are made possible. [22]

3.3 Testing

The previous sections presented the microservice architecture and the development environment of microservices. Section 3.3.1 explains testing methodologies. Next, Section 3.3.2 presents the testing scope. Finally, Section 3.3.3 brings up testing differences in a microservice environment and shows special testing properties of microservices.

3.3.1 Types of testing

Sam Newman [2] emphasizes the purpose of testing. He says that testing is always an additional task for a developer to ensure that the service meets its requirements. Figure 3.10 classifies testing methods by team impact. The tests can be business or technology facing as well as critical or supportive [34]. These categories help us to understand to who we are writing these tests for. For example, engineers usually start from the Q2 tests, which are supportive and helps implementation of a certain user story.

Tests can be roughly categorized into development and maintenance phase tests [34]. Q3 and Q4 test categories require that there is deployable

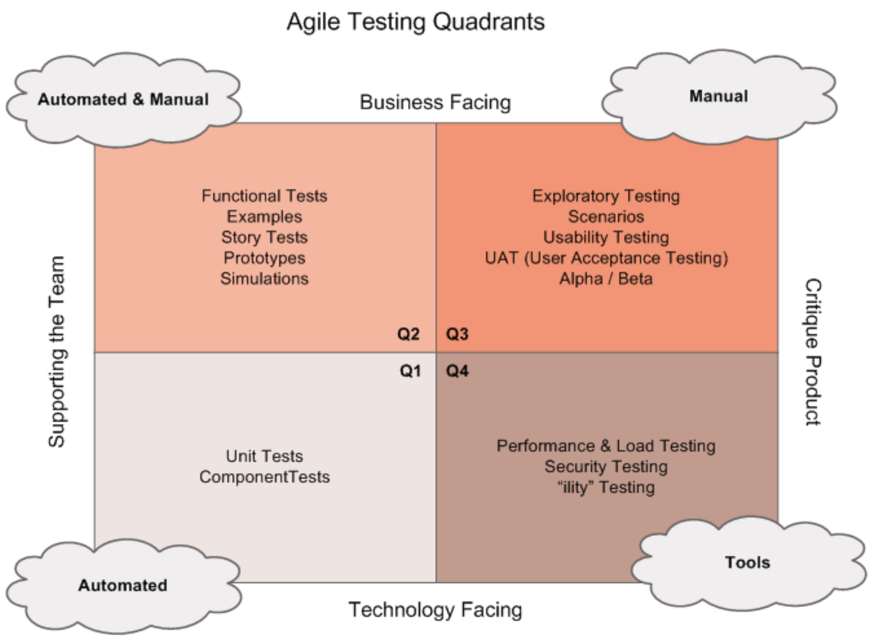

Figure 3.10: Marick‘s testing Quadrants (The markings Q1, Q2, Q3, and Q4 are just used for referring to the categories) [34].

software. Therefore, these tests are usually completed after the functionality is written. Anyway, in some projects performance, security or even usability might be key elements of the service. Hence, the team may prioritize Q3 or Q4 tests over Q1 and Q2 tests, because the pilot test results may change the whole direction of the project.

This thesis mainly concentrates on automated testing. Figure 3.10 shows that automated tests are team supportive, which means that developers create these test in order to speed up the localization of malfunctions. The locating speed depends on the test scope, which is explained in Section 3.3.2. Furthermore, testing can also be divided by methodologies to black-box and white-box testing [35]:

- **Black-box testing.** Black-box testing means that the system under test (SUT) is unknown. The internal logic is hidden and the user can only monitor the system's inputs and outputs. This kind of system requires massive amounts of input tests to expose potential bugs. However, the user cannot determine the expected functionality by creating random inputs, if the system is a total black-box. This makes validating the results more difficult. To be able to validate the results, black-box testing is commonly combined with the white-box testing approach. In white-box testing, the user knows the expected results. Thus, black-box testing is mainly used as a supportive testing method to increase the coverage and probability of finding bugs [35].

- **White-box testing.** White-box testing examines the system's internal behavior. It ensures that all parts of the code are working correctly, but it does not take collaboration with other systems into account [35]. Sometimes white-box tests are described with test code coverage numbers. On the other hand, even if test code coverage would reach 100%, the software may still include bugs because test coverage is only testing a few of the possible paths of the software.

These methods can be applied to different scope levels of a system, which are described in the following Section 3.3.2.

3.3.2 Test scope

Test scope can be explained with the test pyramid concept, which has been developed by Mike Cohn [37]. Figure 3.11 shows the concept. It divides the test area into three main categories, which are the following [2, sec. 7]:

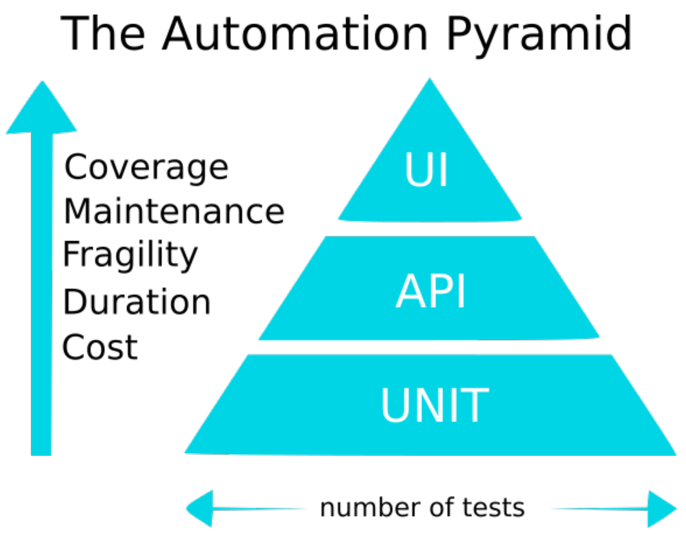

Figure 3.11: Mike Cohn's testing pyramid. The pyramid describes test diversity in traditional software development. Adapted from [36].

- **Unit tests.** These typically test a single function, class or method. Furthermore, unit tests are fast; a modern hardware can run thousands of unit tests in a minute. Thus, unit tests are recommended, because tests are easy to write and fast to perform. For example, in test-driven design (TDD) a developer writes unit tests for every single functionality before implementing the features. Writing tests first helps and speeds up development. Nevertheless, if the requirements are not clear, writing the test cases is a waste of time. Moreover, rapid testing speeds up development process [37]. Additionally, the scope of unit tests is narrow, which allows fast localization of errors. Therefore, locating an error is faster than for broader tests [2].

- **Service tests.** These tests are designed to interact with services directly. Service tests bypass the user interface (UI) in order to make testing easier. Service tests isolate upcoming problems inside one service. The scope of service tests is then broader than for unit tests. Broader testing scope increases the test time, but also improves the test code coverage, which brings more value to the developer.

  Service testing is sometimes called application interface (API) testing. Services include commonly database or other external connections. This brings a problem with how to fake the external relations of the service. One option is to face the database or external relations by using test doubles (see Section 3.3.3). Another common option is to duplicate the database or use an external sandbox API. Usually, external services provide some sandbox API for testing.

- **End-to-end (E2E) tests.** These tests are usually driven through a browser's graphical user interface (GUI), which makes the testing a lot slower and more fragile to changes. These tests cover the main functionality of the service, but when they fail, it may be difficult to locate the problem without additional smaller scope tests.

Overall, all testing ranges are required, but the automation level of testing is dependent on the project's architecture and developer's vision. Outside of the testing triangle is exploratory testing (ET) [2, sec. 7]. In exploratory testing, a developer is manually finding ways to break the system. Exploratory testing is looking for an answer to the question: *"What is the best test I can perform, right now?"*. Thus, it is a powerful tool for providing rapid feedback from a new product or feature. It is also used for initial performance testing, security testing, product analysis and prototyping scripts. For example, security risks are usually discovered by using ET strategies [38]. ET includes everything that is not practical to automate. Nevertheless, developers tend to forget that even exploratory testing has strategies. A test should be implemented in such a way that it gives maximal information for the further process. Furthermore, ET does not have ordinary automated test restrictions, like test maintenance or suitability with other tests. [39]

3.3.3 Testing Microservices

This section describes software testing in a microservice context. First, the section defines testing terminology. This specifies service testing in a microservice context and includes test doubles, which declare different ways to face the service responses. Finally, the sections bring up the best testing practices for microservices.

Testing Terminology with Microservices

Microservices collaborate with other services, which creates a need for test dependency management [40]. Thus, microservices split the traditional service testing category into contract and integration testing (see Figure 3.12).

Figure 3.12 shows that contract testing happens via queries to an API. Testing the API access points verifies each service's calls independently. Con-

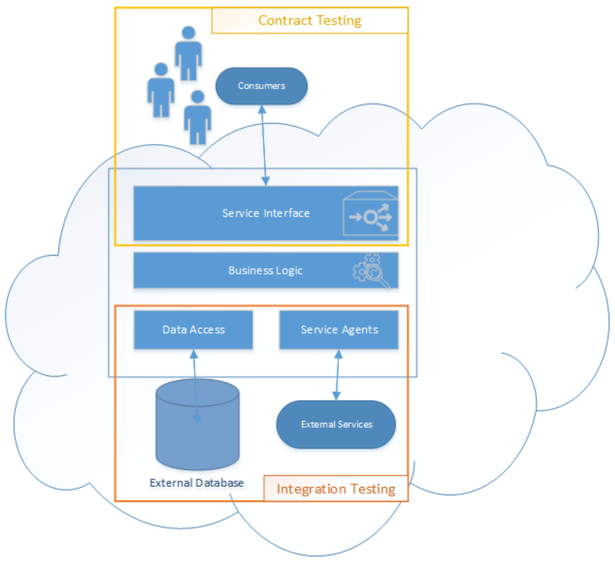

Figure 3.12: The difference between contract and integration tests. Adapted from [41].

tract testing is imagining a service as a black-box, so a tester does not need to know about the inner implementation. Server dependencies should be implemented as stubs in order to restrict interaction with other services. Moreover, stubbing other services decreases unnecessarily complex behavior, which could be caused by external services. [41]

Consumer-contract testing is treating each service as a "contract", for example: a contract-test can send a JSON request to a server and validate the response from the server API. The test ensures that the service gives the right answer from the API at all times. This kind of testing allows building more resilient services and the consumers do not need to change their interface. [41]

There are a few popular mocking frameworks to face responses from external services in order to achieve isolation on selected level. Those include for example Mockito for Java and JustMock for .NET. Furthermore, for consumer-driven contract testing Pact, Pacto, and Janus are well known testing frameworks. These frameworks allow testing services independently by isolating the service from other services and therefore verifying the service's contract's behavior [42].

Integration testing is done after contract testing because it assumes that the service's independent behavior is valid. Moreover, this is an important part of microservice testing, because microservices depend on cross-service communication. Integration testing should include error and success cases. It ensures that the service works with its dependencies as expected. [43]

End-to-End (E2E) testing ensures that there are no errors in the main workflows [44]. E2E tests work similarly to in monolithic architecture, but multiple services just make them more fragile, which increases the flakiness of the E2E tests. Based on Cohn's testing pyramid, tests should always be implemented on the lowest level of the pyramid if possible. Thus, there should not be a large need for E2E testing.

In a microservice context, unit tests can be split into sociable and solitary categories [42]. Sociable unit testing is a black-box testing method, that focuses on testing the modules by observing their state changes. Solitary unit testing concentrates on the interactions between an object and its dependencies.

Figure 3.12 clarifies the categories of unit testing by providing an example of a microservice. Unit tests can be included inside the service interface, business logic, data access and service agents. The services which are not highly state-based should concentrate on solitary unit testing and the rest on sociable unit testing. For example, business logic which is highly state-based requires sociable unit testing, but the other modules can be tested with the solitary unit testing style. Furthermore, the size of the unit tests is not strictly restricted, but typically unit tests are written at the class level to provide better isolation of an error. Moreover, unit test dependencies are commonly replaced by using test doubles. [42].

Test Doubles

Martin Fowler [45] lists different kinds of test doubles which support defining different methods for facing with external relation. He divides test doubles into five different categories, which are the following:

- **Dummy** objects are given as parameters but never used. These are used for example to fill some mandatory field of the request.
- **Fake** objects have a working implementation, but they include shortcuts. These shortcuts include different security risks, so fakes should never end up in production.
- **Stubs** have specified answers to test requests and do not react to other inputs.

- **Spies** have specified answers to test requests, but they also record history data about the calls. Thus, they could be described as stubs that record some information. For example, a mail service could work like this. It delivers messages forward, but at the same time, it internally calculates how many messages the service has processed.

- **Mocks** monitor the requests and have pre-programmed expectations which they compare to the input calls. If a call does not match, mocks throws an exception about unspecified behavior. Mocks should also include a relevant error message for the developer.

Microservice Testing Challenges

Microservices come with increased complexity of a distributed system and its environment, which makes testing more challenging. This means that cross-service testing needs more effort and may be unpractical to implement or maintain.

Hughes [46] argues that microservices should have fewer tests than monolithic applications because splitting the service into microservices has already simplified the testing surface of the services. Thus, he emphasizes the need to pay attention to value adding testing. This sets a challenge that a developer should know what to test instead of testing every typical use case. To know what to test, developers have to maintain an overall picture of the service [2]. Luckily, microservices are developed in agile cross-functional teams, so I think that forming the overall picture is easier than in a waterfall process.

Brown [47] tackles value adding testing by redefining the test pyramid for microservices (see Figure 3.13). He argues that class level testing is not as value adding as service testing in a microservice context. He adds, that the test pyramid balance is not a straight truth. A developer should understand what the most value adding tests in the system are. Hence, based on Brown's [47] experience the test pyramid will be more balanced between service and class tests in a microservice context.

Most of the microservice tests concentrate on service testing, which leaves the overall testing more open [2]. For example, Johansson [22] shows in his research that developers analyze the overall quality of the software as the biggest risk in microservice quality assurance. End-to-end testing includes problems like flakiness, maintenance responsibility, slowness, and deployment problems [2, sec. 7].

In some cases, instead of the comprehensive amount of end-to-end tests, microservices are using a highly developed release process as a testing tool. A new release candidate is running beside the production instance, so that the release candidate receives all synthetic transactions from the live environment. Synthetic transactions is an active monitoring method which contains pre-scripted patterns that simulate a real end-user behavior through the website. Active monitoring is predicting the future network states by recording the current log and processing it with algorithms. If the release candidate has no major concerns, the production instance can be replaced with the release candidate after a testing period. [2] However, this kind of testing requires semantic monitoring which includes investment into the release process, and monitoring tools, such as Graphite, elasticsearch, Riemann or Suro [22].

An alternative option is to use consumer-driven contract (CDC) testing, which simulates the end-users inputs by sending custom made network packages to the system. CDCs cover the core functionality of the service, but the results do not test the end-user service usability. These checks are mainly used as a sanity check for testing that a new update does not break any existing core functionalities. CDCs are commonly used with regression testing because CDCs are faster to perform than typical E2E tests. Nevertheless, E2E and CDCs only cover the main use cases of the application, which means that there are still many untested routes to test.

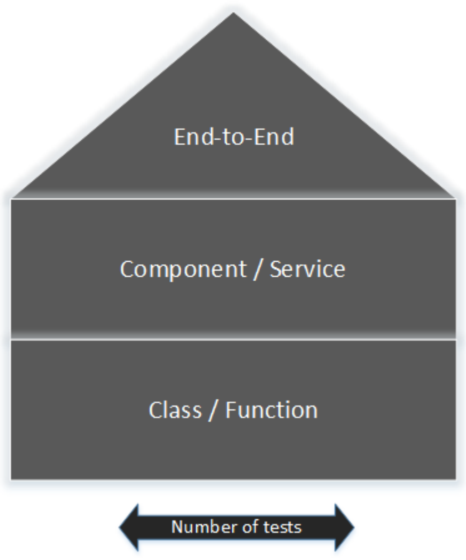

Figure 3.13: Microservice testing pyramid according to Brown. Adapted from an [47].

3.4 Software Testing Practices

Software testing practices focus on finding processes to improve the overall quality of the software. Traditional technical quality metrics include only software correctness, completeness, and security. Software testing practices are taking a step further and consider the impact on integrity, capability, reliability, efficiency, portability, maintainability, compatibility and usability of the software [48].

This section describes the main models behind different testing practices. Section 3.4.1 explains the maturity of the software and its impact on the software process. The next Section 3.4.2 defines testing maturity model which can be used as a guideline when a test should be automated. Finally, Section 3.4.3 describes testing practices that could be used in an agile microservice development process.

3.4.1 Capability Maturity Model

Capability Maturity Model [49] is judging the maturity of the software process. It is used to identify the key practices that are required to improve an organization's software process. The software process usually evolves from ad hoc chaotic processes to mature, disciplined software processes [49]. CMM is divided into five maturity steps which are the following [49]:

1. **Initial.** This period is characterized as chaotic because the organization does not provide a stable environment. The organization lacks management practices and uses reaction-driven development without good planning. The success of the project depends mainly on individual effort and heroics.

2. **Repeatable.** The initial project's scope has been agreed and basic project management processed are established. The project starts tracking different metrics to follow up cost, schedule, and functionality. Successful teams tend to work disciplined towards a minimum viable product in this phase.

3. **Defined.** This period tailors the software process for this project and provides support for project maintenance; documentations, standardized work style, intergroup coordination, and training programs. The team starts to use common code quality monitoring practices like peer reviewing. Software engineering activities are stable and repeatable.

4. **Managed.** The period includes software quality management into the process. The process has a consistent output which is quantitatively understood and controlled. The level can be described as predictable because the process is measured from management and engineering sides and the process operates within measurable limits.

5. **Optimizing.** A stable process requires piloting innovative ideas and technologies to ensure continuous improvement. On this level, the process is once in a while challenged to find better ways to prevent known types of defects from recurring.

3.4.2 Testing Maturity Model

Testing Maturity Model (TMM) [50] is developed to support Capability Maturity Model (CMM) [49]. TMM defines when an organization should improve their testing capabilities. It defines when software testing should be automated. The following five steps describe the behavioral characteristics of the TMM levels [50]:

1. **Initial.** Chaotic phase; Tests are implemented after the coding is done. The testing is done in ad hoc way and its objective is to ensure that the software works. The software does not have any quality assurance, and testing lacks resources, employee training, and tools.

2. **Phase Definition.** Testing has been separated into a planned activity; Testing ensures that the software meets its specifications and it is planned mostly after coding is done. Thus, most of the quality problems occur late in the process. Moreover, there is no code reviewing and most of code testing is done manually.

3. **Integration.** Testing is integrated into the entire software process. Testing is considered as a professional activity and team members are educated and encouraged to do testing properly. The team has communicated the importance of code reviewing and starts using it as a basic quality assurance method.

4. **Management and Measurement.** The testing process is measured and quantified. The software is tested according to quality attributes such as quality, usability, and maintainability. Out-dated test cases are systematically stored for later use. Furthermore, defects are logged and analyzed regularly. The environment supports automated collection of test related metrics and provides analysis of the results.

5. **Optimization, Defect Prevention and Quality Control.** This phase is optimizing the testing process, because at TMM level 4 the testing process is defined, and its effectiveness can be monitored. The performance of automated testing tools and the process is regularly monitored. The testing tools are used to provide support for test case design, allow rerunning of test cases, defect collection and analysis, and to compose and analyze test related metrics.

3.4.3 Agile Software Testing Practices

Agile software development shares code responsibility [51]. The team should form common goals and practices for the pursued code quality. For example, assigning test writing to all team members improves their test awareness and allows more relevant tests to be implemented. For example, test predefining and code reviewing practices are used to ensure that the team keeps up their quality.

Moreover, in agile software development, a dedicated test person may become a bottleneck, because all test changes would go through only one person [51]. Thus, the dedicated test member should share his knowledge with the team. The tester may change during the project, so sharing and facilitating the other team to understand the testable functionality is important. Furthermore, unit and service testing should be mainly the development team's responsibility, but a facilitating tester may become handy after the project's maturity has reached a sufficient level. Additionally, unit testing mainly applies to the white-box testing methodology, so feature knowledge speeds up the testing process. [48]

Agile software development process specifies code measuring process. The product size should be determined by the test size because it describes the code complexity better than the lines of code or specifications. Thus, mature agile software project iterations should include broad regression testing before the builds are delivered forward. Furthermore, agile software development reinforces the meaning of trust. Untested work should not be accepted in a mature agile software project. [51]

End-to-End Testing Practices

The benefits of E2E testing are constantly discussed. Wacker argues that the E2E tests are not worth implementing [52]. E2E tests are implemented so that they simulate real user scenarios. Many team members likes E2E tests; from a developer point of view it outsources the testing responsibility, from a manager point of view it is a clear indicator how a failing test case affects the user, and from a tester's point of view E2E tests include a larger scope, so a test does not need to test every single service so comprehensively.

In practice, Mike Wacker [52] presents that E2E tests should not be the main focus when fixing a problem. Google's strategy [52] is to *"focus on the user (and all else will follow)"*. The E2E tests do not directly benefit the user. The user benefits from a bug fix, where an unintended behavior goes away. The developer should find the real cause of the problem, not only fix the E2E test to pass. He says that common mistakes with E2E testing are flakiness, fixing for the test instead of for the real user, too big number of E2E tests, and sometimes E2E tests should be moved to service test level or split into two separate test cases.

However, Fowler [42] sees that a few E2E tests help the development process. He says that E2E tests allows to test the behavior of the fully integrated system. This allows better quality monitoring. He has introduced the following guidelines to achieve a successful end-to-end testing process:

- *Write as few end-to-end tests as possible*
- *Focus on personas and user journeys*
- *Choose your ends wisely*
- *Rely on infrastructure-as-code for repeatability*
- *Make tests data-independent*

3.5 Summary of Findings

Microservice architecture is still very recent, so it has no exact definition. However, microservices have a recognizable style, which can be used to compare microservices to other systems. In practice, microservices are a distributed system so it includes all features of distributed systems. Hence, microservice's testing can be monitored from a distributed system's viewpoint. Microservices splits a software into more maintainable pieces, which can be built, run, re-factored, and tested separately. Small compact services also reduce the need for unit testing because a microservice does not include many complex functionalities.

The difficulty of testing has moved to the network and configuration layer, where a developer should implement the massive amount of test infrastructure compared to a monolithic solution. Each service requires a separate test database and data initialization to all these services is complex. Broader tests should ensure that all microservices are up and running before running the tests. This can be achieved for example by calling services' health checks.

The distributed database model brings up a test data initialization problem; the data should be inserted into databases so that all services have the correct information. Test data initialization can be done through an API, using separate migration files or by using separate test data scripts. The method varies by the project's maturity, scale, and schedule. However, the overall idea is to empty the test databases and after that add the test data, so that the test databases are always starting with the same initial state.

Furthermore, the testing scope of the microservices is varying by the compactness of the service. The services should be designed for failures, so they will not crash into cascading failures after one service is temporarily off-line.

Generally, microservices have better delivery pipelines built with the services, because the process requires proper continuous delivery tools in order to benefit from a microservice solution. The process allows fast and iterative user acceptance testing, which improves the transparency of software development.

Chapter 4

Results

This section shows the case projects case by case and presents the research results. The section begins by presenting the case projects in Section 4.1. Each case project is briefly explained and the interview results are presented case by case according to the case study methodology. After the case projects review, the research results are shown in Section 4.2. Section 4.2 summarizes and categorizes the main findings from the case projects.

4.1 Case Projects

The following sections concentrate on the case projects' structure, development process, and interview results. The case projects revealed that initial software testing practices depend on team structure, software requirements, and project schedule.

All case projects are iteratively developed, less than three years old, microservice based right from the beginning, and carried out by the public sector. Public sector projects have clear overall requirements for the software right from the beginning. Clear overall requirements support microservice architecture style because the division of the microservices has to be done in early phase [24].

The results are divided into four different categories: team structure, software development process model, software structure, and technical implementation. Each category has an impact on software testing practices. The most crucial categories by overall project success are the team and process based categories. These define the overall testing resources and the testing procedure. Additionally, the software structure specifies the testing objectives and provides a clear application interface to be tested. For ex-

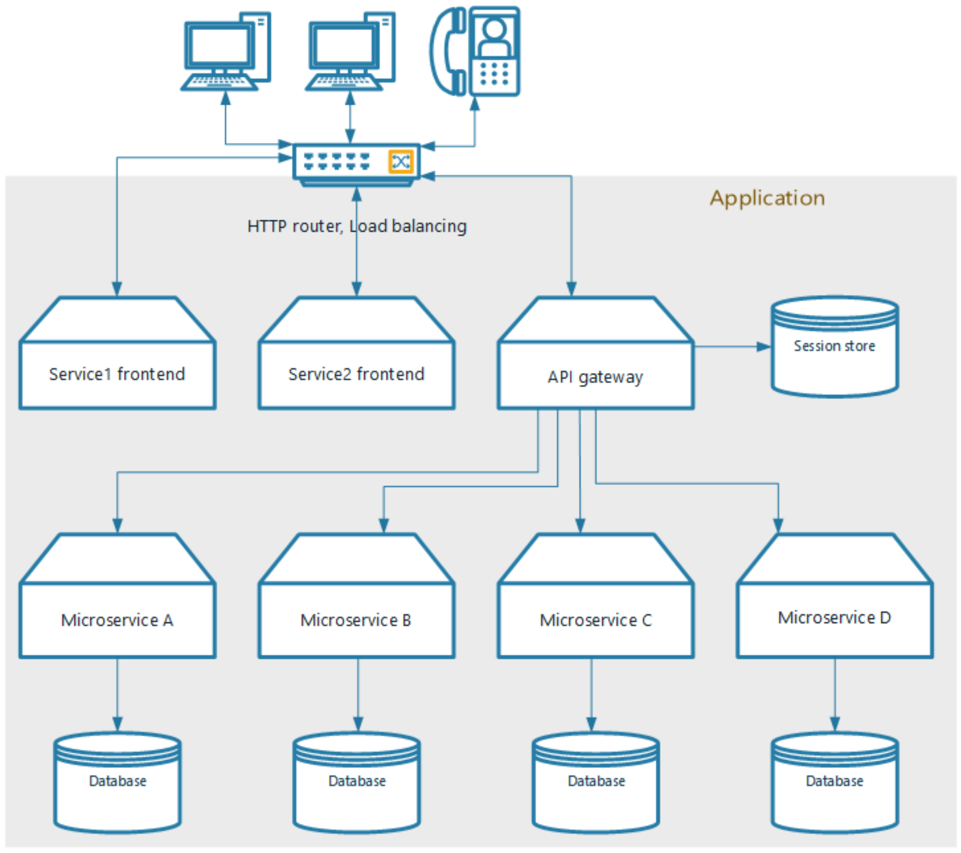

Figure 4.1: Abstraction of the Solmu project's microservice architecture structure.

ample, the software structure defines the amount of work for integration testing. Furthermore, technical testing implementations add more precise details into the process. For example, simulated user testing requires support from the software development team in order to be beneficial.

4.1.1 Case Solmu

Software Architecture

Solmu is using an application programming interface (API) gateway centric design, which means that all requests go through the gateway (see Figure 4.1). The API gateway handles user permissions, composes and delivers answers for the requests. This design is common in small microservices because it is simple to implement and to maintain. The architecture fulfills the loose coupling requirement of microservices; a microservice does not need to know about other services because all requests go through the API gateway. On the other hand, the gateway keeps track of the location of each of the microservices. The API gateway also contains some routing logic, for example, it is responsible for composing answers to API queries from multiple microservices. This means that if a query requires information from microservices A and B, the gateway calls both of these microservices and returns the composed result (see Figure 4.1).

Nevertheless, microservices are emphasizing that an online service's structure should consists of dumb pipes and smart endpoints. Without this ideology an API gateway will extend and look more like an ESB and the architecture ends up having the same problems as SOA. Thus, the case projects Latu and Kapa have divided the API gateway into smaller pieces.

Afterwards, the Solmu project was a bit criticized about splitting the software into too small pieces. This is a so called nanoservice problem [53] where the microservice functionality is separated into too small pieces, which adds unnecessary network complexity to the software. For example, the complexity made overall testing more difficult. Testing a single service is straightforward, but the tests between the services are more challenging. The test data identifiers have to match between the services in order to be testable. Therefore, the single service tests are testing only a small part of the whole software's functionality.

The API interfaces were mainly tested manually by using an exploratory software testing strategy [54]. For example, the Postman-tool was used to create HTTP requests to the API endpoints. The services were easy to test separately manually because each service's endpoint is visible locally.

Testing practices

From a testing perspective it is relatively easy to add service tests into a gateway module and see that it is working with other services. However, the actual Solmu project had no cross-service testing. Automated tests were mainly API tests, which were implemented inside each microservice and ensured that the API worked correctly. However, more algorithm oriented services included also unit testing to ensure the algorithm's functionality. For example, one service calculated the form results, so the calculation process was tested by unit tests.

The API tests were using a private test database. The test database was initialized with hard coded data to allow for automatic testing. Test databases allow realistic testing of the service and allow discovering database specific problems. However, implementing an actual test database introduces a test data synchronization challenge. The test data references need to match between the databases. This was achieved by adding a separate test database migration step. A separate test migration file allowed that the hard-coded test data did not end up in production when the build was promoted from the test environment.

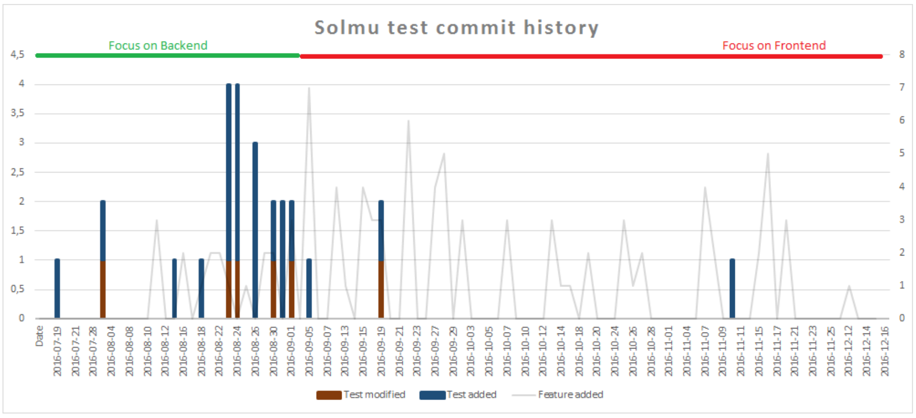

Figure 4.2: The figure presented how many test modifications have been made after the test creation. The graph also visualized the backend services' feature flow.

Figure 4.2 shows the Solmu project's test related commit history. It shows how many test modifications was done after the test was implemented. Most of the test cases were implemented at the beginning of the project. Furthermore, the tests were not frequently modified. The project started with backend development and moved to frontend development after the main infrastructure was made. Thus, most of the tests were done when the new microservices were created.

Additionally, Figure 4.2 presents the feature flow frequency which is analyzed through Git version control merge commits. The team used a pull request practice which means that every new feature should be implemented in a separate branch. Hence, after a pull request was accepted, the feature was merged into the mainline and the merge commit was committed. The data was collected by analyzing the version history commit log.

The graph shows that the code does not include any service testing because there are no updates to tests after frontend development was started. The added feature measurements are only taken from the backend services, so the changes that only affect the frontend service are not visible. The graph shows that either the code maturity is on a sufficient level regarding service tests or that service testing is lacking.

Team process

The Solmu team used an agile software development framework called Scrum for managing the software development process. The project had a full-time product owner from the customer's side to prioritize the upcoming tasks in the product backlog. A full-time product owner made communication in the project a lot easier and most of the UI testing was done by simply showing the local version running on the developer's machine to the product owner. However, the project also included development, acceptance testing, and production environments, but the development environment was mainly used for testing.

The acceptance testing environment was implemented at the end of the project which restricted its usage. User acceptance testing (UAT) mainly revealed environment or browser specific problems. For example, different browsers may render or print the web page items in different order. The testing environment supported only a few browsers and the product owner was responsible for defining which browsers should be supported. User acceptance testing also helped to point out contextual mistakes and usability problems of the software. For example, the testers were able to discover misspelling, unclear functionalities, and misleading notifications.

UAT testing was performed only a few times during the project because the schedule was tight and there was no time to automate test cases to go through the software thoroughly before the software was released. In addition, sometimes inviting the test users was challenging. The test users were not really motivated to go through the whole software. They concentrated on the main functionalities and created a quick user test review. Another study could be made on how to involve test users efficiently.

Interview insights

The interviews brought up team and technology based insights. The team insights are the following:

- Team members' interest in implementing automated tests varied which affected test coverage. However, by instructing teammates how to create effective automated tests, the problem decreased.

- It was easy to forget to run the tests because the tests were processed in a separate process and were not executed automatically when a feature was committed. So, the tests were not ran automatically on CI server.

- The software used mainly API and unit tests. The UI was manually tested because automating the UI tests did not add enough testing value. On the other hand, unit tests worked greatly if the functionality did not have external relations. If the functionality required database access or other external relations, testing became more challenging.

The technology based insights are the following:

- Most of the implemented automated tests focused on special cases, for example, testing that the API returns the correct answer based on the session key.

- A real database is a better solution than mocking a database. However, adding test data to a real database is problematic. One way is to start with a clean test database and add data through API calls. However, in this project test data was hard coded to the test database.

- At the server side, 100% test coverage should be possible. However, this project was in a rapid development phase, so most of the testing was done manually with direct HTTP calls generated by the Postman tool.

- Microservices are easy to test if the services have clear boundaries with compact inner implementation. In this project, the services were split into too small pieces, which made testing more difficult.

4.1.2 Case Latu

Software Architecture

Case Latu's architecture looks like an extended version of Solmu architecture. Latu implements multiple end-user services through the same API gateway. Solmu had only separate front-ends for different users, but Solmu was still only one service. Multiple end-user services create a need for more sophisticated version control practices in order to prevent code duplication. For example, Latu had a common repository for services' styles, which were included and used by each of the services. Furthermore, multiple services increase testing work, because added functionality has to be tested through multiple different services.

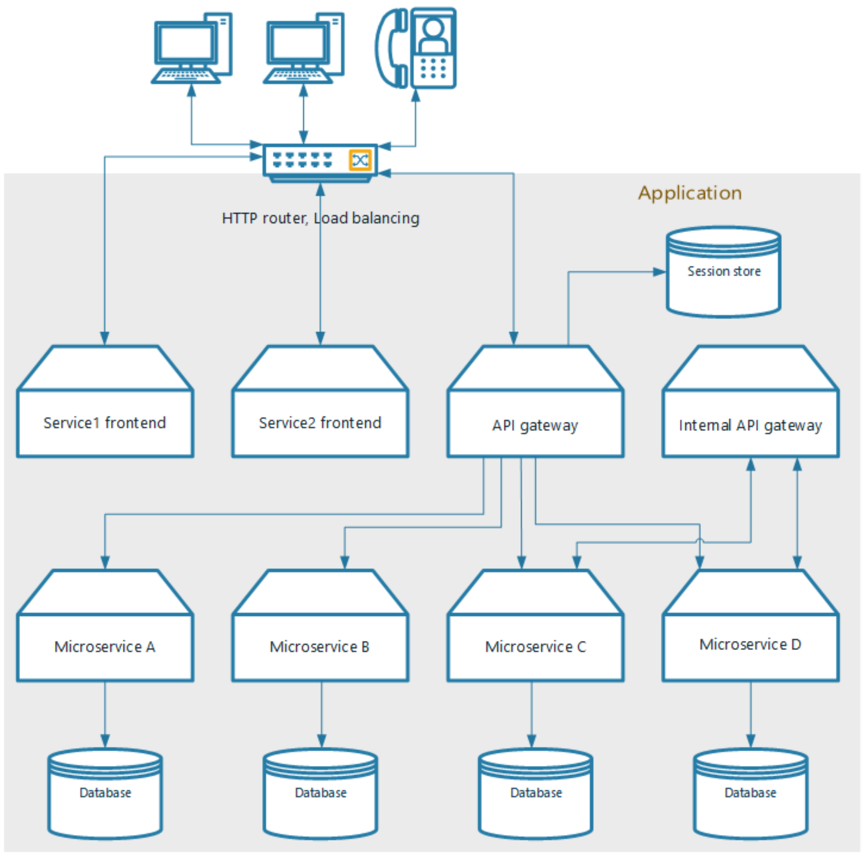

Figure 4.3: Abstraction of Latu-project's microservice architecture structure.

Moreover, Latu extended Solmu's architecture by dividing the API gateway into two separate components (see Figure 4.3). Latu had the main API gateway for public requests and a private internal API gateway for internal communication. This allowed more complex requests to go through the internal API gateway. To clarify the concept, an API gateway only delivered the messages to the end-services. These end-services composed the answers by using the internal API gateway to gather external data dependencies. Therefore, the internal API gateway simplified the main gateway's structure and provided better load balancing support.

Testing practices

Latu also had different testing challenges compared to Solmu. It had many unit tests at the microservice level, but cross-service tests were missing. The unit tests were mocking databases instead of implementing and running an actual database. The project ended up using mocked databases because populating the test databases was too challenging withing the project's schedule.

Furthermore, Latu did not have end-to-end tests at all. These tests were dropped out because the tests were often broken and the schedule was too tight. However, now the team is planning to bring some end-to-end tests back because the software's maturity has reached a decent level. Automatic tests became more useful when a project moved into a maintenance phase. Therefore, the beginning of a project should concentrate on rapid development if the goal is fuzzy and rapid prototyping does not risk the goal. In this project, end-to-end tests were created too early to add actual benefits. However, based on the interviews, tests were automated when the manual testing process had taken too much time. The importance of automated tests was recognized with a delay because a new property's testing time increased gradually.

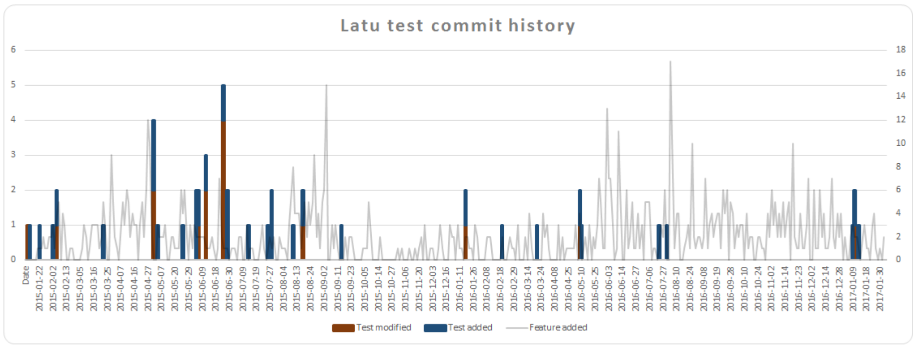

Figure 4.4: The figure presents how many test changes have been done after a test have been added.

Figure 4.4 shows the Latu project's test related commit history. It shows how many test modifications were made after the test had been implemented. Most of the test cases were implemented when the backend services were created. Furthermore, the tests were not frequently modified. The project started with frontend development and moved to backend development after the main user interface was made. Figure 4.4 shows that most of the tests were implemented when the new backend services were created.

The version history data was analyzed in the same way as in the Solmu project (see Section 4.1.1). The graph also shows that the tests have not been frequently updated after test creation. One reason behind this is that the service does not have many cross-service tests. Broader tests would need more maintenance during the sprints.

Team process

The Latu project is using fast and small iterations in order to keep the common main branch up-to-date. At the beginning of the project, the project used Scrum, but after service maintenance responsibility increased, they modified the development process towards Kanban. The Kanban process is a constant workflow, whereas Scrum is divided into sprint planning and sprints. Thus, they switched from Scrum to Kanban after service maintenance got more priority.

Currently, Latu's average pull-request review time was around 10 to 30 minutes. For contrast, in Solmu this was about 5 to 15 minutes, but Solmu had only one end-service. The code review included code checking and testing the updates on your own machine. The team took a pair-review approach into use about a year ago. They said that it has improved the code quality a lot. Before this practice, team members were free to commit their own changes to the main branch. The code review practice created a base for a common coding style and allowed faster tracking of environment specific bugs.

Latu had also a common style guidance and version control strategy. This helped to keep the code in-line between microservices. The style and naming guidelines was extremely important with microservices because the microservice's API can be used from multiple locations and renaming all those end-points is time-consuming and requires broad testing of the whole application.

Interview insights

The interviews brought up team and technology based insights. The team insights are the following:

- The team does not have strict roles. The decisions are discussed within the team and the team members roles vary.
- Scrum is good at the beginning of a project, but Kanban has better

maintenance support. The Kanban process has better support for implementing small changes.

- Common coding guidelines are useless without monitoring. At the beginning of the project, the team used common coding guidelines, but the code quality still decreased over time, because the guidelines were not monitored regularly.
- A code review practice improves code quality a lot and allows more stable development builds. The practice ensures that the code is also compiling on an other environment and checks that the coding style is uniform and correct.

The technology based insights are the following:

- Microservices as small compact entities make testing easier. Microservices do not have as large side-effects from changes as monolithic solutions have. Thus, microservices do not need as comprehensive testing as monoliths.
- Javascript languages are preferred in microservice development because those provide a lightweight, clear, and compact implementation framework for the microservice's needs. Microservices may include many compact create, read, update, and delete (CRUD) services which are quite straight forward to implement and additional extra code adds unnecessary complexity.
- Java virtual machine consumes a lot more memory than a lightweight Javascript framework on default settings.
- Testing a pull request on one's own machine is important because it will reveal environment specific problems.
- Microservice tests are easily built on top of previous tests, which may not be the best way to improve test coverage. For example, if the service has already a few tests it is easy to add a few more tests to the same place, even if it would not be the optimal solution. The reason behind this is partly that microservices were implemented by using varying technologies and a developer may not have the complete understanding of all testing frameworks that the whole service is using. Thus, a developer may end up using the old testing code as a baseline with only a few modifications.

- Initializing a test set up should be done at the same time as a new microservice is created because it will decrease developers' test writing threshold. If developers are unfamiliar with a microservices' framework they are not building the whole test infrastructure around one task.
- Real databases are better than mocked databases because those show the real performance of the database. However, real databases require setting up containers, configuring the network layer, populating the test data and setting up the entire database. The database should be the same as in the production because lightweight test databases do not include all database specific properties.
- If developers do not have good knowledge about a testing framework, they will probably just copy-paste existing test cases. Furthermore, if a service does not have previous tests a developer may not implement the test case because it would need too much infrastructural effort.

4.1.3 Case Kapa

Overall, Kapa is an extremely large software project, which consists of many different services. In this review, I bring up the architecture's sophisticated structure to contrast it with the other case projects. Kapa also includes API gateway centric architecture models, which are similar to project Solmu and Latu.

Software Architecture

One part of Kapa was using a Netflix-styled architecture (see Figure 4.5). Instead of having the API gateway manage the routing to different services, it had a discovery server (Eureka) to inform other services about their connections at startup. This allowed microservices to communicate with each other without going through an API gateway. For example, see from Figure 4.5, how microservice C and D are connected. In this design, service tests can be implemented into each microservice, because the information does not flow through the API gateway. This allows creating more compact services because service-related tests can also lay on the same level.

End-to-end tests required test expertise at the beginning of the process because the tests should be built modularly. For example, the automated test framework Selenium uses page-view patterns, which should be implemented as modularly as possible to provide a good base for test modifica-

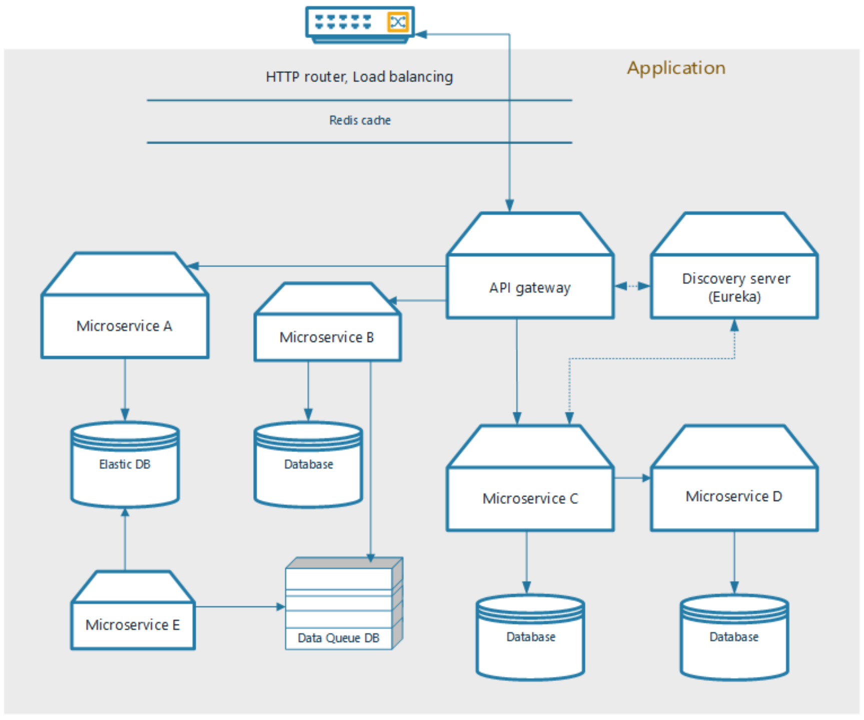

Figure 4.5: Abstraction of the Kapa-project's microservice architecture structure.

tion. After the first tests have been implemented modularly, the extension of tests is easy and less experienced testers or even developers can perform the changes.

Testing practices

The project assigned one tester to each developer team to facilitate and contribute E2E tests, because developers interest, competence, time and project pressure do not leave room for broader tests. The practice was successful. However, the team had to be willing to work with the tester because without co-operation the tester will not bring extra value to the team.

The teams that fully integrated a tester into their team had a better understanding of the developer's test responsibility, for example, implementing additional statical identifiers to UI components right at the start to help a tester to write a test case for that functionality. Creating a test afterward may be a really expensive task and usually the tester asks a developer to fix their solution to be more testable. If the code has to be changed afterwards, it increases the waste time which is against lean software development principles. Modifying a software afterward is almost always more expensive because a developer needs to gather the knowledge about the issue and find a time slot to fix it.

Another challenging task was to unify testing practices inside a team because experienced developers had formed their own ways to develop and test their software. They may have really good arguments, why the testing process should be done in another way. Nevertheless, setting common testing practices will improve overall code quality because of the common testing structure, allows better code reviewing, and provides better co-worker support.

Team process

The Kapa project consists of many teams, and each team had different processes. However, I will reflect on the Netflix-styled architecture team. They were using a Scrumban iterative process. It was a combination of Scrum and Kanban. The team was using 2-week sprints, but they also received some maintenance related tasks behind of the backlog. The team was satisfied with their process, even if it seldom went by the book. The project included so many parties, that ownership of the source code, servers, tests, databases varies and makes the delivery process more difficult. Technically the process was the same, but the developer team's involvement was different. For example, the developer team delivered the software to UAT and another team approved it into production. However, the developer team was still responsible for internal testing of the services, because UAT will be done as black-box testing which means that the testing team was only testing the services from the user point of view.

Kapa was a large software project, so it had many servers just for regression testing. Regression tests required a lot of capacity from the servers and therefore all services cannot be regression tested in parallel. All teams wanted their software tested right after the sprint, which creates a scheduling pressure for regression testing. However, the teams solved this problem by creating variation to sprint end dates.

Interview Insights

The interview insights are grouped into four different categories. These categories are formed and named by the clusters of the composed interview insights. The first category presents the team related testing insights, which are the following:

- An assigned test team member helps to maintain a high testing level (E2E or service tests). However, the team also has to be willing to work with the tester. The tester helps a normal developer team to understand testing requirements and take those into account when developing the software.
- The testing team should be involved in the development process when the project's maturity is on a decent level. Larger scale autonomous testing is a waste of time if the project standard paths do not work properly.
- The developer sees that the testing responsibility is on the testing team if the project has one. The testing team counts on the developers to test their software properly before it reaches the testing environment. Anyway, the product owner is responsible for the overall quality, which covers a lot more than just test coverage.
- Usually, testers do not participate in software functionality planning. Thus, the developers should also have a testing vision in their mind.

They may point out critical points afterward to ensure the software's testability. Improving service's testing supportability is usually cheaper than implementing difficult test cases around the software.

- Developers should not fall in love with their code. A developer may be hard to convince to take a look into their code when the test results are blurry.

The software development process related testing insights are the following:

- Testing should be started by using ET, in order to find the actual problems of the software. Autonomous tests should be used to ensure code functionality, but these should not break regularly in an acceptance testing environment.
- The problems should be found as early as possible, so that fixing would not waste time into re-learning the functionality. When a problem is found in the development phase, it is a lot faster to fix, because the programmer has a clear understanding of the software's structure at the time. Fixing the same problem later will take more time.
- Developers are following agile principles and they concentrate on value-adding testing. On the other hand, the testing team has broader testing targets, which are test maintenance, manual bug hunting and improving test coverage.
- The test team does not have visibility of the inner system. They are running black-box tests, so the inner implementation of the software does not impact on the external testing team. Thus, the test team does not care if the underlying service is a microservice or not.
- Regression testers are not able to see the whole load information during the test because usually in mature big software projects the user rights are limited. Nevertheless, they have a clear area of responsibility which to monitor and to report.

The software structure related testing insights are the following:

- Creating too small microservices produces unnecessary testing complexity.

- Small microservices have more service-level tests than unit tests because dummy database-access service does not include any complex functions. Thus, testing is concentrating on service level, where it adds more testing value. The tests typically go through the whole API access point.
- New properties are manually tested after the autonomous basic tests have been completed.
- Regression testing needs lots of computational resources, so its schedule should not overlap with other teams. Usually, most of the teams want regression test results after the sprint, which may cause overlapping.
- CDC tests are insufficient for UAT. The view of the page is important and CDCs do not test the front-end at all. Anyway, CDCs are a really useful tool for regression testing, because they run faster than E2E tests and are not as fragile. However, a software's UI may be broken after CDC regression test, but the test does not detect it.

The technology related testing insights are the following:

- Maintaining UAT is difficult because many services have specialized needs. Moreover, some services contain cross-references to other services. For example, case Kapa has formed a stateful relation between services, even though every single service is stateless.
- Services should use as real test database if possible. The real database allows realistic load testing, response time measurement, and database query testing.
- Existing databases are easy to double, but test data import varies through services. Teams may end up using mock-tests because test data migration is too troublesome.
- Test data can be inserted into a database by using separate database migrations for test usage. This way the database would always have the same testing data. Another style is to add data by using test functions. For example, first call data insertion functions and later test other functions. Both of these methods were used in the Kapa project.

- End-to-End tests ensure that the core functionality works. Anyway, the interface has to go through extra manual steps, because monitoring the page layout cannot be effectively automated.
- Developers have to take testing into account. For example, if a developer does not give specific identifiers to form fields, creating an autonomous test case will be difficult. With identifiers, test programs can simply select the field by the identifier, but if this is missing, the program has to figure out where the field could be, which is not so accurate.
- Under one percent of the automated tests should be modified weekly, in order to benefit from them.

4.2 Summary of findings

Most of the findings were highly dependent on the project's maturity. Thus, the following result tables describe successful software testing practices with the related software maturity boundaries. The result tables are using CMM levels to describe a project's maturity level.

Moreover, most of the findings were not microservice specific, even though they came up from a microservice environment. To ensure broader use for these results, more case projects are needed.

The results revealed four different categories which are the following: team related (see Table 4.1), software development process (see Table 4.2), software structure (see Table 4.3), and technical (see Table 4.4).

Table 4.1: Team related testing practices

CMM level	Result
2 - 5	The team should have a basic understanding of agile software development responsibilities and take lean principles into account when designing and implementing software tests
3 - 5	One tester in each team improves the team's testing motivation and understanding of testing requirements
4 - 5	E2E tests should have a dedicated person
4 - 5	Regression testing should be scheduled so that it does not overlap with other teams
5	The team should include an active and motivated exploratory tester to find possible problems in the automated testing environment. The basic coverage testing can be outsourced to less experienced testers because the basic structure of the automated tests already exists

Table 4.2: Development process related testing practices

CMM level	Result
1 - 3	Software testing is only a small part of quality assurance. Other tools like code reviewing, static code analysis, and service monitoring allow for efficient code quality control in immature software projects
1 - 4	100% service side test coverage should not be a goal. The goal should be to test real functionalities properly and leave implementing tests just for code coverage out.
1 - 5	Testing should be started by using an ET approach to understanding what kind of tests are useful to automate.
1 - 5	Peer code review is an effective way to improve code quality
1 - 5	Flaky tests should be removed or fixed
4 - 5	Test automation should be started on the E2E level when project maturity has reached CMM model's managed level.
4 - 5	Testing becomes a more important quality assurance tool in mature software projects. Mature software projects tend to be more critical services and therefore the process is trying to eliminate most of the bugs.

Table 4.3: Software structure related testing practices

CMM level	Result
1 - 5	Tests should be implemented according to a lean principle. In practice, this leads to small microservices having more service tests than unit tests.
1 - 3	Microservice testing infrastructure adds initial complexity to test creation. For example implementing one test case into a new service requires setting up the test database, initializing the test environment, and writing the test code.
2 - 4	Automated microservice integration testing requires higher maturity from the software than for a monolithic alternative. For example, automated integration testing should be started at level 3 - 4.
1 - 5	Too small microservices increase testing complexity. For example, functional testing may need to be done for the API gateway instead of related services.
3 - 5	The team should consider when to add a new testing framework into a microservice because new testing frameworks adds testing complexity and requires developers' with a broader technology set.
4 - 5	A new microservice feature should be implemented by only touching one microservice at a time. A mature software projects tend to have many software developers, so modifying multiple locations will cause unnecessary merge conflicts. Thus, in mature software projects developers write test cases for the implementation and ensures that the microservice works as expected before modifying other microservices on the way.

Table 4.4: Practices to solve technical testing challenges

CMM level	Result
1 - 2	Mocking a database is faster to implement than building infrastructure for a test database. So, in a hurry mocking a database is okay, but when the project reaches CMM level 3, the databases should not be faced anymore.
3 - 5	The database should be created as real as possible when the project reaches E2E testing phase, but before that mocking the database is a decent solution.
3 - 5	Test data should be added by running a separate script or by using additional database migrations.
4 - 5	Developers should give individual identifiers to UI components, so testers can easily access UI elements.

Chapter 5

Discussion

This section presents the answers to the research questions and reflects those with the literature review. Section 5.1 presents answers to the research questions. The following Section 5.2 compares the results with the literature review. It brings up the common practices with microservice testing. Section 5.3 explains the limitations of the study. Finally, Section 5.4 presents the possible future work for the study.

5.1 Answers to the research questions

This section presents answers to the overall research question: *What kind of testing practices are software engineers recommending for public sector microservice software projects?* The practices are presented in the previous Section 4.2.

How does testing a microservice architecture differ from testing a monolithic architecture?

Microservices are compact small services, which complete a clear functionality. A microservice is easier to understand and to maintain than a monolithic alternative. Small microservices tend to have more service level tests than unit tests because the microservice itself has decreased the unit testing complexity. Nevertheless, the integration testing between microservices is more complex than with a monolithic solution. Monolithic architecture integration testing allows shortcuts, but with microservices each connection has to be properly implemented or at least faced by using test doubles.

Microservices require a higher maturity level than a monolithic alternative when starting service testing. TMM level 3 defines integration testing of the entire software which is usually achieved at CMM level 3 [50], but with microservices, it is closer to CMM level 4. The cause to this is the additional infrastructure complexity of a microservice process, which has to be stable on each service level before service testing is beneficial. The services could also be stubbed before implementation, but at the beginning of the process (CMM level under 4), the time pressure drives projects towards results instead of testing coverage.

In microservices 100% service test coverage is not needed because the service structure has limited indirect relations between the modules. A monolithic architecture has many relations with other components which a developer may not be aware of. Thus, monoliths are creating broader test cases for the functionalities. On the other hand, microservices are mainly adding tests for complex functionalities.

A large microservice has more monolithic features. In contrast, the simplest microservices are just dumb database CRUD services, which provide basic functionalities to access the database. In small CRUD services, unit testing is mostly useless and the tests are testing the API interface instead.

How does a microservice environment influence testing?

Microservices support agile software development processes. Thus, testing can be done concurrently with software development. Furthermore, testing single services is quite straightforward. However, small tests do not add so much value in compact services, so the testing may move to the service layer. Nevertheless, service tests are more difficult and complex to implement and to maintain. Hence, these tests are usually implemented after the project's maturity has improved into a more stable state (CMM level 4). A stable state means that the core use cases of the software work most of the time and the tests are mainly ensuring that the basic functionality is okay, but the tests should not fail weekly.

Why do project testing practices differ from the recommendations?

The project schedule is not reserving time to build testing infrastructure before problems start to occur. Microservices requires a large testing infrastructure even to small test cases. Typically there is no time to build testing infrastructure while implementing new features. Thus, overall testing is always a bit behind the project, if a project does not have a specific testing team. Moreover, at the beginning of the project automated testing does not add significant value to the software development in an agile software environment, because the automated tests are out-dating constantly. Hence, test automation should be started after a project has a sufficient maturity level. However, maturity level can be identified individually for all microservices. After a microservice's maturity level reaches CMM level 3, automated testing should be implemented and maintained professionally (TMM level 2+).

Another problem is the testing team's vision. Testing practices vary inside the project because a team is mainly reviewing the results instead of test coverage at the beginning of the project. However, the review process concentrates more on testing, when a project's maturity level increases. Moreover, developers need to change their style after the project has reached a more stable point. If a microservice project is implemented in a really small team and at the beginning of a microservice project, a developer creates many changes into multiple repositories in order to complete some functionality. At the maintenance phase or if the team size increases, a developer should implement the functionality in one repository at the time. Moreover, they should ensure the microservice's functionality by creating tests before moving to the next microservice. Incremental testing improves code quality but slows down the development of functionality.

Additionally, all case projects emphasized the importance of a realistic testing database, but despite this the projects ended up mocking the databases. The mocked databases were faster to implement and require less environment specific configurations. Furthermore, a problem with real databases is the initialization of the test data. The projects were using different initialization methods, based on the situation and intuition; separate database test migrations, adding the test data through API or running separate database migration scripts.

Moreover, in a microservice context poorly implemented test cases tend to multiply after an initial implementation. The services are quite similar, so it is easy to cut and paste similar parts of the software and software test implementations are close by copying existing test cases and modifying them. Thus, the initial testing implementation easily spreads around the software, even if it would be better to create a new test case by using a different methodology. Furthermore, microservices allow and encourage to use the best technology for each service. Thus, a developer may need to work with multiple different technologies, which all have different testing frameworks. The lack of knowledge of testing libraries drives developers to use previous implementations as a baseline without thinking about the test case further.

However, microservices are also supporting rapid refactoring. The project should have a common direction because the out-dated test implementations can be replaced by using a new technology after a while. Overall, the project should summarize used testing frameworks, so developers could easily upgrade their knowledge and improve their testing code quality.

How does a project's release process affect testing?

The release process mainly affects user acceptance testing. A modern build pipeline allows rapid monitoring of the upcoming changes. It counts on the product owner who is typically responsible for ensuring the feature's quality. The case projects had part-time product owners to facilitate the developer team. At least in these cases, an active product owner was an extremely important part of the team.

Furthermore, the case projects showed that all testing steps should be automated. If tests are not run beside the build or at the code commit hook, a developer forgets to check the test cases or the code style. The testing steps should be applied one by one when code maturity increases. A mature software release process should include all automated testing steps: code style checking, static code analyzing, unit testing, service testing, end-to-end testing, library security tests, and regression testing. These steps allow good base coverage of the software.

In a large software project, the testing steps should be divided into different categories and scheduled based on the project needs. For example, one common solution is to run tests that require more time during the night.

5.2 Comparison between results and literature

Fowler [42] states that end-to-end testing is a feasible solution if the tests are selected correctly. However, according to the case projects, maturity, team's test interests, and project's schedule affect more end-to-end testing decisions than Fowler's E2E testing practices. Martin Fowler [42] did not clarify when end-to-end testing should be started. Moreover, Sam Newman [2] pointed out the responsibility problem of E2E testing, but he did not describe a solution to the problem. Based on the interviews, a separate E2E testing person helps to focus the E2E testing responsibility.

Furthermore, all projects are not even aiming to have CMM level 5. Too high maturity level adds unnecessary stiffness to projects. In small, not so critical services, the projects are more like experiments which still search the product direction. Thus, the process is commonly more flexible and does not need to standardize everything. Most of the literature, including Sam Newman [2], describes really mature software projects and how software testing should be applied in those cases.

Large software projects which utilize the whole power of the organizational structure [2]. Nevertheless, on smaller projects and less safety critical services, the customers are more willing to be flexible and more easily follow a break fast and fix fast -strategy. Releases are created constantly and upcoming bugs are fixed rapidly. However, a less organized testing process increases the number of bugs in the end product, but with the constant improvement, the bugs are fixed rapidly.

The interviews pointed out that quality assurance in small scale projects is highly dependent on team practices. More mature software projects have more solid testing environment and well-implemented testing practices. On the other hand, small projects are heavily relying on developer's individual effort and motivation.

The traditional TDD approach is mainly used when the service has reached a mature state. A developer has to modify services one by one and test the modifications before moving to the next service. The main reason behind this is that mature software projects tend to have more people and concentrating one service at the time decreases the number of possible conflicts with other developers. For example, Sam Newman [2] supports the TDD approach by presenting multiple ways to implement test doubles to enable service testing.

5.3 Limitations

This study has many limitations. First, the small number of case projects decreases the generalizability of the findings. Second, all case projects were at least partly implemented by a single organization. Furthermore, the case project size varied a lot so the project pool included only one sample of each project size.

All case projects were designed and developed in Finland and implemented for the public sector. Thus, most of the case projects had similar requirements and quite clear specifications right from the beginning.

The interviews also posed some limitations. For example, the interview questions (see appendix A.1) were translated on the fly, so the final question form varied between the interviews. Furthermore, I conducted the interviews alone, even though a case study interview would require multiple observers. Moreover, on the Kapa project, I had a test manager with me to ensure that the interviewees did not reveal business secrets. This may have disturbed the interviewee's answers, but on the other hand, I was able to gather more information at the same time through the test manager.

Personally, I also had some problems with proceeding with the interviews step by step. However, I was able to adapt to the situation and used the questionnaire template more like a checklist than a survey. Therefore, the questions did not have the same format every time.

5.4 Future Work

The found microservice testing practices would require a pilot project to test these findings. The practices should be tested with multiple projects to be able to compare the found practices to the previous approaches. The research should also consider the effect of the agile software process because based on this study the team process affected testing more than the software's architectural model. In addition, further researchers could clarify the test practices' boundaries so that the project could have a more specific guidebook how to apply testing in a microservice context instead of a general lean principle.

Chapter 6

Conclusion

The thesis investigated practices for microservice testing. The problem was approached through case projects which provided real-life examples of testing practices of microservice software projects. The results were compared with the literature to improve reliability of the results. The literature review explained how things should be done if a project would have unlimited resources. However, in a real-life project, resources are limited and the testing phase is usually limited until problems start to occur.

The case study pointed out that literature does not show a clear path when to start testing and leaves the decision to be formed based on lean principles. However, traditional TMM defines a general framework to use to monitor a project's testing needs. On the other hand, microservices have special properties; distributed independent services with a compact implementation. The microservices could have an optimized version of TMM to provide better support for their environment. This study explored the testing features of microservices.

The case projects revealed four different categories, which define the testing practices: team related, development process, software structure, and technical details. The team and development process related categories ended up being the most significant from a testing perspective because testing resources and scheduling is created based on these categories. Nevertheless, software structure and technical details were able to point out typical software development problems and a few solutions to those.

Team related testing practices were that the team should understand the importance of common responsibility of testing. Each team member should implement valid tests for the implemented feature when the project is in the stable development phase. Furthermore, code peer review provides a perfect tool for monitoring and improving code and testing quality. Additionally, when a project's maturity is rising and the project reaches the end-to-end testing phase, the project should dedicate one team member to be responsible for broader test cases, because otherwise the broader tests will out-date rapidly.

Testing process related results were that the testing process is not scheduled optimally to provide maximal value to developers. The projects' resources for testing are extended with a delay; the development process has already decelerated because of lack of automated testing. The project should monitor the project's stage and use for example TMM to optimize usage of testing resources. Implementing large tests too early is not beneficial because the tests are most of the time broken and do not bring more value to the developers.

The software structure had not significant effect to testing practices. Instead, it affected testing implementations which were closer to a developer than a tester member. Microservice unit and single server API testing implementations were described easier and more intuitive to understand on a single service level. Nevertheless, the service and system level tests were seen difficult and the teams who did not have a specific testing team member did not have the time to keep these tests up-to-date.

The technical testing based insights were highly related to data management. Mature microservice testing requires a lot of infrastructure around the actual tests. The tests should set up separate testing databases. The case projects showed that the separate test database should be created after the project's maturity reaches level 3. Before that mocking the databases is acceptable, but broader tests to the system are not realistic if the database is not real.

Overall, as seen in this thesis, microservice testing is still forming its common practices. Currently, most of the teams are using lean software development principles when the team is agreeing on testing practices. Nevertheless, this is a good basic approach because the team and process related practices have a more significant effect on testing than the software's architectural choice. Furthermore, the testing practices could be more customized to a microservice environment to provide better testing scheduling, resource usage, and more stable testing environments.

References

[1] Martin Fowler James Lewis. Microservices. http://www.martinfowler.com/articles/microservices.html. [Online; accessed 2016, December 12.].

[2] Sam Newman. *Building Microservices.* O'Reilly Media, 2015.

[3] Jonathan McAllister. Microservices decoded: Best practices and stacks. https://dzone.com/articles/scalable-cloud-computing-with-microservices. [Online; accessed 2017, March 1.].

[4] Mark Endrei, Jenny Ang, Ali Arsanjani, Sook Chua, Philippe Comte, Pål Krogdahl, Min Luo, and Tony Newling. *Patterns: service-oriented architecture and web services.* IBM Corporation, International Technical Support Organization, 2004.

[5] Martin Fowler. Continuous integration. https://www.martinfowler.com/articles/continuousIntegration.html. [Online; accessed 2016, December 12.].

[6] Catalin Strîmbei, Octavian Dospinescu, Roxana-Marina Strainu, and Alexandra Nistor. Software architectures-present and visions. *Informatica Economica*, 19(4):13, 2015.

[7] Armin Balalaie, Abbas Heydarnoori, and Pooyan Jamshidi. Microservices architecture enables devops: migration to a cloud-native architecture. *IEEE Software*, 33(3):42–52, 2016.

[8] Ian Sommerville. *Software Engineering (9th Edition).* Pearson, 2010.

[9] Per Runeson and Martin Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, 14(2):131, 2008.

[10] Susan K. Soy. The case study as a research method. `https://www.ischool.utexas.edu/~ssoy/usesusers/l391d1b.htm`, 1997. [Online; accessed 2016, December 12.].

[11] Robert K Yin. *Case study research: Design and methods*. Sage publications, 2013.

[12] Lars Bratthall and Magne Jørgensen. Can you trust a single data source exploratory software engineering case study? *Service-oriented science*, 7(1):9–26, 2002.

[13] Kathleen M Eisenhardt. Building theories from case study research. *Academy of management review*, 14(4):532–550, 1989.

[14] Per Runeson, Martin Host, Austen Rainer, and Bjorn Regnell. *Case study research in software engineering: Guidelines and examples*. John Wiley & Sons, 2012.

[15] Robert E Stake. *The art of case study research*. Sage, 1995.

[16] Tom Huston. What is microservices architecture? `https://smartbear.com/learn/api-design/what-are-microservices/`. [Online; accessed 2016, December 12.].

[17] Cesare Pautasso, Olaf Zimmermann, and Frank Leymann. Restful web services vs. big'web services: making the right architectural decision. In *Proceedings of the 17th international conference on World Wide Web*, pages 805–814. ACM, 2008.

[18] John Erickson and Keng Siau. Web services, service-oriented computing, and service-oriented architecture: Separating hype from reality. *Journal of Database management*, 19(3):42, 2008.

[19] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. Microservices: yesterday, today, and tomorrow. *arXiv preprint arXiv:1606.04036*, 2016.

[20] M. Conway. Mel Conway's home page. `http://www.melconway.com/Home/Conways_Law.html`, 1968. [Online; accessed 2016, December 12.].

[21] Martin Fowler. Microservice prerequisites. `http://martinfowler.com/bliki/MicroservicePrerequisites.html`, 2014. [Online; accessed 2017, March 1.].

[22] Simon Johansson. Risks of testing microservices. Master's thesis, NADA, Kungliga Tekniska Högskolan, `http://www.nada.kth.se/~ann/exjobb/simon_johansson.pdf`, 2016. [Online; accessed 2016, December 12.].

[23] Martin Fowler. Microservices resource guide. `http://martinfowler.com/microservices/`. [Online; accessed 2016, December 12.].

[24] Martin Fowler. Monolithfirst. `http://martinfowler.com/bliki/MonolithFirst.html`. [Online; accessed 2016, December 12.].

[25] Martin Fowler. Microservicepremium. `http://martinfowler.com/bliki/MicroservicePremium.html`. [Online; accessed 2016, December 12.].

[26] Stefan Tilkov. Don't start with a monolith. `http://martinfowler.com/articles/dont-start-monolith.html`. [Online; accessed 2016, December 12.].

[27] Ian Foster. Service-oriented science. *Science*, 308(5723):814–817, 2005. [Online; accessed 2017, March 1.].

[28] Sourabh Sharma. *Mastering Microservices with Java.* Packt Publishing Ltd, 2016.

[29] Mary Poppendieck. Lean software development. `http://www.comp.dit.ie/dgordon/Courses/ResearchMethods/Countdown/7Wastes.pdf`. [Online; accessed 2017, March 1.].

[30] Fernando López. Agile concept. `http://backlog.fiware.org/guide/foundation.html`. [Online; accessed 2017, March 1.].

[31] Henrik Kniberg and Mattias Skarin. *Kanban and Scrum-making the most of both.* Lulu. com, 2010.

[32] Jez Humble and David Farley. *Continuous delivery: reliable software releases through build, test, and deployment automation.* Pearson Education, 2010.

[33] John Fink. Docker: a software as a service, operating system-level virtualization framework. *Code4Lib Journal*, 25, 2014.

[34] Lisa Crispin. Using the agile testing quadrants. http://lisacrispin.com/2011/11/08/using-the-agile-testing-quadrants/. [Online; accessed 2016, December 12.].

[35] Glenford J Myers, Corey Sandler, and Tom Badgett. *The Art of Software Testing*. John Wiley & Sons, 2011.

[36] Andi Scharfstein. Types of tests. https://www.cqse.eu/en/blog/junit3-migration/. [Online; accessed 2016, December 12.].

[37] Martin Fowler. Testpyramid. http://martinfowler.com/bliki/TestPyramid.html/. [Online; accessed 2016, December 12.].

[38] Toni Huttunen. Browser cross-origin timing attacks. Master's thesis, Aalto University, http://urn.fi/URN:NBN:fi:aalto-201612226248, 2016-12-12. [Online; accessed 2016, December 12.].

[39] Bach, James. Exploratory testing explained. https://people.eecs.ku.edu/~saiedian/Teaching/Fa07/814/Resources/exploratory-testing.pdf, 2003. [Online; accessed 2016, December 12.].

[40] David Drake. Tiered testing of microservices. http://dev9.com/article/2015/5/zgotxg9hajgch1mg0d2hgbfgsf46ho, 2015. [Online; accessed 2016, December 12.].

[41] Taylor Pickens. Testing strategies for microservices. https://www.credera.com/blog/technology-insights/open-source-technology-insights/testing-strategies-for-microservices/. [Online; accessed 2016, December 12.].

[42] Martin Fowler. Testing strategies in a microservice architecture. http://martinfowler.com/articles/microservice-testing/. [Online; accessed 2016, December 12.].

[43] Taylor Pickens. Testing strategies for microservices. https://www.credera.com/blog/technology-insights/open-source-technology-insights/testing-strategies-for-microservices/, 2015. [Online; accessed 2016, December 12.].

[44] Arvind Sundar. An insight into microservices testing strategies. https://www.infosys.com/it-services/validation-solutions/

white-papers/documents/microservices-testing-strategies.pdf, 2016. [Online; accessed 2016, December 12.].

[45] Martin Fowler. Testdouble. https://martinfowler.com/bliki/TestDouble.html. [Online; accessed 2016, December 12.].

[46] James Hughes. Micro service architecture. https://yobriefca.se/blog/2013/04/29/micro-service-architecture/, 2013. [Online; accessed 2016, December 12.].

[47] Simon Brown. Modularity and testability. http://www.codingthearchitecture.com/2014/10/01/modularity_and_testability.html, 2014. [Online; accessed 2016, December 12.].

[48] Ganesh B Regulwar and Vijay S Gulhane. Software testing practices. *International Journal of Computer Applications*, 1(2):1–7, 2010.

[49] William E Lewis. *Software testing and continuous quality improvement.* CRC press, 2016.

[50] Elfriede Dustin, Jeff Rashka, and John Paul. *Automated software testing: introduction, management, and performance.* Addison-Wesley Professional, 1999.

[51] David Talby, Arie Keren, Orit Hazzan, and Yael Dubinsky. Agile software testing in a large-scale project. *IEEE software*, 23(4):30–37, 2006.

[52] Mike Wacker. Just say no to more end-to-end tests. https://testing.googleblog.com/2015/04/just-say-no-to-more-end-to-end-tests.html. [Online; accessed 2017, March 1.].

[53] Arnon Rotem-Gal-Oz, Eric Bruno, and Udi Dahan. Soa patterns. http://www.rgoarchitects.com/Files/SOAPatterns/TheKnot.pdf, 2012. [Online; accessed 2017, March 1.].

[54] Juha Itkonen. *Empirical studies on exploratory software testing.* PhD thesis, Aalto University, Espoo Finland, 2011.

Appendix A

Appendix A

A.1 Technical interview template

The following is a questionnaire template for technical interviews held in research cases.

1. Interviewee
   (a) Could you tell me about your background (education, experience)?
   (b) What are your roles and tasks in the project?
   (c) When did you start at this project?
   (d) What expectations do you have for this research?
2. Project Background
   (a) How big is your project?
   (b) What is the project's schedule?
   (c) What kind of parties are involved in this project?
   (d) Could you roughly estimate how many people are working on this project?
   (e) Could you estimate how many users your service is going to have?
3. Process Model
   (a) What kind of iterative process are you using? i.e. scrum
   (b) Could you describe your work process from requirements to the product?

(c) How is your delivery chain structured?
(d) How fast are you able to get feedback from your deliveries? How long is your delivery cycle?
(e) Who determines your product's quality?

4. Microservices
   (a) How many microservices do you have?
   (b) How are these connected to each other? (i.e. through gateway or something else)
   (c) What technologies are you using? (programming languages, testing frameworks, build pipeline infrastructure)
   (d) How do you update your services? i.e. one by one or all at once
   (e) Do you have any stateful-services?
5. General Testing
   (a) What kind of testing processes are you using?
   (b) Who is responsible for testing?
   (c) How have you shared the responsibility of maintaining tests?
   (d) Have you noticed any problems with your testing practices? What would you improve?
   (e) What kind of testing methods are you using? When do you use these methods?
   (f) How acceptance testing is done? Are there some problems?
   (g) How is the testing progress followed?
6. Testing Environment
   (a) Have you integrated tests into your service deployment? What are these tests? How much do you trust on the test results?
   (b) How many manual tests do you implement weekly? What kind of tests are these?
   (c) How many tests do you automate weekly? What kind of tests are these?
   (d) What kind of test have you automated?
   (e) How do you rationalize automation of testing to your customer?

   (f) How much time does your testing and feedback process take?

7. Testing scope

   (a) Estimate how many Unit, Contract, Integration, and E2E test do you have? How many would you like to have? What is your service's total test coverage?
   (b) What technologies do you use to implement these tests?

8. Decisions behind testing

   (a) How did you select which service tests to write?
   (b) How did you select which E2E tests to write?
   (c) How do you handle test data management? Do you use pre-existing data in databases or use stub the results?
   (d) How do you face external relations of the service in integration testing?
   (e) Do your test execute the same process as the service? Are test doubles (mocks, dummies, fakes, spies, stubs) implemented inside the same service or used externally over the network? What is your opinion about that?

9. Testing with microservices

   (a) How do you see testing in microservices?
   (b) How handle similarities in microservice testing? i.e. services are very similar, do you write one test for both or tests for every service?
   (c) What is easy in microservice orientated testing, what is difficult? How have you noticed this during your project?
   (d) How would you improve your testing design in a microservice environment?
   (e) How are testing frameworks supporting your testing practices?

A.2 Project management interview template

The following is a questionnaire template is used in project manager's interviews:

1. Interviewee
   (a) Could you tell me about your background (education, experience)?
   (b) What are your roles and tasks in the project?
   (c) When did you start at this project?
   (d) What expectations do you have for this research?
2. Project Background
   (a) How big is your project?
   (b) What is the project's schedule?
   (c) What kind of parties are involved in this project?
   (d) Could you roughly estimate how many people are working on this project?
   (e) Could you estimate how many users your service is going to have?
3. Process Model
   (a) What kind of iterative process are you using? i.e. scrum
   (b) Could you describe your work process from requirements to the product?
   (c) How is your delivery chain structured?
   (d) How fast are you able to get feedback from your deliveries? How long is your delivery cycle?
   (e) Who determines your product's quality?
4. Project Maturity
   (a) How has your project's maturity evolved during the project?
   (b) How the project quality has evolved during the project?
5. General Testing
   (a) What kind of testing processes are you using?

(b) When do you start testing component?

(c) Who is responsible for testing?

(d) Have you noticed any problems with your testing practices? What would you improve?

(e) What kind of testing methods are you using? When do you use these methods?

(f) How acceptance testing is done? Are there some problems?

(g) How is the testing progress followed?

6. Testing Environment

   (a) How do you see continuous delivery pipeline?

   (b) What kind of tests should be automated and what have you automated?

   (c) Do you understand the testing needs of the developer team?

   (d) How does a project's release process affect to testing?

7. Decisions behind testing

   (a) How do you do UAT testing? What is difficult, what is easy?

   (b) Why does the project's differ from recommendations, or do they? What are the recommendations?

8. Testing with microservices

   (a) How do you see testing in microservices?

   (b) How does a microservice environment influence on testing?

   (c) How would you improve your testing design in a microservice environment? Does it differ from a monolithic architecture?