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 pseudonymised 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 Consent can be withdrawn for the publishing of material online by emailing Thomas Dowling; Head of Department at thomas.dowling@lyit.ie to remove items from the ATU Donegal Computing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)

This dissertation addresses the challenge of validating microservice interactions within a DevOps pipeline, extending beyond traditional unit testing. By implementing contract testing frameworks, where consumers define expectations through contracts, integration testing can be enhanced. The central research question examines how contract testing frameworks integrated with CI/CD processes improve microservice interaction validation. The study has three aims: exploring microservices and DevOps synergy, assessing practicality of contract testing, and evaluating benefits of integrating consumer-driven contract testing with CI/CD tools. This research bridges testing gaps and sheds light on the potential of contract testing and CI/CD integration to enhance microservices interactions and overall reliability in modern software development.

# Acronyms

|  |  |  |
| --- | --- | --- |
| Acronym | Definition | Page |
| CI | Continuous Integration | 1 |
| CDC | Consumer-Driven Contract | 4 |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

# Table of Contents

[1 Declaration iv](#_Toc143857054)

[2 Acknowledgements v](#_Toc143857055)

[3 Abstract vi](#_Toc143857056)

[4 Acronyms vii](#_Toc143857057)

[5 Table of Contents viii](#_Toc143857058)

[6 Table of Figures x](#_Toc143857059)

[7 Table of Tables x](#_Toc143857060)

[8 Table of Code Listings xi](#_Toc143857061)

[1. Introduction 1](#_Toc143857062)

[1.1. Purpose 1](#_Toc143857063)

[1.2. Background 2](#_Toc143857064)

[1.3. Problem Statement 3](#_Toc143857065)

[1.4. Research Question 3](#_Toc143857066)

[1.5. Scope and Limitations 4](#_Toc143857067)

[1.6. Report Outline 4](#_Toc143857068)

[2. Literature Review 5](#_Toc143857069)

[2.1. DevOps Pipeline 5](#_Toc143857070)

[2.1.1. Build and Integration Test 6](#_Toc143857071)

[2.1.2. Continuous Deployment 7](#_Toc143857072)

[2.2. Microservice 8](#_Toc143857073)

[2.2.1. Testing Microservices 10](#_Toc143857074)

[2.2.2. Containerization 10](#_Toc143857075)

[2.3. Microservice Architecture Enables DevOps 11](#_Toc143857076)

[2.4. Consumer-Driven Contract Testing 13](#_Toc143857077)

[2.4.1. Contract Testing in DevOps 16](#_Toc143857078)

[2.4.2. Testing Frameworks 17](#_Toc143857079)

[3. Design 20](#_Toc143857080)

[3.1. Pact Contract Test Framework 20](#_Toc143857081)

[3.1.1. Pact Flow 21](#_Toc143857082)

[3.1.2. Pact Broker 22](#_Toc143857083)

[3.1.3. Pact JVM 23](#_Toc143857084)

[3.2. Dockerized Pact Broker 24](#_Toc143857085)

[3.2.1. Docker Compose 25](#_Toc143857086)

[3.3. Jenkins CI/CD Pipeline 27](#_Toc143857087)

[4. Implementation 28](#_Toc143857088)

[4.1 Writing Contract Test with Pact Framework 29](#_Toc143857089)

[4.1.1. Consumer – Creating the Contract 30](#_Toc143857090)

[4.1.2. Provider – Verifying the Contract 33](#_Toc143857091)

[4.2. Integration in Build Pipelines 34](#_Toc143857092)

[4.2.1. Docker Compose Configuration 35](#_Toc143857093)

[4.2.2. Consumer – Publishing Contract to Pact Broker 37](#_Toc143857094)

[4.2.3. Provider – Publishing Verified Result to Pact Broker 38](#_Toc143857095)

[4.2.4. Jenkins Configuration 38](#_Toc143857096)

[4.3. Chapter Conclusion 40](#_Toc143857097)

[5. Test and Results 41](#_Toc143857098)

[5.1 Consumer Unit Test 41](#_Toc143857099)

[5.1.1. Product Unit Test 41](#_Toc143857100)

[5.1.2. Customer Unit Test 45](#_Toc143857101)

[5.2. Provider Verification Test 48](#_Toc143857102)

[5.3 Negative Test 49](#_Toc143857103)

[6. Conclusions 50](#_Toc143857104)

[6.1. Microservice and DevOps Paradigm Shift 50](#_Toc143857105)

[6.2. Contract Testing Frameworks 52](#_Toc143857106)

[6.2.1. Pact Framework 52](#_Toc143857107)

[6.2.2. Docker Compose 53](#_Toc143857108)

[6.2.3. Pact Broker 53](#_Toc143857109)

[6.3. Testing and CI/CD Integration 54](#_Toc143857110)

[6.3.1. Pact Broker and Jenkins 55](#_Toc143857111)

[6.3. Limitations 55](#_Toc143857112)

[9 References 57](#_Toc143857113)

[10 Appendices 61](#_Toc143857114)

[11 Appendix A: Code Listing Location 62](#_Toc143857115)

# Table of Figures

[Figure 1: CI/CD Pipeline high-level flow 6](#_Toc144200657)

[Figure 2: Monolith to Microservices 10](#_Toc144200658)

[Figure 3: Design Framework 21](#_Toc144200659)

[Figure 4: Pact Workflow 23](#_Toc144200660)

[Figure 5: Pact Broker 24](#_Toc144200661)

[Figure 6: Product consumer service retrieving item details from Retailer provider service 29](#_Toc144200662)

[Figure 7: Customer consumer service retrieving order details from Retailer provider service 29](#_Toc144200663)

[Figure 8: Methods in Retailer Controller Class to return item and order details to consumers 30](#_Toc144200664)

[Figure 9: Pact Framework 31](#_Toc144200665)

[Figure 10: Consumer unit test class to generate pact file 33](#_Toc144200666)

[Figure 11: Provider verification class to verify the contract 35](#_Toc144200667)

[Figure 12: Docker Compose File 37](#_Toc144200668)

[Figure 13: Pact configuration in build.gradle 38](#_Toc144200669)

[Figure 14: Provider Jenkins configuration 39](#_Toc144200670)

[Figure 15: Provider Jenkins configuration 40](#_Toc144200671)

[Figure 16: 'can-i-deploy' Jenkins configuration 40](#_Toc144200672)

[Figure 17: Consumer Jenkins workflow 40](#_Toc144200673)

[Figure 18: Provider Jenkins workflow 41](#_Toc144200674)

[Figure 19: Running product unit test 42](#_Toc144200675)

[Figure 20: Product target folder 43](#_Toc144200676)

[Figure 21: Pact broker publishing product pact 43](#_Toc144200677)

[Figure 22: Service Interactions details provided by pact broker 44](#_Toc144200678)

[Figure 23: Product Pact File 45](#_Toc144200679)

[Figure 24: Pact broker publishing customer pact 46](#_Toc144200680)

[Figure 25: Customer target folder 47](#_Toc144200681)

[Figure 26: Customer Pact File 48](#_Toc144200682)

[Figure 27: Retailer verification test 49](#_Toc144200683)

[Figure 28: Product pact verified in pact broker 49](#_Toc144200684)

[Figure 29: Customer pact verified in pact broker 49](#_Toc144200685)

[Figure 30: Negative test failing 50](#_Toc144200686)

[Figure 31: Test failure published in pact broker 50](#_Toc144200687)

# 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, maximize the effectiveness of continuous testing (CT) in a distributed system. For this we design a system that incorporates multiple microservices and integrates contract testing frameworks and CI/CD integration tools 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, 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 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, microservices, 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 microservice 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 contract testing frameworks with CI/CD integration enhance the validation of microservice interactions in a distributed system?

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

1. To explore the microservices architecture paradigm and DevOps principles, examining their mutual reinforcement and the advantages of integrating them
2. To explore and assess the practicality of contract testing frameworks in the context of validating microservice interactions.
3. To assess the effectiveness and benefits of integrating consumer-driven contract testing frameworks with CI/CD integration tools in terms of enhancing the quality, speed and reliability of microservice-based systems.

## Scope and Limitations

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

## Report Outline

This dissertation includes a comprehensive literature review in chapter 2, which examines the relevant research on DevOps Pipeline, Microservices and Consumer-Driven Contract (CDC) Testing. Chapters 3 and 4 focus on the design and implementation of a multiple microservice system that incorporates contract testing frameworks and CI/CD integration tools. These chapters address the research question that was introduced in chapter 1. Finally, in chapter 6, the dissertation concludes by discussing on the theoretical research done on microservices and DevOps, CDC testing framework and CI/CD integration with the practically implemented system and suggesting future areas of research and development in this domain.

# Literature Review

In this literature review, the initial research lies on the DevOps pipeline and microservices, with a detailed exploration of key aspects such as build and integration testing, deployment, and containerization. Furthermore, the study investigates how the adoption of microservice architecture facilitates the implementation of a DevOps pipeline. Finally, it examines the emergence of Consumer Driven Contract (CDC) testing and its significance within the context of microservices and CI/CD pipeline.

## DevOps Pipeline

DevOps encompasses a collection of methodologies designed to minimize the duration between making modifications to a system and deploying those changes to the production environment (Balalaie et al. 2016). DevOps places a significant emphasis on ensuring the quality of software, encompassing both the code itself and the delivery process. Any approach or technique that supports these objectives can be classified as a DevOps practice (Bass et al. 2015; Brunnert et al. 2015). Like agile methodologies, DevOps breaks down software applications into discrete components or modules to enhance processing speed and enhance overall quality (Amrit et al. n.d.). DevOps places emphasis on achieving specific objectives rather than prescribing specific methods. The primary aim of DevOps practices as mentioned before is to minimize the time elapsed between making a change to a system and incorporating that change into the production environment. The longer it takes for a release to reach the market, the fewer benefits can be gained from the new features or quality enhancements introduced in that release. The ideal approach is to have a continuous release cycle enabled by DevOps practices, commonly referred to as continuous delivery or continuous deployment (Bass et al. 2015). However, to ensure the continuous delivery of reliable and high-quality software, software testing is a crucial aspect and its integration into DevOps pipeline is essential. This dissertation specifically concentrates on incorporating microservices and Consumer-Driven Contract (CDC) test frameworks into the pipeline to test the interactions among microservices. By including these testing frameworks, the aim is to enhance the overall testing process and ensure effective validation of the microservices within the DevOps pipeline.

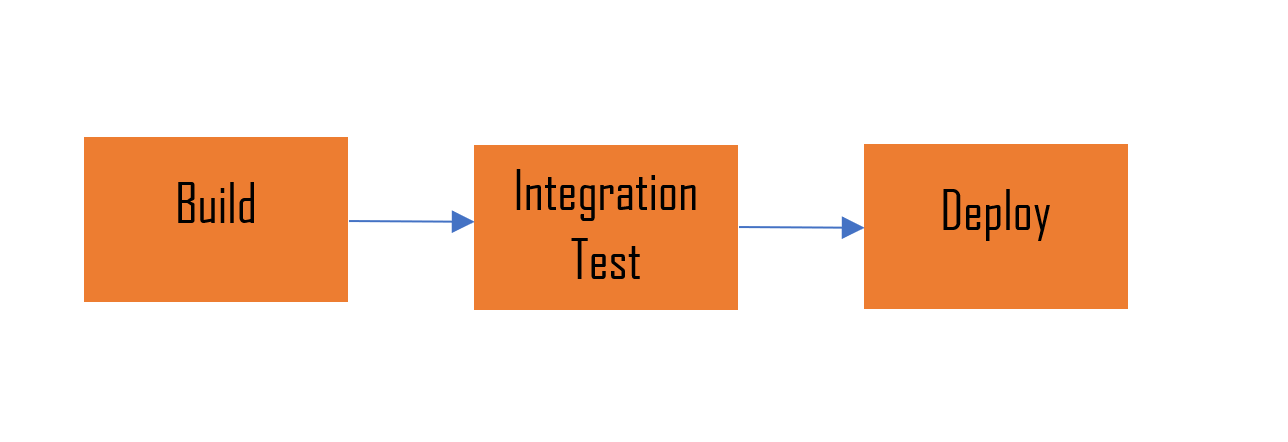


Figure : CI/CD Pipeline high-level flow

### 2.1.1. Build and Integration Test

The build process involves creating an executable artifact by using input such as source code and configuration to generate a deployable output (Bass *et al.* 2015). Continuous integration (CI) servers are responsible for performing the build and integration tests. After the build is finished, a series of automated tests are conducted to verify if the integration with other parts of the system exposes any errors (Bass *et al.* 2015; Jokinen 2020). This step is known as integration testing, where the constructed executable artifact is thoroughly tested.

Testing plays a crucial role in the success of a DevOps pipeline, as it directly impacts the achievement of desired outcomes. According to Agrawal and Rawat (Agrawal and Rawat 2019) expertise in testing process is the key factor that contribute to the advancement of continuous integration, continuous delivery, and continuous deployment. These testing practices are instrumental in ensuring the smooth and reliable functioning of the DevOps pipeline.

Continuous testing is a valuable practice that helps to identify integration issues at earlier stages in the software development life cycle. Continuous testing involves early and frequent testing at various stages, incorporating it throughout the software development life cycle, and automating the testing process (Agrawal and Rawat 2019). By integrating testing throughout the process, defects and issues can be detected and resolved more efficiently and cost-effectively. This approach also allows testers to allocate their time more effectively, as continuous testing reduces the need for repetitive and time-consuming manual testing (Angara *et al.* 2018).

Continuous testing (CT) faces a significant challenge in dealing with heterogeneous environments that do not completely mirror the production environment and application architecture (Agrawal and Rawat 2019). This discrepancy between the testing environment and the actual production environment can lead to potential issues and limitations in accurately assessing the performance and behaviour of the software. It becomes crucial to address this challenge by striving to create test environments that closely resemble the production environment, incorporating similar infrastructure, configurations, and architectural components. By minimizing the disparities between the testing and production environments, organizations can improve the effectiveness and reliability of continuous testing processes.

As teams release updates at varying speeds, the testing team faces the challenge of planning robust end-to-end testing, especially in case of microservices where services depend on different interconnected services. Moreover, running a complete production-like cluster for testing purposes can be expensive, making it impractical for each team to have its own full-scale cluster solely for testing (Amrit *et al.* n.d.). Therefore, it is necessary to find alternative solutions to simulate the production environment effectively and conduct comprehensive testing while optimizing resource utilization.

In order to address the discrepancy between testing environments and production environments in microservice-based DevOps pipelines, various approaches are being employed. One such approach is the use of Consumer-Driven Contract (CDC) test frameworks which enable the execution of isolated integration tests for microservices (Fischer 2021). These frameworks facilitate testing interactions between microservices in a controlled manner, ensuring that each microservice behaves correctly based on the contracts defined by its consumers. The dissertation will thoroughly analyse and assess the benefits, challenges, and potential impact of adopting CDC test frameworks on the testing practices within the DevOps pipeline.

### 2.1.2. Continuous Deployment

Continuous deployment is a software development strategy that involves the direct deployment of new code or changes to the production environment, following a series of comprehensive automated tests (Oberoi 2023). This approach ensures a rapid and seamless delivery of updates to the production environment. In fact, the primary objective of the DevOps CI/CD pipeline is to accelerate software delivery, enabling more frequent and rapid releases. Continuous Testing is indeed a crucial aspect of implementing Continuous Deployment, as stated by Oberoi (Oberoi 2023). Organizations are adopting continuous delivery and deployment in response to dynamic business requirements and increased market competition (Victor 2023). DevOps revolutionizes the software development and delivery landscape, enabling organizations to keep up with the demands of today's fast-paced market (‘10 Best Deployment Tools for DevOps in 2023’ 2023). Thus, DevOps enables organizations to evolve and produce products at a faster pace while maintaining superior quality compared to traditional infrastructure management processes and traditional software development approaches.

Numerous tools are available for managing deployments in the software development process. The advent of lightweight containers and image management tools has greatly facilitated the deployment process for developers, particularly in creating small-scale production-like environments for testing purposes (Bass *et al.* 2015). These tools provide efficient means to package applications and their dependencies into portable containers or images, enabling easy deployment and reproducibility across different environments.

Containers are a popular approach used in software development, packaging, and deployment (Pahl 2015). Containers encapsulate all the necessary components, including the application code, runtime environment, libraries, and system tools, into a single, portable unit (‘What is a Container Deployment? | VMware Glossary’ 2023). With the use of containers and image management tools, developers can swiftly set up isolated testing environments that closely resemble production, allowing for comprehensive testing and validation of software changes before deploying them to the actual production environment.

## Microservice

The architecture of cloud computing has undergone significant transformation, transitioning from a monolithic structure to a more agile and scalable microservices approach (Saboor *et al.* 2022). Over the past few years, cloud platforms such as Amazon, Microsoft, and IBM have become increasingly popular among companies as the preferred method for delivering and operating modern applications (Cito *et al.* 2015). They have gained mainstream adoption as companies recognize the benefits and advantages these platforms offer (Keni and Kak 2020). The Microservices Architecture (MSA) has gained relevance in recent years as an architectural style that allows organizations to harness the benefits of cloud computing(Mazlami *et al.* 2017). According to the International Data Corporation's forecast, by 2021, the majority of application development on cloud platforms is expected to utilize microservices, accounting for approximately 80 percent of the total (Larrucea *et al.* 2018). Companies such as Netflix, Twitter, eBay, Amazon, Hailo, Groupon, and Zalando have adopted cloud-based microservices architecture within their operations (Saboor *et al.* 2022).

Netflix, being at the forefront, was among the early adopters of Microservices Architecture (MSA) when it transitioned its system from a monolithic architecture. Currently, Netflix's platform has evolved into an extensive system comprising hundreds of microservices (*Microservices at Netflix Scale: Principles, Tradeoffs & Lessons Learned • R. Meshenberg • GOTO 2016* 2016). Uber, renowned for its ride-sharing service, has developed a platform consisting of approximately 2,200 microservices (Gluck 2020). This extensive system of microservices powers the various functionalities and operations of the Uber platform.

In the past, large companies like Amazon and eBay relied on the traditional approach of Monolithic Architecture (MA) for software development, which has since been surpassed by the microservices (De Lauretis 2019). In a Monolithic Architecture (MA), all functions are contained within a single application. While monolithic applications have advantages, such as ease of development, testing, and deployment for simpler applications, they pose challenges when the complexity of the application grows (‘From Monolith to Microservices: A Dataflow-Driven Approach | IEEE Conference Publication | IEEE Xplore’ 2023).

In applications based on Microservices Architecture (MSA), the system is divided into small and independent microservices (Lewis and Fowler 2023). In a microservices architecture, the individual services are designed to be loosely coupled and they communicate with each other through platform-independent interfaces, enabling flexibility and interoperability within the architecture (Zimmermann 2016). As a result, the individual microservices within a microservices architecture can be developed, tested, and deployed independently of each other (Newman 2018; Fischer 2021) which enables faster deployment of new features or bug fixes, as only the relevant microservices need to be deployed or updated. And, for each service most of the testing, packaging, and deployment tasks can be automated(‘The Role of Containers in Your Microservice Architecture’ 2021). Each service has the ability to be deployed independently on different platforms and technology stacks. It runs as a separate process and communicates with other services using lightweight communication mechanisms like RESTful (Representational State Transfer) APIs (Balalaie *et al.* 2016).

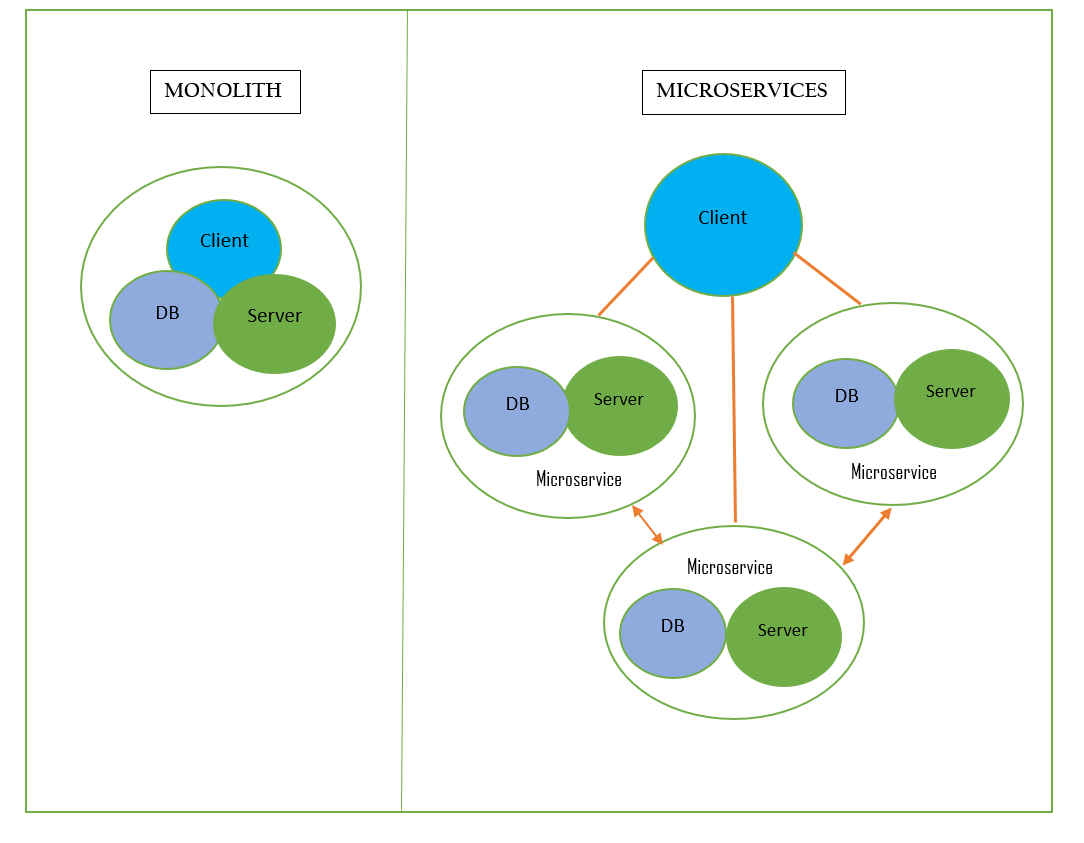


Figure : Monolith to Microservices

### 2.2.1. Testing Microservices

To be integrated into the actual system, individual microservices need to establish interactions with each other using either synchronous or asynchronous messaging protocols. Validating the proper functioning of these interactions is crucial, and integration testing plays a vital role in this process. Integration testing involves testing the collaboration and data exchange between different microservices to ensure they work correctly as a cohesive system (Fischer 2021).

However, integration testing is often highlighted as one of the significant challenges in Microservices Architecture (MSA)-based applications due to its distributed nature and the complexity that arises from coordinating and validating their interactions (Soldani *et al.* 2018; Waseem *et al.* 2020).Also, the need to run tests in isolation without dependent services while accurately validating the interactions with those services is a challenge. To overcome this, various approaches are currently employed to reconcile this discrepancy and enable effective integration testing in Microservices Architecture (MSA)-based applications. To address the challenge, the application of Consumer Driven Contract (CDC) testing practice can help mitigate the issue. CDC testing framework enables a more effective approach to integration testing, alleviating the problem at hand (Fischer 2021).

### 2.2.2. Containerization

Deploying microservices in the development environment can present challenges. Despite having the application code isolated into separate services, developers still need to deploy the dependent services alongside the isolated services to run them on their local machines(Balalaie *et al.* 2016). As companies continue to produce an increasing number of microservices, the deployment environment becomes more complex. Without proper configuration and management, a roadmap of microservices can quickly become unmaintainable (Amrit *et al.* n.d.).

Containerization is a concept that enhances and complements microservices-based models in addressing the challenges of deployment and management (Amrit *et al.* n.d.). Containerization is a virtualization technique that aims to achieve efficient resource isolation by sharing the kernel with the host operating system, allowing for lightweight and efficient encapsulation of applications and their dependencies within containers (Pahl 2015). By encapsulating microservices and their dependencies into containers, containerization provides a standardized and portable environment that simplifies deployment, scalability, and maintenance of microservices-based systems (Singh and Singh 2016; Keni and Kak 2020). Microservices bundled into containers can be deployed on physical hardware, ensuring a consistent software execution environment that remains consistent from the developer's environment to the end consumer's system (Keni and Kak 2020).

The emergence of lightweight containers has facilitated developers in deploying their applications into small-scale production-like environments more effortlessly, primarily for testing purposes. These technologies streamline the process of creating reproducible and isolated testing environments, enabling developers to validate their applications in conditions that closely resemble production settings (Bass *et al.* 2015).

## Microservice Architecture Enables DevOps

DevOps is a cultural approach that integrates updated methodologies, procedures, team dynamics, and tools to optimize an organization's capacity to swiftly deliver applications and services (Mueller, 2018; Sánchez-Gordón and ColomoPalacios, 2018). In fact, DevOps can serve as a process framework that can be utilized for the development, deployment, and management of Microservices Architecture (MSA) (Larrucea *et al.* 2018). The combination of microservices and DevOps facilitates the coexistence of various benefits, including reusability, decentralized data governance, automation, and inherent scalability (Balalaie et al., 2016).

MSA and DevOps share numerous common characteristics, making them highly compatible with each other. Both MSA and DevOps emphasize the concept of breaking down complex problems into smaller components and addressing them through small cross-functional teams (Watts, 2020). DevOps provides continuous integration and deployment, enabling containerized microservices to operate independently and autonomously. While it is not mandatory to adopt Microservices Architecture (MSA) when implementing DevOps, utilizing MSA can effectively address many challenges that arise in the context of DevOps. (Bass et al., 2015). Also, though DevOps can be applied to monolithic software systems as well, but microservices provide an effective environment for implementing DevOps practices by emphasizing the significance of small teams (Balalaie *et al.* 2016).

Indeed, the structure of microservices emerged as a result of the widespread adoption of DevOps principles, which originated from pioneering companies like Amazon, Facebook, Google, Netflix, and SoundCloud (Amrit *et al.* n.d.). By utilizing microservices, DevOps teams gain the ability to develop independent features concurrently. Instead of following a sequential handoff process (e.g., from development to testing to production), cross-functional teams collaboratively build, test, release, monitor, and maintain applications together. This promotes parallel development and efficient collaboration within the DevOps workflow (Amrit *et al.* n.d.). Therefore the popularity of Microservices Architecture (MSA) in the industry has been driven by its numerous advantages, including enhanced availability, flexibility, scalability, loose coupling, and the ability to achieve high velocity in software development and deployment (Hasselbring and Steinacker 2017) .

According to the International Data Corporation (‘IDC: The premier global market intelligence firm.’ 2023), it was projected that by the end of 2021, approximately 80% of cloud-based applications would be developed using Microservices Architecture (MSA) (Larrucea *et al.* 2018). Also, it was predicted that the global DevOps market would reach a value of $5.6 billion in 2021 (Elliot et al., 2018)(‘IDC: The premier global market intelligence firm.’ 2023). According to Google Trends data (Balalaie et al. 2016), both DevOps and microservices have exhibited similar growth rates since 2014, indicating their increasing popularity and adoption within the technology landscape. Another report, authored by (Yousif 2016), highlights that organizations adopt MSA for various reasons, including gaining agility (82%), improving organizational performance (57%), and achieving scalability (78%). The report also indicates that 47% of organizations implemented MSA as a result of their motivation to embrace DevOps (‘Lightstep’ 2018).

For ensuring fast and reliable delivery of microservices through DevOps, the (CI)/(CD) pipeline has to be of high quality and efficiency. In order to fully leverage the advantageous aspect of microservices in DevOps, it is essential to have expedited testing processes. The introduction of microservices in architecture significantly increases the number of accessible interfaces. Since the majority, if not all, of the communication between microservices depends on these interfaces, it presents a new challenge in testing (Nagel n.d.). It is essential to ensure reliable and efficient testing of these interfaces within the DevOps pipeline. Fortunately, there is a pattern available to address this challenge, it is contract testing. In this dissertation, we will examine the effectiveness of integrating contract testing frameworks into the pipeline to enhance efficiency of testing the microservice interfaces.

## Consumer-Driven Contract Testing

Testing plays a crucial role in identifying faults within MSA based systems by uncovering issues that may arise from dependencies between various services. Moreover, these tests need to accurately assess the correctness of interactions with those specific services. An important topic of discussion in microservices architectures (MSAs) revolves around the granularity of services, which often vary in their levels of abstraction (Newman 2019). Different test types in microservices architectures (MSAs) reflect the varying granularity of tests (Vocke 2023). Microservices' individuality, as highlighted by Lewis and Fowler (Lewis and Fowler 2023), necessitates the inclusion of unit tests within the software development lifecycle, as emphasized by Newman (Newman 2019). However, the broader architectural scope of microservices also encompasses other types of tests, such as integration tests, as discussed by Waseem, Liang, and Shahin (Waseem *et al.* 2020). Integration-level, component-level, and system-level testing are essential for validating the interactions of individual microservices and ensuring their seamless integration into the overall system. According to a study (‘Design, monitoring, and testing of microservices systems: The practitioners’ perspective’ 2021), the ability to write effective integration test cases is considered the most crucial skill for adequately testing microservices. End-to-end testing in Microservices-Based Architectures (MSAs) can be challenging due to the decentralized nature of the system and the independent release cycles of different teams. In addition, due to the high cost associated with running a full-scale production cluster, it is often impractical for each team to maintain their own dedicated cluster solely for testing purposes (Amrit *et al.* n.d.).

A recent publication titled "Testing Strategies in a Microservice Architecture" (‘Testing Strategies in a Microservice Architecture’ 2023), states that among various approaches being employed to address this challenge, the use of Consumer-Driven Contract (CDC) testing is considered a potential solution to overcome the challenges associated with integration testing. A substantial number of researches has concluded that CDC testing is the ideal choice for testing microservices (Dai *et al.* 2007; Lehvä *et al.* 2019; Waseem *et al.* 2020; Fischer 2021; Ayas *et al.* 2022; Wu *et al.* 2022; Vocke 2023; Nagel n.d.; Nyman n.d.; Selleby n.d.).

The book DevOps a software architect’s perspective (Bass *et al.* 2015), states that proper implementation of Consumer-Driven Contract (CDC) testing can result in a high level of confidence in the microservice, reducing the need for extensive end-to-end test cases. CDC testing serves as a coordination method and influences the composition and evolution of user stories within a microservice over time. Both consumers and microservice developers collaboratively create and take ownership of the user stories. The definition of CDC becomes a function of how functionality is allocated to the microservice, managed by the service owner as part of the coordination that defines the next iteration. As a result, CDC does not impede the progress of the current iteration.

In a case study conducted by (Lehvä 2019) on an industrial system, the application of Consumer-Driven Contract (CDC) testing revealed several benefits. These advantages encompassed isolated integration testing, enhanced communication between teams, increased flexibility for providers to accommodate consumer needs, and efficient detection of breaking changes in the API. Moreover, the study suggested that CDC tests have the potential to replace traditional integration tests, as they successfully captured all defects arising from the implemented integrations. Based on these findings, it can be concluded that consumer-driven contract testing is a valuable addition to testing strategies, particularly for integration-heavy systems, especially those based on microservices.

In his thesis paper titled "Analysis of Consumer-driven contract tests with asynchronous communication between microservices," Florian Nagel (Nagel n.d.) asserts that Consumer-Driven Contract (CDC) testing provides faster execution and yields reliable results, often detecting errors before the need for extensive end-to-end tests. As a consequence, the reliance on end-to-end tests can be reduced in the testing pipeline, thereby diminishing their intensity and necessity. The implementation of fast and dependable testing pipelines enables frequent and regular software releases, leading to an efficient DevOps pipeline.

Consumer-Driven Contract (CDC) testing is a testing method that verifies the integrations between services and ensures their continued functionality following the introduction of new changes to the system (‘Consumer-Driven Contracts: A Service Evolution Pattern’ 2023). the main concept of Consumer-Driven Contract (CDC) testing is that when an application or service (referred to as the consumer) utilizes an API offered by another service (known as the provider), a contract is established between them (Lehvä *et al.* 2019). The contract encompasses details on how the consumer interacts with the provider, including the specific method of invocation and the data utilized from the responses received. Within the contract, the consumer expresses its expectations to the provider, who subsequently confirms its capability to meet these expectations. Once confirmed, these expectations become binding and form a contractual agreement between the consumer and the provider (Fischer 2021; Ayas *et al.* 2022). Both parties can access the contract which facilitates independent testing.

Through testing against these contracts, the verification process ensures that the consumer and the provider can seamlessly integrate and effectively communicate with each other (‘Testing Microservices - Contract Tests’ 2023). Any changes made to the contract must be effectively communicated between all parties involved (Bass *et al.* 2015; Vocke 2023). CDC tests can extend beyond the scope of individual services as they necessitate communication and collaboration between the teams responsible for the interacting microservices. However, the actual testing itself can be conducted in isolation, focusing on the specific interactions between the consumer and provider without requiring the simultaneous involvement of all microservices(Fischer 2021). When both parties adhere to the contract, they can utilize it as a foundation to verify their respective sides of the integration process. The consumer can employ the contract to create mocks or simulate the behaviour of the provider during their own testing procedures (Lehvä *et al.* 2019). This ensures that the consumer's tests align with the expected behaviour defined by the contract, promoting effective integration verification. On the other hand, the provider can utilize the contract to replay the consumer's requests against its API (Lehvä *et al.* 2019). By doing so, the provider can validate that its API implementation correctly handles the expected requests defined in the contract. This helps ensure that the provider's side of the integration functions as intended and aligns with the agreed-upon contract specifications.

However, it is important to note that CDC tests do not specifically target testing the functionality or business logic of the individual services (Fischer 2021). Instead, their primary focus is on verifying the communication and interaction between the services involved. CDC tests aim to ensure that the consumer and provider can effectively communicate and exchange data according to the contract, rather than validating the internal workings or specific functionalities of each service in isolation.

### 2.4.1. Contract Testing in DevOps

According to a recent study on testing approaches for CI/CD Pipelines (‘Effective Test Automation Approaches for Modern CI/CD Pipelines’ 2023), the increasing popularity of CI/CD has brought about a significant transformation in the field of software testing. Developers now require fast feedback from pipelines to determine the success of their software updates. In order to adapt to this significant change and ensure the delivery of high-quality automated tests, the study proposes several solutions, one of which involves the use of mocks and stubs whenever feasible. To achieve more dependable test results, as well as gain better control over testing efforts and enhance coverage, the recommended approach is to incorporate mocking into the test framework and rely on stubs to intercept complex data patterns instead of relying solely on external functions. This makes CDC testing frameworks a vital suite for modern CI/CD pipelines.

Lastminute.com has recently implemented contract tests to address challenges associated with system-level integration tests, aiming to enhance the feedback cycle and development process. Similarly, eBay is utilizing contract testing to facilitate the safe evolution of their internal APIs and cater to the requirements of client teams. (‘eBay and lastminute.com Adopt Contract Testing to Drive Architecture Evolution’ 2023). Lastminute.com experienced a significant positive impact on their microservice architecture and delivery process by adopting contract tests, resulting in a substantial reduction in test-execution times compared to traditional system-level tests. eBay utilizes contract testing to validate integration points within their platform, facilitating collaborative efforts and ensuring the smooth evolution of internal APIs without compatibility issues.

The integration of contract testing frameworks within a pipeline enhances the validation of microservice interactions by providing a systematic and automated approach to verify the contracts (interfaces) between microservices. Contract testing ensures that each microservice adheres to the agreed-upon contract, and this has several benefits. Contract testing allows you to verify the interactions between microservices during the development phase. By checking if the contracts are correctly implemented, any compatibility issues or mismatches between services can be identified early on, reducing the risk of integration problems at later stages. In a microservices architecture, different teams are often responsible for different services. By defining contracts and using contract testing, teams can agree on the interface specifications, making it easier to collaborate and work independently. This leads to better communication and coordination between teams. Also, as the codebase evolves, contract tests serve as a form of regression testing. Whenever changes are made to a microservice or its contract, contract tests ensure that existing interactions between services remain intact and functional. This helps prevent regressions and unintended side effects. Thus, Contract testing can be integrated into a Continuous Integration (CI) and Continuous Deployment (CD) pipeline. When a new version of a microservice is built, the contract tests can be automatically triggered. If the contract tests fail, the new version won't be deployed, ensuring that only compatible microservices are promoted to production. Throughout this dissertation, we are exploring the automation of contract testing and its integration into the pipeline to validate microservice interactions effectively. This approach aims to ensure the maintenance of a reliable and robust microservices architecture.

### 2.4.2. Testing Frameworks

Contract testing frameworks are tools or libraries designed to facilitate contract testing between services in a microservices architecture. These frameworks help in defining, generating, and executing contract tests to validate the interactions between different services. Contract testing frameworks are crucial for ensuring that services conform to the specified contracts, reducing integration issues and promoting compatibility within the system. Two popular contract testing frameworks include Pact and Sprong Cloud Contract. Both PACT and Spring Cloud are powerful frameworks for contract testing, and they offer different features and integrations depending on the technology stack and frameworks used in the microservices architecture (‘Introduction | Pact Docs’ 2022; ‘Spring Cloud Contract’ 2023).

* **Pact** is a contract testing framework that enables consumer-driven contract testing between services. It allows developers of consumer applications (service consumers) to define the expected interactions they require from a provider service. Pact then generates a contract that represents these expectations. The contract is shared with the provider service developers, who implement their service to fulfil the contract defined by the consumers. During testing, the provider's service is validated against the contract to ensure that it meets all the agreed-upon expectations. Pact supports multiple programming languages and frameworks, making it suitable for diverse microservices architectures. It provides tools and plugins to integrate seamlessly into CI/CD pipelines, enabling automated contract testing as part of the development and deployment workflow.
* **Spring Cloud Contract** is a contract testing framework specifically designed for microservices built using the Spring Cloud ecosystem. It allows developers to define contracts for RESTful APIs or messaging-based interactions using Groovy or YAML DSL (Domain-Specific Language). With Spring Cloud Contract, you define contracts for both the consumer and the provider services. The framework then generates tests for the consumer based on the contracts, and the provider service implements the actual behaviour to satisfy these contracts. The contract definitions are typically stored alongside the code in the respective projects' repositories. These contracts are used for testing during the CI/CD process, ensuring that both consumer and provider services remain compatible. Spring Cloud Contract supports integration with popular testing frameworks, making it easy to incorporate contract testing into the existing testing infrastructure.

In this literature survey, we have encountered compelling evidence showcasing the effectiveness of CDC as the optimal testing strategy for microservices. The seamless integration of DevOps and microservices has further reinforced the importance of CDC. Additionally, various studies have highlighted the pressing need for efficient testing frameworks in modern CI/CD pipelines. In this dissertation paper, our aim is to adopt the CDC testing framework within a CI/CD integration pipeline and investigate the efficiency of distributed software with CDC testing.

# Design

The system design adopts a simplified pipeline methodology using Jenkins for CI/CD workflows. The pipeline integrates the Pact contract test framework, Pact Broker for centralized contract management, and Docker Compose for containerized test environments. The main goal of this design is to create the Pact contract test framework, Pact Broker, and Docker Compose and integrate these within the Jenkins CI/CD workflow. The pipeline automates contract tests, generates detailed test reports, and leverages Docker Compose to create isolated and reproducible test environments.

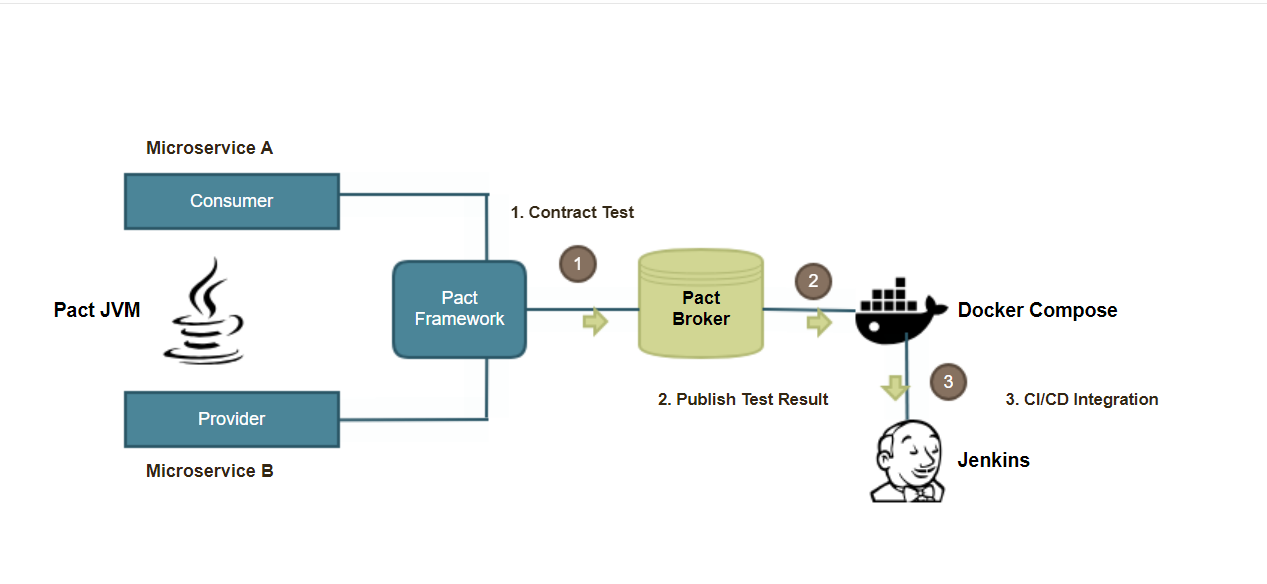


Figure : Design Framework

### 3.1. Pact Contract Test Framework

Pact is a code-first tool designed for testing HTTP and message integrations through contract tests (‘Introduction | Pact Docs’ 2022). These contract tests are employed to verify that the inter-application messages adhere to a mutually agreed-upon understanding documented in a contract.

Before deploying an application to production, it is crucial to ensure its seamless interaction with other integrated applications. Traditionally, this is achieved by running integration tests with "live" deployed applications. While these tests provide confidence for a successful release, they also come with certain drawbacks. Integration tests introduce dependencies, leading to potential complexities, slower feedback cycles, higher susceptibility to breaking, and increased maintenance efforts.

When employing isolated tests, we test each side of an integration point by using simulated versions of the other applications involved. This approach results in two separate sets of tests that run independently, providing rapid feedback, stability, and ease of maintenance. However, the drawback is that these tests do not offer the confidence needed for a release. The reason being that there is no guarantee that the behaviour of the simulated applications accurately reflects that of the real ones.

### 3.1.1. Pact Flow

Pact effectively addresses the challenge of synchronizing two sets of tests by utilizing a "contract" known as the “pact." This contract acts as a mutual agreement between the consumer and the provider. During the consumer tests, every request sent to a Pact mock provider is recorded into the contract file, along with the expected response for each request. After creating a Pact, a simulated consumer replays each request made against the actual provider and then compares the real responses with the expected ones. This verification process ensures that the behaviour of the simulated applications aligns with that of the real applications. When the actual and expected responses match, it confirms that the simulated applications behave identically to the real ones. Consequently, Pact helps ensure that integration between the applications functions reliably and as intended in the production environment.

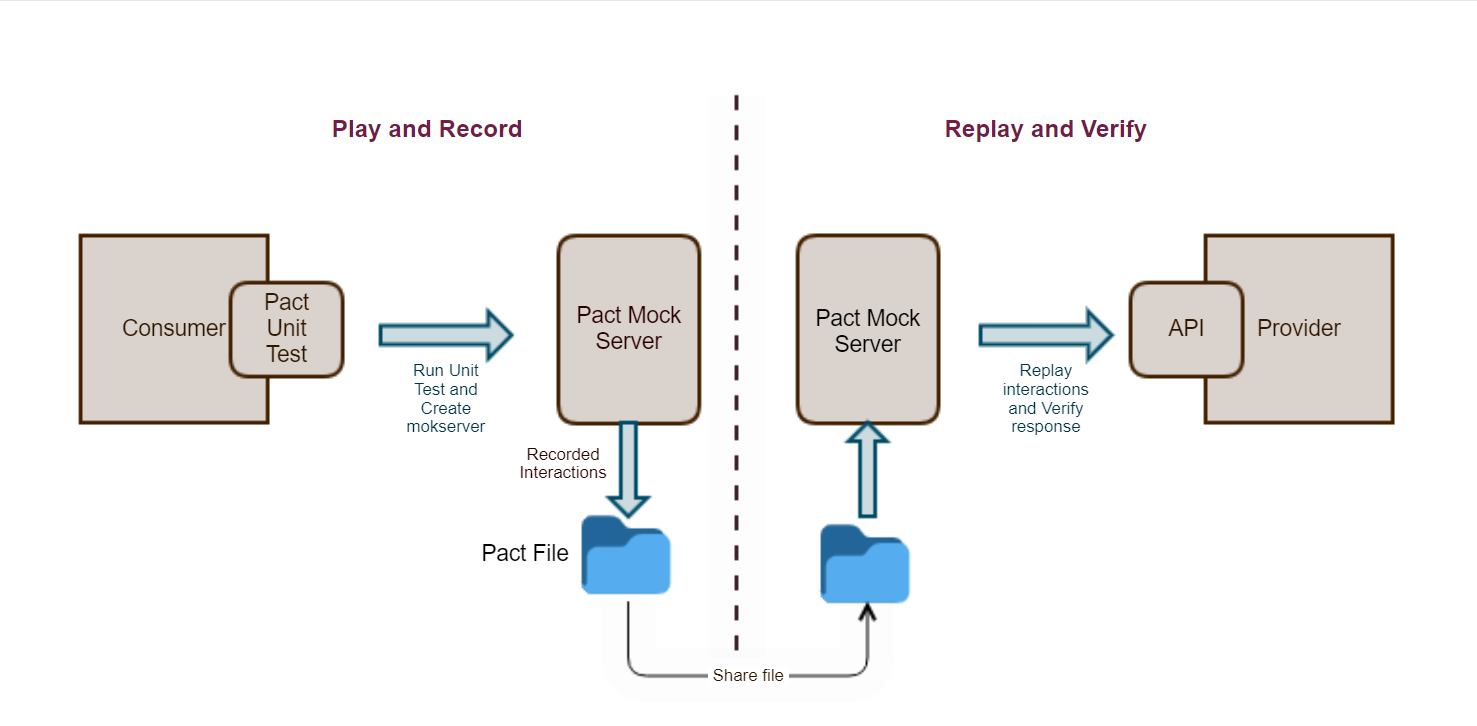


Figure : Pact Workflow

Thus, Pact provides a powerful testing solution that offers numerous benefits for integration testing. The use of contracts ensures that tests run independently, reducing dependencies and complexities. Swift feedback is achieved through rapid verification of simulated requests against real providers, enabling prompt issue detection. The stability of tests is enhanced by comparing actual responses with expected ones, minimizing false positives and negatives. The clear and well-documented contracts facilitate easy maintenance, simplifying future updates and modifications. Pact instils confidence in the release process by guaranteeing that the simulated applications behave identically to their real counterparts, ensuring seamless communication and reliable integration in real-life scenarios.

### 3.1.2. Pact Broker

The Pact Broker is an application that stores all the contracts in a database. It knows for each consumer version which provider version has - or has not - verified the contract.

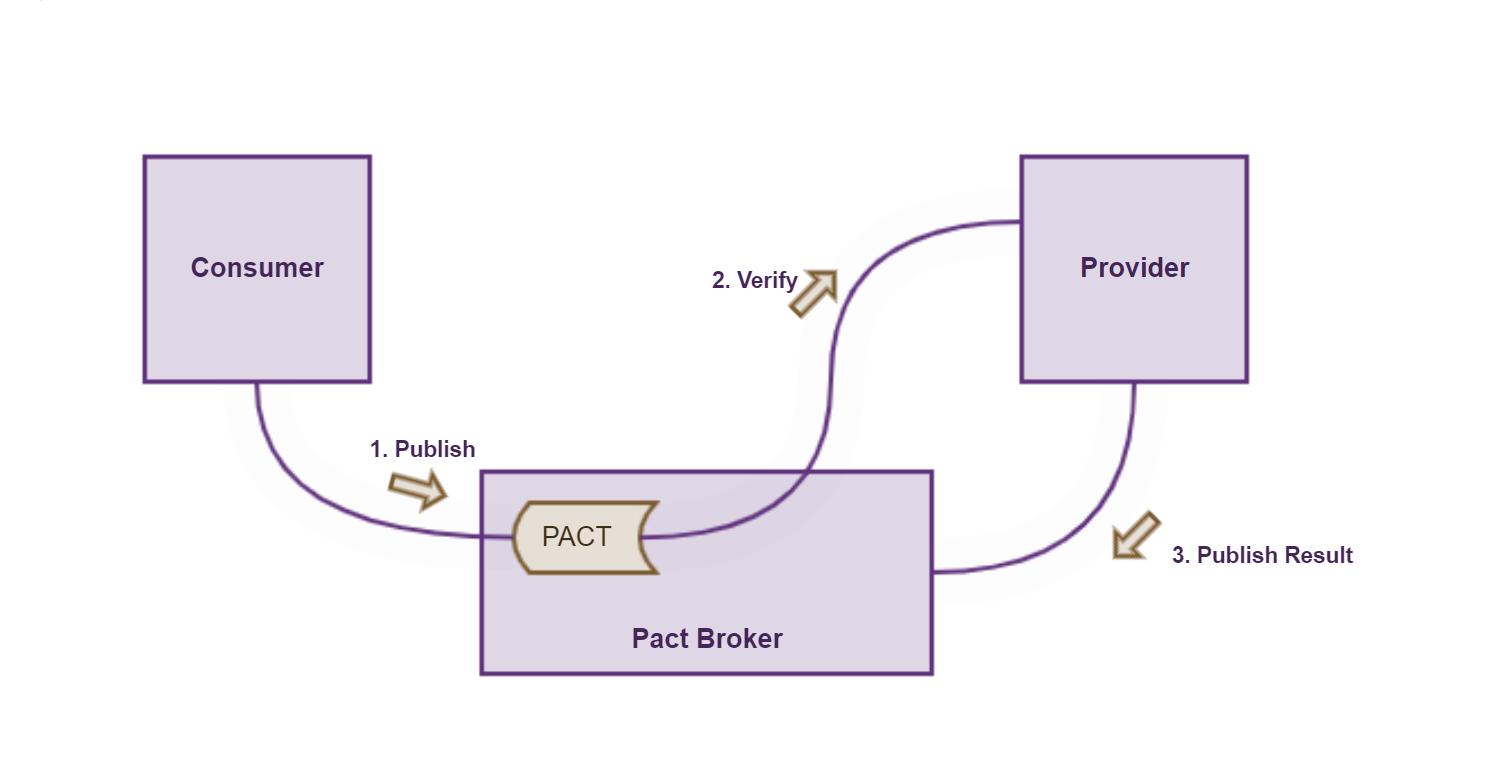


Figure : Pact Broker

Once contract tests are established, an efficient process for managing contract testing becomes essential, and this is where the Pact Broker comes to picture. The Pact Broker is a tool designed for sharing consumer-driven contracts and verification results among different applications (Thirion 2019). The Pact Broker streamlines contract testing by enabling teams to share and collaborate on contracts, manage them across branches and environments, orchestrate builds for safe deployment, and seamlessly integrate into existing processes and tooling (‘Sharing Pacts with the Pact Broker | Pact Docs’ 2023). It serves as a central hub for efficient contract management, ensuring reliable integrations throughout the development and deployment lifecycle. Therefore, Pact Broker facilitates integration and automation of contract testing within your CI/CD release pipelines.

### 3.1.3. Pact JVM

PACT provides support for a variety of programming languages, enabling users to perform contract tests in languages such as Ruby, JVM, .NET, JavaScript, Go, and Python (Muthu. 2019). Pact JVM is a set of libraries and tools that enable Consumer-Driven Contract Testing (CDCT) for JVM-based applications (‘Overview | Pact Docs’ 2021). Pact JVM libraries and tools easily integrate into CI/CD pipelines, automating the verification process.

The main components of Pact JVM include:

1. **Pact JVM Libraries:** Pact JVM provides client and server libraries for different JVM languages (e.g., Java, Kotlin, Scala) that allow developers to create and verify contracts between consumers and providers. The client library is used in consumer applications to define the expected interactions with the provider, generate Pact contract files, and communicate with the Pact Broker to publish the contracts. The server library is used in the provider applications to verify the contracts against the actual behaviour of the API endpoints.
2. **Pact Broker Client:** Pact JVM includes a Pact Broker client that allows applications to interact with a Pact Broker instance, a central repository for storing and sharing contracts between different services.
3. **Pact Gradle Plugin:** Pact JVM provides a Gradle plugin that simplifies the integration of Pact with Gradle-based projects. It allows running Pact tests and publishing contracts to the Pact Broker directly from the build process.
4. **Pact Standalone Verifier:** Pact JVM also includes a standalone verifier tool that can be used to verify contracts without running a provider application. It is useful for running verification checks during the CI/CD process without the need to deploy the actual provider service.

## 3.2. Dockerized Pact Broker

Docker is a software platform that expedites the building, testing, and deployment of applications by packaging them into standardized units called containers (‘What is Docker? | AWS’ 2023). These containers encapsulate all the essential components, such as libraries, system tools, code, and runtime, required for the software to run smoothly. With Docker, deploying and scaling applications in various environments becomes swift and dependable, ensuring the seamless execution of your code.

Docker operates by offering a standardized approach to execute your code. Acting as an operating system for containers, it enables the virtualization of a server's operating system, just as a virtual machine virtualizes server hardware, eliminating the need for direct management. Once Docker is installed on each server, it provides straightforward commands to build, start, or stop containers, streamlining the process of managing and running applications within isolated and portable environments.

### 3.2.1. Docker Compose

Compose is a powerful tool designed for defining and running multi-container Docker application that uses a YAML file to effortlessly configure your application's services. With a single command, you can then create and initiate all the services based on your specified configuration, simplifying the process of managing and launching multiple containers in a coordinated manner.

Compose can operate in various environments, including production, staging, development, testing, and CI workflows. It empowers users with a comprehensive set of commands to manage the entire lifecycle of their applications. These commands enable users to effortlessly start, stop, and rebuild services as needed. The tool also provides an easy way to view the status of running services and stream the log output in real-time.

A Dockerized Pact Broker refers to an instance of the Pact Broker application that is packaged and distributed as a Docker container (‘Dockerized Pact Broker | Pact Docs’ 2023). As mentioned, Pact Broker is the tool used for sharing consumer-driven contracts and verification results among different applications, facilitating seamless contract testing and integration management. Dockerizing the Pact Broker allows users to encapsulate the application and its dependencies within a Docker container, making it easy to deploy and run the Pact Broker in any environment that supports Docker. This approach ensures consistency and portability, as the containerized Pact Broker can be deployed across various systems with minimal setup and configuration, providing a reliable and efficient solution for managing contracts and verification results in a containerized environment. Also, Pact Broker is packaged as a Docker container allows it to be easily integrated into the CI/CD pipeline.

Here's an overview of how the Dockerized Pact Broker works:

Docker Image: The Dockerized Pact Broker is packaged as a Docker image, which contains all the necessary dependencies and configurations to run the Pact Broker service (‘pact\_broker-docker/docker-compose.yml at master · DiUS/pact\_broker-docker’ 2023).

* **Docker Compose or Docker Run:** To deploy the Dockerized Pact Broker, you typically use either Docker Compose or the docker run command. Docker Compose is a tool that allows you to define and manage multi-container applications, while docker run allows you to start a container from a specified image.
* **Exposed Ports**: The Pact Broker runs a web server that listens on a specific port within the container. When running the Pact Broker with Docker, you can map this port to a port on the host system, making the Pact Broker accessible from outside the container.
* **Database Connection:** The Pact Broker requires a database to store the pact files and metadata. In the case of PostgreSQL, which is commonly used with the Pact Broker, you can also run it as a separate Docker container and configure the Pact Broker container to connect to the PostgreSQL container.
* **Configuration:** The Dockerized Pact Broker can be configured through environment variables or a configuration file, allowing you to customize settings such as database connection details, logging, and other configurations.
* **Interaction with CI/CD Tools:** Once the Pact Broker is up and running, it provides a RESTful API that allows interaction with it. CI/CD tools like Jenkins, CircleCI, or Travis CI can use this API to publish pact files generated by consumer tests and retrieve pact files for verification during provider tests.
* **Pact Workflow:** Developers of different services (consumers) create pact files representing the expected interactions with other services (providers). These pact files are published to the Pact Broker. When providers run their tests, they retrieve relevant pact files from the Pact Broker to verify that their services meet the expectations defined by the consumers.

By containerizing the Pact Broker with Docker, you can easily spin up and manage the Pact Broker alongside other services in a consistent and reproducible manner. Docker's isolation also ensures that the Pact Broker operates independently of the host system and other applications, avoiding potential conflicts and simplifying the deployment process.

## 3.3. Jenkins CI/CD Pipeline

Jenkins is a self-contained and open-source automation server that empowers users to automate a wide range of tasks associated with building, testing, delivering, and deploying software (‘Jenkins User Documentation’ 2023). Jenkins Pipeline is a collection of plugins that enables the implementation and integration of continuous delivery pipelines within the Jenkins automation server. These plugins provide robust support for defining and managing complex, end-to-end delivery pipelines, streamlining the process of building, testing, and deploying software in a continuous and automated manner.

The integration of the Pact Broker with Jenkins allows for seamless management of consumer-driven contracts and verification results within the Jenkins automation server. By leveraging plugins and scripts, Jenkins can interact with the Pact Broker to achieve various tasks related to contract testing and deployment.

The integration typically involves the following steps:

1. **Publishing Pacts**: During the CI/CD process, the consumer applications generate Pact files containing the contract information. These files are then published to the Pact Broker, making them available for other applications to use.
2. **Verification**: Once the Pact files are published, Jenkins can trigger the verification process. The provider applications fetch the relevant Pacts from the Pact Broker and execute contract tests against them to ensure compliance with the specified contracts.
3. **Reporting:** The results of the verification process can be captured and reported by Jenkins, allowing the team to monitor contract compliance and identify any issues that need attention.
4. **Integration with CI/CD Pipelines:** Pact Broker integration can be seamlessly incorporated into Jenkins CI/CD pipelines, ensuring that contract testing is an integral part of the development and deployment workflow.

By integrating the Pact Broker with Jenkins, teams can ensure that their applications adhere to the agreed-upon contracts, promoting better collaboration between teams and enhancing the reliability and confidence of application releases.

# Implementation

An online shopping system was implemented, comprising of 3 microservices. A retailer service acted as the provider, responsible for returning data. The consumers were customer service and product service, which retrieved the data from retailer services.

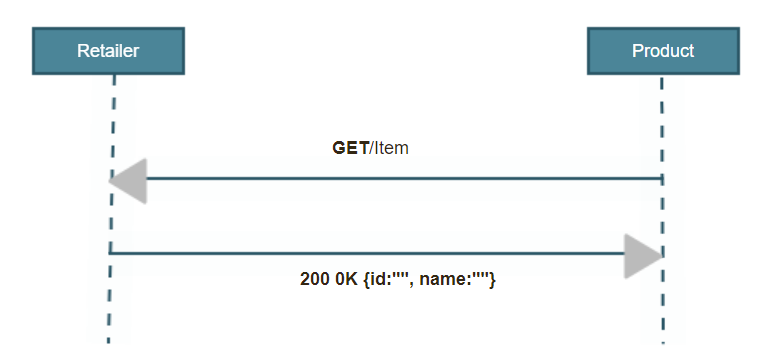


Figure : Product consumer service retrieving item details from Retailer provider service

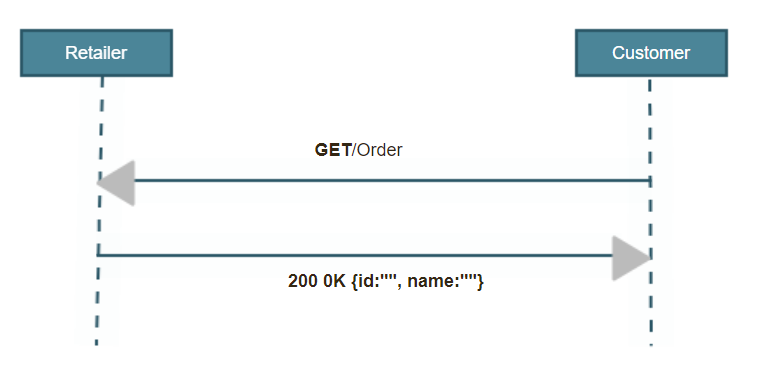


Figure : Customer consumer service retrieving order details from Retailer provider service

The approach involved using Pact JVM within the Spring Boot framework to create tests for both the consumers and the provider. The results of these tests were shared on a Pact Broker which is stored in a Docker Compose container. This was integrated into a Jenkins integration pipeline workflow. The chosen technology stack included Java, JUnit, Gradle, Spring Boot, Docker Compose, and Jenkins.

## 4.1 Writing Contract Test with Pact Framework

The contract tests were authored and executed using the Pact JVM framework offered by the Pact Foundation (‘pact-foundation/pact-jvm: JVM version of Pact.’ 2023). For the online shopping system made for this dissertation, the customer and product services (consumers) send API requests to the retailer service (provider), expecting responses containing order and item details, respectively. These consumers anticipate a successful 200 status code in response to a GET request made to the provider. A retailer controller class is created which exposes two APIs; getItemDetails and getOrderDetails responsible for returning the data to product and customer services respectively.



Figure : Methods in Retailer Controller Class to return item and order details to consumers

Upon successfully passing the unit tests by consumers against the simulated provider, a contract file will be generated on the consumer side. Provider test will ensure provider APIs are satisfying this contract shared by the consumer. Within the Pact framework, the contract is referred to as a "pact." It is represented as a JSON file containing interactions. Each interaction comprises a request and the corresponding expected response.

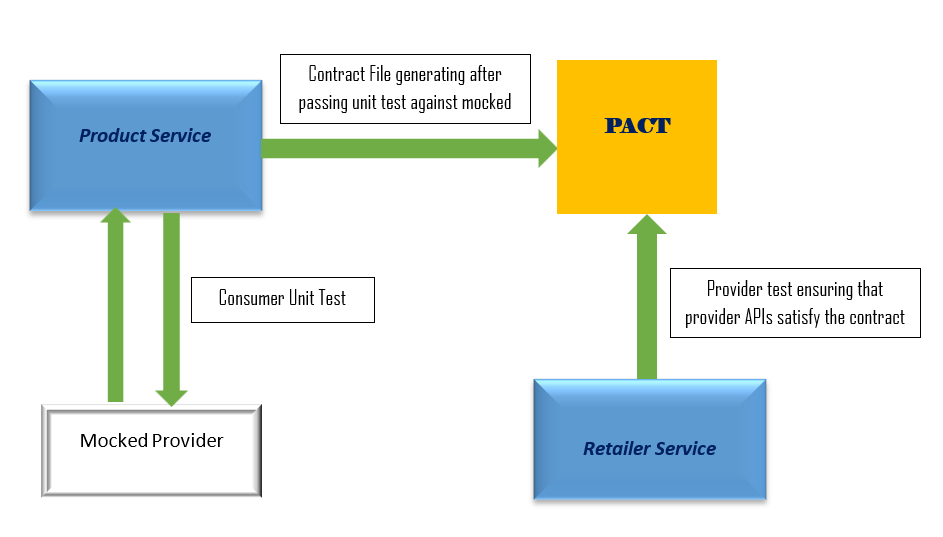


Figure : Pact Framework

### 4.1.1. Consumer – Creating the Contract

A unit test is created and executed by the consumers. The unit test ensures that the code can appropriately handle the expected responses from the provider and it automatically generates the Pact file. As mentioned, the system consists of two consumer services, ‘Product’ and ‘Customer’. The following implementation process will concentrate on the working of Product service for the demonstration purpose. Consequently, the class and method names mentioned in this context pertain specifically to the product service. The exact same implementation will be followed for the customer service, involving the development of equivalent classes and methods.

The 'ProductService' class was created for the product consumer, where it internally consumed the 'getItemDetail' API from the retailer service. The 'build.gradle' file was configured to publish the contracts to a specified location and PactBroker URL. A contract test named 'RetailerContractTest' was developed, for our unit and contact testing to ensure that provider satisfies the requirements.

We will write the unit test for the ProductService class, which serves as the direct interface with the provider. In this scenario, the ProductService's getItemDetail method utilizes a RestTemplate to communicate with the user service. It then parses the response to create an Item object, which is subsequently returned. The unit test will verify the proper functioning of this interaction and, incidentally, generate the Pact file.

The unit test will carry out the below steps;

1. Initialize a mock server to simulate the provider's behaviour based on the specified interactions.
2. Invoke the getItemDetail method, which in turn, will make a call to the mocked provider.
3. Verify and assert the properties of the returned Item object to ensure it aligns with the expected behaviour based on the mocked provider's response.
4. Based on the interactions recorded during the unit test, the Pact file will be automatically generated. It will contain the details of the requests made to the mocked provider and the corresponding expected responses.

The interactions will be defined in a separate method, marked with the @Pact annotation. Steps 1 and 4 will be facilitated by the Pact framework's PactProviderRuleMk2 JUnit rule, while steps 2 and 3 will be executed within a standard @Test method. This setup allows for the creation of the Pact file based on the recorded interactions and ensures the test execution against the mocked provider.

The initial step involves creating the unit test class and adding the test method. Following that, interactions are defined by creating a method annotated with @Pact, specifying the consumer name. This method will return the contract's description using the pact-jvm Lambda DSL (‘pact-foundation/pact-jvm: JVM version of Pact.’ 2023), which allows to describe the expected interactions between the consumer and provider in a clear and expressive manner.

The final step involved setting up the mock server. To achieve this, PactProviderRuleMk2 rule was added to the class. The test method was annotated with @PactVerification and the name of the preceding method (the one that defined the interactions). This annotation will guide the Pact provider rule to initiate the mock server with the interactions outlined in the specified method.



Figure : Consumer unit test class to generate pact file

Upon running the test, it will generate the following Json file which is called as the Pact file containing the recorded interactions between the consumer and the provider. Once the Pact file is successfully created from the consumer side, next is to verify this test on the provider side. The provider will use the same Pact file to validate its behaviour and ensure that it meets the expectations set by the consumer.

### 4.1.2. Provider – Verifying the Contract

Moving on to creating tests on the provider side to verify that the contracts were fulfilled, Spring Boot integration tests were utilized. These tests will ensure that the provider adheres to the specified contract, validating that the responses it provides align with the expectations set by the consumer in the Pact file. To conduct the test, a standard Spring Boot web integration test was developed, leveraging the SpringRestPactRunner JUnit runner. This combination allows to verify the provider's behaviour against the Pact contracts. The Spring Boot web integration test ensures that the provider's endpoints respond as expected, while the SpringRestPactRunner facilitates the verification process, checking whether the responses align with the established contracts (‘pact-foundation/pact-jvm: JVM version of Pact.’ 2023).

Within the class, below elements are specified:

1. @Provider: This annotation specifies the provider name and informs which Pact files to load for verification.
2. @PactFolder: This annotation indicates where the Pact files are located. We create a directory named "pacts" and copy the Pact file generated by the consumer into this directory. This ensures that the provider can access and verify the contract specified by the consumer.
3. The Target: This defines the location where the interactions will be executed and the responses verified. For Spring Boot integration tests, SpringBootHttpTarget is used. These tests run against the application that is started by the integration test on a random port.
4. Methods for Provider States: For each provider state mentioned in the contract, corresponding methods are created in the test class. These methods enable to configure the desired provider state, such as creating the required user in the database or mocking the service to provide the necessary user data. By ensuring the correct provider state, the interactions can be validated in the Pact file against the actual behaviour of the provider during the integration test.



Figure : Provider verification class to verify the contract

Each contract has a unique state associated with it. Since there can be multiple API interactions between the provider and consumer, a state is necessary to distinguish each interaction as unique. For the product consumer, in this case 'Get item details' for retrieving the item details.

## 4.2. Integration in Build Pipelines

In the previous section, Pact framework was implemented to carry out consumer-driven testing with Java. Now, we will focus into the workflow of Consumer Driven Contract testing within the context of continuous integration (CI) and explore how we can integrate it into our build pipelines. For this, our setup involves Pact Broker, Docker Compose and Jenkins. The Pact Broker stores all the contracts created in a database. It knows for each consumer version which provider version has - or has not - verified the contract. Pact Broker is responsible for validating the contract between the provider and consumer, verifying its success or detecting any breaks. Docker Compose will then orchestrate the deployment of Pact Broker. Subsequently, Jenkins will initiate the build process only when the publication of the contract is confirmed as successful. This setup was made to achieve the following objectives:

* Refrain from deploying provider service if it breaks any of the contracts created by consumer
* Refrain from deploying consumer service if it cannot consume the provider API

### 4.2.1. Docker Compose Configuration

Docker compose configuration file was created ‘docker-compose-pactbroker.yml’ to configure Pact Broker service, PostgreSQL database for the Broker and the Jenkins service. The docker image "pact-foundation/pact-broker" was pulled from the pact docker repository (‘Dockerised Pact Broker’ 2023; ‘pact\_broker-docker/docker-compose.yml at master · DiUS/pact\_broker-docker’ 2023) to set up the Pact Broker and PostgreSQL.



Figure : Docker Compose File

Here is the breakdown of three services used in the configuration fie:

1. **pact-broker-postgres:**

* This service is using the official PostgreSQL Docker image.
* It includes health checks to ensure the PostgreSQL database is healthy.
* It maps the host port 5432 to the container port 5432 to allow external access to the database.
* The PostgreSQL data is stored in a Docker volume named postgres-volume.
* Environment variables are provided to configure the PostgreSQL database.

1. **pact-broker:**

* This service uses the official Pact Broker Docker image.
* It maps the host port 9292 to the container port 9292 to make the Pact Broker accessible from the host.
* It depends on the pact-broker-postgres service, indicating that it requires the PostgreSQL service to be up before starting.

1. **jenkins:**

* It maps the host ports 8080 and 50000 to the respective container ports for Jenkins.
* It depends on the pact-broker service, ensuring the Pact Broker service is up before starting.
* An environment variable PACT\_BROKER\_BASE\_URL is set to make Jenkins communicate with the Pact Broker on port 9292.

Once the configuration has been set up, it is ready to be executed when the consumer publishes the contract to the Pact Broker.

### 4.2.2. Consumer – Publishing Contract to Pact Broker

To publish the contract, Pact configuration is made within the Gradle build file. It outlines the settings for publishing contracts to a Pact Broker. The pactDirectory parameter indicates the directory where the generated pact files are located, within the "target/pacts" directory. The pactBrokerUrl parameter specifies the URL of the Pact Broker where the contracts will be published, which is set to localhost 9292 for local testing.



Figure : Pact configuration in build.gradle

By executing the command 'docker-compose up' (‘docker compose up’ 2023) , the Docker Compose file will be run, resulting in the consumer contract becoming accessible through the Pact Broker interface in the specified URL.

### 4.2.3. Provider – Publishing Verified Result to Pact Broker

After the contract has been successfully published to the Pact Broker by the consumer, the provider will execute the test class ProductPactTests and CustomerPactTests for corresponding consumer verification. The test result will then be published to the Pact Broker. Pact Broker URL was configured within the test class.

### 4.2.4. Jenkins Configuration

With the Pact Broker now prepared, with all the contracts and verification results, integration into the Jenkins pipeline can be achieved to finalize the CI/CD integration. Below steps are added to the Jenkins pipeline script for the product consumer, by which whenever the Jenkins job is triggered, it will run the unit test, create the pact file and upload to the Pact Broker.



Figure : Provider Jenkins configuration

Below step is added to the Jenkins pipeline script for the retailer provider, by which whenever the Jenkins job is triggered, it will verify the contract published in the pact broker and share the verification result back to the pact broker.



Figure : Provider Jenkins configuration

Certainly, in the realm of testing, a critical aspect is to account for negative scenarios ensure that the system can handle unexpected conditions. For instance, if the Provider omits a field from the response, the contract test should accurately detect this discrepancy, causing the entire build process to fail and effectively halting the deployment. Consumer's build should be aware of whether the Provider has successfully validated the contracts prior to initiating deployment. This can be facilitated by incorporating the Pact CLI's ‘can-i-deploy’ command into the consumer build pipeline (‘can-i-deploy | Pact Docs’ 2021). If the contracts aren't validated successfully, this command leads to terminating the deployment process.



Figure : 'can-i-deploy' Jenkins configuration

Thus, the Jenkins workflow for ` consumer will go as follows:

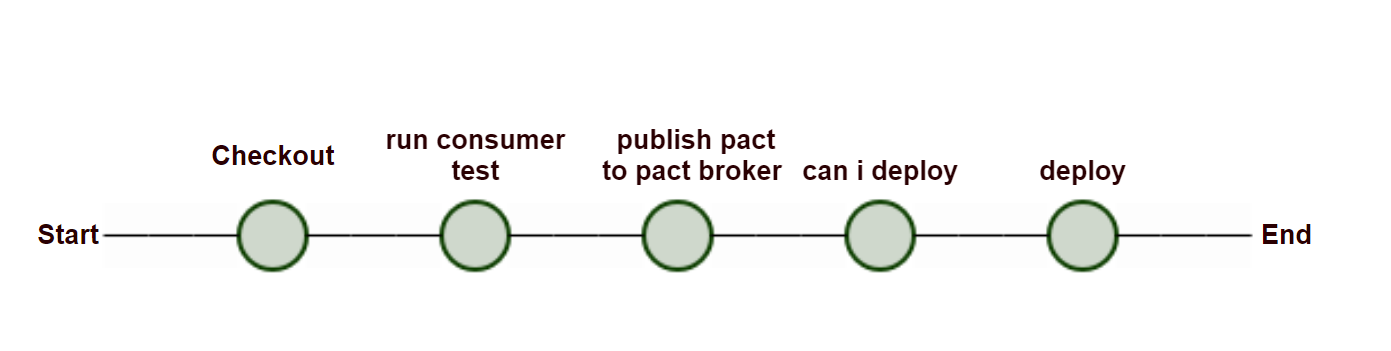


Figure : Consumer Jenkins workflow

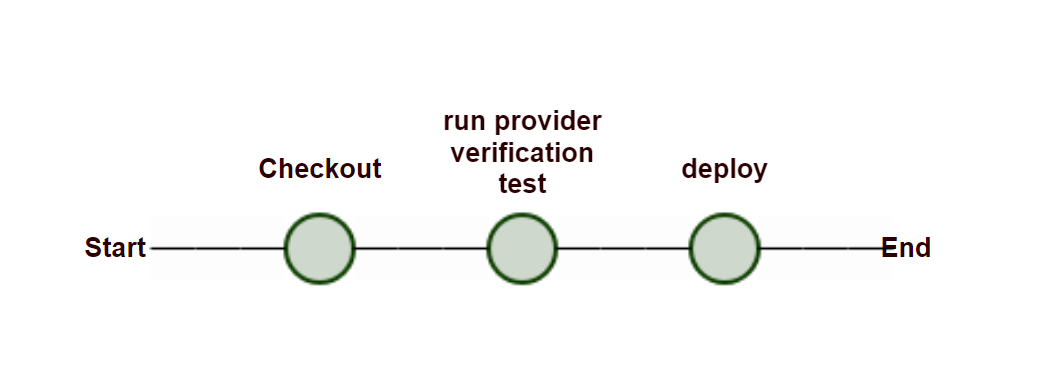


Figure : Provider Jenkins workflow

## 4.3. Chapter Conclusion

The implementation chapter highlighted the successful establishment of testing an online shopping system composed of three microservices. The retailer service acted as provider, serving the customer service and product service as consumers. By utilizing Pact JVM within the Spring Boot framework, robust tests were created for both consumers and the provider, ensuring data consistency and reliability. The results of these tests were effectively managed through a Pact Broker housed in a Docker Compose container, integrated into a Jenkins pipeline workflow. The implementation's success hinged on the careful selection of specific technologies including Java, JUnit, Gradle, Spring Boot, Docker Compose, Pact Broker and Jenkins. Next chapter will concentrate on the testing of this whole scenario and the subsequent analysis of the results.

# Test and Results

This chapter delves into description of the testing process and analysis of the obtained results, on the effectiveness and robustness of our implemented solution. The strategy begins by initially focusing on positive test scenarios involving interactions between the two consumers (product and customer) and the provider (retailer). The objective is to determine whether these tests pass successfully. Subsequently, the approach shifts to negative scenario where a data field will intentionally be omitted. The aim is to observe whether these tests result in failure.

## 5.1 Consumer Unit Test

### 5.1.1. Product Unit Test

Following the execution of the unit tests performed by the product consumer against the mocked provider, the tests were deemed successful (Figure 17). As a result, a pact file (Figure 21) was generated within the designated target folder as shown in Figure 18. This pact file holds the contract between the product service and the retailer service interactions.

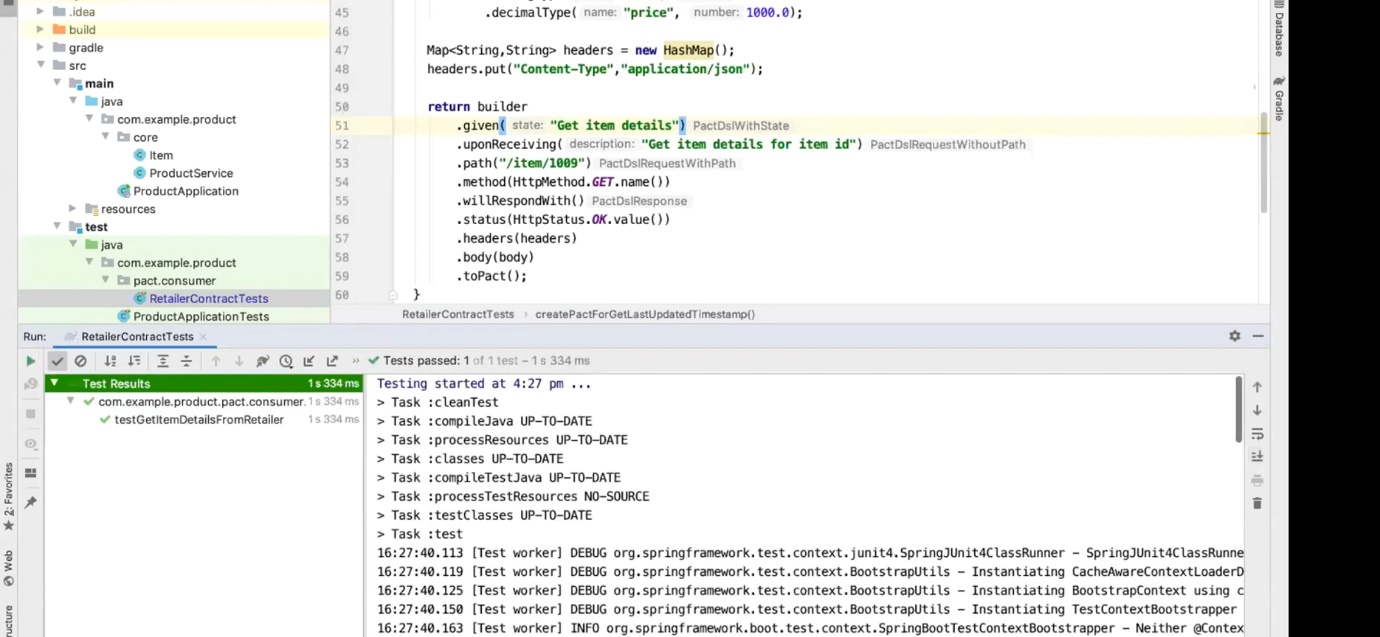




Figure : Running product unit test

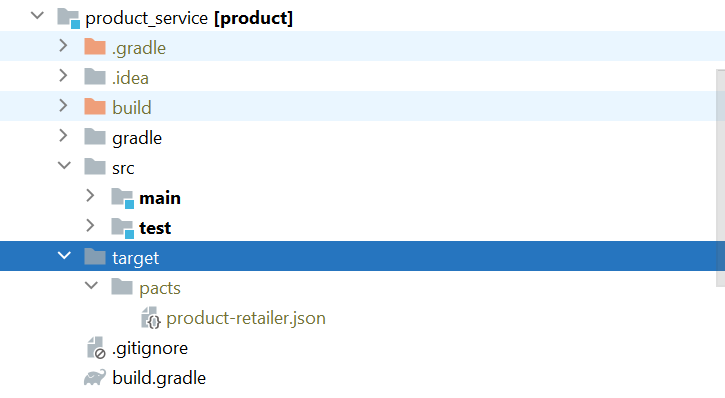




Figure : Product target folder

According to the implemented consumer workflow, the created pact is successfully published in the pact broker, residing in a state without verification, as the execution of the provider test is pending (Figure 19).

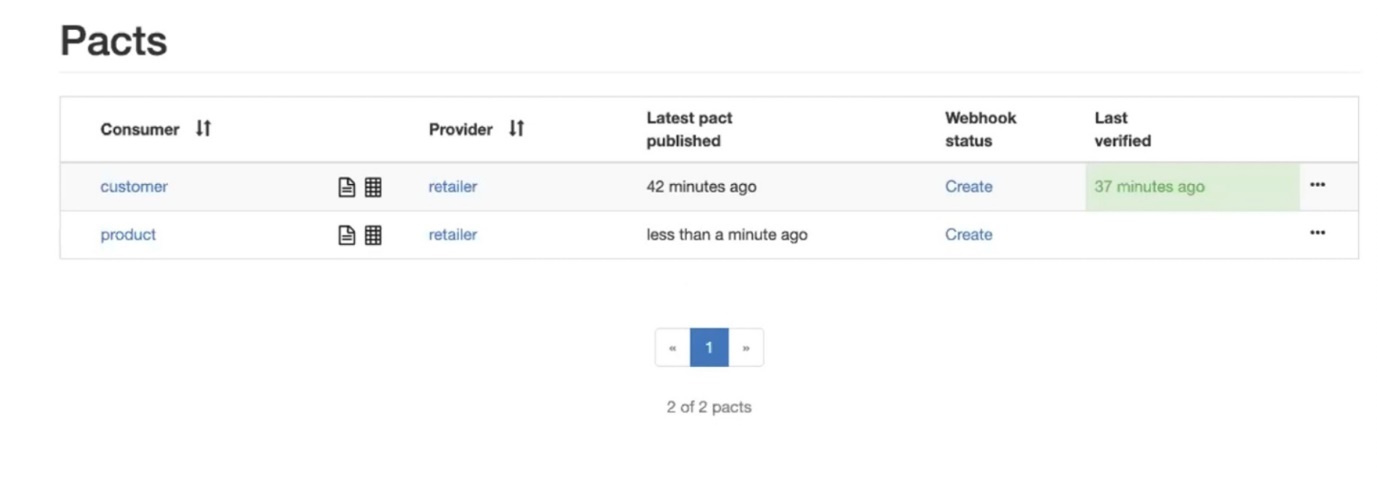


Figure : Pact broker publishing product pact

The Pact Broker provided an elaborate account of the created pact as shown in Figure 20, comprising comprehensive interaction details of the interactions between the product and retailer services.

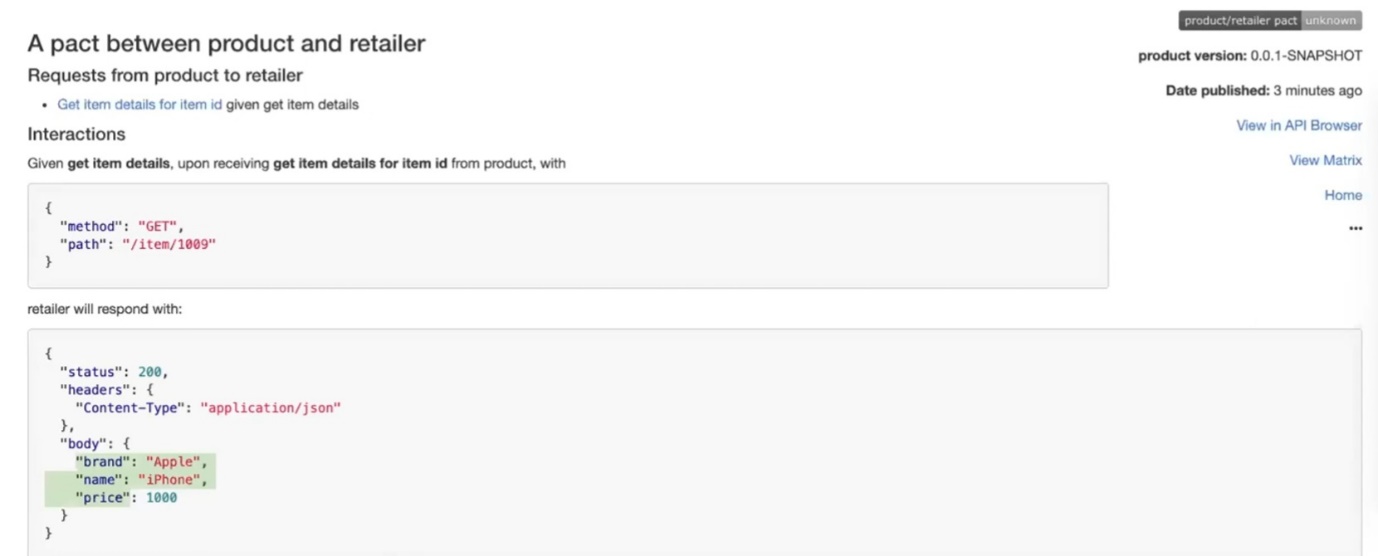


Figure : Service Interactions details provided by pact broker



Figure : Product Pact File

### 5.1.2. Customer Unit Test

The same testing procedures were applied to the customer service as of product service, resulting in the creation of a pact file outlining the interactions between the customer and retailer services. This newly generated pact file was successfully published to the pact broker, as per the implemented consumer workflow.



Figure : Pact broker publishing customer pact

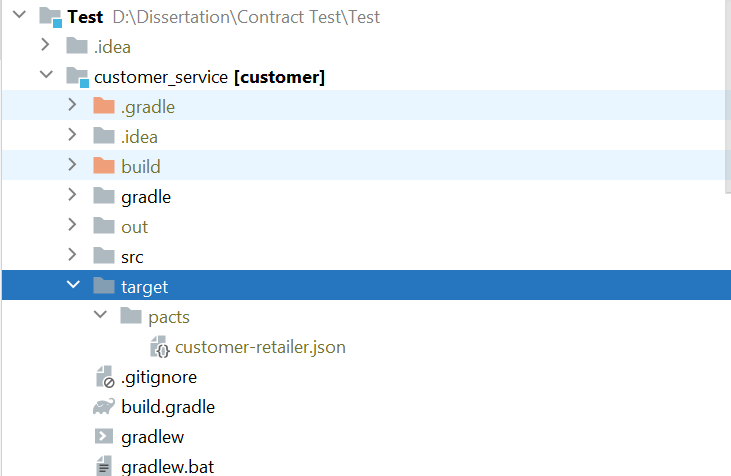


Figure : Customer target folder



Figure : Customer Pact File

## 5.2. Provider Verification Test

With the pact published in the pact broker, now the provider workflow can be executed. In this stage, the provider retailer service can retrieve the pact from pact broker and carry out the verification tests to ensure the contract is fulfilled.

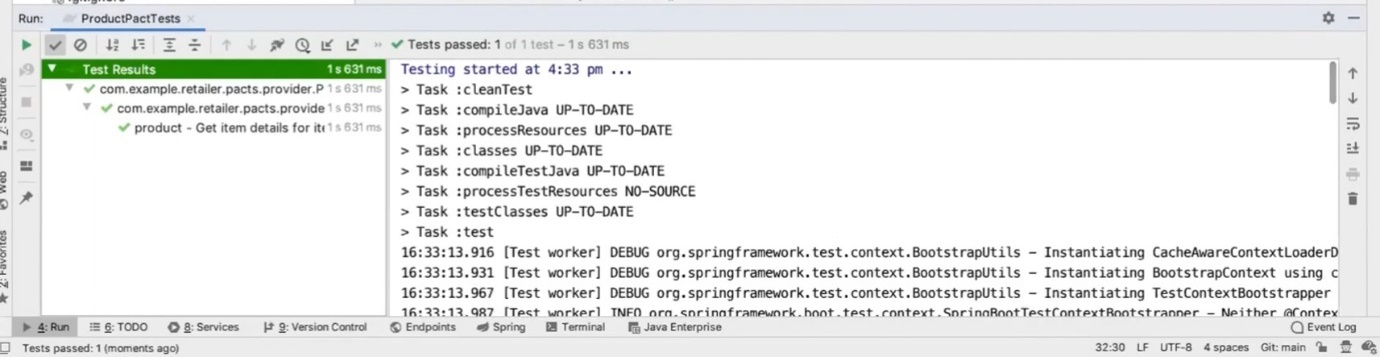


Figure : Retailer verification test

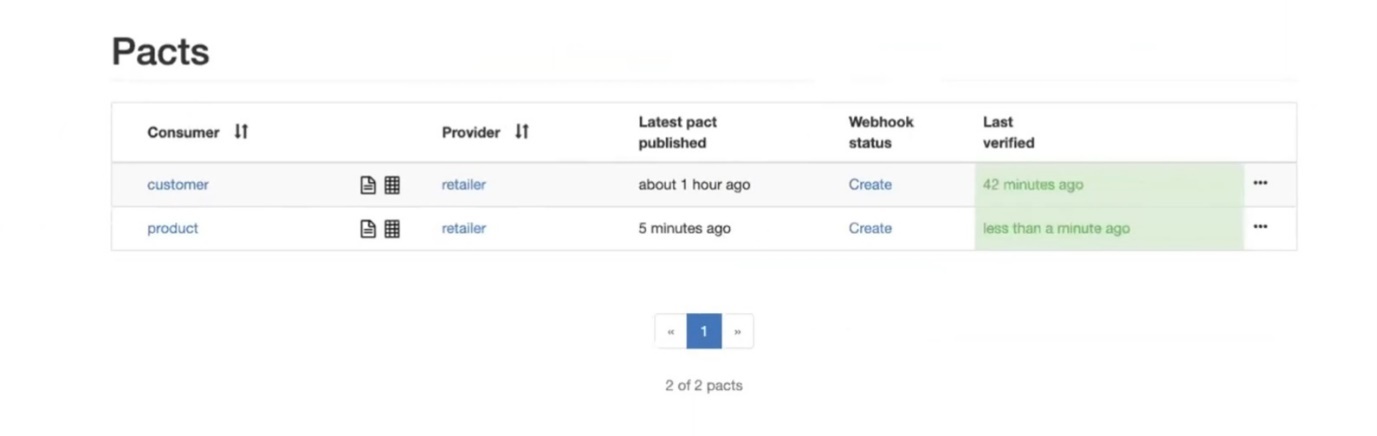


Figure : Product pact verified in pact broker



Figure : Customer pact verified in pact broker

## 5.3 Negative Test

The negative test scenario, involving the intentional omission of a field from the provider's response, was executed. Following the implemented workflow, the test result indicated failure, which was subsequently recorded and published in the pact broker. For this ‘price’ field was removed from the response by the retailer service. On running the verification test, since the field is missing the contract is broken and eventually test got failed.

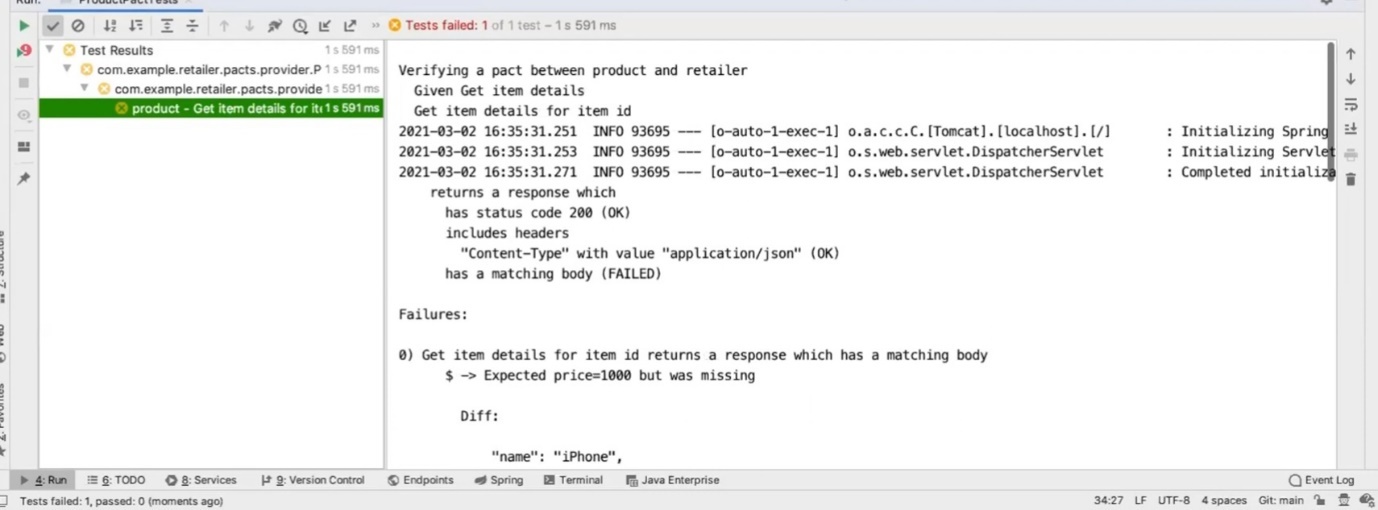


Figure : Negative test failing

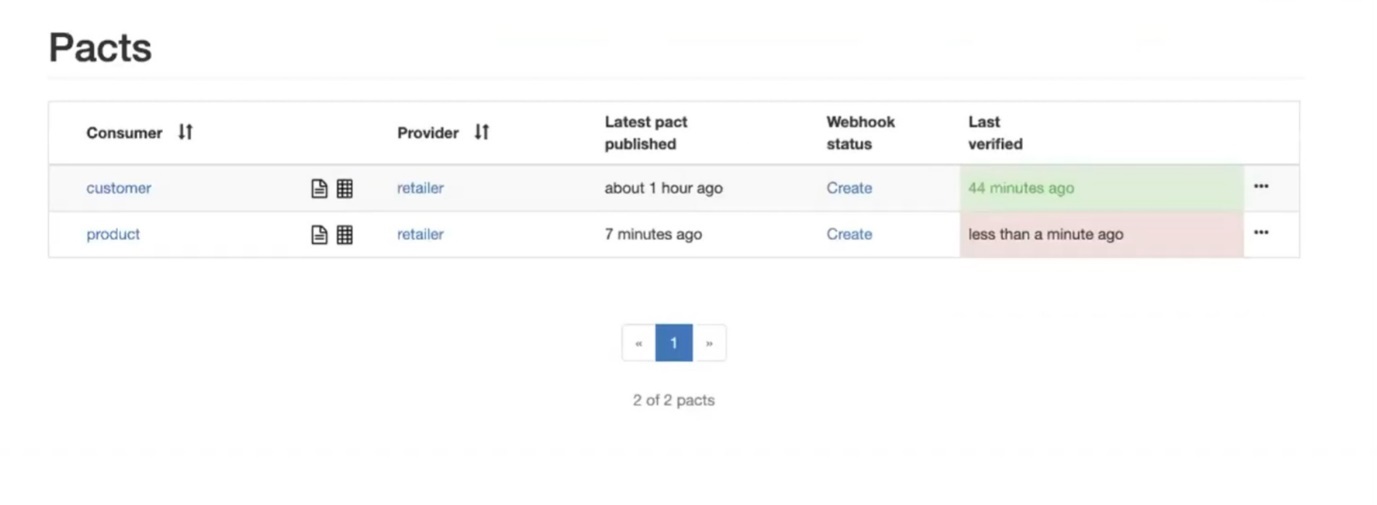


Figure : Test failure published in pact broker

# Conclusions

The objective of this dissertation was to examine how contract testing frameworks with CI/CD integration tools can optimize the efficiency of testing microservices within a distributed system. To accomplish this, an in-depth exploration was undertaken regarding the paradigm shift from monolithic architectural approaches to the adoption of microservices technology, along with an examination of the advantages this transition offers. Despite the advantages it offers, the adoption of microservices technology introduced a high level of complexity within distributed systems. This complexity has led to the emergence of consumer-driven contract testing as a viable solution. Substantial research has been dedicated to investigating this testing methodology as a means to effectively address the challenges arising from the distributed nature of microservices architecture. An in-depth exploratory study was carried out to thoroughly investigate the relationship between the principles and practices of DevOps and the emerging paradigm of microservices architecture. Ultimately, a practical examination was conducted to empirically evaluate the efficacy of consumer-driven contract (CDC) testing in validating a system comprising multiple microservices. This evaluation involved the utilization of CDC test frameworks in conjunction with CI/CD integration tools.

## Microservice and DevOps Paradigm Shift

This dissertation initially delved into the transition from traditional monolithic architectures to the modern adoption of microservices technology. It involved a comprehensive investigation of the advantages that come with this paradigm shift, including improved scalability, flexibility, and development agility. Microservices architecture facilitates a modular approach where individual components can be scaled independently. This fine-grained scalability enables optimal resource allocation, allowing organizations to efficiently handle varying workloads without the need to scale the entire application. As a result, the system can readily adapt to fluctuating demands while maintaining optimal performance. Its architecture enables the decoupling of services, affording the ability to develop, deploy, and update them independently. This agility in managing services translates to faster development cycles, as teams can focus on specific functionalities without affecting the entire application. Consequently, organizations can swiftly respond to evolving market dynamics and evolving customer requirements. It allows development teams to work concurrently on separate components contributing to development agility, accelerating the overall development lifecycle. This concurrent development fosters quicker iteration, testing, and deployment, aligning seamlessly with agile methodologies. Consequently, organizations can maintain a competitive edge by rapidly rolling out new features and enhancements. Furthermore, microservices architecture supports fault isolation and system resilience. In monolithic systems, a single flaw can propagate throughout the entire application. However, microservices' modular nature confines issues to specific components, limiting their impact. This containment enhances the overall robustness of the system and ensures smoother operations even in the face of failures. By embracing scalability, flexibility, development agility, and enhanced fault tolerance, organizations position themselves at the forefront of software innovation, where rapid adaptation and responsiveness are paramount.

The emergence of DevOps pipelines revolutionizes software development by fostering a culture of collaboration, automation, and continuous improvement. These pipelines empower organizations to release high-quality software faster, more frequently, and with enhanced reliability, thereby staying competitive and responsive in the ever-evolving technology landscape. The integration of DevOps practices and microservices architecture fosters a development environment that is highly conducive to rapid iteration. Development teams can work on individual microservices concurrently, leading to faster feature development and shorter time-to-market for new functionalities. DevOps emphasizes automation, testing and continuous integration/continuous deployment (CI/CD), enabling frequent and automated deployments. When combined with microservices' modular nature, this approach ensures that updates and improvements can be swiftly deployed to specific services without affecting the entire application. Individual microservices can be developed, tested, and deployed independently, reducing bottlenecks and enabling parallel workflows. This decomposition aligns with the DevOps emphasis on breaking down monolithic processes for greater agility. DevOps breaks down barriers between development, operations, and other stakeholders. Microservices' modular structure promotes cross-functional collaboration, as each service is owned by a dedicated team. This alignment fosters a culture of shared ownership and collaboration across the entire development lifecycle.

In summary, the impact of Microservices on DevOps principles empowers organizations to accelerate development, ensure reliable deployment, and respond swiftly to changing market dynamics, ultimately leading to enhanced software delivery and user satisfaction.

## Contract Testing Frameworks

While the adoption of microservices technology brings forth significant benefits, it concurrently introduces a notable degree of complexity within distributed systems. This complexity has given rise to the recognition of consumer-driven contract testing as a practical and effective solution. For this dissertation, a shopping system encompassing multiple microservices was developed and utilized as a practical scenario to evaluate the effectiveness of contract testing frameworks in validating interactions among the services.

### 6.2.1. Pact Framework

The testing process utilized the Pact framework, which provided a comprehensive suite of components to facilitate the complete lifecycle of Consumer-Driven Contracts. These components encompassed everything from the initial definition of contracts to their sharing, validation, and management. By leveraging these components, the testing approach facilitated effective communication and integration between microservices.

### 6.2.2. Docker Compose

Docker Compose proved to be a key tool for enhancing CDC testing with Pact. It creates a separate space where microservices, their links, and interactions can be set up together, resembling real situations. This aids in making testing scenarios more authentic. When included in continuous integration CI/CD processes, it helps to automatically check contracts in a setup that looks like the real world. With configurations kept in check to ensure everyone's on the same page and the ability to mimic larger workloads, Docker Compose strengthens CDC testing with Pact. It acts as a versatile solution that simplifies creating thorough, automatic, and standardized testing setups, boosting the dependability and precision of contract checks in systems built on microservices.

### 6.2.3. Pact Broker

Pact Broker played a vital tool in advancing CDC testing with Pact. Serving as a centralized hub, it facilitated the sharing and management of contracts between different services, fostering collaboration and ensuring the consistency of interactions. By enabling version control and easy access to contract history, the Pact Broker streamlines the process of maintaining and evolving contracts as microservices change. This collaborative platform enhances communication, reduces integration hitches, and promotes transparency, ultimately bolstering the reliability and effectiveness of CDC testing in microservices-based systems.

Thus, by conducting the practical analysis, certain observations were uncovered. While these findings were derived from this compact system, they hold the potential to be extended and applied to larger-scale systems.

* **Isolated Validation:** CDC testing focuses on validating the communication and interactions between services in isolation. This approach allows each service to be thoroughly tested without the need to deploy the entire system, resulting in efficient testing that doesn't impact other components.
* **Enhanced Collaboration:** CDC testing promotes collaboration between service providers and consumers. Contracts, which define the expected behavior of services, act as shared agreements. This collaboration ensures that both sides understand and adhere to the established contracts, minimizing misunderstandings and potential conflicts.
* **Reduced Integration Issues:** By validating interactions based on contracts, CDC testing reduces integration issues that may arise when services don't communicate as expected. Contract breaches are identified early, preventing compatibility problems and enabling timely resolution.
* **Regression Testing Confidence:** As services evolve independently, CDC testing ensures that existing contracts remain intact even as changes are made. This provides confidence that new updates won't unintentionally break existing functionalities, thus enhancing regression testing efficacy.
* **Comprehensive Documentation:** Contracts serve as comprehensive documentation of service interactions. This documentation aids in understanding each service's expected behavior and facilitates onboarding new team members or external collaborators.
* **Minimized Communication Overhead**: CDC testing reduces the need for continuous direct communication between service providers o consumers to ensure compatibility. Once contracts are established, the testing framework acts as an automated mediator, relieving the need for constant coordination.
* **Support for CI/CD Integration:** CDC testing aligns seamlessly with continuous integration practices. It can be incorporated into the CI/CD pipeline to ensure that service interactions are validated with each code change, supporting a consistent testing approach.

## 6.3. Testing and CI/CD Integration

The integration of Pact CDC testing framework with Continuous Integration and Continuous Deployment CI/CD practices presents a powerful synergy that enhances the reliability and quality of microservices-based systems. By automating contract verification within the CI/CD pipeline, potential issues are detected early, enabling swift resolutions and minimizing disruptions. This collaboration ensures that microservices interactions align with expected behaviours and prevents regression. Furthermore, the streamlined integration expedites development cycles, improves collaboration between teams, and promotes a culture of consistent testing and deployment. Altogether, the harmonious integration of Pact CDC framework and CI/CD practices empowers organizations to deliver dependable, well-tested microservices applications with agility and confidence.

### 6.3.1. Pact Broker and Jenkins

Integrating Pact Broker with CI/CD practices creates a strong synergy that enhances the integrity and efficiency of microservices-based systems. The Pact Broker serves like a team hub, facilitating the effortless exchange, versioning, and management of contracts among diverse teams and services. This helps everyone stay on the same page as services change, making communication smooth, stopping problems with putting things together, and keeping things open. Combining Jenkins and the Pact Broker for Pact CDC testing in CI/CD processes creates a potent alliance that enhances the reliability and speed of microservices systems. This alliance accelerates the development lifecycle by providing a centralized platform where contracts can be effortlessly shared and accessed. During the CI/CD pipeline, the Pact Broker automatically orchestrates the verification of contracts against real-world interactions, producing tangible results and insights. This real-time validation ensures that the interactions between microservices remain compliant with the defined contracts. These results are then seamlessly integrated into the CI/CD feedback loop, enabling developers to promptly address any discrepancies or issues. As a result, this collaboration enhances collaboration, minimizes regression, and cultivates a culture of continuous improvement, ultimately enhancing the reliability and effectiveness of microservices-based systems.

## Limitations

A limitation of this dissertation lies in the absence of the planned implementation for the live deployment of microservices through containerization. While the utilization of containerization for the test results from Pact Broker for continuous integration is established, the subsequent stage involving the actual deployment of services remains unrealized due to time constraints. This limitation holds back the scope of a comprehensive evaluation of the complete process, directing the dissertation's focus primarily towards initial testing stages and continuous integration. The potential to include real deployment would have rounded off the entire DevOps pipeline, remains constrained due to this factor.

# References

10 Best Deployment Tools for DevOps in 2023 [online] (2023) available: https://www.knowledgehut.com/blog/devops/devops-deployment-tools [accessed 30 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.

Cito, J., Leitner, P., Fritz, T., and Gall, H.C. (2015) ‘The making of cloud applications: an empirical study on software development for the cloud’, in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, New York, NY, USA: Association for Computing Machinery, 393–403, available: https://doi.org/10.1145/2786805.2786826.

Consumer-Driven Contracts: A Service Evolution Pattern [online] (2023) *martinfowler.com*, available: https://martinfowler.com/articles/consumerDrivenContracts.html [accessed 23 Jun 2023].

Dai, G., Bai, X., Wang, Y., and Dai, F. (2007) ‘Contract-Based Testing for Web Services’.

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.

‘Design, monitoring, and testing of microservices systems: The practitioners’ perspective’ (2021) *Journal of Systems and Software*, 182, 111061, available: https://doi.org/10.1016/j.jss.2021.111061.

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].

Docker Compose Up [online] (2023) *Docker Documentation*, available: https://docs.docker.com/engine/reference/commandline/compose\_up/ [accessed 14 Aug 2023].

‘Dockerised Pact Broker’ (2023) available: https://github.com/pact-foundation/pact-broker-docker [accessed 13 Aug 2023].

Dockerized Pact Broker | Pact Docs [online] (2023) available: https://docs.pact.io/pact\_broker/docker\_images [accessed 21 Jul 2023].

EBay and Lastminute.Com Adopt Contract Testing to Drive Architecture Evolution [online] (2023) *InfoQ*, available: https://www.infoq.com/news/2023/05/ebay-contract-testing-evolution/ [accessed 3 Jun 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].

From Monolith to Microservices: A Dataflow-Driven Approach | IEEE Conference Publication | IEEE Xplore [online] (2023) available: https://ieeexplore.ieee.org/document/8305969 [accessed 27 Jun 2023].

Gluck, A. (2020) Introducing Domain-Oriented Microservice Architecture [online], *Uber Blog*, available: https://www.uber.com/en-VN/blog/microservice-architecture/ [accessed 27 Jun 2023].

Hasselbring, W. and Steinacker, G. (2017) ‘Microservice Architectures for Scalability, Agility and Reliability in E-Commerce’, in *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, Presented at the 2017 IEEE International Conference on Software Architecture Workshops (ICSAW), Gothenburg, Sweden: IEEE, 243–246, available: https://doi.org/10.1109/ICSAW.2017.11.

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].

IDC: The Premier Global Market Intelligence Firm. [online] (2023) *IDC: The premier global market intelligence company*, available: https://www.idc.com/ [accessed 7 Jul 2023].

Introduction | Pact Docs [online] (2022) available: https://docs.pact.io/ [accessed 21 Jul 2023].

Jenkins User Documentation [online] (2023) *Jenkins User Documentation*, available: https://www.jenkins.io/doc/ [accessed 21 Jul 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 Franch, X., Männistö, T. and Martínez-Fernández, S., eds., *Product-Focused Software Process Improvement*, Lecture Notes in Computer Science, Cham: Springer International Publishing, 497–512, available: https://doi.org/10.1007/978-3-030-35333-9\_35.

Lewis, J. and Fowler, M. (2023) Microservices [online], *martinfowler.com*, available: https://martinfowler.com/articles/microservices.html [accessed 27 Jun 2023].

Lightstep [online] (2018) available: https://go.lightstep.com/global-microservices-trends-report-2018.html [accessed 7 Jul 2023].

Mazlami, G., Cito, J., and Leitner, P. (2017) ‘Extraction of Microservices from Monolithic Software Architectures’, in *2017 IEEE International Conference on Web Services (ICWS)*, Presented at the 2017 IEEE International Conference on Web Services (ICWS), 524–531, available: https://doi.org/10.1109/ICWS.2017.61.

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].

*Microservices at Netflix Scale: Principles, Tradeoffs & Lessons Learned • R. Meshenberg • GOTO 2016* [online] (2016) available: https://www.youtube.com/watch?v=57UK46qfBLY [accessed 27 Jun 2023].

Muthu. (2019) ‘Contract Test + Spring Boot for Microservices Architecture’, *Automationcalling*, available: https://automationcalling.com/2019/11/10/contract-test-spring-boot-for-microservices-architecture/ [accessed 21 Jul 2023].

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

Newman, S. (2018) *BUILDING MICROSERVICES : Designing Fine-Grained Systems*, O’Reilly Media, Inc.

Newman, S. (2019) *Monolith to Microservices : Evolutionary Patterns to Transform Your Monolith*, Sebastopol: O’Reilly Media, Inc.

Nyman, R. (n.d.) ‘Consumer-Driven Contract Testing: A Framework and Pilot Implementation’.

Oberoi, A. (2023) What Is Continuous Deployment in DevOps? [online], available: https://insights.daffodilsw.com/blog/what-is-continuous-deployment-in-devops [accessed 30 Jun 2023].

Overview | Pact Docs [online] (2021) available: https://docs.pact.io/implementation\_guides/jvm [accessed 21 Jul 2023].

Pact\_broker-Docker/Docker-Compose.Yml at Master · DiUS/Pact\_broker-Docker [online] (2023) *GitHub*, available: https://github.com/DiUS/pact\_broker-docker/blob/master/docker-compose.yml [accessed 7 Aug 2023].

Pact-Foundation/Pact-Jvm: JVM Version of Pact. [online] (2023) available: https://github.com/pact-foundation/pact-jvm [accessed 5 Aug 2023].

Pahl, C. (2015) ‘Containerization and the PaaS Cloud’, *IEEE Cloud Computing*, 2(3), 24–31, available: https://doi.org/10.1109/MCC.2015.51.

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.

Selleby, F. (n.d.) ‘Creating a Framework for Consumer-Driven Contract Testing of Java APIs’.

Sharing Pacts with the Pact Broker | Pact Docs [online] (2023) available: https://docs.pact.io/getting\_started/sharing\_pacts [accessed 21 Jul 2023].

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.

Soldani, J., Tamburri, D.A., and Van Den Heuvel, W.-J. (2018) ‘The pains and gains of microservices: A Systematic grey literature review’, *Journal of Systems and Software*, 146, 215–232, available: https://doi.org/10.1016/j.jss.2018.09.082.

Spring Cloud Contract [online] (2023) *Spring Cloud Contract*, available: https://spring.io/projects/spring-cloud-contract [accessed 31 Jul 2023].

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

Testing Strategies in a Microservice Architecture [online] (2023) *martinfowler.com*, available: https://martinfowler.com/articles/microservice-testing/ [accessed 5 Jul 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].

Thirion, Y. (2019) Pact Broker : The Missing Piece of Your Consumer-Driven Contract Approach (Part 3/3) [online], *AgilePartner*, available: https://www.agilepartner.net/en/pact-broker-the-missing-piece-of-your-consumer-driven-contract-approach-part-3/ [accessed 21 Jul 2023].

Using Can-i-Deploy with Tags | Pact Docs [online] (2021) available: https://docs.pact.io/pact\_broker/client\_cli/can\_i\_deploy\_usage\_with\_tags [accessed 15 Aug 2023].

Victor, A. (2023) All You Need To Know About The DevOps CI/CD Pipeline [online], available: https://insights.daffodilsw.com/blog/devops-ci-cd-pipeline [accessed 5 Jul 2023].

Vocke, H. (2023) The Practical Test Pyramid [online], *martinfowler.com*, available: https://martinfowler.com/articles/practical-test-pyramid.html [accessed 4 Jul 2023].

Waseem, M., Liang, P., Márquez, G., and Di Salle, A. (2020) ‘Testing Microservices Architecture-Based Applications: A Systematic Mapping Study’, available: https://doi.org/10.1109/APSEC51365.2020.00020.

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

What Is a Container Deployment? | VMware Glossary [online] (2023) *VMware*, available: https://www.vmware.com/topics/glossary/content/container-deployment.html [accessed 30 Jun 2023].

What Is Docker? | AWS [online] (2023) *Amazon Web Services, Inc.*, available: https://aws.amazon.com/docker/ [accessed 21 Jul 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].

Wu, C.-F., Ma, S.-P., Shau, A.-C., and Yeh, H.-W. (2022) ‘Testing for Event-Driven Microservices Based on Consumer-Driven Contracts and State Models’, in *2022 29th Asia-Pacific Software Engineering Conference (APSEC)*, Presented at the 2022 29th Asia-Pacific Software Engineering Conference (APSEC), Japan: IEEE, 467–471, available: https://doi.org/10.1109/APSEC57359.2022.00064.

Yousif, M. (2016) ‘Microservices’, *IEEE Cloud Computing*, 3(5), 4–5, available: https://doi.org/10.1109/MCC.2016.101.

Zimmermann, O. (2016) ‘Microservices tenets: Agile approach to service development and deployment’, *Computer Science - Research and Development*, 32, available: https://doi.org/10.1007/s00450-016-0337-0.

# 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>

<https://github.com/L00171183/Contract-Testing.git>