

Master's Programme in Computer, Communication and Information Sciences

Consumer-Driven Contract Testing for Microservices

Practical Evaluation in A Distributed Organization

Tuomas Maanonen

Copyright ©2024 Tuomas Maanonen

Author Tuomas Maanonen

Title of thesis Consumer-Driven Contract Testing for Microservices: Practical Evaluation in A Distributed Organization

Programme Master's Programme in Computer, Communication and Information Sciences

Major Software and Service Engineering

Thesis supervisor Dr Jari Vanhanen

Thesis advisor(s) MSc Boriss Nazarovs

Date 12.02.2024

Number of pages 81

Language English

Abstract

Software development in a distributed organization using a microservices architecture pattern presents unique challenges. System complexity starts moving from inside services to interactions between services. Communication inside teams is organized smoothly using regular daily and weekly meetings, but cross-team communication remains ad hoc. Teams residing in different time zones may find it difficult to find time for meetings, and dependencies between teams slow down development. All of this affects the quality of the system.

Using the action research method, this thesis explores the primary quality assurance challenges that arise during software development in the context of a large, distributed case company. It also implements and evaluates consumer-driven contract testing as a potential solution for these challenges. Consumer-driven contract testing is a testing technique that enables isolated service testing by utilizing the concept of a consumer service contract. During the research process data was gathered for problem diagnosis and evaluation through review of internal documents, literature review and interviews.

We found that distributed software development comes with unique challenges for quality assurance. Appropriate service dimensioning must be achieved to enable working service ownership for teams. Development teams must also be trained in operations and service infrastructure to assume full service ownership. Lastly, dependencies between services arise as a new source of complexity that also creates team coupling. Consumer-driven contract testing is a promising technique for managing complexity in service-to-service interactions. We found that the technique delivers a stable, fast, and low-maintenance isolated testing method. However, the technique has a high barrier of entry due to the number of processes and tools that need to be effectively implemented and utilized. The learning curve also proved to be steep even for developers experienced in automated testing. These factors deter and slow down adoption of the technique.

Keywords Consumer-driven contract testing, software testing, microservices, quality assurance

Tekijä Tuomas Maanonen

Työn nimi Kuluttajasopimusvetoinen testaus mikropalveluille: Käytännön arviointi hajautetussa organisaatiossa

Koulutusohjelma Master's Programme in Computer, Communication and Information Sciences

Pääaine Software and Service Engineering

Vastuopettaja/valvoja TKT Jari Vanhanen

Työn ohjaaja(t) DI Boriss Nazarovs

Päivämäärä 12.02.2024 **Sivumäärä** 81

Kieli Englanti

Tiivistelmä

Ohjelmistokehittäminen hajautetussa organisaatiossa käyttäen mikropalvelu arkkitehtuuria tuo mukanaan omanlaatuisia haasteita. Järjestelmän monimutkaisuus siirtyy palveluiden sisältä niiden väliseen viestintään. Kommunikaatio tiimien sisällä toimii sulavasti päivittäisten tapaamisten avulla, mutta tiimien välinen kommunikaatio on satunnaista. Lisäksi tiimit, jotka toimivat eri aikavyöhykkeiltä pitävät tapaamisten järjestämistä entistä haastavampana. Tämä kaikki hidastaa soveluskehitystä ja vaikuttaa järjestelmän laatuun.

Käyttäen toimintatutkimus menetelmää, tämä diplomityö selvitti hajautetun soveluskehityksen suurimpia haasteita. Tutkimus tehtiin suuressa organisaatiossa, jossa toteutetaan soveluskehitystä jaettuna useisiin tiimeihin ja käyttäen mikropalvelut arkkitehtuuria. Diplomityössä myös toteutetaan kuluttajasopimusvetoinen testaus mahdollisena ratkaisuna löydetyille haasteille. Kuluttajasopimusvetoinen testaus on testausmenetelmä, joka tarjoaa eristettyä palveluntestausta käyttäen hyödykseen kuluttajasopimuksen käsitettä. Tutkimusprosessin aikana kerättiin tietoa organisaatiosta haasteiden selvittämiseen ja testausmenetelmän arviointiin sisäisten dokumenttien, kirjallisuuskatsauksen ja haastatteluiden avulla.

Löysimme erilaisia haasteita, jotka liittyvät hajautettuun ohjelmistokehitykseen mikropalveluilla. Järjestelmän palvelut täytyy mitoittaa oikein, jotta itsenäiset tiimit pystyvät omistamaan omat palvelunsa ongelmitta. Tiimit täytyy kouluttaa ohjelmistokehitykseen ja ylläpitoon, jotta he pystyvät käsittelemään kaikki palveluun liittyvät ongelmat. Lisäksi palveluiden väliset yhteydet osoittautuvat ongelmalliseksi luomalla uutta monimutkaisuutta ja lisäämällä tiimien välisiä riippuvuuksia. Toteamme että kuluttajasopimusvetoinen testaus on lupaava testausmenetelmä viimeisen haasteen vähentämiseksi. Testausmenetelmä on vakaa, nopea ja vaatii vähän ylläpitoa. Kuitenkin menetelmän käyttöönoton kustannukset ovat korkeat ja se on vaikea oppia. Nämä ominaisuudet hidastavat menetelmän käyttöönottoa ja siitä saavutettua hyötyä.

Avainsanat Kuluttajasopimusvetoinen testaus, ohjelmistotestaus, mikropalvelut, laadunhallinta

Table of contents

1	Introduction	6
1.1	Motivation	6
1.2	Research questions.....	7
1.3	Research scope	9
1.4	Outline of the thesis	9
2	Background and related work	10
2.1	Software Quality.....	10
2.2	Microservices architecture.....	14
2.3	HTTP RESTful.....	17
2.4	Testing in microservices architecture.....	18
2.5	Relevant case studies	26
3	Research methodology	30
3.1	Research process	31
3.2	Case overview	34
4	Results	47
4.1	Quality assurance challenges	47
4.2	Identified gaps in the current process	49
4.3	Evaluation of quality assurance strategies	54
4.4	Implementation of consumer-driven contract testing	56
4.5	Evaluation of the implementation	70
5	Conclusions	74
	References.....	78

1 Introduction

Microservices architecture has become a common way to structure scalable, large-scale systems. Organizations applying microservices architecture are often structured as feature teams with microservice ownership split between teams. Distributed software development using microservices presents unique challenges for quality assurance. Complexity in the system shifts from within the service boundary to the integrations between services (André, 2018). Close communication exists inside the teams, supported by daily and weekly meetings. However, organizing communication between teams is often more challenging. When teams act independently and aim for fast iteration cycles, accidents can happen, and knowledge silos can form where knowledge about parts of a system does not transfer between teams. Solving these challenges requires a different approach than with single-team, monolith architecture feature development.

1.1 Motivation

This thesis has two primary objectives. First, it identifies specific quality assurance challenges in organizations doing distributed software development using a microservices architecture. Second, it implements and evaluates a specific technique for addressing one of the identified challenges. Specifically, we have focused on verifying the validity of service-to-service interaction and ensuring the application programming interface (API) stability. This is a relevant challenge for distributed microservices software development as distributed team structure introduces delays in team-to-team communication, and services often form the ownership boundaries of feature teams. Insufficient communication can make service-to-service calls a hotspot for faults and quality issues. As mentioned, microservices are thought to shift the complexity from inside the service to the interactions between services (André, 2018). Team code ownership boundaries often form around services, making their interactions also interactions between teams.

In prior research, consumer-driven contract testing has been effective for verifying service integrations (Lehvä, Mäkitalo and Mikkonen, 2019). Specifically, this research verified contract testing with the Pact framework. Traditional, more widely adopted integration testing methods could be used to cover the same aspects of the system. However, there are numerous warnings in the literature against relying on integrated testing, such as end-to-end tests (Wacker, 2015; Lehvä, Mäkitalo and Mikkonen, 2019; Newman, 2021). In this thesis, we will implement consumer-driven contract testing using Pact in a different context to further test its effectiveness. The implementation will be done with one case company (described in section 3.2). While the previous

case study emphasized the difficulties communicating between teams, the Pact framework was only used in one team. In this thesis, we have implemented the Pact framework with two teams cooperating across service ownership boundaries, creating the implementation scenario we expect to be the most beneficial. Further, the previous case study verified that contract testing with Pact efficiently catches faults using fault injection. In this thesis, we focus on real-world performance. The implementation is done in a context where end-to-end testing was already used. However, the same problems identified in prior literature motivated a search for more isolated testing methodologies.

1.2 Research questions

The overall research problem and motivation were introduced in the previous section. Here, we will state the final research questions more formally and precisely. Three related but separate research questions form the thesis. We used action research as our primary research methodology. This method started with a diagnosis phase before the final research questions were fully formulated. An initial research question was used as input into this first phase of the study:

- **RQ1:** What are the primary quality assurance challenges in distributed software development with microservices architecture?

Quality assurance and microservices architecture are further defined in the background section of the thesis. Distributed software development refers to an organizational structure where development is split across semi-independent teams, sometimes in different geographical areas and time zones.

With RQ1, we aimed to survey the big picture of the case company and form the context for the actions taken in the study. The diagnosis stage also validates that we take an action that relates to an observable challenge in the real world. After the initial phase, the research was scoped to analyze service compatibility-related issues. Further, more actionable questions were formulated:

- **RQ2:** What kind of process or tooling reduces service-to-service interaction faults?
- **RQ3:** How is the developer experience impacted by the processes or tools for reducing service-to-service interaction faults?

In these research questions, service-to-service interaction refers to any communication or information transfer between services using some protocol or method such as Hypertext Transfer Protocol (HTTP), a variant of Remote Procedure Call (RPC), or message transfer using a message queue such as

Kafka. In practice, the thesis focuses on HTTP interactions. Developer experience and developer velocity are defined by the metrics and questions in Table 1 and Table 2 in a measurable manner. The developers participating in the study are the primary data source when evaluating these concepts.

Quantitative data is collected from systems at the case company. The measures/metrics used are defined in Table 1. These measures are thought to relate to the effectiveness of the service verification strategy and factors affecting developer velocity, such as continuous integration (CI) pipeline reliability. We do not touch significantly on the definition of continuous integration. In this context, a continuous integration pipeline is any automated process for building and verifying software applied to every new version.

Table 1. Metrics used for quantitative evaluation of RQ2 and RQ3.

Research question	Metric	Data source
RQ2	Number of incidents in the participating services	Incident tracking tool
RQ3	CI pipeline build times	CI pipeline logs
RQ3	CI pipeline test failures	CI pipeline logs

Further, understanding developer experience requires qualitative data collection. Questions that form the basis for developer experience in this thesis are defined in Table 2. These questions are used to help with the evaluation interviews.

Table 2. Questions used to guide qualitative evaluation of RQ2 and RQ3.

Research question	Question topic
RQ2	Have there been incidents and bugs related to service-to-service interaction?
RQ3	Has contract testing with Pact been easy to implement and learn?
RQ3	Did the contract testing increase developer deployment confidence?
RQ3	Do developers feel like the changes slowed down development?
RQ3	What issues have developers encountered after the changes?

Research question	Question topic
RQ3	How much time do developers spend maintaining the changes?

1.3 Research scope

While this thesis discusses quality assurance, it mainly focuses on verification and sources of software faults causing behavior that differs from the specification. In this context, verification is an active process for ensuring that software deliverables fulfill the specified requirements. We define quality as the extent to which a service fulfills its requirements. For requirement we use a simplistic definition and only consider specified requirements. In practice, many quality issues in software systems are caused by deficiencies in requirements engineering. Insufficient or misinterpreted requirements specifications can cause significant issues for any product or service trying to satisfy user needs. However, this thesis does not focus on the requirements elicitation or specification phase of software engineering. Instead, we focus on software development and verification, assuming an existing and accurate understanding of what should be built exists.

1.4 Outline of the thesis

The remainder of the thesis starts with *Chapter 2*, discussing the background and related work. First, we set the context by defining software quality and placing software testing into its place in the overall software quality management framework. Then, we discuss other relevant concepts, such as microservices architecture and HTTP RESTful. Lastly, we discuss testing in microservices architecture and explain the concept of consumer-driven contract testing and how it relates to other testing methodologies. *Chapter 3* introduces the research methodology, process, and case company. *Chapter 4* explains and discusses the study results, including the challenges discovered, what solution options were considered, how contract testing was implemented in practice, and the evaluation of the result. *Chapter 5* finishes with a discussion and conclusions, summarizing the answer to each research question.

2 Background and related work

This chapter defines the necessary concepts to understand the rest of the thesis. It also provides an overview of the related work used during the thesis. First, we will set the scene by discussing software quality. Then, we will discuss microservices architecture and HTTP RESTful to define the system context. We will spend most of this section discussing testing methodologies. Lastly, we will introduce existing case studies related to the thesis topic.

2.1 Software Quality

Software quality is ensured by following appropriate processes when developing software. The definition of *quality* relates to requirements and attributes (Schulmeyer, 2008). Schulmeyer collects two definitions for requirement. Firstly, a *requirement* can be imposed through a contract, standard, or specification document. Secondly, product requirements include the aspects necessary for a user to solve a problem using the product, even if not specified in a document. From now on, we will call the latter requirements *user needs*. In this model, the *user* can be either the customer or the end user. Schulmeyer also gives the definition for attribute, but this definition is less precise. In ISO 15939, attributes include all distinguishable characteristics. The key to this definition is what distinguishable means. Here, we assume that any attribute must be measurable, either automatically using quantitative methods or manually by a human observer. Quality can be applied to any object or entity, including products, services, and processes. Our understanding of the relationship between the concepts is visualized in Figure 1.

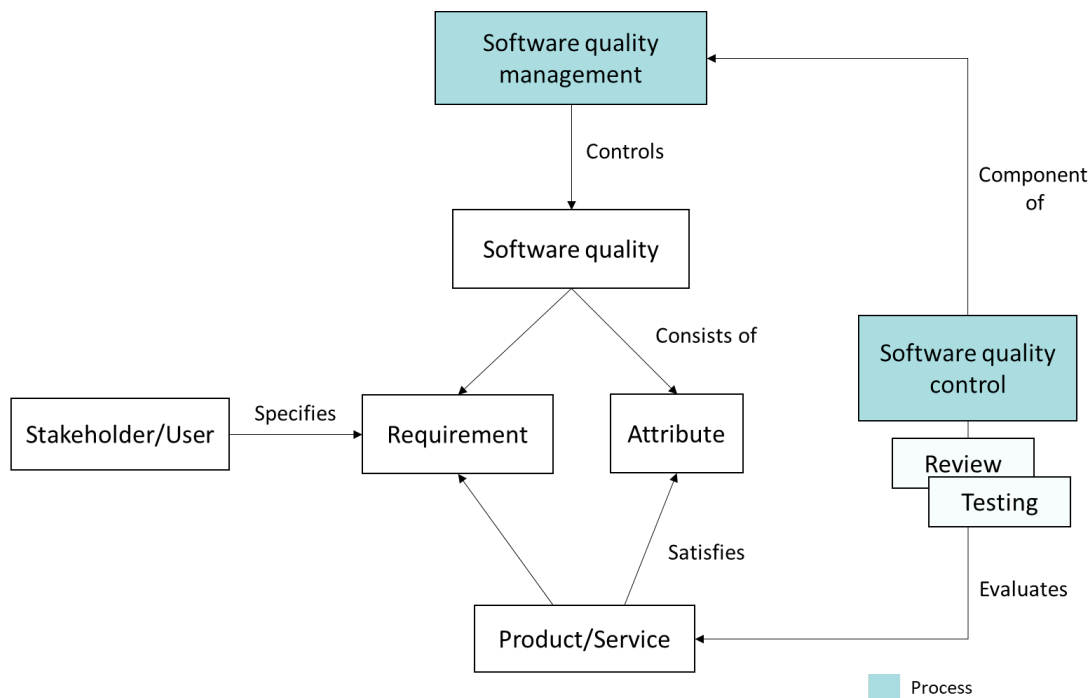


Figure 1. Relationship between the main processes and quality concepts.

Returning to the processes necessary to ensure software quality, we find software quality management (Tekinerdogan *et al.*, 2016). *Software quality management* (SQM) includes all processes for ensuring that software meets the organization's software quality objectives and achieves stakeholder satisfaction. Previously, in our requirement definition, we focused on users and documents. In practice, all stakeholders have quality requirements that can be accounted for through quality attributes. The SQM comprises four main processes: assurance, control, planning, and process improvement. The full names of the processes are shown in Figure 2. One dimension that differentiates these processes is scope. *Software quality assurance* (SQA) is an organizational activity that defines the “quality guide” that can be applied to any organizational project. All standard best practices and tools are defined in SQA. *Software quality planning* (SQP) applies SQA practices to specific projects. In this stage, the quality goals of the project are defined, and the SQA components are selected and customized to fit the specific use case. Critically, for the topic of this thesis, software quality control (SQC) includes activities for evaluating the quality of project artifacts. While the control practices are defined in SQA processes, code inspection, technical reviews, testing, and other evaluation techniques are executed as a part of SQC. Finally, *software process improvement* (SPI) includes projects that aim to improve processes to later achieve higher software quality across the organization. The work done in this thesis could be partially considered an SPI project, improving the SQA practices at the case company.

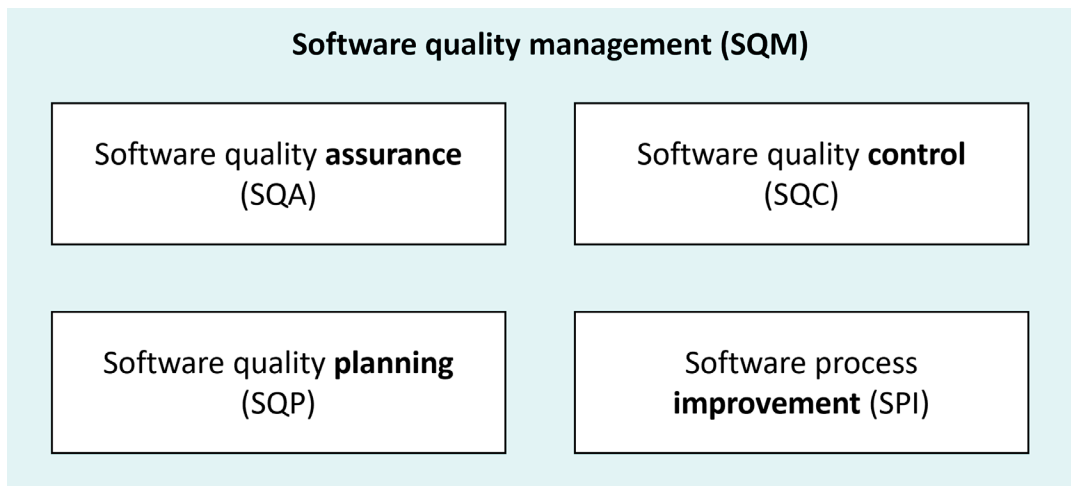


Figure 2. Processes of software quality management. Adapted from Tekinerdogan et al. (2016).

Software quality is not achieved through evaluation alone (Schulmeyer, 2008). However, evaluation is still an essential process in quality assurance. We also believe that evaluation activities contribute indirectly to quality by strengthening the organization's quality culture. For example, code reviews and software testing are as much quality conversation starters as they are methods used in software quality control. Recognizing the context of any methods or practices adopted is essential, and processes should also be evaluated for non-functional attributes.

2.1.1 Process attributes linked to quality

In practice, we needed to identify which parts of the development process at the case company could affect the number of defects and, through that, software quality. We looked at the literature to identify specific attributes that influence quality and ability to prevent faults in a software system. Three such attributes were found, although not all studies agree about their efficacy. This section is rather practical and introduces only the ideas considered during the thesis research.

The first attribute is the code coverage of a testing suite. Mathematical measures precisely define code coverage. There are many such measures, such as statement, branch, and statement frequency coverage (Aghamohammadi, Mirian-Hosseiniabadi and Jalali, 2021). Not all studies agree that there is a significant correlation between these code coverage measures and test suite effectiveness (Inozemtseva and Holmes, 2014). However, the statement frequency metric is relatively new and has not been tested in the referenced study. Simple statement and branch coverage metrics are more common in existing tooling. Despite these uncertainties regarding the effectiveness of

code coverage, other studies show that code coverage correlates with bug removal effectiveness (Kochhar, Thung and Lo, 2015). We believe this is sufficient to use code coverage as a clue for what could be improved in the testing process, especially considering the intuitive plausibility of the idea that higher code coverage should correspond to higher coverage of bug scenarios.

The second attribute is the quality of the code review. This quality can be split into two metrics: code review coverage and code review participation. It has been shown that these two metrics are potentially linked to software quality and the number of faults in a software system (McIntosh *et al.*, 2014). High code review coverage corresponds to most or all the changes being reviewed. High code review participation corresponds to deep reviewer involvement (e.g., detailed and extensive discussion) during the code review process. In a distributed context, the number of teams and participants has also been found to improve reviewer contributions (Dos Santos and Nunes, 2017). However, the extent of the code review comes at a penalty to duration, making it a process of balancing agility with risk. Importantly, it has been shown that the complexity of code changes contributes to the incidence of faults, making it an important aspect to consider during the review process (Hassan, 2009).

The third and last attribute is code ownership. While not entirely uncontroversial (Foucault, Falleri and Blanc, 2014), some studies show that code ownership does correlate with code quality (Bird *et al.*, 2011; Greiler, Herzig and Czerwonka, 2015). Code ownership, in this case, refers to the number of contributors to any single code artifact and how close they are to the given organization's structure. For example, a file with only one contributor writing 100% of the code has 100% code ownership. Meanwhile, having multiple contributors in the same team corresponds to more ownership than multiple contributors across teams. Code ownership could be hypothesized to relate to the extent of knowledge and understanding of the code.

While these three attributes are not uncontroversial, and they do not fully describe the attributes of a quality assurance process, they were selected for their appearance in literature, practicality, and relevance to the teams of the thesis study. Based on the diagnosis results introduced in section 4.1, all three attributes can be linked to statements made by the team members. For example, test coverage was mentioned as a factor affecting deployment confidence. Difficulties with reviewing changes by other teams were mentioned as a significant challenge, and issues related to the complexity of services and extent of knowledge with specific technologies (e.g., infrastructure) were mentioned as concerns when deploying new changes. These could be connected to the review process, complexity, and code ownership.

2.2 Microservices architecture

There are many ways of architecting software systems; microservices architecture is one of the more popular ways in this decade. Baresi and Garriga (2020) described microservices as somewhat of a buzzword and a successor to the now considerably less hyped service-oriented architecture (SOA). However, microservices architecture is popular and successful for distributed software systems. According to Newman (2021), microservices architecture evolved from real-world use and provides critical guidance that was missing from earlier SOA approaches. Microservices closely describes the software architecture used at the case company. This chapter will describe what we mean by microservices and microservices architecture. Knowledge of this architectural style will be used to discuss its benefits and downsides. This discussion will help us understand the core problems explored in this thesis.

Microservices architecture refers to a particular method of designing software as suites of independently deployable services (Baresi and Garriga, 2020). Microservices are, as the name implies, software services that are small or "micro" in scale. However, there is no clear definition of how small. According to Newman (2021), the size is difficult to pinpoint. The size should be kept to a level where the complexity is manageable. Both the number of services and the size of each service affect complexity. The size of a microservice should also be limited to allow new versions to be developed quickly as business needs change (Baresi and Garriga, 2020).

However, microservices architecture is more than just splitting a system into multiple small services. It would hardly differ from other distributed systems architectures if this were the case. Indeed, Baresi and Garriga (2020) argue that microservices aim to provide almost the same benefits as SOA, just with different methods and emphasis. They also state that, unlike SOA, microservices do not focus on reusability. Instead, microservices architecture aims to make systems independent in development, deployment, and maintenance. Essentially, they state, this is accomplished with the opposite of reuse: making very conservative decisions regarding what is shared with other services and sharing as little as possible. One consequence of this approach is that each microservice owns its state, and shared databases are not used (Newman, 2021). Sharing of any kind introduces dependencies between services, and dependencies provide opportunities for cascading failure. Baresi and Garriga point out that it should be possible to fully replace a microservice without touching other microservices, assuming it still fulfills its responsibility. The independence of microservices is also relevant when considering the processes and tools used for development. Although not essential, many enabling technologies could be considered a part of microservices architecture.

2.2.1 Benefits of microservices

Many benefits can be linked to microservices architecture. However, these benefits often revolve around common themes. Many software development organizations choose to organize their people and resources into small and autonomous teams. Teams at the case company are also often organized in this manner. According to Newman (2021), microservices can help align organizational structure and services.

The alignment between small, autonomous teams and microservices becomes clear from the words alone. There is a limit to the responsibilities a small team can bear while maintaining stability and high quality. These teams often have a single clear purpose on which they focus their development efforts. As such, the service they are developing should also focus on this purpose. Domain-driven design, where services are modeled around the business domain, can be used for this purpose (Newman, 2021). Autonomous teams explain the need for independence in service development, deployment, and maintenance. A single team is usually best equipped to deal with these areas. However, as the service and the team do not live in a vacuum, there must be a way to ensure the development work does not cause problems elsewhere. Dependencies in the systems would cause dependencies between teams and reduce autonomy. As microservices aim to minimize these dependencies, it fits this organizational structure well. Minimizing dependencies between services is a key consideration in this thesis.

Other benefits are related to optimizing how well each part of the system fits its purpose. Newman (2021) states that microservices allow different parts of the system to use different technologies that better fit its responsibilities. If we did not split different responsibilities across different systems, using technology in only one part of the system would be much harder without affecting other parts. This problem is especially apparent with programming languages that usually enforce using the same language across the whole system. The same problem appears with scaling of system resources and handling failure of parts of the system. Suppose we fail to establish distinct boundaries between system components. In that case, we cannot operate a single part of the system on more robust hardware or restart a single part without restarting the entire system. This situation will add significant waste to our system resource use and increase downtime. Similarly, upgrading to technologies that better fit our purpose is more challenging to do gradually without the flexibility of microservices.

2.2.2 Challenges of microservices

While microservices benefit certain kinds of systems, they also come with challenges. Baresi and Garriga (2020) mention dimensioning as a primary microservices design challenge. Dimensioning is the process of finding the right balance between the size of each service and the number of services in the system. It was earlier discussed that microservices are a good choice for organizations that prefer smaller, autonomous teams. However, the exact split between different responsibilities and functions is unclear.

Baresi and Garriga also mention security as an open challenge for microservices. Microservices architecture increases the number of different endpoints in a system. This distribution increases the access points adversaries can exploit, making security efforts more widely spread and decentralized. The security approach in the organization needs to be changed when using a microservices architecture. Dedicated security teams must rely more on automated tooling to audit various services, while individual teams must take more security responsibility.

Baresi and Garriga argue that some unique challenges also appear during service development. As in reality, the services are rarely fully isolated, mechanisms for service-to-service communication must exist. The paper mentions RESTful HTTP communication and message queues. Newman (2021) also mentions that remote procedure calls (RPC) are still viable for internal service-to-service communication, particularly with technologies such as gRPC that enable more efficient communication between services. Baresi and Garriga (2020) mention that message queues have only partially made a breakthrough due to the lack of precise methods for asynchronous interaction. Synchronous communication also introduces challenges as it creates coupling between services by requiring both to be fully functional at a specific time for an operation to be completed. Different teams developing microservices must negotiate with each other on changes in service communication, essentially introducing internal communication contracts and endpoint versioning. OpenAPI is widely used in public RESTful APIs, but the paper also mentions it as a viable option for internal microservices contracts. As mentioned in the introduction, these communication-related challenges are explored in this thesis.

Lastly, the complexity of testing is also mentioned as a challenge for microservices (Baresi and Garriga, 2020). End-to-end testing especially presents apparent challenges. Many services must be set up and connected for a full end-to-end test. This process is often challenging to set up, quickly becomes brittle, and must be frequently maintained as the services rapidly evolve. Challenges in end-to-end testing will be further discussed later in the thesis,

along with the other options that have gained popularity with microservices architecture.

2.3 HTTP RESTful

While other technologies exist to communicate between microservices, HTTP is the primary mechanism explored in this thesis. HTTP RESTful is now a relatively well-known concept; however, there is some unclarity regarding what it entails. According to a paper by Pautasso, Wilde, and Alarcon (2014), REST, short for Representational State Transfer, is a style of software architecture that prioritizes scalability and performance. Originating from the other styles, such as layered and client-server models, it organizes systems into three main parts: the origin server, the user agent, and intermediaries, such as proxies. Due to its matching priorities and interoperability across technology stacks, some RESTful practices are often applied in microservices.

One feature distinct to REST is the uniform interface for all architectural components, which leverages globally unique identifiers, such as uniform resource identifiers (URI), to label web resources. This architecture allows the system state to be represented and modified consistently, supporting efficient data exchange between clients and servers. The paper uses the word representation without adequately defining it. Representation refers to how a resource's state is presented externally to the client in a specific format. It is the external interface to the resource. According to the paper, REST mandates that resource state must be retrievable and modifiable through the representation. This exchange must also be done through self-descriptive messages containing the data and metadata necessary to interpret the content of the message. External documentation should be optional for this process.

Finally, REST constraints require that references to other related resources are embedded in the interface itself, usually by embedding the URI in the response metadata. However, this is where the definition of REST and practice differ the most, as RESTful APIs rarely fulfill this constraint. In practice, RESTful APIs often require external documentation for related resource discovery and even for correct interpretation of the messages. These APIs differ from the exact definition of RESTful; however, as APIs that do not entirely fulfill these constraints are still often called RESTful, this thesis will not try to change how RESTful is commonly used.

HTTP is a network protocol that can implement an API following REST design constraints. HTTP contains at least two aspects that work well with the RESTful design principles. Firstly, the HTTP protocol defines a clear set of methods (e.g., GET, POST) that represent how they will affect the resource's

state. The GET method retrieves the resource state without side effects, while the other methods primarily represent different mutations. They give us a way to provide these RESTful functions out of the box. Each of these methods can be consistently combined with any HTTP URI. Further, HTTP protocol contains a predefined way of transmitting metadata within messages and has a set of status codes describing the outcome of an action on the resource. These properties match with the RESTful self-descriptive messages principle.

2.4 Testing in microservices architecture

Testing in a microservices architecture is similar to other architectures but has different priorities. Testing can be split into four primary areas, as described in Figure 3 (Newman, 2021). These areas consider both the business and technology perspectives. Testing ensures the product fulfills the business use cases and supports programmers in building a system that fulfills the specification. It can also include consideration for both functional and non-functional attributes. Non-functional requirements are tested through property and exploratory testing, while functional requirements are tested through acceptance testing. While unit testing also tests functionality, each test usually does not verify a high-level functionality requirement but instead, low-level requirements set by programmers for how each unit should function.

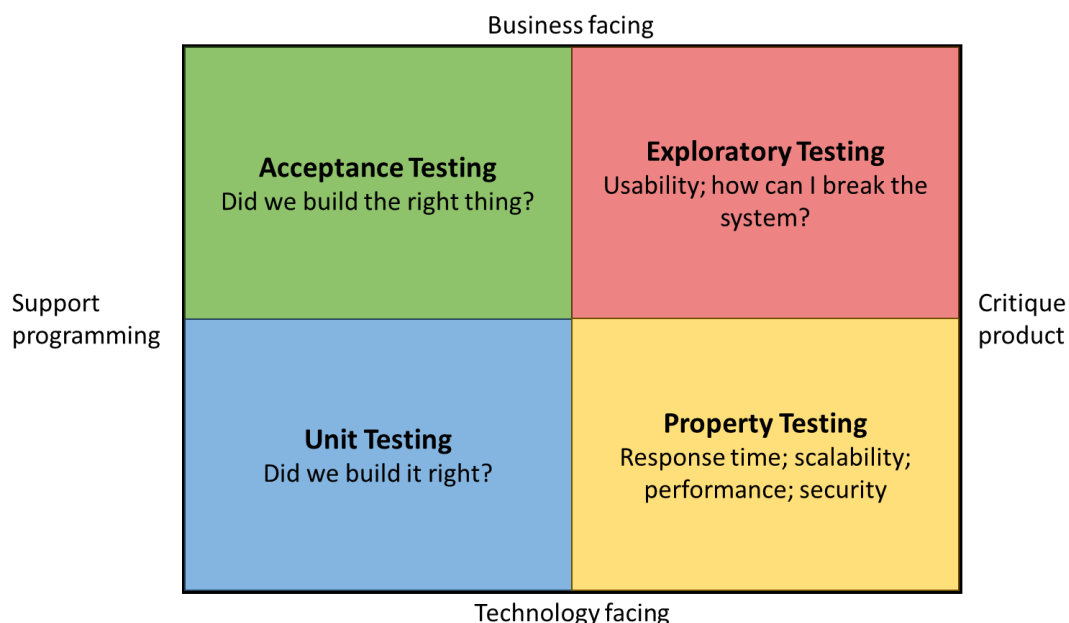


Figure 3. Adaptation of Brian Marick's testing quadrant (Crispin and Gregory, 2009; Newman, 2021).

Recently, there has been an increasing focus on automatic testing (Newman, 2021). Most tests in the testing quadrant can be done automatically, except exploratory testing. Exploratory testing is, by definition, manual. It is a process where completely unknown issues are discovered. Automated tests are usually split into different categories based on granularity. The names of the different categories depend on how we define them. Unit testing is a very established term, but what is meant by unit varies. A unit can be, for example, a single function or method call. Other higher granularity tests can be service tests and end-to-end tests. The granularity of a service depends on the service architecture. End-to-end tests typically test whole business functionalities and can be used as automated acceptance tests. The idea of testing boundaries is illustrated in Figure 4.

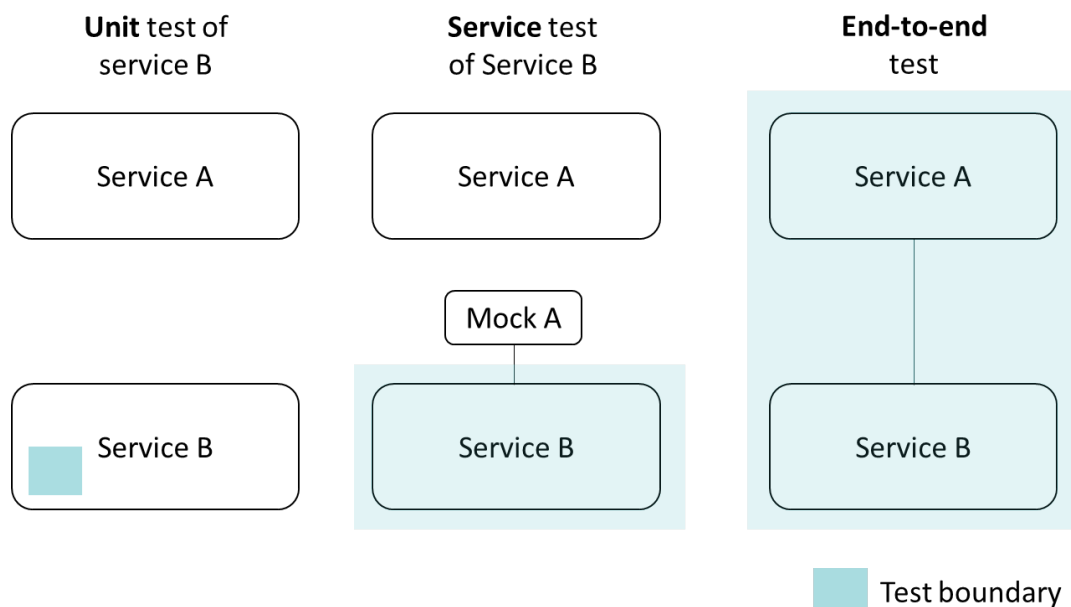


Figure 4. Test boundaries are the main differentiator of test categories. Adapted from Lehvä, Mäkitalo and Mikkonen (2019).

The testing pyramid in Figure 5 further highlights the test categories at different levels of granularity (Newman, 2021). In the pyramid, the size of the triangle base, or the area of each segment, illustrates the number of tests. The pyramid shows that tests at lower granularity usually provide more confidence but slow down the feedback cycle by increasing test runtime and reducing isolation. The consequence is that fewer end-to-end tests than unit tests can be written given the same time budget. The pyramid is a similar illustration to the scope, cost, and time triangle in project management. It shows that tests with a broader scope, while providing more confidence, usually have a cost in execution speed and isolation. When writing tests, confidence is preferred to be balanced with the costs while considering the business's priorities (Newman, 2021). The tooling and how tests are written affect

confidence, speed, and isolation. These factors can affect how many of each test should be written.

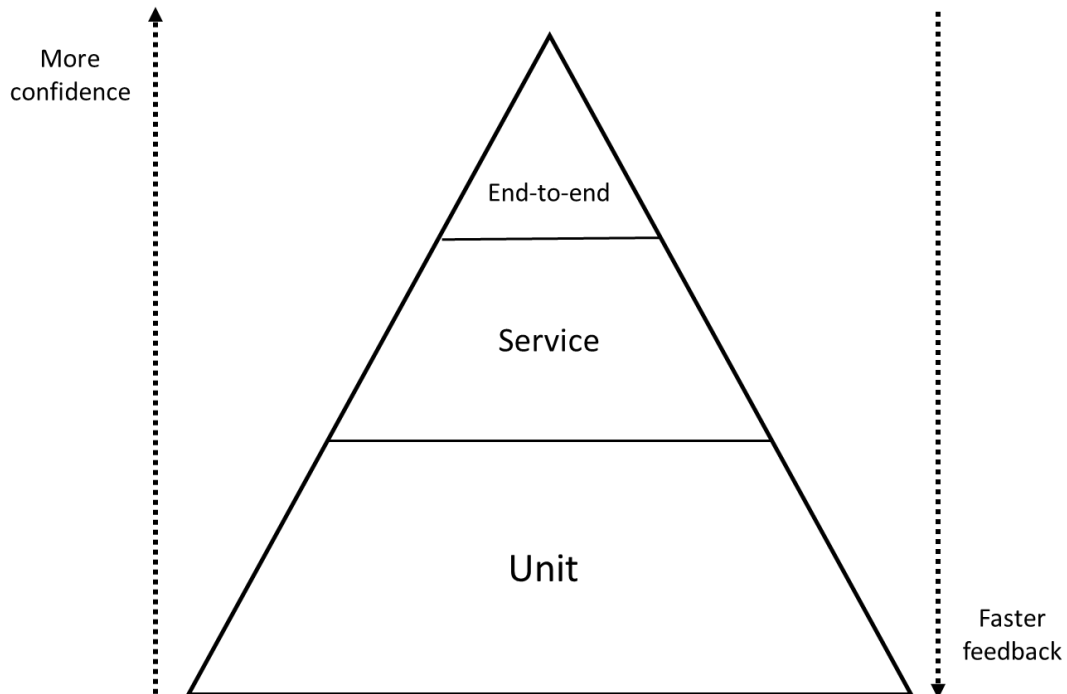


Figure 5. Adaptation of Mike Cohn’s testing pyramid (Cohn, 2010; Newman, 2021).

The tests that differ in a microservices architecture are the service, end-to-end, and contract tests. Newman (2021) also mentions testing in production as an emerging method for microservices testing. However, this is not explored in the thesis. The service test scope is usually one microservice; other downstream services are either stubbed or mocked (Newman, 2021). Stubbing or mocking means the response is replaced with a “fake” response that satisfies the minimum requirements for the tests to execute successfully. In mocking, we verify that the request is in the expected format. Using these techniques, we avoid the need to integrate with all dependencies in each testing scenario. The concept of a service test is specific to service-oriented architectures like microservices. However, fundamentally, it is like API testing with a specific scope.

End-to-end testing can exist in any architecture. End-to-end tests are also sometimes called UI tests (Newman, 2021). End-to-end tests can be more tricky in microservices architecture due to the number of separate build pipelines (Newman, 2021). Each microservice must be set up and deployed in some environment to run end-to-end tests successfully. In practice, this can be tricky. Ideally, the end-to-end tests must also be connected to the build

pipelines of all services to include them in the service development feedback cycles. Since end-to-end tests include all services, and services often work as ownership boundaries, it is sometimes difficult to determine who writes and maintains the tests (Newman, 2021). Ownership is a significant concern, given that the complexity of end-to-end tests can easily lead to brittle tests due to network failures.

2.4.1 Consumer-driven contract testing

Given the difficulties with end-to-end testing, especially in a distributed organization, other ways have been invented to complement service testing. One of the main methods is contract testing (Robinson, 2006; Clemson, 2014; Newman, 2021). *Contracts* form between services when one service couples to a resource provided by some other service (Clemson, 2014). This process is implicit and happens even when the contract is not explicitly written down. The contract consists of the parts of the resource the service requires. If these parts are suddenly changed, the service will fail. In practice, a contract can be defined in two ways, either as a provider contract or a consumer contract. The provider is the service that holds the resource and provides it to other services through an API. The consumer is the service dependent or coupled to the provided resource.

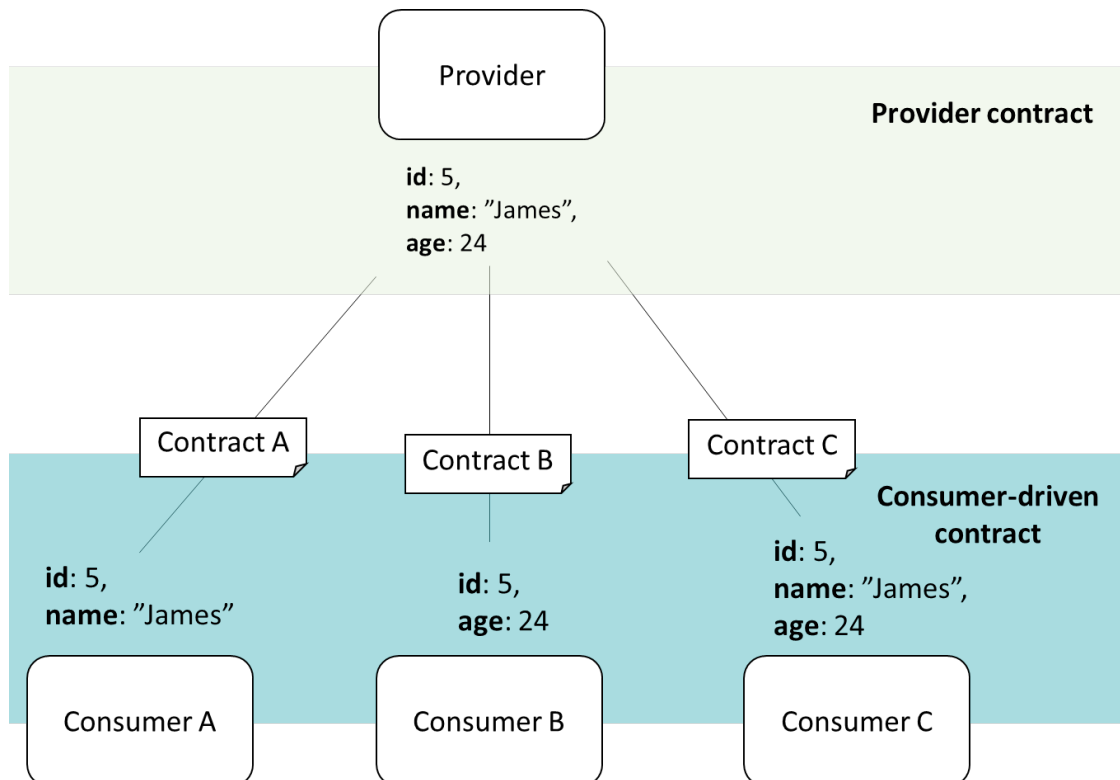


Figure 6. Illustration of provider and consumer contracts. Adapted from Clemson (2014).

The provider service defines *provider contracts*. These contracts describe the provider's exposed capabilities (Robinson, 2006). There is always only one provider contract per service, although it may be separated into multiple units/files in practice. Provider contracts are usually immutable and evolve using versioning. Contracts may be changed without introducing a new version by making backward-compatible changes. These are considered minor version upgrades. Any change requiring the consumers to be updated must be introduced in a completely new version. Figure 6 illustrates what forms the provider contract and where it is defined.

Consumer contracts capture the dependency relationship between a specific consumer and provider. Unlike provider contracts, consumer contracts do not include all providers' capabilities (Robinson, 2006). Each consumer has a separate contract with each provider. Providers have contracts with all the consumers that are dependent on them. This, along with the boundary of a consumer-driven contract, is illustrated in Figure 6. In the figure, each consumer has a consumer contract. However, the provider's consumer-driven contract consists of the sum of all consumer contracts. The benefit of consumer contracts is that they explicitly document the dependency between the services. Provider contracts are a helpful reference for the provided capabilities. However, the provider must also be aware of which capabilities are used. When the provider contract needs to be changed, it is risky because there is no way to ensure the change will not break a dependent service. Using the consumer-driven contract approach resolves this issue. When a provider knows all consumer contracts, it knows precisely which capabilities are in use and where (Robinson, 2006). This knowledge helps the provider evolve safely. Any functionality that the consumers do not use is known and safe to change without consulting other services. Even used functionality can be safely evolved when a complete set of tests is written for the consumer contracts. This is called consumer-driven contract testing.

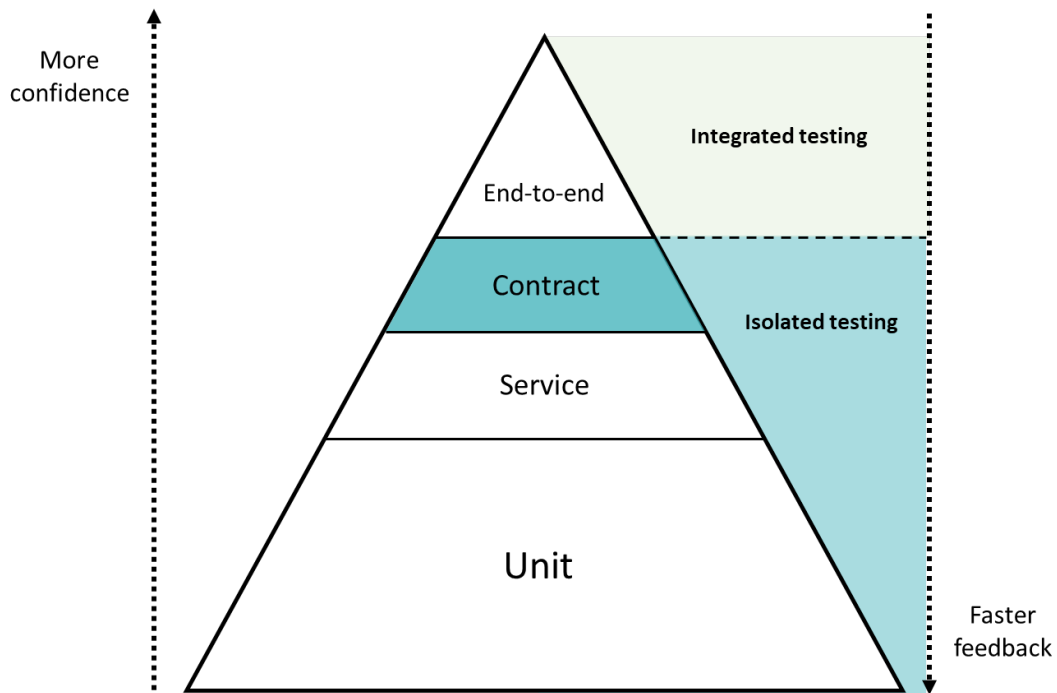


Figure 7. Adaptation of Figure 5 with contract testing included. Idea from Lehvä, Mäkitalo and Mikkonen (2019).

In the testing pyramid shown in Figure 7, contract tests exist at the same level as service tests. This type of test is isolated and does not directly require service integration (Lehvä, Mäkitalo and Mikkonen, 2019). All the test types can be split into isolated and integrated testing. Contract tests can be executed similarly to service tests except with the inclusion of the contract as the source of truth for which tests should be executed. Compared to end-to-end tests, the main benefit of contract tests is that they still manage to avoid explicit service integration while complementing confidence from service tests. Contract tests focus on asserting the content of the interactions between services. In the Figure 6 example, consumer A contract tests may assert that the fields **id** and **name** are in the response body and that the response status is 200 (OK). On the other hand, consumer B contract tests would test for **age** instead of **name** and so on. No additional unused fields should be asserted.

2.4.2 Pact contract testing framework

The *Pact contract testing framework* provides all the tools necessary for consumer-driven contract testing. It can be used to contract test HTTP and message integrations (Pact Foundation, 2023a). Pact consists of command line tools and libraries for different development environments. The Pact Broker is a closely related project (Pact Foundation, 2023b). Pact Broker is a service with tools to integrate contract testing into the continuous integration pipelines and share contracts between consumer and provider test environments.

This is a vital step to always test providers against the latest consumer-driven contract.

In the pact framework, contracts are called *pacts*. Each pact consists of *interactions* (Pact Foundation, 2023a). In HTTP, interactions consist of an expected request from a consumer to the provider and a minimal expected response. When testing against the consumer, pact automatically mocks the provider based on the defined interactions. The pact provider mock receives actual requests from the consumer under test and compares them against the expected request. If the request matches, a mock response is returned to the consumer to complete the test case. When testing against the provider, pact mocks the consumer and sends an expected request to the provider. It then receives the provider's actual response and compares it against the expected response, completing the test case.

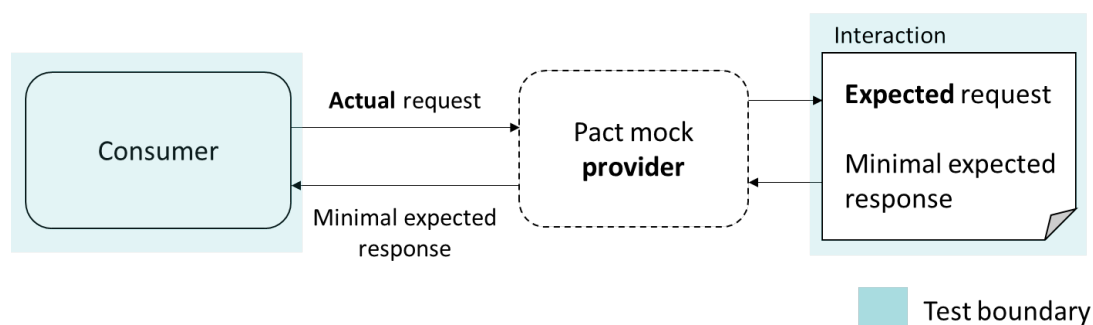


Figure 8. Consumer-side tests **compare** requests and **mock** responses. Adapted from Pact Foundation (2023a).

Pact testing is therefore implemented in two stages (Pact Foundation, 2023a). First, tests are implemented on the consumer side. Consumer-side tests verify that the actual consumer requests match the expected requests recorded in the contract interactions. They also verify that the consumer can function correctly with the recorded minimal expected response by mocking the provider response in consumer-side test cases. The idea is shown in Figure 8. In summary, the interaction itself is tested to ensure the contents are compatible with the consumer.

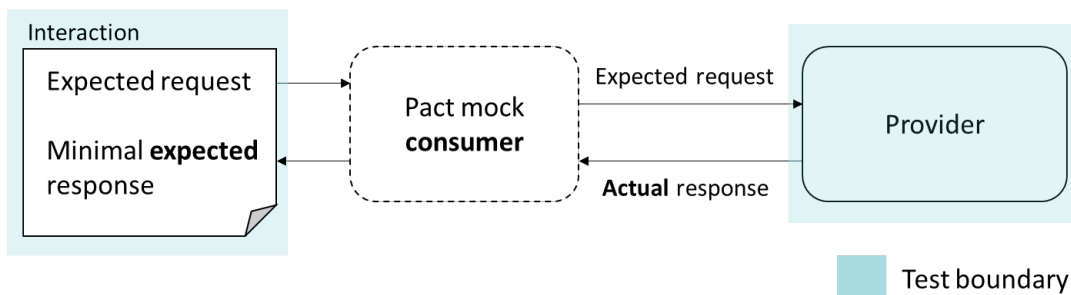


Figure 9. Provider-side tests **mock** requests and **compare** responses. Adapted from Pact Foundation (2023a).

Lastly, the interactions are used in provider-side tests to verify that the actual provider responses fulfill the minimal expected response. A request that matches the expected interaction request is made to the provider. Provider-side testing is illustrated in Figure 9. This process finalizes the verification process. By verifying that the provider is compatible with the interaction, we indirectly verify that the provider is compatible with the consumer, as both sides are provenly compatible with the same interaction. This idea is illustrated in Figure 10.

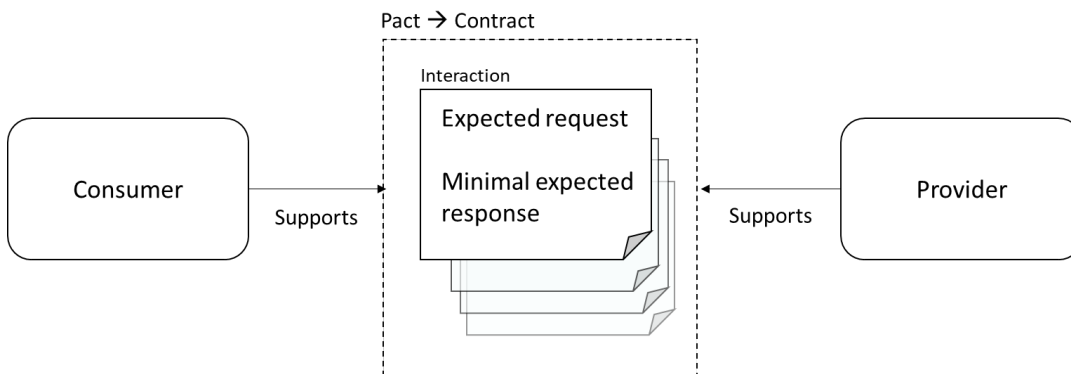


Figure 10. The goal of the process is to verify consumer → provider integration indirectly.

The mock server, the contract format, and the related tooling in coding environments form the core of the Pact framework. As mentioned, the Pact Broker service can optionally share the contract. This service supports publishing contract files from consumers and fetching them to providers. The idea is shown in Figure 11. Pact Broker was developed to address the fragmentation of relevant information across multiple locations (Pact Foundation, 2023b). It helps all continuous integration pipelines to access required information consistently from the same location.

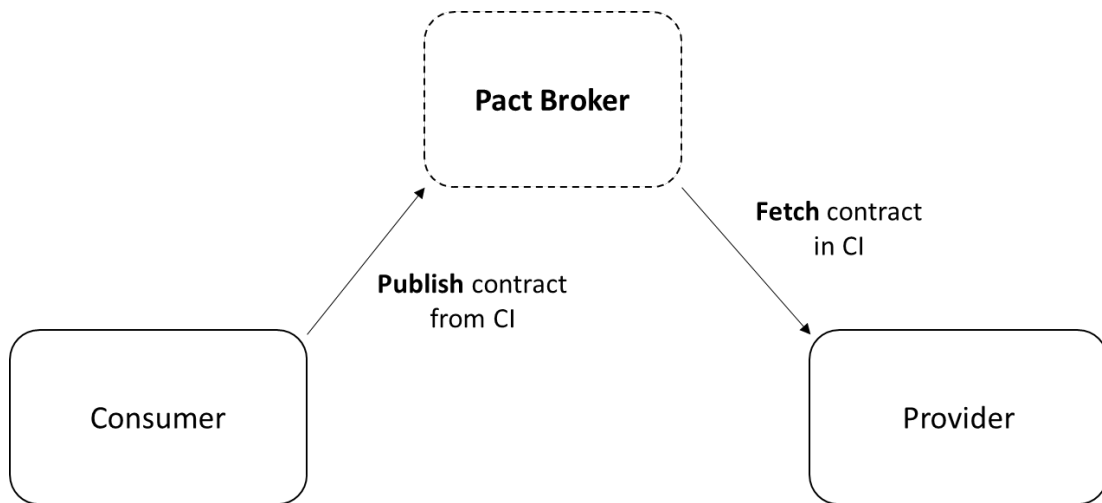


Figure 11. The primary role of a Pact Broker is to facilitate contract sharing.

2.5 Relevant case studies

This section introduces the primary case studies that influenced the decisions made during this thesis. The first case study is a paper on implementing consumer-driven contract testing using Pact in a microservices architecture. This is the most relevant case study, as it is almost the same scenario as in the thesis. It works as the primary prior research on the topic. The second case study introduces some thoughts from the industry through a blog post from Spotify on microservices testing.

2.5.1 Consumer-driven contract tests for microservices

The first case study was done in a system using microservices architecture (Lehvä, Mäkitalo and Mikkonen, 2019). The objective was to study how microservice integrations can be tested more efficiently using the consumer-driven contract testing approach. They discuss how the responsibilities of the testing methods are affected by the introduction of consumer-driven contract testing.

The case study was done with one feature team. This team had earlier implemented unit, service, and end-to-end tests. They had implemented service tests for every endpoint and found them easy to implement. In their service tests, they utilized mocks to isolate the service and verified that services were making external calls correctly.

While the unit- and component-testing approach worked for a while, the team's confidence faded as the number of endpoints and integrations increased. End-to-end tests for the most critical functionalities were the first attempt to increase confidence. However, end-to-end tests came with

challenges. Many challenges are well known from the literature published before this case study (Wacker, 2015). They write that the lack of isolation in end-to-end tests required more effort. The test development required running all services and databases on the developer's local machine. Executing the end-to-end tests required significant planning, with the whole system being first initiated into a specific state with mock data. The team approached this additional challenge by implementing new endpoints and seed SQL files solely for the end-to-end test set up. Still, false positive errors occurred locally and in continuous integration. These were often caused by development or other tests modifying the system's state, requiring manual work to resolve. Additionally, with lost isolation, debugging errors became challenging as errors needed to be traced from the log data of multiple services. The team felt that end-to-end tests were slow, prone to errors, hard to debug, and non-deterministic.

Consumer-driven contract tests were implemented using Pact as an isolated alternative to more end-to-end tests. The team used Pact Broker to share contracts between services. The impression was that consumer and provider test implementation differed significantly. On the consumer side, tests were mainly about writing the expected requests and responses for the Pact mock. The provider side consisted of only ensuring the provider was available and in the right state for test execution. The team always implemented a 200 OK happy path case for each integration. The most tested error case was the 400 Bad Request error.

The study compared the testing methods by seeding defects to the integrations. They found that consumer-driven contract tests could catch every defect from the 23 interactions. Component tests allowed some defects to slip through, and end-to-end tests could catch defects but lacked test coverage. The study concludes that consumer-driven contract testing brought confidence to the testing strategy. Aside from catching all the defects, they found that with contract tests, it was impossible to make the tests pass without fixing the underlying problem. Unlike end-to-end tests, they were also fast to execute and isolated from other services, leading to no false negatives.

Further, they concluded that they could confirm the stated benefits of consumer-driven contract testing. These benefits are:

1. Decoupling testing of the consumer from the provider.
2. Providing fast, stable, and deterministic execution.
3. Providing information about the consumers of a provider API.
4. Preventing breaking changes during the provider test execution.
5. Improving communication between teams using contracts as a tool.

The paper recommends the spike method for determining if consumer-driven contract testing suits the specific use case. In a spike, the change is

first implemented only for select system features. This proved helpful for learning about the method and its pain points. They also mention it as a tool for communication inside the organization. Communication is an aspect they emphasize for contract testing implementation. In this paper, the implementation was done by one team, and no other teams were involved. The cross-team communication is mentioned as a possible additional challenge. However, they also mention that contracts can improve communication if implemented, and this improvement can provide additional benefits. Lastly, they mention that one team could help implement the tests for other teams, reducing the issue of synchronizing teams to adopt the methodology.

The paper states that the ideal testing strategy for microservices consists of unit, component, consumer-driven contract, and end-to-end tests. They state that integration tests can be removed and replaced with contract tests. The reduced number of integrated tests is an additional benefit of contract testing with less flakiness and non-determinism, leading to enhanced automation and debugging. In distributed microservices, the ideal testing strategy depends on team communication. They conclude that good communication should be a top priority.

2.5.2 Testing of microservices at Spotify

An additional industry case study comes from Spotify (André, 2018). The blog post by André provides further perspectives on microservices architecture testing. The blog post highlights that automated testing is not adopted solely for confidence but also to enable faster feature development. They mention that this can be accomplished through fast, reliable feedback and easy maintenance. They highlight that it can be challenging to accomplish this in microservices, especially when following the traditional testing pyramid. They state that the core reason is the shift in complexity from the service itself to the interactions between services. The blog post also criticized unit tests by stating that they often require changes when the code is updated, reducing the confidence they provide.

The blog post's solution is the “microservices testing honeycomb,” where integrated and implementation detail tests are reduced. An adaptation is shown in Figure 12. Implementation detail tests refer to traditional unit tests. They state that this approach essentially makes the microservice the new “unit” that testing revolves around. The integration tests also likely refer to service tests as the testing happens at the interaction points, such as the API of the service. They mention that using service tests over unit tests reduces the feedback's accuracy through reduced isolation. This can make finding the root cause more difficult, but they believe the benefits outweigh this limitation. The blog post states that implementation detail tests should be limited

to parts of the service that are naturally isolated and have high internal complexity.

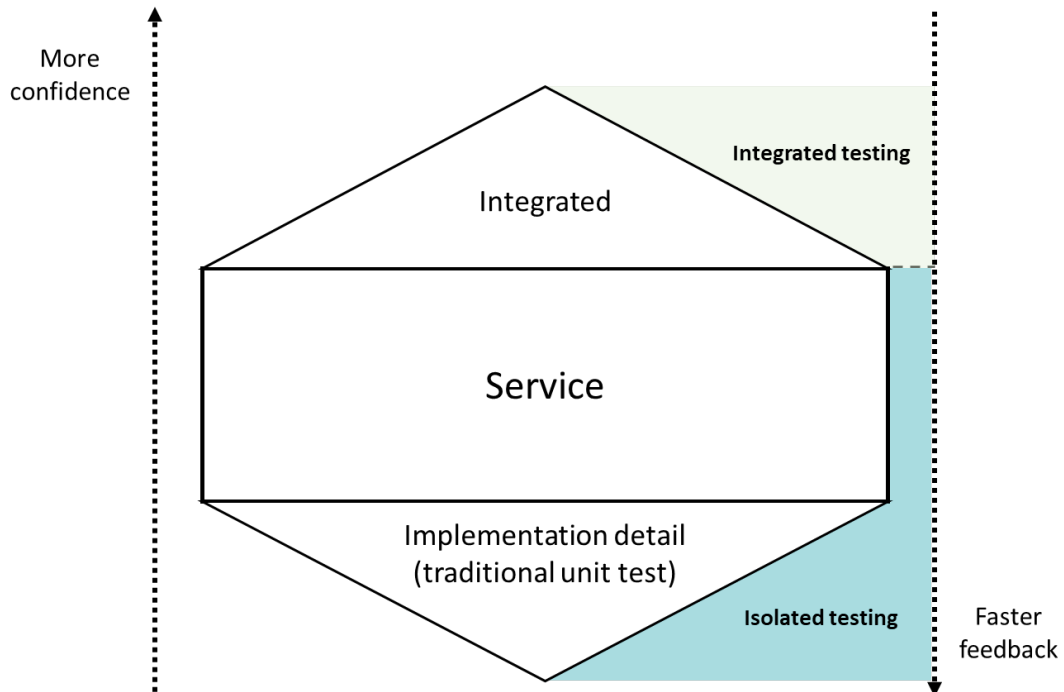


Figure 12. Adaptation of the testing honeycomb from André (2018).

Lastly, they conclude that this approach gives confidence and requires less maintenance than traditional unit tests. The only section where this approach falls short is the speed and accuracy of the feedback. Importantly, they note that this approach is a gateway to gaining even more confidence through *consumer-driven contract testing*, as much of the tooling can be re-used. In summary, the Spotify testing approach involves placing the service as the central abstraction while avoiding integrated tests and excessive testing of implementation details.

3 Research methodology

This chapter presents the research methodology of the thesis. Initially, three different research methodologies were compared, and their suitability for the thesis research was evaluated. These methodologies were case study, action research, and design science. As the primary research objective was to discover and evaluate solutions to a problem, and the research was restricted to one case company from which the problem was derived, *action research* was considered the most appropriate research method. Under different circumstances, a (multiple) case study could also be used to discover and evaluate existing solutions to related problems. Such a study would remove the bias of the researcher being actively involved with the case company and the research problem itself.

In practice, the research process followed the description of action research by Staron (2020). An existing study using similar research methods was used as a reference to further understand the practical application of the methods and validity concerns (Kauppinen, 2005). In the diagnosis phase, grounded theory was used to understand quality assurance challenges at the case company. The definition of grounded theory was based on Lazar (2017) and a description from the reference study. For example, the stages that were applied are

1. Open coding
2. Development of concepts
3. Grouping concepts into categories
4. Formation of a theory.

In the diagnosis and evaluation, we used triangulation with document sources, observations, and conversations/interviews to improve the study's internal validity. However, generally, the internal validity is average. This is due to the small number of interview participants, the short study duration, and the lack of investigator triangulation. The author of this thesis performed all analysis and data collection. Participants from the case company were used to validate the understanding of the architecture, processes, and other results when possible. However, no additional third parties reviewed the analysis or interview questions to further avoid bias. The study's external validity is slightly improved with comparison to prior case studies, especially Lehvä, Mäkitalo, and Mikkonen (2019). However, as only one case company was included in the study, the external validity is limited. In the following sections we will discuss the research process and each stage of the study in detail. A summary of the research methods and the collected data is shown in Table 3.

Table 3. Summary of research methods.

Research methodology		
<i>Method</i>	<i>Duration</i>	<i>Participants</i>
Action research	6 months	One case company, 11 engineers from two teams
Data collection		
<i>Method</i>	<i>Data source</i>	<i>Data</i>
Document review	Incident postmortems	25 incidents over 2 years
	Team retrospective workshop notes	39 data points (post-it notes) from 2 teams over 2 years
Interviews	Team engineers	3 for diagnosis stage, 1 for review of architecture and processes and 4 for evaluation stage
Quantitative metrics	Incident count in participating services	8 incidents related to service combability (2 years) before action, 0 incidents after action (less than 6 months)
	CI test failures	11 unexplainable failures for end-to-end tests over one month, 0 for contract tests
	CI build times	Around 3 minutes for test execution both before and after contract tests
Data analysis		
<i>Method</i>	<i>Participants</i>	<i>Data</i>
Grounded theory	Thesis author	80 post-it notes grouped into 34 concepts under 4 main categories/themes

3.1 Research process

The research process followed the action research cycle. In this cycle, the research is performed in the following phases (Staron, 2020):

1. Diagnosing
2. Action planning
3. Action taking
4. Evaluating
5. Learning

The research process was conducted within six months. During this time, one cycle of action research was executed. In the following segments, the process and methodology for each phase are explained. Additionally, differences between the original plan for each phase and the process in practice are presented.

3.1.1 Diagnosing

The goal of the diagnosing phase is to use data directly from the case company to better formulate the precise research problem and specific research questions (Staron, 2020). The diagnosing process was started with a document review. A collection of incident post-mortem documents and retrospective workshop deliverables were primarily reviewed. Initially, more document sources were planned for use. These are continuous integration logs, version control history, and team ticket management systems. It seemed possible that these systems could contain helpful information for the analysis. However, in practice, while the quantity of data was high, the qualitative depth appeared insufficient for meaningful analysis. These additional sources were instead moved for evaluation use only.

Additionally, observations, conversations, and short interviews were planned to support the analysis process. Extensive interviews, while traditionally a part of diagnosing (Staron, 2020), were deemed unnecessary due to the extensive discussions that preceded the start of this thesis research. More extensive interviews were instead planned for the evaluation phase.

The post-mortem documents contained records of incidents in production systems, a description of what happened, and an analysis of the root cause. The post-mortem practice employed is similar to that described by Google (Lunney and Lueder, 2017). Not all post-mortem documents were filled out; in these cases, further details were searched from instant messaging logs. The document was excluded from the study if sufficient information could not be found.

Retrospective workshops are regular exercises taken by teams to identify areas of improvement. They are usually organized using a form of brainwriting. The format includes a period where each participant independently writes post-it notes based on categories aimed at soliciting what has gone well, what issues have been encountered, and what could be improved. The format is similar to brainwriting as defined by Staron (2020) in the context of action research. The main difference is that retrospective workshops usually focus on improvements based on recent events and use common prompts to solicit ideas. However, sometimes workshops focused on soliciting ideas on specific topics with longer time window are also organized. This format

makes retrospective workshop deliverables a suitable source for identifying what practitioners in the team value and what they consider problematic.

After the document review, clarifying discussions were held to remove ambiguity and confirm understanding. The questions were based on the questions formed during the document review phase. At this stage, three team members from the participating teams were contacted. The gathered data was analyzed using grounded theory on a digital whiteboard. After the initial analysis, the result was used to search for more relevant literature. This literature was used to complement and finalize the analysis results.

Since the data concerned faults in software systems, some existing defect categorization systems were used as references. Wagner (2008) was used to overview different defect classification methods. However, none of these were directly helpful in this context, and the final categories were found creatively based on the data using existing systems only as a reference.

Other documents were also used to understand and analyze the current software engineering processes and architecture. These include architecture diagrams, code repositories, and technical documentation. This analysis was used as a basis for a model of the existing status quo. The results of a literature review and earlier analysis were used to look for clues on possible gaps or deficiencies in the current quality assurance processes. The processes and architecture, as presented in this thesis, were also reviewed by a team member.

Lastly, as mentioned as the purpose of diagnosing, research questions were formulated. Some initial understanding of how these research questions could be evaluated was also formulated. All the results can be found in section 1.3 of the thesis on research questions.

3.1.2 Action planning

Action planning is a collaborative phase where a plan for the diagnosed problem is formulated (Staron, 2020). In practice, the possible actions were discovered through a workshop and earlier discussions with one of the participating teams. In this workshop, team members could propose actions, and each action was evaluated for its value, effort, and impact on others. The actions with the best proportion of effort/value were considered for implementation.

Finally, after comparing the actions and analyzing their feasibility, a single action was chosen for implementation and evaluation. This was further confirmed with other participating teams through a request for comment (RFC)

process and presentations on the action idea. Both participating teams attended the presentation and approved the action idea.

3.1.3 Action taking

Action taking is the phase where the plan and intervention are implemented (Staron, 2020). The implementation was done primarily by one team with collaboration from the second team. At first, the author of this thesis created a small proof-of-concept of the idea (spike). The team reviewed this initial implementation. Then, the final implementation was scheduled for the team roadmap and later executed. The final implementation was done by the thesis author and one other engineer. An additional engineer from the second team was assigned to perform reviews of some parts of the implementation that were most relevant for their team. The use of proof-of-concept or spike implementation and using one team to primarily drive the implementation across services was supported by conclusions in prior research (Lehvä, Mäkitalo and Mikkonen, 2019).

The scope of the action was an initial implementation in three services. When entering the evaluation, many improvements were still left out of scope. For example, as the action related to testing, the test coverage for the services was not complete during the evaluation. One of the services had near-optimal coverage, but the other two were still lacking in significant areas. The process was also immature and could still be iterated for further improvements. These facts place some limitations on the interpretation of the results.

3.1.4 Evaluation

Evaluation is the stage where the impact of the action on the organization is evaluated (Staron, 2020). According to Staron, this stage includes mainly the analysis of data. However, in this thesis, evaluation also included conducting interviews, performing the final data gathering, and analyzing the data. The first step in the evaluation was the collection of incident data and relevant events from source control systems and continuous integration pipelines. We collected logs of who had interacted with the contract testing setup and used this to select interview participants. We also checked for test failures on continuous integration logs and calculated the test runtime. The incident logs were explored for relevant incidents. The goal of the interviews was primarily to uncover participant opinions based on direct experience with the tool. We did not attempt to interview developers lacking experience with our contract testing setup.

The interviews were conducted in a semi-structured manner. There were only four structured questions to lead the discussion. The exact format of

these questions differed depending on the interview and interview language. Some interviews were conducted in Finnish and some in English, depending on the participant's preferred language. The discussion was directed based on the research questions and other material defined in section 1.2. In total, four participants were interviewed in this manner. All participants were software engineers of the case company working in the teams where contract testing was implemented. All interviews were conducted face-to-face in person. The number of interview participants was limited by the number of people with experience using the tool. As the action-taking period was relatively short, only a few engineers had direct experience working with the tool in a natural environment. Even among the four participants, only two had directly implemented new Pact tests. The other two had experience reviewing pull requests with Pact changes and working with the continuous integration pipelines running the tests.

The leading interview questions were similar to the following:

1. In what way have you worked with the Pact contract testing?
2. What challenges did you face while working with the tool?
3. How would you describe your overall experience working with the tool?
4. Do you have any improvement suggestions for how we use the tool or the process surrounding it?

Most of these questions are specific to direct experiences with contract testing. Question three is an open-ended question to uncover the overall mood and any other experiences that participants want to bring up. The last question was to uncover any shortcomings in how the tool was implemented and presented. The relevant points from the interviews were documented with notes, and the interview mood was generally kept casual to avoid causing stress to participants. The answers from participants were combined with direct observation by the author. The raw data from logs and interview notes was grouped and analyzed using the metrics and questions in Table 1 and Table 2. The aim was to find answers to research questions RQ2 and RQ3. The results were summarized using dimensions created based on Table 2 (see Table 7).

3.2 Case overview

The data collection and intervention were performed at one case company. While the case company contains many teams and functions, two teams primarily participated in the study. Both teams are developing features in the company's cloud system and are relatively independent in their feature development and chosen practices, although more general guidelines were followed.

3.2.1 System architecture

The architecture of a large-scale microservices system is highly complex, with many components and moving parts. For the thesis study, there is no need to describe every aspect of the system architecture. The architecture analysis focused on the systems most relevant to the teams participating in the thesis study. A diagram of this simplified view is presented in Figure 13 (see next page).

This architecture could be primarily split into three segments: the user interfaces, the main backend system, and the other systems consuming data from the main backend system. The primary purpose of the main system is to store and manage relevant user data. The component boundaries of the backends are made based on entity relationships. Independent entities are stored in their microservices.

This main system is manipulated through user interfaces. The system has two primary user interfaces: the dashboard and the public API. The requests to the main system do not go directly to the backends; instead, all requests are routed through a gateway service. This gateway provides shared functionality across all microservices, such as authentication, authorization, and rate-limiting.

The other systems consume the user data from the main system through a queue service implemented using Kafka. The main system is not concerned with how other systems use the data. The backend systems provide a contract for the data they publish, and if the contract does not change, other systems are free to consume the data as required. The queue decouples the systems, leaving the contract for the messages as the only shared model.

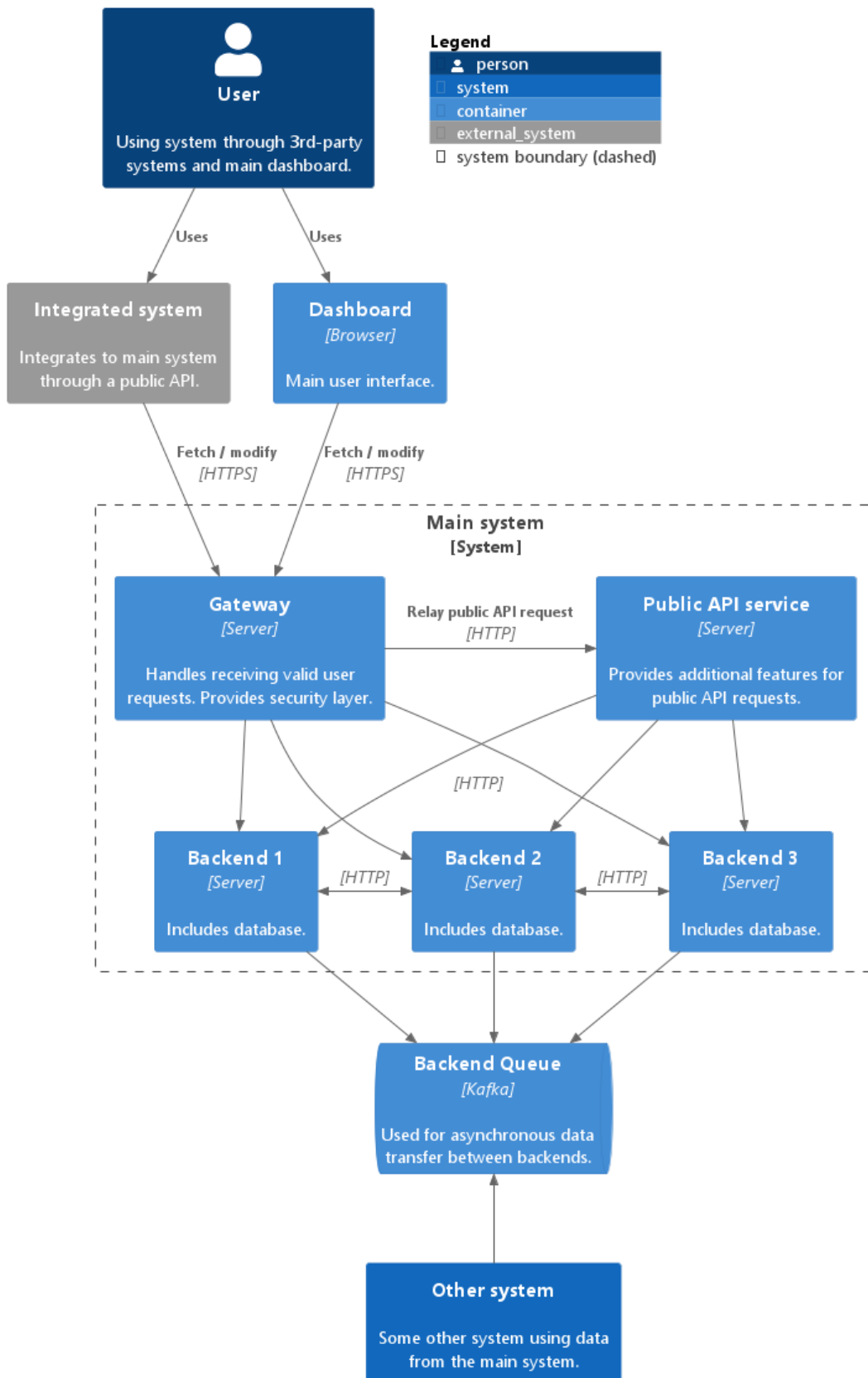


Figure 13. Simplified system architecture modeled using the C4 method.

The dashboard usage flow is the oldest in the system. Many backend services initially operated with the dashboard as their only consumer. In this flow, the same organization controls both sides of the system. Often, the same teams will work on the dashboard and the backend systems to implement features. This model usually means it is easy to change the backend provider systems without worrying about backward compatibility, as the dashboard can be easily changed by the same team changing the backend. The architecture for only the dashboard flow is shown in Figure 14.

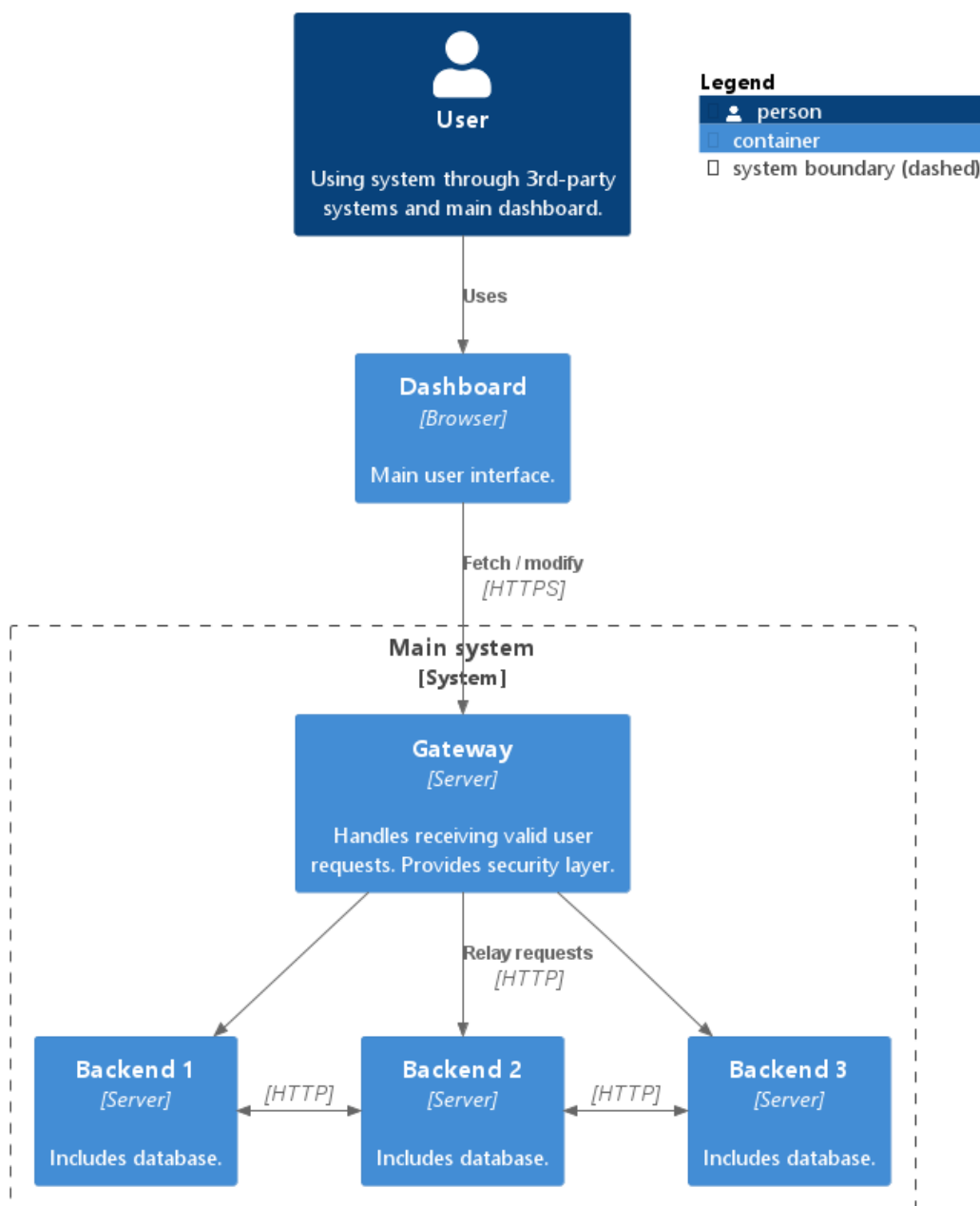


Figure 14. The system has the dashboard as the only consumer.

Public API support was added after the dashboard flow had already matured. This introduced some additional design considerations not previously fulfilled by the original system. The decision was made to implement the public API support through an additional utility service that addressed the unique concerns of the public API. Due to the lack of control of the integrated consumers, versioning and controlled deprecation became paramount. Additionally, detailed endpoint schemas, more descriptive error messages, and documentation were needed to simplify integration. The public API service was developed to address these concerns for all backends simultaneously. The organization structure was also aligned with this decision by forming a new team for the public API service. Figure 15 illustrates the request flow when an integrated system uses the public API.

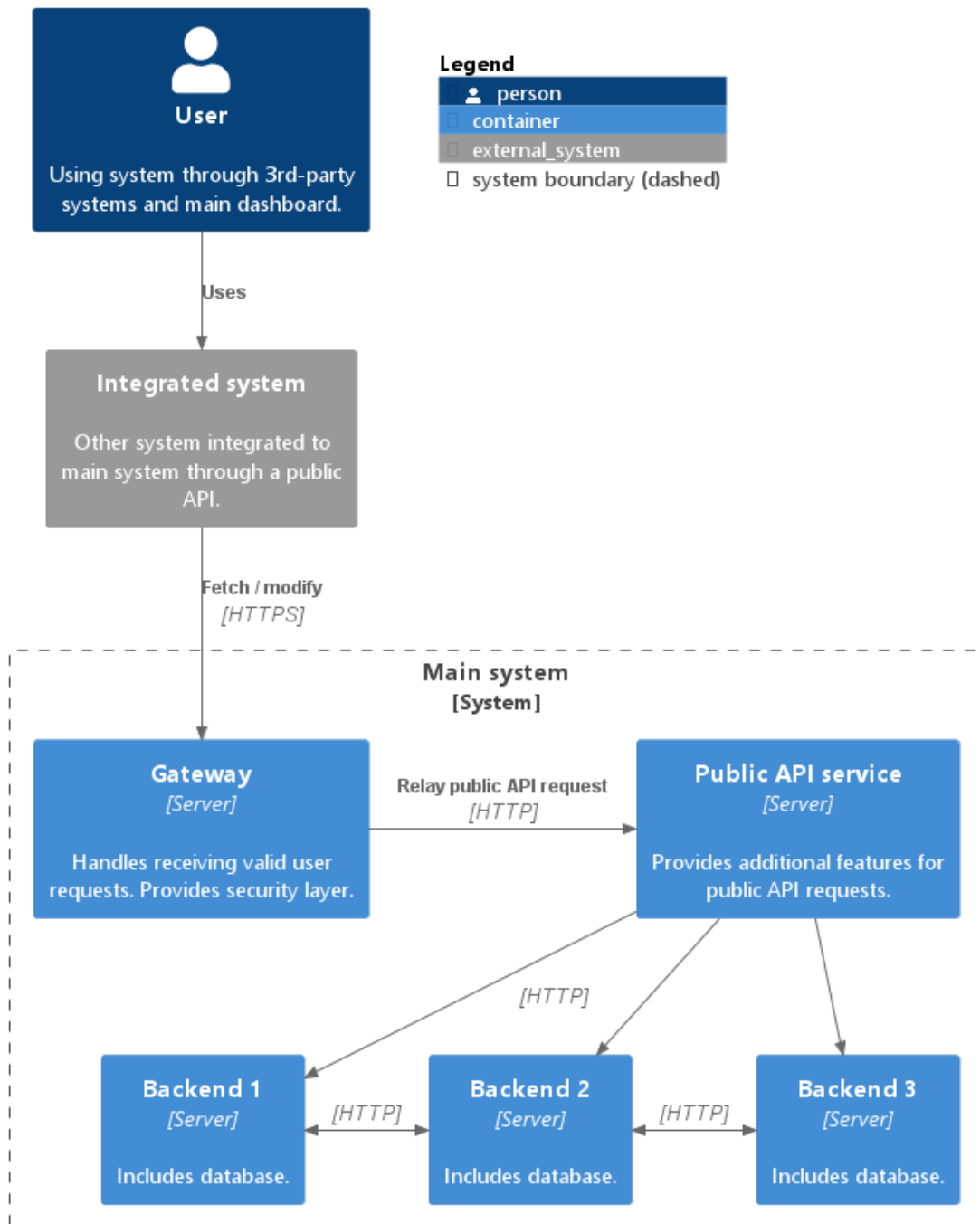


Figure 15. The system with an external integrated system is the only consumer.

3.2.2 System components for verification

Testing the system requires identifying which parts need to be tested. Since the system is split into independent components, it goes without saying that each component needs to be tested. This can be achieved through unit

testing. The unit can be defined to be an individual component or a sub-unit inside the component. In the latter case, sub-units would also need to be integration tested.

Aside from the components, the arrows between the components also model system behavior. Even if all the components behave as specified, the system will fail if the components call each other in ways not supported by the other component. This idea is nicely captured by Parnas (1971) in a way that was repeated in more recent work by Newman (2021):

“The connections between modules are the assumptions which the modules make about each other.”

Therefore, complete testing of the system consists of testing methodologies for both the components themselves and the connections between them. This same idea can be zoomed in or out depending on the desired level of abstraction. Systems marked as external in the architecture are the only aspects that cannot be tested. Interfacing with such systems emphasizes the importance of decoupling through stable, well-defined interfaces that only expose the minimum of what is necessary. This allows for maximum freedom internally without haphazardly breaking the external systems.

3.2.3 Software development process

Next, we will describe the current software development process. The analysis of the processes was scoped to only the most essential aspects of this thesis. This includes development and verification. Requirements engineering-related processes were excluded, even though they are also relevant for quality assurance in practice.

The analysis is started from a point in the process where a feature has been defined, prioritized, and selected for development. Starting from this point forward, there are usually multiple different paths that can be taken. Which path depends on how much communication is anticipated for the completion of the feature, and this usually scales with the complexity and size of the feature. More complex features require more communication on the design, and more extensive features touching upon multiple services require more team alignment.

Two main paths are the following:

- Some up-front design is performed into a technical design document (TDD) along with an explicit design peer review.
- Requirements and approaches are documented on an issue ticket, and work is started with informal discussions.

There is no clearly defined process for which approach to choose. However, as mentioned earlier, the complexity and size of the feature are considered. Writing a design document and performing a design review for all main roadmap features is standard practice. Other tasks, such as bug fixes, small features, and minor improvements, are usually finished without the design review process.

The software development itself could be split into approximately three main phases. The phases and their primary contents are described in Figure 16. The steps marked with a light cyan color are considered most relevant for quality assurance. Based on this process, we can identify the parts where actions could be performed to adjust quality assurance. These could be split into

- Automated code analysis
- Verification through testing
- Code/design peer-review.

We can also conclude that all testing and code analysis methods must be functional in the local development environment and the continuous integration pipeline if significant changes to the overall process are to be avoided.

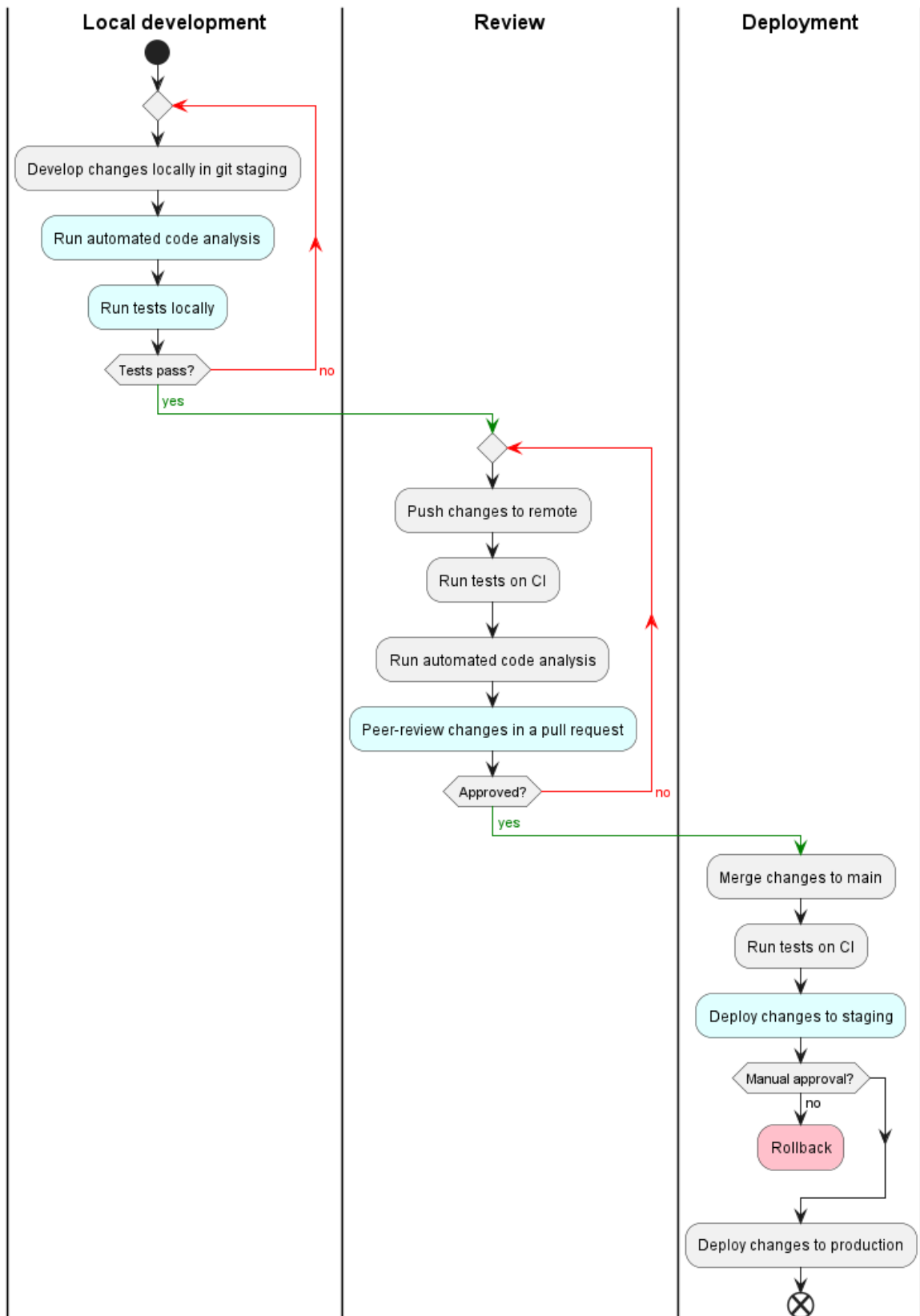


Figure 16. Simplified software development/release process.

3.2.4 Utilised testing methodologies

The type of testing depends on the specific service that is analyzed. However, in general, there are some commonalities between services. All services make extensive use of unit testing. Services also generally contain some integration or service tests against their service API. End-to-end tests are generally only found on the dashboard and public API levels. The dashboard end-to-end tests naturally test the system directly through the UI, whereas public API end-to-end tests run as if they were an external integration into the system. An end-to-end testing framework is utilized for this purpose. The idea is illustrated in Figure 17.

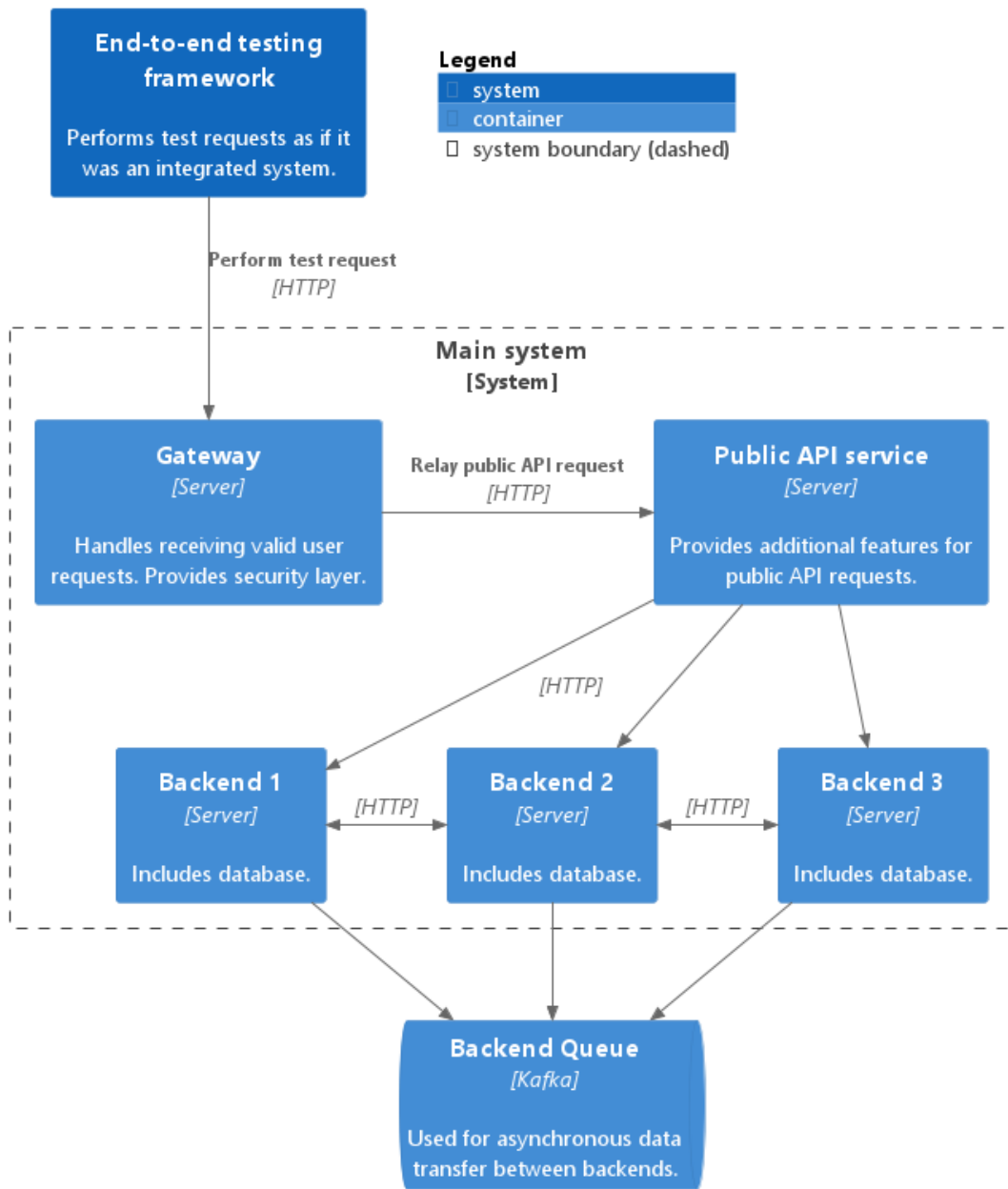


Figure 17. The framework calls the gateway, but only the public API flow.

Since the system contains two separate interfaces, each type of end-to-end test can only test one of the two flows. This means that tests on the dashboard cannot provide confidence about public API functionality and vice versa. In summary, all individual components are internally tested. Connections between components are also partially tested through the end-to-end testing methodology. The actual test coverage and limitations of the process will be further discussed in the results section.

3.2.5 Peer-review guidelines

Peer reviews are the primary method for detecting high-level specification or design mistakes or mistakes in the testing strategies themselves. It is, therefore, essential to explore how such reviews are performed. However, tracing exact methodologies for reviews is much more complex, as each developer generally has their methods. Some commonalities can be found by exploring documented peer-review guidelines.

An overall review/contribution guideline can be found in the context of the teams participating in the thesis study. This guideline primarily works as a reminder of what aspects should be considered when doing the code review. Some of the things that are considered in these guidelines are:

1. Communication style
2. Scope of the changes
3. Reasoning behind the changes
4. The understandability of the code and code style
5. The “cleanliness” of the commit history
6. The risk and impact of the changes
7. The test coverage of the changes
8. The logging and monitoring measures of the changes
9. Dependencies on other changes
10. The ability to verify the changes after deployment
11. The ability to roll back the changes after deployment.

As can be seen, the list is quite comprehensive. However, fundamentally, the judgment of each item is left to the person reviewing the code. Investigating and quantifying the use of this list is challenging. Some hints could be gathered by reviewing existing review comments, but vast amounts of data would be required to grasp the general trends as each review is unique and often made by a different person.

4 Results

This section presents findings from all stages of the research. We start by describing the quality assurance challenges discovered at the case company. We then continue by describing the evaluation of various strategies for addressing the challenges and, finally, the implementation and evaluation of our chosen solution.

4.1 Quality assurance challenges

Various quality assurance challenges were identified at the case company. As a part of the analysis, production incidents were categorized based on which activity in the system the incident was deemed to occur. Table 4 describes the final categories and some considerations for forming them.

Table 4. Categories that were formed from the data.

Category	Associations	Example fault
Service-to-service interaction	HTTP, Kafka, involves multiple services	Service unexpectedly changed the name of a required HTTP header
Data migration	Data model changed	A subset of data becomes invalid after data model migration
Internal service logic	Internal errors, only one service	Internal service operation fails due to a programming error
Infrastructure	Database, memory, CPU	A database runs out of memory

In total, 25 incidents were classified. Table 5 shows the distribution of incidents in each category split based on the overall severity of the incident. Severity was classified by the person who initially documented the incident. The categories are sorted based on the number of incidents while giving high-severity incidents more weight.

Table 5. Number of reviewed incidents per category.

Category	High severity	Low severity
Service-to-service interaction	3/25 faults	5/25 faults
Internal service logic	3/25 faults	5/25 faults
Data migration	4/25 faults	0/25 faults

Category	High severity	Low severity
Infrastructure	2/25 faults	3/25 faults

Possible reasons for the incidents were also explored. Many of the incidents were directly related to unique requirements introduced by microservices infrastructure, such as broken backward compatibility in the service's API. Other incidents were related to limitations of the technology used, such as long delays with message queue consumption, restarting services, and leader selection due to unavailable database nodes. However, the majority of incidents could be traced to limitations in processes.

Based on the data, most issues were already introduced in the design or specification. Pure coding mistakes also occurred, but they were rare. It could be hypothesized that the existing quality assurance practices are already efficient at preventing easily discoverable coding mistakes. Challenges were primarily found with peer reviews and testing. Challenges with peer reviews are related to ownership and occasionally insufficient review of non-code artifacts. At the same time, challenges with testing are related to the stability of testing environments, maintainability of tests, and lack of clarity on test coverage. Additionally, some testing challenges appear from the inconsistencies between testing and production environments.

Ownership in this context relates to a lack of knowledge by the reviewer in some relevant area. The distributed team structure introduced additional review challenges as teams often have limited knowledge of other teams' features. Similarly, especially when working with new services, lacking knowledge of the shape and format of production data or the exact functioning of the libraries and tools used in the service introduces challenges. Most issues occur in complex legacy services where multiple teams work on the same service, and changes can be complex. Infrastructure ownership was also found challenging.

Testing challenges were twofold; on the one hand, services that had end-to-end testing struggled with the test framework's stability; on the other hand, services that lacked such testing struggled with a lack of deployment confidence and insufficient test coverage. End-to-end tests were flaky and more challenging to integrate into the local development workflow. Engineers of the team confirmed that services with a lack of confidence did not have any way to automatically test whether consumer services are still working after a change and relied solely on manual testing for this purpose.

Some of these challenges, especially ownership concerns with infrastructure, appear to relate to the adoption of DevOps practices and the movement of

some legacy infrastructure from operations to developers. It has been shown that DevOps has a steep learning curve for both developers and operations (Lwakatare *et al.*, 2019). Increasing the number of different teams reviewing changes appears to reduce the number of defects but also increases the duration of the review process (Dos Santos and Nunes, 2017). In our data, reviews of changes by other teams were often more complex and taxing on the reviewer. Team roadmaps are sometimes not aligned, creating a conflict of interest between reviewing other teams' changes and progressing with the teams' roadmap.

The challenges with end-to-end testing are also well-known in the industry (Wacker, 2015). Evidence suggests that end-to-end tests may not be necessary to prevent faults in distributed systems and that simple unit testing can catch most issues if the correct unit tests are implemented (Yuan *et al.*, 2014). Regardless, services adopting only unit and service testing appear to lack deployment confidence. This may be due to a lack of visibility on test coverage, a lack of a well-established process for correct testing, or a limitation of unit testing. Not everyone in the industry believes in unit testing and instead chooses to adopt service or contract testing (André, 2018). Even Wacker (2015) does mention that some end-to-end testing may be beneficial.

4.2 Identified gaps in the current process

While various challenges were identified during the diagnosis, it's unclear what part of the process could cause them. Hints from the diagnosis results and the analysis of the architecture and process are used as a basis for further exploration of possible gaps or deficiencies.

4.2.1 A complex legacy service is linked to most production incidents

The first hint of a gap comes from the distribution of the incidents in the system. It was found that most of these incidents (more than half) were related to one specific backend service. This backend service was also identified as being the most complex and having the highest rate of change across all the four services examined. Additionally, while the ownership of this service was centralized to one team, many teams were actively making changes to the code in practice. As mentioned, difficulties with reviewing changes by these other teams were expressed in the workshop deliverables. All these findings correspond nicely with the review process, service complexity, and code ownership measures found in the literature.

The evidence that changes are required to this complex legacy service and its development seems sound. However, it's not a new discovery in the teams, and projects were already launched unrelated to this thesis, aiming to

address both the service complexity and collaboration aspects mentioned above. The diagnosis and literature review results suggest these projects are critical from a quality assurance point of view and should be continued. While this challenge is not entirely related to microservices, it's emphasized in distributed development. The need to evaluate the complexity of any solution, alignment of ownership with service dimensioning and the ease of code review are emphasized by this finding.

4.2.2 Data migrations and infrastructure were a source of significant incidents and discussion

The second hint relates to aspects not significantly explored in the analysis of the architecture and processes, namely data migrations and service infrastructure. These aspects were found in the incident data and workshop deliverables. Data migrations were associated with significant incidents. Migrations are performed on large datasets where every single value combination is hardly testable through traditional unit testing frameworks. Some teams also struggle with understanding what the production data looks like.

Similarly, service infrastructure, in some ways, falls outside the day-to-day code development process. It was found that ancient infrastructure that was set up a long time ago was considered problematic. This is most likely due to a lack of knowledge of this infrastructure or, in other words, a lack of ownership. In general, it's assumed that all teams own their service infrastructure, and given that teams are responsible for incident resolution, this is easily justifiable. The link between quality and ownership also adds further weight to this practice. However, as mentioned earlier, software developers may face a steep learning curve when taking ownership of infrastructure, especially if it was previously managed by a separate operations team (Lwakatare *et al.*, 2019).

These issues were also not entirely new and unknown in the case organization. However, there does not seem to be a straightforward solution to how data migrations and infrastructure should be managed in the teams participating in the study. Some work has been done to improve the reliability of migrations and make it easier to verify the results with production data. Similarly, resolving ownership issues for legacy infrastructure has been a point of discussion within the teams. No definitive solutions have been found for moving legacy infrastructure from operations to development ownership, as it's expected to be under a DevOps model. These findings emphasize the need to evaluate the teams' ownership level and knowledge for applied solutions.

4.2.3 Large number of incidents are related to service compatibility

The third and last hint is the number of incidents related directly to service compatibility. The core of this issue is a change in a service being incompatible with another service that is dependent on it. The diagnosis reveals that service compatibility is among the most significant incident categories. Additionally, there was evidence that lack of deployment confidence is precisely related to uncertainty regarding how other services would behave, given a change to their dependency. Finally, a clarifying discussion with one of the engineers confirmed that no exact methodology has been defined for service compatibility testing. Still, manual and end-to-end testing are sometimes used for this purpose.

The deficiencies with manual testing are pretty clear from a simple logical analysis. Manual testing relies on the developer knowing which services are affected and how. Otherwise, they would be forced to perform comprehensive testing of the whole system, which would be very time-consuming and labor-intensive. Manual testing is generally unsuitable for regression testing that should be performed for every change. Performing manual testing this way is considered a microservices anti-pattern (Taibi, Lenarduzzi and Pahl, 2020).

On the other hand, end-to-end testing appears to be suitable for regression testing of the communication between services at first sight. However, the architecture and testing methodologies analysis reveals that end-to-end testing is only performed at the external interface level (i.e., dashboard and public API interface). This provides a clue to a limitation in the testing strategy. Prior literature also emphasizes difficulties with integrated testing methods such as end-to-end testing (Wacker, 2015; André, 2018; Lehvä, Mäkitalo and Mikkonen, 2019).

Generally, testing in distributed microservices development occurs most naturally during the development process of each specific service. This is a consequence of practices such as continuous delivery. As seen from the process analysis, something of this kind is used at the case company. When developers are making a change and deploying it, the context they're considering consists of the steps in the deployment pipeline of that specific service.

Based on the above premises, an analysis of end-to-end testing with the public API interface is shown in Figure 18 (see page 53). In this figure, a backend service developer has triggered a deployment. All steps, including backend service deployment pipeline tests, pass as expected. The exploratory testing executed in the staging environment does not reveal any obvious faults. However, let's assume some service compatibility issues have passed this testing

process. We would then assume that end-to-end tests will catch this issue, as they will.

The issue is demonstrated not with the ability of the end-to-end testing to catch the issue in question but with the feedback loop of the testing methodology. As the public API end-to-end tests are executed every 24 hours, it's possible, in theory, that the time between a fault being introduced and the detection by end-to-end tests is up to 24 hours. In almost all cases, alerts will provide similar detection in just a few minutes. The feedback loop of this end-to-end test setup cannot prevent service compatibility faults in practice.

This problem was chosen as the core problem to be explored in this thesis. This decision was made because a clear problem could be defined, and potential solutions appeared implementable within the timeline of this thesis. After the literature review, prior literature on this topic also hinted at a potential solution and avenues for additional contributions (i.e., more verification of the method in a real case with multiple teams).

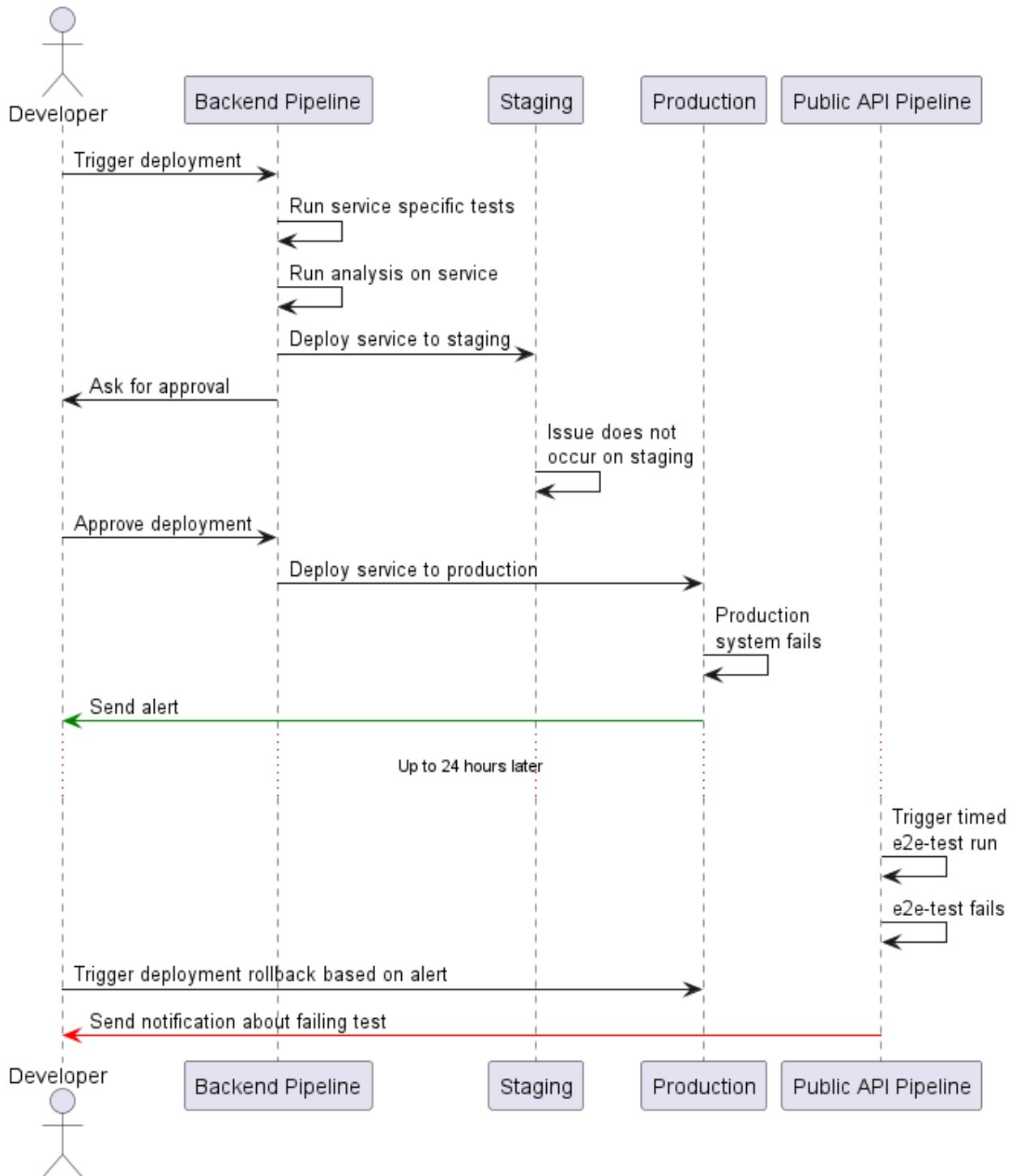


Figure 18. End-to-end testing feedback loop demonstrated with a faulty backend deployment.

4.3 Evaluation of quality assurance strategies

As challenges surrounding service compatibility were chosen as the focus problem for this thesis, this segment presents the evaluation that was the basis for the chosen solution. The relevant action ideas that have been discussed are as follows:

- Encouraging throughout manual testing through pull request template checkboxes.
- Reusing existing end-to-end tests from public API service in backend services.
- Implementing provider contracts for all services.
- Moving all functionality from public API service directly into the backend services.
- Introducing consumer-driven contract testing.

Three ideas introduced above came from a workshop organized with multiple engineers, while two were already found in prior discussions. The prior ideas were manual testing and end-to-end tests in backend services. Both ideas were already largely dismissed before the workshop and, therefore, not included as a part of it. The workshop results are presented in Table 6.

Table 6. Results of a workshop evaluating action ideas.

Action idea	Value	Effort	Impact on others
Implement provider contracts	3/5	3/5	4/5
Move public API functionality	4/5	5/5	5/5
Implement consumer-driven contract testing	5/5	4/5	4/5

Based on the analysis of the process, the following factors could also be considered relevant for any chosen solution:

- **F1:** Any process or tooling must exist in one of the core process stages: local development, pull request review, or deployment pipeline.
- **F2:** Any testing tools must be functional in all core stages.
- **F3:** Testing of a service is only performed in the continuous integration pipeline of that specific service. Testing results from other services are not guaranteed to block deployment.

Encouraging manual testing through pull request checkboxes was one of the first ideas. Based on discussions with the teams, pull request checkboxes were not considered a reliable way to enforce action. Additionally, manual testing is inconsistent with DevOps and is a microservices anti-pattern (Taibi, Lenarduzzi and Pahl, 2020). Manual testing also requires changes to

be bundled so that a full working feature is always in one pull request, such that it can be manually tested. This restricts managing change complexity, another essential factor for reducing faults (Hassan, 2009).

The public API service contains many end-to-end tests against many backend services. Reusing these end-to-end tests was also considered a quick option to combat factor **F3** and give developers of backend services quicker feedback on how their changes affect the public API. This idea was abandoned due to practical implementation issues. Based on the diagnosis results, end-to-end testing places the stability and maintainability of the service at risk. Based on our observations, the more end-to-end tests run, the larger the probability of random failures. There are also warnings from industry experts against the overuse of end-to-end testing (Wacker, 2015), and there is some evidence that more simple testing is already sufficient for catching the majority of faults if properly employed (Yuan *et al.*, 2014). Additionally, it was unclear how the reuse would work in practice. Even though proof-of-concepts were attempted, none of them promised the necessary functionality.

Implementing provider contracts was also considered. Provider contracts already exist in many services. However, some services do not use them. Provider contracts clarify the API of any service and present an opportunity for versioning, which is an essential practice for microservices architecture (Taibi, Lenarduzzi and Pahl, 2020). The main downside of provider contracts has to do with ownership, and ownership is one of the essential factors in reducing faults (Greiler, Herzig and Czerwonka, 2015). Based on the data from diagnosis, teams tend to have only limited knowledge of how other services use their service API. This is natural since the teams develop and deploy their services independently according to the DevOps model. The provider team does not know exactly what data other teams need to implement their features. This fact means that a provider contract lacks information about which parts of the exposed data are used and makes changing the contract difficult. New versions always need to be deployed, and old versions must be supported until all consuming service teams have been contacted and changes have been coordinated.

Another briefly considered idea was moving the public API service functionality directly into the backend services. This was considered because the public API service was a significant consumer involved in many service compatibility-related incidents. The change would have helped with ownership-related issues by bundling backend services and their public API endpoints. However, the effort and scope of this change in the current architectural structure would have been massive (see evaluation in Table 6). Additionally, the services were not designed to provide all the functionality required for the public API. As such, this idea was abandoned, at least in the short term.

Lastly, the introduction of consumer-driven contract testing was considered. This idea quickly got support from the team's engineers. Consumer-driven contract testing promised to avoid many of the issues with end-to-end testing. Further, it allowed running the tests directly in the deployment pipelines of all backend services. Consumer-driven contract testing solutions were found to satisfy all three factors mentioned above and implementing such a setup seemed feasible in a limited time. Additionally, the consumer-driven nature of this testing methodology promises to improve knowledge of how each service's API is used. We hypothesized that this feature would increase service deployment confidence and make API changes faster and less risky. As such, consumer-driven contract testing was chosen as the action to be implemented. For a more detailed explanation of the practice, refer to 2.4.1 in the background and related work section.

4.4 Implementation of consumer-driven contract testing

In this section, we will discuss how consumer-driven contract testing was implemented. The implementation used the Pact framework. This framework is described in more detail in section 2.4.2. A Pact Broker was used to accomplish contract sharing between services. The process of implementation is discussed in more detail in section 3.1.3. In this initial stage, the public API service described in the case architecture description was the primary consumer service. Provider-side testing was implemented in two backend services that provided data to the public API service. All the services involved used a TypeScript or JavaScript-based environment and implemented Pact through the Pact JS library.

4.4.1 Deployment and contract change processes

In this section, we will introduce the big-picture view of the new processes that adopting Pact introduces to the service development and deployment processes. We will also discuss how adopting these processes can resolve the chosen quality assurance challenge described above. First, the consumer and provider share the main stages shown in Figure 19. These stages include the steps from opening a pull request to service deployment. Quality assurance is performed in two stages: manual review and automatic build pipeline.

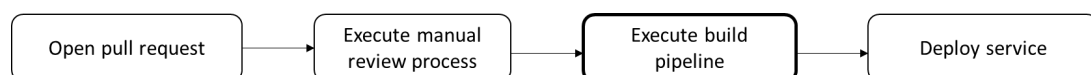


Figure 19. The main four stages are from pull request to deployment.

This section will focus on the “Execute build pipeline” stage. In this stage, we build the service, run tests, and perform static analysis. The zoomed view of this stage is shown in Figure 20. Each stage builds upon the last and can cause the whole pipeline to fail and stop executing. Faster, more isolated tests are executed first to save time on execution and issue discovery in case of failure.

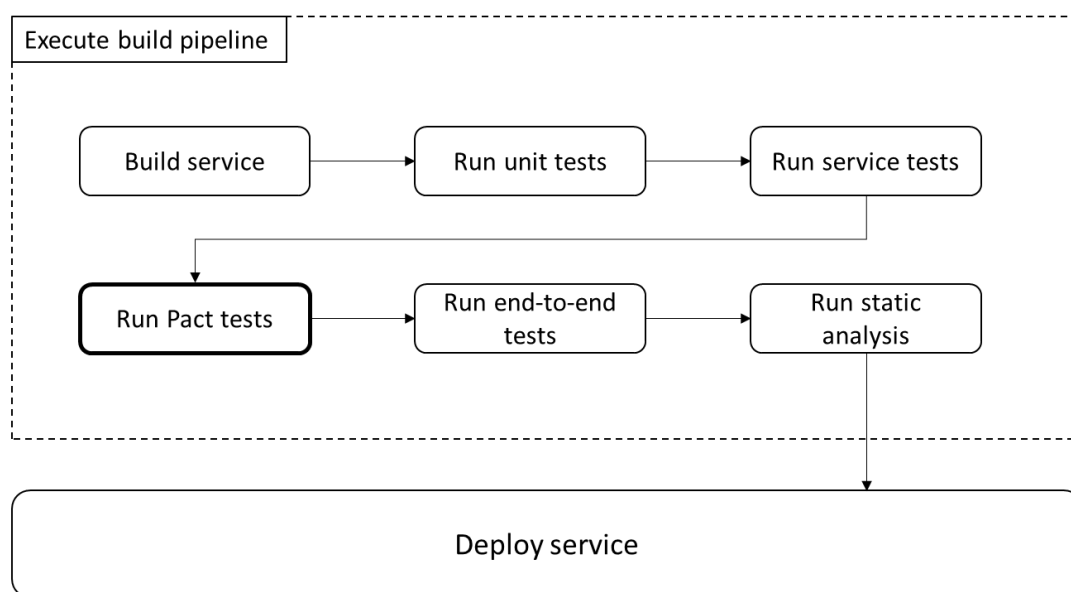


Figure 20. Further breakdown of the build pipeline stage. Assume each stage can fail and exit the pipeline.

In this stage, the only change is including the “Run Pact tests” stage. This stage consists of three steps, shown in Figure 21. First, the tests are executed, and the contract is automatically generated. Then, the contract file is published to the Pact Broker. Lastly, the verification status for the contract is checked. This is where our process differs slightly from the Pact reference process. We do not use webhooks to trigger a verification automatically when the contract is published. If the contract contains no changes, the new version is automatically verified without a need to execute verification on the provider. If the contract changes, this stage will fail. If the stage fails, the verification is manually triggered on the provider to get the verification result, and then the stage is restarted. This process is not entirely smooth and automated, but in our case, contracts rarely change, and webhooks were considered additional complexity for the initial implementation.

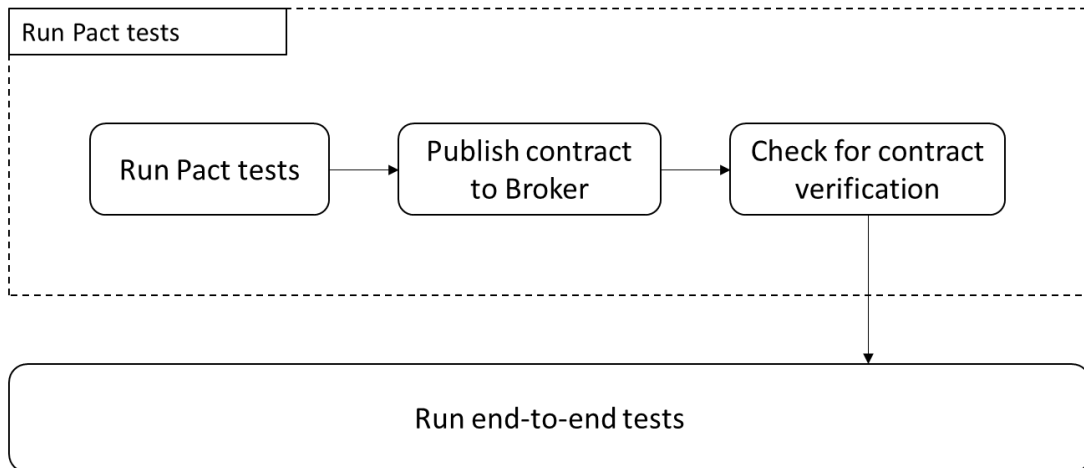


Figure 21. Running Pact tests also include publishing the contract and checking verification results.

The complete sequence of steps across the consumer pipeline, Pact Broker, and provider pipeline are shown in Figure 22. All these steps must be executed for a service version with contract changes to successfully deploy. Time can pass between contract publishing and verification; otherwise, all steps are automated. The benefit of Pact testing is visible in this figure. Both pipelines are decoupled from each other. The dependency is shifted to the Pact Broker service, which is considerably more stable than custom services in active development. However, the services are not entirely decoupled, as the consumer still depends on the provider to publish verification results correctly, and the provider expects the consumer to publish only valid contracts.

When changing the contract on the consumer side, the contract is already published after the pull request is opened. This pull request version can then be verified on the provider side. The work-in-progress and pending pacts features allow new versions of contracts to be published without them causing providers to fail. The new contract only enables Providers' failure after the first successful verification result. This ensures that providers only fail when changed from successful to unsuccessful verification.

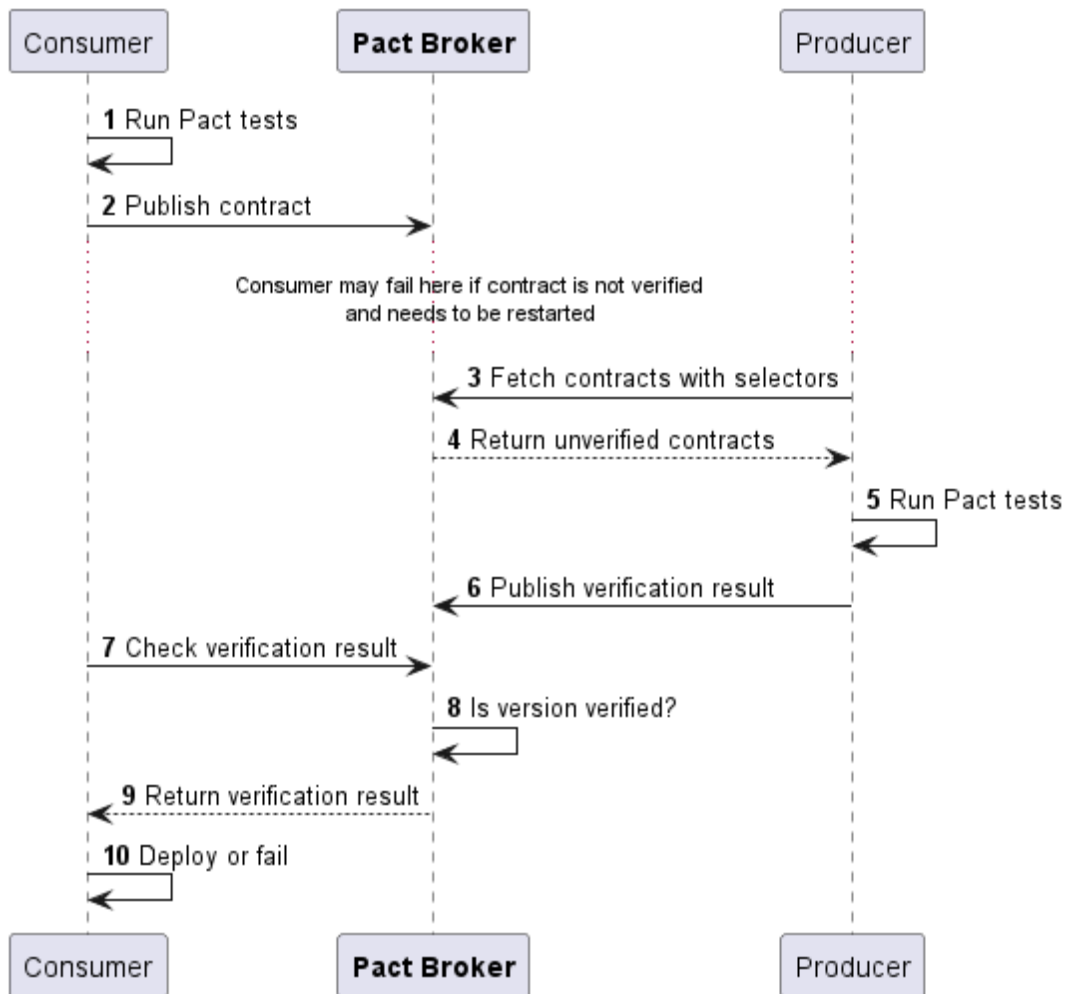


Figure 22. Sequence required for successful deployment. Consumers and providers refer to building pipelines of the services.

Figure 22 shows that consumer and provider feedback is always instant and included in the build pipelines. Even though there is some minor coupling between the services, it's reduced by the Pact Broker while still accomplishing verification of the integration between the services. The automation has some gaps, but solutions exist in the Pact framework if fully used.

4.4.2 Implementation of Pact Broker

Pact Broker was implemented as a self-hosted service using the open-source Pact Broker Docker container image. The image is called `pactfoundation/pact-broker` and is hosted on the public Docker Hub. At the case company, docker containers are hosted on a managed cloud Kubernetes cluster. We also hosted the Pact Broker on Kubernetes with a small node pool using auto-scaling, auto-repair, and auto-upgrade. The configuration of the deployment was done using Helm. The service was exposed only internally

using the appropriate DNS and security policies. Both staging and production versions of the Pact Broker were created. Initial testing was done on the staging version. The resource limits for the deployment were relatively modest at the low initial traffic volumes. Only one replica was used for the production version of the Broker.

Pact Broker requires an external database to operate efficiently. Specifically, Pact Broker uses a PostgreSQL database. A small managed instance of PostgreSQL 15 was created for the Broker. The Pact Broker container was connected to the database through an additional proxy container intended for this purpose. Pact Broker can use a built-in SQLite database for testing purposes. This was used when running the container locally on developer machines. The database is the primary infrastructure requirement for running a self-hosted Pact Broker. Testing the Pact Broker locally with the built-in database is relatively straightforward.

Additionally, Pact Broker provides an automatic database clean-up cron job. This cron job was enabled using an environment variable. The environment variables are shown in Figure 23. Initially, the cron job was scheduled to run at night and clean overwritten contracts over 30 days old. It was also set up to clean contracts that are more than 90 days old unless the contract is the latest contract for the main branch of the repository. This functionality maintains the Pact Broker's performance over time, even as large amounts of contract data are gathered in the database. The cleaning thresholds mentioned above were selected randomly and were not particularly tested in any way. As of now, no performance issues have been observed with the Broker.

```
env:
- name: PACT_BROKER_DATABASE_CLEAN_ENABLED
  value: "true"
- name: PACT_BROKER_DATABASE_CLEAN_CRON_SCHEDULE
  value: "15 2 * * *"
- name: PACT_BROKER_DATABASE_CLEAN_OVERWRITTEN_DATA_MAX_AGE
  value: "30"
- name: PACT_BROKER_DATABASE_CLEAN_KEEP_VERSIONS_SELECTORS
  value: "[{\"branch\": \"master\", \"latest\": true},
  {\"branch\": \"main\", \"latest\": true}, { \"max_age\":
  90 }]"
```

Figure 23. The database cleans up environment variables.

4.4.3 Implementation of consumer-side testing

In this section, we will discuss the implementation of consumer-side testing. The focus will be on the public API service, which served as the primary consumer in the initial implementation initiative. This service already had existing units, services, and end-to-end testing. Much of the existing test setup was also reused for Pact contract tests. This confirms statements by André (2018) that service testing setup can be reused for contract testing. Testing was primarily focused on the internal backend API calls generated when a request is made to the external public API interfaces. Pact consumer interactions must be defined with each backend service's expected request/response pairs. Next, we will walk through some example tests to illustrate our implementation.

Given an HTTP GET endpoint on the public API service, we use supertest to set up and call a mock version of the endpoint. We have a function for creating the mock server and making the endpoint request. The setup for executing the tests differs somewhat between Pact contract testing and simple service API testing with supertest. Even though we use Jest as our test runner, we did not opt to use the jest-pact library. This decision was made because the opinionated test structure of the jest-pact did not play well with our existing test structure. Instead, we used the Pact V3MockServer and executed the tests manually inside our Jest describe blocks.

The tests execute approximately the following steps:

1. Create the PactV3 mock provider for the consumer/provider pair.
2. Import the provider into the tests and set up the necessary interactions.
3. Execute the tests using the provider.executeTest() function.
4. In the function passed to executeTest, set up the mock server and perform the request using supertest.

The first step creates the mock server test double of the provider API. When creating the mock provider, the name of the consumer and provider are given. These will be used to name the outputted contract file so that it can be found and executed in the correct provider-side tests. An example code is shown in Figure 24.

```

import { PactV3 } from '@pact-foundation/pact';
export const backend1ProviderMock = new PactV3({
  consumer: 'public-api-service',
  provider: 'backend1',
  logLevel: 'error',
});

```

Figure 24. A provider mock is created with the definition of the consumer/provider pair.

The second step is the core of the whole Pact testing process. Interactions need to be set up so that the provider mock knows which requests to accept and which responses to mock. We decided to define these interactions outside the actual test files. This was done to clarify precisely what interactions we expect to occur. The interactions and the related provider states are also relevant for backend teams; spreading this information all over the code would make it harder to grasp the dependencies between the services. In the actual tests, we used a before-all or before-each setup step to create the interactions in the mock provider and then execute the tests. An example of this step is shown in Figure 25.

```

import { backend1ProviderMock as provider } from '...';
describe('when receiving a GET request', () => {
  beforeEach(async () => {
    getInteraction(
      provider,
      ServiceState.RESOURCE_EXISTS,
      InteractionType.BACKEND_GET_REQUEST,
    );
    return provider.executeTest(getTestFunction());
  });
});

```

Figure 25. Setting up an interaction in Jest tests and executing tests against provider mock.

In Figure 25, the most relevant parts are implementing the get interaction function and get test function. We used different patterns for setting up the interactions depending on whether the interaction is a get interaction, an update/create interaction or an error interaction. So far, it has been common to test successful requests and requests leading to a validation error. This pattern was also noted in an existing case study (Lehvä, Mäkitalo and Mikkonen, 2019). The goal is to cover cases that could break consumer service if the response unexpectedly changes.

First, looking at the above `getInteraction` case, the request is usually static, meaning it's always the same. There is no request body; the only aspect expected to change is the resource identifiers in the path. The central aspect that varies between interactions is the response body. For this reason, we set up this type of interaction with a hardcoded expected request and vary the mock response based on the expected provider state. This idea is illustrated in Figure 26. The response is selected based on state with simple switch case selection logic.

```
export const getInteraction = (
  provider: PactV3,
  state: ServiceState,
  interactionType: InteractionType
) =>
  provider
    .given(state)
    .uponReceiving(interactionType)
    .withRequest(getRequest())
    .willRespondWith(getResponseByState(state));
```

Figure 26. Function for setting up an interaction on the mock provider.

Regarding update requests, the process differs slightly from the one shown in Figure 26. In this case, we generally do not have a static request body but provide the body from the tests to test scenarios with different types of updates. We decided to make liberal use of Pact matchers for the request bodies. This is because “Pact follows Postel’s law” (Pact Foundation, 2023a). This can be summarized with “Be conservative in what you send” and “Be liberal in what you accept.” The idea is to allow changes to fields that do not break the consumer and only assert what is necessary to maintain correct functionality. This frees up the provider to make changes as freely as possible whenever the change is not considered breaking for the consumer. Generally, we only include fields we need in the mock responses, and the fields where the exact value does not matter are replaced with matchers. An example of a response definition for the provider mock is shown in Figure 27.

```
const getResponse = (additionalResponseFields: object) =>
({
  status: 200,
  headers: commonResponseHeaders,
  body: {
    ...additionalResponseFields,
    amount: MatchersV3.integer(),
    id: MatchersV3.regex(/.../i, '...'),
  }
});
```

Figure 27. Response generation function that uses matchers where possible.

Lastly, we use a similar approach for error responses. With errors, we're primarily interested in the error status code (e.g., status 400 Bad Request), and we do not need to assert the exact error message. Due to legacy architecture decisions, some errors are not distinguishable from each other without considering the error message. We use the regex matcher in these cases, as illustrated in Figure 27. This matches how we currently differentiate between these errors in our code. It still allows the error message to change while ensuring it's still distinguishable in the public API service.

After the interactions have been set up following Postel's law, we run the tests, ensuring that the public API is compatible with our expected interaction. This is one of the benefits of consumer-driven contract testing. All interactions in the contract are programmatically ensured to be valid for the given consumer, assuming the consumer tests themselves are written correctly. The test function we feed to the provider `executeTest` routine is shown in Figure 28.

```

Let server: Express;

const getTestFunction = () => async (mockserver:
V3MockServer) => {
  await setupMockServer(mockserver);
  return doRequest();
};

const setupMockServer = async (mockserver V3MockServer) =>
{
  const backend1Client = createBackend1Client(
    mockLogger(),
    mockserver.url,
    httpClient.httpClientTimeoutMs
  );
  const endpoint = {
    getEndpoint: v1GetEndpoint(backend1Client);
  };
  ({ server }) = await mockServer(endpoint);
};

const doRequest = async () => {
  result = await supertest(server)
    .get(path)
    .set('content-type', 'application/json')
    .send();
};

```

Figure 28. Functions to set up mock service against Pact provider mock and execute test requests using supertest.

The setup shown in Figure 28 is for a GET endpoint. The code uses a functional composition style where the code for calling backends is provided into the service code as parameters. This allows us to provide the backend client with the path to the Pact mock provider during client creation. This will allow us to test the actual backend calling code while making real test calls against the Pact mock server. This backend client function is then fed into the actual public API endpoint handler code we're testing to create the full mock server. Supertest is then used to call this endpoint to execute the test scenario.

What is described above forms the fundamentals of the consumer side test scenarios. Additional aspects also exist, such as the assertions ensuring the consumer executes the scenario correctly given the mock response from the Pact mock server. We chose this testing approach for a couple of reasons. First, we already had the necessary setup to test the service in this manner. Second, by using Pact in actual service tests, we ensure that the interactions we include in the contract are the same as what the service performs in

production. The ability to ensure that the contract is always up-to-date and aligned with production behavior is a significant claimed benefit of consumer-driven contract testing. Lastly, we summarize the testing principles that we followed:

1. Always follow Postel's law using matchers and omitting unnecessary fields from assertions.
2. Use a structured approach for provider states and interactions to ensure visibility of the dependencies between services.
3. Create patterns for common testing scenarios and use helper functions to cover request and response generation.

Following these rules, providers maintain maximum freedom to change their API, and understanding can be maintained as to which states need to be implemented. Additionally, the patterns described above and helper functions progressively simplify the interaction testing process.

4.4.4 Implementation of provider-side testing

Provider-side testing was implemented in two backend services. This implementation was much simpler than the consumer side, as mentioned by previous research (Lehvä, Mäkitalo and Mikkonen, 2019). There are mainly two steps that need to be performed on the provider side:

1. The verifier needs to be set up with the correct settings.
2. The state handlers referenced on the consumer side must be implemented.

Out of these, the second step requires the most work. The settings for the verifier will be explored in 4.4.5. There is almost nothing Pact-specific when it comes to the state handler implementation. Our process was as follows:

1. Copy the list of expected provider service states from the consumer side.
2. Create an object with the implementation for each state in the list.
3. Provide the list of state handlers to the verifier.

An example is shown in Figure 29. The state handlers are implemented in the provider service tests in this code. The verifier is then configured and executed using the `verifyProvider` function. The provider to be tested is started before these tests are executed and available at port `httpInternalPort`. The Pact verifier will request the `providerBaseUrl` address.

```

Import { Verifier } from '@pact-foundation/pact';

// Same list is on the consumer side
enum ServiceState {
  RESOURCE_EXISTS = 'a resource exists',
  RESOURCE_DOES_NOT_EXIST = 'a resource does not exist'
}

Let server; HttpServer;

describe('Public API to backend1 integration', () => {
  beforeAll(async () => {
    server = createServer();
    server.start();
  });

  afterAll(() => {
    server.stop();
  });

  it('satisfies the contract', () => {
    return new Verifier({
      ...pactConfig,
      providerBaseUrl: `http://localhost:${httpInternalPort}`,
      stateHandlers: {
        [ServiceState.RESOURCE_EXISTS]: async () => {
          const resource = generateResource(...);
          await resource.save();
        },
        [ServiceState.RESOURCE_DOES_NOT_EXIST]: async () => {
          await clearDatabase();
        }
      }
    })
    .verifyProvider();
  });
});

```

Figure 29. Example code with state handler and verifier setup.

This code alone captures most of the Pact-specific code that needs to be added to the provider. In the case of the selected backend services, existing methods existed for mocking other services and managing test data. All the mocking and test setup code does not differ between traditional service tests and Pact consumer-driven contract tests. After the provider and state handlers have been implemented, the main concern is connecting the provider to the Pact Broker so it can fetch the contracts to test against. Additionally, the tests need to be integrated into the continuous integration pipelines.

4.4.5 Implementation in continuous integration pipelines

This section will discuss the testing setup in continuous integration pipelines and the integration to Pact Broker. Pact provides various tools to ensure that changes breaking the integration cannot be deployed. The CI setup has two goals: ensuring that the provider cannot make changes that break consumers and that consumers cannot change the contract before the provider is ready. In essence, both changes help teams communicate and coordinate service changes without directly coupling them.

The consumer-side integration is relatively simple. First, the tests are run to generate the contract file. Then, the `pact-broker` command line tool is used to publish the contract to the Pact Broker. When using Pact JS, this tool is included in the package and can be found in the `node_modules` folder. Figure 30 shows the test run and publish commands used in continuous integration. Contracts were chosen to be versioned by Git commit hash and were tagged with the branch name. The branch name is essential for differentiating work-in-progress contracts from production (main) contracts. The versioning schema could be something else, but as Git commit hashes clearly and uniquely correlate with a change to the service, they were considered appropriate for this purpose.

```
npm run test # Contract gets generated
./node_modules/.bin/pact-broker publish pacts -consumer-
app-version="${REVISION}" -branch="${CHANGE_BRANCH}" -bro-
ker-base-url="http://pact-broker"
```

Figure 30. Pact Broker command to publish contracts.

After this, the contract is now available on the Pact Broker but has not yet been verified. In other words, we do not know whether the provider satisfies the contract. This could be checked before publishing the contract, but ideally, contracts will be verified automatically in the continuous integration flow. Pact Broker supports webhooks that automatically trigger verification when a new contract is published. We did not use this as it would have added complexity to the setup. Pact Broker will automatically verify unchanged contracts. As our contracts rarely change, we did not see the benefit of automatic verification trigger with webhooks significant at this stage. Next, we will discuss integration into the provider's continuous integration pipeline.

On the provider side, the Pact tests were included in the test run command of the services. The critical ingredient is the Pact verifier configuration. We enabled two consumer version selectors for all providers: the matching branch selector and the main branch selector. Each provider will verify

against central branch contracts and contracts with a branch name matching the current branch. These selectors are why contracts were tagged with a branch name on the consumer side. First, the main branch selector is the most important. This selector will ensure that all changes that can be merged and deployed are compatible with the latest main version of the consumer service. The matching branch selector is merely a convenience when introducing contract changes. We also enabled pending Pacts and work-in-progress Pacts features to help with contract changing flow. This flow will be introduced later. Lastly, we enable the publish verification results setting when running tests on continuous integration pipelines to save the verification status on the Pact Broker. The main settings are shown in Figure 31.

```
return new Verifier({
  providerBaseUrl,
  stateHandlers,
  consumerVersionSelectors: [
    {
      mainBranch: true
    },
    {
      matchingBranch: true
    }
  ],
  enablePending: true,
  includeWipPactsSince: '2023-12-1',
  pactBrokerUrl,
  providerVersionBranch,
  providerVersion,
  provider,
  publishVerificationResult: process.env.CI === 'true',
});
```

Figure 31. Most important Pact verifier settings. Assume undefined variables are defined elsewhere.

With this setup, we can prevent non-backward compatible changes on the provider. However, the system is one-sided and unfair to the provider service as the consumers can arbitrarily change the contract without first communicating and adding support for the needed functionality. To prevent this, Pact includes a tool that checks whether the current contract version for pull request changes has been verified. If the contract has not been verified, the build will fail and prevent the merge/deployment of the changes. We included this “can-i-deploy” tool in our consumer service pipeline. The configuration is simple: all we need to do is provide the name and version of the consumer, and Pact Broker will do the rest. The contracts with all providers will be checked to ensure they have been verified.

```
./node_modules/.bin/pact-broker can-i-deploy -a, --partic-  
ipant=public-api-service -version ${REVISION} -b, --broker-  
base-url=http://pact-broker
```

Figure 32. The can I deploy command with parameters we used on our consumer service pipeline.

With this inclusion, we ensure that consumers and providers are compatible. This process is not without flaws. First, it creates a soft coupling between the services. If the pipeline for one service is broken or misconfigured, it can affect deployments in the other service. Second, this system is not bulletproof. There are some edge cases where incompatible changes could end up in production. This can occur because there is a delay between running the checks and actual deployment. If changes to both services co-occur, they can end up in a race condition where both sides pass against the old version but never end up testing against each other. Pact broker includes tools to resolve this problem by recording and verifying against final deployments. However, in our case, we generally do not have multiple versions deployed in the same environment at once, so the risk was considered too small for the added complexity.

4.5 Evaluation of the implementation

In this section, we explain our findings from evaluating the action outcome. First, we will discuss the findings based on the metrics defined and then the findings from observations and interviews with participant team members. This segment has many limitations, such as the study length of less than six months and a small number of participants with contract testing experience.

Firstly, we did not observe any relevant incidents in the participating services after the action was implemented. This fact is unlikely to be solely related to contract testing as the study period was short, and other changes that affected the service health were also made. For example, refactoring was done to the complex service mentioned in section 4.1. There were also holiday periods, such as Christmas, with a lower-than-usual rate of change. More data is needed to evaluate the improvement in practice. Earlier literature notes that Pact can catch service integration-related issues (Lehvä, Mäkitalo and Mikkonen, 2019).

Second, the Pact test runner appeared stable during the observation period. There were no test failures with unidentified or “flaky” causes. This stability contrasts with end-to-end tests, where we saw 11 flaky failures during an unstable one-month period. However, we did observe some Pact failures. The

first case was caused by a change in how the tests were executed. An environmental variable required by Pact Broker was misconfigured on a backend service. The misconfiguration also affected the consumer service by publishing invalid verification results to the Pact Broker and failing the consumer service pipeline “can-i-deploy” check. This issue illustrates that while there is no direct coupling between the service deployment pipelines, an issue in the setup of one service can still cause another service to fail deployment without a valid reason. A mature integration of Pact Broker into deployment pipelines is needed to avoid such issues and to benefit from the isolation between services. Valid reasons, such as a lack of verification results for a new contract, caused other failures. We also observed some cases where the tests failed due to incorrectly implemented tests. One case related to test assertions that were too strict was fixed with the better use of Pact test matchers. We did not track the solution’s stability on the developer’s local machines. Interview participants mentioned some cases of test timeout, but logs demonstrating this could not be found. The runtime of the Pact tests was similar to other testing methods, and Pact did not seem to introduce significant runtime overhead. In one of the backend services, Pact tests executed in less than a minute, while service tests took up to 5 minutes. However, there are more service tests than Pact tests, so the numbers are not entirely comparable. The overall pipeline runtime on the consumer also did not significantly change, with the test stage taking about 3 minutes before and after the change.

As for the developer experience, some issues were discovered. The comments from developers who had more or less time with the tool differed significantly. There was a consensus that Pact presents a steep learning curve, even for developers otherwise experienced with automatic testing. The testing process contains more steps than other testing methods, and new tooling needs to be introduced. The process contains unfamiliar steps and concepts, even for experienced developers. For example, Pact includes a feature to verify new contract versions from development branches, which requires the branches to have a matching branch name. Developers must know this detail to use the tooling correctly, and it was expressed that this needs to be clarified for a new developer unfamiliar with the tool. In general, learning the process and tool was difficult and painful for the developers who had not extensively read the Pact documentation and participated in the initial implementation. The text output of the test runner was also sometimes unnecessarily verbose. However, the additional testing did increase subjective developer confidence. At least one developer reported finding a bug with the method. We do not have data to compare the increase in confidence with other testing methods, and the lack of experience with the tool still made it hard to be confident in the test results.

Under a natural development environment, minimal changes were needed to the tests and the test setup. After the initial setup, the maintenance burden of Pact tests may be low. The author was heavily involved with the initial implementation, and the amount of work required to put the tool into use was quite significant. This work came from the setup of Pact Broker and the various custom changes required to build pipelines of the services. The Pact documentation does acknowledge this by describing maturity levels for gradual adoption. Significant communication effort and training are also required when involving two or more teams in the initiative. It was expressed in interviews and prior literature that most of the complexity of the testing process lives in the consumer service. Participants reviewing changes in the backend services found the tests easy to understand. However, consumer service was more complex and more challenging to understand. The effect is partly due to the lower familiarity of the interview participants with the consumer service, but based on observations, the overall sentiment is valid. We assume the approach where consumer service owners were responsible for pushing the initiative was appropriate.

When the concept of contract testing and Pact were first introduced, it was met with general acceptance and enthusiasm. It was also noted during interviews that the underlying concept is clear and easy to understand. During one discussion, we also found that there had been earlier attempts to take the tool into use. This previous attempt was also met with a similar positive response. However, we heard that the earlier initiative died out, probably due to a lack of ownership and shifting of priority to other initiatives. This finding implies that the value perceived from contract testing with Pact compared to other testing methods may not be enough to overcome the hurdle of training and adoption. The author did not directly observe the earlier initiative in any way.

In summary, even after some experience with the tool, participants still viewed it positively and considered it a valuable asset in their toolbox. The tooling was observed to be stable and required little maintenance after adoption. However, the learning curve for the tool was considered steep. The interview participants desired better documentation of the contract testing process. In general, even though presentations were held about the concept and the tool was discussed to some extent, the training given was insufficient for comfort while using the tool, and the processes remained too unclear. Lack of training is partially a limitation of the action-taking process, not the chosen tooling or method. The result across all evaluated dimensions is summarized in Table 7.

Table 7. Summary of evaluation results.

Test effectiveness	Found to be effective for catching service integration bugs in theory through defect injection (Lehvä, Mäkitalo and Mikkonen, 2019). Unverified in real life setting.
Test stability	No unexplained or flaky test failures detected.
Test runtime	No significant runtime overhead compared to other methods. Much faster than end-to-end tests.
Test maintenance	Low maintenance burden during study period (low number of changes required under normal conditions).
Learning process	Steep learning curve due to the complexity of the testing process and number of new tools introduced.
Deployment confidence	Improved confidence but unclear how it compares to other testing methods. Low familiarity with the method reduced confidence.
Development speed	Slowed down development but effect is partially caused by low familiarity with the tool and process.
Ease of adoption	High barrier of adoption due to steep learning curve and number of tools and processes required to automatize and simplify use.

5 Conclusions

In this section, we will answer the research questions, address the study's limitations, and present avenues for future research. The study had three research questions in the context of quality assurance. We specifically focused on unique aspects of distributed software development and microservices architecture.

There was no clear winner for the most significant quality assurance challenge in the context explored (RQ1). Some challenges were specific to distributed development, while others were universal to all software engineering. However, distributed development and microservices contribute to each challenge uniquely.

One highly complex service was found to be involved with the majority of incidents. Code complexity increases developer cognitive load and contributes to the number of defects in any software system. In a microservices architecture, however, service dimensioning presents additional challenges as both the size of individual services and the number of services in the system increase complexity. Distributed development also presents scenarios where poorly dimensioned services are simultaneously worked on by many teams and knowledge of closely interacting elements is distributed to distant teams that do not have a close communication channel. Data showed that when multiple teams with different contexts work on the same service, it is difficult for the service owners to understand the whole. The same issue occurs with legacy code authored by engineers who have already left the company. The issue would be mitigated when logically and organizationally separate parts of the system are separated and only communicate through well-defined interfaces that decouple the systems and hide internal complexity for other teams.

The challenge with data migrations and service infrastructure is perhaps most detached from the main context of distributed development with microservices. However, even here, these practices contribute to related practices that are often bundled together to achieve efficiency. For example, adopting DevOps is often essential for microservices to achieve independent deployability and flexibility. Through DevOps, teams gain ownership of their infrastructure, including databases. The steep learning curve of DevOps can make teams less capable of dealing efficiently with issues caused by their service infrastructure. This learning curve is the initial cost required for efficiency with microservices that is expected to pay off as developers acquire mature operations skills. Frequent opportunities for acquiring these skills and working with infrastructure should be provided.

Lastly, in microservices, complexity moves from inside services to their communication patterns. This movement increases faults caused by incompatible services. Active communication between teams is essential to maintain compatibility even when services are developed and deployed independently. However, in distributed software development, teams can reside in different time zones. There is a significant time lag in asynchronous communication, and synchronous communication becomes rare as working hours have little to no overlap. Even when teams are geographically close, there may be no organizational mechanisms to facilitate active communication. Ideally, we would have a tool that automatically informs us when communication is necessary to maintain maximum team independence. Such tools and processes were the core aspects explored in this thesis through research questions RQ2 and RQ3.

We aimed to discover processes and tools to reduce service incompatibility issues while maintaining a good developer experience and velocity. While changes to service structures were considered potential solutions, the main focus was testing methodologies that work with any service structure. Manual and end-to-end testing were already used in the organization, but both included considerable downsides in testing efficiency and developer experience. As such, a new method, consumer-driven contract testing, was explored. From prior literature, we found that consumer-driven contract testing can reduce service-to-service interaction faults (Lehvä, Mäkitalo and Mikkonen, 2019). Through analysis of the process, we concluded that the feedback loop of consumer-driven contract testing with Pact has the potential to block broken integrations from deploying. We see this as an improvement compared to end-to-end testing, where defects are often detected after deployment. In the prior literature, defect injection has shown that Pact can detect defects in integrations. However, the current study period was too short, and the environment was not sufficiently controlled to determine the effect in an organic development environment with real bugs.

The primary evaluation results related to the developer experience of consumer-driven contract testing with Pact. We conclude that it improves upon other methods, such as end-to-end testing in developer experience through increased stability, reduced runtime, and better problem isolation, leading to earlier bug discovery. However, we also found that this method has a steep learning curve and high implementation cost. While consumer-driven contract testing has existed for many years (Robinson, 2006), we found that developers are still relatively unfamiliar with the process in practice. This contrasts with unit testing, which is so widely adopted that all developers at the case organization have deep knowledge of how it is conducted and how the toolset works. The finding may also explain why contract testing has not fully penetrated the industry despite apparent benefits.

While this thesis only consists of one action research cycle, if we could execute a second cycle, it would likely be focused on clarifying the contract testing process in the organization and improving the associated training and documentation. We suggest that any organization adopting contract testing should focus on the process and training over the tooling. Contract testing should be placed in the team's overall testing strategy. We believe the practice is most beneficial when used to move away from integrated testing. The benefits of contract testing appear when presented in contrast to the downsides of integrated testing. Unit and service tests should still be used for testing the primary logic inside services. This type of testing is equally essential, as confirmed by the distribution of incidents surveyed in this thesis.

5.1.1 Limitations

The results of the thesis are affected by some limitations. First of all, as action research methodology was used, the author of the thesis was heavily involved with all stages of the study. Some of the effects found in the study may be artifacts of how contract testing was implemented and not a general limitation of the concept or tooling. Similarly, the thesis author did all the analysis and data collection without investigator triangulation. The relationship between the author and the participants may have influenced the interview results, especially as the author was involved with the action implementation.

Second, the study was done in the context of a single organization. The focus on a single organization limits the generalizability of the results. We also did not control other changes occurring in the organization in any way, making it hard to isolate the impact of contract testing. The study period was limited to six months, split across the different stages. It was clear from the interviews that participants felt they needed more time to get familiar with the new changes. The lack of time affected the evaluation results and limited the number of interview candidates. The shortness of the period also reduced the chance of observing organic test failures or incidents. Normally action research would be conducted in multiple iterations but in this thesis we had only one iteration.

5.1.2 Further research

We believe additional research that addresses the above limitations could be beneficial. In our literature review, we found only one academic study focusing on consumer-driven contract testing with Pact. While various industry articles exist on the topic, based on our findings the academic research is limited. Studying consumer-driven contract testing with Pact using a longer time frame and more participants could lead to different results as participants learn to use the method and tooling. We also believe a long-term study of the

effectiveness in a natural environment with contract testing as a part of a specific microservices testing strategy could help clarify where it fits in with other testing methodologies. For example, it's currently unclear if it's possible to completely replace end-to-end testing with contract testing. A study focusing on methods for reducing the steepness of the learning curve could also be beneficial. Lastly, surveying how commonly the technique is used in the industry overall could help when making the decision to adopt the technique in new organizations.

References

Aghamohammadi, A., Mirian-Hosseiniabadi, S.-H. and Jalali, S. (2021) ‘Statement frequency coverage: A code coverage criterion for assessing test suite effectiveness’, *Information and Software Technology*, 129, p. 106426. Available at: <https://doi.org/10.1016/j.infsof.2020.106426>.

André, S. (2018) *Testing of Microservices, Spotify Engineering*. Available at: <https://engineering.atspotify.com/2018/01/testing-of-microservices/> (Accessed: 15 October 2023).

Baresi, L. and Garriga, M. (2020) ‘Microservices: The Evolution and Extinction of Web Services?’, in A. Bucchiarone et al. (eds) *Microservices*. Cham: Springer International Publishing, pp. 3–28. Available at: https://doi.org/10.1007/978-3-030-31646-4_1.

Bird, C. et al. (2011) ‘Don’t touch my code!: examining the effects of ownership on software quality’, in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering. ESEC/FSE’11: Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Szeged Hungary: ACM, pp. 4–14. Available at: <https://doi.org/10.1145/2025113.2025119>.

Clemson, T. (2014) *Testing Strategies in a Microservice Architecture, martinowler.com*. Available at: <https://martinfowler.com/articles/microservice-testing/> (Accessed: 11 December 2023).

Cohn, M. (2010) *Succeeding with agile: software development using Scrum*. Pearson Education.

Crispin, L. and Gregory, J. (2009) *Agile testing: a practical guide for testers and agile teams*. Upper Saddle River, NJ: Addison-Wesley (The Addison-Wesley signature series).

Dos Santos, E.W. and Nunes, I. (2017) ‘Investigating the Effectiveness of Peer Code Review in Distributed Software Development’, in *Proceedings of the XXXI Brazilian Symposium on Software Engineering. SBES’17: 31st Brazilian Symposium on Software Engineering*, Fortaleza CE Brazil: ACM, pp. 84–93. Available at: <https://doi.org/10.1145/3131151.3131161>.

Foucault, M., Falleri, J.-R. and Blanc, X. (2014) ‘Code ownership in open-source software’, in *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering. EASE ’14: 18th International Conference on Evaluation and Assessment in Software Engineering*, London England United Kingdom: ACM, pp. 1–9. Available at: <https://doi.org/10.1145/2601248.2601283>.

Greiler, M., Herzig, K. and Czerwonka, J. (2015) 'Code Ownership and Software Quality: A Replication Study', in *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories. 2015 IEEE/ACM 12th Working Conference on Mining Software Repositories (MSR)*, Florence, Italy: IEEE, pp. 2–12. Available at: <https://doi.org/10.1109/MSR.2015.8>.

Hassan, A.E. (2009) 'Predicting faults using the complexity of code changes', in *2009 IEEE 31st International Conference on Software Engineering. 2009 IEEE 31st International Conference on Software Engineering*, Vancouver, BC, Canada: IEEE, pp. 78–88. Available at: <https://doi.org/10.1109/ICSE.2009.5070510>.

Inozemtseva, L. and Holmes, R. (2014) 'Coverage is not strongly correlated with test suite effectiveness', in *Proceedings of the 36th International Conference on Software Engineering. ICSE '14: 36th International Conference on Software Engineering*, Hyderabad India: ACM, pp. 435–445. Available at: <https://doi.org/10.1145/2568225.2568271>.

Kauppinen, M. (2005) *Introducing requirements engineering into product development : towards systematic user requirements definition*. Helsinki University of Technology. Available at: <https://aaltodoc.aalto.fi:443/handle/123456789/2625> (Accessed: 15 October 2023).

Kochhar, P.S., Thung, F. and Lo, D. (2015) 'Code coverage and test suite effectiveness: Empirical study with real bugs in large systems', in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER). 2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pp. 560–564. Available at: <https://doi.org/10.1109/SANER.2015.7081877>.

Lazar, J. (2017) *Research methods in human computer interaction*. 2nd edition. Cambridge, MA: Elsevier.

Lehvä, J., Mäkitalo, N. and Mikkonen, T. (2019) 'Consumer-Driven Contract Tests for Microservices: A Case Study', in X. Franch, T. Männistö, and S. Martínez-Fernández (eds) *Product-Focused Software Process Improvement*. Cham: Springer International Publishing (Lecture Notes in Computer Science), pp. 497–512. Available at: https://doi.org/10.1007/978-3-030-35333-9_35.

Lunney, J. and Lueder, S. (2017) *Google - Site Reliability Engineering*. Available at: <https://sre.google/sre-book/postmortem-culture/> (Accessed: 17 October 2023).

Lwakatare, L.E. *et al.* (2019) 'DevOps in practice: A multiple case study of five companies', *Information and Software Technology*, 114, pp. 217–230. Available at: <https://doi.org/10.1016/j.infsof.2019.06.010>.

McIntosh, S. *et al.* (2014) ‘The impact of code review coverage and code review participation on software quality: a case study of the qt, VTK, and ITK projects’, in *Proceedings of the 11th Working Conference on Mining Software Repositories. ICSE ’14: 36th International Conference on Software Engineering*, Hyderabad India: ACM, pp. 192–201. Available at: <https://doi.org/10.1145/2597073.2597076>.

Newman, S. (2021) *Building microservices: designing fine-grained systems*. Second Edition. Beijing: O’Reilly Media.

Pact Foundation (2023a) *How Pact Works*, docs.pact.io. Available at: https://docs.pact.io/getting_started/how_pact_works (Accessed: 7 December 2023).

Pact Foundation (2023b) *Pact Broker*, docs.pact.io. Available at: https://docs.pact.io/pact_broker (Accessed: 7 December 2023).

Parnas, D.L. (1971) ‘Information distribution aspects of design methodology’.

Robinson, I. (2006) *Consumer-Driven Contracts: A Service Evolution Pattern*, martinfowler.com. Available at: <https://martinfowler.com/articles/consumerDrivenContracts.html> (Accessed: 11 December 2023).

Schulmeyer, G.G. (ed.) (2008) *Handbook of software quality assurance*. 4th ed. Boston: Artech House.

Staron, M. (2020) *Action Research in Software Engineering: Theory and Applications*. Cham: Springer International Publishing. Available at: <https://doi.org/10.1007/978-3-030-32610-4>.

Taibi, D., Lenarduzzi, V. and Pahl, C. (2020) ‘Microservices Anti-patterns: A Taxonomy’, in *Microservices*. Cham: Springer International Publishing, pp. 111–128. Available at: https://doi.org/10.1007/978-3-030-31646-4_5.

Tekinerdogan, B. *et al.* (2016) ‘Quality concerns in large-scale and complex software-intensive systems’, in *Software Quality Assurance*. Elsevier, pp. 1–17. Available at: <https://doi.org/10.1016/B978-0-12-802301-3.00001-6>.

Wacker, M. (2015) ‘Just Say No to More End-to-End Tests’, *Google Testing Blog*, 22 April. Available at: <https://testing.googleblog.com/2015/04/just-say-no-to-more-end-to-end-tests.html> (Accessed: 23 October 2023).

Wagner, S. (2008) ‘Defect classification and defect types revisited’, in *Proceedings of the 2008 workshop on Defects in large software systems. ISSTA ’08: International Symposium on Software Testing and Analysis*, Seattle Washington: ACM, pp. 39–40. Available at: <https://doi.org/10.1145/1390817.1390829>.

Yuan, D. *et al.* (2014) ‘Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributed Data-Intensive Systems’, in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. Broomfield, CO: USENIX Association, pp. 249–265. Available at: <https://www.usenix.org/conference/osdi14/technical-sessions/presentation/yuan>.