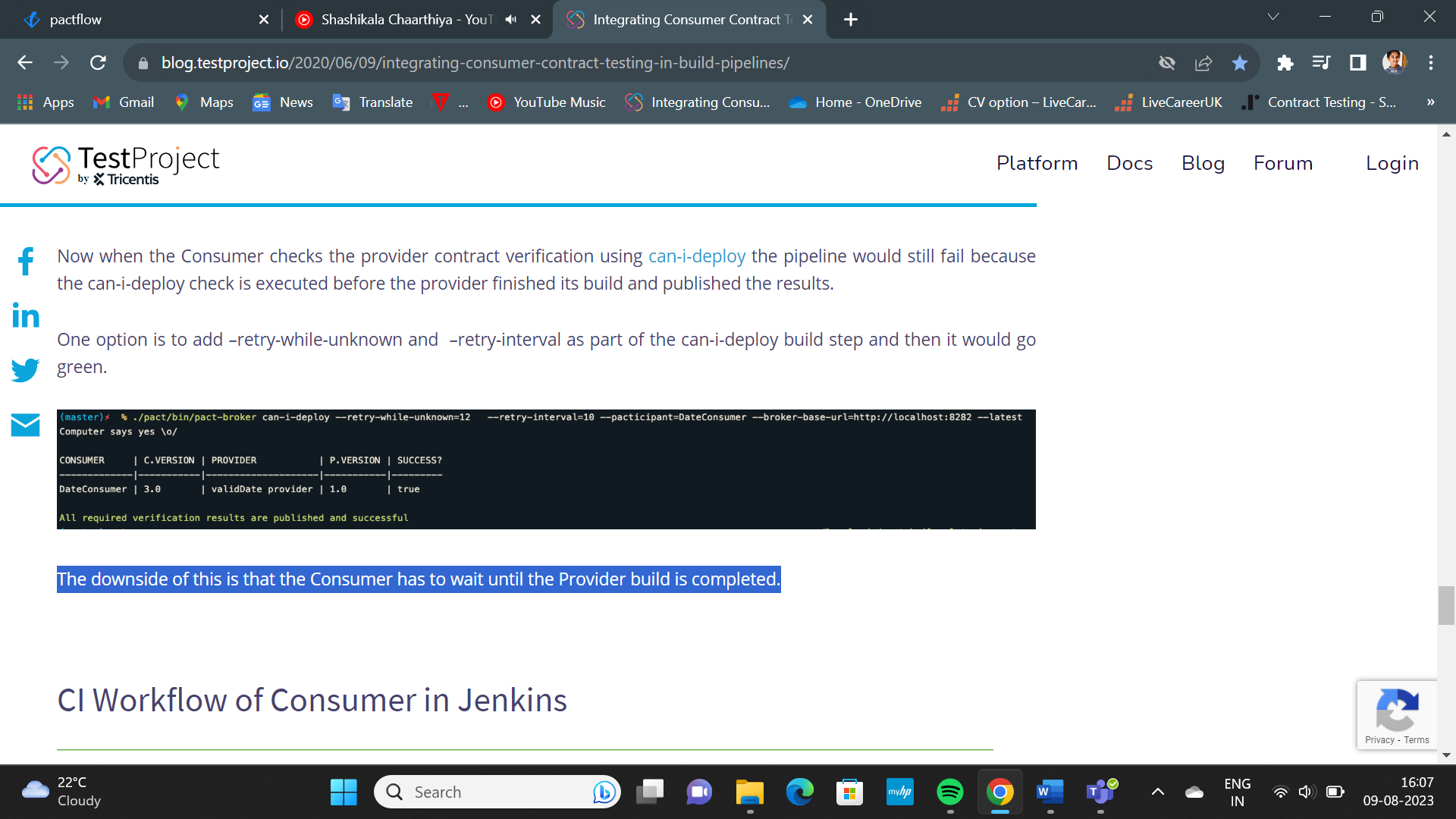
CI, CD, CT, CDC, MSA, MA

Conclusion



<https://www.youtube.com/watch?v=yaAe0aR1G-U&t=1116s> real inttest scenario v/s contracttest. 5 refer throughout the video for advntags. Towards end: disadvntages/limitation

#can be added to implementation intro

Creating a robust and reliable online shopping system demands rigorous testing, particularly when multiple microservices interact. In this context, we are employing the Pact Framework to establish contract tests that ensure seamless communication between our provider and consumer services.

The system comprises three pivotal microservices. The retailer service, functioning as the provider, takes on the responsibility of supplying order data. On the other end, the customer service and product service act as consumers, fetching crucial order details. Our primary goal with contract testing is to guarantee that these microservices interact harmoniously, maintaining data integrity and preventing any inadvertent regressions.

To accomplish this, we are harnessing the power of Pact JVM within the Spring Boot ecosystem. With Pact, we can create a set of well-defined contract tests that outline the expected interactions between the provider and consumer services. These tests validate that data exchanged adheres to predefined expectations, promoting reliability and stability across the entire system.

Our contract test results will be consolidated within a Pact Broker, a centralized platform that provides visibility into the compliance of our microservices with the defined contracts. This visibility is integral for assessing the health of the system and catching any potential discrepancies early on.

Furthermore, our integration with Jenkins, a widely-used automation server, will streamline the entire testing process. By integrating the Pact Broker with our Jenkins pipeline workflow, we establish a seamless and automated mechanism for evaluating contract test outcomes. This integration bolsters our Continuous Integration (CI) and Continuous Deployment (CD) practices, enhancing the system's overall quality and reliability.

With our chosen technology stack encompassing Java, JUnit, Gradle, and Spring Boot, we are well-equipped to implement and execute these contract tests efficiently. By proactively identifying and rectifying any communication mismatches between microservices, we fortify our online shopping system's foundation, ensuring a smooth and dependable experience for both users and stakeholders.

Structure

# Introduction

* Include diagram representation (SDLC to DevOps, Monolith to Microservices, pipeline, integration of CDC framework… high-level).
* Scope and limitation –towards end
* Whole edit throughout and towards end

# Design

Refer for design throughout this doc

Design Introduction: Jenkins Integration with Pact Contract Test Framework, Pact Broker, and Docker Compose

Introduction:

In the realm of software development, ensuring the reliability and compatibility of services and applications is of paramount importance. Contract testing, a technique that allows service providers and consumers to validate their interactions based on predefined contracts, is an effective way to achieve this. To streamline the contract testing process and promote efficient integration within the continuous integration and delivery (CI/CD) workflow, this dissertation aims to design a pipeline using Jenkins. The pipeline will incorporate the Pact contract test framework, Pact Broker for centralized contract management, and Docker Compose for containerized test environments.

Pipeline Overview:

The primary objective of this pipeline design is to establish a seamless integration of the Pact contract test framework, Pact Broker, and Docker Compose within the Jenkins CI/CD workflow. The pipeline will automate the execution of contract tests, provide comprehensive test reports, and leverage Docker Compose for spinning up isolated and reproducible test environments.

Components of the Design:

Jenkins Server: The foundation of the pipeline will be a Jenkins server, which will orchestrate the entire CI/CD process. It will handle source code management, build compilation, test execution, and deployment tasks.

Pact Contract Test Framework: The Pact framework facilitates contract testing by allowing service providers to define contracts and service consumers to validate their interactions against those contracts. The pipeline will incorporate the Pact framework to execute contract tests during the CI/CD process.

Pact Broker: The Pact Broker serves as a centralized repository for storing and managing contract files. It provides a platform for publishing, sharing, and versioning contracts between service providers and consumers. The pipeline will utilize the Pact Broker to store and retrieve contract files, as well as publish test results.

Docker Compose: Docker Compose is a tool for defining and running multi-container Docker applications. It allows the creation of isolated and reproducible test environments, which closely resemble the production environment. The pipeline will leverage Docker Compose to spin up the necessary test environment, including service provider and consumer instances, for executing contract tests.

Pipeline Workflow:

The workflow of the pipeline will follow these essential steps:

Build Compilation: Jenkins will retrieve the source code from the version control system and compile it to generate the build artifacts.

Test Environment Setup using Docker Compose: The pipeline will use Docker Compose to define and deploy the required test environment. This environment will consist of the necessary containers representing the service providers and consumers involved in the contract testing.

Contract Test Execution: Once the test environment is up and running, the pipeline will trigger the Pact contract tests using the Pact framework. The tests will be executed against the deployed service provider and consumer instances within the Docker Compose environment.

Test Result Reporting: After the contract tests are completed, the pipeline will collect the test results and generate comprehensive reports. These reports will provide detailed insights into contract compliance, failures, and potential compatibility issues.

Publishing Results to Pact Broker: The pipeline will publish the test results, including the contract files, to the Pact Broker. This step ensures centralized contract management, enabling service providers and consumers to access and review the test outcomes.

Notification and Alerts: The pipeline can be configured to send notifications or alerts to relevant stakeholders, such as service owners, in case of test failures or compatibility issues. This facilitates prompt identification and resolution of any contract-related problems.

Conclusion:

By integrating the Pact contract test framework, Pact Broker, and Docker Compose within the Jenkins CI/CD workflow, this pipeline design aims to streamline the contract testing process and enhance the reliability and compatibility of services and applications. The automation of contract tests, centralized contract management through the Pact Broker, and the utilization of Docker Compose for isolated test environments contribute to an efficient and quality-driven software development process. This pipeline design promotes seamless collaboration and enables robust contract testing practices throughout the CI/CD pipeline.

# Literature Review

Add Deployment –---- section to DevOps Pipeline. refer Zotero notes

In CDC, add the below scenario:

* + *Smith stated that X, whilst Bloggs stated Y which supports this research question... refer to every Zotero references, take each one – state what is told –* ***CONCLUSIONS/RESULTS****, what is experimented there, what is missing or what we are going to do which is different from theirs’*
  + *Critique*
  + *Map to RQ and your dissertation*

Diagram representation wherever possible

***Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture***

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

First, deployment in the development environment is dif­ficult. Although the application code is now in isolated services, developers must also deploy the dependent services to run the isolated services on their machines. (C). This problem occurred after we introduced dynamic service collaboration.

To solve it, we chose Docker Compose and put a sample deployment description ­ le in each service so that the dependent services can be easily deployed from our private Docker registry. Second, service contracts are critical. Changing so many services that expose their contracts only to each other could be error-prone. Even a small change in the contracts can break part or even all of the system. One possible solution is service versioning, but it could make deploying each service even more complex. So, people usually don’t recommend service versioning for microservices. Thus, techniques such as the Tolerant Reader service design pattern11 Operations Quality assurance Development Core team Cross-functional team Cross-functional team Cross-functional team Cross-functional team Cross-functional team Cross-functional team (a) (b) FIGURE 5. DevOps team formation. (a) Traditional horizontal teams. (b) Vertical teams in DevOps. In DevOps, each team is responsible for a service and contains people with different skills, such as development and operations skills. The team members cooperate from the project’s start to create more value for the particular service’s end users. Authorized licensed use limited to: Atlantic Technological University (ATU). Downloaded on June 21,2023 at 11:31:50 UTC from IEEE Xplore. Restrictions apply. MAY/JUNE 2016 | IEEE SOFTWARE 49 are more advisable to avoid service versioning. Consumer-driven contracts could help greatly in this regard because the team responsible for a service can be con­ dent that most of its customers are satis­fied with the service. (CDC)

***A Software Architect’s Perspective, 2015.***

The DevOps goal of minimizing coordination among various teams can be achieved by using a microservice architectural style where the coordination mechanism, the resource management decisions, and the mapping of architectural elements are all specified by the architecture and hence require minimal inter-team coordination. (D&M). A collection of practices for development can be added to the microservice architectural style to achieve dependability and modifiability, such as identifying and isolating areas of likely change. Adopting a microservice architectural style introduces additional challenges in monitoring, debugging, performance management, and testing. (M) (IT) (CDC)

The limited amount of inter-team coordination may cause misunderstandings between the team developing a client and the team developing a service in terms of the semantics of an interface. In particular, unexpected input to a service or unexpected output from a service can happen. There are several options. First, a team should practice defensive programming and not assume that the input or the results of a service invocation are correct. Checking values for reasonableness will help detect errors early. Providing a rich collection of exceptions will enable faster determination of the cause of an error. Second, integration and end-to-end testing with all or most microservices should be done judiciously. It can be expensive to run these tests frequently due to the involvement of a potentially large number of microservices and realistic external resources. A testing practice called Consumer Driven Contract (CDC) can be used to alleviate the problem. That is, the test cases for testing a microservice are decided and even co-owned by all the consumers of that microservice. Any changes to the CDC test cases need to be agreed on by both the consumers and the developers of the microservice. Running the CDC test cases, as a form of integration testing, is less expensive than running end-to-end test cases. If CDC is practiced properly, confidence in the microservice can be high without running many end-to-end test cases. CDC serves as a method of coordination and has implications on how user stories of a microservice should be made up and evolve over time. Consumers and microservice developers collectively make up and own the user stories. CDC definition becomes a function of the allocation of functionality to the microservice, is managed by the service owner as a portion of the coordination that defines the next iteration, and, consequently, does not delay the progress of the current iteration. (CDC)

***Microservices Evolving DevOps Pipeline***

Enterprises are snappily getting an intricate mesh of numerous operations. As companies produce further and further microservices, their deployment surroundings are getting increasingly elaborate. Without proper configurations, a microservices road chart could snappily come unmaintainable. (C) The microservice architectural style creates a wealth of openings for development brigades to evolve their DevOps channels.

Containerization is another element that broadens and supplements microservices-based models. Bundling each help as a container picture further diminishes the intricacy while smoothing out the persistent conveyance pipeline. Administrations can go about as completely autonomous substances with every one of the conditions and necessities packaged inside the container. This makes the administration's framework rationalist and reusable while permitting them to interface with some other framework. (C)

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

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

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

**Refer Methodology for Design & Implementation ; Discuss on the Conclusions**

***Software development using DevOps tools and CD pipelines : a case study***

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

n their article about using Docker containers as the deployment platform, [Garg and Garg, 2019](S5) provision docker containers as a solution to automating the environment installation and removing the worry on configuration differences. (design docker)

***A Systematic Mapping Study on Microservices Architecture in DevOps***

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

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

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

To understand how MSA is employed in DevOps, we conducted an SMS through a collection of primary studies on MSA in DevOps context. The objective of this SMS is to *identify, analyze, and classify the literature on MSA in DevOps with respect to the research themes, problems, solutions, challenges, description methods, patterns, quality attributes (QAs), tools, and application domains*. The objective of this SMS is further decomposed into a number of Research Questions (RQs) that are listed in Table 1.

The key contributions of this SMS are: (1) A classification of the research themes related to MSA in DevOps. (2) A classification of the problems that practitioners may face during the implementation of MSA in DevOps and the solutions adopt to address the problems. (3) A list of identified research challenges in the context of MSA in DevOps. (4) A classification of the tools that support MSA in DevOps. (5) A list of MSA description methods, MSA patterns, QAs, tools, and application domains.

The rest of this paper is organized as follows: Section 2 briefly introduces MSA and DevOps, existing literature reviews, and motivation of this SMS. Section 3 presents the research method used in this study. Section 4 provides the study results. Section 5 discusses the results. Section 6 describes the threats to validity and Section 8 concludes the study.

DevOps is a culture that combines new or improved practices, processes, team structures and responsibilities, and tools to maximize the ability of an organization to deliver applications and services quickly (Mueller, 2018; Sánchez-Gordón and ColomoPalacios, 2018). DevOps acts as a process framework that can be used for developing, deploying, and managing MSA (Larrucea et al., 2018). The coexistence of microservices and DevOps enables reusability, decentralized data governance, automation, and built-in scalability (Balalaie et al., 2016). MSA and DevOps have many common characteristics that make them a perfect fit for each other. For instance, DevOps practices and MSA promote the idea of decomposing large problems into smaller pieces and then address them through small cross-functional teams (Watts, 2020). Containerized microservices can be realized independently because DevOps gives them a favor of continuous integration and deployment. Although it is not compulsory to design software systems based on MSA in DevOps, most of the challenges arisen in DevOps can be resolved by using MSA (Bass et al., 2015).

## DevOps Pipeline

***Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture***

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

***A Software Architect’s Perspective, 2015.***

Our definition of DevOps focuses on the goals, rather than the means. DevOps is a set of practices intended to reduce the time between committing a change to a system and the change being placed into normal production, while ensuring high quality. (D).

DevOps, in many ways, is a response to the problem of slow releases. The longer it takes a release to get to market, the less advantage will accrue from whatever features or quality improvements led to the release. Ideally, we want to release in a continuous manner. This is often termed continuous delivery or continuous deployment. (D).

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

**Build and Integration Testing**

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

**Packaging,**

The goal of building is to create something suitable for deployment. (IT)(C). There are several standard methods of packaging the elements of a system for deployment. The appropriate method of packaging will depend on the production environment. Some packaging options are: Runtime-specific packages, such as Java archives, web application archives, and federal acquisition regulation archives in Java, or .NET assemblies. Operating system packages. If the application is packaged into software packages of the target OS (such as the Debian or Red Hat package system), a variety of well-proven tools can be used for deployment. VM images can be created from a template image, to include the changes from the latest revision. Alternatively, a new build can be distributed to existing VMs. These options are discussed next. At any rate, VM images can be instantiated for the various environments as needed. One downside of their use is that they require a compatible hypervisor: VMware images require a VMware hypervisor; Amazon Web Services can only run Amazon Machine Images; and so forth. This implies that the test environments must use the same cloud service. If not, the deployment needs to be adapted accordingly, which means that the deployment to test environments does not necessarily test the deployment scripts for production. Lightweight containers are a new phenomenon. Like VM images, lightweight containers can contain all libraries and other pieces of software necessary to run the application, while retaining isolation of processes, rights, files, and so forth. (C). In contrast to VM images, lightweight containers do not require a hypervisor on the host machine, nor do they contain the whole operating system, which reduces overhead, load, and size. Lightweight containers can run on local developer machines, on test servers owned by the organization, and on public cloud resources—but they require a compatible operating system. Ideally the same version of the same operating system should be used, because otherwise, as before, the test environments do not fully reflect the production environment.

The emergence of lightweight containers often assumes one service per container, but with the possibility to have multiple containers per VM

**Integration Testing** – Refer chapter

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

(C) |->

The emergence of lightweight containers and image management tools is helping developers to deploy into small-scale production-like environments more easily for testing.

**Deployment -** Refer chapter

Consistent container for running microservices. Microservice instances are run on individual AWS EC2 instances with a baked AMI that is controlled by the PaaS team, not the microservice developers. This AMI contains necessary runtimes and PaaS infrastructure. Instance size can be controlled by microservice developers (e.g., compute-optimized instances can be specified for CPU-intensive microservices). To deploy a microservice onto the PaaS, developers only need to provide a service descriptor that includes service configuration and metadata (e.g., required resources, environment variables), and an artifact to be run (e.g., a binary JAR file for JVM services, or a Docker image).

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

Refer chapter 5, 11, 12, 13

#### Integration Testing

***Devops, A New Approach To Cloud Development & Testing***

Continuous Testing (CT) considers one of the important aspects of development that ensures the product quality deploy to the end-user. DevOps process focuses on automating all the types of testing and build the appropriate testing environment for the development scenarios. Continuous Testing replaced this traditional testing mentioned in Figure 2. In the continuous process, the occurrence of changes in software from development to testing to deployment is very frequent. So, development process is not the same as was in past where software handoff from one team to another and focus was on increasing revenues for the company. Continuous testing is a process of testing early, testing often, testing everywhere, and automate. One of the challenges in CT is the environment of heterogeneity, which will never reflect actual production environment and application architecture. (IT)

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

***Continuous Testing for DevOps: Evolving Beyond Automation***

Since testing is often one of the greatest constraints in the SDLC, optimizing quality processes to allow testing to begin earlier, as well as shrink the amount of testing required, can have a marked impact on acceleration. Moreover, adopting a bona fide continuous testing process (more than just automated tests running regularly) helps promote all of the core pillars of DevOps.

***DevOps with Continuous Testing Architecture and Its Metrics Model***

The advent of DevOps is to take full advantage of iterative model of development, bring agility in software development life cycle and achieve time to market goal. However, testing becomes roadblock and reduces the rate of speed. Hence, there is a critical need to strategize testing process and align it to continuous planning, continuous integration, continuous deployment and continuous monitoring, and feedback goals of DevOps practice. There is a vital difference between test automation and continuous testing. The former is subset of latter. Continuous testing identifies integration issues much earlier in the life cycle; makes defect resolution cheaper, faster; and frees tester’s precious time for exploratory testing and value-added test activities. (IT). This paper conducts literature survey on various strategies applied for continuous testing and proposes a continuous testing architecture for better implementation. It also presents the conceptual design of few important testing metrics for successful implementation of continuous testing function in the context of DevOps.

#### Continuous Deployment

## Microservices and Containers

### Monolithic to microservices

* refer Zotero, pdf, ( of CDC and containerization too)

***Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture***

A microservices architecture is a cloud-native architecture that aims to realize software systems as a package of small services. Each service is independently deployable on a potentially different platform and technological stack. It can run in its own process while communicating through lightweight mechanisms such as RESTful or RPC-based APIs—for example, Finagle. (REST stands for Representational State Transfer.) (M). In this setting, each service is a business capability that can utilize various programming languages and data stores and is developed by a small team.2 Migrating monolithic architectures to microservices brings in many benefits. In particular, it provides adaptability to technological changes to avoid technology lock-in and, more important, reduced time to-market and better development team structuring around services.3 (M)

***Microservices***

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

Microservice software breaks systems and applications down to a more granular, modular level. microservices are small applications with a single responsibility that can be deployed, scaled, and tested independently.1 This decomposition of the monolith (whose modules can’t be executed independently) into a granular system interacting via messages (through RPC-based APIs or RESTful web services, for instance) enables organizations to achieve better time to market by means of swifter, more continuous deliveries. (REST stands for Representational State Transfer.) It also enables agile teams to structure their work around these services,2 given that microservices are, by definition, autonomously developed.3 Connecting microservices with DevOps will increase software engineering’s impact and benefits.4 (D&M). However, microservices also pose challenges and have disadvantages. Challenges include decomposing the monolith into microservices; continuous architecture monitoring and deployment; more complex testing, (C),(CDC) versioning, and deprecating; and state management. A particular disadvantage is that microservices can involve soft factors such as the need for experienced staff and the difficulty of learning the technology. Companies such as Amazon, Deutsche Telekom, LinkedIn, Netflix, SoundCloud, The Guardian, Uber, and Verizon are quickly adopting microservice-based approaches. Often, microservices are used to modernize legacy applications. The goal is to split such monolithic systems into microservices through refactoring. This supports the incremental modernisation of legacy software and is perhaps less risky than completely redeveloping the whole system into microservices.

Because microservices need DevOps, we recommend starting with a tailored DevOps strategy. It will have immediate value owing to better integration across the lifecycle and can gradually evolve to a microservice delivery model—if appropriate.

Refer to references

***From Monolithic Architecture to Microservices Architecture***

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

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

Refer to references

### Containerization

***Docker Book – Introduction. ( why docker in design)***

***Refer IBM learning paths for microservices, containers, docker***

***Containers & Docker: Emerging roles & future of Cloud technology***

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

A microservices architecture approach makes web-based development agile in the nature and easier to maintain the code base. Docker is an enabler for a Microservices architecture and container-based application deployment.

(refer for design docker justification)

***Adaptive Containerization for Microservices in Distributed Cloud Systems***

In recent years there has been a paradigm shift in software architecture design from an on-premises model to a cloud native approach [1]. (M) This change has been largely driven by the demand for greater system reliability, scalability, and flexibility.

On the other hand, containerization [5] is a form of virtualization that attempts to achieve resource isolation with minimal overhead by sharing the kernel with the host OS. The use of containerization allows us to implement many of the desirable features associated with cloud platforms– elasticity, reliability, and ease of management. (C) Therefore, we focus on container-based virtualization in this paper. Microservices can be conveniently packaged into containers that are then deployed onto physical hardware, thus ensuring a consistent software execution environment from the developer to the consumer. The use of containerization also allows for inherent scalability and creates a redundancy mechanism for machine failure as container instances can be added and removed on demand. (C) While the decomposition of a monolithic application into a set of microservices is the prerogative of the application developer, the deployment of microservices onto physical hardware is a run-time exercise

***Containerized Microservices Orchestration and Provisioning in Cloud Computing: A Conceptual Framework and Future Perspectives***

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

Refer for LR structure

### Testing of Microservices

## Consumer-Driven Contract Testing

* Refer all the **LR** notes
* Frameworks – Spring Cloud and Pact

***Lost all previously stored data………***

***16-06-2023***

***Testing Integrations with Consumer-Driven Contract Tests***

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

The testing method is said to be useful when testing integration-heavy systems such as systems based on microservice architecture. (CDC). Therefore the research question of the thesis is: "with a focus on integrations, is Consumer-Driven Contract testing a viable addition to a testing strategy used to test a system based on microservice architecture, and if so, why?" **RQ** (CDC)

The research question is first approached by taking a look at the most recent literature. The goal is to learn about different testing methods and create a basic understanding of a general testing strategy for microservices. The next step is to figure out how the Consumer-Driven Contract testing fits that picture. The Consumer-Driven Contract testing is introduced thoroughly to gain a good understanding of its core concepts, advantages, disadvantages, and tooling.

After the literature check, the research question is approached by introducing a case study based on a microservice architecture. Its testing strategy is described in detail, and Consumer-Driven Contract tests are implemented for it. The testing methods are compared by systematically implementing defects to the integrations and seeing how the testing methods catch them. Finally, the results and experiences are shared and analysed, and the research question gets answered.

The results based on literature and experiences from the case study proved that the Consumer-Driven Contract testing is a viable way to test integrations. The tests implemented in the case study caught every defect from the integrations, and the case study was able to verify the advantages mentioned in the literature. It was shown that the Consumer-Driven Contract tests could replace the more traditional integration tests completely. That results to more deterministic testing strategy as the integrations are tested in isolation.

It should be emphasized that the teams have to be able to communicate with each other to implement and achieve the benefits of Consumer-Driven Contract testing. The level of communication between the teams has to be mature enough to share the contracts and to coordinate the implementation. Communication is the foundation that enables or disables the testing method. Because of that improving the ways of communication should be a major focus for the teams who want to implement Consumer-Driven Contract tests.

***(26) Contract testing - part one - docker containers | LinkedIn – (For design and implementation)***

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

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

Why it "make sense" to start building the services in this way? The quick answer is: because it defines a process that has a series of benefits hence becoming one of the "best practices" a development team needs to follow.

In software development, it is better to include the testing from the beginning phase of the development process:

* it is better to involve the QA early into the process to get familiarized with the requirements and create the test cases,
* it is better to define a process that can benefit from running test cases that are created based on architecture specifications - contracts - and verifies them constantly during the local development or during the execution of the promotion pipeline.

At the same time, the test cases can be done in isolation, without having to spawn the whole framework: one service can test its dependency on the immediate service and only on this one. Another advantage is the fact that an end-to-end integration test case it is most likely easier to get executed and passed when all the individual integration test cases have been performed successfully for a while. (CDC) One last advantage I want to mention here is: its simplicity. If we can setup an environment where we don't have to write any line of code that will give us the ability to create these test cases, wouldn't be this a great benefit?

(refer practical + other parts II,III… )

***Containers vs Microservices: What’s The Difference***?

A post from Ev Kontsevoy summarized the comparison of these two terminologies in an interesting way:

“A container is a useful resource allocation and sharing technology. It’s something DevOps people get excited about. A microservice is a software design pattern. It’s something developers get excited about.”

In other words, we can sum this up as:

* Microservices are about the design of software.
* Containers are about packaging software for deployment.

So, we can choose whether to use a container for hosting a microservice. But to get full value from both, it is significantly better to run microservices within containers.

Deploying an entire application to a single VM introduces a single point of failure risk, whether or not a microservice architecture has been used. But spreading the application through microservices across multiple containers results in fully exploiting the value of both by providing resilience as well as agility through scaling and improvements targeting specific services without negatively impacting the entire application.

Flexibility is also introduced in that developers can write applications in the language of their choice since the container will allow them to deploy across whatever environment is provided. Efficiency comes from containers using less resources compared to VMs.

An added benefit comes in the form of security through isolation and a broader attack surface that limits the impact should a single microservice or container be subject to a security breach such as a hacking attack.

***DevOps for Microservices - Creating Change Together***

***Previous***

***Next***

***DevOps and Microservices – Creating change together***

The cloud-native software architectures are not complete without microservices. Enterprises use modern practices like DevOps and microservices architectures to design software applications. This enables organizations to deploy these applications as independent services. And since DevOps and microservices are connected, they offer greater agility and operational efficiency.

DevOps helps developers and IT operations to work in close coordination with each other to deliver superior quality software quickly. It also allows developers to gain insights into production settings to improve the quality of their software.

Organizations follow DevOps practices to the cloud for replicating the production environment with better accuracy. DevOps facilitates easy collaboration amid development and IT ops teams and minimizes the resistance between them.

Microservice architecture on the other hand is a type of Service-Oriented Architecture (SOA), a software development technique. Microservices are tiny pieces of functionalities that are used as a service. These are independent building blocks, with the help of which it becomes easier to create, test and understand an application.

Since they offer greater modularity, several DevOps teams can construct and set up microservices side by side. Furthermore, it is easy to deploy microservices in any cloud. Hence, developers do not have to write specific codes for Microsoft Azure and AWS platforms separately.

DevOps practices have a simple principle. It focuses on breaking down huge problems into smaller ones and handling them in order of preference as a team. Likewise, Microservices also operate through small teams and make one functional change at a time. Besides, microservices can easily adjust the scaling without impacting the resource allocations for the remaining system. This simply indicates that the blend of DevOps and microservices in the development process is the fastest way to enhance the output of your teams.

The blend of DevOps and microservices enable organizations to develop faster at a lesser cost. This makes them more agile and productive. Microservices bring added productivity to DevOps as they can work on the same toolset for development and operations. This marks similar terminology and processes for requirements, reliance as well as obstacles. This way Devs and Ops can easily work together. When organizations don’t supplement DevOps with microservices, it can lead to competitive disadvantage and impact them negatively. To know more about how DevOps and microservices can improve your business processes and help in your business growth, talk to our experts

***Effective Test Automation Approaches for Modern CI/CD Pipelines – (Read the whole article – restructure your Dissertation)***

The rise of CI/CD has had a massive impact on the software testing world. With developersrequiring pipelines to provide quick feedback on whether their software update has been successful or not, it has forced many testing teams to revisit their existing test automation approaches and find ways of being able to speed up their delivery without compromising on quality. These two factors often contradict each other in the testing world as time is often the biggest enemy in a tester’s quest to be as thorough as possible in achieving their desired testing coverage.

So, how do teams deal with this significant change to ensure they are able to deliver high-quality automated tests while delivering on the expectation that the CI pipeline returns feedback quickly? Well, there are many different ways of looking at this, but what is important to understand is that the solutions are less technical and more cultural ones - with the approach to testing needing to shift rather than big technical enhancements to the testing frameworks.

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

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

***Microservices and Containers 101 - Learn all About Microservices***

Microservices is an architectural design for building a distributed application. Microservices break an application into independent, loosely-coupled, individually deployable services.

Containers are a lightweight, efficient and standard way for applications to move between environments and run independently. Everything needed (except for the shared operating system on the server) to run the application is packaged inside the container object: code, run time, system tools, libraries and dependencies.

Testing microservices needs a strategy that takes both service dependencies and the isolated nature of microservices into account. In order for the process of microservices testing to work, each microservice that’s in isolation needs to be verified it’s working properly and then proceed with testing the microservices together. There are a variety of tests that can be conducted in order to validate your application: unit testing, contract testing, integration testing, and end-to-end testing

***Testing Microservices - Contract Tests (read full article)***

Testing monolithic applications is relatively easy. We implement unit, integration, and probably end-to-end tests, and try to keep the testing pyramid as perfectly shaped as possible. But when it comes to testing microservices, things complicate a little. The main difference stands for communication which is essential in a microservice architecture. Luckily, for every challenge, we have a pattern and, in this case, it’s contract testing. (CDC)

This article started by pointing out the challenges of testing communication in a microservice architecture. It also reviewed two common approaches to testing it - integration and e2e testing. Then, contract testing was introduced as a solution for the aforementioned problems.

Contract testing is a concept that allows testing communication (both synchronous and asynchronous) between the services in isolation. The main idea behind it is a contract. This is a place where communication rules are agreed upon and written down by the sides. The contract is a single point where the contract is defined. The parties don’t have to synchronize when the contract changes. Both producer’s and consumer’s tests are based on this contract. It ensures that if the tests pass, the services get along in production. (CDC). Next, I explained the difference between Consumer Driven and Producer Driven approaches and when to use them. Finally, I gave you a tip on how not to fall into the trap of overusing contract testing.

***The Role of Containers in Your Microservice Architecture (full article)***

Technology is evolving faster than ever. People depend heavily on the internet for all kinds of regular tasks, from shopping to banking and healthcare. That’s made it critical for service providers to fulfill this ever-increasing consumer demand. This forces service providers to abandon monolith software development methods and adopt Agile and DevOps approaches that help them quickly adapt to changing requirements. Another trend is microservices-based architectures, where applications are built as multiple loosely coupled services. In this article, we will discuss the role of microservices in the DevOps process.

DevOps is a paradigm shift in the way organizations approach software development, deployment, and maintenance. DevOps shifts the whole software development lifecycle (SDLC) to a more collaborative process.

DevOps offers a multitude of benefits for an organization. However, to gain those benefits you must properly implement DevOps with standardized concepts and proper tools such as:

* Continuous integration
* Continuous delivery platforms
* Automated testing

The next consideration is the deployment strategy. Most DevOps deployment stages are targeted at highly available and scalable cloud infrastructure with the popularity of cloud-based applications and all the advantages of cloud-based deployments.

Containerized applications are one of the factors that power this shift to cloud-based deployments. Containers allow users to create isolated and portable application environments that can be deployed anywhere with all the required dependencies. Platforms like Kubernetes and Rancher provide robust orchestration capabilities for containerized deployments. Here, automation also plays a major role by automating software packaging and deployments.

A version-controlled and automated DevOps pipeline with a proper deployment strategy allows organizations to create a pipeline that encompasses all the stages of SDLC. Now that we understand the primary considerations of a DevOps pipeline, let’s see how microservices affect all these factors.

Microservices is an architectural approach to development that contrast with traditional, monolithic applications (where the entire application is considered and developed as a single entity). The microservice architecture breaks the application into different loosely coupled services.

Containerization is another factor that extends and complements microservices-based architectures. Packaging each service as a container image further reduces the complexity while streamlining the continuous delivery pipeline. Services can act as fully independent entities with all the dependencies and requirements bundled within the container. This makes the services system-agnostic and reusable while allowing them to interact with any other system. (C)

In a microservices architecture, most testing, packaging, and deployment tasks can be automated for each service. As each service resides in an independent DevOps pipeline, any issues in a single automated task do not affect the other services. (C),(M), (D&M)

Microservices architecture is tailor-made for DevOps with its services-based approach that allows organizations to break down the application into smaller services. This enables delivery teams to tackle individual services as separate entities—ultimately simplifying the development, testing, and deployment. (This doesn’t mean microservices should be used for every application, however. They do come with certain challenges.) The role microservices plays in DevOps includes streamlining the DevOps process and increasing productivity and quality of the application while moving developments to a flexible architecture. This leads to the development of cloud-native applications that are capable of fulfilling any user demand. (D&M)

***What Are Containerized Microservices? – DreamFactory Software- Blog***

DevOps is the merger of traditional development and IT operations teams. DevOps practices place the focus on collaboration across the entire IT organization. Key components of a DevOps organization are automated and standardized deployments, freeing staff from repetitive tasks. The speed of standard deployments that containers offer makes DevOps and containerized microservices a perfect match.

***For LR – Literature Review…*** ***(refer whole papers’ sections)***

***Testing in microservice systems A repository mining study on open-source systems using contract testing***

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

A third approach is consumer-driven contract (CDC) testing. CDC testing is described as a potential solution [10] or addition [11], [12] to handle the challenges of integration testing. In CDC tests, the consumer of a downstream microservice states which responses it expects for certain requests from the provider (i.e., its API). The providing service verifies that it can fulfill these expectations and both parties enter into a contract. Provider and consumer use this contract to test their interfaces independently. (CDC) Waseem et al. [6] state that current MSA testing literature that discusses new or MSA-adapted testing approaches, like CDC testing, consists mostly of experiments and (single-case) case studies and lack an evaluation in a real-life context. Thus, for this research, CDC testing was used as a starting point and inclusion criteria to study microservice testing.

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

Refer to references

***Analysis of Consumer-driven contract tests with asynchronous communication between microservices (refer full structure)***

In the context of microservice architecture the speed of the delivery of microservices is one of the most important parts in the release cycle. To be able to deliver microservices fast and reliably the continuous integration (CI) and continuous deployment (CD) pipeline have to be efficient, of good quality and optimized. This way we can ensure the maximum speed of reliable updates. To get the maximum out of the most beneficial aspect of microservices, the fast release cycle, we need to have fast testing. With the introduction of microservices in an architecture the amount of interfaces that can be reached is significantly higher. Most, if not all, of the communication between microservices rely on these interfaces. This proves to be a new problem in testing. These interfaces have to be tested reliably and quick. Consumer-driven contract (CDC) tests apply exactly at these interfaces between microservices without needing as much time as end-to-end tests. One of the reasons consumer-driven contract testing is deployed in testing pipelines in the context of microservice structure is their fast execution with reliable enough results to catch many errors. In the best case, before the end-to-end tests are even run. This allows the testing department to run less end-to-end tests and reduce the intensity and necessity of end-to-end tests. (CDC), (D&M). CDC testing is well worked out in synchronous messaging architectures even if it still lacks popularity. It is however not as seemingly integrated in asynchronous communication as it is in its synchronous counterpart.

**(Introduction - structure, RQs)\*\*\*, 2.3, 2.4, 3\*\*\*,7**

***An Empirical Analysis of Microservices Systems Using Consumer-Driven Contract Testing (refer full study)***

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

***Consumer-Driven Contract Tests for Microservices: A Case Study***

Design by contract is a paradigm that aims at capturing the interactions of different software components, and formalizing them so that they can be relied upon in other phases of the design. Such a characteristic is especially helpful in the context of microservice architecture, where each service is an independent entity that can be individually (re)deployed. With contracts, testing of microservice based systems can be improved so that also the integration of different microservices can be tested in isolation by the developers working on the system. In this paper, we study how systems based on microservice architecture and their integrations can be tested more effectively by extending the testing approach with consumer-driven contract tests. Furthermore, we study how the responsibilities and purposes of each testing method are affected when introducing the consumer-driven contract tests to the system.

Consumer-Driven Contract testing [9] is a way to test integrations between services and ensure that all the integrations are still working after new changes have been introduced to the system. The main idea is that when an application or a service (consumer) consumes an API provided by another service (provider), a contract is formed between them. The contract contains information about how the consumer calls the provider and what is being used from the responses. As long as both of the parties obey the contract, they can both use it as a basis to verify their sides of the integration. The consumer can use it to mock the provider in its tests. The provider, on the other hand, can use it to replay the consumer requests against its API. This way the provider can verify that the generated responses match the expectations set by the consumer. With consumer-driven contracts, the provider is always aware of all of its consumers. This comes as a side product when all the consumers deliver their contracts to the provider instead of consumers accepting the contracts offered by the provider. (CDC). In this paper our objective is to study how systems based on the microservice architecture [1,12] and their integrations [8] can be tested more effectively by extending the testing approach with consumer-driven contract tests. In particular, we are interested in how the responsibilities and purposes of each testing method are affected when introducing the consumer-driven contract tests to the system. The rest of this paper is structured as follows. Section 2 provides the background for the paper, and Section 3 introduces the case study. Section 4 presents the results of the case study. Section 5 provides an extended discussion regarding our observations. Finally, Section 6 draws some final conclusions.

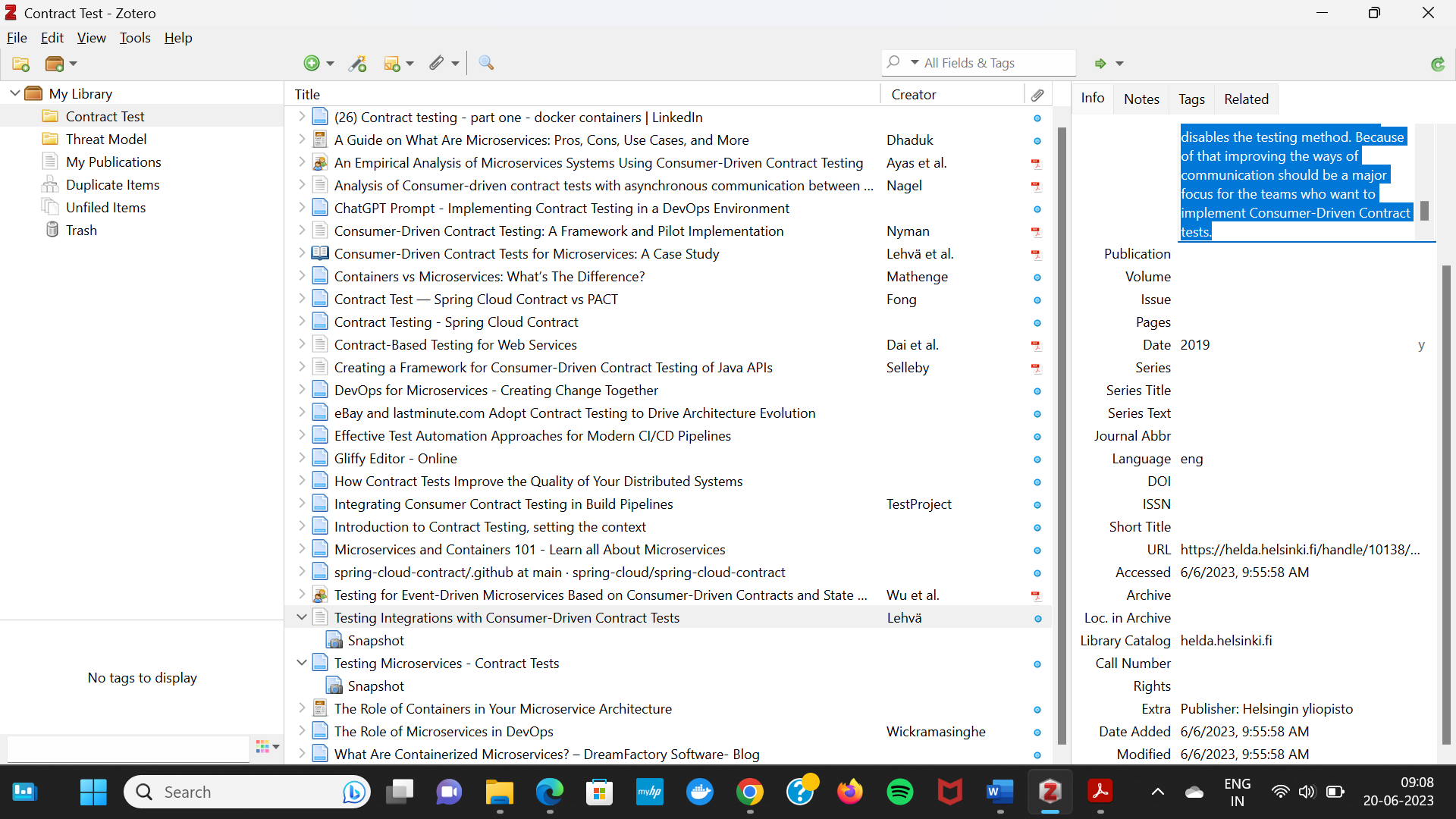
In this paper, we have studied consumer-driven contract testing in the light of a case study based on an industrial system. Our experiences gained from the case study confirmed the benefits commonly associated with such tests: (i) integrations are tested in isolation by decoupling the consumer and the provider using a contract, contributing to fast and stable tests; (ii) the provider knows who are consuming its API and how; (iii) the provider can evolve based on real business needs from its consumers; (iv) the consumer can feel safe as the provider tests always catch breaking changes to the API; and (v) contracts can work as a tool to improve communication between different development teams. Furthermore, our experiences suggest that the consumer-driven contract tests can replace integration tests as they caught all the defects from the integrations that were implemented in the case study. In that light, it can be safely said that consumer-driven contract testing is a viable addition to testing strategies used to test integration-heavy systems, especially those based on microservices. (CDC)

Creating a Framework for Consumer-Driven Contract Testing of Java APIs ***- (refer whole papers’ sections)***

***Creating a Framework for Consumer-Driven Contract Testing of Java APIs - (refer whole papers’ sections)***

***Testing for Event-Driven Microservices Based on Consumer-Driven Contracts and State Models – whole research important***

Microservice architecture has become increasingly popular due to its good maintainability, scalability, fault tolerance, and extensibility. In addition to the REST style of microservices that has been widely used, the event-driven style of microservices is also gaining more and more attention. However, the current software testing methods have little support for event-driven architecture, and the technical complexity of event-driven microservices has further increased the difficulty of testing. In this regard, we propose a software testing tool for event-driven microservice systems called CCTS (Composite Contract Testing Service). By combining consumer-driven contract testing and the event-driven state model, CCTS records the state transitions of event exchange between services, and automatically retrieves the possible transition paths among services. Simultaneously, CCTS analyzes the event logs from the target system to determine whether the event logs conformed with the specified transitions of states and retrieved paths. Besides, CCTS checks the validity of contract testing to ensure that the communication through services. To evaluate CCTS, we conducted functional testing for CCTS using a real-world microservice system. The results show that CCTS can effectively detect potential defects in the event-driven microservice system, such as isolated states, cyclic states, incomplete contract tests, and unqualified event sequences.



## Conclusion

Use (‘Introduction to Consumer-Driven Contract Testing’ 2023) . highlighting the benefits.

**TL;DR:** Contract testing will keep the APIs you produce and consume from breaking unexpectedly, without any manual intervention needed. This makes integration testing and developing new features (including breaking features!) much easier.

#### **1.1 - What is contract testing?**

On a broad level, contract testing can be thought of as testing (and therefore guaranteeing) the communication layer between services. Contract testing tests that any pair of dependent services can properly send and decode messages between each other, but doesn’t test the services’ internal logic. As such, contract testing exists somewhere on the boundary between integration testing and end-to-end testing.

Let’s get some definitions out of the way. There are two main types of actors in contract testing, Providers and Consumers. The Provider is an application responsible for publishing an API; a Consumer of the Provider is another application using (consuming) said API. The Consumers of a Provider will always have basic expectations of the API, such as:

* What endpoints can I use?
* What input do the endpoints take?
* What does the output look like?

Or, more simply,

* How does the API work?

Consumer-driven contract testing is a way to formalize these expectations into a contract between each Consumer-Provider pair. Once this contract is established, contract testing ensures, automatically, that the contract doesn’t break unexpectedly.

#### **1.2 - Why contract testing**

So what we have so far is that contract testing allows you to define contracts and test them. Brilliant. Before you close this tab, let me offer up some motivation.

Contract testing will make your life easier in four main ways. Contract testing:

* Helps Providers make changes without being scared of accidentally breaking their Consumers;
* Lets Consumers know that the APIs they consume won’t suddenly break;
* Allows Consumers to develop against API definitions before the Provider API has actually been developed;
* Makes integrating and testing a service in a microservice landscape easier; and
* Serves as an efficient communication tool between Provider and Consumer teams.

Or again, more simply,

* Contract testing lets everyone relax and be assured that the APIs won’t up and die.

Let’s expound on that a little.

#### **1.3 - Benefits for Providers and Consumers - Relax and be assured that the APIs won’t up and die**

Having APIs not just up and die seems to obviously be A Good Thing™. But there are some less obvious benefits that come from preventing errors from unexpected API changes.

It’s easy for API errors to result in teams blocking each other’s progress. If a Provider team pushes a breaking change, new updates from the Consumer team can’t be deployed until that’s fixed. The contrapositive is also annoying, where a Provider team can’t push their change until they’ve made sure each Consumer team is ready for it. Or even more simply, the Provider is afraid to make a change because they think they impact a Consumer somewhere using it, but they aren’t sure how to make sure. Hopefully they figure out something better than to see if Production goes down or not.

Problems like this do more than just directly slow down development. Having a team’s development be blocked by another team’s breaking changes can result in rising tensions. Rising tensions result in worse communication, and worse communication increases the chance of further miscommunicated breaking changes. One way to get out of a negative feedback loop like this is with good management - or, failing that, by preventing these problems in the first place by using contract testing!

Put simply, contract testing lets Provider teams easily know that changes being made are safe, and lets Consumer teams work without worrying about having the rug pulled out from under them. Moreover, if an API change is breaking, contract testing allows Providers to know which of their Consumers are affected and when it’s safe to push the change.

#### **1.4 - Easier integration testing**

If you’re developing in a microservice landscape, chances are any given one of your services has dependencies on other services. Suppose, for instance, you want to write tests for a Pet Service that speaks to a Cat Service. How do we want to test the parts of our code that depend on responses from the Cat Service?

Well if there’s a Cat Service deployed in a testing environment, you could always write and run integration tests of the Pet Service against this live, running Cat Service. But if you do this, your integration tests are only as stable as your dependencies are. If the testing instance of Cat Service ever acts up, all the impacted Pet Service integration tests will fail, even though there’s nothing wrong with the Pet Service itself. Having false positive test results like this is not only incredibly annoying, but also dangerous, as this train developers to ignore failing tests. If the last hundred times your integration tests failed was because of a flaky Cat Service run by a different team, it’s easy to ignore the one time your test failure actually indicates a problem.

An alternative would be to instead run your integration tests against mocks or stubs of your external dependencies. This way you’ve cut your testing dependency on Cat Service, and any errors you find are actually your fault. But then you face a new problem; how do you know your mock is actually accurately representing your external dependencies? Suppose Pet Service depends on the POST /meow endpoint, but those flaky Cat Service devs change it to POST /purr without letting you know. Now your tests are returning a false negative! Everything looks great, you deploy your application, and everything falls apart.

The reason this falls apart is that our stubs are making assumptions on the behavior of the Cat Service. If this assumption proves to be incorrect, then the stubs are useless. But if we had a way to formalize our assumptions on the Cat Service, and automatically test that those assumptions are still true, then we’d be able to safely run integration tests against stubs. And that’s exactly what contract testing does.

So contract testing lets you do integration testing against mocks while ensuring that your mocks are actually accurate representations of your external dependencies. Neat.

#### **1.5 - Peace in our time**

In addition to all of the above, there are any number of small benefits contract testing brings to the table. Chief among them, in my experience, was that contract testing made inter-team communication much easier and friendlier - easier because it was now clear when the teams had to collaborate on changes, and friendlier because fewer things went wrong!

In practice, contract testing can be consumer-driven or provider-driven. If it’s consumer-driven, then the Consumers define their expectations and the Provider checks that they’re fulfilled; if it’s provider-driven, then the Provider defines the contract, and Consumers check that they’re compatible with it.

Due to a variety of reasons, I would highly recommend the consumer-driven approach over the provider-driven approach. With consumer-driven contracts:

* It’s much easier to properly integrate contract tests into your CI/CD;
* You have more beneficial side-effects, such as each Provider being shown how it is used by its Consumers;
* It’s easier for Consumers to communicate change requests;
* It’s easier for Consumers to develop against requested API changes before they’re actually live;
* There’s better tooling.

The two big frameworks we found for helping out with contract testing are Pact ([www.pact.io](https://docs.pact.io/)) and Spring Cloud Contract (<https://cloud.spring.io/spring-cloud-contract/>). Pact is an open-source language-agnostic consumer-driven language-agnostic framework that comes with some very helpful features, including the Pact Broker for orchestrating contract tests. Spring Cloud Contract, on the other hand, began its life as a stub-runner and was then adapted into a provider-driven framework. We would definitely recommend Pact over Spring Cloud Contract.

Contract Testing as a process can be applied to any kind of API. In this blog series, however, we’ll be focusing on HTTP APIs, and on RESTful APIs in particular.

Contract testing brings a lot of benefits but it’s not trivially easy to do properly. Especially the details of how to set up contract testing properly in your CI can get complicated.

In Pact, the contract itself is called a pact. It is a JSON file that contains interactions. An interaction consists of the request and the expected response. The expected response contains the expected JSON but only the parts that are actually relevant for the consumer.

#### Pact

In Pact, the contract itself is called a pact. It is a JSON file that contains interactions. An interaction consists of the request and the expected response. The expected response contains the expected JSON but only the parts that are actually relevant for the consumer.