Consumer-Driven Contract Tests for Microservices: A Case Study

Jyri Lehvä, Niko Mäkitalo, and Tommi Mikkonen

Department of Computer Science
University of Helsinki, Helsinki, Finland
{jyri.lehva, niko.makitalo, tommi.mikkonen}@helsinki.fi

Abstract. 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.

Keywords: Consumer-driven contract testing · Design by contract · Microservices · Test planning · Integration testing · Test coverage.

1 Introduction

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.

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.

2 Background

Microservice architecture is a relatively new approach to architecting systems that are updated continuously [5]. The fundamental goal of microservices is to make each service self-contained, even if this means implementing similar (or even the same) functions in numerous services [6]. In other words, the goal is to minimize dependencies between the services, so that they can be designed and deployed independently of the other parts of the system [7]. Furthermore, different tools and techniques can be used when implementing microservices because they only need to interact with each other using well-defined APIs [11]. Therefore, designers can apply various techniques in testing individual services. However, when orchestration of numerous microservices is required, a common testing approach is needed to test their interaction. This is often visualized with testing pyramid (Figure 1).

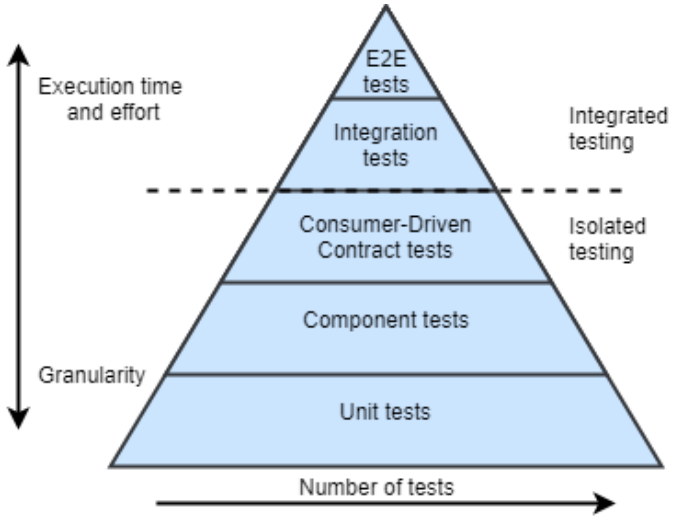

Fig. 1. Test pyramid with consumer-driven contract tests. Adapted from [3].

Since microservices are typically deployed directly to a live environment, traditional end-to-end and integration tests can be challenging to organize. Instead,

a testing approach is needed where microservices can be tested in isolation from the live system, and preferably so that the developers can easily run the tests on their machines.

Consumer-driven contract testing is an approach that allows testing both sides of an integration separately and isolated from each other. It relies on consumer-driven contracts [9] between a consumer and a provider, following the design-by-contract paradigm [4]. They are created by the consumer and then shared to the provider for verification so that each contract describes a set of interactions between the consumer and the provider. A single interaction is a pair of request and response describing how the services communicate with each other. From the consumers perspective, the interaction describes the outgoing request and the response for it from the provider. From the providers perspective, the interaction defines what kind of incoming request it receives from the consumer and what kind of response the consumer expects to be generated for it.

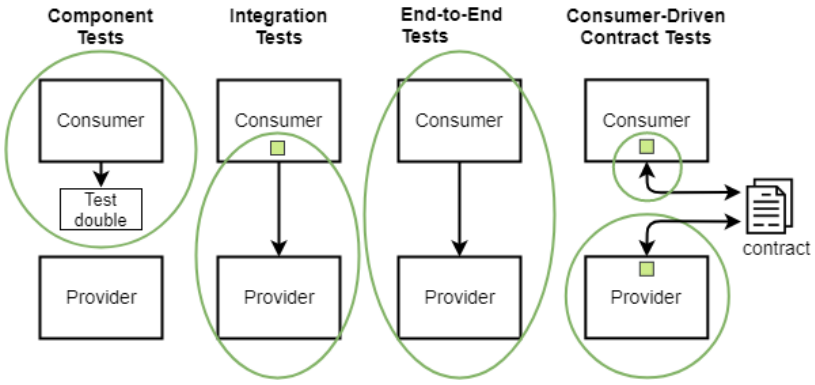

Fig. 2. Scopes of different testing methods.

From the surface, consumer-driven contract testing resembles integration testing – both involve a service provider and a service consumer, and their interaction is being tested. However, in consumer-driven contract tests, they are isolated from each other by an explicit contract instead of being directly connected, whereas in integration testing, more liberal interactions are typically allowed. The catch is that instead of forming a connection between the services, the test is divided into two independent and isolated stages (Figure 2). *In the first stage*, the consumer creates a contract containing the details of each interaction it requires from the provider. The contract is then shared with the provider. *In the second stage*, the provider uses the contract to test its API. After the provider has successfully verified the contract with the tests, the consumer and the provider know they are compatible with each other.

The verified consumer-driven contract describes a specific state of the integration between the consumer and the provider. If that state changes from either side, there is a high chance of introducing defects to the integration. As explained earlier, the consumer and the provider can both be tested based on the contract. The consumer testing can be achieved by comparing the newly implemented changes to the past state of the consumer. That can be done by using a mock that is based on the previous version of the contract. If the mock fails when the consumer sends the request to the provider, it means the implementation has changed, and the consumer no longer obeys the contract. If the consumer changed it on purpose, it is considered as a proposal for a new version of the contract. After that, the mock should be updated to match the changes, and the contract must be verified again by the provider to make sure it is compatible with those changes.

The provider side verification of the contract is done by playing the consumers requests from the contract against the provider and comparing the provider responses to the expected responses from the contract. If they match, the contract is satisfied, and both the consumer and the provider are compatible with each other. Sometimes the provider needs to make breaking changes. In such situations, the changes should be communicated with the consumers. After that, the consumers can create new versions of the contracts that take the breaking changes into account and enables the provider to evolve as planned.

Obviously, consumer-driven contract tests aim at a very specific point in development. Hence, they must be complemented with other types of testing that have been traditionally executed. To understand the exact benefits of consumer-driven contract testing, we conducted a case study in cooperation with a commercial company and its production system.

3 Case Study

3.1 Overview

Our case study is based on a system built on microservice architecture, consisting of of eight services and four databases. The purpose of the system is to enable admin users to create custom product configurations to be sold in web stores to customers. Those web stores are consuming the APIs of the system. The system keeps track of stock levels for the products in different warehouses and provides tools for warehouse workers to fulfill orders placed by the customers.

An overview of the microservice system is shown in Figure 3. The development team owns the microservices inside the black box. Other teams own the rest of the microservices. The arrows between the services are pointing from consumer to provider direction or in other words, from downstream to upstream. Two external calls from the Shop are highlighted with green color. Those are calling the endpoints that are in the focus of this case study. Integrations from web apps, external APIs, and the rest of the shops that act as consumers of the system have been abstracted away to reduce the noise from the two endpoints.

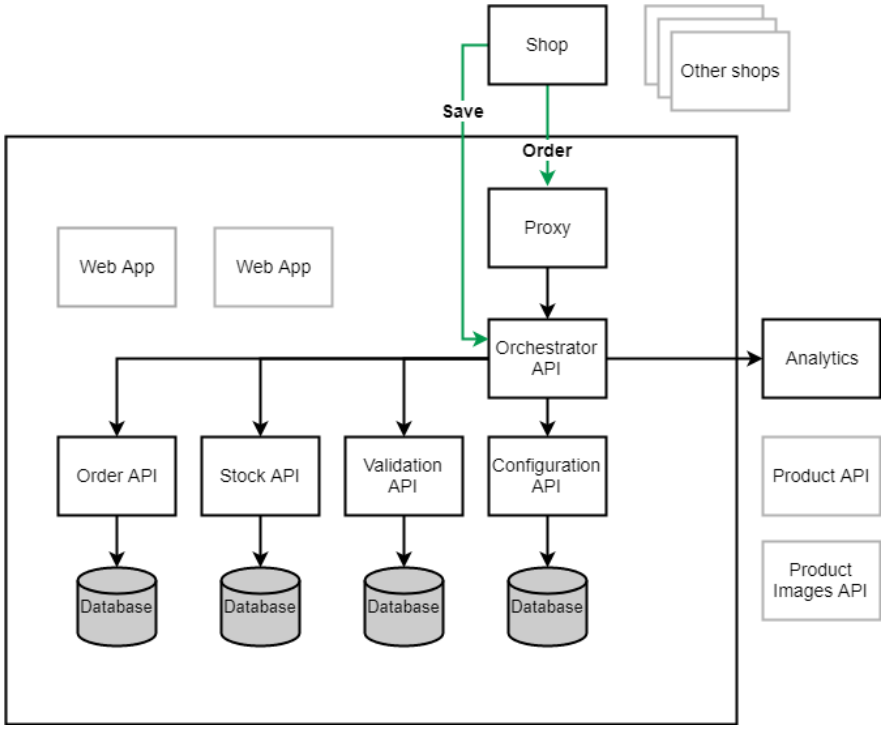

Fig. 3. High level architecture and integrations.

All databases of the system are MySQL[1] databases and the APIs are Express[2] applications written in JavaScript. The API endpoints mostly consist of CRUD operations, the Orchestrator API being an exception. Orchestrator API does not have direct access to the databases, and its purpose is to orchestrate the actions needed to complete the save and the order operations. Those actions consist of validation and calling the other APIs to complete the request.

The system was built to replace an old monolithic system. The transition happened gradually, one endpoint and one functionality at a time. The Proxy was implemented to help in the transition period. It used to contain logic to decide whether to forward the incoming calls to the old or the new system depending on the state of the transition. Currently, the Proxy is only used to keep the old deprecated API endpoints supported until all the consumers have been updated to use the new endpoints. Order requests from the Shop are still going through the Proxy meaning that those have not yet been integrated to use the new endpoints provided by the Orchestrator API.

The scope of the case study consists of the two endpoints: Save and Order. Both are highlighted in Figure 3 with green color. These endpoints were cho-

[1] `https://www.mysql.com/`
[2] `https://expressjs.com/`

sen for the case study because they involve multiple microservices to fulfill the incoming requests. That makes them an exciting target for a spike from the integrations point of view.

3.2 Baseline Test Setup

The system of the case study has been tested with unit-, component- and end-to-end tests. The unit tests are used to test single functions within the microservices. Most commonly, they have been written to help the developers to implement more complex logic to verify that the small piece of code works as intended. They do lift off some burden from the other testing methods, but they do not test any parts of the code that is directly involved in integrations. Because of that, they are not discussed further in the scope of this case study, and the focus will be in the component- and end-to-end tests. The component tests present the majority of the tests in the system. They have been implemented for every single endpoint in every single service, and they extensively test the behavior of them. The tested behavior includes different happy case scenarios, request validation errors, and situations where the services in the upstream or the databases are not functioning correctly.

The team ended up implementing a vast number of component tests because these were comfortable and fast to implement. A single component test involves sending a request to the endpoint and then checking if the endpoint returns the expected response. All the outside integrations are always replaced with mocks. The mocks help to verify if the service calls external services correctly, and helps to emulate different scenarios where the external services behave in different ways. Most importantly, they make the tests isolated and easy to operate.

The team was quite confident in the testing strategy with just the unit- and component tests for quite a while. Together the tests were very throughout at making sure the isolated services worked as expected. The team also had a tactic to avoid making changes to the endpoints that could potentially break integrations. That meant only adding new features instead of changing or removing the existing ones. The confidence slowly faded away when the number of services, endpoints, and integrations kept growing when new features were implemented. As a result of that, the team decided to implement end-to-end tests to cover the most critical functionalities of the system.

The experience from the end-to-end tests was entirely different compared to the component tests. They required much additional effort because the tests were not isolated anymore, and they involved multiple different services and databases. To implement the tests, the developers needed to start all the associated services and databases on their machines. In addition, the end-to-end tests required planning. The whole system had to be in a specific state to make the tests pass. That meant inserting a correct set of data to all the databases – the setting and resetting of the data needed to happen before the tests were run. The team ended up implementing a couple of additional endpoints to the services which were only used by the tests. They also created a couple of seed SQL files that could be used to insert data to the databases manually.

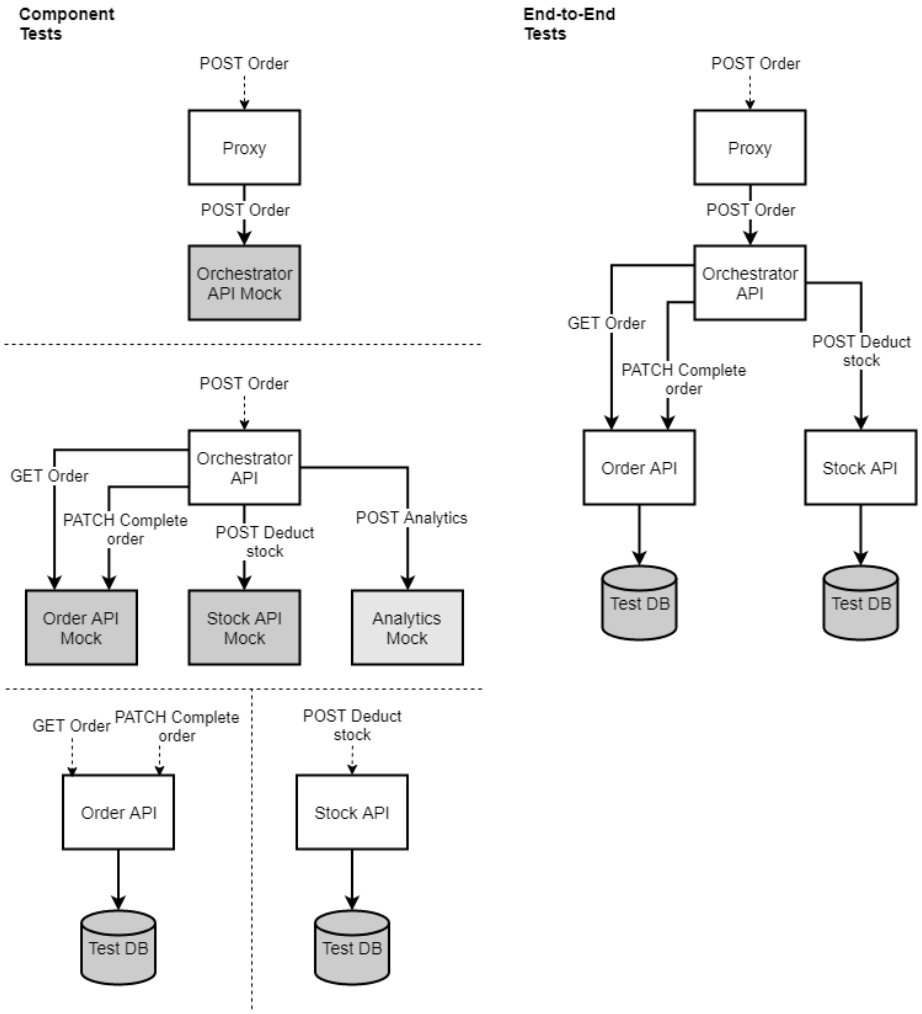

Fig. 4. Test boundaries of the order endpoint.

The end-to-end tests needed to be run in the Continuous Integration (CI) system as well. The team already had an existing development environment that was used by the CI. The development environment had all the services deployed and available for testing purposes. The same errors which happened during developing on local machines often followed to the tests run by the CI in the development environment. Those were mainly caused by other development activities and other tests modifying the data. Often the fix required manual work to rerun the seed SQL files when something in the system had changed. The debugging of the errors became more challenging as the system grew, and the errors had to be traced from logs collected from multiple services.

The general feeling of the end-to-end tests was that they were slow, prone to errors, hard to debug, and non-deterministic in general. Sometimes it even felt like the team avoided running them because of the high effort. It was not uncommon that they failed to data errors, and no one wanted to spend time debugging them. In some cases, the errors in tests were left completely ignored if it was evident that there were no new changes to the system that could have broken the feature. It did not feel rewarding to debug and fix errors that were only related to the testing environment and not to the actual features of the system.

The above experiences and feelings guided the team to avoid implementing the end-to-end tests for every feature. It felt like the growing number of end-to-end tests would shift time and focus from other important things to just debugging false-negative errors. Because of that, they were implemented for just a handful of the most critical features of the system. The end-to-end tests only tested one happy case scenario per feature and did not even try to test all the different error scenarios.

Figure 4 introduces the testing boundaries for Order endpoint. The feature involves four microservices and two databases in total. The component tests isolate the microservices from each other using mocks and the end-to-end test tests if the Order feature works when all of the services are connected.

3.3 Consumer-Driven Contract Tests

Consumer-driven contract tests were implemented using the Pact JS[3], and they used Pact Broker[4] for sharing the contracts. Both the Pact Broker and the tests were run on a local machine. They were not attempted to run in the CI environment.

Calling the Save and Order endpoints initiates a set of interactions between the services. Following the naming convention of consumer-driven contract testing, each interaction happens between a consumer and a provider. Sometimes service can have both of the roles if it needs to consume other services to be able to respond to its consumer. The different roles for the services in this case study are broken down for both endpoints in Tables 1 and 2.

Service	**C**	**P**
Orchestrator API	x	
Configuration API		x
Validation API		x
Stock API		x
Order API		x

Table 1. Save endpoint roles. C and P refer to Consumer and Producer, respectively.

Service	**C**	**P**
Proxy	x	
Orchestrator API	x	x
Stock API		x
Order API		x

Table 2. Order endpoint roles in integrations. C and P refer to Consumer and Producer, respectively.

[3] `https://github.com/pact-foundation/pact-js`
[4] `https://github.com/pact-foundation/pact_broker`

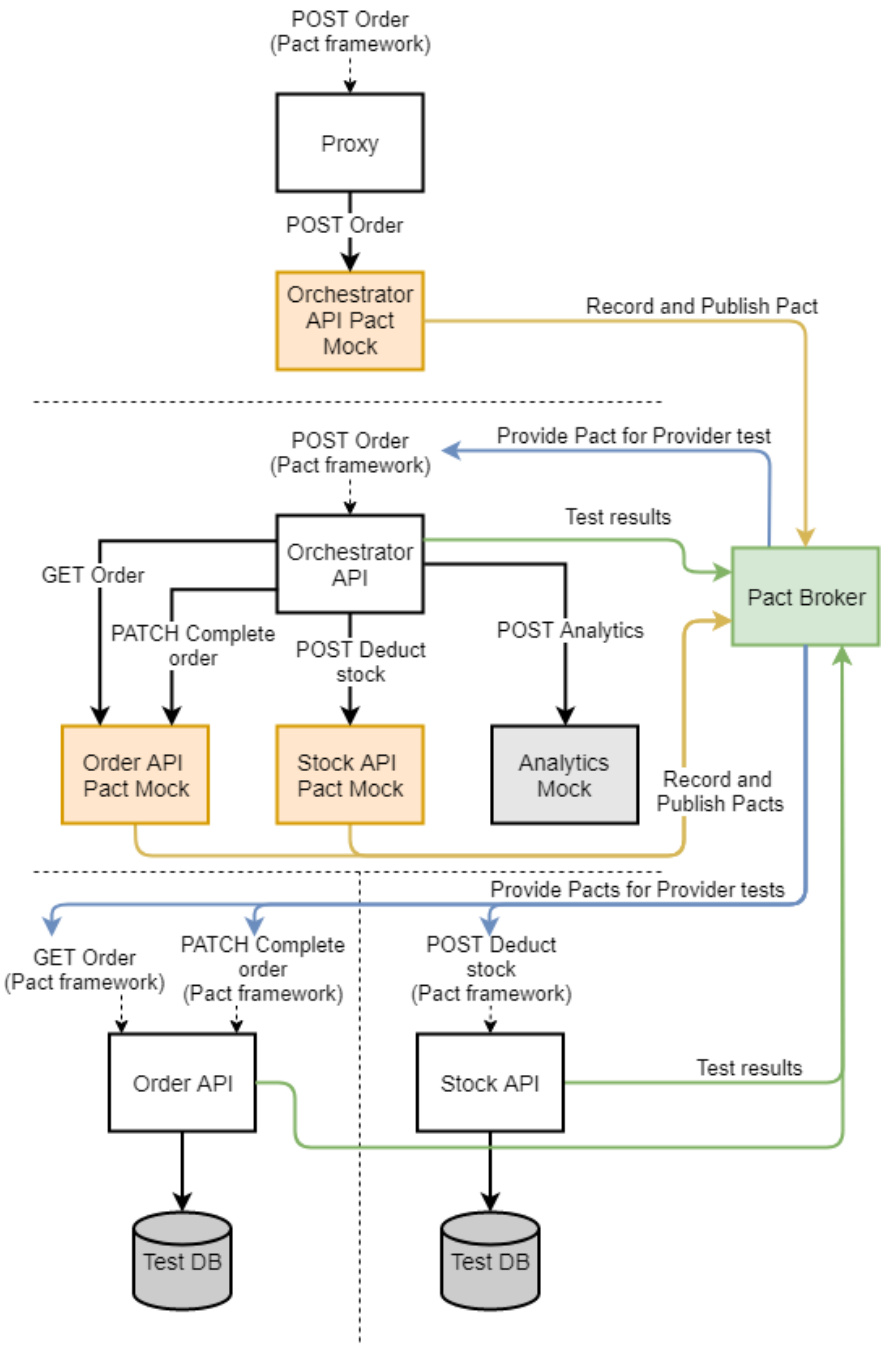

Fig. 5. Consumer-driven contract tests of the Order endpoint.

Consumer-driven contract tests break the testing boundaries between the services when compared to the component tests. This means that the services are no longer fully isolated from each other. The services are not directly connected either, like happened with end-to-end tests or would happen with integration tests. Instead, they are indirectly connected, and they communicate with each other using the contracts as a tool.

Such setting lets the consumer and the provider sides to be tested separately. There is no requirement for them both to be available and connected during the test execution. That still does not lift off the requirement of having to run the tests on both sides to fully verify the integration. Figure 5 illustrates this for Order endpoint. When it is compared to Figure 4 with the component- and end-to-end tests, it is quick to notice that a few differences are standing out between the approaches.

The consumer tests use the mocks to achieve isolation in the same manner as component tests, but in addition to that, the Pact Broker is being utilized to share the contracts to the provider tests to fully verify both sides of the integrations. Unlike component tests, the consumer-driven contract tests do not test the behavior of the consumers. They directly trigger the parts of the code that initiate the external calls to the provider to focus solely on the integrations.

The implementation of the consumer and the provider tests with Pact differed from each other quite a lot. On the consumer side, the tests were all about implementing the Pact mocks. That meant writing a mock with the expected request and response for it. When the tests were executed, Pact compared the actual requests generated by the consumer application to the ones specified in the mock. If they matched, the tests passed, and Pact generated a contract out of the mock and considered the consumer side of the integration verified. At that point, the new contract was automatically uploaded to Pact Broker.

The provider side of tests required much less work compared to the consumer side. They ended up requiring only tens of lines of code. The implementation of the tests consisted of making sure the provider is available, and there is a correct set of data in its database. The provider tests automatically fetched all the contracts from Pact Broker. Then Pact dynamically generated tests out of the contracts and ran them against the provider. If the provider responded with the same responses the consumer specified to the contract, the tests passed. The test results were automatically reported to the Pact Broker after each test run.

Each tested integration between the services of both the Save and the Order endpoints consisted of 2-3 interactions. In total, there were 23 different interactions, and they all had a meaning to the consumers. Every single integration between the services contained the 200 OK happy case interaction. The second most common interaction was the 400 Error, which was a result of a failed request validation on the provider side. All of those interactions were important for the consumers because they had implemented behavior based on them. If the format changes, the consumers fail to handle those scenarios properly.

4 Results

4.1 Comparison of Testing Methods

The comparison of the testing methods was made by seeding defects to the integrations and studying how the tests caught them. The defects were implemented by going through all the interactions one at a time and by separately implementing them to both, to the consumer and to the provider, sides. The goal was to find out how the testing methods can catch those defects that break the integrations.

The seeded defects were violations against the contracts that were already verified on both sides. On the consumer side that meant changing the request that is sent to the provider. A few concrete examples of that would be renaming of query parameters, changing the format of request body or modifying the request headers. On the provider side the violations were changes to the API and its responses.

The comparison of the testing methods revealed that the consumer-driven contract tests were able to catch every single defect from the 23 different interactions. That was not the case with the component tests; they allowed the defects to slip through. The end-to-end tests were able to catch the defects from interactions that were part of a single happy case scenario, but the rest of the defects were left uncaught.

The comparison of the testing methods highlighted that the initial testing strategy was lacking when it came down to testing integrations. It also showed that consumer-driven contract tests were able to fill that hole from the testing strategy. The component tests from the initial testing strategy were very brittle in revealing errors in integrations. They did manage to reveal if something had changed during implementation time, but they did very little to tell if the change was an actual breaking change to a specific integration or just a change to the behavior of the service. Because of that, there is a chance to accidentally or unconsciously change the component tests to match the new functionality without realizing the implications on the other side of the integration. That leaves the integration broken while the component tests are still passing.

The end-to-end tests, on the other hand, proved to catch the breaking changes in the happy case of the Order endpoint. That is great, but there were still many different interactions that were left entirely untested. One good example is the Order endpoint, which had just one test for the happy case scenario and the different error case interactions were untested.

In conclusion, the introduction of the consumer-driven contract tests brought confidence to the testing strategy. The consumer-driven contract tests turned out to be very throughout at testing the integrations between the services. Compared to the component tests, it was not possible to make the tests pass without fixing the broken integrations first. They also caught all the different error cases which were not covered by the end-to-end tests. They were also easy and fast to run as they are always run in isolation from the other services. They did not cause any

false negative errors, which proved they were very deterministic, making them convenient to use.

Integrations between the shop, the case study system, and analytics were not testable with the consumer-driven contract tests. They could and most likely should be tested that way, but they were scoped out from the case study to keep the scope more tightly on the testing method itself. Because of that, there were no attempts to contact the other teams to implement the consumer or the provider tests and to share the contracts. Therefore, the integrations with them were left tested with component tests and mocks that are not being verified in any way. That did not change the initial situation any better or worse as that was the case even before the implementation of consumer-driven contract tests.

4.2 Experiences with Consumer-Driven Contract Testing

The literature suggested that consumer-driven contract testing is a viable option to test integrations. There was just one requirement that could be hard to fulfill in some situations. The requirement was that the consumer and the provider must be able to communicate the process with each other.

The experiences from the case study supported the findings from the literature. The consumer-driven contract tests were very throughout on finding defects from integrations between services. The defects got reliably caught from both of sides of the integrations.

In addition to being a viable option for testing integrations, the literature listed further benefits for the consumer-driven contract testing, including (i) decoupling consumer from the provider and enables testing of both sides in isolation; (ii) fast, stable, and deterministic execution; (iii) ensuring that the provider knows who are consuming its API and how; (iv) enabling the provider to evolve based on real business needs from its consumers; (v) enforcing that the provider tests always catch sudden breaking changes to the API; and (vi) using contracts as a tool to improve communication between teams.

With the case study, we were able to confirm most of the listed benefits. The consumer-driven contract tests were run in isolation, which resulted in them being fast and stable. Due to that, they were also relatively easy to operate compared to the end-to-end tests even though they required the extra step to share the contracts. The sharing was made simple with Pact and the Pact Broker, which both proved out to be prominent tools in the field of implementing consumer-driven contract tests.

Especially the tooling made it possible to visualize who are the consumers for the providers and how they are consuming the APIs. In the case study, the consumer-driven contract tests were implemented after the actual services had already been implemented. That did not let the case study to examine how the provider could have been created and evolved from a scratch based on the needs of the consumers. Still, the case study was able to prove that it is possible to evolve the provider when the requirements from the consumers are visible in the contracts. In the future, consumers can use the contracts to communicate or suggest new changes to the provider.

The approach in the case study was to implement the consumer-driven contract tests in a spike by experimenting and implementing the tests for only a selected few features of the system. That was successful, and it proved that the spikes are an excellent way to experiment with the consumer-driven contract testing for already existing systems. That is an essential feature as it enables teams to experiment with the tests with a smaller scope and see if it fits their purposes. Majority of the time was spent at the beginning on learning the new way of testing and finding out proper tools for the job. After those were sorted out, the implementation of new tests became straightforward.

The spike method had another significant benefit when figuring out if the consumer-driven contract testing should be used. It can work as an excellent way to learn if there are any pain points in the communication inside the organization or between the different teams before fully committing to it. The effect of communication is a good thing to keep in mind when thinking about using the testing method. The system in the case study was initially implemented by one team so the communication would not have been a problem. It could have been challenging to extend the method to the outside consumers (the Shops) who were consuming the system or to the other APIs (the Analytics) that were consumed by the system.

The expansion of the testing coverage outside of the system developed by the team in the case study would have required communication with other teams. The first step would have been to introduce the testing method to them and then convince them that consumer-driven contract testing is something that is needed. That can be hard for many reasons. The other team can, for instance, be busy doing something else, with its own prioritized backlog. Moreover, even if the new testing method would end up to the backlog, it could take a while to get it prioritized high enough for actual implementation. Still, contracts are a great tool to communicate and share the details if all of the parties finally agree to proceed with the implementation. In another scenario, the teams could break the silos between them and cooperate so that the outside team does the initial implementation of the tests to help the other team to get started.

5 Discussion

In this paper, we have used five different testing methods split into two categories: isolated and integrated testing. Ideally, the highest number of tests should be written to the isolated category as they are easier to implement, more stable, and faster to execute, resembling a pyramid in shape (Figure 1 given in Section 2). Every testing method in the pyramid has a different purpose and should focus solely on it to get the best results out of the combination of them all. Together they were said to be an ideal testing strategy for microservices [2].

It was shown that the initial testing strategy in the case study was lacking. It only included end-to-end, component, and unit tests. Compared to the testing pyramid, it was completely missing the consumer-driven contract- and

integration test layers. Because of that, the errors in integrations were not caught adequately by the tests.

The component tests did give a clue if something possibly had changed in the integrations during the development time, but they did not directly reveal if the changes were breaking changes to the integrations. The change could also be related to some simple behavior such as validation rule that does not break the integration but makes the tests fail. When a new breaking change was implemented, the component tests initially failed. After that, the test could be changed without a notice that the other side of the integration is no longer compatible with the new change.

Unlike the component tests, the end-to-end tests did reveal broken integrations and prevented them from being deployed. The problem with end-to-end tests was a massive effort required to implement and operate them. These need much planning to implement, and these were prone to errors related to the testing environment, network, and test data. Debugging the reason for the test failures was troublesome and time-consuming because multiple services were involved in the process. Due to that, they had been implemented just for a couple of happy case interactions, and they lacked the coverage for different error case interactions. It would have been next to impossible to cover all the different interactions with the end-to-end tests, as there are so many different corner cases and branches to consider.

The integration tests would test the integrations using real running instances of the components in a production-like environment. These would be slower and harder to implement and execute compared to the consumer-driven contract tests. The added value would be mostly related to testing if the network and other infrastructure are working as expected. In the end, this would be testing a production-like environment but not the actual production environment. That would not guarantee that the production environment works the same way as the test environment. However, the experiences gained from the case study showed that the consumer-driven contract tests could replace the integration tests.

An open question related to integration tests is if these could help with testing the integrations with the systems that were developed by the other teams (Other shops and Analytics in Figure 3). That is an interesting issue, as consumer-driven contract tests would have required the parties to communicate with each other to make the tests happen. In both cases, we considered the other side of the integration unreachable to reason about the impact on the testing method.

Integration tests addressing the integration with the shops would require the shops to implement the integration tests as these are the consumer. The shops were considered unreachable, so that is out of the question. Even if the shops implemented the integration tests, the testing would remain challenging. Each shop should run the tests every time there are changes to the provider. Ideally, the developer would be able to run the tests while developing the changes on a local machine. That would require having the codebase for all the shops and be able to run their tests, which further does not sound ideal and would require much unnecessary effort.

The integration with analytics was a case where the provider was considered unreachable. From the perspective of integration tests, this does not matter as long as the provider stays available for the tests to call it. Still, the integration tests would be far from ideal in this case as well. The integration tests would only be able to prove that the integration worked when the tests were run. They would not prevent the provider from changing the interface unless the provider is able to run the integration tests of the consumer. This means that the breaking change could happen in any given time, and if it happens, it could take a while until the consumer integration tests are rerun to catch the errors. With just integration tests, there is no way for the consumer to prevent the unreachable provider from publishing the breaking changes. The integration test would merely work as a tool to find out if the integration is already broken and requires fixing. That could and should be done more efficiently with a proper setup of logging, monitoring, and alerts.

To summarize, an ideal testing strategy for microservices based on the lessons learned from the case study as well as literature contains unit, component, consumer-driven contract, and end-to-end tests. The integration tests can be left out, and their responsibility should be given to consumer-driven contract tests. This leaves only end-to-end testing in the integrated testing category, while the rest of the testing methods are in the isolated testing category. An additional benefit that was gained and considered significant is reduced flakiness as a result of replacing the non-deterministic tests with deterministic tests. This enables more enhanced automation when considering debugging of the system. Finally, the ideal testing strategy is highly dependant on cooperation and communication between the teams. Therefore, good communication across the teams is something that should always be a top priority. It cannot be emphasized enough that communication is the foundation that enables the teams to build great things together.

Threats to Validity. The validity of as study is basically about the knowledge claims that can be made based on the results [10]. As our intent was to gain experiences on the usage of a particular testing methodology, one particular issue in terms of validity is that of the role of the testing methodology itself in the results achieved. The separation of the methodology used from the experience of the designers in the actions taken is fundamentally hard. This is something one may need to take into account if aiming to apply (generalise) the results in other cases. In addition, the characteristics of the system used in the case study may have an effect on the results. However, as these characteristics are somewhat typical in microservice based systems, this is not considered an overly restricting issue.

6 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) in-

tegrations 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.

Acknowledgments The work of N. Mäkitalo was supported by the Academy of Finland (project 313973).

References

1. Cerny, T., Donahoo, M.J., Trnka, M.: Contextual understanding of microservice architecture: current and future directions. ACM SIGAPP Applied Computing Review **17**(4), 29–45 (2018)
2. Clemson, T.: Testing Strategies in a Microservice Architecture. https://martinfowler.com/articles/microservice-testing (2014), [Accessed 8.2.2019]
3. Cohn, M.: Succeeding with Agile: Software Development Using Scrum. Addison-Wesley Professional, 1st edn. (2009)
4. Meyer, B.: Applying'design by contract'. Computer **25**(10), 40–51 (1992)
5. Namiot, D., Sneps-Sneppe, M.: On micro-services architecture. International Journal of Open Information Technologies **2**(9), 24–27 (2014)
6. Newman, S.: Building microservices: designing fine-grained systems. " O'Reilly Media, Inc." (2015)
7. Pautasso, C., Zimmermann, O., Amundsen, M., Lewis, J., Josuttis, N.: Microservices in practice, part 1: Reality check and service design. IEEE Software **34**(1), 91–98 (2017)
8. Pautasso, C., Zimmermann, O., Amundsen, M., Lewis, J., Josuttis, N.: Microservices in practice, part 2: Service integration and sustainability. IEEE Software (2), 97–104 (2017)
9. Robinson, I.: Consumer-Driven Contracts: A Service Evolution Pattern. https://martinfowler.com/articles/consumerDrivenContracts.html (2018), [Accessed 21.10.2018]
10. Shadish, W.R., Thomas, C.D., Thomas, C.D.: Experimental and quasi-experimental designs for generalized causal inference. Houghton Mifflin Company (2002)
11. Sill, A.: The design and architecture of microservices. IEEE Cloud Computing **3**(5), 76–80 (2016)
12. Wolff, E.: Microservices: flexible software architecture. Addison-Wesley Professional (2016)