Implementing Consumer-Driven Contract Testing in a DevOps Pipeline

Priya Joy Kaviyil

M.Sc. in Computing in DevOps 2023



Department of Computing, ATU Donegal, Port Road, Letterkenny, Co. Donegal, Ireland.

Implementing Consumer-Driven Contract Testing in a DevOps Pipeline

Author: Priya Joy Kaviyil

Supervised by: Ruth Lennon

A thesis submitted in partial fulfilment of the requirements for the

Master of Science in Computing in DevOps

Submitted to Atlantic Technological University*Arna chur isteach chuig Ollscoil Teicneolaiochta an Atlantaigh* September 2023

# Declaration

I hereby certify that the material, which l now submit for assessment on the programmes of study leading to the award of Master of Science in Computing in DevOps, is entirely my own work and has not been taken from the work of others except to the extent that such work has been cited and acknowledged within the text of my own work. No portion of the work contained in this thesis has been submitted in support of an application for another degree or qualification to this or any other institution. I understand that it is my responsibility to ensure that I have adhered to ATU’s rules and regulations.

I hereby certify that the material on which I have relied on for the purpose of my assessment is not deemed as personal data under the GDPR Regulations. Personal data is any data from living people that can be identified. Any personal data used for the purpose of my assessment has been psudonymised and the data set and identifiers are not held by ATU. Alternatively, personal data has been anonymised in line with the Data Protection Commissioners Guidelines on Anonymisation.

I consent that my work will be held for the purposes of education assistance to future students and will be shared on the ATU Donegal (Computing) website (www.lyitcomputing.com) and Research THEA website (https://research.thea.ie/). I understand that documents once uploaded onto the website can be viewed throughout the world and not just in the Ireland. Consent can be withdrawn for the publishing of material online by emailing Thomas Dowling; Head of Department at thomas.dowling@lyit.ie to remove items from the ATU Donegal Computing website and by email emailing Denise McCaul; Systems Librarian at denise.mccaul@lyit.ie to remove items from the Research THEA website. Material will continue to appear in printed formats once published and as websites are public medium, ATU cannot guarantee that the material has not been saved or downloaded.

Signature of Candidate Date

# Acknowledgements

I would like to thank…(write only the week before submission)

# Abstract

Written here is no more than 250 words of summary of the problem, hypothesis and main conclusions. The abstract will entice people to read the rest of the document. Remember that this abstract may also be used as a descriptor for the storage of dissertation as necessary. Write only when all other sections are complete.

(no references in the abstract; strictly 250 words so that it works with the online repository)

# Acronyms

|  |  |  |
| --- | --- | --- |
| Acronym | Definition | Page |
| CDC | Consumer-Driven Contract | 4 |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

# Table of Contents

[1 Declaration iv](#_Toc138399623)

[2 Acknowledgements v](#_Toc138399624)

[3 Abstract vi](#_Toc138399625)

[4 Acronyms vii](#_Toc138399626)

[5 Table of Contents viii](#_Toc138399627)

[6 Table of Figures x](#_Toc138399628)

[7 Table of Tables x](#_Toc138399629)

[8 Table of Code Listings xi](#_Toc138399630)

[1. Introduction 1](#_Toc138399631)

[1.1. Purpose 1](#_Toc138399632)

[1.2. Background 2](#_Toc138399633)

[1.3. Problem Statement 3](#_Toc138399634)

[1.4. Research Question 3](#_Toc138399635)

[1.5. Scope and Limitations 4](#_Toc138399636)

[1.6. Report Outline 4](#_Toc138399637)

[2. Literature Review 7](#_Toc138399638)

[2.1. Microservices and Containers 7](#_Toc138399639)

[2.1.1 7](#_Toc138399640)

[2.2. Consumer-Driven Contract Testing 7](#_Toc138399641)

[2.2.1 7](#_Toc138399642)

[2.3. DevOps Pipeline 8](#_Toc138399643)

[2.4. Tables 9](#_Toc138399644)

[2.5. Code and Formulae 10](#_Toc138399645)

[2.6. Chapter Conclusions 11](#_Toc138399646)

[3. Design Introduction 12](#_Toc138399647)

[3.1. Design Considerations 12](#_Toc138399648)

[3.2. System Context Diagram 12](#_Toc138399649)

[3.3. Specific Technology/Concept 1 (can you use one of your keywords in this title?) 13](#_Toc138399650)

[3.3.1. Commercial Technology/narrow scope Example1 13](#_Toc138399651)

[3.3.2. Commercial Technology/narrow scope Example2 13](#_Toc138399652)

[3.4. Specific Technology/Concept 2 14](#_Toc138399653)

[3.4.1. Commercial Technology/narrow scope Example1 14](#_Toc138399654)

[3.4.2. Commercial Technology/narrow scope Example2 14](#_Toc138399655)

[3.5. Framework or other relevant title 14](#_Toc138399656)

[3.6. Chapter Conclusions 14](#_Toc138399657)

[4. Implementation 15](#_Toc138399658)

[4.1. Some Title with keyword from RQ 15](#_Toc138399659)

[4.2. Some Title with keyword from RQ 15](#_Toc138399660)

[4.3. Chapter Conclusions 15](#_Toc138399661)

[5. Test Strategy, Results and Analysis 16](#_Toc138399662)

[5.1. Test Strategy 16](#_Toc138399663)

[5.2. Test Type 1 (Name linked to keyword in RQ/Aims) 17](#_Toc138399664)

[5.3. Test Type 1 (Name linked to keyword in RQ/Aims) 17](#_Toc138399665)

[5.4. Chapter Conclusions 17](#_Toc138399666)

[6. Conclusions 18](#_Toc138399667)

[6.1. Conclusions on Theoretical Research 18](#_Toc138399668)

[6.2. Conclusions on Practical Research 18](#_Toc138399669)

[6.2.1. X Discussion (name linked to keyword 1 in RQ/Aim) 18](#_Toc138399670)

[6.2.2. X Discussion (name linked to keyword 2 in RQ/Aim) 19](#_Toc138399671)

[6.2.3. X Discussion (name linked to keyword 3 in RQ/Aim) 19](#_Toc138399672)

[6.2.4. Some other Technology Example Heading 19](#_Toc138399673)

[6.3. Limitations 19](#_Toc138399674)

[6.4. Chapter Conclusions 19](#_Toc138399675)

[9 References 20](#_Toc138399676)

[10 Appendices 21](#_Toc138399677)

[11 Appendix A: Code Listing Location 22](#_Toc138399678)

# Table of Figures

[Figure 1 Atlantic Technological University Donegal Logo 2](#_Toc101374228)

[Figure 2 ATU Donegal, Letterkenny 2](#_Toc101374229)

# Table of Tables

[Table 1. Table Formatting Guidelines 15](#_Toc429429027)

[Table 2 Second Sample Table 16](#_Toc429429028)

# Table of Code Listings

[Code Listing 1 MDBean Message Handling 16](#_Toc429428842)

# Introduction

Technology is currently witnessing a dual transition, shifting from monolithic architectures to microservice architectures and from traditional software development life cycle (SDLC) practices to DevOps. The two transitions are interconnected as both DevOps and microservice deployment stresses on continuous integration (CI) and continuous deployment (CD). With this industry shift, it is important to ensure the speed of release cycle as well as a reliable deployment for which correct testing is required. The goal of fast and reliable testing pipelines is to be able to release software often and regularly (Nagel n.d.). One testing method that has emerged as a testing method specifically designed for microservices is consumer-driven contract (CDC) testing. Consumer-Driven Contract testing is a methodology that enables the testing of communication between services in an isolated manner (‘Testing Microservices - Contract Tests’ 2023). In this dissertation, we study how integration of contract test framework into the DevOps pipeline by executing them within containers, maximize the effectiveness of continuous testing (CT) in a distributed system. For this we design a DevOps pipeline, that incorporates multiple microservices and leverages containerization technology orchestration and deployment. This design integrates contract testing frameworks, ensuring that the interactions between microservices are thoroughly validated based on defined contracts.

## Purpose

The purpose of this dissertation stems from the growing adoption of microservice architecture and the need to ensure its successful integration within DevOps pipelines. With the increasing complexity of distributed systems, the speed and reliability of software releases have become critical (Nagel n.d.). Consumer-Driven Contract testing offer a promising solution in this regard (‘Testing Microservices - Contract Tests’ 2023). Containers, as lightweight and isolated environments, offer an ideal solution for managing and deploying microservices (‘DevOps for Microservices - Creating Change Together’ 2021). By incorporating consumer-driven contract tests into the DevOps pipeline and executing them within containers, organizations can effectively validate service contracts and identify compatibility issues early on. This approach enables teams to effectively validate service contracts, detect compatibility issues early on, and improve the overall quality and reliability of distributed software systems. This research aims to provide insights and practical guidance on integrating consumer-driven contract tests to DevOps pipeline leveraging containers to enhance integration testing within DevOps practices.

## Background

DevOps represents a transformative shift in how organizations approach software development, deployment, and maintenance, encompassing the entire software development lifecycle (SDLC) as a more collaborative process (Wickramasinghe 2023). A microservice architecture is a software design approach that involves breaking down an application into a distributed collection of loosely coupled services (Dhaduk 2022). Containers provide a standardized and lightweight method for applications to seamlessly transition between environments, containing everything required to run the application within the container object itself. This includes the code, runtime, system tools, libraries, and dependencies (‘Microservices and Containers 101 - Learn all About Microservices’ 2023). Containers offer an efficient approach for deploying microservices, as they provide a consistent and isolated environment for each individual service (‘The Role of Containers in Your Microservice Architecture’ 2021). By combining a microservices framework with containers, organizations can create a highly scalable and distributed system. This combination enables the establishment of continuous integration/continuous delivery (CI/CD) pipelines for applications, facilitating rapid and seamless development, testing, and deployment processes (‘What Are Containerized Microservices? – DreamFactory Software- Blog’ 2023). Meanwhile, when dealing with microservices, the testing process becomes more complex due to the crucial role of communication within a microservice architecture. Fortunately, there is a solution in the form of consumer-driven contract testing, which addresses the challenges specific to microservices testing. Consumer-Driven Contract testing is an approach used to verify integrations between services, where a crucial aspect is the establishment of a contract between the consumer (an application or service) and the provider (another service) when the consumer utilizes the API provided by the provider. (Lehvä 2019). There are two popular frameworks for the contract test — Spring Cloud Contract (SCC) and PACT that provide a way for contract definition and automate the test process (Fong 2022). A blend of these emerging technologies, including DevOps practices, microservicess, containerization, and contract testing can have a significant impact on enabling fast and reliable software product releases (‘DevOps for Microservices - Creating Change Together’ 2021). By leveraging these technologies together, organizations can achieve enhanced scalability and maintainability in their software development and deployment processes. This integrated approach facilitates quicker iterations, smoother collaboration between teams, efficient resource utilization, and robust validation of microservice interactions, ultimately leading to accelerated and more dependable software releases.

## Problem Statement

With the rise of microservice architecture and the increasing emphasis on continuous delivery, there is a demand for ensuring the quality and reliability of software systems (‘How Contract Tests Improve the Quality of Your Distributed Systems’ 2023). However, traditional testing approaches often struggle to keep up with the speed and complexity of microservices.

The problem statement is:

Testing distributed components within a DevOps pipeline, validating the interaction between client and server, lacks a comprehensive approach that goes beyond unit testing. By implementing contract testing framework, consumers can define their expectations through contracts, enabling effective integration testing.

The objective of this dissertation is to effectively implement contract testing framework in DevOps pipeline, enabling organizations to validate service contracts, detect compatibility issues early on, and ensure the smooth and reliable deployment of microservices.

## Research Question

The research question considered in this research is:

How does the integration of contract testing frameworks, within a Container-based DevOps pipeline enhance the validation of microservice interactions?

In order to answer this research question three aims were identified. They are:

1. To explore and evaluate the viability with integration of contract testing frameworks within the designed DevOps pipeline.
2. To explore how validation of microservice interactions within containers can be carried out based on defined contracts
3. To assess the effectiveness and benefits of integrating consumer-driven contract testing within containers in terms of enhancing the quality, speed and reliability of microservice-based systems within the DevOps pipeline.

## Scope and Limitations

Outline what is or is not covered and why. This is just a short paragraph.

## Report Outline

The thesis includes a comprehensive literature review in chapter 2, which examines the relevant research on DevOps Pipeline, Microservices, Containerization, and Consumer-Driven Contract (CDC) Testing. Chapters 3 and 4 focus on the design and implementation of a Container-based DevOps pipeline that incorporates contract testing frameworks. These chapters address the research question that was introduced in chapter 1. Finally, in chapter 6, the thesis concludes by discussing the implemented solution's architecture and suggesting future areas of research and development in this domain.

In some sections of the thesis an image may be required. Any image utilised must be referred to within the main body of text. In Figure 1 the ATU Donegal logo can be seen with an appropriate caption.



Figure . Atlantic Technological University Donegal Logo

In order to provide a caption for an image, table or equation the item should be selected. The Microsoft References ribbon should be selected. From there the Insert Caption button should be selected. The label should be set to the most appropriate one. In this example the Figure label was selected. New labels can be created as necessary. Remember not to simply copy and paste from above. Instead insert the image into the file and select it. Add the caption as described.

Where an image does not have a clearly defined border, one should be added. Care should be taken to ensure that all details of images are clearly visible both when in print and when in electronic format. Careful selection of colours should be considered for this purpose. Images as shown in Figure 2 should always be referred to from the main text.



Figure . ATU Donegal, Letterkenny

All text highlighted in yellow is text that you have written and are ready for me to read.

All text highlighted in grey is text you have written as notes to yourself and is not yet ready for me to read.

All text highlighted in green is something that has been reviewed and needs to be rephrased.

All text that no longer has highlight is deemed good enough for now but will be reviewed again during the last edit.

# Literature Review

The chapters should all begin on a new page. Page numbers appear at the bottom right of each page. Page numbers appear from Chapter 1 and onwards. Appendices should be paginated using roman numerals (I, II, etc.).

* Bullets should be aligned with the text.
* Bulleted items should have one blank line above and below.
* If there are only 1 or 2 items a bulleted list is not required.

Any paragraph after the list, image or table should resume its normal position for the given header. When using any acronym such as Some Silly Acronym (SSA) it must be expanded on its first occurrence within the text. All acronyms should appear in an acronyms list preceding the main chapters.

## Microservices

Adopting a microservice architectural style introduces additional challenges in monitoring, debugging, performance management, and testing. (Bass *et al.* 2015)

A microservices architecture is a cloud-native architecture that aims to realize software systems as a package of small services. Each service is independently deployable on a potentially different platform and technological stack. It can run in its own process while communicating through lightweight mechanisms such as RESTful or RPC-based APIs—for example, Finagle. (REST stands for Representational State Transfer.)

Migrating monolithic architectures to microservices brings in many benefits. In particular, it provides adaptability to technological changes to avoid technology lock-in and, more important, reduced time to-market and better development team structuring around services.3 (Balalaie *et al.* 2016)

Microservices are gaining momentum across industries to facilitate agile delivery mechanisms for service-oriented architecture and to migrate function-oriented legacy architectures toward highly flexible service orientation. The International Data Corporation has forecasted that by 2021, 80 percent of application development on cloud platforms will be with microservices. (Larrucea *et al.* 2018)

In recent years, there has a tendency in the software engineering community towards cloud computing. Cloud platforms are gaining mainstream adoption as the preferred delivery and operating model for modern applications by several companies like Amazon, Microsoft and IBM [1]. The changing infrastructural circumstances lead to architectural styles that take advantage of the opportunities given by cloud infrastructures. An architectural style that has become more relevant in the last years and that allows taking advantage of the benefits obtained with cloud computing is the Microservices Architecture (MSA) [2].

Monolithic Architecture (MA), instead, was the traditional approach to software development, used in the past by large companies like Amazon and Ebay. In MA, the functions are encapsulated into a single application. Monolith application, if not complicated, have their own strength, for instance, easiness of development, testing and deployment. However, when the application tends to become more complicated, the monolith structure grows in size, becoming a large, hard to manage and scale piece of software [6]. (De Lauretis 2019)

In recent years there has been a paradigm shift in software architecture design from an on-premises model to a cloud native approach [1]. (Keni and Kak 2020).

Cloud computing is a rapidly growing paradigm which has evolved from having a monolithic to microservices architecture. The importance of cloud data centres has expanded dramatically in the previous decade, and they are now regarded as the backbone of the modern economy. Cloud based microservices architecture is incorporated by firms such as Netflix, Twitter, eBay, Amazon, Hailo, Groupon, and Zalando. (Saboor *et al.* 2022)

In a microservices architecture, most testing, packaging, and deployment tasks can be automated for each service. As each service resides in an independent DevOps pipeline, any issues in a single automated task do not affect the other services. (‘The Role of Containers in Your Microservice Architecture’ 2021)

In recent years, microservice architecture (MSA) has gained increasing popularity, especially for large scale web-services with high traffic rates. In MSA-based applications, the system is divided into small, independent microservices [1]. The individual services are loosely-coupled and communicate through platform-independent interfaces with [2]. Therefore, the individual microservices can be developed, tested, and deployed independently [3] which enables the services to evolve autonomously. The video-streaming platform Netflix was one of the pioneers in migrating its system from a monolithic architecture to MSA. Today Netflix’s platform is an enormous system of hundreds of microservices [4]. Another example is the technology company Uber which is famous for its ride-sharing service. Uber’s platform is a system of circa 2200 microservices [5]. The autonomy of the individual microservice is one of the key principles of MSA that has to be incorporated in the complete software development life-cycle. Without autonomy, microservices cannot be independently developed and released, and the system loses its flexibility and agility. The autonomy is reflected in the system by the loosely-coupling of the microservices as well as organizational (i.e., independent development teams per microservice). Nevertheless, to become part of the actual system, the individual microservices have to interact with each other through synchronous or asynchronous messaging. To validate that these interactions work properly, testing and, in particular, integration testing is essential. Integration testing, besides security- and data storage-related issues, is stated as one of the major challenges in MSA based applications in academic and grey literature [6], [7]. It is the result of the increased complexity caused by the distribution of the system’s functionality across services and the necessary, additional infrastructure for communication. The challenge with integration tests is also rooted in the aim to run the tests in isolation (i.e., without dependent services) while validating the correctness of the interactions with exact these services. Currently, different approaches are used to tackle this discrepancy. (Fischer 2021)

### 2.1.1. Containerization

deployment in the development environment is difficult. Although the application code is now in isolated services, developers must also deploy the dependent services to run the isolated services on their machines. (Balalaie *et al.* 2016)

As companies produce further and further microservices, their deployment surroundings are getting increasingly elaborate. Without proper configurations, a microservices road chart could snappily come unmaintainable. Containerization is another element that broadens and supplements microservices-based models. Bundling each help as a container picture further diminishes the intricacy while smoothing out the persistent conveyance pipeline. Administrations can go about as completely autonomous substances with every one of the conditions and necessities packaged inside the container. This makes the administration's framework rationalist and reusable while permitting them to interface with some other framework. (Amrit *et al.* n.d.)

The goal of building is to create something suitable for deployment. Lightweight containers are a new phenomenon. Like VM images, lightweight containers can contain all libraries and other pieces of software necessary to run the application, while retaining isolation of processes, rights, files, and so forth. The emergence of lightweight containers and image management tools is helping developers to deploy into small-scale production-like environments more easily for testing. (Bass *et al.* 2015)

The Microservice architecture is supported by containers as each microservice can be deployed without interfering with other micro services. Containers provide suitable environment for service deployment in terms of speed, isolation and ease of deployment of new versions.[20] (Singh and Singh 2016)

containerization [5] is a form of virtualization that attempts to achieve resource isolation with minimal overhead by sharing the kernel with the host OS. The use of containerization allows us to implement many of the desirable features associated with cloud platforms– elasticity, reliability, and ease of management. Microservices can be conveniently packaged into containers that are then deployed onto physical hardware, thus ensuring a consistent software execution environment from the developer to the consumer. The use of containerization also allows for inherent scalability and creates a redundancy mechanism for machine failure as container instances can be added and removed on demand. (Keni and Kak 2020)

Containerization is another factor that extends and complements microservices-based architectures. Packaging each service as a container image further reduces the complexity while streamlining the continuous delivery pipeline. Services can act as fully independent entities with all the dependencies and requirements bundled within the container. This makes the services system-agnostic and reusable while allowing them to interact with any other system. (C). In a microservices architecture, most testing, packaging, and deployment tasks can be automated for each service. As each service resides in an independent DevOps pipeline, any issues in a single automated task do not affect the other services (‘The Role of Containers in Your Microservice Architecture’ 2021)

## Microservice Architecture Enables DevOps:

Not only are software vendors (for example, IBM and Microsoft) using microservices and DevOps practices, but also content providers (for example, Net¬flix and the BBC) have adopted and are using them. In addition, Google Trends reveals that both DevOps and microservices are growing concepts, with an equal rate of growth after 2014 (see Figure 1). Although DevOps can also be applied to monolithic software systems, microservices enable effective implementation of DevOps by promoting the importance of small teams. (Balalaie *et al.* 2016)

The DevOps goal of minimizing coordination among various teams can be achieved by using a microservice architectural style where the coordination mechanism, the resource management decisions, and the mapping of architectural elements are all specified by the architecture and hence require minimal inter-team coordination. (Bass *et al.* 2015)

DevOps practices can be applied to monolithic applications, but microservices increase the importance of small groups to enable practical DevOps execution. The microservices structure is a cloud-native structure that creates a product framework that can be a small management package. All of these can be freely deployed in different layers and machine stacks, jogging cycles through lightweight systems. (Balalaie *et al.* 2016)

As found in the literature, it can also be discovered from the interviews that it’s very hard doing DevOps with monoliths and microservices seem to be a part of a solution to allow it. (Jokinen 2020)

Microservices Architecture (MSA) is a cloud-native architectural style, which is inspired by Service-Oriented Architecture (SOA). Typically, microservices are organized as a suite of small granular services that can be implemented (developed, tested, and deployed) on different platforms through multiple technological stacks (Larrucea et al., 2018). Each service of the MSA runs on its own process and communicate with each other through, e.g., RESTful or RPC-based APIs (Balalaie et al., 2016). (D&M)

MSA has become popular in industry because of its benefits, such as availability, flexibility, scalability, loose coupling, and high velocity (Hasselbring and Steinacker, 2017) According to the International Data Corporation (IDC), by the end of 2021, 80% of cloud-based applications will be developed using by MSA (Larrucea et al., 2018). It is also argued that the worldwide DevOps market would grow to $5.6 billion in 2021 (Elliot et al., 2018). Another published report reveals that organizations may adopt MSA for different purposes (Yousif, 2016), for example, to gain agility (82%), to improve organization performance (57%), and scalability (78%). This report also shows that the motivation behind implementing MSA in 47% of organizations was DevOps (LightStep, 2018). (D&M)

DevOps is a set of practices for developing, testing, and deploying software quickly and reliably by promoting collaboration between the developers, testers, and operators (Yousif, 2016). DevOps practices aim “to decrease the time between changing a system and transforming that change into production environment” (Balalaie et al., 2016). Many practitioners and researchers advocate that MSA has a natural progression of embracing DevOps (Gauna, 2018, Humble and Farley, 2010). DevOps brings additional productivity with MSA through using tools chain and a fast feedback mechanism (Stahl et al., 2017). (D&M)

(Waseem *et al.* 2020)

Connecting microservices with DevOps will increase software engineering’s impact and benefits.[4]. (Larrucea *et al.* 2018)

In a microservices architecture, most testing, packaging, and deployment tasks can be automated for each service. As each service resides in an independent DevOps pipeline, any issues in a single automated task do not affect the other services. Microservices architecture is tailor-made for DevOps with its services-based approach that allows organizations to break down the application into smaller services. This enables delivery teams to tackle individual services as separate entities—ultimately simplifying the development, testing, and deployment. (This doesn’t mean microservices should be used for every application, however. They do come with certain challenges.) The role microservices plays in DevOps includes streamlining the DevOps process and increasing productivity and quality of the application while moving developments to a flexible architecture. This leads to the development of cloud-native applications that are capable of fulfilling any user demand. (‘The Role of Containers in Your Microservice Architecture’ 2021)

To be able to deliver microservices fast and reliably the continuous integration (CI) and continuous deployment (CD) pipeline have to be efficient, of good quality and optimized. This way we can ensure the maximum speed of reliable updates. To get the maximum out of the most beneficial aspect of microservices, the fast release cycle, we need to have fast testing. With the introduction of microservices in an architecture the amount of interfaces that can be reached is significantly higher. Most, if not all, of the communication between microservices rely on these interfaces. This proves to be a new problem in testing. These interfaces have to be tested reliably and quick. (Nagel n.d.)

## Consumer-Driven Contract Testing

service contracts are critical. Changing so many services that expose their contracts only to each other could be error-prone. Even a small change in the contracts can break part or even all of the system. One possible solution is service versioning, but it could make deploying each service even more complex. So, people usually don’t recommend service versioning for microservices. Consumer-driven contracts could help greatly in this regard because the team responsible for a service can be con¬ dent that most of its customers are satisfied with the service. (Balalaie *et al.* 2016)

Adopting a microservice architectural style introduces additional challenges in monitoring, debugging, performance management, and testing. integration and end-to-end testing with all or most microservices should be done judiciously. It can be expensive to run these tests frequently due to the involvement of a potentially large number of microservices and realistic external resources. A testing practice called Consumer Driven Contract (CDC) can be used to alleviate the problem. That is, the test cases for testing a microservice are decided and even co-owned by all the consumers of that microservice. Any changes to the CDC test cases need to be agreed on by both the consumers and the developers of the microservice. Running the CDC test cases, as a form of integration testing, is less expensive than running end-to-end test cases. If CDC is practiced properly, confidence in the microservice can be high without running many end-to-end test cases. CDC serves as a method of coordination and has implications on how user stories of a microservice should be made up and evolve over time. Consumers and microservice developers collectively make up and own the user stories. CDC definition becomes a function of the allocation of functionality to the microservice, is managed by the service owner as a portion of the coordination that defines the next iteration, and, consequently, does not delay the progress of the current iteration. (Bass *et al.* 2015)

Challenge in Testing: With teams releasing updates at their own speed, it tends to be trying to plan strong end-to-end testing, particularly when services have dependencies on different services. In addition, running a full creation cluster can be costly, so it's impossible that each team will run its own full cluster at creation scales, only for testing. (Amrit *et al.* n.d.)

Consumer-Driven Contract testing is a way to test integrations between services. The main idea is that when an application or a service (consumer) consumes an API provided by another service (provider) a contract is formed between them. The contract contains information about how the consumer calls the provider and what the consumer needs from the responses. The contract can then be used to test both sides of the integration separately. (CDC)

The testing method is said to be useful when testing integration-heavy systems such as systems based on microservice architecture. (Lehvä 2019)

Therefore the research question of the thesis is: "with a focus on integrations, is Consumer-Driven Contract testing a viable addition to a testing strategy used to test a system based on microservice architecture, and if so, why?"

In a distributed microservices environment with tens of services interacting with each other, it make sense to start building the services - and create test cases incorporating the dependent services - at the time when these dependent services are not built yet but we know how their APIs looks like. (CDC)

Right at the time when the architecture team has completed defining the API's schema for a service by providing a Swagger file, this schema - that is a contract the service will have to respect at runtime - can be used to create a "mock" service that will run in a local "dockerized" environment where test cases can be written and executed against this service.

the test cases can be done in isolation, without having to spawn the whole framework: one service can test its dependency on the immediate service and only on this one. Another advantage is the fact that an end-to-end integration test case it is most likely easier to get executed and passed when all the individual integration test cases have been performed successfully for a while.

(‘(26) Contract testing - part one - docker containers | LinkedIn’ 2023)

Use mocks and stubs wherever possible: A lot of testers might frown on this, as the thought of using lots of mocks and stubs can be seen as avoiding the true integrated behaviour of an application. This is true for end-to-end testing which you still want to automate, but not ideal for pipeline execution. Not only does it slow down pipeline execution, but creates flakiness in your test results as external functions are not operational or out of sync with your changes. The best way to ensure that your test results are more reliable, along with allowing you to take greater control of your testing effort and improve coverage is to build mocking into your test framework and rely on stubs to intercept complex data patterns that an external function to do it for you.

A good DevOps testing strategy requires a solid base of unit tests to provide most of the coverage with mocking to help drive the rest of the automation effort up, leaving only the need for a few end-to-end automated tests to ensure everything works in order and allow your team to take confidence that the pipeline tests will successfully deliver on their quality needs. (‘Effective Test Automation Approaches for Modern CI/CD Pipelines’ 2023)

Testing monolithic applications is relatively easy. We implement unit, integration, and probably end-to-end tests, and try to keep the testing pyramid as perfectly shaped as possible. But when it comes to testing microservices, things complicate a little. The main difference stands for communication which is essential in a microservice architecture. Luckily, for every challenge, we have a pattern and, in this case, it’s contract testing. Contract testing is a concept that allows testing communication (both synchronous and asynchronous) between the services in isolation. The main idea behind it is a contract. This is a place where communication rules are agreed upon and written down by the sides. The contract is a single point where the contract is defined. The parties don’t have to synchronize when the contract changes. Both producer’s and consumer’s tests are based on this contract. It ensures that if the tests pass, the services get along in production. (‘Testing Microservices - Contract Tests’ 2023)

The challenge with integration tests is also rooted in the aim to run the tests in isolation (i.e., without dependent services) while validating the correctness of the interactions with exact these services. Currently, different approaches are used to tackle this discrepancy. (M) (IT)(CDC). A third approach is consumer-driven contract (CDC) testing. CDC testing is described as a potential solution [10] or addition [11], [12] to handle the challenges of integration testing. In CDC tests, the consumer of a downstream microservice states which responses it expects for certain requests from the provider (i.e., its API). The providing service verifies that it can fulfill these expectations and both parties enter into a contract. Provider and consumer use this contract to test their interfaces independently.

Consumer-Driven Contract Testing: Contract testing is based on the paradigm of creating contracts that determine the communication between services. By testing against contracts, the tests verify that a consumer (i.e., a microservice that sends a request) and a provider (i.e., the microservice that responds) can integrate and successfully communicate. However, they do not aim at testing the functionality or business logic of the respective services. The contracts can be driven by the consumer or the provider, depending on which side has the power to set the rules. Nowadays, the most common way is consumer-driven contracts where the consumer states its expectations to the provider. The provider confirms that it can fulfill these expectations and they become contractual. The contract is accessible to both parties for independent testing. Changes in the contract have to be communicated between the involved parties [11]. CDC tests leave the scope of the individual service as they require the communication and collaboration between the interacting microservices’ teams, but the actual testing can be conducted in isolation. (Fischer 2021)

. With the introduction of microservices in an architecture the amount of interfaces that can be reached is significantly higher. Most, if not all, of the communication between microservices rely on these interfaces. This proves to be a new problem in testing. These interfaces have to be tested reliably and quick. Consumer-driven contract (CDC) tests apply exactly at these interfaces between microservices without needing as much time as end-to-end tests. One of the reasons consumer-driven contract testing is deployed in testing pipelines in the context of microservice structure is their fast execution with reliable enough results to catch many errors. In the best case, before the end-to-end tests are even run. This allows the testing department to run less end-to-end tests and reduce the intensity and necessity of end-to-end tests. (Nagel n.d.)

Testing has a prominent role in revealing faults in systems that are based on Microservices-Based Architectures (MSAs). A central discussion point in MSAs is the granularity of services, that are often in different levels of abstraction [1]. Similarly, the granularity of tests in MSAs is reflected in different test types [2]. The individuality of microservices [3], [4] means that unit tests are part of the software development lifecycle [5], but the broader architectural scope of MSAs includes also other types of tests (e.g., for integration) [6]. To become part of the overall system, the interactions of individual microservices need to be validated and thus, integration-level, component-level and system-level testing are also crucial [7]. For instance, writing good integration test cases is ranked as the most important skill to sufficiently test microservices [8]. An emerging approach of integration testing in MSAs is Consumer-Driven-Contract testing. (Ayas *et al.* 2022)

Consumer-Driven Contract testing [9] is a way to test integrations between services and ensure that all the integrations are still working after new changes have been introduced to the system. The main idea is that when an application or a service (consumer) consumes an API provided by another service (provider), a contract is formed between them. The contract contains information about how the consumer calls the provider and what is being used from the responses. As long as both of the parties obey the contract, they can both use it as a basis to verify their sides of the integration. The consumer can use it to mock the provider in its tests. The provider, on the other hand, can use it to replay the consumer requests against its API. This way the provider can verify that the generated responses match the expectations set by the consumer. With consumer-driven contracts, the provider is always aware of all of its consumers. This comes as a side product when all the consumers deliver their contracts to the provider instead of consumers accepting the contracts offered by the provider.

the consumer-driven contract tests can replace integration tests as they caught all the defects from the integrations that were implemented in the case study. In that light, it can be safely said that consumer-driven contract testing is a viable addition to testing strategies used to test integration-heavy systems, especially those based on microservices. (CDC) (Lehvä *et al.* 2019)

--------------------------------------------------------------------------------------------------------------------------

Begin with the wider area of interest and quickly move to the narrow field of study. Do not go back to the beginning of the Internet or the first attempts to create a database management system. Consider your audience. Talk about the area rather than individual tools. Provide an analysis of the broad area and focus down to the narrow area under research. When writing chapter 3 there should be a clear link from the broad topic area in chapter 2 to the specific tools selected in chapter 3.

The text of any given chapter may refer to an interesting idea presented in another book, paper, journal or whitepaper. On-line sources should not normally constitute more than 50% of your references. ALL of your references must be peer-reviewed or small percentage of whitepapers. Further details on this will be given in the Research Workshop and may be obtained from your supervisor. This is supported by research (Bloggs, 2022) carried out how best to reference. The idea is succinctly expressed by Murphy:

“A reference in a thesis should be of the previously demonstrated Harvard Style.” (Murphy, 2021a)

Notice that the quote has indentations on both sides and is surrounded by quotes. If the quote abstracts only part of a sentence double dots should be placed before or after to show where there is missing text. Further, where additions for clarification are used in the text square brackets should be used. According to Murphy:

“A reference in a thesis should be of… Harvard Style.” (Murphy, 2021b)

Quotes should be short. Long quotes are not considered acceptable. Neither should there be too many quotes. It is better to provide a critique of what the person has stated.

While the example above is provided for demonstration purposes it is obviously not a good idea to provide the same quote twice so for the purposes of this example, we will assume this quote was taken from a different book by the same author. Further the general use of a single or a small number of sources multiple times is referred to as ‘over-reliance on a source’ and is deemed plagiarism.

## DevOps Pipeline

DevOps is a set of practices that aim to decrease the time between changing a system and transferring that change to the production environment. However, they also insist on maintaining software quality in terms of both code and the delivery mechanism. Any technique that enables these goals is considered a DevOps practice. [1] [2] (Balalaie et al. 2016).

The structure of microservices arose from the average relevance of DevOps philosophies born in companies such as Amazon, Facebook, Google, Netflix, and SoundCloud. Similar to agile techniques, DevOps divides programming applications into particularly discreet parts or modules to speed up processing and improve quality. DevOps signs are non-stop practices, including regular shuffling, non-stop testing, continuous transmission, and continuous broadcasting. By combining these practices, you can develop programming objects and programming-related samples without interruption. Microservices allow DevOps teams to develop independent features in parallel. Instead of moving code from one specialist to another (for example, from development to testing to production), cross-functional teams build, test, release, monitor, and maintain applications together. (Amrit *et al.* n.d.)

DevOps focuses on the goals, rather than the means. DevOps is a set of practices intended to reduce the time between committing a change to a system and the change being placed into norm. The longer it takes a release to get to market, the less advantage will accrue from whatever features or quality improvements led to the release. Ideally, we want to release in a continuous manner. This is often termed continuous delivery or continuous deployment.al production, while ensuring high quality. (Bass *et al.* 2015)

Unit, system, acceptance and regression test automation are the main automation expertise that may further promote continuous integration, continuous delivery, continuous deployment and continuous monitoring. (Agrawal and Rawat 2019)

### 2.3.1. Build and Test

Adopting a microservice architectural style introduces additional challenges in monitoring, debugging, performance management, and testing.

The build and integration tests are performed by a continuous integration (CI) server. The input to this server should be scripts that can be invoked by a single command. In other words, the only input from an operator or the CI server to create a build is the command “build”; the rest of the action of the continuous integration server is controlled by the scripts. This practice ensures that the build is repeatable and traceable. Repeatability is achieved because the scripts can be rerun, and traceability is achieved because the scripts can be examined to determine the origin of the various pieces that were integrated together.

Build is the process of creating an executable artifact from input such as source code and configuration. As such, it primarily consists of compiling source code (if you are working with compiled languages) and packaging all files that are required for execution (e.g., the executables from the code, interpretable files like HTML, JavaScript, etc.). Once the build is complete, a set of automated tests are executed that test whether the integration with other parts of the system uncovers any errors.

The goal of building is to create something suitable for deployment.

Integration testing is the step in which the built executable artifact is tested. The environment includes connections to external services, such as a surrogate database. Including other services requires mechanisms to distinguish between production and test requests, so that running a test does not trigger any actual transactions, such as production, shipment, or payment. This distinction can be achieved by providing mock services, by using a test version provided by the owner of the service, or—if dealing with test-aware components—by marking test messages as such by using mechanisms built into the protocol used to communicate with that service. If mock versions of services are used, it is good practice to separate the test network from the real services (e.g., by firewall rules) to make absolutely sure no actual requests are sent by running the tests.

Deploying and rolling back microservices in new environments independently without an excessively long-running suite of integration tests. Microservices should be deployed independently of each other, yet changes need to be compatible with other services in the same environment. Integration tests are currently being used to verify this, but as the number of microservices grows this could become a bottleneck. Investigation into concepts such as consumer-driven contracts, maintenance of metadata around test runs, and reliability of specific microservice version combinations is being done, but this is still an outstanding problem.

(Bass *et al.* 2015)

Continuous testing is a process of testing early, testing often, testing everywhere, and automate. One of the challenges in CT is the environment of heterogeneity, which will never reflect actual production environment and application architecture. (Agrawal and Rawat 2019)

Continuous testing identifies integration issues much earlier in the life cycle; makes defect resolution cheaper, faster; and frees tester’s precious time for exploratory testing and value-added test activities. (Angara *et al.* 2018)

Nevertheless, to become part of the actual system, the individual microservices have to interact with each other through synchronous or asynchronous messaging. To validate that these interactions work properly, testing and, in particular, integration testing is essential. Integration testing, besides security- and data storage-related issues, is stated as one of the major challenges in MSA based applications in academic and grey literature [6], [7]. It is the result of the increased complexity caused by the distribution of the system’s functionality across services and the necessary, additional infrastructure for communication. The challenge with integration tests is also rooted in the aim to run the tests in isolation (i.e., without dependent services) while validating the correctness of the interactions with exact these services. Currently, different approaches are used to tackle this discrepancy. (Fischer 2021)

Challenge in Testing: With teams releasing updates at their own speed, it tends to be trying to plan strong end-to-end testing, particularly when services have dependencies on different services. In addition, running a full creation cluster can be costly, so it's impossible that each team will run its own full cluster at creation scales, only for testing. (Amrit *et al.* n.d.)

## Tables

The text of any chapter may include tabular data. To aid legibility some simple guidelines should be adhered to. Refer to Table 2.1.

Table .. Table Formatting Guidelines

|  |  |
| --- | --- |
| Format | Description |
| Size | The table should be able to fit into one page and should not overrun. |
| Margins | The table should not extend past the normal margins of the page |
| Colour | Colour may be used but consideration should be given to both on screen display and printed display. |
| Design | Simple designs are best. At all times consider that the information in the table is more important than the ‘flashy’ design. |

The title for an image or code must appear directly underneath and on the same page. If this is not possible then move the item within the text to ensure that the caption remains with the item. The title for a table must appear directly above the table.

Refer to Table 2.2 for the second short table sample.

Table .. Short Table Example

|  |  |
| --- | --- |
| Format | Description |
| Size | The table should be able to fit into one page and should not overrun. |

Tables should never run over from one page to another. Move or split the tables as needed. Don’t forget to refer to the table such as Table 2.2, or figure from within the main text.

## Code and Formulae

Where code requires listing within the text it should be treated as an image in that it is sectioned off with a border and has a caption directly underneath. Refer to Code Listing 1.1 below. The full code listing should be included in electronic format through a repository. It should not be included in printed format in the main document. Code listings, tables and images should not run over multiple pages.

…

name: CI

# Run this workflow every time a commit gets pushed to main or a pull request gets opened against main

on:

push:

branches:

- main

pull\_request:

branches:

- main

jobs:

build:

name: Call Azure Pipeline

runs-on: ubuntu-latest

steps:

- name: Azure Pipelines Action

uses: Azure/pipelines@v1

with:

azure-devops-project-url: https://dev.azure.com/organization/project-name

azure-pipeline-name: 'My Pipeline'

Code Listing 2.1 GitHub YAML calling Azure

Notice that only minimal commenting is provided within the text. The code is shown in Cambria, 10 point. This reduces the overall text size and clearly distinguishes it from the main text.

If a single line of formula is required it can be referred to within the text as formula (Equation 1) for example with the formula example shown slightly indented and with the formula number to the far right.

Equation 1

A single line space above and below the formula (Tsiolkovsky, 2000) also aids legibility. Note also that the font size is increased by one point. A further item to note is that equations may also be referenced.

## Chapter Conclusions

Make sure that each chapter has a one paragraph chapter conclusions. Make sure this paragraph somehow relates to the research question. Make sure that the last line in the summary introduces the next chapter by relating it to the last chapter.

# Design Introduction

The design of the system should be presented succinctly. Justification for the selection of the design elements should be considered. The titles of the chapters are only samples. However, it is important that the background to the problem, the review of existing work in the area, design, results and conclusions are all discussed in some format.

The selection of tools and techniques are often best served by providing a high-level diagram early in this chapter. This helps the reader to understand where each element of the system is applied. In particular, it is often helpful to use the broad technology terminology from chapter 2 in this high-level diagram at the beginning of the chapter. At the end of the chapter a similar diagram with the exact technologies chosen can then be presented. This is not a requirement but may be considered helpful.

If the dissertation is focused on analytics rather then technology describe the process including, data ingress, etl, analytics, etc.

## Design Considerations

Describe the frameworks, methods, laws or other considerations prior to describing the system.

## System Context Diagram

Describe the solution very briefly by providing a simplified graphical image of the system. This is a high level diagram used to set the context for the technologies you will talk about. This should not be a UML diagram but something that can be easily understood.

In a cyberpsychology example a method may be shown as a flow chart.

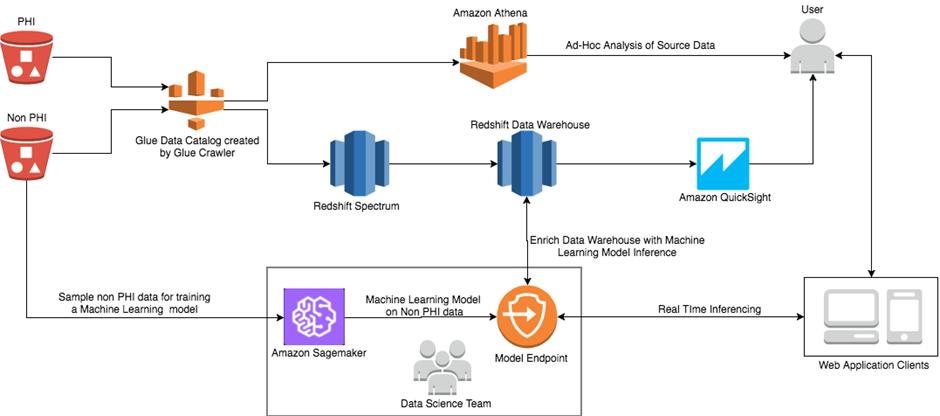


Figure . Top Level Context/System Diagram

## Specific Technology/Concept 1 (can you use one of your keywords in this title?)

Describe commercial implementations available for the Broad Technology in Chapter 2. This could also be a concept you wish to investigate. Mention the options. Mention how you evaluated them. What was your final choice and why. For example, when speaking of technology this time talk about Docker as an implementation of containers. Do this for each of the technologies, algorithms or concepts selected.

### Commercial Technology/narrow scope Example1

Why is this a good example? How do you evaluate the best? Why did you or did you not pick this technology or concept. How is the scope narrowed by using this example.

### Commercial Technology/narrow scope Example2

Why is this a good example? How do you evaluate the best? Why did you or did you not pick this technology or concept. How is the scope narrowed by using this example.

## Specific Technology/Concept 2

Describe commercial implementations available for the Broad Technology in Chapter 2. This could also be a concept you wish to investigate. Mention the options. Mention how you evaluated them. What was your final choice and why. For example, when speaking of technology this time talk about Docker as an implementation of containers. Do this for each of the technologies, algorithms or concepts selected.

### Commercial Technology/narrow scope Example1

Why is this a good example? How do you evaluate the best? Why did you or did you not pick this technology or concept. How is the scope narrowed by using this example.

### Commercial Technology/narrow scope Example2

Why is this a good example? How do you evaluate the best? Why did you or did you not pick this technology or concept. How is the scope narrowed by using this example.

## Framework or other relevant title

Make sure that the broad areas include the key words mentioned in the research question and in the individual aims.

## Chapter Conclusions

Provide specific conclusions relating back to key words mentioned in the research questions and in the individual aims.

# Implementation

The implementation chapter may describe a case study, interviews, observations, combinations of technologies or code structure depending on the type of dissertation you wish to carry out.

The implementation should be presented succinctly. Clearly define how the system/algorithm/process works in practice showing the most important parts and how they relate to each other. Describe your specific contribution making it clear what is yours and what api’s you used from elsewhere.

Show snips of code/configuration/questionnaire or similar. This may include screengrabs of your implementation if a software or hardware artifact is involved. Relate the implementation elements to the sections described in the literature survey. Be very careful to end each second level heading with a statement showing how the section relates to the research question.

## Some Title with keyword from RQ

Some text.

## Some Title with keyword from RQ

Some text.

## Chapter Conclusions

Provide specific conclusions relating back to key words mentioned in the research questions and in the individual aims.

# Test Strategy, Results and Analysis

Test strategy and how it aligns to the problem statement and hypothesis should be provided. This can be integrated with the results chapter or can be separate. This depends on the type of dissertation. Results can be quantitative or qualitative. Discuss this using appropriate, high level terminology.

Results of tests carried out should be presented. Where extensive testing is carried out the majority of the results should be moved to the appendix. Each graph should include values on the axes and should be appropriately labelled. All graphs/tables must be referred to and explained within the main text. Refer to all graphs in the main text. See Figure 5.1

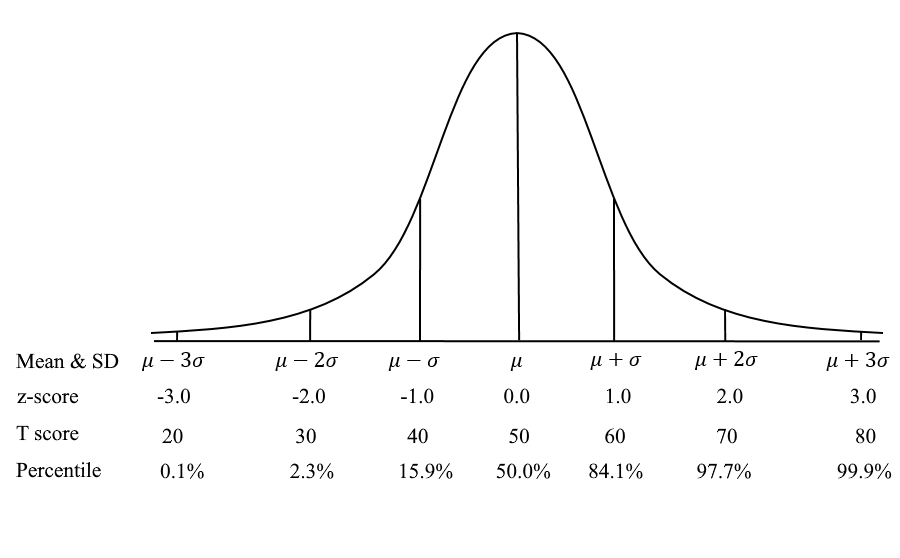


Figure . A graph showing Z score, t score and percentile information

## Test Strategy

Describe the type of tests that were selected as part of the test strategy. Why are these types of tests considered important?

Remember that these graphs cannot simply be the sum or average of figures obtained. You need to carry out detailed analysis and that takes time.

## Test Type 1 (Name linked to keyword in RQ/Aims)

Describe the type of test carried out. Say why it is an important type of test. What were the inputs and outputs. State how many times the test was run or the confidence factor for the test. State the facts drawn from the tests but do not give an opinion on the importance of the result. Repeat for each test type.

## Test Type 1 (Name linked to keyword in RQ/Aims)

Ensure that there is a dozen or so test types each with appropriate graphs or similar demonstration of practical tests. Include tables, graphs and screen shots as needed. Average is not sufficient as a mechanism to evaluate data.

## Chapter Conclusions

Describe how the test strategy included all appropriate test types as could be covered in the time allowed. Describe general observations on the results.

# Conclusions

Each week write at least one or two lines of notes to yourself as to what you might say in the conclusions chapter. You will forget when you get to the end, so it is really important to write this as you go along.

Conclusions should not introduce new material. The conclusions section should present your final thoughts in an organised manner. Try to organise your conclusions into logical paragraphs where points surround a given topic.

It could be argued that references should not appear in this chapter as that would indicate new material other than your own work. Similarly, it could be argued that images should not appear in the conclusions chapter.

Note that the number of pages in the conclusions should generally balance with the number of pages in the introduction. This is because the introduction is created to introduce the problem whilst the conclusions determine the validity of the hypothesis

## Conclusions on Theoretical Research

Try to organise your thoughts under headings as that will help you focus. Jot down your thoughts as you go along rather than waiting until the end to write the conclusions. You may find that you will forget a lot of the important conclusions if you don’t do this. The last statement on each of these topics should relate directly to the research question.

For example:

Bloggs, et al. showed that…indicating a need for research in this area…link to RQ

Smith, et al. described an experiment where…However, they did not consider…link to RQ

From the survey of existing literature has been found that a gap exists where….Link to RQ

## Conclusions on Practical Research

Focus on your practical element to test our hypothesis here. Focus strongly on why this is new.

### X Discussion (name linked to keyword 1 in RQ/Aim)

Discussion on how the theory from chapter 2 relates to the practical work in the remaining chapters. Show how your work agreed with, disagreed with or extended what was already known.

### X Discussion (name linked to keyword 2 in RQ/Aim)

Discussion on how the theory from chapter 2 relates to the practical work in the remaining chapters. Show how your work agreed with, disagreed with or extended what was already known.

### X Discussion (name linked to keyword 3 in RQ/Aim)

Discussion on how the theory from chapter 2 relates to the practical work in the remaining chapters. Show how your work agreed with, disagreed with or extended what was already known.

### Some other Technology Example Heading

Always relate the theory to the practical element when discussing conclusions.

## Limitations

Very short recap on limitations outlining any new discoveries on limitations. How would limitations be overcome if you had more time. Don’t just list the limits. Keep this very short as you don’t want the work to seem unfinished.

## Chapter Conclusions

Discussion on why this is so good there is some further work to be done. Relate your answer to why this can be applied in other domains. Relate to social/real world examples.

# References

(26) Contract Testing - Part One - Docker Containers | LinkedIn [online] (2023) available: https://www.linkedin.com/pulse/contract-testing-1-eugen-frunza/?trk=pulse-article\_more-articles\_related-content-card [accessed 12 Jun 2023].

Agrawal, P. and Rawat, N. (2019) ‘Devops, A New Approach To Cloud Development & Testing’, in *2019 International Conference on Issues and Challenges in Intelligent Computing Techniques (ICICT)*, Presented at the 2019 International Conference on Issues and Challenges in Intelligent Computing Techniques (ICICT), GHAZIABAD, India: IEEE, 1–4, available: https://doi.org/10.1109/ICICT46931.2019.8977662.

Amrit, A., Akhil, P., Pranjal, Raj N, R., and Shylaja, B. (n.d.) ‘Microservices Evolving DevOps Pipeline’, *International Research Journal of Modernization in Engineering Technology and Science*.

Angara, J., Gutta, S., and Prasad, S. (2018) ‘DevOps with Continuous Testing Architecture and Its Metrics Model’, in *Recent Findings in Intelligent Computing Techniques*, Springer, Singapore, 271–281, available: https://doi.org/10.1007/978-981-10-8633-5\_28.

Ayas, H.M., Fischer, H., Leitner, P., and De Oliveira Neto, F.G. (2022) ‘An Empirical Analysis of Microservices Systems Using Consumer-Driven Contract Testing’, in *2022 48th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, Presented at the 2022 48th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), Gran Canaria, Spain: IEEE, 92–99, available: https://doi.org/10.1109/SEAA56994.2022.00022.

Balalaie, A., Heydarnoori, A., and Jamshidi, P. (2016) ‘Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture’, *IEEE Software*, 33(3), 42–52, available: https://doi.org/10.1109/MS.2016.64.

Bass, L., Weber, I., and Zhu, L. (2015) *A Software Architect’s Perspective, 2015.*, Addison-Wesley Professional.

De Lauretis, L. (2019) ‘From Monolithic Architecture to Microservices Architecture’, in *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, Presented at the 2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW), Berlin, Germany: IEEE, 93–96, available: https://doi.org/10.1109/ISSREW.2019.00050.

DevOps for Microservices - Creating Change Together [online] (2021) *softwebsolutions*, available: https://www.softwebsolutions.com/resources/devops-and-microservices.html [accessed 12 Jun 2023].

Dhaduk, H. (2022) ‘A Guide on What Are Microservices: Pros, Cons, Use Cases, and More’, *Simform - Product Engineering Company*, available: https://www.simform.com/blog/what-are-microservices/ [accessed 19 May 2023].

Effective Test Automation Approaches for Modern CI/CD Pipelines [online] (2023) *InfoQ*, available: https://www.infoq.com/articles/test-automation-ci-cd/ [accessed 12 Jun 2023].

Fischer, H. (2021) ‘Testing in microservice systems: a repository mining study on open-source systems using contract testing’.

Fong, G. (2022) Contract Test — Spring Cloud Contract vs PACT [online], *Medium*, available: https://blog.devgenius.io/contract-test-spring-cloud-contract-vs-pact-420450f20429 [accessed 17 May 2023].

How Contract Tests Improve the Quality of Your Distributed Systems [online] (2023) *InfoQ*, available: https://www.infoq.com/articles/contract-testing-spring-cloud-contract/ [accessed 17 May 2023].

Jokinen, O. (2020) ‘Software development using DevOps tools and CD pipelines : a case study’, available: https://helda.helsinki.fi/handle/10138/313590 [accessed 21 Jun 2023].

Keni, N.D. and Kak, A. (2020) ‘Adaptive Containerization for Microservices in Distributed Cloud Systems’, in *2020 IEEE 17th Annual Consumer Communications & Networking Conference (CCNC)*, Presented at the 2020 IEEE 17th Annual Consumer Communications & Networking Conference (CCNC), Las Vegas, NV, USA: IEEE, 1–6, available: https://doi.org/10.1109/CCNC46108.2020.9045634.

Larrucea, X., Santamaria, I., Colomo-Palacios, R., and Ebert, C. (2018) ‘Microservices’, *IEEE Software*, 35(3), 96–100, available: https://doi.org/10.1109/MS.2018.2141030.

Lehvä, J. (2019) ‘Testing Integrations with Consumer-Driven Contract Tests’, available: https://helda.helsinki.fi/handle/10138/304680 [accessed 6 Jun 2023].

Lehvä, J., Mäkitalo, N., and Mikkonen, T. (2019) ‘Consumer-Driven Contract Tests for Microservices: A Case Study’, in *Product-Focused Software Process Improvement*, Presented at the International Conference on Product-Focused Software Process Improvement, Springer, Cham, 497–512, available: https://doi.org/10.1007/978-3-030-35333-9\_35.

Microservices and Containers 101 - Learn All About Microservices [online] (2023) *Avi Networks*, available: https://avinetworks.com/what-are-microservices-and-containers/ [accessed 13 Jun 2023].

Nagel, F. (n.d.) ‘Analysis of Consumer-driven contract tests with asynchronous communication between microservices’.

Saboor, A., Hassan, M.F., Akbar, R., Shah, S.N.M., Hassan, F., Magsi, S.A., and Siddiqui, M.A. (2022) ‘Containerized Microservices Orchestration and Provisioning in Cloud Computing: A Conceptual Framework and Future Perspectives’, *Applied Sciences*, 12(12), 5793, available: https://doi.org/10.3390/app12125793.

Singh, S. and Singh, N. (2016) ‘Containers & Docker: Emerging roles & future of Cloud technology’, in *2016 2nd International Conference on Applied and Theoretical Computing and Communication Technology (ICATccT)*, Presented at the 2016 2nd International Conference on Applied and Theoretical Computing and Communication Technology (iCATccT), Bangalore, India: IEEE, 804–807, available: https://doi.org/10.1109/ICATCCT.2016.7912109.

Testing Microservices - Contract Tests [online] (2023) *SoftwareMill*, available: https://softwaremill.com/testing-microservices-contract-tests/ [accessed 13 Jun 2023].

‘The Role of Containers in Your Microservice Architecture’ (2021) *JFrog*, available: https://jfrog.com/devops-tools/article/role-of-containers-in-your-microservice-architecture/ [accessed 13 Jun 2023].

Waseem, M., Liang, P., and Shahin, M. (2020) ‘A Systematic Mapping Study on Microservices Architecture in DevOps’, *Journal of Systems and Software*, 170, 110798, available: https://doi.org/10.1016/j.jss.2020.110798.

What Are Containerized Microservices? – DreamFactory Software- Blog [online] (2023) available: https://blog.dreamfactory.com/what-are-containerized-microservices/ [accessed 13 Jun 2023].

Wickramasinghe, S. (2023) The Role of Microservices in DevOps [online], *BMC Blogs*, available: https://www.bmc.com/blogs/devops-microservices/ [accessed 12 Jun 2023].

# Appendices

# Appendix A: Code Listing Location

Some description or essential code snips can go here.

Code for this dissertation can be found in the repository listed below. To ensure that the code is accessible the repo must be a public repo.

<https://github.com/studentlnumber/reponame.git>