Testing in microservice systems

A repository mining study on open-source systems using contract testing

Bachelor of Science Thesis in Software Engineering and Management

HARTMUT FISCHER

Department of Computer Science and Engineering
UNIVERSITY OF GOTHENBURG
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2021

The Author grants to University of Gothenburg and Chalmers University of Technology the non-exclusive right to publish the Work electronically and in a non-commercial purpose make it accessible on the Internet.
The Author warrants that he/she is the author to the Work, and warrants that the Work does not contain text, pictures or other material that violates copyright law.

The Author shall, when transferring the rights of the Work to a third party (for example a publisher or a company), acknowledge the third party about this agreement. If the Author has signed a copyright agreement with a third party regarding the Work, the Author warrants hereby that he/she has obtained any necessary permission from this third party to let University of Gothenburg and Chalmers University of Technology store the Work electronically and make it accessible on the Internet.

Studying the use of testing best practices in MSA-based open-source systems

© HARTMUT FISCHER, August 2021.

Supervisor: HAMDY MICHAEL AYAS
Examiner: Richard Berntsson Svensson

University of Gothenburg
Chalmers University of Technology
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
UNIVERSITY OF GOTHENBURG
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2021

Testing in microservice systems: a repository mining study on open-source systems using contract testing

Hartmut Fischer
Department of Computer Science and Engineering
University of Gothenburg
Gothenburg, Sweden
gusfisha@student.gu.se

***Abstract*— Context: There is a trend in the software industry to migrate systems from a monolithic to a microservice architecture (MSA) due to the gains in agility and scalability. An MSA-based system consists of a suite of small microservices which can be developed, tested, and deployed independently.**

Problem: MSA puts challenges on software testing due to the complexity caused by the integration of many autonomous microservices into one system. New best practices with testing strategies in terms of test types and test proportions (pyramid shape) emerged, but are only studied in isolated cases.

Objective: The aim of this study is to explore microservice testing in a real-life context and to compare it to best practices.

Approach: Repository mining is used to identify open-source MSA-based systems. Consumer-driven contract testing (CDC), a method recommended by MSA practitioners, was selected as inclusion and system identification criteria. Four systems with overall 22 microservices were analyzed.

Results: The recommended five test types were identified, but not in all systems. The bigger a microservice, the better its test strategy aligned to the pyramid shape, but the test proportions also dependent on the microservice's distinct function in the system. Using CDC tests seems to minimize the number of system tests.

***Index Terms*—Microservice architecture, integration test, consumer-driven contract test, test pyramid, repository mining**

I. Introduction

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.

One way is the use of test doubles (i.e., stubs or mocks) during testing to replace the depending services with which the service under test (SUT) is interacting. Interactions with downstream services that the SUT is consuming (i.e., receiving data from) are substituted with the help of test doubles. These tests imply a certain untrustworthiness as they cannot guarantee that the latest state of the interactions and dependent interfaces is reproduced by the test doubles [8].

Another option is to test the interactions and inter-communication in a staging environment with only the required microservices [9], or, the entire system deployed [10]. In a (partly) running system, defects cannot only be caused by the services themselves or their inter-communication but also by failures in the infrastructure (e.g. network latency or outages). This makes the testing process time-intensive, less reliable and it gets more difficult to track down defects to their origin.

A third approach is consumer-driven contract (CDC) testing. CDC testing is described as a potential solution [10] or addi-

tion [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.

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.

The objective of this paper is to explore the test strategies of in-production or production-ready MSA-based applications. The focus lies hereby on systems that are using CDC testing as a test type appropriate for the special requirements of MSA. The research is conducted by repository mining to identify relevant open-source projects. This will help to get a better understanding of how microservices systems are tested and how well they follow best practices.

This paper is organized as follows. Section II provides the background about MSA and testing relevant for this paper, and Section III discusses related work. In Section IV the research methodology including the research questions, and the data collection and data analysis process are described. The results are presented in Section V, and, Section VI provides an analysis of the thesis' findings. Threats to validity and the measures to mitigate them are presented in Section VII. Finally, SectionVIII concludes this thesis and describes directions for future work.

II. Background

A. Microservice Architecture Style

The microservice architecture (MSA) style describes a system that consists of a suite of small, independent, interacting microservices. In contrast, applications with a traditional, monolithic architecture consist of a single, unified unit (see Figure 1).

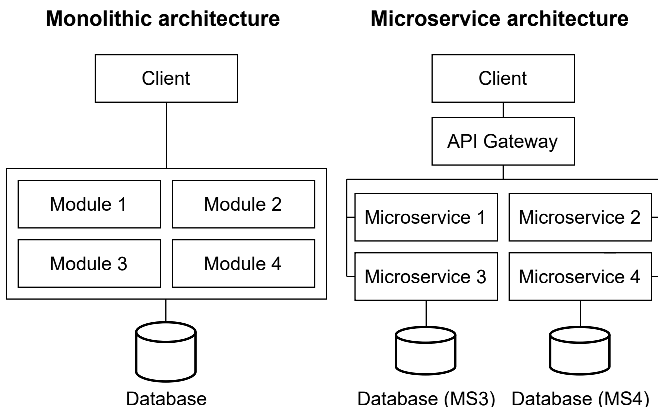

Fig. 1. Exemplary architecture of a monolith and a microservice system (adapted from Savchenko et al. [13])

The MSA style has similarities with the service-oriented architecture (SOA) style that emerged about 15 years ago as both styles focus on service orientation. Some scholars state that MSA is a particular implementation of SOA [2], and that MSA adopted a significant number of patterns and design principles from SOA [14]. Newman even calls MSA as SOA "done right" [3]. On the contrary, other authors see substantial differences between SOA and MSA, and they classify MSA as a new, independent architectural style (e.g., [1], [15]).

Even though MSA is used in many systems nowadays, there exist no general definition of it. The following properties and principles, however, can be found in the majority of descriptions [2], [3], [16], [17]:

- Microservices are small and organized around business capabilities.
- Microservices are self-contained. They encapsulate their data and account for their own data persistence.
- Microservices are technology- and programming language-agnostic.
- Microservices have fine-grained interfaces and use lightweight mechanisms to communicate (like RESTful HTTP). The (inter-)communication can be conducted synchronously or asynchronously.
- Microservices are loosely-coupled. Dependencies within a microservice system are minimized.
- Microservices can be independently developed, built, tested, and released.
- Microservices' autonomy reflects on the organization with decentralized governance and independent developer teams, and its promotion of DevOps culture.

MSA gained popularity due to several advantages enabled by these properties and principles. The small size of the individual services makes it easier to understand the code and its function, and, thus, facilitates the onboarding of new developers [18]. The services' autonomy makes it possible to choose the optimal technology stack for a specific task of a service (e.g., machine learning related services in Python, common user administration in Java). The independence and the loosely-coupling also help to isolate failures. A single service that fails does not necessarily take down the whole system. The system can continue working with limited service until the failing part is restarted or replaced [2]. The services' autonomy facilitates also the scalability of microservice systems. Depending on traffic and demand, services can be replicated or stopped through horizontal scaling. With containerization of services (e.g., Docker) and tools for container-orchestration (e.g., Kubernetes), MSA is predestined for running in a cloud environment.

Nevertheless, MSA is not a "silver bullet" and has its challenges. Microservices' strict autonomy implies that services have their own independent data persistence. Through this, data consistency across services becomes complicated. Data that would be stored together in one place in a monolith system might get spread out on several databases connected to dedicated microservices (see Figure 1).

The biggest challenge of MSA is the significant increase in the complexity of a system [7]. The internal complexity of an application is transferred to its outside and, thus, to its infrastructure [13]. The system relies on the functionality of the services' interfaces and the connections between them. Failures (e.g., long latency) caused by the infrastructure can compromise the system's functionality. Moreover, the exposure of a multitude of APIs increases the attack surface and, therefore, the risk of intrusion into the system.

All these aspects influence an MSA-based application and have to be taken into account during the testing of the individual microservice and the whole system.

B. Testing of microservice systems

The aim of software testing is to validate that the system works as expected. Software testing is nowadays not seen anymore as an inconvenient task after writing code but became an essential part of the software development and maintenance life-cycle. New development practices, like test-driven development, caused a shift of the focus to testing and testable code. Moreover, developers strive to automate testing as far as possible as it simplifies test execution and enables fast feedback while decreasing costs, and accelerating development. Manual testing is time-consuming, unreliable, and seen only as last resort for tests that cannot be automated (e.g., certain UI tests) or dedicated exploratory testing [19]. In traditional software architecture, like monolith architecture, testing processes are well established and researched. It is widely agreed that a combination of unit, integration, and system tests is needed to assure that a system fulfills the expected functionality [20].

With a new architectural style, like MSA, the way of testing has to be adapted. Mainly due to the autonomy of the single services and the complexity of the inter-connections, the test granularity needs to be adjusted for MSA-based applications [12]. The granularity of tests is reflected in different test types, also referred to as "test levels". There is no strict definition of the different test types and, therefore, scope and layout can vary between systems [11]. The following test types are suggested by Clemson [12] for MSA-based applications.

1) Unit Testing: A unit test aims at validating that the "smallest piece of testable software in the application" behaves as expected [12]. The size of a "unit" is not clearly defined but in object-oriented languages, it usually does not exceed a single class. If other methods or components are needed to test the unit, they are replaced by test doubles which mock their behavior. Unit tests can always run in isolation as they do not leave the scope of a single microservice.

2) Integration Testing: An integration test verifies that modules interact correctly with infrastructure and external services. In MSA-systems, these interactions are with data persistence (i.e., databases) or other microservices, within the application or beyond. Examples of integration tests in MSA-systems are API gateway integration testing or data persistence integration testing [12]. Integration tests leave the scope of the single microservice. Therefore, they either need test doubles to mock external services or must be conducted in a staging/testing environment with the dependent services deployed.

3) 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.

4) Component Testing: "A component is any well-encapsulated, coherent and independently replaceable part of a larger system." [12]. In MSA, the components are the microservice themselves. Therefore, component tests are black-box tests of the individual microservice's functionality. To be able to test the whole component in isolation and without the effects of the infrastructure, all interacting components (i.e., microservices or external APIs) are replaced with test doubles.

5) System Testing: With system (or "end-to-end") tests the functionality of the whole application is tested. They shall prove that the application and all its microservices fulfill the high-level requirements. To run such tests on a microservice system, all services and the infrastructure have to be up and running. This makes the tests expensive, time-consuming and flaky (e.g., prone to infrastructure failures). Defects are hard to track to the origin due to the large number of involved parts. A common way to run these tests is to simulate certain use cases or scenarios either by using the UI or, with headless systems, by calls to the application's public API [12]. As these tests will test the behavior of the whole system, they cannot be run in isolation and require a staging or testing environment.

C. Test pyramid

The test pyramid is a best practice recommended by Cohn [21] which describes the proportions of automated tests on the different levels in an application's testing portfolio. The original pyramid only took into consideration the execution time and test costs. In an MSA-based application, it is also relevant that tests can be run in isolation and how reliable they are. Unit tests are cheap and fast but validate only a very small part of the system. A test with a wider scope, like a system test, that uses all components of the system, will give the developer more confidence that the system works as intended. Figure 2 shows the test pyramid for MSA-based systems with, ideally,

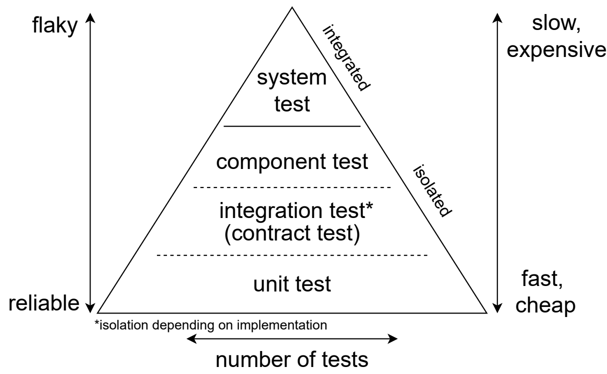

Fig. 2. Test pyramid for MSA-based applications (adapted from Clemson [12])

many unit tests and few system tests. The test pyramid is seen as best practice for MSA-based applications by a variety of practitioners (e.g. [3], [11], [12]).

The proportions of the different test types (i.e., the shape of the "pyramid") are controversial. While many practitioners support the pyramid shape (see above), other proportions, e.g. a honeycomb shape with a higher focus on integration testing and less on unit testing[22], are suggested. Nevertheless, there is a general agreement that an inverse pyramid [3] or, more figuratively, an "ice-cream cone" shape [11] is an anti-pattern that should be avoided.

III. Related Work

In this section, related work that is relevant for this study is presented.

In their systematic mapping study about testing of MSA-based applications, Waseem et al. [6] acknowledge an increasing number of articles on the topic in recent years. Most of these articles are based on case studies or experiments. They note that the most frequently mentioned test types are unit and integration tests but they claim that these types are not enough to test a MSA-based application thoroughly. Furthermore, they detected a trend towards new kinds of testing approaches (e.g., model-based approaches). However, none of these approaches has been used or investigated outside an experimental respective case study environment. The authors identify several challenges in microservice testing of which test automation, faster test feedback, and inter-communication testing are the most frequently mentioned. This study addresses the authors' suggestion to look into MSA-adapted test types, like CDC testing, and their implementation, in an in-production, real-world context.

Lehvä et al. [10] investigate how a microservice-system can be tested more effectively. They evaluate the existing testing approach with unit, component, and system tests and introduce CDC tests to facilitate the testing of integration and inter-communication of services in the system. This change has two main positive effects: Improved defect detection than with established component tests, and increased isolation of testing due to less system tests. They conclude that for the system of their case study the optimal testing strategy consists of unit, component, CDC, and system testing. Traditional integration testing can be successfully replaced by faster and more reliable CDC tests. Chen [9] describes the experiences of a company that migrated a large application to MSA to improve deployability and modifiability. For testing, the company uses CDC tests to ensure that integration points (APIs) work as expected. Additionally, it uses an always-online system-integration test environment that has all services deployed to test dependent endpoints. Chen reports also that instead of system tests, "test in production" and monitoring are used to validate the system as a whole. Chen's paper shows that CDC tests might not be enough to replace traditional integration testing, as stated by Lehvä et al. [10], but to complement it. Moreover, new approaches might be needed to test whole microservice system efficiently.

Current studies are mostly based on single cases and experiments. They suggest rather different approaches in their test strategies for complete integration testing. This study is exploring which test types apart from CDC tests and in which proportions are used in open-source systems.

IV. Research Methodology

MSA-based applications and, in particular, the testing of them is a rather new research area. Its development is mainly driven by practitioners. Therefore, collecting and analyzing the existing work of practitioners by mining publicly available software repositories seems an appropriate way to gain new knowledge. Mining software repositories (MSR) is getting more and more popular in software engineering research. Repository mining offers access to a wide range of valuable data, like source code, issues, review comments and commit messages [23]. MSR has already been applied in MSA-related studies, for example, to identify design patterns in systems [14] or to analyze the issues in projects [24]. In this study, MSR is used to collect the source code and, especially, testing-related code artifacts from open-source projects. The collected code is the basis of a postmortem analysis [25]. Postmortem analysis is used for historical data and to study the past but still has the focus on its context of occurrence. Its aim is to "capture the knowledge and experience from a specific case or activity after it has been finished" [25, p. 32].

The process is conducted in three phases. In the first phase, the research questions that steer the process are defined. In the second phase, the data sources are identified and the data are collected. In the third and final phase, data analysis is conducted on the collected data (see Figure 3).

A. Phase I: Research questions

This study aims at exploring and understanding the testing strategies in MSA-based systems that use CDC tests in a real-life, production-ready context and compare it to best practices. Therefore, the following research questions need to be answered:

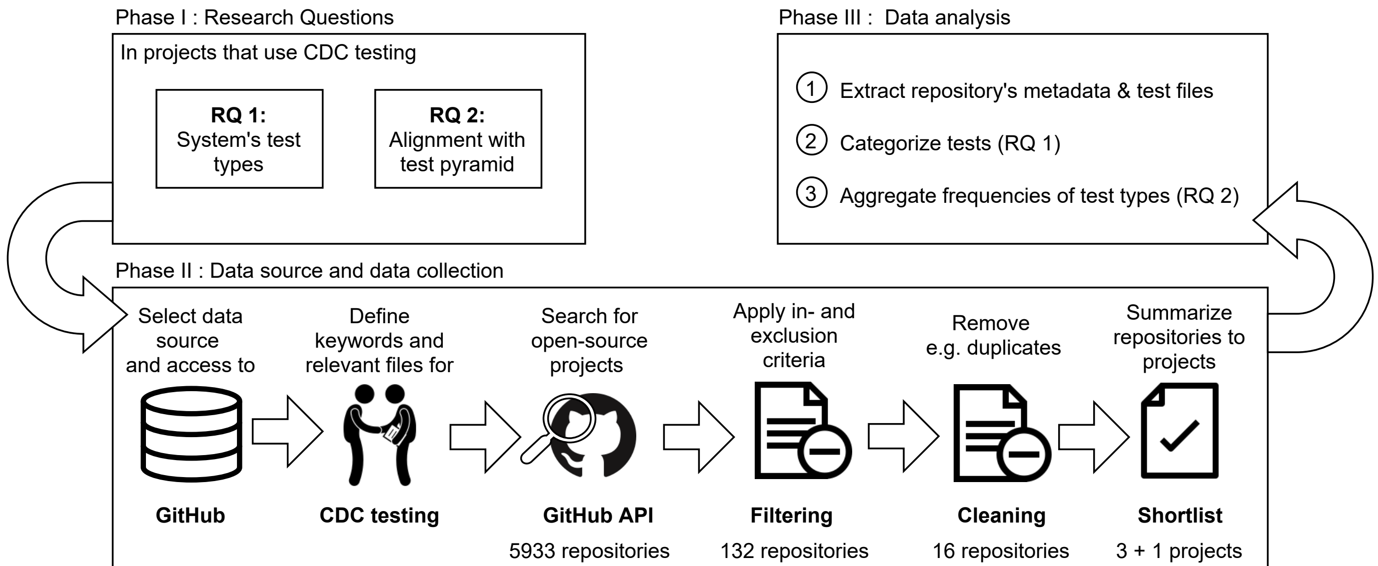

Fig. 3. Phases of the research process

RQ 1: *What test types are used in the open-source, MSA-based systems that implement CDC tests?*

This question aims to explore which different types of tests are used for the validation of open-source systems apart from consumer-driven contract testing. These types can then be used to compare with suggestions in the literature.

RQ 2: *How does the testing strategy of open-source, MSA-based systems that implement CDC tests align with the test pyramid?*

This research question aims to compare the number of test cases of the different test types and their proportions with the model of the test pyramid (see Section II-C) to evaluate how well the best practice is followed.

TABLE I
QUERIES USED TO FIND PACT AND SPRING CLOUD CONTRACT (SCC) IMPLEMENTATIONS

Filename	Extension	Keyword	Framew.	# of results
pom	xml	pact	Pact	1393
build	gradle	pact	Pact	968
docker-compose	yml	pact	Pact	348
	groovey	Contract Make	SCC	3224

B. Phase II: Data source and data collection

Phase II was conducted in five steps.

1) Selecting a data source: To answer the research questions, the source code of MSA-based, open-source systems should be used. GitHub was selected as source for repositories as it is the biggest source code hoster with approximately 200 million repositories. GitHub is used for source code storing and sharing by hobby developers, professionals, and companies (e.g. Microsoft, Netflix, or Google). To find repositories, three different ways of access to GitHub's data are suggested in the literature: GHTorrent[1], GH Archive[2] or GitHub's REST API[3] [26]. GHTorrent and GH Archive are databases generated from regular data dumps from the GitHub API. They offer stable datasets that are easy to query and are replicable. Especially GHTorrent is often used for academic research (e.g., [24]). However, neither GHTorrent nor GH Archive offer to query within the code base but only within metadata (e.g., number of commits, programming language), tags, issues, or commit messages. Furthermore, GHTorrent's most recent dataset (available during this research) dated from June 2019. As it was necessary to search within the code basis (see next step), GitHub's REST API was chosen despite its limitations (e.g., query rate and result limits). These limitations have been considered in the further process.

2) Identifying repositories through keywords: During the initial searches, it became obvious that searching by topic or label (as used to tag repositories) to identify microservices using CDC testing was not going to lead to satisfactory results. Not all repository owners used correct and/or standardized labels and, especially, the way of testing was rarely indicated. Therefore, it appeared most appropriate to focus the search on the used technology and its footprint in the source code. Consumer-driven contract testing requires additional code for implementation on the provider and the consumer side. It is possible to write own CDC test implementations as Selleby [27] shows. However, it is more common and convenient to use established solutions, like Pact[4] and Spring Cloud Contracts[5] which are both open source [28]. It was decided to focus

[1]https://ghtorrent.org/
[2]https://www.gharchive.org/
[3]https://docs.github.com/en/rest
[4]https://pact.io/
[5]https://spring.io/projects/spring-cloud-contract

on these two frameworks as they are well-documented and are the frameworks most frequently mentioned and discussed in practitioners' blogs (e.g., [29], [30]) and books (e.g., [3], [31]). Through the study of the documentation and tutorials, important keywords and code artifacts, and their occurrence in relevant files (see Table I) were identified.

3) Searching on GitHub: The keywords and filenames were used to construct search strings to query the GitHub REST API. The queries were run during May 2021. To circumvent the API limitations (i.e., maximum 1000 results per query), the file size was used as an additional parameter to receive smaller batches of results. The initial number of results can be seen in Table I and a spreadsheet with the all found repositories is available online[6]. The keywords were returning primarily microservices written in Java. The Spring Cloud Contract framework only exists for Java and, although Pact supports a range of languages, the files "pom.xml" and "build.gradle" constrained the search to implementations in Java projects. Even though this focus on Java limited the number of results, it facilitated the comparability of the projects and the results in the further process.

4) Filtering: The aim was to include only production-ready or in-production microservice systems in the analysis. An important role to fulfil this requirement played the repository owner type. GitHub distinguishes between the owner type "organization" and "user". Owner type "organization"[7] seemed appropriate to detect projects with collaborating developers how it is common for professional projects. In more detail, the following inclusion and exclusion criteria were applied:

- *Inclusion criteria:*
  - system is using the microservice architecture style (see Section II-A)
  - CDC testing is implemented
  - public repository
  - repository's owner type is "organization"
  - main programming language is Java
- *Exclusion criteria:*
  - missing or only minimal documentation
  - school projects
  - proof of concepts
  - forks or copies of tutorials/demos

A combination of automated filtering (e.g., repository's owner type, search by keywords like "example", "sample") and manual inspection of the results was used to apply these criteria on the initial dataset with the results of the queries.

5) Cleaning: Depending on the project's code structure, the initial search was triggered several times (e.g., a dependency in pom.xml file and a Pact Broker docker image in docker-compose.yml) on different files so that the same repository appeared more than once. These duplicates were removed. Some repositories could not be allocated to a (whole) MSA-application (i.e., only one single service/repository without a system) and were also removed. The final results also included a (third) governmental system which was excluded because of its size. Having over 60 microservices[8], it did not seem a feasible candidate for the used analysis method. As there was no project using the Spring Cloud Contracts framework in the shortlist, a reference implementation (P4) was added which was initially excluded.

TABLE II
LIST OF IDENTIFIED PROJECTS

#	Description	Framew.	# of MS
P1	IoT application for an intelligent football table	Pact	2
P2	*Public service platform for online payments*	Pact	6
P3	Governmental platform for licensing of exporting and importing	Pact	7
P4	*Microservice reference implementation Food delivery service*	SCC	7

6) Summarizing to projects: A complete microservice system can be stored in one repository (i.e., every service in a different folder) or distributed over several repositories (i.e., one microservice per repository). To take this into account, the remaining repositories were summarized into projects. Every project reflects a single microservice system consisting of several microservices (see Table II). Every microservice was stored in a separate repository in P1, P2, and P3; only P4 had the source code of all services stored in one repository.

C. Data analysis

The data analysis was conducted in three steps (Figure 3).

1) Extracting metadata and test-related files: The repositories of all selected projects were downloaded in June 2021. For every repository, the (main) programming language was checked and the lines of code (LOC) were extracted by using the tool CLOC[9] to illustrate the size of the individual microservices. Additionally, the tool TREE[10] was used to get an overview of the file and directory tree structure of the respective repository. Through this, the folders containing files that are relevant for testing were identified. Files that could help to understand the project's CI/CD pipeline or supported automated testing were also included.

2) Categorizing tests: In every repository, the testing-related files were inspected. For every file, a defined set of data items was extracted (see Table III). The test type categorization is guided by test type descriptions in Section II-B. The main focus lied on the scope and the level of interaction and communication within and between the individual microservices. Frameworks and testing tools, apart from Pact and Spring Cloud Contract, were also taken into consideration for the categorization. The analysis was conducted in a manual and semi-automatized way. At first, several test files were

[6]https://bit.ly/MSA-testing-mining
[7]https://bit.ly/2VLEMwZ
[8]https://hmcts.github.io/reform-api-docs/
[9]https://github.com/AlDanial/cloc
[10]http://mama.indstate.edu/users/ice/tree/

TABLE III
DATA ITEMS EXTRACTED FROM TEST FILES

#	Data item	Description
D1	Index	ID of the test
D2	Filename	Name of the test file
D3	Test type	Category of the test
D4	Quantity	Number of test cases in file
D5	Use of test doubles	Indicates the use of mocks or stubs (if applicable)
D6	Interactions	Identifies which external services the test interacts with (if applicable)

manually inspected to identify patterns. For the larger projects (P2 and P3), a Python script was used to identify keywords and assist with categorization. The automatized categorization was verified by manual inspection as recommended by Hemmati et al. [23]. The results of D3 and D5 were used to answer RQ 1.

3) Aggregating frequencies of test types: For RQ 2, the test cases per test type (determined in D3) were counted. To identify test cases in the test files, the "@Test"-annotation of the test framework JUnit[11] that all projects implemented was used. Consequently, a test case with several assertions only counted as one test case. If a "@ParameterizedTest"-annotation or "@Parameters"-annotation was used, every input parameter was counted as an individual test case. The data (i.e., data items D3 and D4) were aggregated on microservice and system level. These data were then used to compare with the test pyramid (see Section II-C). D6 was finally only used to get a better understanding of the individual microservices.

V. RESULTS

In the following section, the results from the collected and analyzed data are presented.

A. RQ 1: Test types in MSA-based projects

The answer to this research question is extracted from the source code of the four analyzed projects with altogether 22 microservices. In the projects, the following five types of tests are identified (see Figure 4).

1) Unit test: With unit tests, single methods or classes are tested. Unit tests do not exceed the scope of one microservice. In the most narrow case, with this kind of test only a method is validated (e.g., a test validates the correct functionality of a telephone number validator method (P2)). If a unit under test is dependent on other classes within the microservice, these classes are replaced by test doubles that return predefined values. In the projects, a frequently used framework to create test doubles is Mockito[12]. In all the projects which have a publicly accessible CI/CD pipeline (P2, P4), unit tests were the first tests to be run. Unit tests were identified in 21 microservices.

[11]https://junit.org/junit4/javadoc/latest/org/junit/Test.html

[12]https://site.mockito.org/

2) Integration test: With integration tests, the interaction with external modules that belong to the individual microservice is tested. In the analyzed projects, this test type is found for interactions with data persistence and a message queue service. To test these interactions, a Docker container with a database (P2, P4) or the Queue Service (P2) is created, or an "in-memory" database (P3) is used to run the test cases. To validate the interactions with a data persistence different transactions are tested (e.g., save and load, find all, etc.). Not all microservices implement integration tests because they either do not have any data persistence (P1) or the interactions with data persistence are validated in component tests (P3) instead. Integration tests were identified in 8 microservices.

3) Component test: With component tests, the functionality of a single microservice is tested, usually by API requests or messages. Different component tests are used to test particular parts of a service (e.g., by calling different endpoints). In the projects, this was conducted by running the service and, if necessary, the data persistence. Downstream microservices (i.e., their APIs) which the service under test depends on are replaced by test doubles. This applied to downstream services within the microservice system and system external services. Two projects used Wiremock[13] to create test doubles (P2, P3); Docker (Compose) is used in P4. In P1, the publishers to simulate incoming messages are mocked locally.

The component tests have a quite wide scope in all projects as they cover the microservice's functionality, interactions with data persistence, and (mocked) communication with external services (i.e., with services within and beyond the system). Component tests were identified in 14 services. Only in P4 do the developers distinguish between integration and component tests in the code. In the three other projects, integration and component tests were marked as integration tests (i.e., filename containing "IT" or "IntegrationTest"). Consequently, component and integration tests would be executed simultaneously.

4) Consumer-driven contract test: With the consumer tests, the projects validate the actual communication and integration with other services of the system. CDC tests are used to validate synchronous communication via HTTP requests (P2, P3, P4), but also asynchronous messages (e.g. in a publish/subscribe setup or a Message Queuing Service) (P1, P2, P3, P4). Through the contracts between consumer and provider, the correctness of the endpoints and the payload of the requests is verified. For asynchronous interactions, this implies, for example, the correct (subscription) topic. The contracts specify the expected return status and headers of a response, but use "matchers" to check if the type and the format (e.g., in key-value-pairs) in the response or message body are as expected.

The same contract is used in two services: On the consumer side, the contract is created and verified against the actual system. On the provider side, the contract is tested as an API request while the service is in a predefined state (e.g., "User 667 exists in the database" in the service "connector" in P2).

[13]http://wiremock.org/

Fig. 4. Test types and auxiliary tools/frameworks identified in the projects

The response is compared to the expectations in the contract. The contracts are stored in a shared folder in a repository (P1, P4), or a designated service (i.e., Pact Broker[14]) for storing and administrating the contracts is implemented (P3, P2). CDC tests were found in all analyzed microservices as it was one of the inclusion criteria.

5) System test: With system (or end-to-end) tests, the whole microservice system is tested for certain scenarios. In the analyzed projects, only in two projects, end-to-end tests were found (P2, P4). They are conducted either through mocked actions on the user interface or a row of API calls simulating a user journey through the system. For these tests, all microservices of the system and necessary additional services (e.g., databases) are deployed in containers with docker-compose. Services beyond the system are replaced by containerized test doubles (P2).

[14] https://github.com/pact-foundation/pact_broker/

B. *RQ 2: Comparison of test proportions with test pyramid*

For this research question, the number of test cases per test type on the repository level is aggregated (see Table A1-A4). Then it is converted to proportions to facilitate the comparison with the test pyramid. For a better understanding of the results, the characteristics of each project are outlined briefly.

1) P1: P1 (see Figure 5; for absolute numbers see Table A1) consists of four microservices. Two of these services are written in Java and were analyzed. The main characteristic of this system is that the services only communicate via asynchronous messaging using the publish/subscribe-pattern and a MQTT broker. None of the services has a database thus there is no need to test interactions with data persistence (i.e., no integration tests). To test the individual services (component tests), an instance of the MQTT is started and different messages are published to the broker to test the service's correct reaction and output.

As Figure 5 shows, P1 is the smallest system with quite small microservices. The size is also reflected in the low

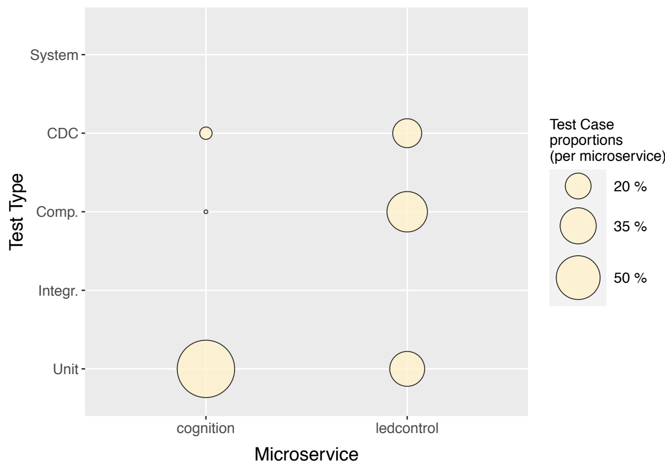

Fig. 5. Test proportions in P1

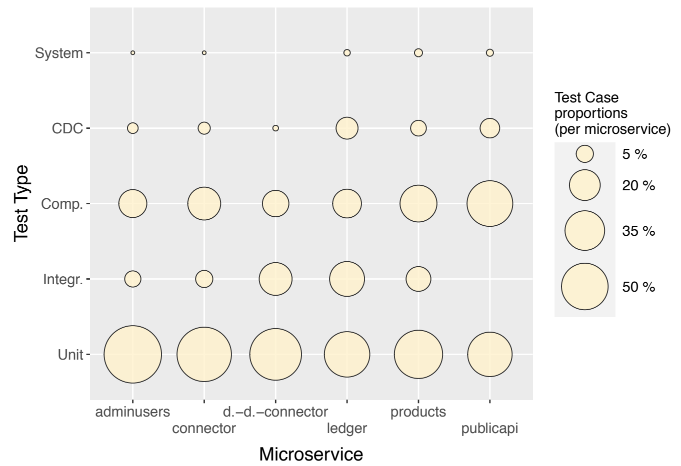

Fig. 6. Test proportions in P2

number of tests. The service "cognition" has a large base of unit tests, a few component tests, and proportionally many CDC tests. The high number of CDC tests and, thus, contracts, indicate a high number of possible interactions up- and downstream as it is acting as consumer and producer for other microservices. The test frequencies of the second service, "ledcontrol", appear rather in the shape of a honeycomb. In this case, the developers focused more on testing the functionality of the whole service than the internal modules. In the repositories, no system tests were found. They might be conducted manually or the developers reached enough confidence about the system with the other tests. Looking at the total number of automated tests conducted in P1 (see Figure 9), there is a big base of unit tests, and less component and CDC tests.

2) Project P2: P2 (see Figure 6; for absolute numbers, see Table A2) consists of ten microservices of which six are written in Java and implement CDC testing. These six services are part of the analysis. P2 is the largest analyzed project - in terms of services' size and the total size of the system. It allows extended insights into its CI/CD pipeline with a separate public CI/CD repository[15].

All microservices of P2 apart from the "publicapi" have their own data storage which are tested with integration tests. The communication between services is mostly conducted via HTTP calls with some additional communication via a message queue system to communicate events (e.g., payments) from "connector" to "ledger" service. The AmazonSQS Queue service is used for message implementation. The queue service is seen as a service in the sphere of a microservice, like databases, and tests covering the interactions to it were coded as integration tests.

The "connector" service has several connections to external payment services that are replaced by test doubles. Communication with the internal service "publicAuth" is also always replaced by test doubles and not tested with CDC tests. The same applies to some interactions with the internal service "direct-debit-connector". Several of the CDC tests are establishing contracts with frontend services, like "selfservice" or "product-ui", which are not in the scope of this analysis as they are written in JavaScript.

All microservices follow well the test pyramid's shape except for "publicapi". They have a broad base of unit tests and a very small number of system tests. Like in the other projects, the number of integration tests varies and is not always directly proportional to the microservice's size. Although the system has quite large individual services and is sufficiently complex, the developers obviously reach enough confidence with unit, integration, component, and CDC tests to validate the system's functionality so that only a minimal number of system tests is needed before deployment. The effort for system tests is further decreased as the number of these tests depends on the actual newly deployed microservice. Interestingly for this project is that publicly accessible coding guidelines encourage developers to follow the test pyramid[16].

3) Project P3: P3 (see Figure 7; for absolute numbers, see Table A3) consists of seven services that are all written in Java and implement CDC testing. The service "permission-finder" operates as frontend and, therefore, contains HTML code. Three services ("user", "customer", "country") work as a proxy and cache calls to a system external API ("SPIRE"). For testing purposes, all requests to "SPIRE" are replaced by test doubles and are not covered by CDC tests as the API is outside of the control of P3. The services communicate via HTTP calls, apart from a message queue to between the "permissions-finder" and "notification" services. P3 includes no publicly accessible, data or files describing the CI/CD pipeline of individual microservices or the system. For this reason, it is not clear whether the project uses system tests.

As Figure 7 shows, "permissions-finder" conducts the most CDC tests which indicates that it has many interactions with other services. It is mostly a consumer of other services, apart

[15]https://github.com/alphagov/pay-ci/

[16]https://bit.ly/3ABbcc2

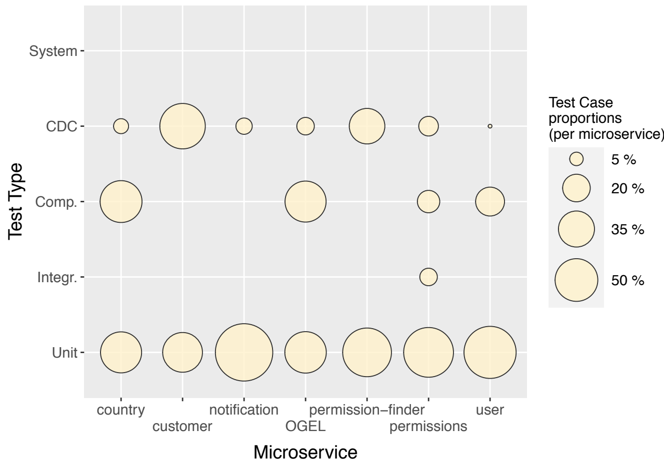

Fig. 7. Test proportions in P3

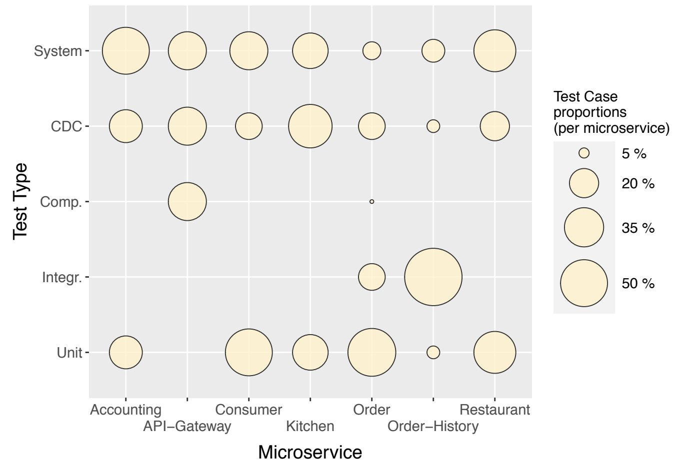

Fig. 8. Test proportions in P4

from a publishing function with the "notification" service. "Permissions-finder", like "notification" and "customer", is not validated by any component tests. The services that act as proxies do not test databases or caches specifically but include these interactions in their component tests. The "OGEL" service validates its interactions with data persistence also only by component tests.

Comparing the individual services with the test pyramid, the results are ambiguous. While the largest backend service "permissions" follows well the pyramid shape, other services have a more dominant number of component respectively CDC tests, especially compared to the number of unit tests. In total, for the whole system (see Figure 9), there is a pyramid shape recognizable when disregarding the low number of integration tests.

4) Project P4: P4 (see Figure 8; for absolute numbers, see Table A4) is a reference implementation of a microservice system and is the only project that uses the Spring Cloud Contract framework for contract testing. It consists of eight microservices where only seven were included. The service "Delivery" service was excluded as it did not implement CDC tests. As it is a reference implementation, there is a focus on the implementation of different test types but not on high test coverage, unlike in a production system. As Figure 8 and 9 show, this results in proportionally many tests with a large scope and few unit tests. Therefore, it was decided to exclude the system from the analysis for RQ 2. It cannot be an adequate representation for an in-production system and does not return realistic results for a comparison.

VI. Discussion

In this section, the results of the four analyzed projects with overall 22 microservices are discussed. The limited number of projects is caused, on the one hand, by the narrow inclusion and exclusion criteria, and, on the other hand, by the limited number of production-ready microservice applications publicly available on GitHub. The curated list of MSA-based

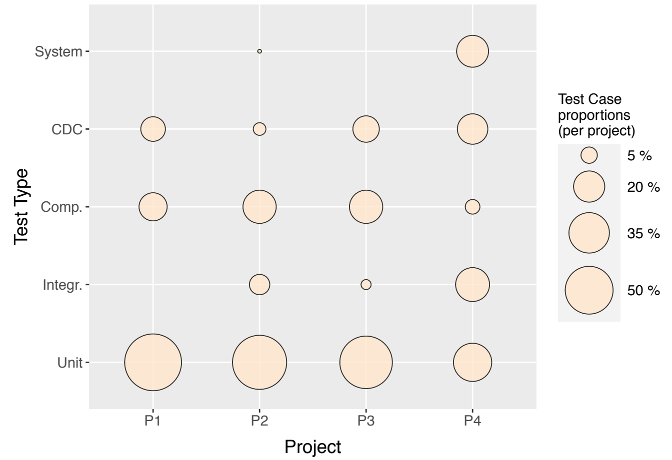

Fig. 9. Aggregated test proportions on project level

systems by the researcher Davide Taibi[17] counts only ten professional open-source systems; none of them is using CDC testing. MSA is still used mainly by large, commercial actors who have the need for scalability and flexibility, but also have the financial and personnel capacities to handle the MSA's complexity and necessary infrastructure. These actors, of course, rarely make their code open source. On the other hand, certain government agencies have open-source policies which explains why two of the identified projects are public service platforms.

A. RQ 1: Test types

In the analyzed projects, five testing types have been identified: unit, integration, component, CDC, and system tests. The identified testing types follow, for the most part, the definitions in Section II-B, but have been slightly adapted to the analyzed systems. The biggest adaption is made in regard to the integration test. They are found to be used only for

[17]https://github.com/davidetaibi/Microservices_Project_List

testing interactions with data persistence and queue services, and not specifically for testing inter-communication with other services. In the analyzed systems, this was expected as all systems implemented CDC tests that could take care of testing the inter-communication with services in the system. However, several exceptions were identified, for example, in P2 where communication with the Authentication service ("publicAuth" service) was not covered with CDC tests. It could not be determined why not all inter-service communication was covered by contracts. It might be caused by missing collaboration with the other team or potential challenges using CDC testing with, for example, authentication services.

The integration respectively communication with external services is to a certain extend also covered in component tests. Component tests can be described as black-box system tests on microservice level as they validate certain functionality or endpoints of a whole microservice. If external services are needed for such tests, they are replaced by test doubles. A challenging aspect is the trustworthiness of these test doubles [8]. Component tests can still pass and, thus, miss defects if a test double did not get adapted to changes in a downstream service's API. This is also stated by Lehvä et al. [10] as an possible issue with component tests. Furthermore, for testing the communication with external services which are beyond the system, there are no alternatives to test doubles, as Vocke [11] states. In the analyzed projects, this is seen, for example, with external payment services (P2) and a third-party service (P3). For external services within the system, the inter-communication can explicitly be verified with a CDC test which should break if a downstream service changes its API unknowingly. This raises the question which tests to prioritize in the testing pipeline; CDC tests to validate the inter-communication or component tests to validate the functionality of the microservice itself. A possible solution could be to use the contracts to also verify the correctness of the mocks. This would solve the trustworthiness issue but add another level of complexity.

A general aspect to consider are the test types themselves. Test types are not standardized and are often open to the developers' interpretation as mentioned in Section II-B. The scopes and implementations of the tests, especially of unit, integration, and component tests, can differ between systems and even between microservices (i.e., because of teams' autonomy to decide about their own testing strategy). This observation matches with Vocke's [11] and Fowler's [32] remarks on the various interpretability of unit respective integration/component tests. Because of this ambiguity, Newman suggested a more pragmatic approach and declared all tests between unit test and end-to-end test (system test) level as "service" tests [3]. In the analyzed systems, this could be observed in the missing differentiated between integration and component tests, according to the naming of the files. As component tests are not common in monolithic architecture, the absence of the distinct naming of "component tests" might root from a missing adaption of testing processes to the distinct characteristics of MSA. However, a differentiation might be very useful in practice to detect defects as soon as possible in the testing process by defining clear testing layers or levels. For example, a failing integration test will clearly pinpoint to a problem with the interaction with the data persistence. It would stop the test pipeline from running more tests with a bigger scope, like component tests.

B. RQ2: Alignment with test pyramid

The comparison between the test pyramid and the test proportions in the project's microservices gives an ambiguous picture. In the smaller systems with small services and few tests, the pyramid shape is not very distinctive, if existent at all. However, in bigger systems with larger microservices, the pyramid-shaped proposition becomes rather obvious.

A closer look at P2, the largest analyzed system, shows that most of its microservices follow the pyramid shape quite distinctly. The pyramid of the individual microservice is "topped" by only between zero to two system tests. The small number of system tests implies that the other tests that can be conducted in isolation from the remaining system give enough confidence that not only the microservice itself works as expected but also that its interactions and integration into the whole system do. These results match with the findings of the case study by Lehvä et al. [10]. CDC tests help to decrease the number of flaky and expensive system tests and, at the same time, extend the possibility of isolated tests to also cover the services' communication. For P1 and P3, this cannot be confirmed or denied as it is not clear whether they implemented system tests. Still, it can be assumed that the CDC tests helped the isolated testing of the individual services, otherwise the developers would probably not have implemented them.

In general, there are several aspects to consider when using the test pyramid as reference that became obvious from the results. These aspects might be especially relevant for practitioners.

First, with the increasing size, a microservice's complexity amplifies and, thus, the need for testing of an increased number of parts. This is reflected in particular in the number of unit, component, and, to a certain degree, CDC tests. For integration tests, which were not found in all microservices, a relation to the size is less marked. The actual function and its use of data persistence seem more relevant for the number of integration tests. For CDC tests, the role of the service in the system (e.g., a gateway with communication with many services) influences the number of necessary tests. Therefore, the number of integration and CDC tests often does not show the same proportional relation to a microservice's size as the other tests. This fact can be found in several microservices that have either no number of integration tests due to no data storage or a proportionally high number of CDC tests due to central inter-communication function.

Second, the system's size is to consider. The projects that are part of this study are rather small compared to systems of larger companies, like Netflix. In the analyzed systems, it is observed that the test type proportions aligned better to the

pyramid shape with increasing system and microservice size. For an MSA-based application an increase in size means that the number of microservices (i.e., their quantity) increases. The individual microservices should still follow the principle of being "small" and not exceed a certain size. On the microservice level, the size of the system does therefore not significantly influence the number of unit, integration, and component tests. Here, the pyramid can account as a valid reference. However, with the size, the number of interconnections, the service integration effort, and the general system complexity will increase. To a certain extent, it will be feasible to cover these points with CDC tests and system tests. But, above a certain system size, as Chen[9] and Sridharan [33] note, these tests might have to be replaced or supported by other methods, like in-production testing and extended monitoring.

Finally, there is no consensus in the literature about the order of test types between unit and system tests. In Section V), the test type order from Lehvä et al. [10] was applied. It arranges the test types according to the decreasing isolation of the tests and, thus, their increasing scope. Whereas Clemson [12] (see Figure 2) focuses with "his" pyramid more on the effort (e.g., time and costs) and stability of the test types. The deviating recommendations which are also related to the discussion in RQ 1 (VI-A) about test types. This ambiguity might make it difficult for practitioners to adopt test pyramid as guideline.

VII. Threat to validity

In the following section, threats to the internal and external validity of this study are presented and the countermeasures which were applied to mitigate them are described.

A. Internal validity

Internal validity refers to the methodological strategy to minimize bias in the data collection and analysis, and, ultimately, in the results. Three potential threats to the internal validity of this research have been identified: (i) the selection of CDC frameworks and search terms for these frameworks, (ii) filtering and selection of projects, (iii) categorization of test types. The following steps were taken to mitigate these threats:

- *Bias on the selection of CDC test frameworks and search terms*: The two frameworks were carefully selected according to discussions on practitioners' blogs (e.g., [30]) and in research literature [28]. They seemed most relevant for a study aiming for in-production or production-ready projects. Deprecated frameworks, like Pactio, or self-written implementations of CDC testing exist [27], so that there is a risk that projects using "isolated solutions" might have been missed. The queries to identify the frameworks were thoroughly selected and tested in different combinations of keywords and filenames. It was still a trade-off between too broad (i.e., too much noise) and too narrow search queries.
- *Bias on filtering and selection of projects*: For the search results, transparent exclusion and inclusion criteria were formulated. The filtering and selection process was conducted in a multi-step approach (described in IV-B4) to minimize possible threats. Furthermore, a spreadsheet with the raw data and the results after each step is available online[18]. Additionally, links, version tags and commit hashes of all inspected repositories can be found in Appendix B1.
- *Bias on test type categorization*: To avoid bias in the categorization procedure, it was conducted in an iterative way as new knowledge was gained continuously from additional test cases. The categorization criteria were constantly adapted and updated. Furthermore, a Python script was used to facilitate and automatize the categorization process. The results of the automated categorization were inspected and manually verified as recommended by Hemmati et al. [23]. A spreadsheet with the test files and the results for all projects is provided online[19].

B. External validity

Threats to external validity are concerned with the potential generalization of the results. It describes to what extent the findings are applicable beyond this paper and the analyzed projects. The research was scoped, among other factors, to systems using CDC tests (i.e., Pact or Spring Cloud Contract Framework) and written in Java. Furthermore, it was limited by the number of publicly available open-source projects. With the resulting (small) number of projects from only one source code hoster (GitHub), findings cannot be statistically generalized. Nevertheless, an analytical generalization that allows the transfer of the results to projects with similar characteristics seems possible. That means that the results of this study could be applied to microservice systems which are using a CDC implementation, are written in Java and have a comparable number of microservices.

VIII. Conclusions

In this paper, the testing strategies of MSA-based in-production or production-ready open-source systems that are using consumer-driven contract testing were studied and compared to the test pyramid. By mining software repositories on GitHub, four different projects with altogether 22 relevant microservices were identified and analyzed.

In these microservices, five test types could be classified: unit, integration, component, CDC, and system tests. These test types were mostly equal to the test types described in the literature. However, integration testing focused on testing interactions with data persistence or queue services, thus, with modules related to the individual service, and not to test communication with other services in the system. This was instead mostly covered by CDC tests. Less than half of all systems used integration tests. In three of four projects, developers did not distinguish between integration and component tests.

[18] https://bit.ly/MSA-testing-mining
[19] https://bit.ly/MSA-testing-results

This might lead to the use of more "expensive" component tests, instead of finding the defect in an earlier state with less complex integration tests. It was suggested that this could be caused by a missing adaption of testing strategies from monolithic to microservice architecture.

The aggregated tests per test types were compared to the test pyramid. The test pyramid indicates the best practice to test a system with a proportional high quantity of fast unit tests and a very small number of slow, expensive system tests. The comparison showed that the larger the system and the individual microservices, the better test proportions align with the pyramid shape. Nevertheless, it also became obvious that the pyramid itself is not an ideal reference for every microservice as the test types can depend on the services' role and function. The exact proportions (i.e., the pyramid's shape) and the test types' order in the pyramid are under discussion by scholars and practitioners. These discussions might lead to further adaptions of best practices regarding test types and test proportions to the microservices architecture, especially for large-scale systems.

Future work should compare the testing strategy between MSA-systems with and without CDC tests and what the differences are in terms of implemented test types and number of, especially, component and system tests. Moreover, another aspect for future research could be the influence of the system's size on the testing strategy and, whether alternative testing methods and techniques (e.g., in-production-testing, extended monitoring and logging) have to be considered.

Acknowledgment

First of all, I would like to thank my supervisor, Hamdy Michael Ayas, for his continuous support and feedback, and his guidance during this thesis writing process. I also would like to thank Richard Berntsson Svensson, my examiner, for the valuable feedback. Last but not least, I have to thank my parents and my friend who backed my decision to change careers and read this bachelor's program.

References

[1] J. Lewis and M. Fowler, *Microservices*, 2014. [Online]. Available: https://martinfowler.com/articles/microservices.html (visited on 11/15/2020).

[2] O. Zimmermann, "Microservices tenets: Agile approach to service development and deployment," *Computer Science - Research and Development*, vol. 32, no. 3-4, pp. 301–310, Jul. 2017.

[3] S. Newman, *Building microservices: designing fine-grained systems*, 1st. Beijing Sebastopol, CA: O'Reilly Media, 2015.

[4] *GOTO 2016 • Microservices at Netflix Scale: Principles, Tradeoffs & Lessons Learned*, Feb. 2016. [Online]. Available: https://www.youtube.com/watch?v=57UK46qfBLY (visited on 03/25/2021).

[5] A. Gluck, *Introducing Domain-Oriented Microservice Architecture*, Jul. 2020. [Online]. Available: https://eng.uber.com/microservice-architecture/ (visited on 07/25/2021).

[6] M. Waseem, P. Liang, G. Márquez, and A. D. Salle, "Testing Microservices Architecture-Based Applications: A Systematic Mapping Study," in *2020 27th Asia-Pacific Software Engineering Conference (APSEC)*, Dec. 2020, pp. 119–128.

[7] J. Soldani, D. A. Tamburri, and W.-J. Van Den Heuvel, "The pains and gains of microservices: A Systematic grey literature review," *Journal of Systems and Software*, vol. 146, pp. 215–232, Dec. 2018.

[8] V. Lenarduzzi and A. Panichella, "Serverless Testing: Tool Vendors' and Experts' Points of View," *IEEE Software*, vol. 38, no. 1, pp. 54–60, 2021.

[9] L. Chen, "Microservices: Architecting for Continuous Delivery and DevOps," in *2018 IEEE International Conference on Software Architecture (ICSA)*, Apr. 2018, pp. 39–397.

[10] J. Lehvä, N. Mäkitalo, and T. Mikkonen, "Consumer-Driven Contract Tests for Microservices: A Case Study," in *Product-Focused Software Process Improvement*, X. Franch, T. Männistö, and S. Martínez-Fernández, Eds., Cham: Springer International Publishing, 2019, pp. 497–512.

[11] H. Vocke, *The Practical Test Pyramid*, Feb. 2018. [Online]. Available: https://martinfowler.com/articles/practical-test-pyramid.html (visited on 02/04/2021).

[12] T. Clemson, *Testing Strategies in a Microservice Architecture*, Nov. 2014. [Online]. Available: https://martinfowler.com/articles/microservice-testing/ (visited on 01/30/2021).

[13] D. Savchenko, G. Radchenko, T. Hynninen, and O. Taipale, "Microservice Test Process: Design and Implementation," vol. 10, p. 12, 2018.

[14] G. Márquez and H. Astudillo, "Actual Use of Architectural Patterns in Microservices-Based Open Source Projects," in *25th Asia-Pacific Software Engineering Conference (APSEC)*, 2018, pp. 31–40.

[15] D. Gupta, M. Palvankar, and C. T. Solutions, "Pitfalls & Challenges Faced During a Microservices Architecture Implementation," Tech. Rep., Feb. 2020, p. 21.

[16] S. Eismann, C.-P. Bezemer, W. Shang, D. Okanović, and A. van Hoorn, "Microservices: A Performance Tester's Dream or Nightmare?" In *Proceedings of the ACM/SPEC International Conference on Performance Engineering*, Edmonton AB Canada: ACM, Apr. 2020, pp. 138–149.

[17] M. Waseem, P. Liang, and M. Shahin, "A Systematic Mapping Study on Microservices Architecture in DevOps," *Journal of Systems and Software*, vol. 170, p. 110 798, Dec. 2020.

[18] J. Lotz, A. Vogelsang, O. Benderius, and C. Berger, "Microservice Architectures for Advanced Driver Assistance Systems: A Case-Study," in *2019 IEEE Interna-*

tional Conference on Software Architecture Companion (ICSA-C), Mar. 2019, pp. 45–52.

[19] D. Spinellis, "State-of-the-Art Software Testing," *IEEE Software*, vol. 34, no. 5, pp. 4–6, 2017.

[20] M. Pezzè and M. Young, *Software testing and analysis: process, principles, and techniques*. Oct. 2008. (visited on 01/30/2021).

[21] M. Cohn, *Succeeding with agile: software development using Scrum*, ser. The Addison-Wesley signature series. Upper Saddle River, NJ: Addison-Wesley, 2010.

[22] A. Schaffer and R. Dybeck, *Testing of Microservices*, Blog, Jan. 2018. [Online]. Available: https://engineering.atspotify.com/2018/01/11/testing-of-microservices/ (visited on 02/16/2021).

[23] H. Hemmati, S. Nadi, O. Baysal, O. Kononenko, W. Wang, R. Holmes, and M. W. Godfrey, "The MSR Cookbook: Mining a decade of research," in *2013 10th Working Conference on Mining Software Repositories (MSR)*, May 2013, pp. 343–352.

[24] M. Waseem, P. Liang, M. Shahin, A. Ahmad, and A. R. Nasab, "On the Nature of Issues in Five Open Source Microservices Systems: An Empirical Study," p. 11, 2021.

[25] C. Wohlin, M. Höst, and K. Henningsson, "Empirical Research Methods in Software Engineering," in *Empirical Methods and Studies in Software Engineering: Experiences from ESERNET*, ser. Lecture Notes in Computer Science, R. Conradi and A. I. Wang, Eds., Berlin, Heidelberg: Springer, 2003, pp. 7–23.

[26] T. Mombach and M. T. Valente, "GitHub REST API vs GHTorrent vs GitHub Archive: A Comparative Study," Tech. Rep., 2018, p. 8.

[27] F. Selleby, "Creating a Framework for Consumer-Driven Contract Testing of Java APIs," Bachelor's Thesis, Linköping University, Linköping, Sweden, 2018.

[28] J. P. Sotomayor, S. C. Allala, P. Alt, J. Phillips, T. M. King, and P. J. Clarke, "Comparison of Runtime Testing Tools for Microservices," in *2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)*, vol. 2, Jul. 2019, pp. 356–361.

[29] A. Weiss, *How to Test Microservice Integration with Pact*, Oct. 2017. [Online]. Available: https://codefresh.io/docker-tutorial/how-to-test-microservice-integration-with-pact/ (visited on 03/22/2021).

[30] M. Sokola, *How to test Microservices with Consumer-Driven Contracts?* 2018. [Online]. Available: https://hackernoon.com/how-to-test-microservices-with-consumer-driven-contracts-9bf5c2c05349 (visited on 03/11/2021).

[31] A. Soto Bueno, A. Gumbrect, and J. Porter, *Testing Java microservices: using Arquillian, Hoverfly, AssertJ, JUnit, Selenium, and Mockito*. Shelter Island, NY: Manning Publications Co, 2018.

[32] M. Fowler, *IntegrationTest*, Jan. 2018. [Online]. Available: https://martinfowler.com/bliki/IntegrationTest.html (visited on 07/07/2021).

[33] C. Sridharan, *Testing Microservices, the sane way*, Blog, 2017. [Online]. Available: https://copyconstruct.medium.com/testing-microservices-the-sane-way-9bb31d158c16 (visited on 05/24/2021).

APPENDIX A
RESULT TABLES

TABLE A1
TEST STRATEGY OF PROJECT P1

Microservice	**Language**	**LOC**	**Unit tests**	**Integration tests**	**Component tests**	**CDC tests**	**System tests**	**Additional tests**
cognition	Java	4210	58	0	5	6	0	architecture
ledcontrol	Java	2125	7	0	9	5	0	
Total	–	–	65	0	14	11	0	

TABLE A2
TEST STRATEGY OF PROJECT P2

Microservice	**Language**	**LOC**	**Unit tests**	**Integration tests**	**Component tests**	**CDC tests**	**System tests***	**Additional tests**
products	Java	9231	94	21	52	7	1	smoke tests
ledger	Java	19800	177	99	64	34	1	smoke tests
publicapi	Java	26255	241	0	256	37	2	smoke tests
adminusers	Java	21198	677	37	139	12	1	smoke tests
connector	Java	84760	1635	118	543	48	1	smoke tests
direct-debit-connector	Java	22809	342	126	77	1	0	smoke tests
Total	–	–	3166	401	1131	139	6	

* Two generic system tests (“card”, “products”); test(s) selection depends on deployed microservice

TABLE A3
TEST STRATEGY OF PROJECT P3

Microservice	Language	LOC	Unit tests	Integration tests	Component tests	CDC tests	System tests	Additional tests
permissions	Java	7310	94	11	18	13	0	
customer	Java	2562	15	0	0	20	0	
user	Java	2852	79	0	23	2	0	
country	Java	2746	41	0	43	5	0	
OGEL	Java	4287	48	0	47	8	0	
permission-finder	Java/HTML	10559	61	0	0	31	0	
notification	Java	766	13	0	0	1	0	
Total	–	–	351	11	131	80	0	

TABLE A4
TEST STRATEGY OF PROJECT P4

Microservice	Language	LOC	Unit tests	Integration tests	Component tests	CDC tests	System tests*	Additional tests
Accounting	Java	515	1	0	0	1	2	
API Gateway	Java	459	0	0	2	2	2	
Consumer	Java	436	3	0	0	1	2	
Kitchen	Java	858	2	0	0	3	2	
Order History	Java	1092	1	12	0	1	2	
Order	Java	3306	12	4	1	4	2	
Restaurant	Java	440	2	0	0	1	2	
Total	–	–	21	16	3	13	14	

* Same system tests are run for every service

Appendix B
Project data

TABLE B1
LINKS, TAGS AND COMMIT HASHES OF ANALYZED PROJECTS

Project	Microservice	URL	Tag	Hash
P1				
	cognition	https://github.com/smart-football-table/smart-football-table-cognition	n/a	6e2945c
	ledcontrol	https://github.com/smart-football-table/smart-football-table-ledcontrol	n/a	e348446
P2				
	adminusers	https://github.com/alphagov/pay-adminusers	alpha_release-879	fca2f16
	connector	https://github.com/alphagov/pay-connector	alpha_release-2253	04c7a69
	direct-debit-connector	https://github.com/alphagov/pay-direct-debit-connector	alpha_release-716	43ed155
	ledger	https://github.com/alphagov/pay-ledger	alpha_release-860	032039d
	products	https://github.com/alphagov/pay-products	alpha_release-563	3fc69c8
	publicapi	https://github.com/alphagov/pay-publicapi	alpha_release-846	4717ef6
P3				
	permissions	https://github.com/uktrade/lite-permissions-service	20190501.093137	f736aee
	customer	https://github.com/uktrade/lite-customer-service	20190501.091218	5311b07
	user	https://github.com/uktrade/lite-user-service	20190430.153446	38fe944
	country	https://github.com/uktrade/lite-country-service	20190430.155728	0f009ac
	OGEL	https://github.com/uktrade/lite-ogel-service	20190430.101159	fd5abd3
	permission-finder	https://github.com/uktrade/lite-permissions-finder	20200128.120620	ecd87ab
	notification	https://github.com/uktrade/lite-notification-service	20200128.120829	8bf0932
P4				
	Accounting API Gateway Consumer Kitchen Order History Order Restaurant	https://github.com/microservices-patterns/ftgo-application*	0.1.0.RELEASE	a835e23

* All microservices stored in a single repository